

Contents

SkiaSharp Graphics in Xamarin.Forms

Drawing Basics

Drawing a Simple Circle

Integrating with Xamarin.Forms

Pixels and Device-Independent Units

Basic Animation

Integrating Text and Graphics

Bitmap Basics

Transparency

Lines and Paths

Lines and Stroke Caps

Path Basics

The Path Fill Types

Polylines and Parametric Equations

Dots and Dashes

Finger Painting

Transforms

The Translate Transform

The Scale Transform

The Rotate Transform

The Skew Transform

Matrix Transforms

Touch Manipulations

Non-Affine Transforms

3D Rotation

Curves and Paths

Three Ways to Draw an Arc

Three Types of Bézier Curves

SVG Path Data

[Clipping with Paths and Regions](#)

[Path Effects](#)

[Paths and Text](#)

[Path Information and Enumeration](#)

[Bitmaps](#)

[Displaying Bitmaps](#)

[Creating and Drawing on Bitmaps](#)

[Cropping Bitmaps](#)

[Segmented Display of Bitmaps](#)

[Saving Bitmaps to Files](#)

[Accessing Bitmap Pixel Bits](#)

[Animating Bitmaps](#)

[Effects](#)

[Shaders](#)

[Linear Gradient](#)

[Circular Gradients](#)

[Bitmap Tiling](#)

[Noise](#)

[Blend Modes](#)

[Porter-Duff Blend Modes](#)

[Separable Blend Modes](#)

[Non-Separable Blend Modes](#)

[Mask Filters](#)

[Image Filters](#)

[Color Filters](#)

[API reference](#)

SkiaSharp Graphics in Xamarin.Forms

3/5/2021 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

Use SkiaSharp for 2D graphics in your Xamarin.Forms applications

SkiaSharp is a 2D graphics system for .NET and C# powered by the open-source Skia graphics engine that is used extensively in Google products. You can use SkiaSharp in your Xamarin.Forms applications to draw 2D vector graphics, bitmaps, and text.

This guide assumes that you are familiar with Xamarin.Forms programming.

[Webinar: SkiaSharp for Xamarin.Forms](#)

SkiaSharp Preliminaries

SkiaSharp for Xamarin.Forms is packaged as a NuGet package. After you've created a Xamarin.Forms solution in Visual Studio or Visual Studio for Mac, you can use the NuGet package manager to search for the **SkiaSharp.Views.Forms** package and add it to your solution. If you check the **References** section of each project after adding SkiaSharp, you can see that various **SkiaSharp** libraries have been added to each of the projects in the solution.

If your Xamarin.Forms application targets iOS, edit its **Info.plist** file to change the minimum deployment target to iOS 8.0.

In any C# page that uses SkiaSharp you'll want to include a `using` directive for the `SkiaSharp` namespace, which encompasses all the SkiaSharp classes, structures, and enumerations that you'll use in your graphics programming. You'll also want a `using` directive for the `SkiaSharp.Views.Forms` namespace for the classes specific to Xamarin.Forms. This is a much smaller namespace, with the most important class being `SKCanvasView`. This class derives from the Xamarin.Forms `View` class and hosts your SkiaSharp graphics output.

IMPORTANT

The `SkiaSharp.Views.Forms` namespace also contains an `SKGLView` class that derives from `View` but uses OpenGL for rendering graphics. For purposes of simplicity, this guide restricts itself to `SKCanvasView`, but using `SKGLView` instead is quite similar.

SkiaSharp Drawing Basics

Some of the simplest graphics figures you can draw with SkiaSharp are circles, ovals, and rectangles. In displaying these figures, you will learn about SkiaSharp coordinates, sizes, and colors. The display of text and bitmaps is more complex, but these articles also introduce those techniques.

SkiaSharp Lines and Paths

A graphics path is a series of connected straight lines and curves. Paths can be stroked, filled, or both. This article encompasses many aspects of line drawing, including stroke ends and joins, and dashed and dotted lines, but stops short of curve geometries.

SkiaSharp Transforms

Transforms allow graphics objects to be uniformly translated, scaled, rotated, or skewed. This article also shows how you can use a standard 3-by-3 transform matrix for creating non-affine transforms and applying transforms to paths.

SkiaSharp Curves and Paths

The exploration of paths continues with adding curves to a path objects, and exploiting other powerful path features. You'll see how you can specify an entire path in a concise text string, how to use path effects, and how to dig into path internals.

SkiaSharp Bitmaps

Bitmaps are rectangular arrays of bits corresponding to the pixels of a display device. This series of articles shows how to load, save, display, create, draw on, animate, and access the bits of SkiaSharp bitmaps.

SkiaSharp Effects

Effects are properties that alter the normal display of graphics, including linear and circular gradients, bitmap tiling, blend modes, blur, and others.

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)
- [SkiaSharp with Xamarin.Forms Webinar \(video\)](#)

SkiaSharp Drawing Basics

3/5/2021 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

Learn the basics of SkiaSharp graphics concepts and coordinates

After you have added the SkiaSharp NuGet package to your Xamarin.Forms application, you can begin using SkiaSharp graphics. The [SkiaSharpFormsDemos](#) solution includes numerous pages that demonstrate SkiaSharp programming techniques in progressively more advanced lessons.

All the sample programs in this section appear under the heading **SkiaSharp Drawing Basics** in the home page of the [SkiaSharpFormsDemos](#) program, and in the **Basics** folder of the solution.

Drawing a Simple Circle

Learn the basics of SkiaSharp drawing, including canvases and paint objects.

Integrating with Xamarin.Forms

Create interactive SkiaSharp graphics by responding to touch input and integrating with Xamarin.Forms elements.

Pixels and Device-Independent Units

Explore the differences between SkiaSharp coordinates and Xamarin.Forms coordinates.

Basic Animation

Discover how to animate your SkiaSharp graphics.

Integrating Text and Graphics

See how to determine the size of rendered text strings to integrate text with SkiaSharp graphics.

Bitmap Basics

Load bitmaps from various sources and display them.

Transparency

Use transparency to combine multiple images into a composite scene.

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Drawing a Simple Circle in SkiaSharp

3/5/2021 • 5 minutes to read • [Edit Online](#)

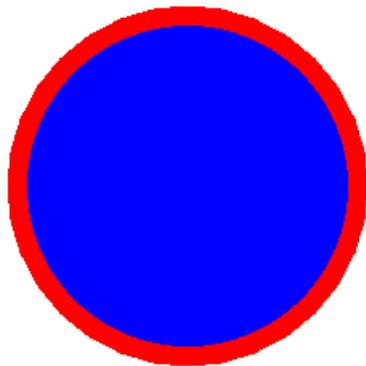


[Download the sample](#)

Learn the basics of SkiaSharp drawing, including canvases and paint objects

This article introduces the concepts of drawing graphics in Xamarin.Forms using SkiaSharp, including creating an `SKCanvasView` object to host the graphics, handling the `PaintSurface` event, and using a `SKPaint` object to specify color and other drawing attributes.

The `SkiaSharpFormsDemos` program contains all the sample code for this series of SkiaSharp articles. The first page is entitled **Simple Circle** and invokes the page class `SimpleCirclePage`. This code shows how to draw a circle in the center of the page with a radius of 100 pixels. The outline of the circle is red, and the interior of the circle is blue.



The `SimpleCircle` page class derives from `ContentPage` and contains two `using` directives for the SkiaSharp namespaces:

```
using SkiaSharp;
using SkiaSharp.Views.Forms;
```

The following constructor of the class creates an `SKCanvasView` object, attaches a handler for the `PaintSurface` event, and sets the `SKCanvasView` object as the content of the page:

```
public SimpleCirclePage()
{
    Title = "Simple Circle";

    SKCanvasView canvasView = new SKCanvasView();
    canvasView.PaintSurface += OnCanvasViewPaintSurface;
    Content = canvasView;
}
```

The `SKCanvasView` occupies the entire content area of the page. You can alternatively combine an `SKCanvasView` with other Xamarin.Forms `View` derivatives, as you'll see in other examples.

The `PaintSurface` event handler is where you do all your drawing. This method can be called multiple times while your program is running, so it should maintain all the information necessary to recreate the graphics display:

```
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    ...
}
```

The `SKPaintSurfaceEventArgs` object that accompanies the event has two properties:

- `Info` of type `SKImageInfo`
- `Surface` of type `SKSurface`

The `SKImageInfo` structure contains information about the drawing surface, most importantly, its width and height in pixels. The `SKSurface` object represents the drawing surface itself. In this program, the drawing surface is a video display, but in other programs an `SKSurface` object can also represent a bitmap that you use SkiaSharp to draw on.

The most important property of `SKSurface` is `Canvas` of type `SKCanvas`. This class is a graphics drawing context that you use to perform the actual drawing. The `SKCanvas` object encapsulates a graphics state, which includes graphics transforms and clipping.

Here's a typical start of a `PaintSurface` event handler:

```
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();
    ...
}
```

The `Clear` method clears the canvas with a transparent color. An overload lets you specify a background color for the canvas.

The goal here is to draw a red circle filled with blue. Because this particular graphic image contains two different colors, the job needs to be done in two steps. The first step is to draw the outline of the circle. To specify the color and other characteristic of the line, you create and initialize an `SKPaint` object:

```
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    ...
    SKPaint paint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = Color.Red.ToSKColor(),
        StrokeWidth = 25
    };
    ...
}
```

The `style` property indicates that you want to *stroke* a line (in this case the outline of the circle) rather than *fill* the interior. The three members of the `SKPaintStyle` enumeration are as follows:

- `Fill`
- `Stroke`

- [StrokeAndFill](#)

The default is `Fill`. Use the third option to stroke the line and fill the interior with the same color.

Set the `Color` property to a value of type `SKColor`. One way to get an `SKColor` value is by converting a Xamarin.Forms `Color` value to an `SKColor` value using the extension method `ToSKColor`. The `Extensions` class in the `SkiaSharp.Views.Forms` namespace includes other methods that convert between Xamarin.Forms values and SkiaSharp values.

The `StrokeWidth` property indicates the thickness of the line. Here it's set to 25 pixels.

You use that `SKPaint` object to draw the circle:

```
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    ...
    canvas.DrawCircle(info.Width / 2, info.Height / 2, 100, paint);
    ...
}
```

Coordinates are specified relative to the upper-left corner of the display surface. X coordinates increase to the right and Y coordinates increase going down. In discussion about graphics, often the mathematical notation (x, y) is used to denote a point. The point (0, 0) is the upper-left corner of the display surface and is often called the *origin*.

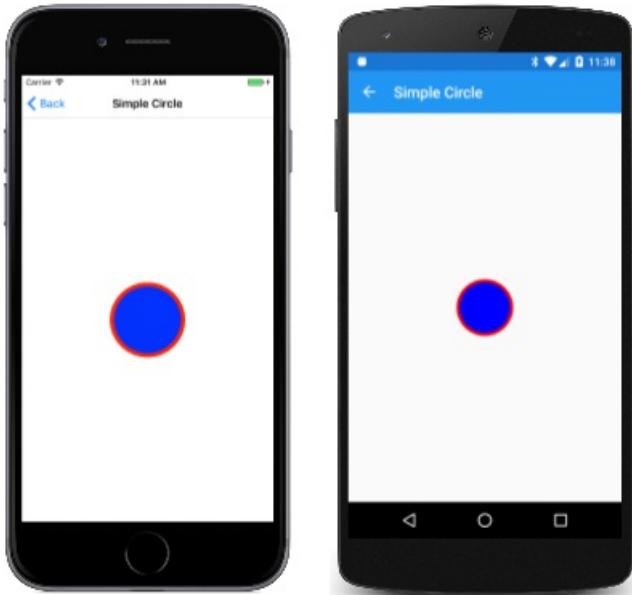
The first two arguments of `DrawCircle` indicate the X and Y coordinates of the center of the circle. These are assigned to half the width and height of the display surface to put the center of the circle in the center of the display surface. The third argument specifies the circle's radius, and the last argument is the `SKPaint` object.

To fill the interior of the circle, you can alter two properties of the `SKPaint` object and call `DrawCircle` again. This code also shows an alternative way to get an `SKColor` value from one of the many fields of the `SKColors` structure:

```
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    ...
    paint.Style = SKPaintStyle.Fill;
    paint.Color = SKColors.Blue;
    canvas.DrawCircle(args.Info.Width / 2, args.Info.Height / 2, 100, paint);
}
```

This time, the `DrawCircle` call fills the circle using the new properties of the `SKPaint` object.

Here's the program running on iOS and Android:



When running the program yourself, you can turn the phone or simulator sideways to see how the graphic is redrawn. Each time the graphic needs to be redrawn, the `PaintSurface` event handler is called again.

It's also possible to color graphical objects with gradients or bitmap tiles. These options are discussed in the section on [SkiaSharp shaders](#).

An `SKPaint` object is little more than a collection of graphics drawing properties. These objects are lightweight. You can reuse `SKPaint` objects as this program does, or you can create multiple `SKPaint` objects for various combinations of drawing properties. You can create and initialize these objects outside of the `PaintSurface` event handler, and you can save them as fields in your page class.

NOTE

The `SKPaint` class defines an `IsAntialias` to enable anti-aliasing in the rendering of your graphics. Anti-aliasing generally results in visually smoother edges, so you'll probably want to set this property to `true` in most of your `SKPaint` objects. For purposes of simplicity, this property is *not* set in most of the sample pages.

Although the width of the circle's outline is specified as 25 pixels — or one-quarter of the radius of the circle — it appears to be thinner, and there's a good reason for that: Half the width of the line is obscured by the blue circle. The arguments to the `DrawCircle` method define the abstract geometric coordinates of a circle. The blue interior is sized to that dimension to the nearest pixel, but the 25-pixel-wide outline straddles the geometric circle — half on the inside and half on the outside.

The next sample in the [Integrating with Xamarin.Forms](#) article demonstrates this visually.

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

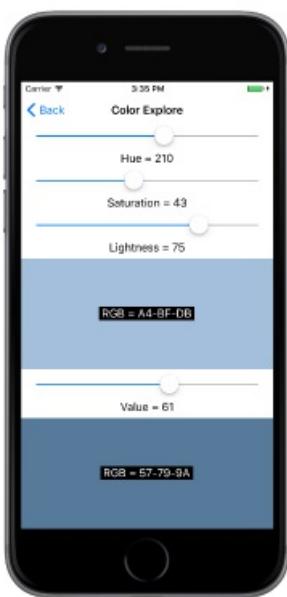
Integrating with Xamarin.Forms

3/5/2021 • 4 minutes to read • [Edit Online](#)

 [Download the sample](#)

Create SkiaSharp graphics that respond to touch and Xamarin.Forms elements

SkiaSharp graphics can integrate with the rest of Xamarin.Forms in several ways. You can combine a SkiaSharp canvas and Xamarin.Forms elements on the same page, and even position Xamarin.Forms elements on top of a SkiaSharp canvas:



Another approach to creating interactive SkiaSharp graphics in Xamarin.Forms is through touch. The second page in the [SkiaSharpFormsDemos](#) program is entitled **Tap Toggle Fill**. It draws a simple circle two ways — without a fill and with a fill — toggled by a tap. The [TapToggleFillPage](#) class shows how you can alter SkiaSharp graphics in response to user input.

For this page, the `SKCanvasView` class is instantiated in the `TapToggleFill.xaml` file, which also sets a Xamarin.Forms `TapGestureRecognizer` on the view:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.TapTogglefillPage"
    Title="Tap Toggle Fill">

    <skia:SKCanvasView PaintSurface="OnCanvasViewPaintSurface">
        <skia:SKCanvasView.GestureRecognizers>
            <TapGestureRecognizer Tapped="OnCanvasViewTapped" />
        </skia:SKCanvasView.GestureRecognizers>
    </skia:SKCanvasView>
</ContentPage>
```

Notice the `skia` XML namespace declaration.

The `Tapped` handler for the `TapGestureRecognizer` object simply toggles the value of a Boolean field and calls the `InvalidateSurface` method of `SKCanvasView`:

```

bool showFill = true;
...
void OnCanvasViewTapped(object sender, EventArgs args)
{
    showFill ^= true;
    (sender as SKCanvasView).InvalidateSurface();
}

```

The call to `InvalidateSurface` effectively generates a call to the `PaintSurface` handler, which uses the `showFill` field to fill or not fill the circle:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    SKPaint paint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = Color.Red.ToSKColor(),
        StrokeWidth = 50
    };
    canvas.DrawCircle(info.Width / 2, info.Height / 2, 100, paint);

    if (showFill)
    {
        paint.Style = SKPaintStyle.Fill;
        paint.Color = SKColors.Blue;
        canvas.DrawCircle(info.Width / 2, info.Height / 2, 100, paint);
    }
}

```

The `StrokeWidth` property has been set to 50 to accentuate the difference. You can also see the whole line width by drawing the interior first and then the outline. By default, graphics figures that are drawn later in the `PaintSurface` event handler obscure those drawn earlier in the handler.

The [Color Explore](#) page demonstrates how you can also integrate SkiaSharp graphics with other Xamarin.Forms elements, and also demonstrates the difference between two alternative methods for defining colors in SkiaSharp. The static `SKColor.FromHsl` method creates an `SKColor` value based on the Hue-Saturation-Lightness model:

```
public static SKColor FromHsl (Single h, Single s, Single l, Byte a)
```

The static `SKColor.FromHsv` method creates an `SKColor` value based on the similar Hue-Saturation-Value model:

```
public static SKColor FromHsv (Single h, Single s, Single v, Byte a)
```

In both cases, the `h` argument ranges from 0 to 360. The `s`, `l`, and `v` arguments range from 0 to 100. The `a` (alpha or opacity) argument ranges from 0 to 255.

The [ColorExplorePage.xaml](#) file creates two `SKCanvasView` objects in a `StackLayout` side by side with `Slider` and `Label` views that allow the user to select HSL and HSV color values:

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

```

<http://schemas.microsoft.com/winfx/2009/xaml>
xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
x:Class="SkiaSharpFormsDemos.Basics.ColorExplorePage"
Title="Color Explore">

<StackLayout>
    <!-- Hue slider -->
    <Slider x:Name="hueSlider"
        Maximum="360"
        Margin="20, 0"
        ValueChanged="OnSliderValueChanged" />

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

    <!-- Saturation slider -->
    <Slider x:Name="saturationSlider"
        Maximum="100"
        Margin="20, 0"
        ValueChanged="OnSliderValueChanged" />

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

    <!-- Lightness slider -->
    <Slider x:Name="lightnessSlider"
        Maximum="100"
        Margin="20, 0"
        ValueChanged="OnSliderValueChanged" />

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

    <!-- HSL canvas view -->
    <Grid VerticalOptions="FillAndExpand">
        <skia:SKCanvasView x:Name="hslCanvasView"
            PaintSurface="OnHslCanvasViewPaintSurface" />

        <Label x:Name="hslLabel"
            HorizontalOptions="Center"
            VerticalOptions="Center"
            BackgroundColor="Black"
            TextColor="White" />
    </Grid>

    <!-- Value slider -->
    <Slider x:Name="valueSlider"
        Maximum="100"
        Margin="20, 0"
        ValueChanged="OnSliderValueChanged" />

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

    <!-- HSV canvas view -->
    <Grid VerticalOptions="FillAndExpand">
        <skia:SKCanvasView x:Name="hsvCanvasView"
            PaintSurface="OnHsvCanvasViewPaintSurface" />

        <Label x:Name="hsvLabel"
            HorizontalOptions="Center"
            VerticalOptions="Center"
            BackgroundColor="Black"
            TextColor="White" />
    </Grid>

```

```

        BackgroundColor="Black"
        TextColor="White" />
    </Grid>
</StackLayout>
</ContentPage>

```

The two `SKCanvasView` elements are in a single-cell `Grid` with a `Label` sitting on top for displaying the resultant RGB color value.

The `ColorExplorePage.xaml.cs` code-behind file is relatively simple. The shared `ValueChanged` handler for the three `Slider` elements simply invalidates both `SKCanvasView` elements. The `PaintSurface` handlers clear the canvas with the color indicated by the `Slider` elements, and also set the `Label` sitting on top of the `SKCanvasView` elements:

```

public partial class ColorExplorePage : ContentPage
{
    public ColorExplorePage()
    {
        InitializeComponent();

        hueSlider.Value = 0;
        saturationSlider.Value = 100;
        lightnessSlider.Value = 50;
        valueSlider.Value = 100;
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        hslCanvasView.InvalidateSurface();
        hsvCanvasView.InvalidateSurface();
    }

    void OnHslCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKColor color = SKColor.FromHsl((float)hueSlider.Value,
                                         (float)saturationSlider.Value,
                                         (float)lightnessSlider.Value);
        args.Surface.Canvas.Clear(color);

        hslLabel.Text = String.Format(" RGB = {0:X2}-{1:X2}-{2:X2} ",
                                     color.Red, color.Green, color.Blue);
    }

    void OnHsvCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKColor color = SKColor.FromHsv((float)hueSlider.Value,
                                         (float)saturationSlider.Value,
                                         (float)valueSlider.Value);
        args.Surface.Canvas.Clear(color);

        hsvLabel.Text = String.Format(" RGB = {0:X2}-{1:X2}-{2:X2} ",
                                     color.Red, color.Green, color.Blue);
    }
}

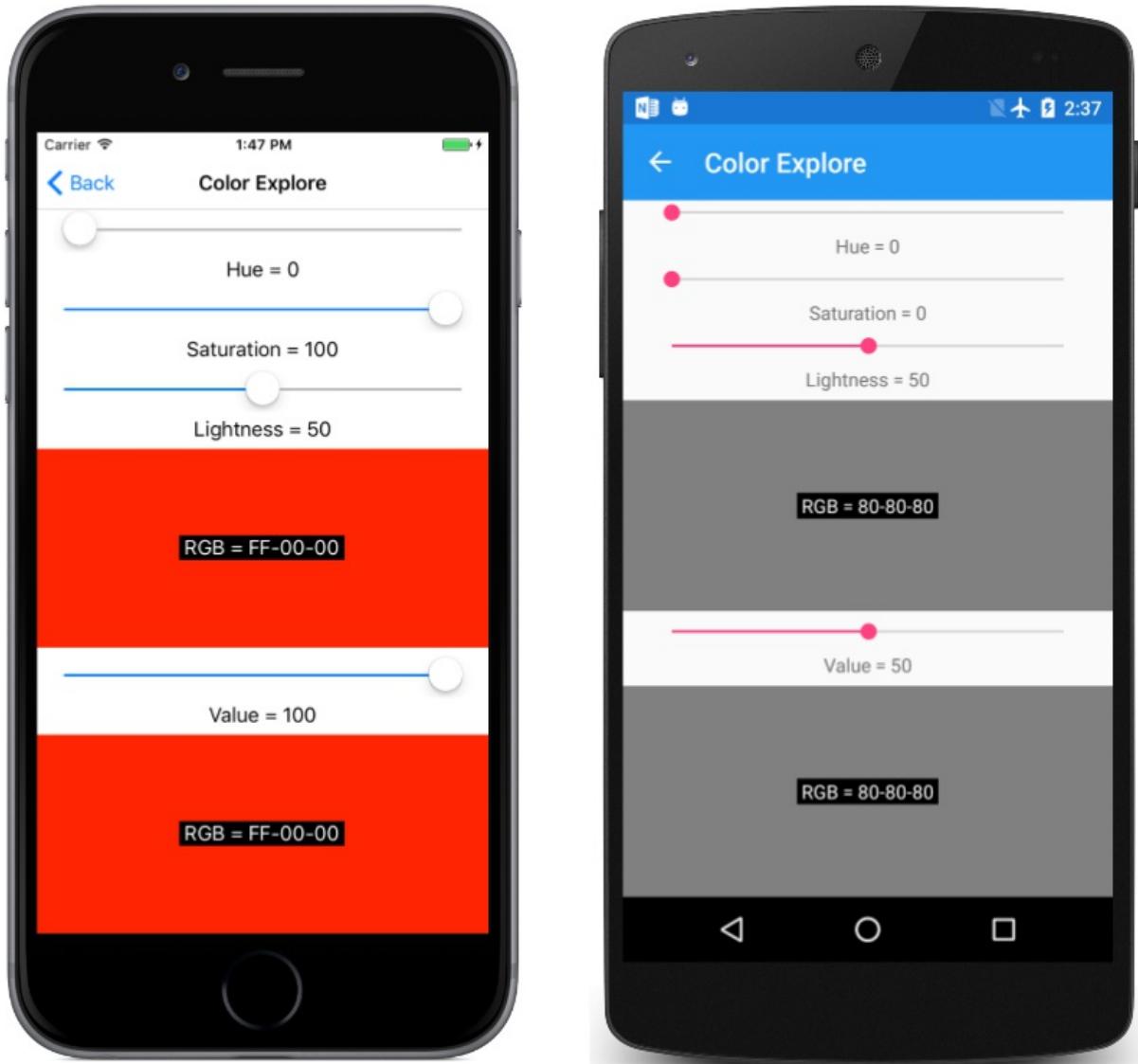
```

In both the HSL and HSV color models, the Hue value ranges from 0 to 360 and indicates the dominant hue of the color. These are the traditional colors of the rainbow: red, orange, yellow, green, blue, indigo, violet, and back in a circle to red.

In the HSL model, a 0 value for Lightness is always black, and a 100 value is always white. When the Saturation value is 0, Lightness values between 0 and 100 are shades of gray. Increasing the Saturation adds more color. Pure colors (which are RGB values with one component equal to 255, another equal to 0, and the third ranging from 0 to 255) occur when the Saturation is 100 and the Lightness is 50.

In the HSV model, pure colors result when both the Saturation and Value are 100. When Value is 0, regardless of any other settings, the color is black. Gray shades occur when the Saturation is 0 and Value ranges from 0 to 100.

But the best way to get a feel for the two models is to experiment with them yourself:



Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

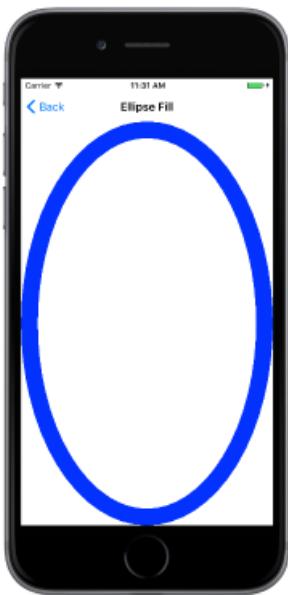
Pixels and Device-Independent Units

3/5/2021 • 6 minutes to read • [Edit Online](#)

 [Download the sample](#)

Explore the differences between SkiaSharp coordinates and Xamarin.Forms coordinates

This article explores the differences in the coordinate system used in SkiaSharp and Xamarin.Forms. You can obtain information to convert between the two coordinate systems and also draw graphics that fill a particular area:



If you've been programming in Xamarin.Forms for a while, you might have a feel for Xamarin.Forms coordinates and sizes. The circles drawn in the two previous articles might seem a little small to you.

Those circles *are* small in comparison with Xamarin.Forms sizes. By default, SkiaSharp draws in units of pixels while Xamarin.Forms bases coordinates and sizes on a device-independent unit established by the underlying platform. (More information on the Xamarin.Forms coordinate system can be found in [Chapter 5. Dealing with Sizes](#) of the book *Creating Mobile Apps with Xamarin.Forms*.)

The page in the [SkewSharpFormsDemos](#) program entitled **Surface Size** uses SkiaSharp text output to show the size of the display surface from three different sources:

- The normal Xamarin.Forms `Width` and `Height` properties of the `SKCanvasView` object.
- The `CanvasSize` property of the `SKCanvasView` object.
- The `Size` property of the `SKImageInfo` value, which is consistent with the `Width` and `Height` properties used in the two previous pages.

The `SurfaceSizePage` class shows how to display these values. The constructor saves the `SKCanvasView` object as a field, so it can be accessed in the `PaintSurface` event handler:

```
SKCanvasView canvasView;

public SurfaceSizePage()
{
    Title = "Surface Size";

    canvasView = new SKCanvasView();
    canvasView.PaintSurface += OnCanvasViewPaintSurface;
    Content = canvasView;
}
```

`SKCanvas` includes six different `DrawText` methods, but this `DrawText` method is the simplest:

```
public void DrawText (String text, Single x, Single y, SKPaint paint)
```

You specify the text string, the X and Y coordinates where the text is to begin, and an `SKPaint` object. The X coordinate specifies where the left side of the text is positioned, but watch out: The Y coordinate specifies the position of the *baseline* of the text. If you've ever written by hand on lined paper, the baseline is the line on which characters sit, and below which descenders (such as those on the letters g, p, q, and y) descend.

The `SKPaint` object allows you to specify the color of the text, the font family, and the text size. By default, the `FontSize` property has a value of 12, which results in tiny text on high-resolution devices such as phones. In anything but the simplest applications, you'll also need some information on the size of the text you're displaying. The `SKPaint` class defines a `FontMetrics` property and several `MeasureText` methods, but for less fancy needs, the `FontSpacing` property provides a recommended value for spacing successive lines of text.

The following `PaintSurface` handler creates an `SKPaint` object for a `FontSize` of 40 pixels, which is the desired vertical height of the text from the top of ascenders to the bottom of descenders. The `FontSpacing` value that the `SKPaint` object returns is a little larger than that, about 47 pixels.

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    SKPaint paint = new SKPaint
    {
        Color = SKColors.Black,
        TextSize = 40
    };

    float fontSpacing = paint.FontSpacing;
    float x = 20;           // left margin
    float y = fontSpacing; // first baseline
    float indent = 100;

    canvas.DrawText("SKCanvasView Height and Width:", x, y, paint);
    y += fontSpacing;
    canvas.DrawText(String.Format("{0:F2} x {1:F2}",
        canvasView.Width, canvasView.Height),
        x + indent, y, paint);
    y += fontSpacing * 2;
    canvas.DrawText("SKCanvasView CanvasSize:", x, y, paint);
    y += fontSpacing;
    canvas.DrawText(canvasView.CanvasSize.ToString(), x + indent, y, paint);
    y += fontSpacing * 2;
    canvas.DrawText("SKImageInfo Size:", x, y, paint);
    y += fontSpacing;
    canvas.DrawText(info.Size.ToString(), x + indent, y, paint);
}

```

The method begins the first line of text with an X coordinate of 20 (for a little margin at the left) and a Y coordinate of `fontSpacing`, which is a little more than what's necessary to display the full height of the first line of text at the top of the display surface. After each call to `DrawText`, the Y coordinate is increased by one or two increments of `fontSpacing`.

Here's the program running:



As you can see, the `CanvasSize` property of the `SKCanvasView` and the `Size` property of the `SKImageInfo` value are consistent in reporting the pixel dimensions. The `Height` and `Width` properties of the `SKCanvasView` are

Xamarin.Forms properties, and report the size of the view in the device-independent units defined by the platform.

The iOS seven simulator on the left has two pixels per device-independent unit, and the Android Nexus 5 in the center has three pixels per unit. That's why the simple circle shown earlier has different sizes on different platforms.

If you'd prefer to work entirely in device-independent units, you can do so by setting the `IgnorePixelScaling` property of the `SKCanvasView` to `true`. However, you might not like the results. SkiaSharp renders the graphics on a smaller device surface, with a pixel size equal to the size of the view in device-independent units. (For example, SkiaSharp would use a display surface of 360 x 512 pixels on the Nexus 5.) It then scales up that image in size, resulting in noticeable bitmap jaggies.

To maintain the same image resolution, a better solution is to write your own simple functions to convert between the two coordinate systems.

In addition to the `DrawCircle` method, `SKCanvas` also defines two `DrawOval` methods that draw an ellipse. An ellipse is defined by two radii rather than a single radius. These are known as the *major radius* and the *minor radius*. The `Drawoval` method draws an ellipse with the two radii parallel to the X and Y axes. (If you need to draw an ellipse with axes that are not parallel to the X and Y axes, you can use a rotation transform as discussed in the article [The Rotate Transform](#) or a graphics path as discussed in the article [Three Ways to Draw an Arc](#).) This overload of the `DrawOval` method names the two radii parameters `rx` and `ry` to indicate that they are parallel to the X and Y axes:

```
public void DrawOval (Single cx, Single cy, Single rx, Single ry, SKPaint paint)
```

Is it possible to draw an ellipse that fills the display surface? The [Ellipse Fill](#) page demonstrates how. The `PaintSurface` event handler in the [EllipseFillPage.xaml.cs](#) class subtracts half the stroke width from the `xRadius` and `yRadius` values to fit the whole ellipse and its outline within the display surface:

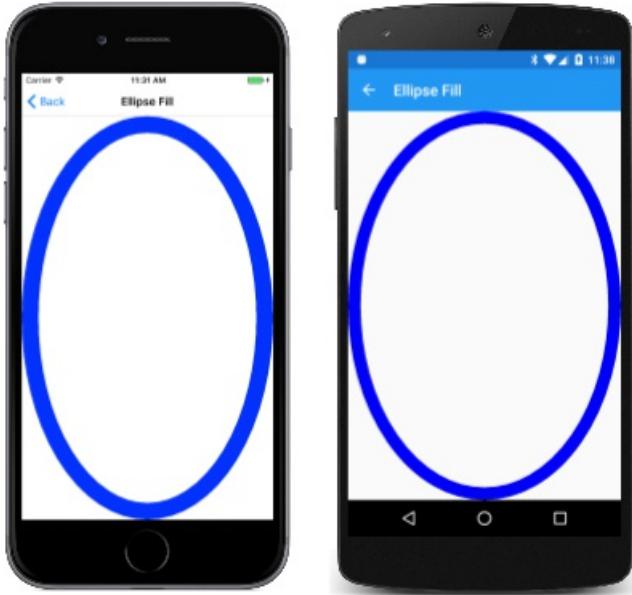
```
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    float strokeWidth = 50;
    float xRadius = (info.Width - strokeWidth) / 2;
    float yRadius = (info.Height - strokeWidth) / 2;

    SKPaint paint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Blue,
        StrokeWidth = strokeWidth
    };
    canvas.DrawOval(info.Width / 2, info.Height / 2, xRadius, yRadius, paint);
}
```

Here it is running:



The other `DrawOval` method has an `SKRect` argument, which is a rectangle defined in terms of the X and Y coordinates of its upper-left corner and lower-right corner. The oval fills that rectangle, which suggests that it might be possible to use it in the **Ellipse Fill** page like this:

```
SKRect rect = new SKRect(0, 0, info.Width, info.Height);
canvas.DrawOval(rect, paint);
```

However, that truncates all the edges of the outline of the ellipse on the four sides. You need to adjust all the `SKRect` constructor arguments based on the `strokeWidth` to make this work right:

```
SKRect rect = new SKRect(strokeWidth / 2,
                        strokeWidth / 2,
                        info.Width - strokeWidth / 2,
                        info.Height - strokeWidth / 2);
canvas.DrawOval(rect, paint);
```

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Basic Animation in SkiaSharp

3/5/2021 • 5 minutes to read • [Edit Online](#)



[Download the sample](#)

Discover how to animate your SkiaSharp graphics

You can animate SkiaSharp graphics in Xamarin.Forms by causing the `PaintSurface` method to be called periodically, each time drawing the graphics a little differently. Here's an animation shown later in this article with concentric circles that seemingly expand from the center:



The [Pulsating Ellipse](#) page in the [SkiaSharpFormsDemos](#) program animates the two axes of an ellipse so that it appears to be pulsating, and you can even control the rate of this pulsation. The [PulsatingEllipsePage.xaml](#) file instantiates a Xamarin.Forms `Slider` and a `Label` to display the current value of the slider. This is a common way to integrate an `SKCanvasView` with other Xamarin.Forms views:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.PulsatingEllipsePage"
    Title="Pulsating Ellipse">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Slider x:Name="slider"
            Grid.Row="0"
            Maximum="10"
            Minimum="0.1"
            Value="5"
            Margin="20, 0" />

        <Label Grid.Row="1"
            Text="{Binding Source={x:Reference slider},
                Path=Value,
                StringFormat='Cycle time = {0:F1} seconds'}"
            HorizontalTextAlignment="Center" />

        <skia:SKCanvasView x:Name="canvasView"
            Grid.Row="2"
            PaintSurface="OnCanvasViewPaintSurface" />
    </Grid>
</ContentPage>

```

The code-behind file instantiates a `Stopwatch` object to serve as a high-precision clock. The `OnAppearing` override sets the `pageIsActive` field to `true` and calls a method named `AnimationLoop`. The `OnDisappearing` override sets that `pageIsActive` field to `false`:

```

Stopwatch stopwatch = new Stopwatch();
bool pageIsActive;
float scale; // ranges from 0 to 1 to 0

public PulsatingEllipsePage()
{
    InitializeComponent();
}

protected override void OnAppearing()
{
    base.OnAppearing();
    pageIsActive = true;
    AnimationLoop();
}

protected override void OnDisappearing()
{
    base.OnDisappearing();
    pageIsActive = false;
}

```

The `AnimationLoop` method starts the `Stopwatch` and then loops while `pageIsActive` is `true`. This is essentially an "infinite loop" while the page is active, but it doesn't cause the program to hang because the loop concludes with a call to `Task.Delay` with the `await` operator, which lets other parts of the program function. The argument to `Task.Delay` causes it to complete after 1/30th second. This defines the frame rate of the animation.

```

async Task AnimationLoop()
{
    stopwatch.Start();

    while (pageIsActive)
    {
        double cycleTime = slider.Value;
        double t = stopwatch.Elapsed.TotalSeconds % cycleTime / cycleTime;
        scale = (1 + (float)Math.Sin(2 * Math.PI * t)) / 2;
        canvasView.InvalidateSurface();
        await Task.Delay(TimeSpan.FromSeconds(1.0 / 30));
    }

    stopwatch.Stop();
}

```

The `while` loop begins by obtaining a cycle time from the `Slider`. This is a time in seconds, for example, 5. The second statement calculates a value of `t` for *time*. For a `cycleTime` of 5, `t` increases from 0 to 1 every 5 seconds. The argument to the `Math.Sin` function in the second statement ranges from 0 to 2π every 5 seconds. The `Math.Sin` function returns a value ranging from 0 to 1 back to 0 and then to -1 and 0 every 5 seconds, but with values that change more slowly when the value is near 1 or -1. The value 1 is added so the values are always positive, and then it's divided by 2, so the values range from $\frac{1}{2}$ to 1 to $\frac{1}{2}$ to 0 to $\frac{1}{2}$, but slower when the value is around 1 and 0. This is stored in the `scale` field, and the `SKCanvasView` is invalidated.

The `PaintSurface` method uses this `scale` value to calculate the two axes of the ellipse:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    float maxRadius = 0.75f * Math.Min(info.Width, info.Height) / 2;
    float minRadius = 0.25f * maxRadius;

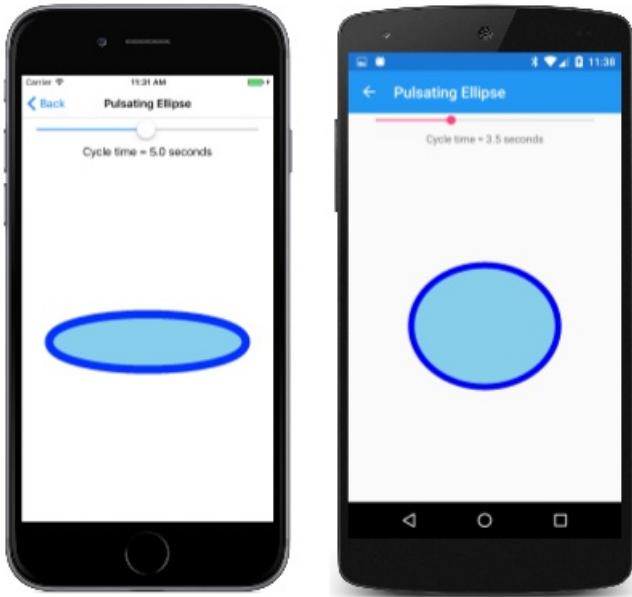
    float xRadius = minRadius * scale + maxRadius * (1 - scale);
    float yRadius = maxRadius * scale + minRadius * (1 - scale);

    using (SKPaint paint = new SKPaint())
    {
        paint.Style = SKPaintStyle.Stroke;
        paint.Color = SKColors.Blue;
        paint.StrokeWidth = 50;
        canvas.DrawOval(info.Width / 2, info.Height / 2, xRadius, yRadius, paint);

        paint.Style = SKPaintStyle.Fill;
        paint.Color = SKColors.SkyBlue;
        canvas.DrawOval(info.Width / 2, info.Height / 2, xRadius, yRadius, paint);
    }
}

```

The method calculates a maximum radius based on the size of the display area, and a minimum radius based on the maximum radius. The `scale` value is animated between 0 and 1 and back to 0, so the method uses that to compute an `xRadius` and `yRadius` that ranges between `minRadius` and `maxRadius`. These values are used to draw and fill an ellipse:



Notice that the `SKPaint` object is created in a `using` block. Like many SkiaSharp classes `SKPaint` derives from `SKObject`, which derives from `SKNativeObject`, which implements the `IDisposable` interface. `SKPaint` overrides the `Dispose` method to release unmanaged resources.

Putting `SKPaint` in a `using` block ensures that `Dispose` is called at the end of the block to free these unmanaged resources. This happens anyway when memory used by the `SKPaint` object is freed by the .NET garbage collector, but in animation code, it's best to be proactive in freeing memory in a more orderly way.

A better solution in this particular case would be to create two `SKPaint` objects once and save them as fields.

That's what the **Expanding Circles** animation does. The `ExpandingCirclesPage` class begins by defining several fields, including an `SKPaint` object:

```
public class ExpandingCirclesPage : ContentPage
{
    const double cycleTime = 1000;           // in milliseconds

    SKCanvasView canvasView;
    Stopwatch stopwatch = new Stopwatch();
    bool pageIsActive;
    float t;
    SKPaint paint = new SKPaint
    {
        Style = SKPaintStyle.Stroke
    };

    public ExpandingCirclesPage()
    {
        Title = "Expanding Circles";

        canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }
    ...
}
```

This program uses a different approach to animation based on the `Xamarin.Forms` `Device.StartTimer` method. The `t` field is animated from 0 to 1 every `cycleTime` milliseconds:

```

public class ExpandingCirclesPage : ContentPage
{
    ...
    protected override void OnAppearing()
    {
        base.OnAppearing();
        pageIsActive = true;
        stopwatch.Start();

        Device.StartTimer(TimeSpan.FromMilliseconds(33), () =>
        {
            t = (float)(stopwatch.Elapsed.TotalMilliseconds % cycleTime / cycleTime);
            canvasView.InvalidateSurface();

            if (!pageIsActive)
            {
                stopwatch.Stop();
            }
            return pageIsActive;
        });
    }

    protected override void OnDisappearing()
    {
        base.OnDisappearing();
        pageIsActive = false;
    }
    ...
}

```

The `PaintSurface` handler draws five concentric circles with animated radii. If the `baseRadius` variable is calculated as 100, then as `t` is animated from 0 to 1, the radii of the five circles increase from 0 to 100, 100 to 200, 200 to 300, 300 to 400, and 400 to 500. For most of the circles the `strokeWidth` is 50 but for the first circle, the `strokeWidth` animates from 0 to 50. For most of the circles, the color is blue, but for the last circle, the color is animated from blue to transparent. Notice the fourth argument to the `SKColor` constructor that specifies the opacity:

```

public class ExpandingCirclesPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        SKPoint center = new SKPoint(info.Width / 2, info.Height / 2);
        float baseRadius = Math.Min(info.Width, info.Height) / 12;

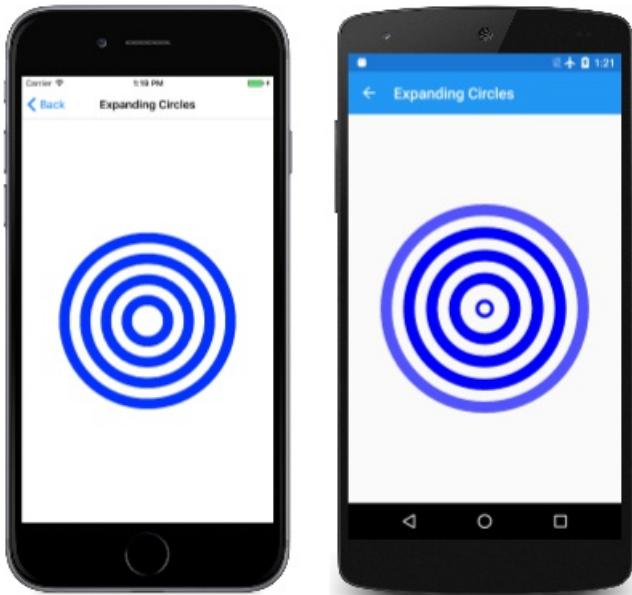
        for (int circle = 0; circle < 5; circle++)
        {
            float radius = baseRadius * (circle + t);

            paint.StrokeWidth = baseRadius / 2 * (circle == 0 ? t : 1);
            paint.Color = new SKColor(0, 0, 255,
                (byte)(255 * (circle == 4 ? (1 - t) : 1)));

            canvas.DrawCircle(center.X, center.Y, radius, paint);
        }
    }
}

```

The result is that the image looks the same when t equals 0 as when t equals 1, and the circles seem to continue expanding forever:



Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Integrating Text and Graphics

3/5/2021 • 6 minutes to read • [Edit Online](#)



[Download the sample](#)

See how to determine the size of rendered text string to integrate text with SkiaSharp graphics

This article demonstrates how to measure text, scale the text to a particular size, and integrate text with other graphics:



Hello SkiaSharp!

That image also includes a rounded rectangle. The SkiaSharp `Canvas` class includes `DrawRect` methods to draw a rectangle and `DrawRoundRect` methods to draw a rectangle with rounded corners. These methods allow the rectangle to be defined as an `SKRect` value or in other ways.

The [Framed Text](#) page centers a short text string on the page and surrounds it with a frame composed of a pair of rounded rectangles. The `FramedTextPage` class shows how it's done.

In SkiaSharp, you use the `SKPaint` class to set text and font attributes, but you can also use it to obtain the rendered size of text. The beginning of the following `PaintSurface` event handler calls two different `MeasureText` methods. The first `MeasureText` call has a simple `string` argument and returns the pixel width of the text based on the current font attributes. The program then calculates a new `TextSize` property of the `SKPaint` object based on that rendered width, the current `TextSize` property, and the width of the display area. This calculation is intended to set `TextSize` so that the text string to be rendered at 90% of the width of the screen:

```
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    string str = "Hello SkiaSharp!";

    // Create an SKPaint object to display the text
    SKPaint textPaint = new SKPaint
    {
        Color = SKColors.Chocolate
    };

    // Adjust TextSize property so text is 90% of screen width
    float textWidth = textPaint.MeasureText(str);
    textPaint.TextSize = 0.9f * info.Width * textPaint.TextSize / textWidth;

    // Find the text bounds
    SKRect textBounds = new SKRect();
    textPaint.MeasureText(str, ref textBounds);
    ...
}
```

The second `MeasureText` call has an `SKRect` argument, so it obtains both a width and height of the rendered

text. The `Height` property of this `SKRect` value depends on the presence of capital letters, ascenders, and descenders in the text string. Different `Height` values are reported for the text strings "mom", "cat", and "dog", for example.

The `Left` and `Top` properties of the `SKRect` structure indicate the coordinates of the upper-left corner of the rendered text if the text is displayed by a `DrawText` call with X and Y positions of 0. For example, when this program is running on an iPhone 7 simulator, `TextSize` is assigned the value 90.6254 as a result of the calculation following the first call to `MeasureText`. The `SKRect` value obtained from the second call to `MeasureText` has the following property values:

- `Left` = 6
- `Top` = -68
- `Width` = 664.8214
- `Height` = 88;

Keep in mind that the X and Y coordinates you pass to the `DrawText` method specify the left side of the text at the baseline. The `Top` value indicates that the text extends 68 pixels above that baseline and (subtracting 68 from 88) 20 pixels below the baseline. The `Left` value of 6 indicates that the text begins six pixels to the right of the X value in the `DrawText` call. This allows for normal inter-character spacing. If you want to display the text snugly in the upper-left corner of the display, pass the negatives of these `Left` and `Top` values as the X and Y coordinates of `DrawText`, in this example, -6 and 68.

The `SKRect` structure defines several handy properties and methods, some of which are used in the remainder of the `PaintSurface` handler. The `MidX` and `MidY` values indicate the coordinates of the center of the rectangle. (In the iPhone 7 example, those values are 338.4107 and -24.) The following code uses these values for the easiest calculation of coordinates to center text on the display:

```
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    ...
    // Calculate offsets to center the text on the screen
    float xText = info.Width / 2 - textBounds.MidX;
    float yText = info.Height / 2 - textBounds.MidY;

    // And draw the text
    canvas.DrawText(str, xText, yText, textPaint);
    ...
}
```

The `SKImageInfo` `info` structure also defines a `Rect` property of type `SKRect`, so you can also calculate `xText` and `yText` like this:

```
float xText = info.Rect.MidX - textBounds.MidX;
float yText = info.Rect.MidY - textBounds.MidY;
```

The `PaintSurface` handler concludes with two calls to `DrawRoundRect`, both of which require arguments of `SKRect`. This `SKRect` value is based on the `SKRect` value obtained from the `MeasureText` method, but it can't be the same. First, it needs to be a little larger so that the rounded rectangle doesn't draw over edges of the text. Secondly, it needs to be shifted in space so that the `Left` and `Top` values correspond to the upper-left corner where the rectangle is to be positioned. These two jobs are accomplished by the `Offset` and `Inflate` methods defined by `SKRect`:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    ...
    // Create a new SKRect object for the frame around the text
    SKRect frameRect = textBounds;
    frameRect.Offset(xText, yText);
    frameRect.Inflate(10, 10);

    // Create an SKPaint object to display the frame
    SKPaint framePaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        StrokeWidth = 5,
        Color = SKColors.Blue
    };

    // Draw one frame
    canvas.DrawRoundRect(frameRect, 20, 20, framePaint);

    // Inflate the frameRect and draw another
    frameRect.Inflate(10, 10);
    framePaint.Color = SKColors.DarkBlue;
    canvas.DrawRoundRect(frameRect, 30, 30, framePaint);
}

```

Following that, the remainder of the method is straight-forward. It creates another `SKPaint` object for the borders and calls `DrawRoundRect` twice. The second call uses a rectangle inflated by another 10 pixels. The first call specifies a corner radius of 20 pixels. The second has a corner radius of 30 pixels, so they seem to be parallel:



You can turn your phone or simulator sideways to see the text and frame increase in size.

If you only need to center some text on the screen, you can do it approximately without measuring the text. Instead, set the `.TextAlign` property of `SKPaint` to the enumeration member `SKTextAlign.Center`. The X coordinate you specify in the `DrawText` method then indicates where the horizontal center of the text is positioned. If you pass the midpoint of the screen to the `DrawText` method, the text will be horizontally centered and *nearly* vertically centered because the baseline will be vertically centered.

Text can be treated much like any other graphical object. One simple option is to display the outline of the text characters:



This is accomplished simply by changing the normal `Style` property of the `SKPaint` object from its default setting of `SKPaintStyle.Fill` to `SKPaintStyle.Stroke`, and by specifying a stroke width. The `PaintSurface` handler of the **Outlined Text** page shows how it's done:

```
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    string text = "OUTLINE";

    // Create an SKPaint object to display the text
    SKPaint textPaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        StrokeWidth = 1,
       FakeBoldText = true,
        Color = SKColors.Blue
    };

    // Adjust TextSize property so text is 95% of screen width
    float textWidth = textPaint.MeasureText(text);
    textPaint.TextSize = 0.95f * info.Width * textPaint.TextSize / textWidth;

    // Find the text bounds
    SKRect textBounds = new SKRect();
    textPaint.MeasureText(text, ref textBounds);

    // Calculate offsets to center the text on the screen
    float xText = info.Width / 2 - textBounds.MidX;
    float yText = info.Height / 2 - textBounds.MidY;

    // And draw the text
    canvas.DrawText(text, xText, yText, textPaint);
}
```

Another common graphical object is the bitmap. That's a large topic covered in depth in the section [SkiaSharp Bitmaps](#), but the next article, [Bitmap Basics in SkiaSharp](#), provides a briefer introduction.

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

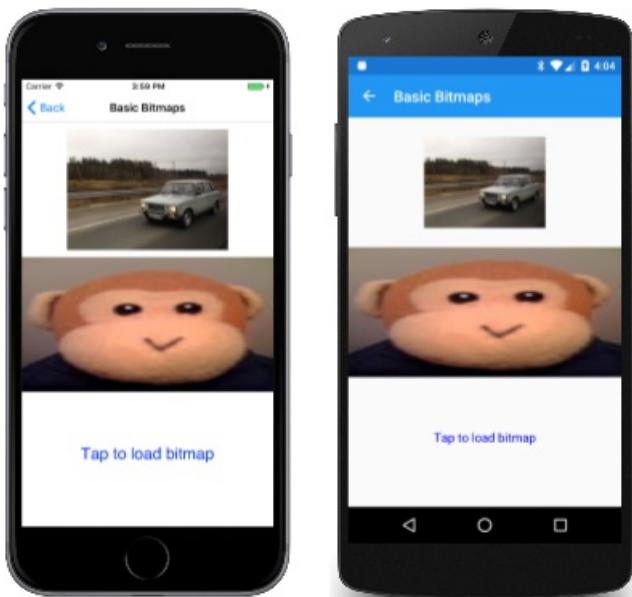
Bitmap Basics in SkiaSharp

3/5/2021 • 6 minutes to read • [Edit Online](#)

 [Download the sample](#)

Load bitmaps from various sources and display them.

The support of bitmaps in SkiaSharp is quite extensive. This article covers only the basics — how to load bitmaps and how to display them:



A much deeper exploration of bitmaps can be found in the section [SkiaSharp Bitmaps](#).

A SkiaSharp bitmap is an object of type `SKBitmap`. There are many ways to create a bitmap but this article restricts itself to the `SKBitmap.Decode` method, which loads the bitmap from a .NET `Stream` object.

The **Basic Bitmaps** page in the **SkiaSharpFormsDemos** program demonstrates how to load bitmaps from three different sources:

- From over the Internet
- From a resource embedded in the executable
- From the user's photo library

Three `SKBitmap` objects for these three sources are defined as fields in the `BasicBitmapsPage` class:

```

public class BasicBitmapsPage : ContentPage
{
    SKCanvasView canvasView;
    SKBitmap webBitmap;
    SKBitmap resourceBitmap;
    SKBitmap libraryBitmap;

    public BasicBitmapsPage()
    {
        Title = "Basic Bitmaps";

        canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
        ...
    }
    ...
}

```

Loading a Bitmap from the Web

To load a bitmap based on a URL, you can use the `HttpClient` class. You should instantiate only one instance of `HttpClient` and reuse it, so store it as a field:

```
HttpClient httpClient = new HttpClient();
```

When using `HttpClient` with iOS and Android applications, you'll want to set project properties as described in the documents on [Transport Layer Security \(TLS\) 1.2](#).

Because it's most convenient to use the `await` operator with `HttpClient`, the code can't be executed in the `BasicBitmapsPage` constructor. Instead, it's part of the `OnAppearing` override. The URL here points to an area on the Xamarin web site with some sample bitmaps. A package on the web site allows appending a specification for resizing the bitmap to a particular width:

```

protected override async void OnAppearing()
{
    base.OnAppearing();

    // Load web bitmap.
    string url = "https://developer.xamarin.com/demo/IMG_3256.JPG?width=480";

    try
    {
        using (Stream stream = await httpClient.GetStreamAsync(url))
        using (MemoryStream memStream = new MemoryStream())
        {
            await stream.CopyToAsync(memStream);
            memStream.Seek(0, SeekOrigin.Begin);

            webBitmap = SKBitmap.Decode(memStream);
            canvasView.InvalidateSurface();
        };
    }
    catch
    {
    }
}

```

The Android operating system raises an exception when using the `Stream` returned from `GetStreamAsync` in the

`SKBitmap.Decode` method because it's performing a lengthy operation on a main thread. For this reason, the contents of the bitmap file are copied to a `MemoryStream` object using `CopyToAsync`.

The static `SKBitmap.Decode` method is responsible for decoding bitmap files. It works with JPEG, PNG, and GIF bitmap formats, and stores the results in an internal SkiaSharp format. At this point, the `SKCanvasView` needs to be invalidated to allow the `PaintSurface` handler to update the display.

Loading a Bitmap Resource

In terms of code, the easiest approach to loading bitmaps is including a bitmap resource directly in your application. The **SkiaSharpFormsDemos** program includes a folder named **Media** containing several bitmap files, including one named **monkey.png**. For bitmaps stored as program resources, you must use the **Properties** dialog to give the file a **Build Action** of **Embedded Resource**!

Each embedded resource has a *resource ID* that consists of the project name, the folder, and the filename, all connected by periods: **SkiaSharpFormsDemos.Media.monkey.png**. You can get access to this resource by specifying that resource ID as an argument to the `GetManifestResourceStream` method of the `Assembly` class:

```
string resourceId = "SkiaSharpFormsDemos.Media.monkey.png";
Assembly assembly = GetType().GetTypeInfo().Assembly;

using (Stream stream = assembly.GetManifestResourceStream(resourceId))
{
    resourceBitmap = SKBitmap.Decode(stream);
}
```

This `stream` object can be passed directly to the `SKBitmap.Decode` method.

Loading a Bitmap from the Photo Library

It's also possible for the user to load a photo from the device's picture library. This facility is not provided by Xamarin.Forms itself. The job requires a dependency service, such as the one described in the article [Picking a Photo from the Picture Library](#).

The `IPhotoLibrary.cs` file in the **SkiaSharpFormsDemos** project and the three `PhotoLibrary.cs` files in the platform projects have been adapted from that article. In addition, the Android `MainActivity.cs` file has been modified as described in the article, and the iOS project has been given permission to access the photo library with two lines towards the bottom of the `info.plist` file.

The `BasicBitmapsPage` constructor adds a `TapGestureRecognizer` to the `SKCanvasView` to be notified of taps. On receipt of a tap, the `Tapped` handler gets access to the picture-picker dependency service and calls `PickPhotoAsync`. If a `Stream` object is returned, then it is passed to the `SKBitmap.Decode` method:

```

// Add tap gesture recognizer
TapGestureRecognizer tapRecognizer = new TapGestureRecognizer();
tapRecognizer.Tapped += async (sender, args) =>
{
    // Load bitmap from photo library
    IPhotoLibrary photoLibrary = DependencyService.Get<IPhotoLibrary>();

    using (Stream stream = await photoLibrary.PickPhotoAsync())
    {
        if (stream != null)
        {
            libraryBitmap = SKBitmap.Decode(stream);
            canvasView.InvalidateSurface();
        }
    }
};

canvasView.GestureRecognizers.Add(tapRecognizer);

```

Notice that the `Tapped` handler also calls the `InvalidateSurface` method of the `SKCanvasView` object. This generates a new call to the `PaintSurface` handler.

Displaying the Bitmaps

The `PaintSurface` handler needs to display three bitmaps. The handler assumes that the phone is in portrait mode and divides the canvas vertically into three equal parts.

The first bitmap is displayed with the simplest `DrawBitmap` method. All you need to specify are the X and Y coordinates where the upper-left corner of the bitmap is to be positioned:

```
public void DrawBitmap (SKBitmap bitmap, Single x, Single y, SKPaint paint = null)
```

Although an `SKPaint` parameter is defined, it has a default value of `null` and you can ignore it. The pixels of the bitmap are simply transferred to the pixels of the display surface with a one-to-one mapping. You'll see an application for this `SKPaint` argument in the next section on [SkiaSharp Transparency](#).

