PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

UT3. Utilización de librerías multimedia integradas

Multimedia

Departamento de Informática y Comunicaciones

CFGS Desarrollo de Aplicaciones Multiplataforma

Segundo Curso

IES Virrey Morcillo

Villarrobledo (Albacete)

Índice

1. Imágenes

Una imagen permite describir la figura, representación, semejanza, aspecto o apariencia de un determinado objeto real o imaginario. Las imágenes son una parte fundamental de la navegación de las aplicaciones, la facilidad de uso y la personalización de la marca corporativa. También pueden ser necesarias para representar iconos y pantallas.

Las aplicaciones de deben ser capaces de compartir y mostrar imágenes en todas las plataformas. Por ello se pueden compartir entre las distintas plataformas utilizando Xamarin.Forms, permitiendo que se pueden cargar o descargar para cada plataforma específica para su presentación.

Xamarin.Forms usa la vista o control **Image** para mostrar imágenes en una página. Tiene dos propiedades importantes:

- **Source** – determina la imagen a mostrar, se trata de una instancia de **ImageSource** que puede ser un archivo, una URI, un recurso o un flujo.
- **Aspect** – determina el tamaño de la imagen dentro de los límites establecidos.

La propiedad **Source** determina el origen de la imagen por medio de instancias de **ImageSource** que pueden obtenerse mediante siguientes métodos estáticos:

- **FromFile** -Requiere un nombre del archivo o ruta del archivo que se puede resolver en cada plataforma.
- **FromUri** -Requiere un objeto de tipo URI o Identificador de Recursos Uniforme, por ejemplo: http://server.com/image.jpg.
- **FromResource** -Requiere un identificador de recurso a un archivo de imagen incrustado en la aplicación o en la biblioteca .NET Standard con un Build Action:EmbeddedResource.
- **FromStream** -Requiere un flujo que proporciona los datos de imagen.

La propiedad **Aspect** determina el modo de ajuste de la imagen al área de presentación establecida:

- **Fill** -Ajusta la imagen para rellenar completa y exactamente el área de presentación. Esto puede dar lugar a que la imagen se distorsione.
- **AspectFill** -Recorta la imagen para que rellene el área de presentación conservando el aspecto, es decir, sin ninguna distorsión.
- **AspectFit** - Preserva la relación de aspecto de la imagen, si es necesario, para que quepa completa en el área de presentación, agregando espacio en blanco a la parte superior, inferior o lados dependiendo de la imagen.

1.1. Imágenes locales

Los archivos de imagen se pueden agregar a cada proyecto de la aplicación y se pueden referenciar desde el código compartido de Xamarin.Forms. Este método de distribución de imágenes es preciso cuando las imágenes son específicas para cada plataforma, como, por ejemplo, al usar distintas resoluciones o distintos diseños.

Para usar una imagen única en todas las aplicaciones, se debe usar el mismo nombre de archivo en todas las plataformas, y debe ser un nombre de recurso válido de Android, es decir, se permiten sólo letras minúsculas, números, el carácter de subrayado y el punto).

En **Android** los archivos con las imágenes se deben almacenar en el directorio Resouces/drawable.

En **iOS** la forma preferida de administrar y admitir imágenes es usar conjuntos de imágenes del catálogo de activos, que debe contener todas las versiones de una imagen que son necesarias para soportar varios dispositivos y factores de escalado para una aplicación. En lugar de confiar en el nombre del archivo de la imagen, los conjuntos de imágenes usan un archivo Json para especificar qué imagen pertenece a qué dispositivo y/o resolución.

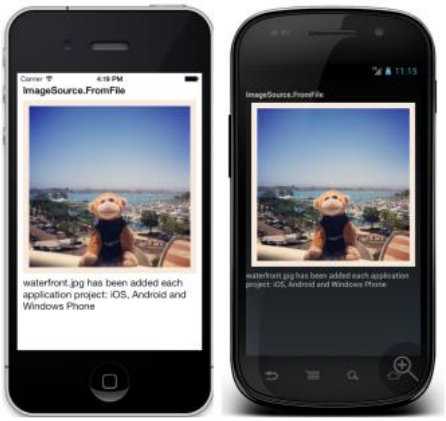
Para crear un nuevo conjunto de imágenes y agregar imágenes a él, hay que abrir el catálogo de activos desde el Explorador de soluciones y en la esquina superior izquierda, hacer clic en el botón Añadir, seleccionar Agregar conjunto de imagen y el editor establece la imagen que se mostrará para el nuevo conjunto de imágenes, desde aquí, arrastra las imágenes para cada uno de los distintos dispositivos y las resoluciones necesarias. Antes de iOS 9, las imágenes normalmente se colocaron en la carpeta Resources.

En **UWP** las imágenes se deben colocar en el directorio raíz de la aplicación.

Un ejemplo de imagen local podría ser el siguiente:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="WorkingWithImages.LocalImagesXaml">
  <StackLayout
      Margin="20">
    <Label
        Text="Image FileSource XAML"
        FontAttributes="Bold"
        HorizontalOptions="Center" />
    <Image
        Source="waterfront.jpg" />
    <Label
        Text="The image is referenced in XAML. On iOS and Android multiple
            resolutions are supplied and resolved at runtime." />
    </StackLayout>

</ContentPage>
```

Las capturas de pantalla siguientes muestran el resultado de mostrar una imagen local en cada plataforma:



1.2. Imágenes incrustadas

También se incluyen imágenes incrustadas con una aplicación, pero en lugar de tener una copia de la imagen en la estructura de archivos de la aplicación el archivo con la imagen está incrustado en el

ensamblado como un recurso. Este método para distribuir las imágenes se recomienda cuando se usan imágenes idénticas para cada plataforma y es especialmente adecuado para la creación de componentes, como por ejemplo cuando la imagen se empaqueta con el código.

Para incrustar una imagen en un proyecto, hay que copiar el archivo con la imagen dentro de la carpeta raíz del proyecto, incluso dentro de una carpeta dentro del proyecto, y agregar un elemento existente al mismo.

Un ejemplo de imagen incrustada podría ser el siguiente:

```
<?xml version="1.0" encoding="UTF-8" ?>
<ContentPage
    xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:WorkingWithImages;assembly=WorkingWithImages"
    x:Class="WorkingWithImages.EmbeddedImagesXaml">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="0, 20, 0, 0" />
        </OnPlatform>
    </ContentPage.Padding>

    <StackLayout
        VerticalOptions="Center"
        HorizontalOptions="Center">
        <Label
            Text="Image Resource Xaml" />
        <!-- uses a custom Extension defined in this project for now -->
        <Image
            Source="{local:ImageResource WorkingWithImages.beach.jpg}" />
        <Label
            Text="WorkingWithImages.beach.jpg embedded resource" />
    </StackLayout>

</ContentPage>
```

1.3. Descarga de imágenes

Las imágenes se pueden descargar automáticamente para su presentación, como se muestra en el XAML siguiente:

```
<?xml version="1.0" encoding="UTF-8" ?>
<ContentPage
    xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="WorkingWithImages.DownloadImagesXaml">

   <ContentPage.Padding>
          <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="0, 20, 0, 0" />
          </OnPlatform>
   </ContentPage.Padding>

  <StackLayout
      VerticalOptions="Center"
      HorizontalOptions="Center">

    <Label
        Text="Image UriSource Xaml" />

    <Image Source="http://xamarin.com/content/images/pages/forms/example-app.png" />
```

```
        <!-- Alternatively you can use attached property to manually alter the
             cache time -->
        <Image
            IsVisible="false">
            <Image.Source>
                <UriImageSource
                    Uri="http://xamarin.com/content/images/pages/forms/example-app.png"
                    CacheValidity="10:00:00.0"
                />
            </Image.Source>
        </Image>

        <Label
            Text="example-app.png gets downloaded from xamarin.com" />
    </StackLayout>
</ContentPage>
```

1.4. Iconos y pantallas de presentación

Aunque no se refiere a una vista o control **Image**, los iconos de aplicación y las pantallas de presentación hacen también un uso importante de imágenes en proyectos de Xamarin.Forms.

Establecer iconos y pantallas de presentación en las aplicaciones de Xamarin.Forms se realiza en cada uno de los proyectos específicos de aplicación. Esto significa generar correctamente un tamaño de imágenes para iOS, Android y UWP. Estas imágenes deben ser llamadas y localizadas según los requisitos de cada una de las plataformas.

2. Transformaciones

Con la ayuda de StackLayout y Grid, Xamarin.Forms hace un buen trabajo de redimensionado y posicionamiento de elementos visuales en la página. A veces, sin embargo, es necesario o conveniente que la aplicación realice algunos ajustes. Es posible que se desee desplazar un poco la posición de los elementos, cambiar su tamaño o incluso rotarlos.

Dichos cambios en la ubicación, el tamaño o la orientación son posibles usando una característica de Xamarin.Forms conocida como **transformaciones**. El concepto de la transformada se originó en la geometría. La transformada es una fórmula que mapea puntos a otros puntos. Por ejemplo, si se desea cambiar un objeto geométrico en un sistema de coordenadas cartesianas, se puede agregar factores de desplazamiento constantes a todas las coordenadas que definen ese objeto.

Estas transformaciones matemáticas y geométricas desempeñan un papel vital en la programación de gráficos por ordenador, donde a veces se conocen como transformaciones matriciales porque son más fáciles de expresar matemáticamente utilizando el álgebra matricial. Sin transformaciones, no puede haber gráficos 3D. Pero a lo largo de los años, las formas de transformación han migrado de la programación de gráficos a la programación de interfaz de usuario. Todas las plataformas compatibles con Xamarin.Forms admiten transformaciones básicas que se pueden aplicar a elementos de la interfaz de usuario como texto, mapas de bits y botones.

Xamarin.Forms admite tres tipos básicos de transformaciones:

- **Traslación**: desplazamiento de un elemento horizontal o verticalmente o ambos.
- **Escalado**: cambiar el tamaño de un elemento.
- **Rotación**: girar un elemento alrededor de un punto o eje.

El escalado soportado por Xamarin.Forms es uniforme en todas las direcciones, técnicamente conocido como escalado isotrópico. No puede usar la escala para cambiar la relación de aspecto de un elemento visual.

La rotación es compatible con la superficie bidimensional de la pantalla y en el espacio 3D. Xamarin.Forms no admite una transformación sesgada o una transformación matricial generalizada.

Xamarin.Forms admite estas transformaciones con ocho propiedades de la clase **VisualElement**. Estas propiedades son todas de tipo Double:

- TranslationX
- TranslationY
- Scale
- Rotation
- RotationX
- RotationY
- AnchorX
- AnchorY

2.1. Translación

Una aplicación utiliza una de las clases de diseño (StackLayout, Grid, AbsoluteLayout o Relative-Layout) para colocar un elemento visual en la pantalla. Llamemos a la posición establecida por el sistema de diseño la "posición de diseño".

Los valores distintos de cero de las propiedades TranslationX y TranslationY cambian la posición de un elemento visual en relación con esa posición de diseño. Los valores positivos de TranslationX desplazan el elemento a la derecha y los valores positivos de TranslationY desplazan el elemento hacia abajo.

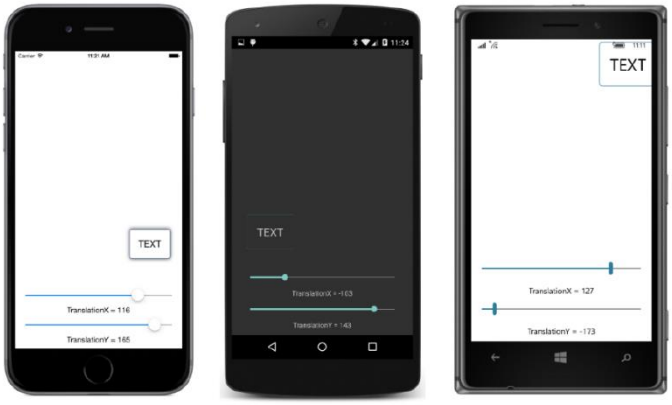
El programa **TranslationDemo** te permite experimentar con estas dos propiedades. Todo está en el archivo XAML:

```
<?xml version="1.0" encoding="utf-8" ?>
<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>
```

Un Frame encierra una Label y está centrado en la parte superior del StackLayout. Dos Slider tienen vinculadas a las propiedades TranslationX y TranslationY del Frame, y se inicializan para un rango de –200 a 200. Cuando ejecuta el programa por primera vez, los dos Slider se configuran con los valores predeterminados de TranslationX y TranslationY, que son cero.

Puedes manipular los controles deslizantes para mover el marco alrededor de la pantalla. Los valores de TranslationX y TranslationY especifican un desplazamiento del elemento en relación con su posición de diseño original:



Una traslación de un elemento como un Frame también afecta a todos los elementos hijos de ese elemento, que en este caso es solo la Label. Se pueden establecer las propiedades TranslationX y TranslationY en cualquier VisualElement, que incluye StackLayout, Grid e incluso Page y sus derivados. La traslación se aplica al elemento ya todos los elementos hijos de ese elemento.

Una aplicación común de TranslationX y TranslationY es aplicar pequeños desplazamientos a los elementos que los deslizan ligeramente desde su posición de diseño. Esto a veces es útil si se tiene varios elementos superpuestos en una cuadrícula de una sola celda y se necesita desplazar uno para que se vea debajo de otro.

Incluso puedes usar esta técnica para efectos de texto comunes. El programa **TextOffsets** sólo para XAML coloca tres pares de elementos Label en tres diseños de Grid de una sola celda. El par de elementos Label en cada Grid es del mismo tamaño y muestra el mismo texto:

```
<?xml version="1.0" encoding="utf-8" ?>
<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">
            <On Platform="iOS" Value="0, 20, 0, 0" />
        </OnPlatform>
```

```
        </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>
```

Normalmente, la primera Label en la colección de Childen de la Grid quedaría oculta por la segunda Label, pero los valores de TranslationX y TranslationY aplicados en la primera Label permiten que sea parcialmente visible. La misma técnica básica da como resultado tres efectos de texto diferentes: una sombra paralela, el texto que parece levantarse de la superficie de la pantalla y el texto que parece cincelado en la pantalla:

El programa **ButtonJump** demuestra que no importa a dónde mueva un botón en la pantalla utilizando la traducción, el Botón todavía responde a los toques de la manera normal. El archivo XAML centra el botón en el centro de la página (menos el relleno de iOS en la parte superior):

```
<?xml version="1.0" encoding="utf-8" ?>
<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">
            <On Platform="iOS" Value="0, 20, 0, 0" />
        </OnPlatform>
    </ContentPage.Padding>

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

</ContentPage>
```

Para cada llamada al controlador OnButtonClicked, el archivo de código subyacente establece las propiedades TranslationX y TranslationY en nuevos valores. Los nuevos valores se calculan aleatoriamente, pero se restringen para que el botón siempre permanezca dentro de los bordes de la pantalla:

```
using System;
using Xamarin.Forms;

namespace ButtonJump {

   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);
```

```
            }
      }
}
```

Unos pocos segundos jugando con el programa ButtonJump probablemente suscita una pregunta: ¿Se puede anular esto? ¿Puede el botón deslizarse hacia el nuevo punto en lugar de simplemente saltar allí?.

Por supuesto. Hay varias formas de hacerlo. El archivo XAML en el programa **ButtonGlide** es el mismo que el de ButtonJump, excepto que el botón ahora tiene un nombre para que el programa pueda hacer referencia fácilmente fuera del controlador de clic:

```
<?xml version="1.0" encoding="utf-8" ?>
<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">
            <On Platform="iOS" Value="0, 20, 0, 0" />
        </OnPlatform>
    </ContentPage.Padding>

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

</ContentPage>
```

El archivo de código subyacente procesa el clic del botón guardando varios datos esenciales como campos: un Point que indica la ubicación de inicio obtenida a partir de los valores actuales de TranslationX y TranslationY, un vector que también es un valor de Point calculado al restar este punto de inicio de un punto de destino aleatorio y el DateTime actual cuando se hace clic en el Button:

```
using System;
using Xamarin.Forms;

namespace ButtonGlide {

   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;
            }
        }
    }
```

La devolución de llamada del temporizador se llama cada 16 milisegundos. Las pantallas de video comúnmente tienen una frecuencia de actualización de hardware de 60 veces por segundo. Por lo tanto, cada fotograma está activo durante unos 16 milisegundos. El ritmo de la animación a este ritmo es óptimo. Una vez cada 16 milisegundos, la devolución de llamada calcula el tiempo transcurrido desde el comienzo de la animación y lo divide por la duración. Ese es un valor llamado t para el tiempo que varía de 0 a 1 en el transcurso de la animación. Este valor se multiplica por el vector y el resultado se agrega a startPoint. Ese es el nuevo valor de la TraducciónX y la TraducciónY.

Aunque la devolución de llamada del temporizador se llama de forma continua mientras la aplicación se está ejecutando, las propiedades de TranslacationX y TranslationY permanecen constantes cuando la animación se ha completado. Sin embargo, no se tiene que esperar hasta que el botón deje de moverse antes de poder tocarlo nuevamente. La nueva animación comienza desde la posición actual del Button y reemplaza por completo a la animación anterior.

2.2. Escala

La clase VisualElement define una propiedad llamada Scale que se puede usar para cambiar el tamaño representado de un elemento. La propiedad Scale no afecta al diseño, no afecta a las propiedades Width y Height del elemento, ni a la propiedad Bounds que incorpora esos valores de Width y Height. Los cambios en la propiedad Scale no hacen que se dispare un evento SizeChanged.

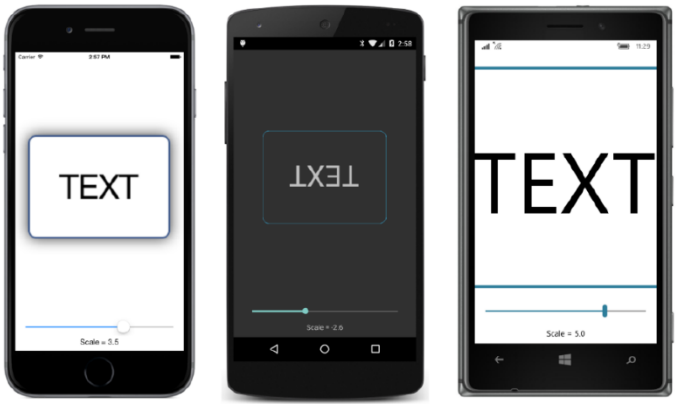
La propiedad Scale afecta a las coordenadas de un elemento visual renderizado, pero de una manera muy diferente a TranslationX y TranslationY. Las dos propiedades de traslación agregan valores a las coordenadas, mientras que la propiedad Scale es multiplicativa. El valor predeterminado de Scale es 1. Los valores superiores a 1 aumentan el tamaño del elemento. Por ejemplo, un valor de 3 hace que el elemento sea tres veces su tamaño normal. Los valores menores a 1 disminuyen el tamaño. Un valor de Scale de 0 es posible pero hace que el elemento sea invisible. Si se está trabajando con Scale y un elemento parece haber desaparecido, hay que verificar si está obteniendo un valor de Scale

de 0. Los valores inferiores a 0 también son legales y hacen que el elemento se gire 180 grados además de que se modifique su tamaño.

Se puede experimentar con la configuración de escala usando el programa **SimpleScaleDemo**. El archivo XAML es el siguiente:

```
<?xml version="1.0" encoding="utf-8" ?>
<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>
```

Aquí está en acción. Observa la configuración de escala negativa en el dispositivo Android:



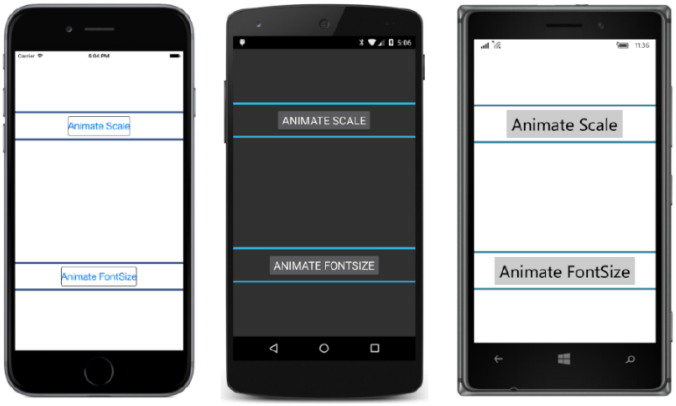
Es posible que se desee utilizar Scale para proporcionar una pequeña retroalimentación a un usuario cuando se hace clic en un botón. El botón puede expandirse brevemente en tamaño y volver a la normalidad nuevamente. Sin embargo, la escala no es la única forma de cambiar el tamaño de un botón. También puede cambiar el tamaño del botón aumentando y disminuyendo la propiedad FontSize. Sin embargo, estas dos técnicas son muy diferentes: la propiedad Scale no afecta al diseño, pero la propiedad FontSize sí lo hace.

Esta diferencia se ilustra en el programa **ButtonScaler**. El archivo XAML consta de dos elementos de Botón intercalados entre dos pares de elementos de BoxView:

```
<?xml version="1.0" encoding="utf-8" ?>
<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>
```

Esto es lo que la página se ve normalmente:



El archivo de código subyacente implementa un método de animación un tanto generalizado. Se generaliza en el sentido de que los parámetros incluyen dos valores que indican el valor de inicio y el valor final de la animación. Estos dos valores a menudo se denominan valor desde y valor hasta. Los argumentos de la animación también incluyen la duración de la animación y un método de devolución de llamada. El argumento del método de devolución de llamada es un valor entre el valor "desde" y el valor "hasta", y el método de llamada puede usar ese valor para hacer lo que sea necesario para implementar la animación.

Sin embargo, este método de animación no está del todo generalizado. En realidad, calcula un valor desde el valor desde hasta el valor hasta durante la primera mitad de la animación, y luego

calcula un valor desde el valor hasta al valor desde durante la segunda mitad de la animación. Esto se llama a menudo una animación de inversión.

El método se llama AnimateAndBack, usa una llamada Task.Delay para controlar la animación y un objeto Stopwach .NET para determinar el tiempo transcurrido:

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;
using Xamarin.Forms;

namespace ButtonScaler {

   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);
          }
   }
}
```

Los controladores Clicked para los dos botones inician una animación independiente. El controlador Clicked para el primer botón anima su propiedad de escala de 1 a 5 y viceversa, mientras que el controlador Clicked para el segundo botón anima su propiedad FontSize con un factor de escala de 1 a 5 y viceversa.

Otro uso de la propiedad Scale es dimensionar un elemento para que se ajuste al espacio disponible. Para ello se propone el programa **ScaleToSize**. Puede hacer algo muy similar con la propiedad Scale, pero no se tendrán que hacer estimaciones ni cálculos recursivos.

El archivo XAML del programa contiene una etiqueta que le falta texto y también que le falta una configuración de escala para agrandar la etiqueta:

```
<?xml version="1.0" encoding="utf-8" ?>
<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>
```

Tanto ContentPage como Label tienen instalados los controladores SizeChanged, y ambos usan el mismo controlador. Este controlador simplemente establece la propiedad Scale de la etiqueta al mínimo del ancho y alto de la página dividido por el ancho y alto de la etiqueta:

```
using System;
using System.Threading.Tasks;
using Xamarin.Forms;

namespace ScaleToSize {

   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);
          }
   }
}
```

Debido a que la configuración de la propiedad Scale no dispara otro evento SizeChanged, no hay peligro de desencadenar un bucle recursivo sin fin. Pero un bucle infinito real que usa Task.Delay mantiene la etiqueta actualizada con la hora actual.

