



Learn by doing: less theory, more results

Adobe Flash 11 Stage3D (Molehill) Game Programming

A step-by-step guide for creating stunning 3D games in
Flash 11 Stage3D (Molehill) using AS3 and AGAL

Beginner's Guide

Christer Kaitila

[PACKT]
PUBLISHING

Adobe Flash 11 Stage3D (Molehill) Game Programming

Beginner's Guide

A step-by-step guide for creating stunning 3D games in
Flash 11 Stage3D (Molehill) using AS3 and AGAL

Christer Kaitila



BIRMINGHAM - MUMBAI

Adobe Flash 11 Stage3D (Molehill) Game Programming

Beginner's Guide

Copyright © 2011 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: November 2011

Production Reference: 1181111

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84969-168-0

www.packtpub.com

Cover Image by Asher Wishkerman (wishkerman@hotmail.com)

Credits

Author

Christer Kaitila

Project Coordinators

Kushal Bhardwaj

Srimoyee Ghoshal

Reviewers

Arthy

Shane Johnson

Proofreader

Kevin McGowan

Acquisition Editor

Tarun Singh

Graphics

Valentina D'souza

Development Editor

Maitreya Bhakal

Production Coordinator

Shantanu Zagade

Technical Editor

Azharuddin Sheikh

Cover Work

Shantanu Zagade

Indexer

Hemangini Bari

About the Author

Christer Kaitila, B.Sc., is a veteran video game developer with 17 years of professional experience. A hardcore gamer, dad, dungeon master, artist, and musician, he never takes himself too seriously and loves what he does for a living: making games! A child of the arcade scene, he programmed his first video game in the Eighties, long before the Internet or hard drives existed.

The first programming language he ever learned was 6809 assembly language, followed by BASIC, Turbo Pascal, VB, C++, Lingo, PHP, Javascript, and finally ActionScript. He grew up as an elite BBS sysop in the MS-DOS era and was an active member of the demo scene in his teens. He put himself through university by providing freelance software programming services for clients. Since then, he has been an active member of the indie game development community and is known by his fellow indies as Breakdance McFunkypants.

Christer frequently joins game jams to keep his skills on the cutting edge of technology, is always happy to help people out with their projects by providing enthusiastic encouragement, and plays an active part helping to find bugs in Adobe products that have not yet been made public. Over the years, he has programmed puzzle games, multiplayer RPGs, action titles, shooters, racing games, chat rooms, persistent online worlds, browser games, and many business applications for clients ranging from 3D displays for industrial devices to simulations made for engineers.

He is the curator of a popular news website called www.videogamecoder.com which syndicates news from hundreds of other game developer blogs. He would love to hear from you on twitter ([www.twitter.com/McFunkypants](https://twitter.com/McFunkypants)) or Google+ (<http://www.mcfunkypants.com/>+) and is always happy to connect with his fellow game developers.

His client work portfolio is available at www.orangeview.net and his personal game development blog is www.mcfunkypants.com where you can read more about the indie game community and his recent projects.

He lives in Victoria, Canada with his beloved wife and the cutest baby son you have ever seen.

This book would not have been possible without valuable contributions of source code, tutorials and blog posts by Thibault Imbert, Ryan Speets, Alejandro Santander, Mikko Haapoja, Evan Miller, Terry Paton, and many other fellow game developers from the ever-supportive Flash community. Thank you for sharing. Your hard work is humbly and respectfully appreciated.

About the Reviewers

Arthy is a French senior flash developer, as well as a big video gaming fan. With 10 years of experience in Flash development for web agencies, his preference is clearly for video games development. Arthy began his video games experience as a junior producer at Atari (Lyon, FRANCE). Since then he went back behind the keyboard and entered the web Flash-based developments in 2001 for a French web agency, Megalos.

Then, to focus on Flash games development, he decided to work on his own as a freelance developer, and so created his company, Le Crabe. After two years of freelancing, he created and joined a freelance association: L'étrange Fabrique.

Recently, he left the freelance world and entered another adventure with a Swiss company: Blue infinity.

Shane Johnson has been working as a developer for the last four years, both freelance and contractually, creating engaging and creative applications for the web, desktop, and also mobile devices with not only Flash, but also any medium that he feels is the right tool to do the job. Primarily an ActionScript developer, Shane also enjoys programming with any language, as it is his real drive to create things using maths and any technology that gets him going.

Since the launch of the Adobe Molehill Incubator program for Flash Player 11, Shane has been involved in experimenting with the new API, creating many examples with some of the new Molehill frameworks that are emerging.

Shane is also an Adobe Certified Expert and maintains a blog at <http://blog.ultravisual.co.uk>, as well as being a consistent contributor to <http://active.tutsplus.com/>.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- ◆ Fully searchable across every book published by Packt
- ◆ Copy and paste, print and bookmark content
- ◆ On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

This book is dedicated to my loving wife, Kristyn, who recently gave birth to our newborn son, James. Every day I am grateful for the joy you bring to my life. You motivate me. Inspire me. Encourage me. Fill me with happiness... and pie! I'm the luckiest man in the world.

Table of Contents

Preface	1
Chapter 1: Let's Make a Game Using Molehill!	7
Your epic adventure awaits!	8
What is Molehill?	8
What Molehill is NOT	9
What you need to know already	9
Basic 3D terminology	10
Common 3D content terms	10
Mesh	11
Polygon	11
Vertex	12
Texture	12
Shaders	13
Vertex program	13
Fragment program	14
3D Content level one achieved!	15
Common 3D coding terms	15
Vectors	16
Normals	17
Matrices	18
3D Coding level one achieved!	19
Summary	22
Level 1 achieved!	22

Table of Contents

Chapter 2: Blueprint of a Molehill	23
The old fashioned way	23
The Molehill way: Stage3D	24
Using 2D Flash text and sprites	25
Why is Stage3D so fast?	25
The structure of a Molehill application	26
Stage	27
Stage3D	27
Context3D	27
VertexBuffer3D	28
IndexBuffer3D	28
Program3D	28
Flowchart of a Molehill program	29
Time for action – things we do once, during the setup	29
Time for action – things we do over and over again	30
Summary	31
Level 2 achieved!	31
Chapter 3: Fire up the Engines!	33
Step 1: Downloading Flash 11 (Molehill) from Adobe	34
Time for action – getting the plugin	34
Time for action – getting the Flash 11 profile for CS5	36
Time for action – upgrading Flex	36
Time for action – upgrading the Flex playerglobal.swc	37
Time for action – using SWF Version 13 when compiling in Flex	38
Time for action – updating your template HTML file	38
Stage3D is now set up!	39
Step 2: Start coding	39
Time for action – creating an empty project	39
Time for action – importing Stage3D-specific classes	41
Time for action – initializing Molehill	42
Time for action – defining some variables	42
Time for action – embedding a texture	43
Time for action – defining the geometry of your 3D mesh	45
Time for action – starting your engines	46
Time for action – adding to the onContext3DCreate function	47
Time for action – uploading our data	48
Time for action – setting up the camera	50
Time for action – let's animate	50

Time for action – setting the render state and drawing the mesh	51
Quest complete – time to reap the rewards	52
Congratulations!	53
The entire source code	53
Summary	56
Level 3 achieved!	56
Chapter 4: Basic Shaders: I can see Something!	57
AGAL: Adobe Graphics Assembly Language	58
What does one line of AGAL look like?	59
What is a register?	59
What is a component?	60
Working with four components at the same time	60
Different registers for different jobs	60
Vertex attribute registers: va0..va7	61
Constant registers: vc0..vc127 and fc0..fc27	61
Temporary registers: vt0..vt7 and ft0..ft7	61
Output registers: op and oc	62
Varying registers: v0..v7	62
Texture samplers: fs0..fs7	62
A basic AGAL shader example	63
The vertex program	64
Time for action – writing your first vertex program	64
The fragment program	65
Time for action – writing your first fragment program	65
Compiling the AGAL source code	66
Time for action – compiling AGAL	66
Time to Render!	67
Time for action – rendering	68
Creating a shader demo	69
Adding an FPS counter	70
Time for action – creating the FPS GUI	70
Time for action – adding the GUI to our inits	72
Time for action – adding multiple shaders to the demo	74
Time for action – initializing the shaders	77
Time for action – animating the shaders	80
Time for action – uploading data to Stage3D	84
Quest complete—time to reap the rewards	84
Congratulations!	85
Summary	87
Level 4 achieved!	87

Table of Contents

Chapter 5: Building a 3D World	89
Creating vertex buffers	90
Importing 3D models into Flash	94
Time for action – coding the Stage3dObjParser class	94
Time for action – creating the class constructor function	96
Time for action – coding the parsing functions	98
Time for action – processing the data	102
Time for action – coding some handy utility functions	105
Our mesh parsing class is complete!	107
The render loop	108
Time for action – starting the render loop	108
Time for action – adding the score to the GUI	113
Time for action – upgrading your init routines	115
Time for action – parsing our mesh data	118
Time for action – animating the scene	118
Quest complete—time to reap the rewards	122
Folder structure	123
Summary	124
Level 5 achieved!	124
Chapter 6: Textures: Making Things Look Pretty	125
Time for a plan: creating a "real" game	126
Using textures in Stage3D	126
Power-of-two	127
UV coordinates	128
Transparent textures	129
Animating UV coordinates in a shader	130
Time for action – updating UV coordinates each frame	130
Texture atlases	131
Animated textures	132
Manipulating texture data	132
Render states	133
Backface culling	133
Time for action – rendering a mesh's backfaces	133
Depth testing	134
Time for action – making a mesh not affect the zbuffer	134
Blend modes	135
Time for action – rendering an opaque mesh	136
Time for action – rendering a mesh with transparent regions	136
Time for action – rendering a mesh so it lightens the scene	137
Increasing your performance	139

Table of Contents

Opaque is faster	140
Avoiding overdraw	140
Avoiding state changes	140
Use simple shaders	141
Draw fewer meshes	141
Adding texture effects to our demo	141
Time for action – embedding the new art	142
Time for action – adding the variables we need	144
Time for action – upgrading the GUI	145
Time for action – listening for key presses	146
Time for action – upgrading our render loop	149
Time for action – upgrading the renderTerrain function	150
Time for action – upgrading our Stage3D inits	150
Time for action – simplifying the initShaders function	152
Time for action – parsing the new meshes	154
Time for action – rendering different meshes as appropriate	154
Time for action – switching textures	156
Time for action – switching blend modes	157
Your demo has been upgraded!	158
Summary	161
Level 6 achieved!	161
Chapter 7: Timers, Inputs, and Entities: Gameplay Goodness!	163
Our current quest	164
Keeping it simple	164
Making it reusable	165
Making our game more interactive	166
Adding a HUD overlay graphic	166
Time for action – adding a GUI overlay	167
Keeping track of time: a game timer class	168
Time for action – creating the game timer class	169
Time for action – adding the GameTimer class constructor	170
Time for action – implementing the tick function	170
A game input class	172
Time for action – creating the GameInput class	173
Time for action – coding the GameInput class constructor	174
Time for action – detecting mouse movement	174
Time for action – detecting the keyboard input	176
Time for action – detecting key release events	177
Time for action – detecting the input focus	178
An abstract entity class	179

Table of Contents

Time for action – creating the Stage3dEntity class	180
Time for action – creating the Stage3dEntity class constructor	182
Hiding complex code by using get and set functions	183
Time for action – getting and setting the transform	183
Time for action – getting and setting the entity position	184
Time for action – getting and setting the entity rotation	186
Time for action – getting and setting the entity's scale	187
Time for action – updating the transform or values on demand	189
Time for action – creating the movement utility functions	190
Time for action – implementing vector utility functions	192
Time for action – adding some handy entity utility functions	194
Time for action – cloning an entity	197
Time for action – rendering an entity	198
Design art for our new improved game world	200
Upgrading our game	201
Time for action – importing our new classes	202
Time for action – adding new variables to our game	203
Time for action – embedding the new art	205
Time for action – upgrading the game init	207
Time for action – upgrading the GUI	208
Time for action – simplifying the shaders	209
Time for action – using the new textures	210
Time for action – spawning some game entities	212
Time for action – upgrading the render function	215
Time for action – creating a simulation step function	216
Time for action – creating a heartbeat function	217
Time for action – upgrading the enterFrame function	218
Let's see all this in action!	219
Summary	221
Level 7 achieved!	221
Chapter 8: Eye-Candy Aplenty!	223
Our current quest	223
Designing for performance	224
Designing for reusability	225
Animating using AGAL	225
A basic particle entity class	226
Time for action – extending the entity class for particles	227
Time for action – adding particle properties	228
Time for action – coding the particle class constructor	228
Time for action – cloning particles	230

Time for action – generating numbers used for animation	230
Time for action – simulating the particles	231
Time for action – respawning particles	232
Time for action – rendering particles	233
Keyframed vertex animation shader	236
Time for action – creating a keyframed particle vertex program	236
Time for action – creating a static particle vertex program	238
Time for action – creating a particle fragment program	238
Time for action – compiling the particle shader	239
A particle system manager class	240
Time for action – coding a particle system manager class	241
Time for action – defining a type of particle	242
Time for action – simulating all particles at once	242
Time for action – rendering all particles at once	243
Time for action – spawning particles on demand	244
Time for action – creating new particles if needed	244
Keyframed particle meshes	246
Selecting a particle texture	246
Time for action – sculpting a single particle	246
Time for action – sculpting a group of particles	247
Time for action – sculpting the second keyframe	248
Incorporating the particle system class in our game	249
Time for action – adding particles to your game	249
Time for action – preparing a type of particle for use	250
Time for action – upgrading the renderScene function	251
Time for action – adding particles to the gameStep function	252
Time for action – keeping track of particle statistics	253
Let's see the new particle system in action!	254
Summary	257
Level 8 achieved!	257
Chapter 9: A World Filled with Action	259
Extending the entity class for "actors"	260
Time for action – creating a game actor class	260
Time for action – extending the actor's properties	261
Time for action – coding the GameActor class constructor	262
Time for action – creating a step animation function	263
Time for action – animating actors	264
Implementing artificial intelligence (AI)	265
Time for action – using timers	265
Time for action – shooting at enemies	266

Table of Contents

Time for action – cloning an actor	268
Time for action – handling an actor's death	270
Time for action – respawning an actor	270
Collision detection	271
Time for action – detecting collisions	271
Time for action – detecting sphere-to-sphere collisions	272
Time for action – detecting bounding-box collisions	272
An "actor reuse pool" system	274
Time for action – creating an actor pool	274
Time for action – defining a clone parent	275
Time for action – animating the entire actor pool	276
Time for action – rendering an actor pool	277
Time for action – spawning an actor	278
Time for action – checking for collisions between actors	279
Restricting display to nearby actors for better framerate	280
Time for action – hiding actors that are far away	280
Time for action – destroying every actor in the pool	282
Easy world creation using a map image	283
Time for action – implementing a level parser class	284
Time for action – spawning actors based on a map image	284
Time for action – parsing the map image pixels	286
Upgrading the input routines	288
Time for action – adding more properties to the input class	289
Time for action – handling click events	290
Time for action – upgrading the key events	291
Summary	297
Level 9 achieved!	297
Chapter 10: 3... 2... 1... ACTION!	299
Our final quest	300
Getting to the finish line	301
Time for action – drawing a title screen	302
Time for action – importing required classes	303
Adding new variables to our game	304
Time for action – tracking the game state	304
Time for action – adding variables for timer-based events	305
Time for action – adding movement related variables	305
Time for action – keeping track of all entities	306
Time for action – upgrading the HUD	307
Time for action – defining variables used by Stage3D	308
Adding art to our game	308

Time for action – embedding our new art assets (AS3 version)	308
Time for action – embedding our new art assets (CS5 version)	311
Time for action – embedding all the meshes	312
Time for action – keeping track of art assets	316
Upgrading the final game source code	317
Time for action – upgrading the inits	317
Time for action – initializing Stage3D	319
Time for action – upgrading the initGUI function	321
Time for action – upgrading the texture inits	324
Time for action – upgrading the shaders	326
Time for action – defining new actor types and behaviors	329
Time for action – initializing the terrain meshes	331
Time for action – initializing the enemies	333
Time for action – initializing the bullets	337
Time for action – initializing the asteroids	338
Time for action – initializing the space stations	339
Time for action – initializing the particle models	340
Time for action – creating the game level	341
Time for action – upgrading the render loop	343
Defining gameplay-specific events	345
Time for action – tracking game events	345
Time for action – handling game over	348
Time for action – updating the score display	350
Time for action – updating the FPS display	351
Time for action – handling collision events	352
Time for action – handling the player input	354
Time for action – upgrading the gameStep function	358
Time for action – upgrading the heartbeat function	361
Time for action – upgrading the enterFrame function	362
Publish... distribute... profit!	363
Summary	365
Level 10 achieved. Universe saved!	365
Where to go from here?	365
A note from the author	366
Appendix A: AGAL Operand Reference	367
What does one line of AGAL look like?	367
Registers available for AGAL programs	368
COPYING DATA	369
ALGEBRAIC OPERANDS	369
MATH OPERANDS	369

Table of Contents

TRIGONOMETRY OPERANDS	370
CONDITIONAL OPERANDS	370
VECTOR and MATRIX OPERANDS	371
TEXTURE SAMPLING OPERAND	372
Appendix B: Pop Quiz Answers	373
Index	377

Preface

Adobe's Stage3D (previously codenamed Molehill) is a set of 3D APIs that has brought 3D to the Flash platform. Being a completely new technology, there were almost no resources to get you acquainted with this revolutionary platform, until now.

This book will show you how to make your very own next-gen 3D games in Flash. If you have ever dreamed of writing your own console-style 3D game in Flash, get ready to be blown away by the hardware accelerated power of Stage3D. This book will lead you step-by-step through the process of programming a 3D game in ActionScript 3 using this exciting new technology. Filled with examples, pictures, and source code, this is a practical and fun-to-read guide that will benefit both 3D programming beginners and expert game developers alike.

Starting with simple tasks such as setting up Flash to render a simple 3D shape, each chapter presents a deeper and more complete video game as an example project. From a simple tech demo, your game will grow to become a finished product—your very own playable 3D game filled with animation, special effects, sounds, and tons of action. The goal of this book is to teach you how to program a complete game in Stage3D that has a beginning, middle, and game over.

As you progress further into your epic quest, you will learn all sorts of useful tricks such as ways to create eye-catching special effects using textures, special blend modes for transparent particle systems, fantastic vertex and fragment programs that are used to design beautiful shaders, and much more. You will learn how to upload the geometry of your 3D models to video RAM for ultra-fast rendering. You will dive into the magical art of AGAL shader programming. You will learn optimization tricks to achieve blazingly fast frame rate even at full screen resolutions. With each chapter, you will "level up" your game programming skills, earning the title of Molehill Master—you will be able to honestly call yourself a 3D game programmer.

This book is written for beginners by a veteran game developer. It will become your trusty companion filled with the knowledge you need to make your very own 3D games in Flash.

What this book covers

Chapter 1, Let's Make a Game Using Molehill! In this chapter, we talk about what Stage3D (Molehill) is, what it can do, and the basic terminology you need to know when dealing with 3D games.

Chapter 2, Blueprint of a Molehill. In this chapter, we compare the differences between old-fashioned Flash games and the new way of doing things, along with a description of the major classes we will be working with and the structure of a typical Stage3D game.

Chapter 3, Fire up the Engines. In this chapter, we take the first step by setting up our tools and programming the initializations for our game. The result is a demo that gets Stage3D to animate a simple 3D mesh.

Chapter 4, Basic Shaders: I can see Something! In this chapter, we learn about programming shaders using AGAL and adding text to the display. The result is an upgraded demo with four different animated shaders.

Chapter 5, Building a 3D World. In this chapter, we create a way to fill the game world with complex 3D models by programming a mesh data file parser. The result is a game demo complete with high-poly spaceships and terrain instead of simple textured squares.

Chapter 6, Textures: Making Things Look Pretty. In this chapter, we upgrade our game demo to include a keyboard input and special render modes that allow us to draw special effects such as transparent meshes, explosions, and more. The result is a demo that highlights these many new effects.

Chapter 7, Timers, Inputs, and Entities: Gameplay Goodness! In this chapter, we program a timer and generic game entity class. In addition, we upgrade the GUI with a heads-up-display overlay and add a chase camera. The result is a demo with a spaceship that can fly around in an asteroid field that looks more like a real video game.

Chapter 8, Eye-Candy Aplenty! In this chapter, we program a highly optimized GPU particle system for use in special effects. All geometry is rendered in large, reusable batches. The result is a game demo that is able to render hundreds of thousands of particles at 60fps.

Chapter 9, A World Filled with Action. In this chapter, we upgrade our game engine to include simple game actor artificial intelligence, collision detection, and a map parsing mechanism to allow for easy world creation. The result is a fully functional engine ready for use in a real game.

Chapter 10, 3... 2... 1... ACTION! In this chapter, we add the final touches to our game project such as a title screen, dealing with the score, damage and game over events, music and sound, and much more. The final result of our efforts is a fully playable 3D shooter game filled with action!

Appendix A, AGAL Operand Reference. This appendix provides operand references that have been used in this book.

Appendix B, Pop Quiz Answers. In this section, we provide answers to the pop quiz.

What you need for this book

Recommended: FlashDevelop 4 (available for free from <http://www.flashdevelop.org>) or Adobe Flash CS5 (or newer).

The code in the book should also compile under any other AS3 development environment such as FlashBuilder, FDT, or Flex with very few changes required.

Optional tools:

- ◆ Image editing software (Photoshop, GIMP, and so on)
- ◆ Mesh editing software (3ds Max, Maya, Blender, and so on)

Who this book is for

If you ever wanted to make your own 3D game in Flash, then this book is for you. This book is a perfect introduction to 3D game programming in Adobe Molehill for complete beginners. You do not need to know anything about Stage3D/Molehill or Flash 11 in order to take advantage of the examples in this book. This book assumes that you have programming experience in AS3 (ActionScript 3).

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Last but not the least; now that everything is set up, an event listener is created that runs the `enterFrame` function every single frame."

A block of code is set as follows:

```
package
{
    [SWF(width="640", height="480", frameRate="60",
        backgroundColor="#FFFFFF")]

    public class Stage3dGame extends Sprite
    {

    }

}
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Alternately, if you are using FlashDevelop, you need to instruct it to use this new version of flex by going into **Tools | Program Settings | AS3 Context | Flex SDK Location** and browsing to your new Flex installation folder."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com or e-mail suggest@packtpub.com.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code for this book

You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Let's Make a Game Using Molehill!

Welcome, brave hero!

You are about to undertake an epic quest: to make a game using the Stage3D API (Molehill). This quest will be filled with fun and excitement, as well as challenges and setbacks. It is one that you are sure to enjoy. Programming a 3D game is not easy—but you CAN do it!

Consider this book your trusty sidekick on an exciting adventure. Much like in a role-playing game, you are the star. You are the hero.

This book will take the tone of a grand adventure story. We will explore Molehill together, overcoming challenges and slowly leveling-up our knowledge. On each step along the way to our goal—the creation of a 3D game—you will earn "experience points". During your adventure in game making, you will be presented with systematic challenges that we will call "quests". At the end of each chapter, you will "level-up".

This book is aimed at complete beginners to Molehill, and is meant to be informal, semi-jovial, light-hearted, and fun to read. You don't just want a boring appendix of every Stage3D command listed in order. That information can be found using Google.

Instead, we will start from scratch and gradually explore deeper into the world of 3D Flash games, one skill at a time. It will be example-heavy and filled with information that you can use in real-life game development using the Flash 11 Stage3D API.

Your epic adventure awaits!

Your mission is simple. Armed with this book and a little hard work, your quest is to program a fully functional 3D game filled with action and movement, loaded with eye-candy and amazing 3D graphics. When you have completed all the exercises in this book, you will be the proud author of your very own 3D game, ready to be played by all your friends... and indeed the whole world.

Your adventures will become the stuff of legend, to be sung by travelling minstrels in all the taverns in the kingdom.

So let's get on with it! Time for the training camp; the best place to start is the very beginning. The basics:

What is Molehill?

Molehill is the code name for Adobe's latest Flash technology, "Stage3D". It is specifically designed to take advantage of the modern hardware-accelerated 3D graphics cards. Just like games seen on consoles such as XBOX 360 and the Playstation 3, hardware 3D graphics processors deliver some serious graphics horsepower.

Compare this to how Flash traditionally renders graphics on-screen: vector graphics are rendered in software into 2D bitmaps and are blitted (drawn) onto the screen as pixels. This is both computationally expensive and extremely slow. If you have ever experimented with drawing a few hundred sprites in a large Flash window, you know that Flash starts to slow down to the point of being unplayable very quickly.

Molehill, on the other hand, moves this entire pixel processing function onto the video card, freeing your computer's CPU to do other things while the 3D graphics are rendered by a specialized **GPU (graphics processing unit)** on your video card. This allows the next generation of Flash games to render hundreds of thousands of sprites on-screen—at full screen resolution—with minimal CPU load.

In previous versions of Flash (10 and below), all the work—including all drawing—was done on the CPU. Moving all those pixels in each and every frame takes billions of tiny calculations. In the new version of Flash, you can have a video card do the time-consuming and calculation-intensive tasks of rendering your scenes, freeing up the CPU to do other things. For example, your CPU can be calculating enemy movements, updating the high score, or downloading files at the same time as your GPU is doing all the drawing.

Anecdotal benchmarks have shown full 3D scenes comprised of over 150,000 polygons running full screen at HD resolution at 60 frames per second (FPS) with less than 10% CPU usage. Compare this to a typical old-fashioned Flash animation which can max out the CPU to 100% usage with only a few thousand sprites in a small window.

The Stage3D API will free artists and game designers to spend more time thinking about the content and less time worrying about performance. You will be able to draw hundreds of explosions or smoke particles or special effects within a massive and complex 3D world filled with moving vehicles, buildings and terrain, and still maintain phenomenal frame-rate.

Why is it so fast? Because Molehill makes your video card do all the work. Flash gets to sit back and take it easy.

What Molehill is NOT

Molehill is a very low-level API. It only provides basic 3D rendering functionality. It is insanely fast, and the brilliant minds at Adobe have made the smart decision to keep it very simple. It does just one thing and it does it well. It is not bloated. It is not a jack-of-all-trades, with a million features plus the kitchen sink. It is a low-level programming interface that deals exclusively with 3D rendering and nothing else.

It is not a game engine (like Unity3d or Unreal or ID Tech). A game engine also includes supplemental game functions such as physics simulation, sounds, music, collision detection, and much more. Instead, you use a low level API such as Molehill to create your own game engine from scratch.

Instead of trying to be everything to everybody, Molehill sticks to being really good at just one thing—3D graphics. Much like OpenGL or Direct3D, Molehill is a collection of very basic functions. They are the basic building blocks of games and game engines.

We will be doing all Molehill programming using a new set of AS3 commands. The low level programming (for shaders) is written in its own language, which is compiled at runtime and uploaded to the video card. Molehill shaders look shockingly similar to those of the most low level of programming languages, assembly language.

Don't let that scare you: the number of commands you will need to learn is small.

What you need to know already

As long as you are armed with a basic understanding of Flash ActionScript, you are ready to embark on this adventure without any worries. You do not need to know a single thing about Molehill. Beginners are welcome.

As you know, AS3 (shorthand for Flash's ActionScript 3) is a programming language that looks very similar to Javascript or C++. You should already be familiar with Flash programming, but you do not yet have to be a 3D expert.

You don't need to be an elite coder, neither do you have to be a mathematician to get amazing results. We are not going to explain what a function or a variable is, nor are we going to show you how to compile your project into a .swf file—you already know how to program in Flash, you just want to learn Stage3D.

If you are not comfortable with AS3 programming quite yet, no problem! There are many great books available to help you with AS3. Get yourself to the level that you can program a simple "hello world" Flash program and then come back. Basic programming skills are beyond the scope of this book, which is only about the Molehill side of Flash.

Assuming that you already know a little bit about AS3, you know everything required to jump right in.

Basic 3D terminology

Your basic training, to get you to "level one" so to speak, should include the knowledge of some prerequisite terminology that you are sure to encounter during your adventures in 3D game development. Most of them are generic words for concepts related specifically to 3D.

You only need to know a few new terms in order to completely understand the 3D coding. It is not as complex as some people would have you think. You probably already know what most of these terms mean, so let's get them out of the way.

Are you ready to use both your left and right brain? We only need to go over two groups of terms: 3D art and 3D code.

Your artistic side, the right side of your brain, loves art. It cannot wait to sense the colors and textures that make up the visual component of your game.

Your logical side, the left side of your brain, hungers for all that is algorithmic. It wants to dive into some tasty code and make things happen. Working together, the art and code are used to create videogames. Here are the basic definitions that you will need in order to forge ahead and dive into Molehill.

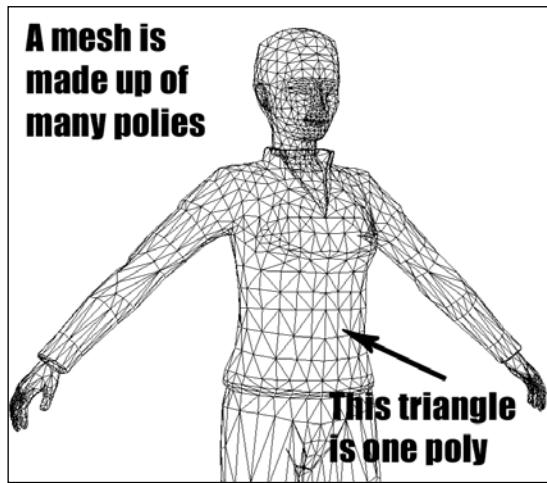
Common 3D content terms

This section is a perfect food for the right side of your brain. It concerns the art.

In 3D games, what you see is simulated in a virtual world that has depth and volume. Instead of a 2D game where you are only concerned with the coordinates on the screen of various images, in a 3D game you can "look around the corner" so to speak; objects are sculpted, just like a real-world building, in three dimensions. In order to do this, the following art asset types are most commonly used.

Mesh

A model in a 3D game is typically called a mesh. This mesh is built in a 3D modeling application such as Maya, 3D Studio Max, or Blender. It defines the 3D shape of an object, as well as the colors used to draw it. Your game's main character will be made up of one (or more) meshes. A vehicle, a building, or a tree is all individual meshes. A single scene can be built from hundreds of meshes.



Polygon

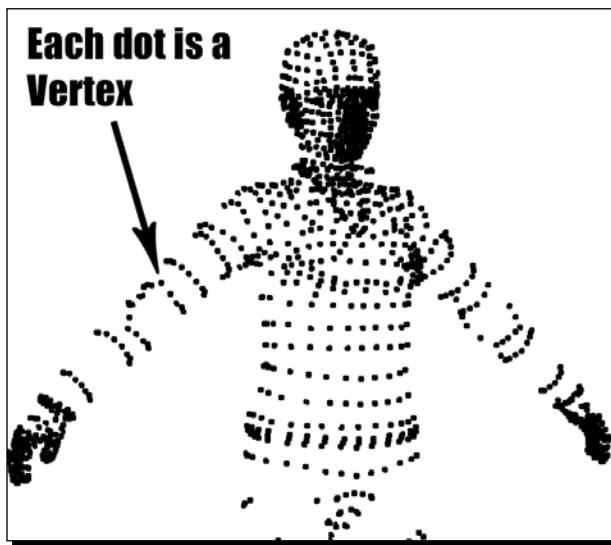
Each facet of a mesh is a flat geometrical shape. One mesh could be made from hundreds or thousands of polygons, or "polies". One poly could be a simple triangle with three sides. It could be a "quad" or a four-sided square or rectangle. Imagine, for example, a 3D cube, a box. It is made from six different squares (quads), one for each side. Each of these quads is called a poly.

Any kind of a polygon can be built from a collection of triangles. For example, a quad can be made from two triangles sitting side by side. As a result, modern graphics cards treat all 3D geometry as a huge collection of triangles. Usually, when somebody tells you the number of polies a particular mesh is made from, they are referring to the number of triangles (or "tris") used to define it. When a video card manufacturer is boasting about their latest and greatest benchmark performance, they will quote how many polies they are rendering and how many times per second they can do so.

Vertex

Each poly is defined by the location of three or more "corners". For example, in a square quad, there are four corners. Each of these four corners is called a vertex. A vertex (the point in space where the corner sits) is defined as a simple location in the 3D world.

In the example of a 3D box that has six sides (each being a square quad as described earlier), there are eight corners. Each of these corners is a vertex.



As meshes are made from hundreds of polies, and each poly is defined by three or more vertex coordinates, when you tell Molehill about the "shape" of a particular model in your game, you generally do so by listing thousands of vertex coordinates in a huge array.

Texture

In order to draw a mesh (unless you simply want to render a wireframe), you will use one or more images, which are called textures, to color the shape. A texture could be thought of as wallpaper: plaster the texture onto the mesh, wrapping it around the mesh, and this image is stretched to conform to the mesh shape. For example, imagine a box. It would be a mesh with six sides, defined by a total of eight vertex coordinates (one for each corner). In order to make the box look like it was made out of rusty metal, you would draw, using Photoshop or GIMP, an image, and save it as a .jpg or a .png file. This image is your texture. By instructing Molehill to wrap it around your box mesh, the game would render it as a rusty metallic cube.

Shaders

Shaders are the definition of a particular visual style. They define "how to draw" something. One shader might be used to render shiny metal while another could be used to draw explosions.

In the Stage3D API, shaders are stored in the `Program3D` class. In order to create one, you need to create both a "fragment program" and a "vertex program". The combination of these two programs is a shader.

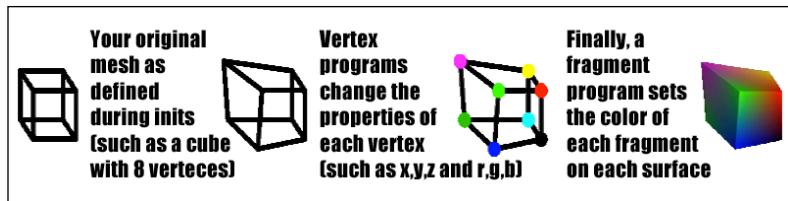
When people ask you if you know how to write shaders, you will soon be able to answer, yes, I am familiar with writing vertex and fragment programs.

Vertex program

A vertex program is run once for every vertex in your mesh, every time you want to render it. It is given whatever data is stored in your vertex array for that particular vertex plus whatever variables you wish to tell it about such as the camera angle. When it is done, it reports the "final position" for that vertex just prior to rendering, plus (optionally) other information, such as the vertex color.

Vertex programs tell your fragment program about the current state of each vertex in your mesh.

A vertex program decides where each vertex is, what color it is, what its texture coordinates are and anything else you may wish to calculate prior to being sent to your fragment program for rendering.



Vertex programs allow you to interact with your vertex data, as well as change things around. In the code, you send tiny bits of data to a vertex program to change the shape or position of the vertices in a mesh every frame. For example, if you program a vertex program to control the movement of an avatar's head, in each frame you could send a number from 0 to 360 to control which way the head is facing.

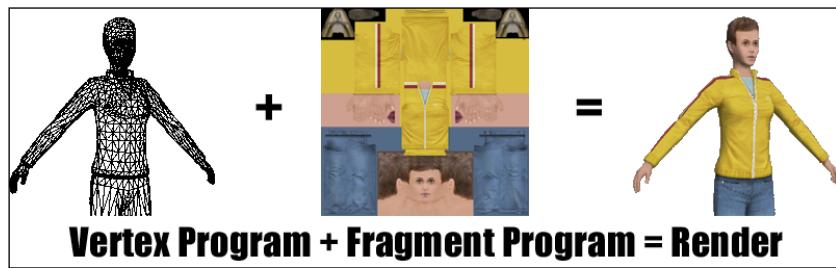
Vertex programs can use these constantly updated real-time parameters in an infinite variety of ways. Instead of a single number, as in the preceding example, perhaps you will create vertex programs that use vector3Ds to change the position of a sentry gun's turret, or an entire array of numbers to completely change the overall shape of your mesh by changing the locations of each vertex.

Every mesh you render will require a vertex program, whether very simple (draw each vertex in their original position) or complex (morph the mesh to animate a walk cycle).

Fragment program

In Molehill, a fragment program is a set of commands that define how to handle the visual rendering of a particular mesh. Sometimes referred to as pixel shaders, fragment programs are all about describing how a given surface/texture responds to light. Different materials respond to light in different ways (depending on how reflective and transparent they are for one thing), and you need different code to simulate that.

Fragment programs often use textures (the bitmaps described earlier), as well as complex functions to decide what color each pixel on the screen should be. Sometimes a fragment program will combine the effects of more than one texture, as well as the RGBA (red, green, blue, alpha transparency) data stored alongside each vertex location, plus any number of parameters to arrive at the final color.



By using fragment programs, you can add special effects to your plain textures, such as adding shine or shadows or real-time animation. Fragment programs let you fade things in, tint them different colors, and tell your video hardware how to draw each poly that makes up a mesh. Every mesh you draw needs a fragment program or nothing is drawn. It could be simple (just use the texture as is) or complex (blending two textures together and tinting it a see-through transparent shiny red that is affected by some lights).

3D Content level one achieved!

With regard to common 3D art asset terms, that's it for now! All you need to remember is that for each object you want to draw on the screen, you will upload some data to your video card.

In our rusty metal box example, you will instruct Molehill of each vertex that defines the shape of the mesh. You will define a texture, the image of rusty metal. You will compile a fragment program, instructing Molehill how to draw that texture, as well as a vertex program, which instructs Molehill what to do with the vertex data.

When the game is running, you might decide to fade the box out by sending a few numbers to your fragment program. Perhaps, you will change the box from rusty metal to wood by replacing the texture it uses. Alternately, you might want to make the box wobble as if it is made out of jelly by sending a few numbers to your vertex program.

You are in control of everything you need to produce spectacular 3D graphics. All this graphics power is sure to give you ideas. Anything you see in today's next-gen games can be accomplished with Molehill's fragment and vertex programs. The possibilities are endless.

Common 3D coding terms

This section is targeted towards the left side of your brain. It concerns the functional side of game programming.

A videogame is like a potion. You take the raw ingredients (the art content) and add a little magic. A sprinkling of 3D code is all it takes to make the game magic happen.

Don't be afraid of a little code, as a programmer you know that this is how things are done. By using the following concepts, you will have the tools to make things move around for you. Luckily, you don't need to be a Math whiz to make a game. Even if you are primarily an artist, you can program a game using Molehill by learning from examples.

Instead of a verbose list of every possible 3D function in Flash, instead of a giant dusty dry tome filled with trigonometry and advanced calculus, only the top four terms are presented here. If you do a Google search for Flash AS3 documentation regarding 3D, then you will find many more.

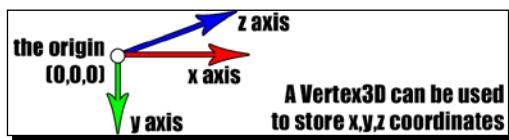
The great news is that 90% of the time you will only have to concern yourself with the following data structures: vectors, normals, and matrices.

Vectors

A Vector3D is an object that contains x, y, and z values that describe a location in the 3D space.

Each component is a number (which can be positive or negative and can be a fraction), and each number is the distance from 0,0,0 (the origin). Just as in 2D Flash, where Point(0,0) usually defines the top-left corner of the stage, a new Vector3D(0,0,0) would represent the location of the "center of the universe" or the very middle of the 3D world. This location (0,0,0) is called the origin. The term used to describe the x or y or z part of a vector is called an axis.

For example, you could use a Vector3D to define the coordinates of each vertex in a mesh. You could also use a Vector3D to define the location of a bullet in your game.



For example, the x-axis (the first number in a Vector3D) typically represents the left-right movement. The y-axis is most commonly used for the up-down movement, and the z-axis describes the in-out movement (that is, moving from close to the camera to far away from the camera). In a normal 3D scene, when the z coordinate increases the object, it is seen to be smaller because it is farther away from the screen. Vectors actually have a fourth component, w, but it is rarely used. It is handy to store a rotation. In the vast majority of cases you will consider a Vector3D to consist of x,y,z.

Imagine that you want to put a model of a tree in a particular spot in a 3D scene. In old-fashioned 2D flash games, you have most likely used the Point class, which has an x and y coordinate, to define where on screen a sprite is supposed to be. In Molehill, the class you need to work with is named Vector3D.

Here are some examples of how to use vectors in Molehill:

```
import flash.geom.Vector3D;  
  
var putTheTreeHere:Vector3D = new Vector3D(0,0,0);  
var locationOfTreasure:Vector3D = new Vector3D(1000,0,-15000);  
  
var aNormalPointingUp:Vector3D = new Vector3D(0,1,0);
```



Downloading the example code for this book

You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

You can probably guess what each line of the preceding code would do. We have declared three variables. Each is a vector, and could be used to define where something is sitting in the world. The `Vector3D` class is defined in the built-in package named `flash.geom`.

Looking back at the preceding example source code, if you were to add the following at the end:

```
trace(locationOfTreasure.z);
```

The output in the debug log after running the code would be `-15000`, the number stored in the `z` component of the variable named `locationOfTreasure`.

Normals

There is a special kind of vector which is usually used to describe a direction, instead of a location. This is called a normal. You store it using the exact same class, a `Vector3D`. The only difference between a positional vector, which could look like `[55,5000,12]` and a directional vector, a normal, is that normals always have a length of one .

Normals are also often called "unit" vectors because they have a length of one unit. The term "normal" literally means perpendicular (for example, normal to a plane, or pointing straight away from a particular surface). For example, a normal vector that describes something as pointing straight up would be `[0,1,0]` (assuming the `y` coordinate is up in this world, this vector describes a direction that is perpendicular to the "ground").

In the preceding example source code, the variable named `aNormalPointingUp` is simply a `Vector3D` that has a length of one.

The reason why normals are so handy is that, because they have a length of one, you can use them to multiply with times or distances or speeds. For example, if you multiply a normal that points straight up with a speed, such as 100, then the resulting locational vector would be 100 units larger in the `y` axis.

You can add vectors together, subtract them, multiply, and divide them. In each case, all that is happening is that each axis in the vector (the `x,y,z`) is added or subtracted, or multiplied by the corresponding axis in the vector it is being added to.

How would you do this? Well, by using `aNormalPointingUp` from the example, we could add it to `locationOfTreasure` like this:

```
locationOfTreasure =  
    locationOfTreasure.add(aNormalPointingUp);  
trace(locationOfTreasure.y);
```

Now the output in the debug log would be 1.

This would result in the original position of the treasure being incremented by one in the y-axis. In simple English, the treasure would now be floating one unit above the ground.

In this example, then, the original value for `locationOfTreasure` was (1000,0,-15000) and after adding `aNormalPointingUp` to it, the new value is (1000,1,-15000). Makes sense, doesn't it?

The variable `aNormalPointingUp` is a special kind of vector. As it has a length of one, it is called a normalized vector, or just "a normal". Why make the distinction between a vector and a normal? A normal is just a special kind of vector. As it has a length of one, you can use it in handy ways, such as multiplying it by the amount of time which has passed since the last frame, so that a missile flies at a certain speed.

You can turn a positional vector, such as [55,5000,12], into a normal by dividing each axis by the length of the vector, which forces the total length of the new vector to be 1. The easiest way to do this is to use the `normalize()` function that is built into the `Vector3D` class.

Why would you want to do this? You might want to know what direction "points" toward a particular location. In this example, the normal of [55,5000,12] is approximately [0.011, 0.987, 0.002]. You can see that the proportions for each axis are the same, now they just add up to one.

Normals are really handy in game programming, and you will see them mentioned all the time. Remember that a normal is just a vector that is only one unit long.

Matrices

Sometimes you need to move and rotate and scale an object all in one go. In order to make your life easier, another AS3 class can be used. A `Matrix3D` is simply a collection of vectors. Each vector inside it describes the position, rotation, and scale. Often, you will need to move something at the same time as rotating it, perhaps also increasing its size. The `Matrix3D` class uses a 4x4 square matrix: a table of four rows and columns of numbers that hold the data for the transformation.

You don't need to know how a matrix is organized. Just know that it is really useful as a way to gather all the movement, rotation, and scaling data for a 3D object, so you can use it in a single line of code. There are many handy functions built into the `Matrix3D` class that can perform the difficult math for you, such as changing the scale or adding rotations or translations (positional changes).

For example, imagine a puff of smoke used in a cool-looking particle system like an explosion. In each frame, that puff will need to rise into the air, spin around, and perhaps grow larger as the smoke dissipates. With a matrix, you can do all this in one line of code. This is usually called a transform, which is just another word for a matrix that is used to describe the movement of an object in space. You usually create a matrix and use it to transform (move, scale, rotate, and so on) an object. Sometimes in 3D programming, you will see the term "matrix transforms" and this is what it means. For example, to transform a vector you might multiply it to be a matrix.

3D Coding level one achieved!

That is it! These are all the coding terms you need to know for now. All this talk of 3D matrix Math and Cartesian-coordinate geometry can be a little bit scary. Fear not, gallant hero.

Like the terrifying stomp of an approaching boss battle, all you need to do is take a deep breath and try your best to forge ahead. Nobody is going to expect you to know how to calculate the dot product of two vectors by hand and if you didn't take calculus in school, you are not at all precluded from becoming the next game programmer hero.

Be brave, wise adventurer, and take pride in the fact that you have been awarded the very first achievement in your quest to make a 3D game.

Pop quiz – earn some experience points

Here are a few quick questions to test your understanding. If you can answer them, you are officially ready to move on to the next step in your grand adventure.

1. Imagine a 3D model of a house. The x, y, and z location in the 3D space of the house in your game world would be defined by a:
 - a. 4x4 Matrix
 - b. Texture
 - c. Vector3D
 - d. Mole wearing a top hat
2. If you wanted to make each wall of your house mesh look like it was made from purple plaid wallpaper, you would draw the plaid in an art program, save it as a jpeg, and use it in a fragment program as a:
 - a. Normal
 - b. Texture
 - c. Polygon
 - d. Secret ingredient

3. Imagine a giant ogre decides to push your house over to the next lot on the street. It shifts in position to the side by a small amount. Let's say its original position was [10,0,10]. The ogre pushed it by this amount: [5,0,0]. Where is it now?
 - a. [15,0,10]
 - b. [15,0,0]
 - c. [5,0,10]
 - d. 42

Have a go hero – your first side quest

In order to really hone your skills, it can be a good idea to challenge yourself for some extra experience. At the end of each chapter, there is a side quest—without a solution provided—for you to experiment with, and it is optional. Just like grinding in an RPG game, challenges like these are designed to make you stronger, so that when you forge ahead, you are more than ready for the next step in your main quest.

Your side quest is to compile the following AS3 source code, so that you have a test bed for further experiments.

If you love to use the Flash IDE, that is great. Just set up a blank project and drop this code wherever you deem to be appropriate. If you use FDT or Flex, no problem, you know how to make a "hello world" project.

Many game developers nowadays use FlashDevelop as an IDE that uses Flex to compile pure AS3 projects without any of the CS5 bloat like timelines. FlashDevelop is completely free and is highly recommended as a more responsive, faster way to make flash files. You can download it from <http://www.flashdevelop.org/>. Whatever your weapon of choice, try to get yourself to the point where you have a simple project to do some experiments in.

Simply fire up your IDE or code editor of choice and see if you can get this quick "hello world" project compiled. It does not draw anything on screen: it simply outputs a couple of lines of text to the debug trace window:

```
package  
{  
    import flash.geom.Vector3D;
```

```
private var putTheTreeHere:Vector3D =  
    new Vector3D(50,0,5000);  
  
private var moveItByThisMuch:Vector3D =  
    new Vector3D(0,0,0);  
  
function moveTheTree():void  
{  
    trace('The tree started here: ' + putTheTreeHere);  
  
    putTheTreeHere =  
        putTheTreeHere.add(moveItByThisMuch);  
  
    trace('The tree is now here: ' + putTheTreeHere);  
}  
  
}
```

Now that you have set up a project, you may have to change the preceding code to compile, depending on whether you are using pure AS3, flex, or flash. It is up to you to set up a basic project in whatever way you are most comfortable with.

Once you have the preceding code compiling properly, your side quest challenge (this should be the easy part) is to change the variable `moveItByThisMuch`, so that the tree's final position is [100,0,5555].

For the adventurous, as an extra challenge try playing around with vectors some more. For example, you can move the tree by a small amount every single frame, as part of an `onFrame()` event. Can you figure out a way to move the tree at a constant rate regardless of the frame rate? Hint: you might find out how many ms have passed since the previous frame using the `getTimer()` function and then multiplying a tiny vector by that amount before adding that total to the tree's position. Use Google if you need it. You can do it!

Summary

We learned a lot in this chapter about common terms used in 3D programming. The good news is that you now know just about everything required to program video games. Any time you need to move something around, rotate it, change its scale, or calculate how far something travelled during a particular `TimeSpan`, you will be working with vectors. Months from now, when you are deep in the middle of creating a fantastic gaming masterpiece, you will most likely be spending most of your time dealing with vectors, normals, and matrices. When you are working on the art, you will be thinking about polygons and textures.

Specifically, we covered:

- ◆ **Vector3D**: containing an x, y, and z component
- ◆ **Normal**: a `Vector3D` that has a length of one
- ◆ **Matrix**: a 4x4 group of vectors with position, rotation, and scale
- ◆ **Vertex**: a point in space that is the corner of a polygon
- ◆ **Polygon**: a shape defined by multiple vertex coordinates
- ◆ **Mesh**: a group of polygons (polies) that make up a model
- ◆ **Texture**: a bitmap image that is like wallpaper for a mesh
- ◆ **Shader**: the combined result of a vertex program and a fragment program
- ◆ **Vertex program**: commands affecting the shape of a mesh
- ◆ **Fragment program**: commands affecting the look of a mesh

Now that we have learned the basics of 3D programming, we are ready to dive into Molehill. We are now armed with sufficient knowledge to have "leveled up".

Level 1 achieved!

Congratulations! As a level one Molehill master, you have opened the door that leads to the inner secrets of the Stage3D API—which is the topic of our next chapter.

2

Blueprint of a Molehill

Level 2, here we come!

Now that you have learned the basic terminology, it is time to embark on the main quest in this exciting adventure. In this chapter, we will learn about the structure of a Molehill application. What goes where? How is everything put together? What is this "Stage3D" we keep hearing about?

Flash 11 graphics are incredibly fast. As we saw in *Chapter 1*, the reason that it runs so efficiently is that Adobe made the intelligent decision to make your video card do all the work. Molehill does not do things the way you are used to. If you are familiar with 2D Flash programming in AS3, it will feel like you are exploring a strange and unfamiliar land. This exploration of the unknown is exciting and new. So, put on your adventure gear and dive in!

The old fashioned way

Imagine a medieval world in which all combat is performed with traditional weapons such as swords. This is like previous versions of Flash, where everything is done without hardware 3D acceleration. Suddenly, magic is discovered. It changes everything. This is like the invention of Molehill.

Tried and trusted techniques and tools still work very well. Now, however, an additional element exists which can be added to the mix, to add a little sparkle, to add a dash of magic.

You will still need your old knowledge—Molehill does not replace anything in Flash—it simply adds to your bag of tricks.

In old fashioned Flash animations, the stage (the image which is drawn on the screen) is prepared in software by your CPU. Each dot you see, each pixel, is calculated by sorting various sprites that are part of the display list and by deciding what its color should be.

This process is incredibly slow. It is handy for animators who like to place layers of vectors and tween them to their heart's content, but it is so terribly inefficient and calculation-intensive that the frame rate of your game will often suffer.

This old approach to drawing Flash games is the reason why you cannot draw very many shapes on the screen at the same time without your game slowing to a crawl. This is the reason why you cannot overlay too many semi-transparent objects over the top of each other before your game looks like a slide show rather than a smooth animation.

This is the reason why Molehill was created: games need frame rate! The old way of doing things was simply too slow.

The Molehill way: Stage3D

Games that use Stage3D will bypass all of this handy-but-inefficient CPU-based heavy lifting. Instead of being part of the standard Flash display list, instead of playing nicely with the old fashioned way of doing things, Molehill games have a direct path to the video RAM. Instead of calculating pixels in RAM and then copying these values to the screen when everything is done, Stage3D actually writes directly onto the video RAM in one pass.

Stage3D relies on DirectX 9 on Windows machines and OpenGL 1.3 on both Mac OS-X and Linux. On mobile devices such as tablets, televisions, and phones, it will reply upon OpenGL ES2. For devices that do not support any of these three APIs, Flash takes advantage of a very fast CPU rasterizer named SwiftShader.

By choosing the fastest technology that will work on any given platform, Stage3D is designed to be able to work on any device. This includes both the super powerful gaming rigs with all the latest hardware and all the newest device drivers to your granny's tired dusty e-mail-only computer. As there is a software-only fallback method (SwiftShader), you can be sure that your games will run even on computers that don't have a GPU (3D video card).

The great news is that you do not have to worry about what is happening under the hood. It does not matter if you are running Windows, OSX, Android or iOS. Desktop PC, phone, tablet or TV, all are about to be Stage3D capable. You never have to choose a different API for OpenGL or DirectX for example. Stage3D makes all the tough decisions and handles everything for you automatically, behind the scenes.

Using 2D Flash text and sprites

Molehill games are still Flash files. You can use any of the older Flash classes in exactly the same way as before. You can add text, play sounds, stream videos, download XML files, and use `addChild()` to put anything you like on stage, just as before. They all sit over the top of your Stage3D objects.



This means that if you want to overlay 2D Flash graphics like sprites or text on top of your 3D graphics, you can. However, you cannot do it the other way around: you cannot draw a Stage3D object over the top of any Flash 2D objects. It is rendered first, underneath any regular Flash.

The great news is that because Molehill can coexist with older Flash 2D `DisplayObjects`, you can use all your old code that deals with buttons, videos, animations, or sounds. You simply overlay any Flash `DisplayObjects` you wish on top of your 3D game.

This is a perfect way to draw the HUD (heads-up-display) for your game. You could use Flash 2D objects to draw health bars, the player's score, buttons, watermarks, title screens, or anything else you wish to do outside of the 3D world.

Why is Stage3D so fast?

Adobe has bypassed many of the bottlenecks used in previous versions of Flash in order to get better performance. One big change from the old fashioned way of doing things is that your Stage3D objects are not `DisplayObjects` (the name for all things in Flash you can draw). `DisplayObjects` are the base class of standard Flash object that are placed in the `DisplayList` and can have rotations, blend modes, filters, and many other effects applied to them.

You can still use all of these older classes in your games. The important thing to note is that Stage3D objects are not DisplayObjects—they enable full GPU acceleration. With this new rendering model called Stage Video, the Flash player draws all the data directly onto the video memory, which is painted on the screen using the GPU.

As Flash got rid of the "middle man", the performance is insanely good. However, there are limitations that you need to be aware of as well: Stage3D objects are not inside the DisplayList in Flash! As they are not DisplayObjects, you cannot apply filters or blend modes. You cannot even put a Stage3D object over the top of other Flash 2D graphics!

Flash has given the control of the region defined by your Stage3D to your GPU. It does not play "by the rules". It achieves incredible performance because it bypasses the DisplayList and paints directly on-screen from the GPU. No more read-back is required to retrieve frames from memory to push them on-screen through the display list.

Molehill is a special case where we give complete control of the rendering to your GPU. Flash takes a hands-off approach when dealing with Stage3D objects. They get priority and they don't come with any of the bells-n-whistles that basic (but bloated and inefficient) Flash DisplayObjects do.

This is a major design constraint, but it is worth it for the speed. The good news is that you can always work around this limitation if you absolutely need 3D rendered on top of your 2D. One way to do this would be to render all your 2D graphics (regular Flash DisplayObjects) off-screen and copy those pixels onto a BitmapData which can be uploaded as the new texture data for use in your 3D world. For example, you could decode a streaming video and copy its BitmapData to a 3D rectangle in your 3D world and place that rectangle behind other 3D graphics.

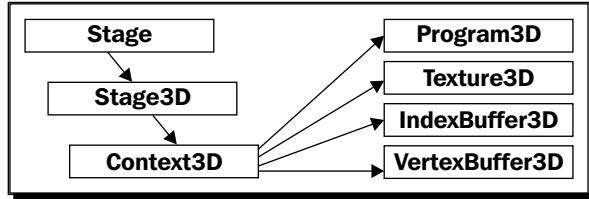
Do not despair, anything is possible. You can draw anything in any order, but you may have to work around this one important consideration: Molehill always gets to draw first—underneath any old fashioned DisplayObjects.

Molehill is super fast, super trim, and super simple. It only does one thing, and does it well!

The structure of a Molehill application

In Flash, the stage is what is drawn on the screen. Sprites of various kinds (bitmaps, text input boxes, vector shapes) are attached to the stage. Molehill-enabled Flash animations attach a special kind of object, called a Stage3D in the same way. You can have more than one Stage3D on your stage. Each Stage3D has a Context3D object.

The Context3D is the "engine". It is your Molehill object.



Your games will generally only put one Stage3D on your Stage, and it will only use a single Context3D. This Context3D is the object that you send all your textures, vertex buffers, and vertex/fragment programs to. The Context3D is the base class that holds all of your game data and does all the work.

As a game developer using Molehill, you interact with two main objects: a Stage3D and a Context3D. You request a Context3D from the Flash player, and it instructs your video card to get ready to draw on-screen.

Stage

This is the regular Flash stage; anything that is supposed to be drawn on the screen is put here.

Stage3D

Stage3D is the object that sits inbetween Flash and all the GPU power. Unlike regular Flash 2D objects such as TextAreas or Sprites, a Molehill Stage3D does not sit in the display list. This means you cannot do an `addChild()` on a Stage3D object. It sits behind all your regular flash content. It is always on the lowest layer of the stage. You can have more than one Stage3D on your stage.

Context3D

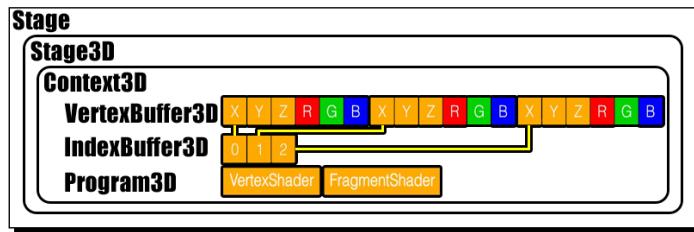
Your Stage3D will contain a Context3D. Similar to how a bitmap object contains BitmapData and when interacting with bitmaps you generally spend all your time actually affecting the BitmapData that it holds, in Molehill you rarely interact with the Stage3D and instead do all the work with the Context3D. You can think of the Context3D as the "3D engine". It is the object to which you send all your 3D data. When you upload the vertex data to your video card, you send an array of numbers to your Context3D.

VertexBuffer3D

As we learned in the previous chapter, a model in a 3D game—called a mesh—is comprised of hundreds or thousands of vertices. Each vertex defines the x,y,z coordinates of a "corner" on your mesh. A VertexBuffer3D is an array of numbers that can be used to define these coordinates.

The handy thing is that you can put any sort of data here that you want. You are not limited to just x,y,z, you are free to put color information (r,g,b), texture UV coordinate data, and even custom data used by your vertex or fragment shaders.

When you are initializing your 3D game, you send this giant array of data to Molehill by sending a VertexBuffer3D to your Context3D. The Context3D will then upload this data directly to your video card's RAM, or VRAM, ready for using during rendering.



IndexBuffer3D

As the VertexBuffer3D is simply a giant chunk of numerical data, Flash needs to know where each "chunk" begins and ends. An index buffer simply tells it which piece of data belongs to which vertex. In the preceding example image, you can see that the index buffer has defined three vertex entries, each having an x,y,z and an r,g,b value. As there are three entries in your IndexBuffer, the mesh you are defining must be a triangle: a polygon with three corners.

Program3D

Vertex programs and fragment programs—also called as shaders—are uploaded to your video card so they can be run on the GPU (the graphics processing unit in your 3d hardware). Programs are compiled from a special language that looks much like the assembly language. They tell your GPU how to draw the mesh for which you just uploaded all that data. In future chapters, we will dive into the creation of these programs, but for now, simply think of them as lists of drawing commands that are stored in VRAM (video ram) alongside your vertex and index buffers.

Flowchart of a Molehill program

When you use Molehill, there are really two parts to your program: the setup and the render loop.

Just like an adventuring hero who needs an hour to put on armor, sharpen weapons and learn some new spells, getting everything ready for your game is a complex process. It is much more complicated than the process of actually drawing your meshes on screen.

During the setup, all your buffers and programs are compiled and sent to the video card. This is done only once. Once set up, Flash can re-draw your meshes in new locations or even multiple times without requiring any further inits. The setup can take a second or two, but once completed all the data is ready and waiting in VRAM, on call to be used over and over again during the game.

All this preparation—creating buffers and programs—helps to improve drawing performance because, once defined, it all resides right "on the metal" (it runs from inside your GPU and is no longer dealt with by the CPU).

Time for action – things we do once, during the setup

1. Request a Context3D instance from Stage3D
2. Set up the Context3D (telling it how big it is, and so on)
3. Create a VertexBuffer (one for each mesh you intend to use)
4. Create an IndexBuffer (each will correspond to a VertexBuffer)
5. Upload this data to the Context3D (send it to your GPU)
6. "Compile" a VertexProgram (turn your shader source code into bytes)
7. "Compile" a FragmentProgram (turn your shader source code into bytes)
8. Create a Program3D which will use the preceding shaders (can be used by more than one mesh)

What just happened?

The preceding steps are more of a general list of common actions performed during init. Some games will do things in different order, but this is a good guide to what is going on behind the scenes in the most common scenarios.

The first steps in getting the Stage3D API up and running are to request that Flash set up a 3D context ready for your game. Once one has been returned (an event will tell you when it is ready), you are free to upload the mesh data, programs, and textures to the context.

All of these eight steps are run only once during the initialization of your game, although games may occasionally have to re-initialize the Context3D due to the way computers are forced by the operating system to "let go" of the 3D card (such as when a computer goes to sleep and re-awakens).

Once everything is set to go, the game will enter what is called the render loop. This is the game's state for the rest of the time it is running. Typically, you will create a recurring event handler (`ENTER_FRAME`) which is fired every single frame.

Time for action – things we do over and over again

1. Clear the Context3D (erase the old view)
2. Tell the Context3D which Program3D to use (such as a "rusty metal" shader)
3. Tell the Context3D which VertexBuffer to use (for example, a spaceship)
4. Setup data for your shaders (variables, such as the new location of a spaceship)
5. Draw some triangles (tell the Context3D to do some drawing)
6. Update the screen (show the user the new scene we just rendered)

What just happened?

The preceding steps are a generalized list of tasks that your game will have to perform each and every frame, as often as possible. This ever-repeating set of actions is called the render loop and could include many other things not listed earlier, such as physics simulation, artificial intelligence (such as moving enemies), stepping through animations, triggering sounds, updating the score, checking for collisions, and much more. As opposed to inits, the render loop is where your gameplay actually takes place.

Pop quiz – earn some experience points

Here are a few quick questions to test your understanding. If you can answer them, then you are officially ready to move on to the next step in your grand adventure.

1. Why is Molehill so fast?
 - a. It uses 3D hardware acceleration
 - b. All mesh and shader data is processed on the GPU
 - c. The CPU is free to do other things and doesn't do the rendering
 - d. All of the above

2. When you upload a VertexBuffer, which object will you send it to?
 - a. Stage
 - b. Stage3D
 - c. Context3D
 - d. Your blog
3. The simplest possible mesh, a triangle, would have how many entries in its index buffer (each corresponding to one vertex)?
 - a. 1
 - b. 3
 - c. 2
 - d. 6

Summary

We learned a lot in this chapter about how a Flash 11 Stage3D (Molehill) app is put together.

Specifically, we covered:

- ◆ **The old fashioned way:** 2D Flash using DisplayObjects
- ◆ **The Molehill way:** Stage3Ds containing Context3Ds
- ◆ **Why Molehill is so fast:** Hardware 3D, baby!
- ◆ **Combining 2D and 3D:** You can use old and new Flash objects together
- ◆ **Flowchart of a Molehill game:** Setup first, then a render loop

Now that we have learned how Molehill works, we are ready to put this knowledge to good use.

Level 2 achieved!

Congratulations! You have "leveled up". As a level two Molehill master, you have earned the right to start rendering your first 3D scenes in Flash, which is the topic of the next chapter.

3

Fire up the Engines!

You are about to reach Level 3.

Now that you have learned the basic terms and structure of the Stage3D API, the time has come to put this knowledge to the test. Prepare for your first "boss battle", your foray into the 3D arena. Get ready to create your first demo!

Grab your weapons and armor, brave adventurer. Some coders like to work in Flex, while others use pure AS3, while still others like to stay within the familiar Adobe Flash IDE. The choice is yours.

In this chapter, we will:

- ◆ Obtain Flash 11 for your browser
- ◆ Get all the tools ready to compile Stage3D games
- ◆ Initialize 3D graphics in Flash
- ◆ Send mesh and texture data to the video card
- ◆ Animate a simple 3D scene

Before we begin programming, there are two simple steps required to get everything ready for some amazing 3D graphics demos. Step 1 is to obtain the Flash 11 plugin and the Stage3D API. Step 2 is to create a template as3 project and test that it compiles to create a working Flash SWF file.

Once you have followed these two steps, you will have properly "equipped" yourself. You will truly be ready for the battle. You will have ensured that your tool-chain is set up properly. You will be ready to start programming a 3D game.

Step 1: Downloading Flash 11 (Molehill) from Adobe

Depending on the work environment you are using, setting things up will be slightly different. The basic steps are the same regardless of which tool you are using but in some cases, you will need to copy files to a particular folder and set a few program options to get everything running smoothly.

If you are using tools that came out before Flash 11 went live, you will need to download some files from Adobe which instruct your tools how to handle the new Stage3D functions. The directions to do so are outlined below in *Step 1*.

In the near future, of course, Flash will be upgraded to include Stage3D. If you are using CS5.5 or another new tool that is compatible with the Flash 11 plugin, you may not need to perform the steps below. If this is the case, then simply skip to *Step 2*.

Assuming that your development tool-chain does not yet come with Stage3D built in, you will need to gather a few things before we can start programming. Let's assemble all the equipment we need in order to embark on this grand adventure, shall we?

Time for action – getting the plugin

It is very useful to be running the debug version of Flash 11, so that your trace statements and error messages are displayed during development. Download Flash 11 (content debugger) for your web browser of choice.

At the time of writing, Flash 11 is in beta and you can get it from the following URL:

<http://labs.adobe.com/downloads/flashplayer11.html>

Naturally, you will eventually be able to obtain it from the regular Flash download page:

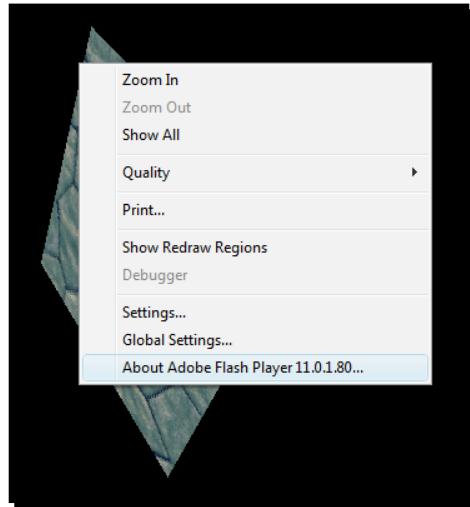
<http://www.adobe.com/support/flashplayer/downloads.html>

On this page, you will be able to install either the Active-X (IE) version or the Plugin (Firefox, and so on) version of the Flash player. This page also has links to an uninstaller if you wish to go back to the old version of Flash, so feel free to have some fun and don't worry about the consequences for now.

Finally, if you want to use Chrome for debugging, you need to install the plugin version and then turn off the built-in version of Flash by typing `about:plugins` in your Chrome address bar and clicking on **Disable** on the old Flash plugin, so that the new one you just downloaded will run.

We will make sure that you installed the proper version of Flash before we continue.

To test that your browser of choice has the Stage3D-capable incubator build of the Flash plugin installed, simply right-click on any Flash content and ensure that the bottom of the pop-up menu lists Version 11,0,1,60 or greater, as shown in the following screenshot. If you don't see a version number in the menu, you are running the old Flash 10 plugin.



Additionally, in some browsers, the 3D acceleration is not turned on by default. In most cases, this option will already be checked. However, just to make sure that you get the best frame rate, right-click on the Flash file and go to **options**, and then enable hardware acceleration, as shown in the following screenshot:



You can read more about how to set up Flash 11 at the following URL:

[http://labs.adobe.com/technologies/flashplatformruntimes/
flashplayer11/](http://labs.adobe.com/technologies/flashplatformruntimes/flashplayer11/)

Time for action - getting the Flash 11 profile for CS5

Now that you have the Stage3D-capable Flash plugin installed, you need to get Stage3D working in your development tools. If you are using a tool that came out after this book was written that includes built-in support for Flash 11, you don't need to do anything—skip to *Step 2*.

If you are going to use Flash IDE to compile your source code and you are using Flash CS5, then you need to download a special .XML file that instructs it how to handle the newer Stage3D functionality. The file can be downloaded from the following URL:

[http://download.macromedia.com/pub/labs/flashplatformruntimes/
incubator/flashplayer_inc_flashprofile_022711.zip](http://download.macromedia.com/pub/labs/flashplatformruntimes/incubator/flashplayer_inc_flashprofile_022711.zip)

If the preceding link no longer works, do not worry. The files you need are included in the source code that accompanies this book. Once you have obtained and unzipped this file, you need to copy some files into your CS5 installation.

- ◆ FlashPlayer11.xml goes in:
Adobe Flash CS5\Common\Configuration\Players
- ◆ playerglobal.swc goes in:
Adobe Flash CS5\Common\Configuration\ActionScript 3.0\FP11

Restart Flash Professional after that and then select 'Flash Player 11' in the publish settings. It will publish to a SWF13 file.

As you are not using Flex to compile, you can skip all of the following sections regarding Flex. Simple as it can be!

Time for action – upgrading Flex

If you are going to use pure AS3 (by using FlashDevelop or Flash Builder), or even basic Flex without any IDE, then you need to compile your source code with a newer version of Flex that can handle Stage3D.

At the time of writing, the best version to use is build 19786. You can download it from the following URL:

<http://opensource.adobe.com/wiki/display/flexsdk/Download+Flex+Hero>

Remember to change your IDE's compilation settings to use the new version of Flex you just downloaded.

For example, if you are using Flash Builder as part of the Adobe Flex SDK, create a new ActionScript project: **File | New | ActionScript project**. Open the project Properties panel (right-click and chose **Properties**). Select **ActionScript Compiler** from the list on the left. Use the **Configure Flex SDK's** option in the upper-right hand corner to point the project to **Flex build 19786** and then click on **OK**.

Alternately, if you are using FlashDevelop, you need to instruct it to use this new version of Flex by going into **Tools | Program Settings | AS3 Context | Flex SDK Location** and browsing to your new Flex installation folder.

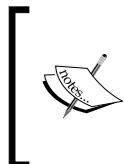
Time for action – upgrading the Flex playerglobal.swc

If you use FlashDevelop, Flash Builder, or another tool such as FDT, all ActionScript compiling is done by Flex. In order to instruct Flex about the Stage3D-specific code, you need a small file that contains definitions of all the new AS3 that is available to you.

It will eventually come with the latest version of these tools and you won't need to manually install it as described in the following section. During the Flash 11 beta period, you can download the Stage3D-enabled `playerglobal.swc` file from the following URL:

http://download.macromedia.com/pub/labs/flashplatformruntimes/flashplayer11/flashplayer11_b1_playerglobal_071311.swc

Rename this file to `playerglobal.swc` and place it into an appropriate folder. Instruct your compiler to include it in your project. For example, you may wish to copy it to your Flex installation folder, in the `flex/frameworks/libs/player/11` folder.



In some code editors, there is no option to target Flash 11 (yet). By the time you read this book, upgrades may have enabled it. However, at the time of writing, the only way to get FlashDevelop to use the SWC is to copy it over the top of the one in the `flex/frameworks/libs/player/10.1` folder and target this new "fake" Flash 10.1 version.

Once you have unzipped Flex to your preferred location and copied `playerglobal.swc` to the preceding folder, fire up your code editor. Target Flash 11 in your IDE—or whatever version number that is associated with the folder, which you used as the location for `playerglobal.swc`. Be sure that your IDE will compile this particular SWC along with your source code.

In order to do so in Flash Builder, for example, simply select "Flash Player 11" in the Publish Settings. If you use FlashDevelop, then open a new project and go into the **Project→Properties → Output Platform Target** drop-down list.

Time for action – using SWF Version 13 when compiling in Flex

Finally, Stage3D is considered part of the future "Version 13" of Flash and therefore, you need to set your compiler options to compile for this version. You will need to target SWF Version 13 by passing in an extra compiler argument to the Flex compiler: `-swf-version=13`.

1. If you are using Adobe Flash CS5, then you already copied an XML file which has all the changes, as outlined below and this is done automatically for you.
2. If you are using Flex on the command line, then simply add the preceding setting to your compilation build script command-line parameters.
3. If you are using Flash Builder to compile Flex, open the project Properties panel (right-click and choose **Properties**). Select **ActionScript Compiler** from the list on the left. Add to the **Additional compiler arguments** input: `-swf-version=13`. This ensures the outputted SWF targets SWF Version 13. If you compile on the command line and not in Flash Builder, then you need to add the same compiler argument.
4. If you are using FlashDevelop, then click on **Project | Properties | Compiler Options | Additional Compiler Options**, and add `-swf-version=13` in this field.

Time for action – updating your template HTML file

You probably already have a basic HTML template for including Flash SWF files in your web pages. You need to make one tiny change to enable hardware 3D.

Flash will not use hardware 3D acceleration if you don't update your HTML file to instruct it to do so. All you need to do is to always remember to set `wmode=direct` in your HTML parameters.

For example, if you use JavaScript to inject Flash into your HTML (such as `SWFObject.js`), then just remember to add this parameter in your source. Alternately, if you include SWFs using basic HTML object and `embed` tags, your HTML will look similar to the following:

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
width="640" height="480"><param name="src" value="Molehill.swf" />
<param name="wmode" value="direct" /><embed type="application/
x-shockwave-flash" width="640" height="480" src="Molehill.swf"
wmode="direct"></embed></object>
```

The only really important parameter is `wmode=direct` (and the name of your swf file)—everything else about how you put Flash into your HTML pages remains the same.

In the future, if you are running 3D demos and the frame rate seems really choppy, you might not be using hardware 3D acceleration. Be sure to view-source of the HTML file that contains your SWF and check that all mentions of the `wmode` parameter are set to direct.

Stage3D is now set up!

That is it! You have officially gone to the weapons store and equipped yourself with everything you require to explore the depths of Flash 11 3D graphics. That was the hard part. Now we can dive in and get to some coding, which turns out to be the easy part.

Step 2: Start coding

In Step one, you downloaded and installed everything you need to get the Stage3D source code to compile. Whether you use Adobe CS5, Flash Builder, FlashDevelop, or you compile Flex from the command line, the actual AS3 source code required to get Stage3D working is exactly the same.

