Chapter 21

Transforms

With the help of `StackLayout` and `Grid`, Xamarin.Forms does a good job of sizing and positioning visual elements on the page. Sometimes, however, it's necessary (or convenient) for the application to make some adjustments. You might want to offset the position of elements somewhat, change their size, or even rotate them.

Such changes in location, size, or orientation are possible using a feature of Xamarin.Forms known as *transforms*. The concept of the transform originated in geometry. The transform is a formula that maps points to other points. For example, if you want to shift a geometric object on a Cartesian coordinate system, you can add constant offset factors to all the coordinates that define that object.

These mathematical, geometric transforms play a vital role in computer graphics programming, where they are sometimes known as *matrix transforms* because they are easiest to express mathematically using matrix algebra. Without transforms, there can be no 3D graphics. But over the years, transforms have migrated from graphics programming to user-interface programming. All the platforms supported by Xamarin.Forms support basic transforms that can be applied to user-interface elements such as text, bitmaps, and buttons.

Xamarin.Forms supports three basic types of transforms:

- *Translation*—shifting an element horizontally or vertically or both.
- *Scale*—changing the size of an element.
- *Rotation*—turning an element around a point or axis.

The scaling supported by Xamarin.Forms is uniform in all directions, technically known as *isotropic* scaling. You cannot use scaling to change the aspect ratio of a visual element. Rotation is supported for both the two-dimensional surface of the screen and in 3D space. Xamarin.Forms does not support a skewing transform or a generalized matrix transform.

Xamarin.Forms supports these transforms with eight properties of the `VisualElement` class. These properties are all of type `double`:

- `TranslationX`
- `TranslationY`
- `Scale`
- `Rotation`
- `RotationX`

- `RotationY`
- `AnchorX`
- `AnchorY`

As you'll see in the next chapter, Xamarin.Forms also has an extensive and extensible animation system that can target these properties. But you can also perform transform animations on your own by using `Device.StartTimer` or `Task.Delay`. This chapter demonstrates some animation techniques and perhaps will help get you into an animation frame of mind in preparation for Chapter 22.

The translation transform

An application uses one of the layout classes—`StackLayout`, `Grid`, `AbsoluteLayout`, or `RelativeLayout`—to position a visual element on the screen. Let's call the position established by the layout system the "layout position."

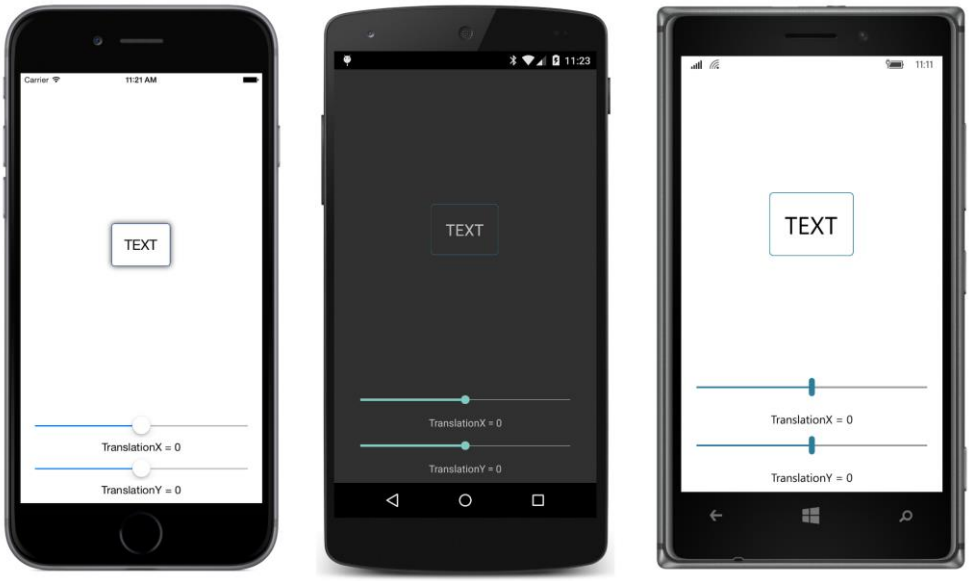
Nonzero values of the `TranslationX` and `TranslationY` properties change the position of a visual element relative to that layout position. Positive values of `TranslationX` shift the element to the right, and positive values of `TranslationY` shift the element down.

The **TranslationDemo** program lets you experiment with these two properties. Everything is in the XAML file:

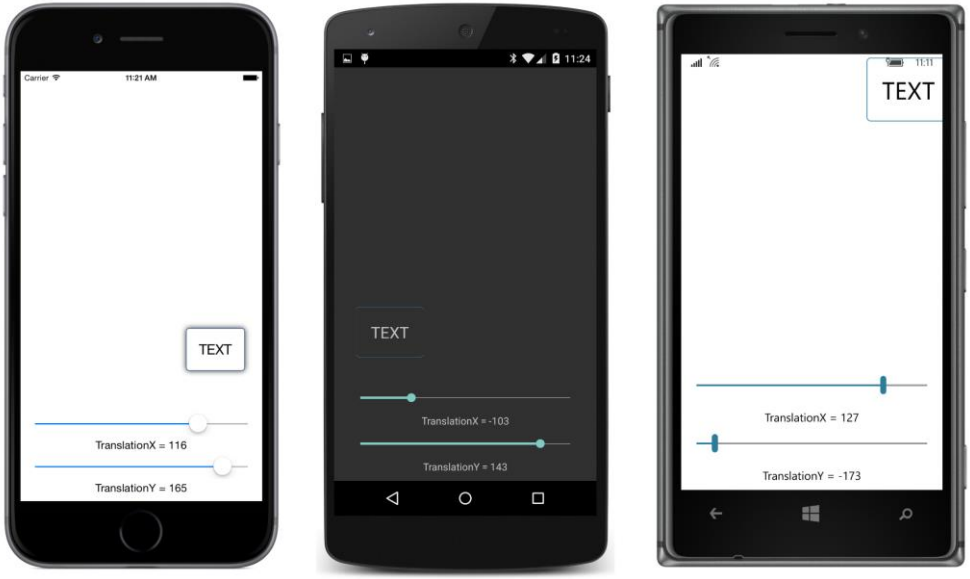
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="TranslationDemo.TranslationDemoPage">
    <StackLayout Padding="20, 10">
        <Frame x:Name="frame"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand"
               OutlineColor="Accent">

            <Label Text="TEXT"
                   FontSize="Large" />
        </Frame>

        <Slider x:Name="xSlider"
                Minimum="-200"
                Maximum="200"
                Value="{Binding Source={x:Reference frame},
                                Path=TranslationX}" />

        <Label Text="{Binding Source={x:Reference xSlider},
                              Path=Value,
                              StringFormat='TranslationX = {0:F0}'}"
               HorizontalTextAlignment="Center" />

        <Slider x:Name="ySlider"
```

```
                Minimum="-200"
                Maximum="200"
                Value="{Binding Source={x:Reference frame},
                                Path=TranslationY }" />

        <Label Text="{Binding Source={x:Reference ySlider},
                              Path=Value,
                              StringFormat='TranslationY = {0:F0}'}"
               HorizontalTextAlignment="Center" />
    </StackLayout>
</ContentPage>
```

A `Frame` encloses a `Label` and is centered in the upper part of the `StackLayout`. Two `Slider` elements have bindings to the `TranslationX` and `TranslationY` properties of the `Frame`, and they are initialized for a range of –200 to 200. When you first run the program, the two sliders are set to the default values of `TranslationX` and `TranslationY`, which are zero:



You can manipulate the sliders to move the `Frame` around the screen. The values of `TranslationX` and `TranslationY` specify an offset of the element relative to its original layout position:



If the values are large enough, the element can be translated to overlap other visuals, or to move off the screen entirely.

A translation of an element such as a `Frame` also affects all the children of that element, which in this case is just the `Label`. You can set the `TranslationX` and `TranslationY` properties on any `VisualElement`, and that includes `StackLayout`, `Grid`, and even `Page` and its derivatives. The transform is applied to the element and all the children of that element.

What might not be so evident without a little investigation is that `TranslationX` and `TranslationY` affect only how the element is *rendered*. These properties do *not* affect how the element is perceived within the layout system.

For example, `VisualElement` defines get-only properties named `X` and `Y` that indicate where an element is located relative to its parent. The `X` and `Y` properties are set when an element is positioned by its parent, and in this example, the `X` and `Y` properties of `Frame` indicate the location of the upper-left corner of the `Frame` relative to the upper-left corner of the `StackLayout`. The `X` and `Y` properties do *not* change when `TranslationX` and `TranslationY` are set. Also, the get-only `Bounds` property—which combines `X` and `Y` along with `Width` and `Height` in a single `Rectangle`—does not change either. The layout system does not get involved when `TranslationX` and `TranslationY` are modified.

What happens if you use `TranslationX` and `TranslationY` to move a `Button` from its original position? Does the `Button` respond to taps at its original layout position or the new rendered position? You'll be happy to know that it's the latter. `TranslationX` and `TranslationY` affect both how the element is rendered and how it responds to taps. You'll see this shortly in a sample program called **ButtonJump**.

If you need to do some extensive movement of elements around the page, you might wonder whether to use `AbsoluteLayout` and specify coordinates explicitly or use `TranslationX` and `TranslationY` to specify offsets. In terms of performance, there's really not much difference. The advantage of `TranslationX` and `TranslationY` is that you can start with a position established by `StackLayout` or `Grid` and then move the elements relative to that position.

Text effects

One common application of `TranslationX` and `TranslationY` is to apply little offsets to elements that shift them slightly from their layout position. This is sometimes useful if you have multiple overlapping elements in a single-cell `Grid` and need to shift one so that it peeks out from under another.

You can even use this technique for common text effects. The XAML-only **TextOffsets** program puts three pairs of `Label` elements in three single-cell `Grid` layouts. The pair of `Label` elements in each `Grid` are the same size and display the same text:

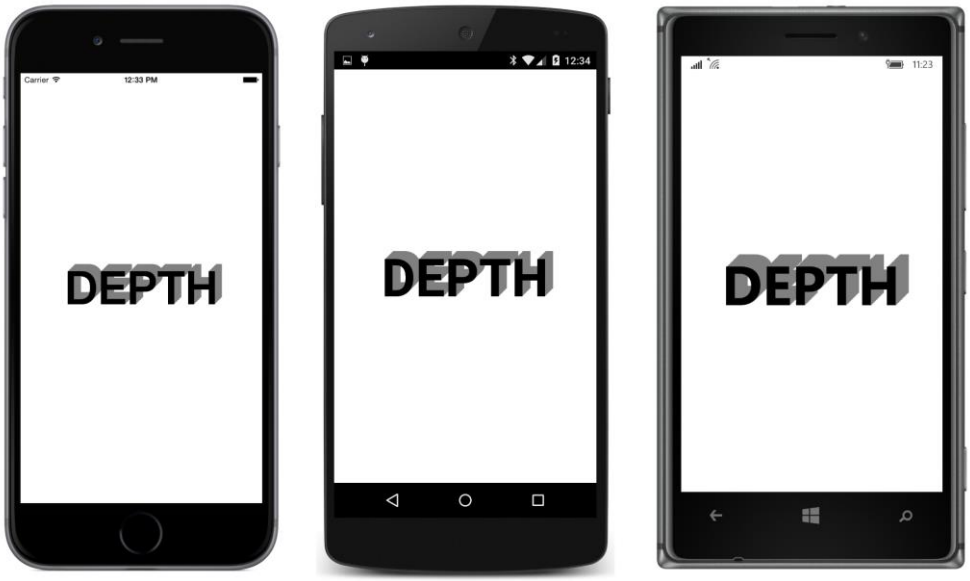
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="TextOffsets.TextOffsetsPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <ContentPage.Resources>
        <ResourceDictionary>
            <Color x:Key="backgroundColor">White</Color>
            <Color x:Key="foregroundColor">Black</Color>

            <Style TargetType="Grid">
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
            </Style>

            <Style TargetType="Label">
                <Setter Property="FontSize" Value="72" />
                <Setter Property="FontAttributes" Value="Bold" />
                <Setter Property="HorizontalOptions" Value="Center" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout BackgroundColor="{StaticResource backgroundColor}">
        <Grid>
            <Label Text="Shadow"
                   TextColor="{StaticResource foregroundColor}"
                   Opacity="0.5"
                   TranslationX="5"
                   TranslationY="5" />

            <Label Text="Shadow"
                   TextColor="{StaticResource foregroundColor}" />
```

```
        </Grid>

        <Grid>
            <Label Text="Emboss"
                   TextColor="{StaticResource foregroundColor}"
                   TranslationX="2"
                   TranslationY="2" />

            <Label Text="Emboss"
                   TextColor="{StaticResource backgroundColor}" />
        </Grid>

        <Grid>
            <Label Text="Engrave"
                   TextColor="{StaticResource foregroundColor}"
                   TranslationX="-2"
                   TranslationY="-2" />

            <Label Text="Engrave"
                   TextColor="{StaticResource backgroundColor}" />
        </Grid>
    </StackLayout>
</ContentPage>
```

