

2

Getting Started With Papervision3D

Emanating from the AS3 are a host of 3D open source endeavors, such as Away3D, Sandy3D, Altivina3D, and Papervision3D. Of the numerous 3D platforms hitting the stage, Papervision3D is by far the most popular and enjoys an ever-growing user group and development team.

Originally conceived by Carlos Ulloa in November 2005 and converted to Great White by Ralph Hauwert, it has grown from approximately 20 classes to over 300. It's maintained by a 10+ core and committer team. And with the release of Papervision3D 2.0, developers worldwide began downloading and contributing to this phenomenal open source project.

In this chapter, you'll start by downloading the basic Papervision3D classes, and examine the basics of 3D used in this book, and the guts of Papervision3D. Then, you'll create a simple hello world application and build a simple Papervision3D class. Finally, you learn how to apply BasicView to simplify the creation of a Papervision3D application.

Getting Papervision3D

You'll start by downloading the Papervision3D classes. Getting Papervision3D is not as simple as just downloading and unzipping it, you need a subversion client. Subversion is a great tool, which helps developers keep track of code changes. Since so many developers are working on Papervision3D it's essential to keeping it up to date and maintained. Here are the steps you need to follow to get the Papervision3D classes:

- Download and install a SVN client. Tortoise for Windows (<http://tortoisesvn.tigris.org/>) and for Mac-OS X Syncro SVN Client (<http://www.syncrosvnclient.com/>)
- For Windows, create an empty folder, name it Papervision3D, right click on it and choose "SVN checkout". For Mac, just add a new SVN project.
- Paste the address <http://papervision3d.googlecode.com/svn/trunk/> in the URL box and press OK
- Finish the download and examine the folder structure.

The repository has one trunk (most stable version) and several branches. Papervision3D is constantly changing and at the time of the writing of this book the CS4 branch was being eliminated and a PapervisionX branch being created. This brings up an essential point. Since Papervision3D is being constantly updated, care should be taken when upgrading a project's Papervision3D classes. Doing so could stop it from working or introduce instabilities. You should be careful to save all projects with their original Papervision3D files before upgrading especially if the project development spans more than a few weeks—that's how fast things are changing.

If you have any difficulty with the steps above, make sure you review the appropriate video on the book's web site. The getting started video goes through the process step-by-step.

Now that you have the Papervision3D classes you'll find what you need for the first part of this book in the AS3 branch. Navigate to it using the following path:

Papervision3D\as3\trunk\src

From the src folder grab the org and nochump folders and place them into your Flash or Flex src project folders. The org folder contains the majority of classes needed for Papervision3D run. The nochump folder is a set of zipping/unzipping classes that will be used for more advanced projects. As your projects become more advanced, you'll be adding other folders as well, which contain packages for physics, the Wii, and tweening...

For your convenience, all the examples in this book already have the appropriate classes bundled with them. They are ready to run. And you could skip the steps above. But to get the most up-to-date version of Papervision3D you must use an SVN.

Finally, make sure you read the license contained in the src folder. Papervision3D is completely open source, licensed under MIT which gives you full authority to modify and distribute it for commercial use-without restriction, including without limitation the rights to use, copy, modify, merge, and publish.

Diving into 3D

There are two different approaches to simulating 3D objects in ActionScript:

- Arranging and animating primitive objects in 3D space. Which entails animating display objects using the x, y and z properties, or setting rotation and scaling properties using the DisplayObject3D class.
- Generating 2D triangles from 3D geometry, and rendering those triangles with textures. To use this approach you must first define and manage data about 3D objects and then convert that data into 2D triangles for rendering.

You'll accomplish this in Papervision3D by using

- primitives and materials
- 3D mathematics, and programming hacks
- 3D modeling software (such as 3DSMax, Blender, and Swift3D)

But before you jump too far into the specifics of building 3D applications, it's important to cover a few important definitions.

Shapes vs. Objects

Depending on where you look there are some variations in 3D definitions. In this book you'll model 3D object and shapes using 3DSMax (Max for short), Blender, and Swift3D. To keep things simple, you'll use Max's definitions of shapes and objects. In Max, a shape refers to something 2D and an object is 3D. This is the difference between a rectangle and a box. The first is a shape, the second an object. 2D shapes make up 3D objects. For example, a circle extrudes into a cylinder.

The pieces that make up objects are called sub-objects. And sub-objects connect together to form what you see on the screen.

Here's the 3D terminology you'll be using throughout the book.

Vertex-a point in space (2D or 3D) that connects to other sub-object components to form a line or surface. It exists in a single x, y, and z coordinate.

Vertices (Verts)-more than one vertex.

Segment-a line between two vertices.

Spline-a combination of vertices and segments.

Shape-a 2D element.

Object-a 3D element.

Edge-a line connecting two vertices in 3D. Same as a segment but now in 3D.

Polygon-a face on an object composed of edges and vertices.

Sub-object-the pieces that make up an object.

The three most important sub-objects that you'll be using throughout this book to create 3D models are the vertex, edge, and face. Both 3DSMax and Blender devote entire panels to these sub-objects and they are key to modeling in 3D.

Splines

Splines are made up of vertices and segments as illustrated below.

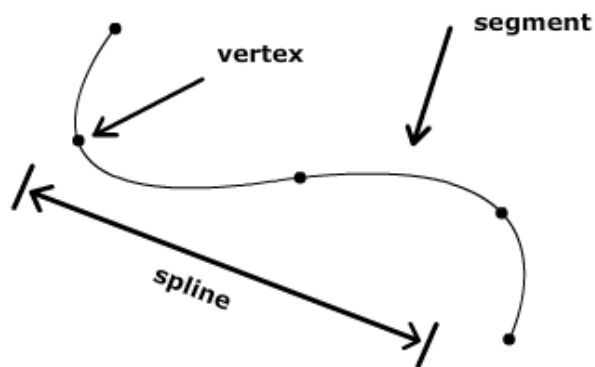


Figure 2-1 Elements in a Shape

You'll use splines to create paths and extrusions.

Objects

Objects consist of triangular sub-objects called faces, as illustrated below.

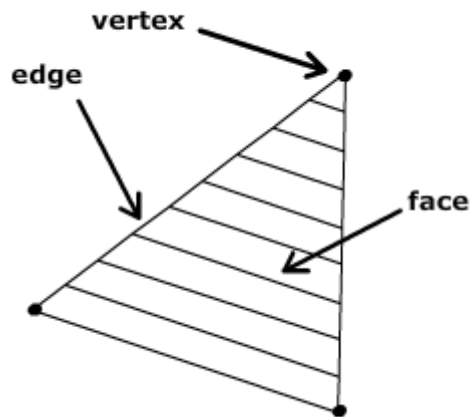


Figure 2-2 Elements in an object

You'll use vertex, edge, and face sub-objects shown above to model 3D objects. The majority of 3D objects are composed of triangles.

Triangles

Triangles are the most important shape in 3D. Since triangle math is easy to implement, the majority of 3D objects are created using triangles. Objects created from triangles are called meshes and the triangles in a mesh are called faces. You can create almost anything using triangles through a technique called box modeling. The book's blog contains extensive examples of box modeling in both Blender and 3DSMax.

Once you've created your 3D object you need to get it rendered so it can be viewed. Rendering refers to the sequence of drawing faces so that they make visual sense. The back bone of Papervision3D's rendering system uses a technique called hidden surface drawing.

Hidden Surface Drawing

In Papervision3D, there is a need for speed. Removing what is not visible is what makes a fast rendering algorithm work. Surfaces drawn that are hidden by other surfaces drawn over them subtract from your computer's available CPU resources. By removing hidden surfaces you reduce the number of polygons and speed up the rendering process. You only want to draw those polygons visible to your viewer. Painter's Algorithms is one of the easiest and most popular hidden surface methods.

Painter's Algorithm

The idea comes from oil painting. Oil painters found it easiest to paint back-to-front. The background is painted first-distant objects, then nearer ones: hence the term Painter's Algorithm. Rendering objects in Papervision3D uses a similar procedure.

Projecting a 3D scene onto a 2D viewport (the place where your image is projected) requires that you decide which polygons are viewable and which are not. The Painter's Algorithm sorts all the polygons in a scene by their depth and then paints them in this order, furthest to closest, painting over the parts that are normally not visible, thus solving the visibility problems

But it comes with the cost of painting redundant areas of distant objects. And the algorithm can fail in certain cases.