This is the demo that you will program in this chapter:

http://www.mcfunkypants.com/molehill/chapter_3_demo/

If you are the impatient type, then you can download the final source code project that you would create if you followed the steps below from the following URL:

<http://www.packtpub.com/adobe-flash11-stage3d-molehill-game-programming-beginners-guide/book>

For the sake of learning, however, why not go through the following steps, so that you actually understand what each line does? It is a poor warrior who expects to hone their skills without training.

Time for action – creating an empty project

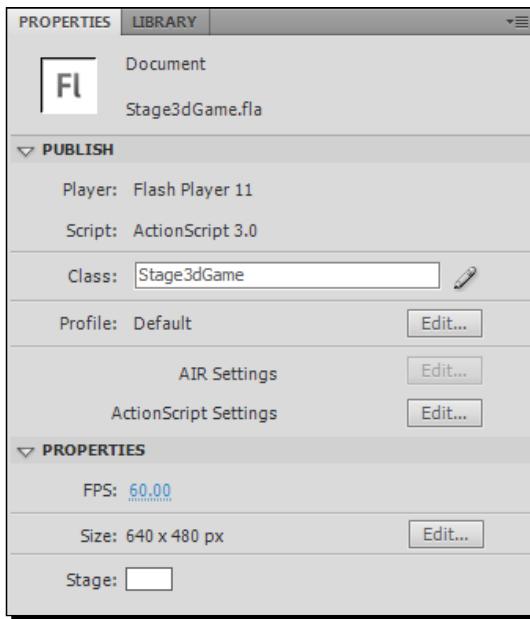
Your next quest is simply to prepare an empty project in whatever tool-set you like.

For the sake of matching the following source code, create a class named `Stage3dGame` that extends `Sprite`, which is defined in an `as3` file named `Stage3dGame.as`.

How you do so depends on your tool of choice.

Fire up the Engines!

Flash veterans and artists often prefer to work within the comfortable Adobe Flash IDE environment, surrounded by their old friends, the timeline, and the library palette. Create a brand new .FLA file, and prepare your project and stage as you see fit. Don't use any built-in Flash MovieClips for now. Simply create an empty .FLA that links to an external .AS3 file, as shown in the following screenshot:



Some game developers prefer to stick with pure AS3 projects, free of any bloat related to the Flash IDE or Flex that uses MXML files. This technique results in the smallest source files, and typically involves use of open source (free) code editors such as FlashDevelop. If this is your weapon of choice, then all you need to do is set up a blank project with a basic as3 file that is set to compile by default.

No matter what tool you are using, once you have set up a blank project, your ultra-simplistic as3 source code should look something like the following:

```
package
{
    [SWF(width="640", height="480", frameRate="60",
        backgroundColor="#FFFFFF")]

    public class Stage3dGame extends Sprite
    {
    }
}
```

What just happened?

As you might imagine, the preceding source code does almost nothing. It is simply a good start, an empty class, ready for you to fill in with all sorts of 3D game goodness. Once you have a "blank" Flash file that uses the preceding class and compiles without any errors, you are ready to begin adding the Stage3D API to it.

Time for action – importing Stage3D-specific classes

In order to use vertex and fragment programs (shaders), you will need some source code from Adobe. The files `AGALMiniAssembler.as` and `PerspectiveMatrix3D.as` are included in the project source code that comes with this book. They belong in your project's source code folder in the subdirectory `com/adobe/utils/`, so they can be included in your project.

Once you have these new `.as` files in your source code folder, add the following lines of code which import various handy functions immediately before the line that reads "public class Stage3dGame extends Sprite".

```
import com.adobe.utils.*;
import flash.display.*;
import flash.display3D.*;
import flash.display3D.textures.*;
import flash.events.*;
import flash.geom.*;
import flash.utils.*;
```

What just happened?

In the lines of the preceding code, you instruct Flash to import various utility classes and functions that you will be using shortly. They include 3D vector math, Stage3D initializations, and the assembler used to compile fragment and vertex programs used by shaders.

Try to compile the source. If you get all sorts of errors that mention unknown Molehill-related classes (such as `display3D`), then your compiler is most likely not set up to include the `playerglobal.swc` we downloaded earlier in this chapter. You will need to check your compiler settings or Flex installation to ensure that the brand new Stage3D-capable `playerglobal.swc` is being used as opposed to an older version.

If you get errors complaining about missing `com.adobe.utils`, then you may not have unzipped the `AGALMiniAssembler.as` and `PerspectiveMatrix3D.as` code into the correct location. Ensure these files are in a subfolder of your source directory called `com/adobe/utils/`.

If your code compiles without errors, you are ready to move on.

Time for action – initializing Molehill

The next step is to actually get the Stage3D API up and running. Add the following lines of code to your project inside the empty class you created by updating the empty Stage3dGame function and adding the init function below it as follows:

```
public function Stage3dGame()
{
    if (stage != null)
        init();
    else
        addEventListener(Event.ADDED_TO_STAGE, init);
}

private function init(e:Event = null):void
{
    if (hasEventListener(Event.ADDED_TO_STAGE))
        removeEventListener(Event.ADDED_TO_STAGE, init);

    // class constructor - sets up the stage
    stage.scaleMode = StageScaleMode.NO_SCALE;
    stage.align = StageAlign.TOP_LEFT;

    // and request a context3D from Stage3d
    stage.stage3Ds[0].addEventListener(
        Event.CONTEXT3D_CREATE, onContext3DCreate);
    stage.stage3Ds[0].requestContext3D();
}
```

What just happened?

This is the constructor for your Stage3dGame class, followed by a simple init function that is run once the game has been added to the stage.

The init function instructs Flash how to handle the stage size and then requests a Context3D to be created. As this can take a moment, an event is set up to instruct your program when Flash has finished setting up your 3D graphics.

Time for action – defining some variables

Next, your demo is going to need to store a few variables. Therefore, we will define these at the very top of your class definition, above any of the functions, as follows:

```
// constants used during inits
private const swfWidth:int = 640;
```

```
private const swfHeight:int = 480;
private const textureSize:int = 512;

// the 3d graphics window on the stage
private var context3D:Context3D;
// the compiled shader used to render our mesh
private var shaderProgram:Program3D;
// the uploaded vertexes used by our mesh
private var vertexBuffer:VertexBuffer3D;
// the uploaded indexes of each vertex of the mesh
private var indexBuffer:IndexBuffer3D;
// the data that defines our 3d mesh model
private var meshVertexData:Vector.<Number>;
// the indexes that define what data is used by each vertex
private var meshIndexData:Vector.<uint>;

// matrices that affect the mesh location and camera angles
private var projectionMatrix:PerspectiveMatrix3D =
    new PerspectiveMatrix3D();
private var modelMatrix:Matrix3D = new Matrix3D();
private var viewMatrix:Matrix3D = new Matrix3D();
private var modelViewProjection:Matrix3D = new Matrix3D();

// a simple frame counter used for animation
private var t:Number = 0;
```

What just happened?

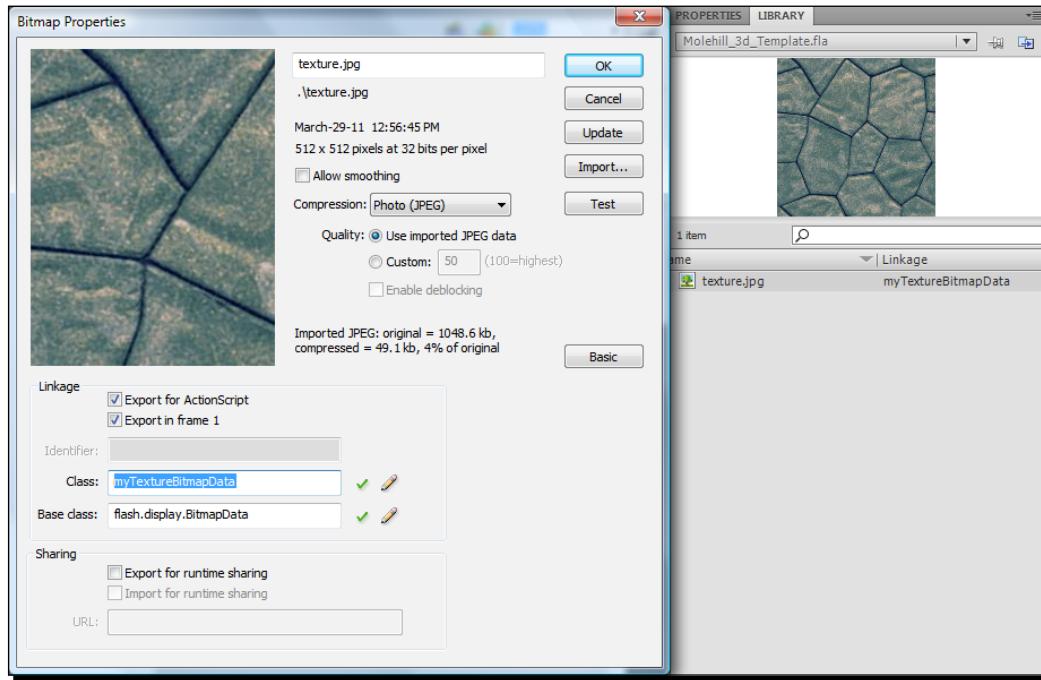
The demo you are writing needs to store things such as the current camera angle, the vertex, and fragment programs that we are about to create, and more. By defining them here, each of the functions we are about to write can access them.

Time for action – embedding a texture

Before we start creating the functions that perform all the work, let's also define a texture. Copy any 512x512 jpeg image into your source folder where you are putting all the files for this demo.

Fire up the Engines!

If you are using Flex or a pure AS3 programming environment such as Flash Builder or FlashDevelop, then you don't need to do anything further. If you are using Adobe Flash CS5, then you will need to open the Library palette (*F11*) and drag-and-drop the *.jpg* file, so that the image is part of your *.FLA* file's library, as shown in the following screenshot:



Once this texture is in the library, right-click on it and open properties. Click on the **Advanced** button to view more options and turn on the check mark that enables **Export for ActionScript** and give the new class the name `myTextureBitmapData`. This will be used below.

If you are using Flash CS5, then add the following code just after the other variables you recently defined:

```
private var myBitmapDataObject:myTextureBitmapData =  
    new myTextureBitmapData(texture_size, texture_size);  
private var myTextureData:Bitmap =  
    new Bitmap(myBitmapDataObject);  
// The Molehill Texture that uses the above myTextureData  
private var myTexture:Texture;
```

If you are using Flex or a pure AS3 environment, you do not have a "library" and instead, can embed assets using a line of code. Instead of the preceding code, define your texture in the following way:

```
[Embed (source = "texture.jpg")] private var myTextureBitmap:Class;
private var myTextureData:Bitmap = new myTextureBitmap();
// The Molehill Texture that uses the above myTextureData
private var myTexture:Texture;
```

What just happened?

The code you just entered embeds the JPG image you selected for use as a texture. This texture will eventually be drawn on the mesh we are about to define.

Time for action – defining the geometry of your 3D mesh

For the purposes of this simple demo, all we need to define is a "quad" (a square). We will define it now as follows:

```
private function initData():void
{
    // Defines which vertex is used for each polygon
    // In this example a square is made from two triangles
    meshIndexData = Vector.<uint>
    (
        [
            0, 1, 2,      0, 2, 3,
        ]);
    // Raw data used for each of the 4 vertexes
    // Position XYZ, texture coordinate UV, normal XYZ
    meshVertexData = Vector.<Number>
    (
        [
            //X,   Y,   Z,   U,   V,   nX,   nY,   nZ
            -1, -1,  1,   0,   0,   0,   0,   1,
            1,  -1,  1,   1,   0,   0,   0,   1,
            1,   1,  1,   1,   1,   0,   0,   1,
            -1,  1,  1,   0,   1,   0,   0,   1
        ]);
}
```

What just happened?

The preceding function fills a couple of variables with numerical data. This data is eventually sent to the video card and is used to define the locations of each vertex in your 3D mesh. For now, all we have defined is a simple square, which is made up of two triangles that use a total of four vertexes. Eventually, your models will be complex sculptures, made up of thousands of polies.

Anything from a sword to an entire city can be constructed by listing the x,y,z locations in space for each of a mesh's vertexes. For now, a simple square will be proof-of-concept. Once we can get a square spinning around in 3D, adding more detail is a trivial process.

Time for action – starting your engines

Recall that the `init()` function requests a `Context3D` object. An event handler was set up that Flash will run when your video card has prepared itself and is ready to receive data. Let's define this event handler.

The perfect place for this new snippet of code is just below the `init()` function:

```
private function onContext3DCreate(event:Event):void
{
    // in case it is not the first time this event fired
    removeEventListener(Event.ENTER_FRAME,enterFrame);

    // Obtain the current context
    var t:Stage3D = event.target as Stage3D;
    context3D = t.context3D;

    if (context3D == null)
    {
        // Currently no 3d context is available (error!)
        return;
    }

    // Disabling error checking will drastically improve performance.
    // If set to true, Flash will send helpful error messages regarding
    // AGAL compilation errors, uninitialized program constants, etc.
    context3D.enableErrorChecking = true;

    // Initialize our mesh data
    initData();
```

What just happened?

Inside the `onContext3DCreate` event handler, all your Stage3D inits are performed. This is the proper moment for your game to upload all the graphics that will be used during play.

The reasons you cannot upload data during the constructor you already wrote are:

- ◆ It can take a fraction of a second before your device drivers, 3D card, operating system, and Flash have prepared themselves for action.
- ◆ Occasionally, in the middle of your game, the Context3D can become invalidated.
- ◆ This can happen, for example, if the user's computer goes to "sleep", or if they hit *Ctrl-Alt-Delete*. For this reason, it is entirely possible that during the play, the mesh and texture data will need to be re-sent to your video RAM. As it can happen more than once, this event handler will take care of everything whenever it is needed.

If you read the comments, you will be able to follow along. Firstly, as the event might fire more than once, any animation is turned off until all data has been re-sent to the video card. A Context3D object is obtained and so we remember it by assigning it to one of the variables we defined earlier. We turn on error checking, which is handy during development. Once we are finished with our game, we will turn this off in order to get a better frame rate.

Time for action – adding to the `onContext3DCreate` function

The next thing we need to do in our `onContext3DCreate` function is to define the size of the area we want to draw to and create a simple shader that instructs Stage3D how to draw our mesh. We will learn what all these shader commands mean in a future chapter. Continue adding to the function as follows:

```
// The 3d back buffer size is in pixels
context3D.configureBackBuffer(swfWidth, swfHeight, 0, true);

// A simple vertex shader which does a 3D transformation
var vertexShaderAssembler:AGALMiniAssembler =
    new AGALMiniAssembler();
vertexShaderAssembler.assemble
(
    Context3DProgramType.VERTEX,
    // 4x4 matrix multiply to get camera angle
    "m44 op, va0, vc0\n" +
    // tell fragment shader about XYZ
    "mov v0, va0\n" +
    // tell fragment shader about UV
```

```
"mov v1, va1\n"
) ;

// A simple fragment shader which will use
// the vertex position as a color
var fragmentShaderAssembler:AGALMiniAssembler
  = new AGALMiniAssembler();
fragmentShaderAssembler.assemble
(
  Context3DProgramType.FRAGMENT,
  // grab the texture color from texture fs0
  // using the UV coordinates stored in v1
  "tex ft0, v1, fs0 <2d,repeat,miplinear>\n" +
  // move this value to the output color
  "mov oc, ft0\n"
) ;

// combine shaders into a program which we then upload to the GPU
shaderProgram = context3D.createProgram();
shaderProgram.upload(vertexShaderAssembler.agalcode,
  fragmentShaderAssembler.agalcode);
```

What just happened?

A back-buffer is set up, which is a temporary bitmap in the video RAM where all the drawing takes place. As each polygon is rendered, this back-buffer slowly becomes the entire scene, which when completed is presented to the user.

Two AGALMiniAssembler objects are created and a string containing **AGAL (Adobe Graphics Assembly Language)** is turned into compiled byte code. Don't worry too much about the specific AGAL code for now, we will dive into fragment and vertex programs in later chapters. Essentially, these AGAL commands instruct your video card exactly how to draw your mesh.

We will continue working with the Context3DCreate function.

Time for action – uploading our data

In order to render the mesh, Stage3D needs to upload the mesh and texture data straight to your video card. This way, they can be accessed repeatedly by your 3D hardware without having to make Flash do any of the "heavy lifting".

```
// upload the mesh indexes
indexBuffer = context3D.createIndexBuffer(meshIndexData.length);
```

```
indexBuffer.uploadFromVector(meshIndexData, 0, meshIndexData.length);

// upload the mesh vertex data
// since our particular data is
// x, y, z, u, v, nx, ny, nz
// each vertex uses 8 array elements
vertexBuffer = context3D.createVertexBuffer(
    meshVertexData.length/8, 8);
vertexBuffer.uploadFromVector(meshVertexData, 0,
    meshVertexData.length/8);

// Generate mipmaps
myTexture = context3D.createTexture(textureSize, textureSize,
    Context3DTextureFormat.BGRA, false);
var ws:int = myTextureData.bitmapData.width;
var hs:int = myTextureData.bitmapData.height;
var level:int = 0; var tmp:BitmapData;
var transform:Matrix = new Matrix();
tmp = new BitmapData(ws, hs, true, 0x00000000);
while (ws >= 1 && hs >= 1) {
    tmp.draw(myTextureData.bitmapData, transform, null, null,
        null, true);
    myTexture.uploadFromBitmapData(tmp, level);
    transform.scale(0.5, 0.5); level++; ws >>= 1; hs >>= 1;
    if (hs && ws) {
        tmp.dispose();
        tmp = new BitmapData(ws, hs, true, 0x00000000);
    }
}
tmp.dispose();
```

What just happened?

In the preceding code, our mesh data is uploaded to the video card. A vertex buffer and an index buffer are sent, followed by your texture data. There is a short loop that creates similar, but smaller versions of your texture and uploads for each one. This technique is called MIP mapping. By uploading a 512x512 image, followed by the one that is 256x256, then 128x128, and so on down to 1x1, the video card has a set of textures that can be used depending on how far away or acutely angled the texture is to the camera. MIP mapping ensures that you don't get any "jaggies" or "moiree patterns" and increases the quality of the visuals.

Time for action – setting up the camera

There is one final bit of code to add in our `onContext3DCreate()` function. We simply need to set up the camera angle and instruct Flash to start the animation. We do this as follows:

```
// create projection matrix for our 3D scene  
projectionMatrix.identity();  
// 45 degrees FOV, 640/480 aspect ratio, 0.1=near, 100=far  
projectionMatrix.perspectiveFieldOfViewRH(  
    45.0, swfWidth / swfHeight, 0.01, 100.0);  
  
// create a matrix that defines the camera location  
viewMatrix.identity();  
// move the camera back a little so we can see the mesh  
viewMatrix.appendTranslation(0,0,-4);  
  
// start animating  
addEventListener(Event.ENTER_FRAME,enterFrame);  
}
```

What just happened?

A set of matrices are defined that are used by your shader to calculate the proper viewing angle of your mesh, as well as the specifics related to the camera, such as the field of a view (how zoomed in the camera is) and the aspect ratio of the scene.

Last but not the least; now that everything is set up, an event listener is created that runs the `enterFrame` function every single frame. This is where our animation will take place.

That is it for the Stage3D setup. We are done programming the `onContext3DCreate` function.

Time for action – let's animate

The `enterFrame` function is run every frame, over and over, during the course of your game. This is the perfect place to change the location of your meshes, trigger sounds, and perform all game logic.

```
private function enterFrame(e:Event):void  
{  
    // clear scene before rendering is mandatory  
    context3D.clear(0,0,0);  
  
    context3D.setProgram ( shaderProgram );
```

```
// create the various transformation matrices
modelMatrix.identity();
modelMatrix.appendRotation(t*0.7, Vector3D.Y_AXIS);
modelMatrix.appendRotation(t*0.6, Vector3D.X_AXIS);
modelMatrix.appendRotation(t*1.0, Vector3D.Y_AXIS);
modelMatrix.appendTranslation(0.0, 0.0, 0.0);
modelMatrix.appendRotation(90.0, Vector3D.X_AXIS);

// rotate more next frame
t += 2.0;

// clear the matrix and append new angles
modelViewProjection.identity();
modelViewProjection.append(modelMatrix);
modelViewProjection.append(viewMatrix);
modelViewProjection.append(projectionMatrix);

// pass our matrix data to the shader program
context3D.setProgramConstantsFromMatrix(
    Context3DProgramType.VERTEX,
    0, modelViewProjection, true );
```

What just happened?

In the preceding code, we first clear the previous frame from the screen. We then select the shader (program) that we defined in the previous function, and set up a new `modelMatrix`. The `modelMatrix` defines the location in our scene of the mesh. By changing the position (using the `appendTranslation` function), as well as the rotation, we can move our mesh around and spin it to our heart's content.

Time for action – setting the render state and drawing the mesh

Continue adding to the `enterFrame()` function by instructing Stage3D which mesh we want to work with and which texture to use.

```
// associate the vertex data with current shader program
// position
context3D.setVertexBufferAt(0, vertexBuffer, 0,
    Context3DVertexBufferFormat.FLOAT_3);
// tex coord
context3D.setVertexBufferAt(1, vertexBuffer, 3,
    Context3DVertexBufferFormat.FLOAT_3);

// which texture should we use?
```

Fire up the Engines!

```
    context3D.setTextureAt(0, myTexture);

    // finally draw the triangles
    context3D.drawTriangles(indexBuffer, 0, meshIndexData.length/3);

    // present/flip back buffer
    context3D.present();
}
```

What just happened?

Once you have moved objects around and prepared everything for the next frame (by instructing Stage3D which vertex buffer to draw and what texture to use), the new scene is rendered by calling `drawTriangles` and is finally presented on the screen.

In the future, when we have a more complex game, there will be more than one mesh, with multiple calls to `drawTriangles` and with many different textures being used. For now, in this simple demo, all we do each frame is spin the mesh around a little and then draw it.

Quest complete – time to reap the rewards

Now that the entire source code is complete, publish your .SWF file. Use your web browser to view the published HTML file. You should see something similar to the following:



If you see nothing on the screen when you view the HTML file that contains your new SWF, then your Flash incubator plugin is probably not being used. With fingers crossed, you will see a fully 3D textured square spinning around in space. Not much to look at yet, but it proves that you are running in the hardware accelerated 3D mode.

Congratulations!

You have just programmed your first Flash 11 Stage3D (Molehill) demo! It does not do much, but already you can see the vast possibilities that lay ahead of you. Instead of a simple square spinning around, you could be rendering castles, monsters, racing cars, or spaceships, along with all the particle systems, eye-candy, and special effects you could imagine. All this and more is waiting for you in the next chapters.

For now, be very proud that you have overcome the hardest part—getting a working 3D demo that compiles. Many before you have tried and failed, either because they did not have the proper version of Flash, or did not have the correct tools setup, or finally could not handle the complex AS3 source code required.

The fact that you made it this far is a testament to your skill and coding prowess. You deserve a break for now. Rest easy in the satisfaction, as you just reached a major milestone toward the goal of creating an amazing 3D game.

The entire source code

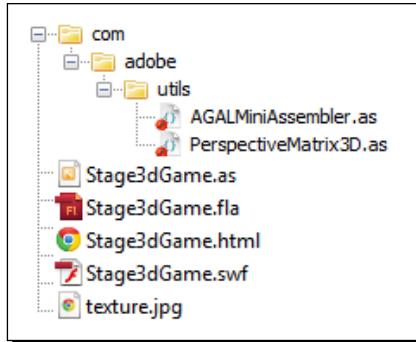
All the code that you entered earlier should go in a file named `stage3dGame.as` alongside your other project files. For reference, or to save typing, all source and support files are available at the following URL:

http://www.mcfunkypants.com/molehill/chapter_3_source_code.zip

The final demo can be run from the following URL:

<http://www.packtpub.com/adobe-flash11-stage3d-molehill-game-programming-beginners-guide/book>

Your folder structure should look similar to the one shown in the following screenshot. You might have used different file names for your html files or texture, but this screenshot may be helpful to ensure you are on the right track:



Pop quiz – earn some experience points

Here are a few quick questions to test your understanding. If you can answer them, then you are officially ready to move on to the next step in your grand adventure.

1. What class is used to compile vertex and fragment programs?
 - a. Context3D
 - b. AGALMiniAssembler
 - c. Vector3D
 - d. I always fly first class
2. Why do we need to wait for an event to fire before we send data to Stage3D?
 - a. The Context3D is not available immediately after requesting it
 - b. The event might fire more than once requiring data to be re-sent during the gameplay
 - c. To prevent Flash from freezing or locking up during initialization
 - d. All of the above
3. What is MIP mapping?
 - a. A special AGAL shader command
 - b. Rendering the same mesh more than once
 - c. Creating successively smaller versions of the same texture to increase the graphics quality
 - d. A way to avoid getting lost

Have a go hero – a fun side quest

In order to really hone your skills, it can be a good idea to challenge yourself for some extra experience. At the end of each chapter, there is a side quest with which you can experiment. It is completely optional. Just like grinding in an RPG game, challenges such as these are designed to make you stronger, so that when you forge ahead, you are more than ready for the next step in your main quest.

Your side quest this time is to experiment with the mesh data. Doing so will help you understand what each number stored in the `meshVertexData` variable (which is defined in the `initData` function) means.

Play with all these numbers. Notice that each line of eight numbers is the data used for one vertex. See what happens if you give them crazy values.

For example, if you change the first vertex position to -3, your square will change shape and will become lopsided with one corner sticking out:

```
meshVertexData = Vector.<Number>
(
    [
        //X, Y, Z, U, V, nX, nY, nZ
        -3, -1, 1, 0, 0, 0, 0, 1,
        1, -1, 1, 1, 0, 0, 0, 1,
        1, 1, 1, 1, 1, 0, 0, 1,
        -1, 1, 1, 0, 1, 0, 0, 1
    ]
);
```

If you tweak the U or V texture coordinates, then you can make the rock texture more zoomed in or tiled multiple times. If you change the vertex normals (the last three numbers of each line above), then nothing will happen. Why? The reason is that the ultra-simplistic shader that we are using does not use normal data.

If all this seems a bit low level, then do not worry. Eventually, you won't be defining mesh data by hand using lists of numbers entered in the code: in the near future, we will upgrade our game engine to parse the model data that we can export from a 3D art program, such as 3D Studio Max, Maya, or Blender.

For now, however, this simple "side quest" is a great way to start becoming familiar with what a vertex buffer really is.

Summary

We were victorious in our first "boss battle" along the way to the creation of our own 3D video game in Flash. It was tricky, but somehow we managed to achieve the following milestones: we learned how to obtain Flash 11 for our browser, we got all the tools ready to compile Stage3D games, we learned how to initialize the Stage3D API, how to upload mesh and texture data to the video card, and how to animate a simple 3D scene.

Now that we have created a simple template 3D demo, we are ready to add more complexity to our project. What was merely a tech demo will soon grow into a fully-fledged video game.

Level 3 achieved!

You have successfully "leveled up" for a third time. As a level three Molehill master, you have earned the right to start designing your own complex and beautiful shaders using AGAL, which is the topic of the next chapter.

4

Basic Shaders: I can see Something!

Welcome to Molehill Level 4.

You have graduated from basic training. Your skills are growing stronger. It is time to learn all about AGAL, the innovative power behind the Stage3D API. Sharpen your blade and get ready for battle!

In the previous chapter, we glossed over the commands that were used to render your simple mesh and apply a texture to it. We are now going to explore the depths of this dungeon called shader programming. Consider this chapter a treasure map that leads the way toward mastery of shader programming.

AGAL stands for **Adobe Graphics Assembly Language**.

Like ancient runes, AGAL looks unfamiliar at first glance, but contains untold power once understood. It is very similar to the machine language low level code that runs directly on your video card. It is almost exactly the same as how your video card "thinks".

AGAL may at first look like a secret code from an ancient spellbook, but once you play around with it, you will find it to be much simpler than expected. The great news? It is small, compact, and uncomplicated. There are only about 30 commands.

The challenge? It is so low level that it might be intimidating at first. Don't let it scare you off!

Skills in creating fragment and vertex programs are at the heart of 3D rendering in Flash 11, and the few who are courageous enough to learn how to create shaders will be in demand. Knowledge of AGAL is gold on a resume. AGAL means business. AGAL means speed.

Take heart in the fact that this is exactly the kind of knowledge that separates the amateurs from the professionals. Learning AGAL is a true test of mettle. Learn AGAL and you can officially call yourself a Molehill master.

In this chapter, we are going to learn the following:

- ◆ The basics of AGAL—Adobe Graphics Assembly Language
- ◆ Compiling a vertex program and a fragment program
- ◆ Combining these two programs to create a shader
- ◆ The different kinds of AGAL registers
- ◆ Sending data to your shaders to animate over time
- ◆ Rendering multiple instances of the same mesh
- ◆ Overlaying Flash 2D text over the top of a 3D scene

AGAL: Adobe Graphics Assembly Language

The term used to define how something is rendered in 3D is typically called a "shader". In the Stage3D API, shaders are constructed from very basic building blocks. As we learned in *Chapter 2, Blueprint of a Molehill*, a shader is the combined output of a vertex program and a fragment program working as a team.

As the term "shader" is used so commonly in 3D graphics, many people will call vertex programs by the equivalent term, "vertex shaders". Similarly, fragment programs are often interchangeably referred to as "pixel shaders".

It can sometimes be confusing to use more than one term to describe the same thing, so for the purpose of clarity, we will avoid using the word shader and stick to vertex and fragment programs. This matches the function and class names that are used in your source code.

Vertex and fragment programs are incredibly fast because they are performed by your video card (GPU)—not your CPU. This means that while these functions execute, your Flash game can be thinking about other things.

When writing programs, remember that each is run millions of times per frame: for example, a vertex program gets run once for each and every vertex in your mesh! A fragment shader runs even more often: once for every pixel.

Although it may seem intimidating at first, like any boss battle, once you know what to expect, it is a piece of cake. Just memorize the movements of the boss and aim for the glowing red parts. You can do it!

What does one line of AGAL look like?

<opcode> <destination> <source 1> <source 2>

This means that for each line of code, the first token (chunk) will be what command or function is going to run. This command or function is called the "opcode". The second token is the "destination" where the answer should go. Subsequent tokens list the source data locations—the parameters used by the command being executed.

For example:

```
mov v0, va1
```

In this example, the value stored in `va1` is copied to `v0`.

Unlike AS3 or C++ which contain thousands of functions, AGAL only uses about 30 different opcodes. The complete list of opcodes can be found in the Stage3D documentation that you downloaded in the previous chapter, as well as in the appendix. In order to get a taste of AGAL commands, here are some very common ones:

```
mov (copies data from source1 to destination)
add (destination = source1 plus source2)
sub (destination = source1 minus source2)
mul (destination = source1 multiplied by source2)
div (destination = source1 divided by source2)
```

Naturally, some of the other commands are more complex—many perform complicated calculus functions that are used very often in game programming, such as dot products and matrix multiplication.

Although you do not need to memorize them, for your convenience there is a complete listing of AGAL opcodes in the appendix.

What is a register?

The "source" and "destinations" in the preceding examples are called registers. You can think of them as variables; locations used to store data. They represent tiny chunks of video RAM (VRAM), optimized for speed.

In AGAL, in order to be blazingly fast, there is a limited supply of registers. Once you dive into the creation of more complex shaders, you will start to run out of them and will have to come up with creative ways to reuse registers because they are in such limited supply.

As opposed to an AS3 variable, which could be any kind of data type from a String to a Vector3D to a Bitmap, registers are all 128 bits wide and contain 4 floating point values. A floating point value is called Number in AS3, meaning it can have a fractional component, such as 12.75 or 3.14.

What is a component?

Each of these four values in a register is called a component. You can access each of these components individually by using x,y,z,w or r,g,b,a. For example, va1.x simply refers to the first component in the register va1. So does va1.r, they mean the same thing: the first component in the register.

It is up to you which naming convention to use, xyzw or rgba, it does not matter. Typically, it is a good practice to use x,y,z,w in vertex programs and r,g,b,a in fragment shaders, for obvious reasons.

Working with four components at the same time

Each opcode (such as `mov` or `add` or `sub`) performs an instruction (such as adding numbers) in a "component-wise" fashion. This means that it adds each register component together in order, component by component.

In the `add` opcode, for example, your video card will add `source1.x` to `source2.x`, `source1.y` to `source2.y`, `source1.z` to `source2.z`, and finally `source1.w` to `source2.w`. For example, if `source1` is a Vector3D with the value (10,100,50,0) and `source2` is (5,1,3,0), then the destination will equal (15,101,53,0).

Different registers for different jobs

There are several different types of register available. In AGAL code, they are referred to by a two-letter token followed by a number when there is more than one. For example, there are eight `va` registers, named `va0`, `va1`, `va2`, `va3`, `va4`, `va5`, `va6` and `va7`. In the preceding one line example, ("`mov v0, va1`") the contents of register `va1` were copied into `v0`, which is one of the eight varying (`v`) registers.

Registers used by vertex programs	Used by both	Registers used by fragment programs
<code>va = Vertex Attribute</code> <code>vc = Vertex Constant</code> <code>vt = Vertex Temporary</code> <code>op = Vertex Output Position</code>	Used by both <code>v=Varying</code>	<code>fc = Fragment Constant</code> <code>ft = Fragment Temporary</code> <code>fs = Fragment Texture Sampler</code> <code>oc = Fragment Output Color</code>

Vertex attribute registers: va0..va7

These registers refer to specific locations inside your VertexBuffer. They are only available in vertex shaders. When you upload all the data for your mesh, you can include x,y,z coordinates, as well as any number of other data. Common values include texture uv coordinates, normals, r,g,b colors and anything else you wish to assign to each vertex.

In order to assign a VertexBuffer to a specific attribute register, use the function `Context3D: setVertexBufferAt()` with the proper index. This function allows you to instruct Flash that, for example, the first three numbers in your byte array refer to x,y,z. Then, from the shader, access the attribute register with the syntax: `va<n>`, where `<n>` is the index number of the attribute register. In this example, `va0` will "point" to the x,y,z location of each vertex. There are a total of eight attribute registers available to vertex shaders. This gives you room for eight different registers (`va0`, `va1`, `va2`, `va3`, `va4`, `va5`, `va6`, and `va7`) for every vertex in your mesh.

Remember that each register holds four floating-point numbers, so you have room for 32 numbers (per vertex) which could be used for any purpose. You are free to include non-standard data here, which could be used in your vertex program in all sorts of creative ways. You could assign a vertex a weight for a particular "bone" used in animation, or a fog value, or some special constant to be used for special effects, for example.

Constant registers: vc0..vc127 and fc0..fc27

These registers serve the purpose of passing parameters from ActionScript to the shaders. This is done through the `Context3D: :setProgramConstants()` family of functions. These registers are accessed from the shader with the syntax: `vc<n>`, for vertex shaders, and `fc<n>` for pixel shaders, where `<n>` is the index number of the constant register.

There are 128 constant registers available to vertex shaders and 28 constant registers for pixel shaders. If you need to pass any numbers into your shader (for example, pi, or a light direction, or the color of fog in your game universe) this is what you would use.

As you have over a hundred registers to play with, each containing four numbers, there is a lot of room to define all sorts of data. These constant registers are not stored in your vertex buffer, however, so they are shared between all verteces or fragments. They are "global variables" for use in your programs in any way you wish.

Temporary registers: vt0..vt7 and ft0..ft7

These registers are available to either kind of program and are meant to be used for temporary calculations. They are accessed with the syntax `vt<n>` (vertex programs) and `ft<n>` (fragment programs) where `<n>` is the register number.

There are eight of these available for vertex shaders and eight for pixel shaders. These can be used to hold values for calculations later in the shader. They are uninitialized before the shader is run on each vertex, so you can think of them as local, temporary storage.

Output registers: op and oc

The output registers, `op` for output position and `oc` for output color, are where vertex and pixel shaders store the final result of their calculations. For vertex programs, this output is a clip space position of the vertex. For fragment programs, it is the color of the pixel. There is obviously only one output register for vertex and for pixel shaders. It holds the "final result" of all your calculations.

Varying registers: v0..v7

These registers are used to pass data from vertex shaders to pixel shaders. The data that is passed is properly interpolated by the GPU, so that the pixel shader receives the correct value for the pixel that is being processed.

Typical data that is passed in this way include the vertex color or the UV coordinates for texturing. These registers can be accessed with the syntax `v<n>`, where `<n>` is the register number. There are eight varying registers available.

For example, in your vertex program you might calculate the color of each vertex based on how far away it is from the camera to simulate fog. By passing these values to the fragment shader (one for each vertex), the fragment shader will interpolate the three colors from each of the three vertexes in any one polygon to smoothly transition from one color to the next. In this way, for example, if one vertex is bright red and its neighbor is blue, the pixels between the two will fade from red to purple to blue in the way you would expect, so that the triangle on-screen is tinted correctly.

The important point to remember here is that varying registers can return slightly different values for every single pixel on the screen: they are smoothly interpolated from one to the next with each iteration of the fragment program. The value received by the fragment shader is interpolated between the three vertexes making up a triangle.

Texture samplers: fs0..fs7

The eight-texture sampler registers are used to pick color values from textures, based on UV coordinates. This is a special register that follows its own set of rules, and is only used in a fragment program. The texture to be used is specified through ActionScript with the call `Context3D::setTextureAt()`.

For example, a standard 2D texture without mip mapping and linear filtering could be sampled into temporary register `ft1` with the following line of AGAL code. Let's assume that the varying register `v0` holds the interpolated texture UVs:

```
tex ft1, v0, fs0 <2d,linear,nomip>
```

The syntax for using texture sampler registers is: `ft<n><flags>`, where `<n>` is the sampler index, and `<flags>` is a set of one or more flags that specifies how the sampling should be made. `<flags>` is a comma separated set of strings, that defines:

1. **texture dimension:** `2d`, `3d`, or `cube`. A 2D texture is the most commonly used format, a rectangular bitmap. A 3D texture has length, width, and depth, and takes up a lot of texture RAM but is great for 3D materials such as wood, grain, or marble. It is rarely used. A cube texture is a specially encoded group of six rectangular bitmaps and is usually used for reflections, as each of these six images maps to a particular direction like the sides on the inside of a box.
2. **mip mapping:** `nomip`, `mipnone`, `mipnearest`, or `miplinear`. If your texture has mip maps (successively smaller versions of itself, generated to avoid jaggies during rendering), then you can instruct Stage3D to smoothly interpolate the texture using them. Mip mapped textures are very useful to increase render quality and avoid the "moiree effect" (flickering) when viewed from far away.
3. **texture filtering:** `nearest` or `linear`. If you prefer a retro, then pixilated look you can use `nearest` here to tell Stage3D not to interpolate the values smoothly, resulting in blocky textures when viewed up close, for example. If you use `linear`, when zoomed into a texture it will be blurry and will have smoother edges.
4. **texture repeat:** `repeat`, `wrap`, or `clamp`. If you want the texture tile properly, then you use `repeat` here. If you are not tiling the texture and run into problems with edge pixels being the color of the opposite edge (which often happens in transparent billboard particles and when rendering bitmap fonts in 3d), then specify `clamp` to force Stage3D not to blur adjacent edge pixels.

A basic AGAL shader example

We will create a shader using AGAL. A shader is the combination of a vertex and fragment program which is stored in the `program3D` class in Flash. For this example, we are going to a minimalistic vertex program, as well as a simple fragment program.

The vertex program

A very simple vertex program might perform the following steps:

1. Change the coordinates of each vertex to match the current camera angle.
2. Move this value to the "output position" AGAL register.
3. Send the UV texture coordinates to the fragment program.

Time for action – writing your first vertex program

Here is your first taste of AGAL: a simple vertex program.

```
// the simplest possible vertex program
m44 op, va0, vc0      // pos to clipspace
mov v0, va1           // copy uv
```

What just happened?

The two lines of the preceding AGAL code are an example of the simplest possible vertex program.

Remember that a vertex program is responsible for defining the x,y,z positions and attributes (such as r,g,b or uv texture coordinates) of each vertex in a mesh. In this example, there are just two commands that are run. They are run once for each vertex in your 3D model. For example, if your mesh is made up of 20,000 vertexes, the preceding vertex program is run 20,000 times.

The first line of code accomplishes the first two steps in the preceding *Time for action* section. It contains the operand `m44`, which is a 4x4-matrix multiplication: meaning a four component Vector3D stored in the register named `va0` (the position of the vertex) is multiplied by a 4x4 Matrix3D in `vc0` (which defines the location of the mesh in relation to the camera).

The result of the calculation is stored in `op`, which stands for output position.

The second line of code in the preceding example uses the operand `mov`, which moves (copies) the value stored in `va1` (the uv texture coordinates of that particular vertex as defined by our vertex buffer) to `v0`, which is the vertex program output register. This value will be used by our fragment program when it looks up what color each pixel should be, by finding the uv coordinate in that texture. So, how does Flash know where the model is in relation to the camera angle, which in the preceding vertex program is accessed in the `vc0` register? This information is stored in a Matrix3D object that we send to the shader before rendering. Later in this chapter, we will use the `setProgramConstantsFromMatrix` function to instruct Flash to store our matrix in the `vc0` vertex constant register for use in our shader.

The contents of this matrix represent the location, on the screen, of our mesh. In subsequent chapters, we will get into how to calculate this value, but for now, just remember that it involves taking the world coordinates of the model and appending those of the camera in such a way that it instructs Stage3D where it is in "clip space" (on screen). To put it simply, you can use any coordinate system you wish in your game, but when it comes to rendering your mesh, your vertex program has to map whatever x,y,z coordinates you used for each vertex to fit inside the area that gets rendered on the screen, which uses internal coordinates from 0 to 1. Even if, in your game, you are using a giant scale and have buildings that are 10,000 units tall—or even a million—once rendered, they will all be converted to fit within the clip space.

In the preceding vertex program, the other two registers that were referred to are the vertex positions and the texture coordinates. We will use the `setVertexBufferAt` function to define `va0` (the vertex positions) and `va1` (the vertex UV coordinates). Later in this chapter, we will look into how to send this data to your shaders. When we upload our mesh vertex buffer to the video card, we will also include some code that instructs Stage3D which chunks of the array are the x,y,z coordinates and which are the uv texture coordinates.

The two lines of the preceding AGAL code are all that is required for our vertex program.

The fragment program

The next and final step in creating a shader is to create the fragment program. Just as a vertex program deals with the vertices in a mesh, fragment programs do the work of deciding what each pixel on the screen needs to look like.

A simple fragment program that uses a texture should simply output a pixel color for rendering by following these steps:

- ◆ sample a texture using the UV coordinates output by the vertex program
- ◆ copy this color to the "output color" AGAL register

Time for action – writing your first fragment program

This is an example of a very simple fragment program:

```
// a simple fragment program
tex ft1, v0, fs0 <2d,linear,nomip>
mov oc, ft1
```

What just happened?

Just as vertex programs are run once for every single vertex in your mesh, fragment programs are run for each and every pixel on the screen that are part of your mesh. This means that one fragment program will often be run a million times per frame.

In this example, the opcode `tex` performs a texture look-up. It samples the texture in `fs0` that would be defined in the ActionScript using the `setTextureAt` function. We will explain how later on.

The uv coordinates that were stored in `v0` by your vertex program above are used to look-up the nearest pixel in that texture and are interpolated in a linear way without the use of mipmaps to arrive at an r,g,b,a color for the fragment in question. It then copies this value to `oc`, which stands for output color.

Compiling the AGAL source code

Now that we have written a vertex program and a fragment program, we can use Adobe's `AGALMiniAssembler` class to compile our AGAL assembly language into bytecode that is uploaded to your video card. The bytecode closely matches the format and order of your assembly language commands except that all comments are stripped out and the string opcodes (such as `add`) are replaced by their numerical equivalents (0x01). The result of compiling AGAL into bytecode is just a buffer filled with binary data.

In order to compile the AGAL source code, you simply:

1. Create a new `AGALMiniAssembler` object
2. Tell it to compile your AGAL source
3. Combine compiled vertex and fragment programs to make a shader
4. Upload this for use during rendering

Time for action – compiling AGAL

In order to accomplish the preceding four steps, we would use something like the following:

```
var vertexShaderAssembler:AGALMiniAssembler =
    new AGALMiniAssembler();

vertexShaderAssembler.assemble(
    Context3DProgramType.VERTEX,
    "m44 op, va0, vc0\n" +
    "mov v0, va1"
);
```

```
var fragmentShaderAssembler:AGALMiniAssembler =
    new AGALMiniAssembler();

fragmentShaderAssembler.assemble(
    Context3DProgramType.FRAGMENT,
    "tex ft1, v0, fs0 <2d,linear, nomip>;\n" +
    "mov oc, ft1"
);

var program:Program3D =
    context3D.createProgram();

program.upload(
    vertexShaderAssembler.agalcode,
    fragmentShaderAssembler.agalcode
);
```

What just happened?

In the preceding example, we created a new `AGALMiniAssembler` object for each of the two programs required. As you can see, the AGAL source code we wrote is sent to the assembler as a string. This `AGALMiniAssembler` object is then instructed to assemble the AGAL source into binary data ready to be used by Stage3D as a shader.

Finally, we upload the two compiled programs to Stage3D. One is the compiled vertex program and the other is your compiled fragment program. These two programs are combined and uploaded to your video card, ready for use.

Time to Render!

You only have to compile your AGAL code once, during the initialization of your game. After that, you simply instruct the Stage3D API which program you want to use for which mesh. In order to do this, carry out the following actions:

1. Clear the Context3D, erasing the previous frame
2. Select which vertex buffers to use
3. Select which texture to use
4. Set the current shader program
5. Apply a matrix for use in your shaders to calculate the location
6. Render the mesh
7. Repeat steps 2 through 6 for every mesh
8. Present the final image to the user

Time for action – rendering

The following is an example of the AS3 code you could use to render your scene once for every frame that follows the preceding steps:

```
// clear your viewport to the background color:  
context3D.clear ( 1, 1, 1, 1 );  
// vertex position to attribute register 0 (va0)  
context3D.setVertexBufferAt (0, vertexbuffer, 0,  
    Context3DVertexBufferFormat.FLOAT_3);  
// uv coordinates to attribute register 1 (va1)  
context3D.setVertexBufferAt(1, vertexbuffer, 3,  
    Context3DVertexBufferFormat.FLOAT_2);  
// assign texture to texture sampler 0 (fs0)  
context3D.setTextureAt( 0, texture );  
// assign shader program  
context3D.setProgram( program );  
// Note how the two vertex buffer streams,  
// vertex positions and uv coordinates,  
// get assigned to two different attribute registers.  
// The texture is assigned to a specific texture sampler.  
// Pass on your transform matrix to the shader.  
var m:Matrix3D = new Matrix3D();  
m.appendRotation(getTimer()/50, Vector3D.Z_AXIS);  
context3D.setProgramConstantsFromMatrix(  
    Context3DProgramType.VERTEX, 0, m, true);  
// The matrix gets stored into constant register 0 here,  
// for the Vertex Shader to use.  
// To render, call drawTriangles passing in the index buffer.  
context3D.drawTriangles(indexBuffer);  
// You may do additional drawTriangle calls in the frame  
// one for each mesh you want rendered.  
// You can use the same or different programs for each meshes.  
// Once all meshes have been drawn, display them on screen:  
context3D.present();
```

What just happened?

In the preceding example, after clearing the screen, we instruct Flash about parts of the mesh vertex buffer to use for two registers. The first `setVertexBufferAt` assigns the first three numbers in the vertex buffer to be readable in our vertex program as `va0`. Here, they are the x,y,z coordinates of the vertex. The next line instructs Flash that the next two numbers in the buffer should be accessible as `va1`. In this case, they will be the uv texture coordinates of the vertex.

Next, we instruct Flash which texture to use with the `setTextureAt` command. It will be available in our fragment shader as `f$0`. Finally, we call the `setProgram` function to assign the assembled AGAL program we created in the previous example.

Now that everything is set up and ready to render, we create a `Matrix3D` to hold the rotation of our mesh and make it spin around the z-axis over time. The `setProgramConstantsFromMatrix` function assigns this new position and rotation to `vc0`, so that our vertex program can access it.

Lastly, we call the `drawTriangles` function to instruct Flash to render the mesh stored in the variable named `indexBuffer`, using the shader program and constants defined earlier. The `present()` function instructs Flash that we are done rendering for this frame, so now it can present the final image to the user on the screen.

Creating a shader demo

We will put all that we have learned thus far into practice. When we are done, we will have a new improved demo that shows off four different shaders, as well as a GUI that displays the frame rate and descriptions of each shader, as shown in the following screenshot:



You can view the demo at the following URL:

http://www.mcfunkypants.com/molehill/chapter_4_demo/

Open the example Stage3dGame project that we created in the previous chapter and start making a few modifications. Let's upgrade our template by using all the experience we just gained.

Although you will learn more if you type the code as you read this chapter, if you prefer to passively follow along, you can download the complete source code for the project at the following URL:

<http://www.packtpub.com/adobe-flash11-stage3d-molehill-game-programming-beginners-guide/book>

Adding an FPS counter

Firstly, it would be nice to be able to display the fantastic performance that we are getting with Flash 11's hardware accelerated graphics horsepower. Let's add an FPS counter that will display how many frames per second we are getting.

Time for action – creating the FPS GUI

Near the top of the .as file, make a few changes as follows in order to use our new GUI so that our demo is capable of displaying more than one shader at once:

```
package
{
    [SWF(width="640", height="480", frameRate="60",
        backgroundColor="#FFFFFF")]

    import com.adobe.utils.*;
    import flash.display.*;
    import flash.display3D.*;
    import flash.display3D.textures.*;
    import flash.events.*;
    import flash.geom.*;
    import flash.utils.*;
    import flash.text.*;

    public class Stage3dGame extends Sprite
    {
        // used by the FPS display
        private var fpsLast:uint = getTimer();
```

```
private var fpsTicks:uint = 0;
private var fpsTf:TextField;

// constants used during inits
private const swfWidth:int = 640;
private const swfHeight:int = 480;
private const textureSize:int = 512;

// the 3d graphics window on the stage
private var context3D:Context3D;
// the compiled shaders used to render our mesh
private var shaderProgram1:Program3D;
private var shaderProgram2:Program3D;
private var shaderProgram3:Program3D;
private var shaderProgram4:Program3D;
// the uploaded vertexes used by our mesh
private var vertexBuffer:VertexBuffer3D;
// the uploaded indexes of each vertex of the mesh
private var indexBuffer:IndexBuffer3D;
// the data that defines our 3d mesh model
private var meshVertexData:Vector.<Number>;
// the indexes that define what data is used by each vertex
private var meshIndexData:Vector.<uint>;

// matrices that affect the mesh location and camera angles
private var projectionmatrix:PerspectiveMatrix3D =
    new PerspectiveMatrix3D();
private var modelmatrix:Matrix3D = new Matrix3D();
private var viewmatrix:Matrix3D = new Matrix3D();
private var modelViewProjection:Matrix3D = new Matrix3D();

// a simple frame counter used for animation
private var t:Number = 0;
// a reusable loop counter
private var looptemp:int = 0;

/* TEXTURE: Pure AS3 and Flex version:
 * if you are using Adobe Flash CS5
 * comment out the next two lines of code */
[Embed (source = "texture.jpg")]
private var myTextureBitmap:Class;
private var myTextureData:Bitmap = new myTextureBitmap();

/* TEXTURE: Flash CS5 version:
```

```
* add the jpg to your library (F11)
* right click it and edit the advanced properties so
* it is exported for use in Actionscript
* and call it myTextureBitmap
* if using Flex/FlashBuilder/FlashDevelop
* comment out the following lines of code */
//private var myBitmapDataObject:myTextureBitmapData =
//    //new myTextureBitmapData(textureSize, textureSize);
//private var myTextureData:Bitmap =
//    //new Bitmap(myBitmapDataObject);

// The Stage3d Texture that uses the above myTextureData
private var myTexture:Texture;
```

What just happened?

In the preceding example, we are simply adding four lines of code to our existing project from the previous chapter. These will give our program the ability to display text on the screen over the top of our 3D graphics.

We define a few new variables, but otherwise, everything is the same as before. Remember that you will use either the pure AS3 "embed" version or the CS5 library version of the code that includes your texture depending in which environment you are programming, same as before.

Time for action – adding the GUI to our inits

Now let's modify our `inits` to enable our fancy new GUI. Tweak the `init` code from the previous chapter as follows. You will find that most of the following code has not changed:

```
public function Stage3dGame()
{
    if (stage != null)
        init();
    else
        addEventListener(Event.ADDED_TO_STAGE, init);
}

private function init(e:Event = null):void
{
    if (hasEventListener(Event.ADDED_TO_STAGE))
        removeEventListener(Event.ADDED_TO_STAGE, init);

    stage.scaleMode = StageScaleMode.NO_SCALE;
```

```
stage.align = StageAlign.TOP_LEFT;

// add some text labels
initGUI();

// and request a context3D from Stage3d
stage.stage3Ds[0].addEventListener(
    Event.CONTEXT3D_CREATE, onContext3DCreate);
stage.stage3Ds[0].requestContext3D();

}

private function initGUI():void
{
    // a text format descriptor used by all gui labels
    var myFormat:TextFormat = new TextFormat();
    myFormat.color = 0xFFFFFFFF;
    myFormat.size = 13;

    // create an FPSCounter that displays the framerate on screen
    fpsTf = new TextField();
    fpsTf.x = 0;
    fpsTf.y = 0;
    fpsTf.selectable = false;
    fpsTf.autoSize = TextFieldAutoSize.LEFT;
    fpsTf.defaultTextFormat = myFormat;
    fpsTf.text = "Initializing Stage3d...";
    addChild(fpsTf);

    // add some labels to describe each shader

    var label1:TextField = new TextField();
    label1.x = 100;
    label1.y = 180;
    label1.selectable = false;
    label1.autoSize = TextFieldAutoSize.LEFT;
    label1.defaultTextFormat = myFormat;
    label1.text = "Shader 1: Textured";
    addChild(label1);

    var label2:TextField = new TextField();
    label2.x = 400;
    label2.y = 180;
    label2.selectable = false;
```

```
label2.autoSize = TextFieldAutoSize.LEFT;
label2.defaultTextFormat = myFormat;
label2.text = "Shader 2: Vertex RGB";
addChild(label2);

var label3:TextField = new TextField();
label3.x = 80;
label3.y = 440;
label3.selectable = false;
label3.autoSize = TextFieldAutoSize.LEFT;
label3.defaultTextFormat = myFormat;
label3.text = "Shader 3: Vertex RGB + Textured";
addChild(label3);

var label4:TextField = new TextField();
label4.x = 340;
label4.y = 440;
label4.selectable = false;
label4.autoSize = TextFieldAutoSize.LEFT;
label4.defaultTextFormat = myFormat;
label4.text = "Shader 4: Textured + setProgramConstants";
addChild(label4);
}
```

What just happened?

In the `init` function, the start-up code remains the same except we add a line that runs the new `initGUI` function we just wrote. This function creates a few `TextField` objects that will be overlaid on top of our `Context3D`.

Time for action – adding multiple shaders to the demo

Lets modify our `onContextCreate()` function to use multiple shaders and to use a new vertex buffer that includes RGB data as well. Most of the following lines are unchanged from last time:

```
private function onContext3DCreate(event:Event):void
{
    // Remove existing frame handler. Note that a context
    // loss can occur at any time which will force you
    // to recreate all objects we create here.
    // A context loss occurs for instance if you hit
    // CTRL-ALT-DELETE on Windows.
    // It takes a while before a new context is available
```

```
// hence removing the enterFrame handler is important!
removeEventListener(Event.ENTER_FRAME,enterFrame);

// Obtain the current context
var t:Stage3D = event.target as Stage3D;
context3D = t.context3D;

if (context3D == null)
{
    // Currently no 3d context is available (error!)
    return;
}

// Disabling error checking will drastically improve performance.
// If set to true, Flash sends helpful error messages regarding
// AGAL compilation errors, uninitialized program constants, etc.
context3D.enableErrorChecking = true;

// Initialize our mesh data
initData();

// The 3d back buffer size is in pixels
context3D.configureBackBuffer(swfWidth, swfHeight, 0, true);

// assemble all the shaders we need
initShaders();

// upload the mesh indexes
indexBuffer = context3D.createIndexBuffer(meshIndexData.length);
indexBuffer.uploadFromVector(
    meshIndexData, 0, meshIndexData.length);

// upload the mesh vertex data
// since our particular data is
// x, y, z, u, v, nx, ny, nz, r, g, b, a
// each vertex uses 12 array elements
vertexBuffer = context3D.createVertexBuffer(
    meshVertexData.length/12, 12);
vertexBuffer.uploadFromVector(meshVertexData, 0,
    meshVertexData.length/12);
```

```
// Generate mipmaps
myTexture = context3D.createTexture(textureSize, textureSize,
    Context3DTextureFormat.BGRA, false);
var ws:int = myTextureData.bitmapData.width;
var hs:int = myTextureData.bitmapData.height;
var level:int = 0; var tmp:BitmapData;
var transform:Matrix = new Matrix();
tmp = new BitmapData(ws, hs, true, 0x00000000);
while ( ws >= 1 && hs >= 1 ) {
    tmp.draw(myTextureData.bitmapData, transform, null, null,
        null, true);
    myTexture.uploadFromBitmapData(tmp, level);
    transform.scale(0.5, 0.5); level++; ws >>= 1; hs >>= 1;
    if (hs && ws) {
        tmp.dispose();
        tmp = new BitmapData(ws, hs, true, 0x00000000);
    }
}
tmp.dispose();
}

// create projection matrix for our 3D scene
projectionmatrix.identity();
// 45 degrees FOV, 640/480 aspect ratio, 0.1=near, 100=far
projectionmatrix.perspectiveFieldOfViewRH(
    45.0, swfWidth / swfHeight, 0.01, 100.0);

// create a matrix that defines the camera location
viewmatrix.identity();
// move the camera back a little so we can see the mesh
viewmatrix.appendTranslation(0,0,-10);

// start animating
addEventListener(Event.ENTER_FRAME,enterFrame);
}
```

What just happened?

As you can see, only a few lines in the preceding code have changed since last time. Firstly, we are calling a new function that inits all the shaders, and our vertex buffer now holds 12 numbers per vertex rather than 8 as in our previous version of the demo.

Time for action – initializing the shaders

Now we need to initialize all the shaders for our demo:

```
// create four different shaders
private function initShaders():void
{
    // A simple vertex shader which does a 3D transformation
    // for simplicity, it is used by all four shaders
    var vertexShaderAssembler:AGALMiniAssembler =
        new AGALMiniAssembler();
    vertexShaderAssembler.assemble
    (
        Context3DProgramType.VERTEX,
        // 4x4 matrix multiply to get camera angle
        "m44 op, va0, vc0\n" +
        // tell fragment shader about XYZ
        "mov v0, va0\n" +
        // tell fragment shader about UV
        "mov v1, va1\n" +
        // tell fragment shader about RGBA
        "mov v2, va2\n"
    );
}
```

The first part of our new improved `initShaders` function defines a single vertex program that will be shaded by all four shaders. Each will use a different fragment program, which we define next.

```
// textured using UV coordinates
var fragmentShaderAssembler1:AGALMiniAssembler
    = new AGALMiniAssembler();
fragmentShaderAssembler1.assemble
(
    Context3DProgramType.FRAGMENT,
    // grab the texture color from texture 0
    // and uv coordinates from varying register 1
    // and store the interpolated value in ft0
    "tex ft0, v1, fs0 <2d,repeat,miplinear>\n" +
    // move this value to the output color
    "mov oc, ft0\n"
);
```

The first fragment program is the one from our earlier examples. It draws the mesh using a texture.

```
// no texture, RGBA from the vertex buffer data
var fragmentShaderAssembler2:AGALMiniAssembler
    = new AGALMiniAssembler();
fragmentShaderAssembler2.assemble
(
    Context3DProgramType.FRAGMENT,
    // grab the color from the v2 register
    // which was set in the vertex program
    "mov oc, v2\n"
);
```

The second fragment program, the preceding one, does not use a texture at all. Instead, it draws the mesh using the vertex colors that are stored in the mesh data. The colors for each pixel will be smoothly blended from the RGBA value of each vertex. Therefore, if one vertex is red and a neighboring one is yellow, then the pixels in between the two will include shades of orange.

```
// textured using UV coordinates AND colored by vertex RGB
var fragmentShaderAssembler3:AGALMiniAssembler
    = new AGALMiniAssembler();
fragmentShaderAssembler3.assemble
(
    Context3DProgramType.FRAGMENT,
    // grab the texture color from texture 0
    // and uv coordinates from varying register 1
    "tex ft0, v1, fs0 <2d,repeat,miplinear>\n" +
    // multiply by the value stored in v2 (the vertex rgb)
    "mul ft1, v2, ft0\n" +
    // move this value to the output color
    "mov oc, ft1\n"
);
```

The third fragment program in the `initShaders` function is a combination of the first two. This one uses a texture that will be blended with the vertex colors. The pixels that are drawn will be a mix of the colors in the texture bitmap and whatever color each vertex has been set to.

```
// textured using UV coordinates and
// tinted using a fragment constant
var fragmentShaderAssembler4:AGALMiniAssembler
    = new AGALMiniAssembler();
```

```
fragmentShaderAssembler4.assemble
(
    Context3DProgramType.FRAGMENT,
    // grab the texture color from texture 0
    // and uv coordinates from varying register 1
    "tex ft0, v1, fs0 <2d,repeat,miplinear>\n" +
    // multiply by the value stored in fc0
    "mul ft1, fc0, ft0\n" +
    // move this value to the output color
    "mov oc, ft1\n"
);
```

The final fragment program in our demo will use the texture and will be blended with the color that will dynamically change at runtime. Before each frame, we will change this color, so that our mesh seems to pulse, like the lights on top of a police car. We will store a different color in the fragment constant `fc0` for each frame:

```
// combine shaders into a program which we then upload to the GPU
shaderProgram1 = context3D.createProgram();
shaderProgram1.upload(
    vertexShaderAssembler.agalcode,
    fragmentShaderAssembler1.agalcode);

shaderProgram2 = context3D.createProgram();
shaderProgram2.upload(
    vertexShaderAssembler.agalcode,
    fragmentShaderAssembler2.agalcode);

shaderProgram3 = context3D.createProgram();
shaderProgram3.upload(
    vertexShaderAssembler.agalcode,
    fragmentShaderAssembler3.agalcode);

shaderProgram4 = context3D.createProgram();
shaderProgram4.upload(
    vertexShaderAssembler.agalcode,
    fragmentShaderAssembler4.agalcode);
}
```

What just happened?

We just wrote the `initShaders()` function.

As this new improved demo will show off four shaders at the same time, in the preceding code we create four different fragment programs that are all run on the same mesh. As the vertex program is the same for each shader, we simply create one that we can reuse four different times. Once we have compiled and uploaded each of the four shader programs, Stage3D is ready to use them to render.

Each program is a simple example that hopefully scratches the surface of all the amazing shaders that will be possible. Shader 1 simply renders our mesh with a plain texture. Shader 2 does not use a texture at all but instead generates RGB colors by interpolating the RGB values stored in the vertex buffer. Shader 3 does both these things: it uses the texture but multiplies the texture color by the RGB values stored in the vertex buffer, effectively tinting the texture.

This technique can be used to add simple shading or lighting effects by baking in color data into each vertex. For example, in 3Ds max you can bake RGB colors into each vertex before exporting that shade for each polygon according to their brightness when lit. This is the easiest and simplest way to simulate complex lighting, and allows artists to design levels that use thousands of light sources "for free".

Finally, shader 4 presents our first "dynamic" shader. It uses the texture and also multiplies it by a fragment constant, stored in `fco`, which is calculated in ActionScript for each frame. This shader will smoothly "pulse" between red and yellow, much like a police car's lights. In future versions of our game, we could use this technique for all sorts of amazing dynamic shaders. For example, we could fade explosions in and out, flash certain models red when they are damaged and are about to die, and much more. For now, a simple pulsing color is the perfect example of a shader that changes every frame.

Time for action – animating the shaders

The next step is to animate our models and shaders every single frame. Modify our existing `enterFrame` function as follows.

```
private function enterFrame(e:Event):void
{
    // clear scene before rendering is mandatory
    context3D.clear(0,0,0);

    // rotate more next frame
    t += 2.0;

    // loop through each mesh we want to draw
```

```
for (looptemp = 0; looptemp < 4; looptemp++)
{
    // clear the transformation matrix to 0,0,0
    modelmatrix.identity();

    // each mesh has a different texture,
    // shader, position and spin speed
    switch(looptemp)
    {
        case 0:
            context3D.setTextureAt(0, myTexture);
            context3D.setProgram ( shaderProgram1 );
            modelmatrix.appendRotation(t*0.7, Vector3D.Y_AXIS);
            modelmatrix.appendRotation(t*0.6, Vector3D.X_AXIS);
            modelmatrix.appendRotation(t*1.0, Vector3D.Y_AXIS);
            modelmatrix.appendTranslation(-3, 3, 0);
            break;
        case 1:
            context3D.setTextureAt(0, null);
            context3D.setProgram ( shaderProgram2 );
            modelmatrix.appendRotation(t*-0.2, Vector3D.Y_AXIS);
            modelmatrix.appendRotation(t*0.4, Vector3D.X_AXIS);
            modelmatrix.appendRotation(t*0.7, Vector3D.Y_AXIS);
            modelmatrix.appendTranslation(3, 3, 0);
            break;
        case 2:
            context3D.setTextureAt(0, myTexture);
            context3D.setProgram ( shaderProgram3 );
            modelmatrix.appendRotation(t*1.0, Vector3D.Y_AXIS);
            modelmatrix.appendRotation(t*-0.2, Vector3D.X_AXIS);
            modelmatrix.appendRotation(t*0.3, Vector3D.Y_AXIS);
            modelmatrix.appendTranslation(-3, -3, 0);
            break;
        case 3:
            context3D.setProgramConstantsFromVector(
                Context3DProgramType.FRAGMENT, 0, Vector.<Number>
                ([ 1, Math.abs(Math.cos(t/50)), 0, 1 ]) );
            context3D.setTextureAt(0, myTexture);
            context3D.setProgram ( shaderProgram4 );
            modelmatrix.appendRotation(t*0.3, Vector3D.Y_AXIS);
            modelmatrix.appendRotation(t*0.3, Vector3D.X_AXIS);
            modelmatrix.appendRotation(t*-0.3, Vector3D.Y_AXIS);
            modelmatrix.appendTranslation(3, -3, 0);
            break;
    }
}
```

In order to begin the function, we clear the screen and the draw four different copies of the same mesh using each of the shaders we defined earlier:

```
// clear the matrix and append new angles
modelViewProjection.identity();
modelViewProjection.append(modelmatrix);
modelViewProjection.append(viewmatrix);
modelViewProjection.append(projectionmatrix);

// pass our matrix data to the shader program
context3D.setProgramConstantsFromMatrix(
    Context3DProgramType.VERTEX,
    0, modelViewProjection, true );
```

Above, we fill the AGAL register vc0 with our model view projection matrix:

```
// associate the vertex data with current shader program
// position
context3D.setVertexBufferAt(0, vertexBuffer, 0,
    Context3DVertexBufferFormat.FLOAT_3);
// tex coord
context3D.setVertexBufferAt(1, vertexBuffer, 3,
    Context3DVertexBufferFormat.FLOAT_2);
// vertex rgba
context3D.setVertexBufferAt(2, vertexBuffer, 8,
    Context3DVertexBufferFormat.FLOAT_4);

// finally draw the triangles
context3D.drawTriangles(
    indexBuffer, 0, meshIndexData.length/3);
}
```

In order to finish our four model loop, we instruct Stage3D which texture, shader, and mesh to use and then draw it. Finally, in the following code, we show the final rendering to the user and update our FPS counter:

```
// present/flip back buffer
// now that all meshes have been drawn
context3D.present();

// update the FPS display
fpsTicks++;
var now:uint = getTimer();
var delta:uint = now - fpsLast;
// only update the display once a second
if (delta >= 1000)
```

```

{
    var fps:Number = fpsTicks / delta * 1000;
    fpsTf.text = fps.toFixed(1) + " fps";
    fpsTicks = 0;
    fpsLast = now;
}
}

```

What just happened?

The `enterFrame` function in the preceding *Time for action* section is run every single frame and it is the function that does all the work during your game. Here, we are first clearing the screen and incrementing a variable that is used to spin the meshes around over time.

Even though we are rendering four meshes on the screen, we are using the same vertex buffer four times rather than defining four different meshes. This technique is very powerful and will be used in your game later on to great effect. Imagine, for example, if you wanted to render a dense, lush forest filled with trees. You could define a single detailed tree mesh once during your game's inits, and then use a loop such as the one here to render hundreds of copies of this tree mesh in various locations. This way, even though you are using very little VRAM to store a single model, your scene can place many copies of that mesh in your game world.

Through the creative use of shaders, you could even change the geometry of that single tree mesh—perhaps some trees are bent or even gracefully blowing in the wind. Perhaps each tree has a slightly different color—some darker or lighter, so that each looks more different. Additionally, by changing the matrix used by each mesh, you could place trees in different locations, at different angles, and with different scales, so that some trees are smaller and fatter or taller and thinner than others are. Your shaders could even gradually tint the green leaves of the trees red in autumn.

In the `select...case` statement, each of the four shaders is applied to the mesh, which is given a different position and rotation by changing how the `modelmatrix` is set. Some of the shaders use a texture, while one does not. The fourth shader also uses the `setProgramConstantsFromVector` function to send a dynamic value to the program. This value is stored in `fco` and is used in the fragment shader to tint the texture color. We upload four numbers to this register, changing the second number (the green component of an r,g,b,a register) for each frame in a smoothly pulsating way.

Next, we construct the `modelViewProjection` matrix, pass it to the shader program, and instruct Stage3D how to interpret the data stored in the vertex buffer. The three `setVertexBufferAt` commands instruct Stage3D how to fill the `va0`, `va1`, and `va2` registers. In this example, we fill `v0` with the x,y,z coordinates, `v2` receives the texture UVs, and `v2` will contain the RGBA for each vertex.

Time for action – uploading data to Stage3D

One of the last things we need to do is upload our mesh and texture data. We do this as follows:

```
private function initData():void
{
    // Defines which vertex is used for each polygon
    // In this example a square is made from two triangles
    meshIndexData = Vector.<uint>
    (
        [
            0, 1, 2,      0, 2, 3,
        ],
        [
            // Raw data used for each of the 4 vertexes
            // Position XYZ, texture coord UV, normal XYZ, vertex RGBA
            meshVertexData = Vector.<Number>
            (
                [
                    //X, Y, Z, U, V, nX, nY, nZ, R, G, B, A
                    -1, -1, 1, 0, 0, 0, 0, 1, 1.0, 0.0, 0.0, 1.0,
                    1, -1, 1, 1, 0, 0, 0, 1, 0.0, 1.0, 0.0, 1.0,
                    1, 1, 1, 1, 1, 0, 0, 1, 0.0, 0.0, 1.0, 1.0,
                    -1, 1, 1, 0, 1, 0, 0, 1, 1.0, 1.0, 0.0, 1.0
                ],
            )
        }
    } // end of class
} // end of package
```

What just happened?

This simple function is used to populate the vertex buffer with the data required to define the locations of each vertex in our mesh. The only difference from last time is that we are including four more data points per vertex, the r,g,b,a. These values are used in some of our vertex programs to output a varying register for use in the fragment shaders to tint the mesh.

Quest complete—time to reap the rewards

Now that the entire source code is complete, publish your .swf file. Use your web browser to browse the demo and make sure it looks just like the screenshot at the beginning of this section. Hopefully, you will be running at a silky smooth 60fps. If you are not, make sure that you are not running multiple programs or more than one instance of the demo. If you still don't get at least 55fps, then ensure that your HTML template includes the line wmode=direct in both the embed and object tags, and finally right click on the Flash file and make sure that the hardware acceleration is ON in your Flash settings.

Notice how each square flies around in 3D and how the fourth shader is animated, pulsing from red to yellow. Note as well how your GUI text stays overlaid at top of the 3D graphics. Can you imagine your game in the near future, when these text overlays are health meters and high score displays? When these 3D squares are complex meshes such as tanks, trees, or castles?

Congratulations!

You have just leveled up. Not only that but the AGAL code in this chapter represents "the wall" in terms of difficulty. If you made it this far, you will easily make it all the way to the end of your adventure. Everything gets much easier from now on. You can be truly proud of yourself for defeating the first boss. By winning this boss battle—the battle of AGAL, you have earned yourself the title of Stage3D expert. You know how to create shaders (vertex and fragment programs). You know how to upload textures and compile these programs. You know how to render multiple instances of the same mesh, how to set up 3D graphics and 2D Flash mixed together in a single .swf, and you officially know how to develop in 3D.

You have officially made it past basic training.

Pop quiz – earn some experience points

Here are a few quick questions to test your understanding. If you can answer them, then you are officially ready to move on to the next step in your grand adventure.

1. If you want to send a value from your vertex shader to your fragment shader, which set of registers would you use?
 - a. v0 to v7 (varying)
 - b. vt0 to vt7 (vertex temporary)
 - c. oc (output color)
 - d. Registered mail might work

2. What register do you write the final vertex position to at the end of your vertex program?
 - a. va1 to va7 (vertex attribute)
 - b. op (output position)
 - c. vc0 to vc127 (vertex constant)
 - d. Probably a gift registry

3. If you wanted to subtract a color stored in fc0 from a color stored in fc1 and store the answer in ft0, what would the AGAL source code look like?
 - a. $ft0 = fc0 - fc1$
 - b. `sub fc0, fc1, ft0`
 - c. `sub ft0, fc0, fc1`
 - d. I could go for a sub right about now

Have a go hero – shader experiments

In order to really hone your skills, it can be a good idea to challenge yourself for some extra experience. At the end of each chapter, there is a side quest—without a solution provided—for you to experiment with and it is optional. Just like grinding in an RPG game, challenges like these are designed to make you stronger, so that when you forge ahead, you are more than ready for the next step in your main quest.

Your side quest is to experiment with all the colors of the rainbow. Tweak the fragment shaders in your new Flash demo, so that all four meshes have a lovely shade of dark purple. Make one of them flicker from black to white rapidly like a strobe light. Edit the `context3D.clear` background color to be pure white instead of black (or make it gradually change over time). Change the texture to be a photo of your best friend. Try making each mesh use a completely different texture by including four different images in your project.

If you want even more of a challenge, tweak the animation of the meshes, so that they move in and out towards the camera, or take up the entire screen. Make one of them spin at extremely high speed while another only moves from left to right, as slow as a snail. Make another vibrate like a bomb about to explode or spin in place like a record player.

Just diving in, without regard to making mistakes, is a great way to familiarize you with some simple AGAL commands. Who cares if it doesn't work the way you planned or results in an error message?

The more comfortable you are—and confident—the less intimidating this new language called AGAL will become. Screw it up on purpose. Try to wreck everything. You have nothing to lose but your fear.

