Foundation Silverlight 2 Animation

Jeff Paries

Foundation Silverlight 2 Animation

ISBN-13 (pbk): 978-1-4302-1569-1

ISBN-13 (electronic): 978-1-4302-1570-7

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

Original Firefly photograph courtesy of Terry Priest, www.frfly.com.

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/info/bulksales.

The source code for this book is freely available to readers at www.friendsofed.com in the Downloads section.

Credits

Lead Editor
Ben Renow-Clarke

Technical Reviewer
Rob Houweling

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

Project Manager
Kylie Johnston

Copy Editor
Damon Larson

Associate Production Director
Kari Brooks-Copony

Production Editor
Laura Esterman

Compositor
Lynn L'Heureux

Proofreader
Dan Shaw

Indexer
Ron Strauss

Cover Image Designer
Corné van Dooren

Interior and Cover Designer
Kurt Krames

Manufacturing Director
Tom Debolski

Chapter 7

SIMULATING 3D IN 2D

In this chapter, we're going to apply some of the concepts from Chapter 6 to emulate 3D object rotations. While Silverlight doesn't yet support true 3D, you can add a little pop to your applications by implementing the concepts we'll talk about here.

3D

As you are aware, the coordinate system in Silverlight has only x and y axes, where x is the horizontal axis and y is the vertical. To imagine a 3D coordinate system like the one shown in Figure 7-1, a z axis line is drawn straight into your computer screen.

We're not going to be coding up a true 3D coordinate system—instead, we'll fake the visual cues that make people see objects as being farther away using some of the trigonometry you learned in Chapter 6.

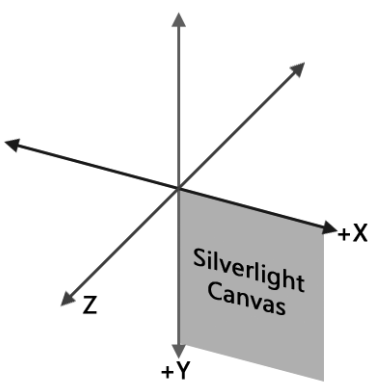

Figure 7-1. The z axis for a 3D coordinate system in Silverlight would run perpendicular to your monitor.

Z axis rotation

We'll start out with an easy one: z axis rotation. Any object you rotate around the z axis on the screen would be moving in a circular or elliptical pattern, as shown in Figure 7-2.

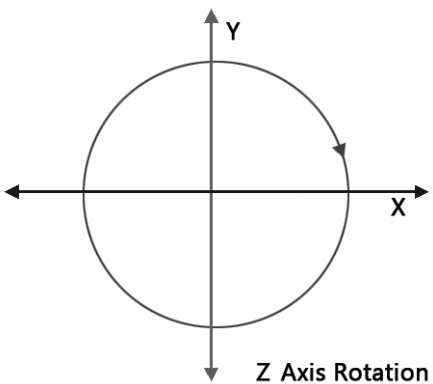

Figure 7-2. Z axis rotation results in a circular or elliptical rotation.

As such, z axis 3D movement isn't really emulating any 3D motion. It will still become part of your toolbox, however, so it's worth taking a look at. You saw how to create circular movements in Chapter 6, but let's do a quick review project that puts our terms in a context that works for 3D. In this example, we will build a project that moves a ball around the z axis.

1. Open the **ZAxis3D** project. This project contains a gradient-filled `Ball` object, and the main page. The `Page.xaml` file also contains a storyboard timer named `MoveBall`.
2. If you were building an application that supported multiple objects traveling in circular or elliptical paths, you might choose to place some of the code in the object's code-behind files. However, for this project, we'll just take a look at the one ball, so all of our code will go into the `Page.xaml.cs` file. Open `Page.xaml.cs` for editing.
3. Start by declaring an instance of the `Ball` object:

```
Ball MyBall = new Ball();
```

4. Next, declare the variables we'll be using for the movement. `Origin` will be the origin point for the motion, and `Radius` will determine the radius of the circle the ball will travel. `Angle` and `Speed` will be used to determine our sine and cosine calculations, and `Position` will determine where on the canvas the ball will be placed.

```
Point Origin;
double Radius = 150;
double Angle = 0;
double Speed = .1;
Point Position;
```

5. Inside the `Page()` constructor, add the following code to initialize the `Origin` variable. We're adjusting the location of the point of origin for the size of the ball by subtracting half of the ball's height or width from the center point on the canvas. This adjustment accounts for the fact that objects are identified by the point at the top left of the object rather than the center.

```
Origin.X = LayoutRoot.Width / 2 - MyBall.Width / 2;
Origin.Y = LayoutRoot.Height / 2 - MyBall.Height / 2;
```

6. Continue adding code inside the `Page()` constructor. The following three lines will place the ball object at the `Origin` position calculated in step 5, and add it to the LayoutRoot Canvas.

```
Canvas.SetLeft(MyBall, Origin.X);
Canvas.SetTop(MyBall, Origin.Y);
LayoutRoot.Children.Add(MyBall);
```

At this point, you can compile and run the application if you'd like. The main canvas will be drawn, and the ball will be positioned at the center of the canvas. When you're done looking at the application, close the browser window and return to the `Page.xaml.cs` file.

7. The next thing we need to do is make the ball move. The following two lines also go into the `Page()` constructor. They set up an event listener for the `Completed` event of the `MoveBall` storyboard, and start the storyboard.

```
MoveBall.Completed += new EventHandler(MoveBall_Completed);
MoveBall.Begin();
```

8. The code block shown here is the complete code for the `MoveBall_Completed()` event handler. The code calculates a new x and y position for the ball based on the cosine and sine of the `Angle`. The ball's position is then updated on the canvas before incrementing the `Angle` and restarting the `MoveBall` storyboard.

```
void MoveBall_Completed(object sender, EventArgs e)
{
    Position.X = Radius * Math.Cos(Angle);
    Position.Y = Radius * Math.Sin(Angle);
    Canvas.SetLeft(MyBall, Position.X + Origin.X);
    Canvas.SetTop(MyBall, Position.Y + Origin.Y);

    Angle += Speed;
    MoveBall.Begin();
}
```

That's all there is to it—compile and run the application, and the ball will move in a circular pattern around the center of the canvas. With a couple of small changes, the path the ball follows can be made elliptical.

9. Change the `Radius` data type to `Point`:

```
Point Radius;
```

10. Inside the `Page()` constructor, initialize `Radius.X` and `Radius.Y` with some values:

```
Radius.X = 300;
Radius.Y = 50;
```

11. Finally, in the `MoveBall_Completed()` event handler, change the `Position` calculations to use the `Radius` values:

```
Position.X = Radius.X * Math.Cos(Angle);
Position.Y = Radius.Y * Math.Sin(Angle);
```

Take some time and look at the **ZAxis3DCompleted** project, shown in Figure 7-3. It contains the code shown here in the example, but I also added a few sliders so you can manipulate some of the values in real time as the application runs.

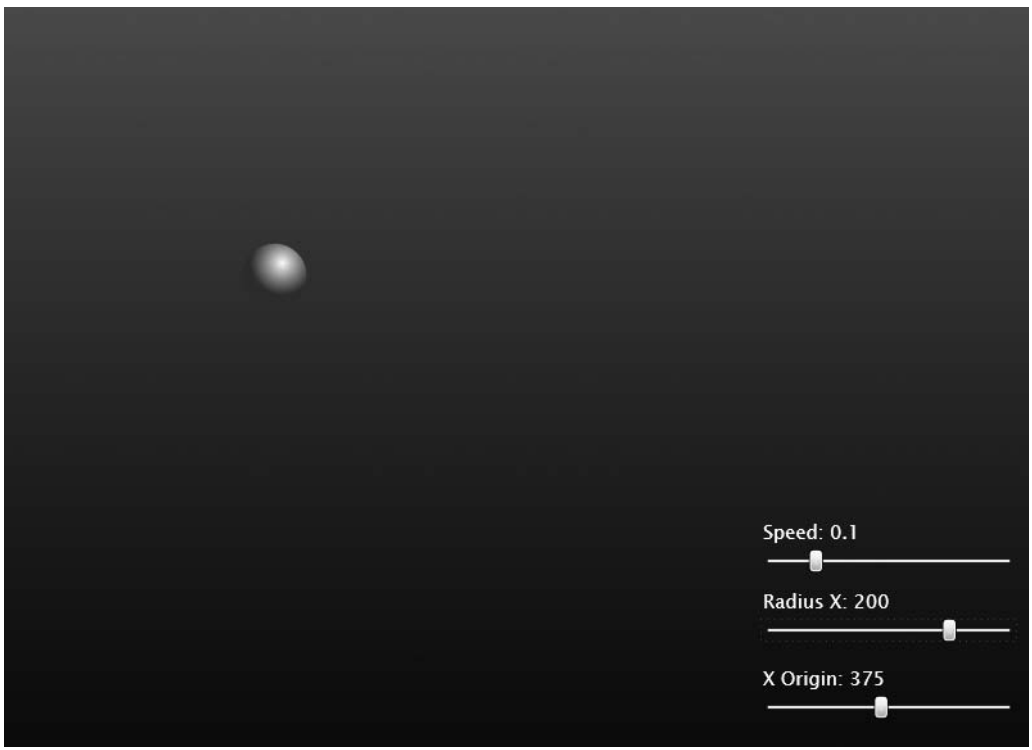

Figure 7-3. The ZAxis3DCompleted project

A model of the inner solar system

So you've built this project and put some thought into it, but maybe you're not entirely sure where something like this can be applied in your own applications. As an example, let's build a real-world, working model of the orbits of the inner planets in our solar system. We will write the program in a way that supports elliptical orbits—the inner planets travel in more circular orbits than the outer planets, but you may choose to augment the program with elliptical orbits at a later time.