A program can obtain the pixel dimensions of a bitmap with the `Width` and `Height` properties. These properties allow the program to calculate coordinates to position the bitmap in the center of the upper-third of the canvas:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    if (webBitmap != null)
    {
        float x = (info.Width - webBitmap.Width) / 2;
        float y = (info.Height / 3 - webBitmap.Height) / 2;
        canvas.DrawBitmap(webBitmap, x, y);
    }
    ...
}

```

The other two bitmaps are displayed with a version of `DrawBitmap` with an `SKRect` parameter:

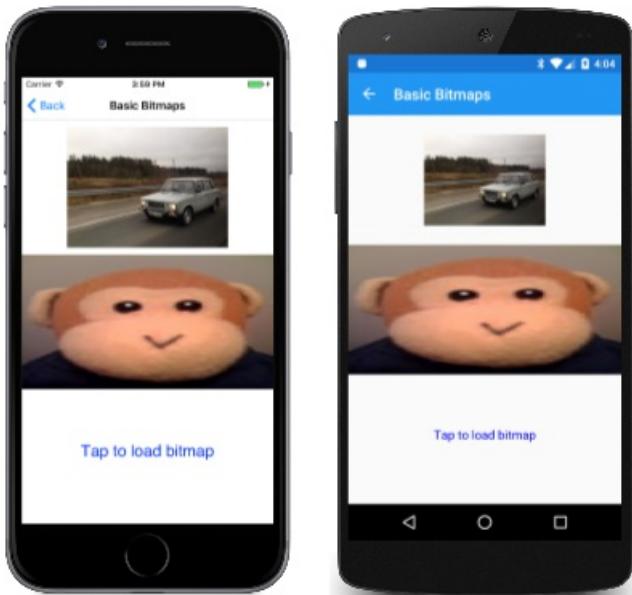
```
public void DrawBitmap (SKBitmap bitmap, SKRect dest, SKPaint paint = null)
```

A third version of `DrawBitmap` has two `SKRect` arguments for specifying a rectangular subset of the bitmap to display, but that version isn't used in this article.

Here's the code to display the bitmap loaded from an embedded resource bitmap:

```
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    ...
    if (resourceBitmap != null)
    {
        canvas.DrawBitmap(resourceBitmap,
            new SKRect(0, info.Height / 3, info.Width, 2 * info.Height / 3));
    }
    ...
}
```

The bitmap is stretched to the dimensions of the rectangle, which is why the monkey is horizontally stretched in these screenshots:



The third image — which you can only see if you run the program and load a photo from your own picture library — is also displayed within a rectangle, but the rectangle's position and size are adjusted to maintain the bitmap's aspect ratio. This calculation is a little more involved because it requires calculating a scaling factor based on the size of the bitmap and the destination rectangle, and centering the rectangle in that area:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    ...
    if (libraryBitmap != null)
    {
        float scale = Math.Min((float)info.Width / libraryBitmap.Width,
                               info.Height / 3f / libraryBitmap.Height);

        float left = (info.Width - scale * libraryBitmap.Width) / 2;
        float top = (info.Height / 3 - scale * libraryBitmap.Height) / 2;
        float right = left + scale * libraryBitmap.Width;
        float bottom = top + scale * libraryBitmap.Height;
        SKRect rect = new SKRect(left, top, right, bottom);
        rect.Offset(0, 2 * info.Height / 3);

        canvas.DrawBitmap(libraryBitmap, rect);
    }
    else
    {
        using (SKPaint paint = new SKPaint())
        {
            paint.Color = SKColors.Blue;
            paint.TextAlign = SKTextAlign.Center;
            paint.TextSize = 48;

            canvas.DrawText("Tap to load bitmap",
                           info.Width / 2, 5 * info.Height / 6, paint);
        }
    }
}

```

If no bitmap has yet been loaded from the picture library, then the `else` block displays some text to prompt the user to tap the screen.

You can display bitmaps with various degrees of transparency, and the next article on [SkiaSharp Transparency](#) describes how.

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)
- [Picking a Photo from the Picture Library](#)

SkiaSharp transparency

3/5/2021 • 4 minutes to read • [Edit Online](#)



[Download the sample](#)

As you've seen, the `SKPaint` class includes a `color` property of type `SKColor`. `SKColor` includes an alpha channel, so anything that you color with an `SKColor` value can be partially transparent.

Some transparency was demonstrated in the [Basic Animation in SkiaSharp](#) article. This article goes somewhat deeper into transparency to combine multiple objects in a single scene, a technique sometimes known as *blending*. More advanced blending techniques are discussed in the articles in the [SkiaSharp shaders](#) section.

You can set the transparency level when you first create a color using the four-parameter `SKColor` constructor:

```
SKColor (byte red, byte green, byte blue, byte alpha);
```

An alpha value of 0 is fully transparent and an alpha value of 0xFF is fully opaque. Values between those two extremes create colors that are partially transparent.

In addition, `SKColor` defines a handy `WithAlpha` method that creates a new color from an existing color but with the specified alpha level:

```
SKColor halfTransparentBlue = SKColors.Blue.WithAlpha(0x80);
```

The use of partially transparent text is demonstrated in the [Code More Code](#) page in the [SkiaSharpFormsDemos](#) sample. This page fades two text strings in and out by incorporating transparency in the `SKColor` values:

```
public class CodeMoreCodePage : ContentPage
{
    SKCanvasView canvasView;
    bool isAnimating;
    Stopwatch stopwatch = new Stopwatch();
    double transparency;

    public CodeMoreCodePage ()
    {
        Title = "Code More Code";

        canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    protected override void OnAppearing()
    {
        base.OnAppearing();

        isAnimating = true;
        stopwatch.Start();
        Device.StartTimer(TimeSpan.FromMilliseconds(16), OnTimerTick);
    }

    protected override void OnDisappearing()
```

```

    {
        base.OnDisappearing();

        stopwatch.Stop();
        isAnimating = false;
    }

    bool OnTimerTick()
    {
        const int duration = 5;      // seconds
        double progress = stopwatch.Elapsed.TotalSeconds % duration / duration;
        transparency = 0.5 * (1 + Math.Sin(progress * 2 * Math.PI));
        canvasView.InvalidateSurface();

        return isAnimating;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        const string TEXT1 = "CODE";
        const string TEXT2 = "MORE";

        using (SKPaint paint = new SKPaint())
        {
            // Set text width to fit in width of canvas
            paint.TextSize = 100;
            float textWidth = paint.MeasureText(TEXT1);
            paint.TextSize *= 0.9f * info.Width / textWidth;

            // Center first text string
            SKRect textBounds = new SKRect();
            paint.MeasureText(TEXT1, ref textBounds);

            float xText = info.Width / 2 - textBounds.MidX;
            float yText = info.Height / 2 - textBounds.MidY;

            paint.Color = SKColors.Blue.WithAlpha((byte)(0xFF * (1 - transparency)));
            canvas.DrawText(TEXT1, xText, yText, paint);

            // Center second text string
            textBounds = new SKRect();
            paint.MeasureText(TEXT2, ref textBounds);

            xText = info.Width / 2 - textBounds.MidX;
            yText = info.Height / 2 - textBounds.MidY;

            paint.Color = SKColors.Blue.WithAlpha((byte)(0xFF * transparency));
            canvas.DrawText(TEXT2, xText, yText, paint);
        }
    }
}

```

The `transparency` field is animated to vary from 0 to 1 and back again in a sinusoidal rhythm. The first text string is displayed with an alpha value calculated by subtracting the `transparency` value from 1:

```
paint.Color = SKColors.Blue.WithAlpha((byte)(0xFF * (1 - transparency)));
```

The `WithAlpha` method sets the alpha component on an existing color, which here is `SKColors.Blue`. The second text string uses an alpha value calculated from the `transparency` value itself:

```
paint.Color = SKColors.Blue.WithAlpha((byte)(0xFF * transparency));
```

The animation alternates between the two words, urging the user to "code more" (or perhaps requesting "more code"):



In the previous article on [Bitmap Basics in SkiaSharp](#), you saw how to display bitmaps using one of the `DrawBitmap` methods of `SKCanvas`. All the `DrawBitmap` methods include an `SKPaint` object as the last parameter. By default, this parameter is set to `null` and you can ignore it.

Alternatively, you can set the `Color` property of this `SKPaint` object to display a bitmap with some level of transparency. Setting a level of transparency in the `Color` property of `SKPaint` allows you to fade bitmaps in and out, or to dissolve one bitmap into another.

Bitmap transparency is demonstrated in the [Bitmap Dissolve](#) page. The XAML file instantiates an `SKCanvasView` and a `Slider`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Effects.BitmapDissolvePage"
    Title="Bitmap Dissolve">
    <StackLayout>
        <skia:SKCanvasView x:Name="canvasView"
            VerticalOptions="FillAndExpand"
            PaintSurface="OnCanvasViewPaintSurface" />

        <Slider x:Name="progressSlider"
            Margin="10"
            ValueChanged="OnSliderValueChanged" />
    </StackLayout>
</ContentPage>
```

The code-behind file loads two bitmap resources. These bitmaps are not the same size, but they are the same aspect ratio:

```

public partial class BitmapDissolvePage : ContentPage
{
    SKBitmap bitmap1;
    SKBitmap bitmap2;

    public BitmapDissolvePage()
    {
        InitializeComponent();

        // Load two bitmaps
        Assembly assembly = GetType().GetTypeInfo().Assembly;

        using (Stream stream = assembly.GetManifestResourceStream(
            "SkiaSharpFormsDemos.Media.SeatedMonkey.jpg"))
        {
            bitmap1 = SKBitmap.Decode(stream);
        }
        using (Stream stream = assembly.GetManifestResourceStream(
            "SkiaSharpFormsDemos.Media.FacePalm.jpg"))
        {
            bitmap2 = SKBitmap.Decode(stream);
        }
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Find rectangle to fit bitmap
        float scale = Math.Min((float)info.Width / bitmap1.Width,
                               (float)info.Height / bitmap1.Height);
        SKRect rect = SKRect.Create(scale * bitmap1.Width,
                                    scale * bitmap1.Height);
        float x = (info.Width - rect.Width) / 2;
        float y = (info.Height - rect.Height) / 2;
        rect.Offset(x, y);

        // Get progress value from Slider
        float progress = (float)progressSlider.Value;

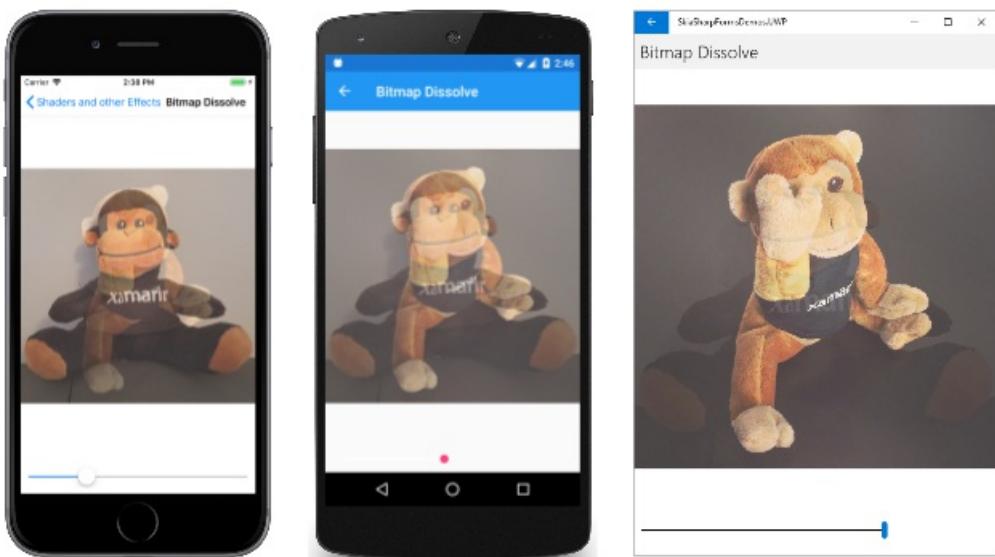
        // Display two bitmaps with transparency
        using (SKPaint paint = new SKPaint())
        {
            paint.Color = paint.Color.WithAlpha((byte)(0xFF * (1 - progress)));
            canvas.DrawBitmap(bitmap1, rect, paint);

            paint.Color = paint.Color.WithAlpha((byte)(0xFF * progress));
            canvas.DrawBitmap(bitmap2, rect, paint);
        }
    }
}

```

The `color` property of the `SKPaint` object is set to two complementary alpha levels for the two bitmaps. When using `SKPaint` with bitmaps, it doesn't matter what the rest of the `Color` value is. All that matters is the alpha channel. The code here simply calls the `WithAlpha` method on the default value of the `Color` property.

Moving the **Slider** dissolves between one bitmap and the other:



In the past several articles, you have seen how to use SkiaSharp to draw text, circles, ellipses, rounded rectangles, and bitmaps. The next step is [SkiaSharp Lines and Paths](#) in which you will learn how to draw connected lines in a graphics path.

Related links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

SkiaSharp Lines and Paths

3/5/2021 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

Use SkiaSharp to draw lines and graphics paths

The [previous section](#) demonstrated that the SkiaSharp `SKCanvas` class includes several methods to draw circles, ovals, rectangles, rounded rectangles, text, and bitmaps. This section and later sections cover the various classes connected with creating and rendering *graphics paths*.

The graphics path is the most generalized approach to drawing lines and curves in SkiaSharp. This section covers using an `SKPath` object to draw straight lines, and to use a collection of tiny straight lines (called a *polyline*) to draw curves that you can define algorithmically. A later section on [SkiaSharp Curves and Paths](#) discusses the various sorts of curves supported by `SKPath`.

All the sample programs in this section appear under the heading **Lines and Paths** in the home page of the [SkiaSharpFormsDemos](#) program, and in the **Paths** folder of that solution.

Lines and Stroke Caps

Learn how to use SkiaSharp to draw lines with different stroke caps.

Path Basics

Explore the SkiaSharp `SKPath` object for combining lines and curves.

The Path Fill Types

Discover the different effects possible with SkiaSharp path fill types.

Polylines and Parametric Equations

Use SkiaSharp to render any line you can define with parametric equations.

Dots and Dashes

Master the intricacies of drawing dotted and dashed lines in SkiaSharp.

Finger Painting

Use your fingers to paint on the canvas.

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Lines and Stroke Caps

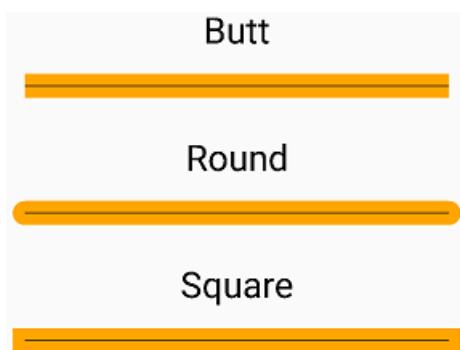
3/5/2021 • 5 minutes to read • [Edit Online](#)



[Download the sample](#)

Learn how to use SkiaSharp to draw lines with different stroke caps

In SkiaSharp, rendering a single line is very different from rendering a series of connected straight lines. Even when drawing single lines, however, it's often necessary to give the lines a particular stroke width. As these lines become wider, the appearance of the ends of the lines also becomes important. The appearance of the end of the line is called the *stroke cap*.



For drawing single lines, `SKCanvas` defines a simple `DrawLine` method whose arguments indicate the starting and ending coordinates of the line with an `SKPaint` object:

```
canvas.DrawLine (x0, y0, x1, y1, paint);
```

By default, the `StrokeWidth` property of a newly instantiated `SKPaint` object is 0, which has the same effect as a value of 1 in rendering a line of one pixel in thickness. This appears very thin on high-resolution devices such as phones, so you'll probably want to set the `StrokeWidth` to a larger value. But once you start drawing lines of a sizable thickness, that raises another issue: How should the starts and ends of these thick lines be rendered?

The appearance of the starts and ends of lines is called a *line cap* or, in Skia, a *stroke cap*. The word "cap" in this context refers to a kind of hat — something that sits on the end of the line. You set the `StrokeCap` property of the `SKPaint` object to one of the following members of the `SKStrokeCap` enumeration:

- `Butt` (the default)
- `Square`
- `Round`

These are best illustrated with a sample program. The **SkiaSharp Lines and Paths** section of the **SkiaSharpFormsDemos** program begins with a page titled **Stroke Caps** based on the `StrokeCapsPage` class. This page defines a `PaintSurface` event handler that loops through the three members of the `SKStrokeCap` enumeration, displaying both the name of the enumeration member and drawing a line using that stroke cap:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    SKPaint textPaint = new SKPaint
    {
        Color = SKColors.Black,
        TextSize = 75,
        TextAlign = SKTextAlign.Center
    };

    SKPaint thickLinePaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Orange,
        StrokeWidth = 50
    };

    SKPaint thinLinePaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Black,
        StrokeWidth = 2
    };

    float xText = info.Width / 2;
    float xLine1 = 100;
    float xLine2 = info.Width - xLine1;
    float y = textPaint.FontSpacing;

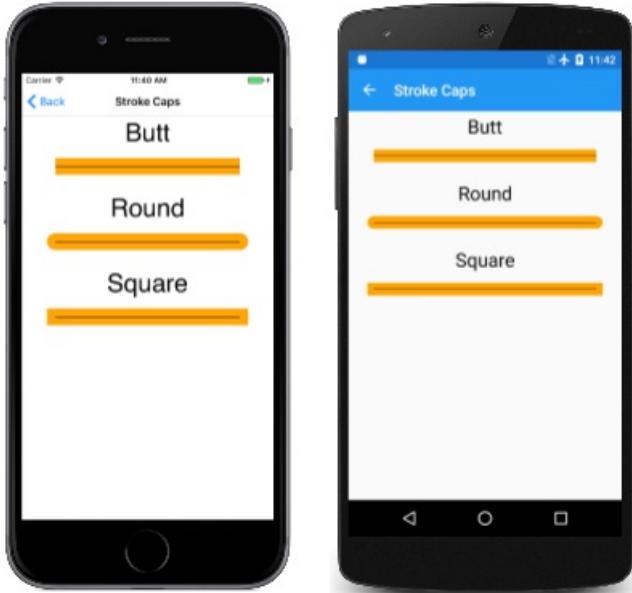
    foreach (SKStrokeCap strokeCap in Enum.GetValues(typeof(SKStrokeCap)))
    {
        // Display text
        canvas.DrawText(strokeCap.ToString(), xText, y, textPaint);
        y += textPaint.FontSpacing;

        // Display thick line
        thickLinePaint.StrokeCap = strokeCap;
        canvas.DrawLine(xLine1, y, xLine2, y, thickLinePaint);

        // Display thin line
        canvas.DrawLine(xLine1, y, xLine2, y, thinLinePaint);
        y += 2 * textPaint.FontSpacing;
    }
}

```

For each member of the `SKStrokeCap` enumeration, the handler draws two lines, one with a stroke thickness of 50 pixels and another line positioned on top with a stroke thickness of two pixels. This second line is intended to illustrate the geometric start and end of the line independent of the line thickness and a stroke cap:



As you can see, the `Square` and `Round` stroke caps effectively extend the length of the line by half the stroke width at the beginning of the line and again at the end. This extension becomes important when it's necessary to determine the dimensions of a rendered graphics object.

The `SKCanvas` class also includes another method for drawing multiple lines that is somewhat peculiar:

```
DrawPoints (SKPointMode mode, points, paint)
```

The `points` parameter is an array of `SKPoint` values and `mode` is a member of the `SKPointMode` enumeration, which has three members:

- `Points` to render the individual points
- `Lines` to connect each pair of points
- `Polygon` to connect all consecutive points

The [Multiple Lines](#) page demonstrates this method. The `MultipleLinesPage.xaml` file instantiates two `Picker` views that let you select a member of the `SKPointMode` enumeration and a member of the `SKStrokeCap` enumeration:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp;assembly=SkiaSharp"
    xmlns:skiaforms="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Paths.MultipleLinesPage"
    Title="Multiple Lines">

    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Picker x:Name="pointModePicker"
            Title="Point Mode"
            Grid.Row="0"
            Grid.Column="0"
            SelectedIndexChanged="OnPickerSelectedIndexChanged">
            <Picker.ItemsSource>
                <x:Array Type="{x:Type skia:SKPointMode}">
                    <x:Static Member="skia:SKPointMode.Points" />
                    <x:Static Member="skia:SKPointMode.Lines" />
                    <x:Static Member="skia:SKPointMode.Polygon" />
                </x:Array>
            </Picker.ItemsSource>
            <Picker.SelectedIndex>
                0
            </Picker.SelectedIndex>
        </Picker>

        <Picker x:Name="strokeCapPicker"
            Title="Stroke Cap"
            Grid.Row="0"
            Grid.Column="1"
            SelectedIndexChanged="OnPickerSelectedIndexChanged">
            <Picker.ItemsSource>
                <x:Array Type="{x:Type skia:SKStrokeCap}">
                    <x:Static Member="skia:SKStrokeCap.Butt" />
                    <x:Static Member="skia:SKStrokeCap.Round" />
                    <x:Static Member="skia:SKStrokeCap.Square" />
                </x:Array>
            </Picker.ItemsSource>
            <Picker.SelectedIndex>
                0
            </Picker.SelectedIndex>
        </Picker>

        <skiaforms:SKCanvasView x:Name="canvasView"
            PaintSurface="OnCanvasViewPaintSurface"
            Grid.Row="1"
            Grid.Column="0"
            Grid.ColumnSpan="2" />
    </Grid>
</ContentPage>

```

Notice that the SkiaSharp namespace declarations are a little different because the `SkiaSharp` namespace is needed to reference the members of the `SKPointMode` and `SKStrokeCap` enumerations. The `SelectedIndexChanged` handler for both `Picker` views simply invalidates the `SKCanvasView` object:

```

void OnPickerSelectedIndexChanged(object sender, EventArgs args)
{
    if (canvasView != null)
    {
        canvasView.InvalidateSurface();
    }
}

```

This handler needs to check for the existence of the `SKCanvasView` object because the event handler is first called when the `SelectedIndex` property of the `Picker` is set to 0 in the XAML file, and that occurs before the `SKCanvasView` has been instantiated.

The `PaintSurface` handler obtains the two enumeration values from the `Picker` views:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    // Create an array of points scattered through the page
    SKPoint[] points = new SKPoint[10];

    for (int i = 0; i < 2; i++)
    {
        float x = (0.1f + 0.8f * i) * info.Width;

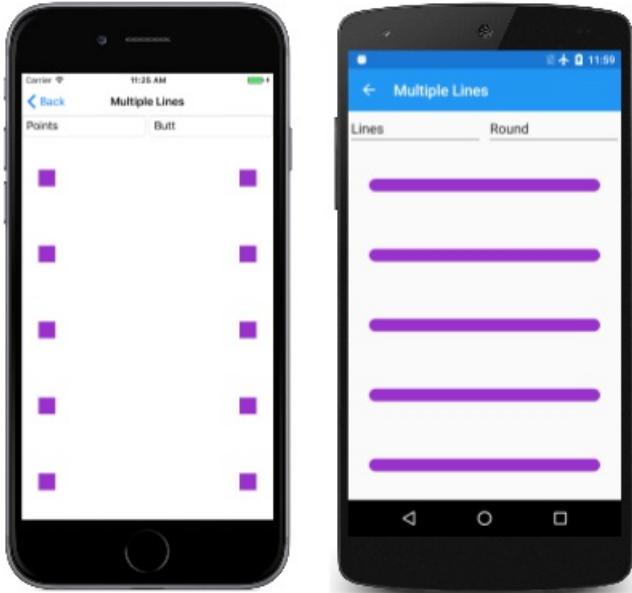
        for (int j = 0; j < 5; j++)
        {
            float y = (0.1f + 0.2f * j) * info.Height;
            points[2 * j + i] = new SKPoint(x, y);
        }
    }

    SKPaint paint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.DarkOrchid,
        StrokeWidth = 50,
        StrokeCap = (SKStrokeCap)strokeCapPicker.SelectedItem
    };

    // Render the points by calling DrawPoints
    SKPointMode pointMode = (SKPointMode)pointModePicker.SelectedItem;
    canvas.DrawPoints(pointMode, points, paint);
}

```

The screenshots show a variety of `Picker` selections:



The iPhone at the left shows how the `SKPointMode.Points` enumeration member causes `DrawPoints` to render each of the points in the `SKPoint` array as a square if the line cap is `Butt` or `Square`. Circles are rendered if the line cap is `Round`.

The Android screenshot shows the result of the `SKPointMode.Lines`. The `DrawPoints` method draws a line between each pair of `SKPoint` values, using the specified line cap, in this case `Round`.

When you instead use `SKPointMode.Polygon`, a line is drawn between the successive points in the array, but if you look very closely, you'll see that these lines are not connected. Each of these separate lines starts and ends with the specified line cap. If you select the `Round` caps, the lines might appear to be connected, but they're really not connected.

Whether lines are connected or not connected is a crucial aspect of working with graphics paths.

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Path Basics in SkiaSharp

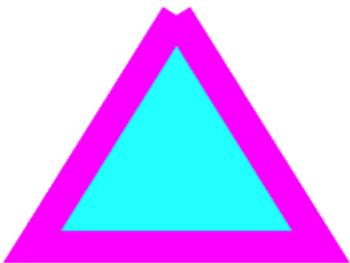
3/5/2021 • 6 minutes to read • [Edit Online](#)



[Download the sample](#)

Explore the SkiaSharp `SKPath` object for combining connected lines and curves

One of the most important features of the graphics path is the ability to define when multiple lines should be connected and when they should not be connected. The difference can be significant, as the tops of these two triangles demonstrate:



A graphics path is encapsulated by the `SKPath` object. A path is a collection of one or more *contours*. Each contour is a collection of *connected* straight lines and curves. Contours are not connected to each other but they might visually overlap. Sometimes a single contour can overlap itself.

A contour generally begins with a call to the following method of `SKPath`:

- `MoveTo` to begin a new contour

The argument to that method is a single point, which you can express either as an `SKPoint` value or as separate X and Y coordinates. The `MoveTo` call establishes a point at the beginning of the contour and an initial *current point*. You can call the following methods to continue the contour with a line or curve from the current point to a point specified in the method, which then becomes the new current point:

- `LineTo` to add a straight line to the path
- `ArcTo` to add an arc, which is a line on the circumference of a circle or ellipse
- `CubicTo` to add a cubic Bezier spline
- `QuadTo` to add a quadratic Bezier spline
- `ConicTo` to add a rational quadratic Bezier spline, which can accurately render conic sections (ellipses, parabolas, and hyperbolas)

None of these five methods contain all the information necessary to describe the line or curve. Each of these five methods works in conjunction with the current point established by the method call immediately preceding it.

For example, the `LineTo` method adds a straight line to the contour based on the current point, so the parameter to `LineTo` is only a single point.

The `SKPath` class also defines methods that have the same names as these six methods but with an `R` at the beginning:

- `RMoveTo`
- `RLineTo`
- `RArcTo`
- `RCubicTo`
- `RQuadTo`
- `RConicTo`

The `R` stands for *relative*. These methods have the same syntax as the corresponding methods without the `R` but are relative to the current point. These are handy for drawing similar parts of a path in a method that you call multiple times.

A contour ends with another call to `MoveTo` or `RMoveTo`, which begins a new contour, or a call to `Close`, which closes the contour. The `close` method automatically appends a straight line from the current point to the first point of the contour, and marks the path as closed, which means that it will be rendered without any stroke caps.

The difference between open and closed contours is illustrated in the [Two Triangle Contours](#) page, which uses an `SKPath` object with two contours to render two triangles. The first contour is open and the second is closed. Here's the [TwoTriangleContoursPage](#) class:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    // Create the path
    SKPath path = new SKPath();

    // Define the first contour
    path.MoveTo(0.5f * info.Width, 0.1f * info.Height);
    path.LineTo(0.2f * info.Width, 0.4f * info.Height);
    path.LineTo(0.8f * info.Width, 0.4f * info.Height);
    path.LineTo(0.5f * info.Width, 0.1f * info.Height);

    // Define the second contour
    path.MoveTo(0.5f * info.Width, 0.6f * info.Height);
    path.LineTo(0.2f * info.Width, 0.9f * info.Height);
    path.LineTo(0.8f * info.Width, 0.9f * info.Height);
    path.Close();

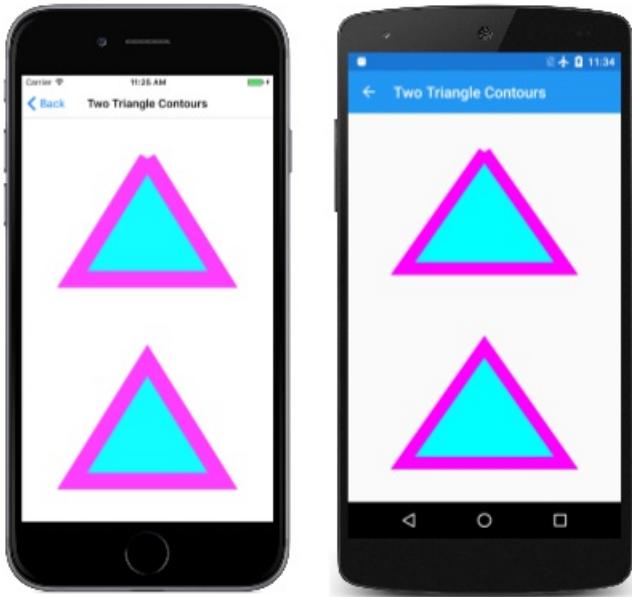
    // Create two SKPaint objects
    SKPaint strokePaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Magenta,
        StrokeWidth = 50
    };

    SKPaint fillPaint = new SKPaint
    {
        Style = SKPaintStyle.Fill,
        Color = SKColors.Cyan
    };

    // Fill and stroke the path
    canvas.DrawPath(path, fillPaint);
    canvas.DrawPath(path, strokePaint);
}

```

The first contour consists of a call to `MoveTo` using X and Y coordinates rather than an `SKPoint` value, followed by three calls to `LineTo` to draw the three sides of the triangle. The second contour has only two calls to `LineTo` but it finishes the contour with a call to `Close`, which closes the contour. The difference is significant:



As you can see, the first contour is obviously a series of three connected lines, but the end doesn't connect with the beginning. The two lines overlap at the top. The second contour is obviously closed, and was accomplished with one fewer `LineTo` calls because the `close` method automatically adds a final line to close the contour.

`SKCanvas` defines only one `DrawPath` method, which in this demonstration is called twice to fill and stroke the path. All contours are filled, even those that are not closed. For purposes of filling unclosed paths, a straight line is assumed to exist between the start and end points of the contours. If you remove the last `LineTo` from the first contour, or remove the `close` call from the second contour, each contour will have only two sides but will be filled as if it were a triangle.

`SKPath` defines many other methods and properties. The following methods add entire contours to the path, which might be closed or not closed depending on the method:

- `AddRect`
- `AddRoundedRect`
- `AddCircle`
- `AddOval`
- `AddArc` to add a curve on the circumference of an ellipse
- `AddPath` to add another path to the current path
- `AddPathReverse` to add another path in reverse

Keep in mind that an `SKPath` object defines only a geometry — a series of points and connections. Only when an `SKPath` is combined with an `SKPaint` object is the path rendered with a particular color, stroke width, and so forth. Also, keep in mind that the `SKPaint` object passed to the `DrawPath` method defines characteristics of the entire path. If you want to draw something requiring several colors, you must use a separate path for each color.

Just as the appearance of the start and end of a line is defined by a stroke cap, the appearance of the connection between two lines is defined by a *stroke join*. You specify this by setting the `StrokeJoin` property of `SKPaint` to a member of the `SKStrokeJoin` enumeration:

- `Miter` for a pointy join
- `Round` for a rounded join
- `Bevel` for a chopped-off join

The [Stroke Joins](#) page shows these three stroke joins with code similar to the [Stroke Caps](#) page. This is the `PaintSurface` event handler in the `StrokeJoinsPage` class:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    SKPaint textPaint = new SKPaint
    {
        Color = SKColors.Black,
        TextSize = 75,
        TextAlign = SKTextAlign.Right
    };

    SKPaint thickLinePaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Orange,
        StrokeWidth = 50
    };

    SKPaint thinLinePaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Black,
        StrokeWidth = 2
    };

    float xText = info.Width - 100;
    float xLine1 = 100;
    float xLine2 = info.Width - xLine1;
    float y = 2 * textPaint.FontSpacing;
    string[] strStrokeJoins = { "Miter", "Round", "Bevel" };

    foreach (string strStrokeJoin in strStrokeJoins)
    {
        // Display text
        canvas.DrawText(strStrokeJoin, xText, y, textPaint);

        // Get stroke-join value
        SKStrokeJoin strokeJoin;
        Enum.TryParse(strStrokeJoin, out strokeJoin);

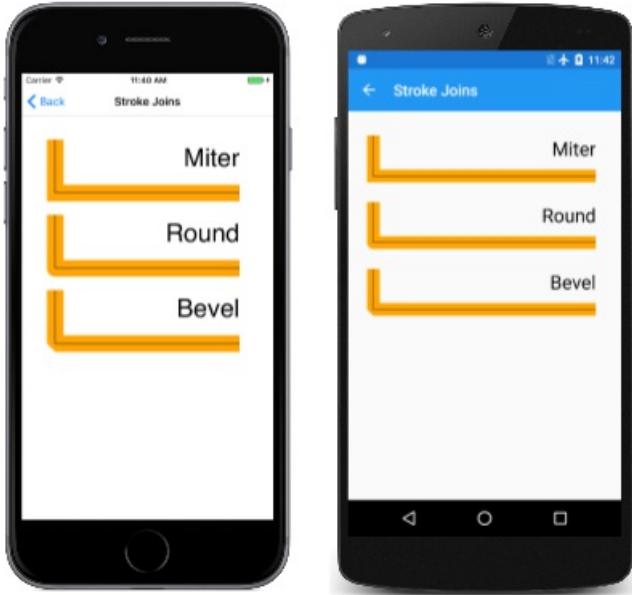
        // Create path
        SKPath path = new SKPath();
        path.MoveTo(xLine1, y - 80);
        path.LineTo(xLine1, y + 80);
        path.LineTo(xLine2, y + 80);

        // Display thick line
        thickLinePaint.StrokeJoin = strokeJoin;
        canvas.DrawPath(path, thickLinePaint);

        // Display thin line
        canvas.DrawPath(path, thinLinePaint);
        y += 3 * textPaint.FontSpacing;
    }
}

```

Here's the program running:



The miter join consists of a sharp point where the lines connect. When two lines join at a small angle, the miter join can become quite long. To prevent excessively long miter joins, the length of the miter join is limited by the value of the `StrokeMiter` property of `SKPaint`. A miter join that exceeds this length is chopped off to become a bevel join.

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

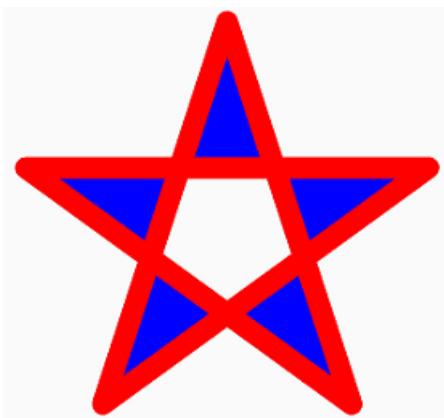
The Path Fill Types

3/5/2021 • 4 minutes to read • [Edit Online](#)

 [Download the sample](#)

Discover the different effects possible with SkiaSharp path fill types

Two contours in a path can overlap, and the lines that make up a single contour can overlap. Any enclosed area can potentially be filled, but you might not want to fill all the enclosed areas. Here's an example:



You have a little control over this. The filling algorithm is governed by the `SKFillType` property of `SKPath`, which you set to a member of the `SKPathFillType` enumeration:

- `Winding`, the default
- `EvenOdd`
- `InverseWinding`
- `InverseEvenOdd`

Both the winding and even-odd algorithms determine if any enclosed area is filled or not filled based on a hypothetical line drawn from that area to infinity. That line crosses one or more boundary lines that make up the path. With the winding mode, if the number of boundary lines drawn in one direction balance out the number of lines drawn in the other direction, then the area is not filled. Otherwise the area is filled. The even-odd algorithm fills an area if the number of boundary lines is odd.

With many routine paths, the winding algorithm often fills all the enclosed areas of a path. The even-odd algorithm generally produces more interesting results.

The classic example is a five-pointed star, as demonstrated in the [Five-Pointed Star](#) page. The `FivePointedStarPage.xaml` file instantiates two `Picker` views to select the path fill type and whether the path is stroked or filled or both, and in what order:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp;assembly=SkiaSharp"
    xmlns:skiaforms="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Paths.FivePointedStarPage"
    Title="Five-Pointed Star">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <Picker x:Name="fillTypePicker"
            Title="Path Fill Type"
            Grid.Row="0"
            Grid.Column="0"
            Margin="10"
            SelectedIndexChanged="OnPickerSelectedIndexChanged">
            <Picker.ItemsSource>
                <x:Array Type="{x:Type skia:SKPathFillType}">
                    <x:Static Member="skia:SKPathFillType.Winding" />
                    <x:Static Member="skia:SKPathFillType.EvenOdd" />
                    <x:Static Member="skia:SKPathFillType.InverseWinding" />
                    <x:Static Member="skia:SKPathFillType.InverseEvenOdd" />
                </x:Array>
            </Picker.ItemsSource>
            <Picker.SelectedIndex>
                0
            </Picker.SelectedIndex>
        </Picker>

        <Picker x:Name="drawingModePicker"
            Title="Drawing Mode"
            Grid.Row="0"
            Grid.Column="1"
            Margin="10"
            SelectedIndexChanged="OnPickerSelectedIndexChanged">
            <Picker.ItemsSource>
                <x:Array Type="{x:Type x:String}">
                    <x:String>Fill only</x:String>
                    <x:String>Stroke only</x:String>
                    <x:String>Stroke then Fill</x:String>
                    <x:String>Fill then Stroke</x:String>
                </x:Array>
            </Picker.ItemsSource>
            <Picker.SelectedIndex>
                0
            </Picker.SelectedIndex>
        </Picker>

        <skiaforms:SKCanvasView x:Name="canvasView"
            Grid.Row="1"
            Grid.Column="0"
            Grid.ColumnSpan="2"
            PaintSurface="OnCanvasViewPaintSurface" />
    </Grid>
</ContentPage>

```

The code-behind file uses both `Picker` values to draw a five-pointed star:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    SKPoint center = new SKPoint(info.Width / 2, info.Height / 2);
    float radius = 0.45f * Math.Min(info.Width, info.Height);

    SKPath path = new SKPath
    {
        FillType = (SKPathFillType)fillTypePicker.SelectedItem
    };
    path.MoveTo(info.Width / 2, info.Height / 2 - radius);

    for (int i = 1; i < 5; i++)
    {
        // angle from vertical
        double angle = i * 4 * Math.PI / 5;
        path.LineTo(center + new SKPoint(radius * (float)Math.Sin(angle),
                                         -radius * (float)Math.Cos(angle)));
    }
    path.Close();

    SKPaint strokePaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Red,
        StrokeWidth = 50,
        StrokeJoin = SKStrokeJoin.Round
    };

    SKPaint fillPaint = new SKPaint
    {
        Style = SKPaintStyle.Fill,
        Color = SKColors.Blue
    };

    switch ((string)drawingModePicker.SelectedItem)
    {
        case "Fill only":
            canvas.DrawPath(path, fillPaint);
            break;

        case "Stroke only":
            canvas.DrawPath(path, strokePaint);
            break;

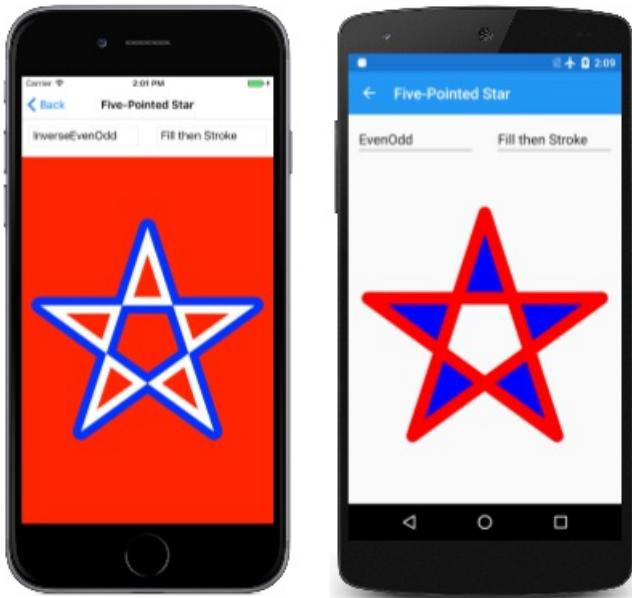
        case "Stroke then Fill":
            canvas.DrawPath(path, strokePaint);
            canvas.DrawPath(path, fillPaint);
            break;

        case "Fill then Stroke":
            canvas.DrawPath(path, fillPaint);
            canvas.DrawPath(path, strokePaint);
            break;
    }
}

```

Normally, the path fill type should affect only fills and not strokes, but the two `Inverse` modes affect both fills and strokes. For fills, the two `Inverse` types fill areas oppositely so that the area outside the star is filled. For strokes, the two `Inverse` types color everything except the stroke. Using these inverse fill types can produce

some odd effects, as the iOS screenshot demonstrates:



The Android screenshot shows the typical even-odd and winding effects, but the order of the stroke and fill also affects the results.

The winding algorithm is dependent on the direction that lines are drawn. Usually when you're creating a path, you can control that direction as you specify that lines are drawn from one point to another. However, the `SKPath` class also defines methods like `AddRect` and `AddCircle` that draw entire contours. To control how these objects are drawn, the methods include a parameter of type `SKPathDirection`, which has two members:

- `Clockwise`
- `CounterClockwise`

The methods in `SKPath` that include an `SKPathDirection` parameter give it a default value of `Clockwise`.

The [Overlapping Circles](#) page creates a path with four overlapping circles with an even-odd path fill type:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    SKPoint center = new SKPoint(info.Width / 2, info.Height / 2);
    float radius = Math.Min(info.Width, info.Height) / 4;

    SKPath path = new SKPath
    {
        FillType = SKPathFillType.EvenOdd
    };

    path.AddCircle(center.X - radius / 2, center.Y - radius / 2, radius);
    path.AddCircle(center.X - radius / 2, center.Y + radius / 2, radius);
    path.AddCircle(center.X + radius / 2, center.Y - radius / 2, radius);
    path.AddCircle(center.X + radius / 2, center.Y + radius / 2, radius);

    SKPaint paint = new SKPaint()
    {
        Style = SKPaintStyle.Fill,
        Color = SKColors.Cyan
    };

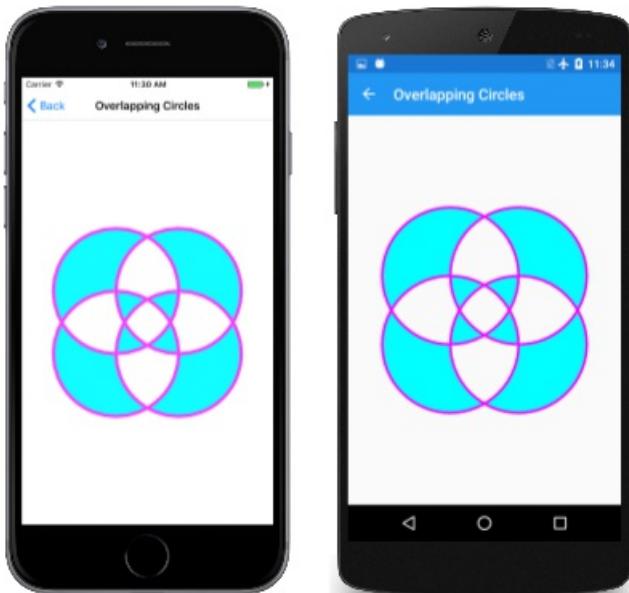
    canvas.DrawPath(path, paint);

    paint.Style = SKPaintStyle.Stroke;
    paint.StrokeWidth = 10;
    paint.Color = SKColors.Magenta;

    canvas.DrawPath(path, paint);
}

```

It's an interesting image created with a minimum of code:



Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Polyline and Parametric Equations

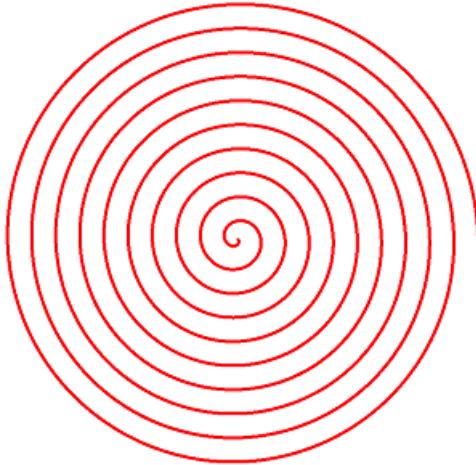
3/5/2021 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

Use SkiaSharp to render any line that you can define with parametric equations

In the [SkiaSharp Curves and Paths](#) section of this guide, you'll see the various methods that `SKPath` defines to render certain types of curves. However, it's sometimes necessary to draw a type of curve that isn't directly supported by `SKPath`. In such a case, you can use a polyline (a collection of connected lines) to draw any curve that you can mathematically define. If you make the lines small enough and numerous enough, the result will look like a curve. This spiral is actually 3,600 little lines:



Generally it's best to define a curve in terms of a pair of parametric equations. These are equations for X and Y coordinates that depend on a third variable, sometimes called `t` for time. For example, the following parametric equations define a circle with a radius of 1 centered at the point (0, 0) for `t` from 0 to 1:

```
x = cos(2πt)
```

```
y = sin(2πt)
```

If you want a radius larger than 1, you can simply multiply the sine and cosine values by that radius, and if you need to move the center to another location, add those values:

```
x = xCenter + radius·cos(2πt)
```

```
y = yCenter + radius·sin(2πt)
```

For an ellipse with the axes parallel to the horizontal and vertical, two radii are involved:

```
x = xCenter + xRadius·cos(2πt)
```

```
y = yCenter + yRadius·sin(2πt)
```

You can then put the equivalent SkiaSharp code in a loop that calculates the various points and adds those to a path. The following SkiaSharp code creates an `SKPath` object for an ellipse that fills the display surface. The loop cycles through the 360 degrees directly. The center is half the width and height of the display surface, and so are the two radii:

```
SKPath path = new SKPath();

for (float angle = 0; angle < 360; angle += 1)
{
    double radians = Math.PI * angle / 180;
    float x = info.Width / 2 + (info.Width / 2) * (float)Math.Cos(radians);
    float y = info.Height / 2 + (info.Height / 2) * (float)Math.Sin(radians);

    if (angle == 0)
    {
        path.MoveTo(x, y);
    }
    else
    {
        path.LineTo(x, y);
    }
}
path.Close();
```

This results in an ellipse defined by 360 little lines. When it's rendered, it appears smooth.

Of course, you don't need to create an ellipse using a polyline because `SKPath` includes an `AddOval` method that does it for you. But you might want to draw a visual object that is not provided by `SKPath`.

The [Archimedean Spiral](#) page has code that similar to the ellipse code but with a crucial difference. It loops around the 360 degrees of the circle 10 times, continuously adjusting the radius:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    SKPoint center = new SKPoint(info.Width / 2, info.Height / 2);
    float radius = Math.Min(center.X, center.Y);

    using (SKPath path = new SKPath())
    {
        for (float angle = 0; angle < 3600; angle += 1)
        {
            float scaledRadius = radius * angle / 3600;
            double radians = Math.PI * angle / 180;
            float x = center.X + scaledRadius * (float)Math.Cos(radians);
            float y = center.Y + scaledRadius * (float)Math.Sin(radians);
            SKPoint point = new SKPoint(x, y);

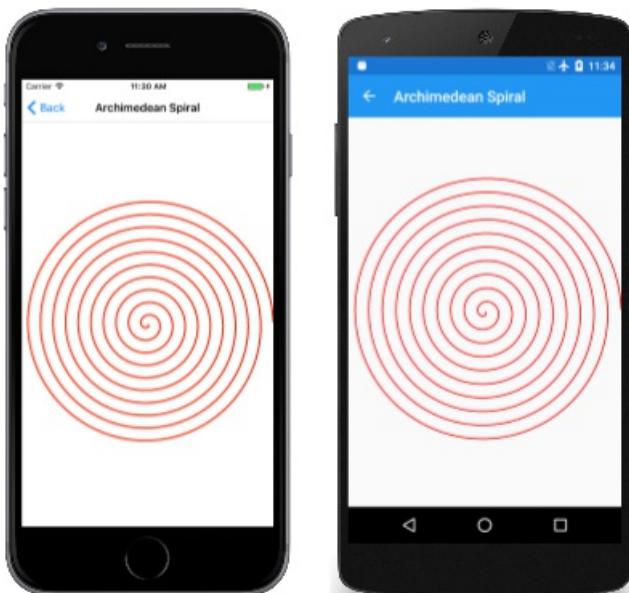
            if (angle == 0)
            {
                path.MoveTo(point);
            }
            else
            {
                path.LineTo(point);
            }
        }

        SKPaint paint = new SKPaint
        {
            Style = SKPaintStyle.Stroke,
            Color = SKColors.Red,
            StrokeWidth = 5
        };

        canvas.DrawPath(path, paint);
    }
}

```

The result is also called an *arithmetic spiral* because the offset between each loop is constant:



Notice that the `SKPath` is created in a `using` block. This `SKPath` consumes more memory than the `SKPath`

objects in the previous programs, which suggests that a `using` block is more appropriate to dispose any unmanaged resources.

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

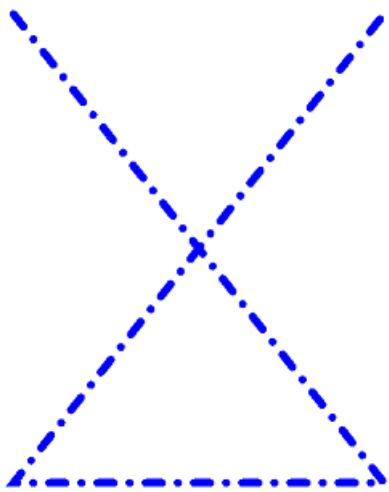
Dots and Dashes in SkiaSharp

3/5/2021 • 5 minutes to read • [Edit Online](#)

 [Download the sample](#)

Master the intricacies of drawing dotted and dashed lines in SkiaSharp

SkiaSharp lets you draw lines that are not solid but instead are composed of dots and dashes:



You do this with a *path effect*, which is an instance of the `SKPathEffect` class that you set to the `PathEffect` property of `SKPaint`. You can create a path effect (or combine path effects) using one of the static creation methods defined by `SKPathEffect`. (`SKPathEffect` is one of six effects supported by SkiaSharp; the others are described in the section [SkiaSharp Effect](#).)

To draw dotted or dashed lines, you use the `SKPathEffect.CreateDash` static method. There are two arguments: This first is an array of `float` values that indicate the lengths of the dots and dashes and the length of the spaces between them. This array must have an even number of elements, and there should be at least two elements. (There can be zero elements in the array but that results in a solid line.) If there are two elements, the first is the length of a dot or dash, and the second is the length of the gap before the next dot or dash. If there are more than two elements, then they are in this order: dash length, gap length, dash length, gap length, and so on.

Generally, you'll want to make the dash and gap lengths a multiple of the stroke width. If the stroke width is 10 pixels, for example, then the array { 10, 10 } will draw a dotted line where the dots and gaps are the same length as the stroke thickness.

However, the `strokeCap` setting of the `SKPaint` object also affects these dots and dashes. As you'll see shortly, that has an impact on the elements of this array.

Dotted and dashed lines are demonstrated on the [Dots and Dashes](#) page. The `DotsAndDashesPage.xaml` file instantiates two `Picker` views, one for letting you select a stroke cap and the second to select a dash array:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp;assembly=SkiaSharp"
    xmlns:skiaforms="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Paths.DotsAndDashesPage"
    Title="Dots and Dashes">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <Picker x:Name="strokeCapPicker"
            Title="Stroke Cap"
            Grid.Row="0"
            Grid.Column="0"
            SelectedIndexChanged="OnPickerSelectedIndexChanged">
            <Picker.ItemsSource>
                <x:Array Type="{x:Type skia:SKStrokeCap}">
                    <x:Static Member="skia:SKStrokeCap.Butt" />
                    <x:Static Member="skia:SKStrokeCap.Round" />
                    <x:Static Member="skia:SKStrokeCap.Square" />
                </x:Array>
            </Picker.ItemsSource>
            <Picker.SelectedIndex>
                0
            </Picker.SelectedIndex>
        </Picker>

        <Picker x:Name="dashArrayPicker"
            Title="Dash Array"
            Grid.Row="0"
            Grid.Column="1"
            SelectedIndexChanged="OnPickerSelectedIndexChanged">
            <Picker.ItemsSource>
                <x:Array Type="{x:Type x:String}">
                    <x:String>10, 10</x:String>
                    <x:String>30, 10</x:String>
                    <x:String>10, 10, 30, 10</x:String>
                    <x:String>0, 20</x:String>
                    <x:String>20, 20</x:String>
                    <x:String>0, 20, 20, 20</x:String>
                </x:Array>
            </Picker.ItemsSource>
            <Picker.SelectedIndex>
                0
            </Picker.SelectedIndex>
        </Picker>

        <skiaforms:SKCanvasView x:Name="canvasView"
            PaintSurface="OnCanvasViewPaintSurface"
            Grid.Row="1"
            Grid.Column="0"
            Grid.ColumnSpan="2" />
    </Grid>
</ContentPage>

```

The first three items in the `dashArrayPicker` assume that the stroke width is 10 pixels. The { 10, 10 } array is for a dotted line, { 30, 10 } is for a dashed line, and { 10, 10, 30, 10 } is for a dot-and-dash line. (The other three will be discussed shortly.)

The [DotsAndDashesPage](#) code-behind file contains the [PaintSurface](#) event handler and a couple of helper routines for accessing the [Picker](#) views:

```
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    SKPaint paint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Blue,
        StrokeWidth = 10,
        StrokeCap = (SKStrokeCap)strokeCapPicker.SelectedItem,
        PathEffect = SKPathEffect.CreateDash(GetPickerArray(dashArrayPicker), 20)
    };

    SKPath path = new SKPath();
    path.MoveTo(0.2f * info.Width, 0.2f * info.Height);
    path.LineTo(0.8f * info.Width, 0.8f * info.Height);
    path.LineTo(0.2f * info.Width, 0.8f * info.Height);
    path.LineTo(0.8f * info.Width, 0.2f * info.Height);

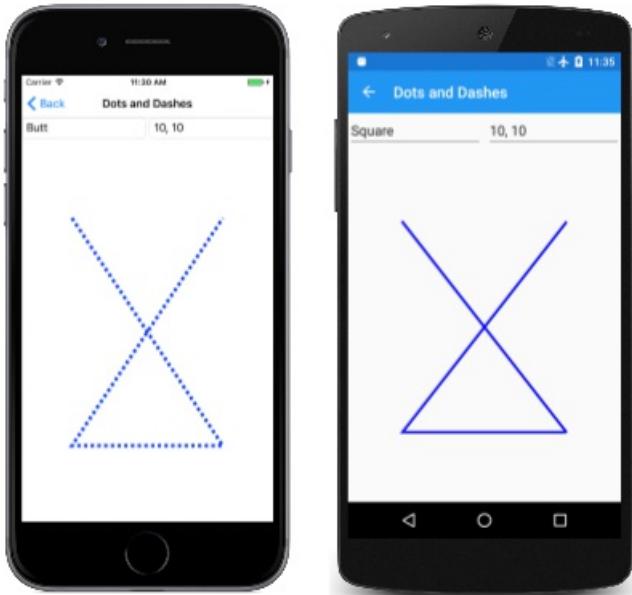
    canvas.DrawPath(path, paint);
}

float[] GetPickerArray(Picker picker)
{
    if (picker.SelectedIndex == -1)
    {
        return new float[0];
    }

    string str = (string)picker.SelectedItem;
    string[] strs = str.Split(new char[] { ' ', ',' }, StringSplitOptions.RemoveEmptyEntries);
    float[] array = new float[strs.Length];

    for (int i = 0; i < strs.Length; i++)
    {
        array[i] = Convert.ToSingle(strs[i]);
    }
    return array;
}
```

In the following screenshots, the iOS screen on the far left shows a dotted line:



However, the Android screen is also supposed to show a dotted line using the array { 10, 10 } but instead the line is solid. What happened? The problem is that the Android screen also has a stroke caps setting of `Square`. This extends all the dashes by half the stroke width, causing them to fill up the gaps.

To get around this problem when using a stroke cap of `Square` or `Round`, you must decrease the dash lengths in the array by the stroke length (sometimes resulting in a dash length of 0), and increase the gap lengths by the stroke length. This is how the final three dash arrays in the `Picker` in the XAML file were calculated:

- { 10, 10 } becomes { 0, 20 } for a dotted line
- { 30, 10 } becomes { 20, 20 } for a dashed line
- { 10, 10, 30, 10 } becomes { 0, 20, 20, 20 } for a dotted and dashed line

The UWP screen shows that dotted and dashed line for a stroke cap of `Round`. The `Round` stroke cap often gives the best appearance of dots and dashes in thick lines.

So far no mention has been made of the second parameter to the `SKPathEffect.CreateDash` method. This parameter is named `phase` and it refers to an offset within the dot-and-dash pattern for the beginning of the line. For example, if the dash array is { 10, 10 } and the `phase` is 10, then the line begins with a gap rather than a dot.

One interesting application of the `phase` parameter is in an animation. The [Animated Spiral](#) page is similar to the [Archimedean Spiral](#) page, except that the `AnimatedSpiralPage` class animates the `phase` parameter using the Xamarin.Forms `Device.Timer` method:

```

public class AnimatedSpiralPage : ContentPage
{
    const double cycleTime = 250;           // in milliseconds

    SKCanvasView canvasView;
    Stopwatch stopwatch = new Stopwatch();
    bool pageIsActive;
    float dashPhase;

    public AnimatedSpiralPage()
    {
        Title = "Animated Spiral";

        canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    protected override void OnAppearing()
    {
        base.OnAppearing();
        pageIsActive = true;
        stopwatch.Start();

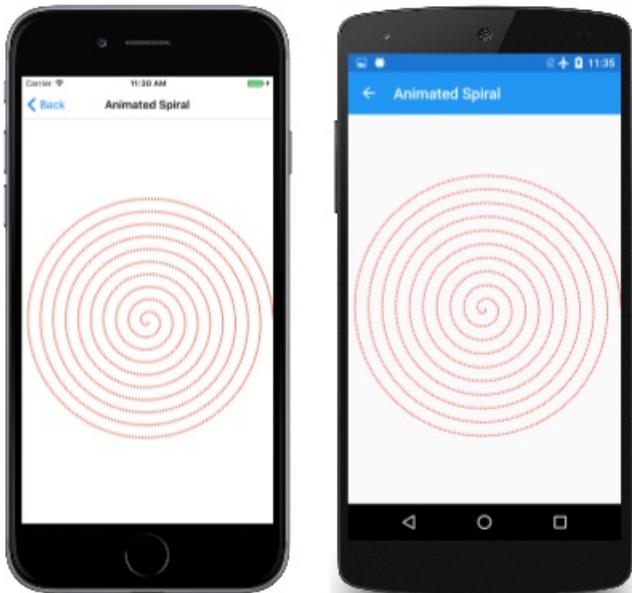
        Device.StartTimer(TimeSpan.FromMilliseconds(33), () =>
        {
            double t = stopwatch.Elapsed.TotalMilliseconds % cycleTime / cycleTime;
            dashPhase = (float)(10 * t);
            canvasView.InvalidateSurface();

            if (!pageIsActive)
            {
                stopwatch.Stop();
            }

            return pageIsActive;
        });
    }
    ...
}

```

Of course, you'll have to actually run the program to see the animation:



Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Finger Painting in SkiaSharp

3/5/2021 • 3 minutes to read • [Edit Online](#)

 [Download the sample](#)

Use your fingers to paint on the canvas.

An `SKPath` object can be continually updated and displayed. This feature allows a path to be used for interactive drawing, such as in a finger-painting program.



The touch support in Xamarin.Forms does not allow tracking individual fingers on the screen, so a Xamarin.Forms touch-tracking effect has been developed to provide additional touch support. This effect is described in the article [Invoking Events from Effects](#). The sample program [Touch-Tracking Effect Demos](#) includes two pages that use SkiaSharp, including a finger-painting program.

The [SkiaSharpFormsDemos](#) solution includes this touch-tracking event. The .NET Standard library project includes the `TouchEvent` class, the `TouchActionType` enumeration, the `TouchActionEventHandler` delegate, and the `TouchEventArgs` class. Each of the platform projects includes a `TouchEvent` class for that platform; the iOS project also contains a `TouchRecognizer` class.

The [Finger Paint](#) page in [SkiaSharpFormsDemos](#) is a simplified implementation of finger painting. It does not allow selecting color or stroke width, it has no way to clear the canvas, and of course you can't save your artwork.

The [FingerPaintPage.xaml](#) file puts the `SKCanvasView` in a single-cell `Grid` and attaches the `TouchEvent` to that `Grid`:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    xmlns:tt="clr-namespace:TouchTracking"
    x:Class="SkiaSharpFormsDemos.Paths.FingerPaintPage"
    Title="Finger Paint">

    <Grid BackgroundColor="White">
        <skia:SKCanvasView x:Name="canvasView"
            PaintSurface="OnCanvasViewPaintSurface" />
        <Grid.Effects>
            <tt:TouchEvent Capture="True"
                TouchAction="OnTouchEffectAction" />
        </Grid.Effects>
    </Grid>
</ContentPage>

```

Attaching the `TouchEvent` directly to the `SKCanvasView` does not work under all platforms.