Normally, the first `Label` in the `Children` collection of the `Grid` would be obscured by the second `Label`, but `TranslationX` and `TranslationY` values applied on the first `Label` allow it to be partially visible. The same basic technique results in three different text effects: a drop shadow, text that appears to be raised up from the surface of the screen, and text that looks like it's chiseled into the screen:

These effects give a somewhat 3D appearance to otherwise flat images. The optical illusion is based on a convention that light illuminates the screen from the upper-left corner. Therefore, shadows are thrown below and to the right of raised objects. The difference between the embossed and engraved effects is entirely due to the relative positions of the obscured black text and the white text on top. If the black text is a little below and to the right, it becomes the shadow of raised white text. If the black text is above and to the left of the white text, it becomes a shadow of text sunk below the surface.

The next example is not something you'll want to use on a regular basis because it requires multiple `Label` elements, but the technique illustrated in the **BlockText** program is useful if you want to supply a little "depth" to your text:



The **BlockText** XAML file uses a single-cell `Grid` to display black text on a white background. The implicit (and extensive) `Style` defined for `Label`, however, specifies a `TextColor` property of `Gray`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="BlockText.BlockTextPage">
    <Grid x:Name="grid"
          BackgroundColor="White">
        <Grid.Resources>
            <ResourceDictionary>
                <Style TargetType="Label">
                    <Setter Property="Text" Value="DEPTH" />
                    <Setter Property="FontSize" Value="72" />
                    <Setter Property="FontAttributes" Value="Bold" />
                    <Setter Property="TextColor" Value="Gray" />
                    <Setter Property="HorizontalOptions" Value="Center" />
                    <Setter Property="VerticalOptions" Value="Center" />
                </Style>
```

```
            </ResourceDictionary>
        </Grid.Resources>

        <Label TextColor="Black" />

    </Grid>
</ContentPage>
```

The constructor in the code-behind file adds several more `Label` elements to the `Grid`. The `Style` ensures that they all get the same properties (including being colored gray), but each of these is offset from the `Label` in the XAML file:

```
public partial class BlockTextPage : ContentPage
{
    public BlockTextPage()
    {
        InitializeComponent();

        for (int i = 0; i < Device.OnPlatform(12, 12, 18); i++)
        {
            grid.Children.Insert(0, new Label
                {
                    TranslationX = i,
                    TranslationY = -i
                });
        }
    }
}
```

Here's another case where `Label` elements overlap each other in the single-cell `Grid`, but now there are many more of them. The black `Label` in the XAML file must be the *last* child in the `Children` collection so that it's on top of all the others. The element with the maximum `TranslationX` and `TranslationY` offset must be the *first* child in the `Children` collection, so it must be on the very bottom of the pile. That's why each successive `Label` is inserted at the beginning of the `Children` collection.

Jumps and animations

The **ButtonJump** program is mostly intended to demonstrate that no matter where you move a `Button` on the screen by using translation, the `Button` still responds to taps in the normal manner. The XAML file centers the `Button` in the middle of the page (less the iOS padding at the top):

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ButtonJump.ButtonJumpPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <ContentView>
```

```
        <Button Text="Tap me!"
                FontSize="Large"
                HorizontalOptions="Center"
                VerticalOptions="Center"
                Clicked="OnButtonClicked" />
    </ContentView>
</ContentPage>
```

For each call to the `OnButtonClicked` handler, the code-behind file sets the `TranslationX` and `TranslationY` properties to new values. The new values are randomly calculated but restricted so that the `Button` always remains within the edges of the screen:

```
public partial class ButtonJumpPage : ContentPage
{
    Random random = new Random();

    public ButtonJumpPage()
    {
        InitializeComponent();
    }

    void OnButtonClicked(object sender, EventArgs args)
    {
        Button button = (Button)sender;
        View container = (View)button.Parent;

        button.TranslationX = (random.NextDouble() - 0.5) * (container.Width - button.Width);
        button.TranslationY = (random.NextDouble() - 0.5) * (container.Height - button.Height);
    }
}
```

For example, if the `Button` is 80 units wide and the `ContentView` is 320 units wide, the difference is 240 units, which is 120 units on each side of the `Button` when it's in the center of the `ContentView`. The `NextDouble` method of `Random` returns a number between 0 and 1, and subtracting 0.5 yields a number between –0.5 and 0.5, which means that `TranslationX` is set to a random value between –120 and 120. Those values potentially position the `Button` up to the edge of the screen but not beyond.

Keep in mind that `TranslationX` and `TranslationY` are properties rather than methods. They are not cumulative. If you set `TranslationX` to 100 and then to 200, the visual element isn't offset by a total of 300 units from its layout position. The second `TranslationX` value of 200 replaces rather than adds to the initial value of 100.

A few seconds playing with the **ButtonJump** program probably raises a question: Can this be animated? Can the `Button` glide to the new point rather than simply jump there?

Of course. There are several ways to do it, including the Xamarin.Forms animation methods discussed in the next chapter. The XAML file in the **ButtonGlide** program is the same as the one in **ButtonJump,** except that the `Button` now has a name so that the program can easily reference it outside the `Clicked` handler:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ButtonGlide.ButtonGlidePage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <ContentView>
        <Button x:Name="button"
                Text="Tap me!"
                FontSize="Large"
                HorizontalOptions="Center"
                VerticalOptions="Center"
                Clicked="OnButtonClicked" />
    </ContentView>
</ContentPage>
```

The code-behind file processes the button click by saving several essential pieces of information as fields: a `Point` indicating the starting location obtained from the current values of `TranslationX` and `TranslationY`; a vector (which is also a `Point` value) calculated by subtracting this starting point from a random destination point; and the current `DateTime` when the `Button` is clicked:

```
public partial class ButtonGlidePage : ContentPage
{
    static readonly TimeSpan duration = TimeSpan.FromSeconds(1);
    Random random = new Random();
    Point startPoint;
    Point animationVector;
    DateTime startTime;

    public ButtonGlidePage()
    {
        InitializeComponent();

        Device.StartTimer(TimeSpan.FromMilliseconds(16), OnTimerTick);
    }

    void OnButtonClicked(object sender, EventArgs args)
    {
        Button button = (Button)sender;
        View container = (View)button.Parent;

        // The start of the animation is the current Translation properties.
        startPoint = new Point(button.TranslationX, button.TranslationY);

        // The end of the animation is a random point.
        double endX = (random.NextDouble() - 0.5) * (container.Width - button.Width);
        double endY = (random.NextDouble() - 0.5) * (container.Height - button.Height);

        // Create a vector from start point to end point.
        animationVector = new Point(endX - startPoint.X, endY - startPoint.Y);
```

```
        // Save the animation start time.
        startTime = DateTime.Now;
    }

    bool OnTimerTick()
    {
        // Get the elapsed time from the beginning of the animation.
        TimeSpan elapsedTime = DateTime.Now - startTime;

        // Normalize the elapsed time from 0 to 1.
        double t = Math.Max(0, Math.Min(1, elapsedTime.TotalMilliseconds /
                                                duration.TotalMilliseconds));

        // Calculate the new translation based on the animation vector.
        button.TranslationX = startPoint.X + t * animationVector.X;
        button.TranslationY = startPoint.Y + t * animationVector.Y;
        return true;
    }
}
```

The timer callback is called every 16 milliseconds. That's not an arbitrary number! Video displays commonly have a hardware refresh rate of 60 times per second. Hence, every frame is active for about 16 milliseconds. Pacing the animation at this rate is optimum. Once every 16 milliseconds, the callback calculates an elapsed time from the beginning of the animation and divides it by the duration. That's a value typically called `t` (for *time*) that ranges from 0 to 1 over the course of the animation. This value is multiplied by the vector, and the result is added to `startPoint`. That's the new value of `TranslationX` and `TranslationY`.

Although the timer callback is called continuously while the application is running, the `TranslationX` and `TranslationY` properties remain constant when the animation has completed. However, you don't have to wait until the `Button` has stopped moving before you can tap it again. (You need to be quick, or you can change the duration property to something longer.) The new animation starts from the current position of the `Button` and entirely replaces the previous animation.

One of the advantages of calculating a normalized value of `t` is that it becomes fairly easy to modify that value so that the animation doesn't have a constant velocity. For example, try adding this statement after the initial calculation of `t`:

```
t = Math.Sin(t * Math.PI / 2);
```

When the original value of `t` is 0 at the beginning of the animation, the argument to `Math.Sin` is 0 and the result is 0. When the original value of `t` is 1, the argument to `Math.Sin` is π/2, and the result is 1. However, the values between those two points are not linear. When the initial value of `t` is 0.5, this statement recalculates `t` as the sine of 45 degrees, which is 0.707. So by the time the animation is half over, the `Button` has already moved 70 percent of the distance to its destination. Overall, you'll see an animation that is faster at the beginning and slower toward the end.

You'll see a couple of different approaches to animation in this chapter. Even when you've become familiar with the animation system that Xamarin.Forms provides, sometimes it's useful to do it yourself.

The scale transform

The `VisualElement` class defines a property named `Scale` that you can use to change the rendered size of an element. The `Scale` property does *not* affect layout (as will be demonstrated in the **ButtonScaler** program). It does *not* affect the get-only `Width` and `Height` properties of the element, or the get-only `Bounds` property that incorporates those `Width` and `Height` values. Changes to the `Scale` property do *not* cause a `SizeChanged` event to be triggered.

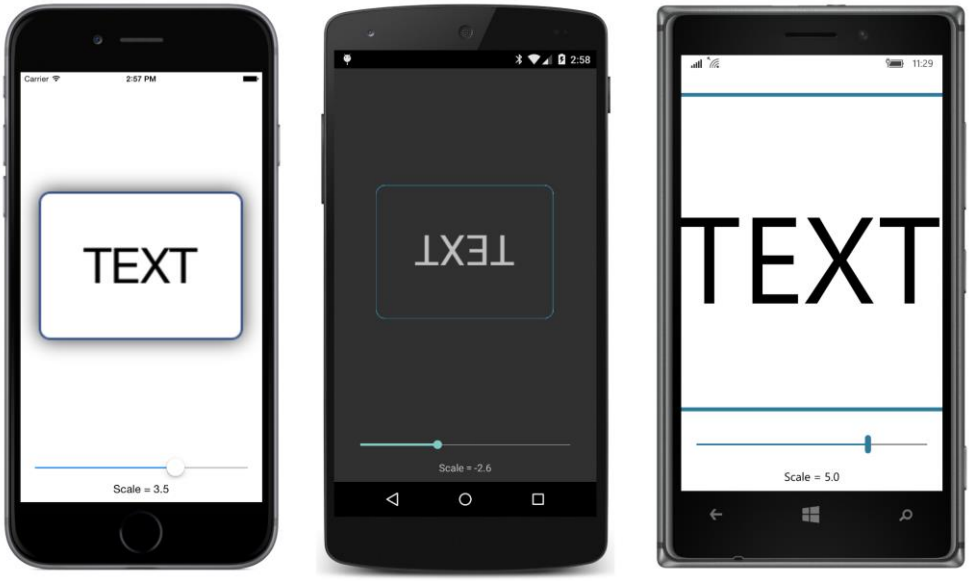
`Scale` affects the coordinates of a rendered visual element, but in a very different way from `TranslationX` and `TranslationY`. The two translation properties add values to coordinates, while the `Scale` property is multiplicative. The default value of `Scale` is 1. Values greater than 1 increase the size of the element. For example, a value of 3 makes the element three times its normal size. Values less than 1 decrease the size. A `Scale` value of 0 is legal but causes the element to be invisible. If you're working with `Scale` and your element seems to have disappeared, check whether it's somehow getting a `Scale` value of 0.

Values less than 0 are also legal and cause the element to be rotated 180 degrees besides being altered in size.

You can experiment with `Scale` settings using the **SimpleScaleDemo** program. (The program has a **Simple** prefix because it doesn't include the effect of the `AnchorX` and `AnchorY` properties, which will be discussed shortly.) The XAML is similar to the **TranslationDemo** program:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="SimpleScaleDemo.SimpleScaleDemoPage">
    <StackLayout Padding="20, 10">
        <Frame x:Name="frame"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand"
               OutlineColor="Accent">

            <Label Text="TEXT"
                   FontSize="Large" />
        </Frame>

        <Slider x:Name="scaleSlider"
                Minimum="-10"
                Maximum="10"
                Value="{Binding Source={x:Reference frame},
                                Path=Scale}" />

        <Label Text="{Binding Source={x:Reference scaleSlider},
                              Path=Value,
                              StringFormat='Scale = {0:F1}'}"
               HorizontalTextAlignment="Center" />
    </StackLayout>
</ContentPage>
```

Here it is in action. Notice the negative `Scale` setting on the Android phone:



On the Windows 10 Mobile display, the `Frame` has been scaled so large that you can't see its left and right sides.

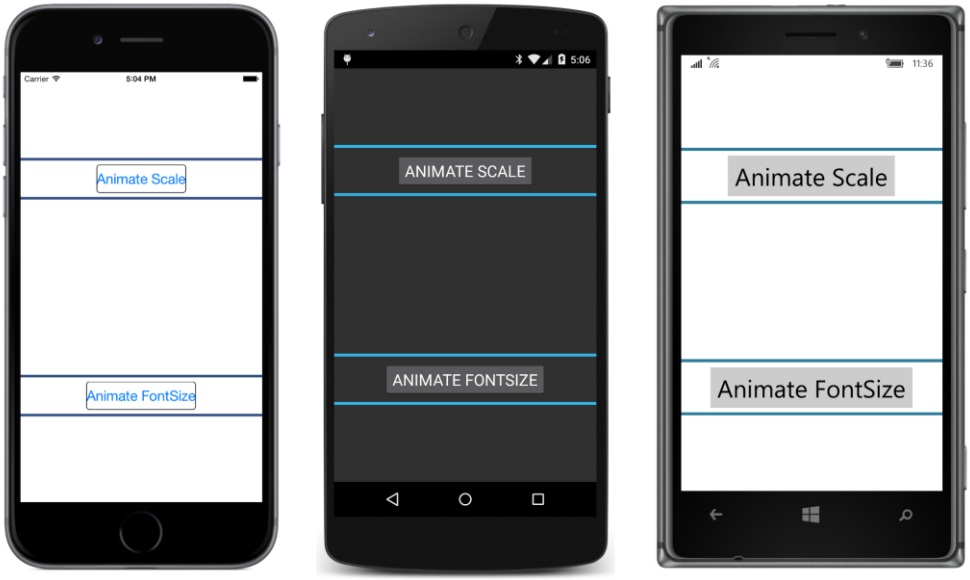
In real-life programming, you might want to use `Scale` to provide a little feedback to a user when a `Button` is clicked. The `Button` can briefly expand in size and go back down to normal again. However, `Scale` is not the only way to change the size of a `Button`. You can also change the `Button` size by increasing and decreasing the `FontSize` property. These two techniques are very different, however: The `Scale` property doesn't affect layout, but the `FontSize` property does.

This difference is illustrated in the **ButtonScaler** program. The XAML file consists of two `Button` elements sandwiched between two pairs of `BoxView` elements:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ButtonScaler.ButtonScalerPage">
    <StackLayout>
        <!-- "Animate Scale" Button between two BoxViews. -->
        <BoxView Color="Accent"
                 HeightRequest="4"
                 VerticalOptions="EndAndExpand" />

        <Button Text="Animate Scale"
                FontSize="Large"
                BorderWidth="1"
                HorizontalOptions="Center"
                Clicked="OnAnimateScaleClicked" />

        <BoxView Color="Accent"
```

```
                HeightRequest="4"
                VerticalOptions="StartAndExpand" />

        <!-- "Animate FontSize" Button between two BoxViews. -->
        <BoxView Color="Accent"
                 HeightRequest="4"
                 VerticalOptions="EndAndExpand" />

        <Button Text="Animate FontSize"
                FontSize="Large"
                BorderWidth="1"
                HorizontalOptions="Center"
                Clicked="OnAnimateFontSizeClicked" />

        <BoxView Color="Accent"
                 HeightRequest="4"
                 VerticalOptions="StartAndExpand" />
    </StackLayout>
</ContentPage>
```

Here's what the page looks like normally:



The code-behind file implements a somewhat generalized animation method. It's generalized in the sense that the parameters include two values indicating the starting value and the ending value of the animation. These two values are often called a *from* value and a *to* value. The animation arguments also include the duration of the animation and a callback method. The argument to the callback method is a value between the "from" value and the "to" value, and the calling method can use that value to do whatever it needs to implement the animation.

However, this animation method is not entirely generalized. It actually calculates a value from the

from value to the *to* value during the first half of the animation, and then calculates a value from the *to* value back to the *from* value during the second half of the animation. This is sometimes called a *reversing* animation.

The method is called `AnimateAndBack`, and it uses a `Task.Delay` call to pace the animation and a .NET `Stopwach` object to determine elapsed time:

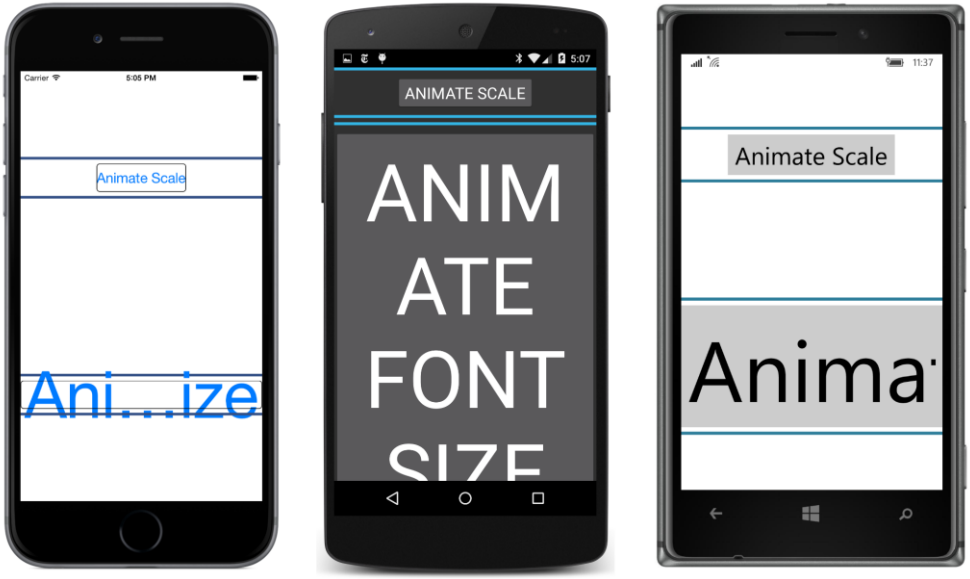
```
public partial class ButtonScalerPage : ContentPage
{
    public ButtonScalerPage()
    {
        InitializeComponent();
    }

    void OnAnimateScaleClicked(object sender, EventArgs args)
    {
        Button button = (Button)sender;
        AnimateAndBack(1, 5, TimeSpan.FromSeconds(3), (double value) =>
            {
                button.Scale = value;
            });
    }

    void OnAnimateFontSizeClicked(object sender, EventArgs args)
    {
        Button button = (Button)sender;