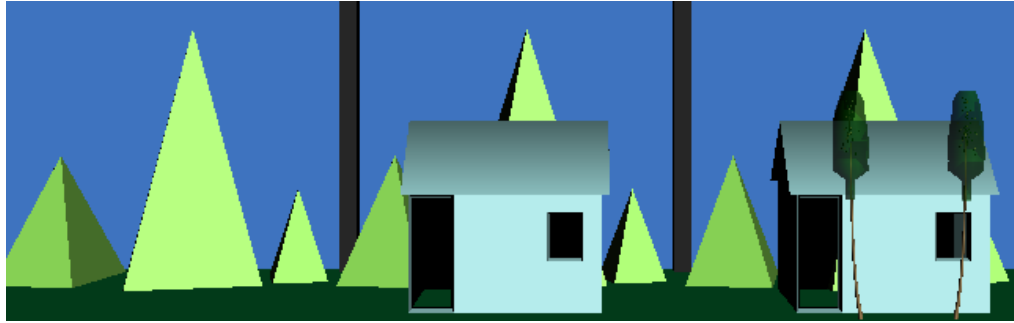


Figure 2-3 Painting furthest to nearest

In Figure 1-3, the distant pyramid mountains are painted first, followed by the closer house then the closest objects in the scene, the trees, are painted.

So why is it called hidden? When painting on a surface you paint over anything that is already there—hiding it. By arranging the polygons nearest the viewer at the end of a list, you paint every polygon into the framebuffer, overwriting the ones furthest away.

When does it fail?

When polygons overlap, as in Figure 1-4 below, you can't determine which polygon is above the other. The problem occurs since in depth sorting, each polygon is represented by only one depth value. But polygons are not just single points in space...they cover large portions of the projection plane.

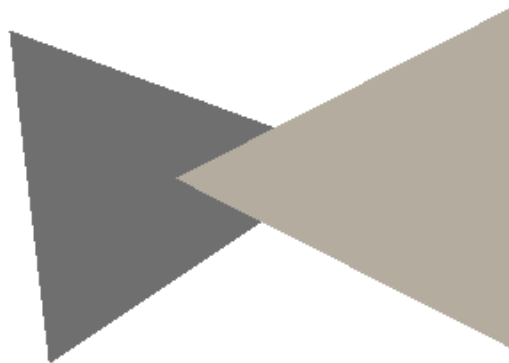


Figure 2-4 Painter's Failure

Such polygons need to be cut to allow sorting, and you'll cover an algorithm designed for this purpose in a later chapter. Flaws in the Painter's Algorithm have led to the development of Z-buffer techniques.

One major addition to Papervision3D to help overcome z-sorting issues has been the addition of the Quadrant Render Engine ported over by Andy Zupko from Away3D. More on that later...

The View Frustum (Culling and Clipping)

The view frustum contains everything that is visible in a 3D scene. The view frustum is pyramid-shaped and at its apex is the camera. The viewable volume is a bounded polyhedron. Its entry plane is

called the near clip plane and the end of its viewing volume is called the far clip plane. As shown below, objects outside of the viewable volume are culled from the scene.

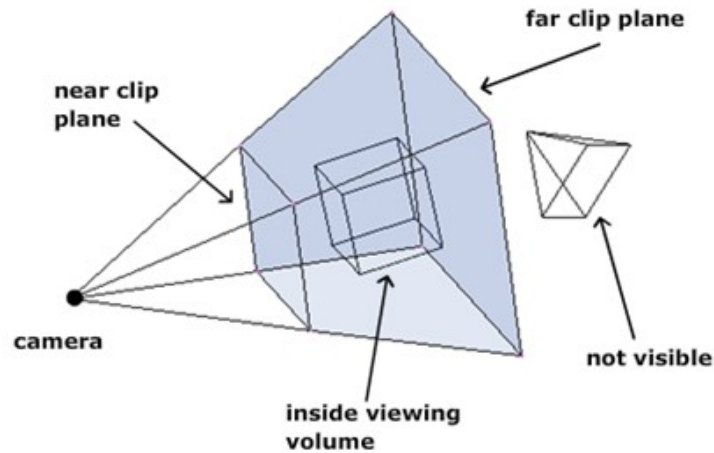


Figure 2-5 View Frustum

The cube inside the viewable volume shown above is projected onto the viewport while the pyramid outside is not drawn; it's culled.

To successfully map x and y coordinates to the projection plane (or viewport) you must map the view frustum shown above to a cube. Without going into all the details, this is accomplished using a projection matrix. Most 3D operations are accomplished using matrix algebra, which is beyond the scope of this book but addressed on the book's websites.

It is the depth in the view frustum that determines the z-sorting values.

Culling

Culling reduces the number of calculations required by your CPU, removing complete polygons whose vertices lie completely outside of the view frustum. For example in Figure 1-5 the pyramid beyond the far clip plane is not drawn. Since objects outside of the view frustum aren't seen by the observer drawing them is unnecessary and saves on required CPU resources. It's simple to determine which polygons should be drawn and is solely based upon a polygon's position with respect to the view frustum. And has nothing to do with the polygon's face angle with respect to the viewer.

When objects are completely in or out of the view frustum culling is a simple process. But when an object is partially in the view frustum it must be clipped to eliminate visual artifacts

Clipping

Clipping eliminates artifacts by modifying those polygons that lie partially in the view frustum. Clipping is more involved than culling and requires that you get a vector normal to the polygon face, and then take the dot product of that vector normal with a vector from the camera position (at the vertex of the view frustum).

The dot product is a very important quantity in Papervision3D and will be discussed in more detail later in this book. But in short, the sign of this dot product determines if the polygon is drawn or not. If the resulting scalar from the dot product is less than zero the polygon faces toward the camera and is drawn, otherwise it is not drawn.

Of the two (clipping and culling), culling gives you the greater performance enhancement since it reduces the number of calculations required. Both are handled automatically in Papervision3D and usually don't require adjusting.

Exploring the Guts of Papervision3D

All Papervision3D applications have the following elements in common. They can be thought of as the guts of Papervision3D.

- **Scene (Stuff):** The scene is where your objects are placed. It contains the 3D environment and manages all objects rendered to the screen in Papervision3D. It extends the DisplayObjectContainer3D class to arrange the display objects.
- **Viewport (Place):** The viewport displays the rendered scene. It's a sprite (rectangular region) that is painted with a snapshot of all the stuff contained in the view frustum. The viewport sprite inherits all the sprite methods and can be moved, positioned, and have effects added just like any normal sprite in Flash.
- **Camera (Eye):** The camera is your eye inside of the 3D scene and defines the view from which a scene will be rendered. Different camera settings will present a scene from different points of view. The camera is a DisplayObject3D and can be moved and rotated. And just like a digital camera you can change its zoom and focus. When rendering, the scene is drawn as if you were looking through the camera lens.
- **Object (Body):** An object is a combination of vertices, edges, and faces which provide a meaningful form for display, such as a car, avatar, or box. Objects are created in Papervision3D using primitives, or from 3D modeling software (such as 3DSMax, Blender, or Swift3D) and are embedded or imported into Papervision3D. They are DisplayObject3D objects added to your scene.
- **Material (Clothes):** The material is the texture, which is applied to an object. Textures can consist of various formats such as bitmaps, swfs, or video, and interact with light sources creating bump map effects and shaders.
- **Renderer (Artist)** The renderer draws a sequence of faces onto the viewport so that they make visual sense. Using your selected camera, it takes the data from your scene and draws it to the viewport. Typically the renderer is set into a loop using onEnterFrame or Timer methods native to the Flash player.

Running a Papervision3D application is like simultaneously recording and projecting your 3D scene as shown in the image below. The scene is the middleman. It's what the camera films and what the projector projects.

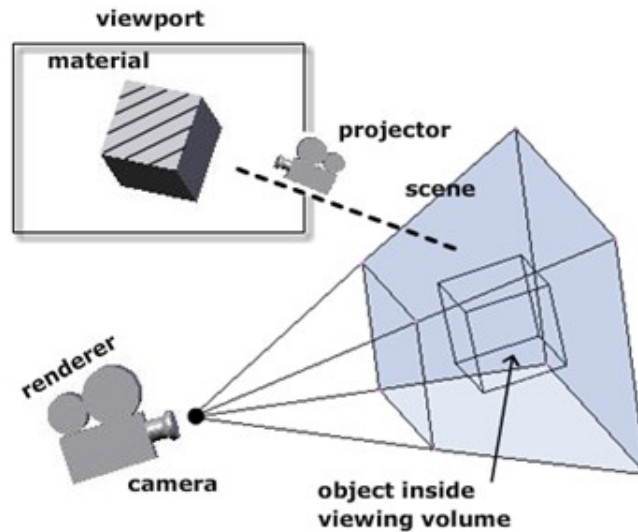


Figure 2-6 Guts of Papervision3D (camera and projector combined)

You can think of the whole process as a combination of a movie camera and projector that both films and projects your 3D scene simultaneously. The **scene** is what the **camera** is pointed at and the **objects** are the items that the camera films. The **materials** are what your objects are dressed in and determine how the light interacts with the scene elements. The reels on your projector are your **renderer** and as your reels turn (using `onEnterFrame` or `Timer` methods) they cast the recorded scene onto your screen which is your **viewport**.

