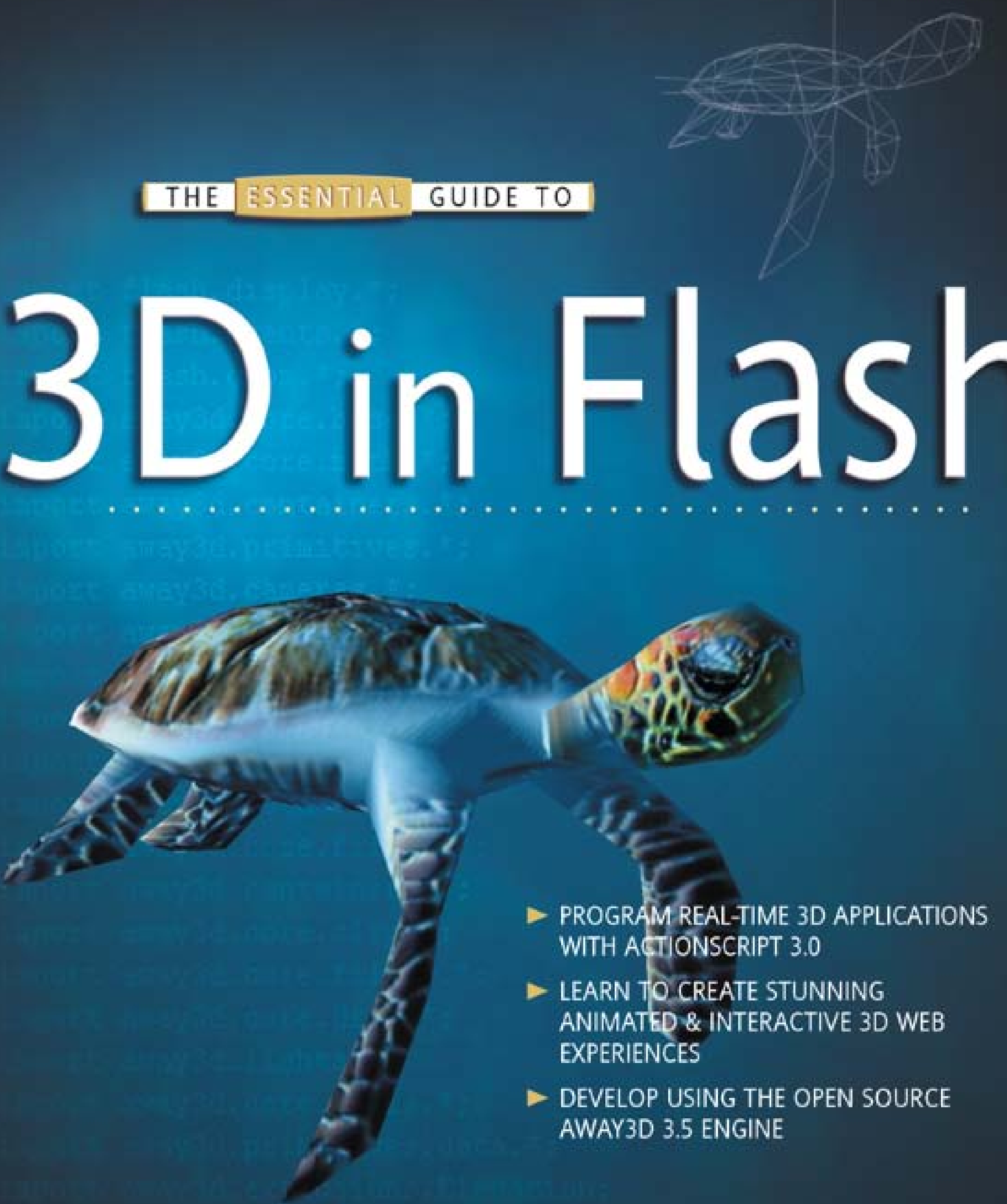
THE ESSENTIAL GUIDE TO

3D in Flash



► PROGRAM REAL-TIME 3D APPLICATIONS WITH ACTIONSCRIPT 3.0

► LEARN TO CREATE STUNNING ANIMATED & INTERACTIVE 3D WEB EXPERIENCES

► DEVELOP USING THE OPEN SOURCE AWAY3D 3.5 ENGINE

ROB BATEMAN AND RICHARD OLSSON

friendsof

WITH CONTRIBUTIONS FROM:

GREGORY CALDWELL, JENS CHRISTIAN BRYNILDSEN, PETER KAPELYAN, DAVID LENAERTS, AND ALEJANDRO SANTANDER

The Essential Guide to 3D in Flash

Rob Bateman & Richard Olsson

friendsof™

DESIGNER TO DESIGNER™

an Apress® company

The Essential Guide to 3D in Flash

ISBN-13 (pbk): 978-1-4302-2541-6

ISBN-13 (electronic): 978-1-4302-2542-6

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Credits

Publisher and President:
Paul Manning

Lead Editor:
Ben Renow-Clarke

Technical Reviewer:
Gregory Caldwell

Editorial Board:
Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Coordinating Editor:
Anne Collett

Copy Editor:
Heather Lang

Compositor:
Bronkella Publishing LLC

Indexer:
Toma Mulligan

Artist:
April Milne

Cover Designer:
Anna Ishchenko

Cover Art and Turtle Model:
Peter Kapelyan

Suzanne Monkey Model:
www.blender.org

Skeleton Model:
www.geocities.jp/oirahakobito2/,
oirahakobito@yahoo.co.jp

Contents at a Glance

Contents

About the Author

Rob Bateman is a web developer and community leader who has been involved in programming for over ten years. He specializes in content for the Flash platform and has always held a particular interest in 3D on the Web. In 2007, he cofounded the Away3D engine with Alexander Zadorozhny and has been leading the development of core features for the last two years.

Rob lives and works in London, UK where his production and consultancy company Away Media Ltd. provides expert services in the field of browser-based 3D content. A regular speaker on the international conference circuit, he is an active member of the Flash community and hosts frequent training programs teaching Away3D to web designers and developers.

His blog at `www.infiniteturtles.co.uk` provides further examples and musing on 3D in Flash, as well as information on upcoming appearances, training courses, and new Away3D releases.

Richard Olsson is a Swedish freelance Flash developer based in the city of Malmö and has been a member of the core Away3D development team since the end of 2008. He found his interest in programming at the early age of 12, writing BASIC on an old Commodore 64 that was passed on to him from a relative who wanted something more modern. After a short detour to 3D graphics, thinking he wanted to be a 3D modeler, he went back to programming, exploring C/C++ and OpenGL before ending up working with the Flash platform.

Richard is commissioned by several advertising agencies across Europe to assist with Flash 3D application development and has worked with some of the world's largest brands. He is also a frequent Flash instructor and a contributor to the Blender open source 3D suite.

On the Web, Richard himself, as well as his work and experiments, can be found at `www.richardolsson.se`.

About the Technical Reviewer

Gregory Caldwell is a software developer with over 15 years' experience specializing in web development, and over the last three years, he has focused on ActionScript and 3D graphics as part of the Away3D team. He lives in the UK and has worked in a range of industries with various technologies, developing software for online financial systems (using ASP.NET, C#, StoryServer) and real-time command and control software (SCADA, PLC, DCS), as well as providing bioinformatics analysis for the Human Genome Project. His web site (`www.geepers.co.uk`) features his latest experiments in 3D graphics.

Acknowledgments

Away3D is a non-commercial, open source project that relies on the free contributions of many designers and developers from within the Flash community. We would, therefore, like to extend our gratitude and admiration to everyone involved in the core development team. Your dedication and effort have made Away3D what it is today, and your experimentation, exploration, and infectious enthusiasm continue to drive development forward.

Another driving force has been the overwhelming support of the user base that has grown up around the project. It is quite heartening to see what people are willing to give back to the project, so we would like to thank all contributors who have helped out over the years filing bugs, offering patches, writing tutorials, and answering technical queries, as well as the larger community for producing some of the coolest looking web sites we've ever seen. The openness and collaborative atmosphere that you all encourage is simply a joy to work with.

For the production of this book, special thanks go to a number of individuals for their assistance, including Peter Kapelyan, Greg Caldwell, Jens Brynildsen, David Lenarts, and Alejandro Santander from the Away3D core development team for their various contributions, Pavel and Edie from Syerit Interactive for their modeling contributions, and Martin Jonasson for help with the setup instructions for FlashDevelop. Thanks and respect must also go to Blender for the free use of its software as well as the monkey model Suzanne, and to Powerflasher for its free FDT license program for open-source projects.

For acting as moral support and helping to coax this book into existence, we'd like to thank Caroline Lindqvist for code sample ideas, Anna and Maria for not repeatedly asking how the book is coming along, and Su for repeatedly asking how the book is coming along. The relief for completion will be mutual!

Finally, a special mention must go to the other cofounder of Away3D, Alexander Zarodozhny. His early code contributions have not only offered many technical firsts for solving 3D Flash problems but also provided a solid base on which to build—something invaluable in any code library. Although Alex is no longer involved in the project directly, his high standards have set a precedent for all team members working on today's new features, and his work continues to provide many designers and developers with the tools they need for creating unparalleled 3D Flash content.

Rob Bateman and Richard Olsson

Introduction

3D on the Web has been something of a perennial pipe dream since the creation of the Internet. As far back as 1994, talk of virtual realities and new ways of visualizing data in the 3D realm presupposed that 3D wasn't just a different method of presentation but an ultimate inevitability for the Web. Around the release of the first 3D web standard—VRML in 1998—the predictions and conjecture began to deflate, and by 2001, it was painfully clear that the utopian dream of flying through *Tron*-like datascapes was not to be, which was probably for the best.

Fast forward five years, and the early lessons of 3D on the Web had been well and truly learned, with several standards and third-party plug-ins attempting to reignite interest along the way. The overriding problem with many of these attempts related to the inherent compatibility problems associated with hardware-accelerated graphics, which were necessary given the personal computing power available at that time. These issues, combined with plug-in maintenance problems, sparse tool support, and counter-productive architecture, sent web professionals everywhere running to the relative safety of 2D alternatives such as HTML, JavaScript, and of course, Flash.

In late 2005, the first full-featured 3D engine for Flash came into being. Called Sandy, it was written in ActionScript 2.0, the available scripting language for Flash at the time. Because of the interpreted nature of ActionScript 2.0 and the somewhat limited capabilities of the graphics renderer in the Flash Player, the 3D output achievable with Sandy was quite basic and generally limited to simple scaling billboards (their use was known as the "postcards in space" approach to 3D). However, the release of Sandy represented the genesis point for real-time 3D in Flash, and the library's open source approach set the standard for future engines.

Following Sandy in late 2006, another open source library called Papervision3D was released for the Flash Player, this time turning the whole sad history of 3D web content on its head. It differed from Sandy in two fundamental ways: it provided a simpler approach to creating 3D content, and it was written in the much more powerful ActionScript 3.0 language, introduced in version 9 of the Flash Player. At this point, consensus began to swing in Flash's favor, with many people beginning to concede that the most important aspect of any 3D web format wasn't how many polygons it churned out but how accessible it was to both creators and audience.

Flash being the king of interactive content meant that, for the first time, production and use of 3D content was being handled from the point of view of interactive web professionals rather than 3D professionals. This shift, coupled with the immense ubiquity of the Flash Player, delivered a killer punch—the world of 3D web sites literally exploded. Designers and developers were happy to experiment with the easily implemented framework, and audiences were charmed by the new look now possible for web content. Papervision3D was a triumph of accessibility over standards; it quickly became the dominant 3D format on the Web and left the surrounding world of browser-based 3D with a lot of catching up to do.

Away3D began its life as a branch of the Papervision3D engine in 2007 but quickly began evolving in a direction intended for stability and ease of use. While Papervision3D had set the benchmark in terms of interactive potential, it was lacking what every other format requires in order to promote longevity—a standardized approach to content that promotes learning and aids future development. Over the ensuing months and years, Away3D provided one of the most consistent upgrade paths of any 3D engine for Flash, with its open source license allowing anyone to contribute a bug fix or feature enhancement.

As a testament to the success of Flash-based 3D, Adobe enhanced the 3D potential of Flash in their 2008 release of Flash Player 10 by including some 3D-specific features, with many of these intended for use by 3D engines. This move retained the accessible nature of 3D in Flash for web professionals, while moving Flash in a direction more suited to 3D production and more familiar to traditional 3D designers and developers.

The approach Away3D offers to the creation of web content should be familiar to anyone working from within the Flash realm. The entire library is constructed using the object-oriented approach enabled by ActionScript 3.0, with the arrangement of 3D objects, containers, and scenes matching the arrangement of Flash's native 2D objects in the display list. Classes are defined as sections of code with specific form and function, and these, in turn, are grouped into packages that hold different varieties of class type. As an example, the code used to construct a 3D cube is held within its own Cube class, and this is stored with other geometric shape definitions inside a package called primitives. The same is true for classes that define different types of camera, material, importer, animation, and so on. This organization helps simplify the learning process for anyone new to the engine, and it offers flexibility when extending the engine's capabilities by clarifying the points at which additions can be made. Extensibility is an extremely useful asset to any open source project, helping to drive contributions and ultimately accelerating the development of new features.

These days, a basic approach to 3D using perspective-projected display objects (which are known as 2.5D objects) is offered with the standard tools available in Flash Professional. This technique is great for creating simple "postcards in space" interfaces, but it has limited potential for any other type of 3D content. A 3D engine provides the necessary tools and interfaces to create a truly 3D experience, and Away3D offers one of the most powerful and easy-to-use frameworks around. As well as being free, open source software, Away3D has an active community providing free technical help to individuals via the official website, www.away3d.com, where you will also find regularly published tutorials, demos and showcases. This book has been created as an Away3D primer for everyone wishing to expand their Flash knowledge into the third dimension, or for that matter, everyone wishing to expand their 3D knowledge into the Flash dimension. We hope you enjoy the ride!

Layout conventions

To keep this book as clear and easy to follow as possible, the following text conventions are used throughout.

Important words or concepts are normally highlighted on the first appearance in **bold type**.

Code is presented in `fixed-width font`.

New or changed code is normally presented in **`bold fixed-width font`**.

Pseudo-code and variable input are written in *`italic fixed-width font`*.

Menu commands are written in the form **Menu ➤ Submenu ➤ Submenu**.

Where we want to draw your attention to something, We've highlighted it like this:

> *Ahem, don't say I didn't warn you.*

Sometimes code won't fit on a single line in a book. Where this happens, we use an arrow like this: ➥.

```
This is a very, very long section of code that should be written all on the same ➥
line without a break.
```

Chapter 1

Getting Started

Before the power of Away3D can be harnessed in a Flash project, the developer environment needs to be set up correctly. The exact setup procedure may vary based on your own workflows, but two steps are always present: obtaining the source code, then setting up your integrated development environment (IDE) for use with Away3D.

For the source code, decide whether you want to obtain the latest trunk version from the Google Code Subversion (SVN) repository, or download a release version ZIP file from `www.away3d.com`.

We will cover all of the major development environments in this chapter, so it's best to follow the setup relevant to your editor of choice and then move on to the next chapter, which describes how to build your first 3D scene in Flash.

Getting the Away3D library

Away3D is a source code library, consisting of 400 or so ActionScript 3.0 source files. These work just like the source files you would typically write yourself (except in this case you don't have to), and they need to be included in your project in much the same way.

The library source code is available to download in two different ways, and the method you choose depends on your experience and requirements. On one hand, the latest official release of the source is available as a simple ZIP archive. You can find this download by browsing to the `Downloads` section of `www.away3d.com`. However, to get the latest ongoing experimental features and bug fixes, you must check out the source code from the project's SVN repository. SVN is a version control system that makes it easier to keep your Away3D source code up to date. This download method is recommended if you want to take advantage of the latest fixes and new features being produced by the Away3D Team.

The working code repository for the Away3D project is hosted at Google Code and can be checked out like any other Google Code project. The project homepage at http://code.google.com/p/away3d describes the general process of obtaining the source using an SVN client. The step-by-step process varies slightly between operating systems and developer environments. Flash Builder and FDT (code editors for both Mac and Windows) both offer SVN client functionality through plug-ins, while other editors might not. If your editor of choice doesn't offer a built-in SVN client, TortoiseSVN for Windows and SyncroSVN or Versions for Mac OS X are good alternatives.

When downloading the Away3D library, it is important to choose the correct source files for your needs. There are currently two versions, one for use with Flash Player 9 and the other with Flash Player 10. Before downloading anything, you need to make a decision about the version of Flash you will be using. The minor differences in terms of functionality between the two libraries may impact your decision, but attempting to compile Away3D source code for the wrong player will result in compile time errors. The ZIP file download has two clearly labeled links, one for Flash 9 and the other for Flash 10. When downloading from the SVN repository, there are two directories to choose from in the trunk: `fp9` and `fp10`.

Regardless of which SVN client you use, please refer to its manual for how to use the checkout URL supplied on the Google Code project page to retrieve the project source files. If you're not familiar with SVN and code repositories, you should stick with the release download from `www.away3d.com`. Once you're feeling more confident, it's a simple matter to switch to the latest SVN release if you want to try that.

Setting up a project

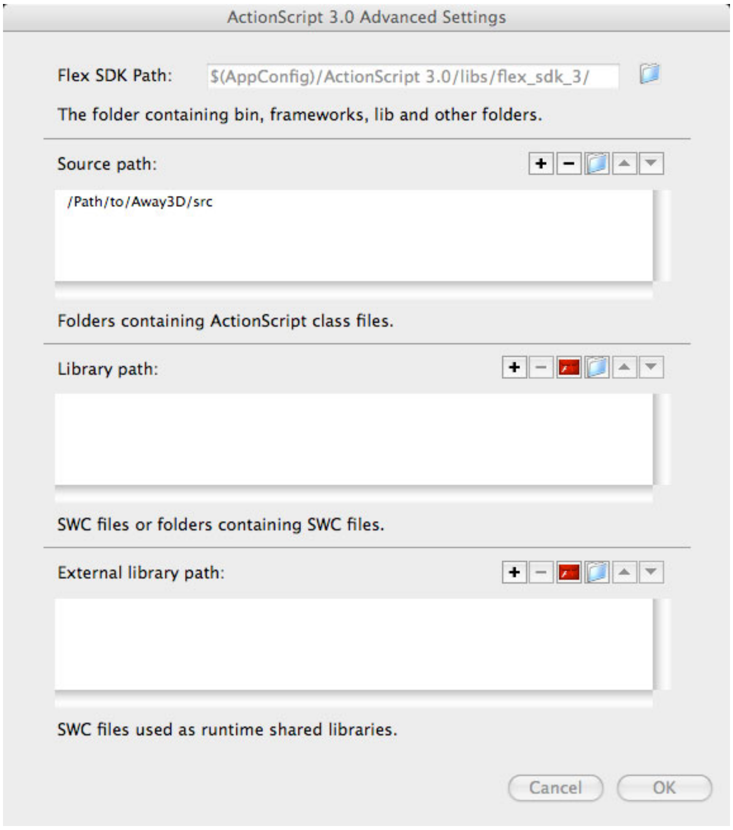
After the source files have been downloaded onto your hard drive, you need to set up your tool chain to work with Away3D. The general idea is to make the tools aware of where the Away3D library files are located, but the exact procedure differs from editor to editor.

Using Adobe Flash CS4/CS5

The Adobe Flash Professional IDE is very simple in the sense that it has no real concept of projects or linking between projects. Hence, setting up a project for use with Away3D consists of telling the IDE where the Away3D source folder is located on the file system by including it in the class path list. When publishing a SWF from Flash CS4, the directory that the FLA file is located in is used as the default class path. By simply putting the contents of the `src` folder from the Away3D distribution files in the same directory, the library will be made available for use in Flash.

An arguably better setup is to download the library, place it in a sensible location somewhere on your hard drive, and add that directory to the Flash class path list. Using this approach, the same installation of Away3D will be made available to all your Flash CS4/CS5 projects, now and in the future.

From the Flash `Preferences` pane, select ActionScript in the list and open the `ActionScript 3.0 Settings` (see Figure 1-1.) Click the folder icon in the `Source Path` section, and browse to the folder named `src` in the Away3D distribution. Away3D will now be available to all Flash projects you create.


ActionScript 3.0 Advanced Settings

Flex SDK Path: $(AppConfig)/ActionScript 3.0/libs/flex_sdk_3/

The folder containing bin, frameworks, lib and other folders.

Source path:

/Path/to/Away3D/src

Folders containing ActionScript class files.

Library path:

SWC files or folders containing SWC files.

External library path:

SWC files used as runtime shared libraries.

Figure 1-1. Adding Away3D to the source path in Flash CS4/CS5

Using Flash Builder

Adobe Flex Builder is more aimed at developers than its sister product Flash Professional. As is typical for a tool of this kind, it promotes order and control but has a slightly steeper learning curve.

Just as with Flash CS4/CS5, Away3D can be used in Flash Builder simply by placing the contents of the Away3D `src` directory into the source directory of your Flash Builder project. The preferred way, however, is to create a Flex library project to hold the Away3D sources, and then reference that project from your own. You can even check out the source for a library project directly from the Google Code SVN, using the integrated SVN client Subclipse, a plug-in for the Eclipse framework (on which Flash Builder is built). For now, let's walk through the method that doesn't use an SVN client.

From the `File` menu, create a new Flex library project by selecting `New` ➤ `Flex Library Project`. The `New Flex Library Project` dialog appears, which lets you enter a name for your project. It also asks for a hard drive location in which to store the project files. For the project name, "Away3D" makes a lot of sense, and for project location, the default folder is fine. Clicking the `Finish` button will create a project folder containing the standard files a Flex library project needs, including empty `bin` and `src` folders.

The next step is to place the Away3D source files into the newly created project. To do this, you can drag the contents of the `src` folder in the Away3D distribution from your file browser into the `src` folder of the project. Next, open the project properties panel by right-clicking the project folder in the Flex Navigator panel on the left and selecting `Properties`, or using the `Project` menu in the menu bar. Under `Flex Library Build Path`, make sure all classes are included in the library (see Figure 1-2.)

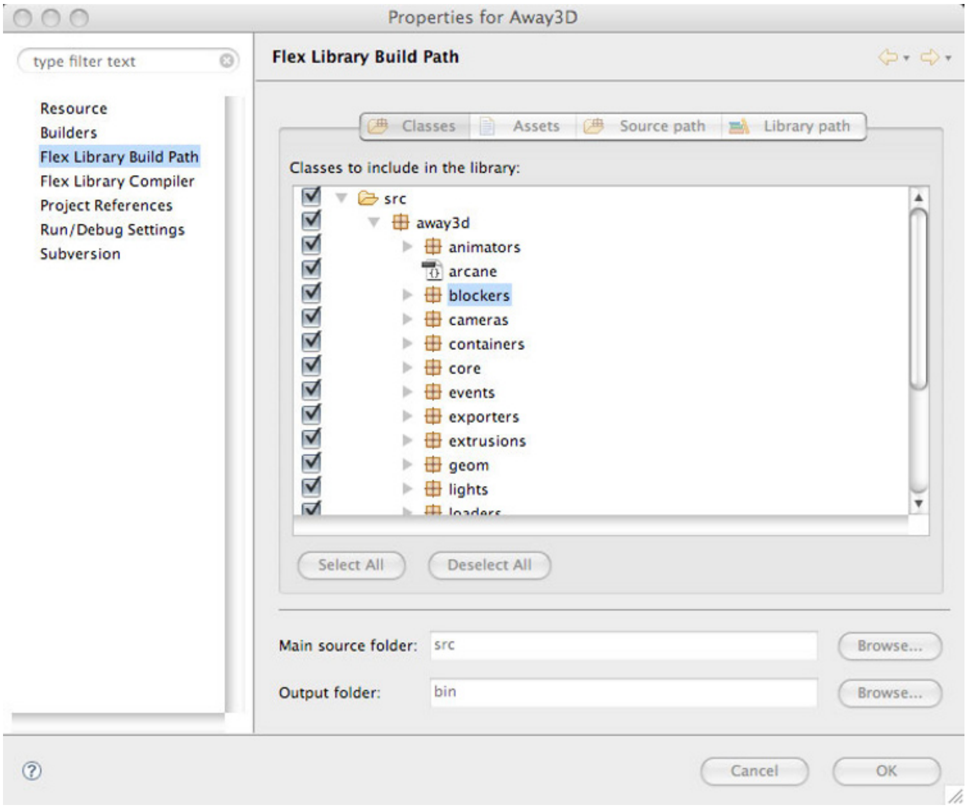
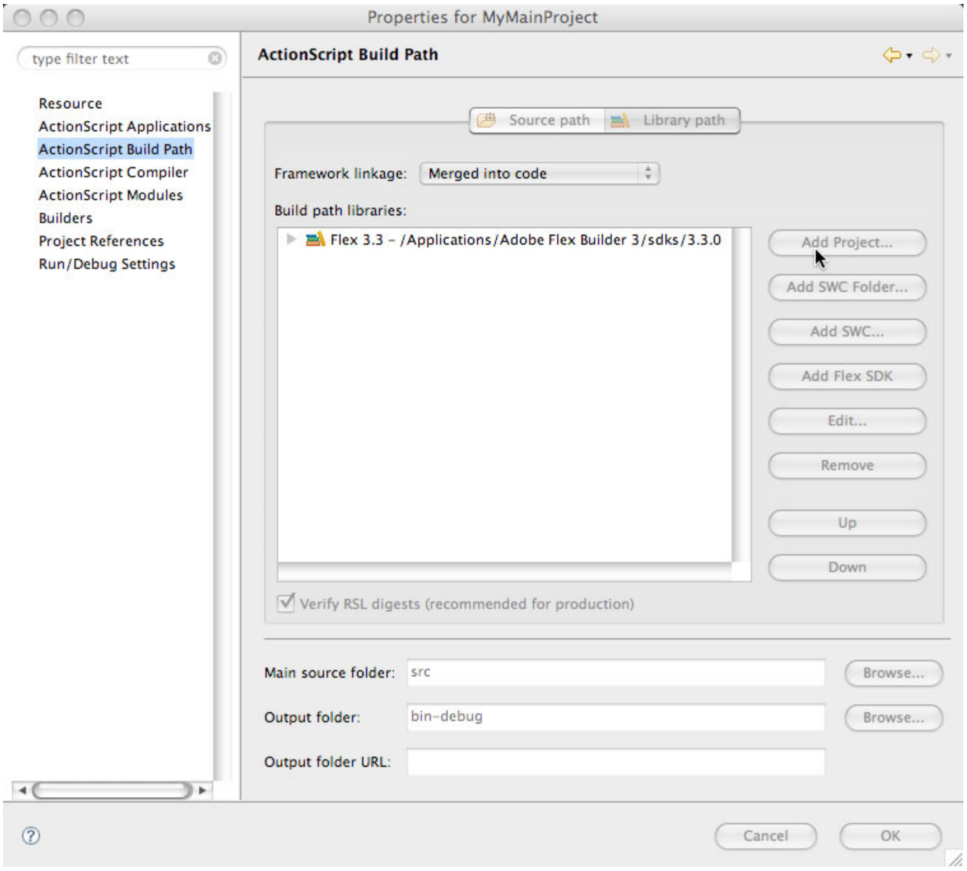


Figure 1-2. All classes included in the Away3D Flex Library project

The Away3D library project can now be referenced from any other project in which you wish to use Away3D functionality. To enable this, open the project properties dialog for that project, and under `Build Path`, in the `Library Path` tab, add a reference to the Away3D project using the `Add Project` button (see Figure 1-3).


Properties for MyMainProject

ActionScript Build Path

Framework linkage: Merged into code

Build path libraries:

Flex 3.3 – /Applications/Adobe Flex Builder 3/sdks/3.3.0

Verify RSL digests (recommended for production)

Main source folder: src

Output folder: bin-debug

Output folder URL:

Figure 1-3. Adding a reference to the Away3D project from another project

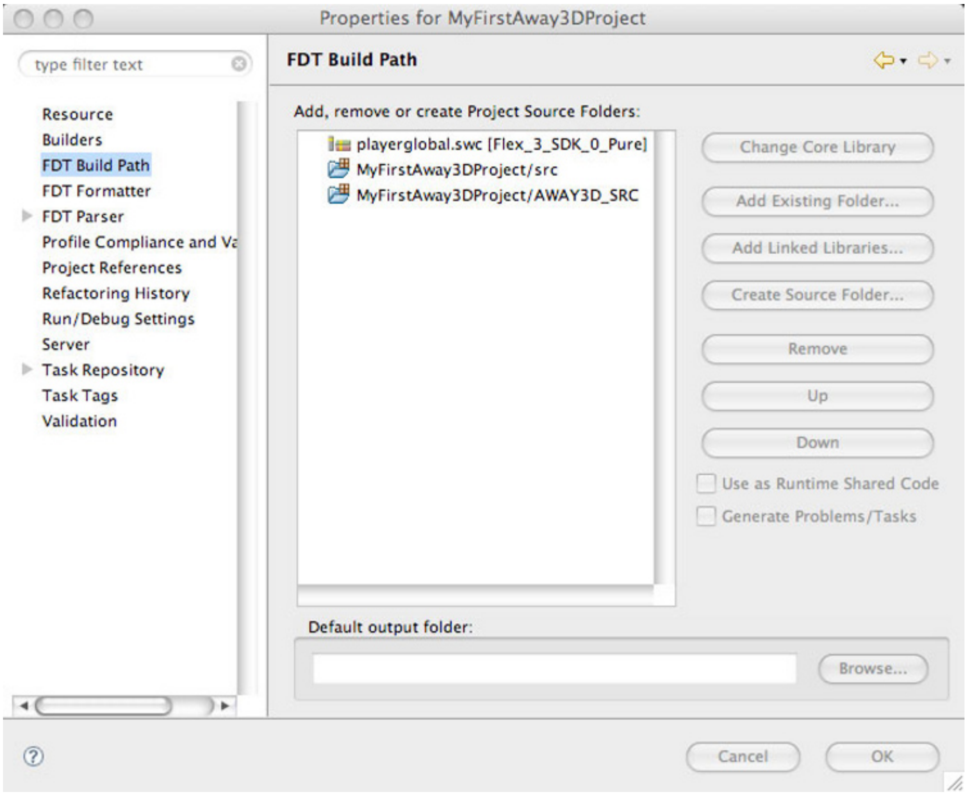
Using FDT

FDT is based on the Eclipse IDE just like Flash Builder, but unlike Flash Builder, it makes no distinction between library projects and regular projects. However, you can still link projects in the same way, as illustrated in the walkthrough in this section.

Create an FDT project for holding the Away3D source files in exactly the same way as a new library project in Flash Builder. Then, using the `New Project` wizard in FDT, create a new Flash project to contain the project that will use the library. You are prompted to enter a name for the new project, and select a language version and compiler SDK. In the second step of the `New Flash Project` wizard, you have the opportunity to add linked libraries. This is also possible once the project is created by going to the project properties dialog (see Figure 4-1), which is accessed by right-clicking the project's folder in the `Flash Explorer` panel on the left and selecting `Properties`, or choosing `Properties` from the `Project` menu in the menu bar.

Clicking the `Add Linked Libraries` button opens the `New Linked Libraries` dialog. Here, click `Add` to open a list of linked libraries that are available. If this is the first time you have done so, Away3D is nowhere to be seen, so you must first create it by clicking the New button. Enter a name (e.g., **AWAY3D_SRC**) for the library path, and browse to the src directory of the Away3D project. Click OK, select the newly created library link, and press OK again. You should see the name (e.g., AWAY3D_SRC) in the list of linked library source folders, which means that you'll be able to use Away3D in your project (see Figure 1-4).

While this process might seem a bit long-winded, it does get better. The AWAY3D_SRC variable will, from here on, be available in every new project you create, allowing you to skip the last couple of steps in future project setups.



Add, remove or create Project Source Folders:

playerglobal.swc [Flex_3_SDK_0_Pure]
MyFirstAway3DProject/src
MyFirstAway3DProject/AWAY3D_SRC

Default output folder:

Figure 1-4. The AWAY3D_SRC path variable added to project linked libraries

Using FlashDevelop

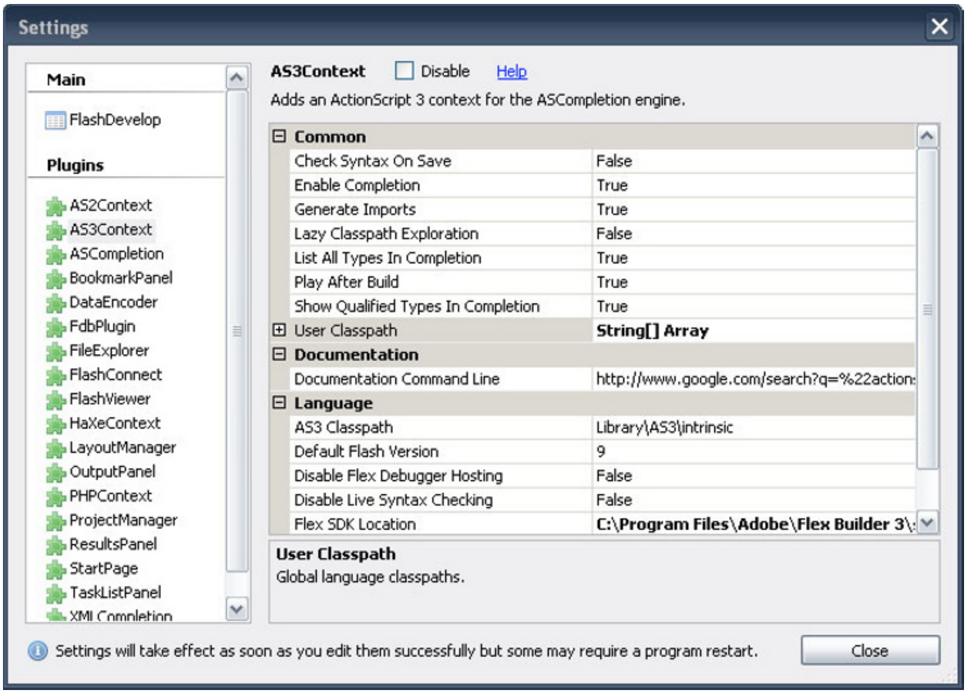
FlashDevelop is another great tool for Flash development. At present, it is only possible to install on a Windows-based system. The simplest way to enable Away3D functionality in FlashDevelop is to add the Away3D source code to the class path used by the compiler (although as with any other editor, it is also possible to simply keep the Away3D source files in the same directory as your project classes if that is your preference).

Just as with Flash Professional, the class path can be set both globally and locally for each individual project. A global class path setting is generally considered the best way, as it means you only ever have to set it once.

First, place the Away3D source code in a location on your hard drive that can be easily retrieved. Using Windows Explorer, make a note of the file path to this location (or copy it to the clipboard) from the Explorer address bar.

Next, open FlashDevelop, and from the `Tools` menu, select `Program Settings`. The `Settings` dialog appears, with all of FlashDevelop's configuration attributes. From the left-hand side category bar, select `AS3 Context` (see Figure 1-5). In the list on the right is one attribute called `User Classpath`, with the value said to be `String[] Array`. Clicking this value brings up a dialog that lets you put in any number of paths to external libraries.

On a blank line, type (or paste) the file path to the Away3D source files on your hard drive. Click `Close` to close the dialog, and you're done. All subsequent projects created in FlashDevelop will be able to reference Away3D classes, making Away3D functionality available to all your applications.



AS3Context ☐ Disable Help

Adds an ActionScript 3 context for the ASCompletion engine.

⊟ **Common**	
Check Syntax On Save	False
Enable Completion	True
Generate Imports	True
Lazy Classpath Exploration	False
List All Types In Completion	True
Play After Build	True
Show Qualified Types In Completion	True
⊞ User Classpath	**String[] Array**
⊟ **Documentation**	
Documentation Command Line	http://www.google.com/search?q=%22action
⊟ **Language**	
AS3 Classpath	Library\AS3\intrinsic
Default Flash Version	9
Disable Flex Debugger Hosting	False
Disable Live Syntax Checking	False
Flex SDK Location	**C:\Program Files\Adobe\Flex Builder 3**

User Classpath
Global language classpaths.

Settings will take effect as soon as you edit them successfully but some may require a program restart.

Figure 1-5. The FlashDevelop AS3Context settings panel, where User Classpath can be configured

Open source workflow using the Flex SDK and Makefiles

With the Flex SDK, Adobe provides a completely free and open source toolset for developing Flash and Flex applications. Away3D works perfectly well with this type of workflow, allowing you to create engaging 3D productions in Flash with zero financial outlay.

Makefiles are an excellent way of creating batch compile scripts in this type of workflow, as GNU Make is open source and comes preinstalled with Linux and Mac OS X. For Windows, a selection of free utilities for Makefiles are available, but the general approach outlined here could just as easily be used with DOS BAT files in a Windows environment.

The Flex SDK, which contains the command-line ActionScript 3 compiler, can be downloaded from `http://opensource.adobe.com`. Place the contents of the SDK distribution anywhere on your hard drive and note the location.

Next, set up a simple project directory structure. Create a folder for your project anywhere on your hard drive. This will be your project root directory, and within it, you need to create a directory named `src` and one named `lib-src`. Your source files will go in the `src` directory, while external libraries (such as Away3D) can be put in the separate `lib-src`. This helps prevent clutter in the `src` directory, where you will be working most of the time. Move the contents of the `src` directory in the Away3D distribution files into `lib-src`.

The last step in setting up the project is to create the Makefile. Entire books have been devoted to GNU Make and the Makefile syntax, so to keep things simple, you can use the template in the following code sample to achieve what we're after. Make sure that the value of the `MXMLC` variable on line 1 reflects the path to the location where you placed the Flex SDK compiler on your hard drive, and that the `APP` and `OUT` variables in the Makefile (lines 3–4) have values that make sense for your project. The template should be saved in the root directory of your project as a file called simply `Makefile`.

```
MXMLC=/Path/to/flex-sdk-4.0/bin/mxmlc
SRC=src lib-src
APP=src/MyAway3DApp.as
OUT=out/MyAway3DApp.swf

ARGS=-sp $(SRC) -o $(OUT)

all:
  $(MXMLC) $(ARGS) -file-specs $(APP)
```

In the `src` folder, create an ActionScript application file with a name that reflects the value of the Makefile `APP` variable, and then type **make** in a terminal window after navigating to the project's root directory. The application will be compiled, and a directory named `out` will be created, containing the compiled SWF file.

Summary

This chapter has described in brief how to obtain the Away3D library source code, either from the `Downloads` page at `www.away3d.com` or using the Away3D SVN repository hosted on Google Code. Furthermore, it has walked through the process of setting up the most common Flash development environments for use with Away3D. In the next chapter, we will go step by step through a basic Away3D project, showing how 3D objects can be created, lit, and rendered, as well as how to move and interact with 3D objects.

Chapter 2

Creating Your First 3D Project

It is time to get down to business. A project has been set up in your favorite editor, and you are ready to create some nice 3D visuals in Flash. One hour, and some 60 lines of code from now, you'll be watching and interacting with your first Away3D SWF!

Starting up the engine

After having worked with Away3D for a while, you'll notice how most projects start out in very much the same way. The basic Away3D objects are created, always including a `View3D` object, usually within what's called the **document class**, the main entry point for your application. While it is also possible to write code on the timeline in Flash Professional, the document class approach will be used throughout this book.

As with all graphics in Flash, Away3D needs to *render* the image produced from the objects contained in its hierarchy. Unlike the native display list hierarchy in Flash, the rendering process has to be triggered manually and is managed by the **view**, represented in Away3D by the `View3D` class. In a typical Away3D project, the view is the place to start, and a very basic application needs no more than the following lines of code:

```
package
{
  import away3d.containers.View3D;
  import flash.display.Sprite;
  import flash.events.Event;

  public class MyFirstApp extends Sprite
  {
    private var view : View3D;

    public function MyFirstApp()
    {
      view = new View3D();
      view.x = 275;
      view.y = 200;
      addChild(_view);
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function onEnterFrame(ev : Event) : void
    {
      view.render();
    }
  }
}
```

In the preceding code, the view is created, positioned, and added to the stage on lines 13–16, shown in bold. The `View3D` class extends `Sprite`, which allows it to be added to the display list of the Flash movie so that the rendered contents can be seen. The (x, y) position of the view represents the **vanishing point**, which in 3D terms means the position in a perspective drawing at which parallel lines appear to converge. To produce natural-looking 3D images, it is best for the vanishing point of the view to be positioned at the center of the Flash movie.

Next, an event listener is created for rendering the view on each frame. The rendering process occurs on line 22 inside the event handler, also shown in bold.

To compile the `MyFirstApp` example into an SWF, it is necessary to first create an ActionScript class file (a text file with the name `MyFirstApp` and extension `.as`) and paste in the above code. Flash Professional users need to place this file in the class path of the FLA being used. (i.e., the same directory location), and make sure the name of the class is specified as the document class in the FLA properties panel. Other ActionScript editors (Flash Builder, FDT, FlashDevelop, etc.) compile into an SWF using a document class by default and only require that you identify the class file as such in your project before compiling.

Compiling the `MyFirstApp` example, you will find that all that shows up is a blank window. This is to be expected, so don't worry! Once you have made it this far, we can start adding some 3D objects to be rendered.

Adding 3D objects to the scene

The root container for 3D objects is generally known as the **scene** and is represented by the `Scene3D` class in Away3D. Working with the scene closely resembles the approaches used when working with the Flash display list in ActionScript, by using methods such as `addChild()` and `removeChild()`. These will be covered in more depth in the next chapter.

When the view is created in the `MyFirstApp` example, an empty scene is created for the view to use by default. It can be accessed through the `scene` property on the `View3D` class instance. For example, creating a `Cube` object and adding it to the scene can be accomplished by adding the following code to the end of the constructor method in `MyFirstApp`:

```
var cube : Cube = new Cube();
_view.scene.addChild(cube);
```

The cube is an example of a **3D primitive** object (covered in more detail in Chapter 4) and needs to be imported from the `away3d.primitives` package. Add this to the other `import` statements in the example to make sure the code will compile without errors.

```
import away3d.primitives.Cube;
```

Recompiling our `MyFirstApp` example will display a square on the screen made up from two triangles. This is actually our cube rendered head-on, with a random solid color and black triangle outlines. In Away3D, as with many 3D engines, the majority of 3D objects are composed from tessellating triangles. The color used for the rendered output of the cube is the signature appearance of the **default material**, `WireColorMaterial`. It is useful for debugging, as it clearly shows both the edges and triangles making up the cube primitive.

Materials define the appearance of a 3D object when it is drawn to screen. We can apply a different material to the cube when we create it to alter its visual appearance. One commonly used type of material in Away3D is `BitmapMaterial`. Using this class will wrap a bitmap image (supplied as a `BitmapData` object) around the 3D object to which it is applied, a process known as **texture mapping**. Replacing the first line with the following code will use a `BitmapData` object containing Perlin noise as the material in place of the default `WireColorMaterial`.

```
var bmp : BitmapData = new BitmapData(200,200);
bmp.perlinNoise(200, 200, 2, Math.random(), true, true);

var mat : BitmapMaterial = new BitmapMaterial(bmp);
var cube : Cube= new Cube({ material: mat });
_view.scene.addChild(cube);
```

Another two `import` statement are needed here, one for the `BitmapMaterial` class located inside the package `away3d.materials`, and one for the native `BitmapData` class located inside the package `flash.display`. Add the following lines to the rest of the `import` statements at the top of the document class.

```
import away3d.materials.BitmapMaterial;
import flash.display.BitmapData;
```

Recompiling the `MyFirstApp` example will display a square much like the previous one but this time, textured with multicolored noise (see Figure 2-1).

Figure 2-1. A cube with a Perlin noise bitmap used as its material, shown straight from the front

Understanding constructors and initialization objects

As you may have noticed, the Cube instantiation in the MyFirstApp example uses an **object literal** (surrounded by curly braces) as a constructor argument. You may think this is some wicked Away3D trick, and depending on your views toward untyped objects, you could be deemed correct!

Object literals can be passed into the constructors of most Away3D objects and are referred to as **initialization objects** (**init objects** for short). They provide a quick and compact way of setting default initial values for the properties of the object being created. Using init objects can be a neat way of working for the experienced user, requiring less typing and fewer lines of code to setup Away3D objects in the desired manner. However, the approach is not so useful for the beginner because of the untyped nature of the object literal. This removes useful type-checking features in your ActionScript editor such as code completion, which you may prefer to have enabled when getting to grips with a new code library.

If you're feeling discouraged, note that any property set using the init object (e.g., the material attribute used when creating the cube in the previous code) can also be set as a public property on the object after instantiation (e.g., cube.material). Hence, the following lines of code for creating a cube achieve exactly the same result as the init object method, while also allowing your ActionScript editor to help you with code completion (if that feature is supported):

```
var cube : Cube = new Cube();
cube.material = mat;
```

Whether you choose to use init objects in your code is entirely up to you. We find that this decision usually comes down to personal preference. Because of the code completion advantages, which can be of great assistance when learning a new library, we will be sticking with the easier to follow public property method for the remainder of the book.

Lighting the scene

Using the same approach to creating 3D primitives as our cube object described previously, we will now add a sphere primitive to the scene and create more of an atmosphere by adding a light source!

By appending the following lines of code to the document class constructor of our MyFirstApp example (below the code for cube creation), a sphere using a material composed of the same Perlin noise bitmap will be created and added to the scene:

```
var sphere : Sphere = new Sphere();
sphere.segmentsW = 32;
sphere.segmentsH = 32;
sphere.material = new BitmapMaterial(bmp);
sphere.x = 200;
_view.scene.addChild(sphere);
```

Just like the Cube class, the Sphere class needs to be imported from the primitives package, by adding an import statement in the usual place.

```
import away3d.primitives.Sphere;
```

The sphere is placed 200 units to the right of the cube and uses a material with exactly the same appearance. Recompiling the example will display a large circle to the right of the square, textured with the same multicolored noise.

At this point, you may be wondering whether the objects on the stage look right for 3D objects; after all, circles and squares are pretty two-dimensional! This is because the difference between a sphere and a circle on a 2D screen (the monitor you are using to view the Flash movie) is depth perception. You need to fool the viewers' eyes into thinking they are viewing a 3D object, and one way of doing that is with a light source. If a light source is placed above the sphere, it will be possible to perceive its depth from the shading that the light applies to the material—darker on the lower hemisphere and lighter on the upper hemisphere. The code used in this experiment so far doesn't use shading, which is why the shapes we have created appear quite flat.

In Away3D, the choice of material determines whether any shading from a light source will be rendered. The simple `BitmapMaterial` used for both the cube and the sphere has no shading capabilities, hence it will always disregard light sources. To enable shading effects, we need to choose a material that supports them, the simplest of which is the `WhiteShadingBitmapMaterial` class. Modifying line 4 shown in bold in the preceding lines of code, where the `Sphere`'s material is set, we can enable shading by replacing the line with the following code. Don't forget to import the `WhiteShadingBitmapMaterial` class from the `away3d.materials` package.

```
sphere.material = new WhiteShadingBitmapMaterial(bmp);
```

Recompiling the application after this alteration gives you your next surprise—the flat circle hasn't miraculously turned into a 3D sphere and instead has lost all color and turned into a black hole in our scene. The explanation for this is simple: there are no light sources in the scene, so any object with a shading-enabled material will be cast entirely in darkness.

Adding a light source will solve this, and luckily adding light is just as simple as adding any other 3D object to the scene. The following lines of code will do the trick:

```
var light : PointLight3D = new PointLight3D();
light.y = 500;
_view.scene.addLight(light);
```

Line 1 of the preceding code creates a new `PointLight3D` object (which is a class imported from the `away3d.lights` package), and represents a single point from which light is emitted in all directions. Line 2 moves our light source's position 500 units up, and line 3 adds it to the scene. When we recompile our `MyFirstApp` example, the sphere will have lit up, and the light and shade on its surface reveal the 3D nature of the primitive `Sphere` object (see Figure 2-2).

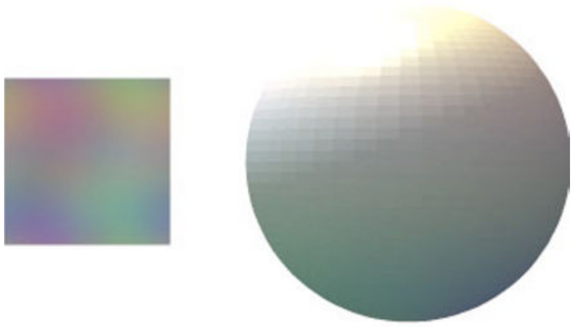


Figure 2-2. A shaded sphere added to the scene, with a light source straight above the (unshaded) cube

Animating objects in 3D

Up until now, rendering the view on every frame has made very little sense. There is no reason to force the CPU to calculate and render a 3D scene 25 times a second when each frame looks exactly like the previous one. Doing so isn't necessary until animation is introduced into the project.

> *The automatic caching system in Away3D ensures that repeatedly rendering a view for which the scene has not changed doesn't waste CPU time with unnecessary redraws. Each time the `render()` method is called, only areas that have changed in the view are updated.*

By incrementally adding to the rotation of the sphere in our scene on each frame, we can perform a rotation on the object that mimics the way the Earth revolves on its axis. But before we can do so, a permanent reference to the sphere object must be stored in the document class so that its rotation property can be updated in the `onEnterFrame` handler. To achieve this, we move the `sphere` variable declaration to the class scope and make it a private variable. As we do so, we prepend its name with an underscore. This is a convention commonly used in object-oriented programming for a private class variable, and is followed throughout the code examples in this book. Because of this change, make sure you add the underscore to all sphere variable references.

```
public class MyFirstApp extends Sprite
{
  private var sphere : _Sphere;
  ...
```

In the `onEnterFrame` handler function, adding 5 to the sphere's vertical (Y) axis rotation will rotate the sphere in the manner we require.

```
private function onEnterFrame(ev : Event) : void
{
  _sphere.rotationY += 5;
  _view.render();
}
```

Recompiling the `MyFirstApp` example will show the sphere slowly rotating around its Y axis. This very simple animation technique in 3D allows many possibilities, and Away3D provides many versatile 3D animation methods, some of which are covered in more depth in Chapter 9.

Enabling interactivity in 3D

Creating a button, perhaps the most basic form of mouse interaction in Flash, is just as simple to achieve in Away3D as it is in a regular Flash movie. It involves adding an event listener to a 3D object that reacts to events of the special type `MouseEvent3D`. The `MouseEvent3D` class resides alongside all custom events for Away3D in the `away3d.events` package.

To illustrate how simple this process can be, we'll set up a listener to make the cube in our scene rotate randomly when it detects a mouse click. The first step toward achieving this is to create the event handler function.

```
private function onClickCube(ev : MouseEvent3D) : void
{
GenericTweener.tween(ev.currentTarget, 1, {
    rotationX: Math.random()*360,
    rotationY: Math.random()*360,
    rotationZ: Math.random()*360 }
  );
}
```

The preceding code handles `MouseEvent3D` events, such as mouse clicks. It uses a small tweening class called `GenericTweener` to smoothly rotate all the cube's axes to random values. The `GenericTweener` class is part of the source files package that can be downloaded from www.friendsofed.com, and resides in the flash3dbook.common package. We will use it further in chapter 9.

Using the regular `addEventListener()` method on the cube object, the `onClickCube` function can be defined as a handler for the `MouseEvent3D.MOUSE_UP` event.

```
cube.addEventListener(MouseEvent3D.MOUSE_UP, onClickCube);
```

Inserting this line directly under the code that deals with our cube creation will complete the process of adding button interactivity to the cube. Recompile our `MyFirstApp` example, and try clicking the cube to see it spin!

Summary

This chapter took you through creating a simple scene in less than an hour, with all the basic features that you would usually want to use in a 3D project: primitives, texturing, lights, shading, animation, and interactivity. All of the features in each of these categories are described in greater detail later in this book.

The complete code for the MyFirstApp example created in this chapter follows:

```
package
{
  import flash3dbook.common.GenericTweener;

  import away3d.events.MouseEvent3D;

  import away3d.containers.View3D;
  import away3d.lights.PointLight3D;
  import away3d.materials.BitmapMaterial;
  import away3d.materials.WhiteShadingBitmapMaterial;
  import away3d.primitives.Cube;
  import away3d.primitives.Sphere;

  import flash.display.BitmapData;
  import flash.display.Sprite;
  import flash.events.Event;

  public class MyFirstApp extends Sprite
  {

    private var _view : View3D;
    private var _sphere : Sphere;

    public function MyFirstApp()
    {

      _view = new View3D();
      _view.x = 275;
      _view.y = 200;
      addChild(view);
      addEventListener(Event.ENTER_FRAME, onEnterFrame);

      var bmp : BitmapData = new BitmapData(200, 200);
      bmp.perlinNoise(200, 200, 2, Math.random(), true, true);

      var mat : BitmapMaterial = new BitmapMaterial(bmp);

      var cube : Cube = new Cube({ material: mat });
      cube.addEventListener(MouseEvent3D.MOUSE_UP, onClickCube);
      _view.scene.addChild(cube);

      _sphere = new Sphere();
      _sphere.segmentsW = 32;
      _sphere.segmentsH = 32;
      _sphere.material = new WhiteShadingBitmapMaterial(bmp);
      _sphere.x = 200;
      _view.scene.addChild(sphere);

      var light : PointLight3D = new PointLight3D();
      light.y = 500;
      _view.scene.addLight(light);
    }


    private function onEnterFrame(ev : Event) : void
    {
      _sphere.rotationY += 5;

      _view.render();
    }


    private function onClickCube(ev : MouseEvent3D) : void
    {
      GenericTweener.tween(ev.currentTarget, 1, {
        rotationX: Math.random() * 360,
        rotationY: Math.random() * 360,
        rotationZ: Math.random() * 360
      });
    }
  }
}
```

In the next chapter, we will be exploring the view and scene in more detail, looking at how the camera object defines the relationship between them, and exploring more complex ways of grouping, moving, and transforming 3D objects.

Chapter 3

The View, Scene, and Camera

In the last chapter, you saw how it is possible to create an Away3D project by simply adding a new `View3D` object to the display list of the document class. More generally, three building blocks make up the foundation of a 3D environment: the view, the scene, and the camera. The latter two objects are created automatically by the view on instantiation but can be overwritten in cases where a greater degree of control is required.

Ways of manipulating these elements range from simple adjustable properties on each object to advanced configuration classes. A large number of core features are covered in this chapter, and learning as many of these as possible will improve the adaptability of your 3D coding, enabling a wide variety of tasks.

Understanding the basics

It's important to have a basic knowledge of the roles performed by the view, scene, and camera in your 3D project, as they are always present even though you may not be explicitly creating them yourself. A brief description of each follows.

View

The **view** holds the 2D representation of the 3D scene being rendered and links the virtual 3D world of your project to the 2D world that is necessary for representation on a computer screen. It is sometimes talked about as the **viewport**, which refers to the rectangular area that contains the visual output of a 3D scene. A good analogy of the viewport is that of a window, cropping the scenery outside, as you look through it from inside a room.

Often, the view is the only Away3D object that needs to be explicitly created. As the `View3D` constructor method is called, `Scene3D` and `Camera3D` objects are created by default as `scene` and `camera` properties on the view instance (as you saw in the previous chapter).

Scene

The **scene** in Away3D is represented by the `Scene3D` class located in the `away3d.containers` package. It acts as a 3D version of the native stage object in Flash, representing a top-level 3D container. Any 3D objects to be included in the rendering process are attached inside the `Scene3D` class.

> *From Flash 10 onward, native display objects in Flash possess a 3D nature almost identical to the 3D nature of scene objects in Away3D. This leads to all display objects (`Sprite`, `MovieClip`, etc.) having position, rotation, and scaling properties in x, y, and z dimensions.*

Just as native Flash classes inheriting from the class `flash.display. DisplayObject` produce object instances that can be placed on the stage, Away3D classes inheriting from the class `away3d.core.base.Object3D` produce object instances that can be placed in a scene. Some of these have already been encountered in the previous chapter (primitives and lights). As you will see, many others exist within the engine, and we can also create our own.

As is the case with the native Flash display list, a 3D display list is made up of a tree hierarchy, in which 3D objects can be nested inside other 3D objects; these, in turn, can be nested inside further 3D objects, and so on all the way back to the `scene` object. These 3D container objects are generally invisible when rendered; only the noncontainers held within are actually displayed. In Away3D, the scene itself (represented by the `Scene3D` class) is a subclass of a 3D container, the `ObjectContainer3D` class. We will discuss how intelligent use of 3D containers can benefit the positioning and animating of 3D objects later in this chapter.

Camera

The `Camera3D` object, a class inheriting from `Object3D`, represents the point of observation inside a scene. It influences what is visible in the view by calculating a projected image based on the camera's position and rotation. It is as if the view is showing you the contents of the scene from the point of view of the `Camera3D` object.

Under the hood, the camera is nothing more than a set of formulas by which the 3D coordinates in the scene are transformed into 2D image coordinates in the view. Aside from position and rotation, the camera can alter how the scene is rendered with properties such as `zoom` and `focus`, and the `lens` property defines the type of projection performed. These attributes affect the rendered output in the same way that selecting different lenses and settings for a real life camera affects the resulting photograph and will be covered in greater detail later in this chapter.

Exploring the fundamentals of 3D

Before diving into the deep end of Away3D content creation, it is a good idea to lay down some basic principles that hold true for any 3D framework. The most basic of these is the concept of a 3D coordinate system and how it differs from the 2D one we use daily in Flash.

Working with coordinates in 3D space

You have likely encountered 2D coordinate systems several times before. If you didn't study them in geometry or algebra courses, you will have probably worked with them (knowingly or not) in Flash, Photoshop, or even word processing software such as Microsoft Word or OpenOffice Writer.

A **coordinate system** is a way of defining a position in space, by decomposing the distance from a zero point into a number of distance components. Each distance component has a direction defined by an axis, which is a line drawn along the points in space that increment or decrement the distance (or **coordinate**) for that direction.

In 2D space, the most common coordinate system uses two component directions arranged perpendicular to each other—the **X axis** and **Y axis**. For image editing software, the position of any graphical element on the screen can be defined using the x and y coordinate values determined by these axes.

Axes have their coordinate values defined as the distance from a single zero point. This point is generally positioned where the two axes cross—either at the top-left corner of the screen (used in image editing software), or the bottom-left corner of the screen (used in graphing software). This special point, in which the distance on each axis is zero, is called the **origin**.

3D coordinate systems have one major difference from 2D systems; they have a third axis, named Z. The Z axis is positioned perpendicular to the other two, and when being considered in computer graphics, is usually characterized as either going into or coming out of the screen.

Figure 3-1 displays the Away3D and native Flash coordinate systems side by side. In both cases, the X axis defines the position from left to right on the screen, and the Z axis defines the position into the screen (the distance from observer to object). However, the Y axis points in opposite directions for each system, resulting in corresponding values for the same point on-screen having opposite signs (positive becomes negative and vice versa). For the native coordinate system in Flash 10, having the Y axis pointing down is a remnant of Flash being 2D-only, with a typical top-left origin. Most 3D engines share the Away3D system of an upward-pointing Y axis, which define positive direction from the bottom to the top of the screen.

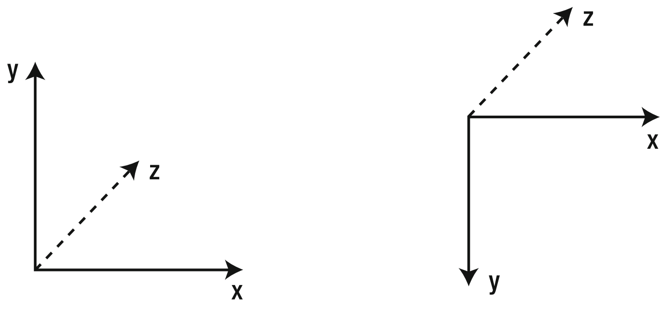


Figure 3-1. The Away3D coordinate system (left) compared to the native Flash coordinate system in Flash 10 (right)

Using our chosen 3D coordinate system, a position can be defined as the combined total of the distance along each axis. This position definition is sometimes referred to as the **vector** of the point, although the term is generally used more widely in 3D graphics terminology than 2D. In 3D, a vector has three components, the X, Y and Z values, corresponding to the point's distance measured along the three respective axes. Such a vector is commonly written (X, Y, Z), that is, the three values in alphabetical order, separated by commas and surrounded by parentheses.

Figure 3-2 shows how a point's vector is calculated from the axes of a coordinate system in 3D.

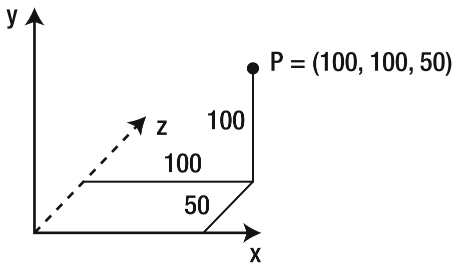


Figure 3-2. A point, P, positioned at X = 100, Y = 100, Z = 50 in the Away3D coordinate system

The rendering process

Armed with the Away3D coordinate system, you are now ready to start considering how objects in the scene are prepared for visualization in the view. Preparation can take any number of steps, but the most basic ones we will look at now are clipping, Z sorting, and perspective projection.

Clipping

Clipping ensures that only the objects visible within the view are considered for rendering. A simple way of determining this is to consider the position of projected vectors relative to the edges of the view (defined either by the edges of a viewport or the edges of the Flash movie). If a vector falls outside the clipping area of the view, it is flagged for removal before the rendering process takes place. In this way, the amount of drawing operations (and therefore the amount of processing) required to render a view can be greatly reduced. Using this technique is explored in more detail in the "Creating and using the view" section of this chapter.

Z sorting

Z sorting is a process that calculates the order objects are drawn to screen. We generally take for granted that, in real life, objects closer to us appear in front of objects further away. In a 3D engine, this order of overlapping has to be calculated every frame so that the 3D objects displayed in a view appear natural. Luckily, our coordinate system holds the information needed to solve this problem—using the Z coordinate of our projected 3D vector for each point in the scene, we can sort all 3D objects into a list arranged by Z value. To display the scene correctly in the view, we then draw the list of objects back to front (largest Z to smallest Z), in a process known as the **painter's algorithm**. Like an oil painting, objects drawn to the view overlap objects already onscreen requiring background objects to be drawn before foreground objects.

From Flash 10 onward, native display objects have positions based in a 3D coordinate system but are missing automatic Z sorting. This means display objects placed far away on the Z axis won't necessarily be drawn behind those nearby, occasionally resulting in unnatural overlapping. Away3D automatically executes Z sorting on objects in a scene, ensuring that all 3D objects are rendered to the view correctly.

Perspective Projection

Perspective Projection is the mechanism that converts the 3D coordinates of 3D objects in a scene to the 2D coordinates required for drawing 2D objects into a view. In Away3D, this process is handled by the lens classes found in the `away3d.cameras.lenses` package. A single lens instance is assigned to the `lens` property of the camera in order to perform perspective projection on a scene. There are many different lenses available for different projection techniques, and these are investigated in more detail in Chapter 10. However the standard process takes the X and Y components of the scene vector and divides them in turn by the Z component to get the X and Y components of the view vector (more often referred to as the **screen vector**). This process is crucial to achieving perspective in a view, which is characterized by objects appearing smaller at greater distances.

Setting up the chapter base class

To provide an illustration of the view, scene, and camera implementations in Away3D, we'll start by creating a base class that all examples in this chapter will extend. Create the `Chapter03SampleBase` class in the `flash3dbook.ch03` package with the following code, and study it carefully, as the rest of the chapter will rely on you understanding and extending its functionality.

```
package flash3dbook.ch03
{
  import away3d.cameras.*;
  import away3d.containers.*;
  import away3d.materials.*;
  import away3d.primitives.*;

  import flash.display.*;
  import flash.events.*;

  [SWF(width="800", height="600")]
  public class Chapter03SampleBase extends Sprite
  {
    protected var _view : View3D;
    protected var _cube1 : Cube;
    protected var _cube2 : Cube;

    public function Chapter03SampleBase()
    {
      _createView();
      _createScene();
      _createCamera();
    }

    protected function _createView() : void
    {
      // Create view and add it to the stage
      _view = new View3D();
      addChild(_view);

      //Relocate center point of view to the center of an 800x600 stage.
      _view.x = 400;
      _view.y = 300;

      //call the view render method on every frame of the Flash movie
      addEventListener(Event.ENTER_FRAME, _onEnterFrame);
    }

    protected function _createScene() : void
    {
      // Create a new scene containing a trident and two cubes
      var scene : Scene3D = new Scene3D();
      var trident : Trident = new Trident(200, true);
      _cube1 = new Cube();
      _cube1.x = -100;
      _cube1.material = new WireColorMaterial(0xFFFFFF);
      _cube2 = new Cube();
      _cube2.x = 100;
      _cube2.material = new WireColorMaterial(0x888888);
      scene.addChild(trident);
      scene.addChild(_cube1);
      scene.addChild(_cube2);

      //Assign the new scene to the view
      _view.scene = scene;
    }

    protected function _createCamera() : void
    {
    }

    protected function _onEnterFrame(ev : Event) : void
    {
      _view.render();
    }
  }
}
```

The class defines four protected methods, `_createView()`, `_createScene()`, `_createCamera()`, and `_onEnterFrame()`. The first three are called from the constructor, with the `_onEnterFrame()` method set up as a handler for the `ENTER_FRAME` event. `_createView()` and `_createScene()` starts out with some code already in place so that we can get some visual feedback in the earlier examples, although we will be dealing more thoroughly with `_createScene()` later in this chapter. `_createCamera()` is not used until the "Creating and using cameras" section.

Creating and using the view

In Away3D, the `View3D` class located in the `away3d.containers` package represents the view. It extends the native `Sprite` class and is displayed by instantiating the class and adding it to the display list of a Flash movie using `addChild()`, as you would any regular display object. For example, take a look at the first few lines of the `_createView()` method in the `Chapter03SampleBase` class.

```
// Create view and add it to the stage
_view = new View3D();
addChild(_view);
```

For the scene and its objects to be made visible, the contents of the scene must be drawn into the view by invoking the `render()` method on the `View3D` instance. Typically, the `render()` method should be executed once per frame, so that any updates in the scene can be redrawn continuously. For this reason, the `render()` method is usually called inside an `ENTER_FRAME` event handler, as illustrated in the `_onEnterFrame()` method at the end of the `Chapter03SampleBase` class definition.

```
protected function _onEnterFrame(ev : Event) : void
{
  _view.render();
}
```

The `_onEnterFrame()` method is set up as a handler for the `ENTER_FRAME` event the end of the `_createView()` method.

```
//call the view render method on every frame of the Flash movie
addEventListener(Event.ENTER_FRAME, _onEnterFrame);
```

Rendering is the single most processor-intensive task performed by a 3D framework. It involves a lot of calculations, performed recursively on the entire scene. The built-in caching system of Away3D optimizes the render pipeline by making sure only those parts of the scene that have changed since the last render will be redrawn. However, for static scenes, halting the render process manually by ceasing `render()` method calls may be more efficient. In these situations, it is worth noting that 3D mouse events will also not execute, as they rely on the `render()` method to update their event model. Chapter 8 deals with 3D mouse events in more detail.

