



Quick answers to common problems

Away3D 3.6 Cookbook

Over 80 practical recipes for creating stunning graphics and effects with the fascinating Away3D engine

Michael Ivanov

[PACKT] open source 
PUBLISHING community experience distilled

Away3D 3.6 Cookbook

Over 80 practical recipes for creating stunning graphics
and effects with the fascinating Away3D engine

Michael Ivanov



BIRMINGHAM - MUMBAI

Away3D 3.6 Cookbook

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: May 2011

Production Reference: 1190511

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-849512-80-0

www.packtpub.com

Cover Image by Charwak A (charwak86@gmail.com)

Credits

Author

Michael Ivanov

Project Coordinator

Michelle Quadros

Reviewer

J. Pradeek

Ronen Tsamir

Proofreader

Joanna McMahon

Development Editor

Maitreya Bhakal

Indexer

Hemangini Bari

Technical Editors

Prashant Macha

Gauri Iyer

Graphics

Nilesh R. Mohite

Production Coordinator

Aparna Bhagat

Copy Editor

Leonard D'Silva

Cover Work

Aparna Bhagat

About the Author

Michael Ivanov, although initially dreaming of becoming an historian while in the middle of his undergraduate studies in Hebrew at the University of Jerusalem, understood that his real passion was computers. Today he is a professional Flash developer working in the field of web and game development for more than 5 years. Being a researcher by nature with a strong hunger for technological exploration, he constantly broadens his professional knowledge by exploring a wide range of technologies from different fields of computer science where real time 3D engines are of primary interest. For the past 2 years, Michael has been working as lead programmer for Neurotech Solutions Ltd, which is a rapidly growing Israeli startup bringing revolutionizing solutions in the field of research and treatment of ADHD. Michael led a development of unique ADHD training game programs which have been based on the Adobe Flash platform powered by open source 3D libraries such as PV3D and Away3D. Although in everyday life he works mostly on RIA and desktop applications development, his true passion is 3D graphics and game programming. In his little spare time, Michael works with such technologies as Java3D Epic's UDK, Unity3D, and a wide range of Flash open source 3D libraries from which Away3D is his favorite. Michael is a zealous promoter of cutting edge 3D technologies especially of open source and misses no opportunity to speak on these subjects at local game industry events and conferences. Born in Russia, Michael has lived and worked in Israel for more than 12 years. When he is not writing a code, he enjoys playing around with his 3-year-old son David as well as reading history books and running.

You can find him on the web on his personal blog <http://blog.alladvanced.net>. Or as an active member of the Away3D developers' community group at <http://groups.google.com/group/away3d-dev?pli=1>.

Acknowledgement

I would like to express my thanks to Ralph Hauwert, John Lindquist, Carlos Ulloa , Mr. Doob, Keith Peters for their incredible work that served me as an inspiration for many recipes in this book. Thanks to the Away3D team who always found time to answer my nagging questions in the middle of their hard work on the next version of the engine. Fabrice3D deserves a special credit for his help and for contributing to the community Prefab3D

Special thanks to GreenSock (www.greensock.com) team who allowed me to use their wonderful tween engine TweenMax.

Many thanks to Tom Tallian (<http://tomtallian.com/>) for creating those cute "Team Fortress" low poly characters (whereas the Spy is my favorite and featuring in this book) and making them available for free of charge.

I would like to thank the Packt publishing for allowing me to put on paper this humble writing. I would not have been able to accomplish this book without the guidance and assistance by Acquisition editors Darshana Shinde and Maitreya Bhakal who accompanied me all the way to publishing. Last, but not the least, I want to express my great appreciation to Ronen Tzamir and Pradek for reviewing my stuff despite their daily workloads.

Finally a word for Flash community in general and especially to Away3D users group. One of the inspirations that served me as a driving force in this endeavor was my dedication and thankfulness to the community and the open source concept. I owe a bulk of my knowledge to these two. Thank you folks and I hope you will enjoy reading this book.

About the Reviewers

J.Pradeek is currently a computer science student at B.S.A Crescent Engg. College in Chennai, India. He has been working with Flash since his school days. He is specifically interested in working on emerging new web technologies and can be found programming in his free time. Contact Pradeek at jpradeek@gmail.com.

Ronen Tsamir is the R&D manager of Revolver LTD where he leads the FreeSpin3D development. FreeSpin3D is a 3D engine for Flash developers and designers.

Ronen also develops different unique 3D games engines according to special requirements.

Ronen has 25 years of experience in games development, starting at the age of twelve years with Spectrum 48k through Apple II up to today's new age cellular telephones.

Since the beginning of this century, he has been focusing especially on 3D games using mainly DirectX and Flash ActionScript.

Ronen's three main strengths are keeping your code simple and flexible, keeping the CPU low, and keeping the user interface accessible.

Ronen Tsamir web (Hebrew / English): <http://Ronen.tsamir.net>.

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 a humble gift to my dear parents who always taught me that
the sky is the limit and only through hard work do dreams come true*

Table of Contents

Preface	1
Chapter 1: Working with Away3D Materials	7
Introduction	7
Using a single MovieClip for multiple materials	8
Controlling the MovieClip of MovieMaterial	12
Streaming VOD from FMS for VideoMaterial	16
Interpolating material color dynamically with DepthBitmapMaterial	21
Creating normal maps in Photoshop	24
Generating normal maps using Away3D NormalMapGenerator	26
Getting your hands dirty with Pixel Bender materials	30
Assembling composite materials	38
Chapter 2: Working with Away3D Cameras	41
Introduction	41
Creating an FPS controller	42
Creating Camera Depth of Field Effect	50
Detecting whether an object is in front of or behind the camera	56
Changing lenses	60
Following a third-person view with a spring camera	67
Tracking screen coordinates of 3D objects	72
Transforming objects in 3D space relative to the camera position	75
Using Quaternion camera transformations for advanced image gallery viewing	78
Chapter 3: Animating the 3D World	85
Introduction	85
Animating (Rigging) characters in 3DsMax	86
Controlling bones animation in Collada	90
Working with MD2 animations	94

Table of Contents

Morphing objects	100
Animating geometry with Tween engines	105
Moving an object on top of the geometry with FaceLink	112
Chapter 4: Fun by Adding Interactivity	119
Introduction	119
Adding rotational interactivity to an Away3D primitive by using	
Mouse movements	120
Implementing advanced object rotation using vector math	125
Creating advanced spherical surface transitions with Quaternions	128
Interactively painting on the model's texture	133
Dragging on geometry by unprojecting mouse coordinates	137
Morphing mesh interactively	141
Creating a controllable non-physical car	145
Chapter 5: Experiencing the Wonders of Special Effects	153
Introduction	153
Exploding geometry	154
Creating advanced bitmap effects using filters	164
Creating clouds	169
Visualizing sound in 3D	173
Creating lens flair effects	179
Masking 3D objects	183
Chapter 6: Using Text and 2D Graphics to Amaze	189
Introduction	189
Setting dynamic text with TextField3D	190
Interactive animation of text along a path	195
Creating 3D objects from 2D vector data	200
Drawing with segments in 3D	204
Creating a 3D illusion with Away3D sprites	208
Chapter 7: Depth-sorting and Artifacts Solutions	213
Introduction	213
Fixing geometry artifacts with Frustum and NearField clipping	214
Removing artifacts from intersecting objects	217
Solving depth-sorting problems with Layers and Render Modes	221
Chapter 8: Prefab3D	227
Introduction	227
Exporting models from Prefab	228
Normal mapping with Prefab	232
Maintaining workflow with AwayConnector	238
UV map editing with Prefab	239

Table of Contents

Creating terrain	244
Generating light maps	248
Creating and animating paths	253
Chapter 9: Working with External Assets	261
Introduction	261
Exporting models from 3DsMax/Maya/Blender	262
Exporting models from 3DsMax to ActionScript class	266
Preparing MD2 models for Away3D in MilkShape	269
Loading and parsing models (DAE, 3ds, Obj, MD2)	274
Storing and accessing external assets in SWF	280
Preloading 3D scene assets	283
Chapter 10: Integration with Open Source Libraries	289
Introduction	289
Setting Away3D with JigLib	290
Creating a physical car with JigLib	295
Morphing Away3D geometry with AS3DMOD	303
Exploding particles with FLINT	310
Setting Augmented Reality with FLARToolkit in Away3D	314
Adding Box2D physics to Away3D objects	320
Chapter 11: Away3D Lite	329
Introduction	329
Setting up Away3D Lite using templates	330
Importing external models in Away3D Lite	332
Manipulating geometry	335
Making 2D shapes appear three dimensional by using Sprite3D	340
Managing Z-Sorting by automatic sorting and using layers	343
Creating virtual trackball	349
Writing Away3D Lite applications with haXe	355
Chapter 12: Optimizing Performance	361
Introduction	361
Defining depth of rendering	362
Restricting the scene poly count	365
Working with LOD objects	368
Optimizing geometry by welding vertices	371
Excluding objects from rendering	375
Image size consideration	380
Important tips for performance optimization	383

Table of Contents —————

Chapter 13: Doing More with Away3D	387
Introduction	387
Moving on uneven surfaces	387
Detecting collisions between objects in Away3D	392
Creating cool effects with animated normal maps	403
Creating skyboxes and their textures	409
Running heavy scenes faster using BSP trees	417
Appendix: Beginning Molehill API	431
Creating a rotating sphere from scratch	432
Index	451

Preface

Three dimensions are better than two — and it's not a secret anymore that 3D is here to stay. Gone are the days when Flash was just used for 2D animations. In the last few years, online Flash content has undergone a revolution with the introduction of real-time 3D engines for Flash. Away3D is the big daddy of them all—which makes it the ultimate resource for top-rated 3D content development and for powering today's coolest games and Flash sites. The Away 3D 3.6 Cookbook is your answer to learning all you need to take your Flash or Away3D skills to the next level — and having fun doing it.

This book is your practical companion that will teach you more than just the essentials of Away3D, and will provide you with all the tools and techniques you need to create a stunning 3D experience. You will find recipes targeting every possible field related to Away3D, 3D development pipelines, and best practices in general. You will find practically relevant content exploring advanced topics, which will clear your way to developing cutting edge applications — not to mention saving hours of searching for help on the internet.

The recipes in this book will teach you diverse aspects of 3D application development with Away3D. They will guide you through essential aspects such as creation of assets in external programs and their integration into Away3D, working with material, animation, interactivity, special effects, and much more. Each topic is packed with recipes targeting different levels of complexity, so that even experienced Away3D developers will find a lot of useful and unique information.

By the time you are done with this book, you'll be creating your own awesome Away 3D applications and games in less time than you can say "design".

What this book covers

Chapter 1, Working with Away3D Materials: In this chapter, you will learn how to create different types of Away3D material including PixelBender-based shaders. The chapter also covers advanced topics such as Normal mapping and FMS VOD streaming for VideoMaterial.

Chapter 2, Working with Away3D Cameras: Here you get acquainted with Away3D cameras. You will also learn to set up a First Person Controller, creating cool camera effects, dive into advanced 3D math by learning to perform complex camera transformations, and so on.

Chapter 3, Animating the 3D World: This chapter is going to teach you how to breathe life into your 3D world. It covers important topics such as the character animation setup in different formats, animation control, mesh morphing effects, and tweening 3D objects with the help of Tween Engine.

Chapter 4, Fun by Adding Interactivity: Time for fun! And what can be more fun than playing interactively within your 3D world. This chapter covers the most essential as well as advanced topics regarding 3D transformations. Also after finishing it, you will be able to create a fully interactive controllable car!

Chapter 5, Experiencing the Wonders of Special Effects: There is no way successful 3D content can exist without having special effects applied. Besides passing through a quick demolition course, you will learn creating sound visualization, realistic animated clouds, as well as advanced bitmap manipulation.

Chapter 6, Using Text and 2D Graphics to Amaze: How can 2D exist inside 3D? Yes it can! Learn how to create 3D text by faking a 3D look using Away3D sprites and place 2D vector shapes inside your 3D environment.

Chapter 7, Depth-sorting and Artifacts Solutions: Knowing how to model in your favorite 3D package is not quantum physics. Another matter is to cause your models to look good inside Away3D, which can turn out to be a serious challenge. Here you will meet the tools supplied by Away3D to help you get rid of the irritating "Z Fighting" and learn important techniques to fix depth sorting of your objects in the scene using layers. The rendering modes and cases will be explained along with when and how they should be used.

Chapter 8, Prefab3D: Prefab3D is an Adobe AIR visual 3D editor created especially for Away3D, which can significantly boost the developed process and save you a lot of precious hours. In this chapter, you will cover most of the software's features including topics such as models export, terrain generation, and light maps. Also, you will learn how to create, extrude, animate, and export paths for instant usage in Away3D.

Chapter 9, Working with External Assets: The chances are high that you will not be satisfied only by the formidable set of primitive Away3D sets at your disposal. This chapter will guide you through all the major techniques you need to know in order to export custom 3D assets from three major 3D packages—Autodesk 3DsMax, Maya, and Blender. Additionally, you will learn such important topics such as multiple object loading and a way to compact your 3D assets for a smaller weight using SWF as a resource.

Chapter 10, Integration with Open Source Libraries: Away3D is awesome, but you can make it even more awesome by incorporating other open source libraries into your projects such as particles, physics, and even Augmented Reality engines! Here you will have a quick start with popular frameworks such as FLINT particles, Box2DFlash, JigLibFlash, FLARManager, and AS3DMOD.

Chapter 11, Away3D Lite: Meet the younger brother of Away3D—Away3D Lite. It is not as robust and feature packed as Away3D 3.6, but it is incredibly fast! In these 30 plus pages, you are introduced to a kick-off crash course with the Away3D Lite engine which is going to add a lot of horsepower to your 3D toolset.

Chapter 12, Optimizing Performance: While finishing the work on your project, don't hurry to wrap it up. The performance issues are always waiting for you around the corner. The recipes in this chapter contain important tips on how to gain more FPS for your application. Usage of LOD objects, selective rendering, depth rendering restriction, and more will allow you to push the boundaries of what it is possible to do inside the Flash Player.

Chapter 13, Doing More with Away3D: This chapter contains an extra or bonus material which is not specifically theme related. Here we cover some advanced topics such as collision detection, moving on uneven terrain, and so on. Moreover, you will get introduced to the powerful BSP trees system that allows creating really vast indoor environments while having your FPS high and steady.

Appendix A, Beginning Molehill API: This appendix has nothing to do with the rest of the book as it introduces the next generation GPU-accelerated Flash Player that includes a 3D low-level API called Molehill allowing you to create true 3D content with unprecedented performance. If a while ago you wondered how to squeeze 10,000 triangles into an Away3D scene without killing the CPU, you can now load hundreds of thousands of polygons and the frame rate will not even blink! The appendix wraps a single recipe, which will give you an in depth introduction to Molehill API with the practical example of creating rotating sphere primitives from scratch.

What you need for this book

- ▶ Adobe Flex 2,3, or Flash Builder4
- ▶ Adobe Flash cs4/cs5
- ▶ Adobe Photoshop cs4/cs5
- ▶ FlashDevelop
- ▶ Autodesk 3DDds Max (version 7 or later)
- ▶ Terragen and TerraSky
- ▶ FMS 3.5
- ▶ BlazeDS
- ▶ MilkShape
- ▶ Prefab

Who this book is for

The book is written for experienced Flash developers who want to work with the Away3D engine as well as for those who are already acquainted with the engine but wish to take their skills to the next level.

Basic knowledge of Away3D and familiarity with ActionScript 3.0 and OOP concepts is assumed.

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: "Set up a basic Away3D scene extending the `AwayTemplate` class and you are ready to go."

A block of code is set as follows:

```
private function setSphericalLens():void{
    _hoverCam.lens=_sphericalLens;
    _hoverCam.fov=56;
}
```

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: "Click **Build** and then **Apply** button. You should see a similar model in your viewport".



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

Working with Away3D Materials

In this chapter, we will cover:

- ▶ Using a single MovieClip for multiple materials
- ▶ Controlling the MovieClip of MovieMaterial
- ▶ Streaming VOD from FMS for VideoMaterial
- ▶ Interpolating material color dynamically with DepthBitmapMaterial
- ▶ Creating normal maps in Photoshop
- ▶ Generating normal maps using Away3D NormalMapGenerator
- ▶ Getting your hands dirty with Pixel Bender materials
- ▶ Assembling composite materials

Introduction

In 3D computer graphics, sophisticated models alone are not of much use in terms of aesthetics and impact on user experience. If you want your 3D scene to look more impressive than an architectural sketch, you have got to use materials. **Materials** are the clothes of your mesh object. You have to be a skilful tailor if you want to achieve neat visual results.

Away3D has got a formidable toolset of materials compared to other Flash 3D engines. It ranges from simple color, up to cutting-edge Pixel Bender filters and shaded materials, which allow you to get astonishing results, bearing in mind that it is all done in Flash. However, with beauty comes the price you have to pay with CPU cycles. It is very easy to be tempted to pick up those materials that will supply the best visual effect. After all, most Flash applications are about user experience. That is where the pitfalls appear for the developer.

Wrapping all your scene objects with complex shaded materials will kill your frame rate. Some of the Away3D materials are quite useless when talking in terms of optimal performance, even though they deliver the most beautiful visual effect. In fact, most of the material may slow down your performance if you don't stick to special guidelines in relation to material creation.

There is no strict rule or single approach possible regarding how to create both impressive and highly optimized material. In most cases, you would count on your personal research and experiments. Nevertheless, in this chapter, we will learn several important techniques relating to texture creation. We will also see how to work with more advanced materials such as Dot3 and **Pixel Bender (PB)** material groups. Besides, we will explore less known features of some relatively basic materials, such as Movie and Video materials, and learn some advanced approaches to using them.

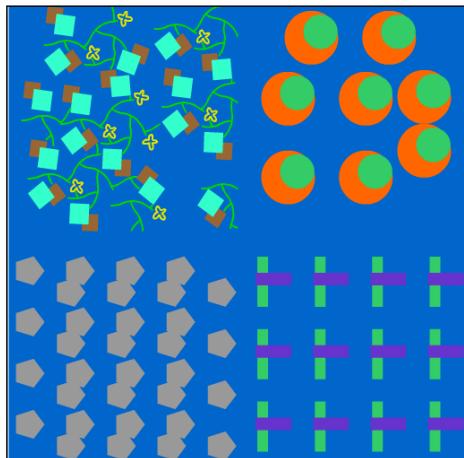
Using a single MovieClip for multiple materials

As we said, the optimization is an essential part of working with materials. Although the optimization is not a major topic of this chapter, you should know that some of the materials have built-in functionality that can help you in improving the overall performance of the application. Let's imagine the following scenario. You have a room in which the walls consist of planes. Each plane should have a different texture. Normally you would create one MovieMaterial that accepts a single MovieClip as its texture having four textures, in our cases, loaded into the application. Well, in a different case, you may have many more walls and each loaded texture adds to the overall application weight. MovieMaterial allows us to define a specific region of the given MovieClip to draw into the material BitmapData. This way, we can use a single MovieClip to supply several different textures for distinct materials.

Getting ready

1. Set up a basic Away3D scene using the AwayTemplate.
2. Open Adobe Flash IDE and create a MovieClip with the dimensions of "power of two". In this example, we will use a 256x256 pixels MovieClip named SplitMovie. The MovieClip area is divided into four regions containing different graphical representations so that we can use each of these for different materials.
3. Now convert the MovieClip to UIMovieClip using the Flex Component Kit command. This is done if the target environment is part of the Flex project (or you can just export it to the swc file).
4. Compile the movie in order to generate the swc file. This file is going to serve us as a texture library for Away3D MovieMaterial.

This is how the shape of the SplitMovie movie clp looks:



How to do it...

In the following program, we create planes to form a wall. Our goal is to apply MovieMaterial to each of them with a different texture using just one single source texture:

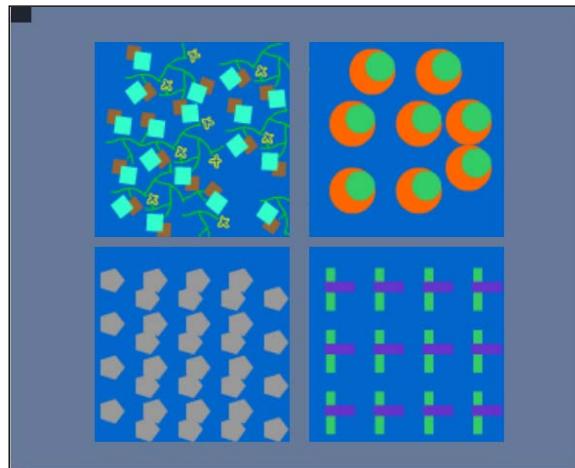
```
package
{
    public class MovieMaterialControl extends AwayTemplate
    {
        private var _movie:MovieClip=new SplitMovie();
        private var _movieMat1:MovieMaterial;
        private var _movieMat2:MovieMaterial;
        private var _movieMat3:MovieMaterial;
        private var _movieMat4:MovieMaterial;
        private var _plane1:Plane;
        private var _plane2:Plane;
        private var _plane3:Plane;
        private var _plane4:Plane;
        private var _planeContainer:ObjectContainer3D;
        public function MovieMaterialControl()
        {
            super();
        }
        override protected function initMaterials() : void{
```

```
_movieMat1=new MovieMaterial(_movie);_movieMat1.smooth=true;
_movieMat2=new MovieMaterial(_movie);_movieMat2.smooth=true;
_movieMat3=new MovieMaterial(_movie);_movieMat3.smooth=true;
_movieMat4=new MovieMaterial(_movie);_movieMat4.smooth=true;
_movieMat1.lockH=128;
_movieMat1.lockW=128;
_movieMat2.scaleX=2;
_movieMat2.scaleY=2;
_movieMat2.offsetX=-256;
_movieMat3.scaleX=2;
_movieMat3.scaleY=2;
_movieMat3.offsetY=-256;
_movieMat4.scaleX=2;
_movieMat4.scaleY=2;
_movieMat4.offsetY=-256;
_movieMat4.offsetX=-256;
}
override protected function initGeometry() : void{
    _planeContainer=new ObjectContainer3D();

    _plane1=new Plane({width:200,height:200,x:-110,y:105,yUp:false});
    _plane2=new Plane({width:200,height:200,x:110,y:105,yUp:false});
    _plane3=new Plane({width:200,height:200,x:-110,y:-
105,yUp:false});
    _plane4=new Plane({width:200,height:200,x:110,y:-
105,yUp:false});
    _plane1.material=_movieMat1;
    _planeContainer.addChild(_plane1);
    _plane2.material=_movieMat2;
    _planeContainer.addChild(_plane2);
    _plane3.material=_movieMat3;
    _planeContainer.addChild(_plane3);
    _plane4.material=_movieMat4;
    _planeContainer.addChild(_plane4);
    _view.scene.addChild(_planeContainer);
    _cam.z=-1000;

}
}
```

Here you can see four plane primitives using different parts of the MovieClip for their material source:



How it works...

Take a look at the `initMaterials()` function. All the texture manipulation is processed there. We tweak the `offsetX` and `offsetY` properties of `MovieMaterial`. We first created four different materials for four planes. Then we start offset texture `MovieClip` position inside each `MovieMaterial`. You can see that `_movieMat1` uses just the `lockH` and `lockW` properties. These properties allow you to define a region in the `MovieClip` to be used by the entire `MovieMaterial`. The only problem with this is that they lock regions only from the top-left corner of the texture `MovieClip` to the defined `lockH` and `lockW` values. In other words, if you want to define a region that starts at 128 pixels on the x-axis and 128 on the y-axis, you can't do it with `lockH` and `lockW`. For the `_movieMat1`, it works fine because:

- ▶ The first texture region starts at (0,0) of the `MovieClip`
- ▶ Its width and height equals 128 pixels, therefore `lockH` and `lockW` do their job perfectly

For the remaining three regions, we first scale the `MovieMaterial` twice because we need each region that is 128x128 by default to match a region of the entire `MovieClip`, which is 256x256. Then we offset the X and Y position of each `MovieClip` so that each texture region fills precisely the whole area of its material.

There's more...

`lockH` and `lockW` properties can serve additional purposes when used in conjunction with the `MovieMaterial` `clipRect` property. `clipRect` accepts a generic rectangle which defines the drawing areas for the `MovieMaterial` `MovieClip`. It allows you to define not just a texture region as we did in the previous example, but the visible area of `MovieMaterial` to be drawn on the model. In order for `clipRect` to work, you first need to lock the texture `MovieClip` using `lockH` and `lockW`. Then create an instance of the rectangle where you define an area for the `MovieMaterial` to draw. The area that falls outside a defined rectangle is clipped, resulting in the rest of the 3D object becoming transparent.

Controlling the MovieClip of MovieMaterial

Away3D `MovieMaterial` is much more powerful than it may seem at first glance. In fact, using it as a texture input only would be a sheer waste of its potential. As `MovieMaterial` uses a generic `MovieClip` object to draw a texture, it opens up a wide range of possibilities before us if we access the `MovieClip` itself. This way we can animate, interact, apply filters, and create additional special effects for your `MovieMaterial` on its `MovieClip` level. In the following example, we will learn how to trigger simple animation transition effects on `MovieMaterial`, which were defined in the resource `MovieClip`, during runtime.

Getting ready

1. Set up a basic Away3D scene using our usual `AwayTemplate`.
2. Also create a `Fla` file, open it in Flash IDE and create four `MovieClips` with different graphics. Position them on top of each other, and then create an alpha transition tween between them. Make sure each tween endpoint has a `stop()` command so that the transitions can't loop freely. Wrap the transition into a single `MovieClip` and assign it a name (in the example the name is `TweenedTexture`). Export it to `SWC` so that we can access it in Flash Builder (Flash users skip this step). Otherwise use the `MovieClip` from the `Fla` called `MatLib1.fla` that is specially created for this demo, which can be found in this chapter's assets folder that is present separately in the code bundle.
3. Put into your project utility class named `ButtonGraphics`, which can be found in the `utils` folder of this demo project. We use it to quickly create a set of simple buttons that will serve us by triggering animations.

Now we are ready to go.

How to do it...

As was said, we create three buttons—each one to trigger different animation on the target MovieClip. To test it visually, we create a simple Away3D plane and assign a MovieMaterial to it which uses our animated MovieClip:

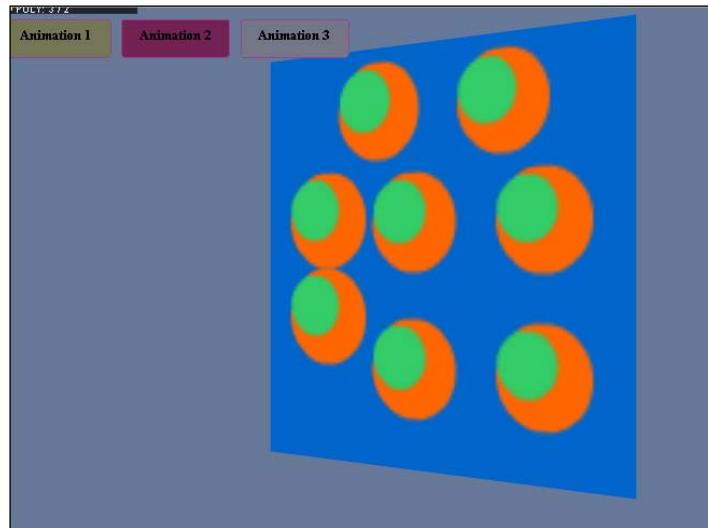
```
package
{
    public class MovieClipControl extends AwayTemplate
    {
        private var _movieClip:MovieClip;
        private var button1:SimpleButton;
        private var button2:SimpleButton;
        private var button3:SimpleButton;
        private var _movieMat:MovieMaterial;
        private var _plane1:Plane;
        public function MovieClipControl()
        {
            super();
            _cam.z=-500;
            initControls();
        }
        override protected function initMaterials() : void{
            _movieMat=new MovieMaterial(new TweenedTexture());
            _movieClip=_movieMat.movie as MovieClip;
            _movieMat.smooth=true;
        }
        override protected function initGeometry() : void{
            _plane1=new Plane({width:200,height:200,yUp:false,material:_movieMat});
            _plane1.bothsides=true;

            _view.scene.addChild(_plane1);
        }
        override protected function onEnterFrame(e:Event) : void{
            super.onEnterFrame(e);
            if(_plane1){
                _plane1.rotationY++;
            }
        }
        private function initControls():void{
            var buttonGraphics1:ButtonGraphics=new ButtonGraphics(0x747755,"Animation 1");
        }
    }
}
```

```
var buttonGraphics2:ButtonGraphics=new ButtonGraphics(0x742255, "Animation 2");
var buttonGraphics3:ButtonGraphics=new ButtonGraphics(0x747788, "Animation 3");
button1=new SimpleButton(buttonGraphics1,buttonGraphics1,buttonGraphics1,buttonGraphics1);
button2=new SimpleButton(buttonGraphics2,buttonGraphics2,buttonGraphics2,buttonGraphics2);
button3=new SimpleButton(buttonGraphics3,buttonGraphics3,buttonGraphics3,buttonGraphics3);
this.addChild(button1);button1.y=80;button2.y=80;button3.y=80;
this.addChild(button2);button2.x=button1.height+5;
this.addChild(button3);button3.x=button2.x+button2.height+5;
button1.addEventListener(MouseEvent.CLICK,onClick);
button2.addEventListener(MouseEvent.CLICK,onClick);
button3.addEventListener(MouseEvent.CLICK,onClick);

}
private function onClick(e:MouseEvent):void{
switch(e.target){
case button1:
_movieClip.gotoAndStop(1);
_movieClip.play();
break;
case button2:
_movieClip.gotoAndStop(20);
_movieClip.play();
break;
case button3:
_movieClip.gotoAndStop(40);
_movieClip.play();
break;
}
}
```

The following image depicts the plane with animated MovieMaterial. Press each of three buttons to interpolate between different animations:

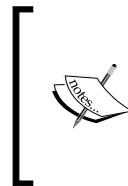


How it works...

After we created a MovieMaterial, we need to reference the MovieClip that it uses. We access the MovieClip using this code:

```
_movieClip=_movieMat.movie as MovieClip;
```

Now we can control the MovieClip of the material through the _movieClip variable. Using the onMouseClick() method, we execute different transitions that reside in the MovieClip, based on the clicked button. You can see that Away3D updates the material render on each frame so that we can see the animation process in real time.



If you have a complex scene with a lot of materials being animated, you can disable automatic material update using _movieMat.autoUpdate=false (true by default) and execute the update only when you need to with the update() method. This approach can save you a couple of precious frames to your overall frame rate.

Streaming VOD from FMS for VideoMaterial

Theoretically, VideoMaterial is pretty straightforward to set up when we talk about its source being a progressive loaded FLV file residing on the same domain. The matter becomes an issue when we wish to load a file from a different domain or especially when it is a streaming video from a media server such as FMS or RED5. It is known that Adobe began enforcing a stricter security sandbox policy for Flash Player from 9.0.115 version and onwards, resulting in many cases of security errors and preventing the streams to play on the client side when those requirements are ignored. In the case of progressive video, the solution is usually a simple one that rests upon the `crossdomain.xml` security file deployment. For streaming videos, in many cases, deployment of such a file is enough. The matter becomes completely different when the goal is to use streaming video as a material for Away3D. The problem arises from the fact that VideoMaterial draws the stream into the `BitmapData` object and this operation is treated as unsecure by the Flash Player, due to the threat of content theft. You would encounter the same problem trying to snapshot the video streamed from RTMP, which also requires the `bitmapdata.draw()` function. Fortunately there are workarounds for this issue which require few server-side as well as client-side settings.

In this recipe, you will see how to set up the video stream from the **video on demand (VOD)** feature of **Flash Media Server (FMS)**. We will configure the locally deployed FMS and code the client side so that the Away3D VideoMaterial is able to draw the video to bitmap texture.

You should know how to install and set up FMS. You also need basic working experience with FMS 3.0.

Getting ready

1. Download and install FMS 3.5 development server version (if you have a version higher than 3.0, it is fine too) from the Adobe site: <http://www.adobe.com/products/flashmediaserver/>.
2. Go to Flash Media Server 3.5\samples\applications\vod, copy the files Application.xml and main.asc, then paste them into C:\Program Files\Adobe\Flash Media Server 3.5\webroot\vod.
3. Select a video file from the sample video files and put it also in Flash Media Server 3.5\webroot\vod. I use a file named sample2_1000kbps.f4v depicting a train running in the pastoral countryside.
4. Open main.asc inside the application.onConnect() function. Uncomment these two lines:
 - `p_client.audioSampleAccess = "/" ;`
 - `p_client.videoSampleAccess = "/" ;`

This enables you to write the raw video and audio data of the stream.

5. Open Application.xml and add the following node inside<Client></Client>
 - <AudioSampleAccess enabled="true"/></AudioSampleAccess>
 - <VideoSampleAccess enabled="true"/></VideoSampleAccess>

Now your server side is ready.

How to do it...

The following program consists of three files:

1. **AwayScene:** Away3D environment setup class extending our well known AwayTemplate. Here we create a plane primitive, which will be assigned a VideoMaterial in a runtime, after we establish the connection with FMS. Also we add to it getter and setter for the VideoMaterial so that we can pass it in from outside the class.
2. **AwayHolder:** Extends UIComponent and serves us as flex container for AwayScene sprite instance.
3. **FMSVideoMaterial.mxml** is the main application that initiates the previously mentioned classes as well as sets up a connection to FMS:

```
AwayScene.as
package
{
    public class AwayScene extends AwayTemplate
    {
        private var _plane:Plane;
        private var _videoMat:VideoMaterial;
        public function AwayScene(stg:Stage=null)
        {
            super(stg);
        }
        override protected function initGeometry():void{
            _plane=new Plane({width:300,height:250,yUp:false});
            _plane.bothsides=true;
            _view.scene.addChild(_plane);
        }
        override protected function onEnterFrame(e:Event):void{
            super.onEnterFrame(e);
            if(_plane){
                _plane.rotationY++;
            }
        }
    }
}
```

```
        }
        public function get videoMat():VideoMaterial
        {
            return _videoMat;
        }

        public function set videoMat(value:VideoMaterial):void
        {
            _videoMat = value;
            _plane.material=_videoMat;
        }
    }
```

AwayHolder.as

```
package
{
    public class AwayHolder extends UIComponent
    {
        private var _away:AwayScene;

        public function AwayHolder()
        {
            super();
        }
        override protected function createChildren():void
        {
            super.createChildren();
            _away=new AwayScene();
            addChild(_away);
        }

        override protected function updateDisplayList(unscaledWidth:Number,
                                                    unscaledHeight:Number):void
        {
            super.updateDisplayList(unscaledWidth, unscaledHeight);

        }
        public function get awayScene():AwayScene{
            return _away;
        }
    }
}
```

```
FMSVideoMaterial.mxml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/
mxml" layout="absolute" minWidth="955" minHeight="600"
creationComplete="init()">
    <mx:Script>
        <![CDATA[
            private var _nc:NetConnection=new NetConnection();
            private var _ns:NetStream;
            private var _videoMat:VideoMaterial;
            private var _container3D:AwayHolder;
            private var _vid:Video;
            private function init():void{
                _nc.addEventListener(NetStatusEvent.NET_
STATUS,onNetStatus);
                _nc.addEventListener(IOErrorEvent.IO_ERROR, onIOError);
                _nc.connect("rtmp://127.0.0.1/vod");
            }
            private function onNetStatus(e:NetStatusEvent):void{
                if(e.info.code=="NetConnection.Connect.
Success") {

                    initStream();
                    initAway3D();
                }
            }
            private function initAway3D():void{
                _container3D=new AwayHolder();
                _container3D.width=800;
                _container3D.height=600;
                this.addChild(_container3D);
                _videoMat=new VideoMaterial();
                _container3D.awayScene.videoMat=_videoMat;
                _videoMat.autoUpdate=false;
                _videoMat.nc=_nc;
                _videoMat.netStream=_ns;
                _videoMat.file="mp4:sample2_1000kbps.f4v";
                _videoMat.video=_vid;
                _videoMat.play();
                setTimeout(timedFunction,1000);
            }
            private function initStream():void{
                _vid=new Video();
                _ns=new NetStream(_nc);
```

```
var client:Object = new Object();
client.onBWDone = onBWDone;
client.onMetaData = onMetaData;
_nc.client=client;
_ns.client=client;
}

private function timedFunction():void{
try {
    _videoMat.autoUpdate=true;

}
catch (error:Error)
{
    trace(error);
}

}

//////Needed for NetStream
private function onMetaData(infoObject:Object):void{

/////////Needed for NetConnection
public function onBWDone(...args):void{

private function onIOError(e:IOErrorEvent):void{
} ]>
</mx:Script>
</mx:Application>
```

Here you can see the resulting plane object with the VideoMaterial on it, streaming video source from FMS:



How it works...

We will concentrate on the FMS connection and the `VideoMaterial` setup in the main application as that is where the action happens.

In the `init()` function, we created a `NetConnection` to the FMS via the localhost RTMP address. Then in the `initStream()` method, we set a video object to hold the video and the `NetStream` responsible for the VOD file streaming. Here you should notice one important detail—we assign an object called `client` to the `client` property of both `NetConnection` and `NetStream` instances, which holds references to two event handlers, namely, `onMetaData()` and `onBWDone`. The reason for this is that when `NetConnection` connects to the server, it receives a call-back that looks at the client side for a handler called `onBWDone`. The same is the case with `NetStream`, which after receiving the stream needs to access an event handler called `onMetaData()`. Not defining these will result in runtime errors.

Now, after the `NetStream` is ready, we initiate our Away3D scene and create a `VideoMaterial` instance in the `initAway3D()` method. Now comes the tricky part. In order to start streaming the video into the material, we pass to it our `NetConnection` instance (`_nc`), `NetStream` (`_ns`), and the video object (`_video`). We also pass the server-side video filename with a prefix `mp4` that serves as video format indicator. However, the crucial line here is `videoMat.autoUpdate=false`. The problem is that if we trigger the `play()` method of `VideoMaterial` right away with `autoUpdate` enabled, we will get a security sandbox violation error. With `autoUpdate=true`, the material attempts to draw an empty stream to `bitmapdata`, which seems to cause that security error. Therefore, we first disable `autoUpdate`, then trigger the material's `play()` function. After that, we set timeout to just one second in order to let the video start streaming. Upon timeout end in the `timedFunction()`, we switch `autoUpdate` to true again and now the material is able to draw the video to `bitmapdata` with no errors. In some cases, this step will be unnecessary. However, if you see that after implementing all the configurations mentioned previously, you keep receiving sandbox security errors, then this hack is your last hope.

Interpolating material color dynamically with DepthBitmapMaterial

The Away3D material bank has got one interesting material called `DepthBitmapMaterial`. The purpose of this material is to add a diffuse color to the bitmap texture of the material so that the amount of fill for that color depends on the minimum and maximum z-coordinate of the mapped object. `DepthBitmapMaterial` may be useful when you wish to add various amounts of color tint dynamically to the bitmap texture of several 3D objects located at different depth, but using the same material. Additionally, you can also create color transforms if you define different color values for `minColor` and `maxColor`. Let's see how to get it done.

Getting ready

Create a basic Away3D scene using `AwayTemplate`. You will also need a tweening engine, which in our case is `TweenMax`, but you can use one of yours instead as well.

How to do it...

In the following program, we create a simple `Sphere` primitive which gets assigned `DepthBitmapMaterial`. Then we tween it from the default position to the maximum depth in order to see the dynamic color fill overlaying the texture map:

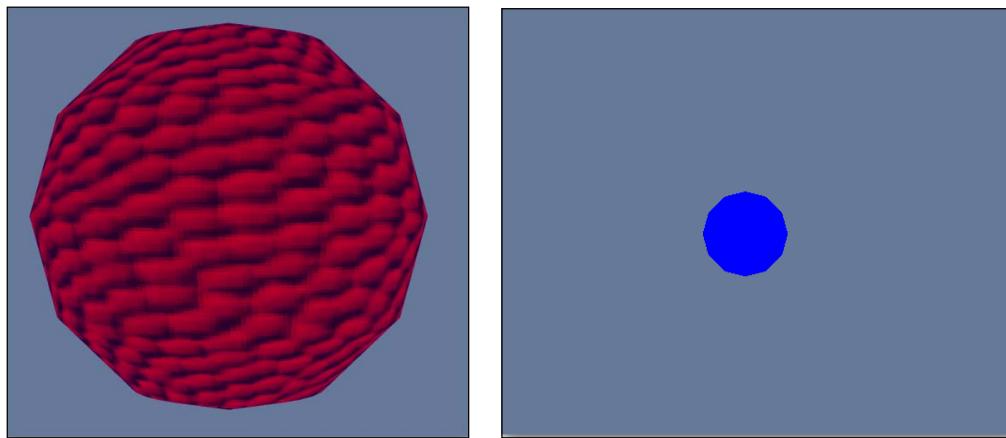
`DepthMaterialDemo.as`

```
package
{
    public class DepthMaterialDemo extends AwayTemplate
    {
        [Embed(source="assets/SimpleBitmap1.png")]
        private var SimpleBitmap:Class;

        private var _depthMat:DepthBitmapMaterial;
        private var _sp:Sphere;
        public function DepthMaterialDemo()
        {
            super();
            _cam.z=-200;
            initTween();
        }
        override protected function initMaterials() : void{
            _depthMat=new DepthBitmapMaterial(Cast.bitmap(new
SimpleBitmap()));
            _depthMat.smooth=true;
            _depthMat.minZ=0;
            _depthMat.maxZ=1200;
            _depthMat.minColor=0xFF0000;
            _depthMat.maxColor=0x0000FF;
        }
        override protected function initGeometry() : void{
            _sp=new Sphere({radius:60,segmetsH:30,segmentsW:30,material:_depthMat});
            _view.scene.addChild(_sp);
        }
    }
}
```

```
_sp.z=0;  
  
}  
override protected function onEnterFrame(e:Event) : void{  
    super.onEnterFrame(e);  
}  
private function initTween():void{  
    TweenMax.fromTo(_sp,4,{z:0},{z:1200,ease:Bounce.easeIn,onComplet  
e:onTweenComplete});  
  
}  
  
private function onTweenComplete():void{  
    initTween();  
}  
  
}  
}
```

In the left image, the ball is at the default position having a red color overlay. When moving further from the camera, it interpolates with the blue color .Eventually, in the right image, the sphere gets the full blue tint at a distance of 1200 points from the camera:



How it works...

Inside the `initMaterials()` method, we set up `DepthBitmapMaterial` passing into its constructor `image` to serve as texture bitmap:

```
_depthMat=new DepthBitmapMaterial(Cast.bitmap(new  
SimpleBitmap()));
```

Now at the position $z = 0$, we want the sphere to have a red color tint and at $z=1200$, it gets completely filled with blue color. The z range is defined by the two properties:

```
_depthMat.minZ=0;  
_depthMat.maxZ=1200;
```

Next we set `minColor` to pure red in order to have a red tint when the object is located at $z=0$, whereas `maxColor` receives a pure blue hexadecimal for $z=1200$:

```
_depthMat.minColor=0xFF0000;  
_depthMat.maxColor=0x0000FF;
```

Creating normal maps in Photoshop

Normal maps, in most simple words, give us the ability to cheat the client. It helps to create an effect of a high detailed mesh to low detailed (low-poly) geometry with the help of surface light direction calculation based on the color data embedded into the normal map. You can achieve astonishing visual detail level for a mesh consisting merely of a couple of hundreds of triangles or even less, applying to it a normal map.

There are different ways to generate a normal map for Away3D. The most professional and best resulting approach is to render a normal map in your favorite 3D modeling software such as Autodesk Maya, 3dsMax, or Blender, based on mesh detail. However, it requires solid experience in 3D modeling and UV mapping. But no reason to panic—there are easier ways to have it done. You can use plugins such as NVIDIA NormalMapFilter to render normal maps based on pixel data of an image. Away3D engine has a `NormalMapGenerator` utility class that renders a normal map from a mapped input geometry during runtime. At last, Prefab3D has got a robust visual normal map generator which produces wonderful quality and is easy to use.

In this recipe, we will learn how to create normal maps within Photoshop using NVIDIA NormalMapFilter.

For more information on normal maps and their practical use, visit the following links:

http://en.wikipedia.org/wiki/Normal_mapping

Getting ready

Generating normal maps with NVIDIA NormalMapFilter:

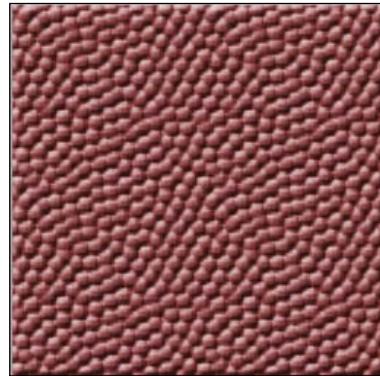
1. It is assumed that you downloaded the plugin from the NVIDIA site (http://developer.nvidia.com/object/photoshop_dds_plugins.html) and installed it into Adobe Photoshop.
2. Open Adobe Photoshop and create a new file with the width and height of 256 pixels.

Now we are ready to create a normal map with a bump effect of the Away3D sphere primitive.

How to do it...

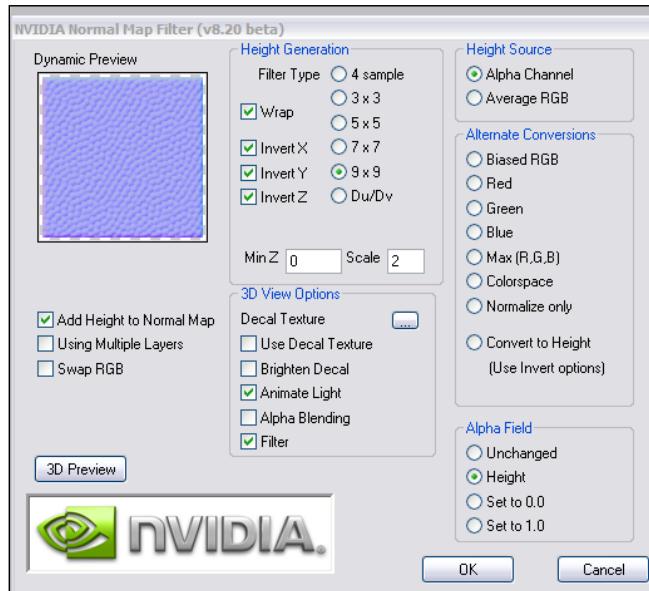
In Photoshop, first create an image with the dimensions of 256x256. I filled it with one of the pre-built image effect styles of Photoshop that has bumpy graphics.

The texture image created with the effect styles preset inside Photoshop looks as shown in the following image:



Now desaturate it to a grayscale (don't convert to grayscale mode as the plugin needs all four channels for processing). Also, it is good to adjust the levels to get a deeper bump detail.

Open the plugin interface from the **Filters** drop-down menu, as shown in the following screenshot:



The optimal settings are those displayed on the previous image (NVIDIA Normal Map Filter settings dialog opened inside Photoshop).

You can use a real-time shaded preview of your normal map, clicking the **3D Preview** button located in the plugin dialog window right above the NVIDIA logo.

Click the **OK** button to generate the normal map. Then export it as **.jpg** or **.png** using the Photoshop.

Now your map is ready to be imported into an Away3D scene.

Generating normal maps using Away3D NormalMapGenerator

Now let's see how we can generate a normal map inside Away3D using the `NormalMapGenerator` class located in the `materials.utils` package.

Getting ready

Create a basic Away3D scene using `AwayTemplate`. Then make sure you load or embed `SimpleBitmap1.png` and `HeightMap.png` image files, which are located in this chapter's assets folder.

How to do it...

`NormalMapGen.as`

```
package
{
    public class NormalMapGen extends AwayTemplate
    {
        [Embed(source="assets/SimpleBitmap1.png")]
        private var SimpleTexture:Class;
        [Embed(source="assets/HeightMap.png")]
        private var HeightMap:Class;
        private var _normalMap:BitmapData;
        private var _normThumb:Bitmap;
        private var _pointLight:PointLight3D;
        private var _sp:Sphere;
        private var _tField:TextField;
        public function NormalMapGen()
        {
    
```

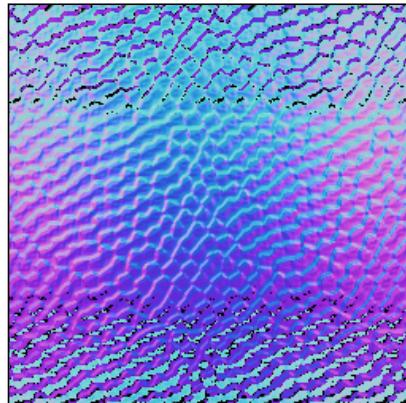
```
super();
setProgressText();
genNormalMap();

}

private function genNormalMap():void{
    var normalGen:NormalMapGenerator=new NormalMapGenerator(_sp
,256,256,Cast.bitmap(new HeightMap()),0,true,50,NormalMapType.TANGENT_
SPACE,false);
    normalGen.addEventListener(TraceEvent.TRACE_
COMPLETE,onNormalGenReady);
    normalGen.addEventListener(TraceEvent.TRACE_
PROGRESS,onTraceWorking);
    normalGen.execute();
}
private function setProgressText():void{
    _tField=new TextField();
    var tFormat:TextFormat=new TextFormat();
    tFormat.size=25;
    tFormat.bold=true;
    tFormat.align="center";
    _tField.defaultTextFormat=tFormat;
    _view.addChild(_tField);
    _tField.width=250;
    _tField.x=-(_tField.width/2);
    _tField.y=-280;
}
private function onTraceWorking(e:TraceEvent):void{
    trace(e.percent);
    _tField.text=Math.floor(e.percent).toString();
}
private function onNormalGenReady(e:TraceEvent):void{
    _tField.text="Normal Map is ready!";
    var normalGen:NormalMapGenerator=e.target as NormalMapGenerator;
    _normalMap=normalGen.normalMap;
    _normThumb=new Bitmap(_normalMap);
    _view.addChild(_normThumb);
    _normThumb.x=-400;
    _normThumb.y=-300;

}
}
```

Away3D generates a normal map result image as follows:



How it works...

In the `initGeometry()` function, we first create a `Sphere` primitive. We then add to it a `BitmapMaterial` containing the texture we want to create a normal map from. Then in the `genNormalMap()` method, we instantiate the `NormalMapGenerator` by passing the following arguments into its constructor:

```
var normalGen:NormalMapGenerator=new NormalMapGenerator(_sp  
,256,256,Cast.bitmap(new HeightMap()),0,true,50,NormalMapType.TANGENT_  
SPACE,false);
```

The first parameter is our 3D object, then we set the normal map resulting bitmap dimensions. Usually the bigger the size, the higher is the quality of the map. However, one must be careful not to overdo it because of performance considerations. Next is an optional parameter for the height map. The height map is usually a grayscale image ranging from black to white and representing height information where the complete black is deepest and full white indicates the highest region. In this example, we want to get a bumpy relief effect for our model. So we define a height map, which is just a grayscale version of the sphere default texture, to embed bump information into our normal map. The rest of the parameters can be left to default except the generation type. Here you have got two options to generate a normal map using object space or tangent space. Without diving into the mathematical explanation of these methods and their differences, we can assume that `TANGENT_SPACE` is considered more precise as well as the more flexible type. Tangent space-generated normal maps can be used for mapping different objects as they are not dependent on the specific object form in comparison to object space-based ones. Also if you consider applying mesh deformation to the target object, tangent normal maps don't get distorted.

The generation process is asynchronous, and sometimes can take a decent amount of time to complete. Therefore, we assign two event listeners to `NormalMapGenerator`:

```
normalGen.addEventListener(TraceEvent.TRACE_COMPLETE, onNormalGenReady);  
normalGen.addEventListener(TraceEvent.TRACE_PROGRESS, onTraceWorking);
```

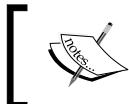
`TraceEvent.TRACE_COMPLETE` is fired when map generation has finished. You must define this event listener as only after it is dispatched, can you proceed with extracting the ready normal `bitmapdata` from the class. `TraceEvent.TRACE_PROGRESS` is an optional but very useful event which helps you to track the generation progress. In this example, we use it to view actual tracing progress in the text field.

`TraceEvent.TRACE_COMPLETE` triggers on the `NormalGenReady()` event handler. Here we access the normal map `bitmapdata` via `_normalMap=normalGen.normalMap`.

From here, you can save it to a file or assign to any material that supports normal maps during runtime.

There's more...

Away3D has got another useful utility class found in the `materials.utils` package and called `NormalBumpMaker`. As the name suggests, it allows us to generate a normal map on the fly from a supplied bump map. This class may serve as a good alternative to the `NormalMapGenerator` because it processes the `bitmapdata` much faster and synchronously.



Additional usage of `NormalBumpMaker` is to add a bump detail to a given normal map. To do that, use the `addBumpsToNormalMap()` function of the class.

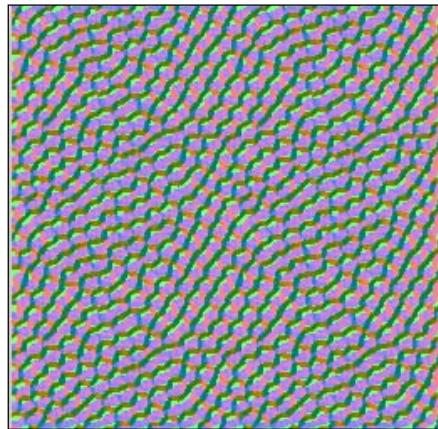


Generating a normal map using the `NormalBumpMaker` utility

Let's add a new function to the previous program right above `genNormalMap()`:

```
private function genNormalViaBump():void{  
    var bumpToNorm:NormalBumpMaker=new NormalBumpMaker();  
  
    var normMap:BitmapData=bumpToNorm.convertToNormalMap(Cast.  
bitmap(new HeightMap()),null,0.8);  
    _normThumb=new Bitmap(normMap);  
    _view.addChild(_normThumb);  
    _normThumb.x=-400;  
    _normThumb.y=-300;  
}
```

Comment the `genNormalMap()` function call in the constructor method of `NormalMapGen`. Add the call to the `genNormalViaBump()` function instead of `genNormalMap()`. Here is the resulting normal map generated using the `NormalBumpMaker` utility:



First we instantiate the `NormalBumpMaker` class. Then we call the following function:

```
bumpToNorm.convertToNormalMap(Cast.bitmap(new HeightMap()),null,0.8);
```

For the bump map, we pass the `HeightMap` that we embedded in the previous example. A height map does not always serve as a bump, but in our case, the height detail is identical to the desired bump. The second parameter is for geometrical function where you can define a customized mapping from the UV coordinates of the bump map to (x, y, and z) coordinates of a particular model. The third parameter is an amount of bump amplitude to be embedded into the normal map. The higher the value, the more detailed the normal map becomes. The default value is 0.05, but you can experiment with it to achieve a desired result.

See also

In Chapter 8, *Prefab3D*, refer to the recipe *Normal Mapping in Prefab*.

Getting your hands dirty with Pixel Bender materials

The Away3D materials package has got a unique group of materials with one common feature which is Pixel Bender shaders. It is easy to identify these materials. Each class in the materials package that has, in the middle of its name, the letters "PB" or "MultiPass" uses Pixel Bender filters for shading. Unlike other shaded materials that use image processing algorithms created with ActionScript, PB materials use external filters which were written and compiled using the Adobe Pixel Bender toolkit.

Pixel Bender shading is more sophisticated and creates much more impressive visual results than the regular shading in Away3D. Pixel Bender processes sampling each pixel of an input bitmap data. In Away3D in order to get the correct 3D shading effect on top of the model surface, the shading process executes on the Texel (texture element) level, which is a basic unit of texture just like a pixel for an image. If you want to learn more about Pixel Bender shaders in Away3D, you can open those classes and see how they work. Also, all the Pixel Bender shaders binaries as well as the source files are included in the `pbks` package of the library trunk. Opening them with the Pixel Bender toolkit will allow you to understand how they work and customize them to your specific needs.



You should remember that the PB materials consume a significant amount of memory, therefore it is not recommended to use them for multiple objects and complex scenes. However, they can be very handy in scenarios where you have a simple scene that needs to deliver short and immediate impact on the user, such as an intro scene or demo application.

In this example, you will learn how to apply some of PB materials as well as their fine-tuning. In fact, most of the materials of this group have the same setup. The main difference is the light source type and some additional parameters which will be also explained here.

Getting ready

Set up a basic Away3D scene using `AwayTemplate`. We also embed two images—`SimpleBitmap1.png` and `NormalMapPthshop.png`: one for the material texture and another is its normal map. They can be found in the `assets` folder. Each PB material requires these two kinds of maps by default.

How to do it...

In this program, we begin with `PhongPBMaterial` which we apply to a sphere. Also notice that you must add at least one light source to the scene, which uses any kind of shading, otherwise it will usually result in a black spot:

```
PBMaterials1.as

package
{
    public class PBMaterials1 extends AwayTemplate
    {
        [Embed(source="assets/SimpleBitmap1.png")]
        private var SimpleTexture:Class;
        [Embed(source="assets/NormalMapPthshop.png")]
        private var NormalReady:Class;
```

```
[Embed(source="assets/HeightMap.png")]
private var HeightMap:Class;
[Embed(source="assets/Sunset.jpg")]
private var EnvMap:Class;
private var _pbmat:PhongPBMaterial;
private var _normalMap:BitmapData;
private var _defaultMap:BitmapData;
private var _pointLight:PointLight3D;
private var _sp:Sphere;
private var _canRotate:Boolean=false;
private static const DIST:Number=200;
private var _angle:Number=0;
private static const DEGR_TO_RADS:Number=Math.PI/180;
private var _hoverCam:HoverCamera3D;
private var _lastPanAngle:Number;
private var _lastTiltAngle:Number;
private var _lastMouseX:Number;
private var _lastMouseY:Number;
private var _canRotateCam:Boolean=false;
public function PBMaterials1()
{
    super();
    setHoverCam();
    initSceneLights();

}

override protected function initListeners() : void{
super.initListeners();
stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
}
override protected function initMaterials() : void{
_sp=new Sphere({radius:50,segmentsH:22,segmentsW:22});
_defaultMap=Cast.bitmap(new SimpleTexture());
_normalMap=Cast.bitmap(new NormalReady());
_envMap=Cast.bitmap(new EnvMap());

_pbmat=new PhongPBMaterial(_defaultMap,TangentToObjectMapper.
transform(_normalMap,_sp,true),_sp);///
_pbmat.smooth=true;
_pbmat.gloss=3;
_pbmat.specular=5;
```

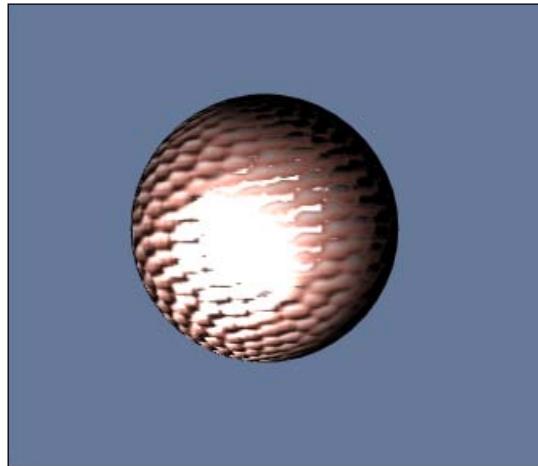
```

        _canRotate=true;
    }
    override protected function initGeometry():void{
        _sp.material=_pbmat;
        _view.scene.addChild(_sp);
        _sp.z=500;
    }
    private function initSceneLights():void{
        _pointLight=new PointLight3D();
        _pointLight.radius=120;
        _pointLight.brightness=1.4;
        _view.scene.addLight(_pointLight);
        _pointLight.x=-50;
        _pointLight.y=60;
        _pointLight.z=470;
    }
    override protected function onEnterFrame(e:Event) : void{
        super.onEnterFrame(e);
        if(_canRotate){
            _sp.rotationY++;
            _angle++;
            _pointLight2.x = _sp.position.x+ Math.sin(_angle*DEGR_TO_
RADS)*DIST;
            _pointLight2.y = 0;
            _pointLight2.z = _sp.position.z+Math.cos(_angle*DEGR_TO_
RADS)*DIST;
        }
        if (_hoverCam) {
            if(_canRotateCam){
                _hoverCam.panAngle = 0.5 * (stage.mouseX- _lastMouseX) +
                _lastPanAngle;
                _hoverCam.tiltAngle = 0.5 * (stage.mouseY - _lastMouseY) +
                _lastTiltAngle;
            }
            _hoverCam.hover();
        }
    }
    private function onMouseDown(e:MouseEvent):void{
        _canRotateCam=true;
        _lastPanAngle = _hoverCam.panAngle;
        _lastTiltAngle = _hoverCam.tiltAngle;
        _lastMouseX = stage.mouseX;
        _lastMouseY = stage.mouseY;
    }
}

```

```
        }
        private function onMouseUp(e:MouseEvent):void{
            _canRotateCam=false;
        }
        private function setHoverCam():void{
            _hoverCam=new HoverCamera3D();
            _hoverCam.minTiltAngle = -80;
            _hoverCam.maxTiltAngle = 20;
            _hoverCam.panAngle = 90;
            _hoverCam.tiltAngle = 0;
            _hoverCam.distance=500;
            _view.camera=_hoverCam;
            _hoverCam.target=_sp;
        }
    }
}
```

What we get is the following PhongPBMaterial in action:



How it works...

First you should know one important thing concerning the lights. PB materials are divided into two groups in this case. Regular PB materials such as PhongPBMaterial, DiffusePBMaterial, FresnelPBMaterial, and SpecularPBMaterial use just one single light source which must be of the type PointLight3D. If you wish to apply shading from several light sources, you should use the MultiPass materials type such as PhongMultiPassMaterial, SpecularMultiPassMaterial, and so on. Notice that when using MultiPass material, you can also add a DirectionalLight3D light source.

In the preceding example, we created one `PointLite3D` that orbits around the sphere and nicely lightens the object surface. In the `initMaterials()` function, we first instantiate the `bitmapdata` objects for diffuse and normal map input of the material:

```
_defaultMap=Cast.bitmap(new SimpleTexture());
_normalMap=Cast.bitmap(new NormalReady());
```

Then we instantiate the `PhongPBMaterial` constructor passing in our regular (diffuse) map, normal map, and the shaded object itself:

```
_pbmat=new PhongPBMaterial(_defaultMap,TangentToObjectMapper.
transform(_normalMap,_sp,true),_sp);
```

Here I use the `TangentToObjectMapper` utility to convert normal map calculations from Tangent to Object space. This transformation is necessary as the `PixelBender` shaders require Object space normal maps.



The difference between **Tangent** and **Object** space is explained in the recipe *Generating normal maps using Away3D NormalMapGenerator*.



Doing this transformation causes the shading effect to be visually less flat and more accurate in this case.

Play with specular and gloss properties of `PhongPBMaterial` to get different light reflection results. Also you can define a whole specular map via the `specularMap` property to define a specular pattern for the mesh.

There's more...

Now let's see how we can enrich the shading effect we created previously with more light. We will add more lights and use `PhongMultiPassMaterial`.

In the `initSceneLights()` function, add one `DirectionalLight3D` and two more `PointLight3D` instances:

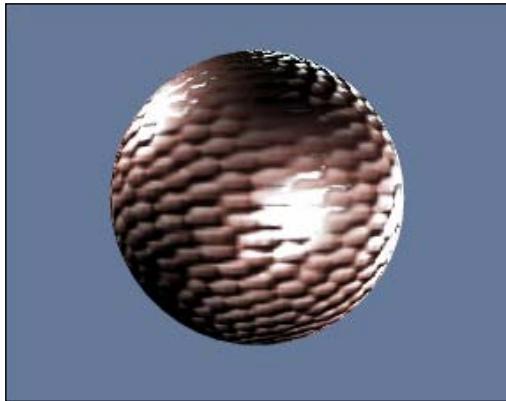
```
_dirLight=new DirectionalLight3D();
_dirLight.color=0xAFF7FF;
_dirLight.diffuse=2;
_dirLight.brightness=0.02;
_dirLight.specular=1;
_view.scene.addLight(_dirLight);
_dirLight.direction=_sp.position;
_pointLight1.brightness=1.4;
_pointLight1.x=30;
_pointLight1.y=0;
```

```
_pointLight1.z=430;  
_pointLight1.radius=120;  
_pointLight2=new PointLight3D();  
_view.scene.addLight(_pointLight2);  
_pointLight2.brightness=2;  
_pointLight2.x=68;  
_pointLight2.y=20;  
_pointLight2.z=440;  
_pointLight2.radius=120;
```

The material setup is exactly the same as with PhongPBMaterial. Add the following line in the `initMaterials()` method and don't forget to declare a global variable for the `_multiPass` instance:

```
_multiPass=new PhongMultiPassMaterial(_defaultMap,TangentToObjectMapper.r.transform(_normalMap,_sp,true),_sp);  
_multiPass.smooth=true;  
_multiPass.gloss=3;
```

At this rendering, you can see three light sources reflected from the sphere's surface which use PhongMultiPassMaterial:



Now you can see two static light sources' reflections on the sphere surface with a soft back light from the `DirectionalLight3D`.

Exploring Fresnel Shader

Let's explore one more shader called `FresnelPBMaterial`. **Fresnel** shaders are a well known type in 3D graphics. Fresnel shader produces a reflection of the environment map onto the object surface. However, the difference from a regular `EnviroBitmapMaterial` that the reflection amount depends on the viewing angle. This way, if you look straight at the object, the surface area which is perpendicular to your view direction vector has no reflection at all. This material can be especially useful to create water effects.

First, let's add another bitmap asset which will serve as an environment map. I use a sample image of the Windows XP default image set found in the `My Pictures` folder. Add the following to the global variables block:

```
[Embed(source="assets/Sunset.jpg")]
privatevar EnvMap:Class;
```

In the `initMaterials()` method, instantiate the `BitmapData` for the environment map:

```
_envMap=Cast.bitmap(new EnvMap());
```

Next, we create the material by inserting this code:

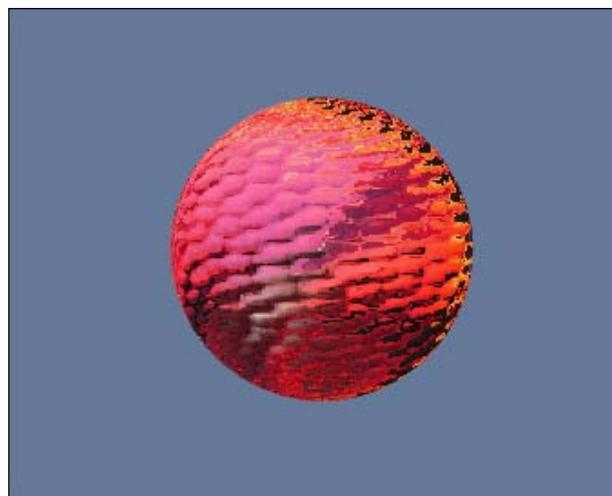
```
_fresnel=new FresnelPBMaterial(_defaultMap,TangentToObjectMapper.
transform(_normalMap,_sp,true),_envMap,_sp);
_fresnel.envMapAlpha=0.9;
_fresnel.exponent=10;
```

In the constructor, we pass a regular diffuse map, normal map, and environment map, which is then reflected. We also pass the 3D object itself as with the rest of the PB materials.

Although `FresnelPBMaterial` is a shader, it doesn't need a scene light source. Instead, it uses the environment map.

We configure two important properties here. `envMapAlpha` is responsible for the visible amount of environment map reflection. `Exponent` property sets the factor of reflection strength in relation to the view angle. The steeper angle areas will get better reflection if the value is lower.

The following is the resulting Fresnel shaded sphere:



Assembling composite materials

In Away3D, it is possible to stack several materials one upon another into one using `CompositeMaterial`, which can give you very interesting visual results, if composed properly. `CompositeMaterial` allows you to add an unlimited amount of different materials which must extend `LayerMaterial`. You can produce very cool patterns by doing the following:

- ▶ Adjusting each nested material's blend mode
- ▶ Adding several different materials with transparent background resulting in brand new texture composition

In this example, we will create a `CompositeMaterial` combining two different `BitmapMaterials`, which by means of a simple blend mode adjustment will give us a nice shaded-like looking material effect.

Getting ready

Set up a new Away3D scene using `AwayTemplate`. Make sure you embed the `PuzzleTile.png` image file into your code, which can be found in this chapter's assets folder. It will serve as a texture for one of the materials.

How to do it...

In this program, we will create two materials of type `BitmapMaterial`. Then we combine them by adding to `CompositeMaterial`. We then use a `TorusKnot` primitive to apply the resulting `CompositeMaterial` to it:

`CompositeMix.as`

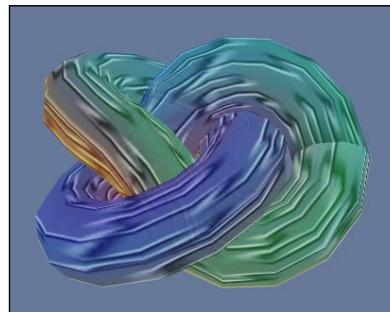
```
package
{
    public class CompositeMix extends AwayTemplate
    {
        [Embed(source="assets/PuzzleTile.png")]
        private var PuzzleTile:Class;

        private var _compose:CompositeMaterial;
        private var _bitMat:BitmapMaterial;
        private var _bitMat1:BitmapMaterial;
        private var _knot:TorusKnot;
        public function CompositeMix()
        {
            super();
            _cam.z=-500;
```

```
    }
    private function createPerlinMap():BitmapData{
        var bdata:BitmapData=new BitmapData(128,128);
        bdata.perlinNoise(128,128,18,12,true,true,7,false);
        return bdata;
    }
    override protected function initMaterials() : void{
        _compose=new CompositeMaterial();
        _bitMat=new BitmapMaterial(createPerlinMap());
        _bitMat.smooth=true;
        _bitMat1=new BitmapMaterial(Cast.bitmap(new PuzzleTile()));

        _bitMat1.smooth=true;
        _bitMat1.alpha=0.9;
        _compose.addMaterial(_bitMat1);
        _compose.addMaterial(_bitMat);
        _bitMat1.blendMode=BlendMode.HARDLIGHT;

    }
    override protected function initGeometry() : void{
        _knot=new TorusKnot({material:_compose});
        _knot.segmentsR=45;
        _knot.segmentsT=9;
        _knot.radius=60;
        _knot.tube=25;
        _view.scene.addChild(_knot);
    }
    override protected function onEnterFrame(e:Event) : void{
        super.onEnterFrame(e);
        if(_knot){
            knot.rotationX++;
        }
    }
}
}
```



The resulting composition of materials applied to `TorusKnot` adds a shaded look to its surface.

How it works...

We set up two different textures as input for two instances of `BitmapMaterial`. For `_bitMat`, we generate the `perlinNoise` map, which is going to produce a shading effect of our `CompositeMaterial`. `_bitMat1` accepts a texture that consists of puzzle lines whose purpose is to blend with the `perlinNoise` map of `_bitMat`. We set the blend mode for `_bitMat1` by writing this line:

```
_bitMat1.blendMode=BlendMode.HARDLIGHT;
```

This type of blend intensifies and saturates the colors of the composed textures. Perlin noise random gradient transitions blend into the `PuzzleTile` texture background, thus giving a smooth multicolor shade to the composition.

2

Working with Away3D Cameras

In this chapter, we will cover:

- ▶ Creating an FPS controller
- ▶ Creating Camera Depth of Field Effect
- ▶ Detecting whether an object is in front of or behind the camera
- ▶ Changing lenses
- ▶ Following a third-person view with a spring camera
- ▶ Tracking screen coordinates of 3D objects
- ▶ Transforming objects in 3D space relative to the camera position
- ▶ Using Quaternion camera transformations for advanced image gallery viewing

Introduction

Cameras are an absolutely essential part of the 3D world of computer graphics. In fact, no real-time 3D engine can exist without having a camera object. Cameras are our eyes into the 3D world.

Away3D has a decent set of cameras, which at the time of writing, consists of `Camera3D`, `TargetCamera3D`, `HoverCamera3D`, and `SpringCam` classes. Although they have similar base features, each one has some additional functionality to make it different. Throughout this chapter, we will see the uniqueness of each camera and the mutual differences among them.

Creating an FPS controller

There are different scenarios where you wish to get a control of the camera in first person, such as in FPS video games. Basically, we want to move and rotate our camera in any horizontal direction defined by the combination of x and y rotation of the user mouse and by keyboard keys input. In this recipe, you will learn how to develop such a class from scratch, which can then be useful in your consequential projects where FPS behavior is needed.

Getting ready

Set up a basic Away3D scene extending `AwayTemplate` and give it the name `FPSDemo`. Then, create one more class which should extend `Sprite` and give it the name `FPSController`.

How to do it...

`FPSController` class encapsulates all the functionalities of the FPS camera. It is going to receive the reference to the scene camera and apply FPS behavior "behind the curtain".

`FPSDemo` class is a basic Away3D scene setup where we are going to test our `FPSController`:

`FPSController.as`

```
package utils
{
    public class FPSController extends Sprite
    {
        private var _stg:Stage;
        private var _camera:Object3D
        private var _moveLeft:Boolean=false;
        private var _moveRight:Boolean=false;
        private var _moveForward:Boolean=false;
        private var _moveBack:Boolean=false;
        private var _controllerHeigh:Number;
        private var _camSpeed:Number=0;
        private static const CAM_ACCEL:Number=2;
        private var _camSideSpeed:Number=0;
        private static const CAM_SIDE_ACCEL:Number=2;
        private var _forwardLook:Vector3D=new Vector3D();
        private var _sideLook:Vector3D=new Vector3D();
        private var _camTarget:Vector3D=new Vector3D();
        private var _oldPan:Number=0;
        private var _oldTilt:Number=0;
        private var _pan:Number=0;
```

```
private var _tilt:Number=0;
private var _oldMouseX:Number=0;
private var _oldMouseY:Number=0;
private var _canMove:Boolean=false;
private var _gravity:Number;
private var _jumpSpeed:Number=0;
private var _jumpStep:Number;
private var _defaultGrav:Number;
private static const GRAVACCEL:Number=1.2;
private static const MAX_JUMP:Number=100;
private static const FRICTION_FACTOR:Number=0.75;
private static const DEGStoRADs:Number = Math.PI / 180;

public function FPSController(camera:Object3D,stg:Stage,height:Number=20,gravity:Number=5,jumpStep:Number=5)
{
    _camera=camera;
    _stg=stg;
    _controllerHeigh=height;
    _gravity=gravity;
    _defaultGrav=gravity;
    _jumpStep=jumpStep;
    init();
}
private function init():void{
    _camera.y=_controllerHeigh;
    addListeners();
}
private function addListeners():void{
    _stg.addEventListener(MouseEvent.MOUSE_DOWN,
onMouseDown,false,0,true);
    _stg.addEventListener(MouseEvent.MOUSE_UP,
onMouseUp,false,0,true);
    _stg.addEventListener(KeyboardEvent.KEY_DOWN,
onKeyDown,false,0,true);
    _stg.addEventListener(KeyboardEvent.KEY_UP,
onKeyUp,false,0,true);

}
private function onMouseDown(e:MouseEvent):void{
    _oldPan=_pan;
    _oldTilt=_tilt;
    _oldMouseX=_stg.mouseX+400;
    _oldMouseY=_stg.mouseY-300;
```

```
        _canMove=true;
    }
    private function onMouseUp(e:MouseEvent):void{
        _canMove=false;
    }
    private function onKeyDown(e:KeyboardEvent):void{
        switch(e.keyCode)
        {
            case 65:_moveLeft = true;break;
            case 68:_moveRight = true;break;
            case 87:_moveForward = true;break;
            case 83:_moveBack = true;break;
            case Keyboard.SPACE:
                if(_camera.y<MAX_JUMP+_controllerHeigh){
                    _jumpSpeed=_jumpStep;
                }else{
                    _jumpSpeed=0;
                }
                break;
        }
    }
    private function onKeyUp(e:KeyboardEvent):void{
        switch(e.keyCode)
        {
            case 65:_moveLeft = false;break;
            case 68:_moveRight = false;break;
            case 87:_moveForward = false;break;
            case 83:_moveBack = false;break;
            case Keyboard.SPACE:_jumpSpeed=0;break;
        }
    }
    public function walk():void{
        _camSpeed *= FRICTION_FACTOR;
        _camSideSpeed*= FRICTION_FACTOR;
        if(_moveForward){ _camSpeed+=CAM_ACCEL;}
        if(_moveBack) { _camSpeed-=CAM_ACCEL;}
        if(_moveLeft){ _camSideSpeed-=CAM_SIDE_ACCEL;}
        if(_moveRight) { _camSideSpeed+=CAM_SIDE_ACCEL;}
        if (_camSpeed < 2 && _camSpeed > -2){
            _camSpeed=0;
        }
        if (_camSideSpeed < 0.05 && _camSideSpeed > -0.05){
            _camSideSpeed=0;
```

```
        }
        _forwardLook=_camera.transform.deltaTransformVector(new
Vector3D(0,0,1));
        _forwardLook.normalize();
        _camera.x+=_forwardLook.x*_camSpeed;
        _camera.z+=_forwardLook.z*_camSpeed;

        _sideLook=_camera.transform.deltaTransformVector(new
Vector3D(1,0,0));
        _sideLook.normalize();
        _camera.x+=_sideLook.x*_camSideSpeed;
        _camera.z+=_sideLook.z*_camSideSpeed;

        _camera.y+=_jumpSpeed;
        if(_canMove){
            _pan = 0.3*(_stg.mousePosition+400 - _oldMouseX) + _oldPan;
            _tilt = -0.3*(_stg.mousePosition-300 - _oldMouseY) + _oldTilt;
            if (_tilt > 70){
                _tilt = 70;
            }

            if (_tilt < -70){
                _tilt = -70;
            }
        }
        var panRADs:Number=_pan*DEGStoRADs;
        var tiltRADs:Number=_tilt*DEGStoRADs;
        _camTarget.x = 100*Math.sin( panRADs) * Math.cos(tiltRADs) +
_camera.x;
        _camTarget.z = 100*Math.cos( panRADs) * Math.cos(tiltRADs) +
_camera.z;
        _camTarget.y = 100*Math.sin(tiltRADs) +_camera.y;
        if(_camera.y>_controllerHeigh){
            _gravity*=GRAVACCEL;
            _camera.y-=_gravity;
        }
        if(_camera.y<=_controllerHeigh ){
            _camera.y=_controllerHeigh;
            _gravity=_defaultGrav;
        }
        _camera.lookAt(_camTarget);
    }
}
```

Working with Away3D Cameras

Now let's put it to work in the main application:

```
FPSDemo.as
package
{
    public class FPSDemo extends AwayTemplate
    {
        [Embed(source="assets/buildings/CityScape.3ds", mimeType="application/octet-stream")]
        private var City:Class;
        [Embed(source="assets/buildings/CityScape.png")]
        private var CityTexture:Class;
        private var _cityModel:Object3D;
        private var _fpsWalker:FPSController;
        public function FPSDemo()
        {
            super();
        }

        override protected function initGeometry() : void{
            parse3ds();
        }
        private function parse3ds():void{
            var max3ds:Max3DS=new Max3DS();
            _cityModel=max3ds.parseGeometry(City);
            _view.scene.addChild(_cityModel);
            _cityModel.materialLibrary.getMaterial("bakedAll [Plane0").material=new BitmapMaterial(Cast.bitmap(new CityTexture()));
            _cityModel.scale(3);
            _cityModel.x=0;
            _cityModel.y=0;
            _cityModel.z=700;
            _cityModel.rotate(Vector3D.X_AXIS,-90);           _cam.z=-1000;
            _fpsWalker=new FPSController(_cam,stage,_view,20,12,250);
        }
        override protected function onEnterFrame(e:Event) : void{
            super.onEnterFrame(e);
            _fpsWalker.walk();
        }
    }
}
```

How it works...

`FPSController` class looks a tad scary, but that is only at first glance. First we pass the following arguments into the constructor:

1. **camera**: Camera3D reference (here `Camera3D`, by the way, is the most appropriate one for FPS).
2. **stg**: References to flash stage because we are going to assign listeners to it from within the class.
3. **height**: It is the camera distance from the ground. We imply here that the ground is at 0,0,0.
4. **gravity**: Gravity force for jump.
5. **JumpStep**: Jump altitude.

Next we define listeners for mouse UP and DOWN states as well as events for registering input from A,W,D,S keyboard keys to be able to move the `FPSController` in four different directions.

In the `onMouseDown()` event handler, we update the old pan, tilt the previous `mouseX` and `mouseY` values as well as by assigning the current values when the mouse has been pressed to `_oldPan`, `_oldTilt`, `_oldMouseX`, and `_oldMouseY` variables accordingly. That is a widely used technique. We need to do this trick in order to have nice and continuous transformation of the camera each time we start moving the `FPSController`. In the methods `onKeyUp()` and `onKeyDown()`, we switch the flags that indicate to the main movement execution code. This will be seen shortly and we will also see which way the camera should be moved according to the relevant key press. The only part that is different here is the block of code inside the `Keyboard.SPACE` case. This code activates jump behavior when the space key is pressed.

On the `SPACE` bar, the camera `jumpSpeed` (that, by default, is zero) receives the `_jumpStep` incremented value and this, in case the camera has not already reached the maximum altitude of the jump defined by `MAX_JUMP`, is added to the camera ground height.

Now it's the `walk()` function's turn. This method is supposed to be called on each frame in the main class:

```
_camSpeed *= FRICTION_FACTOR;  
_camSideSpeed*= FRICTION_FACTOR;
```

Two preceding lines slow down, or in other words apply friction to the front and side movements. Without applying the friction. It will take a lot of time for the controller to stop completely after each movement as the velocity decrease is very slow due to the easing.

Next we want to accelerate the movements in order to have a more realistic result. Here is acceleration implementation for four possible walk directions:

```
if(_moveForward) { _camSpeed+= CAM_ACCEL; }
if(_moveBack) { _camSpeed-= CAM_ACCEL; }
if(_moveLeft) { _camSideSpeed-= CAM_SIDE_ACCEL; }
if(_moveRight) { _camSideSpeed+= CAM_SIDE_ACCEL; }
```

The problem is that because we slow down the movement by continuously dividing current speed when applying the drag, the speed value actually never becomes zero. Here we define the range of values closest to zero and resetting the side and front speeds to 0 as soon as they enter this range:

```
if (_camSpeed < 2 && _camSpeed > -2) {
    _camSpeed=0;
}

if (_camSideSpeed < 0.05 && _camSideSpeed > -0.05) {
    _camSideSpeed=0;
}
```

Now we need to create an ability to move the camera in the direction it is looking. To achieve this we have to transform the forward vector, which present the forward look of the camera, into the camera space denoted by `_camera` transformation matrix. We use the `deltaTransformVector()` method as we only need the transformation portion of the matrix dropping out the translation part:

```
_forwardLook=_camera.transform.deltaTransformVector(new
Vector3D(0,0,1));
_forwardLook.normalize();
_camera.x+=_forwardLook.x*_camSpeed;
_camera.z+=_forwardLook.z*_camSpeed;
```

Here we make pretty much the same change as the previous one but for the sideways movement transforming the side vector by the camera's matrix:

```
_sideLook=_camera.transform.deltaTransformVector(new Vector3D(1,0,0));
_sideLook.normalize();
_camera.x+=_sideLook.x*_camSideSpeed;
_camera.z+=_sideLook.z*_camSideSpeed;
```

And we also have to acquire base values for rotations from mouse movement. `_pan` is for the horizontal (x-axis) and `_tilt` is for the vertical (y-axis) rotation:

```
if(_canMove) {
    _pan = 0.3*(_stg.mousePosition+400 - _oldMouseX) + _oldPan;
    _tilt = -0.3*(_stg.mousePosition-300 - _oldMouseY) + _oldTilt;
```

```
if (_tilt > 70) {
    _tilt = 70;
}

if (_tilt < -70) {
    _tilt = -70;
}
```

We also limit the y-rotation so that the controller would not rotate too low into the ground and conversely, too high into zenith. Notice that this entire block is wrapped into a `_canMove` Boolean flag that is set to true only when the mouse DOWN event is dispatched. We do it to prevent the rotation when the user doesn't interact with the controller.

Finally we need to incorporate the camera local rotations into the movement process. So that while moving, you will be able to rotate the camera view too:

```
var panRADs:Number=_pan*DEGStoRADs;
var tiltRADs:Number=_tilt*DEGStoRADs;
_camTarget.x = 100*Math.sin( panRADs) * Math.cos(tiltRADs) +
_camera.x;
_camTarget.z = 100*Math.cos( panRADs) * Math.cos(tiltRADs) +
_camera.z;
_camTarget.y = 100*Math.sin(tiltRADs) +_camera.y;
```

And the last thing is applying gravity force each time the controller jumps up:

```
if(_camera.y>_controllerHeigh){
    _gravity*=GRAVACCEL;
    _camera.y-=_gravity;
}
if(_camera.y<=_controllerHeigh ){
    _camera.y=_controllerHeigh;
    _gravity=_defaultGrav;
}
```

Here we first check whether the camera y-position is still bigger than its height, this means that the camera is in the "air" now. If true, we apply gravity acceleration to gravity because, as we know, in real life, the falling body constantly accelerates over time. In the second statement, we check whether the camera has reached its default height. If true, we reset the camera to its default y-position and also reset the gravity property as it has grown significantly from the acceleration addition during the last jump.

To test it in a real application, we should initiate an instance of the `FPSController` class. Here is how it is done in `FPSDemo.as`:

```
_fpsWalker=new FPSController(_cam,stage,20,12,250);
```

We pass to it our scene camera3D instance and the rest of the parameters that were discussed previously.

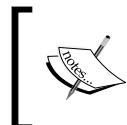
The last thing to do is to set the `walk()` method to be called on each frame:

```
override protected function onEnterFrame(e:Event) : void{  
    super.onEnterFrame(e);  
    _fpsWalker.walk();  
}
```

Now you can start developing the Away3D version of Unreal Tournament!

Creating Camera Depth of Field Effect

The **depth of field (DOF)** in optics is a distance at which a target object is focused. The unfocused objects around become blurry. You have seen this effect probably hundreds of times when filming with your home camera. In 3D, scene DOF effect enables you to define a distance from the camera at which an object is being fully focused, whereas the rest of the objects that are out of range of this distance become gradually blurred. Away3D gives us the ability to simulate this effect when using cameras in conjunction with `DepthOfFieldSprite` billboard. Let's see how to do it.



It is worth noting that depth of field effect is a CPU-intensive process as it makes use of the Flash generic blur filter. It may significantly slow down your application if used extensively.

Getting ready

Create a basic Away3D scene using `AwayTemplate`. Make sure you embed the `dofText.png` image file into the code, as it will serve as graphics fill for `DepthOfFieldSprite` object.

How to do it...

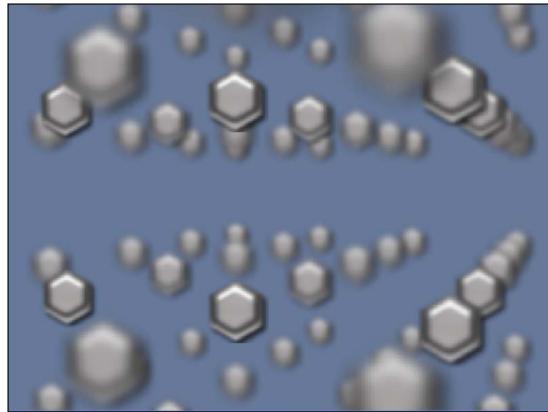
We are going to use here our brand new `FPSController` for camera manipulation. Also, we create a group of `DepthOfFieldSprite` objects and put them in an array of rows and columns that extends into depth of the viewport. That way, we can better see and understand the DOF effect in action:

`DOFDemo.as`

```
package  
{  
    public class DOFDemo extends AwayTemplate
```

```
{  
    [Embed(source="assets/dofText.png")]  
    private var DofTexture:Class;  
    private var _fpsWalker:FPSController;  
    private var dc:DofCache;  
    private var _bitMat:BitmapMaterial;  
    public function DOFDemo()  
    {  
        super();  
    }  
    override protected function initMaterials() : void{  
        _bitMat=new BitmapMaterial(Cast.bitmap(new DofTexture()));  
    }  
    override protected function initGeometry() : void{  
        var xt:Number=0;  
        var yt:Number=0;  
        var zt:Number=0;  
  
        for(var i:int=0;i<7;++i){  
            for(var b:int=0;b<7;++b){  
                for(var c:int=0;c<4;++c){  
                    xt=i*200+200;  
                    yt=b*200+200;  
                    zt=c*200+200;  
                    var dof:DepthOfFieldSprite=new DepthOfFieldSprite(_  
bitMat,10,10);  
                    _view.scene.addSprite(dof);  
                    dof.x=xt;  
                    dof.y=yt-400;  
                    dof.z=zt;  
                }  
            }  
        }  
        DofCache.usedof=true;  
        DofCache.maxblur=10;  
        DofCache.focus=550;  
        DofCache.aperture=45;  
        DofCache.doflevels=10;  
        _fpsWalker=new FPSController(_cam,stage,120,12,250);  
    }  
    override protected function onEnterFrame(e:Event) : void{  
        super.onEnterFrame(e);  
        _fpsWalker.walk();  
    }  
}
```

You should see the following result:



How it works...

First, in the `initGeometry()` method, we run a 3-dimensional loop in order to deploy the sprites in an array so that each row of objects expends into the depth, evenly creating a number of good looking columns of aligned sprites.

Now comes the fun part. We set `static DofCache.usedof=true` in order to enable the DOF effect. `DofCache` is closely related to `DepthOfFieldSprite`. When we create a set of `DepthOfFieldSprite` objects, their bitmaps are automatically cached for future use of `DofCache`. Therefore, tweaking the `DofCache` settings, affects all the sprites.

We set up the following properties of `DofCache`:

- ▶ `DofCache.usedof=true`: This enables the DOF effect.
- ▶ `DofCache.maxblur`: This is a maximum amount of blur to be applied by blur filter.
- ▶ `DofCache.focus`: It is a distance from the camera to objects to be fully focused.
- ▶ `DofCache.aperture`: This one needs some explanation. If you wish to isolate your focused object from the rest, like as if you were focusing your home camera on a flower with the background totally out of focus, you need to set a shallow depth of field. The way to influence DOF is to control camera aperture. The less the value of an aperture property, the larger the DOF. And conversely, to get smaller aperture (deeper DOF), we should increase the aperture property value.
- ▶ `DofCache.doflevels`: It is not easy to understand `doflevels` influence if we do not play around with its values. This property is responsible for the distributing of strength of the blur effect to several levels so that when the focused objects exceeds the defined focus range, its blur doesn't disappear abruptly, but rather diminishes over a number of steps defined by the `levels` property. This creates smoother focus transition over a group of objects.

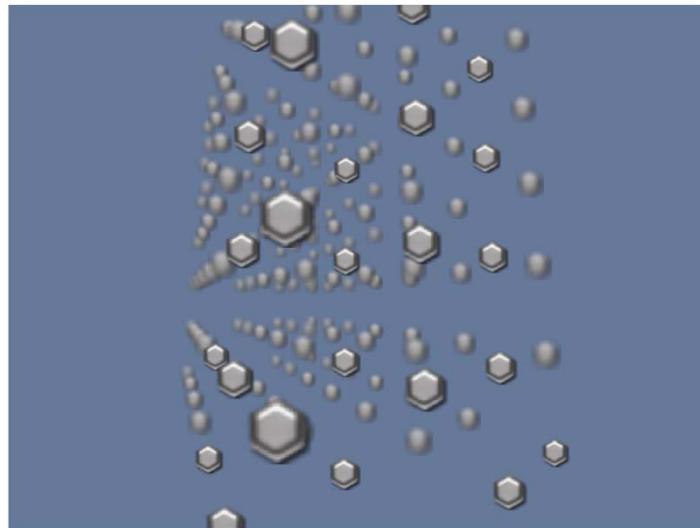
There's more...

There is another way to set DOF. And it is directly through Camera3D properties. Let's try it.

In the previous example, comment out the entire DofCache block and put instead:

```
_cam.enableDof();  
_cam.doflevels=10;  
_cam.aperture=520;  
_cam.maxblur=5;  
_cam.focus=30;
```

Here is our result, an array of sprites with DOF effect applied:



However, there is a problem with using this approach. Even though we basically get the DOF effect, it (the effect) doesn't behave like the effect created with DofCache. The blur transition is not smooth. Also we are forced to use the camera focus property to define the DOF distance. And because this focus influences the real focus of the camera, we get noticeable perspective distortion, as you can see on the preceding image. Nevertheless, in certain scenarios, this approach can be handy.

Creating DOF on Mesh Objects

Now, I will show you how we can get simple DOF effects on a regular geometry. Notice that in this showcase, you will see only a basic implementation of the blur effect based on the distance of the camera to its target, but it gives an idea of how to convert it into a full scale "Mesh DOF Engine".

Working with Away3D Cameras

For this purpose, we can create a basic Away3D scene and add to it our downtown city model. Here is the code:

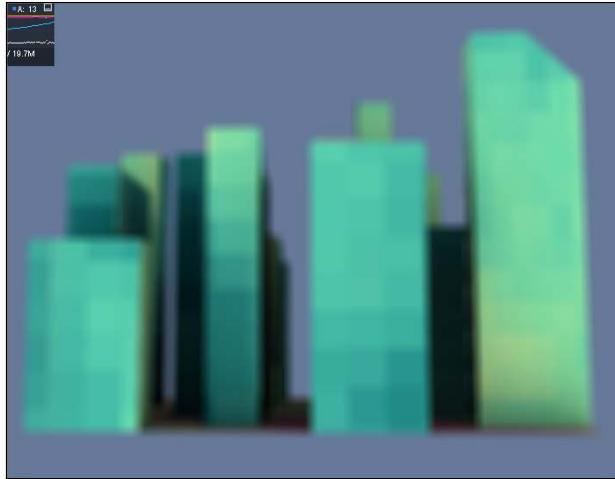
FPSModelBlur.as

```
package
{
    public class FPSModelBlur extends AwayTemplate
    {
        [Embed(source="assets/buildings/CityScape.3ds", mimeType="application/octet-stream")]
        private var City:Class;
        [Embed(source="assets/buildings/CityScape.png")]
        private var CityTexture:Class;
        private var _cityModel:Object3D;
        private var _fpsWalker:FPSCController;
        private var _distToCam:Number;
        private var _blurF:BlurFilter=new BlurFilter();
        private var _blurVal:Number=1;
        private var _oldBlurVal:Number=1;
        private var _oldDistToCam:Number=1;
        private const MINRANGE:Number=500;
        private const MAXRANGE:Number=2500;
        public function FPSModelBlur()
        {
            super();
        }
        override protected function initGeometry() : void{
            parse3ds();
        }

        private function parse3ds():void{
            var max3ds:Max3DS=new Max3DS();
            _cityModel=max3ds.parseGeometry(City);
            _view.scene.addChild(_cityModel);
            _cityModel.materialLibrary.getMaterial("bakedAll [Plane0]").material=new BitmapMaterial(Cast.bitmap(new CityTexture()));
            _cityModel.scale(3);
            _cityModel.x=0;
            _cityModel.y=0;
            _cityModel.z=700;
            _cityModel.rotate(Vector3D.X_AXIS, -90);
            _cam.z=-1000;
            _cityModel.ownCanvas=true;
        }
    }
}
```

```
    _fpsWalker=new FPSController(_cam,stage,20,12,250);
}
override protected function onEnterFrame(e:Event) : void{
    super.onEnterFrame(e);
    if(_cityModel){
        _distToCam=_cityModel.distanceTo(_cam);
        _blurF.blurX=_blurVal/100;
        _blurF.blurY=_blurVal/100;
        _cityModel.filters=[_blurF];
        _blurVal=_distToCam*_oldBlurVal/(_oldDistToCam);
        _oldDistToCam=_distToCam;
        _oldBlurVal=_blurVal;
        if(_distToCam>=MAXRANGE){
            _blurVal=_blurF.blurX*100;
        }
        if(_distToCam<=MINRANGE){
            _blurVal=0;
        }
    }
    _fpsWalker.walk();
}
}
}
```

As seen in the following image, Downtown model mesh is being blurred as we move away from it:



All the "magic" is in the `onEnterFrame()` function. We check each frame's distance from the target object to the camera. Then we set the blur values of the blur filter which is calculated by dividing `_blurVal` by 100 to get a number that fits into valid blur values range of the generic Flash Blur filter. Now we need to put into dependence the blur strength with the current distance of the camera from the target. These lines take care of it:

```
_blurVal=_distToCam*_oldBlurVal/(_oldDistToCam) ;  
_oldDistToCam=_distToCam;  
_oldBlurVal=_blurVal;
```

Finally, we check against predefined `MAXRANGE` and `MINRANGE` values when to stop increasing or decreasing the blur. So, in our case, when the camera distance from the target is greater than 2500, the blur ceases to increase further. And when the camera is in range of 500 pixels from the target, the blur value is zero (actually, no blur).