Finally, at the heart of Papervision3D is the `DisplayObject3D` class.

DisplayObject3D

All objects placed in a Papervision3D scene are display objects. The `DisplayObject3D` class represents instances of 3D objects that are contained in the scene. That includes all objects in the scene, not only those that can be rendered, but also the camera and its target. The `DisplayObject3D` class supports basic functionality like the x, y and z position of an object, as well as `rotationX`, `rotationY`, `rotationZ`, `scaleX`, `scaleY` and `scaleZ` and `visibility`. It also supports more advanced properties of the object such as its `transform Matrix3D`.

`DisplayObject3D` is not an abstract base class; therefore, you can call `DisplayObject3D` directly. And in many instances you will create an empty `DisplayObject3D` object to place elements in. Invoking `new DisplayObject()` creates a new empty object in 3D space, like when you used `createEmptyMovieClip()`. And most importantly, it is the class where you'll add your physics: mass, velocity, acceleration...

Now that you have been introduced to the basic elements, you'll build your first Papervision3D application.

You'll now put all of this into practice by creating your first Papervision3D Class.

Running Papervision3D 4 Different Ways (Hello World)

Since 1974 (thanks to Bell Labs), it has been a tradition to start with a Hello World program when learning a new language. But with Papervision3D there are four ways to say hello and two different programs to say it in.

In Flash you can run Papervision3D as a class package or as a timeline scripting application. Similarly for Flex you can run Papervision3D as an ActionScript package or as an MXML application with the code sandwiched in between ActionScript tags. The four possible options are diagrammed below:

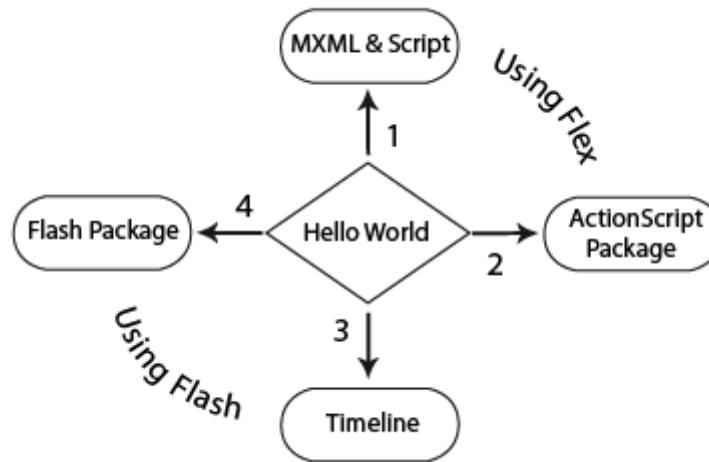


Figure 2-7 Four Ways to Say Hello in Papervision3D

You might ask which one you should use. And it really depends on the application that you are writing. Some things are better done in Flash and others in Flex and in certain cases it is easier to develop applications using timeline ActionScript instead of using packages. If you are not familiar with writing class packages you will certainly find using timeline ActionScript in Flash or the script tags in Flex easier at first. But in this book, an emphasis will be placed on developing ActionScript class packages.

Using Flash or Flex

You’ve certainly heard the famous line “parts are parts” but unfortunately it’s not true for Flex and Flash. They have different component sets. Mixing Flash code that uses Flash components with Flex is not easily done and vice versa. Flex and Flash can only interchange class packages that don’t contain elements of their specific component sets. As soon as you depart from the core language and start adding Flash or Flex components your program becomes non-transferable to the other platform.

For example, Flex does not have the overhead of the Flash IDE. Elements like Flash’s video player are not transferable. Your code loses its transferability as you dip into the component base of either. This makes writing a book on both a little challenging, but definitely more interesting.

To make things easier, you’ll first learn to program Papervision3D in both Flash and Flex, developing classes generic to both. The first part of the book emphasizes Flash. And the second part Flex. And towards the end it really gets exciting. You’ll learn how get Flash, Flex, and Air working together to produce dynamic 3D Flash-Flex content.

There is an important difference when working with Flex applications where you are using MXML and Script tags. In order to add children to the Flex stage you must put them in a canvas or another component and use the rawChildren tag shown below.

```
pv3Canvas.rawChildren.addChild(viewport);
```

This modification allows you to add non-native Flex display Objects to the Flex display stack. This is essential in getting your MXML and Script Tag application to run, otherwise nothing will happen.

For script editing, Flex has become the preferred vehicle. Its debugging capabilities, incorporating code completion and ctrl-click variable searching, are far superior to Flash. Writing code in Flex, as an ActionScript package, has become the preferred vehicle for many Papervision3D developers.

With that said, you'll now build your "Hello 3D World" class package.

Building a Class

To fully leverage the power of AS3 you must learn to build classes. Fundamental to Papervision3D is the class structure. In this section, you'll start at ground zero, building the framework for your first custom class, and then building it out. Then you'll examine the basic Papervision3D class and get your first Papervision3D application running in Flash.

Writing an OOP (Object Oriented) Class

If you've been a Flash programmer for any amount of time, you're probably used to putting items in the library and instantiating them to the stage. In a similar way classes are templates that you will use over and over again just like instances from a library. This will become very apparent when you start working with particles in chapter 3.

But what's a class?

It's a blueprint for how an object acts. For example, movie clips and buttons are classes. They respond and act with certain behaviors and have properties that are contained in their classes. Classes possess two important quantities: properties and methods. Properties hold information. They are like variables. And methods like functions hold what a class can do or how it behaves. The best way to understand classes is to write your own. Writing your own class is called a custom class. This is how you get started...

- Open up Flash CS3 (or any text editor), and from its welcome screen under Create New select ActionScriptFile.
- In the Flash IDE, go to File Save and give your file a name. Now this is important. The name of your file is the name of your class. Always start your class name with a capital letter. In this case call it "Papervision3DClass".
- Select a folder. Select save. Note: The file saved is Papervision3DClass.as
- Place the following code into the Flash code window

```
package
{
    class Papervision3DClass
    {
        function Papervision3DClass()
        {
            trace("Hello 3D World");
        }
    }
}
```

All classes start with the package name. The package statement is a special name that wraps your class. Next you define your class just like you would define a variable or function. The class name must be the same name as your file name. In your case it is Papervision3DClass. Then you create the constructor function. Your constructor function runs automatically when the class is created or instantiated. The

constructor function is the same name as your class and does not have a return class such as void or string. Now place any code inside the constructor that you want to run. In this first example, you put

```
trace("Hello 3D World");
```

When your code runs it sends Hello 3D World to your Flash trace window. Congratulations, you just wrote your first custom class!

Now you'll extend it.

Extending Your Class (Inheritance)

Why reinvent the wheel? Extending your class allows you to use what is already available from another class. That's the real power of using classes. A class can inherit from properties and methods of another class (as long as those properties and methods aren't private). The class that is extended is called the subclass and the class it is inheriting from is called the super class.

In the example below, you'll now extend your Papervision3D class with the Sprite class. In this case, the Sprite class is your super class and the Papervision3DClass is your subclass.

After the "Papervision3DClass" class statement type "extends Sprite" as shown below:

```
class Papervision3DClass extends Sprite
```

This enables your Papervision3DClass to take everything from the Sprite class and use it. Essentially the Sprite class is a display object container that does not carry the overhead of a movie clip. Its use reduces the resources needed to place your 3D objects on stage.

But your class will not run yet; you need to import the Sprite class.

```
import flash.display.Sprite;
```

Import statements go after the package curly bracket, and importing the Sprite class brings in all the information needed to allow your custom class to behave as a Sprite. Your code should look like the code below:

```
package
{
    import flash.display.Sprite;

    class Papervision3DClass extends Sprite
    {
        function Papervision3DClass()
        {
            trace("Hello World");
        }
    }
}
```

In Papervision3D, you'll be extending numerous classes.

Chinese Hello

Here's another example that demonstrates how a subclass can use a method of its super class. Consider the following two classes:

The HelloClass has a chineseHello() method that when executed says hello in Chinese.