1. Open the **InnerSolarSystem** project. The project contains the sun and four planet objects: Mercury, Venus, Earth, and Mars (the planets are not to scale). We will be coding up each planet's code-behind file in order, starting at the center of the solar system and moving outward. There is also a timer storyboard called `MovePlanets`.
2. Open the `Page.xaml.cs` file for editing. Since we're starting from the center and working our way out, we'll begin with the sun. Declare an instance of the `Sun` object just above the `Page()` constructor:

```
Sun MySun = new Sun();
```

3. Inside the Page() constructor, add the following code to position the sun and add it to the main canvas:

```
Canvas.SetLeft(MySun, 395.00);
Canvas.SetTop(MySun, 300.00);
LayoutRoot.Children.Add(MySun);
```

4. If you compile and run the application, you will see the sun object near the center of the black canvas. Next, we'll add Mercury, the planet closest to the sun. Still working in Page.xaml.cs, create an instance of the Mercury user control:

```
Mercury MyMercury = new Mercury();
```

5. Inside of the Page() constructor, add the Mercury object to the main canvas. Notice here that we're not positioning the planet. We'll be calculating its location mathematically, so it's not necessary to specify a starting location. It will automatically be added at 0,0.

```
LayoutRoot.Children.Add(MyMercury);
```

6. We'll add a little code to make the planet move. Open the Mercury.xaml.cs file for editing.

7. Before the Mercury() constructor, declare the following variables. We'll declare doubles for Angle and Speed, and Points for Center and Radius. The smaller a planet's orbit, the faster the planet travels. As such, Mercury will be our fastest-moving planet, and the speed of all of our other planets will be determined using Mercury's speed as a base.

```
private double Angle = 0;
private double Speed = .05;
private Point Center;
private Point Radius;
```

8. Inside the Mercury() constructor, initialize the Radius and Center variables as shown in the following code. The values being used were determined by me from a reference image. Notice that we are once again adjusting the location of the center point to accommodate the way Silverlight references objects by their top-left coordinate.

```
public Mercury()
{
    InitializeComponent();
    Radius.Y = 79.5;
    Radius.X = 78;
    Center.X = 410 - this.Width / 2;
    Center.Y = 328.50 - this.Height / 2;
}
```

9. The last bit of code we need for Mercury will be used to move the object. The following public method is used to update Mercury's position on the screen based on the calculated Angle value. Each time the method is called, the Angle is decremented to move the planet in a counter-clockwise direction.

```
public void MoveMercury()
{
    Canvas.SetLeft(this, Center.X + (Math.Cos(Angle) * Radius.X));
    Canvas.SetTop(this, Center.Y + (Math.Sin(Angle) * Radius.Y));
    Angle -= Speed;
}
```

10. In order to make Mercury orbit the sun, we'll need to add a little more code to the main code-behind, so return to the `Page.xaml.cs` file. In the `Page()` constructor, add the following code to attach a `Completed` event listener to the `MovePlanets` storyboard, and start the storyboard:

```
MovePlanets.Completed += new EventHandler(MovePlanets_Completed);
MovePlanets.Begin();
```

11. The event handler code for the `MovePlanets_Completed()` event is shown following. This code calls the `MoveMercury()` method of the `MyMercury` object, which will update the location of the planet on the main canvas. The storyboard timer is then restarted.

```
void MovePlanets_Completed(object sender, EventArgs e)
{
    MyMercury.MoveMercury();
    MovePlanets.Begin();
}
```

Compile and run the program. Mercury should be orbiting the sun, as shown in Figure 7-4.

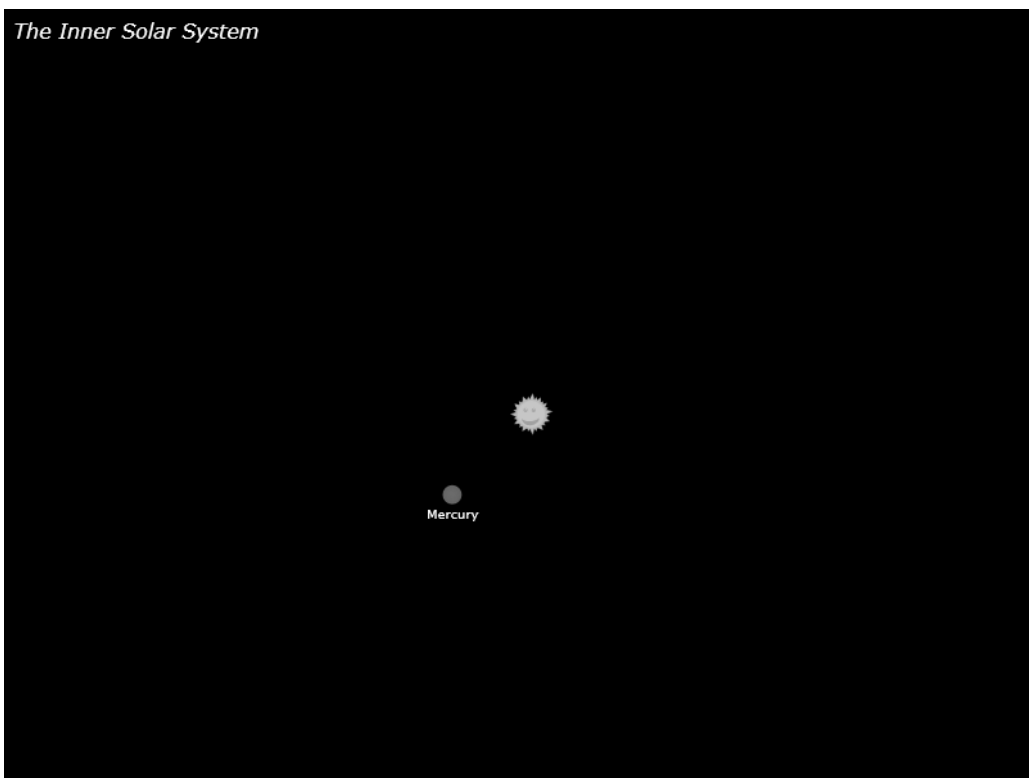

Figure 7-4. Mercury orbiting the sun

12. The next planet is Venus. In the Page.xaml.cs file, declare an instance of the Venus user control:

```
Venus MyVenus = new Venus();
```

13. Add MyVenus to the main canvas:

```
LayoutRoot.Children.Add(MyVenus);
```

14. Open the Venus.xaml.cs file for editing. Add the following variable declarations just above the Venus() constructor. Notice the Speed value here. The average speed of the planet Mercury is 48km/sec. For MyMecury, we used a Speed value of .05, and we need to make the other planets move relative to MyMercury's speed. The average speed of the planet Venus is 35km/sec. To determine the speed of MyVenus in relation to MyMercury, we take 35 / 48 × .05.

```
private double Angle = 0;
private double Speed = .036;
private Point Center;
private Point Radius;
```

15. Next, add the following bold code to the Venus() constructor to initialize the Radius and Center variables:

```
public Venus()
{
    InitializeComponent();
    Radius.Y = 140;
    Radius.X = 140;
    Center.X = 414 - this.Width / 2;
    Center.Y = 315 - this.Height / 2;
}
```

16. Finish up the Venus code-behind by adding the MoveVenus() method. The code inside the method is identical to that used for Mercury.

```
public void MoveVenus()
{
    Canvas.SetLeft(this, Center.X + (Math.Cos(Angle) * Radius.X));
    Canvas.SetTop(this, Center.Y + (Math.Sin(Angle) * Radius.Y));
    Angle -= Speed;
}
```

17. In the Page.xaml.cs file, locate the MovePlanets_Completed() event handler code. Add a call to the MoveVenus() method on the MyVenus object:

```
MyVenus.MoveVenus();
```

Compile and run the application. Both Mercury and Venus should be in view, as shown in Figure 7-5. Notice that Venus is moving more slowly than Mercury as both planets orbit the sun.

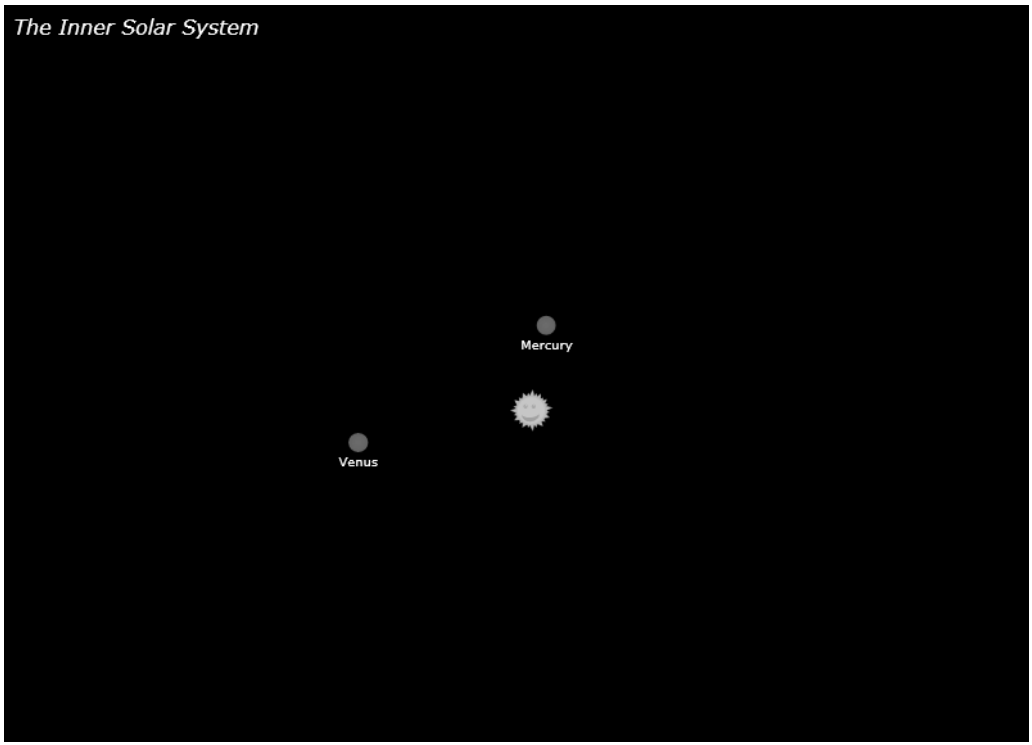

Figure 7-5. The InnerSolarSystem project with both Venus and Mercury in view

18. The last two planets follow the same pattern, so we'll advance the pace a bit. In the `Page.xaml.cs` file, create an instance of the `Earth` user control, and inside of the `Page()` constructor, add it to the main canvas:

```
Earth MyEarth = new Earth();
LayoutRoot.Children.Add(MyEarth);
```

19. In the `Earth.xaml.cs` file, declare the necessary variables. The planet Earth orbits the sun at about 30km/sec, so to determine `MyEarth`'s speed, we take 30 / 48 × .05.

```
private double Angle = 0;
private double Speed = .0312;
private Point Center;
private Point Radius;
```

20. Inside the `Earth()` constructor, initialize the `Radius` and `Center` variables by adding the code shown in bold in the following listing:

```
public Earth()
{
    InitializeComponent();
    Radius.Y = 189;
    Radius.X = 189;
```

```
    Center.X = 414 - this.Width / 2;
    Center.Y = 318 - this.Height / 2;
}
```

21. Create the `MoveEarth()` method:

```
public void MoveEarth()
{
    Canvas.SetLeft(this, Center.X + (Math.Cos(Angle) * Radius.X));
    Canvas.SetTop(this, Center.Y + (Math.Sin(Angle) * Radius.Y));
    Angle -= Speed;
}
```

22. In the `Page.xaml.cs` file, call the `MyEarth.MoveEarth()` method to add the `Earth` object to the storyboard's `Completed` event handler:

```
MyEarth.MoveEarth();
```

23. Running the application at this point will show our first three planets orbiting the sun, which means we have only one more to code up!