2.3. Rotación

La propiedad Rotation gira un elemento visual en la superficie de la pantalla. El valor de la propiedad de Rotation debe ser un ángulo en grados (no radianes). Los ángulos positivos giran el elemento hacia la derecha. Se puede establecer la Rotation en ángulos menores que 0 o mayores que 360. El ángulo de rotación real es el valor de la propiedad Rotation modulo 360. El elemento gira alrededor de un punto relativo a sí mismo especificado con las propiedades AnchorX y AnchorY.

El programa **PlaneRotationDemo** permite experimentar con estas tres propiedades. El archivo XAML es el siguiente:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
```

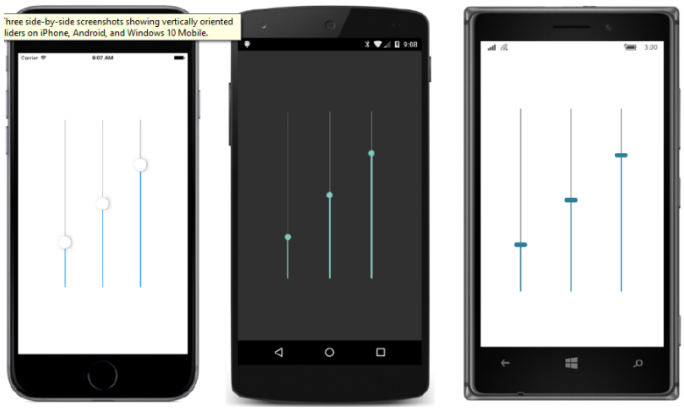
```
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="PlaneRotationDemo.PlaneRotationDemoPage">
    <StackLayout Padding="20, 10">
        <Frame x:Name="frame"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand"
               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>
```

Aquí hay varias combinaciones de ángulos de rotación y centros de rotación:

El programa **VerticalSliders** contiene tres controles deslizantes en un StackLayout, y el mismo StackLayout se gira 90 grados en sentido contrario a las agujas del reloj:

```
<?xml version="1.0" encoding="utf-8" ?>
<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"
                BackgroundColor="Gray"/>

        <Slider Value="0.5"
                BackgroundColor="Gray"/>

        <Slider Value="0.75"
                BackgroundColor="Gray"/>
    </StackLayout>

</ContentPage>
```

Efectivamente, los tres controles deslizantes ahora están orientados verticalmente.



Y funcionan. Se puede manipular estos controles deslizantes verticales como si hubieran sido diseñados para ese propósito. El valor mínimo corresponde a una posición del pulgar en la parte inferior, y el valor máximo corresponde a la parte superior.

Sin embargo, el sistema de diseño de Xamarin.Forms desconoce por completo las nuevas ubicaciones de estos controles deslizantes. Por ejemplo, si se pone el teléfono en modo horizontal, los controles deslizantes cambian de tamaño para el ancho de la pantalla vertical y son demasiado grandes para girarlos a una posición vertical. Se debe dedicar un poco de esfuerzo adicional a la colocación y el tamaño de los controles deslizantes girados de manera inteligente.

3. Animaciones

La animación por ordenador normalmente se refiere a cualquier tipo de cambio visual dinámico. Un botón que simplemente aparece en una página no es una animación. Pero un botón que se difumina en vista o se mueve de su lugar o crece en tamaño desde un punto, eso es animación. Muy a menudo, los elementos visuales responden a las entradas del usuario con un cambio en la apariencia, como un Button flash, un Stepper o un desplazamiento de ListView, eso también es animación.

A veces es deseable que una aplicación vaya más allá de las animaciones automáticas y convencionales y agregue la suya propia.

Xamarin.Forms incluye su propia infraestructura de animación que existe en tres niveles de interfaces de programación correspondientes a las clases **ViewExtensions**, **Animation** y **AnimationExtensions**. Este sistema de animación es lo suficientemente versátil para trabajos complejos, pero excepcionalmente fácil para trabajos simples.

Las clases de animación de Xamarin.Forms se usan generalmente para identificar propiedades de elementos visuales. Una animación típica cambia progresivamente el valor de una propiedad a otro durante un período de tiempo. Las propiedades a las que apuntan las animaciones deben estar respaldadas por propiedades vinculadas. Esto no es un requisito, pero las propiedades vinculadas generalmente están diseñadas para responder a cambios dinámicos a través de la implementación de un controlador de propiedades modificadas. No sirve de nada animar una propiedad de un objeto si el objeto ni siquiera se da cuenta de que se está cambiando la propiedad.

No hay una interfaz XAML para el sistema de animación Xamarin.Forms. En consecuencia, todas las animaciones se realizan en código C#. Sin embargo, puede encapsular animaciones tardías en clases mediante comportamientos (behaviors) y disparadores (triggers) y luego hacer referencia a ellas desde archivos XAML. Los disparadores y los comportamientos son generalmente la forma más fácil y la forma recomendada de incorporar animaciones.

3.1. Animaciones simples

Vamos a considerar con un pequeño programa llamado **AnimationTryout**. El archivo XAML no contiene más que un Button centrado:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="AnimationTryout.AnimationTryoutPage">

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

</ContentPage>
```

Para este ejemplo, ignoremos la función esencial real que el Button realiza supuestamente dentro de la aplicación. Además de querer que el botón realice esa función, supongamos que queremos girarlo en círculo cuando el usuario lo toque. El manejador Clicked en el archivo de código subyacente puede hacerlo llamando a un método llamado RotateTo con un argumento de 360 para la cantidad de grados que deben rotar:

```
using System;
using Xamarin.Forms;
```

```
namespace AnimationTryout {

   public partial class AnimationTryoutPage : ContentPage {

          public AnimationTryoutPage() {
                 InitializeComponent();
          }

          void OnButtonClicked(object sender, EventArgs args) {
                 button.RotateTo(360);
          }
   }
}
```

El método RotateTo es una animación que se dirige a la propiedad Rotation del botón. Sin embargo, el método RotateTo no está definido en la clase VisualElement como la propiedad Rotation. Es, en cambio, un método de extensión definido en la clase ViewExtensions.