HelloClass

```
package
{
    public class HelloClass{
        public function HelloClass()
        {

        }
        public function chineseHello():void
        {
            trace("Ni Hao");
        }
    }
}
```

The GoodByeClass has a chineseGoodBye() method that when executed says goodbye in Chinese.

GoodByeClass

```
package
{
    public class GoodByeClass{
        public function GoodByeClass()
        {

        }
        public function chineseGoodBye():void
        {
            trace("Zai Jian");
        }
    }
}
```

But what if you want to say both hello and goodbye in one class? As opposed to rewriting your HelloClass to contain the chineseGoodBye() method you extend it using the extends method as follows:

```
package
{
    public class HelloClass extends GoodByeClass{
        public function HelloClass()
        {

        }
        public function chineseHello():void
        {
            trace("Ni Hao");
        }
    }
}
```

Placing the following code in the Flash ActionScript editor, you gain access to both functions from a single class using the extended HelloClass as follows:

```
var subclass:HelloClass = new HelloClass();
subclass.chineseHello();
subclass.chineseGoodBye();
```

Even though this is a simple example, it becomes extremely powerful when dealing with classes that are 1000's of lines long.

But what if you want to change a method of the super class (GoodByeClass) in the sub class (HelloClass)? No longer do you want to say goodbye to your new Chinese friend, but something else. To do so you must override the chineseGoodBye() method using the override technique as follows:

```
package
{
    public class HelloClassOverride extends GoodByeClass{

        public function HelloClassOverride()
        {

        }

        public function chineseHello():void
        {
            trace("Ni Hao");
        }

        override public function chineseGoodBye():void{
            trace("Hen gao xing ren shi nin");
            //I am very glad to meet you.
        }

    }
}
```

You can run this class from the Flash ActionScript editor by using the following code:

```
var subclass:HelloClassOverride = new HelloClassOverride();
subclass.chineseHello();
subclass.chineseGoodBye();
```

So now when you call the chineseGoodBye() function you now get a totally different message (I am very glad to meet you) and are now ready for a longer conversation with your new Chinese friend. Of course, this technique only works if the super class method is public.

With this under your belt, you're now ready to start building out your Papervision3D class.

Creating Methods

A method is a fancy name for a function except now it's in a class. You've probably written tons of functions. Put one in a class and it's now a method. But with one twist: a method is a function attached to an object, which means it can be instantiated (or used over and over again).

You place your custom methods after the constructor class. For example, place the following initPV3D() custom method underneath your custom class. You'll build out your code out in the next section on formulating a Papervision3D Class.

```
function initPV3D():void
{
    trace("Initialize Papervision3D Here");
}
```

Only your custom class can run your custom method. Your constructor calls the custom method. You do this by placing a function call inside of your constructor as follows:

```
package
{
    import flash.display.Sprite;

    class Papervision3DClass extends Sprite
    {
        function Papervision3DClass()
        {
            trace("Hello World");

            initPV3D();
        }
    }
}
```

```

    }

    function initPV3D():void
    {

        trace("Initialize viewport here");

    }

}

```

When the constructor runs, it calls the custom method `initPV3D` that executes your custom method. But first, you must set your scope. Scope defines the access level of your properties and methods.

Defining Scope

As discussed earlier a public class is necessary if a super class's method is going to be overridden. When defining class methods and properties, you must define their scope or access modifiers. An access modifier governs the availability of variables, functions, classes, interfaces, and namespaces.

In building Papervision3D classes, you'll use access modifiers to control access to both properties and methods. Class properties use the following attributes to control property access:

- `internal` (default)-visible to references inside the same package.
- `private`-visible to references in the same class.
- `protected`-visible to references in the same class and derived classes.
- `public`-visible to references everywhere.
- `static`-specifies that a property belongs to the class, as opposed to instances of the class.
- `UserDefinedNamespace`-custom namespace name defined by user.

The four access modifiers used to control methods are: `public`, `private`, `protected`, and `internal`. Using **public** makes a method visible anywhere in your script. **Private** makes methods only visible within the defining class. **Protected** makes methods visible within the defining class and subclasses. And **internal** is the default attribute and allows a method to be called within its defining classes.

As an example of using access modifiers, the function `initPV3D` is called from the constructor function, but can't be called outside of the class due to the private restriction.

```

package
{
    import flash.display.Sprite;

    public class Papervision3DClass extends Sprite
    {
        Public function Papervision3DClass()
        {
            trace("Hello World");

            initPV3D();

        }

        private function initPV3D():void
        {

```

```
        trace("Initialize Papervision3D Here");
    }
}
}
```

Custom Properties

You'll now finish up your class by adding some custom properties. Property is a fancy name for variable. For Papervision3D the four main properties/methods needed to create a 3D object for viewing are

- Viewport3D
- Camera3D
- Scene3D
- Render

As discussed earlier, the viewport is where you see the projected scene, the camera the observer's position, the scene where your objects live in 3D space, and the renderer that paints the 3D onto a 2D sprite canvas called a viewport.

You'll find them in every Papervision3D movie. Without them Papervision3D will not work. You'll learn about each one in the next section. To place these properties into your package, paste them right after the class declaration.

```
private var scene:Scene3D;
private var camera:Camera3D;
private var viewport:Viewport3D;
private var renderer:BasicRenderEngine;
```

Don't forget your import statements. For each property to work properly, you'll need its import statement as well.

In summary, custom classes consist of six elements:

- package name
- import statements
- properties
- class declaration
- constructor function
- methods

In addition to properties and methods, Papervision3D classes use a 3rd, very important characteristic called an event. Events determine which instructions are carried out and when. An event gives Papervision3D the ability to listen and respond to elements in your program such as a button click, or

enter frame occurrence. In essence, the Adobe Flash Player (using events) just sits and waits for something to happen, and then executes the designated code when it does.

Using Papervision3D requires that you work with classes and events. And write your own custom classes. In the next section, you'll start by formulating your first Papervision3D class.

Formulating a Papervision3D Class

Starting with the code from the previous section, you're probably wondering... "Where do all those Papervision3D import statements come from?"

Presently they are downloaded from Google Code at

<http://code.google.com/p/Papervision3D/source/checkout>

You'll need an SVN client, such as Tortoise to download them as described earlier. If you have problems, just take a look at the book's website.

```
package
{
    import flash.display.Sprite;

    import org.papervision3d.scenes.Scene3D;
    import org.papervision3d.cameras.Camera3D;
    import org.papervision3d.render.BasicRenderEngine;
    import org.papervision3d.view.Viewport3D;

    public class Papervision3DClass extends Sprite
    {
        private var scene:Scene3D;
        private var camera:Camera3D;
        private var renderer:BasicRenderEngine;
        private var viewport:Viewport3D;

        public function Papervision3DClass()
        {
            trace("Hello World");

            initPV3D();
        }

        private function initPV3D():void
        {
            trace("Initialize Papervision3D Here");
        }
    }
}
```

Assuming that you have downloaded the Papervision3D files correctly, you have all you need to run your first 3D project.

Your goal is simple-to put a wire frame sphere on the stage. How difficult could that be?

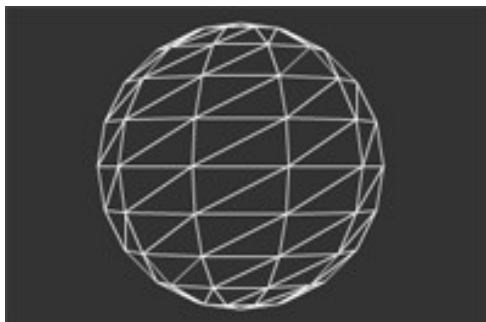


Figure 2-8 Your Goal

So now you'll finish your code. Since you are building a sphere you need a few more import statements such as those shown below for the sphere, its wire frame, and its material. Place this code after your other import statements.

```
//Primitives and Materials
import org.papervision3d.objects.primitives.Sphere;
import org.papervision3d.materials.WireframeMaterial;
import org.papervision3d.core.proto.MaterialObject3D;
```

And add a sphere variable.

```
//Define your sphere variable.
private var sphere:Sphere;
```

The next set of code is the `initPV3D` function, which runs in the constructor function. This function follows the `initPV3D` constructor function. The `initPV3D` function has the following three parts

- viewport, camera, scene, and renderer creation
- object creation
- and creation of a frame iteration mechanism.

In this case you use `ENTER_FRAME` to create your looping action, but you can use a timer to create the loop function as well. The advantage of a timer is that you can throttle your application. You'll become very familiar with throttling frame animation in this book.

```
//Initialize Papervision3D
private function initPV3D():void
{

    // Create the viewport
    viewport = new Viewport3D(0, 0, true, false);
    addChild(viewport);

    // Create the camera
    camera = new Camera3D();

    // Create the scene
    scene = new Scene3D();

    // Create the renderer
```

```
renderer = new BasicRenderEngine();

//Create the Objects
createObjects();

// Create the render loop
this.addEventListener(Event.ENTER_FRAME, loop);
}
```

The creation of a viewport, camera, scene, and render is at the heart of what makes Papervision3D work.

