3D IN FLASH 10

本ドキュメントは、friends of ED 社から発売されている「AdvancED ActionScript 3.0 Animation」（http://www.friendsofed.com/book.html?isbn=9781430216087）の Chapter 7 「3D IN FLASH 10」をヒム・カンパニー　永井勝則が自主的に日本語に訳したものです。

http://www.himco.jp/
knagai@himco.jp
(2009/11)

書籍「AdvancED ActionScript 3.0 Animation」は、
http://www.amazon.co.jp/AdvancED-ActionScript-Animation-Keith-Peters/dp/1430216085/ref=sr_1_1?ie=UTF8&s=english-books&qid=1258429428&sr=8-1
から購入できます。

3D IN FLASH 10

本書は中上級向けの書籍なので、3D に関するある程度の知識を持っていることが求められます。3次元には水平方向の x 軸、垂直方向の y 軸、さらに向こうとこっち方向の z 軸があります。Flash の原点は、少なくとも 2D ではスクリーンの左上にあります。y 軸は、通常の直交座標に慣れている方には、逆さま(y が下へいくほど大きくなる)に思えるかもしれませんが、すぐに慣れます。Flash 10 の 3D では、z 軸の値は、オブジェクトが"スクリーンの中に"向かっていく、つまりビューワ(見ている人)から遠ざかるほど高く(大きく)なります。別の言い方をすると、低い(小さい)z 位置を持つオブジェクトは、それより高い z 位置を持つオブジェクトの前面にあります(図 7-1 参照)。

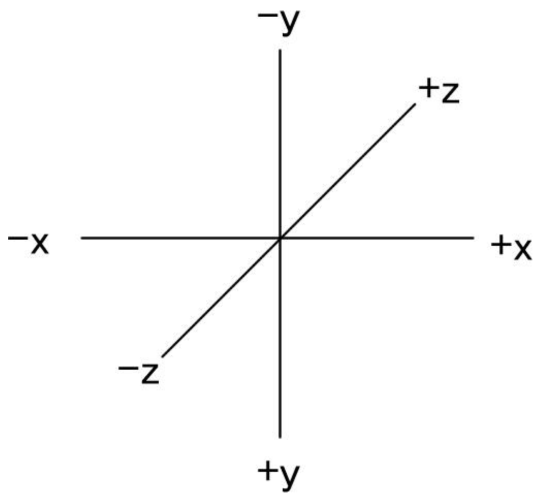

図 7-1:Flash 10 の 3D の座標

また Flash 10 の 3D の回転は、各軸でどのように動作するのかを理解しておくことも大切です。z 軸を回る回転は、正面から見て、角度の増加につれて時計回りに進みます。これも通常の感覚とは逆かもしれません。y 軸の回転も(上から見て)時計回りに進みます。x 軸の回転は、オブジェクトの左側から見て、時計回りに進みます(図 7-2 参照)。

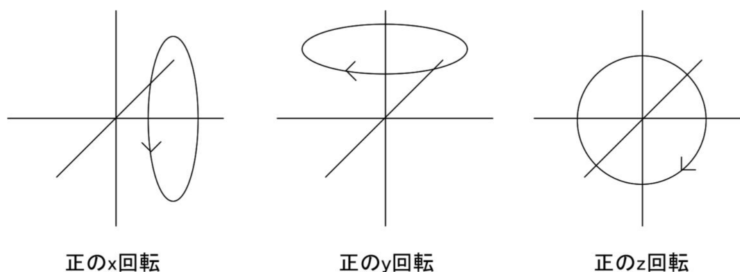

図 7-2:Flash 10 の 3D の回転

知っておくべきもう1つの重要なポイントは、Flash 10 の 3D 回転の角度は、ラジアンではなく度単位であることです。これは、ラジアンの操作に多くの三角関数を使用する 3D のプログラミングに習熟した方には奇妙に思えるかもしれません。しかし Flash の 3D は、エンジニアではなくデザイナー向けのオーサリングツール(Flash CS3 や CS4 のこと)に直結しています。デザイナーはたとえば、Math.PI/4 ラジアン角ではなく、45 度角といった値を扱いたいのです。したがってラジアンと度の間での変換方法を頭に入れておく必要があります。それは次のように行います。

```
radians = degrees * PI / 180
degrees = radians * 180 / PI
```

オーケー、用語を理解したところで、3D API(アプリケーション・プログラミング・インターフェイス)の観点から、Flash 10 で操作する必要のある事柄を見ていきましょう。実際には以下よりも多くの事柄が存在しますが、新しい 3D API 全体のポイントは実は z、rotationX、rotationY、rotationY という4つの新しいプロパティに尽きます。もちろんほかの要素についても見ていきますが、この4つのプロパティは、みなさんが基本的な 3D エフェクトを作成するときのよき相棒となります。

ではさっそく試してみましょう。次のクラスを作成しコンパイルします。

```
package {
	import flash.display.Shape;
	import flash.display.Sprite;
	import flash.display.StageAlign;
	import flash.display.StageScaleMode;
	import flash.events.Event;
	import flash.geom.PerspectiveProjection;
	import flash.geom.Point;

	public class Test3D extends Sprite
	{
		private var _shape:Shape;

		public function Test3D()
		{
			stage.align = StageAlign.TOP_LEFT;
			stage.scaleMode = StageScaleMode.NO_SCALE;
```

```
			_shape = new Shape();
			_shape.graphics.beginFill(0xff0000);
			_shape.graphics.drawRect(-100, -100, 200, 200);
			_shape.x = stage.stageWidth / 2;
			_shape.y = stage.stageHeight / 2;
			addChild(_shape);

			addEventListener(Event.ENTER_FRAME, onEnterFrame);
		}

		private function onEnterFrame(event:Event):void
		{
			_shape.rotationY += 2;
		}
	}
}
```

こりゃたまげたわい！　これが Flash のネイティブ 3D です。

消失点の設定

わたしは、みなさんが早く3D の世界に飛び込んでクールな 3D を作りたくてしょうがないことを知っています。しかし本節はぜひしっかり読んでください。ここには非常に重要な事柄が含まれています。3D を知っている方でも、Flash には“癖”があり、そこで起きていることを把握していないことには、途端に訳が分からなくなります。

前のクラスを FLA ファイルのドキュメントクラスとして使用し、Flash CS4 でコンパイルした場合には、図 7-3 のような結果が表示されます。

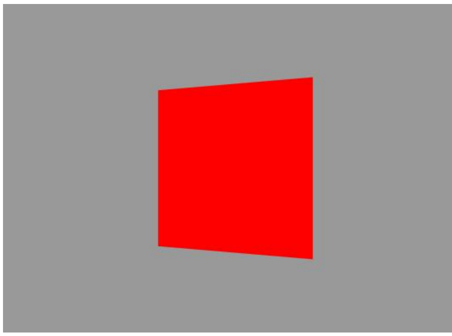

図 7-3：Flash CS4 でコンパイルした平面の回転

しかし Flex Builder で、Flash 10 用にコンパイルするか Flex 4 SDK を設定してコンパイルすると、結果は図 7-4 のようになります。

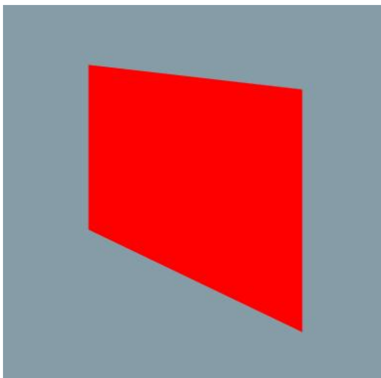

図 7-4：Flex Builder でコンパイルした平面の回転

これは左上方向に引き伸ばされているように見えます。この違いの理由は、ムービーのコンパイル方法にあるのではなく、パブリッシュのやり方が異なることと Flash の消失点の設定方法にあります。3D の消失点とは、すべてのオブジェクトが遠ざかっていくにつれ収束していく点を言います。ここで2本の線路が遠くの地平線まで延びている絵を引き合いに出すこともできますが、そんなことをしなくてもわたしの言いたいことはお分かりでしょう。3D のコードをゼロから書くときには、この消失点になる点を手動で選択し、すべてのオブジェクトがそれに向かって収束していくようにする必要があります。選択される消失点は通常はステージのセンターです。