As well as automatically creating a camera and scene container on instantiation, the view will also create a default center point and clipping parameters. Let's take a closer look at how these are defined and what happens when they are changed.

Centering the vanishing point

The x and y positions of the view on the stage define a center point for all perspective projection calculations. In 3D graphics, this center point is commonly called the **vanishing point** and represents the position toward which objects converge the further away they are. The center point also represents the default screen vector for the projected origin of the scene

By default, the center point of the view is located at (0,0) in the top left-hand corner of the stage. To achieve a natural symmetry in the projection of 3D objects, it is best to move the center point of the view to the center of the viewport (in this case the center of the stage). This is done by setting the x and y properties as you would any other display object after instantiation, as you see in the `_createView()` method of the `Chapter03SampleBase` class following the instantiation of the view.

```
//Relocate the center point of the view to the center of an 800x600 stage
_view.x = 400;
_view.y = 300;
```

Figure 3-3 demonstrates the effects of not doing this step: Compiling the `Chapter03SampleBase` class with the preceding lines commented out displays the result shown on the left. Here, the vanishing point of the view (and the projected origin of the scene) is positioned in the top-left corner of the stage, and the resulting viewport can only display the bottom-right quadrant of the scene in the view. Uncomment the lines to center the view by setting the x and y properties to half the width and height of the stage respectively, and recompile the class to see the result shown on the right. The entire scene is now visible, with the vanishing point correctly located at the center of the stage.

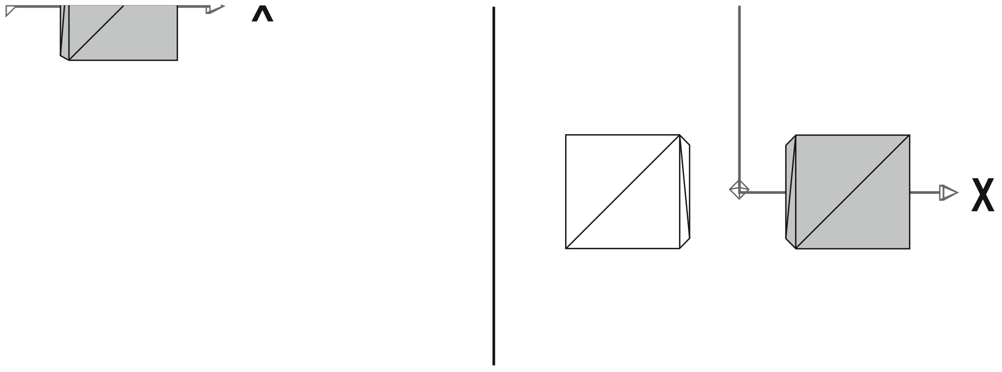


Figure 3-3. On the left, the center point of the view remains unchanged in the default 0,0 position. On the right, the view has been positioned at the center of the stage.

Clipping the viewport

As previously mentioned, the cropped area of the view once it is added to the stage is generally referred to as the viewport. The process of cropping the view to the viewport is called **clipping**, and in Away3D is controlled by one of the clipping classes available in the `away3d.core.clip` package. Like the scene and camera objects, a default clipping instance is created in the `clipping` property of the view object on instantiation. The clipping classes represent a variety of different types of clipping, but all define the viewport's boundaries with the use of `minX`, `maxX`, `minY`, and `maxY` properties.

By default a `RectangleClipping` object is created for the `clipping` property of the view, with boundary properties set to infinity. This forces the view to use the boundaries of the stage as its viewport area, as seen in the rendered output of the current base class displayed on the left in Figure 3-4.

To restrict the view to a smaller area of the screen, the boundary properties of the clipping object can be reset to values that define the maximum and minimum extent of the viewport in the x and y directions, as measured from the vanishing point of the view (the (X, Y) position of the view object on the stage). Let's create a new example that extends the `Chapter03SampleBase` class to investigate the effects of the `clipping` property on the view object.

```
package flash3dbook.ch03
{
  import away3d.core.clip.*;

  [SWF(width="800", height="600")]
  public class ViewportClipping extends Chapter03SampleBase
  {
  protected override function _createView() : void
    {
      super._createView();

      //defining a viewport size of 200 x 200 pixels
      _view.clipping.minX = -100;
      _view.clipping.maxX = 100;
      _view.clipping.minY = -100;
      _view.clipping.maxY = 100;
    }
  }
}
```

The preceding code sets the boundary values of the clipping object to +/- 100 pixels, defining a viewport size of 200 × 200 around our vanishing point. Recompiling the `Chapter03SampleBase` class will display the result shown on the right in Figure 3-4.

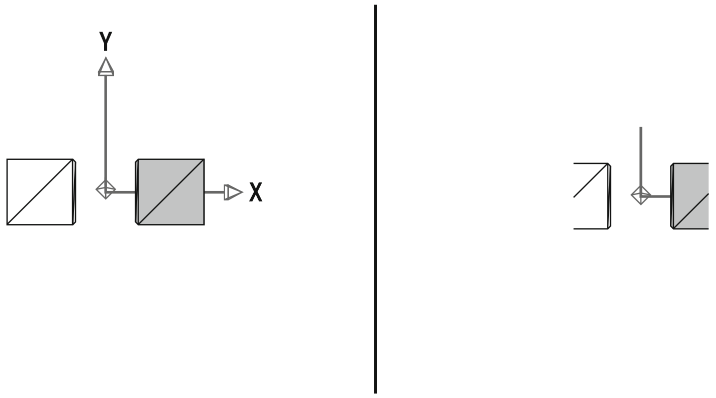


Figure 3-4. On the left, the default RectangleClipping object of the view displays all content right up to the edges of the Flash movie. On the right, the custom clipping values crops the scene to a 200 × 200 viewport.

An alternative method of clipping can be achieved by resetting the `clipping` property on the `View3D` object using a new `RectangleClipping` instance with the boundary properties already set. To demonstrate this, replace the `_createView()` method of our `ViewportClipping` example with the following:

```
protected override function _createView() : void
{
  super._createView();

  //defining a new clipping object
  var clipping:RectangleClipping = new RectangleClipping();
  clipping.minX = -100;
  clipping.maxX = 100;
  clipping.minY = -100;
  clipping.maxY = 100;

  //resetting the clipping instance on the view
  _view.clipping = clipping
}
```

Recompiling should display exactly the same result shown on the right in Figure 3-4.

Managing the scene

As mentioned previously, the scene is to Away3D what the stage is to a standard Flash movie. To make a 3D object available for rendering, it needs to be added to the `Scene3D` object in the same way a `Sprite` needs to be added to the root document instance of the movie. This is what we've been doing so far with cubes, spheres, and tridents in the `_createScene` method of the `Chapter03SampleBase` class.

The Scene3D object can contain a hierarchy of nested 3D objects just like a Flash display list. This hierarchy is more commonly known as a **scene graph**.

Adding and removing 3D objects

The majority of renderable objects in Away3D inherit from the Object3D base class. Objects of this type are valid as nodes in the scene graph and can be added using the addChild method of the scene. Looking at the _createScene() method of the Chapter03SampleBase class, you can see how renderable 3D objects are added one at a time in this manner.

```
scene.addChild(trident);
scene.addChild(_cube1);
scene.addChild(_cube2);
```

The addChild method (and other methods related to the management of child objects) is inherited from the ObjectContainer3D class from which Scene3D extends. This is a generic container class for the scene graph that can used to create a nested hierarchy for objects contained within the scene. The ObjectContainer3D class extends the Object3D base class, once again bearing great similarity to the native display list classes in Flash (see Figure 3-5).

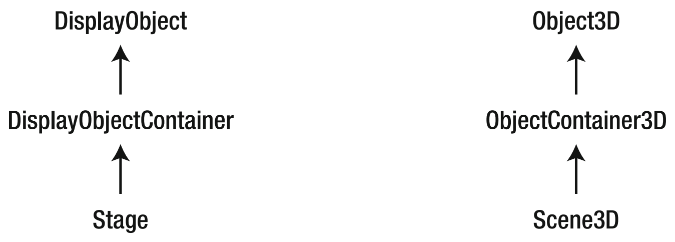


Figure 3-5. Comparison between a simplified display list class hierarchy in Flash (left) and scene graph class hierarchy in Away3D (right).

Let's create a new example that extends the Chapter03SampleBase class to investigate the properties of the Scene3D and ObjectContainer3D objects.

```
package flash3dbook.ch03
{
import away3d.primitives.Cube;
import away3d.containers.ObjectContainer3D;
import away3d.core.base.Object3D; [SWF(width="800", height="600")]
  public class SceneProperties extends Chapter03SampleBase
  {
    protected override function _createScene() : void
    {
      super._createScene();
      _view.scene.removeChild(_cube2); // Remove from scene
    }
  }
}
```

To remove objects that already exist in the scene from the scene graph, you can use the `removeChild` method (again inherited from the `ObjectContainer3D` class). Child objects are removed from their parent by executing this function on the parent object container. In the preceding code, we use the `removeChild()` method to delete the second cube (gray) from our scene before it is displayed in the view. Compiling this example will display just the first cube (white) and the trident. Again, this bears a striking resemblance to how native display objects are removed from the display list.

> *Even though the API for adding and removing scene objects in Away3D is very similar to the API used for display objects in Flash, they are not interchangeable. You cannot add an instance of a Flash display object to an `ObjectContainer3D` instance, or an Away3D scene object to a `Sprite` instance.*

Accessing 3D objects in the scene

One way of accessing the children of an `ObjectContainer3D` instance is to use its `children` property. This returns an `Array` object containing all 3D objects that are children of the 3D container and can be used for determining certain useful information about the child objects contained within. For example, the `length` property of the `children` array can be used to retrieve the total number of child objects in a container. Replacing the `_createScene()` method of the `SceneProperties` example with the following code and recompiling will return the number 3 in the trace output window, because three objects exist as children of the scene:

```
protected override function _createScene() : void
{
  super._createScene();
  trace(_view.scene.children.length);
}
```

Retrieving a 3D object in a scene if it is known in what order it was added to a 3D container is easily done by accessing the correct index of the `children` property. Replacing the `_createScene()` method of the `SceneProperties` example with the following code and recompiling will return `true` in our trace output window, because the first child added to the scene is a cube object.

```
protected override function _createScene() : void
{
  super._createScene();
  trace(_view.scene.children[0] is Cube); // returns true
}
```

3D objects in a scene can also be retrieved by using the `name` property, which is a unique string ID usually set before adding each object to the scene. Once added, the 3D object is retrieved using the `getChildByName()` method of the scene. As a demonstration, replace the `_createScene()` method of the `SceneProperties` example with the following code. Recompiling returns `true` in our trace output window because "`theFirstCube`" is the name of the `_cube1` object in the scene.

```
protected override function _createScene() : void
{
  super._createScene();
  _cube1.name = "theFirstCube";
  trace(_view.scene.getChildByName("theFirstCube") is Cube); // returns true
}
```

One advantage of using the `getChildByName()` method is its recursive nature. The method can be used to return an object from any 3D container in the scene graph, even if it is buried in a list of subcontainers.

Here's a note on performance: Although 3D objects can at any time be retrieved using names or index values, storing a direct reference to the object by declaring a global variable will always be faster. If you're planning to access a 3D object many times, such as in a recursive code loop, it will always be more efficient to take the direct reference approach.

There is no easy way of adding a 3D object at a specific index of the `children` array, because object order is of little significance in a scene graph. In Away3D, Z-sorting algorithms define the order in which items are drawn to screen. In the native display list framework in Flash, no Z sorting exists, so the order in which objects are added is the order in which they are drawn. This explains why methods such as `getChildIndex()` and `setChildIndex()` are omitted from the `ObjectContainer3D` and `Scene3D` classes.

Working with nested 3D objects

As you have seen, the scene graph container object `ObjectContainer3D` inherits from `Object3D` and can therefore be added as the child of another container. This allows the nesting of container objects to whatever depth you desire. For example, replacing the `_createScene()` method of the `SceneProperties` example with the following code and recompiling returns `"true" "true" "true"` in the trace output window.

```
protected override function _createScene() : void
{
  super._createScene();
  _cube1.name = "theFirstCube";
  var myGrandParent : ObjectContainer3D = new ObjectContainer3D();
  var myParent : ObjectContainer3D = new ObjectContainer3D();
  var mySelf : Object3D = new Object3D;

  myParent.addChild(mySelf);
  myGrandParent.addChild(myParent);
  _view.scene.addChild(myGrandParent);

  trace(mySelf.parent == myParent); // prints true
  trace(mySelf.parent.parent == myGrandParent); // prints true
  trace(myGrandParent.children[0].children[0] == mySelf); // prints true
}
```

Here, having two generations of children in a scene graph is perfectly acceptable. The `parent` property exists for all `Object3D` instances and points to the `ObjectContainer3D` instance in which they reside. A 3D object can only have one parent; if `addChild()` is performed on an object that is already added elsewhere in the scene graph, it is removed from that location before being added to the new 3D container.

On the last line of the preceding code, we directly access the `children` array to return the first (and only) child of `myGrandParent`, and in turn the first (and only) child of that child. As is confirmed by the equality test, `mySelf` is the grandchild of `myGrandParent`.

For performance reasons, we advise you not to nest scene graph objects any deeper than necessary. As you will see, nesting can be used to avoid advanced mathematics when animating objects, but a deeply nested scene graph requires more processing to render. To optimize your scene, it is best to only nest objects when required to avoid unnecessary performance costs.

Keep in mind that to remove 3D objects from the scene graph, the `removeChild()` method needs to be executed on the immediate parent of the object. Adding the following code to the `_createScene()` method above will work because `mySelf` is a child of `myParent`.

```
myParent.removeChild(mySelf); // Works, myParent is the parent of mySelf
```

However, adding the following code fails, because `myself` is not a child of `myGrandParent`:

```
myGrandParent.removeChild(mySelf); // Fails
```

Moving, rotating, and scaling in 3D

In a scene graph, 3D object coordinates are not locked to the global coordinate system of the scene but are influenced by the 3D containers above them in the scene graph hierarchy. This concept can be confusing at first, but you will eventually see how this can be used to our advantage when carrying out more complex rotating and translating operations.

The concept of hierarchical coordinates systems is nothing new, and in Flash is something we are used to working with in 2D every day. As a demonstration, create a new `MovieClip` object in Flash, add it to the stage, and rotate it 45 degrees. Create another `MovieClip` inside the first, and rotate this one 45 degrees as well. You will notice how the inner clip appears rotated by 90 degrees, even though when you trace its rotation value, it returns 45. This is not a mistake—what we are seeing here is the different levels of rotation in a coordinate system that imposes the transformations of a container on its children in a cumulative manner. Such an arrangement is know as a **hierarchical coordinate system** and is frequently used in graphic design software.

Adding an extra dimension makes things difficult to follow when talking about cumulative transformations, so for now, let's stick to the world of 2D for a more abstract example. Imagine a point on the Y axis in a 2D coordinate system. Since it lies on the Y axis, we know that its coordinate value on the X axis is 0. Such a point is shown on the left in Figure 3-6.

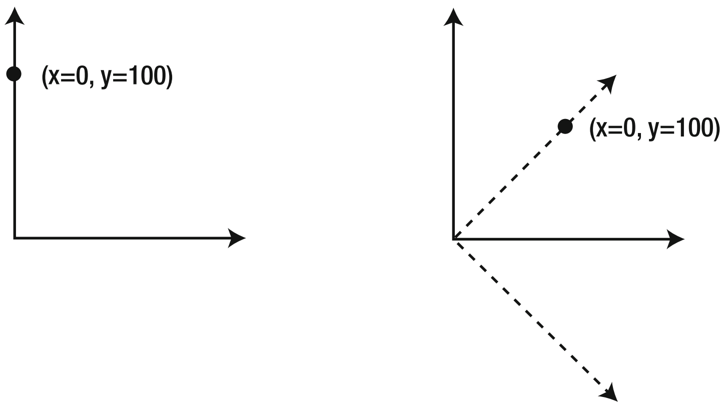


Figure 3-6. Rotating a coordinate system affects the correlation between the global position and local position of a point.

Now, let's rotate the coordinate system of the point 45 degrees, as if it were in a display object container being rotated. The result is displayed on the right in Figure 3-6. You can easily see that the global position of the point no longer matches the local position, even though its local position vector has not changed.

Understanding this concept in 2D, you can now take the step into 3D by investigating how rotating the axes of an `ObjectContainer3D` will affect its local coordinate system. For 2D, we only have the option of rotating in the plane of the 2D surface (around the theoretical Z axis). In 3D, we have the option to rotate around the X, Y, or Z axis, or all 3 at once!

In Figure 3-7, we stick to representing only two axes for the sake of clarity (the X and Y axes), but this time, we will rotate in two separate stages around the Z and Y axes respectively. The image on the left represents the starting state, with a 3D block aligned along the Y axis and centered on the X and Z axes.

In the first step, we apply the same rotation used in Figure 3-6, 45 degrees around the Z axis (which you can imagine as a perpendicular line pointing into the page) of the 3D container. The rotation ends with the Y axis pointing diagonally up and to the right. The block follows the rotation, remaining relative to its parent coordinate system and ends up in the orientation shown in the center image of Figure 3-7.

The second step is the one crucial to understanding the nature of a hierarchical coordinate system. Here, we rotate the block 45 degrees around its local Y axis. Because we first rotated the coordinate system of the 3D container, the rotation will now be performed around the transformed representation of the Y axis. The revolving arrow in the center image of Figure 3-7 depicts this rotation.

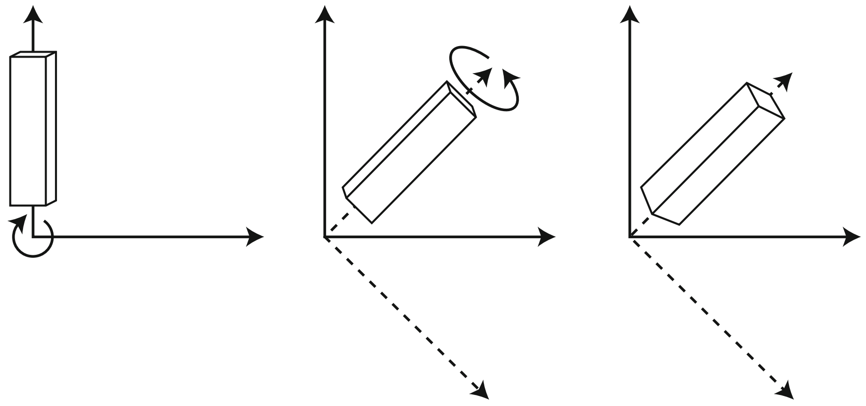


Figure 3-7. Two consecutive rotation operations are applied to a 3D container with a rectangular block contained within. The local Y axis is transformed by the rotation performed around the Z axis in the first step, affecting the rotation performed around the Y axis in the second step.

The end result, after having rotated the block 45 degrees around its local Y axis, is displayed in the right image of Figure 3-7.

> *When rotating a single 3D object around more than one axis, the order of rotation operations is important. Preceding rotations can affect the resulting direction of axes used for subsequent rotations. This is a common problem in 3D known as* ***gimbal lock*** *and is discussed in more detail in Chapter 10.*

Using containers as pivots

From the previous section in this chapter, you should now be familiar with the workings of nested 3D containers. Next, let's explore how this can be applied to our advantage in the scene graph of the Away3D engine. If you're not a master of 3D math, you will soon discover that rotating a 3D object around an arbitrary point in space (such as a simulation of the Earth revolving around the sun or a pendulum swinging around its pivot) is not a trivial task. However, it can be made a little simpler by taking advantage of the scene graph's hierarchical coordinate system.

In Away3D, the `rotationX`, `rotationY`, and `rotationZ` properties apply a rotation around the respective X, Y, and Z axes of a 3D object. For geometric primitive objects created internally (such as cubes, spheres and cylinders), the origin of the local coordinate system around which rotations are performed is usually found at the center of the geometry. For example, when you rotate a sphere around any axis, the rotation is performed around the center of the sphere. Creating 3D objects centered on their local origin is also common practice in 3D modeling software, resulting in many imported models having the same characteristics.

However, centered rotations are not always what we are after. In the case of a pendulum, the rotation needs to be performed around a point at one end of the object. Let's create a new example that extends the `Chapter03SampleBase` class to explore how this can be done using Away3D.

```
package flash3dbook.ch03
{
import away3d.containers.*;
import away3d.primitives.*;

import flash.events.Event;
import flash.utils.getTimer;
  [SWF(width="800", height="600")]
  public class PendulumContainer extends Chapter03SampleBase
  {
    private var _pendulum:ObjectContainer3D;

    protected override function _createScene() : void
    {
      // Create a new scene containing a pendulum
      var scene : Scene3D = new Scene3D();
      var trident : Trident = new Trident(100, true);
      var sphere : Sphere = new Sphere();
      sphere.radius = 20;

      //create a new pendulum container
      _pendulum = new ObjectContainer3D();
    }
  }
}
```

The preceding code overrides the `_createScene()` method and creates a new scene object, a trident object, and a sphere primitive with a radius of 20. We also create a 3D container object in a global variable called `_pendulum` that can be accessed from anywhere in the class, by defining a new instance of `ObjectContainer3D`. This will act as our pendulum container.

Next, we set the y position of `sphere` to –100 and add it as a child of **`_pendulum`** by adding the following code to the end of the **`_createScene`** method. This creates a sphere inside the pendulum container with a local coordinate offset of –100 units along the container's Y axis.

```
//offset the local position of the sphere to (0, -100, 0)
sphere.y = -100;

//add the sphere to the pendulum container
_pendulum.addChild(sphere);
```

To visually identify where the local origin sits for the pendulum container, we add the following code to the end of the `_createScene()` method to add the newly created `trident` object as another child of `_pendulum`.

```
//add the trident to the pendulum container
_pendulum.addChild(trident);
```

Finally, we add the following code to the end of the `_createScene()` method to add the pendulum container as a child of the scene and assign the new scene to the view so that it is displayed on rendering.

```
//add the pendulum container to the scene
scene.addChild(_pendulum);

//Assign the new scene to the view
_view.scene = scene;
```

Rotating the sphere primitive at this point would simply cause the object to revolve around its own axis. But if we rotate the pendulum container, the sphere will swing in an arc similar to the movement seen in Figure 3-6 where the global position of an object is altered by rotating its parent.

To animate the pendulum container, we can use the native `getTimer` method on every frame of the Flash movie as a linearly incrementing value to drive the swing. For a realistic swinging motion, a sine function can be used on the `getTimer()` value. Add the following code to the end of our `PendulumContainer` class definition to override the `_onEnterFrame()` method:

```
protected override function _onEnterFrame(ev : Event) : void
{
  _pendulum.rotationZ = 45 * Math.sin(getTimer() / 500);
  _view.render();
}
```

Compiling the code will animate the sphere in a swinging motion, rotating the pendulum container back and fourth between 45 and –45 degrees around its origin. For comparison, try switching the `pendulum.rotationZ` property to `pendulum.children[0].rotationZ` (the `rotationZ` property of the sphere) and recompiling to see the difference between rotating the pendulum container and rotating the sphere primitive directly.

The pendulum-like motion seen in the `PendulumContainer` example could have been achieved using trigonometry in place of nested 3D containers. However, the approach used here is a much simpler and more intuitive method, especially if we start to perform more complex nested rotations.

Creating and using cameras

Away3D offers several different types of camera object, each represented by its own class found in the `away3d.cameras` package. While the primary job of a camera is to represent the viewer's position in space relative to the rest of the scene, the camera object also defines `zoom` and `focus` properties that affect the projection of the scene into the view, something we will look at here in more detail. Each camera type in Away3D allows different methods of movement, and the one you use depends on the type of viewing you require.

The basic camera in Away3D is the `Camera3D` class, an instance of which is created by default in the `camera` property of the `View3D` object. This camera produces a standard, front-facing perspective projection of the scene by positioning itself –1,000 units along the Z axis and facing directly toward the scene's origin.

The Camera3D object

Let’s create a new document class that extends `Chapter03SampleBase` with the following code, overriding the `_createCamera()` method to explore the various camera properties we have available for manipulating the `Camera3D` object.

```
package flash3dbook.ch03
{
  import away3d.cameras.*;

  [SWF(width="800", height="600")]
  public class CameraProperties extends Chapter03SampleBase
  {
  protected override function _createCamera() : void
    {
      // Create a new camera object
      var camera : Camera3D = new Camera3D();
      camera.x = 0;
      camera.y = 0;
      camera.z = -1000;

      //Assign the new camera to the view
      _view.camera = camera;
    }
  }
}
```

In the preceding code, we replace the default camera by resetting the `camera` property on the view with a new `Camera3D` instance. We start this example by recreating the view’s default camera position, and because we have used a `Camera3D` instance, we have total control over the motion the camera can perform. We will take a look at more bespoke camera types later; for now, let’s explore how we can adjust the `Camera3D` properties with our new setup to produce different results in the view.

Moving the camera

Compiling the `CameraProperties` class at this stage will display what you have already seen in previous examples, shown in the top left-hand image of Figure 3-8. With this setup, we have the ability to change the `x`, `y` and `z` properties of the camera, directly controlling the X, Y, and Z components of its position vector in the 3D space of the scene. Changing the position of our camera will affect the output of the rendered view, as we can see with the four different setups represented in the four images of Figure 3-8. You can try these out for yourself by replacing our new `Camera3D` object’s `x`, `y`, and `z` property values with the corresponding x, y, and z values displayed for each image.

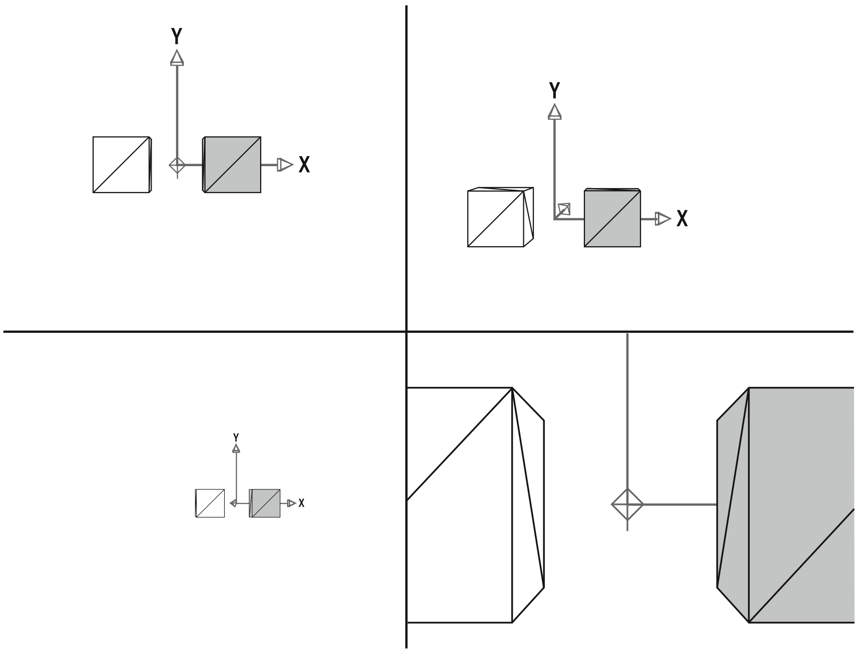


Figure 3-8. The same scene from four different camera positions: The top-left image uses the default camera position, x = 0, y = 0, z = –1.000. In the top-right image, the camera is moved up and right to x = 150, y = 100, z = –1,000. In the bottom-left image, it is moved back and left to x = –200, y = 0, z = –2,000. And in the bottom-right image, it is moved forward to x = 0, y = 0, z = –200.

Rotating the camera

Camera objects in Away3D are rotated using the same `rotationX`, `rotationY`, and `rotationZ` properties that regular 3D objects use. The names of each rotation property reflect the axis around which the rotation is carried out, and their values represent the amount of rotation, in degrees. However, the visual effect of rotating a camera appears different to rotating an object, because the camera is acting as your point of view. This means that rotating around the X axis results in a movement similar to nodding your head, rotating around the Y axis is similar to shaking your head to say "no", and rotating around the Z axis is similar to leaning your head left or right, as you might do if you had water in your ear!

Let's look at the result of a simple camera rotation by replacing the contents of the `_createCamera()` method in the `CameraProperties` example with the following code:

```
// Create a new camera object
var camera : Camera3D = new Camera3D();
camera.x = 0;
camera.y = 0;
camera.z = -1000;

// Rotate the camera by 10 degrees around the Y-axis
camera.rotationY = 10;
```

```
//Assign the new camera to the view
_view.camera = camera;
```

Here, we are resetting the camera position to the default set by the view, then rotating the camera 10 degrees around the Y axis. Recompiling `CameraProperties` will display the result shown in Figure 3-9, with the view appearing as though you have turned your head 10 degrees to the right. The left cube (white) has disappeared from your field of view, and the right cube (gray) is now displayed slightly to the left of the center of the stage.

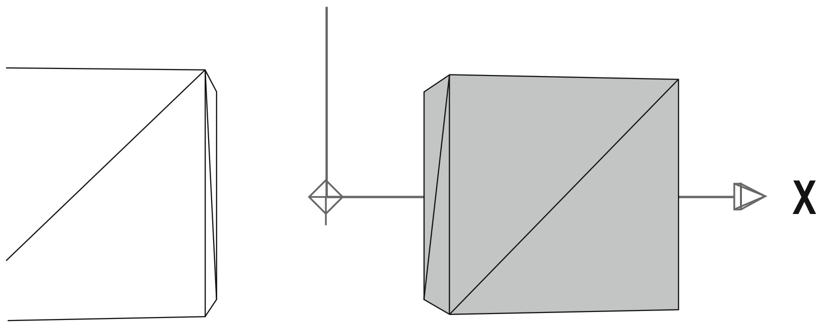


Figure 3-9. CameraProperties example viewed after rotating the camera 10 degrees around the Y axis, effectively twisting the field of view to the right.

Adjusting the zoom and focus properties

As mentioned previously, the camera in Away3D has properties that allow you to make image adjustments similar to those found on a real camera. For this sort of application, we aren't talking about exposure and shutter speed settings, but there are a few familiar sounding properties to be found on the `Camera3D` object such as `zoom`, `focus`, and `lens`. Here, we will look at the effects of adjusting the first two properties in that list. Lens controls are slightly more complex and are dealt with in more detail in Chapter 10.

To explore the effects of the `zoom` and `focus` properties, we can replace the contents of the `_createCamera()` method in the `CameraProperties` example with the following code:

```
// Create a new camera object
var camera : Camera3D = new Camera3D();
camera.x = 0;
camera.y = 0;
camera.z = -500;

//set the zoom and focus properties
camera.zoom = 10;
camera.focus = 100;

//Assign the new camera to the view
_view.camera = camera;
```

Here, we set the camera's z position to –500, which is half the default distance to the center of the scene. We also recreate the default settings of the `zoom` and `focus` properties of 10 and 100 respectively, so that we may adjust these in subsequent compilations. Recompiling the `CameraProperties` example will display the image shown in the top left-hand corner of Figure 3-10.

Let us start our adjustments by setting the `zoom` property to 20, or twice the default amount. The effect on the view is much the same as you would expect for a real life camera zoom control; the resulting display is scaled in proportion to the zoom value. Recompiling the `CameraProperties` example will output the image shown in the top-right corner of Figure 3-10, with the scene displayed at twice the scale of the image produced with our previous settings.

Next, let's explore the effect of adjusting the `focus` property. In this case, the value of `focus` does not relate to the focus setting in a real life camera but represents the distance between the camera position and the viewing plane. In 3D graphics, a **viewing plane** is an invisible surface in space that is used as the basis for projecting the scene to the view. If you imagine the computer screen as your viewing plane, the `focus` property adjusts the theoretical distance between the surface of this plane and your camera's position in front of the plane. A small focus value results in a camera very close to the viewing plane, with an extremely wide angle of view. If we adjust our focus property to `10` and recompile the `CameraProperties` example, the result (shown in the bottom-left image of Figure 3-10) looks similar to that of a wide-angle lens, with much more of the scene included in the viewport and very high perspective distortion. A large focus value places the camera further from the viewing plane, resulting in a narrower field of view and a loss of perspective distortion. Setting the `focus` property to `500` and the `zoom` property back to `10` and recompiling shows the image in the bottom right-hand corner of Figure 3-10, with a highly scaled scene where the sides of the cubes exhibit almost no perspective at all.

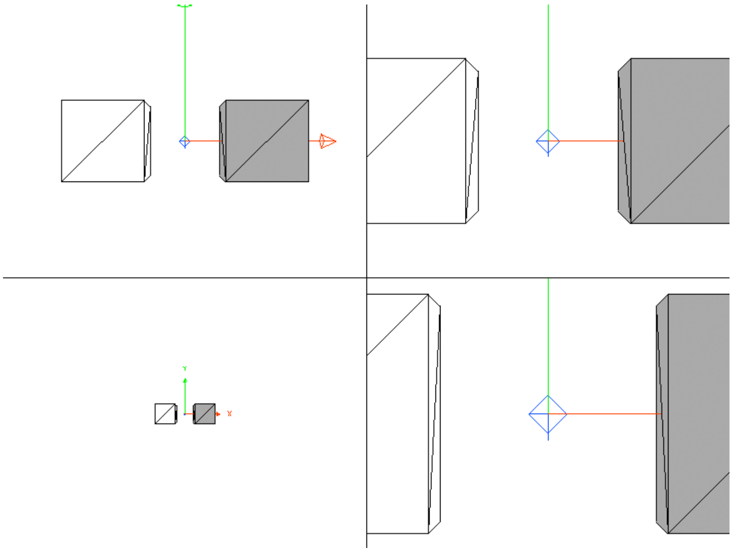


Figure 3-10. The same scene with four different zoom and focus settings on the camera: In the top-left image, the camera properties are set to the default values of zoom = 10, focus = 100. In the top-right image, the camera properties are set to zoom = 20, focus = 100. In the bottom-left image, the camera properties are set to zoom = 20, focus = 10. And in the bottom-right image, the camera properties are set to zoom = 10, focus = 500.

As the preceding example has demonstrated, the `focus` and `zoom` properties of the camera object allow us to adjust the amount of perspective distortion and scaling that is applied when projecting the contents of the scene into the view. These properties are available for all camera types in Away3D.

Aiming at objects using lookAt()

Often, the reason for rotating the camera is to center the viewport on some specific point in space, for example, an object of interest. Calculating the precise rotation needed to center an object is not a trivial task, which is why Away3D has a method to achieve this called `lookAt()`, defined on all camera classes.

The `lookAt()` method requires a position vector of the type `away3d.math.Number3D` (a 3D variant of the native `Point` class in Flash). This is used as the position in the scene toward which the camera must rotate. Let's create a new example that extends the `Chapter03SampleBase` class with the following code to explore the operation of the `lookAt` method.

```
package flash3dbook.ch03
{
  import away3d.cameras.*;
  import away3d.core.math.Number3D;

  [SWF(width="800", height="600")]
  public class CameraLookAt extends Chapter03SampleBase
  {
  protected override function _createCamera() : void
    {
      // Create a new camera object
      var camera : Camera3D = new Camera3D();
      camera.x = 1000;
      camera.y = 500;
      camera.z = -1000;

      //Use lookAt() to point the camera towards the center of the scene
      camera.lookAt(new Number3D(0, 0, 0));

      //Assign the new camera to the view
      _view.camera = camera;
    }
  }
}
```

In the `_createCamera()` method, a new `Camera3D` object is created with its position vector set to (1000, 500, –1000) in the scene (up and to the right). We then use the `lookAt()` method on the camera to point it towards the scene's origin at (0, 0, 0). Compiling the code should display the output shown in Figure 3-11.

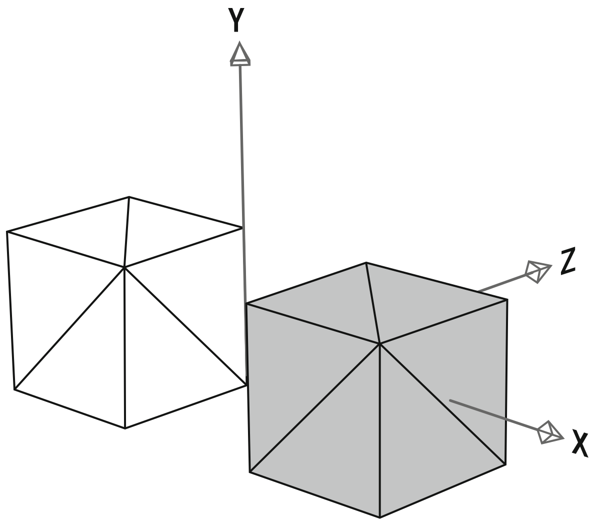


Figure 3-11. Sample scene after moving the camera up and to the right and using lookAt() to point it toward the scene origin

More often, we want to center the viewport on an object rather than an arbitrary point in space. All 3D objects have a `position` property that represents the position vector of the object's local coordinate system as a `Number3D` object. This property can be used with the same `lookAt()` method call in our example to point the camera toward an object's position. Let's try this by replacing the `lookAt` line in the `CameraLookAt` example with the following code to center the viewport on the second (gray) cube primitive, which is located to the right of the scene's origin:

```
//Use lookAt() to point the camera towards the cube2 primitve
camera.lookAt(_cube2.position);
```

The `lookAt()` method on the `Camera3D` object is inherited from the `Object3D` class, meaning that any 3D object in Away3D can be rotated to point toward a position vector in the scene. However, in practice, its most common use is in manipulating camera rotations.

The TargetCamera3D object

Most of the time, the Away3D applications you create will be animated, interactive scenes rather than still images. At some point, it will make sense to have the camera track a moving object so that the object stays in view at all times. If you've been following the previous section, you might already have solved that problem in your head, thinking a good solution would be to use the `lookAt()` method on every frame.

You'd be right, but since tracking an object is such a regular requirement in Away3D, the `TargetCamera3D` object will do this for you automatically. The `target` property tells a `TargetCamera3D` object which 3D object to track, and the camera rotation will update automatically each time you render the view.

Let's create another example extending the `Chapter03SampleBase` class and set up a target camera to track one of the cubes while the camera moves.

```
package flash3dbook.ch03
{
  import away3d.cameras.*;
  import flash.events.Event;

  [SWF(width="800", height="600")]
  public class TargetCameraMovement extends Chapter03SampleBase
  {
  protected override function _createCamera() : void
    {
      // Create a new camera object
      var camera : TargetCamera3D = new TargetCamera3D();
      camera.z = -1000;

      //Assign the camera target as cube2
      camera.target = _cube2;

      //Assign the new camera to the view
      _view.camera = camera;
    }
  }
}
```

The target camera in the preceding code has been configured to target the `cube2` object in our scene, which is the gray cube visible to the right of the scene's origin. Compiling the example renders the view with the `cube2` object at its center, in exactly the same orientation we saw in the previous `CameraLookAt` example demonstrating the `lookAt` method.

To confirm the `TargetCamera3D` object is updating its `lookAt` routine every frame, we need to add some movement. Override the `_onEnterFrame` method by adding the following code to the end of our `TargetCameraMovement` class definition:

```
protected override function _onEnterFrame(ev : Event) : void
{
  _view.camera.y = -(stage.mouseY - stage.stageHeight/2);
  _view.camera.x = stage.mouseX - stage.stageWidth/2;
  _view.render();
}
```

Recompiling the `TargetCameraMovement` example, we see that the x and y coordinates of the camera now update with the x and y coordinates of the mouse cursor. At all positions, the camera keeps the `cube2` object firmly locked to the center of the view.

The HoverCamera3D object

A common 3D application in Flash is one that allows the user to navigate all sides of a 3D model by rotating the camera around using the mouse. For this kind of interaction, the `HoverCamera3D` class is an ideal camera type, as it simplifies the necessary rotation and position updates required to orbit around a target object with the use of some extra features.

The HoverCamera3D class inherits from the TargetCamera3D class and adds some custom properties to fully serve its purpose. Let's build an example to demonstrate these new properties by extending the Chapter03SampleBase class with the following code:

```
package flash3dbook.ch03
{
   import away3d.cameras.*;
   import flash.events.Event;

   [SWF(width="800", height="600")]
   public class HoverCameraMovement extends Chapter03SampleBase
   {
     private var _hoverCamera : HoverCamera3D;

     protected override function _createCamera() : void
     {
       // Create a new camera object
       _hoverCamera = new HoverCamera3D();
       _hoverCamera.distance = 2000;
       _hoverCamera.tiltAngle = 10;

       //Assign the new camera to the view
       _view.camera = _hoverCamera;
     }

     protected override function _onEnterFrame(ev : Event) : void
     {
       _hoverCamera.panAngle = stage.mouseX - stage.stageWidth/2;
       _hoverCamera.hover();
       _view.render();
     }
   }
}
```

The _createCamera() method in the preceding code creates a new HoverCamera3D object and sets two of its properties, called distance and tiltAngle. Further down in the _onEnterFrame() method, the HoverCamera3D object has its _panAngle property updated from the x value of the mouse position. Compiling the movie should see the x coordinate of the mouse controlling the horizontal rotation at which the scene is viewed, with the camera looking down at an elevated angle onto the cube objects.

To explain what is going on, we need to analyze what the three new properties introduced in the hover camera are doing. Going over them again, these are the distance, panAngle, and tiltAngle properties of the HoverCamera3D object.

In Figure 3-12, the panAngle and tiltAngle properties represent the rotations (in degrees) performed on the hover camera by and β respectively. Here, a single cube and camera is displayed from above in the left image and from the side in the right image. Incrementing or decrementing the panAngle value causes the camera to orbit around the target object in a horizontal plane, as represented in the left image.

Incrementing or decrementing the `tiltAngle` value causes the camera to increase or decrease it's elevation angle, as represented in the right image. The `distance` property defines the radius on which both these rotations are performed. Two angles and a radius used to define a position in space in this manner are sometimes referred to as **polar coordinates**.

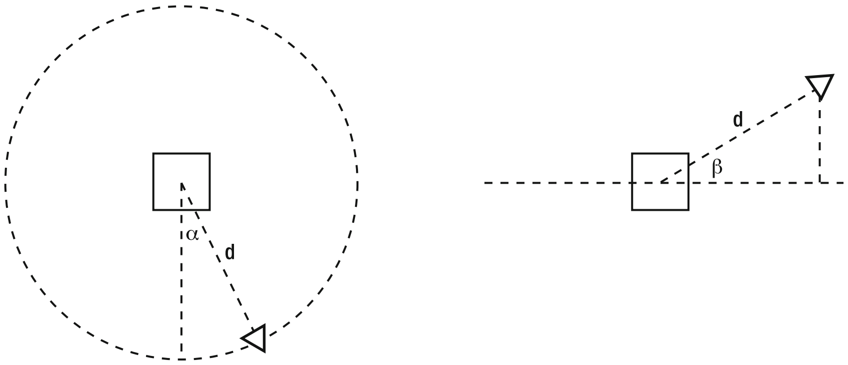


Figure 3-12. In this schematic view of a simple scene using the hover camera, the left image show the scene displayed from above, with denoting the panAngle property. The right images shows the scene displayed from the side, with β denoting the camera tiltAngle property. Note that d denotes the distance property in both cases.

In our example, `tiltAngle` is set to a constant value of 10 degrees, giving the camera a slightly elevated view of the scene. `panAngle` is set to update with mouse position, giving you full control over the horizontal angle at which the scene is viewed. The actual position and rotation values of the `HoverCamera3D` object are recalculated every frame from the `tiltAngle`, `panAngle`, and `distance` properties by calling the `hover()` method on the hover camera in `_onEnterFrame()`, directly before the view is rendered.

You may have noticed that our example has some inertial damping applied to the camera motion relative to the movement of the mouse. This is because `HoverCamera3D` updates its coordinates by default using a series of steps, easing into its new position. The severity of the ease can be modified using the `steps` property, allowing you to set very slow easing, medium easing or no easing at all should you wish. The default value is 8. You can experiment with this setting by adding the following line to the end of the `_createCamera()` method in our `HoverCameraMovement` example.

```
hoverCamera.steps = 0;
```

With the value of `steps` set to `0` in the preceding code, there is no easing at all on camera motion, and recompiling the example with this value should make things feel much more snappy. Alternatively, recompiling with a value of `16` for the `steps` property creates the opposite effect, with all sudden movements of the mouse smoothed into a fluid camera motion.

Summary

In this chapter, we have covered essential topics such as the basics of 3D geometry and how to simplify its use, as well as the most recurrent Away3D classes that you will use in every project. If you have found things a bit of a struggle so far, don't worry! You are well on your way to mastering 3D in Flash.

Before you move on, make sure you can recall and feel comfortable with the following concepts and ideas:

- The view is the hub that connects the scene and the camera together.
    - The view is represented in Away3D by the `View3D` class, found in the `away3d.containers` package, and is an extension of the Flash display object `Sprite`.
    - The clipping classes located in the `away3d.core.clip` package define the view boundaries (the viewport) and crop the visible area of the scene. A new clipping object is applied to the view by resetting the `clipping` property of the `View3D` object.
    - The vanishing point of a view is controlled by its (x,y) position on the stage and can also represent the projected origin of the scene.
    - The view must be manually rendered for it to display anything. This is done using the `render()` method of the `View3D` object, typically once per frame.
- The scene is like a 3D display list, containing all renderable 3D objects.
    - The scene is represented in Away3D by the `Scene3D` class, found in the `away3d.containers` package.
    - 3D objects are added to and removed from the scene using `addChild()` and `removeChild()` methods, which are also available on `ObjectContainer3D` objects.
    - Moving, scaling or rotating a 3D container affects all contained children.
    - 3D containers can be used as pivots, to rotate a contained 3D object around a point other than its local origin.
- The camera serves as the point of observation used by the view.
    - `Camera3D`, `TargetCamera3D`, and `HoverCamera3D` are types of camera object in Away3D, and differ mainly in how their position and rotation properties are controlled.
    - Using the `lookAt()` method, a camera can center the view on any point in a scene such as the position of a 3D object or an arbitrary position vector.

The next chapter covers the different visual types of object encountered in Away3D, and explains how they are best used in your 3D projects.

Chapter 4

Primitives, Models, and Sprites

In the majority of Away3D projects, the contents of a scene can be split into three categories. **Primitives** are simple 3D geometric shapes (such as the `Cube` primitive shown in the previous chapter) generated internally by the engine from preset collections of properties. **Models** derive their geometry from imported 3D file formats such as `.dae`, `.3ds`, or `.obj` files that are created using 3D modeling software. **Sprites** in 3D are flat images that scale with distance but ignore rotation, as if they are constantly facing the camera.

You obviously don't need to know everything about the different content types to produce high-quality 3D content, but knowledge of what's possible is a great catalyst for creativity! A practical example can be found at the end of this chapter; it demonstrates how the topics covered here can be used together for maximum effect in a single project.

Knowing the basic terminology

The term "primitive" is a word commonly used in 3D graphics. In the majority of cases, including Away3D, it describes a simple three-dimensional geometric model such as a cube, sphere, or cylinder. The term is also used at times to speak of the most elementary visual part of a 3D object, such as a face or line segment, but in Away3D, we refer to these as "elements". **Elements** are visual surfaces or lines defined in 3D and have a form determined by the most fundamental of all 3D data, vertices.

Vertices

A **vertex** is a single point in space represented by an X, Y, Z vector that serves as the building block of a larger shape. It is an invisible entity by itself, but visual elements are constructed using several vertices grouped together. For example, two vertices define a line; three vertices define a triangle, and so on. When defining a surface in 3D, a triangle is the simplest (and most common) representation, although any number of vertices above this is allowed. In Away3D, lines and surfaces are represented by the classes

`Segment` and `Face` respectively, while vertices are represented by the `Vertex` class. All are found in the `away3d.core.base` package.

Faces and segments

Faces and segments are the two most common visual elements in Away3D. In other 3D applications, a surface that consists of more than three vertices will sometimes be referred to as a **polygon**, whereas a three-vertex surface will be called a **triangle**. These are valid terms, but they don't appear very often in Away3D. **Face** is the generic name used for a surface, and **segment** is the generic name used for a line. Each can be made up of any number of vertices and can, therefore, have any number of straight edges.

Where faces and segments differ is in their permitted drawing routines. Faces will be drawn as filled areas; segments will be drawn as connecting lines. This aspect is explored in more detail in Chapter 5.

Meshes and primitives

A **mesh** is a collection of vertices made visible by a collection of elements that use those vertices. As a saving measure, vertices can be reused by multiple elements in a mesh; it is rarely necessary to build elements with unique vertex points. A simple cube, for example, can have its shape defined by eight vertices, positioned at each corner. To create a solid object, these vertices are shared between twelve faces; six sides composed of two triangles each.

The cube belongs to a group of 3D objects called **primitives**. These are mesh-generating pieces of code that use simple geometric equations to produce their geometry. In Away3D, primitive objects are created using the classes found in the `away3d.primitives` package. Each primitive class controls the process of mesh generation via a series of properties that update the internal mesh configuration when changed.

Billboards and sprites

A **billboard** is a 2D element that can be positioned inside a 3D space. It contains no rotational information, so when rendered it appears to face the camera as a flat image that scales with distance. Consequently, it only requires one vertex point representing its position in the scene. Billboards belong to the family of 3D objects commonly referred to as **sprites**, although there are some fundamental differences between these and the sprites we are accustomed to in Flash. The basic type can be created with the `Sprite3D` class located in the `away3d.sprites` package.

3D sprites were heavily used in the early years of 3D gaming, because they can easily depict a complex shape without the need for excessive drawing operations. Many different types exist—billboards being the simplest form that can replace objects with **spherical symmetry** (i.e., objects that look roughly the same from all angles), such as particles, spheres, and clouds. More complex sprites allow different images to be used for different viewing angles (a technique seen in old-school first-person shooter games for drawing the enemies) or permit some rotation to represent objects with **axial symmetry** (objects that look roughly the same from a restricted set of angles) such as barrels or trees.

Setting up this chapter's base class

As preparation for the following code samples, let's set up the base class that we will extend to create the sample projects throughout this chapter. The `Chapter04SampleBase` class is created in the `flash3dbook.ch04` package and is written as follows:

```
package flash3dbook.ch04
{
  import away3d.cameras.*;
  import away3d.containers.*;

  import flash.display.*;
  import flash.events.*;

  public class Chapter04SampleBase extends Sprite
  {
    protected var _camera : HoverCamera3D;
    protected var _view : View3D;

    public function Chapter04SampleBase()
    {
      super();

      _createView();
      _createScene();
    }

    protected function _createView() : void
    {
      _camera = new HoverCamera3D();
      _camera.distance = 1000;
      _camera.tiltAngle = 10;
      _camera.panAngle = 180;

      _view = new View3D();
      _view.x = 400;
      _view.y = 300;
      _view.camera = _camera;
      addChild(_view);
      addEventListener(Event.ENTER_FRAME, _onEnterFrame);
    }

    protected function _createScene() : void
    {
      // To be overridden
    }

    protected function _onEnterFrame(ev : Event) : void
    {
      _camera.panAngle += (stage.mouseX - stage.stageWidth/2) / 100;
      _camera.hover();

      _view.render();
    }
  }
}
```

We start by importing the packages we will be using and then define the new class extending from `Sprite` (as is required with any document class). In the constructor, we call two methods: `_createView()` which sets up our basic view framework and the `_onEnterFrame()` handler method for rendering updates, and `_createScene()` which is where 3D content will be created and added to the scene. This is left blank in the `Chapter04SampleBase` class, to be overridden by our subsequent example classes.

Understanding common primitives

Some primitives are more commonly used than others. Countless sphere models of the earth are sprinkled across the Web as part of a variety of projects. Planes are probably used in even more places, serving as image cards in product carousels and pages in sites with creative 3D navigation. These objects are all easy to create using Away3D.

For our first look at primitives, we create a document class extending `Chapter04SampleBase`:

```
package flash3dbook.ch04
{
  import away3d.materials.*;
  import away3d.primitives.*;

  import flash3dbook.ch04.*;

  [SWF(width="800", height="600")]
  public class CommonPrimitives extends Chapter04SampleBase
  {

    public function CommonPrimitives()
    {
      super();
    }

    protected override function _createScene() : void
    {
      // Create default material
      var mat : WireColorMaterial = new WireColorMaterial(0xcccccc);
    }
  }
}
```

The preceding code sets up our SWF with a width of 800 and height of 600, overrides the `_createScene()` method to allow us to add our custom content, and creates a new `WireColorMaterial` object that we will apply as our default material to our newly created Away3D primitive objects.

The plane primitive

The plane primitive provides a natural step from two to three dimensions, thanks to its flat appearance. Its geometry consists of a 2D mesh in the shape of a square or rectangle. Because of this, planes are frequently used as a way of adding subtle 3D effects to an interface by projecting 2D content (such as images or text) onto the plane's surface as a material. The content can then be rotated in three dimensions, enabling some creative methods of display and interaction. This form of interface is commonly referred to as **postcards in space**, because the planes appear like individual postcards onto which 2D content is drawn.

Plane primitives are created using the `Plane` class from the `away3d.primitives` package. The `yUp` property defines whether the plane has its geometry built with its normal (the vector perpendicular to the plane's surface) pointing along the Y axis (`true`) or Z axis (`false`). The majority of Away3D primitives contain a `yUp` property in order to make this distinction. We can see the effect it has on a `Plane` object by adding the following lines to the `_createScene()` method.

```
var plane : Plane = new Plane();
plane.yUp = false;
plane.material = mat;
_view.scene.addChild(plane);
```

Here, we create a new plane primitive by instantiating a `Plane` object, configure its `yUp` and `material` properties, and add it to the scene. Notice that the plane's `material` property is set to the material instance we defined earlier. Try not to worry about what this is doing for now, as Chapter 5 will handle materials in more depth.

Compiling the `CommonPrimitives` example at this point will draw a front-facing plane with a grey outlined material. The plane's mesh is made up of two triangles by default. In some scenarios, you will want to subdivide the mesh into smaller triangles, for example, if you plan to deform the plane like a piece of paper. This type of subdivision, or segmentation, is achieved by using the `segmentsW` and `segmentsH` properties to subdivide along the width and height of the plane respectively. We can see the effects of subdivision by adding the following lines of code to the end of our `_createScene()` method from the previous sample:

```
plane.segmentsW = 10;
plane.segmentsH = 10;
```

Recompiling the `CommonPrimitives` example will display the result shown in Figure 4-1.

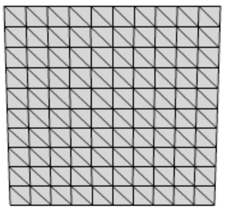


Figure 4-1. Plane with segmentsW and segmentsH both set to 10

The size of a plane is determined by the `width` and `height` properties of the `Plane` object. These are both set to 100 by default but can be updated in a similar manner to the `segmentsW` and `segmentsH` properties by adding the following lines of code to the end of our `_createScene()` method.

```
plane.width = 200;
plane.height = 200;
```

Recompiling the `CommonPrimitives` example will display the same plane shown in Figure 4-1, only this time at twice the width and height.

Back-face culling

Pause for a moment, run the code again, and roll over the Flash movie with your mouse to rotate the camera so that the plane is observed from behind. At this angle the plane turns invisible, because of an optimization technique known as **back-face culling**. Any face pointing away from the camera is ignored by default in the render process. The direction in which a face points is defined by a 3D vector known as the **normal** vector, which can be obtained in Away3D by using the getter property `normal` on the `Face` object. The normal vector is calculated as a vector perpendicular to the surface of the face. The direction the normal vector points is calculated such that if we were to observe the face by looking along its normal vector (so that it is pointing away from us), the vertices making up the face would be arranged in a counterclockwise order. It follows that if back-face culling is enabled on an object's mesh, a triangle prepared for drawing to screen with a clockwise ordering of its vertices will be omitted from the render process.

Back-face culling is usually a good thing, because most mesh objects (such as cubes and spheres) are built in such a manner that no faces are visible from their reverse side while viewing from outside the object. For example, you can't see the inside of a 3D box created by a cube primitive unless the camera is positioned inside the cube's mesh.

Planes are an exception to this rule, so you'll often want to turn off back-face culling for your plane objects. You can do this by setting the property `bothsides` to `true`.

```
plane.bothsides = true;
```

Add the preceding line of code to the end of our `_createScene()` method and recompile `CommonPrimitives` to see both sides of the plane when you rotate it with the mouse.

The cube primitive

The cube is another commonly used primitive and is created in Away3D using the `Cube` class found in the `away3d.primitives` package. To continue our `CommonPrimitives` example, add the following code to the end of the `_createScene()` method to create a cube and position it 200 units to the right:

```
var cube : Cube = new Cube();
cube.material = mat;
cube.x = 200;
_view.scene.addChild(cube);
```

To make room for the cube, add the following line of code to position the plane 200 units to the left:

```
plane.x = -200;
```

Recompiling the `CommonPrimitives` example displays the cube and plane primitives side by side. The cube uses similar properties as the plane for defining segmentation and size, adding a `depth` and `segmentsD` property for the extra depth dimension of the cube. Adding the following code to the end of the `_createScene()` method will subdivide the cube object and double its width, height, and depth dimensions:

```
cube.segmentsW = 10;
cube.segmentsH = 10;
cube.segmentsD = 10;
cube.width = 200;
cube.height = 200;
cube.depth = 200;
```

The sphere primitive

Sphere primitives are created in Away3D using the `Sphere` class, again found in the `away3d.primitives` package. The size of a sphere is not determined by any width, height or depth properties we have seen in previous primitives, but by a single property called `radius`. To demonstrate this, add the following lines of code to the `_createScene()` method.

```
var sphere : Sphere = new Sphere();
sphere.radius = 50;
sphere.material = mat;
_view.scene.addChild(sphere);
```

Recompiling the `CommonPrimitives` example will display the sphere in between the existing cube and plane objects. The sphere on the left in Figure 4-2 displays the rendered output. Here, we can instantly spot a problem with the sphere primitive, in that the small amount of faces making up the geometry causes it to look very blocky. To create a smoother looking sphere we need to use more faces. Luckily, this is possible by setting segmentation in the same way as the previous primitives, using `segmentsW` and `segmentsH` properties on the sphere primitive. Add the following lines of code to the `_createScene()` method.

```
sphere.segmentsW = 24;
sphere.segmentsH = 12;
```

Recompiling the `CommonPrimitives` example will display a sphere similar to the one on the right in Figure 4-2. In the preceding code, the values used for the `segmentsW` and `segmentsH` properties are not equal, as they were for the cube and plane primitives. This is because the construction for the sphere's geometry is different, with the overall distance between the first and last height segments half that of the distance between the first and last width segments (the latter of which would more commonly be referred to as the circumference). Taking this into account, the optimal configuration for an even segmentation of a sphere is to set half the amount of width segments compared to the amount of height segments.

It might be tempting to raise the `segmentsW` and `segmentsH` properties of the sphere to a very high number to achieve a perfectly smooth surface. However, this approach has a large effect on the amount of faces rendered each frame and thus impacts heavily on performance. There are other ways of making surfaces look smooth, as we will see when covering materials in chapter 5.

Figure 4-2. Default sphere (left) created with segmentsW = 8, segmentsH = 6, and sphere with higher level of detail (right), created with segmentsW = 24, segmentsH =12

Understanding wire primitives and line segments

So far, you have seen how to create solid objects composed of face elements in Away3D. But suppose we wanted to create a wireframe representation of a primitive, consisting of line segments forming the edges of a cube or edges of a plane? We have the ability to draw lines for the edges of faces as seen in the preceding example using the sphere primitive, but this is not ideal if the edges of our faces form polygons with more that three sides. We need a new way of constructing primitives in Away3D using segment elements. Happily, this can be accomplished using the **wire primitive** classes located in the same `away3d.primitives` package.

For our first look at wire primitives, we create a new document class extending `Chapter04SampleBase`.

```
package flash3dbook.ch04
{
  import away3d.materials.*;
  import away3d.primitives.*;

  import flash3dbook.ch04.*;

  [SWF(width="800", height="600")]
  public class CommonWirePrimitives extends Chapter04SampleBase
  {
    public function CommonWirePrimitives()
    {
      super();
    }
    protected override function _createScene() : void
    {
      // Create default material
      var mat : WireframeMaterial = new WireframeMaterial(0x000000);
    }
  }
}
```

In the preceding code, the default material is set as a `WireFrameMaterial` object, a material that extends the base class `SegmentMaterial` and is therefore compatible with segment-based objects.

Wireframe primitives

Most of our face-based primitives have equivalent segment-based primitives in the wireframe family. In Away3D, these wireframe primitives are created with classes named the same as regular primitives, but prefixed with the word "Wire". The following code is a reconstruction of the code added to the `_createScene()` method of the `CommonPrimitives` example in the previous section, with wireframe primitives in place of regular primitives:

```
var plane : WirePlane = new WirePlane();
plane.yUp = false;
plane.x = -200;
plane.width = 200;
plane.height = 200;
plane.material = mat;
_view.scene.addChild(plane);

var cube : WireCube = new WireCube();
cube.x = 200;
cube.width = 200;
cube.height = 200;
cube.depth = 200;
cube.material = mat;
_view.scene.addChild(cube);

var sphere : WireSphere = new WireSphere();
sphere.radius = 50;
sphere.segmentsW = 24;
sphere.segmentsH = 12;
sphere.material = mat;
_view.scene.addChild(sphere);
```

Adding this code to the end of the `_createScene()` method in the `CommonWirePrimitives` example and compiling will display the same three primitive objects as before, only this time rendered entirely with line segments using the `WireframeMaterial` object as their material.

Combining wireframe and regular primitives

A useful effect can be achieved by combining a regular primitive with its wireframe counterpart. Try positioning a `WireCube` and a regular `Cube` on top of each other by creating the following new document class extending `Chapter04SampleBase`.

```
package flash3dbook.ch04
{
  import away3d.materials.*;
  import away3d.primitives.*;

  import flash3dbook.ch04.*;

  [SWF(width="800", height="600")]
  public class CombinedWireAndRegularCube extends Chapter04SampleBase
  {
    public function CombinedWireAndRegularCube ()
    {
      super();
    }

  protected override function _createScene() : void
  {
    var wireCube : WireCube = new WireCube();
    wireCube.material = new WireframeMaterial(0x000000);
    _view.scene.addChild(wireCube);

    var regularCube : Cube = new Cube();
    regularCube.material = new ColorMaterial(0xcccccc);
    regularCube.scale(0.99);
    _view.scene.addChild(regularCube);
  }
}
```

Here, we are creating a `WireCube` object and `Cube` object at the same position in space. The wire cube uses the same `WireframeMaterial` object seen before to define the color of the segments in the `WireCube` primitive, while the regular cube uses a `ColorMaterial` object for defining the color of the faces used in the `Cube` primitive. Compiling the `CombinedWireAndRegularCube` example will display what appears to be a single cube with its edges outlined in black, as shown in Figure 4-3. The regular cube object is scaled to 0.99 of its original size by using the `scale()` method (available to any 3D object in Away3D). This is done to ensure the segments of the wire cube always overlay the faces of the regular cube. It is an amount large enough to influence the sorting order calculated by the Z-sorting algorithm but small enough to not be obviously visible in the scene.

Compare the output in Figure 4-3 with the previous rendering of a regular cube with a `WireColorMaterial` material in the `CommonPrimitives` example to see the visual difference between rendering with wire primitives and rendering outlined faces with regular primitives.

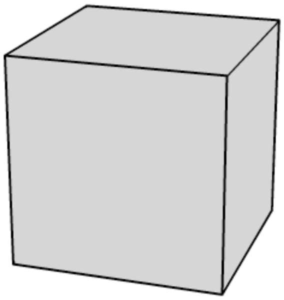


Figure 4-3. Using a combination of wire primitives and regular primitives, a cube with outlined edges can be constructed.

Drawing irregular lines in space

The segment element can be implemented as the building block of more user-defined geometry by using the `LineSegment` primitive class, also located in the `away3d.primitives` package. This creates a simple 3D line in space, by specifying the start and end point vectors. To experiment with this, let's create the following new document class extending `Chapter04SampleBase`:

```
package flash3dbook.ch04
{
  import away3d.core.math.*;
  import away3d.primitives.*;

  import flash3dbook.ch04.*;

  [SWF(width="800", height="600")]
  public class LinesInSpaceWithLineSegment extends Chapter04SampleBase
  {
    public function LinesInSpaceWithLineSegment ()
    {
      super();
    }

    protected override function _createScene() : void
    {
      var i : int, p1 : Number3D, p2 : Number3D, seg : LineSegment;

      p1 = new Number3D();
      p2 = new Number3D();

      for (i=0; i < 500; i++) {
        p2.x = (Math.random()-0.5) * 200;
        p2.y = (Math.random()-0.5) * 200;
        p2.z = (Math.random()-0.5) * 200;
```

```
                p2.add(p2, p1);

                seg = new LineSegment();
                seg.start = p1;
                seg.end = p2;
                _view.scene.addChild(seg);

                p1.clone(p2);
            }
        }
    }
}
```

Here, we create 500 `LineSegment` objects with `Number3D` objects used for the start and end positions of the line. It is ensured that the `start` property of the next `LineSegment` object is the same as the `end` property of the previous `LineSegment` object. This creates a continuous line moving between random points in space. Compiling the code should display the output shown in Figure 4-4.