Viewport

The viewport has four parameters. The first two are the width and height. These are critical parameters when it comes to optimizing your application.

Why? The viewport is your projection screen. And two thirds of the processing that occurs requires taking Papervision3D's 3D objects and projecting them on the viewport. The viewport is a sprite in Flash. The larger the viewport the larger the amount of work your processor needs to do.

If you set these first two parameters to 0, 0 Papervision3D uses the entire stage to render. This is a big processor resource hit if you have a complex scene. Make the viewport smaller if you can. The next two parameters are Booleans. The first Boolean if set to true resizes the viewport with the browser window (auto scale to stage), the second sets the viewport interactivity.

```
// Create the viewport
viewport = new Viewport3D(0, 0, true, false);
addChild(viewport);
```

There are two more Booleans not shown here. They are clipping and culling which are set to true by default. Most of the time, you'll leave these alone and accept the default values.

Then you add the viewport to the stage, but keep in mind that you can have multiple viewports. And creating multiple viewports is a way of solving the z-sorting problems that commonly arise with Papervision3D.

For example, when placing a house on a simple terrain you'll have sorting issues. Instead of placing the house and terrain in the same viewport, create two viewports and place your terrain in one viewport and your house in the other. Not only will this solve many of your sorting issues, it gives you the ability to render them differently.

Think of the viewport as the canvas that you paint your image on. It's where Papervision3D meets the 2D world of Flash.

Camera

There are three cameras in Papervision3D-free, frustum, and target. The command for adding a camera is

```
// Create the camera
camera = new Camera3D();
```

The camera has a number of properties such as zoom, and focus. When working with 3D carousels, for example, you'll find it useful to use focus and zoom. Especially when bringing panels in for close observation. What's the difference between zoom and focus?

Focusing is equivalent to magnifying elements at the viewport, but you are not really getting closer. Opening up the focus too large will distort your objects in the 3D scene.

Zooming essentially changes the distance from the camera to the view frustum. Zooming in too much will put you on the other side of you object and it will seem as if you object has disappeared.

Scene

This is where the 3D stuff lives, and is added to Papervision3D by using the following command

```
// Create the scene
scene = new Scene3D();
```

In reality (or virtual reality that is) you never see the stuff in the scene. It is being handled mathematically and when it is rendered it is cast onto the viewport. This is about one third of the work on your CPU, the other two thirds goes into the rendering process when the scene is culled, clipped, and painted onto your viewport.

Renderer

The renderer performs all the necessary calculations to paint your 3D data into a 2D displayable sprite called the viewport.

It's simple to create the sphere with a wire frame. Instantiate a material for the sphere (a wireframe in this case), and instantiate a sphere, then add your new material to the sphere. Then you add the sphere to the stage by using `scene.addChild(sphere)`.

```
private function createObjects():void
{
    // Create a material for the sphere
    var sphereMaterial:MaterialObject3D = new
WireframeMaterial(0xFFFFFFFF);

    // Create use wireframe material, radius 100, default 0,0,0
    var sphere:Sphere = new Sphere(sphereMaterial, 100, 10, 10);

    // Add Your Sphere to the Scene
    scene.addChild(sphere);
}
```

Finally, you need a looping function. The function below is very simple. You will use it to create complex interactions of primitives in the next section.

```
//Start rendering
private function loop(evt:Event):void
{
    renderer.renderScene(scene, camera, viewport);
}
```

Putting it all together you have the final base code.

Final Base Code

After making all the changes above your final code should look like the code below. Now it's time to have some fun and run your code. Check out the next section on Running Papervision3D.

```
package
{
    //Flash imports
    import flash.display.Sprite;
    import flash.events.Event;

    //Basic Papervision3D Engine
    import org.papervision3d.scenes.Scene3D;
    import org.papervision3d.cameras.Camera3D;
    import org.papervision3d.render.BasicRenderEngine;
    import org.papervision3d.view.Viewport3D;

    //Primitives and Materials
    import org.papervision3d.objects.primitives.Sphere;
    import org.papervision3d.materials.WireframeMaterial;
    import org.papervision3d.core.proto.MaterialObject3D;

    //Define your class

    public class Papervision3DClass extends Sprite
    {
        //Define your properties
        private var scene:Scene3D;
        private var camera:Camera3D;
        private var renderer:BasicRenderEngine;
        private var viewport:Viewport3D;

        //Define your sphere variable.
        private var sphere:Sphere;

        public function Papervision3DClass()
        {
            trace("Hello World");

            initPV3D();
        }

        //Define your methods

        //Initialize Papervision3D
        private function initPV3D():void
        {

            // Create the viewport
            viewport = new Viewport3D(0, 0, true, false);
            addChild(viewport);

            // Create the camera
            camera = new Camera3D();

            // Create the scene
            scene = new Scene3D();

            // Create the renderer
            renderer = new BasicRenderEngine();

            //Create the Objects
            createObjects();
        }
    }
}
```

```

// Create the render loop
this.addEventListener(Event.ENTER_FRAME, loop);

}

//Create your objects
private function createObjects():void
{

    // Create a material for the sphere
    var sphereMaterial:MaterialObject3D = new
WireframeMaterial(0xFFFFFFFF);

    // Create use wireframe material, radius 100,
default 0,0,0
    var sphere:Sphere = new Sphere(sphereMaterial, 100,
10, 10);

    // Add Your Sphere to the Scene
    scene.addChild(sphere);

}

//Start rendering
private function loop(evt:Event):void
{

    renderer.renderScene(scene, camera, viewport);

}

}
}

```

The final base code consists of two methods, which are `initPV3D` and `createObjects`. The `initPV3D` function sets up the `Papervision3D` engine, which renders your object discussed above. And the `createObjects` function creates the object to be rendered. It consists of two parts. Adding a sphere primitive and adding a material. The code to accomplish this is given below

```

// Create a material for the sphere
var sphereMaterial:MaterialObject3D = new WireframeMaterial(0xFFFFFFFF);

// Create use wireframe material, radius 100, default 0,0,0
var sphere:Sphere = new Sphere(sphereMaterial, 100, 10, 10);

```

And once the sphere primitive is added and the material is added to the primitive (a wire mesh in this case). The next chapter goes into the details of adding primitives and materials. For now it's enough to understand that every primitive requires a material to be seen. Materials in `Papervision3D` have a life of their own and are more like objects themselves as opposed to just static bitmaps. They can contain animation, video, and interact with light. It is through the material that your object becomes interactive. And in addition, you can create your own materials and primitives-more on that in chapter 2 and 3.

Finally, the sphere is added to the scene using the `addChild` method shown below;

```

// Add Your Sphere to the Scene
scene.addChild(sphere);

```

AddChild adds a visual object to the scene. And like all XML, this child has a parent. To determine its parent you use the trace statement as follows:

```
trace(sphere.parent)
```

The results of the trace statement above is[object DisplayList]. The parent of the sphere is the DisplayList class. Whenever you add a child to anything, the parent becomes the object that uses the addChild method. This will become very important when you create linked objects in Papervision3D. Comparing this to Second Life, it is the parent that holds all the other objects linked to and it is the parent that holds the code.

Similarly you can remove an object from the stage using the removeChild command. Later on you'll exam numerous ways to work with children. These include referencing display objects by index number and name, controlling display objects different movies, and using addChild to change parents

Running Papervision3D

You have a number of options for running your custom class. In Flash, you can use the document class, a movie clip, or class path to run your custom class. In addition to Flash, Flex is an attractive alternative to running your custom class. It has a more powerful editor than Flash. And if you are a student or teacher, it's free from Adobe. You'll first run your custom class using Flash's document class.

In order for the code to run, you must have the Papervision3D classes included in the same folder with the FLA as shown below. The Flash player looks for the import classes in the same folder as your FLA. Those folders are com, fl, nochump, and org as shown below. You will learn more about them in later chapters.

If you want to place the Papervision3D classes in another folder you will need to set the class path to that folder. Setting the class path is covered in the section on class path below.



Figure 2-9 Folder Contents

Make sure you have placed the Papervision3D class folders in the same folder as your FLA. Otherwise, your program will not run.