Still in the `Page.xaml.cs` file, create an instance of the `Mars` user control, and add it to the main canvas of the application:

```
Mars MyMars = new Mars();
LayoutRoot.Children.Add(MyMars);
```

24. In the `Mars.xaml.cs` file, declare the requisite variables. The planet Mars orbits the sun at about 24km/sec. In relation to `MyMercury`'s orbital speed, we get 24 / 48 × .05.

```
private double Angle = 0;
private double Speed = .0250;
private Point Center;
private Point Radius;
```

25. Add the following bold code to the `Mars()` constructor to initialize the `Radius` and `Center` values:

```
public Mars()
{
    InitializeComponent();
    Radius.Y = 281;
    Radius.X = 281;
    Center.X = 391 - this.Width / 2;
    Center.Y = 305 - this.Height / 2;
}
```

26. Create the `MoveMars()` method as shown:

```
public void MoveMars()
{
    Canvas.SetLeft(this, Center.X + (Math.Cos(Angle) * Radius.X));
    Canvas.SetTop(this, Center.Y + (Math.Sin(Angle) * Radius.Y));
    Angle -= Speed;
}
```

27. In the `Page.xaml.cs` file, locate the `MovePlanets_Completed()` event handler, and add a call to `MyMars.MoveMars()`:

```
MyMars.MoveMars();
```

Compile and run the program, and you'll see all four planets orbiting the sun. The **InnerSolarSystemCompleted** project, shown in Figure 7-6, contains the code covered in this example. I also added a check box that allows you to toggle the orbits of the planets, which are ellipses that were manually added.

Think about some of the ways this program could be augmented. Certainly, adding more planets is an option. What about adding moons? How would you go about doing that? Can you figure out a way to "seed" the starting angle for each planet so it starts at a random location along its orbit? Is it possible to calculate each planet's distance from the sun?

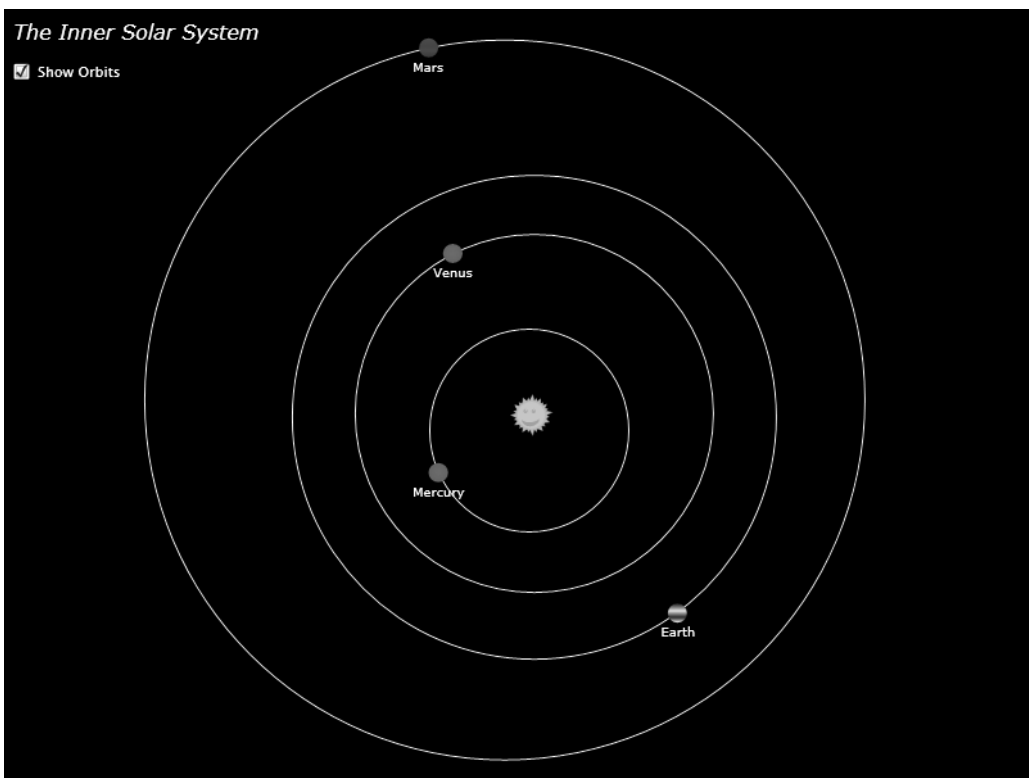

Figure 7-6. The completed InnerSolarSystem project with orbits visible

Y axis rotation

In this section, we're going to look at how we can go about emulating movement of an object around the y axis, as illustrated in Figure 7-7.

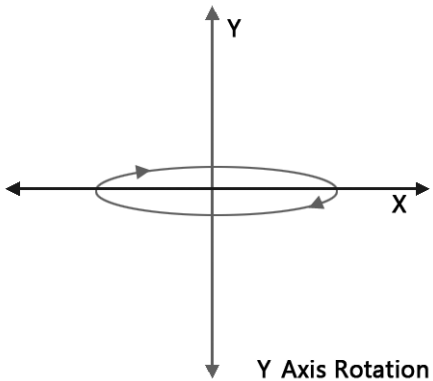

Figure 7-7. Y axis rotation causes an object to move up and down, front to back.

One of the biggest visual clues we have to tell how far we are from an object is scale. Objects closer to us are larger, and those farther away are smaller, as illustrated by Figure 7-8. Our brains are pretty good at comparing the relative sizes of objects we know, such as houses, trees, and vehicles, in order to estimate how large an object may be.

Generally speaking, the farther away an object is, the harder it is to see—our view becomes obscured by haze, and distant objects are not as well defined. Next, you're going to learn how to exploit scale and translucency in order to make objects appear to be moving either toward or away from the viewer.

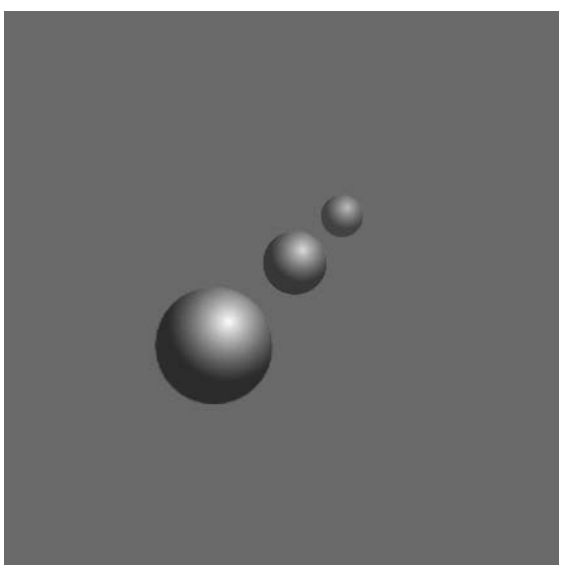

Figure 7-8. Objects appear to get more distant as they are scaled down and made more translucent.

Let's code up another example. This time, we'll write some code that emulates motion around the y axis.

1. Open the **YAxis3D** project. This project contains a simple layout Canvas, our trusty `Ball` user control, and a storyboard timer called `MoveBall`.
2. All of the code for this project will go into the `Page.xaml.cs` file, so open that file for editing.
3. Above the `Page()` constructor, declare an instance of the `Ball` object as well as the variables shown. `Origin` will be used as the origin for the rotation. As with the previous examples, `Radius` determines the distance the ball will travel. `Angle` and `Speed` handle the rotation for us. `Distance` and `Perspective` are not things you have seen yet. These two variables work together to create a modifier for scaling the object during rotation. `Position` is used to place the ball on the canvas.

```
Ball MyBall = new Ball();
Point Origin;
Point Radius;
double Angle = 0;
double Speed = .1;
double Distance;
double Perspective = 400;
Point Position;
```

4. Inside the `Page()` constructor, add the following code to initialize the `Origin` variable, and place the ball object at that location before adding it to the main canvas. As with the previous examples, the `Origin` variable is adjusted to align to the center point of the ball rather than the top left.

```
Origin.X = LayoutRoot.Width / 2 - MyBall.Width / 2;
Origin.Y = LayoutRoot.Height / 2 - MyBall.Height / 2;
Canvas.SetLeft(MyBall, Origin.X);
Canvas.SetTop(MyBall, Origin.Y);
LayoutRoot.Children.Add(MyBall);
```

5. Continuing with variable initialization, set the Y `Position` variable to match the `Origin.Y` value. This is where the ball is currently located.

```
Position.Y = Origin.Y;
```

6. Assign values to the `Radius` variable. This will control the height and width of the movement.

```
Radius.X = 200;
Radius.Y = 50;
```

7. Finish up the constructor code by adding a `Completed` event listener to the `MoveBall` storyboard and starting the storyboard:

```
MoveBall.Completed += new EventHandler(MoveBall_Completed);
MoveBall.Begin();
```

8. Add the code to the MoveBall_Completed() event handler. The first two lines calculate the position of the ball. The third line calculates the scale modifier based on the y position of the ball and the Perspective variable. The next line adjusts the scale of the ball based on the Distance calculation. The position of the ball on the canvas is then updated before incrementing the Angle value and restarting the storyboard.

```
void MoveBall_Completed(object sender, EventArgs e)
{
    Position.X = Radius.X * Math.Cos(Angle);
    Position.Y = Radius.Y * Math.Sin(Angle);
    Distance = 1 / (1 - (Position.Y / Perspective));
    MyBall.BallScale.ScaleX = MyBall.BallScale.ScaleY = Distance;

    Canvas.SetLeft(MyBall, Position.X + Origin.X);
    Canvas.SetTop(MyBall, Position.Y + Origin.Y);
    Angle += Speed;
    MoveBall.Begin();
}
```

9. Compile and run the application. The ball will follow a circular path that appears to be 3D. There's one small addition we can make to help with the illusion. Inside the MoveBall_Completed() function, adjust the Opacity property of the ball along with the scale. Now the ball will also fade out as it scales down on the back side of the rotational movement.

```
MyBall.Opacity = MyBall.BallScale.ScaleX =
                 MyBall.BallScale.ScaleY = Distance;
```

10. Try some different values and see what happens when you run the program. One of the things you'll notice is that the Perspective variable, when used in conjunction with a decreasing x Radius, will act similarly to the depth of field on a camera, "flattening" the motion. Here are a couple of values with which you can experiment:

    a. Radius.X: 200; Radius.Y: 0

    b. Elliptical path: Radius.X: 200; Radius.Y: 200; Perspective: 1000

All of the code for this example is in the **YAxis3DCompleted** project. I also added a few sliders and a check box to the project, as shown in Figure 7-9. The sliders will allow you to change the Speed, Radius, and Perspective values as the application is running. The check box allows you to toggle the transparency so you can see what effect that has on the object.