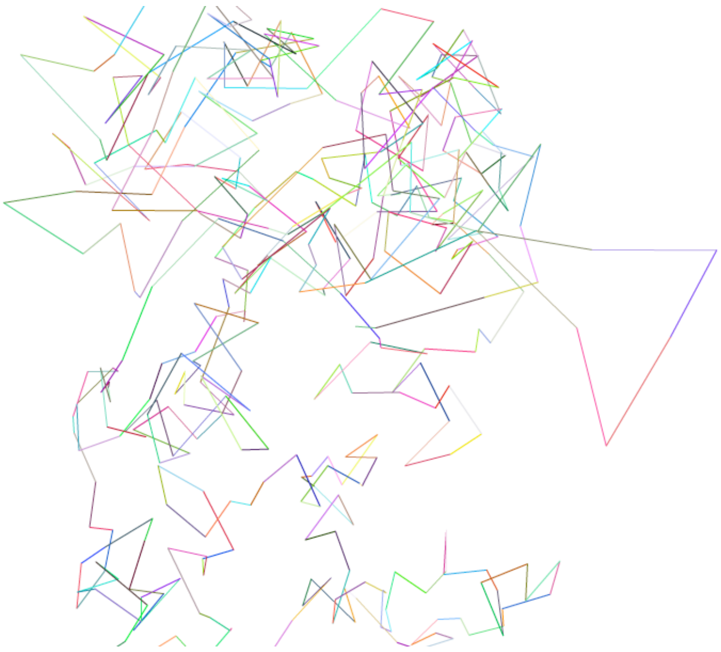


Figure 4-4. Randomly generated pattern from the LinesInSpaceWithLineSegment example using LineSegments

Using regular polygons

Earlier in this chapter, we talked about the plane primitive having a square or rectangular shape to its sides. This is a restriction of the `Plane` class in Away3D, but it is also possible to create a plane with any number of sides using the `RegularPolygon` class. Here, the resulting convex geometry has rotational symmetry, forming a mesh in the shape of a pentagon, hexagon, octagon, and so on. The number of sides is set by the `sides` property of the `RegularPolygon` object.

To explore the different results possible, let's create a new document class by extending `Chapter04SampleBase`:

```
package flash3dbook.ch04
{
  import away3d.materials.*;
  import away3d.primitives.*;

  import flash3dbook.ch04.*;

  [SWF(width="800", height="600")]
  public class PolygonsWithRegularPolygon extends Chapter04SampleBase
  {
    public function PolygonsWithRegularPolygon ()
    {
      super();
    }

    protected override function _createScene() : void
    {
      //create a pentagon
      _createPoly(5, 0xdddddd, -250);

      //create a dodecahedron
      _createPoly(12, 0x999999, 0);

      //create a circle
      _createPoly(100, 0x222222, 250);
    }

    protected function _createPoly(sides : int, color : int, x : Number) : void
    {
      var polygon : RegularPolygon;

      polygon = new RegularPolygon();
      polygon.sides = sides;
      polygon.material = new ColorMaterial(color);
      polygon.x = x;
      polygon.yUp = false;
      polygon.bothsides = true;
```

```
            _view.scene.addChild(polygon);
        }
    }
}
```

In the preceding code, creation of the various `RegularPolygon` objects has been broken out into a separate class method called `_createPoly()`, which accepts three arguments: `sides`, `color`, and `x`. Each `RegularPolygon` object is created inside this method using the `sides` argument for its number of sides, `color` argument for its material color, and `x` argument for its X axis position. Aside from these properties, each new object also has its `yUp` property set to `false` and `bothsides` property set to `true`, in much the same way our `Plane` object had in the `CommonPrimitives` example earlier in this chapter.

Using the `_createPoly()` method, three `RegularPolygon` objects are created with varying numbers of sides and varying material color. Compiling the example should display the image shown in Figure 4-5.

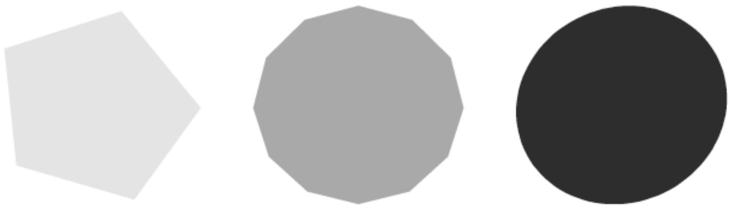


Figure 4-5. Three polygons created using the RegularPolygon class

Like the common primitives explored earlier, the `RegularPolygon` primitive has a line segment equivalent called `WireRegularPolygon`. To see it in action, replace the contents of the `_createPoly()` method with the following code:

```
var wirePolygon : WireRegularPolygon;

wirePolygon = new WireRegularPolygon();
wirePolygon.sides = sides;
wirePolygon.material = new WireframeMaterial(color);
wirePolygon.x = x;
wirePolygon.yUp = false;
wirePolygon.bothsides = true;

view.scene.addChild(wirePolygon);
```

Here, we have replaced all `RegularPolygon` objects with `WireRegularPolygon objects` and the `ColorMaterial` with a `WireframeMaterial`. The resulting geometry on compiling is identical to that shown in Figure 4-5, only now the primitives are drawn using segments instead of faces as their mesh elements.

Working with external models

Primitives are a good starting point for geometry creation in Away3D and can be extremely useful in creating basic 3D effects. But let's face it; they can only take us so far. For more advanced 3D scenes, such as those required in game development, the 3D content we want will require more complexity. In these cases, it is usually necessary to create the model geometry using software dedicated to the task and then save the result it in a 3D file format that Away3D can understand.

Luckily, Away3D supports most major 3D formats, making it compatible with virtually every piece of 3D modeling software available. Table 4-1 lists some of the major software packages next to the compatible export options available for use in Away3D. As well as importing files, there is also a neat way to embed models directly into an Away3D project, which is covered in the next section.

Table 4-1. Major 3D Modeling Software Titles Alongside Available Away3D File Export Options

Software	Away3D-Compatible Formats
Maya	OBJ and COLLADA (requires plug-in)
3ds Max	OBJ, 3DS, and COLLADA (requires plug-in)
SoftImage	OBJ, 3DS, and COLLADA
Cinema4D	COLLADA and 3DS
LightWave	OBJ and COLLADA
Blender	COLLADA, OBJ, 3DS, and ActionScript (requires plug-in)
Google SketchUp	KMZ, COLLADA, OBJ, and 3DS

Workflow when loading a model

The loading and parsing procedure for 3D files in Away3D is the same regardless of format. It consists of five steps:

1. Create a loader object.
2. Create a parser object.
3. Link the parser object to the loader object.
4. Set up event listeners on the loader object.
5. Define the file path of the object to be loaded and commence loading.

To maintain a level of continuity, Away3D uses an approach for loading 3D content similar to the display list's approach for loading 2D content with the native `Loader` class in Flash. Because the object that is being loaded is unavailable until the loading finishes, the actual loader object serves as a placeholder that can be added to the scene. Let's have a look at some familiar code when working with images in Flash.

```
var loader : Loader = new Loader();
loader.contentLoaderInfo.addEventListener(Event.COMPLETE, _onComplete);
loader.load(new URLRequest('image.jpg'))
addChild(loader);
```

This approach of triggering the download and adding the loader to the display list (regardless of whether the object has finished loading) is handled exactly same way in Away3D.

```
var loader : Loader3D = new Loader3D();
loader.addEventListener(Loader3DEvent.LOAD_SUCCESS, _onSuccess);
loader.loadGeometry("model.3ds", new Max3DS());
scene.addChild(loader);
```

The biggest difference arises from Away3D having separate parsing classes for interpreting the results of loading, while Flash automatically detects the content type and parses it accordingly. In the preceding Away3D example, the parser is defined as a new `Max3DS` object in an argument of the `loadGeometry()` call, which tells the engine to use the 3DS parser when reading the loaded file.

All parsers and loaders in Away3D are found in the `away3d.loaders` package. `Loader3D` is the default loader, acting in much the same way as the native `Loader` class. `LoaderCube` is a specialized loader class that acts as a 3D loading indicator that is visible throughout the loading process. Let's create an example using the `LoaderCube` object by extending the `Chapter04SampleBase` class and overriding the `_createScene()` method.

```
package flash3dbook.ch04
{
import away3d.containers.*;
import away3d.core.base.*;
import away3d.core.utils.*;

import away3d.events.*;
import away3d.exporters.*;
import away3d.loaders.*;

import flash.system.*;
import flash.events.*;

import flash3dbook.ch04.*;
  [SWF(width="800", height="600")]
  public class LoadingExternalModels extends Chapter04SampleBase
  {
  private var _loader:LoaderCube;
    public function LoadingExternalModels ()
    {
      super();
    }

    protected override function _createScene() : void
    {
    _loader = new LoaderCube();
```

```
      var url : String = '../../assets/ch04/monkey.3ds';

      _loader.addEventListener(Loader3DEvent.LOAD_SUCCESS, _onSuccess);
      _loader.loadGeometry(url, new Max3DS());
      _loader.scale(10);
      _view.scene.addChild(loader);
    }

    protected function _onSuccess(ev : Loader3DEvent) : void
    {
      trace('Finished loading!')
    }
  }
}
```

In the preceding code, the `monkey.3ds` model is obtained from the chapter resource files. This is a zip file containing all content used in the chapter and can be downloaded along with all other chapters resources for the book from the `downloads` section of `www.friendsofed.com`. Once you have the chapter files on your local machine, make sure that the file path written for the `url` property matches the path to your local file. Because `monkey.3ds` is being loaded at runtime, the path must be correct from the location of the compiled SWF file.

Compiling the example will display the loading indicator as it tracks loading progress. Once the model finishes loading, it replaces the loading indicator on screen. You should also see the trace output `Finished loading!` appear in the console window. Because loading a file from a location on your local machine is very fast, the `LoaderCube` object may not get a chance to display its loading indicator before the content completes loading. If you want a closer look at the `LoaderCube` object, you may have to upload the monkey model to a location online in order to slow down its loading progress. However, it is easy to see the `LoaderCube` object when a load fails, because it doesn't ever get replaced by the loading model. Typing an incorrect file path in the `url` property and recompiling will display a red cube with an error message printed on it. This visual state is taken by the `LoaderCube` object when a load fails for any reason and can be useful when debugging an Away3D application.

Optimizing external resources for size and speed

Few 3D file formats come optimized for use on the Web or for use with Flash. Some are not designed with limited bandwidth in mind, like the verbose COLLADA format. Others rely on binary decompression algorithms like ZIP that require extra processing to decompress, slowing down parsing times.

For these reasons, Away3D offers an export option that can process 3D model data already in Away3D's scene graph and output it as an ActionScript class. The process uses the `AS3Exporter` class located in the `away3d.exporters` package and requires only a few steps to carry out the conversion:

1. Load your 3DS, COLLADA, or similar model into Away3D using the `Loader3D` class.
2. Use the `AS3Exporter` class to convert the model data to ActionScript code, and paste the output to the clipboard.

3. Create a new class file in your project folder, and paste the contents of the clipboard into the class.
4. Recompile your project, this time creating the model by instantiating the newly created class, instead of loading a separate file.

Once this process has been carried out on your model files, you can easily embed your model in your SWF application file or compile a separate SWF that can then be loaded using the regular Flash `Loader` class. SWF files are compressed using ZIP and are very compact as a result, plus the decompression occurs natively in the Flash Player, so it's extremely fast! As well as this, parsing the model data is highly efficient because of the optimized methods contained in the created class file.

The open-source 3D modeling package Blender can export to Away3D-compatible ActionScript files directly, taking away the need for the first three steps in the `AS3Exporter` optimization process. This requires a plug-in script written by Dennis Ippel, available from www.rozengain.com.

Converting a model to ActionScript

As an example of the conversion process, let's now use the `AS3Exporter` class to create an ActionScript version of our `monkey.3ds` file by building on the previous `LoadingExternalModels` example. Add the following line of code to the end of the `_onSuccess()` method:

```
stage.addEventListener(MouseEvent.CLICK, _onClick);
```

Because we are copying text to the clipboard, the process needs to be triggered by user interaction. We therefore use a `CLICK` event handler to avoid a security error. In the preceding code, we have set up a listener for the mouse event using a handler called `_onClick()`. We now need to create the `_onClick()` method used to receive this event, by adding the following code to the end of the `LoadingExternalModels` class definition.

```
protected function _onClick(ev : MouseEvent) : void
{
        var exporter:AS3Exporter = new AS3Exporter();
        exporter.addEventListener(ExporterEvent.COMPLETE, _onComplete);
        exporter.export(_loader.handle, 'MonkeyMesh', 'flash3dbook.common');
}

protected function _onComplete(ev : ExporterEvent) : void
{
        trace('Export completed!');
        System.setClipboard(ev.data);
}
```

The `AS3Exporter` class is configured to produce an ActionScript file called `MonkeyMesh` that is contained in a package named `flash3dbook.ch04`. After executing the AS3Exporter with the `export()` method, the `_onComplete()` handler method is triggered, and the generated class string is extracted from the data property of the ExporterEvent object. This is then placed in the system clipboard, ready for you to paste into a newly created class file.

Using the converted model

In your ActionScript editor of choice, create a new class file called MonkeyMesh in the package flash3dbook.common, and paste in the contents of the clipboard (usually performed by the keyboard shortcut CTRL+V on Windows or CMD+V on Mac OS X). Note that what has been created by the AS3Exporter is a well-formed, ready-to-compile class definition using the name and package strings supplied in our previous _onClick() method definition.

To use the file in an Away3D project, simply instantiate the MonkeyMesh class, and add it to the scene as you would any other 3D object. In this case, only a simple mesh was exported, but there is no limit to the complexity of the output—we could have converted a container object with an entire scene enclosed if we had wanted.

With an ActionScript model, it is extremely easy to create copies of the model at runtime by instantiating the class over and over. Let's create an example using our newly generated MonkeyMesh object by extending the Chapter04SampleBase class and overriding the _createScene() method:

```
package flash3dbook.ch04
{
  import away3d.core.base.Mesh;
  import away3d.lights.*;
  import away3d.materials.*;

  import flash.display.*;
  import flash.events.*;
  import flash.net.*;
  import flash.utils.*;

  import flash3dbook.common.*;

  [SWF(width="800", height="600")]
  public class LoadingAS3Models extends Chapter04SampleBase
  {
    public function LoadingAS3Models ()
    {
      super();
    }

    protected override function _createScene() : void
    {
      var i : int;
      var material : ShadingColorMaterial = new ShadingColorMaterial(0x888888);
      for (i=0; i < 5; i++) {
        var monkeyMesh : MonkeyMesh = new MonkeyMesh();
        monkeyMesh.material = material;
        monkeyMesh.x = (Math.random()-0.5) * 400;
        monkeyMesh.y = (Math.random()-0.5) * 400;
        monkeyMesh.z = (Math.random()-0.5) * 400;

        _view.scene.addChild(monkeyMesh);
      }

      var light : PointLight3D = new PointLight3D();
      light.x = 300;
      light.z = -400;
      light.y = 500;
      _view.scene.addLight(light);
    }
  }
}
```

Notice how each `monkeyMesh` model is created by instantiating a new `monkeyMesh` object inside a `for` loop. Compiling the code will create five randomly spaced `MonkeyMesh` models with a grey `ShadingColorMaterial` applied, as shown in Figure 4-6.

> *As an alternative to the method of using the `AS3Exporter` class described here, there is an Adobe AIR utility, called Prefab3D, available from the Away3D website that will let you create ActionScript model classes by simply dragging and dropping model files.*

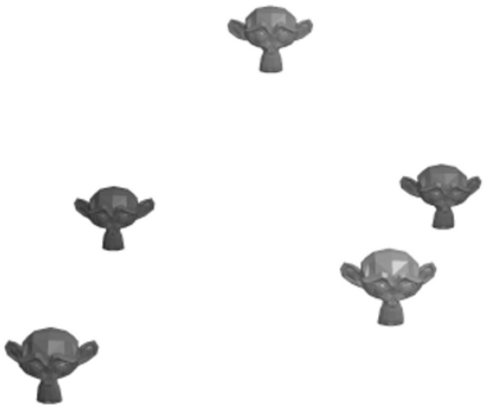


Figure 4-6. Five monkey meshes created with the ActionScript class MonkeyMesh and positioned randomly in the scene

Creating a library of models

When using a large number of ActionScript models, it is a good idea to embed them in a separate SWF file to get the best of both worlds: a quick initial load combined with small, efficient model files loaded incrementally. The idea of using a separate SWF file as an asset library is certainly not unique to the world of 3D, but it is a concept that fits perfectly with the use of 3D ActionScript models.

When loading an external SWF file in Flash with the native `Loader` class, all assets embedded within that SWF are made available to the running application. This includes graphics and sounds, as well as any ActionScript classes defined by the SWF's source code. Let's create a library SWF file containing our model and then use it in our `LoadingAS3Models`example.

Create the new document class `MyLibraryClass` in the package `flash3dbook.ch04` with the following code:

```
package flash3dbook.ch04
{
  import flash3dbook.common.*;

  import flash.display.*;

  public class MyLibraryClass extends MovieClip
  {
    // Force mesh classes to be included in the SWF
    private var mesh01 : MonkeyMesh;
  }
}
```

Compile this into our model library SWF file called `assets.swf`. Obviously, a library SWF such as this would normally contain more than one model! We can now use this in our `LoadingAS3Models` example by replacing the `_createScene()` method with the following code:

```
protected override function _createScene() : void
{
  var loader : Loader = new Loader();
  loader.contentLoaderInfo.addEventListener(Event.COMPLETE, _onComplete);
  loader.load(new URLRequest('assets.swf'));
}
```

Once the load is complete, any classes in the loaded SWF will be available to the main application. In the preceding code, we have defined an event handler for the `COMPLETE` event called `onComplete()`. We now need to create this method by adding the following code to the end of the `LoadingAS3Models` class definition.

```
private function _onComplete(ev : Event):void
{
  var MClass : Class;
  MClass = getDefinitionByName('flash3dbook.common.MonkeyMesh') as Class;
  var monkeyMesh : Mesh = new MClass();
  _view.scene.addChild(monkeyMesh);
}
```

To avoid errors at compile time (and to avoid inadvertently compiling `MonkeyMesh` with our document class), we need to extract the `MonkeyMesh` class definition using the Flash utility function `getDefinitionByName()`. This method takes the string supplied in its argument and returns the class or object instance defined by that string. Calling `getDefinitionByName()` here returns a reference to the `MonkeyMesh` class and stores it in the `MClass` variable. We then invoke the `MClass` constructor to create a new instance of the `MonkeyMesh` class, which can be cast to `Mesh` (its inherited object type) and added to the scene. Compiling the code will display the same result as before, but without the `LoaderCube` placeholder because, in this case, the loading is handled by the native `Loader` class.

Applying bitmap filter effects to 3D objects

Bitmap filter effects (blurs, glows, drop shadows, etc.) that you are familiar with applying to display list objects in Flash will work just as well applied to scene graph objects in Away3D. In fact, the way they are applied is identical to display list objects, and the exact same filter classes from the `flash.filters` package can be used. Let's create an example to demonstrate this feature:

```
package flash3dbook.ch04
{
  import away3d.primitives.*;
  import away3d.materials.*;

  import flash3dbook.ch04.*;

  import flash.filters.*;

  [SWF(width="800", height="600")]
  public class BitmapFilterModels extends Chapter04SampleBase
  {
    public function BitmapFilterModels ()
    {
      super();
    }

    protected override function _createScene() : void
    {
      var cube : Cube = new Cube();
      cube.material = new ColorMaterial(0xcccccc);
      cube.filters = [ new GlowFilter(0) ];
      cube.ownCanvas = true;
      _view.scene.addChild(cube);
    }
  }
}
```

Compiling the preceding code will display the output shown in Figure 4-7. As well as applying filters in an array object using the familiar `filters` property, we need to explicitly tell Away3D to render the 3D object in a separate `Sprite` object using the `ownCanvas` property. When `ownCanvas` is set to `true`, a designated `Sprite` object is created to act as a wrapper for the visible output of the 3D object, and the filter array is automatically applied to this wrapper to render the contained filter effects.

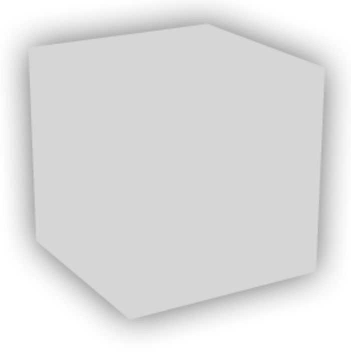


Figure 4-7. A Cube with the ownCanvas property set to true and a black-colored GlowFilter applied

The disadvantage of using the `ownCanvas` property to enable filter effects on 3D objects is the limitation it imposes on Z sorting. With the visible output of an object grouped into a `Sprite` container, we lose the ability to individually sort each face (or segment) element of the object with other object's elements, restricting the Z-sorting algorithm to only deal with the object as a single sorted entity. This can lead to unwanted sorting artifacts if, for example, two 3D objects exist as two intersecting meshes that require different sorting depths across their elements. The solution is to only apply `ownCanvas` to objects that need it, splitting objects that could cause sorting problems into separate meshes that make more sense to the Z-sorting algorithm.

Using 3D sprites

As mentioned earlier, when dealing with 3D, the term sprite generally refers to a 3D object represented by a 2D image projected onto a flat plane. The most basic type of sprite is a billboard (or spherical) sprite, which operates by ensuring its 2D image is always rendered facing the camera. Sprites are regularly used for optimizing the rendering process, by faking the appearance of more complex objects that look essentially the same from all angles.

Examples of 3D objects that can potentially take advantage of the sprite approach to rendering are trees and smoke. Trees are typically implemented using cylindrical sprites that turn to face the camera around a restricting axis, because trees can be considered to have axial symmetry. Smoke can be created using spherical sprites that always face the camera, because a cloud of smoke can be considered to have spherical symmetry.

Creating smoke using 3D sprites

There are many complex ways to create realistic smoke effects using particle engines, but in applications where performance is of extreme importance (such as computer games and real-time 3D on the Web), scattering large spherical sprites is perhaps the most popular method of faking it.

A basic 3D sprite in Away3D is created using the `Sprite3D` class located in the `away3d.sprites` package and uses a material object to define its 2D image. There is also a 3D sprite variant called `MovieClipSprite` that uses a native `Sprite` object as its 2D image; this has some advantages over

Sprite3D, such as faster animation and interaction, but comes with the restriction of allowing only one MovieClipSprite per Sprite source, and when using Flash 9, only in spherical sprite mode. Since the puffs of smoke in our example will be static, we'll use the more applicable Sprite3D class.

Let's begin by creating a sample class that create a large number of sprites and scatters them randomly around the scene origin.

```
package flash3dbook.ch04
{
  import away3d.sprites.*;
  import away3d.materials.*;

  import flash.display.*
  import flash.filters.*;
  import flash.geom.*;

  [SWF(width="800", height="600")]
  public class SmokeWithSprites extends Chapter04SampleBase
  {
    private var _material : BitmapMaterial;

    public function SmokeWithSprites ()
    {
      super();
    }

    protected override function _createScene() : void
    {
      _createMaterial();
      var i : int;

      for (i=0; i<50; i++) {
        var sprite : Sprite3D = new Sprite3D(_material);

        sprite.x = (Math.random()-0.5) * 200;
        sprite.y = (Math.random()-0.5) * 200;
        sprite.z = (Math.random()-0.5) * 200;

        _view.scene.addSprite(sprite);
      }
    }

    protected function _createMaterial() : void
    {
      var bmp : BitmapData = new BitmapData(100, 100, false,↵
0xffffff*Math.random());
      _material = new BitmapMaterial (bmp);
    }
  }
}
```

Notice that in our _createScene() method, we use a similar technique to the one in the LoadingAS3Models example, with a for loop creating objects and assigning them random positions. In this case, the objects are 3D sprites. One notable difference is that instead of using the addChild() method to include our 3D sprites in the scene, we use an addSprite() method.

Compiling the preceding code will display 50 randomly colored 100 × 100–pixel squares, scattered randomly within a 100-unit cube area centered around the scene's origin. Moving our mouse over the scene will rotate the cloud of sprites to show you that they are indeed positioned in 3D space. Sprites further away from the camera will appear smaller than sprites closer to the camera, but they still feel very 2D because a flat color is hardly representative of an object with spherical symmetry.

The next step is to create a texture for the sprites that resembles a puff of smoke. To achieve this, replace the contents of the _createMaterial() method in the SmokeWithSprites example with the following code:

```
var puff : Shape = new Shape();
var dia : Number = Math.random() * 40 + 30;
puff.graphics.beginFill(0xcccccc, Math.random());
puff.graphics.drawEllipse(-dia/2, -dia/2, dia, dia);

var bmp : BitmapData = new BitmapData(100, 100, true, 0);
bmp.draw(puff, new Matrix(1, 0, 0, 1, 50, 50));

var blur : BlurFilter = new BlurFilter(32, 32, 2);
bmp.applyFilter(bmp, bmp.rect, new Point, blur);

_material = new BitmapMaterial (bmp);
```

Here, the smoke material is created by first drawing a circle using the ActionScript drawing API. The circle has a diameter of anywhere between 40 and 70 pixels and is colored gray with a random level of transparency. The circle is then drawn into a bitmap data object, and a blur effect is applied to give it a hazy look.

Recompiling the SmokeWithSprites example now generates 50 randomly sized puffs of smoke, scattered across the same 3D space as before. The overall effect should be one of a 3D smoke cloud that can be rotated with the mouse to view it from all angles. Figure 4-8 depicts the result.

Figure 4-8. Smoke effect created using 3D sprite objects

Because 3D sprites are a general technique for the fast rendering of complex objects, we could increase the complexity of the effect in the `SmokeWithSprites` example to create a greater sense of realism, without running into too many performance problems. For example, the cloud could be animated or have filter effects applied to produce something even more abstract.

Tutorial: Creating a twisted image gallery

At this stage, you are hopefully starting to feel confident about creating 3D objects in Flash. But so far, we have yet to build a practical application. For this, we need to look at creating a more complex example.

One very typical use for Flash on the Web is as an image gallery interface. There are many different ways to implement this type of interface in Flash, so to stand out from the crowd, we will need one that is a bit…. twisted. This project pulls together several of the topics we have covered in this chapter, so it should serve as a nice recap. We will use the following features and workflows in the production of our image gallery:

- The hover camera (to navigate our way around the scene)
- The plane primitive (to project our gallery images into 3D)
- ActionScript models (to import our custom geometry)
- Billboards (to optimize the render process)
- A 3D container object (to move several objects at once)

The gallery images will be displayed inside TV screens, which are stacked on top of each other and randomly rotated around the Y axis. The user navigates the image gallery with a simple menu built from Flash text fields or using the mouse to spin the hover camera around, panning up and down through the images. Have a look at Figure 4-9 for a sneak preview of the completed application.

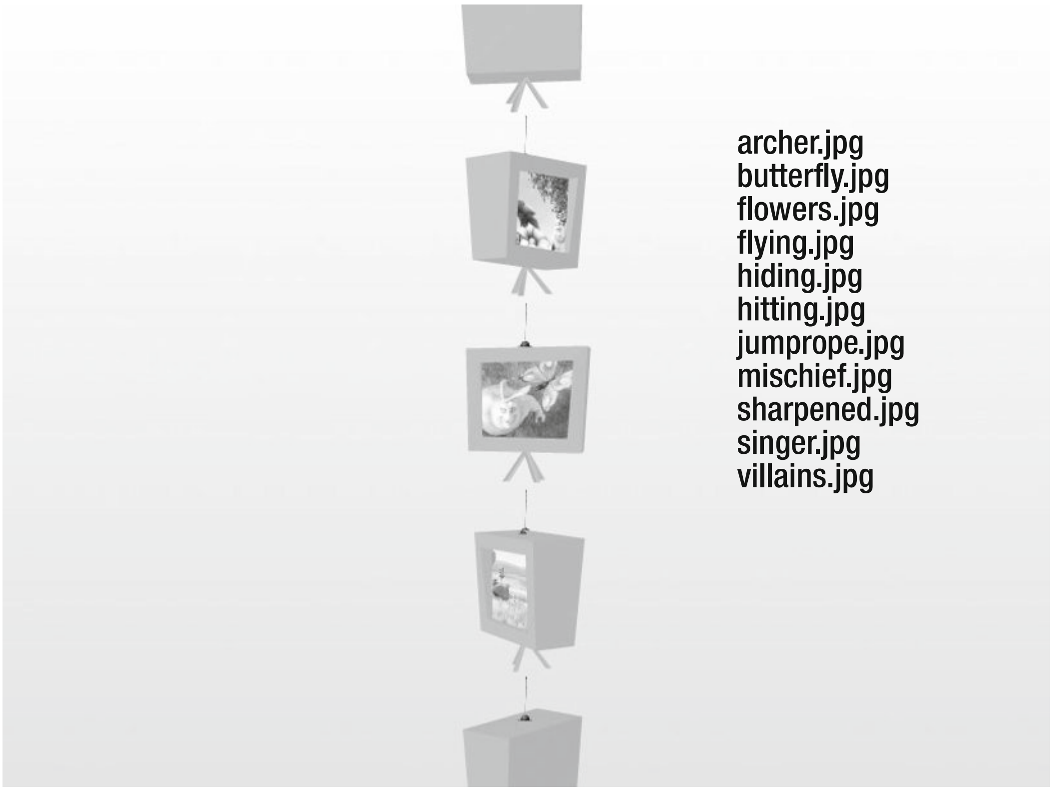


Figure 4-9. The image gallery built in this tutorial

Laying out the application shell

The application is built from four classes, all located in the `flash3dbook.ch04.tutorial` package:

- `TwistedImageGallery`: The main application class that is responsible for creating all content, using an array of images names
- `TVBox`: Represents a single TV object that groups the elements making up a single TV into an object container
- `TVBoxMesh`: The mesh data of the TV model as an ActionScript class, created using the `AS3Exporter` class
- `ImageMenuItem`: A single menu item that holds a reference to its associated TV container and handles basic user interaction, such as rollover effects

Let's begin by looking at the document class `TwistedImageGallery`, which is responsible for setting up all content, including the Away3D view and text-based menu:

```
package flash3dbook.ch04.tutorial
{
  import away3d.cameras.*;
  import away3d.containers.*;
  import away3d.core.base.*;
  import away3d.lights.*;
  import away3d.primitives.*;

  import flash3dbook.ch04.tutorial.*;

  import flash.display.*;
  import flash.events.*;
  import flash.geom.*;
  import flash.filters.*;

  [SWF(width="800", height="600")]
  public class TwistedImageGallery extends Sprite
  {
    private var _view : View3D;
    private var _camera : HoverCamera3D;

    public function TwistedImageGallery()
    {
      super();

      _createScene();
      _createContent();
    }

    private function _createScene() : void
    {
      _camera = new HoverCamera3D();
      _camera.distance = 600;
      _camera.tiltAngle = 0;
      _camera.steps = 4;

      _view = new View3D();
      _view.x = 400;
      _view.y = 300;
      _view.camera = _camera;
      addChild(_view);

      addEventListener(Event.ENTER_FRAME, _onEnterFrame);

      // Light positioned up and back, to shade the scene
      var light : PointLight3D = new PointLight3D();
      light.ambient = 0.8
      light.position = new Number3D(1000, 500, -1000);
      _view.scene.addLight(light);

      // Beautiful gradient background
      var matrix : Matrix = new Matrix();
      matrix.createGradientBox(800, 600, Math.PI/2);
      graphics.beginGradientFill(GradientType.LINEAR,↩
[0xffffff, 0xdddddd], [1,1], [0, 0xff], matrix);
      graphics.drawRect(0, 0, 800, 600);
    }


    private function _createContent() : void
    {
    }

    private function _onEnterFrame(ev : Event) : void
    {
      if (stage.mouseX < 160 || stage.mouseX > 640)
        _camera.panAngle += (stage.mouseX - stage.stageWidth/2) / 60;

      _camera.hover();
      _view.render();
    }
  }
}
```

The preceding code defines the shell of the `TwistedImageGallery` class. On instantiation, the constructor calls two initializing methods, `_createScene()` and `_createContent()`. The `_createContent()` method currently exists as a stub that will be added to later. Inside the `_createScene()` method, we have the basic code for setting up our view.

First, a hover camera is created and has its `distance`, `tiltAngle`, and `steps` properties initialized to what we want for the camera movement. Next, the view is created and positioned at the center of the stage, with its camera property set to our newly created `HoverCamera3D` object.

As we have come to expect, an `ENTER_FRAME` listener is created to allow us to update the hover camera position and invoke the `render()` method of the view on every frame of the Flash movie. The handler method `_onEnterFrame` rotates the camera relative to the x coordinate of the mouse, with some restrictions on maximum and minimum rotation to keep the motion under control.

Next, a light source is added to the scene in the form of a `PointLight3D` object that will be used by any shading materials we assign to a mesh. Lights and shading materials are covered in greater detail in Chapter 5.

Finally, a subtle gradient is drawn across the entire stage background using the standard ActionScript drawing API, to add some variation to the background of the scene.

Creating the TV sets

Now, let's prepare the external model to be used for the 3D TV set. We want to use an ActionScript class of the mesh, which can be created with the `tvbox.3ds` file downloaded in the chapter resource files mentioned earlier. Conversion can be done using the `AS3Exporter` class, the Blender export scripts, or the export utility from `www.away3d.com`. In this case, we'll take a look at the `AS3Exporter` approach, which requires a modification to our previously created `LoadingExternalModels` example to convert our `tvbox.3ds` file.

If you have not already created the `LoadingExternalModels` example from earlier in this chapter, you can skip this step by using the ready-made `TVBoxMesh` file supplied with the chapter download files, available from the downloads section of `www.friendsofed.com`.

In the `_createScene()` method of our `LoadingExternalModels` example, replace the `url` variable definition with the following code:

```
var url : String = 'http://flash3dbook.com/files/chapter4/tvbox.3ds';
```

We also need to adjust the output name of the ActionScript file, which can be done by replacing the contents of the `_onClick()` method with the following code:

```
new AS3Exporter(loader.handle, 'TvBoxMesh', 'flash3dbook.ch04.tutorial');
```

Recompiling the `LoadingExternalModels` example imports the `tvbox.3ds` file into the Flash movie. Once it is loaded, click the mouse anywhere on the stage to copy the ActionScript class definition `TVBoxMesh` to the clipboard. In your ActionScript editor, create a new class called `TVBoxMesh` in the package `flash3dbook.ch04.tutorial`, and paste in the clipboard contents. Save the file, and you're ready to use the `TVBoxMesh` class in the application.

Next, we need to create the container for the elements of a single TV item in the gallery, called `TVBox`. This class extends `ObjectContainer3D`, which allows us to group several elements in a scene simply by adding them as children of the container. Wrapping 3D objects in this manner is a common practice when creating complex, reusable 3D assets. We'll start by creating the shell of the `TVBox` class.

```
package flash3dbook.ch04.tutorial
{
  import away3d.containers.*;
  import away3d.materials.*;
  import away3d.primitives.*;

  import flash.display.*;
  import flash.events.*;
  import flash.net.*;

  public class TVBox extends ObjectContainer3D
  {
    public function TVBox(imageUrl : String)
    {
      super();
      _createChildren();
      _loadImage(imageUrl);
    }

    private function _createChildren() : void
    {
    }

    private function _loadImage(url : String) : void
    {
    }
  }
}
```

As you can see, the constructor takes one parameter—the URL string of the image that the TV is to display. This is passed to the `_loadImage` method, which currently exists as a stub that we will fill out shortly. Before that, the constructor calls the `_createChildren` method, inside which we wish to create the visual assets of the `TVBox` class from three pieces:

- The TV box and stand are both parts of the mesh geometry encoded in the `TVBoxMesh` ActionScript class, which is built automatically on instantiation.
- The TV picture is the gallery image, made from a simple plane primitive configured with a bitmap material.
- The TV antenna is a flat PNG, drawn using a `Sprite3D` object and positioned on top of the TV box.

Starting with the creation of the TV box and stand, we add the following code to the `_createChildren()` method:

```
var tv : TVBoxMesh = new TVBoxMesh();
tv.material = new ShadingColorMaterial(0xcccccc);
tv.scale(30);
addChild(tv);
```

An instance of the `TVBoxMesh` class is created, and its material property is set to a new `ShadingColorMaterial` object. We then adjust its scaling to 30 times the default, because the original model was created in a modeling application with a world scale different from the one used here.

Next, we create a plane primitive representing the TV screen, onto which the loaded image will be mapped. Because we want to have easy access to the plane both now and later when the image finishes loading, the first thing we do is to create a global class variable inside the definition of the `TVBox` class, that will hold the plane primitive instance.

```
private var _image : Plane;
```

With this done, we can now create the plane primitive by adding the following code to the `_createChildren()` method:

```
_image = new Plane();
```

```
_image.yUp = false;
_image.width = 50;
_image.height = 50;
_image.x = 2;
_image.z = -10;
_image.pushback = true;
_image.material = new ColorMaterial(0x000000);
addChild(_image);
```

The plane is built facing forward rather than up (thanks to yUp being set to false) and is placed in a suitable position with regard to the geometry of the TVBoxMesh object. For now, it is uses a black color material, but this will be reset once our gallery image is loaded. By setting the `pushback` property to true, we ensure that the image is never drawn on top of the TV mesh. This could have been a problem when the TV is rendered from behind, but because backface culling will render the plane invisible from those angles, we can safely use this method to prevent Z-sorting artifacts from faces being too close to each other.

Finally, we build the antenna for the TV using the image `antenna.png`, which is distributed inside the same chapter download files mentioned before. We embed the image into the SWF by adding an ActionScript `[Embed]` meta-tag inside the definition of the `TVBox` class.

```
[Embed('../../../../assets/ch04/tutorial/antenna.png')]
private var AntennaBitmap : Class;
```

This will assign a bitmap asset definition to the `AntennaBitmap` variable as if it were a regular class definition. Instantiating the class referenced by this variable will create a new `BitmapAsset` object— a class definition that extends the standard `Bitmap` class in Flash.

The `[Embed]` meta-tag is available for use in the Flex SDK and can be used with the most recent Flash editors, including Flash Professional CS4 and CS5. Flash Professional CS3, however, does not support this syntax. To compile using CS3, import the image as a library item and set its linkage class name to `AntennaBitmap`. This approach achieves essentially the same outcome as using `Embed` in the previous code.

We can now create the 3D sprite for the antenna by adding the following code to the `_createChildren` method:

```
var bmp : BitmapData = Bitmap(new AntennaBitmap()).bitmapData;
var spriteMaterial : BitmapMaterial = new BitmapMaterial(bmp);
spriteMaterial.smooth = true;
var antenna : Sprite3D = new Sprite3D(spriteMaterial);
antenna.scaling = 0.15;
antenna.y = 40;
addSprite(antenna);
```

This creates a new bitmap material using the bitmap data extracted from an instance of the `AntennaBitmap` class and uses it as the material definition of a new 3D sprite object. The material has smoothing enabled to keep the bitmap from looking pixilated, and the antenna is scaled and positioned to align neatly with the top of the `TVBoxMesh` object.

Loading the gallery image

To handle the loading of the gallery images, we need to add the following code to the `_loadImage` method of the `TVBox` class:

```
var loader : Loader = new Loader();
var info : LoaderInfo = loader.contentLoaderInfo;

info.addEventListener(Event.COMPLETE, _onImageComplete);
info.addEventListener(IOErrorEvent.IO_ERROR, _onImageError);
loader.load(new URLRequest(url));
```

This defines a new native `Loader` object that takes the `url`argument of the method and creates a new `URLRequest` object for the loader. Before the load is triggered, two handler functions, `_onImageComplete()` and `_onImageError()`, are set to trigger from `COMPLETE` and `IO_ERROR` events dispatching from the loader. We now need to create these functions by adding the following code to the end of the `LoadingExternalModels` class definition:

```
private function _onImageComplete(ev : Event) : void
{
  var info : LoaderInfo = ev.currentTarget as LoaderInfo;
  var bmp : BitmapData = Bitmap(info.loader.content).bitmapData;
  var imageMaterial : BitmapMaterial = new BitmapMaterial(bmp);
  imageMaterial.smooth = true;
  _image.material = new BitmapMaterial(bmp);
}
private function _onImageError(ev : Event) : void
{
  trace("Error loading image");
}
```

In the preceding `_onImageComplete` method, the `BitmapData` object is retrieved from the loaded bitmap and used to create a new bitmap material for the `_image` plane. The `_onImageError` method traces an error message to the output window so that we are notified if the application has had a problem loading its images.

Creating the menu items

So far, we have yet to see the result of the `TVBox` class created in the last section. Before we start to piece everything together in the document class, we need to create one more subclass that represents an item in the navigation menu. This is constructed as an extension of the `Sprite` class, with a text field contained within and some very simple mouse interaction.

```
package flash3dbook.ch04.tutorial
{
  import flash.display.*;
  import flash.events.*;
  import flash.text.*;
  import flash.filters.DropShadowFilter;
  import flash.utils.getTimer;

  public class ImageMenuItem extends Sprite
  {
    private var _tv : TVBox;
    private var _tf : TextField;

    public function ImageMenuItem(str : String, tv : TVBox)
    {
      _tv = tv;
      _createText(str);
    }

    private function _createText(str : String) : void
    {
      _tf = new TextField();
      _tf.defaultTextFormat = new TextFormat('Arial', 11);
      _tf.autoSize = TextFieldAutoSize.LEFT;
      _tf.text = str;
      _tf.selectable = false;
      _tf.mouseEnabled = false;

      addChild(_tf);
      addEventListener(MouseEvent.MOUSE_OVER, _onMouseOver);
      addEventListener(MouseEvent.MOUSE_OUT, _onMouseOut);
    }

    public function get tv() : TVBox
    {
      return _tv;
    }

    private function _onMouseOver(ev : MouseEvent) : void
    {
      _tf.textColor = 0x666666;
    }

    private function _onMouseOut(ev : MouseEvent) : void
    {
      _tf.textColor = 0;
    }
  }
}
```

The class constructor requires two arguments: one for the text displayed by the menu item and the other for the instance reference of the `TVBox` container representing the gallery item. First, we save the reference to the TVBox instance as a local variable. Next, the _createText() method is called; it creates a new TextField object and adds it to the display list. Event listeners for MOUSE_OVER and MOUSE_OUT events are added, which change the color of the text to gray when the over event is triggered and back to black when the out event is triggered.

Now that we have our TVBox and ImageMenuItem classes defined, we can glue everything together with the TwistedImageGallery document class.

Displaying the content

To start testing the visual output of what we have been building, we need to fill the empty _createContent method with the code that will instantiate both the TVs and the menu items.

The TV objects will be added to the scene within a single container, allowing us to simultaneously pan all 3D content up and down through the gallery images. Because this container needs to be accessed from several methods, we create it in a global variable added to the TwistedImageGallery class definition.

```
private var _pivot : ObjectContainer3D;
```

The _pivot container instance is created along with the rest of the application content by adding the following code to the _createContent method:

```
var i : int;
var last_angle : Number = 0;
var images : Array = [
  'archer.jpg',
  'butterfly.jpg',
  'flowers.jpg',
  'flying.jpg',
  'hiding.jpg',
  'hitting.jpg',
  'jumprope.jpg',
  'mischief.jpg',
  'sharpened.jpg',
  'singer.jpg',
  'villains.jpg'
];

_pivot = new ObjectContainer3D();
_view.scene.addChild(_pivot);

for (i=0; i < images.length; i++) {
  var url_base : String = '../../../../assets/ch04/tutorial/';
  var tv : TVBox = new TVBox(url_base + images[i]);
  tv.y = -i*100;
  tv.rotationY = last_angle + Math.random() * 90 + 45;
  last_angle = tv.rotationY;
  _pivot.addChild(tv);

  var item : ImageMenuItem = new ImageMenuItem(images[i], tv);
  item.x = 550;
  item.y = i*18 + 140;
  item.buttonMode = true;
  item.addEventListener(MouseEvent.CLICK, _onClickMenuItem);
  addChild(item);
}
```

In the preceding code, each entry in the `images` array has corresponding `TVBox` and `ImageMenuItem` objects created. The `TVBox` objects are added to the `_pivot` container, and the `ImageMenuItem` objects directly to the stage.

> *For the purposes of this example, the file names of the images have been hard-coded straight into an array in the application source. In a real-world application, it may be more useful to make this data source configurable using some external file definition, such as an XML document. Note that your path in the url_base variable may differ from the one that we have here. Make sure that you enter the correct path to your tutorial files*

The TVs are oriented somewhat randomly by rotating each model between 45 and 90 degrees relative to the previous one, around the Y axis. The URL used to load each image is prefixed with a base URL pointing to the location of the local sample files.

The menu items are positioned on the stage as a simple vertical list. Each item has a listener function called `_onClickMenuItem()` that is set to trigger from a `CLICK` event. To allow us a test compile at this point with no errors, we can create an empty `_onClickMenuItem()` method by adding the following code to the end of the `TwistedImageGallery` class definition.

```
private function _onClickMenuItem(ev : MouseEvent) : void
{
}
```

Compiling the class will display the result shown in Figure 4-10. Moving the mouse left and right over the stage rotates the column of TVs left and right. To complete the gallery, we need to add code to the inside of the `_onClickMenuItem()` method that will allow us to navigate between each gallery item.

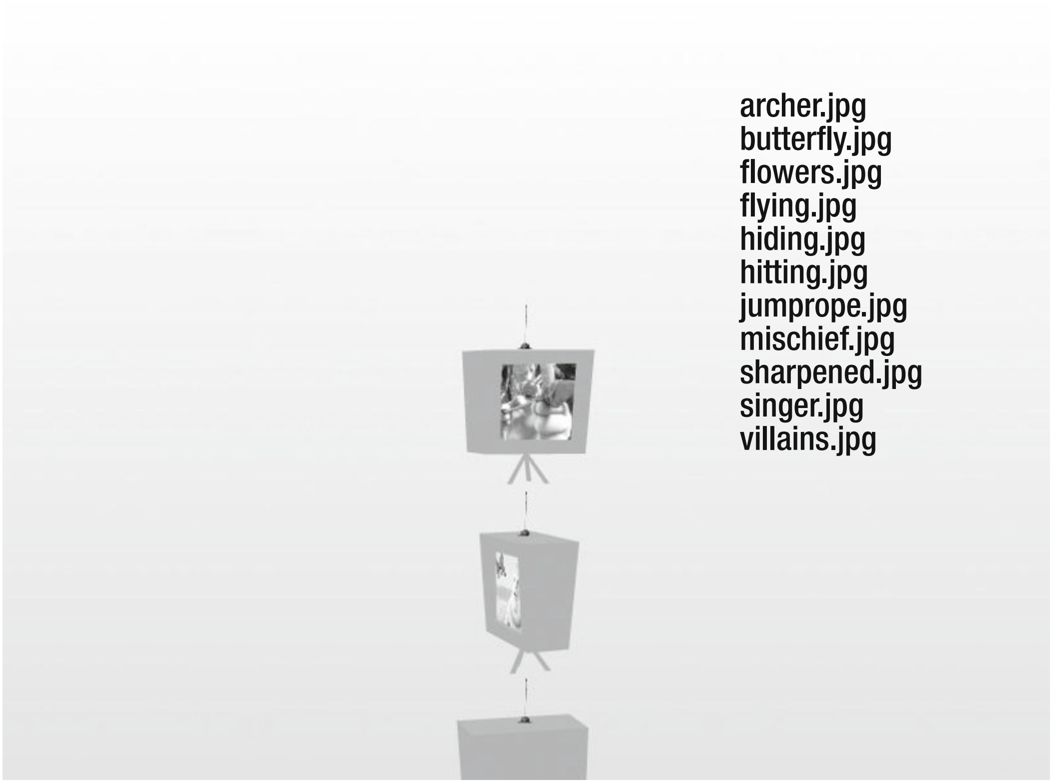


Figure 4-10. The image gallery before adding interactivity

Adding movement and interactivity

The movement we wish to add to the application will be triggered when the user clicks a menu item. We want to rotate the camera until it faces the TV screen corresponding to the item clicked and pan the `_pivot` container up or down so that the selected TV ends up at the center of the stage. As an extra subtlety, we will give the selected model a drop shadow filter to lift it off the page.

To react to a `CLICK` event from a menu item, we need to add some code to the `_onClickMenuItem()` method. Before we do that, let's define three new global class variables that will be used here and elsewhere in the `TwistedImageGallery` class.

```
private var _target_y : Number = 0;
private var _flying_to_tv : Boolean;
private var _last_active_tv : TVBox;
```

Insert this code at the start of the class definition, and then add the following to the `_onClickMenuItem()` method:

```
var tv : TVBox = (ev.currentTarget as ImageMenuItem).tv;
```

```
if (_last_active_tv) {
  _last_active_tv.ownCanvas = false;
  _last_active_tv.filters = [];
}

tv.ownCanvas = true;
tv.filters = [new DropShadowFilter(0, 0, 0, 1, 16, 16, 0.5, 2)];

_flying_to_tv = true;

_target_y = -tv.y;
_camera.panAngle = tv.rotationY - 180;

_last_active_tv = tv;
```

In the preceding code, we grab a reference to the selected TV instance from the `tv` property of the `ImageMenuItem` object returned in the event's `currentTarget` property. Next, we reset the `ownCanvas` and `filters` properties of any previously selected `TVBox` object. The newly selected `TVBox` object has its `ownCanvas` and `filters` properties modified to enable the drop shadow filter effect, and the motion mode `_flying_to_tv` is set to `true`, informing the application to take control of the camera and container movements and temporarily deactivate all other user interaction.

Two properties are required to define the movement required to move to the selected TV screen. The `_target_y` variable stores the y coordinate that aligns the `_pivot` container object so that the selected TV is positioned at the center of the stage. The camera `panAngle` properties stores the rotation value required by the camera to face the selected TV's screen image. Each variable is used as the end value in a tweening movement so that the change in position is performed smoothly. In the case of `panAngle`, we use the built-in tweening methods of the `HoverCamera3D` class.

As a final step in the `_onClickMenuItem()` method, the `_last_active_tv` variable is updated to the currently selected `TVBox` object, so that its `ownCanvas` and `filters` properties can be reset on the next menu selection.

Now that we have defined the motion mode and variables of a menu item selection, we need to modify the `_onEnterFrame()` method so that it will disable the mouse-controlled camera panning motion when the TV selection motion is taking place. We also need to use our `_target_y` variable to define the position tween of the `_pivot` container. Replace the `onEnterFrame()` method with the following code:

```
private function _onEnterFrame(ev : Event) : void
{
  if (!_flying_to_tv) {
    if (stage.mouseY < 120 || stage.mouseY > 480) {
      var max_y : Number = _pivot.maxY - _pivot.minY - 100;

      _target_y += (stage.mouseY - stage.stageHeight/2) / 30;
      _target_y = Math.max(0, Math.min(max_y, _target_y));
    }

    if (stage.mouseX < 160 || stage.mouseX > 640)
      _camera.panAngle += (stage.mouseX-stage.stageWidth/2) / 60;
  }
  else if (Math.abs(_pivot.y-_target_y) < 0.5) {
    _flying_to_tv = false;
  }

  _pivot.y += (_target_y - _pivot.y) / 4;

  // Wobble a bit up and down
  _pivot.y += 2 * Math.sin(getTimer() / 700);

  _camera.hover();
  _view.render();
}
```

The previously existing code in `_onEnterFrame()` for the mouse-controlled camera motion is now accompanied by a control for moving the `_pivot` container up or down depending on the y coordinate of the mouse, all wrapped within an `if` statement that checks if the camera is already animating on its way to a selected TV. If it is, the mouse-controlled motion is skipped, and instead, a check is made to see if the automatic camera control has completed its tween. Only when the y position of the `_pivot` container is within a tolerance of 0.5 units to the `_target_y` property will control be handed back to the mouse.

Notice that when the mouse-controlled motion is active, the total height of `_pivot` is calculated by subtracting its `minY` property from its `maxY` property. The same calculation can be made for width (`maxX-minX`) and depth (`maxZ-minZ`) of any 3D object in Away3D.

Following the `if` statement, the y position of the `_pivot` container is updated using the `_target_y` property, with a slight easing effect. It then has an offset applied using a sine wave output, to achieve a "hovering" motion effect as the TVs move. The calls to the hover camera update `hover()` and view rendering method `render()` at the end of the `_onEnterFrame()` method remain the same as before.

Recompiling the application, you should instantly see a difference in interaction as the TVs can now be moved up and down as well as rotated left and right with the mouse. Clicking a menu item will automatically animate the application to the correct viewing position for the selected TV. You have completed the creation of the twisted image gallery!

Summary

In this chapter, we have introduced and compared the most common types of 3D objects: primitives that are geometric shapes created internally and custom models created externally that can be either loaded at runtime or embedded within the application using a converted ActionScript model class. We have also created your first practical 3D application using Away3D.

This chapter included some new 3D terminology that may not be all that familiar. The important terms, concepts, and techniques are recapped in the following list that will hopefully assist the creative process when building your own 3D applications in Flash:

- Vertices, faces, and segments are the most basic visual elements in any 3D model but are rarely accessed directly.
- Primitives are basic 3D geometric shapes, such as spheres and planes, represented by classes located in the `away3d.primitives` package.
- Polygons in Away3D are a particular type of primitive, represented by the `RegularPolygon` and `WireRegularPolygon` classes.
- Segments are used to draw the wire primitive classes and can form abstract networks by using the `LineSegment` class.
- Custom models can be loaded from a variety of different file formats using `Loader3D` and the parsing classes available in the `away3d.loaders` package.
- Encoding 3D geometry as an ActionScript model using the `AS3Exporter` class in the `away3d.exporters` package has both size and speed benefits.
- 3D sprites can be used to simplify and speed up the rendering of nondescript symmetric objects such as smoke clouds. A variety of different types of 3D sprite exist in the `away3d.sprites` package.

Later chapters will cover using other types of content in Away3D, such as vector graphics and text, and procedural meshes using more complex generative tools. Before that, Chapter 5 looks in more detail at how to use lighting and materials to improve the visual impact of a 3D project.

Chapter 5

Materials, Lights, and Shading

As you saw in the previous chapter, all meshes (including internally created primitives and imported models) are built from geometric elements such as triangles and line segments that have their shapes defined by vertex points. For the renderer to be able to draw these elements to the view, an appearance definition similar to the line and fill style settings used in the native drawing API in Flash is required. In 3D, this definition is frequently referred to as a **material**, and in Away3D, it takes the form of a class instance that can be set in the `material` property of an individual element or global mesh object. Materials can be used to paint solid colors or bitmap images onto the surface of 3D objects, and special types can define how an object should react to light in a scene.

The process of simulating light in the field of 3D graphics is known as **shading**. It can involve a high degree of processing per-frame due to the methods used for shading calculations (in Flash, this is generally accomplished by building up layers of color that are resolved into a single blended layer at runtime). Creating convincing shading on a 3D object can be considered a bit of an art form, requiring expertise in the familiar area of design versus programming where the perfect balance is sought between performance and aesthetics.

Understanding Away3D materials

Despite Away3D offering many different types of material, every effort has been made to keep the programming interface consistent. This consistency should assist in your general understanding of material types and creating easily interchangeable materials, a great timesaver when searching for the one that looks and performs the best in your application. As a general rule in this chapter, **simple materials** are better from a performance point of view, while more-complex **shading materials** have the potential to start chewing heavily on the CPU and need to be used sparingly.

All materials are represented by classes in the `away3d.materials` package and are applied to a mesh object by setting its `material` property to an instance of the desired material type. This property has already been used in code examples seen earlier in the book, such as the following snippet from an early example in Chapter 3:

```
_cube1.material = new WireColorMaterial(0xFFFFFF);
```

Here, we set see the `material` property of the cube primitive `cube1` being set to a `WireColorMaterial` instance using a white color for its shape fill, and a (default) black color for its outline. Of course, different material classes have different properties, some of which require setting before the material can be rendered. The `BitmapMaterial` object requires a `BitmapData` object passed in its constructor, whereas the `ColorMaterial` object needs only a color value. But aside from these minor differences, Away3D materials are all created and applied in the same way. Let's dive into some code by setting up a base class for the examples in this chapter, starting with the following class shell:

```
package flash3dbook.ch05
{
  import away3d.cameras.*;
  import away3d.containers.*;
  import away3d.primitives.*;

  import flash3dbook.common.MonkeyMesh;

  import flash.display.*;
  import flash.events.*;

  [SWF(width="800", height="600")]
  public class Chapter05SampleBase extends Sprite
  {
    protected var _view : View3D;
    protected var _camera : HoverCamera3D;
    protected var _cube : Cube;
    protected var _sphere : Sphere;
    protected var _ape : MonkeyMesh;

    protected var _state : int = 0;

    public function Chapter05SampleBase()
    {
      _createView();
      _createScene();
      _createMaterials();
    }

    protected function _createView() : void
    {
    }

    protected function _createScene() : void
    {
    }
    protected function _createMaterials() : void
    {
    }

    protected function _toggle() : void
    {
    }

    protected function _onClick(ev : MouseEvent) : void
    {
      _toggle();
    }

    protected function _onEnterFrame(ev : Event) : void
    {
    }
  }
}
```

The preceding code defines global variables for the view, camera, and three mesh objects we will use throughout our examples as test objects for different materials, as well as stub methods that we will fill out before moving on to our first example. The global `_state` variable will be used when necessary to toggle between different material states in our examples (e.g., between different light types or different material types).

Let's start by creating and setting up the basic elements for an Away3D application. First, we create the camera by adding the following lines of code to the `_createView()` method:

```
_camera = new HoverCamera3D();
_camera.distance = 150;
_camera.tiltAngle = 10;
```

Here, we are using a hover camera, with a default distance 150 units away from the scene's origin and a tilt angle of 10 degrees to elevate our viewing position by a small amount.

Next, we create the view by adding the following lines of code to the end of the `_createView()` method:

```
_view = new View3D();
_view.x = 400;
_view.y = 300;
_view.camera = _camera;
addChild(_view);
```

Here, we align the position of the view with the center of our 800 × 600 stage, set the `camera` property to use our newly created hover camera, and add the view to the Flash display list so it can be seen.

Before moving on, we need to define two event listeners by adding the following lines of code to the end of the `_createView()` method:

```
stage.addEventListener(Event.ENTER_FRAME, _onEnterFrame);
stage.addEventListener(MouseEvent.CLICK, _onClick);
```

The ENTER_FRAME event handler _onEnterFrame() will render the view and handle some basic camera movement, while the CLICK event handler _onClick() is set up so that we can toggle between different materials and light settings when the mouse button is clicked anywhere inside the Flash movie.

Now, we add the code to create the 3D objects used as test cases in our examples for this chapter: these are a cube, a sphere, and an instance of an imported monkey model. Let's start by creating a cube primitive with the following lines of code added to the end of the _createScene() method:

```
_cube = new Cube();
_cube.width = 30;
_cube.height = 30;
_cube.depth = 30;
_cube.x = -70;
_view.scene.addChild(_cube);
```

This creates a cube primitive that is 30 units in all dimensions and positions it 70 units to the left of the scene's origin. Next, we add the following to the end of the _createScene() method to create a sphere primitive with a radius of 25 and a position 70 units to the right of the scene's origin.

```
_sphere = new Sphere();
_sphere.radius = 25;
_sphere.x = 70;
_view.scene.addChild(_sphere);
```

Finally, we create an instance of the imported class MonkeyMesh by adding the following to the end of the _createScene() method:

```
_ape = new MonkeyMesh();
_view.scene.addChild(_ape);
```

The MonkeyMesh class is a model that has been converted to ActionScript using the ActionScript exporter for the 3D modeling package Blender. The conversion process is covered in more detail in Chapter 4, but the resulting class can be instantiated and added to a scene in Away3D just like a regular primitive class.

Since we don't adjust the position of the MonkeyMesh instance, it will be centered on the scene's origin by default, sandwiched between our two other primitives. The ActionScript file for the MonkeyMesh class is assumed to exist at its correct location inside the flash3dbook.common package. If you do not have this file, it can be downloaded online inside the examples resource file for this chapter by going to the Downloads section of www.friendsofed.com.

Let's take a quick look at the remaining empty methods from our Chapter05SampleBase class definition. The _createMaterials() and _toggle() methods can be left blank, as they will both be overridden with custom functionality by subsequent example class definitions. _createMaterials() will instantiate and apply the different material classes to be investigated, and _toggle() will control our viewing mode for comparing different material settings, triggered by the event handler method _onClick() mentioned earlier.

The _onEnterFrame() method controls the camera position according to the position of the mouse cursor. To enable the camera movement, we add the following lines of code to this method:

```
_camera.panAngle -= (stage.mouseX - stage.stageWidth / 2) / 100;
_camera.hover();
```

The hover camera's `panAngle` property is incremented on every frame by an amount dependent on the x position of the mouse cursor, with half the stage width subtracted from the mouse position to give a coordinate relative to the center of the view. Incrementing the `panAngle` property causes the camera to rotate either left or right, with the speed of rotation dependent on the distance of the cursor from the center of the stage. The incrementing value is divided by 100 to keep the rotation speed within controllable limits.

All that is left to do is render the view, which is done by adding the usual `render` method call to the end of the `_onEnterFrame()` method.

```
_view.render();
```

Compiling the `Chapter05SampleBase` class at this point will display the output shown in Figure 5-1. Moving the mouse cursor left and right will rotate the camera left and right, allowing you to view the three mesh objects in the scene from any angle. Because we have yet to set the `material` property on any of our meshes, they are all displayed using the default material type—a `WireColorMaterial` object with a random color defined for the surface fill. We will investigate this material type in more detail later in this chapter, but for now, let's kick off our first example with a look at some basic material types available in Away3D.

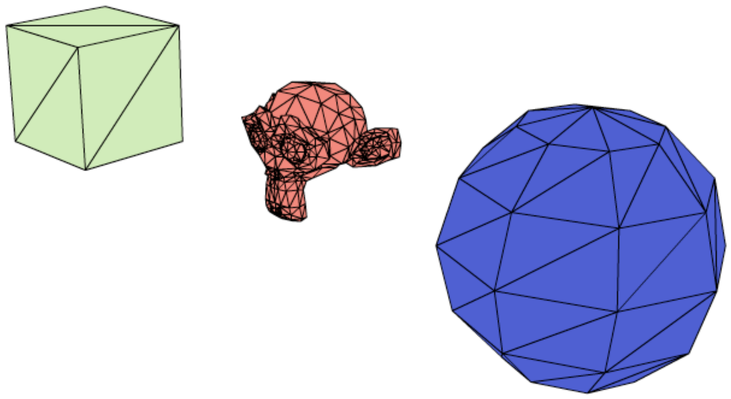


Figure 5-1. The result of compiling the chapter base class Chapter5SampleBase, before any materials are applied to the three mesh objects

Using color and bitmap materials

Color materials use simple color values to paint the visible elements of a 3D object. Various types of color material exist, differing in the way they apply color to an object and the way the object is rendered. By contrast, **bitmap materials** use an image to texture the surface of a 3D object as if it were covered in a piece of gift wrap. For bitmap materials to work, data points called **UV coordinates** must be available for each vertex of each face in the mesh. A UV data point is a 2D vector that represents the (x, y) position on the surface of the texture image to be mapped to the corresponding 3D (X, Y, Z) vertex position on the surface of a face. The material then uses **bilinear interpolation** to stretch the texture's pixel data between UV positions for each vertex in the face (which is typically a triangle or quadrilateral polygon). The entire process is known as **texture mapping**.

UV coordinates for imported models are usually created by the 3D artist in a modeling package prior to export, but we will take a closer look at how UV coordinates can be generated and modified from inside Away3D in Chapter 7 when we build our own custom 3D object from scratch.

To create our first materials example, let's start by extending the newly created `Chapter05SampleBase` class with the following document class, overriding the `_createMaterials()` and `_toggle()` methods with stubs ready for custom use:

```
package flash3dbook.ch05
{
  import away3d.core.utils.*;
  import away3d.materials.*;
  import flash.display.*;

  [SWF(width="800", height="600")]
  public class SimpleMaterials extends Chapter05SampleBase
  {
    [Embed(source="../../../assets/ch05/redapple.jpg")]
    private var AppleImage : Class;

    private var _bitmapMaterial : BitmapMaterial;
    private var _colorMaterial : ColorMaterial;

    public function SimpleMaterials()
    {
      super();

      _toggle();
    }

    protected override function _createMaterials() : void
    {
    }

    protected override function _toggle() : void
    {
    }
  }
}
```

The first defined global variable `AppleImage` is an embedded image asset, created from a file called `redapple.jpg`. Make sure the path reflects the actual location of the file on your hard drive. If you are working with the chapter download files from `www.friendsofed.com`, this path should be correct from the outset.

The [Embed] meta-tag used in the preceding example compiles images and other file assets into project SWFs created with Flex Builder, Flash Develop, FDT, or any other editor that uses the Adobe Flex SDK to compile. This method of asset inclusion mimics the organization and compilation of library assets contained in an FLA. Since the introduction of CS4, [Embed] tags will compile in Flash Professional as well, but if you're using CS3, you will need to modify the code and update the library assets of the container FLA in the following way to compile: Import the image file into the library, open its properties panel, and check the Export for Actionscript option. Underneath, enter the name of the variable you find directly under the [Embed] tag line (in this case AppleImage) as the class name. Also, make sure you remove the [Embed] line and the one that directly follows it from the source code. The modification will allow this and any subsequent examples using the [Embed] meta-tag to work in CS3.

The remaining two private variables called `_bitmapMaterial` and `_colorMaterial` are global placeholders for the two material classes we will be testing: the `BitmapMaterial` and `ColorMaterial` classes.

To complete our example, we need to fill out the code required in our `_createMaterials()` and `_toggle()` method stubs. First, we need to create instances of our material classes by adding the following code to the `_createMaterials()` method:

```
_bitmapMaterial = new BitmapMaterial(Cast.bitmap(AppleImage));
_colorMaterial = new ColorMaterial(0xFFAA00);
```

For the `BitmapMaterial` instance, we see the first use of the `Cast` class. This handy Away3D utility resides in the `away3d.core.utils` package and contains static methods for converting one type of ActionScript object to another. In this case, the `BitmapMaterial` class requires a `BitmapData` object representing the texture to be passed in its constructor argument. However, an object created by an `[Embed]` meta-tag class will be of type `Class`. In previous chapters, we created an instance of the `AppleImage` class variable and directly cast it into a regular ActionScript `Bitmap` object, then extracted the bitmap data from its `bitmapData` property. However, this approach requires typing a couple of extra lines of code every time you needed a new bitmap material. With `Cast`, the work is done for you in a quick, neat fashion.

The `ColorMaterial` instance is passed a constructor argument that sets the color value of the material on instantiation. Color values are unsigned integers by type, and in this example, we define the color in hexadecimal as `0xffaa00`, a bright orange. Hexadecimal is a common notation for color values as it is generally easier to read than the equivalent decimal value. `ColorMaterial` requires no constructor arguments by default but will end up using a random color if none is given. The color could also be set after instantiation by using the `color` property of the `ColorMaterial` object.