Flash 10 の 3D はこの消失点をみなさんに代わって自動的に設定し、ステージセンターに設けます。しかしこれが行われるのは、SWF がロードされる瞬間の一度きりです。Flash でムービーをプレビューするときには、[ドキュメントプロパティ]ダイアログボックスで、[3D 遠近の角度を調整して、現在のステージ投影法を保持します]チェックボックスが選ばれていれば、消失点は[ドキュメントプロパティ]ダイアログボックスのサイズ設定にもとづいて自動的にステージセンターに設定されるので、問題はありません。

一方 Flex Builder では SWF のデフォルトサイズは 500 x 375 ピクセルなので、消失点は 250、187.5 になります。しかし stage.align を StageAlign.TOP_LEFT に、stage.scaleMode を stageScaleMode.NO_SCALE に設定するやいなや、ステージサイズが大きくなってしまうのです（この設定は、表示領域が大きくなった場合でも SWF のコンテンツが拡大しないようにするための一般的なプラクティスで、ただステージだけが大きくなります）。これは次のように、設定の前後で値を出力してみると分かります。

```
trace(stage.stageWidth, stage.stageHeight);           // 500, 375
stage.align = StageAlign.TOP_LEFT;
stage.scaleMode = StageScaleMode.NO_SCALE;
trace(stage.stageWidth, stage.stageHeight);           // 1440, 794
```

後の出力では、ブラウザのサイズにもとづいた異なる値が得られます。したがってステージサイズが変わり、スプライトは、720、397 といったステージの新しいセンターに配置されるのです。しかし消失点は 250、187.5 のまま変わっていません。これが問題なのです。

これを修正する最も簡単な方法は、SWF メタタグで幅と高さを明示的に設定する方法です。

```
[SWF(backgroundColor=0xffffff, width=800, height=800)]
```

これは消失点の計算前に実行されるので、消失点は 800、800 のセンターとして計算されます。ステージサイズは、align と scaleMode の設定後も変わらないので、スプライトは同じセンターに配置されます。

しかしおそらくみなさんは、ブラウザのウィンドウサイズに関係なくウィンドウを埋める可変的なステージサイズも欲しいでしょう。これはもう少し複雑で、PerspectiveProjection という新しいクラスを使用する必要があります。このクラスは、消失点など 3D の投影のレンダリング方法に関するさまざまな局面を制御します。Flash 10 の各表示オブジェクトは、3D のレンダリング方法を制御するために割り当てられる PerspectiveProjection を持つことができます。これは transform プロパティの perspectiveProjection プロパティに割り当てられます。たとえば s という名前のスプライトでは、次のようにアクセスします。

```
s. transform. perspectiveProjection
```

PerspectiveProjection クラスには、Point クラスのインスタンスである projectionCenter という名前のプロパティがあります。実はこれがわれわれが消失点と呼んできた点です。したがって表示オブジェクトの消失点をステージセンターに設定するには、次のようにします。

```
s. transform. perspectiveProjection. projectionCenter =
	new Point(stage.stageWidth / 2, stage.stageHeight / 2);
```

これによってこのオブジェクトに消失点が設定されますが、オブジェクトのすべての子の消失点も設定されます。ムービー内のすべてのオブジェクトの消失点を設定したい場合には、次のようにルートレベルで設定します。

```
root. transform. perspectiveProjection. projectionCenter =
	new Point(stage.stageWidth / 2, stage.stageHeight / 2);
```

次のクラスでは、ステージの align と scaleMode の設定後、これを実行しています。これによって前に見た問題は修正されます。

```
package {
	import flash.display.Shape;
	import flash.display.Sprite;
	import flash.display.StageAlign;
	import flash.display.StageScaleMode;
	import flash.events.Event;
	import flash.geom.PerspectiveProjection;
	import flash.geom.Point;

	[SWF(backgroundColor=0xffffff)]

	public class Test3D extends Sprite
	{
		private var _shape:Shape;

		public function Test3D()
		{
			stage.align = StageAlign.TOP_LEFT;
			stage.scaleMode = StageScaleMode.NO_SCALE;
			root.transform.perspectiveProjection.projectionCenter =
				new Point(stage.stageWidth / 2,
					stage.stageHeight / 2);
```

```
			_shape = new Shape();
			_shape.graphics.beginFill(0xff0000);
			_shape.graphics.drawRect(-100, -100, 200, 200);
			_shape.x = stage.stageWidth / 2;
			_shape.y = stage.stageHeight / 2;
			addChild(_shape);

			addEventListener(Event.ENTER_FRAME, onEnterFrame);
		}

		private function onEnterFrame(event:Event):void
		{
			_shape.rotationY += 2;
		}
	}
}
```

さらにもう一歩進めると、次のようにすることで、ステージのサイズが変更されてもセンターポイントを変えることができます。

```
package {
	import flash.display.Shape;
	import flash.display.Sprite;
	import flash.display.StageAlign;
	import flash.display.StageScaleMode;
	import flash.events.Event;
	import flash.geom.PerspectiveProjection;
	import flash.geom.Point;

	public class Test3D extends Sprite
	{
		private var _shape:Shape;

		public function Test3D()
		{
			stage.addEventListener(Event.RESIZE, onResize);
```

```
                stage.align = StageAlign.TOP_LEFT;
                stage.scaleMode = StageScaleMode.NO_SCALE;

                _shape = new Shape();
                _shape.graphics.beginFill(0xff0000);
                _shape.graphics.drawRect(-100, -100, 200, 200);
                _shape.x = stage.stageWidth / 2;
                _shape.y = stage.stageHeight / 2;
                addChild(_shape);

                addEventListener(Event.ENTER_FRAME, onEnterFrame);
            }


            private function onResize(event:Event):void
            {
                root.transform.perspectiveProjection.projectionCenter =
                    new Point(stage.stageWidth / 2,
                        stage.stageHeight / 2);
                if(_shape != null)
                {
                    _shape.x = stage.stageWidth / 2;
                    _shape.y = stage.stageHeight / 2;
                }
            }

            private function onEnterFrame(event:Event):void
            {
                _shape.rotationY += 2;
            }
        }
}
```

最初のアクションとして resize イベントの監視から始めているので、われわれがステージ設定を変えてステージのサイズが変わるとすぐに onResize()メソッドが呼び出され、即座に投影の中心が設定されます。またこのメソッドは、ユーザーがブラウザをリサイズしてステージサイズを変更するときにも呼び出されます。ここでは

また、シェイプが確実にステージのセンターに来るように再配置しています。しかし onResize()が初めて呼び出されるのはシェイプの作成前なので、シェイプを確実に存在させておく必要があります。このためここでは条件文を使っています。

簡易性を保つために、本章では今後、1つめのステージサイズの設定の解決方法を使っていきます。

オーケー、以上がムービーの投影を適切に見せるために知っておくべき重要なポイントです。では 3D で実現できるほかのことについて見ていきましょう。

3D の位置取り

この意味はいたって明瞭です。みなさんにはおそらく、x と y 軸でのオブジェクトの位置の変更方法を解説する必要はないでしょう。z 軸での位置の変更も簡単です。次の Position3D クラスでは、シェイプが遠くとこちらを繰り返し行きつ戻りつするサイン波を設定しています。またシェイプはマウスの x 位置と y 位置に追随します。

```
package
{
	import flash.display.Shape;
	import flash.display.Sprite;
	import flash.events.Event;

	[SWF(width=800, height=800, backgroundColor=0xffffff)]
	public class Position3D extends Sprite
	{
		private var _shape:Shape;
		private var _n:Number = 0;

		public function Position3D()
		{
			_shape = new Shape();
			_shape.graphics.beginFill(0x00ff00);
			_shape.graphics.drawRect(-100, -100, 200, 200);
			_shape.graphics.endFill();
			addChild(_shape);
```

```
			addEventListener(Event.ENTER_FRAME, onEnterFrame);
		}

		private function onEnterFrame(event:Event):void
		{
			_shape.x = mouseX;
			_shape.y = mouseY;
			_shape.z = 10000 + Math.sin(_n += .1) * 10000;
		}
	}
}
```