The [FingerPaintPage.xaml.cs](#) code-behind file defines two collections for storing the `SKPath` objects, as well as an `SKPaint` object for rendering these paths:

```

public partial class FingerPaintPage : ContentPage
{
    Dictionary<long, SKPath> inProgressPaths = new Dictionary<long, SKPath>();
    List<SKPath> completedPaths = new List<SKPath>();

    SKPaint paint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Blue,
        StrokeWidth = 10,
        StrokeCap = SKStrokeCap.Round,
        StrokeJoin = SKStrokeJoin.Round
    };

    public FingerPaintPage()
    {
        InitializeComponent();
    }
    ...
}

```

As the name suggests, the `inProgressPaths` dictionary stores the paths that are currently being drawn by one or more fingers. The dictionary key is the touch ID that accompanies the touch events. The `completedPaths` field is a collection of paths that were finished when a finger that was drawing the path lifted from the screen.

The `TouchAction` handler manages these two collections. When a finger first touches the screen, a new `SKPath` is added to `inProgressPaths`. As that finger moves, additional points are added to the path. When the finger is released, the path is transferred to the `completedPaths` collection. You can paint with multiple fingers simultaneously. After each change to one of the paths or collections, the `SKCanvasView` is invalidated:

```

public partial class FingerPaintPage : ContentPage
{
    ...
    void OnTouchEffectAction(object sender, TouchEventArgs args)
    {
        switch (args.Type)
        {
            case TouchActionType.Pressed:
                if (!inProgressPaths.ContainsKey(args.Id))
                {
                    SKPath path = new SKPath();
                    path.MoveTo(ConvertToPixel(args.Location));
                    inProgressPaths.Add(args.Id, path);
                    canvasView.InvalidateSurface();
                }
                break;

            case TouchActionType.Moved:
                if (inProgressPaths.ContainsKey(args.Id))
                {
                    SKPath path = inProgressPaths[args.Id];
                    path.LineTo(ConvertToPixel(args.Location));
                    canvasView.InvalidateSurface();
                }
                break;

            case TouchActionType.Released:
                if (inProgressPaths.ContainsKey(args.Id))
                {
                    completedPaths.Add(inProgressPaths[args.Id]);
                    inProgressPaths.Remove(args.Id);
                    canvasView.InvalidateSurface();
                }
                break;

            case TouchActionType.Cancelled:
                if (inProgressPaths.ContainsKey(args.Id))
                {
                    inProgressPaths.Remove(args.Id);
                    canvasView.InvalidateSurface();
                }
                break;
        }
    }

    ...
    SKPoint ConvertToPixel(Point pt)
    {
        return new SKPoint((float)(canvasView.CanvasSize.Width * pt.X / canvasView.Width),
                           (float)(canvasView.CanvasSize.Height * pt.Y / canvasView.Height));
    }
}

```

The points accompanying the touch-tracking events are Xamarin.Forms coordinates; these must be converted to SkiaSharp coordinates, which are pixels. That's the purpose of the `ConvertToPixel` method.

The `PaintSurface` handler then simply renders both collections of paths. The earlier completed paths appear underneath the paths in progress:

```

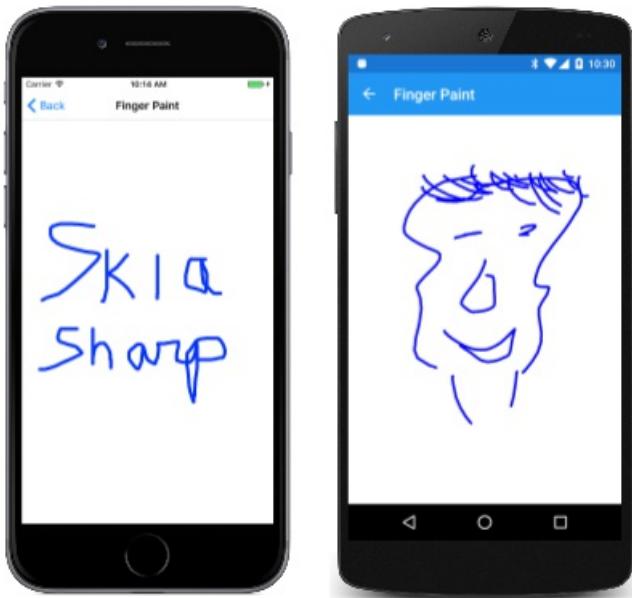
public partial class FingerPaintPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKCanvas canvas = args.Surface.Canvas;
        canvas.Clear();

        foreach (SKPath path in completedPaths)
        {
            canvas.DrawPath(path, paint);
        }

        foreach (SKPath path in inProgressPaths.Values)
        {
            canvas.DrawPath(path, paint);
        }
    }
    ...
}

```

Your finger paintings are only limited by your talent:



You've now seen how to draw lines and to define curves using parametric equations. A later section on [SkiaSharp Curves and Paths](#) covers the various types of curves that `SKPath` supports. But a useful prerequisite is an exploration of [SkiaSharp Transforms](#).

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)
- [Touch-Tracking Effect Demos \(sample\)](#)
- [Invoking Events from Effects](#)

SkiaSharp Transforms

3/5/2021 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

Learn about transforms for displaying SkiaSharp graphics

SkiaSharp supports traditional graphics transforms that are implemented as methods of the `SKCanvas` object. Mathematically, transforms alter the coordinates and sizes that you specify in `SKCanvas` drawing functions as the graphical objects are rendered. Transforms are often convenient for drawing repetitive graphics or for animation. Some techniques — such as rotating bitmaps or text — are not possible without the use of transforms.

SkiaSharp transforms support the following operations:

- `Translate` to shift coordinates from one location to another
- `Scale` to increase or decrease coordinates and sizes
- `Rotate` to rotate coordinates around a point
- `Skew` to shift coordinates horizontally or vertically so that a rectangle becomes a parallelogram

These are known as *affine* transforms. Affine transforms always preserve parallel lines and never cause a coordinate or size to become infinite. A square is never transformed into anything other than a parallelogram, and a circle is never transformed into anything other than an ellipse.

SkiaSharp also supports non-affine transforms (also called *projective* or *perspective* transforms) based on a standard 3-by-3 transform matrix. A non-affine transform allows a square to be transformed into any convex quadrilateral, which is a four-sided figure with all interior angles less than 180 degrees. Non-affine transforms can cause coordinates or sizes to become infinite, but they are vital for 3D effects.

Differences between SkiaSharp and Xamarin.Forms Transforms

Xamarin.Forms also supports transforms that are similar to those in SkiaSharp. The `Xamarin.Forms.VisualElement` class defines the following transform properties:

- `TranslationX` and `TranslationY`
- `Scale`
- `Rotation`, `RotationX`, and `RotationY`

The `RotationX` and `RotationY` properties are perspective transforms that create quasi-3D effects.

There are several crucial differences between SkiaSharp transforms and Xamarin.Forms transforms:

The first difference is that SkiaSharp transforms are applied to the entire `SKCanvas` object while the Xamarin.Forms transforms are applied to individual `VisualElement` derivatives. (You can apply the Xamarin.Forms transforms to the `SKCanvasView` object itself, because `SKCanvasView` derives from `VisualElement`, but within that `SKCanvasView`, the SkiaSharp transforms apply.)

The SkiaSharp transforms are relative to the upper-left corner of the `SKCanvas` while Xamarin.Forms transforms are relative to the upper-left corner of the `VisualElement` to which they are applied. This difference is important when applying scaling and rotation transforms because these transforms are always relative to a particular point.

The really big difference is that SkiaSharp transforms are *methods* while the Xamarin.Forms transforms are *properties*. This is a semantic difference beyond the syntactical difference: SkiaSharp transforms perform an operation while Xamarin.Forms transforms set a state. SkiaSharp transforms apply to subsequently drawn graphics objects, but not to graphics objects that are drawn before the transform is applied. In contrast, a Xamarin.Forms transform applies to a previously rendered element as soon as the property is set. SkiaSharp transforms are cumulative as the methods are called; Xamarin.Forms transforms are replaced when the property is set with another value.

All the sample programs in this section appear in the **SkiaSharp Transforms** section of the **SkiaSharpFormsDemos** program. Source code can be found in the **Transforms** folder of the solution.

The Translate Transform

Learn how to use the translate transform to shift SkiaSharp graphics.

The Scale Transform

Discover the SkiaSharp scale transform for scaling objects to various sizes.

The Rotate Transform

Explore the effects and animations possible with the SkiaSharp rotate transform.

The Skew Transform

See how the skew transform can create tilted graphical object.

Matrix Transforms

Dive deeper into SkiaSharp transforms with the versatile transform matrix.

Touch Manipulations

Use matrix transforms to implement touch manipulations for dragging, scaling, and rotation.

Non-Affine Transforms

Go beyond the ordinary with non-affine transform effects.

3D Rotation

Use non-affine transforms to rotate 2D objects in 3D space.

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

The Translate Transform

3/5/2021 • 7 minutes to read • [Edit Online](#)



[Download the sample](#)

Learn how to use the translate transform to shift SkiaSharp graphics

The simplest type of transform in SkiaSharp is the *translate* or *translation* transform. This transform shifts graphical objects in the horizontal and vertical directions. In a sense, translation is the most unnecessary transform because you can usually accomplish the same effect by simply changing the coordinates that you're using in the drawing function. When rendering a path, however, all the coordinates are encapsulated in the path, so it's much easier applying a translate transform to shift the entire path.

Translation is also useful for animation and for simple text effects:

The image displays three lines of text demonstrating different SkiaSharp text effects. The first line, 'SHADOW', has a pink-to-white gradient shadow effect. The second line, 'ENGRAVE', has a dark gray engraved effect. The third line, 'EMBOSS', has a light gray embossed effect. All text is in a bold, sans-serif font.

The `Translate` method in `SKCanvas` has two parameters that cause subsequently drawn graphics objects to be shifted horizontally and vertically:

```
public void Translate (Single dx, Single dy)
```

These arguments may be negative. A second `Translate` method combines the two translation values in a single `SKPoint` value:

```
public void Translate (SKPoint point)
```

The **Accumulated Translate** page of the [SkiaSharpForms](#) sample program demonstrates that multiple calls of the `Translate` method are cumulative. The `AccumulatedTranslatePage` class displays 20 versions of the same rectangle, each one offset from the previous rectangle just enough so they stretch along the diagonal. Here's the `PaintSurface` event handler:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

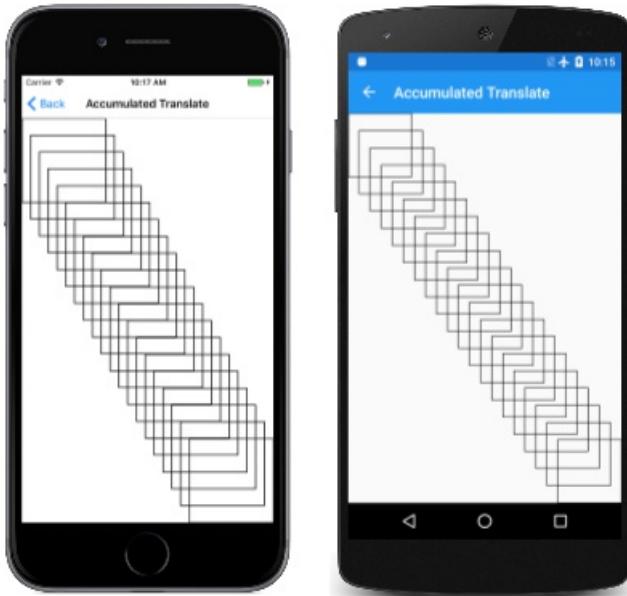
    using (SKPaint strokePaint = new SKPaint())
    {
        strokePaint.Color = SKColors.Black;
        strokePaint.Style = SKPaintStyle.Stroke;
        strokePaint.StrokeWidth = 3;

        int rectangleCount = 20;
        SKRect rect = new SKRect(0, 0, 250, 250);
        float xTranslate = (info.Width - rect.Width) / (rectangleCount - 1);
        float yTranslate = (info.Height - rect.Height) / (rectangleCount - 1);

        for (int i = 0; i < rectangleCount; i++)
        {
            canvas.DrawRect(rect, strokePaint);
            canvas.Translate(xTranslate, yTranslate);
        }
    }
}

```

The successive rectangles trickle down the page:



If the accumulated translation factors are dx and dy , and the point you specify in a drawing function is (x, y) , then the graphical object is rendered at the point (x', y') , where:

$$x' = x + dx$$

$$y' = y + dy$$

These are known as the *transform formulas* for translation. The default values of dx and dy for a new `SKCanvas` are 0.

It is common to use the translate transform for shadow effects and similar techniques, as the [Translate Text Effects](#) page demonstrates. Here's the relevant part of the `PaintSurface` handler in the `TranslateTextEffectsPage` class:

```

float textSize = 150;

using (SKPaint textPaint = new SKPaint())
{
    textPaint.Style = SKPaintStyle.Fill;
    textPaint.TextSize = textSize;
    textPaint.FakeBoldText = true;

    float x = 10;
    float y = textSize;

    // Shadow
    canvas.Translate(10, 10);
    textPaint.Color = SKColors.Black;
    canvas.DrawText("SHADOW", x, y, textPaint);
    canvas.Translate(-10, -10);
    textPaint.Color = SKColors.Pink;
    canvas.DrawText("SHADOW", x, y, textPaint);

    y += 2 * textSize;

    // Engrave
    canvas.Translate(-5, -5);
    textPaint.Color = SKColors.Black;
    canvas.DrawText("ENGRAVE", x, y, textPaint);
    canvas.ResetMatrix();
    textPaint.Color = SKColors.White;
    canvas.DrawText("ENGRAVE", x, y, textPaint);

    y += 2 * textSize;

    // Emboss
    canvas.Save();
    canvas.Translate(5, 5);
    textPaint.Color = SKColors.Black;
    canvas.DrawText("EMBOSS", x, y, textPaint);
    canvas.Restore();
    textPaint.Color = SKColors.White;
    canvas.DrawText("EMBOSS", x, y, textPaint);
}

```

In each of the three examples, `Translate` is called for displaying the text to offset it from the location given by the `x` and `y` variables. Then the text is displayed again in another color with no translation effect:



Each of the three examples shows a different way of negating the `Translate` call:

The first example simply calls `Translate` again but with negative values. Because the `Translate` calls are cumulative, this sequence of calls simply restores the total translation to the default values of zero.

The second example calls `ResetMatrix`. This causes all transforms to return to their default state.

The third example saves the state of the `SKCanvas` object with a call to `Save` and then restores the state with a call to `Restore`. This is the most versatile way to manipulate transforms for a series of drawing operations. These `Save` and `Restore` calls function like a stack: You can call `Save` multiple times, and then call `Restore` in reverse sequence to return to previous states. The `Save` method returns an integer, and you can pass that integer to `RestoreToCount` to effectively call `Restore` multiple times. The `SaveCount` property returns the number of states currently saved on the stack.

You can also use the `SKAutoCanvasRestore` class for restoring the canvas state. The constructor of this class is intended to be called in a `using` statement; the canvas state is automatically restored at the end of the `using` block.

However, you don't have to worry about transforms carrying over from one call of the `PaintSurface` handler to the next. Each new call to `PaintSurface` delivers a fresh `SKCanvas` object with default transforms.

Another common use of the `Translate` transform is for rendering a visual object that has been originally created using coordinates that are convenient for drawing. For example, you might want to specify coordinates for an analog clock with a center at the point (0, 0). You can then use transforms to display the clock where you want it. This technique is demonstrated in the [Hendecagram Array] page. The `HendecagramArrayPage` class begins by creating an `SKPath` object for an 11-pointed star. The `HendecagramPath` object is defined as public, static, and read-only so that it can be accessed from other demonstration programs. It is created in a static constructor:

```
public class HendecagramArrayPage : ContentPage
{
    ...
    public static readonly SKPath HendecagramPath;

    static HendecagramArrayPage()
    {
        // Create 11-pointed star
        HendecagramPath = new SKPath();
        for (int i = 0; i < 11; i++)
        {
            double angle = 5 * i * 2 * Math.PI / 11;
            SKPoint pt = new SKPoint(100 * (float)Math.Sin(angle),
                                    -100 * (float)Math.Cos(angle));
            if (i == 0)
            {
                HendecagramPath.MoveTo(pt);
            }
            else
            {
                HendecagramPath.LineTo(pt);
            }
        }
        HendecagramPath.Close();
    }
}
```

If the center of the star is the point (0, 0), all the points of the star are on a circle surrounding that point. Each point is a combination of sine and cosine values of an angle that increases by 5/11ths of 360 degrees. (It's also possible to create an 11-pointed star by increasing the angle by 2/11th, 3/11ths, or 4/11th of the circle.) The radius of that circle is set as 100.

If this path is rendered without any transforms, the center will be positioned at the upper-left corner of the

`SKCanvas`, and only a quarter of it will be visible. The `PaintSurface` handler of `HendecagramPage` instead uses `Translate` to tile the canvas with multiple copies of the star, each one randomly colored:

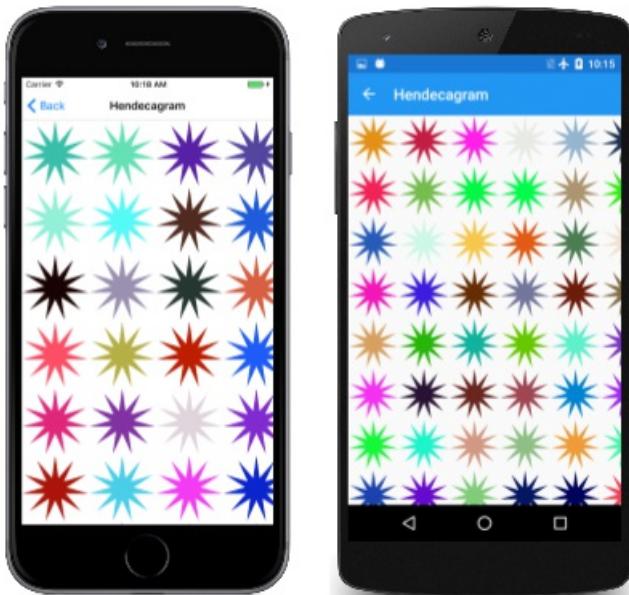
```
public class HendecagramArrayPage : ContentPage
{
    Random random = new Random();
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        using (SKPaint paint = new SKPaint())
        {
            for (int x = 100; x < info.Width + 100; x += 200)
                for (int y = 100; y < info.Height + 100; y += 200)
                {
                    // Set random color
                    byte[] bytes = new byte[3];
                    random.NextBytes(bytes);
                    paint.Color = new SKColor(bytes[0], bytes[1], bytes[2]);

                    // Display the hendecagram
                    canvas.Save();
                    canvas.Translate(x, y);
                    canvas.DrawPath(HendecagramPath, paint);
                    canvas.Restore();
                }
        }
    }
}
```

Here's the result:



Animations often involve transforms. The **Hendecagram Animation** page moves the 11-pointed star around in a circle. The `HendecagramAnimationPage` class begins with some fields and overrides of the `OnAppearing` and `OnDisappearing` methods to start and stop a Xamarin.Forms timer:

```

public class HendecagramAnimationPage : ContentPage
{
    const double cycleTime = 5000;      // in milliseconds

    SKCanvasView canvasView;
    Stopwatch stopwatch = new Stopwatch();
    bool pageIsActive;
    float angle;

    public HendecagramAnimationPage()
    {
        Title = "Hedecagram Animation";

        canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    protected override void OnAppearing()
    {
        base.OnAppearing();
        pageIsActive = true;
        stopwatch.Start();

        Device.StartTimer(TimeSpan.FromMilliseconds(33), () =>
        {
            double t = stopwatch.Elapsed.TotalMilliseconds % cycleTime / cycleTime;
            angle = (float)(360 * t);
            canvasView.InvalidateSurface();

            if (!pageIsActive)
            {
                stopwatch.Stop();
            }

            return pageIsActive;
        });
    }

    protected override void OnDisappearing()
    {
        base.OnDisappearing();
        pageIsActive = false;
    }
    ...
}

```

The `angle` field is animated from 0 degrees to 360 degrees every 5 seconds. The `PaintSurface` handler uses the `angle` property in two ways: to specify the hue of the color in the `SKColor.FromHsl` method, and as an argument to the `Math.Sin` and `Math.Cos` methods to govern the location of the star:

```

public class HendecagramAnimationPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

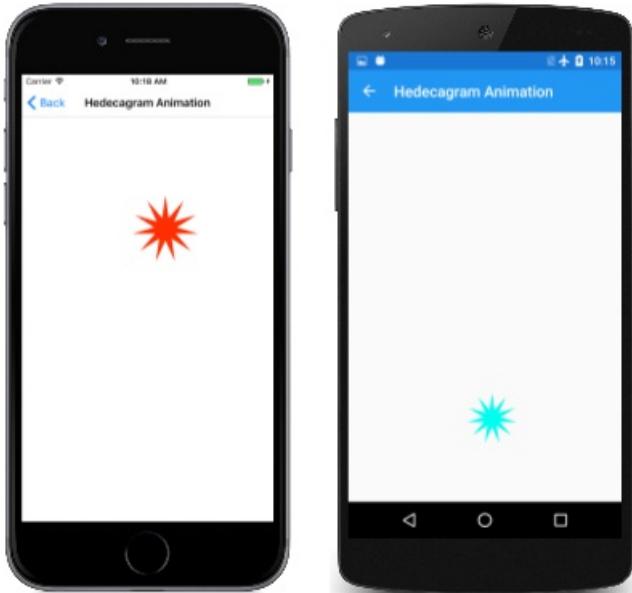
        canvas.Clear();
        canvas.Translate(info.Width / 2, info.Height / 2);
        float radius = (float)Math.Min(info.Width, info.Height) / 2 - 100;

        using (SKPaint paint = new SKPaint())
        {
            paint.Style = SKPaintStyle.Fill;
            paint.Color = SKColor.FromHsl(angle, 100, 50);

            float x = radius * (float)Math.Sin(Math.PI * angle / 180);
            float y = -radius * (float)Math.Cos(Math.PI * angle / 180);
            canvas.Translate(x, y);
            canvas.DrawPath(HendecagramPage.HendecagramPath, paint);
        }
    }
}

```

The `PaintSurface` handler calls the `Translate` method twice, first to translate to the center of the canvas, and then to translate to the circumference of a circle centered around (0, 0). The radius of the circle is set to be as large as possible while still keeping the star within the confines of the page:



Notice that the star maintains the same orientation as it revolves around the center of the page. It doesn't rotate at all. That's a job for a rotate transform.

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

The Scale Transform

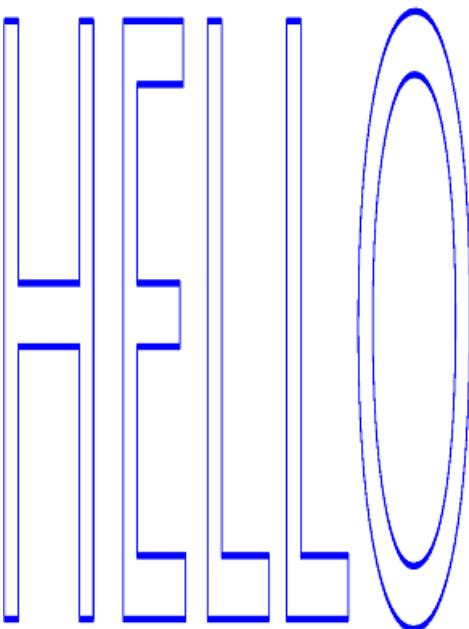
3/5/2021 • 11 minutes to read • [Edit Online](#)



[Download the sample](#)

Discover the **SkiaSharp** scale transform for scaling objects to various sizes

As you've seen in [The Translate Transform](#) article, the translate transform can move a graphical object from one location to another. In contrast, the scale transform changes the size of the graphical object:



The scale transform also often causes graphics coordinates to move as they are made larger.

Earlier you saw two transform formulas that describe the effects of translation factors of `dx` and `dy`:

$$x' = x + dx$$

$$y' = y + dy$$

Scale factors of `sx` and `sy` are multiplicative rather than additive:

$$x' = sx \cdot x$$

$$y' = sy \cdot y$$

The default values of the translate factors are 0; the default values of the scale factors are 1.

The `SKCanvas` class defines four `Scale` methods. The first `Scale` method is for cases when you want the same horizontal and vertical scaling factor:

```
public void Scale (Single s)
```

This is known as *isotropic* scaling — scaling that is the same in both directions. Isotropic scaling preserves the object's aspect ratio.

The second `Scale` method lets you specify different values for horizontal and vertical scaling:

```
public void Scale (Single sx, Single sy)
```

This results in *anisotropic* scaling. The third `Scale` method combines the two scaling factors in a single `SKPoint` value:

```
public void Scale (SKPoint size)
```

The fourth `Scale` method will be described shortly.

The **Basic Scale** page demonstrates the `Scale` method. The [BasicScalePage.xaml](#) file contains two `Slider` elements that let you select horizontal and vertical scaling factors between 0 and 10. The [BasicScalePage.xaml.cs](#) code-behind file uses those values to call `Scale` before displaying a rounded rectangle stroked with a dashed line and sized to fit some text in the upper-left corner of the canvas:

```
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear(SKColors.SkyBlue);

    using (SKPaint strokePaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Red,
        StrokeWidth = 3,
        PathEffect = SKPathEffect.CreateDash(new float[] { 7, 7 }, 0)
    })
    using (SKPaint textPaint = new SKPaint
    {
        Style = SKPaintStyle.Fill,
        Color = SKColors.Blue,
        TextSize = 50
    })
    {
        canvas.Scale((float)xScaleSlider.Value,
                    (float)yScaleSlider.Value);

        SKRect textBounds = new SKRect();
        textPaint.MeasureText("Title", ref textBounds);

        float margin = 10;
        SKRect borderRect = SKRect.Create(new SKPoint(margin, margin), textBounds.Size);
        canvas.DrawRoundRect(borderRect, 20, 20, strokePaint);
        canvas.DrawText("Title", margin, -textBounds.Top + margin, textPaint);
    }
}
```

You might wonder: How do the scaling factors affect the value returned from the `MeasureText` method of `SKPaint`? The answer is: Not at all. `Scale` is a method of `SKCanvas`. It does not affect anything you do with an `SKPaint` object until you use that object to render something on the canvas.

As you can see, everything drawn after the `Scale` call increases proportionally:



The text, the width of the dashed line, the length of the dashes in that line, the rounding of the corners, and the 10-pixel margin between the left and top edges of the canvas and the rounded rectangle are all subject to the same scaling factors.

IMPORTANT

The Universal Windows Platform does not properly render anisotropically scaled text.

Anisotropic scaling causes the stroke width to become different for lines aligned with the horizontal and vertical axes. (This is also evident from the first image on this page.) If you don't want the stroke width to be affected by the scaling factors, set it to 0 and it will always be one pixel wide regardless of the `Scale` setting.

Scaling is relative to the upper-left corner of the canvas. This might be exactly what you want, but it might not be. Suppose you want to position the text and rectangle somewhere else on the canvas and you want to scale it relative to its center. In that case you can use the fourth version of the `Scale` method, which includes two additional parameters to specify the center of scaling:

```
public void Scale (Single sx, Single sy, Single px, Single py)
```

The `px` and `py` parameters define a point that is sometimes called the *scaling center* but in the SkiaSharp documentation is referred to as a *pivot point*. This is a point relative to the upper-left corner of the canvas that is not affected by the scaling. All scaling occurs relative to that center.

The [Centered Scale](#) page shows how this works. The `PaintSurface` handler is similar to the **Basic Scale** program except that the `margin` value is calculated to center the text horizontally, which implies that the program works best in portrait mode:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear(SKColors.SkyBlue);

    using (SKPaint strokePaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Red,
        StrokeWidth = 3,
        PathEffect = SKPathEffect.CreateDash(new float[] { 7, 7 }, 0)
    })
    using (SKPaint textPaint = new SKPaint
    {
        Style = SKPaintStyle.Fill,
        Color = SKColors.Blue,
        TextSize = 50
    })
    {
        SKRect textBounds = new SKRect();
        textPaint.MeasureText(Title, ref textBounds);
        float margin = (info.Width - textBounds.Width) / 2;

        float sx = (float)xScaleSlider.Value;
        float sy = (float)yScaleSlider.Value;
        float px = margin + textBounds.Width / 2;
        float py = margin + textBounds.Height / 2;

        canvas.Scale(sx, sy, px, py);

        SKRect borderRect = SKRect.Create(new SKPoint(margin, margin), textBounds.Size);
        canvas.DrawRoundRect(borderRect, 20, 20, strokePaint);
        canvas.DrawText(Title, margin, -textBounds.Top + margin, textPaint);
    }
}

```

The upper-left corner of the rounded rectangle is positioned `margin` pixels from the left of the canvas and `margin` pixels from the top. The last two arguments to the `Scale` method are set to those values plus the width and height of the text, which is also the width and height of the rounded rectangle. This means that all scaling is relative to the center of that rectangle:



The `Slider` elements in this program have a range of -10 to 10. As you can see, negative values of vertical scaling (such as on the Android screen in the center) cause objects to flip around the horizontal axis that passes through the center of scaling. Negative values of horizontal scaling (such as in the UWP screen on the right) cause objects to flip around the vertical axis that passes through the center of scaling.

The version of the `Scale` method with pivot points is a shortcut for a series of three `Translate` and `Scale` calls. You might want to see how this works by replacing the `Scale` method in the **Centered Scale** page with the following:

```
canvas.Translate(-px, -py);
```

These are the negatives of the pivot point coordinates.

Now run the program again. You'll see that the rectangle and text are shifted so that the center is in the upper-left corner of the canvas. You can barely see it. The sliders don't work of course because now the program doesn't scale at all.

Now add the basic `Scale` call (without a scaling center) *before* that `Translate` call:

```
canvas.Scale(sx, sy);
canvas.Translate(-px, -py);
```

If you're familiar with this exercise in other graphics programming systems, you might think that's wrong, but it's not. Skia handles successive transform calls a little differently from what you might be familiar with.

With the successive `Scale` and `Translate` calls, the center of the rounded rectangle is still in the upper-left corner, but you can now scale it relative to the upper-left corner of the canvas, which is also the center of the rounded rectangle.

Now, before that `Scale` call, add another `Translate` call with the centering values:

```
canvas.Translate(px, py);
canvas.Scale(sx, sy);
canvas.Translate(-px, -py);
```

This moves the scaled result back to the original position. Those three calls are equivalent to:

```
canvas.Scale(sx, sy, px, py);
```

The individual transforms are compounded so that the total transform formula is:

$$x' = sx \cdot (x - px) + px$$

$$y' = sy \cdot (y - py) + py$$

Keep in mind that the default values of `sx` and `sy` are 1. It's easy to convince yourself that the pivot point (px, py) is not transformed by these formulas. It remains in the same location relative to the canvas.

When you combine `Translate` and `Scale` calls, the order matters. If the `Translate` comes after the `Scale`, the translation factors are effectively scaled by the scaling factors. If the `Translate` comes before the `Scale`, the translation factors are not scaled. This process becomes somewhat clearer (albeit more mathematical) when the subject of transform matrices is introduced.

The `SKPath` class defines a read-only `Bounds` property that returns an `SKRect` defining the extent of the coordinates in the path. For example, when the `Bounds` property is obtained from the hendecagram path

created earlier, the `Left` and `Top` properties of the rectangle are approximately `-100`, the `Right` and `Bottom` properties are approximately `100`, and the `Width` and `Height` properties are approximately `200`. (Most of the actual values are a little less because the points of the stars are defined by a circle with a radius of `100` but only the top point is parallel with the horizontal or vertical axes.)

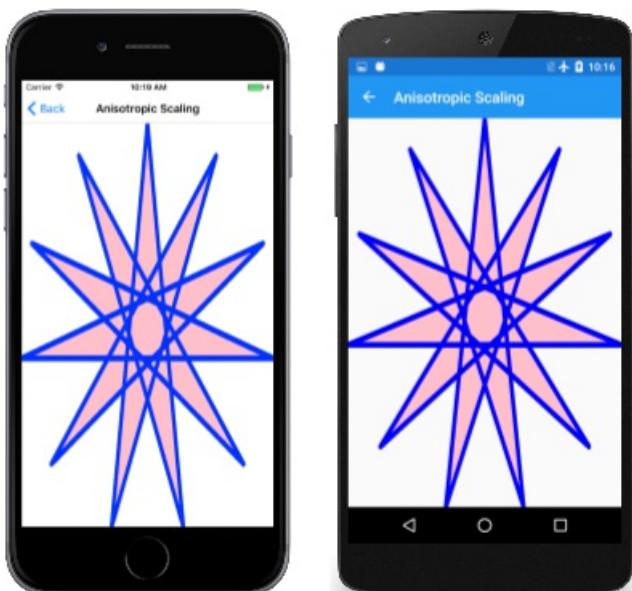
The availability of this information implies that it should be possible to derive scale and translate factors suitable for scaling a path to the size of the canvas. The [Anisotropic Scaling](#) page demonstrates this with the 11-pointed star. An *anisotropic* scale means that it's unequal in the horizontal and vertical directions, which means that the star won't retain its original aspect ratio. Here's the relevant code in the `PaintSurface` handler:

```
SKPath path = HendecagramPage.HendecagramPath;
SKRect pathBounds = path.Bounds;

using (SKPaint fillPaint = new SKPaint
{
    Style = SKPaintStyle.Fill,
    Color = SKColors.Pink
})
using (SKPaint strokePaint = new SKPaint
{
    Style = SKPaintStyle.Stroke,
    Color = SKColors.Blue,
    StrokeWidth = 3,
    StrokeJoin = SKStrokeJoin.Round
})
{
    canvas.Scale(info.Width / pathBounds.Width,
                info.Height / pathBounds.Height);
    canvas.Translate(-pathBounds.Left, -pathBounds.Top);

    canvas.DrawPath(path, fillPaint);
    canvas.DrawPath(path, strokePaint);
}
```

The `pathBounds` rectangle is obtained near the top of this code, and then used later with the width and height of the canvas in the `Scale` call. That call by itself will scale the coordinates of the path when it's rendered by the `DrawPath` call but the star will be centered in the upper-right corner of the canvas. It needs to be shifted down and to the left. This is the job of the `Translate` call. Those two properties of `pathBounds` are approximately `-100`, so the translation factors are about `100`. Because the `Translate` call is after the `Scale` call, those values are effectively scaled by the scaling factors, so they move the center of the star to the center of the canvas:



Another way you can think about the `Scale` and `Translate` calls is to determine the effect in reverse sequence:

The `Translate` call shifts the path so it becomes fully visible but oriented in the upper-left corner of the canvas.

The `Scale` method then makes that star larger relative to the upper-left corner.

Actually, it appears that the star is a little larger than the canvas. The problem is the stroke width. The `Bounds` property of `SKPath` indicates the dimensions of the coordinates encoded in the path, and that's what the program uses to scale it. When the path is rendered with a particular stroke width, the rendered path is larger than the canvas.

To fix this problem you need to compensate for that. One easy approach in this program is to add the following statement right before the `Scale` call:

```
pathBounds.Inflate(strokePaint.StrokeWidth / 2,
                   strokePaint.StrokeWidth / 2);
```

This increases the `pathBounds` rectangle by 1.5 units on all four sides. This is a reasonable solution only when the stroke join is rounded. A miter join can be longer and is difficult to calculate.

You can also use a similar technique with text, as the [Anisotropic Text](#) page demonstrates. Here's the relevant part of the `PaintSurface` handler from the [AnisotropicTextPage](#) class:

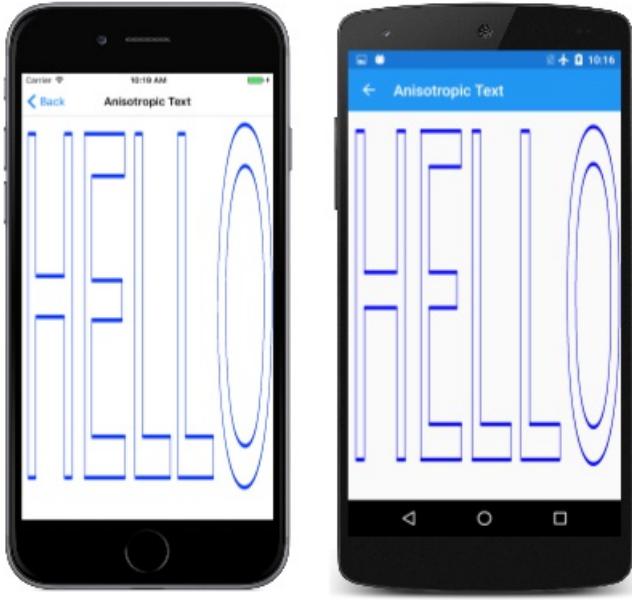
```
using (SKPaint textPaint = new SKPaint
{
    Style = SKPaintStyle.Stroke,
    Color = SKColors.Blue,
    StrokeWidth = 0.1f,
    StrokeJoin = SKStrokeJoin.Round
})
{
    SKRect textBounds = new SKRect();
    textPaint.MeasureText("HELLO", ref textBounds);

    // Inflate bounds by the stroke width
    textBounds.Inflate(textPaint.StrokeWidth / 2,
                      textPaint.StrokeWidth / 2);

    canvas.Scale(info.Width / textBounds.Width,
                info.Height / textBounds.Height);
    canvas.Translate(-textBounds.Left, -textBounds.Top);

    canvas.DrawText("HELLO", 0, 0, textPaint);
}
```

It's similar logic, and the text expands to the size of the page based on the text bounds rectangle returned from `MeasureText` (which is a little larger than the actual text):



If you need to preserve the aspect ratio of the graphical objects, you'll want to use isotropic scaling. The [Isotropic Scaling](#) page demonstrates this for the 11-pointed star. Conceptually, the steps for displaying a graphical object in the center of the page with isotropic scaling are:

- Translate the center of the graphical object to the upper-left corner.
- Scale the object based on the minimum of the horizontal and vertical page dimensions divided by the graphical object dimensions.
- Translate the center of the scaled object to the center of the page.

The [IsotropicScalingPage](#) performs these steps in reverse order before displaying the star:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    SKPath path = HendecagramArrayPage.HendecagramPath;
    SKRect pathBounds = path.Bounds;

    using (SKPaint fillPaint = new SKPaint())
    {
        fillPaint.Style = SKPaintStyle.Fill;

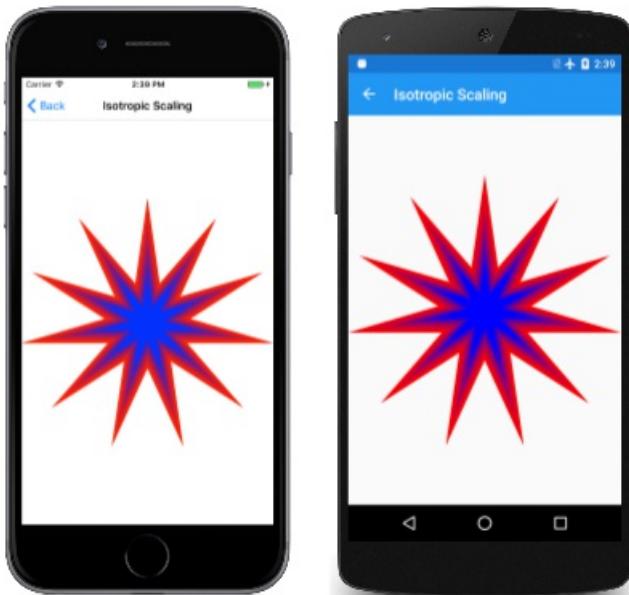
        float scale = Math.Min(info.Width / pathBounds.Width,
                               info.Height / pathBounds.Height);

        for (int i = 0; i <= 10; i++)
        {
            fillPaint.Color = new SKColor((byte)(255 * (10 - i) / 10),
                                         0,
                                         (byte)(255 * i / 10));
            canvas.Save();
            canvas.Translate(info.Width / 2, info.Height / 2);
            canvas.Scale(scale);
            canvas.Translate(-pathBounds.MidX, -pathBounds.MidY);
            canvas.DrawPath(path, fillPaint);
            canvas.Restore();

            scale *= 0.9f;
        }
    }
}

```

The code also displays the star 10 more times, each time decreasing the scaling factor by 10% and progressively changing the color from red to blue:



Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

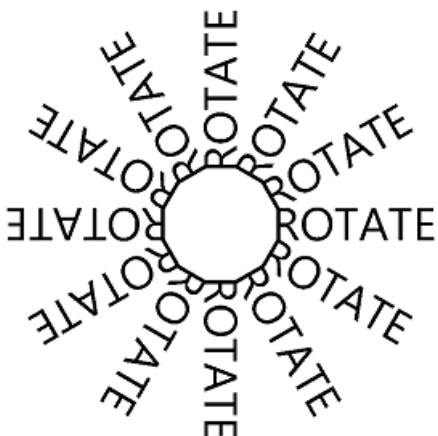
The Rotate Transform

3/5/2021 • 9 minutes to read • [Edit Online](#)

 [Download the sample](#)

Explore the effects and animations possible with the SkiaSharp rotate transform

With the rotate transform, SkiaSharp graphics objects break free of the constraint of alignment with the horizontal and vertical axes:



For rotating a graphical object around the point (0, 0), SkiaSharp supports both a `RotateDegrees` method and a `RotateRadians` method:

```
public void RotateDegrees (Single degrees)  
public void RotateRadians (Single radians)
```

A circle of 360 degrees is the same as twoπ radians, so it's easy to convert between the two units. Use whichever is convenient. All the trigonometric functions in the .NET `Math` class use units of radians.

Rotation is clockwise for increasing angles. (Although rotation on the Cartesian coordinate system is counter-clockwise by convention, clockwise rotation is consistent with Y coordinates increasing going down as in SkiaSharp.) Negative angles and angles greater than 360 degrees are allowed.

The transform formulas for rotation are more complex than those for translate and scale. For an angle of α , the transform formulas are:

$$x' = x \cdot \cos(\alpha) - y \cdot \sin(\alpha)$$

$$y' = x \cdot \sin(\alpha) + y \cdot \cos(\alpha)$$

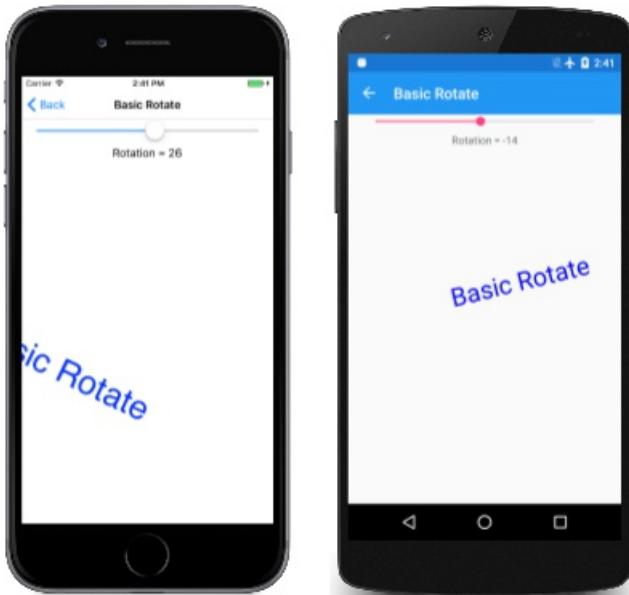
The **Basic Rotate** page demonstrates the `RotateDegrees` method. The `BasicRotate.xaml.cs` file displays some text with its baseline centered on the page and rotates it based on a `Slider` with a range of -360 to 360. Here's the relevant part of the `PaintSurface` handler:

```

using (SKPaint textPaint = new SKPaint
{
    Style = SKPaintStyle.Fill,
    Color = SKColors.Blue,
    TextAlign = SKTextAlign.Center,
    TextSize = 100
})
{
    canvas.RotateDegrees((float)rotateSlider.Value);
    canvas.DrawText(Title, info.Width / 2, info.Height / 2, textPaint);
}

```

Because rotation is centered around the upper-left corner of the canvas, for most angles set in this program, the text is rotated off the screen:



Very often you'll want to rotate something centered around a specified pivot point using these versions of the `RotateDegrees` and `RotateRadians` methods:

```

public void RotateDegrees (Single degrees, Single px, Single py)
public void RotateRadians (Single radians, Single px, Single py)

```

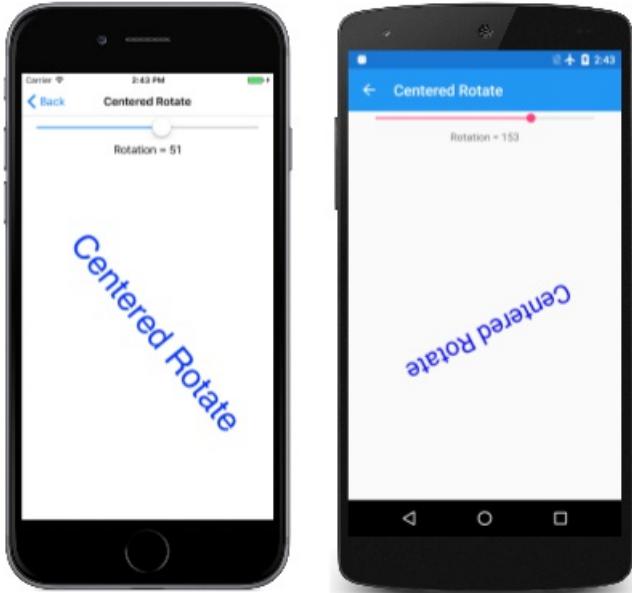
The **Centered Rotate** page is just like the **Basic Rotate** except that the expanded version of the `RotateDegrees` is used to set the center of rotation to the same point used to position the text:

```

using (SKPaint textPaint = new SKPaint
{
    Style = SKPaintStyle.Fill,
    Color = SKColors.Blue,
    TextAlign = SKTextAlign.Center,
    TextSize = 100
})
{
    canvas.RotateDegrees((float)rotateSlider.Value, info.Width / 2, info.Height / 2);
    canvas.DrawText(Title, info.Width / 2, info.Height / 2, textPaint);
}

```

Now the text rotates around the point used to position the text, which is the horizontal center of the text's baseline:



As with the centered version of the `Scale` method, the centered version of the `RotateDegrees` call is a shortcut. Here's the method:

```
RotateDegrees (degrees, px, py);
```

That call is equivalent to the following:

```
canvas.Translate(px, py);
canvas.RotateDegrees(degrees);
canvas.Translate(-px, -py);
```

You'll discover that you can sometimes combine `Translate` calls with `Rotate` calls. For example, here are the `RotateDegrees` and `DrawText` calls in the **Centered Rotate** page:

```
canvas.RotateDegrees((float)rotateSlider.Value, info.Width / 2, info.Height / 2);
canvas.DrawText(Title, info.Width / 2, info.Height / 2, textPaint);
```

The `RotateDegrees` call is equivalent to two `Translate` calls and a non-centered `RotateDegrees`:

```
canvas.Translate(info.Width / 2, info.Height / 2);
canvas.RotateDegrees((float)rotateSlider.Value);
canvas.Translate(-info.Width / 2, -info.Height / 2);
canvas.DrawText(Title, info.Width / 2, info.Height / 2, textPaint);
```

The `DrawText` call to display text at a particular location is equivalent to a `Translate` call for that location followed by `DrawText` at the point (0, 0):

```
canvas.Translate(info.Width / 2, info.Height / 2);
canvas.RotateDegrees((float)rotateSlider.Value);
canvas.Translate(-info.Width / 2, -info.Height / 2);
canvas.Translate(info.Width / 2, info.Height / 2);
canvas.DrawText(Title, 0, 0, textPaint);
```

The two consecutive `Translate` calls cancel each other out:

```
canvas.Translate(info.Width / 2, info.Height / 2);
canvas.RotateDegrees((float)rotateSlider.Value);
canvas.DrawText(Title, 0, 0, textPaint);
```

Conceptually, the two transforms are applied in the order opposite to how they appear in the code. The `DrawText` call displays the text in the upper-left corner of the canvas. The `RotateDegrees` call rotates that text relative to the upper-left corner. Then the `Translate` call moves the text to the center of the canvas.

There are usually several ways to combine rotation and translation. The [Rotated Text](#) page creates the following display:



Here's the `PaintSurface` handler of the [RotatedTextPage](#) class:

```

static readonly string text = "      ROTATE";
...
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    using (SKPaint textPaint = new SKPaint
    {
        Color = SKColors.Black,
        TextSize = 72
    })
    {
        float xCenter = info.Width / 2;
        float yCenter = info.Height / 2;

        SKRect textBounds = new SKRect();
        textPaint.MeasureText(text, ref textBounds);
        float yText = yCenter - textBounds.Height / 2 - textBounds.Top;

        for (int degrees = 0; degrees < 360; degrees += 30)
        {
            canvas.Save();
            canvas.RotateDegrees(degrees, xCenter, yCenter);
            canvas.DrawText(text, xCenter, yText, textPaint);
            canvas.Restore();
        }
    }
}

```

The `xCenter` and `yCenter` values indicate the center of the canvas. The `yText` value is a little offset from that. This value is the Y coordinate necessary to position the text so that it is truly vertically centered on the page. The `for` loop then sets a rotation based on the center of the canvas. The rotation is in increments of 30 degrees. The text is drawn using the `yText` value. The number of blanks before the word "ROTATE" in the `text` value was determined empirically to make the connection between these 12 text strings appear to be a dodecagon.

One way to simplify this code is to increment the rotation angle by 30 degrees each time through the loop after the `DrawText` call. This eliminates the need for calls to `Save` and `Restore`. Notice that the `degrees` variable is no longer used within the body of the `for` block:

```

for (int degrees = 0; degrees < 360; degrees += 30)
{
    canvas.DrawText(text, xCenter, yText, textPaint);
    canvas.RotateDegrees(30, xCenter, yCenter);
}

```

It's also possible to use the simple form of `RotateDegrees` by prefacing the loop with a call to `Translate` to move everything to the center of the canvas:

```

float yText = -textBounds.Height / 2 - textBounds.Top;

canvas.Translate(xCenter, yCenter);

for (int degrees = 0; degrees < 360; degrees += 30)
{
    canvas.DrawText(text, 0, yText, textPaint);
    canvas.RotateDegrees(30);
}

```

The modified `yText` calculation no longer incorporates `yCenter`. Now the `DrawText` call centers the text vertically at the top of the canvas.

Because the transforms are conceptually applied opposite to how they appear in code, it's often possible to begin with more global transforms, followed by more local transforms. This is often the easiest way to combine rotation and translation.

For example, suppose you want to draw a graphical object that rotates around its center much like a planet rotating on its axis. But you also want this object to revolve around the center of the screen much like a planet revolving around the sun.

You can do this by positioning the object in the upper-left corner of the canvas, and then using an animation to rotate it around that corner. Next, translate the object horizontally like an orbital radius. Now apply a second animated rotation, also around the origin. This makes the object revolve around the corner. Now translate to the center of the canvas.

Here's the `PaintSurface` handler that contains these transform calls in reverse order:

```

float revolveDegrees, rotateDegrees;
...
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    using (SKPaint fillPaint = new SKPaint
    {
        Style = SKPaintStyle.Fill,
        Color = SKColors.Red
    })
    {
        // Translate to center of canvas
        canvas.Translate(info.Width / 2, info.Height / 2);

        // Rotate around center of canvas
        canvas.RotateDegrees(revolveDegrees);

        // Translate horizontally
        float radius = Math.Min(info.Width, info.Height) / 3;
        canvas.Translate(radius, 0);

        // Rotate around center of object
        canvas.RotateDegrees(rotateDegrees);

        // Draw a square
        canvas.DrawRect(new SKRect(-50, -50, 50, 50), fillPaint);
    }
}

```

The `revolveDegrees` and `rotateDegrees` fields are animated. This program uses a different animation technique based on the `Xamarin.Forms Animation` class. (This class is described in [Chapter 22 of Creating Mobile Apps with Xamarin.Forms](#)) The `OnAppearing` override creates two `Animation` objects with callback methods and then calls `Commit` on them for an animation duration:

```
protected override void OnAppearing()
{
    base.OnAppearing();

    new Animation((value) => revolveDegrees = 360 * (float)value).
        Commit(this, "revolveAnimation", length: 10000, repeat: () => true);

    new Animation((value) =>
    {
        rotateDegrees = 360 * (float)value;
        canvasView.InvalidateSurface();
    }).Commit(this, "rotateAnimation", length: 1000, repeat: () => true);
}
```

The first `Animation` object animates `revolveDegrees` from 0 degrees to 360 degrees over 10 seconds. The second one animates `rotateDegrees` from 0 degrees to 360 degrees every 1 second and also invalidates the surface to generate another call to the `PaintSurface` handler. The `OnDisappearing` override cancels these two animations:

```
protected override void OnDisappearing()
{
    base.OnDisappearing();
    this.AbortAnimation("revolveAnimation");
    this.AbortAnimation("rotateAnimation");
}
```

The **Ugly Analog Clock** program (so called because a more attractive analog clock will be described in a later article) uses rotation to draw the minute and hour marks of the clock and to rotate the hands. The program draws the clock using an arbitrary coordinate system based on a circle that is centered at the point (0, 0) with a radius of 100. It uses translation and scaling to expand and center that circle on the page.

The `Translate` and `Scale` calls apply globally to the clock, so those are the first ones to be called following the initialization of the `SKPaint` objects:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    using (SKPaint strokePaint = new SKPaint())
    using (SKPaint fillPaint = new SKPaint())
    {
        strokePaint.Style = SKPaintStyle.Stroke;
        strokePaint.Color = SKColors.Black;
        strokePaint.StrokeCap = SKStrokeCap.Round;

        fillPaint.Style = SKPaintStyle.Fill;
        fillPaint.Color = SKColors.Gray;

        // Transform for 100-radius circle centered at origin
        canvas.Translate(info.Width / 2f, info.Height / 2f);
        canvas.Scale(Math.Min(info.Width / 200f, info.Height / 200f));
        ...
    }
}

```

There are 60 marks of two different sizes that must be drawn in a circle around the clock. The `DrawCircle` call draws that circle at the point (0, -90), which relative to the center of the clock corresponds to 12:00. The `RotateDegrees` call increments the rotation angle by 6 degrees after every tick mark. The `angle` variable is used solely to determine if a large circle or a small circle is drawn:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    ...
    // Hour and minute marks
    for (int angle = 0; angle < 360; angle += 6)
    {
        canvas.DrawCircle(0, -90, angle % 30 == 0 ? 4 : 2, fillPaint);
        canvas.RotateDegrees(6);
    }
    ...
}

```

Finally, the `PaintSurface` handler obtains the current time and calculates rotation degrees for the hour, minute, and second hands. Each hand is drawn in the 12:00 position so that the rotation angle is relative to that:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    ...
    DateTime dateTime = DateTime.Now;

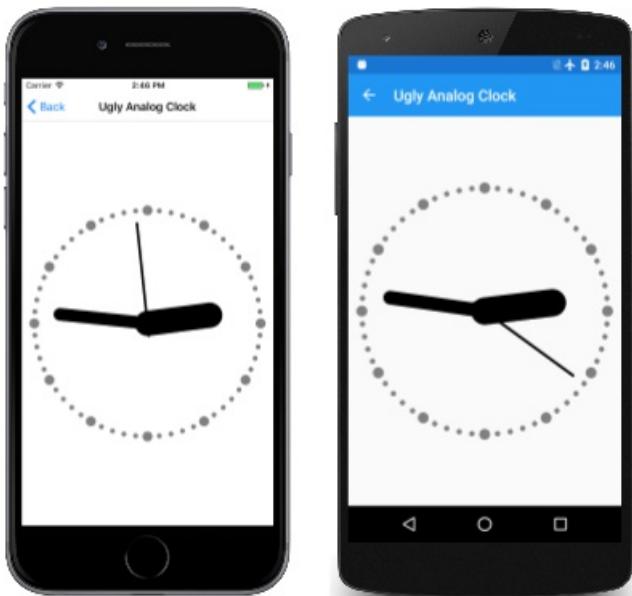
    // Hour hand
    strokePaint.StrokeWidth = 20;
    canvas.Save();
    canvas.RotateDegrees(30 * dateTime.Hour + dateTime.Minute / 2f);
    canvas.DrawLine(0, 0, 0, -50, strokePaint);
    canvas.Restore();

    // Minute hand
    strokePaint.StrokeWidth = 10;
    canvas.Save();
    canvas.RotateDegrees(6 * dateTime.Minute + dateTime.Second / 10f);
    canvas.DrawLine(0, 0, 0, -70, strokePaint);
    canvas.Restore();

    // Second hand
    strokePaint.StrokeWidth = 2;
    canvas.Save();
    canvas.RotateDegrees(6 * dateTime.Second);
    canvas.DrawLine(0, 10, 0, -80, strokePaint);
    canvas.Restore();
}
}

```

The clock is certainly functional although the hands are rather crude:



For a more attractive clock, see the article [SVG Path Data in SkiaSharp](#).

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

The Skew Transform

3/5/2021 • 5 minutes to read • [Edit Online](#)

 [Download the sample](#)

See how the skew transform can create tilted graphical objects in SkiaSharp

In SkiaSharp, the skew transform tilts graphical objects, such as the shadow in this image:



The skew turns a rectangle into a parallelogram, but a skewed ellipse is still an ellipse.

Although Xamarin.Forms defines properties for translation, scaling, and rotations, there is no corresponding property in Xamarin.Forms for skew.

The `Skew` method of `SKCanvas` accepts two arguments for horizontal skew and vertical skew:

```
public void Skew (Single xSkew, Single ySkew)
```

A second `Skew` method combines those arguments in a single `SKPoint` value:

```
public void Skew (SKPoint skew)
```

However, it's unlikely that you'll be using either of these two methods in isolation.