Next, we add the code required for toggling between our materials at runtime. The idea here is to be able to hot swap materials on all our objects, so that direct comparisons can be made. Remember that the base class sets up the `_toggle()` method to be invoked when we click with the mouse anywhere on the stage. By updating the `_state` class variable when this occurs, we will know what to do the next time the `_toggle()` method is called. For our `SimpleMaterials` example, we have two possible states to toggle between. Using the `_state` integer variable, we identify these as `0` and `1` in a simple `switch` statement by adding the following to the `_toggle()` method:

```
switch (_state) {
  case 0:
    _cube.material = _colorMaterial;
    _sphere.material = _colorMaterial;
    _ape.material = _colorMaterial;
    _state = 1;
    break;
  case 1:
    _cube.material = _bitmapMaterial;
    _sphere.material = _bitmapMaterial;
    _ape.material = _bitmapMaterial;
    _state = 0;
    break;
}
```

In the preceding code, we see that if the `_state` variable returns `0`, the color material is applied to all mesh objects by resetting their `material` properties, and `_state` is reset to `1`, ready for the next click. Likewise, if the `_state` variable returns `1`, the bitmap material is applied to all mesh objects, and `_state` is reset to `0`.

Compiling the example, our three mesh objects are rendered with our `AppleImage` texture applied using the bitmap material option. Moving the mouse left and right rotates the camera left and right around the origin of the scene. The bitmap material option is used initially because the `_toggle)` method is called once in the constructor of the `SimpleMaterials` class, setting up a default bitmap material state. Clicking once anywhere in the Flash movie swaps the bitmap material for the orange-colored material on all objects. This looks very flat by comparison—so much so that if the camera remains still by keeping the mouse in the center of the movie, you'd be hard pressed to tell you were looking at a 3D scene and not just three irregular orange shapes. Jumping back to the bitmap material option with a further click, we can really start to see the extra detail texture mapping adds when applied to a mesh object. A comparison of the two results is displayed in Figure 5-2.

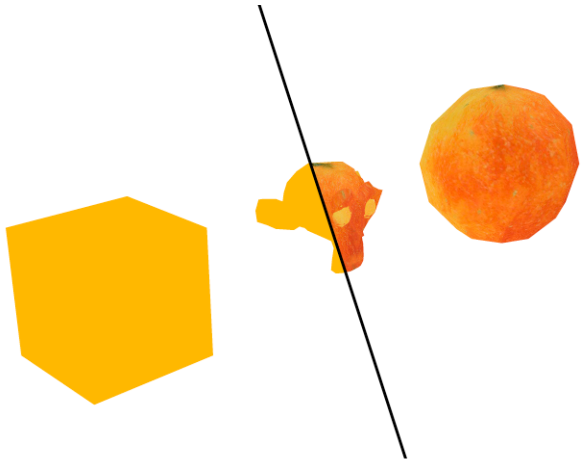


Figure 5-2. The two states of the SimpleMaterials example side by side: ColorMaterial with orange faces on the left and BitmapMaterial with textured faces on the right

The advantage of using bitmap materials over color materials doesn't stop with detail enhancement. Textures can be created to simulate objects lit by a light source, providing a greater level of realism in a scene. However, this method of static shading (or **texture baking** as its more commonly known) can only go so far; for maximum realism Away3D can apply shading in real time, as you will see in the shading materials section later in this chapter.

Working with wire materials

Wire materials are so called because they provide a wireframe look to a 3D mesh object. In Away3D, all materials inherit from a basic wire material represented by the `WireframeMaterial` class found in the `away3d.materials` package including the default `WireColorMaterial` class, which is set to use a black color for its wire component. This material doesn't win any prizes for aesthetics but is useful for debugging purposes thanks to its ability to draw all the lines that connect the vertices in a face, as well as filling all faces with a solid color.

As an alternative look, the `WireframeMaterial` class can be used when we want to only draw the outlines of faces, without a solid fill. Let's extend our chapter base class to see how these two materials compare:

```
package flash3dbook.ch05
{
  import away3d.materials.*;

  [SWF(width="800", height="600")]
  public class WireMaterials extends Chapter05SampleBase
  {
    private var _wfMaterial : WireframeMaterial;
    private var _wcMaterial :  WireColorMaterial;

    public function WireMaterials() {
            super();
            _toggle();
    }

    protected override function _createMaterials() : void
    {
    }

    protected override function _toggle() : void
    {
    }
  }
}
```

In the preceding code, we define two private variable placeholders for an instance of `WireframeMaterial` and `WireColorMaterial`. The constructor calls the `_toggle()` method once to execute the code we eventually create for initializing our materials. We also override the `_createMaterials()` and `_toggle()` methods, ready for our custom functionality. Now, let's create our material instances by adding the following code to the `_createMaterials()` method:

```
_wfMaterial = new WireframeMaterial();
_wfMaterial.wireColor = 0x000000;

_wcMaterial = new WireColorMaterial();
_wcMaterial.color = 0XCCCCCC;
_wcMaterial.wireColor = 0x666666;
```

The first two lines of code create a `WireframeMaterial` object and set its `wireColor` property to black. This will produce a black line drawn around all triangles. The remaining code creates a `WireColorMaterial` specified by two properties, `color` for the face color and `wireColor` for the line color. Here we are using property setters for updating the color values of our materials, but in the same way you saw in the previous example for the `ColorMaterial` class, we can also set the color for each material in the class constructor, corresponding to `wireColor` for the `WireframeMaterial` class and color for the `WireColorMaterial` class.

Next, we fill out our script for switching between materials by adding the following lines of code to the `_toggle` method:

```
switch (_state) {
    case 0:
          _cube.material = _wfMaterial;
          _sphere.material = _wfMaterial;
          _ape.material = _wfMaterial;

          _cube.bothsides = true;
          _sphere.bothsides = true;
          _ape.bothsides = true;
          _state = 1;
          break;
    case 1:
          _cube.material = _wcMaterial;
          _sphere.material = _wcMaterial;
          _ape.material = _wcMaterial;

          _cube.bothsides = false;
          _sphere.bothsides = false;
          _ape.bothsides = false;
          _state = 0;
          break;
}
```

Once again, we start by checking the value of the `_state` variable and react accordingly by setting the `material` property of all mesh objects to the `WireframeMaterial` instance if `_state` returns `0`, and the `WireColorMaterial` instance if `_state` returns `1`. We also set reset the bothsides property of the mesh objects depending on the type of material being used - when wire materials are used we need to see both sides of the geometry and the bothsides property is reset to true, but when color materials are used we only see the front faces of the geometry and the bothsides property can be reset to false. In each case, _state is reset to the alternative state value, ready for the next mouse click.

Compile the code, and click the stage anywhere to toggle between the two materials. Figure 5-3 portrays the appearance of both states side by side. Each material is useful for debugging a mesh when you need to see the exact orientation of the faces, and the application of the `WireframeMaterial` instance has the added bonus of only drawing the triangle outlines, allowing both sides of the object to be displayed at once as long as the `bothsides` property of the mesh object is set to `true`. The stylized effect of a wire material can also be useful in certain design situations.

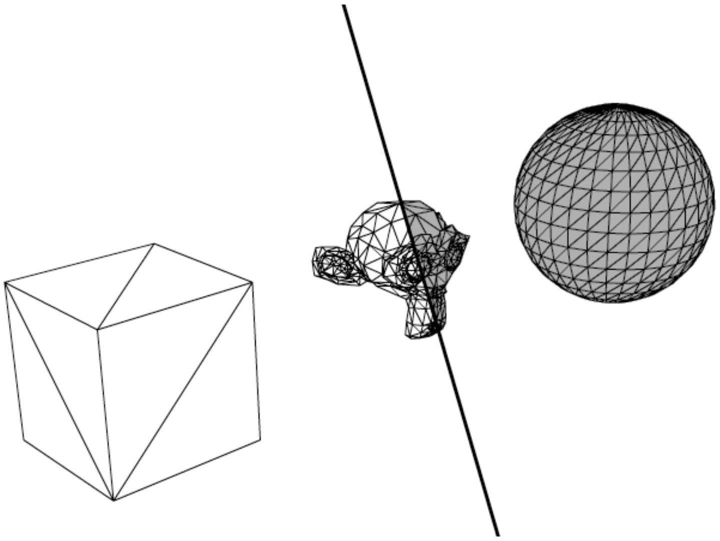


Figure 5-3. The two states of the WireMaterials example side by side: on the left, WireframeMaterial is used as the mesh material, and on the right, WireColorMaterial is used.

Using lights and shading materials

In the real world, light reflected from the surfaces of the objects around us enters our eyes and produces a perceived image of the scene. Without a light source, everything would appear black. Simulating this process in a computer is known as **shading** and typically requires a large amount of processing.

One method of shading calculates virtual light rays for every pixel in the view, in a technique known as **ray tracing.** This method is used by the majority of professional 3D modeling programs to render high-quality 3D images. However, real-time 3D engines require a fast render speed and consequently aim to approximate a lot of real-world processes rather than carrying out faithful simulations. Instead of measuring light intensities for every pixel in an image, values are estimated using a variety of techniques ranging from precalculating intensities in a texture (the previously mentioned texture baking technique) to texel-based normal mapping that uses an extra texture image to calculate light intensities across the surface of an entire object's texture in a single step. We will look at the latter technique in more detail later in this chapter.

In the majority of preceding chapter examples, we have gotten away with not using any shading techniques on our materials. This approach is perfectly acceptable when speed is a priority over image quality. But if we want to produce 3D scenes with a higher degree of realism, shading materials are one option we can use. When a shading material is applied to a face, the resulting rendered surface can be brighter or darker than the actual color, depending on the shading calculations. In simple terms, a surface that directly faces toward a light source will appear brighter, while one that directly faces away will appear darker. A shading material produces an overall intensity map of light to be applied to the underlying texture or color of the object, commonly referred to as the light map of the object.

In Away3D, several different types of shading material exist, offering varying degrees of detail and requiring varying amounts of processing. As is the case with much real-time 3D content, finding a happy balance between these two quantities is the main basis on which choices are made.

Lighting in Away3D

For a shading material to work, we need to define at least one **light source**. In Away3D, lighting is achieved by adding light source objects to a scene. Three types of light source objects can be used: point, directional, and ambient. These are represented by the classes `PointLight3D`, `DirectionalLight3D`, and `AmbientLight3D` found in the `away3d.lights` package, and each type of light source produces a different shading result when applied to a shading material.

Omnidirectional lighting with point lights

Point light sources work in a way similar to a single light bulb and are represented in Away3D by the `PointLight3D` object. They emit light in all directions from a definable position in the scene. If a point light is positioned between two objects, the objects will appear lit from opposite directions because the relative position of each object with regard to the light source is reversed. An illustration of this shown in Figure 5-4.

A defining characteristic of the point light source is the falloff over distance exhibited by the intensity. As in the real world, the overall intensity of reflected light from an object's surface is proportional to the inverse square of the distance from the surface to the light source. This means that the brightness of a light beam decays with the distance (d) from the light by an amount $1/d^2$. The overall brightness of a light source can be adjusted using the `brightness` property of the `PointLight3D` object, but the decay of intensity over distance will always follow the same curve.

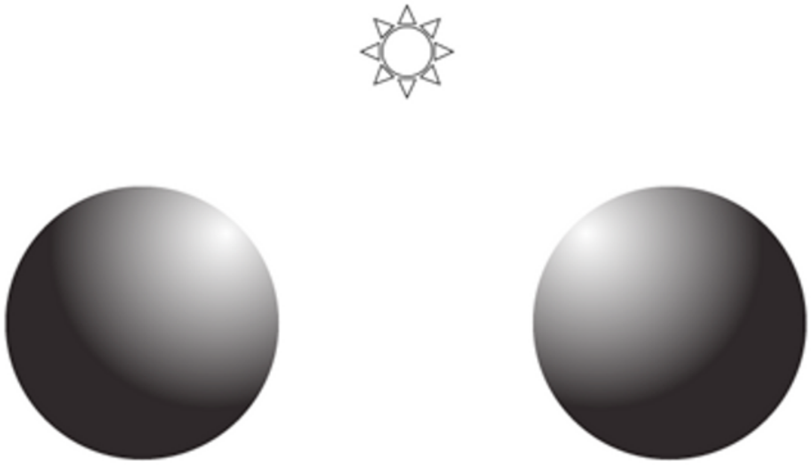


Figure 5-4. An example of two objects lit from a single point light source emitting in all directions

Parallel beam lighting with directional lights

Directional light sources are represented in Away3D by the `DirectionalLight3D` class and can be imagined as light-emitting, infinitely large planes with all emitting rays pointing in the same direction. This is in contrast to point lights whose rays emit in all directions from a center point.

A directional light source is a simplification of the kind of light field generated by a point light source with near-infinite brightness positioned at a near-infinite distance. In this scenario, the natural falloff of the point light would be barely noticeable over the differing distances in our scene, and to use a point light object for such a setup would be impractical. All that is required to set up a directional light source is a 3D direction vector that defines the direction of the light rays. This is set on a `DirectionalLight3D` object using the `direction` property. For a shading material, the most important piece of information when dealing with a directional light is the angle between the light source direction and the object's surface. It is assumed that the intensity of the light is constant at all distances.

Figure 5-5 illustrates an example scenario, with a directional light source (drawn as a wall of parallel beams) emanating from the right-hand side of the screen. In Away3D, there is no such thing as a position for a directional light source; here, we are imagining one for illustrative purposes. Both the right and left spheres are lit identically, despite the left sphere being further away from the light source than the right sphere. This characteristic of directional lights makes them ideal for simulating sunlight in an outdoor scene, as well as being generally more efficient than their closely related point light counterparts, requiring fewer calculations to produce a similar shading effect.

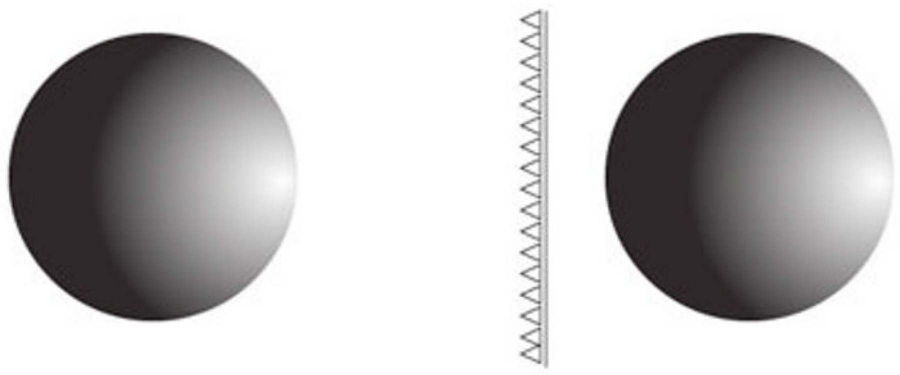


Figure 5-5. An example of two sphere primitives lit from a single directional light source. The parallel nature of the source can be considered a close approximation of the light field generated by a point light source positioned at a near-infinite distance with a near-infinite brightness.

Background lighting with AmbientLight3D

Ambient light sources are represented in Away3D by the `AmbientLight3D` class and can be considered one of the simplest types of light source. Adding an ambient light source to a scene will light all faces of all meshes in the scene equally, regardless of angle or position.

In Away3D, ambient lights have an effect on only a few shading materials. They are commonly used together with the other light source types to boost the overall brightness in a scene. For those shading materials that do not react to ambient lights, directional lights and point lights have their own internal ambient coefficients to provide an ambient component to their lighting effect.

Creating and configuring light sources

Creating a light source and preparing it for use with a shading material is as easy as instantiating the respective class, setting its `color` property, and adding it to the scene using the custom `addLight` method. This process is identical for all light types. For example, a red point light source would be created in the following manner:

```
var light : PointLight3D = new PointLight3D();
light.color = 0xFF0000;
myView.scene.addLight(light);
```

Each type of light brings its own configuration properties, the one common property being `color`, which sets the emitting color value for the light object. Ambient lights don't require any further configuration due to their simplicity, so we will take a closer look at the remaining two types of light source on offer.

The `brightness` property on the `DirectionalLight3D` and `PointLight3D` classes controls the overall intensity of the light that is emitted from the light source. For a directional light source, this means the overall intensity at any position in the scene. The effects of a light source on the surface of a shading material are cumulative, so an Away3D scene with more than one light object will have the resulting intensities added together (through color components) to calculate the total light intensity to be applied. A point light source's intensity is attenuated over distance, so in this case the `brightness` property represents the intensity of the light 500 units away from the source. The default `brightness` value is 1.0 for both `PointLight3D` and `DirectionalLight3D`.

Controlling the intensity of a light source

The basic principal of the rendering process for any shaded material is to calculate the intensity of the light entering the camera from any point on the surface of a face. If you investigate how different materials appear around you in the real world, you will quickly come to realize that they each react to light in subtly different ways. Some are reflective and glossy, while others are diffuse and dull. Real real-world materials have physical properties on a microscopic level that influence these different characteristics, but in a real-time 3D engine, replicating these surface perturbations in the same terms would be too complex. For most engines, the reflected light from the surface of a material is represented by three distinct components: ambient, diffuse, and specular reflections. These add up to a cumulative shading intensity called a **light map**, which is applied to the underlying shading material color to produce the final appearance of the material.

The **ambient component** of a light source is applied in a similar manner to the output of an `AmbientLight3D` object. It represents the fraction of light incident on the surface of a shading material that originates as ambient light from the light source. For example, a point light source inside a room has its ambient component created from diffuse reflection of the interior walls bouncing light back into the room. The resulting component is simulated by applying a uniform intensity to all shading materials encountered in a scene.

The **diffuse component** of a light source represents the fraction of incident light that originates directly from the source and is then scattered in all directions from the surface of a shading material. Its purpose is to simulate the light scattering performed by a soft material such as a ball of putty, that at a microscopic level is pitted and irregular, diffusing light uniformly regardless of incident angle. Because of its scattered nature, the diffuse component of a light source is ambivalent to viewing angle and is only concerned with the concentration of the beam incident on the surface of a shading material to calculate its reflected intensity. This is directly affected by the **incident angle**, which can be considered as the angle at which light from the light source arrives at the material surface.

The **specular component** of a light source represents the fraction of incident light direct from the source that is then reflected in a mirror-like fashion from the surface of a shading material. For this to occur, the incident angle of the light must be near equal the viewing angle of the camera, as is the case for a real mirror. For example, the reflected light seen in a mirror from someone holding a torch would be considered the specular component of that light source. As a comparison, if the same torch were shone on a stone wall, the reflected light seen would be the diffuse component of the light source.

Figure 5-6 illustrates how the resulting ambient, diffuse, and specular components of a light source are blended together to produce the light map for a sphere object with a shading material applied. The properties on point and directional lights that control the individual intensities of ambient, diffuse, and specular shading are unsurprisingly called `ambient`, `diffuse`, and `specular`. They represent the fraction of total light (taken from the brightness) to be used for that shading component, as a decimal number between 0 and 1. For example, the intensities seen in Figure 5-6 would be set using the following code snippet:

```
myPointLight.ambient = 0.3;
myPointLight.diffuse = 0.7;
myPointLight.specular = 1;
```

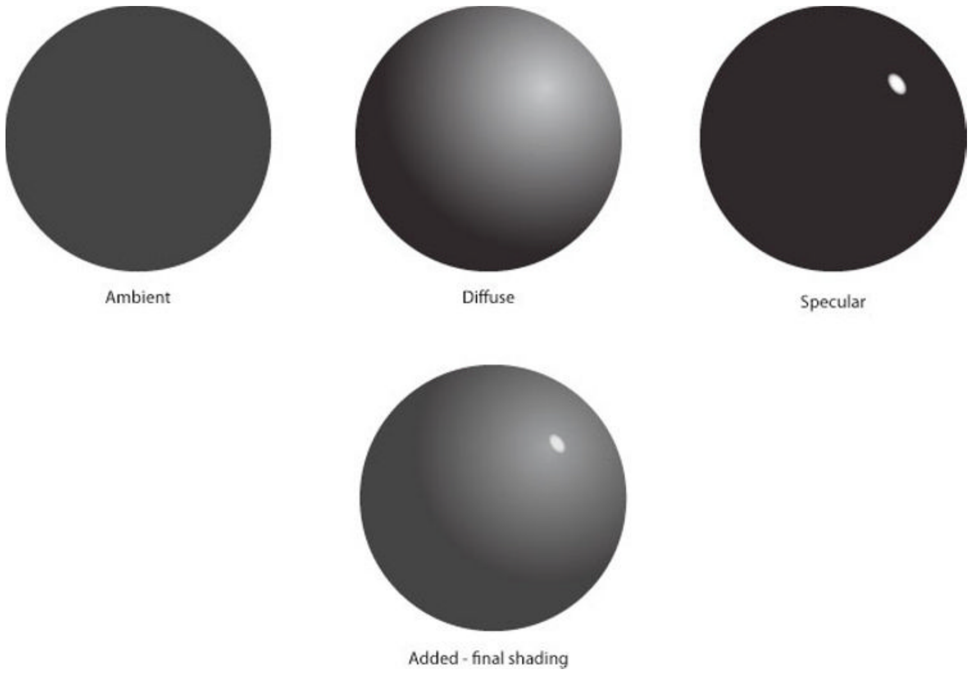


Figure 5-6. Ambient, diffuse, and specular light intensities are calculated separately and added together to create the final light map for an object with a shading material.

Shading materials in Away3D

As previously mentioned, shading materials are specific material types in Away3D that react to light. In this section, we will take a tour through the most frequently used types, highlighting the strengths and weaknesses of each. It is worth reiterating that almost all shading materials are very processor intensive and need to be used sparingly in an Away3D project. Recent advances in the Flash 10 Player have allowed some optimizations to be performed on certain shading material classes, but if the frame rate of an application is to remain smooth, the amount of shading materials use will always need to be kept within limits set by the processing overheads.

Flat shading materials

One of the simplest and most efficient methods of shading a 3D mesh object is known as **flat shading**. This calculates a single reflected intensity for each face based on the angle of the face to the light source. The result is applied to the base color of the material, producing the final output. There are two classes that perform this type of shading: `ShadingColorMaterial` and `WhiteShadingBitmapMaterial`. Both of these material classes work similarly to their respective nonshading counterparts, without the use of any processor-intensive layering, and are therefore useful as fast shading options.

For simplicity, the `WhiteShadingBitmapMaterial` class assumes the light color to be white for all light sources. The `ShadingColorMaterial` class is more versatile in its coloring, reacting to different colored lights and allowing you to configure different material colors for the three components of the light source. For example, a `ShadingColorMaterial` can use a red hue for its ambient color, a blue hue for its diffuse color, and a green hue for its specular color. The disadvantage of using `ShadingColorMaterial` is its lack of texture mapping, meaning that any color settings are applied across the entire material surface.

Let's create a sample class where the two types of flat shading material can be compared, starting with the following document class definition extending the `Chapter05SampleBase` class:

```
package flash3dbook.ch05
{
  import away3d.lights.*;
  import away3d.core.math.*
  import away3d.core.utils.*
  import away3d.materials.*;

  import flash.text.TextField;

  public class FlatShadingMaterials extends Chapter05SampleBase
  {
    private var _tf : TextField;

    private var _pointLight : PointLight3D;
    private var _dirLight : DirectionalLight3D;

    [Embed(source="../../../assets/ch05/redapple.jpg")]
    private var AppleImage : Class;

    private var _bMaterial : WhiteShadingBitmapMaterial;
    private var _cMaterial : ShadingColorMaterial;
    public function FlatShadingMaterials()
    {
      super();

      _createLights();

      addChild(_tf = new TextField());

      _toggle();
    }

    protected function _createLights() : void
    {
    }

    protected override function _createMaterials() : void
    {
    }

    protected override function _toggle() : void
    {
    }
  }
}
```

In the preceding code, we start by declaring a text field variable that will give us an indication of state for the toggle functionality. We then define two private variables called `_pointLight` and `_dirLight` to be used as our comparative light sources. Next, we embed the same `redapple.jpg` image file used earlier in this chapter, and as usual, the file path should work fine if you are working with the downloaded sample files. Similarly, the remaining two private variables called `_bMaterial` and `_cMaterial` are global placeholders for the two materials we will be testing in this example.

Aside from the usual initializing call to the `_toggle()` method in the constructor, we create a new `TextField` object for our `_tf` variable and add it to the display list. We then call a new method specific to this example called `_createLights()`, which for now is defined further down as an empty stub. We finish with the usual `_createMaterials()` and `_toggle` method overrides, ready for some custom functionality.

To start with we will create two lights objects using an instance of the `PointLight3D` class and `DirectionalLight3D` class, in order to compare the relative effects of each. We begin configuring our point light source by adding the following lines of code to the `_createLights()` method:

```
_pointLight = new PointLight3D();
_pointLight.position = new Number3D(70, 200, -200);
_pointLight.ambient = 0.1;
_pointLight.diffuse = 0.5;
_pointLight.specular = 1;
_pointLight.brightness = 1;
```

Here, the position of the point light is set to be 200 units above the scene's origin and 70 units to the right, locating it over the top of the sphere primitive in our scene. The `ambient`, `diffuse`, and `specular` properties of the light source are then set to provide a decent lighting range across the shading calculations. These values often require a little tweaking to achieve the desired effect, but the numbers used here are usually a good starting point. The last property configured is `brightness`, set to ensure the overall light levels from the point light source are appropriate, given the falloff over distance exhibited with this type of source.

Next, we configure our directional light with the following code added to the `_createLights()` method:

```
_dirLight = new DirectionalLight3D();
_dirLight.direction = new Number3D(70, 200, -200);
_dirLight.ambient = 0.1;
_dirLight.diffuse = 0.5;
_dirLight.specular = 1;
_dirLight.brightness = 3;
```

Here, we set the `direction` vector of our `DirectionalLight3D` object to match the same vector created for the position of the point light relative to the center of the scene. All component properties of the light source are set to match the point light values as closely as possible, the only major difference being the `brightness` value, which takes into account the uniform nature of the directional light intensity. Brightness can be set to any number, but generally, the aim is to provide a sufficient intensity that doesn't wash out the resulting shading with nothing but white light.

Now, we set up our instances of `ShadingColorMaterial` and `WhiteShadingBitmapMaterial` on the `material` properties of the mesh objects by adding the following to the `_createMaterials()` method:

```
_bMaterial = new WhiteShadingBitmapMaterial(Cast.bitmap(AppleImage));

_cMaterial = new ShadingColorMaterial();
_cMaterial.ambient = 0xff0000;
_cMaterial.diffuse = 0x008800;
_cMaterial.specular = 0x0000ff;

_cube.material = _cMaterial;
_sphere.material = _cMaterial;
_ape.material = _bMaterial;
```

Looking at the first line in the preceding code, you can see that a `WhiteShadingBitmapMaterial` object is instantiated in a similar way a nonshading `BitmapMaterial` object, requiring a `BitmapData` object passed in its constructor argument that contains the texture to be used for the material. As shown earlier in the `SimpleMaterials` example, using the `Cast` class on our embedded `AppleImage` asset is a simple way to extract this data. Next, we instantiate a `SimpleMaterials` object and use the overriding component colors rather than the base `color` property to define its shading by setting the `ambient`, `diffuse`, and `specular` properties to a selection of color values. Finally, we apply our shading materials to the desired mesh objects, with the `WhiteShadingBitmapMaterial` instance applied to the `MonkeyMesh` model and the `ShadingColorMaterial` instance applied to the `Cube` and `Sphere` primitive objects.

To complete the `FlatShadingMaterials` example, we want to enable light source swapping at runtime using the same toggle system as before. We do this by adding the following code to the `_toggle()` method:

```
switch (_state) {
  case 0:
    _view.scene.addLight(_pointLight);
    _view.scene.removeLight(_dirLight);
    _tf.text = 'POINT';
    _state = 1;
    break;
  case 1:
    _view.scene.addLight(_dirLight);
    _view.scene.removeLight(_pointLight);
    _tf.text = 'DIR';
    _state = 2;
    break;
  case 2:
    _view.scene.removeLight(_dirLight);
    _view.scene.removeLight(_pointLight);
    _tf.text = 'NONE';
    _state = 0;
    break;
}
```

This adds and removes the relevant light source from the scene and updates our descriptive text of what is currently being used as a light source. Starting with the default value of the `_state` variable `0`, `PointLight3D` is activated and our descriptive text is set to `POINT`. A single mouse click will advance the state to `1`, activating our `DirectionalLight3D` object and setting the descriptive text to `DIR`. A second click will use no lights at all and set the descriptive text to `NONE`. A further mouse click will start the cycle again, returning us to the `0` state with `PointLight3D` active.

Compiling the example reveals the three mesh objects displayed with shading materials. The monkey model has the shading equivalent of a bitmap material applied, while the cube and sphere primitives have the shading equivalent of a color material applied, with the results looking similar to Figure 5-7. Because we have defined separate material colors for use by the ambient, diffuse, and specular components of the light source on the `ShadingColorMaterial` object, the shading color applied to the face elements of the cube and sphere primitives should clearly show what intensity from the source is attributed to each component. Faces pointing away from the light source will be colored red (the ambient component), whereas those directly reflecting the light source will be colored blue (the specular component). All other faces will appear green (the diffuse component) or some mixed shade of green, red, and blue where the component intensities combine. Of course, this explanation does not apply to the monkey model, because `WhiteShadingBitmapMaterial` can only shade in varying degrees of white. However, the monkey model does retain a texture to its surface, which is one advantage of using this type of shading material.

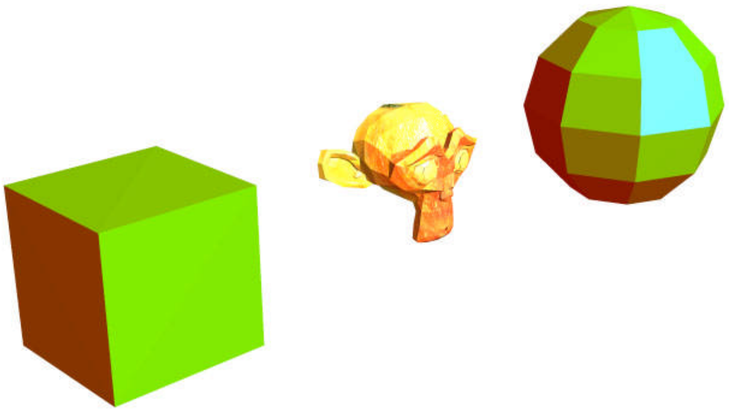


Figure 5-7. The POINT state of the FlatShadingMaterials example, rendered using simple shading materials. The cube and sphere primitives both use the ShadingColorMaterial object, while the monkey model has the WhiteShadingBitmapMaterial object applied.

Clicking once anywhere on the stage will swap the default `PointLight3D` object for the `DirectionaLight3D` object. With this light source active, you can see that intensities for reflected components of the light vary uniformly across all objects, demonstrating the uniform nature of the directional light source. Also, the angles of incidence for the incoming light on each object will match, creating matching areas of shadow (areas lit by the ambient component only) for each object. A final click will remove lights from the scene altogether, causing all mesh objects to be rendered black as you would expect in a real-world scenario.

Because we are generally restricted to relatively small numbers of faces in an Away3D scene, flat shading can be somewhat limiting. You only have to look at the sphere from our `FlatShadingMaterials` example to realize that the big square shading areas are somewhat revealing of our low polygon count. To achieve some smoother shading effects, we need to start looking at materials that generate light maps with a level of detail independent of polygon count.

Using normal map shading

Normal map shading (or **DOT3 shading**) is an approach to shading materials that can be used to work around the low polygon problems associated with real-time 3D. It is a little more expensive in processing terms than the simple shading techniques described in the previous section, but in certain scenarios, the visual benefits will vastly outweigh the increase in processing time.

The idea behind normal map shading is to process incident light intensities at a texel level rather than a face level. A **texel** is the representation of a single pixel of a bitmap texture as seen on the surface of an object with a texture mapping material applied. With separate light calculations made for each texel in a material, the detail of a mesh object's light map can be far greater than that achieved by single face calculations.

In normal map shading, we want to process the perceived light intensity of each texel of a material's texture. With flat shading, light intensity calculations are made by using a vector called the **face normal** that represents the 3D vector perpendicular to the surface of the face and determines the direction in which the face is pointing. In Figure 5-8, a simple cube is depicted showing two of its face normal vectors, on the top and right side. For normal map shading to work, we need the same normal vector information for each texel of the material texture, stored in an easily accessible data format. This takes the form of a second bitmap image created alongside the texture, known as a **normal map**.

A normal map is created programmatically in a process called **normal mapping**, which usually occurs at the model creation stage. We want the normal values in a normal map to vary in a much smoother way to the normals produced by face calculations, and there are two ways this can be done. The first is to start with a more detailed model that contains many more triangles and iterate through each texel of the material, generating a normal value based on a smoothed face normal value. The second is to use a height map (or **bump map**) texture to perturb the surface normals of the texels, again smoothing out the variation between face normal values but, in this case, producing a surface that has added relief detail. Typically, normal maps store normal information in color channels, with red, green and blue (RGB) values corresponding to the x, y, and z values of the normal vector. Once a normal map has been created, the amount of detail in an object's light map is no longer dependent on the polygon count of the model, and the geometry can be simplified to whatever is acceptable for the 3D engine to render. However, each normal map is tied to the object it was created for; consequently, each normal mapping material can only be applied to the object used in its creation.

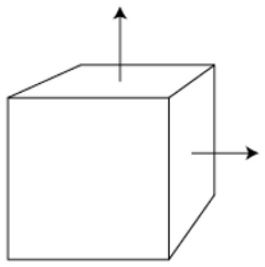


Figure 5-8. Face normal vectors for the top and right-hand sides of a cube primitive

Because a normal map stores values relating to the surface normal of a texel, each pixel in the normal map image must refer to a unique texel on the model surface. Unfortunately, many models are built with areas of a texture map that are reused for more than one area of a model's surface. For example, when texturing a car, the same piece of a texture can potentially be used for the left and right sides of the body. This results in a problem known as **overlapping UVs**, where a reversed-engineered map of a mesh's face's UV coordinates printed onto the texture creates overlapping polygons. A model that has no overlapping UVs is required for effective normal mapping.

Figure 5-9 depicts the normal map of a sphere (one that we will be using in our next code example). Normal maps produce quite colorful images because of the channel mapping that goes into their creation. To see the map in full color, look for the `sphere_normals.jpg` image inside the sample files for this chapter.

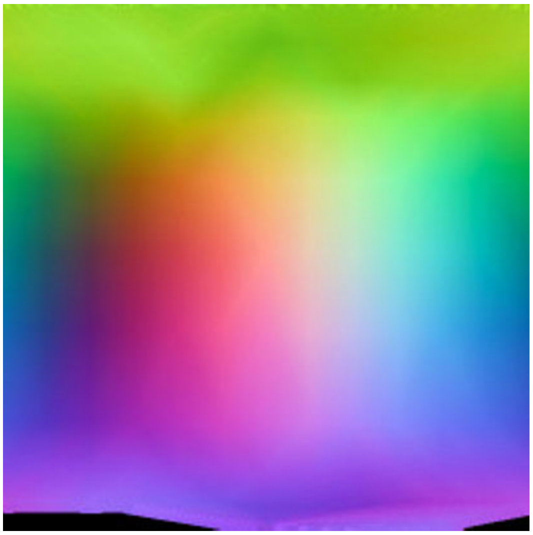


Figure 5-9. Generated normal map for a sphere

With a bit of effort, you can deduce some information about the geometry of the related object just from looking at a normal map. Remember that the X, Y, Z vector of the texel's normal is represented by the RGB value of the corresponding pixel. Along the top edge of Figure 5-9, all pixels are some shade of green, signifying that the texel normals are pointing up along the Y axis. If we consider that the texture of this material maps onto a sphere similarly to the way a map of the world maps onto a globe, the top edge of the texture will converge around the north pole of the sphere creating upwardly-pointing normals and confirming our green values. Using the same logic, we can deduce that the color shift from blue to magenta to red to light blue and back again across the central line of pixels in the normal map represents a horizontal texel normal vector rotating in a circle around the equator of the sphere.

Generating a normal map

The task of generating a normal map generally falls to the 3D designer or texture artist and for fairly good reasons; the best tools available to create normal maps are modeling and 3D rendering packages such as Blender, 3ds Max, Maya, and Cinema4D. For those without access to such software, there also exists a free Adobe AIR application called Prefab3D, downloadable from the Adobe AIR Marketplace at `www.adobe.com`. Developed by core Away3D developer Fabrice Closier to produce normal map images and more, this application works particularly well alongside Away3D in the preparation of 3D assets.

As you can imagine, the exact steps to render a normal map will differ from application to application, but here are a few pointers to help generate maps for use in Away3D:

- The UV coordinates on a high-polygon mesh used to render a normal map needs to match that of the low-polygon mesh used in the engine.
- Overlapping UVs will cause problems when rendering the resulting light maps for the object.
- Always use the object's local coordinate space (commonly referred to as **object space**) for the normal vector values in a normal map.

- Make sure your mesh object's geometry is oriented the same way in your modeler as it appears in Away3D. If the local axes don't align, rotate your model to match them up before generating the normal map.
- The expected mapping of color channels to normal vector coordinates in Away3D is RGB to XYZ (red is X, green is Y, and blue is Z).

Once your normal maps have been created, they can be imported into Away3D as JPG or PNG images and used with any DOT3 material to produce n rmal map shading.

Using DOT3 materials in Away3D

Materials in Away3D that use normal mapping in some way have a prefix of `Dot3` in their class name. Let's extend the `Chapter05SampleBase` class to create an example using normal map shading on all our objects.

```
package flash3dbook.ch05
{
  import away3d.core.math.*;
  import away3d.core.utils.*;
  import away3d.lights.*;
  import away3d.materials.*;
  import away3d.primitives.*;

  import flash.display.*;
  import flash.events.*;

  [SWF(width="800", height="600")]
  public class UsingNormalMaps extends Chapter05SampleBase
  {
    private var _dirLight : DirectionalLight3D;

    [Embed(source="../../../assets/ch05/sphere_normals.jpg")]
    private var SphereNormals : Class;

    [Embed(source="../../../assets/ch05/ape_normals.jpg")]
    private var ApeNormals : Class;

    [Embed(source="../../../assets/ch05/cube_normals.jpg")]
    private var CubeNormals : Class;

    [Embed(source="../../../assets/ch05/redapple.jpg")]
    private var AppleImage : Class;

    private var _angle : Number = 0;

    public static const RADS_PER_DEG : Number = Math.PI / 180;
    public function UsingNormalMaps()
    {
      super();

      _cube.mappingType = CubeMappingType.MAP6;

      _createLights();
    }

    protected function _createLights() : void
    {
    }

    protected override function _createMaterials() : void
    {
    }
  }
}
```

We start by embedding three pregenerated normal map images, one for each mesh object. The `ape_normal.jpg` and `sphere_normal.jpg` images are generated by smoothing face normals of the existing geometry, while the `cube_normal.jpg` has been created with an additional height map to add a bit of extra detail to the cube's surface, as you will see. Our next embedded image, the `redapple.jpg` shown in previous examples, will be used for our base material texture on each object. All images can be found in the chapter files from the `Downloads` section of `www.friendsofed.com`. We finish our property definitions with a constant `RADS_PER_DEG` and a global variable `_angle` that will be used for rotation effects later in the class.

The constructor in the preceding code modifies the `mappingType` property of the cube primitive to `MAP6`, a special texture-mapping mode that is a requirement for the `cube_normal.jpg` texture, because it uses a different area of the texture for mapping each side of the cube. We then call `_createLights()`, a new method included in the class to setup the lights in a similar manner to the previous `FlatShadingMaterials` example. Finally, we override the `_createMaterials()` method, ready for our custom code. Before we define our materials, let's create our light sources by adding the following code to the `_createLights()` method:

```
_dirLight = new DirectionalLight3D();
_dirLight.direction = new Number3D(70, 500, -70);
_dirLight.ambient = 0.1;
_dirLight.diffuse = 0.5;
_dirLight.specular = 1;
_dirLight.brightness = 2;
_view.scene.addLight(_dirLight);
```

This creates a directional light source identical to the one in the `FlatShadingMaterials` example, so that a direct comparison can be made.

Note that DOT3 materials react to point light sources in the current Flash 10 version of Away3D but not in the Flash 9 version. In the interest of maintaining maximum compatibility, we will only be looking at the directional light implementation in our example.

Now, we need to create the three `BitmapData` objects used for each of our mesh object normal maps and the one `BitmapData` object used for all base textures. Once again, this is easily done using the `Cast` class by adding the following code to the `_createMaterials()` method:

```
var texture : BitmapData = Cast.bitmap(AppleImage);
var sphereMap : BitmapData = Cast.bitmap(SphereNormals);
var apeMap : BitmapData = Cast.bitmap(ApeNormals);
var cubeMap : BitmapData = Cast.bitmap(CubeNormals);
```

With these in place, we can start creating the DOT3 materials. We use the `Dot3BitmapMaterial` class located in the `away3d.materials` package, adding the following code to the `_createMaterials()` method:

```
var sphereMaterial : Dot3BitmapMaterial;
sphereMaterial = new Dot3BitmapMaterial(texture, sphereMap);
sphereMaterial.specular = 0xFFFFFF;

var apeMaterial : Dot3BitmapMaterial;
apeMaterial = new Dot3BitmapMaterial(texture, apeMap);
apeMaterial.specular = 0xFFFFFF;

var cubeMaterial : Dot3BitmapMaterial;
cubeMaterial = new Dot3BitmapMaterial(texture, cubeMap);
cubeMaterial.specular = 0xFF0000;
```

The `Dot3BitmapMaterial` object requires two arguments in its constructor: the first for the bitmap data of the texture map (the same data used in a standard `BitmapMaterial`) and the second for the bitmap data of the normal map. Notice that in the preceding code, we set a `specular` property on the `Dot3BitmapMaterial` objects after instantiation. This is interpreted in a similar way to the `specular` property seen earlier on the `ShadingColorMaterial` object, defining the color of the reflected specular component of the light source. In the cases of `sphereMaterial` and `apeMaterial`, the `specular` property is set to white, but for `cubeMaterial`, it is set to red.

Now, we just need to apply these materials to their respective mesh objects by adding the following code to the `_createMaterials()` method:

```
_cube.material = cubeMaterial;
_cube.ownCanvas = true;
_sphere.material = sphereMaterial;
_sphere.ownCanvas = true;
_ape.material = apeMaterial;
_ape.ownCanvas = true;
```

While setting the `material` properties of each object, we also isolate each material to a unique rendering sprite for each object, known as a canvas. Because many shading materials are layered, isolating the objects that use them to their own canvas helps with the speed and consistency of the rendering process. This setup is easily configured by setting the `ownCanvas` property of each mesh object to true.

Compiling the code at this point will display something similar to Figure 5-10. On the sphere and monkey model objects, the shading appears smooth without a hint of the underlying polygons making up the mesh. The normal map on the cube illustrates how normal mapping can be used to add surface detail to an otherwise flat object—the apparent indentations on each side make it look like a die, despite the fact that no geometry has been added to the cube primitive.

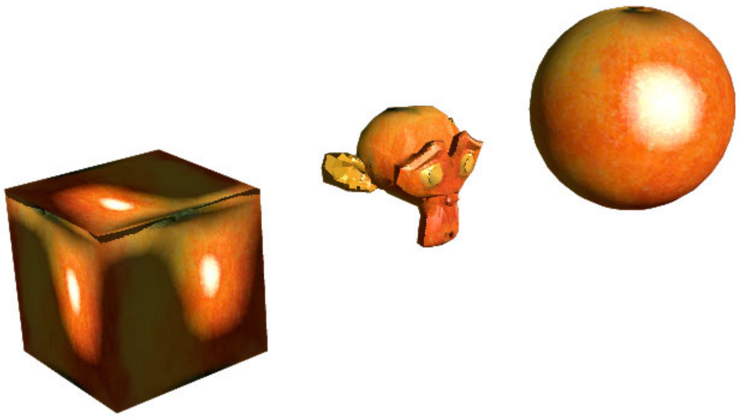


Figure 5-10. Normal map shading applied to our test objects, using externally created normal map images in the UsingNormalMaps example

As a final touch to the `UsingNormalMaps` example, let's override the `_onEnterFrame()` method to create a moving light source. Add the following code to the bottom of the class:

```
override protected function _onEnterFrame(ev : Event) : void
{
  _angle = (_angle + 5) % 360;

  var x : Number= 100 * Math.cos(RADS_PER_DEG * _angle);
  var y : Number = 50;
  var z : Number= 100 * Math.sin(RADS_PER_DEG * _angle);

  _dirLight.direction = new Number3D(-x, -y, -z);

  super._onEnterFrame(ev);
}
```

Here, we use our global `_angle` variable to increment a rotation value of 5 degrees every frame and then apply it to the direction vector of our directional light source with the help of some trigonometry. We calculate the x, y, and z components required for the `direction` vector of the directional light and then apply them in a new `Number3D` object. To ensure the rest of the example functions normally, we finish our modifications by calling the superclass `_onEnterFrame()` method. Recompiling the example rotates the directional light object around the scene with the light maps of the DOT3 materials updating in real time.

Using environment shading

When considering highly reflective materials, the standard shading options you have seen so far are a little lacking in detail. What if you want an object to appear made of glass or chrome? The answer lies in a technique known as **environment shading**. This uses a cached image of the surroundings as the basis for a light map, overlaying a calculated reflection of the image onto an underlying color or texture.

Environment shading employs a technique called **environment mapping** to draw its light map. Let's create an example to demonstrate the effect by implementing the `EnviroBitmapMaterial` and `EnviroColorMateiral` classes in Away3D. Once again, we begin by extending the `Chapter05SampleBase` class with the following code:

```
package flash3dbook.ch05
{
  import away3d.core.utils.*;
  import away3d.materials.*;

  import flash.display.*;
  [SWF(width="800", height="600")]
  public class EnvironmentMaterials extends Chapter05SampleBase
  {
    [Embed(source="../../../assets/ch05/environment.jpg")]
    private var EnviroMap : Class;

    [Embed(source="../../../assets/ch05/redapple.jpg")]
    private var AppleImage : Class;

    private var _colorMaterial : EnviroColorMaterial;
    private var _bitmapMaterial : EnviroBitmapMaterial;

    public function EnvironmentMaterials()
    {
      super();

      _toggle();
    }

    protected override function _createMaterials() : void
    {
    }

    protected override function _toggle() : void
    {
    }
  }
}
```

As usual, we start our class definition by embedding the textures used in the example. You are familiar enough with the `redapple.jpg` texture by now, but the following `environment.jpg` file is a new image to be used as our environment mapping texture. Opening it up, you will see an image of a garden that has been morphed as though viewed through a fish-eye lens. Panoramic projections such as this work well with environment mapping materials and can be easily generated with the right image editing software. However, perfectly acceptable results can be achieved using a regular image of the scene you wish to see reflected in the surface of the material.

The next global variables we declare are two placeholders for the environment mapping material objects. The `EnviroColorMaterial` object applies an environment map to a standard color material, while the `EnviroBitmapMaterial` object does the same to a bitmap material. The constructor in the preceding code contains an initializing call to the `_toggle()` method, and the class is rounded off with the usual `_createMaterial()` and `_toggle()` stub methods.

To complete the example, we first need to fill out the `_createMaterials` method with the following code:

```
var texture : BitmapData = Cast.bitmap(AppleImage);
var envMap : BitmapData = Cast.bitmap(EnviroMap);

_colorMaterial = new EnviroColorMaterial(0xffcc66, envMap);
_bitmapMaterial = new EnviroBitmapMaterial(texture, envMap);

_cube.material = _colorMaterial;
_sphere.material = _colorMaterial;
_ape.material = _bitmapMaterial;
```

Here, we start with the familiar extraction of embedded image bitmap data using the `Cast` class and then use the result to define our environment materials. The `EnviroColorMaterial` takes a color value as its first constructor argument, while the `EnviroBitmapMaterial` takes the bitmap data for the material texture. They both expect an environment image as their second constructor parameter, which will be blended with the base texture or color of the material. We finish by applying the `EnviroColorMaterial` object to the cube and sphere primitives, and the `EnviroBitmapMaterial` object to the monkey model.

Compiling the example will display the output shown in Figure 5-11. As you rotate the view with the mouse, you'll see that both materials appear to be reflecting an invisible background environment.

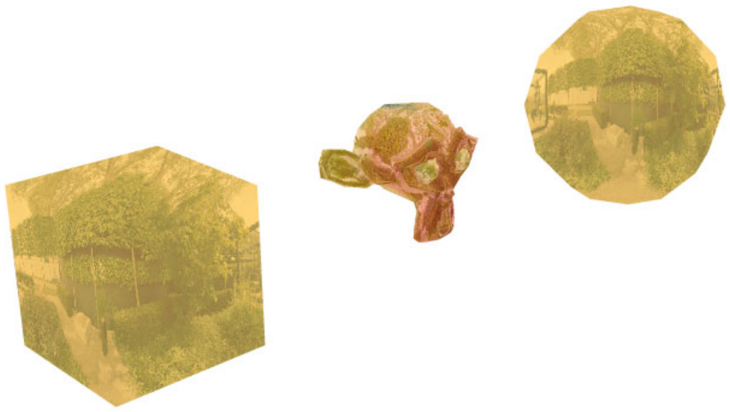


Figure 5-11. Environment materials applied to all mesh objects in the `EnvironmentMaterials` example

In Away3D, environment materials use a `reflectiveness` property to control the degree to which an environment texture is reflected in the material's surface. This is represented by a fractional value between 0 and 1, with the default set to 0.5 (what you have seen so far). To compare different settings, add the following lines of code to the `_toggle()` method:

```
switch (_state) {
  case 0:
      _colorMaterial.reflectiveness = 0.1;
      _bitmapMaterial.reflectiveness = 0.1;
    _state = 1;
    break;
  case 1:
      _colorMaterial.reflectiveness = 0.9;
      _bitmapMaterial.reflectiveness = 0.9;
    _state = 0;
    break;
}
```

Recompile the `EnvironmentMaterials` example, and click the mouse anywhere inside the stage to see the `reflectiveness` property swap between 0.1 and 0.9 for both color and bitmap environment materials. Figure 5-12 shows the appearance of each setting side by side. Environment shading is one of the most efficient rendering shading techniques in Away3D, which make it a very versatile material option.

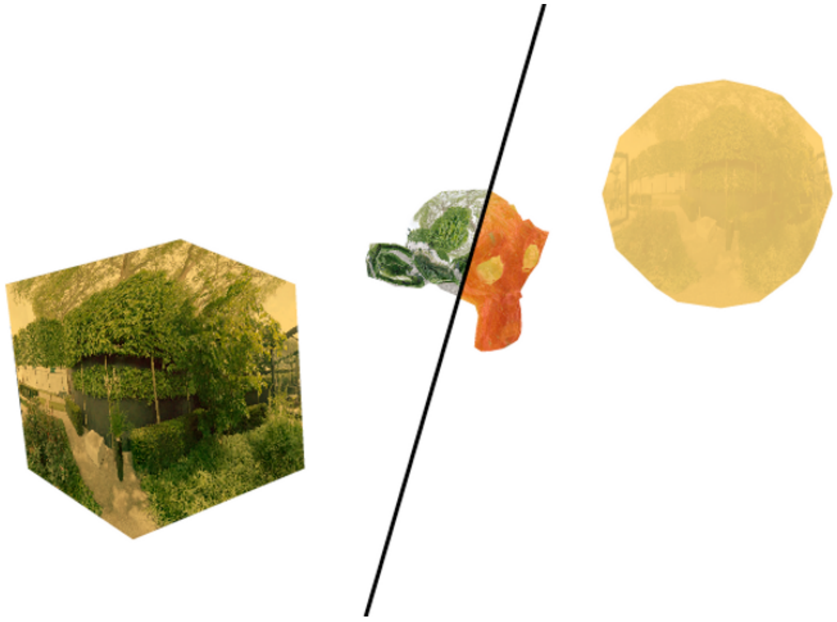


Figure 5-12. Adjusting the reflectiveness property of the environment materials in the EnvironmentMaterials example. On the left, reflectiveness is set to 0.9, on the right it is set to 0.1.

Using animated and interactive materials

An **animated material** extends the functionality of a standard bitmap material to allow the contents of the texture image to be animated in real time. Using the MovieMaterial class in Away3D, it is possible to project time-line–based animations contained in a MovieClip or code-based animations contained in a Sprite onto the surface of a 3D mesh object. As the examples in this section will demonstrate, animated materials can also be made interactive through the surface of a 3D mesh object by setting the interactive property of a MovieMaterial object to true.

Using the MovieMaterial class

The MovieMaterial class is found in the usual away3d.materials package. Let's explore the functionality available by extending the Chapter05SampleBase class with the following document class:

```
package flash3dbook.ch05
{
  import away3d.primitives.*;
  import away3d.materials.*;
  import away3d.events.*;

  [SWF(width="800", height="600")]
  public class UsingAnimatedMaterials extends Chapter05SampleBase
  {
    [Embed(source="../../../assets/ch05/animatedTexture.swf")]
    private var AnimatedTexture : Class;

    [Embed(source="../../../assets/ch05/interactiveTexture.swf")]
    private var InteractiveTexture : Class;

    private var _movieMat :MovieMaterial;

    public function  UsingAnimatedMaterials()
    {
      super();
    }

    protected override function _createScene():void
    {
    }

    protected override function _createMaterials() : void
    {
    }
  }
}
```

In the preceding code, we embed two SWF files as global properties AnimatedTexture and InteractiveTexture. These will be cast to Sprite on instantiation for use in our MovieMaterial object.

In this case, we override the `_createScene()` method, because we want to reconfigure our scene to display a single `_cube` primitive. We finish the class definition by overriding the `_createMaterials()` method with the usual method stub.

Now, let's create our single cube object by adding the following code to the `_createScene()` method:

```
_cube = new Cube();
_cube.width = 75;
_cube.height = 75;
_cube.depth = 75;
_view.scene.addChild(_cube);
```

This creates a cube with a size in all dimensions of 75 units and adds it to the scene. To complete the example, we need to apply a `MovieMaterial` object to the surface of the cube by adding the following code to the `_createMaterials()` method.

```
_movieMat = new MovieMaterial(new AnimatedTexture());
_cube.material = _movieMat;
```

Here, we use an instance of the `AnimatedTexture` class containing the embedded SWF file `animatedTexture.swf` for the required movie clip instance, passed in the constructor of the `MovieMaterial` object. Both embedded SWF files used here can be found in this chapter's resource download available at `www.friendsofed.com`. The `animatedTexture.swf` file contains a simple looping animated movie that can be previewed by opening the file using the stand-alone Flash Player. Compiling the code will display a single cube primitive with the animating movie clip projected onto every side. Notice that the default mapping used for this cube's UV coordinates projects an identical image of the texture onto each side of the cube, unlike the mapping used in the previous `UsingNormalMaps` example.

We can adapt the `UsingAnimatedMaterials` example to display an interactive movie material by replacing the preceding code for the `_createMaterials()` method with the following:

```
_movieMat = new MovieMaterial(new InteractiveTexture());
_movieMat.interactive = true;
_movieMat.smooth = true;
_cube.material = _movieMat;
```

Here, we pass an instance of the `InteractiveTexture` class in the constructor argument of the `MovieMaterial` object. Opening the associated `inteactiveTexture.swf` file in the stand-alone Flash Player will display a collection of Flash Professional UI components. These are ideal for use in an interactive texture demonstration, because they react to mouse events such as rollovers and clicks. To enable interactivity, we set the `interactive` property of the `MovieMaterial` object in the preceding code to `true`. A property called `smooth` is also set to `true` to produce an antialiased result for the texture mapping on the surface of the cube object. This helps when dealing with interface components in a texture, as items such as text and buttons appear clearer when antialiased.

Recompiling the example displays a cube similar to the one shown in Figure 5-13. Each side has the contents of the `inteactiveTexture.swf` file projected onto its surface, and using the mouse, you will quickly discover that interactivity is preserved.

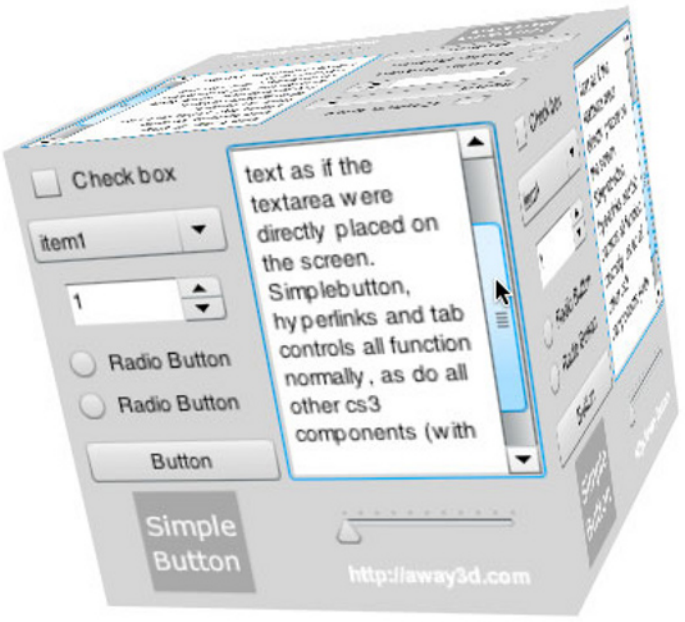

Check box
item1
1
Radio Button
Radio Button
Button
Simple Button
text as if the textarea were directly placed on the screen. Simplebutton, hyperlinks and tab controls all function normally, as do all other cs3 components (with
http://away3d.com

Figure 5-13. Interactive UI components mapped onto the surface of a cube using the MovieMaterial object

At times you might want to use `MovieMaterial` for movie clips that are not constantly animating. When the movie clip is static, redrawing the material is a waste of resources. To prevent a `MovieMaterial` from automatically redrawing, set its `autoUpdate` property to `false`. Setting it to `true` will resume updating every time the view is rendered. If at any point you need to refresh a `MovieMaterial`, without activating automatic redrawing, you can invoke its `update()` method.

Using the VideoMaterial class

The `VideoMaterial` class is an animated material that extends the functionality of the `MovieMaterial` class to accept FLV files as its animating source. The playback of Flash video inside a `VideoMaterial` object is achieved through transport control methods on the object that mirror the standard video component controls in Flash. In its simplest form, a `VideoMaterial` implementation is set up by creating a class instance and setting its `file` property to the location of an FLV file. We can try this out by replacing the contents of the `_createMaterials()` method in the previous `UsingAnimatedMaterials` example with the following code:

```
var videoMat : VideoMaterial = new VideoMaterial();
videoMat.file = '../assets/ch05/Away3D_Showreel2010.flv'
_cube.material = videoMat;
```

Here, we are setting the `file` property of the `VideoMaterial` object to a string value representing the path to our FLV file, which once again can be obtained through the resource download available at `www.friendsofed.com`. Recompiling the example will display the chosen video file playing back on the surface of the cube primitive. You can experiment further by using the `play()` and `pause()` methods of the `VideoMaterial` object to start and stop playback and the `seek()` method to jump back and forth in the video stream, as you would with a native `flash.net.NetStream` object.

Summary

In this chapter, we have covered the majority of material types that are available in Away3D. An important concept to take with you is that all materials are applied to 3D objects in the same way. However, a light source object in the scene is an extra requirement of a shading material if it is to function correctly.

With more complex material types such as those concerned with shading, the required real-time computations can be very processor heavy. Where possible, it is best to try to avoid using a lot of light sources and shading materials at the same time; often, it is possible to fake shading using the texture baking technique that requires no real-time processing overhead. It is also possible to optimize the use of animated materials with the use of the `autoUpdate` property. In Chapter 10, we will look further at some of the ways we can optimize the usage of materials.

Here is the summary of topics covered in this chapter:

- Materials are created by instantiating classes from the `away3d.materials` package. They are applied by setting the `material` property on a 3D mesh object or 3D element that supports them.
- Shading can be achieved in Away3D by adding at least one instance of a light source class from the `away3d.lights` package to the scene and applying a shading material to any 3D mesh object in the scene to produce shading results.
    - The `PointLight3D` class creates a light source that emits light in all directions.
    - The `DirectionalLight3D` class creates a light source that emits light in a single direction from a near-infinite distance.
    - The `AmbientLight3D` class creates an ambient light source that raises the overall lighting level in a scene.
- Visual results from shading materials are controlled from properties set on the light source (such as `ambient`, `diffuse`, and `specular` properties) as well as on the material object (such as the `color` property on a `ShadingColorMaterial` object).
- Normal maps are images that define the normal vectors of each texel on the surface of the 3D mesh object and can be used to separate light map detail from mesh detail.
    - Normal maps are usually created in a 3D modeling application such as Blender or 3ds Max.
    - Normal maps are implemented in Away3D using DOT3 materials.
    - In Flash 9, normal maps are restricted to working with directional light sources only.
- The mouse can interact with the contents of a movie clip texture through the surface of a 3D mesh using a `MovieMaterial` object with its `interactive` property set to `true`.
- Materials can be animated by using a `MovieMaterial` object with an animated movie clip, or by using a `VideoMaterial` object with an FLV video file.

In the next chapter, we will explore how the use of 3D vector shapes in Away3D can completely transform the way you import SWF files and text into your 3D projects, as well as the visual benefits of using curves instead of lines in a scene.

Chapter 6

Vector Shapes and Text in 3D

Up to this point, the 3D geometry we have created and imported into Away3D has been defined using groups of faces and segments that consist of vertices connected by straight lines. This approach is the most commonly used by 3D modeling applications and the majority of 3D engines for defining 3D geometry, partly because it allows for easy interchange between applications and partly because the hardware acceleration used by many graphical libraries is designed to work with collections of straight-edged polygons.

However, this format does no favors for those of us wanting to create smoothly varying surfaces in 3D, because the rendering process for any shape following a curved outline involves tessellating many straight edged polygons together. An accelerated engine has little problem working this way, but the same approach is pretty detrimental for Flash where every extra polygon impacts performance due to the limitations of software rendering.

From working with any of the graphical tools available for producing Flash content, you'll be familiar with the concept of vector graphics that allows the creation of shapes with curves as well as straight edges. Because we are not pandering to the restrictions of a GPU, the same drawing techniques can be applied in 3D to produce perfectly curved surfaces without the need for tessellation. This chapter gives an overview of the use of vector graphics in Away3D, exploring some of the advantages and limitations that are encountered. To start with, let's take a look at some general concepts surrounding vector graphics.

Working with vector graphics

In the world of digital graphics, there are generally two methods used for creating and storing images. The first is raster (or bitmap) graphics, and the second is vector graphics. **Raster graphics** are stored as individual pixel color values that are drawn directly to screen when viewed, while **vector graphics** are stored as a series of shape definitions that are rasterized on the fly when required to be displayed.

Vector graphics vs. raster graphics

One huge advantage of vector graphics over raster graphics is their ability to scale with no loss of quality. A raster image is stored as a 2D array of pixels that has an optimum width and height on screen. Stretching or compressing a raster image to fit different dimensions almost always results in some loss of fidelity, because the color values of on-screen pixels have to be interpolated between the color values of pixels in a source image. As a result, straight lines become stepped, curves become blocky, and while antialiasing can go some way to disguise these artifacts, you are unlikely to achieve the same crisp lines as your original image. Vector graphics have none of these issues because the rasterization of an image occurs only when the graphics are drawn to screen, allowing image pixels to match up perfectly with screen pixels. Images are stored as a series of drawing commands, which can often be an efficient approach in terms of file size. If a vector image is scaled or transformed in any way, the coordinates in the drawing commands are adjusted before the rasterizing process begins, ensuring a crisp output.

As an example of the differences between raster graphics and vector graphics, consider a red circle with a radius of 50 pixels stored in both formats. To store the data as a raster image, we would require 100 × 100 pixels, each with a color value of 0xFFFF0000 (solid red) or 0x00000000 (transparent) for a total of 10,000 32-bit integer values. Resizing this image to twice its original size would double the area represented by each pixel onscreen and cause a pixilated look. To store the same red circle as a vector image, we would require the drawing instructions necessary to re-create the circle. The specific instructions would depend entirely on the graphics engine being used to draw them but, in this case, could be something as simple as "draw a red circle with a 50 pixel radius." Not only is this vastly more efficient in terms of storage space, but scaling comes very naturally. Resizing the image to twice its original size would simply replace the radius value of 50 pixels with a radius value of 100 pixels, and the next drawing operation would rasterize the circle to screen at twice the size while retaining the same level of fidelity.

Creating lines and curves

In Flash, we have a well-defined set of instructions for drawing vector graphics programmatically. The most basic of these are the `moveTo()`, `lineTo()`, and `curveTo()` commands from the `Graphics` class that allow us to draw curves and lines in a sequential order. We can create a closed loop to define a filled shape or an open length to define a line segment. Armed with these drawing commands, it is possible to represent almost any shape imaginable. For example, the circle mentioned previously can be constructed from four curves, while a rectangle can be constructed from four lines.

The same basic commands form the building blocks of our vector drawing routines in Away3D. Before we have a closer look at how they are implemented, let's briefly consider the process involved in converting these commands to rasterized images on the screen.

The definition of a line can be stored as two position vectors: a start position and an end position. To draw this data, the software renderer in Flash colors individual pixels on the screen using a line algorithm such as the **Bresenham line algorithm**, producing the final bitmap output displayed by our monitor.

The definition of a curve can take many forms, but the most common definition used in vector graphics is known as a **Bézier curve**. This type of curve is quick to rasterize, and the position at any point along its path can be easily calculated, hence its popularity.

In Flash, we are able to draw quadratic Bézier curves programmatically using the native drawing API. Quadratic Bézier curves are so called because of the quadratic formula used to trace their paths, and they require three position vectors for their definition: a start position, a control position, and an end position. Figure 6-1 depicts an example of a 2D quadratic Bézier curve, displaying its three defining points and the resulting path.