Summary

We were victorious in completing the next quest along the way to the creation of our own 3D video game in Flash. There was a lot of material covered in this chapter, and you don't have to memorize it all. You can always come back to remind yourself what the different registers are, or how to set up a basic shader program. We learned the basics of AGAL: how to compile vertex and fragment programs, what the different registers are used for, how to send data to your shaders, how to render multiple instances of the same mesh, and how to overlay Flash 2D text and sprites over the top of a 3D scene.

We have created a solid foundation for the creation of a 3D Flash game. Not only do we have a working renderer with multiple shaders, but also we have a GUI that can display the frame rate. This GUI could be extended to show the player's score or health. Your shader library will eventually grow to include all the effects needed for a beautiful looking game. You will soon be rendering more complex meshes, anything from detailed mountains to futuristic buildings.

Level 4 achieved!

Congratulations are in order, brave hero. You have successfully "leveled up" again. As a level four Molehill master, you have earned the right to start including complex models in your game, animating the camera, and adding gameplay—evolving this tech demo into a real game. This is what awaits you in the next chapter!

5

Building a 3D World

Things are about to get really fun.

Now that you know about getting things initialized, you must be hungering for something a little bit more impressive. This chapter promises to be tons of fun because we are about to move beyond simple rotating squares to rendering complex 3D models using multiple shaders. Your demo will look awesome!

In the previous chapter, we learned how to write AGAL shaders—some with textures, some with vertex colors, and some that were even animated. Everything, however, was using the simplest of meshes, a plain old boring square, floating in space. Enough of such simplicity! Time to enter the big leagues. You are ready to render some beautiful looking 3D art.

In this chapter, we will write a basic parser for one of the most common 3D file formats. In this way, you will be able to sculpt amazing models such as spaceships or cars or buildings or trees and watch in awestruck amazement as Flash renders hundreds of thousands of polygons without so much as a whimper.

Up to this point in our adventure, we have performed simple quests that barely scratched the surface of the power of the Stage3D API. Like a level one fetch quest or having to fight a hundred rabid cave rats, it was great for practice, but was not particularly impressive. The great news is that you are officially ready for "level five" challenges. You are ready to create a full 3D game world, filled with detail and animation.

In this chapter, we are going to learn:

- ◆ All about the render loop
- ◆ Parsing .OBJ mesh data and creating vertex buffers
- ◆ Rendering more than one mesh at once
- ◆ Animating your meshes to make them move

You are officially ready for larger prey. Sharpen your sword and get ready to battle an opponent worthy of your prowess.

Creating vertex buffers

In this chapter, we are going to do lots of updating each frame in a render loop: we will scroll a background terrain and move a spaceship around.

Before we start animating things, we will import some better looking art, so that we have something nice to look at.

You can view the completed demo here:

http://www.mcfunkypants.com/molehill/chapter_5_demo/

You can download the complete source code here:

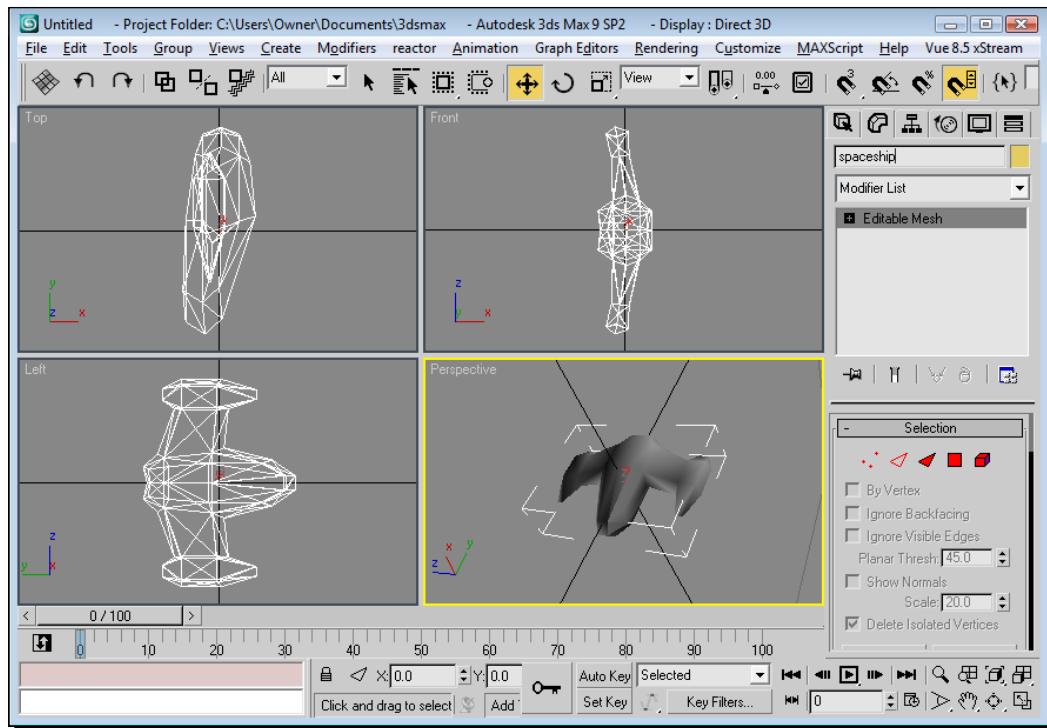
<http://www.packtpub.com/adobe-flash11-stage3d-molehill-game-programming-beginners-guide/book>

As we know, Stage3D sends all model data to your video card for speedy rendering. Once we send a list of vertex coordinates and other related data during initialization, we never have to do so again and we can repeatedly render those meshes during runtime with one line of code.

In previous examples, we used a single `VertexBuffer3D` for our mesh vertex xyz coordinates, texture uv coordinates and vertex rgb values. This is optional—you can put nearly any number of values all together in a single vertex buffer, or you can create multiple smaller buffers for use in your vertex program. For example, you could create separate buffers for each type of data, such as one for just the xyz and another for the uv coordinates. This is done by using the `setVertexBufferAt` function, which you can call multiple times on multiple buffers before rendering your mesh with the `drawTriangles` function.

Let's write a class that can gobble up complex models and spit out vertex buffers ready for use in Flash.

Before we start using this new class, you are going to need a good looking 3D model to work with. The example project for this chapter contains a few `.OBJ` files for use as an example, but if you wish to import your own art, now is a great time. Fire up the 3D modeling app of your choice (3D Studio Max, Maya, Blender, and so on) and either open an existing file or create one from scratch.

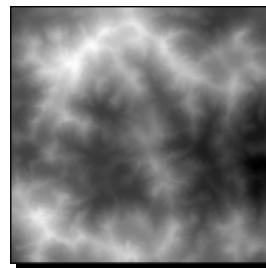


The example model we want to use for this chapter should not be super complex, perhaps 500 to 5,000 polygons. In order to avoid any tech issues, make it just a single mesh (no complex object hierarchy). For simplicity's sake, choose a mesh that uses a single texture. Using Photoshop or GIMP or any other art editor, make sure the texture that your mesh uses is square and a power-of-two in size (128x128, 256x256, 512x512, 1024x1024). When you have this sample model ready, export it as an .OBJ file.

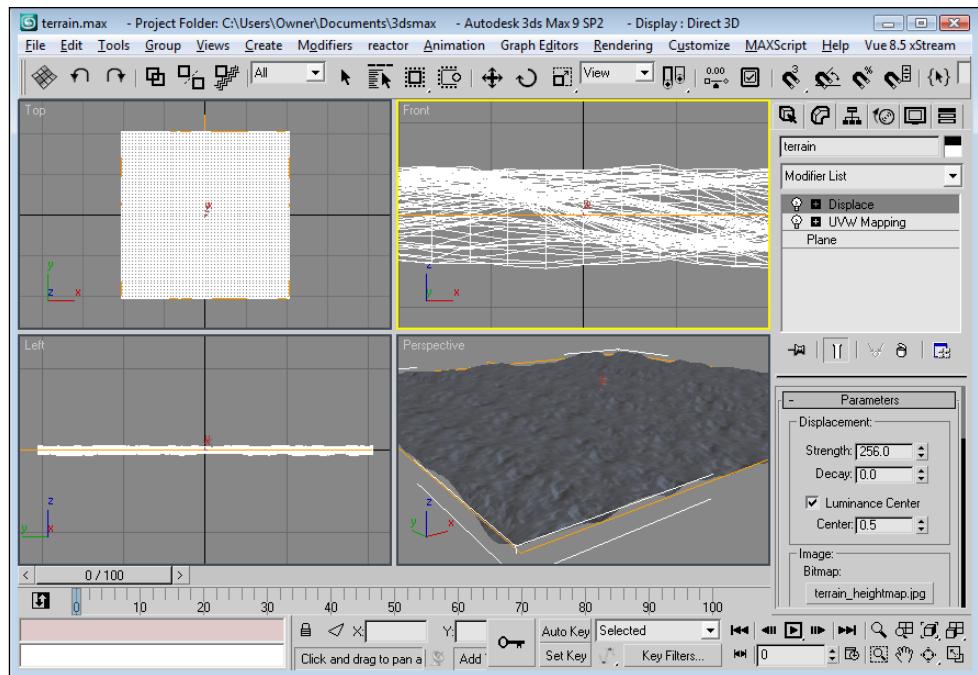


If you want to really challenge yourself, make a few models and textures. Just for fun, take a look at the insides of one of the files using a text editor. You will see how simple it is to parse the data.

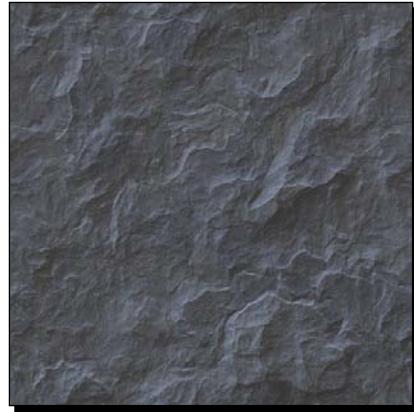
For the example that will be used in this book, we want to animate a spaceship flying over a sci-fi terrain. The preceding simple ship model will do for our "main character" and all that remains is to sculpt the terrain. One handy technique for terrains is to create a heightmap image, where white is high altitude and black is low altitude. You can use a Photoshop filter for this, or real-world terrain data.



By using this heightmap image, you can make your 3D editor (such as 3D Studio Max, Maya, or Blender) and generate a mesh terrain using a "displacement" modifier on a flat plane comprised of many polygons.



Finally, we want to texture the terrain, so create or find something "rocky".



Once you have created the art you want, you need to export it for use in your game. A great file format to start with is the `.OBJ` format, which is extremely common—almost every single 3D editor can export files in this format. It is very simple, and if you were to look inside an `.OBJ` file, you would see that it contains text data. It is also very sparse, without any bloated XML markup, which is great because it keeps the file size down and compresses nicely when you compile your `.SWF`.

This is what `.OBJ` data looks like:

```
v -0.257 0.191 0.423  
vt 0.623 0.227  
f 1/1 2/2 3/3
```

The first line is vertex x,y,z coordinate data, the second line is texture uv coordinates, and the third line is a face (poly) definition which is used to generate an index buffer. Naturally, a complex model will have hundreds or thousands of lines like these and might include more data types such as vertex normals, but it helps to be able to imagine how all your model data is being saved. As you can see, an `.OBJ` file is dirt simple: perfect for our needs!

The simplicity of the `.OBJ` file format (being a text file) makes it super easy to handle with regard to writing an AS3 import class, which we are going to do right now. There are many other file formats that you may eventually want to work with. Some will be more feature rich, such as COLLADA, and most will require more complex parsing functionality.

For now, however, choosing `.OBJ` is wise. Though you have graduated from the proverbial "kill ten rats" quest, you don't want to start attacking a giant boss quite yet. Consider this your first real opponent—a level five goblin, so to speak.

Importing 3D models into Flash

Now that you have some art to work with, we need to be able to import it into your game engine. Open your current template AS3 project that you created in the previous chapter and add a brand new class named `Stage3dObjParser`.

Let's create the most simple file parser that we can—this class should do only one thing. It should be able to parse the data inside your file and output the required vertex buffer and index buffer. Instead of making this class capable of doing any rendering, let's leave any shaders (vertex or fragment programs) for later and focus on nothing more than importing 3D models here.

Create a class that can parse object data and turn it into a Stage3D vertex buffer by carrying out the following steps:

1. Create a new blank class named `Stage3dObjParser`
2. Read in the data and turn it from a string to an array
3. Iterate through each value in the data and store them in the proper arrays
4. Loop through all the data and create properly formed vertex buffers

Time for action – coding the `Stage3dObjParser` class

In order to perform all the preceding steps, first create a blank `.as` file and begin by importing the classes we need and setting some constants and variables that our parser will need. We do this as follows:

```
package
{
    // gratefully adapted from work by Alejandro Santander

    import flash.geom.Vector3D;
    import flash.geom.Matrix3D;
    import flash.utils.ByteArray;
    import flash.display3D.Context3D;
    import flash.display3D.IndexBuffer3D;
    import flash.display3D.VertexBuffer3D;
```

Next, let's define our new class and create a few private variables and constants that it will require.

```
public class Stage3dObjParser
{
    // older versions of 3dsmax use an invalid vertex order:
    private var _vertexDataIsZxy:Boolean = false;
    // some exporters mirror the UV texture coordinates
```

```
private var _mirrorUv:Boolean = false;
// OBJ files do not contain vertex colors
// but many shaders will require this data
// if false, the buffer is filled with pure white
private var _randomVertexColors:Boolean = true;
```

The vars above will be set during inits to account for different file formats you may encounter.

```
// constants used in parsing OBJ data
private const LINE_FEED:String = String.fromCharCode(10);
private const SPACE:String = String.fromCharCode(32);
private const SLASH:String = "/";
private const VERTEX:String = "v";
private const NORMAL:String = "vn";
private const UV:String = "vt";
private const INDEX_DATA:String = "f";
```

The preceding constants are used in the .OBJ file parser and represent the characters in the file that delimit each type of data.

```
// temporary vars used during parsing OBJ data
private var _scale:Number;
private var _faceIndex:uint;
private var _vertices:Vector.<Number>;
private var _normals:Vector.<Number>;
private var _uvs:Vector.<Number>;
private var _cachedRawNormalsBuffer:Vector.<Number>;
// the raw data that is used to create Stage3d buffers
protected var _rawIndexBuffer:Vector.<uint>;
protected var _rawPositionsBuffer:Vector.<Number>;
protected var _rawUvBuffer:Vector.<Number>;
protected var _rawNormalsBuffer:Vector.<Number>;
protected var _rawColorsBuffer:Vector.<Number>;
// the final buffers in Stage3d-ready format
protected var _indexBuffer:IndexBuffer3D;
protected var _positionsBuffer:VertexBuffer3D;
protected var _uvBuffer:VertexBuffer3D;
protected var _normalsBuffer:VertexBuffer3D;
protected var _colorsBuffer:VertexBuffer3D;
// the context3D that we want to upload the buffers to
private var _context3d:Context3D;
```

What just happened?

Our new class needs to import various functions from Flash such as those that define the `VertexBuffer3D` class and other Stage3D-specific functionality. After the `import` statements, we create a number of private and protected variables that will be used internally by our class. None of these variables will need to be accessed from outside the class, as we will ask for them in our main render loop by using `get` functions.

Time for action – creating the class constructor function

Add the following function at the bottom of the file. This is the constructor for our new class, and this is where the parsing begins.

```
// the class constructor - where everything begins
public function Stage3dObjParser(objfile:Class,
    acontext:Context3D, scale:Number = 1,
    dataIsZxy:Boolean = false, textureFlip:Boolean = false)
{
    _vertexDataIsZxy = dataIsZxy;
    _mirrorUv = textureFlip;

    _rawColorsBuffer = new Vector.<Number>();
    _rawIndexBuffer = new Vector.<uint>();
    _rawPositionsBuffer = new Vector.<Number>();
    _rawUvBuffer = new Vector.<Number>();
    _rawNormalsBuffer = new Vector.<Number>();
    _scale = scale;
    _context3d = acontext;

    // Get data as string.
    var definition:String = readClass(objfile);

    // Init raw data containers.
    _vertices = new Vector.<Number>();
    _normals = new Vector.<Number>();
    _uvs = new Vector.<Number>();

    // Split data into lines and parse all lines.
    var lines:Array = definition.split(LINE_FEED);
    var loop:uint = lines.length;
    for(var i:uint = 0; i < loop; ++i)
        parseLine(lines[i]);
}
```

The preceding class constructor is run whenever a new object of this kind of class is created in our game. We pass the .OBJ data, a pointer to our already-initialized Context3D and optionally some settings. The data is then parsed by looping through each line.

```
private function readClass(f:Class) :String
{
    var bytes:ByteArray = new f();
    return bytes.readUTFBytes(bytes.bytesAvailable);
}
```

The preceding function simply turns the [embedded] data into a String, ready for processing. The following `parseLine` function does the work of deciding which kind of data is stored on the next line of text in our .OBJ file and runs the appropriate parsing function.

```
private function parseLine(line:String):void
{
    // Split line into words.
    var words:Array = line.split(SPACES);

    // Prepare the data of the line.
    if (words.length > 0)
        var data:Array = words.slice(1);
    else
        return;

    // Check first word and delegate remainder to proper parser.
    var firstWord:String = words[0];
    switch (firstWord)
    {
        case VERTEX:
            parseVertex(data);
            break;
        case NORMAL:
            parseNormal(data);
            break;
        case UV:
            parseUV(data);
            break;
        case INDEX_DATA:
            parseIndex(data);
            break;
    }
}
```

What just happened?

This constructor will create a new object in our Flash program called a `Stage3dObjParser`. In our Stage3D game, we will create one (or more) of these classes, and store them to variables ready for rendering.

When we create the class, we accept two parameters: the model data itself and a scale value, which can be used to size the mesh larger or smaller if our game world uses a different scale than was used during the modeling.

For simplicity, and so that your `.SWF` file does not need to reside on a web server and has no external file dependencies, we will be sending embedded data to this class. This is a good technique when you are using a small number of models that are not too massive. It is especially important when you plan to upload your game to portal websites, which generally require that your game resides in a single `.SWF` file without any other data files. The good news is that mesh data compresses very nicely and you can easily store dozens of very high poly models in your `.SWF` without taking too much bandwidth.

After initializing some variables that will be used internally, we call a function that turns the embedded data into a string. We then split the string into an array of strings (one item for each line of text) and then loop through each line, running the `parseLine` function on whatever data is stored there.

Time for action – coding the parsing functions

The next step is to define all the functions required to parse the `.OBJ` data we sent to the constructor above. Add the following functions at the bottom of your file.

```
private function parseVertex(data:Array) :void
{
    if ((data[0] == '') || (data[0] == ' '))
        data = data.slice(1); // delete blanks
    if (_vertexDataIsZxy)
    {
        //if (!_vertices.length) trace('zxy parseVertex: '
        // + data[1] + ',' + data[2] + ',' + data[0]);
        _vertices.push(Number(data[1])*_scale);
        _vertices.push(Number(data[2])*_scale);
        _vertices.push(Number(data[0])*_scale);
    }
    else // normal operation: x,y,z
    {
        //if (!_vertices.length) trace('parseVertex: ' + data);
        var loop:uint = data.length;
        if (loop > 3) loop = 3;
```

```

        for (var i:uint = 0; i < loop; ++i)
        {
            var element:String = data[i];
            _vertices.push(Number(element)*_scale);
        }
    }
}

```

The preceding function parses any vertex (xyz coordinate) data and turns that data into a number that is stored in a temporary array.

```

private function parseNormal(data:Array) :void
{
    if ((data[0] == '') || (data[0] == ' '))
        data = data.slice(1); // delete blanks
    //if (!_normals.length) trace('parseNormal:' + data);
    var loop:uint = data.length;
    if (loop > 3) loop = 3;
    for (var i:uint = 0; i < loop; ++i)
    {
        var element:String = data[i];
        if (element != null) // handle 3dsmax extra spaces
            _normals.push(Number(element));
    }
}

```

The `parseNormal` function does the same thing, except for normal data. Not all .OBJ files contain normal data, but it is handy for use in shaders that add light or shines, as the shader needs to know in what direction each poly in your mesh is facing to determine how the light will affect it.

```

private function parseUV(data:Array) :void
{
    if ((data[0] == '') || (data[0] == ' '))
        data = data.slice(1); // delete blanks
    //if (!_uvs.length) trace('parseUV:' + data);
    var loop:uint = data.length;
    if (loop > 2) loop = 2;
    for (var i:uint = 0; i < loop; ++i)
    {
        var element:String = data[i];
        _uvs.push(Number(element));
    }
}

```

The preceding function parses the texture coordinates (uv data) for your mesh. If your mesh uses a texture, each vertex in your mesh will have a u and v coordinate to define where in the texture image the pixels should come from.

```
private function parseIndex(data:Array):void
{
    //if (!_rawIndexBuffer.length) trace('parseIndex:' + data);
    var triplet:String;
    var subdata:Array;
    var vertexIndex:int;
    var uvIndex:int;
    var normalIndex:int;
    var index:uint;

    // Process elements.
    var i:uint;
    var loop:uint = data.length;
    var starthere:uint = 0;
    while ((data[starthere] == '') || (data[starthere] == ' '))
        starthere++; // ignore blanks

    loop = starthere + 3;

    // loop through each element and grab values stored earlier
    // elements come as vertexIndex/uvIndex/normalIndex
    for(i = starthere; i < loop; ++i)
    {
        triplet = data[i];
        subdata = triplet.split(SLASH);
        vertexIndex = int(subdata[0]) - 1;
        uvIndex     = int(subdata[1]) - 1;
        normalIndex = int(subdata[2]) - 1;

        // sanity check
        if(vertexIndex < 0) vertexIndex = 0;
        if(uvIndex < 0) uvIndex = 0;
        if(normalIndex < 0) normalIndex = 0;
    }
}
```

This function starts by parsing a line of "index" data. The index buffer is used because in some models more than one vertex can use the same xyz coordinate, for example, sharing the same data. Each vertex, therefore, simply uses an index that instructs Stage3D which element in an array of xyz positions to use for that particular vertex. We loop through all the data collected so far and, using the index to look up which array element in our temp arrays we need next, we copy all the data in the correct order. These new arrays are filled in just the right way to be ready for uploading to Flash.

```

// Extract from parse raw data to mesh raw data.

// Vertex (x,y,z)
index = 3*vertexIndex;
_rawPositionsBuffer.push(_vertices[index + 0],
    _vertices[index + 1], _vertices[index + 2]);

// Color (vertex r,g,b,a)
if (_randomVertexColors)
    _rawColorsBuffer.push(Math.random(),
        Math.random(), Math.random(), 1);
else
    _rawColorsBuffer.push(1, 1, 1, 1); // pure white

// Normals (nx,ny,nz) - *if* included in the file
if (_normals.length)
{
    index = 3*normalIndex;
    _rawNormalsBuffer.push(_normals[index + 0],
        _normals[index + 1], _normals[index + 2]);
}

// Texture coordinates (u,v)
index = 2*uvIndex;
if (_mirrorUv)
    _rawUvBuffer.push(_uvs[index+0], 1-_uvs[index+1]);
else
    _rawUvBuffer.push(1-_uvs[index+0], 1-_uvs[index+1]);
}

// Create index buffer - one entry for each polygon
_rawIndexBuffer.push(_faceIndex+0, _faceIndex+1, _faceIndex+2);
_faceIndex += 3;

}

```

What just happened?

In the preceding code, we first define the very simple `readClass` function, which simply takes an embedded class (any data at all) and turns it into a string, ready for parsing. We then create the `parseLine` function, which does all the heavy lifting. It splits the line of data into words and then runs a function on the line depending on whether the line starts with a particular character.

The .OBJ file format contains data on every line and each line starts with either a "v" to denote that the next few values represent vertex x,y,z coordinates, or "vt" which instructs the parser that the next two values are u,v, texture coordinates, and so on.

This function simply decides what each line type is and then calls the appropriate function. Each of these functions grabs the data and pushes the data into an array, ready for further processing.

Time for action – processing the data

Now we simply need to create Stage3D-compatible VertexBuffer3Ds for use during the render loop. We have already parsed the data stored in the .OBJ file. The following routines simply set everything up for rendering:

```
// These functions return Stage3d buffers
// (uploading them first if required)

public function get colorsBuffer():VertexBuffer3D
{
    if(!_colorsBuffer)
        updateColorsBuffer();
    return _colorsBuffer;
}

public function get positionsBuffer():VertexBuffer3D
{
    if(!_positionsBuffer)
        updateVertexBuffer();
    return _positionsBuffer;
}

public function get indexBuffer():IndexBuffer3D
{
    if(!_indexBuffer)
        updateIndexBuffer();
    return _indexBuffer;
}

public function get indexBufferCount():int
{
    return _rawIndexBuffer.length / 3;
}

public function get uvBuffer():VertexBuffer3D
```

```
{  
    if(!_uvBuffer)  
        updateUvBuffer();  
    return _uvBuffer;  
}  
  
public function get normalsBuffer():VertexBuffer3D  
{  
    if(!_normalsBuffer)  
        updateNormalsBuffer();  
    return _normalsBuffer;  
}
```

The preceding functions do nothing but return the proper `VertexBuffer3D` if it has already been created. If it has yet to be created, we first generate it and upload it to `Stage3D`. Doing things this way ensures that vertex buffers are only created once—the first time we ask for them. After that, we can re-use them over and over.

```
// convert RAW buffers to Stage3d compatible buffers  
// uploads them to the context3D first  
  
public function updateColorsBuffer():void  
{  
    if(_rawColorsBuffer.length == 0)  
        throw new Error("Raw Color buffer is empty");  
    var colorsCount:uint = _rawColorsBuffer.length/4; // 4=rgba  
    _colorsBuffer = _context3d.createVertexBuffer(colorsCount, 4);  
    _colorsBuffer.uploadFromVector(  
        _rawColorsBuffer, 0, colorsCount);  
}  
  
public function updateNormalsBuffer():void  
{  
    // generate normals manually  
    // if the data file did not include them  
    if (_rawNormalsBuffer.length == 0)  
        forceNormals();  
    if(_rawNormalsBuffer.length == 0)  
        throw new Error("Raw Normal buffer is empty");  
    var normalsCount:uint = _rawNormalsBuffer.length/3;  
    _normalsBuffer = _context3d.createVertexBuffer(normalsCount, 3);  
    _normalsBuffer.uploadFromVector(  
        _rawNormalsBuffer, 0, normalsCount);  
}
```

```
public function updateVertexBuffer():void
{
    if(_rawPositionsBuffer.length == 0)
        throw new Error("Raw Vertex buffer is empty");
    var vertexCount:uint = _rawPositionsBuffer.length/3;
    _positionsBuffer = _context3d.createVertexBuffer(vertexCount, 3);
    _positionsBuffer.uploadFromVector(
        _rawPositionsBuffer, 0, vertexCount);
}

public function updateUvBuffer():void
{
    if(_rawUvBuffer.length == 0)
        throw new Error("Raw UV buffer is empty");
    var uvsCount:uint = _rawUvBuffer.length/2;
    _uvBuffer = _context3d.createVertexBuffer(uvsCount, 2);
    _uvBuffer.uploadFromVector(
        _rawUvBuffer, 0, uvsCount);
}

public function updateIndexBuffer():void
{
    if(_rawIndexBuffer.length == 0)
        throw new Error("Raw Index buffer is empty");
    _indexBuffer =
        _context3d.createIndexBuffer(_rawIndexBuffer.length);
    _indexBuffer.uploadFromVector(
        _rawIndexBuffer, 0, _rawIndexBuffer.length);
}
```

What just happened?

The first group of the preceding functions simply returns the vertex buffers they have been asked for. They first check to see if the buffer has been created, and if not, functions that call `createIndexBuffer()` are run first, so that all the data has been uploaded to Stage3D. This is only done the first time the buffer is requested, so that in subsequent frames the buffers are not uploaded again.

The second group of functions actually creates these Stage3D vertex buffers by uploading the data stored in our "raw" private variables. The vertex xyz, uv, rgba, and index buffers at this point are actually stored in video RAM (on your 3D card) and can be used to render very efficiently.

Time for action – coding some handy utility functions

Some .OBJ file exporters can be configured to exclude vertex normals. This helps to reduce file size and is the best choice if your shader does not use this data. Some shaders, however, require vertex normals (such as those that use dynamic lighting or shininess). In these cases, it is best to export your meshes with vertex normal data in the .OBJ file, but if you are unable to do so the following functions will manually calculate the normal for each vertex and fill in the missing data for you:

```

public function restoreNormals():void
{
    // utility function
    _rawNormalsBuffer = _cachedRawNormalsBuffer.concat();
}

public function get3PointNormal(
    p0:Vector3D, p1:Vector3D, p2:Vector3D):Vector3D
{
    // utility function
    // calculate the normal from three vectors
    var p0p1:Vector3D = p1.subtract(p0);
    var p0p2:Vector3D = p2.subtract(p0);
    var normal:Vector3D = p0p1.crossProduct(p0p2);
    normal.normalize();
    return normal;
}

public function forceNormals():void
{
    // utility function
    // useful for when the OBJ file doesn't have normal data
    // we can calculate it manually by calling this function
    _cachedRawNormalsBuffer = _rawNormalsBuffer.concat();
    var i:uint, index:uint;
    // Translate vertices to vector3d array.
    var loop:uint = _rawPositionsBuffer.length/3;
    var vertices:Vector.<Vector3D> = new Vector.<Vector3D>();
    var vertex:Vector3D;
    for(i = 0; i < loop; ++i)
    {
        index = 3*i;
        vertex = new Vector3D(_rawPositionsBuffer[index],
            _rawPositionsBuffer[index + 1],
            _rawPositionsBuffer[index + 2]);
        vertices.push(vertex);
    }
}

```

```
// Calculate normals.  
loop = vertices.length;  
var p0:Vector3D, p1:Vector3D, p2:Vector3D, normal:Vector3D;  
_rawNormalsBuffer = new Vector.<Number>();  
for(i = 0; i < loop; i += 3)  
{  
    p0 = vertices[i];  
    p1 = vertices[i + 1];  
    p2 = vertices[i + 2];  
    normal = get3PointNormal(p0, p1, p2);  
    _rawNormalsBuffer.push(normal.x, normal.y, normal.z);  
    _rawNormalsBuffer.push(normal.x, normal.y, normal.z);  
    _rawNormalsBuffer.push(normal.x, normal.y, normal.z);  
}  
}
```

The preceding functions are used to manually calculate the normals for each vertex, which is handy for times when your source data file does not include that data.

```
// utility function that outputs all buffer data  
// to the debug window - good for compiling OBJ to  
// pure as3 source code for faster inits  
public function dataDumpTrace():void  
{  
    trace(dataDumpString());  
}  
// turns all mesh data into AS3 source code  
public function dataDumpString():String  
{  
    var str:String;  
    str = "// Stage3d Model Data begins\n\n";  
  
    str += "private var _Index:Vector.<uint> " ;  
    str += "= new Vector.<uint>([";  
    str += _rawIndexBuffer.toString();  
    str += "]);\n\n";  
  
    str += "private var _Positions:Vector.<Number> " ;  
    str += "= new Vector.<Number>([";  
    str += _rawPositionsBuffer.toString();  
    str += "]);\n\n";  
  
    str += "private var _UVs:Vector.<Number> = " ;  
    str += "new Vector.<Number>([";  
    str += _rawUvBuffer.toString();
```

```
str += "]);\n\n";  
  
str += "private var _Normals:Vector.<Number> = ";  
str += "new Vector.<Number>([";  
str += _rawNormalsBuffer.toString();  
str += "]);\n\n";  
  
str += "private var _Colors:Vector.<Number> = ";  
str += "new Vector.<Number>([";  
str += _rawColorsBuffer.toString();  
str += "]);\n\n";  
  
str += "// Stage3d Model Data ends\n";  
return str;  
}  
  
} // end class  
  
} // end package
```

What just happened?

We took the time to create some helper functions. We created functions that automatically generate (or reset) the mesh normal data by manually calculating the normals. This is handy when you need normals for your shaders, but your .OBJ file does not contain them.

The preceding final function is a simple debug output function that lists the contents of each buffer in the Flash debug log. This is handy if you want to avoid any processing of model data and wish to simply embed the actual source code for all the mesh data directly into your game. Doing so would eliminate almost all startup initialization time.

Our mesh parsing class is complete!

This is all that is required to be able to embed .OBJ data into your Flash files. The preceding functions turn this data into vertex buffers, ready for speedy rendering in your game.

Let's make this happen, shall we?

The render loop

In game programming discussions, you will often hear mention of "the render loop". This refers to the continuous rendering over time that your game keeps doing over and over. Once per frame, a typical game engine will calculate new positions for meshes, update scores, and track events. It will then render the new scene and present it to the user. This process is looped "forever" or at least until the user quits the game.

For example, you might want to do the following each frame:

- ◆ Update the GUI (score, and so on)
- ◆ Change the camera transform
- ◆ Update object positions
- ◆ Animate the shader registers
- ◆ Select the proper vertex buffers
- ◆ Render your geometry

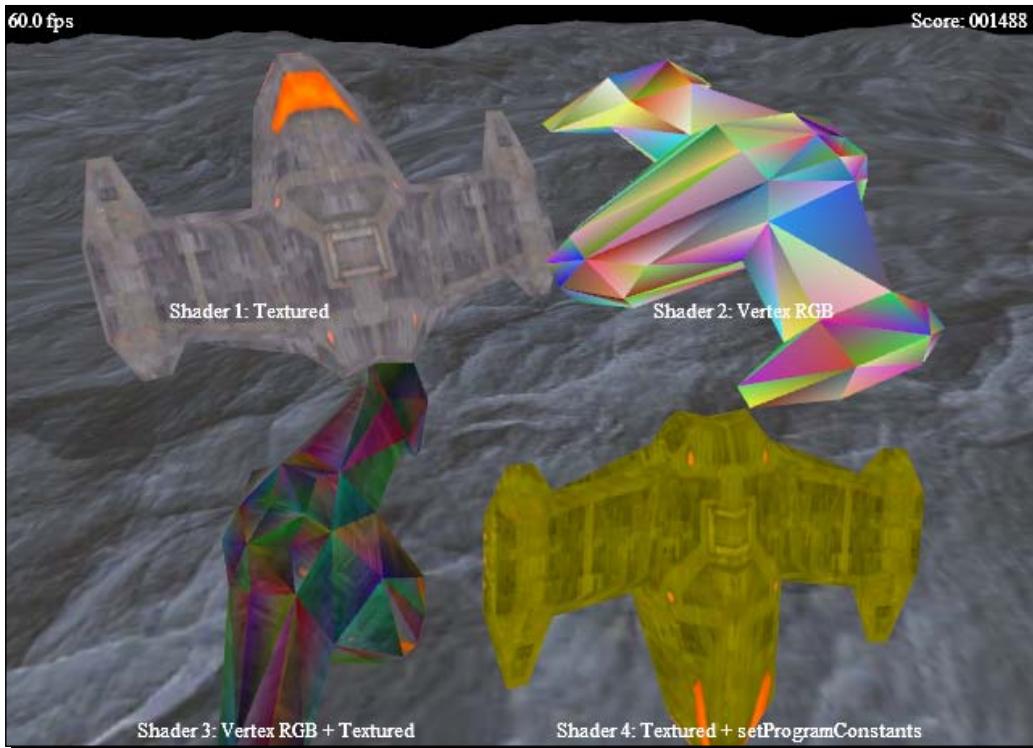
Almost every game ever written has a function that handles the so-called render loop. Naturally, your game might have different states, such as when it is waiting at the main menu or has a game-over display, and perhaps different routines are called depending on what is going on in your game. Regardless of the state your game is in, most games will animate every single frame as fast as it can. Over and over, often 60 times per second, your game engine has to be able to update what is being displayed.

Just like individual frames in a film, each frame that is rendered during this loop is a still image on the screen. Repeatedly updating what is displayed on the screen as fast as possible looks like smooth animation or movement. For example, as a spaceship flies over a moonscape, or as an adventurer runs around in a dungeon, each change in the location of your models needs to be reflected in a brand new image, so that users see the animated movement. That is what the render loop does.

In future versions of your game, we might add keyboard input checks, sound effects, collision detection, AI, and particle systems here.

Time for action – starting the render loop

Just to whet your appetite, we are aiming for a scene that looks like the following image. Imagine the terrain scrolling by while the score increases over time:



Let's upgrade your demo to make this happen. Close the `Stage3dObjParser.as` class file that we just wrote and return to the `Stage3dGame.as` that you created in the previous chapter. Instead of a bunch of rotating squares, we are going to upgrade our game engine to include all the great-looking art we just created.

```

//  

// Stage3D Game Template - Chapter Five  

//  

package  

{  

[SWF(width="640", height="480", frameRate="60",  

backgroundColor="#FFFFFF")]  

import com.adobe.utils.*;  

import flash.display.*;  

import flash.display3D.*;  

import flash.display3D.textures.*;  

import flash.events.*;  

import flash.geom.*;  

import flash.utils.*;  

import flash.text.*;

```

So far, everything in our game class remains exactly the same as in the previous chapter. Above, import the Flash classes we will need below.

```
public class Stage3dGame extends Sprite
{
    // used by the GUI
    private var fpsLast:uint = getTimer();
    private var fpsTicks:uint = 0;
    private var fpsTf:TextField;
    private var scoreTf:TextField;
    private var score:uint = 0;

    // constants used during inits
    private const swfWidth:int = 640;
    private const swfHeight:int = 480;
    // for this demo, ensure ALL textures are 512x512
    private const textureSize:int = 512;

    // the 3d graphics window on the stage
    private var context3D:Context3D;
    // the compiled shaders used to render our mesh
    private var shaderProgram1:Program3D;
    private var shaderProgram2:Program3D;
    private var shaderProgram3:Program3D;
    private var shaderProgram4:Program3D;

    // matrices that affect the mesh location and camera angles
    private var projectionmatrix:PerspectiveMatrix3D =
        new PerspectiveMatrix3D();
    private var modelmatrix:Matrix3D = new Matrix3D();
    private var viewmatrix:Matrix3D = new Matrix3D();
    private var terrainviewmatrix:Matrix3D = new Matrix3D();
    private var modelViewProjection:Matrix3D = new Matrix3D();

    // a simple frame counter used for animation
    private var t:Number = 0;
    // a reusable loop counter
    private var looptemp:int = 0;
```

In the preceding code, we define a few variables, exactly like in the previous chapter. The only difference is that we don't bother with vars related to vertex buffers and the like since these will now be handled by our fancy new OBJ parser class.

```
/* TEXTURES: Pure AS3 and Flex version:  
 * if you are using Adobe Flash CS5  
 * comment out the following: */  
// [Embed (source = "art/spaceship_texture.jpg")]  
//private var myTextureBitmap:Class;  
//private var myTextureData:Bitmap = new myTextureBitmap();  
// [Embed (source = "art/terrain_texture.jpg")]  
//private var terrainTextureBitmap:Class;  
//private var terrainTextureData:Bitmap = new terrainTextureBitmap();  
  
/* TEXTURE: Flash CS5 version:  
 * add the jpgs to your library (F11)  
 * right click and edit the advanced properties  
 * so it is exported for use in Actionscript  
 * and call them myTextureBitmap and terrainTextureBitmap  
 * if you are using Flex/FlashBuilder/FlashDevelop/FDT  
 * comment out the following: */  
private var myBitmapDataObject:myTextureBitmapData =  
    new myTextureBitmapData(textureSize, textureSize);  
private var myTextureData:Bitmap =  
    new Bitmap(myBitmapDataObject);  
private var terrainBitmapDataObject:terrainTextureBitmapData =  
    new terrainTextureBitmapData(textureSize, textureSize);  
private var terrainTextureData:Bitmap =  
    new Bitmap(terrainBitmapDataObject);  
  
// The Stage3d Texture that uses the above myTextureData  
private var myTexture:Texture;  
private var terrainTexture:Texture;  
  
// The spaceship mesh data  
[Embed (source = "art/spaceship.obj",  
        mimeType = "application/octet-stream")]  
private var myObjData:Class;  
private var myMesh:Stage3dObjParser;  
  
// The terrain mesh data  
[Embed (source = "art/terrain.obj",  
        mimeType = "application/octet-stream")]  
private var terrainObjData:Class;  
private var terrainMesh:Stage3dObjParser;
```

In the preceding code, we embed the texture and mesh data that will be used by our `Stage3dObjParser` class during the inits.

```
public function Stage3dGame()
{
    if (stage != null)
        init();
    else
        addEventListener(Event.ADDED_TO_STAGE, init);
}

private function init(e:Event = null):void
{
    if (hasEventListener(Event.ADDED_TO_STAGE))
        removeEventListener(Event.ADDED_TO_STAGE, init);
    // class constructor - sets up the stage
    stage.scaleMode = StageScaleMode.NO_SCALE;
    stage.align = StageAlign.TOP_LEFT;

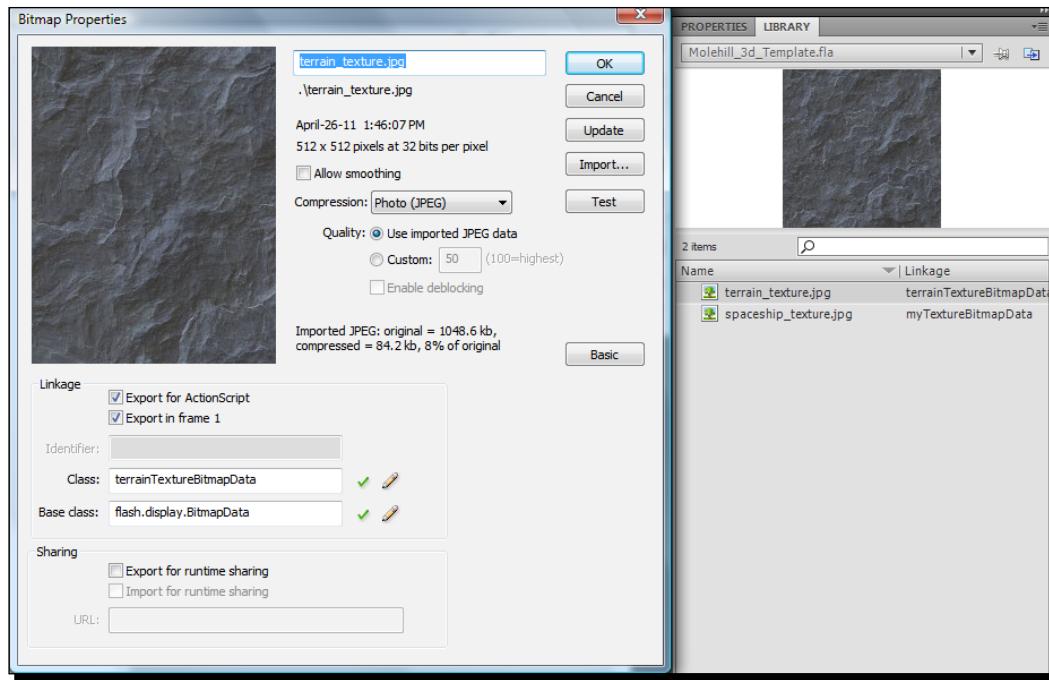
    // add some text labels
    initGUI();

    // and request a context3D from Stage3d
    stage.stage3Ds[0].addEventListener(
        Event.CONTEXT3D_CREATE, onContext3DCreate);
    stage.stage3Ds[0].requestContext3D();
}
```

What just happened?

You will notice that there are only a few small changes from what you typed last time. In particular, there are two textures and two meshes being embedded. There is also a new variable and GUI element for displaying a score on the screen.

In order for the two textures to be included (if you are using CS5), you will need to open the Library palette (*F11*) and update the `texture.jpg` file to use the new `spaceship_texture.jpg`. You will also have to create a brand new texture in the same way as last time (drag `terrain_texture.jpg` onto the library and then edit its properties, calling it `terrainTextureBitmap` so it can be accessed from the code).



Time for action – adding the score to the GUI

Next, let's upgrade our GUI to include a score display as seen in virtually all video games. Only a few minor edits are required. We do this as follows:

```

private function updateScore():void
{
    // for now, you earn points over time
    score++;
    // padded with zeroes
    if (score < 10) scoreTf.text = 'Score: 00000' + score;
    else if (score < 100) scoreTf.text = 'Score: 0000' + score;
    else if (score < 1000) scoreTf.text = 'Score: 000' + score;
    else if (score < 10000) scoreTf.text = 'Score: 00' + score;
    else if (score < 100000) scoreTf.text = 'Score: 0' + score;
    else scoreTf.text = 'Score: ' + score;
}

private function initGUI():void
{
    // a text format descriptor used by all gui labels

```

```
var myFormat:TextFormat = new TextFormat();
myFormat.color = 0xFFFFFFFF;
myFormat.size = 13;

// create an FPSCounter that displays the framerate on screen
fpsTf = new TextField();
fpsTf.x = 0;
fpsTf.y = 0;
fpsTf.selectable = false;
fpsTf.autoSize = TextFieldAutoSize.LEFT;
fpsTf.defaultTextFormat = myFormat;
fpsTf.text = "Initializing Stage3d...";
addChild(fpsTf);

// create a score display
scoreTf = new TextField();
scoreTf.x = 560;
scoreTf.y = 0;
scoreTf.selectable = false;
scoreTf.autoSize = TextFieldAutoSize.LEFT;
scoreTf.defaultTextFormat = myFormat;
scoreTf.text = "000000";
addChild(scoreTf);

// add some labels to describe each shader
var label1:TextField = new TextField();
label1.x = 100;
label1.y = 180;
label1.selectable = false;
label1.autoSize = TextFieldAutoSize.LEFT;
label1.defaultTextFormat = myFormat;
label1.text = "Shader 1: Textured";
addChild(label1);

var label2:TextField = new TextField();
label2.x = 400;
label2.y = 180;
label2.selectable = false;
label2.autoSize = TextFieldAutoSize.LEFT;
label2.defaultTextFormat = myFormat;
label2.text = "Shader 2: Vertex RGB";
addChild(label2);

var label3:TextField = new TextField();
```

```

label3.x = 80;
label3.y = 440;
label3.selectable = false;
label3.autoSize = TextFieldAutoSize.LEFT;
label3.defaultTextFormat = myFormat;
label3.text = "Shader 3: Vertex RGB + Textured";
addChild(label3);

var label4:TextField = new TextField();
label4.x = 340;
label4.y = 440;
label4.selectable = false;
label4.autoSize = TextFieldAutoSize.LEFT;
label4.defaultTextFormat = myFormat;
label4.text = "Shader 4: Textured + setProgramConstants";
addChild(label4);
}

```

What just happened?

Again, only a few lines of code have changed. We have created a new text field on the screen that will be used to display the score. A routine that is called every frame (during the render loop) updates the score. As we don't have any enemies to destroy or coins to pick up for now, the score will simply increase over time. Eventually, of course, the `updateScore` function will change to reflect the deeper gameplay we are soon to add.

Time for action – upgrading your init routines

Next, we want to enhance our initialization routines to reflect the new functionality we are implementing. First let's put our mipmap creation routines into a function, so that our code is cleaner.

```

public function uploadTextureWithMipmaps(
    dest:Texture, src:BitmapData):void
{
    var ws:int = src.width;
    var hs:int = src.height;
    var level:int = 0;
    var tmp:BitmapData;
    var transform:Matrix = new Matrix();
    var tmp2:BitmapData;

    tmp = new BitmapData( src.width, src.height, true, 0x00000000 );

```

```
while ( ws >= 1 && hs >= 1 )
{
    tmp.draw(src, transform, null, null, null, true);
    dest.uploadFromBitmapData(tmp, level);
    transform.scale(0.5, 0.5);
    level++;
    ws >>= 1;
    hs >>= 1;
    if (hs && ws)
    {
        tmp.dispose();
        tmp = new BitmapData(ws, hs, true, 0x00000000);
    }
}
tmp.dispose();
}
```

Next, we should upgrade our onContext3DCreate function as follows.

```
private function onContext3DCreate(event:Event):void
{
    // Remove existing frame handler. Note that a context
    // loss can occur at any time which will force you
    // to recreate all objects we create here.
    // A context loss occurs for instance if you hit
    // CTRL-ALT-DELETE on Windows.
    // It takes a while before a new context is available
    // hence removing the enterFrame handler is important!

    if (hasEventListener(Event.ENTER_FRAME))
        removeEventListener(Event.ENTER_FRAME,enterFrame);

    // Obtain the current context
    var t:Stage3D = event.target as Stage3D;
    context3D = t.context3D;

    if (context3D == null)
    {
        // Currently no 3d context is available (error!)
        return;
    }

    // Disabling error checking will drastically improve performance.
    // If set to true, Flash sends helpful error messages regarding
    // AGAL compilation errors, uninitialized program constants, etc.
    context3D.enableErrorChecking = true;
```

Remember to set the above line to true when you are done programming your game—this will increase your frame rate. Additionally, compiling your SWF in the "release" mode as opposed to the "debug" mode has a huge effect upon frame rate.

```
// Initialize our mesh data
initData();

// The 3d back buffer size is in pixels (2=antialiased)
context3D.configureBackBuffer(swfWidth, swfHeight, 2, true);

// assemble all the shaders we need
initShaders();

myTexture = context3D.createTexture(
    textureSize, textureSize,
    Context3DTextureFormat.BGRA, false);
uploadTextureWithMipmaps(
    myTexture, myTextureData.bitmapData);

terrainTexture = context3D.createTexture(
    textureSize, textureSize,
    Context3DTextureFormat.BGRA, false);
uploadTextureWithMipmaps(
    terrainTexture, terrainTextureData.bitmapData);
```

In the preceding code, we are using our new mipmap generation function to upload the two textures our demo requires in all the sizes required.

```
// create projection matrix for our 3D scene
projectionmatrix.identity();
// 45 degrees FOV, 640/480 aspect ratio, 0.1=near, 100=far
projectionmatrix.perspectiveFieldOfViewRH(
    45.0, swfWidth / swfHeight, 0.01, 5000.0);

// create a matrix that defines the camera location
viewmatrix.identity();
// move the camera back a little so we can see the mesh
viewmatrix.appendTranslation(0,0,-3);

// tilt the terrain a little so it is coming towards us
terrainviewmatrix.identity();
terrainviewmatrix.appendRotation(-60,Vector3D.X_AXIS);

// start the render loop!
addEventListener(Event.ENTER_FRAME,enterFrame);
}
```

What just happened?

The preceding routines are very similar to previous versions of our demo, except that the texture mipmap uploading has been turned into a function (since it is called more than once). We initialize our two textures here, but otherwise everything else remains unchanged.

The same four shaders (vertex and fragment programs) are being used in this demo as in the last, except they are being used to render more complex geometry, so leave the `initShaders()` function alone.

Time for action – parsing our mesh data

Finally, our `initData` function has changed significantly. Instead of manually defining thousands of vertexes in a massive array in code, we simply create two new objects. These objects are created in the `Stage3dObjParser.as` file we created earlier.

```
private function initData():void
{
    // parse the OBJ file and create buffers
    myMesh = new Stage3dObjParser(
        myObjData, context3D, 1, true, true);
    // parse the terrain mesh as well
    terrainMesh = new Stage3dObjParser(
        terrainObjData, context3D, 1, true, true);
}
```

What just happened?

In the constructor for this new `Stage3dObjParser` class, the `.OBJ` data that was embedded into our Flash game is parsed. All this data is stored in temporary variables, ready for action. As explained earlier in this chapter when we programmed the `.OBJ` file parser class, the first time the vertex buffers are requested during the render loop, fresh `VertexBuffer3Ds` will be uploaded to your video card.

That is it for all the setup!

Time for action – animating the scene

Now we will animate our scene. First, let's define a function that renders the terrain as follows:

```
private function renderTerrain():void
{
    context3D.setTextureAt(0, terrainTexture);
```

```

// simple textured shader
context3D.setProgram ( shaderProgram1 );
// position
context3D.setVertexBufferAt(0, terrainMesh.positionsBuffer,
    0, Context3DVertexBufferFormat.FLOAT_3);
// tex coord
context3D.setVertexBufferAt(1, terrainMesh.uvBuffer,
    0, Context3DVertexBufferFormat.FLOAT_2);
// vertex rgba
context3D.setVertexBufferAt(2, terrainMesh.colorsBuffer,
    0, Context3DVertexBufferFormat.FLOAT_4);
// set up camera angle
modelmatrix.identity();
// make the terrain face the right way
modelmatrix.appendRotation( -90, Vector3D.Y_AXIS);
// slowly move the terrain around
modelmatrix.appendTranslation(
    Math.cos(t/300)*1000,Math.cos(t/200)*1000 + 500,-130);
// clear the matrix and append new angles
modelViewProjection.identity();
modelViewProjection.append(modelmatrix);
modelViewProjection.append(terrainviewmatrix);
modelViewProjection.append(projectionmatrix);
// pass our matrix data to the shader program
context3D.setProgramConstantsFromMatrix(
    Context3DProgramType.VERTEX,
    0, modelViewProjection, true );
context3D.drawTriangles(terrainMesh.indexBuffer,
    0, terrainMesh.indexBufferCount);
}

```

Finally, we need to upgrade our `enterFrame` function to display our newly improved scene.

```

private function enterFrame(e:Event) :void
{
    // clear scene before rendering is mandatory
    context3D.clear(0,0,0);
    // move or rotate more each frame
    t += 2.0;
    // scroll and render the terrain once
    renderTerrain();
    // how far apart each of the 4 spaceships is
    var dist:Number = 0.8;
    // loop through each mesh we want to draw
    for (looptemp = 0; looptemp < 4; looptemp++)
    {

```

Here, we are looping through four copies of the same mesh in order to demonstrate the four different shaders we programmed in the previous chapter.

```
// clear the transformation matrix to 0,0,0
modelmatrix.identity();
// each mesh has a different texture,
// shader, position and spin speed
switch(looptemp)
{
    case 0:
        context3D.setTextureAt(0, myTexture);
        context3D.setProgram ( shaderProgram1 );
        modelmatrix.appendRotation(t*0.7, Vector3D.Y_AXIS);
        modelmatrix.appendRotation(t*0.6, Vector3D.X_AXIS);
        modelmatrix.appendRotation(t*1.0, Vector3D.Y_AXIS);
        modelmatrix.appendTranslation(-dist, dist, 0);
        break;
    case 1:
        context3D.setTextureAt(0, null);
        context3D.setProgram ( shaderProgram2 );
        modelmatrix.appendRotation(t*-0.2, Vector3D.Y_AXIS);
        modelmatrix.appendRotation(t*0.4, Vector3D.X_AXIS);
        modelmatrix.appendRotation(t*0.7, Vector3D.Y_AXIS);
        modelmatrix.appendTranslation(dist, dist, 0);
        break;
    case 2:
        context3D.setTextureAt(0, myTexture);
        context3D.setProgram ( shaderProgram3 );
        modelmatrix.appendRotation(t*1.0, Vector3D.Y_AXIS);
        modelmatrix.appendRotation(t*-0.2, Vector3D.X_AXIS);
        modelmatrix.appendRotation(t*0.3, Vector3D.Y_AXIS);
        modelmatrix.appendTranslation(-dist, -dist, 0);
        break;
    case 3:
        context3D.setProgramConstantsFromVector(
            Context3DProgramType.FRAGMENT, 0, Vector.<Number>
            ([ 1, Math.abs(Math.cos(t/50)), 0, 1 ]) );
        context3D.setTextureAt(0, myTexture);
        context3D.setProgram ( shaderProgram4 );
        modelmatrix.appendRotation(t*0.3, Vector3D.Y_AXIS);
        modelmatrix.appendRotation(t*0.3, Vector3D.X_AXIS);
        modelmatrix.appendRotation(t*-0.3, Vector3D.Y_AXIS);
        modelmatrix.appendTranslation(dist, -dist, 0);
        break;
}
```

Next, we need to set up the camera, instruct Flash which textures and buffers to use and render each mesh.

```
// clear the matrix and append new angles
modelViewProjection.identity();
modelViewProjection.append(modelmatrix);
modelViewProjection.append(viewmatrix);
modelViewProjection.append(projectionmatrix);

// pass our matrix data to the shader program
context3D.setProgramConstantsFromMatrix(
    Context3DProgramType.VERTEX,
    0, modelViewProjection, true );

// draw a spaceship mesh
// position
context3D.setVertexBufferAt(0, myMesh.positionsBuffer,
    0, Context3DVertexBufferFormat.FLOAT_3);
// tex coord
context3D.setVertexBufferAt(1, myMesh.uvBuffer,
    0, Context3DVertexBufferFormat.FLOAT_2);
// vertex rgba
context3D.setVertexBufferAt(2, myMesh.colorsBuffer,
    0, Context3DVertexBufferFormat.FLOAT_4);
// render it
context3D.drawTriangles(myMesh.indexBuffer,
    0, myMesh.indexBufferCount);
}
```

Now that all four meshes (and the terrain) have been rendered, we simply get Flash to display the finished image on the screen and update our FPS counter and GUI.

```
// present/flip back buffer
// now that all meshes have been drawn
context3D.present();

// update the FPS display
fpsTicks++;
var now:uint = getTimer();
var delta:uint = now - fpsLast;
// only update the display once a second
if (delta >= 1000)
{
    var fps:Number = fpsTicks / delta * 1000;
    fpsTf.text = fps.toFixed(1) + " fps";
```

```
    fpsTicks = 0;
    fpsLast = now;
}

// update the rest of the GUI
updateScore();
}

} // end of class
} // end of package
```

What just happened?

In our current project, the render loop is performed each frame during the `enterFrame` event.

We created a new function named `renderTerrain` which scrolls the terrain over time. It first instructs `Stage3D` how to draw the mesh using the `setTextureAt` and `setProgram` functions. Once we have instructed Flash which texture and which shader (vertex and fragment shader) to use, we must instruct Flash where the data for our terrain geometry is located. We do this by using `setVertexBufferAt()` for each buffer.

As mentioned, accessing `terrainMesh.positionsBuffer` and the other buffers the first time forces our `Stage3dObjParser` to upload the data to the video card. On all subsequent requests, it simply sends a buffer number to `Stage3D`, which is very efficient.

Next, we update the camera angles and model position, and send the final matrix to `Stage3D` using the `setProgramConstantsFromMatrix` function, so that it knows "where" to render the terrain. Finally it renders the mesh by calling `drawTriangles()`.

The rest of the render loop is virtually unchanged, except the four squares from our previous demo are replaced with nicely spaced duplicates of the spaceship mesh. Lastly, the GUI is updated to show the FPS and the new score.

Quest complete—time to reap the rewards

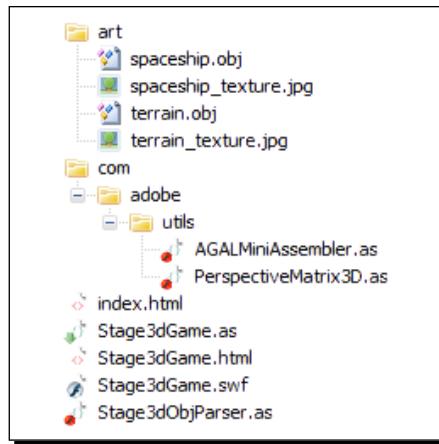
You have now upgraded your 3D demo with improved graphics and more advanced animation. It is really starting to look like a game!

This is certainly feeling like an epic quest, isn't it? Not only have we successfully overcome some serious boss battles (such as learning AGAL), but more importantly, the rewards are starting to be more obvious. There is undoubtedly treasure at the bottom of this dungeon.

The next step? More advanced special effects, cool new texture formats and rendering modes, transparent textures that are great for special effects like explosions or magic and much more. We have some functional 3D graphics, so the next step should be to really make it look pretty!

Folder structure

As you are now dealing with multiple files in multiple folders, just for a reference this is what your project folder should look like. As you can see, now that we are embedding multiple art assets, they have been moved into their own folder to keep things clean.



Pop quiz – earn some experience points

Here are a few quick questions to test your understanding. If you can answer them, then you are officially ready to move on to the next step in your grand adventure.

1. What is the simplest way to add movement in your game?
 - a. Program an AGAL vertex program that tweaks vertex coordinates
 - b. Pre-render all possible positions into a giant texture
 - c. Update the model's position every frame in the render loop
 - d. Beg and plead to the Molehill spirits

2. Before rendering a mesh using `drawTriangles()`, what function would you use to instruct it which vertex x,y,z buffer to use?
 - a. `context3D.setProgram`
 - b. `context3D.setVertexBufferAt`
 - c. `context3D.setProgramConstantsFromMatrix`
 - d. If I lift weights, I'm sure to get buffer
3. What is the most common way to set up a render loop?
 - a. Perform all updates in an `enterFrame` event
 - b. Run the same function over and over in a `while()` loop
 - c. Copy the same line of code to every frame in the timeline
 - d. Keep refreshing the web page as fast as I can

Have a go hero – a fun side-quest: make some more models!

Your side quest is to upgrade the render loop to include some buildings, turrets, rocks, or trees on the ground. Create a third mesh and texture, add it to your project, and draw it in the `renderTerrain` function along with the ground.

For added challenge, render the mesh multiple times. Just like the four spaceships on screen, you can render the same mesh using different shaders and at different positions. This means that you could, if you wanted, create a tree and render it in dozens of different locations to simulate a lush forest!

Summary

We were victorious in completing the next quest along the way to the creation of our own 3D video game in Flash. We learned all about the render loop, how to parse .OBJ mesh data, how to create vertex buffers, how to render more than one mesh at once, and ways of animating your meshes to make them move over time.

We have created a pretty impressive 3D demo, complete with scrolling terrain and a functional GUI. From now on, our Flash project will have transcended from being a mere demo to feeling more and more like a game as we adventure onwards.

Level 5 achieved!

Congratulations. You have successfully "levelled up" again. As a level five Molehill master, you are now very strong. You have all the skills required to write a fully functional video game in Flash.

6

Textures: Making Things Look Pretty

Prepare for Level 6.

Your game engine now boasts shaders, meshes, movement, a GUI, and all the inits required to get Stage3D rendering fast. It is time to take it to the next level and experiment with advanced texture effects.

So far so good. We have achieved a lot in our quest to create a 3D video game in Flash. Now that the basics have been taken care of, we can start to focus on making things really look good. From shiny models such as metallic weapons to transparent effects such as steam, smoke, or explosions, the Stage3D API can render textures in all sorts of different ways.

In this chapter, we are going to learn how to process bitmap data, how to convert textures into specialized formats that are useful for effects, as well as being efficient on limited hardware. In this chapter, we are going to learn about:

- ◆ Using multiple textures and several meshes
- ◆ Rendering textures with special transparent blend modes
- ◆ Rendering special effects such as explosions
- ◆ Changing whether backfaces are culled on meshes
- ◆ Considering the zbuffer when rendering
- ◆ Detecting key presses for user control of the action

Time for a plan: creating a "real" game

We are about to move on from mere tech demo to a real game. Now that we have ventured deep "into the Molehill", it is a good time for you to start brainstorming what kind of game you want to make with the Stage3D API.

You probably had something in mind from the very outset, but if not, now that there is light at the end of the tunnel you might as well come up with a plan.

An important consideration at this stage is to make your first 3D game as simple as possible. As any veteran developer knows, even things that seem simple turn out much more complex than expected. With this in mind, pick a genre of game that you feel would take a minimum of work; something arcade-inspired, simplistic, and easy to make. Your future games can each grow in size and complexity, so for now, start visualizing the most basic game you would want to make.

At this stage, success is defined by simply finishing your game so you can show it off to your friends. It would be naïve to try to write a **massively multiplayer online role-playing game (MMORPG)** for example. Once you have reached the milestone of having a finished playable 3D game under your belt, you can quickly and easily move on to improved games.

With simplicity in mind, the example game that will be created over the remainder of this book is an arcade shooter. Inspired by Galaxian, 1942, R-Type, Giga Wing, Ikaruga, Tiger Heli, Raiden, Geometry Wars, and even Space Invaders. Many people call these kinds of games "bullet hell" games.

An arcade shoot-em-up (SHMUP) is perfectly suited for our upcoming experiments, and free of complex technical requirements such as physics engines, advanced artificial intelligence, or network communications. They feature tons of action, lots of explosions, and eye-candy galore.

From now on, while creating the content for the Stage3D demo project that we keep upgrading each chapter, choose textures and meshes that you actually want to appear in your game.

Using textures in Stage3D

The basic order of operations to use a texture is as follows:

1. Embed an image into the SWF
2. Create a `Texture` object in code
3. Upload the texture to the video RAM
4. Generate mipmaps and upload them too

5. Create an AGAL vertex program that uses texture coordinates
6. Create an AGAL fragment program that samples a texture
7. Instruct Flash to use this texture during a render

In order to embed an image in Flash, you will either use the `[embed]` flex command or you will add it to your library in the Flash IDE. You then create a `Texture` object by instantiating a new class of this type, with the `context3D.createTexture()` function.

In order to send the bitmap image data to your video card for use during rendering, you have to upload it and optionally create mipmaps (small versions of it used for perspective correction) and upload them too, using your `Texture.uploadFromBitmapData()` function.

Next, you need to compile AGAL programs that take advantage of this data. In your fragment program, you will use a command similar to `"tex ft0, v1, fs0 <2d,linear,repeat,miplinear>"` to sample the texture to come up with the proper color.

Finally, in your render loop, you would use the `context3D.setTextureAt()` function to select this texture prior to rendering your mesh with the `context3D.drawTriangles()` function.

Power-of-two

In previous chapters, we glossed over the use of textures and presented the simplest possible explanation of how things are done. We also used only a single texture in our demo, and it was always 512x512 in size. Why did we choose this strange sounding size for our image?

Although not all video cards insist that textures be of a certain size anymore, the math involved in rendering textures in 3D runs faster when you use textures of certain sizes. On some machines, textures won't even work unless they are in a particular range of sizes. Therefore, it is essential that all textures you use for your game are sized as powers-of-two.

What does this mean? It means that textures must have a length and width that fall into these values: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, and so on.

Additionally, in many situations your textures have to be square (so the height must equal the width) rather than being rectangular. Sometimes, you can get away with a 64x256 texture, but in general, you are forced to use square, power-of-two sized textures, such as images that are 256x256 pixels in size.

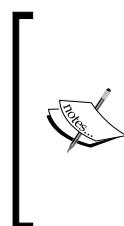
The larger your texture, the more video RAM it will take up and on certain hardware it will also render a bit slower. Therefore, it is always a wise choice to use textures that are just big enough to look good.

When creating a pair of dice, for example, if you know they will never take up the entire screen it would be very silly to use a gigantic 1024x1024 texture for them. Tiny objects in your world such as these will look exactly the same with a 64x64 or 128x128 sized texture. Important objects, and textures used for walls and floors that might be viewed up close, are ideal candidates for larger texture; 256x256 or 512x512 being the norm.

There is nothing wrong with using a few really big textures, but as with all things in game programming, the leaner it is the faster it runs. If every texture used by your game is 1024x1024, your .SWF file will require a really long download and you might even run out of texture RAM.

Therefore, when creating textures, work in high resolution (4096x4096), but then save it as 128x128 for use in your game. During testing, if you find that the texture in question is looking blurry, try 256x256 or 512x512. As the source art is saved in high-def format, you have extra pixels to work with and when you resize your assets the resulting image won't get blurry.

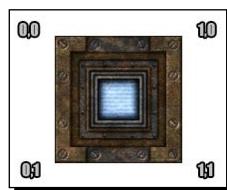
It is always wise to work in high resolution but export in low resolution and increase the fidelity only when you notice that it is required. This will help keep your Flash files tiny and your game running smoothly.



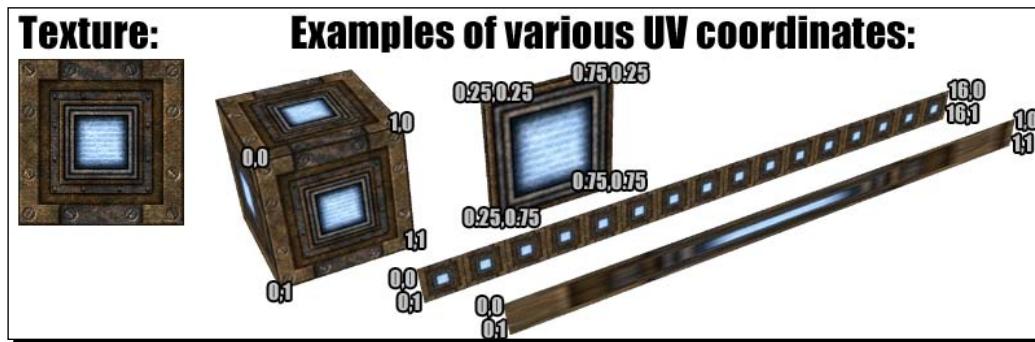
It is a common mistake to include textures that are larger than required: naturally, you want your game to look as good as possible. Therefore, a good metric that will help you avoid this is to estimate the maximum amount of in-game screen real estate your mesh will use. If the texture in question is meant to cover a tiny in-game object such as a rivet or lightswitch that will never take more than 50 pixels on the screen, there is no reason to use a texture that is 1024x1024; 64X64 will suffice in this example.

UV coordinates

As you know, a texture is an image that is wrapped around a 3D mesh, much like wrapping paper on a birthday present. How it gets wrapped is controlled by UV coordinates, which are uploaded as a vertex buffer alongside x,y,z coordinates for each vertex. The UV coordinate for any given vertex is defined by two numbers (u and v) and is typically in the range 0 to 1. The coordinate U is the horizontal axis in a texture, while V represents the vertical axis. Therefore, the bottom of a texture has a V of 1 and the far right corner would have a U of 1.



For example, in the simple rotating square mesh we used in the first few chapters, the top left corner of the square (the first vertex) had the UV coordinate defined as 0,0. This means that the pixels that are drawn on the screen by your fragment shader come from the top left corner of your texture image. The bottom right corner of this bitmap would be accessed by giving a vertex a UV of 1,1.



You can use numbers higher than one. For example, if you wanted a texture to repeat 16 times horizontally across the surface of a polygon, you could give a vertex UV coordinates of (16,1), so that at that particular vertex the texture would have tiled 16 times in the U and once in the V axis. You can also use numbers that are less than one. For example, if you start from 0.25, then you are starting a quarter of the way across the texture. This happens in many meshes when you want a face to only use a small portion of the total texture.

Transparent textures

You don't have to use a .JPG image for your source image. You can also use a .PNG file, complete with variable transparency. This can be really great for effects such as explosions, sparks and smoke, as well as see-through windows. Additionally, you can simulate complex geometry (such as a fence, or a grate, or leaves and branches of a tree) by drawing complex shapes on your texture but using a single quad (square polygon).



The preceding texture is completely see-through in the areas not covered by the green leaf. It is a .PNG file that has been saved to include alpha (transparency) information. Instead of modeling a mega-complex tree where each leaf is comprised of hundreds of polygons, you can simulate complex geometry by using a texture such as this on a simple quad. Groups of these simple "billboard sprites" can be used to great effect by giving the impression of a scene with millions of polygons where in actual fact only a few dozen are being used.

Animating UV coordinates in a shader

One useful technique that is employed in many games is to scroll or wobble the texture UVs of a mesh based on time. For example, you could tile a texture that looks like water on a strip of polygons in the shape of a river and then, instead of actually moving the model, simply increment the UV coordinates a little each frame.

In order to do this, you would write a short AGAL vertex program that accepts a vertex constant that you set for each frame. Before outputting the UV coordinates, you would use the ADD opcode to offset the UVs in one or both dimensions, so that each frame the part of the texture that gets used in your fragment program is taken from a different part of your texture.

In this flowing water example, then, all we would need to do is add a little code to our render loop. First, we would increment a variable a little each frame, and send this value to the vertex program with the `setProgramConstantsFromVector()` method.

For example, assuming that during inits you created a Vector list of Numbers called `uvOffsets`, during your render loop, you could do something like this to scroll the texture horizontally a little bit each frame:

Time for action – updating UV coordinates each frame

```
// increment the first Number in our list
// which the fragment shader will add to the UV coords
uvOffsets[0] += 0.01;
// send this vector to the AGAL variable vc4
context3D.setProgramConstantsFromVector(
    Context3DProgramType.VERTEX,
    4, uvOffsets, true );
```

Then, in your AGAL vertex program, you would add this vector to the stored UV coordinates that were uploaded along with your mesh. Assuming that they are stored in AGAL register va1 and are sent to your fragment program in v1, we first add the value we set earlier (in vc4) to those default coordinates so that v1 now contains the new coordinates.

```
add v1, va1, vc4
```

What just happened?

For each frame, just before we render our mesh, we update a value that is sent to Stage3D to be used in our vertex program as vc4. This value is added to the existing UV data for our mesh, so that for each frame the U (horizontal) coordinate is shifted a little to the side. This technique can be used to shift textures constantly, such as when scrolling textures like water a little bit each frame, or when animating textures or using texture atlases to select which "frame" (or part of the texture) we want to use.

Texture atlases

A great technique to increase your game's framerate and allow more meshes to be rendered using the same shader is to pack multiple textures into a large bitmap called a texture atlas. For example, imagine you need 16 different kinds of gem for treasure as playing pieces in your game. Instead of creating 16 tiny textures, you could use Photoshop or any image editor to pack them into one larger texture and then model various 3D meshes in such a way that different models use different UV coordinates in the same texture.

This is great for framerate because you can batch a whole bunch of differently textured models into one giant `drawTriangles` call. Additionally, there is a small overhead each time you switch textures or shaders when rendering, and if you can get away with rendering many meshes in between calls to `context3D.setTextureAt()`, you can save time which results in better fps.

Using texture atlases is definitely the best practice for rendering fonts in your 3D world. Almost every game that needs letters rendered in 3D (as opposed to simply overlaid on top of your Stage3D) uses a bitmap font texture atlas.



As you can see, the preceding texture is a more efficient way of texturing quads for use as text inside your 3D world, compared to the alternative, which would be to create and upload a hundred or more individual tiny textures, one for each letter. As it is slower to switch active textures than it is to use the same texture for many meshes, using texture atlases will improve performance.

Animated textures

There are many ways to code an animated texture and many situations where this is the perfect method to achieve an awesome effect. Imagine, for example, that you wanted to "project" a streaming video or cool-looking animation onto a surface in your game. This could be in a virtual movie theater, or for the screen of some futuristic video watch, smartphone or other special effect.

One way to do this would be to simply upload a series of textures and loop through them based on time passing. For example, in your render loop you could switch the texture you send to `context3D.setTextureAt()` every few frames. This is handy for smaller animations such as looped water caustics, screen static, or even for simple explosions.

You could also create a texture atlas that is comprised of multiple frames of video so that the entire animation is stored on a single texture. Then, simply use the UV animation technique above to update the UV coordinates of a quad over time. Take this texture, for example:



To use a texture like this, you would first create a simple quad mesh (a four vertex square) with UV coordinates that cover the entire square with only the first frame of the animation. Then, using the AGAL example above, we simply send a new offset in a `setProgramConstantsFromVector()` call to shift the texture coordinates over to the next frame. In the example image above, each frame is $1 / 13$ th offset from each other, so frame one would use a U of 0.0 while each frame after that would need to increment the U coordinate by about 0.077.