The [Skew Experiment](#) page lets you experiment with skew values that range between -10 and 10. A text string is positioned in the upper-left corner of the page, with skew values obtained from two `Slider` elements. Here is the `PaintSurface` handler in the `SkewExperimentPage` class:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    using (SKPaint textPaint = new SKPaint
    {
        Style = SKPaintStyle.Fill,
        Color = SKColors.Blue,
        TextSize = 200
    })
    {
        string text = "SKEW";
        SKRect textBounds = new SKRect();
        textPaint.MeasureText(text, ref textBounds);

        canvas.Skew((float)xSkewSlider.Value, (float)ySkewSlider.Value);
        canvas.DrawText(text, 0, -textBounds.Top, textPaint);
    }
}

```

Values of the `xSkew` argument shift the bottom of the text right for positive values or left for negative values.

Values of `ySkew` shift the right of the text down for positive values or up for negative values:



If the `xSkew` value is the negative of the `ySkew` value, the result is rotation, but also scaled somewhat.

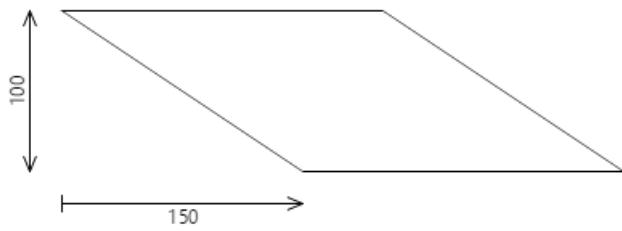
The transform formulas are as follows:

$$x' = x + xSkew \cdot y$$

$$y' = ySkew \cdot x + y$$

For example, for a positive `xSkew` value, the transformed `x'` value increases as `y` increases. That's what causes the tilt.

If a triangle 200 pixels wide and 100 pixels high is positioned with its upper-left corner at the point (0, 0) and is rendered with an `xSkew` value of 1.5, the following parallelogram results:



The coordinates of the bottom edge have `y` values of 100, so it is shifted 150 pixels to the right.

For non-zero values of `xSkew` or `ySkew`, only the point (0, 0) remains the same. That point can be considered the center of skewing. If you need the center of skewing to be something else (which is usually the case), there is no `Skew` method that provides that. You'll need to explicitly combine `Translate` calls with the `Skew` call. To center the skewing at `px` and `py`, make the following calls:

```
canvas.Translate(px, py);
canvas.Skew(xSkew, ySkew);
canvas.Translate(-px, -py);
```

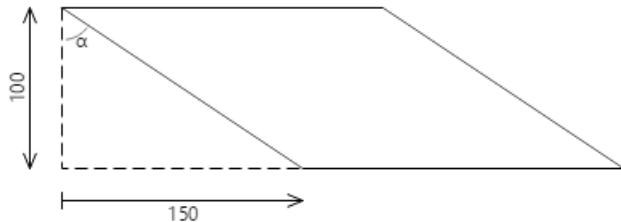
The composite transform formulas are:

$$x' = x + xSkew \cdot (y - py)$$

$$y' = ySkew \cdot (x - px) + y$$

If `ySkew` is zero, then the `px` value is not used. The value is irrelevant, and similarly for `ySkew` and `py`.

You might feel more comfortable specifying skew as an angle of tilt, such as the angle α in this diagram:



The ratio of the 150-pixel shift to the 100-pixel vertical is the tangent of that angle, in this example 56.3 degrees.

The XAML file of the **Skew Angle Experiment** page is similar to the **Skew Angle** page except that the `Slider` elements range from -90 degrees to 90 degrees. The `SkewAngleExperiment` code-behind file centers the text on the page and uses `Translate` to set a center of skewing to the center of the page. A short `SkewDegrees` method at the bottom of the code converts angles to skew values:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    using (SKPaint textPaint = new SKPaint
    {
        Style = SKPaintStyle.Fill,
        Color = SKColors.Blue,
        TextSize = 200
    })
    {
        float xCenter = info.Width / 2;
        float yCenter = info.Height / 2;

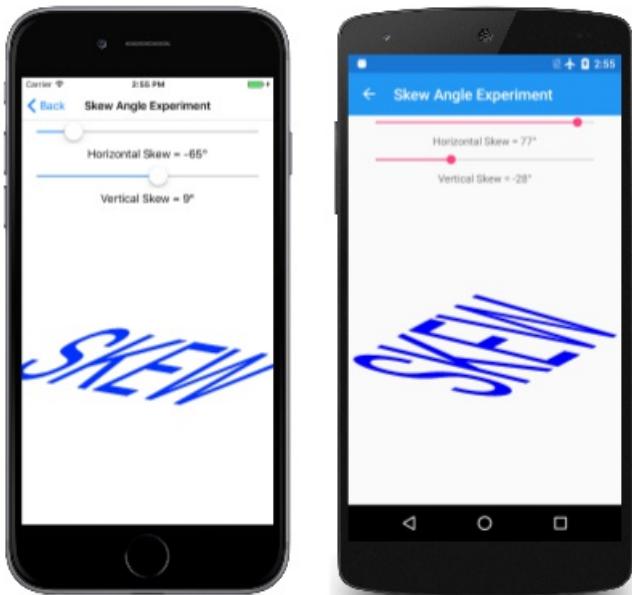
        string text = "SKEW";
        SKRect textBounds = new SKRect();
        textPaint.MeasureText(text, ref textBounds);
        float xText = xCenter - textBounds.MidX;
        float yText = yCenter - textBounds.MidY;

        canvas.Translate(xCenter, yCenter);
        SkewDegrees(canvas, xSkewSlider.Value, ySkewSlider.Value);
        canvas.Translate(-xCenter, -yCenter);
        canvas.DrawText(text, xText, yText, textPaint);
    }
}

void SkewDegrees(SKCanvas canvas, double xDegrees, double yDegrees)
{
    canvas.Skew((float)Math.Tan(Math.PI * xDegrees / 180),
               (float)Math.Tan(Math.PI * yDegrees / 180));
}

```

As an angle approaches positive or negative 90 degrees, the tangent approaches infinity, but angles up to about 80 degrees or so are usable:



A small negative horizontal skew can mimic oblique or italic text, as the **Oblique Text** page demonstrates. The [ObliqueTextPage](#) class shows how it's done:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    using (SKPaint textPaint = new SKPaint()
    {
        Style = SKPaintStyle.Fill,
        Color = SKColors.Maroon,
        TextAlign = SKTextAlign.Center,
        TextSize = info.Width / 8      // empirically determined
    })
    {
        canvas.Translate(info.Width / 2, info.Height / 2);
        SkewDegrees(canvas, -20, 0);
        canvas.DrawText("Oblique Text", 0, 0, textPaint);
    }
}

void SkewDegrees(SKCanvas canvas, double xDegrees, double yDegrees)
{
    canvas.Skew((float)Math.Tan(Math.PI * xDegrees / 180),
                (float)Math.Tan(Math.PI * yDegrees / 180));
}

```

The `TextAlign` property of `SKPaint` is set to `Center`. Without any transforms, the `DrawText` call with coordinates of (0, 0) would position the text with the horizontal center of the baseline at the upper-left corner. The `SkewDegrees` skews the text horizontally 20 degrees relative to the baseline. The `Translate` call moves the horizontal center of the text's baseline to the center of the canvas:



The [Skew Shadow Text](#) page demonstrates how to use a combination of a 45-degree skew and vertical scale to make a text shadow that tilts away from the text. Here's the pertinent part of the `PaintSurface` handler:

```

using (SKPaint textPaint = new SKPaint())
{
    textPaint.Style = SKPaintStyle.Fill;
    textPaint.TextSize = info.Width / 6; // empirically determined

    // Common to shadow and text
    string text = "Shadow";
    float xText = 20;
    float yText = info.Height / 2;

    // Shadow
    textPaint.Color = SKColors.LightGray;
    canvas.Save();
    canvas.Translate(xText, yText);
    canvas.Skew((float)Math.Tan(-Math.PI / 4), 0);
    canvas.Scale(1, 3);
    canvas.Translate(-xText, -yText);
    canvas.DrawText(text, xText, yText, textPaint);
    canvas.Restore();

    // Text
    textPaint.Color = SKColors.Blue;
    canvas.DrawText(text, xText, yText, textPaint);
}

```

The shadow is displayed first and then the text:



The vertical coordinate passed to the `DrawText` method indicates the position of the text relative to the baseline. That is the same vertical coordinate used for the center of skewing. This technique will not work if the text string contains descenders. For example, substitute the word "quirky" for "Shadow" and here's the result:



The shadow and text are still aligned at the baseline, but the effect just looks wrong. To fix it, you need to obtain the text bounds:

```
SKRect textBounds = new SKRect();
textPaint.MeasureText(text, ref textBounds);
```

The `Translate` calls need to be adjusted by the height of the descenders:

```
canvas.Translate(xText, yText + textBounds.Bottom);
canvas.Skew((float)Math.Tan(-Math.PI / 4), 0);
canvas.Scale(1, 3);
canvas.Translate(-xText, -yText - textBounds.Bottom);
```

Now the shadow extends from the bottom of those descenders:



Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Matrix Transforms in SkiaSharp

3/5/2021 • 19 minutes to read • [Edit Online](#)

 [Download the sample](#)

Dive deeper into SkiaSharp transforms with the versatile transform matrix

All the transforms applied to the `skCanvas` object are consolidated in a single instance of the `SKMatrix` structure. This is a standard 3-by-3 transform matrix similar to those in all modern 2D graphics systems.

As you've seen, you can use transforms in SkiaSharp without knowing about the transform matrix, but the transform matrix is important from a theoretical perspective, and it is crucial when using transforms to modify paths or for handling complex touch input, both of which are demonstrated in this article and the next.



The current transform matrix applied to the `skCanvas` is available at any time by accessing the read-only `TotalMatrix` property. You can set a new transform matrix using the `SetMatrix` method, and you can restore that transform matrix to default values by calling `ResetMatrix`.

The only other `skCanvas` member that directly works with the canvas's matrix transform is `Concat` which concatenates two matrices by multiplying them together.

The default transform matrix is the identity matrix and consists of 1's in the diagonal cells and 0's everywhere else:

	1	0	0	
	0	1	0	
	0	0	1	

You can create an identity matrix using the static `SKMatrix.MakeIdentity` method:

```
SKMatrix matrix = SKMatrix.MakeIdentity();
```

The `SKMatrix` default constructor does *not* return an identity matrix. It returns a matrix with all of the cells set to zero. Do not use the `SKMatrix` constructor unless you plan to set those cells manually.

When SkiaSharp renders a graphical object, each point (x, y) is effectively converted to a 1-by-3 matrix with a 1 in the third column:

$$\begin{vmatrix} x & y & 1 \end{vmatrix}$$

This 1-by-3 matrix represents a three-dimensional point with the Z coordinate set to 1. There are mathematical reasons (discussed later) why a two-dimensional matrix transform requires working in three dimensions. You can think of this 1-by-3 matrix as representing a point in a 3D coordinate system, but always on the 2D plane where Z equals 1.

This 1-by-3 matrix is then multiplied by the transform matrix, and the result is the point rendered on the canvas:

$$\begin{vmatrix} x & y & 1 \end{vmatrix} \times \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} x' & y' & z' \end{vmatrix}$$

Using standard matrix multiplication, the converted points are as follows:

$$x' = x$$

$$y' = y$$

$$z' = 1$$

That's the default transform.

When the `Translate` method is called on the `SKCanvas` object, the `tx` and `ty` arguments to the `Translate` method become the first two cells in the third row of the transform matrix:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ tx & ty & 1 \end{vmatrix}$$

The multiplication is now as follows:

$$\begin{vmatrix} x & y & 1 \end{vmatrix} \times \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ tx & ty & 1 \end{vmatrix} = \begin{vmatrix} x' & y' & z' \end{vmatrix}$$

Here are the transform formulas:

$$x' = x + tx$$

$$y' = y + ty$$

Scaling factors have a default value of 1. When you call the `Scale` method on a new `SKCanvas` object, the resultant transform matrix contains the `sx` and `sy` arguments in the diagonal cells:

$$\begin{vmatrix} & \begin{vmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{vmatrix} \\ \begin{vmatrix} x & y & 1 \end{vmatrix} \times & = \begin{vmatrix} x' & y' & z' \end{vmatrix} \end{vmatrix}$$

The transform formulas are as follows:

$$x' = sx \cdot x$$

$$y' = sy \cdot y$$

The transform matrix after calling `Skew` contains the two arguments in the matrix cells adjacent to the scaling factors:

$$\begin{vmatrix} & \begin{vmatrix} 1 & ySkew & 0 \\ xSkew & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix} \\ \begin{vmatrix} x & y & 1 \end{vmatrix} \times & = \begin{vmatrix} x' & y' & z' \end{vmatrix} \end{vmatrix}$$

The transform formulas are:

$$x' = x + xSkew \cdot y$$

$$y' = ySkew \cdot x + y$$

For a call to `RotateDegrees` or `RotateRadians` for an angle of α , the transform matrix is as follows:

$$\begin{vmatrix} & \begin{vmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{vmatrix} \\ \begin{vmatrix} x & y & 1 \end{vmatrix} \times & = \begin{vmatrix} x' & y' & z' \end{vmatrix} \end{vmatrix}$$

Here are the transform formulas:

$$x' = \cos(\alpha) \cdot x - \sin(\alpha) \cdot y$$

$$y' = \sin(\alpha) \cdot x + \cos(\alpha) \cdot y$$

When α is 0 degrees, it's the identity matrix. When α is 180 degrees, the transform matrix is as follows:

$$\begin{vmatrix} & \begin{vmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{vmatrix} \\ \end{vmatrix}$$

A 180-degree rotation is equivalent to flipping an object horizontally and vertically, which is also accomplished by setting scale factors of -1.

All these types of transforms are classified as *affine* transforms. Affine transforms never involve the third column of the matrix, which remains at the default values of 0, 0, and 1. The article [Non-Affine Transforms](#) discusses non-affine transforms.

Matrix Multiplication

One significant advantage with using the transform matrix is that composite transforms can be obtained by matrix multiplication, which is often referred to in the SkiaSharp documentation as *concatenation*. Many of the transform-related methods in `SKCanvas` refer to "pre-concatenation" or "pre-concat." This refers to the order of multiplication, which is important because matrix multiplication is not commutative.

For example, the documentation for the `Translate` method says that it "Pre-concats the current matrix with the specified translation," while the documentation for the `Scale` method says that it "Pre-concats the current

matrix with the specified scale."

This means that the transform specified by the method call is the multiplier (the left-hand operand) and the current transform matrix is the multiplicand (the right-hand operand).

Suppose that `Translate` is called followed by `Scale`:

```
canvas.Translate(tx, ty);
canvas.Scale(sx, sy);
```

The `Scale` transform is multiplied by the `Translate` transform for the composite transform matrix:

$$\begin{vmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ tx & ty & 1 \end{vmatrix} = \begin{vmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ tx & ty & 1 \end{vmatrix}$$

`Scale` could be called before `Translate` like this:

```
canvas.Scale(sx, sy);
canvas.Translate(tx, ty);
```

In that case, the order of the multiplication is reversed, and the scaling factors are effectively applied to the translation factors:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ tx & ty & 1 \end{vmatrix} \times \begin{vmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ tx \cdot sx & ty \cdot sy & 1 \end{vmatrix}$$

Here is the `Scale` method with a pivot point:

```
canvas.Scale(sx, sy, px, py);
```

This is equivalent to the following translate and scale calls:

```
canvas.Translate(px, py);
canvas.Scale(sx, sy);
canvas.Translate(-px, -py);
```

The three transform matrices are multiplied in reverse order from how the methods appear in code:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -px & -py & 1 \end{vmatrix} \times \begin{vmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ px & py & 1 \end{vmatrix} = \begin{vmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ px - px \cdot sx & py - py \cdot sy & 1 \end{vmatrix}$$

The SKMatrix Structure

The `SKMatrix` structure defines nine read/write properties of type `float` corresponding to the nine cells of the transform matrix:

ScaleX	SkewY	Persp0
SkewX	ScaleY	Persp1
TransX	TransY	Persp2

`SKMatrix` also defines a property named `Values` of type `float[]`. This property can be used to set or obtain the nine values in one shot in the order `ScaleX`, `SkewX`, `TransX`, `SkewY`, `ScaleY`, `TransY`, `Persp0`, `Persp1`, and `Persp2`.

The `Persp0`, `Persp1`, and `Persp2` cells are discussed in the article [Non-Affine Transforms](#). If these cells have their default values of 0, 0, and 1, then the transform is multiplied by a coordinate point like this:

$$\begin{vmatrix} & \begin{vmatrix} ScaleX & SkewY & 0 \\ SkewX & ScaleY & 0 \\ TransX & TransY & 1 \end{vmatrix} \\ \begin{vmatrix} x & y & 1 \end{vmatrix} \times & = \begin{vmatrix} x' & y' & z' \end{vmatrix} \end{vmatrix}$$

$$x' = ScaleX \cdot x + SkewX \cdot y + TransX$$

$$y' = SkewX \cdot x + ScaleY \cdot y + TransY$$

$$z' = 1$$

This is the complete two-dimensional affine transform. The affine transform preserves parallel lines, which means that a rectangle is never transformed into anything other than a parallelogram.

The `SKMatrix` structure defines several static methods to create `SKMatrix` values. These all return `SKMatrix` values:

- [MakeTranslation](#)
- [MakeScale](#)
- [MakeScale](#) with a pivot point
- [MakeRotation](#) for an angle in radians
- [MakeRotation](#) for an angle in radians with a pivot point
- [MakeRotationDegrees](#)
- [MakeRotationDegrees](#) with a pivot point
- [MakeSkew](#)

`SKMatrix` also defines several static methods that concatenate two matrices, which means to multiply them. These methods are named `Concat`, `PostConcat`, and `PreConcat`, and there are two versions of each. These methods have no return values; instead, they reference existing `SKMatrix` values through `ref` arguments. In the following example, `A`, `B`, and `R` (for "result") are all `SKMatrix` values.

The two `Concat` methods are called like this:

```
SKMatrix.Concat(ref R, A, B);
SKMatrix.Concat(ref R, ref A, ref B);
```

These perform the following multiplication:

$$R = B \times A$$

The other methods have only two parameters. The first parameter is modified, and on return from the method call, contains the product of the two matrices. The two `PostConcat` methods are called like this:

```
SKMatrix.PostConcat(ref A, B);
SKMatrix.PostConcat(ref A, ref B);
```

These calls perform the following operation:

```
A = A × B
```

The two `PreConcat` methods are similar:

```
SKMatrix.PreConcat(ref A, B);  
SKMatrix.PreConcat(ref A, ref B);
```

These calls perform the following operation:

```
A = B × A
```

The versions of these methods with all `ref` arguments are slightly more efficient in calling the underlying implementations, but it might be confusing to someone reading your code and assuming that anything with a `ref` argument is modified by the method. Moreover, it's often convenient to pass an argument that is a result of one of the `Make` methods, for example:

```
SKMatrix result;  
SKMatrix.Concat(result, SKMatrix.MakeTranslation(100, 100),  
                SKMatrix.MakeScale(3, 3));
```

This creates the following matrix:

3	0	0
0	3	0
100	100	1

This is the scale transform multiplied by the translate transform. In this particular case, the `SKMatrix` structure provides a shortcut with a method named `SetScaleTranslate`:

```
SKMatrix R = new SKMatrix();  
R.SetScaleTranslate(3, 3, 100, 100);
```

This is one of the few times when it's safe to use the `SKMatrix` constructor. The `SetScaleTranslate` method sets all nine cells of the matrix. It is also safe to use the `SKMatrix` constructor with the static `Rotate` and `RotateDegrees` methods:

```
SKMatrix R = new SKMatrix();  
  
SKMatrix.Rotate(ref R, radians);  
  
SKMatrix.Rotate(ref R, radians, px, py);  
  
SKMatrix.RotateDegrees(ref R, degrees);  
  
SKMatrix.RotateDegrees(ref R, degrees, px, py);
```

These methods do *not* concatenate a rotate transform to an existing transform. The methods set all the cells of the matrix. They are functionally identical to the `MakeRotation` and `MakeRotationDegrees` methods except that they don't instantiate the `SKMatrix` value.

Suppose you have an `SKPath` object that you want to display, but you would prefer that it have a somewhat different orientation, or a different center point. You can modify all the coordinates of that path by calling the `Transform` method of `SKPath` with an `SKMatrix` argument. The [Path Transform](#) page demonstrates how to do this. The `PathTransform` class references the `HendecagramPath` object in a field but uses its constructor to apply a

transform to that path:

```
public class PathTransformPage : ContentPage
{
    SKPath transformedPath = HendecagramArrayPage.HendecagramPath;

    public PathTransformPage()
    {
        Title = "Path Transform";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;

        SKMatrix matrix = SKMatrix.MakeScale(3, 3);
        SKMatrix.PostConcat(ref matrix, SKMatrix.MakeRotationDegrees(360f / 22));
        SKMatrix.PostConcat(ref matrix, SKMatrix.MakeTranslation(300, 300));

        transformedPath.Transform(matrix);
    }
    ...
}
```

The `HendecagramPath` object has a center at (0, 0), and the 11 points of the star extend outward from that center by 100 units in all directions. This means that the path has both positive and negative coordinates. The **Path Transform** page prefers to work with a star three times as large, and with all positive coordinates. Moreover, it doesn't want one point of the star to point straight up. It wants instead for one point of the star to point straight down. (Because the star has 11 points, it can't have both.) This requires rotating the star by 360 degrees divided by 22.

The constructor builds an `SKMatrix` object from three separate transforms using the `PostConcat` method with the following pattern, where A, B, and C are instances of `SKMatrix`:

```
SKMatrix matrix = A;
SKMatrix.PostConcat(ref A, B);
SKMatrix.PostConcat(ref A, C);
```

This is a series of successive multiplications, so the result is as follows:

`A × B × C`

The consecutive multiplications aid in understanding what each transform does. The scale transform increases the size of the path coordinates by a factor of 3, so the coordinates range from -300 to 300. The rotate transform rotates the star around its origin. The translate transform then shifts it by 300 pixels right and down, so all the coordinates become positive.

There are other sequences that produce the same matrix. Here's another one:

```
SKMatrix matrix = SKMatrix.MakeRotationDegrees(360f / 22);
SKMatrix.PostConcat(ref matrix, SKMatrix.MakeTranslation(100, 100));
SKMatrix.PostConcat(ref matrix, SKMatrix.MakeScale(3, 3));
```

This rotates the path around its center first, and then translates it 100 pixels to the right and down so all the coordinates are positive. The star is then increased in size relative to its new upper-left corner, which is the point (0, 0).

The `PaintSurface` handler can simply render this path:

```

public class PathTransformPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

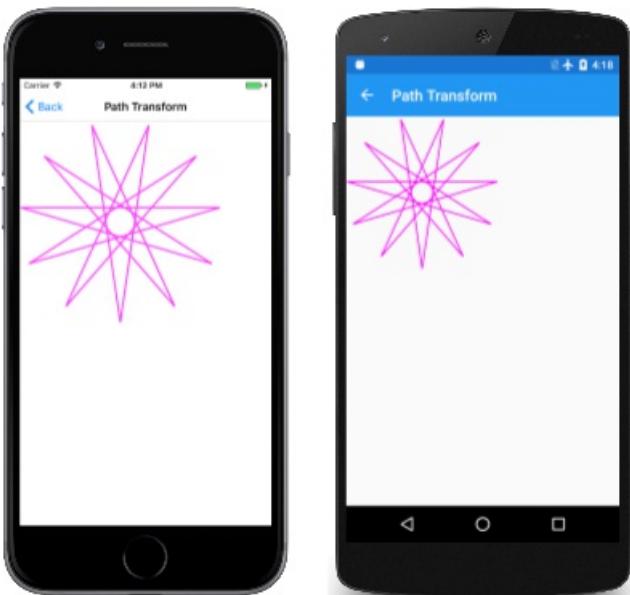
        canvas.Clear();

        using (SKPaint paint = new SKPaint())
        {
            paint.Style = SKPaintStyle.Stroke;
            paint.Color = SKColors.Magenta;
            paint.StrokeWidth = 5;

            canvas.DrawPath(transformedPath, paint);
        }
    }
}

```

It appears in the upper-left corner of the canvas:



The constructor of this program applies the matrix to the path with the following call:

```
transformedPath.Transform(matrix);
```

The path does *not* retain this matrix as a property. Instead, it applies the transform to all of the coordinates of the path. If `Transform` is called again, the transform is applied again, and the only way you can go back is by applying another matrix that undoes the transform. Fortunately, the `SKMatrix` structure defines a `TryInvert` method that obtains the matrix that reverses a given matrix:

```

SKMatrix inverse;
bool success = matrix.TryInverse(out inverse);

```

The method is called `TryInverse` because not all matrices are invertible, but a non-invertible matrix is not likely to be used for a graphics transform.

You can also apply a matrix transform to an `SKPoint` value, an array of points, an `SKRect`, or even just a single number within your program. The `SKMatrix` structure supports these operations with a collection of methods that begin with the word `Map`, such as these:

```
SKPoint transformedPoint = matrix.MapPoint(point);

SKPoint transformedPoint = matrix.MapPoint(x, y);

SKPoint[] transformedPoints = matrix.MapPoints(pointArray);

float transformedValue = matrix.MapRadius(floatValue);

SKRect transformedRect = matrix.MapRect(rect);
```

If you use that last method, keep in mind that the `SKRect` structure is not capable of representing a rotated rectangle. The method only makes sense for an `SKMatrix` value representing translation and scaling.

Interactive Experimentation

One way to get a feel for the affine transform is by interactively moving three corners of a bitmap around the screen and seeing what transform results. This is the idea behind the [Show Affine Matrix](#) page. This page requires two other classes that are also used in other demonstrations:

The `TouchPoint` class displays a translucent circle that can be dragged around the screen. `TouchPoint` requires that an `SKCanvasView` or an element that is a parent of an `SKCanvasView` have the `TouchEffect` attached. Set the `Capture` property to `true`. In the `TouchAction` event handler, the program must call the `ProcessTouchEvent` method in `TouchPoint` for each `TouchPoint` instance. The method returns `true` if the touch event resulted in the touch point moving. Also, the `PaintSurface` handler must call the `Paint` method in each `TouchPoint` instance, passing to it the `SKCanvas` object.

`TouchPoint` demonstrates a common way that a SkiaSharp visual can be encapsulated in a separate class. The class can define properties for specifying characteristics of the visual, and a method named `Paint` with an `SKCanvas` argument can render it.

The `center` property of `TouchPoint` indicates the location of the object. This property can be set to initialize the location; the property changes when the user drags the circle around the canvas.

The [Show Affine Matrix Page](#) also requires the `MatrixDisplay` class. This class displays the cells of an `SKMatrix` object. It has two public methods: `Measure` to obtain the dimensions of the rendered matrix, and `Paint` to display it. The class contains a `MatrixPaint` property of type `SKPaint` that can be replaced for a different font size or color.

The `ShowAffineMatrixPage.xaml` file instantiates the `SKCanvasView` and attaches a `TouchEffect`. The `ShowAffineMatrixPage.xaml.cs` code-behind file creates three `TouchPoint` objects and then sets them to positions corresponding to three corners of a bitmap that it loads from an embedded resource:

```

public partial class ShowAffineMatrixPage : ContentPage
{
    SKMatrix matrix;
    SKBitmap bitmap;
    SKSize bitmapSize;

    TouchPoint[] touchPoints = new TouchPoint[3];

    MatrixDisplay matrixDisplay = new MatrixDisplay();

    public ShowAffineMatrixPage()
    {
        InitializeComponent();

        string resourceId = "SkiaSharpFormsDemos.Media.SeatedMonkey.jpg";
        Assembly assembly = GetType().GetTypeInfo().Assembly;

        using (Stream stream = assembly.GetManifestResourceStream(resourceId))
        {
            bitmap = SKBitmap.Decode(stream);
        }

        touchPoints[0] = new TouchPoint(100, 100); // upper-left corner
        touchPoints[1] = new TouchPoint(bitmap.Width + 100, 100); // upper-right corner
        touchPoints[2] = new TouchPoint(100, bitmap.Height + 100); // lower-left corner

        bitmapSize = new SKSize(bitmap.Width, bitmap.Height);
        matrix = ComputeMatrix(bitmapSize, touchPoints[0].Center,
                               touchPoints[1].Center,
                               touchPoints[2].Center);
    }
    ...
}

```

An affine matrix is uniquely defined by three points. The three `TouchPoint` objects correspond to the upper-left, upper-right, and lower-left corners of the bitmap. Because an affine matrix is only capable of transforming a rectangle into a parallelogram, the fourth point is implied by the other three. The constructor concludes with a call to `ComputeMatrix`, which calculates the cells of an `SKMatrix` object from these three points.

The `TouchAction` handler calls the `ProcessTouchEvent` method of each `TouchPoint`. The `scale` value converts from Xamarin.Forms coordinates to pixels:

```

public partial class ShowAffineMatrixPage : ContentPage
{
    ...
    void OnTouchEffectAction(object sender, TouchEventArgs args)
    {
        bool touchPointMoved = false;

        foreach (TouchPoint touchPoint in touchPoints)
        {
            float scale = canvasView.CanvasSize.Width / (float)canvasView.Width;
            SKPoint point = new SKPoint(scale * (float)args.Location.X,
                                         scale * (float)args.Location.Y);
            touchPointMoved |= touchPoint.ProcessTouchEvent(args.Id, args.Type, point);
        }

        if (touchPointMoved)
        {
            matrix = ComputeMatrix(bitmapSize, touchPoints[0].Center,
                                   touchPoints[1].Center,
                                   touchPoints[2].Center);
            canvasView.InvalidateSurface();
        }
    }
    ...
}

```

If any `TouchPoint` has moved, then the method calls `ComputeMatrix` again and invalidates the surface.

The `computeMatrix` method determines the matrix implied by those three points. The matrix called `A` transforms a one-pixel square rectangle into a parallelogram based on the three points, while the scale transform called `S` scales the bitmap to a one-pixel square rectangle. The composite matrix is `S × A`:

```

public partial class ShowAffineMatrixPage : ContentPage
{
    ...
    static SKMatrix ComputeMatrix(SKSize size, SKPoint ptUL, SKPoint ptUR, SKPoint ptLL)
    {
        // Scale transform
        SKMatrix S = SKMatrix.MakeScale(1 / size.Width, 1 / size.Height);

        // Affine transform
        SKMatrix A = new SKMatrix
        {
            ScaleX = ptUR.X - ptUL.X,
            SkewY = ptUR.Y - ptUL.Y,
            SkewX = ptLL.X - ptUL.X,
            ScaleY = ptLL.Y - ptUL.Y,
            TransX = ptUL.X,
            TransY = ptUL.Y,
            Persp2 = 1
        };

        SKMatrix result = SKMatrix.MakeIdentity();
        SKMatrix.Concat(ref result, A, S);
        return result;
    }
    ...
}

```

Finally, the `PaintSurface` method renders the bitmap based on that matrix, displays the matrix at the bottom of the screen, and renders the touch points at the three corners of the bitmap:

```

public partial class ShowAffineMatrixPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Display the bitmap using the matrix
        canvas.Save();
        canvas.SetMatrix(matrix);
        canvas.DrawBitmap(bitmap, 0, 0);
        canvas.Restore();

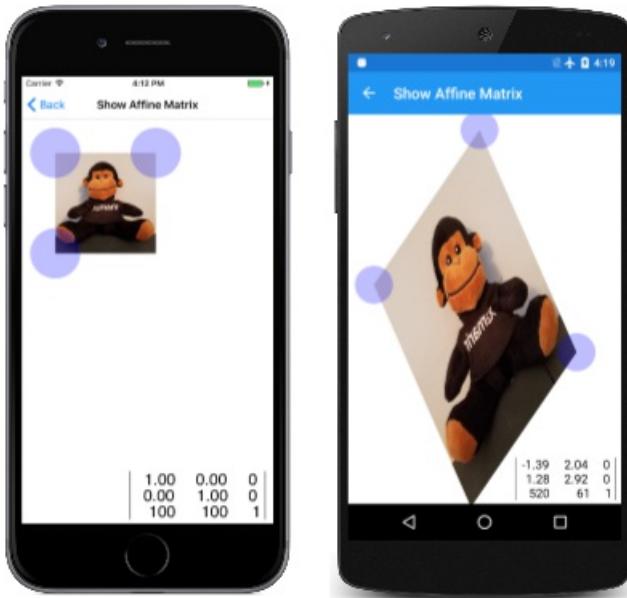
        // Display the matrix in the lower-right corner
        SKSize matrixSize = matrixDisplay.Measure(matrix);

        matrixDisplay.Paint(canvas, matrix,
            new SKPoint(info.Width - matrixSize.Width,
                info.Height - matrixSize.Height));

        // Display the touchpoints
        foreach (TouchPoint touchPoint in touchPoints)
        {
            touchPoint.Paint(canvas);
        }
    }
}

```

The iOS screen below shows the bitmap when the page is first loaded, while the two other screens show it after some manipulation:



Although it seems as if the touch points drag the corners of the bitmap, that's only an illusion. The matrix calculated from the touch points transforms the bitmap so that the corners coincide with the touch points.

It is more natural for users to move, resize, and rotate bitmaps not by dragging the corners, but by using one or two fingers directly on the object to drag, pinch, and rotate. This is covered in the next article [Touch Manipulation](#).

The Reason for the 3-by-3 Matrix

It might be expected that a two-dimensional graphics system would require only a 2-by-2 transform matrix:

$$\begin{vmatrix} x & y \end{vmatrix} \times \begin{vmatrix} \text{ScaleX} & \text{SkewY} \\ \text{SkewX} & \text{ScaleY} \end{vmatrix} = \begin{vmatrix} x' & y' \end{vmatrix}$$

This works for scaling, rotation, and even skewing, but it is not capable of the most basic of transforms, which is translation.

The problem is that the 2-by-2 matrix represents a *linear* transform in two dimensions. A linear transform preserves some basic arithmetic operations, but one of the implications is that a linear transform never alters the point (0, 0). A linear transform makes translation impossible.

In three dimensions, a linear transform matrix looks like this:

$$\begin{vmatrix} x & y & z \end{vmatrix} \times \begin{vmatrix} \text{ScaleX} & \text{SkewYX} & \text{SkewZX} \\ \text{SkewXY} & \text{ScaleY} & \text{SkewZY} \\ \text{SkewXZ} & \text{SkewYZ} & \text{ScaleZ} \end{vmatrix} = \begin{vmatrix} x' & y' & z' \end{vmatrix}$$

The cell labeled `SkewXY` means that the value skews the X coordinate based on values of Y; the cell `SkewXZ` means that the value skews the X coordinate based on values of Z; and values skew similarly for the other `skew` cells.

It's possible to restrict this 3D transform matrix to a two-dimensional plane by setting `SkewZX` and `SkewZY` to 0, and `scaleZ` to 1:

$$\begin{vmatrix} x & y & z \end{vmatrix} \times \begin{vmatrix} \text{ScaleX} & \text{SkewYX} & 0 \\ \text{SkewXY} & \text{ScaleY} & 0 \\ \text{SkewXZ} & \text{SkewYZ} & 1 \end{vmatrix} = \begin{vmatrix} x' & y' & z' \end{vmatrix}$$

If the two-dimensional graphics are drawn entirely on the plane in 3D space where Z equals 1, the transform multiplication looks like this:

$$\begin{vmatrix} x & y & 1 \end{vmatrix} \times \begin{vmatrix} \text{ScaleX} & \text{SkewYX} & 0 \\ \text{SkewXY} & \text{ScaleY} & 0 \\ \text{SkewXZ} & \text{SkewYZ} & 1 \end{vmatrix} = \begin{vmatrix} x' & y' & 1 \end{vmatrix}$$

Everything stays on the two-dimensional plane where Z equals 1, but the `SkewXZ` and `SkewYZ` cells effectively become two-dimensional translation factors.

This is how a three-dimensional linear transform serves as a two-dimensional non-linear transform. (By analogy, transforms in 3D graphics are based on a 4-by-4 matrix.)

The `SKMatrix` structure in SkiaSharp defines properties for that third row:

$$\begin{vmatrix} x & y & 1 \end{vmatrix} \times \begin{vmatrix} \text{ScaleX} & \text{SkewY} & \text{Persp0} \\ \text{SkewX} & \text{ScaleY} & \text{Persp1} \\ \text{TransX} & \text{TransY} & \text{Persp2} \end{vmatrix} = \begin{vmatrix} x' & y' & z' \end{vmatrix}$$

Non-zero values of `Persp0` and `Persp1` result in transforms that move objects off the two-dimensional plane where Z equals 1. What happens when those objects are moved back to that plane is covered in the article on [Non-Affine Transforms](#).

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Touch Manipulations

3/5/2021 • 29 minutes to read • [Edit Online](#)



[Download the sample](#)

Use matrix transforms to implement touch dragging, pinching, and rotation

In multi-touch environments such as those on mobile devices, users often use their fingers to manipulate objects on the screen. Common gestures such as a one-finger drag and a two-finger pinch can move and scale objects, or even rotate them. These gestures are generally implemented using transform matrices, and this article shows you how to do that.



All the samples shown here use the `Xamarin.Forms` touch-tracking effect presented in the article [Invoking Events from Effects](#).

Dragging and Translation

One of the most important applications of matrix transforms is touch processing. A single `SKMatrix` value can consolidate a series of touch operations.

For single-finger dragging, the `SKMatrix` value performs translation. This is demonstrated in the [Bitmap Dragging](#) page. The XAML file instantiates an `SKCanvasView` in a `Xamarin.Forms Grid`. A `TouchEffect` object has been added to the `Effects` collection of that `Grid`:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    xmlns:tt="clr-namespace:TouchTracking"
    x:Class="SkiaSharpFormsDemos.Transforms.BitmapDraggingPage"
    Title="Bitmap Dragging">

    <Grid BackgroundColor="White">
        <skia:SKCanvasView x:Name="canvasView"
            PaintSurface="OnCanvasViewPaintSurface" />
        <Grid.Effects>
            <tt:TouchEvent Capture="True"
                TouchAction="OnTouchEffectAction" />
        </Grid.Effects>
    </Grid>
</ContentPage>

```

In theory, the `TouchEvent` object could be added directly to the `Effects` collection of the `SKCanvasView`, but that doesn't work on all platforms. Because the `SKCanvasView` is the same size as the `Grid` in this configuration, attaching it to the `Grid` works just as well.

The code-behind file loads in a bitmap resource in its constructor and displays it in the `PaintSurface` handler:

```

public partial class BitmapDraggingPage : ContentPage
{
    // Bitmap and matrix for display
    SKBitmap bitmap;
    SKMatrix matrix = SKMatrix.MakeIdentity();
    ...

    public BitmapDraggingPage()
    {
        InitializeComponent();

        string resourceId = "SkiaSharpFormsDemos.Media.SeatedMonkey.jpg";
        Assembly assembly = GetType().GetTypeInfo().Assembly;

        using (Stream stream = assembly.GetManifestResourceStream(resourceId))
        {
            bitmap = SKBitmap.Decode(stream);
        }
    }
    ...

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Display the bitmap
        canvas.SetMatrix(matrix);
        canvas.DrawBitmap(bitmap, new SKPoint());
    }
}

```

Without any further code, the `SKMatrix` value is always the identify matrix, and it would have no effect on the display of the bitmap. The goal of the `OnTouchEffectAction` handler set in the XAML file is to alter the matrix value to reflect touch manipulations.

The `OnTouchEffectAction` handler begins by converting the Xamarin.Forms `Point` value into a SkiaSharp

`SKPoint` value. This is a simple matter of scaling based on the `Width` and `Height` properties of `SKCanvasView` (which are device-independent units) and the `CanvasSize` property, which is in units of pixels:

```
public partial class BitmapDraggingPage : ContentPage
{
    ...
    // Touch information
    long touchId = -1;
    SKPoint previousPoint;
    ...
    void OnTouchEffectAction(object sender, TouchEventArgs args)
    {
        // Convert Xamarin.Forms point to pixels
        Point pt = args.Location;
        SKPoint point =
            new SKPoint((float)(canvasView.CanvasSize.Width * pt.X / canvasView.Width),
                        (float)(canvasView.CanvasSize.Height * pt.Y / canvasView.Height));

        switch (args.Type)
        {
            case TouchActionType.Pressed:
                // Find transformed bitmap rectangle
                SKRect rect = new SKRect(0, 0, bitmap.Width, bitmap.Height);
                rect = matrix.MapRect(rect);

                // Determine if the touch was within that rectangle
                if (rect.Contains(point))
                {
                    touchId = args.Id;
                    previousPoint = point;
                }
                break;

            case TouchActionType.Moved:
                if (touchId == args.Id)
                {
                    // Adjust the matrix for the new position
                    matrix.TransX += point.X - previousPoint.X;
                    matrix.TransY += point.Y - previousPoint.Y;
                    previousPoint = point;
                    canvasView.InvalidateSurface();
                }
                break;

            case TouchActionType.Released:
            case TouchActionType.Cancelled:
                touchId = -1;
                break;
        }
    }
    ...
}
```

When a finger first touches the screen, an event of type `TouchActionType.Pressed` is fired. The first task is to determine if the finger is touching the bitmap. Such a task is often called *hit-testing*. In this case, hit-testing can be accomplished by creating an `SKRect` value corresponding to the bitmap, applying the matrix transform to it with `MapRect`, and then determining if the touch point is inside the transformed rectangle.

If that is the case, then the `touchId` field is set to the touch ID, and the finger position is saved.

For the `TouchActionType.Moved` event, the translation factors of the `SKMatrix` value are adjusted based on the current position of the finger, and the new position of the finger. That new position is saved for the next time through, and the `SKCanvasView` is invalidated.

As you experiment with this program, take note that you can only drag the bitmap when your finger touches an area where the bitmap is displayed. Although that restriction is not very important for this program, it becomes crucial when manipulating multiple bitmaps.

Pinching and Scaling

What do you want to happen when two fingers touch the bitmap? If the two fingers move in parallel, then you probably want the bitmap to move along with the fingers. If the two fingers perform a pinch or stretch operation, then you might want the bitmap to be rotated (to be discussed in the next section) or scaled. When scaling a bitmap, it makes most sense for the two fingers to remain in the same positions relative to the bitmap, and for the bitmap to be scaled accordingly.

Handling two fingers at once seems complicated, but keep in mind that the `TouchAction` handler only receives information about one finger at a time. If two fingers are manipulating the bitmap, then for each event, one finger has changed position but the other has not changed. In the **Bitmap Scaling** page code below, the finger that has not changed position is called the *pivot* point because the transform is relative to that point.

One difference between this program and the previous program is that multiple touch IDs must be saved. A dictionary is used for this purpose, where the touch ID is the dictionary key and the dictionary value is the current position of that finger:

```
public partial class BitmapScalingPage : ContentPage
{
    ...
    // Touch information
    Dictionary<long, SKPoint> touchDictionary = new Dictionary<long, SKPoint>();
    ...

    void OnTouchEffectAction(object sender, TouchEventArgs args)
    {
        // Convert Xamarin.Forms point to pixels
        Point pt = args.Location;
        SKPoint point =
            new SKPoint((float)(canvasView.CanvasSize.Width * pt.X / canvasView.Width),
                        (float)(canvasView.CanvasSize.Height * pt.Y / canvasView.Height));

        switch (args.Type)
        {
            case TouchActionType.Pressed:
                // Find transformed bitmap rectangle
                SKRect rect = new SKRect(0, 0, bitmap.Width, bitmap.Height);
                rect = matrix.MapRect(rect);

                // Determine if the touch was within that rectangle
                if (rect.Contains(point) && !touchDictionary.ContainsKey(args.Id))
                {
                    touchDictionary.Add(args.Id, point);
                }
                break;

            case TouchActionType.Moved:
                if (touchDictionary.ContainsKey(args.Id))
                {
                    // Single-finger drag
                    if (touchDictionary.Count == 1)
                    {
                        SKPoint prevPoint = touchDictionary[args.Id];

                        // Adjust the matrix for the new position
                        matrix.TransX += point.X - prevPoint.X;
                        matrix.TransY += point.Y - prevPoint.Y;
                        canvasView.InvalidateSurface();
                    }
                    // Double-finger scale and drag
                }
        }
    }
}
```

```

        else if (touchDictionary.Count >= 2)
        {
            // Copy two dictionary keys into array
            long[] keys = new long[touchDictionary.Count];
            touchDictionary.Keys.CopyTo(keys, 0);

            // Find index of non-moving (pivot) finger
            int pivotIndex = (keys[0] == args.Id) ? 1 : 0;

            // Get the three points involved in the transform
            SKPoint pivotPoint = touchDictionary[keys[pivotIndex]];
            SKPoint prevPoint = touchDictionary[args.Id];
            SKPoint newPoint = point;

            // Calculate two vectors
            SKPoint oldVector = prevPoint - pivotPoint;
            SKPoint newVector = newPoint - pivotPoint;

            // Scaling factors are ratios of those
            float scaleX = newVector.X / oldVector.X;
            float scaleY = newVector.Y / oldVector.Y;

            if (!float.IsNaN(scaleX) && !float.IsInfinity(scaleX) &&
                !float.IsNaN(scaleY) && !float.IsInfinity(scaleY))
            {
                // If something bad hasn't happened, calculate a scale and translation matrix
                SKMatrix scaleMatrix =
                    SKMatrix.MakeScale(scaleX, scaleY, pivotPoint.X, pivotPoint.Y);

                SKMatrix.PostConcat(ref matrix, scaleMatrix);
                canvasView.InvalidateSurface();
            }
        }

        // Store the new point in the dictionary
        touchDictionary[args.Id] = point;
    }

    break;

    case TouchActionType.Released:
    case TouchActionType.Cancelled:
        if (touchDictionary.ContainsKey(args.Id))
        {
            touchDictionary.Remove(args.Id);
        }
        break;
    }
}
...
}

```

The handling of the `Pressed` action is almost the same as the previous program except that the ID and touch point are added to the dictionary. The `Released` and `Cancelled` actions remove the dictionary entry.

The handling for the `Moved` action is more complex, however. If there's only one finger involved, then the processing is very much the same as the previous program. For two or more fingers, the program must also obtain information from the dictionary involving the finger that is not moving. It does this by copying the dictionary keys into an array and then comparing the first key with the ID of the finger being moved. That allows the program to obtain the pivot point corresponding to the finger that is not moving.

Next, the program calculates two vectors of the new finger position relative to the pivot point, and the old finger position relative to the pivot point. The ratios of these vectors are scaling factors. Because division by zero is a possibility, these must be checked for infinite values or NaN (not a number) values. If all is well, a scaling transform is concatenated with the `SKMatrix` value saved as a field.

As you experiment with this page, you'll notice that you can drag the bitmap with one or two fingers, or scale it with two fingers. The scaling is *anisotropic*, which means that the scaling can be different in the horizontal and vertical directions. This distorts the aspect ratio, but also allows you to flip the bitmap to make a mirror image. You might also discover that you can shrink the bitmap to a zero dimension, and it disappears. In production code, you'll want to guard against this.

Two-finger rotation

The **Bitmap Rotate** page allows you to use two fingers for either rotation or isotropic scaling. The bitmap always retains its correct aspect ratio. Using two fingers for both rotation and anisotropic scaling does not work very well because the movement of the fingers is very similar for both tasks.

The first big difference in this program is the hit-testing logic. The previous programs used the `Contains` method of `SKRect` to determine if the touch point is within the transformed rectangle that corresponds to the bitmap. But as the user manipulates the bitmap, the bitmap might be rotated, and `SKRect` cannot properly represent a rotated rectangle. You might fear that the hit-testing logic needs to implement rather complex analytic geometry in that case.

However, a shortcut is available: Determining if a point lies within the boundaries of a transformed rectangle is the same as determining if an inverse transformed point lies within the boundaries of the untransformed rectangle. That's a much easier calculation, and the logic can continue to use the convenient `Contains` method:

```
public partial class BitmapRotationPage : ContentPage
{
    ...
    // Touch information
    Dictionary<long, SKPoint> touchDictionary = new Dictionary<long, SKPoint>();
    ...
    void OnTouchEffectAction(object sender, TouchEventArgs args)
    {
        // Convert Xamarin.Forms point to pixels
        Point pt = args.Location;
        SKPoint point =
            new SKPoint((float)(canvasView.CanvasSize.Width * pt.X / canvasView.Width),
                        (float)(canvasView.CanvasSize.Height * pt.Y / canvasView.Height));

        switch (args.Type)
        {
            case TouchActionType.Pressed:
                if (!touchDictionary.ContainsKey(args.Id))
                {
                    // Invert the matrix
                    if (matrix.TryInvert(out SKMatrix inverseMatrix))
                    {
                        // Transform the point using the inverted matrix
                        SKPoint transformedPoint = inverseMatrix.MapPoint(point);

                        // Check if it's in the untransformed bitmap rectangle
                        SKRect rect = new SKRect(0, 0, bitmap.Width, bitmap.Height);

                        if (rect.Contains(transformedPoint))
                        {
                            touchDictionary.Add(args.Id, point);
                        }
                    }
                }
                break;

            case TouchActionType.Moved:
                if (touchDictionary.ContainsKey(args.Id))
                {
                    // Single-finger drag
                    if (touchDictionary.Count == 1)
```

```

        ...
    }

    SKPoint prevPoint = touchDictionary[args.Id];

    // Adjust the matrix for the new position
    matrix.TransX += point.X - prevPoint.X;
    matrix.TransY += point.Y - prevPoint.Y;
    canvasView.InvalidateSurface();
}

// Double-finger rotate, scale, and drag
else if (touchDictionary.Count >= 2)
{
    // Copy two dictionary keys into array
    long[] keys = new long[touchDictionary.Count];
    touchDictionary.Keys.CopyTo(keys, 0);

    // Find index non-moving (pivot) finger
    int pivotIndex = (keys[0] == args.Id) ? 1 : 0;

    // Get the three points in the transform
    SKPoint pivotPoint = touchDictionary[keys[pivotIndex]];
    SKPoint prevPoint = touchDictionary[args.Id];
    SKPoint newPoint = point;

    // Calculate two vectors
    SKPoint oldVector = prevPoint - pivotPoint;
    SKPoint newVector = newPoint - pivotPoint;

    // Find angles from pivot point to touch points
    float oldAngle = (float)Math.Atan2(oldVector.Y, oldVector.X);
    float newAngle = (float)Math.Atan2(newVector.Y, newVector.X);

    // Calculate rotation matrix
    float angle = newAngle - oldAngle;
    SKMatrix touchMatrix = SKMatrix.MakeRotation(angle, pivotPoint.X, pivotPoint.Y);

    // Effectively rotate the old vector
    float magnitudeRatio = Magnitude(oldVector) / Magnitude(newVector);
    oldVector.X = magnitudeRatio * newVector.X;
    oldVector.Y = magnitudeRatio * newVector.Y;

    // Isotropic scaling!
    float scale = Magnitude(newVector) / Magnitude(oldVector);

    if (!float.IsNaN(scale) && !float.IsInfinity(scale))
    {
        SKMatrix.PostConcat(ref touchMatrix,
            SKMatrix.MakeScale(scale, scale, pivotPoint.X, pivotPoint.Y));

        SKMatrix.PostConcat(ref matrix, touchMatrix);
        canvasView.InvalidateSurface();
    }
}

// Store the new point in the dictionary
touchDictionary[args.Id] = point;
}

break;

case TouchActionType.Released:
case TouchActionType.Cancelled:
    if (touchDictionary.ContainsKey(args.Id))
    {
        touchDictionary.Remove(args.Id);
    }
    break;
}
}

```

```

    float Magnitude(SKPoint point)
    {
        return (float)Math.Sqrt(Math.Pow(point.X, 2) + Math.Pow(point.Y, 2));
    }
    ...
}

```

The logic for the `Moved` event starts out like the previous program. Two vectors named `oldVector` and `newVector` are calculated based on the previous and the current point of the moving finger and the pivot point of the unmoving finger. But then angles of these vectors are determined, and the difference is the rotation angle.

Scaling might also be involved, so the old vector is rotated based on the rotation angle. The relative magnitude of the two vectors is now the scaling factor. Notice that the same `scale` value is used for horizontal and vertical scaling so that scaling is isotropic. The `matrix` field is adjusted by both the rotation matrix and a scale matrix.

If your application needs to implement touch processing for a single bitmap (or other object), you can adapt the code from these three samples for your own application. But if you need to implement touch processing for multiple bitmaps, you'll probably want to encapsulate these touch operations in other classes.

Encapsulating the Touch Operations

The [Touch Manipulation](#) page demonstrates the touch manipulation of a single bitmap, but using several other files that encapsulate much of the logic shown above. The first of these files is the `TouchManipulationMode` enumeration, which indicates the different types of touch manipulation implemented by the code you'll be seeing:

```

enum TouchManipulationMode
{
    None,
    PanOnly,
    IsotropicScale,      // includes panning
    AnisotropicScale,    // includes panning
    ScaleRotate,         // implies isotropic scaling
    ScaleDualRotate      // adds one-finger rotation
}

```

`PanOnly` is a one-finger drag that is implemented with translation. All the subsequent options also include panning but involve two fingers: `IsotropicScale` is a pinch operation that results in the object scaling equally in the horizontal and vertical directions. `AnisotropicScale` allows unequal scaling.

The `ScaleRotate` option is for two-finger scaling and rotation. Scaling is isotropic. As mentioned earlier, implementing two-finger rotation with anisotropic scaling is problematic because the finger movements are essentially the same.

The `ScaleDualRotate` option adds one-finger rotation. When a single finger drags the object, the dragged object is first rotated around its center so that the center of the object lines up with the dragging vector.

The `TouchManipulationPage.xaml` file includes a `Picker` with the members of the `TouchManipulationMode` enumeration:

```

<?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:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    xmlns:tt="clr-namespace:TouchTracking"
    xmlns:local="clr-namespace:SkiaSharpFormsDemos.Transforms"
    x:Class="SkiaSharpFormsDemos.Transforms.TouchManipulationPage"
    Title="Touch Manipulation">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Picker Title="Touch Mode"
            Grid.Row="0"
            SelectedIndexChanged="OnTouchModePickerSelectedIndexChanged">
            <Picker.ItemsSource>
                <x:Array Type="{x:Type local:TouchManipulationMode}">
                    <x:Static Member="local:TouchManipulationMode.None" />
                    <x:Static Member="local:TouchManipulationMode.PanOnly" />
                    <x:Static Member="local:TouchManipulationMode.IsotropicScale" />
                    <x:Static Member="local:TouchManipulationMode.AnisotropicScale" />
                    <x:Static Member="local:TouchManipulationMode.ScaleRotate" />
                    <x:Static Member="local:TouchManipulationMode.ScaleDualRotate" />
                </x:Array>
            </Picker.ItemsSource>
            <Picker.SelectedIndex>
                4
            </Picker.SelectedIndex>
        </Picker>

        <Grid BackgroundColor="White"
            Grid.Row="1">

            <skia:SKCanvasView x:Name="canvasView"
                PaintSurface="OnCanvasViewPaintSurface" />
            <Grid.Effects>
                <tt:TouchEffect Capture="True"
                    TouchAction="OnTouchEffectAction" />
            </Grid.Effects>
        </Grid>
    </Grid>
</ContentPage>

```

Towards the bottom is an `SKCanvasView` and a `TouchEffect` attached to the single-cell `Grid` that encloses it.

The `TouchManipulationPage.xaml.cs` code-behind file has a `bitmap` field but it is not of type `SKBitmap`. The type is `TouchManipulationBitmap` (a class you'll see shortly):

```

public partial class TouchManipulationPage : ContentPage
{
    TouchManipulationBitmap bitmap;
    ...

    public TouchManipulationPage()
    {
        InitializeComponent();

        string resourceId = "SkiaSharpFormsDemos.Media.MountainClimbers.jpg";
        Assembly assembly = GetType().GetTypeInfo().Assembly;

        using (Stream stream = assembly.GetManifestResourceStream(resourceId))
        {
            SKBitmap bitmap = SKBitmap.Decode(stream);
            this.bitmap = new TouchManipulationBitmap(bitmap);
            this.bitmap.TouchManager.Mode = TouchManipulationMode.ScaleRotate;
        }
    }
    ...
}

```

The constructor instantiates a `TouchManipulationBitmap` object, passing to the constructor an `SKBitmap` obtained from an embedded resource. The constructor concludes by setting the `Mode` property of the `TouchManager` property of the `TouchManipulationBitmap` object to a member of the `TouchManipulationMode` enumeration.

The `SelectedIndexChanged` handler for the `Picker` also sets this `Mode` property:

```

public partial class TouchManipulationPage : ContentPage
{
    ...
    void OnTouchModePickerSelectedIndexChanged(object sender, EventArgs args)
    {
        if (bitmap != null)
        {
            Picker picker = (Picker)sender;
            bitmap.TouchManager.Mode = (TouchManipulationMode)picker.SelectedItem;
        }
    }
    ...
}

```

The `TouchAction` handler of the `TouchEffect` instantiated in the XAML file calls two methods in `TouchManipulationBitmap` named `HitTest` and `ProcessTouchEvent`:

```

public partial class TouchManipulationPage : ContentPage
{
    ...
    List<long> touchIds = new List<long>();
    ...
    void OnTouchEffectAction(object sender, TouchEventArgs args)
    {
        // Convert Xamarin.Forms point to pixels
        Point pt = args.Location;
        SKPoint point =
            new SKPoint((float)(canvasView.CanvasSize.Width * pt.X / canvasView.Width),
                        (float)(canvasView.CanvasSize.Height * pt.Y / canvasView.Height));

        switch (args.Type)
        {
            case TouchActionType.Pressed:
                if (bitmap.HitTest(point))
                {
                    touchIds.Add(args.Id);
                    bitmap.ProcessTouchEvent(args.Id, args.Type, point);
                    break;
                }
                break;

            case TouchActionType.Moved:
                if (touchIds.Contains(args.Id))
                {
                    bitmap.ProcessTouchEvent(args.Id, args.Type, point);
                    canvasView.InvalidateSurface();
                }
                break;

            case TouchActionType.Released:
            case TouchActionType.Cancelled:
                if (touchIds.Contains(args.Id))
                {
                    bitmap.ProcessTouchEvent(args.Id, args.Type, point);
                    touchIds.Remove(args.Id);
                    canvasView.InvalidateSurface();
                }
                break;
            }
        }
    ...
}

```

If the `HitTest` method returns `true` — meaning that a finger has touched the screen within the area occupied by the bitmap — then the touch ID is added to the `TouchIds` collection. This ID represents the sequence of touch events for that finger until the finger lifts from the screen. If multiple fingers touch the bitmap, then the `touchIds` collection contains a touch ID for each finger.

The `TouchAction` handler also calls the `ProcessTouchEvent` class in `TouchManipulationBitmap`. This is where some (but not all) of the real touch processing occurs.

The `TouchManipulationBitmap` class is a wrapper class for `SKBitmap` that contains code to render the bitmap and process touch events. It works in conjunction with more generalized code in a `TouchManipulationManager` class (which you'll see shortly).

The `TouchManipulationBitmap` constructor saves the `SKBitmap` and instantiates two properties, the `TouchManager` property of type `TouchManipulationManager` and the `Matrix` property of type `SKMatrix`:

```

class TouchManipulationBitmap
{
    SKBitmap bitmap;
    ...

    public TouchManipulationBitmap(SKBitmap bitmap)
    {
        this.bitmap = bitmap;
        Matrix = SKMatrix.MakeIdentity();

        TouchManager = new TouchManipulationManager
        {
            Mode = TouchManipulationMode.ScaleRotate
        };
    }

    public TouchManipulationManager TouchManager { set; get; }

    public SKMatrix Matrix { set; get; }
    ...
}

```

This `Matrix` property is the accumulated transform resulting from all the touch activity. As you'll see, each touch event is resolved into a matrix, which is then concatenated with the `SKMatrix` value stored by the `Matrix` property.

The `TouchManipulationBitmap` object draws itself in its `Paint` method. The argument is an `SKCanvas` object. This `SKCanvas` might already have a transform applied to it, so the `Paint` method concatenates the `Matrix` property associated with the bitmap to the existing transform, and restores the canvas when it has finished:

```

class TouchManipulationBitmap
{
    ...
    public void Paint(SKCanvas canvas)
    {
        canvas.Save();
        SKMatrix matrix = Matrix;
        canvas.Concat(ref matrix);
        canvas.DrawBitmap(bitmap, 0, 0);
        canvas.Restore();
    }
    ...
}

```

The `HitTest` method returns `true` if the user touches the screen at a point within the boundaries of the bitmap. This uses the logic shown earlier in the **Bitmap Rotation** page:

```

class TouchManipulationBitmap
{
    ...
    public bool HitTest(SKPoint location)
    {
        // Invert the matrix
        SKMatrix inverseMatrix;

        if (Matrix.TryInvert(out inverseMatrix))
        {
            // Transform the point using the inverted matrix
            SKPoint transformedPoint = inverseMatrix.MapPoint(location);

            // Check if it's in the untransformed bitmap rectangle
            SKRect rect = new SKRect(0, 0, bitmap.Width, bitmap.Height);
            return rect.Contains(transformedPoint);
        }
        return false;
    }
    ...
}

```

The second public method in `TouchManipulationBitmap` is `ProcessTouchEvent`. When this method is called, it has already been established that the touch event belongs to this particular bitmap. The method maintains a dictionary of `TouchManipulationInfo` objects, which is simply the previous point and the new point of each finger:

```

class TouchManipulationInfo
{
    public SKPoint PreviousPoint { set; get; }

    public SKPoint NewPoint { set; get; }
}

```

Here's the dictionary and the `ProcessTouchEvent` method itself:

```

class TouchManipulationBitmap
{
    ...
    Dictionary<long, TouchManipulationInfo> touchDictionary =
        new Dictionary<long, TouchManipulationInfo>();

    ...
    public void ProcessTouchEvent(long id, TouchActionType type, SKPoint location)
    {
        switch (type)
        {
            case TouchActionType.Pressed:
                touchDictionary.Add(id, new TouchManipulationInfo
                {
                    PreviousPoint = location,
                    NewPoint = location
                });
                break;

            case TouchActionType.Moved:
                TouchManipulationInfo info = touchDictionary[id];
                info.NewPoint = location;
                Manipulate();
                info.PreviousPoint = info.NewPoint;
                break;

            case TouchActionType.Released:
                touchDictionary[id].NewPoint = location;
                Manipulate();
                touchDictionary.Remove(id);
                break;

            case TouchActionType.Cancelled:
                touchDictionary.Remove(id);
                break;
        }
    }
    ...
}

```

In the `Moved` and `Released` events, the method calls `Manipulate`. At these times, the `touchDictionary` contains one or more `TouchManipulationInfo` objects. If the `touchDictionary` contains one item, it is likely that the `PreviousPoint` and `NewPoint` values are unequal and represent the movement of a finger. If multiple fingers are touching the bitmap, the dictionary contains more than one item, but only one of these items has different `PreviousPoint` and `NewPoint` values. All the rest have equal `PreviousPoint` and `NewPoint` values.

This is important: The `Manipulate` method can assume that it's processing the movement of only one finger. At the time of this call none of the other fingers are moving, and if they really are moving (as is likely), those movements will be processed in future calls to `Manipulate`.