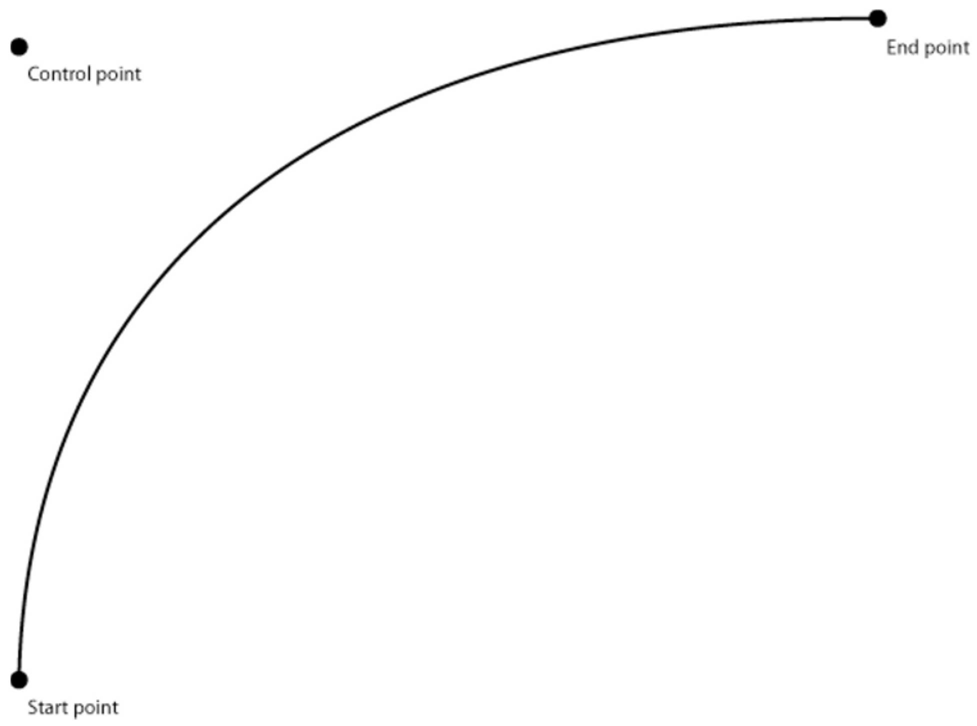


Figure 6-1. An example of the path drawn by a 2D quadratic Bézier curve, showing the position of the three points that define the shape of its path.

Bézier curves can work with any number of dimensions, which makes them ideal for defining curves in 3D. As you have seen in previous chapters, 3D polygons are built from a series of 3D vertices connected by straight lines. It is therefore possible to build a polygon with an irregular outline using vertices coupled with drawing commands, defining a series of 3D curves and lines that produce an open-ended segment or a closed vector shape.

While projecting a 3D line definition to screen is a trivial process, we have no native drawing routine in Flash for projecting a 3D curve to screen. To visualize these new 3D constructs, we have to use an approximation when it comes to rendering. In Away3D, vertex positions are projected to screen in the rendering process, producing a 2D set of position vectors. For 3D Bézier curves, drawing the projected representation of a shape using 2D Bézier curves is a close approximation to the actual representation and is easily done with the projected vertex positions and the native drawing API of Flash. The approach ignores any perspective distortion that would exist for the projected curves, but the difference is unnoticeable in the majority of cases.

Using the Away3D drawing API

As a generalization, all visible output drawn into the view in Away3D can be considered a collection of vector shapes. The most commonly seen shape in a `Mesh` object is the triangle, which is a vector shape made up of three vertex points connected by straight lines in a closed loop. This data is defined in the

Face class, but the definitions aren't required to stick to simple triangle geometry. To construct an irregular vector shape, we can use a set of methods on the Face class that mimic the native drawing API of the Graphics object in Flash, accessed through the graphics property of a Shape or Sprite object. For the purposes of analogy, a Face definition in Away3D can be treated in a similar way to a Shape definition in Flash. As an example, consider the following code snippet written to create an irregular vector shape in Flash:

```
var shape : Shape = new Shape();
shape.graphics.beginFill(0xffcc00);
shape.graphics.moveTo(-10, -10);
shape.graphics.lineTo(10, -10);
shape.graphics.lineto(10, 10);
shape.graphics.lineTo(-10, -10);
shape.graphics.endFill();
```

The preceding code executes a number of methods on the graphics property of a Shape instance, drawing a small, right-angled yellow triangle 20 pixels high by 20 pixels wide. If the Shape instance were attached to the display list, its contents would be rasterized on the stage of the Flash movie during the next frame draw. In Away3D, the code for creating an equivalent 3D shape looks very similar:

```
var face : Face = new Face();
face.material = new ColorMaterial(0xffcc00);
face.moveTo(-10, -10, 0);
face.lineTo(10, -10, 0);
face.lineTo(10, 10, 0);
face.lineTo(-10, -10, 0);
```

When we compare the preceding code to the previous native Flash code, there are three main differences. First, we are creating a new Face instance instead of a new Shape instance, which comes with its own set of custom drawing methods for constructing a complex shape outline. Second, rather than defining a fill style for the shape, we implement the same material configuration method used for Mesh objects in Away3D, setting the material property of the Face object to a yellow instance of the ColorMaterial class. Finally, because we are working in three dimensions, a third property is added to any drawing commands used on the Face object. The positions defined in the drawing commands are used to create vertices for the resulting shape that are then projected to screen in the usual way, defining the representation of the shape in the view.

Before we look further into the use of 3D drawing commands in Away3D, let's set up the base class required for all sample code in this chapter.

Preparing the chapter base class

The following class is used as a starting point for subsequent code examples. It defines a few basic controls for the camera, while carrying out the usual tasks to initialize the view. As with all code examples, copies of the classes created can be found inside the resource files for this chapter, available for download at www.friendsofed.com.

```
package flash3dbook.ch06
{
  import away3d.cameras.*;
  import away3d.containers.*;

  import flash.display.*;
  import flash.events.*;

  [SWF(width="800", height="600")]
  public class Chapter06SampleBase extends Sprite
  {
    protected var _camera : Camera3D;
    protected var _view : View3D;

    public function Chapter06SampleBase()
    {
      _createView();
      _createScene();
    }

    protected function _createView() : void
    {
      _camera = new TargetCamera3D();
      _camera.z = -1000;

      _view = new View3D();
      _view.x = 400;
      _view.y = 300;
      _view.camera = _camera;
      addChild(_view);
      addEventListener(Event.ENTER_FRAME, _onEnterFrame);
    }

    protected function _createScene() : void
    {
      // To be overridden
    }

    protected function _onEnterFrame(ev : Event) : void
    {
      _camera.x -= (_camera.x - 3*(mouseX - stage.stageWidth/2))/4;
      _camera.y -= (_camera.y + 2*(mouseY - stage.stageHeight/2))/4;
      _view.render();
    }
  }
}
```

In the preceding code, the regular base class methods and objects are constructed. One difference in the `_createView()` method is the use of the `TargetCamera3D` class for our view camera, which offers a subtly different way of navigating the scene compared to the usual `HoverCamera3D` class. The event listener created at the end of the `_createView()` method for the `ENTER_FRAME` event triggers the familiar _onEnterFrame() handler method on every frame so that our camera position, and view contents are updated. Also, the _createScene() method remains empty, so it can be overridden to add custom content in our example classes.

Drawing 3D vector shapes

As you saw in the previous section, Away3D has a set of drawing commands that mimic the native Flash drawing API for creating irregular 3D shapes. Let's start with an example exploring the various types of shape that can be created in this manner, extending the Chapter06SampleBase class with the following document class definition:

```
package flash3dbook.ch06
{
  import away3d.core.base.*;
  import away3d.materials.*;

  [SWF(width="800", height="600")]
  public class SimpleVectorShapes extends Chapter06SampleBase
  {
    override protected function _createScene():void
    {
      var mesh:Mesh = new Mesh();
      mesh.bothsides = true;

      var material : WireColorMaterial = new WireColorMaterial(0xFF0000);
      material.wireColor = 0x000000;
      material.thickness = 2;
      mesh.material = material;

      _view.scene.addChild(mesh);
    }
  }
}
```

Here, we override the _createScene() method with some code that adds an instance of the Mesh class to the scene with bothsides set to true and material set to an instance of the WireColorMaterial class, defining a red color fill with a 2-pixel black border. The mesh object will act as our canvas for adding a variety of vector shapes, represented by individual Face objects. Faces in Away3D always represent closed loops—if you want an outline that doesn't close the start and end points in a shape definition, you need to use a Segment object instead.

Creating simple shapes with straight lines

To begin with, let's take a look at drawing some regular 3D shapes that have straight edges. The simplest of these is the familiar triangle, although this being somewhat of a standard, it is also possible to create a triangle face using the built-in vertex definitions in the Face class constructor. We'll be investigating this approach in more detail in Chapter 7, but for now, let's stick to the process of shape creation using the Away3D drawing commands, adding the following code to the end of the _createScene() method:

```
// Triangle.
var face0:Face = new Face();
face0.moveTo(-50, -50, 0);
face0.lineTo(50, 50, 0);
face0.lineTo(-50, 50, 0);
face0.lineTo(-50, -50, 0);
face0.offset(-300, 0, 0);
mesh.addFace(face0);
```

Here, we create an instance of the `Face` class and use the 3D `moveTo()` and `lineTo()` methods to draw our triangle shape. Imagine an invisible **active drawing position** that exists like a pen you can move over a piece of paper. The `moveTo()` method repositions the pen without affecting the paper, while the `lineTo()` method draws a straight line with the pen from its current position to the one given by the (x, y, z) arguments in the `lineTo()` method.

The active drawing position of a shape defaults to the origin (0, 0, 0) of the containing mesh. In the preceding code, we reset this position to (–50, –50, 0) with a `moveTo()` command, defining the location of the bottom-left corner of our triangle. From here, we draw a line diagonally up to the top-right corner of our triangle at (50, 50, 0) and then straight across to the top-left corner at (–50, 50, 0). The final line closes the path by returning the drawing position to (–50, –50, 0).

Before our new face is added to `mesh`, we call the `offset()` method. All preceding drawing instructions have been stored internally in our created `Face` instance, and the `offset()` method updates the stored coordinates of each command by incrementing the x, y, and z values a specified amount. This offers a useful way of repositioning the geometry inside a single face after it has been created, allowing `moveTo()` and `lineTo()` methods to use easily visualized, localized coordinates that are then incremented by the `offset()` method. Here, we offset our created face 300 units to the left to leave room for future shapes, and then add it to be visible in the scene using the `addFace()` method of our mesh object.

When using the drawing commands in Away3D, we are not limited to creating polygons with a set number of sides. To illustrate this, let's create a square in our example by adding the following code to the end of the `_createScene()` method..

```
// Square.
var face1:Face = new Face();
face1.moveTo(-50, -50, 0);
face1.lineTo(50, -50, 0);
face1.lineTo(50, 50, 0);
face1.lineTo(-50, 50, 0);
face1.lineTo(-50, -50, 0);
face1.offset(-180, 0, 0);
mesh.addFace(face1);
```

The approach used is nearly identical to our previously created triangle shape, with one additional `lineTo()` method. The process begins by moving the active drawing position to the bottom-left corner of the square at (–50, –50, 0), and lines are then drawn sequentially between each of the four corners, in a counterclockwise fashion.

As with the triangle shape, we use localized coordinates that are globally incremented by the `offset()` method to leave room for subsequent shapes.

Compiling the code at this point should give you a similar output to the one displayed in Figure 6-2, with a triangle and a square positioned side by side, colored red with a black border.

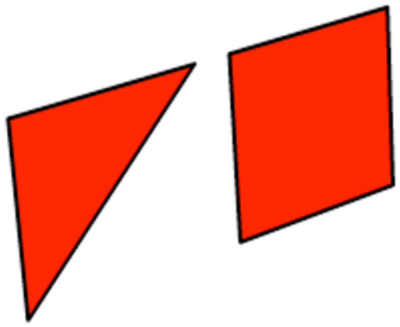


Figure 6-2. Triangle and square shapes drawn using the Away3D vector drawing commands in the SimpleVectorShapes example

Creating curved shapes

So far, we have drawn nothing in the `SimpleVectorShapes` example that couldn't be emulated in an imported mesh model. However, the next step for us is to apply the same API to the creation of curved shapes, opening a whole world of new possibilities.

The brute force approach of creating a perfectly curved line with a collection of standard straight lines is quite a wasteful process in Away3D, and you are often restricted to only creating the necessary detail for a single level of zoom. Defining a 3D shape using vector definitions has the advantages of being quick to draw and retaining a smooth appearance from any distance. As an addition to the `SimpleVectorShapes` example, let's add the following code to the `_createScene()` method that creates a circle shape from a collection of curved lines using the `curveTo()` drawing command:

```
// Curved shape.
var face2:Face = new Face();
face2.moveTo(-50, 0, 0);
face2. curveTo(-50, -50, 0, 0, -50, 0);
face2.curveTo(50, -50, 0, 50, 0, 0);
face2.curveTo(50, 50, 0, 0, 50, 0);
face2.curveTo(-50, 50, 0, -50, 0, 0);
face2.offset(-60, 0, 0);
mesh.addFace(face2);
```

Because we are restricted to using quadratic Bézier curves in our 2D drawing operations, we have the same restriction imposed on our 3D drawing operations. Here, our curve commands specify two 3D positions to define a Bézier curve with a similar form to the representation in Figure 6-1. The active drawing position is used as the start point of the curve, the first `Number` triplet in the method arguments is used as the control point of the curve, and the second `Number` triplet is used as the end point or the curve. After tracing our circle outline using four such curve commands, we use the `offset()` method once again to shift the entire shape definition 60 units to the left. Recompiling the example will display the output shown in Figure 6-3.

It is important to note that the resulting circle representation is only an approximation, as quadratic Bézier curves do not produce exactly the same curves as those required for a perfectly circular shape. We do not currently have a `drawCircle()` method available from our 3D drawing commands in Away3D, but the approximation we have generated here is a close match. For greater accuracy, it would be possible to create the same shape from a greater number of approximating Bézier curves, but this approach would generate more vertices in the shape and therefore add to the processing requirements during a render.

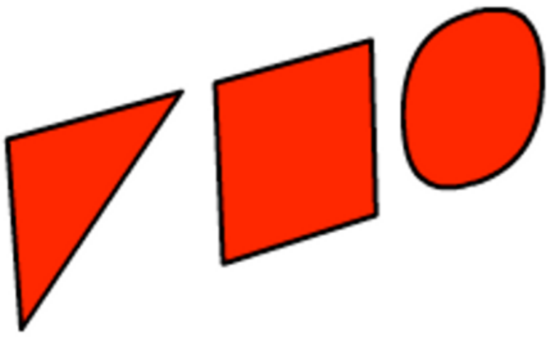


Figure 6-3. Faces using the Away3D vector drawing commands from our previous version of the SimpleVectorShapes example, with a new face added that implements the curveTo() command to create an approximation of a circle shape

Creating open-ended line segments

As mentioned at the beginning of this section, we discriminate between open-ended outlines and closed loops in Away3D by using different classes. So far, the `Face` class we have used will always close the loop of the shape defined, creating a form that represents a solid surface. If we don't want to close our shape, we can use the `Segment` class to create irregular line segments. To demonstrate this, we can include an irregular line segment our `SimpleVectorShapes` example by adding the following code to the end of the `_createScene()` method:

```
// Open-ended line segment
var segment0:Segment = new Segment();
segment0.moveTo(10, 50, 0);
segment0.lineTo(60, 50, 0);
segment0.lineTo(60, 0, 0);
segment0.curveTo(110, 0, 0, 110, -50, 0);
mesh.addSegment(segment0);
```

The drawing commands available on the `Segment` class are identical to the ones available on the `Face` class, but the result only renders the outline of the created shape, ignoring any definitions in the material relating to surface fills. Once we have built up our shape definition, the segment object is added to the scene using the `addSegment()` method available on our mesh object.

Recompiling the SimpleVectorShapes example displays the Segment shape alongside our previously created shapes. The visual style used for the line is taken from the same WireColorMaterial object set on the mesh, which applies a black stroke 2 pixels thick to our line segment.

Creating nonplaner shapes

At this point, all our vector shapes in the SimpleVectorShapes example have been drawn on a 2D XY plane in our scene. This is frequently what we require for irregular faces, but it doesn't always have to be the case. It is possible to use a collection of defining positions placed anywhere in space to produce a nonplaner shape, although there are certain visual restrictions to be aware of when doing this. As an example, add the following code to the end of the createScene() method:

```
// Non-planer shape.
var face3:Face = new Face();
face3.moveTo(-50, -50, 0);
face3.curveTo(0, -50, 50, 50, -50, 0);
face3.lineTo(50, 50, 0);
face3.curveTo(0, 50, 50, -50, 50, 0);
face3.lineTo(-50, -50, 0);
face3.offset(180, 0, 0);
mesh.addFace(face3);
```

At a glance, this looks very similar to the approach we have been using to construct our previous vector shapes. The difference here is that we are now using more than a single Z value for the positions passed to our drawing commands, so that the vertices produced are no longer all located in the same plane.

Recompiling the code reveals that the new shape is a variant of our previously created square, with its top and bottom edges bent inward as if the face were a piece of card being gently pinched at the sides. Panning the camera around to get a good look at the shape, you will notice some peculiarities at certain orientations. This output is illustrated in Figure 6-4, with our new shape displayed from various angles. The image on the far right demonstrates the limitations of such a shape in 3D, with the renderer having a hard time realizing the solid nature of the shape object from the projected 2D drawing commands it is given. In this case, we would expect an extra vertical line to be defined at the back of the shape to maintain the perceived volume of the object we are trying to represent. This is currently a limitation of using the Away3D drawing commands in this manner.

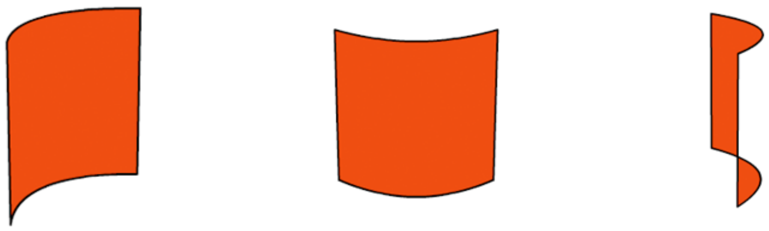


Figure 6-4. This is the displayed output of the nonplaner shape created in the SimpleVectorShapes example, depicted from a number of different angles. The illusion is maintained until the shape is viewed from the direction used in the image on the right, whereupon the basic rendering technique produces an incorrect representation.

Creating shapes with holes

It is possible to create shapes with holes using the Away3D drawing commands, even though it might not be immediately apparent how. The procedure relies on the **winding** property of a shape (the drawing direction) that can be either positive or negative. This property is used in the native drawing API of Flash, where intersecting shapes of different winding result in one shape subtracting itself from the other.

Because we are ultimately using the native drawing API to render our shapes in the view, we can apply the same winding logic to objects in 3D. A shape has positive winding if its vertices are arranged in a clockwise fashion and has negative winding if its vertices are arranged in a counterclockwise fashion. To create a shape with a hole, all we need do is define an outer shape with its winding in one direction and then define an inner shape with its winding in the other direction. Let's illustrate the effect in our `SimpleVectorShapes` example, starting with the outer shape created by adding the following lines of code to the `_createScene()` method:

```
// Shape with a hole.
var face4:Face = new Face();
face4.moveTo(-50, -50, 0);
face4.lineTo(50, -50, 0);
face4.lineTo(50, 50, 0);
face4.lineTo(-50, 50, 0);
face4.lineTo(-50, -50, 0);
```

You may recognize these drawing commands from our square shape created earlier in this chapter. The points are defined in counterclockwise order, starting at the bottom-left corner, followed by the bottom-right, top-right, top-left, and back to bottom-left corner. To create a hole in this arrangement, we need to add a further series of drawing commands to the `face4` object, ordered in a clockwise order so that the resulting shape is subtracted from the one defined here.

```
face4.moveTo(30, 30, 0);
face4.lineTo(0, -30, 0);
face4.lineTo(-30, 30, 0);
face4.lineTo(30, 30, 0);
```

In this case, the active drawing position is first moved to a region in the upper-right corner of the square at (30, 30, 0), and lines are then drawn in a clockwise order to a region in the lower center of the square at (0, –30, 0), the upper-left corner of the square at (–30, 30, 0) and finally back to our starting position in the upper-right corner.

We complete the creation of our new shape by adding the following lines of code to the end of the `_createScene()` method, applying an offset 300 units to the right and adding our `Face` instance to the mesh in the usual way.

```
face4.offset(300, 0, 0);
mesh.addFace(face4);
```

Recompiling the `SimpleVectorShapes` example will display something similar to the output shown in Figure 6-5, with our final shape drawn as a square with an inverted triangle cut out of its center.

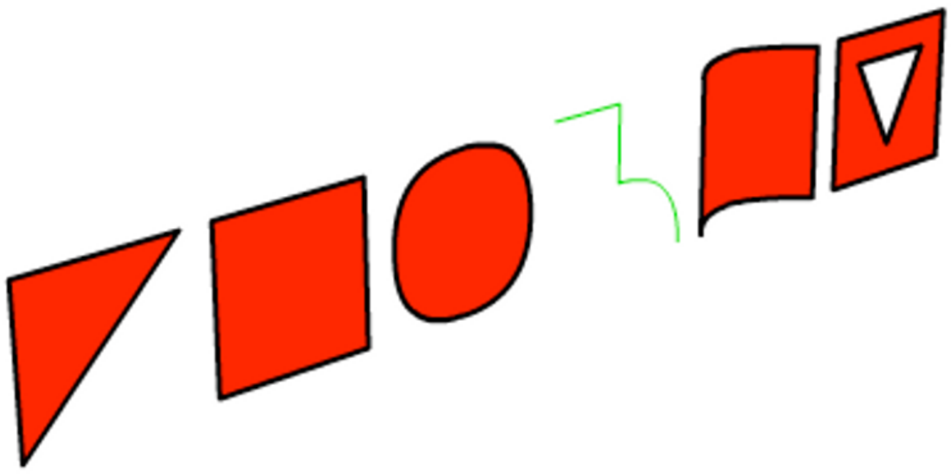


Figure 6-5. The final output of the SimpleVectorShapes example, showing all six shapes created in this section using the Away3D vector drawing commands

The methods explained here are useful for getting into the nitty-gritty of irregular shape creation in Away3D. However, using them to produce anything remotely complex can often be a laborious task. The next section looks at a way of importing prebuilt shapes in much the same way we import prebuilt 3D models. In this case, the editor used to create the content isn't some fancy 3D modeling software but the familiar Flash IDE!

Importing 3D vector shapes

When constructing 3D shapes such as the ones seen in the previous section, using a visual tool is quite often quicker and easier than typing in drawing commands. In Flash Professional, the currently available drawing tools allow us to create complex 2D shapes with ease. Away3D enables the creation of 3D content with these tools by offering a custom importer for the SWF format that converts the created 2D shapes into 3D shape data. The method of importing an SWF file for use in this manner is similar to the approach discussed in Chapter 4 for importing 3D models.

Extracting vector shapes from an SWF file

Depending on the environment you are using for developing your Away3D project, a typical workflow for importing vector shapes from an external SWF file includes the following steps:

1. Open a new FLA file, and create some vector shapes using the drawing tools of the Flash IDE.
2. Place your shapes inside a library symbol (or number of symbols), giving each symbol a linkage class name by selecting the `Export For ActionScript` check box in the `Properties` panel.
3. Publish the FLA as an SWF file.

4. From inside your Away3D project, create a new instance of the Loader3D class found in the away3d.loaders package and a new instance of the Swf class for parsing SWF files. Set the libraryClips property of the Swf instance to an array of symbol names you wish to convert to 3D shapes.
5. Call the loadGeometry() method on the created instance of the Loader3D class, using the file path string of the SWF file containing your vector shapes and the instance of the Swf class as arguments.
6. Once the SWF file has finished loading, each parsed library symbol is represented as a Mesh object. All meshes are stored as children of the generated 3D container of the Swf class, accessed from the handle property on the Loader3D instance.

This foolproof approach works in every development environment including Flash Builder, FDT, FlashDevelop and Flash Professional. However, if your Away3D project is constructed inside Flash Professional, you can use a simpler approach to importing vector shapes that does away with using a separate SWF file:

1. Inside your project FLA, create your linked library symbols containing the vector shapes you wish to use in Away3D.
2. Create an instance of the Swf class, setting the libraryClips property to an array of symbol names you wish to convert to 3D objects.
3. Call the parseGeometry() method on the created instance of Swf class, passing the global property root.loaderInfo.bytes as an argument. This ByteArray object contains the raw bytes of the running SWF file and can be parsed at runtime as easily as an externally loaded SWF file.

This approach is certainly handy in Flash Professional if you want to avoid setting up the listeners and handlers required for asynchronous file loading. However, what if you want to avoid this extra work in Flash Builder, FDT, and FlashDevelop? Using the familiar [Embed] meta tag, we can package an external assets SWF file as raw bytes inside the code of the project class. At runtime, the data can be passed to the parseGeometry() method of the Swf class in exactly the same manner as the root.loaderInfo.bytes property used in the workflow for Flash Professional.

Let's take a look at an example that uses the [Embed] meta tag approach by extending the Chapter06SampleBase class with the following document class definition:

```
package flash3dbook.ch06
{
  import away3d.containers.*;
  import away3d.core.base.*;
  import away3d.loaders.*;

  import flash.events.*;
  import flash.utils.*;

  [SWF(width="800", height="600")]
  public class ImportingVectorShapes extends Chapter06SampleBase
  {
    [Embed(source='../../../assets/ch06/snowman.swf',
      mimeType="application/octet-stream")]
    private var SnowmanSwf : Class;

    private var _mesh : Mesh;

    override protected function _createScene() : void
    {
    }
  }
}
```

Here, we embed a previously created SWF file called `snowman.swf`, which can be found in the chapter resource files downloaded from `www.friendsofed.com`. The `mimeType="application/octet-stream"` line in the `[Embed]` meta tag instructs the compiler to embed the file as binary data, which is then represented at runtime as a `ByteArray` class variable called `SnowmanSwf`.

The chapter resources also include the FLA file used to create `snowman.swf`. Opening this file, we see it contains a symbol that has been exported for ActionScript with the linkage class name `Snowman`. If you are not using the sample files from `www.friendsofed.com`, you can create your own `snowman.swf` file by following steps 1 to 3 in the workflow for importing vector shapes from an external SWF file described earlier in this section.

To parse the vector shapes contained within our embedded SWF file in the `ImportingVectorShapes` example, we add the following code to the empty `_createScene()` method:

```
var snowmanSwf: ByteArray = new SnowmanSwf() as ByteArray;

var swf : Swf = new Swf();
swf.libraryClips = ["Snowman"];

var snowman : ObjectContainer3D = swf.parseGeometry(snowmanSwf) as
ObjectContainer3D;
_view.scene.addChild(snowman);
```

The first line of code creates an instance of the embedded SWF asset as a `ByteArray` object. The next two lines create a new `Swf` parser object and set its `libraryClips` property to an array containing the name of our library symbol we want to convert. Finally, we use the `parseGeometry()` method of the `Swf` parser object to return the 3D container that is the parent of our parsed library symbol (now represented as a 3D mesh) and add it to the scene.

Compiling the `ImportingVectorShapes` example displays the output shown in Figure 6-6. Despite the `Snowman` symbol containing many curved edges, the rendering of the 3D mesh representation is perfectly smooth thanks to the use of Away3D's vector drawing commands inside the `Swf` parser object to reconstruct the shape.

The parsing of the vector data from a binary SWF file is performed by a third-party open source library called SwfVector, developed by Guojian Wu. The code is distributed along with the Away3D library and can be found in the `wumedia` package of the source files.

To achieve high-quality results, keep in mind the following guidelines when creating vector shapes in the Flash IDE that are to be imported to Away3D in this manner.

- Avoid grouped shapes, movie clips, and text field elements in your library symbol. If you are importing graphics from Illustrator, this can be achieved by repeatedly using the `Break Apart` feature, found in the `Modify` menu of Flash Professional, until all graphics are represented as raw shapes.
- Stroke styles are not yet supported in the `Swf` parser object. If your symbols contain strokes, make sure you convert them to fills using `Convert lines to fills` from the `Shape` submenu of `Modify`.
- Keep in mind that Away3D groups shapes into faces according to the layers in a symbol. If you want all your shapes in the same face, arrange them in the same layer. Fewer faces consume less processing time when rendering, but potential sorting problems can occur in Away3D when the Z-depth calculations from a complex group of shapes produces ambiguous sorting values for a single face.

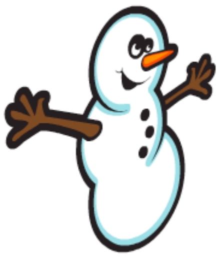


Figure 6-6. The Snowman symbol used in the ImportingVectorShapes example, converted to 3D from the vector shape data contained in the snowman.swf file

Animating imported vector shapes

When importing SWF files in this manner, it is possible to apply any number of 3D transformation techniques on the resulting vector shapes once our 2D symbols have been converted into 3D data. In the previous example, a single mesh is created from the `Snowman` symbol, which means that all the vertices used in the 3D vector shapes in Figure 6-6 are contained within the `vertices` property of the snowman mesh. Let's extend our `ImportingVectorShapes` example to animate these vertices along the Z axis. To begin with, we add the following code to the end of the `_createScene()` method:

```
_mesh = snowman.children[0] as Mesh;
```

This assigns the global variable `_mesh` to the mesh object of the converted `Snowman` symbol, allowing us easy access to the `vertices` array for updating vertex positions. Next, we add the following code to the end of the `ImportingVectorShapes` class definition:

```
override protected function _onEnterFrame(ev : Event) : void
{
  super._onEnterFrame(ev);
  for each (var vertex : Vertex in _mesh.vertices)
    vertex.z = 50*Math.sin(vertex.x/50 + getTimer()/200);
}
```

Here, we loop through all vertices in the `_mesh` object, adjusting their `z` values to update over time creating movement along the Z axis of the mesh.

Recompiling the `ImportingVectorShapes` example displays an output similar to the one shown in Figure 6-7. The snowman mesh animates with a horizontal rippling effect that moves across the object as if it were a flag blowing in the wind. This type of motion benefits from the 3D nature of the symbol shapes after conversion and is a good example of how irregular shapes in Away3D can be used to produce smooth 3D vector effects.

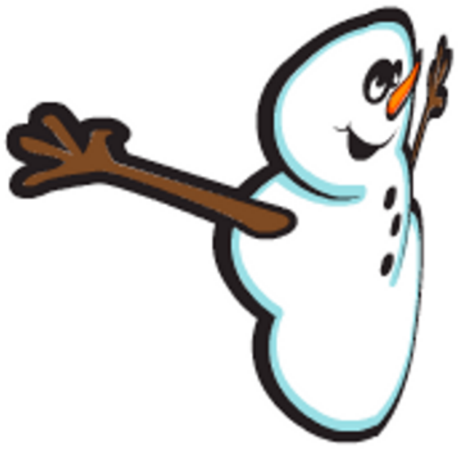


Figure 6-7. The imported Snowman symbol animated in 3D by offsetting the vertices of the resulting mesh object along the Z axis

Importing 3D Text

When dealing with text in Flash, the characters in a font are drawn using the same vector techniques discussed in the previous sections of this chapter. The vector shape of a single character is known as a **glyph**, and a text field defines a collection of glyphs that are drawn to screen using a familiar set of Bézier curve and line definitions contained within each glyph object. When a font is embedded in an FLA in Flash Professional, the SWF format stores its glyph data in a similar way to the vector shape data of library symbols. This can be extracted from the SWF file using a similar approach to the extraction of vector shapes in the previous section, converting 2D glyph data to 3D shape data and allowing the creation of 3D text fields in Away3D.

Extracting vector data from a font

In the Flash Player, the main difference between the process of rendering vector symbols and vector text is in the method used for assembling the graphical output. Symbols are rendered by creating instances that have to be manually managed in your display list, while text fields define a string of characters that are replaced by glyph data when drawn to screen. The same differentiation exists in Away3D, but in the case of a 3D text field, we first have to import the font to be used and convert all glyphs to 3D data. Let's create a new example to demonstrate this process by extending the `Chapter06SampleBase` class with the following class definition:

```
package flash3dbook.ch06
{
  import away3d.primitives.*;
  import wumedia.vector.*;

  import flash.utils.*;

  [SWF(width="800", height="600")]
  public class UsingTextField3D extends Chapter06SampleBase
  {
    [Embed(source="../../../assets/ch06/verdana.swf",
      mimeType="application/octet-stream")]
    private var VerdanaSwf : Class;

    protected override function _createScene() : void
    {
    }
  }
}
```

Here, we embed an SWF file called `verdana.swf`, which like our previous example can be found in the chapter resource files download from `www.friendsofed.com`. The FLA used to create the `verdana.swf` file contains a single text field on the stage with its font set to Verdana and its embedding options set to include the `Basic Latin` glyphs set. Once the resulting SWF has been embedded as binary data in the `VerdanaSwf` class, its font data can be parsed by adding the following code to the empty `_createScene()` method:

```
var verdanaSwf : ByteArray = new VerdanaSwf() as ByteArray;
VectorText.extractFont(verdanaSwf);
```

As you saw in the previous `ImportingVectorShapes` example, our embedded SWF asset is first instantiated as a `ByteArray` object. To process the font glyphs inside, we use the `VectorText` class found in the `wumedia.vector` package, passing our byte array as an argument of the static method `extractFont()`. This creates an internal lookup table for any embedded fonts found in the SWF file, producing the 3D data required for re-creating each glyph of each font and storing it ready for subsequent use by the `Textfield3D` class.

The `Textfield3D` class is found in the `away3d.primitives` package and behaves in a similar manner to any other Away3D primitive. It has a number of configuration property getters and setters that work just like regular primitive properties, only in this case, a redraw is performed by recreating the faces inside the primitive using the parsed font data found in the VectorText class to represent the string of glyph definitions found in the text property. For this reason, one required argument is specified in the constructor of the Textfield3D class to define the name string of the font to be used. If the specified font cannot be found inside the lookup table of the VectorText class, an error will be thrown. For this reason it is important that any fonts being used by Textfield3D objects are parsed using the VectorText class at the start of an Away3D application.

If you are uncertain what name string is used internally in an SWF file for a particular font, embed it in a text field on stage in a dummy FLA in Flash Professional, and trace the defaultTextFormat.font property of the TextField instance, or select 'Copy Font Name for ActionScript' from the Command menu, and paste it into your code..

To create a Textfield3D object in our UsingTextField3D example, add the following code to the end of the _createScene() method:

```
var tf3D : TextField3D = new TextField3D('Verdana');
tf3D.text = 'Vector text in 3D!';
tf3D.size = 100;
tf3D.leading = 20;
tf3D.width = 600;
tf3D.x = -300;
tf3D.y = 150;
_view.scene.addChild(tf3D);
```

Here, we create a new instance of the Textfield3D class, set up to use the Verdana font parsed by the VectorText class in the previous code. We then configure our 3D text field using a number of properties. Some of these (such as text, width, x, and y) will be familiar to anyone using the native TextField class in Flash and apply to the TextField3D class in exactly the same way. The remaining configuration properties are similar to the properties set on a native TextFormat object, controlling the formatting applied to the text contained within. These properties include the following:

- size: This Number value defines the size of the font. Because the notion of a pixel holds no real meaning in 3D, the number represents the size in local units. The default value is 20.
- leading: This Number value defines the vertical space, in local units, added between each horizontal line of text. The default value is 20.
- letterSpacing: This Number value defines the horizontal space, in local units, added between all characters in the text field. The default value is 0.

The final step in the preceding code is to add the TextField3D instance to the scene, which is done using the standard addChild() method. Compiling the code will display an output similar to the one shown in Figure 6-8.

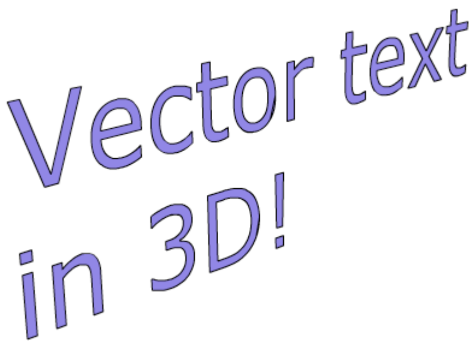


Figure 6-8. 3D text created in the UsingTextField3D example using the TextField3D class

Extruding text

One way we can enhance our new 3D text format is to apply an extrusion along the Z axis, creating characters that appear solid, as if they were carved from stone. This is done with a custom extrusion class found in the `away3d.extrusions` package called `TextExtrusion`. We can test this class by creating a new example from the following document class definition:

```
package flash3dbook.ch06
{
  import away3d.core.base.*;
  import away3d.materials.*;
  import away3d.extrusions.*;
  import away3d.primitives.*;
  import wumedia.vector.*;

  import flash.utils.*;

  [SWF(width="800", height="600")]
  public class ExtrudingTextField3D extends Chapter06SampleBase
  {
    [Embed(source="../../../assets/ch06/verdana.swf",
      mimeType="application/octet-stream")]
    private var VerdanaSwf : Class;

    protected override function _createScene() : void
    {
    }
  }
}
```

We start out with code identical to the previous example and add the following similar code setting up our 3D text field to the empty `_createScene()` method:

```
var verdanaSwf : ByteArray = new VerdanaSwf() as ByteArray;
```

```
VectorText.extractFont(verdanaSwf);

var tf3D : TextField3D = new TextField3D('Verdana');
tf3D.text = 'Extruded vector text';
tf3D.size = 100;
tf3D.leading = 20;
tf3D.width = 600;
tf3D.x = -300;
tf3D.y = 150;

_view.scene.addChild(tf3D);
```

Now, we can apply an extrusion to our created `Textfield3D` instance by adding the following code to the end of the `_createScene()` method. The extra geometry for the extrusion is created inside its own mesh object that has to be added to the scene separately to be visible in the view:

```
var extrusion : TextExtrusion = new TextExtrusion(tf3D);
extrusion.bothsides = true;
_view.scene.addChild(extrusion);
```

The material used for the `TextExtrusion` object is inherited from the originating 3D text field passed in the class constructor, although it can also be set using the `material` property of the extrusion mesh. The `bothsides` property is set to `true` in order to ensure no elements of the extruded geometry are removed with back-face culling. This is an unfortunate necessity with imported vector text, as the exterior winding values of font glyphs have no convention for clockwise versus counterclockwise ordering. Compiling the code will display the output shown in Figure 6-9.

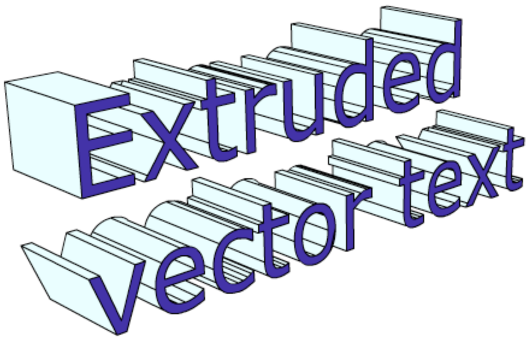


Figure 6-9. Extruded text in the ExtrudingTextField3D example using the TextExtrusion class

Warping text along a path

A final enhancement we will look at for text involves warping a 3D text field along the axis of a spline curve, using the `PathAlignModifier` class found in the `away3d.modifers` package. Technically, this modifier can be applied to any mesh object, but in this case the effect produced is similar to the “text on a path” feature that exists in many graphics applications.

A **spline** in 3D is a collection of Bézier curves that join up to form a continuous irregular curve through space. In Away3D, a spline is defined using the `Path` class found in the `away3d.core.geom` package. Combined with this class, the `PathAlignModifier` class can produce many different warping effects such as text wrapped around a circle or twisted in a spiral.

To demonstrate how the `PathAlignModifier` class works, let's create an example that warps a text field along a rollercoaster-like path by extending the `Chapter06SampleBase` class with the following document class definition:

```
package flash3dbook.ch06
{
  import away3d.core.base.*;
  import away3d.core.geom.*;
  import away3d.core.math.*;
  import away3d.modifiers.*;
  import away3d.primitives.*;

  import flash.events.*;
  import flash.utils.*;

  import wumedia.vector.*;

  [SWF(width="800", height="600")]
  public class WarpingTextField3D extends Chapter06SampleBase
  {
    [Embed(source="../../../assets/ch06/verdana.swf",
      mimeType="application/octet-stream")]
    private var VerdanaSwf : Class;

    private var _pathAlignModifier : PathAlignModifer;
    private var _maxOffset : Number;
    private var _offset : Number = 0;
    private var _speed : int = 10;

    protected override function _createScene() : void
    {
    }
  }
}
```

As with the previous two examples, we start by embedding our font SWF file in the global class property `VerdanaSwf` and add a few global variables to be used in subsequent interactions. We then create our 3D text field by adding some familiar-looking code to the `_createScene()` method:

```
var verdanaSwf : ByteArray = new VerdanaSwf() as ByteArray;
VectorText.extractFont(verdanaSwf);

var tf3D : TextField3D = new TextField3D('Verdana');
tf3D.text = 'Text along a path';
```

```
tf3D.size = 355;
tf3D.width = 355;
tf3D.leading = 20;
tf3D.x = -450;
tf3D.y = 50;

_view.scene.addChild(tf3D);
```

Next, we create a path onto which the text field can be aligned. This is done using an array of Number3D objects representing the control points along a spline curve. We use this form of curve data to define a **continuous curve** (that is, one without breaks or kinks) in our `Path` class definition to provide a smooth path for our text object to follow. To create the path object, add the following lines of code to the end of the `_createScene()` method:

```
var points : Array = new Array();

for (var i : int=0; i<10; i++)
    points.push(new Number3D(i*100, i%2? 50 : -25, 0));

var path : Path = new Path();
path.continuousCurve(points);
```

Here, an array of control points is created using a `for` loop and passed to a new instance of the `Path` object via the `continuousCurve()` method, creating our spline data. The `Number3D` objects created inside the `points` array have smoothly incrementing x coordinates, while the y coordinate alternates between 50 and –25. When passed to the `Path` object, this arrangement creates a smooth undulating curve orientated along the X axis, which is perfect for our purposes.

To visualize the curve created inside the `Path` object, we create an instance of the `Segment` class and convert the spline data into drawing commands by adding the following code to the end of the `_createScene()` method:

```
var segment : Segment = new Mesh();
segment.drawPath(path);

var mesh : Mesh = new Mesh();
mesh.x = tf3D.x;
mesh.y = tf3D.y;
mesh.addSegment(segment);
_view.scene.addChild(mesh);
```

The conversion from `Path` object to `Segment` object is handled using the `drawPath()` method of our newly created `Segment` instance. To attach this to the scene, we create a new `Mesh` object and adjust its position to match that of the `Textfield3D` object, and then add the segment to the mesh and the mesh to the scene.

Completing the example, we add the following code to the end of the `_createScene()` method, creating an instance of the `PathAlignModifier` class with our `Textfield3D` object and `Path` object passed as arguments in the class constructor:

```
_pathAlignModifier = new PathAlignModifier(tf3D, path);
_pathAlignModifier.execute();
```

After the PathAlignModifier class has been instantiated, the modifier is applied to the mesh object specified in the constructor by calling the execute() method. Compile the code to see the output displayed in Figure 6-10.

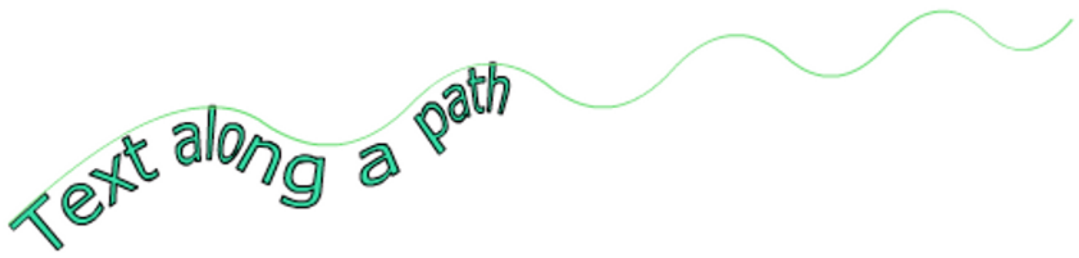


Figure 6-10. Text transformed along a path in the WarpingTextField3D example, using the TextField3D and PathAlignModifier classes

The WarpingTextField3D example applies a static effect to our Textfield3D object, but it is also possible to produce an animated scrolling effect by calling the execute() method of the PathAlignModifier class on every frame while adjusting the offset property. This property defines a Number3D value that represents the vector offset applied to the mesh object before it is transformed by the modifier. Incrementing the x property of offset will appear to scroll the text along the axis of our path. To test this in the WarpingTextField3D example, we first define the global variable _maxOffset by adding the following code to the end of the _createScene() method:

```
_maxOffset = _pathAlignModifier.pathLength - tf3D.width;
```

This variable determines the maximum offset that can be applied to the PathAlignModifier object before the 3D text field begins to moves outside the boundary of our defined spline curve. We can use this value to contain the text scrolling effect to the confines of the Path object, keeping track of the offset value and reversing the direction of scroll when we reach a boundary value. This motion is applied by adding the following lines of code to the end of the WarpingTextField3D class definition:

```
override protected function _onEnterFrame(ev : Event) : void
{
  super._onEnterFrame(ev);

  if (_offset + _speed > _maxOffset || _offset + _speed < 0)
      _speed *= -1;

    _offset += _speed;
    _pathAlignModifier.offset.x = _offset;
    _pathAlignModifier.execute();
}
```

Here, we begin by running a check to determine whether our _offset value is within the limits defined by the boundaries of the Path object, with 0 taking the place of our minimum allowed offset value. The global variable _speed holds the incrementing value of _offset and has its sign reversed if a boundary is detected.

After our boundary check, we add the incrementing value of the `_speed` property to the `_offset` property and update the x value of the `offset` property on the `PathAlignModifier` object. To recalculate the results of the modifier with our property update, we call the `execute()` method, updating the transformation applied to the vertices in the `Textfield3D` object.

Recompiling the `WarpingTextField3D` example will display the 3D text field scrolling along our spline curve, with our boundary conditions causing it to ping-pong between maximum and minimum positions.

By default, the `PathAlignModifier` class uses specially formulated algorithms when calculating the offset along its length. This is because Bézier curves can be a little devious when it comes to calculating distances along a curve, as standard step calculations do not result in a constant step length. This holds special significance when it comes to animations, as these differences result in variable speed calculations that can be a lot more noticeable.

Arc-length parameterization is an approach that attempts to correct these inconsistencies in distance calculations, creating a lookup table of fractional steps that traverse the same distance along all sections of a Bézier curve. While this is necessary to produce animations with constant speed, it means a lot of extra work for the CPU every time `execute()` is invoked on the `PathAlignModifier` object. The accuracy of arc-length parameterization can be varied to produce high levels of accuracy with large lookup tables and a high amount of processing or low levels of accuracy with smaller lookup tables and a small amount of processing. To adjust the balance between accuracy and speed, the `arcLengthPrecision` property of the `PathAlignModifier` class can be set to whatever value you require. Smaller values result in higher precision but lower performance, with the default set at 0.01. As a test, you can adjusting the `arcLengthPrecision` property in the `WarpingTextField3D` example by adding the following line of code to the end of the `_onEnterFrame()` method:

```
_pathAlignModifier.arcLengthPrecision = 1/(1 + mouseY/3);
```

Recompiling the example allows you to control the level of arc-length precision via the Y coordinate of the mouse cursor, with a high precision value set when the cursor is near the top of the screen, and a low precision value set when the cursor is near the bottom of the screen.

Knowing the limitations of vector graphics in Away3D

Generally, there is a fine balance between practicality and possibility when using vector graphics in Away3D. Creating irregular faces or segments consisting of more than three vertices will often help with overall rendering speeds when directly compared to triangle tessellating approaches, and produce better looking results when dealing with curves. However, many calculations made under the hood often don’t work quite as well in these circumstances and can frequently lead to rendering artifacts such as Z-sorting issues. The following list covers the main pitfalls to be aware of when creating or importing vector graphics in the ways we have covered in this chapter:

- Irregular vector shapes can only have basic material types applied to their surfaces such as `ColorMaterial`, `WireColorMaterial`, `WireframeMaterial`, and `ShadingColorMaterial`.
- `RectangleClipping` is the only reliable clipping option when a scene contains irregular vector shapes.
- Results can vary when importing font data and heavily depend on the skill of the font designer to minimize the use of extraneous outlines.

- The `Textfield3D` object will often require its `bothsides` property set to `true` because of arbitrary winding directions used by individual character glyphs. This also causes problems for face normal calculations, occasionally leading to shading complications when using shading materials.

Overall, Away3D's vector graphics abilities provide a powerful set of features that can be used to enhance any elements in a scene that benefit from irregular shape definitions drawn with smooth lines and curves.

Summary

This chapter has investigated how smooth vector shapes and text can be rendered in Away3D. We have seen how vector graphics are generated from scratch using the Away3D drawing commands, as well as how vector shapes and font data can be loaded from external sources. We have also had a look at the different options for manipulating 3D vector graphics once they are created.

The following list is a short summary of the key topics you should take with you from this chapter:

- Vector graphics are drawn using an instance of the `Face` or `Segment` class and are added to a `Mesh` instance attached to the scene in order to be rendered in the view.
    - The `moveTo()` method available on faces or segments moves the active drawing position to a given point in 3D space.
    - The `lineTo()` method available on faces or segments draws a straight line from the active drawing position to a given point in 3D space.
    - The `curveTo()` method available on faces or segments draws a quadratic Bézier curve using the active drawing position and a given control point and anchor point in 3D space.
- Vector shapes drawn in Flash Professional can be imported by parsing the raw bytes of an SWF file with the help of the `Swf` class located in the `away3d.loaders` package. This data is either loaded at runtime using an externally created SWF asset file with the `Loader3D` class or embedded in the application SWF as binary data and instantiated as a `ByteArray` object, which is then parsed by the `Swf` class. The application SWF file itself can be specified as the `ByteArray` object to be parsed by using the `root.loaderInfo.bytes` property of the Flash movie.
    - Vector shapes can be easily animated in Away3D by updating the position of vertices in the `vertices` array of the containing `Mesh` object.
- Text can be displayed as vector graphics in Away3D by using the `TextField3D` class. The required font data is extracted from the raw bytes of an SWF file using the `VectorText` class located in the `wumedia.vector` package.
    - A 3D text field can have a linear extrusion effect applied to its geometry with the help of the `TextExtrusion` class located in the `away3d.extrusions` package.

- A 3D text field can be transformed to follow the curves in a `Path` object by using the `PathAlignModifier` class located in the `away3d.modifiers` package.

In the next chapter, we will look at the various ways Away3D allows us to generate our own geometry at runtime, from the basic construction methods used in primitives to the advanced structures possible with the extrusion and modifier tools.

Chapter 7

Procedural 3D Content

With the built-in primitive classes in Away3D, simple geometric 3D shapes can be created on the fly without the need for importing an externally created model. However, primitives are quite limited in their configuration, with only a handful of properties available to adjust their sizes and shapes. For more complex internally generated geometry, we need to consider using some of the more advanced features of the Away3D engine.

The advantage of generating 3D objects inside your application extends beyond simply avoiding the need to load a model. Imagine a scenario where a 3D object is created from user interaction or where you require many unique objects with similar characteristics. It would be impractical to store a model for each of these results, as well as being a large overhead for the size of your SWF file.

This chapter will take a look at the process behind creating your own 3D geometry within Away3D from scratch. We will also investigate some of the more advanced geometric tools available for creating a wide variety of shapes.

Preparing the chapter base class

To provide a basic viewing mechanism for the examples created in this chapter, let's set up a chapter base class with a simple hover camera that can be used as our starting point for subsequent class files.

```
package flash3dbook.ch07
{
  import away3d.cameras.*;
  import away3d.containers.*;

  import flash.display.*;
  import flash.events.*;
  public class Chapter07SampleBase extends Sprite
  {
    protected var _view : View3D;
    protected var _camera : HoverCamera3D;

    public function Chapter07SampleBase()
    {
      _createView();
      _createScene();
    }

    protected function _createView() : void
    {
      _camera = new HoverCamera3D();
      _camera.distance = 1000;
      _camera.tiltAngle = 10;
      _camera.panAngle = 180;
      _view = new View3D();
      _view.x = 400;
      _view.y = 300;
      _view.camera = _camera;

      addChild(_view);
      addEventListener(Event.ENTER_FRAME, _onEnterFrame);
    }

    protected function _createScene() : void
    {
    }

    protected function _onEnterFrame(ev : Event) : void
    {
      _camera.panAngle -= (stage.mouseX - stage.stageWidth/2) / 100;
      _camera.hover();

      _view.render();
    }
  }
}
```

In the preceding code, the `_createView()` method instantiates our view and camera objects, which are held in the global class properties `_view` and `_camera` respectively. The camera uses an instance of the `HoverCamera3D` class to offer easy navigation around the scene. The view is rendered using the `_onEnterFrame()` method, which is set up as an `ENTER_FRAME` event handler at the end of the `_createView()` method. The `_createScene` method is written as an empty stub so that it can be overridden in our following example classes to add custom content to the scene.

Building a pyramid primitive

To demonstrate building a 3D object from scratch, we will focus on creating a simple geometric shape that doesn't currently exist in the `away3d.primitives` package. One such shape is a square-based pyramid. It is theoretically possible to build a pyramid shape from the cone primitive, but let's ignore that for the sake of this example.

A good way to manage our custom code is to create a new class that will represent the 3D object in a scene, in the same manner as existing primitive classes. Attributes of the object such as width and height can then be tweaked using custom property getters and setters that update the contained geometry on the fly, allowing us to adjust the overall look of the object even after instantiation.

Starting with AbstractPrimitive

Looking at the source code for the existing primitives in Away3D, we see that all classes extend a generic `AbstractPrimitive` class, found in the `away3d.primitives` package. The `AbstractPrimitive` class contains useful functionality for anyone wishing to create a custom geometric object, making it easier to build and update the vertices and faces of the underlying mesh.

If we are to extend the `AbstractPrimitive` class for our custom pyramid primitive, we will need to follow a few conventions in order to take advantage of the existing functionality. Our pyramid class must have the following:

- A constructor that accepts an `init` object
- A `buildPrimitive()` method that overrides the existing method in `AbstractPrimitive`, executing the code necessary to create the vertices and faces for the 3D mesh of the new primitive object
- Getter and setter methods for the externally facing properties that define the geometry of the object, with all setters invalidating the existing geometry and flagging it for a rebuild

We will go through each one of these requirements in more detail shortly, but first, let's create the base code of our primitive by creating a class called `Pyramid` that extends `AbstractPrimitive` and is located in the `flash3dbook.ch07.primitives` package.

```
package flash3dbook.ch07.primitives
{
  import away3d.arcane;
  import away3d.primitives.*;
  import away3d.core.base.*;

  use namespace arcane;

  public class Pyramid extends AbstractPrimitive
  {
    private var _height : Number;
    private var _width : Number;
    private var _depth : Number;

    public function Pyramid(init : Object = null)
    {
      super(init);

      buildPrimitive();
    }
  }
}
```

Here, we have created a number of internal configuration variables and a constructor that accepts an `init` object. Now, let's take a closer look at the code required for the constructor method.

> *In the preceding code, an ActionScript namespace called `arcane` is imported at the top of the class definition. This namespace is defined by Away3D and is somewhat jokingly called "arcane" because it is applied to elements of a class not intended for external access in normal operation.*
>
> *Nonpublic functions and properties exist all over the Away3D library, but the ones given the namespace `arcane` still need to be accessed between different classes and packages from within the library. The standard Flash namespaces such as `private` or `protected` are simply not suitable for this purpose, so the custom namespace `arcane` is created.*
>
> *When extending the engine, it is quite possible you will need to access Away3D `arcane` methods if you want your extensions to be fully integrated into the existing framework. In this case, we will use the `arcane` namespace to access the custom methods of the `AbstractPrimitive` class that are intended for use inside other geometric primitive classes.*

Setting up the constructor

As is common when extending a class, the first thing to do in the constructor is call the superclass constructor. This ensures any initializing code is triggered and in this case, passes our `init` object from the constructor argument to the parent class to ensure it is received and processed by any superclass properties.

Because Away3D allows initializing properties of many new object instances to be passed in an `init` argument as name/value pairs, a utility class called `Init` exists to help parse this data. The initial `init` argument is wrapped inside an instance of the `Init` class that can then offer typed object extraction of the various properties held within. The `Init` instance is stored in the protected `ini` variable, so once the superclass constructor is called, all further `init` values are extracted through `ini`. For example, the following line uses the `ini` variable to retrieve the numerical value of a variable called `width`:

```
width = ini.getNumber("width", 100, {min:1});
```

This will look up the property called `width` in the `init` object and cast any returned value as a `Number` object type. If no value is found, or if the returned value cannot be cast to a `Number`, the default value of

100 is used. In this particular piece of code, we also use a further configuration argument in the getNumber() method to specify that the minimum value of the returned number can be no less than 1. If circumstances require it, you may also set a maximum value in the same way:

```
width = ini.getNumber("width", 100, {min:1, max: 1000});
```

Using the Init class will help you ensure type safety on any init argument properties. Similar methods exist for all basic data types such as String, int, and Boolean, as well as frequently used Away3D data types such as BitmapMaterial and Number3D.

Let's set up our constructor to parse the init argument for the property values we will be using in our pyramid primitive. Add the following lines of code to the end of the Pyramid class constructor:

```
_height = ini.getNumber('height', 180, { min: 1 });
_width = ini.getNumber('width', 180, { min: 1 });
_depth = ini.getNumber('depth', 180, { min: 1 });
```

Adding public properties

According to Away3D convention, all configuration values of a 3D object should be adjustable via public properties on the class instance. This allows the modification of an object's mesh structure after instantiation, avoiding the need for re-creating an object simply to reconstruct it. The public property names should correspond to those used in the init object, with any adjustment propagating to the mesh the next time the 3D object is rendered.

The common setup for the configuring properties of Away3D primitives has a getter/setter pair that flags the object as modified (or dirty) when updated. This will notify the view on the next render call to rebuild the geometry of the mesh. Rebuilding the mesh of a 3D object is achieved by invoking the buildPrimitive method, which contains the custom code for constructing the mesh from the configuring properties. The code required here for our pyramid primitive will be created shortly.

First, let's take a look at the getters and setters required for the properties of the pyramid primitive. We begin with the width property by adding the following code to the Pyramid class definition:

```
public function get width() : Number
{
  return _width;
}

public function set width(val : Number) : void
{
  if (_width == val)
    return;

  _width = val;
  _primitiveDirty = true;
}
```

Most of the time, a getter of a primitive property will do nothing more than return its internal value, which, in this case, is stored in our global variable _width. There is more going on in the setter, though. First, the setting value is checked to ensure the property is not just being set to its current value. If the value is the same, nothing more is done because the geometry doesn't require updating. If the value is different, the new value is stored in the internally defined global variable `_width`, and the primitive is flagged for an update on the next `render()` call by setting the `_primitiveDirty` variable (inherited from `AbstractPrimitive`) to `true`.

Using the same arrangement we have written for the `width` property, we create getters and setters for the remaining configuration properties of the `Pyramid` class by adding the following code to the class definition:

```
public function get height() : Number
{
  return _height;
}
public function set height(val : Number) : void
{
  if (_height == val)
    return;

  _height = val;
  _primitiveDirty = true;
}


public function get depth() : Number
{
  return _depth;
}

public function set depth(val : Number) : void
{
  if (_depth == val)
    return;

  _depth = val;
  _primitiveDirty = true;
}
```

Building the Pyramid mesh

Now, we can turn our attention to the creation of the custom `buildPrimitive` method that will be triggered when our pyramid primitive requires rebuilding. Let's begin by considering the shape we are trying to create. All 3D mesh objects are created from a collection of vertices that are grouped into face definitions, more often than not existing as triangles made out of three vertex points each. As shown in Figure 7-1, a pyramid requires only five corners to create its geometry, so these will be our five vertex positions. We need to define six faces: one for each triangular side and two to create the rectangular base of the pyramid.

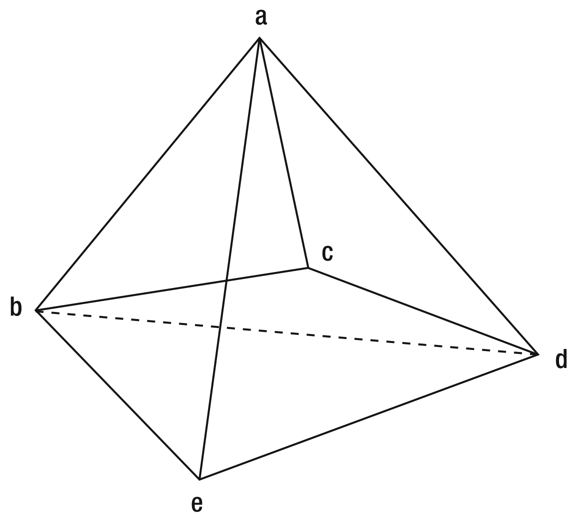


Figure 7-1. The vertices and faces of our pyramid primitive

The order in which vertices are instantiated is not important, as long as we keep track of the vertex objects created. However, the order in which vertices are bound to faces *is* important, because of the automated process of **back-face culling** (described in more detail in Chapter 4). With back-face culling active, the visibility of face is determined on the order of its observed vertices, with a counterclockwise order being visible and a clockwise order being invisible. Therefore, when we create our faces, we order the vertices so that each face when viewed from its outer side will appear with vertices arranged in a counterclockwise order, ensuring the outside of the pyramid is always visible. Because the pyramid is a closed geometric shape (i.e., its faces form a completely enclosed interior), faces observed as invisible will always be covered by faces observed as visible, giving the appearance of a solid object.

Let's go ahead and create our custom `buildPrimitive` method by adding the following code to the end of the `Pyramid` class definition and overriding the base method from the `AbstractPrimitive` class:

```
protected override function buildPrimitive() : void
{
  super.buildPrimitive();
}
```

We begin by calling the superclass method, which clears any previously generated mesh elements, ready for the new geometry generation. Using Figure 7-1 as a guide, we then create five vertex objects `a` to `e` at the appropriate positions in 3D space by adding the following code to the end of our newly created `buildPrimitive()` method:

```
var a : Vertex = createVertex(0, _height/2, 0);
var b : Vertex = createVertex(-_width/2, -_height/2, _depth/2);
var c : Vertex = createVertex(_width/2, -_height/2, _depth/2);
var d : Vertex = createVertex(_width/2, -_height/2, -_depth/2);
var e : Vertex = createVertex(-_width/2, -_height/2, -_depth/2);
```

Our _width, _height, and _depth properties are used to define the precise locations of all vertices, arranging them in a way that centers the resulting geometry around the local origin of the primitive. Rather than use the Vertex class to create a new vertex, we use the inherited AbstractPrimitive method createVertex(). This assists memory usage by recycling any previously used Vertex instances. Later, you'll see the same technique being used for creating face and UV coordinate objects.

The createVertex() method requires nothing more than three arguments representing the x, y, and z coordinates of the vertex being created. Figure 7-2 offers an alternative view of our pyramid geometry from above, with the vertices using the same labels as in Figure 7-1.

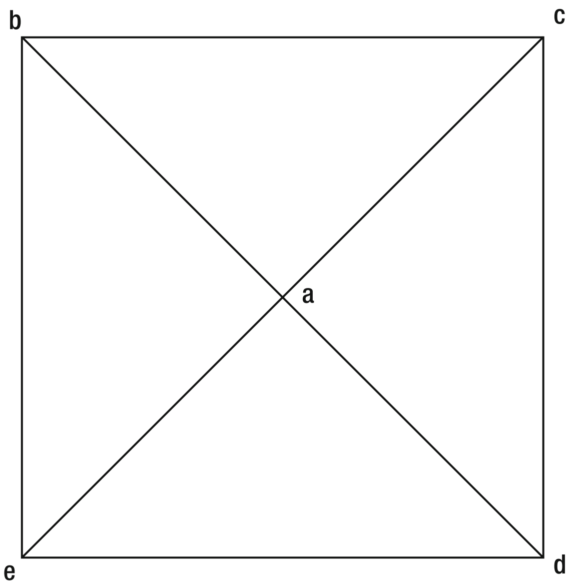


Figure 7-2. The geometry to be constructed by the pyramid primitive, as seen from above

Now, we need to construct the faces that make up the visible surfaces of the pyramid primitive. We do this by adding the following lines to the end of our newly created buildPrimitive method:

```
addFace(createFace(a, c, b));
addFace(createFace(a, d, c));
addFace(createFace(a, e, d));
addFace(createFace(a, b, e));
addFace(createFace(b, d, e));
addFace(createFace(b, c, d));
```

Here, we are using two methods for the creation of each face. The addFace() method inherited from the Away3D base class Mesh adds a Face instance to the mesh geometry, ready for rendering. The createFace() method inherited from the AbstractPrimitive class operates in a similar way to the createVertex() method you've seen previously and requires at least three arguments representing the three Vertex objects to be used for the face. It is here that vertex ordering is taken into account, applying our previously defined rule to ensure back-face culling is performed correctly on the resulting mesh.

At this stage, we have created all the geometry required for our pyramid primitive and can easily test it by extending the `Chapter07SampleBase` class with the following document class definition:

```
package flash3dbook.ch07
{
  import away3d.core.utils.*;
  import away3d.materials.*;

  import flash3dbook.ch07.primitives.*;

  [SWF(width="800", height="600")]
  public class PyramidTest extends Chapter07SampleBase
  {
    public function PyramidTest()
    {
      super();
    }

    protected override function _createScene() : void
    {
      var pyramid : Pyramid = new Pyramid();
      pyramid.width = 200;
      pyramid.depth = 200;
      _view.scene.addChild(pyramid);
    }
  }
}
```

Compiling the code will display a pyramid centered around the scene origin with a base dimension of 200 × 200 and a default height of 100. The texture used on the mesh is a default `WireColorMaterial` object, because we haven't specified any material for the primitive to use in this example. However, this brings to our attention a problem with our current `Pyramid` class definition—we have no way of texturing the pyramid with a `BitmapMaterial` object, because we have defined no UV coordinates in our geometry. To rectify this problem, let's go back to the `buildPrimitive()` method in our `Pyramid` class to add in the necessary UV definitions.

Mapping UV coordinates

Recall from Chapter 4 that UV mapping is a process that defines how a texture material is applied to the different faces in a mesh object. The default primitives in Away3D come prebuilt with UV mapping, so bitmap textures will wrap nicely around the created 3D mesh. Because we are creating our pyramid primitive from scratch, we will have to generate our own UV mapping coordinates if we want to apply any type of bitmap material to the resulting mesh.