        AnimateAndBack(button.FontSize, 5 * button.FontSize,
                       TimeSpan.FromSeconds(3), (double value) =>
        {
            button.FontSize = value;
        });
    }

    async void AnimateAndBack(double fromValue, double toValue,
                              TimeSpan duration, Action<double> callback)
    {
        Stopwatch stopWatch = new Stopwatch();
        double t = 0;
        stopWatch.Start();

        while (t < 1)
        {
            double tReversing = 2 * (t < 0.5 ? t : 1 - t);
            callback(fromValue + (toValue - fromValue) * tReversing);
            await Task.Delay(16);
            t = stopWatch.ElapsedMilliseconds / duration.TotalMilliseconds;
        }

        stopWatch.Stop();
        callback(fromValue);
    }
}
```

The `Clicked` handlers for the two buttons each start an independent animation. The `Clicked` handler for the first `Button` animates its `Scale` property from 1 to 5 and back again, while the `Clicked` handler for the second `Button` animates its `FontSize` property with a scaling factor from 1 to 5 and back again.

Here's the animation of the `Scale` property about midway through:

As you can see, the scaling of the `Button` takes no regard of anything else that might be on the screen, and on the iOS and Windows 10 Mobile screens you can actually see through transparent areas of the `Button` to the top `BoxView` elements, while the opaque Android `Button` entirely hides that top `BoxView`. The `BoxView` below that top `Button` actually sits on top of the `Button` and is visible on all three platforms.

An animated increase of the `FontSize` property is handled a little differently on the three platforms:



On iOS, the `Button` text is truncated in the middle and the `Button` remains the same height. On Android, the `Button` text wraps and the enlarged `Button` pushes the two `BoxView` elements aside. The Windows Runtime `Button` also truncates the text but in a different way than iOS, and like Android, the increased `Button` height also pushes the two `BoxView` elements away.

Animating the `Scale` property does not affect layout, but animating the `FontSize` property obviously does affect layout.

The little animation system implemented in **ButtonScaler** can animate the two buttons independently and simultaneously, but it nevertheless has a severe flaw. Try tapping a `Button` while that `Button` is currently being animated. A new animation will start up for that `Button`, and the two animations will interfere with each other.

There are a couple of ways to fix this. One possibility is to include a `CancellationToken` value as an argument to the `AnimateAndBack` method so that the method can be cancelled. (You can pass this same `CancellationToken` value to the `Task.Delay` call.) This would allow the `Clicked` handler for the `Button` to cancel any ongoing animations before it begins a new one.

Another option is for `AnimateAndBack` to return a `Task` object. This allows the `Clicked` handler for the buttons to use the `await` operator with `AnimateAndBack`. The `Button` can easily disable itself before calling `AnimateAndBack` and reenable itself when `AnimateAndBack` has completed the animation.

At any rate, if you want to implement feedback to the user with a brief increase and decrease in `Button` size, it's safer and more efficient to animate `Scale` rather than `FontSize`. You'll see other techniques to do this in the next chapter on animation, and in Chapter 23, "Triggers and behaviors."

Another use of the `Scale` property is sizing an element to fit the available space. You might recall the **FitToSizeClock** program toward the end of Chapter 5, "Dealing with sizes." You can do something very similar with the `Scale` property, but you won't need to make estimations or recursive calculations.

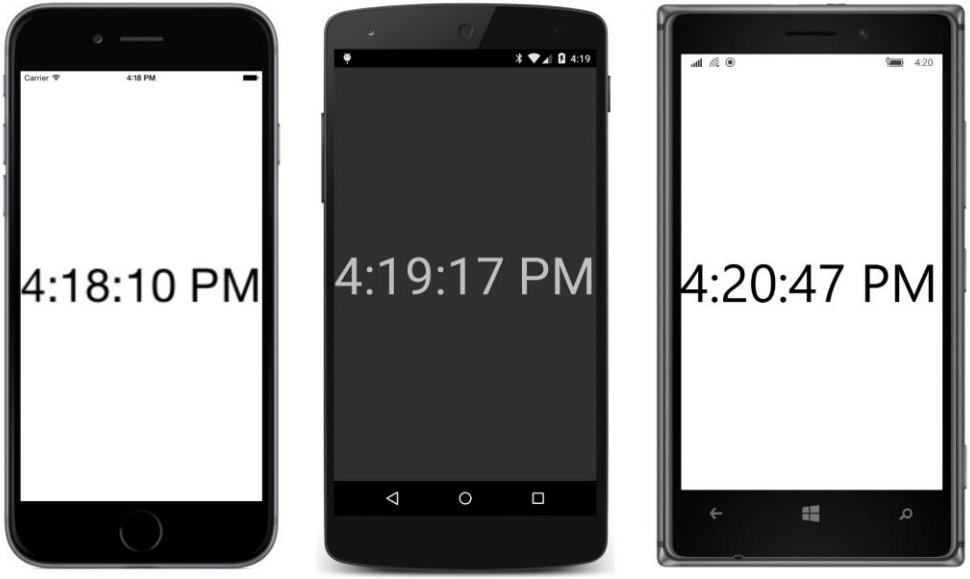
The XAML file of the **ScaleToSize** program contains a `Label` missing some text and also missing a `Scale` setting to make the `Label` larger:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ScaleToSize.ScaleToSizePage"
             SizeChanged="OnSizeChanged">

    <Label x:Name="label"
           HorizontalOptions="Center"
           VerticalOptions="Center"
           SizeChanged="OnSizeChanged" />

</ContentPage>
```

Both the `ContentPage` and the `Label` have `SizeChanged` handlers installed, and they both use the same handler. This handler simply sets the `Scale` property of the `Label` to the minimum of the width and height of the page divided by the width and height of the `Label`:

```
public partial class ScaleToSizePage : ContentPage
{
    public ScaleToSizePage()
    {
        InitializeComponent();
        UpdateLoop();
    }

    async void UpdateLoop()
    {
        while (true)
        {
            label.Text = DateTime.Now.ToString("T");
            await Task.Delay(1000);
        }
    }

    void OnSizeChanged(object sender, EventArgs args)
    {
        label.Scale = Math.Min(Width / label.Width, Height / label.Height);
    }
}
```

Because setting the `Scale` property doesn't trigger another `SizeChanged` event, there's no danger of triggering an endless recursive loop. But an actual infinite loop using `Task.Delay` keeps the `Label` updated with the current time:



Of course, turning the phone sideways makes the `Label` larger:

And here you can detect a little difference in the implementation of the `Scale` property in iOS compared with Android and the Windows Runtime. On Android and Windows, the resultant text looks as though it were drawn with a large font. However, the text on the iOS screen looks a little fuzzy. This fuzziness occurs when the operating system *rasterizes* the prescaled `Label`, which means that the operating system turns it into a bitmap. The bitmap is then expanded based on the `Scale` setting.

Anchoring the scale

As you've experimented with the `Scale` property, you've probably noticed that any expansion of the visual element occurs outward from the center of the element, and if you shrink a visual element down to nothing, it contracts toward the center as well.

Here's another way to think about it: The point in the very center of the visual element remains in the same location regardless of the setting of the `Scale` property.

If you're using the `Scale` property to expand a `Button` for visual feedback, or to fit a visual element within a particular space, that's probably precisely what you want. However, for some other applications, you might instead prefer that another point remains in the same location with changes to the `Scale` property. Perhaps you want the upper-left corner of the visual element to remain in the same spot and for expansion of the object to occur toward the right and bottom.

You can control the scaling center with the `AnchorX` and `AnchorY` properties. These properties are of type `double` and are relative to the element being transformed. An `AnchorX` value of 0 indicates the left side of the element, and a value of 1 is the right side of the element. An `AnchorY` value of 0 is the top and 1 is the bottom. The default values are 0.5, which is the center. Setting both properties to 0 allows scaling to be relative to the upper-left corner of the element.

You can also set the properties to values less than 0 or greater than 1, in which case the center of scaling is outside the bounds of the element.

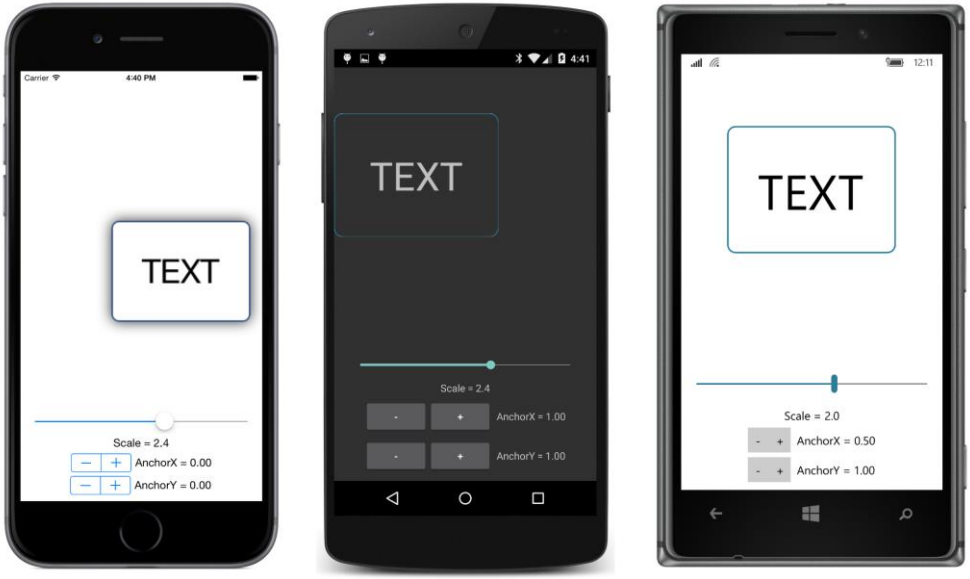
As you'll see, the `AnchorX` and `AnchorY` properties also affect rotation. Rotation occurs around a particular point called the *center of rotation*, and these two properties set that point relative to the element being rotated.

The **AnchoredScaleDemo** program lets you experiment with `AnchorX` and `AnchorY` as they affect the `Scale` property. The XAML files contains two `Stepper` views that let you change the `AnchorX` and `AnchorY` properties from –1 to 2 in increments of 0.25:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="AnchoredScaleDemo.AnchoredScaleDemoPage">
    <StackLayout Padding="20, 10">
        <Frame x:Name="frame"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand"
               OutlineColor="Accent">
            <Label Text="TEXT"
                   FontSize="Large" />
        </Frame>

        <Slider x:Name="scaleSlider"
                Minimum="-10"
                Maximum="10"
                Value="{Binding Source={x:Reference frame},
                                Path=Scale}" />
```

```
        <Label Text="{Binding Source={x:Reference scaleSlider},
                              Path=Value,
                              StringFormat='Scale = {0:F1}'}"
               HorizontalTextAlignment="Center" />

        <StackLayout Orientation="Horizontal"
                     HorizontalOptions="Center">
            <Stepper x:Name="anchorXStepper"
                     Minimum="-1"
                     Maximum="2"
                     Increment="0.25"
                     Value="{Binding Source={x:Reference frame},
                                     Path=AnchorX}" />

            <Label Text="{Binding Source={x:Reference anchorXStepper},
                                  Path=Value,
                                  StringFormat='AnchorX = {0:F2}'}"
                   VerticalOptions="Center"/>
        </StackLayout>

        <StackLayout Orientation="Horizontal"
                     HorizontalOptions="Center">
            <Stepper x:Name="anchorYStepper"
                     Minimum="-1"
                     Maximum="2"
                     Increment="0.25"
                     Value="{Binding Source={x:Reference frame},
                                     Path=AnchorY}" />