The `Manipulate` method first copies the dictionary to an array for convenience. It ignores anything other than the first two entries. If more than two fingers are attempting to manipulate the bitmap, the others are ignored. `Manipulate` is the final member of `TouchManipulationBitmap`:

```

class TouchManipulationBitmap
{
    ...
    void Manipulate()
    {
        TouchManipulationInfo[] infos = new TouchManipulationInfo[touchDictionary.Count];
        touchDictionary.Values.CopyTo(infos, 0);
        SKMatrix touchMatrix = SKMatrix.MakeIdentity();

        if (infos.Length == 1)
        {
            SKPoint prevPoint = infos[0].PreviousPoint;
            SKPoint newPoint = infos[0].NewPoint;
            SKPoint pivotPoint = Matrix.MapPoint(bitmap.Width / 2, bitmap.Height / 2);

            touchMatrix = TouchManager.OneFingerManipulate(prevPoint, newPoint, pivotPoint);
        }
        else if (infos.Length >= 2)
        {
            int pivotIndex = infos[0].NewPoint == infos[0].PreviousPoint ? 0 : 1;
            SKPoint pivotPoint = infos[pivotIndex].NewPoint;
            SKPoint newPoint = infos[1 - pivotIndex].NewPoint;
            SKPoint prevPoint = infos[1 - pivotIndex].PreviousPoint;

            touchMatrix = TouchManager.TwoFingerManipulate(prevPoint, newPoint, pivotPoint);
        }

        SKMatrix matrix = Matrix;
        SKMatrix.PostConcat(ref matrix, touchMatrix);
        Matrix = matrix;
    }
}

```

If one finger is manipulating the bitmap, `Manipulate` calls the `OneFingerManipulate` method of the `TouchManipulationManager` object. For two fingers, it calls `TwoFingerManipulate`. The arguments to these methods are the same: the `prevPoint` and `newPoint` arguments represent the finger that is moving. But the `pivotPoint` argument is different for the two calls:

For one-finger manipulation, the `pivotPoint` is the center of the bitmap. This is to allow for one-finger rotation. For two-finger manipulation, the event indicates the movement of only one finger, so that the `pivotPoint` is the finger that is not moving.

In both cases, `TouchManipulationManager` returns an `SKMatrix` value, which the method concatenates with the current `Matrix` property that `TouchManipulationPage` uses to render the bitmap.

`TouchManipulationManager` is generalized and uses no other files except `TouchManipulationMode`. You might be able to use this class without change in your own applications. It defines a single property of type `TouchManipulationMode`:

```

class TouchManipulationManager
{
    public TouchManipulationMode Mode { set; get; }
    ...
}

```

However, you'll probably want to avoid the `AnisotropicScale` option. It's very easy with this option to manipulate the bitmap so that one of the scaling factors becomes zero. That makes the bitmap disappear from sight, never to return. If you truly do need anisotropic scaling, you'll want to enhance the logic to avoid undesirable outcomes.

`TouchManipulationManager` makes use of vectors, but since there is no `SKVector` structure in SkiaSharp, `SKPoint` is used instead. `SKPoint` supports the subtraction operator, and the result can be treated as a vector. The only vector-specific logic that needed to be added is a `Magnitude` calculation:

```
class TouchManipulationManager
{
    ...
    float Magnitude(SKPoint point)
    {
        return (float)Math.Sqrt(Math.Pow(point.X, 2) + Math.Pow(point.Y, 2));
    }
}
```

Whenever rotation has been selected, both the one-finger and two-finger manipulation methods handle the rotation first. If any rotation is detected, then the rotation component is effectively removed. What remains is interpreted as panning and scaling.

Here's the `OneFingerManipulate` method. If one-finger rotation has not been enabled, then the logic is simple — it simply uses the previous point and new point to construct a vector named `delta` that corresponds precisely to translation. With one-finger rotation enabled, the method uses angles from the pivot point (the center of the bitmap) to the previous point and new point to construct a rotation matrix:

```

class TouchManipulationManager
{
    public TouchManipulationMode Mode { set; get; }

    public SKMatrix OneFingerManipulate(SKPoint prevPoint, SKPoint newPoint, SKPoint pivotPoint)
    {
        if (Mode == TouchManipulationMode.None)
        {
            return SKMatrix.MakeIdentity();
        }

        SKMatrix touchMatrix = SKMatrix.MakeIdentity();
        SKPoint delta = newPoint - prevPoint;

        if (Mode == TouchManipulationMode.ScaleDualRotate) // One-finger rotation
        {
            SKPoint oldVector = prevPoint - pivotPoint;
            SKPoint newVector = newPoint - pivotPoint;

            // Avoid rotation if fingers are too close to center
            if (Magnitude(newVector) > 25 && Magnitude(oldVector) > 25)
            {
                float prevAngle = (float)Math.Atan2(oldVector.Y, oldVector.X);
                float newAngle = (float)Math.Atan2(newVector.Y, newVector.X);

                // Calculate rotation matrix
                float angle = newAngle - prevAngle;
                touchMatrix = SKMatrix.MakeRotation(angle, pivotPoint.X, pivotPoint.Y);

                // Effectively rotate the old vector
                float magnitudeRatio = Magnitude(oldVector) / Magnitude(newVector);
                oldVector.X = magnitudeRatio * newVector.X;
                oldVector.Y = magnitudeRatio * newVector.Y;

                // Recalculate delta
                delta = newVector - oldVector;
            }
        }

        // Multiply the rotation matrix by a translation matrix
        SKMatrix.PostConcat(ref touchMatrix, SKMatrix.MakeTranslation(delta.X, delta.Y));

        return touchMatrix;
    }
    ...
}

```

In the `TwoFingerManipulate` method, the pivot point is the position of the finger that's not moving in this particular touch event. The rotation is very similar to the one-finger rotation, and then the vector named `oldVector` (based on the previous point) is adjusted for the rotation. The remaining movement is interpreted as scaling:

```

class TouchManipulationManager
{
    ...
    public SKMatrix TwoFingerManipulate(SKPoint prevPoint, SKPoint newPoint, SKPoint pivotPoint)
    {
        SKMatrix touchMatrix = SKMatrix.MakeIdentity();
        SKPoint oldVector = prevPoint - pivotPoint;
        SKPoint newVector = newPoint - pivotPoint;

        if (Mode == TouchManipulationMode.ScaleRotate ||
            Mode == TouchManipulationMode.ScaleDualRotate)
        {
            // Find angles from pivot point to touch points
            float oldAngle = (float)Math.Atan2(oldVector.Y, oldVector.X);
            float newAngle = (float)Math.Atan2(newVector.Y, newVector.X);

            // Calculate rotation matrix
            float angle = newAngle - oldAngle;
            touchMatrix = SKMatrix.MakeRotation(angle, pivotPoint.X, pivotPoint.Y);

            // Effectively rotate the old vector
            float magnitudeRatio = Magnitude(oldVector) / Magnitude(newVector);
            oldVector.X = magnitudeRatio * newVector.X;
            oldVector.Y = magnitudeRatio * newVector.Y;
        }

        float scaleX = 1;
        float scaleY = 1;

        if (Mode == TouchManipulationMode.AnisotropicScale)
        {
            scaleX = newVector.X / oldVector.X;
            scaleY = newVector.Y / oldVector.Y;

        }
        else if (Mode == TouchManipulationMode.IsotropicScale ||
                  Mode == TouchManipulationMode.ScaleRotate ||
                  Mode == TouchManipulationMode.ScaleDualRotate)
        {
            scaleX = scaleY = Magnitude(newVector) / Magnitude(oldVector);
        }

        if (!float.IsNaN(scaleX) && !float.IsInfinity(scaleX) &&
            !float.IsNaN(scaleY) && !float.IsInfinity(scaleY))
        {
            SKMatrix.PostConcat(ref touchMatrix,
                SKMatrix.MakeScale(scaleX, scaleY, pivotPoint.X, pivotPoint.Y));
        }

        return touchMatrix;
    }
    ...
}

```

You'll notice there is no explicit translation in this method. However, both the `MakeRotation` and `MakeScale` methods are based on the pivot point, and that includes implicit translation. If you're using two fingers on the bitmap and dragging them in the same direction, `TouchManipulation` will get a series of touch events alternating between the two fingers. As each finger moves relative to the other, scaling or rotation results, but it's negated by the other finger's movement, and the result is translation.

The only remaining part of the **Touch Manipulation** page is the `PaintSurface` handler in the `TouchManipulationPage` code-behind file. This calls the `Paint` method of the `TouchManipulationBitmap`, which applies the matrix representing the accumulated touch activity:

```

public partial class TouchManipulationPage : ContentPage
{
    ...
    MatrixDisplay matrixDisplay = new MatrixDisplay();
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

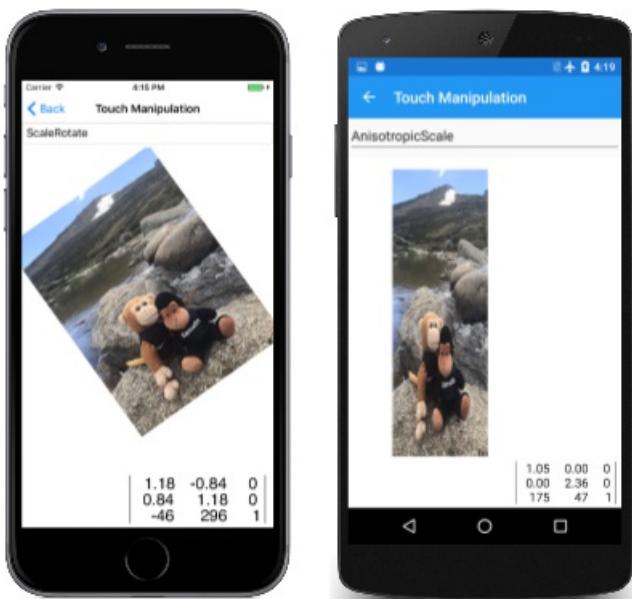
        // Display the bitmap
        bitmap.Paint(canvas);

        // Display the matrix in the lower-right corner
        SKSize matrixSize = matrixDisplay.Measure(bitmap.Matrix);

        matrixDisplay.Paint(canvas, bitmap.Matrix,
            new SKPoint(info.Width - matrixSize.Width,
                info.Height - matrixSize.Height));
    }
}

```

The `PaintSurface` handler concludes by displaying a `MatrixDisplay` object showing the accumulated touch matrix:



Manipulating Multiple Bitmaps

One of the advantages of isolating touch-processing code in classes such as `TouchManipulationBitmap` and `TouchManipulationManager` is the ability to reuse these classes in a program that allows the user to manipulate multiple bitmaps.

The `Bitmap Scatter View` page demonstrates how this is done. Rather than defining a field of type `TouchManipulationBitmap`, the `BitmapScatterPage` class defines a `List` of bitmap objects:

```

public partial class BitmapScatterViewPage : ContentPage
{
    List<TouchManipulationBitmap> bitmapCollection =
        new List<TouchManipulationBitmap>();
    ...
    public BitmapScatterViewPage()
    {
        InitializeComponent();

        // Load in all the available bitmaps
        Assembly assembly = GetType().GetTypeInfo().Assembly;
        string[] resourceIDs = assembly.GetManifestResourceNames();
        SKPoint position = new SKPoint();

        foreach (string resourceId in resourceIDs)
        {
            if (resourceId.EndsWith(".png") ||
                resourceId.EndsWith(".jpg"))
            {
                using (Stream stream = assembly.GetManifestResourceStream(resourceId))
                {
                    SKBitmap bitmap = SKBitmap.Decode(stream);
                    bitmapCollection.Add(new TouchManipulationBitmap(bitmap))
                    {
                        Matrix = SKMatrix.MakeTranslation(position.X, position.Y),
                    });
                    position.X += 100;
                    position.Y += 100;
                }
            }
        }
    }
    ...
}

```

The constructor loads in all of the bitmaps available as embedded resources, and adds them to the `bitmapCollection`. Notice that the `Matrix` property is initialized on each `TouchManipulationBitmap` object, so the upper-left corners of each bitmap are offset by 100 pixels.

The `BitmapScatterView` page also needs to handle touch events for multiple bitmaps. Rather than defining a `List` of touch IDs of currently manipulated `TouchManipulationBitmap` objects, this program requires a dictionary:

```

public partial class BitmapScatterViewPage : ContentPage
{
    ...
    Dictionary<long, TouchManipulationBitmap> bitmapDictionary =
        new Dictionary<long, TouchManipulationBitmap>();

    ...
    void OnTouchEffectAction(object sender, TouchEventArgs args)
    {
        // Convert Xamarin.Forms point to pixels
        Point pt = args.Location;
        SKPoint point =
            new SKPoint((float)(canvasView.CanvasSize.Width * pt.X / canvasView.Width),
                        (float)(canvasView.CanvasSize.Height * pt.Y / canvasView.Height));

        switch (args.Type)
        {
            case TouchActionType.Pressed:
                for (int i = bitmapCollection.Count - 1; i >= 0; i--)
                {
                    TouchManipulationBitmap bitmap = bitmapCollection[i];

                    if (bitmap.HitTest(point))
                    {
                        // Move bitmap to end of collection
                        bitmapCollection.Remove(bitmap);
                        bitmapCollection.Add(bitmap);

                        // Do the touch processing
                        bitmapDictionary.Add(args.Id, bitmap);
                        bitmap.ProcessTouchEvent(args.Id, args.Type, point);
                        canvasView.InvalidateSurface();
                        break;
                    }
                }
                break;

            case TouchActionType.Moved:
                if (bitmapDictionary.ContainsKey(args.Id))
                {
                    TouchManipulationBitmap bitmap = bitmapDictionary[args.Id];
                    bitmap.ProcessTouchEvent(args.Id, args.Type, point);
                    canvasView.InvalidateSurface();
                }
                break;

            case TouchActionType.Released:
            case TouchActionType.Cancelled:
                if (bitmapDictionary.ContainsKey(args.Id))
                {
                    TouchManipulationBitmap bitmap = bitmapDictionary[args.Id];
                    bitmap.ProcessTouchEvent(args.Id, args.Type, point);
                    bitmapDictionary.Remove(args.Id);
                    canvasView.InvalidateSurface();
                }
                break;
            }
        }
    ...
}

```

Notice how the `Pressed` logic loops through the `bitmapCollection` in reverse. The bitmaps often overlap each other. The bitmaps later in the collection visually lie on top of the bitmaps earlier in the collection. If there are multiple bitmaps under the finger that presses on the screen, the topmost one must be the one that is manipulated by that finger.

Also notice that the `Pressed` logic moves that bitmap to the end of the collection so that it visually moves to the top of the pile of other bitmaps.

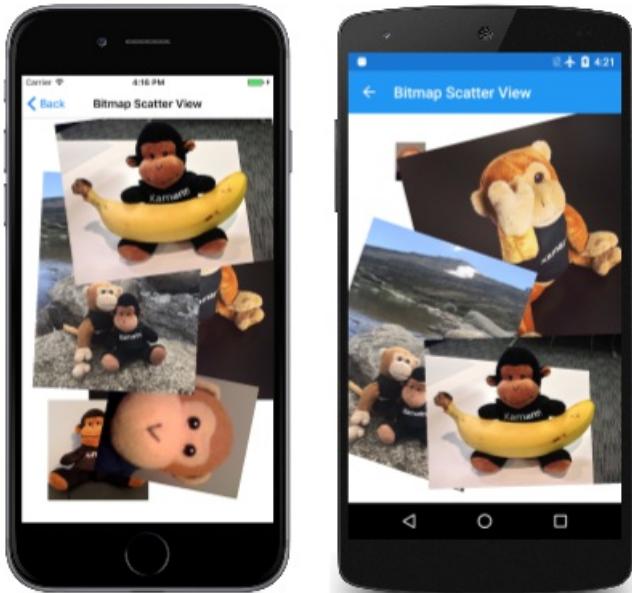
In the `Moved` and `Released` events, the `TouchAction` handler calls the `ProcessingTouchEvent` method in `TouchManipulationBitmap` just like the earlier program.

Finally, the `PaintSurface` handler calls the `Paint` method of each `TouchManipulationBitmap` object:

```
public partial class BitmapScatterViewPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKCanvas canvas = args.Surface.Canvas;
        canvas.Clear();

        foreach (TouchManipulationBitmap bitmap in bitmapCollection)
        {
            bitmap.Paint(canvas);
        }
    }
}
```

The code loops through the collection and displays the pile of bitmaps from the beginning of the collection to the end:



Single-Finger Scaling

A scaling operation generally requires a pinch gesture using two fingers. However, it's possible to implement scaling with a single finger by having the finger move the corners of a bitmap.

This is demonstrated in the [Single Finger Corner Scale](#) page. Because this sample uses a somewhat different type of scaling than that implemented in the `TouchManipulationManager` class, it does not use that class or the `TouchManipulationBitmap` class. Instead, all the touch logic is in the code-behind file. This is somewhat simpler logic than usual because it tracks only one finger at a time, and simply ignores any secondary fingers that might be touching the screen.

The [SingleFingerCornerScale.xaml](#) page instantiates the `SKCanvasView` class and creates a `TouchEffect` object for tracking touch events:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    xmlns:tt="clr-namespace:TouchTracking"
    x:Class="SkiaSharpFormsDemos.Transforms.SingleFingerCornerScalePage"
    Title="Single Finger Corner Scale">

    <Grid BackgroundColor="White"
        Grid.Row="1">

        <skia:SKCanvasView x:Name="canvasView"
            PaintSurface="OnCanvasViewPaintSurface" />
        <Grid.Effects>
            <tt:TouchEffect Capture="True"
                TouchAction="OnTouchEffectAction" />
        </Grid.Effects>
    </Grid>
</ContentPage>

```

The [SingleFingerCornerScalePage.xaml.cs](#) file loads a bitmap resource from the **Media** directory and displays it using an `SKMatrix` object defined as a field:

```

public partial class SingleFingerCornerScalePage : ContentPage
{
    SKBitmap bitmap;
    SKMatrix currentMatrix = SKMatrix.MakeIdentity();
    ...

    public SingleFingerCornerScalePage()
    {
        InitializeComponent();

        string resourceId = "SkiaSharpFormsDemos.Media.SeatedMonkey.jpg";
        Assembly assembly = GetType().GetTypeInfo().Assembly;

        using (Stream stream = assembly.GetManifestResourceStream(resourceId))
        {
            bitmap = SKBitmap.Decode(stream);
        }
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        canvas.SetMatrix(currentMatrix);
        canvas.DrawBitmap(bitmap, 0, 0);
    }
    ...
}

```

This `SKMatrix` object is modified by the touch logic shown below.

The remainder of the code-behind file is the `TouchEffect` event handler. It begins by converting the current location of the finger to an `SKPoint` value. For the `Pressed` action type, the handler checks that no other finger is touching the screen, and that the finger is within the bounds of the bitmap.

The crucial part of the code is an `if` statement involving two calls to the `Math.Pow` method. This math checks if the finger location is outside of an ellipse that fills the bitmap. If so, then that's a scaling operation. The finger is

near one of the corners of the bitmap, and a pivot point is determined that is the opposite corner. If the finger is within this ellipse, it's a regular panning operation:

```
public partial class SingleFingerCornerScalePage : ContentPage
{
    SKBitmap bitmap;
    SKMatrix currentMatrix = SKMatrix.MakeIdentity();

    // Information for translating and scaling
    long? touchId = null;
    SKPoint pressedLocation;
    SKMatrix pressedMatrix;

    // Information for scaling
    bool isScaling;
    SKPoint pivotPoint;
    ...

    void OnTouchEffectAction(object sender, TouchEventArgs args)
    {
        // Convert Xamarin.Forms point to pixels
        Point pt = args.Location;
        SKPoint point =
            new SKPoint((float)(canvasView.CanvasSize.Width * pt.X / canvasView.Width),
                        (float)(canvasView.CanvasSize.Height * pt.Y / canvasView.Height));

        switch (args.Type)
        {
            case TouchActionType.Pressed:
                // Track only one finger
                if (touchId.HasValue)
                    return;

                // Check if the finger is within the boundaries of the bitmap
                SKRect rect = new SKRect(0, 0, bitmap.Width, bitmap.Height);
                rect = currentMatrix.MapRect(rect);
                if (!rect.Contains(point))
                    return;

                // First assume there will be no scaling
                isScaling = false;

                // If touch is outside interior ellipse, make this a scaling operation
                if (Math.Pow((point.X - rect.MidX) / (rect.Width / 2), 2) +
                    Math.Pow((point.Y - rect.MidY) / (rect.Height / 2), 2) > 1)
                {
                    isScaling = true;
                    float xPivot = point.X < rect.MidX ? rect.Right : rect.Left;
                    float yPivot = point.Y < rect.MidY ? rect.Bottom : rect.Top;
                    pivotPoint = new SKPoint(xPivot, yPivot);
                }

                // Common for either pan or scale
                touchId = args.Id;
                pressedLocation = point;
                pressedMatrix = currentMatrix;
                break;
            case TouchActionType.Moved:
                if (!touchId.HasValue || args.Id != touchId.Value)
                    return;

                SKMatrix matrix = SKMatrix.MakeIdentity();

                // Translating
                if (!isScaling)
                {
                    SKPoint delta = point - pressedLocation;
                    matrix.Translate(delta.X, delta.Y);
                }
                else
                {
                    // Scaling
                    float scale = Math.Sqrt(Math.Pow((point.X - rect.MidX) / (rect.Width / 2), 2) +
                        Math.Pow((point.Y - rect.MidY) / (rect.Height / 2), 2));
                    matrix.Scale(scale, scale);
                    matrix.Translate(rect.MidX, rect.MidY);
                }
                currentMatrix = matrix;
                break;
        }
    }
}
```

```

        delta = point - pressedLocation,
        matrix = SKMatrix.MakeTranslation(delta.X, delta.Y);
    }
    // Scaling
    else
    {
        float scaleX = (point.X - pivotPoint.X) / (pressedLocation.X - pivotPoint.X);
        float scaleY = (point.Y - pivotPoint.Y) / (pressedLocation.Y - pivotPoint.Y);
        matrix = SKMatrix.MakeScale(scaleX, scaleY, pivotPoint.X, pivotPoint.Y);
    }

    // Concatenate the matrices
    SKMatrix.PreConcat(ref matrix, pressedMatrix);
    currentMatrix = matrix;
    canvasView.InvalidateSurface();
    break;

    case TouchActionType.Released:
    case TouchActionType.Cancelled:
        touchId = null;
        break;
    }
}
}

```

The `Moved` action type calculates a matrix corresponding to the touch activity from the time the finger pressed the screen up to this time. It concatenates that matrix with the matrix in effect at the time the finger first pressed the bitmap. The scaling operation is always relative to the corner opposite to the one that the finger touched.

For small or oblong bitmaps, an interior ellipse might occupy most of the bitmap and leave tiny areas at the corners to scale the bitmap. You might prefer a somewhat different approach, in which case you can replace that entire `if` block that sets `isScaling` to `true` with this code:

```

float halfHeight = rect.Height / 2;
float halfWidth = rect.Width / 2;

// Top half of bitmap
if (point.Y < rect.MidY)
{
    float yRelative = (point.Y - rect.Top) / halfHeight;

    // Upper-left corner
    if (point.X < rect.MidX - yRelative * halfWidth)
    {
        isScaling = true;
        pivotPoint = new SKPoint(rect.Right, rect.Bottom);
    }
    // Upper-right corner
    else if (point.X > rect.MidX + yRelative * halfWidth)
    {
        isScaling = true;
        pivotPoint = new SKPoint(rect.Left, rect.Bottom);
    }
}
// Bottom half of bitmap
else
{
    float yRelative = (point.Y - rect.MidY) / halfHeight;

    // Lower-left corner
    if (point.X < rect.Left + yRelative * halfWidth)
    {
        isScaling = true;
        pivotPoint = new SKPoint(rect.Right, rect.Top);
    }
    // Lower-right corner
    else if (point.X > rect.Right - yRelative * halfWidth)
    {
        isScaling = true;
        pivotPoint = new SKPoint(rect.Left, rect.Top);
    }
}

```

This code effectively divides the area of the bitmap into an interior diamond shape and four triangles at the corners. This allows much larger areas at the corners to grab and scale the bitmap.

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)
- [Invoking Events from Effects](#)

Non-Affine Transforms

3/5/2021 • 10 minutes to read • [Edit Online](#)



[Download the sample](#)

Create perspective and taper effects with the third column of the transform matrix

Translation, scaling, rotation, and skewing are all classified as *affine* transforms. Affine transforms preserve parallel lines. If two lines are parallel prior to the transform, they remain parallel after the transform. Rectangles are always transformed to parallelograms.

However, SkiaSharp is also capable of non-affine transforms, which have the capability to transform a rectangle into any convex quadrilateral:



A convex quadrilateral is a four-sided figure with interior angles always less than 180 degrees and sides that don't cross each other.

Non-affine transforms result when the third row of the transform matrix is set to values other than 0, 0, and 1. The full `SKMatrix` multiplication is:

$$\begin{vmatrix} x & y & 1 \end{vmatrix} \times \begin{vmatrix} \text{ScaleX} & \text{SkewY} & \text{Persp0} \\ \text{SkewX} & \text{ScaleY} & \text{Persp1} \\ \text{TransX} & \text{TransY} & \text{Persp2} \end{vmatrix} = \begin{vmatrix} x' & y' & z' \end{vmatrix}$$

The resultant transform formulas are:

$$x' = \text{ScaleX}\cdot x + \text{SkewX}\cdot y + \text{TransX}$$

$$y' = \text{SkewY}\cdot x + \text{ScaleY}\cdot y + \text{TransY}$$

$$z' = \text{Persp0}\cdot x + \text{Persp1}\cdot y + \text{Persp2}$$

The fundamental rule of using a 3-by-3 matrix for two-dimensional transforms is that everything remains on the plane where Z equals 1. Unless `Persp0` and `Persp1` are 0, and `Persp2` equals 1, the transform has moved the Z coordinates off that plane.

To restore this to a two-dimensional transform, the coordinates must be moved back to that plane. Another step is required. The x', y', and z' values must be divided by z':

$$x'' = x' / z'$$

$$y'' = y' / z'$$

$$z'' = z' / z' = 1$$

These are known as *homogeneous coordinates* and they were developed by mathematician August Ferdinand Möbius, much better known for his topological oddity, the Möbius Strip.

If z' is 0, the division results in infinite coordinates. In fact, one of Möbius's motivations for developing homogeneous coordinates was the ability to represent infinite values with finite numbers.

When displaying graphics, however, you want to avoid rendering something with coordinates that transform to infinite values. Those coordinates won't be rendered. Everything in the vicinity of those coordinates will be very large and probably not visually coherent.

In this equation, you do not want the value of z' becoming zero:

$$z' = \text{Persp0}\cdot x + \text{Persp1}\cdot y + \text{Persp2}$$

Consequently, these values have some practical restrictions:

The `Persp2` cell can either be zero or not zero. If `Persp2` is zero, then z' is zero for the point $(0, 0)$, and that's usually not desirable because that point is very common in two-dimensional graphics. If `Persp2` is not equal to zero, then there is no loss of generality if `Persp2` is fixed at 1. For example, if you determine that `Persp2` should be 5, then you can simply divide all the cells in the matrix by 5, which makes `Persp2` equal to 1, and the result will be the same.

For these reasons, `Persp2` is often fixed at 1, which is the same value in the identity matrix.

Generally, `Persp0` and `Persp1` are small numbers. For example, suppose you begin with an identity matrix but set `Persp0` to 0.01:

$$\begin{vmatrix} 1 & 0 & 0.01 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

The transform formulas are:

$$x' = x / (0.01\cdot x + 1)$$

$$y' = y / (0.01\cdot x + 1)$$

Now use this transform to render a 100-pixel square box positioned at the origin. Here's how the four corners are transformed:

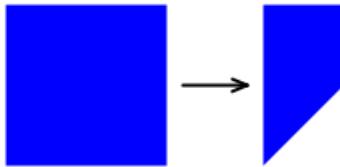
$$(0, 0) \rightarrow (0, 0)$$

$$(0, 100) \rightarrow (0, 100)$$

$$(100, 0) \rightarrow (50, 0)$$

$$(100, 100) \rightarrow (50, 50)$$

When x is 100, then the z' denominator is 2, so the x and y coordinates are effectively halved. The right side of the box becomes shorter than the left side:



The `Persp` part of these cell names refers to "perspective" because the foreshortening suggests that the box is now tilted with the right side further from the viewer.

The **Test Perspective** page allows you to experiment with values of `Persp0` and `Persp1` to get a feel for how they work. Reasonable values of these matrix cells are so small that the `Slider` in the Universal Windows Platform can't properly handle them. To accommodate the UWP problem, the two `Slider` elements in the [TestPerspective.xaml](#) need to be initialized to range from -1 to 1:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Transforms.TestPerspectivePage"
    Title="Test Perpseptive">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Grid.Resources>
            <ResourceDictionary>
                <Style TargetType="Label">
                    <Setter Property="HorizontalContentAlignment" Value="Center" />
                </Style>

                <Style TargetType="Slider">
                    <Setter Property="Minimum" Value="-1" />
                    <Setter Property="Maximum" Value="1" />
                    <Setter Property="Margin" Value="20, 0" />
                </Style>
            </ResourceDictionary>
        </Grid.Resources>

        <Slider x:Name="persp0Slider"
            Grid.Row="0"
            ValueChanged="OnPersp0SliderValueChanged" />

        <Label x:Name="persp0Label"
            Text="Persp0 = 0.0000"
            Grid.Row="1" />

        <Slider x:Name="persp1Slider"
            Grid.Row="2"
            ValueChanged="OnPersp1SliderValueChanged" />

        <Label x:Name="persp1Label"
            Text="Persp1 = 0.0000"
            Grid.Row="3" />

        <skia:SKCanvasView x:Name="canvasView"
            Grid.Row="4"
            PaintSurface="OnCanvasViewPaintSurface" />
    </Grid>
</ContentPage>
```

The event handlers for the sliders in the `TestPerspectivePage` code-behind file divide the values by 100 so that they range between -0.01 and 0.01. In addition, the constructor loads in a bitmap:

```
public partial class TestPerspectivePage : ContentPage
{
    SKBitmap bitmap;

    public TestPerspectivePage()
    {
        InitializeComponent();

        string resourceId = "SkiaSharpFormsDemos.Media.SeatedMonkey.jpg";
        Assembly assembly = GetType().GetTypeInfo().Assembly;

        using (Stream stream = assembly.GetManifestResourceStream(resourceId))
        {
            bitmap = SKBitmap.Decode(stream);
        }
    }

    void OnPersp0SliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        Slider slider = (Slider)sender;
        persp0Label.Text = String.Format("Persp0 = {0:F4}", slider.Value / 100);
        canvasView.InvalidateSurface();
    }

    void OnPersp1SliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        Slider slider = (Slider)sender;
        persp1Label.Text = String.Format("Persp1 = {0:F4}", slider.Value / 100);
        canvasView.InvalidateSurface();
    }

    ...
}
```

The `PaintSurface` handler calculates an `SKMatrix` value named `perspectiveMatrix` based on the values of these two sliders divided by 100. This is combined with two translate transforms that put the center of this transform in the center of the bitmap:

```

public partial class TestPerspectivePage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Calculate perspective matrix
        SKMatrix perspectiveMatrix = SKMatrix.MakeIdentity();
        perspectiveMatrix.Persp0 = (float)persp0Slider.Value / 100;
        perspectiveMatrix.Persp1 = (float)persp1Slider.Value / 100;

        // Center of screen
        float xCenter = info.Width / 2;
        float yCenter = info.Height / 2;

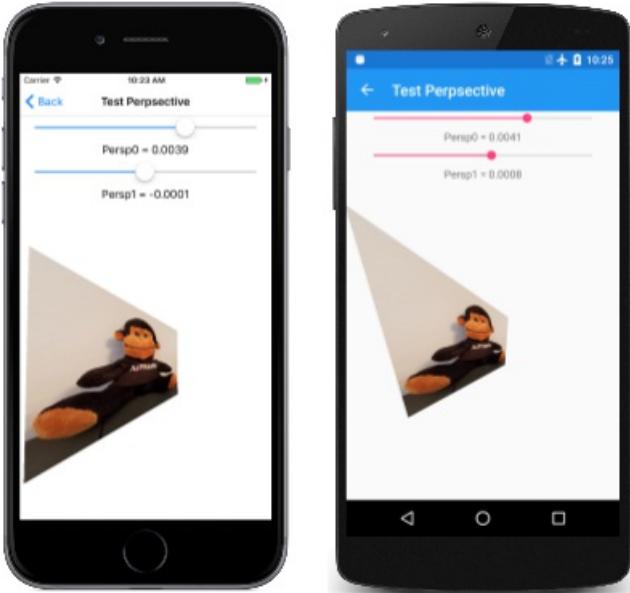
        SKMatrix matrix = SKMatrix.MakeTranslation(-xCenter, -yCenter);
        SKMatrix.PostConcat(ref matrix, perspectiveMatrix);
        SKMatrix.PostConcat(ref matrix, SKMatrix.MakeTranslation(xCenter, yCenter));

        // Coordinates to center bitmap on canvas
        float x = xCenter - bitmap.Width / 2;
        float y = yCenter - bitmap.Height / 2;

        canvas.SetMatrix(matrix);
        canvas.DrawBitmap(bitmap, x, y);
    }
}

```

Here are some sample images:



As you experiment with the sliders, you'll find that values beyond 0.0066 or below -0.0066 cause the image to suddenly become fractured and incoherent. The bitmap being transformed is 300-pixels square. It is transformed relative to its center, so the coordinates of the bitmap range from -150 to 150. Recall that the value of z' is:

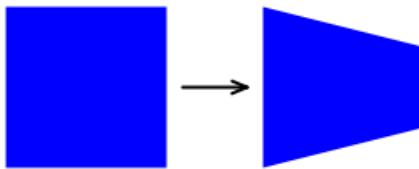
$$z' = \text{Persp0}\cdot x + \text{Persp1}\cdot y + 1$$

If `Persp0` or `Persp1` is greater than 0.0066 or below -0.0066, then there is always some coordinate of the bitmap that results in a z' value of zero. That causes division by zero, and the rendering becomes a mess. When

using non-affine transforms, you want to avoid rendering anything with coordinates that cause division by zero.

Generally, you won't be setting `Persp0` and `Persp1` in isolation. It's also often necessary to set other cells in the matrix to achieve certain types of non-affine transforms.

One such non-affine transform is a *taper transform*. This type of non-affine transform retains the overall dimensions of a rectangle but tapers one side:



The `TaperTransform` class performs a generalized calculation of a non-affine transform based on these parameters:

- the rectangular size of the image being transformed,
- an enumeration that indicates the side of the rectangle that tapers,
- another enumeration that indicates how it tapers, and
- the extent of the tapering.

Here's the code:

```
enum TaperSide { Left, Top, Right, Bottom }

enum TaperCorner { LeftOrTop, RightOrBottom, Both }

static class TaperTransform
{
    public static SKMatrix Make(SKSize size, TaperSide taperSide, TaperCorner taperCorner, float
taperFraction)
    {
        SKMatrix matrix = SKMatrix.MakeIdentity();

        switch (taperSide)
        {
            case TaperSide.Left:
                matrix.ScaleX = taperFraction;
                matrix.ScaleY = taperFraction;
                matrix.Persp0 = (taperFraction - 1) / size.Width;

                switch (taperCorner)
                {
                    case TaperCorner.RightOrBottom:
                        break;

                    case TaperCorner.LeftOrTop:
                        matrix.SkewY = size.Height * matrix.Persp0;
                        matrix.TransY = size.Height * (1 - taperFraction);
                        break;

                    case TaperCorner.Both:
                        matrix.SkewY = (size.Height / 2) * matrix.Persp0;
                        matrix.TransY = size.Height * (1 - taperFraction) / 2;
                        break;
                }
                break;

            case TaperSide.Top:
                matrix.ScaleX = taperFraction;
                matrix.ScaleY = taperFraction;
                matrix.Persp1 = (taperFraction - 1) / size.Height;
        }
    }
}
```

```

        switch (taperCorner)
        {
            case TaperCorner.RightOrBottom:
                break;

            case TaperCorner.LeftOrTop:
                matrix.SkewX = size.Width * matrix.Persp1;
                matrix.TransX = size.Width * (1 - taperFraction);
                break;

            case TaperCorner.Both:
                matrix.SkewX = (size.Width / 2) * matrix.Persp1;
                matrix.TransX = size.Width * (1 - taperFraction) / 2;
                break;
        }
        break;

    case TaperSide.Right:
        matrix.ScaleX = 1 / taperFraction;
        matrix.Persp0 = (1 - taperFraction) / (size.Width * taperFraction);

        switch (taperCorner)
        {
            case TaperCorner.RightOrBottom:
                break;

            case TaperCorner.LeftOrTop:
                matrix.SkewY = size.Height * matrix.Persp0;
                break;

            case TaperCorner.Both:
                matrix.SkewY = (size.Height / 2) * matrix.Persp0;
                break;
        }
        break;

    case TaperSide.Bottom:
        matrix.ScaleY = 1 / taperFraction;
        matrix.Persp1 = (1 - taperFraction) / (size.Height * taperFraction);

        switch (taperCorner)
        {
            case TaperCorner.RightOrBottom:
                break;

            case TaperCorner.LeftOrTop:
                matrix.SkewX = size.Width * matrix.Persp1;
                break;

            case TaperCorner.Both:
                matrix.SkewX = (size.Width / 2) * matrix.Persp1;
                break;
        }
        break;
    }

    return matrix;
}
}

```

This class is used in the **Taper Transform** page. The XAML file instantiates two **Picker** elements to select the enumeration values, and a **Slider** for choosing the taper fraction. The **PaintSurface** handler combines the taper transform with two translate transforms to make the transform relative to the upper-left corner of the bitmap:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    TaperSide taperSide = (TaperSide)taperSidePicker.SelectedItem;
    TaperCorner taperCorner = (TaperCorner)taperCornerPicker.SelectedItem;
    float taperFraction = (float)taperFractionSlider.Value;

    SKMatrix taperMatrix =
        TaperTransform.Make(new SKSize(bitmap.Width, bitmap.Height),
                           taperSide, taperCorner, taperFraction);

    // Display the matrix in the lower-right corner
    SKSize matrixSize = matrixDisplay.Measure(taperMatrix);

    matrixDisplay.Paint(canvas, taperMatrix,
        new SKPoint(info.Width - matrixSize.Width,
                   info.Height - matrixSize.Height));

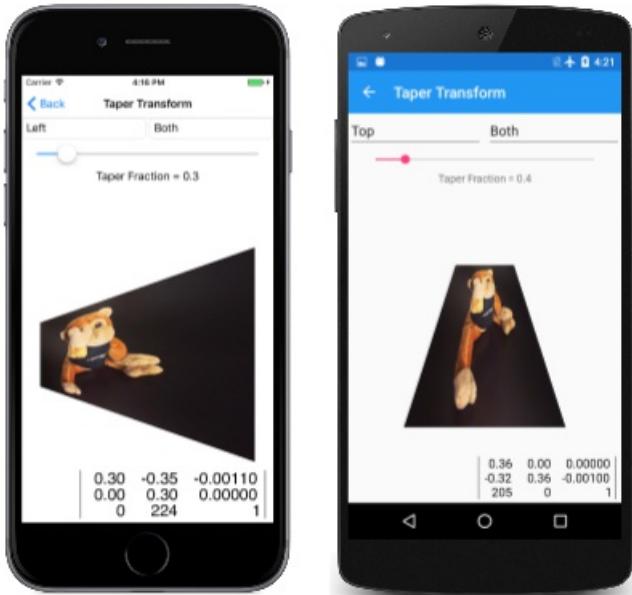
    // Center bitmap on canvas
    float x = (info.Width - bitmap.Width) / 2;
    float y = (info.Height - bitmap.Height) / 2;

    SKMatrix matrix = SKMatrix.MakeTranslation(-x, -y);
    SKMatrix.PostConcat(ref matrix, taperMatrix);
    SKMatrix.PostConcat(ref matrix, SKMatrix.MakeTranslation(x, y));

    canvas.SetMatrix(matrix);
    canvas.DrawBitmap(bitmap, x, y);
}

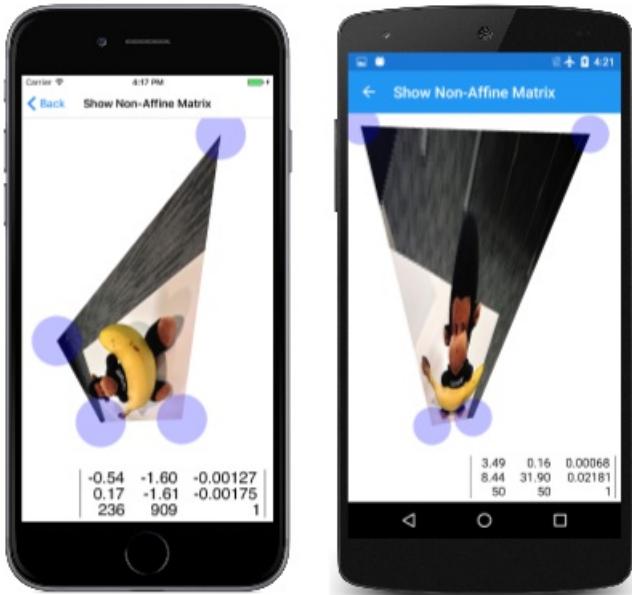
```

Here are some examples:



Another type of generalized non-affine transforms is 3D rotation, which is demonstrated in the next article, [3D Rotations](#).

The non-affine transform can transform a rectangle into any convex quadrilateral. This is demonstrated by the [Show Non-Affine Matrix](#) page. It is very similar to the [Show Affine Matrix](#) page from the [Matrix Transforms](#) article except that it has a fourth `TouchPoint` object to manipulate the fourth corner of the bitmap:



As long as you don't attempt to make an interior angle of one of the corners of the bitmap greater than 180 degrees, or make two sides cross each other, the program successfully calculates the transform using this method from the [ShowNonAffineMatrixPage](#) class:

```

static SKMatrix ComputeMatrix(SKSize size, SKPoint ptUL, SKPoint ptUR, SKPoint ptLL, SKPoint ptLR)
{
    // Scale transform
    SKMatrix S = SKMatrix.MakeScale(1 / size.Width, 1 / size.Height);

    // Affine transform
    SKMatrix A = new SKMatrix
    {
        ScaleX = ptUR.X - ptUL.X,
        SkewY = ptUR.Y - ptUL.Y,
        SkewX = ptLL.X - ptUL.X,
        ScaleY = ptLL.Y - ptUL.Y,
        TransX = ptUL.X,
        TransY = ptUL.Y,
        Persp2 = 1
    };

    // Non-Affine transform
    SKMatrix inverseA;
    A.TryInvert(out inverseA);
    SKPoint abPoint = inverseA.MapPoint(ptLR);
    float a = abPoint.X;
    float b = abPoint.Y;

    float scaleX = a / (a + b - 1);
    float scaleY = b / (a + b - 1);

    SKMatrix N = new SKMatrix
    {
        ScaleX = scaleX,
        ScaleY = scaleY,
        Persp0 = scaleX - 1,
        Persp1 = scaleY - 1,
        Persp2 = 1
    };

    // Multiply S * N * A
    SKMatrix result = SKMatrix.MakeIdentity();
    SKMatrix.PostConcat(ref result, S);
    SKMatrix.PostConcat(ref result, N);
    SKMatrix.PostConcat(ref result, A);

    return result;
}

```

For ease of calculation, this method obtains the total transform as a product of three separate transforms, which are symbolized here with arrows showing how these transforms modify the four corners of the bitmap:

$(0, 0) \rightarrow (0, 0) \rightarrow (0, 0) \rightarrow (x_0, y_0)$ (upper-left)

$(0, H) \rightarrow (0, 1) \rightarrow (0, 1) \rightarrow (x_1, y_1)$ (lower-left)

$(W, 0) \rightarrow (1, 0) \rightarrow (1, 0) \rightarrow (x_2, y_2)$ (upper-right)

$(W, H) \rightarrow (1, 1) \rightarrow (a, b) \rightarrow (x_3, y_3)$ (lower-right)

The final coordinates at the right are the four points associated with the four touch points. These are the final coordinates of the corners of the bitmap.

W and H represent the width and height of the bitmap. The first transform S simply scales the bitmap to a 1-pixel square. The second transform is the non-affine transform N , and the third is the affine transform A . That affine transform is based on three points, so it's just like the earlier affine [ComputeMatrix](#) method and doesn't involve the fourth row with the (a, b) point.

The `a` and `b` values are calculated so that the third transform is affine. The code obtains the inverse of the affine transform and then uses that to map the lower-right corner. That's the point (a, b) .

Another use of non-affine transforms is to mimic three-dimensional graphics. In the next article, [3D Rotations](#) you see how to rotate a two-dimensional graphic in 3D space.

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

3D Rotations in SkiaSharp

3/5/2021 • 14 minutes to read • [Edit Online](#)



[Download the sample](#)

Use non-affine transforms to rotate 2D objects in 3D space.

One common application of non-affine transforms is simulating the rotation of a 2D object in 3D space:



This job involves working with three-dimensional rotations, and then deriving a non-affine `SKMatrix` transform that performs these 3D rotations.

It is hard to develop this `SKMatrix` transform working solely within two dimensions. The job becomes much easier when this 3-by-3 matrix is derived from a 4-by-4 matrix used in 3D graphics. SkiaSharp includes the `SKMatrix44` class for this purpose, but some background in 3D graphics is necessary for understanding 3D rotations and the 4-by-4 transform matrix.

A three-dimensional coordinate system adds a third axis called Z. Conceptually, the Z axis is at right angles to the screen. Coordinate points in 3D space are indicated with three numbers: (x, y, z). In the 3D coordinate system used in this article, increasing values of X are to the right and increasing values of Y go down, just as in two dimensions. Increasing positive Z values come out of the screen. The origin is the upper-left corner, just as in 2D graphics. You can think of the screen as an XY plane with the Z axis at right angles to this plane.

This is called a left-hand coordinate system. If you point the forefinger for your left hand in the direction of positive X coordinates (to the right), and your middle finger in the direction of increasing Y coordinates (down), then your thumb points in the direction of increasing Z coordinates — extending out from the screen.

In 3D graphics, transforms are based on a 4-by-4 matrix. Here is the 4-by-4 identity matrix:

	1	0	0	0	
	0	1	0	0	
	0	0	1	0	
	0	0	0	1	

In working with a 4-by-4 matrix, it is convenient to identify the cells with their row and column numbers:

M11	M12	M13	M14
M21	M22	M23	M24
M31	M32	M33	M34
M41	M42	M43	M44

However, the SkiaSharp `Matrix44` class is a little different. The only way to set or get individual cell values in `SKMatrix44` is by using the `Item` indexer. The row and column indices are zero-based rather than one-based, and the rows and columns are swapped. The cell M14 in the above diagram is accessed using the indexer `[3, 0]` in a `SKMatrix44` object.

In a 3D graphics system, a 3D point (x, y, z) is converted to a 1-by-4 matrix for multiplying by the 4-by-4 transform matrix:

$$\begin{vmatrix} x & y & z & 1 \end{vmatrix} \times \begin{vmatrix} M11 & M12 & M13 & M14 \\ M21 & M22 & M23 & M24 \\ M31 & M32 & M33 & M34 \\ M41 & M42 & M43 & M44 \end{vmatrix} = \begin{vmatrix} x' & y' & z' & w' \end{vmatrix}$$

Analogous to 2D transforms that take place in three dimensions, 3D transforms are assumed to take place in four dimensions. The fourth dimension is referred to as W, and the 3D space is assumed to exist within the 4D space where W coordinates are equal to 1. The transform formulas are as follows:

$$\begin{aligned} x' &= M11 \cdot x + M21 \cdot y + M31 \cdot z + M41 \\ y' &= M12 \cdot x + M22 \cdot y + M32 \cdot z + M42 \\ z' &= M13 \cdot x + M23 \cdot y + M33 \cdot z + M43 \\ w' &= M14 \cdot x + M24 \cdot y + M34 \cdot z + M44 \end{aligned}$$

It's obvious from the transform formulas that the cells `M11`, `M22`, `M33` are scaling factors in the X, Y, and Z directions, and `M41`, `M42`, and `M43` are translation factors in the X, Y, and Z directions.

To convert these coordinates back to the 3D space where W equals 1, the x', y', and z' coordinates are all divided by w':

$$\begin{aligned} x'' &= x' / w' \\ y'' &= y' / w' \\ z'' &= z' / w' \\ w'' &= w' / w' = 1 \end{aligned}$$

That division by w' provides perspective in 3D space. If w' equals 1, then no perspective occurs.

Rotations in 3D space can be quite complex, but the simplest rotations are those around the X, Y, and Z axes. A rotation of angle α around the X axis is this matrix:

1	0	0	0
0	$\cos(\alpha)$	$\sin(\alpha)$	0
0	$-\sin(\alpha)$	$\cos(\alpha)$	0
0	0	0	1

Values of X remain the same when subjected to this transform. Rotation around the Y axis leaves values of Y unchanged:

$\cos(\alpha)$	0	$-\sin(\alpha)$	0
0	1	0	0
$\sin(\alpha)$	0	$\cos(\alpha)$	0
0	0	0	1

Rotation around the Z axis is the same as in 2D graphics:

$\cos(\alpha)$	$\sin(\alpha)$	0	0
$-\sin(\alpha)$	$\cos(\alpha)$	0	0
0	0	1	0
0	0	0	1

The direction of rotation is implied by the handedness of the coordinate system. This is a left-handed system, so if you point the thumb of your left hand towards increasing values for a particular axis — to the right for rotation around the X axis, down for rotation around the Y axis, and towards you for rotation around the Z axis — then the curve of your other fingers indicates the direction of rotation for positive angles.

`SKMatrix44` has generalized static `CreateRotation` and `CreateRotationDegrees` methods that allow you to specify the axis around which the rotation occurs:

```
public static SKMatrix44 CreateRotationDegrees (Single x, Single y, Single z, Single degrees)
```

For rotation around the X axis, set the first three arguments to 1, 0, 0. For rotation around the Y axis, set them to 0, 1, 0, and for rotation around the Z axis, set them to 0, 0, 1.

The fourth column of the 4-by-4 is for perspective. The `SKMatrix44` has no methods for creating perspective transforms, but you can create one yourself using the following code:

```
SKMatrix44 perspectiveMatrix = SKMatrix44.CreateIdentity();
perspectiveMatrix[3, 2] = -1 / depth;
```

The reason for the argument name `depth` will be evident shortly. That code creates the matrix:

1	0	0	0
0	1	0	0
0	0	1	-1/depth
0	0	0	1

The transform formulas result in the following calculation of w' :

```
w' = -z / depth + 1
```

This serves to reduce X and Y coordinates when values of Z are less than zero (conceptually behind the XY plane) and to increase X and Y coordinates for positive values of Z. When the Z coordinate equals `depth`, then w' is zero, and coordinates become infinite. Three-dimensional graphics systems are built around a camera metaphor, and the `depth` value here represents the distance of the camera from the origin of the coordinate system. If a graphical object has a Z coordinate that is `depth` units from the origin, it is conceptually touching the lens of the camera and becomes infinitely large.

Keep in mind that you'll probably be using this `perspectiveMatrix` value in combination with rotation matrices. If a graphics object being rotated has X or Y coordinates greater than `depth`, then the rotation of this object in 3D space is likely to involve Z coordinates greater than `depth`. This must be avoided! When creating `perspectiveMatrix` you want to set `depth` to a value sufficiently large for all coordinates in the graphics object no matter how it is rotated. This ensures that there is never any division by zero.

Combining 3D rotations and perspective requires multiplying 4-by-4 matrices together. For this purpose, `SKMatrix44` defines concatenation methods. If `A` and `B` are `SKMatrix44` objects, then the following code sets `A` equal to $A \times B$:

```
A.PostConcat(B);
```

When a 4-by-4 transform matrix is used in a 2D graphics system, it is applied to 2D objects. These objects are flat and are assumed to have Z coordinates of zero. The transform multiplication is a little simpler than the transform shown earlier:

$$\begin{vmatrix} & M_{11} & M_{12} & M_{13} & M_{14} \\ | & x & y & 0 & 1 \\ | & x & M_{21} & M_{22} & M_{23} & M_{24} \\ | & & M_{31} & M_{32} & M_{33} & M_{34} \\ | & & M_{41} & M_{42} & M_{43} & M_{44} \end{vmatrix} = \begin{vmatrix} x' & y' & z' & w' \end{vmatrix}$$

That 0 value for z results in transform formulas that do not involve any cells in the third row of the matrix:

$$x' = M_{11} \cdot x + M_{21} \cdot y + M_{41}$$

$$y' = M_{12} \cdot x + M_{22} \cdot y + M_{42}$$

$$z' = M_{13} \cdot x + M_{23} \cdot y + M_{43}$$

$$w' = M_{14} \cdot x + M_{24} \cdot y + M_{44}$$

Moreover, the z' coordinate is irrelevant here as well. When a 3D object is displayed in a 2D graphics system, it is collapsed to a two-dimensional object by ignoring the Z coordinate values. The transform formulas are really just these two:

$$x'' = x' / w'$$

$$y'' = y' / w'$$

This means that the third row *and* third column of the 4-by-4 matrix can be ignored.

But if that is so, why is the 4-by-4 matrix even necessary in the first place?

Although the third row and third column of the 4-by-4 are irrelevant for two-dimensional transforms, the third row and column *do* play a role prior to that when various `SKMatrix44` values are multiplied together. For example, suppose that you multiply rotation around the Y axis with the perspective transform:

$$\begin{vmatrix} \cos(\alpha) & 0 & -\sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1/\text{depth} \\ 0 & 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} \cos(\alpha) & 0 & -\sin(\alpha) & \sin(\alpha)/\text{depth} \\ 0 & 1 & 0 & 0 \\ \sin(\alpha) & 0 & \cos(\alpha) & -\cos(\alpha)/\text{depth} \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

In the product, the cell `M14` now contains a perspective value. If you want to apply that matrix to 2D objects, the third row and column are eliminated to convert it into a 3-by-3 matrix:

$$\begin{vmatrix} \cos(\alpha) & 0 & \sin(\alpha)/\text{depth} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Now it can be used to transform a 2D point:

$$\begin{vmatrix} & \cos(\alpha) & 0 & \sin(\alpha)/\text{depth} \\ | & x & y & 1 \\ | & x & 0 & 0 \\ | & 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} & x' & y' & z' \end{vmatrix}$$

The transform formulas are:

$$x' = \cos(\alpha) \cdot x$$

$$y' = y$$

$$z' = (\sin(\alpha)/\text{depth}) \cdot x + 1$$

Now divide everything by z' :

$$x'' = \cos(\alpha) \cdot x / ((\sin(\alpha)/\text{depth}) \cdot x + 1)$$

$$y'' = y / ((\sin(\alpha)/\text{depth}) \cdot x + 1)$$

When 2D objects are rotated with a positive angle around the Y axis, then positive X values recede to the background while negative X values come to the foreground. The X values seem to move closer to the Y axis (which is governed by the cosine value) as coordinates furthest from the Y axis becomes smaller or larger as they move further from the viewer or closer to the viewer.

When using `SKMatrix44`, perform all the 3D rotation and perspective operations by multiplying various `SKMatrix44` values. Then you can extract a two-dimensional 3-by-3 matrix from the 4-by-4 matrix using the `Matrix` property of the `SKMatrix44` class. This property returns a familiar `SKMatrix` value.

The **Rotation 3D** page lets you experiment with 3D rotation. The `Rotation3DPage.xaml` file instantiates four sliders to set rotation around the X, Y, and Z axes, and to set a depth value:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Transforms.Rotation3DPage"
    Title="Rotation 3D">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

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

                <Style TargetType="Slider">
                    <Setter Property="Margin" Value="20, 0" />
                    <Setter Property="Maximum" Value="360" />
                </Style>
            </ResourceDictionary>
        </Grid.Resources>

        <Slider x:Name="xRotateSlider"
            Grid.Row="0"
            Minimum="0" Maximum="360" Value="0" />
        <Slider x:Name="yRotateSlider"
            Grid.Row="1"
            Minimum="0" Maximum="360" Value="0" />
        <Slider x:Name="zRotateSlider"
            Grid.Row="2"
            Minimum="0" Maximum="360" Value="0" />
        <Label x:Name="depthLabel"
            Grid.Row="3"
            Text="Depth" />
        <Slider x:Name="depthSlider"
            Grid.Row="4"
            Minimum="0" Maximum="1000" Value="1000" />
    </Grid>
</ContentPage>
```

```

        valueChanged="OnSliderValueChanged" />

<Label Text="{Binding Source={x:Reference xRotateSlider},
                     Path=Value,
                     StringFormat='X-Axis Rotation = {0:F0}'}"
       Grid.Row="1" />

<Slider x:Name="yRotateSlider"
         Grid.Row="2"
         ValueChanged="OnSliderValueChanged" />

<Label Text="{Binding Source={x:Reference yRotateSlider},
                     Path=Value,
                     StringFormat='Y-Axis Rotation = {0:F0}'}"
       Grid.Row="3" />

<Slider x:Name="zRotateSlider"
         Grid.Row="4"
         ValueChanged="OnSliderValueChanged" />

<Label Text="{Binding Source={x:Reference zRotateSlider},
                     Path=Value,
                     StringFormat='Z-Axis Rotation = {0:F0}'}"
       Grid.Row="5" />

<Slider x:Name="depthSlider"
         Grid.Row="6"
         Maximum="2500"
         Minimum="250"
         ValueChanged="OnSliderValueChanged" />

<Label Grid.Row="7"
       Text="{Binding Source={x:Reference depthSlider},
                     Path=Value,
                     StringFormat='Depth = {0:F0}'}" />

<skia:SKCanvasView x:Name="canvasView"
                     Grid.Row="8"
                     PaintSurface="OnCanvasViewPaintSurface" />

```

</Grid>

</ContentPage>

Notice that the `depthSlider` is initialized with a `Minimum` value of 250. This implies that the 2D object being rotated here has X and Y coordinates restricted to a circle defined by a 250-pixel radius around the origin. Any rotation of this object in 3D space will always result in coordinate values less than 250.

The [Rotation3DPage.cs](#) code-behind file loads in a bitmap that is 300 pixels square:

```
public partial class Rotation3DPage : ContentPage
{
    SKBitmap bitmap;

    public Rotation3DPage()
    {
        InitializeComponent();

        string resourceId = "SkiaSharpFormsDemos.Media.SeatedMonkey.jpg";
        Assembly assembly = GetType().GetTypeInfo().Assembly;

        using (Stream stream = assembly.GetManifestResourceStream(resourceId))
        {
            bitmap = SKBitmap.Decode(stream);
        }
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        if (canvasView != null)
        {
            canvasView.InvalidateSurface();
        }
    }
    ...
}
```

If the 3D transform is centered on this bitmap, then X and Y coordinates range between –150 and 150, while the corners are 212 pixels from the center, so everything is within the 250-pixel radius.