Now, I bet some of you reading this are thinking that the motion looks a bit like what you might use on a carousel-style interface, and you'd be right. Getting from here to there requires a few extra steps, but it's not as hard as it might seem. In the next exercise, we're going to take a detailed look at how to create a horizontal carousel. We'll create a carousel that uses proxy containers for the carousel items, to which you can add whatever functionality you'd like (images, movies, etc.).

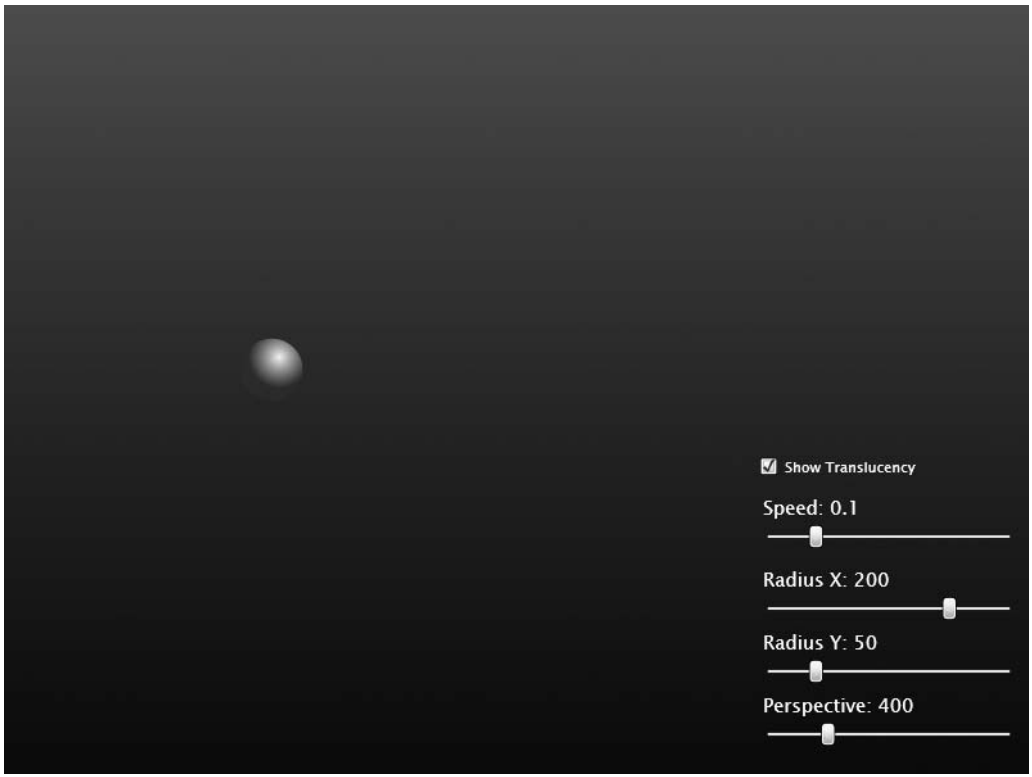

Figure 7-9. The YAxis3DCompleted project includes some sliders to change values in real time.

A horizontal carousel

This time, you're going to do almost all of the work. I've set up a base project so you can follow along, but to really understand how everything fits together, it will be of more value to you to work through the project from the very beginning.

1. Open the **HorizontalCarousel** project. The project currently contains only the main page XAML, which consists of a gradient background and a storyboard timer called `Spin`.
2. Right-click the HorizontalCarousel C# project name in Visual Studio's Solution Explorer and select Add ➤ New Item, as shown in Figure 7-10.
3. When the Add New Item dialog opens, select Silverlight User Control. Type in the name CarouselItem and click the Add button. Visual Studio will create the new object for you and add it to Solution Explorer. This object will become the basis for each carousel item added to the application.
4. Edit the `CarouselItem.xaml` file—you can do this either directly in Visual Studio or by right-clicking CarouselItem.xaml in Solution Explorer and selecting Open in Expression Blend from the pop-up menu.

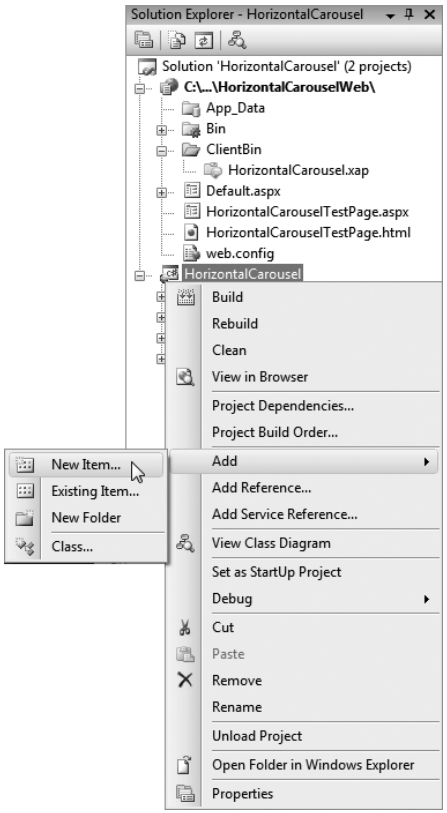

Figure 7-10. Adding a new item through Visual Studio's Solution Explorer

5. Change the size of the user control to 150×100—change both the user control and LayoutRoot Canvas dimensions. Add a white rectangle with a black stroke that is 150×100 with rounded corners—both the `RadiusX` and `RadiusY` properties should be 5. Name this rectangle RectBackground. In addition, add two TextBlocks. These will be used to identify which item is which. Center the first TextBlock near the top of the rectangle and add the text Carousel Item. Name the TextBlock MsgItem. Center the second TextBlock beneath the first, add the text 00, and name it MsgNumber. Also, add a transform group to the LayoutRoot Canvas. Name the `ScaleTransform` ItemScale. The XAML that goes inside the user control for this object is shown in the following listing. Figure 7-11 shows the object.

```
<Canvas x:Name="LayoutRoot" RenderTransformOrigin="0.5,0.5"
        Width="150" Height="100">
    <Canvas.RenderTransform>
        <TransformGroup>
            <ScaleTransform x:Name="ItemScale" ScaleX="1" ScaleY="1"/>
        </TransformGroup>
    </Canvas.RenderTransform>
```

```
    <Rectangle Height="100" Width="150" Fill="#FFFFFFFF"
               RadiusX="5" RadiusY="5"
               x:Name="RectBackground" Stroke="#FF000000"/>
    <TextBlock Text="Carousel Item" TextWrapping="Wrap"
               Canvas.Left="25.769" Canvas.Top="10"
               x:Name="MsgItem"/>
    <TextBlock Text="00" TextWrapping="Wrap"
               Canvas.Top="50" Canvas.Left="65.726"
               x:Name="MsgNumber"/>
</Canvas>
```

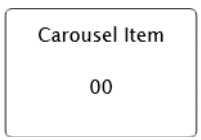

Figure 7-11. This figure shows what the carousel item should look like.

6. Open the `CarouselItem.xaml.cs` file for editing. We will be using code to distribute the carousel items along an elliptical path when creating the carousel. In order to position each item appropriately, we need to store a unique angle for each carousel item. Above the `CarouselItem()` constructor, add the following code to create a publicly accessible `Angle` variable:

```
public double Angle = 0;
```

7. Open the `Page.xaml` file for editing. This is the main application page. One thing I have found useful when working with carousels is to create a canvas to contain the carousel rather than generating it directly on the main canvas. This makes the carousel much more manageable later—move the canvas and the carousel goes with it.

   Add a canvas container to `Page.xaml`. Enter the following code two lines up from the bottom of the XAML, just before the closing `</Canvas>` tag. Notice that this canvas is identical in size to one of the carousel items. This will make adjustments for centering later very easy.

```
<Canvas Height="100" Width="150" x:Name="CarouselCanvas"/>
```

8. Now we'll start coding up the carousel. Open the `Page.xaml.cs` file for editing, and add the following variable declarations just before the `Page()` constructor. Each element added to the carousel will be stored in a `List` called `Items`. The `NumItems` variable allows easy modification to the number of elements in the carousel. By now, you should be familiar with the functionality provided by `Radius`, `Position`, `Speed`, `Distance`, and `Perspective`.

```
List<CarouselItem> Items;
int NumItems = 8;
Point Radius;
Point Position;
double Speed = .0125;
double Distance;
double Perspective = 300;
```

9. Inside the `Page()` constructor, add code to define a length for the `Items List` in the carousel. The length of the list is determined by the `NumItems` variable initialized in step 8.

```
Items = new List<CarouselItem>(NumItems);
```

10. Add code to initialize the Radius variable. Using a negative value for the radius will make the carousel appear to be tipped forward.

```
Radius.X = 300;
Radius.Y = -50;
```

11. Position the canvas that will contain the carousel:

```
Canvas.SetLeft(CarouselCanvas, LayoutRoot.Width / 2
                - CarouselCanvas.Width / 2);
Canvas.SetTop(CarouselCanvas, LayoutRoot.Height / 2
               - CarouselCanvas.Height / 2);
```

12. Next, we will build a function called BuildCarousel() that will be used to populate the carousel. The process of defining each element is done inside of a for loop. The basic structure for the function is shown in the following code listing:

```
private void BuildCarousel()
{
    for (int i = 0; i < NumItems; i++)
    {
    }
}
```