Cuando ejecuta este programa y toca el botón, el método RotateTo anima el botón para que gire en un círculo completo de 360 grados. Aquí está en progreso:

El viaje completo toma 250 milisegundos (un cuarto de segundo), que es la duración predeterminada de esta animación RotateTo.

Sin embargo, este programa tiene un defecto. Después de haber visto girar el botón, intente volver a tocarlo. No gira: en la primera llamada a OnButtonClicked, el método RotateTo obtiene la propiedad de rotación actual, que es 0, y luego elimina la animación de la propiedad de rotación de ese valor a el argumento de RotateTo, que es 360. Cuando la animación concluye después de un cuarto de segundo, la propiedad Rotación se deja en 360. La próxima vez que se presiona el botón, el valor actual es 360 y el argumento de RotateTo también es 360. Internamente, la animación todavía ocurre, pero la propiedad Rotación no cede. Está atascado en 360.

Aquí hay una pequeña variación del controlador Clicked en AnimationTryout. No soluciona el problema con varios toques del botón, pero extiende la animación a dos segundos para que se pueda disfrutar de la animación por más tiempo. La duración se especifica en milisegundos como el segundo argumento de RotateTo. Ese segundo argumento es opcional y tiene un valor predeterminado de 250:

```
void OnButtonClicked(object sender, EventArgs args) {
       button.RotateTo(360, 2000);
}
```

Una solución al problema de los toques subsiguientes es usar RelRotateTo, que obtiene la propiedad de Rotation actual para el inicio de la animación y luego agrega su argumento a ese valor para el final de la animación. Aquí hay un ejemplo:

```
void OnButtonClicked(object sender, EventArgs args) {
       button.RelRotateTo(360, 1000);
```

```
}
```

Cada toque inicia una animación que gira el botón 360 grados en el transcurso de un segundo. Si toca el botón mientras una animación está en progreso, una nueva animación comienza desde esa posición, por lo que podría terminar en una posición que no sea un incremento de 360 grados. No hay cambios en la velocidad con múltiples toques porque la animación siempre va a una velocidad de 360 grados por segundo.

Otra forma de solucionar el problema con los toques posteriores es inicializar la propiedad Rotation antes de la llamada a RotateTo:

```
void OnButtonClicked(object sender, EventArgs args) {
	button.Rotation = 0;
	button.RotateTo(360, 1000);
}
```

Ahora se puede presionar el botón nuevamente una vez que se haya detenido y comenzará la animación desde el principio. Los toques repetidos mientras el botón está girando también se comportan de manera diferente, comienzan desde 0 grados.

3.2. Animaciones asíncronas

Curiosamente, esta ligera variación en el código no permite grifos posteriores:

```
void OnButtonClicked(object sender, EventArgs args) {
	button.RotateTo(360, 1000);
	button.Rotation = 0;
}
```

Esta versión se comporta igual que la versión con solo el método RotateTo. Parece como si establecer la propiedad de Rotation a 0 después de esa llamada no haga nada.

No funciona porque el método RotateTo es asíncrono. El método vuelve rápidamente después de iniciar la animación, pero la animación se produce en el fondo. La configuración de la propiedad Rotation en 0 en el momento en que el método RotateTo devuelve no tiene ningún efecto aparente porque la configuración RotateTo de fondo reemplaza muy rápidamente la configuración.

Debido a que el método es asíncrono, RotateTo devuelve un objeto Task, más específicamente, un objeto Task<bool>, y eso significa que puede llamar a ContinueWith para especificar una función de devolución de llamada que se invoca cuando finaliza la animación. Luego, la devolución de llamada puede establecer la propiedad Rotaton en 0 después de que la animación se haya completado:

```
void OnButtonClicked(object sender, EventArgs args) {
	button.RotateTo(360, 2000).ContinueWith((task) =>
		{
			button.Rotation = 0;
		});
}
```

El objeto Task pasado a ContinueWith es de tipo Task<bool>, y la devolución de llamada ContinueWith puede usar la propiedad Result para obtener ese valor booleano. El valor es True si la animación fue cancelada y False si se completó.

Es más probable que se utilice await con el método RotateTo que con ContinueWith:

```
async void OnButtonClicked(object sender, EventArgs args) {
	bool wasCancelled = await button.RotateTo(360, 2000);
	button.Rotation = 0;
}
```

O, si no importa el valor de retorno:

```
async void OnButtonClicked(object sender, EventArgs args) {
       await button.RotateTo(360, 2000);
       button.Rotation = 0;
}
```

Observa el modificador async en el controlador, que se requiere para cualquier método que contenga operadores await.

Usar await es particularmente útil para apilar animaciones secuenciales:

```
async void OnButtonClicked(object sender, EventArgs args) {
       await button.RotateTo(90, 250);
       await button.RotateTo(-90, 500);
       await button.RotateTo(0, 250);
}
```

El botón gira 90 grados en el sentido de las agujas del reloj en el primer cuarto de segundo, luego 180 grados en sentido contrario a las agujas del reloj en la siguiente media segundo, y luego 90 grados en el sentido de las agujas del reloj para terminar nuevamente en 0 grados. Debes esperar en los dos primeros para que sean secuenciales, pero no lo necesitas en el tercero si no hay nada más que ejecutar en el controlador de clic después de que se haya completado la tercera animación.

3.3. Animaciones compuestas

Se puede mezclar llamadas asíncronas y síncronas para crear animaciones compuestas. Por ejemplo, supongamos que se desea que el botón gire alrededor de 360 grados al mismo tiempo que se expanda y se contraiga:

```
async void OnButtonClicked(object sender, EventArgs args) {
       button.Rotation = 0;
       button.RotateTo(360, 2000);
       await button.ScaleTo(5, 1000);
       await button.ScaleTo(1, 1000);
}
```

3.4. Funciones de aceleración

Como puede apreciar en las animaciones anteriores, la animación no se ve todo lo bien que debería. El movimiento parece muy mecánico y robótico porque las rotaciones tienen una velocidad angular constante. Se puede controlar los cambios de velocidad en las animaciones con el uso de **funciones de aceleración** (easing functions).