If you want to achieve similar effect of DOF as in the previous example, you need to create a system that will manage a DOF of each object separately. The best way here is to take the code from the `onEnterFrame()` function and to put it in a separate class (that is, to be a DOF manager) which would receive a reference to an object and to the scene camera. This way each primitive will receive its own blur depending on the distance from the camera. Also before attempting this, you should know that applying filters on geometry is quite a memory-intensive task, as you probably might have noticed in this example. Therefore, applying such effects to many meshes can completely render your application useless.

Detecting whether an object is in front of or behind the camera

Now let's see how we can check whether a certain object is located in front of or behind the camera. Such a check may be useful if you develop an FPS game where you need an enemy agent or any other **non player character (NPC)** to be able to detect whether other agents are behind or in front of it. Fortunately, this task is easy to accomplish. Linear math has a calculation called **dot** or **inner** product. The Dot product is a sum of products between corresponding components of vectors that should lie in the same plane. Also if one of the vectors is zero, the Dot product would be zero as well. In Away3D, you don't need to deal with that math directly. `Vector3D` class has got already built-in a `dot()` method that calculates the Dot product of two vectors. Let's see how it works.

Getting ready

Set up the basic Away3D scene using `AwayTemplate`.

How to do it...

We position our camera with default direction facing the positive z-axis. Then from the distance of 3000 pixels in front of the camera, we start moving spheres towards it until they get behind the camera. For the sake of demonstration and for visual confirmation, we imply that we can't really see the translated sphere's actual place because for the sake of demo our camera is an NPC agent. Then, by getting the Dot product value, we would know whether the approaching object is in front of or behind the camera:

```
FrontBackDetect.as
package
{
    public class FrontBackDetect extends AwayTemplate
    {
        private var _sp:Sphere;
        private var _distVect:Vector3D;
        private var _dot:Number;
        private var r:BasicRenderer;
        public function FrontBackDetect()
        {
            super();
            _cam.rotationY=0;
        }
        override protected function initListeners() : void{
            super.initListeners();
        }
        override protected function initGeometry() : void{
            startTweens();
        }
        private function startTweens():void{
            _sp=new Sphere({radius:30,material:new
ColorMaterial(0x394949)});
            _sp.z=3000;
            TweenMax.fromTo(_sp,7,{z:3000,x:0},{z:-500,x:Math.random()*Math.
random()*800-400,onComplete:onTweenComplete});
            _view.scene.addChild(_sp);
        }
        private function onTweenComplete():void{
            _view.scene.removeChild(_sp);
            _sp=null;
            startTweens();
        }
    }
}
```

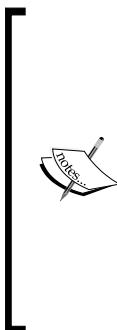
Working with Away3D Cameras

```
override protected function onEnterFrame(e:Event) : void{
    super.onEnterFrame(e);
    if(_sp){
        _distVect=Matrix3DUtils.getForward(_cam.transform);
        var objPos:Vector3D=_sp.position;
        objPos.subtract(_cam.position);
        _distVect.normalize();
        objPos.normalize();
        _dot=_distVect.dotProduct(objPos);
        trace(dot);
    }
}
```

The Dot product returns a single number. If the Dot product number is greater than zero—the tested object is in front of the camera. If it is zero—it is perpendicular to the camera direction vector (90 degrees to it). And if the product is less than zero—you know that the object is behind the camera.

Additional power of the Dot product is an ability to extract the angle between the two vectors. Therefore, besides knowing whether the object is behind or not, you can also get the angle between the camera facing direction vector and the tested object.

Before doing this calculation, you need to just normalize the vectors in order to cancel out additional calculations.



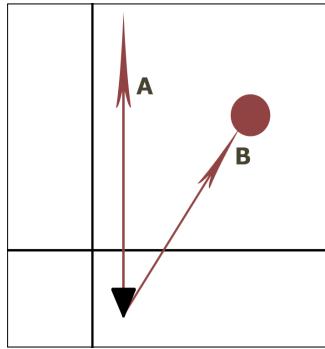
Normalizing vector operations turns vector magnitude to one. Normalized vectors are also called unit vectors. In the cases where the length of a vector is not important but only their direction, it is often recommended to normalize the vector for further calculations because the unit magnitude cancels out the need in additional mathematical operations. The mathematical expression of Dot product is $u \cdot v = |u| |v| \cos \theta$: where $|u|$ and $|v|$ are the corresponding vectors lengths. By normalizing these vectors, using the `Vector3D.normalize()` method, we can drop their length as they are equal to one and we are left only with this formula to calculate a Dot product: $u \cdot v = \cos \theta$ where θ is the angle between two vectors.

As you can see, we have already normalized the vectors. So to extract the angle between two vectors, we use `acos` of their Dot product. Add the following line after the `_dot` calculation:

```
var angl:Number=Math.acos(dot)*180/Math.PI
```

How it works...

We need to define two vectors in order to extract their Dot product:



As depicted in the preceding image, vector A is a vector of the face direction of the camera.
We get this vector by the following code:

```
_distVect=Matrix3DUtils.getForward(_cam.transform);
```

Vector B is calculated by finding the vector between camera and object position.

The easiest way is to take object position and subtract the camera position from it:

```
var objPos:Vector3D=_sp.position;  
objPos.subtract(_cam.position);
```

Now we normalize both vectors as we will use them for angle calculation:

```
_distVect.normalize();  
objPos.normalize();
```

Now we use Away3D's built-in dot () method to get the Dot product between the camera and the given object:

```
dot=_distVect.dotProduct(objPos);
```

The last operation is to find the angle between them:

```
var angl:Number=Math.acos(dot)*180/Math.PI;
```

The preceding equation is the programming representation of this formula: $\theta = \text{acos}(u \cdot v)$.

Don't forget to convert the angle to degrees.

There's more...

If you noticed our calculated Dot product is "camera-based", that is, we checked the dot by relation of the object to the camera from the camera's point of view. We can also do it from the side of object as well. So in this case, our tweened spheres are the objects that calculate direction vector to the camera and then use it to find the Dot product.

This approach is almost identical to the first one:

```
var dot:Number=objPos.dotProduct(_distVect);
```

Here we calculate the vector with the direction from the sphere to the camera and then extract the Dot product with the camera direction vector as in the first example.

Changing lenses

In 3D graphics, geometry is projected onto a two dimensional plane before being drawn on the screen. This operation is essential because the computer screen by nature has only two dimensions and projection just solves that problem by converting 3D coordinates of an object in a 3D scene into 2D by projecting them. There are different types of projection methods. The most known are Perspective and Orthographic projections. There is a clear technical and visual difference between the two:

- ▶ **Perspective projections** tend to imitate the view as is perceived by human eyes. That is, for instance, if you stand on a straight railroad, you can see that the rails eventually intersect somewhere on the line of the horizon (also called vanishing point in computer graphics). That is the basic example for visual representation of Perspective projection.
- ▶ The **Orthographic** (also called Orthogonal) projection is impossible to give an example from real life because of the way our eyes work. But this type of projection is widespread in architectural visualization, mapping, and old top view arcade games. If we take the example of the rails and try to apply it to Orthographic projection in theory, the rails would never intersect because the coordinates are projected in parallel onto projection plane and therefore a rendered object projection is not subject to skew operation based on its size and distance from the camera.

Away3D gives us the ability to choose the kind of projection we wish to apply by applying the right lens type. Away3D has different types of lens found in the `cameras.lens` package. In this specific example, we are concerned only with Perspective and Orthogonal lenses, as the chances are really high that you would be tempted to use them.

Getting ready

First set up a basic Away3D scene using `AwayTemplate`. We put some geometry in order to test our lenses on. In this example, we use a City downtown model, created for these tests which is found in this chapter's assets folder.

Then create a `HoverCamera3D`, as it is most suitable for interactively moving around the object.

How to do it...

We first instantiate the `PerspectiveLens` and `OrthogonalLens` classes found in the `cameras.lens` package. Then we change the lenses by applying one of their instances to the camera and you will instantly see the difference between Perspective and Orthographic projection:

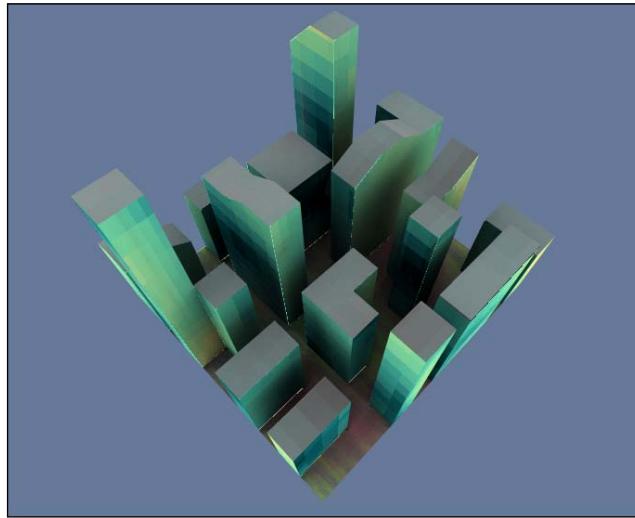
```
package
{
    public class ChangeLens extends AwayTemplate
    {
        [Embed(source="assets/buildings/CityScape.3ds", mimeType="application/octet-stream")]
        private var City:Class;
        [Embed(source="assets/buildings/CityScape.png")]
        private var CityTexture:Class;
        private var _cityModel:Object3D;
        private var _hoverCam:HoverCamera3D;
        private var _oldMouseX:Number=0;
        private var _oldMouseY:Number=0;
        private static const EASE_FACTOR:Number=0.5;
        private var _step:Number=10;
        private var _canMove:Boolean=false;
        private var _perspLens:PerspectiveLens=new PerspectiveLens();
        private var _orthoLens:OrthogonalLens=new OrthogonalLens();
        public function ChangeLens()
        {
            super();
        }
        override protected function initListeners() : void{
            super.initListeners();
            stage.addEventListener(MouseEvent.MOUSE_DOWN,
onMouseDown,false,0,true);
            stage.addEventListener(MouseEvent.MOUSE_UP,
onMouseUp,false,0,true);
        }
    }
}
```

Working with Away3D Cameras

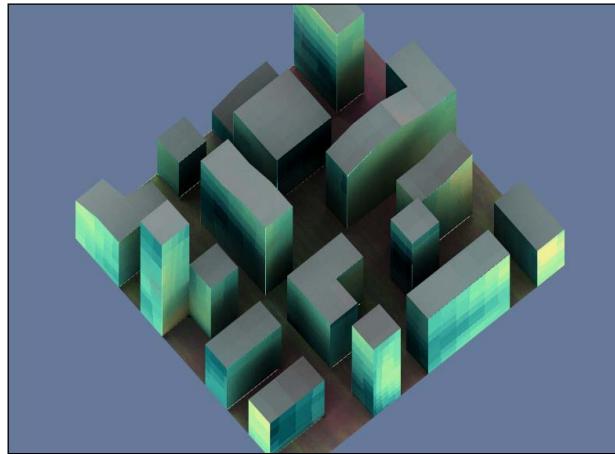
```
        }
    override protected function initGeometry() : void{
        parse3ds();
        setHowerCamera();
        var timer:Timer=new Timer(3000,0);
        timer.addEventListener(TimerEvent.TIMER,swapLens,false,0,true) ;
        timer.start();
    }
    private function onMouseDown(e:MouseEvent):void{
        _oldMouseX=stage.mouseX;
        _oldMouseY=stage.mouseY
        _canMove=true;
    }
    private function onMouseUp(e:MouseEvent):void{
        _canMove=false;
    }
    private function parse3ds():void{
        var max3ds:Max3DS=new Max3DS();
        _cityModel=max3ds.parseGeometry(City);
        _view.scene.addChild(_cityModel);
        _cityModel.materialLibrary.getMaterial("bakedAll [Plane0]").material=new BitmapMaterial(Cast.bitmap(new CityTexture()));
        _cityModel.scale(3);
        _cityModel.z=700;
        _cityModel.rotate(Vector3D.X_AXIS,-90);
    }
    private function setHowerCamera():void{
        _hoverCam=new HoverCamera3D();
        _view.camera=_hoverCam;
        _hoverCam.target=_cityModel;
        _hoverCam.distance = 1200;
        _hoverCam.maxTiltAngle = 80;
        _hoverCam.minTiltAngle = 0;
        _hoverCam.steps=8;
        _hoverCam.yfactor=1;
    }
    private function swapLens(e:TimerEvent):void{
        if(_hoverCam.lens==_perspLens){
            setOrthogonalLens();
        }else{
            setPerspectiveLens();
        }
    }
}
```

```
private function setPerspectiveLens():void{
    _hoverCam.lens=_perspLens;
    _hoverCam.zoom=12;
}
private function setOrthogonalLens():void{
    _hoverCam.lens=_orthoLens;
    _hoverCam.zoom=60;
}
override protected function onEnterFrame(e:Event) : void{
    super.onEnterFrame(e);
    if(_hoverCam){
        if(_canMove){
            _hoverCam.panAngle = (stage.mouseX - _oldMouseX)*EASE_FACTOR
;
            _hoverCam.tiltAngle = (stage.mouseY - _oldMouseY)*EASE_
FACTOR ;
        }
        _hoverCam.hover();
    }
}
```

The PerspectiveLens projection looks like this:



The Orthogonal lens projection looks like this:



As you can see from the images, the Perspective projection gives the model a natural world perspective look, whereas the Orthographic projection looks more like an architectural sketch with the parts of the model retaining the same scale—both closer and further from the camera.

How it works...

We define two types of lenses—Perspective and Orthogonal within two functions:

```
private function setPerspectiveLens():void{
    _hoverCam.lens=_perspLens;
    _hoverCam.zoom=12;
}
private function setOrthogonalLens():void{
    _hoverCam.lens=_orthoLens;
    _hoverCam.zoom=60;
}
```

Inside the `initGeometry()` method after parsing scene geometry and setting `HoverCamera3D`, we define a timer which is going to swap between the lenses every three seconds (3,000 milliseconds):

```
override protected function initGeometry() : void{
    parse3ds();
    setHowerCamera();
    var timer:Timer=new Timer(3000,0);
```

```
timer.addEventListener(TimerEvent.TIMER, swapLens, false, 0, true) ;  
    timer.start() ;  
  
}
```

Every three seconds, the following function is triggered by timer's TimerEvent.TIMER event:

```
private function swapLens(e:TimerEvent):void{  
    if(_hoverCam.lens==_perspLens){  
        setOrthogonalLens();  
    }else{  
        setPerspectiveLens();  
    }  
}
```

In the `swapLens()` method, we check what type of lens the camera is assigned. Then assign the different lens to it.

There's more...

The Away3D team is made up of some very serious lads, therefore they could stand the temptation to write additional lenses for us. That is why the current version of Away3D has two additional lenses which are `SphericalLens` and `ZoomFocusLens`.

`SphericalLens` gives us a slight perspective distortion of the rendered object that looks bulgy especially from closer view. It is also called the **fish-eye** effect.

Let's add to the global variables list in the following instance:

```
private var _sphericalLens:SphericalLens=new SphericalLens();
```

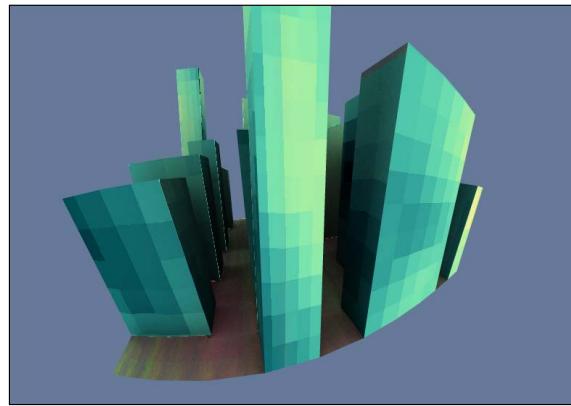
Add the following method that we will use to set the `SphericalLens`:

```
private function setSphericalLens():void{  
    _hoverCam.lens=_sphericalLens;  
    _hoverCam.fov=56;  
}
```

Write a call to this method in the `initGeometry()` function. Before we run the application, let's offset a pivot point a little bit. This way, the camera zooms in and out as it orbits around the model, and you can clearly see the projection effect. So offset the pivot by writing this line at the bottom of the `parse3ds()` method:

```
_cityModel.movePivot(50,300,0);
```

Now you can run the application and see a cool bulgy distortion, as seen in the following image:

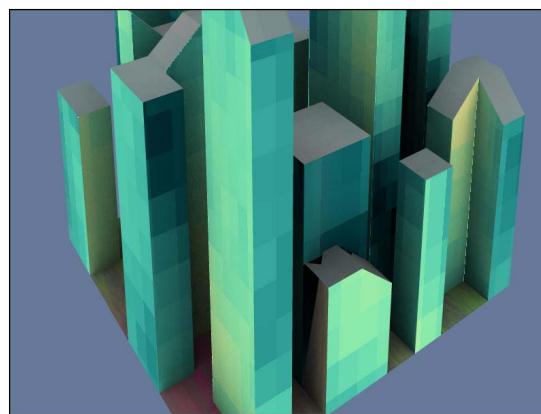


Want to get a cooler result? No problem, let's hack the `SphericalLens` class.

Go to the `SphericalLens` class and find its `project()` method. This method projects the geometry on screen using view matrix, geometry vertices, and screen position of the last. We don't mess with the Matrix and Vertex stuff; instead we would like to distort some more the screen coordinates of the vertices, as those are responsible for the final rendered screen view. We can, for example, control the amount of vertical or horizontal distortion by adding a multiplication factor to the initial `screenVertex` calculation:

```
screenVerts[index2] = _wx * _persp*5; //multiplied by 5  
screenVerts[uint(index2+1)] = _wy * _persp*8; //multiplied by 8  
uvts[uint(index1+2)] = 1/_sz;
```

And Voila!



And if we want to go totally mad, we can do this:

```
screenVerts[index2] = _wx * _persp*Math.random()*5;
screenVerts[uint(index2+1)] = _wy * _persp*Math.
random()*5;
uvts[uint(index1+2)] = 1/_sz;
```

And here is the result of this experiment:



Following a third-person view with a spring camera

The Away3D camera arsenal has got an additional camera class called `SpringCam`. `SpringCam` is actually a `Camera3D` extended into a camera with additional physical behavior which is as the camera's name implies—springing. `SpringCam` is well suited to create 3D person camera control systems. If you plan to develop a racing game or any other 3D person view application, `SpringCam` is just for that.

`SpringCam` has got three important physical properties to set. No need to take a classic mechanics crash course to set them, but without understanding their meaning, it can take you hours to set up the desired behavior for the camera.

Stiffness—controls the stretching factor of the spring. That is how far the camera can stretch during the spring movement. The bigger value means less expansion and more fixed spring behavior. You should be careful though if the damping and mass are low. When increasing the stiffness, you can encounter some crazy spring bounces. It is recommended to keep in the range of 1 and 20.

Damping—is a friction force of the spring. Its purpose is to control the strength of the spring bounce. The higher the value, the weaker the springs force.

Mass—is the camera mass. It controls the weight of the camera. Setting it higher will slow down `SpringCam` movement after its target. It is best to keep the value below 120.



It is important to understand that these three parameters simulate real world physics forces. As such, they are interdependent. When tweaking each of them, you should take into account the values of the other two, as their magnitudes dictate to a great extent how high or low should the value be of the third. Eventually the best approach is to experiment with different ratios till you find the best match.



Getting ready

Set up a basic Away3D scene using AwayTemplate.

In this demo we also use a slightly modified version of FPSController called SimpleWalker. So that, instead of a camera, it will wrap a geometry model. It is found in this chapter's source code folder.

How to do it...

In this program we set up environment with some geometry dispersed around so that we can have a better view of the spring camera behavior. We also create a cube primitive which imitates the target object of the SpringCam. In a real life scenario it could be a human character or a car:

SpringCamFly.as

```
package
{
    public class SpringCamFly extends AwayTemplate
    {
        private var _springCam:SpringCam;
        private var _camTarget:Cube;
        private var _colMat:ColorMaterial;
        private var _fpsWalker:SimpleWalker;
        private var _angle:Number=0;
        private var _numBuilds:int=5;
        private var _radius:int=70;
        private var _buildIter:int=1;
        private var _playerMat:BitmapMaterial;
        public function SpringCamFly()
        {
            super();
            initCamera();
        }
    }
}
```

```

private function initCamera():void{
    _springCam=new SpringCam();
    _springCam.stiffness=2.05;
    _springCam.damping=5;
    _springCam.positionOffset=new Vector3D(0,5,0);
    _springCam.zoom=5;
    _springCam.mass=45;
    _view.camera=_springCam;
    _springCam.target=_camTarget;
}
override protected function initMaterials() : void{
    _colMat=new ColorMaterial(Math.floor(Math.random()*0xffffffff));
    var bttmp:BitmapData=new BitmapData(32,32);
    bttmp.perlinNoise(5,5,12,12345,true,true);
    _playerMat=new BitmapMaterial(bttmp);
}
override protected function initListeners() : void{
    super.initListeners();
}
override protected function initGeometry() : void{
    _camTarget=new Cube({material:_playerMat});
    _camTarget.width=20;
    _camTarget.height=60;
    _camTarget.depth=20;
    _view.scene.addChild(_camTarget);
    _fpsWalker=new SimpleWalker(_camTarget,stage,_view,20,5,5);
    seedBuildings();
}
private function seedBuildings():void{
    for(var b:int=0;b<_numBuilds;++b){
        var h:Number=Math.floor(Math.random()*400+80);
        _angle=Math.PI*2/_numBuilds*b;
        _colMat=new ColorMaterial(Math.round(Math.random()*0x565656));
        var build:Cube=new Cube({width:20,height:h,depth:20});
        build.cubeMaterials.back=_colMat;
        build.cubeMaterials.bottom=_colMat;
        build.cubeMaterials.front=_colMat;
        build.cubeMaterials.left=_colMat;
        build.cubeMaterials.right=_colMat;
        build.cubeMaterials.top=_colMat;
        build.movePivot(0,-build.height/2,0);
        build.x=Math.cos(_angle)*_radius*_buildIter*2;
        build.z=Math.sin(_angle)*_radius*_buildIter*2;
        build.y=0;
    }
}

```

```
        _view.scene.addChild(build);
    }
    _buildIter++;
    if(_buildIter<7){
        _numBuilds*=_buildIter/2;
        seedBuildings();
    }
}
override protected function onEnterFrame(e:Event) : void{
    super.onEnterFrame(e);
    if(_springCam&&_fpsWalker){
        _springCam.view;
        _fpsWalker.walk();
    }
}
}
```

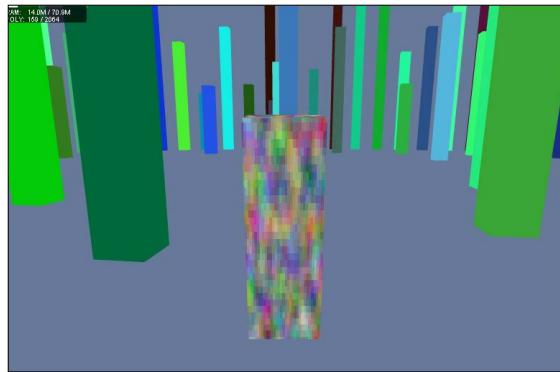
How it works...

We set our `SpringCam` in the `initCamera()` function. Then in the `initGeometry()` function, we first created the camera target called `_camTarget` which was passed into the `SimpleWalker` class, which was the modified version of `FPSController`. The next method, named `seedBuilding()`, creates for us cubes of different sizes and spreads them in a circular array around the scene. In the `onEnterFrame()` function, we have to call the `_springCam.view` property which activates `SpringCam` spring behavior and also the `walk()` method of the `SimpleWalker` class that executes target movement control in real time as it does in `FPSController`.

The most important task is to find and assign optimal values for desired camera behavior to damping, stiffness, and mass properties. Also, it is important not to forget to call `_springCam.view` in the `onEnterFrame()` method, as not doing it will leave the camera motionless.

As the target moves, the camera follows it adjusting its rotation to the target's direction. The camera spring behavior is expressed when its distance to the target stretches when the last accelerates and, conversely, when its speed starts to die, the camera accelerates towards its default position relative to the target.

In the following image, we have a perlin noise textured cube that stands for the 3rd person model. As you move around the scene, the camera smoothly follows the character:



There's more...

The SpringCam can also serve as a first person camera. We can achieve this by just two lines of code. This is when `lookOffset` and `positionOffset` properties come into play.

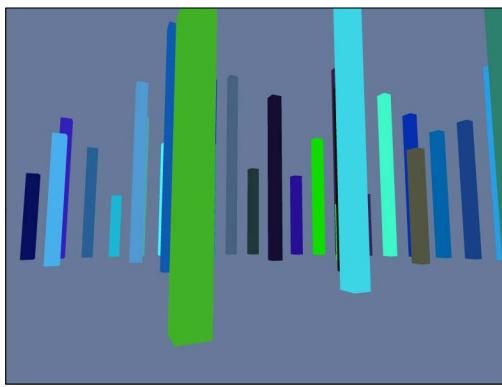
First let's move the z-position of the camera forward so that, by default, it is located in front of the target. In `initCameraFunction()`, change the `positionOffset` to this:

```
_springCam.positionOffset=new Vector3D(0,15,200);
```

After that, we need to offset z-value for `lookOffset`, because currently the camera looks at the target position. Do it by writing this line:

```
_springCam.lookOffset=new Vector3D(0,0,400);
```

Here is a result, by applying the `lookOffset`, we get a first person view:



And we are done! Nice FPS with just two lines of code!

Tracking screen coordinates of 3D objects

Now, let's say you create a flash version of a flight simulation game where your plane should lock on a target with a cool marker in order to launch an air-to-air missile. In such a scenario, a jet-plane is a 3D model that is located in 3D space, whereas the target marker we can make from a regular 2D shape using Flash Sprite. The only solution we need to find is how to translate the 3D coordinates of the Jet Plane into a flash stage 2D coordinates system. There are several ways to do this, but the easiest one is by calculating the screen position of the vertices.

Getting ready

We set up a basic Away3D scene using `AwayTemplate`.

For now, instead of an F-22 jet-fighter, we will be using a Sphere primitive. Copy a `swc` file called `graphicsLib.swc` from this chapter's assets folder to your project and link it as you do with any `swc` library. It contains graphics for the 2D marker that we will use to track a 3D object screen position.

How to do it...

`ScreenVertexTrace.as`

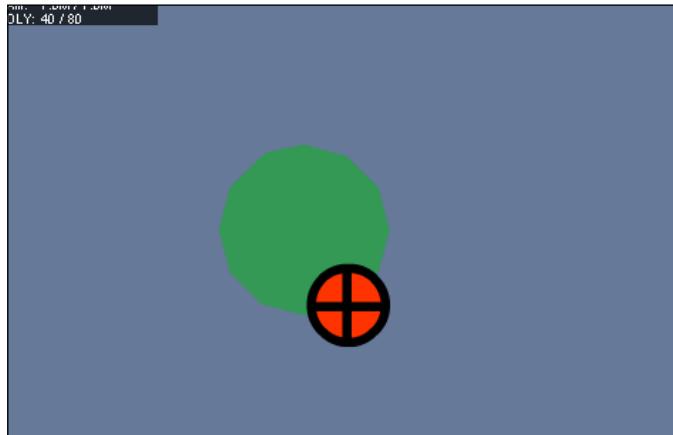
```
package
{
    public class ScreenVertexTrace extends AwayTemplate
    {
        private var _tracker:TargetMarker;
        private var _sp:Sphere;
        private var _spriteX:Number=0;
        private var _spriteY:Number=0;
        private var _screenVertex:Vector3D;
        public function ScreenVertexTrace()
        {
            super();
            createShape();
            beginTween();
        }
        override protected function initGeometry():void{
            _sp=new Sphere({radius:30,material:new
ColorMaterial(0x339955)});
```

```
_view.scene.addChild(_sp);
_sp.z=500;
_sp.y=0;
_sp.x=0;
}
private function createShape():void{
    _tracker= new TargetMarker();
    _tracker.graphics.beginFill(0x121212);
    _tracker.graphics.drawCircle(0,0,15);
    _tracker.graphics.endFill();
    stage.addChild(_tracker);
}
private function beginTween():void{
    TweenMax.to(_sp, 8, {bezierThrough:pathArray(), repeat:-1});
}
private function pathArray():Array{
    var tempArr:Array=
        [{x:-132, y:109}, {x:100, y:189}, {x:156, y:120}, {x:178,
y:0}, {x:178, y:-170}, {x:41, y:-190}, {x:-141, y:-120}];

    return tempArr;
}
override protected function onEnterFrame(e:Event) : void{
    super.onEnterFrame(e);
    if(_sp&&_tracker){
        var randomVertexNum:int=Math.floor(Math.random() *_sp.vertices.
length);
        _screenVertex=_cam.screen(_sp,_sp.vertices[
randomVertexNum]);//
        _spriteX=_screenVertex.x+_view.width*.5;
        _spriteY=_screenVertex.y + _view.height*.5;
        _tracker.x=_spriteX;
        _tracker.y=_spriteY;

    }
}
```

On the following image, we have a target shape which is a MovieClip added to the flash stage. It is moving around using transformed x and y 3D vertex coordinates that we pick randomly from the green sphere mesh:



How it works...

In this program, we created an instance of a sphere that moves in the scene on a bezier path. We also create a regular sprite object called `_tracker` using the `createShape()` method. Now, all the important things happen inside the `onEnterFrame()` function. We extract the screen vertex position using the `_cam.screen()` method which by default requires one argument—`Object3D` reference whose screen vertices we need. Note that if we don't assign a particular vertex for the second argument, then the `screen()` function returns Screen Vertex at the center of the `Object3D`. Now, because we need to transform the Screen Vertex coordinates from the coordinates system, where the x- and y-axis begin in the middle of the screen to the regular flash system with x and y being in the top-left corner, we write this code that resolves the issue by offsetting the screen vertex coordinates according to the view width and height:

```
_spriteX=_screenVertex.x+_view.width*.5;  
_spriteY=_screenVertex.y + _view.height*.5;
```

Now the x and y values can be assigned to the `_tracker` object as they match the flash stage coordinates system:

```
_tracker.x=_spriteX;  
_tracker.y=_spriteY;
```

There's more...

You can, if you wish, track each individual vertex of the given object to produce some fancy effects. Just change the code block in the `onEnterFrame()` function with the following:

```
var randomVertexNum:int=Math.floor(Math.random()*_sp.vertices.length);
    _screenVertex=_cam.screen(_sp,_sp.vertices[
randomVertexNum]);//
    _spriteX=_screenVertex.x+_view.width*.5;
    _spriteY=_screenVertex.y + _view.height*.5;
    _tracker.x=_spriteX;
    _tracker.y=_spriteY;
```

Here, on each frame, we pick, in random order, a different vertex and assign its screen coordinates to the marker. Run the application to see the result.

See also

The Transforming objects in 3D space relative to camera position recipe.

Transforming objects in 3D space relative to the camera position

Let's say you have an idea to create a weapon marker for your FPS Camera in 3D space and not by just using a sprite object positioned in the center of the view. Or maybe you wish to create a full scale 3D inventory or navigation menu that is always positioned relative to the camera transformation. Well, with the help of a basic vector math, you can do it in no time at all.

Getting ready

Create, as always, a basic Away3D scene.

We are going to also use the `FPSController` class. Create some random geometry to have a visual reference to our 3D world.

How to do it...

Here we create an Away3D Sphere primitive, which is going to stay always in the center of the camera view with an arbitrary z-axis offset. In real life, you can put a weapon marker bitmap inside Sprite2D and use it instead:

```
package
{
    public class ObjectToCamTransform extends AwayTemplate
    {
        private var _fpsWalker:FPSController;
        private var _colMat:ColorMaterial;
        private var _angle:Number=0;
        private var _numBuilds:int=5;
        private var _radius:int=70;
        private var _buildIter:int=1;
        private var _marker:Sphere;
        public function ObjectToCamTransform()
        {
            super();
        }
        private function seedBuildings():void{
            for(var b:int=0;b<_numBuilds;++b){
                var h:Number=Math.floor(Math.random()*400+80);
                _angle=Math.PI*2/_numBuilds*b;
                _colMat=new ColorMaterial(Math.round(Math.random()*0x565656));
                var build:Cube=new Cube({width:20,height:h,depth:20});
                build.cubeMaterials.back=_colMat;
                build.cubeMaterials.bottom=_colMat;
                build.cubeMaterials.front=_colMat;
                build.cubeMaterials.left=_colMat;
                build.cubeMaterials.right=_colMat;
                build.cubeMaterials.top=_colMat;
                build.movePivot(0,-build.height/2,0);
                build.x=Math.cos(_angle)*_radius*_buildIter*2;
                build.z=Math.sin(_angle)*_radius*_buildIter*2;
                build.y=0;
                _view.scene.addChild(build);
            }
            _buildIter++;
            if(_buildIter<7){
                _numBuilds*=_buildIter/2;
            }
        }
    }
}
```

```
    seedBuildings();
}
}
override protected function initGeometry() : void{
    seedBuildings();
    _marker=new Sphere({radius:10,material:_colMat});
    _view.scene.addChild(_marker);
    _fpsWalker=new FPSController(_cam,stage,20,5,40);
}
private function transformMarker():void{
    var matrix:Matrix3D=new Matrix3D();
    matrix=_cam.sceneTransform.clone();
    var pos:Vector3D;
    pos=matrix.transformVector(new Vector3D(0,0,300));
    _marker.x=pos.x;
    _marker.y=pos.y;
    _marker.z=pos.z;
}
override protected function onEnterFrame(e:Event) : void{
    super.onEnterFrame(e);
    _fpsWalker.walk();
    if(_marker){
        transformMarker();
    }
}
```

How it works...

All the dark magic happens in the `transformMarker()` function that runs on each frame. The first thing we need to do is to set the marker object transformation to be the same as the camera. We do it because basically when the camera moves or rotates, you want the marker to move accordingly. To achieve this, we need to multiply camera and marker transformation matrices or in other words, transform the marker into camera space by applying to it the camera's rotation matrix with distance offset. If we only clone the matrix of the camera and then apply it to the marker, you will still see that we have done nothing special. If you run the project now, you will see that the marker moves together with the camera and actually has the position of the camera. We need an ability to offset the position of the marker relative to camera orientation. This is done by extracting a new position vector from multiplication with the matrix:

```
pos=matrix.transformVector(new Vector3D(0,0,300));
```

As you can see, we pass into the `transformVector()` method a vector with an offset of 300 pixels in the z-direction. This should give us a result of the marker showing always 300 pixels in front of the camera.

Having said this, now you can also offset the x and y position of the marker if you need it to be in a different position than the camera view center.

That vector math is not that scary as you could see in this really primitive example, but it can get much more complicated in certain scenarios as you will learn further.

Using Quaternion camera transformations for advanced image gallery viewing

Quaternion, in 3D graphics, is an alternative approach to transformations. Although they are harder to grasp than Axis or Euler angles and Matrix transformation, there are obvious advantages in using them in certain scenarios. We will not dive into any mathematical explanation of how Quaternion works as it is quite a complicated topic. Simply put, Quaternion allows us to interpolate from one transformation state into another in a smoother and shorter way using the **spherical linear interpolation method (SLERP)**. Another advantage of Quaternions is low memory consumption. Matrix uses nine numbers to represent orientation, whereas the Quaternion needs only four.

Although the Away3D transformation system is Matrix-based, the library has a Quaternion class that resides in the `core.math` package and can be used as an alternative approach for 3D object transform operations.

In this recipe, you will learn how to use Quaternion methods in Away3D to produce cool camera transformation on images of 3D gallery. From this example, you will see how Quaternion smoothes different axis orientation transforms during the interpolation process.

Getting ready

Set up a basic Away3D scene using `AwayTemplate`.

We prepare a set of several images to use for a gallery.

We also need to extend the `Camera3D` so that it has a public property called `slerp`. We need it in order to track and increment the Camera's Quaternion SLERP.

Extend the `Camera3D` as follows:

```
package utils
{
    public class SlerpCam extends Camera3D
```

```
{  
    public var slerp:Number=0;//new property  
    public function SlerpCam(init:Object=null)  
    {  
        super(init);  
    }  
}
```

Also, we have to modify the Away3D Quaternion class. For some unknown reason, at the time of this writing it lacks three important methods—`createFromMatrix()`, `quaternion2Matrix()`, and `slerp()`. The first converts the matrix transformation into Quaternion, the second does the opposite, while the `slerp()` processes SLERP. In order to shorten the amount of code, I put the modified Quaternion class in Chapter 2's source code folder. Take it from there and put it into your project. Let's get to work!

How to do it...

Here is the final program. The step-by-step explanation is in the next section:

```
package  
{  
    public class QuaternionCamera extends AwayTemplate  
    {  
        [Embed(source="assets/images/img1.png")]private var Img1:Class;  
        [Embed(source="assets/images/img2.png")]private var Img2:Class;  
        [Embed(source="assets/images/img3.png")]private var Img3:Class;  
        [Embed(source="assets/images/img4.png")]private var Img4:Class;  
        [Embed(source="assets/images/img5.png")]private var Img5:Class;  
        [Embed(source="assets/images/img6.png")]private var Img6:Class;  
        private static const NUM_OF_IMAGES:int=60;  
        private var _images:Array=[Img1,Img2,Img3,Img4,Img5,Img6];  
        private var _materialsArr:Array=[];  
        private var _slerpCam:SlerpCam;  
        private var _currentPlane:Plane;  
        private var _cameraDummyTarget:Object3D;  
        private var _camQuaternion:Quaternion;  
        private var _targetQuaternion:Quaternion;  
        private var _slerpQuaternion:Quaternion;  
        private var _cameraDefaultDummy:Object3D;  
        private var _zoomedIn:Boolean=false;  
        public function QuaternionCamera()  
        {
```

Working with Away3D Cameras

```
super();
init();
}

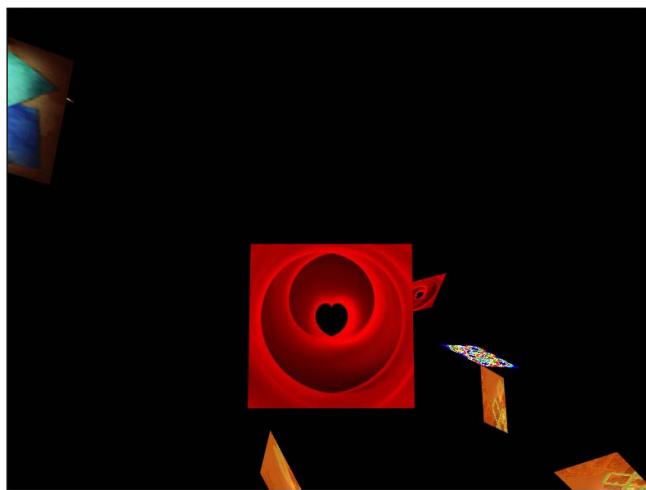
private function init():void{
    _slerpCam=new SlerpCam();
    _cameraDummyTarget=new Object3D();
    _cameraDefaultDummy=new Object3D();
    _cameraDefaultDummy.lookAt(new Vector3D(0,0,500),Vector3D.Y_AXIS);
    _cameraDefaultDummy.x=0;
    _cameraDefaultDummy.y=0;
    _cameraDefaultDummy.z=-100;
    _view.scene.addChild(_cameraDefaultDummy);
    _view.camera=_slerpCam;
}
override protected function initMaterials():void{
    for(var i:int=0;i<6;++i){
        var bitmapMat:BitmapMaterial=new BitmapMaterial(Cast.bitmap(_images[i]));
        bitmapMat.smooth=true;

        _materialsArr.push(bitmapMat);
    }
}
override protected function initGeometry():void{
    for(var i:int=0;i<NUM_OF_IMAGES;++i){
        var plane:Plane=new Plane({width:140,height:140,material:_materialsArr[Math.floor(Math.random() *_materialsArr.length)],bothsides:true});
        plane.x=Math.random()*3000-1500;
        plane.y=Math.random()*3000-1500;
        plane.z=Math.random()*4000+100;
        plane.rotationX = Math.random() * 120 -60;
        plane.rotationY = Math.random() * 120 -60;
        plane.rotationZ = Math.random() * 120 -60;
        _view.scene.addChild(plane);
        plane.addEventListener(MouseEvent3D.MOUSE_DOWN, onMouse3DDown);
    }
    stage.addEventListener(MouseEvent.CLICK, onMouseDown);
}
private function onMouse3DDown(e:MouseEvent3D):void{
    e.stopImmediatePropagation();
    var plane:Plane=e.target as Plane;
    _currentPlane=plane;
```

```
_cameraDummyTarget.transform=plane.transform;
_cameraDummyTarget.moveBackward(40);
_cameraDummyTarget.moveUp(400);
_cameraDummyTarget.pitch(80);
_slerpCam.slerp=0;
var tweenObject:Object={};
tweenObject.x = _cameraDummyTarget.x;
tweenObject.y = _cameraDummyTarget.y;
tweenObject.z = _cameraDummyTarget.z;
tweenObject.slerp=1;//slerp
tweenObject.onUpdate=onTweenProgress;
tweenObject.onComplete=onTweenComplete;
_camQuaternion= Quaternion.createFromMatrix(_slerpCam.
transform); _targetQuaternion=Quaternion.createFromMatrix(_
cameraDummyTarget.transform);///
TweenMax.to(_slerpCam,3,tweenObject);
}
private function onMouseDown(e:MouseEvent):void{
if(!_zoomedIn){
e.preventDefault();
}else{
_zoomedIn=false;
}
_targetQuaternion=Quaternion.createFromMatrix(_cameraDefaultDummy.
transform);///_cameraDefaultDummy
.camQuaternion= Quaternion.createFromMatrix(_slerpCam.
transform);
_slerpCam.slerp=0; TweenMax.to(_slerpCam,3,{x:_
cameraDefaultDummy.x,y:_cameraDefaultDummy.y,z:_cameraDefaultDummy.z,s
lerp:1,onUpdate:onCameraBackTween});
}
private function onCameraBackTween():void{
_slerpQuaternion=Quaternion.slerp(_camQuaternion,_targetQuaternion,_
slerpCam.slerp);
var endMatrix:Matrix3D=Quaternion.quaternion2matrix(_
slerpQuaternion);
var rotMatr:Matrix3D=endMatrix.clone();
rotMatr.position=_slerpCam.position;
_slerpCam.transform=rotMatr;
}
private function onTweenProgress():void {
_slerpQuaternion=Quaternion.slerp(_camQuaternion,_targetQuaternion,_
slerpCam.slerp);
var endMatrix:Matrix3D=Quaternion.quaternion2matrix(_
slerpQuaternion);
```

```
var rotMatr:Matrix3D=endMatrix.clone();
rotMatr.position=_slerpCam.position;
_slerpCam.transform=rotMatr;
}
private function onTweenComplete():void{
    _zoomedIn=true;
}
}
```

After clicking the target gallery image plane, the camera moves towards the object smoothly by interpolating with its transformation. Such a cool effect could be a tricky thing to achieve by means of matrix transformations only:



How it works...

First, we disperse planes all over the scene with random position and rotation using the `initGeometry()` method. Each plane gets a random image assigned with `BitmapMaterial`. We also assign the `MouseEvent3D MOUSE_DOWN` listener for each plane in order to catch mouse presses. There are several important things that can happen when we click on the plane. In the `onMouse3DDown()` function, we get the current clicked plane instance. Now, we created `_cameraDummyTarget`, which is just an empty `Object3D` that is a kind of helper (dummy) that we use in slerp calculation as the target Quaternion. The reason we don't use the target plane is that we want to position the camera opposite to the image and not in the same place. `_cameraDummyTarget` serves us as a customizable position tool for our camera at the end of the transformation process. Next we define a tween object that feeds tween parameters for the `TweenMax`.

You can type these directly inside the TweenMax's `to()` function if you wish. Before we can start the Quaternion transformation of our camera, we need to acquire Quaternion values for the camera as well as for its target. We extract them from their transform matrices:

```
_camQuaternion= Quaternion.createFromMatrix(_slerpCam.transform);  
_targetQuaternion=Quaternion.createFromMatrix(_  
cameraDummyTarget.transform);
```

Then we initiate the tween.

During the tween, TweenMax executes a callback on each position update (using its `onUpdate` event handler) which triggers the `onTweenProgress()` method. In that method, we convert back the current transformation values from the camera Quaternion to its transform matrix in order to rotate it. (Remember that Away3D rotations are matrices-based?). We do it here:

```
slerpQuaternion=Quaternion.slerp(_camQuaternion,_targetQuaternion,_  
slerpCam.slerp);  
var endMatrix:Matrix3D=Quaternion.quaternion2matrix(_  
slerpQuaternion);  
var rotMatr:Matrix3D=endMatrix.clone();  
rotMatr.position=_slerpCam.position;  
_slerpCam.transform=rotMatr;
```

When the camera has arrived at its target, we surely won't be able to tween it back to the default position. For this task, we have another dummy called `_cameraDefaultDummy` that is located at the camera default position. By clicking in the space outside the image plane, we trigger `flash.MouseEvent.CLICK` event that creates a new Quaternion from the `_cameraDefaultDummy` object transformation and another Quaternion from the current camera transformation state and starts again the same routine with `Quaternion.slerp()` as we did before but this time we return to camera default position. Note that we have to reset the camera `slerp` property to zero in order to start a new interpolation. Now let's see where all the Quaternion wonder occurs:

```
Quaternion.slerp(_camQuaternion,_targetQuaternion,_slerpCam.slerp
```

The `slerp()` method basically interpolates between the start transformation Quaternion, that is, which belongs to the camera and the target Quaternion transformation, which is defined by our target dummy object. The `Slerp` property here is a factor that dictates the amount of interpolation to accomplish. If you look at it in percentage, for example, if `slerp` is 0.5 (the range is always between 0 and 1) then only half (50 percent) of the interpolation will be finished. The target value of one fully completes the interpolation.

See also

Learn more on Quaternions on the web or in the books specializing in Math for 3D graphics:

<http://mathworld.wolfram.com/Quaternion.html> and

<http://www.gamedev.net/reference/programming/features/qpowers/default.asp>

3

Animating the 3D World

In this chapter, we will cover:

- ▶ Animating (Rigging) characters in 3DsMax
- ▶ Controlling Bones animation in Collada
- ▶ Working with MD2 animations
- ▶ Morphing objects
- ▶ Animating geometry with Tween engines
- ▶ Moving an object on top of the geometry with Facelink

Introduction

Living in the age of Web2.0 and the realm of **Rich Internet Applications (RIA)**, it is impossible to imagine a world of computer graphics without animation. Today, even the most mind blowing designed website is destined to perish and be escaped by the user who is bored to death in a matter of minutes if it lacks any decent GUI animation. All the more so when we talk about Flash as this program was initially developed to bring a motion into the web. It is obvious that 3D graphics have deeper impact on the user experience. However, no matter how cool your 3D scene looks, containing neat materials with state-of-the-art models, if it is motionless, the chance is high that all your hard work would be underrated or become a complete failure. The world around us is 3D and in constant motion. If you bring the 3D content into web, the next natural step is to animate it.

In this chapter, we will focus on different animation techniques in Away3D. The difference here is—we are going to animate visual objects in 3D space as opposed to 2D in regular Flash applications (well, since introducing Flash Player 10 it is not exactly true). We will learn animating of primitive geometry in 3D space using advanced full featured tween engines as well as using Away3D built-in animation utilities. You will also learn how to set up and animate bones in a 3D modeling program and how to control them inside Away3D. We will see how to morph a 3D object animating its vertices and faces. We are also going to cover some advanced topics such as Inverse Kinematics that will give you some insight into what can be done with a little more math.

Let's get started.

Animating (Rigging) characters in 3DsMax

One of the animation types in 3D is a character animation. Another popular term to call this kind of animation is **character rigging** which means **skeletal animation** in professional circles. If you develop an application such as a virtual world or RPG/FPS game, you will want to have animated models of avatars. Fortunately, the Away3D team developed a full set of tools to parse and control pre-defined animation data of the model. That means you have to animate or rig your model in a 3D program such as 3DsMax or Maya, to get it ready for Away3D.



Away3D deals with two types of external animation—Vertex and Bone animation. **Vertex** animation stores series of individual vertices position and interpolate between them over time. The downside of this technique is a parsing time as models with a large number of animated vertices produce really heavy files. From the other side, this type of animation is best suited for organic deformations such as face or cloth as a smooth result is required which cannot be achieved with bone animation. **Bone** or **skeleton** animation on the other hand works by transforming groups of vertices based on bones transformation matrix data. This type yields much less animation data to parse, but since, in Away3D, it is stored in Collada format which is an XML, the parsing is in many cases slower than that of MD2 vertex animations, which is binary.

Getting ready

Open Autodesk 3dsMax version 7 or later. Go to this chapter's assets folder and open the `spy.max` model file. Alternatively, you can work with your own low poly character model in this example.

How to do it...

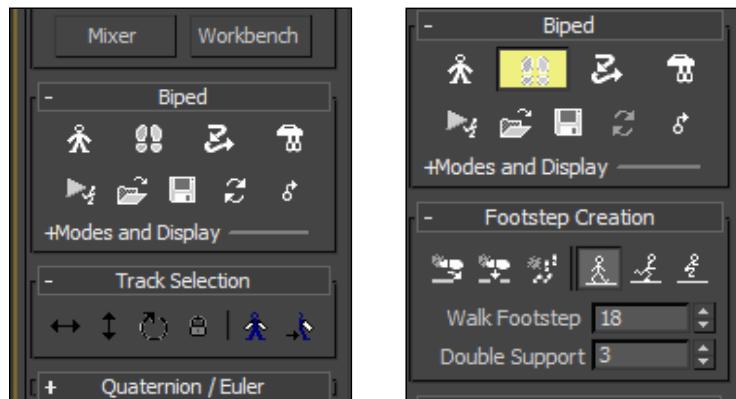
1. In 3DsMax, go to the top menu and click **Create**, and then in the drop-down list, go to **Systems** and select **biped**. Now you can see in the tools side bar the Character studio menu opens.
2. In the active viewport, click the left mouse button and drag it to create a skeleton of the size you need.
3. Center the Spy mesh and the skeleton at the center of the scene.
4. Now select limb bones and move them to position in the center of the related limb of the mesh. This part is very important because if a certain bone doesn't contain all the relevant mesh vertices in its bounding radius, it may result in really ugly artifacts and distortions when animating.
5. The end result should look like this:



In both images, you can see the skeleton bone system filling the Spy mesh. Pay attention at the left (wireframe) view—all the bones are contained exactly inside the mesh.

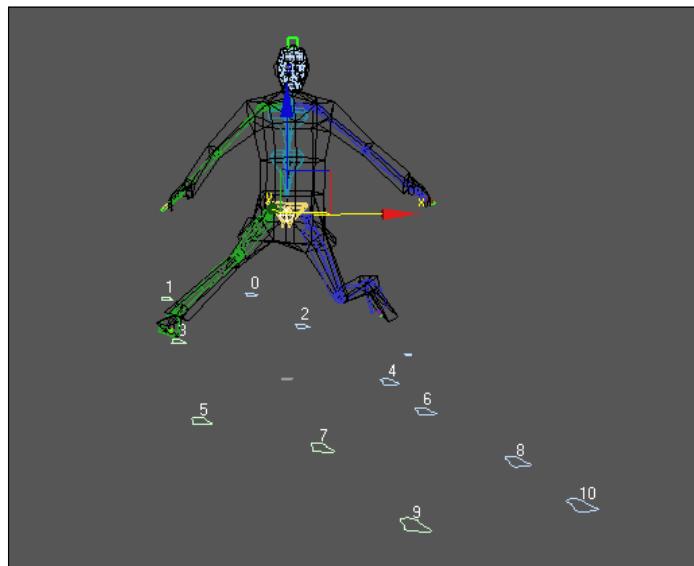
Now we have to apply the skeleton bones to the mesh in order to be able to further rig the character. For this, we use the **Skin** modifier. Select the **Spy mesh** in the viewport and in the side bar, open the modifiers drop-down list, and select **Skin**.

6. In the Skin modifier parameters menu, go to the Bones dialog and click on **Add** button and select all the bones of the biped from the list. Now we can start animating our Spy.
7. We can rig the character the easy or the hard way. The hard one is to define manually the keyframes for each new motion state of character. The easy way is to use the automatic footstep system. For this example, we will choose the first. Character studio allows us to define walk, run, and jump animations in a few easy steps. Also you can load motion capture files if you wish to have a realistic looking animation sequence. Here we will define two sequences—jump and walk using the Character studio's Footstep mode.
8. Click the yellow pelvis of the skeleton in order to access the Character Studio interface at the side bar. In the **Biped** dialog, click a **Footstep mode** button. In the **Footstep** drop-down dialog, click the **Jump** button, then click the **Create steps** button in the same dialog. The following images show the 3DsMax Character Studio Biped Interface.



9. Now focus on the viewport. You can see that the default mouse cursor has changed to the footstep icon. Click inside the viewport to define footsteps for the character. Set six steps for the jump mode.
10. When you are finished, go back to the **Biped** side bar menu and in the Footstep Operations drop-down dialog, click the **Create keys for inactive footsteps**. Now the defined animation is registered to the timeline keyframes.

11. Now, we want to define a walk sequence. Click and drag the timeline scrubber to the frame 50. In the Footstep Creation dialog, select Walk mode and repeat preceding steps 10 and 11.



Now the Spy walks and jumps. As you can see here, the character's motion is defined by the footsteps system which allows us to create pretty astounding results in a matter of minutes.

Now we can export the rigged Spy using the ColladaMax or OpenCollada exporter. When exporting, in the Collada export dialog, you should check **Enable Export** and **Sample animation** checkboxes and adjust the timeline range to your defined animation sequence length.

How it works...

The Biped system is definitely the best choice when one needs to animate a human character. As you have seen it gives us a prebuilt **IK** skeleton consisting of joined bone objects. Biped structure, being a part of the character animation system is a formidable character animation tool. You can find royalty-free sophisticated motion capture files and load them into the 3DsMax to achieve astonishingly realistic human animation effects.

Of course you need to set up a custom rigging that is different from human or animal that you can do with the regular Bones system of 3DsMax. This approach is usually more time consuming than the biped setup.

There's more...

In this example, we touched on only a tiny part of what can be done within 3DsMax character studio. There are many books and online tutorials that can teach you various aspects and advanced techniques of the animation creation process in this 3D program. You can use these resources to improve your animating skills.

See also

- ▶ In *Chapter 9, Working with External Assets*, the following recipe: *Preparing MD2 models for Away3D in MilkShape*
- ▶ *Exporting Model from 3DsMax/Maya/Blender* (using different exporters) from *Chapter 9*

Controlling bones animation in Collada

Having created a rigged model in the 3D modeling program, the next step in the developing process would be importing it into Away3D and setting control on its embedded animation data. Away3D gives us advanced tools to control externally created bones and vertices-based animations. In this example, you will learn how to access and control the bones-based Collada animation. Also, you will see how to separate single animation clips into loops of different animation sequences and how to switch between them.

Getting ready

1. Create a new Away3D scene extending our `AwayTemplate` class.
2. You also need to add to your assets—an animated Collada version of the Spy model. The file resides in the `assets` folder inside this chapter's files directory under the name `spyJumpAndWalk.dae`.
3. From the same folder, also import the `spyObjRe.jpg` image file that we use as the Spy material texture source.

How to do it...

In the following program, we initiate an embedded animated character. We create two control buttons which would activate two different animation sequences—"jump" and "walk", which we set in 3DsMax initially:

ColladaAnimationDemo.as

```
package
{
    public class ColladaAnimationDemo extends AwayTemplate
    {
        [Embed(source="../assets/animatedSpy/spyJumpAndWalk.
dae", mimeType="application/octet-stream")]
        private var SpyModel:Class;
        [Embed(source="../assets/animatedSpy/spyObjRe.jpg")]
        private var SpyTexture:Class;
        private var _bitMat:BitmapMaterial;
        private var _modelDAE:ObjectContainer3D;
        private var _animMode:String="stand";//"jump","walk"//
        private var _skinAnimator:BonesAnimator;
        private var _jumpAnimBut:Button;
        private var _walkAnimBut:Button;
        public function ColladaAnimationDemo()
        {
            super();
            initUI();
        }
        override protected function initMaterials() : void{
            _bitMat=new BitmapMaterial(Cast.bitmap(new SpyTexture()));
        }
        override protected function initGeometry() : void{
            parseDAE();
        }
        private function initUI():void{
            _jumpAnimBut=new Button("start jumping",120,30);
            _walkAnimBut=new Button("start walking",120,30);
            _view.addChild(_jumpAnimBut);_jumpAnimBut.x=-400;_jumpAnimBut.
y=-280;
            _view.addChild(_walkAnimBut);_walkAnimBut.x=-400;_walkAnimBut.
y=-240;
            _jumpAnimBut.addEventListener(MouseEvent.CLICK, onMousePress, fal
se,0,true);
            _walkAnimBut.addEventListener(MouseEvent.CLICK, onMousePress, fal
se,0,true);
        }
        private function onMousePress(e:MouseEvent):void{
            switch(e.currentTarget){
                case _jumpAnimBut:
                    _skinAnimator.gotoAndPlay(1);

```

```
        _animMode="jump";
        break;
    case _walkAnimBut:
        _skinAnimator.stop();
        _skinAnimator.gotoAndPlay(50);
        _animMode="walk";
        break;
    }
}

private function accessAnimationData():void{
    var animLib:AnimationLibrary=_modelDAE.animationLibrary;
    _skinAnimator=animLib.getAnimation("default").animator as
BonesAnimator;
    _skinAnimator.interpolate=true;
    _skinAnimator.addOnEnterKeyFrame(onAnimKeyFrameEnter);
}
private function parseDAE():void{
    var _dae:Collada=new Collada();
    _dae.centerMeshes=true;
    _modelDAE=_dae.parseGeometry(SpyModel)as ObjectContainer3D;
    _view.scene.addChild(_modelDAE);
    _modelDAE.materialLibrary.getMaterial("spy_red").material=_bitMat;
    _modelDAE.scale(1);
    _modelDAE.x=120;
    _modelDAE.y=-170;
    _modelDAE.z=2500;
    accessAnimationData();
}
private function onAnimKeyFrameEnter(e:AnimatorEvent):void{
    if(e.animator.currentFrame>=50){
        if(_animMode=="jump"){
            e.animator.gotoAndPlay(1);
        }
    }
    if(e.animator.currentFrame>=75){
        if(_animMode=="walk"){
            e.animator.gotoAndPlay(50);
        }
    }
}
}
```

And here is the result—our Spy can now walk and jump!



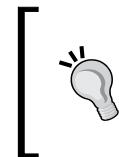
How it works...

When Away3D finishes parsing our embedded model in the `parseDAE()` function, we call the next method named `accessAnimationData()`. In this method, we first access the `AnimationLibrary` of our model that holds all the animation data such as channels and bones:

```
var animData:AnimationLibrary=_modelDAE.animationLibrary;
```

Next we need to access the `AnimationData` of the `AnimationLibrary`. We do it by passing the name of the `AnimationData`. When exporting Collada animation from the 3D modeling program using Collada exporters, the name of the animation is always "default". Therefore we pass it in the following line:

```
_skinAnimator=animLib.getAnimation("default").animator as  
BonesAnimator;
```



It is important to note that we assign our `AnimationData` to `BonesAnimator` variable because the animation is bone-based opposite to using `VertexAnimator` for vertex-based animations like it is done in MD2 format.

Our Spy holds two different animation sequences. The problem is that they are embedded in the same timeline and in Away3D represented as one channel set inside a single `AnimationData` object. That means we have to improvise in order to be able to play these two animations separately. To accomplish that, we first add the following line inside the `accessAnimationData()` function:

```
_skinAnimator.addOnEnterKeyFrame(onAnimKeyFrameEnter);
```

This enables us to get a callback each time the next animation frame is reached. So basically we can track the animation frame position during the animation play.

In the 3DsMax, we have two animation frames ranges—the "jump" animation lasts from 0 to 50 frames and the "walk" animation begins at frame 50 until 75. In the `onAnimKeyFrameEnter()` function, we check these ranges by writing the following code:

```
if(e.animator.currentFrame>=50) {
    if(_animMode=="jump") {
        e.animator.gotoAndPlay(1);

    }
    if(e.animator.currentFrame>=75) {
        if(_animMode=="walk") {
            e.animator.gotoAndPlay(50);
        }
    }
}
```

When one of the two buttons is pressed, inside the `onMousePress()` method, the flag `_animMode` sets the selected animation type to "walk" or "jump". Then the selected sequence starts playing. When the play head of this sequence gets to one of the end frames, which are defined in the previous function, we check whether it is a "walk" or "jump" sequence and reset the play head to the first frame of the related animation span. By this setting, continuous loop is interrupted only when another button is pressed.

See also

Animating (Rigging) characters in 3DsMax.

Working with MD2 animations

MD2 animation format is quite popular among Away3D developers. This is a lightweight binary format and therefore parsed much faster than XML-based Collada animations. In *Chapter 9, Working with External Assets*, you can learn how to set and export MD2 animation using MilkShape software. Here you will learn how to set it up and play in Away3D.

Getting ready

There is a little compatibility problem with the current MilkShape version. When you check the animation sequence name in Flex debugger after the parsing has been finished, you would notice that there are some non-alphabetic characters added to the initial name such as these "-½{x". You have two options in this case—just copy and paste such a name into the `accessAnimationByName()` method parameter right from the debugger, or you can hack a code of Away3D MD2 parser so that when reading the bites for the name property, all the non-alphabetical characters will be ignored. We will go for the second solution:

1. In the `Md2.as`, go to the function `parseFrames(num_frames:int)`.
2. Find the following code block and add the marked lines in the exact places as depicted here:

```
var canAdd:Boolean=true;
for (var j:int = 0; j <16; j++)
{
    var char:int = md2.readUnsignedByte();
    if (char != 0) {
        var str:String = String.fromCharCode(char);
        if (isNaN(Number(str)))

            if ((str.charCodeAt(0) >=65&&str.charCodeAt(0)<=90) || (str.charCodeAt(0) >=97&&str.charCodeAt(0)<=122))
            {
                if (canAdd) {
                    name += str;
                }
            }
        }else if(str.charCodeAt(0)<=48) {
            canAdd=false;
        }
    }
```

Now any name that consists only of letter characters will be concatenated to the name property string. If you want the numeric values to be included as well, just add an additional character range that embraces numbers into the preceding `if()` statement. Also, if you are acquainted with regular expressions, you can implement them instead. Although in this scenario, it would be tricky to define a good expression.

Now set up a basic Away3D scene extending `AwayTemplate`.

Make sure you have the `spyAnimatedReady.md2` and `spyObjRe.jpg` files embedded in your code. It is found in this chapter's assets folder.

How to do it...

```
MD2Animation.as
package
{
    public class MD2Animation extends AwayTemplate
    {
        [Embed(source="../assets/animatedSpy/spyAnimatedReady.
        md2", mimeType="application/octet-stream")]
        private var SpyModel:Class;
        [Embed(source="../assets/animatedSpy/spyObjRe.jpg")]
        private var SpyTexture:Class;
        private var _hoverCam:HoverCamera3D;
        private var _bitMat:BitmapMaterial;
        private var _md2:Md2;
        private var _model:Mesh;
        private var _standBut:Button;
        private var _walksBut:Button;
        private var _move:Boolean = false;
        private var _lastPanAngle:Number;
        private var _lastTiltAngle:Number;
        private var _lastMouseX:Number;
        private var _lastMouseY:Number;
        private var _animStand:VertexAnimator;
        private var _walkAnim:VertexAnimator;
        public function MD2Animation()
        {
            super();
            initButtons();
        }
        private function initHoverCam():void
        {
            _hoverCam = new HoverCamera3D();
            _hoverCam.zoom=3;
            _hoverCam.panAngle = 45;
            _hoverCam.tiltAngle = 20;
            _hoverCam.hover();
            _view.camera = _hoverCam;
        }
        override protected function initMaterials() : void
        {
            _bitMat = new BitmapMaterial(Cast.bitmap(SpyTexture));
        }
    }
}
```

```
override protected function initGeometry() : void{
    _md2 = new Md2();
    _model = _md2.parseGeometry(SpyModel) as Mesh;
    _model.material = _bitMat;
    _model.scale(0.05);
    _view.scene.addChild(_model);
    _animStand=accessAnimationByName("stand",0,false,true,20);
    _walkAnim=accessAnimationByName("walk",0,true,true,20);
}
private function accessAnimationByName(str:String,delay:Number=0,loop:Boolean=true,interp:Boolean=true,fps:Number=30):VertexAnimator{
    var animdata:AnimationLibrary=_model.animationLibrary;
    var anim:VertexAnimator=animdata.getAnimation(str).animator as VertexAnimator;
    anim.delay=delay;
    anim.loop=loop;
    anim.interpolate=interp;
    anim.fps=fps;
    return anim;
}

private function initButtons():void
{
    _standBut = new Button("Stand", 100);
    _standBut.x = 580;
    _standBut.y = 40;
    addChild(_standBut);
    _walksBut = new Button("Walk", 100);
    _walksBut.x = 580;
    _walksBut.y = 80;
    addChild(_walksBut);
}
override protected function initListeners():void
{
    initHoverCam();
    super.initListeners();
    addEventListener(MouseEvent.CLICK, onButtonClick);
    stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
    stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
}
private function onButtonClick(event:Event):void
{
    var button:Button = event.target as Button;
    switch(button) {
        case _standBut:_animStand.play();break;
        case _walksBut:_walkAnim.play();break;
    }
}
```

```
        }
    }
    override protected function onEnterFrame(e:Event):void
    {
        super.onEnterFrame(e);
        if (_model)
            _model.rotationY += 0.5;

        if (_move) {
            _hoverCam.panAngle = 0.3 * (stage.mouseX - _lastMouseX) + _lastPanAngle;
            _hoverCam.tiltAngle = 0.3 * (stage.mouseY - _lastMouseY) + _lastTiltAngle;
        }
        _hoverCam.hover();
    }
    private function onMouseDown(e:MouseEvent):void
    {
        _lastPanAngle = _hoverCam.panAngle;
        _lastTiltAngle = _hoverCam.tiltAngle;
        _lastMouseX = stage.mouseX;
        _lastMouseY = stage.mouseY;
        _move = true;
    }
    private function onMouseUp(e:MouseEvent):void
    {
        _move = false;
    }
}
```



This is an Away3D image of the loaded MD2 model of the Spy. Press **Stand** and **Walk** buttons to trigger different types of animation.

How it works...

In the `initGeometry()` method, after the Spy model has been parsed, we call a custom method `accessAnimationByName()` that we created to access the `AnimationData` of the model:

```
private function accessAnimationByName(str:String,delay:Number=0,loop:Boolean=true,interp:Boolean=true,fps:Number=30):VertexAnimator{
    anim.delay=delay;
    anim.loop=loop;
    anim.interpolate=interp;
    anim.fps=fps;
    return anim;
}
```

The function accepts as its arguments the `AnimationData` name as it was defined when exporting from MilkShape. Other parameters are optional such as delay, loop, interpolation, and fps.

Inside the function, we first access the model's `AnimationLibrary`:

```
var animdata:AnimationLibrary=_model.animationLibrary;
```

Then we access the `AnimationData` itself by name which is vertex animation; therefore we cast the animator to `VertexAnimator` data type:

```
var anim:VertexAnimator=animdata.getAnimation(str).animator as VertexAnimator;
```

At last, before returning it, we assign the method's arguments values to the `VertexAnimator` additional properties:

```
anim.delay=delay;
anim.loop=loop;
anim.interpolate=interp;
anim.fps=fps;
```

Now we have two variables:

```
_animStand=accessAnimationByName("stand",0,false,true,20);
_walkAnim=accessAnimationByName("walk",0,true,true,20);
```

Each one holding the reference to the one different animation sequence of our MD2 model. We call this function by clicking one of the buttons we created to trigger animations:

```
private function onButtonClick(event:Event):void
{
    var button:Button = event.target as Button;
    switch(button) {
        case _standBut:_animStand.play();break;
        case _walksBut:_walkAnim.play();break;
    }
}
```

In the preceding method, we play different animations by clicking the button type.

That is all. MD2 is that easy, isn't it?

See also

- ▶ *Controlling Bones animation in Collada* recipe in this chapter
- ▶ In Chapter 9, *Working with External Assets*, refer to the recipe *Preparing MD2 models for Away3D in MilkShape*

Morphing objects

Dynamic deformation, or morphing of 3D mesh surface, is a widely-used technique in 3D content development. That can be achieved via several ways. You can morph the geometry by moving individual vertices or faces manually. A more elegant and easier way is to use specially developed modifier libraries such as AS3DMOD for this purpose. (See Chapter 10, *Integration with Open Source Libraries*). However, before you run for a non-Away3D solution, you should know that the library has reserved a special class named `Morpher` that resides in the `core.base` package.

In the following example, you will learn how to morph a sphere primitive in runtime using the Away3D `Morpher` tool.

Getting ready

Set up a new Away3D scene using `AwayTemplate` as the base class and we are ready to go.

How to do it...

In the following program, we create three spheres. The first is to be morphed. The second serves as a deformation pattern target for the first sphere. The additional sphere has no deformation applied and serves for resetting the deformed sphere shape to the initial.

MorphingDemo.as

```
package
{
    public class MorphingDemo extends AwayTemplate
    {
        private var _sphere:Sphere;
        private var _morphPattern:Sphere;
        private var _phongBit:PhongBitmapMaterial;
        private var _dirLight:DirectionalLight3D;
        private var _button:Button;
        private var _morpher:Morpher;
        private var _hoverCam:HoverCamera3D;
        private var _move:Boolean = false;
        private var _lastPanAngle:Number;
        private var _lastTiltAngle:Number;
        private var _lastMouseX:Number;
        private var _lastMouseY:Number;
        private var _defaultSphere:Sphere;
        public function MorphingDemo()
        {
            super();
            initHoverCam();
            initLight();
            initUI();
        }
        private function initHoverCam():void
        {
            _hoverCam = new HoverCamera3D();
            _hoverCam.zoom=22;
            _hoverCam.panAngle = 45;
            _hoverCam.tiltAngle = 20;
            _hoverCam.hover();
            _view.camera = _hoverCam;
            _hoverCam.target=_sphere;
        }
        private function initLight():void{
```

```
_dirLight=new DirectionalLight3D();
_dirLight.color=0xf3c3c5;
_dirLight.specular=0.23;
_dirLight.brightness=4.23;
_dirLight.diffuse=4;
_dirLight.ambient=4;
_view.scene.addLight(_dirLight);
_dirLight.direction=_sphere.position;

}
override protected function initListeners():void
{
    super.initListeners();
    stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
    stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
}
override protected function initMaterials() : void{
    var btmd:BitmapData=new BitmapData(256,256);
    btmd.perlinNoise(24,24,9,17,true,true,7,false);
    _phongBit=new PhongBitmapMaterial(btmd);
    _phongBit.smooth=true;
}
override protected function initGeometry() : void{
    _defaultSphere=new Sphere({radius:70,segmentsW:20,segmentsH:20})
    _sphere=new Sphere({radius:70,material:_phongBit,segmentsW:20,s
egmentsH:20});
    _morphPattern=new Sphere({radius:70,material:_phongBit,segmentsW
:20,segmentsH:20});
    _view.scene.addChild(_sphere);
    _sphere.z=500;
    _morphPattern.y=-140;
    _morphPattern.x=100;
    _morphPattern.z=500;
    deformPattern();
    _morpher=new Morpher(_sphere);

}
private function initUI():void{
    _button=new Button("distort more",140);
    _button.y=-260;
    _button.x=-260;
    _view.addChild(_button);
    _button.addEventListener(MouseEvent.CLICK,onMouseClick,false,0,
true);
```

```

        }

    private function onMouseClick(e:MouseEvent):void{
        if(_morpher){
            deformPattern();
        }
    }

    private function deformPattern():void{
        for(var i:int=0;i<_morphPattern.vertices.length;++i){
            var vert:Vertex=_morphPattern.vertices[i] as Vertex;
            var multFactor:Number=Math.random()*0.5;
            var vector:Vector3D=vert.position;
            vector.scaleBy(multFactor);
            vert.add(vector);
        }
    }

    override protected function onEnterFrame(e:Event) : void{
        super.onEnterFrame(e);
        if(_morpher){
            _morpher.start();
            _morpher.mix(_morphPattern,1+Math.sin(getTimer()/1000));

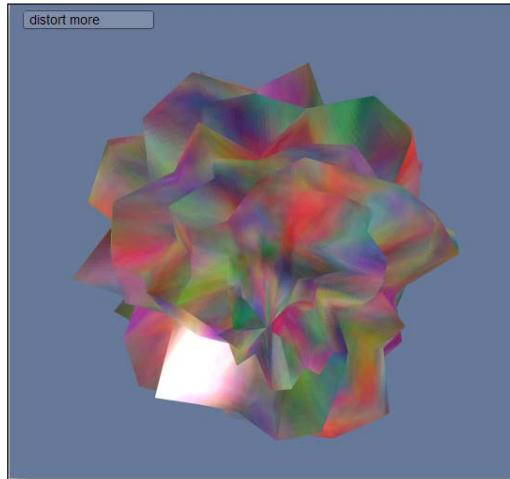
            _morpher.finish(_defaultSphere);
        }
        if (_move) {
            _hoverCam.panAngle = 0.3 * (stage.mouseX - _lastMouseX) + _lastPanAngle;
            _hoverCam.tiltAngle = 0.3 * (stage.mouseY - _lastMouseY) + _lastTiltAngle;
        }
        _hoverCam.hover();
    }

    private function onMouseDown(e:MouseEvent):void
    {
        _lastPanAngle = _hoverCam.panAngle;
        _lastTiltAngle = _hoverCam.tiltAngle;
        _lastMouseX = stage.mouseX;
        _lastMouseY = stage.mouseY;
        _move = true;
    }

    private function onMouseUp(e:MouseEvent):void
    {
        _move = false;
    }
}

```

Here is a nice looking flower-like deformation effect of the sphere that you should see when running the preceding code:



How it works...

Let's explain the preceding code. Inside the `initGeometry()` method, we create three spheres:

```
_defaultSphere=new Sphere({radius:70,segmentsW:20,segmentsH:20})
    _sphere=new Sphere({radius:70,material:_phongBit,segmentsW:20,s
egmentsH:20});
    _morphPattern=new Sphere({radius:70,material:_phongBit,segmentsW
:20,segmentsH:20});
    _view.scene.addChild(_sphere);
```

Each of those is used by the `Morpher` modifier. The `Morpher` class works in the following way. For its arguments, it gets the object to deform; in our case, it is `_sphere`. The second object, `_morphPattern`, holds target vertices extrusions coordinates, which we set beforehand by randomly pushing the `_morphPattern` vertices in the following method:

```
private function deformPattern():void{
    for(var i:int=0;i<_morphPattern.vertices.length;++i){
        var vert:Vertex=_morphPattern.vertices[i] as Vertex;
        var multFactor:Number=Math.random()*0.5;
        var vector:Vector3D=vert.position;
        vector.scaleBy(multFactor);
        vert.add(vector);
    }
}
```

Morpher animates the `_sphere` extrusion until its vertices reach the same object local coordinates as those of `_morphPattern`. The last is the `_defaultSphere` object that undergoes no deformation and is used by the Morpher as a pattern to reset vertex extrusions to their initial position at the end of each animation interaction, which is processed in `onEnterFrameFunction()`:

```
_morpher.start();
_morpher.mix(_morphPattern, 1+Math.sin(getTimer()/1000));
_morpher.finish(_defaultSphere);
```

It is important to note that the `Morpher mix()` method's second parameter sets the amount of target distortion and we use it to animate the morphing process, while adding a sinusoidal behavior using this line:

```
_morpher.mix(_morphPattern, 1+Math.sin(getTimer()/1000));
```

Also in the `initUI()` method, we create a button to process the `_morphPattern` sphere vertices deformation again. Click it and you will see that the morphed sphere becomes distorted even more.

Now go along and have fun with it!

See also

- ▶ In Chapter 13, Doing More with Away3D, the following recipe: *Creating cool effects with animated normal maps*

Animating geometry with Tween engines

When you need to animate your 3D objects position, tween engines are usually the best solutions not only for basic transformations, but even for powering complex transitions such as those found in particle effects. There is a wide range of open-source as well as commercial tweening engines such as TweenMax (Green Sock), GTween (Grant Skinner), Tweener (Zeh Fernando), BeTween, and more. Here, as well as in the rest of the book, we will use GreenSock's TweenMax engine which is considered to be one of the fastest and best featured on the market. It is free for non-commercial use.

In the following program, we are going to create a transition of a group of sprites in Away3D space between three different initial formations—random disperse, cube formation, and sphere formation. You will see that pretty cool effects may be accomplished when leveraging the potential of a tween engine.

The credit goes to Mr. Doob (<http://mrdoob.com/blog>) who wrote a breathtaking demo "Depth of Field" which served as the inspiration for this recipe.



Theoretically, any tweening engine is suitable to use for this example if it enables you to create groups of sequential tweens. Please refer to the documentation of your favorite engine in order to determine whether such a feature exists.

Getting ready

Create a basic Away3D scene extending our `AwayTemplate` class. Make sure you have the `TweenMax swc` library attached to your project.

Now we are ready to go.

How to do it...

In the following example, we will create three arrays of vertex positions of cube and sphere primitives as well as one random array. Then, using `TweenMax`, we interpolate the transitions of `DepthOfFieldSprite` particles objects position based on the previously mentioned arrays of vertices coordinates:

`TweenEngineDemo.as`

```
package
{
    public class TweenEngineDemo extends AwayTemplate
    {
        private static const MAX_NUM:int = 180;
        private var _animObjectsArr:Array=[];
        private var _matArrL:Array=[];
        private var _targetVertices:Array=[];
        private var _canProcess:Boolean=false;
        private var _hoverCam:HoverCamera3D;
        private var _move:Boolean = false;
        private var _lastPanAngle:Number;
        private var _lastTiltAngle:Number;
        private var _lastMouseX:Number;
        private var _lastMouseY:Number;
        private var _mainContainer:ObjectContainer3D;
        private var _spriteMat:BitmapMaterial;

        public function TweenEngineDemo()
        {
            super();
        }
    }
}
```

```
initTweenData();
initHoverCam();
DofCache.usedof=true;
DofCache.maxblur=28;
DofCache.focus=20;
DofCache.aperture=23;
DofCache.doflevels=10;
}
override protected function initListeners() : void{
    super.initListeners();
    stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
    stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
}
override protected function initMaterials() : void{
    var bdata:BitmapData=new BitmapData(64,64);
    bdata.perlinNoise(12,12,15,18,false,true,7,false);
    _spriteMat=new BitmapMaterial(bdata);

}
private function initHoverCam():void
{
    _hoverCam = new HoverCamera3D();
    _hoverCam.zoom=3;
    _hoverCam.panAngle = 45;
    _hoverCam.tiltAngle = 20;
    _view.camera = _hoverCam;
    _hoverCam.target=_mainContainer;

}
private function initTweenData():void{
    _mainContainer=new ObjectContainer3D();
    _view.scene.addChild(_mainContainer);
    var vertexPosArr:Array=[];
    vertexPosArr[0]=[];
    var cube:Cube=new Cube({width:1200,height:300,depth:1200});
    cube.segmentsD=10;
    cube.segmentsH=10;
    cube.segmentsW=10;
    for (var i:int =0; i < cube.vertices.length /3-10; i++){
        vertexPosArr[ 0 ][ i ] = new Vector3D(cube.vertices[ i*3 ].x, cube.vertices[ i*3 + 1 ].y, cube.vertices[ i*3 + 2 ].z);
    }
}
```

```
var sphere:Sphere=new Sphere({radius:500});
sphere.segmentsH=25;
sphere.segmentsW=25;
vertexPosArr [1] = [];
for (var f:int = 0; f < sphere.vertices.length / 3-10; ++f)  {
    vertexPosArr[ 1 ][ f ] = new Vector3D(sphere.vertices[ f*3
].x, sphere.vertices[ f*3 + 1 ].y, sphere.vertices[ f*3 + 2 ].z);
}
vertexPosArr[2]=[];
for (var k:int=0; k < MAX_NUM; ++k){

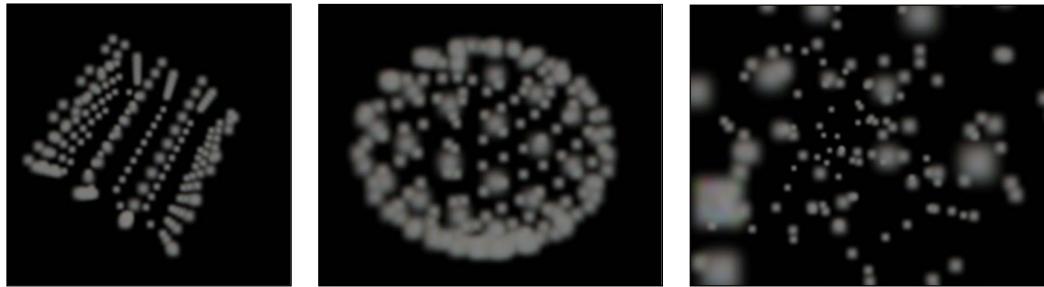
    vertexPosArr[ 2 ][ k ] = new Vector3D((Math.random() - 0.5) *
2000, (Math.random() - 0.5) * 2000, (Math.random() - 0.5) * 2000);
}
for (var v:int = 0; v < MAX_NUM; v++) {
    var colMat:ColorMaterial=new ColorMaterial(Math.floor(Math.
random()*0xffffffff));
    _matArrL[v]=colMat;
    var sprite:DepthOfFieldSprite=new DepthOfFieldSprite(_
spriteMat);
    _view.scene.addSprite(sprite);
    _animObjectsArr[v]=sprite;

}
/////////init tweens///////////
var tweenGroup:Array=[];
var tweenSequence:TimelineMax=new TimelineMax();
for (var d:int = 0; d < MAX_NUM; d++) {
    var tween1:Vector3D = vertexPosArr[ 0 ][ d ];
    var tween2:Vector3D = vertexPosArr[ 1 ][ d ];
    var tween3:Vector3D = vertexPosArr[ 2 ][ d ];
    _targetVertices[d]=tween2;
    var tMax1:TweenMax=new TweenMax(_targetVertices[d],4,{ x:
tween1.x, y: tween1.y, z: tween1.z,ease:Linear.easeInOut});
    var tMax1a:TweenMax=new TweenMax(_targetVertices[d],4,{ x:
tween3.x, y: tween3.y, z: tween3.z,ease:Linear.easeInOut });
    var tMax2:TweenMax=new TweenMax(_targetVertices[d],4,{ x:
tween2.x, y: tween2.y, z: tween2.z,ease:Linear.easeInOut });
    var tMax2a:TweenMax=new TweenMax(_targetVertices[d],4,{ x:
tween1.x, y: tween1.y, z: tween1.z,ease:Linear.easeInOut });
    var tMax3:TweenMax=new TweenMax(_targetVertices[d],4,{ x:
tween3.x, y: tween3.y, z: tween3.z,ease:Linear.easeInOut });
    var tMax3a:TweenMax=new TweenMax(_targetVertices[d],4,{ x:
tween2.x, y: tween2.y, z: tween2.z,ease:Linear.easeInOut });


```

```
tweenSequence.  
insertMultiple([tMax1,tMax1a,tMax2,tMax2a,tMax3,tMax3a],0,TweenAlign.  
SEQUENCE,1);  
}  
tweenSequence.delay=1;  
tweenSequence.repeat=1200;  
tweenSequence.play();  
_canProcess=true;  
}  
private function onMouseDown(e:MouseEvent):void  
{  
    _lastPanAngle = _hoverCam.panAngle;  
    _lastTiltAngle = _hoverCam.tiltAngle;  
    _lastMouseX = stage.mouseX;  
    _lastMouseY = stage.mouseY;  
    _move = true;  
}  
private function onMouseUp(e:MouseEvent):void  
{  
    _move = false;  
}  
override protected function onEnterFrame(e:Event) : void{  
    super.onEnterFrame(e);  
    if(_canProcess){  
        for (var i:int = 0; i < MAX_NUM; i++) {  
            _animObjectsArr[i].x=_targetVertices[i].x;  
            _animObjectsArr[i].y=_targetVertices[i].y;  
            _animObjectsArr[i].z=_targetVertices[i].z;  
        }  
    }  
    if (_move) {  
        _hoverCam.panAngle = 0.3 * (stage.mouseX - _lastMouseX) + _  
lastPanAngle;  
        _hoverCam.tiltAngle = 0.3 * (stage.mouseY - _lastMouseY) +  
_lastTiltAngle;  
        _hoverCam.hover();  
    }  
}
```

Here you can see the resulting three transformations of the sprites. The first two are based on mesh vertices vectors and the last is a random formation:



How it works...

As you can see, all the fun stuff is inside the `initTweens()` method. We create the `_mainContainer` object to serve us as the scene center dummy object to rotate around with `HoverCamera`. Then we start a routine of filling a two dimensional array `vertexPosArr` with vertex position data, which we run three times, each time for the different geometrical form. First for a cube primitive formation:

```
var cube:Cube=new Cube({width:1200,height:300,depth:1200});  
cube.segmentsD=10;  
cube.segmentsH=10;  
cube.segmentsW=10;  
cube.visible=false;  
_view.scene.addChild(cube);  
for (var i:int =0; i < cube.vertices.length /3-10; i++){  
  
    vertexPosArr[ 0 ][ i ] = new Vector3D(cube.vertices[ i*3  
].x, cube.vertices[ i*3 + 1 ].y, cube.vertices[ i*3 + 2 ].z);  
  
}
```

In the preceding code block, inside the for loop, we have a `vertexPosArr` array where its first dimension is the primitive type (three in total) and the second keeps position in `Vector3D` for each vertex that we select on a cube mesh. Notice that I deliberately pick up a vertex whose index is three times smaller than the one from the same one from the next loop. This way, we create a nicely recessed effect of our particles formation.

Next, we run the same loop, but this time for the `Sphere` primitive:

```
varsphere:Sphere=new Sphere({radius:500});  
sphere.segmentsH=25;  
sphere.segmentsW=25;  
vertexPosArr [1] = [];
```

```
for (var f:int = 0; f < sphere.vertices.length / 3-10; ++f) {  
    vertexPosArr[ 1 ][ f ] = new Vector3D(sphere.vertices[ f*3  
] .x, sphere.vertices[ f*3 + 1 ].y, sphere.vertices[ f*3 + 2 ].z);  
}
```

And the last formation that we need to store to our `vertexPosArr` array is random positions. We define an area of 2000 pixels in width, height, and depth to calculate random coordinates:

```
vertexPosArr [2] = [] ;  
for (var k:int=0; k < MAX_NUM; ++k){  
  
    vertexPosArr[ 2 ][ k ] = new Vector3D((Math.random() - 0.5) *  
2000, (Math.random() - 0.5) * 2000, (Math.random() - 0.5) * 2000);  
}
```

Next we create the actual objects to serve us as particles. We use `DepthOfFieldSprite` sprites so that we can apply DOF effect to them which will give us a cool puffy look:

```
for (var v:int = 0; v < MAX_NUM; v++) {  
    var colMat:ColorMaterial=new ColorMaterial(Math.floor(Math.  
random()*0xffffffff));  
    _matArrL[v]=colMat;  
  
    var sprite:DepthOfFieldSprite=new DepthOfFieldSprite(_spriteMat);  
    _view.scene.addSprite(sprite);  
  
    _animObjectsArr[v]=sprite;  
}
```

Now comes the tweening part. Our goal here is to create a set of tween transformations between our predefined three different formations by grouping them into sequences of tweens.

We extract three different vertices position data formations for three different types of tweens this way:

```
var tween1: Vector3D =vertexPosArr[ 0 ][ d ];  
var tween2: Vector3D = vertexPosArr[ 1 ][ d ];  
var tween3: Vector3D = vertexPosArr[ 2 ][ d ];
```

Then we define which formation of vertices is going to be default:

```
_targetVertices [d]=tween2;
```

`_targetVertices` array contains the vertices position in `Vector3D` which we will tween.

Now we should create three pairs of tweens—first transforms from formation one to three, second from two to one, and third from three to two. By packing all these tweens into a group, we receive an animation sequence consisting of these three pairs of tweens:

```
tweenSequence.  
insertMultiple([tMax1,tMax1a,tMax2,tMax2a,tMax3,tMax3a],0,TweenAlign.  
SEQUENCE,1);
```

Note that in `tweenSequence`, which is the instance of `TimelineMax`, we use the `TweenAlign.SEQUENCE` mode in order to tween each tween consequentially.

Now, by starting up the application, we will see nothing because what we tween are sheer numbers that represent the vertex's position in 3D space. We need to connect our sprites that we created earlier to these numbers. We do it through the `onEnterFrame()` method:

```
for (var i:int = 0; i < MAX_NUM; i++) {  
    _animObjectsArr[i].x=_targetVertices[i].x;  
    _animObjectsArr[i].y=_targetVertices[i].y;  
    _animObjectsArr[i].z=_targetVertices[i].z;  
}
```

Now you can see what kind of stuff you can do with or without a single line of trigonometry using just your favorite tween engine.

Moving an object on top of the geometry with FaceLink

You may have a scenario where you wish to animate the position of one object on top of the surface of another. The Away3D `FaceLink` class, located in the `away3d.animators` package, serves that goal. `FaceLink` calculates for you the average coordinates of three vertices of each face and lets you tween your object from one face position to another in a sequential manner as well as to offset that position by demand.

In the next example, we will create a utility which allows us to adjust the position of a sapper relative to the Spy's mesh. This way, instead of guessing manually where Spy hand vertices are located, we run through all of them, animating the position of the sapper until it comes to the hand area.

Getting ready

Set up a new Away3D scene using `AwayTemplate`. In this example, we use a knob control to animate the sapper position from the great components library by Keith Peters called **MinimalComps**. You can get it for free here at <http://www.minimalcomps.com/>.

Make sure you embed the following models (found in Chapter 3's assets folder) into the application:spyObjReady.3ds, spyRadio.3ds and their related textures:spyObjReady.jpg, sapper_flat.jpg.

How to do it...

In the following program, we will add the Spy model and his sapper kit to the scene. Using the knob control, we will animate the position of the sapper on top of the Spy mesh surface:

FaceLinkDemo.as

```
package
{
    public class FaceLinkDemo extends AwayTemplate
    {
        [Embed(source="../assets/spyDifferentFormatsExport/spyObjReady.3ds",
        mimeType="application/octet-stream")]
        private var SpyModel3ds:Class;
        [Embed(source="../assets/spyDifferentFormatsExport/spyObjReady.
        jpg")]
        private var SpyTexture:Class;
        [Embed(source="../assets/spyDifferentFormatsExport/spyRadio.3ds",
        mimeType="application/octet-stream")]
        private var SpyRadio:Class;
        [Embed(source="../assets/spyDifferentFormatsExport/sapper_flat.
        jpg")]
        private var RadioTexture:Class;
        private var _knobUI:Knob;
        private var _model3ds:ObjectContainer3D;
        private var _radio:ObjectContainer3D;
        private var _bitMat:BitmapMaterial;
        private var _hoverCam:HoverCamera3D;
        private var _move:Boolean = false;
        private var _lastPanAngle:Number;
        private var _lastTiltAngle:Number;
        private var _lastMouseX:Number;
        private var _lastMouseY:Number;
        private var _faceLink:FaceLink;
        private var _demoSph:Sphere;
        private var _extractedMesh:Mesh;
        private var _radioMaterial:BitmapMaterial;
        public function FaceLinkDemo()
        {
            super();
        }
    }
}
```

```
    initHoverCam();
    initUI();
    initFaceLink();
}

private function initHoverCam():void
{
    _hoverCam = new HoverCamera3D();
    _hoverCam.zoom=22;
    _hoverCam.panAngle = 45;
    _hoverCam.tiltAngle = 20;
    _view.camera = _hoverCam;
    _hoverCam.target=_extractedMesh;
}
private function initUI():void{
    var fps:FPSMeter=new FPSMeter(_view,250,-250);
    fps.scaleX=2;
    fps.scaleY=2;
    _knobUI=new Knob(_view,-250,-250,"animate position",onKnobMove);
    _knobUI.draw();
    _knobUI.mouseRange=400;
    _knobUI.labelPrecision=0;
    _knobUI.minimum=0;
    _knobUI.maximum=_extractedMesh.faces.length;
    fps.draw();
    fps.start();
}
private function initFaceLink():void{
    _faceLink=new FaceLink(_radio,_extractedMesh,
_extractedMesh.faces[0],10,true);
}
private function parse3ds():void{
    var max3ds:Max3DS=new Max3DS();
    _model3ds=max3ds.parseGeometry(SpyModel3ds)as ObjectContainer3D;
    _model3ds.materialLibrary.getMaterial("spy_red").material=_bitMat;
    _extractedMesh=_model3ds.children[0];
    _view.scene.addChild(_extractedMesh);
    _extractedMesh.z=300;
    max3ds=new Max3DS();
    _radio=max3ds.parseGeometry(SpyRadio)as ObjectContainer3D;
    _radio.materialLibrary.getMaterial("sapper").material=_radioMaterial;
    _view.scene.addChild(_radio);
}
private function onKnobMove(e:Event):void{
    var indexVal:uint=Math.floor(e.target.value);
```

```
if(indexVal<_extractedMesh.faces.length){
    _faceLink.face=_extractedMesh.faces[indexVal];
}
    _faceLink.update();
}
override protected function initListeners() : void{
    super.initListeners();
    stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
    stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
}
override protected function initMaterials() : void{
    _bitMat=new BitmapMaterial(Cast.bitmap(new SpyTexture()));
    _radioMaterial=new BitmapMaterial(Cast.bitmap(new
RadioTexture()));
}
override protected function initGeometry() : void{
    parse3ds();
}

}
override protected function onEnterFrame(e:Event) : void{
    super.onEnterFrame(e);
    if (_move) {
        _hoverCam.panAngle = 0.3 * (stage.mouseX - _lastMouseX) + _lastPanAngle;
        _hoverCam.tiltAngle = 0.3 * (stage.mouseY - _lastMouseY) + _lastTiltAngle;
    }
    _hoverCam.hover();
}
private function onMouseDown(e:MouseEvent):void
{
    _lastPanAngle = _hoverCam.panAngle;
    _lastTiltAngle = _hoverCam.tiltAngle;
    _lastMouseX = stage.mouseX;
    _lastMouseY = stage.mouseY;
    _move = true;
}

private function onMouseUp(e:MouseEvent):void
{
    _move = false;
}
}
```

Now the Spy has the equipment in his hand. Thanks to FaceLink, you save a lot of your precious time trying to locate the desired vertex area of your mesh:



How it works...

Let's go over the crucial parts of the code. In `initUI()`, we initialize Knob control (see the documentation for more info on the MinimalComps web page). We assign the knob values range from zero to the number of total faces of the Spy model. This way, when we drag the knob to its maximum, we reach the last face position with our animated sapper object:

```
_knobUI.minimum=0;  
_knobUI.maximum=_extractedMesh.faces.length;
```

Next, in the `initFaceLink()` method, we initiate an instance of the `FaceLink` class, passing to its constructor the following arguments by order: mesh to link (our sapper), mesh to link to (Spy model), initial face position, position offset of the linked object, and whether the object gets aligned to the normal vector of the face:

```
_faceLink=new FaceLink(_radio,_extractedMesh,_extractedMesh.  
faces[0],10,true);
```



Note that `_extractedMesh` is the actual mesh object of the Spy we extracted from the initial `ObjectContainer3D`, right after completing the Spy model parsing and added it to the scene. If you passed to `FaceLink` just the reference to the nested mesh (like `this:_model3ds.children[0]`), located by default inside the Spy `ObjectContainer3D`, you would get alternating distance of tweened objects from the model's faces during the animation. That is because, in such a case, the mesh vertices coordinates are relative to the parent object and not to the global space.

Finally, in the `onKnobMove()` method, which is called each time the knob values change, we shift the position of the sapper by assigning the current face according to the index value provided by knob control. After each new assignment, we update the new position of the linked object by calling the `_faceLink.update()` method:

```
private function onKnobMove(e:Event):void{
    var indexVal:uint=Math.floor(e.target.value);
    if(indexVal<_extractedMesh.faces.length){
        _faceLink.face=_extractedMesh.faces[indexVal];
    }
    _faceLink.update();
}
```

Now run the program and see how easily we can adjust the position of the sapper in the hand of the Spy model. `faceLink` may serve you by being a very useful tool when mapping your areas of an avatar for all kinds of attachments.

4

Fun by Adding Interactivity

In this chapter, we will cover:

- ▶ Adding rotational interactivity to an Away3D primitive by using Mouse movements
- ▶ Implementing advanced object rotation using vector math
- ▶ Creating advanced spherical surface transitions with Quaternions
- ▶ Interactively painting on the model's texture
- ▶ Dragging on geometry by unprojecting mouse coordinates
- ▶ Morphing mesh interactively
- ▶ Creating a controllable non-physical car

Introduction

One of the major differences between the Internet and other types of media is that it enables users to interact with the content in a more direct way. When we surf a website or play online games, we expect them to be featured with an advanced set of interactive GUI. In fact, today we can't even imagine a web page with no ability to touch, turn, and move around the content. Otherwise, we could be quite satisfied just watching a TV set just like we did in the past. When we talk about 3D virtual environment, the interactivity plays a critical role as well. Imagine if in the real world we weren't able to control physical objects around us! The same regards to a real-time 3D application. Having a mind-blowing 3D scene with animated meshes flying all around it is not enough unless the user is also enabled to control them. That is where concealed ultimate user experience is.