Manipulating texture data

For animated textures, which are derived from videos or even Flash animations, it might take up too much video RAM to store each and every frame ahead of time. In this case, you would need to upload pixels each frame by copying the bitmap data you want for your texture and repeatedly calling `uploadFromBitmapData()` whenever the frame has changed.

This is really bad for FPS, especially if you need to generate mipmaps each frame, so it is best to avoid this technique unless your project absolutely needs it. When used sparingly, this technique opens up all sorts of interesting possibilities for "dynamic textures".

As you can update your textures whenever you want, an infinite variety of cool effects become available to you. For example, you could generate dynamic textures using Flash's perlin noise functions, fractals and bitmap filters. Anything in AS3 code you have around that can generate bitmap data that can be used as the source pixels for your textures.

Not only could you copy the bitmap data from each frame of an .FLV video, but you could also capture images from a webcam and put them into your 3D world! The possibilities are endless.

Render states

In the same way that textures can be set to be rendered in different ways, polygons also follow default rules that are begging to be broken! Flash defaults to render meshes in the fastest and most efficient way possible. This is generally what we want, but naturally a game would not have any pizazz if it did not take advantage of some special effects. You can change the "render state" in the Stage3D API to take advantage of custom behaviors that open up an infinite number of rendering possibilities.

Backface culling

Backface culling is the process in which meshes are rendered with only one side of each polygon being visible. This helps to speed up rendering as in, for example, a cube with opaque sides you would never be able to see the inside walls. When backface culling is turned on, which is the default, these inside faces (the backfaces) are never rendered. Generally, this works great but there are times when you want both sides of every polygon to be drawn.

Time for action – rendering a mesh's backfaces

For example, if you are using transparent textures on a cube in this example, you will probably want to see the inside walls behind the ones that are closer to the camera. In order to do this, you simply need to instruct Flash to render both sides of the mesh as follows:

```
// do not cull any faces: draw both sides  
// of every polygon including backfaces  
context3D.setCulling(Context3DTriangleFace.NONE);
```

What just happened?

We instructed the Stage3D API to stop culling (skipping the drawing of) the backfaces of whatever mesh we render next. This means that the "inside" faces will be drawn along with those that are pointing "out".

The different parameters allowed for the `setCulling` function are as follows:

- ◆ `Context3DTriangleFace.NONE` (do not cull anything)
- ◆ `Context3DTriangleFace.BACK` (the default, where backfaces are culled)
- ◆ `Context3DTriangleFace.FRONT` (opposite from normal behavior, where front faces would not be drawn)
- ◆ `Context3DTriangleFace.BOTH` (which is practically useless since both sides of every face in your polygon would be culled, resulting in nothing being rendered)

This kind of setting is called a "render state". Once you set it to something, it stays in effect until you set it to something else. You only need to call it once beforehand if you are about to render several meshes in the same way.

Depth testing

When Flash is rendering your scene, how far away each polygon is from the camera matters. For example, a wall that is near the camera will intuitively obscure anything behind it. In order to detect what is in front and behind what, each time a pixel is rendered to the Stage3D the r,g,b colors are stored alongside a depth component. The data that stores the depth of each pixel on the screen is called the depth buffer (or the zbuffer).

The default in Stage3D is to only render pixels that are closest to the camera, which makes perfect sense as evidenced by the preceding wall example. However, there are many times when you do not want to store any depth information for a particular mesh (for example, when using special effects such as explosions, or transparent regions like a see-through window) or when you do not want to check the depth buffer when rendering (for x-ray vision, where you can see a mesh even if it is completely obscured by a wall).

Time for action – making a mesh not affect the zbuffer

In order to change how Flash deals with depth testing, you can instruct it to either not write to the zbuffer or not check it, or any combination of the two, by using the `setDepthTest` function.

```
context3D.setDepthTest(false, Context3DCompareMode.LESS);
```

What just happened?

This function takes two parameters. The first is a Boolean (true or false) value that is true if you want the next mesh you render to write to the depth buffer. The second parameter can be one of the following:

- ◆ Context3DCompareMode.ALWAYS
- ◆ Context3DCompareMode.EQUAL
- ◆ Context3DCompareMode.GREATER
- ◆ Context3DCompareMode.GREATER_EQUAL
- ◆ Context3DCompareMode.LESS
- ◆ Context3DCompareMode.LESS_EQUAL
- ◆ Context3DCompareMode.NEVER
- ◆ Context3DCompareMode.NOT_EQUAL

The pixel in question will only be rendered when a comparison between it and the destination pixel are TRUE based on the preceding criteria. The default in Stage3D is to only render pixels that have a depth value that is LESS than what is already on the screen, closer to the camera.

The preceding example is handy if you are going to render a transparent mesh like a window that you don't want to draw on the zbuffer (so that other objects are always visible behind it), but you DO want it to be obscured by furniture in the room that is in front of it.

Like with other render states, this will be in effect until you set it to something else. You only need to call it once and that criteria will be used until you instruct Flash otherwise.

Blend modes

The Stage3D API provides a powerful array of blend modes, which affect how a texture is rendered. For example, you can instruct it to draw a texture using transparency, or to mix the colors in your texture with those behind it, so that they lighten or darken the screen. These kinds of additive, multiplicative, or subtractive blend modes are how particle systems such as smoke, fire, explosions, sparks, and the like are drawn in modern games.

Before you draw a mesh, you can use the `Context3D.setBlendFactors()` function to instruct Flash how you want to draw the texture.

In order to set a blend mode, you send two blend factors to the function. The first is the source factor and the second is the destination factor. The resulting pixel on the screen is calculated as follows: $\text{framebuffercolor} = (\text{fragmentcolor} * \text{srcBlend}) + (\text{framebuffercolor} * \text{destBlend})$.

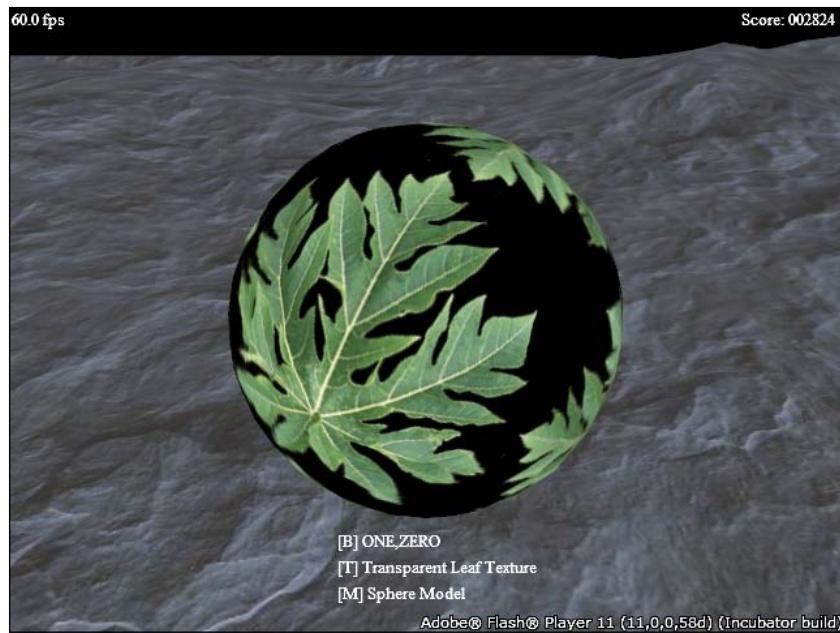
In the following *Time for action* sections, there are three common examples of blend modes in action: rendering an opaque mesh, rendering a mesh with transparent regions (such as a leaf), and rendering a mesh so that it lightens the scene (such as an explosion).

Time for action – rendering an opaque mesh

```
// this is the Molehill default: nice for opaque textures
// all of the color on screen comes from the fragment color
context3D.setBlendFactors(
    Context3DBlendFactor.ONE,
    Context3DBlendFactor.ZERO);
```

What just happened?

We instructed Stage3D to take all of the color from the result of your fragment shader and to omit color from the framebuffer (what is behind the mesh). It will look similar to the following screenshot:

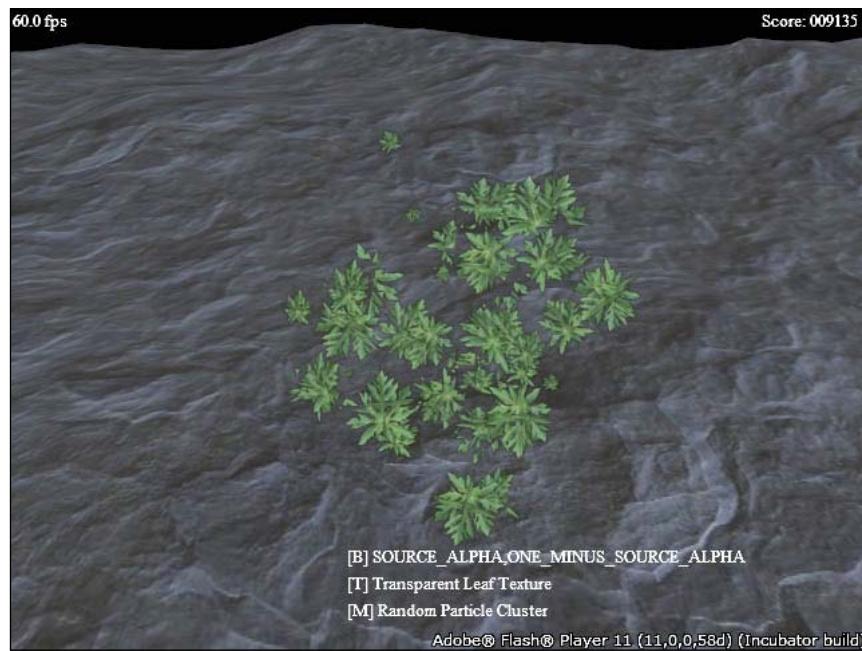


Time for action – rendering a mesh with transparent regions

```
// this is the proper setting to use .png textures
// that have transparent regions
// perfect for foliage/fences/fonts
context3D.setBlendFactors(
    Context3DBlendFactor.SOURCE_ALPHA,
    Context3DBlendFactor.ONE_MINUS_SOURCE_ALPHA);
```

What just happened?

We just instructed Stage3D that the next mesh to be rendered should take all color information from the output of our fragment shader wherever the alpha (transparency) in our texture is pure white, and blend in a color from the framebuffer behind the mesh in an amount proportional with how black a pixel in our texture's alpha channel is. In the example of our leaf image, the green leafy bits have a pure white alpha and the transparent regions in the texture would be pure black if we were to look only at the alpha data encoded into the image.

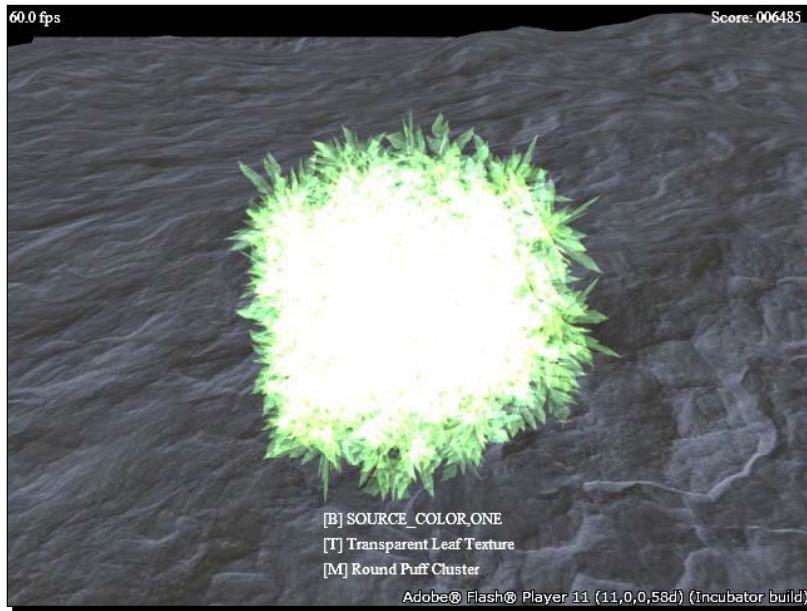


Time for action – rendering a mesh so it lightens the scene

```
// this setting makes it lighten the scene only
// good for particles stored as .jpg
// (only white sections add to the screen - black is not drawn)
context3D.setBlendFactors(
    Context3DBlendFactor.SOURCE_COLOR,
    Context3DBlendFactor.ONE);
```

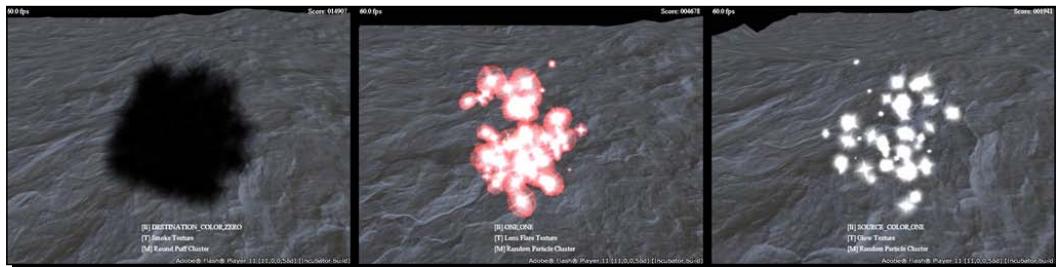
What just happened?

We just instructed Stage3D that we want the next mesh rendered to add the pixel color of our texture to the scene, so that each pixel rendered will bring the image closer to pure white.



Creative use of blend modes is essential for particle systems and other special effects. Here is one last example of a few of the many possible blend modes.

The following three images depict blend modes of DESTINATION_COLOR , ZERO , ONE , ONE , and SOURCE_COLOR , ONE respectively:



As parameters to the `Context3D.setBlendFactors()` function you have the following choices, which are all of the `Context3DBlendFactor` type:

- ◆ `Context3DBlendFactor.DESTINATION_ALPHA`
- ◆ `Context3DBlendFactor.DESTINATION_COLOR`
- ◆ `Context3DBlendFactor.ONE`
- ◆ `Context3DBlendFactor.ONE_MINUS_DESTINATION_ALPHA`
- ◆ `Context3DBlendFactor.ONE_MINUS_DESTINATION_COLOR`
- ◆ `Context3DBlendFactor.ONE_MINUS_SOURCE_ALPHA`
- ◆ `Context3DBlendFactor.ONE_MINUS_SOURCE_COLOR`
- ◆ `Context3DBlendFactor.SOURCE_ALPHA`
- ◆ `Context3DBlendFactor.SOURCE_COLOR`
- ◆ `Context3DBlendFactor.ZERO`

Blend modes, backface culling, and depth testing are all render states, meaning that you only have to set them once and that behavior continues to be in effect until you set it to something else.

This can be the source of strange rendering errors at times: if you find that after experimenting with blend modes, other models are now unintentionally rendering transparently, you may have forgotten to reset the blend mode back to the default (`ONE,ZERO`). To revert to exactly the same "fully opaque" blend mode that is active when Stage3D first initializes, you would use this code:

```
context3D.setBlendFactors(  
    Context3DBlendFactor.ONE, Context3DBlendFactor.ZERO);
```

Increasing your performance

Although the Stage3D API is blazingly fast and can handle hundreds of thousands of polies at realtime speeds, you need to avoid biting off more than your video card can chew. This is especially important when targeting low-powered systems such as cheap netbooks or mobile devices.

Some common techniques for achieving the best rendering performance (and obtaining the highest framerate as a result) are to follow the age-old KISS rule (keep it simple, stupid!)

Opaque is faster

For example, use opaque .JPG textures as often as possible, since transparent polygons render much more slowly. Why? Because your video card has to first sample the pixel colors underneath the poly you are currently rendering in order to mix it with the new color. If your poly is completely opaque, then this calculation can be avoided.

Avoiding overdraw

Overdraw refers to the situation when the same pixel on screen is repeatedly drawn onto during the composition of a frame. Each layer is a waste of time. Sometimes, of course, this time is well worth it to achieve some great visuals, but whenever possible you want to minimize overdraws and use opaque polygons (and the ONE,ZERO blendmode) as often as possible.

The depth-buffer (or z-buffer) is a powerful ally in avoiding overdraws. When your video card renders a scene, the r,g,b color of each pixel is stored in the video RAM along with another value: the current z depth of that pixel. Then, in subsequent calls to render additional geometry, before a pixel is rendered at the same location, the z-buffer is checked and if the previous polygon is "in front" of the one about to be rendered, your video card can ignore it entirely, saving a lot of time.

As zbuffers allow polygons to be rejected before needing to be rendered, render opaque meshes that are closest to the camera first whenever possible. If you have a complex scene made up of hundreds of models, then if possible, call the `renderTriangles` function for meshes in order from closest to farthest away. In this way, for example, if you draw an opaque wall directly in front of the player, any attempts to draw geometry that are obscured by that wall occur almost instantaneously since all of their pixels can be rejected early. After all, they would not be visible!

Avoiding state changes

Another technique to improve performance is the use of as few textures as possible. For example, take advantage of texture atlases as described earlier. It is not really the number of textures that is a problem; it is the overhead in switching states. Each time you instruct Stage3D to change blend modes, shaders, or textures, it takes some time.

Therefore, whenever possible, group your meshes by what texture and shader they use. If four models in your game all use the same texture, blendmode and AGAL vertex and fragment programs, you need only instruct Flash to use them at the beginning and then you can draw all four meshes without any state changes in between.

Use simple shaders

Additionally, the more lines of AGAL code that your shaders use, the slower they will render. Super-complex shaders lower the framerate of your game. As with anything in programming games, you will need to experiment and find the "sweet spot" in terms of balancing good looks with performance.

Draw fewer meshes

One 10,000 poly mesh renders faster than a thousand 10 poly meshes. Combine meshes so you call drawTriangles() less often. Additionally, only draw meshes that are actually visible. This is called calculating the **PVS (potentially visible set)**. You can Google super-advanced optimization techniques by searching for algorithms such as quad-trees, BSP trees, and portals. Although PVS calculation is beyond the scope of this beginner's guide, it is easy to imagine a dirt-simple kind of PVS algorithm as follows.

Imagine that you have a game world with an outside terrain as well as a cave. If it is impossible to see outside once you are inside the cave, then you might be able to set a simple variable based on the player's location that instructs your game engine to skip drawing the outdoor terrain geometry if the player cannot see it. This is one very simple portal-like PVS optimization that you can easily program yourself. Another dirt-simple PVS solution is to set your maximum draw distance close enough to the camera that extremely distant meshes never get rendered.

The general rule of thumb to get amazingly smooth 60fps is: the simpler the content, the faster the framerate. Use simple shaders, few state changes and textures, limit the number of polygons in your meshes, use opaque textures whenever possible, draw polies that are near the camera first, don't try to render things that are too far away or impossible to see, and you will be sure to enjoy fabulous performance.

Adding texture effects to our demo

Now that we have covered many of the important issues regarding textures, blend modes and the depth-buffer, let's modify our current game demo.

You can view the finished *Chapter 6* demo here:

http://www.mcfunkypants.com/molehill/chapter_6_demo/

The source code we are about to write is available on the book's web page:

<http://www.packtpub.com/adobe-flash11-stage3d-molehill-game-programming-beginners-guide/book>

Starting from the demo project you created in the last chapter, make a few changes as follows.

1. Create a few new meshes for your game (spheres, cubes, scattered polies, and so on)
2. Draw some new textures (a transparent .png, some .jpg particles, and so on)
3. Embed this new content into your SWF and make the game parse them at init
4. Add keyboard events so we can switch models, textures, and blendmodes
5. Update the text GUI to reflect the new state
6. Upgrade your render loop to account for the changes

In the following *Time for Action* sections, we will implement all the preceding upgrades.

Begin by sculpting some new meshes (perhaps objects you want to appear in your new game) and textures to go with them. This book is about programming, so the creation of the art is left to the reader as a fun task. In the following example code, instead of including dozens of similar lines for each of our new textures and meshes, we simply show a single one as an example. If you get lost, then just open up the full source code from the .zip file at the preceding link.

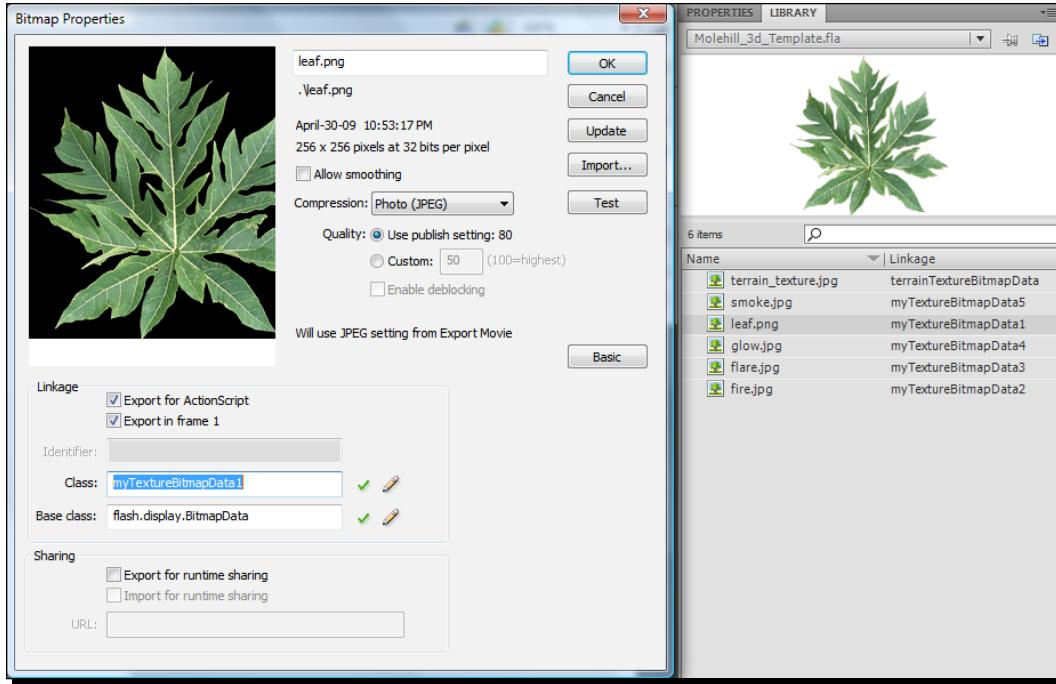
Once you have created some interesting meshes in the .OBJ format, as well as a couple of new texture .PNG and .JPG files, you are ready to start coding. In detail, the preceding steps will require that you add the following code to the Stage3dGame.as source file.

Time for action – embedding the new art

Firstly, to embed all this new content you will edit the very top of the file, adding new embed commands. It should look something like the following if you are using Flex or a pure AS3 editor such as FlashDevelop:

```
[Embed (source = "art/leaf.png")]
private var myTextureBitmap1:Class;
private var myTextureData1:Bitmap = new myTextureBitmap1();
// etc...
```

If you are using the Flash IDE rather than a pure AS3 project, then you will need to add your new art to the library in the same way as we did in previous chapters. You want your .FLA library to end up looking similar to the following screenshot:



Note the class names we are going to use for each image file. In order to insert images into your .FLA, remember that all you need to do is drag the file onto your library area and then right-click on the new item that appears to edit the properties to match the preceding example. In your source code, instead of the preceding Flex-style embed code, you would use the following syntax:

```
private var myTextureData1:Bitmap = // leaf.png
    new Bitmap(new myTextureBitmapData1(256,256));
// etc...
```

Now do the same for your new meshes, so that they are all embedded inside your SWF. We do this as follows:

```
// The mesh data
[Embed (source = "cluster.obj",
        mimeType = "application/octet-stream")]
private var myObjData1:Class;
private var myMesh1:Stage3dObjParser;
// etc...
```

What just happened?

We embedded our new art assets into the project. Remember that instead of just one new texture and mesh, you can embed several others. Let's assume that we are embedding five textures (myTextureData1,2,3,4,5) plus the one for the terrain that we included in the previous chapter, as well as five new meshes (myMesh1,2,3,4,5).

Time for action – adding the variables we need

You will also need some variables to store the Stage3D classes used by this new content, as well as for use in switching from one mode to the next and displaying the GUI:

```
// The Stage3d Textures that use the above
private var myTexture1:Texture;
private var myTexture2:Texture;
private var myTexture3:Texture;
private var myTexture4:Texture;
private var myTexture5:Texture;
private var terrainTexture:Texture;

// Points to whatever the current mesh is
private var myMesh:Stage3dObjParser;

// available blend/texture/mesh
private var blendNum:int = -1;
private var blendNumMax:int = 4;
private var texNum:int = -1;
private var texNumMax:int = 4;
private var meshNum:int = -1;
private var meshNumMax:int = 4;

// used by the GUI
private var label1:TextField;
private var label2:TextField;
private var label3:TextField;
```

What just happened?

We need to remember a few new values for this demo, which we added at the top of our class in the preceding code, such as some new labels, textures and a way to keep track of the currently selected render state.

Time for action – upgrading the GUI

For this demo, we have changed the GUI around and will be changing three labels each time a key is pressed.

```
private function initGUI():void
{
    // a text format descriptor used by all gui labels
    var myFormat:TextFormat = new TextFormat();
    myFormat.color = 0xFFFFFFFF;
    myFormat.size = 13;

    // create an FPSCounter that displays the framerate on screen
    fpsTf = new TextField();
    fpsTf.x = 0;
    fpsTf.y = 0;
    fpsTf.selectable = false;
    fpsTf.autoSize = TextFieldAutoSize.LEFT;
    fpsTf.defaultTextFormat = myFormat;
    fpsTf.text = "Initializing Stage3d...";
    addChild(fpsTf);

    // create a score display
    scoreTf = new TextField();
    scoreTf.x = 560;
    scoreTf.y = 0;
    scoreTf.selectable = false;
    scoreTf.autoSize = TextFieldAutoSize.LEFT;
    scoreTf.defaultTextFormat = myFormat;
    addChild(scoreTf);

    // add some labels to describe each shader
    label1 = new TextField();
    label1.x = 250;
    label1.y = 400;
    label1.selectable = false;
    label1.autoSize = TextFieldAutoSize.LEFT;
    label1.defaultTextFormat = myFormat;
    addChild(label1);

    label2 = new TextField();
    label2.x = 250;
    label2.y = 420;
    label2.selectable = false;
```

```
label2.autoSize = TextFieldAutoSize.LEFT;
label2.defaultTextFormat = myFormat;
addChild(label2);

label3 = new TextField();
label3.x = 250;
label3.y = 440;
label3.selectable = false;
label3.autoSize = TextFieldAutoSize.LEFT;
label3.defaultTextFormat = myFormat;
addChild(label3);

// force these labels to be set
nextMesh();
nextTexture();
nextBlendmode();
}
```

What just happened?

We extended our GUI init code to account for the new labels. We first create a `TextFormat` object and assign it some style data for use on all the new `TextFields` that we are going to add to the GUI. For each new label, we position it on the screen, ensure that the user cannot select the text, maintain its size when updated, and finally add it to the stage.

Time for action – listening for key presses

In order to start using the keyboard to control your game, you need to add an event handler that listens for key presses. At the top of your `init()` function, instruct Flash to start listening for the `KEY_DOWN` event.

```
// get keypresses
stage.addEventListener(KeyboardEvent.KEY_DOWN, keyPressed);
```

Now create the following `keyPressed` function, as well as functions that will switch which mesh, blendmode, and texture is to be used during the render loop:

```
private function keyPressed(event:KeyboardEvent):void
{
    switch(event.charCodeAt)
    {
        case 98: // the b key
            nextBlendmode();
            break;
```

```
        case 109: // the m key
            nextMesh();
            break;
        case 116: // the t key
            nextTexture();
            break;
    }
}
```

The preceding function will be called whenever a key is pressed. If the user presses the *b*, *m*, or *t* keys, then the appropriate function is run which will switch render modes.

```
private function nextBlendmode():void
{
    blendNum++;
    if (blendNum > blendNumMax) blendNum = 0;
    switch(blendNum)
    {
        case 0:
            label1.text = '[B] ONE,ZERO';
            break;
        case 1:
            label1.text = '[B] SOURCE_ALPHA,ONE_MINUS_SOURCE_ALPHA';
            break;
        case 2:
            label1.text = '[B] SOURCE_COLOR,ONE';
            break;
        case 3:
            label1.text = '[B] ONE,ONE';
            break;
        case 4:
            label1.text = '[B] DESTINATION_COLOR,ZERO';
            break;
    }
}
```

The preceding function will increment the currently selected blend mode, optionally looping back to the first mode when we pass the last available one. It then updates our GUI to reflect this change.

```
private function nextTexture():void
{
    texNum++;
    if (texNum > texNumMax) texNum = 0;
    switch(texNum)
```

```
{  
    case 0:  
        label2.text = '[T] Transparent Leaf Texture';  
        break;  
    case 1:  
        label2.text = '[T] Fire Texture';  
        break;  
    case 2:  
        label2.text = '[T] Lens Flare Texture';  
        break;  
    case 3:  
        label2.text = '[T] Glow Texture';  
        break;  
    case 4:  
        label2.text = '[T] Smoke Texture';  
        break;  
}  
}  
}
```

The preceding function will increment the currently selected texture, and similarly it also optionally loops back to the first texture when we pass the last available one and updates our GUI to reflect this change. In the same way, the following function does the same except it will toggle which mesh is being rendered.

Remember to change these functions to reflect whatever variable names you used for your new art. If, for example, you added a dozen new meshes, then you will have to add a few more lines to the following `switch` statement and also change the `meshNumMax` variable that we defined earlier:

```
private function nextMesh():void  
{  
    meshNum++;  
    if (meshNum > meshNumMax) meshNum = 0;  
    switch(meshNum)  
    {  
        case 0:  
            label3.text = '[M] Random Particle Cluster';  
            break;  
        case 1:  
            label3.text = '[M] Round Puff Cluster';  
            break;  
        case 2:  
            label3.text = '[M] Cube Model';  
            break;  
        case 3:    }
```

```

        label3.text = '[M] Sphere Model';
        break;
    case 4:
        label3.text = '[M] Spaceship Model';
        break;
    }
}

```

In the preceding code, we implemented the ability to cycle through a list of available blend modes, textures, and meshes by reacting to key presses.

Time for action – upgrading our render loop

Now you need to upgrade your render function to take into account this new system of switching what is drawn. This function is pretty much the same as before, but for simplicity we will remove the model drawing code and instead move it to a new function named `renderMesh`:

```

private function enterFrame(e:Event) :void
{
    // clear scene before rendering is mandatory
    context3D.clear(0,0,0);
    // move or rotate more each frame
    t += 2.0;
    // scroll and render the terrain once
    renderTerrain();
    // render whatever mesh is selected
    renderMesh();
    // present/flip back buffer
    // now that all meshes have been drawn
    context3D.present();
    // update the FPS display
    fpsTicks++;
    var now:uint = getTimer();
    var delta:uint = now - fpsLast;
    // only update the display once a second
    if (delta >= 1000)
    {
        var fps:Number = fpsTicks / delta * 1000;
        fpsTf.text = fps.toFixed(1) + " fps";
        fpsTicks = 0;
        fpsLast = now;
    }
    // update the rest of the GUI
    updateScore();
}

```

What just happened?

Our `enterFrame` function is now considerably simpler since we have moved much of the mesh rendering code into new functions.

All that remains is to write routines that handle the rendering of these meshes. The `renderTerrain` function remains almost the same (except we reset the blend mode to the defaults), but there is a new `renderMesh` function that checks the variables set by your keyboard events and changes the state (texture, blendmode, mesh) before rendering.

Time for action – upgrading the renderTerrain function

Add the following to the top of your existing `renderTerrain` function:

```
// texture blending: no blending at all - opaque
context3D.setBlendFactors(
    Context3DBlendFactor.ONE,
    Context3DBlendFactor.ZERO);
// draw to depth zbuffer and do not draw polies that are obscured
context3D.setDepthTest(true, Context3DCompareMode.LESS);
```

What just happened?

In the preceding code, we ensured that the render state set in our demo is "reset" to the default (opaque) mode required by our terrain.

Time for action – upgrading our Stage3D inits

As we are only drawing one mesh at a time unlike in the last version of our demo, we should tweak the camera angle so that it is centered on the screen. Additionally, we need to create all five textures. Change your `onContext3DCreate` function to account for these changes as follows:

```
private function onContext3DCreate(event:Event):void
{
    // Remove existing frame handler. Note that a context
    // loss can occur at any time which will force you
    // to recreate all objects we create here.
    // A context loss occurs for instance if you hit
    // CTRL-ALT-DELETE on Windows.
    // It takes a while before a new context is available
    // hence removing the enterFrame handler is important!

    if (hasEventListener(Event.ENTER_FRAME))
```

```

removeEventListener(Event.ENTER_FRAME,enterFrame);

// Obtain the current context
var t:Stage3D = event.target as Stage3D;
context3D = t.context3D;

if (context3D == null)
{
    // Currently no 3d context is available (error!)
    trace('ERROR: no context3D - video driver problem?');
    return;
}

// Disabling error checking will drastically improve performance.
// If set to true, Flash sends helpful error messages regarding
// AGAL compilation errors, uninitialized program constants, etc.
context3D.enableErrorChecking = true;

// Initialize our mesh data
initData();

// The 3d back buffer size is in pixels (2=antialiased)
context3D.configureBackBuffer(swfWidth, swfHeight, 2, true);

// assemble all the shaders we need
initShaders();

```

The preceding code is identical to that used in previous chapters. Next, change the texture init code to properly initialize all the new artwork we embedded this time around.

```

myTexture1 = context3D.createTexture(
    myTextureData1.width, myTextureData1.height,
    Context3DTextureFormat.BGRA, false);
uploadTextureWithMipmaps(
    myTexture1, myTextureData1.bitmapData);

// etc...


myTexture5 = context3D.createTexture(
    myTextureData5.width, myTextureData5.height,
    Context3DTextureFormat.BGRA, false);
uploadTextureWithMipmaps(
    myTexture5, myTextureData5.bitmapData);

```

```
    terrainTexture = context3D.createTexture(  
        terrainTextureData.width, terrainTextureData.height,  
        Context3DTextureFormat.BGRA, false);  
    uploadTextureWithMipmaps(  
        terrainTexture, terrainTextureData.bitmapData);  
  
    // create projection matrix for our 3D scene  
    projectionmatrix.identity();  
    // 45 degrees FOV, 640/480 aspect ratio, 0.1=near, 100=far  
    projectionmatrix.perspectiveFieldOfViewRH(  
        45.0, swfWidth / swfHeight, 0.01, 5000.0);  
  
    // create a matrix that defines the camera location  
    viewmatrix.identity();  
    // move the camera back a little so we can see the mesh  
    viewmatrix.appendTranslation(0, 0, -1.5);  
  
    // tilt the terrain a little so it is coming towards us  
    terrainviewmatrix.identity();  
    terrainviewmatrix.appendRotation(-60,Vector3D.X_AXIS);  
  
    // start the render loop!  
    addEventListener(Event.ENTER_FRAME,enterFrame);  
}
```

What just happened?

We tweaked our inits to account for the new textures and changed the camera angle slightly so all of the meshes in our demo would be visible. You might have to change the numbers used in the line of the `viewmatrix` variable if your meshes are differently sized than those used in the demo.

Time for action – simplifying the initShaders function

There is only one vertex program and one fragment program now, so you can remove programs 2,3, and 4 from before (or just comment them out for now).

```
private function initShaders():void  
{  
    // A simple vertex shader which does a 3D transformation  
    // for simplicity, it is used by all four shaders  
    var vertexShaderAssembler:AGALMiniAssembler =  
        new AGALMiniAssembler();  
    vertexShaderAssembler.assemble
```

```
(  
    Context3DProgramType.VERTEX,  
    // 4x4 matrix multiply to get camera angle  
    "m44 op, va0, vc0\n" +  
    // tell fragment shader about XYZ  
    "mov v0, va0\n" +  
    // tell fragment shader about UV  
    "mov v1, va1\n" +  
    // tell fragment shader about RGBA  
    "mov v2, va2"  
) ;  
  
// textured using UV coordinates  
var fragmentShaderAssembler1:AGALMiniAssembler  
    = new AGALMiniAssembler();  
fragmentShaderAssembler1.assemble  
(  
    Context3DProgramType.FRAGMENT,  
    // grab the texture color from texture 0  
    // and uv coordinates from varying register 1  
    // and store the interpolated value in ft0  
    "tex ft0, v1, fs0 <2d,linear,repeat,miplinear>\n"+  
    // move this value to the output color  
    "mov oc, ft0\n"  
) ;  
  
// combine shaders into a program which we then upload to the GPU  
shaderProgram1 = context3D.createProgram();  
shaderProgram1.upload(  
    vertexShaderAssembler.agalcode,  
    fragmentShaderAssembler1.agalcode);  
}
```

What just happened?

All we did in the preceding code was remove three unused shaders—vestiges of the previous chapter's demo.

Time for action – parsing the new meshes

We also need to ensure that four models are loaded. Tweak the initData function as follows:

```
private function initData():void
{
    // parse the OBJ file and create buffers
    trace("Parsing the meshes...");
    myMesh1 = new Stage3dObjParser(
        myObjData1, context3D, 1, true, true);
    myMesh2 = new Stage3dObjParser(
        myObjData2, context3D, 1, true, true);
    myMesh3 = new Stage3dObjParser(
        myObjData3, context3D, 1, true, true);
    myMesh4 = new Stage3dObjParser(
        myObjData4, context3D, 1, true, true);
    myMesh5 = new Stage3dObjParser(
        myObjData5, context3D, 1, true, true);
    // parse the terrain mesh as well
    trace("Parsing the terrain...");
    terrainMesh = new Stage3dObjParser(
        terrainObjData, context3D, 1, true, true);
}
```

What just happened?

We upgraded the mesh init function to account for our new artwork. As before, ensure that the variable names match whatever meshes you have created. If your models are too big or small, then you can change the scale during parsing in the preceding lines by changing the 1's to bigger or smaller numbers.

Time for action – rendering different meshes as appropriate

Now write the new render mesh function which will display a single mesh, centered on the screen, depending on what the current state is as discussed earlier.

```
private function renderMesh():void
{
    if (blendNum > 1)
        // ignore depth zbuffer
        // always draw polies even those that are behind others
        context3D.setDepthTest(false, Context3DCompareMode.LESS);
    else
```

```
// use the depth zbuffer
context3D.setDepthTest(true, Context3DCompareMode.LESS);

// clear the transformation matrix to 0,0,0
modelmatrix.identity();
context3D.setProgram ( shaderProgram1 );
setTexture();
setBlendmode();
modelmatrix.appendRotation(t*0.7, Vector3D.Y_AXIS);
modelmatrix.appendRotation(t*0.6, Vector3D.X_AXIS);
modelmatrix.appendRotation(t*1.0, Vector3D.Y_AXIS);
// clear the matrix and append new angles
modelViewProjection.identity();
modelViewProjection.append(modelmatrix);
modelViewProjection.append(viewmatrix);
modelViewProjection.append(projectionmatrix);
// pass our matrix data to the shader program
context3D.setProgramConstantsFromMatrix(
    Context3DProgramType.VERTEX,
    0, modelViewProjection, true );
```

In the preceding code, we update the current render state (selecting textures, blend modes and depth testing) as appropriate, and then update the camera angle and model matrix:

```
switch(meshNum)
{
    case 0:
        myMesh = myMesh1;
        break;
    case 1:
        myMesh = myMesh2;
        break;
    case 2:
        myMesh = myMesh3;
        break;
    case 3:
        myMesh = myMesh4;
        break;
    case 4:
        myMesh = myMesh5;
        break;
}
```

As before, you may need to tweak the `switch` statement above to reflect whatever number of new models you embedded in your own demo.

```
// draw a mesh
// position
context3D.setVertexBufferAt(0, myMesh.positionsBuffer,
    0, Context3DVertexBufferFormat.FLOAT_3);
// tex coord
context3D.setVertexBufferAt(1, myMesh.uvBuffer,
    0, Context3DVertexBufferFormat.FLOAT_2);
// vertex rgba
context3D.setVertexBufferAt(2, myMesh.colorsBuffer,
    0, Context3DVertexBufferFormat.FLOAT_4);
// render it
context3D.drawTriangles(myMesh.indexBuffer,
    0, myMesh.indexBufferCount);

}
```

What just happened?

The preceding function calls a couple of handy routines to set the render state. These are run just prior to rendering and simply set whatever render state we need depending on what has been selected during our key press events. The appropriate mesh is selected, Stage3D is instructed which buffers to use and finally the mesh is rendered.

Time for action – switching textures

We need to create a function that selects the appropriate texture when rendering depending on the current selection as chosen by the user.

```
private function setTexture():void
{
    switch(texNum)
    {
        case 0:
            context3D.setTextureAt(0, myTexture1);
            break;
        case 1:
            context3D.setTextureAt(0, myTexture2);
            break;
        case 2:
            context3D.setTextureAt(0, myTexture3);
```

```

        break;
    case 3:
        context3D.setTextureAt(0, myTexture4);
    break;
    case 4:
        context3D.setTextureAt(0, myTexture5);
    break;
    case 5:
        context3D.setTextureAt(0, myCubeTexture);
        // needs different AGAL
    break;
}
}

```

What just happened?

The preceding function is called by the `renderMesh` function. It instructs Stage3D to use a texture that the user has selected.

Time for action – switching blend modes

We also need to create a function that selects the appropriate blend mode when rendering, depending on the current selection.

```

private function setBlendmode():void
{
    // All possible blendmodes:
    // Context3DBlendFactor.DESTINATION_ALPHA
    // Context3DBlendFactor.DESTINATION_COLOR
    // Context3DBlendFactor.ONE
    // Context3DBlendFactor.ONE_MINUS_DESTINATION_ALPHA
    // Context3DBlendFactor.ONE_MINUS_DESTINATION_COLOR
    // Context3DBlendFactor.ONE_MINUS_SOURCE_ALPHA
    // Context3DBlendFactor.ONE_MINUS_SOURCE_COLOR
    // Context3DBlendFactor.SOURCE_ALPHA
    // Context3DBlendFactor.SOURCE_COLOR
    // Context3DBlendFactor.ZERO
    switch(blendNum)
    {
        case 0:
            // the default: nice for opaque textures
            context3D.setBlendFactors(
                Context3DBlendFactor.ONE,
                Context3DBlendFactor.ZERO);

```

```
        break;
    case 1:
        // perfect for transparent textures
        // like foliage/fences/fonts
        context3D.setBlendFactors(
            Context3DBlendFactor.SOURCE_ALPHA,
            Context3DBlendFactor.ONE_MINUS_SOURCE_ALPHA);
    break;
    case 2:
        // perfect to make it lighten the scene only
        // (black is not drawn)
        context3D.setBlendFactors(
            Context3DBlendFactor.SOURCE_COLOR,
            Context3DBlendFactor.ONE);
    break;
    case 3:
        // just lightens the scene - great for particles
        context3D.setBlendFactors(
            Context3DBlendFactor.ONE,
            Context3DBlendFactor.ONE);
    break;
    case 4:
        // perfect for when you want to darken only (smoke, etc)
        context3D.setBlendFactors(
            Context3DBlendFactor.DESTINATION_COLOR,
            Context3DBlendFactor.ZERO);
    break;
}
}
```

What just happened?

The preceding function is also called by the `renderMesh` function. It instructs Stage3D to use the blend mode that the user has selected.

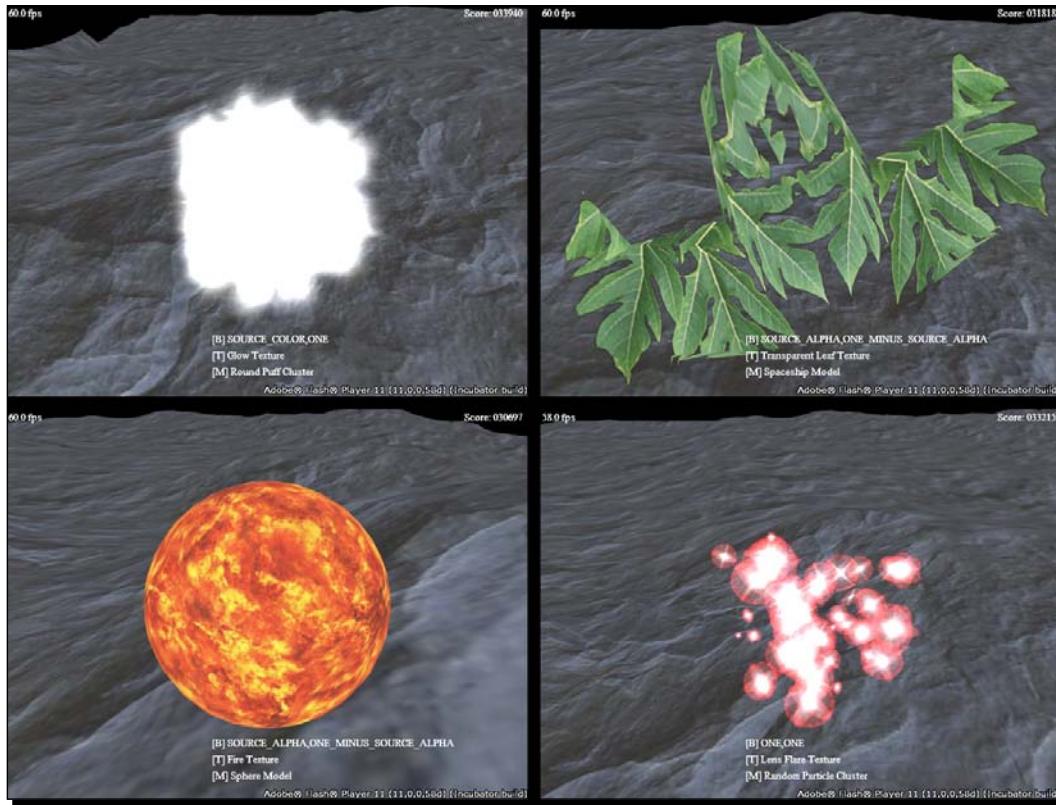
Your demo has been upgraded!

That is it for now. With the preceding modifications, you have upgraded your Stage3D demo to include multiple meshes, textures, and blend modes. Try compiling your .SWF and remember that if you run into any errors, you can check that everything is in the right place by simply downloading the complete source code from the book page:

<http://www.packtpub.com/adobe-flash11-stage3d-molehill-game-programming-beginners-guide/book>

Once you are able to run your game, click on it to make sure it has the focus and try pressing the *T* key to switch between the different textures. Press the *B* key to switch blend modes and the *M* key to change models.

Cycle through all sorts of textures, meshes, blend modes, and notice how some look great and others only make sense for certain textures. Check out the following screenshot:



As you can easily imagine, these new render modes can be used to achieve all sorts of awesome effects. For example, you could render the spaceship with a standard opaque texture, but as an added effect you might want to include an extra model behind it that is set to lighten the screen to look like the flame coming out of a rocket engine. Explosions and smoke could be all over the screen during your game, and lasers that your enemies fire could also be rendered with transparency and additive effects.

All this and more can be achieved through clever use of the `setBlendFactors()` function, along with tweaks to the culling and z-buffer routines that are handled by the `setCulling()` and `setDepthTest()` functions.

Pop quiz – earn some experience points

Here are a few quick questions to test your understanding. If you can answer them, then you are officially ready to move on to the next step in your grand adventure.

1. Which line of code would allow you to use a .PNG texture with transparency without any additive or subtractive blending (like the leaf image we used in the example)?
 - a. context3D.setBlendFactors (ZERO, ONE)
 - b. context3D.setBlendFactors (SOURCE_ALPHA, ONE_MINUS_SOURCE_ALPHA)
 - c. context3D.setBlendFactors (SOURCE_COLOR, SOURCE_COLOR)
 - d. context3D.setBlendFactors (PING, PONG)
2. If you wanted to draw a cluster of polies that each has a simple texture that was a white glowing blob on top of a black background, what blend mode could you use so the black was not rendered and the white added to the scene so it looked like a glowing explosion?
 - a. setBlendFactors (SOURCE_COLOR, SOURCE_COLOR)
 - b. setBlendFactors (SOURCE_COLOR, ZERO)
 - c. setBlendFactors (ONE, ONE)
 - d. setBlendFactors (BLOB, GLOWING)
3. If we wanted to ensure that both sides of each polygon are rendered (perfect for cases where you want the insides or backfaces of a mesh to be visible) what function should we run?
 - a. setCulling (Context3DTriangleFace.NONE)
 - b. setCulling (Context3DTriangleFace.FRONT)
 - c. setDepthTest (false, Context3DCompareMode.LESS)
 - d. setDepthTest (maybe, Context3DCompareMode.GREATER_EQUAL)

Your side quest this time is to enhance the demo to show two models at the same time. Instead of slowly rotating the models every frame, simply have the spaceship mesh point straight ahead without moving. Render it with an opaque texture such as the `spaceship.jpg` from last chapter.

Now take one of the particle blobs from the current demo and render it with a `blendFactor` of `ONE, ONE` and the `glow.jpg` texture, so that it looks like a big round glowing mass of particles. Have this model reside at the very back of your spaceship model, scaled down so it is the right size to look just like the fire coming out of your ship's engines.

Summary

In this chapter, we focused primarily upon three functions that affect how your meshes are rendered: `setBlendFactors`, `setCulling`, and `setDepthTest`. Creative use of these special render states open your game world up to all sorts of special effects.

In implementing this new functionality, we achieved a few milestones. Our game engine is now using multiple textures and several meshes; we have code for rendering textures using new blend modes which allow us to render special effects such as explosions and smoke. We can change whether backfaces are culled on meshes, so that we can draw the insides of transparent objects, we can ignore or use the zbuffer as desired, and we are now detecting key presses for user control of the action.

This is the last chapter where our project will be a "tech demo" and moving forward we will be able to rightly call it "a game".

Level 6 achieved!

Congratulations on yet another level-up. Your skills are, as always, growing stronger with each training session. By now, you are probably eager to really dive in and start programming your game.

It is finally time to blow stuff up! You are ready to control your spaceship with the arrow keys and start wreaking some havoc in the game world. We will cover all this and more in the next chapter.

7

Timers, Inputs, and Entities: Gameplay Goodness!

You are now entering Level 7.

The next step in our grand adventure is to implement the player controls, so that it really feels like you are actually doing something! Beautiful graphics and a fantastic framerate are no longer enough—the time has come to grab the reins and ride this beast by adding timers and a game entity class for easier control of the action.

Interactivity is the hallmark of a game. Gameplay is the result of the feedback from input that results in some exciting output. Action and reaction. Press a button and fire a bullet. Fire a rocket launcher and blast an alien to smithereens. Destroy a boss and get to the next level. It is this feedback loop that turns a demo into a game: the cause-and-effect that combine to result in gameplay.

In this chapter, we will add the ability to move around in our game world. Without the ability to control things, our game thus far is really more of a graphics demo. Additionally, we want to be able to control many kinds of game entities, from chase cameras to spaceships to bullets and beyond.

In this chapter, we will create:

- ◆ A timer class that measures time for use in animations
- ◆ An input class that detects player keyboard and mouse input
- ◆ A game entity class for use by all in-game objects
- ◆ A heads-up-display overlay
- ◆ A more detailed game world

Our current quest

We are going to combine the player input, timer-based animation, and a chase camera in order to implement a new and improved scene that is based on the work we have done in previous chapters. We will enhance our game prototype so that there is true interactivity: you will be about to fly around in a detailed universe where things move around.

Your speed will be framerate-independent, meaning that it will be based on time and won't speed up or slow down if the FPS changes. The game world will have increased in size and complexity with the addition of some asteroids and a sky in the background. The game engine GUI will have been upgraded through the addition of a HUD (heads-up-display) graphic that is overlaid on top of the 3D graphics. Finally, your spaceship will now have an animated engine glow for a little more eye-candy.

The final product of our efforts will look like this:



Before we dive in and program all this interactivity, let us temper our enthusiasm with a little wisdom by adhering to the following two philosophies: keep it simple and make it reusable.

Keeping it simple

As in many an adventure, the path to our goal leads us through temptation. We might be tempted to add so much gameplay, so many features that we are left with an overly complex, bug-ridden game.

We will start with the most basic gameplay feedback loop: simplistic actions that result in gameplay reactions. We will create a basic framework of common game classes that is easy to build upon.

Instead of aiming to perform a million different things or please everybody, we should try to do one or two things well first and then add more polish and a deeper feature-set. With this in mind, let's plan to first implement the simplest of gameplay elements and slowly refine them to "find the fun".

We can add complex features such as AI or collision detection later. Instead of trying to program every possible game-like interaction, the wise warrior chooses the most direct route to the finish line.

Making it reusable

Now that we are deep into the development of our game, the time has come to pull some of the routines we have been piecing together and package them into reusable classes. While we are at it, some very basic functionality used in all games can be abstracted and made ready for use in a variety of ways.

For example, in previous chapters, we quickly hacked together a means to switch shaders and textures using keyboard events. Now is a great time to refine this code, add mouse control, and prepare a simple little game input class.

Additionally, although we have already discussed the render loop and have successfully created simplistic animations by changing the rotations of some meshes in each frame, in a real-world game situation, there exists a need to have more control over what happens in each frame.

In order to do this, we should create a reusable game timer class that keeps track of how much time has elapsed in the game so far, as well as exactly how much time has passed since the previous frame. This is important because the FPS will constantly fluctuate (unless the user is playing on a super-powerful computer and the framerate never dips below the maximum). Each frame in a game will take a different amount of time and occasionally there will be large discrepancies between frame times during the course of a play session. When there is a lot going on, or when there are many meshes on the screen at once, it takes longer to render the scene and update the simulation.

In a perfect world, our game would never dip below 60fps but in the real world, there will be times when the framerate is lower (especially on older computers) and we don't want the game to play differently depending on the computer being used or how many enemies are visible.

Time-based simulation is essential to ensure there are no "hiccups" during play and that all users get the same experience regardless of their computer hardware. This "elapsed time per frame" information will be used frequently in a typical game to control the movements of your player, bullets and enemies so that their speed is constant regardless of scene complexity.

Finally, instead of filling one ActionScript source file with thousands of lines of code related to the creation and animation of various game objects such as spaceships or terrain, it would be wise to create an abstract game entity class. This entity class would be responsible for initializing objects, calculating the transform matrix of each object, rendering itself, and so on.

Instead of all this code residing in bits and pieces scattered throughout the main code-base of our game, it should be collected in one place and made reusable. In this way, one class could handle players, enemies, bullets, terrain, special effects and more.

Making our game more interactive

In order to successfully complete the quest outlined earlier, we need to perform the following:

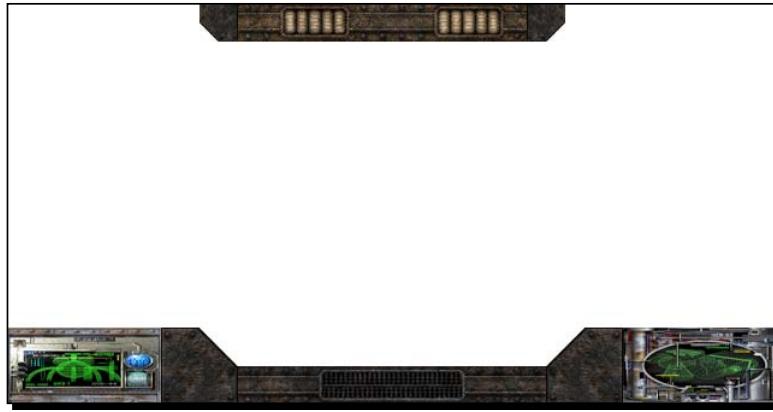
- ◆ Add a HUD overlay graphic
- ◆ Program a game timer class
- ◆ Create a simple game input class
- ◆ Implement an abstract entity class
- ◆ Design art for our new improved game world
- ◆ Upgrade our game engine to handle all this new stuff

The preceding steps briefly outline the tasks required to upgrade our game to support true player interactivity, as well as creating the infrastructure required for future game-goodness like enemies, particle systems, and other special effects.

To begin with, the simplest of the previous tasks is the creation of a fancy looking heads-up-display. This can really add to the overall look of your game. It makes it feel like a game and has a great deal of bang-for-the-buck.

Adding a HUD overlay graphic

We are going to overlay an image on top of our Stage3D view that acts as an attractive viewport for our 3D graphics. Create a .PNG file in your preferred image-editing program, such as Photoshop. Ensure that the majority of this image is transparent. You could add a targeting reticle (aimer), or the glass from a ship's cockpit, or little computer terminals. For our example spaceship shooter game, this is the image we are going to use:



A little bit of techno-metal and a couple of computer screens are all that is required to give a little pizazz to our game. You could use multiple images or just one large image (as above) that is exactly of the same size as your game stage. Import this image into Flash by dragging it into your library palette and ensuring that it is using the correct name and is available in your ActionScript. If you are using pure AS3 or Flex, then just [Embed] it in code.

Time for action – adding a GUI overlay

Create a reference to your new HUD overlay image in the code at the very top of your `Stage3dGame.as`, somewhere below the beginning of the class definition as follows:

```
public class Stage3dGame extends Sprite
{
    // the HUD overlay image:
    // Pure AS3 version:
    [Embed (source = "hud_overlay.png")]
    private var hudOverlayData:Class;
    private var hudOverlay:Bitmap = new hudOverlayData();
    // Adobe CS5 version:
    private var hudOverlay:Bitmap =
        new Bitmap(new hudTextureBitmapData(640,480));
```

Use only the AS3 embed code or the CS5 version depending on your work environment. Now all we need to do is add this new `hudOverlay` bitmap to the stage during your game's inits. Add the following line at the top of your existing `initGUI()` function as follows:

```
private function initGUI():void
{
    // heads-up-display overlay
    addChild(hudOverlay);
```

What just happened?

Now when you compile and run your SWF, you should see the overlay on top of your Stage3D view. Remember that the reason that you can add this to the stage at any time (before or after creating the Stage3D) is because 3D graphics are always rendered first, underneath all normal Flash sprites.

Keeping track of time: a game timer class

The next step in our quest to upgrade our game is to implement a simple class that will eventually be the pulse of our game. It should keep track of time as it passes, so that we know when things should move or explode or be updated.

One very handy technique that can be used to keep your game's framerate blazingly fast is to only perform complex calculations every once in a while rather than at every single frame. Although much of your game logic should be done each frame as part of the render loop, a great optimization technique that most master game coders rely upon is what is sometimes called a "heartbeat" function. This is an optimization technique that most AAA games use to keep the FPS as high as possible.

This heartbeat function is called only from time to time. In our example game, let's call it only once per second. For now, we won't do much during this phase, but this is the proper place for slow or complex routines that only need to be performed occasionally. Examples include calculating the visibility of objects when you have a gigantic game world: instead of looping through each and every object of potentially tens of thousands every single frame, just update the visibility depending on the distance from the camera once a second.

In future revisions to our game, we might choose to use it as an ideal place for **LOD (level-of-detail)**, **AI (artificial intelligence)**, and game events such as countdown timers, as would be used in a time bomb, or to signal the end of a level, triggering a door to close, or turning off a light after a preset interval. Perhaps during a heartbeat, you would trigger an ambient effect such as flashing of lightning and the sound of thunder, or start the next piece of music. Routines like these need not be run every single frame.

For now, our heartbeat function will be an empty shell, ready for future use. Just remember that whenever your framerate starts to suffer (due to too many calculations happening every single frame), you may want to consider moving some of this code to your heartbeat function so it only runs from time to time. If you have code that does not need to be run every single frame, put it into the heartbeat function and run it only once in a while instead.

Examples of things you could do in a heartbeat function are:

- ◆ Checking to see if a download has finished
- ◆ Sending positional information to a multiplayer server
- ◆ Checking for visibility for a rudimentary PVS (potentially visible set) solution
- ◆ Running extremely complex AI routines such as path finding that takes more than one frame to calculate

In order to successfully implement the preceding features, we need to perform the following:

1. Create a new class with variables to record times and counters.
2. Make the constructor take a function pointer for use as the heartbeat.
3. Program a tick function that is run every frame and measures the elapsed time.
4. Run the heartbeat function only when appropriate.

Time for action – creating the game timer class

The preceding steps briefly outline what needs to be programmed for our GameTimer class to be ready for use in our game. To begin with, let's create a brand new class. Create a brand new file in your project folder named `GameTimer.as` and flesh out the timer as follows.

```
// Stage3d game timer routines version 1.0
//
package
{

    import flash.utils.*;

    public class GameTimer
    {
        // when the game started
        public var gameStartTime:Number = 0.0;
        // timestamp: previous frame
        public var lastFrameTime:Number = 0.0;
        // timestamp: right now
        public var currentFrameTime:Number = 0.0;
        // how many ms elapsed last frame
        public var frameMs:Number = 0.0;
        // number of frames this game
        public var frameCount:uint = 0;
        // when to fire this next
        public var nextHeartbeatTime:uint = 0;
    }
}
```

```
// how many ms so far?  
public var gameElapsedTime:uint = 0;  
// how often in ms does the heartbeat occur?  
public var heartbeatIntervalMs:uint = 1000;  
// function to run each heartbeat  
public var heartbeatFunction:Function;
```

What just happened?

In the previous code, all we have done is create a new class named `GameTimer` which stores a few handy public variables that our game can refer to as needed. During our game's render loop, we will update our timer and these values will change.

Time for action – adding the GameTimer class constructor

Now we will program the timer's constructor. Our game will want to pass a reference to the heartbeat function that we want the timer to call and how often we want to run it:

```
// class constructor  
public function GameTimer(  
    heartbeatFunc:Function = null,  
    heartbeatMs:uint = 1000)  
{  
    if (heartbeatFunc != null)  
        heartbeatFunction = heartbeatFunc;  
  
    heartbeatIntervalMs = heartbeatMs;  
}
```

What just happened?

The preceding code simply initializes our brand new `GameTimer` class by stating references to a callback function that our game may need to have run at specific intervals.

Time for action – implementing the tick function

Our game needs to update the timer each frame, so that it gets a chance to measure the elapsed time since the previous frame and possibly to run the heartbeat function if needed. The following `tick` function will do the work for us.

```
public function tick():void  
{  
    currentFrameTime = getTimer();
```

```
if (frameCount == 0) // first frame?
{
    gameStartTime = currentFrameTime;
    trace("First frame happened after "
        + gameStartTime + "ms");
    frameMs = 0;
    gameElapsed = 0;
}
else
{
    // how much time has passed since the last frame?
    frameMs = currentFrameTime - lastFrameTime;
    gameElapsed += frameMs;
}

if (heartbeatFunction != null)
{
    if (currentFrameTime >= nextHeartbeatTime)
    {
        heartbeatFunction();
        nextHeartbeatTime = currentFrameTime
            + heartbeatIntervalMs;
    }
}

lastFrameTime = currentFrameTime;
frameCount++;

}

} // end class

} // end package
```

What just happened?

That is it for our fancy new `GameTimer` class! As you can see, this is a very simple piece of code. That said, it is also very useful and quite powerful: virtually every game ever made makes use of something like this to keep track of time and trigger events based on time passing.

The `getTimer` function returns a value in milliseconds. By remembering the previous value, our timer class can calculate how many ms have elapsed since the start of the game or since the last time the `tick` function was called.

Why measure time at all? Why not simply update locations and animations every frame? The reason is that we can never be sure that our framerate does not fluctuate. A person running your game on a less powerful machine will have a much lower framerate than someone running the game on a powerful gaming rig. By using timers, your game will run at exactly the same speed regardless of the FPS, and by extension, it will run at exactly the same speed regardless of how powerful the machine is.

In your game, instead of changing the location of a moving missile by a hard-coded distance each frame, for example, you would move it the correct distance by multiplying its speed by the time since it was last updated. This is called framerate-independent simulation.

Measuring time is the first step in creating interactivity in a game. Animations need to know how much time has passed and game events often occur only at certain intervals. Some objects in our game might need to know their "age" in order to explode after a certain amount of time or to change state. For now, this is all we need in order to move on to the next step.

A game input class

You have now added an accurate timer to your game. The next step in adding true interactivity is to empower the user to be in direct control of movement. In our example game, this control should be of the spaceship and the user will probably try to do so using the arrow keys.

As many games might instead use the mouse for movement or to look around, as would be handy in a first-person-shooter, we should also track the location of the mouse and detect button clicks and drags. Finally, it is a convention that many games will use the keyboard keys *W, A, S, D* for movement as an alternative to using the arrow keys. This input scheme is very common in shooters and is also handy to ensure that users running your game on devices without arrow keys (such as certain phones) can still control the game.

In order to program the preceding features, we need to perform the following:

1. Create a new class named `GameInput`
2. Add variables used to track the current input state
3. Program a constructor that instructs Flash to fire events when inputs change
4. Handle the input events by updating the current input state

Time for action – creating the GameInput class

With the previous steps in mind, create a brand new class file in your project folder named `GameInput.as` as follows:

```
// Stage3d game input routines version 1.0
//
package
{

    import flash.display.Stage;
    import flash.ui.Keyboard;
    import flash.events.*;

    public class GameInput
    {
        // the current state of the mouse
        public var mouseIsDown:Boolean = false;
        public var mouseClickX:int = 0;
        public var mouseClickY:int = 0;
        public var mouseX:int = 0;
        public var mouseY:int = 0;

        // the current state of the keyboard controls
        public var pressing:Object =
        { up:0, down:0, left:0, right:0, fire:0 };

        // if mouselook is on, this is added to the chase camera
        public var cameraAngleX:Number = 0;
        public var cameraAngleY:Number = 0;
        public var cameraAngleZ:Number = 0;

        // if this is true, dragging the mouse changes the camera angle
        public var mouseLookMode:Boolean = true;

        // the game's main stage
        public var stage:Stage;
    }
}
```

What just happened?

In the preceding code, we have imported the Flash functionality required for mouse and keyboard events. We then define our `GameInput` class and store variables that will be used by our game to query the current state of the inputs. For example, in our game we will eventually be able to check to see if `ourinputclass.pressing.left` is true or not and then react accordingly.

Time for action – coding the GameInput class constructor

Immediately below the variable declarations we just added, add the class constructor that will be run whenever we create a new `GameInput` object in our game. We do this as follows:

```
// class constructor
public function GameInput(theStage:Stage)
{
    stage = theStage;
    // get keypresses and detect the game losing focus
    stage.addEventListener(KeyboardEvent.KEY_DOWN, keyPressed);
    stage.addEventListener(KeyboardEvent.KEY_UP, keyReleased);
    stage.addEventListener(Event.DEACTIVATE, lostFocus);
    stage.addEventListener(Event.ACTIVATE, gainFocus);
    stage.addEventListener(MouseEvent.MOUSE_DOWN, mouseDown);
    stage.addEventListener(MouseEvent.MOUSE_UP, mouseUp);
    stage.addEventListener(MouseEvent.MOUSE_MOVE, mouseMove);
}
```

What just happened?

As input routines require a reference to the game's stage, we need to pass it to this routine. We then instruct Flash to inform us whenever any of the events related to the player input occur. These include key up and down events, mouse events and events that inform the game when input "focus" has been lost (such as when a player clicks a different browser window and is no longer looking at the game). This is important because when the game loses keyboard focus, we may want to pause the game or assume that the movement keys previously being held down are no longer in that state: we might never receive `KEY_UP` events if the keys are released after the game has lost input focus.

Time for action – detecting mouse movement

The next step is to program a means to determine the location of the mouse, whether the button is pressed or not, and to remember the start of a click-and-drag motion, so that we can measure the distance the mouse has traveled before the button is released. We do this as follows:

```
private function mouseMove(e:MouseEvent):void
{
    mouseX = e.stageX;
    mouseY = e.stageY;
    if (mouseIsDown && mouseLookMode)
    {
```

```
        cameraAngleY = 90 * ((mouseX - mouseClickX) /  
            stage.width);  
        cameraAngleX = 90 * ((mouseY - mouseClickY) /  
            stage.height);  
    }  
}
```

The preceding function will be run whenever the player moves the mouse around. In situations where the mouse button is being held down, we want to track the distance moved from when the button was first pressed in order to properly implement "mouse-look" in our game.

```
private function mouseDown(e:MouseEvent):void  
{  
    trace('mouseDown at '+e.stageX+', '+e.stageY);  
    mouseClickX = e.stageX;  
    mouseClickY = e.stageY;  
    mouseIsDown = true;  
}
```

The previous function is triggered when the mouse button is first pressed. The current location of the mouse cursor is recorded for use in the preceding `mouseMove` function, so that we know where the "mouse drag" first started, allowing us to measure the distance it has traveled on the screen.

```
private function mouseUp(e:MouseEvent):void  
{  
    trace('mouseUp at '+e.stageX+', '+e.stageY+' drag distance:' +  
        (e.stageX-mouseClickX) + ', ' + (e.stageY-mouseClickY));  
    mouseIsDown = false;  
    if (mouseLookMode)  
    { // reset camera angle  
        cameraAngleX = cameraAngleY = cameraAngleZ = 0;  
    }  
}
```

What just happened?

In the preceding code, each time a new mouse event is triggered, we store the information for use in our game. For mouse-look functionality, we measure the distance the mouse has moved and calculate the proper camera angle by multiplying 90 degrees times the percentage of the stage size that the mouse has traveled.

At any time in our game's render loop, we can see where the mouse is by checking `ourInputObject.mouseX` and `Y` and we can find out the location of a click by referring to the `mouseClickX` and `Y` class variables. If we wanted to fire bullets constantly whenever the mouse has been pressed, we can just check the `mouseIsDown` property.

Time for action – detecting the keyboard input

Finally, in addition to mouse controls, we want to be able to handle common keyboard inputs for our games. These routines could be extended to track additional keys as required, but for now, simple directional inputs plus spacebar-to-fire are all we need.

```
private function keyPressed(event:KeyboardEvent):void
{
    // qwer 81 87 69 82
    // asdf 65 83 68 70
    // left right 37 39
    // up down 38 40

    //trace("keyPressed " + event.keyCode);

    switch(event.keyCode)
    {
        case Keyboard.UP:
        case 87: // the W key
            pressing.up = true;
            break;

        case Keyboard.DOWN:
        case 83: // the S key
            pressing.down = true;
            break;

        case Keyboard.LEFT:
        case 65: // the A key
            pressing.left = true;
            break;

        case Keyboard.RIGHT:
        case 68: // the D key
            pressing.right = true;
            break;

        case Keyboard.SPACE:
            pressing.fire = true;
            break;
    }
}
```

What just happened?

The previous `keyPressed` function is triggered by Flash whenever the player starts pressing a key. If the spacebar or arrow keys (or *W*, *A*, *S*, *D* as an alternative to using the arrows) are pressed, we update the public class variable named `pressing` to reflect this state so our game can react to it.

Time for action – detecting key release events

We also need to turn off the input flags when the player lets go of a movement key. We do this as follows:

```
private function keyReleased(event:KeyboardEvent):void
{
    switch(event.keyCode)
    {
        case Keyboard.UP:
        case 87:
            pressing.up = false;
            break;

        case Keyboard.DOWN:
        case 83:
            pressing.down = false;
            break;

        case Keyboard.LEFT:
        case 65:
            pressing.left = false;
            break;

        case Keyboard.RIGHT:
        case 68:
            pressing.right = false;
            break;

        case Keyboard.SPACE:
            pressing.fire = false;
            break;
    }
}
```

What just happened?

In the same way that we detect keys being pressed, we need to react when the key is released. This way, the ship will, for example, only move left while the left arrow key (or A) is being held down and movement will stop when it is released.

Time for action – detecting the input focus

Focus events are called when the user clicks the game (`gainFocus`) or leaves the game (either because they clicked another program or started browsing in another tab). For security reasons, Flash only "listens" to the keyboard when it has the focus. Otherwise, Flash windows could record people typing an e-mail in another program.

```
private function gainFocus(event:Event):void
{
    trace("Game received keyboard focus.");
}

// if the game loses focus, don't keep keys held down
private function lostFocus(event:Event):void
{
    trace("Game lost keyboard focus.");
    pressing.up = false;
    pressing.down = false;
    pressing.left = false;
    pressing.right = false;
    pressing.fire = false;
}

} // end class

} // end package
```

What just happened?

In the preceding code, we detect when the game gains and loses the keyboard focus. This can happen when, for example, the user clicks another browser tab or minimizes their browser window. The reason we want to be notified of these events is that when losing focus, the `keyup` events will not fire and the game will continue to think the key is being held down.

This would be an ideal place to add a "pause" function that automatically stops all animation until focus is again restored. For now, let's create a template function to handle gaining and losing focus. We don't yet need to do anything when the game gains focus, but this function is where you would put your "unpause" functionality when you need it.

In the `lostFocus` function, we make sure that all keys are assumed to be "up", so that your player's ship does not continue to fire or move when the game is not being controlled by the user.

That is all that we need in order to control our game. Once we update our game to create an object of this new class, we can then query it to determine whether or not arrow keys are being pressed at any time, and we can also update the mouse look camera angles in the same fashion.

An abstract entity class

In the quest towards the goal of this chapter, this next step is the big daddy of them all—the "boss battle". It is also the most important part of any game engine. Gear up, quaff a tankard of your preferred strength potion (a cup of coffee, for example), and get ready for some mega-hardcore game coding.

An entity can be anything that might exist in your game world. For example, the player's spaceship is an entity, as is the chase camera, the terrain, an explosion, or a bullet. This abstract base game entity class can be used to control any possible kind of "thing" in your game.

In order to keep things simple, we are going to program a very basic entity class that provides only the simplest of functionality. Functionality that is used in virtually all possible game objects such as keeping track of its position, rotation, scale and rendering a mesh in the proper location.

In the future, you can really go to town adding all sorts of interesting properties and functions. The best route would be to create your own custom classes that are derived from this one by creating new custom classes using code similar to:

```
public class purpleDragon extends Stage3dEntity
```

Then, you would add properties and methods specific to the behavior of a purple dragon. For now, we are going to use this base class to control the objects in our game world, since at this point everything we want in our game world will only have a transform (location, rotation, and so on) and a single mesh that is rendered using a particular shader.

In order to create a new class that encapsulates any kind of in-game entity as described earlier, we have to carry out the following general steps:

1. Create a new class named `Stage3dEntity`.
2. Define variables for all parts of the transform, as well as the mesh and texture, and so on.
3. Code a class constructor `init` function that records Stage3D-specific values as needed.
4. Program the `get` and `set` functions for updating parts of the transform.
5. Create functions that update the transform or values when the other has been changed.
6. Design several handy utility functions to perform some common matrix math for us.
7. Code a mechanism for one entity to follow another automatically.
8. Implement a `clone` function to allow for re-use of the same mesh in multiple locations.
9. Program the `render` function that draws the mesh using Stage3D.