13. Begin filling in the for loop with the following code. Here, we create an instance of CarouselItem() called item. Once the instance has been defined, we populate one of the TextBlocks with the element number. This is strictly for reference for this carousel—if you had a set of images or videos in your carousel, this is where you would assign the Source property.

```
CarouselItem item = new CarouselItem();
item.MsgNumber.Text = String.Format("{0:00}", i);
```

14. Next, each item has a value assigned to its public Angle variable. The Angle is then used to calculate the Position variable:

```
item.Angle = i * ((Math.PI * 2) / NumItems);
Position.X = Math.Cos(item.Angle) * Radius.X;
Position.Y = Math.Sin(item.Angle) * Radius.Y;
```

15. Once Position has been calculated, place the element at the value stored in the variable:

```
Canvas.SetLeft(item, Position.X);
Canvas.SetTop(item, Position.Y);
```

16. The Distance variable is calculated to determine a modifier value that will be used for scaling the elements and adjusting their opacity. With Distance calculated, the Opacity and ItemScale are adjusted.

```
Distance = 1 / (1 - (Position.Y / Perspective));
item.Opacity = item.ItemScale.ScaleX =
               item.ItemScale.ScaleY = Distance;
```

17. To finish out the function, add the item to the Items List and the CarouselCanvas:

```
Items.Add(item);
CarouselCanvas.Children.Add(item);
```

18. The function that was just created needs to be called from the `Page()` constructor in order to create the carousel when the application is loaded. Add the following code as the last line in the `Page()` constructor:

```
BuildCarousel();
```

Compile and run the application. When the browser opens, you should see an application similar to the one shown in Figure 7-12. The application calls the `BuildCarousel()` function, which adds eight elements to the `CarouselCanvas`. The elements are evenly distributed based on the `Angle` and `Radius` variables that were defined.

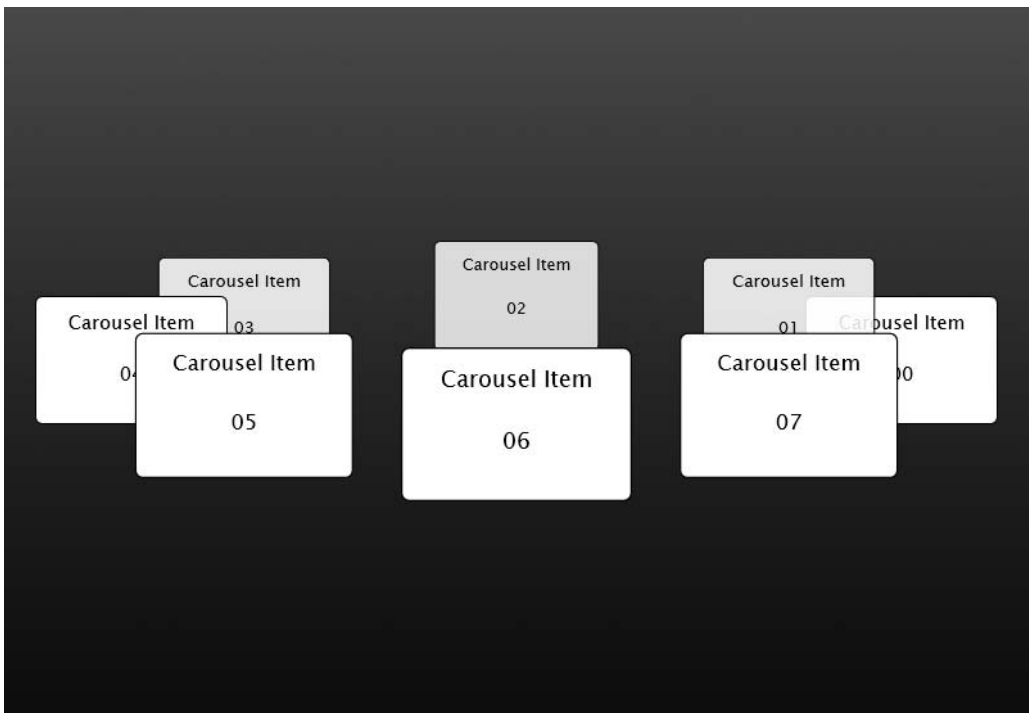

Figure 7-12. The carousel is populated and drawn on the screen.

19. Now we need to make it move. If you're thinking this is done by manipulating the angle of each object, you're right! To start, we'll need to add a bit of code at the bottom of the `Page()` constructor. The two lines shown following attach an event listener to the `Completed` event of the `Spin` storyboard, and start the storyboard. Notice that the event handler being called is named `MoveCarousel()`.

```
Spin.Completed += new EventHandler(MoveCarousel);
Spin.Begin();
```

20. The `MoveCarousel()` event handler takes advantage of the `foreach` loop, which will step through each element in a `List` like the one we're using. The basic structure for the event handler is shown in the following listing:

```
private void MoveCarousel(object sender, EventArgs e)
{
    foreach (CarouselItem item in Items)
    {
    }
}
```

21. Inside the `foreach` loop, add the following code. This code decrements the angle for each element, recalculates the position, and repositions the object on the `CarouselCanvas`.

```
item.Angle -= Speed;
Position.X = Math.Cos(item.Angle) * Radius.X;
Position.Y = Math.Sin(item.Angle) * Radius.Y;
Canvas.SetLeft(item, Position.X);
Canvas.SetTop(item, Position.Y);
```

22. As the objects rotate, the Z-index needs to be adjusted—the items in the front need to appear in front of the items in the back. To do this, the Z-index is tied to an item's y position. A test is done against the y `Radius` value to determine if it is greater than or equal to 0. If so, the distance is calculated and the Z-index of the item adjusted. If the y radius is less than 0, the distance calculation is performed opposite the first method. This ensures that the carousel will draw correctly regardless of the y `Radius` value. Be careful when you type in this code—the difference between the two distance calculations is very small, but makes a big difference in the end result.

```
if (Radius.Y >= 0)
{
    Distance = 1 * (1 - (Position.Y / Perspective));
    Canvas.SetZIndex(item, -(int)Position.Y);
}
else
{
    Distance = 1 / (1 - (Position.Y / Perspective));
    Canvas.SetZIndex(item, (int)Position.Y);
}
```

23. With the `Distance` variable having been recalculated based on the y position of the carousel item, the scale and transparency of the object can be updated:

```
item.Opacity = item.ItemScale.ScaleX
             = item.ItemScale.ScaleY = Distance;
```

24. That's all for the `foreach` loop, but we still need to restart the storyboard. Just after the closing curly brace (}) for the `foreach` loop, add the following code to restart the storyboard:

```
Spin.Begin();
```

Now when you compile and run the program, the carousel will load, draw, and begin spinning from right to left (clockwise when viewed from above). If you wanted the carousel to spin in the opposite direction, you would increment the `Angle` variable in step 21 rather than decrement it. We still need to augment our functionality a little bit—the carousel isn't going to be very useful unless it's interactive. Let's begin adding to the carousel by creating some mouse events for the carousel items.

25. Inside the `BuildCarousel()` function, add event listeners for `MouseEnter` and `MouseLeave`. Since we're creating our carousel items inside of a loop, we only need to add the following two lines to create event listeners for every item on the carousel:

```
item.MouseEnter += new MouseEventHandler(item_MouseEnter);
item.MouseLeave += new MouseEventHandler(item_MouseLeave);
```

26. Inside the `MouseEnter()` event handler, add code to stop the carousel from spinning:

```
void item_MouseEnter(object sender, MouseEventArgs e)
{
    Spin.Stop();
}
```

27. When the mouse pointer leaves an item, we want to restart the carousel spinning. The `item_MouseLeave()` event handler code is shown here:

```
void item_MouseLeave(object sender, MouseEventArgs e)
{
    Spin.Begin();
}
```

Now when you run the application, placing the mouse over any of the items on the carousel will cause the carousel to stop spinning. Moving the mouse off of the item will start the carousel back up. Chances are you will at some point need to further augment the functionality to access the properties of each carousel item.

For example, if the carousel contained images, when a user clicked on one of the images, you might want to display the image on another panel elsewhere in the application. The basis for retrieving information from the carousel items is the same regardless of the type of property you're trying to get, so we'll add some basic functionality to access which item has been clicked.

28. Open the `Page.xaml` file for editing. Just below the code that defines the `CarouselCanvas` element, add the following code to create a TextBlock named `MsgSelected`:

```
<TextBlock x:Name="MsgSelected" Text="You clicked: 00"
           TextWrapping="Wrap" Foreground="#FFFFFFFF"/>
```

29. Back in the `BuildCarousel()` function in the `Page.xaml.cs` file, add a `MouseLeftButtonUp` event listener.

```
item.MouseLeftButtonUp +=
     new MouseButtonEventHandler(item_MouseLeftButtonUp);
```

30. Since we know that the items being clicked are of the type `CarouselItem`, the code inside the event handler captures the `sender` as a `CarouselItem`. We then have access to all of the objects, properties, and so on that are part of the `CarouselItem`, so it's easy to assign the value of the `MsgNumber` TextBlock in the selected item to the TextBlock we just added to the main page.

```
void item_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
{
    CarouselItem WhichItem = sender as CarouselItem;
    MsgSelected.Text = "You clicked: " + WhichItem.MsgNumber.Text;
}
```

Compile and run the application. Each time you click an item in the carousel, the text in the TextBlock at the top left of the screen should update, as shown in Figure 7-13.