In this chapter, we will learn some common as well as advanced techniques to make your project interactive. We will see how to perform advanced interactive transformations in 3D space, objects dragging, and even create a full-featured interactive car.

Sometimes implementation of advanced interactivity demands from you a decent knowledge of math. Here you will also learn that this is not as scary as it sounds. Let's get started.

Adding rotational interactivity to an Away3D primitive by using Mouse movements

In this recipe, you will learn how to add basic interactivity to an Away3D primitive that is rotated according to Mouse movements on the screen. Although this feature is far from being rocket science, it is not uncommon to see an incorrect implementation of it in many applications which results in quite unnatural or imperfect behavior of interacted objects.

Getting ready

Set up a new Away3D scene extending our `AwayTemplate` class and you are ready to go.

How to do it...

In the next program, we create a typical Away3D sphere primitive. Our goal is to rotate the sphere around its x- and y- axis according to the mouse's x and y direction movement on the stage:

`InteractiveRotate.as`

```
package
{
    public class InteractiveRotate extends AwayTemplate
    {
        private var _sphere:Sphere;
        private var _bitMat:BitmapMaterial;
        private var _lastSphereXRot:Number=0;
        private var _lastSphereYRot:Number=0;
        private var _lastMouseX :Number=0;
        private var _lastMouseY :Number=0;
        private var _lastMouseXUP:Number=0;
        private var _lastMouseYUP :Number=0;
    }
}
```

```
private var _canMove:Boolean=false;
private const EASE:Number=0.2;
public function InteractiveRotate()
{
    super();
}
override protected function initGeometry():void{
    _sphere=new Sphere({radius:80,material:_bitMat,segmentsH:20,seg
mentsW:20});
    _view.scene.addChild(_sphere);
    _sphere.z=600;
}
override protected function initListeners():void{
    super.initListeners();
    stage.addEventListener(MouseEvent.MOUSE_DOWN,
onMouseDown,false,0,true);
    stage.addEventListener(MouseEvent.MOUSE_UP,
onMouseUp,false,0,true);
}
override protected function initMaterials():void{
    var bdata:BitmapData=new BitmapData(256,256);
    bdata.perlinNoise(26,26,18,1534,false,true,7,true);
    _bitMat=new BitmapMaterial(bdata);
}
override protected function onEnterFrame(e:Event):void{
    super.onEnterFrame(e);
    _sphere.rotationY =(stage.mouseX - _lastMouseX)*EASE+ _lastSphereXRot;
    _sphere.rotationX=(stage.mouseY - _lastMouseY)*EASE+ _lastSphereYRot;
}
private function onMouseDown(e:MouseEvent):void
{
    _lastSphereXRot = _sphere.rotationX;
    _lastSphereYRot =_sphere.rotationY;
    _lastMouseX = stage.mouseX;
    _lastMouseY = stage.mouseY;
    _canMove = true;
    stage.addEventListener(Event.MOUSE_LEAVE, onStageMouseLeave);
}
private function onMouseUp(e:MouseEvent):void
{
```

```
_lastMouseXUP=Math.abs(_lastMouseX-stage.mouseX)/1000;
_lastMouseYUP=Math.abs(_lastMouseY-stage.mouseY)/1000;
trace(_lastMouseXUP);
_canMove = false;
stage.removeEventListener(Event.MOUSE_LEAVE, onStageMouseLeave);
}
private function onStageMouseLeave(event:Event):void
{
    _canMove = false;
    stage.removeEventListener(Event.MOUSE_LEAVE, onStageMouseLeave);
}
}
```

How it works...

The magic formula from all the preceding code consists of just two lines of code which run in the `onEnterFrame()` function:

```
_sphere.rotationY=(stage.mouseX - _lastMouseX)+_lastSphereYRot
_sphere.rotationX=(stage.mouseY - _lastMouseY)+_lastSphereXRot;
```

 It is important to note that this approach to object rotation in 3D space is in no way considered a precise one as it might seem from the first glance. The object is being rotated by the values derived from the delta of the last and current mouse position. More sophisticated interactivity is implemented in a system called **Track** or **Ark Ball**, in which rotation is totally 3D vector math-based and therefore it delivers results that are much more realistic. It will be discussed in the following recipe.

We add the `MouseEvent.MOUSE_DOWN` and `MouseEvent.MOUSE_UP` to the stage as we are concerned about rotating the sphere when pressing and dragging the mouse in a certain direction. Inside the `onMouseDown()` method, we update the following variables:

```
_lastSphereXRot = _sphere.rotationX;
_lastSphereYRot = _sphere.rotationY;
_lastMouseX = stage.mouseX;
_lastMouseY = stage.mouseY;
_canMove = true;
```

We register the `_lastSphereXRot` and `_lastSphereYRot` so that when the rotation starts, we resume the new rotation from the current one, thus preventing rotation reset which otherwise results in visually inconsistent behavior. We also set `_lastMouseX` and `_lastMouseY` in order to calculate a delta value between the `MOUSE_DOWN` click coordinates and the current coordinates values being acquired when we are dragging the mouse across the stage. These delta values then increment the rotation values of the sphere.

There's more...

Now let's add more realism to the behavior we created in the preceding example. It would be nice if the sphere could spin around with some kind of acceleration, if the mouse moves fast, and easing down when the mouse has been released. Here is how you can accomplish this.

In the member variables block, add these two lines:

```
private var _lastMouseXUP:Number=0;  
private var _lastMouseYUP :Number=0;
```

Add a new event listener for the `MOUSE_MOVE` event inside the `onMouseDown()` function:

```
stage.addEventListener(MouseEvent.MOUSE_MOVE, onPressedMouseMove) ;
```

And in the `onMouseUp` handler, we should remove this event, so add the following line of code:

```
stage.removeEventListener(MouseEvent.MOUSE_MOVE, onPressedMouseMove) ;
```

Now create the event handler method for `MOUSE_MOVE` called `onPressedMouseMove` and add the following lines into it:

```
_lastMouseXUP=Math.abs(_lastMouseX-stage.mouseX)/1000;  
_lastMouseYUP=Math.abs(_lastMouseY-stage.mouseY)/1000;
```

These two new variables are assigned the values for acceleration and easing which we use in the `onEnterFrame()` function to accelerate and slow down the sphere rotation when we leave the mouse. The values are in the range between zero and one. The greater the value, the less easing is applied to the sphere on each frame, and it spins faster and longer. We must update these values while we drag the rotation with the mouse. Then after we release the button, the rotation is updated but also eased by the last `_lastMouseXUP` and `_lastMouseYUP` values before the mouse release was executed.

Now comment out the previous code in the `onEnterFrame()` method and add the following instead:

```
if (_canMove) {  
    _sphere.rotationY=(stage.mouseX - _lastMouseX)*EASE+ _  
    lastSphereXRot;
```

```
_sphere.rotationX=(stage.mouseY - _lastMouseY)*EASE+ _  
lastSphereYRot;  
    }else{  
        if(_lastMouseXUP>=1){  
            _lastMouseXUP=0.95;  
        }  
        if(_lastMouseYUP>=1){  
            _lastMouseYUP=0.95;  
        }  
        _lastMouseX*=_lastMouseXUP;  
        _lastMouseY*=_lastMouseYUP;  
        _sphere.rotationY +=_lastMouseX;  
        _sphere.rotationX +=_lastMouseY;  
    }  
}
```

Here we divide the rotation into two states—when the mouse is pressed and dragged across the stage and the state after it has been released. In the first state, we handle the sphere rotation just as we did in the first scenario. But when we release the mouse, we are simulating the acceleration of the rotation velocity of the object with the following lines:

```
_lastMouseX*=_lastMouseXUP;  
_lastMouseY*=_lastMouseYUP;  
_sphere.rotationY +=_lastMouseX;  
_sphere.rotationX +=_lastMouseY;
```

Also, we need to check whether the easing value that holds `_lastMouseXUP` and `_lastMouseYUP` is smaller than one, otherwise the sphere will rotate continuously. We achieve this by inserting the following lines before applying the easing:

```
if(_lastMouseXUP>=1){  
    _lastMouseXUP=0.95;  
}  
if(_lastMouseYUP>=1){  
    _lastMouseYUP=0.95;  
}
```

And we are done.

See also

Implementing advanced object rotation using vector math recipe in this chapter.

Implementing advanced object rotation using vector math

In this recipe, you will learn how to write more realistic rotations of sphere in 3D space which is called rotation, also known as **virtual track ball**. This effect is accomplished through vector calculations derived from Mouse scene position and the rotation itself is processed not around the default local axis but rather based on an arbitrary one, which is acquired through pretty simple vector math as well. The advantage of this approach is that the result is very realistic and interactively accurate.

Getting ready

Set up a new Away3D scene using AwayTemplate and you are ready to begin.

How to do it...

The following program creates a sphere which is rotated by a user's mouse drag using vector product calculations based on Mouse input and not simple screen mouse position delta values:

```
AdvancedRotation.as

package
{
    public class AdvancedRotation extends AwayTemplate
    {
        private var _sphere:Sphere;
        private var _bitMat:BitmapMaterial;
        private var _canMove:Boolean=false;
        private var _sphereMatrix:Matrix3D=new Matrix3D();
        public function AdvancedRotation()
        {
            super();
            _cam.z=-800;
        }
        override protected function initGeometry():void{
            _sphere=new Sphere({radius:100,material:_bitMat,segmentsH:20,se
gmentsW:20});
            _view.scene.addChild(_sphere);
            _sphere.position=new Vector3D(0,0,400);
        }
    }
}
```

Fun by Adding Interactivity —————

```
override protected function initListeners():void{
    super.initListeners();
    stage.addEventListener(MouseEvent.MOUSE_DOWN,
onMouseDown,false,0,true);
    stage.addEventListener(MouseEvent.MOUSE_UP,
onMouseUp,false,0,true);
}
override protected function initMaterials():void{
    var bdata:BitmapData=new BitmapData(256,256);
    bdata.perlinNoise(26,26,18,1534,false,true,7,true);
    _bitMat=new BitmapMaterial(bdata);
}

override protected function onEnterFrame(e:Event):void
{
    if(_canMove){
        var vector:Vector3D=new Vector3D(0,0,1);
        var mouseVector:Vector3D=new Vector3D(400 - mouseX, mouseY -
300, 0);
        var resVector:Vector3D;
        resVector=vector.crossProduct(mouseVector);
        resVector.normalize();
        var mtr:Matrix3D=new Matrix3D();
        var rotDegrees:Number=Math.PI*Math.sqrt(Math.pow( mouseX-400 ,
2) + Math.pow(300-mouseY, 2))/100;
        mtr.appendRotation(rotDegrees ,new Vector3D(resVector.x,
resVector.y,resVector.z),_sphere.pivotPoint);
        _sphereMatrix.append(mtr);
        _sphereMatrix.position=_sphere.position;
        _sphere.transform=_sphereMatrix;
    }
    super.onEnterFrame(e);
}

private function onMouseDown(e:MouseEvent):void
{
    _canMove = true;
    stage.addEventListener(Event.MOUSE_LEAVE, onStageMouseLeave);
}
private function onMouseUp(e:MouseEvent):void
{
    _canMove = false;
```

```
        stage.removeEventListerner(Event.MOUSE_LEAVE, onStageMouseLeave);
    }
    private function onStageMouseLeave(e:Event):void
    {
        _canMove = false;
        stage.removeEventListerner(Event.MOUSE_LEAVE, onStageMouseLeave);
    }
}
```

How it works...

The only block of code that is interesting here is inside the `onEnterFrame()` method. Our first goal is to calculate a rotation axis for the sphere. Because we do not want to fix the rotation to the default x-, y-, and z- local axes of the object, but rather find it according to the mouse direction angle, we need to get a cross product which returns a vector perpendicular to the two other vectors lying in the same plane. We acquire the cross product vector from the local z-axis and from the vector that we filled with the mouse X and Y positions:

```
var vector:Vector3D=new Vector3D(0,0,1);
var mouseVector:Vector3D=new Vector3D(400 - mouseX, mouseY - 300, 0);
var resVector:Vector3D;
resVector=vector.crossProduct(mouseVector);
resVector.normalize();
```

We normalize the resulting vector as we are not interested in its magnitude but in the direction only. The next step is to create a matrix and populate its rotation portion with our new rotation axis and the angle by which we wish to rotate, and then to assign it to the sphere in order to transform it (in our case, just rotate). We set it with the following lines:

```
var mtr:Matrix3D=new Matrix3D();
var rotDegrees:Number=Math.PI*Math.sqrt(Math.pow( mouseX-400 , 2 ) + Math.pow(300-mouseY, 2))/100;
mtr.appendRotation(rotDegrees ,new Vector3D(resVector.x, resVector.y,resVector.z),_sphere.pivotPoint);
    sphereMatrix.append(mtr);
```

The last thing you have to do in case your 3D object is not positioned on 0,0,0 of the scene is to populate the resulting matrix translation values as well. Otherwise, when you start rotating, the sphere will move to the center of the coordinate system because the resulting Matrix3D translation values equal zero by default. We fix this easily by adding the position to them manually:

```
sphereMatrix.position= sphere.position;
```

The last thing to do is to assign the resulting matrix to the transform setter of the sphere:

```
sphere.transform = sphereMatrix;
```

As you can see, there was quite a lot of math here, but the result is pretty impressive. Math is definitely our best friend when we need to create more sophisticated results.

See also

Chapter 11, Creating Virtual Trackball.

Creating advanced spherical surface transitions with Quaternions

There are scenarios when you want to move an object on top of a surface of geometry according to the mouse coordinates. Initially, the task is quite simple if we deal with plain surfaces. But once you would like to do such a movement on top of spherical objects, things get much more complicated. If you attempt to tween your (let's call it) satellite, assigning it just the coordinates derived from the surface mouse click point, you will see that if the transition distance is large, your satellite will go through the sphere mesh on its path to the destination coordinates. The solution is a spherical movement or **spherical interpolation (SLERP)**, which can be achieved using Quaternion. Quaternion math is pretty hard to grasp for a regular human being, but Away3D supplies us with a Quaternion class which encapsulates all the required functions that processes all those complex calculations. So the only thing we should know is how and when to use it.

Getting ready

Set up a new Away3D scene using `AwayTemplate` and don't panic as we will dive a little bit deeper into 3D math.

Make sure you copy to your project a modified Quaternion class that is found in the `ActionScript Classes utils` folder. The difference from Away3D Quaternion is some additional methods that are essential to accomplish our task. Alternatively, you can just copy all those additional functions into Away3D Quaternion.

How to do it...

In this program, we set up a sphere and a small plane that moves on top of a sphere's surface at some distance to the position based on coordinates acquired when clicking the sphere with the mouse:

```
SurfaceMoving.as

package
{
    public class SurfaceMoving extends AwayTemplate
```

```
{  
    private var _sphere:Sphere;  
    private var _bitMat:BitmapMaterial;  
    private var _tracker:Plane;  
    private var _clickAngleX:Number=0;  
    private var _clickAngleY:Number=0;  
    private var _clickAngleZ:Number=0;  
    private var clickVectorOld:Vector3D=new Vector3D(0,0,0);  
    private var clickVector:Vector3D=new Vector3D(0,0,0);  
    private static const DEGStoRADS:Number=Math.PI/180;  
    private var tweenObj:Object;  
    private var _endMatr:Matrix3D;  
    private var _endQ:utils.Quaternion;  
    private var qStart:utils.Quaternion;  
    private var _tweenFinished:Boolean=true;  
    public function SurfaceMoving()  
    {  
        super();  
        _cam.z=-800;  
    }  
    override protected function initGeometry():void{  
        _sphere=new Sphere({radius:100,material:_bitMat,segmentsH:20,se  
gmentsW:20});  
        _view.scene.addChild(_sphere);  
        _sphere.position=new Vector3D(0,0,0);  
        _sphere.addOnMouseDown(onObjMouseDown);  
        _tracker=new Plane({width:40,height:40,material:new  
ColorMaterial(0x229834)});  
        _tracker.bothsides=true;  
        _tracker.yUp=false;  
        _view.scene.addChild(_tracker);  
        _tracker.position=new Vector3D(100,100,0);  
        _tracker.lookAt(_sphere.position);  
    }  
    override protected function initListeners():void{  
        super.initListeners();  
    }  
    override protected function initMaterials():void{  
        var bdata:BitmapData=new BitmapData(256,256);  
        bdata.perlinNoise(26,26,18,1534,false,true,7,true);  
        _bitMat=new BitmapMaterial(bdata);  
    }  
}
```

Fun by Adding Interactivity —————

```
        }
        private function onObjMouseDown(e:MouseEvent3D):void{
            if(_tweenFinished){
                clickVector=new Vector3D(e.sceneX,e.sceneY,e.sceneZ);
                clickVector.scaleBy(2.5);
                clickVector.normalize();
                qStart=new Quaternion();
                qStart.x=_tracker.x;
                qStart.y=_tracker.y;
                qStart.z=_tracker.z;
                qStart.w=0;
                var startVec:Vector3D=_tracker.position;
                startVec.normalize();
                var cross:Vector3D=startVec.crossProduct(clickVector);
                var rotAngle:Number=Math.acos(startVec.dotProduct(clickVector));
                var q:Quaternion=new Quaternion();
                q.axis2quaternion(cross.x,cross.y,cross.z,rotAngle);
                var invertedQ:Quaternion=Quaternion.conjugate(q);
                _endQ=new Quaternion();
                _endQ.multiply(q,qStart);
                _endQ.multiply(q,_endQ);
                tweenObj=new Object();
                tweenObj.intrp=0;
                TweenMax.to(tweenObj,1,{intrp:1,onUpdate:onTweenUpdate,
                onComplete:onTweenFinished});
                    _tweenFinished=false;
                    trace(_tracker.position);
                }
            }
            private function onTweenFinished():void{
                _tweenFinished=true;
            }
            private function onTweenUpdate():void{
                var slerped:utils.Quaternion=utils.Quaternion.slerp(qStart,
                endQ, tweenObj.intrp);
                var endMatr:Matrix3D=new Matrix3D();
                endMatr.appendTranslation(slerped.x,slerped.y,slerped.z);
                _tracker.x=endMatr.position.x;
                _tracker.y=endMatr.position.y;
                _tracker.z=endMatr.position.z;
                _tracker.lookAt(_sphere.position);
            }
        }
    }
```

How it works...

Now let's explain step-by-step what is going on here. The show begins in the `onObjMouseDown()` function. First, we get the mouse click vector of the sphere surface, and scale it up a little in order to keep our satellite plain in some distance from the sphere surface:

```
clickVector=new Vector3D(e.sceneX,e.sceneY,e.sceneZ);  
clickVector.scaleBy(2.5);  
clickVector.normalize();
```

The next step we need is to prepare two Quaternions—one for the default transformation and another for the target. Later we are going to interpolate between them using the SLERP method of the `Quaternion` class. First comes the default position of the `_tracker` (our satellite):

```
_qStart=new Quaternion();  
_qStart.x=_tracker.x;  
_qStart.y=_tracker.y;  
_qStart.z=_tracker.z;  
_qStart.w=0;
```

In order to prepare the target Quaternion, we have some more work to do. First we should calculate the cross product of current tracker position and target position in order to get a new rotation axis:

```
var startVec:Vector3D=_tracker.position;  
startVec.normalize();  
var cross:Vector3D=new Vector3D();  
cross=startVec.crossProduct(clickVector);
```

Now we need to calculate a Quaternion for the acquired rotational axis. We extract this Quaternion from our cross product and the angle between the default `_tracker` position and the target which is derived from dot product calculation:

```
var rotAngle:Number=Math.acos(startVec.dotProduct(clickVector));  
var q:Quaternion=new Quaternion();  
q.axis2quaternion(cross.x, cross.y, cross.z,rotAngle);
```

Next, we will negate this Quaternion as not doing so will move our `_tracker` to the target location but on the opposite side of the sphere:

```
var invertedQ:Quaternion=Quaternion.conjugate(q);
```

Finally, we approach the last step in setting up the Quaternions. We calculate the target Quaternion by first multiplying between inverted and the default `_tracker` position Quaternion. Then we multiply once again the resulting Quaternion from the first multiplication with the axis Quaternion we acquired in the previous step:

```
endQ=new Quaternion();
```

```
_endQ.multiply(q,qStart);  
_endQ.multiply(q,_endQ);
```

Now we are ready to interpolate between our Quaternions. TweenMax is going to perform the tween of the SLERP value which ranges always between zero and one. To complete the interpolation we should tween the value to one (that is like 100%):

```
_tweenObj=newObject();  
_tweenObj.intrp=0;  
TweenMax.to(_tweenObj,1,{intrp:1, onUpdate:onTweenUpdate, onComplete:  
onTweenFinished});
```

Now in TweenMax, as you can see, we define a callback method for the `onUpdate` event that is called on each tween update. Inside this handler, we perform the actual transformation update of our `_tracker` calculating the current SLERPed Quaternion, and then converting it to the Matrix3D and finally to transform property of the object:

```
private function onTweenUpdate():void{  
  
    var slerped:utilsQuaternion=utilsQuaternion.slerp(qStart,_endQ,  
    tweenObj.intrp);  
    var endMatr:Matrix3D=new Matrix3D();  
    endMatr.appendTranslation(slerped.x,slerped.y,slerped.z);  
    _tracker.x=endMatr.position.x;  
    _tracker.y=endMatr.position.y;  
    _tracker.z=endMatr.position.z;  
    _tracker.lookAt(_sphere.position);  
}
```

Also note that during the translation, we force the `_tracker` to look at the center of the `_sphere`. That causes the `_tracker` plane to face the same direction as the surface normal of the face it hovers above. That is to be perpendicular to it. In this case, we could make a more precise calculation based on the actual normal direction of each face beneath the tracker, but the final result we get is pretty much the same.

There's more...

You can learn more on Quaternion transformations from the following sources:

www.mathworld.wolfram/Quaternion.html

Book: "Visualizing Quaternions", (Morgan Kaufmann series) by A J Hanson.

See also

Chapter 2, Working with Away3D Cameras—Using Quaternion camera transformations for advanced image gallery viewing recipe.

Interactively painting on the model's texture

Adding interactivity to your object not always means to just make it responsive to the mouse click, or transform it around the scene. You may have much more fun if you are able to also change the way your object looks. In this recipe, you will learn how to paint on the material of your 3D object in the runtime interactively using the mouse as a brush.

Getting ready

Set up a new Away3D scene using `AwayTemplate`.

In this example, we use a `ColorChooser` component to switch between different colors from a drop-down palette it supplies or by hex values input. The `ColorChooser` is one of many great components developed by Keith Peters and called **MinimalComps**. You can get it for free here at <http://www.minimalcomps.com/>.

How to do it...

In the following scenario, we set up a sphere primitive to which we apply a `MovieMaterial` with the input of Perlin Noise `BitmapData`. By clicking and dragging the mouse on the sphere surface, you are able to paint on it with different colors defined via the `ColorChooser` component:

```
DragMapAndPaint.as

package
{
    public class DragMapAndPaint extends AwayTemplate
    {
        private var _sphere:Sphere;
        private var _bitMat:BitmapMaterial;
        private var _movieMat:MovieMaterial;
        private var brush:Sprite=new Sprite();
        private var _canPaint:Boolean=false;
        private var _selectedColor:uint=0xffffffff;
        public function DragMapAndPaint()
        {
            super();
        }
    }
}
```

Fun by Adding Interactivity —————

```
    setLens();
    initColorPicker();
}

private function setLens():void{
    _view.camera.lens=new PerspectiveLens();
}

override protected function initGeometry():void{
    _sphere=new Sphere({radius:100,material:_movieMat,segmentsH:20,segmentsW:20});
    _view.scene.addChild(_sphere);
    _sphere.position=new Vector3D(0,0,400);
    _sphere.addOnMouseDown(onMouse3DDown);
    _sphere.addOnMouseUp(onMouseUp);
    _sphere.addOnMouseMove(onMouseMove);
}

override protected function initMaterials():void{
    var bdata:BitmapData=new BitmapData(256,256);
    bdata.perlinNoise(26,26,18,1534,false,true,7,false);
    var defaultSprite:Sprite=new Sprite();
    defaultSprite.graphics.beginBitmapFill(bdata);
    defaultSprite.graphics.drawRect(0,0,256,256);
    defaultSprite.graphics.endFill();
    _movieMat=new MovieMaterial(defaultSprite);
    _movieMat.interactive=true;
    _movieMat.smooth=true;
}

private function initColorPicker():void{
    var colorPiker:ColorChooser=new ColorChooser(this,100,100,0xcccc
cc,onColorPick);
    colorPiker.usePopup=true;
    colorPiker.draw();
}

private function onColorPick(e:Event):void{
    var validLengthString:String=e.target.value.toString();
    _selectedColor=e.target.value;
}

private function onMouse3DDown(e:MouseEvent3D):void{
    _canPaint=true;
}

private function onMouseUp(e:MouseEvent3D):void{
    _canPaint=false;
}
```

```
private function onMouseMove(e:MouseEvent3D):void{
    if(_canPaint){
        draw(e.uv.u,1-e.uv.v);
    }
}
private function draw(u:Number,v:Number):void{
    var mClip:Sprite=_movieMat.movie as Sprite;
    mClip.graphics.beginFill(_selectedColor);
    mClip.graphics.drawCircle(u*mClip.width,v*mClip.height,2);
    mClip.graphics.endFill();

}
}
}
```

This is not a Picasso masterpiece, but it can be real fun for your end users to be able to paint in the 3D world:



How it works...

First we set up a sphere primitive in the `initGeometry()` method and assign to it three event listeners—all for `MouseEvent3D`:

```
_sphere=new Sphere({radius:100,material:_movieMat,segmentsH:20,segmen
tsW:20});
_view.scene.addChild(_sphere);
```

```
_sphere.position=new Vector3D(0,0,400);  
_sphere.addOnMouseDown(onMouse3DDown);  
_sphere.addOnMouseUp(onMouseUp);  
_sphere.addOnMouseMove(onMouseMove);
```

In the next step, we apply a MovieMaterial which contains a procedural Perlin Noise BitmapData:

```
var bdata:BitmapData=new BitmapData(256,256);  
bdata.perlinNoise(26,26,18,1534,false,true,7,false);  
var defaultSprite:Sprite=new Sprite();  
defaultSprite.graphics.beginBitmapFill(bdata);  
defaultSprite.graphics.drawRect(0,0,256,256);  
defaultSprite.graphics.endFill();  
_movieMat=new MovieMaterial(defaultSprite);
```

Now the behavior for our brush is when the user presses the mouse, the painting starts and continues as long as the mouse is being pressed. The familiar paint effect you can see in Adobe Photoshop is called an **Airbrush**. To achieve this, we call the `onMouseDown` event handler once the mouse was pressed, where we set a Boolean `_canPaint` to true. Additionally, we have an event handler `onMouseMove()` that is called with each mouse movement. Inside `onMouseMove()`, when `_canPaint` is set to true, we start calling the `draw()` method which executes the drawing on the MovieMaterial's MovieClip:

```
private function onMouseMove(e:MouseEvent3D):void{  
    if(_canPaint){  
        draw(e.uv.u,1-e.uv.v);  
    }  
}
```

This way, it is enough to press the mouse just once and the drawing will stay continuous until it is released. The `draw()` method receives two arguments which are uv coordinates of the material texture. uv-coordinates system range from zero to one for the u- and v-axis (similar to x and y), therefore we need to convert it into Cartesian coordinates and match it to the dimensions of our material texture. Also note that we subtract the `e.uv.v` value from one when we pass it into the `draw()` method. That is because contrary to the Cartesian system, the v-(y) axis in uv or barycentric space is flipped:

```
var mClip:Sprite=_movieMat.movie as Sprite;  
mClip.graphics.beginFill(_selectedColor);  
mClip.graphics.drawCircle(u*mClip.width,v*mClip.height,2);  
mClip.graphics.endFill();
```

In the preceding block, we draw a circle directly to the `MovieMaterialMovieClip` with the color passed through the `_selectedColor` variable which gets its value from the `ColorChooser` component.

There's more...

One limitation of the drawing technique previously described is that we draw the vector data directly in the `MovieClip`. If you wish to change the blending mode for each brush stroke or even apply filters to it, then you need to draw `shape` objects through which you are able to add filters or change `BlendingMode` property.

Inside the `draw()` function, comment out all the lines and put the following instead:

```
var shape:Shape=new Shape();
shape.blendMode=BlendMode.ADD;
shape.graphics.beginFill(_selectedColor);
shape.graphics.drawCircle(u*mClip.width,v*mClip.height,2);
shape.graphics.endFill();
mClip.addChild(shape);
```

Now we wrap the drawing data inside the `shape` object which allows us to apply to it different blending modes and then add it to material's `MovieClip` as a regular `DisplayObject` using the `addChild()` method.

We are done!

See also

In *Chapter 6, Using Text and 2D Graphics to Amaze*, the following recipe *Drawing with segments in 3D*.

Dragging on geometry by unprojecting mouse coordinates

You can manipulate your 3D object in 3D space without any need for complex vector math calculations. `MouseEvent3D` enables you to extract global 3D coordinates of the mesh based on the mouse position. The only job left for you is to assign those coordinates to any object on the scene and you have a 3D mouse-based interactive transformation. In the following example, you will learn how to move an object (a small plane) with the help of the mouse on top of the surface of another plane which is slightly elevated.

Getting ready

Set up a new Away3D scene using `AwayTemplate` and you are ready to go.

How to do it...

PlaneUnproject.as

```
package
{
    public class PlaneUnproject extends AwayTemplate
    {
        private var _tracker:Plane;
        private var _unprojVector:Vector3D;
        private var _phongMat:PhongColorMaterial;
        private var _light:PointLight3D;
        private var _plane:Plane;
        public function PlaneUnproject()
        {
            super();
            _view.camera.lens=new PerspectiveLens();
            _cam.z=-500;
            _cam.y=-400;
            initLight();
        }
        override protected function initMaterials() : void{
            _phongMat=new PhongColorMaterial(0x339944);
        }
        override protected function initGeometry() : void{
            _plane=new Plane({width:1000,height:1000,material:new ShadingColorMaterial(0x229933)});
            _plane.segmentsH=_plane.segmentsW=8;
            _view.scene.addChild(_plane);
            _plane.rotationX=30;
            _plane.position=new Vector3D(0,0,800);
            _tracker=new Plane({width:100,height:100,material:new ColorMaterial(0x446633)});
            _tracker.bothsides=true;
            _view.scene.addChild(_tracker);
            var plpos:Vector3D=_plane.position;
            plpos.y+=200;
            plpos.z-=100;
            _cam.lookAt(plpos);
        }
        override protected function initListeners() : void{
```

```
super.initListeners();
    _plane.addEventListener(MouseEvent3D.MOUSE_MOVE, onMouseMove);
}
override protected function onEnterFrame(e:Event) : void{

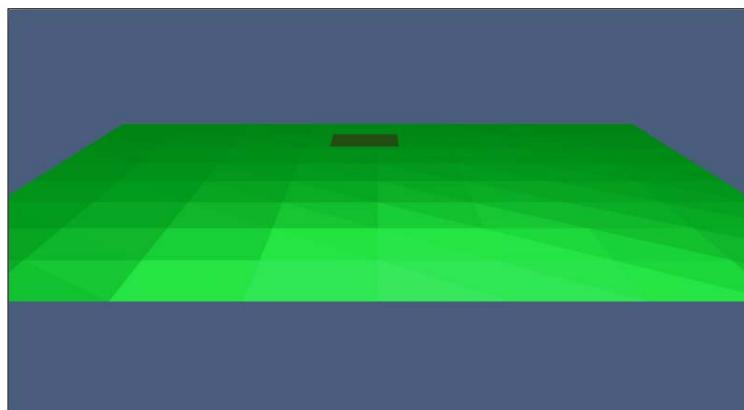
    if(_unprojVector) {

        _tracker.transform=_plane.transform.clone();
        _tracker.x=_unprojVector.x;
        _tracker.y=_unprojVector.y;
        _tracker.z=_unprojVector.z;
        _tracker.translate(Vector3D.Y_AXIS,50);
    }

    super.onEnterFrame(e);
}
private function initLight():void{
    _light=new PointLight3D();
    _view.scene.addLight(_light);
    _light.position=new Vector3D(50,50,50);
}
private function onMouseMove(e:MouseEvent3D):void{
    _unprojVector=new Vector3D(e.sceneX,e.sceneY,e.sceneZ);

}
}
```

In the next image, we have a ground plane on top of which floats a small marker plane. The marker plane is moved in 3D space receiving its position from mouse screen coordinates which are transformed (unprojected) into 3D space:



How it works...

After we initiated the scene geometry in the `initGeometry()` method, we add the `MouseEvent3D.MOUSE_MOVE` event listener to the `_plane` object in order to retrieve 3D coordinates for its surface when the mouse pointer overlaps it. When the `onMouseMove()` function gets called, we extract the 3D coordinates on the `_plane` from the `MouseEvent3D` event:

```
_unprojVector=new Vector3D(e.sceneX,e.sceneY,e.sceneZ);
```

The last thing we need to do is to update the `_tracker` position as well as rotation (so it faces the sloped `_plane` floor).

These lines inside the `onEnterFrame()` method take care of it:

```
override protected function onEnterFrame(e:Event) : void{  
  
    if(_unprojVector){  
  
        _tracker.transform=_plane.transform.clone();  
        _tracker.x=_unprojVector.x;  
        _tracker.y=_unprojVector.y;  
        _tracker.z=_unprojVector.z;  
        _tracker.translate(Vector3D.Y_AXIS,50);  
    }  
    super.onEnterFrame(e);  
}
```

Because we need to offset Y of the `_tracker` along plane's normal, we perform an additional matrix calculation adding an offset to the y-translation portion; then we reassign the updated matrix to `_tracker.transform`.

Piece of cake, isn't it?

See also

- ▶ *Creating advanced spherical surface transitions with Quaternions* recipe in this chapter
- ▶ *Transforming objects in 3D space, relative to the camera position*

Morphing mesh interactively

Have you ever dreamt of being a sculptor? Well if you had, now you have an opportunity to make your dream true. If not, it can still be useful to know how to deform (morph) a mesh of your 3D object with a mouse if you plan to develop a first flash 3D real-time modeling application.

This recipe will teach you how to perform basic mesh morphing by pushing object faces in interactive ways. This is not going to be as powerful as Pixologic ZBrush, but it is all up to you as a developer to make maximum use of it.

Getting ready

Set up a new Away3D scene using `AwayTemplate` and you are ready to go.

How to do it...

In the following program, we set up a plane primitive which we are going to be able to deform by pushing or pulling its faces according to the mouse location:

`InteractiveMorph.as`

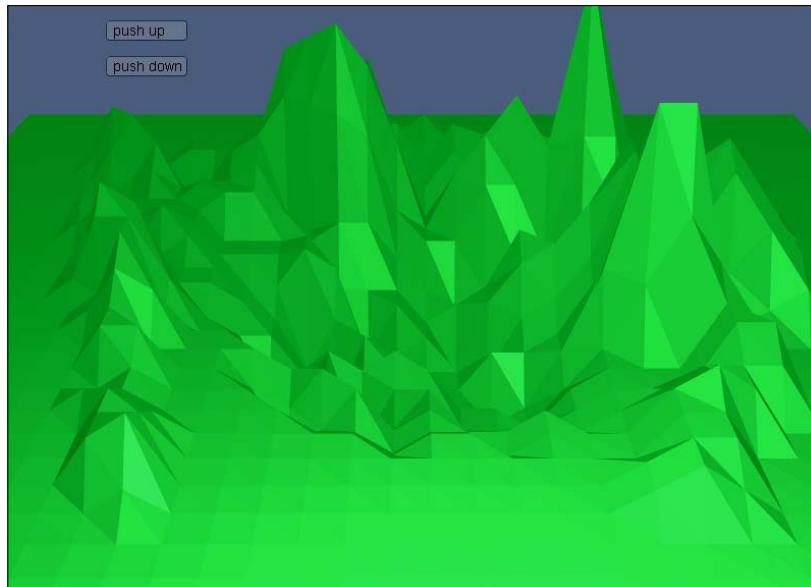
```
package
{
    public class InteractiveMorph extends AwayTemplate
    {
        private var _light:PointLight3D;
        private var _plane:Plane;
        private var _pushVector:Vector3D=new Vector3D(0,20,0);
        private var _but1:Button;
        private var _but2:Button;
        private var _canExtrude:Boolean=false;
        public function InteractiveMorph()
        {
            super();
            _view.camera.lens=new PerspectiveLens();
            _cam.z=-100;
            initLight();
            initControls();
        }
        override protected function initGeometry() : void{
```

Fun by Adding Interactivity —————

```
    _plane=new Plane({width:1000,height:1000,material:new ShadingColorMaterial(0x229933)});  
    _plane.segmentsH=_plane.segmentsW=35;  
    _view.scene.addChild(_plane);  
    _plane.rotationX=30;  
    _plane.position=new Vector3D(0,0,800);  
    _plane.ownCanvas=true;  
  
}  
override protected function initListeners() : void{  
    super.initListeners();  
    _plane.addEventListener(MouseEvent3D.MOUSE_DOWN,onMouse3dDown);  
    _plane.addEventListener(MouseEvent3D.MOUSE_MOVE,onMouse3dMove);  
    _plane.addEventListener(MouseEvent3D.MOUSE_UP,onMouse3dUp);  
    _plane.addEventListener(MouseEvent3D.MOUSE_OUT,onMouse3dOut);  
}  
override protected function onEnterFrame(e:Event) : void{  
  
    super.onEnterFrame(e);  
}  
private function initControls():void{  
    _but1=new Button("push up");  
    _but2=new Button("push down");  
    _view.addChild(_but1);_view.addChild(_but2);  
    _but1.x=_but2.x=-300;  
    _but1.y=-280;_but2.y=_but1.y+35;  
    _but1.addEventListener(MouseEvent.CLICK,onMouseClicked);  
    _but2.addEventListener(MouseEvent.CLICK,onMouseClicked);  
}  
private function onMouseClicked(e:MouseEvent):void{  
    switch(e.currentTarget){  
        case _but1:  
            _pushVector=new Vector3D(0,20,0);  
            break;  
        case _but2:  
            _pushVector=new Vector3D(0,-20,0);  
            break;  
    }  
}  
private function initLight():void{  
    _light=new PointLight3D  
    _view.scene.addLight(_light);
```

```
    _light.position=new Vector3D(50,50,50);
}
private function onMouse3dDown(e:MouseEvent3D):void{
    _canExtrude=true;
}
private function onMouse3dMove(e:MouseEvent3D):void{
    var face:Face=FaceVO(e.elementVO).face;
    if(_canExtrude&&face is Face){
        for(var i:int=0;i<face.vertices.length;++i){
            face.vertices[i].setValue(face.vertices[i].x+_
pushVector.x,face.vertices[i].y+_pushVector.y,face.vertices[i].z+_
pushVector.z);
        }
    }
}
private function onMouse3dUp(e:MouseEvent3D):void{
    _canExtrude=false;
}
private function onMouse3dOut(e:MouseEvent3D):void{
    _canExtrude=false;
}
}
```

The next image shows a simple landscape carved interactively. Adding more polygons to the mesh will give much smoother results, unfortunately at the cost of performance:



How it works...

After setting up our plane in the `initGeometry()` function, next we need to add four listeners in order to register mouse events like when user press, moves, releases, and releases outside the geometry:

```
override protected function initListeners() : void{
    super.initListeners();
    _plane.addEventListener(MouseEvent3D.MOUSE_DOWN, onMouse3dDown);
    _plane.addEventListener(MouseEvent3D.MOUSE_MOVE, onMouse3dMove);
    _plane.addEventListener(MouseEvent3D.MOUSE_UP, onMouse3dUp);
    _plane.addEventListener(MouseEvent3D.MOUSE_OUT, onMouse3dOut);
}
```

In the `onMouse3dDown()`, `onMouse3dUp()`, and `onMouse3dOut()` handlers, we change the `_canExtrude` Boolean to true or false respectively in order to allow the extrusion while the mouse is pressed and dragged. The actual face extrusion happens inside the `onMouse3dMove()` method:

```
private function onMouse3dMove(e:MouseEvent3D):void{
    var face:Face=FaceVO(e.elementVO).face;
    if(_canExtrude&&face is Face){
        for(var i:int=0;i<face.vertices.length;++i){
            face.vertices[i].setValue(face.vertices[i].x+_
pushVector.x,face.vertices[i].y+_pushVector.y,face.vertices[i].z+_
pushVector.z);
        }
    }
}
```

As you can see in the preceding code block, we extract the current face reference under mouse click from `elementVO`, which we must cast as `Face` to get its properties. Then, using the `face.vertices[i].setValue()` method, we offset vertices position which cause the face they belong, to get extruded. The push values for x-, y-, and z-coordinates are received from `_pushVector`, which we switch by swapping extruding mode in this function which is called when one of the GUI buttons is clicked:

```
private function onMouseClick(e:MouseEvent):void{
    switch(e.currentTarget){
        case _but1:
            _pushVector=new Vector3D(0,20,0);
            break;
        case _but2:
            _pushVector=new Vector3D(0,-20,0);
            break;
    }
}
```

Now that's simple! Now you can start sculpting your Michelangelo's David.

See also

In Chapter 3, *Morphing objects* recipe.

Creating a controllable non-physical car

An interactively controllable car is a great feature that can be found today in many online flash applications such as casual games, featured sites (the most prominent may be helloenjoy.com of Carlos Ulloa) advertisements, and so on. What can have more impact on user experience than a cool Ferrari sports car racing right at your website front page while the player has complete control of its movement!

Interactive car setup in Away3D is pretty simple. The only time consuming part that can potentially complicate all the process is a car 3D model issues. The most common of those is local coordinate's mismatch of the car's parts to the Away3D coordinates system. There is no universal rule about how to avoid these problems but it rather depends on what 3D modeling software you use, as well as on the modeling routine itself. In the following example, we are going to set up a controllable car, whereas the model was downloaded from the royalty-free stock of www.3dtotal.com. You will see some of those model-related issues here and how to fix them within Away3D.

Getting ready

Set up a new Away3D scene using `AwayTemplate` and you are ready to go.

Make sure you include in your project the following assets. `PoliceCar.dae` and `TextureDPW.jpg`, which can be found in the *Chapter 4* assets folder in a subfolder named `vehicle`.

How to do it...

```
InterActiveCar.as

package
{
    public class InterActiveCar extends AwayTemplate
    {
        [Embed(source="../assets/vehicle/PoliceCar.
dae", mimeType="application/octet-stream")]
        private var CarModel:Class;
        [Embed(source="../assets/vehicle/TextureDPW.jpg")]
        private var CarTexture:Class;
```

Fun by Adding Interactivity —————

```
private var _loader:Loader3D;
private var _model:ObjectContainer3D;
private var _w1:ObjectContainer3D;
private var _w2:ObjectContainer3D;
private var _w3:ObjectContainer3D;
private var _w4:ObjectContainer3D;
private var _carBody:ObjectContainer3D;
private var _bitmat:BitmapMaterial;
private var _speed:Number=0;
private var _steer:Number=0;
private var _moveForward:Boolean=false;
private var _moveBackward:Boolean=false;
private var _moveRight:Boolean=false;
private var _moveLeft:Boolean=false;
private var _maxSpeed:Number=0;
private var _defaultWheelsRot:Number=90;
private var _springCam:SpringCam;
private var _numBuilds:int=5;
private var _angle:Number=0;
private var _radius:int=70;
private var _buildIter:int=1;
public function InterActiveCar()
{
    super();
    _view.clipping=new FrustumClipping();
    seedBuildings()

}

override protected function initListeners() : void{
    super.initListeners();
    stage.addEventListener(KeyboardEvent.KEY_
DOWN,onKeyDown,false,0,true);
    stage.addEventListener(KeyboardEvent.KEY_
UP,onKeyUp,false,0,true);
}
override protected function initMaterials() : void{
    _bitmat=new BitmapMaterial(Bitmap(new CarTexture()).bitmapData);
}
override protected function initGeometry() : void{
    setSpringCam();
    parseModel();
}
```

```

private function seedBuildings():void{
    for(var b:int=0;b<_numBuilds;++b){
        var h:Number=Math.floor(Math.random()*400+80);
        _angle=Math.PI*2/_numBuilds*b;
        var colMat:ColorMaterial=new ColorMaterial(Math.round(Math.
random()*0x565656));
        var build:Cube=new Cube({width:20,height:h,depth:20});
        build.cubeMaterials.back=colMat;
        build.cubeMaterials.bottom=colMat;
        build.cubeMaterials.front=colMat;
        build.cubeMaterials.left=colMat;
        build.cubeMaterials.right=colMat;
        build.cubeMaterials.top=colMat;
        build.movePivot(0,-build.height/2,0);
        build.x=Math.cos(_angle)*_radius*_buildIter*2;
        build.z=Math.sin(_angle)*_radius*_buildIter*2;
        build.y=0;
        _view.scene.addChild(build);
    }
    _buildIter++;
    if(_buildIter<5){
        _numBuilds*=_buildIter/2;
        seedBuildings();
    }
}
private function setSpringCam():void{
    _springCam=new SpringCam();
    _view.camera=_springCam;
    _springCam.stiffness=0.55;
    _springCam.damping=5;
    _springCam.mass=45;
    _springCam.zoom=25;
    _springCam.positionOffset=new Vector3D(0,45,170);
}
private function parseModel():void{
    var dae:Collada=new Collada();
    _model=dae.parseGeometry(new CarModel())as ObjectContainer3D;
    _view.scene.addChild(_model);
    _w1=_model.getChildByName("node-Cylinder01")as
ObjectContainer3D;//wheels:01,07,08,06///left back
    _w2=_model.getChildByName("node-Cylinder07")as
ObjectContainer3D;//wheels:01,07,08,06///right back
    _w3=_model.getChildByName("node-Cylinder08")as
ObjectContainer3D;//wheels:01,07,08,06///right front

```

Fun by Adding Interactivity

```
_w4=_model.getChildByName("node-Cylinder06")as
ObjectContainer3D;///wheels:01,07,08,06///left front
_carBody=_model.getChildByName("node-camaro01")as
ObjectContainer3D;

Mesh(_w1.children[0]).material=_bitmat;
Mesh(_w1.children[0]).scaleX=-1;
Mesh(_w2.children[0]).material=_bitmat;
Mesh(_w3.children[0]).material=_bitmat;
Mesh(_w4.children[0]).material=_bitmat;
Mesh(_w4.children[0]).scaleX=-1;
Mesh(_carBody.children[0]).material=_bitmat;
TweenMax.to(_carBody,0.2,{y:5,yoyo:true,repeat:9999,ease:Linear.
easeNone});
_model.scaleX=-1;
_springCam.target=_model;
}
private function onKeyDown(e:KeyboardEvent ):void
{
switch( e.charCode )
{
case 119: _moveForward=true;_moveBackward=false;break;/////
forward
case 115:_moveForward=false;_moveBackward=true;break; //////
backward
case 97: _moveRight=false;_moveLeft=true;break;////left
case 100:_moveRight=true;_moveLeft=false;break; // ////right
}
}
private function onKeyUp(e:KeyboardEvent ):void
{
switch( e.charCode )
{
case 119: _moveForward=false;break; // ////forward
case 115: _moveBackward=false;break; // ////backward
case 97: _moveLeft=false;break;////left
case 100: _moveRight=false;break; // ////right
}
}
private function updateCar():void{
// Speed
if( _moveForward ){_maxSpeed = 5;}
else if( _moveBackward ) { _maxSpeed = -5;}
else { _maxSpeed = 0;}
```

```
_speed -= ( _speed - _maxSpeed ) / 20;
if( _moveRight )
{
    if(_w3.rotationY >= 120 ){
        _steer=0;
    }else{
        _steer+=8;
    }
}
else if( _moveLeft )
{
    if( _w4.rotationY >=120 ){
        _steer=0;
    }else{
        _steer+=-8;
    }
}
else{
    _steer=0;
}
}
override protected function onEnterFrame(e:Event) : void{
    super.onEnterFrame(e);
    if(_model){
        _springCam.view;
        _w1.children[0].yaw(-_speed*2)
        _w2.children[0].yaw(_speed*2);
        _w3.children[0].yaw(_speed*2);
        _w4.children[0].yaw(-_speed*2);
        _w3.rotate(new Vector3D(0,1,0),_steer);
        _w4.rotate(new Vector3D(0,1,0),-_steer);
        var modelRot:Number=(Math.ceil(_w4.rotationY)-_
defaultWheelsRot)* Math.abs(_speed)/100;///*10 ;
        trace(_carBody.y);
        if(_moveForward){
            _model.yaw(modelRot );
        }else{
            _model.yaw(-modelRot );
        }
        _model.moveBackward( _speed );
        updateCar();
    }
}
```

Fun by Adding Interactivity

Now our car model is fully controllable! Use the WASD keyboard keys to drive it around the scene, similar to the one shown in the following image:



How it works...

First, in the `initListeners()` method, we set the `KeyBoardKEY_DOWN` and `KEY_UP` event listeners in order to control our car with the WASD keys:

```
stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown, false, 0, true);  
stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp, false, 0, true);
```

The next important step is a car model parsing that happens in the `parseModel()` function. As you can see, the `_model ObjectContainer3d` contains five nested `ObjectContainer3DS`, each of them encapsulating a mesh for car body and four wheels.

We also assign materials explicitly to each mesh object of the car because each separate part doesn't do it automatically. We assign the same baked map to all the parts and because each of them has got uv-mapping data from the 3D software it was modeled in, each wheel as well as the car body detects which part of the map is relevant to it:

```
Mesh(_w1.children[0]).material=_bitmat;  
Mesh(_w1.children[0]).scaleX=-1;  
Mesh(_w2.children[0]).material=_bitmat;  
Mesh(_w3.children[0]).material=_bitmat;  
Mesh(_w4.children[0]).material=_bitmat;  
Mesh(_w4.children[0]).scaleX=-1;  
Mesh(_carBody.children[0]).material=_bitmat;  
model.scaleX=-1;
```

Notice that we perform normal flips of `_w1` and `_w4` and of the `_model` itself. That step is unique to this specific model setup as there were flipped normal issues inherited from the modeling program.

In the `onKeyDown` and `onKeyUp` event handlers, we change Boolean flags indicating car moving and steering directions, which are used to define a car's movement and rotation modes inside the `updateCar()` method:

```
if( _moveForward ){_maxSpeed = 5;}  
else if( _moveBackward ) { _maxSpeed = -5; }  
else { _maxSpeed = 0; }  
  
_speed -= ( _speed - _maxSpeed ) / 20;  
if( _moveRight )  
{  
    if(_w3.rotationY >= 120 ){  
        _steer=0;  
    }else{  
        _steer+=8;  
    }  
}  
else if( _moveLeft )  
{  
    if( _w4.rotationY >=120 ){  
        _steer=0;  
    }else{  
        _steer+=-8;  
    }  
}else{  
    _steer=0;  
}  
}
```

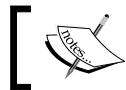
Inside the preceding block, we set up forward and backward speed for the car based on keyboards keys input. Then we add steer values for right and left steering of the front wheels. This specific models default steer (wheels face forward) is 90 degrees. We limit the rotation property of the wheels to 120 degrees so that when turning the car left or right will lock the wheels rotation to default +30 degrees to each side.

The last step is to update the actual car movement and steering based on the values from the `updateCar()` method. All that is accomplished in the `onEnterFrame()` function. We use the `yaw()` method and not roll for each wheel mesh because of local coordinates issues in this particular case. Also because of the same reason, we flip the yaw direction for the wheels `_w1` and `_w4`:

```
_w1.children[0].yaw(-_speed*2)  
_w2.children[0].yaw(_speed*2);  
_w3.children[0].yaw(_speed*2);
```

```
_w4.children[0].yaw(-_speed*2);  
  
_w3.rotate(new Vector3D(0,1,0),_steer);  
w4.rotate(new Vector3D(0,1,0),-_steer);
```

Front wheels `_w3` and `_w4` are steered (rotated around parent Y) by rotating their containers and not the meshes. Performing it directly on the mesh causes local objects position and rotational issues of the wheels.



Keep in mind that this entire setup is a particular model-specific. Some of the settings may slightly vary when using a different car model.



The last step in the `onEnterFrame()` function is to move and rotate the whole car. We do it in the following lines of code:

```
var modelRot:Number=(Math.ceil(_w4.rotationY)-_defaultWheelsRot)*  
Math.abs(_speed)/100;  
if(_moveForward){  
    _model.yaw(modelRot);  
}else{  
    _model.yaw(-modelRot);  
}  
_model.moveBackward(_speed);  
updateCar();  
}
```

We calculate the car rotation value for the `modelRot` variable in the following way. We find delta rotation from the default front wheels position and current steering angle, then multiplying it by the current car speed (absolute value because when the car moves backward its speed has a negative value while it is meaningless for the car steer) and dividing it by a factor of 100 to get a smaller rotation step. Next, we steer the car right or left according to its driving direction (backward/forward). The car is moved by using `moveBackward(_speed)`. You can also use `moveForward()` method just don't forget to negate `_speed` variable in this case. In the last line, we call the `updateCar()` method to update speed and steer values.

That is all.

After this, creating your own car-racing game should be a breeze!

See also

In Chapter 10, *Integration with Open Source Libraries* the following recipe: *Creating a physical car with JigLib*.

5

Experiencing the Wonders of Special Effects

In this chapter, we will cover:

- ▶ Exploding geometry
- ▶ Creating advanced bitmap effects using filters
- ▶ Creating clouds
- ▶ Visualizing sound in 3D
- ▶ Creating lens flare effects
- ▶ Masking 3D objects

Introduction

Special effects are something that Hollywood is all about. Talking in relation to Flash content, it is hardly a secret that the highest rated stuff created with it is almost always the result of some cool effects involved, which leaves users for a few seconds with their jaws dropped. Today when **Rich Internet Application (RIA)** rules the virtual world, it is hard to knock someone off his feet by delivering neat design or by animations going all around the screen only. Special effects are an essential part of flash applications, especially when it concerns 3D. A third dimension adds value to the effect and the impact that you impose on the user. One may suggest that 3D is, by itself, an effect when we talk about Flash environment. That was true a couple of years ago when it was tricky to get something to look three dimensional inside your application. But today, when 3D content is something people get used to, your challenge as a developer is to find new ways to surprise them and that is where special effects come to your rescue.

In this chapter, we will walk through recipes which would not necessarily turn your stuff into a masterpiece such as "The Matrix", but you will learn a few useful techniques and approaches for special effects creation with Away3D library. These techniques and approaches will save you a lot of time of know-how and point you in the right direction towards really advanced stuff.

Exploding geometry

It may be useful in certain scenarios to know how to blow your objects into pieces, especially when you work on a shooter game. We can blow up the geometry in Away3D in a relatively easy way using a utility class `Explode`, which is located in the `away3d.tools` package. So let's go through a quick demolition course.

Getting ready

As usual, set up a basic scene using the `AwayTemplate` class and you're ready to go.

How to do it...

In the following program, we create a regular sphere primitive. Then we trigger its explosion to pieces that are mesh triangles:

`SimpleExplode.as`

```
package
{
    public class SimpleExplode extends AwayTemplate
    {
        private var _explodeContainer:ObjectContainer3D;
        private var _sphere:Sphere;
        private var _light:PointLight3D;
        private var _colShade:ShadingColorMaterial;
        private var _canReset:Boolean=false;
        public function SimpleExplode()
        {
            super();
            _cam.z=-1000;
            initLight();
            initGUI();
        }
        override protected function initGeometry() : void{
            setGeometry()
        }
    }
}
```

```

override protected function initMaterials():void{
    _colShade=new ShadingColorMaterial(0xFD2D2D);
    _colShade.shininess=3;
    _colShade.specular=2.4;
    _colShade.ambient=0x127623;
}
override protected function initListeners():void{
    super.initListeners();
    _view.addEventListener(MouseEvent3D.MOUSE_DOWN,explodeAll);
}
private function initLight():void{
    _light=new PointLight3D({color:0xFD2D2D, ambient:0.25,
diffuse:0.13, specular:6});
    _light.brightness=4;
    _view.scene.addLight(_light);
    _light.position=new Vector3D(150,150,0);
}
private function setGeometry():void{
    _sphere=new Sphere({radius:50,material:_colShade,bothsides:true,
ownCanvas:true,segmentsH:16,segmentsW:16});
    _view.scene.addChild(_sphere);
}
private function explodeAll(e:MouseEvent3D):void{
    _view.scene.removeLight(_light);
    var explode:Explode=new Explode(true,true);
    _explodeContainer=explode.apply(_sphere) as ObjectContainer3D
    _view.scene.addChild(_explodeContainer);
    _view.scene.removeChild(_sphere);

    for each(var item:Mesh in _explodeContainer.children)
    {
        item.scaleX = 0.5;
        item.bothsides=true;
        var vector:Vector3D=item.position;
        TweenMax.to(item,25,{x:vector.x*(25+Math.random()*1000-
500),y:vector.y*(25+Math.random()*1000-500),z:vector.z*(25+Math.
random()*1000-500), ease:Linear.easeOut, onComplete:deleteTri,onComple
teParams:[item]});
        TweenMax.to(item,12,{rotationX:Math.random()*3000-
1500,rotationY:Math.random()*3000-1500,rotationZ:Math.
random()*3000-1500});
    }
    _canReset=true;
}
private function initGUI():void{
    var button:Button=new Button("reset");

```

```
        this.addChild(button);
        button.x=100;
        button.y=50;
        button.addEventListener(MouseEvent.CLICK,onButtonClick);
    }
    private function onButtonClick(e:MouseEvent):void{
        if(_canReset){
            resetGeometry();
            _canReset=false;
        }
    }
    private function resetGeometry():void{
        _view.scene.removeChild(_sphere);
        _sphere=null;
        for(var i:int=0;i<_explodeContainer.children.length;i++){
            _explodeContainer.removeChild(_explodeContainer.children[i]);
            _explodeContainer.children[i]=null;
        }
        _view.scene.removeChild(_explodeContainer);
        _explodeContainer=null;
        _view.scene.addLight(_light);
        setGeometry();
    }
    private function deleteTri(item:Mesh):void{
        if(_explodeContainer){
            _explodeContainer.removeChild(item);
        }
    }
}
```

Here you can see the debris fractures presented by mesh triangles dispersing all around after the explosion has been triggered:



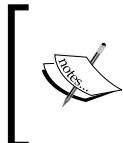
How it works...

First we set up a sphere primitive `_sphere` that we are going to explode later. What we need to do is to make all the triangles of the sphere faces detached from each other so that we can manipulate them freely in space. To do this, we use the `Explode` class which extracts each triangle to a separate mesh. When you click the sphere, the `explodeAll()` function is triggered. Then we first remove the scene lights for performance purposes because it is memory intensive to shade multiple flying triangles during runtime:

```
_view.scene.removeLight(_light);
```

Then we initiate an instance of `Explode` where we set two important parameters—first is `unicmeshes=true`, which means we want each triangle to be wrapped to its own mesh object and the other is `recenter=true`, which we need in order to have the new mesh centered in its local space:

```
var explode:Explode=new Explode(true,true);
```



Leaving the second parameter false will give us an interesting effect when all the triangles, after being exploded, keep moving inside the radius of their container. Try this out and maybe you will find it useful in certain scenarios.

Next we fill an empty `ObjectContainer3D` `_explodeContainer` with the brand new triangle meshes from the `_sphere` by calling the `explode.apply()` method which returns `ObjectContainer3D` if `unique meshes` is set to true, which is so in our case:

```
_explodeContainer=explode.apply(_sphere) as ObjectContainer3D
_view.scene.addChild(_explodeContainer);
_view.scene.removeChild(_sphere);
```

After breaking our `_sphere` to pieces, we can get rid of the original sphere as we no longer need it. Now comes the fun part where we are going to animate all the nested triangle meshes inside the `_explodeContainer`, thus creating an effect of exploding. The following lines are responsible for this:

```
for each(var item:Mesh in _explodeContainer.children)
{
    item.scaleX = 0.5;
    item.bothsides=true;
    var vector:Vector3D=item.position;
    TweenMax.to(item,25,{x:vector.x*(25+Math.random()*1000-
500),y:vector.y*(25+Math.random()*1000-500),z:vector.z*(25+Math.
random()*1000-500), ease:Linear.easeOut, onComplete:deleteTri,onComple
teParams:[item]});
```

```
TweenMax.to(item,12,{rotationX:Math.random()*3000-
1500,rotationY:Math.random()*3000-1500,rotationZ:Math.
random()*3000-1500});
}
```

As you can see from the preceding snippet, we iterate through all the children of the `_explodeContainer` and assign a random direction vector tween powered by a `TweenMax`. Another tween is responsible for a random rotation of each triangle mesh while they are traveling away.

There's more...

Here, our goal is basically the same as in "Exploding geometry (Simple way)". But here we will take it to the next level, and instead of just dividing the whole mesh into triangles, we will separate it into random chunks consisting of several triangles or faces grouped together. This way, the explosion debris will look more realistic. Let's see how to do it.

Getting ready

Set up a basic scene using the `AwayTemplate` class and you are ready to go.

How to do it...

Use the following code for `RandomChunksExplode.as`:

```
package
{
    public class RandomChunksExplode extends AwayTemplate
    {
        private var _explodeContainer:ObjectContainer3D;
        private var _sphere:Sphere;
        private var _light:PointLight3D;
        private var _colShade:ShadingColorMaterial;
        private var _canReset:Boolean=false;
        private var _mergedContainer:ObjectContainer3D;
        private var _merger:Merge=new Merge();
        public function RandomChunksExplode()
        {
            super();
            _cam.z=-1500;
            initLight();
            initGUI();
        }
    }
}
```

```

override protected function initGeometry() : void{
    setGeometry()
}
override protected function initMaterials():void{
    _colShade=new ShadingColorMaterial(0xFD2D2D);
    _colShade.shininess=3
    _colShade.specular=2.4;
    _colShade.ambient=0x127623;
}
override protected function initListeners():void{
    super.initListeners();
_view.addEventListener(MouseEvent3D.MOUSE_DOWN,explodeAll);
}
private function initLight():void{
    _light=new PointLight3D({color:0xFD2D2D, ambient:0.25,
diffuse:0.13, specular:6});
    _light.brightness=4;
    _view.scene.addLight(_light);
    _light.position=new Vector3D(150,150,0);
}
private function setGeometry():void{
    _sphere=new Sphere({radius:70,material:_colShade,bothsides:true,
ownCanvas:true,segmentsH:16,segmentsW:16});
    _view.scene.addChild(_sphere);
}
private function explodeAll(e:MouseEvent3D):void{
    _view.scene.removeLight(_light);
    var explode:Explode=new Explode(true,true);
    _explodeContainer=explode.apply(_sphere) as ObjectContainer3D
    _view.scene.removeChild(_sphere);
    _mergedContainer=new ObjectContainer3D();
    generateRandomExpl(_explodeContainer,_mergedContainer,6);
    _view.scene.addChild(_mergedContainer);
    for each(var item:Mesh in _mergedContainer.children)
    {
        item.centerPivot();
        var vector:Vector3D=item.scenePosition;
        TweenMax.to(item,25,{x:vector.x*(25+Math.random())*1000-
500,y:vector.y*(25+Math.random())*1000-500,z:vector.z*(25+Math.
random())*1000-500}, ease:Linear.easeOut, onComplete:deleteTri,onComple
teParams:[item]);
        TweenMax.to(item,12,{rotationX:Math.random()*3000-
1500,rotationY:Math.random()*3000-1500,rotationZ:Math.
random()*3000-1500});
    }
}

```

```
        }
        _canReset=true;
    }
    private function generateRandomExpl(sourceObj:ObjectContainer3D,targetObj:ObjectContainer3D,breakFactor:uint=15):void{
        var resObj:Mesh=new Mesh();
        _merger.unicgeometry=false;
        _merger.objectspace=false;
        var chLength:Number=sourceObj.children.length;
        var randNum:Number=Math.floor(Math.random()*breakFactor);
        for(var i:int=0;i<randNum;++i){
            if(chLength>0){
                _merger.apply(resObj,sourceObj.children[0]as Mesh); //r1
                sourceObj.children.splice(0,1); //r1
                chLength=sourceObj.children.length;
            }
        }
        resObj.bothsides=true;
        for each (var f:Face in resObj.faces){
            f.material=_colShade;
        }
        targetObj.addChild(resObj);
        if(chLength>0){
            generateRandomExpl(sourceObj,targetObj,breakFactor);
        }else{
            return ;
        }
    }
    private function initGUI():void{
        var button:Button=new Button("reset");
        this.addChild(button);
        button.x=100;
        button.y=50;
        button.addEventListener(MouseEvent.CLICK,onButtonClick);
    }
    private function onButtonClick(e:MouseEvent):void{
        if(_canReset){
            resetGeometry();
            _canReset=false;
        }
    }
}
```

```
private function resetGeometry():void{
    _view.scene.removeChild(_sphere);
    _sphere=null;
    for(var i:int=0;i<_mergedContainer.children.length;i++){
        _mergedContainer.removeChild(_mergedContainer.children[i]);
        _mergedContainer.children[i]=null;
    }
    _view.scene.removeChild(_mergedContainer);
    _mergedContainer=null;
    _explodeContainer=null;
    _view.scene.addLight(_light);
    setGeometry();
}
private function deleteTri(item:Mesh):void{
    if(_mergedContainer){
        _mergedContainer.removeChild(item);
    }
}
```

In the following image, you can see more realistic-looking debris than in the previous program, as they now consist of varying triangle groups:



How it works...

Unlike the simple explosion program here, the whole process is divided into two stages. First we isolate each triangle of the sphere primitive we want to explode to a unique mesh. We then run through them attaching them randomly into small mesh groups while still keeping the overall spherical formation.

Let's run through the code to see how it is done. All the fun begins when we click the sphere, the `explodeAll()` method is triggered. Inside it we initiate the `Explode` object passing our `_sphere` into the `explode.apply()` method that returns an `ObjectContainer3D` holding all the `_sphere` triangle meshes. After this, we remove the original sphere as we don't need it anymore:

```
var explode:Explode=new Explode(true,true);
_explodeContainer=explode.apply(_sphere) as ObjectContainer3D;
_view.scene.removeChild(_sphere);
```

Now we set another `ObjectContainer3D` container called `_mergedContainer`, which is going to accept the randomly regrouped triangle meshes from the `_explodeContainer`. `generateRandomExpl()` method takes care of that process for us as you will immediately see:

```
_mergedContainer=new ObjectContainer3D();
generateRandomExpl(_explodeContainer,_mergedContainer,6);
```

Before we proceed with the rest of the flow in this method, let's take a look at the `generateRandomExpl()` method and understand how it works. The function accepts the parameters which are, by order, `ObjectContainer3D` with triangle meshes, empty target `ObjectContainer3D` for the generated chunks, and `breakFactor` number which defines the upper limit for the random number of the triangles in a single chunk. Inside the function, we use a recursive approach. At the function call, we initiate a merge instance called `_merger` and with its help, merge triangle meshes from the source container to a single mesh inside a `for` loop statement, which runs a random number of times during each function call:

```
var resObj:Mesh=new Mesh();
_merger.unicgeometry=false;
_merger.objectspace=false;
var chLength:Number=sourceObj.children.length;
var randNum:Number=Math.floor(Math.random()*breakFactor);
for(var i:int=0;i<randNum;++i){
    if(chLength>0){ _merger.apply(resObj,sourceObj.
    children[0]);
        sourceObj.children.splice(0,1);
        chLength=sourceObj.children.length;
    }
}
```

Each time we run the `for` loop, we check whether the number of triangle meshes in the source objects is more than zero as we splice each one after it has been replicated into the target container. After the loop finishes, the next step is to apply the initial material back to each chunk. Otherwise, you will see random color mapping of each face:

```
resObj.bothsides=true;
for each (var f:Face in resObj.faces) {
    f.material=_colShade;
}
```

The final step in this method is recursive call of the function in case there are still triangle meshes left on the `sourceObj` container, which still have not been taken care of in the `for` loop statement previously. If this is the case, the `generateRandomExpl()` function calls itself, otherwise the statement is exited with a `return` operator and the function completes executing:

```
if(chLength>0) {
    generateRandomExpl(sourceObj,targetObj,breakFactor);
} else{
    return ;
}
```

Now let's go back to the `explodeAll()` method and see what we do after the `generateRandomExpl()` method execution.

Here we run a `for each` loop where we iterate through each chunk mesh and tween it with `TweenMax` in a random direction, thus simulating an explosion velocity of the chunks. Additionally, we rotate the chunks during their translation using the same approach:

```
for each(var item:Mesh in _mergedContainer.children)
{
    item.centerPivot();
    var vector:Vector3D=item.scenePosition;
    TweenMax.to(item,25,{x:vector.x*(25+Math.random()*1000-
500),y:vector.y*(25+Math.random()*1000-500),z:vector.z*(25+Math.
random()*1000-500), ease:Linear.easeOut, onComplete:deleteTri,onComple
teParams:[item]});
    TweenMax.to(item,12,{rotationX:Math.random()*3000-
1500,rotationY:Math.random()*3000-1500,rotationZ:Math.
random()*3000-1500});
}
    _canReset=true;
}
```

Now you have an idea how to code some more advanced explosion effects with Away3D.

See also

In Chapter 10, *Integration with Open Source Libraries*, the following recipe, *Exploding particles with FLINT*.

Creating advanced bitmap effects using filters

Bitmap processing based on Flash generic filters in correlation with `BitmapData` input is widely used to create quite stunning visual effects in 2D as well as in 3D. Unfortunately, at the time of this writing, Away3D has no built-in functionality for bitmap effects like its rival PV3D has (`BitmapEffectLayer`), nevertheless we can easily overcome this by writing those effects from scratch by ourselves. Let's see how to do it.

Getting ready

Just set up a basic scene using the `AwayTemplate` class and you are ready to begin.

How to do it...

In the following example, we generate a bunch of spheres which move randomly across the 3D space by means of implementing the "Brownian Motion" algorithm. During runtime, we apply to them a set of filters while drawing the whole application to `BitmapData`. The result is pretty cool.

Use the following code for `ColorEffectsDemo.as`:

```
package
{
    public class ColorEffectsDemo extends AwayTemplate
    {
        private var _sphereArr:Array=[];
        private var _bitmapContainer:Bitmap;
        private var _renderBData:BitmapData;
        private var _blurF:BlurFilter;
        private var _colMatrixF:ColorMatrixFilter;
        private var _colMat:ColorMaterial;
        private var _displF:DisplacementMapFilter;
        private var _hoverCam:HoverCamera3D;
        private var _camDummy:ObjectContainer3D;
        private var _oldMouseX:Number=0;
        private var _oldMouseY:Number=0;
        private static const EASE_FACTOR:Number=0.5;
```

```
public function ColorEffectsDemo()
{
    super();

    _cam.z=-400;
    setHowerCamera();
    initFilters();
    _renderBData=new BitmapData(stage.stageWidth,stage.stageHeight);
    _bitmapContainer=new Bitmap(_renderBData);
    this.addChild(_bitmapContainer);
}
private function initFilters():void{
    _blurF=new BlurFilter(3,3,2);
    var colArr:Array=[0.989,0,2,0,38,
                      0,0.827,0,0,38,
                      0,0,0.876,0,38,
                      0,0,0,1.1,0
                     ];
    _colMatrixF=new ColorMatrixFilter(colArr);
    var dispBmd:BitmapData=new BitmapData(stage.stageWidth,stage.
stageHeight);
    dispBmd.perlinNoise(25,25,12,34543,true,true,7,true);
    _displF=new DisplacementMapFilter(dispBmd,null,BitmapDataChann
el.RED,BitmapDataChannel.BLUE,12,12,DisplacementMapFilterMode.WRAP);

}
private function applyFilters():void{
    _renderBData.lock();
    _renderBData.draw(this);
    _renderBData.applyFilter(_renderBData,_renderBData.rect,new
Point(0,0),_colMatrixF);
    _renderBData.applyFilter(_renderBData,_renderBData.rect,new
Point(0,0),_blurF);
    _renderBData.applyFilter(_renderBData,_renderBData.rect,new
Point(0,0),_displF);
    _renderBData.unlock();
}
private function setHowerCamera():void{
    _hoverCam=new HoverCamera3D();
    _view.camera=_hoverCam;
    _hoverCam.target=_camDummy;
    _hoverCam.distance = 500;
    _hoverCam.maxTiltAngle = 80;
    _hoverCam.minTiltAngle = 0;
    _hoverCam.wrapPanAngle=true;
    _hoverCam.steps=16;
```

```
        _hoverCam.yfactor=1;
    }
    private function updateSceneObjectsPos():void{
        var arrLength:uint=_sphereArr.length;
        for(var i:int=0;i<arrLength;++i){
            _sphereArr[i].vx+=Math.random()*0.5-0.25;
            _sphereArr[i].vy+=Math.random()*0.5-0.25;
            _sphereArr[i].vz+=Math.random()*0.5-0.25;
            _sphereArr[i].x+=_sphereArr[i].vx;
            _sphereArr[i].y+=_sphereArr[i].vy;
            _sphereArr[i].z+=_sphereArr[i].vz;
            if(_sphereArr[i].x>400){
                _sphereArr[i].x=-400;
            }else if(_sphereArr[i].x<-400){
                _sphereArr[i].x=400;
            }

            if(_sphereArr[i].y>300){
                _sphereArr[i].y=-300;
            }else if(_sphereArr[i].y<-300){
                _sphereArr[i].y=300;
            }

            if(_sphereArr[i].z>300){
                _sphereArr[i].z=-300;
            }else if(_sphereArr[i].z<-300){
                _sphereArr[i].z=300;
            }
        }
    }
    override protected function initMaterials() : void{
        _colMat=new ColorMaterial(0x229933);
    }
    override protected function initGeometry() : void{
        for(var i:int=0;i<24;++i){
            var sphere:FSphere=new FSphere({radius:20,material:new ColorMaterial(Math.floor(Math.random()*0xFFFFFF))});
            sphere.ownCanvas=true;
            sphere.segmentsH=5;
            sphere.segmentsW=5;
            _sphereArr.push(sphere);
            sphere.x=Math.random()*800-400;
            sphere.y=Math.random()*600-300;
            sphere.z=Math.random()*600-300;
        }
    }
}
```

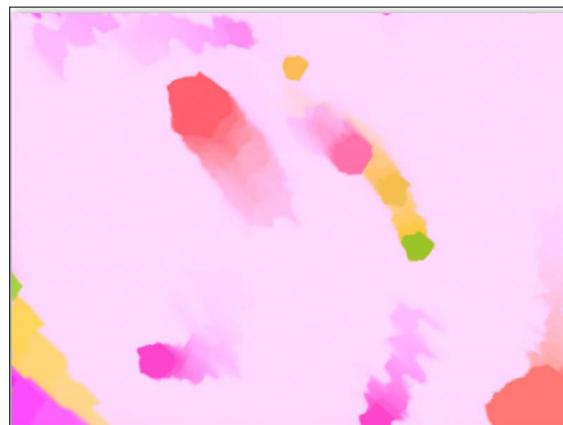
```
        _view.scene.addChild(sphere);
    }
    _camDummy=new ObjectContainer3D();
    _camDummy.position=new Vector3D(0,0,100);

}
override protected function onEnterFrame(e:Event) : void{
    super.onEnterFrame(e);
    applyFilters();
    updateSceneObjectsPos();
    if(_hoverCam){

        _hoverCam.panAngle = (stage.mouseX - _oldMouseX)*EASE_FACTOR
    ;
        _hoverCam.tiltAngle = (stage.mouseY - _oldMouseY)*EASE_
FACTOR ;
        _hoverCam.hover();
    }
}
import away3d.primitives.Sphere;

class FSphere extends Sphere{
    public var vx:Number=0;
    public var vy:Number=0;
    public var vz:Number=0;
    public function FSphere(init:Object=null){
        super(init);
    }
}
```

In this image, you can see the result: 3D spheres get a blurry 2D look with a nice dissolving trail following each of them:



How it works...

First, inside the constructor, we initiate three generic flash filters by calling the `initFilters()` method. The filters we are going to use are `BlurFilter`, `ColorMatrixFilter`, and `DisplacementMapFilter`. The purpose of the `BlurFilter` is obvious. The `ColorMatrixFilter` is responsible for pixel's color transformation job and `DisplacementMapFilter` serves us to add a dissolution effect that you can see applied to the moving spheres and to their trails.

Next, we instantiate a `BitmapData` object named `_renderBData` to which we eventually draw all the Away3D scenes and then pass it to a `Bitmap` variable called `_bitmapContainer`:

```
_renderBData=new BitmapData(stage.stageWidth,stage.stageHeight);
_bitmapContainer=new Bitmap(_renderBData);
this.addChild(_bitmapContainer);
```

Now let's focus for a second on scene geometry itself. As you can see, at the bottom of the code file, I put another class that extends the sphere primitive. It is not the best practice to write more than one class in the same file, but here it is for the sake of saving space. The only difference the FSphere has are `vx`, `vy`, and `vz` public variables that we use to set an accelerated motion of the spheres, which is executed with the help of the Brownian Motion Algorithm inside the `updateSceneObjectsPos()` method.

We call the `updateSceneObjectsPos()` function inside `onEnterFrame()` and at the same time we apply our filters to the `_renderBData` constantly calling the `applyFilters()` function:

```
private function applyFilters():void{
    _renderBData.draw(this);
    _renderBData.applyFilter(_renderBData,_renderBData.rect,new
Point(0,0),_colMatrixF);
    _renderBData.applyFilter(_renderBData,_renderBData.rect,new
Point(0,0),_blurF);
    _renderBData.applyFilter(_renderBData,_renderBData.rect,new
Point(0,0),_displF);
}
```

As you can see, we draw the whole sprite (as `AwayTemplate` extends it) containing the `Away3DView3D` to the `_renderBData` `BitmapData`, updating this way the whole appearance.

See also

You can create much cooler effects and with less coding using open source frameworks. One of those is "HYPE" created by the renowned Flash artist, Joshua Davis. Using **HYPE**, you can set stunning effects to your projects with minimal efforts. You can get it here: www.hypeframework.org.

Creating clouds

In this recipe, you are going to learn how to generate quite realistic clouds in Away3D. In fact, we will create fake 3D clouds which, in essence, are procedural perlin noise rendered to textures of several plane primitives, positioned in front of each other at some distance. The result is pretty convincing and not too CPU intensive.

Getting ready

Set up a basic Away3D scene extending `AwayTemplate` and you are good to go.

How to do it...

Use the following code for `PerlinClouds.as`:

```
package
{
    public class PerlinClouds extends AwayTemplate
    {
        private var _planes:Array=[];
        private var _numberOfPlanes:int=6;
        private var _perlin3D:Array=[];
        private var _plane:Plane;
        private var _planeSize:int=700;
        private var _offsets:Array;
        private var _colTransform:ColorTransform=new
ColorTransform(1,1,4,0.6,8,21,63,-48);
        private var _glowF:GlowFilter=new GlowFilter(0x156EC6,1,3,3,2,2,f
alse,false);
        private var _mainContainer:ObjectContainer3D;
        public function PerlinClouds()
        {
            super();
        }
        override protected function initGeometry() : void{
            _mainContainer=new ObjectContainer3D();
            _view.scene.addChild(_mainContainer);
            _mainContainer.z=4200;
            initCloudPlanes();
            _cam.lookAt(_mainContainer.position,Vector3D.Y_AXIS);
```

```
        }
    override protected function onEnterFrame(e:Event) : void{
        super.onEnterFrame(e);
        redraw();
    }
    private function initCloudPlanes():void{
        _offsets = [new Point(0, 0), new Point(0, 0), new Point(0, 0)];
        var scaleFactor:int=1;
        while (_numberOfPlanes--)
        {
            scaleFactor++;
            _plane = new Plane({material:null,width: _planeSize
*scaleFactor * 3, height:_planeSize * scaleFactor * 3});
            _plane.rotationX=90;
            _mainContainer.addChild(_plane);
            _plane.z = _numberOfPlanes * _planeSize/3;
            _planes.push(_plane);
        }
    }
    private function draw3DPerlin():void{
        var textureBdata:BitmapData;
        _perlin3D.pop();
        while (_perlin3D.length < 6)
        {
            textureBdata = new BitmapData(64, 64, true, 0);///
            textureBdata.perlinNoise(16,13,17,123456, true, true, 15,
false, _offsets);
            textureBdata.colorTransform(textureBdata.rect, _colTransform);
            textureBdata.applyFilter(textureBdata,textureBdata.rect,new
Point(0,0),_glowF);
            _offsets[0].x = _offsets[0].x + 1 ;
            _offsets[0].y = _offsets[0].y + 1 ;
            _offsets[2].x = _offsets[2].x - 1 ;
            _offsets[2].y = _offsets[2].y + 0.5 ;
            _offsets[1].x = _offsets[1].x - 1 ;
            _offsets[1].y = _offsets[1].y - 2.7;
            _perlin3D.splice(0, 0, textureBdata);
        }
    }
    private function redraw():void{
        var counter:int=0;
        draw3DPerlin();
        while (counter < _planes.length)
        {
```

```
var bitMat:BitmapMaterial= new BitmapMaterial(_  
perlin3D[counter]);  
bitMat.smooth=true;  
_planes[counter].material=bitMat;  
++counter;  
}  
}  
}  
}
```

How it works...

We begin with creating a container named `_mainContainer` inside the `initGeometry()` method, which would hold all the plane primitives as its children. Then we set up the planes on which we are going to render the clouds calling the `initCloudPlanes()` function.

Inside the `initCloudPlanes()` function, we initiate the `_offsets` array which holds three `Point` objects. This array is important as it is going to serve us as input for the `offset` parameter of the `perlinNoise()` method, which we use to generate animated perlin noise effect:

```
_offsets = [new Point(0, 0), new Point(0, 0), new Point(0, 0)];
```

Then we set up six planes with the following lines:

```
var scaleFactor:int=1;  
while (_numberOfPlanes--)  
{  
    scaleFactor++;  
    _plane = new Plane({material:null,width: _planeSize  
*scaleFactor * 3, height:_planeSize * scaleFactor * 3});  
  
    _plane.rotationX=90;  
    _mainContainer.addChild(_plane);  
    _plane.z = _numberOfPlanes * _planeSize/3;  
    _planes.push(_plane);  
}
```

In the preceding code block, we scale each successful plane as it is located further from the camera. We have to do that in order to hide the edges of more distant planes as they begin to appear smaller because of their perspective.

Now let's see where the actual fun happens. Inside `draw3DPerlin()` function we create a `textureBdata` object to which we then draw a perlin noise:

```
var textureBdata:BitmapData;
```

Another important step that we make is to drop the last member of `_perlin3D` array that always stores six instances of `textureBdata`:

```
_perlin3D.pop();
```

We do it because we need to progressively update each plane texture with a newly generated perlin `BitmapData`. When each perlin bitmap from the array has gone through each of the six planes, we pop it out from the array and instead insert a new one in the beginning. Because we only pop out one element before each iteration (except the first one where we fill the array with six elements), the `while` loop will run only once each successive time. Now comes the block that we execute inside the `while` statement:

```
textureBdata = new BitmapData(64, 64, true, 0);
    textureBdata.perlinNoise(16,12,17,12345, true, true, 15,
false, _offsets);
    textureBdata.colorTransform(textureBdata.rect, _
colTransform);
    textureBdata.applyFilter(textureBdata,textureBdata.rect,new
Point(0,0),_glowF);
    _offsets[0].x = _offsets[0].x + 1 ;
    _offsets[0].y = _offsets[0].y + 1 ;
    _offsets[2].x = _offsets[2].x - 1 ;
    _offsets[2].y = _offsets[2].y + 0.5 ;
    _offsets[1].x = _offsets[1].x - 1 ;
    _offsets[1].y = _offsets[1].y - 2.7;
    _perlin3D.splice(0, 0, textureBdata);
}
```

In the preceding block we draw the perlin noise to the `BitmapData`, then apply a color transform in order to set a desired color for the clouds. Then we apply a `GlowFilter` which we instantiated previously. I will not cover in detail about the `perlinNoise()` parameters as well as filters setting as it depends greatly on the desired result as well as on playing around with different settings. Now back to business. Next we updated the offset points assigning incremented values to the points inside the `_offsets` array. The last thing in this method, as you can see, is `_perlin3D.splice(0, 0, textureBdata)`; which basically inserts a new `textureBdata` object at the beginning of the array.

The last function is `redraw()`, which is called in the `onEnterFrame()` method. Inside it, we first call the `draw3DPerlin()`; in order to update the `_perlin3D` array with a new `BitmapData`. Then we run through each plane and assign to it an updated material:

```
private function redraw():void{
    var counter:int=0;
    draw3DPerlin();
    while (counter < _planes.length)
{
```

```
var bitMat:BitmapMaterial= new BitmapMaterial(_  
perlin3D[counter]);  
bitMat.smooth=true;  
_planes[counter].material=bitMat;  
++counter;  
  
}
```

As the material update happens from the furthest plane to the nearest, the resulting effect looks as if the observer flies through the clouds. Looks cool and quite 3D, doesn't it?

See also

If you are interested in a static clouds effect such as fog, you can use Away3D's built-in fog filter. You can find the class in the `away3d.core.filter` package.

Visualizing sound in 3D

If you decide to start developing a flash mp3 jukebox, you will be happy for sure to add some sound visualization effects. With the combination of Away3D and ActionScript 3.0 `SoundMixer` class, you can achieve some really cool effects. So let's have some fun.

Getting ready

Set up the Away3D scene using `AwayTemplate` as your base class.

Make sure you include in your assets folder `s3.mp3` and `dofText.png`, which can be found in this chapter's assets folder. Alternatively, you can use your own mp3 sound track or a bitmap file for sprite texture.

How to do it...

In the following program, we create a field of `Sprite2D` objects around a sphere which receives their transformation values from the `SoundMixer.computeSpectrum()` method:

`SoundVisDemo.as`

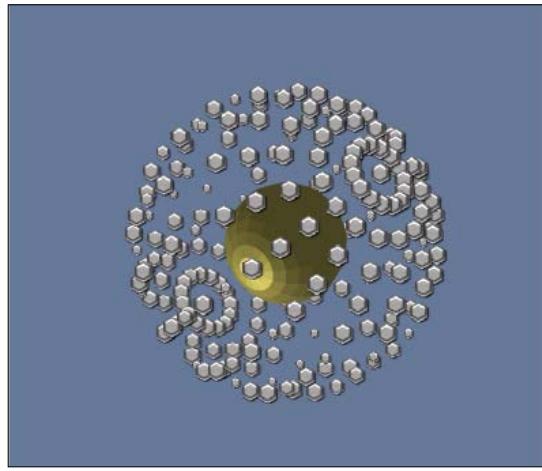
```
package  
{  
    public class SoundVisDemo extends AwayTemplate  
    {  
        [Embed(source="assets/s4.mp3")]  
        public var Sound1:Class;  
        [Embed(source="assets/dofText.png")]
```

```
private var SpriteTexture:Class;
private var _sound:Sound;
private var _barray:ByteArray=new ByteArray();
private const CHANNEL_LENGTH:int = 256;
private var _sphere:Sphere;
private var _mat:ShadingColorMaterial;
private var _light:DirectionalLight3D;
private var _colMat:ColorMaterial;
private var _spritesArr:Array=[];
private var _container:ObjectContainer3D;
private var _timer:Timer;
private var _spectrumArr:Vector.<Number>=new Vector.<Number>();
private var _spriteMat:BitmapMaterial;
public function SoundVisDemo()
{
    super();
    _cam.z=-300;
    initLight();
    initSprites();
    initSound();
    _timer=new Timer(20,0);
    _timer.addEventListener(TimerEvent.TIMER,processSound);
    _timer.start();
}
override protected function initGeometry(): void{
    _sphere=new Sphere({radius:20,segmentsW:15,segmentsH:15,material:_mat});
    _view.scene.addChild(_sphere);
    _container=new ObjectContainer3D();
    _view.scene.addChild(_container);
}
override protected function initMaterials(): void{
    _mat=new ShadingColorMaterial(0x928843);
    _colMat=new ColorMaterial(0x229933);
    _spriteMat=new BitmapMaterial(Cast.bitmap(SpriteTexture));
}
override protected function onEnterFrame(e:Event): void{
    super.onEnterFrame(e);
    _sphere.rotate(new Vector3D(1,1,0),1.3);
    var matr:Matrix3D;
    matr=_sphere.transform.clone();
    matr.invert();
    _container.transform=matr;
}
```

```
private function initSprites():void{
    var leng:uint=_sphere.vertices.length;
    for(var i:int=0;i<leng;++i){
        var sprite:Sprite3D=new Sprite3D(_spriteMat,10,10,0,"center",0.1);
        _container.addSprite(sprite);
        var vertex:Vertex=_sphere.vertices[i];
        var posVec:Vector3D;
        posVec=vertex.position.clone();
        posVec.scaleBy(2);
        sprite.x=posVec.x;
        sprite.y=posVec.y;
        sprite.z=posVec.z;
        _spritesArr.push({spr:sprite,pos:posVec});
    }
}
private function initLight():void{
    _light=new DirectionalLight3D();
    _view.scene.addLight(_light);
    _light.direction=new Vector3D(110,110,0);
}
private function initSound():void{
    _sound=new Sound1() as Sound;
    _sound.play(0,99999);
}
private function processSound(e:TimerEvent):void{
    _barray.position=0;
    SoundMixer.computeSpectrum(_barray,false,0);
    for (var i:int=0;i<CHANNEL_LENGTH-44;++i){
        _spectrumArr[i]=_barray.readFloat();
        var sprite:Sprite3D=_spritesArr[i].spr;
        sprite.x=_spritesArr[i].pos.x;
        sprite.y=_spritesArr[i].pos.y;
        sprite.z=_spritesArr[i].pos.z;
        var soundData:Number=_spectrumArr[i];
        var posVec:Vector3D=new Vector3D(sprite.x,sprite.y,sprite.z);
        posVec=posVec.add(new Vector3D(0,Math.abs(soundData)*15,0));
        sprite.x=posVec.x;
        sprite.y=posVec.y;
        sprite.z=posVec.z;
        if(Math.abs(soundData)>0.21){
            sprite.scaling=0.21;
        }else if(Math.abs(soundData)<=0.05){
            sprite.scaling=0.1
        }
    }
}
```

```
        }else{
            sprite.scaling=Math.abs(soundData);
        }
    }
    _sphere.scale(Math.sin(Math.abs(soundData)+2));
}
}
```

Following is our sound visualizer in action:



As a matter of fact, only your imagination and math knowledge sets the limit here. You can really set mind-blowing transformations by casting different algorithms on the spectrum data.

How it works...

Let's run through the most important functions of this demo. First we create a typical sphere primitive inside the `initGeometry()` method. It serves as a nucleus of our sprites layer as well as position coordinates source for the sprite objects. Immediately after `_sphere` initialization, we also create `ObjectContainer3D` named `_container`, which is going to hold all the sprites. This way, we can rotate them all as a group, keeping their initial formation.

Next, we create the sprites calling the `initSprites()` method. With the following lines of code, we iterate with the `for` through `_sphere` vertices, creating a `Sprite3D` instances then adding those to `_container`:

```
for(var i:int=0;i<leng;++i){
    var sprite:Sprite3D=new Sprite3D(_spriteMat,10,10,0,"center",0.1);
    _container.addSprite(sprite);
```

Next we want to position each of them at some distance from the relevant vertex position. To do this, we scale each `Vertex` vector in our case by two, then we assign it to the `x`, `y`, and `z` properties of each sprite:

```
var vertex:Vertex=_sphere.vertices[i];
var posVec:Vector3D;
posVec=vertex.position.clone();
posVec.scaleBy(2);
sprite.x=posVec.x;
sprite.y=posVec.y;
sprite.z=posVec.z;
```

The last thing we do inside this function is store each sprite and its initial position in an array object called `_spriteArr`. We need it for two purposes—iterate through each sprite inside the array in the runtime to update the position based on sound spectrum output, and, after each iteration, to reset each sprite to its initial position which we store in the `pos` object inside the `_spriteArr`. This way, after every oscillation caused by the spectrum data, the sprites reset their positions once again; otherwise they would disperse like particles all over the scene:

```
_spritesArr.push({spr: sprite, pos: posVec});
```

In the next step, we call the `initSound()` function which starts playing the sound. The last thing we initiate in the constructor is the timer object `_timer` which, on each count, calls the `processSound()` function which we will deal with shortly. The reason to use a timer to update the sprites and on `onEnterFrame()` is that the timer allows us to define more frequent callbacks than `onEnterFrame()`.

Now let's see how we actually visualize the sound. Inside the `processSound()` method, we first extract the sound spectrum values which range from one to one (for a deeper explanation on SoundMixer, see ActionScript3.0 Adobe Documentation) to `_barray` `ByteArray`. The returned `ByteArray` actually has a length of 512 floating-point values—256 for left and 256 for right channels. For our example, we will use only the first 256. After we extracted the wave spectrum, we start looping through each byte stored in `_barray` and push it into the `_spectrumArr` array so that we can use those values later for the sprites position:

```
for (var i:int=0;i<CHANNEL_LENGTH-44;++i){
    _spectrumArr[i]=_barray.readFloat();
```

Next, we reset each sprite to its original position before oscillating it:

```
var sprite:Sprite3D=_spritesArr[i].spr;
sprite.x=_spritesArr[i].pos.x;
sprite.y=_spritesArr[i].pos.y;
sprite.z=_spritesArr[i].pos.z;
```

Then we get a value from the stored spectrum data bank array and use it to move the sprite. Inside the `_container` we give the position vector `posVec` some offset distance at y-axis by adding to the current position vector another vector that scales the y-axis by a value from the sound spectrum multiplied by 15:

```
var soundData:Number=_spectrumArr[i];
    var posVec:Vector3D=new Vector3D(sprite.x,sprite.y,sprite.z);
    posVec=posVec.add(new Vector3D(0,Math.abs(soundData)*15,0));
    sprite.x=posVec.x;
    sprite.y=posVec.y;
    sprite.z=posVec.z;
```

Now to enrich the effect, I also added sprites scaling based on the current spectrum number value range:

```
if(Math.abs(soundData)>0.21){
    sprite.scaling=0.21;
} else if(Math.abs(soundData)<=0.05) {
    sprite.scaling=0.1
} else{

    sprite.scaling=Math.abs(soundData);
}
}
```

And we add one more effect for some overall shaking to the inner sphere:

```
_sphere.scale(Math.sin(Math.abs(soundData)+2));
```

There's more...

If you were fascinated by the miracles that sound can do in computer graphics, you can further broaden your knowledge and get inspired by visiting pages of two renowned Flash gurus and sound processing pioneers Andre Michelle and Joa Ebert. You can visit the following sites to know more about them:

blog.andre-michelle.com

blog.joa-ebert.com

Creating lens flair effects

Lens flare effects may be useful in the scenes with outdoor environment when you have got a daylight skybox setup. This way, you can add more realism to such a scene in a way that when the user's camera turns in the direction of the light source, it would be affected by the lens flare effect coming from the same direction. In this recipe, you are going to learn how to set up a lens flare effect using Away3D's built in `LensFlair` tool.

Getting ready

Set up a basic Away3D scene extending `AwayTemplate`.

`Away3D` `LensFlair` class requires external graphics input for the flair (halo) rings. Make sure you connect to your project `graphicsLib.swc` file that contains a set of halo rings created initially using Adobe illustrator's `LensFlair` tool. If you wish to learn how to create customized lens flare rings, please refer to the Illustrator manual (CS4) on the following link: http://help.adobe.com/en_US/Illustrator/14.0/WS714a382cdf7d304e7e07d0100196cbc5f-6201a.html.

How to do it...