思うに、このサンプルの価値を低める重要なことは、z 位置を変更すると、表示オブジェクトの x と y はもはやスクリーンの座標を直接参照しなくなり、3D 空間の座標を参照するということです。シェイプの x と y は、マウスを移動しないと変わりませんが、スクリーン上では変化しています。オブジェクトの x と y は、z がゼロに等しいときのみ、スクリーンの x と y に一致します。これは、Flash は z がゼロより小さいときオブジェクトを拡大し、z がゼロより大きいときオブジェクトを縮小しますが、z がゼロのときにはそのスケールが 100%になるからです。

深度のソート

複数のオブジェクトを作成し 3D 空間に配置するようになると、遠くにあるオブジェクト（高い z 値）が、それより近くにあるオブジェクトの前面に表示される場面に出くわすようになります。みなさんはそれをうまく処理してくれるプロパティやメソッドが欲しいと思われていると、わたしは想像しますが、残念ながら、オブジェクトを正確にソートしてくれるプロパティやメソッドはありません。

Flash 10 の 3D API は、投影の拡大縮小と歪曲を個々の表示オブジェクトで処理し、それは、そのオブジェクトが子をいくつ持っていてもその子に及びます。しかしその処理はスクリーンに描画される重なり順には影響を与えません。これは依然として Flash 9 での 2D オブジェクトと同じ方法で処理され、addChild()メソッドで表示リストに置かれたオブジェクトはどれも、同じコンテナにそれまでに追加されたオブジェクトの前面に表示されます。これを変更する唯一の方法は、addChild()や addChildAt()、swapChildren()、removeChild()といった表示リストを管理するさまざまなメソッドを駆使する方法です。また表示オブジェクトコンテナにはソートするメソッドがないので、深度のソートはすべて手動で行う必要があります。

この問題を解決する方法を探るため、まずはこの問題を示すサンプルを見てみましょう。木がたくさんある森はどうでしょう？

```
package
{
	import flash.display.Shape;
	import flash.display.Sprite;
	import flash.display.StageAlign;
	import flash.display.StageScaleMode;

	[SWF(width=800, height=800, backgroundColor = 0xccffcc)]
	public class DepthSort extends Sprite
	{
		public function DepthSort()
		{
			stage.align = StageAlign.TOP_LEFT;
			stage.scaleMode = StageScaleMode.NO_SCALE;

			for(var i:int = 0; i < 500; i++)
			{
				var tree:Shape = new Shape();
				tree.graphics.beginFill(Math.random() * 255 << 8);
				tree.graphics.lineTo(-10, 0);
				tree.graphics.lineTo(-10, -30);
				tree.graphics.lineTo(-40, -30);
				tree.graphics.lineTo(0, -100);
				tree.graphics.lineTo(40, -30);
				tree.graphics.lineTo(10, -30);
				tree.graphics.lineTo(10, 0);
				tree.graphics.lineTo(0, 0);
				tree.graphics.endFill();
				tree.x = Math.random() * stage.stageWidth;
				tree.y = stage.stageHeight - 100;
				tree.z = Math.random() * 10000;
				addChild(tree);
			}
		}
	}
```

```
}
```

ここではシェイプを一気に作成し、描画 API を使って緑のランダムな陰影のついた木を作成しています。それぞれの木はランダムな x、z 位置に置いています。Flash デザインとしては全然大したことはありませんが、これで目的は果たしています。その見栄えは図 7-5 に示すようにはなはだよろしくありません。

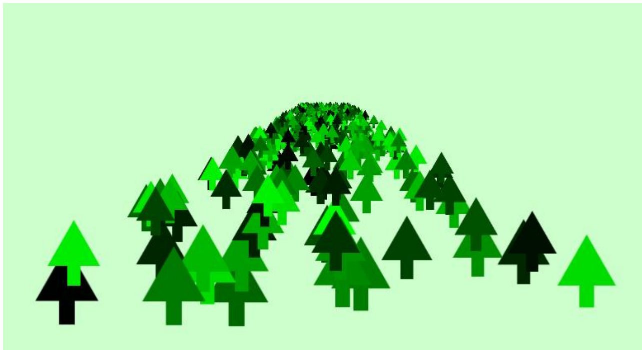

図 7-5：投影は問題ないが、深度ソートされていない

表示リストのソートはできませんが、配列のソートは可能です。したがってここで行ったように各木を表示リストに追加するのではなく、配列に入れていきましょう。すると配列をソートし、適切な順番で木を表示リストに追加することができます。より高い z 値を持つものを先に追加し、それより低い z 値を持つものを後で追加するのです。

```
package
{
	import flash.display.Shape;
	import flash.display.Sprite;
	import flash.display.StageAlign;
	import flash.display.StageScaleMode;

	[SWF(width=800, height=800, backgroundColor = 0xccffcc)]
	public class DepthSort extends Sprite
	{
		private var _trees:Array;

		public function DepthSort()
		{
			stage.align = StageAlign.TOP_LEFT;
```

```
				stage.scaleMode = StageScaleMode.NO_SCALE;

				_trees = new Array();

				for(var i:int = 0; i < 500; i++)
				{
					var tree:Shape = new Shape();
					tree.graphics.beginFill(Math.random() * 255 << 8);
					tree.graphics.lineTo(-10, 0);
					tree.graphics.lineTo(-10, -30);
					tree.graphics.lineTo(-40, -30);
					tree.graphics.lineTo(0, -100);
					tree.graphics.lineTo(40, -30);
					tree.graphics.lineTo(10, -30);
					tree.graphics.lineTo(10, 0);
					tree.graphics.lineTo(0, 0);
					tree.graphics.endFill();
					tree.x = Math.random() * stage.stageWidth;
					tree.y = stage.stageHeight - 100;
					tree.z = Math.random() * 10000;
					_trees.push(tree);
				}

				_trees.sortOn("z", Array.NUMERIC | Array.DESCENDING);
				for(i = 0; i < 500; i++)
				{
					addChild(_trees[i] as Shape);
				}
			}
		}
	}
```

結果は図 7-6 に示すように決して芸術的ではありませんが、実際の森と同じように、少なくとも遠くの木は近い木の後ろにあります。

図 7-6：深度ソートを行った森

3D コンテナ

この API をいじり始めたとき、Flash 10 の 3D に関してわたしを本当にハッピーにさせたことの１つは、表示オブジェクトコンテナは、それが変形（変換）されると、その子も変形させるという事実です。言い方を変えると、表示オブジェクトをスプライトに追加しそのコンテナスプライトを移動させると、平板に見えるスプライトが 3D 内を単一のオブジェクトとして移動するのではないのです。各子も実際に変形されるので、各子が 3D 空間を独立して移動しているように見えるのです。

これは文字で述べるよりも実際に見た方が理解の早い例の１つなので、次のサンプルでこれを示しましょう。無論正方形などのシェイプ以外のものを使っていけない理由はありません。テキストフィールドも表示オブジェクトなので、これまでとまったく同じ方法で 3D 内を移動させることができます。次のサンプルではスプライトを作成し、その中にランダムな文字を持ったたくさんのテキストフィールドを投げ込みます。そしてそのスプライトを移動させます。

```
package
{
		import flash.display.Sprite;
		import flash.display.StageAlign;
		import flash.display.StageScaleMode;
		import flash.events.Event;
		import flash.text.TextField;
		import flash.text.TextFormat;

		[SWF(width=800, height=800, backgroundColor=0xffffff)]
		public class Container3D extends Sprite
		{
				private var _sprite:Sprite;
```

```
		private var _n:Number = 0;

		public function Container3D()
		{
			stage.align = StageAlign.TOP_LEFT;
			stage.scaleMode = StageScaleMode.NO_SCALE;

			_sprite = new Sprite();
			_sprite.y = stage.stageHeight / 2;

			for(var i:int = 0; i < 100; i++)
			{
				var tf:TextField = new TextField();
				tf.defaultTextFormat = new TextFormat("Arial", 40);
				tf.text = String.fromCharCode(65 +
						Math.floor(Math.random() * 25));
				tf.selectable = false;
				tf.x = Math.random() * 300 - 150;
				tf.y = Math.random() * 300 - 150;
				tf.z = Math.random() * 1000;
				_sprite.addChild(tf);
			}

			addChild(_sprite);

			addEventListener(Event.ENTER_FRAME, onEnterFrame);
		}

		private function onEnterFrame(event:Event):void
		{
			_sprite.x = stage.stageWidth / 2 + Math.cos(_n) * 200;
			_n += .05;
		}
	}
}
```