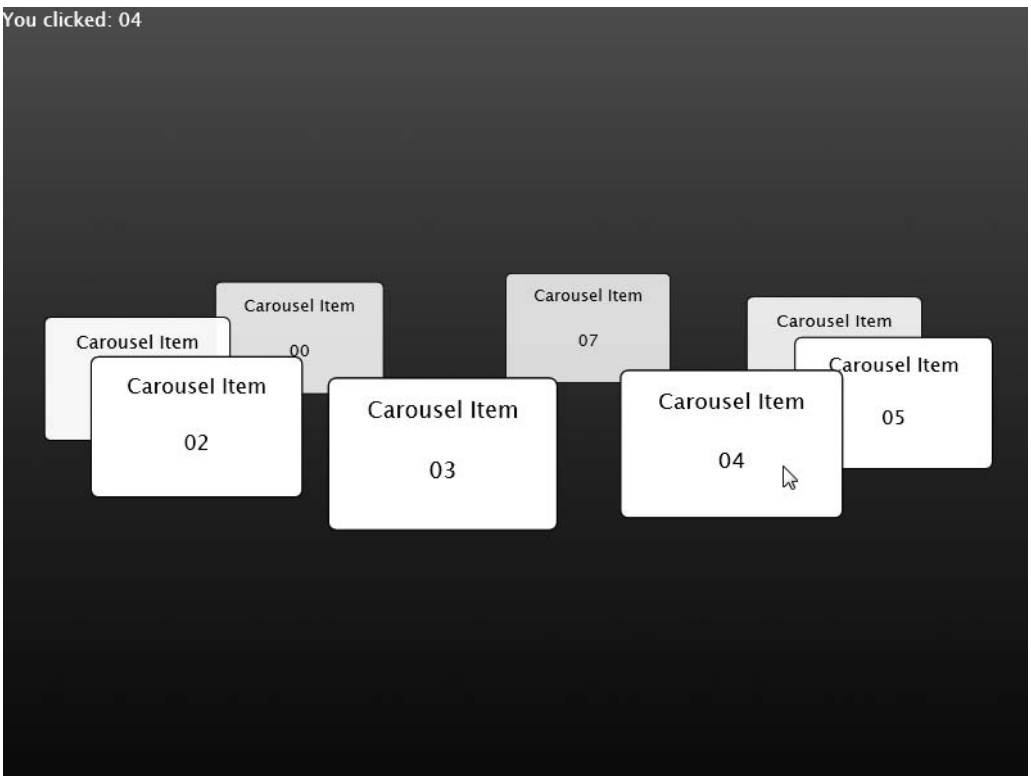

Figure 7-13. The carousel with some click functionality in place

The last bit of functionality we will be adding to the carousel will allow the mouse to control the speed and direction of spin. A lot of carousel controls have a tendency to be a little twitchy when it comes to control, so I'll show you a way to make the carousel behave the way we want, and you can change it as you see fit.

The functionality we will add is to place a rectangle in the `CarouselCanvas` object and hook up a `MouseMove` event to the rectangle. As the mouse moves, we will determine the offset from the center of the canvas, and use that calculation to spin the carousel in one direction or the other, allowing the speed to change as the mouse moves farther from center.

31. Open the `Page.xaml` file for editing. Add a rectangle called `MouseControl` inside of the `CarouselCanvas` container. The rectangle size will be manipulated via code, so a 100×100 rectangle is fine. Save the `Page.xaml` file.

```
<Canvas Height="100" Width="150" x:Name="CarouselCanvas">
    <Rectangle Height="100" Width="100" Fill="#FFFFFFFF"
        x:Name="MouseControl" Opacity="0"/>
</Canvas>
```

32. Open the Page.xaml.cs file for editing. Since the MouseControl rectangle has already been added to the XAML, we will make some adjustments to its size and location. The following code goes into the Page() constructor. Start by adjusting the Width property—here, the code makes the MouseControl element twice as wide as the Radius.X value and adds the Width of the CarouselCanvas to account for the top-left positioning Silverlight uses. Otherwise, our control would stop at the left side of the rightmost carousel item. The height of 100 is fine as-is. If you need a taller control for mouse input, you can modify the Height property to suit your needs.

```
MouseControl.Width = Math.Abs(Radius.X * 2) + CarouselCanvas.Width;
```

33. Next, we will position the MouseControl element. The element is moved left the equivalent of the Radius.X value. The top is moved down twice as far as the Radius.Y value. Note that while these values work well for the settings on this carousel, they may not work for every carousel. As the y radius is increased or decreased, you may need to adjust the top location for the MouseControl. An easy way to see where it's located is to set the Opacity of the MouseControl rectangle to .5. Once it has been positioned where you want it, you can then set the Opacity back to 0. The location of the rectangle for this example is shown in Figure 7-14.

```
Canvas.SetLeft(MouseControl, -(Math.Abs(Radius.X)));
Canvas.SetTop(MouseControl, (Math.Abs(Radius.Y) * 2));
```

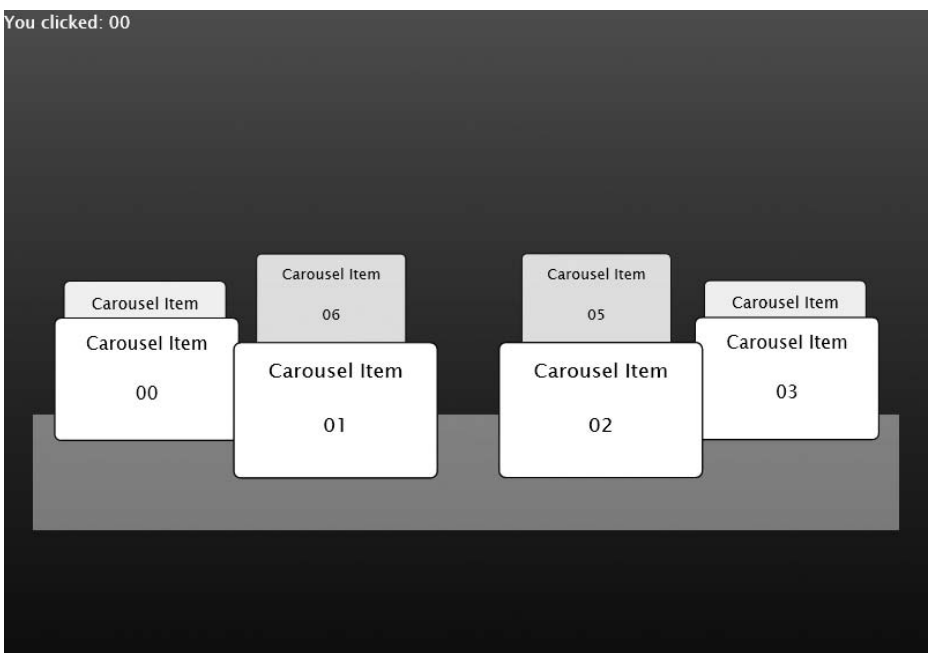

Figure 7-14. The location of the MouseControl rectangle for the carousel project

34. Finish up the work in the Page() constructor by adding an event listener for the MouseMove event on the MouseControl element:

```
MouseControl.MouseMove +=
        new MouseEventHandler(MouseControl_MouseMove);
```

35. The following code shows the `MouseControl_MouseMove()` event handler function. This code adds the functionality illustrated in Figure 7-15. The farther the mouse pointer moves from the center of the control rectangle, the faster the carousel will spin in the direction the mouse is moving. As the pointer approaches the center of the rectangle, the movement of the carousel slows or stops, eventually changing direction as the pointer crosses the center point.

This code gets the current position of the mouse pointer, and then calculates an offset from the center of the control rectangle. The speed is then calculated by dividing the offset by 10,000. The divisor you use depends upon the initial speed of your carousel, so you will likely need to modify that value based on your carousel design. The fewer decimal places represented in your Speed variable, the lower the divisor you will use. The function finishes up by checking the Speed value and limiting it to a maximum of two times the original speed.

```
void MouseControl_MouseMove(object sender, MouseEventArgs e)
{
    Point MousePoint = e.GetPosition(MouseControl);
    double OffsetCalc = MousePoint.X - (MouseControl.Width / 2);
    Speed = -OffsetCalc / 10000;
    if (Speed < 0 && Speed < -0.0250) Speed = -0.0250;
    else if (Speed > 0 && Speed > 0.0250) Speed = 0.0250;
}
```

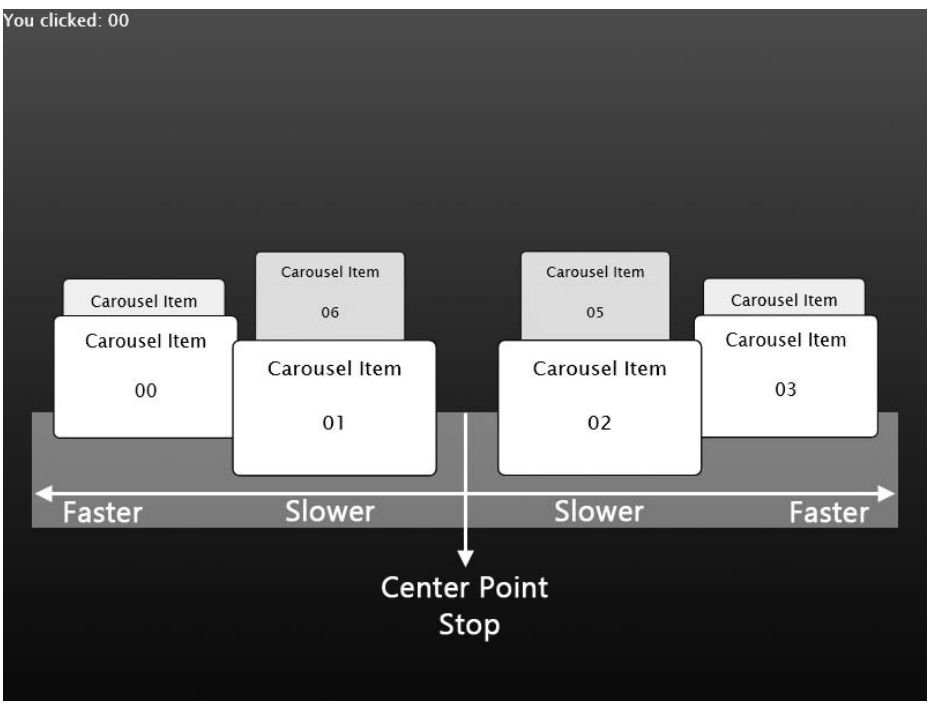

Figure 7-15. The mouse control for the carousel application

Now the application is pretty much complete. You can compile and run it to see the results. Moving the mouse just below the carousel will change the speed and/or direction of spin. Placing the mouse over an item on the carousel stops the movement, while moving the pointer off an object restarts the movement. Clicking an item in the carousel reports back which item was selected. While I am certain there are some further code optimizations that can be made, the end result of this application is worth mentioning—6.5K. Of course, the application size would grow as content were added to the carousel items, but we packed a lot of functionality into just about 130 lines of code.

All of the code described in this example is included in the **HorizontalCarouselCompleted** project. I also added an extra project for you to compile and run: **HorizontalCarouselHelper**. This project, shown in Figure 7-16, contains sliders that allow you to control the x and y origin, x and y radius, perspective, speed, and number of items for the demonstration carousel. You may find the application helpful in determining what settings you want to use for your own carousel applications before you dig in and start building. Take a good look at the code for the helper application—it illustrates how to move a carousel in two axes—the x to tilt, and the y for the rotation of the carousel.