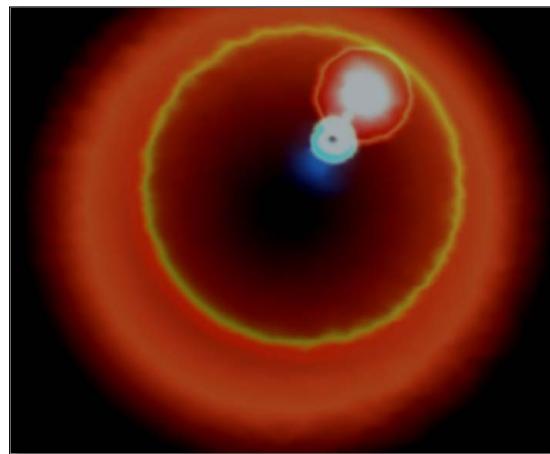
`LensFlairDemo.as`:

```
package
{
    public class LensFlairDemo extends AwayTemplate
    {
        private var _lensFlair:LensFlare;
        private var _source:Object3D;
        private var _oldMouseX:Number=0;
        private var _oldMouseY:Number=0;
        private static const EASE_FACTOR:Number=0.5;
        private var _hoverCam:HoverCamera3D;
        private var _camTarget:Object3D=new Object3D();
        private var _bitmapContainer:Bitmap;
        private var _renderBData:BitmapData;
        private var _overlay:Sprite;
        private var _blurF:BlurFilter;
        private var _colMatrixF:ColorMatrixFilter;
        private var _drawMatr:Matrix;
        private var _displF:DisplacementMapFilter;
        public function LensFlairDemo()
        {
            super();
        }
    }
}
```

```
initFilters();
setHowerCamera();
initLensFlair();
_overlay=_view.overlay;
_renderBData=new BitmapData(stage.stageWidth,stage.stageHeight);
_bitmapContainer=new Bitmap(_renderBData);
this.addChild(_bitmapContainer);
_overlay.visible=false;
}
override protected function initGeometry() : void{
_source=new Object3D();
_view.scene.addChild(_source);
_source.z=1500;
_source.y=200;
_source.x=200;
_view.scene.addChild(_camTarget);
_camTarget.transform.position=new Vector3D(0,0,0);
}
override protected function onEnterFrame(e:Event) : void{
super.onEnterFrame(e);
if(_hoverCam){
_hoverCam.panAngle = (stage.mouseX - _oldMouseX)*EASE_FACTOR ;
_hoverCam.tiltAngle = (stage.mouseY - _oldMouseY)*EASE_FACTOR
;
_hoverCam.hover();
}
applyFilters();
}
private function initLensFlair():void{
_lensFlair=new LensFlare(_source,_view.camera);
_lensFlair.haloScaleFactor=79;
_lensFlair.setHaloAsset(new HaloRing ());
_lensFlair.addFlareAsset(new FlairRing1());
_lensFlair.addFlareAsset(new FlairRing2());
_lensFlair.addFlareAsset(new FlairRing3());
_lensFlair.addFlareAsset(new FlairRing4());
_lensFlair.addFlareAsset(new FlairRing5());
_view.addOverlay(_lensFlair);
_lensFlair.blendMode=BlendMode.ADD;
}
private function initFilters():void{
.blurF=new BlurFilter(3,3,3);
var colArr:Array=[0.99,0,0,0,-32,
0,0.99,0,0,-21,
0,0,0.99,0,-17,
0,0,0,0.72,0
];
}
```

```
    _colMatrixF=new ColorMatrixFilter(colArr);
    var dispBmd:BitmapData=new BitmapData(stage.stageWidth,stage.
stageHeight);
    dispBmd.perlinNoise(36,23,12,34543,true,true,7,true);
    _displF=new DisplacementMapFilter(dispBmd,null,BitmapDataChann
el.BLUE,BitmapDataChannel.RED,4,4,DisplacementMapFilterMode.WRAP);
    _drawMatr=new Matrix(1,0,0,1,500,180);
}
private function applyFilters():void{
    _renderBData.draw(_overlay,_drawMatr);
    _renderBData.applyFilter(_renderBData,_renderBData.rect,new
Point(0,0),_blurF);
    _renderBData.applyFilter(_renderBData,_renderBData.rect,new
Point(0,0),_colMatrixF);
    _renderBData.applyFilter(_renderBData,_renderBData.rect,new
Point(0,0),_displF);
}
private function setHoverCamera():void{
    _hoverCam=new HoverCamera3D();
    _view.camera=_hoverCam;
    _hoverCam.target=_camTarget;
    _hoverCam.distance = 1800;
    _hoverCam.maxTiltAngle = 0;
    _hoverCam.minTiltAngle = 0;
}
}
```

The following image is what you should see after running the preceding code. Away3D
LensFlare overlay combined with Bitmap effects produces a nice glowing effect for each ring:



How it works...

Let's begin first from the `LensFlair` setup. We instantiate it inside the `initLensFlair()` function. To the constructor, we pass the source of the flair which is just empty (dummy) `Object3D _source` positioned inside the scene and the reference to the active camera:

```
_lensFlair=new LensFlare(_source,_view.camera);
```

`haloScaleFactor` is responsible for scaling the halo ring proportionally to the camera direction angle (gets max value when facing the camera perpendicularly), which is the most distant one from the rest and virtually represents the light source halo effect:

```
_lensFlair.haloScaleFactor=79;
```

Next we assign one ring for the previously mentioned halo and the rest are flair rings. You can have as many of them as you wish:

```
_lensFlair.setHaloAsset(new HaloRing ());
_lensFlair.addFlareAsset(new FlairRing1());
_lensFlair.addFlareAsset(new FlairRing2());
_lensFlair.addFlareAsset(new FlairRing3());
_lensFlair.addFlareAsset(new FlairRing4());
_lensFlair.addFlareAsset(new FlairRing5());
```

The last step here is to add it to the `_view`. In this case, we use `addOverlay()` and not the `addChild()` method because the first is designed to add overlay sprite layers above the `View3D` itself and because `LensFlair` implements `IOverlay` interface, it is added this way:

```
_view.addOverlay(_lensFlair);
```

Basically, this is enough to run the effect, but I decided to add some Bitmap filtering to it in order to get a more unique look to the end result. Look at the constructor method. We retrieve our `LensFlair` overlay sprite with the following line:

```
_overlay=_view.overlay;
```

Then we create a `BitmapData` object to which we then draw our `LensFlair` sprite and pass it into the `_bitmapContainer` bitmap, which is what we are actually going to see on the screen:

```
_renderBData=new BitmapData(stage.stageWidth,stage.stageHeight);
_bitmapContainer=new Bitmap(_renderBData);
this.addChild(_bitmapContainer);
```

Last, we set the `LensFlair` overlay visibility to false as we are not going to see it anyway:

```
_overlay.visible=false;
```

Inside the `initFilters()` method, we set up three filters—`ColorMatrixFilter`, `DisplacementMapFilter`, and `BlurFilter`. Also we need to position the `_overlay` anew at the flash stage when we draw it to the `_renderBData` object. For this, we create a Matrix object with the following line:

```
_drawMatr=new Matrix(1,0,0,1,500,180);
```

Next, we call the `applyFilters()` function inside `onEnterFrame()`, inside which we draw our `LensFlair` overlay to the `BitmapData` and then apply to it the filters, as shown in the following:

```
_renderBData.draw(_overlay,_drawMatr);
    _renderBData.applyFilter(_renderBData,_renderBData.rect,new
Point(0,0),_blurF);
    _renderBData.applyFilter(_renderBData,_renderBData.rect,new
Point(0,0),_colMatrixF);
    _renderBData.applyFilter(_renderBData,_renderBData.rect,new
Point(0,0),_displF);
```

There's more...

You can add more Sprite overlays for any kind of 2D/3D effects easily. `LensFlare` implements the `IOverlay` interface. So all you need to do is to implement this interface in your `Sprite`/`MovieClip` class and then you are able to plug it into Away3D's overlays system naturally.

Masking 3D objects

Sometimes you may wish to mask a 3D object just as you would normally do with regular 2D flash graphics. In this recipe, you will learn how to mask a 3D geometry. In Away3D, you can achieve this by accessing the sprite container that wraps a particular 3D model. This way, you can create a wide range of mask effects for your project.

Getting ready

Just set up an Away3D scene using the `AwayTemplate` class and you are good to go.

How to do it...

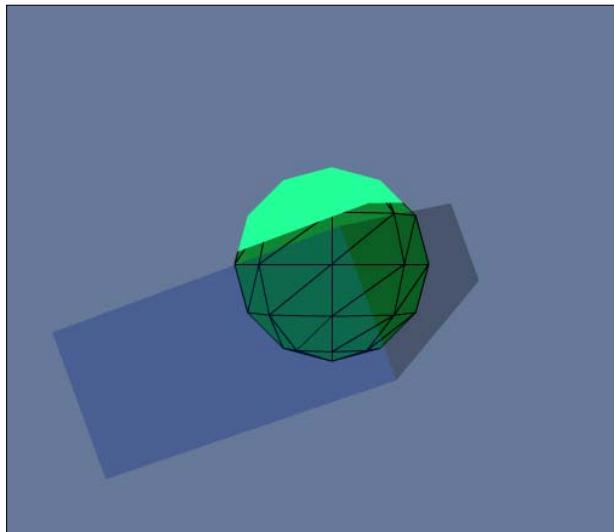
In this program, we create two spheres and set them one on top of another, then apply two different materials to them. We mask one of the spheres with a cube primitive, which we then tween over it in order to reveal the masked sphere:

```
ViewMasking.as
package
{
    public class ViewMasking extends AwayTemplate
```

```
{  
    private var _v1Sphere:Sphere;  
    private var _v2Sphere:Sphere;  
    private var _maskCube:Mesh;  
    private var _realCube:Cube;  
    private var _colMat:ColorMaterial;  
    private var _wireMat:WireColorMaterial;  
    public function ViewMasking()  
    {  
        super();  
        _cam.z=-200;  
    }  
    override protected function initMaterials() : void{  
        _colMat=new ColorMaterial(0x23ff99);  
        _wireMat=new WireColorMaterial(0x128733);  
    }  
    override protected function initGeometry() : void{  
        _v1Sphere=new Sphere({radius:40,material:_  
        wireMat,ownCanvas:true});  
        _v1Sphere.z=200;  
        _v1Sphere.screenZOffset=10;  
        _v2Sphere=new Sphere({radius:40,material:_  
        colMat,ownCanvas:true});  
        _v2Sphere.z=200;  
        _v2Sphere.screenZOffset=20;  
        _view.scene.addChild(_v2Sphere);  
        _view.scene.addChild(_v1Sphere);  
        var maskedSprite:Sprite=_v1Sphere.session.getContainer(_view) as  
        Sprite;  
        _realCube=new Cube({material:new ColorMaterial(0x092287),width:3  
        00,height:40,depth:80,ownCanvas:true});  
        _realCube.cubeMaterials.front=new ColorMaterial(0x000000);  
        _realCube.cubeMaterials.back=new ColorMaterial(0x000000);  
        _realCube.z=150;  
        _realCube.y=100;  
        _realCube.rotationZ=20;  
        _realCube.alpha=0.3;  
        _view.scene.addChild(_realCube);  
        _maskCube=_realCube.clone()as Mesh;  
        _maskCube.z=150;  
        _view.scene.addChild(_maskCube);  
        _maskCube.ownCanvas=true;  
        var maskSprite:Sprite=_maskCube.ownSession.getContainer(_view)  
        as Sprite;  
        maskedSprite.mask=maskSprite;
```

```
        TweenMax.to(_realCube, 3, {y:-100, rotationY:180, repeat:-1,yoyo:true});  
    }  
    override protected function onEnterFrame(e:Event) : void{  
        super.onEnterFrame(e);  
        _maskCube.transform=_realCube.transform;  
    }  
  
}  
}
```

In this image, we see that as the cube overlays the sphere, it reveals the masked second sphere, resulting in an effect of material transition:



How it works...

All the important setup is located inside the `initGeometry()` function. First we create two identical spheres, namely, `_v1Sphere` and `_v2Sphere` assigning both the same position. The only difference is materials as we want to get an effect of material change in the masked area. Notice also that we set the `ownCanvas` property of each sphere to true. This way, we are able to access the wrapping sprite of each 3D object:

```
_v1Sphere=new Sphere({radius:40,material:_wireMat,ownCanvas:true});  
_v1Sphere.z=200;
```

As we want the wireframed sphere to be at the front even that its z is identical to the second sphere we should use screenZOffset property when ownCanvas is set to true:

```
_v1Sphere.screenZOffset=10; - It's important to explain what is  
screenZOffset,  
_v2Sphere=new Sphere({radius:40,material:_  
colMat,ownCanvas:true});  
_v2Sphere.z=200;  
_v2Sphere.screenZOffset=20;  
_view.scene.addChild(_v2Sphere);  
_view.scene.addChild(_v1Sphere);
```

Next, we access the sprite container of the sphere to be masked with the following line:

```
var maskedSprite:Sprite=_v1Sphere.session.getContainer(_view) as  
Sprite;
```

Now we set up two cubes—one is going to serve as a mask and therefore will be invisible. The second one is an exact clone of the first and is going to inherit the transformation of the mask cube so that in the runtime, it will look like the visible cube primitive when going over the sphere like a scanner reveals the "hidden" texture. Here is the cubes setup code block:

```
_realCube=new Cube({material:new ColorMaterial(0x092287),width:3  
00,height:40,depth:80,ownCanvas:true});  
_realCube.cubeMaterials.front=new ColorMaterial(0x000000);  
_realCube.cubeMaterials.back=new ColorMaterial(0x000000);  
_realCube.z=150;  
_realCube.y=100;  
_realCube.rotationZ=20;  
_realCube.alpha=0.3;  
_view.scene.addChild(_realCube);  
_maskCube=_realCube.clone() as Mesh;  
_maskCube.z=150;  
_view.scene.addChild(_maskCube);  
_maskCube.ownCanvas=true;
```

We set up ownCanvas=true for the cubes too because we want to adjust opacity for one and to get a wrapping sprite of the other for the mask setup, as shown in the following line:

```
var maskSprite:Sprite=_maskCube.ownSession.getContainer(_view)  
as Sprite;  
maskedSprite.mask=maskSprite;
```

Lastly, in this method, we apply the tween to the scanning cube which moves up and down passing over the masked sphere and revealing the sphere underneath:

```
TweenMax.to(_realCube,3,{y:-100,rotationY:180,repeat:-  
1,yoyo:true});
```

An additional thing you should perform is to synchronize the transformation of both the cubes that we do in `onEnterFrame()` method copying the transform Matrix of one cube to another:

```
_maskCube.transform= _realCube.transform; ;
```

There's more...

The effect in our demo is pretty simple, but you can take it to the next level with a little help of the imagination as well known flash developer Den Ivanov had done (not my relative actually) in a showcase called "park seasons" on his website <http://www.cleoag.ru/labs/flex/parkseasons/>. The example is done with PV3D, but the approach is the same.

6

Using Text and 2D Graphics to Amaze

In this chapter, we will cover:

- ▶ Setting dynamic text with TextField3D
- ▶ Interactive animation of text along a path
- ▶ Creating 3D objects from 2D vector data
- ▶ Drawing with segments in 3D
- ▶ Creating a 3D illusion with Away3D sprites

Introduction

The essential part of almost every flash application, especially if you develop a website, is text. Although it depends on you to decide how much text content to put into your project and how much to leave for other graphical elements, the truth is that it is impossible not to insert even a single line of it. Flash text is basically 2D. In many scenarios, the developers of 3D content keep the text flat as it is. In other cases, you would like to have it 3D like the rest of the graphic elements in the scene. Also, you may consider incorporating 2D shapes in 3D space as well. It might be useful because of the performance consideration or for the sake of some unique design specification.

All this is possible with Away3D. You can get 2D text displayed and transformed in 3D space. If you wish, you can turn it into 3D by extruding it. Besides, Away3D enables you to get into scene 2D vector graphics and manipulate it just as any other object in the scene. To learn how to do these things is the goal of this chapter. So let's get started.

Setting dynamic text with TextField3D

Inside the Away3D primitives package, among numerous primitives, there is also one called `TextField3D`. It is quite different from the rest mainly because its form is 2D, which can be manipulated in a three dimensional world. In the following example you will learn how to set up a dynamic text with `TextField3D` and how to add to its shape a third dimension.

Getting ready

1. The first thing we need to do is to embed fonts inside our application as `TextField3D` uses embedded fonts only. We will prepare an `swf` resource file with our font embedded into it. Open Adobe Flash; type some dynamic text right on the stage so that when compiling `swf`, it would contain the font data.
2. Now, inside the text properties panel, make sure that the text is dynamic and you set font family to *Arial*.
3. Click the **Character Embedding ...** button and in the opened list of different characters, select only the following: Uppercase, Lowercase, Numerals, and Punctuation. Save the file with the name `fontsEmbed.fla` and compile the `swf`.
4. Set up a new Away3D scene using the `AwayTemplate` class. Make sure you put the `fontsEmbed.swf` inside your project as we are going to embed it into the application. Alternatively, you can find the ready version of it inside the assets folder of this chapter's source code.
5. In this example, we will use the "InputText" and "CheckBox" controls from a great components library by Keith Peters called *MinimalComps*. You can get it here: <http://www.minimalcomps.com/>.

How to do it...

In this demo, we set up an `InputText` field and connect it to the text property of the `TextField3D`. Any characters you type inside the `InputText` are updated instantly inside the `TextField3D` object:

`Text3DDemo.as`

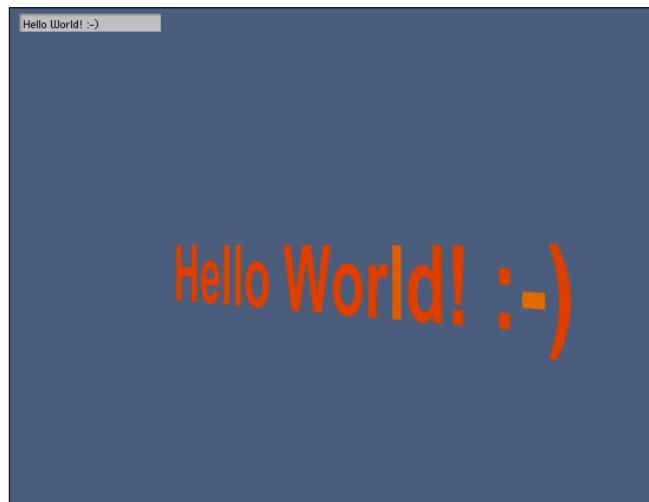
```
package
{
    public class Text3DDemo extends AwayTemplate
    {
        [Embed("assets/fontsEmbed.swf", mimeType="application/octet-
stream")]
        private var Fonts:Class;
        private var _text3d:TextField3D;
```

```
private var _textExtrude:TextExtrusion;
private var _pointLight:PointLight3D;
private var _shadeMat:ShadingColorMaterial;
public function Text3DDemo()
{
    super();
    initLights();
    initGUI();
}
override protected function initMaterials() : void{
    _shadeMat=new ShadingColorMaterial(0xff3400);
}
override protected function initGeometry() : void{

    VectorText.extractFont(new Fonts());
    _text3d=new TextField3D("Arial");
    _text3d.text="type your text";
    _text3d.bothsides=true;
    _text3d.material=_shadeMat;
    _view.scene.addChild(_text3d);
    _text3d.z=200;
    _text3d.x=-50;
    _text3d.rotationY=55;

}
private function initGUI():void{
    var input:InputText=new InputText(_view,-300,-200,"type your
text",onTextInput);
    input.width=120;
    input.draw();
}
private function onTextInput(e:Event):void{
    _text3d.text=e.target.text;
}
private function initLights():void{
    _pointLight=new PointLight3D();
    _pointLight.position=new Vector3D(0,200,0);
    _view.scene.addLight(_pointLight);
}
}
```

Here is the famous "Hello World" typed in a `TextField3D` instance:



How it works...

Let's explain what was done here. All the important stuff is located inside the `initGeometry()` method. Here we first extract the font from the embedded `fontsEmbed.swf` using a utility class `VectorText.extractFont()` which resides inside the `wumedia.vector` package. This class automatically extracts all the embedded fonts from the `swf`. Next, we set up a `TextField3D` as follows:

```
_text3d=new TextField3D("Arial");
_text3d.text="type your text";
_text3d.bothsides=true;
_text3d.material=_shadeMat;
_view.scene.addChild(_text3d);
```

Make sure you specify the font Family name inside the `TextField3D` constructor exactly as it appears in the Flash IDE text property panel.

Next, we set up the `InputText` control where we type our text. We do it inside the `initGUI()` function:

```
private function initGUI():void{
    var input:InputText=new InputText(_view,-300,-200,"type your
    text",onTextInput);
    input.width=120;
    input.draw();
}
```

Inside the `InputText()` constructor, we assign the `onTextInput` event handler which is called each time you type a character inside the text field. Inside the event handler, we update our `TextField3D` instance text property, as shown in the following lines:

```
private function onTextInput(e:Event):void{
    _text3d.text=e.target.text;
}
```

Pretty easy, isn't it?

There's more...

Let's take the previous example further and add a third dimension to the text shape! Away3D has got a special extruder class located inside the `away3d.extrusions` package named **TextExtrusion**. It is going to do all the magic for us.

Add an instance of the `TextExtrusion` class inside the global variables as follows:

```
private var _textExtrude:TextExtrusion;
```

Now let's add additional GUI control called `CheckBox` (from "MinimalComps" library) to the `initGUI()` function. We are going to use it to switch back and forth from 2D to 3D text. Add this line inside the `initGUI()` method:

```
var cbox:CheckBox=new CheckBox(_view,-300,-250,"Set
3d",onCheckBoxPress);
cbox.draw();
```

We set an event handler for the `CheckBox` called `onCheckBoxPress()` which takes care of any 2D/3D appearance change of the `TextField3D`. Add the following code to your class:

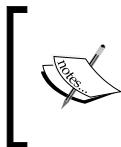
```
private function onCheckBoxPress(e:Event):void{
    if(e.target.selected){
        extrudeText();
    }else{
        resetExtrude();
    }
}
```

Inside the previous method, we call two functions. depending on `e.target` value. If the `CheckBox` is selected `extrudeText()` method is called which extrudes the text. Let's add it to the stack:

```
private function extrudeText():void{
    _text3d.visible=false;
    _textExtrude=new TextExtrusion(_text3d,{depth:20,bothsides:tr
ue});
    _view.scene.addChild(_textExtrude);
    _textExtrude.z=200;
```

```
_textExtrude.x=0;  
_textExtrude.rotationY=55;  
_textExtrude.scale(1.2);  
}
```

As you can see from the preceding block, the `TextExtrusion` implementation is very straightforward. The important properties to specify are extrusion, depth, and `bothsides`.



Note that if you specify both sides as false, then the `TextExtrusion` will show the material on outer sides of the faces only. So that when you look at the inner surface of the geometry, it would appear transparent because the normals face outwards from the shape.

When the user unselects the `CheckBox`, we want to reset our 3D text to 2D. In order to achieve this, we add the following method to the stack:

```
private function resetExtrude():void{  
    if(_textExtrude){  
        _view.scene.removeChild(_textExtrude);  
        _textExtrude=null;  
    }  
    _text3d.visible=true;  
}
```

The last thing we need to add is the rotation of the text while it is extruded. We add the rotation inside the `onEnterFrame()` method as follows:

```
override protected function onEnterFrame(e:Event) : void{  
    super.onEnterFrame(e);  
    if(_textExtrude){  
        _textExtrude.rotationY++;  
    }  
}
```

In this image, the previous flat version of "Hello World" now has got a depth which makes it appear in full 3D text:



There is one major flaw that you can find in the extrusion example and it is lack of capping at the front and at the back of 3D text. That is currently the way Away3D creates text extrusions. There is a quick hack to solve it. Just create two more instances of your text, but flat with no extrusion and position them one at the front and another at the back of the extrusion. Align their transformations and you will get a fully-capped 3D text object.

Interactive animation of text along a path

You can create more fun with `TextField3D` by attaching it to a path. Away3D has got a special modifier class named `PathAlignModifier` that attaches a `TextField3D` text to a user specified path, transforming its shape accordingly to the path curve. Let's see how it works!

Getting ready

1. Follow the same steps as described in the *Getting ready* section of the previous recipe.
2. Create a new class that extends `AwayTemplate` and give it a name `Text3DAnimDemo.as`.

How to do it...

In the following example, we will create interactive animation of the text along a predefined path:

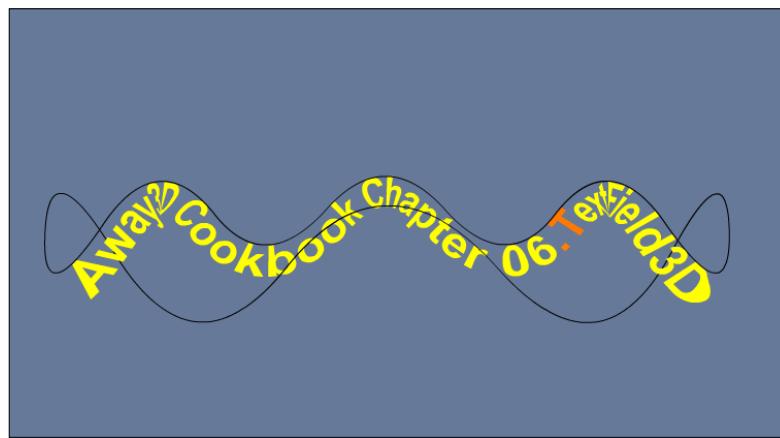
`Text3DAnimDemo.as`

```
package
{
    public class Text3DAnimDemo extends AwayTemplate
    {
        [Embed("assets/fontsEmbed.swf", mimeType="application/octet-
stream")]
        private var Fonts:Class;
        private var _text3d:TextField3D;
        private var _pointLight:PointLight3D;
        private var _shadeMat:ShadingColorMaterial;
        private var _pathModifier:PathAlignModifier;
        private var _lastMouseX:Number=0;
        private var _isDown:Boolean=false;
        private var _xVel:Number=0;
        private var _currentOffset:Number=0;
        public function Text3DAnimDemo()
```

```
{  
    super();  
    initLights();  
    _cam.position=new Vector3D(0,100,-500);  
}  
override protected function initListeners() : void{  
    super.initListeners();  
    stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);  
    stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);  
    stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);  
}  
override protected function initMaterials() : void{  
    _shadeMat=new ShadingColorMaterial(0xff3400);  
}  
override protected function initGeometry() : void{  
    VectorText.extractFont(new Fonts());  
    _text3d=new TextField3D("Arial");  
    _text3d.text="Away3D Cookbook Chapter 06.TextField3D";  
    _text3d.width=1200;  
    _text3d.size=32;  
    _text3d.bothsides=true;  
    _text3d.material=_shadeMat;  
    _view.scene.addChild(_text3d);  
    attachToPath();  
}  
override protected function onEnterFrame(e:Event) : void{  
    super.onEnterFrame(e);  
    if(!_isDown){  
        if(_xVel>=1){_xVel=0.95;}  
        _lastMouseX*=_xVel;  
        _pathModifier.offset.x+=_lastMouseX;  
        _pathModifier.execute();  
    }  
}  
private function onMouseDown(e:MouseEvent):void{  
    _isDown=true;  
    _lastMouseX=mouseX;  
    _currentOffset=_pathModifier.offset.x;  
}  
private function onMouseUp(e:MouseEvent):void{  
    _isDown=false;  
    _xVel=Math.abs(_lastMouseX-mouseX)/1000;
```

```
        }
        private function onMouseMove(e:MouseEvent):void{
            if(_isDown){
                var mouseOffset:Number=_lastMouseX-mouseX);
                _pathModifier.offset.x=mouseOffset+_currentOffset;
                _pathModifier.execute();
                trace(mouseOffset);
            }
        }
        private function initLights():void{
            _pointLight=new PointLight3D();
            _pointLight.position=new Vector3D(0,200,0);
            _view.scene.addLight(_pointLight);
        }
        private function attachToPath():void{
            var path:Path=new Path();
            path.continuousCurve(fillPPathData(),true);
            path.debugPath(_view.scene);
            path.showAnchors=false;
            _pathModifier=new PathAlignModifier(_text3d,path);
            _pathModifier.execute();
        }
        private function fillPPathData():Array{
            var tempArr:Array=new Array();
            tempArr.push(new Vector3D(-200,0, 0));
            tempArr.push(new Vector3D(-150, 100, 80));
            tempArr.push(new Vector3D(-90, 0, 170));
            tempArr.push(new Vector3D(0, 100, 200));
            tempArr.push(new Vector3D(90, 0, 170));
            tempArr.push(new Vector3D(150, 100, 90));
            tempArr.push(new Vector3D(200, 0, 0));
            tempArr.push(new Vector3D(170, 100,-90));
            tempArr.push(new Vector3D(90, 0, -170));
            tempArr.push(new Vector3D(0, 100, -200));
            tempArr.push(new Vector3D(-80, 0, -170));
            tempArr.push(new Vector3D(-170, 100, -80));
            return tempArr;
        }
    }
}
```

In the next image, you can see our `TextField3D` text attached to the Bezier path. The path itself is visible for debugging purposes:

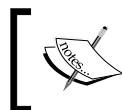


How it works...

We start from the `TextField3D` setup, which is executed inside the `initGeometry()` method:

```
override protected function initGeometry() : void{
    VectorText.extractFont(new Fonts());
    _text3d=new TextField3D("Arial");
    _text3d.text="Away3D Cookbook Chapter 06.TextField3D";
    _text3d.width=1200;
    _text3d.size=32;
    _text3d.bothsides=true;
    _text3d.material=_shadeMat;
    _view.scene.addChild(_text3d);
    attachToPath();
}
```

In the preceding code, we first extract the font from the embedded `fontsEmbed.swf` using the `VectorText` utility class. The last line of the method calls the `attachToPath()` function which snaps our created text to a path that we define inside the `fillPPathData()` method.



The array of `Vector3Ds` inside `fillPPathData` was created manually representing a circular path. However, you can also use `PreFab` to generate even more complex paths automatically.

Now let's go through the `attachToPath()` method. Inside the function, we first define a path. Then we call the `continuousCurve()` method by passing to its constructor an array of `Vector3Ds` points which is returned by the `fillPPathData()` function and setting the second parameter to true which tells the method to close the path:

```
var path:Path=new Path();
path.continuousCurve(fillPPathData(),true);
```

`continuousCurve()` function allows us to draw a smooth curvy path between the points.

Next, we turn on the debug mode of the path in order to be able to see how our text moves along it:

```
path.debugPath(_view.scene);
path.showAnchors=false;
```

Last step here is to apply the `PathAlignModifier` whose constructor requires two arguments: `TextField3D` and the path which we have already created:

```
_pathModifier=new PathAlignModifier(_text3d,path);
_pathModifier.execute();
```

In the beginning of this recipe, it was mentioned that it is going to be an interactive animation. What we are going to do is a kind of interactive throw of the text, but instead of throwing it in a direction of the mouse's movement, we are going to move it along the path. So here are additional steps we need to take to achieve this. Inside the `initListeners()` method, we add the following event listeners:

```
stage.addEventListener(MouseEvent.MOUSE_DOWN,onMouseDown);
stage.addEventListener(MouseEvent.MOUSE_MOVE,onMouseMove);
stage.addEventListener(MouseEvent.MOUSE_UP,onMouseUp);
```

When the user presses the mouse, the `onMouseDown()` method gets called. Inside it, we register the `mouseX` position in order to calculate the velocity of the text as a factor of the mouse movement distance till its release:

```
_lastMouseX=mouseX;
```

We also register the current position of the text on the path:

```
_currentOffset=_pathModifier.offset.x;
```

When the user pressed the mouse and started dragging it, the `onMouseMove()` method gets called continuously. It moves the text along the path while the mouse is pressed. Notice that after each `_pathModifier.offset.x` change, we call `_pathModifier.execute()` in order to update the visual transformation of the text along the path:

```
if(_isDown){
    var mouseOffset:Number=_lastMouseX-mouseX;
    _pathModifier.offset.x=mouseOffset+_currentOffset;
    _pathModifier.execute();
}
```

When we release the mouse, we set the `_isDown` Boolean to false so that the `onEnterFrame()` method can proceed with the animation from then on. Also, we calculate an easing factor which is the function of the distance from the `mouseX` position that was registered in `onMouseDown()` to the `mouseX` position when the user released the mouse:

```
private function onMouseUp(e:MouseEvent):void{
    _isDown=false;
    _xVel=Math.abs(_lastMouseX-mouseX)/1000;
}
```

In the final step, the text is being animated inside the `onEnterFrame()` function.

First we check that the mouse was released, then we check the range of the easing factor `_xVel` that we set in the `onMouseUp()` function. We must make sure that the max value of `_xVel` is less than 1.0. If the registered `_xVel` equals or is greater than 1.0, our text will never slow down and will spin along the path unless you close the program:

```
if(!_isDown){
    if(_xVel>=1){_xVel=0.95;}
```

Next we set easing to the `_lastMouseX` by multiplying it with `_xVel`. Then we use `_lastMouseX` to increase the `_pathModifier.offset.x` position which updates our text x position along the path:

```
_lastMouseX*=_xVel;
_pathModifier.offset.x+=_lastMouseX;
_pathModifier.execute();
```

We are done.

See also

In Chapter 8, *Prefab3D*, the following recipe: *Creating and animating paths*.

Creating 3D objects from 2D vector data

SWF movies can serve not only for material in Away3D, but also for creating 3D mesh objects. In this recipe, you will learn how to convert Flash native vector data into Away3D objects.

Getting ready

1. Open Adobe Flash and draw a circle in the middle of the stage.
2. Wrap it with `MovieClip`. The last thing we need to do before we are done is to set a linkage name so that Away3D can find the graphic symbol it needs to parse. Right-click on the `MovieClip` and select the **Export for ActionScript** checkbox.

3. Publish the movie.
4. Now set up a basic Away3D scene using the `AwayTemplate` class. Make sure you put the compiled SWF into your project's assets folder. Alternatively, you can use `AnimatedSWF.swf` which you can find inside the `assets` folder of this chapter's directory.

 If you wish to export bitmap graphics, you need to use the "Break Apart" command to convert it into a vector shape. Also, if you have got several shapes and would like them to be as separate meshes in the Away3D, just distribute them into different layers inside the `MovieClip`.

How to do it...

In Away3D, we use the `SWF` class to parse or load our `SWF` file, as it is shown in the following example:

`SWF3DDemo.as`

```
package
{
    public class SWF3DDemo extends AwayTemplate
    {
        [Embed(source="assets/animatedSWF.swf", mimeType="application/
octet-stream")]
        private var SwfData:Class;
        private var _loadedSWF:ObjectContainer3D;
        private var _counter:Number=0;
        public function SWF3DDemo()
        {
            super();
            _cam.z=-700;
            loadSWF();
        }
        private function loadSWF():void{
            _loadedSWF=Swf.parse(SwfData,{libraryClips:["SwfShape"]},
perspectiveOffset:150, perspectiveFocus:125,scaling:0.6});
            _view.scene.addChild(_loadedSWF);
            _loadedSWF.z=600;
            _loadedSWF.x=0;
            _loadedSWF.y=0;
            _loadedSWF.rotationX=45;
        }
    }
}
```

```
override protected function onEnterFrame(e:Event) : void{
    super.onEnterFrame(e);
    _loadedSWF.rotationY+=2;
}
}
```

We parse the SWF using the `Swf.parse()` method. Inside the optional parameters object, we put the linkage name of the Movie Clip containing the vector shape. This property is optional if you have only one `MovieClip` that you wish to parse. Otherwise, for each additional `MovieClip`, you push its linkage name into the `libraryClips` array. Now you can run the application and see the result.

The disadvantage of the SWF parser is the inability to import SWF animation. If you set a motion tween for the `MovieClip` you export, its animation would be discarded in Away3D. Also, if you set Shape tween for the Vector shapes that reside inside the `MovieClip`, the SWF parser will not import the shape at all.

How it works...

The 3D object you get in Away3D is not really 3D, although it inherits from `ObjectContainer3D`. It is rather a container with a single child mesh that contains a set of faces. Each face corresponds to a separate shape from SWF.

There's more...

As mentioned previously, we can't import animated SWFs, but isn't it boring to behold our vector-based 3D object motionless in the scene? Let's animate it!

There are two additional properties we can define inside the `Swf.parse()`. These are `perspectiveOffset` and `perspectiveFocus`. These are useful if you import SWF containing several shapes stacked inside the `MovieClip`. `perspectiveOffset` creates an even space between those shapes. `perspectiveFocus` scales the stacked shapes so that the scale of each successive shape grows up in perspective projection like geometrical form. To show what I mean, let's open Flash with our previous file and inside our `SwfShape` Movie Clip, duplicate seven more circles (I actually created rings for more fun), and distribute them to different layers. Publish the SWF.

Inside your `SWF3DDemo` class, add a global variable named `_counter`:

```
private var _counter:Number=0;
```

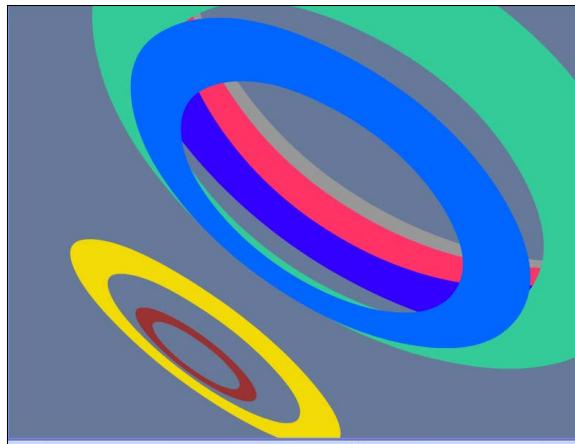
Change the SWF parser line to the new one:

```
_loadedSWF=Swf.parse(SwfData,{libraryClips:["SwfShape"],  
perspectiveOffset:150, perspectiveFocus:125,scaling:0.6});
```

After the SWF parsing line, let's add the following code:

```
var lib:Mesh=_loadedSWF.children[0]as Mesh;  
for each(var face:Face in lib.faces){  
    _counter=Math.floor(Math.random()*15-35);  
  
    for each (var v:Vertex in face.vertices){  
        TweenMax.to(v,2,{z:15*_counter,yoyo:true,repeat:-  
1,ease:Bounce.easeOut});  
    }  
}
```

If you got everything right, you should get the following result:



Now it looks cool and can serve as a background for a website page. What we have done in the code:

1. First—we access the faces array that resides inside the child mesh of the SWF container.
2. Then we run double `for each` loop statements so we can access each Face inside the elements array that stands for graphical ring.
3. Then, in the second loop, we access all the vertices for that Face in order to set an animation tween to the whole Face. For this specific example, we apply a random range of tween values along the local z-axis of the Faces so it gives a nice effect of a randomly shrinking spiral. Of course, you can experiment with various more complex vertex animation approaches to get even cooler results.

Drawing with segments in 3D

Segments in 3D are lines that build a mesh of the object, which is then covered with faces. Basically, these are 2D objects which can also be manipulated in 3D. In the following recipe, we will create a simple drawing application where we are going to paint our scene with a segment brush.

Getting ready

Set up a basic Away3D scene using AwayTemplate and we can start.

How to do it...

SegmentDrawDemo.as

```
package
{
    public class SegmentDrawDemo extends AwayTemplate
    {
        private var _container:Mesh;
        private var _canDraw:Boolean=false;
        private var _seg:Segment;
        private var _hoverCam:HoverCamera3D;
        private var _oldMouseX:Number=0;
        private var _oldMouseY:Number=0;
        private var _canMove:Boolean=true;
        public function SegmentDrawDemo()
        {
            super();
            setHowerCamera();
        }

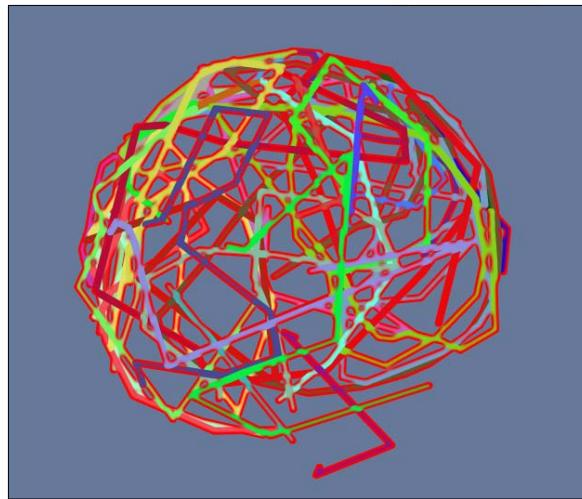
        override protected function initListeners() : void{
            super.initListeners();
            _view.scene.addEventListener(MouseEvent3D.MOUSE_
DOWN,onMouse3Down);
            _view.scene.addEventListener(MouseEvent3D.MOUSE_-
MOVE,onMouse3Move);
            _view.scene.addEventListener(MouseEvent3D.MOUSE_UP, onMouse3Up);
            stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
        }
    }
}
```

```
override protected function initGeometry() : void{
    _container=new Mesh();
    _container.ownCanvas=true;
    _container.filters=[new GlowFilter(16711680,1,5,5,5,2,true,fal
se)];
    _view.scene.addChild(_container);
    var sphere:Sphere=new Sphere({radius:120,material:new
ColorMaterial(0x993344)});
    _view.scene.addChild(sphere);
    sphere.ownCanvas=true;
    sphere.alpha=0;
}
override protected function onEnterFrame(e:Event) : void{
    super.onEnterFrame(e);
    if(_hoverCam){
        if(_canMove){
            _hoverCam.panAngle = (_view.mouseX - _oldMouseX);
            _hoverCam.tiltAngle = (stage.mouseY - _oldMouseY);
            _hoverCam.hover();
        }
    }
}
private function onMouseDown(e:MouseEvent):void{
    _oldMouseX=stage.mouseX;
    _oldMouseY=stage.mouseY
}

private function onMouse3Down(e:MouseEvent3D):void{
    _canMove=false;
    _seg=new Segment();
    var wireMat:WireColorMaterial=new WireColorMaterial(0xffffffff);
    wireMat.thickness=8;
    wireMat.wireColor=Math.floor(Math.random()*0xffffffff);
    _seg.material=wireMat;
    _seg.moveTo(e.sceneX,e.sceneY,e.sceneZ);
    _canDraw=true;
}
private function onMouse3Move(e:MouseEvent3D):void{
    if(_canDraw){
        _seg.lineTo(e.sceneX,e.sceneY,e.sceneZ);
        _container.addSegment(_seg);
    }
}
```

```
        }
        private function onMouse3Up(e:MouseEvent3D):void{
            _canMove=true;
            _canDraw=false;
        }
        private function setHoverCamera():void{
            _hoverCam=new HoverCamera3D();
            _view.camera=_hoverCam;
            _hoverCam.target=_container;
            _hoverCam.distance = 400;
            _hoverCam.wrapPanAngle=true;
            _hoverCam.steps=16;
            _hoverCam.yfactor=1;
        }
    }
```

As can be seen from the following image, you can have lots of fun with segment drawing:



How it works...

As you can see from this example, the downside of this approach is that you must have some Away3D geometry in the background of your scene. Otherwise, you will never get `MouseEvent3D` triggered. An alternative approach is to use the `camera.unproject()` method which returns your `Vector3D` coordinates in the scene of x and y input from the mouse. This technique implementation is embedded in the source file of the previous example, which can be found in this chapter's source code folder.

First we set up four listeners inside the `initListeners()` method. The first three are of the `MouseEvent3D` type and one of ActionScript generic `MouseEvent`. These events manage the drawing process based on user mouse input. Now let's go step-by-step through each of those event handlers and see what is going on there.

`onMouseDown()` event handler serves us only for registering the variables `_oldMouseX` and `_oldMouseY` for the `HoverCamera` transformations.

Now inside the `onMouse3Down()` handler, we draw the start point of the actual segment which has the drawing API pretty similar to the flash generic one. We also assign a `WireColorMaterial` because, in this case, we are able to specify thinness of the lines, which adds to them a third dimension:

```
_seg=new Segment();
var wireMat:WireColorMaterial=new WireColorMaterial(0xffffffff);
wireMat.thickness=8;
wireMat.wireColor=Math.floor(Math.random()*0xffffffff);
_seg.material=wireMat;
_seg.moveTo(e.sceneX,e.sceneY,e.sceneZ);
```

After the user presses the mouse and starts dragging it, the `onMouse3Move()` function is called where we draw the line from the position specified on `onMouse3Down()` to the location defined by the Mouse 3D scene position:

```
_seg.lineTo(e.sceneX,e.sceneY,e.sceneZ);
```

At the end of this operation, we add the drawn segment line to an empty mesh container which we created for that purpose:

```
_container.addSegment(_seg);
```



Segments can't be added directly to an `Away3D` scene, but only to objects that extend `Object3D`.



In `onMouse3Up()`, we just reset two Boolean flags—`_canMove` and `_canDraw`. The first we use to stop our hover camera's rotation while drawing and the second indicates to the `onMouse3Move()` event handler that the Mouse is in the pressed state.

The last method we need to cover is `initGeometry()`. Here we initiate a `_container` (`mesh`) which would hold the drawn segment. We also add to it a glow filter to set a nice effect to the lines:

```
_container=new Mesh();
_container.ownCanvas=true;
_container.filters=[new GlowFilter(16711680,1,5,5,5,2,true,false)];
_view.scene.addChild(_container);
```

Then we set a sphere primitive which is a kind of hack. It is going to serve us as a dummy object to register the MouseEvent3D input:

```
var sphere:Sphere=new Sphere({radius:120,material:new  
ColorMaterial(0x993344)});  
_view.scene.addChild(sphere);  
sphere.ownCanvas=true;  
sphere.alpha=0;
```

There's more...

You can get a very similar effect with much less CPU, if you use empty Sprite/MovieClip as texture and draw the line on the MovieClip.

See also

In Chapter 4, *Fun by Adding Interactivity*, the following recipe, *Interactively painting on the model's texture*.

Creating a 3D illusion with Away3D sprites

Often it is worth knowing how to "cheat" 3D with sprites that are by nature 2D objects. This recipe will show you how exactly that is done. Away3D has got a decent set of different sprites that are located inside the away3d.sprites package. Most of them contain a single Bitmap or MovieClip graphics, which always looks at the camera (billboard). Generally, if the model's geometrical shape is uniform like that of a sphere or an even tree's foliage, they are traditionally presented by regular billboards because their view doesn't change from different angles. But let's say you develop an RTS with hundreds of soldier models that you want to place in the scene and look 3D. A human model is not a sphere, it looks different from varying angles and to convince the player that the soldier is a 3D shape, you should enable him to view the models from different angles. It is obvious that putting into your scene even a couple of dozens of soldier meshes of relatively low poly can freeze the flash player. In such a case, the rescue comes from DirectionalSprite, which allows you to fake a 3D model with a set of images each one representing a model's view from a particular angle. The technique was very popular in the early stages of 3D games and some very prominent titles such as DukeNukem and many more utilized this approach. Let's see how to implement it in Away3D.

Getting ready

You can create a set of bitmaps which depict a 3D object from different angles by using a 3D package such as **Autodesk 3dsMax**. In our example, it is an extremely high poly of a man's head which was rendered from eight different angles. The more different angles of your 3D object you capture, the better the 3D effect of DirectionalSprite is going to look. You can use already rendered images of the head, which are found inside the assets folder of this chapter.

Set up a basic Away3D scene using `AwayTemplate` and you are ready to go.

How to do it...

In the following program, we set up a `DirectionalSprite` instance which holds a set of bitmaps with the head rendered from eight sides. Then the sprite is being rotated, and while changing its angle relative to camera, it presents a different image which was defined for that particular direction:

`DirSpriteDemo.as`

```
package
{
    public class DirSpriteDemo extends AwayTemplate
    {
        [Embed(source = "/assets/face360/front.png")]
        private var FrontBmp:Class;
        [Embed(source = "/assets/face360/back.png")]
        private var BackBmp:Class;
        [Embed(source = "/assets/face360/left.png")]
        private var LeftBmp:Class;
        [Embed(source = "/assets/face360/right.png")]
        private var RightBmp:Class;
        [Embed(source = "/assets/face360/frontLeft.png")]
        private var FrontLeftBmp:Class;
        [Embed(source = "/assets/face360/frontRight.png")]
        private var FrontRightBmp:Class;

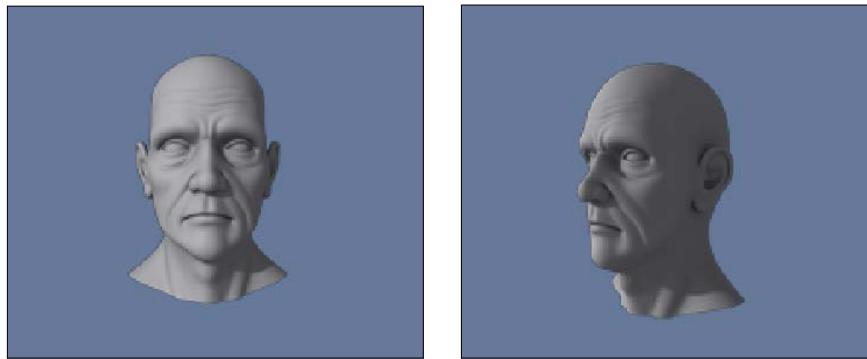
        [Embed(source = "/assets/face360/backLeft.png")]
        private var BackLeftBmp:Class;
        [Embed(source = "/assets/face360/backRight.png")]
        private var BackRightBmp:Class;
        private var _dirSprite:DirectionalSprite;
        private var _hoverCam:HoverCamera3D;
        private var _lastPanAngle:Number;
        private var _lastTiltAngle:Number;
        private var _lastMouseX:Number;
        private var _lastMouseY:Number;
        private var _canRotate:Boolean=false;
        public function DirSpriteDemo()
        {

            super();
            setHoverCam();
        }
    }
}
```

```
        }
        override protected function initGeometry() : void{
            _dirSprite=new DirectionalSprite();
            _dirSprite.addDirectionalMaterial(new Vertex(1,0,0),new
BitmapMaterial(Cast.bitmap(new FrontBmp())));///front
            _dirSprite.addDirectionalMaterial(new Vertex(0,0,1), new
BitmapMaterial(Cast.bitmap(new LeftBmp())));///left
            _dirSprite.addDirectionalMaterial(new Vertex(0,0,-1),new
BitmapMaterial(Cast.bitmap(new RightBmp())));///right
            _dirSprite.addDirectionalMaterial(new Vertex (-0.8,0,0.8), new
BitmapMaterial(Cast.bitmap(new BackLeftBmp())));///back left
            _dirSprite.addDirectionalMaterial(new Vertex (-0.8,0,-0.8),new
BitmapMaterial(Cast.bitmap(new BackRightBmp())));///back right
            _dirSprite.addDirectionalMaterial(new Vertex (0.8, 0, 0.8),new
BitmapMaterial(Cast.bitmap(new FrontLeftBmp())));//front left
            _dirSprite.addDirectionalMaterial(new Vertex (0.8, 0, -0.8),new
BitmapMaterial(Cast.bitmap(new FrontRightBmp())));///front right
            _dirSprite.addDirectionalMaterial(new Vertex (-1,0, 0),new
BitmapMaterial(Cast.bitmap(new BackBmp())));////back
            _view.scene.addSprite(_dirSprite);
        }
        override protected function initListeners() : void{
            super.initListeners();
            stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
            stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
        }
        override protected function onEnterFrame(e:Event) : void{
            super.onEnterFrame(e);
            if (_hoverCam) {
                if(_canRotate){
                    _hoverCam.panAngle = 0.5 * (stage.mouseX- _lastMouseX) +
_lastPanAngle;
                }
                _hoverCam.hover();
            }
        }
        private function onMouseDown(e:MouseEvent):void{
            _canRotate=true;
            _lastPanAngle = _hoverCam.panAngle;
            _lastTiltAngle = _hoverCam.tiltAngle;
            _lastMouseX = stage.mouseX;
            _lastMouseY = stage.mouseY;
        }
        private function onMouseUp(e:MouseEvent):void{
            _canRotate=false;
```

```
        }
        private function setHoverCam():void{
            _hoverCam=new HoverCamera3D();
            _hoverCam.panAngle = 90;
            _hoverCam.tiltAngle = 0;
            _hoverCam.distance=500;
            _view.camera=_hoverCam;
        }
    }
}
```

Here you can see the man's head from two different angles. The more images from different angles you add to the sprite, the smoother and more realistic looking your final result will be.



How it works...

In comparison to the previous examples, this one was really short and simple. All the important settings are located inside the `initGeometry()` function. We use the `dirSprite.addDirectionalMaterial()` method to assign a specific image to a particular direction. The first three arguments of the `addDirectionalMaterial()` method are local direction vectors for the fourth argument which is a `BitmapData`. Notice that the vectors for the angles such as 40-45 degrees, we use x/z coordinates 0.8 or -0.8. Actually the range between 0.7 to 0.8 (or -0.7 to -0.8) work best for such angles. Of course, you can define even more directions with more images assigned to them in order to achieve smoother results when moving around the sprite.

See also

For usage of `DepthOfFieldSprite`, see:

In *Chapter 2, Working with Away3D Cameras*, the following recipe, *Creating Camera Depth of Field Effect*.

For usage of `Sprite3D`, see:

In *Chapter 3, Animating the 3D World*, the following recipe, *Animating geometry with Tween Engines*.

7

Depth-sorting and Artifacts Solutions

In this chapter, we will cover:

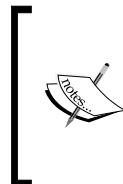
- ▶ Fixing geometry artifacts with Frustum And NearField Clipping
- ▶ Removing artifacts from intersecting objects
- ▶ Solving depth-sorting problems with Layers and Render Modes

Introduction

As we know, the objects in 3D space are positioned in three dimensions that can be described in the terms of computer screen as width, height, and depth. The problem is that usually only the third dimension imposes a whole bunch of problems on 3D model appearance, resulting in something called **geometry artifacts**. The cause of that problem rests in the very nature of the computer screen which is basically two dimensional, whereas the third dimension (depth) should be faked with projection of the objects on it, depending on their z-position in 3D space. To achieve an effect of 3D perspective is a relatively easy task, but with it comes to the need of depth sorting (**Z-Sorting**), which is an essential process for correct presentation of objects in their respective z-positions. Here comes a well-known issue—object geometry artifacts that are expressed in broken or intersecting faces. This problem stems not only from incorrect Z-Sorting against another object in the scene, but may also result from wrong modeling practices.

Fortunately, the Away3D engine incorporates a number of Z-Sorting algorithms that are capable of solving depth-sorting issues. These will help you to solve most of the depth-sorting problems for you. Nevertheless, you should know that a greater responsibility rests upon you when it concerns preparing a 3D model for Away3D. Wrong modeling may result in acute Z-Sorting issues which would not be possible to fix using Away3D tools.

That is why this chapter is dedicated to learning how to solve general Z-Sorting issues and how to take care of geometry anomalies. You will also learn a few techniques dealing with depth sorting in complex scenes as well as some useful tips on how to model for a real time 3D engine in the right way in order to reduce possible sorting issues to a minimum.



The main issue is that in GPU-based 3D rendering, the engine can use Z-buffer rendering which checks Z-Sorting for every pixel on the rendering screen. But in CPU-based rendering, such as the Flash based engine, we use a kind of Painters algorithm, which paints the polygons from the most far away to the nearest, and that does give us a desirable result in most cases.

Fixing geometry artifacts with Frustum and NearField clipping

One of the most stumbled on and irritating artifacts is the disappearance of faces in the region of NearField clipping (regions close to the screen boundary). These are known as clipping artifacts. If you work on a third person shooter or a racing game, you will certainly encounter that weird geometry disappearing close to the camera boundaries. The problem is that the default clipping in Away3D is RectangleClipping, which removes any face whose vertices fall outside x and y of the defined clipping boundaries. In order to solve this, we need to use more advanced clipping classes such as NearfieldClipping or FrustumClipping which reside inside the `away3d.core.clip` package. Let's see how to implement them.

Getting ready

Set up a basic Away3D scene extending the `AwayTemplate` class and you are ready to go.

How to do it...

The following program example has got a default `RectangleClipping` mode. The scene contains a kind of tunnel created from the `Cylinder` primitives which move towards the camera. Running this example, you can clearly see the problem when the faces of the cylinder, which are approaching the screen boundaries, are being removed before they pass the near plane:

`ClippingDemo.as`

```
package
{
    public class ClippingDemo extends AwayTemplate
    {
}
```

```
private var _cyl:Cylinder;
private var _backCyl:Object3D;
private var _bitMat:BitmapMaterial;
private var _wireMat:WireframeMaterial;

public function ClippingDemo()
{
    super();

}

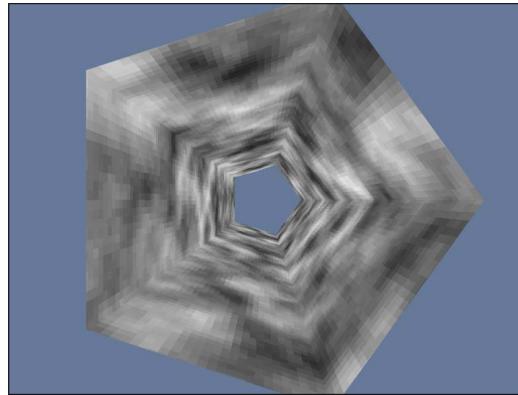
override protected function initMaterials() : void{
    var btd:BitmapData=new BitmapData(128,128);
    btd.perlinNoise(15,16,12,23456,true,true,7,true);
    _bitMat=new BitmapMaterial(btd);
    _wireMat=new WireframeMaterial();
}
override protected function initGeometry():void{
    _cyl=new Cylinder({height:1200,radius:100,segmentsH:4,segmentsW:
5,material:_bitMat});
    _cyl.movePivot(0,-_cyl.height/2,0);
    _cyl.openEnded=true;
    _view.scene.addChild(_cyl);
    _cyl.z=1200;
    _cyl.rotationX=90;
    _cyl.invertFaces();
    initBackCyl();

}

override protected function onEnterFrame(e:Event) : void{
    super.onEnterFrame(e);
    if(_cyl.z<=0){
        _cyl.z=_backCyl.z+1200;
    }
    _cyl.z-=5;
    if(_backCyl.z<=0){
        _backCyl.z=_cyl.z+1200;
    }
    _backCyl.z-=5;
}
private function initBackCyl():void{
    _backCyl=_cyl.clone();
    _backCyl.z=_cyl.z+_cyl.height*2;
    _view.scene.addChild(_backCyl);
}

}
```

In this screenshot, you have a perfect example of the lack of near plane clipping which is expressed by the automatic removal of the closest faces to the camera:



How it works...

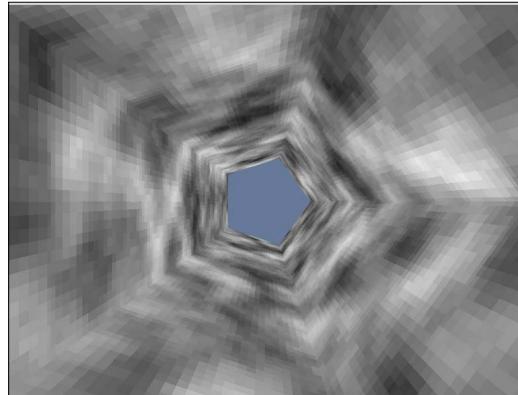
It may seem from the preceding image that the cylinder's front edges are at some distance from the screen boundaries. But, in fact, the faces that are closest to the Frustum near plane are removed, resulting in this undesired appearance.

Let's take care of that issue. Away3D's NearfieldClipping class solves the problem of NearField geometry disappearance by using face subdivision along the screen boundaries.

Add the following code inside the ClippingDemo constructor:

```
var _nearClip:NearfieldClipping=new NearfieldClipping();
_view.clipping=_nearClip;
```

Now we have the expected behavior where the faces that are closest to the screen boundaries remain intact:



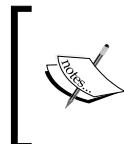
There's more...

As was said before, NearfieldClipping takes care of the artifacts in the region of the Frustum's near plane. Away3D Frustum contains eight planes. In some scenarios, especially when the application contains interior environments, the clipping artifacts may occur along other Frustum planes. In such a case, you can use the FrustumClipping class which processes clipping checking against all the planes.

In the preceding program, remove or comment the NearfieldClipping line and write the following instead:

```
var _frustumClip:FrustumClipping=new FrustumClipping();  
_view.clipping=_frustumClip;
```

The visual result is the same as in this particular example; the scene geometry always intersects with the near plane only.



Note that NearfieldClipping as well as FrustumClipping are memory-intensive processes. In fact, FrustumClipping eats even more CPU cycles. Therefore, if you have only NearField artifacts, there is no reason to use FrustumClipping.



Removing artifacts from intersecting objects

Generally, it is a bad idea to have different geometry objects intersecting intentionally. The reason for this is that the cleaning of the resulted artifacts is quite difficult and also memory intensive. However, in certain cases you will need to deal with 3D objects intersections, and this recipe is going to show you how to reduce the intersection artifacts to a minimum.

Getting ready

Set up a basic Away3D scene extending AwayTemplate class and you are ready to go.

How to do it...

In the following program, we set up two planes and one cylinder primitive at the same position in order to cause them to intersect. We also create a static plane which serves as floor whereas the rest of the objects intersect it moving through its surface by animation tween. The initial code has no intersection sorting algorithm applied:

```
IntersectRenderDemo.as
```

```
package  
{
```

```
public class IntersectRenderDemo extends AwayTemplate
{
    private var _p1:Plane;
    private var _p2:Plane;
    private var _cyl:Cylinder;
    public function IntersectRenderDemo()
    {
        super();

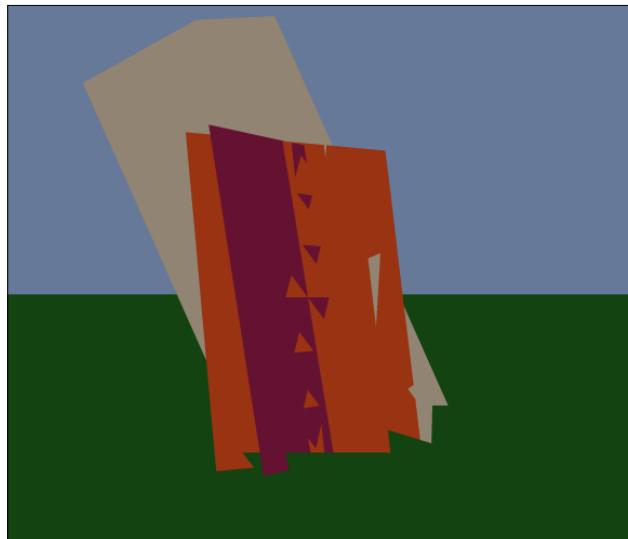
    }
    override protected function initGeometry() : void{
        setFloor();
        setCylinder();
        _p1=new Plane({material:new ColorMaterial(0x993312),width:200,height:270,bothsides:true,yUp:false});
        _p1.segmentsH=6;
        _p1.segmentsW=6;
        _view.scene.addChild(_p1);
        _p1.z=800;
        _p2=new Plane({material:new ColorMaterial(0x651232),width:200,height:270,bothsides:true,yUp:false});
        _p2.segmentsH=6;
        _p2.segmentsW=6;
        _view.scene.addChild(_p2);
        _p2.z=800;
        _p2.rotationY=90;

        TweenMax.to(_cyl,3,{y:300,yoyo:true,repeat:-1});

    }
    override protected function onEnterFrame(e:Event) : void{
        super.onEnterFrame(e);
        _p1.rotationY+=3;
        _p1.rotationX--;
        _p2.rotationY-=3;
        _p2.rotationX++;
    }
    private function setFloor():void{
        var floor:Plane=new Plane({material:new ColorMaterial(0x124311),width:1000,height:1000,bothsides:true});
        floor.segmentsH=floor.segmentsW=8;
        _view.scene.addChild(floor);
        floor.y=-100;
        floor.z=900;
```

```
    floor.rotationX=12;
}
private function setCylinder():void{
    _cyl=new Cylinder({height:600, radius:100, segmentsH:5, segmentsW:5
,material:new ColorMaterial(0x928473)});
    _view.scene.addChild(_cyl);
    _cyl.rotationZ=24;
    _cyl.z=900;
    _cyl.y=-400;
}
}
```

In the next image, you can see the artifacts all over the intersecting geometry. Basically, the more faces your object contains, the more artifacts would appear:



Now let's add intersection sorting to our scene using the `Renderer` class. Add the following code to the `IntersectRenderDemo` constructor:

```
_view.renderer=Renderer.INTERSECTING_OBJECTS;
```

`Renderer.INTERSECTING_OBJECTS` removes almost all the artifacts, as is seen in the following screenshot:



However, it is also a real frame rate killer, especially for the scene with a high polygon count.

How it works...

`Renderer` is a static class which allows us a quick access to different Away3D renderers which are, at the time of writing, two classes: `BasicRenderer` and `QuadrantRenderer`. Calling the `Renderer.INTERSECTING_OBJECTS` getter, we in fact assign a `QuadrantRenderer` to the `_view.renderer` property. So instead of using `Renderer`, we can also set up a `QuadrantRenderer` directly as follows:

```
var quadFilter:QuadrantRiddleFilter=new QuadrantRiddleFilter(23000);
var anotherF:AnotherRivalFilter=new AnotherRivalFilter(10000);
var quadRend:QuadrantRenderer=new QuadrantRenderer(quadFilter,anothe
rF);
_view.renderer=quadRend;
```

`QuadrantRenderer` accepts two sorting filters to the constructor arguments—`QuadrantRiddleFilter` and `AnotherRivalFilter`. If you need to solve intersecting objects sorting, you need to pass both these filters to the `QuadrantRenderer` instance. In case you are interested only in correct Z-Sorting, you should use only `AnotherRivalFilter`.

Again, using `QuadrantRenderer` for most of the application would be impractical because even in the preceding example, which is relatively lightweight, the frame rate drops dramatically. Therefore, you should try avoiding geometry intersection as much as possible in order to cancel the need to use `QuadrantRenderer` for fixing those kinds of problems.

Solving depth-sorting problems with Layers and Render Modes

In this recipe, we are going to master Z-Sorting issues, which you will encounter quite frequently during your development with Away3D. By default, the Away3D engine deals pretty well with depth sorting when the scene geometry positioning isn't complicated. For instance if you disperse a few hundred spheres around your scene, each with different z positions, it is likely that you will see no Z-Sorting issues, except when some of those objects intersect. The potentially troublemaking scenarios are when different objects are located closely to each other and viewed at sloped angles, or cases where there is a background at close distance to the rest of the geometry. The best example is when you develop a racing game; the ground plane with different objects deployed on it would cause numerous Z-Sorting issues, especially if the player's view is third person, at a sharp angle, and located close to the ground.

In this recipe, we are going to review an example which resembles the aforementioned scenario and learn how to fix those issues with just a few lines of code.

Getting ready

Set up a basic Away3D scene extending the `AwayTemplate` class and you are ready to.

Now we need to extend the Away3D `Sphere` primitive so that we can use it in the "Brownian Motion" algorithm for our example. For this quick demo, we put the `FSphere` class inline right after the `LayersDemo` class block. Generally, it is bad practice to have more than one class in a file. Put the following code on a new line after the last curly bracket of the `LayersDemo` class:

```
import away3d.primitives.Sphere;

class FSphere extends Sphere{
    public var vx:Number=0;
    public var vy:Number=0;
    public var vz:Number=0;
    public function FSphere(init:Object=null) {
        super(init);
    }
}
```

How to do it...

In the following example, we set up a basic scene with default Z-Sorting in order to demonstrate the issue. We create a ground plane and a bunch of spheres which are deployed randomly in X and Z directions very closely to the surface of the plane:

LayersDemo.as

```
package
{
    public class LayersDemo extends AwayTemplate
    {
        private var _pl1:Plane;
        private var _sphereArr:Array=[];
        public function LayersDemo()
        {
            super();
            _view.clipping=new FrustumClipping();
            _cam.y=80;
            _cam.lookAt(_pl1.position,Vector3D.Y_AXIS);

        }
        override protected function initGeometry() : void{
            initFloor();
            initSpheres();
        }
        override protected function onEnterFrame(e:Event) : void{
            super.onEnterFrame(e);
            updateSceneObjectsPos();
        }
        private function initFloor():void{
            _pl1=new Plane({material:new ColorMaterial(0x993312),width:1024,
height:2048,bothsides:false,yUp:true});
            _view.scene.addChild(_pl1);
            _pl1.z=1024;
            _pl1.y=-100;
        }
        private function initSpheres():void{
            for(var i:int=0;i<6;++i){
                var sphere:FSphere=new FSphere({radius:20,material:new
ColorMaterial(Math.floor(Math.random()*0xFFFFFF))});
                sphere.segmentsH=6;
```

```

        sphere.segmentsW=6;
        _sphereArr.push(sphere);
        sphere.x=Math.random()*1000-500;
        sphere.y=-80;
        sphere.z=Math.random()*2000-1000;
        _view.scene.addChild(sphere);
    }
}

private function updateSceneObjectsPos():void{
    var arrLength:uint=_sphereArr.length;
    for(var i:int=0;i<arrLength;++i){
        _sphereArr[i].vx+=Math.random()*0.5-0.25;
        _sphereArr[i].vz+=Math.random()*0.5-0.25;
        _sphereArr[i].x+=_sphereArr[i].vx;
        _sphereArr[i].z+=_sphereArr[i].vz;
        if(_sphereArr[i].x>500){
            _sphereArr[i].x=-500;
        }else if(_sphereArr[i].x<-500){
            _sphereArr[i].x=500;
        }
        if(_sphereArr[i].z>2048){
            _sphereArr[i].z=0;
        }else if(_sphereArr[i].z<0){
            _sphereArr[i].z=2048;
        }
    }
}

}
}

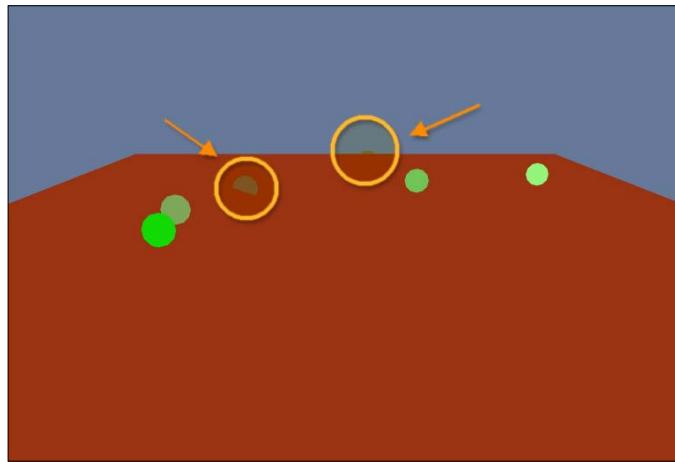
}

//-----extended Sphere with member vars for velocity
import away3d.primitives.Sphere;

class FSphere extends Sphere{
    public var vx:Number=0;
    public var vy:Number=0;
    public var vz:Number=0;
    public function FSphere(init:Object=null){
        super(init);
    }
}

```

In the next image, you can clearly see depth-sorting issues—some of the spheres are overlapped by the underlying plane:



After compiling and running your code, you will see numerous artifacts as those marked in the previous image with yellow circles. That is a typical kind of Z-Sorting issue when viewing geometry against a background plane at such an angle.

Now is the time to fix that problem. First add the following member variables above the `LayersDemo()` constructor:

```
private var _session1:SpriteSession;
private var _session2:SpriteSession;
```

Next, inside the `initGeometry()` method, put these two lines before the `initFloor()` function:

```
_session1=new SpriteSession();
_session2=new SpriteSession();
```

And two more lines after the `initSpheres()` method:

```
_session1.screenZ=1;
_session2.screenZ=100;
```

Inside the `initFloor()` function, put the next line after the rest of the code:

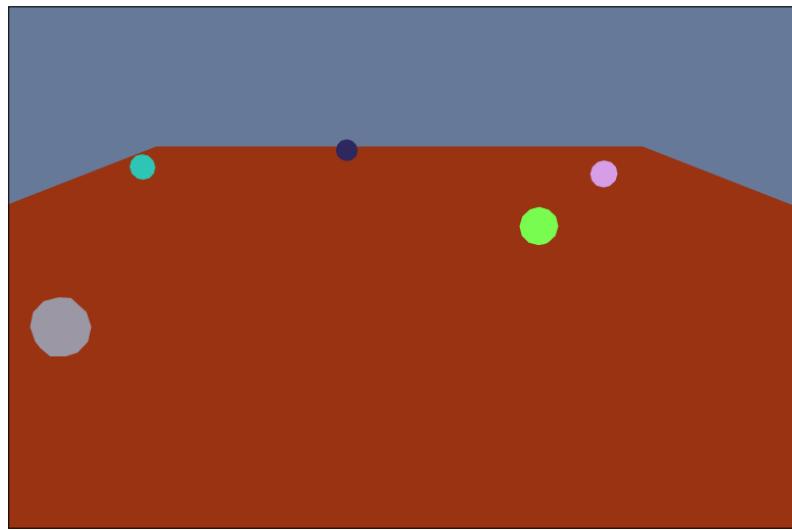
```
_pl1.ownSession=_session2;
```

Finally, add the following line inside the "for" loop statement of the `initSpheres()` method after the `_view.scene.addChild(sphere);` line:

```
sphere.ownSession=_session1;
```

Run the updated application and you should see that all the previous artifacts have disappeared!

The following screenshot shows us a clean sorting after we have separated the spheres and the plane by assigning them to different layers:



How it works...

Now let's see what we have done. Basically, we have made use of layers. In Away3D, layering of groups of objects is achieved via `SpriteSession` use. This class creates a sprite container which wraps all those objects which register to its instance through the `.ownSession` setter, which is found on any `Object3D` extended class. In our example, we first defined two such instances in order to set two separate layers; one for the floor plane and the second for all the spheres:

```
_session1=new SpriteSession();
_session2=new SpriteSession();
```

Then we defined the z order for those layers with the following lines:

```
_session1.screenZ=1;
_session2.screenZ=100;
```

This way, the `_session1` layer content is going to be always rendered above the `_session2`.

Finally, inside the `initFloor()` and `initSpheres()` functions, we assign those sessions to the `_p1` plane and all the sphere instances accordingly.

There's more...

Although the `SpriteSession` approach is quick as well as light in terms of memory consumption, it still may be useful to get acquainted with other Z-Sorting techniques that Away3D reserved for you. First usage of `SpriteSession` is very effective in scenarios where you wish to control depth sorting of a group of multiple objects. If you don't need such control, you can use `pushback` and `pushfront` properties of any `Object3D` subclass to get the similar result.

Let's comment in the previous example program all the lines that related to `SpriterSession`. Inside the `initFloor()` method, after `_pl1` initiation, add this line:

```
_pl1.pushback=true;
```

Now go to the `initSpheres()` method and inside the `for` loop, add the following line of code after the sphere initiation line:

```
sphere.pushfront=true;
```

Now run the program and you will see the same result as we achieved using `SpriterSession`, but only with two lines of code!

Z-Sorting using QuadrantRenderer

Of course, we are not to forget that Away3D has got a very advanced Z-Sorting tool which is `QuadrantRenderer`. The easiest way to set it up is via `Renderer.CORRECT_Z_ORDER`.

In the previous program, you can put the following line of code after commenting out `pushback/pushfront` implementations :

```
_view.renderer=Renderer.CORRECT_Z_ORDER;
```

Otherwise you can set it explicitly through `QuadrantRenderer` direct instancing as follows:

```
var quadR: QuadrantRenderer =new QuadrantRenderer(new  
AnotherRivalFilter());
```

Note that for Z-Sorting, we need to set up only `AnotherRivalFilter()`, which is passed to the `QuadrantRenderer` constructor arguments.

Last step, instead of `_view.renderer=Renderer.CORRECT_Z_ORDER`; assign to `_view.renderer quadR` instance:

```
_view.renderer= quadR;
```

 QuadrantRenderer is a real CPU killer. Try to avoid using it for depth sorting in your applications. As was previously shown, `SpriteRenderSession` or `pushfront/pushback` is the best alternative.

8

Prefab3D

In this chapter, we will cover:

- ▶ Exporting models from Prefab
- ▶ Normal mapping with Prefab
- ▶ Maintaining workflow with AwayConnector
- ▶ UV map editing with Prefab
- ▶ Creating terrain
- ▶ Generating light maps
- ▶ Creating and animating paths

Introduction

Prefab is a desktop program based on Adobe AIR and developed by Fabrice Closier (Away3D development team member) to speed up different aspects of Away3D development with the help of an interactive visual interface. Although far more primitive Prefab conceptually resembles one of the 3D modeling packages such as 3DsMax or Blender. Having a viewport and packed with a robust set of tools for creating, editing, mapping, exporting, and much more, this piece of software should become your close friend when developing Away3D powered projects.

In this chapter, our goal is to learn how to perform most important operations in Prefab, which will help you to save time for your project. One thing to note is that since Prefab is constantly updated and new features are added almost on a monthly basis, it is possible that the reader will find some differences related to primary software's UI. Nevertheless, the core features and their implementation pipeline are basically the same as they are described in this chapter.

Exporting models from Prefab

Prefab features several different exporters. You can export Wavefront .obj (including the mtl file), the ActionScript file which represents a mesh object as well as the additional Away3D native format called AWD (Away data). Additionally, there is an option to export to AC3D as a format which is still not integrated inside Away3D. In this recipe, you will be shown how to export your models from Prefab to the ActionScript and AWD formats.

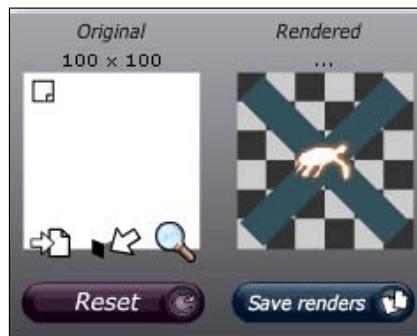
Getting ready

Make sure you have the latest version of Prefab installed on your machine. You can download the program from <http://www.closier.nl/prefab/>.

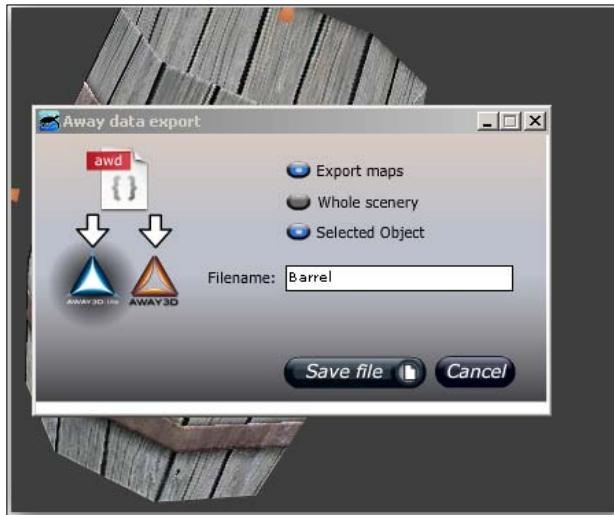
In the following couple of recipes, we make use of a low poly model of a wooden barrel. You can find the model, which is in 3ds format, inside the assets/models folder in the source directory of this chapter. Alternatively, you may use your own model if you wish.

How to do it...

1. Open Prefab.
2. Open the folder containing the file barrel_empty_low.3ds. Drag-and-drop the model into the Prefab viewport.
3. In some cases after the model import, you may receive a warning that the model is too small and Prefab would like to scale it up. That is not a bug; many models because of unit differences of their modeling programs become very small or too large in Away3D. So, in this case, just scale up the model manually so that it has decent dimensions in the viewport without the need to zoom the camera to the maximum. Now let's assign a texture to our model. In Prefab on the sidebar, click on the **Load and set external map** button located at the bottom to the right.



4. In the file explorer screenshot that opened, go to the assets/models/images folder of the source directory for this chapter and select barrel96.png.
5. Now, let's export the model to AWD.
6. In Prefab, click on the **Export** button at the top toolbar. In the drop-down list, select **Export data only (awd)**. The following screenshot should have opened:



7. Note that the ID you select is the **Export maps** option. Prefab will generate a folder called images in the same directory where you save your awd file with all the relevant maps enclosed.
8. Name the file **Barrel** and click on the **Save file** button.

The following is the test-drive program for the exported model. Here we load our model which is, by now, in the awd format into our simple Away3D scene. Before running this, make sure you put the exported Barrel.awd and enclosed images folder inside the assets directory of your ActionScript project:

AWDataLoadingDemo.as

```
package
{
    public class AWDataLoadingDemo extends AwayTemplate
    {
        [Embed(source="assets/models/images/barrel96.png")]
        private var BarrelTexture:Class;

        private var _obj3D:Object3D;
        private var _bitMat:BitmapMaterial;
        public function AWDataLoadingDemo()
```

```
{  
    super();  
}  
override protected function initMaterials() : void{  
    _bitMat=new BitmapMaterial(Cast.bitmap(new  
BarrelTexture()));  
}  
override protected function initGeometry() : void{  
    loadAWD();  
}  
override protected function onEnterFrame(e:Event) : void{  
    super.onEnterFrame(e);  
    if(_obj3D){  
        _obj3D.rotationX++;  
        _obj3D.rotationY++;  
    }  
}  
private function loadAWD():void{  
    var loader:Loader3D=AWData.load("assets/models/Barrel.awd");  
    loader.addOnSuccess(onLoadComplete);  
}  
private function onLoadComplete(e:Loader3DEvent):void{  
    _obj3D=e.loader.handle as Object3D;  
    _view.scene.addChild(_obj3D);  
    _obj3D.z=500;  
}  
}
```

How it works...

In fact Prefab makes use of Away3D exporters, which can be accessed from the library inside the `away3d/exporters` package and used manually inside the Away3D project.

You might surely ask the following question: "Why on earth would I want to import a model into Prefab and then export it back?" Well here are a few reasons:

- ▶ Away3d data format (awd) is highly optimized and parsed relatively fast so you are encouraged to use it for loading external models.
- ▶ ActionScript is good for the same reason (just doesn't support animation) and even better than that, it is not to be parsed, as it already contains complete Away3D code for the specific model.

- ▶ Also Prefab has got a decent toolbox for models optimization. You can use options such as "Weld" and "Remove double faces" to remove useless or disconnected vertices and faces from your model and achieve, in some cases, far better performance with the optimized version.

There's more...

Exporting to ActionScript is as easy as the previous example.

We are going to use the same source file `barrel_empty_low.3ds`. Load it into the Prefab in the same way as you have done in the previous example. Assign the texture to the model.

In the top toolbar, click **Export** and in the drop-down list, select **Export to Away3d AS3 Class**. The following screenshot should have opened:



If you use the latest Away3D FP10 build-in exporting options, select **Away3D F10 3.6+**; otherwise choose one of the two preceding options.

At the right-side options column, select **Embed originals** as we are going to embed the models texture right into its `as3` file. This way, when instantiating it, the texture will be applied automatically.

Now let's test the generated model. We use the previous example code to which we should make a little clean up. Of course, you can set a brand new class if you wish. Remove or comment out all the methods except `initGeomtry()` and `onEnterFrame()`. Inside the `initGeomtry()` method, create an instance of the `Barrel` class and add it to the scene. Inside `onEnterFrame()`, we added to it as well X and Y rotations, as is shown in the following code:

```
AWDataLoadingDemo.as

package
{
    public class AWDataLoadingDemo extends AwayTemplate
```

```
{  
    private var _barrel:Barrel;  
    public function AWDataLoadingDemo()  
    {  
        super();  
    }  
    override protected function initGeometry() : void{  
        _barrel=new Barrel();  
        _view.scene.addChild(_barrel);  
        _barrel.z=400;  
    }  
    override protected function onEnterFrame(e:Event) : void{  
        super.onEnterFrame(e);  
        if(_barrel){  
            _barrel.rotationX++;  
            _barrel.rotationY++;  
        }  
    }  
}
```

Now you can clearly see the benefits of using `as3` export. Just two lines of code to set up your model!

See also

- ▶ In *Chapter 9, Working with External Assets*, the following recipe:
 - *Exporting models from 3dsMax/Maya/Blender*
 - *Exporting models from 3dsMax to ActionScript class*
 - *Preparing MD2 models for Away3D (MilkShape)*

Normal mapping with Prefab

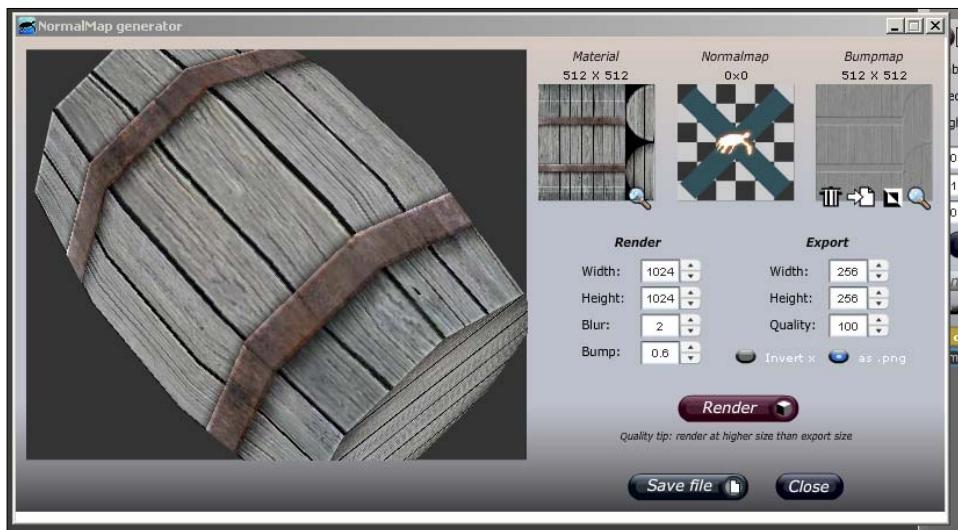
Prefab allows you to generate normal maps visually with a built-in Normal Map generator interface in no time and then assign them immediately to your model. (For more explanation on normal maps, read in *Chapter 1, Working with Away3D Materials*, the following recipe *Generating normal maps using Away3D NormalMapGenerator*). Let's see how to do it.

Getting ready

Repeat the *Getting ready* steps from the previous recipe.

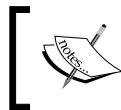
How to do it...

1. Open Prefab.
2. Open the folder containing the file `barrel_empty_low.3ds`. Drag-and-drop the model into the Prefab viewport.
3. Now assign a texture to our model. In Prefab in the sidebar, click the **Load and set external map** button located at the bottom to the right. In the file explorer dialog that opened, go to the `assets/models/images` folder of the source directory for this chapter and select the `barrel96.png` bitmap.
4. In the Prefab top toolbar, click on **Render**, then in the drop-down list, select **Render Normalmap**. The following screenshot should have opened:



5. Inside the **NormalMap** generator dialog, you can already see the default texture map that is going to serve as a source for the normal map. If we render the normal map in this state, we will not gain much as it would lack all those cracked details that you can see at the wooden surface of the barrel. In order to add that data to the normal map, we should incorporate a bump map as well within the rendering process. Usually, bump maps are grayscale bitmaps. However, in this particular example, I am going to use a normal map as a bump map input which was generated inside 3DsMax.

- In the **NormalMap** generator dialog below the **Bumpmap** window, click on the **Load external bumpmap** button and navigate to this chapter's assets/models/images/barrel1_512_NRM.jpg file and open it.



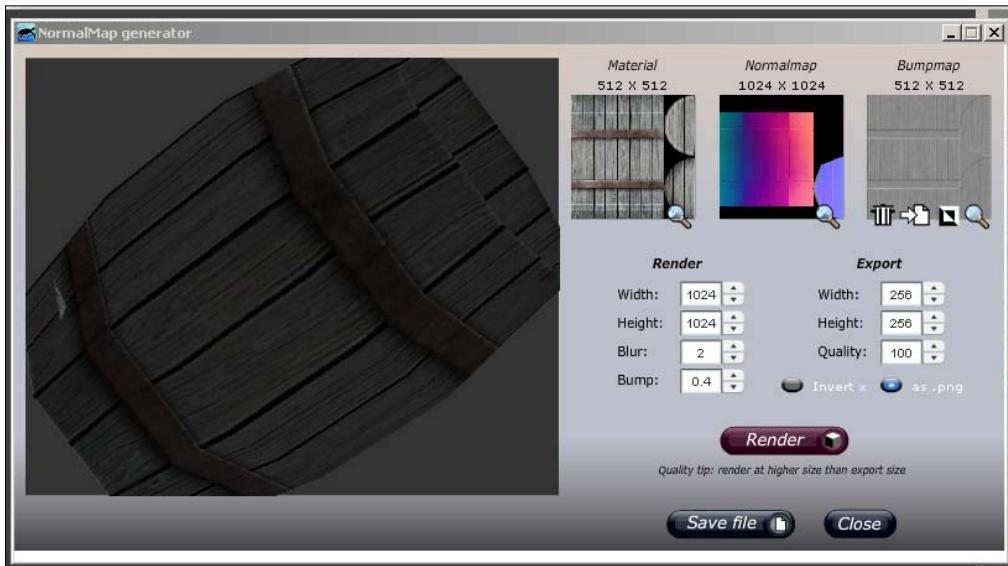
When generating a normal map in Prefab, make sure that the rendered bitmap is of the same size as the diffuse map (render map).

- Concerning the rest of the settings, there is one important thing to know; in order to get a higher quality normal map, it is recommended to define the render map dimensions to be much bigger than those of the resulting (export) bitmap. In our example, we set up the render map size to 1024x1024 whereas the export map is set to 256x256.



Be careful with bump value. For most of the cases, the default 0.4 is enough for nice detail. Setting it to high may result in visual artifacts on top of the texture with the normal map applied.

- Click on the **Render** button. When the rendering finishes, you should get a similar result to the one shown in the following screenshot:



- Now click the **Save file** button and save the normal map file under the name barrel196_nm.jpg to the images folder inside your project.
- Basically, we are done. Let's run a quick test to see the normal map in a simple Away3D scene. Set up a basic Away3D scene using AwayTemplate.

11. Make sure you have the barrel96.png and barrel96_nm.jpg images inside the assets/models/images folder of your ActionScript project.

12. Copy the following code into your file:

```
package
{
    public class PreFabNMDemo extends AwayTemplate
    {
        [Embed(source="assets/models/images/barrel96.png")]
        private var BarrelTexture:Class;
        [Embed(source="assets/models/images/barrel96_nm.jpg")]
        private var BarrelNMap:Class;

        private static const DIST:Number=200;
        private var _angle:Number=0;
        private static const DEGR_TO_RADS:Number=Math.PI/180;
        private var _pointLight:PointLight3D;
        private var _barrel:Barrel;
        private var _phongPB:PhongPBMaterial;
        public function PreFabNMDemo()
        {
            super();
            _cam.z=-100;
            initSceneLights();
        }
        override protected function initGeometry() : void{
            _barrel=new Barrel();
            _barrel.bothsides=false;
            _view.scene.addChild(_barrel);
            _phongPB=new PhongPBMaterial(Cast.bitmap(new
BarrelTexture()),Cast.bitmap(new BarrelNMap()),_barrel.meshes[0]
as Mesh);
            _barrel.meshes[0].material=_phongPB;
            _barrel.z=200;
            _barrel.rotationX=90;
        }
        override protected function onEnterFrame(e:Event) : void{
            super.onEnterFrame(e);
            if(_pointLight){
                _angle++;
                _pointLight.x = _barrel.position.x+ Math.sin(_angle*DEGR_TO_
RADS)*DIST;
                _pointLight.y = 0;
            }
        }
    }
}
```

Prefab3D

```
        _pointLight.z = _barrel.position.z+Math.cos(_angle*DEGR_TO_
RADS)*DIST;
    }
}
private function initSceneLights():void{
    _pointLight=new PointLight3D();
    _view.scene.addLight(_pointLight);
    _pointLight.radius=60;
    _pointLight.brightness=1.4;

}
}
}
```

Run the application. Here is the result you should get. It has a nice specular reflection with pretty realistically looking wood texture detail. Thanks to normal mapping.



How it works...

Prefab generates a normal map using the utility called `NormalMapGenerator` that is found inside the `away3d.materials.utils` package of the Away3D library. The advantage, as you should have already understood, is that in Prefab, you can see the resulting effect of the generated normal map immediately feature that can speed up your development workflow significantly.

Now let's see how the testing program works. In order to see whether the normal map works, we need to set up two things:

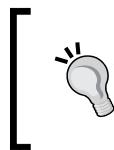
- ▶ Scene lights
- ▶ Material that supports normal mapping

We set up a light of a type `PointLight3D` so that we can spin it around our model and see the effect of normal map in action. We set the light inside the `initSceneLights()` method:

```
private function initSceneLights():void{
    _pointLight=new PointLight3D();
    _view.scene.addLight(_pointLight);
    _pointLight.radius=60;
    _pointLight.brightness=1.4;

}
```

For the material that supports normal maps, I chose `PhongPBMaterial` that makes use of the `PixelBender` shader, supplying much more sophisticated rendering quality than the materials from the `Dot3` family.



When you select a material from the **PB (PixelBender)** group, make sure you check what type of light each of them requires. It may be `PointLight3D` or `DirectionalLight3D`. Multipass materials use both. In our case, `PhongPBMaterial` works with `Pointlight3D`.

For `PhongPBMaterial`, we put constructor arguments, the default barrel texture bitmap, the normal map that we generated inside Prefab, and the reference to the object to which the material is applied:

```
_phongPB=new PhongPBMaterial(Cast.bitmap(new BarrelTexture()),Cast.bitmap(new BarrelNMap()),_barrel.meshes[0] as Mesh);
_barrel.meshes[0].material=_phongPB;
```

That is all folks.

See also

In *Chapter 1, Working with Away3D Materials*, the following recipes:

- Creating normal maps in Photoshop*
- Generating normal maps using Away3D NormalMapGenerator*

Maintaining workflow with AwayConnector

In this recipe, you learn what `AwayConnector` is and how to use it. `AwayConnector` enables you to export geometry directly from an `Away3D` scene to `Prefab` in runtime. This feature is useful if you need, for example, to edit UV data or clean up a mesh of a model in your scene that is not an external asset like one of `Away3D`'s generic primitives.

Getting ready

Repeat the *Getting ready* steps from the previous recipe.

Also set up a basic `Away3D` scene which extends `AwayTemplate` and give it a name `AwayConnectDemo`.

How to do it...

1. Launch `Prefab`.
2. First we need to get the `AwayConnector.as` class into the `Away3D` library because, as at the time of writing, it is not a part of it. We can get it from `Prefab`. The program generates the class for you on demand. In `Prefab`, click on the **File** button at the top toolbar and select the **Import from Away connection** option. The following window should have opened:



3. In the **LocalConnection feed** window, press the **Connector class** button, and you will be prompted to save the `AwayConnector.as` file. Navigate to the `Away3D` source and put the class inside the `away3d.materials.utils.data` package.
4. Don't close the **LocalConnection feed** window as it is listening for the incoming communication attempt from your program.
5. Next, inside your `AwayConnectDemo` class, add the following code:

```
AwayConnectDemo.as
package
{
```

```

public class AwayConnectDemo extends AwayTemplate
{
    public function AwayConnectDemo()
    {
        super();
    }
    override protected function initGeometry() : void{
        var cone:Cone=new Cone({height:200, radius:60});
        _view.scene.addChild(cone); //not must
        var ac:AwayConnector=new AwayConnector();
        ac.send(cone);
    }
}

```

6. Run the application. Then go back to Prefab and you will see the Cone primitive that we have just instantiated in our Away3D scene in the runtime showing in the viewport.

How it works...

This trick looks cool doesn't it? Well, in fact, the mechanism behind it is nothing but a generic ActionScript3.0 LocalConnection object instance that connects with Prefab and transfers to it the model in an awd format. Nothing magical really.



Note that the Prefab accepts only the mesh from AwayConnector, the material applied to the model inside Away3D is discarded.

UV map editing with Prefab

UV mapping is one of the coolest features Prefab provides. Prefab has quite an advanced interactive interface where you can edit the UV coordinates mapping of your geometry. If you work with the UVWUnwrap modifier in 3DsMax, you will find your way through very quickly. In this recipe, we are going to export a cone primitive from an Away3D scene via AwayConnector into Prefab and then edit its UV data.

Getting ready

1. Make sure you have the latest version of Prefab installed on your machine. You can download the program from <http://www.closier.nl/prefab/>.
2. Make sure you copy a bitmap called simpleUVMap.png from this chapter's assets/models/images folder into your project assets directory.

-
3. If this is the first time you are working with `AwayConnector`, please refer to the recipe *Maintaining workflow with AwayConnector* to learn how to set it up.
 4. Set up a basic Away3D scene extending `AwayTemplate` and give the file a name, `PrefabUVMapDemo.as`.

How to do it...

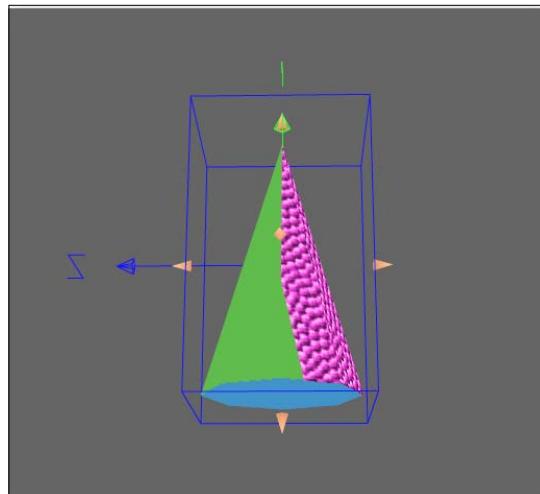
1. Launch Prefab. In the Top toolbar, click **File** and then in the opened drop-down list, select **Import from Away connection**.
2. Now, inside your Flash application, add the following code and run it:

```
PrefabUVMapDemo.as
package
{
    public class PrefabUVMapDemo extends AwayTemplate
    {
        [Embed(source="assets/models/images/simpleUVMap.png")]
        private var ConeMap:Class;
        private var _cone:Cone;
        public function PrefabUVMapDemo()
        {
            super();

        }
        override protected function initGeometry() : void{
            _cone=new Cone({height:200,radius:60});
            _view.scene.addChild(_cone);
            _cone.z=500;
            _cone.x=100;
            var ac:AwayConnector=new AwayConnector();
            ac.send(_cone);

        }
    }
}
```

3. Switch back to Prefab and you should see the cone primitive inside the viewport. Now, at the right panel below the object texture icon, press the **Load and set external map** button, navigate to the `simpleUVMap.png` bitmap that you got before and open it. Now you should see your cone wrapped with a material, but obviously with the wrong UV mapping, as you can see in the following screenshot:



4. In Prefab, select the cone and then click on the **UV Editor** button located in the bottom toolbar. The following window should have opened:



5. We are not going to explain each button of the UV editor. The usage is quite intuitive. Just select the polygons in the right window and then you can manipulate them whether manually by face or vertex in the left window or by using control panel options located in the middle.