Time for action – creating the `Stage3dEntity` class

Begin by creating a fresh new file named `Stage3dEntity.as` in your project folder and define the properties and imports for our new class as follows:

```
// Stage3d game entity class version 1.3
// gratefully adapted from work by Alejandro Santander
package
{
    import com.adobe.utils.*;
    import flash.display.Stage3D;
    import flash.display3D.Context3D;
    import flash.display3D.Context3DProgramType;
    import flash.display3D.Context3DTriangleFace;
    import flash.display3D.Context3DVertexBufferFormat;
    import flash.display3D.IndexBuffer3D;
    import flash.display3D.Program3D;
    import flash.display3D.VertexBuffer3D;
    import flash.display3D.*;
    import flash.display3D.textures.*;
    import flash.geom.Matrix;
    import flash.geom.Matrix3D;
    import flash.geom.Vector3D;
```

The preceding code simply imports the functionality that our entity class is going to require. We will be doing a lot of 3D matrix and vector operations, so we should next define some class variables to store all sorts of location, rotation and scale settings, as well as various Stage3D objects that we will be interacting with as follows:

```
public class Stage3dEntity
{
    // Matrix variables (position, rotation, etc.)
    private var _transform:Matrix3D;
    private var _inverseTransform:Matrix3D;
    private var _transformNeedsUpdate:Boolean;
    private var _valuesNeedUpdate:Boolean;
    private var _x:Number = 0;
    private var _y:Number = 0;
    private var _z:Number = 0;
    private var _rotationDegreesX:Number = 0;
    private var _rotationDegreesY:Number = 0;
    private var _rotationDegreesZ:Number = 0;
    private var _scaleX:Number = 1;
    private var _scaleY:Number = 1;
    private var _scaleZ:Number = 1;
    private const RAD_TO_DEG:Number = 180/Math.PI;

    // Stage3d objects (public so they can be inherited)
    public var context:Context3D;
    public var vertexBuffer:VertexBuffer3D;
    public var indexBuffer:IndexBuffer3D;
    public var shader:Program3D;
    public var texture:Texture;
    public var mesh:Stage3dObjParser;
    // Render modes:
    public var cullingMode:String = Context3DTriangleFace.FRONT;
    public var blendSrc:String = Context3DBlendFactor.ONE;
    public var blendDst:String = Context3DBlendFactor.ZERO;
    public var depthTestMode:String = Context3DCompareMode.LESS;
    public var depthTest:Boolean = true;
    public var depthDraw:Boolean = true;

    // used only for stats
    public var polycount:uint = 0;

    // if this is set entity is "stuck" to another
    public var following:Stage3dEntity;
```

```
// optimize what data we need to send to Stage3D
// this depends on what the shaders require
public var shaderUsesUV:Boolean = true;
public var shaderUsesRgba:Boolean = true;
public var shaderUsesNormals:Boolean = false;
```

What just happened?

In the preceding code, we import the Stage3D functionality we will need and then define a whole slew of handy variables. Only the render modes are public (so that we can change them on the fly at any time in our render loop) while the rest will be, for the most part, controlled by get and set functions and therefore are best left as private variables, inaccessible by code outside this class.

Time for action – creating the Stage3dEntity class constructor

We will start with our class constructor and then implement all these matrix math "helper" routines. Once programmed, they will make your life so much easier that you will be glad you got them out of the way.

```
// Class Constructor
public function Stage3dEntity(
    mydata:Class = null,
    mycontext:Context3D = null,
    myshader:Program3D = null,
    mytexture:Texture = null,
    modelscale:Number = 1,
    flipAxis:Boolean = true,
    flipTexture: Boolean = true)
{
    _transform = new Matrix3D();
    context = mycontext;
    shader = myshader;
    texture = mytexture;
    if (mydata && context)
    {
        mesh = new Stage3dObjParser(
            mydata, context, modelscale, flipAxis, flipTexture);
        polycount = mesh.indexBufferCount;
        trace("Mesh has " + polycount + " polygons.");
    }
}
```

What just happened?

In the short constructor function above, we store references to the game's Context3D and whatever shader and texture our game wants us to use for this entity. We also parse a mesh by using the `Stage3dObjParser` class we programmed in the previous chapter.

Hiding complex code by using get and set functions

Get and set functions allow a class to intercept changes to a property. They are really handy in that once defined they "act" just like a regular public variable. For example, we are going to program a get and set function for "x", which will automatically update our object's Matrix3D transform.

In our game code, we will be able to change the x coordinate of a game entity by simply setting `myEntity.x = 10`, for example. Internally, changing the x value will trigger the "set x" function, so that we can update other private variables as need be.

The reason we are doing it this way is to save ourselves from having to perform all sorts of highly complex and counter-intuitive matrix math whenever we want to do something like rotating a model. Rather than forcing you to append vectors to matrices and the like, your game code will be cleaner and easier to debug by "hiding" all that math and doing the heavy lifting for you under the hood.

Time for action – getting and setting the transform

Remember that an object's "transform" is the word used to describe a Matrix3D, which holds positional, rotational, and scale settings. As we are going to program a way to change parts of the transform without having to bother with counter-intuitive matrix math, we need to make this a function rather than a public variable, so we can run extra code before returning a value.

```
public function get transform():Matrix3D
{
    if (_transformNeedsUpdate)
        updateTransformFromValues();
    return _transform;
}
public function set transform(value:Matrix3D):void
{
    _transform = value;
    _transformNeedsUpdate = false;
    _valuesNeedUpdate = true;
}
```

What just happened?

In the previous code, we have implemented a way to set the transform directly, so we will be able to simply send a fully configured Matrix3D straight to our entity. The `get` function needs to first determine if the private variable holds the most recent data: if we had earlier set just one component of our transform, such as the `x` coordinate, we will have set `_transformNeedsUpdate` to be true so our class knows that the transform needs to be recalculated to reflect the most recent changes.

The same holds true for all `set` functions, where we store the value for just one part of the transform and set a Boolean (true or false) variable (which is called a "flag") to remind our class whether or not the transform or the local variables representing each individual component of that transform need to be regenerated.

What is a "dirty" flag?

A variable that is set to true to indicate when something else has recently "changed" and updates to other things is required. It is said that the thing in question is "dirty" and requires updating to be made current. This is an optimization technique that helps to queue updates until the next time they are needed rather than performing the update excessively often. If you wanted, you could use a different adjective such as `isClean` and in that case, the flag would hold the opposite value but have the same effect. The key thing to remember is that we are recording whether or not an expensive calculation needs to be run as an optimization meant to increase performance by avoiding running the time consuming code more often than needed.



In the current example, this technique of marking the game entity's transform as no longer being current saves on needless calculations by waiting until the last minute to create a new matrix. Imagine setting `mySpaceship.x=1` and `mySpaceship.z=2`, and so on. We don't want the `updateTransformFromValues()` to be run each and every time we change any of the many values, since we will often change several before needing the updated transform when it is time to render. It is more efficient to only run that function once, when we next request the transform.

Time for action – getting and setting the entity position

Let's start with a `get` and `set` function for just the `x,y,z` position part of our entity's transform. This way we can change the location of something without having to worry about the rotation or scale.

```
public function set position(value:Vector3D):void
{
    _x = value.x;
    _y = value.y;
```

```
        _z = value.z;
        _transformNeedsUpdate = true;
    }

private var _posvec:Vector3D = new Vector3D();
public function get position():Vector3D
{
    if (_valuesNeedUpdate)
        updateValuesFromTransform();
    // optimization: avoid creating temporary variable
    // e.g. return new Vector3D(_x, _y, _z);
    _posvec.setTo(_x, _y, _z);
    return _posvec;
}

public function set x(value:Number):void
{
    _x = value;
    _transformNeedsUpdate = true;
}
public function get x():Number
{
    if (_valuesNeedUpdate)
        updateValuesFromTransform();
    return _x;
}

public function set y(value:Number):void
{
    _y = value;
    _transformNeedsUpdate = true;
}
public function get y():Number
{
    if (_valuesNeedUpdate)
        updateValuesFromTransform();
    return _y;
}

public function set z(value:Number):void
{
    _z = value;
    _transformNeedsUpdate = true;
}
```

```
public function get z():Number
{
    if (_valuesNeedUpdate)
        updateValuesFromTransform();
    return _z;
}
```

What just happened?

We just wrote a series of functions that will be run automatically whenever our game sets or gets a class variable. For example, our game will now be able to execute a line of code, such as `spaceship.z=10`; and behind the scenes the set `z` function will be run (updating the private variable `z` and marking the spaceship's transform as "dirty"), so that the next time we access it, the `z` component of the entity's transform matrix will also be 10. Conversely, if we run code similar to `temp=spaceship.z`; the `get z` function will be run, which returns the private `_z` variable, optionally first extracting it from a "dirty" transform if there is one.

Time for action – getting and setting the entity rotation

These get and set functions are all virtually identical. They update or return one particular component of your game object's transform matrix.

```
public function set rotationDegreesX(value:Number):void
{
    _rotationDegreesX = value;
    _transformNeedsUpdate = true;
}
public function get rotationDegreesX():Number
{
    if (_valuesNeedUpdate)
        updateValuesFromTransform();
    return _rotationDegreesX;
}

public function set rotationDegreesY(value:Number):void
{
    _rotationDegreesY = value;
    _transformNeedsUpdate = true;
}
public function get rotationDegreesY():Number
{
    if (_valuesNeedUpdate)
        updateValuesFromTransform();
    return _rotationDegreesY;
}
```

```

}

public function set rotationDegreesZ(value:Number) :void
{
    _rotationDegreesZ = value;
    _transformNeedsUpdate = true;
}
public function get rotationDegreesZ() :Number
{
    if(_valuesNeedUpdate)
        updateValuesFromTransform();
    return _rotationDegreesZ;
}

```

Time for action – getting and setting the entity's scale

Just like with the rotations in the preceding code, the following get and set functions operate on the scale and need no further explanation:

```

public function set scale(vec:Vector3D) :void
{
    _scaleX = vec.x;
    _scaleY = vec.y;
    _scaleZ = vec.z;
    _transformNeedsUpdate = true;
}
private var _scalevec:Vector3D = new Vector3D();
public function get scale():Vector3D
{
    if(_valuesNeedUpdate)
        updateValuesFromTransform();
    //return new Vector3D(_scaleX, _scaleY, _scaleZ, 1.0);

    // optimization: avoid creating a temporary variable
    _scalevec.setTo(_scaleX, _scaleX, _scaleZ);
    _scalevec.w = 1.0;
    return _scalevec;
}
public function set scaleXYZ(value:Number) :void
{
    _scaleX = value;
    _scaleY = value;
    _scaleZ = value;
    _transformNeedsUpdate = true;
}

```

```
        }
    public function get scaleXYZ():Number
    {
        if(_valuesNeedUpdate)
            updateValuesFromTransform();
        return _scaleX; // impossible to determine
        _transformNeedsUpdate = true;
    }
```

The preceding functions refer to the entire scale, which has an x, y, and z component. You might want to work with just one of these; for example, to make something fatter but not taller. We will create all the getters and setters to enable you to do so with ease.

```
public function set scaleX(value:Number):void
{
    _scaleX = value;
    _transformNeedsUpdate = true;
}
public function get scaleX():Number
{
    if(_valuesNeedUpdate)
        updateValuesFromTransform();
    return _scaleX;
}

public function set scaleY(value:Number):void
{
    _scaleY = value;
    _transformNeedsUpdate = true;
}
public function get scaleY():Number
{
    if(_valuesNeedUpdate)
        updateValuesFromTransform();
    return _scaleY;
}

public function set scaleZ(value:Number):void
{
    _scaleZ = value;
    _transformNeedsUpdate = true;
}
public function get scaleZ():Number
{
    if(_valuesNeedUpdate)
        updateValuesFromTransform();
    return _scaleZ;
}
```

What just happened?

As you can see, the previous functions are simple shortcuts to save you from bothering with any matrix math. You can simply scale up your mesh and let this class do the work of calculating a new transform matrix for use when rendering.

In each previous function, dirty flags are set that inform our entity class about whether or not the values or the entire transform will need to be regenerated the next time they are accessed.

This is where the magic happens with regard to handling the get/set functions. Whenever we change just the x coordinate, for example, the transform will need to be regenerated when it is next accessed (for example, prior to rendering).

The inverse also holds true: if we have recently changed just the transform matrix, the next time we ask for just the x coordinate, it will need to be "pulled out" of the transform.

Time for action – updating the transform or values on demand

This update-only-when-required technique saves millions of calculations during the course of a game and will help to keep your framerate high. We will create these two functions now.

```
public function updateTransformFromValues():void
{
    _transform.identity();

    _transform.appendRotation(
        _rotationDegreesX, Vector3D.X_AXIS);
    _transform.appendRotation(
        _rotationDegreesY, Vector3D.Y_AXIS);
    _transform.appendRotation(
        _rotationDegreesZ, Vector3D.Z_AXIS);

    // avoid matrix error #2183:
    // scale values must not be zero
    if (_scaleX == 0) _scaleX = 0.0000001;
    if (_scaleY == 0) _scaleY = 0.0000001;
    if (_scaleZ == 0) _scaleZ = 0.0000001;
    _transform.appendScale(_scaleX, _scaleY, _scaleZ);

    _transform.appendTranslation(_x, _y, _z);

    _transformNeedsUpdate = false;
}
```

```
public function updateValuesFromTransform():void
{
    var d:Vector.<Vector3D> = _transform.decompose();

    var position:Vector3D = d[0];
    _x = position.x;
    _y = position.y;
    _z = position.z;

    var rotation:Vector3D = d[1];
    _rotationDegreesX = rotation.x*RAD_TO_DEG;
    _rotationDegreesY = rotation.y*RAD_TO_DEG;
    _rotationDegreesZ = rotation.z*RAD_TO_DEG;

    var scale:Vector3D = d[2];
    _scaleX = scale.x;
    _scaleY = scale.y;
    _scaleZ = scale.z;

    _valuesNeedUpdate = false;
}
```

What just happened?

The two preceding functions are run whenever a get or set function requires data to be updated first. If your game were to set the transform of your entity, for example, the `_valuesNeedUpdate` flag would be set to true. If, subsequently, you were to get just the x position of that entity, the get function in the preceding code would see that the current internal value for x needs to first be updated by running the `updateValuesFromTransform()` function. If we then immediately afterward access the y component, then values will still be current and the dirty flag will be false, so the get y function above would be able to quickly return the the answer without having to run the update function a second time.

Time for action – creating the movement utility functions

The following functions are handy utilities that are used very often in game programming. Sometimes we want to get a vector that represents which direction an entity is facing. For example, you might orient a missile to be pointing 45 degrees to the left and 5 degrees up. We could then quickly ask it what its "front vector" is and this class would return the direction in which the entity is facing. This vector could be multiplied by a speed value to move the missile "forward". As with much of the routines presented above, these functions will help you to avoid doing any complex Matrix3D math in your game.

```
// Movement Utils:

// move according to the direction we are facing
public function moveForward(amt:Number):void
{
    if (_transformNeedsUpdate)
        updateTransformFromValues();
    var v:Vector3D = frontvector;
    v.scaleBy(-amt)
    transform.appendTranslation(v.x, v.y, v.z);
    _valuesNeedUpdate = true;
}
public function moveBackward(amt:Number):void
{
    if (_transformNeedsUpdate)
        updateTransformFromValues();
    var v:Vector3D = backvector;
    v.scaleBy(-amt)
    transform.appendTranslation(v.x, v.y, v.z);
    _valuesNeedUpdate = true;
}
public function moveUp(amt:Number):void
{
    if (_transformNeedsUpdate)
        updateTransformFromValues();
    var v:Vector3D = upvector;
    v.scaleBy(amt)
    transform.appendTranslation(v.x, v.y, v.z);
    _valuesNeedUpdate = true;
}
public function moveDown(amt:Number):void
{
    if (_transformNeedsUpdate)
        updateTransformFromValues();
    var v:Vector3D = downvector;
    v.scaleBy(amt)
    transform.appendTranslation(v.x, v.y, v.z);
    _valuesNeedUpdate = true;
}
public function moveLeft(amt:Number):void
{
    if (_transformNeedsUpdate)
        updateTransformFromValues();
    var v:Vector3D = leftvector;
```

```
v.scaleBy(amt)
transform.appendTranslation(v.x, v.y, v.z);
_valuesNeedUpdate = true;
}
public function moveRight(amt:Number):void
{
if (_transformNeedsUpdate)
    updateTransformFromValues();
var v:Vector3D = rightvector;
v.scaleBy(amt)
transform.appendTranslation(v.x, v.y, v.z);
_valuesNeedUpdate = true;
}
```

What just happened?

The move functions above are shortcuts to help do things typically done with game entities. Instead of having to perform all sorts of complex math with sine and cosine and dot products and vectors just to shift a missile forward in whichever direction it is currently facing, you can simply run the `moveForward` function.

Time for action – implementing vector utility functions

In order to determine what vector is forward or to the right of a missile, for example, we need to calculate it depending on the direction it is facing.

```
// optimization: these vectors are defined as constants
// to avoid creation of temporary variables each frame
private static const vecft:Vector3D = new Vector3D(0, 0, 1);
private static const vecbk:Vector3D = new Vector3D(0, 0, -1);
private static const veclf:Vector3D = new Vector3D(-1, 0, 0);
private static const vecrt:Vector3D = new Vector3D(1, 0, 0);
private static const vecup:Vector3D = new Vector3D(0, 1, 0);
private static const vecdn:Vector3D = new Vector3D(0, -1, 0);

public function get frontvector():Vector3D
{
    if(_transformNeedsUpdate)
        updateTransformFromValues();
    return transform.deltaTransformVector(vecft);
}

public function get backvector():Vector3D
{
```

```
if(_transformNeedsUpdate)
    updateTransformFromValues();
return transform.deltaTransformVector(vecbk);
}

public function get leftvector():Vector3D
{
    if(_transformNeedsUpdate)
        updateTransformFromValues();
    return transform.deltaTransformVector(veclf);
}

public function get rightvector():Vector3D
{
    if(_transformNeedsUpdate)
        updateTransformFromValues();
    return transform.deltaTransformVector(vecrt);
}

public function get upvector():Vector3D
{
    if(_transformNeedsUpdate)
        updateTransformFromValues();
    return transform.deltaTransformVector(vecup);
}

public function get downvector():Vector3D
{
    if(_transformNeedsUpdate)
        updateTransformFromValues();
    return transform.deltaTransformVector(vecdn);
}
```

The previous functions are a quick way to ask an entity, for example, "what vector points to your left?" without having to calculate it manually. As each entity may be facing a completely different direction, "left" is relative to the angle in which it is facing.

Time for action – adding some handy entity utility functions

Finally, without a thorough explanation, here are a set of handy utility functions that are likely to be of some use later on. Some are useful for debugging while others are simply shortcuts for common game-related tasks. There might be a couple of routines here that you never need to use, but they are included to save you time and effort if you ever need them.

```
// Handy Utils:

public function get rotationTransform():Matrix3D
{
    var d:Vector.<Vector3D> = transform.decompose();
    d[0] = new Vector3D();
    d[1] = new Vector3D(1, 1, 1);
    var t:Matrix3D
    = new Matrix3D();
    t.recompose(d);
    return t;
}

public function get reducedTransform():Matrix3D
{
    var raw:Vector.<Number> = transform.rawData;
    raw[3] = 0; // Remove translation.
    raw[7] = 0;
    raw[11] = 0;
    raw[15] = 1;
    raw[12] = 0;
    raw[13] = 0;
    raw[14] = 0;
    var reducedTransform:Matrix3D = new Matrix3D();
    reducedTransform.copyRawDataFrom(raw);
    return reducedTransform;
}

public function get invRotationTransform():Matrix3D
{
    var t:Matrix3D = rotationTransform;
    t.invert();
    return t;
}

public function get inverseTransform():Matrix3D
{
```

```
    _inverseTransform = transform.clone();
    _inverseTransform.invert();

    return _inverseTransform;
}
```

The preceding functions are rarely used but might come in handy when you want the transform matrix of an entity that has been stripped of the translation (positional data), or if you need inverted rotations for a "reflection" or "lighting" shader for example.

```
public function get positionVector():Vector.<Number>
{
    return Vector.<Number>([_x, _y, _z, 1.0]);
}

public function posString():String
{
    if (_valuesNeedUpdate)
        updateValuesFromTransform();

    return _x.toFixed(2) + ','
        + _y.toFixed(2) + ','
        + _z.toFixed(2);
}

public function rotString():String
{
    if (_valuesNeedUpdate)
        updateValuesFromTransform();

    return _rotationDegreesX.toFixed(2) + ','
        + _rotationDegreesY.toFixed(2) + ','
        + _rotationDegreesZ.toFixed(2);
}

public function follow(thisentity:Stage3dEntity):void
{
    following = thisentity;
}
```

What just happened?

Along with specialized functions the return modified transforms for special needs, the last two functions in the preceding code are great for debugging (or writing a level editor!) because they return a text version of just the position or rotation of an entity. They could be used in debug-only `trace()` function calls to check to see if something is in the location you expected. Finally, we made a simple function that keeps track of a "parent" entity, so that something can exactly follow something else.

It can be really handy to be able to "glue" entities together in this way. For example, you might want to attach a gun to a spaceship model, or a hat to an avatar, or a special effect like a "glow" to a model of a lamppost. Doing so is very easy, and this function allows our game to inform our entity that it should follow another entity. When it is rendered, it appends the transform of the entity it is following to its own. In this way, we can set a local x, y, or z coordinate that is relative to the entity we are following. We will be able to nudge the gun or glow to be in the correct location relative to its "parent", and when that parent moves or rotates, the follower will too. For now, the `follow()` function just sets a variable, but for certain games you may want to add more lines of code here to play a sound, or update the score when something gets attached to the player, for example.

That is it for all the handy-dandy matrix and vector utility functionality of the obscure variety. The last utility function we are going to write is one that you are sure to use often in your games.

It is a `clone` function. Clones are ubiquitous in games today. For example, in a next-gen game that has a vast lush forest to explore, there are usually only a handful of unique tree models each cloned numerous times to flesh out the scene. After all, you would not want to have to sculpt an infinite variety of trees or doors or rocks or bullets when you could instead reuse these models in multiple locations.

[ Using model clones is essential to keep your file sizes small and your level creation labor manageable. It also makes sense to be able to place identical assets in multiple locations in your game world without having to parse the entire .OBJ mesh or create a unique Stage3D vertex buffer for each and every one. This speeds up init times, lowers RAM consumption, and allows for gigantic scenes comprised of hundreds of thousands of polygons.]

The even greater news is that you can clone a mesh, but place it at a slightly different orientation or give it a different scale to make it look different from others in your scene.

Time for action – cloning an entity

When you want to clone a mesh, you can use this function instead of creating a new Stage3dEntity each time.

```
// create an exact duplicate in the game world
// while re-using all Stage3d objects
public function clone():Stage3dEntity
{
    if(_transformNeedsUpdate)
        updateTransformFromValues();
    var myclone:Stage3dEntity = new Stage3dEntity();
    myclone.transform = this.transform.clone();
    myclone.mesh = this.mesh;
    myclone.texture = this.texture;
    myclone.shader = this.shader;
    myclone.vertexBuffer = this.vertexBuffer;
    myclone.indexBuffer = this.indexBuffer;
    myclone.context = this.context;
    myclone.polycount = this.polycount;
    myclone.shaderUsesNormals = this.shaderUsesNormals;
    myclone.shaderUsesRgba = this.shaderUsesRgba;
    myclone.shaderUsesUV = this.shaderUsesUV;
    myclone.updateValuesFromTransform();
    return myclone;
}
```

What just happened?

The `clone()` function in the preceding code returns a new Stage3dEntity that is a copy of another that was initialized previously. In this way, only one copy of the Stage3D vertex buffers, texture, or shader is needed to render multiple copies of the same kind of object.

The very last step in the creation of our abstract entity class is the `render` function. This is where we instruct Stage3D to draw your model on the screen at the correct location and with the proper texture and shader.

Time for action – rendering an entity

This function will be called for every single frame in your game's render loop, once the camera has been moved. Therefore, we need to generate the proper matrix to send to Flash, that encapsulates the entity's transform, the camera's view matrix, plus the projection matrix (which defines the aspect ratio, field-of-view, and near and far planes for our camera).

```
// optimization: reuse the same temporary matrix
private var _rendermatrix:Matrix3D = new Matrix3D();
```

As an optimization, this matrix is a private temporary variable that is created once, outside this function, to avoid having to allocate any RAM during the rendering. As it will be called millions of times during the game, this may help to avoid framerate hiccups when the GC (garbage collection) routines in Flash decide to clean up memory by eliminating unused variables.

```
// renders the entity, changing states only if required
public function render(
    view:Matrix3D,
    projection:Matrix3D,
    statechanged:Boolean = true):void
{
    // used only for debugging:
    if (!mesh) trace("Missing mesh!");
    if (!context) trace("Missing context!");
    if (!shader) trace("Missing shader!");

    // only render if these are set
    if (!mesh) return;
    if (!context) return;
    if (!shader) return;

    //Reset our matrix
    _rendermatrix.identity();
    _rendermatrix.append(transform);
    if (following) _rendermatrix.append(following.transform);
```

The previous code checks to make sure the mesh, Context3D, and shader are set and exits the function if any of this data is missing. This can happen, for example, when the entity class is used to encapsulate some invisible helper entity that does not need to be drawn. Next, the transform matrix of the current entity is created. If we happen to be following another entity, then we also append that entity's transform to our own.

```
_rendermatrix.append(view);
_rendermatrix.append(projection);

// Set the vertex program register vc0 to our
// model view projection matrix = vc0
```

```

context.setProgramConstantsFromMatrix(
    Context3DProgramType.VERTEX, 0, _rendermatrix, true);

// optimization: only change render state
// if the previously rendered actor is not
// using an identical mesh/shader as the current one
if (statechanged)
{

```

By tracking whether or not the state has been changed, our game engine will be able to draw groups of the same mesh faster because the mesh, shader, blend modes, and texture remain the same. If we don't need to instruct Stage3D to update the render state, then we eliminate countless useless state changes and increase performance significantly. The perfect example here would be rendering a group of 100 identical trees. The transform matrix for each one is still updated above, but each tree shares the same geometry and texture so we can batch render them all by only rendering the first tree with the `statechanged` parameter set to true.

```

// Set the AGAL program
context.setProgram(shader);

// Set the fragment program register ts0 to a texture
if (texture) context.setTextureAt(0,texture);

// position
context.setVertexBufferAt(0, mesh.positionsBuffer,
    0, Context3DVertexBufferFormat.FLOAT_3);
// tex coord
if (shaderUsesUV)
    context.setVertexBufferAt(1, mesh.uvBuffer,
        0, Context3DVertexBufferFormat.FLOAT_2);
// vertex rgba
if (shaderUsesRgba)
    context.setVertexBufferAt(2, mesh.colorsBuffer,
        0, Context3DVertexBufferFormat.FLOAT_4);
// vertex normal
if (shaderUsesNormals)
    context.setVertexBufferAt(3, mesh.normalsBuffer,
        0, Context3DVertexBufferFormat.FLOAT_3);

context.setBlendFactors(blendSrc, blendDst);
context.setDepthTest(depthTest, depthTestMethod);
context.setCulling(cullingMode);
context.setColorMask(true, true, true, depthDraw);
}

```

Finally, once we have instructed Flash which shader, texture, render state (transparency and blend mode) and vertex and index buffer we are using (if necessary), we are ready to render the triangles that make up our mesh.

```
// render it
context.drawTriangles(mesh.indexBuffer,
    0, mesh.indexBufferCount);

}

} // end class

} // end package
```

What just happened?

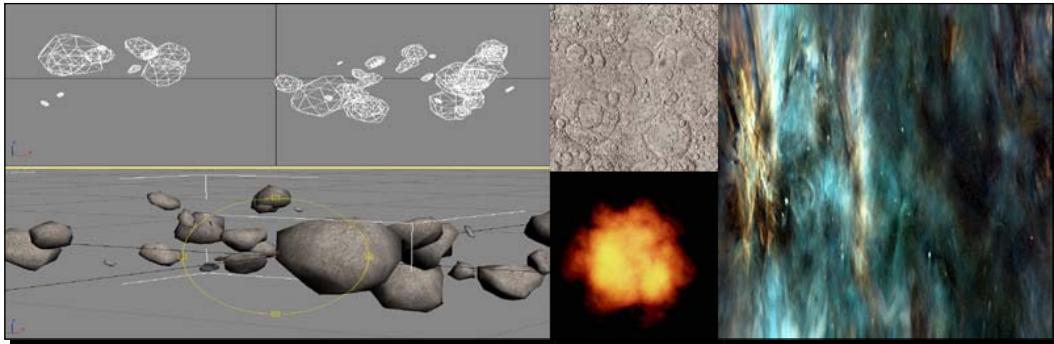
The previous function draws an entity. It will typically be called every frame. Several optimizations have been implemented which are sure to increase your framerate.

That is it for the hardest part in the creation of our interactive, animated game world! Our fancy and eminently reusable abstract game entity class will become essential in our ongoing quest to create an amazing 3D flash game. You will use it for bullets, spaceships, cars, trees, explosions, and virtually anything else that might make an appearance in your game.

The very last step is to incorporate all this new functionality into your main game script! Before we do so, let's take a breather and have some fun creating some more art to put in our game.

Design art for our new improved game world

As this book is all about programming, we are not going to go into the process of how to sculpt meshes in programs such as 3D Studio Max or Blender, nor will we go into the techniques to make cool-looking textures in Photoshop. That said, just for fun, here are the assets that we are going to add to our game world. Go crazy and have a little fun making your own.



Firstly, as the example game for this book is a shoot-em-up in space, what better location to fly around than in an asteroid field? Using 3ds Max, a few simple shapes were sculpted and exported in the .OBJ format by creating spheres and squashing them to look like asteroids.

Now we need to make them look like asteroids by creating a texture for them. An appropriately crater-covered asteroid texture was created in Photoshop. In this example, real public domain NASA photographs of the moon were used to create a seamless image that can be tiled on a 3D mesh. It was made seamless by liberal use of the stamp tool to make the edges wrap nicely.

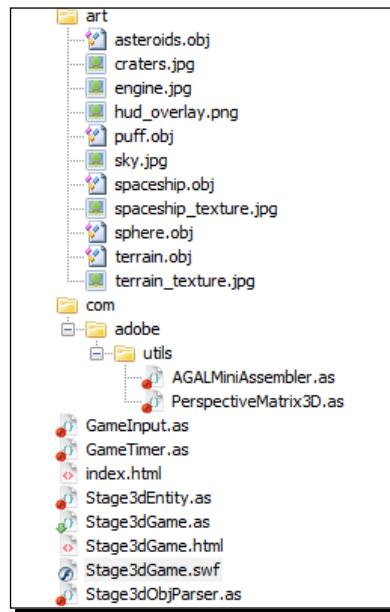
A particle texture was made that looks really nice when rendered with a blend mode that additively lightens the screen (ONE, ONE). When used with the "puff.obj" model from the previous chapter, a glowing fireball perfectly suited for use as the rocket engine of our ship was ready to be attached to the rear of our player's ship in the game.

Finally, until now our game demo had an ugly plain black background. In each frame, when the Stage3D is cleared, it is filled with pure black and this void is visible behind all of our models. Let's upgrade our game universe to include a "sky" sphere. By wrapping an attractive image of a nebula with stars in it onto the "sphere.obj" model from the previous chapter and then scaling this model to be gigantic, we can put the rest of our game world inside it.

Upgrading our game

So far, we have programmed a timer class, a player input class, and a game entity class. We have created some nice looking new art for our game. The last step is to incorporate all this new stuff into our game by editing the `Stage3dGame.as` file.

Before we do so, to avoid any confusion about file names or locations, this is what your project folder should look like now:



Make sure that your project folder matches the preceding image, so that the changes listed below will work. Naturally, if you are following the example project in this book but are making a different game, you will need to make minor changes to the following code. In any case, the primary change between this chapter and the last is the removal of unused text fields, keyboard controls, and the related code for the "blend modes" demo. Instead of parsing models and rendering the scene directly in this file, all input, timer, and model related code is now done in the new files we just made.

Time for action – importing our new classes

`Stage3dGame.as` starts off virtually the same as before, except we import our new classes:

```
//  
// Stage3D Game Template - Chapter Seven  
//  
package  
{  
    [SWF(width="640", height="480", frameRate="60",  
        backgroundColor="#000000")]  
  
    import com.adobe.utils.*;
```

```
import flash.display.*;
import flash.display3D.*;
import flash.display3D.textures.*;
import flash.events.*;
import flash.geom.*;
import flash.utils.*;
import flash.text.*;

import Stage3dEntity;
import GameTimer;
import GameInput;
```

What just happened?

The preceding changes to our example game are minor and are included here only for completeness. The three new classes we created in this chapter are now part of your main codebase. If you use your own package names and have a different folder structure than that in the previous example image, than add package name prefixes to your class names as appropriate.

Time for action – adding new variables to our game

Next, we add a few new variables and remove others that are no longer used. As our game will eventually be filled with hundreds of objects from explosions to enemies and beyond, it makes sense to store them all in a list that we can loop through and manipulate. Therefore, let's create a Vector array of Stage3dEntities for a few common types of game object, so that eventually we can simulate and render different kinds of objects, like hundreds of bullets, without having to use hundreds of individual variables.

```
public class Stage3dGame extends Sprite
{
    // handles all timers for us
    private var gametimer:GameTimer;

    // handles keyboard and mouse inputs
    private var gameinput:GameInput;

    // all known entities in the world
    private var chaseCamera:Stage3dEntity;
    private var player:Stage3dEntity;
    private var props:Vector.<Stage3dEntity>;
    private var enemies:Vector.<Stage3dEntity>;
    private var bullets:Vector.<Stage3dEntity>;
```

```
private var particles:Vector.<Stage3dEntity>;  
  
// reusable entity pointer (for speed and to avoid GC)  
private var entity:Stage3dEntity;  
// we want to remember these for use in gameStep()  
private var asteroids1:Stage3dEntity;  
private var asteroids2:Stage3dEntity;  
private var asteroids3:Stage3dEntity;  
private var asteroids4:Stage3dEntity;  
private var engineGlow:Stage3dEntity;  
private var sky:Stage3dEntity;  
  
// used by gameStep()  
private const moveSpeed:Number = 1.0; // units per ms  
private const asteroidRotationSpeed:Number = 0.001; // deg per ms  
  
// used by the GUI  
private var fpsLast:uint = getTimer();  
private var fpsTicks:uint = 0;  
private var fpsTf:TextField;  
private var scoreTf:TextField;  
private var score:uint = 0;  
  
// the 3d graphics window on the stage  
private var context3D:Context3D;  
// the compiled shader used to render our meshes  
private var shaderProgram1:Program3D;  
  
// matrices that affect the mesh location and camera angles  
private var projectionmatrix:PerspectiveMatrix3D =  
    new PerspectiveMatrix3D();  
private var viewmatrix:Matrix3D = new Matrix3D();
```

What just happened?

As you can see, much of the previous code is the same as what we created in the last chapter, except we have created new variables for our new entities. Instead of using the `Stage3dObjParser` class for our in-game models, we will let our new `Stage3dEntity` do the work.

Time for action – embedding the new art

Next, we simply create references to our textures and meshes. As before, the method of doing so is different depending on whether you are using the Flash IDE and the library or a simple AS3/Flex project and [embed].

```

/* TEXTURES: Pure AS3 and Flex version:
 * if you are using Adobe Flash CS5
 * comment out the following: */
[Embed (source = "art/spaceship_texture.jpg")]
private var playerTextureBitmap:Class;
private var playerTextureData:Bitmap = new playerTextureBitmap();
[Embed (source = "art/terrain_texture.jpg")]
private var terrainTextureBitmap:Class;
private var terrainTextureData:Bitmap = new terrainTextureBitmap();
[Embed (source = "art/craters.jpg")]
private var cratersTextureBitmap:Class;
private var cratersTextureData:Bitmap = new cratersTextureBitmap();
[Embed (source = "art/sky.jpg")]
private var skyTextureBitmap:Class;
private var skyTextureData:Bitmap = new skyTextureBitmap();
[Embed (source = "art/engine.jpg")]
private var puffTextureBitmap:Class;
private var puffTextureData:Bitmap = new puffTextureBitmap();
[Embed (source = "art/hud_overlay.png")]
private var hudOverlayData:Class;
private var hudOverlay:Bitmap = new hudOverlayData();

/* TEXTURES: Flash CS5 version:
 * add the jpgs to your library (F11)
 * right click and edit the advanced properties
 * so it is exported for use in Actionscript
 * and using the proper names as listed below
 * if you are using Flex/FlashBuilder/FlashDevelop/FDT
 * comment out the following: */
/*
private var playerTextureData:Bitmap = // spaceship_texture.jpg
    new Bitmap(new playerTextureBitmapData(512,512));
private var terrainTextureData:Bitmap = // terrain_texture.jpg
    new Bitmap(new terrainTextureBitmapData(512,512));
private var cratersTextureData:Bitmap = // craters.jpg
    new Bitmap(new cratersTextureBitmapData(512,512));
private var skyTextureData:Bitmap = // sky.jpg
    new Bitmap(new skyTextureBitmapData(512,512));

```

```
private var puffTextureData:Bitmap = // engine.jpg
    new Bitmap(new engineTextureBitmapData(128,128));
private var hudOverlay:Bitmap = // hud_overlay.png
    new Bitmap(new hudTextureBitmapData(640,480));
/*
// The Stage3d Textures that use the above
private var playerTexture:Texture;
private var terrainTexture:Texture;
private var cratersTexture:Texture;
private var skyTexture:Texture;
private var puffTexture:Texture;
```

Now we need to embed the new 3D meshes that we created for this chapter, as well as a few from last time:

```
// the player
[Embed (source = "art/spaceship.obj",
    mimeType = "application/octet-stream")]
private var myObjData5:Class;

// the engine glow
[Embed (source = "art/puff.obj",
    mimeType = "application/octet-stream")]
private var puffObjData:Class;

// The terrain mesh data
[Embed (source = "art/terrain.obj",
    mimeType = "application/octet-stream")]
private var terrainObjData:Class;

// an asteroid field
[Embed (source = "art/asteroids.obj",
    mimeType = "application/octet-stream")]
private var asteroidsObjData:Class;

// the sky
[Embed (source = "art/sphere.obj",
    mimeType = "application/octet-stream")]
private var skyObjData:Class;
```

That is it for all the art and variables needed by our game.

Time for action – upgrading the game inits

Now we want to change the constructor function for our game to initialize everything. This is virtually the same as last time except we are creating a few empty Vector arrays for our new entity lists:

```
public function Stage3dGame()
{
    if (stage != null)
        init();
    else
        addEventListener(Event.ADDED_TO_STAGE, init);
}

private function init(e:Event = null):void
{
    if (hasEventListener(Event.ADDED_TO_STAGE))
        removeEventListener(Event.ADDED_TO_STAGE, init);

    // start the game timer and inputs
    gametimer = new GameTimer(heartbeat);
    gameinput = new GameInput(stage);

    // create some empty arrays
    props = new Vector.<Stage3dEntity>();
    enemies = new Vector.<Stage3dEntity>();
    bullets = new Vector.<Stage3dEntity>();
    particles = new Vector.<Stage3dEntity>();

    // set up the stage
    stage.scaleMode = StageScaleMode.NO_SCALE;
    stage.align = StageAlign.TOP_LEFT;

    // add some text labels
    initGUI();

    // and request a context3D from Stage3d
    stage.stage3Ds[0].addEventListener(
        Event.CONTEXT3D_CREATE, onContext3DCreate);
    stage.stage3Ds[0].requestContext3D();
}

}
```

What just happened?

In our `init()` function, we create new timer and input objects. Our game can also keep track of time and respond to the keyboard and mouse input! Finally, we create a few lists of different types of entity to make it easy to loop through multiple game objects.

Two functions from the previous chapter are not listed here. We can leave the `updateScore()` function alone—it remains unchanged. The `uploadTextureWithMipmaps()` function has not changed at all. You can refer to them in the source code download for this chapter or by flipping back to the previous chapter.

Time for action – upgrading the GUI

Apart from removing the unused GUI labels, the `initGUI()` function is the same, except that the font has been increased in size and the color has changed.

```
private function initGUI():void
{
    // heads-up-display overlay
    addChild(hudOverlay);

    // a text format descriptor used by all gui labels
    var myFormat:TextFormat = new TextFormat();
    myFormat.color = 0xFFFFAA;
    myFormat.size = 16;

    // create an FPSCounter that displays the framerate on screen
    fpsTf = new TextField();
    fpsTf.x = 4;
    fpsTf.y = 0;
    fpsTf.selectable = false;
    fpsTf.autoSize = TextFieldAutoSize.LEFT;
    fpsTf.defaultTextFormat = myFormat;
    fpsTf.text = "Initializing Stage3d...";
    addChild(fpsTf);

    // create a score display
    scoreTf = new TextField();
    scoreTf.x = 540;
    scoreTf.y = 0;
    scoreTf.selectable = false;
    scoreTf.autoSize = TextFieldAutoSize.LEFT;
    scoreTf.defaultTextFormat = myFormat;
    addChild(scoreTf);
}
```

Time for action – simplifying the shaders

The same holds true for the `initShaders()` function which is virtually unchanged except in that the unused code has been deleted:

```

private function initShaders():void
{
    // A simple vertex shader which does a 3D transformation
    // for simplicity, it is used by all four shaders
    var vertexShaderAssembler:AGALMiniAssembler =
        new AGALMiniAssembler();
    vertexShaderAssembler.assemble
    (
        Context3DProgramType.VERTEX,
        // 4x4 matrix multiply to get camera angle
        "m44 op, va0, vc0\n" +
        // tell fragment shader about XYZ
        "mov v0, va0\n" +
        // tell fragment shader about UV
        "mov v1, va1\n" +
        // tell fragment shader about RGBA
        "mov v2, va2"
    );

    // textured using UV coordinates
    var fragmentShaderAssembler1:AGALMiniAssembler
        = new AGALMiniAssembler();
    fragmentShaderAssembler1.assemble
    (
        Context3DProgramType.FRAGMENT,
        // grab the texture color from texture 0
        // and uv coordinates from varying register 1
        // and store the interpolated value in ft0
        "tex ft0, v1, fs0 <2d,linear,repeat,miplinear>\n"+
        // move this value to the output color
        "mov oc, ft0\n"
    );

    // combine shaders into a program which we then upload to the GPU
    shaderProgram1 = context3D.createProgram();
    shaderProgram1.upload(
        vertexShaderAssembler.agalcode,
        fragmentShaderAssembler1.agalcode);
}

```

Time for action – using the new textures

Next, the `onContext3DCreate` function needs some minor edits. In particular, we need to load our new textures and change the projection matrix to have a much larger "far plane", so that we can fit our entire larger game world in a view.

```
private function onContext3DCreate(event:Event):void
{
    // Remove existing frame handler. Note that a context
    // loss can occur at any time which will force you
    // to recreate all objects we create here.
    // A context loss occurs for instance if you hit
    // CTRL-ALT-DELETE on Windows.
    // It takes a while before a new context is available
    // hence removing the enterFrame handler is important!

    if (hasEventListener(Event.ENTER_FRAME))
        removeEventListener(Event.ENTER_FRAME,enterFrame);

    // Obtain the current context
    var t:Stage3D = event.target as Stage3D;
    context3D = t.context3D;

    if (context3D == null)
    {
        // Currently no 3d context is available (error!)
        trace('ERROR: no context3D - video driver problem?');
        return;
    }

    // Disabling error checking will drastically improve performance.
    // If set to true, Flash sends helpful error messages regarding
    // AGAL compilation errors, uninitialized program constants, etc.
    context3D.enableErrorChecking = true;

    // The 3d back buffer size is in pixels (2=antialiased)
    context3D.configureBackBuffer(stage.width, stage.height, 2, true);

    // assemble all the shaders we need
    initShaders();

    playerTexture = context3D.createTexture(
        playerTextureData.width, playerTextureData.height,
        Context3DTextureFormat.BGRA, false);
```

```
uploadTextureWithMipmaps(
    playerTexture, playerTextureData.bitmapData);

terrainTexture = context3D.createTexture(
    terrainTextureData.width, terrainTextureData.height,
    Context3DTextureFormat.BGRA, false);
uploadTextureWithMipmaps(
    terrainTexture, terrainTextureData.bitmapData);

cratersTexture = context3D.createTexture(
    cratersTextureData.width, cratersTextureData.height,
    Context3DTextureFormat.BGRA, false);
uploadTextureWithMipmaps(
    cratersTexture, cratersTextureData.bitmapData);

puffTexture = context3D.createTexture(
    puffTextureData.width, puffTextureData.height,
    Context3DTextureFormat.BGRA, false);
uploadTextureWithMipmaps(
    puffTexture, puffTextureData.bitmapData);

skyTexture = context3D.createTexture(
    skyTextureData.width, skyTextureData.height,
    Context3DTextureFormat.BGRA, false);
uploadTextureWithMipmaps(
    skyTexture, skyTextureData.bitmapData);

// Initialize our mesh data - requires shaders and textures first
initData();

// create projection matrix for our 3D scene
projectionmatrix.identity();
// 45 degrees FOV, 640/480 aspect ratio, 0.1=near, 150000=far
projectionmatrix.perspectiveFieldOfViewRH(
    45, stage.width / stage.height, 0.01, 150000.0);

// start the render loop!
addEventListener(Event.ENTER_FRAME,enterFrame);
}
```

What just happened?

We upgraded our `onContext3DCreate` event to take into account our new game art. We also changed the camera angles (by changing the values stored in the projection matrix), so that the entire scene wouldl be visible.

The second last change to our game is to modify the `initData()` function to create some new `Stage3dEntities`. Instead of parsing the `.OBJ` files here, our new entity class will do all the heavy lifting. Once we instantiate these new objects, we put some of them inside our entity lists, so that we can loop through them during rendering.

One potential "gotcha!" moment here is that your exported `.OBJ` files may be in a slightly different format from the ones included in the example project. Some exporters output vertex data in the wrong order and the `Stage3dObjParser.as` class constructor function has two Boolean values that instruct it to handle data in this order.

Therefore, if your models all look completely messed up when you compile the source, edit your `Stage3dEntity.as` file and change line 68 in the public function `Stage3dEntity()` from:

```
mesh = new Stage3dObjParser(mydata, context, 1, true, true);
```

to something like:

```
mesh = new Stage3dObjParser(mydata, context, 1, false, false);
```

You may need to experiment with that one line depending on what exporter you are using. Apart from this potential file format annoyance, getting your models working with this new class should be a breeze.



Time for action – spawning some game entities

Let's update our `initData()` function to include the new art and use our fancy new `Stage3dEntity` class:

```
private function initData():void
{
    // create the camera entity
    trace("Creating the camera entity...");
    chaseCamera = new Stage3dEntity();
```

Notice that in the beginning of this function, we create a "chase camera" entity without any model data. This is perfectly acceptable—the chase camera is a game entity but does not need to be rendered. As our entity class has all sorts of handy matrix math routines programmed for us, using it even for invisible things like a camera is ideal.

```
// create the player model
trace("Creating the player entity...");
```

```
player = new Stage3dEntity(
    myObjData5,
    context3D,
    shaderProgram1,
    playerTexture);
// rotate to face forward
player.rotationDegreesX = -90;
player.z = 2100;

trace("Parsing the terrain...");
// add some terrain to the props list
var terrain:Stage3dEntity =
    new Stage3dEntity(
        terrainObjData,
        context3D,
        shaderProgram1,
        terrainTexture);
terrain.rotationDegreesZ = 90;
terrain.y = -50;
props.push(terrain);

trace("Cloning the terrain...");
// use the same mesh in another location
var terrain2:Stage3dEntity = terrain.clone();
terrain2.z = -4000;
props.push(terrain2);
```

In the preceding code, we create a player entity (the spaceship), as well as two copies of the terrain mesh in order to make a nice big world to play around in. Notice that we are using the `clone()` function, so that the second copy of our terrain will share the same vertex and UV buffers, as well as texture and shader.

```
trace("Parsing the asteroid field...");
// add an asteroid field to the props list
asteroids1 = new Stage3dEntity(
    asteroidsObjData,
    context3D,
    shaderProgram1,
    cratersTexture);
asteroids1.rotationDegreesZ = 90;
asteroids1.scaleXYZ = 200;
asteroids1.y = 500;
asteroids1.z = -1000;
props.push(asteroids1);
```

```
trace("Cloning the asteroid field...");  
// use the same mesh in multiple locations  
asteroids2 = asteroids1.clone();  
asteroids2.z = -5000;  
props.push(asteroids2);  
  
asteroids3 = asteroids1.clone();  
asteroids3.z = -9000;  
props.push(asteroids3);  
asteroids4 = asteroids1.clone();  
asteroids4.z = -9000;  
asteroids4.y = -500;  
props.push(asteroids4);
```

In the previous code, we create an entity that uses our new asteroids mesh and then clones it a few times and puts it in different locations and at different angles, thus filling the sky with floating space debris.

```
trace("Parsing the engine glow...");  
engineGlow = new Stage3dEntity(  
    puffObjData,  
    context3D,  
    shaderProgram1,  
    puffTexture);  
// follow the player's ship  
engineGlow.follow(player);  
// draw as a transparent particle  
engineGlow.blendSrc = Context3DBlendFactor.ONE;  
engineGlow.blendDst = Context3DBlendFactor.ONE;  
engineGlow.depthTest = false;  
engineGlow.cullingMode = Context3DTriangleFace.NONE;  
engineGlow.y = -1.0;  
engineGlow.scaleXYZ = 0.5;  
particles.push(engineGlow);
```

In the preceding code, we create a "special effect" entity that uses a particle system mesh and an additive blend mode for a transparent, screen-lightening look that will be used as the "fire" that comes out of the player's engines. We "attach" it to the player entity, so that wherever the spaceship goes the engine glow will follow. Finally, we create a gigantic sphere that uses our new space background art, so that in the distance we will see stars and nebula rather than just a black void.

```
trace("Parsing the sky...");  
sky = new Stage3dEntity(  
    skyObjData,
```

```

        context3D,
        shaderProgram1,
        skyTexture);
    // follow the player's ship
    sky.follow(player);
    sky.depthTest = false;
    sky.depthTestMode = Context3DCompareMode.LESS;
    sky.cullingMode = Context3DTriangleFace.NONE;
    sky.z = 2000.0;
    sky.scaleX = 40000;
    sky.scaleY = 40000;
    sky.scaleZ = 10000;
    sky.rotationDegreesX = 30;
    props.push(sky);
}

```

What just happened?

We created several Stage3dEntity objects for use in our new improved game. Level props such as terrain and asteroids and the "sky", as well as a player spaceship and a glowing flame coming out of the engines will help to add detail to our game.

We are ready to program our new improved render function. It used to be messy and full of code, but we have since upgraded our game engine to use the entity class to do all the hard work.

Time for action – upgrading the render function

All that needs to be done to render our scene is to update the view matrix to match the location and orientation of our chase camera (while adding any angles required by the mouse-look functionality we coded in the GameInput class), and then loop through each of the entities in our game world and instruct them to render themselves.

```

private function renderScene():void
{
    viewmatrix.identity();
    // look at the player
    viewmatrix.append(chaseCamera.transform);
    viewmatrix.invert();
    // tilt down a little
    viewmatrix.appendRotation(15, Vector3D.X_AXIS);
    // if mouselook is on:
    viewmatrix.appendRotation(gameinput.cameraAngleX,
        Vector3D.X_AXIS);
    viewmatrix.appendRotation(gameinput.cameraAngleY,

```

```
    Vector3D.Y_AXIS);
viewmatrix.appendRotation(gameinput.cameraAngleZ,
    Vector3D.Z_AXIS);

// render the player mesh from the current camera angle
player.render(viewmatrix, projectionmatrix);

// loop through all known entities and render them
for each (entity in props)
    entity.render(viewmatrix, projectionmatrix);
for each (entity in enemies)
    entity.render(viewmatrix, projectionmatrix);
for each (entity in bullets)
    entity.render(viewmatrix, projectionmatrix);
for each (entity in particles)
    entity.render(viewmatrix, projectionmatrix);
}
}
```

What just happened?

We simplified our `renderScene` function by moving all low level Stage3D functionality to the new `Stage3dEntity` class. We first set the view projection matrix as in previous chapters, but now we can add extra rotations if the player is dragging the mouse to look around. We then instruct the player entity to render itself and then loop through all known props to render anything else we may have added to the world.

Not only do we want to render everything once per frame but we also need to update the in-game world. This simulation update step is a simple function now, but could eventually be extended to include physics, artificial intelligence, and all sorts of game event timer triggers and the like. For now, let's program a `step` function that takes into consideration the number of milliseconds that have elapsed since the last frame and moves various animated entities the appropriate distance.

Time for action – creating a simulation step function

Let's make this function update the player position depending on what inputs are being pressed, as well as change the location of our chase camera, smoothly animate the floating motion of the asteroids, and make the engine glow behind our spaceship pulsate.

```
private function gameStep(frameMs:uint):void
{
    // handle player input
    var moveAmount:Number = moveSpeed * frameMs;
    if (gameinput.pressing.up) player.z -= moveAmount;
    if (gameinput.pressing.down) player.z += moveAmount;
    if (gameinput.pressing.left) player.x -= moveAmount;
    if (gameinput.pressing.right) player.x += moveAmount;
```

```

//if (pressing.fire) // etc...

// follow the player
chaseCamera.x = player.x;
chaseCamera.y = player.y + 1.5; // above
chaseCamera.z = player.z + 3; // behind

// animate the asteroids
asteroids1.rotationDegreesX +=
    asteroidRotationSpeed * frameMs;
asteroids2.rotationDegreesX -=
    asteroidRotationSpeed * frameMs;
asteroids3.rotationDegreesX +=
    asteroidRotationSpeed * frameMs;
asteroids4.rotationDegreesX -=
    asteroidRotationSpeed * frameMs;

// animate the engine glow - spin fast and pulsate slowly
engineGlow.rotationDegreesZ += 10 * frameMs;
engineGlow.scaleXYZ = Math.cos(
    gametimer.gameElapsed / 66) / 20 + 0.5;
}

```

What just happened?

The preceding gameStep function will be called with a parameter that holds the number of milliseconds that have elapsed since the previous frame. This number is calculated by our new game timer class. In the preceding code, we then use this number to determine how far some animated entities need to be moved, so that they rotate at the same speed no matter what the framerate.

Time for action – creating a heartbeat function

For efficiency, this function only runs occasionally, which is ideal for calculations that don't need to be run every frame.

```

private function heartbeat():void
{
    trace('heartbeat at ' + gametimer.game_elapsed_time + 'ms');
    trace('player ' + player.pos_string());
    trace('camera ' + chase_camera.pos_string());
}

```

Our current heartbeat function is merely an empty routine, ready to be used as a place to put expensive calculations once your game becomes more complex. For now, all we do is output some debug information to the Flash debug window.

Time for action – upgrading the enterFrame function

We are nearly finished upgrading our game! All that remains is to tweak the function that is run every frame. Modify the `enterFrame` function as follows.

```
private function enterFrame(e:Event):void
{
    // clear scene before rendering is mandatory
    context3D.clear(0,0,0);

    // count frames, measure elapsed time
    gametimer.tick();

    // update all entities positions, etc
    gameStep(gametimer.frameMs);

    // render everything
    renderScene();

    // present/flip back buffer
    // now that all meshes have been drawn
    context3D.present();

    // update the FPS display
    fpsTicks++;
    var now:uint = getTimer();
    var delta:uint = now - fpsLast;
    // only update the display once a second
    if (delta >= 1000)
    {
        var fps:Number = fpsTicks / delta * 1000;
        fpsTf.text = fps.toFixed(1) + " fps";
        fpsTicks = 0;
        fpsLast = now;
    }
    // update the rest of the GUI
    updateScore();
}

} // end of class
} // end of package
```

What just happened?

Above is our trusty workhorse function, the `enterFrame` "render loop" which is run every single frame. As always, we clear the screen first. Then, we update the timer and step the game simulation to account for time passing. Finally, we render the scene, make it visible on the screen, and update the FPS display.

Let's see all this in action!

Now that we have made a massive upgrade to our game engine, you are probably excited about the implications of all this new processing power. Although we have just scratched the surface of what we can achieve with the new input, timer, and entity classes, you are undoubtedly filled with ideas for future enhancements that use them.

For now, we have a spaceship with an animated engine glow that you can control with the arrow keys or *W, A, S, D*. There is a chase camera that follows the player around. This camera has mouse-look aiming implemented, so you can hold the mouse button down and drag the cursor around to rotate the camera like a first person shooter. There is a sky in the background, and an attractive, animated asteroid field in our newly improved game world. Not bad for one chapter's work!

You can view the finished *Chapter 7* demo here:

http://www.mcfunkypants.com/molehill/chapter_7_demo/

You can view the full source code here:

<http://www.packtpub.com/adobe-flash11-stage3d-molehill-game-programming-beginners-guide/book>



Compile your project to ensure there are no typos and then run the `index.html` in your project folder. If everything went as planned, you should see a scene identical to the preceding image.

Pop quiz – earn some experience points

Here are a few quick questions to test your understanding. If you can answer them, you are officially ready to move on to the next step in your grand adventure.

1. Why does our `game_step` function require a time value parameter?
 - a. For debugging purposes
 - b. So it knows when to run
 - c. So it can animate things at the same speed no matter what the framerate
 - d. So it does not get bored
2. When adding things to our game world, why would we want to use the `Stage3dEntity.clone()` function?
 - a. To re-use the same vertex buffer to draw multiple meshes in different locations
 - b. So the game initializes faster
 - c. To use less RAM
 - d. All of the above
3. Why do we set the "dirty flag" `_transformNeedsUpdate` to true when we set one of the many location, rotation, or scale values in our `GameEntity` class?
 - a. So that our robots know when to transform into vehicles
 - b. So that the game engine knows which mesh to render
 - c. So that the next time the transform is requested we know it needs to be updated
 - d. So we know when it is time to wash our flags

Have a go hero – a fun side-quest: engage the afterburners!

Your side quest this time is to make something happen when the user presses the spacebar. At the moment, in your `gameStep()` function, you are able to check to see if `gameinput.pressing.fire` is true. Program a function that triggers some change in the game when you hit space.

A fun idea would be to give the spaceship a speed boost by temporarily increasing the value of `moveAmount` to a larger number. In order to make it temporary, store the current time (`gametimer.currentTime`) at the beginning of the speed boost and reset the speed to the original value after a couple of seconds have passed. This is a great use of the `GameInput` and `GameTimer` classes and is a good way to come to grips with working with them both.

Summary

In this chapter, we managed to achieve the following milestones:

- ◆ We now have a class that measures time for use in animations
- ◆ We also have a new class that detects the player keyboard and mouse input
- ◆ We created a game entity class for use by all in-game objects
- ◆ We programmed a simple chase camera with mouse-look
- ◆ We upgraded our game to include a heads-up-display overlay and many new in-game models and effects

Level 7 achieved!

Congratulations, brave hero. Yet again, you have successfully risen to the challenge and emerged victorious. Your game is evolving at an exponential pace. Not only is starting to really feel like a game, but also the infrastructure is now in place to be able to focus on gameplay. No longer are we struggling with inits or learning the fundamentals. We are now working on reusable, abstract classes that provide common game-specific functionality.

You have definitely leveled-up your skills. This is getting exciting!

You have earned the right to start coding next-gen special effects such as explosions, sparks, glows, smoke, and particles. It is finally time to focus on the eye candy. This and more await you in the next chapter.

8

Eye-Candy Aplenty!

You have reached Level 8.

The next step in our quest for graphical glory is to implement some eye-candy. Particle systems and special effects are essential for giving your game a visual impact.

Things are getting really exciting now. You can probably see the light at the end of the tunnel already: our game is taking shape and already we have learned so much. Now that we have achieved some basic gameplay—movement, animation, a GUI heads-up-display, timers and input classes—the next step is to make it look good.

In this chapter, we are going to implement a system capable of pumping out massive amounts of special effects. There are serious performance considerations to take into account—after all, we don't want our special effects to destroy the silky smooth framerate.

Our current quest

With optimization and abstract reuseability in mind, let's create a particle system manager capable of rendering insane numbers of particles while having little effect upon the framerate. We want to be able to use it to pump out everything from explosions, smoke, sparks, shock waves, blood splatters, lens flares, splashes of water or lava, and atmospheric motes of dust floating around in the air.

Eye-Candy Aplenty!

What we want is to be able to render TONS of particles with great framerate:



Before we dive in, we should take note of some design goals for our fancy new particle system.

Designing for performance

Many Flash 3D game engine authors lately have been having trouble achieving good particle system performance. The reason is that our first instinct is to simply draw thousands of billboard sprites individually. Of course, it turns out that there is overhead in calculating the transform matrix and render state for each draw call and the only way we can achieve large numbers of particles in Flash is to "batch" them in larger chunks. Even better, if we can avoid doing any of the animation simulation on the CPU and offload all calculations on the GPU using AGAL, our particle engines can benefit from virtually no CPU use.

We want a particle system that allows mega complex particle systems of nearly infinite numbers of polies. One possible solution is to create a specialized vertex buffer for the entire particle system.

With this technique, all particle system animation would be handled on the GPU with virtually no rendering overhead for animation of gigantic particle systems. Why? The entire thing is rendered in ONE draw call. Essentially, you are "pre-calculating" the entire animation for your explosions and other effects. No worrying about each and every sprite in an explosion, you just draw the whole group at the same time and get the GPU to do all the heavy lifting.

By using this technique, we will be able to implement a fully GPU animated particle system (60fps with 40,000 particles each moving in different directions, rotating, and scaling larger while fading out) that uses AGAL for all simulation—not the CPU.

Designing for reusability

We don't want to hard-code our special effects rendering for any particular kind of effect. Therefore, we will aim for the ability to send any kind of mesh to the system: from single billboard sprites consisting of a basic quad to an entire group of polygons that make up a complex explosion.

Another form of reuseability is the ability to reuse previously created particles over and over. We also don't want to bog down the CPU and eat up tons of RAM with each frame by creating and destroying thousands of meshes over and over. Doing so will kill the framerate because the time it takes to upload mesh data (or even create simple AS3 objects) adds up—if we are going to be spawning particles over and over during the game there is no reason to continually initialize them and then destroy them when they disappear.

Therefore, apart from doing all simulation on the GPU, another optimization technique used here is to create a "particle pool", so that particles are reused if inactive. This avoids any GC (garbage collection) issues which result in framerate hiccups: we don't "spawn" new particles each time there is a new explosion, we just reuse old ones and only create new ones if there are not any inactive ones available. The result of doing so is that the framerate stays near 60fps, whether we have forty thousand particles on the screen or none at all.

Animating using AGAL

The AGAL and scene setup for the technique outlined earlier is relatively simplistic. We create two vertex buffers. One contains the start positions and the other the end positions, pre-calculated, for each vertex. For example, instead of creating thousands of GameEntities, create a single 10,000 poly "mesh" with vertex buffer data as follows: $x1,y1,z1,x2,y2,z2$. Instead of moving each and every particle around through AS3 matrix math, the motion of the sprite is pre-calculated and stored in the vertex buffer itself.

Once we have created this particle "batch geometry", we can send a delta value as a vertex buffer constant register that varies from 0..1 over time. In each frame, when we render the mesh, we will have your AGAL vertex program interpolate the x,y,z position of each vertex from the starting to the ending position. Each vertex will smoothly move from the first to the second position.

By pre-calculating the motions of each particle once and then simply getting AGAL to smoothly move from one location to the next, we can simulate gravity, wind, the expanding size of things such as smoke dissipating into the air, and more.

In order to achieve great performance during gameplay, we don't want to do any complex calculations in AS3. Instead, once we have filled the two vertex buffers described earlier, all we need to do for each frame is instruct Stage3D how much time has passed. Before we render our particle mesh, we send this value to Stage3D for use in the shader.

The key design consideration here is to NOT render or simulate each particle separately. Depending on your computer's horsepower (and in particular your video card's fill rate), it is entirely feasible that your game can boast scenes containing 100,000+ particles while still running at 60fps—with room to spare for all your other scene models, as we are not using the CPU for any of the heavy lifting.

In order to successfully complete this quest, we need to perform the following:

- ◆ Create a basic particle entity class
- ◆ Program the AGAL for keyframed vertex animation
- ◆ Code a particle system manager to control multiple particles
- ◆ Design some nice looking effect art
- ◆ Incorporate the particle system into our game

Time to upgrade our game's eye-candy. This efficient "batched GPU particle system" technique will give you the pizazz required for some awesome screenshots and in-game action sequences. Let's get started!

A basic particle entity class

The first step in adding eye-candy to our game is to define what a particle is. Eventually, our game will spawn hundreds of particles for use in all sorts of special effects, so let's make it abstract and able to be used for numerous kinds of particles—from splashes of water to sparks that fly from clashing swords.

One particle in our special effects system can be a single polygon or a premade "batch" of many polygons. As we want to have many polies with minimal framerate overhead, we are going to treat entire effects (such as an explosion) as a single entity even though the mesh it uses will consist of hundreds or thousands of polies.

We want our particle entity class to hold mesh data, as well as a texture and shader. This sounds suspiciously similar to the `Stage3dEntity` class we have already programmed. The great news is that we only need to add a few minor extras to this basic entity class, so instead of creating a brand new kind of AS3 object from scratch, we will simply extend the basic entity class and add a few extra functions and variables.

Time for action – extending the entity class for particles

Create a new file in your source code folder named `Stage3dParticle.as`. Import the basic functions we will need and simply extend the base class as follows:

```
// Game particle class version 1.0
//
package
{

    import com.adobe.utils.*;
    import flash.display.Stage3D;
    import flash.display3D.Context3D;
    import flash.display3D.Context3DProgramType;
    import flash.display3D.Context3DTriangleFace;
    import flash.display3D.Context3DVertexBufferFormat;
    import flash.display3D.IndexBuffer3D;
    import flash.display3D.Program3D;
    import flash.display3D.VertexBuffer3D;
    import flash.display3D.*;
    import flash.display3D.textures.*;
    import flash.geom.Matrix;
    import flash.geom.Matrix3D;
    import flash.geom.Vector3D;

    public class Stage3dParticle extends Stage3dEntity
    {
}
```

What just happened?

The last line in the preceding code instructs Flash to "extend" or take all the functions that were implemented in our `Stage3dEntity` class and assume they are the same in this new class. In this way, our particle class instantly has all the helper functions in its repertoire that we worked so hard on before.

This use of class inheritance helps save a lot of time and effort by allowing us to reuse the code. It keeps basic functionality in one place. If we upgrade this base class or fix a bug, then all classes that extend it automatically get upgraded as well.

Time for action – adding particle properties

Next, we want to give our new class a few extra properties related to particles. We do this as follows:

```
public var active:Boolean = true;
public var age:uint = 0;
public var ageMax:uint = 1000;
public var stepCounter:uint = 0;

private var mesh2:Stage3dObjParser;
private var ageScale:Vector.<Number> =
    new Vector.<Number>([1, 0, 1, 1]);
private var rgbaScale:Vector.<Number> =
    new Vector.<Number>([1, 1, 1, 1]);
private var startSize:Number = 0;
private var endSize:Number = 1;
// we only want to compile the shaders once
private static var particleshader1mesh:Program3D = null;
private static var particleshader2mesh:Program3D = null;
```

What just happened?

These properties will be used to simulate our particle systems. By keeping track of age, we can calculate the proper size and color over time, so that an explosion, for example, starts small and expands in size while fading out. The times are stored in milliseconds, so the defaults above will create a particle that exists for exactly one second.

Time for action – coding the particle class constructor

Now that we have defined the variables we need, let's write a new class constructor function. We don't want to use the one that comes with our original `Stage3dEntity` class, so we use the keyword "override" to replace it with this new version. Though it is similar to a basic entity, our particles can have an optional second mesh that is used for keyframed vertex position animation. We also don't need to pass in a shader as we will be defining a special AGAL shader later.

```
// Class Constructor - the second mesh defines the
// ending positions of each vertex in the first
public function Stage3dParticle(
    mydata:Class = null,
    mycontext:Context3D = null,
    mytexture:Texture = null,
```

```
mydata2:Class = null
)
{
    transform = new Matrix3D();
    context = mycontext;
    texture = mytexture;

    // use a shader specifically designed for particles
    // this version is for two frames: interpolating from one
    // mesh to another over time
    if (context && mydata2) initParticleShader(true);
    // only one mesh defined: use the simpler shader
    else if (context) initParticleShader(false);

    if (mydata && context)
    {
        mesh = new Stage3dObjParser(
            mydata, context, 1, true, true);
        polycount = mesh.indexBufferCount;
        trace("Mesh has " + polycount + " polygons.");
    }

    // parse the second mesh
    if (mydata2 && context)
        mesh2 = new Stage3dObjParser(
            mydata2, context, 1, true, true);

    // default a render state suitable for particles
    blendSrc = Context3DBlendFactor.ONE;
    blendDst = Context3DBlendFactor.ONE;
    cullingMode = Context3DTriangleFace.NONE;
    depthTestMode = Context3DCompareMode.ALWAYS;
    depthTest = false;
}
```

What just happened?

This constructor simply sets up a particle entity to be rendered with the correct blend mode. If we pass a second mesh in the parameters, that mesh is also parsed and ready for use in our shader. Depending on whether this is a particle that uses a single mesh or interpolates in shape from one to the next, the appropriate shader is created.

Time for action – cloning particles

As we only need to compile the AGAL shader or upload mesh geometry to Stage3D once for each kind of particle, we avoid wasting RAM by creating a special clone function:

```
public function cloneparticle():Stage3dParticle
{
    var myclone:Stage3dParticle = new Stage3dParticle();
    updateTransformFromValues();
    myclone.transform = this.transform.clone();
    myclone.mesh = this.mesh;
    myclone.texture = this.texture;
    myclone.shader = this.shader;
    myclone.vertexBuffer = this.vertexBuffer;
    myclone.indexBuffer = this.indexBuffer;
    myclone.context = this.context;
    myclone.updateValuesFromTransform();
    myclone.mesh2 = this.mesh2;
    myclone.startSize = this.startSize;
    myclone.endSize = this.endSize;
    myclone.polycount = this.polycount;
    return myclone;
}
```

What just happened?

All this function does is copy the variables needed to a new copy of the current particle. This is one of the various optimizations that will allow our game to render TONS of particles using minimal system resources.

Time for action – generating numbers used for animation

The next two functions are simply handy routines that you may want to use for particle simulation. They smoothly interpolate a value using the sine function such that it "wobbles" in and out. This function is really handy as you design particles that are supposed to bob up and down, or fade in and then fade out smoothly.

```
private var twoPi:Number = 2*Math.PI;
// returns a float from -amp to +amp in wobbles per second
private function wobble(
    ms:Number = 0, amp:Number = 1, spd:Number = 1):Number
{
    var val:Number;
    val = amp*Math.sin((ms/1000)*spd*twoPi);
```

```

        return val;
    }

    // returns a float that oscillates from 0..1..0 each second
    private function wobble010(ms:Number) :Number
    {
        var retval:Number;
        retval = wobble(ms-250, 0.5, 1.0) + 0.5;
        return retval;
    }
}

```

What just happened?

What do these "wobble" functions do? They are used to generate numbers that smoothly change from zero to one (and possibly back again) based on how much time has passed. They are sent to the AGAL code in our particle shader to fade the particle in and out and to make it grow in size.

Time for action – simulating the particles

The next step is to code a `step` function. This function will eventually be called once every single frame, for every single particle, by your particle system manager class. Therefore, it is important for this function to be as simple as possible.

```

public function step(ms:uint) :void
{
    stepCounter++;
    age += ms;
    if (age >= ageMax)
    {
        //trace("Particle died (" + age + "ms)");
        active = false;
        return;
    }
    // based on age, change the scale for starting pos (1..0)
    ageScale[0] = 1 - (age / ageMax);
    // based on age, change the scale for ending pos (0..1)
    ageScale[1] = age / ageMax;
    // based on age: go from 0..1..0
    ageScale[2] = wobble010(age);
    // ensure it is within the valid range
    if (ageScale[0] < 0) ageScale[0] = 0;
    if (ageScale[0] > 1) ageScale[0] = 1;
    if (ageScale[1] < 0) ageScale[1] = 0;
}

```

```
if (ageScale[1] > 1) ageScale[1] = 1;
if (ageScale[2] < 0) ageScale[2] = 0;
if (ageScale[2] > 1) ageScale[2] = 1;
// fade alpha in and out
rgbaScale[0] = ageScale[0];
rgbaScale[1] = ageScale[0];
rgbaScale[2] = ageScale[0];
rgbaScale[3] = ageScale[2];
}
```

What just happened?

Using the time elapsed since the previous frame, this function uses the wobble functions to generate a list of numbers that smoothly go from zero to one, or one to zero, depending on what is required.

It first checks to see if the age of the particle has advanced beyond its desired lifetime. If so, the particle no longer needs to be updated or rendered. If it is instead still active, we calculate the `ageScale` and `rgbaScale` values which will be used by our shader to do the animation.

Time for action – respawning particles

As mentioned earlier, we only want to create brand new particles when there are not any inactive particles that could be reused instead. This is how we are able to get so many polies on the screen with such great performance. By avoiding creating and destroying new objects for every frame, we keep the memory requirements constant during the game loop.

```
public function respawn(
    pos:Matrix3D, maxage:uint = 1000,
    scale1:Number = 0, scale2:Number = 50):void
{
    age = 0;
    stepCounter = 0;
    ageMax = maxage;
    transform = pos.clone();
    updateValuesFromTransform();
    rotationDegreesX = 180; // point "down"
    // start at a random orientation each time
    rotationDegreesY = Math.random() * 360 - 180;
    updateTransformFromValues();
    ageScale[0] = 1;
    ageScale[1] = 0;
    ageScale[2] = 0;
```

```

ageScale[3] = 1;
rgbaScale[0] = 1;
rgbaScale[1] = 1;
rgbaScale[2] = 1;
rgbaScale[3] = 1;
startSize = scale1;
endSize = scale2;
active = true;
//trace("Respawned particle at " + posString());
}

```

What just happened?

The respawn function is used by our upcoming particle system manager to reset the variables required, so that an existing particle is ready for "a new life". Whenever our games need to create a new explosion, for example, we simply need to instruct an existing particle to start animating at a new location. We can also tweak the size or lifespan of our particle, if needed, so we can do such things as create bigger or smaller explosions depending on the amount of damage inflicted.

One important note: we rotate our explosion mesh to "point down" in this example because the example data in our second .OBJ file has each particle in a slightly higher location. Depending on your source art, you may not want to include this line. You might also want to enhance this routine to be able to turn off the random orientation, for example, which is used here to make each explosion look more different from the previous.

The ageScale and rgbaScale variables are initialized with values that make sense for the very first frame of our particle shader. We will be changing these values in every frame in order to fade in or grow the mesh by sending them to our AGAL shader for use as vertex and fragment constants.