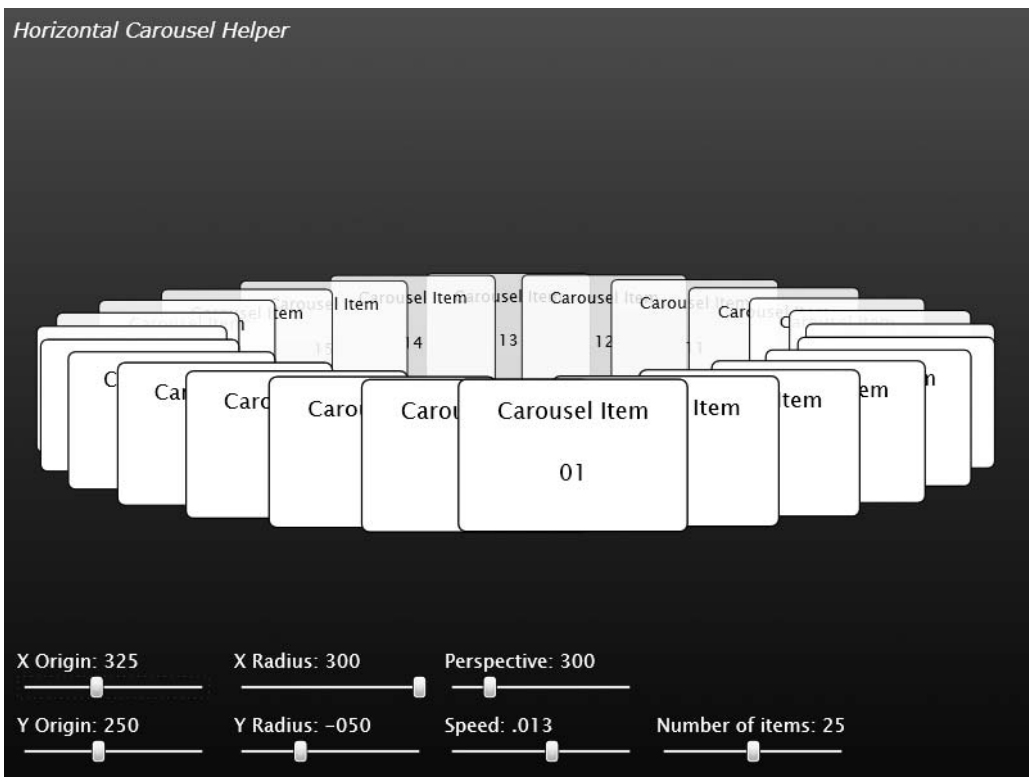

Figure 7-16. The HorizontalCarouselHelper application may help you determine optimal settings for your carousel application before you build.

Can you figure out how to add reflections to each object added on the carousel? What about rotating it with buttons rather than the mouse control? Would it work to put 30 items on a carousel, each item containing a frame from an animation, and then spin the carousel fast enough to see the motion from the individual frames?

X axis rotation

In this section, we'll finish out our look at emulating 3D movement in a 2D environment by looking at x axis rotation, which is illustrated in Figure 7-17.

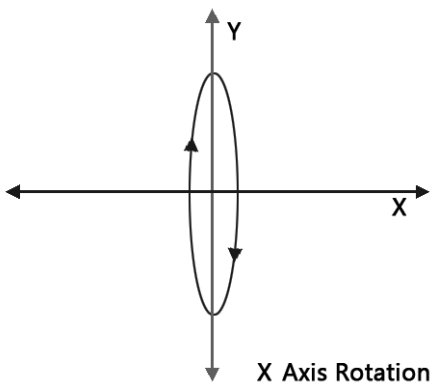

Figure 7-17. An illustration of rotation around the x axis

As with y axis rotations, scale and transparency will once again be our clues as to the location of the object within the context of 3D space. The technique for doing an x axis rotation is the same as it is for a y axis rotation, but is applied slightly differently.

1. Open the **XAxis3D** project to code along with this example. The project contains the main page canvas, as well as the `Ball` object we have been using.
2. All of the code for this example will once again go into the `Page.xaml.cs` file, so open that file for editing.
3. Add an instance of the `Ball` user control and the variables necessary to code up this example. You should be pretty familiar with each of the variables by now.

```
Ball MyBall = new Ball();
Point Origin;
Point Radius;
double Angle = 0;
double Speed = .1;
double Distance;
double Perspective = 400;
Point Position;
```

4. Inside the Page() constructor, initialize the Origin variables, and position the instance of the Ball object before adding it to the main canvas:

```
Origin.X = LayoutRoot.Width / 2 - MyBall.Width / 2;
Origin.Y = LayoutRoot.Height / 2 - MyBall.Height / 2;
Canvas.SetLeft(MyBall, Origin.X);
Canvas.SetTop(MyBall, Origin.Y);
LayoutRoot.Children.Add(MyBall);
```

5. Set the x and y Radius values:

```
Radius.X = 50;
Radius.Y = 200;
```

6. Finish up the constructor by adding a Completed event listener to the MoveBall storyboard and starting the storyboard:

```
MoveBall.Completed += new EventHandler(MoveBall_Completed);
MoveBall.Begin();
```

7. Inside the MoveBall_Completed() event handler, place the code to move the ball. This code is very nearly identical to that used to simulate y axis rotations in the previous section of the chapter. The only difference is that when calculating the Distance, Position.X is used rather than Position.Y.

```
void MoveBall_Completed(object sender, EventArgs e)
{
    Position.X = Radius.X * Math.Cos(Angle);
    Position.Y = Radius.Y * Math.Sin(Angle);
    Distance = 1 / (1 - (Position.X / Perspective));
    MyBall.Opacity = MyBall.BallScale.ScaleX
                   = MyBall.BallScale.ScaleY = Distance;
    Canvas.SetLeft(MyBall, Position.X + Origin.X);
    Canvas.SetTop(MyBall, Position.Y + Origin.Y);
    Angle += Speed;
    MoveBall.Begin();
}
```

Compile and run the application, and the ball will travel in an elliptical path around the x axis. The code shown in this example is included in the **XAxis3DCompleted** project. The project, shown in Figure 7-18, has been augmented with sliders and a check box that allow you to modify the program's values in real time.

Figure 7-18. The XAxis3DCompleted project includes some sliders to change values in real time.

A vertical carousel

Let's go ahead and apply this to a real-world situation. Since we created a horizontal carousel for the y axis rotation, we'll create a vertical, Rolodex-style carousel for our x axis example. Because the majority of the code for the vertical carousel is similar to that of the horizontal carousel, we'll start out a little farther along in the project.

1. Open the **VerticalCarousel** project. This project contains the main canvas, along with the `CarouselCanvas`, rectangle `MouseControl` object, and messaging TextBlock. The `CarouselItem` object is also already present in the project, and has had the publicly accessible `Angle` variable added.
2. All of the code we will be adding will go into the `Page.xaml.cs` file, so open that file for editing.

3. Begin by declaring a `List` to contain all of the `CarouselItem` objects that will be used in the application. Once again, declare a variable to control the number of elements in the carousel, as well as `Radius`, `Position`, `Speed`, `Distance`, and `Perspective` variables.

```
List<CarouselItem> Items;
int NumItems = 8;
Point Radius;
Point Position;
double Speed = .0125;
double Distance;
double Perspective = 300;
```

4. Inside the `Page()` constructor, add the following code to initialize the `List` of `CarouselItem` objects:

```
Items = new List<CarouselItem>(NumItems);
```

5. Assign values for the `Radius.X` and `Radius.Y` variables:

```
Radius.X = -25;
Radius.Y = 225;
```

6. Code up the resizing and positioning for the `MouseControl` rectangle element. The code here differs slightly from that used in the horizontal carousel. It is height-adjusted rather than width-adjusted, and the positioning calculation is a little different—recall that you may need to customize this a little depending upon the style of your carousel. Figure 7-19 shows where the `MouseControl` element is located for the vertical carousel.

```
MouseControl.Height = Math.Abs(Radius.Y * 2) + CarouselCanvas.Height;
Canvas.SetLeft(MouseControl, CarouselCanvas.Width);
Canvas.SetTop(MouseControl, -(Math.Abs(Radius.Y)));
MouseControl.MouseMove +=
        new MouseEventHandler(MouseControl_MouseMove);
```

7. Continue coding the `Page()` constructor by positioning the `CarouselCanvas`:

```
Canvas.SetLeft(CarouselCanvas, LayoutRoot.Width / 2
                - CarouselCanvas.Width / 2);
Canvas.SetTop(CarouselCanvas, LayoutRoot.Height / 2
               - CarouselCanvas.Height / 2);
```

8. Finish up the constructor by calling the `BuildCarousel()` function (which we have yet to code), adding an event listener to the `Completed` event for the `Spin` storyboard, and starting the storyboard:

```
BuildCarousel();
Spin.Completed +=new EventHandler(MoveCarousel);
Spin.Begin();
```