6. Here is the result of the remapping we have done for the cone:



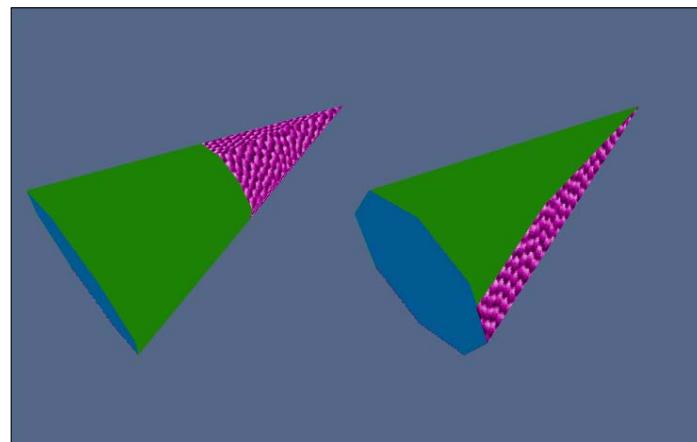
We manually moved the side faces of the cone so that half of them overlap the green area of the map and another half the purple bumpy region. At the right window of the preceding screenshot, you can see the resulting mapping of the cone.

7. When you are done with the mapping job, click the **Apply** button.
8. Now export the cone to ActionScript3.0 object and name it `UVMappedCone.as` and specify "as3models" as the package name. Put it in your Flash project "as3models" package.
9. Here is the test drive for the newly remapped cone. We just add a `UVMappedCone` class instantiation to the existing code of the preceding example:

```
PreafabUVMapDemo.as
```

```
package
{
    public class PreafabUVMapDemo extends AwayTemplate
    {
        [Embed(source="assets/models/images/simpleUVMap.png")]
        private var ConeMap:Class;
        private var _cone:Cone;
        private var _bitMat:BitmapMaterial;
        private var _mappedCone:UVMappedCone;
        public function PreafabUVMapDemo()
        {
            super();
        }
    }
}
```

```
override protected function initMaterials() : void{
    _bitMat=new BitmapMaterial(Cast.bitmap(new ConeMap()));
}
override protected function initGeometry() : void{
    _cone=new Cone({height:200, radius:60, material:_bitMat});
    _view.scene.addChild(_cone);
    _cone.z=500;
    _cone.x=100;
    initUVMappedCone();
}
override protected function onEnterFrame(e:Event) : void{
    super.onEnterFrame(e);
    _cone.rotationY=_mappedCone.rotationY+=1;
    _cone.rotationX=_mappedCone.rotationX=-.5;
}
private function initUVMappedCone():void{
    _mappedCone=new UVMappedCone();
    _mappedCone.material=_bitMat;
    _view.scene.addChild(_mappedCone);
    _mappedCone.z=500;
    _mappedCone.x=-100;
}
```



From the left, you can see the remapped cone and from the right is the default one.

How it works...

UV is a vertex property. When you move, rotate, or scale the texture, you actually change the UV property of all the vertices that are involved in the movements.

Then the Prefab exports the new model data with the change in the UV coordinate.

Creating terrain

If you develop a game of the first/third person shooter or quest type, it is likely that you will need to create an outdoor terrain so that your characters could roam in it. Prefab includes a Terrain Generator editor which allows you to create a terrain surface in a matter of seconds. This recipe explains how to use it.

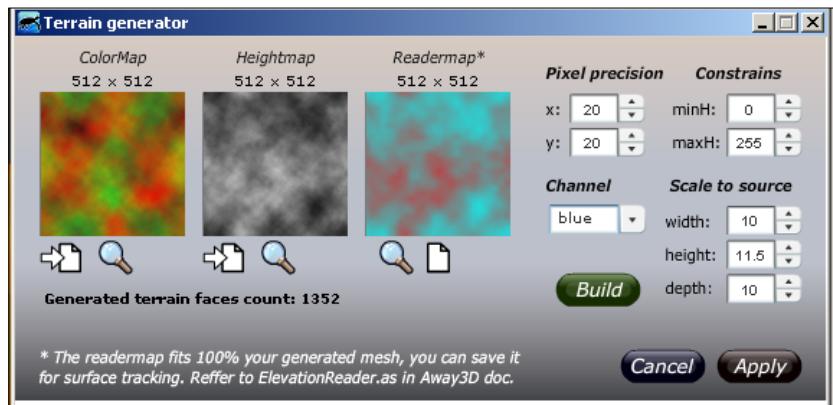
Getting ready

Make sure you have the latest version of Prefab installed on your machine. You can download the program from <http://www.closier.nl/prefab/>.

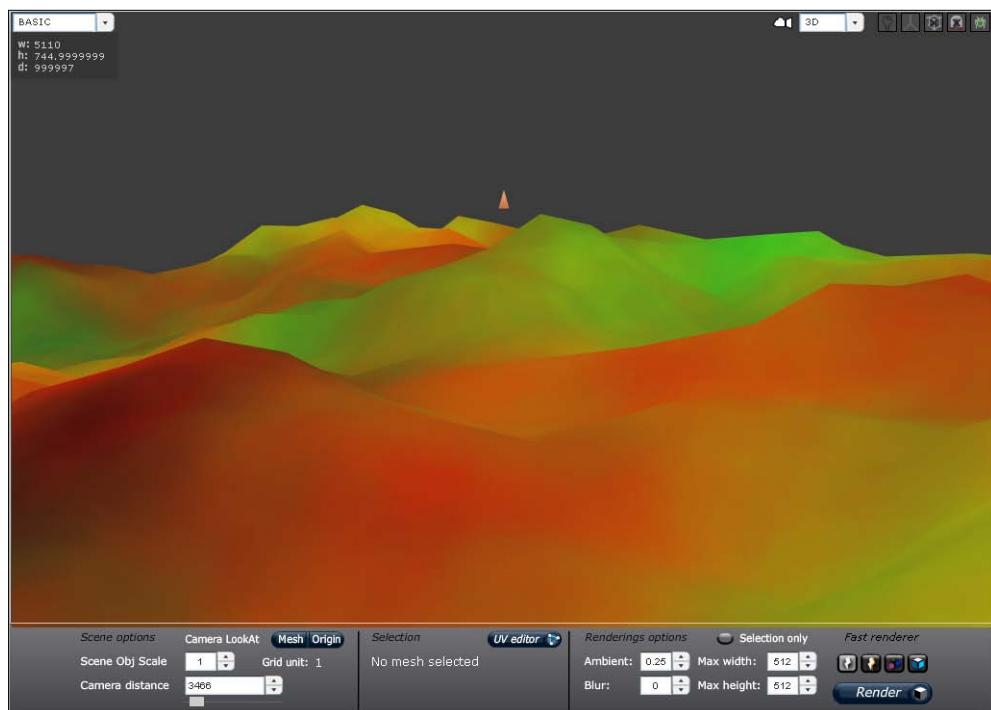
In order to generate a terrain, we also need a height map. Go to this chapter's assets folder and copy HeightMap.png into your project assets directory. Alternatively, you can use Photoshop to generate a quick height map for mountainous topography using the Clouds filter. Also paste from the same folder the bitmap named ColorMap.png, which will serve us as a basic material texture for the generated terrain.

How to do it...

1. Open Prefab.
2. In the top toolbar, click **Geometry** then in the drop-down list, select **Terrain generator**. The following window should have opened up:



3. As you can see from the preceding screenshot, the **Terrain generator** window has got three image slots. We have to assign the bitmaps for the **Heightmap** slot only. The **ColorMap** is optional because you can assign a texture to the generated mesh later. **Readermap** is generated automatically when we build the terrain used for the **ElevationReader** class which we will learn how to use in *Chapter 13, Doing more with Away3D*. Now assign to the ColorMap our ColorMap.png and to the HeightMap slot HeightMap.png. If you want to constrain the minimum and maximum height of the terrain, use the "Constrains" fields which have a range of a color channel that is from 0 to 255.
4. Click on the **Build** button to generate the terrain. Look into the viewport, you should see the terrain appear, then press the **Apply** button to exit the generator window.



5. Export the terrain to the ActionScript3.0 file. Make sure you define the export setting, as shown in the next screenshot. We checked the **Embed originals** options so that all the bitmaps would be embedded in the exported file:



6. Copy `Terrain.as` into your project into the `as3models` folder.

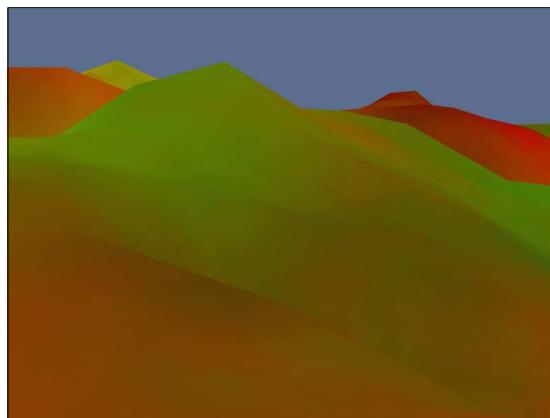
Set up an Away3D scene with `AwayTemplate` as a super class and give it a name `PrefabTerrainDemo.as`. Following is the code which instantiates the terrain we have just created in Prefab:

```
PrefabTerrainDemo.as

package
{
    public class PrefabTerrainDemo extends AwayTemplate
    {
        private var _terrain:Terrain;
        private var _hoverCam:HoverCamera3D;
        private var _oldMouseX:Number=0;
        private var _oldMouseY:Number=0;
        private var _canMove:Boolean=true;
        public function PrefabTerrainDemo()
        {
            super();
            setHowerCamera();
        }
        override protected function initGeometry() : void{
            _terrain=new Terrain();
            _view.scene.addChild(_terrain);
        }
        override protected function initListeners() : void{
```

```
super.initListeners();
stage.addEventListener(MouseEvent.MOUSE_DOWN,
onMouseDown,false,0,true);
stage.addEventListener(MouseEvent.MOUSE_UP,
onMouseUp,false,0,true);
}
override protected function onEnterFrame(e:Event) : void{
super.onEnterFrame(e);
if(_hoverCam){
_hoverCam.hover();
if(_canMove){
_hoverCam.panAngle = (stage.mouseX - _oldMouseX);
_hoverCam.tiltAngle = (stage.mouseY - _oldMouseY);
}
}
private function setHoverCamera():void{
_hoverCam=new HoverCamera3D();
_view.camera=_hoverCam;
_hoverCam.target=_terrain;
_hoverCam.distance = 1600;
_hoverCam.maxTiltAngle = 15;
_hoverCam.minTiltAngle = 5;
_hoverCam.steps=12;
_hoverCam.yfactor=1;
}
private function onMouseDown(e:MouseEvent):void{
_canMove=true;
}
private function onMouseUp(e:MouseEvent):void{
_oldMouseX=stage.mouseX;
_oldMouseY=stage.mouseY;
_canMove=false;
}
}
```

Here is the result:



How it works...

The Terrain Generator implements the `HeightMapModifier` class that is located inside the `away3d.modifiers` package. That means that you can also generate a terrain manually using this modifier in Away3D.

See also

Away3D's `ElevationModifier`, located in `away3d.extrusion` package, is another utility to generate terrain-like mesh extrusions.

Generating light maps

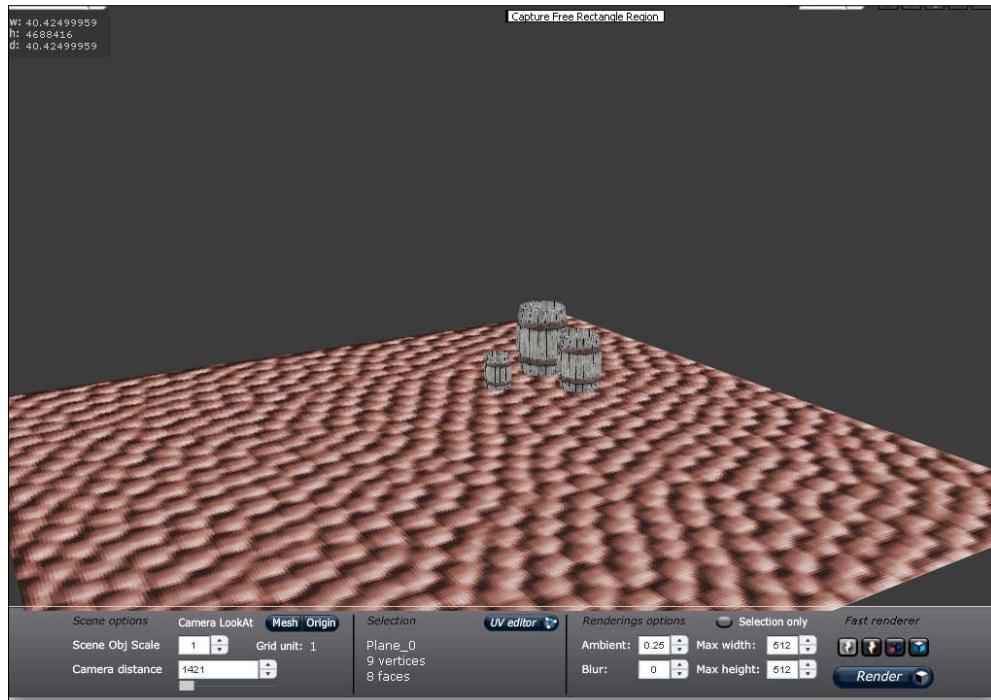
Light map is a widely known technique in the 3D gaming industry. With the help of light maps, we can bake scene lights information such as shadows and illuminated areas right into the texture of a 3D object. Then you can replace completely or reduce to a minimum, the dynamic lighting in the scene with the light maps (what is actually done in many 3D games today), thus saving a lot of precious CPU cycles. Usually, in order to make a light map, a 3D package such as 3DsMax or Maya is used with built-in light map baking utilities. Not always generating a light map in these programs is easy and rapid especially for beginners. But here's some good news; Prefab allows you to bake light maps with no pain and all that in a matter of minutes. So let's go and bake some light!

Getting ready

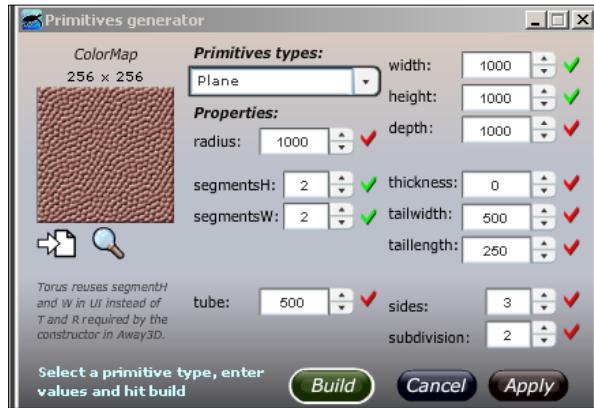
1. Make sure you have the latest version of Prefab installed on your machine. You can download the program from <http://www.closier.nl/prefab/>.
2. In the following couple of recipes, we make use of a low poly model of a wooden barrel. You can find the model, which is in 3ds format, inside the assets/models folder in the source directory of this chapter. Alternatively, you may use your own model if you wish.
3. Also set up a basic Away3D scene which extends AwayTemplate and gives it a name LightMapDemo.as.

How to do it...

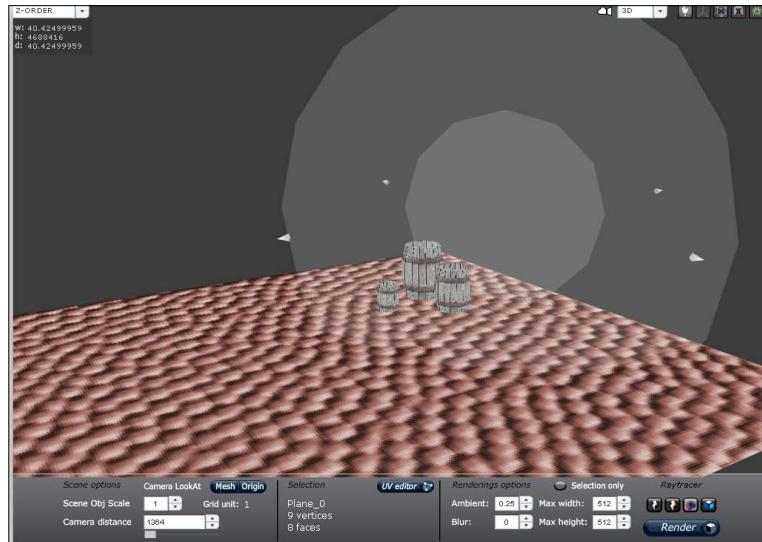
1. Open Prefab.
2. Open the folder containing the file barrel_empty_low.3ds. Drag-and-drop the model into the Prefab viewport.
3. Duplicate two more instances of the barrel by using the **Ctrl + C/Ctrl + V** keys.



4. Position them as shown in the previous screenshot. We will create the floor right inside Prefab using the feature called **Primitives generator**. In the top toolbar, go to **Geometry** and select **Primitives generator**. The following window should have opened:



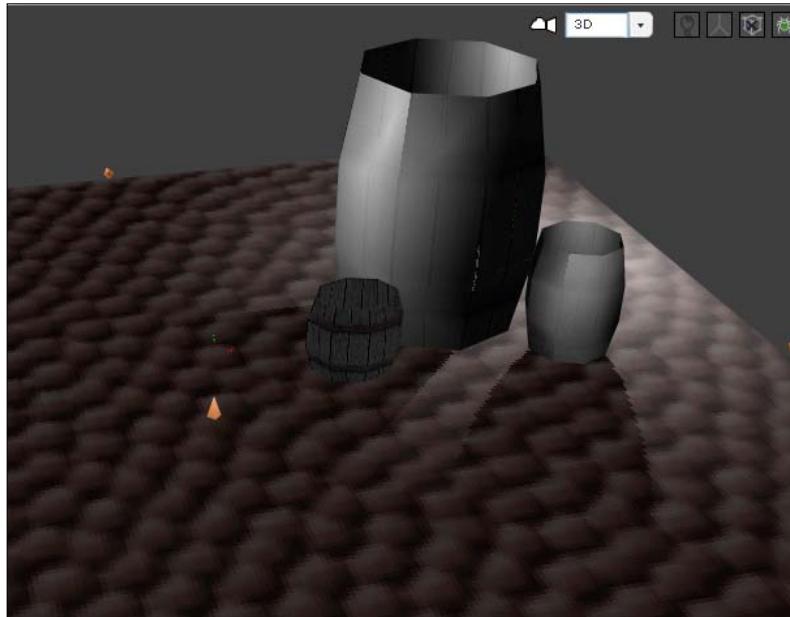
5. Make sure you select a **Plane** primitive from the **Primitive types** drop-down list. Inside the **ColorMap** slot, click on the **Load external map** button and navigate to this chapter's assets/models/images folder. Select **SimpleBitmap1.png** and open it.
6. In the **Primitives generator** window, click **Build** and then the **Apply** button.
7. Adjust the position of the plane on the y-axis so that there is no space left between it and the bottoms of the barrels (see the previous viewport snapshot).
8. Next we need to set up a light source which will then be rendered using Ray Tracing Renderer. Move the point light instance, which, by default, should already exist in the viewport, to the position as is depicted in the following screenshot:



The light source is located a bit behind the barrels with its radius around 350 units as we want to imprint the shadows of the barrels into the floor surface.

 Make sure that the visible area of the light overlaps the barrels objects, otherwise there would be no shadows produced and the unlit models will be unaffected during the rendering.

9. Now when our scene setup is ready, we can go and render the light map. In the bottom toolbar, you can see four possible rendering modes:
 - The first two are fast renderers which render fast but cost you a reduced quality of baked data
 - Third is a raytracing which is far more sophisticated type of rendering, consuming more time but giving a smoother result
 - The fourth type is used to render cube maps for environment shading, which we are not going to discuss here
10. Select **Ray Tracing** mode. Then press the **Render** button and wait till the tracing is completed. Here is the rendered result:



11. Now let's export the scene to the ActionScript3.0 file embedding the rendered light maps as geometry textures. In the top toolbar, click **Export** then press the **Export to Away3d AS3** class. These are the export settings you should define:



Notice, we selected the **Embed renderings** option in order to embed in the class the rendered light maps. Additionally, you should check the **Whole scenery** option in order to embed the floor plane and three barrels to the **as3** class.

12. Let's test the exported scene in Away3D application. Open the `LightMapDemo.as` file which we created in the beginning and paste the following code into it:

```
LightMapDemo.as
package
{
    public class LightMapDemo extends AwayTemplate
    {
        private var _bakedLight:BakedLight;
        public function LightMapDemo()
        {
            super();
            _cam.y=300;
            _cam.lookAt (_bakedLight.position,Vector3D.Y_AXIS);
        }
        override protected function initGeometry() : void{
            _bakedLight=new BakedLight();
            _view.scene.addChild(_bakedLight);
            _bakedLight.z=1000;
            _bakedLight.rotationY=15;
        }
    }
}
```

There's more...

As you can see from the screenshots, the quality of the baked light maps created in Prefab is not perfect. Still, the best approach to create top quality light maps is to use external modeling software such as Autodesk 3dsMax, Maya, or Blender.

You can see online tutorials on how to render lights to texture in 3D packages.

Creating and animating paths

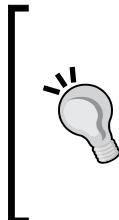
Prefab contains a set of tools to create, edit, extrude, and even animate paths. Those of you planning to create a racing game can rejoice as it is extremely easy to build a clean and nice looking road loop interactively using the visual path editor and then set an animation to a camera or any other object along that path. Then all that is left is to export it to AS3, add a few more lines in your Flash project, and you are halfway to your very own car-racing game. Let's see how to do all that in Prefab.

Getting ready

1. Make sure you have the latest version of Prefab installed on your machine. You can download the program from <http://www.closier.nl/prefab/>.
2. You will need to copy to your project images folder a bitmap called `roadSpan.png`, which is found in this chapter's `assets/models/images` directory and will serve us as a texture for an extruded path.
3. Also set up a basic Away3D scene using `AwayTemplate` and name it `pathDemo`.

How to do it...

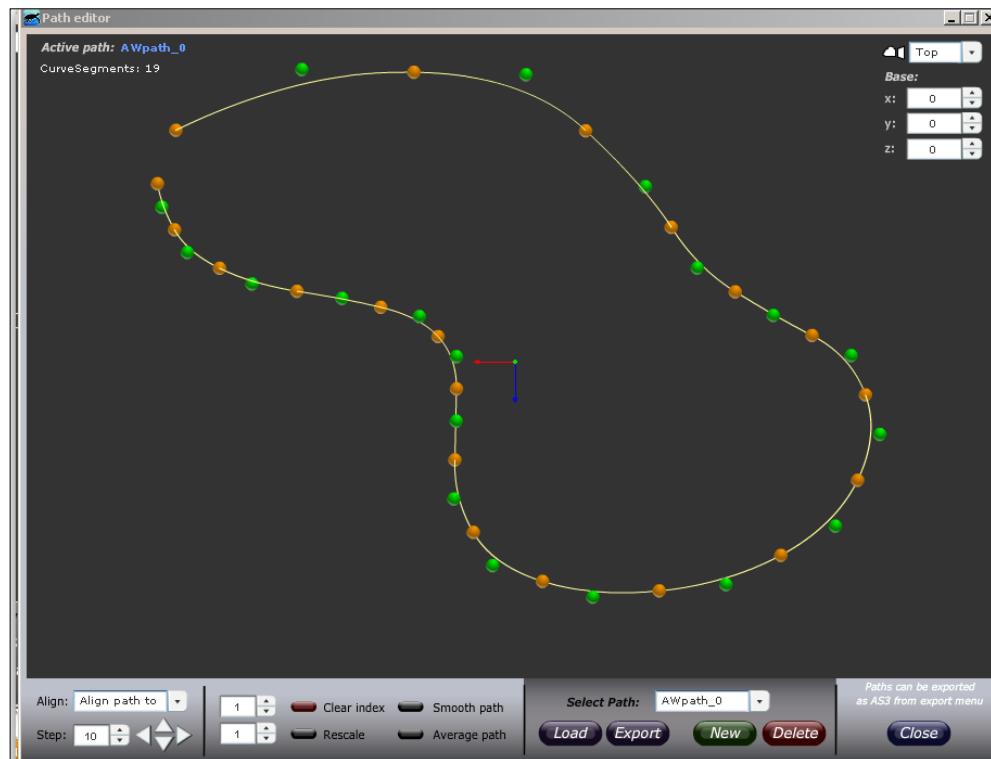
1. Open Prefab.



It is recommended to save your ready path to a file using the **Export** button in the Path editor window. This allows you to use the same path many times, importing it into the editor.

If you did not save your path, no problem, you can find one in this chapter's `assets` folder named `AWpath_0.awp`. We are going to make use of it in the second part of this recipe.

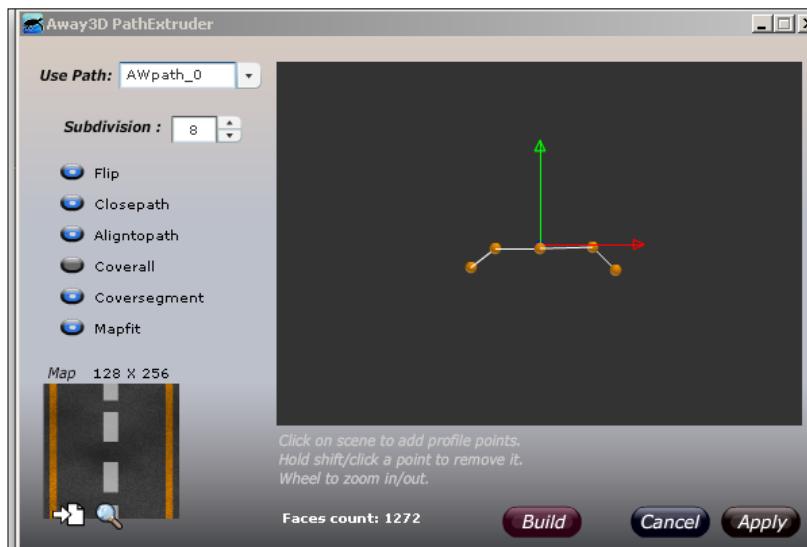
2. In the top toolbar, click **Geometry** then in the Path category, select **Path Editor**. The following window should have opened:



At the bottom of the window, click the **New** button and start putting points within the viewport of the editor till you achieve a similar shape, as shown in the previous screenshot. After you finish adding the points, click on the **Smooth path** button in order to smooth the curves.

3. When you are done, click on the **Close** button.

4. Next we want to extrude it so that it will look like a racing road. In Prefab, click **Geometry** then select **Path extruder**. You should see the following window open:



First we need to define the profile of extrusion. In order to understand what that means, just imagine a road span in front of a view cut off, so the outline of it is a profile cut. You define it the same way as you did with the path points in the Path editor. Notice that the centre of the profile path is a green (y-axis) arrow.

5. When you have finished with the profile, assign a `roadSpan.png` image to the map slot, then select the following options above it: **Flip**, **Closepath**, **Alignopath**, **Coversegment**, and **Mapfit**. It is important to select **Alignopath** so that the map details would align to the geometry curves of the road. **Coversegment** allows us to tile our map. Use this option with seamless textures; otherwise you will see the tiling artifacts. **Mapfit** adjusts the map to the geometry dimensions.
6. Click **Build** and then the **Apply** button. You should see a similar model in your viewport:



7. Now let's export it to the ActionScript 3.0 object. In the top toolbar, click **Export** then **Export to Away3D AS3 class**. Here are the settings we need to define in the exporter screenshot:



8. Save the file to your `as3models` package inside your Flash project. Make sure you copy the `aw_1.png` that is located inside the enclosed images folder to the similar folder that should reside inside the `as3models` package. The name of the file in this chapter's attached project is `aw_11.png` as `aw_1.png` and is occupied by another image. Make sure you rename it if you have got such a file already in your project.
9. Let's test our road inside `PathDemo.as`. We set it up with just four lines of code. Here it is:

```
package
{
    public class PathDemo extends AwayTemplate
    {
        private var _road:RacingField;
        public function PathDemo()
        {
            super();
        }
        override protected function initGeometry() : void{
            _road=new RacingField();
            _view.scene.addChild(_road);
            _cam.y=25000;
            _cam.lookAt(_road.position);

        }
    }
}
```

How it works...

Prefab really makes it easy to create sophisticated paths and their extrusion supplying us with visual editors. If you were to do it manually inside Flex, you would quickly understand what a nightmare this work is. Each point of the path is a Vector3D. So, unless you imagine the surroundings of your world daily as a set of numbered points, such a routine can make you crazy when working on complex paths. The same applies to path extrusion as the Away3D class PathExtrusion also accepts a set of Vector3Ds for the profile definition.

As you can see from this example, Prefab can make your life easier in many aspects, saving you precious time which you may dedicate to other tasks.

There's more...

Besides building and extruding the path, Prefab enables us to animate any object along it.

Let's see how to do it:

1. If you closed Prefab after finishing the previous example, open it again and repeat all the steps you had learned previously to create and extrude the path.



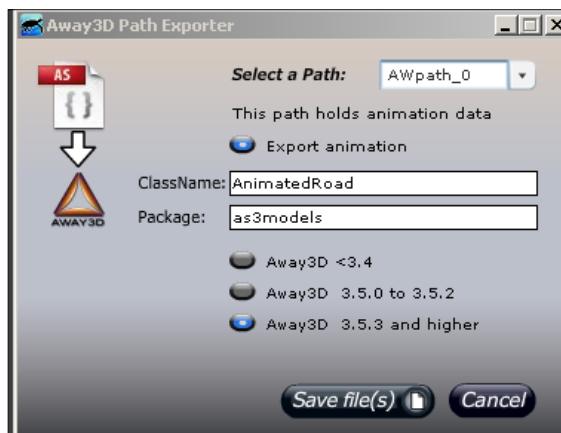
Use the AWpath_0.awp file (located in the assets.models folder of the source code directory for this chapter) to load a prebuilt path into the path editor.

2. Select the extruded path, in the top menu, click **Geometry | Path | PathAnimator**. The path animation settings dialog should open.
3. In the **Use Path** combo, select your already existing path. Then click on the **New** button to create a new animation.
4. The animator allows you to animate any object extending Object3D. In our case, we will use the camera object. So in the dialog, check **Use Camera**, give it an offset on the y-axis to around 50-100 pixels so that it would hover at some distance from the road.
5. Check **align to path** as we want the camera looking at the path direction while being animated.
6. Next, set animation time to be expressed in milliseconds. Setting a number in the range of 5000-10000 will be enough for this particular path.

7. Use the **preview** button to test your animation:



8. Now it is time to export the animation for testing in Away3D. In the top menu bar, click **Export | Export AS3 class | Path to Away3D**.
9. In the Path exporter dialog, select our path from the combobox. Check the **select animation** option. The rest of the settings should be like the one shown in the following screenshot:



10. Hit **Save files** and we can now proceed to Away3D.

The animated path export was created for two files—`AnimatedRoad.as` that extends the `PathAnimator` class, and `PathAWpath_0.as` that extends `Path` that contains path data. Prefab makes your life extremely easy as you don't need to tweak any of these two to get the animation working inside an Away3D scene. The only step you need to perform is to instantiate the `AnimatedRoad` class and pass to its constructor an object you wish to animate along the path.

First put the `PathAnimator.as` file into an `as3models` package of your project. Do the same for the `data` folder containing `PathAWpath_0.as`.

For this example, we will use the class from the first part of the recipe.

Open `PathDemo.as`. Inside `initGeometry()`, comment out these lines:

```
_cam.y=25000;  
_cam.lookAt(_road.position);
```

Add this code:

```
_view.clipping=new NearfieldClipping();  
var pathAnim:AnimatedRoad=new AnimatedRoad(_cam);  
pathAnim.play();
```

We add `NearClipping` to fix the disappearance of the road mesh faces close to the screen boundaries.

Run the file and you should see the camera moving along the path! Not bad for just two lines of code.

9

Working with External Assets

In this chapter, we will cover:

- ▶ Exporting models from 3DsMax/Maya/Blender
- ▶ Exporting models from 3DsMax to the ActionScript class
- ▶ Preparing MD2 models for Away3D in MilkShape
- ▶ Loading and parsing models (DAE, 3ds, Obj, MD2)
- ▶ Storing and accessing external assets in SWF
- ▶ Preloading 3D scene assets

Introduction

The Away3D library contains a large set of 3D geometric primitives such as Cube, Sphere, Plane, and many more. Nevertheless, when we think of developing breathtaking and cutting edge 3D applications, there is really no way to get it done without using more sophisticated models than just basic primitives. Therefore, we need to use external 3D modeling programs such as Autodesk 3DsMax and Maya, or Blender to create complex models. The Power of Away3D is that it allows us to import a wide range of 3D formats for static meshes as well as for animations.

Besides the models, the not less important part of the 3D world is textures. They are critical in making the model look cool and influencing the ultimate user experience.

In this chapter, you will learn essential techniques to import different 3D formats into Away3D. You will learn how to access and manipulate different properties of the models in order to adjust them to your needs. You also get acquainted with preparing textures for different purposes and their import into Away3D.

Exporting models from 3DsMax/Maya/Blender

You can export the following modeling formats from 3D programs: (Wavefront), Obj, DAE (Collada), 3ds, Ase (ASCII), MD2, Kmz, 3DsMax, and Maya can export natively Obj, DAE, 3ds, and ASCII. One of the favorite 3D formats of Away3D developers is DAE (Collada), although it is not the best in terms of performance because the file is basically an XML which becomes slow to parse when containing a lot of data. The problem is that although 3DsMax and Maya have got a built-in Collada exporter, the models from the output do not work in Away3D. The work around is to use open source Collada exporters such as ColladaMax/ColladaMaya, OpenCollada. The only difference between these two is the software versions support.

	ColladaMax/ColladaMaya	OpenCollada
Autodesk 3dsMax8	+	+
Autodesk 3dsMax 9	+	+
Autodesk 3dsMax2008	+	+
Autodesk 3dsMax2009/ MaxDesign2009	+	+
Autodesk 3dsMax2010		+
Autodesk 3dsMax2011/ MaxDesign2011		+
Autodesk Maya 8, 8.5	+	+
Autodesk Maya 2008	+	+
Autodesk Maya 2009	+	+
Autodesk Maya 2010		+
Autodesk Maya 2011		+

Getting ready

Go to <http://opencollada.org/download.html> and download the OpenCollada plugin for the appropriate software (3DsMax or Maya).

Go to <http://sourceforge.net/projects/colladamaya/files/> and download the ColladaMax or colladamaya plugin.

Follow the instructions of the installation dialog of the plugin. The plugin will get installed automatically in the 3dsMax/Maya plugins directory (taking into account that the software was installed into the default path).

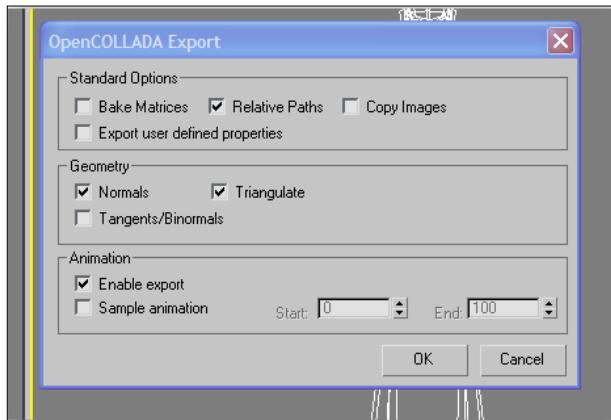
How to do it...

► 3DsMax:

Here is how to export Collada using OpenCollada plugin in 3DsMax2011.

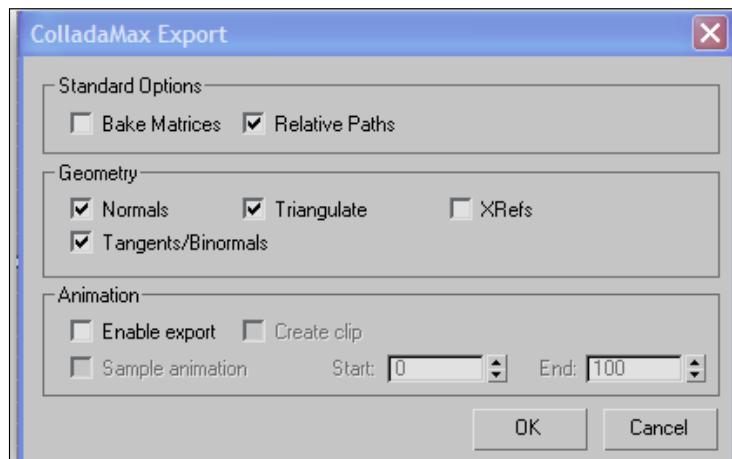
In order to export Collada (DAE) from 3DsMax, you should do the following:

1. In 3DsMax, go to **File** and click **Export or Export Selected** (target model selected).
2. Select the **OpenCOLLADA(*.DAE)** format from the formats drop-down list.



► ColladaMax export settings: (Currently 3DsMax 2009 and lower)

ColladaMax export settings are almost the same as those of OpenCollada. The only difference you can see in the exporting interface is the lack of **Copy Images** and **Export user defined properties** checkboxes.



Select the checkboxes as is shown in the previous screenshot.

Relative paths: Makes sure the texture paths are relative.

Normals: Exporting object's normals.

Copy Images: Is optional. If we select this option, the exporter outputs a folder with related textures into the same directory as the exported object.

Triangulate: In case some parts of the mesh consist of more than three angled polygons, they get triangulated.

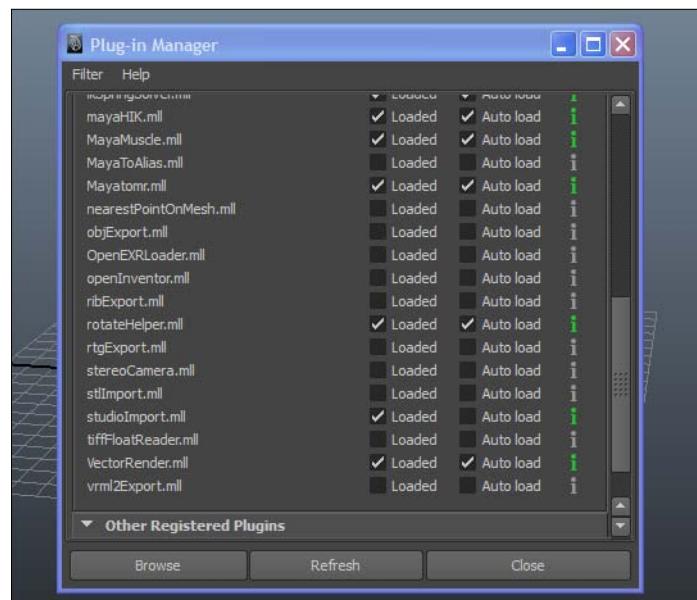
Animation settings:

Away3D supports bones animations from external assets. If you set bones animation and wish to export it, then check the *Sample animation* and set the *begin* and *end* frame for animation span that you want to export from the 3DsMax animation timeline.

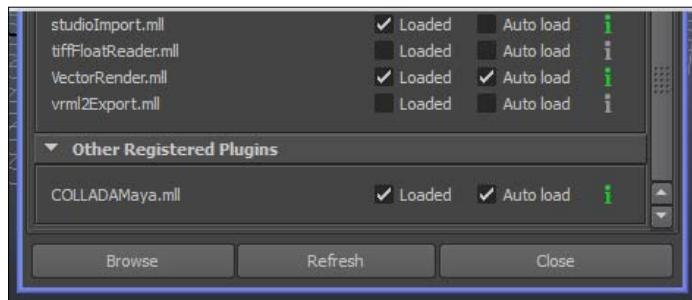
Maya:

For showcase purposes, you can download a 30-day trial version of Autodesk Maya 2011. The installation process in Maya is slightly different:

1. Open Maya.
2. Go to top menu bar and select **Window**.
3. In the drop-down list, select **Settings/Preferences**, in the new drop-down list, select **Plug-in manager**. Now you should see the **Plug-in Manager** interface:



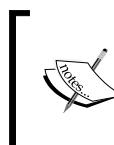
- Now click on the **Browse** button and navigate to the directory where you extracted the OpenCollada ZIP archive. Select the **COLLADAMaya.mll** file and open it.



- Now you should see the OpenCollada plugin under the **Other Registered Plugins** category.
- Check the **AutoLoad** checkbox if you wish for the plugin to be loaded automatically the next time you start the program.
- After your model is ready for export, click **File | Export All** or **Export** selected. The export settings for ColladaMaya are the same as for 3DsMax.

How it works...

The Collada file is just another XML but with a different format name (.dae). When exporting a model in a Collada format, the exporter writes into the XML nodes tree all essential data describing the model structure as well as animation data when one exports bone-based animated models.



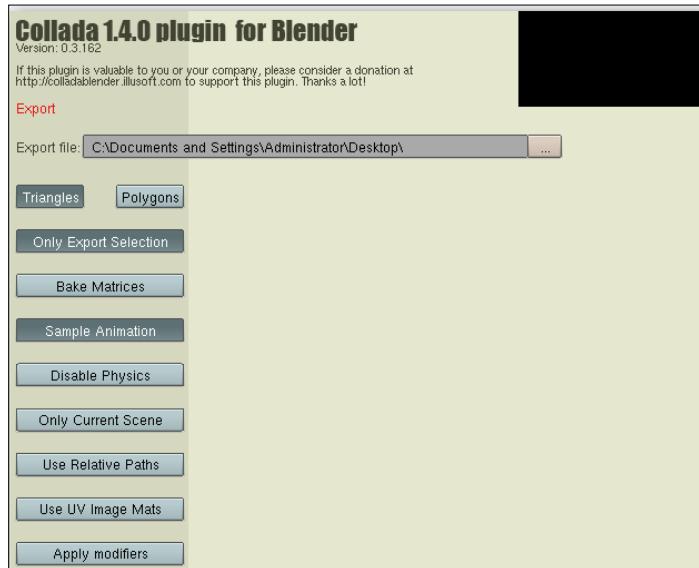
When deploying your DAE models to the web hosting directory, don't forget to change the .DAE extension to .XML. Forgetting will result in the file not being able to load because .DAE extension is ignored by most servers by default.

There's more...

Besides the Collada, you can also export OBJ, 3Ds, and ASE. Fortunately, for exporting these formats, you don't need any third party plugins but only those already located in the software.

Free programs such as Blender also serve as an alternative to expansive commercial software such as Maya, or 3DsMax. Blender comes with already built-in Collada exporter. Actually, it has two such exporters. At the time of this writing, these are 1.3 and 1.4. You should use 1.4 as 1.3 seems to output corrupted files that are not parsed in Away3D. The export process looks exactly like the one for 3dsMax.

Select your model. Go to **File**, then **Export**. In the drop-down list of different formats, select **Collada 1.4**. The following interface opens:



Select **Triangles**, **Only Export Selection** (if you wish to export only selected object), and **Sample Animation**. Set exporting destination path and click **Export** and close.

You are done.

See also

In Chapter 8, *Prefab3D*, the following recipe: *Exporting models from Prefab*.

Exporting models from 3DsMax to ActionScript class

The biggest disadvantage of using external model formats is that they consume precious time while being loaded and parsed. There is an alternative way to export meshes that you should be aware of and this is exporting directly into the ActionScript class. Apparently, the advantage is obvious as we don't need to load and parse the XML text of Collada, .3ds, or .obj formats. That saves us a lot of initiation time. So basically you would ask why not always use the ActionScript export. Well, there is one noticeable limitation to the current exporter—it doesn't export animations. Also if your application requires loading of multiple unique models, AS3 would not be the best solution. In such a case all the AS3 models are compiled directly into the program causing an unnecessary size increase of the whole application. So for the scenario like this, the external loading could be a better solution.

Getting ready

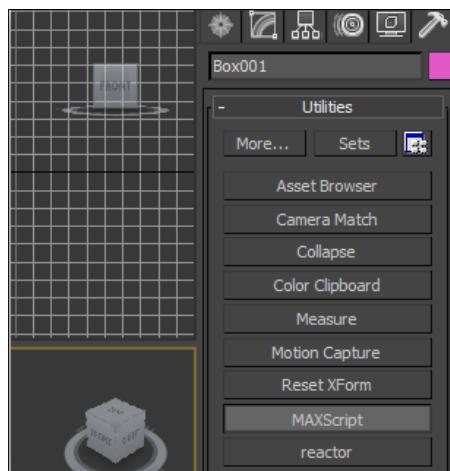
The script was written by Shirotokoro (also developer of WOW Physics engine) and can be downloaded at:

<http://drawlogic.com/2007/07/30/as3-geom-class-exporter-for-3ds-max-for-pv3d-sandy-and-away3d/>.

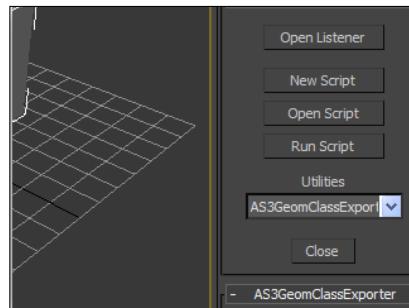
After the download, unzip the content into any easy to remember place as you will need to target the script file from inside 3DsMax.

How to do it...

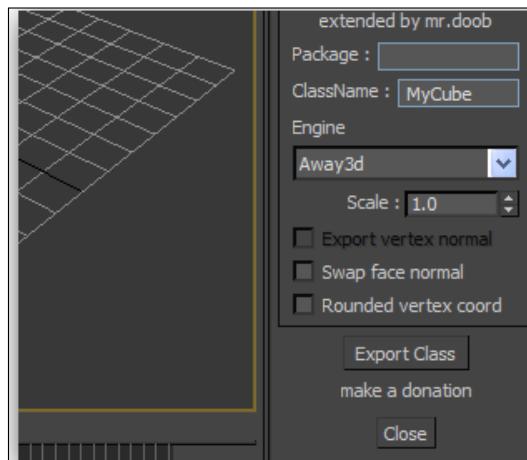
Open 3DsMax, in the side toolbar, open the **Utilities** dialog and select the **MAXScript** category:



Next click on the **Run Script** button and navigate to the directory where you saved the exporter files. Select the file `AS3GeomClassExporter.ms` from the `script3ds max` folder and open it. Now you can see it in the **Utilities** drop-down dialog. In order to activate it, click the drop-down list and select the **AS3GeomClassExport**:



Now we can see the exporter's interface opened:



Let's see the options we have:

- ▶ **Package:** Defines class package namespace
- ▶ **ClassName:** Add name for your **ActionScript** class
- ▶ **Engine:** You should select Away3D
- ▶ **Scale:** Relative scaling of the Mesh object
- ▶ **Swap face normal:** Check this if you want to invert the faces
- ▶ **Round vertex coord:** Rounding vertex coordinates floating number
- ▶ Click the **Export Class** button and you are done

Inside Away3D, you can insert the model you exported from 3DsMax by instantiating the class. In the following example, we create a new instance of the `MyCube()` model class, which we previously exported from 3DsMax:

```
private function initGeometry():void{
    var cb:MyCube=new MyCube();
    cb.material=new BitmapFileMaterial("material1.jpg");
    cb.rotationY=45;
    cb.scale(0.5);
    _view.scene.addChild(cb);

}
```

The applied texture wraps the model according to the UV mapping from 3DsMax.

How it works...

All the magic happens with the help of the MaxScript, which generates an Away3d class that extends the mesh super class. The exported class contains vertex coordinates, UV mapping, and faces indices data—all this is already in ActionScript with no need to load and parse external data.

There's more...

Check out Chapter 8, the following recipe, *Exporting models from Prefab*, to learn how to export geometry to the ActionScript class using Prefab.

Preparing MD2 models for Away3D in MilkShape

The MD2 format was introduced by ID software in 1997 and was initially used to create animated models for Quake 2. With time, many other game engines started to use this format. MD2 is a compact binary format that is easy to use for controlling character animation. MD2 is probably the best choice at the time of writing when you need to import animation into Away3D. Due to its binary nature, MD2 is loaded and parsed much faster than the Collada format.

Getting ready

Although some of the 3D packages such as Blender contain a native MD2 exporter, I suggest you use a program named **MilkShape** by **ChumBaumSoft**.

MilkShape is an easy to use multi-format importing, editing, and animating software that is really the perfect choice to create animated models for Away3D:

1. Go to <http://chumbalum.swissquake.ch/> and download MilkShape 3D version that is given free for 60 days trial. Install it.
2. In the following example, as well as in consequential ones, we are going to use great low poly models of characters from Valve's "Team Fortress". These were modeled by a talented 3D artist, Tom Tallian, and are free to download from his web page at <http://www.tomtallian.com/>.

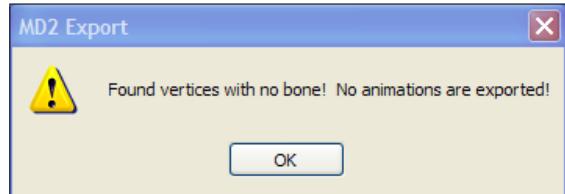
How to do it...

Initially, our spy is in .max format. In order to import it into MilkShape, we should convert it to one of MilkShape's recognizable formats. In 3DsMax, we export it as .3ds and it is ready for work in MilkShape.

Working with External Assets

Now let's open MilkShape. In the top menu bar, go to **File | Import**. From the multitude of the formats you can see in the drop-down menu, select Autodesk 3Ds.

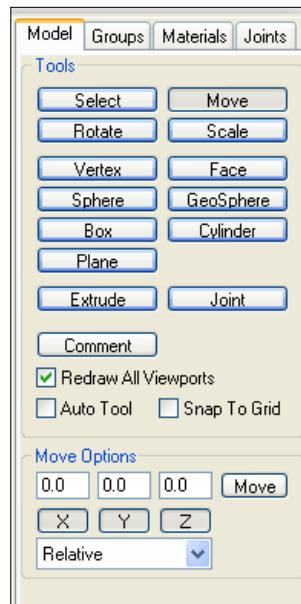
If we try to export our model as it is to MD2, we would get the following notification:



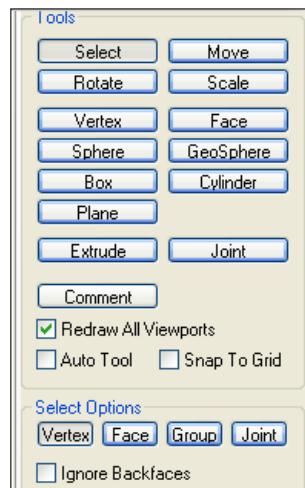
The problem is that MD2 requires bones to be attached to the vertices in order to compile.

Even if you don't need a rigged character but just a static MD2 model, you still have to define at least one joint and attach it to the mesh's vertices.

On the right side of the toolbar, click on the **Joint** button:



Create a joint in the region of a model's head. Now click on the **Select** button and in the selection type menu, click the **Vertex** button:



Switch to the **Joints** panel, double-click the joint (it is named **joint1** by default), then holding the **SHIFT** key, select all the meshes of the model. Now click the **Assign** button and the model is ready for MD2 export:



Go to **File | Export**. In the drop-down list, select Quake 2 MD2. You are done.

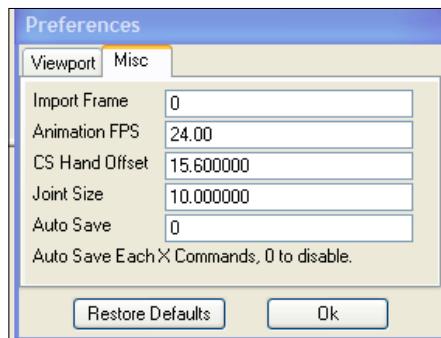
How it works...

As you can see, the MD2 export forces us to apply joints to our mesh. It is enough to apply a single joint to the whole mesh for the sake of a static model export which doesn't require animation. There is no need to worry about the possibility of wasted file size in this case, as applying the joint without setting actual key frame animation sequences doesn't make our file "heavy".

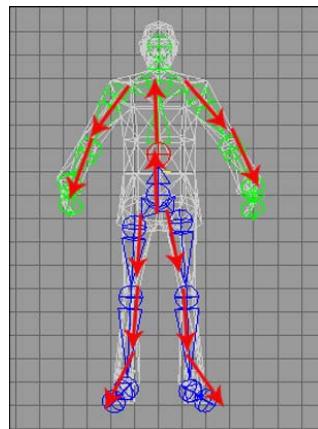
There's more...

Being done with static MD2 export, let's go and rig our character so that it can be animated. Believe me it is fun.

Before we begin, let's first scale up our joints a little so that it is going to be more convenient to control them. Go to **File | Preferences**. On the following screenshot, increase the number of the **Joint Size** to whatever value you think looks good for your joint circles:



As you now know how to set joints, let's create a full skeleton for our spy. To create an intact complete skeleton, we need to follow a certain sequence:



As you can see from the creation stream screenshot in the previous image, the first point of the skeleton is located in the basin area of the body. The workflow is as follows:

1. Create a joint in the basin area.
2. Then continue creating additional joints downwards for the left leg.
3. Then once again select the basin joint and continue to create joints for the right leg.
4. Now when we are done with the legs, let's go up. Select the basin joint and then create the next two—first in the neck region and the second in the head center.
5. Now all that is left is to do the hands. Select neck joint, and create sequence of additional joints along the right hand down. Then again select the neck joint and return over the same routine for the left hand.
6. Now adjust the joint in such a way that each joint corresponds to a body's anatomical joints and also check that the bones are centered to the surrounding mesh.

In order to animate your model, click the **Anim** button in the bottom-right corner of the program and you will see the timeline become active. Each animated movement should register with a key frame, otherwise the animation data would be lost. After defining the position of a joint, use **Ctrl + K** to set a key frame for that transformation. Proceed with this routine till you get the desired result.

The last part before we can export it is to define the animation sequence names. Unless you wish to have only one single animation type of your model, you need to give names to different sequences of the animation in order to use them later in Away3D to access actual animations.

Defining and naming of animation sequences is made outside the MilkShape program. You need to create a file with the .QC extension having the following structure:

```
// Sample MD2 config, copy into export directory
$modelname spyAnimatedReady.md2
$origin 0.0 0.0 0.0

// skins
$skinwidth 128
$skinheight 128
$skin skin.pcx
$skin pain.pcx

// sequences
$sequence stand 1 3
$sequence walk 4 84
```

We are interested in the last block. There we actually define the animations name, as well as a start and end frame of the given sequence. Notice that it is critical to have a single space gap between sequence definition {name} {start frame} {endframe}.

After defining your sequences, save the file to the same directory where you export the MD2 file.

Now export the MD2 file as we have done with the static model and now it is ready for Away3D.

See also

- ▶ In *Chapter 3, Animating the 3D World*, the recipe, *Working with MD2 animations*.

Loading and parsing models (DAE, 3ds, Obj, MD2)

Before we can start to manipulate external models in Away3D, we need to load them first into our application. Most of the Away3D supported formats are basically plain text (ASCII) files. Flash needs to load all that data first and only after the loading has completed, the Away3D relevant format parser starts to convert that data into actual geometry information. In this recipe, we see how to bring into scene different types of model formats using Away3D loading and parsing tools.

Getting ready

Let's export our lovely Spy model to the following formats: DAE, Obj, 3ds, and MD2. You can find already exported versions of the model inside the assets/SpyDifferentFormatsExport folder of this chapter.

Set up a default Away3D scene using our `AwayTemplate` class.

Let's begin with the Wavefront Obj loading, as there are few complications related to that format. In order to load .Obj, you have to supply an additional file with the .mtl extension. Its job is to supply models materials definitions. The model references this file in order to map the mesh correctly, as it was defined in the exporting program. When you export .Obj, the mtl is exported automatically with it.

There is a little problem with the mtl structure exported from 3DsMax, at the time of writing, by default, when we load obj into Away3D. The supplied texture map should be loaded and applied to the mesh automatically. This wouldn't happen if the mtl is corrupted. That is the case with 3ds max export. Fortunately, the solution is really a simple one, but you can spend hours if you don't know the precise problem. Let's see what is wrong with the original mtl.

Here is a screenshot of the corrupted `mtl` file exported from 3DsMax:

```

1 newmtl spy_flat
2 Ns 10.0000
3 Ni 1.5000
4 d 1.0000
5 Tr 0.0000
6 Tf 1.0000 1.0000 1.0000
7 illum 2
8 Ka 0.5880 0.5880 0.5880
9 Kd 0.5880 0.5880 0.5880
10 Ks 0.0000 0.0000 0.0000
11 Ke 0.0000 0.0000 0.0000
12 map_Ka C:\Documents and Settings\Administrator\Desktop\SpyDifferentFormatsExport\spyObjReady.jpg
13 map_Kd C:\Documents and Settings\Administrator\Desktop\SpyDifferentFormatsExport\spyObjReady.jpg

```

And this is what a working `mtl` file should look like. Also pay attention on lines indentation:

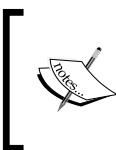
```

1
2 newmtl spy_flat
3
4 Ns 10.0000
5 Ni 1.5000
6 d 1.0000
7 Tr 0.0000
8 Tf 1.0000 1.0000 1.0000
9 illum 2
10 Ka 0.5880 0.5880 0.5880
11 Kd 0.5880 0.5880 0.5880
12 Ks 0.0000 0.0000 0.0000
13 Ke 0.0000 0.0000 0.0000
14 map_Ka spyObjReady.jpg
15 map_Kd spyObjReady.jpg
16

```

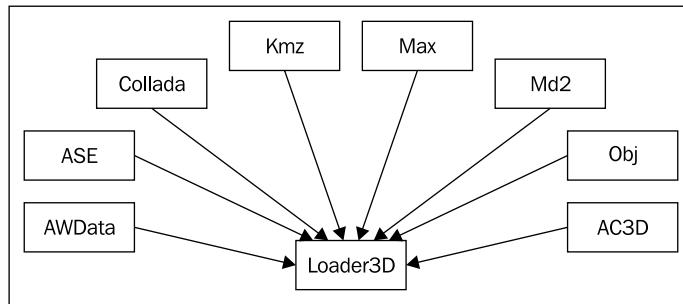
The original `mtl` needs two fixes. The second one is less noticeable. First, as you can see, the path string for texture location directory is absolute. So make them relative to the `model.obj` file. (The best thing to do is to put `mtl` obj and texture files in the same directory). The second problem is an indentation; there is a space between the left margin and the pathstring.

Probably the parser can't get over this gap. If you eliminate that space and push the strings to the left, so it is aligned with the new `mtl spy_flat`, the parser will process the file with no errors.



If you still experience a problem with `mtl` loading, or you don't wish to mess with `mtl` hacks, load the model into Prefab and export it to the `obj` anew. Prefab exports the `obj` format flawlessly along with `mtl`.

Away3D's Loader3D is capable of loading the following formats:



Sometimes the model is so big that you can't see it on the stage.

How to do it...

In the following example, we load three different formats. Inside the `initGeometry()` method, just uncomment one or more functions to invoke a loading of 3ds, dae, or obj type of the Spy model.



In 3ds max 2011 when exporting 3ds, the exporter warns you that it changes the texture path name in the file (actually it shortens it for some reason). You must change the actual texture name to the same; otherwise your model will not load. The best practice, in this case, is to keep the texture names short.

```
package
{
    public class MultipleFormatsLoadDemo extends AwayTemplate
    {

        private var _loader:Loader3D;
        private var _model:Object3D;
        public function MultipleFormatsLoadDemo()
        {
            super();
        }
        override protected function initGeometry() : void{

            //loadDAEModel(); ////open this line to load DAE
    
```

```
//loadMaxModel();////open this line to load 3ds
    loadObjModel();////open this line to load .Obj
}
////////obj load/////////
private function loadObjModel():void{
    _loader=Obj.load("assets/SpyDifferentFormatsExport/spyObjReady.
obj");
    _loader.addEventListener(Loader3DEvent.LOAD_
SUCCESS,onLoadReady);
}
///3ds load/needed to rename the texture to the name that was
given by 3ds max
private function loadMaxModel():void{
    _loader=Max3DS.load("assets/SpyDifferentFormatsExport/
spyObjReady.3ds");
    _loader.addEventListener(Loader3DEvent.LOAD_
SUCCESS,onLoadReady);
}

///Collada////
private function loadDAEModel():void{
    _loader=Collada.load("assets/SpyDifferentFormatsExport/
spyObjReady.dae");///
    _loader.addEventListener(Loader3DEvent.LOAD_
SUCCESS,onLoadReady);
}
private function onLoadReady(e:Loader3DEvent):void{
    _model=_loader.handle ;
    _model.scale(2);
    _view.scene.addChild(_model);
    _model.x=0;
    _model.y=0;
    _model.z=2000;
    _cam.lookAt(_model.position,Vector3D.Y_AXIS);

}
override protected function onEnterFrame(e:Event) : void{
    super.onEnterFrame(e);
    if(_model!=null){
        _model.rotationY++;
    }
}
```

How it works...

One thing you should notice when trying to use different loaders is the loaded object scale. In some formats such as ASE, the object will be huge after it is loaded in the stage, while in others, too small. So you should always play with scaling properties after loading to adjust the size to what you need.

Loader3D loads and then parses the data according to the loader type as shown previously. Because the loading operation is asynchronous, we have to set a listener and event handler for the LOAD_SUCCESS event. Only then we assign the loaded data to the actual 3D object that is usually Object3D or ObjectContainer3D using the handle() method of the loader.



The Loader3D has also graphical representation in form of 3D cube which displays loading progress of the object. If you wish to show this cube (many developers find it annoying), just add the _loader itself to the display list of the viewport after its initiation. When the loading process finishes, the cube unloads automatically.

There's more...

You need to use Loader3D only when you load external files. As you already know of the possibility of embedding models directly into an application, you should also get acquainted with the parse() method that reads the already existing model's data and transforms it into Object3D structure.

The embedded model parsing is quite straightforward. Here we don't need the Loader3D class. Instead, we are using a relevant format loader and parse the model with its parse() method.

Here is an example of parsing MD2, 3ds, DAE(Collada) models:

ModelsParsingDemo.as

```
package
{
    public class ModelsParsingDemo extends AwayTemplate
    {
        [Embed(source="assets/spyDifferentFormatsExport/spyObjReady.
dae", mimeType="application/octet-stream")]
        private var SpyModelDAE:Class;
    }
}
```

```
[Embed(source="assets/spyDifferentFormatsExport/spyObjReady.3ds", mimeType="application/octet-stream")]
private var SpyModel3ds:Class;
[Embed(source="assets/spyAnimatedReady.md2", mimeType="application/octet-stream")]
private var SpyModel:Class;

[Embed(source="assets/spyDifferentFormatsExport/spyObjReady.jpg")]
private var SpyTexture:Class;
private var _modelDAE:ObjectContainer3D;
private var _model3ds:ObjectContainer3D;
private var _modelObj:ObjectContainer3D;
private var _modelMD2:Mesh;
private var _bitMat:BitmapMaterial;

public function ModelsParsingDemo()
{
    super();
}
override protected function initMaterials() : void{
    _bitMat=new BitmapMaterial(Bitmap(new SpyTexture()).bitmapData);
}
override protected function initGeometry() : void{
    parseDAE();
    parse3ds();
    parseMD2()
}
private function parseDAE():void{
    var _dae:Collada=new Collada();
    _dae.centerMeshes=true;
    _modelDAE=_dae.parseGeometry(SpyModelDAE)as ObjectContainer3D;
    _view.scene.addChild(_modelDAE);
    _modelDAE.materialLibrary.getMaterial("spy_red").material=_bitMat;
    _modelDAE.scale(0.4);
    _modelDAE.x=100;
    _modelDAE.y=100;
    _modelDAE.z=500;
}
private function parse3ds():void{
    var max3ds:Max3DS=new Max3DS();
    max3ds.centerMeshes=true;
    _model3ds=max3ds.parseGeometry(SpyModel3ds)as ObjectContainer3D;
    _view.scene.addChild(_model3ds);
```

```
_model3ds.materialLibrary.getMaterial("spy_red").material=_bitMat;
    _model3ds.scale(0.4);
    _model3ds.x=-100;
    _model3ds.y=100;
    _model3ds.z=500;

}
private function parseMD2():void{
    var md2:Md2 = new Md2();
    _modelMD2 = md2.parseGeometry(SpyModel) as Mesh;
    _modelMD2.material = _bitMat;
    _modelMD2.scale(0.005);
    _modelMD2.x=-100;
    _modelMD2.y=-70;
    _modelMD2.z=500;
    _view.scene.addChild(_modelMD2);
}
override protected function onEnterFrame(e:Event) : void{
    super.onEnterFrame(e);
    _modelDAE.rotationY++;
    _model3ds.rotationX++;
    _modelMD2.rotationX--;
}
}
```

Do note that we also need to embed the models material and assign them to the models in runtime.

Storing and accessing external assets in SWF

If we look for simpler and less time consuming ways to set external assets pull for our application, we can do it right away with a few steps without ever going outside flex. We can pack the assets into SWF files and then retrieve them by either loading the compiled SWF or embedding it as we do with the rest of the assets and then parsing it.

In this example, we will use SWF embed, but that is the same process with loaded SWF.

Getting ready

First we are going to create an SWF serving as an assets bank. Basically, it is empty from methods class with embedded assets using the embed [] metatag:

SWFResourcesFile.as

```
package
{
    public class SWFResourcesFile extends Sprite
    {
        [Embed(source="assets/spyDifferentFormatsExport/spyObjReady.
dae", mimeType="application/octet-stream")]
        public var SpyModelDAE:Class;
        [Embed(source="assets/spyDifferentFormatsExport/spyObjReady.jpg")]
        private var SpyTexture:Class;
        public function SWFResourcesFile()
        {
        }
    }
}
```

Here we embedded our Spy model and its texture.

Now let's see how to access these classes from the main Away scene application.

How to do it...

SWFResourceDemo.as:

```
package
{
    public class SWFResourceDemo extends AwayTemplate
    {
        [Embed(source="assets/spyDifferentFormatsExport/SWFResourcesFile.
swf", mimeType="application/octet-stream")]
        private var ModelResource:Class;
        private var _bytesLoader:Loader;
        private var _model:Object3D;
        public function SWFResourceDemo()
        {
            super();
        }
    }
}
```

```
override protected function initGeometry() : void{
    loadModel();
}
private function loadModel():void{
    _bytesLoader=new Loader();
    _bytesLoader.addEventListener(Event.ENTER_FRAME,waitForLoad,fal
se,0,true);
    var lc:LoaderContext=new LoaderContext(false,ApplicationDomain.
currentDomain);
    _bytesLoader.loadBytes(new ModelResource(),lc);
}
private function waitForLoad(e:Event):void{
    var loader:Loader=e.target as Loader;
    if(loader.contentLoaderInfo.content!=null){
        _bytesLoader.removeEventListener(Event.ENTER_
FRAME,waitForLoad);
        parseModel();
    }
}
private function parseModel():void{
    var appDomain:ApplicationDomain=_bytesLoader.contentLoaderInfo.
applicationDomain;

    var modelClass:Class=appDomain.getDefinition("SWFResourcesFile_"
SpyModelDAE")as Class;
    var Texture:Class=appDomain.getDefinition("SWFResourcesFile_"
SpyTexture")as Class;
    var _mat:BitmapMaterial=new BitmapMaterial(Bitmap(new
Texture()).bitmapData);
    var dae:Collada=new Collada();
    _model=dae.parseGeometry(new modelClass());
    _model.materialLibrary.getMaterial("spy_red").material=_mat;
    _model.z=1000;
    _model.scale(0.5);
    _view.scene.addChild(_model);
}
}
```

How it works...

A few important things to understand here: If you work with embedded SWF, as in this example, you should use the `loadBytes()` method of the Flash Loader class. That is because we are parsing `byteArray` and not loading external SWF. The second and most important is when we execute `loadBytes()`, we should wait for one frame exactly before handling loaded data, even if it is not actually loaded from an external source. That is why in the previous example we created the `waitForLoad(e:Event)`: function. This function checks when `contentLoaderInfo.content` is not null anymore and only then it triggers the `parseModel()` function which retrieves the reference to the embedded sources classes from the loaded content that is represented in the form of `ApplicationDomain`. After assigning the retrieved content to the class object, we treat them the same way as we normally do with the embedded assets in Away3D.

Using SWF as assets, source speeds up loading not just because the data is binary but also because the packed SWF has a much reduced size in comparison with raw models weight. For example, here the Spy collada model is 53k, its texture is 18k, and in total 71k of data to load. But our `SWFResourcesFile.swf` which contains them both weighs just 32k. Nice optimization, isn't it?

See also

In *Chapter 6, Using Text and 2D Graphics to Amaze*, the recipe: *Creating 3D objects from 2D vector data*.

Preloading 3D scene assets

One widespread mistake that beginners in Away3D make is with the assets preloading routine. As a matter of fact, in many cases, one can see a total lack of preloading process that causes the scene content parts to appear suddenly without any logical order. Another well known issue is sequential loading of multiple objects. When we load a big amount of external assets, it is crucial to establish full monitoring of the loading process. In scenarios where there are multiple server calls and there is strong assets inter-dependency, loading errors may happen, and if you have no mechanism to get out of this gracefully, all your applications can crash even before they have initiated.

Getting ready

1. You are going to learn two techniques. One is really simple and can be done without the need of any third party content (just your coding skills). In the second part, we will use `BulkLoader` by Arthur Debert that is a robust open source set of tools for managing data loading in Flash. You should download the Library from this link:
<http://code.google.com/p/bulk-loader/>.

2. We have created five primitive models in 3Ds max for this example and exported them to 3Ds format along with five related texture maps. Get them from the assets/bulkassets folder in this chapter's source code. You can use your own models, but pay attention that because 3ds is binary and the code for non-binary formats will be slightly different.
3. Create a new class based on our AwayTemplate, name it MultiLoading.as, and add the code given in the next section.

How to do it...

MultiLoading.as

```
package
{
    public class MultiLoading extends AwayTemplate
    {
        private var _assetsToLoad:Array;
        private var _loadedStack:Array=[];
        private var _loader:Loader3D;
        private var _counter:uint=0;
        public function MultiLoading()
        {
            _assetsToLoad= ["cap.3ds", "Cone.3ds", "Knot.3ds", "Piramid.3ds", "star.3ds"];
            super();
            _cam.z=-600;
        }
        override protected function initGeometry() : void{
            _loader=Max3DS.load("assets/bulkassets/" + _assetsToLoad[_counter]);
            _loader.addOnSuccess(onLoadReady);
            _loader.addOnError(onLoadError);
        }
        private function onLoadReady(e:Loader3DEvent):void{
            var obj3d:Mesh=_loader.handle as Mesh;
            obj3d.scale(2);
            obj3d.centerPivot();
            obj3d.z=500;
            obj3d.x=Math.random()*_view.width-400;
            obj3d.y=Math.random()*_view.height-300;
            _loadedStack.push(obj3d);
        }
    }
}
```

```

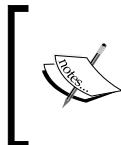
        if(_counter<_assetsToLoad.length-1) {
            _counter++;
            _loader=Max3DS.load("assets/bulkassets/"+_assetsToLoad[_counter]);
            _loader.addOnSuccess(onLoadReady);
            _loader.addOnError(onLoadError);

        }else{
            trace("all assets have been loaded");
            _loader.removeOnError(onLoadError);
            _loader.removeOnSuccess(onLoadReady);
            for(var i:uint=0;i<_loadedStack.length;++i){
                _view.scene.addChild(_loadedStack[i]);
                TweenMax.to(_loadedStack[i],4,{rotationX:360,rotationY:180,yoyo:0,delay:Math.random()*2});
            }
        }
    }
    private function onLoadError(e:Loader3DEvent):void{
        trace(_loader.IOErrorText);
    }
    override protected function onEnterFrame(e:Event) : void{
        super.onEnterFrame(e);
    }
}
}
}

```

How it works...

This example is straightforward. We create an array of file paths for the geometry to load. Then we initiate the loading process by calling only once the function `initGeometry()`. When the `Loader3D.COMPLETE` event is triggered, we initiate the loaded asset and push it into an array for future use, but not adding it to the scene till the rest of the objects have finished loading. Each time any new asset has been loaded, we increment `_counter` variable until it equals the length of the assets paths array. When all the assets are loaded, we iterate through the array containing all the loaded models and add them to the scene.



Note that in the `onLoadReady()` function, we add the event listeners `addOnSuccess` and `addOnError` again. That is because each time the loader completes a single load, its listeners get removed automatically.

There's more...

At first glance, you can feel that working with BulkLoader adds much more coding. Well, it is partially true. You do have to code a few more lines, as, for instance, you have to add the materials manually to the loaded content. That is, as you will see now, due to the fact that after the BulkLoader finishes loading the data, we handle it to Away3D parsers and from then on proceed with the regular model parsing routine. On the other hand, using BulkLoader you receive very powerful tools for controlling and monitoring multiple assets loading routines. It is really out of the scope of this book to present all the features of this library, but we will use the most important ones in the following example:

`BulkLoaderDemo.as`

```
package
{
    public class BulkLoaderDemo extends AwayTemplate
    {
        private var _assetsToLoad:Array;
        private var _mapsToLoad:Array;
        private var _loadedModels:Array=[];
        private var _loadedTextures:Array=[];
        private var _bloader:BulkLoader;// loader for models
        public function BulkLoaderDemo()
        {
            _assetsToLoad=["cap.3ds","Cone.3ds","Knot.3ds","Pyramid.3ds","star.3ds"];//model input for load
            _mapsToLoad=["m1.jpg","m2.jpg","m3.jpg","m4.jpg","m5.jpg"];//texture input for load
            super();
            _cam.z=-600;
        }
        override protected function initGeometry() : void{
            initBulkLoader();
        }
        private function initBulkLoader():void{
            _bloader=new BulkLoader("mainLoader",BulkLoader.DEFAULT_NUM_
CONNECTIONS,BulkLoader.LOG_INFO);
            _bloader.addEventListener(BulkProgressEvent.
COMPLETE,onLoadReady);
            for(var i:uint=0;i<_assetsToLoad.length;++i){
                _bloader.add("assets/bulkassets/"+_assetsToLoad[i],{type:BulkL
oader.TYPE_BINARY});
            }
        }
        private function onLoadReady(event:BulkProgressEvent)
        {
            if(event.type==BulkLoader.COMPLETE)
            {
                var loader:Loader=_bloader.loader;
                var models:Vector<Object>=loader.content;
                var textures:Vector<Object>=loader.textures;
                var materials:Vector<Object>=loader.materials;
                var geometry:Vector<Object>=loader.geometries;
                var count:int=0;
                for(var i:int=0;i<models.length;i++)
                {
                    count+=models[i].vertices.length;
                }
                trace("Count of vertices: "+count);
            }
        }
    }
}
```

There are four stages in the overall process here. First we defined two arrays—one for holding the file paths for the models with the second for texture images. Then we initiate the `BulkLoader` instance triggering the `initBulkLoader()` method.

When instantiating the `BulkLoader`, there is one very powerful feature in the constructor called `logLevel`. You can choose different types of logging during the loading process. `LOG_INFO` is the deepest one allowing you to watch all the processes in detail.

We add to the loading queue both models and the images from related arrays. It is critical to mark which type of the data to load within the curly brackets of the `add()` method. Not doing so with models will throw you errors from the Away3D binary parser later as it will get text data instead of `ByteArray`. Triggering the `_loader.start()` method will initiate loading.

The third step is when all the assets have been loaded. There is no need for iterators and recursive calls because once the `bulkLoader` has loaded all the files, only then it dispatches the `COMPLETE` event. Now we call the `parseData()` function that accepts, as an argument, the array of all the loaded objects. Before we can start parsing this content, there is one more step to accomplish. `_loader.items` array stores all the assets data together. That means that we have got an array of two types—`ByteArray` and `Bitmap`. So before handing it to the Away3D, we need to sort this stuff by the data type. The first `for` loop statement in the `parseData()` function does just that. We know that behind the scenes—`URLLoader` and `Loader`-ActionScript, generic classes manage the loads. We also know that `URLLoader` doesn't load image files, but data such as bytes and texts. Checking for the content loader type, we sort the assets by their data types and store them into separate arrays.

The last step is to hand the data to Away3D's Max3D parser. That is done in the second `for` loop statement of the `parseData()` function. Here we pull each model from the `_loadedModels` array, parse it, and also add relevant texture to its material. And we are done.

As you have seen from this example, working with third-party loader is a more complex task, but you gain much more control of the process. For instance, with `BulkLoader`, you can tag with ID's loaded content, get information about load speed, and the number of loaded objects. You can set loading priority of different assets and initiate multiple loading connections. You can exit failed loads graciously without breaking the overall process. Use these tools as they can make your life easier in most cases.

There's more

Away3D has got a couple of classes that allow you to load textures sequentially. These are `TextureLoader` and `TextureLoaderQueue` that reside in the `away3d.loaders.utils` package. In fact, you can adjust them for loading other data types as well.

10

Integration with Open Source Libraries

In this chapter, we will cover:

- ▶ Setting Away3D with JigLib
- ▶ Creating a physical car with JigLib
- ▶ Morphing Away3D geometry with AS3DMOD
- ▶ Exploding particles with FLINT
- ▶ Setting augmented reality with FLARToolkit in Away3D
- ▶ Adding Box2D physics to Away3D objects

Introduction

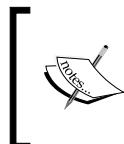
There is no doubt that Away3D has got the biggest set of features among open source Flash 3D engines. However, its scope (at the time of writing) is limited mostly to the 3D content creation and rendering. At the time of writing, Away3D has neither a built-in physics engine nor a particle system. Of course, one can develop those on one's own, by first refreshing the classic mechanics knowledge from high school. There are open source libraries which are developed and maintained by teams such as that of Away3D, whose purpose is to make your life easy when you need to plug physical behavior, or include advanced particle effects into your 3D environment.

In this chapter, we are going to taste several mainstream libraries which are `JigLibFlash`, `FLINT`, `FLAR`, and `AS3DMOD`. We are not going to dive deeply into their advanced features. Instead, our purpose here is to learn the setup routine of each library and how to integrate it with Away3D objects in the least painful way.

Setting Away3D with JigLib

JigLibFlash is a full-scale physics 3D engine originally written in C++ and then ported to ActionScript. JigLibFlash has a robust set of physical behaviors which can be applied to 3D entities. At the time of writing, the main disadvantage of the engine is its heavy impact on CPU. Truth be told—it is not suited for scenes that are filled with multiple physical interactions. Although the development team works hard on the engine's optimization, it is still recommended to reduce to a minimum the number of 3D objects which should be affected by it. The library is free for both private and commercial use. You can visit the project's home page at www.jiglibflash.com.

In this recipe, we are going to learn how to set up the JigLibFlash engine and how to integrate the Away3D geometry into it.



If you are completely new to JigLibFlash, or conversely, wish to extend your knowledge of the library, visit: www.jiglibflash.com where you can find a bunch of useful links to tutorials or enter the forum where you can get advice and explanations on different topics related to the engine use.

Getting ready

Set up a basic Away3D scene using AwayTemplate.

Make sure you downloaded and linked to your project the source code of the latest JigLibFlash build from the following URL: <http://code.google.com/p/jiglibflash/>.

How to do it...

In the following program, we set up an Away3D scene which contains a floor on which we drop sphere objects from an altitude by mouse clicks. The scene geometry is then wrapped with the JigLibFlash engine which is responsible for the physical behavior of a scene's objects:

JigLibSetupDemo.as

```
package
{
    public class JigLibSetupDemo extends AwayTemplate
    {
        private var _awayPhys:Away3DPhysics;
        private var _floor:RigidBody;
        private var _sp:Sphere;
        private var _ballsArr:Vector.<RigidBody>=new Vector.<RigidBody>();
    }
}
```

```
private const INIT_POS:Vector3D=new Vector3D(0,800,0);
private const MAX_NUM_OF_BALLS:uint=5;
public function JigLibSetupDemo()
{
    super();
    initRigBodies();
    _cam.y=400;
    _cam.z=-2000;
    _view.clipping=new NearfieldClipping();

}
override protected function initListeners() : void{
    super.initListeners();
    stage.addEventListener(MouseEvent.CLICK,onClick,false,0,true);
}
override protected function initGeometry() : void{
    initJigLib();
}
override protected function onEnterFrame(e:Event) : void{
    super.onEnterFrame(e);
    _awayPhys.step();
    if(_ballsArr.length>MAX_NUM_OF_BALLS){
        var rb:RigidBody=_ballsArr.shift();
        _awayPhys.removeBody(rb);
        var mesh:Mesh=_awayPhys.getMesh(rb);
        _view.scene.removeChild(mesh);
        rb=null;
        mesh=null;
    }
}
private function initJigLib():void{
    _awayPhys=new Away3DPhysics(_view,10);

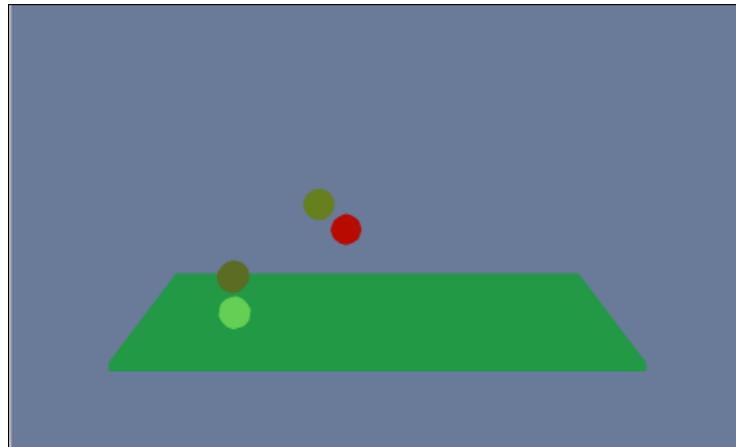
}
private function initRigBodies():void{

    var finiteFloor:RigidBody=_awayPhys.createCube({width:600,height:10,depth:600});
    finiteFloor.moveTo(new Vector3D(0,0,0));
    finiteFloor.movable=false;
    _awayPhys.getMesh(finiteFloor).pushback=true;
    _awayPhys.getMesh(finiteFloor).material=new
    ColorMaterial(0x239944);
```

```
        _view.camera.lookAt(new Vector3D(0,0,0),Vector3D.Y_AXIS);
    }

    private function dropBall():void{
        var sp:Sphere=new Sphere({radius:20,material:new
ColorMaterial(Math.floor(Math.random()*0xffffffff))});
        var pBall:RigidBody=new JSphere(new Away3dMesh(sp),20);
        _view.scene.addChild(sp);
        sp.position=INIT_POS;
        pBall.mass=25;
        pBall.restitution=15;
        pBall.friction=0.23;
        _awayPhys.addBody(pBall);
        pBall.moveTo(INIT_POS);
        pBall.addBodyTorque(new Vector3D(23,0,15));
        pBall.addWorldForce(new Vector3D(60,0,50),new Vector3D(1,0,1));
        _ballsArr.push(pBall);
    }
    private function onMouseClick(e:MouseEvent):void{
        dropBall();
    }
}
```

Here is the result of the physical balls bouncing on the plane surface:



How it works...

First let's begin from the JigLib engine initiation. JigLibFlash contains a module called plugins for several 3D engines such as PV3D, Sandy, Away3D, and more. These plugins are the entry point for the engine setup. In Away3D, we should instantiate the `Away3DPhysics()` plugin, which accepts to its constructor the reference to a scene's `View3D`, whereas the second argument is the speed of physics update. We set it up inside the `initJigLib()` method:

```
private function initJigLib():void{
    _awayPhys=new Away3DPhysics(_view,10);

}
```

The second step is the calling of the `_awayPhys.step()` method for a physics update, which should usually be placed inside the `onEnterFrame()` function for a continuous update of the physical engine. From this point on, your physics environment is working.

Next, we set up a floor which is going to resist the impact of the falling spheres. In this particular example, I created a floor based on the Cube rigid body:

```
var finiteFloor:RigidBody=_awayPhys.createCube({width:600,height:10,d
epth:600});
```

The other ways are by using the `JPlane()` rigid body or the `_awayPhys.createGround()` method. You can find these implementations commented in the source code file of this example. The reason I used the cube approach is that the other two create infinite planes so that the objects moving on top of them never reach the edge of the surface.

Now we need to add a material to the geometry and push it back to prevent Z-Sorting issues. Because we have no direct access to the Away3D primitive of the floor cube (as it was created indirectly with the `_awayPhys.createCube()` method) in order to reach it through its `RigidBody` layer, we use the `getMesh()` function:

```
finiteFloor.moveTo(new Vector3D(0,0,0));
finiteFloor.movable=false;
_awayPhys.getMesh(finiteFloor).pushback=true;
_awayPhys.getMesh(finiteFloor).material=new
ColorMaterial(0x239944);
```

We also set the `finiteFloor.movable` property to false so that the floor will not be affected either by gravity force or by the impact from the falling bodies.

The final feature we implement is the dropping of the spheres towards the floor, each time the user clicks the mouse. On each click, we call the `dropBall()` method, inside which we create a regular Away3D `Sphere` instance:

```
var sp:Sphere=new Sphere({radius:20,material:new ColorMaterial(Math.
floor(Math.random()*0xffffffff))});
```

Then, we wrap it up with JigLib RigidBody of the JSphere type so that it can react to physical forces:

```
var pBall:RigidBody=new JSphere(new Away3dMesh(sp),20);  
_view.scene.addChild(sp);
```

 In the world of physics, the bodies which have physical properties and are reactants to physical forces without their shape being deformed are called rigid bodies. (There are also soft bodies, but these are irrelevant to our discussion.) In JigLibFlash, each mesh should be assigned a RigidBody that matches its geometrical classification such as Away3D Sphere wrapped by JSphere. Any geometry that is not a RigidBody remains unaffected by the physical environment.

Next, we initiate the pBall RigidBody essential physical properties such as default position mass, friction, and restitution:

```
sp.position=INIT_POS;  
pBall.mass=25;  
pBall.restitution=15;  
pBall.friction=0.23;
```

Don't forget to register the RigidBody to the engine in order to integrate it into the physical world:

```
_awayPhys.addBody(pBall);
```

The last step here is to apply some forces to the pBall so that its motion, while falling, would be non-linear. We add a body torque which is in simple world rotational force. Another force we add is a kind of impulse which pushes the body in a direction and with the force defined by two vectors:

```
pBall.addBodyTorque(new Vector3D(23,0,15));  
pBall.addWorldForce(new Vector3D(60,0,50),new Vector3D(1,0,1));
```

We add each new pBall to an array which then serves us to keep count of the total number of spheres on the scene, and by deleting one of them each time, this number passes the predefined limit. This way, we don't kill our frame rate with the accumulated instances of pBall:

```
_ballsArr.push(pBall);
```

We execute the count and cleanup inside the onEnterFrame() function:

```
if(_ballsArr.length>MAX_NUM_OF_BALLS){  
    var rb:RigidBody=_ballsArr.shift();  
    _awayPhys.removeBody(rb);
```

```
var mesh:Mesh=_awayPhys.getMesh(rb);
_view.scene.removeChild(mesh);
rb=null;
mesh=null;
}
```

As you can see from the preceding block, we remove the pBall RigidBody as well as the Away3D mesh, which is the Sphere primitive instance.

See also

For more information on JigLib, detailed tutorials, and a forum, you can visit the webpage of the engine at <http://www.jiglibflash.com/>.

Creating a physical car with JigLib

One of the most popular scenarios where you would like to integrate physics is a racing game simulating physical driving behavior. In this case, each car should become a physical body which contains built-in functionality of a car and reacts to the forces applied to it by the environment. Those of you who are not fond of complex physics, and I bet most of you are not, would not wish to write a physical model for a car from the ground up. So here is the good news, JigLibFlash has got a generics `JCar` class which allows you to set up (rig) your physical car relatively easily. Let's see how to do that.

Getting ready

Set up a basic Away3D scene using `AwayTemplate`.

If you still have not downloaded and connected to your Flash project, you can go to the latest build of the JigLibFlash library and get it from the following URL: <http://code.google.com/p/jiglibflash/>.

In this example, we will make use of a car model called `PoliceCar.dae`, which is located in this chapter's source code directory in the `assets/models` folder. Make sure you copy it into your project along with its texture `TextureDPW.jpg`.

How to do it...

The following program sets up a scene with a keyboard keys controlled car which incorporates JigLibFlash JCar physics:

JigLibCarDemo.as

```
package
{
    public class JigLibCarDemo extends AwayTemplate
    {
        [Embed(source="/assets/models/PoliceCar.dae", mimeType="application/octet-stream")]
        private var CarModel:Class;
        [Embed(source="/assets/models/TextureDPW.jpg")]
        private var CarTexture:Class;
        private var _awayPhys:Away3DPhysics;
        private var _carModel:ObjectContainer3D;
        private var _carBody:JCar;
        private var _wheelFR:ObjectContainer3D;
        private var _wheelFL:ObjectContainer3D;
        private var _wheelBR:ObjectContainer3D;
        private var _wheelBL:ObjectContainer3D;
        private var _bitmat:BitmapMaterial;
        public function JigLibCarDemo()
        {
            super();
            initTargetCam();
            _view.clipping=new NearfieldClipping();
        }
        override protected function initListeners() : void{
            super.initListeners();
            stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDownHandler);
            stage.addEventListener(KeyboardEvent.KEY_UP, keyUpHandler);
        }
        private function initJigLib():void{
            _awayPhys = new Away3DPhysics(_view, 5);
            var physGround:RigidBody = _awayPhys.createGround({width: 1500,
height: 1500}, 0);
            _awayPhys.getMesh(physGround).material=new
ColorMaterial(0x993377);
            physGround.moveTo(new Vector3D(0,0,0));
            _view.scene.addChild(_awayPhys.getMesh(physGround));
        }
    }
}
```

```

        _cam.lookAt (_awayPhys.getMesh (physGround).position);
        _awayPhys.getMesh (physGround).pushback=true;
    }
    override protected function initMaterials() : void{
        _bitmat=new BitmapMaterial(Bitmap(new CarTexture()).bitmapData);
    }
    override protected function initGeometry() : void
    {
        initJigLib();
        setupJCar();
    }
    override protected function onEnterFrame(e:Event) : void
    {
        super.onEnterFrame(e);
        _awayPhys.engine.integrate(0.24);
        updateWheelSkin();
    }
    private function setupJCar():void{
        var dae:Collada=new Collada();
        _carModel=dae.parseGeometry(new CarModel())as ObjectContainer3D;
        _carModel.scale(5);
        _view.scene.addChild(_carModel);
        _carBody = new JCar(new Away3dMesh(_carModel));
        _carBody.setCar(45, 4, 800);

        _carBody.chassis.moveTo(new Vector3D(0, 20, 0));
        _carBody.chassis.rotationY = 90;
        _carBody.chassis.mass = 110;
        _carBody.chassis.sideLengths = new Vector3D(40, 20, 90);
        _awayPhys.addBody(_carBody.chassis);
        _carBody.setupWheel("WheelFL", new Vector3D(-20, 0, 35), 1.2,
4, 3, 12, 0.4, 0.7, 2);
        _carBody.setupWheel("WheelFR", new Vector3D(20, 0, 35), 1.2,
4, 3, 12, 0.4, 0.7, 2);
        _carBody.setupWheel("WheelBL", new Vector3D(-20, 0, -35), 1.2,
4, 3, 12, 0.4, 0.7, 2);
        _carBody.setupWheel("WheelBR", new Vector3D(20, 0, -35), 1.2,
4, 3, 12, 0.4, 0.7, 2);
        _wheelFL = _carModel.getChildByName("node-Cylinder08") as
ObjectContainer3D;
        _wheelFR = _carModel.getChildByName("node-Cylinder06") as
ObjectContainer3D;
        _wheelBL = _carModel.getChildByName("node-Cylinder07") as
ObjectContainer3D;
    }
}

```

```
_wheelBR = _carModel.getChildByName("node-Cylinder01") as  
ObjectContainer3D;  
  
Mesh(_wheelBL.children[0]).invertFaces();  
Mesh(_wheelFL.children[0]).material=_bitmat;  
Mesh(_wheelFR.children[0]).material=_bitmat;  
Mesh(_wheelBL.children[0]).material=_bitmat;  
Mesh(_wheelBR.children[0]).material=_bitmat;  
var carCabine:ObjectContainer3D=_carModel.getChildByName("node-  
camaro01")as ObjectContainer3D;  
Mesh(carCabine.children[0]).material=_bitmat;  
Mesh(carCabine.children[0]).invertFaces();  
}  
private function keyDownHandler(event:KeyboardEvent):void  
{  
switch (event.keyCode)  
{  
case Keyboard.UP:  
_carBody.setAccelerate(-1);  
break;  
case Keyboard.DOWN:  
_carBody.setAccelerate(2);  
break;  
case Keyboard.LEFT:  
_carBody.setSteer(["WheelFL", "WheelFR"], 1);  
break;  
case Keyboard.RIGHT:  
_carBody.setSteer(["WheelFL", "WheelFR"], -1);  
break;  
case Keyboard.SPACE:  
_carBody.setHBrake(1);  
break;  
}  
}  
  
private function keyUpHandler(event:KeyboardEvent):void  
{  
switch (event.keyCode)  
{  
case Keyboard.UP:  
_carBody.setAccelerate(0);  
break;  
case Keyboard.DOWN:
```

```
    _carBody.setAccelerate(0);
    break;
  case Keyboard.LEFT:
    _carBody.setSteer(["WheelFL", "WheelFR"], 0);
    break;
  case Keyboard.RIGHT:
    _carBody.setSteer(["WheelFL", "WheelFR"], 0);
    break;
  case Keyboard.SPACE:
    _carBody.setHBrake(0);
    break;
}
}

private function updateWheelSkin():void
{
  _wheelFL.rotateTo(0,-90-_carBody.wheels["WheelFL"].
getSteerAngle(),0);
  _wheelFR.rotateTo(0,90-_carBody.wheels["WheelFR"].
getSteerAngle(),0);

  _wheelFL.children[0].yaw(_carBody.wheels["WheelFL"].
getRollAngle());
  _wheelFR.children[0].yaw(-_carBody.wheels["WheelFR"].
getRollAngle());

  _wheelBL.children[0].yaw (-_carBody.wheels["WheelBL"].
getRollAngle());
  _wheelBR.children[0].yaw(- _carBody.wheels["WheelBR"].
getRollAngle());
}

private function initTargetCam():void{
  var targetCam:TargetCamera3D=new TargetCamera3D();
  targetCam.target=_carModel;
  _view.camera=targetCam;
  targetCam.zoom=25;
  targetCam.z=-950;
  targetCam.y=40;
}
}
```

The following image shows the final result of our physical car:



How it works...

The preceding code looks complex and confusing at first glance, but as you will see shortly, the setup is not as scary as it appears to be. Now let's dive into the code.

The entry point after the Away3D scene initiation is JigLib engine start-up, which we set off inside the `initJigLib()` method. You can check out the previous recipe in order to learn how to initialize JigLibFlash. The core of the program is the `setupJCar()` function. Here we first parse the model of `PoliceCar.dae`, which is embedded into the application. Right after that, we instantiate a new `JCar` object passing the model of our car into its constructor:

```
_carBody = new JCar(new Away3dMesh(_carModel));
```

Next we define the steering properties of the car, which are by order steering angle, speed of steering rotation (on global y-axis), and the amount of drive torque which defines a car's torque force when steering:

```
_carBody.setCar(45, 3, 800);
```

From now on, the setup consists of two parts—chassis and wheels configuration. The chassis function, as you may understand from the name, is pretty much the same as a real world car's chassis. Obviously, the same is valid in relation to the wheels.



It is important to note that for each car model, the numeric values in the chassis and wheels setups will vary, as it always depends on the car's geometrical dimension and the coordinate system's compatibility. Don't take this for granted and paste the values you see here into your code, if you use a different car model. Instead, you should tweak those values until you achieve the desired physical behavior.

```
_carBody.chassis.moveTo(new Vector3D(0, 20, 0));
    _carBody.chassis.rotationY = 90;
    _carBody.chassis.mass = 110;
    _carBody.chassis.sideLengths = new Vector3D(40, 20, 90);
    _awayPhys.addBody(_carBody.chassis);
```

In the preceding block, we move the car to some default arbitrary position by assigning the `moveTo()` vector to its chassis. Then we define a mass, chassis side length, which basically means width-height-length. Now you should remember that the chassis is the skeleton of the car, which is responsible for its physical behavior and therefore it is registered to the engine and not the `JCar _carBody` instance, as one might imply.