Time for action – rendering particles

Just as we extended the default Stage3dEntity class constructor function, we also want to write our own particle rendering function. Override the render function as follows:

```

// optimization: reuse the same temporary matrix
private var _rendermatrix:Matrix3D = new Matrix3D();
override public function render(
    view:Matrix3D,
    projection:Matrix3D,
    statechanged:Boolean = true):void
{
    // only render if these are set
    if (!active) return;
}

```

Eye-Candy Aplenty!

```
if (!mesh) return;
if (!context) return;
if (!shader) return;
if (!texture) return;

// get bigger over time
scaleXYZ = startSize +
    ((endSize - startSize) * ageScale[1]);

//Reset our matrix
_rendermatrix.identity();
_rendermatrix.append(transform);
if (following) _rendermatrix.append(following.transform);
_rendermatrix.append(view);
_rendermatrix.append(projection);

// Set the vertex program register vc0
// to our model view projection matrix
context.setProgramConstantsFromMatrix(
    Context3DProgramType.VERTEX, 0, _rendermatrix, true);
```

We start by first updating the scale of our mesh to grow or shrink our particle depending on the values we passed to the respawn function. This simple scaling updates our transform matrix using the `ageScale` variable. It is then mixed with the view and projection matrix in the same way as we have been doing in all previous rendering examples.

```
// Set the vertex program register vc4
// to our time scale from (0..1)
// used to interpolate vertex position over time
context.setProgramConstantsFromVector(
    Context3DProgramType.VERTEX, 4, ageScale);

// Set the fragment program register fc0
// to our time scale from (0..1)
// used to interpolate transparency over time
context.setProgramConstantsFromVector(
    Context3DProgramType.FRAGMENT, 0, rgbaScale);
```

Next, we send our interpolation values to the shader. By filling a vertex program constant with `ageScale` and fragment program constant with `rgbaScale`, our Stage3D shader will be able to do the rest of the work for us. Remember that these two variables contain a smoothly interpolated list of numbers (four in each) that go from 1 to 0 or 0 to 1 over time as needed. The `step()` function is where these values are updated in each frame.

```
// Set the AGAL program
context.setProgram(shader);
// Set the fragment program register ts0 to a texture
context.setTextureAt(0,texture);
// starting position (va0)
context.setVertexBufferAt(0, mesh.positionsBuffer,
    0, Context3DVertexBufferFormat.FLOAT_3);
// tex coords (va1)
context.setVertexBufferAt(1, mesh.uvBuffer,
    0, Context3DVertexBufferFormat.FLOAT_2);
// final position (va2)
if (mesh2)
{
    context.setVertexBufferAt(2, mesh2.positionsBuffer,
        0, Context3DVertexBufferFormat.FLOAT_3);
}

// set the render state
context.setBlendFactors(blendSrc, blendDst);
context.setDepthTest(depthTest,depthTestMethod);
context.setCulling(cullingMode);

// render it
context.drawTriangles(mesh.indexBuffer,
    0, mesh.indexBufferCount);
} // render function ends
```

What just happened?

The particle render function sends numbers based on how much time has passed to our special particle shader, so that the mesh will be rendered at the correct size and transparency. These numbers are calculated in the `step()` function and are stored in AGAL registers for use in our vertex and fragment programs.

Finally, we instruct Flash which vertex buffers and shader to use, set the render state so that it uses transparency properly (in this example, additively brightening the scene), and render our mesh.

There is only one missing function in our particle class—the AGAL shader—which deserves a more detailed discussion.

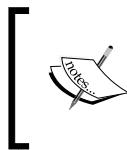
Keyframed vertex animation shader

The final touch required to complete our particle class is to program the shader that does all the heavy lifting during our game loop. In the constructor function for our particle class, we call this `initParticleShader` function and it returns a shader that uses either two meshes (for keyframed animation), or just one if that is all that is needed.

As mentioned in previous chapters, a Stage3D shader is made up of a vertex program and a fragment program. The former handles the positions, UV coordinates, and more for each vertex in your mesh, while the latter decides what color and transparency each pixel on the screen is when your mesh is rendered.

As we want to get your video card's GPU to do all the matrix math for the movement of our particles (freeing up the CPU to think about other things), we need to write an AGAL vertex program that can move your mesh's vertexes around over time.

One way to do this is to store the final positions of each vertex alongside the start positions and simply interpolate from one value to the next over time. An easy way to set this up is to sculpt an explosion, for example, in whatever 3D modeling application you prefer. Then, after exporting the first version of your mesh as an `.OBJ` file for use in your game, scale, rotate, and move the shape around and export this second mesh.



This is a very simplistic form of "keyframe" animation. Frame one of your animations is the first mesh and frame two is the second. The shader morphs the shape of our mesh from the first to the second over time.



This two-mesh technique is optional: if you wish, you can pass only one mesh to your constructor and simply scale the entire mesh as a whole (using the parameters of the `respawn` function) while fading it out. However, for added motion (and so that each particle in your effect can move independently rather than as a whole) this two-frame animation approach is easy to achieve and results in great-looking visuals.

Time for action – creating a keyframed particle vertex program

Add the following to the very end of your `Stage3dParticle` class:

```
private function initParticleShader(twomodels:Boolean=false):void
{
    var vertexShader:AGALMiniAssembler =
        new AGALMiniAssembler();
    var fragmentShader:AGALMiniAssembler
        = new AGALMiniAssembler();
    if (twomodels)
```

```

{
    if (particleshader2mesh)
    { // already compiled previously?
        shader = particleshader2mesh;
        return;
    }
    trace("Compiling the TWO FRAME particle shader..."); 
    vertexShader.assemble
    (
        Context3DProgramType.VERTEX,
        // scale the starting position
        "mul vt0, va0, vc4.xxxx\n" +
        // scale the ending position
        "mul vt1, va2, vc4.yyyy\n" +
        // interpolate the two positions
        "add vt2, vt0, vt1\n" +
        // 4x4 matrix multiply to get camera angle
        "m44 op, vt2, vc0\n" +
        // tell fragment shader about UV
        "mov v1, val"
    );
}

```

What just happened?

The preceding AGAL code is used to interpolate the vertex positions of mesh 1 with the vertex positions defined by mesh 2. In our render function, we use the function `setVertexBufferAt` to set up `va0` to contain the first vertex buffer and `va2` as the second. We also sent the variable `ageScale`, the list of values that smoothly change from 0 to 1 over time to `Stage3D`, using the `setProgramConstantsFromVector`. This four number list now resides in the `vc4` vertex constant register, ready for use.

For each vertex, the preceding shader multiplies the xyz position of the first mesh (stored in `va0`) by the first number stored in `vc4` (which in our example starts at 1 and gradually goes to 0). Therefore, the first line of AGAL above scales the location and stores this value in the temporary register `vt0`.

The second line of AGAL above does the same for the "frame two" mesh, except it multiplies it by the second number in `ageScale` (the y component of `vc4`) and stores it in the `vt1` temporary register.

Finally, these two-scaled locations (`vt0` and `vt1`) are added together and stored in `vt2`, which is now the interpolated value. Depending on how much time has passed, it represents a point somewhere between the first and second vertex position. This new location is a matrix multiplied by the model view projection matrix in the same way that all previous vertex programs have, so that our camera angle is taken into account. Finally, we pass the UV texture coordinates for each vertex in varying register `v1` for use in our fragment shader.

Time for action – creating a static particle vertex program

Continuing with this function, add the following:

```
else
{
    if (particleShader1mesh)
        { // already compiled previously?
            shader = particleShader1mesh;
            return;
        }
    trace("Compiling the ONE FRAME particle shader...");  
vertexShader.assemble
(
    Context3DProgramType.VERTEX,
    // get the vertex pos multiplied by camera angle
    "m44 op, va0, vc0\n" +
    // tell fragment shader about UV
    "mov v1, val"
);
}
```

What just happened?

In the preceding code, if we are using a single mesh particle (with no keyframe animation), then the AGAL for the vertex shader is extremely simplistic: it just instructs Stage3D to account for the camera angle and stores the proper vertex position in the op (output position) register.

Time for action – creating a particle fragment program

Now that we have programmed an AGAL vertex program that can smoothly interpolate from one location to the next over time, we can do something similar in the fragment program to generate the proper pixel color on the screen. Continuing with the `initParticleShader` function:

```
// textured using UV coordinates
fragmentShader.assemble
(
    Context3DProgramType.FRAGMENT,
    // grab the texture color from texture 0
    // and uv coordinates from varying register 1
    // and store the interpolated value in ft0
    "tex ft0, v1, fs0 <2d,linear,repeat,miplinear>\n" +
    // multiply by "fade" color register (fc0)
    "mul ft0, ft0, fc0\n" +
    // move this value to the output color
    "mov oc, ft0\n"
);
```

What just happened?

In the AGAL above, we create a fragment program that samples the texture that our render function defined as `fs0` at the proper location based on the UV vertex coordinates that our vertex program passed in varying register `v1`. The color of the texture at this point is stored in a fragment program temporary register `ft0`.

Before rendering using this value, we want to be able to smoothly fade out over time, so that our explosions or smoke puffs don't pop out abruptly, but instead fade to nothingness. In our render function, we sent the variable `rgbaScale` to Stage3D in `fc0`, a fragment constant register. Just as the timescale used in our vertex program, the four numbers stored in `fc0` will range from zero to one to zero again—perfect for a nice smooth fade.

In the second line of AGAL above, we simply multiply the texture color that we got in the first step by the fading value stored in `fc0`. We then store the final, properly faded out color in the `oc` (output color) register.

Time for action – compiling the particle shader

Now that we have written the proper AGAL for use in our particles, we simply compile them together to form a shader and upload it to the graphics card.

```
// combine shaders into a program and upload to the GPU
shader = context.createProgram();
shader.upload(
    vertexShader.agalcode,
    fragmentShader.agalcode);

// remember this shader for reuse
if (twomodels)
    particleshader2mesh = shader;
else

    particleshader1mesh = shader;
} // end initParticleShader function

} // end class

} // end package
```

That is it for our fancy new `Stage3dParticle` class! You could start using this class in place of our now-familiar `Stage3dEntity` in your game if you wanted to spawn a single unique particle here and there. As this class is derived from `Stage3dEntity`, you could even make it "follow" something else. This would be perfect if you wanted to attach a "glow" to a lamppost, for example.

This class is perfect for one-offs—particles that are meant to persist for a long time in your game. However, for temporary special effects such as puffs of smoke or blood splatters, we need a particle system manager capable of iterating through lists of hundreds of particles and reusing them over and over.

A particle system manager class

In addition to batching geometry and doing the vertex animation and fade-out in AGAL, the other great optimization used for achieving good particle performance has nothing to do with the Stage3D API specifically and is instead a standard game system optimization: a "pool" of particles that can be reused.

The second class we need to program in order to fill our game with the eye-candy we desire is a manager of numerous particles. This system will track time passing and will call the `step` and `render` functions for all active particles. Whenever our game needs to create a special effect of some kind, this particle system will get the job done.



For performance reasons, we don't want to be creating and destroying objects during the render loop, so instead we simply reuse inactive particles whenever possible. If the entire queue of particle entities is currently visible, only THEN do we create a new object.

By avoiding any variable allocations or destructions during the game, we never run into GC (garbage collection) issues. GC issues cause stutters in your game and reduce the framerate. They are caused by temporary variables that are no longer in use being destroyed by Flash. Even worse, if you don't minimize the creation of variables during game play, you will get what is called a "memory leak"—rising use of system RAM that never ends.

As the computer's memory becomes full of useless junk, the browser will eventually crash. This is why we have been careful to avoid the creation of too many temporary variables, and why we would never want to program a particle system that creates a new `Stage3dParticle` each time we want a special effect to take place—eventually there would be millions in memory, slowing things down.

By creating a class that intelligently reuses inactive objects that were created earlier, we save all these hassles and gain the benefit of much smoother framerates!

Time for action – coding a particle system manager class

This controller class will be much simpler than the particle class we programmed earlier. The particle system manager is simply a control mechanism that tries to reuse inactive particles whenever possible. Create a new file in your project folder named GameParticlesystem.as and define our new class as follows:

```
// Game particle system manager version 1.0
// creates a pool of particle entities on demand
// and reuses inactive ones whenever possible
//
package
{
    import flash.utils.Dictionary;
    import flash.geom.Matrix3D;
    import flash.geom.Vector3D;
    import Stage3dParticle;

    public class GameParticlesystem
    {
        // contains one source particle for each kind
        private var allKinds:Dictionary;
        // contains many cloned particles of various kinds
        private var allParticles:Dictionary;
        // temporary variables - used often
        private var particle:Stage3dParticle;
        private var particleList:Vector.<Stage3dParticle>;
        // used only for stats
        public var particlesCreated:uint = 0;
        public var particlesActive:uint = 0;
        public var totalpolycount:uint = 0;

        // class constructor
        public function GameParticlesystem()
        {
            trace("Particle system created.");
            allKinds = new Dictionary();
            allParticles = new Dictionary();
        }
    }
}
```

What just happened?

In the preceding code, we simply define our new class and set up a few handy variables. The `allKinds` list contains the "master copy" of each new type of particle that we need for our game.

The `allParticles` list is where the renderable particles are stored. This list will eventually be filled with hundreds of particles of different kinds, both active and inactive. For each frame we will check this list to see which particles need updating or rendering.

Time for action – defining a type of particle

```
// names a particular kind of particle
public function defineParticle(
    name:String, cloneSource:Stage3dParticle):void
{
    trace("New particle type defined: " + name);
    allKinds[name] = cloneSource;
}
```

What just happened?

The preceding function is meant to be called during your game initialization. We pass a name as well as a `Stage3dParticle` (which contains the mesh geometry, shader, texture, and the like). This particle will be used as the source for all future particles of that particular kind. None of these particles is ever rendered: they are simply associated with a string name such as "explosion" or "water splash" and are used to provide something to clone.

Time for action – simulating all particles at once

The `step` function is called in every frame during our game loop. In this way, our game needs only to instruct the entire system to step to ensure that each of the potentially thousands of particles is updated.

```
// updates the time step shader constants
public function step(ms:uint):void
{
    particlesActive = 0;
    for each (particleList in allParticles)
    {
        for each (particle in particleList)
        {
            if (particle.active)
            {
                particlesActive++;
                particle.step(ms);
            }
        }
    }
}
```

What just happened?

When running the particle system's `step` function, our game engine will need to pass the elapsed time in milliseconds following the previous frame. In this way, all particle animations are not framerate-dependent and will animate at the same rate regardless of system performance.

It loops through each particle type (based on name) in the `allParticles` list, and then each particle of that type that has been created so far. Any particle that is currently active is stepped forward in time. Each particle's `step` function we defined in our `Stage3dParticle` class earlier is run, so that we can update the movements and transparency as necessary.

Time for action – rendering all particles at once

As above, when the particle system needs to be rendered, we need to loop through every particle and instruct it to draw itself.

```
// renders all active particles
public function render(view:Matrix3D,projection:Matrix3D):void
{
    totalpolycount = 0;
    for each (particleList in allParticles)
    {
        for each (particle in particleList)
        {
            if (particle.active)
            {
                totalpolycount += particle.polycount;
                particle.render(view, projection);
            }
        }
    }
}
```

What just happened?

The particle system's `render` function iterates through all active particles and runs their corresponding `render` function, so that they are displayed on the screen. We are keeping track of how many polygons are rendered in each frame (primarily for boasting purposes).

Time for action – spawning particles on demand

The final function needed by our particle system manager class is responsible for either creating new particles or reusing old, inactive ones. Our game engine is going to use the `spawn` function to "ask" that a new particle be triggered.

```
// either reuse an inactive particle or create a new one
public function spawn(
    name:String, pos:Matrix3D, maxage:Number = 1000,
    scale1:Number = 1, scale2:Number = 50):void
{
    var reused:Boolean = false;
    if (allKinds[name])
    {
        if (allParticles[name])
        {
            for each (particle in allParticles[name])
            {
                if (!particle.active)
                {
                    //trace("A " + name + " was reused.");
                    particle.respawn(pos, maxage, scale1, scale2);
                    particle.updateValuesFromTransform();
                    reused = true;
                    return;
                }
            }
        }
    }
}
```

What just happened?

We first check to see if that particular kind of particle has been previously defined by name. If it is a known type, then we iterate through all previously created particles of that kind and look for an inactive one that is available for reuse. If we find one, we simply call its `respawn()` function to reset some variables and get it animating again in the proper new location.

Time for action – creating new particles if needed

We need to account for situations where all available particles are currently active and none are available for reuse. Continuing with the particle system's `spawn()` function:

```
else
{
    trace("This is the first " + name + " particle.");
```

```

        allParticles[name] = new Vector.<Stage3dParticle>;
    }
    if (!reused) // no inactive ones were found
    {
        particlesCreated++;
        trace("Creating a new " + name);
        trace("Total particles: " + particlesCreated);
        var newParticle:Stage3dParticle =
            allKinds[name].cloneparticle();
        newParticle.respawn(pos, maxage, scale1, scale2);
        newParticle.updateValuesFromTransform();
        allParticles[name].push(newParticle);
    }
}
else
{
    trace("ERROR: unknown particle type: " + name);
}
}

} // end class
} // end package

```

What just happened?

If no inactive particles of the desired type are available, then it is time to use up a little more memory and create a fresh new clone of our appropriate "master" particle. We make a new one and from this point forward, our `allParticles` list is a little bit bigger.

That is it for our fancy new particle system class. As you can see, by creating particles "on demand" we only create enough to satisfy the needs of our game engine. This technique has many advantages, including the fact that RAM consumption stays relatively constant once a certain number of particles have been created.

It is still entirely possible to bog down your computer by spawning so many particles that the system simply cannot handle the load. In particular, if you spawn too many particles in close succession, you might get into situations where all available particles are currently visible on the screen and new ones need to be created. This can happen when you give particles too long a `maxAge`, so that they exist in an active state for too long.

Even with this new optimized particle system solution, we have to be careful to only create as many particles as needed and not go overboard. If you are generating scenes with more than 100,000 polies used just for particles, then you should rethink your special effects. In particular, remember that even if your game is running smoothly on your power gaming PC, users trying to play it on wimpy netbooks (or tablets and smartphones) will not have the same amount of horsepower.

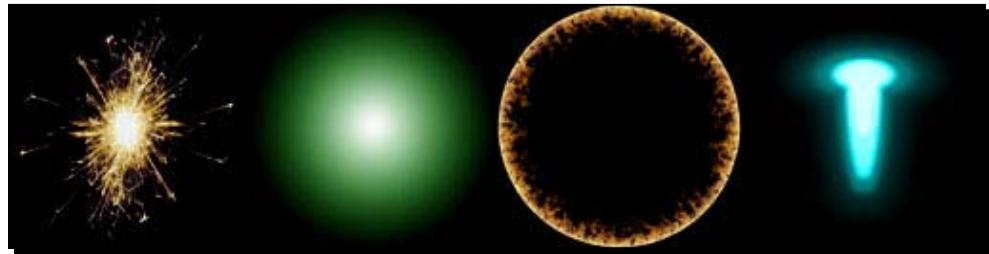
Perhaps you can get by with larger particles that use fewer polies, or make your particles fade out more quickly so fewer are on the screen at any one time. Finally, remember that one mesh with 500 polygons renders much faster than 500 meshes of one polygon each.

Keyframed particle meshes

In order to test our fancy new particle system, we need some cool-looking art assets to use. We will need a few new meshes and some transparent textures. The key technique here is the creation of two meshes for use as the keyframes for our particle system. We are aiming to create the "before" and "after" for each effect.

Selecting a particle texture

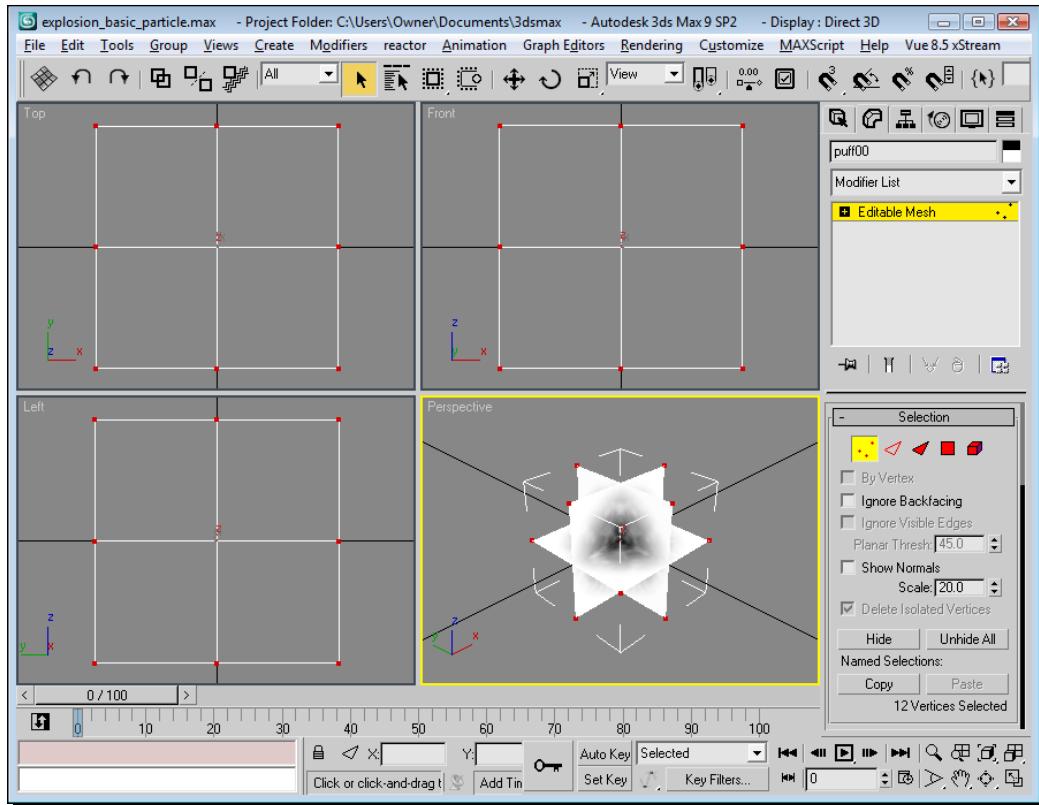
As the purpose of this book is to focus on the programming side of the game design, the art creation work is up to you. Here are four textures that we could use in this example. These were made using Photoshop and saved as .PNG files with transparency. As we are going to render with an additive (whitening) blendmode, we can save them as small (128x128) .JPG images with a black background in order to keep the file size of your SWF minuscule:



Time for action – sculpting a single particle

Now that we have some lovely looking textures, let's get them rendered in 3D. We don't want to waste CPU by animating each and every sprite individually in our game, so instead we want to "sculpt" some nice looking "blobs" comprised of multiple polygons. Fire up your favorite 3D modeling application such as 3D Studio Max, Blender, or Maya, and create a still version of a few kinds of special effect that you wish to use.

As a case study, we will go through the process of making a frequently used example: an explosion. Start by creating a flat quad plane and texture it. This is a single particle.



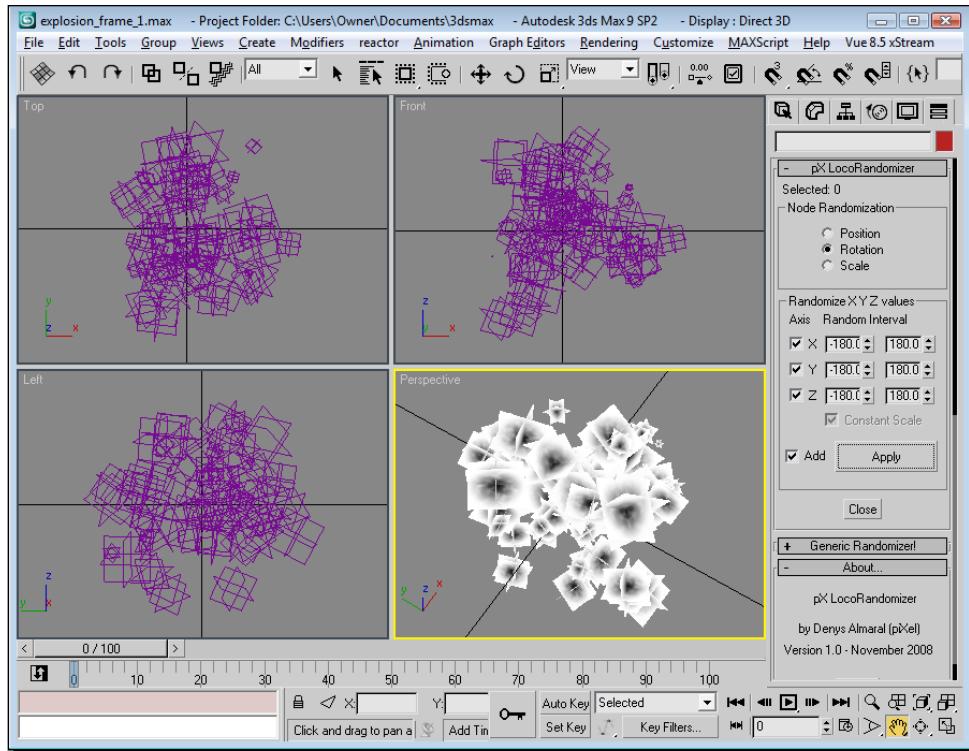
Time for action – sculpting a group of particles

In a typical explosion or water splash or group of sparks, we generally expect more than one. Clone this plane several times until you have a cluster of particles that looks close to the real thing.

One optional technique you may wish to use (depending on the shape and style you are hoping to achieve) is to create a three-quad base mesh and clone it rather than use each plane individually. As a single quad viewed from certain angles will have no thickness, a great technique is to use three quads, each intersecting at perpendicular angles, so that there is one facing each axis (x, y, and z). In other words, make little "stars" that can be viewed from any camera angle.

Eye-Candy Aplenty!

Group those together and start cloning this shape many times for a truly great looking particle cloud. Scatter it about randomly—perhaps more closely packed together in the center of the cluster, with smaller variants near the edges.



When you are happy with the look of your explosion, export it as a triangulated .OBJ mesh.

Time for action – sculpting the second keyframe

Now start working on the second frame of your particle animation. This is a "keyframe" that will be used during the interpolated animation in your game. Nudge each particle slightly upward, scale each one a little larger, and rotate it a tiny bit. There are macros that can do this randomly for multiple objects at the same time. Save this as a second .OBJ file, making sure that your two meshes have the exact same number of polies, each in the same order.

What just happened?

In the preceding steps, we briefly outlined a way to create two meshes for use as the before and after state for our particle system. The `Stage3dParticle` class takes these two meshes and morphs the first into the second over time by using a custom shader.

Repeat this process with all the special effects you wish to render in your game. You can really have a lot of fun with this step. Use your imagination and come up with other interesting shapes. For example, you might sculpt a circular or ring-shaped shock wave. Another frequently used particle shape is a small cluster of "shooting stars"—a cluster of particles that points in a certain direction like some sort of directional spray. This mesh could be used for splashes of water, lava, blood, oil, or sparks. Conical shapes with clusters of particles pointing in one direction spreading outward are also perfect for engine glows, blowtorches, and muzzle-flashes from guns.

As this book is about the programming, we won't go into the art any further, but this step is a great place to go crazy and let your imagination run wild. Stage3D is so fast that you don't really have to worry about how many polygons you are using. That said, the fewer you use the better the framerate, so try to create meshes in the under 1,000 polygon range. You can often create beautiful clusters of particles using far fewer than that.

Incorporating the particle system class in our game

The final step in our quest to deliver gratuitous amounts of glorious eye-candy is to get it up and running inside our game. This step is very simple and will only require a few extra lines of code and a couple changes here and there.

Time for action – adding particles to your game

Start with the example project from the previous chapter and open your `Stage3dGame.as` file. Import our new classes by adding the following lines at the top of your file near all the other import statements:

```
import Stage3dParticle;
import GameParticlesystem;
```

Next, add a few more variables inside the class definition, just below the Vector lists of enemies, bullets, and props:

```
// used for our particle system demo
private var nextShootTime:uint = 0;
private var shootDelay:uint = 0;
private var explo:Stage3dParticle;
private var particleSystem:GameParticlesystem;
private var scenePolycount:uint = 0;
```

Now embed the new meshes you have created along with any new textures in the same fashion as is already used for our spaceship, asteroids, and sky models:

```
// explosion start - 336 polygons
[Embed (source = "explosion1.obj",
        mimeType = "application/octet-stream")]
private var explosion1Data:Class;

// explosion end - 336 polygons
[Embed (source = "explosion2.obj",
        mimeType = "application/octet-stream")]
private var explosion2Data:Class;
```

Do the same for any other particle meshes you have designed.

Make sure to embed all new textures you created in the same way as the others, using either [embed] or the Flash IDE library. See the previous chapter source for examples.

Time for action – preparing a type of particle for use

Now that your new art content has been embedded, ensure that it is parsed—again in the same way as the existing models and textures in our game. Specifically, add any new textures to your `onContext3DCreate()` function by copy-n-pasting the working code used by our other textures.

```
particle1texture = context3D.createTexture(
    particle1bitmap.width, particle1bitmap.height,
    Context3DTextureFormat.BGRA, false);
uploadTextureWithMipmaps(
    particle1texture, particle1bitmap.bitmapData);
```

In order to define this new particle type, add the following lines to the very bottom of your `initData()` function:

```
// create a particle system
particleSystem = new GameParticlesystem();

// define an explosion particle type
particleSystem.defineParticle("explosion",
    new Stage3dParticle(explosion1Data, context3D,
    particle1texture, explosion2Data));
```

What just happened?

The preceding code first creates a new `GameParticlesystem` object that will, from now on, handle all the special effects in our game. We then create a new `Stage3dParticle` object by sending both frames of mesh data along with an already initialized texture and `Context3D`. This is the "master clone source" version that will be duplicated on demand by our particle system. We give this type of particle a descriptive name ("explosion") and instruct it what to clone each time we want to spawn a new particle.

In the example game from this chapter, there are five different kinds of particle defined in the same way as illustrated earlier. For simplicity, it is left up to the reader to implement different kinds of particle. Use your imagination!

Time for action – upgrading the renderScene function

We need to insert code in our render loop to draw all the particles. As an added bonus, let's keep track of the number of polygons being rendered in each scene. A few additional lines are required to add them together. This is what your final `renderScene` function should look like:

```
private function renderScene():void
{
    scenePolycount = 0;

    viewmatrix.identity();
    // look at the player
    viewmatrix.append(chaseCamera.transform);
    viewmatrix.invert();
    // tilt down a little
    viewmatrix.appendRotation(15, Vector3D.X_AXIS);
    // if mouselook is on:
    viewmatrix.appendRotation(gameinput.cameraAngleX,
        Vector3D.X_AXIS);
    viewmatrix.appendRotation(gameinput.cameraAngleY,
        Vector3D.Y_AXIS);
    viewmatrix.appendRotation(gameinput.cameraAngleZ,
        Vector3D.Z_AXIS);

    // render the player mesh from the current camera angle
    player.render(viewmatrix, projectionmatrix);
    scenePolycount += player.polycount;

    // loop through all known entities and render them
    for each (entity in props)
```

```
{  
    entity.render(viewmatrix, projectionmatrix);  
    scenePolycount += entity.polycount;  
}  
  
particleSystem.render(viewmatrix, projectionmatrix);  
scenePolycount += particleSystem.totalpolycount;  
}
```

What just happened?

The only changes since the last time are the addition of the polycount stats and the `particleSystem.render` call. Remember that the `particleSystem.render()` function will loop through each and every particle and draw it for us. We simply pass the view and projection matrix so the camera angles are respected and let it do its work.

Time for action – adding particles to the gameStep function

Now that we have set things up, let's experiment with particles, so that when you hold down the spacebar, more and more particles appear on the screen. Edit the `gameStep()` function such that a new particle is spawned in each frame when the "fire" input is pressed. Additionally, we need to instruct the particle system to update each frame based on the elapsed time. Here is the new version of this function:

```
private function gameStep(frameMs:uint):void  
{  
    // handle player input  
    var moveAmount:Number = moveSpeed * frameMs;  
    if (gameinput.pressing.up) player.z -= moveAmount;  
    if (gameinput.pressing.down) player.z += moveAmount;  
    if (gameinput.pressing.left) player.x -= moveAmount;  
    if (gameinput.pressing.right) player.x += moveAmount;  
    if (gameinput.pressing.fire)  
    {  
        if (gametimer.gameElapsedTime >= nextShootTime)  
        {  
            //trace("Fire!");  
            nextShootTime =  
                gametimer.gameElapsedTime + shootDelay;  
            // random location somewhere ahead of player  
            var groundzero:Matrix3D = new Matrix3D;  
            groundzero.prependTranslation(  
                player.x + Math.random() * 200 - 100,  
                player.y + Math.random() * 100 - 50,
```

```

        player.z + Math.random() * -1000 - 250);
        // create a new particle (or reuse an inactive one)
        particleSystem.spawn("explosion", groundzero, 2000);
    }
}
// follow the player
chaseCamera.x = player.x;
chaseCamera.y = player.y + 1.5; // above
chaseCamera.z = player.z + 3; // behind
// animate the asteroids
asteroids1.rotationDegreesX += asteroidRotationSpeed * frameMs;
asteroids2.rotationDegreesX -= asteroidRotationSpeed * frameMs;
asteroids3.rotationDegreesX += asteroidRotationSpeed * frameMs;
asteroids4.rotationDegreesX -= asteroidRotationSpeed * frameMs;
// animate the engine glow - spin fast and pulsate slowly
engineGlow.rotationDegreesZ += 10 * frameMs;
engineGlow.scaleXYZ =
    Math.cos(gametimer.gameElapsedTime / 66) / 20 + 0.5;
// advance all particles based on time
particleSystem.step(frameMs);
}

```

What just happened?

The preceding code steps the particle system (by passing the number of milliseconds that have elapsed since the previous frame) and also spawns a new explosion at a random location that is set to last for 2,000 milliseconds (or two seconds) if the spacebar is pressed. As this function is called every frame, if you hold down the spacebar a massive number of particles will be spawned.

Time for action – keeping track of particle statistics

As we are getting to the point that screenshots and boastful stats become fun to share with friends, a little extra information can go a long way to prove the efficiency of our particle routines.

Firstly, in our heartbeat function (which is only run every few seconds and at the moment is only used for debug purposes), let's report some more statistics. We do this as follows:

```

// for efficiency, this function only runs occasionally
// ideal for calculations that don't need to be run every frame
private function heartbeat():void
{
    trace('heartbeat at ' + gametimer.gameElapsedTime + 'ms');
}

```

Eye-Candy Aplenty!

```
trace('player ' + player.posString());
trace('camera ' + chaseCamera.posString());
trace('particles active: ' + particleSystem.particlesActive);
trace('particles total: ' + particleSystem.particlesCreated);
trace('particles polies: ' + particleSystem.totalpolycount);
}
```

Finally, along with the framerate, we will report how many polies are being rendered so our screenshots can have something to boast about. Edit the one line in our `enterFrame` function that updates the FPS text and add the `scenePolycount` as follows:

```
fpsTf.text = fps.toFixed(1) +
    " fps (" + scenePolycount + " polies);
```

What just happened?

We upgraded our game to keep track of the number of polygons being rendered in each frame. This will help during testing to give benchmarks over time in the debug log (to determine if the game is becoming too filled with enemies or particles after long play sessions) and so people can compare the framerate with the number of polies on the screen at any given moment.

As many popular particle systems currently get bogged down with only a few thousand particles (because they are simulating on the CPU and render each particle in a separate draw call), this "batched GPU simulated" particle system is sure to impress your friends with its incredible performance. You can expect to be able to animate 50,000+ particles per frame and still maintain a smooth 60fps on decent gaming rigs.

That is it! Our game can now pump out massive numbers of particles of various kinds. The preceding example code only adds the "explosion" particle type, but in your own game you can do the same for several types of particle.

Let's see the new particle system in action!

You can view the finished *Chapter 8* demo here:

http://www.mcfunkypants.com/molehill/chapter_8_demo/

In addition, the full source code is available at the following URL:

<http://www.packtpub.com/adobe-flash11-stage3d-molehill-game-programming-beginners-guide/book>



Now that we have implemented a fancy new particle entity class, AGAL shader code to do all particle animation on the GPU for speed and a particle system that maintains a pool of reusable particles, our game is capable of some incredible eye-candy. As we are batching geometry and rendering hundreds or even thousands of particles in a single `drawTriangles` call, our game can easily render tens of thousands of particles on the screen without even breaking a sweat.

Rendering GPU particles using the batched, reusable, hyper-optimized technique outlined in this chapter frees up your CPU for other things. This means that your game has lots of horsepower left to render other kinds of meshes in your level, as well as the CPU being available to process inputs, trigger sounds, perform AI and collision detection, and more without having to do any of the heavy lifting required for effects.

Pop quiz – earn some experience points

Here are a few quick questions to test your understanding. If you can answer them, you are officially ready to move on to the next step in your grand adventure:

1. Why could we not just render thousands of individual particles separately instead of in large batches?
 - a. Because Stage3D can only display low-polygon scenes
 - b. Because each draw call has overhead and too many will result in poor framerate
 - c. Because we would have to write a unique shader for each one
 - d. Because it would be too boring for our computer
2. Why did we create two meshes for our explosion instead of just one?
 - a. As a means to pre-calculate the trajectories of numerous polygons
 - b. So that we don't have to use the CPU to do any simulation work
 - c. Because we use the second to smoothly animate particles in an AGAL shader
 - d. All of the above
3. What technique do we use to increase the framerate and to avoid memory leaks and GC (garbage collection) of temporary variables during our render loop?
 - a. We made all our variables private
 - b. We assumed all users have infinite RAM
 - c. We made a "pool" of reusable particles that can be respawned over and over
 - d. We compiled our source code very carefully

Have a go hero –inventing some cool particle effects

Your side quest this time is to design a bunch of cool-looking particle effects for your game. Instead of just a simple expanding explosion, why not add a swirling tornado (made up of many dust motes or hazy cloud sprites)? Alternatively, create a texture made up of small blue droplets and design a "splash" appropriate for the front of a sailing ship on the ocean. Perhaps your game will be a shooter and you want gratuitous gore. If so, make blood splatters that emerge from gunshot wounds along with glowing muzzle flashes coming from your guns.

Particles can also be used in other less obvious ways. For example, in a detective game, faint footprints in the sand could be spawned every few feet of movement by players. Debris such as litter and paper in a ghetto blowing in the wind can add atmosphere to a game. In an underwater game, small specks of organic material floating in the water can give your movement more depth. In a dating simulator or visual novel, perhaps a character is so enamored with another that little red hearts float upward from their eyes as they fall madly in love.

Summary

In this chapter, we managed to achieve the following milestones:

- ◆ We wrote an AGAL shader that smoothly animates and fades out particles over time
- ◆ We designed a basic particle class with two-frame keyframed mesh data
- ◆ We created a particle system that manages and reuses multiple types of particle
- ◆ We upgraded our game to show off these new special effects

Level 8 achieved!

Congratulations. You are nearly at the end of your epic journey. Though our game is still very simple, we have achieved a major milestone along the way towards the creation of a high-poly, next-gen, amazing-looking 3D game in Flash. On the horizon are all the final changes that make a video game. These last pieces in the puzzle represent the exciting conclusion to our grand adventure.

There is one last quest to complete. The tying up of loose ends: the polish. This is the challenge that awaits us in the next two chapters. The treasure that we get as a reward, after all our hard work is that we will soon be able to play a complete 3D game of our own design.

9

A World Filled with Action

Time to design the world and fill it with lethal enemies!

The second last step in our adventure is to create some game-specific classes that will flesh out the world and allow for easy level creation. We also need to be able to spawn entities that move and react to the player and know when they collide.

In order to make a finished, polished, and playable 3D game, we need more than simple rendering functionality. We will program a way to create a complex game world by drawing levels in an image editor rather than having to manually enter them in the source code. While we are at it, we will design a way to control enemies and make them move around in space by implementing simple artificial intelligence. We want the game to detect when things are bumping into each other so we can trigger explosions or grant the player more points, so this is the ideal time to implement collision detection. When we are done, our game will be filled with enemies that shoot at the player and level props that the player can crash into.

In order to successfully upgrade our game engine to be able to provide all this new action, we need to do the following:

- ◆ Extend the entity class with game "actor" functionality
- ◆ Program collision detection
- ◆ Implement an "actor reuse pool" system
- ◆ Optimize our game to only display actors that are nearby
- ◆ Program a map parsing mechanism to allow for easy world creation
- ◆ Upgrade the input routines

Rendering and inits (the "low-level" functionality) are already taken care of. The preceding steps outline the basic order of operations we need to follow to implement the "high-level" building blocks for our game. Once we make it through this chapter, our game engine will be ready for the final polishing phase, where we finally get to play a "real" game that has a beginning, middle, and end.

Extending the entity class for "actors"

Our `Stage3dEntity` class is a general-purpose mesh-rendering class that is great for any kind of game entity. It is simple and fast, and contains all the useful helper functionality to allow us to move objects around and manipulate rotations, and so on.

Basing more advanced classes upon this class is a smart idea because it keeps the one master class from becoming too cluttered and eliminates the need for "bloated" classes that are catch-all do-anything classes. We will make a specialized class that extends `Stage3dEntity` that has some game-specific functionality perfect for things like bullets and spaceships.

Time for action – creating a game actor class

Create a new file in your source folder named `GameActor.as` and begin by extending the `Stage3dEntity` class with some fancy new properties as follows:

```
// Game actor class version 1.1
// an entity that can move, shoot, spawn particles
// trigger sounds and detect collisions
//
package
{

    import flash.geom.Matrix3D;
    import flash.geom.Vector3D;
    import flash.display3D.Context3D;
    import flash.display3D.Program3D;
    import flash.display3D.textures.Texture;
    import flash.media.Sound;

    public class GameActor extends Stage3dEntity
    {
        // game-related stats
        public var name:String = ''; // unique
        public var classname:String = ''; // not unique
```

```

public var owner:GameActor; // so you can't shoot yourself
public var touching:GameActor; // last collision detected
public var active:Boolean = true; // animate?
public var visible:Boolean = true; // render?
public var health:Number = 1; // when zero, die
public var damage:Number = 250; // inflicted on others
public var points:Number = 25; // score earned if destroyed
public var collides:Boolean = false; // check hits?
public var collidemode:uint = 0; // 0=sphere, 1=aabb
public var radius:Number = 1; // used for sphere collision
public var aabbMin:Vector3D = new Vector3D(1, 1, 1, 1);
public var aabbMax:Vector3D = new Vector3D(1, 1, 1, 1);

```

What just happened?

In the preceding code, we defined a few handy class properties. Some are references to other GameActors, such as the owner property which could be used, for example, to ensure that points earned upon destroying something are attributed to whoever shoots a particular bullet. The health property is used whenever a bullet hits us—the higher this number, the more times an actor will need to be shot before it explodes.

Time for action – extending the actor's properties

Let's add a few more properties to really flesh out our game actor class as follows:

```

// callback functions
public var runConstantly:Function;
public var runConstantlyDelay:uint = 1000;
public var runWhenNoHealth:Function;
public var runWhenMaxAge:Function;
public var runWhenCreated:Function;
// time-related vars
public var age:uint = 0;
public var ageMax:uint = 0;
public var stepCounter:uint = 0;
// animation vars in units per SECOND (1000ms)
public var posVelocity:Vector3D;
public var rotVelocity:Vector3D;
public var scaleVelocity:Vector3D;
public var tintVelocity:Vector3D;
// automatically shoot bullets when 'near' an enemy
public var bullets:GameActorpool;
public var shootName:String = '';
public var shootDelay:uint = 4000;

```

```
public var shootNext:uint = 0;
public var shootRandomDelay:Number = 2000;
public var shootDist:Number = 100;
public var shootAt:GameActor = null;
public var shootVelocity:Number = 50;
public var shootSound:Sound;
// spawnable particles
public var particles:GameParticlesystem;
public var spawnConstantly:String = '';
public var spawnConstantlyDelay:uint = 0;
public var spawnConstantlyNext:uint = 0;
public var spawnWhenNoHealth:String = '';
public var spawnWhenMaxAge:String = '';
public var spawnWhenCreated:String = '';
// sound effects
public var soundConstantlyDelay:uint = 1000;
public var soundConstantlyNext:uint = 0;
public var soundConstantly:Sound;
public var soundWhenNoHealth:Sound;
public var soundWhenMaxAge:Sound;
public var soundWhenCreated:Sound;
```

What just happened?

These properties are really handy to bring depth and gameplay to your game. For example, you could set up a sound to be played whenever you destroy a spaceship, or define a particle that you want the entity to produce every few milliseconds, such as a puff of smoke coming from a damaged ship.

Time for action – coding the GameActor class constructor

Let's move on to set up the class and all the functionality promised by these new properties.

```
public function GameActor(mydata:Class = null,
    mycontext:Context3D = null,
    myshader:Program3D = null,
    mytexture:Texture = null,
    modelscale:Number = 1,
    flipAxis:Boolean = true,
    flipTexture: Boolean = true)
{
    super(mydata, mycontext, myshader, mytexture,
        modelscale, flipAxis, flipTexture);
}
```

What just happened?

The class constructor in the preceding code simply runs the original version of the constructor that is defined in our old `Stage3dEntity` class. This is done by calling "super" which refers to the class constructor of the class we have extended (`Stage3dEntity`).

As always, note the `flipAxis` and `flipTexture` parameters, which instruct the `OBJ` file parser whether or not the data in your model file has been flipped around, which tends to only happen with older file exporters such as the one used by 3D Studio Max 9. Newer versions generally do not need these to be set to TRUE, so if you have problems with your models seeming "inside out", try different values for these and your problem will go away.

Time for action – creating a step animation function

Next, we can create a `step` function that is run every frame in our render loop as follows:

```
public function step(ms:uint):void
{
    if (!active) return;

    age += ms;
    stepCounter++;

    if (health <= 0)
    {
        //trace(name + " out of health.");
        if (particles && spawnWhenNoHealth)
        {
            trace(name + " exploding into " + spawnWhenNoHealth);
            var spawnxform:Matrix3D = new Matrix3D();
            spawnxform.position = position.clone();
            //particles.spawn(spawnWhenNoHealth,transform);
            particles.spawn(spawnWhenNoHealth,spawnxform,5555,0,10);
        }
        if (soundWhenNoHealth)
            soundWhenNoHealth.play();
        if (runWhenNoHealth != null)
            runWhenNoHealth();
        die();
        return;
    }

    if ((ageMax != 0) && (age >= ageMax))
    {
        //trace(name + " old age.");
    }
}
```

```
if (particles && spawnWhenMaxAge)
    particles.spawn(spawnWhenMaxAge, transform);
if (soundWhenMaxAge)
    soundWhenMaxAge.play();
if (runWhenMaxAge != null)
    runWhenMaxAge();
die();
return;
}
```

What just happened?

The preceding code simply updates the age of the actor and performs anything required of it if a maximum age has been set. This is handy to ensure that bullets, for example, don't fly onwards forever if they never hit anything. If the actor has died (run out of health), then we optionally spawn a particle system and run a callback function if it has been set. This callback function could be used to trigger a sound, give the player some points, or signal the level to end.

Time for action – animating actors

Next, we handle the basic animation that can be set when we create a new actor. This could include a constant movement, so that it flies forward, or you could set up an asteroid to slowly rotate in each frame or a bubble to increase in size over time. Add the following code to your `step()` function:

```
if (posVelocity)
{
    x += posVelocity.x * (ms / 1000);
    y += posVelocity.y * (ms / 1000);
    z += posVelocity.z * (ms / 1000);
}

if (rotVelocity)
{
    rotationDegreesX += rotVelocity.x * (ms / 1000);
    rotationDegreesY += rotVelocity.y * (ms / 1000);
    rotationDegreesZ += rotVelocity.z * (ms / 1000);
}

if (scaleVelocity)
{
    scaleX += scaleVelocity.x * (ms / 1000);
    scaleY += scaleVelocity.y * (ms / 1000);
    scaleZ += scaleVelocity.z * (ms / 1000);
}
```

Implementing artificial intelligence (AI)

All games have some form of artificial intelligence. You may need enemies to move toward the player, avoid obstacles, or even plot complex strategies. AI is a subject deep enough to warrant an entire book series, and vast resources are available online to take you further down the path of designing autonomous actors in your game.

For our purposes, we are going to define extremely simplistic behaviors for the enemies in the game. We need to give our game entities a mechanism to keep track of time passing, so that their actions can be spaced apart appropriately. Simple "perception" in the form of locational awareness for our bad guys will allow them to decide when it is a good time to shoot at the player.

In order to keep things simple, nothing more than simplistic timers and shooting actions are going to be programmed here. In your game, you may want to give your enemies tactical prowess. Try googling various AI terms such as "a-star pathfinding" and "game AI" to reveal a wealth of fascinating algorithms for programming the behavior of your in-game actors.

Time for action – using timers

The first AI technique we are going to implement is the ability to trigger events to occur after a set period of time. For example, we might want a puff of smoke to appear a few times per second, or we might want an enemy to fire a bullet after a randomly varying amount of time. To do so, we simply record a timestamp of when the last event occurred and compare that to the actor's age to see if it is time to do it again. Continuing with the `step()` function, add the following code:

```
// maybe spawn a new particle
if (visible && particles && spawnConstantlyDelay > 0)
{
    if (spawnConstantly != '')
    {
        if (age >= spawnConstantlyNext)
        {
            //trace("actor spawn " + spawnConstantly);
            spawnConstantlyNext =
                age + spawnConstantlyDelay;
            particles.spawn(spawnConstantly, transform);
        }
    }
}

// maybe trigger a sound
if (visible && soundConstantlyDelay > 0)
```

```
{  
    if (soundConstantly)  
    {  
        if (age >= soundConstantlyNext)  
        {  
            soundConstantlyNext =  
                age + soundConstantlyDelay;  
            soundConstantly.play();  
        }  
    }  
}
```

Time for action – shooting at enemies

The "shoot" AI is the same as the above, but with a little extra complexity: actors will only fire at enemies that are within a certain ranged distance if an enemy has been defined. Continue adding to the `step()` function as follows:

```
// maybe "shoot" (spawn an actor)  
if (visible && bullets && (shootName != ''))  
{  
    var shouldShoot:Boolean = false;  
    if (age >= shootNext)  
    {  
        shootNext = age + shootDelay +  
            (Math.random() * shootRandomDelay);  
  
        if (shootDist < 0)  
            shouldShoot = true;  
        else if (shootAt &&  
            (shootDist > 0) &&  
            (Vector3D.distance(  
                position, shootAt.position) <= shootDist))  
        {  
            shouldShoot = true;  
        }  
  
        if (shouldShoot)  
        {  
            var b:GameActor =  
                bullets.spawn(shootName, transform);  
  
            // remember who this bullet belongs to  
            b.owner = this;  
        }  
    }  
}
```

```
// aim towards an enemy?  
if (shootAt)  
{  
    b.transform.pointAt(  
        shootAt.transform.position);  
    b.rotationDegreesY -= 90;  
  
    b.posVelocity =  
        b.transform.position.subtract(  
            shootAt.transform.position);  
    b.posVelocity.normalize();  
    b.posVelocity.negate();  
    b.posVelocity.scaleBy(shootVelocity);  
}  
// otherwise we simply fire in whatever  
// direction was given during spawn  
// when the game set posVelocity  
  
if (shootSound)  
    shootSound.play();  
}  
}  
}  
}  
} // end of the step function
```

What just happened?

That is it for our fancy new GameActor step function. As you can see, this adds a wealth of gameplay variety to your bag of tricks. We just implemented actor animation, timed events such as playing sounds or emitting a particle effect, and AI behavior for shooting at enemies when they are nearby.

This function could very easily be extended to react differently depending on what kind of objects are nearby. For example, imagine a game with a cowardly enemy that retreats if faced with too powerful an opponent, always moves toward nearby powerups, changes direction randomly every few seconds, or one that drops time bombs (or "footprints") at set intervals.

Time for action – cloning an actor

The next function is very simple, but handy: the clone function. We want to be able to clone a source definition of an actor type (so we can instruct the game engine what an "evil goblin" is and then spawn multiple copies simply) just like we did for particles.

We don't want to copy every single property, so we don't want to iterate in a loop through every single variable defined in our class. We also need to clone Matrix3Ds rather than share references between clones, so that moving one actor won't unintentionally make another move as well. Therefore, we need to manually copy data as needed.

```
public function cloneactor():GameActor
{
    var myclone:GameActor = new GameActor();
    updateTransformFromValues();
    myclone.transform = this.transform.clone();
    myclone.updateValuesFromTransform();
    myclone.mesh = this.mesh;
    myclone.texture = this.texture;
    myclone.shader = this.shader;
    myclone.vertexBuffer = this.vertexBuffer;
    myclone.indexBuffer = this.indexBuffer;
    myclone.context = this.context;
    myclone.polycount = this.polycount;
    myclone.blendSrc = this.blendSrc;
    myclone.blendDst = this.blendDst;
    myclone.cullingMode = this.cullingMode;
    myclone.depthTestMode = this.depthTestMode;
    myclone.depthTest = this.depthTest;
    myclone.depthDraw = this.depthDraw;
    // game-related stats
    myclone.name = this.name;
    myclone.classname = this.classname;
    myclone.owner = this.owner;
    myclone.active = this.active;
    myclone.visible = this.visible;
    myclone.health = this.health;
    myclone.damage = this.damage;
    myclone.points = this.points;
    myclone.collides = this.collides;
    myclone.collidemode = this.collidemode;
    myclone.radius = this.radius;
    myclone.aabbMin = this.aabbMin.clone();
    myclone.aabbMax = this.aabbMax.clone();
```

```
// callback functions
myclone.runConstantly = this.runConstantly;
myclone.runConstantlyDelay = this.runConstantlyDelay;
myclone.runWhenNoHealth = this.runWhenNoHealth;
myclone.runWhenMaxAge = this.runWhenMaxAge;
myclone.runWhenCreated = this.runWhenCreated;
// time-related vars
myclone.age = this.age;
myclone.ageMax = this.ageMax;
myclone.stepCounter = this.stepCounter;
// animation-related vars - per ms
myclone.posVelocity = this.posVelocity;
myclone.rotVelocity = this.rotVelocity;
myclone.scaleVelocity = this.scaleVelocity;
myclone.tintVelocity = this.tintVelocity;
// bullets
myclone.bullets = this.bullets;
myclone.shootName = this.shootName;
myclone.shootDelay = this.shootDelay;
myclone.shootNext = this.shootNext;
myclone.shootRandomDelay = this.shootRandomDelay;
myclone.shootDist = this.shootDist;
myclone.shootAt = this.shootAt;
myclone.shootVelocity = this.shootVelocity;
myclone.shootSound = this.shootSound;
// spawnable particles
myclone.particles = this.particles;
myclone.spawnConstantly = this.spawnConstantly;
myclone.spawnConstantlyDelay = this.spawnConstantlyDelay;
myclone.spawnConstantlyNext = this.spawnConstantlyNext;
myclone.spawnWhenNoHealth = this.spawnWhenNoHealth;
myclone.spawnWhenMaxAge = this.spawnWhenMaxAge;
myclone.spawnWhenCreated = this.spawnWhenCreated;
// sound effects
myclone.soundConstantlyDelay = this.soundConstantlyDelay;
myclone.soundConstantlyNext = this.soundConstantlyNext;
myclone.soundConstantly = this.soundConstantly;
myclone.soundWhenNoHealth = this.soundWhenNoHealth;
myclone.soundWhenMaxAge = this.soundWhenMaxAge;
myclone.soundWhenCreated = this.soundWhenCreated;
myclone.active = true;
myclone.visible = true;
return myclone;
}
```

Time for action – handling an actor's death

The `die` function is extremely simplistic and is called when the actor runs out of health or dies of "old age". As we will eventually be creating a "pool" of actors that can be reused during the game, all we need to do here is turn the actor "off", so that in the future the game can reuse it. In this way, as the player destroys enemy ships, for example, future enemies can come from this reusable pool without any further inits or memory consumption.

```
public function die():void
{
    //trace(name + " dies!");
    active = false;
    visible = false;
}
```

Time for action – respawning an actor

The `respawn` function is very similar to the one we programmed for our particle class. It resets a few key properties and makes a previously "dead" or an inactive actor come back to life, ready to be reused. As mentioned earlier, this is how the actor pool will be able to reuse a "dead" actor by bringing it back to "life":

```
public function respawn(pos:Matrix3D = null):void
{
    age = 0;
    stepCounter = 0;
    active = true;
    visible = true;

    // don't shoot immediately
    shootNext = Math.random() * shootRandomDelay;

    if (pos)
    {
        transform = pos.clone();
    }

    if (soundWhenCreated)
        soundWhenCreated.play();

    if (runWhenCreated != null)
        runWhenCreated();
}
```

```

        if (particles && spawnWhenCreated)
            particles.spawn(spawnWhenCreated, transform);

        //trace("Respawned " + name + " at " + posString());
    }
}

```

Collision detection

The final functionality that we need to implement in our actor class is the one thing that will add interactivity to your game like no other: collisions! The ability to detect whether one actor is touching another is a key to being able to simulate bullets, for example. Collision detection is a subject broad enough to warrant its own book, so for now we will program the simplest of collision detection routines.

For advanced collision detection and response (bouncing, friction, colliding with complex shapes), you would be better served by using a real physics engine, which is beyond the scope of this book and is left as an exercise for the reader.

Time for action – detecting collisions

We will create our basic, simplistic collision detection routines now.

```

// used for collision callback performed in GameActorpool
public function colliding(checkme:GameActor):GameActor
{
    if (collidemode == 0)
    {
        if (isCollidingSphere(checkme))
            return checkme;
        else
            return null;
    }
    else
    {
        if (isCollidingAabb(checkme))
            return checkme;
        else
            return null;
    }
}

```

Time for action – detecting sphere-to-sphere collisions

Our simple game does not need to do anything more other than to detect how close one actor is to another based on their transform position and their radius. This is called **sphere-to-sphere** collision detection. It is fast, easy to understand, and is perfect for the shoot-em-up example game we have been creating.

```
// simple sphere to sphere collision
public function isCollidingSphere(checkme:GameActor):Boolean
{
    // never collide with yourself
    if (this == checkme) return false;
    // only check if these shapes are collidable
    if (!collides || !checkme.collides) return false;
    // don't check your own bullets
    if (checkme.owner == this) return false;
    // don't check if no radius
    if (radius == 0 || checkme.radius == 0) return false;

    var dist:Number =
        Vector3D.distance(position, checkme.position);

    if (dist <= (radius+checkme.radius))
    {
        // trace("Collision detected at distance="+dist);
        touching = checkme; // remember who hit us
        return true;
    }

    // default: too far away
    // trace("No collision. Dist = "+dist);
    return false;
}
```

As you can see, the preceding function simply checks to see if the actors are set to bother checking for collisions, and if they are, a measurement is made and the result is returned.

Time for action – detecting bounding-box collisions

For completeness, although our example game won't be using it, we should implement box-shaped collisions as well. The fastest and simplest way to do this is called **aabb**, which stands for **axis-aligned-bounding-boxes**. The term axis-aligned means that the rectangle defined by the min and max coordinates never rotates: it is stuck on "the grid" and is used to quickly determine if one box is touching another.

```

// axis-aligned bounding box collision detection
// not used in the example game but here for convenience
private function aabbCollision(
    min1:Vector3D, max1:Vector3D,
    min2:Vector3D, max2:Vector3D ):Boolean
{
    if ( min1.x > max2.x ||
        min1.y > max2.y ||
        min1.z > max2.z ||
        max1.x < min2.x ||
        max1.y < min2.y ||
        max1.z < min2.z )
    {
        return false;
    }
    return true;
}

public function isCollidingAabb(checkme:GameActor):Boolean
{
    // never collide with yourself
    if (this == checkme) return false;
    // only check if these shapes are collidable
    if (!collides || !checkme.collides) return false;
    // don't check your own bullets
    if (checkme.owner == this) return false;
    // don't check if no aabb data
    if (aabbMin == null ||
        aabbMax == null ||
        checkme.aabbMin == null ||
        checkme.aabbMax == null)
        return false;

    if (aabbCollision(
        position + aabbMin,
        position + aabbMax,
        checkme.position + checkme.aabbMin,
        checkme.position + checkme.aabbMax))
    {
        touching = checkme; // remember who hit us
        return true;
    }

    // trace("No collision.");
}

```

```
        return false;  
    }  
  
} // end package  
  
} // end class
```

That is it for our feature-rich, vaguely intelligent, eminently reusable `GameActor` class. By adding all this new functionality to a class that is derived from the `Stage3dEntity` class, we have kept separate the rendering and matrix transform code from the higher-level gameplay-specific logic.

An "actor reuse pool" system

In the same way that we created a reusable "pool" of particles that only created new entities on demand and reuses old ones when available, we are going to do the same for our actor class. By doing things this way, you will be able to efficiently populate your game world with thousands of objects while keeping your RAM levels near to constant.

This improves framerate, keeps code cleaner, and allows you to define a "parent" actor that is cloned as needed multiple times. This class will also be able to run routines over an entire set of actors at the same time. For example, in our game's render loop we might want to `step()` each of our enemy spaceships, so that they fly towards the player.

Time for action – creating an actor pool

Create a new file in your source folder named `GameActorpool.as` and begin by defining the class properties and constructor function as follows:

```
// Game actor pool version 1.2  
// creates a pool of actor entities on demand  
// and reuses inactive ones them whenever possible  
//  
package  
{  
    import flash.utils.Dictionary;  
    import flash.geom.Matrix3D;  
    import flash.geom.Vector3D;  
    import GameActor;  
  
    public class GameActorpool  
    {  
        // list of string names of each known kind  
        private var allNames:Vector.<String>;
```

```

// contains one source actor for each kind
private var allKinds:Dictionary;
// contains many cloned actors of various kinds
private var allActors:Dictionary;
// temporary variables - used often
private var actor:GameActor;
private var actorList:Vector.<GameActor>;
// used only for stats
public var actorsCreated:uint = 0;
public var actorsActive:uint = 0;
public var totalpolycount:uint = 0;
public var totalrendered:uint = 0;
// so we can "pause" any stepping for actors
// as a group for efficiency
public var active:Boolean = true;
// so we can hide parts of the map if required
// perfect for rooms/portals/pvs/lod
public var visible:Boolean = true;

// class constructor
public function GameActorpool()
{
    trace("Actor pool created.");
    allKinds = new Dictionary();
    allActors = new Dictionary();
    allNames = new Vector.<String>();
}

```

The preceding properties define things like the list of actors and actor "types", as well as a means to track statistics or to turn the entire pool on and off in one call in the same way that you can make a single actor invisible or inactive. In the class constructor function, we simply initialize the lists that we will need later.

Time for action – defining a clone parent

The first important function to implement is a mechanism to define a reusable "clone source" of a particular kind of actor. For example, you might set up a "bullet" or "rock" actor, give it an appropriate name, and then use it later on as the base parent to copy future clones aplenty.

```

// names a particular kind of actor
public function defineActor(
    name:String, cloneSource:GameActor):void
{

```

```
    trace("New actor type defined: " + name);
    allKinds[name] = cloneSource;
    allNames.push(name);
}
```

Time for action – animating the entire actor pool

Next, we can implement the GameActorpool's `step()` function that our game engine will call every frame. It will iterate through all known actors and animate them one by one.

```
public function step(ms:uint,
                     collisionDetection:Function = null,
                     collisionReaction:Function = null):void
{
    // do nothing if entire pool is inactive (paused)
    if (!active) return;

    actorsActive = 0;
    for each (actorList in allActors)
    {
        for each (actor in actorList)
        {
            if (actor.active)
            {
                actorsActive++;
                actor.step(ms);

                if (actor.collides &&
                    (collisionDetection != null))
                {
                    actor.touching = collisionDetection(actor);
                    if (actor.touching &&
                        (collisionReaction != null))
                        collisionReaction(actor, actor.touching);
                }
            }
        }
    }
}
```

What just happened?

To animate an entire pool of actors in one go, our game can use the `step()` function shown above. This function accepts a parameter that is the amount of time that passed since the previous frame, so that actors move the proper distance, and so on.

We also accept optional parameters for collision detection function callbacks. If they are set, additional calls to each actor's collision routine is run, and the callback function is run if they are touching. This callback is dependent upon your game, and could be an "explode" or "goal reached" trigger function, for example.

Time for action – rendering an actor pool

One of the simpler functions in our new `GameActorpool` class is the one responsible for drawing all visible actors. It simply iterates through the list of known actors and draws those that should be drawn. We also track some stats for future bragging rights.

```
// renders all active actors
public function render(view:Matrix3D,projection:Matrix3D):void
{
    // do nothing if entire pool is invisible
    if (!visible) return;

    totalpolycount = 0;
    totalrendered = 0;
    var stateChange:Boolean = true;
    for each (actorList in allActors)
    {
        stateChange = true; // v2
        for each (actor in actorList)
        {
            if (actor.active && actor.visible)
            {
                totalpolycount += actor.polycount;
                totalrendered++;
                actor.render(view, projection, stateChange);
            }
        }
    }
}
```

Time for action – spawning an actor

The spawn function is very similar to the one we made for our particle system in the previous chapter. We search our list of "defined" parent clone source actors for the name specified in the function parameters and clone it if none are available or respawn an old ("dead") one if possible.

```
// either reuse an inactive actor or create a new one
// returns the actor that was spawned for further use
public function spawn(
    name:String, pos:Matrix3D = null):GameActor
{
    var spawned:GameActor = null;
    var reused:Boolean = false;
    if (allKinds[name])
    {
        if (allActors[name])
        {
            for each (actor in allActors[name])
            {
                if (!actor.active)
                {
                    //trace("A " + name + " was reused.");
                    actor.respawn(pos);
                    spawned = actor;
                    reused = true;
                    return spawned;
                }
            }
        }
    else
    {
        //trace("This is the first " + name + " actor.");
        allActors[name] = new Vector.<GameActor>();
    }
if (!reused) // no inactive ones were found
{
    actorsCreated++;
    //trace("Creating a new " + name);
    //trace("Total actors: " + actorsCreated);
    spawned = allKinds[name].cloneactor();
    spawned.classname = name;
    spawned.name = name + actorsCreated;
    spawned.respawn(pos);
    allActors[name].push(spawned);
}
```

```

        //trace("Total " + name + "s: "
        //+ allActors[name].length);
        return spawned;
    }
}
else
{
    trace("ERROR: unknown actor type: " + name);
}
return spawned;
}

```

Time for action – checking for collisions between actors

One of the most important functions required in our actor pool class is the one that checks for collisions between actors. It simply runs the proper collision routine when appropriate and is called by our game engine when needed.

```

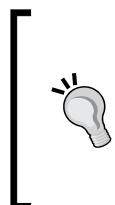
public function colliding(checkthis:GameActor) :GameActor
{
    if (!checkthis.visible) return null;
    if (!checkthis.active) return null;
    var hit:GameActor;
    var str:String;
    for each (str in allNames)
    for each (hit in allActors[str])
    {
        if (hit.visible &&
            hit.active &&
            checkthis.colliding(hit))
        {
            //trace(checkthis.name +
            // " is colliding with " + hit.name);
            return hit;
        }
    else
    {
        //trace(checkthis.name + " is NOT colliding with " +
        hit.name);
    }
}
return null;
}

```

Restricting display to nearby actors for better framerate

We will soon be dealing with massive levels filled with fantastic-looking environments and a huge number of bullets, particles, and enemies. All this content will slow down the game if we draw things that are so far away from the camera that they would only take up a single pixel of the screen space.

Even if a model is barely visible on-screen, all that processing of each and every vertex, running of vertex and fragment programs, as well as switching the render state to select the proper texture and blend modes takes a lot of time. If the model in question is really far away (or behind the camera), it makes sense to cull it from the scene temporarily and skip any rendering until it is closer.



This technique, while extremely simple, is a great way to permit levels comprised of well over a million polygons, as we are changing the "draw distance" of each actor, depending on their size, to minimize what gets rendered to include only the actors worth bothering with. Even a gigantic ten million poly game level will run at 60fps if you only draw the closest hundred thousand polies.

Time for action – hiding actors that are far away

All we need to do is iterate through our enemies, bullets and level props, and determine their distance from the camera by comparing their current location to that of the camera itself.

```
// optimize the scene: make actors that are far away
// or outside a bounding box invisible
public function hideDistant(pos:Vector3D,
    // scaled by radius:
    maxdist:Number = 500,
    // radius is added to these:
    maxz:Number = 0,
    minz:Number = 0,
    maxy:Number = 0,
    miny:Number = 0,
    maxx:Number = 0,
    minx:Number = 0
    ):void
{
    for each (actorList in allActors)
    {
        for each (actor in actorList)
```

What just happened?

We searched for any actors that are too far away to bother rendering. Entities that are too far away need to be "switched off", so that they are skipped during the render phase. As some actors are huge while others are small, we scale the distance threshold to draw larger actors from farther away, so that you notice less "pop-in".

Time for action – destroying every actor in the pool

The final function that we need is a convenience function that simply iterates through all known actors in a particular actor pool and "kills" them all.

```
// to "clear" the scene, "kill" all known entities
// good for in between new levels or after a gameover
public function destroyAll():void
{
    for each (actorList in allActors)
    {
        for each (actor in actorList)
        {
            // ready to be respawned
            actor.active = false;
            actor.visible = false;
        }
    }
}

} // end class

} // end package
```

What just happened?

The ability to quickly and easily turn off all actors in a pool is handy for switching between different levels. When the player reaches the end of "level one" for example, simply call this function to remove them all from the world. When you build "level two", any actors that have been marked as inactive will be reused and given new locations.

That is it for the `GameActorpool` class. This is a really handy and efficient way to create a game filled with tons of moving objects of various kinds. You might want to create several different actor pools in your game rather than stuffing a single one full of every possible type of game entity. In this way, you only need to check for collisions between actors of certain kinds, or perhaps you don't even need to run the `step()` function for certain non-interactive entities.

In any case, you are sure to be using this class a lot, as you create a complex and action-packed game. It is ready and waiting to be extended in many fun ways.

Easy world creation using a map image

All this "actor" functionality allows us to define types of entity by a simple string name and spawn multiple copies wherever we like. By designing our game engine in this way, we have opened up the possibility of creating massive levels that are built out of basic building blocks.

For example, you could sculpt a wall and then spawn copies all over your game levels in order to create a maze-like shape. You could sculpt one or two kinds of tree and then spawn dozens randomly to create a forest.

The question arises, however: how do we design these levels? There are many ways to do this. One would be to implement a level data file parsing algorithm of a common kind of level format, perhaps using XML or JSON or a specialized binary data format. Another would be to create an in-game level editor that allows creators to use the arrow keys to build levels. Finally, you could make levels "procedurally" by generating them using pseudo-random or fractal number generation routines. For example, by using the "perlin noise" functions built into Flash, you could create an infinite number of levels automatically.

These ideas and more are left as a side-quest for the particularly adventurous. The only problem with randomly created levels is that they don't always look great or play in a fun way. Mazes can have loops or dead ends, for example. Therefore, it is often a good idea to scatter certain entities (such as tufts of grass or pebbles) randomly, while maintaining direct authorial control over the macroscopic level designs.

A great way to allow you to design your own levels is to open up the possibility of simply drawing them. Like a blueprint, you could draw the locations of entities in your level using whatever art tool you currently use to design graphics.



The preceding screenshot is an example of a level that we will create—each pixel in the image on the left corresponds to the location in the game of an enemy or other mesh. The image on the right is called a "key image" which is simply a definition of the RGB colors values that correspond to a particular entity.

This allows us to choose any color we like to represent an entity.

Time for action – implementing a level parser class

We will take advantage of our handy `GameActorpool` class to spawn entities wherever pixels are found, that match the key image. To do so, all we need to do is write a routine that scans a small bitmap for pixel colors that match a list of entity names. Let's create this now. Make a brand new file in your source folder and name it `GameLevelParser.as`.

```
// Game Level Parser Class - version 1.1
// spawns meshes based on the pixels in an image
// uses a tiny key image to define what RGB = what mesh #
//
package
{
    import flash.display.*;
    import flash.geom.Matrix3D;
    import flash.geom.Vector3D;
    import flash.geom.Point;
    import GameActorpool;

    public class GameLevelParser
    {
        // create an array of the level for future use
        // to avoid having to look at the map pixels again
        // in our example game this is not used but
        // this would be handy for pathfinding AI for example
        public var leveldata:Array = [];

        public function GameLevelParser()
        {
            trace("Created a new level parser.");
        }
    }
}
```

What just happened?

The preceding code is very straightforward. We define our new class, import the functionality we need, and create a single class variable that holds the level data that was produced during the parsing of the map layout image.

Time for action – spawning actors based on a map image

In order to be able to spawn entities using more than one image (to stack them, for example, or to store different kinds of entity in different actor pools), we don't add any functionality within the class constructor above, but instead define a public function named `spawnActors()` that will do all the work for us.

```

public function spawnActors(
    keyImage:BitmapData,           // tiny image defining rgb
    mapImage:BitmapData,           // blueprint of the level
    thecast:Array,                // list of entity names
    pool:GameActorpool,           // where to spawn them
    offsetX:Number = 0,           // shift in X space
    offsetY:Number = 0,           // shift in X space
    offsetZ:Number = 0,           // shift in Z space
    tileW:int = 1,                // size in world units...
    tileH:int = 1,                // ...for each pixel
    trenchlike:Number = 0,         // U-shaped? #=direction:-1,0,+1
    spiral:Number = 0             // spiral shaped? #=num spirals
    ):Number // returns the "length" of the map
{
    trace("Spawning level entities...");
    trace("Actor List length = " + thecast.length);
    trace("keyImage is ",keyImage.width,"x",keyImage.height);
    trace("mapImage is ",mapImage.width,"x",mapImage.height);
    trace("Tile size is ", tileW, "x", tileH);
    trace("Total level size will be ",
        mapImage.width * tileW,
        "x", mapImage.height * tileH);

    var pos:Matrix3D = new Matrix3D();
    var mapPixel:uint;
    var keyPixel:uint;
    var whichtile:int;
    var ang:Number;
    var degreesToRadians:Number = Math.PI / 180;

```

What just happened?

To begin our function, we define some parameters that will allow us to do interesting things with our levels. Firstly, we make this function take two `BitmapData`s, one being the map itself and the other is the key image as described earlier. Secondly, we want an array of string names that correspond to the key image so we can determine which pixel color corresponds to what entity type.

The first pixel in the key image should always be the "blank" color (the RGB value that does not result in anything being spawned). Subsequent pixels in the key image map directly to index positions in our cast array. In this way, each map pixel RGB that is found in the key image results in the actor pool spawning the appropriate actor in the correct position.

We need a reference to an actor pool to spawn entities into, so we request one of the pool parameters that will be the recipient of our many `spawn()` requests. We might want to offset all the positions away from (0,0,0), so we need to be able to define them here with the offset parameters. The `tileW` and `tileH` parameters allow us to define the spacing of each element. Higher values will result in entities being spawned farther apart. These values should match the size in game units that you want each pixel in your map image to correspond to.

Finally, we define two optional parameters, `trenchlike` and `spiral`. These are added just for fun. If `trenchlike` is not zero, then the level will be "curled" to form a U-shape rather than a simple flat plane. If `spiral` is not zero, then it will "twist" the level like a ribbon as many times as specified. This is used to make really cool looking enemy spaceship formations.