Document Class

In previous versions of Flash, the main timeline was a movie clip. In CS3, you can now define your own custom class for your document. To find the document class go to the property inspector in Flash. Note: If it is not shown in the properties box click on the grey space off of your stage. Next click in the Document class area and type in the name of your class Papervision3DClass. Click off the stage and test your movie.

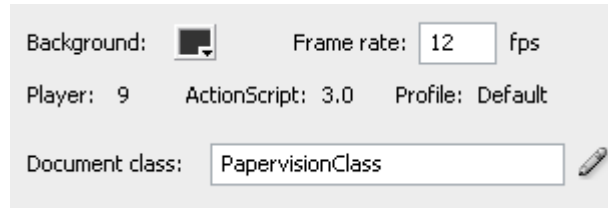


Figure 2-10 Using Document class

Class Path

Class path is a good way to organize your classes. As opposed to placing your classes in your FLA folder every time you create a new Flash project you can specify a folder outside of your FLA folder anywhere on your hard drive. And Flash can import your classes from that external folder at runtime. Using class path you can have several different Flash projects using the same external classes.

To test this out, create a folder on your desktop and place all your Papervision3D classes in that folder. To set your class path do the following:

- Go to the properties inspector
- Select publish setting
- Select Flash
- Select Settings next to the ActionScript version (ActionScript 3.0)
- In the Classpath area at the bottom of the screen, click the target button and browse to your folder on the desktop containing your Papervision3D class.
- Select the folder and select choose and click OK and OK again. You now have a class path specified.
- Make sure all movie linkage properties, package names, and import names are set correctly. Test your movie.

If you don't want to do this, just drop your Papervision3D classes into the same folder as your FLA. And Flash knows to look first for the classes that it needs to import from the same folder that its FLA is in.

In this book, we provide the entire set of classes with each code sample. This will ensure that your downloaded code sample will run and you'll not have to go scampering for the version of Papervision that we used for the demo.

Flex Builder

You can run the program you created above, Papervision3DClass in Flex. As mentioned earlier, there are actually several ways to run and construct a Papervision3D program in both Flex or Flash. Flash and Flex programs are not always interchangeable. They don't have the same component base.

Flex has two distinct advantages over Flash.

- a better code editor,

- and streamlined data handling.

The big disadvantage is the learning curve, but with AS3 the learning curve is already high. So why not learn Flex anyway. Chances are in the course of your career you will need to use both.

So if you are running data rich Papervision3D applications, Flex is the way to go. And if you hate the Flex look? In this book, you'll learn how to use Flash assets to enhance Flex applications. Your user will not know the difference. You'll learn how to make a Flex application look like Flash with the addition of dynamic container animation.

To create your custom class in Flex,

- Open up Flex. Select new
- Create an Action Script Project and name it Papervision3DClass as before. Note: Flex automatically creates the class package for you, shown below.

```
package {
    import flash.display.Sprite;

    public class Papervision3DClass extends Sprite
    {
        public function Papervision3DClass()
        {
        }
    }
}
```

- Add in the script that you created in the previous exercise. Essentially just replace what Flex generated with the Flash Papervision3DClass file created earlier in this chapter.
- Make sure that you put all the Papervision3D classes in the Flex Papervision3DClass folder as shown below (or you'll get a million errors).

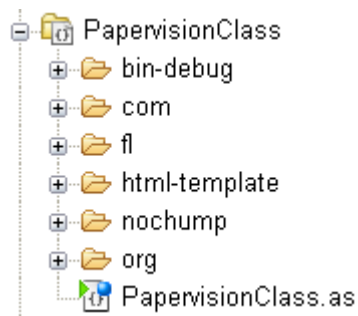


Figure 2-11 Include the Papervision3D classes in the Flex Package

- Finally, hit the run button. Oops... your sphere appears at the lower right of your screen.
- Change the position of the sphere by changing its x, y coordinate in the createObjects() function.

```
sphere.x=-300;  
sphere.y=100;
```

A better way to set the initial position is to use the Stage class, which is covered in the section on BasicView. Your sphere in Flex should look like the one in Figure 1-8. You'll now add motion.

Adding Motion

Your sphere is just sitting motionless on the screen. And you might be wondering if this is all there is to Papervision3D? Should you ask for a refund? Not yet... you'll extend this code to do some pretty amazing things in the coming chapters. But for now you'll add some simple motion.

Adding motion to your sphere is really easy. You'll add three types of motion in this section

- rotation,
- translation,
- and scaling.

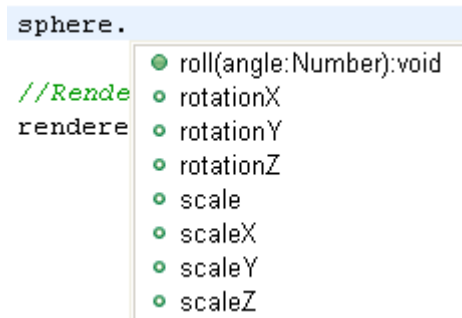
Since you are already iterating the renderer, in order to get your sphere moving, all you have to do is put some code in the myLoop function in your base code.

But what do you add and how do you find that code?

There are three ways to discover the commands that are available to use. You can open up the DisplayObject3D class and look at all the available functions. Or you can open up the sphere class contained in the Papervision3D classes (org /Papervision3D /objects /primitives /Sphere.as) and in the comments you'll read

```
It includes x, y, z, rotationX, rotationY, rotationZ, scaleX, scaleY  
scaleZ and a user defined extra object.
```

The other way is to open your application in Flex and use code hinting to find the available commands. Typing a period after the instantiated name sphere brings up all the possible commands that you can use with your sphere. The figure below shows what you see in Flex.

A screenshot of an IDE showing code completion for the variable 'sphere'. The text 'sphere.' is entered in a light blue box. Below it, the code '//Render' and 'rendere' is visible in green. A dropdown menu is open, listing several methods with a green circle icon next to each: roll(angle:Number):void, rotationX, rotationY, rotationZ, scale, scaleX, scaleY, and scaleZ.

```
sphere.  
//Render  
rendere  
● roll(angle:Number):void  
● rotationX  
● rotationY  
● rotationZ  
● scale  
● scaleX  
● scaleY  
● scaleZ
```

Figure 2-12 Using Flex Code Hinting to Discover Motion Function

From Flex you find that the commands for translation are x, y, and z. The commands for rotation are rotationX, rotationY, and rotationZ. (later in the book you learn about pitch, yaw, and roll). The commands for scaling are scaleX, scaleY, and scaleZ.

So your render function should be modified to look like the code below

```
private function myLoop(evt:Event):void
{
    //Rotate around x-axis
    sphere.rotationX+=2;

    //Increase angle
    myTheta += 2;
    //plot a circular orbit sin and cos
    myX = Math.cos(myTheta * Math.PI / 180) * 100-300;
    myZ = Math.sin(myTheta * Math.PI / 180) * 100;
    //Set X and Z of spher
    sphere.x = myX;
    sphere.z = myZ;

    //Fattening and scrinking
    myY = 1+Math.cos(myTheta * Math.PI / 180)*.7;
    sphere.scaleX=sphere.scaleZ= 1+Math.sin(myTheta * Math.PI /
180)*.7;
    //Elongate
    sphere.scaleY =myY;

    //Render Scene
    renderer.renderScene(scene, camera, viewport);
}
```

In the first part you add rotation around the x-axis. By using

```
//Rotate around x-axis
sphere.rotationX+=2;
```

with each iteration of the loop the sphere rotates around the x-axis 2 degrees. How do you know it is degrees and not radians? While in Flex builder hold down the ctrl key and roll over the function you want to investigate. Click on that function and it will take you to its class. In this case, the rotation class states clearly in its code that you are using degrees.

```
//Increase angle
myTheta += 2;
//plot a circular orbit sin and cos
myX = Math.cos(myTheta * Math.PI / 180) * 100-300;
myZ = Math.sin(myTheta * Math.PI / 180) * 100;
//Set X and Z of spher
sphere.x = myX;
sphere.z = myZ;
```

Finally, you stretch and shrink the sphere by using trig functions (cosine and sine) and the scale functions.

In the next part, translation is added by doing a simple trig rotation around a circle by incrementing the myTheta variable.

```
//Fattening and scrinking
myY = 1+Math.cos(myTheta * Math.PI / 180)*.7;
sphere.scaleX=sphere.scaleZ= 1+Math.sin(myTheta *
Math.PI / 180)*.7;
```

```
//Elongate  
sphere.scaleY =myY;
```

Adding animation and interactivity to Papervision3D is loads of fun. And you will make use of the techniques developed in this section many times in this book.