            <Label Text="{Binding Source={x:Reference anchorYStepper},
                                  Path=Value,
                                  StringFormat='AnchorY = {0:F2}'}"
                   VerticalOptions="Center"/>
        </StackLayout>
    </StackLayout>
</ContentPage>
```

Here are some screenshots showing (from left to right) scaling that is relative to the upper-left corner, relative to the lower-right corner, and relative to the center bottom:



If you are familiar with iOS programming, you know about the similar `anchorPoint` property. In iOS, this property affects both positioning and the transform center. In Xamarin.Forms, the `AnchorX` and `AnchorY` properties specify only the transform center.

This means that the iOS implementation of Xamarin.Forms must compensate for the difference between `anchorPoint` and the `AnchorX` and `AnchorY` properties, and in the latest version of Xamarin.Forms available as this edition was going to print, that compensation is not working quite right.

To see the problem, deploy the **AnchoredScaleDemo** program to an iPhone or iPhone simulator. Leave `Scale` set at its default value of 1, but set both `AnchorX` and `AnchorY` to 1. The `Frame` with the `Label` should not move from the center of its slot in the `StackLayout` because the `AnchorX` and `AnchorY` properties should only affect the center of scaling and rotation.

Now change the orientation of the phone or simulator from portrait to landscape. The `Frame` is no longer centered. Now change it back to portrait. It doesn't return to its original centered position.

This problem affects every program in this chapter (and the next chapter) that use nondefault values of `AnchorX` and `AnchorY`. Sometimes the sample programs in these chapters set `AnchorX` and `AnchorY` after an element has been resized to try to avoid the problem, but as long as the phone can change orientation from portrait to landscape, the problem cannot be circumvented, and there's nothing an application can do to compensate for the problem.

The rotation transform

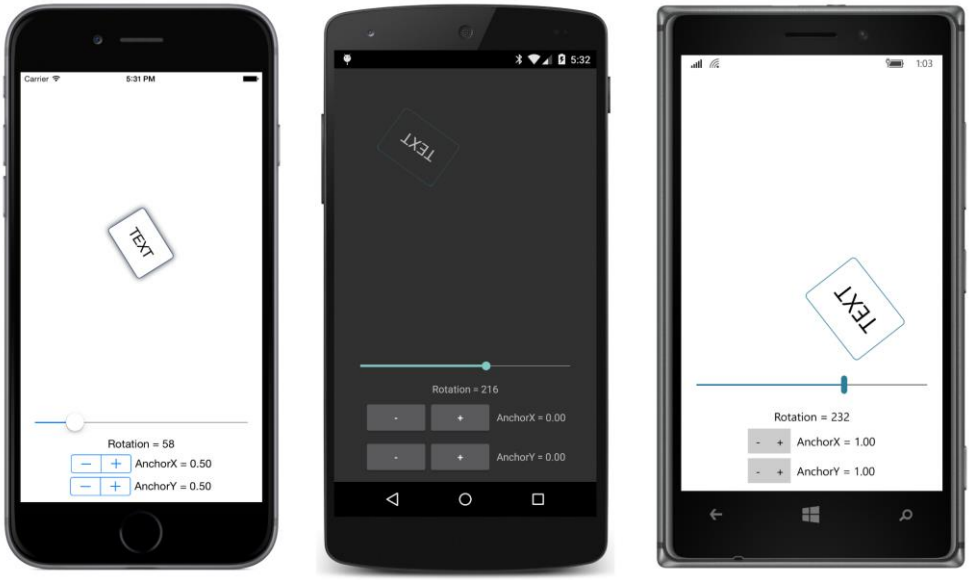
The `Rotation` property rotates a visual element on the surface of the screen. Set the `Rotation` property to an angle in degrees (not radians). Positive angles rotate the element clockwise. You can set `Rotation` to angles less than 0 or greater than 360. The actual rotation angle is the value of the `Rotation` property modulo 360. The element is rotated around a point relative to itself specified with the `AnchorX` and `AnchorY` properties.

The **PlaneRotationDemo** program lets you experiment with these three properties. The XAML file is very similar to the **AnchoredScaleDemo** program:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="PlaneRotationDemo.PlaneRotationDemoPage">
    <StackLayout Padding="20, 10">
        <Frame x:Name="frame"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand"
               OutlineColor="Accent">
            <Label Text="TEXT"
                   FontSize="Large" />
        </Frame>

        <Slider x:Name="rotationSlider"
                Maximum="360"
                Value="{Binding Source={x:Reference frame},
                                Path=Rotation}" />

        <Label Text="{Binding Source={x:Reference rotationSlider},
                              Path=Value,
                              StringFormat='Rotation = {0:F0}'}"
               HorizontalTextAlignment="Center" />

        <StackLayout Orientation="Horizontal"
                     HorizontalOptions="Center">
            <Stepper x:Name="anchorXStepper"
                     Minimum="-1"
                     Maximum="2"
                     Increment="0.25"
                     Value="{Binding Source={x:Reference frame},
                                     Path=AnchorX}" />

            <Label Text="{Binding Source={x:Reference anchorXStepper},
                                  Path=Value,
                                  StringFormat='AnchorX = {0:F2}'}"
                   VerticalOptions="Center"/>
        </StackLayout>

        <StackLayout Orientation="Horizontal"
                     HorizontalOptions="Center">
            <Stepper x:Name="anchorYStepper"
```

```
                    Minimum="-1"
                    Maximum="2"
                    Increment="0.25"
                    Value="{Binding Source={x:Reference frame},
                                    Path=AnchorY}" />

            <Label Text="{Binding Source={x:Reference anchorYStepper},
                                  Path=Value,
                                  StringFormat='AnchorY = {0:F2}'}"
                   VerticalOptions="Center"/>
        </StackLayout>
    </StackLayout>
</ContentPage>
```

Here are several combinations of `Rotation` angles and rotation centers:



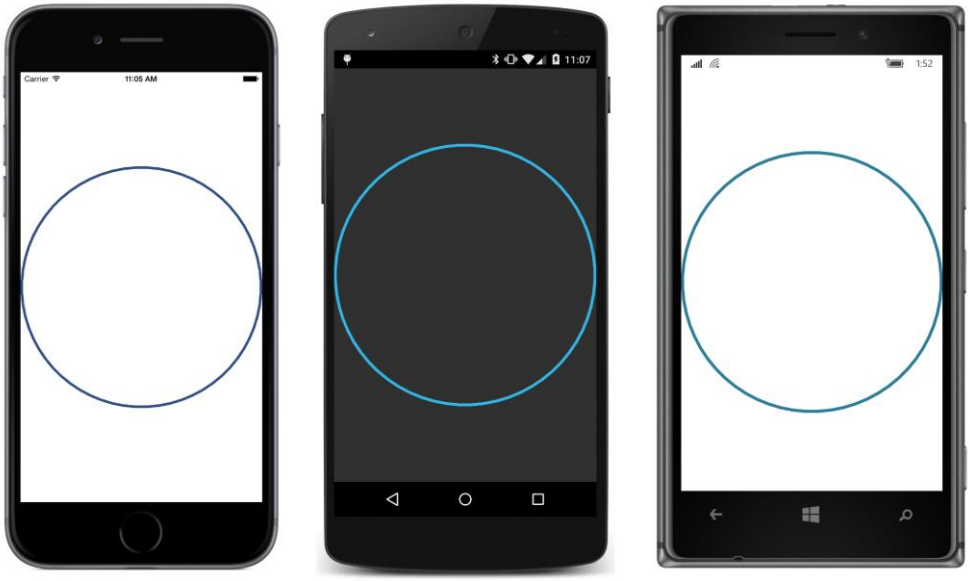
The iOS screen shows rotation around the center of the element (which is always safe on iOS despite the `AnchorX` and `AnchorY` bug), while the rotation on the Android screen is around the upper-left corner, and the rotation on the Windows 10 Mobile screen is centered on the bottom-right corner.

Rotated text effects

Rotation is fun. It's more fun when rotation is animated (as you'll see in the next chapter), but it's fun even with static images.

Several of the rotation examples in this chapter and the next involve arranging visual elements in a circle, so let's begin by attempting to display a simple circle. Of course, without an actual graphics system in Xamarin.Forms, we'll need to be inventive and construct this circle with `BoxView`. If you use many small `BoxView` elements and arrange them properly, it should be possible to create something

that looks like a smooth round circle, like this:



Each circle is composed of 64 `BoxView` elements, each of which is 4 units in thickness. These two values are defined as constants in the code-only **BoxViewCircle** program:

```
public class BoxViewClockPage : ContentPage
{
    const int COUNT = 64;
    const double THICKNESS = 4;

    public BoxViewClockPage()
    {
        AbsoluteLayout absoluteLayout = new AbsoluteLayout();
        Content = absoluteLayout;

        for (int index = 0; index < COUNT; index++)
        {
            absoluteLayout.Children.Add(new BoxView
                {
                    Color = Color.Accent,
                });
        }

        absoluteLayout.SizeChanged += (sender, args) =>
            {
                Point center = new Point(absoluteLayout.Width / 2, absoluteLayout.Height / 2);
                double radius = Math.Min(absoluteLayout.Width, absoluteLayout.Height) / 2;
                double circumference = 2 * Math.PI * radius;
                double length = circumference / COUNT;

                for (int index = 0; index < absoluteLayout.Children.Count; index++)
```

```
                {
                    BoxView boxView = (BoxView)absoluteLayout.Children[index];

                    // Position every BoxView at the top.
                    AbsoluteLayout.SetLayoutBounds(boxView,
                        new Rectangle(center.X - length / 2,
                                      center.Y - radius,
                                      length,
                                      THICKNESS));

                    // Set the AnchorX and AnchorY properties so rotation is
                    //      around the center of the AbsoluteLayout.
                    boxView.AnchorX = 0.5;
                    boxView.AnchorY = radius / THICKNESS;

                    // Set a unique Rotation for each BoxView.
                    boxView.Rotation = index * 360.0 / COUNT;
                }
            };
    }
}
```

All the calculations occur in the `SizeChanged` handler of the `AbsoluteLayout`. The minimum of the width and height of the `AbsoluteLayout` is the radius of a circle. Knowing that radius allows calculating a circumference, and hence a length for each individual `BoxView`.

The `for` loop positions each `BoxView` in the same location: at the center top of the circle. Each `BoxView` must then be rotated around the center of the circle. This requires setting an `AnchorY` property that corresponds to the distance from the top of the `BoxView` to the center of the circle. That distance is the `radius` value, but it must be in units of the `BoxView` height, which means that `radius` must be divided by `THICKNESS`.

There's an alternative way to position and rotate each `BoxView` that doesn't require setting the `AnchorX` and `AnchorY` properties. This approach is better for iOS. The `for` loop begins by calculating `x` and `y` values corresponding to the center of each `BoxView` around the perimeter of the circle. These calculations require using sine and cosine functions with a `radius` value that compensates for the thickness of the `BoxView`:

```
for (int index = 0; index < absoluteLayout.Children.Count; index++)
{
    BoxView boxView = (BoxView)absoluteLayout.Children[index];

    // Find point in center of each positioned BoxView.
    double radians = index * 2 * Math.PI / COUNT;
    double x = center.X + (radius - THICKNESS / 2) * Math.Sin(radians);
    double y = center.Y - (radius - THICKNESS / 2) * Math.Cos(radians);

    // Position each BoxView at that point.
    AbsoluteLayout.SetLayoutBounds(boxView,
        new Rectangle(x - length / 2,
                      y - THICKNESS / 2,
```

```
                    length,
                    THICKNESS));

    // Set a unique Rotation for each BoxView.
    boxView.Rotation = index * 360.0 / COUNT;
}
```

The `x` and `y` values indicate the position desired for the center of each `BoxView`, while `AbsoluteLayout.SetLayoutBounds` requires the location of the top-left corner of each `BoxView`, so these `x` and `y` values are adjusted for that difference when used with `SetLayoutBounds`. Each `BoxView` is then rotated around its own center.