Within next four lines, we define the physical properties of the car's wheels using the `setupWheel()` method of the `_carBody` object:

```
_carBody.setupWheel("WheelFL", new Vector3D(-20, 0, 35), 1.2, 4, 3,
12, 0.4, 0.7, 2);
```

The most important arguments here are the names of the wheel, which we will access later. In order to steer and roll it, the physical position of the wheel on the chassis and chassis radius is the fourth argument from the end. If you don't fine-tune these properties, you can find your car springing continuously into the air or jumping as if it were moving on top of rocky terrain. As for the rest of the argument, it is recommended to tweak them in order to achieve the desired behavior of the wheel, although most of them are fine when used with their default values.

Next, we need to access each wheel of our 3D car model and assign to it a variable name so that, later, we can steer and rotate them by applying the numeric values which are produced by the physical wheels:

```
_wheelFL = _carModel.getChildByName("node-Cylinder08") as
ObjectContainer3D;
```

As you can guess, the child name we access in the preceding line is the one you defined inside a 3D package for each wheel.

Now we assign a material to each part of the car:

```
Mesh(_wheelBL.children[0]).invertFaces();
    Mesh(_wheelFL.children[0]).material=_bitmat;
    Mesh(_wheelFR.children[0]).material=_bitmat;
    Mesh(_wheelBL.children[0]).material=_bitmat;
    Mesh(_wheelBR.children[0]).material=_bitmat;
    var carCabine:ObjectContainer3D=_carModel.getChildByName("node-
camaro01")as ObjectContainer3D;
    Mesh(carCabine.children[0]).material=_bitmat;
    Mesh(carCabine.children[0]).invertFaces();
```

We assign the same material to each wheel mesh, which is the only child of the parent ObjectContainer3D, as well as to the car's body which contains a baked texture of the whole car. Because each part of the car is UV mapped to a particular region of the texture, the same material applies a relevant map area to each part. Also note that I inverted the faces of a couple of meshes in order to fix the face normals direction issue, which may happen sometimes as a result of mesh import problems or incorrect modeling.

The next two methods, `keyDownHandler()` and `keyUpHandler()`, are called on keyboard user input `keyDownHandler`, as its name suggests, handles the arrows keys press while `keyUpHandler` their release. *UP* and *DOWN* arrow keys control forward and backward acceleration of the car, *LEFT* and *RIGHT* keys manage steering of the front wheels, and the *SPACE* bar triggers the car's brakes.

Now comes the last important method which updates wheels steering and rolling on each frame:

```
private function updateWheelSkin():void
{
    _wheelFL.rotateTo(0, -90-_carBody.wheels["WheelFL"] .
getSteerAngle(),0);
    _wheelFR.rotateTo(0, 90-_carBody.wheels["WheelFR"] .
getSteerAngle(),0);

    _wheelFL.children[0].yaw( _carBody.wheels["WheelFL"] .
getRollAngle());
    _wheelFR.children[0].yaw(- _carBody.wheels["WheelFR"] .
getRollAngle());

    _wheelBL.children[0].yaw (-_carBody.wheels["WheelBL"] .
getRollAngle());
    _wheelBR.children[0].yaw(- _carBody.wheels["WheelBR"] .
getRollAngle());

}
```

As you can see from the preceding code, we retrieve the steering angle values from JigLib JWHEEL instances and assign them to the front wheel meshes. We use the `rotateTo()` method and not `rotationY` for the steering because we want to set the local rotation of wheel mesh only around the y-axis. Using `rotationY` will cause rotation of the wheel in more than one axis simultaneously because it would affect child mesh world orientation. Now, when we call the `rotateTo()` function, we reset rotations on x and z axes to zero, thus preventing rotation anomalies caused by parallel rotations around the other two axes as well.

Also, inside the `rotateTo()` method, we subtract -90 degrees from the left wheel steer angle and subtract 90 degrees from the right one. This is the hack too. Due to the incorrect default y-rotation of the wheels container which has got an offset of 90 degrees, we need to execute this subtraction in order to remove that offset in runtime. The alternative is to open the model in the modeling program and readjust the local rotations of the wheels.

You should have noticed that for wheels rolling, we used the `yaw()` method that by default rotates the object around its local y-axis, whereas it is logical that the rolling is usually z-axis rotation. Normally, if your object's local position is aligned correctly with the global coordinate system, you should indeed use the `roll()` method. In this particular example, the default position of a wheel's mesh container is not aligned to the Away3D coordinate system. So, in order to compensate for this, we use `yaw()`. In fact the presumed local z-axis of each child wheel mesh points is its y direction and because the container's y is aligned with world's z, the wheels appear to be positioned right in the scene.

That is basically all you should know in order to get your physical car on the road. Remember, the most difficult task, as you can see from the previous example, is to adjust the JigLib physics configuration to your car model. This is usually achieved by continuous tweaking and testing. It pays to be patient in this case as car behavior depends on the amount of time you have invested into tuning it.

See also

Chapter 4, Fun by Adding Interactivity, the recipe: Creating a controllable non-physical car.

Morphing Away3D geometry with AS3DMOD

The AS3DMOD library is quite popular between Flash 3D developers. In fact, the library is the smallest of all the rest we discussed in this chapter. It contains a set of classes known as modifiers whose purpose is to deform the geometry of 3D objects. Some of them, such as Bend and Skew, modify the geometry in the way that their names imply, others such as Perlin or Cloth add cool physical behaviors to mesh surface like wavering banner in wind.

 AS3DMOD also includes some unique modifiers that don't affect mesh but allow adjusting different properties of a 3D object. These are `Pivot` and `Wheel` modifiers. `Pivot` allows you to manipulate a model's pivot position, whereas `Wheel` is designed to help solve orientation issues related to steering and rolling of the car wheel. (See the recipe *Creating a controllable non-physical car* in Chapter 4 where we fix this problem manually when creating an interactive car).

In this example, we will play around with several modifiers. Actually, we will develop a mini AS3DMOD explorer where we can switch between four different modifiers. The setup of most of them is pretty similar. Once you go through this recipe, you can easily master the rest of them.

Getting ready

Set up a new Away3D scene class extending `AwayTemplate`.

Make sure you download and connect to your project, the latest AS3DMOD build, which you can download from the Google code repository at <http://code.google.com/p/as3dmod/>.

In this example, we make use of several GUI controls which are part of a Flash components library developed by Keith Peters and called **MinimalComps**. Go and get it for free from <http://www.minimalcomps.com/>. Put the .swc into your SWC folder or connect it directly to your project.

How to do it...

With the help of the following code, we are going to set up a user interface to control several AS3DMOD modifiers whose purpose is to deform a plain primitive in various ways:

`AS3MODDemo.as`:

```
package
{
    public class AS3MODDemo extends AwayTemplate
    {
        private var _mat1:ShadingColorMaterial;
        private var _dirLight:DirectionalLight3D;
        private var _plane:Plane;
        private var _stack:ModifierStack;
        private var _skewM:Skew;
        private var _twistM:Twist;
        private var _bendM:Bend;
```

```

private var _perlinM:Perlin;

private var _radButtonBend:RadioButton;
private var _radButtonSkew:RadioButton;
private var _radButtonPerlin:RadioButton;
private var _radButtonTwist:RadioButton;
private var _slider:Slider;
private var _modType:String="";
private var _canRender:Boolean=true;
public function AS3MODDemo()
{
    super();
    initLight();
    initGUI();
    initModifiers();
    applyModifier("bend");
}
override protected function initMaterials() : void{
    _mat1=new ShadingColorMaterial(0x229933);
}
override protected function initGeometry() : void{
    resetGeometry();
}
override protected function onEnterFrame(e:Event) : void{

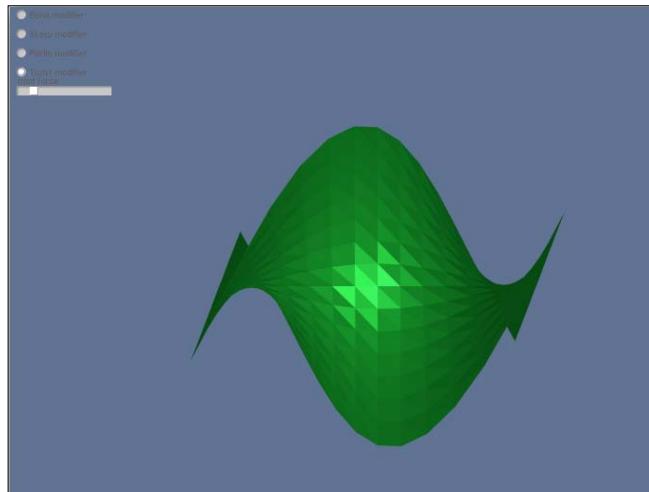
    if(_canRender){
        _stack.apply();
    }
    super.onEnterFrame(e);
}
private function initGUI():void{
    _slider=new Slider("horizontal",this,20,160,onSliderMove);
    _slider.minimum=0;_slider.maximum=5;_slider.tick=0.2;
    var l1:Label=new Label(this,_slider.x,_slider.y-15,"mod force");
    _radButtonBend=new RadioButton(this,20,80,"Bend
modifier",true,onRadioSelect);
    _radButtonSkew=new RadioButton(this,20,100,"Skew
modifier",false,onRadioSelect);
    _radButtonPerlin=new RadioButton(this,20,120,"Perlin
modifier",false,onRadioSelect);
    _radButtonTwist=new RadioButton(this,20,140,"Twist
modifier",false,onRadioSelect);
    _radButtonBend.groupName=_radButtonPerlin.groupName=_radButtonSkew.groupName=_radButtonTwist.groupName="gr1";
}

```

```
private function initModifiers():void{
    _bendM=new Bend(0,0.5);
    _skewM=new Skew(90);
    _twistM=new Twist();
    _twistM.vector=new Vector3(1,0,0);
    _perlinM=new Perlin();
    _perlinM.speedX=1;
    _perlinM.speedY=1;
    _perlinM.force=2;
}
private function onRadioSelect(e:MouseEvent):void{
    switch (e.target){
        case _radButtonBend:
            _slider.minimum=0;
            _slider.maximum=5;
            applyModifier("bend");
            break;
        case _radButtonSkew:
            applyModifier("skew");
            _slider.minimum=0;
            _slider.maximum=360;
            break;
        case _radButtonPerlin:
            applyModifier("perlin");
            _slider.minimum=0;
            _slider.maximum=5;
            break;
        case _radButtonTwist:
            applyModifier("twist");
            _slider.minimum=0;
            _slider.maximum=10;
            break;
    }
}
private function onSliderMove(e:Event):void{
    if(_modType=="bend"){
        _bendM.force=e.target.value;
    }else if(_modType=="skew"){
        _skewM.force=e.target.value;
    }else if(_modType=="perlin"){
        _perlinM.force=e.target.value
    }else{
        _twistM.angle=e.target.value;
    }
}
```

```
        }
        private function applyModifier(modType:String) :void{
            _modType=modType;
            resetGeometry();
            if(modType=="bend"){
                _stack.clear();
                _stack.addModifier(_bendM);
            }else if(modType=="perlin"){
                _stack.clear();
                _stack.addModifier(_perlinM);
            }else if(modType=="skew"){
                _stack.clear();
                _stack.addModifier(_skewM);
            }else{
                _stack.clear();
                _stack.addModifier(_twistM);
            }
        }
        private function resetGeometry():void{
            _canRender=false;
            if(_plane){
                _view.scene.removeChild(_plane);
                _plane=null;
                _stack=null;
            }
            _plane=new Plane({width:200,height:200,material:_mat1,segmentsW:14,segmentsH:14,bothsides:true});
            _plane.ownCanvas=true;
            _plane.yUp=false;
            _plane.position=new flash.geom.Vector3D(0,0,500);
            _view.scene.addChild(_plane);
            _stack=new ModifierStack(new LibraryAway3d(),_plane);
            _canRender=true;
        }
        private function initLight():void{
            _dirLight=new DirectionalLight3D();
            _dirLight.brightness=12;
            _view.scene.addLight(_dirLight);
            _dirLight.direction=_plane.position.add(new flash.geom.Vector3D(0,100,0));
        }
    }
}
```

Here is our AS3DMOD program in action:



How it works...

The principle of AS3DMOD initialization is pretty simple. The steps resemble the modifiers from 3dsMax. The starting point for this is to define a modifier stack which would virtually hold all the future added modifiers. We set up the stack inside the `resetGeometry()` method which also serves us to set and reset a `Plane` primitive to which we apply our modifiers:

```
_stack=new ModifierStack(new LibraryAway3d(),_plane);
```

`ModifierStack` constructor requires the relevant library plug-in, which is, in our case, `LibraryAway3d` and the 3D geometry object to work on. Now let's go to the method `initModifiers()` and see how we define them:

```
private function initModifiers():void{
    _bendM=new Bend(0,0.5);
    _skewM=new Skew(90);
    _twistM=new Twist();
    _twistM.vector=new Vector3(1,0,0);

    _perlinM=new Perlin();
    _perlinM.speedX=1;
    _perlinM.speedY=1;
    _perlinM.force=2;

}
```

In the preceding block, we defined `Bend`, `Skew`, `Twist`, and `Perlin` modifiers.

[ We are not going to explain the parameters of each modifier, but in most cases, it is the force factor of modifier influence as well as several more properties which vary between different modifiers, and you can easily learn what they do by looking into AS3DMOD documentation.]

Next, we set up a GUI using the `MinimalComps` components for switching between our modifiers. We create four radio buttons—one for each modifier and one slider which is going to control the force of their impact on a mesh surface. Each radio button is assigned an event handler called `onRadioSelect()`. When it is triggered, it checks which radio button is the target, then inside an appropriate case block, we first adjust the `_slider` minimum and maximum range for each modifier force property as the amount of force that needs to be delivered to different modifiers varies. Then we trigger the `applyModifier()` function to which we pass a string indicating what type of modifier we want to apply.

Inside the `applyModifier()` method we first clean the current scene geometry and replace it with the new one calling the already familiar `resetGeometry()` method. The reason we do it each time we switch modifiers is that at the time of writing, when removing the modifiers from the stack the deformation caused by it doesn't disappear. Let's move on. Now we compare inside whether statements of the modifier type match the `modType` string we passed into the function and add a new modifier (let's say `modType` is "bend") to the stack with the following line:

```
_stack.addModifier(_bendM);
```

The last step in all these processes is to apply the modifier to the mesh after it has been added to the stack. We do it continuously inside the `onEnterFrame()` method calling:

```
_stack.apply();
```

We call it on each frame so that the changes of the mesh caused by the current modifier can be seen instantly. Also note that `_stack.apply()` is wrapped with the `if (_canRender)` condition. The reason for this is that when we switch to a new modifier, as you may remember, we reset the scene geometry as well as the `_stack` instance. We have to stop calling the `apply()` method before the new `_stack` is created otherwise we will get null object exception. You can take a look at the `resetGeometry()` method where we first set `_canRender` to false stopping by means of this `apply()` call inside `onEnterFrame()` then after we re-initialized the stack, we reset it to true once again.

See also

In *Chapter 3, Animating the 3D World* the following recipe: *Morphing objects*.

Exploding particles with FLINT

There are quite a few Flash particle system engines on the market available for free, and to be honest with you, FLINT is not the best in terms of performance. However, the rest of the advantages it has over others make it definitely the best, and at the time of writing, it is the only choice for 3D particles simulations. The engine developed and maintained by Richard Lord is divided into 2D and 3D sub-libraries, as you can see by the source SWCs files, Flint2d.swc and Flint3D.swc. The engine is capable of producing real time particles integrating diverse complex physical behaviors and effects. Flint3D contains modules for plugging in 3D engines and you may have guessed Away3D is one of them.

In the following recipe, you will learn how to set up the FLINT engine and how to incorporate it into our Away3D environment. Besides, we will build a particle system which is going to use Away3D primitives as particles within an explosion effect powered by FLINT. So let's do it.

Getting ready

Set up a new Away3D scene class extending AwayTemplate.

Make sure you download and connect to your project the latest FLINT library build, which you can get from <http://flintparticles.org/source-code>.

Go to this chapter's source code directory and copy the dofText.png image from the assets/models/images folder into your project assets location. We are using this bitmap as a texture for our particles.

How to do it...

The following program creates a spherical field of particles which behaves like an explosion dispersing accelerated particles, which are instances of Away3D Sprite3D, in all directions:

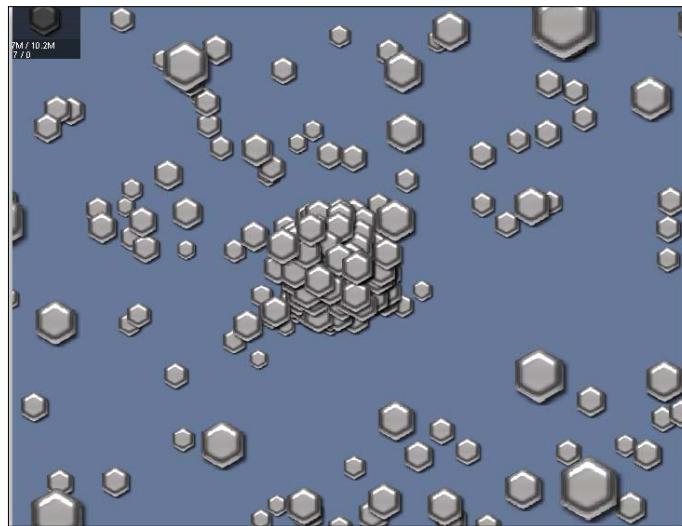
FlintDemo.as

```
public class FlintDemoV3 extends AwayTemplate
{
    [Embed(source="assets/models/images/dofText.png")]
    private var SpriteTexture:Class;
    private var _pRender:Away3DRenderer;
    private var _emitter:Emitter3D;
    private var _camDummy:ObjectContainer3D;
    private var _bitMat:Material;
    private var _hoverCam:HoverCamera3D;
    private var _oldMouseX:Number=0;
    private var _oldMouseY:Number=0;
```

```
private static const EASE_FACTOR:Number=0.5;
public function FlintDemoV3()
{
    super();
    setHoverCamera();
}
override protected function initMaterials() : void{
    _bitMat=new BitmapMaterial(Cast.bitmap(new SpriteTexture()));
}
override protected function initGeometry() : void{
    _camDummy=new ObjectContainer3D();
    _camDummy.position=new Vector3D(0,0,900);
    initFlint();
}
override protected function onEnterFrame(e:Event) : void{
    super.onEnterFrame(e);
    _hoverCam.panAngle = (stage.mouseX - _oldMouseX)*EASE_FACTOR ;
    _hoverCam.tiltAngle = (stage.mouseY - _oldMouseY)*EASE_FACTOR ;
    _hoverCam.hover();
}
private function initFlint():void{
    _pRender=new Away3DRenderer(_view.scene);
    _emitter=new Emitter3D();
    _emitter.counter=new Pulse(1.5,300);
    _pRender.addEmitter(_emitter);
    var awayParticle1:A3DObjectClass=new A3DObjectClass(Sprite3D,{material: _bitMat,width:3,height:3,scaling:1});///material:_bitMat,
    _emitter.addInitializer(awayParticle1);
    _emitter.addInitializer(new Position(new SphereZone(new Vector3D(0,0,900),100,20)));
    _emitter.addInitializer(new Velocity(new SphereZone(new Vector3D(0,0,0),500,50)));
    _emitter.addInitializer(new Lifetime(1,3));
    _emitter.addAction(new Move());
    _emitter.addAction(new Age());
    _emitter.addAction(new Accelerate(new Vector3D(7,7,7)));
    _emitter.addAction(new RandomDrift(1500,1500,1500));
    _emitter.start();
}
private function setHoverCamera():void{
    _hoverCam=new HoverCamera3D();
    _view.camera=_hoverCam;
    _hoverCam.target=_camDummy;
    _hoverCam.distance = 1600;
```

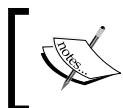
```
    _hoverCam.maxTiltAngle = 90;
    _hoverCam.minTiltAngle = 0;
    _hoverCam.wrapPanAngle=true;
    _hoverCam.steps=16;
    _hoverCam.yfactor=1;
}
}
}
```

Here are our resulting particles, powered by FLINT:



How it works...

We are going to cover the basic mechanics and principles of FLINT particles setup in parallel to the explanation of the preceding code.



We have not gone into the other libraries in depth. For more information, you can refer to the FLINT engine's website at <http://flintparticles.org>.

The core classes of the FLINT engine (3D version) are Emitter3D, and as we work with Away3D, Away3DRenderer. These two are the starting point for each FLINT application. Emitter3D is the particle source and Away3DRenderer serves as a bridge between FLINT and Away3D APIs. As you can see inside the `initFlint()` function, we initialize Away3DRenderer passing to its constructor the container where the emitter is going to reside:

```
_pRender=new Away3DRenderer(_view.scene);
```

In our case, we set the Away3D scene, but it can be any object extending ObjectContainer3D. Then we initiate our emitter:

```
_emitter=new Emitter3D();
```

The emitter must be assigned a counter. **Counter** controls the rate and type of particles emission. You can check many different counters within the `org.flintparticles.common.counters` package. Here we use the `Pulse` counter, which emits an amount of particles with a defined time interval:

```
_emitter.counter=new Pulse(1.5,100);
```

Then we should register the emitter to our `Away3DRenderer` instance. You can have several emitters:

```
_pRender.addEmitter(_emitter);
```

Next we define a type of particle. In FLINT for Away3D, you have an option to pass geometry as particles or any `DisplayObject` such as bitmap that then is wrapped by FLINT inside the Away3D scene. In this following example, we set `Sprite3D` object as a particle:

```
var awayParticle1:A3DObjectClass=new A3DObjectClass(Sprite3D,{material:_bitMat,width:3,height:3,scaling:1});
```

Note that the second parameter of the `A3DObjectClass` class is an object holding the parameters for `Sprite3D` constructor.

Now comes the crucial stuff without which you will never utilize the power of FLINT. A particle's life cycle is controlled by three types of, let's call them *controllers*. These are initializers, actions, and activities. Think of initializers as properties of the particles and of their behavior during their lifetime. Actions execute the behaviors of the particles based on initializers definitions as well as their unique behaviors throughout the particle's life span. Activities are objects that supply behavior functionality not to particles but to their emitters such as `Emitter3D`. Through the following lines, we define initializers for particles type which we defined previously. Then we set the particle initial position and the particles life time with the following lines:

```
_emitter.addInitializer(awayParticle1);

_emitter.addInitializer(new Position(new SphereZone(new
Vector3D(0,0,900),100,20)));
_emitter.addInitializer(new Velocity(new SphereZone(new
Vector3D(0,0,0),500,50)));
_emitter.addInitializer(new Lifetime(1,3));
```



Note that Lifetime initializer is one of the most important, as without it the particles would never die but continue to accumulate inside the scene slowing down the performance significantly.

Now we want those initializer definitions to get executed over time. Here comes the action! First we set the essential actions:

```
_emitter.addAction(new Move());
_emitter.addAction(new Age());
```

These are responsible for particles rotation, movement, and dying accordingly based on initializers data. Next we add two more actions to enhance the particle's behavior:

```
_emitter.addAction(new Accelerate(new Vector3D(7,7,7)));
_emitter.addAction(new RandomDrift(1500,1500,1500));
```

Accelerate, as its name implies, accelerates the particles velocity over time and RandomDrift adds to each particle an effect of random (Brownian) motion.

Now we are ready to launch. The last line after everything is setup, which you should try never to forget (otherwise it may take you hours to figure out that there are no bugs) is this:

```
_emitter.start();
```

You are free to explore more actions and initializers. Play with them and learn their roles and influence on particles behavior. You can produce truly unbelievable visual effects by combining different types of behaviors and tweaking their properties in the right way.

See also

There is another particle engine which became pretty popular in the last year called **Stardust**. Its support enables you to generate both 2D and 3D particles and supplies Away3D modules just like FLINT. You can get it from <http://code.google.com/p/stardust-particle-engine>.

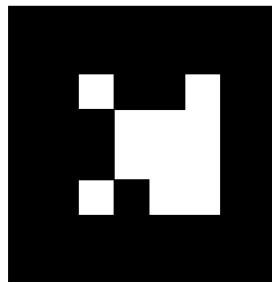
Setting Augmented Reality with FLARToolkit in Away3D

In this recipe, you will be quickly introduced to the augmented reality which has been making quite a lot of buzz in the Flash developers' community in the last couple of years since FLARToolkit has been introduced. The precise definitions of what is **Augmented Reality (AR)** can be found in places such as Wikipedia if you never heard of it, but here it can be described in the most simple words as a technology that allows you to see virtual stuff that is computer graphics in the real world via the lens of your webcam.

The AR API has not originated in ActionScript from scratch, but was ported from other languages. The official AS3 API is named `FLARToolkit`, as I mentioned previously. In the following example, we will use an additional library called `FLARManager`, which includes `FLARToolkit` as well as plugins for most Flash 3D engines, whereas Away3D is one of them. I strongly suggest working with `FLARManager` as it extremely simplifies the setup of FLAR allowing you to set up an AR scene just in a couple of minutes. So let's have some fun with AR!

Getting ready

1. First, go to <http://words.transmote.com> and download the latest version of `FLARManager`. It includes all you need for setting AR (except the Away3D source). Connect the source to your ActionScript project.
2. Now you need to print a marker. A marker is a fancy sign which is detected and then tracked by FLAR and its role is to serve as an anchor point for the rendered 3D object. Inside the `FLARManager` root folder, navigate to the `trunk/resources/flarToolkit/patterns` folder and print one or more of the available `png` marker images:



3. Next, we need to put some FLAR configuration files into our project root. Copy the `trunk/resources` folder and paste it into the root directory of your project. The folder contains several important files such as `flarConfig.xml` which holds all the important configurations of FLAR engine and we are going to address it shortly.

Now open `flarConfigs.xml`, which is found inside the `resources/flar` directory. The important XML node that you will edit frequently is `<flarSourceSettings>`. It contains the following attributes:

```
<flarSourceSettings
    sourceWidth="640"
    sourceHeight="480"
    displayWidth="800"
    displayHeight="600"
    framerate="30"
    trackerToSourceRatio="0.5" />
```

Set the values for the first four attributes as they appear in the preceding block.

As FLARToolkit is memory intensive, it is suggested to decrease the camera source dimension as well as those of the viewport in order to gain more FPS. It is best to set `sourceWidth` and `height` to be less than `displayWidth/displayheight`.



Notice that the resolutions of video input and visual quality doesn't change significantly if you don't reduce it lower than 640x480 when the display dimensions are 800x600. However, very low values will make the video look pixilated.

4. Eventually, set up a basic Away3D scene extending `AwayTemplate`. We also need, once again, our best friend Spy model which you can get together with its texture inside the `assets/models/` folder of this chapter's source code. Paste it into the `assets/models/` folder of your project. Now we can move to the application code.

How to do it...

```
FLARDemo.as
package
{
    public class FLARDemo extends AwayTemplate
    {
        [Embed(source="assets/models/spyObjReady.3ds", mimeType="application/octet-stream")]
        private var SpyModel:Class;
        [Embed(source="assets/models/spy_flat_blu.png")]
        private var SpyTexture:Class;
        private var _flarManager:FLARManager;
        private var _spy3ds:Object3D;
        private var _bitMat:BitmapMaterial;
        private var _flarMarker:FLARMarker;
        private var _flarCamera:FLARCamera_Away3D;
        private var _markerExists:Boolean=false;
        public function FLARDemo()
        {
            super();
            _view.camera.lens=new PerspectiveLens();

        }
        override protected function initGeometry() : void{
            parse3ds();
            initFLAR();
        }
    }
}
```

```
        }
        override protected function initMaterials() : void{
            _bitMat=new BitmapMaterial(Bitmap(new SpyTexture()).bitmapData);
        }
        override protected function onEnterFrame(e:Event) : void{
            super.onEnterFrame(e);
            if(_flarMarker&&_markerExists){

                _spy3ds.transform =AwayGeomUtils.convertMatrixToAwayMatrix(_
flarMarker.transformMatrix);
                _spy3ds.rotate(new Vector3D(1,0,0),180);
            }
        }
        private function parse3ds():void{

            var max3ds:Max3DS=new Max3DS();
            max3ds.centerMeshes=true;
            _spy3ds=max3ds.parseGeometry(SpyModel)as Object3D;
            _view.scene.addChild(_spy3ds);
            _spy3ds.materialLibrary.getMaterial("spy_red").material=_bitMat;
            _spy3ds.scale(0.2);
            _spy3ds.centerPivot();
        }
        private function initFLAR():void{
            _flarManager = new FLARManager( ".../resources/flar/flarConfig.
xml", new FLARToolkitManager(), this.stage);
            this.addChildAt(Sprite(_flarManager.flarSource),0);
            var fpsStats:FramerateDisplay= new FramerateDisplay();
            this.addChild(fpsStats);
            initFLARListeners();
        }
        private function initFLARListeners():void{
            _flarManager.addEventListener(ErrorEvent.ERROR,
onFlarInitError);
            _flarManager.addEventListener(FLARMarkerEvent.MARKER_
ADDED,onMarkerAdd);
            _flarManager.addEventListener(FLARMarkerEvent.MARKER_
UPDATED,onMarkerUpdate);
            _flarManager.addEventListener(FLARMarkerEvent.MARKER_
REMOVED,onMarkerRemove);
            _flarManager.addEventListener(Event.INIT, onFlarManagerReady);
        }
        private function onFlarInitError(e:ErrorEvent):void{
            trace(e);
        }
    }
```

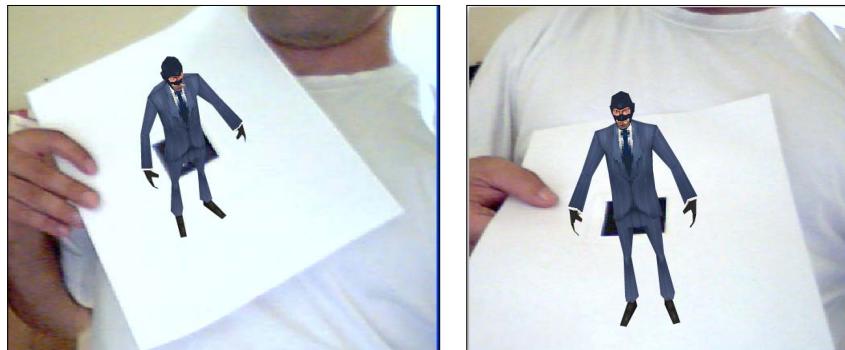
```
private function onMarkerAdd(e:FLARMarkerEvent):void{
    _flarMarker=e.marker;
    _markerExists=true;
}
private function onMarkerUpdate(e:FLARMarkerEvent):void{
    _markerExists=true;
}
private function onMarkerRemove(e:FLARMarkerEvent):void{
    _markerExists=false;
}
private function onFlarManagerReady(e:Event):void{
    _flarManager.removeEventListener(ErrorEvent.ERROR,
onFlarInitError);
    _flarManager.removeEventListener(Event.INIT,onFlarManagerReady);
    _flarCamera=new FLARCamera_Away3D(_flarManager,new
Rectangle(0,0,this.stage.stageWidth, this.stage.stageHeight));
    _view.camera=_flarCamera;
}

}
```

After pasting the code into your application, make sure your webcam is connected and functional.

Run the application. On the application start up, you should receive a Flash player setting dialog which will ask you to allow your local content (SWF) to access the webcam. Click on **allow**.

Now grab your marker, as is done in the following pictures. Wait for few seconds so that the tracker can lock on the marker and here we go! Now the spy is literally in your hands!



Here is our spy standing on the piece of paper I am holding in my hand.

How it works...

We are not going to dive into the internals of FLARToolkit as it is a complex framework based on sophisticated camera pixel data analyzing. For us, the end users, the most important thing is configuration and set up. So as it was said, FlarManager makes our life easy. Here's how it works. First, as usual, we set up a default Away3D scene.

Then we parse our Spy model within the `parse3ds()` function and add it to the scene. Now we proceed with the FlarManager initialization. Inside the `initFLAR()` method, we first create an instance of the `FLARManager` class:

```
_flarManager = new FLARManager( ".../resources/flar/flarConfig.xml",
    new FLARToolkitManager(), this.stage);
```

We pass to its constructor the path of `flarConfig.xml` which we reviewed previously, a new instance of `FlarToolkitManager`, and the reference to the stage:

Next we add the video source object to our `_view`. We use the `addChildAt` zero depth because we need it to be behind `_view` `Sprite` (because, by default, `_view` instantiated before) so that we can see Away3D scene in front of the video.

```
this.addChildAt(Sprite(_flarManager.flarSource), 0);
```

We also add performance info display:

```
var fpsStats:FramerateDisplay= new FramerateDisplay();
this.addChild(fpsStats);
```

Next, we have to define several listeners for `_flarManager`. Inside `initFLARListeners()`, we assign listeners for the following events:

```
ErrorEvent.ERROR,
FLARMarkerEvent.MARKER_ADDED,
FLARMarkerEvent.MARKER_UPDATED,
FLARMarkerEvent.MARKER_REMOVED,
Event.INIT
```

`ErrorEvent` handler is triggered if there is an error during the `_flarManager` initialization. `FLARMarkerEvent.MARKER_ADDED` is handled by the `onMarkerAdd()` function. It is triggered when the marker is caught in the camera by FLAR. When it happens, we retrieve the reference to `FLARMarker` with this line `_flarMarker=e.marker;`.

And assign it to the instance `_flarMarker`, which was created as a member variable. As you will see shortly, `_flarMarker` plays a crucial role in presenting the graphics in the AR. `FLARMarkerEvent.MARKER_UPDATED` is fired while the marker is transformed inside the camera view. Now because many times your paper marker is moving, it gets out of track. When it occurs, `FLARMarkerEvent.MARKER_REMOVED` is dispatched. Inside its event handler `onMarkerRemove()`, we set `_markerExists=false`. This way, we stop assigning non-existent matrix transformation data of the marker to the model till the marker is caught by the tracker again.

Now comes the most important event `Event.INIT`. `flarManager` fires it when it has been successfully initiated. Only after this event can we proceed with the rest of the executions. `Event.INIT` triggers the `onFlarManagerReady()` handler. Here we initialize a camera that replaces generic Away3D's `Camera3D`:

```
_flarCamera=new FLARCamera_Away3D(_flarManager,new Rectangle(0,0,this.stage.stageWidth, this.stage.stageHeight));  
_view.camera=_flarCamera;
```

The use of this camera is essential as it projects the scene objects based on the data supplied by the tracker transformations.

The final step is to update our 3D model transformation according to the marker. We do this inside the `onEnterFrame()` function:

```
_spy3ds.transform =AwayGeomUtils.convertMatrixToAwayMatrix(_  
flarMarker.transformMatrix);
```

`AwayGeomUtils` class helps us to convert the `_flarMarker` matrix to Away matrix, which is then assigned to the `_spy3ds.transform` setter.

The last line is specific to this particular 3D model:

```
_spy3ds.rotate(new Vector3D(1,0,0),180);
```

We need to correct its orientation, which, by default, places the Spy upside down. We fix that by updating the x-axis rotation 180 degrees counter clockwise.

Adding Box2D physics to Away3D objects

`Box2DFlash` is a robust real-time 2D physics engine for the Flash platform. At the time of writing, it is the most optimized and lightweight open source physics engine which is highly popular in the Flash game development community. The engine is a porting from C++ `Box2D` library to which it owes its outstanding performance. Although `Box2D` is a 2D engine—that means it lacks a third dimension, with some hacking it can be used in the 3D world and therefore is a smart choice to incorporate in Away3D projects that require CPU-intensive physics simulations. Of course, it still can't close the gap in terms of 3D functionality with engines such as `WOW` or `JigLib` but currently the last two are just too heavy to manage multiple physical simulations in the 3D world.

In the following example, you will learn only the basics of incorporating `Box2DFlash` with Away3D. We are limited with page count and the topic is too broad to cover more advanced implementations. You can learn more advanced topics on using `Box2D` at the following websites:

<http://www.box2dflash.org/docs/>
<http://www.emanueleferonato.com>
<http://pv3d.org/>

Getting ready

1. Download the Box2dFlash version 2.0.2 from here: <http://sourceforge.net/projects/box2dflash/develop>, or you can try the latest release at your own risk. Connect the library to your project.

At the time of writing, the latest release is 2.1a which is alpha release. This version currently works very badly with Away3D. Also, its API is far from being complete (from the words of the developer). Therefore, we have decided to use the older stable version for the test case and if you find, while reading this book, that 2.1 will become a milestone release, you can try it out. In this case, there will be no escape from moderating the existing code to the new version changes.

2. In this example, we make use of GUI controls which are part of a Flash components library developed by Keith Peters and called **MinimalComps**. Go and get it for free from here: <http://www.minimalcomps.com/>. Put the .swc file into your swc folder or connect it directly to your project.
3. Create a new ActionScript3.0 class that extends AwayTemplate, name it Box2dDemo, and paste into it the next code.

How to do it...

In this program, we will set up a basic Box2D physics environment which we then fill with different Away3D geometry-based rigid bodies:

Box2dDemo.as

```
package
{
    public class Box2dDemo extends AwayTemplate
    {
        private const WORLD_SCALE:Number = 30;
        private const WORLD_WIDTH:Number = 800;
        private const WORLD_HEIGHT:Number = 600;
        private var _physWorld:b2World;
        private const ITERATION:int = 10;
        private const STEP:Number = 1.0/30.0;
        private var _pMat:WhiteShadingBitmapMaterial;
        public function Box2dDemo()
        {
            super();
            _cam.z=-800;
            _view.camera.lens=new PerspectiveLens();
        }
    }
}
```

```
override protected function initGeometry() : void{
    initPhysicsWorld();
    initGUI();
    initWorldBounds();
    createBodies();
    initLights();
}
override protected function initMaterials() : void{
    var bdata:BitmapData=new BitmapData(32,32);
    bdata.perlinNoise(23,23,13,34324,true,true);
    _pMat=new WhiteShadingBitmapMaterial(bdata);
}
private function initLights():void{
    var pLight:PointLight3D=new PointLight3D();
    pLight.radius=250;
    pLight.color=0x126789;
    pLight.brightness=2;
    pLight.position=new Vector3D(-100,-50,0);
    _view.scene.addLight(pLight);
}
private function initGUI():void{
    var but:PushButton=new PushButton(this,10,100,"reset",onResetMousePress);
}
private function onResetMousePress(e:MouseEvent):void{
    for (var bb:b2Body = _physWorld.m_bodyList; bb; bb = bb.m_next) {
        if (bb.m(userData is Object3D&&Object3D(bb.m.userData).name!="floor"))
        {
            var obj3d:Object3D=bb.m.userData;
            _view.scene.removeChild(obj3d);
            obj3d=null;
            _physWorld.DestroyBody(bb);
        }
    }
    ///create the bodies anew///
    createBodies();
}
//setting floor anf walls for the bodies to react against//
private function initWorldBounds():void{
    ///////////////////////////////floor/////////////////////////////
    var floorSD:b2PolygonDef = new b2PolygonDef();
    var floorBD:b2BodyDef = new b2BodyDef();
```

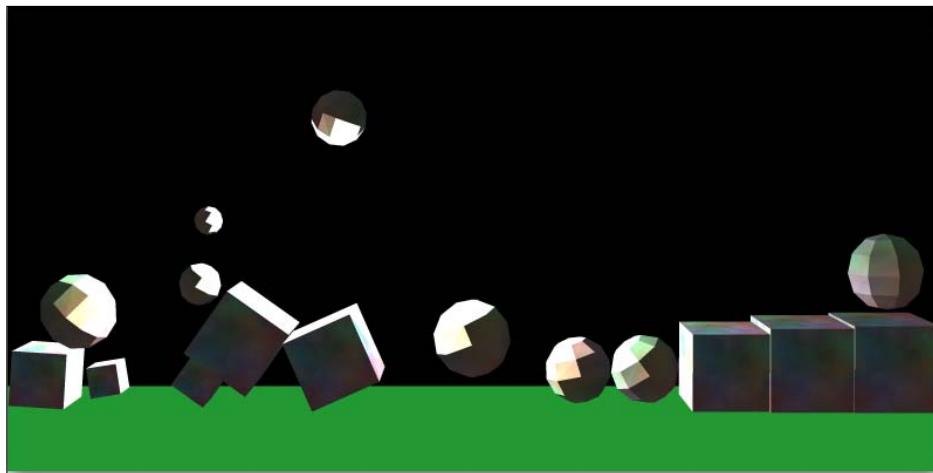
```
    floorSD.SetAsBox( (WORLD_WIDTH+40)/WORLD_SCALE/2, 100/WORLD_
SCALE) ;
    floorBD.position = new b2Vec2(WORLD_WIDTH/WORLD_SCALE/2, (WORLD_
HEIGHT+40)/WORLD_SCALE) ;
    var floorBody:b2Body= _physWorld.CreateBody(floorBD) ;
    floorBody.CreateShape(floorSD) ;
    floorBody.SetMassFromShapes() ;
    var floor3d:Cube=new Cube({material:new ColorMaterial(0x229933),
width:WORLD_WIDTH,height:200}) ;
    floor3d.name=floor;
    floor3d.ownCanvas=true;
    floor3d.pushback=true;
    _view.scene.addChild(floor3d) ;
    floorBody.m(userData=floor3d;
    ////////////////////walls///////////////
    var wallSD:b2PolygonDef = new b2PolygonDef() ;
    var wallBD:b2BodyDef = new b2BodyDef() ;

    wallSD.SetAsBox(100/WORLD_SCALE, (WORLD_HEIGHT+40)/WORLD_
SCALE/2) ;
    // Left Wall///////////////////////////
    wallBD.position = new b2Vec2(40 / WORLD_SCALE, WORLD_HEIGHT/
WORLD_SCALE/2) ;
    var wallBody:b2Body = _physWorld.CreateBody(wallBD) ;
    wallBody.CreateShape(wallSD) ;
    // Right Wall///////////////////
    wallBD.position = new b2Vec2((WORLD_WIDTH+60) / WORLD_SCALE,
WORLD_HEIGHT/WORLD_SCALE/2) ;
    wallBody = _physWorld.CreateBody(wallBD) ;
    wallBody.CreateShape(wallSD) ;
}
private function initPhysicsWorld():void {
    var bounds:b2AABB = new b2AABB();
    bounds.lowerBound = new b2Vec2( 0, 0 ) ;
    bounds.upperBound = new b2Vec2(WORLD_WIDTH/WORLD_SCALE,WORLD_
HEIGHT/WORLD_SCALE) ;
    var grav:b2Vec2 = new b2Vec2(0, 8) ;
    _physWorld = new b2World(bounds, grav,true) ;
}
private function createBodies():void
{
    for (var i:Number = 0; i < 8; i++)
{
```

```
        var radius:Number = Math.random() * 25 + 5;
        var circleBD:b2BodyDef = new b2BodyDef();
        circleBD.position = new b2Vec2(Math.random() * WORLD_WIDTH /
WORLD_SCALE, Math.random() * 50 /WORLD_SCALE);
        var circleBody:b2Body = _physWorld.CreateBody(circleBD);
        var circleSD:b2CircleDef = new b2CircleDef();
        circleSD.radius = radius/WORLD_SCALE;
        circleSD.density = 1;
        circleSD.friction = .7;
        circleSD.restitution = .7;
        circleBody.CreateShape(circleSD);
        circleBody.SetMassFromShapes();
        var sp:Sphere = new Sphere({radius:radius,material:_pMat});
        _view.scene.addChild(sp);
        circleBody.m(userData = sp;
    }
//////////cubes///////////
for (var c:Number = 0; c < 8; ++c)
{
    var bwidth:Number =Math.random() * 20 + 10;
    var boxBD:b2BodyDef = new b2BodyDef();
    boxBD.position = new b2Vec2(Math.random() * WORLD_WIDTH /
WORLD_SCALE, Math.random() * 50 /WORLD_SCALE);
    var boxBody:b2Body = _physWorld.CreateBody(boxBD);
    var boxSD:b2PolygonDef = new b2PolygonDef();
    boxSD.SetAsBox(bwidth/WORLD_SCALE, bwidth/WORLD_SCALE);
    boxSD.density = .6;
    boxSD.restitution = .3;
    boxSD.friction = .5;
    boxBody.CreateShape(boxSD);
    boxBody.SetMassFromShapes();
    var cube:Cube = new Cube({material:_pMat,width: bwidth *
2,height: bwidth * 2,depth: bwidth * 2});
    _view.scene.addChild(cube);
    boxBody.m(userData = cube;
}
}
override protected function onEnterFrame(e:Event):void
{
    super.onEnterFrame(e);
    _physWorld.Step(STEP,ITERATION);
    for (var bb:b2Body = _physWorld.m_bodyList; bb; bb = bb.m_next)
    {
        if (bb.m(userData is Object3D)
        {
```

```
        bb.m(userData.x = bbGetPosition().x * WORLD_SCALE - (WORLD_
WIDTH+100) * .5;
        bb.m(userData.y = -bbGetPosition().y* WORLD_SCALE + (WORLD_
HEIGHT+100) * .5;
        bb.m(userData.rotationZ = -bbGetAngle() * (180/Math.PI));
    }
}
}
}
```

As can be seen from the following image, although the primitive's positions are constrained to x and z axes, they are wrapped within a Box2D rigid body, which provides them with the sophisticated physics behavior:



How it works...

The Box2D code may look scary at first glance, but when you get the paradigm, you will see that it is quite simple. The API makes heavy use of factories and some instantiations such as those of rigid bodies include several steps which add significantly to the overall code volume. Now let's discuss some important steps from the preceding program.

The first very important thing you have to understand is unit conversion. The Box2DFlash engine works with **meters-kilogram-second (MKS)** units, whereas the graphics dimensions and positions belong to pixel realm. The conversion factor is 1 meter=30 pixels. If we assign pixel values to the dimensions of rigid body shapes, we will get extremely incorrect simulations, because they are going to be interpreted as meters making a box with the height of 150 pixels treated by the engine as skyscraper! In our program, `WORLD_SCALE` constant serves us a conversion factor.

Let's move on. First we set up a physics world inside the `initPhysicsWorld()` function. Here we first define the world's boundaries using the `b2AABB` class:

```
var bounds:b2AABB = new b2AABB();
bounds.lowerBound = new b2Vec2( 0, 0 );
bounds.upperBound = new b2Vec2(WORLD_WIDTH/WORLD_SCALE,WORLD_
HEIGHT/WORLD_SCALE);
var grav:b2Vec2 = new b2Vec2(0, 8);
```

Always make sure that the boundaries are bigger than the space you will use for simulations, because when the bodies contact with the world boundaries, they freeze forever. Next, we define a gravity vector and initiate the physical world passing to `b2World` the world boundaries, the gravity vector we created before. The third parameter `doSleep` means that the engine will stop simulating the bodies which accomplished their positions. This saves us needless CPU cycles:

```
var grav:b2Vec2 = new b2Vec2(0, 8);
physWorld = new b2World(bounds, grav,true);
```

Now, back to the `initGeomtry()` method. The next function we call is `initGUI()`, which we put aside for a while. Next is `initWorldBounds()`. Inside this method, we set floor and walls to which the bodies which we will create soon should react on impact. Each wall and the floor are rigid bodies too, but they remain static as we don't define for them properties such as density, friction, and restriction. Let's see the steps required to set up a rigid body by example of floor setup. Any rigid body needs two definitions before it can be constructed. These are shape and body definitions. We define the shape definition for the floor with this line, setting its geometric shape to rectangle:

```
var floorSD:b2PolygonDef = new b2PolygonDef();
floorSD.SetAsBox((WORLD_WIDTH+40)/WORLD_SCALE/2, 100/WORLD_SCALE);
```

Next we set the body definitions which we need to define a position of the body:

```
var floorBD:b2BodyDef = new b2BodyDef();
floorBD.position = new b2Vec2(WORLD_WIDTH/WORLD_SCALE/2, (WORLD_
HEIGHT+40)/WORLD_SCALE);
```

Now we can create our floor body by calling `_physWorld.CreateBody(floorBD)` to which we pass the floor body definition variable. Now when we have the instance of the body, we set its shape:

```
floorBody.CreateShape(floorSD);
```

Also, we need to set its mass. The easiest way to do it is by calling:

```
floorBody.SetMassFromShapes();
```

The engine will determine the mass relative to the shape's size.

Now the body exists, but it lacks visual representation. We create a cube primitive and assign it to `floorBody` to serve its graphics:

```
var floor3d:Cube=new Cube({material:new ColorMaterial(0x229933),width:  
WORLD_WIDTH,height:200});  
floor3d.name=floor;  
floor3d.ownCanvas=true;  
floor3d.pushback=true;  
_view.scene.addChild(floor3d);  
floorBody.m(userData=floor3d;
```

The same process is used to set the walls, although we omitted their graphics to save page space. Now when we have the world's container set, it is time to add dynamic bodies. We do it inside the `createBodies()` function. The only difference here from the process described previously is that for each body, we also define density, friction, and restitution properties which are essential for moving bodies.

Well it's time is to set it all in motion. Go to the `onEnterFrame()` method. Here we call `physWorld.Step(STEP, ITERATION)` on each frame. Step method requires two parameters. First is a physical time step which is used to update the simulations. The range between 1/30 to 1/60 (30Hz-60Hz) is optimal. The second argument is constraint solver, which updated all constraints in the simulations and although in this example we don't have any, the parameter is required (more on this in Box2DFlash docs).

Now think about it this way—when the simulation begins, the rigid bodes are updated by the engine by default based on forces applied to them. But the problem is that the graphic object which we assigned to each of them is ignored by default. What we need to do is to extract the position and rotation values from the bodies, convert them to our graphics world units (in this case, Away3D scene), and assign them explicitly to the geometry. This `for` loop does the job:

```
for (var bb:b2Body = _physWorld.m_bodyList; bb; bb = bb.m_next)  
{  
    if (bb.mUserData is Object3D)  
    {  
        bb.mUserData.x = bbGetPosition().x * WORLD_SCALE - (WORLD_  
WIDTH+100) * .5;  
        bb.mUserData.y = -bbGetPosition().y* WORLD_SCALE + (WORLD_  
HEIGHT+100) * .5;  
        bb.mUserData.rotationZ = -bbGetAngle() * (180/Math.PI);  
    }  
}
```

Notice that we convert not only from meters to pixels multiplying by `WORLD_SCALE`, but also we shift the coordinate system to match Away3D world by subtracting from x `(WORLD_WIDTH+100)*5` and adding to y `(WORLD_HEIGHT+100)*5`. The additional 100 pixels is a correction to compensate a visual offset of the world caused by camera distance from the scene.

The last thing we will see is how to clean up the physical world from the bodies and the scene from the meshes simultaneously. Inside `initGUI()`, we set a button called "Reset". When the user hits it, `onResetMousePress()` handler is triggered. Inside it we iterate through all the bodies. We check that the user data of the bodies we are going to remove is of `Object3D` type, so not to remove the walls. We also check that the name of those objects is not "floor", which is assigned to the floor only in order to separate it from the rest of the 3D objects which are to be removed. If both parts of the statement are true, we first remove the `Away3D` object which is wrapped by the corresponding body:

```
var obj3d:Object3D=bb.m(userData;  
view.scene.removeChild(obj3d);  
obj3d=null;
```

And we delete the body itself:

```
physWorld.DestroyBody(bb);
```

Having finished the cleaning, we deploy the bodies anew by calling `createBodies()` once again.

That's all for Box2DFlash basics.

There's more...

Read the tutorials and documentations from the links in the beginning of the recipe to learn more advanced techniques to create amazing physical simulations in `Away3D` using `Box2DFlash`.

11

Away3D Lite

In this chapter, we will cover:

- ▶ Setting up Away3D Lite using templates
- ▶ Importing external models in Away3D Lite
- ▶ Manipulating geometry
- ▶ Making 2D shapes appear three dimensional by using Sprite3D
- ▶ Managing Z-Sorting by automatic sorting and using layers
- ▶ Creating virtual trackball
- ▶ Writing Away3D Lite applications with haXe

Introduction

Away3D Lite is a compact version of the major Away3D engine which we use in most of these chapters. Away3D Lite was designed to deliver the best possible performance to the developer and the simplicity of usage. Indeed, at the time of writing, Away3D Lite is the most lightweight and the strongest in terms of performance engine on the market of open source Flash-based counterparts. The supreme performance is due to full integration of the native Flash Player 10 3D API. According to the Away3D team, Away3D Lite is four times faster than the regular version of the engine. While the Away3D Flash Player 10 version makes only partial use of it (Vector3D, Matrix3D), Away3D Lite takes advantage of the whole API processing all the expensive calculations in the Flash Player. Another important feature is the project weight differences. The size of the Away3D Lite library is only around 25 KB and that, unfortunately, is due to the limited amount of features Away3D Lite offers, which is significantly less than that of Away3D. Moreover, while it is quite enough for simple projects, if you plan something complex, you will have to go for the "Big Cannon", that is, Away3D. Having said that, it is worth noticing that most of the features found in Away3D can be ported into Away3D Lite once you understand the engine's internals.

Enough talking, let's get our hands dirty with Away3D Lite!

Setting up Away3D Lite using templates

Basically, setting up an Away3D Lite scene is no different from one for Away3D. In addition to manual instantiation of all the essential classes to run an Away3D Lite application, you can utilize built-in templates which perform the setup routine, and you only need to extend these, similar to AwayTemplate we use throughout this book.

Getting ready

Make sure you connected Away3D Lite source to your project.

Create a new ActionScript file, name it as `AwaySetupDemo` and make it extend `FastTemplate`.

How to do it...

The following application is one of the shortest you can find inside this book. As `FastTemplate` performed all the setup behind the curtain, all that we should do is to test the program. For this, we add some geometry just to make sure it is rendered and our setup is fine:

`AwaySetupDemo.as`

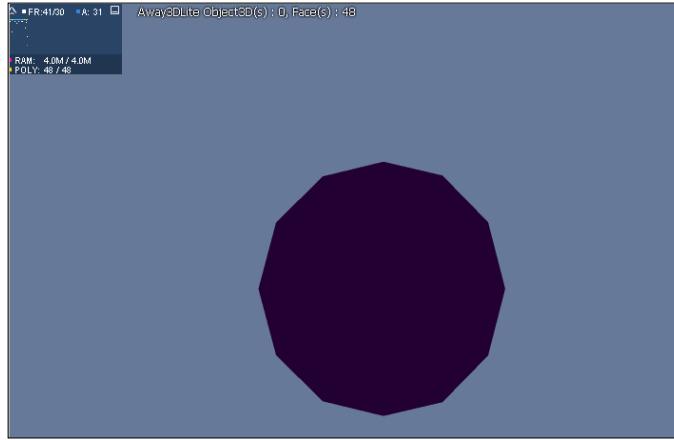
```
package
{
    public class AwaySetupDemo extends FastTemplate
    {
        private var _hammerModel:Mesh;
        public function AwaySetupDemo()
        {

        }
        override protected function onInit() : void{

            super.onInit();
            camera.z=-500;
            initGeometry();
        }
        override protected function onPostRender() : void{
            if(_sp){
                _sp.rotate(5,Vector3D.Y_AXIS);
            }
        }
        private function initGeometry():void{
    }
```

```
        _sp=new Sphere(new ColorMaterial(0x220033),80);
        scene.addChild(_sp);
    }
}
```

Here is our "Hello world" Away3D Lite program:



How it works...

Away3D Lite gives us two templates; `BasicTemplate` and `FastTemplate`. The only difference is that the last, as suggests its name, should perform faster. In reality, the frame rate difference between the two is hardly noticeable.

However, if you use Sprite3D in your scene, your should work with BasicTemplate or if you are setting up your scene manually, use Clipping() class instead of RectangleClipping(). Otherwise, you will find that the textures are not being drawn on Sprite3D instances.

Using one of the templates, you should override the protected `onInit()` method where you can initiate your code. The method is called inside the templates right after all the required classes to run Away are set up. One important thing you should notice here is that clipping, renderer, as well as `mouseEnabled3D` are defined inside the templates after the `onInit()` call. That means you can't modify them during the initiation stage. In such a case, it might be better to set up your scene manually or modify the template itself. For each frame execution, the templates give us two functions, namely, `onPreRender()` and `onPostRender()`. The first method gets called inside template's `onEnterFrame()` function before the call for `_view.render()`. `onPostRender()` method is called after `_view.render` inside `onEnterFrame()` function. Just override them and put code which needs to be called on every frame there.

Importing external models in Away3D Lite

Away3D Lite enables you to import external models just like Away3D either by loading in runtime or via direct embedding during compilation and then parsing them. As you will see shortly, the process is almost identical to how you would do it inside Away3D.

Getting ready

Set up a new Away3D Lite scene extending `FastTemplate`. Give it the name `AwayLiteAssetLoadDemo`.

In this demo, we are going to use a barrel model which is in `AWD` (Away3D data native format). See *Chapter 8, Prefab3D* (recipe 1) to learn how to export `AWD`. Navigate to Chapter 12's `assets` folder and copy into your project assets the `Barrel.awd` and `barrel96.png` files.

How to do it...

The following program loads a model of a barrel using `Loader3D` and `AWData` parser:

`AwayLiteAssetLoadDemo.as`

```
package
{
    public class AwayLiteAssetLoadDemo extends FastTemplate
    {
        private var _barrelModel:Mesh;

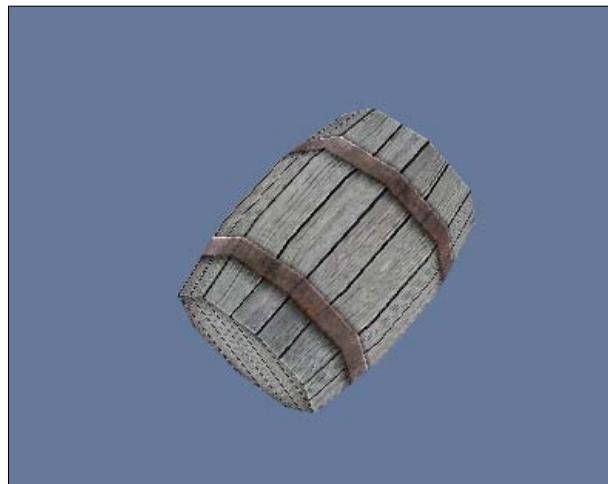
        public function AwayLiteAssetLoadDemo ()
        {
        }

        override protected function OnInit() : void{
            super.onInit();
            camera.z=-500;
            loadBarrel ();
        }

        private function loadBarrel():void{
            var loader:Loader3D=new Loader3D();
            loader.addEventListener(Loader3DEvent.LOAD_SUCCESS,onLoadComplete,f
            alse,0,true);
            _loader.loadGeometry("../assets/Barrel.awd",new AWData());
        }

        private function onLoadComplete(e:Loader3DEvent):void{
            _barrelModel=e.loader.handle as Mesh ;
        }
    }
}
```

```
barrelModel.material=new BitmapFileMaterial("../assets/images/
barrel96.png");
scene.addChild(_barrelModel);
_barrelModel.rotationX=90;
}
override protected function onPostRender() : void{
if(_barrelModel){
_barrelModel.roll(3);
_barrelModel.yaw(3);
}
}
}
}
```



Here we loaded a model of a barrel previously converted to the AWD format in Prefab.

How it works...

We initialize the loading process inside the `loadBarrel()` method which we call from the `onInit()` function. In `loadBarrel`, we first create an instance of the `Loader3D` class:

```
var loader:Loader3D=new Loader3D();
```

As the loading process is asynchronous, we have to assign a listener to `Loader3D` object in order to get the notification that the model has completely loaded:

```
loader.addEventListener(Loader3DEvent.LOAD_SUCCESS,onLoadComplete,fal
se,0,true);
```

The last step here is to call the `loadGeometry()` method on `_loader` into which we pass the URL of the model as well as the instance of the relevant parser, which is, in our case, `AWData()`:

```
_loader.loadGeometry("../assets/Barrel.awd", new AWData());
```

When the model has finished loading `_loader` fires a `LOAD_SUCCESS` event which is caught by the `onLoadComplete` event handler. Inside the method, we assign the loaded content to `_barrelModel`, which has a `Mesh` type:

```
_barrelModel=e.loader.handle as Mesh;
```

Then we add a material using `BitmapFileMaterial`, which handles loading of the textures for us:

```
barrelModel.material=new BitmapFileMaterial("../assets/images/barrel96.png");
```

Finally, we add the model to the scene:

```
scene.addChild(_barrelModel);
```

And that is it—piece of cake.

There's more...

As you already know, in addition to the dynamic asset loading method, you also have the ability to embed your models directly into the application. Let's see how it is done in Away3D Lite.

Parsing embedded models of embedded models in Away3D Lite is similar to Away3D. Let's see how to do it.

Inside the program we just created, comment call to the `loadBarrel()` function inside the `onInit()` block. Now first we need to embed our barrel model inside the class. Add the following right after the class declaration:

```
[Embed(source="assets/Barrel.awd", mimeType="application/octet-stream")]
private var Barrel:Class;
```

Let's create a function with the name `parseBarrel()`. Inside it, we instantiate AWData parser:

```
private function parseBarrel():void{
    var awdParser:AWData=new AWData();
```

Then we call its `parseGeometry()` method to which we pass an instance of the embedded model class:

```
_barrelModel=awdParser.parseGeometry(new Barrel()) as Mesh;
```

Now what is left to do is to assign it a material and add it to the scene:

```
_barrelModel.material=new BitmapFileMaterial("../assets/images/  
barrel96.png");  
scene.addChild(_barrelModel);  
_barrelModel.rotationX=90;
```



In a real world scenario, you should try using embedded or preloaded texture. `BitmapFileMaterial` loads the texture asynchronously and it may occur that after your model has already been parsed and added to the scene, its texture would still be in the middle of its loading process.

Manipulating geometry

The manipulation of objects in 3D space generally consists of rotation and translation (displacement) operations along three axes. In 3D graphics, these are usually done with matrices and vectors. Before Adobe's introduction of 3D API in Flash Player 10, open source library creators had to develop 3D Math APIs on their own to allow us to move and rotate objects in 3D space. The major greatness of Away3D Lite lies in the fact that it utilizes Flash Player 10 math API completely and this speeds up the engine's performance considerably as Matrix3D calculations are processed by the Flash Player natively. Away3D Lite simplifies for you many of the math operations by presenting a set of common rotation and translation functions via its API. For more advanced calculations, you can work with Flash Player 10 Math API directly. In this recipe, you will learn how to move and rotate your 3D objects using Away3D Lite methods as well as Flash Player 10 Math API.

Getting ready

Set up a new Away3D Lite scene extending `FastTemplate`. Give it the name `ObjectsManipDemo` and you are ready to go.

How to do it...

In the following example, we set up a cube which we wish to be able to move up, down, left, and right using keyboard input. Also, we are going to move it to the position of mouse click, which is represented by a vector in 3D space while rotating it towards mouse.



Before running this example, make sure you change the variable `view.mouseEnabled3D` to true inside of `FastTemplate`. Otherwise, the application will not get Mouse3D input.

ObjectsManipDemo.as

```
package
{
    public class ObjectsManipDemo extends FastTemplate
    {
        private var _block:Cube6;
        private var _bitMat:BitmapMaterial;
        private var _bitMat1:BitmapMaterial;
        private var _bg:Plane;
        private var _mouseVector:Vector3D;
        private const EASE_FACTOR:Number=0.2;
        private const RADSTODEGs:Number=180/Math.PI;
        public function ObjectsManipDemo()
        {
        }
        override protected function onInit() : void{
            super.onInit();
            camera.z=-600;
            initMaterials();
            initGeometry();
            setListeners();
        }
        private function initMaterials():void{
            var bdata:BitmapData=new BitmapData(256,256);
            bdata.noise(3456,0,120,2);
            var bdata1:BitmapData=new BitmapData(128,128);
            bdata1.perlinNoise(14,20,8,23556,true,true);
            _bitMat=new BitmapMaterial(bdata);
            _bitMat1=new BitmapMaterial(bdata1);
        }
        private function initGeometry():void{
            _bg=new Plane(_bitMat,1000,1000,2,2,false);
            _bg.transform.matrix3D.position=new Vector3D(0,0,200);
            scene.addChild(_bg);
            _block=new Cube6(_bitMat1,100,20,20);
            _block.transform.matrix3D.position=new Vector3D(4,4,0);
            scene.addChild(_block);
        }
    }
}
```

```

        }
        private function setListeners():void{
        _bg.addEventListener(MouseEvent3D.MOUSE_DOWN,onMouse3DDown);
        _bg.addEventListener(MouseEvent3D.MOUSE_UP,onMouse3DUp);
        stage.addEventListener(KeyboardEvent.KEY_DOWN,onKeyDown);

    }
    private function onKeyDown(e:KeyboardEvent):void{
        _mouseVector=null;
        switch(e.keyCode){
            case Keyboard.UP:
                _block.moveUp(5);
                break;
            case Keyboard.DOWN:
                _block.moveDown(5);
                break;
            case Keyboard.LEFT:
                _block.moveLeft(5);
                break;
            case Keyboard.RIGHT:
                _block.moveRight(5);
                break;
            default:
        }
    }
    private function onMouse3DDown(e:MouseEvent3D):void{
        _mouseVector=new Vector3D(e.scenePosition.x,e.scenePosition.y,
e.scenePosition.z);
    }
    private function onMouse3DUp(e:MouseEvent3D):void{
        _block.stopDrag();
    }
    override protected function onPostRender() : void{
        if(_mouseVector&&_block){

            var xVel:Number=(_mouseVector.x-_block.x);
            var yVel:Number=(_mouseVector.y-_block.y);

            //first approach//
            //_block.x+=xVel*EASE_FACTOR;
            //_block.y+=yVel*EASE_FACTOR;
            //second approach//
            _block.transform.matrix3D.appendTranslation(xVel*EASE_
FACTOR,yVel*EASE_FACTOR,_block.z);
        }
    }
}

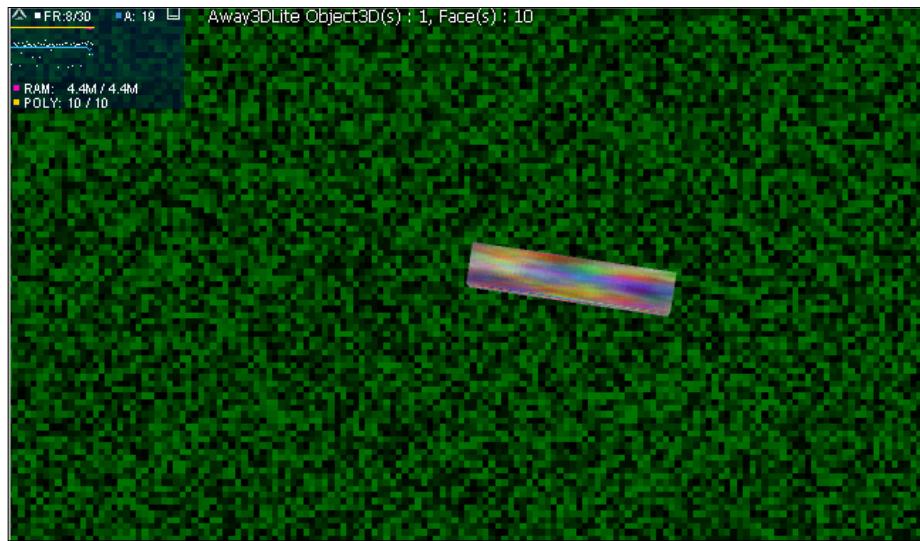
```

```

        //-----rotation-----//
        _block.rotationZ=Math.atan2( yVel, xVel)*RADStoDEGs;
    }
}
}
}
}

```

At this screenshot, we have the cube which moves and rotates to the location derived from the mouse click:



How it works...

Let's skip the explanation on scene elements initiation this time, as the process is straightforward. Instead, we will focus on the transformation operations we perform in the application. First we translate (move) the cube using the keyboard's input arrows. As you can see inside the `onKeyDown()` function, we call to `moveLeft`, `moveRight`, `moveUp`, `moveDown` commands to move the cube left, right, up, and down accordingly in its local space. Away3D simplified those operation for us which can be made otherwise directly via ActionScript-generic API. If you work with Flash Builder, just double-click one of these methods and you will be taken into the `Object3D` class where you can see the code behind these methods. So for `moveLeft()` or any other of this group, you could write it directly as it is coded inside the `translate` function of `Object3D`:

```

public function translate(axis:Vector3D, distance:Number):void
{
    axis.normalize();
    var _matrix3D:Matrix3D = transform.matrix3D;

```

```
    axis.scaleBy(distance);
    _matrix3D.position = _matrix3D.transformVector(axis);
}
```

Notice that axis and distance arguments are defined by you. When you pick in which direction to move, the object axis gets one of the following:

```
new Vector3D(1, 0, 0) -for moveLeft(),moveRight()
new Vector3D(0, 1, 0) -for moveUp(),moveDown()
new Vector3D(0, 0, 1) -for moveForward(),moveBackward()
```

Then the axis vector is scaled by the number you specified as an argument for one of these methods. The reason we scale the vector is to resize its magnitude, which is then used to set a new position for your object with the following line:

```
_matrix3D.position = _matrix3D.transformVector(axis);
```

The usage of the `transformVector()` method of `Matrix3D` assures that the translation is performed in an object's space.

In addition to keyboard-based movement, our demo features a cube's translation and rotation based on the mouse's position. Let us see how it works.

In order to move the object located in 3D space to a mouse position, we need to get a mouse position in 3D space by converting its 2D screen position into 3D. Fortunately, `Away3D Lite MouseEvent3D` does the job for you.

When the user clicks the mouse, the `onMouse3DDown()` event handler is called and inside it we acquire mouse 3D position inside the scene with this line:

```
_mouseVector=new Vector3D(e.scenePosition.x,e.scenePosition.y,
e.scenePosition.z);
```

Now, inside the `onPostRender()` function, we calculate the distance between x and y components of position vectors of mouse click and the current cube's position:

```
var xVel:Number=(_mouseVector.x-_block.x);
var yVel:Number=(_mouseVector.y-_block.y);
```

These two values will serve us to increment velocity for the x and y direction of the `_block` object. Now, to perform the translation, we can try a relatively orthodox approach by increasing directly the x and y properties of `_block` (as shown inside the commented block). Or we can do it just with one line of code using FP 10 API like this:

```
_block.transform.matrix3D.appendTranslation(xVel*EASE_
FACTOR,yVel*EASE_FACTOR,_block.z);
```

To rotate the object towards the mouse position, we first find the angle between them using the well known `Math.atan2()` method:

```
_block.rotationZ=Math.atan2( yVel, xVel)*RADStoDEGs;
```

In this specific case, using `_block.transform.matrix3D.appendRotation()` would not work. If you try to implement this approach, you will see the object rotating constantly. That is because `appendRotation()` increments the current rotation by the specified value, whereas `rotationZ` assigns it.

There's more...

You are strongly advised to carefully read the ActionScript3.0 online documentation where you can learn about the rest of the 3D math classes and their methods. Also, looking through sites such as www.flashandmath.com will help you to get a practical understanding of their usage.

Making 2D shapes appear three dimensional by using Sprite3D

3D sprites or billboards are 2D objects put in 3D space, which always face the camera and give an illusion of 3D. You would want to use them when a significant number of objects are needed to be placed in the scene while keeping the frame rate high. Away3D Lite contains the `Sprite3D` class which comes to the rescue in such scenarios. Let's see how it works.

Getting ready

Set up a new Away3D Lite scene extending `BasicTemplate`. Give it the name `Sprite3DDemo` and you are ready to begin.

How to do it...

In the following demo, we create 1,000 particles which consist of `Sprite3D` instances, while the frame rate (at least on my machine) is 30 out of 30:

`Sprite3DDemo.as`

```
package
{
    public class Sprite3DDemo extends BasicTemplate
    {
        [Embed(source="../assets/images/spriteTexture.png")]
        private var Tex:Class;
    }
}
```

```
private const NUM_TO_SPAWN:int=1000;
private var _bitMat:BitmapMaterial;
private var _camTarget:Vector3D=new Vector3D();
private var _pan:Number=0;
private var _tilt:Number=0;
private var _oldMouseX:Number=0;
private var _oldMouseY:Number=0;
private var DEGStoRADs:Number = Math.PI / 180;
private var _oldPan:Number=0;
private var _oldTilt:Number=0;
public function Sprite3DDemo()
{
}

override protected function onInit() : void{
    initMaterials();
    deploySprites();
    view.mouseEnabled3D = false;
    view.mouseEnabled=true;
    camera.z=0;
}
private function initMaterials():void{
    _bitMat=new BitmapMaterial(Cast.bitmap(new Tex()));
}
private function deploySprites():void{

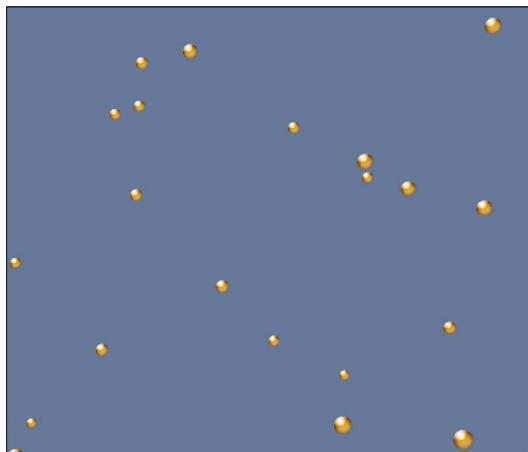
    for(var i:int=0;i<NUM_TO_SPAWN;++i){
        var x_pos:Number=Math.floor(Math.random()*2000-1000);
        var y_pos:Number=Math.floor(Math.random()*2000-1000);
        var z_pos:Number=Math.floor(Math.random()*2000-1000);
        var spr:Sprite3D=new Sprite3D(_bitMat);
        spr.alignmentType=AlignmentType.VIEWPOINT;
        spr.width=16;
        spr.height=16;
        spr.x=x_pos;
        spr.y=y_pos;
        spr.z=z_pos;
        scene.addSprite(spr);
    }
}
override protected function onPostRender() : void{
    _pan = (stage.mouseX+400)*0.5;
    _tilt =(stage.mouseY-300 )*0.5;
```

```

        if (_tilt > 120){ _tilt = 120;}
        if (_tilt < -120){ _tilt = -120;}
        var panRADs:Number=_pan*DEGStoRADs;
        var tiltRADs:Number=_tilt*DEGStoRADs;
        _camTarget.x = 100*Math.sin( panRADs) * Math.cos(tiltRADs) +
        camera.x;
        _camTarget.z = 100*Math.cos( panRADs) * Math.cos(tiltRADs) +
        camera.z;
        _camTarget.y = 100*Math.sin(tiltRADs) +camera.y;
        camera.lookAt(_camTarget);
    }
}
}

```

As is seen from the screenshot, we have a nice illusion of small 3D balls dispersed in 3D space. Not every end user would guess that these are just flat objects always facing the camera:



How it works...

One important thing you should know is that if you use `Sprite3D` inside a class that extends one of the templates, it has got to be `BasicTemplate` as it uses `RectangleClipping` and `BasicRenderer`. Using `FastRenderer` or `Clipping` will cause a distortion or shapes with no materials drawn on top of their surface.

`Sprite3D` setup is very easy and straightforward. In our example, we created 1,000 sprites inside the `deploySprites()` method, assigning each of them a random position:

```

var x_pos:Number=Math.floor(Math.random()*2000-1000);
var y_pos:Number=Math.floor(Math.random()*2000-1000);
var z_pos:Number=Math.floor(Math.random()*2000-1000);

```

The sprite initiation is very short. You have to pass only a material into its constructor:

```
var spr:Sprite3D=new Sprite3D(_bitMat);
```

An additional setting you should be familiar with is how the plane of the sprite is aligned towards the viewer. In our case, we set it to face the viewer point, which is the camera position. The second option is `AlignmentType.VIEWPLANE`, which aligns the plane with the view plane:

```
spr.alignmentType=AlignmentType.VIEWPOINT;
```

Next, we set up a position and dimensions for our sprites. Notice that `Sprite3D` has no `Matrix3D` property:

```
spr.width=16;  
spr.height=16;  
spr.x=x_pos;  
spr.y=y_pos;  
spr.z=z_pos;
```

Last step is to add it to the scene. For sprites, you should use the `scene.addSprite()` method:

```
scene.addSprite(spr);
```

Managing Z-Sorting by automatic sorting and using layers

Away3D Lite, being a compact engine, has a limited set of tools for solving Z-Sorting issues in comparison to Away3D. Basically, there are two ways to sort an object's depth. First is by automatically sorting which Away3D Lite performs by default. The second approach is manual and which implies usage of layers. In this chapter, you are going to learn how to use both of them.

Getting ready

Create a new class which extends `FastTemplate` and name it `SortingDemo`.

In this example, we make use of GUI controls which are part of a Flash components library developed by Keith Peters and called *MinimalComps*. Go and get it for free here: <http://www.minimalcomps.com/>. Put the `.swc` into your `SWC` folder or connect it directly to your project.

How to do it...

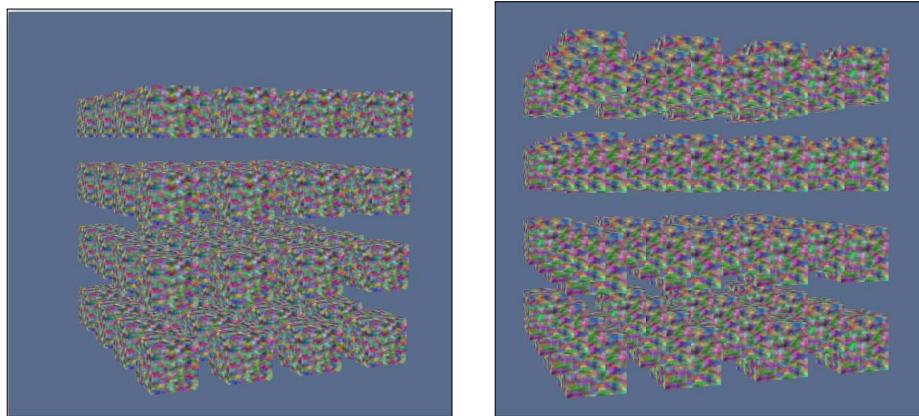
SortingDemo.as

```
package
{
    public class SortingDemo extends FastTemplate
    {
        private var _bitMat:BitmapMaterial;
        private var _container:ObjectContainer3D;
        private var _hoverCam:HoverCamera3D;
        private var _oldMouseX:Number=0;
        private var _oldMouseY:Number=0;
        public function SortingDemo()
        {
            super();
        }
        override protected function OnInit() : void{
            initMaterials();
            initGeometry();
            setHoverCamera();
            initGUI();
        }
        override protected function onPostRender() : void{
            if(_hoverCam) {
                _hoverCam.panAngle = (stage.mouseX - _oldMouseX) ;
                _hoverCam.tiltAngle = (stage.mouseY - _oldMouseY);
                _hoverCam.hover();
            }
        }
        private function initGUI():void{
            var cb:CheckBox=new CheckBox(this,10,100,"Enable
sorting",onCBCheck);
            cb.selected=true;
        }
        private function onCBCheck(e:Event):void{
            if(e.target.selected){
                renderer.sortObjects=true;
            }else{
                renderer.sortObjects=false;
            }
        }
    }
}
```

```
private function initMaterials():void{
    var bdata:BitmapData=new BitmapData(256,256);
    bdata.perlinNoise(14,14,7,23235,true,true);
    _bitMat=new BitmapMaterial(bdata);

}
private function initGeometry() : void{
    for(var i:int=0;i<4;++i){
        for(var b:int=0;b<4;++b){
            for(var c:int=0;c<4;++c){
                var cube:Cube6=new Cube6(_bitMat,60,60,60);
                cube.x=i*100-200;
                cube.y=b*100-80;
                cube.z=c*100-700;
                scene.addChild(cube);
            }
        }
    }
}
private function setHoverCamera():void{
    _hoverCam=new HoverCamera3D();
    view.camera=_hoverCam;
    var dummy:ObjectContainer3D=new ObjectContainer3D();
    scene.addChild(dummy);
    dummy.transform.matrix3D.position=new Vector3D(30,40,-600);
    _hoverCam.target=dummy;
    _hoverCam.distance = 1000;
    _hoverCam.maxTiltAngle = 80;
    _hoverCam.minTiltAngle = 0;
    _hoverCam.wrapPanAngle=true;
    _hoverCam.steps=16;
    _hoverCam.yfactor=1;
}
}
```

In the following left screenshot, the sorting is set to true, while in the right, it is disabled:



How it works...

If you carefully look at the two previous screenshots, you can see that only the left one depicts correct Z-Sorting of geometry. If you unselect the checkbox in the demo application by doing this, you set the `renderer.sortObjects` property to false which commands the engine to stop sorting. So basically, you have a control on whether to sort your objects or not. Usually, you would not want to disable Z-Sorting as in terms of performance you will hardly gain any frame rate improvement.

There's more...

In this section, we are going to learn how to use layers to help you manage Z-Sorting your geometry manually.

Sorting using layers

Sometimes you may wish to control depth position of a single or group of objects manually. Away3D Lite allows us to do this task by using layers. A layer in Away3D Lite is simply a generic Sprite which serves as a wrapper for the objects to which it is assigned. Let's see how we can use them.

Create a new class which extends `BasicTemplate`. Give it the name `LayersDemo`. Copy into it the following code:

`LayersDemo.as`

```
package
{
    public class LayersDemo extends BasicTemplate
```

```

{
    private var _layer1:Sprite=new Sprite();
    private var _layer2:Sprite=new Sprite();
    private var _layer3:Sprite=new Sprite();
    private var _bdata:BitmapData;
    public function LayersDemo()
    {
        super();
    }
    override protected function onInit() : void{
        camera.z=-500;
        initMaterials();
        setLayers(_layer1,0xFFFF00,-50);
        setLayers(_layer2,0x871299,0);
        setLayers(_layer3,0x12FF33,50);
        view.addChild(_layer3);
        view.addChild(_layer2);
        view.addChild(_layer1);
    }
    override protected function onPreRender():void
    {
        scene.rotationY++;
    }

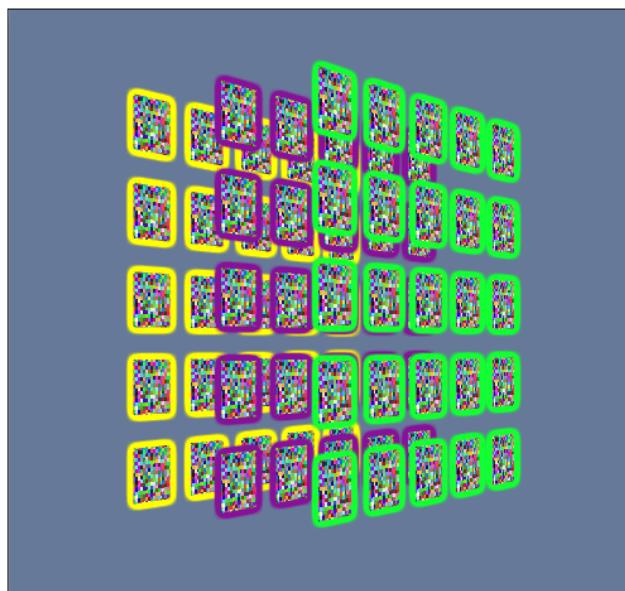
    private function initMaterials():void{
        _bdata=new BitmapData(16,16);
        for(var i:int=0;i<_bdata.width;++i){
            for(var c:int=0;c<_bdata.height;++c){
                _bdata.setPixel(i,c,Math.floor(Math.random()*0xFFFFFF));
            }
        }
    }
    private function initGeometry(depth:Number):Vector.<Plane>{
        var spArr:Vector.<Plane>=new Vector.<Plane>();
        for(var i:int=0;i<5;++i){
            for(var c:int=0;c<5;++c){
                var sp:Plane=new Plane(new BitmapMaterial(_bdata),25,25,1,1,false);
                sp.bothsides=true;
                sp.transform.matrix3D.position=new Vector3D(i*40-100,c*40-100,depth);
                scene.addChild(sp);
                spArr.push(sp);
            }
        }
    }
}

```

```
        }
        return spArr;
    }
    private function onClick(e:MouseEvent):void
    {
        var layer:Sprite = event.target as Sprite;
        view.addChildAt(layer,view.numChildren-1);
    }
    private function setLayers(layer:Sprite,glowColor:uint,depth:Number):void{
        var arr1:Vector.<Plane>=initGeometry(depth);
        var gFilter:GlowFilter=new GlowFilter(glowColor, 1, 7, 7, 16,
2);
        for(var i:int=0;i<arr1.length;++i){
            arr1[i].layer=layer;
        }
        arr1[0].layer.filters=[gFilter];
        arr1[0].layer.addEventListener(MouseEvent.CLICK, onClick);

    }
}
```

In the screenshot, each layer is represented by the unique colored set of quadratic planes. Click the planes in each of the sets to change their depth-sorting order:



When you run the application, you can see three layers of small planes positioned at some depth from each other and marked by different outline colors. You can also see that only from one side of rotation their z order is correct. When they get to 180 degrees, still the most distant layer, which was in the beginning in the front, it keeps its initial z position. The reason for it is that sprite objects, by default, get Z-Sorting by their order in display list and not by their z position value. In most cases, this fact is undesired, but in certain cases, it gives you the power to obtain manual control of depth sorting. In the previous example, we assigned to each of three layers `MouseEvent.CLICK` so that you can change the z order of the layer by moving it higher in the display list hierarchy of the parent which is, in our case, view. When the user clicks one of the layers, we move that layer to the top-most index with this line of code inside the `onClick` handler:

```
view.addChildAt(layer,view.numChildren-1);
```

And this is all the magic. Layers can be helpful in tricky scenarios to enforce custom z ordering for single objects as well as for groups.

Creating virtual trackball

Virtual trackball or Arcball is a widely used technique in 3D graphics which allows performing realistic interactive 3D rotation using so called virtual or imaginary sphere. There is some complex math involved behind the process which we are not going to discuss here in depth. Instead, with the following example, you will learn how to implement the concept practically within Away3D Lite. Special credits go to Ralph Hauwert who wrote the initial application with pure ActionScript, which I then ported into Away3D Lite.

Getting ready

Create a new class which extends `FastTemplate` and give it the name `TrackballDemo`.

How to do it...

In the following demo, we create an interactive sphere which is rotated in an extremely natural way using the mouse input:

`TrackballDemo.as`

```
package
{
    public class TrackballDemo extends FastTemplate
    {
        private var _ball:Sphere;
        private var _startVector:Vector3D;
        private var _startQuat:Vector3D;
```

```
private var _dragVector:Vector3D;
private var _dragQuat:Vector3D;
private var _newQuat:Vector3D;
private var _canDrag:Boolean=false;
private var _bitmapMat:BitmapMaterial;
private var _transformComponents:Vector.<Vector3D>;

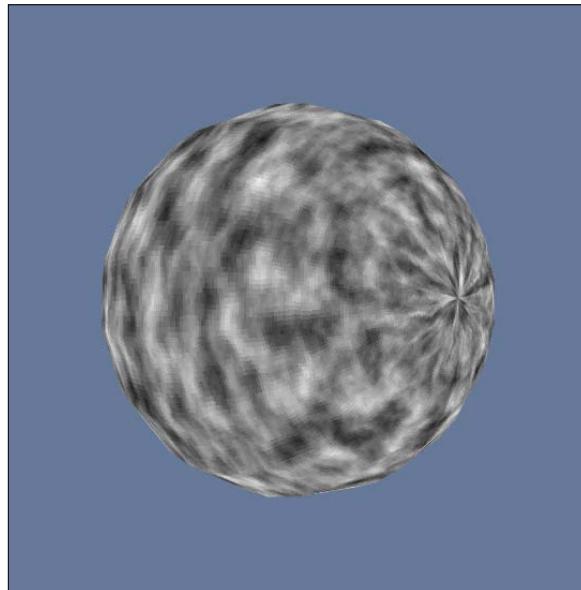
public function TrackballDemo()
{
    super();
}
override protected function onInit() : void{
    initMath();
    initMaterials();
    initGeometry();
    initListeners();
}
override protected function onPostRender() : void{
    if(_canDrag){
        _dragVector = coordinate2DTOSphere(scene.mouseX,scene.mouseY,_dragVector);
        dragTo(_dragVector);
    }
}
private function initMath():void{
    _startVector=new Vector3D();
    _dragVector=new Vector3D();
    _startQuat=new Vector3D();
    _dragQuat=new Vector3D();
    _newQuat=new Vector3D();
}
private function initMaterials():void{
    var bdata:BitmapData=new BitmapData(256,256);
    bdata.perlinNoise(12,12,12,17,false,true,7,true);
    _bitmapMat=new BitmapMaterial(bdata);
}
private function initGeometry():void{
    _ball=new Sphere(_bitmapMat,200,16,16,true);
    scene.addChild(_ball);
    _ball.z=0;
}
private function initListeners():void{
stage.addEventListener(MouseEvent.MOUSE_DOWN, onMouse3dDown);
}
```

```
private function onMouse3dDown(e:MouseEvent):void{
    stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
    _startVector=coordinate2DToSphere(scene.mouseX, scene.mouseY, _startVector);

    startDragThis(_startVector);
    _canDrag=true;
}
private function onMouseUp(e:MouseEvent):void{
    e.stopPropagation();
    stage.removeEventListener(MouseEvent.MOUSE_UP, onMouseUp);
    _canDrag=false;
}
private function coordinate2DToSphere(x:Number, y:Number,
targetVector:Vector3D):Vector3D
{
    targetVector.x = (_ball.x-x)/_ball.radius;
    targetVector.y = (_ball.y-y)/_ball.radius;
    targetVector.z = 0;
    if(targetVector.lengthSquared > 1){
        targetVector.normalize();
    }else{
        targetVector.z = Math.sqrt(1-targetVector.lengthSquared);
    }
    return targetVector;
}
private function startDragThis(vector3D:Vector3D):void
{
    _transformComponents = _ball.transform.matrix3D.
decompose(Orientation3D.QUATERNION);
    _startQuat= _transformComponents[1]as Vector3D;
    _dragQuat=new Vector3D(0,0,0,0);
}
private function dragTo(vector:Vector3D):void
{
    var v:Vector3D = _startVector.crossProduct(_dragVector);
    _dragQuat.x = v.x;
    _dragQuat.y = v.y;
    _dragQuat.z = v.z;
    _dragQuat.w = _startVector.dotProduct(_dragVector);
    multiplyQuats(_dragQuat, _startQuat, _newQuat);
    _transformComponents[1] = _newQuat;
    _ball.transform.matrix3D.recompose(_transformComponents,
Orientation3D.QUATERNION);
```

```
        }
        private function multiplyQuats(q1:Vector3D, q2:Vector3D,
out:Vector3D):void
{
    out.x = q1.w*q2.x+q1.x*q2.w+q1.y*q2.z-q1.z*q2.y;
    out.y = q1.w*q2.y+q1.y*q2.w+q1.z*q2.x-q1.x*q2.z;
    out.z = q1.w*q2.z+q1.z*q2.w+q1.x*q2.y-q1.y*q2.x;
    out.w = q1.w*q2.w-q1.x*q2.x-q1.y*q2.y-q1.z*q2.z;
}
}
```

The resulting ball which you should be able to rotate around arbitrary axes using your mouse is shown in the following screenshot:



How it works...

First calm down and take a deep breath. Some of the operations found in the preceding code are not easy to understand as they require solid understanding of 3D math. Let's examine the code, and after that I am sure you will feel much more comfortable with it.

Let's begin with the member variables we have set. Inside `initMath()` we instantiate five vectors where three of them represent Quaternions. `_startVector` holds the 3D position of mouse click on the sphere, `_dragVector` receives the updated position of the mouse on the sphere. Both these vectors serve us to fill Quaternions, as you will see shortly.

Now the intermediate calculation of rotations we perform using Quaternions and we set up three of them. We calculate the final Quaternion named `_newQuat` executing a multiplication of initial Quaternion `_startQuat`, which is acquired when we press the mouse and the current Quaternion `_dragQuat` that is calculated while we drag the mouse to rotate the ball. Now when you know what these variables mean, let us see how the actual calculations are performed.

When the user clicks the mouse, `onMouse3dDown()` handler is triggered. Inside it we convert 2D mouse coordinates to 3D on top of the ball surface with this line of code:

```
_startVector=coordinate2DToSphere(scene.mouseX,scene.mouseY,_  
startVector);
```

Let's take a look how the `coordinate2DToSphere()` function works. First we calculate x, y, and z on the surface using the following expressions:

```
targetVector.x = (_ball.x-x)/_ball.radius;  
targetVector.y = (_ball.y-y)/_ball.radius;  
targetVector.z = 0;
```

Next we should check if the point is not outside the ball. We find it checking if the square root of the `targetVector` components is bigger than one. If so, we set the coordinates to the closest point from the offset on the ball's surface by normalizing the vector:

```
if(targetVector.lengthSquared > 1){  
targetVector.normalize();  
}
```

If the point is on the surface, we calculate z by using the following formula:

```
targetVector.z = Math.sqrt(1-targetVector.lengthSquared);
```

Ok, now that we found that the 3D position of mouse click is on top of the sphere, we can proceed with calculating the rotation. Inside `onMouse3dDown()`, we call the `startDragThis()` method to which we pass our initial 3D position of the mouse as a parameter. Inside `startDragThis()`, we first extract the Quaternion of the current rotation with this line:

```
_transformComponents = _ball.  
transform.matrix3D.decompose(Orientation3D.QUATERNION);
```

`Matrix3D.decompose()` returns an array of three vectors, which represent translation, rotation, and scale values of the `matrix3D`. As we are interested in the rotational part, we take the second vector:

```
_startQuat= _transformComponents[1]as Vector3D;
```

Now we have got a Quaternion for the initial rotation and we reset `_dragQuaternion` from the last drag in order to start a new one:

```
_dragQuat=new Vector3D(0,0,0,0);
```

Finally, we call the `dragTo()` method on each frame inside the `onPostRender()` method where we perform the actual drag of the ball. Here we continuously update `_dragVector` by reading and converting the current mouse 2D position to 3D. Then we pass this vector as an argument for `dragTo.dragTo()` method is the heart of the process. Inside it, we first calculate a new y axis which is perpendicular to the dragging direction. We calculate it with cross product:

```
var v:Vector3D = _startVector.crossProduct(_dragVector);
```

Now when we have the axis around which we should rotate, we update the dragging Quaternion:

```
_dragQuat.x = v.x;  
_dragQuat.y = v.y;  
_dragQuat.z = v.z;
```

We also have to calculate so called forth (w) or homogenous coordinates of the Quaternion, which gives the amount of rotation around the defined axis:

```
dragQuat.w = _startVector.dotProduct(_dragVector);
```

Having `_dragQuat`, our next task is to calculate a new Quaternion which is based on the initial and current ones. Here we come to Quaternion multiplication. ActionScript doesn't have a built-in function for such an operation, but you can find out how to do it on the web. In our example, we have the `multiplyQuats()` function which receives two Quaternions to multiply and outputs their product, which is, in our case, `_newQuat`:

```
multiplyQuats(_dragQuat, _startQuat, _newQuat);
```

Now we update the rotation portion of a decomposed matrix by new Quaternion values:

```
_transformComponents[1] = _newQuat;
```

The last step left is to convert the Quaternion to rotation matrix because Flash rotations are matrix-based. To do this, we use the `recompose()` method of the `Matrix3D` class, which allows us to rebuild the matrix rotation portion from the `_newQuat` data:

```
_ball.transform.matrix3D.recompose(_transformComponents,  
Orientation3D.QUATERNION);
```

The preceding line transforms `_ball`'s actual rotation by updating its matrix.

There's more...

You are recommended to get a deeper explanation on trackball math here:
<http://viewport3d.com/trackball.htm>.

Writing Away3D Lite applications with haXe

So what is haXe and why would you ever want to use it? Well, very quickly, haXe is a multiplatform language that allows you to compile code to different languages such as PHP, C++, JavaScript, and of course ActionScript. Using haXe gives you several advantages. First, the compile time is extremely fast, second the language structure is very robust especially in comparison to ActionScript, and the compiled binary (SWF) is better optimized than regular Flash compilations that means your code performs faster. Also, in many cases, you will find that haXe compiled SWF are lighter than similar binaries produced with Flash.

Although haXe language has got a built-in standard Flash library you can't use a regular Away3D API with it as it is written in ActionScript and therefore not compatible with haXe language specifications. Fortunately, the Away3D team at the time of writing deploys Away3D Lite branch for haXe, and we are going to learn how to get started with it here. Go to <http://haxe.org/> to learn more about the language structure and how to write Flash applications with it. In this recipe, we will port one of the recipes of this chapter to haXe language. Let's do it.

Getting ready

In this recipe, we will shut down Flash or FlashBuilder for a while as we are going to work in open source and arguably the best Flash development IDE called **FlashDevelop**. The reason for this is that the IDE allows us to write a haXe application as a built-in option, which allows you to enjoy such features as code hints and completion, debugging, and more. Alternatively, you can write your own haXe code in any text editor and compile it with command line.



As FlashDevelop is not available for Mac and Linux, alternate options can be found at <http://haxe.org/com/ide>.

1. Go to <http://www.flashdevelop.org/> and download the latest version of FlashDevelop. Install it after the download has finished.
2. Go to <http://haxe.org/download> and download the haXe installer through your operating system. Install haXe. Restart the machine after the installation.