Xamarin.Forms incluye la clase **Easing** que permite especificar una función de transferencia simple que controla cómo las animaciones se aceleran o deceleran a medida que se ejecutan.

Las animaciones generalmente involucran una variable llamada t o progreso que aumenta de 0 a 1 en el transcurso de la animación. Esta variable t se usa entonces en una interpolación entre los valores desde y hasta:

valor = valorDesde + t * (valorHasta − valorDesde)

La función de aceleración introduce una pequeña función de transferencia en este cálculo:

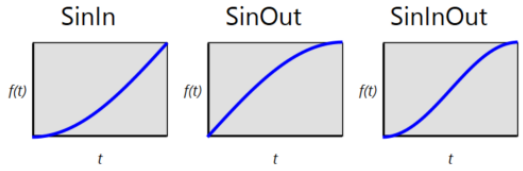
valor = valorDesde + FunciónAceleración(t) * (valorHasta − valorDesde)

La clase Easing define un método llamado Ease que realiza este trabajo. Para una entrada de 0, el método Ease devuelve 0, y para una entrada de 1, Ease devuelve 1. Entre esos dos valores, algunas matemáticas, le dan a la animación una velocidad no constante. Como se verá más adelante, no es del todo necesario que el método de Ease asigne 0 a 0 y 1 a 1, pero ese es ciertamente el caso normal.

Se pueden definir funciones de aceleración propias, pero la clase de Easing define once campos estáticos de sólo lectura de tipo Easing:

- Linear (el valor por defecto)
- SinIn, SinOut, and SinInOut
- CubicIn, CubicOut, and CubicInOut
- BounceIn and BounceOut
- SpringIn and SpringOut

Los sufijos in y out indican si el efecto es prominente al principio de la animación, al final o ambos. Las funciones de aceleración SinIn, SinOut y SinInOut se basan en las funciones seno y coseno:



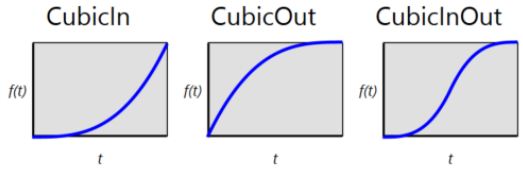
En cada uno de estos gráficos, el eje horizontal es el tiempo lineal, de izquierda a derecha de 0 a 1. El eje vertical muestra la salida del método Ease, de 0 a 1 de abajo hacia arriba. Una pendiente más pronunciada y vertical es más rápida, mientras que una pendiente más horizontal es más lenta.

Debido a que el movimiento armónico se describe mejor mediante curvas sinusoidales, estas funciones de aceleración son ideales para que un botón se mueva de un lado a otro. Se puede especificar un objeto de tipo Easing como último argumento para los métodos RotateTo:

```
async void OnButtonClicked(object sender, EventArgs args) {
       await button.RotateTo(90, 250, Easing.SinOut);
       await button.RotateTo(-90, 500, Easing.SinInOut);
       await button.RotateTo(0, 250, Easing.SinIn);
}
```

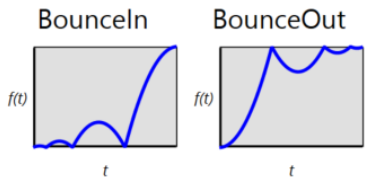
Ahora el movimiento es mucho más natural. El botón se ralentiza a medida que se acerca al punto en que invierte el movimiento y luego se acelera nuevamente.

La función de aceleración de CubicIn es simplemente la entrada elevada a la tercera potencia. El CubicOut es el reverso de eso, y CubicInOut combina los dos efectos:



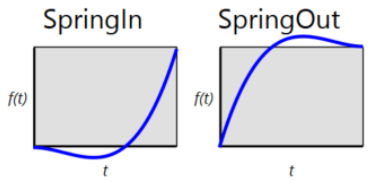
La diferencia en la velocidad es más acentuada que el alivio del seno.

Los rebotes BounceIn y BounceOut al principio o al final, respectivamente:



Como se puede suponer, el BounceOut es ideal para animar transformaciones que parecen encontrarse con un obstáculo.

La salida de las funciones SpringIn y SpringOut en realidad va más allá del rango de 0 a 1. La SpringIn tiene una salida que cae por debajo de 0 inicialmente, y la salida de SpringOut va más allá del valor de 1:



En otros sistemas de animación, estos patrones SpringIn y SpringOut generalmente se conocen como funciones de retroceso. De hecho, puede volver a escribir el método Convert en SecondBackEaseConverter de esta manera y funcionará de la misma manera.

El programa **BounceButton** tiene un archivo XAML que es lo mismo que AnimationTryout, y el controlador de clic para el botón tiene solo tres declaraciones:

```
using System;
using System.Threading.Tasks;
using Xamarin.Forms;

namespace BounceButton {

   public partial class BounceButtonPage : ContentPage {

     public BounceButtonPage() {
       InitializeComponent();
     }

     async void OnButtonClicked(object sender, EventArgs args) {
      await button.TranslateTo(0, (Height - button.Height) / 2, 1000, Easing.BounceOut);
      await Task.Delay(2000);
      await button.TranslateTo(0, 0, 1000, Easing.SpringOut);
     }
   }
}
```

El método TranslateTo anima las propiedades TranslationX y TranslationY. Los primeros dos argumentos se denominan x e y, e indican los valores finales que se establecerán en TranslationX y TranslationY. La primera llamada TranslateTo aquí no mueve el botón horizontalmente, por lo que el primer argumento es 0. El segundo argumento es la distancia entre la parte inferior del botón y la parte inferior de la página. El botón está centrado verticalmente en la página, de modo que la distancia es la mitad de la altura de la página menos la mitad de la altura del botón.

Esa primera animación se realiza en 1000 milisegundos. Luego hay una demora de dos segundos, y el botón vuelve a su posición original con los argumentos x e y de 0. La segunda animación TranslateTo utiliza la función Easing.SpringOut, por lo que probablemente se espere que el botón sobre dispare su marca y luego vuelve a colocarte en su posición final.