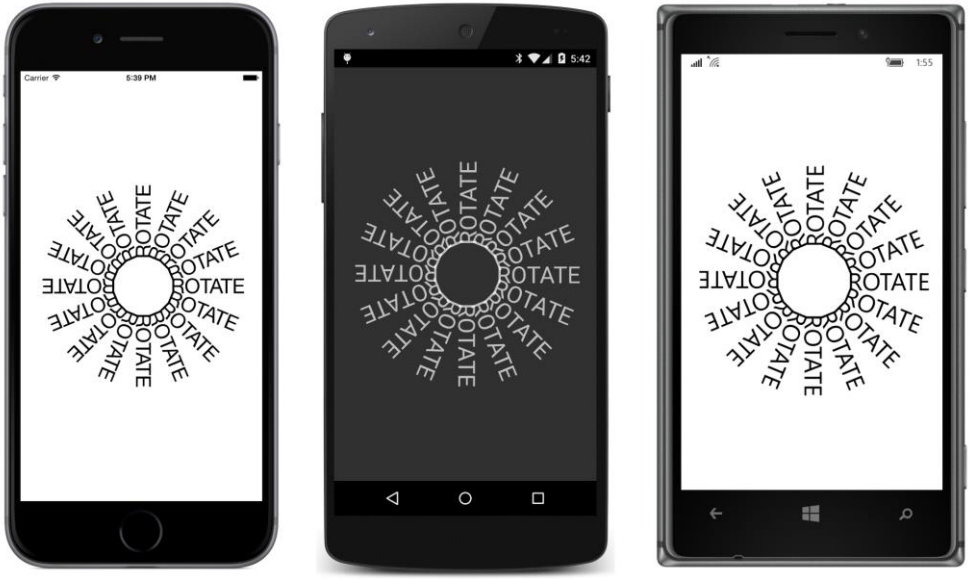
Now let's rotate some text. The **RotatedText** program is implemented entirely in XAML:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="RotatedText.RotatedTextPage">
    <Grid>
        <Grid.Resources>
            <ResourceDictionary>
                <Style TargetType="Label">
                    <Setter Property="Text" Value="     ROTATE" />
                    <Setter Property="FontSize" Value="32" />
                    <Setter Property="Grid.Column" Value="1" />
                    <Setter Property="VerticalOptions" Value="Center" />
                    <Setter Property="HorizontalOptions" Value="Start" />
                    <Setter Property="AnchorX" Value="0" />
                </Style>
            </ResourceDictionary>
        </Grid.Resources>

        <Label Rotation="0" />
        <Label Rotation="22.5" />
        <Label Rotation="45" />
        <Label Rotation="67.5" />
        <Label Rotation="90" />
        <Label Rotation="112.5" />
        <Label Rotation="135" />
        <Label Rotation="157.5" />
        <Label Rotation="180" />
        <Label Rotation="202.5" />
        <Label Rotation="225" />
        <Label Rotation="246.5" />
        <Label Rotation="270" />
        <Label Rotation="292.5" />
        <Label Rotation="315" />
        <Label Rotation="337.5" />
    </Grid>
</ContentPage>
```

The program consists of 16 `Label` elements in a `Grid` with an implicit `Style` setting six properties, including the `Text` and `FontSize`. Although this `Grid` might seem to be only a single cell, it's actually a two-column `Grid` because the `Style` sets the `Grid.Column` property of each `Label` to 1, which is

the second column. The `Style` centers each `Label` vertically within the second column and starts it at the left of that column, which is the center of the page. However, the text begins with several blank spaces, so it seems to start a bit to the right of the center of the page.

The `Style` concludes by setting the `AnchorX` value to 0, which sets the center of rotation to the vertical center of the left edge of each `Label`. Each `Label` then gets a unique `Rotation` setting:



Obviously, the spaces preceding the "ROTATE" string were chosen so that the vertical bars of the R combine to form a 16-sided polygon that seems almost like a circle.

You can also rotate individual letters in a text string if each letter is a separate `Label` element. You begin by positioning these `Label` elements in an `AbsoluteLayout` and then apply a `Rotation` property to make it appear as if the letters follow a particular nonlinear path. The **CircularText** program arranges these letters in a circle.

CircularText is a code-only program and is similar to the alternate **BoxViewCircle** algorithm. The constructor is responsible for creating all the individual `Label` elements and adding them to the `Children` collection of the `AbsoluteLayout`. No positioning or rotating is performed during the constructor because the program doesn't yet know how large these individual `Label` elements are, or how large the `AbsoluteLayout` is:

```
public class CircularTextPage : ContentPage
{
    AbsoluteLayout absoluteLayout;
    Label[] labels;

    public CircularTextPage()
    {
```

```
        // Create the AbsoluteLayout.
        absoluteLayout = new AbsoluteLayout();
        absoluteLayout.SizeChanged += (sender, args) =>
            {
                LayOutLabels();
            };
        Content = absoluteLayout;

        // Create the Labels.
        string text = "Xamarin.Forms makes me want to code more with ";
        labels = new Label[text.Length];
        double fontSize = 32;
        int countSized = 0;

        for (int index = 0; index < text.Length; index++)
        {
            char ch = text[index];

            Label label = new Label
            {
                Text = ch == ' ' ? "-" : ch.ToString(),
                Opacity = ch == ' ' ? 0 : 1,
                FontSize = fontSize,
            };
            label.SizeChanged += (sender, args) =>
                {
                    if (++countSized >= labels.Length)
                        LayOutLabels();
                };

            labels[index] = label;
            absoluteLayout.Children.Add(label);
        }
    }

    void LayOutLabels()
    {
        // Calculate the total width of the Labels.
        double totalWidth = 0;

        foreach (Label label in labels)
        {
            totalWidth += label.Width;
        }

        // From that, get a radius of the circle to center of Labels.
        double radius = totalWidth / 2 / Math.PI + labels[0].Height / 2;
        Point center = new Point(absoluteLayout.Width / 2, absoluteLayout.Height / 2);
        double angle = 0;

        for (int index = 0; index < labels.Length; index++)
        {
            Label label = labels[index];
```

```
            // Set the position of the Label.
            double x = center.X + radius * Math.Sin(angle) - label.Width / 2;
            double y = center.Y - radius * Math.Cos(angle) - label.Height / 2;

            AbsoluteLayout.SetLayoutBounds(label, new Rectangle(x, y, AbsoluteLayout.AutoSize,
                                                                AbsoluteLayout.AutoSize));
            // Set the rotation of the Label.
            label.Rotation = 360 * angle / 2 / Math.PI;

            // Increment the rotation angle.
            if (index < labels.Length - 1)
            {
                angle += 2 * Math.PI * (label.Width + labels[index + 1].Width) / 2 / totalWidth;
            }
        }
    }
}
```

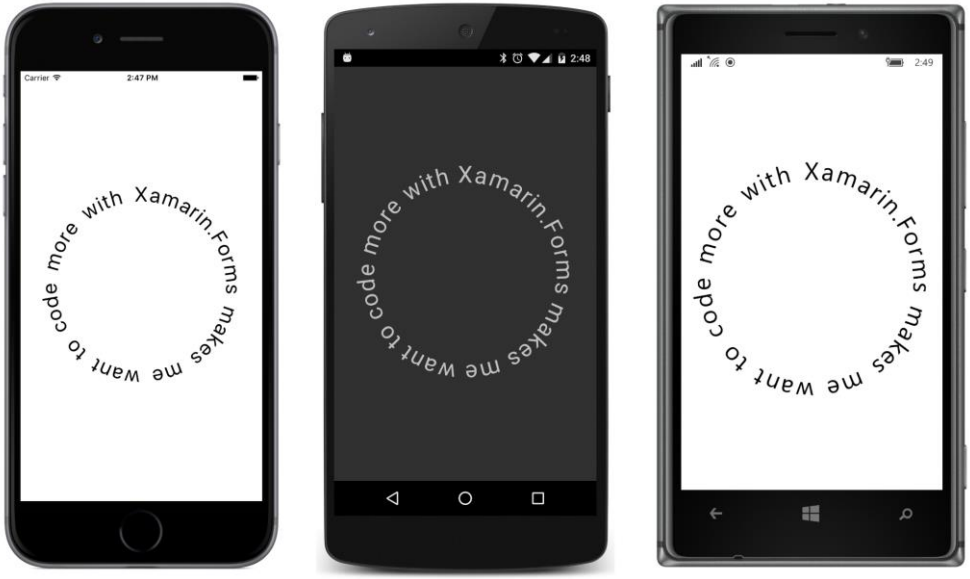
Notice the code that creates each `Label` element: If the character in the original text string is a space, the `Text` property of the `Label` is assigned a dash, but the `Opacity` property is set to 0 so that the dash is invisible. This is a little trick to fix a problem that showed up on the Windows Runtime platforms: If the `Label` contains only a space, then the width of the `Label` is calculated as zero and all the words run together.

All the action happens in the `LayOutLabels` method. This method is called from two `SizeChanged` handlers expressed as lambda functions in the constructor. The `SizeChanged` handler for the `AbsoluteLayout` is called soon after the program starts up or when the phone changes orientation. The `SizeChanged` handler for the `Label` elements keeps track of how many have been sized so far, and only calls `LayOutLabels` when they are all ready.

The `LayOutLabels` method calculates the total width of all the `Label` elements. If that's assumed to be the circumference of a circle, then the method can easily compute a radius of that circle. But that radius is actually extended by half the height of each `Label`. The endpoint of that radius thus coincides with the center of each `Label`. The `Label` is positioned within the `AbsoluteLayout` by subtracting half the `Label` width and height from that point.

An accumulated angle is used both for finding the endpoint of the radius for the next `Label` and for rotating the `Label`. Because the endpoint of each radius coincides with the center of each `Label`, the angle is incremented based on half the width of the current `Label` and half the width of the next `Label`.

Although the math is a bit tricky, the result is worth it:



This program does not set nondefault values of `AnchorX` and `AnchorY`, so there is no problem changing the phone orientation on iOS.

An analog clock

One of the classic sample programs for a graphical user interface is an analog clock. Once again, `BoxView` comes to the rescue for the hands of the clock. These `BoxView` elements must be rotated based on the hours, minutes, and seconds of the current time.

Let's first take care of the rotation mathematics with a class named `AnalogClockViewModel`, which is included in the **Xamarin.FormsBook.Toolkit** library:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class AnalogClockViewModel : ViewModelBase
    {
        double hourAngle, minuteAngle, secondAngle;

        public AnalogClockViewModel()
        {
            UpdateLoop();
        }

        async void UpdateLoop()
        {
            while (true)
            {
                DateTime dateTime = DateTime.Now;
                HourAngle = 30 * (dateTime.Hour % 12) + 0.5 * dateTime.Minute;
                MinuteAngle = 6 * dateTime.Minute + 0.1 * dateTime.Second;
```

```
                SecondAngle = 6 * dateTime.Second + 0.006 * dateTime.Millisecond;

                await Task.Delay(16);
            }
        }

        public double HourAngle
        {
            private set { SetProperty(ref hourAngle, value); }
            get { return hourAngle; }
        }

        public double MinuteAngle
        {
            private set { SetProperty(ref minuteAngle, value); }
            get { return minuteAngle; }
        }

        public double SecondAngle
        {
            private set { SetProperty(ref secondAngle, value); }
            get { return secondAngle; }
        }
    }
}
```

Each of the three properties is updated 60 times per second in a loop paced by a `Task.Delay` call. Of course, the hour hand rotation angle is based not only on the hour, but on a fractional part of that hour available from the `Minute` part of the `DateTime` value. Similarly, the angle of the minute hand is based on the `Minute` and `Second` properties, and the second hand is based on the `Second` and `Millisecond` properties.

These three properties of the ViewModel can be bound to the `Rotation` properties of the three hands of the analog clock.