Below is your final code. It forms the basis for much of what you will do in this book.

```
package  
{  
    //Flash imports  
    import flash.display.Sprite;  
    import flash.events.Event;  
    import flash.display.StageAlign;  
    import flash.display.StageScaleMode;  
    //Papervision3D Imports  
    import org.papervision3d.cameras.Camera3D;  
    import org.papervision3d.core.proto.MaterialObject3D;  
    import org.papervision3d.materials.WireframeMaterial;  
    import org.papervision3d.objects.primitives.Sphere;  
    import org.papervision3d.render.BasicRenderEngine;  
    import org.papervision3d.scenes.Scene3D;  
    import org.papervision3d.view.Viewport3D;  
  
    //Define your class  
    public class Papervision3DClass extends Sprite  
    {  
        //Define your properties  
        private var viewport:Viewport3D;  
        private var camera:Camera3D;  
        private var scene:Scene3D;  
        private var renderer:BasicRenderEngine;  
  
        //Define your sphere variable.  
  
        private var sphere:Sphere;  
        private var sphereMaterial:MaterialObject3D;  
        private var myTheta:Number=0;  
        private var myX:Number=0;  
        private var myY:Number=0;  
        private var myZ:Number=0;  
  
        public function Papervision3DClass()  
        {  
            trace("Hello World");  
  
            //Intiate Papervision3D  
            initPV3D();  
  
            //Create Your Objects  
            createObjects();  
  
            //Create Renderer  
            createRenderer();  
        }  
  
        //Define your methods  
  
        //Initialize Papervision3D  
        private function initPV3D():void  
        {
```

```

        // Create the viewport
        viewport = new Viewport3D(0, 0, true, false);
        addChild(viewport);

        // Create the camera
        camera = new Camera3D();

        // Create the scene
        scene = new Scene3D();

        // Create the renderer
        renderer = new BasicRenderEngine();

        //Initialize Stage
        stage.align = StageAlign.TOP;
        stage.scaleMode = StageScaleMode.NO_SCALE;
    }

    //Create your objects
    private function createObjects():void
    {
        // Create a material for the sphere
        sphereMaterial = new WireframeMaterial(0xFFFFFFFF);

        // Create use wireframe material, radius 100,
        sphere = new Sphere(sphereMaterial, 100, 10, 10);

        sphere.x=-300;
        sphere.y=130;

        // Add Your Sphere to the Scene
        scene.addChild(sphere);
    }

    //Loop Renderer
    private function createRenderer():void{
    addEventListener(Event.ENTER_FRAME, myLoop);
    }

    //Single Loop
    private function myLoop(evt:Event):void
    {
        //Rotate around x-axis
        sphere.rotationX+=2;

        //Increase angle
        myTheta += 2;
        //plot a circular orbit sin and cos
        myX = Math.cos(myTheta * Math.PI / 180) * 100-300;
        myZ = Math.sin(myTheta * Math.PI / 180) * 100;
        //Set X and Z of spher
        sphere.x = myX;
        sphere.z = myZ;

        //Fattening and scrinking
        myY = 1+Math.cos(myTheta * Math.PI / 180)*.7;
        sphere.scaleX=sphere.scaleZ= 1+Math.sin(myTheta *
Math.PI / 180)*.7;
        //Elongate
        sphere.scaleY =myY;
    }

```

```

        //Render Scene
        renderer.renderScene(scene, camera, viewport);
    }
}
}

```

You should have gotten your figure to distort from a disk to an ellipse as shown below;

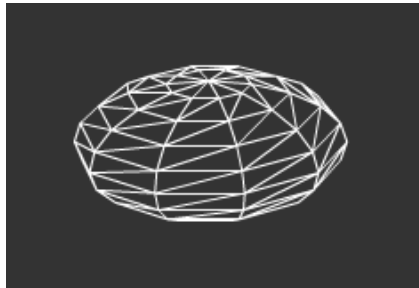


Figure 2-13 Undulating Sphere

In addition to adding motion in the example above, you removed the sphere positioning and set up the stage so that you movie sizes as you change the stage size. This is accomplished using the following commands:

```

//Initialize Stage
stage.align = StageAlign.TOP;
stage.scaleMode = StageScaleMode.NO_SCALE;

```

The selection of no scale in the scaleMode keeps the display objects from scaling. It keeps the content the same size as you have previously specified. By setting the align property of the stage to TOP_LEFT you force the stage to the (0,0) point. TOP_LEFT and NO_SCALE are generally chosen for most Papervision3D experiments that you commonly find on web blogs.

Setting the Stage

When the screen is resized, the Flash Player automatically adjusts the Stage contents to compensate. You can control this process using the Stage class.

The Stage class represents the main drawing area. For SWF content running in the browser, the Stage represents the entire area where Flash content is shown. A Flash application has only one Stage object. You can create and load multiple display objects into the display list, but the stage property of each display object refers to the same stage.

During the course of this book, you'll be primarily using three stage methods: align, scaleMode, and StageQuality. You'll now take a look at each one in more detail.

Align

There are many options for setting your align. The various options stage align are given below:

- StageAlign.TOP (Top, Center)

- StageAlign.BOTTOM (Bottom, Center)
- StageAlign.LEFT (Center, Left)
- StageAlign.RIGHT (Center, Right)
- StageAlign.TOP_LEFT (Top, Left)
- StageAlign.TOP_RIGHT (Top, Right)
- StageAlign.BOTTOM_LEFT (Bottom, Left)
- StageAlign.BOTTOM_RIGHT (Bottom, Right)

The best way to understand them is to just try different options and see what works best for your particular application.

Scale Mode

There are four possible scale modes, three of which are designed to scale contents to fit within their boundaries. The fourth one is NO_SCALE and keeps the contents from resizing.

- EXACT_FIT-scales the SWF proportionally
- NO_BORDER-determines whether the content can be partially cropped
- SHOW_ALL-determines whether a border appears
- NO_SCALE-contents maintain their defined size

Over all, having scaleMode set to StageScaleMode.NO_SCALE allows you to have greater control over how the screen contents adjust to the window resizing.

Quality

Setting stage quality from low to best can significantly tax your processor. Most Papervision3D applications set stage quality to low to reduce processor requirements. But there are times when the higher quality is needed. Below are the four different stage quality settings and what they mean:

- LOW-graphics are not anti-aliased, and bitmaps are not smoothed.
- MEDIUM-graphics are anti-aliased using a 2 x 2 pixel grid, but bitmaps are not smoothed.
- HIGH-graphics are anti-aliased using a 4 x 4 pixel grid, and bitmaps are smoothed if the movie is static.
- BEST-graphics are anti-aliased using a 4 x 4 pixel grid and bitmaps are always smoothed.

For most Papervision3D projects setting quality to low doesn't change the look of your graphics, but greatly enhance your application performance-sometimes as much as 50%. When it comes to optimizing a Papervision3D application for the web, quality should be the first item on your list to check-set it to low. In most cases, it will not change the appearance of your graphics.

An alternative to directly coding your viewport, camera, scene, and render loop, is to use BasicView class. BasicView is a great way to get a Papervision3D project up and running quickly and is discussed next.

Using BasicView (Hello Plane)

In a nutshell, BasicView acts as a wrapper class, combining the viewport, camera, scene and render together. It was developed to make creating a Papervision3D application easier. It's great for small projects (ones that don't require multiple cameras and viewports). You'll use it often in coming chapters. The example below is a rotating perlin plane.

```
package
{
    //Flash Imports
    import flash.display.BitmapData;
    import flash.events.Event;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    //Papervision3D Imports
    import org.papervision3d.materials.BitmapMaterial;
    import org.papervision3d.objects.primitives.Plane;
    import org.papervision3d.view.BasicView;

    //Declare the Hello Plane class that extends BasicView
    public class HelloPlane extends BasicView
    {
        //Declare the plane, bitmapdata, and material variables
        protected var myPlane:Plane;
        protected var planeBitmapData:BitmapData;
        protected var planeMaterial:BitmapMaterial;

        //Create the Hello Plane Constructor function
        public function HelloPlane()
        {
            super(1, 1, true, false);
            initPV3D();
        }
        //Set up stage and plane
        private function initPV3D():void
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
            opaqueBackground = 0;
            //Set up plane
            planeBitmapData = new BitmapData(512,256,false,0);
            planeBitmapData.perlinNoise(512,256,4, 123456, true,false);
            planeMaterial = new BitmapMaterial(planeBitmapData);
            myPlane = new Plane(planeMaterial,300,300, 10,10);
            scene.addChild(myPlane);
            startRendering();
        }
        //Override protected function
        override protected function onRenderTick(event:Event=null):void
        {
            //Rotate the hello plane primitive.
            myPlane.rotationY+=4;
            super.onRenderTick(event);
        }
    }
}
```

This short application uses **perlin noise** to fill a plane as shown below. Perlin noise is pseudo-random noise that creates fractal-like distortions on a data set. It has a large number of uses such as creating fire, marble, wood, clouds, and terrain. And you'll use it in upcoming chapters.