Sin embargo, el método TranslateTo sujeta la salida de cualquier función de aceleración que se encuentre fuera del rango de 0 a 1. Más adelante en este capítulo verá una solución para esa falla en el método TranslateTo.

3.5. Animaciones de entrada

Un tipo común de animación en la programación de la vida real ocurre cuando una página se hace visible por primera vez. Los diversos elementos de la página se pueden animar brevemente antes de establecerse en sus estados finales.

Esto a menudo se llama animación de entrada y puede implicar:

- Translation, para mover elementos a sus posiciones finales.
- Scale, para ampliar o reducir los elementos a sus tamaños finales.
- Cambios en la opacidad, Opacity, para desvanecer los elementos a la vista.
- Rotación 3D para que parezca que una página entera se abre a la vista.

En general, se pretende que los elementos de la página se apoyen en los valores predeterminados de estas propiedades: valores de TranslationX y TranslationY de 0, Scale y Opacity de 1 y todas las propiedades de Rotation en 0.

En otras palabras, las animaciones de entrada deben terminar en el valor predeterminado de cada propiedad, lo que significa que comienzan en valores no predeterminados. Este enfoque también permite que el programa aplique otras transformaciones a estos elementos en un momento posterior sin tener en cuenta las animaciones de entrada.

A modo de ejemplo, se puede considerar el programa **FadingEntrance**, aquí hay una página con varios elementos únicamente con fines de demostración:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="FadingEntrance.FadingEntrancePage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="10, 20, 10, 10" />
            <On Platform="Android, UWP" Value="10" />
        </OnPlatform>
    </ContentPage.Padding>

    <StackLayout x:Name="stackLayout">
        <Label Text="The App"
               Style="{DynamicResource TitleStyle}"
               FontAttributes="Italic"
               HorizontalOptions="Center" />

        <Button Text="Countdown"
                FontSize="Large"
                HorizontalOptions="Center" />

        <Label Text="Primary Slider"
               HorizontalOptions="Center" />

        <Slider Value="0.5" />

        <ListView HorizontalOptions="Center"
                  WidthRequest="200">
            <ListView.ItemsSource>
                <x:Array Type="{x:Type Color}">
                    <Color>Red</Color>
                    <Color>Green</Color>
                    <Color>Blue</Color>
                    <Color>Aqua</Color>
                    <Color>Purple</Color>
                    <Color>Yellow</Color>
                </x:Array>
            </ListView.ItemsSource>

            <ListView.ItemTemplate>
                <DataTemplate>
                    <ViewCell>
```

```
                        <BoxView Color="{Binding}" />
                    </ViewCell>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>

        <Label Text="Secondary Slider"
               HorizontalOptions="Center" />

        <Slider Value="0.5" />

        <Button Text="Launch"
                FontSize="Large"
                HorizontalOptions="Center" />
    </StackLayout>
</ContentPage>
```

El archivo de código subyacente reescribe el método OnAppearing. El método OnAppearing se llama después de que la página se distribuya, pero antes de que la página sea visible. Todos los elementos de la página se han dimensionado y colocado, por lo que si se necesita obtener esa información, se puede hacer durante este método. El método OnAppearing establece la propiedad Opacity del StackLayout a 0 (haciendo que todo lo que esté dentro del StackLayout sea invisible) y luego lo anima a 1:

```
using System;
using Xamarin.Forms;

namespace FadingEntrance {

   public partial class FadingEntrancePage : ContentPage {

          public FadingEntrancePage() {
                 InitializeComponent();
          }

          protected override void OnAppearing() {
                 base.OnAppearing();
                 stackLayout.Opacity = 0;
                 stackLayout.FadeTo(1, 3000);
          }
   }
}
```

3.6. Animaciones infinitas

En el extremo opuesto de las animaciones de entrada están las animaciones para siempre o infinitas. Una aplicación puede implementar una animación que continúa *para siempre*, o al menos hasta que el programa finalice. A menudo, el único propósito de tales animaciones es demostrar las capacidades de un sistema de animación, pero preferiblemente de una manera agradable o divertida.

El primer ejemplo se llama **FadingTextAnimation** y usa FadeTo para atenuar dos elementos de tipo Label hacia adentro y hacia afuera. El archivo XAML coloca ambos elementos de etiqueta en una cuadrícula de una sola celda para que se superpongan. La segunda tiene su propiedad Opacity establecida en 0:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="FadingTextAnimation.FadingTextAnimationPage"
             BackgroundColor="White"
             SizeChanged="OnPageSizeChanged">
```

```
        <ContentPage.Resources>
            <ResourceDictionary>
                <Style TargetType="Label">
                    <Setter Property="HorizontalTextAlignment" Value="Center" />
                    <Setter Property="VerticalTextAlignment" Value="Center" />
                </Style>
            </ResourceDictionary>
        </ContentPage.Resources>

        <Grid>
            <Label x:Name="label1"
                   Text="MORE"
                   TextColor="Blue" />

            <Label x:Name="label2"
                   Text="CODE"
                   TextColor="Red"
                   Opacity="0" />
        </Grid>
    </ContentPage>
```

Una forma sencilla de crear una animación que se ejecute para siempre es colocar todo su código de animación, utilizando aguardar, por supuesto, dentro de un bucle while con una condición de verdadero. Entonces llama a ese método desde el constructor:

```
using System;
using System.Threading.Tasks;
using Xamarin.Forms;

namespace FadingTextAnimation {

   public partial class FadingTextAnimationPage : ContentPage {

          public FadingTextAnimationPage() {
                 InitializeComponent();
                 // Start the animation going.
                 AnimationLoop();
          }

          void OnPageSizeChanged(object sender, EventArgs args) {
                 if (Width > 0) {
                        double fontSize = 0.3 * Width;
                        label1.FontSize = fontSize;
                        label2.FontSize = fontSize;
                 }
          }

          async void AnimationLoop() {
                 while (true) {
                        await Task.WhenAll(label1.FadeTo(0, 1000),
                                                   label2.FadeTo(1, 1000));

                        await Task.WhenAll(label1.FadeTo(1, 1000),
                                                   label2.FadeTo(0, 1000));
                 }
          }
   }
}
```