Time for action – parsing the map image pixels

The next step is to read each pixel in the map image and check to see if that RGB value exists in the key image. If it does, an entity is spawned in the appropriate location.

```
// read all the pixels in the map image
// and place entities in the level
// in the correponding locations
for (var y:int = 0; y < mapImage.height; y++)
{
    levedata[y] = [];

    for (var x:int = 0; x < mapImage.width; x++)
    {
        mapPixel = mapImage.getPixel(x, y);

        for (var keyY:int = 0; keyY < keyImage.height; keyY++)
        {
            for (var keyX:int = 0; keyX < keyImage.width; keyX++)
            {
                keyPixel = keyImage.getPixel(keyX, keyY);
                if (mapPixel == keyPixel)
                {
                    whichtile = keyY * keyImage.width + keyX;

                    if (whichtile != 0)
                    {
                        pos.identity();

                        // turn to face "backwards"
                        // facing towards the camera
```

```

pos.appendRotation(180, Vector3D.Y_AXIS);

pos.appendTranslation(
    (x * tileW),
    0,
    (y * tileH));

```

In this loop, we are iterating through each pixel and determining the proper position and rotation for our soon-to-be spawned entity.

```

if (trenchlike != 0)
{
    // trenchlike means U-shaped
    ang = x / mapImage.width * 360;
    pos.appendTranslation(
        0,
        trenchlike *
            Math.cos(ang * degreesToRadians)
            / Math.PI * mapImage.width * tileW,
        0);
}

if (spiral != 0)
{
    // spiral formation: like a corkscrew
    ang = (((y / mapImage.height * spiral) * 360) -
        180);
    pos.appendRotation(-ang, Vector3D.Z_AXIS);
}

```

The preceding code is for the special occasions where you want the entities to be in a trench or spiral shaped configuration. This is here just for fun, and as a simple way to make "flat" levels more interesting. It is a simple way to design U-shaped maps or to define squadrons of enemy fighters that appear in the game in an attractive helix-like spiral formation.

To finish off the `spawnActors` function (and also the entire class, as it only has this one function inside it), we simply need to append any offset locations that may have been passed in the parameters of the function, tell the actor pool to spawn the entity in question, and finally store some handy information for later use in-game.

```

pos.appendTranslation(
    offsetX,
    offsetY,
    offsetZ);

```

```
// does a name exist for this index?  
if (thecast[whichtile-1])  
    pool.spawn(thecast[whichtile-1], pos);  
  
// store the location for later use  
// good for pathfinding ai, etc  
leveldata[y][x] = whichtile;  
}  
break;  
}  
}  
}  
}  
  
// tell the game the coordinate of the "finish line"  
// or farthest position in the map in the z axis  
// this is used in the example game to determine when  
// you have reached the end of a level  
  
return mapImage.height * tileH;  
}  
  
} // end class  
  
} // end package
```

What just happened?

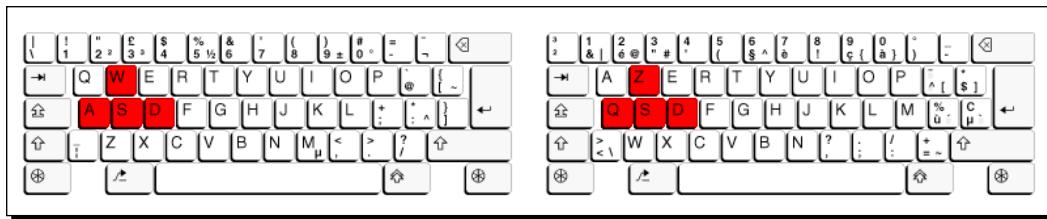
That is it for our fancy new `GameLevelParser` class! This class can now process various bitmaps that you can draw in any image editing program. This could be considered the simplest possible of all level editor applications. Simply draw your map (be it a maze or a forest) and let the `GameLevelParser` class spawn hundreds or thousands of `GameActors` in the proper places. It will allow you to sculpt giant, complex, and interesting levels by simply drawing them as a blueprint.

Upgrading the input routines

We will need to detect when the mouse has been clicked, so let's add that functionality to your `GameInput` class by allowing the game to pass an additional argument to the constructor function that is a function you want to be called when the user clicks.

While we are at it, let's extend this class to allow for extra movement keys that could be useful in other types of games. Instead of just the spacebar being used for the fire button, let's allow any of the keys that Flash games typically use for the fire button, such as *CRTL* or the letters on the bottom left of the keyboard. Finally, it is common that you want to switch weapons with the 0..9 keys, so let's add them here too.

Some international users have AZERTY keyboards (instead of the standard QWERTY layout). So, to appease them, we can also allow for alternate movement keys as they don't interfere with the conventional *W, A, S, D* configuration.



The arrow keys are still being used, but for comfort, many gamers like to use the left side of the keyboard for movement, especially if the game requires that a mouse be used as well. Users with AZERTY keyboards (as found in several European countries such as France) frequently complain about games that force them to use *W, A, S, D* for movement. The good news is that these keys can remain active while concurrently our game can support international keyboards by allowing movement as follows: *Z, Q, S, D*. They both work at the same time without problems!

Small upgrades such as making your game playable by an international audience are the kinds of little things that make a big difference. Why not be accessible to as many users as possible? Our goal is an input class designed such that whatever the user picks by gut instinct will "just work" without requiring instructions.

Time for action – adding more properties to the input class

Edit GameInput.as and make a few minor changes as follows:

```
// Stage3d game input routines version 1.2
//
package
{
    import flash.display.Stage;
    import flash.ui.Keyboard;
    import flash.events.*;
}
```

```
public class GameInput
{
    // the current state of the mouse
    public var mouseIsDown:Boolean = false;
    public var mouseClickX:int = 0;
    public var mouseClickY:int = 0;
    public var mouseX:int = 0;
    public var mouseY:int = 0;

    // the current state of the keyboard controls
    public var pressing:Object =
    { up:0, down:0, left:0, right:0, fire:0,
    strafeLeft:0, strafeRight:0,
    key0:0, key1:0, key2:0, key3:0, key4:0,
    key5:0, key6:0, key7:0, key8:0, key9:0 };

    // if mouselook is on, this is added to the chase camera
    public var cameraAngleX:Number = 0;
    public var cameraAngleY:Number = 0;
    public var cameraAngleZ:Number = 0;

    // if this is true, dragging the mouse changes the camera angle
    public var mouseLookMode:Boolean = true;

    // the game's main stage
    public var stage:Stage;

    // for click events to be sent to the game
    private var _clickfunc:Function = null;
```

The only changes from our previous version are the addition of a few extra class properties, to support the extra movement keys, and a callback function that will be called whenever the mouse is clicked.

Time for action – handling click events

The `_clickfunc` function pointer defined above needs to be set in our constructor, so edit that as follows:

```
// class constructor
public function GameInput(
    theStage:Stage,
    clickfunc:Function = null)
{
```

```

stage = theStage;
// get keypresses and detect the game losing focus
stage.addEventListener(KeyboardEvent.KEY_DOWN, keyPressed);
stage.addEventListener(KeyboardEvent.KEY_UP, keyReleased);
stage.addEventListener(Event.DEACTIVATE, lostFocus);
stage.addEventListener(Event.ACTIVATE, gainFocus);
stage.addEventListener(MouseEvent.MOUSE_DOWN, mouseDown);
stage.addEventListener(MouseEvent.MOUSE_UP, mouseUp);
stage.addEventListener(MouseEvent.MOUSE_MOVE, mouseMove);

// handle clicks in game
_clickfunc = clickfunc;
}

```

The `mouseMove` and `mouseUp` functions remain unchanged, so we won't include them here.

```

private function mouseDown(e:MouseEvent):void
{
    trace('mouseDown at '+e.stageX+', '+e.stageY);
    mouseClickX = e.stageX;
    mouseClickY = e.stageY;
    mouseIsDown = true;
    if (_clickfunc != null) _clickfunc();
}

```

The only changes in the `mouseDown` code above, from what we already have, is the one line where we optionally run the `click` function callback if it has been set.

Time for action – upgrading the key events

Finally, account for the extra movement keys we discussed above in the key events.

```

private function keyPressed(event:KeyboardEvent):void
{
    // qwer 81 87 69 82
    // asdf 65 83 68 70
    // left right 37 39
    // up down 38 40
    // 0123456789 = 48 to 57
    // zxcv = 90 88 67 86

    // trace("keyPressed " + event.keyCode);

    if (event.ctrlKey ||
        event.altKey ||

```

```
    event.shiftKey)
    pressing.fire = true;

    switch(event.keyCode)
    {
        /*
        case 81: // Q
            pressing.strafeLeft = true;
            break;
        case 69: // E
            pressing.strafeRight = true;
            break;
        */

        case Keyboard.UP:
        case 87: // W
        case 90: // Z
            pressing.up = true;
            break;

        case Keyboard.DOWN:
        case 83: // S
            pressing.down = true;
            break;

        case Keyboard.LEFT:
        case 65: // A
        case 81: // Q
            pressing.left = true;
            break;

        case Keyboard.RIGHT:
        case 68: // D
            pressing.right = true;
            break;

        case Keyboard.SPACE:
        case Keyboard.SHIFT:
        case Keyboard.CONTROL:
        case Keyboard.ENTER:
        // case 90: // z
        case 88: // x
        case 67: // c
            pressing.fire = true;
    }
```

```
        break;

    case 48: pressing.key0 = true; break;
    case 49: pressing.key1 = true; break;
    case 50: pressing.key2 = true; break;
    case 51: pressing.key3 = true; break;
    case 52: pressing.key4 = true; break;
    case 53: pressing.key5 = true; break;
    case 54: pressing.key6 = true; break;
    case 55: pressing.key7 = true; break;
    case 56: pressing.key8 = true; break;
    case 57: pressing.key9 = true; break;

    }
}
```

The focus events are virtually identical to previous versions, except that we reset even more flags in the lostFocus event.

```
private function gainFocus(event:Event):void
{
    trace("Game received keyboard focus.");
}

// if the game loses focus, don't keep keys held down
private function lostFocus(event:Event):void
{
    trace("Game lost keyboard focus.");
    pressing.up = false;
    pressing.down = false;
    pressing.left = false;
    pressing.right = false;
    pressing.strafeLeft = false;
    pressing.strafeRight = false;
    pressing.fire = false;
    pressing.key0 = false;
    pressing.key1 = false;
    pressing.key2 = false;
    pressing.key3 = false;
    pressing.key4 = false;
    pressing.key5 = false;
    pressing.key6 = false;
    pressing.key7 = false;
    pressing.key8 = false;
    pressing.key9 = false;
}
```

Finally, just as we did for the `keyPressed` event, we need to add the extra keys to our `keyReleased` function.

```
private function keyReleased(event:KeyboardEvent):void
{
    switch(event.keyCode)
    {
        /*
        case 81: // Q
            pressing.strafeLeft = false;
        break;
        case 69: // E
            pressing.strafeRight = false;
        break;
        */

        case Keyboard.UP:
        case 87: // W
        case 90: // Z
            pressing.up = false;
        break;

        case Keyboard.DOWN:
        case 83: // S
            pressing.down = false;
        break;

        case Keyboard.LEFT:
        case 65: // A
        case 81: // Q
            pressing.left = false;
        break;

        case Keyboard.RIGHT:
        case 68: // D
            pressing.right = false;
        break;

        case Keyboard.SPACE:
        case Keyboard.SHIFT:
        case Keyboard.CONTROL:
        case Keyboard.ENTER:
        // case 90: // z
        case 88: // x
```

```
case 67: // c
    pressing.fire = false;
break;

case 48: pressing.key0 = false; break;
case 49: pressing.key1 = false; break;
case 50: pressing.key2 = false; break;
case 51: pressing.key3 = false; break;
case 52: pressing.key4 = false; break;
case 53: pressing.key5 = false; break;
case 54: pressing.key6 = false; break;
case 55: pressing.key7 = false; break;
case 56: pressing.key8 = false; break;
case 57: pressing.key9 = false; break;

}
}

} // end class

} // end package
```

What just happened?

That is it for our upgraded input routines. The only difference in this version is the addition of a `mouseDown` event that can accept a callback function, and a few extra keys to support international keyboards.

Now that we have implemented some handy upgrades to our game engine, we are ready to make the final changes to our actual game code. All the low-level engine routines are in place for virtually any game to be made!

Pop quiz – earn some experience points

Here are a few quick questions to test your understanding.

1. Why is it a good idea to reuse actors in a respawn pool system?
 - a. Because it saves memory
 - b. Because it allows us to define a type of actor once and spawn multiple copies
 - c. Because it reduces the amount of garbage collection of temporary variables
 - d. All of the above

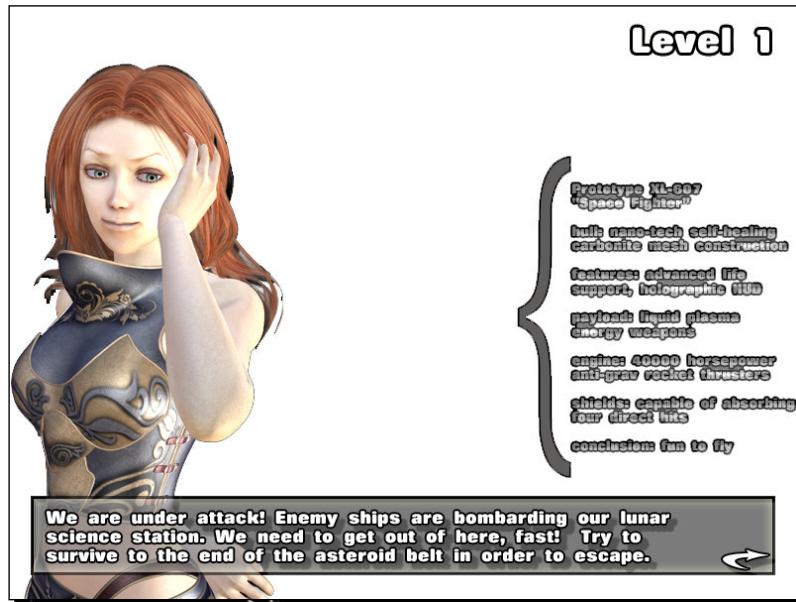
2. Why did we create a `GameActor` class that extends the base `Stage3dEntity` class?
 - a. In order to separate game-specific functionality from the low-level rendering code
 - b. Because `Stage3dEntity` is incapable of containing movement code
 - c. So that collision detection would work
 - d. To comply with the demands of the actor's union

Have a go hero – a fun side-quest: art attack!

Your side quest this time is to prepare some attractive game art for use in your final game. In preparation for *Chapter 10*, take a break from coding and sculpt a few new meshes. Draw a few new textures. Compose some intro music. Record some bombastic sound effects for things such as explosions and laser guns.

If you are feeling particularly adventurous, try writing a script for the game. Perhaps an cinematic intro, complete with voiceovers and subtitles. Why are we flying around in space blowing up bad guys? Is the Earth under attack? Have a little fun and come up with a theme for your game, complete with a cast of characters. Make up a fun little plot to flesh out the game universe.

An example of the kind of simple intro images you can add to your game without having to resort to fully animated 3D sequences is this simple transparent PNG image, ready to overlay on top of your game:



Summary

In this chapter, we managed to achieve the following milestones: we created a game actor class, we programmed a reusable actor pool, we implemented a level design map parser, and we upgraded our input routines. Our engine is ready for use in a real game.

Level 9 achieved!

The final treasure in your quest awaits. The final product. Nothing more—in terms of the low-level engine functionality—is needed. The next and final step is to utilize these new classes in our game, which is the subject of the next chapter.

10

3... 2... 1... ACTION!

You have reached the final boss battle. Level 10.

The final step in our grand adventure is to polish our game. To evolve it from tech demo to a real, finished, playable game that has a beginning, middle, and an end.

The final stages in the life cycle of a game's development can take the most time and effort. The good news is that you are already armed with all the skills required to complete it. We don't need to learn anything radically new: from here on, the primary challenge is the commitment to quality and the time it takes to get the job done.

Polish is labor-intensive busy work. None of it is particularly difficult, but it all adds up. Therefore, don't expect this final stretch in the project to breeze by: it will take longer than you expect. Why? Polish and quality assurance is where the true soul of a game takes shape. It is where you design levels, boss battles, menus, and tweak the game mechanics.

You might decide to speed up the player's movements or ask friends to play a sample level and react to their feedback. You might want to create more content, or revise the old content. You might spend weeks polishing, adding new art, tweaking particle systems, adding a few more effects, and composing music. This process can go on forever, so it is a good idea to allow yourself to cut features as often as you add new ones.

In this final chapter, we are going to finish the example "bullet hell" arcade shooter game that we have been working on. As there is a lot of work that could be done on the art side of things or in the gameplay programming that is very basic and not necessarily related to Stage3D, we will gloss over a few sections rather than spelling everything out.

You are about to continue the adventure on your own.

3... 2... 1... ACTION!

Our final quest

In order to "finish" the game, a few basic mechanics are required. We want some sort of title screen, for example, along with a logo and a start button. We also want the ability to destroy all sorts of enemies. Our enemies also need to be able to damage the player's ship, and after enough damage, we want a GAME OVER screen to be displayed.

In addition to the beginning (title screen) and ending (game over), we want there to be a game world worth playing in: a fully fleshed-out level. Therefore, along with basic gameplay elements and GUI screens, we need to design a vast level filled with enemies.

Finally, little touches of polish should be added throughout. We will discuss some simple techniques to improve the performance of the game and ways to add little extras that are needed to make it full of action.

This is what our final example game is going to look like:



If you don't want to type any code and would prefer to simply follow along with the following discussions after downloading the full source code for the final game, you can obtain it from the following URL:

<http://www.packtpub.com/adobe-flash11-stage3d-molehill-game-programming-beginners-guide/book>

You can play the game here:

http://www.mcfunkypants.com/molehill/chapter_10_demo/

Getting to the finish line

This is the phase in many game projects that can seem to take forever. The good news is that this last quest in our adventure can also be the most fun! With the core engine programming finally out of the way, we are free to let our imagination run wild. To experiment with new gameplay modes, fancy new weapons, the writing of dialogue, plot and characters, plus the creation of all sorts of great-looking art.

This is the phase in which you will want to start to show the game to your friends. Play testing is the key to making a great game. Don't make a mad dash to the finish line and expect your creation to be a big hit until you take some time afterwards to get people to play-test the game, offer feedback, find bugs, and give you some fresh insight or ideas for new features.

For the purposes of the example shoot-em-up "bullet hell" spaceship game that is the example project used during the course of this book, the first order of business is to create some more art. Design some more cool-looking spaceships, particle effects, asteroids, space stations, and decorations to flesh out your game. Use the `GameLevelParser` class we created in *Chapter 9* to implement a challenging level filled with enemies and things to blow up.

As we have already covered how to add new models to our game, instead of explaining exactly what was done to the example game, use your imagination. At this stage, you can go wild and create all sorts of interesting models and textures. You no longer need step-by-step instructions on how to get them into your game.

Apart from new art assets, polishing our shooter game should be a relatively simple process. There are a few features that are conventions in games that we should add.

Some of the final bits of polish and gameplay we need to implement include:

- ◆ Implementing a title screen
- ◆ Upgrading player movement and keeping the player within bounds
- ◆ Upgrading our GUI
- ◆ Embedding our new art assets
- ◆ Upgrading our init code
- ◆ Defining new actor types and behavior
- ◆ Creating the game level
- ◆ Improving the render loop
- ◆ Implementing game events (start, end, damage)
- ◆ Upgrading player controls
- ◆ Upgrading the step simulation function
- ◆ Publish... distribute... profit!

3... 2... 1... ACTION!

Once the game engine is "done", that one last final step might realistically take months if you decide to add a hundred levels and a deep plot! For the purposes of this beginner's guide example game, we will keep it simple. We will focus on getting the engine up and running, but let's assume the game we make is not as feature-rich as what your original masterpiece will be.

Getting your game in front of the public will, of course, most likely be the most work. It is full of the intangible "quality assurance" steps that might continue for some time until you are completely satisfied with your game. Marketing, social networking, and monetization is left as an exercise to the reader.

For this reason, that very last final polish step will be glossed over while the meat of the project, the reusable Flash 11 3D game engine, will be our primary focus. With this in mind, let's dive in!

Time for action – drawing a title screen

In order to get a title screen working in our game, all we need to do is design an attractive image using Photoshop or any other image editor. Much like in previous chapters where we created an in-game **HUD (heads-up-display overlay image)** that was saved as a large transparent .PNG file, do the same for a title screen. For example:



When your full sized image is ready to overlay on top of the game at start-up, simply embed it within your SWF in the same way you have been embedding other graphics assets—either through the [embed] flex tag or using the Flash library palette.

Once we add the title screen to the stage in our `initGUI` function, it will be displayed when the game starts. The only problem now is that it stays visible during gameplay, which itself starts immediately upon loading the game. What we really want is to track the "state" of the game and pause it until the player is ready to start.

Time for action – importing required classes

For the remainder of this chapter, all code listings are for `Stage3dGame.as`, the main file in our project. Begin by making a few changes to the top of the file to account for the classes we created in the previous chapter, plus some additional Flash functionality we are about to use.

```
package
{
    [SWF(width="640", height="480", frameRate="60",
        backgroundColor="#FFFFFF")]

    import com.adobe.utils.*;
    import flash.events.*;
    import flash.geom.*;
    import flash.utils.*;
    import flash.text.*;
    import flash.filters.GlowFilter;
    import flash.media.Sound;
    import flash.media.SoundChannel;
    import flash.system.System;
    import flash.system.Capabilities;
    import flash.system.ApplicationDomain;
    import flash.display.*;
    import flash.display3D.*;
    import flash.display3D.textures.*;
    import flash.net.*;
    import GameActor;
    import GameActorpool;
    import GameInput;
    import GameLevelParser;
    import GameParticlesystem;
    import GameTimer;
    import Stage3dEntity;
    import Stage3dParticle;
```

What just happened?

The preceding code simply imports the functionality we need for our game engine—there are just a few new lines to add for some fancy text effects, as well as our recently implemented classes. Now upgrade your class variables to account for our title screen, textFields and more.

Adding new variables to our game

The next few sections will involve adding numerous variables required for the final functionality in our game.

Time for action – tracking the game state

We are going to add tons of new variables to our game in the next few "time for action" sections. To begin with, we will implement a means to track the game state so we can have a title screen that is displayed when the game is not yet being played. Edit the properties of our Stage3dGame class as follows.

```
public class Stage3dGame extends Sprite
{
    private const VERSION_STRING:String = "Version: 1.10";

    // the current state of the game (0 = title screen)
    private const STATE_TITLESCREEN:int = 0;
    private const STATE_PLAYING:int = 1;
    private var gameState:int = STATE_TITLESCREEN;
    private var titlescreenTf:TextField;
```

What just happened?

We need to define a "state" variable for our game engine that remembers what "mode" the game is in. Perhaps we can set it such that zero means the game is sitting at the title screen and is waiting for the user to click to play. In your game, you might want to include other states, such as one that is used in between levels or a boss battle mode or a cinematic event such as a cutscene. In our example game, we only use two states: one for when we are sitting at the title screen and another for when the game is being played.

Time for action – adding variables for timer-based events

Continue adding to our game class. We will be triggering various events and keeping track of time and need some variables to hold these timestamps and the timer itself.

```
// keeps track of time passing in the game
private var gametimer:GameTimer;

// used with timers to trigger events
private var introOverlayEndTime:int = 0;
private var nextShootTime:uint = 0;
private var shootDelay:uint = 200;
private var invulnerabilityTimeLeft:int = 0;
private var invulnerabilityMsWhenHit:int = 3000;
private var screenShakes:int = 0;
private var screenShakeCameraAngle:Number = 0;
```

What just happened?

We are going to be triggering time-based game events such as controlling how often you can shoot, shaking the screen when we get hit, making the player invulnerable for a period of time after getting damaged, and a simple "intro cinematic" where the camera angle changes for the first few seconds of the game. These variables will be used to hold timestamps to keep track of when things need to be triggered.

When the user hits the fire button (spacebar, and so on), we want to shoot bullets forward. We don't however, want a bullet to be spawned, every single frame, but instead after a specific amount of time. The shoot time and delay variables in the preceding code account for this.

When we get damaged by an enemy, a common shooter game convention is that the player cannot be hit for a small amount of time while they regain control and move farther away from whatever threat just hurt them. A small window of time where the player is invulnerable sounds like a fun game mechanic.

Time for action – adding movement related variables

```
// handles keyboard and mouse inputs
private var gameinput:GameInput;

// nice close behind chase camera
private var chaseCameraXangle:Number = -5;
private var chaseCameraDistance:Number = 4;
```

3... 2... 1... ACTION!

```
private var chaseCameraHeight:Number = 2;

// how fast we move forward in units per ms
private var flySpeed:Number = 1;
// how fast we move up/down/left/right units per ms
private var moveSpeed:Number = 0.02;
// a slow orbit of the asteroids in deg per ms
private var asteroidRotationSpeed:Number = 0.002;

// force the ship to stay within bounds (0 = no restriction)
private var playerMaxx:Number = 32;
private var playerMaxy:Number = 32;
private var playerMaxz:Number = 0;
private var playerMinx:Number = -32;
private var playerMiny:Number = 0.1;
private var playerMinz:Number = 0;
private var levelCompletePlayerMinz:Boolean = true;
```

What just happened?

The preceding variables are used to hold the game input class and affect the movement of the player's ship. Speeds and maximum and minimum locations help to keep the player near the action and moving quickly enough to be exciting.

In this game, we want the ship to continue to move forward at all times, so that the arrow keys can be used to dodge bullets and enemies. Therefore, the speed at which we fly forward, as well as the arrow key dodge speed, should be defined.

Additionally, we don't want the player to hold down a movement key and slowly leave the entire level behind, so we should restrict the player movement to a range that forces them to stay near the action. Some maximum and minimum position coordinates will do the trick.

If the level complete Boolean value in the preceding code is true, the game is "over" when the finish line has been reached. This position will be defined by the size of our level.

Time for action – keeping track of all entities

We need to keep track of several actor pools and particle systems, along with single game entities that are used during the game.

```
// all known entities in the world
private var chaseCamera:Stage3dEntity
private var props:Vector.<Stage3dEntity>;
private var player:GameActor;
private var enemies:GameActorpool;
```

```
private var playerBullets:GameActorpool;
private var enemyBullets:GameActorpool;
private var explo:Stage3dParticle;
private var particleSystem:GameParticlesystem;
// we want to remember these for use in gameStep()
private var asteroids1:Stage3dEntity;
private var asteroids2:Stage3dEntity;
private var asteroids3:Stage3dEntity;
private var asteroids4:Stage3dEntity;
private var engineGlow:Stage3dEntity;
private var sky:Stage3dEntity;
// reusable generic entity (for speed and to avoid GC)
private var entity:Stage3dEntity;
private var actor:GameActor;
```

What just happened?

The lists of entities in the preceding code have been modified to make use of our feature-rich actor pool system, where appropriate. From now on, we want to track bullets and enemies in an actor pool, so that they can be animated and checked for collisions.

Time for action – upgrading the HUD

Next, the GUI text elements are defined, along with variables used to keep track of the values that will be displayed on the screen during the game as part of the HUD.

```
// used by the GUI
private var fpsLast:uint = getTimer();
private var fpsTicks:uint = 0;
private var fpsTf:TextField;
private var gpuTf:TextField;
private var scoreTf:TextField;
private var healthTf:TextField;
private var score:uint = 0;
private var combo:uint = 0;
private var comboBest:uint = 0;
private var scenePolycount:uint = 0;
// used for the health bar display
private var healthText_0:String = '';
private var healthText_1:String =
'| | | | |';
private var healthText_2:String =
'| | | | | | | | | |';
private var healthText_3:String =
'| | | | | | | | | | | | | | | |';
```

What just happened?

Of interest is a "health meter" which in the example game is a simple set of glowing lines that reflect how much damage has been inflicted upon us.

Additionally, a fun game mechanic that can add to the depth of many shooter games, is a "combo" counter. This combo counter keeps track of how many enemies have been hit in succession. Players with true finesse may try to avoid spamming bullets everywhere in order to achieve an impressive combo count.

Time for action – defining variables used by Stage3D

We will be using the following variables that are related to Stage3D:

```
// the 3d graphics window on the stage  
private var context3D:Context3D;  
// the compiled shader used to render our meshes  
private var shaderProgram1:Program3D;  
// a simple shader with one static light  
private var shaderWithLight:Program3D;  
// matrices that affect the mesh location and camera angles  
private var projectionmatrix:PerspectiveMatrix3D =  
    new PerspectiveMatrix3D();  
private var viewmatrix:Matrix3D = new Matrix3D();
```

What just happened?

The preceding variables are similar to those used in most of the previous demos in other chapters to keep track of the Context3D, camera angles, and shaders. There is one new shader that we will be implementing—one that has light.

Adding art to our game

The next few sections will involve embedding all the art assets required for our game into the code.

Time for action – embedding our new art assets (AS3 version)

We need to embed all the images and MP3 files required to deliver attractive visuals and sound. If you are using the Adobe Flash IDE, then skip this section as it only applies if you are using FlashDevelop, FlashBuilder, FDT, Flex, haXe, and so on.

```
[Embed (source = "art/player.jpg")]  
private var playerTextureBitmap:Class;
```

```

private var playerTextureData:Bitmap = new playerTextureBitmap();
[Embed (source = "art/terrain_texture.jpg")]
private var terrainTextureBitmap:Class;
private var terrainTextureData:Bitmap = new terrainTextureBitmap();
[Embed (source = "art/craters.jpg")]
private var cratersTextureBitmap:Class;
private var cratersTextureData:Bitmap = new cratersTextureBitmap();
[Embed (source = "art/sky.jpg")]
private var skyTextureBitmap:Class;
private var skyTextureData:Bitmap = new skyTextureBitmap();
[Embed (source = "art/engine.jpg")]
private var puffTextureBitmap:Class;
private var puffTextureData:Bitmap = new puffTextureBitmap();
[Embed (source = "art/particle1.jpg")]
private var particle1data:Class;
private var particle1bitmap:Bitmap = new particle1data();
[Embed (source = "art/particle2.jpg")]
private var particle2data:Class;
private var particle2bitmap:Bitmap = new particle2data();
[Embed (source = "art/particle3.jpg")]
private var particle3data:Class;
private var particle3bitmap:Bitmap = new particle3data();
[Embed (source = "art/particle4.jpg")]
private var particle4data:Class;
private var particle4bitmap:Bitmap = new particle4data();

```

Next, we embed a few new textures for use on the enemies and other structures such as space stations that appear in the game.

```

[Embed (source = "art/badguys.jpg")]
private var badguyTextureData:Class;
private var badguyTextureBitmap:Bitmap = new badguyTextureData();
[Embed (source = "art/hulltech.jpg")]
private var techTextureData:Class;
private var techTextureBitmap:Bitmap = new techTextureData();

```

Additional bitmaps are needed for the GUI: a title screen, an in-game overlay, and a little intro screen where we instruct the player what their "mission" is.

```

[Embed (source = "art/title_screen.png")]
private var titleScreenData:Class;
private var titleScreen:Bitmap = new titleScreenData();
[Embed (source = "art/hud_overlay.png")]
private var hudOverlayData:Class;
private var hudOverlay:Bitmap = new hudOverlayData();

```

3... 2... 1... ACTION!

```
[Embed (source = "art/intro.png")]
private var introOverlayData:Class;
private var introOverlay:Bitmap = new introOverlayData();
```

Now that we have a level map parsing class, we can embed the key image (defining what colors represent what type of game entity) and the map image (the blueprint for our game world).

```
[Embed(source = 'art/level_key.gif')]
private static const LEVELKEYDATA:Class;
private var levelKey:Bitmap = new LEVELKEYDATA();
[Embed(source = 'art/level_00.gif')]
private static const LEVELDATA:Class;
private var levelData:Bitmap = new LEVELDATA();
```

This is the font that is used by the GUI. We define the ranges of characters we need to make the .SWF smaller.

```
[Embed (source = 'art/gui_font.ttf',
embedAsCFF = 'false',
fontFamily = 'guiFont',
mimeType = 'application/x-font-truetype',
unicodeRange='U+0020-U+002F, U+0030-U+0039, U+003A-U+0040,
U+0041-U+005A, U+005B-U+0060, U+0061-U+007A, U+007B-U+007E')
private const GUI_FONT:Class;
```

Finally, we embed some sounds and music as follows:

```
[Embed (source = "art/sfxmusic.mp3")]
public var introMp3:Class;
[Embed (source = "art/sfxblast.mp3")]
public var blastMp3:Class;
[Embed (source = "art/sfxexplode.mp3")]
public var explodeMp3:Class;
[Embed (source = "art/sfxgun.mp3")]
public var gunMp3:Class;
```

What just happened?

The artwork required by our GUI and the in-game textures is embedded in exactly the same way as in previous chapters.

The LEVELDATA and LEVELKEYDATA art assets that are needed by our game are the level blueprint map. In the previous chapter, we wrote a simple GameLevelParser class that takes a tiny key image to use for looking up RGB values, plus a level data "map" image upon which you drew the actual level.

Instead of using the default textField font (which in most cases is an ugly Times New Roman), we can embed fancy fonts. In the art subfolder of our source code directory, we can include a truetype font that will be included inside our SWF. The `unicodeRange` parameter instructs Flash not to include the entire font but only the characters that are needed such as letters, numbers, and punctuation, which save on file size by not including foreign language glyphs.

Finally, MP3 files are also being embedded, so that our game will be able to output sound and music.

Time for action – embedding our new art assets (CS5 version)

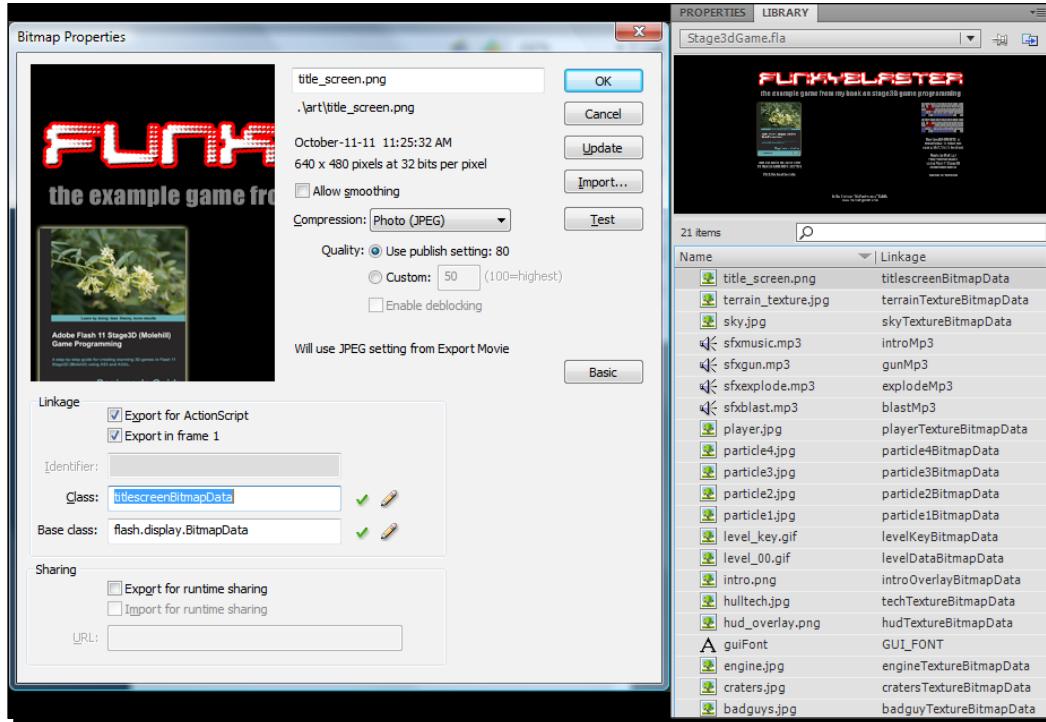
The following code can be skipped if you used the preceding section, but is required if you are using Adobe Flash CS5. Remember that, instead of `[embed]`, we need to drag the files onto the library palette in the Flash editor and right-click on the entry to edit its properties. This is also the same as last time except we are also adding a font and some sounds.

```
private var playerTextureData:Bitmap = // player.jpg
    new Bitmap(new playerTextureBitmapData(256,256));
private var badguyTextureBitmap:Bitmap = // badguys.jpg
    new Bitmap(new badguyTextureBitmapData(256,256));
private var techTextureBitmap:Bitmap = // hulltech.jpg
    new Bitmap(new techTextureBitmapData(256,256));
private var terrainTextureData:Bitmap = // terrainTexture.jpg
    new Bitmap(new terrainTextureBitmapData(512,512));
private var cratersTextureData:Bitmap = // craters.jpg
    new Bitmap(new cratersTextureBitmapData(512,512));
private var skyTextureData:Bitmap = // sky.jpg
    new Bitmap(new skyTextureBitmapData(512,512));
private var puffTextureData:Bitmap = // engine.jpg
    new Bitmap(new engineTextureBitmapData(128,128));
private var particle1bitmap:Bitmap = // particle1.jpg
    new Bitmap(new particle1BitmapData(128,128));
private var particle2bitmap:Bitmap = // particle2.jpg
    new Bitmap(new particle2BitmapData(128,128));
private var particle3bitmap:Bitmap = // particle3.jpg
    new Bitmap(new particle3BitmapData(128,128));
private var particle4bitmap:Bitmap = // particle4.jpg
    new Bitmap(new particle4BitmapData(128,128));
private var titleScreen:Bitmap = // titleScreen.png
    new Bitmap(new titlescreenBitmapData(640,480));
private var hudOverlay:Bitmap = // hudOverlay.png
    new Bitmap(new hudTextureBitmapData(640,480));
private var introOverlay:Bitmap = // intro.png
```

3... 2... 1... ACTION!

```
new Bitmap(new introOverlayBitmapData(640,480));
private var levelKey:Bitmap = // level_key.gif
    new Bitmap(new levelKeyBitmapData(32,4));
private var levelData:Bitmap = // level_00.gif
    new Bitmap(new levelDataBitmapData(15,640));
```

This is what your Flash library palette should look like when you are done:



What just happened?

Just as before, all we did was embed the required art assets into our Flash file. Take note of the names used and remember that each is made to be exported for ActionScript in frame 1.

Time for action – embedding all the meshes

The final step required to include the art used in our game is to embed the .OBJ files that define all the mesh geometry. This step is the same no matter which development environment you are working in.

```
// MESH DATA
// the player - 142 polygons
```

```
[Embed (source = "art/spaceship_funky.obj",
        mimeType = "application/octet-stream")]
private var myObjData5:Class;
// the engine glow - 336 polygons
[Embed (source = "art/puff.obj",
        mimeType = "application/octet-stream")]
private var puffObjData:Class;
```

After meshes used by the player's spaceship, we embed various meshes used for the game environment—the terrain, the sky, and an asteroid field.

```
// The terrain mesh data - 8192 polygons
[Embed (source = "art/terrain.obj",
        mimeType = "application/octet-stream")]
private var terrainObjData:Class;
// an asteroid field - 3280 polygons
[Embed (source = "art/asteroids.obj",
        mimeType = "application/octet-stream")]
private var asteroidsObjData:Class;
// the sky - 768 polygons
[Embed (source = "art/sky.obj",
        mimeType = "application/octet-stream")]
private var skyObjData:Class;
```

Next, we add some meshes to be used in our particle systems, and for bullets.

```
// explosion start - 336 polygons
[Embed (source = "art/explosion1.obj",
        mimeType = "application/octet-stream")]
private var explosion1_data:Class;
// explosion end - 336 polygons
[Embed (source = "art/explosion2.obj",
        mimeType = "application/octet-stream")]
private var explosion2_data:Class;
// a "thicker" explosion end - 336 polygons
[Embed (source = "art/explosionfat.obj",
        mimeType = "application/octet-stream")]
private var explosionfat_data:Class;
// bullet - 8 polygons
[Embed (source = "art/bullet.obj",
        mimeType = "application/octet-stream")]
private var bulletData:Class;
```

3... 2... 1... ACTION!

We will add some models for enemies we want to destroy: several sentry guns and fuel tanks and a huge boss ship for variety.

```
// sentry guns - all share one texture
[Embed (source = "art/sentrygun01.obj",
        mimeType = "application/octet-stream")]
private var sentrygun01_data:Class;
[Embed (source = "art/sentrygun02.obj",
        mimeType = "application/octet-stream")]
private var sentrygun02_data:Class;
[Embed (source = "art/sentrygun03.obj",
        mimeType = "application/octet-stream")]
private var sentrygun03_data:Class;
[Embed (source = "art/sentrygun04.obj",
        mimeType = "application/octet-stream")]
private var sentrygun04_data:Class;
// fuel tanks - uses same texture as sentries
[Embed (source = "art/fueltank01.obj",
        mimeType = "application/octet-stream")]
private var fueltank01_data:Class;
[Embed (source = "art/fueltank02.obj",
        mimeType = "application/octet-stream")]
private var fueltank02_data:Class;
// large boss ship
[Embed (source = "art/boss.obj",
        mimeType = "application/octet-stream")]
private var bossData:Class;
```

Additionally, we want there to be a bunch of asteroids scattered all over the level that the player will need to dodge or destroy.

```
private var astroTunnelData:Class;
// a single asteroid
[Embed (source = "art/asteroid_00.obj",
        mimeType = "application/octet-stream")]
private var asteroid_00_data:Class;
[Embed (source = "art/asteroid_01.obj",
        mimeType = "application/octet-stream")]
private var asteroid_01_data:Class;
[Embed (source = "art/asteroid_02.obj",
        mimeType = "application/octet-stream")]
private var asteroid_02_data:Class;
// a large flat "mesa" (circular rock with a flat top)
[Embed (source = "art/island.obj",
        mimeType = "application/octet-stream")]
private var islandData:Class;
```

To add a little more visual pizzaz, some huge space stations are going to be floating around off the side of the action. These are meant to be nothing more than eye-candy in the background.

```
// space stations and "slabs of tech"
[Embed (source = "art/slab01.obj",
        mimeType = "application/octet-stream")]
private var slab01_data:Class;
[Embed (source = "art/slab02.obj",
        mimeType = "application/octet-stream")]
private var slab02_data:Class;
[Embed (source = "art/slab03.obj",
        mimeType = "application/octet-stream")]
private var slab03_data:Class;
[Embed (source = "art/slab04.obj",
        mimeType = "application/octet-stream")]
private var slab04_data:Class;
[Embed (source = "art/station01.obj",
        mimeType = "application/octet-stream")]
private var station01_data:Class;
[Embed (source = "art/station02.obj",
        mimeType = "application/octet-stream")]
private var station02_data:Class;
[Embed (source = "art/station03.obj",
        mimeType = "application/octet-stream")]
private var station03_data:Class;
[Embed (source = "art/station04.obj",
        mimeType = "application/octet-stream")]
private var station04_data:Class;
[Embed (source = "art/station05.obj",
        mimeType = "application/octet-stream")]
private var station05_data:Class;
[Embed (source = "art/station06.obj",
        mimeType = "application/octet-stream")]
private var station06_data:Class;
[Embed (source = "art/station07.obj",
        mimeType = "application/octet-stream")]
private var station07_data:Class;
[Embed (source = "art/station08.obj",
        mimeType = "application/octet-stream")]
private var station08_data:Class;
[Embed (source = "art/station09.obj",
        mimeType = "application/octet-stream")]
private var station09_data:Class;
```

3... 2... 1... ACTION!

```
[Embed (source = "art/station10.obj",
        mimeType = "application/octet-stream")]
private var station10_data:Class;
```

Finally, we need something to represent the "finish line" for our simple game. At the very end of the level, there will be a spinning wormhole that is meant to provide a means of escape and serves as the goal.

```
// a spinning wormhole
[Embed (source = "art/wormhole.obj",
        mimeType = "application/octet-stream")]
private var wormholeData:Class;
```

Time for action – keeping track of art assets

The preceding embedded art data will be turned into textures and sounds for use in the game. We need to keep track of these objects by storing them in the following variables:

```
// all textures used in the game
private var playerTexture:Texture;
private var terrainTexture:Texture;
private var cratersTexture:Texture;
private var skyTexture:Texture;
private var puffTexture:Texture;
private var particle1texture:Texture;
private var particle2texture:Texture;
private var particle3texture:Texture;
private var particle4texture:Texture;
private var badguytexture:Texture;
private var techtexture:Texture;
// the sounds used in the game
public var introSound:Sound = (new introMp3) as Sound;
public var blastSound:Sound = (new blastMp3) as Sound;
public var explodeSound:Sound = (new explodeMp3) as Sound;
public var gunSound:Sound = (new gunMp3) as Sound;
public var musicChannel:SoundChannel;
```

What just happened?

That is it for all the game-related variables and embedded art assets! We have successfully embedded all sorts of new artwork, including bitmaps used as GUI overlays, as well as textures and 3D meshes along with some sound and music.

Upgrading the final game source code

We are done with the art side of things. On to the code. The following sections will explain all the changes and upgrades that our game requires to bring it to a playable state.

Time for action – upgrading the inits

Now that we have reached the final stage of our game's development, let's use all the best practices for initializing Stage3D, including detecting previous versions of Flash that don't yet support it.

```
public function Stage3dGame()
{
    if (stage != null)
        init();
    else
        addEventListener(Event.ADDED_TO_STAGE, init);
}

private function init(e:Event = null):void
{
    if (hasEventListener(Event.ADDED_TO_STAGE))
        removeEventListener(Event.ADDED_TO_STAGE, init);

    // class constructor - sets up the stage
    stage.scaleMode = StageScaleMode.NO_SCALE;
    stage.align = StageAlign.TOP_LEFT;

    // add some text labels
    initGUI();

    // are we running Flash 11 with Stage3D available?
    var stage3DAvailable:Boolean =
        ApplicationDomain.currentDomain.hasDefinition
            ("flash.display.Stage3D");
    if (stage3DAvailable)
    {
        // start the game timer
        gametimer = new GameTimer(heartbeat,3333);
        gameinput = new GameInput(stage, gamestart);
        // used to hold certain level decorations
        props = new Vector.<Stage3dEntity>();
        // detect when we get a context3D
        stage.stage3Ds[0].addEventListener(

```

3... 2... 1... ACTION!

```
    Event.CONTEXT3D_CREATE, onContext3DCreate);
    // detect when the swf is not using wmode=direct
    stage.stage3Ds[0].addEventListener(
        ErrorEvent.ERROR, onStage3DError);
    // request hardware 3d mode now
    stage.stage3Ds[0].requestContext3D();
}
else
{
    trace("stage3DAvailable is false!");
    titlescreenTf.text = 'Flash 11 Required.\n' +
        'Your version: ' + Capabilities.version +
        '\nThis game uses Stage3D.\n' +
        'Please upgrade to Flash 11\n' +
        'so you can play 3d games!';
}
}
```

The following error event is fired if the swf is not using wmode=direct. Stage3D requires that the `embed` tag in your HTML file uses this parameter.

```
private function onStage3DError(e:ErrorEvent):void
{
    trace("onStage3DError!");
    titlescreenTf.text = 'Your Flash 11 settings\n' +
        'have 3D turned OFF.\n' +
        'Ensure that you use\n' +
        'wmode=direct\n' +
        'in your html file.';
}
```

What just happened?

The preceding init code is almost exactly the same as in previous chapters except that we have to account for our title screen and game state as discussed earlier. The game starts in the "paused" mode and does not begin until the player clicks on the title screen to begin. Therefore, we have added a reference to the `gamestart` function to the `gameinput` constructor at the top of the `init` routine.

We have also implemented a bit more error detection logic than before by performing a check to see if the version of Flash being used by the user actually supports Stage3D. If it doesn't, we need to simply output an error message as the game will not run in Flash 10. Additionally, we also create an event to detect any Flash 11 specific errors and report on those as well.

Little bits of polish like this are the difference between a tech demo and a finished game that gracefully adapts to errors during init.

Time for action – initializing Stage3D

As in previous chapters, this function does all the heavy lifting when it comes to setting up Flash for 3D graphics:

```
private function onContext3DCreate(event:Event):void
{
    // Remove existing frame handler. Note that a context
    // loss can occur at any time which will force you
    // to recreate all objects we create here.
    // A context loss occurs for instance if you hit
    // CTRL-ALT-DELETE on Windows.
    // It takes a while before a new context is available
    // hence removing the enterFrame handler is important!

    if (hasEventListener(Event.ENTER_FRAME))
        removeEventListener(Event.ENTER_FRAME,enterFrame);

    // Obtain the current context
    var mystage3D:Stage3D = event.target as Stage3D;
    context3D = mystage3D.context3D;

    if (context3D == null)
    {
        // Currently no 3d context is available (error!)
        trace('ERROR: no context3D - video driver problem?');
        return;
    }

    // detect software mode (html might not have wmode=direct)
    if ((context3D.driverInfo == Context3DRenderMode.SOFTWARE)
        || (context3D.driverInfo.indexOf('oftware')>-1))
    {
        //Context3DRenderMode.AUTO
        trace("Software mode detected!");
        titlescreenTf.text = 'Your Flash 11 settings\n' +
            'have 3D turned OFF.\n' +
            'Ensure that you are\n' +
            'using a shader 2.0\n' +
            'compatible 3d card.';
    }
}
```

3... 2... 1... ACTION!

```
// if this is too big, it changes the stage size!
gpuTf.text = 'Flash Version: ' + Capabilities.version +
    ' - Game ' + VERSION_STRING +
    ' - 3D mode: ' + context3D.driverInfo;

Disabling error checking will drastically improve performance. As this is now the final version
of our game, we no longer want to be working in any form of "debug mode". All we want at
this point is the fastest possible rendering speed. You may want to leave this set to true until
the game is working perfectly—just remember to disable error checking in the final game
that you send to friends or upload to websites.

// If set to true, Flash sends helpful error messages regarding
// AGAL compilation errors, uninitialized program constants, etc.
context3D.enableErrorChecking = false;

// The 3d back buffer size is in pixels (2=antialiased)
context3D.configureBackBuffer(stage.width, stage.height, 2, true);

// assemble all the shaders we need
initShaders();

// init all textures
playerTexture = initTexture(playerTextureData);
terrainTexture = initTexture(terrainTextureData);
cratersTexture = initTexture(cratersTextureData);
puffTexture = initTexture(puffTextureData);
skyTexture = initTexture(skyTextureData);
particle1texture = initTexture(particle1bitmap);
particle2texture = initTexture(particle2bitmap);
particle3texture = initTexture(particle3bitmap);
particle4texture = initTexture(particle4bitmap);
badguytexture = initTexture(badguyTextureBitmap);
techtexture = initTexture(techTextureBitmap);

// Initialize our mesh data - requires shaders and textures first
initData();

// create projection matrix for our 3D scene
projectionmatrix.identity();
// 45 degrees FOV, 640/480 aspect ratio, 0.1=near, 150000=far
projectionmatrix.perspectiveFieldOfViewRH(
    45, stage.width / stage.height, 0.01, 150000.0);

// start the render loop!
addEventListener(Event.ENTER_FRAME,enterFrame);
}
```

What just happened?

As before, the `onContext3DCreate` function ensures that all data required by our game is uploaded to the video card. All textures are created and meshes are parsed and stored in video memory, ready to be rendered. Shaders are compiled and everything is set up to be ready for rendering.

The one major difference compared to all previous iterations of this function is that we have turned OFF `context3D.enableErrorChecking` for better performance. This one line is very important and should only be set to false when the game is ready for public use and you want the very best performance possible.

 The reason `context3D.enableErrorChecking = false` improves performance is that when error checking is turned off, the rendering is allowed to happen in parallel with other code execution rather than "blocking" further code by having to wait to see if there were any errors. In complex games, this non-blocking behavior means that your graphics card will be happily drawing things at the same time as your Flash game is busy doing other things; thereby really taking advantage of hardware accelerated 3D graphics by working the GPU and the CPU at the same time.

Time for action – upgrading the initGUI function

Our final `initGUI` function is very similar to those from previous chapters except that now we have included our new title screen overlay image and are also using an embedded font with some fancy glow effects. Hopefully, this will make things look more game-like and professional.

```
private function initGUI():void
{
    // titlescreen
    addChild(titleScreen);

    var myFont:Font = new GUI_FONT();
```

Next, we add a title screen "game over" notice as follows:

```
var gameoverFormat:TextFormat = new TextFormat();
gameoverFormat.color = 0x000000;
gameoverFormat.size = 24;
gameoverFormat.font = myFont.fontName;
gameoverFormat.align = TextFormatAlign.CENTER;
titlescreenTf = new TextField();
titlescreenTf.embedFonts = true;
```

3... 2... 1... ACTION!

```
titlescreenTf.x = 0;
titlescreenTf.y = 180;
titlescreenTf.width = 640;
titlescreenTf.height = 300;
titlescreenTf.autoSize = TextFieldAutoSize.NONE;
titlescreenTf.selectable = false;
titlescreenTf.defaultTextFormat = gameoverFormat;
titlescreenTf.filters = [
    new GlowFilter(0xFF0000, 1, 2, 2, 2, 2),
    new GlowFilter(0xFFFFFFF, 1, 16, 16, 2, 2)];
titlescreenTf.text = "CLICK TO PLAY\n\n"
    + "Your objective:\n"
    + "reach the wormhole!";
addChild(titlescreenTf);
```

We add a heads-up-display overlay to the screen below.

```
addChild(hudOverlay);
```

The following code sets up a text format descriptor used by all GUI labels. It uses the font that we embedded earlier.

```
var myFormat:TextFormat = new TextFormat();
myFormat.color = 0xFFFFAA;
myFormat.size = 16;
myFormat.font = myFont.fontName;
```

Next, we create an FPSCOUNTER that displays the framerate on the screen as follows:

```
fpsTf = new TextField();
fpsTf.embedFonts = true;
fpsTf.x = 0;
fpsTf.y = 0;
fpsTf.selectable = false;
fpsTf.autoSize = TextFieldAutoSize.LEFT;
fpsTf.defaultTextFormat = myFormat;
fpsTf.filters = [new GlowFilter(0x000000, 1, 2, 2, 2, 2)];
fpsTf.text = "Initializing Stage3d...";
addChild(fpsTf);
```

We also initialize a nicely formatted, glowing score display, a debug display that resides at the bottom of the screen for listing the driver details and Flash version and a health meter. Instead of using bitmaps, the health meter is made out of text for simplicity.

```
scoreTf = new TextField();
scoreTf.embedFonts = true;
scoreTf.x = 540;
scoreTf.y = 0;
scoreTf.selectable = false;
scoreTf.autoSize = TextFieldAutoSize.LEFT;
scoreTf.defaultTextFormat = myFormat;
scoreTf.filters = [new GlowFilter(0xFF0000, 1, 2, 2, 2, 2)];
scoreTf.text = VERSION_STRING;
addChild(scoreTf);

// debug info: list the video card details
var gpuFormat:TextFormat = new TextFormat();
gpuFormat.color = 0xFFFFFFFF;
gpuFormat.size = 10;
gpuFormat.font = myFont.fontName;
gpuFormat.align = TextFormatAlign.CENTER;
gpuTf = new TextField();
gpuTf.embedFonts = true;
gpuTf.x = 0;
gpuTf.y = stage.height - 16;
gpuTf.selectable = false;
gpuTf.width = stage.width-4;
gpuTf.autoSize = TextFieldAutoSize.NONE;
gpuTf.defaultTextFormat = gpuFormat;
gpuTf.text = "Flash Version: " +
    Capabilities.version + " - Game " + VERSION_STRING;
addChild(gpuTf);

// add a health meter
healthTf = new TextField();
healthTf.embedFonts = true;
healthTf.x = 232;
healthTf.y = 3;
healthTf.selectable = false;
healthTf.autoSize = TextFieldAutoSize.LEFT;
healthTf.defaultTextFormat = myFormat;
healthTf.filters = [new GlowFilter(0xFFFFFFF, 1, 4, 4, 4, 2)];
healthTf.text = healthText_3;
addChild(healthTf);

}
```

3... 2... 1... ACTION!

What just happened?

The preceding code will have upgraded our GUI and enabled a starting screen that waits for the user to click before starting the gameplay. It should look like this:



Time for action – upgrading the texture inits

In the previous version of our game engine, each texture was initialized using very similar-looking code. As any programmer knows, if code is getting copy-n-pasted more than twice in any section, consideration should be made towards creating an abstract function that can do the work.

```
public function uploadTextureWithMipmaps(
    dest:Texture, src:BitmapData):void
{
    var ws:int = src.width;
    var hs:int = src.height;
    var level:int = 0;
    var tmp:BitmapData;
```

```
var transform:Matrix = new Matrix();

tmp = new BitmapData( src.width, src.height, true, 0x00000000);

while ( ws >= 1 && hs >= 1 )
{
    tmp.draw(src, transform, null, null, null, true);
    dest.uploadFromBitmapData(tmp, level);
    transform.scale(0.5, 0.5);
    level++;
    ws >>= 1;
    hs >>= 1;
    if (hs && ws)
    {
        tmp.dispose();
        tmp = new BitmapData(ws, hs, true, 0x00000000);
    }
}
tmp.dispose();
}

private function initTexture(bmp:Bitmap):Texture
{
    var tex:Texture;

    tex = context3D.createTexture(bmp.width, bmp.height,
        Context3DTextureFormat.BGRA, false);

    uploadTextureWithMipmaps(tex, bmp.bitmapData);

    return tex;
}
```

What just happened?

The `uploadTextureWithMipmaps` remains identical from last time but now we have implemented a fancy new `initTexture` function, so that we can create multiple textures without so much duplicated code.

Time for action – upgrading the shaders

The `initShaders` routine from previous chapters was fine, but to add a more professional touch to our renderings, we will add a second shader with dynamic lighting.

```
private function initShaders():void
{
    trace("initShaders...");
    // A simple vertex shader which does a 3D transformation
    var vertexShaderAssembler:AGALMiniAssembler =
        new AGALMiniAssembler();
    vertexShaderAssembler.assemble
    (
        Context3DProgramType.VERTEX,
        // 4x4 matrix multiply to get camera angle
        "m44 op, va0, vc0\n" +
        // tell fragment shader about XYZ
        "mov v0, va0\n" +
        // tell fragment shader about UV
        "mov v1, va1\n" +
        // tell fragment shader about RGBA
        "mov v2, va2\n" +
        // v2 tell fragment shader about normals
        "mov v3, va3"
    );
    // textured using UV coordinates
    var fragmentShaderAssembler:AGALMiniAssembler
        = new AGALMiniAssembler();
    fragmentShaderAssembler.assemble
    (
        Context3DProgramType.FRAGMENT,
        // grab the texture color from texture 0
        // and uv coordinates from varying register 1
        // and store the interpolated value in ft0
        "tex ft0, v1, fs0 <2d,linear,repeat,miplinear>\n" +
        // move this value to the output color
        "mov oc, ft0\n"
    );
    // combine shaders into a program which we then upload to the GPU
    shaderProgram1 = context3D.createProgram();
    shaderProgram1.upload(
        vertexShaderAssembler.agalcode,
        fragmentShaderAssembler.agalcode);

    trace("Shader 1 uploaded.");
}
```

The preceding code is verbatim the same as in previous chapters. It creates a fragment and vertex program that simply draws textured polygons on the screen.

```
// this is a simple textured shader
// with dynamic lighting
trace("Compiling lit vertex program...");

var litVertexShaderAssembler:AGALMiniAssembler =
    new AGALMiniAssembler();
litVertexShaderAssembler.assemble
(
    Context3DProgramType.VERTEX,
    // 4x4 matrix multiply to get camera angle
    "m44 vt0, va0, vc0\n" +
    // output position
    "mov op, vt0 \n" +
    // tell fragment shader about XYZ
    "mov v0, va0\n" +
    // tell fragment shader about UV
    "mov v1, va1\n" +
    // tell fragment shader about RGBA
    "mov v2, va2\n" +
    // tell fragment shader about normals
    // without regard to camera angle:
    // for simplistic lighting
    "mov v3, va3"
);

```

The vertex program for our dynamically lit shader is much like previous ones except we send the normals for each vertex to our fragment program, so that it knows in which direction each polygon is facing and can shade our models appropriately depending on the angle of the light.

```
// textured using UV coordinates and LIT with a static light
trace("Compiling lit fragment program...");

var litFragmentShaderAssembler:AGALMiniAssembler =
    new AGALMiniAssembler();
litFragmentShaderAssembler.assemble
(
    Context3DProgramType.FRAGMENT,
    // grab the texture color from texture 0
    // and uv coordinates from varying register 1
    // and store the interpolated value in ft0
    "tex ft0, v1, fs0 <2d,linear,repeat,miplinear>\n" +
    // calculate dotproduct of vert normal and light normal
    // fc15 = light normal

```

3... 2... 1... ACTION!

```
"dp3 ft1 fc15 v3\n" +
// add the ambient light (fc16)
"add ft1 ft1 fc16\n" +
// multiply texture color by light value
"mul ft0 ft0 ft1\n" +
// move this value to the output color
"mov oc, ft0\n"
);
trace("Uploading lit shader....");
shaderWithLight = context3D.createProgram();
shaderWithLight.upload(
    litVertexShaderAssembler.agalcode,
    litFragmentShaderAssembler.agalcode);
```

The fragment program for our lighting shader not only samples the texture much like previous examples, but also takes into account the normal for each vertex and compares it with the light normal (the direction the light is facing). Finally, it adds the ambient light to this value to arrive at the final amount of light being hit by that particular fragment and then multiplies it with the color of the sampled texture, effectively "tinting" the texture so that areas not facing the light source will be darker.

The light normal and ambient light color are stored in fragment program registers `fc15` and `fc16` and are defined below. By tweaking these values, you could make the entire scene look like it is being lit by a colored light and make the shadows appear on the surface at a different angle.

```
trace("Setting lit shader constants....");

// for simplicity, the light position never changes
// so we can set it here once rather than every frame
var lightDirection:Vector3D = new Vector3D(1.5, 1, -2, 0);
lightDirection.normalize();
var lightvector:Vector.<Number> = new Vector.<Number>();
lightvector[0] = lightDirection.x;
lightvector[1] = lightDirection.y;
lightvector[2] = lightDirection.z;
lightvector[3] = lightDirection.w;
trace("Light normal will be: " + lightvector);
// fc15 = [light normal]
context3D.setProgramConstantsFromVector(
    Context3DProgramType.FRAGMENT, 15, lightvector);
// fc16 = ambient light
var lightambient:Vector.<Number> = new Vector.<Number>();
lightambient[0] = 0.2;
lightambient[1] = 0.2;
```

```

        lightambient[2] = 0.2;
        lightambient[3] = 0.2;
        context3D.setProgramConstantsFromVector(
            Context3DProgramType.FRAGMENT, 16, lightambient);
    }
}

```

What just happened?

The first shader, as seen in previous chapters, simply renders each mesh with a texture in full brightness. In order to give added depth to our models, the second shader, `shaderWithLight`, goes a bit deeper into AGAL. It takes into account each vertex normal (the direction that it is facing) and calculates the amount of light that each face in our mesh should receive.

In this way, depending on the angle of the light which is defined in the `lightvector` variable and uploaded to the video card as the `fc15` vertex constant register, each face will be lighter or darker depending on the angle it faces.

To brighten up the scene some ambient light is also added to each pixel that is output by our fragment shader. You could, for example, make the scene a little redder by increasing the first number in the `lightambient` variable.

This is the most basic possible dynamic lighting shader. In order to make the light truly dynamic, you could use `setProgramConstantsFromVector` to upload a new `lightvector` in each frame, so that the values stored in `fc15` change constantly. For now, this light source will never move and it is enough to define `fc15` once during `init` rather than during the render loop.

As a beginner's guide, this book cannot delve too deeply into super complex AGAL shaders, which are themselves worthy of an entire book. This should be enough to give you a head start in the creation of fancy shaders that use multiple lights, or more advanced effects such as specular shines, normal map textures for extra detail, multiple lights and more.

Time for action – defining new actor types and behaviors

It is finally time to instruct our game about all the meshes that our game needs. As we will be using our new actor pool system to spawn multiple copies of certain entities, we generally create one "master" game actor that is used as the definition of that particular kind of actor.

```

private function initData():void
{
    // create object pools
    particleSystem = new GameParticlesystem();
    enemies = new GameActorpool();
    playerBullets = new GameActorpool();
}

```

3... 2... 1... ACTION!

```
enemyBullets = new GameActorpool();
// create the camera entity
chaseCamera = new Stage3dEntity();
// the player spaceship and engine glow
initPlayerModel();
// non interactive terrain, asteroid field, sky
initTerrain();
// evil spaceships, sentry guns, fuel tanks
initEnemies();
// good and evil bullets
initBullets();
// various shootable asteroids
initAsteroids();
// space stations and slabs of "tech"
initSpaceStations();
// explosions, sparks, debris
initParticleModels();
}
```

What just happened?

We first create a few empty object pools to take advantage of our highly optimized and efficient reusable entity classes from the previous chapters. Next, the chase camera is created, which is simply a Stage3dEntity that never gets drawn because it does not have any meshes associated with it.

Finally, we are going to define numerous types of game entity; everything from the player's ship to all sorts of bad guys and level props, each with slightly different meshes, shaders, movement speeds, and AI behaviors. Some will be non-interactive and are there merely to look nice. Others will move and shoot and damage the player if hit.

For readability, these entity setups have been split into smaller functions. Take a look at how the different kinds of entity are defined by changing the object's properties. Note that each one is given a string name that is used later on to spawn multiple copies of this "master" clone source.

This means that most of the entities and particles defined below never appear in the game: they are used as the definition of a certain kind of game actor and during runtime, the object pools will clone and then respawn them as needed.

Time for action – initializing the terrain meshes

We will begin by setting up a few non-interactive decorative meshes. There is no need to check for collisions with these meshes as they are off the play area and are meant to simply look nice.

```
private function initTerrain():void
{
    trace("Parsing the sky...");
    sky = new Stage3dEntity(
        skyObjData,
        context3D,
        shaderProgram1,
        skyTexture);
    sky.depthTest = false;
    sky.depthTestMode = Context3DCompareMode.LESS;
    sky.cullingMode = Context3DTriangleFace.NONE;
    sky.depthDraw = false;
    sky.z = -10000.0;
    sky.scaleX = 20;
    sky.scaleY = 10;
    sky.scaleZ = 20;
    sky.rotationDegreesY = -90;
    sky.rotationDegreesX = -90;
    props.push(sky);
}
```

After creating a giant inside-out sphere for use as the "sky" background, we create a terrain mesh and add it to the "props" list as follows:

```
trace("Parsing the terrain...");
var terrain:Stage3dEntity =
    new Stage3dEntity(
        terrainObjData,
        context3D,
        shaderProgram1,
        terrainTexture, 0.5);
terrain.rotationDegreesZ = 90;
terrain.y = -50;
props.push(terrain);
```

3... 2... 1... ACTION!

Next, we do the same for several asteroid meshes.

```
trace("Parsing the asteroid field...");  
asteroids1 = new Stage3dEntity(  
    asteroidsObjData,  
    context3D,  
    shaderWithLight,  
    cratersTexture);  
asteroids1.rotationDegreesZ = 90;  
asteroids1.scaleXYZ = 200;  
asteroids1.y = 500;  
asteroids1.z = -1100;  
props.push(asteroids1);  
trace("Cloning the asteroid field...");  
// use the same mesh in multiple locations  
asteroids2 = asteroids1.clone();  
asteroids2.z = -1500;  
props.push(asteroids2);  
asteroids3 = asteroids1.clone();  
asteroids3.z = -1900;  
props.push(asteroids3);  
asteroids4 = asteroids1.clone();  
asteroids4.z = -1900;  
asteroids4.y = -500;  
props.push(asteroids4);
```

Finally, we create an actor definition for some rocky "islands" that float in space and are used to place sentry guns and fuel tanks upon, as well as the "finish line" which is a spinning wormhole effect.

```
actor = new GameActor(  
    islandData,  
    context3D,  
    shaderWithLight,  
    cratersTexture, 1, false);  
actor.shaderUsesNormals = true;  
actor.collides = false;  
actor.radius = 64;  
enemies.defineActor("island", actor);  
  
actor = new GameActor(  
    wormholeData,  
    context3D,  
    shaderProgram1,  
    particle1texture,
```

```
    0.1);
    actor.collides = false;
    actor.radius = 128;
    actor.blendSrc = Context3DBlendFactor.ONE;
    actor.blendDst = Context3DBlendFactor.ONE;
    actor.cullingMode = Context3DTriangleFace.NONE;
    actor.depthTestMode = Context3DCompareMode.LESS;
    actor.depthDraw = false;
    actor.depthTest = false;
    actor.rotVelocity = new Vector3D(0, 0, -120); // spinning
    enemies.defineActor("wormhole", actor);

}
```

What just happened?

The actors defined above are used to make the game world more attractive. They are set not to collide with the player, and thus never have to be checked for collisions during the render loop. The sky is set not to write to the z-buffer, so that it is always in the background and never obscures the other in-game objects. The wormhole "finish line" for our game is given a `rotVelocity` with a large value in the z-axis so that it spins, as well as special blend modes to be transparent.

Time for action – initializing the enemies

The enemies in the game include spaceships and other destroyable objects. The enemy cruisers should move towards the player and spin, and will shoot at the player.

```
private function initEnemies():void
{
    // create enemies
    actor = new GameActor(
        myObjData5,
        context3D,
        shaderWithLight,
        badguytexture,4);
    actor.collides = true;
    actor.radius = 2;
    actor.particles = particleSystem;
    actor.bullets = enemyBullets;
    actor.shootName = 'evilbolt';
    actor.shootDelay = 500;
    actor.shootRandomDelay = 1500;
    actor.shootDist = 250;
```

3... 2... 1... ACTION!

```
actor.shootVelocity = 35;
actor.shootAt = player;
actor.posVelocity = new Vector3D(0, 0, 20); // fly towards us
actor.rotVelocity = new Vector3D(0, 0, 40); // spinning
enemies.defineActor("enemyCruiser", actor);
```

The preceding actor definition sets up the common enemy spaceship. It will shoot at the player, spin, and move towards the camera and emit particles when shooting or getting destroyed. Next, we define a similar actor, but this time is a big boss that flies backwards, takes several direct hits to destroy, earn the player many points, and shoots a lot more often.

```
actor = new GameActor(
    bossData,
    context3D,
    shaderWithLight,
    badguytexture, 32); // scaled up
actor.collides = true;
actor.radius = 16;
actor.posVelocity = new Vector3D(0, 0.5, -20);
actor.particles = particleSystem;
actor.bullets = enemyBullets;
actor.shootName = 'evilbolt';
actor.shootDelay = 250;
actor.shootRandomDelay = 250;
actor.shootDist = 250;
actor.shootVelocity = 30; // bullets move slowish
actor.shootAt = player;
actor.points = 1111; // reward for victory
actor.health = 20; // #hits to destroy
enemies.defineActor("enemyBoss", actor);
```

Next, we define a bunch of sentry guns that don't move, but shoot at the player.

```
actor = new GameActor(
    sentrygun01_data,
    context3D,
    shaderWithLight,
    badguytexture, 4);
actor.collides = true;
actor.radius = 2;
actor.particles = particleSystem;
actor.bullets = enemyBullets;
actor.shootName = 'evilbolt';
actor.shootDelay = 500;
actor.shootRandomDelay = 1500;
```

```
actor.shootDist = 250;
actor.shootVelocity = 25;
actor.shootAt = player;
enemies.defineActor("sentrygun01", actor);
actor = new GameActor(
    sentrygun02_data,
    context3D,
    shaderWithLight,
    badguytexture,4);
actor.collides = true;
actor.radius = 2;
actor.particles = particleSystem;
actor.bullets = enemyBullets;
actor.shootName = 'evilbolt';
actor.shootDelay = 500;
actor.shootRandomDelay = 1500;
actor.shootDist = 250;
actor.shootVelocity = 25;
actor.shootAt = player;
enemies.defineActor("sentrygun02", actor);
actor = new GameActor(
    sentrygun03_data,
    context3D,
    shaderWithLight,
    badguytexture,4);
actor.collides = true;
actor.radius = 2;
actor.particles = particleSystem;
actor.bullets = enemyBullets;
actor.shootName = 'evilbolt';
actor.shootDelay = 500;
actor.shootRandomDelay = 1500;
actor.shootDist = 250;
actor.shootVelocity = 25;
actor.shootAt = player;
enemies.defineActor("sentrygun03", actor);
actor = new GameActor(
    sentrygun04_data,
    context3D,
    shaderWithLight,
    badguytexture,4);
actor.collides = true;
actor.radius = 2;
actor.particles = particleSystem;
```

3... 2... 1... ACTION!

```
actor.bullets = enemyBullets;
actor.shootName = 'evilbolt';
actor.shootDelay = 500;
actor.shootRandomDelay = 1500;
actor.shootDist = 250;
actor.shootVelocity = 25;
actor.shootAt = player;
enemies.defineActor("sentrygun04", actor);
```

Finally, we define the actor types for a few destroyable "fuel tanks" that are merely there to give the player some targets to destroy that don't fight back.

```
actor = new GameActor(
    fueltank01_data,
    context3D,
    shaderWithLight,
    badguytexture,4);
actor.collides = true;
actor.radius = 2;
enemies.defineActor("fueltank01", actor);
actor = new GameActor(
    fueltank02_data,
    context3D,
    shaderWithLight,
    badguytexture,4);
actor.collides = true;
actor.radius = 2;
enemies.defineActor("fueltank02", actor);
}
```

What just happened?

The various enemies in our game are defined above. By setting the `collides` property to true, these actors will be checked for collisions with the player and the player's bullets. If they get hit, they will be destroyed and may damage the player's ship if bumped into. The `radius` properties ensure that the collisions are checked within the properly sized bounds and also help to determine how far away from the camera each actor must be before it is deemed too far away to bother rendering, which increases performance.

The enemy cruisers are given additional AI properties which make them shoot at the player if they are close enough. By defining the particles emitted when they are destroyed and the actor name of the bullet projectile that they will shoot, we have created a moving, collidable, autonomous enemy type that can be destroyed, will earn points for the player when shot, and are animated to fly forward while spinning.

Time for action – initializing the bullets

We only need two types of bullet to be defined for this simple demo game. The first kind of bullet is going to be the bright blue bolt of energy that the player shoots at enemies. The second is a red energy bolt fired from enemies at the player.

```
private function initBullets():void
{
    actor = new GameActor(
        bulletData,
        context3D,
        shaderProgram1,
        particle1texture);
    actor.collides = true;
    actor.radius = 1;
    actor.ageMax = 5000;
    actor.particles = particleSystem;
    actor.spawnWhenNoHealth = "sparks";
    actor.spawnWhenMaxAge = "sparks";
    actor.spawnWhenCreated = "sparks";
    actor.blendSrc = Context3DBlendFactor.ONE;
    actor.blendDst = Context3DBlendFactor.ONE;
    actor.cullingMode = Context3DTriangleFace.NONE;
    actor.depthTestMode = Context3DCompareMode.LESS;
    actor.depthDraw = false;
    actor.depthTest = false;
    actor.posVelocity = new Vector3D(0, 0, -100);
    actor.runWhenMaxAge = resetCombo;
    actor.shaderUsesNormals = false;
    playerBullets.defineActor("bolt", actor);

    actor = new GameActor(
        bulletData,
        context3D,
        shaderProgram1,
        particle2texture);
    actor.collides = true;
    actor.radius = 1;
    actor.ageMax = 5000;
    actor.particles = particleSystem;
    actor.spawnWhenNoHealth = "sparks";
    actor.spawnWhenMaxAge = "sparks";
    actor.spawnWhenCreated = "sparks";
    actor.blendSrc = Context3DBlendFactor.ONE;
```

3... 2... 1... ACTION!

```
actor.blendDst = Context3DBlendFactor.ONE;
actor.cullingMode = Context3DTriangleFace.NONE;
actor.depthTestMode = Context3DCompareMode.LESS;
actor.depthDraw = false;
actor.depthTest = false;
actor.posVelocity = new Vector3D(0, 0, 25);
actor.shaderUsesNormals = false;
enemies.defineActor("evilbolt", actor);
}
```

What just happened?