3. Start-up FlashDevelop. In the top menu bar, go to **Project | New Project**. Inside the dialog window that has opened, select **haXe AS3 Project**. Give it a name and click **OK**.
4. Now we need to connect to the project haXe version of Away3D Lite. Inside FlashDevelop, right-click the root project folder in the project hierarchy panel, click the **Properties**.**Select Classpath** tab, and click the **Add Classpath** button. Navigate to **Away3d SVN**. Go to **trunk/haxe/Away3dLite/src/**, then click **OK**. Now Away3D Lite haXe is available to you inside the project.

How to do it...

Inside your haXe project, you can see that you have an automatically generated class `Main.hx`. This class is essential and serves as an entry point for your program. Any classes you want to instantiate should be initialized inside the `main()` method of `Main.hx`. If you programmed with languages such as JAVA, you will find this concept familiar. Our `Main.hx`. Copy the following code inside your `Main.hx`:

```
package ;
import away3dlite.materials.BitmapMaterial;
import flash.display.StageQuality;
import flash.Lib;
class Main
{
    static function main()
    {
        Lib.current.addChild(new AwayContainer());
        Lib.current.stage.frameRate = 30;
        Lib.current.stage.quality = StageQuality.MEDIUM;
        Lib.current.stage.stageWidth = 800;
        Lib.current.stage.stageHeight = 600;
    }
}
```

Now create another class inside your project and give it the name `AwayContainer`.

Paste into it the following code:

```
package ;

import away3dlite.events.MouseEvent3D;
import away3dlite.materials.BitmapMaterial;
import away3dlite.primitives.Cube6;
import away3dlite.primitives.Plane;
import away3dlite.templates.FastTemplate;
import flash.display.BitmapData;
```

```
import flash.events.KeyboardEvent;
import flash.geom.Vector3D;
import flash.ui.Keyboard;
class AwayContainer extends FastTemplate
{
    private var _block:Cube6;
    private var _bitMat:BitmapMaterial;
    private var _bitMat1:BitmapMaterial;
    private var _bg:Plane;
    private var _mouseVector:Vector3D;
    inline static var EASE_FACTOR:Float=0.2;
    inline static var RADSToDEGs:Float = 180 / Math.PI;

    public function new()
    {
        super();
    }
    private override function onInit():Void {
        super.onInit();
        camera.z=-600;
        initMaterials();
        initGeometry();
        setListeners();
    }
    private function initMaterials():Void {
        var bdata:BitmapData=new BitmapData(256,256);
        bdata.noise(3456,0,120,2);
        var bdata1:BitmapData=new BitmapData(128,128);
        bdata1.perlinNoise(14,20,8,23556,true,true);
        _bitMat=new BitmapMaterial(bdata);
        _bitMat1=new BitmapMaterial(bdata1);

    }
    private function initGeometry():Void{
        _bg=new Plane(_bitMat,1000,1000,2,2,false);
        _bg.transform.matrix3D.position=new Vector3D(0,0,200);
        scene.addChild(_bg);
        _block=new Cube6(_bitMat1,100,20,20);
        _block.transform.matrix3D.position=new Vector3D(4,4,0);
        scene.addChild(_block);
    }
    private function setListeners():Void{
```

```
_bg.addEventListener(MouseEvent3D.MOUSE_DOWN, onMouse3DDown);
_bg.addEventListener(MouseEvent3D.MOUSE_UP, onMouse3DUp);
stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);

}

private function onKeyDown(e:KeyboardEvent):Void{
    _mouseVector = null;
    switch(e.keyCode){
        case Keyboard.UP:
            _block.moveUp(5);

        case Keyboard.DOWN:
            _block.moveDown(5);
        case Keyboard.LEFT:
            _block.moveLeft(5);
        case Keyboard.RIGHT:
            _block.moveRight(5);
        default:
    }
}

private function onMouse3DDown(e:MouseEvent3D):Void{
    _mouseVector=new Vector3D(e.scenePosition.x,e.scenePosition.y,e.scenePosition.z);
}

private function onMouse3DUp(e:MouseEvent3D):Void{
    _block.stopDrag();
}

private override function onPostRender(): Void {
    if (!view.mouseEnabled3D) {
        view.mouseEnabled3D = true;
    }

    if(_mouseVector!=null&&_block!=null){
        var xVel:Float=(_mouseVector.x-_block.x);
        var yVel:Float=(_mouseVector.y-_block.y);
        _block.transform.matrix3D.appendTranslation(xVel*EASE_FACTOR,yVel*EASE_FACTOR,_block.z);
        _block.rotationZ=Math.atan2(yVel, xVel)*RADSToDEGs;
    }
}
}
```

How it works...

If you noticed, we ported the demo code of the *Manipulating geometry* recipe in this chapter to haXe. Let's see how this thing works. We begin with `Main.hx`. Inside the `main()` method, we first define some global properties for the Flash stage such as dimension, frame rate, and quality, which we access with the following code:

```
Lib.current.stage.frameRate = 30;
Lib.current.stage.quality = StageQuality.MEDIUM;
Lib.current.stage.stageWidth = 800;
Lib.current.stage.stageHeight = 600;
```

Then we add to it our class which contains all the Away3D Lite applications in it:

```
Lib.current.addChild(new AwayContainer());
```

We are not going through every line of `AwayContainer` because basically this is the same code as in the *Manipulating geometry* recipe. But I will outline a few major differences between ActionScript and haXe, which can be found in the code of our demo application.

ActionScript	HAXE
Decimals data type	Number
	Float
var xVel:Number=_mouseVector.x-_block.x;	var xVel:Float=_mouseVector.x-_block.x;
Integer constant	int
	Int
const	const
	inline
private const EASE_FACTOR:Number=0.2;	inline static var EASE_FACTOR:Float=0.2;

Additionally, haXe doesn't allow us to instantiate member (global) variables upon their declaration or assign default values. Nevertheless; in the case of constants, you must assign the values upon the declaration. haXe doesn't have protected scope. If you want a function to be visible for overriding or calling from outside, define it as public. When writing switch statements, each case block is written without a break; escape.

By the way, haXe compilation of the following example is almost 9K lighter than the same done with Flash. Try comparing it yourself.

There's more...

To learn more about haXe language, its different language modules, tutorials, and documentation, go to the project home page at <http://haxe.org>.

12

Optimizing Performance

In this chapter, we will cover:

- ▶ Defining depth of rendering
- ▶ Restricting the scene poly count
- ▶ Working with LOD objects
- ▶ Optimizing geometry by welding vertices
- ▶ Excluding objects from rendering
- ▶ Image size consideration

Introduction

Now consider the following case: You studied Away3D API from 360 degrees to the level that you remember each class by heart and know where and how to implement it. You also designed and imported all the necessary assets, then assembled it all into your final first commercial 3D application. You run your program expecting to see cool 3D stuff spinning around that you created with your own hands and yes, you see it, the application looks just as cool as you expected it to be, but in many cases, you will notice something else that you had not expected to show up. The thing that may not just bring down your spirits but also ruin months of hard work is called performance.

Away3D is a relatively powerful Flash platform-based 3D engine, when compared to its competitors on the market, but still we deal with Flash here, and Flash player capabilities are still extremely limited in relation to 3D rendering (no GPU support).

While console 3D engines such as Epic's Unreal Engine or some new technologies such as Unity3D are able to "eat" hundreds, thousands of polygons per frame, multiple dynamic lights, complex shaders, and physics all together without even a slightest frame rate drop, an Away3D scene may completely freeze at merely 4-6 thousands of triangles with no scene lights or complex shading. However, there is some good news which will encourage you though. Away3D can potentially eat more than 4-6 thousands of triangles and render quite impressive graphics using lights and shaders with decent frame rate, if you know how to optimize your program. Utilizing all possible performance tricks and techniques, you can considerably reduce CPU consumption and make your application run really fast. In this chapter, you will learn some core approaches to the performance optimization in Away3D as well as a bunch of useful tips which we suggest you strongly consider at the project planning stage.



It is impossible to outline here all the possible tricks and techniques that will cause your scene to run perfectly first because there are many of them and second various scenarios may require different approaches. Putting it simply—a lot of tweaking and experimenting on your own will bring the best results.

Defining depth of rendering

In certain scenarios, you may have a setup with multiple geometry dispersed across enormous environments. It may be, for example, an outdoor virtual world or a first person shooter. Some of those objects that lie too far in the direction of the camera, even if they are very small and distant or obscured by other objects, are still rendered by the engine. In this case, you may wish to exclude those distant objects from rendering until they reach a certain range from the camera.

Imagine a forest with a depth of 10,000 pixels. The density of the trees allows you to look into the depth of only 1,000 pixels and still the forest is endlessly deep (depends of course on how many trees you have placed). So moving through it with the maximum rendering depth set to 1,000 will reveal more and more distant trees as soon as they enter the radius of 1000 pixels from the camera, instead of drawing all of them continuously. Away3D makes it easy to set this kind of filtering. Let's see how it is done.

Getting ready

1. Set up a default Away3D scene extending the `AwayTemplate` class.
2. Go to the source code directory of *Chapter 11* and copy the `FPSController.as` class first person camera control utility that we created in *Chapter 2*.
3. In your project, create a new package, name it `utils`, and paste `FPSController.as` into it. Now we are able to set a first person controllable camera.

How to do it...

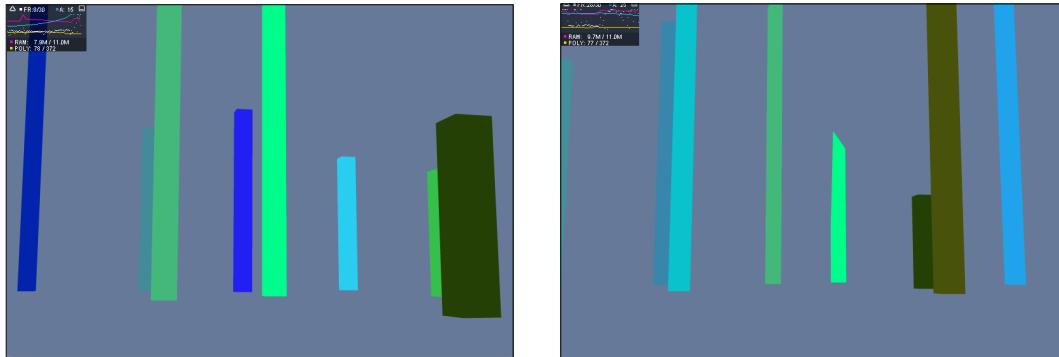
In this demo program we generate a pretty big number of cubic pillars and disperse them randomly on the scene in all directions. Restricting depth rendering, you will notice that only the pillars located inside the render range of the camera are visible.

MaxZRenderDemo.as

```
package
{
    public class MaxZRenderDemo extends AwayTemplate
    {
        private var _fpsWalker:FPSController;
        private var _angle:Number=0;
        private var _radius:int=70;
        private var _buildIter:int=1;
        private var _numBuilds:int=5;
        public function MaxZRenderDemo()
        {
            super();
        }
        override protected function initGeometry() : void{
            initFrustumClipping();
            seedBuildings();
            _fpsWalker=new FPSController(_cam,stage,20,5,40);
        }
        override protected function onEnterFrame(e:Event) : void{
            super.onEnterFrame(e);
            _fpsWalker.walk();
        }
        private function seedBuildings():void{
            for(var b:int=0;b<_numBuilds;++b){
                var h:Number=Math.floor(Math.random()*400+80);
                _angle=Math.PI*2/_numBuilds*b;
                var colMat:ColorMaterial=new ColorMaterial(Math.round(Math.
random()*0x565656));
                var build:Cube=new Cube({width:20,height:h,depth:20});
                build.ownCanvas=true;
                build.material=colMat;
                build.movePivot(0,-build.height/2,0);
                build.x=Math.cos(_angle)*_radius*_buildIter*2;
                build.z=Math.sin(_angle)*_radius*_buildIter*2;
                build.y=0;
                _view.scene.addChild(build);
            }
        }
    }
}
```

```
_buildIter++;
if(_buildIter<5){
    _numBuilds*=_buildIter/2;
    seedBuildings();
}
}
private function initFrustumClipping():void{
    var frustumClip:FrustumClipping=new
    FrustumClipping({maxZ:1000});
    _view.clipping=frustumClip;
}
}
```

In the left screenshot, you can see all the pillars intact, as they are located inside the depth range of the camera. While we move away from the pillars in the right screenshot, you notice the furthest pillars begin to disappear as they fall out of the defined depth:



How it works...

The setup is simple. Inside the `initGeometry()` method, we put some geometry and disperse it in different depths so that we would be able to see the removing of the objects when those fall out of the defined clipping range. Next, we initiate an instance of `FrustumClipping` inside `initFrustumClipping()`:

```
private function initFrustumClipping():void{
    var frustumClip:FrustumClipping=new
    FrustumClipping({maxZ:1000});
    _view.clipping=frustumClip;
}
```

Here we set the maximum visible distance along the Z axis to 1,000 pixels using `maxZ` property.

You are already acquainted with `Nearfield` and `FrustumClipping`. If you aren't, please refer to *Chapter 7* before you proceed with this recipe. `NearfieldClipping` and `FrustumClipping` allow you to define the maximum Z axis distance at which the objects are not culled. `maxZ` property is responsible for this. Behind the curtains, we move the invisible far plane of the Frustum closer or further from the camera so that every object that falls behind that plane is removed from the rendering stack.



Note that although `RectangularClipping`, which is set by default, has got the `maxZ` property which is not functional. For depth filtering, use the `NearfieldClipping` or `FrustumClipping` class.



Of course, you are not restricted to define the rendering range only on the Z axis. You also have `maxX` and `maxY`, which define geometry culling boundaries at the X and Y axis. Also you may want to set the minimum range, which means from which closest distance to the camera the objects become visible. So, for example, if you defined `minZ=500` and `maxZ=1000`, then only those objects that are located within the range (between 500 and 1,000 pixels) from the camera will be drawn by the renderer.

See also

Chapter 2's Creating an FPS (first person shooter) controller.

Restricting the scene poly count

Away3D allows you to control the maximum number of overall scene polygons. It has got a generic class called `MaxPolyFilter`, which keeps the total number of polygons in the scene to a defined limit. Don't be mistaken as this filter can in no way replace a low poly modeling approach, but it may be useful in scenarios where you do not want to remove unnecessary objects at once, as is the case when you use `ZDepthFilter`, but to cause them to disappear gradually.

Getting ready

Set up a default Away3D scene extending the `AwayTemplate` class.

Go to the source code directory of *Chapter 11* and copy the `FPSController.as` class (first person camera control utility that we created in *Chapter 2*).

In your project, create a new package, name it `utils`, and paste `FPSController.as` into it. Now we are able to set a first person controllable camera.

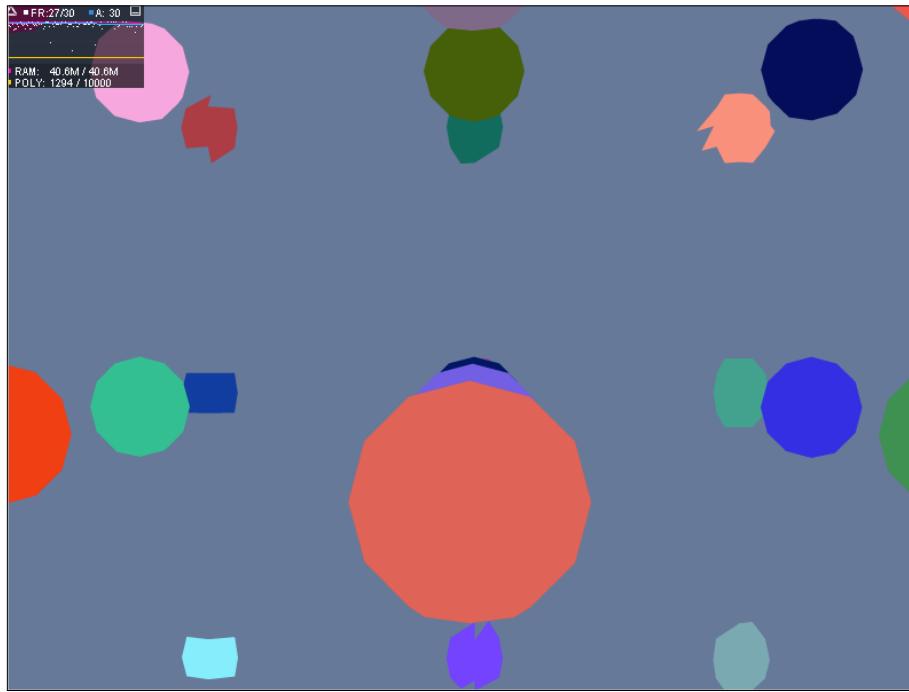
How to do it...

In the following program, we create an array of rows and columns of spheres which extend into the depth. This formation will allow us to see the `MaxPolyFilter` in action:

PolyFilterDemo.as

```
package
{
    public class PolyFilterDemo extends AwayTemplate
    {
        private var _fpsWalker:FPSController;
        public function PolyFilterDemo()
        {
            super();
            initFilter();
        }
        override protected function initGeometry() : void{
            for(var i:int=0;i<5;++i){
                for(var b:int=0;b<5;++b){
                    for(var c:int=0;c<5;++c){
                        var sp:Sphere=new Sphere({radius:15,material:new ColorMaterial(Math.floor(Math.random()*0xFFFFFFFF))});
                        _view.scene.addChild(sp);
                        sp.x=i*100-200;
                        sp.y=b*100-100;
                        sp.z=c*100+40;
                    }
                }
            }
            _fpsWalker=new FPSController(_cam,stage,20,5,40);
        }
        override protected function onEnterFrame(e:Event) : void{
            super.onEnterFrame(e);
            _fpsWalker.walk();
        }
        private function initFilter():void{
            _view.renderer=new BasicRenderer(new MaxPolyFilter(500));
        }
    }
}
```

As you can see in the following screenshot, more distant spheres' geometry is partially stripped from the triangles as they exceed the poly count allowed by us:



How it works...

`MaxPolyFilter` starts to remove the faces exceeding the defined limit from the furthest objects first. This way, it behaves partially like `ZDepthFilter`, but instead of removing whole objects, it removes their faces until the complete disappearance of the model. Now let's see the code. With the `initGeometry()` method, we set up an array of spheres which, as you can see, extends into the depth. Then we initiate a new `BasicRenderer` and pass into its constructor a new instance of `MaxPolyFilter`, setting the maximum number of polygons to 500:

```
private function initFilter():void{
    _view.renderer=new BasicRenderer(new MaxPolyFilter(500));
}
```

See also

Chapter 2, Creating an FPS (first person shooter) controller.

Working with LOD objects

LOD (Level Of Detail) objects is a widespread concept in real-time game engines. The basic idea behind it is that the further the object from the player position, the lower its level of detail. This way, you reduce the number of faces to be rendered on the distant objects which don't even need that level of detail because they are hardly visible by the viewer anyway. You may find this technique useful, especially when constructing big urban scenes. Now let's see how to get it to work.

Getting ready

Set up a default Away3D scene, extending the `AwayTemplate` class.

Go to the source code directory of *Chapter 11* and copy the `FPSController.as` class (first person camera control utility that we created in *Chapter 2*).

In your project, create a new package, name it `utils`, and paste `FPSController.as` into it. Now we are able to set a first person controllable camera.

How to do it...

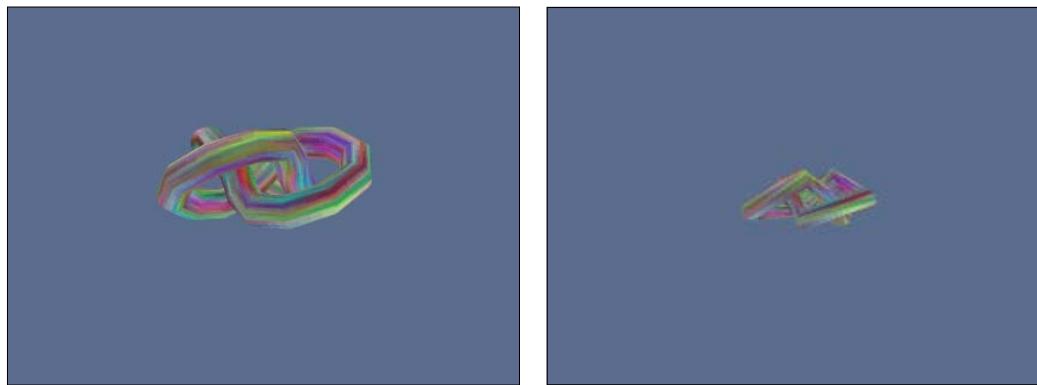
In this example, we set up a `TorusKnot` primitive that is going to have three levels of detail. As we move with our FPS controller closer or further from the model, you will notice how the shape switches its level of detail to the appropriate LOD definition:

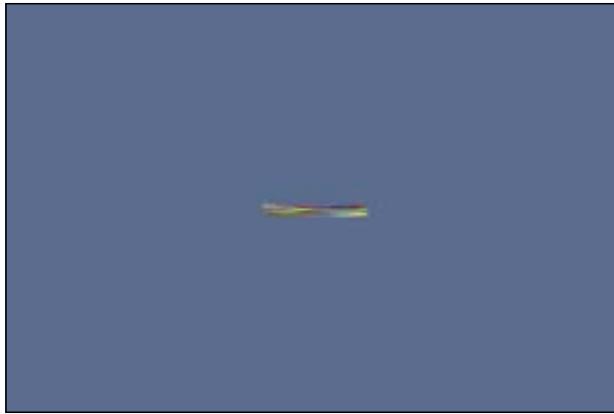
`LODDemo.as`

```
public class LODEmo extends AwayTemplate
{
    private var _fpsWalker:FPSController;
    private var _bitMat:BitmapMaterial;
    public function LODEmo()
    {
        super();
    }
    override protected function initMaterials() : void{
        var bdata:BitmapData=new BitmapData(128,128);
        bdata.perlinNoise(15,15,8,123456,true,true,7);
        _bitMat=new BitmapMaterial(bdata);
    }
    override protected function initGeometry() : void{
```

```
var knotHigh:TorusKnot=new TorusKnot({segmentsR:30,segmentsT:30
, radius:20,material:_bitMat,tube:5});
var knotMedium:TorusKnot=new TorusKnot({segmentsR:10,segmentsT:
10,radius:20,material:_bitMat,tube:5});
var knotLow:TorusKnot=new TorusKnot({segmentsR:6,segmentsT:6,ra
dius:20,material:_bitMat,tube:5});
var highLOD:LODOBJECT=new LODOBJECT(knotHigh);
highLOD.minp=2.3;
highLOD.maxp=Infinity;
var medLOD:LODOBJECT=new LODOBJECT(knotMedium);
medLOD.minp=0.8;
medLOD.maxp=2.3;
var lowLOD:LODOBJECT=new LODOBJECT(knotLow);
lowLOD.minp=0;
lowLOD.maxp=0.8;
_view.scene.addChild(highLOD);
_view.scene.addChild(medLOD);
_view.scene.addChild(lowLOD);
highLOD.z=medLOD.z=lowLOD.z=300;
_fpsWalker=new FPSController(_cam,stage,20,5,40);
}
override protected function onEnterFrame(e:Event) : void{
super.onEnterFrame(e);
_fpsWalker.walk();
}
}
```

At this sequence of screenshots, we can see the resulting three different levels of detail for the same TorusKnot object:





How it works...

The setup of LOD objects consists of three steps. First we have to instantiate the models with different levels of detail.



You are not limited to differentiate only between the mesh quality properties. You may change any property of the model. In the end, it is all about swapping between different models.



In our case, these are three Away3D TorusKnot primitive instances with a different number of polygons, as you can see in the `initGeometry()` function:

```
var knotHigh:TorusKnot=new TorusKnot({segmentsR:30,segmentsT:30,rad  
ius:20,material:_bitMat,tube:5});  
var knotMedium:TorusKnot=new TorusKnot({segmentsR:10,segmentsT:  
10,radius:20,material:_bitMat,tube:5});  
var knotLow:TorusKnot=new TorusKnot({segmentsR:6,segmentsT:6,ra  
dius:20,material:_bitMat,tube:5});
```

Next we define a unique `LODObject` instance for each of these models:

```
var highLOD:LODObject=new LODObject(knotHigh);  
highLOD.minp=2.3;  
highLOD.maxp=Infinity;  
var medLOD:LODObject=new LODObject(knotMedium);  
medLOD.minp=0.8;  
medLOD.maxp=2.3;  
var lowLOD:LODObject=new LODObject(knotLow);  
lowLOD.minp=0;  
lowLOD.maxp=0.8;
```

LODObject constructor accepts the model as its parameter. For each LODObject, we need to define the range at which it is going to appear. For this, we use `minp` and `maxp` properties. One thing you should understand is that the numeric value you assign is not a distance in pixels, but a value of the perspective projection scale at which the object is viewed. It works like this. `minp` property defines the furthest boundary of the range and `maxp` does the closest to the camera. Setting `minp` to zero means that the distance is infinite because projection zero doesn't project at all. In our case, lowLOD object is set when the projection value equals or is less than 0.8. Now if you look at the opposite side, closer to the camera, we defined highLOD object to appear when the projection is at 2.3, which is pretty close to the camera position and its `maxp` is set to infinity, which means the position of the camera where the perspective scale is infinite.

The last step is to add the LOD objects we have created to the scene. It is important to position them at the same place so that their swaps will look seamless. In the following code, we just added all of them directly to the scene defining z position =300:

```
_view.scene.addChild(highLOD);
_view.scene.addChild(medLOD);
_view.scene.addChild(lowLOD);
highLOD.z=medLOD.z=lowLOD.z=300;
```



An alternative approach, when we talk about more than 3 LOD objects, may be the creation of ObjectContainer3D, adding all the LODs to it, and then to manipulate the world position of that container only, instead of tweaking each LODObject separately.

Optimizing geometry by welding vertices

Good modeling practices are essential for optimization. The model quality depends primarily on the proficiency of a modeler. However, you may find yourself in many situations using models that were purchased from third party sources or even models created by a professional, but without careful inspection of errors. You can get additional optimization of your models checking them for duplicated vertices and UVs, which are very widespread modeling flaws. Removal of extra vertices may help improve the performance (especially if there are lots of these) because each vertex is involved in matrix transformation calculations, which are memory intensive by default, but an even more intensive process is the drawing of unnecessary triangles, which can be eliminated by cleaning the geometry. In this recipe, you will learn how to optimize your model by removing double vertices and UVs by using the Away3D Weld tool.

Getting ready

1. Set up a default Away3D scene with `AwayTemplate` as a base class.
2. In this example, we make use of GUI controls, which are part of a Flash components library developed by Keith Peters and called `MinimalComps`. Go and get it for free here: <http://www.minimalcomps.com/>. Put the SWC into your SWC folder or connect it directly to your project.
3. You also need to import a model of the Hummer vehicle which you can find in *Chapter 11*'s `assets/models` folder under the name `H2.3ds` and its texture `H2.jpg`. Put it into the corresponding assets folder of your project.

How to do it...

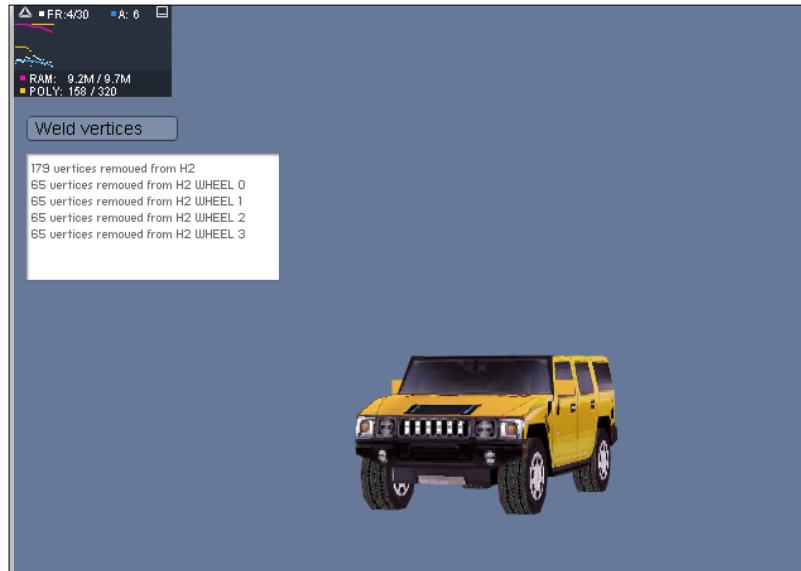
In the following example program, we load a model which has got a bunch of double vertices. Then we pass all its parts through the `Weld` class which cleans all the junk for us:

`WeldDemo.as`

```
package
{
    public class WeldDemo extends AwayTemplate
    {
        [Embed(source="/assets/models/H2.3DS", mimeType="application/octet-
stream")]
        private var CarModel:Class;
        [Embed(source="/assets/models/H2.jpg")]
        private var CarTexture:Class;
        private var _carModel:ObjectContainer3D;
        private var _bitmat:BitmapMaterial;
        private var _tField:Text;
        private var _weld:Weld;
        public function WeldDemo()
        {
            super();
            initGUI();
            _cam.z=-500;
        }
        override protected function initMaterials() : void{
            _bitmat=new BitmapMaterial(Bitmap(new CarTexture()).bitmapData);
        }
        override protected function initGeometry() : void{
            _view.renderer=Renderer.CORRECT_Z_ORDER;
            var dae:Collada=new Collada();
```

```
var max3d:Max3DS=new Max3DS();
    _carModel= max3d.parseGeometry(new CarModel())as
ObjectContainer3D;
    _carModel.scale(0.6);
    _carModel.rotate(Vector3D.X_AXIS,-90);
    _view.scene.addChild(_carModel);
    for each(var child:Mesh in _carModel.children){
        child.material=_bitmat;
    }
}
override protected function onEnterFrame(e:Event) : void{
    super.onEnterFrame(e);
    _carModel.rotationY+=3;
}
private function initGUI():void{
    var but:Button=new Button("Weld vertices",120);
    but.x=10;
    but.y=90;
    this.addChild(but);
    but.addEventListener(MouseEvent.CLICK,onClick);
    _tField=new Text(this,10,120);
}
private function onClick(e:MouseEvent):void{
    applyWeld();
}
private function applyWeld():void{
    _weld=new Weld();
    _tField.text="";
    for each(var child:Mesh in _carModel.children){
        _weld.apply(child);
        _tField.text+="_weld.countvertices+" +"vertices removed
from"+ "+child.name+"\n";
    }
}
}
```

The following screenshot shows our yellow hummer. The window showing the information about the welded vertices is located at the top-left corner:



How it works...

We will skip the model parsing section as you can get all the explanation on this in *Chapter 9*. Let's begin with the `initGUI()` function. We set up a button which is going to trigger the execution of the Weld process. We also add a Text control where we will print the results of the welding so that you can see how many vertices have been deleted from each part of the model.

The welding itself is executed within the `applyWeld()` method, which gets called when the **Weld vertices** button is clicked:

```
private function applyWeld():void{
    _weld=new Weld();
    _tField.text="";
```

First we create a new instance of the `Weld()` class. Now, because `_carModel` is an `ObjectContainer3D` which contains several child objects which are four wheels and the cabin of the Hummer, we should run the `_weld.apply()` command on each of these parts and not on their parent container, which in this case will throw an error. So using `for each` loop statement we iterate through each child object of `_carModel` applying to it `weld()`:

```
for each(var child:Mesh in _carModel.children){
    _weld.apply(child);
    _tField.text+="_weld.countvertices+" +"vertices removed
from"+ " "+child.name+"\n";
}
```

You will surely be surprised to see that, despite the fact that we have removed 439 unnecessary vertices; the overall performance has not really changed. The reason for this is not because Weld is not effective, but rather because the removed quantity is still not enough in this particular case in order to improve the overall performance. If you build a scene containing many models that you pass through a similar process, the performance will increase and it will then be much more noticeable.

There's more...

Additionally, you can weld your model's vertices inside Prefab with a single button click. Inside Prefab, we have to go through the following steps:

- ▶ Load your 3D model into Prefab
- ▶ If the model contains several sub-objects, select each of them then in the top toolbar and select **Geometry**.
- ▶ Inside the **Geometry** drop-down list, choose **Weld (selection)**.
- ▶ After clicking, you will be shown a pop-up window which informs you how many vertices and UVs were deleted. If you see that the number is a zero, it means that your model is fine and doesn't contain double vertices and UVs.

Now you can export your model to one of the destination formats using different Prefab exporters.



Note that less vertices will also shorten the overall parsing time for the model.



See also

Chapter 8, Prefab3D.

Excluding objects from rendering

This short recipe will teach you the ways to exclude your 3D objects from the rendering process. You may want to be able to stop rendering some of your scene geometry mostly for the sake of achieving a better performance. Fortunately, Away3D does most of the job for you automatically.

Getting ready

Set up a default Away3D scene with `AwayTemplate` as a base class.

In this example, we make use of GUI controls which are part of a Flash components library developed by Keith Peters and called `MinimalComps`. Go and get it for free from <http://www.minimalcomps.com/>. Put the `.SWC` into your `SWC` folder or connect it directly to your project.

How to do it...

In the following example, we set up a group of plane primitives, which are positioned in a spherical formation. Every plane consists of 18 triangles intentionally so that we can get a slightly "heavy" scene. Use one of the two checkboxes located at the right side to switch between two possible ways to stop objects from being rendered:

`SelectiveRenderDemo.as`

```
package
{
    public class SelectiveRenderDemo extends AwayTemplate
    {
        private const PHI_STEP:Number=22.5;
        private static const EASE_FACTOR:Number=0.5;
        private const RADIUS:Number=200;
        private var _horizontalStep:Vector.<Number>=new Vector.<Number>();
        private var _vertTilt:Vector.<Number>=new Vector.<Number>();
        private var _numOfObjsPerPass:Vector.<Number>=new
        Vector.<Number>();
        private var _hoverCam:HoverCamera3D;
        private var _oldMouseX:Number=0;
        private var _oldMouseY:Number=0;
        private var _planesArr:Vector.<Plane>=new Vector.<Plane>();
        private var _canRender:Boolean=true;
        public function SelectiveRenderDemo()
        {
            _horizontalStep=Vector.<Number>([0,60,36,30,24,30,36,60,0]);
            _num
            OfObjsPerPass=Vector.<Number>([1,6,10,12,15,12,10,6,1]);
            super();
            initGUI();
        }
        override protected function initGeometry() : void{
```

```

        setHowerCamera();
        buildSphere();
    }
    override protected function onEnterFrame(e:Event) : void{
        super.onEnterFrame(e);
        if(_canRender){
            _hoverCam.panAngle = (stage.mouseX - _oldMouseX)*EASE_FACTOR ;
            _hoverCam.tiltAngle = (stage.mouseY - _oldMouseY)*EASE_FACTOR
        ;
            _hoverCam.hover();
        }
    }
    private function buildSphere():void {
        var colMat:ColorMaterial=new ColorMaterial(0x120922);
        var horizStep:Number;
        var vertStep:Number=PHI_STEP*Math.PI/180;
        for(var i:int=0;i<9;++i){
            horizStep=_horizontalStep[i]*Math.PI/180;
            for(var j:int=0;j<_numOfObjsPerPass[i];++j){

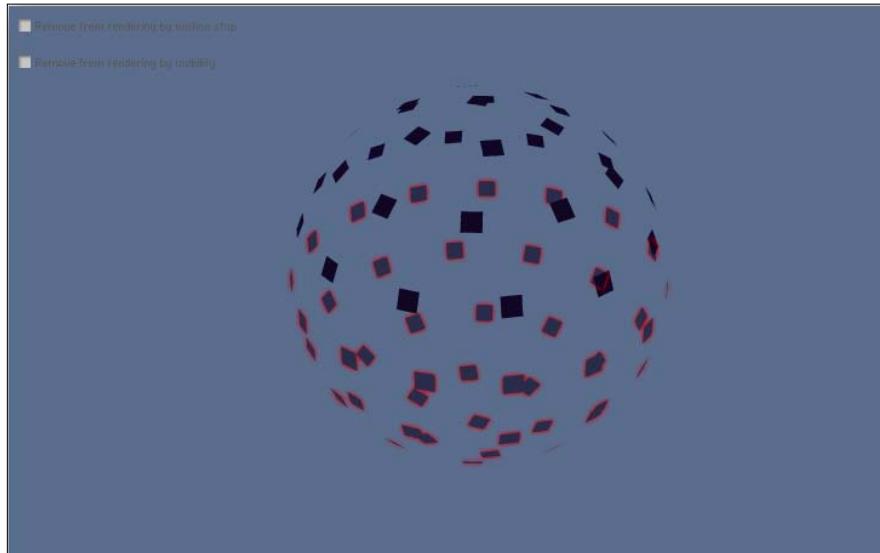
                var pl:Plane=new Plane({width:20,height:20,bothsides:true,
material:colMat,segmentsW:3,segmentsH:3});
                if(j==0&&i==0){
                    pl.yUp=true;
                }else{
                    pl.yUp=false;
                }
                pl.position=new Vector3D(RADIUS*Math.sin(i*vertStep)*Math.
sin(j*horizStep),-RADIUS*Math.cos(i*vertStep),-RADIUS*Math.
sin(i*vertStep)*Math.cos(j*horizStep));
                _view.scene.addChild(pl);
                pl.lookAt(new Vector3D(0,0,0));
                _planesArr.push(pl);
            }
        }
    }
    private function setHowerCamera():void{
        _hoverCam=new HoverCamera3D();
        _view.camera=_hoverCam;
        var dummy:ObjectContainer3D=new ObjectContainer3D();
        _view.scene.addChild(dummy);
        dummy.position=new Vector3D(0,0,0);
        _hoverCam.target=dummy;
    }
}

```

```
    _hoverCam.distance = 1200;
    _hoverCam.maxTiltAngle = 80;
    _hoverCam.minTiltAngle = 0;
    _hoverCam.wrapPanAngle=true;
    _hoverCam.steps=16;
    _hoverCam.yfactor=1;
}
private function initGUI():void{
    var cb:CheckBox=new CheckBox(this,20,90,"Remove from rendering
by motion stop",onCBSelect);
    var cb1:CheckBox=new CheckBox(this,20,120,"Remove from rendering
by visibility",onCB1Select);
}
private function onCBSelect(e:Event):void{
    if(e.target.selected){
        processPlanes(true);
        _canRender=false;
    }else{
        processPlanes(false);
        _canRender=true;
    }
}
private function onCB1Select(e:Event):void{
    if(e.target.selected){
        makeInvisible(true);
    }else{
        makeInvisible(false);
    }
}
private function processPlanes(exclude:Boolean):void{
    for(var i:int=0;i<_planesArr.length;++i){
        if(exclude){
            _planesArr[i].ownCanvas=true;
        }else{
            _planesArr[i].ownCanvas=false;
        }
    }
}
private function makeInvisible(exclude:Boolean):void{
    for(var i:int=0;i<_planesArr.length;++i){
        if(exclude){
            if(i%2==0){
                _planesArr[i].visible=false;
            }
        }
    }
}
```

```
        }else{
            _planesArr[i].visible=true;
        }
    }
}
```

And here is the resulting sphere that you should see on your screen:



How it works...

We begin by creating small planes and positioning them in a spherical form. We position the planes by implementing a standard spherical coordinate equation:

```
X=Radius * (Math.sin(phi)*Math.sin(theta));  
Y=Radius*- (Math.cos(phi));  
Z=Radius*- (Math.sin(phi)*Math.cos(theta));
```

We also set each plane number of triangles to 18 in order to overload the CPU a little bit.

Basically, what we do in the previous example program is exclude objects from the rendering process by either making them invisible or ceasing the camera's transformation completely. Turning on the `Invisibility()` method is pretty logical because the invisible objects are simply not drawn by the engine.



There is an exception to this concept if you deal with render sessions such as `SpriteSession` or with the `ownCanvas` containers, which wrap the rendered objects with sprites. Using these, you can achieve zero visibility of the objects by setting their `alpha` property =0. In this case, the objects, despite being invisible, are still being rendered and you can see it by the unchanged frame rate.

Now, for the second method, Away3D implements a technique called triangle cache, which keeps track of the objects which are not transformed. Each object that is not being moved, scaled, or rotated during the runtime is automatically skipped by the renderer.



If you want to force all the static objects to be rendered anyway, you can set the `View3D forceUpdate` property to true.

Image size consideration

Although Flash performs a fairly good job compressing external images on import, it is still important to optimize them before importing into your project. You should always remember two important factors when you prepare your bitmaps for Flash in a whole. These are image dimensions (size) and quality (resolution). We won't dive into pixel theory, but the rule of thumb is to reduce both of them to the level which allows you to keep your image appearance as close as possible to its original quality. Many times, developers tend to create huge bitmaps, especially when these are baked textures because they mistakenly suppose that if the texture maps across many different surfaces, it has to be big. But the truth is that in most cases, you may have the same texture with a size such as 256x256 which will appear the same on your model as 1024x1024 because it matches the target object scale.

Getting ready

Create a new class, give it the name `BitMapDemo`, and make sure it extends `AwayTemplate`.

Then, in the source code directory of *Chapter 11*, find the `utils` folder which contains the `FPSController.as` class and copy it into your project root directory.

How to do it...

For this demonstration, we will create six cube primitives. We will also generate a set of six `BitmapData` objects with Perlin Noise fill, which will serve as texture maps for the side material of each cube. The bitmaps have been intentionally created with dimensions of

1024x1024 in order to show how it influences the performance. Play around with WIDTH and HEIGHT constants to get an idea of how important it is to keep texture sizes as low as possible:

BitMapDemo.as

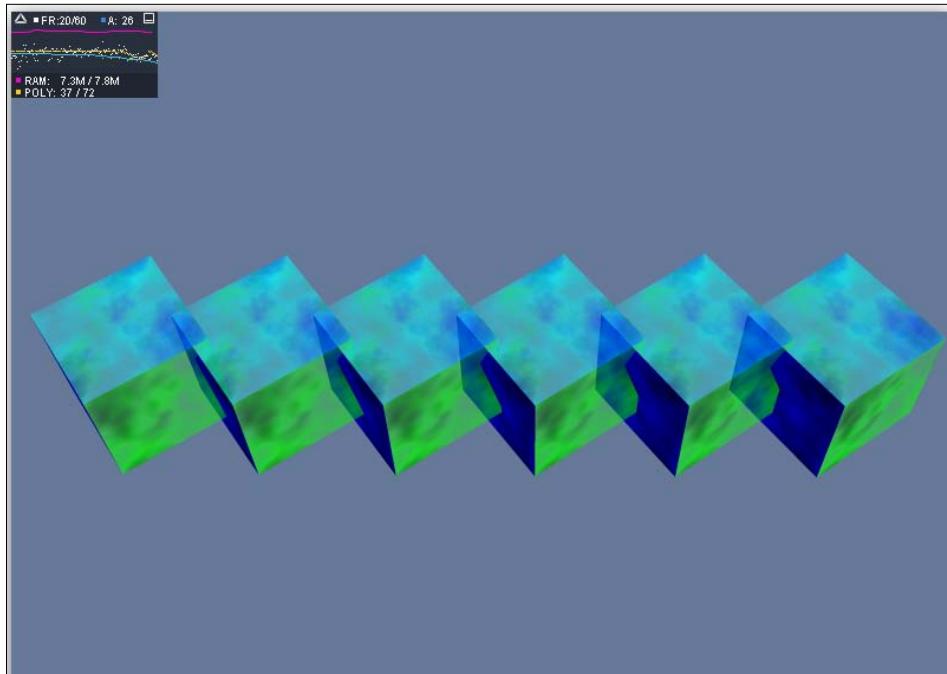
```
package
{
    public class BitMapDemo extends AwayTemplate
    {
        private const WIDTH:Number=1024;
        private const HEIGHT:Number=1024;
        private var _cube:Cube;
        private var _bDataArr:Vector.<BitmapData>=new
        Vector.<BitmapData>();
        private var _fpsWalker:FPSController;
        private var _cubesArr:Vector.<Cube>=new Vector.<Cube>();
        public function BitMapDemo()
        {
            super();
            _fpsWalker=new FPSController(_cam,stage,20,5,40);
        }
        override protected function initMaterials() : void{
            for(var i:int=0;i<6;++i){
                var bData:BitmapData=new BitmapData(WIDTH,HEIGHT);
                bData.perlinNoise(128,128,Math.floor(Math.random()*10),2345,true,
                    true,i*2+4,false,[i*10,i*10/2]);
                _bDataArr.push(bData);
            }
        }
        override protected function initGeometry() : void{
            for(var i:int=0;i<6;++i){
                _cube=new Cube({width:130,height:130,depth:130});
                _cube.cubeMaterials.front=new BitmapMaterial(_
                bDataArr[0],{smooth:false});
                _cube.cubeMaterials.back=new BitmapMaterial(_
                bDataArr[1],{smooth:false});
                _cube.cubeMaterials.top=new BitmapMaterial(_
                bDataArr[2],{smooth:false});
                _cube.cubeMaterials.bottom=new BitmapMaterial(_
                bDataArr[3],{smooth:false});
                _cube.cubeMaterials.left=new BitmapMaterial(_
                bDataArr[4],{smooth:false});
            }
        }
    }
}
```

Optimizing Performance

```
    _cube.cubeMaterials.right=new BitmapMaterial(_
bDataArr[5],{smooth:false});
    _view.scene.addChild(_cube);
    _cube.position=new Vector3D(0,0,i*140);
    _cubesArr.push(_cube);
}

}
override protected function onEnterFrame(e:Event) : void{
super.onEnterFrame(e);
_fpsWalker.walk();
for(var i:int=0;i<_cubesArr.length;++i){
    _cubesArr[i].rotationX+=3;
    _cubesArr[i].rotationZ+=2;
}
}
}
}
```

Here is the view you should get after moving the `FPSController` in front of the cubes' row:



How it works...

If you tried to run the previous program with the texture sizes of 1024x1024 and after that with 256x256, setting the camera at the same position, as it is shown in the screenshot, you may have noticed the following difference (of course, the actual numbers vary depending on your machine) at AwayStats window in the top-left corner:

6 bitmaps-256x256	6 bitmaps-1024x1024
FPS:28-39 (out of 60)	FPS:20-29 (out of 60)
RAM 7.5	RAM 30.1

Although, on average, the scene with the textures of 1024x1024 still gives around 24-26 FPS, look at how much larger RAM consumption has become. If you add a few more textures of the same dimensions into such a scene, its frame rate would decline very quickly. At the same time, using maps with 256x256 gives a better frame rate, three times lower RAM, and the same image quality. In fact, for these cubes, you could even go further and make the textures only 128x128 and still get the same quality.

Remember that in this quick demo, we generated the bitmaps on the fly inside Flash. Therefore, we did not have much control over their optimization, except for dimension. While working with external images, you should use professional image processing software such as Adobe Photoshop to optimize your textures.

Important tips for performance optimization

General Flash-related rules:

1. Use the `stage.quality` property during runtime to relieve the CPU during memory intensive operations. For example, while you perform a camera flythrough, you can set `stage.quality` to `LOW` or `MEDIUM` as the camera transformation involves memory eating matrix calculations. At the end of the process, you can switch back to your preferred stage quality.
2. The size of the stage matters. Keep the stage as small as your application specifications allow you to.
3. Make use of bitwise operators as they speed up math calculations, in some cases, 100-300%.
4. When working with loops where you iterate through long arrays of data, try using ActionScript 3.0 Vector instead of Array.
5. ActionScript filters are memory intensive.



To Flex users: Don't get frustrated too fast if the frame rate of your application appears lower than you have expected in debug mode. The debug mode compilation is always heavier due to the additional utilities Flex loads during the runtime. In many cases, the frame rate of the release build is twice as fast than the debug version.

Away3D-related tips:

1. Using different renderers and rendering modes makes a huge difference on the frame. `rate.BasicRenderer` is much faster than `QuadrantRenderer`. And within the `BasicRenderer` rendering modes, `Renderer.CORRECT_Z_ORDER` and `Renderer.INTERSECTING_OBJECTS` are considerably CPU intensive.
2. If you don't really need `Frustum` or `Nearfield` clipping, try to avoid using them.
3. Use the simplest materials possible. The most memory consuming materials are of Pixel Bender shaded series. Also, all the shaded materials especially those which use a light source input may completely kill your CPU if applied to many objects. If you use `MovieMaterial`, you can turn off the `autoUpdate` property so that the `MovieClip` content would not be redrawn constantly. Use the `update()` method to switch on redraws if its content needs to be redrawn.
4. Fake scene light sources using baked light maps (see *Chapter 8, Prefab3D*). Dynamic lights have a strong impact on the overall performance.
5. When using shaded material, you can save CPU cycles by caching the light information. Setting the property `surfaceCache` to `true` does the job. The material surface will be updated only when the light source position changes.
6. Recycle material sources. There is no need to create a new material with the same texture for several different objects. You can use the same one for all of them without populating more memory space.
7. Make sure to run a thorough clean up of your disposed objects. Remember to use weak reference on event listeners in order to enable Flash Player garbage collector to clean all the removed objects. If you remove an object which had a listener attached, it would still have a pointer in the memory and will be skipped by the garbage collector for this reason. Also, remember to remove bitmap data objects by using the `dispose()` command.
8. Don't render your entire scene continuously if you don't need to update transformations or visual appearance. Try to use the `View3D.render()` command only when you wish to update objects whose states have changed.
9. Rotations and translations in 3D space are memory intensive as they involve complex matrix calculations. Although the current version of Away3D implements Flash Player 10 3D API `Vector3D` and `Matrix3D` classes still try to avoid stuffing your scene with numerous simultaneous transformations.

10. Model your 3D objects as low poly as possible. At the time of writing, Away3D scene can manage without significant frame rate drop around 3,000-5,000 triangles inside a well optimized application.
11. If you don't need to present a 3D model from different angles use Away3D Sprites such as Sprite2D/3D, MovieClipSprite, which are able to fake 3D objects appearance with billboarding.
12. If you plan to develop big indoor scenes consider using Away3D BSP trees. The approach is relatively complex but you will benefit eventually from the huge performance difference if compared with the same scene created in the regular way.
13. Make sure your imported image assets are in "power of two" (128x128, 256x256, and so on). This is essential for better UV mapping as well as for Flash Player native MIP mapping process where the scaled down bitmaps are resized automatically by the player. For this to work, the bitmaps must be in power of two, not bitmap cached (MovieClips with cacheAsBitmap=true would not work). Also, the image dimensions must be even. For example, 1024x1024 is fine, but 512x1024 would not work.
14. When creating models which consist of several parts with different textures, make use of texture baking into one single map instead of assigning each mesh its own texture file.
15. If you need to incorporate physics in your project, try to avoid using physics libraries such as JigLib, which are heavy in terms of memory consumption. If you need basic physical behaviors, try to code on your own instead, or use 2D physics engines such as **Box2DFlash**, which are lighter in terms of performance.
16. If you are familiar with Adobe Pixel Bender, you should know that besides its regular usage, that is, image data processing, it can be used to calculate any type of data such as vector and matrix operations, because Pixel Bender filter runs on a separate thread, you can defer a significant load of intensive calculations to it from the parent application (for advanced programmers).

13

Doing More with Away3D

In this chapter, we will cover:

- ▶ Moving on uneven surfaces
- ▶ Detecting collisions between objects in Away3D
- ▶ Creating cool effects with animated normal maps
- ▶ Creating skyboxes and their textures
- ▶ Running heavy scenes faster using BSP trees

Introduction

In this chapter, you are going to learn how to move the FPS camera on a sloped terrain using a built-in Away3D topography elevation reading functionality, how to check intersection (collision) between 3D objects, and how to create big indoor scenes with minimal CPU impact using BSP trees. So let's get started.

Moving on uneven surfaces

If you work on an application such as a First Person Shooter (FPS), the chances are high that you would need your character to move not only on perfectly horizontal surfaces, but also on an elevated terrain. Away3D has got a cool utility exactly for this purpose called `ElevationReader`, which allows you to get the pixel data of a specific coordinate on the height map and convert it into the elevation (Y) coordinate value. In this recipe, you will learn how to set up FPS moving on uneven terrain using `ElevationReader`.

Getting ready

1. Create a new class which extends `AwayTemplate` and name it `ElevationReaderDemo`.
2. In this example, we make use of GUI controls which are part of a Flash components library developed by Keith Peters and called `MinimalComps`. Go and get it for free here: <http://www.minimalcomps.com/>. Put the `.SWC` into your `SWC` folder or connect it directly to your project.
3. We use two bitmaps, one is height map, which is used to generate a terrain, and another one is for the terrain's material. Go to the `assets/images` folder of this chapter's source code directory and paste the `readermap1.jpg` and `terrainMap1.jpg` images into your project's assets folder. You can generate your own height map for `ElevationReader` inside Prefab using Terrain Generator (see the Prefab chapter for Terrain Generator use).
4. Create a new package, name it `utils`, and copy into it the `FPSController.as` file from the source code directory. This is a slightly modified version of the FPS controller we have been using throughout the book. The exception is that for its default y-axis value, it gets its elevation from the `ElevationReader`.

How to do it...

`ElevationReaderDemo.as`

```
package
{
    public class ElevationReaderDemo extends AwayTemplate
    {
        [Embed(source="../assets/images/readermap1.jpg")]
        private var ElevMap : Class;

        [Embed(source="../assets/images/terrainMap1.jpg")]
        private var HeightMapTexture : Class;
        private var _elevation:Elevation;
        private var _skinExtrud:SkinExtrude;
        private var _bitMat:BitmapMaterial;
        private var _reader:ElevationReader;
        private var _walker:FPSController;
        private var _bdata:BitmapData;
        private var _tf:Label;
        public function ElevationReaderDemo()
        {
    
```

```
super();
_view.clipping=new NearfieldClipping();
initGUI();
}
private function initGUI():void{
    var tFormat:TextFormat=new TextFormat(null,11,0xFF0011);
    var label:Label=new Label(this,10,100,"Current Elevation:");
    _tf=new Label(this,label.x+110,100);
    _tf.textField.defaultTextFormat=tFormat;
    tFormat.color=0x000000;
    label.textField.defaultTextFormat=tFormat;
}
override protected function initMaterials() : void{
    _bitMat=new BitmapMaterial(Cast.bitmap(new HeightMapTexture()));
    _bdata=Cast.bitmap(new ElevMap());
}
override protected function initGeometry() : void{
    _elevation=new Elevation();
    _elevation.minElevation=0;
    _elevation.maxElevation=255;

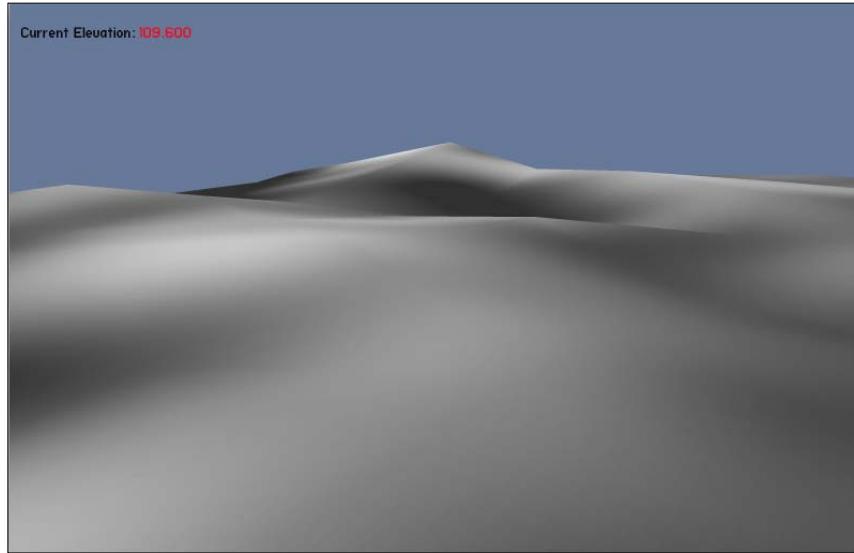
    var elevatonData:Array=_elevation.generate(_bdata,"r",70,70,5,5,0.8);
    _skinExtrud=new SkinExtrude(elevatonData,{recenter:true,
closepath:false, coverall:true, subdivision:1, bothsides:false,
flip:false});
    _skinExtrud.material=_bitMat;
    _skinExtrud.rotationX=90;
    _view.scene.addChild(_skinExtrud);
    _skinExtrud.position=new Vector3D(-100,-100,0);
    _reader=new ElevationReader(3);
    _reader.traceLevels(_bdata,"r",70,70,5,5,0.8);

    _cam.x=_skinExtrud.x;
    _cam.y=1000;
    _cam.z=_skinExtrud.z+200;
    _skinExtrud.ownCanvas=true;
    _walker=new FPSController(_view.camera,stage);
}
override protected function onEnterFrame(e:Event) : void{

    var level:Number=_reader.getLevel(_cam.x,_cam.z,20);
    _tf.text=level.toPrecision(6);
    _walker.walk(level);
```

```
        super.onEnterFrame(e);  
    }  
}  
}  
}
```

Here is our procedural terrain:



How it works...

There are basically three steps (besides creating a height map) you need to take to accomplish this application. First you have to set up `Elevation` objects which generate mesh coordinates based on height map bitmap data. We instantiate the object inside the `initGeometry()` method:

```
_elevation=new Elevation();  
_elevation.minElevation=0;  
_elevation.maxElevation=255;
```

`minElevation/maxElevation` allows you to define the color range of the height channel to use for coordinates generation. 0 to 255 means we use all the color data in the image for the extrusion factor.

Next, we extract the mesh data into an elevation array object, which we will need soon to create a terrain:

```
var elevationData:Array=  
_elevation.generate(_bdata,"r",70,70,5,5,0.8);
```

It is important to pass into `_elevation.generate()` the height map source bitmap data channel to read from (in this case, red because the height map uses the red channel to represent the elevation gradient). Also `subdivisionX` and `subdivision` are important values which enable you to define the mesh triangles quantity. The lower the values, the more triangles are in the mesh. `FactorX` and `FactorY` define the scale of the mesh and the last argument is the amount of the elevation (a.k.a. extrusion).

The second step is to render to a visible object the mesh data generated by the `_elevation` object. To achieve this, we use the `SkinExtrude()` class which generates a terrain based on the array input of the `Elevation()` object:

```
_skinExtrud=new SkinExtrude(elevatonData,{recenter:true,  
closepath:false, coverall:true, subdivision:1, bothsides:false,  
flip:false});  
_skinExtrud.material=_bitMat;  
_skinExtrud.rotationX=90;  
_view.scene.addChild(_skinExtrud);  
_skinExtrud.position=new Vector3D(-100,-100,0);
```

`SkinExtrude` inherits from `mesh`, so we treat it as a regular `Object3D` display object assigning to it a material and adding it to the scene.

The last step is to set the `ElevationReader` instance. We instantiate it by passing into its constructor a smoothing factor which is important to reduce the jerky behavior of the camera when its elevation (Y coordinate) is updated:

```
_reader=new ElevationReader(3);
```

Now we call the `traceLevels()` method of `_reader` and pass into it exactly the same arguments we passed earlier into `_elevation.generate()`. We should do it so that the reader would extract the correct bitmap data values matching `_reader`'s current position:

```
_reader.traceLevels(_bdata,"r",70,70,5,5,0.8);
```

Next, we set up an instance of `FPSController` which has to be passed camera and stage references. The actual elevation reading is executed inside the `onEnterFrame()` function. We call the `_reader.getLevel()` method which returns an elevation number value based on the camera X and Z position:

```
var level:Number=_reader.getLevel(_cam.x,_cam.z,20);
```

Then we pass this value as a Y coordinate into the `_walker.walk()` method which updates the FPS controller camera's Y position to the current elevation value:

```
_walker.walk(level);
```

See also

Chapter 8, *Prefab3D, Creating terrain* recipe.

Detecting collisions between objects in Away3D

This recipe will teach you the fundamentals of collision detection between objects in 3D space. We are going to learn how to perform a few types of intersection tests. These tests can hardly be called collision detection in their physical meaning, as we are not going to deal here with any simulation of collision reaction between two bodies. Instead, the goal of the recipe is to understand the collision tests from a mathematical point of view. Once you are familiar with intersection test techniques, the road to creating of physical collision simulations is much shorter. There are many types of intersection tests in mathematics. These include some simple tests such as **AABB (axially aligned bounding box)**, Sphere - Sphere, or more complex such as Triangle - Triangle, Ray - Plane, Line - Plane, and more. Here, we will cover only those which we can achieve using built-in Away3D functionality. These are AABB and AABS (axially aligned bounding sphere) intersections, as well as Ray-AABS and the more complex Ray-Triangle. The rest of the methods are outside of the scope of this book and you can learn about applying them from various 3D math resources.

Getting ready

Setup an Away3D scene in a new file extending `AwayTemplate`. Give the class a name `CollisionDemo`.

How to do it...

In the following example, we perform an intersection test between two spheres based on their bounding boxes volumes. You can move one of the spheres along X and Y with arrow keys onto the second sphere. On the objects overlapping, the intersected (static) sphere glows with a red color.

AABB test:

`CollisionDemo.as`

```
package
{
    public class CollisionDemo extends AwayTemplate
    {
        private var _objA:Sphere;
        private var _objB:Sphere;
```

```
private var _matA:ColorMaterial;
private var _matB:ColorMaterial;
    private var _gFilter:GlowFilter=new GlowFilter();
public function CollisionDemo()
{
    super();
    _cam.z=-500;
}
override protected function initMaterials() : void{
    _matA=new ColorMaterial(0xFF1255);
    _matB=new ColorMaterial(0x00FF11);
}
override protected function initGeometry() : void{
    _objA=new Sphere({radius:30,material:_matA});
    _objB=new Sphere({radius:30,material:_matB});
    _view.scene.addChild(_objA);
    _view.scene.addChild(_objB);

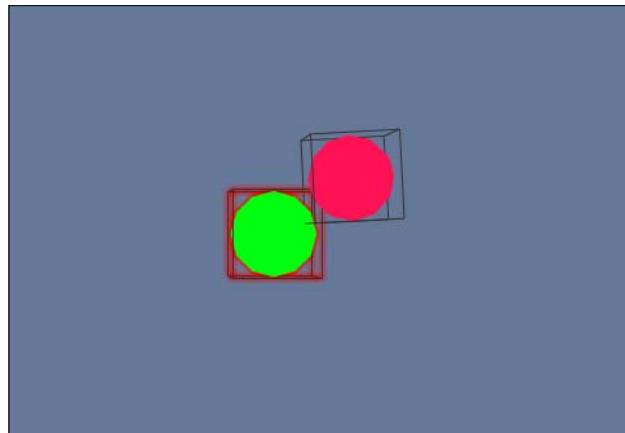
    _objB.ownCanvas=true;
    _objA.debugbb=true;
    _objB.debugbb=true;
    _objA.transform.position=new Vector3D(-80,0,400);
    _objB.transform.position=new Vector3D(80,0,400);

}
override protected function initListeners() : void{
    super.initListeners();
    stage.addEventListener(KeyboardEvent.KEY_DOWN,onKeyDown);
}
override protected function onEnterFrame(e:Event) : void{

    super.onEnterFrame(e);
    if(AABBTest()){
        _objB.filters=[_gFilter];
    }else{
        _objB.filters=[];
    }
}
private function AABBTest():Boolean{
    if(_objA.parentMinX>_objB.parentMaxX||_objB.parentMinX>_objA.parentMaxX){
        return false;
    }
}
```

```
        if(_objA.parentMinY>_objB.parentMaxY| _objB.parentMinY>_objA.  
parentMaxY) {  
    return false;  
}  
        if(_objA.parentMinZ>_objB.parentMaxZ| _objB.parentMinZ>_objA.  
parentMaxZ) {  
    return false;  
}  
    return true;  
}  
private function onKeyDown(e:KeyboardEvent):void{  
switch(e.keyCode){  
    case 38:_objA.moveUp(5); break;  
    case 40:_objA.moveDown(5); break;  
    case 37:_objA.moveLeft(5); break;  
    case 39:_objA.moveRight(5); break;  
    case 65:_objA.rotationZ-=3; break;  
    case 83:_objA.rotationZ+=3; break;  
    default:  
    }  
}  
}  
}
```

In this screenshot, the green sphere bounding box has a red glow while it is being intersected by the red sphere's bounding box:



How it works...

Testing intersections between two AABBs is really simple. First, we need to acquire the boundaries of the object for each axis. The box boundaries for each axis of any Object3D are defined by a minimum value for that axis and maximum value. So let's look at the `AABBTest()` method. Axis boundaries are defined by `parentMin` and `parentMax` for each axis, which are accessible for each object extending Object3D.

 You can see that Object3D also has `minX,minY,minZ` and `maxX,maxY,maxZ`. These properties define the bounding box boundaries too, but in objects space and therefore aren't helpful in AABB tests between two objects.

So in order for a given bounding box to intersect a bounding box of other objects, three conditions have to be met for each of them:

- ▶ Minimal X coordinate for each of the objects should be less than maximum X of another.
- ▶ Minimal Y coordinate for each of the objects should be less than maximum Y of another.
- ▶ Minimal Z coordinate for each of the objects should be less than maximum Z of another.

If one of the conditions is not met for any of the two AABBs, there is no intersection. The preceding algorithm is expressed in the `AABBTest()` function:

```
private function AABBTest():Boolean{  
    if(_objA.parentMinX>_objB.parentMaxX||_objB.parentMinX>_objA.  
    parentMaxX){  
        return false;  
    }  
    if(_objA.parentMinY>_objB.parentMaxY||_objB.parentMinY>_objA.  
    parentMaxY){  
        return false;  
    }  
    if(_objA.parentMinZ>_objB.parentMaxZ||_objB.parentMinZ>_objA.  
    parentMaxZ){  
        return false;  
    }  
    return true;  
}
```

As you can see, if all of the conditions we listed previously are met, the execution will skip all the return false blocks and the function will return true, which means the intersection has occurred.

There's more...

Now let's take a look at the rest of the methods for collision detection, which are AABS-AABS, Ray-AABS, and Ray-Triangle.

AABS test

The intersection test between two bounding spheres is even simpler to perform than AABBs. The algorithm works as follows.

If the distance between the centers of two spheres is less than the sum of their radius, then the objects intersect. Piece of cake! Isn't it? Let's implement it within the code.



The AABS collision algorithm gives us the best performance. While there are many other even more sophisticated approaches, try to use this test if you are not after extreme precision. (Most of the casual games can live with this approximation).



First, let's switch the debugging mode of `_objA` and `_objB` to bounding spheres. In the last application we built, go to the `initGeometry()` function and change:

```
_objA.debugbb=true;  
_objB.debugbb=true;
```

To:

```
_objA.debugbs=true;  
_objB.debugbs=true;
```

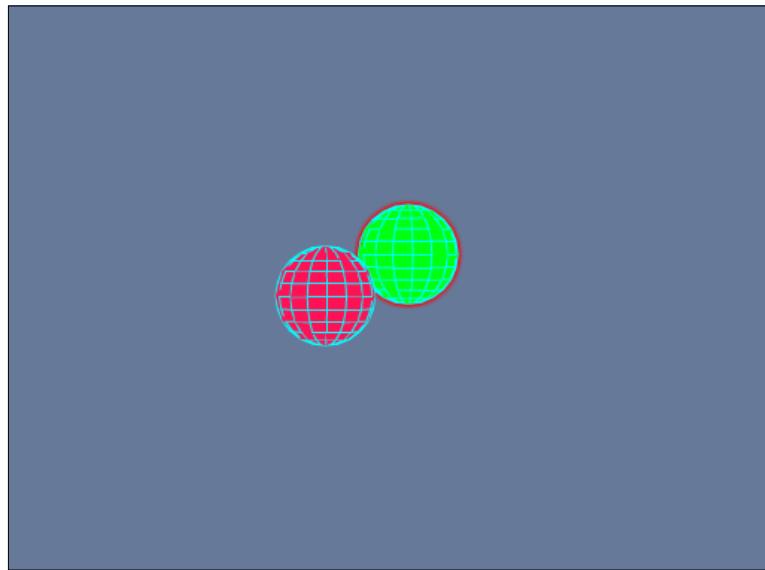
Next, we add the function to the class which implements the algorithm we described previously:

```
private function AABSTest():Boolean{  
    var dist:Number=Vector3D.distance(_objA.position,_objB.  
    position);  
    if(dist<=(_objA.radius+_objB.radius)){  
        return true;  
    }  
    return false;  
}
```

Finally, we add the call to the method inside `onEnterFrame()`:

```
if(AABSTest()){  
    _objB.filters=[_gFilter];  
}else{  
    _objB.filters=[];  
}
```

Each time `AABSTest` returns `true`, the intersected sphere is highlighted with a red glow:



Ray – AABS intersection test

Away3D contains a Ray utility which is used in 3D math to test for intersection. In most general terms, a Ray is a line which has an origin, direction vectors, and length. Ray is used rapidly in game development when there is a need to detect the collision of firing weapons hitting their targets, driving simulators, convert screen points into 3D world coordinates, and much more.

Ray intersection calculations are a relatively complex calculation for one who is not deep into 3D math. Away3D Ray class supplies us with a couple of predefined test methods which save us most of the math intensive calculations and which you are going to learn how to use. In this part, we are going to use the Ray utility to check an intersection with the bounding sphere of a 3D object using the `Ray.intersectBoundingRadius()` method.

So we will continue with the class we have built in the examples. First put the following global variables above the class constructor:

```
private var _lineSeg:LineSegment;
private var _wireMat:WireColorMaterial;
private var _startVertex:Vertex=new Vertex(0,0,0);
private var _endtVertex:Vertex=new Vertex(0,0,0);
```

LineSegment will serve us as a debug-tracer for the ray. As the ray is actually an invisible line, it would be more convenient for us to get its graphical representations. `_startVertex` and `_endVertex` stand for start and end point of the segment line which will receive origin and direction values from the ray as you will see shortly. Now insert the following function into the class:

```
private function RayAABSTest():Boolean{
    var ray:Ray=new Ray();
    ray.orig=_objA.position;
    var d:Vector3D=new Vector3D(0,200,0);
    var dir:Vector3D=_objA.transform.transformVector(d);
    ray.dir=dir;
    _startVertex.x=ray.orig.x;
    _startVertex.y=ray.orig.y;
    _startVertex.z=ray.orig.z;
    _lineSeg.start=_startVertex;
    _endtVertex.x=ray.dir.x;
    _endtVertex.y=ray.dir.y;
    _endtVertex.z=ray.dir.z;
    _lineSeg.end=_endtVertex;
    _view.scene.addChild(_lineSeg);
    _lineSeg.material=_wireMat;
    ray.orig.normalize();
    ray.dir.normalize();
    return ray.intersectBoundingRadius(_objB.position,_objB.boundingRadius);
}
```

Inside `onEnterFrame` method change existing `If .. else...` condition to be like this:

```
if(RayAABSTest()){
    _objB.filters=[_gFilter];
}else{
    _objB.filters=[];
}
```

 There is one important thing you should know regarding this test in Away3D. The method `ray.intersectBoundingRadius()` returns an intersection only if the ray direction (end point) vector is inside the bounding sphere of the object. That is, if the ray is intersecting the sphere, but its direction vector endpoint doesn't lay inside the bounding radius, the intersection test returns false.

Let's see in detail what is going on inside `RayAABSTest()`. We have to define the origin for the ray. We set it to be the position of the `_objA` sphere, which we are able to move around the scene:

```
ray.orig=_objA.position;
```

Now we have to define the ray direction which is also a vector. We wish that the direction vector of the ray will face the same direction as the `_objA` sphere local y-axis (upwards). These lines help us to achieve these tasks by transforming the direction vector we defined into the `_objA` space:

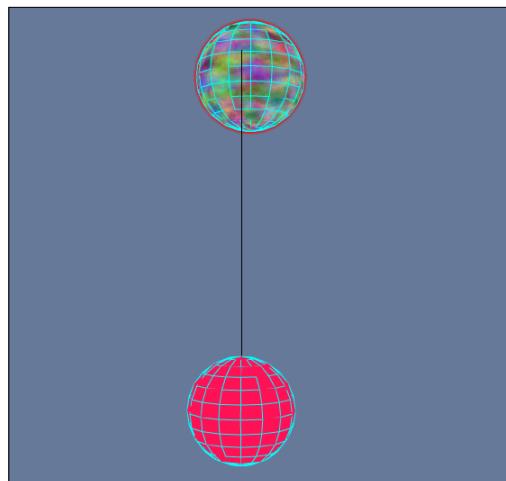
```
var d:Vector3D=new Vector3D(0,200,0);
var dir:Vector3D=_objA.transform.transformVector(d);
ray.dir=dir;
```

Next, we draw a segment line from the ray origin to its direction end vector to get a visual trace of it:

```
_startVertex.x=ray.orig.x;
_startVertex.y=ray.orig.y
_startVertex.z=ray.orig.z
_lineSeg.start=_startVertex;
_endtVertex.x=ray.dir.x;
_endtVertex.y=ray.dir.y;
_endtVertex.z=ray.dir.z;
_lineSeg.end=_endtVertex;
```

Then we must normalize the ray's origin and direction vectors, otherwise the equation which performs the intersection calculation will return a huge positive value which can never reach a negative number and therefore you will always receive intersection indication, even if there is actually none. Last we call `ray.intersectBoundingRadius()`; to which we pass the target object position and its bounding radius. It's return type is Boolean, so that if the ray hits the volume of the bounding sphere, it returns true.

As is seen from the following screenshot, the ray cast from the red sphere is intersecting the textured sphere's bounding volume, causing it to glow with a red outline:



Ray Triangle test

The following intersection is going to be fun and actually it has quite a lot of practical use. We are going to test ray intersection with mesh triangles. Working on a shooter game, this is one of the handiest features you are desperately going to need in order to trace mesh hit points to apply bullet decay textures, or maybe add a damage to a mesh by removing triangles from its surface which received ray hit. In the following demo, I am going to show you how to do the second thing. We are going to strip the sphere primitive from its triangles by eliminating them when they receive a ray hit. Let's have some fun!

Copy the following method inside the class we have used previously:

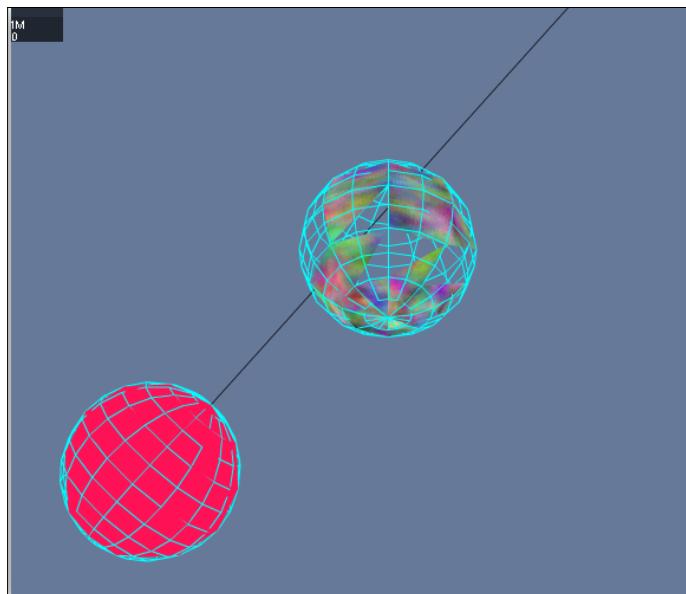
```
private function RayFaceTest () :Vector3D{  
    var intersectVector:Vector3D;  
    var ray:Ray=new Ray();  
  
    for(var i:int=0;i<_objB.faces.length;i++){  
        var p0:Vector3D=_objB.sceneTransform.transformVector(_objB.  
faces[i].vertices[0].position);  
        var p1:Vector3D=_objB.sceneTransform.transformVector(_objB.  
faces[i].vertices[1].position);  
        var p2:Vector3D=_objB.sceneTransform.transformVector(_objB.  
faces[i].vertices[2].position);  
  
        ray.orig=_objA.position;  
        var dd:Vector3D=new Vector3D(0,400,0);  
        var dird:Vector3D=_objA.transform.transformVector(dd);  
  
        ray.dir=dird;  
        _startVertex.x=ray.orig.x;  
        _startVertex.y=ray.orig.y  
        _startVertex.z=ray.orig.z  
        _lineSeg.start=_startVertex;  
        _endtVertex.x=ray.dir.x;  
        _endtVertex.y=ray.dir.y;  
        _endtVertex.z=ray.dir.z;  
        _lineSeg.end=_endtVertex;  
        intersectVector=ray.getIntersect(ray.orig,ray.dir,p0,p1,p2);  
  
        if(intersectVector){  
            trace(intersectVector);  
            var fc:Face=_objB.faces[i];  
            _objB.removeFace(fc);  
            break;  
        }  
    }  
}
```

```
        }
        return intersectVector;
    }
```

Inside `onEnterFrame()`, comment out the `if...else` condition block and add a single call to the function we have just inserted. Add the following line to rotate the target sphere:

```
_objB.rotate(Vector3D.X_AXIS, 5);
```

Run the application. Move the ray onto the region of the static sphere. You should see that the triangles which are intersecting with the ray are being deleted. Move the ray so that it hits the target from different angles, and in a few moments, you have just the bounding sphere left. All the triangles are gone!



Now it is time to dig into the code behind this demo. Let's see how it works inside `RayFaceTest()`. In order to find the intersection point coordinates of the ray on a particular triangle, you have to pass, besides the ray origin and direction, three `vector3D` values which represent three vertices of each triangle in a mesh into the `ray.getIntersect()` method. The common mistake many inexperienced developers make in this step is to forget converting vertex coordinates from the object's local space to the world (scene) space. This step is critical as the ray is cast in the scene space and not in the objects. The following block transforms the vertices from object's local to scene space:

```
for(var i:int=0;i<_objB.faces.length;i++) {
    var p0:Vector3D=_objB.sceneTransform.transformVector(_objB.
faces[i].vertices[0].position);
```

```
var p1:Vector3D=_objB.sceneTransform.transformVector(_objB.  
faces[i].vertices[1].position);  
var p2:Vector3D=_objB.sceneTransform.transformVector(_objB.  
faces[i].vertices[2].position);
```

Notice that this block, as well as the rest of the function, is wrapped with a `for` loop. We need to iterate through each triangle of the mesh till we detect the ray intersected triangle.

Now when we have the vertices coordinates transformed, we can start to define a ray's origin and direction vectors. This step is similar to the previous example on the Ray-AABS test:

```
ray.orig=_objA.position;  
var dd:Vector3D=new Vector3D(0,400,0);  
var dird:Vector3D=_objA.transform.transformVector(dd);  
  
ray.dir=dird;  
_startVertex.x=ray.orig.x;  
_startVertex.y=ray.orig.y  
_startVertex.z=ray.orig.z  
_lineSeg.start=_startVertex;  
_endtVertex.x=ray.dir.x;  
_endtVertex.y=ray.dir.y;  
_endtVertex.z=ray.dir.z;  
_lineSeg.end=_endtVertex;
```

Next we cast the ray with the origin on the `_objA` sphere and direction defined towards the sphere's local positive Y (UP). The three other arguments are three vertex position vectors of the currently tested triangle:

```
intersectVector=ray.getIntersect(ray.orig,ray.dir,p0,p1,p2);
```

If the ray hits the triangle, `intersectVector` returns the intersection world coordinates, otherwise it returns null. This way, we can detect which triangle got hit and further isolate it for our needs. The next block does just that:

```
if(intersectVector){  
var fc:Face=_objB.faces[i];  
_objB.removeFace(fc);  
break;  
}
```

The preceding condition block works like this.

If the currently tested triangle is hit, then `intersectVector` inside `if()` becomes true. In such a case, we get the reference to that very triangle and remove it from the mesh.

Afterwards, we exit the wrapping `for` loop immediately, because inside the current iteration, there is no reason to continue the test as the intersecting triangle has already been detected and we want to pass the return value for the function which is the intersection `Vector3D`.

Then we start over the loop routine and iterate through the array of remaining faces again.



If you want to boost the frame rate of the previous examples, comment out these lines inside the `initGeometry()` method:



```
// _objA.debugbs=true;  
// _objB.debugbs=true;
```

Redrawing bounding spheres are memory intensive.

See also

If you are fine now with these basics and want to broaden your knowledge with more advanced collision-detection material, you should read the following book:

Real-Time Collision Detection, by Christer Ericson

Creating cool effects with animated normal maps

As you already know, normal maps allow us to fake a high detail appearance on actually low poly geometry. Furthermore, in correlation with some unique shaders such as Fresnel (`FresnelPBMaterial`), which we have already discussed in *Chapter 1*, normal maps can yield an effect of water surface waves by distorting the primary texture and producing reflections based on an environment map. For this, Away3D has got a utility called `WaterMap`, which extends bitmap data and animates a tiled composed texture by drawing each tile in a sequence. This utility comes in very handy if we want to animate the waves, but its functionality may be also extended to other purposes as you will see shortly.

In this recipe, you will learn how to create a nice effect of sphere surface deformation in combination with Fresnel material-based water surface simulation.

Getting ready

First you should create an image consisting of a tiled normal maps animation sequence. The more tiles you have, the smoother the animation of the bitmap will be. So the natural question is how to create a sequence of animated normal maps. Well, the easiest way I found is to use a 3D package such as 3DsMax. Here are the steps you should follow in 3DsMax:

1. Create a plane primitive of even dimensions.
2. Select from the modifiers stack **Noise Modifier** and apply it to the plane. In the modifier dialog, check the **Fractal** checkbox. Set the strength factor for the z-axis to extrude waves. Check **Animate Noise** so that the animation of the mesh is going to be generated automatically in the 3Ds Max timeline. Now, basically the animation span is 100 frames long. We need only 25 animated images. So in the timeline, click the right-most key frame and drag it to frame 25.
3. In the top menu, click **Rendering** then **Render Setup**. In the **Common Parameters** tab, make sure you select **Range** from **-0 to 25** in the **Time Output** dialog. Scroll down, open the **Assign Renderer** dialog, and make sure **Default Scanline Renderer** is selected.
4. Now we are going to render the deformed plane mesh to normal map by using the **render to texture** option. First, create a new plane of similar dimensions as the deformed, position it at the same coordinates, and move it along the z-axis above the deformed plane so that even the top-most wave of the deformed plane don't protrude through its surface. The reason for this is that 3DsMax projects the surface of the deformed plane on the plane above it and any faces that are not on the same side of the projection plane would not render to the map.
5. Select the top plane. Click **Rendering** in the top menu and select the **Render to Texture** option. In the dialog that has opened, first specify in the **Output** category the path to save the map. Now, below, locate the **Projection Mapping** category and select the **Enabled** checkbox. To the right of the projection drop-down list, click the **Pick** button and select the deformed plane from the opened window.
6. Scroll down till you locate the **Output** category. Here we have to add a type of texture we want to render. Click the **Add** button and from the **Available elements** window, select **NormalsMap**. Click **AddElements** to close the dialog.
7. Back inside the Output area define a filename and format (select **.png**) with the same directory path you set previously for the output path. For the map size, click the **128x128** button.
8. Click the **Render** button at the bottom of **Render to the Texture** window. The program should process 25 images of normal maps into the folder specified by you.

9. The next step is a composition of a final tiled image inside Adobe Photoshop. Unfortunately, this is a manual task. But don't be afraid, as it will not take you too much time to accomplish. In Photoshop, create a new file with the dimension of 640x640 (5 rows and 5 columns of 128x128 normal tiles). Now insert the rendered 25 images into Photoshop and arrange them from left to right in five rows in a sequential order of their animation. Export the file to .jpg or png.
10. You can find a ready example of the file inside the assets/images folder named waterNormalMap.png. We will use it in the following demo.
11. Now when we are done with the normal map, there are two more images we need to get. First is an environment map. The best source for environment maps are HDRI images which may be found for royalty free usage on the web. If you don't find one just get one from the assets/images folder called envHDRI.jpg and copy it into your project assets folder.
12. The last image we need is for height map in order to animate the surface deformation of a sphere. We want to synchronize the surface texture distortion by normals animation with the deformation. Therefore, we need to create a height map from the normal map we have previously created. The fastest way is to use the Prefab "Generate Terrain" feature where you can generate a height map. Refer to Chapter 8, *Prefab3D*, to learn how to do it. For now, copy a ready height map named readermapRed.jpg from the same location you got the previous maps into your project assets folder.
13. Now we are fully equipped for the coding part of the recipe. Create a new class and call it NormalAnimDemo and make sure it extends AwayTemplate. Copy the following code into your file.

How to do it...

NormalAnimDemo.as

```
package
{
    [SWF(backgroundColor="#677999", frameRate="15",
    quality="LOW", width="800", height="600")]
    public class NormalAnimDemo extends AwayTemplate
    {

        [Embed(source="../assets/images/waterNormalMap.png")]
        private var WaterNormalMap : Class;
        [Embed(source="../assets/images/envHDRI.jpg")]
        private const EnvMap : Class;
        [Embed(source="../assets/images/readermapRed.jpg")]
        private const BumpMap : Class;
```

```
private const CAM_DIST : Number = 500;
private var _waterNormalMap : WaterMap;
private var _sphere:Sphere;
private var _fresMat:FresnelPBMaterial;
private var _extrMod:HeightMapModifier;
private var _bumpAnim:WaterMap;
public function NormalAnimDemo()
{
    super();

}

private function initPBMaterials():void{
    var objTexture:BitmapData=new BitmapData(512,512);
    objTexture.perlinNoise(65,65,5,23235,true,true,7);
    var normBTM:BitmapData=Cast.bitmap(new WaterNormalMap());
    var envBTM:BitmapData= Cast.bitmap(new EnvMap());
    _waterNormalMap = new WaterMap(objTexture.width, objTexture.height, 128, 128,normBTM );
    _bumpAnim = new WaterMap(objTexture.width, objTexture.height, 128, 128,Cast.bitmap(new BumpMap()) );
    _fresMat=new FresnelPBMaterial(objTexture, _waterNormalMap,
envBTM, _sphere,{ envMapAlpha: 0.4, refractionStrength: 69 });
    _fresMat.color=0xFF0500;
}

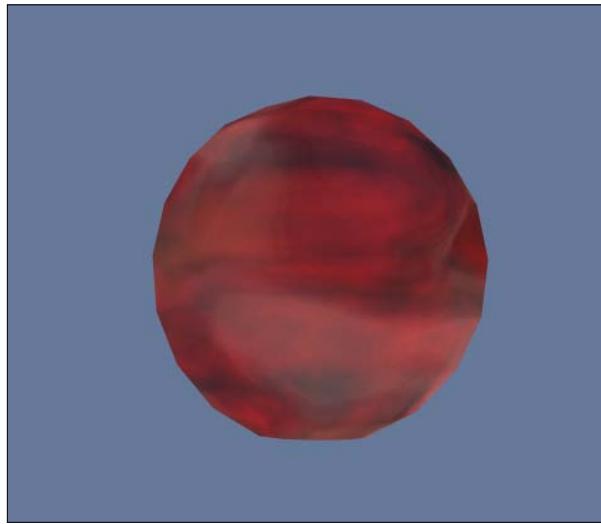
override protected function initGeometry() : void{
    _sphere=new Sphere({radius:150,segmentsW:8,segmentsH:8});
    initPBMaterials();
    _sphere.material=_fresMat;
    _view.scene.addChild(_sphere);
    _extrMod=new HeightMapModifier(_sphere,_bumpAnim,HeightMapDataChannel.RED,255,0.2,10);
}

override protected function onEnterFrame(e:Event) : void
{
    super.onEnterFrame(e);

    _view.camera.x = Math.sin(-_view.mouseX / 400)*CAM_DIST;
    _view.camera.y = Math.sin( _view.mouseY / 120)*CAM_DIST;
    _view.camera.z = -(Math.cos(-_view.mouseX / 400)+Math.cos( _view.mouseY / 120))*CAM_DIST;
    _view.camera.lookAt(new Vector3D(0,0,0));
    _extrMod.execute();
```

```
    _bumpAnim.showNext();
    _waterNormalMap.showNext();
}
}
}
```

The sphere on the screenshot is only a humble example of what you can do with the `WaterMap` class:



How it works...

First we create an instance of `FresnelPBMaterial` inside the `initPBMaterials()` method. You should pass to its constructor a reference to a bitmap data of the texture, which is, in this case, `objTexture` with generated perlin noise:

```
var objTexture:BitmapData=new BitmapData(512,512);
objTexture.perlinNoise(65,65,5,23235,true,true,7);
```

Next we set up instances for normal and environment maps:

```
var normBTM:BitmapData=Cast.bitmap(new
WaterNormalMap());
var envBTM:BitmapData= Cast.bitmap(new EnvMap());
```

Now we can create a `WaterMap` instance which accepts the defined `objTexture` into constructor arguments. We also pass the dimensions of the base texture as well as dimensions of a single tile in the normal map, which is, as you already know, 128x128 pixels. The last parameter is the normal map itself:

```
_waterNormalMap = new WaterMap(objTexture.width, objTexture.height,  
128, 128, normBTM );
```

We create an additional instance of the `WaterMap` class, but this one is to animate a height map which we pass into it:

```
_bumpAnim = new WaterMap(objTexture.width, objTexture.height, 128,  
128, Cast.bitmap(new BumpMap()) );
```

Finally, we set up `FresnelPBMaterial` passing into it a basic map, a normal map which is wrapped with the `WaterMap` instance, an environment map, and a target object which is a sphere primitive:

```
_fresMat=new FresnelPBMaterial(objTexture, _waterNormalMap, envBTM,  
_sphere, { envMapAlpha: 0.4, refractionStrength: 69 } );
```

Now let's check out the `initGeometry()` method. Here, besides setting the sphere, we call the `initPBMaterials()` function and the reason for it is that we can't instantiate the `PixelBender` material before the target geometry, as it requires a reference to not null instance of the last.

Next, we setup a `HeightMapModifier` which deforms a mesh surface based on height map input. We pass into its constructor `_bumAnim`, which is an instance of `WaterMap` and as you already guessed, is going to animate the height map:

```
_extrMod=new HeightMapModifier(_sphere,_bumpAnim,HeightMapDataChannel.  
RED,255,0.2,10);
```

In the preceding setup, it is important to define a correct map data channel. In my case, it has to be red, as the height map that we generated in Prefab is based on the red channel.

Now it's showtime. Inside `onEnterFrame()`, we iterate through the tiles in `_bumpAnim` and `_waterNormalMap`, updating by this the animation of respective textures at the surface of the sphere:

```
_bumpAnim.showNext();  
_waterNormalMap.showNext();
```

We also call on each frame `_extrMod.execute()` a method which updates the height map extrusion of the `HeightMapModifier` instance which has got an animated height map as its input and therefore changes on every frame as well.

See also

Chapter 3, Animating the 3D World, Morphing objects recipe.

Creating skyboxes and their textures

One of the essential parts of an outdoor 3D scene is a sky. The effect of surrounding sky in a 3D world is achieved by creating skybox or skydome. In Away3D, we will use a skybox. The skybox itself is just a cube with its faces inverted. The part that is responsible for the sky effects is the texture itself. Each side of the cube has to receive a texture that must be seamlessly matching the edges of all the other four textures adjacent to it. The process of creating such a set is not a straightforward one and there are special techniques as well as software that can help you to get it done relatively easy. This recipe is divided into two parts: first you will see how to create skybox texture set using Terragen and Photoshop and in the second part we

will learn how to set up skyboxes in Away3D.

Getting ready

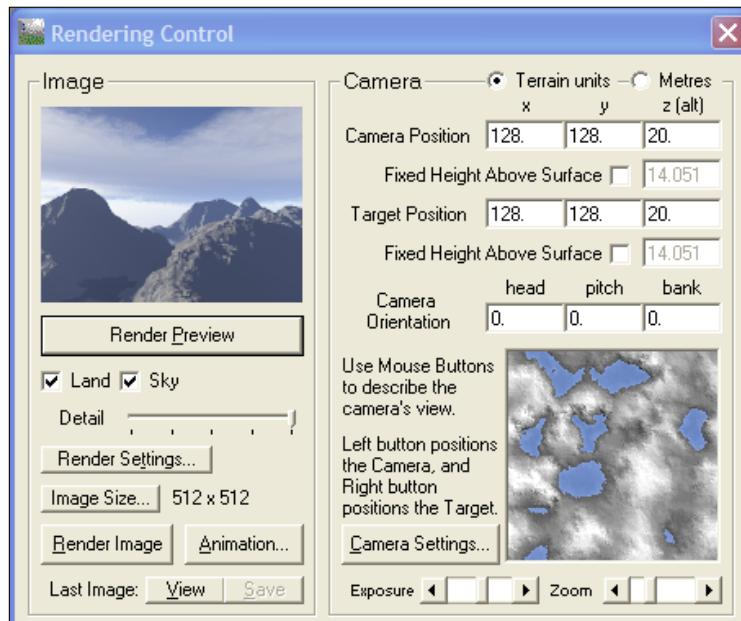
You need to download a version of Terragen. Go to <http://www.planetside.co.uk/content/view/16/28/> where you can download the trial version of Terragen Classic (be careful not to download Terragen 2 as its pipeline is different from the one described).

We also need a plugin called TerraSky that is going to help us in exporting the textures for the skybox. Get TerraSky plugin here for free <http://www.planetpointy.com/editing/files/terrasky.shtml>.

Install both programs. Read the manual for TerraSky carefully in order to set it up with Terragen. Now let's adjust the settings for the nice-looking sky in Terragen.

How to do it...

Open Terragen. In the Water dialog, set the Water Level to Zero. In the **Cloudscape** dialog, set sky size to 16384. In the **Rendering Control** dialog box, set the values as shown in the following screenshot:



Click the **Render Settings** button and in the opened dialog, drag both Accuracy sliders to the maximum to get the highest image quality.

If you wish to render the Atmosphere, only then unselect the **Land** checkbox in Rendering Control.

Now let's open the TerraSky Script. In the Terragen top menubar, click **Terragen | Execute Script**. Navigate to the directory where you put the file named `sky.tgs`. I suggest you put it in the Terragen root folder so that you will not need to look for it each time in a different place. Before we run the script, let's solve one more issue. If we now run the script when the terrain is set to be rendered, we would find that the camera view is intersected with the terrain mesh. The problem is that TerraSky script, by default, has the camera elevation defined too low. We need to define higher camera altitude inside the script in order to produce the image without side effects. Don't be afraid, the script is very simple to understand. Open the `sky.tgs` file with a notepad. It should look like this:

```
initanim, "C:\Program Files (x86)\TerraSky\Input\sky", 1  
;framenum, 1  
zoom 1
```

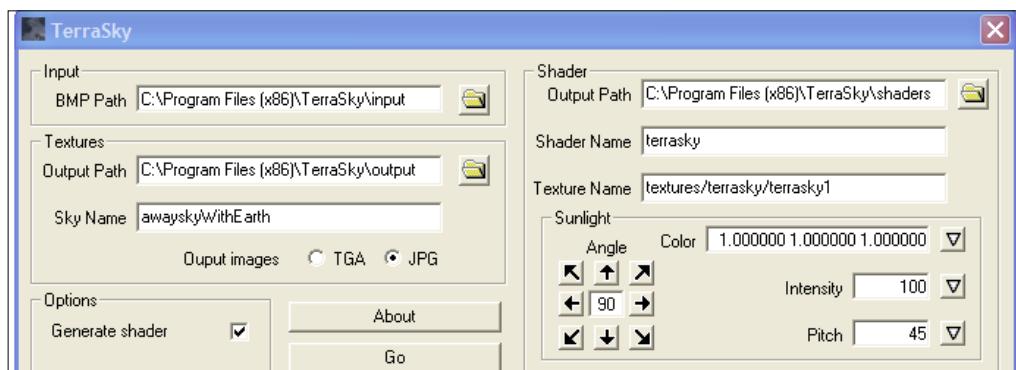
```

campos 128,128,20
tarpos 128,256,20
frend
;framenum,2
campos 128,128,20
tarpos 256,128,20
frend
;framenum,3
campos 128,128,20
tarpos 128,0,20
frend
;framenum,4
campos 128,128,20
tarpos 0,128,20
frend
;framenum,5
campos 128,128,20
tarpos 128,128,128
frend
;framenum,6
campos 128,128,20
tarpos 128,128,-128
frend

```

Highlighted numbers are those you need to tweak. Be aware that the units are Terragen terrain units and not meters. So don't set large numbers as you will find the camera outside the environment bounds. It is important to notice that in each render block of the script, you should define the same number for `campos` and for `tarpos`, which stand for camera position and target position respectively. In the last two blocks, you play only with the `campos` because the direction vectors should be upwards and downwards for top and bottom image renderings. Feel free to experiment with the settings. When you have finished, select the script and click **Open**. The rendering sequence should begin.

After you are done, open TerraSky:



In the program interface, make sure the input folder is the folder where you rendered the Terragen sky images. Set the names for the sky images. Leave the rest as it is and click the **Go** button.

Your skybox images are ready for Away3D!

Creating a SkyBox6-compatible texture

There is another type of skybox in Away3D that is called **SkyBox6**. It is different from the regular skybox as it requires a single image to map all six faces. This can save us a lot of file weight if we load only one bitmap instead of six. The screenshot should look like this:



Basically, you should grab the skybox set we created in Terragen and create a composition with the same structure in a program such as Photoshop, as shown in the screenshot. Be patient, as most likely you will not get the correct result from the first attempt. The skybox UV mapping slightly differs between Terragen skybox and Away3D Skybox6 classes. That means that you will probably need to go back and forth between graphics editing programs several times to rotate some of the compound images in order to match them seamlessly to their neighbors.