各テキストフィールドは、その親スプライト内の３次元にランダムに配置されます。スプライトは単に x 軸を左右に繰り返し移動するだけですが、文字が 3D 内を往復して動いているパララックス（視差）効果が見て取れます（遠くのものほどゆっくり、近いものほど速く動いて見えます）。

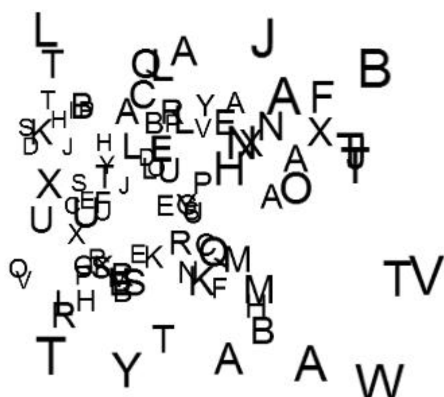

図 7-7：3D コンテナの移動。実際に動かしてみないとこのよさは分からない！

3D の回転

3D 空間では移動のほかに、表示オブジェクトをすべての軸で回転させることができます。われわれはすでに本章の冒頭で、シェイプを y 軸で回転させる簡単なサンプルを見ているので、x や z 軸についてこれと同じサンプルを再度示す必要はないでしょう。なぜならみなさんはおそらく独力で理解できるでしょうし、もっと先に進んで、３軸すべてを一度に回転させるようなクラスを作成されているかもしれないからです（とはいえ、そうでない方は x や z 軸について自分で試してみてくださいね）。

準備が整ったら、内部に表示オブジェクトを持ったコンテナの回転に進みましょう。われわれは１つめの実験を作成した後修正を加えていきます。シェイプはスプライトの中に置きます。

```
package {
		import flash.display.Shape;
		import flash.display.Sprite;
		import flash.events.Event;

		[SWF(width=800, height=800, backgroundColor = 0xffffff)]
		public class RotateAndPosition extends Sprite
		{
				private var _holder:Sprite;
```

```
		public function RotateAndPosition()
		{
			_holder = new Sprite();
			_holder.x = stage.stageWidth / 2;
			_holder.y = stage.stageHeight / 2;
			addChild(_holder);

			var shape:Shape = new Shape();
			shape.graphics.beginFill(0xff0000);
			shape.graphics.drawRect(-100, -100, 200, 200);
			_holder.addChild(shape);

			addEventListener(Event.ENTER_FRAME, onEnterFrame);
		}

		private function onEnterFrame(event:Event):void
		{
			_holder.rotationY += 2;
		}
	}
}
```

これによって前と同じようにただ正方形が回転する結果が得られます。次はこの正方形をコンテナ内で移動させてみましょう。まずは x 軸を試してみます。コンストラクタのコードに次の太字の行を追加します。

```
var shape:Shape = new Shape();
shape.x = 200;
shape.graphics.beginFill(0xff0000);
shape.graphics.drawRect(-100, -100, 200, 200);
_holder.addChild(shape);
```

すると単なる回転ではなく、中心の回りを周回するような動きに変わります。さらに z 軸でシェイプを移動させておくとまた別の結果が得られます。

```
var shape:Shape = new Shape();
```

```
shape.z = 200;
shape.graphics.beginFill(0xff0000);
shape.graphics.drawRect(-100, -100, 200, 200);
_holder.addChild(shape);
```

実に見事です。つづいて正方形を1つ加えましょう。

```
package {
	import flash.display.Shape;
	import flash.display.Sprite;
	import flash.events.Event;

	[SWF(width=800, height=800, backgroundColor = 0xffffff)]
	public class RotateAndPosition extends Sprite
	{
		private var _holder:Sprite;

		public function RotateAndPosition()
		{
			_holder = new Sprite();
			_holder.x = stage.stageWidth / 2;
			_holder.y = stage.stageHeight / 2;
			addChild(_holder);

			var shape1:Shape = new Shape();
			shape1.z = 200;
			shape1.graphics.beginFill(0xff0000);
			shape1.graphics.drawRect(-100, -100, 200, 200);
			_holder.addChild(shape1);

			var shape2:Shape = new Shape();
			shape2.z = -200;
			shape2.graphics.beginFill(0xff0000);
			shape2.graphics.drawRect(-100, -100, 200, 200);
			_holder.addChild(shape2);
```

```
                addEventListener(Event.ENTER_FRAME, onEnterFrame);
            }

            private function onEnterFrame(event:Event):void
            {
                _holder.rotationY += 2;
            }
        }
}
```

正方形の一方は 200 の z 値に、片方は-200 の z 値に置いています。コンテナを回転させると、互いが片方の周りを回っているように見えます。しかし rotationY の回転だけに限っておく理由はありません。onEnterFrame()のコードに別の軸での回転を加えてみましょう。

```
private function onEnterFrame(event:Event):void
{
        _holder.rotationY += 2;
        _holder.rotationX += 1.5;
}
```

ちぇっ！ これはいささか簡単すぎましたね。次は正方形をもっと追加しましょう。今度は x 軸での左と右方向、90 度だけ回転させて追加します。

```
var shape3:Shape = new Shape();
shape3.x = 200;
shape3.rotationY = 90;
shape3.graphics.beginFill(0xff0000);
shape3.graphics.drawRect(-100, -100, 200, 200);
_holder.addChild(shape3);

var shape4:Shape = new Shape();
shape4.x = -200;
shape4.rotationY = -90;
shape4.graphics.beginFill(0xff0000);
shape4.graphics.drawRect(-100, -100, 200, 200);
_holder.addChild(shape4);
```

これで互いが周回し合うように見える4つの壁ができ上がりました。これで終わりますか？　みなさんはおそらくわたしよりもずっと先におられるでしょうが、次のように床と天井を加えてみましょう。

```
var shape5:Shape = new Shape();
shape5.y = 200;
shape5.rotationX = 90;
shape5.graphics.beginFill(0xff0000);
shape5.graphics.drawRect(-100, -100, 200, 200);
_holder.addChild(shape5);

var shape6:Shape = new Shape();
shape6.y = -200;
shape6.rotationX = -90;
shape6.graphics.beginFill(0xff0000);
shape6.graphics.drawRect(-100, -100, 200, 200);
_holder.addChild(shape6);
```

結果は図 7-8 に示すようになります。

図 7-8：キューブの回転！

どうです、見事なもんでしょ？　そして驚くほど簡単だと思われませんか？　三角関数は1行たりとも使っていません。わたしにはみなさんが今考えていることが分かります。それはフォトキューブ（写真の立方体）でし

ょう？　そのものズバリではないかもしれませんが、みなさんはたぶん、この赤ばかりの正方形は面白くないので、何か別のものが必要だ、と思っておられるでしょう。しかし実は、カラーなどの変更に手をつけてしまうと、わたしがここまで作ってきた錯覚が水泡に帰すことになるのです。分かりました。それを試してみましょう。各正方形を別々のカラーにするには、beginFill()呼び出しの 16 進数値を変更します。希望する場合には次のようにランダムにすることもできます。

```
shape1.graphics.beginFill(Math.random() * 0xffffff);
```