As you know, some clocks have a smoothly gliding second hand, while the second hand of other clocks moves in discrete ticks. The `AnalogClockViewModel` class seems to impose a smooth second hand, but if you want discrete ticks, you can supply a value converter for that purpose:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class SecondTickConverter : IValueConverter
    {
        public object Convert(object value, Type targetType,
                              object parameter, CultureInfo culture)
        {
            return 6.0 * (int)((double)value / 6);
        }

        public object ConvertBack(object value, Type targetType,
                                  object parameter, CultureInfo culture)
        {
```

```
            return (double)value;
        }
    }
}
```

The name of this class and even the tiny code might be obscure if you didn't know what it was supposed to do: The `Convert` method converts an angle of type `double` ranging from 0 to 360 degrees with fractional parts into discrete angle values of 0, 6, 12, 18, 24, and so forth. These angles correspond to the discrete positions of the second hand.

The **MinimalBoxViewClock** program instantiates three `BoxView` elements in its XAML file and binds the `Rotation` properties to the three properties of `AnalogClockViewModel`:

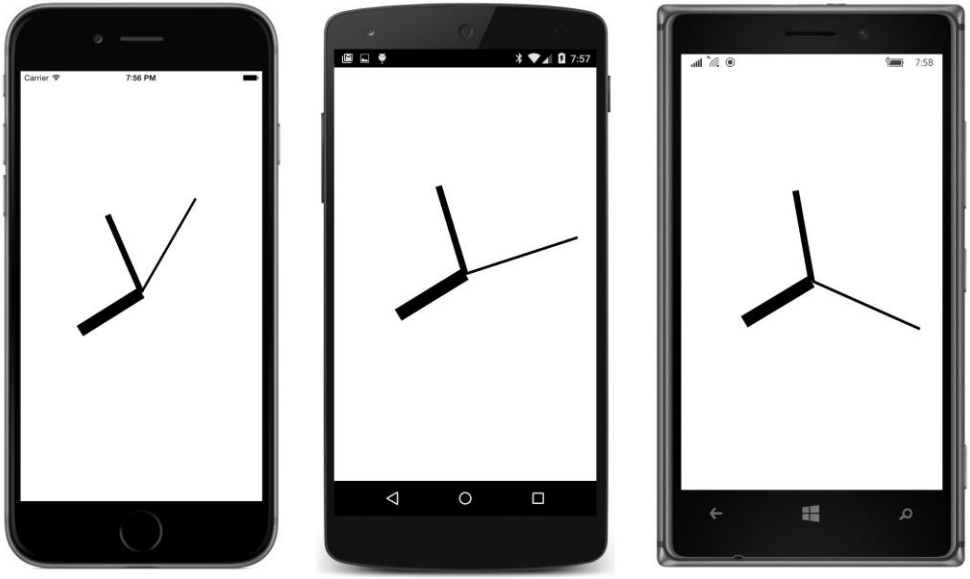
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:toolkit=
                 "clr-namespace:Xamarin.FormsBook.Toolkit;assembly=Xamarin.FormsBook.Toolkit"
             x:Class="MinimalBoxViewClock.MinimalBoxViewClockPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <ContentPage.Resources>
        <ResourceDictionary>
            <toolkit:SecondTickConverter x:Key="secondTick" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <AbsoluteLayout BackgroundColor="White"
                    SizeChanged="OnAbsoluteLayoutSizeChanged">

        <AbsoluteLayout.BindingContext>
            <toolkit:AnalogClockViewModel  />
        </AbsoluteLayout.BindingContext>

        <BoxView x:Name="hourHand"
                 Color="Black"
                 Rotation="{Binding HourAngle}" />

        <BoxView x:Name="minuteHand"
                 Color="Black"
                 Rotation="{Binding MinuteAngle}" />

        <BoxView x:Name="secondHand"
                 Color="Black"
                 Rotation="{Binding SecondAngle, Converter={StaticResource secondTick}}" />
    </AbsoluteLayout>
</ContentPage>
```

The code-behind file sets the sizes of these `BoxView` clock hands based on the size of the `AbsoluteLayout`, and it sets the locations so that all hands point up from the center of the clock in the

12:00 position:

```
public partial class MinimalBoxViewClockPage : ContentPage
{
    public MinimalBoxViewClockPage()
    {
        InitializeComponent();
    }

    void OnAbsoluteLayoutSizeChanged(object sender, EventArgs args)
    {
        AbsoluteLayout absoluteLayout = (AbsoluteLayout)sender;

        // Calculate a center and radius for the clock.
        Point center = new Point(absoluteLayout.Width / 2, absoluteLayout.Height / 2);
        double radius = Math.Min(absoluteLayout.Width, absoluteLayout.Height) / 2;

        // Position all hands pointing up from center.
        AbsoluteLayout.SetLayoutBounds(hourHand,
            new Rectangle(center.X - radius * 0.05,
                          center.Y - radius * 0.6,
                          radius * 0.10, radius * 0.6));

        AbsoluteLayout.SetLayoutBounds(minuteHand,
            new Rectangle(center.X - radius * 0.025,
                          center.Y - radius * 0.7,
                          radius * 0.05, radius * 0.7));

        AbsoluteLayout.SetLayoutBounds(secondHand,
            new Rectangle(center.X - radius * 0.01,
                          center.Y - radius * 0.9,
                          radius * 0.02, radius * 0.9));

        // Set the anchor to bottom center of BoxView.
        hourHand.AnchorY = 1;
        minuteHand.AnchorY = 1;
        secondHand.AnchorY = 1;
    }
}
```

For example, the hour hand is given a length of 0.60 of the clock's radius and a width of 0.10 of the clock's radius. This means that the horizontal position of the hour hand's top-left corner must be set to half its width (0.05 times the radius) to the left of the clock's center. The vertical position of the hour hand is the hand's height above the clock's center. The settings of `AnchorY` ensure that all rotations are relative to the center bottom of each clock hand:



Of course, this program is called **MinimalBoxViewClock** for a reason. It doesn't have convenient tick marks around the circumference, so it's a little hard to discern the actual time. Also, the clock hands should more properly overlap the center of the clock face so that they at least seem to be attached to a rotating pin or tube.

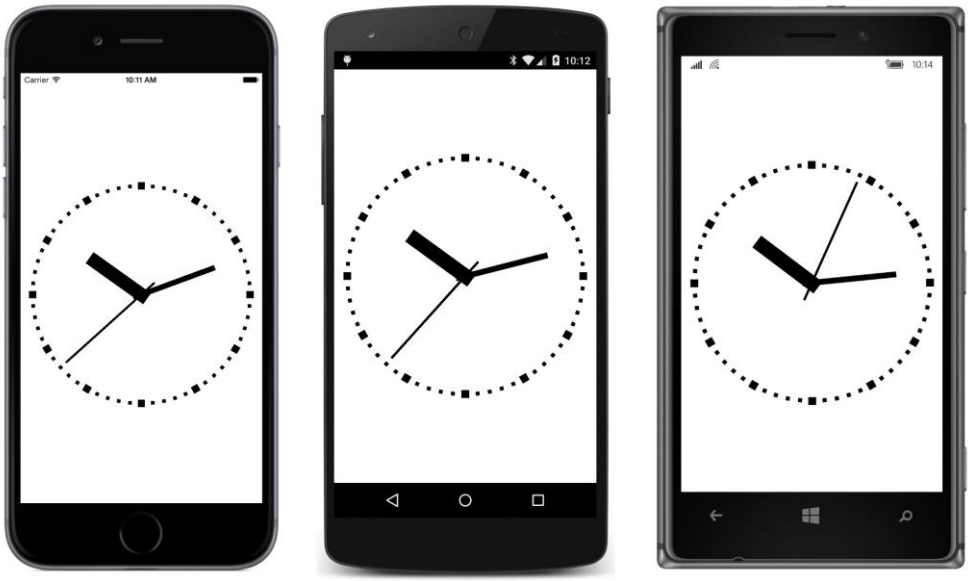
Both these problems are addressed in the nonminimal **BoxViewClock**. The XAML file is very similar to **MinimalBoxViewClock**, but the code-behind file is more extensive. It begins with a small structure named `HandParams`, which defines the size of each hand relative to the radius but also includes an `Offset` value. This is a fraction of the total length of the hand, indicating where it aligns with the center of the clock face. It also becomes the `AnchorY` value for rotations:

```
public partial class BoxViewClockPage : ContentPage
{
    // Structure for storing information about the three hands.
    struct HandParams
    {
        public HandParams(double width, double height, double offset) : this()
        {
            Width = width;
            Height = height;
            Offset = offset;
        }

        public double Width { private set; get; }   // fraction of radius
        public double Height { private set; get; }  // ditto
        public double Offset { private set; get; }  // relative to center pivot
    }

    static readonly HandParams secondParams = new HandParams(0.02, 1.1, 0.85);
```

```
    static readonly HandParams minuteParams = new HandParams(0.05, 0.8, 0.9);
    static readonly HandParams hourParams = new HandParams(0.125, 0.65, 0.9);

    BoxView[] tickMarks = new BoxView[60];

    public BoxViewClockPage()
    {
        InitializeComponent();

        // Create the tick marks (to be sized and positioned later).
        for (int i = 0; i < tickMarks.Length; i++)
        {
            tickMarks[i] = new BoxView { Color = Color.Black };
            absoluteLayout.Children.Add(tickMarks[i]);
        }
    }

    void OnAbsoluteLayoutSizeChanged(object sender, EventArgs args)
    {
        // Get the center and radius of the AbsoluteLayout.
        Point center = new Point(absoluteLayout.Width / 2, absoluteLayout.Height / 2);
        double radius = 0.45 * Math.Min(absoluteLayout.Width, absoluteLayout.Height);

        // Position, size, and rotate the 60 tick marks.
        for (int index = 0; index < tickMarks.Length; index++)
        {
            double size = radius / (index % 5 == 0 ? 15 : 30);
            double radians = index * 2 * Math.PI / tickMarks.Length;
            double x = center.X + radius * Math.Sin(radians) - size / 2;
            double y = center.Y - radius * Math.Cos(radians) - size / 2;
            AbsoluteLayout.SetLayoutBounds(tickMarks[index], new Rectangle(x, y, size, size));
            tickMarks[index].Rotation = 180 * radians / Math.PI;
        }

        // Position and size the three hands.
        LayoutHand(secondHand, secondParams, center, radius);
        LayoutHand(minuteHand, minuteParams, center, radius);
        LayoutHand(hourHand, hourParams, center, radius);
    }

    void LayoutHand(BoxView boxView, HandParams handParams, Point center, double radius)
    {
        double width = handParams.Width * radius;
        double height = handParams.Height * radius;
        double offset = handParams.Offset;

        AbsoluteLayout.SetLayoutBounds(boxView,
            new Rectangle(center.X - 0.5 * width,
                          center.Y - offset * height,
                          width, height));

        // Set the AnchorY property for rotations.
        boxView.AnchorY = handParams.Offset;
    }
```

```
}
```

The tick marks around the circumference of the clock face are also `BoxView` elements, but there are 60 of them with two different sizes, and they are positioned using techniques you've already seen. The visuals are surprisingly good considering the absence of a Xamarin.Forms graphics system:



Best of all, you can actually tell the time.

This clock has another interesting feature that makes the movement of the hands quite mesmerizing. The second hand neither glides from second to second or makes discrete jumps; instead it has a more complex movement. It pulls back slightly, then jumps ahead but slightly overshooting its mark, and then backs up and comes to rest. How is this done?