The bullet actors are given special blendmodes that make them transparent and lighten the scene while not drawing to the z-buffer. They are given a maximum age of five seconds, so that, if they miss their target they won't fly on forever, consuming memory and slowing performance. Each is given a movement velocity so they shoot ahead of their owner, and explode into various particles when they hit their mark.

Time for action – initializing the asteroids

In addition to various enemy ships that move and shoot at the player, let's define a few simple asteroids that can be destroyed or bumped into to give the player more to aim at and avoid.

```
private function initAsteroids():void
{
    // need to hit them 3 times to destroy

    actor = new GameActor(
        asteroid_00_data,
        context3D,
        shaderWithLight,
        cratersTexture, 0.5);
    actor.collides = true;
    actor.radius = 6;
    actor.health = 3;
    actor.spawnWhenNoHealth = 'debris01,debris02,debris03';
    actor.particles = particleSystem;
    enemies.defineActor("asteroid_00", actor);

    actor = new GameActor(
        asteroid_01_data,
        context3D,
        shaderWithLight,
```

```
        cratersTexture, 0.5);
    actor.collides = true;
    actor.radius = 6;
    actor.health = 3;
    actor.spawnWhenNoHealth = 'debris01,debris02,debris03';
    actor.particles = particleSystem;
    enemies.defineActor("asteroid_01", actor);

    actor = new GameActor(
        asteroid_02_data,
        context3D,
        shaderWithLight,
        cratersTexture, 0.5);
    actor.collides = true;
    actor.radius = 6;
    actor.health = 3;
    actor.spawnWhenNoHealth = 'debris01,debris02,debris03';
    actor.particles = particleSystem;
    enemies.defineActor("asteroid_02", actor);
}
```

What just happened?

We defined several similar asteroid actors that will be spawned by our level parser. These asteroids will be liberally scattered about in our game world.

Time for action – initializing the space stations

Much like the preceding terrain geometry, these space stations (and simple "slabs" of tech) are meant to decorate the background in our game.

```
private function initSpaceStations():void
{
    actor = new GameActor(
        slab01_data,context3D,shaderWithLight,techtexture,1,false);
    actor.collides = false;
    actor.radius = 64;
    enemies.defineActor("slab01", actor);
```

3... 2... 1... ACTION!

Repeat the preceding code for each of the four "slabs" and ten "space station" props. They all have the same parameters and the only difference is the name and the mesh data used. The entire block of code can be found in the .zip download of the source code that comes with the book.

```
// and so on for slab02 to slab04 and station01 to station09...
```

```
actor = new GameActor(  
    station10_data, context3D, shaderWithLight, techtexture, 1, false);  
actor.collides = false;  
actor.radius = 64;  
enemies.defineActor("station10", actor);  
}
```

What just happened?

These space stations will be spawned off to the side of our play area and thus will not be collided with. They are simple, non-interactive meshes meant to do nothing more than add something interesting to look at as we speed towards our goal.

Time for action – initializing the particle models

The final step in the inits for our art assets in-game is to set up the explosions and other special effects.

```
private function initParticleModels():void  
{  
    trace("Creating particle system models...");  
    // define the types of particles  
    particleSystem.defineParticle("explosion",  
        new Stage3dParticle(explosion1_data, context3D,  
            puffTexture, explosionfat_data));  
    particleSystem.defineParticle("bluebolt",  
        new Stage3dParticle(explosion1_data, context3D,  
            particle1texture, explosion2_data));  
    particleSystem.defineParticle("greenpuff",  
        new Stage3dParticle(explosion1_data, context3D,  
            particle2texture, explosion2_data));  
    particleSystem.defineParticle("ringfire",  
        new Stage3dParticle(explosion1_data, context3D,  
            particle3texture, explosion2_data));  
    particleSystem.defineParticle("sparks",  
        new Stage3dParticle(explosion1_data, context3D,  
            particle4texture, explosion2_data));  
}
```

What just happened?

These particles are defined in the same way as our game actors, but they use the `Stage3dParticle` class which is optimized for transparent, glowing, efficiently simulated particle systems. When various game actors are destroyed, they will use our particle system manager to respawn an explosion or sparks as appropriate to their location.

That is it for our art data setup. All models, textures, particles, sounds, and GUI images are good to go. The next step is to build a level out of all these great looking assets.

Time for action – creating the game level

After all the different types of actor are defined, from spinning asteroids to bullets, sentry guns and decorations, we use our new `GameLevelParser` to spawn enemies in the proper locations according to our level map image.

```
private function spawnTheLevel():void
{
    // create a level using
    // our map images as the blueprint
    trace("Spawning the game level parser...");
    var level:GameLevelParser = new GameLevelParser();

    // how far ahead of the player is the first actor
    var levelStartDistance:Number = 350;

    // list of entities that we want spawned in a spiral map
    var enemyNames:Array =
        ["enemyCruiser", "asteroid_00", "asteroid_01", "asteroid_02"];

    // spiral enemy formation - works great
    level.spawnActors(
        levelKey.bitmapData,
        levelData.bitmapData,
        enemyNames,
        enemies,
        0, 16,
        player.z - levelData.height * 16 - levelStartDistance,
        4, 16, 0, 8);

    // list of entities that we want spawned in a flat map
    var rockNames:Array =
        [null,null,null,null, // skip the types used above
        "sentrygun01", "sentrygun02", "sentrygun03", "sentrygun04",
        "fueltank01", "fueltank02", "enemyBoss", "island",
```

3... 2... 1... ACTION!

```
"slab01", "slab02", "slab03", "slab04",
"station01", "station02", "station03", "station04",
"station05", "station06", "station07", "station08",
"station09", "station10", "wormhole", "astroTunnel"] ;

// flat enemy formation
var gameLevelLength:Number;
gameLevelLength = level.spawnActors(
    levelKey.bitmapData,
    levelData.bitmapData,
    rockNames,
    enemies,
    -8, 0,
    player.z - levelData.height * 8 - levelStartDistance,
    2, 8, 0, 0);

// now that we know how long the map is
// remember what z coord is the "finish line"
playerMinz = player.z - gameLevelLength - levelStartDistance;
}
```

What just happened?

We create a GameLevelParser object and use it to place actors in locations defined by our tiny map image blueprint. In this particular example, we parse the same map image in two different ways, using two arrays of names. One is set to spawn actors in a "spiral" formation. All other level actors are placed on a flat plane.

The map image and key image you use to lay out your game level might look something like this:



The array of names that we build corresponds to the order of the map key image pixels, so that the first entry in the array of names that we pass to our level parser is the actor type, that matches the RGB values of the first pixel in our map key image.

In this way, for example, if we scatter blue pixels all over our map image and then include a blue pixel in in the fifth column of our "key" image, the fifth actor name in our array is what will be spawned there.

Time for action – upgrading the render loop

Our game engine has initialized Stage3D, our art has been processed, our level has been spawned, and now we are ready to upgrade the function that does the rendering of our scenes!

```
private function renderScene():void
{
    scenePolycount = 0;

    viewmatrix.identity();
    // look at the player
    viewmatrix.append(chaseCamera.transform);
    viewmatrix.invert();
    // tilt down a little
    viewmatrix.appendRotation(15, Vector3D.X_AXIS);
    // if mouselook is on:
    viewmatrix.appendRotation(
        gameinput.cameraAngleX, Vector3D.X_AXIS);
    viewmatrix.appendRotation(
        gameinput.cameraAngleY, Vector3D.Y_AXIS);
    viewmatrix.appendRotation(
        gameinput.cameraAngleZ, Vector3D.Z_AXIS);
    // if we are shaking the screen (got hit)
    viewmatrix.appendRotation(
        screenShakeCameraAngle, Vector3D.X_AXIS);
```

The preceding code sets up the camera angle. Next, we render the player mesh and all enemies as follows:

```
player.render(viewmatrix, projectionmatrix);
scenePolycount += player.polycount;

// loop through all known entities and render them
for each (entity in props)
{
    entity.render(viewmatrix, projectionmatrix);
    scenePolycount += entity.polycount;
}
```

3... 2... 1... ACTION!

In order to increase the framerate of our game, we hide enemies that are far away or behind us. These distances are scaled by the entity's radius, so that big meshes are visible from farther away to prevent "pop-in".

```
enemies.hideDistant(player.position, 150, 2);
playerBullets.hideDistant(player.position, 150, 2);
enemyBullets.hideDistant(player.position, 150, 2);

enemies.render(viewmatrix, projectionmatrix);
scenePolycount += enemies.totalpolycount;

playerBullets.render(viewmatrix, projectionmatrix);
scenePolycount += playerBullets.totalpolycount;

enemyBullets.render(viewmatrix, projectionmatrix);
scenePolycount += enemyBullets.totalpolycount;

particleSystem.render(viewmatrix, projectionmatrix);
scenePolycount += particleSystem.totalpolycount;
```

Finally, to provide feedback to the player to ensure they are aware of when they take damage, we shake the screen when they get hit by an enemy bullet.

```
if (screenShakes > 0)
{
    screenShakes--;
    if (screenShakes > 0)
    {
        if (screenShakes % 2) // alternate
        {
            this.y += screenShakes / 2;
        }
        else
        {
            this.y -= screenShakes / 2;
        }
        screenShakeCameraAngle = this.y / 2;
    }
    else
    {
        // we are done shaking: reset
        this.x = 0;
        this.y = 0;
        screenShakeCameraAngle = 0;
    }
}
```

What just happened?

We first need to ensure that the chase camera is in the correct location, slightly above and behind the player. We account for any mouse look angles, and arrive at the proper location and angle for the chase camera.

Once we calculate the proper view transform to send to Stage3D, we simply iterate through our actor pools and draw any actors that are visible. By using our fancy new actor and particle pools from the previous chapters, all we need to do is to instruct the entire pool to render itself. After each render step, we take into account the number of polygons drawn for use in our stats display.

We also do some tricks to ensure that our framerate stays at a silky smooth 60fps. We cull any actors that are too far away to bother drawing from the scene before rendering. For our purposes, a simple "don't draw if it is too far away" optimization solution suits our needs wonderfully.



This optimization technique is often called **PVS (potentially visible set)** and could be extended to take into account visibility blockers, the view frustum, or even to use complex occlusion techniques such as BSP trees, Octrees, or portals. These terms are great for doing some googling and are left as a challenging side-quest for developers interested in further optimizations of massively complex scenes.

We are nearing the end of our game engine upgrades! Next up, we want to define some gameplay-specific events, so that things happen when they should.

Defining gameplay-specific events

Now that the graphics have been initialized and all the art assets embedded and processed to create a nice looking level filled with objects, we need to take into consideration the various states of the game and the behaviors of these game entities. The next few sections will deal with all this gameplay-related code.

Time for action – tracking game events

The `gamestart` function will instruct our game that the player is ready to start blasting away.

```
private function gamestart():void
{
    if (gameState == STATE_TITLESCREEN)
    {
```

3... 2... 1... ACTION!

```
// did they click the book link?  
if (gameinput.mouseClickX < 140)  
{  
    try  
    {  
        navigateToURL(new URLRequest(  
            "https://www.packtpub.com/" +  
            "adobe-flash11-stage3d-molehill-game" +  
            "-programming-beginners-guide/book"));  
    }  
    catch (error:Error)  
    {  
        trace('Link denied by browser security.');//  
    }  
    return;  
}  
  
trace('GAME START!');
```

After checking to ensure that the game is ready to begin and optionally opening a window to the book page, we stop any previously playing music, start the in-game soundtrack, and set up the GUI to display the "mission objectives" screen for seven seconds.

```
if (musicChannel) musicChannel.stop();  
// start the background music  
musicChannel = introSound.play();  
  
// clear the title screen  
if (contains(titleScreen))  
    removeChild(titleScreen);  
if (contains(titlescreenTf))  
    removeChild(titlescreenTf);  
  
// the intro "cinematic"  
addChild(introOverlay);  
// it will be removed in about 7s  
introOverlayEndTime = 7000;
```

Next, we reset the game state and player location, along with game variables such as number of consecutive hits, score, and health.

```
gameState = STATE_PLAYING;  
score = 0;  
combo = 0;
```

```
comboBest = 0;
player.health = 3;
healthTf.text = healthText_3;
player.x = 0;
player.y = 8; // slightly above the "ground"
player.z = 0;
player.active = true;
player.visible = true;
player.collides = true;
nextShootTime = 0;
invulnerabilityTimeLeft = 0;
player.spawnConstantly = '';
player.spawnConstantlyDelay = 0;
player.updateTransformFromValues();
// reset any mouse look camera angles
gameinput.cameraAngleX = 0;
gameinput.cameraAngleY = 0;
```

If this is not the very first game played, we want to clear out all the game actors that might still exist from the previous game. We then spawn the level, reset any timers, and clear any temporary variables from memory so that the game runs smoothly from the start.

```
enemies.destroyAll();
enemyBullets.destroyAll();
playerBullets.destroyAll();

// create all enemies, etc.
spawnTheLevel();

// reset the game timer
gametimer.gameElapsedTime = 0;
gametimer.gameStartTime =
gametimer.lastFrameTime =
gametimer.currentFrameTime =
getTimer();

// now is a good time to release temporary
// variables from memory
System.gc();

}
```

What just happened?

The gamestart function is run at the beginning of gameplay. It is called by our input manager when the mouse is clicked.

As part of the demo that comes with the book, a check is made to see if the click occurred near the book icon on the title screen; if so, a new web page is opened. In your game, you may want to include a link to a sponsor or to your own website instead.

Next, the title screen is removed and replaced by the "intro" GUI screen where the player is given a mission. We reset the variables associated with where the player is in our game world, reset the score to zero, and update the game state so that from now on everything will be moving. Any actors from a previous game are destroyed (and are set to be ready for respawning in our actor pools) and the level is spawned. The timer is reset and everything is ready for a new game. We instruct Flash that now is a good time to clear out any garbage in RAM that is no longer being used so that everything starts fresh.

Time for action – handling game over

Every game needs a beginning, middle, and end or in other words a title screen, a render loop, and a gameover. Let's get the beginning and end out of the way so that the remainder of this chapter can focus on what happens in between.

```
private function gameover(beatTheGame:Boolean = false):void
{
    if (gameState != STATE_TITLESCREEN)
    {
        trace('GAME OVER!');
        gameState = STATE_TITLESCREEN;
        if (contains(introOverlay))
            removeChild(introOverlay);
        addChild(titleScreen);
        if (beatTheGame)
        {
            titlescreenTf.text = "VICTORY!\n"
                + "You reached the wormhole\nin " + Math.round(
                    gametimer.gameElapsed / 1000) + " seconds\n"
                + "Your score: " + score + "\n"
                + "Highest combo: " + comboBest + "x\n";
        }
        else // player was destroyed
        {
            titlescreenTf.text = "GAME OVER\n"
                + "You were destroyed\nafter " + Math.round(
                    gametimer.gameElapsed / 1000) + " seconds\n"
                + "Your score: " + score + "\n"
                + "Highest combo: " + comboBest + "x\n";
        }
    }
}
```

```

    // turn off the engines
    engineGlow.scaleXYZ = 0.00001;

    // make the gameover text visible
    addChild(titleScreenTf);
}

// release some temporary variables from memory
System.gc();

}

```

What just happened?

This function is called because the player was destroyed or reached the finish line and beat the game. The game state is changed to the title screen mode again. We put the title screen back onto the stage, update the text message that reports to the player how well they did, and make our game ready to be played once again.



Time for action – updating the score display

During the gameplay, the GUI needs to be updated to reflect changes in score and our combo state.

```
private var combocomment:String = '';
private function updateScore():void
{
    if (combo < 10) combocomment = combo + "x combo";
    else if (combo < 20) combocomment =
        combo + "x combo\nNot bad!";
    else if (combo < 30) combocomment =
        combo + "x combo\nGreat!";
    else if (combo < 40) combocomment =
        combo + "x combo\nSkilled!";
    else if (combo < 50) combocomment =
        combo + "x combo\nIncredible!";
    else if (combo < 60) combocomment =
        combo + "x combo\nAmazing!";
    else if (combo < 70) combocomment =
        combo + "x combo\nUnbelievable!";
    else if (combo < 80) combocomment =
        combo + "x combo\nInsane!";
    else if (combo < 90) combocomment =
        combo + "x combo\nFantastic!";
    else if (combo < 100) combocomment =
        combo + "x combo\nHeroic!";
    else combocomment = combo + "x combo\nLegendary!";
    if (combo > comboBest) comboBest = combo;

    // padded with zeroes
    if (score < 10) scoreTf.text =
        'Score: 00000' + score + "\n" + combocomment;
    else if (score < 100) scoreTf.text =
        'Score: 0000' + score + "\n" + combocomment;
    else if (score < 1000) scoreTf.text =
        'Score: 000' + score + "\n" + combocomment;
    else if (score < 10000) scoreTf.text =
        'Score: 00' + score + "\n" + combocomment;
    else if (score < 100000) scoreTf.text =
        'Score: 0' + score + "\n" + combocomment;
    else scoreTf.text = 'Score: ' + score + "\n" + combocomment;
}
```

What just happened?

This function simply outputs a nicely formatted score on the screen that is padded with zeroes. Now that we are tracking the player's "combo" count of successful consecutive hits, all we have to do is upgrade our existing updateScore function to account for it. Just for fun, we can include a bit of positive encouragement with a lighthearted exclamatory adjective.

Time for action – updating the FPS display

Our game will be keeping track of various stats such as the frames per second (FPS), as well as how much memory is being used and more. As the performance of your game engine is impressive, let's show off with some stats so that any screenshots players take report the good news.

```
private function updateFPS():void
{
    // update the FPS display
    fpsTicks++;
    var now:uint = getTimer();
    var delta:uint = now - fpsLast;
    // only update the display once a second
    if (delta >= 1000)
    {
        // also track ram usage
        var mem:Number = Number((System.totalMemory *
            0.000000954).toFixed(1));
        var fps:Number = fpsTicks / delta * 1000;
        fpsTf.text = fps.toFixed(1) +
            " fps"
            + " (" + scenePolycount + " polies)\n"
            + "Memory used: " + mem + " MB";
        //+ enemies.totalrendered + " enemies\n"
        //+ (enemyBullets.totalrendered
        //+ playerBullets.totalrendered) + " bullets\n"
        //+ "[step=" + (profilingEnd - profilingStart) + "ms]"
        ;
        fpsTicks = 0;
        fpsLast = now;
    }
}
```

What just happened?

The GUI is updated to display the current framerate, the number of polygons rendered in the previous frame, and how many megabytes of RAM have been consumed. When debugging your game, you may want to include extra information here, such as the location of the player, how many particles are active or how many ms the `gameStep` function took.

Now that we are done with inits, the GUI and handling the start and end of each game, all the functionality remaining relates to what is going on during the time the game is actually being played.

Time for action – handling collision events

There are some additional game event triggers that we want to implement. These are in response to collisions (usually between bullets and spaceships). When a bullet does indeed hit an enemy (or us), we need to instruct the game what to do.

```
private function resetCombo() :void
{
    trace("Player bullet missed: resetting combo.");
    combo = 0;
}
```

We want to call a routine whenever a bullet is removed from the scene because it missed and flew off into space without ever colliding with anything. This function will reset our combo counter of successive hits.

```
private function hitAnEnemy(
    culprit:GameActor, victim:GameActor) :void
{
    trace(culprit.name
        + " at " + culprit.posString()
        + " hitAnEnemy " + victim.name
        + " at " + victim.posString());
    blastSound.play();
    victim.health--;
    if (victim.health <= 0)
    {
        particleSystem.spawn("explosion",
            victim.transform);
        score += victim.points;
        // ensure that die(), sounds, particles are triggered:
        victim.step(0);
    }
    particleSystem.spawn("sparks",
        culprit.transform);
    culprit.die();
    combo++;
}
```

If the player manages to successfully hit an enemy, we spawn an explosion, play a sound, award the player some points, and increment their combo meter. Conversely, when the player is hit, we reduce the player's health and check to see if they are dead.

```
private function playerGotHit(  
    culprit:GameActor, victim:GameActor):void  
{  
    trace("Player got hit!");  
    screenShakes = 30;  
    particleSystem.spawn("explosion",  
        victim.transform);  
    particleSystem.spawn("sparks",  
        culprit.transform);  
    culprit.die();  
    explodeSound.play();  
    player.health--;  
    trace("Player health = " + player.health);  
    invulnerabilityTimeLeft =  
        invulnerabilityMsWhenHit;  
    if (player.health == 3)  
    {  
        healthTf.text = healthText_3;  
    }  
    if (player.health == 2)  
    {  
        healthTf.text = healthText_2;  
    }  
    if (player.health == 1)  
    {  
        trace("Player is almost dead!");  
        player.spawnConstantly = 'explosion';  
        player.spawnConstantlyDelay = 100;  
        player.spawnConstantlyNext = 0;  
        healthTf.text = healthText_1;  
    }  
    if (player.health <= 0)  
    {  
        trace("Player's health is zero!");  
        healthTf.text = healthText_0;  
        gameover();  
    }  
}
```

What just happened?

As the collision detection routines run callback functions whenever something is hit, we can track how many hits in a row are made and use that to report on the player's accuracy by updating a "combo" meter.

When a collision happens, we trigger an explosion sound and particle effect, update the score and health of the "victim", and destroy it if it has run out of health. If the victim is the player, we start shaking the screen to provide additional feedback that something bad happened. In order to avoid instant-deaths when there are several enemy bullets near to each other, once hit, the player is temporarily invulnerable for a short period of time to allow them to take evasive action.

Time for action – handling the player input

We need to move the player according to their key presses. In previous chapters, we simply changed the player's x, y, and z locations depending on what keys were being held down. This new version has been upgraded to account for movement in any direction.

```
private function handlePlayerInput(frameMs:uint) :void
{
    // handle player input
    var moveAmount:Number = moveSpeed * frameMs;
```

In our game, we are always flying forward while moving up/down/left/right to dodge. The commented-out code in the following example shows a simple way to slide around which works for our particular game but assumes you are always facing down the z-axis:

```
// if (gameinput.pressing.up) player.y += moveAmount;
// if (gameinput.pressing.down) player.y -= moveAmount;
// if (gameinput.pressing.left) player.x -= moveAmount;
// if (gameinput.pressing.right) player.x += moveAmount;
```

An improved method for doing the above, that does the same thing, but would also work if we were facing in other directions, because the motion is relative to our angles, should be used instead.

```
if (gameinput.pressing.up)
    player.moveUp(moveAmount);
if (gameinput.pressing.down)
    player.moveDown(moveAmount);
if (gameinput.pressing.left)
    player.moveLeft(moveAmount);
if (gameinput.pressing.right)
    player.moveRight(moveAmount);
```

Here are three additional examples of handling the movement. The first is the simplistic way to implement FPS-style 360 degree turning. It is left here as an example of what not to do because, although it looks correct, there are certain angles that will "flip" the axes. If you google the term "gimbal lock", you will be able to read more about this effect.

The second example is a working alternative for this which would be perfect for a six-degrees-of-freedom system, which would be perfect for a flying game where you can turn to fly in any direction.

The final commented-out example shows a good way to handle FPS shooters, where the keyboard controls move you forward and back and let the user "strafe" (or slide from side to side) while actual viewing angles would be controlled by the mouse.

```
// EXAMPLE: turning the ship via euler angles
// will break due to gimbal lock at 180...-180
// do not rotate things using this technique:
// if (gameinput.pressing.left)
//   player.rotationDegreesY += moveAmount*4;
// if (gameinput.pressing.right)
//   player.rotationDegreesY -= moveAmount*4;
// if (gameinput.pressing.up)
//   player.rotationDegreesX -= moveAmount*4;
// if (gameinput.pressing.down)
//   player.rotationDegreesX += moveAmount*4;

// EXAMPLE: FPS style 6DOF rotation
// avoids gimbal lock by using quaternion rotation
// which is a way of rotating using a single axis
// if (gameinput.pressing.left)
//   player.transform.prependRotation(moveAmount * 4,
//   Vector3D.Y_AXIS);
// if (gameinput.pressing.right)
//   player.transform.prependRotation(-moveAmount * 4,
//   Vector3D.Y_AXIS);
// if (gameinput.pressing.up)
//   player.transform.prependRotation(-moveAmount * 4,
//   Vector3D.X_AXIS);
// if (gameinput.pressing.down)
//   player.transform.prependRotation(moveAmount * 4,
//   Vector3D.X_AXIS);

// EXAMPLE: slide forward and backward
// and "strafe" sliding side to side
// good for walking such as in an FPS game
// if (gameinput.pressing.up)
```

3... 2... 1... ACTION!

```
// player.moveForward(moveAmount);  
// if (gameinput.pressing.down)  
//     player.moveBackward(moveAmount);  
// if (gameinput.pressing.strafeLeft)  
//     player.moveLeft(moveAmount);  
// if (gameinput.pressing.strafeRight)  
//     player.moveRight(moveAmount);
```

The preceding examples are included simply to help you achieve whatever kind of movement system you desire. With luck, they will help you save some time. To continue with the shoot-em-up movement style that our example game needs, we next "push" the user's spaceship forward based on how much time has elapsed. We check to see if the fire button has been pressed and handle that event appropriately.

```
// keep moving forward: in our flying game  
// we constantly move and use arrow keys to dodge  
player.moveForward(flySpeed);  
  
if (gameinput.pressing.fire)  
{  
    if (gametimer.gameElapsedTime >= nextShootTime)  
    {  
        //trace("Fire!");  
        nextShootTime =  
            gametimer.gameElapsedTime + shootDelay;  
  
        // shoot a bullet  
        actor = playerBullets.spawn("bolt", player.transform);  
        actor.updateValuesFromTransform();  
        actor.rotationDegreesY = 90;  
        // we don't want to be able to shoot ourselves  
        actor.owner = player;  
        gunSound.play();  
    }  
}
```

Finally, we want to force the player to stay within bounds. The player can dodge bullets, but we don't want them flying far away from the action or going past the finish line.

```
if ((playerMaxx != 0) && (player.x > playerMaxx))  
    player.x = playerMaxx;  
if ((playerMaxy != 0) && (player.y > playerMaxy))  
    player.y = playerMaxy;  
if ((playerMaxz != 0) && (player.z > playerMaxz))  
    player.z = playerMaxz;
```

```
if ((playerMinx != 0) && (player.x < playerMinx))
    player.x = playerMinx;
if ((playerMiny != 0) && (player.y < playerMiny))
    player.y = playerMiny;
if ((playerMinz != 0) && (player.z < playerMinz))
{
    player.z = playerMinz;
    if (levelCompletePlayerMinz)
    {
        // we got to the "finish line"
        trace('LEVEL COMPLETE!');
        gameover(true)
    }
}
}
```

What just happened?

We first determine how much time has passed and use that to calculate how far the player should move in this particular frame.

By moving forward according to the angle in which the player's ship is pointing, you could change the game to allow for free 6DOF (six degrees of freedom) movement in any direction, as in a flight simulator.

For our shoot-em-up game, we want the ship to constantly move forward and the arrow keys to dodge left, right, up, and down rather than allowing the player to come to a full stop.

You could, conversely, make the game more like a first-person-shooter and make the player run and strafe as in Call of Duty, Quake, or Doom. Examples are included in the comments to help you achieve these goals.

Next, we optionally fire a bullet, trigger a gun sound, and create a muzzle flash particle system if the player is holding down the fire button (and if enough time has passed since the previous bullet was fired).

Finally, we force the player to stay within the boundaries of our game level and check to see if the player has crossed the "finish line". In a game with more than one level, this would be where you load the next scene. In this example game, the player has reached the wormhole and escapes to safety, triggering a game over.

Time for action – upgrading the gameStep function

This function is the workhorse of our game. During gameplay, it is run once per frame.

```
private var profilingStart:uint = 0;
private var profilingEnd:uint = 0;

private function gameStep(frameMs:uint):void
{
    // time how long this function takes
    profilingStart = getTimer();
```

As the step function may be where the framerate is most affected, the timer is queried to allow for profiling. By measuring the number of milliseconds that each frame's step function takes, you can determine if a new feature is slowing things down too much.

```
// if we are at the title screen, don't move the player
if (gameState == STATE_PLAYING)
{
    handlePlayerInput(frameMs);
    // intro "cinematic" pans the camera for a few sec
    if (gametimer.gameElapsedTime < introOverlayEndTime)
    {
        screenShakeCameraAngle = 45 * (1 -
            (gametimer.gameElapsedTime / introOverlayEndTime));
    }
}
```

The chase camera should follow behind the player no matter what direction it is facing.

```
chaseCamera.x = player.x;
chaseCamera.y = player.y;
chaseCamera.z = player.z;
chaseCamera.rotationDegreesY = player.rotationDegreesY;
chaseCamera.rotationDegreesX = chaseCameraXangle;
chaseCamera.moveBackward(chaseCameraDistance);
chaseCamera.moveUp(chaseCameraHeight);

// during the title screen, add some camera wobble
if (gameState != STATE_PLAYING)
{
    var wobbleSize:Number = 8;
    var wobbleMs:Number = 1000;
    chaseCamera.moveUp(
        Math.cos(gametimer.gameElapsedTime / wobbleMs)
```

```
    / Math.PI * wobbleSize);
chaseCamera.moveLeft(
    Math.sin(gametimer.gameElapsedTime / wobbleMs)
    / Math.PI * wobbleSize);
chaseCamera.moveForward(
    Math.sin(-1 * gametimer.gameElapsedTime / wobbleMs)
    / (Math.PI * wobbleSize * 5) - (wobbleSize / 2));

// force some swivelling as well
gameinput.cameraAngleX =
    Math.sin(gametimer.gameElapsedTime / wobbleMs)
    / Math.PI * 60;
gameinput.cameraAngleY =
    Math.sin(gametimer.gameElapsedTime / wobbleMs / 3)
    / Math.PI * 90;
}
```

After adding a little camera "wobble" when sitting at the title screen, we animate any particles and the distance asteroid fields.

```
// animate the asteroids in the background
asteroids1.rotationDegreesX += asteroidRotationSpeed * frameMs;
asteroids2.rotationDegreesX -= asteroidRotationSpeed * frameMs;
asteroids3.rotationDegreesX += asteroidRotationSpeed * frameMs;
asteroids4.rotationDegreesX -= asteroidRotationSpeed * frameMs;

// advance all particles based on time
particleSystem.step(frameMs);

// don't process any more if we are at the title screen
if (gameState != STATE_PLAYING) return;
```

The preceding code is run whether or not we are sitting at the title screen, or actively playing the game and provides some minimal animation even when the game is effectively paused. The code that follows is only run when the game state is not set to the title screen "paused" state. In this way, enemies and the player don't move when we are not actually playing.

```
// animate the engine glow - spin fast and pulsate slowly
engineGlow.rotationDegreesZ += 10 * frameMs;
engineGlow.scaleXYZ =
    Math.cos(gametimer.gameElapsedTime / 66) / 20 + 0.75;

playerBullets.step(frameMs, enemies.colliding, hitAnEnemy);

// when the player gets damaged, they become
```

3... 2... 1... ACTION!

```
// invulnerable for a short period of time
if (invulnerabilityTimeLeft > 0)
{
    invulnerabilityTimeLeft -= frameMs;
    if (invulnerabilityTimeLeft <= 0)
    {
        trace("Invulnerability wore off.");
    }
}

// check for collisions with the player
// between enemy ships and player ship
// unless we are invulnerable
if (invulnerabilityTimeLeft <= 0)
{
    // step enemy bullets and check for collisions
    enemyBullets.step(frameMs, player.colliding, playerGotHit);

    // step enemies and check to see if we collided with one
    enemies.step(frameMs, player.colliding, playerGotHit);

}
else
{
    // player is invulnerable:
    // step bullets but don't check collisions
    enemyBullets.step(frameMs);
    // step enemies but don't check collisions
    enemies.step(frameMs);
}

// allow the player to update things like particles
player.step(frameMs);

// time how long this function takes
profilingEnd = getTimer();

}
```

What just happened?

Much like in our render function, the gameStep function iterates through all known, active game actors and advances the simulation according to how much time has passed since the previous frame. We also check for collisions when appropriate and trigger the preceding functions when bullets and enemies hit something.

Checking for too many collisions between too many active actors can really slow things down. If you need to be checking for millions of potential collisions, more advanced heuristics for optimization of collision detection should be created.

For example, you could split the world into discrete "zones" and only check bullets and entities that are in the same zone. For now, going through the entire list of actors and checking to see if they are touching one another works fine as we are only checking the player's bullets to enemies and enemies to the player.

As enemy bullets are not allowed to hit their allies, we don't need to check as many possible collisions. It is for this reason that player bullets are kept in a separate actor pool from enemy bullets.

Finally, the player is also stepped (animated and updated) and the profiling timer is stopped. This profiling timer value will be reported in the debug log and can be used to measure exactly how much time the `gameStep` function takes.

Time for action – upgrading the heartbeat function

As in previous chapters, a heartbeat function is called in every few seconds.

```
// for efficiency, this function only runs occasionally
// ideal for calculations that don't need to be run every frame
// such as pathfinding, complex AI, streaming downloads, etc.
private function heartbeat():void
{
    trace('heartbeat at ' + gametimer.gameElapsedTime + 'ms');
    trace('player pos ' + player.posString());
    trace('player rot ' + player.rotString());
    trace('camera ' + chaseCamera.posString());
    trace('particles active: ' + particleSystem.particlesActive);
    trace('particles total: ' + particleSystem.particlesCreated);
    trace('particles polies: ' + particleSystem.totalpolycount);
    if (gameState == STATE_PLAYING)
        trace('step: ' + (profilingEnd - profilingStart) + 'ms');

    // time to remove the intro "cinematic"?
    if (gametimer.gameElapsedTime > introOverlayEndTime)
    {
        if (contains(introOverlay))
            removeChild(introOverlay);
        screenShakeCameraAngle = 0;
    }
}
```

What just happened?

The heartbeat function is still being used only for debug purposes by reporting on some interesting statistics that are helpful during debugging. When debugging your game, you can use these trace messages to see how many particles are currently active, where the player is and exactly how many milliseconds the `gameStep` function is taking in each frame.

Don't forget that this is the perfect place to put advanced AI routines, PVS culling, pathfinding and even streaming level loading, checking for completed downloads, changes to the music or other routines that only need to be run every once in a while rather than every frame.

Time for action – upgrading the `enterFrame` function

The final function in our game engine is the `enterFrame` function which updates the game timer, advances the game simulation, and kicks off the render loop. All the preceding functions are called by this function when appropriate.

```
private function enterFrame(e:Event) :void
{
    // clear scene before rendering is mandatory
    context3D.clear(0,0,0);

    // count frames, measure elapsed time
    gametimer.tick();

    // update all entities positions, etc
    gameStep(gametimer.frameMs);

    // render everything
    renderScene();

    // present/flip back buffer
    // now that all meshes have been drawn
    context3D.present();

    // update the GUI
    if (gameState == STATE_PLAYING)
    {
        updateScore();
    }

    updateFPS();
}

} // end of class
} // end of package
```

What just happened?

After all game inits have occurred the `enterFrame` event is triggered in every frame. It is used to call all the other gameplay functionality we programmed in our game, from movement and input handling to AI and simulation of particles. We also update the GUI with the framerate and current score as necessary.

That is it! We are done! Our game is ready to be played by friends and family.

Publish... distribute... profit!

This, brave adventurer, is why we went on this perilous quest. To take advantage of the sheer gaming power of Stage3D. We have successfully climbed the Molehill and built a fully functional 3D game in Flash.

After some beta testing, you might gather your player feedback and upgrade the game some more, adding polish and tweaks to really perfect the game. When you are satisfied, publish the game!

Here is what the final version of our game looks like. Notice how many bullets, explosions, enemy ships, asteroids and more are being rendered. Notice the incredible framerate. Notice that we are rendering over 250,000 polygons.



3... 2... 1... ACTION!

As this is a 3D Flash game filled with action, special effects, and tons of fun, your game can be played on virtually any computer in the world. It can be uploaded to websites, played on certain mobile devices and in the future might even be playable on a new TV or game console.

The limits to reach are only your imagination. There are literally millions of computers that can play your game right now—and millions of people that might enjoy playing it.

Upload your Flash game to a blog, Flash game portals like Kongregate and Newgrounds and a thousand other smaller game portals, integrate it into Facebook, create a standalone projector executable or sell the game on Steam... the choice is yours.

Have a go hero – a fun side quest

Your side quest this time is to take what you have learned, hack away happily at the example source code included with this book, and create your own fully polished, eminently playable, super-fun awesome game.

You probably have tons of ideas for games you want to make. Some are so simple that you might not really need half the code in this project. A perfect example would be a puzzle game. Use the rendering, input, timer, and GUI code to make a brilliant 3D puzzle game.

Some ideas are overly complex, like creating something with production values and scope that can compare favorably with Call of Duty, Fallout 3, or other multi-million dollar console games. Remember that, unless you have a big team, big bucks, and a couple of years of free time, it is better to aim for something smaller.

Others are just right; sitting in that sweet spot reserved for truly successful indie game developers. Try to choose something that will allow you to reach the finish line. Remember, many big budget games have 100 people working on them for two years. Working by yourself, this means you would have to put 200 years of labor into such a game. Obviously, simplicity is best.

Whatever you choose, try to bring the idea to fruition. Don't give up when you "hit the wall".

You CAN do it!

Summary

In this chapter, we managed to upgrade our engine to provide a fully playable game. All the little extras are there. We upgraded the render loop, improved the GUI, added game events such as collision detection, and fleshed out the game with a detailed level filled with new kinds of decorations and enemies.

Level 10 achieved. Universe saved!

Congratulations. You beat the final boss. You have successfully reached the end of your epic adventure. This is a major accomplishment. To have persevered though difficult challenges is something you should be very proud of.

You have programmed a simple but action-packed 3D Flash game, complete with animation, sounds, a title screen, player movement, collision detection, timers, particle systems, bullets, explosions, health, a score and more.

You made something with a beginning, middle, and end. From the title screen to game over, your creation can stand alone as a finished product.

You can now officially consider yourself a 3D video game developer.

Where to go from here?

This book was meant to be a very basic introduction to Adobe Stage3D (Molehill) game programming. There are so many ways in which your game could be improved.

From adding a true physics engine, looping sounds, fractal-generated geometry, more 3D file formats, multiple levels, cinematic movies, and advanced AGAL shaders, the "sky is the limit" in terms of what could have been added.

If you wanted to create complex physics simulations so that you could create a racing game or Angry Birds style ballistics simulation, you might be interested in googling the "Box2D", "bulletflash", or the "jiglib" physics engines.

Now that you know the basics of Stage3D, you might want to replace our simplistic shaders and .OBJ file parser with a large and feature-rich engine such as Away3D, Minko, Alternativa, Flare3D, and the like. These engines will handle the rendering side for you, but the majority of your game's source code (with all the timers, input routines, AI, title screen, score and combo displays and other "game-specific" functionality) will remain. You could easily drop in a complex renderer but still use everything else. You have a working "game engine" which resides on a higher level than most "3D engines" you will find online which typically only handle cameras and shaders.

For a wonderful list of game tools written in AS3, be sure to Google "AS3 Game Gears".

A note from the author

Thank you very much for buying this book. I have really enjoyed the adventure alongside you. I hope that the effort that went into writing it was worthwhile. If you learned something, had a little fun, or simply satisfied some of your curiosity, my mission has been accomplished.

As a loyal reader, access to some top-secret revisions to the game engine created here may be available. Be sure to check out www.mcfunkypants.com, which is my personal blog. There may also be new games to play, more source code to download, addenda and errata and much more. You are warmly encouraged to follow me on Twitter (<http://twitter.com/McFunkypants>) and Google Plus (<http://www.mcfunkypants.com/>+).

Finally, if you make your own game using what you have learned in this book, I would love to play it. Please contact me on my blog and let me know about it. Perhaps I will play your game and provide positive feedback and words of encouragement. I may also be able to send some additional players your way by spreading the word through social media.

I would be proud and honored to know that I helped you on your way to writing a 3D Flash game. Hearing from you would make my day, so please feel welcome to reach out.

Congratulations, brave adventurer. You did it. I salute you.

Sincerely,

Christer Kaitila
(aka Breakdance McFunkypants)



A

AGAL Operand Reference

AGAL (Adobe Graphics Assembly Language) is a high-performance language for the creation of vertex and fragment programs (shaders). See *Chapter 4, Basic Shaders: I can see Something!* for more details. Another good reference is available here:

http://help.adobe.com/en_US/FlashPlatform/beta/reference/actionscript/3/flash/display3D/Program3D.html.

What does one line of AGAL look like?

<opcode> <destination> <source 1> <source 2>

This means that for each line of code, the first token (chunk) will be the command or function that is going to run. This command or function is called the "opcode" which is a short form for operation code. The second token is the "destination"—where the answer should go. Subsequent tokens list the source data locations—the parameters used by the command being executed.

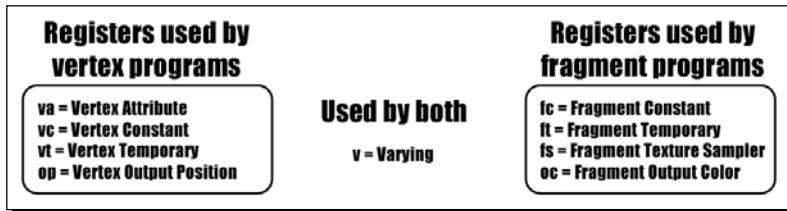
For example:

```
mov v0, val
```

In this example, the value stored in `val` is copied to `v0`.

Registers available for AGAL programs

AGAL opcodes work with registers. Some represent data that has been uploaded to the video card either through `Context3D:setVertexBufferAt` or `Context3D: :setProgramConstants` while others are merely used as temporary storage.



There are eight vertex attribute registers, `va0` to `va7`. This is where you access your mesh's vertex data that was sent to Stage3D with the `setVertexBufferAt` function. There are 128 vertex constant registers, `vc0` to `vc127` and 28 fragment constant registers, `fc0` to `fc27` which are sent to Stage3D with the `setProgramConstants` family of functions.

There are eight temporary registers for each kind of program, `vt0` to `vt7` (for vertex programs) and `ft0` to `ft7` (for fragment programs) which can be used for any purpose. The eight varying registers, `v0` to `v7` are used to pass information from a vertex program to the fragment program. In a fragment program, the eight texture samplers (used to access the textures defined by the `setTextureAt` function) are accessed through `fs0` to `fs7`.

Every register contains four numbers, which are accessed through `x,y,z,w` or `r,g,b,a` (you can use either—they mean the same thing). If you write code that just refers to the register (for example, `v0`), it will assume you mean all four components. You can refer to specific components of a register by specifying them. For example, you could refer to just `v1.x` instead of the entire `v1` register. Alternately, you can even "swizzle" the vector by specifying components in any order you wish, such as `v1.zzzz` or `v1.zxy` for example.

The output position (final result) of a vertex program is stored in `op`.

The output color (final result) of a fragment program is stored in `oc`.

Component-wise



The term "**component-wise**" means that the operation is performed on each component of the registers in question.

For example, as the `add` opcode adds together two registers in a component-wise fashion, the destination register would contain [`source1.x+source2.x`, `source1.y+source2.y`, `source1.z+source2.z`, `source1.w+source2.w`].

COPYING DATA

mov: destination = source.
Copies data from source to destination.

ALGEBRAIC OPERANDS

min: destination = min(source1, source2).
Minimum between two registers, component-wise.

max: destination = max(source1, source2).
Maximum between two registers, component-wise.

sqrt: destination = sqrt(source1).
Square root of one register, component-wise

rsq: destination = 1 / sqrt(source1).
Reciprocal of the square root of one register, component-wise.

pow: destination = pow(source1, source2).
Raise source1 register to the power of source2, component-wise.

log: destination = log2(source1).
Base 2 logarithm of one register, component-wise.

exp: destination = 2^source1.
2-based exponential function of one register, component-wise.
(Each element of the source register is evaluated as two raised to the power of itself).

nrm: destination = normalize(source1).
Normalize one register to a length of 1.

abs: destination = abs(source1).
Absolute value of one register, component-wise.

sat: destination = max(min(source1,1),0).
Clamp (saturate) one register in the range (0..1), component-wise.
(Ensures that each component in source1 is never less than zero or greater than one)

MATH OPERANDS

add: destination = source1 + source2.
Add two registers, component-wise.

sub: destination = source1 - source2.
Subtract two registers, component-wise.

mul: destination = source1 * source2.

Multiply two registers, component-wise.

div: destination = source1 / source2.

Division between two registers, component-wise.

rcp: destination = 1 / source1.

Reciprocal of one register, component-wise.

frc: destination = source1 - (float)floor(source1).

Fractional part of one register, component-wise.

neg: destination = -source1.

Negation of one register, component-wise.

TRIGONOMETRY OPERANDS

sin: destination = sin(source1).

Sine function of one register, component-wise.

cos: destination = cos(source1).

Cosine function of one register, component-wise

CONDITIONAL OPERANDS

kil: if one of source register components is < 0, pixel is discarded and not drawn
(a pixel shader only opcode).

sge: destination = source1 >= source2 ? 1 : 0.

Set destination to 1 if source1 is greater than or equal to source 2 or 0 otherwise,
component-wise

slt: destination = source1 < source2 ? 1 : 0.

Set destination to 1 if source1 is less than source 2 or 0 otherwise, component-wise

seq: destination = source1 == source2 ? 1 : 0

Set destination to 1 if source1 is equal to source 2 or 0 if not equal, component-wise

sne: destination = source1 != source2 ? 1 : 0

Set destination to 1 if source1 is not equal to source 2 or 0 if equal, component-wise

VECTOR and MATRIX OPERANDS

crs: Cross product between two registers.

destination.x = source1.y * source2.z - source1.z * source2.y

destination.y = source1.z * source2.x - source1.x * source2.z

destination.z = source1.x * source2.y - source1.y * source2.x

dp3: Dot product between two registers, 3 components.

destination = source1.x*source2.x + source1.y*source2.y + source1.z*source2.z

dp4: Dot product between two registers, 4 components.

destination = source1.x*source2.x + source1.y*source2.y + source1.z*source2.z +
source1.w*source2.w.

m33: matrix multiply a 3 component vector by a 3x3 matrix.

destination.x = (source1.x * source2[0].x) + (source1.y * source2[0].y) + (source1.z *
source2[0].z)

destination.y = (source1.x * source2[1].x) + (source1.y * source2[1].y) + (source1.z *
source2[1].z)

destination.z = (source1.x * source2[2].x) + (source1.y * source2[2].y) + (source1.z *
source2[2].z)

(produces only a 3 component result, destination must be masked to .xyz or less)

m34: matrix multiply a 4 component vector by a 3x4 matrix.

destination.x = (source1.x * source2[0].x) + (source1.y * source2[0].y) + (source1.z *
source2[0].z) + (source1.w * source2[0].w)

destination.y = (source1.x * source2[1].x) + (source1.y * source2[1].y) + (source1.z *
source2[1].z) + (source1.w * source2[1].w)

destination.z = (source1.x * source2[2].x) + (source1.y * source2[2].y) + (source1.z *
source2[2].z) + (source1.w * source2[2].w)

(produces only a 3 component result, destination must be masked to .xyz or less)

m44: matrix multiply a 4 component vector by a 4x4 matrix.

destination.x = (source1.x * source2[0].x) + (source1.y * source2[0].y) + (source1.z *
source2[0].z) + (source1.w * source2[0].w)

destination.y = (source1.x * source2[1].x) + (source1.y * source2[1].y) + (source1.z *
source2[1].z) + (source1.w * source2[1].w)

destination.z = (source1.x * source2[2].x) + (source1.y * source2[2].y) + (source1.z *
source2[2].z) + (source1.w * source2[2].w)

destination.w = (source1.x * source2[3].x) + (source1.y * source2[3].y) + (source1.z *
source2[3].z) + (source1.w * source2[3].w)

TEXTURE SAMPLING OPERAND

tex: Sample a texture in source2 at UV coordinates in source1. Only used in a pixel shader.

Example:

```
// grab the texture color from texture 0  
// and uv coordinates from varying register 1  
// and store the interpolated value in ft0  
tex ft0, v1, fs0 <2d,linear,repeat,miplinear>
```

Possible flags for the tex opcode:

For mip-mapping, one of: mipnone, nomip, mipnearest, miplinear

For filtering, one of: nearest, linear

For texture repeat, one of: repeat, wrap, clamp

B

Pop Quiz Answers

Chapter 1

Let's Make a Game Using Molehill!

1	C
2	B
3	A

Chapter 2

Blueprint of a Molehill

1	D
2	C
3	B

Chapter 3

Fire up the Engines

1	B
2	D
3	C

Chapter 4

Basic Shaders: I can see something!

1	A
2	B
3	C

Chapter 5

Building a 3D World

1	C
2	B
3	A

Chapter 6

Textures: Making Things Look Pretty

1	B
2	C
3	A

Chapter 7

Timers, Inputs, and Entities: Gameplay Goodness!

1	C
2	D
3	C

Chapter 8

Eye-Candy Aplenty!

1	B
2	D
3	C

Chapter 9

A World Filled with Action

1	D
2	A

Index

Symbols

- 2D bitmaps** 8
 - 2D texture** 63
 - 3D texture** 63
 - 3D coding terms**
 - about 15
 - axis 16, 17
 - matrices 18, 19
 - normal 17, 18
 - Vector3D 16
 - 3D content terms**
 - about 10
 - fragment program 14
 - mesh 11
 - polygon 11
 - shaders 13
 - texture 12
 - vertex 12
 - vertex program 13, 14
 - 3D demo**
 - blend modes, switching 157, 158
 - creating 89
 - folder structure 123
 - GUI, upgrading 145, 146
 - initShaders function, simplifying 152, 153
 - key presses, listening for 146-148
 - meshes, parsing 154
 - meshes, rendering 154, 156
 - new art, embedding 142, 143
 - render loop 108
 - render loop, starting 108-112
 - render loop, upgrading 149, 150
 - renderTerrain function, upgrading 150
 - score, adding to GUI 113
 - Stage3D inits, upgrading 150-152
 - texture effects, adding 141, 142
 - textures, switching 156, 157
 - variables, adding 144
 - vertex buffers, creating 90
 - 3D models, importing to Flash**
 - about 94
 - class constructor function, creating 96-98
 - data, processing 102-104
 - handy utility functions, coding 105-107
 - parsing functions, coding 98-102
 - Stage3dObjParser class, coding 94-96
 - 3D studio max** 200
 - 3D terminology** 10
 - 3D texture** 63
 - 6DOF (six degrees of freedom)** 357
 - _clickfunc function** 290
 - [embed] flex tag** 303
 - .FLA library** 143
 - .OBJ format** 201
 - .swf file**
 - publishing 84
 - _transformNeedsUpdate** 184
 - _valuesNeedUpdate flag** 190
- ## A
- AAA games** 168
 - ActionScript** 167
 - ActionScript source file** 166
 - actor**
 - spawning 278

actor pool
 animating 276
 rendering 277

actor reuse pool system
 about 274
 actor pool, animating 276
 actor pool, creating 274, 275
 actor pool, rendering 277
 actor, spawning 278, 279
 clone parent 275
 collisions, checking 279

actors
 animating 264
 destroying 282
 displaying 280
 hiding 280, 281
 spawning, based on map image 284, 285

actor's properties
 extending 261, 262

addChild() 25, 27

add opcode 60

Adobe CS5 39

Adobe Flash IDE 308

Adobe Graphics Assembly Language. *See AGAL*

AGAL
 about 48, 57, 225, 367
 algebraic operands 369
 code structure 59
 compiling 66
 component 60
 conditional operands 370
 data, copying 369
 features 57
 line of code 367
 math operands 369
 multiple components, working at once 60
 register 59
 registers 368
 rendering 67
 texture operand 372
 trigonometry operands 370
 types of registers 60
 used, for animating 225
 vector and matrix operands 371

AGAL commands
 add 59
 div 59
 mov 59
 mul 59
 sub 59

AGALMiniAssembler class 66

AGALMiniAssembler object 48, 67

AGAL shader example
 about 63
 fragment program 65
 fragment program, writing 65
 vertex program 64
 vertex program, writing 64

AGAL source code
 compiling 66, 67

ageScale 232

ageScale variable 234

AI
 implementing 265

AI implementation
 about 265
 actor, cloning 268, 269
 actor, respawning 270
 actor's death, handling 270
 enemies, shooting 266, 267
 timers, using 265

algebraic operands, AGAL
 abs 369
 exp 369
 log 369
 max 369
 min 369
 nrm 369
 pow 369
 rsq 369
 sat 369
 sqrt 369

Anecdotal benchmarks 8

animated textures
 about 132
 using 132

aNormalPointingUp 17

appendTranslation function 51

art
 embedding 205

art assets
 adding to game 308
 embedding, (AS3 version) 308-310
 embedding, (CS5 version) 311, 312

meshes, embedding 312-316
 tracking 316
art, for game world
 designing 200
artificial intelligence. *See* **AI**
AS3 editor 142
AS3 version
 about 59, 167
 embedding 308
asteroids
 initializing 338, 339
axis 16
axis-aligned-bounding-boxes (aabb) 272
AZERTY keyboards 289

B

backface culling
 about 133
 mesh backfaces, rendering 133
BitmapData 26
bitmap filters 133
Blender 200
blend modes
 about 135
 mesh, rendering for lightening the scene
 138, 139
 mesh, rendering with transparent regions 137
 opaque mesh, rendering 136
bounding-box collisions
 detecting 272
build19786 36
bullet hell arcade shooter game
 final quest 300
 finishing 299
 finish line 301
 variables, adding 304
bullets
 initializing 337, 338

C

C++ 59
callback function 264
Chapter 7 demo
 URL 219
clamp 63

class constructor function
 creating 96-98
classes
 importing 303
click events
 handling 290
click function 291
clone() function 196, 197
clone parent, actor reuse system 275
clones 196
collision detection
 about 271
 bounding-box collisions, detecting 272, 274
 collisions, detecting 271
 sphere-to-sphere collisions, detecting 272
complex code
 hiding, get and set functions used 183
component 60
conditional operands, AGAL
 kil 370
 seq 370
 sge 370
 slt 370
 sne 370
constant registers
 about 61
 fc<n> syntax 61
Context3D 27
Context3DCreate function 48
context3D.createTexture() function 127
context3D.drawTriangles() function 127
Context3D.setBlendFactors() function 135, 139
 parameters 139
Context3D::setProgramConstants() function 61
context3D.setTextureAt() function 127, 132
Context3D::setTextureAt() function 63
Context3D::setVertexBufferAt() function 61
crater-covered asteroid texture 201
createIndexBuffer() 104
cube texture 63

D

data
 uploading, to Stage3D 84
depth testing
 about 134
 mesh, creating without affecting zbuffer 134

die function 270
DirectX 9 24
dirty flag 184
DisplayObjects 25
drawTriangles function 52, 69, 90, 122
dynamic textures 132

E

embed tag 318
enemies
 initializing 333-336
enterFrame event 122
enterFrame function
 about 50, 80, 83, 150, 218
 upgrading 218, 362, 363
entity
 cloning 197
 rendering 198, 200
entity class
 extending, for particles 227
 extending, with game actor class 260
entity position
 getting 184, 186
 setting 184, 186
entity rotation
 getting 186
 setting 186
entity scale
 getting 187-189
 setting 187-189
entity transform
 updating 189, 190
eye-candy
 implementing 223
 quest 223

F

Facebook 364
fc0 79
fc15 328
fc16 328
FDT 308
final game source code
 actor types, defining 329, 330
 asteroids, initializing 338, 339
 behaviors, defining 329, 330

 bullets, initializing 337, 338
 enemies, initializing 333-336
 game level, creating 341-343
 initGUI function, upgrading 321-324
 inits, upgrading 317, 318
 particle models, initializing 340, 341
 render loop, upgrading 343-345
 shaders, upgrading 326-328
 space stations, initializing 339, 340
 Stage3D, initializing 319-321
 terrain meshes, initializing 331-333
 texture inits, upgrading 324, 325
 upgrading 317
final version, game
 demo 363, 364
finish line, 3D game
 about 301
 classes, importing 303, 304
 features, adding 301, 302
 title screen, drawing 302, 303
Flash 11 plugin
 downloading, from Adobe 34, 35
 download page 34
 Flash 11 profile, for CS5 36
 Flex playerglobal.swc, upgrading 37
 Flex, upgrading 36
 SWF Version 13, using 38
 template HTML file, updating 38
Flash 11 profile, for CS5 36
Flash ActionScript 9, 10
Flash Builder 39, 308
FlashDevelop
 about 20, 39, 40, 142, 308
 URL 20
Flex
 about 142, 167, 308
 download link 37
 upgrading 36
Flex playerglobal.swc
 upgrading 37
flipAxis parameter 263
flipTexture parameter 263
floating point value 59
focus events 293
folder structure, 3D demo 123
follow() function 196

FPS counter
adding 70
data, uploading to Stage3D 84
FPS GUI, creating 70-72
GUI, adding to inits 72-74
multiple shaders, adding to demo 74-76
shaders, animating 80-83
shaders, initializing 77-80
FPS display
updating 351-354
FPS GUI
creating 70, 71
fractal 133
fragment program
about 14, 58, 65
writing 65
framerate-independent 164
framerate-independent simulation 172

G

GameActor class
about 274
creating 260, 261
GameActor class constructor
coding 262
GameActorpool.as 274
GameActorpool class 277, 284
game code
entity, cloning 197
entity position, getting 184, 186
entity position, setting 184, 186
entity, rendering 198-200
entity rotation, getting 186
entity rotation, setting 186
entity scale, getting 187-189
entity scale, setting 187-189
entity transform, updating 189, 190
entity values, updating 189, 190
handy entity utility functions, adding 194-196
movement utility functions, creating 190-192
transform, getting 183, 184
transform, setting 183, 184
vector utility functions, implementing 192, 193
game code, hiding
get and set functions used 183

game entities
spawning 212-214
game events
tracking 345-347
game inits
upgrading 207, 208
GameInput class
about 172, 288
creating 173
gameinput constructor 318
gameinput.pressing.fire 220
game level
creating 341-343
GameLevelParser.as 284
GameLevelParser class 288, 301, 310
GameLevelParser object 342
game over
handling 348, 349
gameplay 163
gameplay-specific events
collision events, handling 352-354
defining 345
enterFrame function, upgrading 362, 363
FPS display, updating 351
game events, tracking 345-347
game over, handling 348, 349
gameStep function, upgrading 358-360
heartbeat function, upgrading 361, 362
player input, handling 354-357
score display, updating 350, 351
gamestart function 318, 348
game state
tracking 304
gameStep() function
about 217, 220
upgrading 358-360
GameTimer class
about 168
creating 169
GameTimer class constructor
adding 170
game, upgrading
enterFrame function, creating 218
game entities, spawning 212-214
game inits, upgrading 207, 208
GUI, upgrading 208

heartbeat function, creating 217
 new art, embedding 205, 206
 new classes, importing 202
 new textures, using 210, 211
 render function, upgrading 215, 216
 shaders, simplifying 209
 simulation step function, creating 216, 217
 variable, adding 203, 204
GC (garbage file) 225, 240
get function 182
getTimer() function 21, 171
get y function 190
GPU (graphics processing unit) 8
GUI
 adding, to inits 72-74
 upgrading 113, 208
GUI overlay
 adding, to game 167

H

handy entity utility functions
 adding 194-196
 coding 105-107
hardware 3D graphics processors 8
haXe 40, 308
health property 261
heartbeat function
 about 168
 creating 217
 examples 169
 features, implementing 169
 upgrading 361, 362
HUD (heads-up-display overlay image)
 about 302
 upgrading 307
hud_overlay bitmap 167
HUD overlay graphic
 adding, to game 166

I

indexBuffer 69
IndexBuffer3D 28
init code 72, 318
initData() function
 about 212
 tweaking 154

init() function 42, 74, 146, 208
initGUI() function
 about 74, 167, 208, 303
 upgrading 321-324
initParticleShader function 236
init routines
 upgrading 115, 116
inits
 upgrading 317, 318
initShaders() function
 about 77, 78, 80, 118, 209
 simplifying 152
initTexture function 325
input focus
 detecting 178
input routines, upgrading
 about 288
 click events, handling 290
 key events, upgrading 291, 293, 295
 properties, adding to input class 289
interactivity 163
isClean 184

K

keyboard input
 detecting 176, 177
key events
 upgrading 291
keyframed particle meshes
 about 246
 group of particles, sculpting 247, 248
 particle, sculpting 246
 particle texture, selecting 246
 second keyframe, sculpting 248
keyframed particle vertex program
 creating 236, 237
keyframed vertex animation shader
 about 236
 keyframed particle vertex program, creating
 236, 237
 non-keyframed particle vertex program,
 creating 238
 particle fragment program, creating 238, 239
 particle shader, compiling 239
keyPressed event 294

keyPressed function 177
about 177
creating 146
keyReleased function 294
key release events
detecting 177, 178
keyup events 178
KEY_UP events 174
Kongregate 364

L

LEVELDATA art asset 310
LEVELKEYDATA art asset 310
level parser class
implementing 284
lightambient variable 329
lightvector variable 329
line of code, AGAL 367
linear 63
LOD (level-of-detail) 168
lostFocus event 293
lostFocus function 179

M

Mac OS-X 24
map image pixels
parsing 286-288
massive levels
creating 283
math operands, AGAL
add 369
div 370
frc 370
mul 370
neg 370
rcp 370
sub 369
matrices 18
Matrix3D class 18
mesh 11
mesh data
parsing 118
meshes
embedding 312-314
meshNumMax variable 148

meshVertexData variable 55
miplinear 63
MIP mapping 49, 63
mipnearest 63
mipnone 63
modelMatrix 51, 83
modelViewProjection matrix 83
moiree effect 63
moiree patterns 49
Molehill

3D World, building 89
about 8, 9
initializing 42
real game, creating 126
Molehill application
Context3D 27
IndexBuffer3D 28
Program3D 28
Stage 27
Stage3D 27
structure 26
VertexBuffer3D 28

Molehill program
flowchart 29
setting up 29
mouseIsDown property 175
mouseMove function 175, 291
mouse movement
detecting 174, 175
mouseUp function 291
move_amount 221
moveForward function 192
moveItByThisMuch 21
movement utility functions
creating 190-192
multiple shaders
adding, to demo 74-76
MXML files 40

N

nearest 63
Newgrounds 364
non-keyframed particle vertex program
creating 238
normal 17, 18
normalize() function 18

O

OBJ file parser 263
onContext3DCreate event 212
onContext3DCreate event handler 47
onContext3DCreate function
 about 47, 116, 150
 using 210
onContextCreate() function 74
onFrame() event 21
opcode 59
OpenGL ES2 24
ourinputclass.pressing.left 173
output registers 62
overdraw
 about 140
 avoiding 140

P

parseLine function 97
parseNormal function 99
parsing functions
 coding 98-102
particle class
 constructor, coding 228
particle class constructor
 coding 228, 229
particle entity class
 about 226
 extending 227
 numbers, generating 230, 231
 particle class constructor, coding 228, 229
 particle properties, adding 228
 particles, cloning 230
 particles, rendering 233-235
 particles, respawning 232, 233
 particles, simulating 231, 232
particle fragment program
 creating 238, 239
particle pool 225
particle models
 initializing 340, 341
particle properties
 adding 228
particles
 adding, to game 249
 cloning 230

entity class, extending for 227
rendering 233
respawning 232
simulating 231, 232
particle shader
 compiling 239
particle system class
 incorporating, in game 249
particle system class, incorporating in game
 particles, adding to game 249, 250
 particles, adding to gameStep function
 252, 253
 particle statistics, tracking 253, 254
 particle type, preparing 250, 251
 renderScene function, upgrading 251, 252
particle system manager class
 about 240
 coding 241
 particles, creating 244, 245
 particles, rendering 243
 particles, simulating 242, 243
 particles, spawning 244
 particle type, defining 242
particle type
 defining 242
performance
 designing for 224
perline noise functions 133
Photoshop 200
pixel shaders 58
pizazz 167
player input
 handling 354-357
Playstation 3 8
polygon 11
power-of-two sized textures 127, 128
present() function 69
private_z variable 186
Program3D 13, 28, 63
puff.obj model 201
PVS (potentially visible set) 141

R

readClass function 101
real game
 creating 126

rendering performance, achieving 139
render states 133
 texture effects, adding to demo 141
 textures, using in Stage3D 126, 127
registers
 about 59
 constant registers 61
 output registers 62
 temporary registers 61
 texture samplers 62
 types 60
 varying registers 62
 vertex attribute registers 61
registers, AGAL
 about 368
 fc0 to fc27 368
 fs0 to fs7 368
 oc 368
 op 368
 v0 to v7 368
 va0 to va7 368
 vc0 to vc127 368
render function 233, 360
 upgrading 215, 216
rendering 67, 68
rendering performance
 fewer meshes, drawing 141
 opaque .JPG textures, using 140
 overdraw, avoiding 140
 simple shaders, using 141
 state changes, avoiding 140
render loop
 about 108
 init routines, upgrading 115-117
 mesh data, passing 118
 scene, animating 118-121
 score, adding to GUI 113-115
 starting 108-112
 upgrading 343-345
renderMesh function 149, 150, 157
render states
 about 133
 backface culling 133
 blend modes 135
 depth testing 134
renderTerrain function
 about 150
 upgrading 150
renderTriangles function 140
repeat 63
respawn function 233, 236, 270
reusable game timer class
 creating 165
reuse-ability
 designing for 225
rgbaScale 232

S

score display
 updating 350, 351
setBlendFactors() function 159
setCulling() function
 about 159
 parameters 134
setDepthTest function
 about 134, 159
 parameters 134
set function 182
setProgramConstantsFromMatrix function
 64, 69
setProgramConstantsFromVector() function 83, 132, 329
setProgram function 69
setTextureAt command 69
setTextureAt function 66
setVertexBufferAt commands 83
setVertexBufferAt function 65, 68, 90, 122
shader 58
shader demo
 creating 69, 70
 URL 70
shaders
 about 13
 animating 80, 82
 initializing 77-80
 upgrading 326-329
shaderWithLight 329
simulation step function
 creating 216, 217
source code
 URL 53, 219
spaceship_texture.jpg 112
space stations

initializing 339, 340
spawnActors() function 284, 2877
spawn function 278
spawn() requests 286
sphere-to-sphere collisions
detecting 272
spiral 286
Stage3D game
2D Flash text, using 25
about 8, 27
abstract entity class 179, 180
animating, AGAL used 225
art, designing 200
collision detection 271
complex code, hiding 183
features 25, 26
game input class 172
GameInput class constructor, coding 174
GameInput class, creating 173
game timer class 168
GameTimer class constructor, adding 170
game timer class, creating 169
GUI overlay, adding 167
HUD overlay graphic, adding 166
initializing 319, 320
input focus, detecting 178
input routines, upgrading 288
keyboard input, detecting 176, 177
key release events, detecting 177, 178
level parser class, implementing 284
levels, creating 283
making interactive 166
making reusable 165
Molehill way 24
mouse movement, detecting 174, 175
player input, combining with timer-based
animation and chase camera 164
quest 164
simplistic actions 164
sprites, using 25
Stage3dEntity class constructor, creating
182, 183
Stage3dEntity class, creating 180, 182
tick function, implementing 170, 171
time, measuring 172
traditional approach 23, 24
upgrading 201

Stage3D API 9
Stage3D coding
3D mesh geometry, defining 45, 46
about 39
camera, setting up 50
data, uploading 48, 49
empty project, creating 39-41
enterFrame function 50, 51
init() function 46
mesh, drawing 51
Molehill, initializing 42
onContext3DCreate function, adding 47, 48
render state, setting 51
Stage3D-specific classes, importing 41
texture, embedding 43, 45
variables, defining 42
Stage3D-enabled playerglobal.swc file
download link 37
Stage3dEntity class
about 212, 226, 260
creating 179-182
Stage3dEntity class constructor
creating 182, 183
Stage3dGame 39
Stage3dGame.as source file 142
Stage3dGame class 304
Stage3dGame project 70
Stage3D inits
upgrading 150
Stage3D objects 25
Stage3dObjParser class
about 183, 204
coding 94-96
Stage3dParticle class 341
Stage Video 26
statechanged parameter 199
step animation function
creating 263, 264
step() function 216, 231, 234, 263, 264
SWF Version 13
used, for compiling Flex 38
SwiftShader 24

T

template HTML file
updating 38

temporary registers
 ft<n> syntax 61
 vt<n> syntax 61

terrain meshes
 initializing 331-333

terrainTextureBitmap 112

TextField objects 74

TextFields 146

TextFormat object 146

texture 12

texture atlases
 about 131
 using 132

texture data
 manipulating 132

texture dimension 63

texture filtering 63

texture inits
 upgrading 324, 325

Texture object 127

textures
 animated textures 132
 power-of-two sized textures 127, 128
 texture atlases 131
 texture data, manipulating 132
 transparent textures 129
 using 127
 UV coordinates 128, 129
 UV coordinates, animating in shader 130

texture repeat 63

texture samplers
 about 62
 ft<n> <flags> syntax 63
 mip mapping 63
 texture dimension 63
 texture filtering 63
 texture repeat 63

texture sampling operand, AGAL
 tex 372

Texture.uploadFromBitmapData() function 127

tick function
 implementing 170, 171

tileX parameter 286

tileY parameter 286

time-based simulation 166

title screen
 drawing 302, 303

trace function 196

transform
 about 19
 getting 183
 setting 183

transparent textures 129

trenchlike 286

trigonometry operands, AGAL
 cos 370
 sin 370

truetype font 311

U

unicodeRange parameter 311

update_score() function 208

updateTransformFromValues() 184

updateValuesFromTransform() function 190

uploadFromBitmapData() 132

uploadTextureWithMipmaps() function 208, 325

UV coordinates
 about 128
 animating, in shader 130
 example 129
 updating 131

uvOffsets 130

V

va registers
 about 60
 va0 60
 va1 60
 va2 60
 va3 60
 va4 60
 va5 60
 va6 60
 va7 60

variables
 adding, to game 203, 304
 entities, keeping track of 306, 307
 for timer-based events 305
 HUD, upgrading 307
 track the game state, implementing 304
 used, for controlling player 305, 306

variables, used by Stage3D
 defining 308

varying registers
about 62
 $v<n>$ syntax 62

Vector3D 16, 17

vector and matrix operands, AGAL
crs 371
dp3 371
dp4 371
m33 371
m34 371
m44 371

vectors 16

vector utility functions
implementing 192, 193

vertex 12

vertex attribute registers
about 61
 $va<n>$ syntax 61

VertexBuffer 61
assigning, to specific attribute register 61

VertexBuffer3D 28, 90

vertex buffers
3D models, importing to Flash 94
creating 90-93
Stage3dObjParser class, coding 94, 95

vertex program
about 13, 14, 58, 64
writing 64

vertex shaders 58

video card (GPU) 58

viewmatrix variable 152

W

wmode=direct 39, 84

wobble functions 231

word shader 58

wrap 63

X

XBOX 360 8



Thank you for buying **Adobe Flash 11 Stage3D (Molehill) Game Programming Beginner's Guide**

About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.PacktPub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

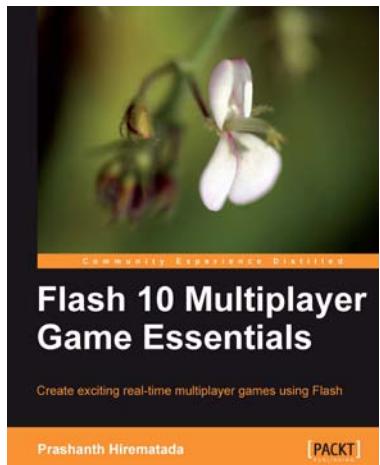


Flash Game Development by Example

ISBN: 978-1-84969-090-4 Paperback: 328 pages

Build 10 classic Flash games and learn game development along the way

1. Build 10 classic games in Flash. Learn the essential skills for Flash game development
2. Start developing games straight away. Build your first game in the first chapter
3. Fun and fast paced. Ideal for readers with no Flash or game programming experience.
4. The most popular games in the world are built in Flash.



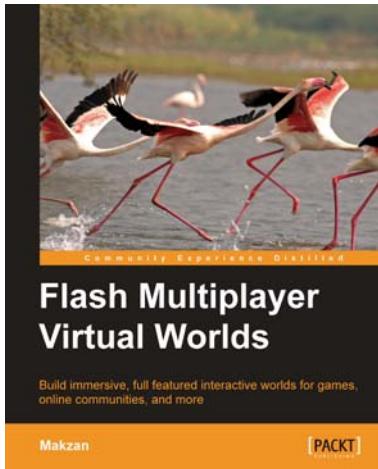
Flash 10 Multiplayer Game Essentials

ISBN: 978-1-847196-60-6 Paperback: 336 pages

Create exciting real-time multiplayer games using Flash

1. A complete end-to-end guide for creating fully featured multiplayer games
2. The author's experience in the gaming industry enables him to share insights on multiplayer game development
3. Walk-through several real-time multiplayer game implementations
4. Packed with illustrations and code snippets with supporting explanations for ease of understanding

Please check www.PacktPub.com for information on our titles

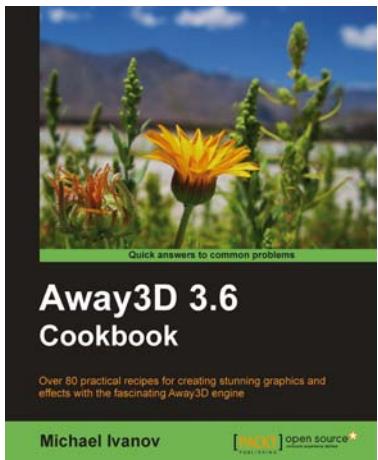


Flash Multiplayer Virtual Worlds

ISBN: 978-1-849690-36-2 Paperback: 412 pages

Build immersive, full-featured interactive worlds for games, online communities, and more

1. Build virtual worlds in Flash and enhance them with avatars, non player characters, quests, and by adding social network community
2. Design, present, and integrate the quests to the virtual worlds
3. Create a whiteboard that every connected user can draw on
4. A practical guide filled with real-world examples of building virtual worlds



Away3D 3.6 Cookbook

ISBN: 978-1-84951-280-0 Paperback: 480 pages

Over 80 practical recipes for creating stunning graphics and effects with the fascinating Away3D engine

1. Invaluable tips and techniques to take your Away 3D applications to the top
2. Reveals the secrets of cleaning your scene from z-sorting artifacts without killing your CPU
3. Get 2D objects into the 3D world by learning to work with TextField3D and extracting graphics from vector graphics
4. Learn essential topics like collision detection, elevation reading, terrain generation, skyboxes, and much more

Please check www.PacktPub.com for information on our titles