The `PaintSurface` handler creates `SKMatrix44` objects based on the sliders and multiplies them together using `PostConcat`. The `SKMatrix` value extracted from the final `SKMatrix44` object is surrounded by translate transforms to center the rotation in the center of the screen:

```

public partial class Rotation3DPage : ContentPage
{
    SKBitmap bitmap;

    public Rotation3DPage()
    {
        InitializeComponent();

        string resourceId = "SkiaSharpFormsDemos.Media.SeatedMonkey.jpg";
        Assembly assembly = GetType().GetTypeInfo().Assembly;

        using (Stream stream = assembly.GetManifestResourceStream(resourceId))
        {
            bitmap = SKBitmap.Decode(stream);
        }
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        if (canvasView != null)
        {
            canvasView.InvalidateSurface();
        }
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Find center of canvas
        float xCenter = info.Width / 2;
        float yCenter = info.Height / 2;

        // Translate center to origin
        SKMatrix matrix = SKMatrix.MakeTranslation(-xCenter, -yCenter);

        // Use 3D matrix for 3D rotations and perspective
        SKMatrix44 matrix44 = SKMatrix44.CreateIdentity();
        matrix44.PostConcat(SKMatrix44.CreateRotationDegrees(1, 0, 0, (float)xRotateSlider.Value));
        matrix44.PostConcat(SKMatrix44.CreateRotationDegrees(0, 1, 0, (float)yRotateSlider.Value));
        matrix44.PostConcat(SKMatrix44.CreateRotationDegrees(0, 0, 1, (float)zRotateSlider.Value));

        SKMatrix44 perspectiveMatrix = SKMatrix44.CreateIdentity();
        perspectiveMatrix[3, 2] = -1 / (float)depthSlider.Value;
        matrix44.PostConcat(perspectiveMatrix);

        // Concatenate with 2D matrix
        SKMatrix.PostConcat(ref matrix, matrix44.Matrix);

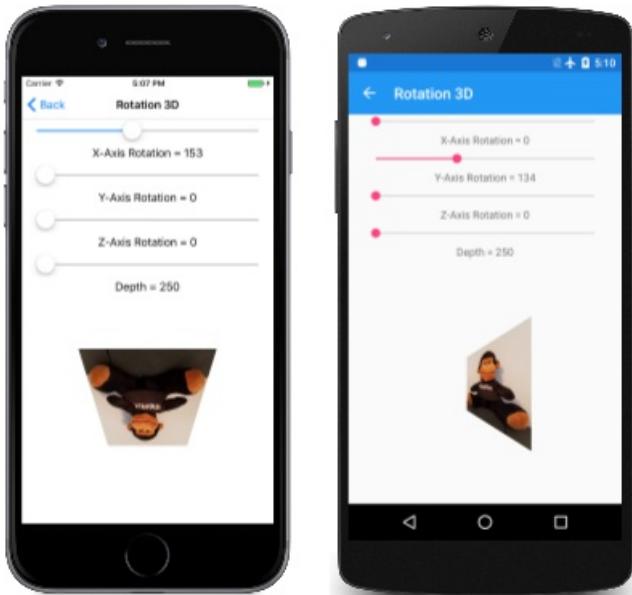
        // Translate back to center
        SKMatrix.PostConcat(ref matrix,
            SKMatrix.MakeTranslation(xCenter, yCenter));

        // Set the matrix and display the bitmap
        canvas.SetMatrix(matrix);
        float xBitmap = xCenter - bitmap.Width / 2;
        float yBitmap = yCenter - bitmap.Height / 2;
        canvas.DrawBitmap(bitmap, xBitmap, yBitmap);
    }
}

```

When you experiment with the fourth slider, you'll notice that the different depth settings don't move the object

further away from the viewer, but instead alter the extent of the perspective effect:



The **Animated Rotation 3D** also uses `SKMatrix44` to animate a text string in 3D space. The `textPaint` object set as a field is used in the constructor to determine the bounds of the text:

```
public class AnimatedRotation3DPage : ContentPage
{
    SKCanvasView canvasView;
    float xRotationDegrees, yRotationDegrees, zRotationDegrees;
    string text = "SkiaSharp";
    SKPaint textPaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Black,
        TextSize = 100,
        StrokeWidth = 3,
    };
    SKRect textBounds;

    public AnimatedRotation3DPage()
    {
        Title = "Animated Rotation 3D";

        canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;

        // Measure the text
        textPaint.MeasureText(text, ref textBounds);
    }
    ...
}
```

The `OnAppearing` override defines three `Xamarin.Forms` `Animation` objects to animate the `xRotationDegrees`, `yRotationDegrees`, and `zRotationDegrees` fields at different rates. Notice that the periods of these animations are set to prime numbers (5 seconds, 7 seconds, and 11 seconds) so the overall combination only repeats every 385 seconds, or more than 10 minutes:

```
public class AnimatedRotation3DPage : ContentPage
{
    ...
    protected override void OnAppearing()
    {
        base.OnAppearing();

        new Animation((value) => xRotationDegrees = 360 * (float)value).
            Commit(this, "xRotationAnimation", length: 5000, repeat: () => true);

        new Animation((value) => yRotationDegrees = 360 * (float)value).
            Commit(this, "yRotationAnimation", length: 7000, repeat: () => true);

        new Animation((value) =>
        {
            zRotationDegrees = 360 * (float)value;
            canvasView.InvalidateSurface();
        }).Commit(this, "zRotationAnimation", length: 11000, repeat: () => true);
    }

    protected override void OnDisappearing()
    {
        base.OnDisappearing();
        this.AbortAnimation("xRotationAnimation");
        this.AbortAnimation("yRotationAnimation");
        this.AbortAnimation("zRotationAnimation");
    }
    ...
}
```

As in the previous program, the `PaintCanvas` handler creates `SKMatrix44` values for rotation and perspective, and multiplies them together:

```

public class AnimatedRotation3DPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Find center of canvas
        float xCenter = info.Width / 2;
        float yCenter = info.Height / 2;

        // Translate center to origin
        SKMatrix matrix = SKMatrix.MakeTranslation(-xCenter, -yCenter);

        // Scale so text fits
        float scale = Math.Min(info.Width / textBounds.Width,
                               info.Height / textBounds.Height);
        SKMatrix.PostConcat(ref matrix, SKMatrix.MakeScale(scale, scale));

        // Calculate composite 3D transforms
        float depth = 0.75f * scale * textBounds.Width;

        SKMatrix44 matrix44 = SKMatrix44.CreateIdentity();
        matrix44.PostConcat(SKMatrix44.CreateRotationDegrees(1, 0, 0, xRotationDegrees));
        matrix44.PostConcat(SKMatrix44.CreateRotationDegrees(0, 1, 0, yRotationDegrees));
        matrix44.PostConcat(SKMatrix44.CreateRotationDegrees(0, 0, 1, zRotationDegrees));

        SKMatrix44 perspectiveMatrix = SKMatrix44.CreateIdentity();
        perspectiveMatrix[3, 2] = -1 / depth;
        matrix44.PostConcat(perspectiveMatrix);

        // Concatenate with 2D matrix
        SKMatrix.PostConcat(ref matrix, matrix44.Matrix);

        // Translate back to center
        SKMatrix.PostConcat(ref matrix,
                           SKMatrix.MakeTranslation(xCenter, yCenter));

        // Set the matrix and display the text
        canvas.SetMatrix(matrix);
        float xText = xCenter - textBounds.MidX;
        float yText = yCenter - textBounds.MidY;
        canvas.DrawText(text, xText, yText, textPaint);
    }
}

```

This 3D rotation is surrounded with several 2D transforms to move the center of rotation to the center of the screen, and to scale the size of the text string so that it is the same width as the screen:



Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

SkiaSharp Curves and Paths

3/5/2021 • 2 minutes to read • [Edit Online](#)

 [Download the sample](#)

Learn how to use SkiaSharp to draw curves and use path features

The exploration of `SKPath` methods and properties began in the [SkiaSharp Lines and Paths](#) article. The articles here continue with methods that add curves to an `SKPath` object, and exploit other powerful path features. You'll see how you can specify an entire path in a concise text string, how to use path effects, and how to dig into path internals.

All the sample programs in this section can be found in the [SkiaSharp Curves and Paths](#) page of the [SkiaSharpFormsDemos](#) program, and in the [Curves](#) folder of the solution.

Three Ways to Draw an Arc

Learn how to use SkiaSharp to define arcs in three different ways.

Three Types of Bézier Curves

Explore how to use SkiaSharp to render cubic, quadratic, and conic Bézier curves

SVG Path Data

Define paths using text strings in the Scalable Vector Graphics format

Clipping with Paths and Regions

Use paths to clip graphics to specific areas, and to create regions.

Path Effects

Discover the various path effects that allow paths to be used for stroking and filling.

Paths and Text

Explore the intersection of paths and text

Path Information and Enumeration

Get information about paths and enumerate the contents

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Three Ways to Draw an Arc

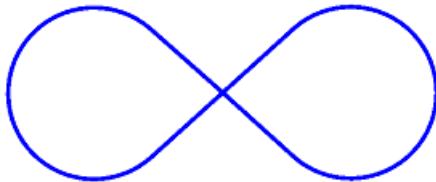
3/5/2021 • 16 minutes to read • [Edit Online](#)



[Download the sample](#)

Learn how to use SkiaSharp to define arcs in three different ways

An arc is a curve on the circumference of an ellipse, such as the rounded parts of this infinity sign:



Despite the simplicity of that definition, there is no way to define an arc-drawing function that satisfies every need, and hence, no consensus among graphics systems of the best way to draw an arc. For this reason, the `SKPath` class does not restrict itself to just one approach.

`SKPath` defines an `AddArc` method, five different `ArcTo` methods, and two relative `RArcTo` methods. These methods fall into three categories, representing three very different approaches to specifying an arc. Which one you use depends on the information available to define the arc, and how this arc fits in with the other graphics that you're drawing.

The Angle Arc

The angle arc approach to drawing arcs requires that you specify a rectangle that bounds an ellipse. The arc on the circumference of this ellipse is indicated by angles from the center of the ellipse that indicate the beginning of the arc and its length. Two different methods draw angle arcs. These are the `AddArc` method and the `ArcTo` method:

```
public void AddArc (SKRect oval, Single startAngle, Single sweepAngle)  
  
public void ArcTo (SKRect oval, Single startAngle, Single sweepAngle, Boolean forceMoveTo)
```

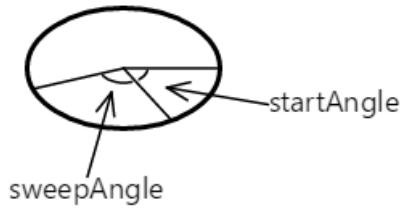
These methods are identical to the Android `AddArc` and `[ArcTo]xref:Android.Graphics.Path.ArcTo*` methods. The iOS `AddArc` method is similar but is restricted to arcs on the circumference of a circle rather than generalized to an ellipse.

Both methods begin with an `SKRect` value that defines both the location and size of an ellipse:

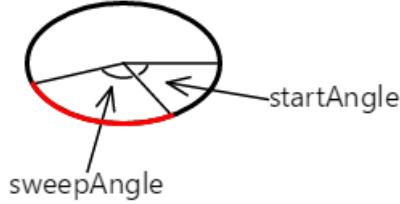


The arc is a part of the circumference of this ellipse.

The `startAngle` argument is a clockwise angle in degrees relative to a horizontal line drawn from the center of the ellipse to the right. The `sweepAngle` argument is relative to the `startAngle`. Here are `startAngle` and `sweepAngle` values of 60 degrees and 100 degrees, respectively:



The arc begins at the start angle. Its length is governed by the sweep angle. The arc is shown here in red:



The curve added to the path with the `AddArc` or `ArcTo` method is simply that part of the ellipse's circumference:



The `startAngle` or `sweepAngle` arguments can be negative: The arc is clockwise for positive values of `sweepAngle` and counter-clockwise for negative values.

However, `AddArc` does *not* define a closed contour. If you call `LineTo` after `AddArc`, a line is drawn from the end of the arc to the point in the `LineTo` method, and the same is true of `ArcTo`.

`AddArc` automatically starts a new contour and is functionally equivalent to a call to `ArcTo` with a final argument of `true`:

```
path.ArcTo (oval, startAngle, sweepAngle, true);
```

That last argument is called `forceMoveTo`, and it effectively causes a `MoveTo` call at the beginning of the arc. That begins a new contour. That is not the case with a last argument of `false`:

```
path.ArcTo (oval, startAngle, sweepAngle, false);
```

This version of `ArcTo` draws a line from the current position to the beginning of the arc. This means that the arc can be somewhere in the middle of a larger contour.

The **Angle Arc** page lets you use two sliders to specify the start and sweep angles. The XAML file instantiates two `Slider` elements and an `SKCanvasView`. The `PaintCanvas` handler in the `AngleArcPage.xaml.cs` file draws both the oval and the arc using two `SKPaint` objects defined as fields:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

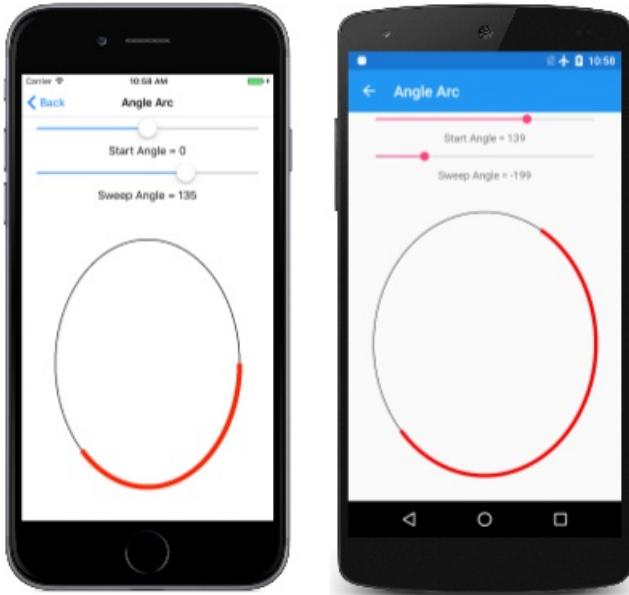
    SKRect rect = new SKRect(100, 100, info.Width - 100, info.Height - 100);
    float startAngle = (float)startAngleSlider.Value;
    float sweepAngle = (float)sweepAngleSlider.Value;

    canvas.DrawOval(rect, outlinePaint);

    using (SKPath path = new SKPath())
    {
        path.AddArc(rect, startAngle, sweepAngle);
        canvas.DrawPath(path, arcPaint);
    }
}

```

As you can see, both the start angle and the sweep angle can take on negative values:



This approach to generating an arc is algorithmically the simplest, and it's easy to derive the parametric equations that describe the arc. Knowing the size and location of the ellipse, and the start and sweep angles, the start and end points of the arc can be calculated using simple trigonometry:

```
x = oval.MidX + (oval.Width / 2) * cos(angle)
```

```
y = oval.MidY + (oval.Height / 2) * sin(angle)
```

The `angle` value is either `startAngle` or `startAngle + sweepAngle`.

The use of two angles to define an arc is best for cases where you know the angular length of the arc that you want to draw, for example, to make a pie chart. The **Exploded Pie Chart** page demonstrates this. The [ExplodedPieChartPage](#) class uses an internal class to define some fabricated data and colors:

```
class ChartData
{
    publicChartData(int value, SKColor color)
    {
        Value = value;
        Color = color;
    }

    public int Value { private set; get; }

    public SKColor Color { private set; get; }
}

ChartData[] chartData =
{
    new ChartData(45, SKColors.Red),
    new ChartData(13, SKColors.Green),
    new ChartData(27, SKColors.Blue),
    new ChartData(19, SKColors.Magenta),
    new ChartData(40, SKColors.Cyan),
    new ChartData(22, SKColors.Brown),
    new ChartData(29, SKColors.Gray)
};
```

The `PaintSurface` handler first loops through the items to calculate a `totalValues` number. From that, it can determine each item's size as the fraction of the total, and convert that to an angle:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    int totalValues = 0;

    foreach (ChartData item in chartData)
    {
        totalValues += item.Value;
    }

    SKPoint center = new SKPoint(info.Width / 2, info.Height / 2);
    float explodeOffset = 50;
    float radius = Math.Min(info.Width / 2, info.Height / 2) - 2 * explodeOffset;
    SKRect rect = new SKRect(center.X - radius, center.Y - radius,
                           center.X + radius, center.Y + radius);

    float startAngle = 0;

    foreach (ChartData item in chartData)
    {
        float sweepAngle = 360f * item.Value / totalValues;

        using (SKPath path = new SKPath())
        using (SKPaint fillPaint = new SKPaint())
        using (SKPaint outlinePaint = new SKPaint())
        {
            path.MoveTo(center);
            path.ArcTo(rect, startAngle, sweepAngle, false);
            path.Close();

            fillPaint.Style = SKPaintStyle.Fill;
            fillPaint.Color = item.Color;

            outlinePaint.Style = SKPaintStyle.Stroke;
            outlinePaint.StrokeWidth = 5;
            outlinePaint.Color = SKColors.Black;

            // Calculate "explode" transform
            float angle = startAngle + 0.5f * sweepAngle;
            float x = explodeOffset * (float)Math.Cos(Math.PI * angle / 180);
            float y = explodeOffset * (float)Math.Sin(Math.PI * angle / 180);

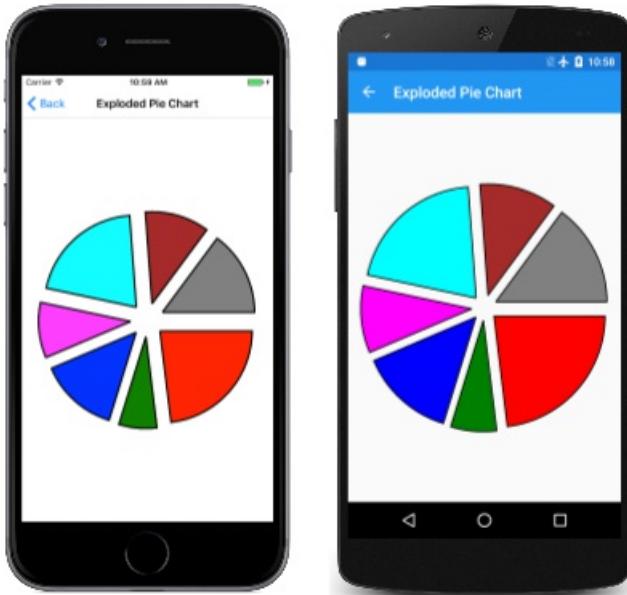
            canvas.Save();
            canvas.Translate(x, y);

            // Fill and stroke the path
            canvas.DrawPath(path, fillPaint);
            canvas.DrawPath(path, outlinePaint);
            canvas.Restore();
        }

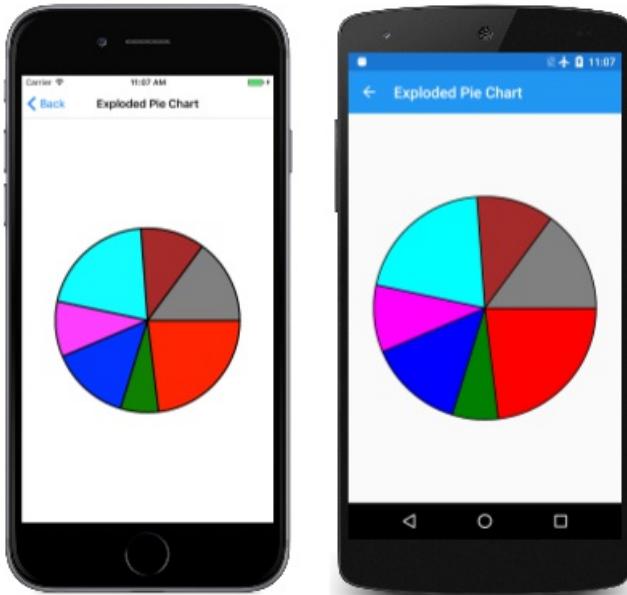
        startAngle += sweepAngle;
    }
}

```

A new `SKPath` object is created for each pie slice. The path consists of a line from the center, then an `ArcTo` to draw the arc, and another line back to the center results from the `close` call. This program displays "exploded" pie slices by moving them all out from the center by 50 pixels. That task requires a vector in the direction of the midpoint of the sweep angle for each slice:



To see what it looks like without the "explosion," simply comment out the `Translate` call:



The Tangent Arc

The second type of arc supported by `SKPath` is the *tangent arc*, so called because the arc is the circumference of a circle that is tangent to two connected lines.

A tangent arc is added to a path with a call to the `ArcTo` method with two `SKPoint` parameters, or the `ArcTo` overload with separate `Single` parameters for the points:

```
public void ArcTo (SKPoint point1, SKPoint point2, Single radius)
public void ArcTo (Single x1, Single y1, Single x2, Single y2, Single radius)
```

This `ArcTo` method is similar to the PostScript `arct` (page 532) function and the iOS `AddArcToPoint` method.

The `ArcTo` method involves three points:

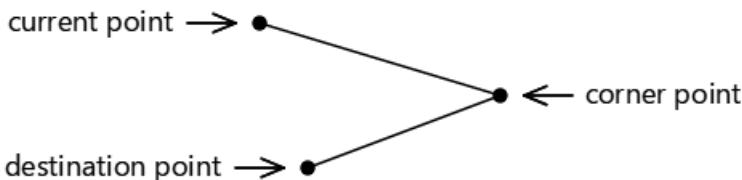
- The current point of the contour, or the point $(0, 0)$ if `MoveTo` has not been called
- The first point argument to the `ArcTo` method, called the *corner point*
- The second point argument to `ArcTo`, called the *destination point*.

current point → •

• ← corner point

destination point → •

These three points define two connected lines:



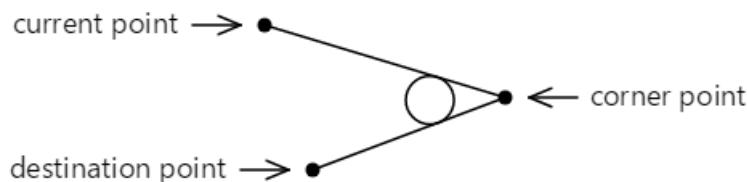
If the three points are colinear — that is, if they lie on the same straight line — no arc will be drawn.

The `ArcTo` method also includes a `radius` parameter. This defines the radius of a circle:

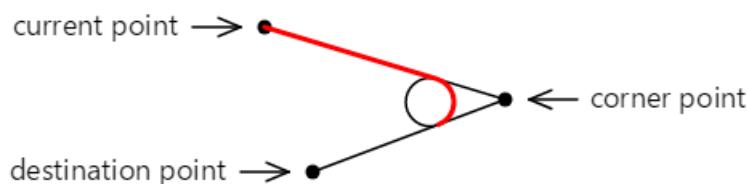


The tangent arc is not generalized for an ellipse.

If the two lines meet at any angle, that circle can be inserted between those lines so that it is tangent to both lines:



The curve that is added to the contour does not touch either of the points specified in the `ArcTo` method. It consists of a straight line from the current point to the first tangent point, and an arc that ends at the second tangent point, shown here in red:



Here's the final straight line and arc that is added to the contour:



The contour can be continued from the second tangent point.

The [Tangent Arc](#) page allows you to experiment with the tangent arc. This is the first of several pages that derive from [InteractivePage](#), which defines a few handy [SKPaint](#) objects and performs [TouchPoint](#) processing:

```

public class InteractivePage : ContentPage
{
    protected SKCanvasView baseCanvasView;
    protected TouchPoint[] touchPoints;

    protected SKPaint strokePaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Black,
        StrokeWidth = 3
    };

    protected SKPaint redStrokePaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Red,
        StrokeWidth = 15
    };

    protected SKPaint dottedStrokePaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Black,
        StrokeWidth = 3,
        PathEffect = SKPathEffect.CreateDash(new float[] { 7, 7 }, 0)
    };

    protected void OnTouchEffectAction(object sender, TouchEventArgs args)
    {
        bool touchPointMoved = false;

        foreach (TouchPoint touchPoint in touchPoints)
        {
            float scale = baseCanvasView.CanvasSize.Width / (float)baseCanvasView.Width;
            SKPoint point = new SKPoint(scale * (float)args.Location.X,
                                         scale * (float)args.Location.Y);
            touchPointMoved |= touchPoint.ProcessTouchEvent(args.Id, args.Type, point);
        }

        if (touchPointMoved)
        {
            baseCanvasView.InvalidateSurface();
        }
    }
}

```

The `TangentArcPage` class derives from `InteractivePage`. The constructor in the `TangentArcPage.xaml.cs` file is responsible for instantiating and initializing the `touchPoints` array, and setting `baseCanvasView` (in `InteractivePage`) to the `SKCanvasView` object instantiated in the `TangentArcPage.xaml` file:

```

public partial class TangentArcPage : InteractivePage
{
    public TangentArcPage()
    {
        touchPoints = new TouchPoint[3];

        for (int i = 0; i < 3; i++)
        {
            TouchPoint touchPoint = new TouchPoint
            {
                Center = new SKPoint(i == 0 ? 100 : 500,
                                      i != 2 ? 100 : 500)
            };
            touchPoints[i] = touchPoint;
        }

        InitializeComponent();

        baseCanvasView = canvasView;
        radiusSlider.Value = 100;
    }

    void sliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        if (canvasView != null)
        {
            canvasView.InvalidateSurface();
        }
    }
}

```

The `PaintSurface` handler uses the `ArcTo` method to draw the arc based on the touch points and a `Slider`, but also algorithmically calculates the circle that the angle is based on:

```

public partial class TangentArcPage : InteractivePage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Draw the two lines that meet at an angle
        using (SKPath path = new SKPath())
        {
            path.MoveTo(touchPoints[0].Center);
            path.LineTo(touchPoints[1].Center);
            path.LineTo(touchPoints[2].Center);
            canvas.DrawPath(path, dottedStrokePaint);
        }

        // Draw the circle that the arc wraps around
        float radius = (float)radiusSlider.Value;

        SKPoint v1 = Normalize(touchPoints[0].Center - touchPoints[1].Center);
        SKPoint v2 = Normalize(touchPoints[2].Center - touchPoints[1].Center);

        double dotProduct = v1.X * v2.X + v1.Y * v2.Y;
        double angleBetween = Math.Acos(dotProduct);
        float hypotenuse = radius / (float)Math.Sin(angleBetween / 2);
        SKPoint vMid = Normalize(new SKPoint((v1.X + v2.X) / 2, (v1.Y + v2.Y) / 2));
        SKPoint center = new SKPoint(touchPoints[1].Center.X + vMid.X * hypotenuse,
                                      touchPoints[1].Center.Y + vMid.Y * hypotenuse);

        canvas.DrawCircle(center.X, center.Y, radius, this.strokePaint);

        // Draw the tangent arc
        using (SKPath path = new SKPath())
        {
            path.MoveTo(touchPoints[0].Center);
            path.ArcTo(touchPoints[1].Center, touchPoints[2].Center, radius);
            canvas.DrawPath(path, redStrokePaint);
        }

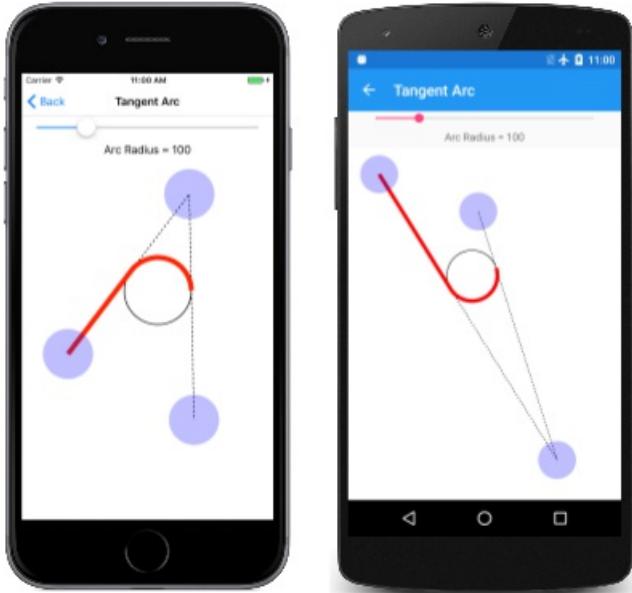
        foreach (TouchPoint touchPoint in touchPoints)
        {
            touchPoint.Paint(canvas);
        }
    }

    // Vector methods
    SKPoint Normalize(SKPoint v)
    {
        float magnitude = Magnitude(v);
        return new SKPoint(v.X / magnitude, v.Y / magnitude);
    }

    float Magnitude(SKPoint v)
    {
        return (float)Math.Sqrt(v.X * v.X + v.Y * v.Y);
    }
}

```

Here's the **Tangent Arc** page running:



The tangent arc is ideal for creating rounded corners, such as a rounded rectangle. Because `SKPath` already includes an `AddRoundedRect` method, the [Rounded Heptagon](#) page demonstrates how to use `ArcTo` for rounding the corners of a seven-sided polygon. (The code is generalized for any regular polygon.)

The `PaintSurface` handler of the [RoundedHeptagonPage](#) class contains one `for` loop to calculate the coordinates of the seven vertices of the heptagon, and a second to calculate the midpoints of the seven sides from these vertices. These midpoints are then used to construct the path:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    float cornerRadius = 100;
    int numVertices = 7;
    float radius = 0.45f * Math.Min(info.Width, info.Height);

    SKPoint[] vertices = new SKPoint[numVertices];
    SKPoint[] midPoints = new SKPoint[numVertices];

    double vertexAngle = -0.5f * Math.PI;           // straight up

    // Coordinates of the vertices of the polygon
    for (int vertex = 0; vertex < numVertices; vertex++)
    {
        vertices[vertex] = new SKPoint(radius * (float)Math.Cos(vertexAngle),
                                         radius * (float)Math.Sin(vertexAngle));
        vertexAngle += 2 * Math.PI / numVertices;
    }

    // Coordinates of the midpoints of the sides connecting the vertices
    for (int vertex = 0; vertex < numVertices; vertex++)
    {
        int prevVertex = (vertex + numVertices - 1) % numVertices;
        midPoints[vertex] = new SKPoint((vertices[prevVertex].X + vertices[vertex].X) / 2,
                                         (vertices[prevVertex].Y + vertices[vertex].Y) / 2);
    }

    // Create the path
    using (SKPath path = new SKPath())
    {
        // Begin at the first midpoint
        path.MoveTo(midPoints[0]);

        for (int vertex = 0; vertex < numVertices; vertex++)
        {
            SKPoint nextMidPoint = midPoints[(vertex + 1) % numVertices];

            // Draws a line from the current point, and then the arc
            path.ArcTo(vertices[vertex], nextMidPoint, cornerRadius);

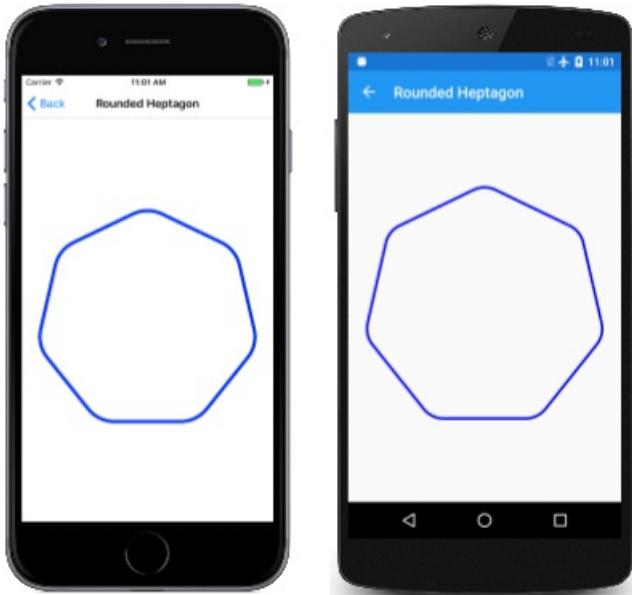
            // Connect the arc with the next midpoint
            path.LineTo(nextMidPoint);
        }
        path.Close();

        // Render the path in the center of the screen
        using (SKPaint paint = new SKPaint())
        {
            paint.Style = SKPaintStyle.Stroke;
            paint.Color = SKColors.Blue;
            paint.StrokeWidth = 10;

            canvas.Translate(info.Width / 2, info.Height / 2);
            canvas.DrawPath(path, paint);
        }
    }
}

```

Here's the program running:



The Elliptical Arc

The elliptical arc is added to a path with a call to the `ArcTo` method that has two `SKPoint` parameters, or the `ArcTo` overload with separate X and Y coordinates:

```
public void ArcTo (SKPoint r, Single xAxisRotate, SKPathArcSize largeArc, SKPathDirection sweep, SKPoint xy)
public void ArcTo (Single rx, Single ry, Single xAxisRotate, SKPathArcSize largeArc, SKPathDirection sweep,
Single x, Single y)
```

The elliptical arc is consistent with the [elliptical arc](#) included in Scalable Vector Graphics (SVG) and the Universal Windows Platform [ArcSegment](#) class.

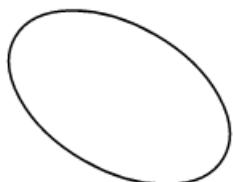
These `ArcTo` methods draw an arc between two points, which are the current point of the contour, and the last parameter to the `ArcTo` method (the `xy` parameter or the separate `x` and `y` parameters):

current point → • • ← specified point

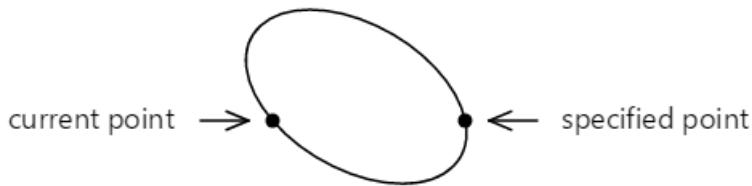
The first point parameter to the `ArcTo` method (`r`, or `rx` and `ry`) is not a point at all but instead specifies the horizontal and vertical radii of an ellipse;



The `xAxisRotate` parameter is the number of clockwise degrees to rotate this ellipse:

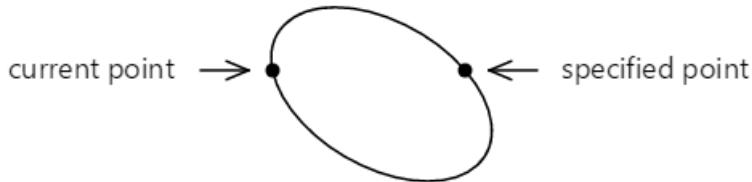


If this tilted ellipse is then positioned so that it touches the two points, the points are connected by two different arcs:



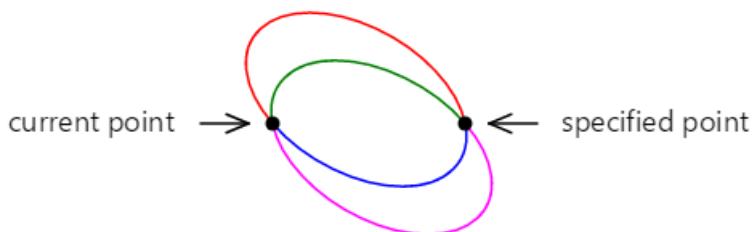
These two arcs can be distinguished in two ways: The top arc is larger than the bottom arc, and as the arc is drawn from left to right, the top arc is drawn in a clockwise direction while the bottom arc is drawn in a counter-clockwise direction.

It is also possible to fit the ellipse between the two points in another way:



Now there's a smaller arc on top that's drawn clockwise, and a larger arc on the bottom that's drawn counter-clockwise.

These two points can therefore be connected by an arc defined by the tilted ellipse in a total of four ways:



These four arcs are distinguished by the four combinations of the `SKPathArcSize` and `SKPathDirection` enumeration type arguments to the `ArcTo` method:

- red: `SKPathArcSize.Large` and `SKPathDirection.Clockwise`
- green: `SKPathArcSize.Small` and `SKPathDirection.Clockwise`
- blue: `SKPathArcSize.Small` and `SKPathDirection.CounterClockwise`
- magenta: `SKPathArcSize.Large` and `SKPathDirection.CounterClockwise`

If the tilted ellipse is not large enough to fit between the two points, then it is uniformly scaled until it is large enough. Only two unique arcs connect the two points in that case. These can be distinguished with the `SKPathDirection` parameter.

Although this approach to defining an arc sounds complex on first encounter, it is the only approach that allows defining an arc with a rotated ellipse, and it is often the easiest approach when you need to integrate arcs with other parts of the contour.

The **Elliptical Arc** page allows you to interactively set the two points, and the size and rotation of the ellipse.

The `EllipticalArcPage` class derives from `InteractivePage`, and the `PaintSurface` handler in the

`EllipticalArcPage.xaml.cs` code-behind file draws the four arcs:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    using (SKPath path = new SKPath())
    {
        int colorIndex = 0;
        SKPoint ellipseSize = new SKPoint((float)xRadiusSlider.Value,
                                           (float)yRadiusSlider.Value);
        float rotation = (float)rotationSlider.Value;

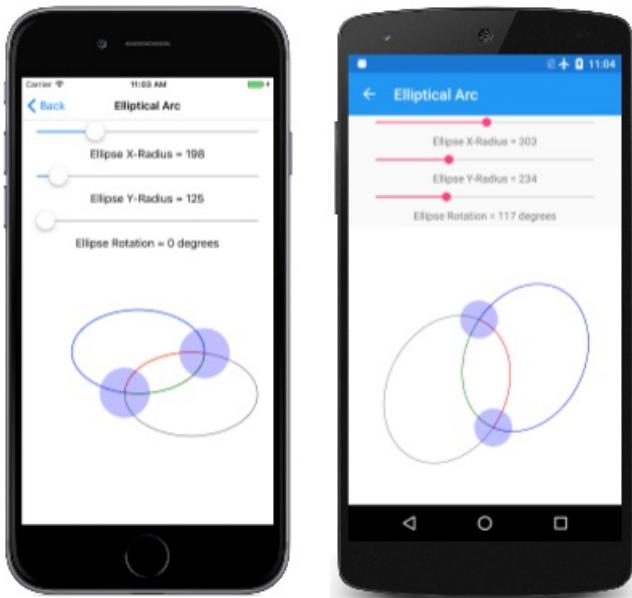
        foreach (SKPathArcSize arcSize in Enum.GetValues(typeof(SKPathArcSize)))
            foreach (SKPathDirection direction in Enum.GetValues(typeof(SKPathDirection)))
            {
                path.MoveTo(touchPoints[0].Center);
                path.ArcTo(ellipseSize, rotation,
                           arcSize, direction,
                           touchPoints[1].Center);

                strokePaint.Color = colors[colorIndex++];
                canvas.DrawPath(path, strokePaint);
                path.Reset();
            }
    }

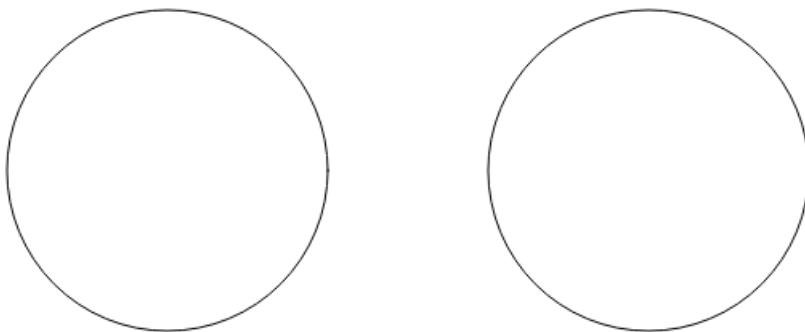
    foreach (TouchPoint touchPoint in touchPoints)
    {
        touchPoint.Paint(canvas);
    }
}

```

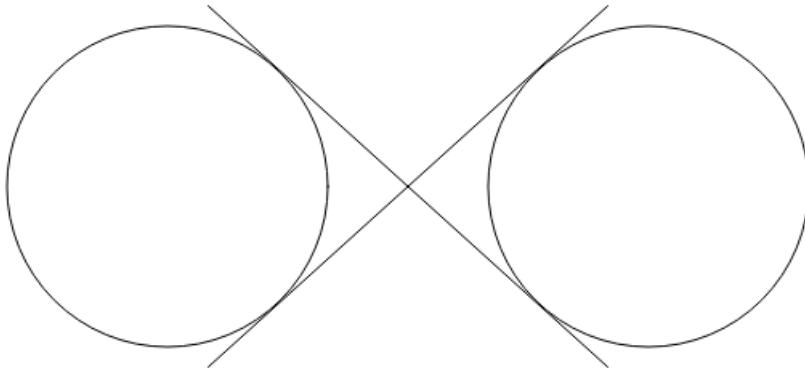
Here it is running:



The **Arc Infinity** page uses the elliptical arc to draw an infinity sign. The infinity sign is based on two circles with radii of 100 units separated by 100 units:

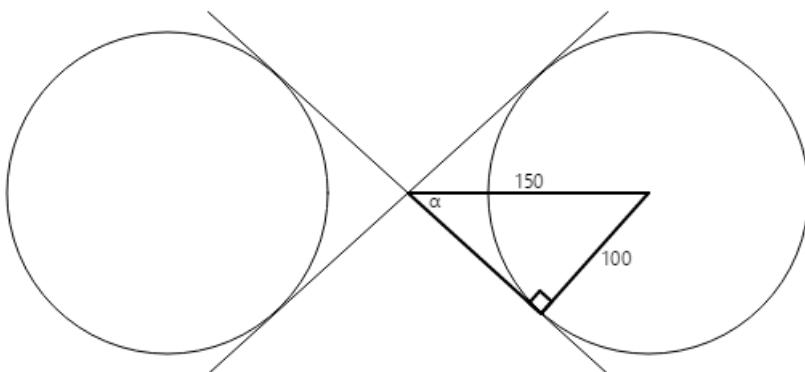


Two lines crossing each other are tangent to both circles:



The infinity sign is a combination of parts of these circles and the two lines. To use the elliptical arc to draw the infinity sign, the coordinates where the two lines are tangent to the circles must be determined.

Construct a right rectangle in one of the circles:



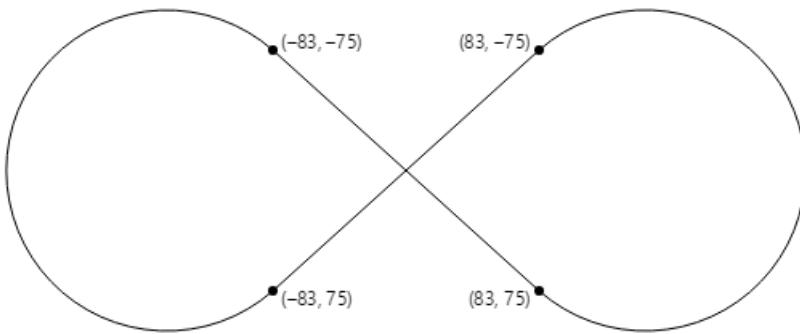
The radius of the circle is 100 units, and the hypotenuse of the triangle is 150 units, so the angle α is the arcsine (inverse sine) of 100 divided by 150, or 41.8 degrees. The length of the other side of the triangle is 150 times the cosine of 41.8 degrees, or 112, which can also be calculated by the Pythagorean theorem.

The coordinates of the tangent point can then be calculated using this information:

$$x = 112 \cdot \cos(41.8) = 83$$

$$y = 112 \cdot \sin(41.8) = 75$$

The four tangent points are all that's necessary to draw an infinity sign centered on the point (0, 0) with circle radii of 100:



The `PaintSurface` handler in the `ArcInfinityPage` class positions the infinity sign so that the $(0, 0)$ point is positioned in the center of the page, and scales the path to the screen size:

```
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    using (SKPath path = new SKPath())
    {
        path.LineTo(83, 75);
        path.ArcTo(100, 100, 0, SKPathArcSize.Large, SKPathDirection.CounterClockwise, 83, -75);
        path.LineTo(-83, 75);
        path.ArcTo(100, 100, 0, SKPathArcSize.Large, SKPathDirection.Clockwise, -83, -75);
        path.Close();

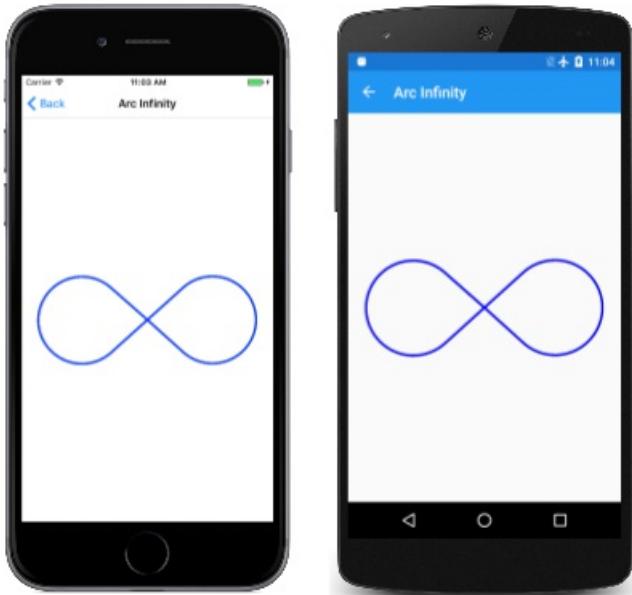
        // Use path.TightBounds for coordinates without control points
        SKRect pathBounds = path.Bounds;

        canvas.Translate(info.Width / 2, info.Height / 2);
        canvas.Scale(Math.Min(info.Width / pathBounds.Width,
                             info.Height / pathBounds.Height));

        using (SKPaint paint = new SKPaint())
        {
            paint.Style = SKPaintStyle.Stroke;
            paint.Color = SKColors.Blue;
            paint.StrokeWidth = 5;

            canvas.DrawPath(path, paint);
        }
    }
}
```

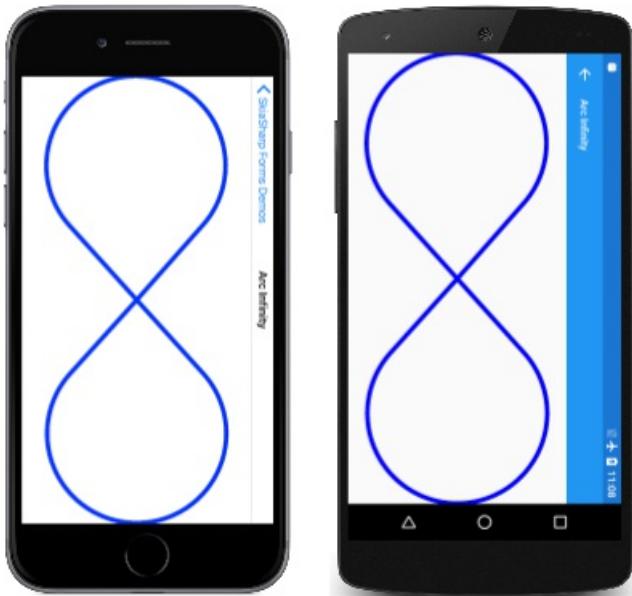
The code uses the `Bounds` property of `SKPath` to determine the dimensions of the infinity sine to scale it to the size of the canvas:



The result seems a little small, which suggests that the `Bounds` property of `SKPath` is reporting a size larger than the path.

Internally, Skia approximates the arc using multiple quadratic Bézier curves. These curves (as you'll see in the next section) contain control points that govern how the curve is drawn but are not part of the rendered curve. The `Bounds` property includes those control points.

To get a tighter fit, use the `TightBounds` property, which excludes the control points. Here's the program running in landscape mode, and using the `TightBounds` property to obtain the path bounds:



Although the connections between the arcs and straight lines are mathematically smooth, the change from arc to straight line might seem a little abrupt. A better infinity sign is presented in the next article on [Three Types of Bézier Curves](#).

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Three Types of Bézier Curves

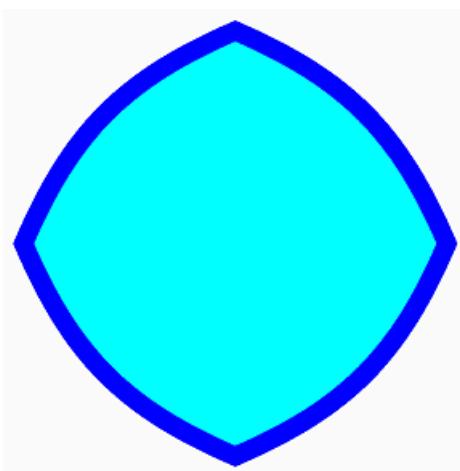
3/5/2021 • 18 minutes to read • [Edit Online](#)

 [Download the sample](#)

Explore how to use SkiaSharp to render cubic, quadratic, and conic Bézier curves

The Bézier curve is named after Pierre Bézier (1910 – 1999), a French engineer at the automotive company Renault, who used the curve for the computer-assisted design of car bodies.

Bézier curves are known for being well-suited to interactive design: They are well behaved — in other words, there aren't singularities that cause the curve to become infinite or unwieldy — and they are generally aesthetically pleasing:



Character outlines of computer-based fonts are usually defined with Bézier curves.

The Wikipedia article on [Bézier curve](#) contains some useful background information. The term *Bézier curve* actually refers to a family of similar curves. SkiaSharp supports three types of Bézier curves, called the *cubic*, the *quadratic*, and the *conic*. The conic is also known as the *rational quadratic*.

The Cubic Bézier Curve

The cubic is the type of Bézier curve that most developers think of when the subject of Bézier curves comes up.

You can add a cubic Bézier curve to an `SKPath` object using the `CubicTo` method with three `SKPoint` parameters, or the `CubicTo` overload with separate `x` and `y` parameters:

```
public void CubicTo (SKPoint point1, SKPoint point2, SKPoint point3)  
  
public void CubicTo (Single x1, Single y1, Single x2, Single y2, Single x3, Single y3)
```

The curve begins at the current point of the contour. The complete cubic Bezier curve is defined by four points:

- start point: current point in the contour, or (0, 0) if `MoveTo` has not been called
- first control point: `point1` in the `CubicTo` call
- second control point: `point2` in the `CubicTo` call
- end point: `point3` in the `CubicTo` call

The resultant curve begins at the start point and ends at the end point. The curve generally does not pass

through the two control points; instead the control points function much like magnets to pull the curve towards them.

The best way to get a feel for the cubic Bézier curve is by experimentation. This is the purpose of the **Bezier Curve** page, which derives from `InteractivePage`. The `BezierCurvePage.xaml` file instantiates the `SKCanvasView` and a `TouchEffect`. The `BezierCurvePage.xaml.cs` code-behind file creates four `TouchPoint` objects in its constructor. The `PaintSurface` event handler creates an `SKPath` to render a Bézier curve based on the four `TouchPoint` objects, and also draws dotted tangent lines from the control points to the end points:

```
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    // Draw path with cubic Bezier curve
    using (SKPath path = new SKPath())
    {
        path.MoveTo(touchPoints[0].Center);
        path.CubicTo(touchPoints[1].Center,
                     touchPoints[2].Center,
                     touchPoints[3].Center);

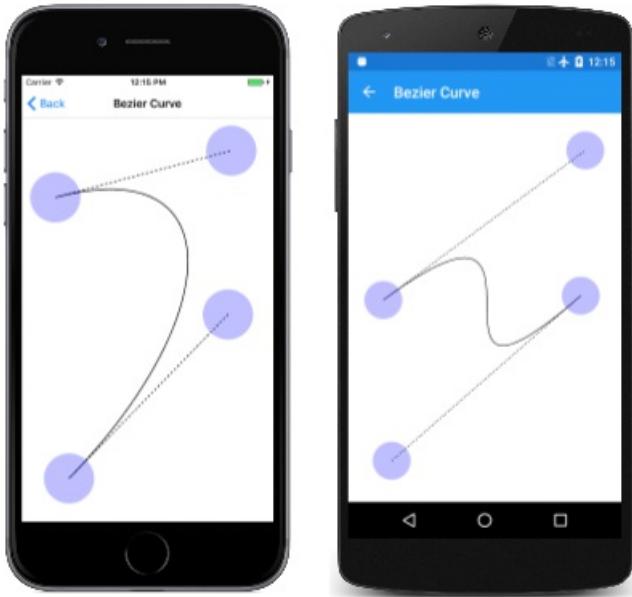
        canvas.DrawPath(path, strokePaint);
    }

    // Draw tangent lines
    canvas.DrawLine(touchPoints[0].Center.X,
                   touchPoints[0].Center.Y,
                   touchPoints[1].Center.X,
                   touchPoints[1].Center.Y, dottedStrokePaint);

    canvas.DrawLine(touchPoints[2].Center.X,
                   touchPoints[2].Center.Y,
                   touchPoints[3].Center.X,
                   touchPoints[3].Center.Y, dottedStrokePaint);

    foreach (TouchPoint touchPoint in touchPoints)
    {
        touchPoint.Paint(canvas);
    }
}
```

Here it is running:



Mathematically, the curve is a cubic polynomial. The curve intersects a straight line at three points at most. At the start point, the curve is always tangent to, and in the same direction as, a straight line from the start point to the first control point. At the end point, the curve is always tangent to, and in the same direction as, a straight line from the second control point to the end point.

The cubic Bézier curve is always bounded by a convex quadrilateral connecting the four points. This is called a *convex hull*. If the control points lie on the straight line between the start and end point, then the Bézier curve renders as a straight line. But the curve can also cross itself, as the third screenshot demonstrates.

A path contour can contain multiple connected cubic Bézier curves, but the connection between two cubic Bézier curves will be smooth only if the following three points are colinear (that is, lie on a straight line):

- the second control point of the first curve
- the end point of the first curve, which is also the start point of the second curve
- the first control point of the second curve

In the next article on [SVG Path Data](#), you'll discover a facility to ease the definition of smooth connected Bézier curves.

It is sometimes useful to know the underlying parametric equations that render a cubic Bézier curve. For t ranging from 0 to 1, the parametric equations are as follows:

$$x(t) = (1 - t)^3 x_0 + 3t(1 - t)^2 x_1 + 3t^2(1 - t)x_2 + t^3 x_3$$

$$y(t) = (1 - t)^3 y_0 + 3t(1 - t)^2 y_1 + 3t^2(1 - t)y_2 + t^3 y_3$$

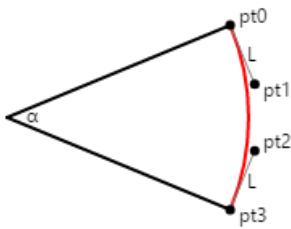
The highest exponent of 3 confirms that these are cubic polynomials. It is easy to verify that when t equals 0, the point is (x_0, y_0) , which is the start point, and when t equals 1, the point is (x_3, y_3) , which is the end point. Near the start point (for low values of t), the first control point (x_1, y_1) has a strong effect, and near the end point (high values of t) the second control point (x_2, y_2) has a strong effect.

Bezier Curve Approximation to Circular Arcs

It is sometimes convenient to use a Bézier curve to render a circular arc. A cubic Bézier curve can approximate a circular arc very well up to a quarter circle, so four connected Bézier curves can define a whole circle. This approximation is discussed in two articles published over 25 years ago:

Tor Dokken, et al, "Good Approximation of Circles by Curvature-Continuous Bézier curves," *Computer Aided Geometric Design* 7 (1990), 33-41.

The following diagram shows four points labeled `pto`, `pt1`, `pt2`, and `pt3` defining a Bézier curve (shown in red) that approximates a circular arc:



The lines from the start and end points to the control points are tangent to the circle and to the Bézier curve, and they have a length of L . The first article cited above indicates that the Bézier curve best approximates a circular arc when that length L is calculated like this:

$$L = 4 \times \tan(a / 4) / 3$$

The illustration shows an angle of 45 degrees, so L equals 0.265. In code, that value would be multiplied by the desired radius of the circle.

The **Bezier Circular Arc** page allows you to experiment with defining a Bézier curve to approximate a circular arc for angles ranging up to 180 degrees. The `BezierCircularArcPage.xaml` file instantiates the `skcanvasView` and a `Slider` for selecting the angle. The `PaintSurface` event handler in the `BezierCircularArcPage.xaml.cs` code-behind file uses a transform to set the point (0, 0) to the center of the canvas. It draws a circle centered on that point for comparison, and then calculates the two control points for the Bézier curve:

```
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    // Translate to center
    canvas.Translate(info.Width / 2, info.Height / 2);

    // Draw the circle
    float radius = Math.Min(info.Width, info.Height) / 3;
    canvas.DrawCircle(0, 0, radius, blackStroke);

    // Get the value of the Slider
    float angle = (float)angleSlider.Value;

    // Calculate length of control point line
    float length = radius * 4 * (float)Math.Tan(Math.PI * angle / 180 / 4) / 3;

    // Calculate sin and cosine for half that angle
    float sin = (float)Math.Sin(Math.PI * angle / 180 / 2);
    float cos = (float)Math.Cos(Math.PI * angle / 180 / 2);

    // Find the end points
    SKPoint point0 = new SKPoint(-radius * sin, radius * cos);
    SKPoint point3 = new SKPoint(radius * sin, radius * cos);

    // Find the control points
    SKPoint point0Normalized = Normalize(point0);
    SKPoint point1 = point0 + new SKPoint(length * point0Normalized.Y,
                                         -length * point0Normalized.X);
    SKPoint point2 = point3 - new SKPoint(length * point0Normalized.Y,
                                         -length * point0Normalized.X);
    SKPoint point4 = point3 + new SKPoint(length * point0Normalized.Y,
                                         -length * point0Normalized.X);

    // Create the Bezier curve
    canvas.DrawBezier(point0, point1, point2, point3, blackStroke);
}
```

```

SKPoint point3Normalized = Normalize(point3);
SKPoint point2 = point3 + new SKPoint(-length * point3Normalized.Y,
                                         length * point3Normalized.X);

// Draw the points
canvas.DrawCircle(point0.X, point0.Y, 10, blackFill);
canvas.DrawCircle(point1.X, point1.Y, 10, blackFill);
canvas.DrawCircle(point2.X, point2.Y, 10, blackFill);
canvas.DrawCircle(point3.X, point3.Y, 10, blackFill);

// Draw the tangent lines
canvas.DrawLine(point0.X, point0.Y, point1.X, point1.Y, dottedStroke);
canvas.DrawLine(point3.X, point3.Y, point2.X, point2.Y, dottedStroke);

// Draw the Bezier curve
using (SKPath path = new SKPath())
{
    path.MoveTo(point0);
    path.CubicTo(point1, point2, point3);
    canvas.DrawPath(path, redStroke);
}
}

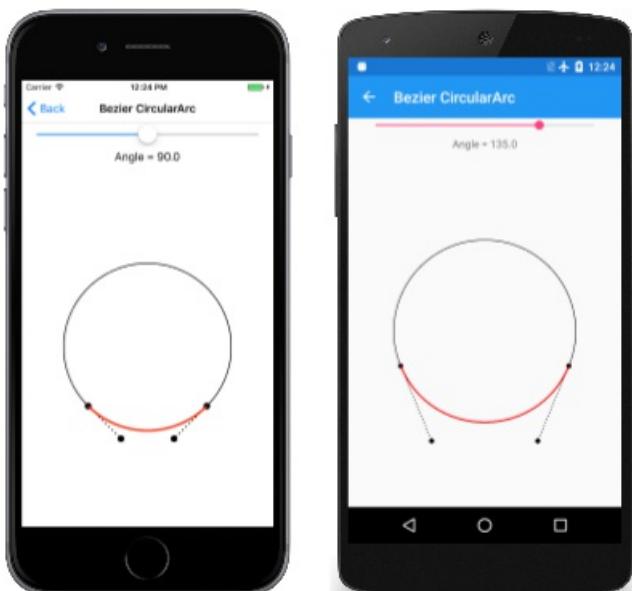
// Vector methods
SKPoint Normalize(SKPoint v)
{
    float magnitude = Magnitude(v);
    return new SKPoint(v.X / magnitude, v.Y / magnitude);
}

float Magnitude(SKPoint v)
{
    return (float)Math.Sqrt(v.X * v.X + v.Y * v.Y);
}

```

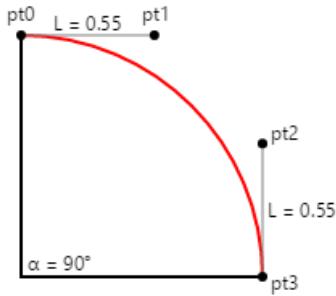
The start and end points (`point0` and `point3`) are calculated based on the normal parametric equations for the circle. Because the circle is centered at $(0, 0)$, these points can also be treated as radial vectors from the center of the circle to the circumference. The control points are on lines that are tangent to the circle, so they are at right angles to these radial vectors. A vector at a right angle to another is simply the original vector with the X and Y coordinates swapped and one of them made negative.

Here's the program running with different angles:



Look closely at the third screenshot, and you'll see that the Bézier curve notably deviates from a semicircle when the angle is 180 degrees, but the iOS screen shows that it seems to fit a quarter-circle just fine when the angle is 90 degrees.

Calculating the coordinates of the two control points is quite easy when the quarter circle is oriented like this:



If the radius of the circle is 100, then L is 55, and that's an easy number to remember.

The [Squaring the Circle](#) page animates a figure between a circle and a square. The circle is approximated by four Bézier curves whose coordinates are shown in the first column of this array definition in the [SquaringTheCirclePage](#) class:

```
public class SquaringTheCirclePage : ContentPage
{
    SKPoint[,] points =
    {
        { new SKPoint(  0,  100), new SKPoint(      0,     125), new SKPoint() },
        { new SKPoint(  55,  100), new SKPoint( 62.5f,  62.5f), new SKPoint() },
        { new SKPoint( 100,   55), new SKPoint( 62.5f,  62.5f), new SKPoint() },
        { new SKPoint( 100,    0), new SKPoint( 125,      0), new SKPoint() },
        { new SKPoint( 100,  -55), new SKPoint( 62.5f, -62.5f), new SKPoint() },
        { new SKPoint(  55, -100), new SKPoint( 62.5f, -62.5f), new SKPoint() },
        { new SKPoint(   0, -100), new SKPoint(      0,    -125), new SKPoint() },
        { new SKPoint( -55, -100), new SKPoint(-62.5f, -62.5f), new SKPoint() },
        { new SKPoint(-100, -55), new SKPoint(-62.5f, -62.5f), new SKPoint() },
        { new SKPoint(-100,   0), new SKPoint( -125,      0), new SKPoint() },
        { new SKPoint(-100,  55), new SKPoint(-62.5f,  62.5f), new SKPoint() },
        { new SKPoint( -55,  100), new SKPoint(-62.5f,  62.5f), new SKPoint() },
        { new SKPoint(   0,  100), new SKPoint(      0,     125), new SKPoint() }
    };
    ...
}
```

The second column contains the coordinates of four Bézier curves that define a square whose area is approximately the same as the area of the circle. (Drawing a square with the *exact* area as a given circle is the classic unsolvable geometric problem of [squaring the circle](#).) For rendering a square with Bézier curves, the two control points for each curve are the same, and they are colinear with the start and end points, so the Bézier curve is rendered as a straight line.