Our pyramid primitive has UV mapping applied by creating a series of UV coordinates and matching them with the vertex instances defined in each face. A UV coordinate is a 2D vector that represents the fractional position value of a point on the surface of the texture image. These coordinates describe the texture positions that map to the positions defined by the associated vertices on each face of the mesh.

UV coordinate positions represent a fractional value of the width and height of the texture image between 0 and 1, in order to be independent of texture dimensions. In a bitmap that is 200 pixels wide, a U value of 0.5 translates to a horizontal position 100 pixels to the left, whereas in a larger texture of 500 pixels, the same U value translates to a horizontal position 250 pixels to the left.

The origin of a UV coordinate system (the point at which the UV coordinate value in a texture is (0,0)) is defined as the bottom-left corner of the bitmap image used for the texture data. Similarly, the point at which the UV coordinate of a texture is (1,1) is defined as the top-right corner of the bitmap image. This is unlike normal 2D coordinate systems, where the origin is defined as being in the top-left corner, but is a standard approach for UV mapping. Figure 7-3 depicts the UV mapping we will be applying to our pyramid primitive, with the representations of each face projected into the UV coordinate system of the texture. This form of UV coordinate representation is often referred to as **UV unwrapping**.

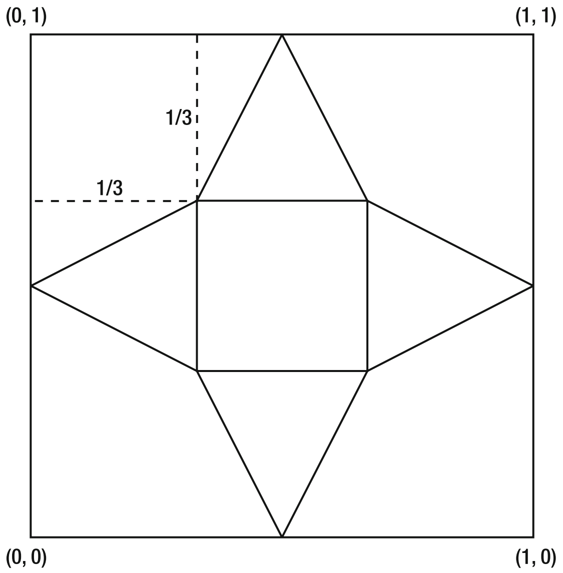


Figure 7-3. The proposed UV mapping of our custom pyramid primitive is projected onto a representation of the texture. Note the origin (0,0) of the UV coordinate system located in the bottom-left corner of the bitmap image.

Going back to our `Pyramid` class definition, let's add UV coordinates to our faces with the help of the UV map shown in Figure 7-3. We'll start by creating the necessary `UV` objects with the following lines of code added inside the `buildPrimitive()` method, directly after creating our five `Vertex` objects:

```
var uva0 : UV = createUV(0.5, 1);
var uva1 : UV  = createUV(1, 0.5);
var uva2 : UV  = createUV(0.5, 0);
var uva3 : UV  = createUV(0, 0.5);
var uvb : UV  = createUV(1/3, 2/3);
var uvc : UV  = createUV(2/3, 2/3);
var uvd : UV  = createUV(2/3, 1/3);
var uve : UV  = createUV(1/3, 1/3);
```

The first thing to note in Figure 7-3 is that the vertex for the top point of the pyramid occurs in four places in the UV map. It is important to understand that UV mapping is applied to each face, not to each vertex, so a single vertex being used multiple times in a map is not uncommon. In the preceding code, we represent the four occurrences of this vertex with the first four `UV` objects: `uva0`, `uva1`, `uva2`, and `uva3`. The remaining four `UV` objects represent the UV coordinates assigned to the four corners of the pyramid base. In this case, each face has the same UV coordinate assigned to the associated vertex object.

To apply these UV coordinates to the relevant `Face` objects, we need to modify the lines that create our faces in the `buildPrimitive()` method. Locate these lines and replace them with the following similar lines of code:

```
addFace(createFace(a, c, b));
addFace(createFace(a, d, c));
addFace(createFace(a, e, d));
addFace(createFace(a, b, e));
addFace(createFace(b, d, e));
addFace(createFace(b, c, d));
```

Here, we have added a few more arguments to each of the `createFace()` methods to pass the created `UV` objects to their correct faces. The order of `UV` objects here is important, as they need to map to the existing `Vertex` objects in the same sequence so that the first UV coordinate is mapped to the position of the first vertex point, the second to the second vertex point, and so on. The `null` argument inserted between the `Vertex` and `UV` arguments for the `createFace()` method is intended for assigning a custom material instance to a face and is not being used in this example.

All the modifications we have made here should leave us with a `buildPrimitive` method containing the following code:

```
protected override function buildPrimitive() : void
{
  super.buildPrimitive();

  var a : Vertex = createVertex(0, _height/2, 0);
  var b : Vertex = createVertex(-_width/2, -_height/2, _depth/2);
  var c : Vertex = createVertex(_width/2, -_height/2, _depth/2);
  var d : Vertex = createVertex(_width/2, -_height/2, -_depth/2);
  var e : Vertex = createVertex(-_width/2, -_height/2, -_depth/2);

  var uva0 : UV = createUV(0.5, 1);
  var uva1 : UV  = createUV(1, 0.5);
  var uva2 : UV  = createUV(0.5, 0);
  var uva3 : UV  = createUV(0, 0.5);
  var uvb : UV  = createUV(1/3, 2/3);
  var uvc : UV  = createUV(2/3, 2/3);
```

```
    var uvd : UV  = createUV(2/3, 1/3);
    var uve : UV  = createUV(1/3, 1/3);

    addFace(createFace(a, c, b, null, uva0, uvc, uvb));
    addFace(createFace(a, d, c, null, uva1, uvd, uvc));
    addFace(createFace(a, e, d, null, uva2, uve, uvd));
    addFace(createFace(a, b, e, null, uva3, uvb, uve));
    addFace(createFace(b, d, e, null, uvb, uvd, uve));
    addFace(createFace(b, c, d, null, uvb, uvc, uvd));
}
```

To test the texture mapping of the modified `Pyramid` class, we can update our previous `PyramidTest` example to apply a `BitmapMaterial` object in place of the default `WireColorMaterial` object. We first need an image for the texture, which we can import by adding the following lines of code to the class definition:

```
    [Embed(source="../../../assets/ch07/pyramidcolor.jpg")]
    private var PyramidImage : Class;
```

Check that the file path in the `[Embed]` tag matches the path to the `pyramidcolor.jpg` image downloaded inside the chapter resource files from `www.friendsofed.com`. To use this for our material, replace the existing code inside the `_createScene()` method with the following:

```
var pyramid : Pyramid = new Pyramid();
pyramid.material = new BitmapMaterial(Cast.bitmap(PyramidImage));
pyramid.width = 200;
pyramid.depth = 200;
_view.scene.addChild(pyramid);
```

Recompiling the `PyramidTest` example should display the output in Figure 7-4, with our pyramid primitive texture printing a different color on each side of the pyramid primitive.

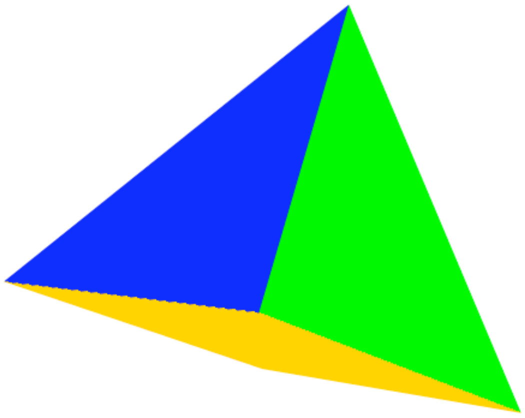


Figure 7-4. The updated pyramid primitive rendered with a bitmap material using the **`pyramidcolor.jpg`** image, resulting in a different color on each side

Using the approach outlined in our PyramidTest example, it is possible to build an infinite variety of custom geometric objects from scratch. If you feel like donating particularly useful creations of your own, it is worth submitting new classes to the Away3D team for consideration as an official addition to the engine.

In the next section, we will look at some of the more advanced classes that already exist in Away3D for generating geometry. The area we will concentrate on includes a group of classes known as the extrusion tools. While these use a set of prebuilt classes rather than the "build from scratch" approach we have investigated here, the tools provide a way of generating highly configurable 3D objects with a greater degree of control compared to the standard primitive classes.

Using the extrusions tools

The process of **extruding** (in 3D terms) generally means the creation of a 3D model from the surface traced by the outline of a 2D shape dragged through space. In Away3D, a set of classes known as the **extrusion tools** enable many different 3D extrusion techniques and can be found inside the away3d.extrusions package. In this section, we will take a look at some of the classes on offer.

Creating a ribbon using the PathExtrusion class

The PathExtrusion class allows us to take an array of points, usually referred to as a **profile**, and extrude its defined shape along a smooth Bézier curve. The profile we use is generally an outline of the cross section of the 3D object we want to create. For example, an extrusion forming a girder object would require a profile in the shape of an "H".

The PathExtrusion class is ideal for creating objects such as roads or ribbons dynamically. Let's take a look at how we would go about creating a ribbon by extending our chapter base class with the following document class definition:

```
package flash3dbook.ch07
{
  import away3d.core.geom.*;
  import away3d.core.math.*;
  import away3d.extrusions.*;

  [SWF(width="800", height="600")]
  public class UsingPathExtrusion extends Chapter07SampleBase
  {
    public function UsingPathExtrusion()
    {
      super();
    }

    protected override function _createScene() : void
    {
    }
  }
}
```

Here, we create no more than an empty `_createScene()` method, ready for some custom code. The first thing we need is an instance of the `Path` class, a basic geometric definition found in the `away3d.core.geom` package, which defines a series of **Bézier curves** in a 3D shape known as a **spline**. For more about Bézier curves and splines, read the section on animating along a path in Chapter 9.

Each curve in a spline consists of three points: a start point, an end point, and a control point that defines curvature. The constructor of the `Path` class accepts an array argument that contains these points as triplets of `Number3D` instances. To create a `Path` object to be used in our path extrusion, add the following code inside the empty `_createScene()` method of the `UsingPathExtrusion` class:

```
var path : Path = new Path([
  // First curve:
  new Number3D(-200, 0, 0),
  new Number3D(-100, 0, 200),
  new Number3D(0, 0, 0),

  // Second curve:
  new Number3D(0, 0, 0),
  new Number3D(100, 0, -200),
  new Number3D(200, 0, 0)
]);
```

Here, we have two curves, each represented by three points. If you look closely, you'll notice that the last point in the first curve is identical to the first point in the second curve. This ensures the resulting spline is continuous, with no breaks between the two curve segments.

Next, we need to define the profile shape we will use in our path extrusion. This is an array of points that trace out the positions to be extruded along the path. For the sake of simplicity, we will define this in our example as two points forming a straight line by adding the following code to the end of the `_createScene()` method:

```
var profile : Array = [
  new Number3D(0, -20, 0),
  new Number3D(0, 20, 0)
];
```

Here, we define a vertical line 40 units long, beginning at a point 20 units above the origin and ending at a point 20 units below the origin. Extruding this profile along our path will create a vertical wall. If we wanted to create a road, we could use a similar profile that was oriented horizontal, for example, along the X axis.

We can now create our path extrusion by combining the previously created `path` and `profile` data objects inside a `PathExtrusion` object. Add the following lines of code to the end of the `_createScene()` method:

```
var extrusion : PathExtrusion = new PathExtrusion(path, profile);
extrusion.subdivision = 10;
extrusion.bothsides = true;
_view.scene.addChild(extrusion);
```

Here, our `path` and `profile` properties are passed to the `PathExtrusion` object as arguments in the constructor. As with any other 3D object, we have the option to set configuration properties after instantiation. In the preceding code, we assign new values to the `bothsides` and `subdivision` properties:

`bothsides` is a general property for all 3D mesh objects that ensures both sides of all faces in a mesh are visible by disabling the default process of back-face culling. The `subdivision` property is specific to the `PathExtrusion` class, and its use is described in the following list, along with the use of several other path extrusion properties:

- `subdivision`: This `Number` value defines how each Bézier curve definition is subdivided in the resulting path extrusion. A curve is approximated by a series of vertices created at regular intervals along the path of each curve. In our example, the two points in the profile shape will be duplicated ten times along each curve, with faces generated between each set of points to create a continuous surface. The default value is `1`.
- `coverAll`: This `Boolean` value defines how UV coordinates are applied to the geometry of the extrusion object. If it's set to `true`, UV values are created to stretch the texture from the beginning to the end of an extrusion. If it's set to `false`, the same geometry repeats the texture across each curve in the path. The default value is `true`.
- `closePath`: This `Boolean` value defines whether an extra curve is automatically created to join the last position in the extrusion path with the first, creating a closed loop. It defaults to `false`, leaving the resulting extrusion geometry open at both ends.

Compiling the code will display the output seen in Figure 7-5. The potential for different shapes using the `PathExtrusion` class is virtually limitless, and we hope this introduction has got you thinking of your own uses for this nifty extrusion object.

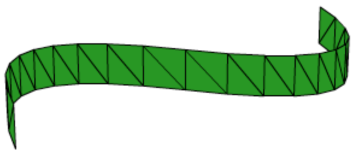


Figure 7-5. A ribbon created in the UsingPathExtrusion example with the PathExtrusion class

Creating a vase with the LatheExtrusion class

In carpentry, the process of lathing involves carving a piece of wood or metal using a machine known as a **lathe.** This cuts a profile into a block of the desired material as it spins, producing an object with axial symmetry (such as a table leg or baseball bat). A 3D object with **axial symmetry** has a single axis around which it can rotate with no changes in profile. This form can be reproduced in Away3D using the `LatheExtrusion` class.

The process used to construct a 3D mesh in this manner requires a profile consisting of a series of points (similar to the profile definition used by the `PathExtrusion` class in our previous example), which is then rotated around an axis to produce a lathed mesh object. Let's look at a simple example using the `LatheExtrusion` class by extending our `Chapter07SampleBase` class with the following document class definition:

```
package flash3dbook.ch07
{
  import away3d.core.math.*;
  import away3d.extrusions.*;

  [SWF(width="800", height="600")]
  public class UsingLatheExtrusion extends Chapter07SampleBase
  {
    public function UsingLatheExtrusion()
    {
      super();
    }

    protected override function _createScene() : void
    {
    }
  }
}
```

In this example, we will create a vase shape from a simple four-point profile of the side of a vase. We begin by adding the following code to the empty `_createScene()` method:

```
var profile : Array = [
  new Number3D(-50, 200, 0),
  new Number3D(-40, 150, 0),
  new Number3D(-60, 120, 0),
  new Number3D(-40, 0, 0)
];
```

Here, we create a vase profile from an array of `Number3D` points defined in the XY plane, starting at a point above the origin and finishing at a point horizontal to the origin. Note that all x coordinates are negative, placing them to the left of the Y axis. It is generally necessary to create a profile for lathe extrusions that has all points positioned to one side of the axis of rotation, to ensure we create a mesh that doesn't intersect with itself. For our profile array defined here, any point positioned to the right of the Y axis would have the potential to come into conflict with points positioned to the left when the profile is rotated to create the resulting geometry. To avoid potential collisions, we keep all profile points on one side of the Y axis, which is the default axis of rotation.

As with the `PathExtrusion` class, we pass our profile array in the constructor of the `LatheExtrusion` object, creating our vase object by adding the following code to the end of the `_createScene()` method:

```
var vase : Lathe = new Lathe(profile);
vase.subdivision = 12;
vase.centerMesh = true;
vase.thickness = 10;
_view.scene.addChild(vase);
```

Once again, we set a few configuration properties on our extrusion object after instantiation. The following descriptions cover the most frequently used properties for the `LatheExtrusion` class.

- `axis`: This `Number3D` object defines the axis around which the profile is rotated as the mesh is created. The default value is `(0, 1, 0)`, representing the Y axis.

- `subdivision`: This `Number` value defines the number of vertices added to the mesh as the profile is revolved around the axis. As with the `PathExtrusion` class, faces are generated between each set of profile points to create a continuous surface. The default value is `2`.
- `centerMesh`: This `Boolean` value indicates whether the mesh created by the lathing operation should be automatically translated so that it's local origin matches the center of the geometry. Without this, the position of the resulting mesh is determined by the absolute positions of the profile points after rotating. The default value is `false`.
- `thickness`: This `Number` value determines if the resulting mesh has a back and a front to the rotated profile. Its default value is 0, causing the lathe to create a single set of faces from the profile points. If we set the value to anything other than 0, the lathe will add thickness to the resulting mesh by creating two surfaces, an inner one and an outer one, separated by a distance defined by the `thickness` property. This is a great feature for our vase, as it allows us to define a general contour for the vase shape as a single line, and then create a hollow object with some thickness to its walls.

Compiling the code should display the image shown in Figure 7-6.

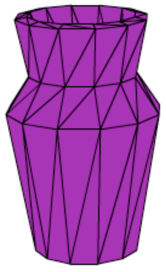


Figure 7-6. A vase created using the LatheExtrusion class in the **`UsingLatheExtrusion`** example

Using mesh modifiers

Mesh modifiers are classes in Away3D that modify existing geometry rather than creating it from scratch. There are many different modifier types available from the `away3d.modifers` package. A general feature of mesh modifiers is their ability to modify the vertices and faces of a mesh (or meshes) to change the appearance or configuration of the resulting mesh object (or objects).

We will be taking a look at one type of mesh modifier in particular, represented by the `HeightMapModifier` class. This allows you to perturb the vertices of any existing mesh object along their normal vectors to create deformations in the mesh surface.

Creating a terrain using the HeightMapModifier

The `HeightMapModifier` class can be used on any existing mesh object or 3D object that extends the `Mesh` class. All modifiers in Away3D operate on the principal that mesh objects need to be explicitly defined for the modifier. It is not possible to apply a modifier to a 3D container object, because a container can contain any number of meshes.

A height map is generally considered to be a bitmap image that contains data relating to the relief of a 3D mesh's surface. The pixel values are interpreted as height values, and the resulting image describes a continuous contour in 3D. This can be applied to a mesh using the UV coordinates on each face to map the height map as a texture map. A position offset is then applied to the vertices of the mesh by taking the average of the offset values for the height map around the UV positions of each vertex point.

Because a height map only defines the offset amount. The direction of offset is calculated for each vertex in the mesh using a vector called the **vertex normal**. This vector is similar to the face normal vector described in Chapter 4 and is calculated automatically for each vertex by taking the weighted average of the face normal vectors of all faces using the vertex.

To demonstrate the effect achieved with the `HeightMapModfier` class, we will use a plane primitive as our input mesh object, as this will produce results that are easy to interpret. Let's begin by extending the `Chapter07SampleBase` class with the following document class definition:

```
package flash3dbook.ch07
{
  import away3d.modifiers.*;
  import away3d.primitives.*;

  import flash.display.*;
  import flash.events.*;
  import flash.ui.*;

  [SWF(width="800", height="600")]
  public class UsingHeightMapModifier extends Chapter07SampleBase
  {
    private var _scale : Number = 0;
    private var _modifier : HeightMapModifier;

    protected override function _createScene() : void
    {
    }
  }
}
```

There are a few more `import` statements seen here compared to previous examples, because we will be adding a bit of keyboard interaction later in the code to control our modifier object and require the native classes for events and keyboard objects. But before we deal with interaction, we need to create a height map to use in our `HeightMapModifier` class. This is generated by adding the following lines of code to the `_createScene()` method:

```
var bmp : BitmapData = new BitmapData(200, 200, false);
bmp.perlinNoise(20, 20, 2, 0, true, true, 7, true);
```

Here, we use Perlin noise to create a 200 × 200 bitmap data image with a smoothly varying grayscale pixel value. To check what it looks like, let's draw the resulting `BitmapData` object to the stage by adding the following lines of code to the end of the `_createScene()` method:

```
graphics.beginBitmapFill(bmp);
graphics.drawRect(0, 0, 200, 200);
```

Compiling the code should display something similar to the output shown in Figure 7-7.

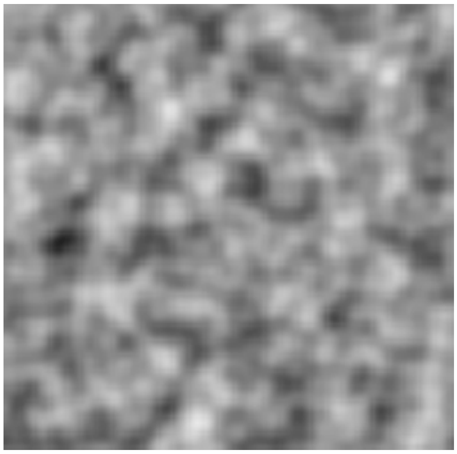


Figure 7-7. The bitmap image used as our height map for the HeightMapModifier class in the UsingHeightMapModifier example, created using a low-resolution Perlin noise filter

Next, we create the plane primitive to which the modifier will be applied. A key point to consider is that the `HeightMapModifer` class can do no more than offset the position of individual vertices in a mesh. To ensure reasonable results, we must subdivide the plane into an appropriate number of faces. We use a plane with 20 subdivisions along both the width and height dimensions, a sufficient density for our purposes, by adding the following code to the end of the `_createScene()` method:

```
var _plane : Plane = new Plane();
_plane.width = 500;
_plane.height = 500;
_plane.segmentsW = 20;
_plane.segmentsH = 20;
_view.scene.addChild(_plane);
```

All that's left to create is the modifier object itself, which requires two arguments in its constructor: the mesh object it will operate on and the `BitmapData` object containing the height map data. To achieve this, add the following code to the end of the `_createScene()` method:

```
_modifier =  new HeightMapModifier(_plane, bmp);
```

With the plane primitive now linked to our `HeightMapModifier` object, we can control its geometry by adjusting properties of the modifier. To dynamically control this process, let's create a keyboard interaction that allows us to make adjustments on the fly. Add the following code to the end of the `_createScene()` method to set up a handler for the `KeyboardEvent.KEY_DOWN` event:

```
stage.addEventListener(KeyboardEvent.KEY_DOWN, _onKeyDown);
```

The event handler is defined as a method called `_onKeyDown()`, which we create by adding the following code to the end of the `UsingHeightMapModifier` class definition:

```
private function _onKeyDown(ev : KeyboardEvent) : void
{
  switch (ev.keyCode) {
    case Keyboard.UP:
      _scale += 0.05;
      break;
    case Keyboard.DOWN:
      _scale -= 0.05;
      break;
  }
  _modifier.scale = _scale;
  _modifier.offset = -_scale * 127;
  _modifier.execute();
}
```

In the preceding handler, we begin by executing a switch statement to determine whether an up or down cursor key was pressed. In the case of `UP`, we increase the value of our global variable `_scale` by `0.05`. In the case of `DOWN`, we decrease the value of `_scale` by `0.05`. This variable is then applied to the `scale` property of the `HeightMapModifier` object, a value that controls the degree to which the pixel values in the height map offset the vertices in the mesh.

To update the modified position of the vertices in our plane mesh, we follow setting the `scale` property in the `_onKeyDown()` method with a call to the `execute()` method of the `HeightMapModifier` object. This call allows us to control the overall offset scale of the `HeightMapModifier` object while our Flash movie is running.

The `execute` method is a common feature of all modifier classes in Away3D. It has to be called manually to propagate updates in a modifier to updates in any associated mesh objects. The `scale` property in the `HeightMapModifier` class is an absolute fractional value that defaults to `1`. It can be set to negative values as well as positive values, which has the effect of reversing the direction of applied offset on the mesh vertices. This can be seen occurring in the `UsingHeightMapModifier` example if we continually press the `DOWN` key.

Because the vertices in the plane primitive all belong to faces pointing in the same direction, they all have the same vertex normal. This means that any perturbation applied by the `HeightMapModifier` object will move all vertices in the same direction—in this case, either up or down.

To keep the mesh centered in the scene while this movement occurs, we counterbalance the effect of the height map by adjusting the `offset` property of the `HeightMapModifier` object in the opposite direction. The height map values taken from the supplied Perlin noise bitmap range from 0 to 255, so to center these offsets, we need adjust the base `offset` property by an appropriately scaled –127 units. This is done in the final line of code of the preceding `_onKeyDown()` method by multiplying –127 by `_scale` to match the scaling applied to our height map values, which results in an overall vertex offset applied equally in both up and down directions from the starting position of the vertices in the plane.

Compile the code and press the up and down cursor keys to see a display similar to that shown in Figure 7-8.

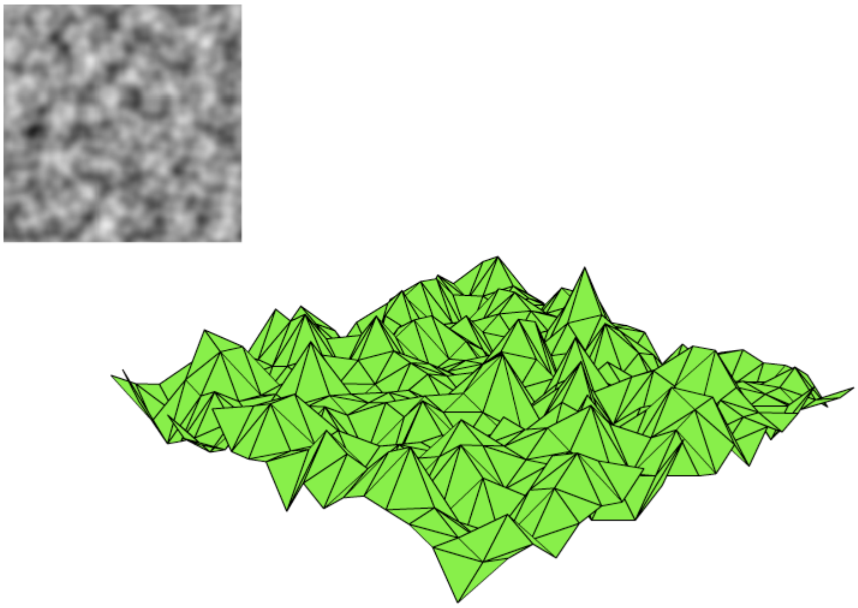


Figure 7-8. A Perlin noise height map applied to a simple plane primitive using the HeightmapModifier object in the UsingHeightMapModifier example

With planes, the `HeightmapModifier` class can be great for creating objects that simulate the terrain of a rocky landscape. However, we are not restricted to applying height maps to flat surfaces. The same can be done on any 3D mesh you create or import that has UV coordinate values. As an alternative example, let's use a sphere primitive as our base mesh by replacing the code creating the plane in the `_createScene()` method of the `UsingHeightMapModifier` example with the following code:

```
var _sphere : Sphere = new Sphere();
_sphere.radius = 200;
_sphere.segmentsW = 40;
_sphere.segmentsH = 20;
_view.scene.addChild(_sphere);
```

We also need to update the line that creates our `HeightMapModifier` object to replace the old `plane` variable with our new `sphere` variable.

```
_modifier =  new HeightMapModifier(_sphere, bmp);
```

Recompiling the example will display a sphere primitive in place of the plane. The cursor keys control the effect of the height map modifier in the same way; only this time, the vertex normal vectors are arranged in a radial configuration. This means that the position of each vertex is perturbed along its own path perpendicular to the sphere's surface, creating a rocky globe!

For a greater degree of control, you will most likely want to start experimenting with drawing your own heights maps. This approach has many potential applications, such as creating dents in a car model as it crashes against a wall or artificially ageing a face model to appear more wrinkled. The `HeightMapModfier` class should prove to be a versatile tool in your Away3D applications.

Summary

This chapter has walked through some of the basic geometry-creation methods that exist in Away3D, as well as a selection of advanced geometry classes and mesh modification tools. Several more classes exist in the `away3d.extrusions` and `away3d.modifiers` packages that we aren't able to delve into here, but you are welcome to read about them in the documentation section of the `www.away3d.com` site.

The following list recaps the main topics you should take with you from this chapter:

- Creating a custom 3D object from scratch can be easily done by extending the `AbstractPrimitive` class. This approach gives you access to some useful utility methods such as `createFace()`, `createVertex()`, and `createUV()`.
- According to Away3D conventions, all configuration properties on a class should be available to set via public properties as well as inside the `init` object of the constructor.
- UV mapping is a process that associates each vertex of a 3D face with its position on a 2D texture map of the mesh.
    - A mesh that has no UV data cannot have any type of bitmap material applied.
    - UV coordinates are stored as fractional values from 0 to 1 in two dimensions, where 1 represents the total width or height of the texture image.
    - The origin point of a UV map is defined as the bottom-left corner of the texture image.
- The extrusion classes allow you to create more complex geometry from simple 3D path and profile definitions.
    - The `PathExtrusion` class extrudes a profile shape along a smooth Bézier spline and is ideal for creating roads and ribbons.
    - The `LatheExtrusion` class rotates a profile shape around a defined axis and is ideal for creating a 3D object with axial symmetry, such as a vase or lamp.
- The modifier classes allow you to update the geometry of one or more existing mesh objects at the vertex level.
    - The `HeightMapModifier` class applies a 2D height map to any mesh with UV mapping. The data used for the height map is created from a bitmap image where pixel values represent the offset amount along the normal vector of each vertex.

In the next chapter, we will explore some of the options available to us when it comes to interactivity in Away3D. We will demonstrate the advantage of using 3D mouse events and cover some alternative control methods for advanced camera movement.

Chapter 8

Interactivity

The interactivity of any real-time 3D experience is key to its effectiveness, and in Flash is one of the best reasons to use Away3D over alternative approaches. If your application isn't interactive, then it's likely you have options available to you such as prerendered still images or video that potentially produce better visual results, due to the current processing limits imposed on rendering a real-time 3D scene. Having said that, interactive 3D can produce compelling visual results that are simply unattainable with conventional approaches, thanks to the relative freedom it gives the movement and animation of an interface.

In preceding chapters, the only type of interactivity covered in any great depth concerns the movement of a `HoverCamera3D` object. In this chapter, we will take a look at alternative ways to interact with 3D objects in a scene using the mouse, as well as general topics such as keyboard interaction.

Setting up the chapter base class

As always, let's start by setting up a base class. Create `Chapter08SampleBase` using the following stub code, which will serve as a starting point for our examples throughout this chapter. The theme in this case is chess, and the base class will construct a scene composed of a chessboard with a rook piece standing in the center.

```
package flash3dbook.ch08
{
  import away3d.containers.*;
  import away3d.core.math.*;
  import away3d.core.utils.*;
  import away3d.lights.*;
  import away3d.materials.*;
  import away3d.primitives.*;

  import flash.display.*;
  import flash.events.*;

  import flash3dbook.ch08.misc.*;
  public class Chapter08SampleBase extends Sprite
  {
    protected var _rook : RookMesh;
    protected var _board : Plane;
    protected var _view : View3D;

    [Embed('../../../assets/ch08/chessboard.jpg')]
    private var ChessBoardBitmapClass : Class;

    public function Chapter08SampleBase()
    {
      _createView();
      _createScene();
    }

    protected function _createView() : void
    {
    }


    protected function _createScene() : void
    {
    }

    protected function _onEnterFrame(ev : Event) : void
    {
      _view.render();
    }
  }
}
```

In the preceding code, we begin by declaring some global variables to allow quick access to key objects anywhere in the application. The private variable `ChessBoardBitmapClass` contains a bitmap image called `chessboard.jpg` embedded using the Flex `[Embed]` meta-tag (a technique discussed in greater detail in Chapters 4 and 5). The image consists of a black-and-white checkered pattern and will be used as the texture for the plane that represents our 3D chessboard object. At this point, it is a good idea to check that the path used in the meta-tag is pointing to the correct location of the image file on your hard drive. If you are building the examples from this chapter's resource files downloaded from www.friendsofed.com, you should already have the `chessboard.jpg` image located in the correct directory.

The remainder of the `Chapter08SampleBase` code contains stub definitions for the usual `_createView()` and `_createScene()` methods and a familiar `_onEnterFrame()` method triggering the `render()` method of the view.

Before continuing, we need to fill out the rest of our base class with code that sets up our 3D environment. Let's start by creating a view object and a light object by adding the following lines to the `_createView()` method:

```
_view = new View3D();
_view.x = 400;
_view.y = 300;
_view.camera.y = 700;
_view.camera.lookAt(new Number3D(0,0,0));
addChild(_view);

var light : PointLight3D = new PointLight3D();
light.x = 500;
light.y = 800;
light.z = -500;
light.ambient = 0.2;
_view.scene.addLight(light);

stage.addEventListener(Event.ENTER_FRAME, _onEnterFrame);
```

In the preceding code, the first task is to create the view object and position it correctly at the center of the stage. The default camera at (0, 0, –1000) is moved up by 700 units and uses the `lookAt()` method to point it at the scene's origin. Next, we create a `PointLight3D` object with a position 800 units above the scene's origin and 500 units to the right and front of the scene. Finally, we register the method `_onEnterFrame()` as a handler for the `ENTER_FRAME` event dispatched from the stage, so that the view renders its scene on every frame of the Flash movie.

Now, let's take a look at the code we need to add to the `_createScene()` method. Here, we will create a chessboard from a plane primitive and a rook piece from a model that has been converted to the ActionScript class `RookMesh` (a process that is covered in more detail in Chapter 4). Just like the `chessboard.jpg` image, the `RookMesh` class is an asset supplied in this chapter's resource files and is assumed to be located inside the `flash3dbook.ch08.misc` package.

```
var boardBitmap : BitmapData = Cast.bitmap(ChessBoardBitmapClass);

_board = new Plane();
_board.yUp = true;
_board.segmentsW = 8;
_board.segmentsH = 8;
_board.width = 800;
_board.height = 800;
_board.material = new WhiteShadingBitmapMaterial(boardBitmap);
_board.pushback = true;
_view.scene.addChild(_board);

_rook = new RookMesh();
_rook.material = new ShadingColorMaterial(0xffffff);
_view.scene.addChild(_rook);
```

In the preceding code, the first line takes the class definition of the embedded image `ChessBoardBitmapClass` and extracts its `BitmapData` to a local variable `boardBitmap`, using the utility class `Cast` (a technique introduced in Chapter 5). The next two blocks of code set up the 3D objects representing the chessboard and rook piece respectively. The board is created from a regular `Plane` primitive, with the black and white tiles' bitmap data `boardBitmap` wrapped in a `WhiteShadingBitmapMaterial` object to achieve some realistic shading. The rook piece is instantiated from the `RookMesh` class and has a shading color material applied with a base color of white. Compiling the `Chapter08SampleBase` class at this point displays the result shown in Figure 8-1. Let's move on to create some interactivity!

Away3D occasionally has a hard time with its standard Z-sorting algorithm, producing incorrect results for triangle sorting. This can cause rendering artifacts seen as flickering triangles where mesh elements take turns being in front and behind each other and is appropriately named Z fighting. In the preceding code for the `_createScene()` method, setting the `pushback` property of the board to `true` gives the mesh object a reduced sorting priority relative to others in the scene, keeping it behind objects where appropriate at close distances. Further tricks to avoid Z-fighting artifacts are covered in Chapter 10.

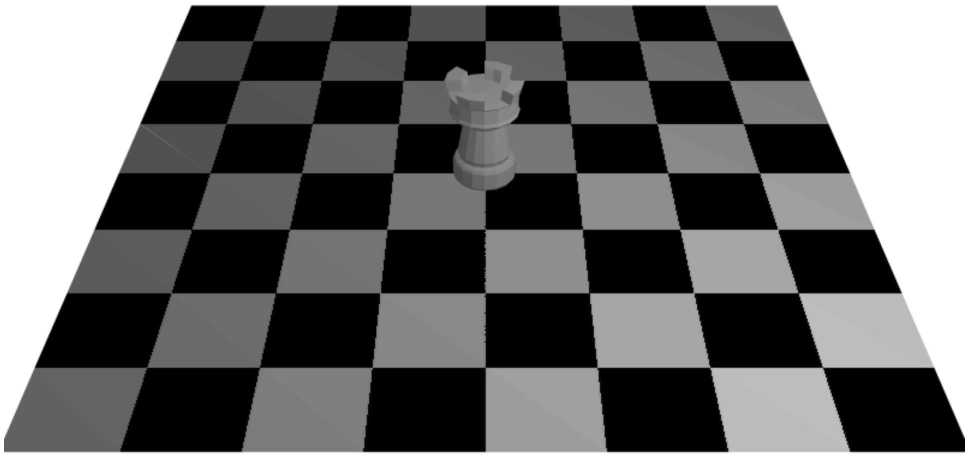


Figure 8-1. The newly created scene from the Chapter8SampleBase class that we will use in subsequent examples

Interacting with 3D objects in a scene

In regular Flash development, you are likely familiar with creating 2D objects such as buttons and sprites that can react to mouse events. In Away3D, the same type of mouse interaction is possible on 3D objects such as primitives and imported models. As you will see, the variety of 3D mouse events on offer covers the majority of native 2D mouse events and allows the creation of many interesting ways to interact with a 3D scene.

Introducing the MouseEvent3D object

Development using native mouse events in Flash tends to be centered around the `MouseEvent` class, available in the `flash.events` package. The equivalent event class in Away3D is the `MouseEvent3D` class, available in the `away3d.events` package, and it is used in much the same way.

To demonstrate some common types of mouse interaction in Away3D, we can extend our `Chapter08SampleBase` class to create an example that reacts to 3D mouse events broadcast from our rook model. Let's start by creating the shell of our example class with the following code:

```
package flash3dbook.ch08
{
  import away3d.core.base.*;
  import away3d.events.*;

  import flash.net.*;
  [SWF(width="800", height="600")]
  public class UsingMouseEvent3D extends Chapter08SampleBase
  {
    public function UsingMouseEvent3D()
    {
      super();
    }
  }
}
```

In this case, we will be listening for rollover, rollout, and mouse up events. To react to these, we need three event handler methods for the `MOUSE_OVER`, `MOUSE_OUT`, and `MOUSE_UP` events of the `MouseEvent3D` class.

First, let's create our event handlers for `MOUSE_OVER` and `MOUSE_OUT` events by adding class methods `_onMouseOver()` and `_onMouseOut()` to the end of the `UsingMouseEvent3D` class definition. These need to accept a single argument of the type `MouseEvent3D`, in a similar way to native `MouseEvent` handlers.

```
private function _onMouseOver(ev : MouseEvent3D) : void
{
  var obj : Object3D = ev.object;

  obj.scale(1.1);
}

private function _onMouseOut(ev : MouseEvent3D) : void
{
  var obj : Object3D = ev.object;

  obj.scale(1);
}
```

The `object` property of the `MouseEvent3D` argument received by the event handlers contains a reference to the target 3D object where the event originated. In this case, it is the 3D object that the mouse cursor either rolled over or rolled out of.

The preceding code scales the target 3D object to 110 percent when the `_onMouseOver()` method is triggered and back to 100 percent when the `_onMouseOut()` method is triggered. Now, we add a final method `_onMouseUp()` to the end of the `UsingMouseEvent3D` class definition, to be used as a handler for the `MOUSE_UP` event.

```
private function _onMouseUp(ev : MouseEvent3D) : void
{
  if (ev.object == _rook) {
    navigateToURL(new URLRequest('http://www.flash3dbook.com'));
  }
}
```

Here, we check whether the `MOUSE_UP` event originates from the rook model (which should always be true in this example) and then submit a `URLRequest` object to navigate to the 3D book web site.

3D mouse events in Away3D bubble up the scene graph hierarchy in the same way native Flash events bubble up the display list. So in the `UsingMouseEvent3D` example, our handlers for events dispatched from the rook model could just as easily be listening for events dispatched from the scene, as long as all handler methods check the origin object of the event, using the technique seen in the `onMouseUp()` method.

To hook up our event handlers to our event dispatchers, we need to add the following code to the `UsingMouseEvent3D` class constructor:

```
_rook.addEventListener(MouseEvent3D.MOUSE_OVER, _onMouseOver);
_rook.addEventListener(MouseEvent3D.MOUSE_OUT, _onMouseOut);
_rook.addEventListener(MouseEvent3D.MOUSE_UP, _onMouseUp);
```

Compile the `UsingMouseEvent3D` example, and try using your mouse to interact with the rook model. Rolling over the object scales it up, and rolling out scales it back down. Clicking the model when rolled over dispatches a URL request for the `www.flash3dbook.com` site address.

There is currently no event named `CLICK` in Away3D, but listening for `MOUSE_UP` events serves a similar purpose and will suffice in most situations.

It is often good practice to display a hand cursor when a clickable object is directly under the mouse position. This is an effective way of providing visual feedback to the user that whatever is under the cursor at that moment is clickable and is a default technique used by native Flash `SimpleButton` objects. Setting the `useHandCursor` property to `true` on a 3D object (such as our rook model) will enable the same feature in Away3D.

```
_rook.useHandCursor = true;
```

Add the preceding code to the end of the `UsingMouseEvent3D` constructor, and recompile the example to see the hand cursor displayed when the mouse is positioned over the rook model.

Using MouseEvent3D's scene coordinates

From regular mouse events in Flash, you may be familiar with the use of stageX, stageY, localX, and localY properties on MouseEvent objects. These act as handy references to the mouse position in relation to the broadcasting object when a particular mouse event is dispatched. In Away3D, the MouseEvent3D class has similar properties available that serve the same purpose for 3D positioning.

The sceneX, sceneY, and sceneZ properties of a MouseEvent3D object reveal the position at which a mouse event took place in global scene coordinates. The coordinates refer to the point on the surface of the object broadcasting the 3D mouse event that is located directly under the mouse cursor. Scene coordinates from a MouseEvent3D object can be used to update the location of a second object from the position returned, creating a visual representation of a mouse event occurring on the surface of the broadcasting object. This can be of great use when designing the interaction for a 3D scene.

To create an example that uses the scene coordinates from a MouseEvent3D object, let's extend the Chapter08SampleBase class with the following code:

```
package flash3dbook.ch08
{
  import away3d.events.*;

  [SWF(width="800", height="600")]
  public class DraggingObjectsInSpace extends Chapter08SampleBase
  {
    public function DraggingObjectsInSpace()
    {
      super();

      _rook.addEventListener(MouseEvent3D.MOUSE_DOWN, _onMouseDown);
      _rook.addEventListener(MouseEvent3D.MOUSE_UP, _onMouseUp);

      _rook.useHandCursor = true;
    }

    private function _onMouseDown(ev : MouseEvent3D) : void
    {
      _board.addEventListener(MouseEvent3D.MOUSE_MOVE, _onMouseMove);
    }

    private function _onMouseMove(ev : MouseEvent3D) : void
    {
    }

    private function _onMouseUp(ev : MouseEvent3D) : void
    {
      _board.removeEventListener(MouseEvent3D.MOUSE_MOVE, _onMouseMove);
    }
  }
}
```

The constructor method in the preceding code begins by setting up a listener for a 3D `MOUSE_DOWN` event and a 3D `MOUSE_UP` event dispatched from the `_rook` model. We also set the `useHandCursor` property of the rook to `true`, so that interaction with the mouse is implied on rollover.

When the mouse button is depressed while the cursor is over the rook model, the `MOUSE_DOWN` event is triggered and we start listening for 3D `MOUSE_MOVE` events dispatching from the `_board` object. The `_onMouseMove()` handler will continue to trigger until a 3D `MOUSE_UP` event is received from the `_board` object when the mouse button is released. To complete this example, we need to flesh out the `_onMouseMove()` method, adding the following code to place the rook in the scene position of the 3D mouse event received from the `_board` object:

```
_rook.x = ev.sceneX;
_rook.z = ev.sceneZ;
```

Since the object broadcasting the 3D `MOUSE_MOVE` event is a plane oriented along the X and Z axes of the scene, the preceding code will move the rook model to the position on the X and Z axes of the scene where the `MOUSE_MOVE` event was recorded. Because the rook is a direct child of the scene, we can rely on the `sceneX` and `sceneZ` properties to directly correlate with the `x` and `z` properties of the `_rook` object, without the need for any transformations of coordinate space.

If an object being manipulated by the scene position properties of a 3D mouse event were contained within a transformed (e.g., rotated) container, we would need to use some simple matrix math to convert from one coordinate space to another before setting position values for the object. More information about such a scenario can be found in Chapter 10.

Compiling the `DraggingObjectsInSpace` example at this point allows you to click and drag the rook model with the mouse, demonstrating the practical use mentioned earlier for the scene position information returned in a 3D mouse event. You will notice that as you drag the rook around, the motion is rather jerky. This is due to the `MOUSE_MOVE` event of the chessboard being interrupted by the geometry of the rook model whenever its position updates. The `_board` object will only dispatch `MOUSE_MOVE` events when its geometry is found directly under the cursor, without anything else being rendered in front. So our code will not update the rook's position until the cursor rolls off the rook model and has uninterrupted sight of the chessboard. This is not an ideal situation for the desired interaction, so to fix the problem, we can exclude the rook from registering mouse events while we are dragging, using the `mouseEnabled` property.

```
_rook.mouseEnabled = false;
_board.useHandCursor = true;
```

Add the preceding code to the end of the `_onMouseDown()` method. The `mouseEnabled` property is available on all scene objects in Away3D, and causes a 3D object to ignore 3D mouse events when set to `false`. For good measure, we are also making use of the hand cursor on the `_board` object while dragging, so that it remains visible while the mouse is clicked and held over the rook (see Figure 8-2).

With the preceding modification, the `MOUSE_UP` handler will not fire when the mouse button is released, because the `_rook` model has had its mouse interaction disabled. In order to ensure the `_onMouseUp()` method is triggered, we need to modify the constructor code to look like the following:

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

  _rook.addEventListener(MouseEvent3D.MOUSE_DOWN, _onMouseDown);
  _board.addEventListener(MouseEvent3D.MOUSE_UP, _onMouseUp);

  _rook.useHandCursor = true;
}
```

Here, the line in bold has been changed to listen to `MOUSE_UP` events on the `_board` object instead of the `_rook` model.

Now, we need to reverse our modifications in the `_onMouseDown` method when the mouse button is released, requiring the following code to be added to the end of the `_onMouseUp` method.

```
_rook.mouseEnabled = true;
_board.useHandCursor = false;
```

Recompiling the `DraggingObjectsInSpace` example will give us a much smoother, more responsive interaction when dragging the rook with the mouse (see Figure 8-2). Next, let's take a look at some other useful properties returned in the `MouseEvent3D` object.

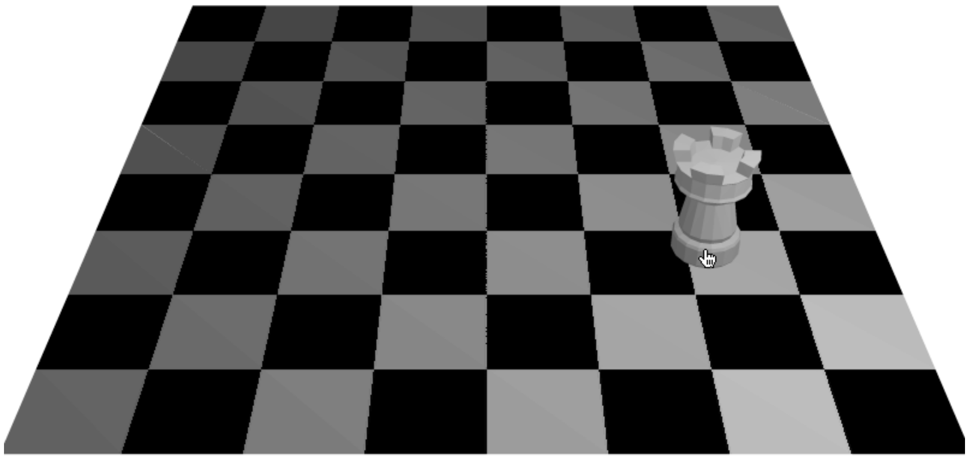


Figure 8-2. Dragging the rook model around the chessboard using the mouse (Notice the hand cursor active on the rook model.)

Using MouseEvent3D's UV coordinates

As mentioned in Chapter 5, UV coordinates represent 2D position vectors in texture space, using numbers between zero and one. Recall that a UV value of (0,0) maps to the lower left-hand corner of a material's `BitmapData` object, and a UV value of (1,1) maps to the upper right-hand corner.

When Away3D broadcasts a 3D mouse event, the 3D scene position mentioned in the previous section is not the only piece of information available. If the material of the broadcasting object is one that uses UV coordinates, the precise UV value under the mouse cursor is calculated (i.e., the position on the object's texture from which the event was broadcast). This information can be accessed as a UV object from the uv property on the MouseEvent3D object received by the event handler.

One potential use for this information reveals itself when we construct a painting interface that uses the mouse cursor to draw directly onto the surface of an object textured with a bitmap material, by painting into the BitmapData of the texture image. Let's create another example class extending Chapter08SampleBase, and implement a texture-painting tool on the surface of the chessboard.

```
package flash3dbook.ch08
{
  import away3d.events.*;
  import away3d.materials.*;

  import flash3dbook.ch08.*

  import flash.display.*;
  import flash.geom.*;

  [SWF(width="800", height="600")]
  public class PaintingOnObjects extends Chapter08SampleBase
  {
    private var _brush : Shape;

    public function PaintingOnObjects()
    {
      super();

      _brush = new Shape();
      _brush.graphics.beginFill(0xff0000, 0.5);
      _brush.graphics.drawEllipse(-10, -10, 20, 20);

      _board.useHandCursor = true;
      _board.addEventListener(MouseEvent3D.MOUSE_DOWN, _onMouseDown);
      _board.addEventListener(MouseEvent3D.MOUSE_UP, _onMouseUp);
    }

    protected function _onMouseDown(ev : MouseEvent3D) : void
    {
      _board.addEventListener(MouseEvent3D.MOUSE_MOVE, _onMouseMove);
    }

    protected function _onMouseMove(ev : MouseEvent3D) : void
    {
    }

    protected function _onMouseUp(ev : MouseEvent3D) : void
    {
      _board.removeEventListener(MouseEvent3D.MOUSE_MOVE, _onMouseMove);
    }
    }
}
```

In the preceding code, the listener setup for 3D mouse events is similar to the previous `DraggingObjectsInSpace` example, the main difference being that all 3D mouse event listeners are set up on the `_board` object. We also create a `Shape` object called `_brush` in the `PaintingOnObjects` constructor, which consists of a single semitransparent red circle. This will be used as our paintbrush to draw into the `BitmapData` object of the chessboard's material. The UV coordinate returned by the `MouseEvent3D` object is accurate to the pixel, but in this case, it is much more practical to paint with a big brush as opposed to individual points.

Once again, we need to add some code to the `_onMouseMove()` method to complete the example. We start by locating the material object of the chessboard and creating a variable `bmp` as a reference to the `BitmapData` object containing the texture of the chessboard's material.

```
var mat : WhiteShadingBitmapMaterial = _board.material as
WhiteShadingBitmapMaterial;
var bmp : BitmapData = mat.bitmap;
```

Next, we create a `Matrix` object `mtx`, which will be used by the `draw` method of the `BitmapData` object to position our brush. It requires only one piece of information—the 2D position vector in pixels that draws the red circle of `_brush` into the `BitmapData` object `bmp` at the precise location of the mouse cursor. This position is calculated from the `uv` property of the `MouseEvent3D` object and is applied by setting the `tx` and `ty` properties of `mtx` to the desired offset.

```
var mtx : Matrix = new Matrix();
mtx.tx = ev.uv.u * bmp.width;
mtx.ty = (1 - ev.uv.v) * bmp.height;
bmp.draw(_brush, mtx);
```

As mentioned earlier, UV coordinates are stored as a 2D position vector with each coordinate value represented as a number between 0 and 1. We calculate the position on the `BitmapData` object by multiplying these coordinates by the respective width and height values of the texture (the `width` and `height` properties of `bmp`) to convert them to pixel units.

The v property of a UV coordinate is a common gotcha, since contrary to the majority of 2D coordinate systems, 0 represents the bottom edge of the texture image, and 1 the top edge. This is consistent with the Y axis orientation in Away3D but not with the coordinate system used in a `BitmapData` object! To convert it, we are required to subtract the `v` value from 1 before multiplying by the height of the texture.

The final task to perform in the `_onMouseMove()` method is to reset the `bitmap` property for the chessboard's material object. This flags the material for a redraw when the `render()` method of the view is next executed. Without this, Away3D would be unaware that the contents of the material had changed, since the action of updating pixel values in a `BitmapData` object alone broadcasts no events.

```
mat.bitmap = bmp;
```

This is the simplest way to perform a material reset on a bitmap material and has very little impact on performance compared to, say, re-creating the material using a new `WhiteShadingBitmapMaterial` object.

> *Away3D utilizes a namespace called `arcane`, through which you can access properties used in internal engine systems. One such internal system is a series of Boolean properties that temporarily mark an object as dirty (requiring a redraw). In this case, the property used to flag a material for redrawing is called `_materialDirty`. If you are familiar with namespaces, using the `arcane` namespace and setting `_materialDirty` to `true` on the `mat` object in the preceding code might be considered a cleaner solution for flagging a material for redrawing, as it rids us of the need for an extra `bmp` variable in our code.*

Compile the `PaintingOnObjects` example, and try clicking and dragging the mouse over the chessboard. The interaction should act like a paintbrush on the surface of the plane, applying the red circles of the `_brush` object to the portion of the texture directly underneath the mouse cursor, with results similar to the output displayed in Figure 8-3.

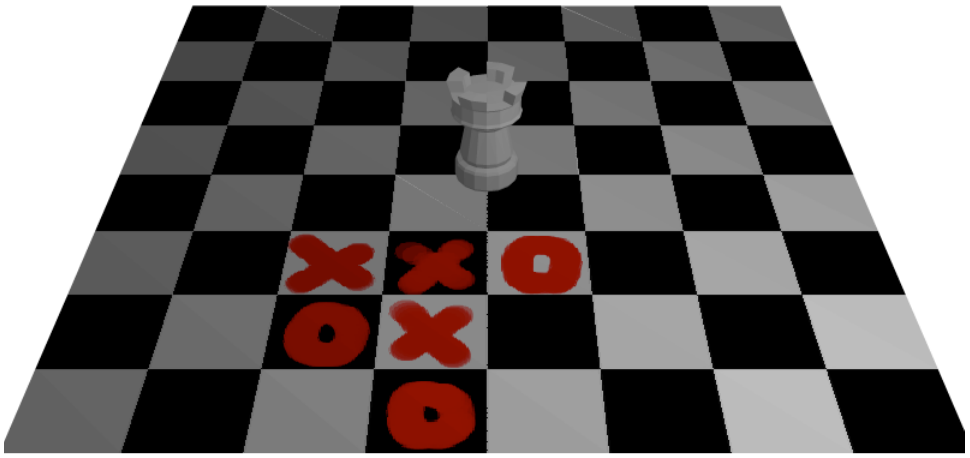


Figure 8-3. Painting on the chessboard using the UV coordinates returned in the MouseEvent3D object

First-person camera keyboard controls

Now that you are an expert at interaction with the scene on an object level, what other methods can we use to control a scene? A commonly used mechanism for navigating a 3D environment involves moving the camera based on keyboard and mouse input, as if you were moving within the scene yourself, with the camera representing your position. This kind of control is referred to as a first-person camera control. Using the keyboard to interact with an Away3D scene is really no different from using the keyboard in any other type of Flash application. It involves listening for native Flash keyboard events and the state of specific keys before updating the scene and rendering it to the view.

In our first example, we will create a simple first-person navigation system using the keyboard. Subsequent examples will extend functionality to add complimentary mouse interaction for looking around.

Walking with the keyboard

In many first-person games and applications, the W, A, S, and D keyboard keys are used to move the camera forward, left, back, and right. Keeping in mind that not everyone will have a keyboard layout that suits this interaction, it is a good idea to provide an alternate set of keys to use as a backup control method. We will use the arrow keys for this purpose, as they are positioned in the same inverted T arrangement for the majority of keyboards.

There are several coding techniques that can be used to achieve the type of keyboard interaction we are aiming for. The common approach we will use here is to keep track of individual key states: keys can either be down (pressed) or up (released). Each time the scene updates (i.e., immediately before we call `render()` on the view), the camera updates its position according to the keys that are currently registered in their down state.

To test this interaction, let's extend the `Chapter08SampleBase` class to set up a new example.

```
package flash3dbook.ch08
{
  import flash.events.*;
  import flash.ui.*;

  import flash3dbook.ch08.*;

  [SWF(width="800", height="600")]
  public class FirstPersonCamera extends Chapter08SampleBase
  {
    public function FirstPersonCamera()
    {
      super();
    }

    protected override function _onEnterFrame(ev : Event) : void
    {
      _view.render();
    }
  }
}
```

Add the following variable definitions for key state to the top of our new class, one for each key. These properties are Boolean values, because each key exists in one of only two possible states.

```
protected var _keyUp : Boolean = false;
protected var _keyDown : Boolean = false;
protected var _keyLeft : Boolean = false;
protected var _keyRight : Boolean = false;
```

Now, let's create some event listeners for keyboard events and update the key state variables accordingly. We'll start by creating a method handler for the `KEY_DOWN` event, dispatched when a user presses a key, by adding the following `onKeyDown()` method to the end of our `FirstPersonCamera` class definition:

```
private function _onKeyDown(ev : KeyboardEvent) : void
{
  if (ev.charCode == 119 ||  ev.keyCode == Keyboard.UP) {
    _keyUp = true;
  }
  else if (ev.charCode == 97 ||  ev.keyCode == Keyboard.LEFT) {
    _keyLeft = true;
  }
  else if (ev.charCode == 115 ||  ev.keyCode == Keyboard.DOWN) {
    _keyDown = true;
  }
  else if (ev.charCode == 100 ||  ev.keyCode == Keyboard.RIGHT) {
    _keyRight = true;
  }
}
```

This method will execute every time a keyboard key is pressed. Our key state variables are defined by their function, grouping the outcome for alphanumeric and arrow keys by the resulting direction of movement. The first `if` statement in the `_onKeyDown()` method checks whether the key pressed is the W key (ASCII code 119) or the up arrow key. If either are true, we set the value of the `_keyUp` state to `true`, indicating that a key relating to the up direction has been pressed. We go on to do the same for all other directions.

A similar method handler needs to be created for when the user releases a key, the difference being that any released keys set their respective key state variables to `false`. For this, we add the following `_onKeyUp()` method to the end of our `FirstPersonCamera` class definition:

```
private function _onKeyUp(ev : KeyboardEvent) : void
{
  if (ev.charCode == 119 ||  ev.keyCode == Keyboard.UP) {
    _keyUp = false;
  }
  else if (ev.charCode == 97 ||  ev.keyCode == Keyboard.LEFT) {
    _keyLeft = false;
  }
  else if (ev.charCode == 115 ||  ev.keyCode == Keyboard.DOWN) {
    _keyDown = false;
  }
  else if (ev.charCode == 100 ||  ev.keyCode == Keyboard.RIGHT) {
    _keyRight = false;
  }
}
```

Connecting up our event handlers to trigger on keyboard events, we register the `_onKeyUp()` and `_onKeyDown()` methods as handlers for their respective keyboard events by adding the following lines of code to the end of the constructor method of the `FirstPersonCamera` class:

```
stage.addEventListener(KeyboardEvent.KEY_DOWN, _onKeyDown);
stage.addEventListener(KeyboardEvent.KEY_UP, _onKeyUp);
```

Now, when a user presses or releases a relevant key, that key will be logged as pressed (`true`) or released (`false`) in our key state variables.

Before we compile the `FirstPersonCamera` example, we need to add some code that moves our camera in reaction to our changing key state variables. This is best done in the `_onEnterFrame` method so that continuously holding down a key (or several keys) will continuously update the camera movement, until the key is released.

For the forward and backward movement, we will use the `moveForward()` and `moveBackward()` methods on the camera to move forward and back when the `_keyUp` and `_keyDown` variables are active. As you saw in Chapter 3, these methods transform an object's position relative to its local axes. However, the camera is tilted down slightly, so in order to keep our movement along a horizontal plane, we will need to compensate our local forward and back motion with some local up and down motion. This can be accomplished with the `moveUp()` and `moveDown()` methods and some simple trigonometry. In order to rotate the camera we can increment its rotationY property. Add the following code to the start of the `_onEnterFrame()` method of the `FirstPersonCamera` class, before `render()` is called on the view:

```
// Move forward or backward
if (_keyUp) {
  _view.camera.moveForward(15*Math.cos(_view.camera.rotationX*Math.PI/180));
  _view.camera.moveUp(-15*Math.sin(_view.camera.rotationX*Math.PI/180));
} else if (_keyDown) {
  _view.camera.moveBackward(15*Math.cos(_view.camera.rotationX*Math.PI/180));
  _view.camera.moveDown(-15*Math.sin(_view.camera.rotationX*Math.PI/180));
}

// Turn left/right
if (_keyLeft)
       _view.camera.rotationY -= 3;
else if (_keyRight)
       _view.camera.rotationY += 3;
```

Compile the `FirstPersonCamera` example, and try navigating around the chessboard using the arrow keys or W, A, S, and D alphanumeric keys.

When compiling the `FirstPersonCamera` example, you may have to first click the stage of the Flash movie with the mouse for it to gain focus and start receiving keyboard events.

In this example, we are relying solely on keyboard interaction to navigate around the scene, which limits our available movements somewhat. Next, let's take a look at some more advanced approaches to first-person interaction that use the mouse alongside the keyboard for a greater degree of control.

Looking around by dragging the mouse

A common approach for first-person interaction in many commercial 3D games allows the user to look around a scene by moving the mouse. We can achieve the same interaction to some extent here, with one significant limitation—in Flash, we have information only about the cursor position on screen, and usually only while the cursor is positioned over the Flash movie window. For most Flash applications, this limitation isn't a problem, but in a first-person camera interface, it limits our ability to continuously rotate the camera while looking around our scene. Even if Flash is running in full screen-mode and we can be certain the cursor is always inside the Flash movie window, when the cursor hits the edge of the screen, our perceived mouse movement halts until the cursor is moved back across the screen in the other direction.

One way to work around this problem is to use a **dragging** interaction. This rotates the camera only when the user holds down the mouse button and drags the cursor, and removes the maximum rotation limit as rotations can be incremented with consecutive mouse drags. It also has the advantage of detecting mouse movement outside the Flash movie window, as long as the mouse button is initially pressed down while the cursor is still inside.

To accomplish this interaction, we first need to listen for a standard `MOUSE_DOWN` event and, when that occurs, start listening to `MOUSE_MOVE` events to update the rotation of the camera. Each time the mouse moves, we apply a rotation that reflects the distance traveled by the cursor since the last received `MOUSE_MOVE` event. We also need to listen for `MOUSE_UP` events to know when to stop reacting to `MOUSE_MOVE` events. Let's use the previous example class `FirstPersonCamera` as our starting point and extend its functionality with the following class:

```
package flash3dbook.ch08
{
  import flash.events.*;
  import flash.geom.*;

  import flash3dbook.ch08.*;

  [SWF(width="800", height="600")]
  public class FirstPersonCameraWithDrag extends FirstPersonCamera
  {
    private var _lastPoint : Point = new Point();

    public function FirstPersonCameraWithDrag()
    {
      super();

      stage.addEventListener(MouseEvent.MOUSE_DOWN, _onMouseDown);
      stage.addEventListener(MouseEvent.MOUSE_UP, _onMouseUp);
    }

    private function _onMouseDown(ev : MouseEvent) : void
    {
      stage.addEventListener(MouseEvent.MOUSE_MOVE, _onMouseMove);
      stage.addEventListener(Event.MOUSE_LEAVE, _onMouseLeave);
    }

    private function _onMouseMove(ev : MouseEvent) : void
    {
    }

    private function _onMouseUp(ev : MouseEvent) : void
    {
      stage.removeEventListener(MouseEvent.MOUSE_MOVE, _onMouseMove);
      stage.removeEventListener(Event.MOUSE_LEAVE, _onMouseLeave);
    }

    private function _onMouseLeave(ev : Event) : void
    {
      stage.removeEventListener(MouseEvent.MOUSE_MOVE, _onMouseMove);
      stage.removeEventListener(Event.MOUSE_LEAVE, _onMouseLeave);
    }
  }
}
```

The preceding code is laid out in a very similar manner to the click-and-drag functionality applied to 3D objects earlier in this chapter. When the mouse button is clicked and held, the listener for `MOUSE_MOVE` events will activate, and when the button is released, the listener will deactivate. Notice that to stop listening for mouse movement, we actually require two event handlers: a `MOUSE_UP` handler for when the mouse button is released with the cursor position still over the Flash movie window and a `MOUSE_LEAVE` handler for when the mouse button is released with the cursor position outside the Flash movie window, after having been dragged out.

To complete the interaction, we add the following code to the `_onMouseMove()` method to calculate the rotation of the camera when a `MOUSE_MOVE` event is broadcast. First, we calculate the distance the mouse has travelled since the last `MOUSE_MOVE` event and use that value to increment the camera rotation. Then, we prepare for the next event by storing the current mouse position in our global class variable `_lastPoint`.