[ There are many more tools and techniques out there to create skybox textures. If you have built 3D environments in 3DsMax, you can render them to skybox textures using Max's Cube Map rendering feature of Reflection/ Refraction Maps. This approach is very useful to create unique environment maps and is perfect for skybox texture sets rendering.]

Skybox setup in Away3D

Now let's take the skybox texture we created previously and test it in our Away3D scene. You are going to create two classes. One that renders a regular skybox that uses six separate skybox textures. Another class will demonstrate a setup of Skybox6.

Create two new classes and name them `SkyBoxDemo` and `SkyBox6Demo`. Both should extend the `AwayTemplate` class. Copy the following code into these files according to their names:

Skybox:

`SkyBoxDemo.as`

```
package
{
    public class SkyBoxDemo extends AwayTemplate
    {
        [Embed(source="assets/skyBoxSet/awayskyWithEarth_bk.jpg")]
        private var BackSky:Class;
        [Embed(source="assets/skyBoxSet/awayskyWithEarth_dn.jpg")]
        private var BottomSky:Class;
        [Embed(source="assets/skyBoxSet/awayskyWithEarth_ft.jpg")]
        private var FrontSky:Class;
        [Embed(source="assets/skyBoxSet/awayskyWithEarth_lf.jpg")]
        private var LeftSky:Class;
        [Embed(source="assets/skyBoxSet/awayskyWithEarth_rt.jpg")]
        private var RightSky:Class;
        [Embed(source="assets/skyBoxSet/awayskyWithEarth_up.jpg")]
        private var TopSky:Class;

        private var _skyBox:Skybox;
        private var _frontM:BitmapMaterial;
        private var _backM:BitmapMaterial;
        private var _topM:BitmapMaterial;
        private var _bottomM:BitmapMaterial;
        private var _leftM:BitmapMaterial;
        private var _rightM:BitmapMaterial;
        private var _oldMsX:Number=0;
        private var _oldMsY:Number=0;
        private var _howCam:HoverCamera3D;
        private var _lastPanAngle:Number;
        private var _lastTiltAngle:Number;
```

```
private var _canMove:Boolean=false;
public function SkyBoxDemo()
{
    super();
    initHoverCam()

}

override protected function initMaterials() : void{
    _frontM=new BitmapMaterial(Bitmap(new BackSky()).bitmapData);
    _leftM=new BitmapMaterial(Bitmap(new RightSky()).bitmapData);
    _backM=new BitmapMaterial(Bitmap(new FrontSky()).bitmapData);
    _rightM=new BitmapMaterial(Bitmap(new LeftSky()).bitmapData);
    _topM=new BitmapMaterial(Bitmap(new TopSky()).bitmapData);
    _bottomM=new BitmapMaterial(Bitmap(new BottomSky()).bitmapData);
}
override protected function initGeometry() : void{
    _skyBox=new Skybox(_frontM,_rightM,_backM,_leftM,_topM,
_bottomM);
    _view.scene.addChild(_skyBox);
    stage.addEventListener(MouseEvent.MOUSE_DOWN,
onMouseDown,false,0,true);
    stage.addEventListener(MouseEvent.MOUSE_UP,
onMouseUp,false,0,true);
}
private function initHoverCam():void{
    _howCam=new HoverCamera3D();
    _view.camera=_howCam;
    _howCam.focus = 40;
}
private function onMouseDown(e:MouseEvent):void
{
    _lastPanAngle = _howCam.panAngle;
    _lastTiltAngle = _howCam.tiltAngle;
    _oldMsX = stage.mouseX;
    _oldMsY = stage.mouseY;
    _canMove = true;
}
private function onMouseUp(e:MouseEvent):void
{
    _canMove = false;
}
override protected function onEnterFrame(e:Event):void{
    super.onEnterFrame(e);
```

```
        if (_canMove) {
            _howCam.panAngle = 0.3 * (stage.mouseX - _oldMsX) + _lastPanAngle;
            _howCam.tiltAngle = 0.3 * (stage.mouseY - _oldMsY) + _lastTiltAngle;
        }
        _howCam.hover();
    }
}
```

Skybox6:

SkyBox6Demo.as

```
package
{
    public class SkyBox6Demo extends AwayTemplate
    {
        [Embed(source="assets/skyBoxSet/SkyBox6TextureReady.jpg")]
        private var SkyBoxTxt:Class;
        private var _skyBoxMat:BitmapMaterial;
        private var _skyBox:Skybox6;
        private var _oldMsX:Number=0;
        private var _oldMsY:Number=0;
        private var _howCam:HoverCamera3D;
        private var _lastPanAngle:Number;
        private var _lastTiltAngle:Number;
        private var _canMove:Boolean=false;
        public function SkyBox6Demo()
        {
            super();
            initHoverCam();
        }
        override protected function initMaterials() : void{
            _skyBoxMat=new BitmapMaterial(Cast.bitmap(new SkyBoxTxt()));
            _skyBoxMat.smooth=true;
        }
        override protected function initGeometry() : void{
            _skyBox=new Skybox6(_skyBoxMat);
            _view.scene.addChild(_skyBox);
        }
    }
}
```

```
        stage.addEventListener(MouseEvent.MOUSE_DOWN,
onMouseDown,false,0,true);
        stage.addEventListener(MouseEvent.MOUSE_UP,
onMouseUp,false,0,true);

    }
private function initHoverCam():void{
    _howCam=new HoverCamera3D();
    _view.camera=_howCam;
    _howCam.focus = 40;
}
private function onMouseDown(e:MouseEvent):void
{
    _lastPanAngle = _howCam.panAngle;
    _lastTiltAngle = _howCam.tiltAngle;
    _oldMsX = stage.mouseX;
    _oldMsY = stage.mouseY;
    _canMove = true;
}

private function onMouseUp(event:MouseEvent):void
{
    _canMove = false;
}
override protected function onEnterFrame(e:Event):void{
    super.onEnterFrame(e);
    if (_canMove) {
        _howCam.panAngle = 0.3 * (stage.mouseX - _oldMsX) + _lastPanAngle;
        _howCam.tiltAngle = 0.3 * (stage.mouseY - _oldMsY) + _lastTiltAngle;
    }
    _howCam.hover();
}
}
```

How it works...

As you can see, the setup for both demos is almost identical. The only difference is in the skyboxes. In the `SkyBoxDemo` program, we work with the `Skybox()` class which we instantiate as follows inside the `initGeometry()` method:

```
_skyBox=new Skybox(_frontM,_rightM,_backM,_leftM,_topM,_bottomM);
```

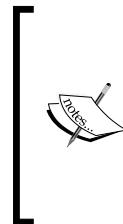
The constructor requires six bitmap materials for each size of the skybox cube, which we instantiate within the `initMaterials()` function.

Now look at `SkyBox6Demo`. The instantiation of `Skybox6` is much shorter, as there is no need to create six different materials for each side of the cube. Here, we create only one, which has its source and the texture that contains all six parts, as we have seen previously, and pass it into the `Skybox6` constructor:

```
_skyBox=new Skybox6 (_skyBoxMat) ;
```

Running heavy scenes faster using BSP trees

Until recently, one could only dream of creating a complex indoor environment in Away3D. To set up a scene with a system of several rooms was an extremely difficult task in terms of performance and although some companies quite succeeded in creating a relatively complex environment, they were still highly restricted in the amount of geometry they could afford as well as the optimization work consumed a bulk of the total development time. Eventually, the Away3D team came up with the solution known as BSP trees—a term widely used in industrial real-time game engines. Without entering into details on how those trees work, you should know that they significantly increase performance of quite heavy scenes by managing, culling, and sorting of the geometry in binary space partition trees. The BSP technique implies that the world geometry is sorted into tree nodes having children nodes and so on. The tree is generated during compile time, cancelling out the need to sort all the objects in runtime. Moreover, the system implements the **PVS (potentially visible sets)** technique to calculate, in advance, which objects are occluded and therefore can be removed from the rendering during runtime. All these features allow you, the developer, to significantly lower the restrictions imposed on you earlier when creating big and geometry-stuffed indoor scenes.



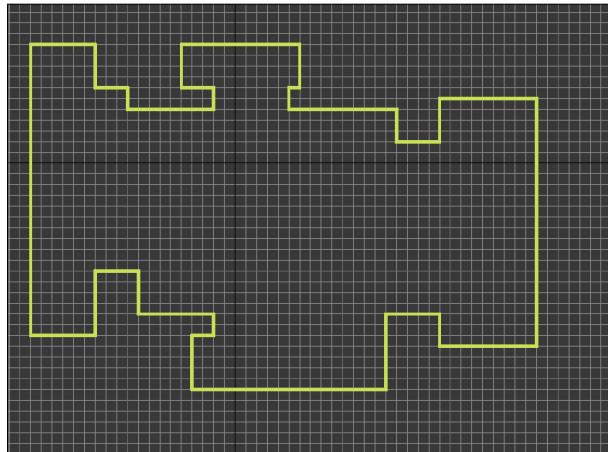
As you probably have noticed, we pointed out in several places that we talk about constructing indoor scenes. The thing is that BSP in Away3D can only process completely sealed geometry, which means that no gaps or openings are allowed on the geometry surface. That still doesn't mean that you can't create outdoor scenes, it only means that it would be a tricky task. But how to do this is outside the scope of this book.

So enough theory, let's build some BSP!

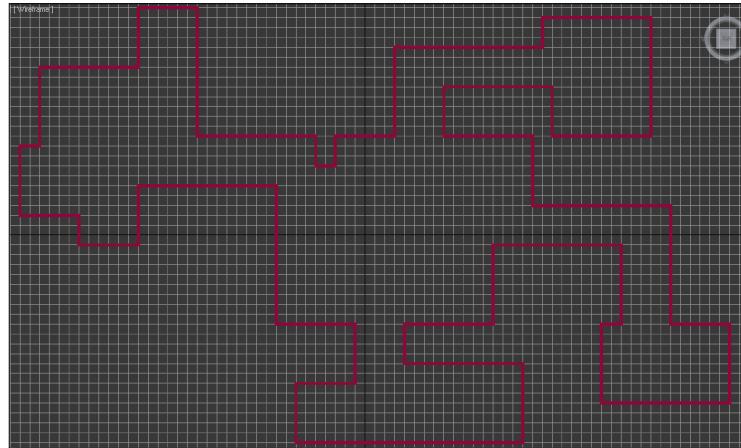
Getting ready

1. First we need to create a 3D model of our indoor scene. For this task we used 3DsMax to create a simple labyrinth system in just 30 minutes. Of course, working on something serious may take you several weeks. It is impossible to describe here in detail how this demo model was created, but here are a few important rules you must follow in order to create BSP-tree compatible meshes:
 - The overall model must be completely closed with no breaches into the outside world. Otherwise, the PVS will get stuck. Therefore, inspect your model carefully so that the mesh has no open regions such as missing polygons.
 - Try to keep your geometry axis aligned as it speeds up the BSP calculations.
 - Set grid snapping in the modeling program. It will ensure proper BSP splitting of the geometry.
 - If you need multiple materials to be applied (Away3D doesn't support them), split the structure into separate objects and map them with different materials. Make sure, in this case, you implement the first rule.
 - Try to design your indoor environment in a way that from any viewpoint there is not much geometry visible. Try to create less big hall-like rooms which connect with other rooms via big and open passages. Instead, plan your building in such a way that the different parts of it are isolated via long and relatively narrow passages. Ignoring this rule may lead to a slow rendering and PVS build. Here is an example:

Bad design:



Desirable design:



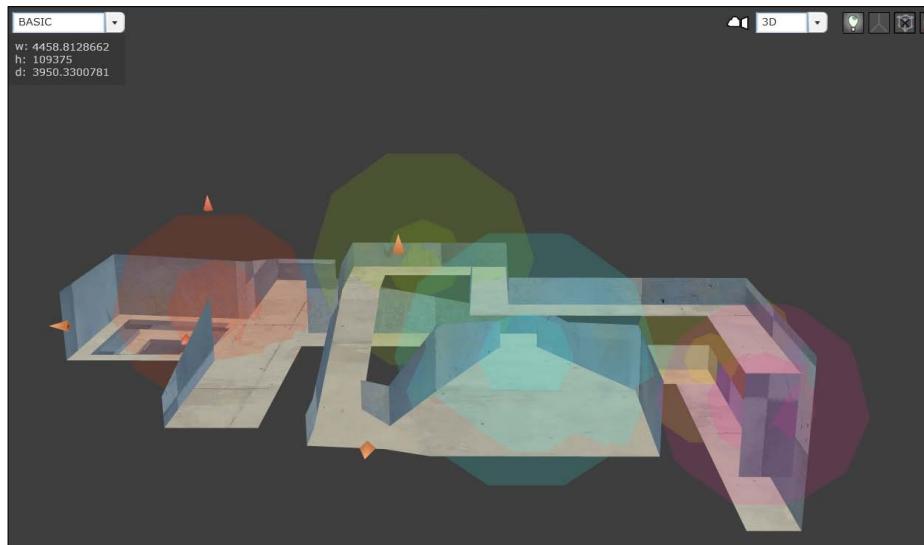
- Consider adding small objects that are not part of the building base architecture later (during runtime) with `addChild()`. This keeps your tree clean and more efficient.
- 2. Although it is possible to do the entire process with manual coding, we all know that time is an important factor for us developers. Therefore, why should we go the hard way if we have the wonderful program Prefab. Prefab allows you to generate a BSP tree and export a ready-to-run scene with zero lines of code from your side. If you still have not installed Prefab, this is the time to do it. Get it from www.closier.nl/prefab/.

How to do it...

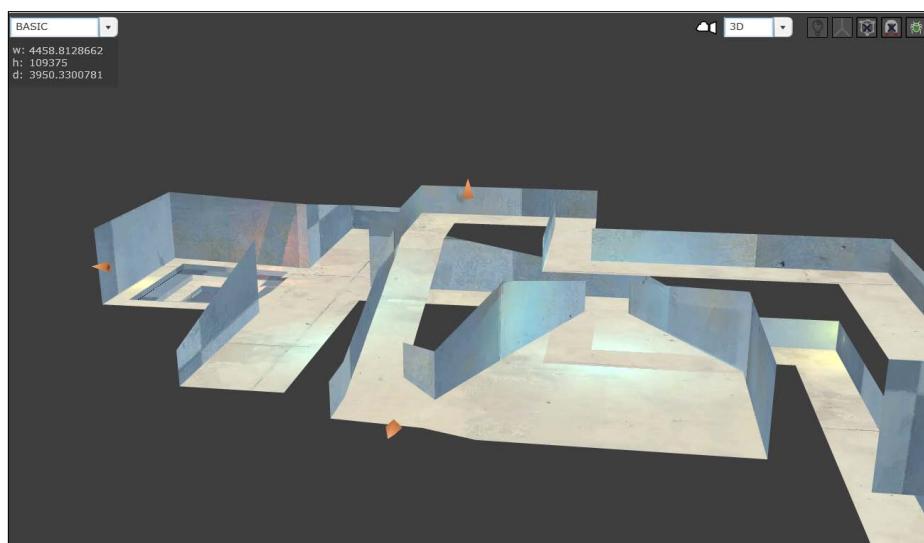
Prefab part:

1. Open Prefab. Import a 3ds Max-created scene from this chapter's assets/bsp/ folder called `BSPTunnel.3ds`. The import has got a rotation x-axis offset of 90 -90 degrees. In the Properties window, set X rotation to 90 and click the **Apply rotations** button. We also scale the model to have its inner space larger. Set scale X, Y, and Z to 2 and click **Apply scale**. Reset the model's position as well by setting its X, Y, and Z values to zero and click **Apply positions**.

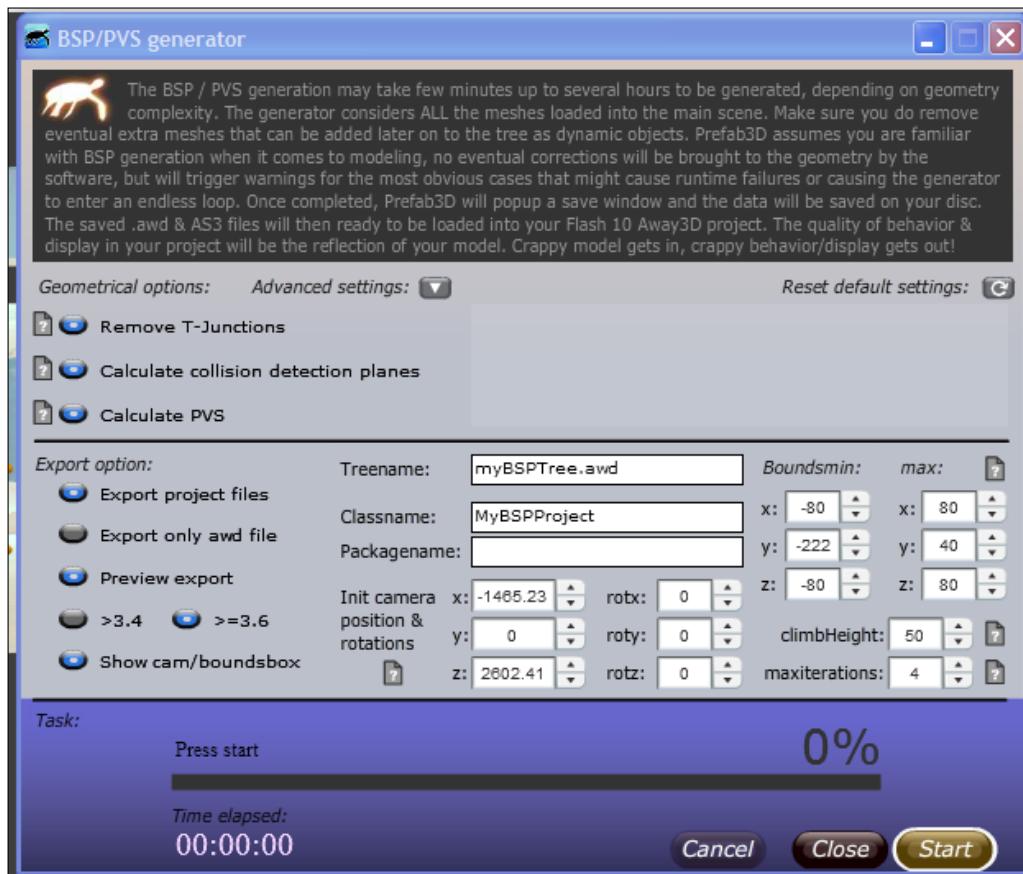
2. Assign to the model a texture found in the `assets/bsp/images` folder named `Box03.jpg`. The map has already light baked into it by me using Prefab's ray tracing option. Here is the screenshot of the lights setup before the baking:



3. In case you want to bake your own map, first set up the lights. Then select rendering mode (ray tracing is most precise) in the bottom menu and click the **Render** button. After the rendering is done, save the texture from the "Rendered" slot in the right panel to file and put it into your project assets folder. For this example, the texture is `Box03.jpg` and it is already baked:

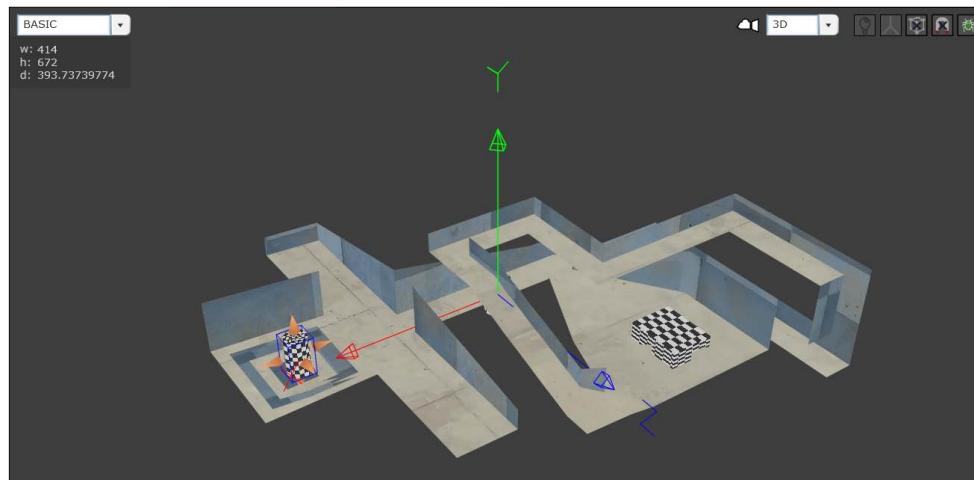


4. Before we start BSP generation, it is good to optimize the model by deleting double faces and removing unnecessary vertices and UVs. In the top menu, go to **Face** and click **Remove double faces**. Then click the **Geometry** menu and select **Weld (selection)**.
5. We have to invert faces because the player is going to be inside the model. In the top menu, click **Faces** then **Invert Faces (selection)**.
6. Open the BSP generation window by clicking **Geometry**, and selecting **BSP/PVS generator**. Here is a setup for our model:



7. The most important settings for the BSP generator are explained as follows:
 - Calculate PVS—calculate potentially visible sets which will speed up occluded geometry culling in the scene during runtime.
 - Preview export—after the generation is complete, Away3D creates an interactive Flash movie with your scene, enabling you to move inside it in first person (set to true).

- ❑ Init camera position and rotation—this one is important as it enables you to define the initial position of the camera inside the tree. If you look into the Prefab viewport while the BSP generator window is opened, you can see a small camera helper there. Tweak the X, Y, and Z settings while you visually define the position of the camera. Make sure it is not outside the geometry.
- ❑ Bounds min/max—the camera is wrapped with a collider object which is simply AABB and tests its collisions against the geometry. These settings allow you to define the bounding box of the collider. If you wish for the camera to be higher above the ground, set the Bounds min to a larger negative value. The set up you can see in the picture above is fine for the model we are using here.
- ❑ Now we are ready to generate the tree. Click the **Start** button inside the BSP/PVS generator window. After several seconds, you should get the window's explorer opened with the .awd file which you need to save. Right after you save, you should get another window with a fully interactive Away3D scene where you can run in first person inside the environment you have built. If it doesn't happen, it is likely that your model had some errors and the tree was not generated.
- ❑ We are not done yet. So you would also like to add geometry into your generated scene during runtime. There is no problem with it as BSP trees allow you to do this the same way as you add objects to a regular Away3D display object. But it may be extremely efficient to map the positions of the object you would add later already in Prefab. This way, you can save a lot of time of object position adjusting blindly. In this example, we create a cube and a cylinder inside the Prefab using Primitive Generator and position them as shown in the screenshot. Write down the coordinates of these objects as you are going to need them when you add those primitives inside your Away3D scene:



8. Ok. Now we are ready to move on to assemble our scene inside the Flash project. Go to the location where you have saved the tree from Prefab. You should see a folder called `myBSPTree`, which is generated for you and contains the tree as the `myBSPTree.awd` file, the `images` folder containing all the images that the model uses as its textures. In our case, it has only one. But most important is the third file, `myBSPTree.as`, which is ready to run the Away3D scene including FPS camera, which you can simply drop into your project and fire with not a single line of code needed from your side. Put this file into the `src.default` package of the Flex ActionScript folder. Create a folder called `bsp` inside the assets folder of the project and put the `images` folder and `myBSPTree.as` there. The following code is the content of `myBSPTree.as`, which I copied into a new file called `MyBSPProject.as`. The difference is that it has embedded object paths modified to the designation of the assets in my project. Also, as you can see, we add here two primitives to the scene positioning them at the locations we defined in Prefab:

```
package
{
    // Class generated by Prefab3D. David Lenaerts & Fabrice
    Closier, 2010.

    public class MyBSPProject extends Sprite
    {
        [Embed(source="../assets/bsp/myBSPTree.awd",
        mimeType="application/octet-stream")]
        private var BSPFile:Class;

        private static const MOVE_SPEED:Number = 90;
        private static const JUMP_STRENGTH:Number = 300;
        private static const ACCELERATION:Number = 40;
        private static const RUN_MULTIPLIER:Number = 2;
        private static const GRAVITY:Number = 30;
        private static const FRICTION:Number = .3;
        private var _lastMouseX:Number;
        private var _lastMouseY:Number;
        private var _lastRotX:Number;
        private var _lastRotY:Number;
        private var _rotX:Number;
        private var _rotY:Number;
        private var _right:Boolean;
        private var _left:Boolean;
        private var _rotaccel:Number = 0.0;
        private var _tree:BSPTree;
        private var _view:View3D;
        private var _xDir:Number = 0.0;
        private var _zDir:Number = 0.0;
```

```
private var _mouseDown:Boolean;
private var _dragX:Number = 0;
private var _dragY:Number = 0;
private var _bspCollider:BSPCollider;
private var _force:Vector3D = new Vector3D();
private var _flightMode:Boolean = false;
private var _speedMultiplier:Number = 1.0;
private var _noClip:Boolean = false;
private var bitMat:BitmapMaterial;
private var cube:Cube;

public function MyBSPProject()
{
    addEventListener(Event.ADDED_TO_STAGE, init);
}

private function init(e:Event):void
{
    initStage();
    initAway3D();
    var stats:AwayStats=new AwayStats(_view);
    this.addChild(stats);

}
private function initStage():void
{
    stage.addEventListener(KeyboardEvent.KEY_DOWN, KeyIsDown);
    stage.addEventListener(KeyboardEvent.KEY_UP, KeyIsUP);
    stage.addEventListener(MouseEvent.MOUSE_DOWN,
onMouseIsDown);
    stage.addEventListener(MouseEvent.MOUSE_UP, onMouseIsUp);
}

private function initAway3D():void
{
    _view = new View3D({x: stage.stageWidth*.5, y: stage.
stageHeight*.5, clipping: new FrustumClipping()});
    _view.camera.lens = new PerspectiveLens();
    _view.camera.zoom = 10;
    _view.camera.x = -1465.23;
    _view.camera.y = 0;
    _view.camera.z = 2602.41;
    _view.camera.rotationX = 0;
    _view.camera.rotationY = 0;
```

```
_view.camera.rotationZ = 0;
_tree = BSPTree(AWData.parse(new BSPFile(), {customPath:"../
assets/bsp/images/"}) );
_view.scene.addChild(_tree);
_tree.usePVS = true;

_bspCollider = new BSPCollider(_view.camera, _tree);
_bspCollider.testMethod = BSPTree.TEST_METHOD_ELLIPSOID;
_bspCollider.flyMode = _flightMode;
_bspCollider.maxClimbHeight = 50;
_bspCollider.maxIterations = 4;
_bspCollider.minBounds = new Vector3D(-80, -222, -80);
_bspCollider.maxBounds = new Vector3D(80, 40, 80);
addChild(_view);
addExternalGeometry();
addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function KeyIsDown(event:KeyboardEvent):void
{
    switch(event.keyCode) {
        case 90:
        case 87:
        case Keyboard.UP:_zDir = 1; break;
        case 83:
        case Keyboard.DOWN:_zDir = -1;break;
        case 81:
        case 65:
        case Keyboard.LEFT:_xDir = -1;break;
        case 68:
        case Keyboard.RIGHT:_xDir = 1;break;
        case Keyboard.SPACE:
            if(_bspCollider && _bspCollider.onSolidGround) _force.y
= JUMP_STRENGTH;
            break;
        case Keyboard.SHIFT:
            _speedMultiplier = RUN_MULTIPLIER;
            break;
    }
}

private function KeyIsUP(event:KeyboardEvent):void
{
    switch(event.keyCode) {
```

```
        case 90:
        case 87:
        case 83:
        case Keyboard.UP:
        case Keyboard.DOWN:_zDir = 0;break;
        case 65:
        case 81:
        case Keyboard.LEFT:
        case 68:
        case Keyboard.RIGHT:_xDir = 0;break;
        case Keyboard.ENTER:_tree.usePVS = !_tree.usePVS;break;
        case Keyboard.SHIFT:_speedMultiplier = 1;break;
        case Keyboard.F1:_noClip = !_noClip;break;
    }
}

private function onEnterFrame(event:Event):void
{
    var maxSpeed:Number = MOVE_SPEED*_speedMultiplier;
    if(_mouseDown){
        _rotX = _lastMouseX - (mouseY - _lastRotY)*0.7;
        _rotY = _lastMouseY + (mouseX - _lastRotX)*0.7;

        if(_rotX > 90) _rotX = 70;
        if(_rotX < -90) _rotX = -70;
        _view.camera.rotationX += (_rotX - _view.camera.
rotationX)/4;
        _view.camera.rotationY += (_rotY - _view.camera.
rotationY)/4;
    }

    if(_noClip || !_bspCollider) {
        _view.camera.moveForward(_zDir*maxSpeed);
    }
    else {
        var dx:Number = _xDir*ACCELERATION;
        var dz:Number = _zDir*ACCELERATION;

        _force.x += dx;
        _force.z += dz;
        if(_force.x > maxSpeed || _force.x < -maxSpeed) _force.x
-= dx;
        if(_force.z > maxSpeed || _force.z < -maxSpeed) _force.z
-= dz;
    }
}
```

```

        if(_flightMode) {
            _bspCollider.move(_xDir * maxSpeed, 0, _zDir *
maxSpeed);
        } else {
            _force.y = _bspCollider.move(_force.x, _force.y, -
_force.z).y;
            _force.x /= (1+FRICITION);
            _force.z /= (1+FRICITION);
            _force.y -= GRAVITY;
        }
    }
    _view.render();
    redrawBitMap();
}

private function redrawBitMap():void{
    if(bitMat.bitmap){
        bitMat.bitmap.lock();
        var seed:Number=Math.floor(Math.random()*999999);
        var p:Point=new Point(Math.floor(Math.random()*bitMat.
bitmap.width),Math.floor(Math.random()*bitMat.bitmap.height));
        bitMat.bitmap.pixelDissolve(bitMat.bitmap,bitMat.bitmap.
rect,p,seed,12,Math.floor(Math.random()*0xFFFFFFFF));
        for(var i:int=0;i<1000;++i){
            var p1:Point=new Point(Math.floor(Math.random()*bitMat.
bitmap.width),Math.floor(Math.random()*bitMat.bitmap.height));
            bitMat.bitmap.setPixel(p1.x,p1.y,Math.floor(Math.
random()*0xFFFFFFFF));
        }
        var bdata1:BitmapData=new BitmapData(64,64);
        bdata1.noise(seed,0,255,16);
        var rect:Rectangle=new Rectangle(0,0,Math.random()*bitMat.
bitmap.width,Math.random()*bitMat.bitmap.height);
        bitMat.bitmap.merge(bdata1,rect,p,Math.random()*255,Math.
random()*255,Math.random()*255,1);
        bdata1.dispose();
        bitMat.bitmap.unlock();
    }
}
private function onMouseIsDown(event:MouseEvent):void
{
    _lastMouseX = _view.camera.rotationX;
    _lastMouseY = _view.camera.rotationY;
    _lastRotX = mouseX;
}

```

```
    _lastRotY = mouseY;
    _mouseDown = true;
    _dragX = mouseX;
    _dragY = mouseY;
}

private function onMouseIsUp(event:MouseEvent):void
{
    _mouseDown = false;
}
private function addExternalGeometry():void{
    var bdata:BitmapData=new BitmapData(128,128);
    bdata.noise(225354,23,255,7);
    bitMat=new BitmapMaterial(bdata);
    cube=new Cube({material:bitMat,width:299,height:160,depth:299});
    cube.transform.position=new Vector3D(-1748.52,-300,2385.33);
    cube.name="cube1";
    cube.collider=true;
    _tree.addChild(cube);
    var cylc:Cylinder=new Cylinder({height:600,material:bitMat,radius:230});
    cylc.transform.position=new Vector3D(3416,-210,-677);
    cylc.name="cylind1";
    cylc.collider=true;
    _tree.addChild(cylc);
}
}
```

How it works...

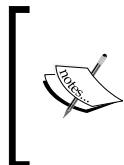
Don't be scared of the amount of code in the preceding class. Most of it deals with keyboard input control. We are going to concentrate only on the parts which are related to the BSP tree setup. So as you probably noticed, the class structure is different from the one we were working throughout the book and this is because it was generated by Prefab and the credits for the programming go to Fabrice Closier and David Lenaerts, as you can see at the class header. We start from the `initAway3D()` method where Away3D scene is set up. Right after it, we instantiate the BSP tree, which is parsed from `myBSPTree.awd` that we created in Prefab:

```
_tree = BSPTree(AWData.parse(new BSPFile(), {customPath:"../assets/bsp/images/"}) );
_view.scene.addChild(_tree);
_tree.usePVS = true;
```

The tree is added to the scene directly as it extends `ObjectContainer3D`. We also turn on `.usePVS` so that we can benefit from the usage of the pre-calculated occlusion culling date for the tree geometry.

The next lines create and assign a collider for the camera so that the user will be able to move in a first person in a realistic manner without going through the walls. The `minBounds` and `maxBounds` setting are derived from the setting we inserted in the BPS generator dialog and here you can customize them at will:

```
    _bspCollider = new BSPCollider(_view.camera,  
    _tree);  
    _bspCollider.testMethod = BSPTree.TEST_METHOD_ELLIPSOID;  
    _bspCollider.flyMode = _flightMode;  
    _bspCollider.maxClimbHeight = 50;  
    _bspCollider.maxIterations = 4;  
    _bspCollider.minBounds = new Vector3D(-80,-222,-80);  
    _bspCollider.maxBounds = new Vector3D(80,40,80);
```



Every 3D object in Away3D can be assigned a collider in the simplest way by setting `Object3D.collider` to true. While this approach is the shortest, it lacks all the functionality `BSPCollider` grants you. You will want to use `BSPCollider` mostly for the controlled-movable objects.

And this is all you need to know to set up a BSP tree scene. The rest of the code in this example deals with camera control and it is pretty straightforward, especially because you were already introduced to `FPSController` in the previous chapter. The only additional function you should look at is `addExternalGeometry()` and it was added by me. It adds two primitives to the scene setting them to the coordinates we mapped in Prefab beforehand. Also, we have `redrawBitmap()` which animates the bitmaps of the primitives' material using several AS3 bitmap data-generic methods in combination with many random values. Actually, the result is pretty cool!

Beginning Molehill API

In this appendix we will create a rotating sphere from scratch.

The following recipe is dedicated to the next generation Flash Player version 11 code named Molehill which contains a set of GPU-accelerated 3D APIs exposed via ActionScript3.0. The API allows us developers to create high-end 3D content that would be pretty close to what is done in today's video consoles and PC desktop games.

Practically speaking with the Molehill, you are able to render hundreds of thousands of triangles at each frame with not even the slightest performance drop!

Molehill API pipeline overview

First you may ask—how is GPU acceleration different from CPU-based graphics processing? The secret lies in the fact that modern GPUs contain hundreds of cores (GeForce GTX 280, for instance, contains 240 processing cores) which are capable of processing data in parallel speeding up the calculations hundreds of times faster than any multi-core CPU! Moreover modern graphic processing units perform operations of rasterization and depth sorting natively with incredible speed. Additionally the modern GPUs are programmable which means using shader programs; you can tell your graphics card how you want to process your content.

Having said this and before getting to the actual hands on with the API, you should know that the **Molehill** is an extremely low-level toolset. The communication with the GPU is performed via filling buffer objects with raw geometry data such as vertices, indices, colors, UV coordinates, and dispatching them to the graphics card for processing. In addition to it you have to supply the GPU with a shader program, containing the orders how to process that data. Adobe presents a brand new assembly-based low-level language called **Adobe Graphics Assembly Language (AGAL)**. AGAL is used to write the shader programs which tell the GPU how to process the vertices and render fragments based on the data we supply within the buffers. Without defining these programs, your GPU will not be able to know how to deal with the vertex and color data you have sent it.

AGAL is set of operation code (opcode) instructions which are interpreted at machine level. Writing complex shader programs with AGAL can be a highly challenging task even for experienced 3D programmers. Fortunately Adobe released **PixelBender3D**—a 3D version of PixelBender language which at the time of writing is available as a prerelease version from Adobe labs. If you are familiar with the 2D version of PixelBender you will be able to take advantage of the human-friendly API really fast and will be able to write shaders in a much less painful way than with raw AGAL commands. PixelBender3D is outside the scope of this book.

You should remember that Molehill API is intended primarily to serve as a toolkit for 3D engine development. Although it is completely possible to write fully featured 3D applications using the API directly, it would cost you a tremendous amount of time and effort. You are highly encouraged to use Molehill-based 3D engines such as Away3D, Flare3D, Alternativa3D, and others to develop the state-of-the-art 3D content, rapidly getting the maximum from API.

Creating a rotating sphere from scratch

Now let's get to the business. In the following pages you will get acquainted with the core terms and concepts of Molehill API. You will be presented with the Molehill programming pipeline via a practical example with detailed explanations on each step. In this recipe you will learn how to set up the native 3D environment in Flash. Then we are going to create a sphere primitive and animate its rotation using the pure Molehill API.

Getting ready

1. The first thing you have to do is to install the latest prelease build of Flash Player 11 called incubator which at the time of writing is 11,0,0,58d. on your system.
You should get the player from this page: http://labs.adobe.com/downloads/flashplatformruntimes_incubator.html.
2. Another thing we are going to use is the latest build of Flex SDK 4.5 which is at the time of writing: 4.5.0.19786: <http://opensource.adobe.com/wiki/display/flexsdk/Download+Flex+Hero>.
3. AGALMiniAssembler and PerspectiveMatrix3D are additional utilities we have to include when writing Molehill-based applications. You can get these classes from here: <http://www.bytearray.org/?p=2555>.



The PerspectiveMatrix3D class is optional. It can save you precious time and headaches to write your own perspective projection matrix which you need in order to draw your geometry onto the screen. You can always "roll your own" if you feel more comfortable doing all the stuff by yourself.

4. Here are the links to the detailed step-by-step installation guide which you must read in order to accomplish the installations successfully:

http://labs.adobe.com/wiki/index.php/Flash_Player_Incubator.

<http://www.allforthecode.co.uk/aftc/forum/user/modules/forum/article.php?index=5&subindex=1&aid=263>.



Don't forget to uninstall the previous version of the Flash Player before installing the incubator.

5. Go to Appendix source code directory and copy into your project package:utils which contains ResourceUtil class. This class integrates a method that will help us to construct a sphere primitive. The method snippet is originally a part of Away3D 4.0 pre-release.
6. Make sure you also put into your project the com.adobe.utils page containing PerspectiveMatrix3D and AGALMiniAssembler classes mentioned previously.

How to do it...

In the following program we are going to create a basic Molehill API setup and render a simple rotating sphere. Paste this code into a new ActionScript file named RawMolehillBasic:

```
package
{
    public class RawMolehillBasic extends Sprite
    {
        private var _indexBuffer:IndexBuffer3D;
        private var _context3D:Context3D;
        private var _prog:Program3D;
        private var _vertexBuffer:VertexBuffer3D;
        private var _vertexBuffer1:VertexBuffer3D;
        private var _matr:Matrix3D;
        private var _stage3D:Stage3D;
        private var _perspectiveMatr:PerspectiveMatrix3D;
        private var _rotation:Number = 0;
        private var _triangleVertsArr:Vector.<Number>=new
        Vector.<Number>();
        private var _colorsArr:Vector.<Number>=new Vector.<Number>();
        private var _triangleIndicesArr:Vector.<uint>= new
        Vector.<uint>();
        public function RawMolehillBasic()
        {

```

```
    initGeomtryData();
    init();
}
private function init():void{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;
    _stage3D= stage.stage3Ds[0];      _stage3D.addEventListener(Event.
CONTEXT3D_CREATE,onContextReady);
    _stage3D.requestContext3D();
    _stage3D.viewPort=new Rectangle(0,0,800,600);
}
private function initGeomtryData():void{
    var res:ResourcesUtil=new ResourcesUtil();
    _triangleVertsArr=res.sphereVertices;
    _triangleIndicesArr=res.sphereIndices;
}
private function onContextReady(e:Event):void{
    _context3D=_stage3D.context3D;
    _context3D.configureBackBuffer(800,600,4,true);
    _context3D.enableErrorChecking=true;
    _context3D.setCulling(Context3DTriangleFace.BACK);
    _context3D.setDepthTest(true,Context3DCompareMode.LESS_EQUAL);
    initGeomBuffers();
    initColorBuffer();
    var vertexAssemby:AGALMiniAssembler = new AGALMiniAssembler();
    vertexAssemby.assemble(Context3DProgramType.VERTEX, __
vertexShader);
    var fragmentAssembly:AGALMiniAssembler = new
AGALMiniAssembler();
    fragmentAssembly.assemble(Context3DProgramType.FRAGMENT, __
fragmentShader);
    _prog=_context3D.createProgram();      _prog.upload(vertexAssemby.
agalcode,fragmentAssembly.agalcode);
    _context3D.setProgram(_prog);
    _matr=new Matrix3D();
    _perspectiveMatr=new PerspectiveMatrix3D();
    _perspectiveMatr.perspectiveFieldOfViewRH(0.4, 4 / 3, 0.1,
1000);
    _context3D.setProgramConstantsFromMatrix(Context3DProgramType.
VERTEX, 4, _perspectiveMatr, true);
    addEventListener(Event.ENTER_FRAME,onEnterFrame);
}
private function getColorsArr():Vector.<Number>{
    var colVector:Vector.<Number>=new Vector.<Number>();
```

```

        for(var i:int=0;i<_triangleVertsArr.length;++i){
            colVector.push(Math.cos(Math.random()*3));
        }
        return colVector;
    }
    private function initColorBuffer():void{
        _colorsArr= getColorsArr();
        _vertexBuffer1=_context3D.createVertexBuffer(_colorsArr.length/3,3);
        _vertexBuffer1.uploadFromVector(_colorsArr,0,_colorsArr.length/3);
    }
    private function initGeomBuffers():void{
        _vertexBuffer=_context3D.createVertexBuffer(_triangleVertsArr.length/3,3);
        _vertexBuffer.uploadFromVector(_triangleVertsArr,0,_triangleVertsArr.length/3);
        _indexBuffer=_context3D.createIndexBuffer(_triangleIndicesArr.length);
        _indexBuffer.uploadFromVector(_triangleIndicesArr,0,_triangleIndicesArr.length);
    }
    private function onEnterFrame(e:Event):void{
        _context3D.clear(0.5,0.2,0.2);
        _context3D.setVertexBufferAt(0,_vertexBuffer,0,Context3DVertexBufferFormat.FLOAT_3);
        _context3D.setVertexBufferAt(1,_vertexBuffer1,0,Context3DVertexBufferFormat.FLOAT_3);
        renderBall(1,30,1,150);
        _context3D.present();
        _rotation+=3;
        if(_rotation>359){
            _rotation=0;
        }
    }
    private function renderBall(scale:Number,circleRadius:Number,rotDir:int,depth:int):void{
        _matr.identity();
        _matr.prependTranslation(0,0,-depth);
        _matr.prependScale(scale,scale,scale);
        var rotVector:Vector3D=new Vector3D(1,0,1);
        rotVector.normalize();
        _matr.prependRotation(_rotation*rotDir,rotVector);
        _context3D.setProgramConstantsFromMatrix(Context3DProgramType.VERTEX,0,_matr,true);
        _context3D.drawTriangles(_indexBuffer,0,_triangleIndicesArr.length/3);
    }
}

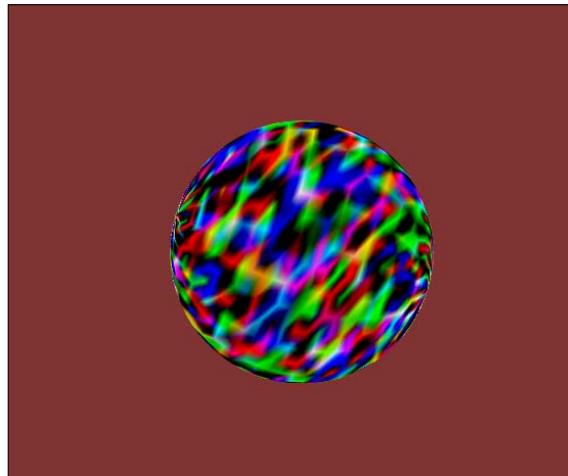
```

```
protected var _vertexShader:String = [
    "m44 vt0, va0, vc0 \n" +
    "m44 op, vt0, vc4 \n" +
    "mov v0, va1"
].join("\n");

protected var _fragmentShader:String = [
    "mov oc, v0"
].join("\n");
}

}
```

The following image is our sphere, by courtesy of GPU. It consists of 1,089 vertices which are 1,984 triangles and if you did not forget to set the Flash Player wmode=direct you should get a steady 60 out of 60 fps.

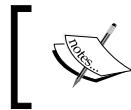


How it works...

If from the first glance in the preceding code you are getting the feeling of panic, just calm down because now we will explain step-by-step all those Molehill API-related objects and methods and even the terrible assembly syntax.

The first function we call from the `RawMolehillBasic` constructor is `initGeometryData()`. Inside that function we fill two vectors `_triangleVertsArr` and `_triangleIndicesArr` with vertex and indices data respectively which we need to construct the sphere primitive. The `_triangleVertsArr` vector contains 3,267 numbers which represent groups of three numbers for each vertex. `_triangleIndicesArr` contains a sequence of numbers which define the order of triangles drawing out of the vertex data.

We will get back to these two variables soon when you will see how we initiate buffers. For now let's move on to the next method which is `init()`. In `init()` we set an instance of `Stage3D`. `Stage3D` is the new layer presented in Flash Player 11 and its sole purpose is to wrap 3D content defined by `Context3D` about which we are going to talk next.



Stage3D layer is always located behind the regular 2D stage. There is no way to put it in front of it or exchange display objects between the two except rendering the `Context3d` to texture.



As you can see we access `Stage3D` instance via `stage.stage3Ds[0]`. `Stage` allows you to define up to four `stage3Ds` at a time.



Keep in mind it can reduce the performance so better stick to one `Stage3D` instance per application.



Next we are going to get reference to `Context3D` instance which is supplied by `_stage3D`. Each `Stage3D` has got only one `Context3D`. We assign event listener of the type `Event.CONTEXT3D_CREATE` before we call `_stage3D.requestContext3D()`. You must not forget this step as the request for `Context3D` is an asynchronous operation. `Context3D` class is of supreme importance for 3D programs. Think of it as an entity consisting of all the resources and the required commands which are passed to GPU to process the graphics. All the buffers, colors, textures, shader programs, and more are set in `Context3D` which then accesses the specified graphics subsystem to process the state and display the result. The last line in `init()` function is `_stage3D.viewPort=new Rectangle(0,0,800,600)`; which simply defines the view port window dimensions for `Stage3D`. So when the `CONTEXT3D_CREATE` event is fired `onContextReady()` function gets called. Here the fun begins. First we access the reference of `_context3D` object via `_stage3D`. Then configure a back buffer:

```
_context3D.configureBackBuffer(800,600,4,true);
```

Back buffer in 3D graphics is responsible for drawing the next frame content off the screen before it is rendered. The technique is known as double buffering. When the content of the back buffer is ready for show, it is swapped with the current frame buffer on the screen. This process prevents the visual content from flickering during the rendering and generally smoothes the render process.

Usually you would define the dimensions of the back buffer the same as of the viewport (try a smaller size and you get some cool pixilated results).

Next we set `_context3D.enableErrorChecking=true`. This setting enables the flash to check for the success of the rendering looking for potential errors. Try to avoid using it in the production mode as it slows down the render process.

Now we define the object's culling mode:

```
_context3D.setCulling(Context3DTriangleFace.BACK);
```

We set it to back face culling which means that the triangles which are not in the view don't face the camera in the range 0 to 180 degrees and will not be processed. Right after this comes depth-sorting definition:

```
_context3D.setDepthTest(true, Context3DCompareMode.LESS_EQUAL);
```

We are not going to dive into a detailed explanation of each sorting algorithm but usually you would use LESS or LESS_EQUAL for right depth sorting and ALWAYS if you need no sorting to be run. Also you can disable the depth test by setting the first parameter to false.

Next we come to the vertex and index buffers. Molehill supplies you with two important classes—`VertexBuffer3D` and `IndexBuffer3D`. Their job is to deliver vertices and indices for your geometry onto the graphic device. It is very important to understand how to define them otherwise you can spend hours trying to ignite your program. Let's take a look at `initGeomBuffers()` function. First we instantiate `_vertexBuffer` via `_context3D`. You are not limited in the number of buffers you can set. The first parameter of `createVertexBuffer()` method defines the total number of vertices. To tell the buffer how many vertices to pass, we divide the `_vertexBuffer` length by three (each vertex requires three numbers for XYZ location) to get the total number of vertices that will be created. The second parameter specifies how many numbers in the array are designated for each vertex. In our case it is three that means every three consecutive numbers are used for a single vertex construction. After we defined the buffer and allocated the space for its data in the memory, we have to fill it with the actual data using `uploadFromVector`. Here we use our resource vectors we set in the beginning of the program:

```
_vertexBuffer.uploadFromVector(_triangleVertsArr, 0, _triangleVertsArr.length/3);
```

You pass the following arguments into the constructor: the source vector containing the vertex data, then the start position in the array from where the buffer should start reading the data. The last argument specifies the number of vertices you are going to create. With the next two lines we set up `_indexBuffer`:

```
_indexBuffer=_context3D.createIndexBuffer(_triangleIndicesArr.length); _indexBuffer.uploadFromVector(_triangleIndicesArr, 0, _triangleIndicesArr.length);
```

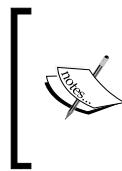
You can see that this is the same routine as for the vertex buffer. The only difference is that we use `IndexBuffer3D` class to store the indices.

We are done with mesh structure definition. Now we have to take care about the appearance. The next buffer we set up is color buffer calling `initColorBuffer()` function. In 3D graphics you are able to assign color values to vertices. While it doesn't paint each vertex like a point as you might have thought, it comes into play during the rasterization process when the triangle surface gets painted by a fragment shader. If we don't use a texture, the simplest way to color your model is to assign colors to each vertex. The colors of each three vertices interpolate across the triangle surface producing interesting gradient transition like the one you have seen at the ball surface in our program. Because the colors are float numbers, we should use `VertexBuffer3D` to store this kind of data.

You may find a wide range of examples on the internet regarding loading colors for the vertices in Molehill where the approach is to use a single `VertexBuffer3D` instance to store the vertices and colors together. This is fine when you deal with simple objects, but if you generate vertex data for complex meshes such as sphere or teapot, it may complicate things as you will have to supply a vector that holds six or seven numbers for each vertex which look as follows: (Vertex X position, Vertex Y position, Vertex Z position, Vertex R color, Vertex B Color, Vertex G color, Vertex Alpha (optional)). It may reduce the flexibility of customization of color data later therefore in this chapter you are shown how to combine between different buffers each one having its specific task.

So back to business. Inside `initColorBuffer()` we generate random colors from zero to three using `getColorsArr()` filling `_colorsArr` vector the length identical to that of vertex buffer. Next, once again we should set a buffer which is `_vertexBuffer1` in this case to store this data. Similar to vertex buffer we defined for the vertices, here we specify as arguments the number of vertices and how many numbers the buffer should use for each vertex. You should remember that here each vertex set is an RGB value which will be assigned by a vertex shader program to each vertex of the mesh then passed into a fragment shader to paint the surface of the triangles.

We are finished with the resources setup. Now it is time to define the commands for GPU. As was said earlier, graphics cards processing units are fully programmable. The truth is that if you don't pass the instruction on how you want to render the graphics, your graphic card will have no idea what you want from it. Therefore you have to set programs which tell the graphic card processors what to do and how. By programming the GPU, the sky is the only limit on what kind of crazy stuff you can put on your screen. Having said this, we come to AGALMiniAssembler. This utility is supplied by Adobe and its purpose is to assemble the vertex and fragment programs you write using raw AGAL opcodes into `ByteArray`.



PixelBender3D is a 3D GPU-accelerated version of a well known PixelBender language which is supposed to make our life easier when writing shader programs for Molehill API. At the time of writing, Adobe released the alpha version which is extremely unstable. Therefore to the end of this chapter we will stick to raw AGAL opcodes.

We first create an instance of AGALMiniAssembler for a vertex program:

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

Then we call assemble() method in which the first argument is a program type and the second, the program itself which is a string of operations written with opcodes. The same we do for the fragment shader but this time specifying FRAGMENT type and passing the fragment program string:

```
var fragmentAssembly:AGALMiniAssembler = new AGALMiniAssembler();
fragmentAssembly.assemble(Context3DProgramType.FRAGMENT, _  
    fragmentShader);
```

Next we instantiate Program3D and put the vertex and fragment shader Byte arrays into it:

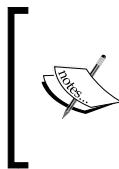
```
_prog=_context3D.createProgram();      _prog.  
upload(vertexAssembly.agalcode,fragmentAssembly.agalcode);
```

Finally set the program to the _context3D. We will skip the discussion on vertex and fragment shader and leave it for the end as we first need to accomplish the essential setup required by those programs.

Now comes an unpleasant part—**Matrices**. As it was said Molehill is a pretty low-level API so all the transformations are done manually—no Away3D magic under the hood. Usually in 3D graphics you should set at least two types of matrices for your world: modelview matrix, and projection matrix. In more complex cases when you would like to group objects in containers, you will use additional matrices for coordinate's space separation. In this simple example we have to first define the position and orientation of the sphere then project it onto the screen using perspective projection. Right after _context3D.setProgram(_prog); we instantiate _matr which will be used to position and rotate the model on each frame. Next we define a perspective projection matrix—utility which is supplied by Adobe along with AGALMiniAssembler:

```
_perspectiveMatr=new PerspectiveMatrix3D();
_perspectiveMatr.perspectiveFieldOfViewRH(0.4, 4 / 3, 0.1, 1000);
```

Among the numerous methods that this class has got, we use `perspectiveFieldOfViewRH()` which defines the perspective projection matrix for a right-handed coordinate system.



A right-handed coordinate system is when z-axis negative values are away from the viewer, x-axis positive to right, and y-axis positive is up. Contrary to Away3D 3.6 which has a left-hand coordinate system Molehill is defined based on what type of function you use in PerspectiveMatrix3D class.

The function arguments are: Near Plane position, screen aspect ratio, far plane (furthest from the viewer render boundary). The next line is extremely important to understand. It is not enough to define a matrix. You need to tell of its existence to the shader program and point the reference to the matrix so that during the execution the vertex or fragment program, it can be able to use the matrix in the calculation process. In the following line we define a program constant of a matrix type:

```
_context3D.setProgramConstantsFromMatrix(Context3DProgramType.VERTEX,  
4, _perspectiveMatr, true);
```

You will see how we access this constant from the opcodes later, for now let's see what we pass as arguments into the method. First you specify to which program this constant will be available. Because we use `_perspectiveMatr` to transform vertices it is passed into the vertex program. The next argument is the position of the first register in the constants stack where the shader program should access this specific constant. Let's get into it in more detail. Each register may have a maximum of four floating point numbers. We set `_perspectiveMatr` from the fourth register onwards because of two reasons:

1. The matrix is 4x4 and in total has 16 numbers which means it should occupy four registers.
2. The registers from 0 to 4 are occupied by `_matr` variable which we defined earlier and we pass it as the program constant on each frame. That is because we update this matrix (its rotation) continuously.

You may want to put the projection matrix constant at register 0 and the modelview matrix at position four, but in this case you must switch between constant variables in opcode expression. The last Boolean argument is matrix transposition. If true, the matrix is rearranged so that its columns and rows swap their respective position. We set it to true as we need it for a right-hand coordinate system. The last line here is to add `Event.ENTERFRAME` event listener which triggers `onEnterFrame()` method inside it. Finally draw the 3D content on the screen.

The first operation we have to execute before each draw is to clear the buffers:

```
_context3D.clear(0.5, 0.2, 0.2);
```

`clear()` method clears the buffers on `_context3D` with specified RGB color that present a background in the stage3D. After the buffers are cleared we reset them again:

```
_context3D.setVertexBufferAt(0, _vertexBuffer, 0, Context3DVertexBufferFormat.FLOAT_3);  
_context3D.setVertexBufferAt(1, _vertexBuffer1, 0, Context3DVertexBufferFormat.FLOAT_3);
```

In our case we set two buffers one for vertices, and one for vertex colors. These buffers are accessed by the vertex program using their index property that we set as the first parameter for each `setVertexBufferAt()` method. Other arguments are the source buffer containing the actual data buffer format which is `FLOAT_3` because as you remember we use three numbers for each vertex in both buffers.

After we set the buffers, we can proceed with the geometry drawing. We do this in a separate method called `renderBall()`. The function accepts parameters for sphere scale, radius, rotation direction, and z-position which can be customized during the runtime. Inside the method, we first take care of the modelview matrix which we defined in `onContextReady()` method but left it in default state. First we reset the matrix after the last frame operation:

```
_matr.identity();
```

Next we define its translation, scale, and rotation portions. Be careful as the sequence of these operations matters a lot:

```
_matr.prependTranslation(0,0,-depth);
_matr.prependScale(scale,scale,scale);
var rotVector:Vector3D=new Vector3D(1,0,1);
rotVector.normalize();
_matr.prependRotation(_rotation*rotDir,rotVector);
```

Now when the matrix is ready we pass it as the constant to the vertex shader program just as we have done with the perspective projection matrix before:

```
_context3D.setProgramConstantsFromMatrix(Context3DProgramType.
VERTEX,0,_matr,true);
```

The matrix now is accessible from the vertex program. Last thing left is to draw the geometry:

```
_context3D.drawTriangles(_indexBuffer,0,_triangleIndicesArr.length/3);
```

Inside the method you pass `_indexBuffer`, the start position from which the drawing begins and the number of triangles to draw. Back to the `onEnterFrame()` function right after `renderBall(1,30,1,150)` we call `_context3Dpresent()`; which swaps the back buffer with the current one and the last three lines we increment `_rotation` property that serves for the `_matr` as the rotation degrees input:

```
_rotation+=3;
if(_rotation>359){
    _rotation=0;
}
```

The last thing we are left with is vertex and fragment programs. It is important to get a basic understanding of how they work if you want to proceed with Molehill API towards serious applications. Fear not. This is not as scary as it looks at first glance.

Vertex shader:

Now let's decompose the shader programs we have written at the bottom of the class.

This is in no way an AGAL primer. Please see online documentation to learn in detail various opcodes and their implementations.

Also the following links may be useful: http://download.macromedia.com/pub/labs/flashplatformruntimes/incubator/flashplayer_inc_langref_022711.zip

<http://iflash3d.com/shaders/my-name-is-agal-i-come-from-adobe-1/>.

We begin with a vertex program as it is executed first in the rendering pipeline. First let's understand the structure of the program and the opcodes we use in this example so that you will get a better understanding of each expression later on. The variable `_vertexShader` contains a string of expressions, three in total for this example. Each expression is separated by `\n` delimiter. This is important as forgetting to terminate each expression by `\n` will produce compile errors because `AGALMiniAssembler` will try to treat two expressions as one.

Each expression consists of several members taking part in the operation. The first member is always an opcode which defines a type of an operation. In this case the first line first member is `m44` opcode which stands for 4×4 matrix. (See Molehill API for full list of available opcodes). It means that we are going to execute matrix multiplication here which requires additional three operands called registers which represent places in the memory storing a specific data. For different opcodes operations the number of required register varies. In our example:

vt[x]—temporary variable to store the multiplication result whereas `x` is the unique number that makes this variable differ from others. If you have got several different temporary variables in the program you could name them `vt0`, `vt2`, `vt3`, and so on.

va[x]—pointer to the vertex buffer source. `[x]` suffix matches the index parameter of loaded buffer in `_context3D.setVertexBufferAt()` method. In our case we use the buffer with vertex data which has an index of 0 in `_context3D` so we reference it with `va0`.

vc[x]—is the register holding the program constant defined by `_context3D.setProgramConstantsFromMatrix()` or `_context3D.setProgramConstantsFromVector()` methods. `[x]` number in `vc` should match the 3D parameter called `firstRegister` in one of the previously mentioned methods. In our program if you remember we set two constants—one for modelview matrix, and one for projection matrix. The first matrix we use for the sphere transformation is `_matr` and we set its `firstRegister=0`. Perspective matrix was set to register number 4 because registers 0,1,2,4 are occupied by `_matr`. Therefore when we need to access the perspective matrix constant, we will use `vc4` register. Let's summarize the first expression.

We multiply the vertices in the vertex buffer `[vt0]` by matrix 4×4 `[m44]` from the constant `[vc0]`.

Second expression "`m44 op, vt0, vc4 \n`" should by now make more sense for you except one new register—`op`:

`op`—is the final vertex output. You use this register to store the resulting vertex data when no more operations on it are required. This way the second line means the following.

We multiply the vertices in the temporary register `vt0` (from the previous operation) by the perspective matrix `m44` which we access via the matrix constant `vc4`.

The last line in the vertex shader sends the second vertex buffer we defined for vertex colors called `_vertexBuffer1` to the fragment shader program that we will talk about next. We must do it so that the fragment program can be able to color the triangles surfaces based on color interpolations between the RGB values assigned to the vertices via this buffer.

`mov`—opcode moves the data.

`v0`—is the varying register used to pass data to fragment shader.

And the third line basically moves the vertex data—`v0` of `_vertexBuffer1`—`va1` into the fragment program.

Fragment program:

The fragment program in our example is the simplest on earth. It outputs the color of the fragments based on the color vertex buffer it is supplied. Here we move vertex color data received from the vertex shader—`v0` into the color output for the fragments—`oc`. This is it. Very simple. Of course real life shaders are much more complex and usually involve many lines of expressions to produce cutting edge results.

There's more...

Now you surely ask yourself how you can put several objects into the scene. Well it is much simpler than you could imagine.

Duplicate the class `RawMolehillBasic` into the class named `RawMolehill`. Make sure you have changed the class and its constructor names accordingly. Go to the `onEnterFrame()` method and delete all the code following this line:

```
_context3D.setVertexBufferAt(1,_vertexBuffer1,0,Context3DVertexBufferFormat.FLOAT_3);
```

Instead put this snippet:

```
for(var i:int=1;i<14;++i){  
    var rotFactor:Number;  
    i%2==0?rotFactor=1:rotFactor=-1;  
    var depth:int=i*14;
```

```
var scale:Number=Math.sin(getTimer()/1000)/i/1.5;
    renderBall(scale,30/i,rotFactor,depth);
}
_context3D.present();
trot+=3;
if(trot>359){
    trot=0;
}
```

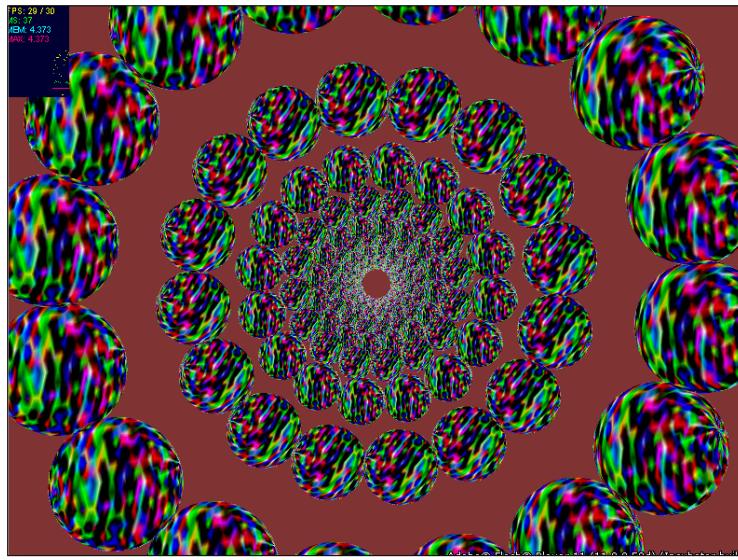
Next delete `renderBall()` function and instead insert this new version of `renderBall()`:

```
private function renderBall(scale:Number,circleRadius:Number,rotDir:int,depth:int):void{
    for(var i:int=0;i<NUM_BALLS;++i){
        _matr.identity();
        var angle:Number = Math.PI * 2 /NUM_BALLS * i;
        var xpos:Number = Math.cos(angle) * circleRadius;
        var ypos:Number = Math.sin(angle) * circleRadius;
        var rotMatr:Matrix3D=new Matrix3D();
        rotMatr.appendRotation(trot,new Vector3D(1,0,0));
        _matr.appendTranslation(xpos,ypos,-depth);
        _matr.prependScale(scale,scale,scale);
        _matr.appendRotation(trot*rotDir,Vector3D.Z_AXIS,new
        Vector3D(0,0,0));      _context3D.setProgramConstantsFromMatrix(Conte
xt3DProgramType.VERTEX,8,_matr,true);      _context3D.setProgramCo
nstantsFromMatrix(Context3DProgramType.VERTEX,0,_matr,true);
        context3D.drawTriangles(_indexBuffer,0,_triangleIndicesArr.length/3);
    }
}
```

The last thing we need to change is the `_vertexShader` variable content. Replace all the content with this string:

```
"m44 vt0, va0, vc8 \n" +
"m44 vt1, vt0, vc0 \n" +
"m44 op, vt1, vc4 \n" +
"mov v0, val"
```

Run the program and you should get this:



Although by now you should be already familiar with the Molehill basics we will be briefly explaining this program to get the idea what has happened here.

Let's begin with `renderBall()` function. We made some changes here. Now the function includes the `for` loop statement which creates us an array of spheres positioned in a ring formation. These three lines calculate the x and y position for each sphere on the ring:

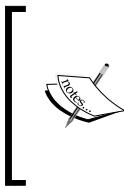
```
var angle:Number = Math.PI * 2 /NUM_BALLS * i;
var xpos:Number = Math.cos(angle) * circleRadius;
var ypos:Number = Math.sin(angle) * circleRadius;
```

Next comes the matrix stuff as we need to apply the transformations to the spheres. First you can see that we have an additional `matrix3D` instance called `rotMatr`:

```
var rotMatr:Matrix3D=new Matrix3D();
rotMatr.appendRotation(trot,new Vector3D(1,0,0));
```

This matrix will be applied first between the three to each object and its purpose to set the rotation for each individual sphere on its local axis. Here we rotate each sphere around its local x-axis. After this we proceed with the second matrix instance `_matr`. It's task is to deploy all the spheres in the ring formation then to rotate the whole ring. It is also responsible for scale:

```
_matr.appendTranslation(xpos,ypos,-depth);
_matr.prependScale(scale,scale,scale);
_matr.appendRotation(trot*rotDir,Vector3D.Z_AXIS);
```



The matrix operations order is very important. In this case we want to rotate the whole ring of the spheres around its center. We first perform translation followed by rotation. Doing conversely will cause the same result for which `rotMatr` is responsible for the local rotation of each sphere.

Next we define the program constants to hold `rotMatr` and `_matr` values. Finally we call `drawTriangles()` method to render each sphere on the screen.

Let's take a look at the changes we made to the vertex shader program. We will explain each line:

```
"m44 vt0, va0, vc8 \n"
```

The vertex buffer—`va0` is multiplied by the rotation matrix—(`rotMatr`) `vc8` and stored in temp register `vt0`:

```
"m44 vt1, vt0, vc0 \n"
```

The data from the temp register—`vt0` is again multiplied but this time by the transformation matrix (`_matr`) `vc0` and stored in the new temp register—`vt1`:

```
"m44 op, vt1, vc4 \n" +
```

The data from temp register—`vt1` is multiplied by the last, the projection matrix (`_perspectiveMatr`), and outputted:

```
"mov v0, va1"
```

This one you already know. It passes the second vertex buffer holding vertex colors to the fragment shader.

Finally in the `onEnterFrame()` method we set the `for` loop statement which sets 14 sphere rings of decreasing depth by calling the `renderBall()` method on each iteration. You can see a use of `getTimer()` method to produce a looping scale effect over time.

In total we render 224 balls on each frame. If you remember, each ball contains 1,984 triangles. Doing simple math we get 444,416 triangles each frame! At my relatively weak notebook, I get 30 steady FPS. I hope you can imagine what would happen if you attempt to make it in Flash Player 10.

Compacting the matrix operations

A few more words should be said regarding the matrix operations we performed in the previous example. We can reduce the number of matrix constants we load into the vertex shader. In fact we can pack all the transformations into one single matrix. If you remember, in the preceding sample we set three different matrices:

- ▶ Rotation matrix—to perform a local rotation of each sphere

- ▶ Transformation matrix–positions array of spheres in circular formations and rotates them around the circle center
- ▶ Perspective projection matrix–projects the geometry to the screen space

In order to union all these matrices into one, we can create a new matrix and multiply it sequentially by each one of three matrices. Then we can load only this resulting matrix as constant into the vertex program.



The reason we broke the matrix operations into several constants was to demonstrate the concept of constant registers manipulation in AGAL. In the real world application, you would probably prefer to upload as few constants as possible in order to keep the constants stack free. That may be crucial if you need to apply multiple matrix calculations because you can load a maximum of 128 constants per program.

So let's add some modifications into the last program. First add a new matrix variable into the global variables list:

```
private var _finalMatr:Matrix3D;
```

Now inside `onContextReady()` method first comment this line:

```
_context3D.setProgramConstantsFromMatrix(Context3DProgramType.VERTEX,  
4, _perspectiveMatr, true);
```

So we don't load the constant for the projection matrix into the shader anymore. Right beneath that line instantiate the `_finalMatr`:

```
_finalMatr=new Matrix3D();
```

Now we move to `renderBall()` method. Here we need to comment out the following lines:

```
_context3D.setProgramConstantsFromMatrix(Context3DProgramType.  
VERTEX,8,rotMatr,true);  
_context3D.setProgramConstantsFromMatrix(Context3DProgramType.  
VERTEX,0,_matr,true);
```

The same is here; we cancel the loading of these constants because finally only a single matrix will be passed into the vertex program.

Now it is time to fill the `_finalMatrix` with the data from all the three previously listed matrices. Insert the following code into `renderBall()` method right after `_matr.appendRotation(trot*rotDir,Vector3D.Z_AXIS);` line:

```
    _finalMatr.identity();
    _finalMatr.append(rotMatr);
    _finalMatr.append(_matr);
    _finalMatr.append(_perspectiveMatr);
    _context3D.setProgramConstantsFromMatrix(Context3DProgramType.
VERTEX, 0, _finalMatr, true);
```

Let's explain this. By appending each one of the three matrices to `_finalMatr`, we basically multiply them. After these operations `_finalMatr` holds the final data needed to transform and project the geometry on the stage. The last line sets only `_finalMatr` as the program constant at the register zero.

The final step is to moderate the AGAL code for the vertex program. Comment out the whole `_vertexShader` variable. Instead put this modified version:

```
protected var _vertexShader:String = [
    "m44 op, va0, vc0 \n" +
    "mov v0, val"
].join("\n");
```

As you can see this version is much more compact than the previous one. That is because we perform only one single matrix constant "vc0" multiplication of the vertices "va0" instead of three steps we have done before that. Run your program and you should see the same result as you had in the preceding example.

See also

From here you are suggested to see more info on Molehill API at Adobe Labs:

http://labs.adobe.com/downloads/flashplatformruntimes_incubator.html

BiteArray.org

<http://www.bytearray.org>

And of course go to the Away3D homepage to download the new Molehill-based version—Away3D 4:

www.away3D.com

Index

Symbols

2D shapes

making appear 3D, Sprite 3D used 340-343

3D illusion

creating, with Away3D sprites 208-211

3D objects

creating, from 2D vector data 200-203

masking 183-186

3ds 262

3D scene assets

preloading 283-288

3dsMax

character rigging 86-89

3DsMax 239, 261, 262

_awayPhys.createGround() method 293

_awayPhys.step() method 293

_barry 177

_bloader.items array 288

_bloader.start() method 288

_cam.screen() method 74

_camTarget 70

_canDraw flag 207

_canMove flag 207

_colorsArr vector 439

_dragVector 354

_elevation.generate() 391

=_elevation.generate() 391

_explodeContainer 162

_extrMod.execute() a method 409

_faceLink.update() method 117

<flarSourceSettings> 315

_lastMouseX 200

_loadedModels array 288

_mainContainer object 110

_mergedContainer 162

_morphPattern vertices 104
_pathModifier.execute() 199
_perlin3D array 172
_rotation property 442
_springCam.view property 70
_spriteArr 177
_stage3D.requestContext3D() 437
_startVector 352
_targetVertices array 111
_triangleIndicesArr vector 436
_triangleVertsArr vector 436
_walker.walk() method 391
_waterNormalMap 408
_weld.apply() command 374

A

A3DObjectClass class 313

AABB (axially aligned bounding box) 392

AABBTest() function 395

AABBTest() method 395

AABS test

about 396

working 396

AC3D 228

accessAnimationByName() method 95, 99

accessAnimationData() function 94

accessAnimationData() method 93

ActionScript3.0 LocalConnection object

instance 239

ActionScript class 290

addChild method 182

addChild() method 419

addDirectionalMaterial() method 211

addExternalGeometry() 429

add() method 288

addError event listener 285

addOnSuccess event listener **285**
addOverlay() method **182**
Adobe Graphics Assembly Language. *See AGAL*
advanced bitmap effects
 creating, filters used **164-167**
 working **168**
advanced object rotation
 implementing, vector math used **125-127**
advanced spherical surface transitions
 creating, with quaternions **128-132**
AGAL **431**
AGALMiniAssembler **432, 439**
Airbrush **136**
AnimatedRoad class **259**
AnimatedSWF.swf **201**
AnimationData **93**
AnimationData object **94**
AnimationLibrary **93**
AnotherRivalFilter **220**
appendRotation() **340**
applyFilters() function **168, 183**
apply() method **309**
applyModifier() function **309**
applyWeld() method **374**
Arcball. *See virtual trackball*
AS3MOD **289**
AS3MOD library **303**
AS3MODDemo.as **304**
Ase (ASCII) **262**
assemble() method **440**
attachToPath() function **198**
attachToPath() method **199**
Augmented Reality (AR)
 about **314**
 setting, with FLARToolkit **314-318**
Autodesk **261**
Autodesk 3dsMax **208**
Autodesk 3dsMax version 7, 86
Away3D
 3D illusion, creating with Away3D sprites **208-211**
 3D objects, creating from 2D vector data **200-203**
 3D objects, masking **183-186**
 3D scene assets, preloading **283-288**
 about **7**
 advanced bitmap effects, creating using filters **164-167**
 advanced object rotation, implementing using vector math **125-127**
 advanced spherical surface transitions, creating with quaternions **128-132**
 artifacts, removing from intersecting objects **217-220**
 cameras **41**
 clouds, creating **169-171**
 collisions between objects, detecting **392-395**
 controllable non-physical car, creating **145-152**
 cool effects, creating with animated normal maps **403-409**
DepthBitmapMaterial **21**
 depth sorting problems, fixing with Layers and Render Mode **221-226**
 dynamic text, setting with **TextField3D** **190-195**
 external assets, accessing in SWF **280-283**
 external assets, storing in SWF **280-283**
 geometry artifacts, fixing with Frustum and NearField clipping **214-217**
 geometry, dragging by un-projecting mouse coordinates **137-140**
 geometry, exploding **154-158**
 heavy scenes, running faster using BSP trees **417-429**
 lens flare effects, creating **179-182**
 light maps, generating **248-252**
 material color, interpolating with **DepthBitmapMaterial** **21-23**
 mesh, morphing interactively **141-144**
 models, exporting from 3DsMax/Maya/Blender **262**
 models, loading **277-280**
 models, parsing **277-280**
 model structure, painting interactively **133-136**
MovieClip of **MovieMaterial**, controlling **12-15**
MovieClip, using for multiple materials **8-12**
MovieMaterial **8**
 normal maps, creating with **NormalBumpMaker** utility **29, 30**

normal maps, creating with NormalMapGenerator 26-29
particles, exploding with FLINT 310-314
performance, optimizing 361, 362
PixelBender materials 30
rotating sphere, creating using Molehill API 432-447
segments, drawing with 204-207
setting up, with JigLib 290-294
SkyBox6-compatible texture, creating 412
skyboxes, creating 409-412
skybox textures, creating 409-412
skybox setup 413-415
sound, visualizing 173-178
terrain, creating 244-248
TextField3D text, attaching to user specified path 195-199
uneven surfaces, moving 387-391
VideoMaterial 16
workflow, maintaining with AwayConnector 238, 239

Away3D FaceLink class 112

Away3D geometry
morphing, with AS3DMOD 303-307

Away3D LensFlair class 179

Away3D Lite
about 329
BasicTemplate 331
embedded models, parsing 334
external models, importing 332, 333
FastTemplate 331
features 329
geometry, manipulating 335-339
setting up, templates used 330
Sprite 3D 340
virtual trackball, creating 349-354
working 331
Z-Sorting, layers used 346-349
Z-Sorting, managing 343-346

Away3D Lite applications
writing, with haXe 355-359

Away3D Morpher tool 100

Away3D objects
Box2D physics, adding 320-328

Away3DPhysics() plugin 293

Away3D primitive
Away3D primitive rotational interactivity, adding using Mouse movements 120-124

AwayConnector
about 238
workflow, maintaining with 238, 239

AwayHolder 17

AwayLiteAssetLoadDemo.as 332

AwayScene 17

AwaySetupDemo.as 330

AwayTemplate class 90, 106, 154, 362

AWData() 334

AWDataLoadingDemo.as 231

AWD format 228

B

barrel96.png bitmap 233
barrel_empty_low.3ds 228

BasicRenderer 220

BasicTemplate 331

BeTween 105

BitmapData 8, 164
bitmapdata.draw() function 16

BitmapData object 182

BitMapDemo.as 381

Blender 261

BlendingMode property 137

BlurFilter 168

BonesAnimator variable 93

bones-based Collada animation
controlling 90-92
working 93, 94

Box2DFlash 320, 385

Box2D physics
adding, to Away3D objects 320-328

BulkLoader 283

BulkLoaderDemo.as 286

ButtonGraphics, utility class 12

byteArray 283
ByteArray 177

C

cameras
about 41
Camera3D 41

DOF effect, creating 50-53
 FPS controller, creating 42-46
HoverCamera3D 41
 lens, changing 60-66
 object location, detecting 56-59
 objects in 3D space, transforming relative to
 camera position 75-77
 quaternion camera transformations, using
 78-83
 screen coordinates of 3D object, tracking
 72-74
SpringCam 41, 67
TargetCamera3D 41
camera.unproject() method 206
campos 411
character rigging 86
character rigging, in 3dsMax 86-89
clear() method 441
clipping artifacts 214
Clipping() class 331
ClippingDemo constructor 216
clouds
 creating 169, 170
Collada
 bones-based animation, controlling 90-94
Collada exporter 262
ColladaMax exporter 89
CollisionDemo.as 392
collisions, between objects
 detecting 392-395
ColorChooser component 133
ColorMatrixFilter 168
COMPLETE event 288
CompositeMaterial 38
Context3D class 437
CONTEXT3D_CREATE event 437
context3D object 437
continuousCurve() function 199
continuousCurve() method 199
controllable non-physical car
 creating 145-152
cool effects
 creating, with animated normal maps 403-
 408
coordinate2DToSphere() function 353
counter 313
createFromMatrix() method 79
createShape() method 74
createVertexBuffer() method 438

D

DAE (Collada) 262
default _tracker position 131
deltaTransformVector() method 48
deploySprites() method 342
DepthBitmapMaterial
 about 21
 used, for interpolating material color 21-23
DepthOfFieldSprite 50
DepthOfFieldSprite particles 106
depth of rendering
 defining 362-365
depth sorting 213
depth sorting problems
 solving, with Layers and Render Modes 221-
 226
DirectionalSprite 208
dirSprite.addDirectionalMaterial method 211
DisplacementMapFilter 168
DisplayObject 313
dispose() command 384
DofCache 52
 properties 52
DofCache settings 52
DOF effect
 creating 50-53
 creating, on Mesh Objects 53-56
dot() method 56
dot or inner product 56
dragTo.dragTo() method 354
dragTo() method 354
draw3DPerlin() function 171, 172
draw() method 136
drawTriangles() method 447
dropBall() method 293
dynamic text
 setting, with TextField3D 190-195

E

ElevationModifier 248
ElevationReader 387
ElevationReaderDemo 388
ElevationReaderDemo.as 388

EnvironmentMapmaterial 36
envMapAlpha 37
ErrorEvent handler 319
explodeAll() function 157
explodeAll() method 162
explode.apply() method 157, 162
Explode class 157
external assets
 accessing, in SWF 280-283
 storing, in SWF 280-283
external models
 importing, in Away3D Lite 332-334

F

FaceLink 112
FastTemplate 331
file barrel_empty_low.3ds 233
fillPPPathData() function 199
fillPPPathData() method 198
FLAR 289
FLARManager
 about 315
 download link 315
FLARToolkit 314
FlashDevelop
 about 355
 URL 355
Flash Media Server (FMS) 16
FLINT
 about 289
 used, for exploding particles 310-314
FlintDemo.as 310
FLINT library build
 URL 310
FMS 3.5 Development server version
 downloading 16
FMSVideoMaterial.mxml 17
Footstep mode 88
for each loop 163
FPS controller
 creating 42-46
FPS Controller
 working 47-49
FPSController.as class 362, 380
FPSController class 42, 49, 391
FPSDemo class 42

fragment program 444
FresnelPBMaterial 36, 37
Fresnel shaded sphere 37
Fresnel shader
 about 36
 exploring 36-38
FrustumClipping 214, 217, 365
FSphere 168

G

generateRandomExpl() method 162
genNormalMap() method 28
genNormalViaBump() function 30
geometry
 animating, with tween engines 105-112
 dragging, by un-projecting mouse coordinates 137-140
 exploding 154-158
 manipulating, in Away3D Lite 335-339
 optimizing, by welding vertices 371-375
geometry artifacts
 about 213
 fixing, with Frustum and NearField clipping 214-217
 removing, from intersecting objects 217-220
getColorsArr() 439
getMesh() function 293
getTimer method 447
GlowFilter 172
GTween 105

H

haloScaleFactor 182
haXe
 about 355
 URL 355
HeightMapModifier class 248
HeightMapModifier 408
HYPE
 URL 168

I

IK skeleton 89
image size consideration
 about 380-383

IndexBuffer3D 438
initAway3D() method 21, 428
initBulkLoader() method 287
initCamera() function 70
initCloudPlanes() function 171
initColorBuffer() function 439
initFaceLink() method 116
initFilters() method 168, 183
initFLAR() method 319
initFlint() function 312
initFloor() function 224
init() function 21, 437
initGeomBuffers() function 438
initGeometry() method 64, 82, 99, 135, 176, 192, 198, 224, 231, 276, 364, 390, 408
initGUI() function 192, 374
initGUI() method 193
initJigLib() method 293, 300
initLensFlair() function 182
initListeners() method 150, 199, 207
initMaterials() function 11, 35, 417
initModifiers() method 308
initPBMaterials() function 408
initPBMaterials() method 407
initSceneLights() function 35
initSceneLights() method 237
initSound() function 177
initSpheres() method 224, 226
initSprites() method 176
initStream() method 21
initTweens() method 110
initUI() method 105
inOnMouseClick() method 15
InputText() constructor 193
InputText field 190
intersectBoundingRadius() method 397
intersection tests
 line - plane 392
 ray - plane 392
 sphere - sphere 392
 triangle - triangle 392
IntersectRenderDemo.as 217
intersectVector 402
Invisibility method 379

J

JCar class 295
JigLibCarDemo.as 296
JigLibFlash
 about 290
 physical car, creating 295-299
 URL 290
 using, for setting up Away3D 290-294
JigLibSetupDemo.as 290

K

keyDownHandler() method 302
keyUpHandler() method 302
Kmz 262

L

LayerMaterial 38
LayersDemo.as 346
LayersDemo class 221
LayersDemo() constructor 224
lenses, cameras
 changing 60-64
lens flair effects
 about 179, 182
 creating 179-181
LensFlair tool 179
LibraryAway3d 308
libraryClips array 202
light maps
 about 248
 generating 248-252
loadBarrel() function 334
loadBytes() method 283
Loader3D class 333
loadGeometry() method 334
lockH property 11
lockW property 11
LODDemo.as 368
LOD(Level Of Detail) objects
 about 368
 working with 368-371

M

Main.hx class 356
main() method 356
material color
 interpolating, with DepthBitmapMaterial 21,
 23
materials
 about 7
 DepthBitmapMaterial 21
 DiffusePBMaterial 34
 FresnelPBMaterial 34
 MovieMaterial 8
 PhongMultiPassMaterial 35
 PhongPBMaterial 35
 PixelBender materials 30
 SpecularPBMaterial 34
 VideoMaterial 16
Math.atan2() method 340
matrices 440
Matrix 154
Matrix3D.decompose() 353
matrix operations
 compacting 447, 449
MaxZRenderDemo.as 363
Maya 261
MD2 262
MD2 animations
 about 94
 working with 95-100
mesh
 morphing interactively 141-144
Mesh DOF Engine 53
minElevation/maxElevation 390
MinimalComps
 about 112
 URL 190, 133, 304, 343
modelRot variable 152
models
 exporting, from 3DsMax/Maya/Blender 262
 exporting, from Prefab 228-232
 loading 276-280
 parsing 277-280
model structure
 painting interactively 133-136
Modifier() method 309
ModifierStack constructor 308
modType string 309
Molehill API
 about 431
 overview 431
Morpher class 104
Morpher mix() method 105
morphPattern sphere 105
MouseEvent3D 135
MouseEvent.MOUSE_DOWN 122
MouseEvent.MOUSE_UP 122
mouseX position 199
moveForward() method 152
MovieClip
 about 8
 using, for multiple materials 8-12
MovieMaterial
 about 8
 MovieClip ,controlling 12-15
MultiLoading.as 284
multiplyQuats() function 354

N

NearfieldClipping 214, 365
NinimalComps components 309
NormalAnimDemo.as 405
NormalBumpMaker utility
 about 29
 used, for creating normal maps 29, 30
NormalGenReady() event handler 29
normal map
 generating, Prefab used 232-237
NormalMapFilter 24
NormalMapGen 30
NormalMapGenerator 236
 used, for creating normal maps 26-29
NormalMapGenerator utility class 24
normal maps
 creating, Away3D NormalMapGenerator used
 26-29
 creating, NormalBumpMaker utility used 29,
 30
 creating, NVIDIA NormalMapFilter used 24-26
Number3D class 56

O

Obj 262
Object3D class 338
ObjectContainer3D 157, 202
object location
 object locationdetecting 56-59
object position
 animating on top of geometry, FaceLink used 112-117
objects
 excluding, from rendering 375-379
 morphing 100-105
objects in 3D space
 transforming, relative to camera position 75-77
ObjectsManipDemo.as 336
onAnimKeyFrameEnter() function 94
onBWDone event handler 21
onCheckBoxPress() 193
onContextReady function 437
onContextReady() method 442, 448
onEnterFrame() function 56, 74, 122, 151, 172, 293, 391, 442
onEnterFrameFunction() 105
onEnterFrame() method 70, 112, 187, 194, 200, 231, 444
onInit() block 334
onInit() function 333
onInit() method 331
onKeyDown() function 338
onKeyDown() method 47
onKeyUp() method 47
onKnobMove() method 117
onLoadComplete event handler 334
onLoadReady function 285
onMetaData() event handler 21
onMouse3DDown() event handler 339
onMouse3DDown() function 82
onMouse3dDown() handler 144, 353
onMouse3DMove() method 144
onMouse3dOut() handler 144
onMouse3dDown() handler 207
onMouse3dUp() handler 144
onMouse3Move() function 207
onMouseDown() event handler 47, 207
onMouseDown function 123

onMouseDown() method 122, 199
onMouseMove() event handler 136
onMouseMove() method 199
onMousePress() method 94
onMouseUp() function 200
onMouseUp handler 123
onObjMouseDown function 131
onPostRender() function 331, 339
onPostRender() method 354
onPreRender() function 331
onPressedMouseMove method 123
onRadioSelect() 309
onTextInput event handler 193
onTweenProgress() method 83
OpenCollada exporter 89
ownCanvas property 185

P

parse3ds() function 319
parse3ds() method 65
parseBarrel() function 334
parseDAE() function 93
parseData() function 288
parseGeometry() method 335
parseModel() function 150, 283
PathAlignModifier class 195
PathAnimator class 259
PathExtrusion 257
paths
 animating 253-259
 creating 253-259
performance 361
performance optimization
 depth of rendering, defining 362-365
 geometry, optimizing by welding vertices 371-375
 image size consideration 380-383
 important tips 383
 LOD(Level Of Detail) objects, working with 368-371
 objects, excluding from rendering 375-379
 scene poly count, restricting 365, 367
performance optimization, important tips
 Away3D-related tips 384
 Flash-related rules 383
perlinNoise map 40

perlinNoise() method 171
perlinNoise() parameters 172
perspectiveFieldOfViewRH() 440
PerspectiveMatrix3D class 432
PhongMultiPassMaterial 34
PhongPBMaterial
 about 237
 applying, to sphere 31-34
Photoshop
 normal maps, creating with NVIDIA Normal-
 MapFilter 24-26
physical car
 creating, with JigLib 295-299
 physical carworking 300-303
PixelBender3D 432, 439
PixelBender materials
 about 30
 applying 31
 DiffusePBMaterial 34
 FresnelPBMaterial 34
 PhongPBMaterial 31
 SpecularPBMaterial 34
Pixel Bender (PB) material group 8
PixelBender shaders
 Fresnel shader 36
PixologicZBrush 141
PointLight3D 237
PointLite3D 35
Point objects 171
PoliceCar.dae 295
PrefabUVMapDemo.as 240
Prefab
 about 227
 models, exporting from 228-232
 normal map, generating 232-237
 Terrain Generator editor 244
 UV map, editing 239-243
 UV mapping 239
primitive_sphere 157
processSound() function 177
processSound() method 177
project() method 66
PVS (potentially visible sets) technique 417
QuadrantRiddleFilter 220
Quaternion
 about 78
 Quaternionreference link 84
quaternion2Matrix() method 79
quaternion camera transformations
 using 78-83
Quaternion.slerp() 83

R

RandomDrift 314
RawMolehillBasic class 444
RawMolehillBasic constructor 436
Ray - AABS intersection test
 about 397
 working 398, 399
RayFaceTest() 401
ray.getIntersect() method 401
ray.intersectBoundingRadius() method 398
Ray Triangle test
 about 400
 working 400-403
realistic-looking debris 158-161
recompose() method 354
RectangleClipping() 214, 331
redrawBitMap() 429
redraw() function 172
renderBall() function 445
renderBall() method 442
render() command 384
Renderer.INTERSECTING_OBJECTS 220
renderer.sortObjects property 346
resetGeometry() method 308, 309
ResourceUtil class 433
Rich Internet Application (RIA) rules 153
Rich Internet Applications (RIA) 85
roll() method 303
rotateTo() method 303
rotating sphere
 creating, Molehill API used 432-447
rotational interactivity
 adding to Away3D primitive, Mouse move-
 ments used 120-124

Q

QuadrantRenderer 220 226

S

scene.addSprite() method 343
scene poly count
 restricting 365, 367
scenes (heavy scenes)
 running faster, BSP trees used 417-429
screen coordinates of 3D object
 tracking 72, 74
screen() function 74
seedBuilding() method 70
SegmentDrawDemo.as 204
segments, in 3D
 about 204
 drawing with 204-207
SelectiveRenderDemo.as 376
set.Inside initMath() 352
setupJCar() function 300
setupWheel() method 301
setVertexBufferAt() method 441
simpleUVMap.png bitmap 240
SimpleWalker class 70
 skeletal animation 86
SkinExtrude() class 391
Skin modifier 87
SkyBox6 412
SkyBox6-compatible texture
 creating 412
SkyBoxDemo program 416
skyboxes
 creating 409-412
 textures, creating 409-412
skybox setup
 about 413-415
 working 416
skybox UV mapping 412
skydome 409
slerp() method 79, 83
SortingDemo.as 344
sound
 visualizing 173-178
sourceObj container 163
spectrumArr array 177
specularMap property 35
SpecularMultiPassMaterial 34
Sphere primitive 168
SphericalLens 65

SphericalLens class 66
spherical linear interpolation method (SLERP)
 78
SpringCam
 about 67, 71
 damping 67
 mass 67
 setting up 68, 69
 stiffness 67
 working 70
Sprite 3D
 about 340
 using, for making 2D shapes appear 3D 340-343
Sprite3D 176
Sprite3D constructor 313
Sprite3DDemo 340
SpriterSession 226
spyObjRe.jpg image 90
Stage3D 437
Stardust 314
startDragThis() method 353
swapLens() method 65
SWF3DDemo class 202
SWF files 280
Swf.parse() method 202
SWFResourcesFile.as 281

T

TANGENT_SPACE 28
TangentToObjectMapper utility 35
Terragen
 about 409
 downloading 409
terrain
 creating 244-248
TerraSky 409
TextExtrusion 193
TextField3D 190
TextField3D text
 attaching, to user specified path 195-199
textureBdata object 171
TextureLoader 288
TextureLoaderQueue 288
timedFunction() 21
TimelineMax 112

TimerEvent.TIMER event 65
TraceEvent.TRACE_PROGRESS class 29
traceLevels() method 391
TrackballDemo.as 349
transformMarker() function 77
transformVector() method 78, 339
tween engines 105, 107, 108, 109, 110, 111, 112
Tweener 105
TweenMax 82, 105, 132

U

uneven surfaces
moving 387-391
Universal Description and Discovery Information. *See* **UDDI**
updateCar() method 151
updateSceneObjectsPos() function 168
updateSceneObjectsPos() method 168
UV map
editing, with Prefab 239-243
working 244
UVMappedCone class 242
UV mapping 239
UVWUnwrap modifier 239

V

Vector3D.normalize() method 58
VectorText.extractFont() method 192
VectorText utility class 198
Vertex animation store 86
VertexAnimator 93
VertexBuffer3D 438
vertexPosArr array 110
Vertex shader 442

VideoMaterial
about 16
VOD from FMS, streaming 16-21
virtual trackball
creating 349-354
VOD, from FMS
streaming 16-21

W

waitForLoad 283
walk() function 47
Walk mode 89
WaterMap 403
Wavefront 262
Wavefront.obj 228
Weld() class 374
WeldDemo.as 372
WireColorMaterial 207
workflow
maintaining, with AwayConnector 238, 239

Y

yaw() method 303

Z

ZoomFocusLens 65
Z-Sort algorithms 213
Z-Sorting
about 213
using QuadrantRenderer 226
Z-Sorting, Away3D Lite
layers used 346-349
managing, by automatic sorting 343-346



Thank you for buying Away3D 3.6 Cookbook

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.

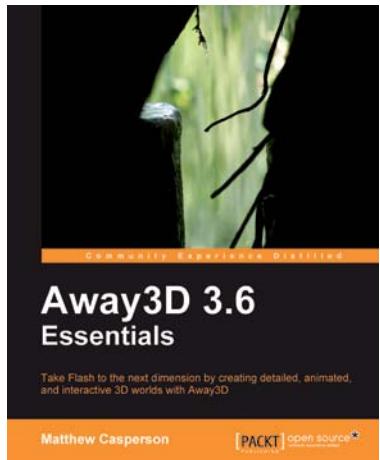
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

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.



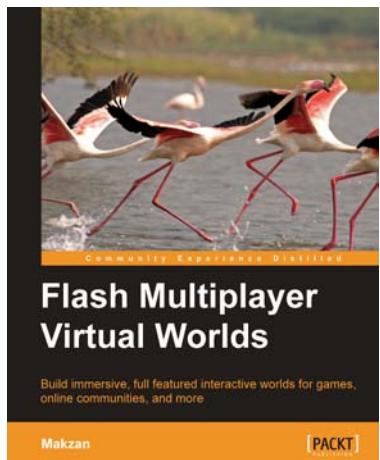
Away3D 3.6 Essentials

ISBN: 978-1-849512-06-0

Paperback: 400 pages

Take Flash to the next dimension by creating detailed, animated, and interactive 3D worlds with Away3D

1. Create stunning 3D environments with highly detailed textures
2. Animate and transform all types of 3D objects, including 3D Text
3. Eliminate the need for expensive hardware with proven Away3D optimization techniques, without compromising on visual appeal
4. Written in a practical and illustrative style, which will appeal to Away3D beginners and Flash developers alike



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

Please check www.PacktPub.com for information on our titles

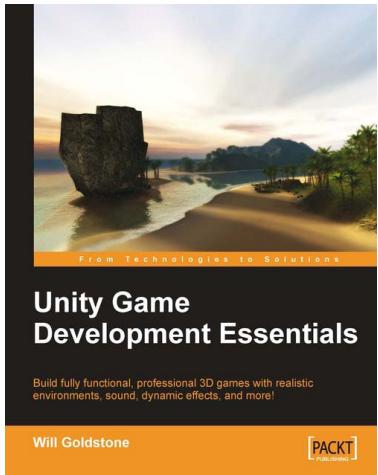


Flash Game Development by Example

ISBN: 978-1-849690-90-4 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



Unity Game Development Essentials

ISBN: 978-1-847198-18-1 Paperback: 316 pages

Build fully functional, professional 3D games with realistic environments, sound, dynamic effects, and more!

1. Kick start game development, and build ready-to-play 3D games with ease
2. Understand key concepts in game design including scripting, physics, instantiation, particle effects, and more
3. Test & optimize your game to perfection with essential tips-and-tricks
4. Written in clear, plain English, this book is packed with working examples and innovative ideas

Please check www.PacktPub.com for information on our titles