結果は図 7-9 のようになります。

図 7-9：アドビさん、問題が起きてしまいました。。。

正方形はもう赤ではないので、問題が露見してしまいました。この画面写真でははっきりとしていないかもしれませんが、ここでは背後にあるべきキューブの一部が前面に出てきているのです。わたしは、みなさんがこれは深度ソートの問題だとすぐに気づかれているだろうと思います。もちろんわれわれは深度ソートとその処理方法についてすでに学んでいるので、すぐにこれを直すことができます。次の RotateAndPosition2 クラスは、シェイプの z プロパティをソートし、その順番で表示リストにシェイプを追加するという、前に学んだ方法で深度の問題を解決しようとするクラスです。ここでは makeShape()メソッドを追加してコードの重複をなくしています。

```
package
{
	import flash.display.Shape;
	import flash.display.Sprite;
	import flash.events.Event;
```

```
[SWF(width=800, height=800, backgroundColor = 0xffffff)]
public class RotateAndPosition2 extends Sprite
{
	private var _holder:Sprite;
	private var _shapes:Array;

	public function RotateAndPosition2()
	{
		_holder = new Sprite();
		_holder.x = stage.stageWidth / 2;
		_holder.y = stage.stageHeight / 2;
		addChild(_holder);

		var shape1:Shape = makeShape();
		shape1.z = 200;

		var shape2:Shape = makeShape();
		shape2.z = -200;

		var shape3:Shape = makeShape();
		shape3.x = 200;
		shape3.rotationY = 90;

		var shape4:Shape = makeShape();
		shape4.x = -200;
		shape4.rotationY = -90;

		var shape5:Shape = makeShape();
		shape5.y = 200;
		shape5.rotationX = 90;

		var shape6:Shape = makeShape();
		shape6.y = -200;
		shape6.rotationX = 90;

		_shapes = [shape1, shape2, shape3, shape4, shape5, shape6];
```

```
                sortShapes();

                addEventListener(Event.ENTER_FRAME, onEnterFrame);
            }

            private function makeShape():Shape
            {
                var shape:Shape = new Shape();
                shape.graphics.beginFill(Math.random() * 0xffffff);
                shape.graphics.drawRect(-100, -100, 200, 200);
                _holder.addChild(shape);
                return shape;
            }

            private function sortShapes():void
            {
                _shapes.sortOn("z", Array.NUMERIC |
                                        Array.DESCENDING);
                for(var i:int = 0; i < _shapes.length; i++)
                {
                    _holder.addChildAt(_shapes[i] as Shape, i);
                }
            }

            private function onEnterFrame(event:Event):void
            {
                _holder.rotationY += 2;
                _holder.rotationX += 1.5;
                sortShapes();
            }
        }
}
```

コードの整理に加え、すべてのシェイプを含む_shapes という名前の配列も作成しています。sortShapes()メソッドは各回転の後に呼び出されます。このメソッドはシェイプの配列を z でソートし、_holder スプライトに適切な z 順で各シェイプを追加し直します。

しかしこれをプレビューすると、実は何も解決されていないことが分かります。問題は、コンテナ内部の z 軸でエレメントをソートしていることにあります。したがってたとえ z について正しくソートされていても、コンテナが回転して前後がひっくり返ると途端にすべてが逆の順番になってしまうのです。われわれに必要なことは、正方形が_holder の外から見える順でソートすることです。言い換えると、オブジェクト A がオブジェクト B より低い z 値を持っている場合、コンテナが回転すると z 軸に関して"逆向き"になるので、オブジェクト A はオブジェクト B の背後に表示されるべきだ、ということです。

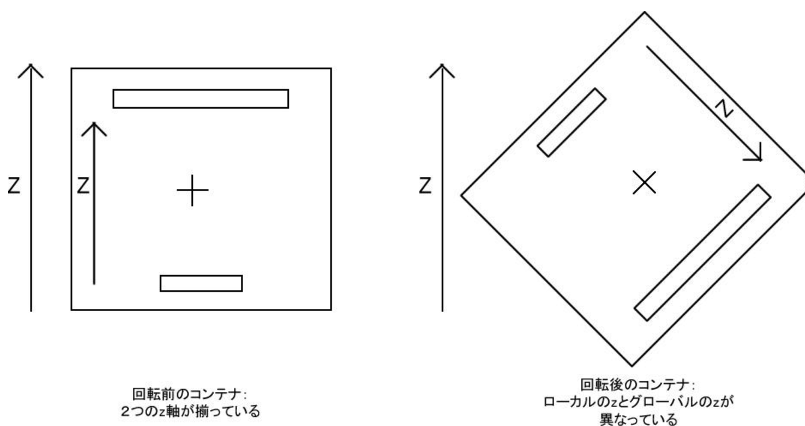

図 7-10：3D コンテナを回転させたときの影響

そのためには、コンテナ内のローカル座標をワールド（つまりステージやルート）座標に変換し、それにもとづいてソートするカスタムのソート関数を記述する必要があります。Array.sort()メソッドにはパラメータとして関数を渡すことができます。この関数はソート中、２つのオブジェクトを渡して複数回呼び出します。この関数は、配列内で１つめのオブジェクトが２つめのオブジェクトの前に配置されるべきときには負数を返し、１つめのオブジェクトが２つめのオブジェクトの後に配置されるべきときには正数を返し、そしてそのままにしておくべきときにはゼロを返すようにします。

したがって次に必要なのは、ローカル座標をワールド座標に変換する方法です。これには、多くの複雑な三角関数をともなう手動による座標回転など、たくさんの方法がありますが、ラッキーなことに、Flash 10 には flash.geom.Matrix3D という新しいクラスがあり、このクラスには 3D 座標の操作に関するあらゆる種類の便利なメソッドが含まれています。このクラスを使う場合でも、われわれが行おうとしていることを実現する方法はおそらく複数種類あるでしょう。わたしには以下に示す方法が最良かどうかは分かりませんが、十分に用をなし、難しすぎて手が出ないという方法でもありません。

その方法では、Matrix3D クラスの deltaTransformVector()という名前のメソッドを利用します。このメソッドは基本的に、3D の点（Vector3D オブジェクトに保持される）を取り、それに 3D 行列の回転と拡大縮小の部分を適用します。ごく簡単に言うと、すべてのシェイプの位置を、コンテナの回転に準じて回転さ

せ、グローバルの 3D 空間のどこにそれがあるのかを教えてくれます。

初めに必要なのは、コンテナの回転を表す Matrix3D です。これは次のように入力することで得られます。

```
container.transform.matrix3D
```

次いで deltaTransformVector()メソッドに点の 3D 位置を渡してこれを呼び出します。すると回転後の位置が得られます。

```
rotatedPosition = _holder. transform.matrix3D. deltaTransformVector(
                                                    originalPosition)
```

２つの別々のシェイプの位置にこれを行うと、グローバルな視点から見たときのように、z 軸に関してどちらが遠くにあるのかが分かります。残された最後のパズルは、表示オブジェクトの位置を表す Vector3D の取得方法です。これはオブジェクトの x、y、z プロパティを使用することですぐに作成できますが、ここではたまたますでに存在しています。

```
displayObject.transform.Matrix3D.position
```

これで２つの表示オブジェクトのグローバル座標を取得するために必要なものは全部揃いました。

```
var posA:Vector3D = objA.transform.matrix3D.position;
posA = _holder.transform.matrix3D.deltaTransformVector(posA);
var posB:Vector3D = objB.transform.matrix3D.position;
posB = _holder.transform.matrix3D.deltaTransformVector(posB);
```

このコードを使うと、ソートする比較関数が作成できます。これは２つの表示オブジェクトのどっちが前面にあるのかを判定します。

```
private function depthSort(objA:DisplayObject, objB:DisplayObject):int
{
	var posA:Vector3D = objA.transform.matrix3D.position;
	posA = _holder.transform.matrix3D.deltaTransformVector(posA);
	var posB:Vector3D = objB.transform.matrix3D.position;
	posB = _holder.transform.matrix3D.deltaTransformVector(posB);
	return posB.z - posA.z;
```

```
}
```

回転後のグローバルな視点から見て、objA が objB より遠くにある場合には、この関数は負数を返します。これは、配列内で objA は objB より前にソートされるべきだということを示しています。これは sortShapes()メソッドに簡単に組み込めます。

```
private function sortShapes():void
{
		_shapes.sort(depthSort);
		for(var i:int = 0; i < _shapes.length; i++)
		{
			_holder.addChildAt(_shapes[i] as Shape, i);
		}
}
```

訳者注:
クラスの冒頭で、flash.display.DisplayObject と flash.geom.Vector3D をインポートしておく必要があります。また_holder の z プロパティを設定しないと matrix3D オブジェクトが自動生成されない(depthSort()が動作しない)ので、コンストラクタで、_holder.z = 0;を加えておきます。