```
_view.camera.rotationX -= (stage.mouseY - _lastPoint.y) / 5;
_view.camera.rotationY += (stage.mouseX - _lastPoint.x) / 5;

_lastPoint.x = stage.mouseX;
_lastPoint.y = stage.mouseY;
```

The pitch increment of the camera is calculated using the y increment of the mouse position, and the yaw increment of the camera is calculated using the x increment of the mouse position. Converting from pixels to degrees, we divide each result by 5 to give an appropriate ratio of rotation speed to mouse movement. As we saw in the previous example, rotation of a first person camera can use standard rotation properties, so we apply our calculated pitch and yaw increments to the respective `rotationX` and `rotationY` properties of the camera object.

Before compiling our `FirstPersonCameraWithDrag` example, we need to make sure that the first time the mouse movement is calculated in a mouse drag, the increment value starts from the point where the user started dragging (i.e., the point where the mouse button was first clicked). To achieve this, we can copy the last two lines of the preceding code snippet and add them to the start of the `_onMouseDown()` method so that it now looks like the following:

```
private function _onMouseDown(ev : MouseEvent) : void
{
  _lastPoint.x = stage.mouseX;
  _lastPoint.y = stage.mouseY;

  stage.addEventListener(MouseEvent.MOUSE_MOVE, _onMouseMove);
  stage.addEventListener(Event.MOUSE_LEAVE, _onMouseLeave);
}
```

Here, the additional lines to the `_onMouseDown()` method are highlighted in bold. Compile the `FirstPersonCameraWithDrag` example, and try clicking and dragging the mouse to look around. The camera should rotate in the direction you drag, with movement in both a horizontal and vertical direction. You are still able to walk around as before, and better still, both interactions can be used simultaneously. The left and right arrow keys (and the A and D keys) still control the camera rotation as before, but have been made redundant in this example by the introduced mouse interaction. Next, let's take a look at a different approach for using the mouse to look around.

Looking around by scrubbing the mouse

The main downside with the dragging approach to mouse controlled rotation in the previous `FirstPersonCameraWithDrag` example is that a user has to repeatedly click and drag the mouse to execute sharp turns, because a single drag has a finite distance and therefore executes a finite rotation. An alternative method that doesn't suffer the same limitations uses an interactive technique known as **scrubbing**.

A scrubbing interface in an audio or video application tends to consist of a handle or wheel that can be dragged left or right to control the speed and direction of playback. The speed depends on how far off center you drag the control, and when released, the control jumps back to its center position and playback stops. The same interaction can be applied to our camera rotation control, so that when the mouse cursor is scrubbed from anywhere inside the Flash movie, the camera will rotate continuously in the direction of scrub until the mouse button is released.

To implement this interaction, let's again use the `FirstPersonCamera` class as our starting point and extend its functionality with the following:

```
package flash3dbook.ch08
{
  import flash.events.*;
  import flash.geom.*;

  import flash3dbook.ch08.*;

  [SWF(width="800", height="600")]
  public class FirstPersonCameraWithScrub extends FirstPersonCamera
  {
    private var _lastPoint:Point = new Point();
    private var _scrub:Boolean;

    public function FirstPersonCameraWithScrub()
    {
      super();

      stage.addEventListener(MouseEvent.MOUSE_DOWN, _onMouseDown);
      stage.addEventListener(MouseEvent.MOUSE_UP, _onMouseUp);
    }

    private function _onMouseDown(ev : MouseEvent) : void
    {
      _scrub = true;

      _lastPoint.x = stage.mouseX;
      _lastPoint.y = stage.mouseY;

      stage.addEventListener(Event.MOUSE_LEAVE, _onMouseLeave);
    }

    private function _onMouseUp(ev : MouseEvent) : void
    {
      _scrub = false;
      stage.removeEventListener(Event.MOUSE_LEAVE, _onMouseLeave);
    }


    private function _onMouseLeave(ev : Event) : void
    {
      _scrub = false;
      stage.removeEventListener(Event.MOUSE_LEAVE, _onMouseLeave);
    }

    protected override function _onEnterFrame(ev : Event) : void
    {
      _view.render();
    }
  }
}
```

The preceding code uses the same listener setup as the `FirstPersonCameraWithDrag` example for detecting when the cursor is being dragged, but instead of using a `MOUSE_MOVE` event for updating the camera rotation, we simply monitor the up and down state of the mouse button with a global class variable `_scrub`.

The `_onEnterFrame()` method at the end of the class overrides the original keyboard interaction of the `FirstPersonCamera` class. Let's start by adding back in that functionality, this time with a sidestepping motion instead of a rotation for left and right keypresses. The following code needs to be inserted before the `render` call in the `_onEnterFrame()` method:

```
// Move forward or backward
if (_keyUp) {
       _view.camera.moveForward(15*Math.cos(_view.camera.rotationX*Math.PI/180));
       _view.camera.moveUp(-15*Math.sin(_view.camera.rotationX*Math.PI/180));
} else if (_keyDown) {
       _view.camera.moveBackward(15*Math.cos(_view.camera.rotationX*Math.PI/180));
       _view.camera.moveDown(-15*Math.sin(_view.camera.rotationX*Math.PI/180));
}

// Sidestep left/right
if (_keyLeft) {
       _view.camera.x -= 15*Math.cos(_view.camera.rotationY*Math.PI/180);
       _view.camera.z += 15*Math.sin(_view.camera.rotationY*Math.PI/180);
} else if (_keyRight) {
       _view.camera.x += 15*Math.cos(_view.camera.rotationY*Math.PI/180);
       _view.camera.z -= 15*Math.sin(_view.camera.rotationY*Math.PI/180);
}
```

Previously, the code for reacting to left and right keypresses incremented or decremented the `rotationY` property of the camera, but here we are using a similar technique to that seen in the code for moving forward and back, incrementing the x and z position properties of the camera. Compiling the code at this point will demonstrate that sidestepping with the camera is now possible using the left and right cursor keys (or A and D character keys).

Now, let's add the code for rotating the camera using the scrubbing method. The rotation code is only executed when the value of the `_scrub` variable returns `true` and is updated on every frame by adding the following code to the start of the `_onEnterFrame()` method.

```
// Rotate camera up/down and left/right
if (_scrub) {
        _view.camera.rotationX -= (stage.mouseY - _lastPoint.y) / 200;
        _view.camera.rotationY += (stage.mouseX - _lastPoint.x) / 200;
}
```

When a scrub is in progress, the X and Y distance from the start of a scrub (the point at which the mouse button was pressed) can be calculated using the `_lastPoint` class variable and the current X and Y position of the cursor. Once again, we need to convert between pixels and degrees for the `pitch()` and `yaw()` increment arguments. Because we are incrementing the total distance scrubbed by the mouse *every frame*, the dividing value needs to be considerably higher to achieve a useable rotation speed, and is chosen here as 200.

Recompile the code, and try out the scrubbing interaction by clicking and dragging the mouse anywhere in the Flash movie. You'll notice that as long as the mouse button is held down, the rotation of the camera will continue in the direction of drag. The speed and direction of motion is relative to the distance the mouse cursor moves from the starting position of the scrub. Releasing the mouse button will halt the motion of the camera.

The advantage of using a scrubbing interaction over others is its ability to rotate the camera all the way round its axes with a single click. However, it demonstrates less precision when positioning the camera compared to the dragging interaction in the `FirstPersonCameraWithDrag` example seen earlier.

The mouse interactions described in the last two sections rely on the overall frame rate of the Flash movie for their absolute speeds of motion. This rate can vary from machine to machine depending on performance. To accommodate for this, and prevent inconsistent speeds of motion across machines, the elapsed duration between frames needs to be taken into account. This can be done by calling the Flash method getTimer() each frame and comparing it to the previous.

Summary

This chapter has described a variety of different interactions that can be applied to a 3D scene based on the input from mouse and keyboard. You have seen how the 3D mouse event API available in Away3D is arranged in a similar fashion to the native mouse event system available in Flash and how you can use the properties available from the `MouseEvent3D` object to perform advanced interactions based on event information. We have also explored a variety of first-person camera interactions. We hope these forms of 3D interaction inspire you to create some great interactive experiences with 3D in Flash!

Take some time to study the following list, and make sure that you are happy with your understanding of the topics covered in this chapter.

- 3D mouse events in Away3D provide specific 3D information on mouse interactions, and are represented by the `MouseEvent3D` class.
- Supported 3D mouse events include `MOUSE_DOWN`, `MOUSE_UP`, `MOUSE_OVER`, `MOUSE_OUT`, `MOUSE_MOVE`, `ROLL_OVER`, and `ROLL_OUT`.
- The `sceneX`, `sceneY`, and `sceneZ` properties on `MouseEvent3D` objects represent the location of a 3D mouse event on the surface of the broadcasting object.
- The `uv` property on `MouseEvent3D` objects represents the texture coordinate under the mouse position of the broadcasting object.
- The `mouseEnabled` property of a 3D object can be used to disable mouse interactions for that object. The `useHandCursor` property turns the mouse cursor into a link hand symbol when the cursor is hovered over the 3D object.
- Standard keyboard events applied in Flash can be used in exactly the same way to create keyboard controls in 3D.

The next chapter takes a tour of the various forms of animation available in Away3D.

Chapter 9

Animation

Right from its inception, a main strength of Flash has been its animation capabilities. Despite the arrival of ActionScript programming shifting the focus somewhat, animation (or **tweening** in Flash authoring terms) is still considered a core feature of Flash. As yet, we have no timeline functionality for animating 3D objects aside from some limited 2.5 effects (the "postcards in space" approach) using sprites in Flash CS4. Away3D helps to fill in the gaps by offering a number of different options for 3D animation, as we will see in this chapter.

Tweening in code is something you've most likely encountered already as an alternative to tweening on the timeline. This is one of many approaches adopted by Away3D for its animation controls, although all techniques discussed here will require some form of scripted control. We will also examine how to import animations created in external 3D modeling packages such as 3ds Max, Maya, and Blender.

The basics of scripted animation

One defining element of all animation in Flash, including video, is the concept of frames. By quickly stepping through a sequence of images, we can simulate continuous motion. When animating using the timeline in Flash Professional, we get to create **keyframes**. These are frames in which we explicitly define how an object is oriented at that precise moment. In-between frames are then created by the tool to achieve a smooth motion (or **tween**) between the two defined keyframe positions.

When animating in ActionScript, we still have to work within the same frame-based approach seen in the timeline, gradually adjusting property values over time (often from a start value to an end value). Assuming that we have an Away3D scene set up containing a cube primitive held in the `_cube` property, the following `ENTER_FRAME` event handler would achieve the motion displayed in Figure 9-1:

```
private function _onEnterFrame(ev : Event) : void
{
  _cube.rotationY += 10;
}
```

The preceding code increments the cube's rotation by 10 degrees, creating a continuously rotating cube moving at the rate of 10 degrees per frame.

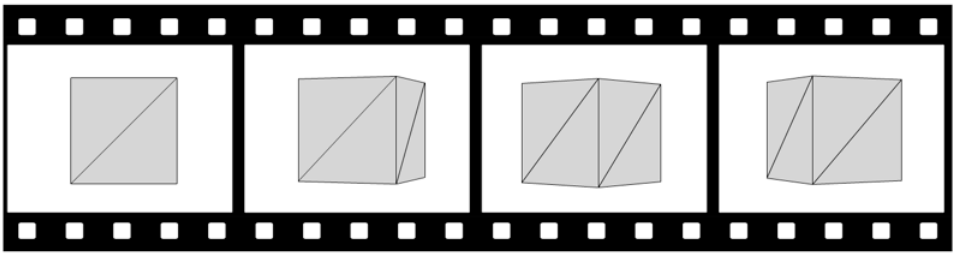


Figure 9-1. All animation in Flash is based around the concept of frames. Individual images are played in sequence to create continuous motion.

As animations get more and more complex, keeping track of our tweening values can become tedious. Luckily, a wide variety of free, open source libraries called **tweening engines** track these values for us. Examples of popular tweening engines include GTween, TweenLite, Tweensy, and Tweener. In this chapter, we will use a custom class called `GenericTweener` to keep our examples self-contained and generic. `GenericTweener` can be found in the `flash3dbook.ch09.misc` package inside this chapter's resource download available from `www.friendsofed.com`, but the code using `GenericTweener` can be easily modified if you prefer to use your tweening engine of choice.

Before we continue, let's set up our simple base class to take care of the generic Away3D code for the examples in this chapter.

```
package flash3dbook.ch09
{
  import away3d.containers.*;

  import flash3dbook.ch09.misc.*;

  import flash.display.*;
  import flash.events.*;

  public class Chapter09SampleBase extends Sprite
  {
    protected var _view : View3D;

    public function Chapter09SampleBase()
    {
      _createView();
      _createScene();
    }

    protected function _createView() : void
    {
      _view = new View3D();
      _view.x = 400;
      _view.y = 300;
      _view.camera.y = 500;
      _view.camera.z = -1000;
      _view.camera.lookAt(_view.scene.position);

      addChild(_view);
      addEventListener(Event.ENTER_FRAME, _onEnterFrame);
    }

    protected function _createScene() : void
    {
    }

    protected function _onEnterFrame(ev : Event) : void
    {
      _view.render();
    }
  }
}
```

Here, we have defined our usual base class methods, with `_createView` setting up a `View3D` instance with a standard camera 500 units above the origin and 1,000 units in front, `_createScene` empty and ready for custom scene instructions, and `_onEnterFrame()` calling our `render()` method on the view. Now, we are ready to try out the animation features of Away3D, starting with a look at how to apply tweening to the motion of a 3D object.

Using basic tweening

Using an ActionScript 3.0 tweening engine with Away3D is really no different from using it in any other context. Let's take a quick look at how it can work, before we move on to more complex animation features specific to the Away3D engine.

In a very simple example that extends the `Chapter09SampleBase` class and overrides the `_createScene()` method, we make a variant of one of the chessboard examples from Chapter 8. When we click the surface of a plane primitive, another 3D object—in this case, a cube primitive—will tween from its current position to the scene position of the 3D mouse event. Let's start by creating the document class definition, which contains two global variables for holding the plane and cube instances, as well as all the necessary class imports.

```
package flash3dbook.ch09
{
  import away3d.primitives.*;
  import away3d.events.*;
  import away3d.materials.*;
  import flash3dbook.ch09.*;
  import flash3dbook.common.*;
  [SWF(width="800", height="600")]
  public class TweeningIn3D extends Chapter09SampleBase
  {
    private var _plane : Plane;
    private var _cube : Cube;

    public function TweeningIn3D()
    {
      super();
    }

    protected override function _createScene() : void
    {
    }
  }
}
```

As usual, the `_createScene()` method is where we set up our 3D objects to be used in the example. First, let's create a cube primitive by adding the following code to the method:

```
_cube = new Cube();
_cube.y = 50;
_cube.material = new ColorMaterial(0x888888);
_view.scene.addChild(_cube);
```

Here, we assign the cube a simple flat color material imported from the `away3d.materials` package. We then position the cube 50 units above the scene's origin by setting its y property to 50. Because its height is 100 units by default and its registration point is in the center, this positioning allows the cube to sit directly on top of the plane primitive we create next, which by default is centered horizontally on the scene's origin:

```
_plane = new Plane();
_plane.yUp = true;
_plane.material = new ColorMaterial(0xcccccc);
_plane.scale(10);
_plane.pushback = true;
_plane.addEventListener(MouseEvent3D.MOUSE_UP, _onClick);
_view.scene.addChild(_plane);
```

The plane primitive is created with its `yUp` property set to `true` so that it will be oriented horizontally and assigned a color material slightly lighter than the cube primitive. The default dimension of 100 × 100 units is much too small for our purposes, so we increase its overall size using the `scale()` method. We also apply a trick from Chapter 8 that forces the plane to ensure its triangles do not unnecessary overlap the cube (and any other object) using the `pushback` property. To enable mouse interaction, we add an event listener for `MOUSE_UP` events on the plane and finish by adding the plane to the scene. The `_onClick` handler method is something we need to create by adding the following code to the end of the `TweeningIn3D` class definition:

```
private function _onClick(ev : MouseEvent3D) : void
{
  GenericTweener.tween(_cube, 0.5, { x: ev.sceneX, z : ev.sceneZ });
}
```

Here, we are using our `GenericTweener` class to animate the cube to the position in the scene returned from the 3D mouse event. From Chapter 8, you may recall a similar demonstration using the `sceneX`, `sceneY`, and `sceneZ` properties from the `MouseEvent3D` object. In this case, we take the `sceneX` and `sceneZ` positions and use them as the target `x` and `z` positions in our tween declaration for the cube primitive.

Let's take a closer look at what at what is going on inside our `GenericTweener` class. The static method `tween()` takes a minimum of three arguments. The first argument is the object to be animated, referred to as the **target**. The second argument represents the duration of the desired tween in seconds. The third argument is an untyped ActionScript object (typically defined using the curly brace notation seen in the Away3D initializer object), in which we define the property names and values we wish to tween. The values defined here represent the finishing values for the tweening animation. In the preceding code for the `_onClick()` method, we instruct `GenericTweener` to tween the `x` and `z` properties of the cube to the values of `ev.sceneX` and `ev.sceneZ` respectively, over a period of 0.5 seconds.

Compile the example, and click anywhere on the plane to tween the cube to that position. The animation runs for half a second from start to finish, and if you click again in the middle of one animation, a new tween will be created that overrides the existing tween.

With this approach, it is easy to animate objects in a straight line from one point to another. You can also apply a tween to the scale or rotation properties of a 3D object, replacing the `x` and `z` properties in the `TweeningIn3D` example with `scaleX`, `scaleY`, and `scaleZ`, or `rotationX`, `rotationY`, and `rotationZ` properties.

So far, we have demonstrated how familiar ActionScript tweening tools can be used to animate 3D objects. But Away3D offers several animation tools and techniques of its own, enabling many other options for tackling 3D animation. These tools will be the focus of our attention for the remainder of this chapter, starting with a look at how we can tween objects along a path more complex than a straight line.

Path tweening

From Flash Professional CS4, there are new timeline options for tweening along a path in 3D. However, in CS3 we are stuck with tweening in 2D. This omission can make complex animation movement difficult if we stick to using simple tweening methods. Even now in CS4 and CS5, we are restricted to tweening the 3D transforms of 2D display objects, which isn't ideal. What we really need is a way of defining a path in 3D space that can then be used as a basis for tweening the position and rotation of a 3D object. Luckily, Away3D provides just that in the form of the `Path` and `PathAnimator` classes.

An Away3D `Path` object is a data structure with a variety of uses, but in this case, we are interested in how it can be used for animation. The class is located in the `away3d.core.geom` package and allows us to define a spline in 3D space as a series of `Number3D` objects. A **spline** is a collection of Bézier curves laid end to end forming a highly configurable complex curve. Bézier curves are discussed in more detail in Chapter 6, and an example of a single quadratic **Bézier curve** is displayed in Figure 9-2. As you can see, it is defined using three separate vector positions, represented in the `Path` object by a triplet of `Number3D` objects.

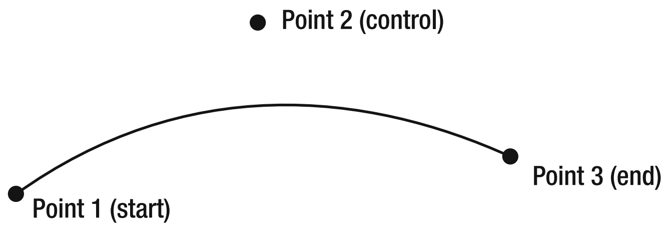


Figure 9-2. In this curved path created using a quadratic Bézier curve, a triplet of Number3D objects defines the corresponding start, control, and end points of the curve.

Once a `Path` object is created, it can be used for animating a 3D object with the help of the `PathAnimator` class. This acts as a control mechanism for a single 3D object, using a property called `progress` to adjust the position of the object along the spline defined by the given `Path` object. `progress` is a generic property used in all animators in Away3D and takes a fractional value between 0 and 1, with 0 representing the start of the animation and 1 representing the end of the animation. With this setup, the `progress` property can be used by a tweening engine to control the animation of the 3D object, with the `PathAnimator` object updating the relevant variable values of the associated 3D object in relation to the `progress` property value.

Let's create an example that uses the `Path` and `PathAnimator` classes by extending the base class and overriding the `_createScene()` method:

```
package flash3dbook.ch09
{
  import away3d.animators.*;
  import away3d.core.geom.*;
  import away3d.core.math.*;
  import away3d.materials.*;
  import away3d.primitives.*;

  import flash3dbook.common.*;

  [SWF(width="800", height="600")]
  public class AnimatingAlongPaths extends Chapter09SampleBase
  {
    private var _cube : Cube;
    private var _path : Path;
    private var _animator : PathAnimator;

    public function AnimatingAlongPaths()
    {
      super();
    }

    protected override function _createScene() : void
    {
    }
  }
}
```

The preceding code declares three global properties we will use for our path tween. The `_cube` property will hold the 3D object to be animated, `_path` will hold an instance of the `Path` class containing our spline data, and `_animator` will hold an instance of the `PathAnimator` class that links everything together and creates our tweening interface.

Now, let's fill out the `_createScene()` method, starting with the code for creating our cube primitive.

```
_cube = new Cube();
_cube.scale(0.5);
_cube.material = new ColorMaterial(0x888888);
_view.scene.addChild(_cube);
```

Here, we create a cube primitive with default dimensions of 100 × 100 × 100, but adjust its overall size to 50 percent with the `scale()` method. We use the same simple color material seen in the previous example and finish by adding the object to the scene.

Next, we create our path from an array of positions in 3D space, defined as `Number3D` objects. Each curve of a path's spline is a quadratic Bézier definition composed of three positions: start, control, and end. Our path is composed of three curves in the shape of a shield, created by adding the following code to the end of the `_createScene()` method:

```
var pathArray : Array = [
 // First segment
 new Number3D(100, 0, 100),
 new Number3D(100, 0, -100),
 new Number3D(0, 0, -150),

 // Second segment
 new Number3D(0, 0, -150),
 new Number3D(-100, 0, -100),
 new Number3D(-100, 0, 100),

 // Third segment
 new Number3D(-100, 0, 100),
 new Number3D(0, 0, 100),
 new Number3D(100, 0, 100)
];
_path = new Path(pathArray);
```

The constructor for the `Path` class requires an array passed as an argument that contains at least three `Number3D` objects (the minimum for defining one quadratic Bézier curve). Here, we define three such triplets, using new lines within the array definition for easier reading. Each `Number3D` object is initialized with the desired vector values by accepting three constructor arguments that correspond to the x, y, and z values of the position.

If you look closely, you'll see that the first position in each successive curve is identical to the last position in the preceding curve. This is done to ensure the resulting path is continuous. Figure 9-3 illustrates the shape of our path, as seen looking straight down from above. You can clearly see how some positions are shared between curve definitions.

For situations in a path where a straight edge is required, the control point can be positioned directly between the start and end point positions, as is the case with the path between points E and A in Figure 9-3.

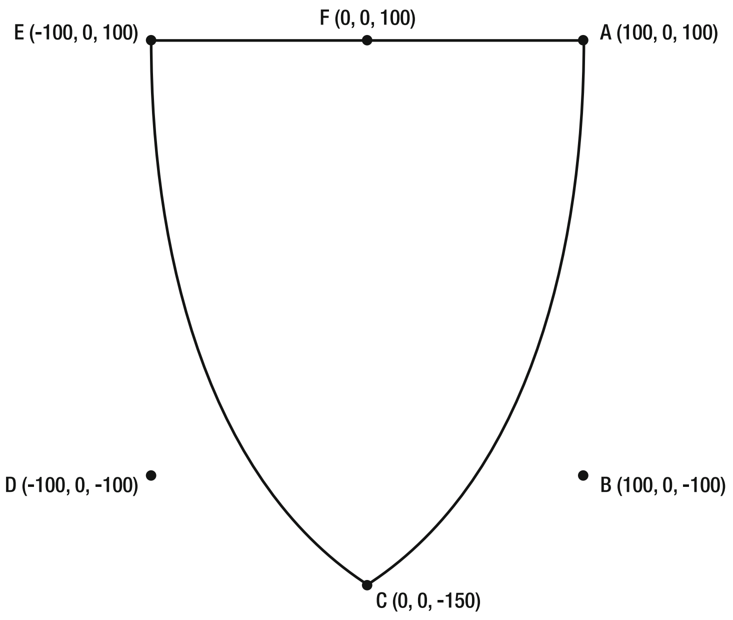


Figure 9-3. The spline defined by our Path object (from above) in the AnimatingAlongPaths example, which will be used to tween the cube's position

Now that we have defined our `Path` object, we can set about creating a `PathAnimator` class to control the movement of our cube. We do this by adding the following code to the end of the `_createScene()` method:

```
_animator = new PathAnimator(_path, _cube);
```

The `PathAnimator` object associates the `_cube` object with the `_path` object by passing them as required arguments in its constructor. Once the instance is created, there are several properties that can be set to configure the method in which the animator object adjusts the position of the 3D object when its `progress` property is updated. Two of the most significant of these are described here:

- alignToPath: This Boolean value defines whether the animating 3D object should be rotated to align its local Z axis in the direction of travel. This property operates in a similar way to the motion tween option Orient to path we see when using a guide layer in Flash Professional. The default setting is true.
- offset: This Number3D object defines an additional vector offset added to the calculated position vector of the 3D object along the path. This is useful for situations when, for example, your path follows a road and you want to animate the camera above the road surface. If the path used in the generation of the road were also used for the camera tween, without an offset vector the camera position would end up directly intersecting the road surface. Applying an offset value of (0, 50, 0) in this case would raise the tweened position to 50 units above the road, a much better vantage point.

All that's left to do is set up our tweening engine to control the progress property of the PathAnimator object and animate our cube. Once again we use the GenericTweener class to perform tweening duties with the following line of code added to the end of the _createScene() method:

```
GenericTweener.tween(_animator, 5.0, { progress: 1 });
```

Compiling the code will display the cube animating around the shield shape of our defined path. Because the PathAnimator object initializes the position of its animating 3D object to the start of the path, the preceding code will tween the _cube object's position from the start to the end of the path over a period of 5 seconds. As it moves, the cube is rotated to always align with the direction of motion thanks to the default setting of the alignToPath property.

To better illustrate the position vectors used in the construction of a path, a small sphere primitive can be drawn in the scene for each position in the path's Number3D array. We can modify the AnimatingAlongPaths example to achieve this by looping through the locally created pathArray object with the following code added to the end of the _createScene() method:

```
for each (var p : Number3D in pathArray) {
  var marker : Sphere = new Sphere();
  marker.scale(0.1);
  marker.material = new ColorMaterial(0x000000);
  marker.x = p.x;
  marker.y = p.y;
  marker.z = p.z;

  _view.scene.addChild(marker);
}
```

For each Number3D object p in the array, a sphere called marker is created and positioned according to the x, y, and z properties of p. A black color material is applied, and the sphere is scaled to 0.1 (10 percent) of its default size so as not to be obtrusive. Recompiling the code displays an output similar to Figure 9-4, with the cube animating along the same path as before.

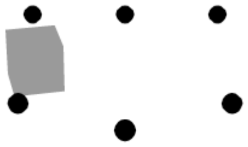


Figure 9-4. The cube primitive in the AnimatingAlongPaths example, animating along a path created using an array of positions illustrated by the black spheres

Importing animation

Because 3D mesh objects are generally composed of hundreds of tessellating triangles, it can be difficult to animate them in code using anything more than simple object transformations. For more complex movements, it makes sense to consider using a dedicated 3D animation package to create a 3D animation that is saved in one of the common 3D file formats and then import that animation for use in Flash.

Away3D offers the option to import 3D animations in both COLLADA and MD2 file formats—two of the most widely used formats in real-time 3D animation. Both of these offer animation at a mesh level (i.e., the movement of individual vertex positions in a mesh to manipulate the shape of a 3D object over time). This type of animation comes in handy for tasks such as character animation, where a model requires limbs that move independently of its body while being connected to the same 3D mesh. Let's take a closer look at the comparable restrictions and benefits to using the two animation formats on offer and how they are used in an Away3D application.

Working with MD2 animations

The **MD2** file format is a binary format, meaning that the information it contains is stored as one long indexed byte array. A byte array is one of the most compact ways of storing raw data, an important consideration when dealing with 3D animations that have the potential to contain large amounts of vertex position references.

MD2 files use a frame-based vertex animation format, storing animation data as a series of keyframes. Each keyframe contains an array of position vectors representing the position of every vertex in the model for that frame. The shift in vertex positions between keyframes can be interpolated (tweened) so that the amount of required data for an animation can be reduced. Typically, an MD2 animation runs at no more than 5 or 6 keyframes per second.

Importing an MD2 file

MD2 files are parsed into Away3D using the `MD2` class located in the `away3d.loaders` package. The loading process is very similar to the loading process used in Away3D for any external model resource, which is covered in more detail in Chapter 4. Let's create a new example to import and test an MD2 file by extending the `Chapter09SampleBase` class with the following document class definition:

```
package flash3dbook.ch09
{
  import away3d.animators.*;
  import away3d.core.base.*;
  import away3d.events.*;
  import away3d.loaders.*;
  import away3d.loaders.utils.*;
  import away3d.materials.*;

  public class UsingMD2Animation extends Chapter09SampleBase
  {
    private var _loader : Loader3D;
    private var _animator : VertexAnimator;

    public function UsingMD2Animation()
    {
      super();
    }

    private function _onLoadSuccess(ev : Loader3DEvent) : void
    {
    }

    protected override function _createScene() : void
    {
    }
  }
}
```

To load any 3D file into Away3D, we require an instance of the `Loader3D` class. In the preceding code, we create a global variable `_loader` to store a reference to the `Loader3D` instance we will use in this example. Next, we define a class variable `_animator` to store an instance of the `VertexAnimator` object. This class is found in the `away3d.animators` package alongside the `PathAnimator` object we used in the previous example, although in this case, we will be extracting the data for our `_animator` object from our parsed MD2 animation file, rather than creating one from scratch.

The preceding class definition continues with a definition of a new method stub `_onLoadSuccess()` to be used as our handler method for a `LOAD_SUCCESS` event from the `Loader3D` object. Once again, we finish with the usual `Loader3D` method override, ready for our custom code.

The first step in loading any model file is to create a parser for the correct file format. In the case of this example, add the following code to the empty `_createScene` method:

```
var md2 : Md2 = new Md2();
md2.material = new BitmapFileMaterial('../assets/ch09/turtle.jpg');
```

Here, we create an instance of the `MD2` parser class and then set the `material` property of the `MD2` instance to a new type of bitmap material. The `BitmapFileMaterial` class operates as a standard bitmap material once created, but its constructor replaces the required `BitmapData` object of a `BitmapMaterial` class with a string defining an image file path that `BitmapFileMaterial` automatically loads at runtime. Once loaded, the bitmap data for the material's texture is extracted from the image. This type of material can be easily used in place of a standard bitmap material and is particularly useful in situations when you don't have the necessary texture data for the model embedded in your SWF file from the outset.

In the UsingMD2Animation example, a BitmapFileMaterial is used because an MD2 file does not explicitly define a material name in its file format, and therefore cannot load one automatically in the parser. The MD2 format is a simple structure and only allows for one mesh to be defined in the file data. Therefore, we can easily use a single texture image for the mesh, which in this case is a JPEG file called turtle.jpg. Check that the file path matches the location on your hard drive, although as usual, this should already be the case if you are using the chapter resource download from www.friendsofed.com. Remember that, in this instance, the file path needs to be correct from the location of the compiled SWF rather than the location of the ActionScript file.

The next step in the UsingMD2Animation example is to create and initialize the Loader3D object we require by adding the following lines of code to the end of the _createScene() method:

```
_loader = new Loader3D();
_loader.addEventListener(Loader3DEvent.LOAD_SUCCESS, _onLoadSuccess);
_loader.loadGeometry('../assets/ch09/seaturtle.md2', md2);
_view.scene.addChild(_loader);
```

We begin by instantiating a Loader3D object and adding a listener for the LOAD_SUCCESS event, using _onLoadSuccess() as our event handler. We then call the loadGeometry() method, which instructs the Loader3D object to load and parse the MD2 file specified in the file path string. Again, check to make sure that the path reflects the location of your MD2 file.

As explained in Chapter 4, the Loader3D object serves as a container for the geometry of the file being loaded. This means we can add it to the scene before the file has finished loading, knowing that once loading is complete, the container will be replaced by the contents of the file. Compiling the code at this point will load and display the sea turtle model shown in Figure 9-5. All that's left to do is add some code to the _onLoadSuccess() method to trigger the playback of our animation sequence.

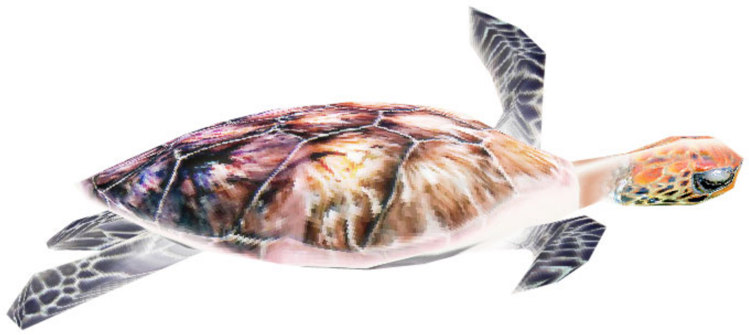


Figure 9-5. The sea turtle mesh loaded from an MD2 file in the **UsingMD2Animation** example

Playing an MD2 animation

To play back the animation stored in an MD2 file, we need access to the AnimationLibrary object created by the MD2 parser, as this is where our keyframe information is stored. The AnimationLibrary object can be retrieved either from the resulting mesh object created by the parser or from the MD2 parser object itself. In both cases, the AnimationLibrary object is stored in a property called animationLibrary. We will use the former method in our UsingMD2Animation example, accessing the mesh object from a property on the Loader3D object called handle by adding the following code inside the empty _onLoadSuccess() method:

```
var mesh : Mesh = _loader.handle as Mesh;
mesh.x = 100;
var animationLibrary : AnimationLibrary = mesh.animationLibrary;
```

Here, the handle property returns the Away3D model resulting from the parsing process. In this case, we want to cast the returned value to Mesh because we know that the MD2 format contains only a single mesh definition, and handle is cast by default as a generic Object3D object.

Next, we need to extract our definition of the animation we wish to play from the retrieved AnimationLibrary object. MD2 files contain all keyframes of animation in a single block of data, but several different animation loops can be defined from within this data with the help of keyframe names. Each keyframe is assigned a name string that is read into the animation object during parsing. To discern between different animation loops, different keyframe name prefixes are used, with the remainder of the string usually used for the frame number of the loop (e.g., walk01, walk02, walk03, run01, run02, and run03). The MD2 parser automatically sorts frames into different animation loops according to their prefixes, then stores them in the AnimationLibrary object. Because we are dealing with vertex animation, the animation loops created are represented by VertexAnimator objects.

In Away3D, trace() statements that communicate information about internal operations can be revealed using the Debug class found in the away3d.core.utils package. In the case of file parsers, information is displayed about the contents of the file being parsed. This is especially useful for the MD2 parser, where the trace() output reveals the frame name of all keyframes encountered in the file. To activate this output, enter the line Debug.active = true; before the Loader3D object is triggered in our example, and import the necessary Debug class from away3d.core.utils.Debug

In the UsingMD2Animation example, we want to retrieve the VertexAnimator object for the animation prefix “swim”. We do this by adding the following line of code to the end of the _onLoadSuccess method:

```
_animator = animationLibrary.getAnimation("swim").animator as VertexAnimator;
```

Here, we extract our desired animator instance from the animator property on the getAnimation() method of the AnimatorLibrary object. The returned value is cast by default as a generic Animator object because AnimationLibrary handles more than one type of animator, so we need to recast the output to VertexAnimator and store the result in our class property, _animator.

Now that we have our MD2 animation data contained in our _animator property, we have more than one option for playback. Besides allowing the use of the generic progress property you saw in the previous section’s AnimatingAlongPaths example, the VertexAnimator class introduces the concept of time with the update() method. Because a vertex animation is separated into a number of discrete frames, we can apply a frames-per-second definition to the playback speed and hence derive a virtual animation timeline mimicking the operation of the timeline in Flash Professional. A time value can then be set to control playback of the animation, representing the position of a virtual playhead in our constructed timeline setup.

The `update` method accepts a single argument representing the time in seconds we want to jump to in our animation. The idea is to trigger `update` on every frame of the Flash movie with an incrementing time argument to move the playhead of the animation to the next position on our animation timeline. The `VertexAnimator` object holds a number of configuring properties that affect the internal representation of the timeline, a selection of which are defined here:

- `delay`: A Number value that represents the delay to the start of the animation in second (the time before the animation starts playing). Defaults to `0`.
- `loop`: A `Boolean` value that defines whether the animation plays continuously or halts after a single run through, defaults to `true`.
- `fps`: An `int` value that controls the rate at which keyframes are played relative to time (in other words, the keyframes-per-second value), defaults to `6`.
- `smooth`: A `Boolean` value that defines whether in-between frames (tweening frames) are linearly interpolated to avoid jerky movement, defaults to `true`.

As a test, let's set a few configuring properties on our `VertexAnimator` instance before we continue, by adding the following code to the end of the `_onLoadSuccess` method:

```
_animator.delay = 0;
_animator.loop = true;
_animator.fps = 5;
_animator.interpolate = true;
```

To complete the `UsingMD2Animation` example, we add a call to the `update` method on every frame of the Flash movie by overriding the existing `_onEnterFrame` method of the chapter base class with the following code added to the end of the `UsingMD2Animation` class definition:

```
protected override function _onEnterFrame(ev : Event) : void
{
if (_animator)
  _animator.update(getTimer()/1000);
  super._onEnterFrame(ev);
}
```

Here, we provide a steadily incrementing time value for `update()` by using the native `getTimer` method to return the time passed from the start of the Flash movie, dividing it by 1,000 to get a value in seconds. Recompiling the code displays the same output shown in Figure 9-5; only now, the sea turtle gently swims through space.

As an alternative to using the `update()` method on every frame, the `VertexAnimator` object comes equipped with a few methods you are familiar with using on native `MovieClip` objects to control timeline-based animations. The `play()`, `stop()`, `gotoAndPlay()`, and `gotoAndStop()` methods can all be used in a similar way on the `VertexAnimator` object, giving us the option to control MD2 animations in a more familiar manner. As an example, try deleting the previous override made to the `_onEnterFrame` method and add the following code to the end of the `_onLoadSuccess` method:

```
_animator.play();
```

Recompiling the code produces exactly the same result achieved using the `update` method; the only difference being that the timeline clock used to advance the playhead is calculated internally by the `VertexAnimator` object.

Working with COLLADA animations

The **COLLADA** (Collaborative Design Activity) file format is an XML-based format that stores data as human-readable text. It is currently one of the most widely compatible exchange formats around thanks to its open format policy, and is used in many professional 3D modeling applications and 3D engines.

Because COLLADA files (which generally use the extension .dae) are essentially text files, they can be easily opened and edited using text editors. However, text encoding also results in large file sizes, which is a problem compounded by the verbose nature of COLLADA files. Compared to the equivalent binary formats, such as MD2 and 3DS, COLLADA files are bloated in size, leading to longer download and parsing times -- not an ideal format for the Web.

Animation data in a COLLADA file is stored using a technique known as **bones animation**; instead of storing arrays of vertex positions, bones animation uses a virtual skeleton to animate a mesh based on the position of various limbs. Each vertex in a mesh is attached to a bone or number of bones, and animations are executed by transforming each bone's position in much the same way that the earlier `TweeningIn3D` example transformed the position of a cube primitive. Thus, one advantage of using bones animation over vertex animation is less data and more precision.

Importing a COLLADA file

COLLADA files are parsed in Away3D using the `Collada` class located in the `away3d.loaders` package. Let's set up a new example to import and test a COLLADA file by extending the `Chapter09SampleBase` class with the following document class:

```
package flash3dbook.ch09
{
  import away3d.animators.*;
  import away3d.containers.*;
  import away3d.events.*;
  import away3d.loaders.*;
  import away3d.loaders.utils.*;

  public class UsingColladaAnimation extends Chapter09SampleBase
  {
    private var _loader : Loader3D;
    private var _animator : BonesAnimator;

    public function UsingColladaAnimation()
    {
      super();
    }

    private function _onLoadSuccess(ev : Loader3DEvent) : void
    {
    }

    protected override function _createScene() : void
    {
    }
  }
}
```

The preceding code is very similar to the previous `UsingMD2Animation` example, the only difference being the object type of our `_animator` variable that has been swapped to a `BonesAnimator` object. The COLLADA parser uses this type of animator object to store the bones animation data extracted from the file. As before, we start by filling out the code required for the `_createScene` method:

```
var collada : Collada = new Collada();
_loader = new Loader3D();
_loader.addEventListener(Loader3DEvent.LOAD_SUCCESS, _onLoadSuccess);
_loader.loadGeometry('../assets/ch09/puma_run.dae', collada);
_view.scene.addChild(_loader);
```

Here, we don't require a material definition to be set, as the COLLADA file stores a reference to the image files it requires for textures and loads them automatically. Once again, be sure to check that the path used in the `loadGeometry` method matches the location of your downloaded COLLADA file for this example.

Compiling the `UsingColladaAnimation` example will display the puma model shown in Figure 9-6.

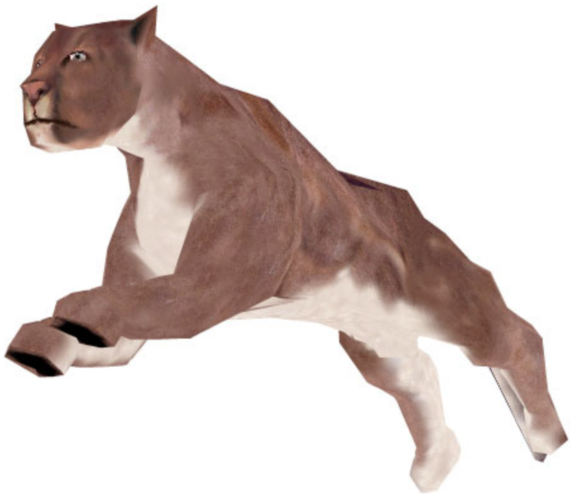


Figure 9-6. A puma model loaded from a COLLADA file in the UsingColladaAnimation example

Playing a COLLADA animation

To trigger the animation, we first need to extract the animator object using the same approach seen for our MD2 file in the previous `UsingMD2Animation` example by adding the following code to the `_onLoadSuccess()` method:

```
var container : ObjectContainer3D = _loader.handle as ObjectContainer3D;
container.scale(50);
container.x = 50;
container.y = -50;
container.rotationY = 135;
var animationLibrary : AnimationLibrary = container.animationLibrary;
_animator = animationLibrary.getAnimation("default").animator as BonesAnimator;
_animator.play();
```

Here, our `handle` property from the loader is cast as an `ObjectContainer3D` object, which is the expected output from a parsed COLLADA file. The extracted `BonesAnimator` object has some similar properties to the `BonesAnimator` object, such as `start` and `loop`. However, in this case, we are sticking with the default values. The transport methods are also present, which allows us to call `play()` straight away on the `BonesAnimator` object to set things in motion.

One difference to note between the expected content for MD2 and COLLADA animations shown in these examples is in the general handling of animation loops. While it is perfectly possible for a COLLADA file to have differently named loops for its bones animations, many 3D software exporters favor exporting a single loop of animation with no name definitions at all. In these cases, the string `"default"` can be used in the `getAnimation()` method of the `AnimationLibrary` object to extract a single global timeline containing all animation data. The same approach can be used with MD2 files, although loop names are far more common with this format.

Recompiling the `UsingColladaAnimation` example will animate the puma model in a run sequence. As with vertex animations, many bones animations are designed for looping in order to provide an easy way to continuously animate a character.

Creating programmatic animation with bones

The animation loops shown in the previous two examples were predefined in an external modeling application and offer little in the way of modification once loaded. We can play them forward and backward, pause them, and jump to a new loop, but what if we want to define our own movement within an animation timeline? Sometimes, an application such as a game requires finer control over the way a model moves in order to disguise prerendered animation loops as something a little more interactive. In these situations, we can take a programmatic approach to animation, controlling the movement of vertices directly rather than through an imported animation sequence.

As an example, predefined animations of a falling character might end with the model lying flat on the ground. But if the character falls against a box, its body should deform to follow the shape of the box, producing a different animation outcome. This kind of effect is often achieved using the same skeleton rig created for bones animation, as the amount of individual elements requiring programmatic animation for this format is much less than the equivalent animation created with keyframe vertices. Let's take a look at how we can go about setting up a COLLADA model for animation in this manner.

Defining an animation rig

In the world of 3D, bones are a way to define a skeletal structure in a 3D model, sometimes called an **armature** or **rig**. Instead of animating a mesh by pushing vertices around, an animator or programmer can rotate, translate, and scale bone objects, and the rigged mesh will deform around the new positions. An important point to note is that a bone can be parented to another bone, resulting in the child bone following the movement of the parent. If you think of the bones in your arm as an example of an animated rig, your lower arm is parented to your upper arm. Your lower arm follows any transformation your upper arm performs, but your lower arm can also have its own transformation applied cumulatively to the upper arm transformation.

Figure 9-7 shows an example of a simple rig applied to the puma model used in the previous `UsingColladaAnimation` example. The lower backbone in the middle of the body serves as the top-level bone, to which all other bones are connected via parenting. If we were to move or rotate the backbone, the entire rig would follow, moving or rotating the entire mesh. Four bones are connected to the backbone: a pair that makes up the pelvis, one that represents the upper back, and one that marks the start of the tail. This hierarchy continues all the way out to bones representing the head, feet, and tail. In this example, rotating the right pelvis bone would rotate the entire right leg, and all the mesh vertices attached to that part of the rig would be affected. Likewise, if you were to only rotate the lower leg bone in the same limb, only the mesh vertices attached to the lower leg bone would be affected.

In Away3D, there is currently no way of creating an animation rig from scratch. In order to programmatically control a mesh animation with bones, a model needs to be rigged in an external 3D modeling program and then exported as a COLLADA file, ready for use. Methods of creating a rig in an external modeler are outside the scope of this book, but the process should be familiar to any professional 3D animator.

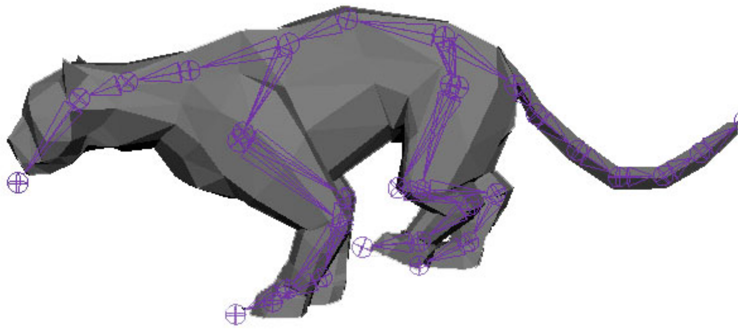


Figure 9-7. Representation of the animation rig used for the puma model seen in the previous UsingColladaAnimation example.

Bone tweening

The puma model uses a quadrupedal rig to create a convincing running motion, with the animation data stored as a series of bone transformations. In Away3D, it is a relatively easy matter to ignore this data and interactively manipulate bones with just the imported rig. Let's take a look at how this is done with a rigged COLLADA model by creating a new document class with the following code:

```
package flash3dbook.ch09
{
  import away3d.animators.data.*;
  import away3d.containers.*;
  import away3d.events.*;
  import away3d.loaders.*;
  import flash.events.*;

  [SWF(width="800", height="600")]
  public class AnimatingColladaBones extends Chapter09SampleBase
  {
    private var _loader : Loader3D;
    private var _skeleton : ObjectContainer3D;
    private var _left_arm : Bone;
    private var _left_leg : Bone;
    private var _right_arm : Bone;
    private var _right_leg : Bone;

    public function AnimatingColladaBones()
    {
      super();
    }

    private function _onLoadSuccess(ev : Loader3DEvent) : void
    {
    }

    protected override function _createScene() : void
    {
    }
  }
}
```

As with the previous two examples, we start with a class variable called `_loader` used to store a `Loader3D` object. `_skeleton` will be used to store the loaded COLLADA model, and the remaining variables will be used for storing references to the specific bone objects we will be animating. Finishing off the class definition is our familiar `_onLoadSuccess()` method, stub, and our override for the `_createScene()` method, which we fill out with the following code:

```
var collada : Collada = new Collada();
_loader = new Loader3D();
_loader.addEventListener(Loader3DEvent.LOAD_SUCCESS, _onLoadSuccess);
_loader.loadGeometry('../assets/ch09/skeleton.dae', collada);
_view.scene.addChild(_loader);
```

This is almost identical to the code used for the previous `_createScene()` method in the `UsingColladaAnimation` example, aside from the file path, which uses a COLLADA file called `skeleton.dae`. The file contains no animation data, just the animation rig for the vertices in the mesh that in this case is a bipedal rig of a human skeleton. The underlying bones structure (the rig) mimics the visible bones structure (the mesh) to a degree—we stop short of having bone objects for fingers and toes! Check that the file path used is correct for the location of your download chapter files, then move on by adding the following code to the `_onLoadSuccess` method:

```
_skeleton = _loader.handle as ObjectContainer3D;
_skeleton.y = -200;
_skeleton.scale(20);
```

Here, we extract the skeleton model from the loader, using its `handle` property. Once again, we cast `handle` as an `ObjectContainer3D` object, as this is the expected root object returned for an imported COLLADA file.

Adjusting the position and size of an imported model is a common requirement in Away3D, as there are no standards set for these attributes in the 3D modeling software used to create the files. In the preceding code, we move the position of the model 200 units down the Y axis, and scale it up 20 times to produce an orientation and size we can work with.

Now, we need to extract the bone objects from the 3D container object and store them in our previously created class variables. For this task, we use the method `getBoneByName()` on the 3D container object of the COLLADA file, supplying the name string of each bone we wish to return as an argument that adds the following code to the end of the `_onLoadSuccess` method:

```
_left_arm = _skeleton.getBoneByName('arm_l');
_left_leg = _skeleton.getBoneByName('leg_l');
_right_arm = _skeleton.getBoneByName('arm_r');
_right_leg = _skeleton.getBoneByName('leg_r');
```

In Away3D, bone objects inherit from 3D container objects and, therefore, allow you to add and remove child bones to create a bone hierarchy. The `getBoneByName` method returns a bone object contained at any point within the hierarchy, as long as a match is found for the given name.

As with MD2 files, it is sometimes useful to view a `trace()` output of the contents of the COLLADA file being imported to discover which name strings to use when accessing the contained elements. Again, we can use the debug trace methods of Away3D to get a printout of the parser by adding the line `Debug.active = true;` to the `_createScene()` method, at a point before the `Loader3D` object is triggered.

Compiling the code at this point will load and display the skeleton shown in Figure 9-8, standing vertically with its arms positioned horizontally. This pose is a common default for a bipedal rig. Now, we can use the bone references we have extracted to apply some interactive animation. As a straightforward example of how this might work, let's turn our skeleton into a Jumping Jack!

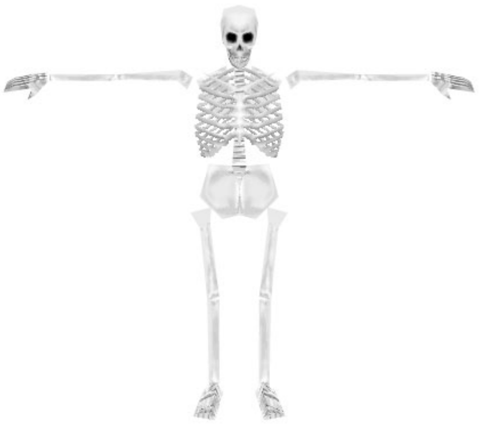


Figure 9-8. The skeleton.dae model loaded in the AnimatingColladaBones example, before animation

Because bones inherit from 3D container objects, they can be transformed in the same way with rotation, position, and scaling values. Let's apply some rotation to our extracted bones by overriding the existing `_onEnterFrame()` method of the chapter base class with the following code added to the end of the `AnimatingColladaBones` class definition:

```
protected override function _onEnterFrame(ev : Event) : void
{
  if (_skeleton) {
    _left_arm.rotationZ = (stage.mouseY - stage.stageHeight/2) / 4;
    _right_arm.rotationZ = -(stage.mouseY - stage.stageHeight/2) / 4;

    _left_leg.rotationZ = (stage.mouseY - stage.stageHeight) / 6;
    _right_leg.rotationZ = -(stage.mouseY - stage.stageHeight) / 6;
  }

  super._onEnterFrame(ev);
}
```

The first thing we do here is to confirm the model has loaded by checking the contents of the `_skeleton` variable. Because the process of loading a file is not instantaneous, we need to make sure our COLLADA model exists in the scene before we start trying to move its limbs. If we don't, we will get a runtime error, because we are trying to set rotation properties on variables that have `null` values. Once the `_skeleton` variable returns something other than `null`, we know that everything is ready for animating.

Inside the `if()` statement in the preceding code, we update the `rotationZ` property of each of our extracted bone objects relative to the coordinates of the mouse cursor. The rotation values for the left and right arms are calculated by measuring the offset of the y coordinate of the mouse from the center of the stage. This means that when the cursor is halfway up the height of the stage, the rotation of the arms will be 0, as they appear in the unanimated model in Figure 9-8. Because the stage is 600 pixels high, the maximum and minimum values here would be 300 and –300 respectively, which is quite a large range for a rotation in degrees. To scale the result to something more suitable, we divide by 4 to give us a range of

75 to –75 degrees. To mirror the rotation effect for the left and right arms, the rotation on the left arm is applied in a positive direction, and the rotation on the right arm is applied in a negative direction.

The same rotation effect is applied to the legs, but this time, we calculate the rotation value using the offset of the y coordinate of the mouse from the bottom of the stage rather than the center. Using this offset means that when the cursor is around the bottom edge of the Flash movie, the rotation of the legs will be 0, as they appear in the unanimated model in Figure 9-8. In this case, the resulting offset value is divided by 6 before setting the `rotationZ` property, simply because we don't want the legs rotating as far as the arms.

Recompiling the code, we can move the mouse up and down to see the Jumping Jack skeleton wildly flapping his arms and legs. A still of the output can be seen in Figure 9-9, taken with the mouse position nearly two-thirds of the way up the stage window.

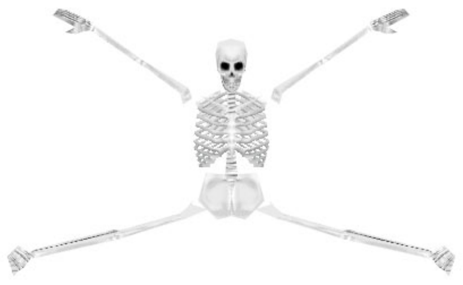


Figure 9-9. The skeleton.dae model loaded in the AnimatingColladaBones example, animated by rotating his arms and legs on mouse input

Controlling a rigged model in this way has virtually limitless potential and is possibly a topic for an entire book in itself! We have only really scratched the surface here— everything from multiple animation merges to ragdoll interactions can be accomplished using programmatically controlled bones.

Summary

In this chapter, we have shown you how to apply various 3D animation techniques to Away3D content, including how the familiar tweening methods used by tweening libraries can be applied to 3D objects in a scene. We have also covered how externally created animation can be imported for use in Away3D. With this knowledge, you should now be ready to create some beautifully animated 3D productions in Flash!

Here are some key ideas to take with you from this chapter.

- The majority of animations in Away3D are set up in the same way as general code-based animation in ActionScript. It is a good idea to have your tweening library of choice to hand.
    - The `PathAnimator` class allows any 3D object to be tweened along a predefined `Path` object in 3D space.
- Externally created mesh animations can be imported to Away3D from MD2 or COLLADA files.

    - Loading an animated model in Away3D uses the same approach shown in Chapter 4 for loading a file without animation. Animation data can be accessed and triggered once the model has been loaded from the `animationLibrary` property.

- Rigging a model with bones can be used as a way to create dynamic character animations and more.

    - Animation rigs for use in Away3D are created using an external 3D software package and then imported using the COLLADA file format.
    - The `getBoneByName()` method invoked on the loaded 3D container of a COLLADA file can be used to retrieve bone object references in order to control the movement of an animation rig programmatically.

In the next chapter, we will take a look at a variety of optimization techniques and utility classes available in Away3D that can help with some of the trickier aspects of 3D content generation and management.

Chapter 10

Optimizing Tips and Tricks

At this point, you should have a solid understanding of the Away3D API and are hopefully already building your next 3D masterpiece! However, as is the case with many web formats, Flash comes with a fairly harsh limitation on processing power. 3D graphics can be more of a processor drain than most, so getting the best possible performance out of the Flash Player when using Away3D is of key importance.

In this chapter, we cover a variety of techniques that can help optimize your Away3D application. Whether you're creating a game, web site, or widget, aiming to achieve the best possible output with the resources at your disposal is always a good idea! We will begin by looking at ways you can optimize your geometry in Away3D, where the majority of savings come from reducing the number of polygons rendered per frame. Next, we will take a look at material optimization, which centers around the steps you can take to produce better visual results with shading and texturing, at less cost to the processor. Finally, we will look at some general tips and advice on ways of using the Away3D framework to ease the complexity of day-to-day 3D tasks.

Preparing the chapter base class

To provide a basic viewing mechanism for the examples created in this chapter, we set up a base class that can be used as our starting point for subsequent document class files.

```
package flash3dbook.ch10
{
  import away3d.cameras.*;
  import away3d.containers.*;

  import flash.display.*;
  import flash.events.*;

  public class Chapter10SampleBase extends Sprite
  {
    protected var _view : View3D;
    protected var _camera : HoverCamera3D;

    public function Chapter10SampleBase()
    {
      _createView();
      _createScene();
    }

    protected function _createView() : void
    {
      _camera = new HoverCamera3D();
      _camera.distance = 1000;
      _camera.tiltAngle = 10;
      _camera.panAngle = 180;

      _view = new View3D();
      _view.x = 400;
      _view.y = 300;
      _view.camera = _camera;

      addChild(_view);
      addEventListener(Event.ENTER_FRAME, _onEnterFrame);
    }

    protected function _createScene() : void
    {
    }

    protected function _onEnterFrame(ev : Event) : void
    {
      _camera.panAngle -= (stage.mouseX - stage.stageWidth/2) / 100;
      _camera.hover();

      _view.render();
    }
  }
}
```

In the preceding code, the `_createView()` method instantiates our view and camera objects, which are held in the global class properties `_view` and `_camera` respectively. The camera uses an instance of the `HoverCamera3D` class to enable easy navigation around the scene. The view is rendered using the `_onEnterFrame()` method, which is setup as an `ENTER_FRAME` event handler at the end of the `_createView()` method. The `_createScene()` method here is written as an empty stub so that it can be overridden in the example class definitions to add custom content to the scene.

Optimizing geometry

The biggest limitation for any 3D engine is the amount of polygons per second that can be realistically rendered to screen. This amount relies on the processing power available and the graphics architecture being used to make all the necessary drawing commands. With the Flash Player, we are restricted to what's known as a **software rendering** architecture for the graphics. It is a simultaneous blessing and curse for Flash users, for while software rendering is a highly compatible solution (and one of the main reasons for Flash's ubiquity), it also imposes comparatively high demands on the CPU of a home computer.

When it comes to raw 3D rendering speeds, software rendering is simply not able to compete with the more specialized **hardware rendering** approach used in the majority of 3D console and desktop game engines. This takes advantage of the purpose-built hardware architectures of the graphics processing unit (GPU)—the graphics card in your machine—and can produce equivalent drawing speeds an order of magnitude faster than that achievable on the CPU.

Having said all that, we shouldn't grumble. The trump card Flash holds is its ubiquity, and without a prevalent cross-platform solution for the browser, hardware rendering is only good for those of us with the right hardware! An interesting consideration to make while others lament the restrictive nature of the rendering speed of Flash is that the latest Flash Player 10 now has around the same level of 3D power that was available in an original Sony PlayStation games console. A lot of cool content was created for that platform, so what's stopping us?

Since the act of drawing is the biggest drain on processing power in Flash, we need to consider ways of reducing the number of rendered polygons without sacrificing visual quality. Let's take a look at some of the tricks available to us in Away3D specifically for this purpose.

Using level-of-detail objects

One obvious way of reducing polygon counts in a 3D application is to reduce the detail of your 3D models. Low-polygon modeling is a fundamental requirement for any model rendered in Away3D, but with a bit of careful surgery, it is possible to reduce the number of polygons to a minimum while maintaining the desired detail. However, what if care and efficiency isn't enough? If your application demands several models onscreen at any one time, you may still be hitting a performance wall and need to reduce the detail in your models further still, destroying their carefully constructed surfaces to a blocky mess.

Level-of-detail (LOD) objects can help in this situation. They work on the assumption that several models needn't be close to the camera at the same time, and while a model is far away from the camera (and therefore smaller on screen), it can be swapped for a simplified mesh representation rather than unnecessarily retaining the detail of its more complex close-up representation. On their own, low-polygon models betray their angular surfaces when viewed close up, while high-polygon models suck unnecessary processing power when viewed at a distance. But if we were to swap a model representation between a low-polygon version and a high-polygon version at the appropriate moment, we could potentially avoid these disadvantages and get the best of both worlds.

The `LODObject` class in Away3D provides just such a service by acting as a 3D container that automatically adjusts its visibility depending on its distance from the camera. More specifically, the container is given a perspective scale range relating to the absolute scale of the container on screen, outside of which the `LODObject` is invisible. Using this capability allows you to automatically switch between any number of models for a single 3D representation in a scene by ensuring a model of appropriate complexity is always visible.

Let's create an example that uses the `LODObject` class to switch between some dummy sphere primitives of varying complexity by extending the base class and overriding the `_createScene()` method with the following code:

```
package flash3dbook.ch10
{
  import away3d.containers.*;
  import away3d.primitives.*;

  import flash.events.*;

  [SWF(width="800", height="600")]
  public class LODObjectTest extends Chapter10SampleBase
  {
    public function  LODObjectTest()
    {
      super();

      stage.addEventListener(MouseEvent.MOUSE_MOVE, _onMouseMove);
    }

    private function _onMouseMove(event : MouseEvent) : void
    {
      _camera.distance = event.stageY*5;
    }

    protected override function _createScene() : void
    {
    }
  }
}
```

We start in the class constructor by adding an event listener for the `MOUSE_MOVE` event that triggers the handler method `_onMouseMove()`, controlling the `z` property of the camera with the y position of the mouse over the stage. We then override the `_createScene()` method with an empty stub ready for some custom code. Let's begin by filling out this method, adding the following code to create our dummy spheres:

```
var s0:Sphere = new Sphere();
s0.radius = 200;
s0.segmentsW = 4;
s0.segmentsH = 4;

var s1:Sphere = new Sphere();
s1.radius = 200;
s1.segmentsW = 8;
s1.segmentsH = 6;

var s2:Sphere = new Sphere();
```

```
s2.radius = 200;
s2.segmentsW = 16;
s2.segmentsH = 10;

var s3:Sphere = new Sphere();
s3.radius = 200;
s3.segmentsW = 32;
s3.segmentsH = 18;
```

In a real-life scenario, the sphere primitives created here would be replaced with imported models. The majority of 3D modeling applications provide the capability to reduce the number of faces in a model while keeping the general shape of the mesh intact. It would be a simple matter to start at your desired maximum level of detail and export your model to an Away3D compatible format, reduce the number of faces, and export again until you have a sufficient number of models that cover the levels of detail you require. In the preceding code, this arrangement is simulated by creating a distant version of our sphere with a minimal amount of faces and gradually increasing the polygon count of each successive sphere until we reach our final close-up version, which requires a significantly greater number of faces to maintain a spherical appearance.

To add these objects to the scene, we first need to create our level-of-detail containers by adding the following code to the end of the `_createScene()` method:

```
var lod0:LODObject = new LODObject(s0);
lod0.minp = 0.0;
lod0.maxp = 0.4;

var lod1:LODObject = new LODObject(s1);
lod1.minp = 0.4;
lod1.maxp = 0.8;

var lod2:LODObject = new LODObject(s2);
lod2.minp = 0.8;
lod2.maxp = 1.2;

var lod3:LODObject = new LODObject(s3);
lod3.minp = 1.2;
lod3.maxp = Infinity;
```

The `LODObject` class is an extension of the `ObjectContainer3D` class, and like any object container, allows child objects to be automatically added on instantiation by receiving a 3D object or group of objects as arguments in its constructor. In the preceding code, we attach each sphere primitive inside its own level of detail object so that we can control its visibility. Two additional properties, `minp` and `maxp`, are then set on each object, defining the minimum and maximum perspective scale range within which the contained sphere objects will be visible. Figure 10-1 illustrates the effect of this arrangement.

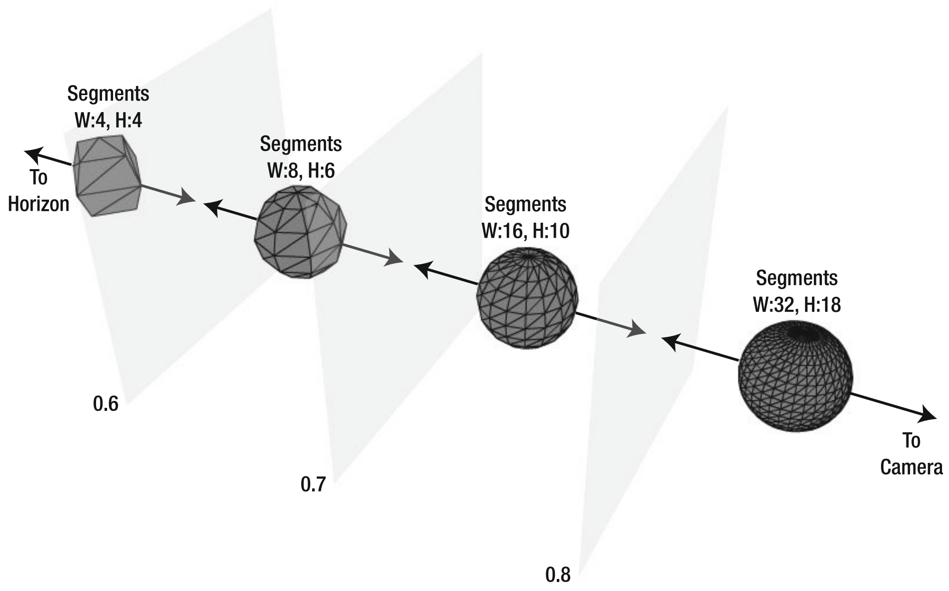


Figure 10-1. In this schematic representation of the setup generated in the `LODObjectTest` example, the planes represent the position from the camera at which the visible sphere primitive switches to one of lesser or greater detail depending on whether the camera moves toward or away from the object.

The `minp` and `maxp` properties of the `LODObject` class have a range of possible values between 0 and infinity, where a value of `0` represents an infinitely far away object (with a perspective scale of 0) and a value of infinity represents an object at the same position as the camera's location (with a perspective scale of infinity).

To create a convincing effect, make sure all objects have the same positions and orientations inside their LOD objects. We also assume that the `minp` and `maxp` ranges do not overlap between levels of detail, ensuring only one object is visible at a time.

To complete our example, we need to add our LOD objects to the scene so that their contents are rendered. To avoid having to individually assign each `LODObject` instance the same position, rotation, and scaling values if we ever want to move our LOD setup, it is good practice to group all objects in an `ObjectContainer3D` instance by adding the following code to the end of the `_createScene()` method:

```
var _lodContainer : ObjectContainer3D;
_lodContainer = new ObjectContainer3D(lod0, lod1, lod2, lod3);
_view.scene.addChild(_lodContainer);
```

Compile the code, and move the mouse cursor up or down over the stage to adjust the proximity of the camera to our LOD setup. Away3D automatically determines which of the LOD objects to display and which to hide, seamlessly switching them in and out and removing unnecessary polygons from the scene. Figure 10-2 depicts the visible output of each sphere displayed, with the leftmost sphere visible when far away and the rightmost visible when near to the camera.

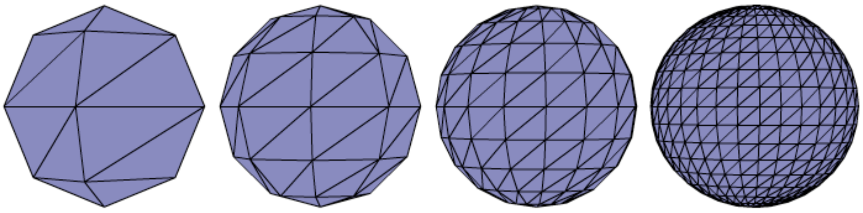


Figure 10-2. These four levels of detail of our sphere representation are shwn in the LODObjectTest example. The leftmost object is rendered at the smallest perspective scale, and the rightmost object at the largest perspective scale.

Culling and clipping polygons and meshes

A complex 3D scene composed of many 3D objects can be a strain on the overall performance of the Flash Player because of the large number of projection calculations and drawing operations required to render all polygons to the view. Culling and clipping are two processes that remove unnecessary polygon processing by optimizing the polygon lists sent to the view: **culling** removes whole polygons (or sets of polygons), and **clipping** subdivides the existing meshes to perform more accurate viewport cropping. Both these techniques are available to use in Away3D, and the following subsections take a look at their implementation in more detail.

Back-face culling

The process of **back-face culling** applies to the rendering of `Face` objects in Away3D and is already used in many of our examples because it is applied to all faces by default. The technique assumes that the triangles in a mesh possess only one visible side (the front side) and only if that side is facing the camera will the triangle be drawn on screen. This culling process is enabled by default because of the number of 3D objects that benefit from its application. As an example, consider the cube primitive shown in Figure 10-3. Viewing it from any position in the scene, we never observe an interior surface, because the shape is a closed volume. If the faces that make up the surface are arranged so that their front sides are all pointing outward (forming the exterior), we can ignore drawing the faces in the view that are being observed from behind, because these are always obscured by faces observed from the front.

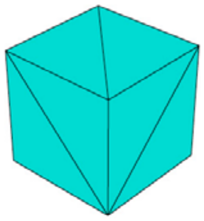


Figure 10-3. A visible face on the surface of a mesh with back-face culling enabled is one aligned with its front side facing the camera. If the faces in a mesh belong to a closed geometry, such as the cube primitive shown here, the faces observed from their front will always obscure the faces observed from their back, and back-face culling has no visual consequence.

Controlling back-face culling is achieved on a per-mesh basis by setting the `bothsides` property of the `Mesh` object as follows:

```
// turn on backface culling (default)
mesh.bothsides = false;

// turn off backface culling
mesh.bothsides = true;
```

There are only a few cases where back-face culling should be disabled:

- When the material of the mesh is semitransparent and the back faces are visible through the front faces
- When the camera is allowed to position itself inside the mesh as well as outside
- When the mesh is an object with a discontinuous surface (such as a plane)

Viewport clipping

In many situations, a scene is only partially visible through the viewport, resulting in significant numbers of polygons in the scene being unnecessary when rendering the view. In these cases, Away3D has a number of **viewport clipping** options represented by the clipping classes located in the `away3d.core.clip` package; these offer different techniques designed to reduce the amount of polygons rendered in any one frame to just those visible through the viewport.

The `View3D` class applies a clipping algorithm to the view by setting its `clipping` property to an instance of the desired clipping class. In Away3D, each clipping class uses the same set of properties to define its clipping boundaries, a topic that we touched on in Chapter 3. The following code snippet is a reminder of how these are set on the clipping object instance for the view:

```
_view.clipping.minX = -400;     // cull or clip if x < -400
_view.clipping.maxX = 400;      // cull or clip if x > 400
_view.clipping.minY = -300;     // cull or clip if y < -300
_view.clipping.maxY = 300;      // cull or clip if y > 300
_view.clipping.minZ = 50;       // culled if z < 50
_view.clipping.maxZ = 2000;     // culled if z > 2000
```

All the boundary properties defined here are optional. By default, the edges of the stage are used as the viewport boundaries for defining the `minX`, `maxX`, `minY`, and `maxY` properties. The z boundaries defined by the `minZ` and `maxZ` properties represent the positions of the **near-field** and **far-field** clipping planes respectively, and are set to `-focus/2` and infinity by default. These values represent the z position of two planes parallel to the projection plane of the camera, clipping polygons in the same manner as the x and y boundaries of the viewport.

Rectangle clipping is the simplest type of viewport clipping available and takes up the least amount of extra processing for a scene. It is represented in Away3D by the `RectangleClipping` class and is set as the default clipping type in the `clipping` property of the `View3D` object.

If all the projected vertices of a polygon fall outside the x and y clipping boundaries of a `RectangleClipping` object, the polygon is removed from the drawing process and prevented from taking up any further processing time. The z boundaries of a `RectangleClipping` object are applied slightly differently, because any polygon vertex projected from behind the camera can produce wildly inaccurate results due to a singularity (division by zero) present in the calculations. To keep us safe from this eventuality, the `RectangleClipping` object will cull a polygon with one or more vertices falling outside the near-field clipping boundary. However, this approach can sometimes be a little too cautious, suffering from rendering artifacts that appear as missing polygons in the visible portion of the scene being viewed.

Near-field clipping is a similar process to rectangular clipping but uses an advanced algorithm that avoids near-field artifacts. It is represented in Away3D by the `NearfieldClipping` class and handles polygons that cross the near-field plane by splitting them along their intersecting lines, passing the polygon fragments that appear on the visible side of the clipping plane for drawing to screen.

Let's create an example to illustrate the advantage of using near-field clipping over rectangle clipping by extending the `Chapter10SampleBase` class with the following code:

```
package flash3dbook.ch10
{
  import away3d.containers.*;
  import away3d.core.clip.*;
  import away3d.primitives.*;

  import flash.events.*;

  [SWF(width="800", height="600")]
  public class ClippingTest extends Chapter10SampleBase
  {
    public function  ClippingTest()
    {
      super();
      _camera.tiltAngle = 0;
      _camera.zoom = 5;
    }

    protected override function _createScene() : void
    {
    }
  }
}
```

We begin filling out the stub override of the `_createScene()` method with the following code to create the interior geometry of a room:

```
var cube : Cube = new Cube();
cube.segmentsW = 4;
cube.segmentsH = 4;
cube.segmentsD = 4;
cube.width = 3000;
cube.height = 1000;
cube.depth = 3000;
cube.bothsides = true;
_view.scene.addChild(cube);
```

Here, a cube primitive is used to form the walls, floor, and ceiling of a room that contains our camera position. Because we are positioned inside the cube's geometry, the `bothsides` property of the cube needs to be set to `true` so that back-face culling doesn't render all of the triangles in the mesh invisible.

Compiling the example at this point will display the output shown in the left-hand image of Figure 10-4. This demonstrates the clipping artifacts seen when using the default `RectangleClipping` class. The example suffers from disappearing triangles in the floor and ceiling of the room, caused by a combination of their close proximity to the camera and large surface area.

The near-field clipping process can assist in these situations, so let's add the following code to the end of the `_createScene()` method to switch our clipping class:

```
_view.clipping = new NearfieldClipping();
```

Recompiling the `ClippingTest` example displays the output shown in the right-hand image of Figure 10-4. The missing triangle artifacts have gone, and moving around the room by moving the mouse cursor over the stage demonstrates that the correction is consistent.

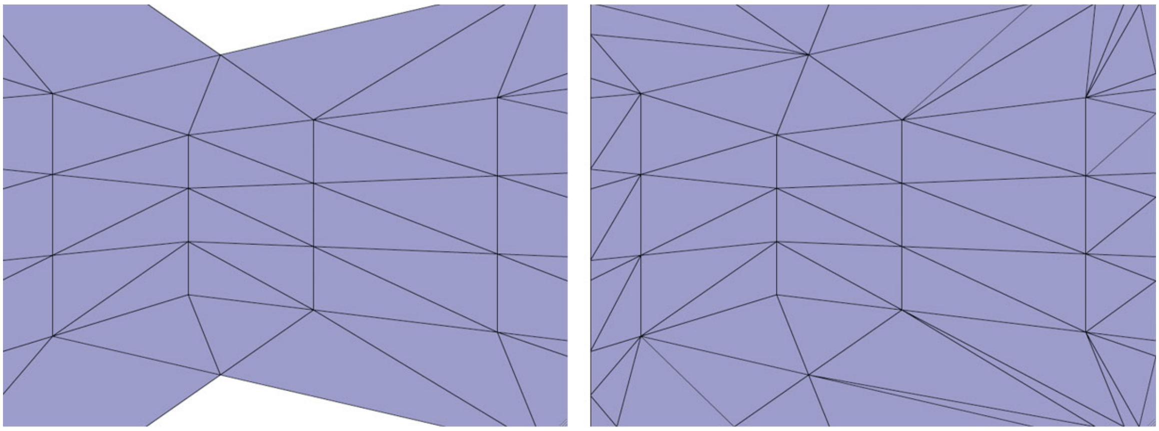


Figure 10-4. Two types of clipping class applied to the view of our ClippingTest example: On the left, the artifacts caused by the default RectangleClipping class are visible; on the right the same scene is displayed using the NearfieldClipping class with no clipping artifacts.

As a final comparison, let's adjust our code again to apply yet another type of clipping object by replacing the previous line of code added to the `_createScene()` method with the following:

```
_view.clipping = new FrustumClipping();
```

Frustum clipping is similar to near-field clipping in the way it handles near and far-field planes but differs in the way it handles viewport clipping for the x and y boundary properties. In this case, polygons can intersect with up to a total of six different clipping planes that represent the precise area in the scene that is visible in the view, known as the **frustum**. Each plane handles itself in the same way, splitting intersecting polygons along their intersecting lines and passing on the polygon fragments that lie on the visible side of the clipping plane for drawing to screen.

Recompiling the example will display a similar output to what's shown in Figure 10-4 for near-field clipping, only, in this case, the triangles around the edges of the view appear dynamically tessellated with the viewport boundaries. This approach has little aesthetic advantage but can improve processing speeds in certain situations where a small portion of a larger scene is rendered thanks to the frustum removing projection calculations as well as drawing calculations for polygons excluded by the clipping planes.

Object culling

All culling techniques we've discussed so far are tested on a per-polygon basis. For 3D objects that are completely out of view, it would be much faster if we could discard the entire object with a single calculation. This is a process known as **object culling** and is possible in Away3D with the `objectCulling` property present in all clipping classes. It is disabled by default in the standard `RectangleClipping` class but can be enabled by setting its value to `true`, as illustrated in the following code snippet:

```
_view.clipping.objectCulling = true;
```

The idea behind object culling is to use a fast method of checking whether a 3D object in the scene is at all visible in the view. If it is determined to be out of sight, no further projecting or drawing operations need be executed for the object. This early out speeds up the rendering process by reducing the amount of overall work done by the renderer. To keep this test as simple as possible, we approximate the 3D extent of an object in Away3D using its bounding radius, a value calculated automatically and accessed from the `boundingRadius` property available on any 3D object. This value represents the radius of a sphere centered around the object's origin that encloses the entire contents of its geometry.

The same frustum calculations used for the `FrustumClipping` class are used to check whether the bounding radius of a 3D object intersects with the frustum of the view, and the contents of the scene are classified into objects that are either contained by the frustum, intersecting the frustum, or outside the frustum. Those inside are considered a trivial case and continue on to be rendered; those outside are discarded and removed from the remainder of the render loop; and those that intersect a frustum plane are flagged for further processing depending on whether the 3D object in question is a container or mesh object. Eventually, all objects in the scene are classified, and the scene is drawn.

Because near-field clipping and frustum clipping both use frustum calculations, object culling is enabled by default in both the `NearFieldClipping` and `FrustumClipping` classes. The `RectangleClipping` class has an `objectCulling` property that defaults to `false`, because in this case the process adds extra frustum calculations that can have an impact on the amount of overall processing. In many cases, it would be considered overkill to run a check for object culling on a scene, especially in scenarios where all 3D objects are visible in the view at all times.

Manual culling

In certain scenarios, despite all the automated culling options that Away3D offers, it can still be preferential to roll up your sleeves and perform some culling yourself. If an object is attached in the scene graph but is not required to be rendered, simply setting its `visible` property to `false` will remove it from the render loop, as demonstrated in the following code snippet:

```
_myObject3D.visible = false;
```

This approach to culling uses your own external rulings to control of the visibility of objects, which can sometimes be more beneficial to optimization than any general-purpose culling techniques.

Overall, the processes of culling and clipping are always present in an Away3D view, with the default settings catering for the generic case. The potential processing reductions using the various extended options described here rely heavily on the contents of the scene and the camera's orientation within it. It is often a good idea to experiment and see what gives the best results.

Using models effectively

A common scenario when designing a 3D project involves the acquisition and import of complex model data. Maybe the model is created from scratch, or maybe it is downloaded from a paid-for resource. Either way, it is worth being mindful of the restrictions imposed by the software renderer of the Flash Player and how to make the most of these restrictions at the 3D design stage.

Polygon counts

A suitable polygon count to aim for when designing a scene can be set at around 4,000 triangles. This is an upper limit of what is expected to achieve a reasonably smooth frame rate in Flash, as 4,000 polygon redraws per-frame adds up to 120,000 redraws per second for a Flash movie running at 30 frames-per-second—that is a high number of redrawing operations for Flash to handle! The recent 3D features added in the Flash 10 Player are helping to push the upper limit higher, but as demand for 3D increases and the range of machines on which quality results are required forever widens, it is best to consider similar figures for the majority of projects.

With that in mind, after having downloaded your perfect-looking dragon model containing 180,000 polygons, you will still need to perform a fair bit of reduction work in a 3D modeling package before it can be used in your Away3D project! This process requires more than a little skill, as generic polygon-reducing tools often leave a lot of residual tidying up to do on the resulting 3D mesh. Areas that demand specific attention include polygons that are never seen by the renderer, high-detail areas such as those for the hands and feet of an avatar, and surfaces that overlap extremely close to one another.

Intersecting polygons

In Away3D, every polygon visible in the view requires sorting in order of Z depth before being drawn to screen, in a process known as Z sorting (covered in more detail in Chapter 3). The default rendering option uses a simple algorithm to sort the rendered order of polygons, calculating a polygon's Z depth value based on the average `z` value of all projected vertices in the polygon. This process is prone to error when dealing with certain geometric scenarios, the most common being when two polygons intersect.

Away3D has some advanced sorting options to deal with such problems, but the extra calculations in their algorithms make them too slow for anything but the simplest geometry. It is usually more effective to manually correct any problem areas in your 3D modeling package before importing your models to Flash, so the default sorting option in Away3D can be used and the best render speeds attained.

As an example, consider the geometry represented in the image on the left in Figure 10-5. Two triangles are shown with a clear intersecting axis that will cause trouble with the default sorting option in Away3D. In the 3D modeler's preview window, polygons are rendered with a pixel-based sorting algorithm performed on the GPU, and our two triangles appear as they should. In Away3D, the triangles will be rendered sequentially in reverse order of Z depth, which gives the option of either the left or right triangle appearing in front. This simplified sorting method will clearly be unable to represent the true nature of our geometry. When rendering in Away3D, such arrangements are usually identifiable as flickering areas where the resolved order of Z depth keeps swapping between problem triangles in a process known as **Z fighting**. The solution is to subdivide one or both of the triangles at the modeling stage into a series of smaller triangles that do not intersect. The correct sorting order then becomes something solvable with the basic sorting option in Away3D. The image on the right in Figure 10-5 depicts such a subdividing solution.

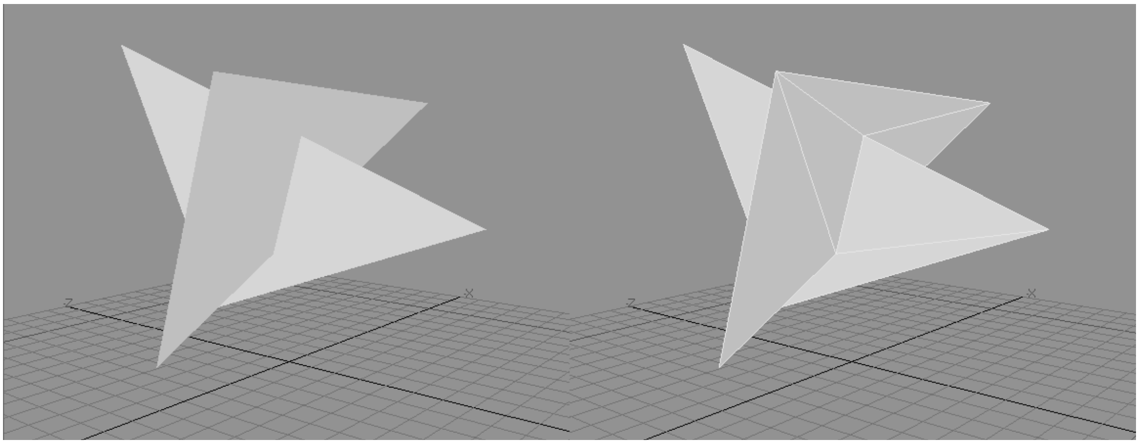


Figure 10-5. On the left, two triangles in a 3D modeling package are arranged so that they intersect. This arrangement is impossible to resolve correctly with the default sorting option in Away3D. On the right, the same geometry has been subdivided to avoid any unresolvable intersections between triangles.

The example here represents a simple occurrence of an intersection problem. In practice, these problems occur in complex model meshes on a regular basis, thanks to the majority of 3D engines (for which the models are designed) having access to hardware-based per-pixel Z sorting that can easily handle such configurations. Any models that exhibit the telltale Z fighting flicker when rendered in Away3D will most likely have some sort of polygonal intersection occurring in their geometry, and subdividing the problem areas in the manner described here is one of the most effective solutions.

Double-sided geometry

In certain cases, it is desirable to render a mesh that has both sides of its geometry visible. As mentioned in the previous section on culling and clipping, setting the `bothsides` property of a mesh object to `true` will achieve this in Away3D by disabling the back-face culling algorithm of the renderer. However, this solution doesn't always produce satisfactory results; there are occasional rendering issues relating to sorting or performance. Let's take a look at an example of each problem, and the potential solutions available to us at the modeling stage.

Sorting problems tend to occur when the `bothsides` property of a mesh is set to `true` on very simple geometry. Figure 10-6 illustrates a typical example with a model of an open-ended box. On the left-hand side, the model is displayed correctly in the modeling package before export; on the right hand side, the same model is displayed after being imported in Away3D. Because the faces of the box are made up of pairs of large triangles, sorting artifacts start to appear in our resulting output in Flash.

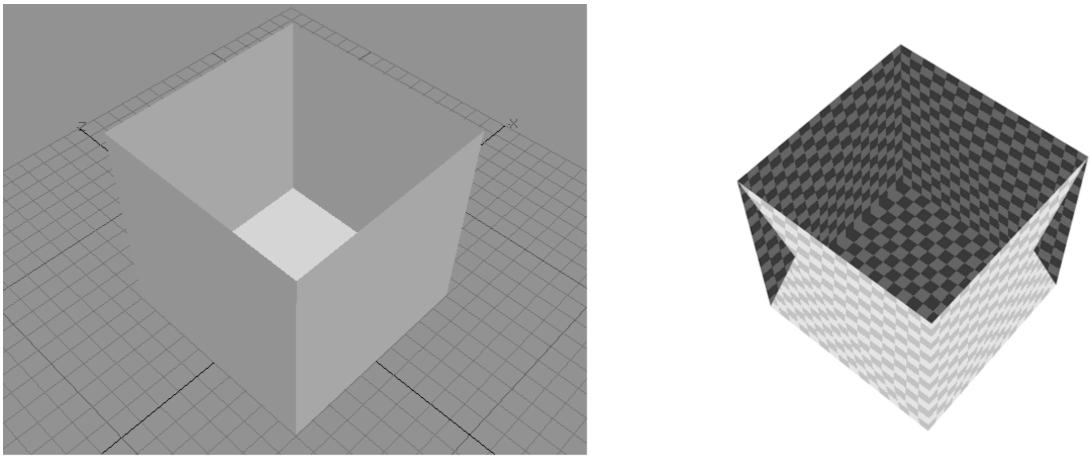


Figure 10-6. A simple open-ended box creates sorting artifacts when rendered in Away3D with the bothsides property set to true. On the left, the box is shown as it appears in the modeller before export; on the right is the resulting imported geometry in Away3D with sorting artifacts visible.

These artifacts are once again caused by the default simplistic sorting method we are compelled to use in Away3D if we want to keep an optimum performance level. One simple solution to the problem is demonstrated in Figure 10-7. Instead of relying on the `bothsides` property, we create the box as a closed geometric shape with separate polygons for its inner and outer sides. On the left, Figure 10-7 shows the geometry in the modeler with some thickness added to its walls, and on the right, the rendered result of this approach in Away3D. Because the inner and outer faces have a small separation in depth, the sorting algorithm of the renderer is not so easily confused, and sorting artifacts are removed.

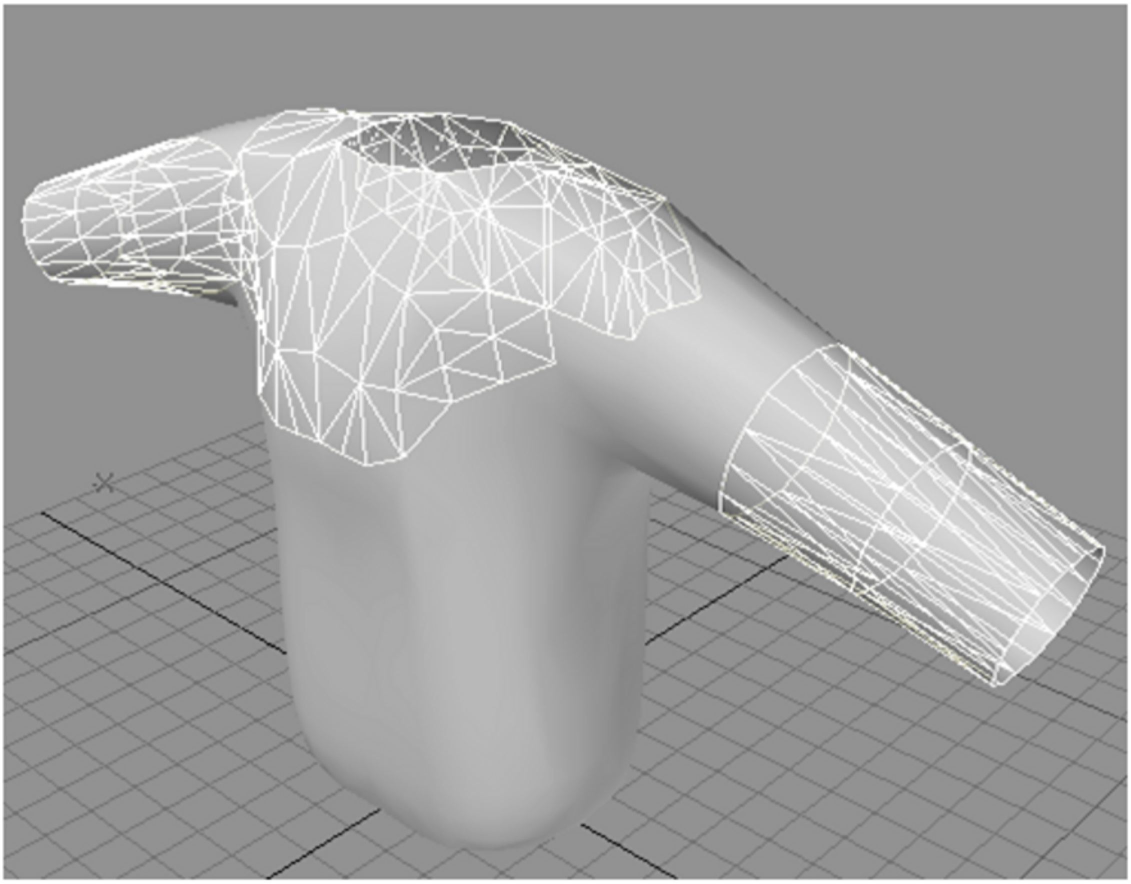


Figure 10-7. The previous sorting artifacts in Figure 10-6 have been removed by modelling the geometry again, with additional faces representing the internal surfaces of the box. On the left is the new geometry in the modelling package before export, and on the right, the resulting model displayed correctly in Away3D.

If the solution in Figure 10-7 isn't suitable for the 3D content you require, it is possible to use one of the more advanced sorting options (thanks to the relatively low polygon count of our example geometry) to render our original box without sorting artifacts. The `renderer` object of an Away3D scene controls the Z-sorting options applied to the polygons in the scene and is set using the `renderer` property of the view object. Various options exist as statically typed variables on the `Renderer` class in the `away3d.core.render` package, including `BASIC` (the default setting), `CORRECT_Z_ORDER` (for resolving problematic triangles that aren't intersecting), and

`INTERSECTING_OBJECTS` (for resolving problematic triangles that are intersecting). In the case of our original box geometry in Figure 10-6, the `CORRECT_Z_ORDER` rendering option should suffice and is applied by setting the `renderer` property of the view to `Renderer.CORRECT_Z_ORDER`.

A second problem that arises with the use of the `bothsides` property relates to the extra processing involved at the drawing stage of rendering. Back-face culling is an easy way of reducing polygon counts, as any polygons that face away from the camera are removed at an early stage of the render loop. Without these savings, the number of polygons drawn to screen will approximately double, causing a significant drop in performance with complex models of 2,000 or more triangles.

As an example, consider a model of a t-shirt that takes the shape of an invisible wearer. The areas around the neck and sleeves are hollow, requiring the backs of faces to be visible. If we were to set the `bothsides` property to `true` after the model was imported to Away3D, we would achieve the desired effect but the renderer would have twice as many polygons to draw in a single frame, noticeably affecting the overall frame rate of the application.

An alternative approach is possible at the design stage, as illustrated in Figure 10-8. Here, we have created some duplicate faces around the areas of the t-shirt that are visible on the inside (the white triangles), reversing their face normals so their perceived front sides are facing in. Combined with the original faces definitions, this arrangement allows us to selectively determine which faces are rendered from their reverse side, and which remain invisible. The vertices used in the extra face definitions are shared among the existing faces, so no extra projection information is required. When this model is imported into Away3D, the `bothsides` property can remain set to `false`, and the extra faces with the reversed face normals take care of rendering the inside of the t-shirt around the neck and sleeve areas, keeping the total number of rendered polygons to a minimum.

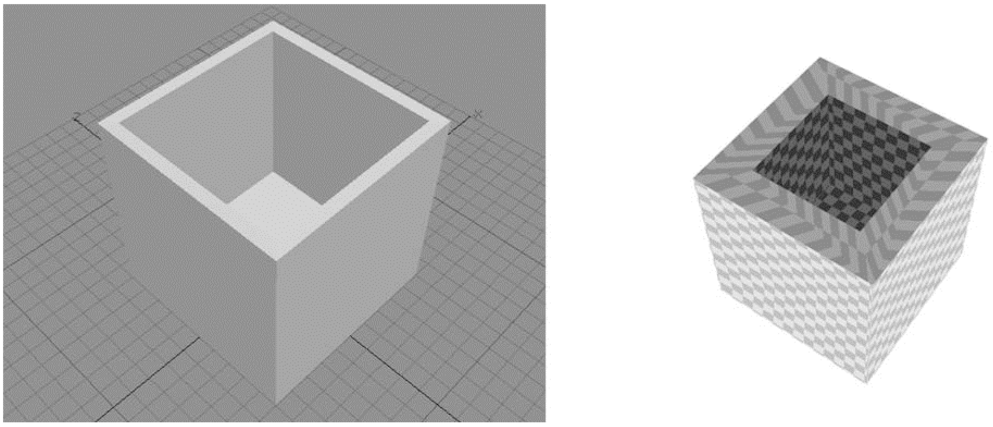


Figure 10-8. This t-shirt model is modifed to selectively render faces on the inside of its geometry. The white triangles are duplicated from the original mesh with reversed normal vectors, making them visible when the front facing triangles are invisible.

Optimizing materials

Materials in Away3D vary wildly in complexity and style, but you can take many optimizing approaches to minimize the memory and processing power they consume. The more complex materials such as those using textures or lighting can be a huge drain on resources if not used efficiently, so it is always worth taking some time to ensure against being needlessly wasteful with materials in your Away3D projects. This section takes a look at how you can avoid some of the common pitfalls to maximize the quality of your materials with the runtime resources at your disposal in the Flash Player.

Optimizing shading

When exploring any 3D Flash engine, some of the most exciting features to discover involve dynamic real-time shading. The practical aspects of shading materials in Away3D are explored in more detail in Chapter 5. By using shading materials, you can build scenes that contain multiple light sources, with controls over various shading options similar to the ones seen in non-Flash engines. However, when dynamic lighting is used to any great extent in Flash, a performance limitation is quickly reached. Thanks to the software renderer of the current Flash Player, it is almost impossible to completely shade a complex scene in real time. When shading is required, we currently have to make a choice between static and dynamic shading for individual objects, with a limit placed on the amount dynamic shading. Let’s take a look at how and when we can apply static shading to objects in Away3D.

Static shading

One option commonly seen for static shading is an approach known as **texture baking**. This process is applied to the texture of a material before the image is imported, multiplying the bitmap data in the texture with a static light map. Because the shading step is performed outside the Flash Player, a shading effect can be achieved with a standard bitmap material in Away3D at no extra processing cost. The technique is best used on background meshes and objects that don’t move relative to their perceived light source.

Professional 3D programs such as Maya, LightWave, Blender, and 3ds Max have a plethora of lighting options for texture baking, allowing you to achieve the precise look you are after before exporting the static texture image. Alternatively, a free program called Prefab3D can be downloaded from the Adobe AIR Marketplace (or direct from `http://www.closier.nl/prefab/`). This application is a 3D tool made for preprocessing textures and other 3D effects and can integrate seamlessly with any Away3D workflow (the program itself is actually written in Away3D!). As well as performing texture baking with multiple light sources, Prefab3D handles advanced shading processes, such as shadow casting, with ease and offers the option to create normal maps and texture maps (diffuse or specular) that can be used with dynamic shading materials.

For 3D objects that occasionally need to update their light maps, it is sometimes appropriate to use a static shading technique known as **surface caching** to generate a static output of a shading material at runtime, with the option of updating the light map on an occasional basis. Surface caching is an option available on the majority of bitmap shading materials in Away3D (including `PhongBitmapMaterial` and `Dot3BitmapMaterial`) and is activated by setting the `surfaceCache` property of the material to `true`. Materials with surface caching activated will appear very similar to uncached materials but will render almost as fast as regular bitmap materials. Because the material will update its cached texture when the light map changes, it is not wise to apply surface caching to a shading material that requires updating on every frame. Light maps are generally affected by the position of the light source relative to the position of the object. If specular shading is enabled, a light map will also be affected by the position of the camera relative to the position of the object.

Figure 10-9 displays four representations of the same model with four different shading options applied. The `No shading` and `Baked` representations use a standard bitmap material but produce very different results, nicely illustrating the potential enhancement texture baking can offer. The `Away3D Dot3` and `Away3D Phong` representations use a DOT3 bitmap material and a phong bitmap material respectively. Both produce dynamic smooth shading over the surface of a model, and both have the option of surface caching to increase the overall performance of the scene when their light maps are static.

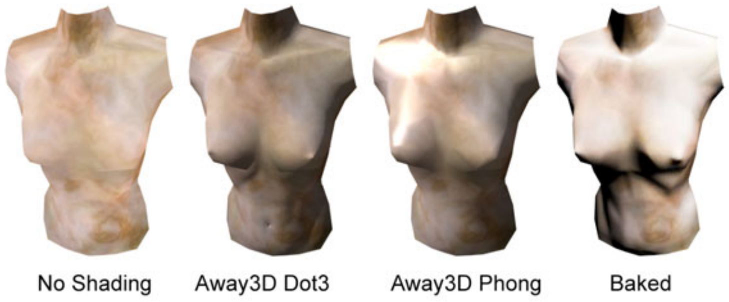


Figure 10-9. This model of a bust is displayed with various shading material options. No shading and Baked are examples of permenant static materials, while Away3D Dot3 and Away3D Phong are examples of dynamic materials that can be temporarily flattened to static mateirals using surface caching.

Normal map images

Of all the dynamic shading options available in Away3D, DOT3 materials are one of most efficient and versatile shading material types on offer. However, in order to function, they require a normal map image—something that can be difficult to generate without access to professional modeling software. The `NormalMapGenerator` class located in the `away3d.materials.utils` package offers a solution to this problem by providing a simple method of normal map generation at runtime using any 3D mesh as input, as long as the UV data in the supplied geometry contains no overlapping UV coordinates (these are discussed in more detail in Chapter 5).

To demonstrate how the `NormalMapGenerator` class is used, let's create a new example by extending the `Chapter10SampleBase` class with the following document class definition:

```
package flash3dbook.ch10
{
  import away3d.core.base.*;
  import away3d.core.math.*;
  import away3d.core.utils.*;
  import away3d.events.*;
  import away3d.lights.*;
  import away3d.loaders.*;
  import away3d.materials.*;
  import away3d.materials.utils.*;

  import flash3dbook.ch10.models.*;
  [SWF(width="800", height="600")]
  public class UsingNormalMapGenerator extends Chapter10SampleBase
  {
    [Embed(source=”../../assets/ch10/headtexture.jpg”)]
    private var HeadTexture : Class;

    private var _head : ObjectContainer3D;
    private var _generator : NormalMapGenerator;
    private var _light : DirectionalLight3D;

    public function UsingNormalMapGenerator()
    {
      super();
}

    private function _onTraceProgress(event:TraceEvent):void {
      trace("Processing normal map : " + event.percent.toFixed(0) + "% complete");
    }

    private function _onTraceComplete(event:TraceEvent):void {
      trace("Completed normal map");
    }

    protected override function _createScene() : void
    {
    }
  }
}
```

To generate a normal map image, we must first import a model to use as the source geometry. In its default mode, the `NormalMapGenerator` class will generate a map that represents the smoothed vector data between vertex normal vectors of a 3D mesh object. The class variable `_head` in the preceding code is defined as a placeholder for the geometry, which we will now create by adding the following code to the empty `_createScene()` method:

```
_head = new HeadModel();
_head.scale(30);
_view.scene.addChild(_head);
```

The `HeadModel` class is an ActionScript model, created with the `AS3Exporter` class using a technique covered in the optimizing external resources section in Chapter 4. Compiling the code at this point will display the output seen in the left-hand image of Figure 10-10.

To use a shaded material on our head model, we set up a new directional light source by adding the following code to the end of the `_createScene()` method:

```
_light = new DirectionalLight3D();
_light.ambient = 0.3;
_light.diffuse = 0.5;
_light.specular = 0.5;
_light.direction = new Number3D(0.5, -0.5, 0);
_view.scene.addLight(_light);
```

Next, we pass the instance of the `HeadModel` class to the constructor of a new `NormalMapGenerator` object, adding the following code to the end of the `_createScene()` method:

```
_generator = new NormalMapGenerator(_head, 512, 512);
_generator.addEventListener(TraceEvent.TRACE_PROGRESS, _onTraceProgress);
_generator.addEventListener(TraceEvent.TRACE_COMPLETE, _onTraceComplete);
_generator.execute();
```

Because the creation of a normal map image is quite a processor-intensive task, the `NormalMapGenerator` class builds the image in short steps over a number of frames. This requires us to add event handlers for `TRACE_PROGRESS` and `TRACE_COMPLETE` events after instantiation of the `NormalMapGenerator` object, in order to wait for the generating process to complete before applying the image to our head model. The handler methods `_onTraceProgress()` and `_onTraceComplete()` already exist in the `UsingNormalMapGenerator` class definition, containing `trace()` statements to monitor progress. As a final step, we apply a new DOT3 material to the head model after receiving the `TRACE_COMPLETE` event from our `NormalMapGenerator` object by adding the following code to the end of the `_onTraceComplete()` handler method:

```
_head.children[0].children[0].material = new
Dot3BitmapMaterial(Cast.bitmap(HeadTexture), _generator.normalMap);
```

Recompiling the `UsingNormalMapGenerator` example will display the output seen in the right-hand image of Figure 10-10, and the actual image generated for the normal map texture is shown in the center image. If you compare this to the `headtexture.jpg` image used for the base texture, you'll see the unwrapped UV coordinates match exactly—a requirement of the DOT3 shading process.

DOT3 shading materials offer additional depth and quality to the graphical output of an Away3D scene, and using the NormalMapGenerator class to create the required normal map images puts their use within easy reach of any designer or developer wishing to take advantage of this type of shading. Rotating the camera with the mouse, you'll notice that the direction of the light is stationary in relation to the head. If, instead, we want to keep the light stationary in relation to the camera, we can add the following code to the end of the class definition:

```
override protected function _onEnterFrame(ev : Event) : void
{
  super._onEnterFrame(ev);

  var angle:Number = (_camera.rotationY)*Math.PI/180;
  _light.direction = new Number3D(-Math.cos(angle), -0.5, Math.sin(angle));
}
```

Recompiling the example will show the head model as before with the light souce adjusting its direction in relation to the camera position.

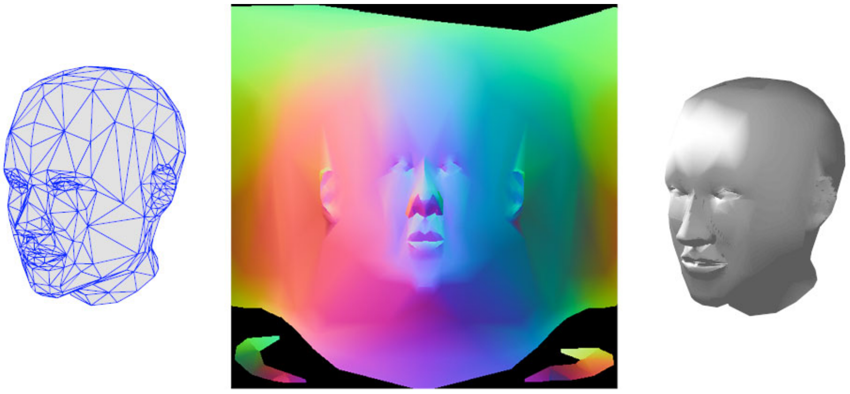


Figure 10-10. These stages of the shading process are encountered in the UsingNormalMapGenerator example. The left-hand image shows the head model before a shading material is applied; the center image shows the image generated from the NormalMapGenerator class, and the right-hand image shows the final shaded head model using the generated normal map with an instance of the Dot3BitmapMaterial class.

Conserving material instances

Like any Flash application, reducing the number of new object instances in an Away3D project will reduce the overall memory consumption and improve performance. The most common way of achieving this is through object pooling and reuse. In Away3D, bitmap material instances are a prime candidate for reuse because of their large memory footprints.

One scenario for optimization occurs when several objects are created that share the same texture image. Let's create a new example to illustrate this setup by extending the `Chapter10SampleBase` class with the following document class definition:

```
package flash3dbook.ch10
{
  import away3d.core.utils.*;
  import away3d.materials.*;
  import away3d.primitives.*;

  [SWF(width="800", height="600")]
  public class BitmapInstanceTest extends Chapter10SampleBase
  {
    [Embed(source=”../../assets/ch10/cubetexture.jpg”)]
    private var CubeTexture : Class;

    public function  BitmapInstanceTest()
    {
      super();
    }
    protected override function _createScene() : void
    {
    }
  }
}
```

Here, we define the global variable CubeTexture containing the embedded image data for our texture and override the _createScene() method with a stub. We then add the following code to the empty _createScene() method, creating 50 randomly positioned cubes with 50 separate instances of a bitmap material and using the Cast utility class to extract the required bitmap data from the CubeTexture class.

```
// create 50 cubes
for (var i : int = 0; i < 50; i++) {
        var cube : Cube = new Cube();

        // assign a random position to the cube
        cube.x = (Math.random()-.5)*1000;
        cube.y = (Math.random()-.5)*1000;
        cube.z = (Math.random()-.5)*1000;

        // assign a new bitmap material to the cube
        cube.material = new BitmapMaterial(Cast.bitmap(CubeTexture));

        _view.scene.addChild(cube);
}
```

Compiling the code will display the scene with no obvious issues, but if you take a look at the memory consumption of the Flash movie (by opening the Task Manager in Windows or the Activity Monitor in OS X) you will notice an abnormally high value caused by each bitmap material generating an internal copy of the bitmap data supplied for its texture. In Away3D bitmap materials are designed for reuse, allowing us to optimize memory consumption by assigning the same material to multiple mesh objects in cases where the same texture is being used. We can therefore rewrite the BitmapInstanceTest example to take advantage of this feature, replacing the contents of the _createScene() method with the following code:

```
var material : BitmapMaterial = new BitmapMaterial(Cast.bitmap(CubeTexture));

// create 50 cubes
for (var i : int = 0; i < 50; i++) {
        var cube : Cube = new Cube();

        // assign a random position to the cube
        cube.x = (Math.random()-.5)*1000;
        cube.y = (Math.random()-.5)*1000;
        cube.z = (Math.random()-.5)*1000;

        // assign the same bitmap material to the cube
        cube.material = material;
        _view.scene.addChild(cube);
}
```

Recompiling the example with display exactly the same output with the memory consumption at a fraction of what it was, thanks to the preceding code using the same bitmap material instance for all 50 primitives. This setup has one restriction in that any adjustment of material properties (such as color and alpha) will globally affect all cube primitives. However, a workaround that allows material properties to be independently set on a small number of individual primitives could involve separate material instances created for this purpose, swapped on specific cubes as and when they are required.

Exploring general best practice techniques

As a final section on optimization, we will now look at a variety of shortcuts available in Away3D that can assist with getting the most out of the engine. Some help simplify development; others help simplify the rendering process.

Switching between 3D coordinate systems

In Chapter 3, we briefly touched on the concept of hierarchical coordinate systems. The framework that underpins the calculations necessary for this system is full of useful properties we can tap into, and this section explains how to take advantage of the framework to extract useful spatial data from a scene.

A single 3D coordinate system is defined by three axes and a point of origin. In Away3D, every object in a scene has its own local coordinate system that works by concatenating the transformation of its parent container with its own transformation. The directions of the three axes depend on the rotation and scaling values applied to the 3D object, and the position of the origin is considered the object's center point. This means that when we rotate, scale, or position an object with its transformation properties, we are essentially updating its coordinate system.

When expressing the vector position of a point in space, there are five main types of coordinate system to consider:

- **Object space**: This is the local coordinate system of a 3D object such as a mesh, equivalent to the coordinate system of a native `DisplayObject` in Flash. The position vectors of vertices inside a mesh are represented in object space.
- **Parent space**: This is the coordinate system of a 3D object relative to its parent container, equivalent to the coordinates of vertices inside a mesh after the transformation of their local coordinate system is applied. The position vectors of objects inside a container are represented in parent space.
- **Scene space (or world space)**: This is the global coordinate system of the scene, equivalent to the coordinate system of the native Stage object in Flash. Describing a 3D object in scene space is similar to converting the coordinates of a native `DisplayObject` from local to global coordinates.

- **Camera space (or view space)**: This is the scene space of a 3D object relative to the camera. The inverted transformation of the camera object is applied to the representation of the object in scene space, giving its position and transformation as viewed by the camera.
- **Screen space**: This is the coordinate system into which the scene contents are projected and then drawn. The projection transformation of the camera's lens object is applied to the representation of the 3D object in camera space, giving its position and transformation after being projected to screen.

A schematic representation of these coordinate systems is shown in Figure 10-11. As you can see, 3D objects that share a scene will also share the same scene space, camera space, and screen space. Because Mesh A is a direct child of the scene, its resulting transformation in scene space is the same as its transformation in parent space.

With the exception of screen space, every coordinate system described here is an orthogonal transformation of the 3D object's local coordinate system relative to another 3D object, be it local (object space), parent (parent space), scene (scene space), or camera (camera space).

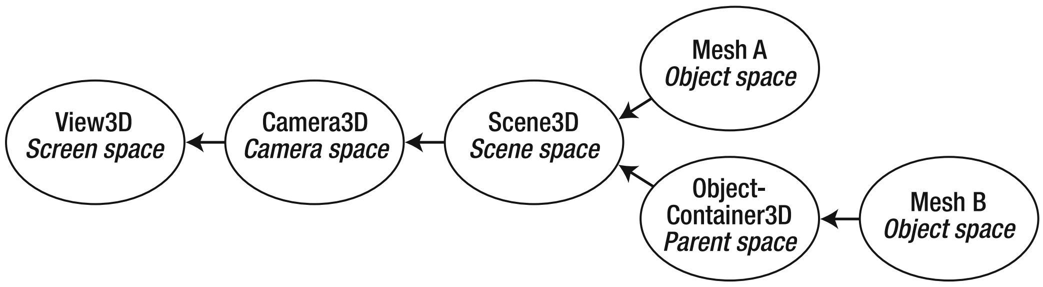


Figure 10-11. Schematic representation of the different coordinate systems available in Away3D, depicting the relationship between systems for a scene containing two 3D mesh objects

Now that you've seen an overview of the various types of coordinate system, let's take a look at how these can be accessed and utilized in an Away3D project. We'll begin by extending the `Chapter10SampleBase` class with the following code to use as a starting point for the subsequent examples in this section:

```
package flash3dbook.ch10
{
  import away3d.containers.*;
  import away3d.core.draw.*;
  import away3d.core.math.*;
  import away3d.materials.*;
  import away3d.primitives.*;

  import flash.display.*;
  import flash.events.*;

  [SWF(width="800", height="600")]
  public class TransformTest extends Chapter10SampleBase
  {
    private var _plane : Plane;
    private var _planeContainer : ObjectContainer3D;
    private var _cube : Cube;
    private var _cubeContainer : ObjectContainer3D;
    private var _marker : Sphere;
    private var _shape : Shape;

    public function TransformTest()
    {
      super();
    }

    protected override function _createScene() : void
    {
    }

    protected override function _onEnterFrame(event : Event) : void
    {
      super._onEnterFrame(event);
    }
  }
}
```

Here, we create some class variables to use as 3D object references for the scene content and override both the `_createScene()` and `_onEnterFrame()` methods to prepare them for some custom code. We begin by adding the following code to the `_createScene()` method to set up a generic scene for our purposes:

```
_plane = new Plane();
_plane.segmentsW = 10;
_plane.segmentsH = 10;
_plane.material = new WireColorMaterial(0x808080);
_plane.width = 200;
_plane.height = 200;
_plane.bothsides = true;
_plane.yUp = false;
_plane.position = new Number3D(-200, 0, 0);

_planeContainer = new ObjectContainer3D(new Trident(100, true));
_planeContainer.x = -200;
_planeContainer.addChild(_plane);
_view.scene.addChild(_planeContainer);

_cube = new Cube();
_cube.material = new WireColorMaterial(0xFFFFFF);
_cube.pivotPoint = new Number3D(200, 0, 0);

_cubeContainer = new ObjectContainer3D(new Trident(100, true));
```

```
_cubeContainer.x = 200;
_cubeContainer.addChild(_cube);
_view.scene.addChild(_cubeContainer);
```

The preceding code creates a simple scene hierarchy with a cube and a plane primitive contained inside their own 3D container objects, which are, in turn, contained inside the scene. As a visual aid, both the 3D container objects have trident primitives added to their children, so that the local coordinate system of the containers can be seen.

Now, we create some movement by adding the following code to the end of the `_onEnterFrame()` method:

```
_cube.rotationY += 5;

_planeContainer.rotationY += 5;
```

Compiling the `TransformTest` example at this point will display a plane and a cube performing identical orbiting motions around two different positions in the scene. The orbiting motion is achieved in the preceding code using the two distinct methods described here:

- For the cube primitive, a rotation is applied around the Y axis of the primitive object. Normally, this would rotate the cube around its center, but in this case, the previous code added to the `_createScene()` method has modified the local position of the cube's center point. By setting the `pivotPoint` property of the cube to (100, 0, 0), its local coordinate system is offset in object space by the given `Number3D` value. This would be equivalent to moving all the vertices in the cube 100 units along the local X axis, resulting in the cube's geometry no longer being centered around the local origin and producing an orbiting motion when it is rotated. The local origin of the cube is then offset to the left by the 3D container for the cube, moving the center of rotation in the scene –200 units along the X axis.
- For the plane primitive, the same rotation is applied to the 3D container of the plane. In the previous code added to the `_createScene()` method, the position of plane is offset in parent space by (–100, 0, 0) using the `position` property of the primitive. Rotating the plane with this setup wouldn't produce the same orbiting motion we see for the cube, because the local coordinate system doesn't contain the applied offset. However, the 3D container of the plane does contain the offset, so it produces the same orbiting effect as the setup for the cube when rotated. In this case, the local origin of the 3D container is offset to the right, moving the center of rotation in the scene 200 units along the X axis.

Converting from object space to scene space

Scene space is useful for performing any kind of global operation on a 3D object. Collision detection is a good example of such a scenario, where the absolute position of an object in a scene is required. If we are dealing with a collection of 3D objects that are all direct children of the scene, the object's position in scene space will match the object's position in parent space. In this case, we can simply use the `x`, `y` and `z` properties for the object's position in scene space, but for a different scene hierarchy, this may not always be the case.

For the position vector of a 3D object, Away3D provides us with the `position` property that returns a `Number3D` value representing the 3D object's position in parent space. The same position is represented in scene space with the `scenePosition` property, calculated automatically from the `position` property when an object is added to the scene graph. We can test this functionality by adding the following code to the `_createScene()` method in our `TransformTest` example:

```
_marker = new Sphere();
_marker.radius = 50;
_marker.material = new WireColorMaterial(0xFFFFFF);
_view.scene.addChild(_marker);
```

Here, we create an instance of a sphere primitive inside the global property `_marker`, to be used as a marker for the scene position of our plane primitive. This instance is added to the scene graph as a direct child of the scene, so its `position` property can be set to represent the scene position of any object we choose. To represent the scene position of the plane primitive, we add the following code to the end of the `_onEnterFrame()` method:

```
_marker.position = _plane.scenePosition;
```

Recompiling the `TransformTest` example will display the output shown in Figure 10-12. The cube and plane primitives animate in the same way as before, and the newly added white marker sphere follows the scene position of the plane primitive exactly.

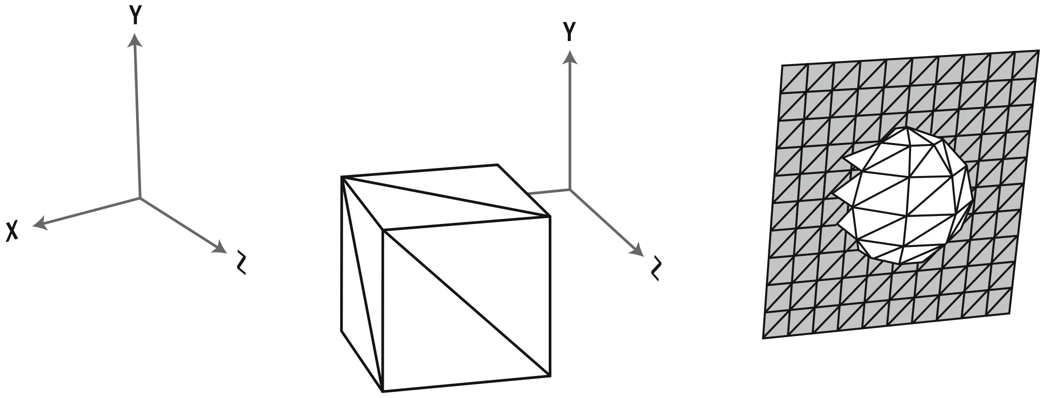


Figure 10-12. A sphere and plane primitive orbiting around opposite points on the X axis of the scene in the TransformTest example, with a white marker sphere following the positoin of the plane in scene space.

It is important to note in the code that the `position` property of `_marker` is updated after all transformations have been performed on the primitive objects and their respective containers. This is a necessary precaution, as the `scenePosition` property calculates its value from the current transformations applied to all objects in its scene hierarchy. If these transformations are altered prior to the next `render()` call, the scene position would need to be reacquired from the `scenePosition` property to remain in sync with the visible scene position of the objects on screen.

Converting from scene space to object space

In the previous example, imagine that our marker object was a child of the cube container in our scene, rather than being a child of the scene itself. To continue following the scene position of the plane with our marker, we would need to take into account the scene position of the cube container as well as the scene position of the plane container. Let's set up such a scenario from the code in our current `TransformTest` example by updating the last line of the `_createScene()` method to the following:

```
_cubeContainer.addChild(_marker);
```

Recompiling the `TransformTest` example will show the effects of ignoring the cube position in this setup. The position of the marker is offset by the position of the `_cubeContainer` object, destroying our visual link between the marker object and the position of the plane primitive.

To fix our example, we need to apply the inverse of the cube container's scene position to the marker position, converting the scene space received from our plane to the object space determined by our cube container. Applying the inverse of a position vector is the same as subtracting it, so we replace the last line of the `_onEnterFrame()` method with the following code:

```
var position : Number3D = new Number3D();
position.sub(_plane.scenePosition, _cubeContainer.scenePosition);
_marker.position = position;
```

Here, a new `Number3D` object is created for calculating the subtraction, and the position is then re-applied to the `position` property of the marker sphere. This is necessary to flag the position of the marker sphere for update, which is done on the setter of the `position` property. Recompiling the example displays the marker once more following the position of the plane exactly.

One point to note about our current setup in the `TransformTest` example is that the marker follows the position of the plane but doesn't follow the rotation. If we want the two objects to be transformed in exactly the same way, we need to use a property similar to `scenePosition` available on all 3D objects called `sceneTransform`. This represents the overall transformation of an object in scene space, calculated from the entire cumulative transformation of a 3D object through its scene hierarchy (including all rotation, scaling, and positioning properties).

We can update the code in our `TransformTest` example to use the `sceneTransform` property in place of `scenePosition`, allowing us to retain all transforming aspects of the objects in our scene. We do this by replacing the last three lines of the `_onEnterFrame()` method with the following code:

```
_marker.sceneTransform.multiply(_cubeContainer.inverseSceneTransform,↩
_plane.sceneTransform);
_marker.transform = _marker.sceneTransform;
```

In this instance, we set the local `transform` property of the marker sphere to the resulting transform of the `sceneTransform` property of the plane already multiplied with the `inverseSceneTransform` property of the cube container. Multiplying transformation properties has the effect of adding together their influences, although in this case, we need to use the inverse of the `sceneTransform` property of the cube container, because we actually want to subtract its influence. The `inverseSceneTransform` property is another automatically calculated property on all 3D objects in an Away3D scene.

As with the `position` property, the `transform` property of the marker sphere needs to be updated via its setter, but in this example, we can use a transformation that already exists (the `sceneTransform` property of the marker sphere) to hold the result of the multiplication. We then reapply the new transform to the `transform` property of the marker sphere. Recompiling the `TransformTest` example displays our marker sphere following both the position and rotation of the plane, as if both objects were children of the same 3D container.

Converting from object space to screen space

Screen space is the coordinate system used to draw the visible representation of a scene to the view and is, therefore, a useful way of overlaying standard 2D content that is linked in some way to the displayed 3D content. Because screen space is used for drawing operations, the x and y coordinates match the native 2D coordinate system of Flash, while the redundant z coordinate represents the perpendicular distance of objects from the viewing plane and is only really useful for calculating distance scaling values.

Let's modify our current `TransformTest` example to demonstrate a practical use of screen space by adding the following lines to the end of the `_createScene()` method:

```
_shape = new Shape();
_shape.graphics.beginFill(0x000000);
_shape.graphics.drawEllipse(-20, -20, 40, 40);
_view.foreground.addChild(_shape);
```

Here, we create a native `Shape` object and add it as a child of the `foreground` property of the view. This property represents a `Sprite` object automatically placed in front of everything else contained within the view's display list, and is best used when we want to add any native `DisplayObject` as an overlay to the rendered contents of the view. The x and y coordinates of the `Shape` object now match the same coordinate system used for the screen space of the view, so we can use `_shape` as a marker for a 3D object's screen position. The camera object contains a handy `screen()` method that returns the screen coordinates of any 3D object in the scene as a `ScreenVertex` object. In this case, let's follow the screen position of the cube primitive by adding the following code to the end of the `_onEnterFrame()` method:

```
var _screenPosition : ScreenVertex = _view.camera.screen(_cube);
_shape.x = _screenPosition.x;
_shape.y = _screenPosition.y;
```

Recompiling the `TransformTest` example displays the black circle of the `Shape` object following the position of our cube primitive on the screen. This is done in the preceding code by setting the `x` and `y` properties of our `Shape` object to the `x` and `y` values of the returned screen vertex to match the resulting position of the cube in screen space.

Changing camera lenses

In a real-world camera, the lens barrel can be interchanged to produce different results in your photographs. A similar configuration process is used in Away3D, with the `lens` property of the camera object allowing the interchange of various lens types to produce a different result in the view.

When the contents of a scene are projected to screen space, the transformation of coordinate systems is calculated via the lens object present in the `lens` property. As with many other Away3D configurations, this object is instantiated from one of a variety of class types available. In this case, lens classes found in the `away3d.cameras.lenses` package can be swapped into the `lens` property of the camera object at any time, each producing a different screen projection. Let's take a look at some of the lens options available,

Traditional perspective projections

In Away3D, the default lens uses a combination of `zoom` and `focus` properties to calculate the projection of the scene to screen space. These properties are discussed in more detail in the cameras section of Chapter 3 and are utilized by the default `ZoomFocusLens` class to produce the effects demonstrated. However, the `ZoomFocusLens` class is the only lens to use these properties in this manner. In all other lenses, `zoom` and `focus` are used as interchangeable multipliers for the overall scale of the view, and it is usually necessary to adjust only one value, such as the `zoom` property.

The `ZoomFocusLens` class is a bit of a legacy projection method from the early days of 3D in Flash. When attempting to match the projection of an Away3D scene with the projection seen in some other 3D application such as a 3D modeling package, you are better off using the standardized `PerspectiveLens` class as your projection method.

Extreme wide-angle projections

For wide-angle lens effects in Away3D, it is possible to reduce the `zoom` property of the camera to increase the field of view of the viewport. However, the standard projection method becomes less and less accurate with wider angles. Extreme wide-angle projections should give the view a spherical look, as if the scene were being reflected in the surface of a glass sphere. This effect is sometimes referred to as a **fish-eye lens** and can be achieved in Away3D using the `SphericalLens` class. We can demonstrate the effect by extending our `Chapter10SampleBase` class with the following document class definition:

```
package flash3dbook.ch10
{
  import away3d.cameras.lenses.*;
  import away3d.materials.*;
  import away3d.primitives.*;

  [SWF(width="800", height="600")]
  public class UsingSphericalLens extends Chapter10SampleBase
  {
    public function UsingSphericalLens()
    {
      super();

      _camera.lens = new SphericalLens();
    }

    protected override function _createScene() : void
    {
    }
  }
}
```

In the constructor method, we define a new `SphericalLens` class instance on the `lens` property of our camera. We also create our usual empty override for the `_createScene()` method, into which we write the following code:

```
var cube : Cube = new Cube();
cube.scale(10);
cube.material = new WireColorMaterial(0xCCCCCC);
_view.scene.addChild(cube);
```

Compiling the example will display a slightly distorted looking cube, and hovering over the stage with the mouse will cause the distortion effect to alter as the cube turns. This is the spherical lens doing its work, but it is hard to see precisely what is going on because the lens projection can only act on individual vertex positions, and a standard cube only has 8 vertices with which to project.

To better visualize the effect, we can add more vertices to our cube by adding the following code to the end of the `_createScene()` method:

```
cube.segmentsW = 10;
cube.segmentsH = 10;
cube.segmentsD = 10;
```

Recompiling the `UsingSphericalLens` example gives us a clearer picture of the projection achieved with the `SphericalLens` object. The geometry out to the edges of the viewport appears further away than the geometry closer to the center of the viewport, and the entire effect distorts straight lines that appear tangential to the center of the view, while preserving straight lines that appear radial to the center of the view. This mimics the expected projection of a real life fish-eye lens and is useful in scenarios that demand such a projection effect.

Isometric projections

So far, we have only considered projections with some form of perspective scaling, producing recognizably 3D views with a central vanishing point. However, it is sometimes desirable to render a 3D scene without perspective scaling, in a manner that preserves size and keeps parallel lines parallel. This type of projection is frequently referred to as **isometric** and is something that should be familiar to any strategy gamer or CAD designer.

In Away3D, an isometric view is achieved using the OrthogonalLens class. Let's create another example to demonstrate its effects with the following document class definition:

```
package flash3dbook.ch10
{
  import away3d.cameras.lenses.*;
  import away3d.primitives.*;
  import away3d.materials.*;
  [SWF(width="800", height="600")]
  public class UsingOrthogonalLens extends Chapter10SampleBase
  {
    public function UsingOrthogonalLens()
    {
      super();
      camera.lens = new OrthogonalLens();
      camera.zoom = 100;
    }

    protected override function _createScene() : void
    {
    }
  }
}
```

Here, we set the `lens` property of our hover camera to a new instance of the OrthogonalLens class and adjust the `zoom` value to something more suitable for our purposes (seeing as our scene will no longer be scaling with distance). In this case, we will create two cubes and a trident primitive in our scene by adding the following code to the empty `_createScene()` method:

```
var cube1 : Cube = new Cube();
cube1.x = -100;
cube1.material = new WireColorMaterial(0xFFFFFF);
_view.scene.addChild(cube1);

var cube2 : Cube = new Cube();
cube2.x = 100;
cube2.material = new WireColorMaterial(0x808080);
_view.scene.addChild(cube2);

_view.scene.addChild(new Trident(200, true));
```

Compiling the example will display the image shown in Figure 10-13. Without any scaling, the two cubes appear identical in size and shape, whatever angle they are viewed from. This form of projection is useful for certain games that are constructed from a series of tiled objects. With this arrangement, it is possible to replicate the rendered output of a single complex object as a 3D sprite that is then tiled across a large area to create many representations of the object, without the need for further rendering. It is also possible to create large scrolling areas that use this type of tiled arrangement, because the rendering costs of a small group of 3D sprites is a lot lower than the same number of complex objects projected with a perspective lens class.

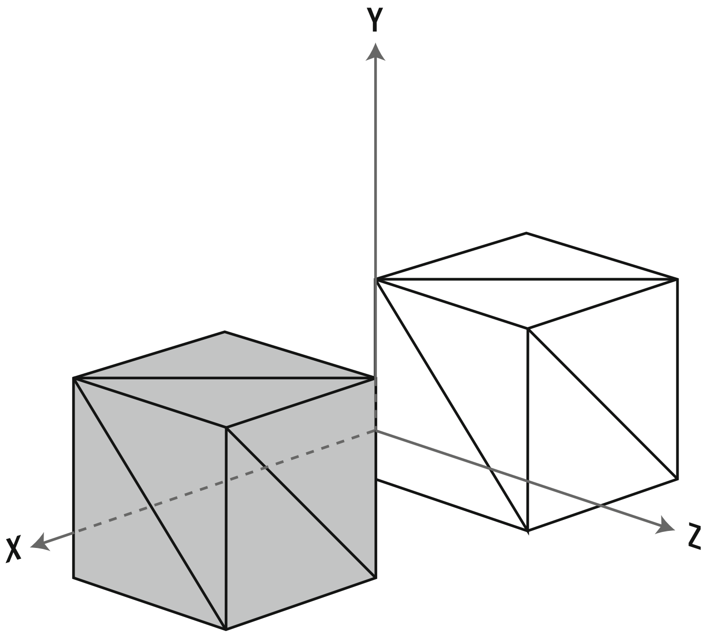


Figure 10-13. Using the OrthogonalLens object in the UsingOrthogonalLens example to produce an isometric projection of the scene, where scale is preserved and parallel lines remain parallel

Summary

In this chapter, we have highlighted a number of optimizing tricks to assist with the performance and visual consistency of your Away3D projects. We have also presented a collection of tips to assist in day-to-day 3D production such as how to create a normal map, how to convert between coordinate systems, and how to apply different projection methods to a scene using the built-in features Away3D has to offer.

The following topics have been covered in this chapter:

- The total number of polygons projected and rendered in any one frame must be kept to a minimum in order to maintain a smooth frame rate.
    - Level-of-detail (LOD) objects in Away3D can reduce the number of rendered polygons in the view by automatically adjusting the detail revealed in an object's geometry depending on its proximity to the camera and size on the screen.
    - Clipping reduces the number of rendered polygons in the view by restricting the rendering area to the defined boundaries of the viewport, with alternative clipping techniques preventing culling artifacts by trading accuracy with overall rendering performance.

    - Object culling can reduce the number of projection calculations carried out for large scenes by using the bounding radius of individual mesh objects to cull them in a single step when out of sight of the viewport.
    - Polygon counts and rendering artifacts can be reduced at the modeling stage by being mindful of sorting restrictions and limiting the use of double-sided materials.
- Materials can hog memory and processing resources, depending on the type and number used in an Away3D scene.
    - Static shading is a way of retaining a shaded look in a scene without the performance overheads created by real-time shading.
    - Bitmap materials need to be shared between models wherever possible to save on memory consumption.
- Various coordinate systems of a 3D object including object space, parent space, scene space, and screen space can be easily accessed using the built-in properties and methods in Away3D.
- Different lens objects are used for different projection methods in Away3D.
    - The `SphericalLens` class produces a fish-eye lens effect for views that require large fields of view.
    - The `OrthogonalLens` class produces an isometric effect that preserves scaling and parallel lines in a scene.
    - The `PerspectiveLens` class produces a standard projection method that is best used when trying to emulate the camera setting in another 3D application.
    - The default `ZoomFocusLens` class is a legacy projection method that produces a general-purpose projection effect.

Congratulations! You have made it to the end of this book. We hope that you have found some 3D enlightenment along the way and that you have already felt compelled to start producing some of your own Flash projects and experiments using Away3D. If you are interested in learning more, you can always join the Away3D mailing list, or check out the tutorials and showcase sites over at www.away3d.com.

Adding a whole extra dimension to a traditionally 2D environment is no small step to master but can be hugely rewarding for those looking for an alternative approach to interactive web content. The animation and interaction capabilities of Flash combined with the 3D tools and interfaces of the Away3D engine offer a powerful platform that is worth exploring if you are keen to develop 3D applications and games that are easily accessible on the Web. Have fun!

Index