In the next chapter, you'll see that Xamarin.Forms implements several *easing functions* that can add realism to an animation by changing the animation's velocity—by speeding it up and slowing it down—over the course of the animation. Such easing functions have become fairly standard throughout the computer industry, and **Xamarin.FormsBook.Toolkit** contains a value converter that implements an easing function called the *back ease*:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class SecondBackEaseConverter : IValueConverter
    {
        public object Convert(object value, Type targetType,
                              object parameter, CultureInfo culture)
        {
            int seconds = (int)((double)value / 6);     // 0, 1, 2, ... 60
            double t = (double)value / 6 % 1;           // 0 --> 1
            double v = 0;                               // 0 --> 1
```

```
            // Back-ease in and out functions from http://robertpenner.com/easing/
            if (t < 0.5)
            {
                t *= 2;
                v = 0.5 * t * t * ((1.7 + 1) * t - 1.7);
            }
            else
            {
                t = 2 * (t - 0.5);
                v = 0.5 * (1 + ((t - 1) * (t - 1) * ((1.7 + 1) * (t - 1) + 1.7) + 1));
            }

            return 6 * (seconds + v);
        }

        public object ConvertBack(object value, Type targetType,
                                  object parameter, CultureInfo culture)
        {
            return (double)value;
        }
    }
}
```

This converter is referenced in the **BoxViewClock** XAML file:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:toolkit=
                 "clr-namespace:Xamarin.FormsBook.Toolkit;assembly=Xamarin.FormsBook.Toolkit"
             x:Class="BoxViewClock.BoxViewClockPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <ContentPage.Resources>
        <ResourceDictionary>
            <toolkit:SecondBackEaseConverter x:Key="secondBackEase" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <AbsoluteLayout x:Name="absoluteLayout"
                    BackgroundColor="White"
                    SizeChanged="OnAbsoluteLayoutSizeChanged">

        <AbsoluteLayout.BindingContext>
            <toolkit:AnalogClockViewModel />
        </AbsoluteLayout.BindingContext>

        <BoxView x:Name="hourHand"
                 Color="Black"
                 Rotation="{Binding HourAngle}" />
```

```
        <BoxView x:Name="minuteHand"
                 Color="Black"
                 Rotation="{Binding MinuteAngle}" />

        <BoxView x:Name="secondHand"
                 Color="Black"
                 Rotation="{Binding SecondAngle, Converter={StaticResource secondBackEase}}" />
    </AbsoluteLayout>
</ContentPage>
```

You'll see more easing functions in the next chapter.

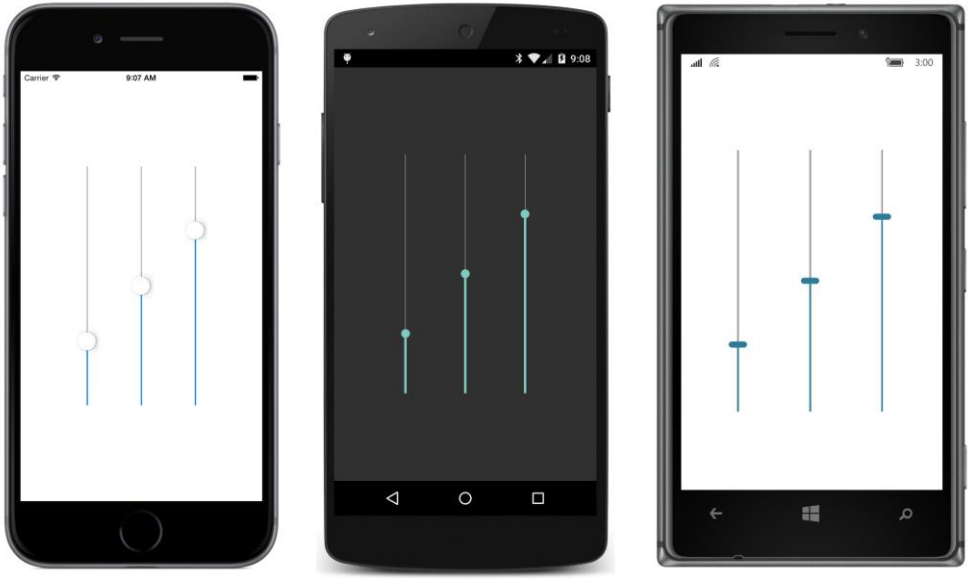
Vertical sliders?

Can certain views be rotated and still work as they should? More specifically, can the normal horizontal `Slider` elements of Xamarin.Forms be rotated to become vertical sliders?

Let's try it. The **VerticalSliders** program contains three sliders in a `StackLayout`, and the `StackLayout` itself is rotated 90 degrees counterclockwise:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="VerticalSliders.VerticalSlidersPage">

    <StackLayout VerticalOptions="Center"
                 Spacing="50"
                 Rotation="-90">

        <Slider Value="0.25" />

        <Slider Value="0.5" />

        <Slider Value="0.75" />
    </StackLayout>
 </ContentPage>
```

Sure enough, all three sliders are now oriented vertically:



And they work! You can manipulate these vertical sliders just as though they had been designed for that purpose. The `Minimum` value corresponds to a thumb position at the bottom, and the `Maximum` value corresponds to the top.

However, the Xamarin.Forms layout system is completely unaware of the new locations of these sliders. For example, if you turn the phone to landscape mode, the sliders are resized for the width of the portrait screen and are much too large to be rotated into a vertical position. You'll need to spend some extra effort in getting rotated sliders positioned and sized intelligently.

But it does work.

3D-ish rotations

Even though computer screens are flat and two-dimensional, it's possible to draw visual objects on these screens that give the appearance of a third dimension. Earlier in this chapter you saw some text effects that give the hint of a third dimension, and Xamarin.Forms supports two additional rotations, named `RotationX` and `RotationY`, that also seem to break through the inherent two-dimensional flatness of the screen.

When dealing with 3D graphics, it's convenient to think of the screen as part of a 3D coordinate system. The X axis is horizontal and the Y axis is vertical, as usual. But there is also an implicit Z axis that is orthogonal to the screen. This Z axis sticks out from the screen and extends through the back of the screen.

In 2D space, rotation occurs around a point. In 3D space, rotation occurs around an axis. The `RotationX` property is rotation around the X axis. The top and bottom of a visual object seem to move toward the viewer or away from the viewer. Similarly, `RotationY` is rotation around the Y axis. The left and right sides of a visual object seem to move toward the viewer or away from the viewer. By extension, the basic `Rotation` property is rotation around the Z axis. For consistency, the `Rotation` property should probably be named `RotationZ`, but that might confuse people who are thinking only in two dimensions.

The **ThreeDeeRotationDemo** program allows you to experiment with combinations of `RotationX`, `RotationY`, and `Rotation`, as well as explore how the `AnchorX` and `AnchorY` affect these two additional rotation properties:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ThreeDeeRotationDemo.ThreeDeeRotationDemoPage">
    <StackLayout Padding="20, 10">
        <Frame x:Name="frame"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand"
               OutlineColor="Accent">

            <Label Text="TEXT"
                   FontSize="72" />
        </Frame>

        <Slider x:Name="rotationXSlider"
                Maximum="360"
                Value="{Binding Source={x:Reference frame},
                                Path=RotationX}" />

        <Label Text="{Binding Source={x:Reference rotationXSlider},
                              Path=Value,
                              StringFormat='RotationX = {0:F0}'}"
               HorizontalTextAlignment="Center" />

        <Slider x:Name="rotationYSlider"
                Maximum="360"
                Value="{Binding Source={x:Reference frame},
                                Path=RotationY}" />

        <Label Text="{Binding Source={x:Reference rotationYSlider},
                              Path=Value,
                              StringFormat='RotationY = {0:F0}'}"
               HorizontalTextAlignment="Center" />

        <Slider x:Name="rotationZSlider"
                Maximum="360"
                Value="{Binding Source={x:Reference frame},
                                Path=Rotation}" />

        <Label Text="{Binding Source={x:Reference rotationZSlider},
                              Path=Value,
```

```
                                StringFormat='Rotation(Z) = {0:F0}'}"
                    HorizontalTextAlignment="Center" />

            <StackLayout Orientation="Horizontal"
                         HorizontalOptions="Center">
                <Stepper x:Name="anchorXStepper"
                         Minimum="-1"
                         Maximum="2"
                         Increment="0.25"
                         Value="{Binding Source={x:Reference frame},
                                         Path=AnchorX}" />

                <Label Text="{Binding Source={x:Reference anchorXStepper},
                                      Path=Value,
                                      StringFormat='AnchorX = {0:F2}'}"
                       VerticalOptions="Center"/>
            </StackLayout>

            <StackLayout Orientation="Horizontal"
                         HorizontalOptions="Center">
                <Stepper x:Name="anchorYStepper"
                         Minimum="-1"
                         Maximum="2"
                         Increment="0.25"
                         Value="{Binding Source={x:Reference frame},
                                         Path=AnchorY}" />

                <Label Text="{Binding Source={x:Reference anchorYStepper},
                                      Path=Value,
                                      StringFormat='AnchorY = {0:F2}'}"
                       VerticalOptions="Center"/>
            </StackLayout>
        </StackLayout>
    </ContentPage>
```

Here's a sample screen showing combinations of all three rotations:

You'll discover that the `AnchorY` property affects `RotationX` but not `RotationY`. For the default `AnchorY` value of 0.5, `RotationX` causes rotation to occur around the horizontal center of the visual object. When you set `AnchorY` to 0, rotation is around the top of the object, and for a value of 1, rotation is around the bottom.

Similarly, the `AnchorX` property affects `RotationY` but not `RotationX`. An `AnchorX` value of 0 causes `RotationY` to rotate the visual object around its left edge, while a value of 1 causes rotation around the right edge.

The directions of rotation are consistent among the three platforms, but they are best described in connection with conventions of 3D coordinate systems:

You might think there are many ways to arrange orthogonal X, Y, and Z axes. For example, increasing values of X might increase corresponding to leftward or rightward movement on the X axis, and increasing values of Y might correspond with up or down movement on the Y axis. However, many of these variations become equivalent when the axes are viewed from different directions. In reality, there are only two different ways to arrange X, Y, and Z axes. These two ways are known as *right-hand* and *left-hand* coordinate systems.

The 3D coordinate system implied by the three `Rotation` properties in Xamarin.Forms is left-handed: If you point the forefinger of your left hand in the direction of increasing X coordinates (which is to the right), and your middle finger in the direction of increasing Y coordinates (which is down), then your thumb points in the direction of increasing Z coordinates, which are coming out of the screen.

Your left hand can also be used to predict the direction of rotation: For rotation around a particular

axis, first point your thumb in the direction of increasing values on that axis. For rotation around the X axis, point your left thumb right. For rotation around the Y axis, point your left thumb down. For rotation around the Z axis, point your left thumb coming out of the screen. The curl of the other fingers of your left hand indicates the direction of rotation for positive angles.

In summary:

- For increasing angles of `RotationX`, the top goes back and the bottom comes out.
- For increasing angles of `RotationY`, the right side goes back and the left side comes out.
- For increasing angles of `Rotation`, the rotation is clockwise.

Aside from these conventions, `RotationX` and `RotationY` do not exhibit much visual consistency among the three platforms. Although all three platforms implement perspective—that is, the part of the object seemingly closest to the view is larger than the part of the object farther away—the amount of perspective you'll see is platform specific. There is no `AnchorZ` property that might allow fine-tuning these visuals.

But what's perhaps most obvious is that these various `Rotation` properties would be very fun to animate.