これでシェイプの回転はうまくソートされ、図 7-11 に示すようにナイスな 3D オブジェクトが作成できました。

図 7-11:コンテナを回転させても適切な深度ソートが行われている

これに少し手を加えると、メリーゴーランド風にレイアウトした 3D が作成できます。これは通常(逆に陳腐かもしれませんが)、ナビゲーションやイメージギャラリーの表示に使用されます。次のサンプルでは実際にイメージはロードしませんが、スプライトをローダーに置き換え、URL のリストを加えることは簡単です。

```
package
{
	import flash.display.DisplayObject;
	import flash.display.Sprite;
	import flash.events.Event;
	import flash.geom.Vector3D;

	[SWF(width=800, height=800, backgroundColor = 0xffffff)]
	public class Carousel extends Sprite
	{
		private var _holder:Sprite;
		private var _items:Array;
		private var _radius:Number = 200;
		private var _numItems:int = 10;

		public function Carousel()
		{
			_holder = new Sprite();
			_holder.x = stage.stageWidth / 2;
			_holder.y = stage.stageHeight / 2;
			_holder.z = 0;
			addChild(_holder);

			_items = new Array();
			for(var i:int = 0; i < _numItems; i++)
			{
				var angle:Number = Math.PI * 2 / _numItems * i;
				var item:Sprite = makeItem();
				item.x = Math.cos(angle) * _radius;
				item.z = Math.sin(angle) * _radius;
				item.rotationY = -360 / _numItems * i + 90;
				_items.push(item);
			}
			sortItems();
```

```
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function makeItem():Sprite
{
            var item:Sprite = new Sprite();
            item.graphics.beginFill(Math.random() * 0xffffff);
            item.graphics.drawRect(-50, -50, 100, 100);
            _holder.addChild(item);
            return item;
}

private function sortItems():void
{
            _items.sort(depthSort);
            for(var i:int = 0; i < _items.length; i++)
            {
                        _holder.addChildAt(_items[i] as Sprite, i);
            }
}

private function depthSort(objA:DisplayObject,
                                          objB:DisplayObject):int
{
            var posA:Vector3D = objA.transform.matrix3D.position;
            posA =
            _holder.transform.matrix3D.deltaTransformVector(posA);
            var posB:Vector3D =
            objB.transform.matrix3D.position;
            posB =
            _holder.transform.matrix3D.deltaTransformVector(posB);
            return posB.z - posA.z;
}

private function onEnterFrame(event:Event):void
{
```

```
                _holder.rotationY += (stage.stageWidth / 2 - mouseX) * .01;
                _holder.y += (mouseY - _holder.y) * .1;
                sortItems();
            }
        }
}
```

大きな変更箇所は太字で示しています。イメージを最初に配置し回転させる部分と、onEnterFrame()メソッド内のコンテナを動かす部分です。ここでは各正方形を手動で配置する代わりに、ループでそれを行い、Math.PI * 2 ラジアン（360 度）をアイテム数（_numItems、正方形の数）で割ったものに、現在のアイテム数（i）を掛けることで角度を求めています。その角度を半径とともに三角関数で使用すると、各アイテムの x と z 位置が得られます。次いで似たような計算を rotaeionY プロパティで行っていますが、ここでは度を直接使っています。enterFrame ハンドラでは、コンテナを y 軸でマウス位置にもとづいて回転させるとともに、y 軸でマウスに追随する上下の移動を行っています。

図 7-12 はその結果を示しています。

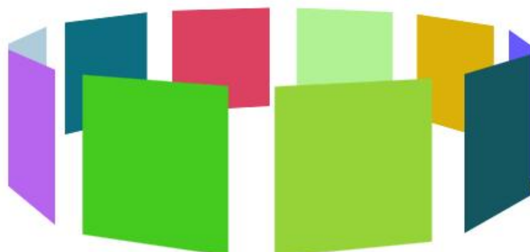

図 7-12：3D メリーゴーランド

3D における位置取りと回転の方法に関する基本的な考えが分かったので、次は 3D の見かけを調整する方法を見ていきましょう。

視野（field of view）と焦点距離（focal length）

当然のことながら、フラットな画面でイメージを見ているときにはどんなイメージであっても、実際に見ているのは２次元のイメージです。3D で事物をレンダリングするプログラムはいくつものトリックを使って、2D 平面に３次元の錯覚を作ります。これらのさまざまなトリックは透視の部類に入ります。

透視のトリックの１つは、遠くにあるはずのものを、それより近くに表示させることです。われわれはこれを、前節深度ソードを行ったときに扱いました。もう1つのトリックは、遠くのあるものを、霧がかったように次第に消していくことです。同様に、ある奥行きにあるオブジェクトに焦点を合わせ、それより近いまたは遠いオブジェクトの焦点をぼかすこともできます。これは被写界深度と呼ばれます。

しかし間違いなく最も影響のあるトリックは、遠くにある事物を小さくし、遠ざかるにつれさらに小さくしながら消失点に近づけていくことです。もちろん、深度ソートも３次元の表現として極めて重要です。間違った深度ソートは、すでに見たように 3D の錯覚を台無しにしてしまいます。しかし深度ソートのみを行い、その深度にしたがってオブジェクトの拡大縮小を行わない場合には、これもまた 3D の実感を大いにそぐことになります。

最も大きな疑問は、オブジェクトが近づくまたは遠ざかるにつれて、どれだけそれを拡大または縮小するのか、ということです。ありがたいことにこの疑問は、パーソナルコンピュータが存在するずっと前から存在し、画家やエンジニア、写真家によって問われ答えが出されています。それは光学の問題であり、要は目やカメラのレンズの働き方に帰着します。写真を撮ったことのある方なら、レンズには広角や望遠のレンズ（そしてその間の範囲のレンズも）があることをご存知でしょう。超広角の“魚眼”レンズというものもあります。

広角レンズは広い視野（field of view）を持っています。別の言い方をすると、このレンズの前で円錐を投影したとすると（レンズに直角に円錐が突き刺さっているような状態）、レンズがカバーする領域は幅の広い円錐となって“見え”、領域の多くがカバーされます。魚眼レンズではほぼ 180 度の領域を見ることができます。一方望遠レンズではこの円錐の幅が狭くなり、レンズの前の幅の狭い世界がカバーされます。

この視野に加え、焦点距離（focal length）という概念があります。これはレンズの中心から焦点（focal point、レンズを通って光線が集まる点）までの距離を言います。図 7-13 を見てください。

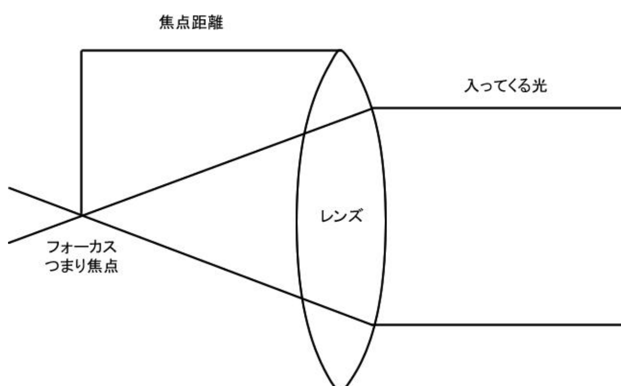

図 7-13：レンズの焦点距離

焦点距離はアドビのヘルプファイルでは少し異なる説明がされていますが、結果的には同じ概念です。焦

点距離と視野は密接に関連しており、どれだけの拡大縮小と歪曲で表示するかを決定します。広い視野では、広角レンズで見たときのように、焦点距離は短く、拡大が大きくなります（魚眼レンズで撮られた写真の多くが歪曲しているのはこのためです）。狭い視野では、望遠レンズで見たときのように、焦点距離が非常に長くなり、歪曲がかなり減少します。これのよい例が野球の試合を外野から撮った写真で、バッターは、ピッチャーよりも遠くにいるにもかかわらず、ピッチャーとほぼ同じサイズで見えます。人間の知覚では、この種の歪曲がほとんどまったくない写真を見ると、実際には歪曲があるように見え、バッターの方が大きい印象を持ちます。図 7-14 と 7-15 では視野と焦点距離の関係を示しています。