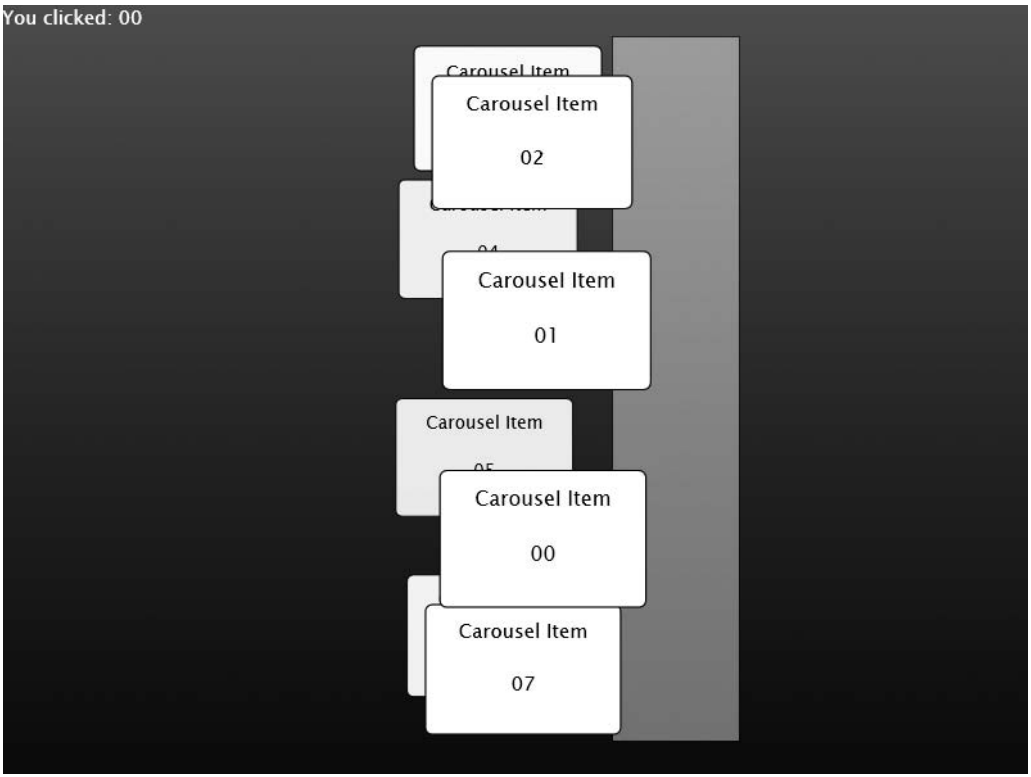

Figure 7-19. The MouseControl rectangle for the vertical carousel

9. We're working top-down through the code this time. We have three functions to build: `MouseControl_MouseMove()`, `BuildCarousel()`, and `MoveCarousel()`. We'll also be adding three more event listeners for the mouse events on the carousel items. We'll start with the `BuildCarousel()` function. This code is nearly identical to that used in the horizontal carousel, with one exception—the `Distance` calculation is based on the `Position.X` variable rather than `Position.Y`.

```
private void BuildCarousel()
{
    for (int i = 0; i < NumItems; i++)
    {
        CarouselItem item = new CarouselItem();
        item.MsgNumber.Text = String.Format("{0:00}", i);
        item.Angle = i * ((Math.PI * 2) / NumItems);
        Position.X = Math.Cos(item.Angle) * Radius.X;
        Position.Y = Math.Sin(item.Angle) * Radius.Y;
        Canvas.SetLeft(item, Position.X);
        Canvas.SetTop(item, Position.Y);
```

```
            item.MouseEnter += new MouseEventHandler(item_MouseEnter);
            item.MouseLeave += new MouseEventHandler(item_MouseLeave);
            item.MouseLeftButtonUp +=
                    new MouseButtonEventHandler(item_MouseLeftButtonUp);
            Distance = 1 / (1 - (Position.X / Perspective));
            item.Opacity = item.ItemScale.ScaleX
                         = item.ItemScale.ScaleY = Distance;
            Items.Add(item);
            CarouselCanvas.Children.Add(item);
        }
    }
```

10. Next, we'll tackle the three event handlers that were attached in step 9. The event handler code is identical to the code used in the horizontal carousel:

```
void item_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
{
    CarouselItem WhichItem = sender as CarouselItem;
    MsgSelected.Text = "You clicked: " + WhichItem.MsgNumber.Text;
}
void item_MouseLeave(object sender, MouseEventArgs e)
{
    Spin.Begin();
}
void item_MouseEnter(object sender, MouseEventArgs e)
{
    Spin.Stop();
}
```

11. Let's code up the `MouseControl_MouseMove()` event handler. The differences between this and the horizontal carousel are related to using y values rather than x. The `Speed` calculation is modified slightly to account for the vertical orientation of the carousel.

```
void MouseControl_MouseMove(object sender, MouseEventArgs e)
{
    Point MousePoint = e.GetPosition(MouseControl);
    double OffsetCalc = MousePoint.Y - (MouseControl.Height / 2);
    Speed = OffsetCalc / 10000;
    if (Speed < 0 && Speed < -0.0250) Speed = -0.0250;
    else if (Speed > 0 && Speed > 0.0250) Speed = 0.0250;
}
```

12. We'll finish off by adding the `MoveCarousel()` function shown following. The difference in code between the vertical and horizontal carousels comes in the test that is done against the `Radius.X` value, not the `Radius.Y` value.

```
private void MoveCarousel(object sender, EventArgs e)
{
    foreach (CarouselItem item in Items)
    {
        item.Angle -= Speed;
        Position.X = Math.Cos(item.Angle) * Radius.X;
        Position.Y = Math.Sin(item.Angle) * Radius.Y;
        Canvas.SetLeft(item, Position.X);
        Canvas.SetTop(item, Position.Y);
        if (Radius.X >= 0)
        {
            Distance = 1 * (1 - (Position.X / Perspective));
            Canvas.SetZIndex(item, -(int)Position.X);
        }
        else
        {
            Distance = 1 / (1 - (Position.X / Perspective));
            Canvas.SetZIndex(item, (int)Position.X);
        }
        item.Opacity = item.ItemScale.ScaleX
                     = item.ItemScale.ScaleY = Distance;
    }
    Spin.Begin();
}
```

When you compile and run, the application will look like the one shown in Figure 7-20. You can move the mouse up or down just to the right of the carousel to alter the speed/direction of rotation. Placing the pointer over an element stops the carousel, and moving the pointer off of an element will restart the carousel. Clicking an item will display the selected item number in the TextBlock at the top left of the screen.

As with the horizontal carousel, you can configure the application to meet your needs by adjusting the x or y radius values, perspective, or spin speed. All of the code covered in this example is available in the **VerticalCarouselCompleted** project.

As with the horizontal carousel, I included an extra project called **VerticalCarouselHelper**, which is shown in Figure 7-21. The program allows you to adjust many of the carousel parameters in real time in order to help you with some of the planning on your carousel application projects.

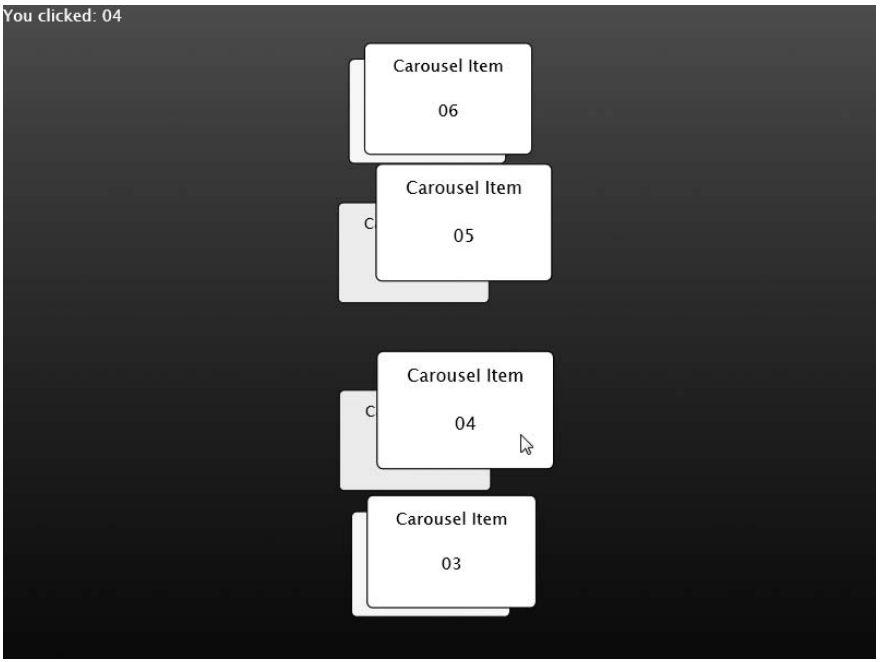

Figure 7-20. The vertical carousel application

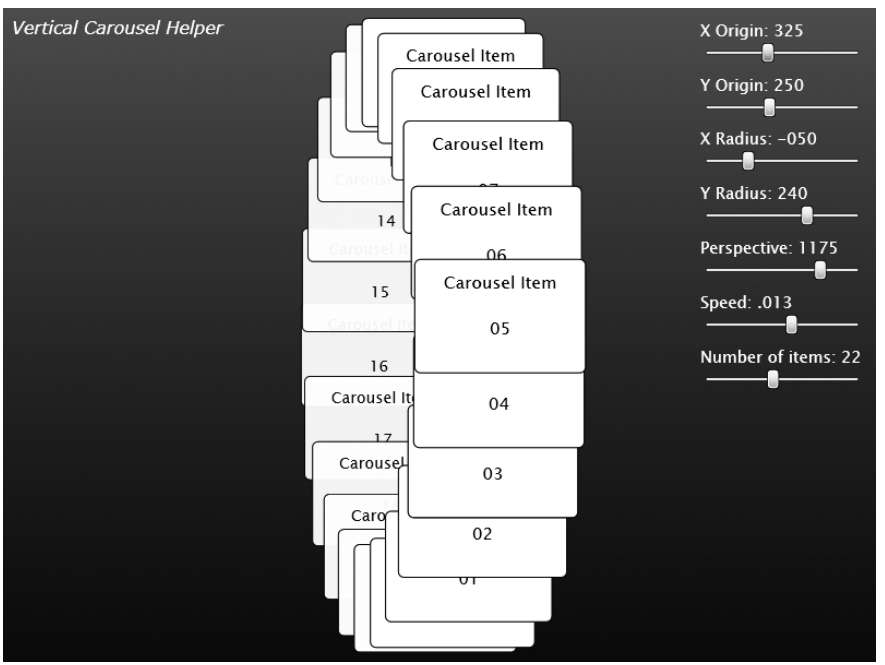

Figure 7-21. The vertical carousel helper application

Summary

In this chapter, we talked about how we can use some of the techniques from Chapter 6 to make objects appear as though they are moving in a 3D environment on our 2D canvas. We can trick our brains into seeing objects as 3D by using scale and transparency (distant objects are smaller and visually obscured, while close objects are large and well defined) in conjunction with Z-index.

Rotations around the z axis are really nothing more than circular or elliptical movements. You saw how several z axis rotations could be combined to create a simulation of the inner solar system.

In order to simulate a y axis rotation, an object must have its `Scale` and `Opacity` properties tied to its location on the path it is traveling. This will cause the object to scale down and become more transparent as it "moves away." As the object "moves forward," the object will become larger and more opaque. We explored y axis rotations by creating a horizontal carousel.

Our example of x axis rotation worked much like that for y axis rotation, except that the y radius we were using was larger than the x radius. As with y axis rotations, the `Scale` and `Opacity` properties for x axis rotation are tied to an object's location along the path it is traveling. To simulate a 3D x axis rotation, we built a vertically oriented carousel.

In Chapter 8, we're going to take a look at different methods we can use for collision detection in Silverlight. We'll create projects that demonstrate a few different scenarios, and hopefully give you a few ideas that you can apply in your own projects.