The third column of the array is for interpolated values for an animation. The page sets a timer for 16 milliseconds, and the [PaintSurface](#) handler is called at that rate:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    canvas.Translate(info.Width / 2, info.Height / 2);
    canvas.Scale(Math.Min(info.Width / 300, info.Height / 300));

    // Interpolate
    TimeSpan timeSpan = new TimeSpan(DateTime.Now.Ticks);
    float t = (float)(timeSpan.TotalSeconds % 3 / 3);    // 0 to 1 every 3 seconds
    t = (1 + (float)Math.Sin(2 * Math.PI * t)) / 2;      // 0 to 1 to 0 sinusoidally

    for (int i = 0; i < 13; i++)
    {
        points[i, 2] = new SKPoint(
            (1 - t) * points[i, 0].X + t * points[i, 1].X,
            (1 - t) * points[i, 0].Y + t * points[i, 1].Y);
    }

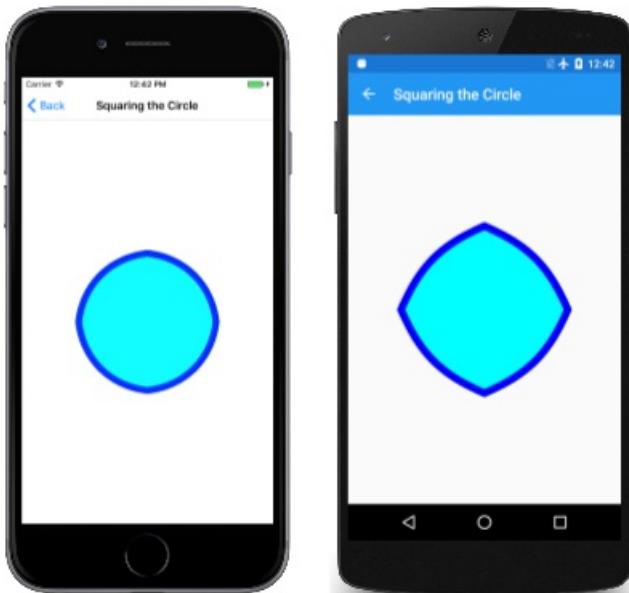
    // Create the path and draw it
    using (SKPath path = new SKPath())
    {
        path.MoveTo(points[0, 2]);

        for (int i = 1; i < 13; i += 3)
        {
            path.CubicTo(points[i, 2], points[i + 1, 2], points[i + 2, 2]);
        }
        path.Close();

        canvas.DrawPath(path, cyanFill);
        canvas.DrawPath(path, blueStroke);
    }
}

```

The points are interpolated based on a sinusoidally oscillating value of t . The interpolated points are then used to construct a series of four connected Bézier curves. Here's the animation running:



Such an animation would be impossible without curves that are algorithmically flexible enough to be rendered as both circular arcs and straight lines.

The **Bezier Infinity** page also takes advantage of the ability of a Bézier curve to approximate a circular arc. Here's the `PaintSurface` handler from the `BezierInfinityPage` class:

```
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    using (SKPath path = new SKPath())
    {
        path.MoveTo(0, 0); // Center
        path.CubicTo( 50, -50, 95, -100, 150, -100); // To top of right loop
        path.CubicTo( 205, -100, 250, -55, 250, 0); // To far right of right loop
        path.CubicTo( 250, 55, 205, 100, 150, 100); // To bottom of right loop
        path.CubicTo( 95, 100, 50, 50, 0, 0); // Back to center
        path.CubicTo( -50, -50, -95, -100, -150, -100); // To top of left loop
        path.CubicTo(-205, -100, -250, -55, -250, 0); // To far left of left loop
        path.CubicTo(-250, 55, -205, 100, -150, 100); // To bottom of left loop
        path.CubicTo( -95, 100, -50, 50, 0, 0); // Back to center
        path.Close();

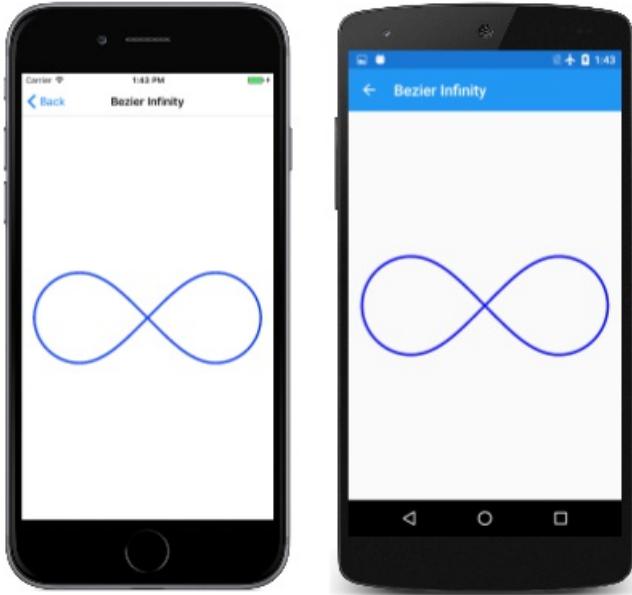
        SKRect pathBounds = path.Bounds;
        canvas.Translate(info.Width / 2, info.Height / 2);
        canvas.Scale(0.9f * Math.Min(info.Width / pathBounds.Width,
                                      info.Height / pathBounds.Height));

        using (SKPaint paint = new SKPaint())
        {
            paint.Style = SKPaintStyle.Stroke;
            paint.Color = SKColors.Blue;
            paint.StrokeWidth = 5;

            canvas.DrawPath(path, paint);
        }
    }
}
```

It might be a good exercise to plot these coordinates on graph paper to see how they are related. The infinity sign is centered around the point (0, 0), and the two loops have centers of (-150, 0) and (150, 0) and radii of 100. In the series of `CubicTo` commands, you can see X coordinates of control points taking on values of -95 and -205 (those values are -150 plus and minus 55), 205 and 95 (150 plus and minus 55), as well as 250 and -250 for the right and left sides. The only exception is when the infinity sign crosses itself in the center. In that case, control points have coordinates with a combination of 50 and -50 to straighten out the curve near the center.

Here's the infinity sign:



It is somewhat smoother towards the center than the infinity sign rendered by the [Arc Infinity](#) page from the [Three Ways to Draw an Arc](#) article.

The Quadratic Bézier Curve

The quadratic Bézier curve has only one control point, and the curve is defined by just three points: the start point, the control point, and the end point. The parametric equations are very similar to the cubic Bézier curve, except that the highest exponent is 2, so the curve is a quadratic polynomial:

$$x(t) = (1 - t)^2 x_1 + 2t(1 - t)x_2 + t^2 x_3$$

$$y(t) = (1 - t)^2 y_1 + 2t(1 - t)y_2 + t^2 y_3$$

To add a quadratic Bézier curve to a path, use the `QuadTo` method or the `QuadTo` overload with separate `x` and `y` coordinates:

```
public void QuadTo (SKPoint point1, SKPoint point2)  
  
public void QuadTo (Single x1, Single y1, Single x2, Single y2)
```

The methods add a curve from the current position to `point2` with `point1` as the control point.

You can experiment with quadratic Bézier curves with the [Quadratic Curve](#) page, which is very similar to the [Bezier Curve](#) page except it has only three touch points. Here's the `PaintSurface` handler in the [QuadraticCurve.xaml.cs](#) code-behind file:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    // Draw path with quadratic Bezier
    using (SKPath path = new SKPath())
    {
        path.MoveTo(touchPoints[0].Center);
        path.QuadTo(touchPoints[1].Center,
                    touchPoints[2].Center);

        canvas.DrawPath(path, strokePaint);
    }

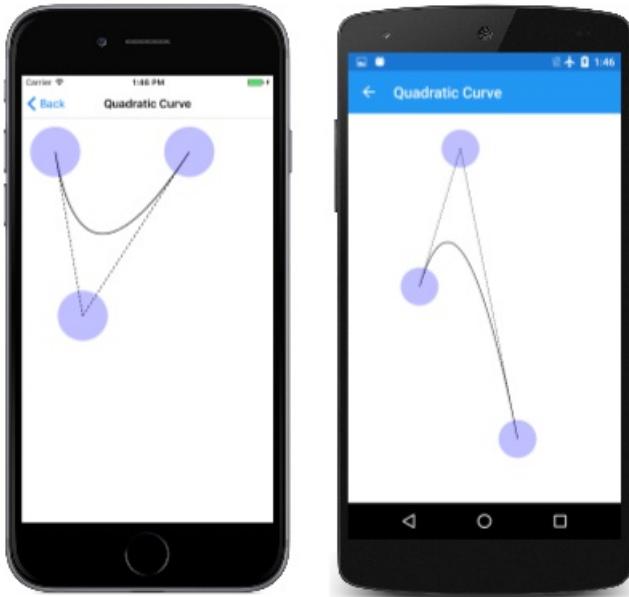
    // Draw tangent lines
    canvas.DrawLine(touchPoints[0].Center.X,
                    touchPoints[0].Center.Y,
                    touchPoints[1].Center.X,
                    touchPoints[1].Center.Y, dottedStrokePaint);

    canvas.DrawLine(touchPoints[1].Center.X,
                    touchPoints[1].Center.Y,
                    touchPoints[2].Center.X,
                    touchPoints[2].Center.Y, dottedStrokePaint);

    foreach (TouchPoint touchPoint in touchPoints)
    {
        touchPoint.Paint(canvas);
    }
}

```

And here it is running:



The dotted lines are tangent to the curve at the start point and end point, and meet at the control point.

The quadratic Bézier is good if you need a curve of a general shape, but you prefer the convenience of just one control point rather than two. The quadratic Bézier renders more efficiently than any other curve, which is why it's used internally in Skia to render elliptical arcs.

However, the shape of a quadratic Bézier curve is not elliptical, which is why multiple quadratic Béziers are

required to approximate an elliptical arc. The quadratic Bézier is instead a segment of a parabola.

The Conic Bézier Curve

The conic Bézier curve — also known as the rational quadratic Bézier curve — is a relatively recent addition to the family of Bézier curves. Like the quadratic Bézier curve, the rational quadratic Bézier curve involves a start point, an end point, and one control point. But the rational quadratic Bézier curve also requires a *weight* value. It's called a *rational* quadratic because the parametric formulas involve ratios.

The parametric equations for X and Y are ratios that share the same denominator. Here is the equation for the denominator for t ranging from 0 to 1 and a weight value of w :

$$d(t) = (1 - t)^2 + 2wt(1 - t) + t^2$$

In theory, a rational quadratic can involve three separate weight values, one for each of the three terms, but these can be simplified to just one weight value on the middle term.

The parametric equations for the X and Y coordinates are similar to the parametric equations for the quadratic Bézier except that the middle term also includes the weight value, and the expression is divided by the denominator:

$$x(t) = ((1 - t)^2 x_1 + 2wt(1 - t)x_1 + t^2 x_1) \div d(t)$$

$$y(t) = ((1 - t)^2 y_1 + 2wt(1 - t)y_1 + t^2 y_1) \div d(t)$$

Rational quadratic Bézier curves are also called *conics* because they can exactly represent segments of any conic section — hyperbolas, parabolas, ellipses, and circles.

To add a rational quadratic Bézier curve to a path, use the `ConicTo` method or the `ConicTo` overload with separate `x` and `y` coordinates:

```
public void ConicTo (SKPoint point1, SKPoint point2, Single weight)

public void ConicTo (Single x1, Single y1, Single x2, Single y2, Single weight)
```

Notice the final `weight` parameter.

The [Conic Curve](#) page allows you to experiment with these curves. The `ConicCurvePage` class derives from `InteractivePage`. The `ConicCurvePage.xaml` file instantiates a `slider` to select a weight value between -2 and 2. The `ConicCurvePage.xaml.cs` code-behind file creates three `TouchPoint` objects, and the `PaintSurface` handler simply renders the resultant curve with the tangent lines to the control points:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    // Draw path with conic curve
    using (SKPath path = new SKPath())
    {
        path.MoveTo(touchPoints[0].Center);
        path.ConicTo(touchPoints[1].Center,
                     touchPoints[2].Center,
                     (float)weightSlider.Value);

        canvas.DrawPath(path, strokePaint);
    }

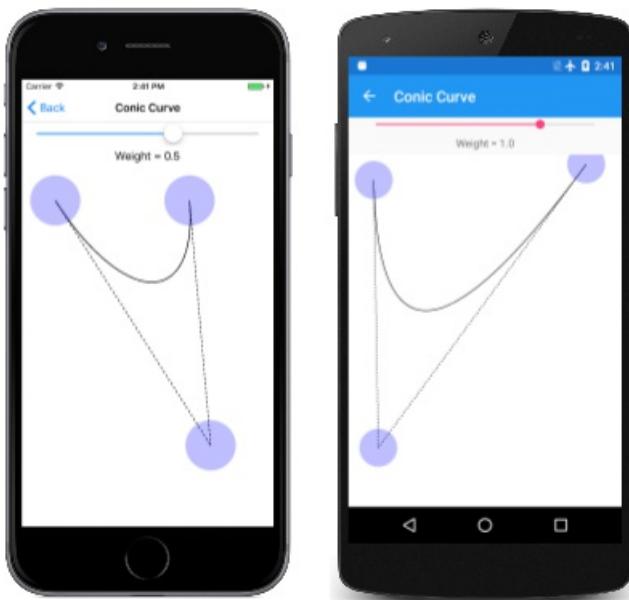
    // Draw tangent lines
    canvas.DrawLine(touchPoints[0].Center.X,
                    touchPoints[0].Center.Y,
                    touchPoints[1].Center.X,
                    touchPoints[1].Center.Y, dottedStrokePaint);

    canvas.DrawLine(touchPoints[1].Center.X,
                    touchPoints[1].Center.Y,
                    touchPoints[2].Center.X,
                    touchPoints[2].Center.Y, dottedStrokePaint);

    foreach (TouchPoint touchPoint in touchPoints)
    {
        touchPoint.Paint(canvas);
    }
}

```

Here it is running:

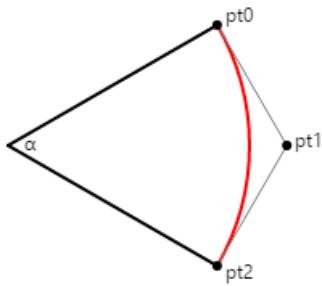


As you can see, the control point seems to pull the curve towards it more when the weight is higher. When the weight is zero, the curve becomes a straight line from the start point to the end point.

In theory, negative weights are allowed, and cause the curve to bend *away* from the control point. However, weights of -1 or below cause the denominator in the parametric equations to become negative for particular values of t . Probably for this reason, negative weights are ignored in the `ConicTo` methods. The **Conic Curve**

program lets you set negative weights, but as you can see by experimenting, negative weights have the same effect as a weight of zero, and cause a straight line to be rendered.

It is very easy to derive the control point and weight to use the `ConicTo` method to draw a circular arc up to (but not including) a semicircle. In the following diagram, tangent lines from the start and end points meet at the control point.



You can use trigonometry to determine the distance of the control point from the circle's center: It is the radius of the circle divided by the cosine of half the angle α . To draw a circular arc between the start and end points, set the weight to that same cosine of half the angle. Notice that if the angle is 180 degrees, then the tangent lines never meet and the weight is zero. But for angles less than 180 degrees, the math works fine.

The **Conic Circular Arc** page demonstrates this. The `ConicCircularArc.xaml` file instantiates a `Slider` for selecting the angle. The `PaintSurface` handler in the `ConicCircularArc.xaml.cs` code-behind file calculates the control point and the weight:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    // Translate to center
    canvas.Translate(info.Width / 2, info.Height / 2);

    // Draw the circle
    float radius = Math.Min(info.Width, info.Height) / 4;
    canvas.DrawCircle(0, 0, radius, blackStroke);

    // Get the value of the Slider
    float angle = (float)angleSlider.Value;

    // Calculate sin and cosine for half that angle
    float sin = (float)Math.Sin(Math.PI * angle / 180 / 2);
    float cos = (float)Math.Cos(Math.PI * angle / 180 / 2);

    // Find the points and weight
    SKPoint point0 = new SKPoint(-radius * sin, radius * cos);
    SKPoint point1 = new SKPoint(0, radius / cos);
    SKPoint point2 = new SKPoint(radius * sin, radius * cos);
    float weight = cos;

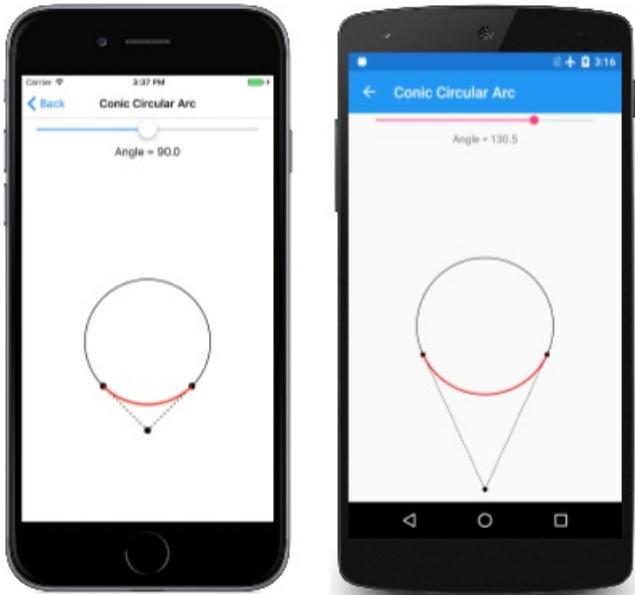
    // Draw the points
    canvas.DrawCircle(point0.X, point0.Y, 10, blackFill);
    canvas.DrawCircle(point1.X, point1.Y, 10, blackFill);
    canvas.DrawCircle(point2.X, point2.Y, 10, blackFill);

    // Draw the tangent lines
    canvas.DrawLine(point0.X, point0.Y, point1.X, point1.Y, dottedStroke);
    canvas.DrawLine(point2.X, point2.Y, point1.X, point1.Y, dottedStroke);

    // Draw the conic
    using (SKPath path = new SKPath())
    {
        path.MoveTo(point0);
        path.ConicTo(point1, point2, weight);
        canvas.DrawPath(path, redStroke);
    }
}

```

As you can see, there is no visual difference between the `ConicTo` path shown in red and the underlying circle displayed for reference:



But set the angle to 180 degrees, and the mathematics fail.

It is unfortunate in this case that `ConicTo` does not support negative weights, because in theory (based on the parametric equations), the circle can be completed with another call to `ConicTo` with the same points but a negative value of the weight. This would allow creating a whole circle with just two `ConicTo` curves based on any angle between (but not including) zero degrees and 180 degrees.

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

SVG Path Data in SkiaSharp

3/5/2021 • 14 minutes to read • [Edit Online](#)

 [Download the sample](#)

Define paths using text strings in the Scalable Vector Graphics format

The `SKPath` class supports the definition of entire path objects from text strings in a format established by the Scalable Vector Graphics (SVG) specification. You'll see later in this article how you can represent an entire path such as this one in a text string:



SVG is an XML-based graphics programming language for web pages. Because SVG must allow paths to be defined in markup rather than a series of function calls, the SVG standard includes an extremely concise way of specifying an entire graphics path as a text string.

Within SkiaSharp, this format is referred to as "SVG path-data." The format is also supported in Windows XAML-based programming environments, including the Windows Presentation Foundation and the Universal Windows Platform, where it is known as the [Path Markup Syntax](#) or the [Move and draw commands syntax](#). It can also serve as an exchange format for vector graphics images, particularly in text-based files such as XML.

The `SKPath` class defines two methods with the words `SvgPathData` in their names:

```
public static SKPath ParseSvgPathData(string svgPath)  
public string ToSvgPathData()
```

The static `ParseSvgPathData` method converts a string to an `SKPath` object, while `ToSvgPathData` converts an `SKPath` object to a string.

Here's an SVG string for a five-pointed star centered on the point (0, 0) with a radius of 100:

```
"M 0 -100 L 58.8 90.9, -95.1 -30.9, 95.1 -30.9, -58.8 80.9 Z"
```

The letters are commands that build an `SKPath` object: `M` indicates a `MoveTo` call, `L` is `LineTo`, and `Z` is `Close` to close a contour. Each number pair provides an X and Y coordinate of a point. Notice that the `L` command is followed by multiple points separated by commas. In a series of coordinates and points, commas and whitespace are treated identically. Some programmers prefer to put commas between the X and Y coordinates rather than between the points, but commas or spaces are only required to avoid ambiguity. This is perfectly legal:

```
"M0-100L58.8 90.9-95.1-30.9 95.1-30.9-58.8 80.9Z"
```

The syntax of SVG path data is formally documented in [Section 8.3 of the SVG specification](#). Here is a summary:

MoveTo

```
M x y
```

This begins a new contour in the path by setting the current position. Path data should always begin with an `M` command.

LineTo

```
L x y ...
```

This command adds a straight line (or lines) to the path and sets the new current position to the end of the last line. You can follow the `L` command with multiple pairs of *x* and *y* coordinates.

Horizontal LineTo

```
H x ...
```

This command adds a horizontal line to the path and sets the new current position to the end of the line. You can follow the `H` command with multiple *x* coordinates, but it doesn't make much sense.

Vertical Line

```
V y ...
```

This command adds a vertical line to the path and sets the new current position to the end of the line.

Close

```
Z
```

The `C` command closes the contour by adding a straight line from the current position to the beginning of the contour.

ArcTo

The command to add an elliptical arc to the contour is by far the most complex command in the entire SVG path-data specification. It is the only command in which numbers can represent something other than coordinate values:

```
A rx ry rotation-angle large-arc-flag sweep-flag x y ...
```

The *rx* and *ry* parameters are the horizontal and vertical radii of the ellipse. The *rotation-angle* is clockwise in degrees.

Set the *large-arc-flag* to 1 for the large arc or to 0 for the small arc.

Set the *sweep-flag* to 1 for clockwise and to 0 for counter-clockwise.

The arc is drawn to the point (x, y) , which becomes the new current position.

CubicTo

```
C x1 y1 x2 y2 x3 y3 ...
```

This command adds a cubic Bézier curve from the current position to (x_3, y_3) , which becomes the new current position. The points (x_1, y_1) and (x_2, y_2) are control points.

Multiple Bézier curves can be specified by a single `C` command. The number of points must be a multiple of 3.

There is also a "smooth" Bézier curve command:

```
S x2 y2 x3 y3 ...
```

This command should follow a regular Bézier command (although that's not strictly required). The smooth Bézier command calculates the first control point so that it is a reflection of the second control point of the previous Bézier around their mutual point. These three points are therefore colinear, and the connection between the two Bézier curves is smooth.

QuadTo

```
Q x1 y1 x2 y2 ...
```

For quadratic Bézier curves, the number of points must be a multiple of 2. The control point is (x_1, y_1) and the end point (and new current position) is (x_2, y_2) .

There is also a smooth quadratic curve command:

```
T x2 y2 ...
```

The control point is calculated based on the control point of the previous quadratic curve.

All these commands are also available in "relative" versions, where the coordinate points are relative to the current position. These relative commands begin with lower-case letters, for example `c` rather than `C` for the relative version of the cubic Bézier command.

This is the extent of the SVG path-data definition. There is no facility for repeating groups of commands or for performing any type of calculation. Commands for `conicTo` or the other types of arc specifications are not available.

The static `SKPath.ParseSvgPathData` method expects a valid string of SVG commands. If any syntax error is detected, the method returns `null`. That is the only error indication.

The `ToSvgPathData` method is handy for obtaining SVG path data from an existing `SKPath` object to transfer to another program, or to store in a text-based file format such as XML. (The `ToSvgPathData` method is not demonstrated in sample code in this article.) Do *not* expect `ToSvgPathData` to return a string corresponding exactly to the method calls that created the path. In particular, you'll discover that arcs are converted to multiple `QuadTo` commands, and that's how they appear in the path data returned from `ToSvgPathData`.

The **Path Data Hello** page spells out the word "HELLO" using SVG path data. Both the `SKPath` and `SKPaint`

objects are defined as fields in the [PathDataHelloPage](#) class:

```
public class PathDataHelloPage : ContentPage
{
    SKPath helloPath = SKPath.ParseSvgPathData(
        "M 0 0 L 0 100 M 0 50 L 50 50 M 50 0 L 50 100" +           // H
        "M 125 0 C 60 -10, 60 60, 125 50, 60 40, 60 110, 125 100" +   // E
        "M 150 0 L 150 100, 200 100" +                                // L
        "M 225 0 L 225 100, 275 100" +                                // L
        "M 300 50 A 25 50 0 1 0 300 49.9 Z");                      // O

    SKPaint paint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Blue,
        StrokeWidth = 10,
        StrokeCap = SKStrokeCap.Round,
        StrokeJoin = SKStrokeJoin.Round
    };
    ...
}
```

The path defining the text string begins at the upper-left corner at the point(0, 0). Each letter is 50 units wide and 100 units tall, and letters are separated by another 25 units, which means that the entire path is 350 units wide.

The 'H' of "Hello" is composed of three one-line contours, while the 'E' is two connected cubic Bézier curves. Notice that the `C` command is followed by six points, and two of the control points have Y coordinates of `-10` and `110`, which puts them outside the range of the Y coordinates of the other letters. The 'L' is two connected lines, while the 'O' is an ellipse that is rendered with an `A` command.

Notice that the `M` command that begins the last contour sets the position to the point `(350, 50)`, which is the vertical center of the left side of the 'O'. As indicated by the first numbers following the `A` command, the ellipse has a horizontal radius of 25 and a vertical radius of 50. The end point is indicated by the last pair of numbers in the `A` command, which represents the point `(300, 49.9)`. That's deliberately just slightly different from the start point. If the endpoint is set equal to the start point, the arc will not be rendered. To draw a complete ellipse, you must set the endpoint close to (but not equal to) the start point, or you must use two or more `A` commands, each for part of the complete ellipse.

You might want to add the following statement to the page's constructor, and then set a breakpoint to examine the resultant string:

```
string str = helloPath.ToSvgPathData();
```

You'll discover that the arc has been replaced with a long series of `Q` commands for a piecemeal approximation of the arc using quadratic Bézier curves.

The `PaintSurface` handler obtains the tight bounds of the path, which does not include the control points for the 'E' and 'O' curves. The three transforms move the center of the path to the point `(0, 0)`, scale the path to the size of the canvas (but also taking the stroke width into account), and then move the center of the path to the center of the canvas:

```

public class PathDataHelloPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        SKRect bounds;
        helloPath.GetTightBounds(out bounds);

        canvas.Translate(info.Width / 2, info.Height / 2);

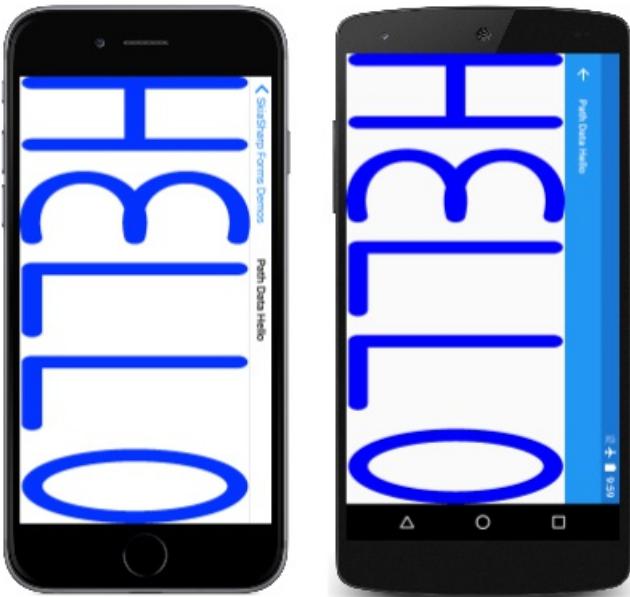
        canvas.Scale(info.Width / (bounds.Width + paint.StrokeWidth),
                    info.Height / (bounds.Height + paint.StrokeWidth));

        canvas.Translate(-bounds.MidX, -bounds.MidY);

        canvas.DrawPath(helloPath, paint);
    }
}

```

The path fills the canvas, which looks more reasonable when viewed in landscape mode:



The **Path Data Cat** page is similar. The path and paint objects are both defined as fields in the [PathDataCatPage](#) class:

```

public class PathDataCatPage : ContentPage
{
    SKPath catPath = SKPath.ParseSvgPathData(
        "M 160 140 L 150 50 220 103" +           // Left ear
        "M 320 140 L 330 50 260 103" +           // Right ear
        "M 215 230 L 40 200" +                   // Left whiskers
        "M 215 240 L 40 240" +
        "M 215 250 L 40 280" +
        "M 265 230 L 440 200" +                 // Right whiskers
        "M 265 240 L 440 240" +
        "M 265 250 L 440 280" +
        "M 240 100" +                           // Head
        "A 100 100 0 0 1 240 300" +
        "A 100 100 0 0 1 240 100 Z" +
        "M 180 170" +                           // Left eye
        "A 40 40 0 0 1 220 170" +
        "A 40 40 0 0 1 180 170 Z" +
        "M 300 170" +                           // Right eye
        "A 40 40 0 0 1 260 170" +
        "A 40 40 0 0 1 300 170 Z");

    SKPaint paint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Orange,
        StrokeWidth = 5
    };
    ...
}

```

The head of a cat is a circle, and here it is rendered with two `A` commands, each of which draws a semicircle. Both `A` commands for the head define horizontal and vertical radii of 100. The first arc begins at (240, 100) and ends at (240, 300), which becomes the start point for the second arc that ends back at (240, 100).

The two eyes are also rendered with two `A` commands, and as with the cat's head, the second `A` command ends at the same point as the start of the first `A` command. However, these pairs of `A` commands do not define an ellipse. The width of each arc is 40 units and the radius is also 40 units, which means that these arcs are not full semicircles.

The `PaintSurface` handler performs similar transforms as the previous sample, but sets a single `Scale` factor to maintain the aspect ratio and provide a little margin so the cat's whiskers don't touch the sides of the screen:

```

public class PathDataCatPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear(SKColors.Black);

        SKRect bounds;
        catPath.GetBounds(out bounds);

        canvas.Translate(info.Width / 2, info.Height / 2);

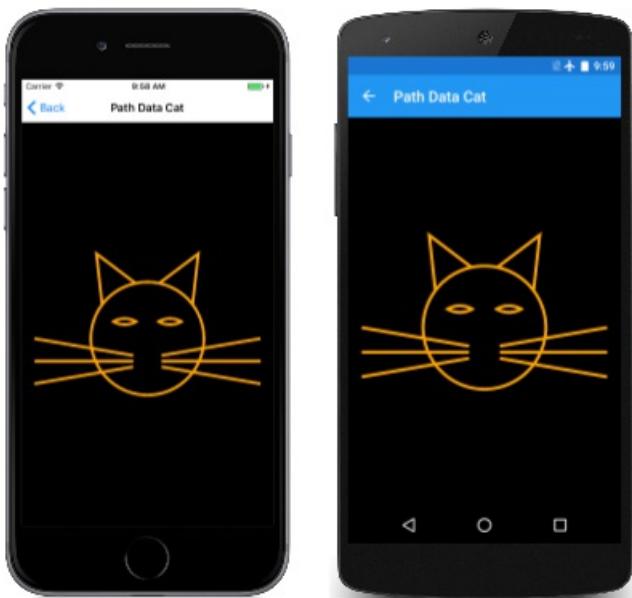
        canvas.Scale(0.9f * Math.Min(info.Width / bounds.Width,
                                     info.Height / bounds.Height));

        canvas.Translate(-bounds.MidX, -bounds.MidY);

        canvas.DrawPath(catPath, paint);
    }
}

```

Here's the program running:



Normally, when an `SKPath` object is defined as a field, the contours of the path must be defined in the constructor or another method. When using SVG path data, however, you've seen that the path can be specified entirely in the field definition.

The earlier **Ugly Analog Clock** sample in the [The Rotate Transform](#) article displayed the hands of the clock as simple lines. The **Pretty Analog Clock** program below replaces those lines with `SKPath` objects defined as fields in the [PrettyAnalogClockPage](#) class along with `SKPaint` objects:

```

public class PrettyAnalogClockPage : ContentPage
{
    ...
    // Clock hands pointing straight up
    SKPath hourHandPath = SKPath.ParseSvgPathData(
        "M 0 -60 C 0 -30 20 -30 5 -20 L 5 0" +
        "C 5 7.5 -5 7.5 -5 0 L -5 -20" +
        "C -20 -30 0 -30 0 -60 Z");

    SKPath minuteHandPath = SKPath.ParseSvgPathData(
        "M 0 -80 C 0 -75 0 -70 2.5 -60 L 2.5 0" +
        "C 2.5 5 -2.5 5 -2.5 0 L -2.5 -60" +
        "C 0 -70 0 -75 0 -80 Z");

    SKPath secondHandPath = SKPath.ParseSvgPathData(
        "M 0 10 L 0 -80");

    // SKPaint objects
    SKPaint handStrokePaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Black,
        StrokeWidth = 2,
        StrokeCap = SKStrokeCap.Round
    };

    SKPaint handFillPaint = new SKPaint
    {
        Style = SKPaintStyle.Fill,
        Color = SKColors.Gray
    };
    ...
}

```

The hour and minute hands now have enclosed areas. To make those hands distinct from each other, they are drawn with both a black outline and gray fill using the `handStrokePaint` and `handFillPaint` objects.

In the earlier [Ugly Analog Clock](#) sample, the little circles that marked the hours and minutes were drawn in a loop. In this [Pretty Analog Clock](#) sample, an entirely different approach is used: the hour and minute marks are dotted lines drawn with the `minuteMarkPaint` and `hourMarkPaint` objects:

```

public class PrettyAnalogClockPage : ContentPage
{
    ...
    SKPaint minuteMarkPaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Black,
        StrokeWidth = 3,
        StrokeCap = SKStrokeCap.Round,
        PathEffect = SKPathEffect.CreateDash(new float[] { 0, 3 * 3.14159f }, 0)
    };

    SKPaint hourMarkPaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Black,
        StrokeWidth = 6,
        StrokeCap = SKStrokeCap.Round,
        PathEffect = SKPathEffect.CreateDash(new float[] { 0, 15 * 3.14159f }, 0)
    };
    ...
}

```

The [Dots and Dashes](#) article discussed how you can use the `SKPathEffect.CreateDash` method to create a dashed line. The first argument is a `float` array that generally has two elements: The first element is the length of the dashes, and the second element is the gap between the dashes. When the `StrokeCap` property is set to `SKStrokeCap.Round`, then the rounded ends of the dash effectively lengthen the dash length by the stroke width on both sides of the dash. Thus, setting the first array element to 0 creates a dotted line.

The distance between these dots is governed by the second array element. As you'll see shortly, these two `SKPaint` objects are used to draw circles with a radius of 90 units. The circumference of this circle is therefore 180π , which means that the 60 minute marks must appear every 3π units, which is the second value in the `float` array in `minuteMarkPaint`. The 12 hour marks must appear every 15π units, which is the value in the second `float` array.

The `PrettyAnalogClockPage` class sets a timer to invalidate the surface every 16 milliseconds, and the `PaintSurface` handler is called at that rate. The earlier definitions of the `SKPath` and `SKPaint` objects allow for very clean drawing code:

```

public class PrettyAnalogClockPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Transform for 100-radius circle in center
        canvas.Translate(info.Width / 2, info.Height / 2);
        canvas.Scale(Math.Min(info.Width / 200, info.Height / 200));

        // Draw circles for hour and minute marks
        SKRect rect = new SKRect(-90, -90, 90, 90);
        canvas.DrawOval(rect, minuteMarkPaint);
        canvas.DrawOval(rect, hourMarkPaint);

        // Get time
        DateTime dateTime = DateTime.Now;

        // Draw hour hand
        canvas.Save();
        canvas.RotateDegrees(30 * dateTime.Hour + dateTime.Minute / 2f);
        canvas.DrawPath(hourHandPath, handStrokePaint);
        canvas.DrawPath(hourHandPath, handFillPaint);
        canvas.Restore();

        // Draw minute hand
        canvas.Save();
        canvas.RotateDegrees(6 * dateTime.Minute + dateTime.Second / 10f);
        canvas.DrawPath(minuteHandPath, handStrokePaint);
        canvas.DrawPath(minuteHandPath, handFillPaint);
        canvas.Restore();

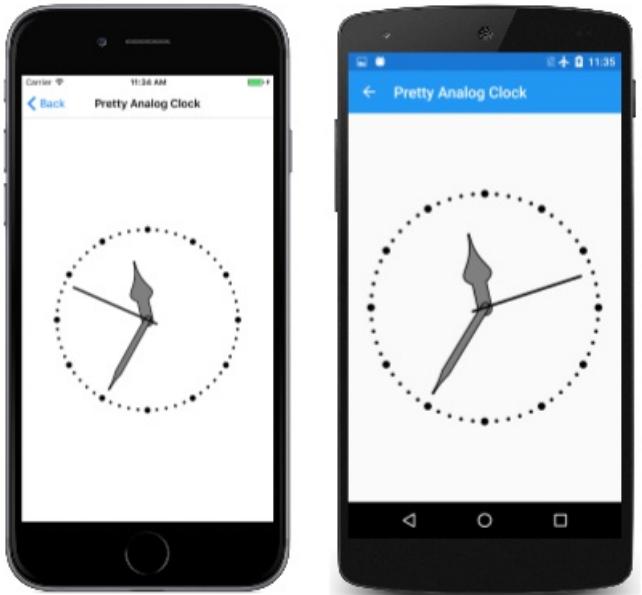
        // Draw second hand
        double t = dateTime.Millisecond / 1000.0;

        if (t < 0.5)
        {
            t = 0.5 * Easing.SpringIn.Ease(t / 0.5);
        }
        else
        {
            t = 0.5 * (1 + Easing.SpringOut.Ease((t - 0.5) / 0.5));
        }

        canvas.Save();
        canvas.RotateDegrees(6 * (dateTime.Second + (float)t));
        canvas.DrawPath(secondHandPath, handStrokePaint);
        canvas.Restore();
    }
}

```

Something special is done with the second hand, however. Because the clock is updated every 16 milliseconds, the `Millisecond` property of the `DateTime` value can potentially be used to animate a sweep second hand instead of one that moves in discrete jumps from second to second. But this code does not allow the movement to be smooth. Instead, it uses the Xamarin.Forms `SpringIn` and `SpringOut` animation easing functions for a different kind of movement. These easing functions cause the second hand to move in a jerkier manner — pulling back a little before it moves, and then slightly over-shooting its destination, an effect that unfortunately can't be reproduced in these static screenshots:



Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Clipping with Paths and Regions

3/5/2021 • 12 minutes to read • [Edit Online](#)

 [Download the sample](#)

Use paths to clip graphics to specific areas, and to create regions

It's sometimes necessary to restrict the rendering of graphics to a particular area. This is known as *clipping*. You can use clipping for special effects, such as this image of a monkey seen through a keyhole:



The *clipping area* is the area of the screen in which graphics are rendered. Anything that is displayed outside of the clipping area is not rendered. The clipping area is usually defined by a rectangle or an `SKPath` object, but you can alternatively define a clipping area using an `SKRegion` object. These two types of objects at first seem related because you can create a region from a path. However, you cannot create a path from a region, and they are very different internally: A path comprises a series of lines and curves, while a region is defined by a series of horizontal scan lines.

The image above was created by the **Monkey through Keyhole** page. The `MonkeyThroughKeyholePage` class defines a path using SVG data and uses the constructor to load a bitmap from program resources:

```

public class MonkeyThroughKeyholePage : ContentPage
{
    SKBitmap bitmap;
    SKPath keyholePath = SKPath.ParseSvgPathData(
        "M 300 130 L 250 350 L 450 350 L 400 130 A 70 70 0 1 0 300 130 Z");

    public MonkeyThroughKeyholePage()
    {
        Title = "Monkey through Keyhole";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;

        string resourceId = "SkiaSharpFormsDemos.Media.SeatedMonkey.jpg";
        Assembly assembly = GetType().GetTypeInfo().Assembly;

        using (Stream stream = assembly.GetManifestResourceStream(resourceId))
        {
            bitmap = SKBitmap.Decode(stream);
        }
    }
    ...
}

```

Although the `keyholePath` object describes the outline of a keyhole, the coordinates are completely arbitrary and reflect what was convenient when the path data was devised. For this reason, the `PaintSurface` handler obtains the bounds of this path and calls `Translate` and `Scale` to move the path to the center of the screen and to make it nearly as tall as the screen:

```

public class MonkeyThroughKeyholePage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Set transform to center and enlarge clip path to window height
        SKRect bounds;
        keyholePath.GetTightBounds(out bounds);

        canvas.Translate(info.Width / 2, info.Height / 2);
        canvas.Scale(0.98f * info.Height / bounds.Height);
        canvas.Translate(-bounds.MidX, -bounds.MidY);

        // Set the clip path
        canvas.ClipPath(keyholePath);

        // Reset transforms
        canvas.ResetMatrix();

        // Display monkey to fill height of window but maintain aspect ratio
        canvas.DrawBitmap(bitmap,
            new SKRect((info.Width - info.Height) / 2, 0,
                (info.Width + info.Height) / 2, info.Height));
    }
}

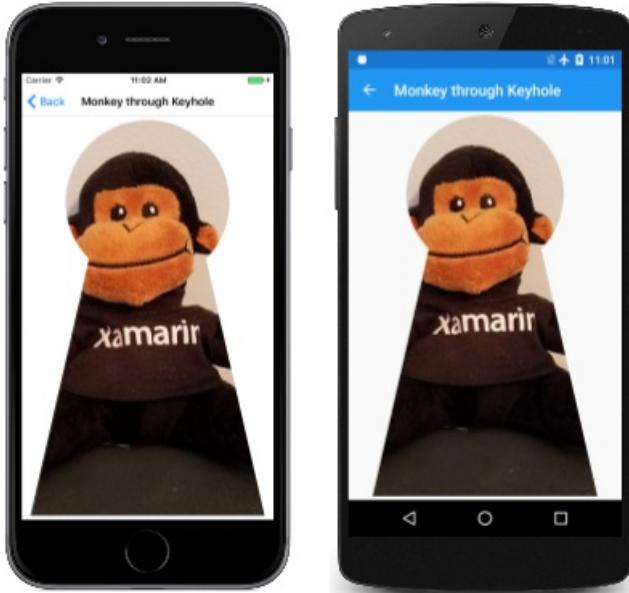
```

But the path is not rendered. Instead, following the transforms, the path is used to set a clipping area with this

statement:

```
canvas.ClipPath(keyholePath);
```

The `PaintSurface` handler then resets the transforms with a call to `ResetMatrix` and draws the bitmap to extend to the full height of the screen. This code assumes that the bitmap is square, which this particular bitmap is. The bitmap is rendered only within the area defined by the clipping path:



The clipping path is subject to the transforms in effect when the `ClipPath` method is called, and not to the transforms in effect when a graphical object (such as a bitmap) is displayed. The clipping path is part of the canvas state that is saved with the `Save` method and restored with the `Restore` method.

Combining Clipping Paths

Strictly speaking, the clipping area is not "set" by the `ClipPath` method. Instead, it is combined with the existing clipping path, which begins as a rectangle equal in size to the canvas. You can obtain the rectangular bounds of the clipping area using the `ClipBounds` property or the `ClipDeviceBounds` property. The `ClipBounds` property returns an `SKRect` value that reflects any transforms that might be in effect. The `ClipDeviceBounds` property returns a `RectI` value. This is a rectangle with integer dimensions, and describes the clipping area in actual pixel dimensions.

Any call to `ClipPath` reduces the clipping area by combining the clipping area with a new area. The full syntax of the `ClipPath` method is:

```
public void ClipPath(SKPath path, SKClipOperation operation = SKClipOperation.Intersect, Boolean antialias = false);
```

There is also a `ClipRect` method that combines the clipping area with a rectangle:

```
public Void ClipRect(SKRect rect, SKClipOperation operation = SKClipOperation.Intersect, Boolean antialias = false);
```

By default, the resultant clipping area is an intersection of the existing clipping area and the `SKPath` or `SKRect` that is specified in the `ClipPath` or `ClipRect` method. This is demonstrated in the [Four Circles Intersect Clip](#) page. The `PaintSurface` handler in the [FourCircleInteresectClipPage](#) class reuses the same `SKPath` object to create four overlapping circles, each of which reduces the clipping area through successive calls to `ClipPath`:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    float size = Math.Min(info.Width, info.Height);
    float radius = 0.4f * size;
    float offset = size / 2 - radius;

    // Translate to center
    canvas.Translate(info.Width / 2, info.Height / 2);

    using (SKPath path = new SKPath())
    {
        path.AddCircle(-offset, -offset, radius);
        canvas.ClipPath(path, SKClipOperation.Intersect);

        path.Reset();
        path.AddCircle(-offset, offset, radius);
        canvas.ClipPath(path, SKClipOperation.Intersect);

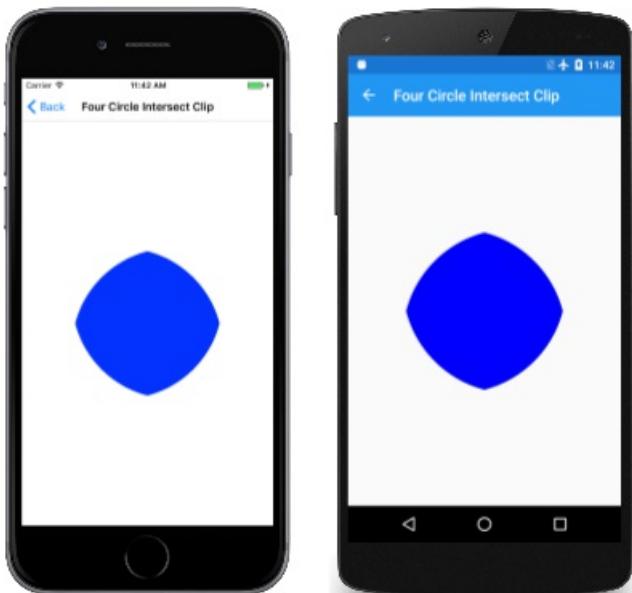
        path.Reset();
        path.AddCircle(offset, -offset, radius);
        canvas.ClipPath(path, SKClipOperation.Intersect);

        path.Reset();
        path.AddCircle(offset, offset, radius);
        canvas.ClipPath(path, SKClipOperation.Intersect);

        using (SKPaint paint = new SKPaint())
        {
            paint.Style = SKPaintStyle.Fill;
            paint.Color = SKColors.Blue;
            canvas.DrawPaint(paint);
        }
    }
}

```

What's left is the intersection of these four circles:



The `SKClipOperation` enumeration has only two members:

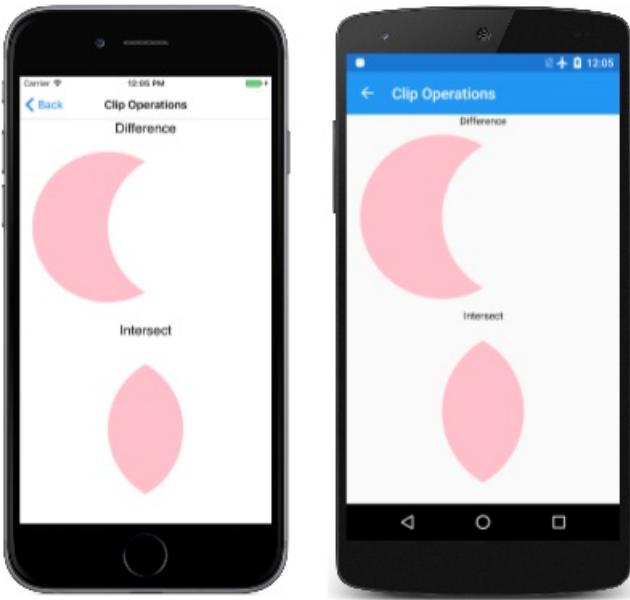
- `Difference` removes the specified path or rectangle from the existing clipping area
- `Intersect` intersects the specified path or rectangle with the existing clipping area

If you replace the four `SKClipOperation.Intersect` arguments in the `FourCircleIntersectClipPage` class with `SKClipOperation.Difference`, you'll see the following:



Four overlapping circles have been removed from the clipping area.

The **Clip Operations** page illustrates the difference between these two operations with just a pair of circles. The first circle on the left is added to the clipping area with the default clip operation of `Intersect`, while the second circle on the right is added to the clipping area with the clip operation indicated by the text label:



The `clipOperationsPage` class defines two `SKPaint` objects as fields, and then divides the screen up into two rectangular areas. These areas are different depending on whether the phone is in portrait or landscape mode. The `DisplayClipOp` class then displays the text and calls `clipPath` with the two circle paths to illustrate each clip operation:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    float x = 0;
    float y = 0;

    foreach (SKClipOperation clipOp in Enum.GetValues(typeof(SKClipOperation)))
    {
        // Portrait mode
        if (info.Height > info.Width)
        {
            DisplayClipOp(canvas, new SKRect(x, y, x + info.Width, y + info.Height / 2), clipOp);
            y += info.Height / 2;
        }
        // Landscape mode
        else
        {
            DisplayClipOp(canvas, new SKRect(x, y, x + info.Width / 2, y + info.Height), clipOp);
            x += info.Width / 2;
        }
    }
}

void DisplayClipOp(SKCanvas canvas, SKRect rect, SKClipOperation clipOp)
{
    float textSize = textPaint.TextSize;
    canvas.DrawText(clipOp.ToString(), rect.MidX, rect.Top + textSize, textPaint);
    rect.Top += textSize;

    float radius = 0.9f * Math.Min(rect.Width / 3, rect.Height / 2);
    float xCenter = rect.MidX;
    float yCenter = rect.MidY;

    canvas.Save();

    using (SKPath path1 = new SKPath())
    {
        path1.AddCircle(xCenter - radius / 2, yCenter, radius);
        canvas.ClipPath(path1);

        using (SKPath path2 = new SKPath())
        {
            path2.AddCircle(xCenter + radius / 2, yCenter, radius);
            canvas.ClipPath(path2, clipOp);

            canvas.DrawPaint(fillPaint);
        }
    }

    canvas.Restore();
}

```

Calling `DrawPaint` normally causes the entire canvas to be filled with that `SKPaint` object, but in this case, the method just paints within the clipping area.

Exploring Regions

You can also define a clipping area in terms of an `SKRegion` object.

A newly created `SKRegion` object describes an empty area. Usually the first call on the object is `SetRect` so that

the region describes a rectangular area. The parameter to `SetRect` is an `SKRectI` value — a rectangle with integer coordinates because it specifies the rectangle in terms of pixels. You can then call `SetPath` with an `SKPath` object. This creates a region that is the same as the interior of the path, but clipped to the initial rectangular region.

The region can also be modified by calling one of the `Op` method overloads, such as this one:

```
public Boolean Op(SKRegion region, SKRegionOperation op)
```

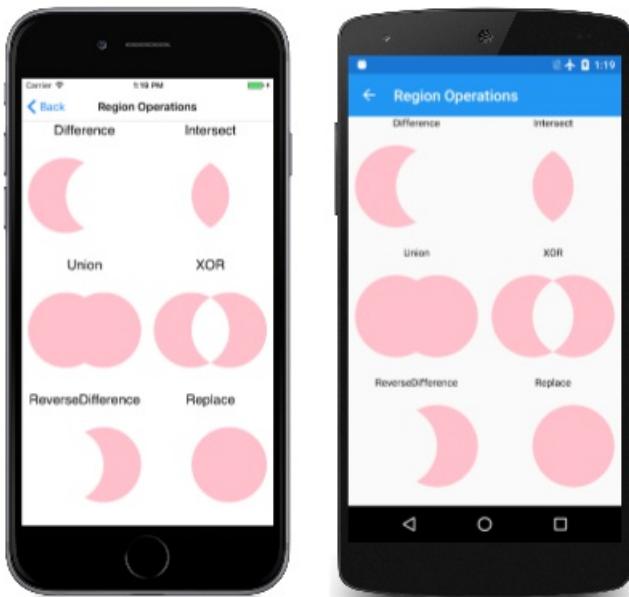
The `SKRegionOperation` enumeration is similar to `SKClipOperation` but it has more members:

- `Difference`
- `Intersect`
- `Union`
- `XOR`
- `ReverseDifference`
- `Replace`

The region that you're making the `Op` call on is combined with the region specified as a parameter based on the `SKRegionOperation` member. When you finally get a region suitable for clipping, you can set that as the clipping area of the canvas using the `ClipRegion` method of `SKCanvas`:

```
public void ClipRegion(SKRegion region, SKClipOperation operation = SKClipOperation.Intersect)
```

The following screenshot shows clipping areas based on the six region operations. The left circle is the region that the `Op` method is called on, and the right circle is the region passed to the `Op` method:



Are these all the possibilities of combining these two circles? Consider the resultant image as a combination of three components, which by themselves are seen in the `Difference`, `Intersect`, and `ReverseDifference` operations. The total number of combinations is two to the third power, or eight. The two that are missing are the original region (which results from not calling `Op` at all) and an entirely empty region.

It's harder to use regions for clipping because you need to first create a path, and then a region from that path, and then combine multiple regions. The overall structure of the **Region Operations** page is very similar to

Clip Operations but the [RegionOperationsPage](#) class divides the screen up into six areas and shows the extra work required to use regions for this job:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    float x = 0;
    float y = 0;
    float width = info.Height > info.Width ? info.Width / 2 : info.Width / 3;
    float height = info.Height > info.Width ? info.Height / 3 : info.Height / 2;

    foreach (SKRegionOperation regionOp in Enum.GetValues(typeof(SKRegionOperation)))
    {
        DisplayClipOp(canvas, new SKRect(x, y, x + width, y + height), regionOp);

        if ((x += width) >= info.Width)
        {
            x = 0;
            y += height;
        }
    }
}

void DisplayClipOp(SKCanvas canvas, SKRect rect, SKRegionOperation regionOp)
{
    float textSize = textPaint.TextSize;
    canvas.DrawText(regionOp.ToString(), rect.MidX, rect.Top + textSize, textPaint);
    rect.Top += textSize;

    float radius = 0.9f * Math.Min(rect.Width / 3, rect.Height / 2);
    float xCenter = rect.MidX;
    float yCenter = rect.MidY;

    SKRectI recti = new SKRectI((int)rect.Left, (int)rect.Top,
                                (int)rect.Right, (int)rect.Bottom);

    using (SKRegion wholeRectRegion = new SKRegion())
    {
        wholeRectRegion.SetRect(recti);

        using (SKRegion region1 = new SKRegion(wholeRectRegion))
        using (SKRegion region2 = new SKRegion(wholeRectRegion))
        {
            using (SKPath path1 = new SKPath())
            {
                path1.AddCircle(xCenter - radius / 2, yCenter, radius);
                region1.SetPath(path1);
            }

            using (SKPath path2 = new SKPath())
            {
                path2.AddCircle(xCenter + radius / 2, yCenter, radius);
                region2.SetPath(path2);
            }

            region1.Op(region2, regionOp);

            canvas.Save();
            canvas.ClipRegion(region1);
            canvas.DrawPaint(fillPaint);
            canvas.Restore();
        }
    }
}

```

Here's a big difference between the `ClipPath` method and the `clipRegion` method:

IMPORTANT

Unlike the `ClipPath` method, the `clipRegion` method is not affected by transforms.

To understand the rationale for this difference, it's helpful to understand what a region is. If you've thought about how the clip operations or region operations might be implemented internally, it probably seems very complicated. Several potentially very complex paths are being combined, and the outline of the resultant path is likely an algorithmic nightmare.

This job is simplified considerably if each path is reduced to a series of horizontal scan lines, such as those in old-fashioned vacuum tube TVs. Each scan line is simply a horizontal line with a start point and an end point. For example, a circle with a radius of 10 pixels can be decomposed into 20 horizontal scan lines, each of which starts at the left part of the circle and ends at the right part. Combining two circles with any region operation becomes very simple because it's simply a matter of examining the start and end coordinates of each pair of corresponding scan lines.

This is what a region is: A series of horizontal scan lines that define an area.

However, when an area is reduced to a series of scan lines, these scan lines are based on a particular pixel dimension. Strictly speaking, the region is not a vector graphics object. It is closer in nature to a compressed monochrome bitmap than to a path. Consequently, regions cannot be scaled or rotated without losing fidelity, and for this reason they are not transformed when used for clipping areas.

However, you can apply transforms to regions for painting purposes. The `Region Paint` program vividly demonstrates the inner nature of regions. The `RegionPaintPage` class creates an `SKRegion` object based on an `SKPath` of a 10-unit radius circle. A transform then expands that circle to fill the page:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    int radius = 10;

    // Create circular path
    using (SKPath circlePath = new SKPath())
    {
        circlePath.AddCircle(0, 0, radius);

        // Create circular region
        using (SKRegion circleRegion = new SKRegion())
        {
            circleRegion.SetRect(new SKRectI(-radius, -radius, radius, radius));
            circleRegion.SetPath(circlePath);

            // Set transform to move it to center and scale up
            canvas.Translate(info.Width / 2, info.Height / 2);
            canvas.Scale(Math.Min(info.Width / 2, info.Height / 2) / radius);

            // Fill region
            using (SKPaint fillPaint = new SKPaint())
            {
                fillPaint.Style = SKPaintStyle.Fill;
                fillPaint.Color = SKColors.Orange;

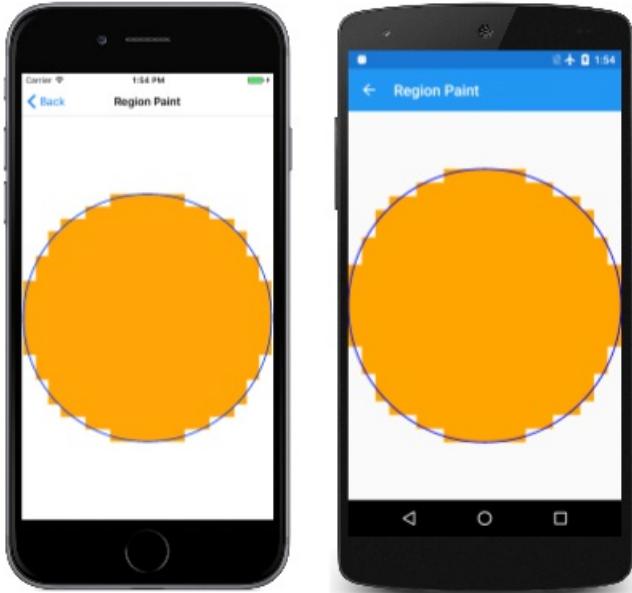
                canvas.DrawRegion(circleRegion, fillPaint);
            }

            // Stroke path for comparison
            using (SKPaint strokePaint = new SKPaint())
            {
                strokePaint.Style = SKPaintStyle.Stroke;
                strokePaint.Color = SKColors.Blue;
                strokePaint.StrokeWidth = 0.1f;

                canvas.DrawPath(circlePath, strokePaint);
            }
        }
    }
}

```

The `DrawRegion` call fills the region in orange, while the `DrawPath` call strokes the original path in blue for comparison:



The region is clearly a series of discrete coordinates.