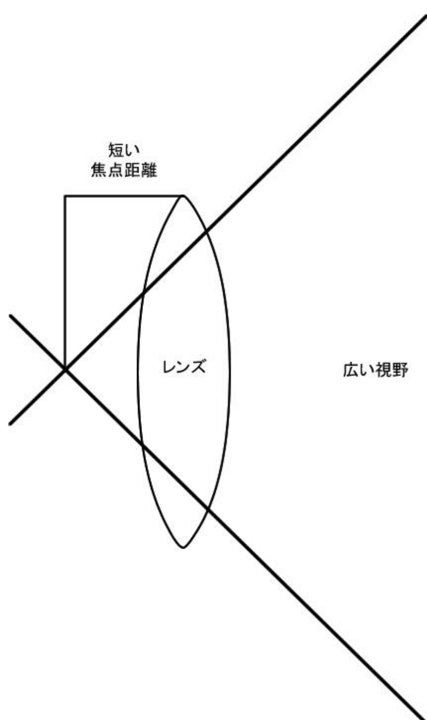

図 7-14：広角レンズでは焦点距離が短い

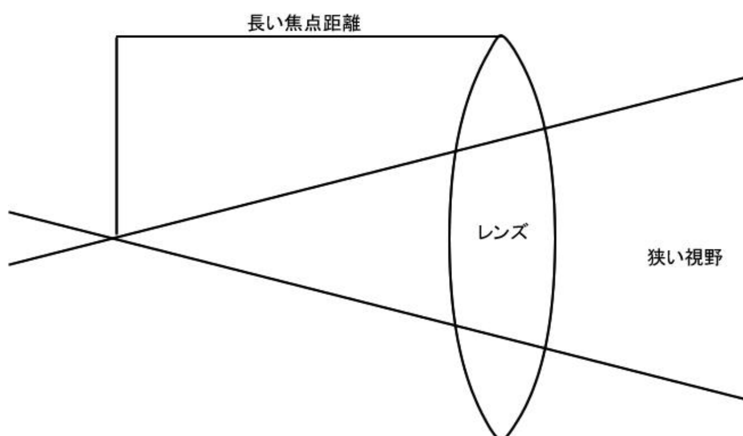

図 7-15：狭い角度の（望遠）レンズでは、焦点距離が長い

Flash 10 3D では、この歪曲を焦点距離か視野を設定することで制御できます。実際にはどちらか一方を設定するともう一方も変更されるので、扱いやすい方を使うようにするとよいでしょう。この設定は、表示オブジェクトの transform プロパティの perspectiveProjection プロパティの２つのプロパティ、focalLength と fieldOfView を通して行います。一般的には、異なるコンテナやオブジェクトに異なる透視を行わせる必要がない限り（おそらくこれは別のウィンドウで 3D を見せたいような場合でしょう）、ムービーのルートに設定するのが最善です。

視野の単位は度なので、0 より大きく 180 より小さくする必要があり、そうしないとエラーが発生します。視野を 0 に設定するということは、何も見られないことになります。視野を 180 に設定すると焦点距離が 0 になり、計算やイメージのレンダリングに問題が発生するだろうということは容易に想像できます。ともかくこういった極端な値に近い値は使わない方がよいでしょう。ゼロに近い視野は焦点距離を無限大に近づけます。これはすべての 3D 透視スケールの無効に有効です。180 に近い視野によって焦点距離はごくわずかになり、レンダリングされるイメージは極度に歪曲します。実際の物理的な世界では、視野を 180 より大きくすることは可能です。わたしの知る限りでは、最も広角のレンズの視野は 220 度です。

ここで Flash に戻りましょう。これらの値を操作しその結果を観察することはよいことです。これは前の Carousel サンプルですぐに試すことができます。コンストラクタに次の太字の行を追加してください。

```
public function Carousel()
{
        root.transform.perspectiveProjection.fieldOfView = 110;
        _holder = new Sprite();
        _holder.x = stage.stageWidth / 2;
        _holder.y = stage.stageHeight / 2;
        _holder.z = 0;
        addChild(_holder);
        …
```

この追加によって透視が非常に顕著になります。“カメラ”に近い正方形が、“カメラ”から遠い正方形よりも非常に大きく見えるので、これはおそらくフォトギャラリーのようなアプリケーションに適した設定でしょう。図 7-16 はその結果です。

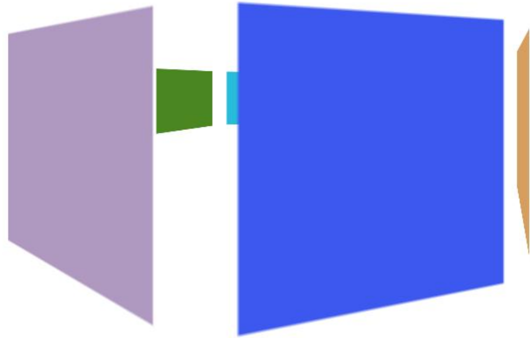

図 7-16：広い視野によって透視の歪曲が増す

次は視野をかなり狭く、25 に下げてみましょう。今度は透視のスケールがほとんど分からないくらいに小さくなります。図 7-17 はその結果です。

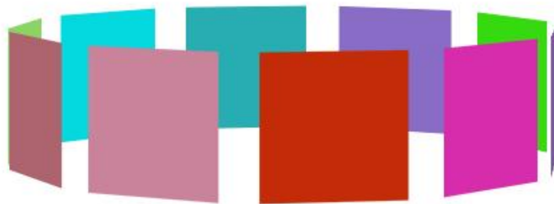

図 2-17：狭い視野では歪曲が小さくなる

また焦点距離でも高低さまざまな値を試すことができます。焦点距離を試すには次のようにします。

```
root.transform.perspectiveProjection.focalLength = 300;
```

ここでも短い焦点距離によって歪曲が多く生じ、高い値からは小さな歪曲が発生します。

スクリーンと 3D 座標

場合によっては、3D 空間の点に対応したスクリーン座標が必要なときがあります。また逆に、スクリーン上の座標が 3D 空間のどこに移動するかを知りたい場合もあります。ラッキーなことに、表示オブジェクトにはまさにこの目的のための local3DToGlobal()と globalToLocal3D()というビルトインメソッドがあります。local3DToGlobal()は flash.geom.Vector3D オブジェクトを 2D の flash.geom.Point オブジェクトに変換し、globalToLocal3D()はその逆の変換を行います。

まずは local3DToGlobal()メソッドの動作を見ていきましょう。次のサンプルでは、スプライトを作成しそれを 3D 内で動かします。そのスプライト内の x=200、y=0、z=0 で円を描画します（マッチ棒のようなスプライ

ト、_sprite）。さらにスプライトをもう1つ作成し（赤いわっかの_tracker）、_sprite のローカル 3D 点（200, 0, 0）をグローバルのスクリーン座標に変換することで移動させ、_sprite を追跡させます。

```
package
{
	import flash.display.Sprite;
	import flash.events.Event;
	import flash.geom.Point;
	import flash.geom.Vector3D;

	[SWF(width=800, height=800)]
	public class LocalGlobal extends Sprite
	{
		private var _sprite:Sprite;
		private var _tracker:Sprite;
		private var _angle:Number = 0;


		public function LocalGlobal()
		{
			_sprite = new Sprite();
			_sprite.graphics.lineStyle(10);
			_sprite.graphics.lineTo(200, 0);
			_sprite.graphics.drawCircle(200, 0, 10);
			_sprite.x = 400;
			_sprite.y = 400;
			addChild(_sprite);

			_tracker = new Sprite();
			_tracker.graphics.lineStyle(2, 0xff0000);
			_tracker.graphics.drawCircle(0, 0, 20);
			addChild(_tracker);
			addEventListener(Event.ENTER_FRAME, onEnterFrame);
		}
```

```
            private function onEnterFrame(event:Event):void
            {
                _sprite.rotationX += 1;
                _sprite.rotationY+= 1.2;
                _sprite.rotationZ += .5;
                _sprite.x = 400 + Math.cos(_angle) * 100;
                _sprite.y = 400 + Math.sin(_angle) * 100;
                _sprite.z = 200 + Math.cos(_angle * .8) * 400;
                _angle += .05;
                var p:Point =
                    _sprite.local3DToGlobal(new Vector3D(200, 0, 0));
                _tracker.x = p.x;
                _tracker.y = p.y;
            }
        }
}
```

コンストラクタでは 3D スプライト（_sprite）と追跡スプライト（_tracker）を作成し、その内部にグラフィックを置いています。enterFrame ハンドラはほとんど、3D 空間で_sprite を移動させるコードで構成されており、_sprite がランダムに移動するように見えます。_sprite は、わたしが無作為に入れた数値を使って、３軸全部で移動し回転します。重要な箇所は local3DToGlobal()の行で、ここでは_sprite が(200, 0, 0)を 2D の Point オブジェクトに変換し、その位置を_tracker スプライトの位置に代入しています。これを実行すると、_sprite は３次元のさまざまな場所を移動するにもかかわらず、_tracker は何の問題もなく_sprite の円に追随します。図 7-18 はその結果です。