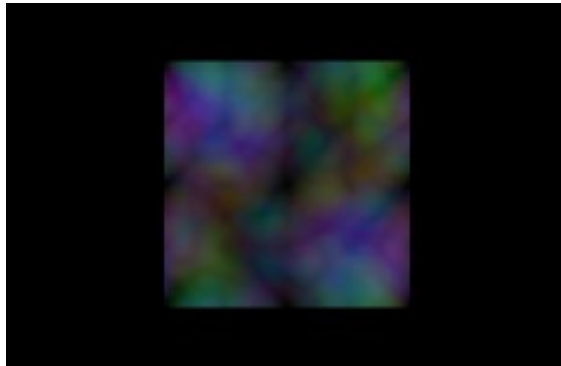


Figure 2-14 Perlin Noise Plane

As you run the rotating perlin plane example above, you'll notice that as the plane rotates 180 degrees out of view it will disappear for half a revolution. It hasn't disappeared. It's just invisible. This is because in order to see the plane (or any 3D object) its normal vectors from the surface must be facing you. When the plane disappears its normal vectors are facing away from you and you can't see it.

The key differences between using BasicView and what you have done before (creating the Papervision3DClass) are listed below:

- missing declaration of scene, camera, and viewport—which are encapsulated in the BasicView class
- addition of startRendering()-which is hidden in the AbstractView class and takes the place of the addEventListener(Event.ENTER_FRAME, myLoop) method
- addition of onRenderTick which is the event listener that corresponds to the startRendering method.

You should notice the use of the override statement, which as discussed in the section on Extending Your Class is used to extend the onRenderTick method. You've heard the term encapsulation a few time now let's encapsulate the entire Hello Plane example above.

Encapsulation

If you ever take a RMI Papervision3D Certification course under Ralph Hauwert (core team member and author of Great White), expect to see a heavy dose of encapsulation.

The power of Papervision3D is its use of OOP (object oriented programming). And if you've ever put a movie inside of a movie in Flash then you already understand how encapsulation works. But in this case, you're encapsulating classes (putting classes in classes), enabling you to: provide a friendlier object interface, remove clutter, and isolate functionality. Encapsulation simplifies programming allowing your program to become a collection of objects as opposed to a set of instructions.

```
package {  
    //Flash Imports  
    import flash.display.MovieClip;  
    import flash.display.StageAlign;  
    import flash.display.StageScaleMode;
```

```

//Not needed when in same directory as Main File
import HelloPlane;

//Papervision3D Imports
import org.papervision3d.view.BasicView;

//Extend Movie Class
public class Main extends MovieClip
{
    private var view:BasicView;
    public function Main()
    {
        stage.scaleMode = StageScaleMode.NO_SCALE;
        stage.align = StageAlign.TOP_LEFT;

        //Setup the "view"
        view = new HelloPlane();
        //Add the view to stage.
        addChild(view);

        //Start rendering.
        view.startRendering();
    }
}

```

The Main class above calls your HelloPlane class (developed previously with stage moved to main). It's much cleaner, and can be used to call any class. It's a convenient mechanism of which you should be aware. And if you take Ralph's class you'll need it. Now that you have seen the basics of BasicView, you'll apply it to your previous undulating sphere example.

Undulating Sphere using BasicView

Applying BasicView to the undulating sphere example only requires a few changes. The code still uses the initPV3D method in its constructor to set everything into motion, the createObjects method to create your sphere, and the startRendering method to start the rendering process. But it no longer needs to create a separate viewport, scene, camera, and renderer since that is handled by BasicView.

To get BasicView operational you need to make the following additions:

- **Super** comes from extending your class.
- **setStage** is used to orient your scene, but is not required for BasicView to run.
- **onRenderTick** is needed for BasicView to iterate your scene.

The addition of Super and onRenderTick were necessary to get BasicView operational. The setStage method was added to enhance the visual appeal. The finished code is listed below.

```

package
{
    //Flash imports
    import flash.display.BitmapData;
    import flash.events.Event;

    //Basic Papervision3d Engine
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

```

```

import org.papervision3d.materials.BitmapMaterial;
import org.papervision3d.objects.primitives.Sphere;
import org.papervision3d.view.BasicView;
import org.papervision3d.core.proto.MaterialObject3D;
import org.papervision3d.materials.WireframeMaterial;

// BasicViewTest extends BasicView, and automatically sets up
Scene, Camera, Renderer and Viewport for you.

public class BasicViewTest extends BasicView
{

protected var planeBitmapData:BitmapData;
protected var planeMaterial:BitmapMaterial;

//Define your sphere variable.

private var sphere:Sphere;
private var sphereMaterial:MaterialObject3D;
private var myTheta:Number=0;
private var myX:Number=0;
private var myY:Number=0;
private var myZ:Number=0;

//This allows for easy Papervision3d initialization.
//Call to the BasicView constructor
public function BasicViewTest()
{
//Width and Height are set to 1, since scaleToStage is set to
true, these will be overridden.

super(1, 1, true, false);
initPV3D();

}

private function initPV3D():void
{
//Set the Stage Scale
setStage();

//Color the background of this basicview / helloworld instance
black.
opaqueBackground = 0;

//Create the materials and primitives.
createObjects();

//Call the native startRendering function, to render every frame.
startRendering();

}

protected function setStage():void{
stage.align = StageAlign.TOP_LEFT;
stage.scaleMode = StageScaleMode.NO_SCALE;
}

//createObjects creates the needed primitives, and materials.

protected function createObjects():void
{
// Create a material for the sphere

```

```

sphereMaterial = new WireframeMaterial(0xFFFFFFFF);

// Create use wireframe material, radius 100, default 0,0,0
sphere = new Sphere(sphereMaterial, 100, 10, 10);

// Add Your Sphere to the Scene
scene.addChild(sphere);
}

// onRenderTick can be overridden so you can execute code on a per
render basis, using basicview.

override protected function onRenderTick(event:Event=null):void
{
//Rotate around x-axis
sphere.rotationX+=2;

//Increase angle
myTheta += 2;
//plot a circular orbit sin and cos
myX = Math.cos(myTheta * Math.PI / 180) * 100;
myZ = Math.sin(myTheta * Math.PI / 180) * 100;
//Set X and Z of spher
sphere.x = myX;
sphere.z = myZ;

//Fattening and scrinking
myY = 1+Math.cos(myTheta * Math.PI / 180)*.7;
sphere.scaleX=sphere.scaleZ= 1+Math.sin(myTheta * Math.PI /
180)*.7;
//Elongate
sphere.scaleY =myY;

//Render Scene
renderer.renderScene(scene, camera, viewport);
//Call the super.onRenderTick function, which renders the scene to the
viewport using the renderer and camera classes.
super.onRenderTick(event);
}
}
}

```

In using the BasicView class there is some savings in the code required to set up the application, and the renderer is now automatic. But you still need to set up your objects and materials. The down side to using BasicView is that you lose control over the render process. So even though the BasicView wrapper class is good for small experiments, when working with more complex applications you might find it more transparent to create your own scene using the earlier approach in this chapter.

Incorporating CS4

With the release of Flash Player 10, many improvements were made to the Flash Drawing API. Through out this book you'll leverage many of those improvements to enhance your Papervision3d applications. Below is a list of a number of those changes that have the potential of improving Papervision3d:

- Vectors-or typed arrays providing enhanced performance for the drawing API
- Perspective rendering directly incorporated into triangle drawing and 3D transformation translation

- Powerful drawing methods found in the `Flash.display.Graphics` classes such as `drawTriangles` and `drawPaths`.
- Bone animation capability through Inverse Kinematics
- UV mapping of textures onto 3D triangulated structures at runtime
- Non-zero winding rules for drawing a complex shape or gradients
- Shaders and Pixel Bender

In chapter 1, using CS4 you built a torus worm, carousel, and image ball. In the next chapter, along with learning about primitives, you'll build your first CS4 primitive.

Summary

In this chapter, you were introduced to the basics of `Papervision3d`, Painter's Algorithm, and the view frustum. You learned about culling and clipping and examined the guts of `Papervision3d`. You instantiated your first primitive (a sphere) and added a wireframe material. You extended your application using `BasicView`, and examined the different ways to run an application. In the next chapter, you'll build upon what you've learned by adding a new host of primitives to your repertoire.