図 7-18：3D の点を 2D で追跡する3つの場面

おそらくこの方法には、3D オブジェクトがスクリーンの外に出たタイミングを知るなど、さまざまな使い道があります。3D オブジェクトの x や y 位置は、z 軸ではるか後方にある場合には、スクリーン座標よりもかなり大きく、それでもオブジェクトは見えているので、オブジェクトがスクリーン上にあるかどうかを知りたいときに役立ちます。

方向を逆にすると、スクリーン位置からローカルの 3D 座標に変換することができます。次の GlobalLocal クラスは前のファイルを少し変えたもので、主な変更箇所は太字で示しています。

```
package
{
	import flash.display.Sprite;
	import flash.events.Event;
	import flash.geom.Point;
	import flash.geom.Vector3D;

	[SWF(width=800, height=800)]
	public class GlobalLocal extends Sprite
	{
		private var _sprite:Sprite;
		private var _tracker:Sprite;
		private var _angle:Number = 0;


		public function GlobalLocal()
		{
			_sprite = new Sprite();
			_sprite.graphics.lineStyle(5);
			_sprite.graphics.drawRect(-200, -200, 400, 400);
			_sprite.x = 400;
			_sprite.y = 400;
			addChild(_sprite);

			_tracker = new Sprite();
			_tracker.graphics.lineStyle(2, 0xff0000);
			_tracker.graphics.drawCircle(0, 0, 20);
			_sprite.addChild(_tracker);
			addEventListener(Event.ENTER_FRAME, onEnterFrame);
		}

		private function onEnterFrame(event:Event):void
```

```
            {
                _sprite.rotationX += 1;
                _sprite.rotationY+= 1.2;
                _sprite.rotationZ += .5;
                _sprite.x = 400 + Math.cos(_angle) * 100;
                _sprite.y = 400 + Math.sin(_angle) * 100;
                _sprite.z = 200 + Math.cos(_angle * .8) * 400;
                _angle += .05;
                var p:Vector3D =
                _sprite.globalToLocal3D(new Point(mouseX, mouseY));
                _tracker.x = p.x;
                _tracker.y = p.y;
            }
        }
}
```

ここでは回転させるスプライト（_sprite）に大きな正方形を描画し、_tracker もその内部に置いています。enterFrame ハンドラでは、_sprite で globalToLocal3D()を呼び出し、現在のマウス位置に関する 3D 座標を取得しています。これは Vector3D オブジェクトとして返されます。この Vector3D の z プロパティはこの場合つねに 0 になるので、その x と y プロパティを使うだけで、回転するスプライト内の_tracker の位置が設定できます。お分かりのように、_tracker は 3D 内を移動するものの、2D を移動するマウスに追随します。図 7-19 はその結果を示しています。

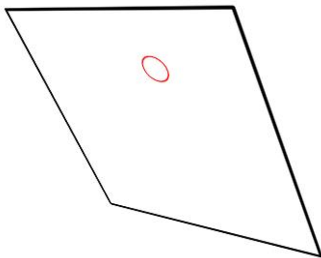

図 7-19：3D 内で 2D 座標に追随する

これはこれでクールですが、わたしはこれと同じことを、回転するスプライトのローカルのマウス座標を使って、もっと簡単に行う方法を見つけました。3D に変換されたオブジェクトの mouseX と mouseY プロパティにアクセスすると、そのグローバルからローカル 3D への変換が自動的に行われるのです。したがって onEnterFrame ハンドラはもっと簡単になります。

```
private function onEnterFrame(event:Event):void
{
        _sprite.rotationX += 1;
        _sprite.rotationY+= 1.2;
        _sprite.rotationZ += .5;
        _sprite.x = 400 + Math.cos(_angle) * 100;
        _sprite.y = 400 + Math.sin(_angle) * 100;
        _sprite.z = 200 + Math.cos(_angle * .8) * 400;
        _angle += .05;
        _tracker.x = _sprite.mouseX;
        _tracker.y = _sprite.mouseY;
}
```

これは前とまったく同じことを行っていますが、マウス座標を変換する場合のみ動作することを理解しておいてください。ステージ上のオブジェクトの座標をローカルの 3D 座標に変換したい場合には、変換関数を使用する必要があります。

何かを指す

Flash 10 の 3D に慣れてきたら、関連するさまざまなクラスをヘルプファイルで調べてみたくなるでしょう。調べ始めるのに適した場所は flash.geom パッケージで、その中には Matrix3D や Orientation3D、PerspectiveProjection、Utils3D、Vector3D といったクラスが含まれています。これらのクラスには 3D のあらゆる計算に役立つメソッドがぎっしり詰まっています。Matrix3D クラスを調べているときに見つけたものの中に pointAt()メソッドがあります。

pointAt()メソッドはターゲットとしてその方向を指し示す Vector3D オブジェクトを取ります。表示オブジェクトの transform プロパティの matrix3D オブジェクトから呼び出すと、指定された位置の方向を指し示すように、その表示オブジェクトを回転させます。次のサンプルではこれを試しています。

```
package
{
        import flash.display.Sprite;
        import flash.events.Event;
        import flash.events.MouseEvent;
        import flash.geom.Vector3D;
```

```
[SWF(width=800, height=800, backgroundColor = 0xffffff)]
public class FollowMouse3D extends Sprite
{
	private var _sprite:Sprite;
	private var _angleX:Number = 0;
	private var _angleY:Number = 0;
	private var _angleZ:Number = 0;

	public function FollowMouse3D()
	{
		_sprite = new Sprite();
		_sprite.x = 400;
		_sprite.y = 400;
		_sprite.z = 200;
		_sprite.graphics.beginFill(0xff0000);
		_sprite.graphics.moveTo(0, 50);
		_sprite.graphics.lineTo(-25, 25);
		_sprite.graphics.lineTo(-10, 25);
		_sprite.graphics.lineTo(-10, -50);
		_sprite.graphics.lineTo(10, -50);
		_sprite.graphics.lineTo(10, 25);
		_sprite.graphics.lineTo(25, 25);
		_sprite.graphics.lineTo(0, 50);
		_sprite.graphics.endFill();
		addChild(_sprite);

		addEventListener(Event.ENTER_FRAME, onEnterFrame);
	}

	private function onEnterFrame(event:Event):void
	{
		_sprite.x = 400 + Math.sin(_angleX += .11) * 200;
		_sprite.y = 400 + Math.sin(_angleY += .07) * 200;
		_sprite.z = Math.sin(_angleZ += .09) * 200;
		_sprite.transform.matrix3D.pointAt(new Vector3D(
```

```
                                                                        mouseX,
                                                                        mouseY,
                                                                        0));
                        }
                }
}
```

コンストラクタでは、ただ一連の lineTo()を使ってスプライト(_sprite)の中に矢印を描画して追加しているだけです。

onEnterFrame()メソッドはほとんど、LocalGlobal サンプルと異なり、そのスプライトを 3D のいたるところで移動させるコードで構成されています。最後の行では、スプライトの transform プロパティの matrix3D オブジェクトで pointAt()メソッドを呼び出しています。ここでは、マウスの x と y 座標に加え、z 軸での 0 から成る新しい Vector3D オブジェクトを渡しています。すると突然、矢印を持ったこのスプライトは各フレームでマウスの方を指すようになります。スプライトはさまざまな場所を移動し、マウスもさまざまな場所を移動できますが、スプライトは決してマウスを見失うことはありません。図 7-20 はその結果です。

図 7-20:マウスを 3D で指し示す

このサンプルはこのままで何かの役に立つものではありませんが、このテクニックは 3D ゲームのハンドル操作や狙撃などに役立つと思います。

まとめ

本章では多くの事柄を取り上げましたが、それでもまだ Flash 10 の 3D のごく一部をなぞったにすぎません。次章ではさらに、新しい描画 API 機能の 3D トピックを探っていきますが、本章が 3D において実現できる事柄のきっかけになればうれしく思います。flash.display.DisplayObject と flash.geom パッケージに関するドキュメンテーションは特に進んで調べるようにしてください。するとさらに 3D の楽しみが増えていきます。