If you don't need to use transforms in connection with your clipping areas, you can use regions for clipping, as the **Four-Leaf Clover** page demonstrates. The [FourLeafCloverPage](#) class constructs a composite region from four circular regions, sets that composite region as the clipping area, and then draws a series of 360 straight lines emanating from the center of the page:

```
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    float xCenter = info.Width / 2;
    float yCenter = info.Height / 2;
    float radius = 0.24f * Math.Min(info.Width, info.Height);

    using (SKRegion wholeScreenRegion = new SKRegion())
    {
        wholeScreenRegion.SetRect(new SKRectI(0, 0, info.Width, info.Height));

        using (SKRegion leftRegion = new SKRegion(wholeScreenRegion))
        using (SKRegion rightRegion = new SKRegion(wholeScreenRegion))
        using (SKRegion topRegion = new SKRegion(wholeScreenRegion))
        using (SKRegion bottomRegion = new SKRegion(wholeScreenRegion))
        {
            using (SKPath circlePath = new SKPath())
            {
                // Make basic circle path
                circlePath.AddCircle(xCenter, yCenter, radius);

                // Left leaf
                circlePath.Transform(SKMatrix.MakeTranslation(-radius, 0));
                leftRegion.SetPath(circlePath);

                // Right leaf
                circlePath.Transform(SKMatrix.MakeTranslation(2 * radius, 0));
                rightRegion.SetPath(circlePath);

                // Make union of right with left
                leftRegion.Op(rightRegion, SKRegionOperation.Union);

                // Top leaf
                circlePath.Transform(SKMatrix.MakeTranslation(0, -radius));
                topRegion.SetPath(circlePath);
            }

            // Make union of top with left
            leftRegion.Op(topRegion, SKRegionOperation.Union);

            // Make union of top with right
            rightRegion.Op(topRegion, SKRegionOperation.Union);
        }
    }

    canvas.Clip(wholeScreenRegion);
    canvas.DrawLines(new List<SKLine>());
}
```

```

        circlePath.Transform(SKMatrix.MakeTranslation(-radius, -radius));
        topRegion.SetPath(circlePath);

        // Combine with bottom leaf
        circlePath.Transform(SKMatrix.MakeTranslation(0, 2 * radius));
        bottomRegion.SetPath(circlePath);

        // Make union of top with bottom
        bottomRegion.Op(topRegion, SKRegionOperation.Union);

        // Exclusive-OR left and right with top and bottom
        leftRegion.Op(bottomRegion, SKRegionOperation.XOR);

        // Set that as clip region
        canvas.ClipRegion(leftRegion);

        // Set transform for drawing lines from center
        canvas.Translate(xCenter, yCenter);

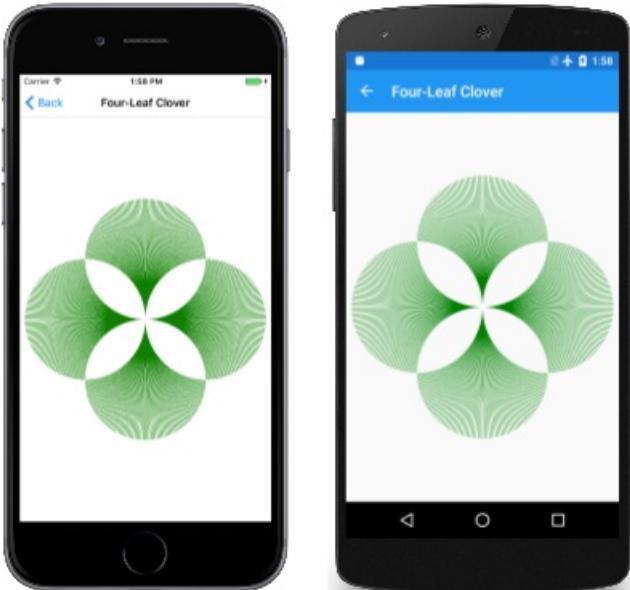
        // Draw 360 lines
        for (double angle = 0; angle < 360; angle++)
        {
            float x = 2 * radius * (float)Math.Cos(Math.PI * angle / 180);
            float y = 2 * radius * (float)Math.Sin(Math.PI * angle / 180);

            using (SKPaint strokePaint = new SKPaint())
            {
                strokePaint.Color = SKColors.Green;
                strokePaint.StrokeWidth = 2;

                canvas.DrawLine(0, 0, x, y, strokePaint);
            }
        }
    }
}

```

It doesn't really look like a four-leaf clover, but it's an image that might otherwise be hard to render without clipping:



Related Links

- SkiaSharp APIs
 - SkiaSharpFormsDemos (sample)

Path Effects in SkiaSharp

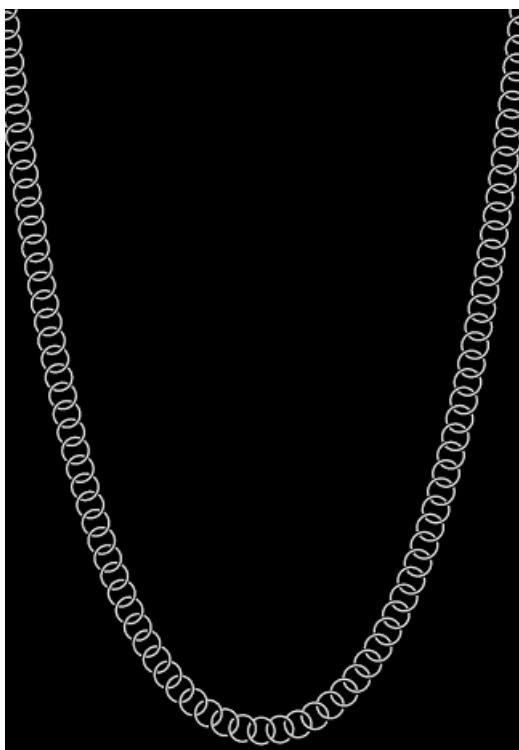
3/5/2021 • 39 minutes to read • [Edit Online](#)



[Download the sample](#)

Discover the various path effects that allow paths to be used for stroking and filling

A *path effect* is an instance of the `SKPathEffect` class that is created with one of eight static creation methods defined by the class. The `SKPathEffect` object is then set to the `PathEffect` property of an `SKPaint` object for a variety of interesting effects, for example, stroking a line with a small replicated path:



Path effects allow you to:

- Stroke a line with dots and dashes
- Stroke a line with any filled path
- Fill an area with hatch lines
- Fill an area with a tiled path
- Make sharp corners rounded
- Add random "jitter" to lines and curves

In addition, you can combine two or more path effects.

This article also demonstrates how to use the `GetFillPath` method of `SKPaint` to convert one path into another path by applying properties of `SKPaint`, including `StrokeWidth` and `PathEffect`. This results in some interesting techniques, such as obtaining a path that is an outline of another path. `GetFillPath` is also helpful in connection with path effects.

Dots and Dashes

The use of the `PathEffect.CreateDash` method was described in the article [Dots and Dashes](#). The first argument of the method is an array containing an even number of two or more values, alternating between

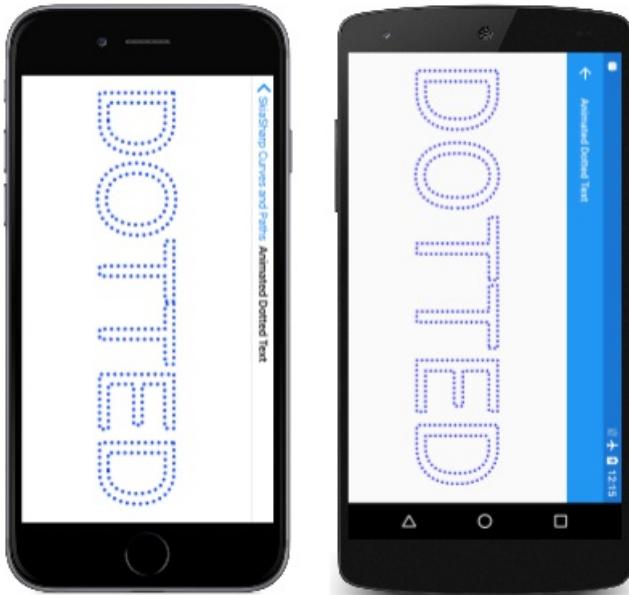
lengths of dashes and lengths of gaps between the dashes:

```
public static SKPathEffect CreateDash (Single[] intervals, Single phase)
```

These values are *not* relative to the stroke width. For example, if the stroke width is 10, and you want a line composed of square dashes and square gaps, set the `intervals` array to { 10, 10 }. The `phase` argument indicates where within the dash pattern the line begins. In this example, if you want the line to begin with the square gap, set `phase` to 10.

The ends of the dashes are affected by the `StrokeCap` property of `SKPaint`. For wide stroke widths, it is very common to set this property to `skStrokeCap.Round` to round the ends of the dashes. In this case, the values in the `intervals` array do *not* include the extra length resulting from the rounding. This fact means that a circular dot requires specifying a width of zero. For a stroke width of 10, to create a line with circular dots and gaps between the dots of the same diameter, use an `intervals` array of { 0, 20 }.

The **Animated Dotted Text** page is similar to the **Outlined Text** page described in the article [Integrating Text and Graphics](#) in that it displays outlined text characters by setting the `Style` property of the `SKPaint` object to `SKPaintStyle.Stroke`. In addition, **Animated Dotted Text** uses `SKPathEffect.CreateDash` to give this outline a dotted appearance, and the program also animates the `phase` argument of the `SKPathEffect.CreateDash` method to make the dots seem to travel around the text characters. Here's the page in landscape mode:



The `AnimatedDottedTextPage` class begins by defining some constants, and also overrides the `OnAppearing` and `OnDisappearing` methods for the animation:

```

public class AnimatedDottedTextPage : ContentPage
{
    const string text = "DOTTED";
    const float strokeWidth = 10;
    static readonly float[] dashArray = { 0, 2 * strokeWidth };

    SKCanvasView canvasView;
    bool pageIsActive;

    public AnimatedDottedTextPage()
    {
        Title = "Animated Dotted Text";

        canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    protected override void OnAppearing()
    {
        base.OnAppearing();
        pageIsActive = true;

        Device.StartTimer(TimeSpan.FromSeconds(1f / 60), () =>
        {
            canvasView.InvalidateSurface();
            return pageIsActive;
        });
    }

    protected override void OnDisappearing()
    {
        base.OnDisappearing();
        pageIsActive = false;
    }
    ...
}

```

The `PaintSurface` handler begins by creating an `SKPaint` object to display the text. The `TextSize` property is adjusted based on the width of the screen:

```

public class AnimatedDottedTextPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Create an SKPaint object to display the text
        using (SKPaint textPaint = new SKPaint
        {
            Style = SKPaintStyle.Stroke,
            StrokeWidth = strokeWidth,
            StrokeCap = SKStrokeCap.Round,
            Color = SKColors.Blue,
        })
        {
            // Adjust TextSize property so text is 95% of screen width
            float textWidth = textPaint.MeasureText(text);
            textPaint.TextSize *= 0.95f * info.Width / textWidth;

            // Find the text bounds
            SKRect textBounds = new SKRect();
            textPaint.MeasureText(text, ref textBounds);

            // Calculate offsets to center the text on the screen
            float xText = info.Width / 2 - textBounds.MidX;
            float yText = info.Height / 2 - textBounds.MidY;

            // Animate the phase; t is 0 to 1 every second
            TimeSpan timeSpan = new TimeSpan(DateTime.Now.Ticks);
            float t = (float)(timeSpan.TotalSeconds % 1 / 1);
            float phase = -t * 2 * strokeWidth;

            // Create dotted line effect based on dash array and phase
            using (SKPathEffect dashEffect = SKPathEffect.CreateDash(dashArray, phase))
            {
                // Set it to the paint object
                textPaint.PathEffect = dashEffect;

                // And draw the text
                canvas.DrawText(text, xText, yText, textPaint);
            }
        }
    }
}

```

Towards the end of the method, the `SKPathEffect.CreateDash` method is called using the `dashArray` that is defined as a field, and the animated `phase` value. The `SKPathEffect` instance is set to the `PathEffect` property of the `SKPaint` object to display the text.

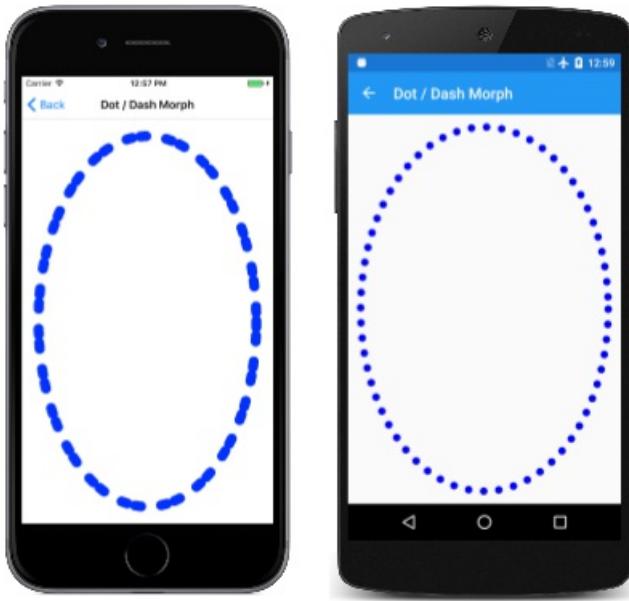
Alternatively, you can set the `SKPathEffect` object to the `SKPaint` object prior to measuring the text and centering it on the page. In that case, however, the animated dots and dashes cause some variation in the size of the rendered text, and the text tends to vibrate a little. (Try it!)

You'll also notice that as the animated dots circle around the text characters, there is a certain point in each closed curve where the dots seem to pop in and out of existence. This is where the path that defines the character outline begins and ends. If the path length is not an integral multiple of the length of the dash pattern (in this case 20 pixels), then only part of that pattern can fit at the end of the path.

It's possible to adjust the length of the dash pattern to fit the length of the path, but that requires determining

the length of the path, a technique that is covered in the article [Path Information and Enumeration](#).

The **Dot / Dash Morph** program animates the dash pattern itself so that dashes seem to divide into dots, which combine to form dashes again:



The `DotDashMorphPage` class overrides the `OnAppearing` and `OnDisappearing` methods just as the previous program did, but the class defines the `SKPaint` object as a field:

```
public class DotDashMorphPage : ContentPage
{
    const float strokeWidth = 30;
    static readonly float[] dashArray = new float[4];

    SKCanvasView canvasView;
    bool pageIsActive = false;

    SKPaint ellipsePaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        StrokeWidth = strokeWidth,
        StrokeCap = SKStrokeCap.Round,
        Color = SKColors.Blue
    };
    ...

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Create elliptical path
        using (SKPath ellipsePath = new SKPath())
        {
            ellipsePath.AddOval(new SKRect(50, 50, info.Width - 50, info.Height - 50));

            // Create animated path effect
            TimeSpan timeSpan = new TimeSpan(DateTime.Now.Ticks);
            float t = (float)(timeSpan.TotalSeconds % 3 / 3);
            float phase = 0;

            if (t < 0.25f) // 1, 0, 1, 2 --> 0, 2, 0, 2
            {
                float tsub = 4 * t;
                dashArray[0] = strokeWidth * (1 - tsub);
                dashArray[1] = strokeWidth * tsub;
                dashArray[2] = 0;
                dashArray[3] = 0;
            }
            else if (t < 0.5f) // 0, 2, 0, 2 --> 1, 0, 1, 2
            {
                float tsub = 4 * (t - 0.25f);
                dashArray[0] = strokeWidth * (1 - tsub);
                dashArray[1] = strokeWidth * tsub;
                dashArray[2] = 0;
                dashArray[3] = 0;
            }
            else // 1, 0, 1, 2
            {
                float tsub = 4 * (t - 0.5f);
                dashArray[0] = strokeWidth * (1 - tsub);
                dashArray[1] = strokeWidth * tsub;
                dashArray[2] = 0;
                dashArray[3] = 0;
            }
        }

        canvas.DrawPath(ellipsePath, ellipsePaint);
    }
}
```

The `PaintSurface` handler creates an elliptical path based on the size of the page, and executes a long section of code that sets the `dashArray` and `phase` variables. As the animated variable `t` ranges from 0 to 1, the `if` blocks break up that time into four quarters, and in each of those quarters, `tsub` also ranges from 0 to 1. At the very end, the program creates the `SKPathEffect` and sets it to the `SKPaint` object for drawing.

From Path to Path

The `GetFillPath` method of `SKPaint` turns one path into another based on settings in the `SKPaint` object. To see how this works, replace the `canvas.DrawPath` call in the previous program with the following code:

```
SKPath newPath = new SKPath();
bool fill = ellipsePaint.GetFillPath(ellipsePath, newPath);
SKPaint newPaint = new SKPaint
{
    Style = fill ? SKPaintStyle.Fill : SKPaintStyle.Stroke
};
canvas.DrawPath(newPath, newPaint);
```

In this new code, the `GetFillPath` call converts the `ellipsePath` (which is just an oval) into `newPath`, which is then displayed with `newPaint`. The `newPaint` object is created with all default property settings except that the `style` property is set based on the Boolean return value from `GetFillPath`.

The visuals are identical except for the color, which is set in `ellipsePaint` but not `newPaint`. Rather than the simple ellipse defined in `ellipsePath`, `newPath` contains numerous path contours that define the series of dots and dashes. This is the result of applying various properties of `ellipsePaint` (specifically, `StrokeWidth`, `StrokeCap`, and `PathEffect`) to `ellipsePath` and putting the resultant path in `newPath`. The `GetFillPath` method returns a Boolean value indicating whether or not the destination path is to be filled; in this example, the return value is `true` for filling the path.

Try changing the `Style` setting in `newPaint` to `SKPaintStyle.Stroke` and you'll see the individual path contours outlined with a one-pixel-width line.

Stroking with a Path

The `SKPathEffect.Create1DPath` method is conceptually similar to `SKPathEffect.CreateDash` except that you specify a path rather than a pattern of dashes and gaps. This path is replicated multiple times to stroke the line or curve.

The syntax is:

```
public static SKPathEffect Create1DPath (SKPath path, Single advance,  
                                         Single phase, SKPath1DPathEffectStyle style)
```

In general, the path that you pass to `Create1DPath` will be small and centered around the point (0, 0). The `advance` parameter indicates the distance between the centers of the path as the path is replicated on the line. You usually set this argument to the approximate width of the path. The `phase` argument plays the same role here as it does in the `CreateDash` method.

The `SKPath1DPathEffectStyle` has three members:

- `Translate`
- `Rotate`
- `Morph`

The `Translate` member causes the path to remain in the same orientation as it is replicated along a line or curve. For `Rotate`, the path is rotated based on a tangent to the curve. The path has its normal orientation for horizontal lines. `Morph` is similar to `Rotate` except that the path itself is also curved to match the curvature of the line being stroked.

The **1D Path Effect** page demonstrates these three options. The [OneDimensionalPathEffectPage.xaml](#) file defines a picker containing three items corresponding to the three members of the enumeration:

```

<?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:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Curves.OneDimensionalPathEffectPage"
    Title="1D Path Effect">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Picker x:Name="effectStylePicker"
            Title="Effect Style"
            Grid.Row="0"
            SelectedIndexChanged="OnPickerSelectedIndexChanged">
            <Picker.ItemsSource>
                <x:Array Type="{x:Type x:String}">
                    <x:String>Translate</x:String>
                    <x:String>Rotate</x:String>
                    <x:String>Morph</x:String>
                </x:Array>
            </Picker.ItemsSource>
            <Picker.SelectedIndex>
                0
            </Picker.SelectedIndex>
        </Picker>

        <skia:SKCanvasView x:Name="canvasView"
            PaintSurface="OnCanvasViewPaintSurface"
            Grid.Row="1" />
    </Grid>
</ContentPage>

```

The [OneDimensionalPathEffectPage.xaml.cs](#) code-behind file defines three `SKPathEffect` objects as fields.

These are all created using `SKPathEffect.Create1DPath` with `SKPath` objects created using `SKPath.ParseSvgPathData`. The first is a simple box, the second is a diamond shape, and the third is a rectangle. These are used to demonstrate the three effect styles:

```

public partial class OneDimensionalPathEffectPage : ContentPage
{
    SKPathEffect translatePathEffect =
        SKPathEffect.Create1DPath(SKPath.ParseSvgPathData("M -10 -10 L 10 -10, 10 10, -10 10 Z"),
                                24, 0, SKPath1DPathEffectStyle.Translate);

    SKPathEffect rotatePathEffect =
        SKPathEffect.Create1DPath(SKPath.ParseSvgPathData("M -10 0 L 0 -10, 10 0, 0 10 Z"),
                                20, 0, SKPath1DPathEffectStyle.Rotate);

    SKPathEffect morphPathEffect =
        SKPathEffect.Create1DPath(SKPath.ParseSvgPathData("M -25 -10 L 25 -10, 25 10, -25 10 Z"),
                                55, 0, SKPath1DPathEffectStyle.Morph);

    SKPaint pathPaint = new SKPaint
    {
        Color = SKColors.Blue
    };

    public OneDimensionalPathEffectPage()
    {
        InitializeComponent();
    }

    void OnPickerSelectedIndexChanged(object sender, EventArgs args)
    {
        if (canvasView != null)
        {
            canvasView.InvalidateSurface();
        }
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        using (SKPath path = new SKPath())
        {
            path.MoveTo(new SKPoint(0, 0));
            path.CubicTo(new SKPoint(2 * info.Width, info.Height),
                        new SKPoint(-info.Width, info.Height),
                        new SKPoint(info.Width, 0));

            switch ((string)effectStylePicker.SelectedItem)
            {
                case "Translate":
                    pathPaint.PathEffect = translatePathEffect;
                    break;

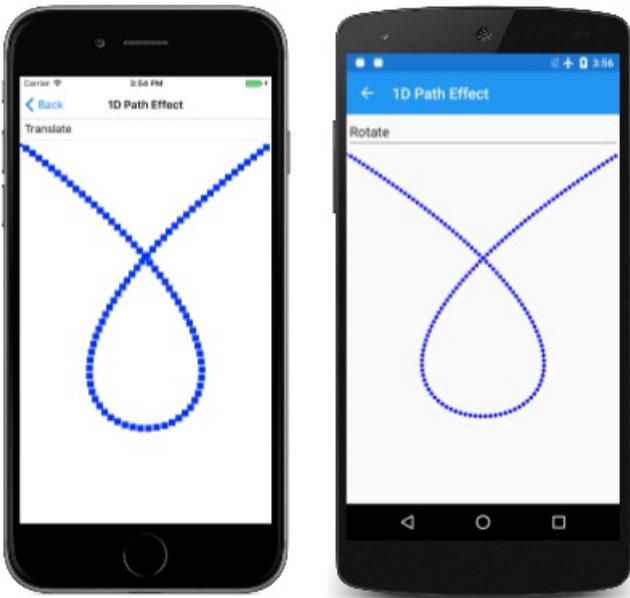
                case "Rotate":
                    pathPaint.PathEffect = rotatePathEffect;
                    break;

                case "Morph":
                    pathPaint.PathEffect = morphPathEffect;
                    break;
            }

            canvas.DrawPath(path, pathPaint);
        }
    }
}

```

The `PaintSurface` handler creates a Bézier curve that loops around itself, and accesses the picker to determine which `PathEffect` should be used to stroke it. The three options — `Translate`, `Rotate`, and `Morph` — are shown from left to right:



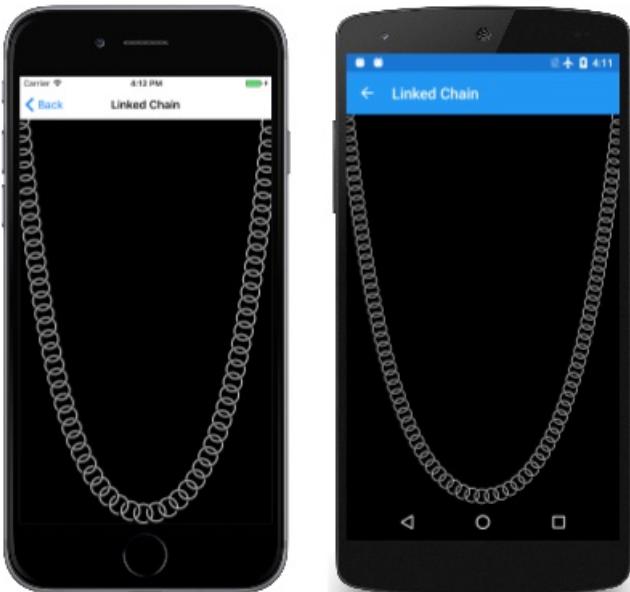
The path specified in the `SKPathEffect.Create1DPath` method is always filled. The path specified in the `DrawPath` method is always stroked if the `SKPaint` object has its `PathEffect` property set to a 1D path effect. Notice that the `pathPaint` object has no `Style` setting, which normally defaults to `Fill`, but the path is stroked regardless.

The box used in the `Translate` example is 20 pixels square, and the `advance` argument is set to 24. This difference causes a gap between the boxes when the line is roughly horizontal or vertical, but the boxes overlap a little when the line is diagonal because the diagonal of the box is 28.3 pixels.

The diamond shape in the `Rotate` example is also 20 pixels wide. The `advance` is set to 20 so that the points continue to touch as the diamond is rotated along with the curvature of the line.

The rectangle shape in the `Morph` example is 50 pixels wide with an `advance` setting of 55 to make a small gap between the rectangles as they are bent around the Bézier curve.

If the `advance` argument is less than the size of the path, then the replicated paths can overlap. This can result in some interesting effects. The **Linked Chain** page displays a series of overlapping circles that seem to resemble a linked chain, which hangs in the distinctive shape of a catenary:



Look very close and you'll see that those aren't actually circles. Each link in the chain is two arcs, sized and positioned so they seem to connect with adjoining links.

A chain or cable of uniform weight distribution hangs in the form of a catenary. An arch built in the form of an inverted catenary benefits from an equal distribution of pressure from the weight of an arch. The catenary has a seemingly simple mathematical description:

$$y = a \cdot \cosh(x / a)$$

The *cosh* is the hyperbolic cosine function. For x equal to 0, *cosh* is zero and y equals a . That's the center of the catenary. Like the *cosine* function, *cosh* is said to be *even*, which means that $\cosh(-x)$ equals $\cosh(x)$, and values increase for increasing positive or negative arguments. These values describe the curves that form the sides of the catenary.

Finding the proper value of a to fit the catenary to the dimensions of the phone's page is not a direct calculation. If w and h are the width and height of a rectangle, the optimum value of a satisfies the following equation:

$$\cosh(w / 2 / a) = 1 + h / a$$

The following method in the [LinkedChainPage](#) class incorporates that equality by referring to the two expressions on the left and right of the equal sign as `left` and `right`. For small values of a , `left` is greater than `right`; for large values of a , `left` is less than `right`. The `while` loop narrows in on an optimum value of a .

```
float FindOptimumA(float width, float height)
{
    Func<float, float> left = (float a) => (float)Math.Cosh(width / 2 / a);
    Func<float, float> right = (float a) => 1 + height / a;

    float gtA = 1;           // starting value for left > right
    float ltA = 10000;       // starting value for left < right

    while (Math.Abs(gtA - ltA) > 0.1f)
    {
        float avgA = (gtA + ltA) / 2;

        if (left(avgA) < right(avgA))
        {
            ltA = avgA;
        }
        else
        {
            gtA = avgA;
        }
    }

    return (gtA + ltA) / 2;
}
```

The `skPath` object for the links is created in the class's constructor, and the resultant `SKPathEffect` object is then set to the `PathEffect` property of the `SKPaint` object that is stored as a field:

```

public class LinkedChainPage : ContentPage
{
    const float linkRadius = 30;
    const float linkThickness = 5;

    Func<float, float, float> catenary = (float a, float x) => (float)(a * Math.Cosh(x / a));

    SKPaint linksPaint = new SKPaint
    {
        Color = SKColors.Silver
    };

    public LinkedChainPage()
    {
        Title = "Linked Chain";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;

        // Create the path for the individual links
        SKRect outer = new SKRect(-linkRadius, -linkRadius, linkRadius, linkRadius);
        SKRect inner = outer;
        inner.Inflate(-linkThickness, -linkThickness);

        using (SKPath linkPath = new SKPath())
        {
            linkPath.AddArc(outer, 55, 160);
            linkPath.ArcTo(inner, 215, -160, false);
            linkPath.Close();

            linkPath.AddArc(outer, 235, 160);
            linkPath.ArcTo(inner, 395, -160, false);
            linkPath.Close();

            // Set that path as the 1D path effect for linksPaint
            linksPaint.PathEffect =
                SKPathEffect.Create1DPath(linkPath, 1.3f * linkRadius, 0,
                                         SKPath1DPathEffectStyle.Rotate);
        }
    }
    ...
}

```

The main job of the `PaintSurface` handler is to create a path for the catenary itself. After determining the optimum a and storing it in the `optA` variable, it also needs to calculate an offset from the top of the window. Then, it can accumulate a collection of `SKPoint` values for the catenary, turn that into a path, and draw the path with the previously created `SKPaint` object:

```

public class LinkedChainPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear(SKColors.Black);

        // Width and height of catenary
        int width = info.Width;
        float height = info.Height - linkRadius;

        // Find the optimum 'a' for this width and height
        float optA = FindOptimumA(width, height);

        // Calculate the vertical offset for that value of 'a'
        float yOffset = catenary(optA, -width / 2);

        // Create a path for the catenary
        SKPoint[] points = new SKPoint[width];

        for (int x = 0; x < width; x++)
        {
            points[x] = new SKPoint(x, yOffset - catenary(optA, x - width / 2));
        }

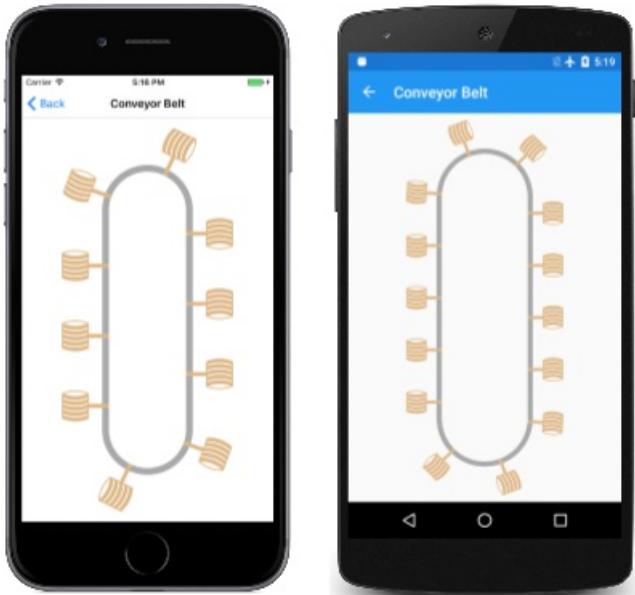
        using (SKPath path = new SKPath())
        {
            path.AddPoly(points, false);

            // And render that path with the linksPaint object
            canvas.DrawPath(path, linksPaint);
        }
    }
    ...
}

```

This program defines the path used in `Create1DPath` to have its (0, 0) point in the center. This seems reasonable because the (0, 0) point of the path is aligned with the line or curve that it's adorning. However, you can use a non-centered (0, 0) point for some special effects.

The **Conveyor Belt** page creates a path resembling an oblong conveyor belt with a curved top and bottom that is sized to the dimensions of the window. That path is stroked with a simple `SKPaint` object 20 pixels wide and colored gray, and then stroked again with another `SKPaint` object with an `SKPathEffect` object referencing a path resembling a little bucket:



The `(0, 0)` point of the bucket path is the handle, so when the `phase` argument is animated, the buckets seem to revolve around the conveyor belt, perhaps scooping up water at the bottom and dumping it out at the top.

The `ConveyorBeltPage` class implements animation with overrides of the `OnAppearing` and `OnDisappearing` methods. The path for the bucket is defined in the page's constructor:

```

public class ConveyorBeltPage : ContentPage
{
    SKCanvasView canvasView;
    bool pageIsActive = false;

    SKPaint conveyerPaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        StrokeWidth = 20,
        Color = SKColors.DarkGray
    };

    SKPath bucketPath = new SKPath();

    SKPaint bucketsPaint = new SKPaint
    {
        Color = SKColors.BurlyWood,
    };

    public ConveyorBeltPage()
    {
        Title = "Conveyor Belt";

        canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;

        // Create the path for the bucket starting with the handle
        bucketPath.AddRect(new SKRect(-5, -3, 25, 3));

        // Sides
        bucketPath.AddRoundedRect(new SKRect(25, -19, 27, 18), 10, 10,
            SKPathDirection.CounterClockwise);
        bucketPath.AddRoundedRect(new SKRect(63, -19, 65, 18), 10, 10,
            SKPathDirection.CounterClockwise);

        // Five slats
        for (int i = 0; i < 5; i++)
        {
            bucketPath.MoveTo(25, -19 + 8 * i);
            bucketPath.LineTo(25, -13 + 8 * i);
            bucketPath.ArcTo(50, 50, 0, SKPathArcSize.Small,
                SKPathDirection.CounterClockwise, 65, -13 + 8 * i);
            bucketPath.LineTo(65, -19 + 8 * i);
            bucketPath.ArcTo(50, 50, 0, SKPathArcSize.Small,
                SKPathDirection.Clockwise, 25, -19 + 8 * i);
            bucketPath.Close();
        }

        // Arc to suggest the hidden side
        bucketPath.MoveTo(25, -17);
        bucketPath.ArcTo(50, 50, 0, SKPathArcSize.Small,
            SKPathDirection.Clockwise, 65, -17);
        bucketPath.LineTo(65, -19);
        bucketPath.ArcTo(50, 50, 0, SKPathArcSize.Small,
            SKPathDirection.CounterClockwise, 25, -19);
        bucketPath.Close();

        // Make it a little bigger and correct the orientation
        bucketPath.Transform(SKMatrix.MakeScale(-2, 2));
        bucketPath.Transform(SKMatrix.MakeRotationDegrees(90));
    }
    ...
}

```

The bucket creation code completes with two transforms that make the bucket a little bigger and turn it sideways. Applying these transforms was easier than adjusting all the coordinates in the previous code.

The `PaintSurface` handler begins by defining a path for the conveyor belt itself. This is simply a pair of lines and a pair of semi-circles that are drawn with a 20-pixel-wide dark-gray line:

```
public class ConveyorBeltPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        float width = info.Width / 3;
        float verticalMargin = width / 2 + 150;

        using (SKPath conveyerPath = new SKPath())
        {
            // Straight verticals capped by semicircles on top and bottom
            conveyerPath.MoveTo(width, verticalMargin);
            conveyerPath.ArcTo(width / 2, width / 2, 0, SKPathArcSize.Large,
                SKPathDirection.Clockwise, 2 * width, verticalMargin);
            conveyerPath.LineTo(2 * width, info.Height - verticalMargin);
            conveyerPath.ArcTo(width / 2, width / 2, 0, SKPathArcSize.Large,
                SKPathDirection.Clockwise, width, info.Height - verticalMargin);
            conveyerPath.Close();

            // Draw the conveyor belt itself
            canvas.DrawPath(conveyerPath, conveyerPaint);

            // Calculate spacing based on length of conveyer path
            float length = 2 * (info.Height - 2 * verticalMargin) +
                2 * ((float)Math.PI * width / 2);

            // Value will be somewhere around 200
            float spacing = length / (float)Math.Round(length / 200);

            // Now animate the phase; t is 0 to 1 every 2 seconds
            TimeSpan timeSpan = new TimeSpan(DateTime.Now.Ticks);
            float t = (float)(timeSpan.TotalSeconds % 2 / 2);
            float phase = -t * spacing;

            // Create the buckets PathEffect
            using (SKPathEffect bucketsPathEffect =
                SKPathEffect.Create1DPath(bucketPath, spacing, phase,
                    SKPath1DPathEffectStyle.Rotate))
            {
                // Set it to the Paint object and draw the path again
                bucketsPaint.PathEffect = bucketsPathEffect;
                canvas.DrawPath(conveyerPath, bucketsPaint);
            }
        }
    }
}
```

The logic for drawing the conveyor belt does not work in landscape mode.

The buckets should be spaced about 200 pixels apart on the conveyor belt. However, the conveyor belt is probably not a multiple of 200 pixels long, which means that as the `phase` argument of `SKPathEffect.Create1DPath` is animated, buckets will pop into and out of existence.

For this reason, the program first calculates a value named `length` that is the length of the conveyor belt. Because the conveyor belt consists of straight lines and semi-circles, this is a simple calculation. Next, the

number of buckets is calculated by dividing `length` by 200. This is rounded to the nearest integer, and that number is then divided into `length`. The result is a spacing for an integral number of buckets. The `phase` argument is simply a fraction of that.

From Path to Path Again

At the bottom of the `DrawSurface` handler in **Conveyor Belt**, comment out the `canvas.DrawPath` call and replace it with the following code:

```
SKPath newPath = new SKPath();
bool fill = bucketsPaint.GetFillPath(conveyerPath, newPath);
SKPaint newPaint = new SKPaint
{
    Style = fill ? SKPaintStyle.Fill : SKPaintStyle.Stroke
};
canvas.DrawPath(newPath, newPaint);
```

As with the previous example of `GetFillPath`, you'll see that the results are the same except for the color. After executing `GetFillPath`, the `newPath` object contains multiple copies of the bucket path, each positioned in the same spot that the animation positioned them at the time of the call.

Hatching an Area

The `SKPathEffect.Create2DLines` method fills an area with parallel lines, often called *hatch lines*. The method has the following syntax:

```
public static SKPathEffect Create2DLine (Single width, SKMatrix matrix)
```

The `width` argument specifies the stroke width of the hatch lines. The `matrix` parameter is a combination of scaling and optional rotation. The scaling factor indicates the pixel increment that Skia uses to space the hatch lines. The separation between the lines is the scaling factor minus the `width` argument. If the scaling factor is less than or equal to the `width` value, there will be no space between the hatch lines, and the area will appear to be filled. Specify the same value for horizontal and vertical scaling.

By default, hatch lines are horizontal. If the `matrix` parameter contains rotation, the hatch lines are rotated clockwise.

The **Hatch Fill** page demonstrates this path effect. The `HatchFillPage` class defines three path effects as fields, the first for horizontal hatch lines with a width of 3 pixels with a scaling factor indicating that they are spaced 6 pixels apart. The separation between the lines is therefore three pixels. The second path effect is for vertical hatch lines with a width of six pixels spaced 24 pixels apart (so the separation is 18 pixels), and the third is for diagonal hatch lines 12 pixels wide spaced 36 pixels apart.

```
public class HatchFillPage : ContentPage
{
    SKPaint fillPaint = new SKPaint();

    SKPathEffect horzLinesPath = SKPathEffect.Create2DLine(3, SKMatrix.MakeScale(6, 6));

    SKPathEffect vertLinesPath = SKPathEffect.Create2DLine(6,
        Multiply(SKMatrix.MakeRotationDegrees(90), SKMatrix.MakeScale(24, 24)));

    SKPathEffect diagLinesPath = SKPathEffect.Create2DLine(12,
        Multiply(SKMatrix.MakeScale(36, 36), SKMatrix.MakeRotationDegrees(45)));

    SKPaint strokePaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        StrokeWidth = 3,
        Color = SKColors.Black
    };
    ...
    static SKMatrix Multiply(SKMatrix first, SKMatrix second)
    {
        SKMatrix target = SKMatrix.MakeIdentity();
        SKMatrix.Concat(ref target, first, second);
        return target;
    }
}
```

Notice the matrix `Multiply` method. Because the horizontal and vertical scaling factors are the same, the order in which the scaling and rotation matrices are multiplied doesn't matter.

The `PaintSurface` handler uses these three path effects with three different colors in combination with `fillPaint` to fill a rounded rectangle sized to fit the page. The `style` property set on `fillPaint` is ignored; when the `SKPaint` object includes a path effect created from `SKPathEffect.Create2DLine`, the area is filled regardless:

```

public class HatchFillPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        using (SKPath roundRectPath = new SKPath())
        {
            // Create a path
            roundRectPath.AddRoundedRect(
                new SKRect(50, 50, info.Width - 50, info.Height - 50), 100, 100);

            // Horizontal hatch marks
            fillPaint.PathEffect = horzLinesPath;
            fillPaint.Color = SKColors.Red;
            canvas.DrawPath(roundRectPath, fillPaint);

            // Vertical hatch marks
            fillPaint.PathEffect = vertLinesPath;
            fillPaint.Color = SKColors.Blue;
            canvas.DrawPath(roundRectPath, fillPaint);

            // Diagonal hatch marks -- use clipping
            fillPaint.PathEffect = diagLinesPath;
            fillPaint.Color = SKColors.Green;

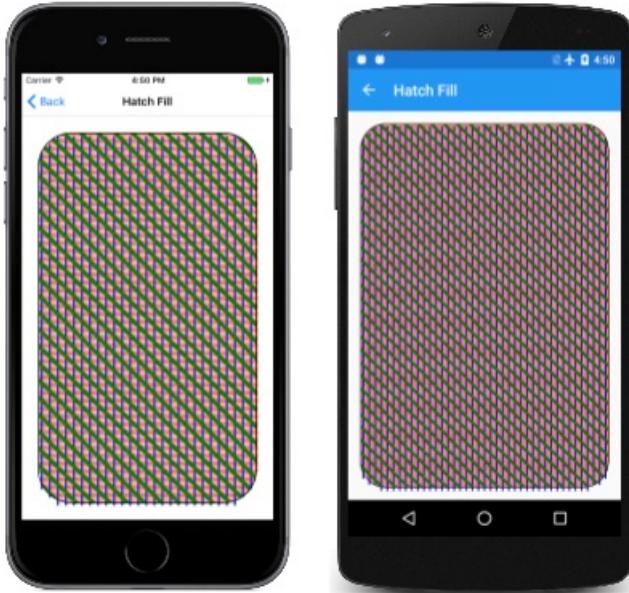
            canvas.Save();
            canvas.ClipPath(roundRectPath);
            canvas.DrawRect(new SKRect(0, 0, info.Width, info.Height), fillPaint);
            canvas.Restore();

            // Outline the path
            canvas.DrawPath(roundRectPath, strokePaint);
        }
    }
    ...
}

```

If you look carefully at the results, you'll see that the red and blue hatch lines aren't confined precisely to the rounded rectangle. (This is apparently a characteristic of the underlying Skia code.) If this is unsatisfactory, an alternative approach is shown for the diagonal hatch lines in green: The rounded rectangle is used as a clipping path and the hatch lines are drawn on the entire page.

The `PaintSurface` handler concludes with a call to simply stroke the rounded rectangle, so you can see the discrepancy with the red and blue hatch lines:



The Android screen doesn't really look like that: The scaling of the screenshot has caused the thin red lines and thin spaces to consolidate into seemingly wider red lines and wider spaces.

Filling with a Path

The `SKPathEffect.Create2DPath` allows you to fill an area with a path that is replicated horizontally and vertically, in effect tiling the area:

```
public static SKPathEffect Create2DPath (SKMatrix matrix, SKPath path)
```

The `SKMatrix` scaling factors indicate the horizontal and vertical spacing of the replicated path. But you can't rotate the path using this `matrix` argument; if you want the path rotated, rotate the path itself using the `Transform` method defined by `SKPath`.

The replicated path is normally aligned with the left and top edges of the screen rather than the area being filled. You can override this behavior by providing translation factors between 0 and the scaling factors to specify horizontal and vertical offsets from the left and top sides.

The [Path Tile Fill](#) page demonstrates this path effect. The path used for tiling the area is defined as a field in the `PathTileFillPage` class. The horizontal and vertical coordinates range from -40 to 40, which means that this path is 80 pixels square:

```

public class PathTileFillPage : ContentPage
{
    SKPath tilePath = SKPath.ParseSvgPathData(
        "M -20 -20 L 2 -20, 2 -40, 18 -40, 18 -20, 40 -20, " +
        "40 -12, 20 -12, 20 12, 40 12, 40 40, 22 40, 22 20, " +
        "-2 20, -2 40, -20 40, -20 8, -40 8, -40 -8, -20 -8 Z");
    ...

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

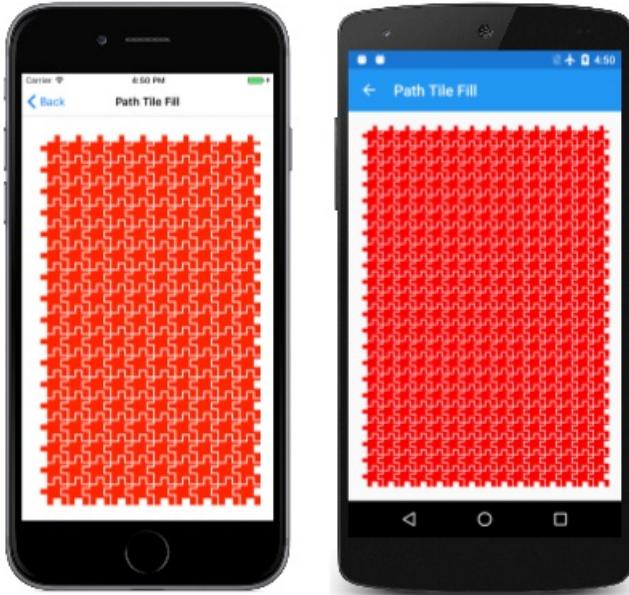
        using (SKPaint paint = new SKPaint())
        {
            paint.Color = SKColors.Red;

            using (SKPathEffect pathEffect =
                SKPathEffect.Create2DPath(SKMatrix.MakeScale(64, 64), tilePath))
            {
                paint.PathEffect = pathEffect;

                canvas.DrawRoundRect(
                    new SKRect(50, 50, info.Width - 50, info.Height - 50),
                    100, 100, paint);
            }
        }
    }
}

```

In the `PaintSurface` handler, the `SKPathEffect.Create2DPath` call sets the horizontal and vertical spacing to 64 to cause the 80-pixel square tiles to overlap. Fortunately, the path resembles a puzzle piece, meshing nicely with adjoining tiles:



The scaling from the original screenshot causes some distortion, particularly on the Android screen.

Notice that these tiles always appear whole and are never truncated. On the first two screenshots, it's not even evident that the area being filled is a rounded rectangle. If you want to truncate these tiles to a particular area, use a clipping path.

Try setting the `style` property of the `SKPaint` object to `Stroke`, and you'll see the individual tiles outlined

rather than filled.

It's also possible to fill an area with a tiled bitmap, as shown in the article [SkiaSharp bitmap tiling](#).

Rounding Sharp Corners

The [Rounded Heptagon](#) program presented in the [Three Ways to Draw an Arc](#) article used a tangent arc to curve the points of a seven-sided figure. The [Another Rounded Heptagon](#) page shows a much easier approach that uses a path effect created from the `SKPathEffect.CreateCorner` method:

```
public static SKPathEffect CreateCorner (Single radius)
```

Although the single argument is named `radius`, you must set it to half the desired corner radius. (This is a characteristic of the underlying Skia code.)

Here's the `PaintSurface` handler in the [AnotherRoundedHeptagonPage](#) class:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    int numVertices = 7;
    float radius = 0.45f * Math.Min(info.Width, info.Height);
    SKPoint[] vertices = new SKPoint[numVertices];
    double vertexAngle = -0.5f * Math.PI;           // straight up

    // Coordinates of the vertices of the polygon
    for (int vertex = 0; vertex < numVertices; vertex++)
    {
        vertices[vertex] = new SKPoint(radius * (float)Math.Cos(vertexAngle),
                                         radius * (float)Math.Sin(vertexAngle));
        vertexAngle += 2 * Math.PI / numVertices;
    }

    float cornerRadius = 100;

    // Create the path
    using (SKPath path = new SKPath())
    {
        path.AddPoly(vertices, true);

        // Render the path in the center of the screen
        using (SKPaint paint = new SKPaint())
        {
            paint.Style = SKPaintStyle.Stroke;
            paint.Color = SKColors.Blue;
            paint.StrokeWidth = 10;

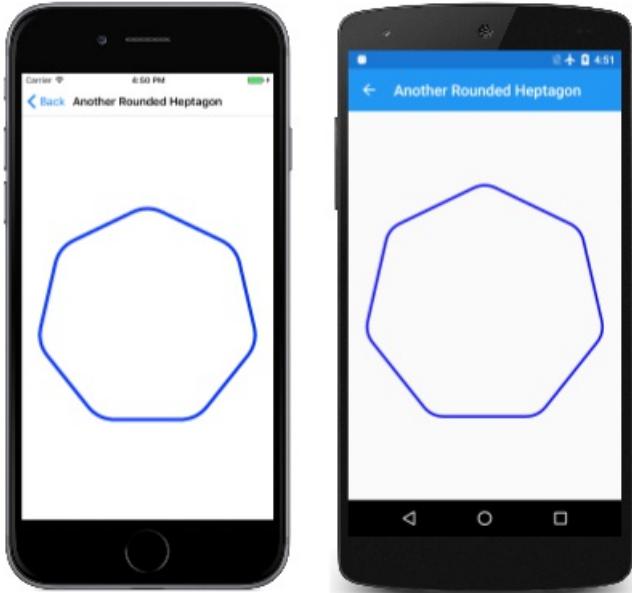
            // Set argument to half the desired corner radius!
            paint.PathEffect = SKPathEffect.CreateCorner(cornerRadius / 2);

            canvas.Translate(info.Width / 2, info.Height / 2);
            canvas.DrawPath(path, paint);

            // Uncomment DrawCircle call to verify corner radius
            float offset = cornerRadius / (float)Math.Sin(Math.PI * (numVertices - 2) / numVertices / 2);
            paint.Color = SKColors.Green;
            // canvas.DrawCircle(vertices[0].X, vertices[0].Y + offset, cornerRadius, paint);
        }
    }
}
}

```

You can use this effect with either stroking or filling based on the `style` property of the `SKPaint` object. Here it is running:



You'll see that this rounded heptagon is identical to the earlier program. If you need more convincing that the corner radius is truly 100 rather than the 50 specified in the `SKPathEffect.CreateCorner` call, you can uncomment the final statement in the program and see a 100-radius circle superimposed on the corner.

Random Jitter

Sometimes the flawless straight lines of computer graphics are not quite what you want, and a little randomness is desired. In that case, you'll want to try the `SKPathEffect.CreateDiscrete` method:

```
public static SKPathEffect CreateDiscrete (Single segLength, Single deviation, UInt32 seedAssist)
```

You can use this path effect for either stroking or filling. Lines are separated into connected segments — the approximate length of which is specified by `segLength` — and extend in different directions. The extent of the deviation from the original line is specified by `deviation`.

The final argument is a seed used to generate the pseudo-random sequence used for the effect. The jitter effect will look a little different for different seeds. The argument has a default value of zero, which means that the effect is the same whenever you run the program. If you want different jitter whenever the screen is repainted, you can set the seed to the `Millisecond` property of a `DateTime.Now` value (for example).

The **Jitter Experiment** page allows you to experiment with different values in stroking a rectangle:



The program is straightforward. The `JitterExperimentPage.xaml` file instantiates two `Slider` elements and an `SKCanvasView`:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Curves.JitterExperimentPage"
    Title="Jitter Experiment">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

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

                <Style TargetType="Slider">
                    <Setter Property="Margin" Value="20, 0" />
                    <Setter Property="Minimum" Value="0" />
                    <Setter Property="Maximum" Value="100" />
                </Style>
            </ResourceDictionary>
        </Grid.Resources>

        <Slider x:Name="segLengthSlider"
            Grid.Row="0"
            ValueChanged="sliderValueChanged" />

        <Label Text="{Binding Source={x:Reference segLengthSlider},
            Path=Value,
            StringFormat='Segment Length = {0:F0}'}"
            Grid.Row="1" />

        <Slider x:Name="deviationSlider"
            Grid.Row="2"
            ValueChanged="sliderValueChanged" />

        <Label Text="{Binding Source={x:Reference deviationSlider},
            Path=Value,
            StringFormat='Deviation = {0:F0}'}"
            Grid.Row="3" />

        <skia:SKCanvasView x:Name="canvasView"
            Grid.Row="4"
            PaintSurface="OnCanvasViewPaintSurface" />
    </Grid>
</ContentPage>

```

The `PaintSurface` handler in the `JitterExperimentPage.xaml.cs` code-behind file is called whenever a `Slider` value changes. It calls `SKPathEffect.CreateDiscrete` using the two `Slider` values and uses that to stroke a rectangle:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    float segLength = (float)segLengthSlider.Value;
    float deviation = (float)deviationSlider.Value;

    using (SKPaint paint = new SKPaint())
    {
        paint.Style = SKPaintStyle.Stroke;
        paint.StrokeWidth = 5;
        paint.Color = SKColors.Blue;

        using (SKPathEffect pathEffect = SKPathEffect.CreateDiscrete(segLength, deviation))
        {
            paint.PathEffect = pathEffect;

            SKRect rect = new SKRect(100, 100, info.Width - 100, info.Height - 100);
            canvas.DrawRect(rect, paint);
        }
    }
}

```

You can use this effect for filling as well, in which case the outline of the filled area is subject to these random deviations. The [Jitter Text](#) page demonstrates using this path effect to display text. Most of the code in the `PaintSurface` handler of the [JitterTextPage](#) class is devoted to sizing and centering the text:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    string text = "FUZZY";

    using (SKPaint textPaint = new SKPaint())
    {
        textPaint.Color = SKColors.Purple;
        textPaint.PathEffect = SKPathEffect.CreateDiscrete(3f, 10f);

        // Adjust TextSize property so text is 95% of screen width
        float textWidth = textPaint.MeasureText(text);
        textPaint.TextSize *= 0.95f * info.Width / textWidth;

        // Find the text bounds
        SKRect textBounds = new SKRect();
        textPaint.MeasureText(text, ref textBounds);

        // Calculate offsets to center the text on the screen
        float xText = info.Width / 2 - textBounds.MidX;
        float yText = info.Height / 2 - textBounds.MidY;

        canvas.DrawText(text, xText, yText, textPaint);
    }
}

```

Here it is running in landscape mode:



Path Outlining

You've already seen two little examples of the `GetFillPath` method of `SKPaint`, which exists two versions:

```
public Boolean GetFillPath (SKPath src, SKPath dst, Single resScale = 1)  
public Boolean GetFillPath (SKPath src, SKPath dst, SKRect cullRect, Single resScale = 1)
```

Only the first two arguments are required. The method accesses the path referenced by the `src` argument, modifies the path data based on the stroke properties in the `SKPaint` object (including the `PathEffect` property), and then writes the results into the `dst` path. The `resScale` parameter allows reducing the precision to create a smaller destination path, and the `cullRect` argument can eliminate contours outside a rectangle.

One basic use of this method does not involve path effects at all: If the `SKPaint` object has its `Style` property set to `SKPaintStyle.Stroke`, and does *not* have its `PathEffect` set, then `GetFillPath` creates a path that represents an *outline* of the source path as if it had been stroked by the paint properties.

For example, if the `src` path is a simple circle of radius 500, and the `SKPaint` object specifies a stroke width of 100, then the `dst` path becomes two concentric circles, one with a radius of 450 and the other with a radius of 550. The method is called `GetFillPath` because filling this `dst` path is the same as stroking the `src` path. But you can also stroke the `dst` path to see the path outlines.

The **Tap to Outline the Path** demonstrates this. The `SKCanvasView` and `TapGestureRecognizer` are instantiated in the `TapToOutlineThePathPage.xaml` file. The `TapToOutlineThePathPage.xaml.cs` code-behind file defines three `SKPaint` objects as fields, two for stroking with stroke widths of 100 and 20, and the third for filling:

```
public partial class TapToOutlineThePathPage : ContentPage
{
    bool outlineThePath = false;

    SKPaint redThickStroke = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Red,
        StrokeWidth = 100
    };

    SKPaint redThinStroke = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Red,
        StrokeWidth = 20
    };

    SKPaint blueFill = new SKPaint
    {
        Style = SKPaintStyle.Fill,
        Color = SKColors.Blue
    };

    public TapToOutlineThePathPage()
    {
        InitializeComponent();
    }

    void OnCanvasViewTapped(object sender, EventArgs args)
    {
        outlineThePath ^= true;
        (sender as SKCanvasView).InvalidateSurface();
    }
    ...
}
```

If the screen has not been tapped, the `PaintSurface` handler uses the `blueFill` and `redThickStroke` paint objects to render a circular path:

```

public partial class TapToOutlineThePathPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

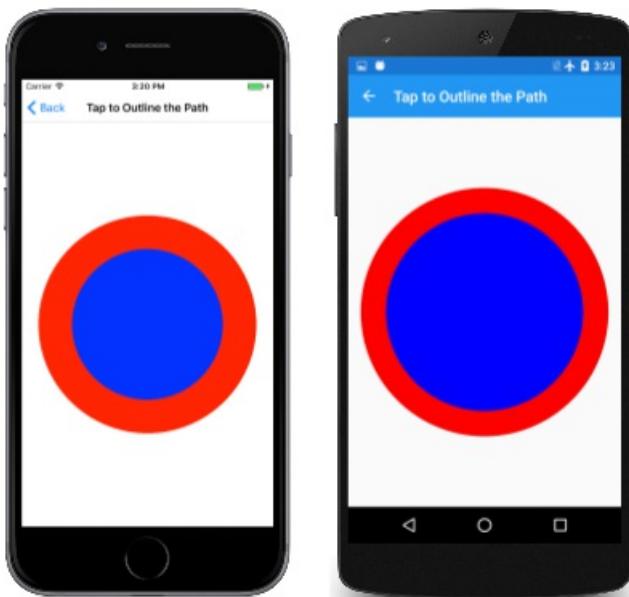
        using (SKPath circlePath = new SKPath())
        {
            circlePath.AddCircle(info.Width / 2, info.Height / 2,
                Math.Min(info.Width / 2, info.Height / 2) -
                redThickStroke.StrokeWidth);

            if (!outlineThePath)
            {
                canvas.DrawPath(circlePath, blueFill);
                canvas.DrawPath(circlePath, redThickStroke);
            }
            else
            {
                using (SKPath outlinePath = new SKPath())
                {
                    redThickStroke.GetFillPath(circlePath, outlinePath);

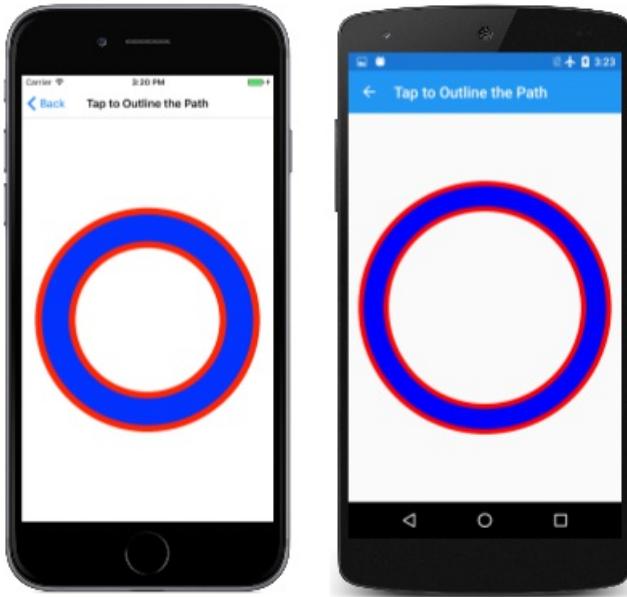
                    canvas.DrawPath(outlinePath, blueFill);
                    canvas.DrawPath(outlinePath, redThinStroke);
                }
            }
        }
    }
}

```

The circle is filled and stroked as you'd expect:



When you tap the screen, `outlineThePath` is set to `true`, and the `PaintSurface` handler creates a fresh `SKPath` object and uses that as the destination path in a call to `GetFillPath` on the `redThickStroke` paint object. That destination path is then filled and stroked with `redThinStroke`, resulting in the following:



The two red circles clearly indicate that the original circular path has been converted into two circular contours.

This method can be very useful in developing paths to use for the `SKPathEffect.Create1DPath` method. The paths you specify in these methods are always filled when the paths are replicated. If you don't want the entire path to be filled, you must carefully define the outlines.

For example, in the **Linked Chain** sample, the links were defined with a series of four arcs, each pair of which were based on two radii to outline the area of the path to be filled. It's possible to replace the code in the `LinkedChainPage` class to do it a little differently.

First, you'll want to redefine the `linkRadius` constant:

```
const float linkRadius = 27.5f;
const float linkThickness = 5;
```

The `linkPath` is now just two arcs based on that single radius, with the desired start angles and sweep angles:

```
using (SKPath linkPath = new SKPath())
{
    SKRect rect = new SKRect(-linkRadius, -linkRadius, linkRadius, linkRadius);
    linkPath.AddArc(rect, 55, 160);
    linkPath.AddArc(rect, 235, 160);

    using (SKPaint strokePaint = new SKPaint())
    {
        strokePaint.Style = SKPaintStyle.Stroke;
        strokePaint.StrokeWidth = linkThickness;

        using (SKPath outlinePath = new SKPath())
        {
            strokePaint.GetFillPath(linkPath, outlinePath);

            // Set that path as the 1D path effect for linksPaint
            linksPaint.PathEffect =
                SKPathEffect.Create1DPath(outlinePath, 1.3f * linkRadius, 0,
                                         SKPath1DPathEffectStyle.Rotate);
        }
    }
}
```

The `outlinePath` object is then the recipient of the outline of `linkPath` when it is stroked with the properties specified in `strokePaint`.

Another example using this technique is coming up next for the path used in a method.

Combining Path Effects

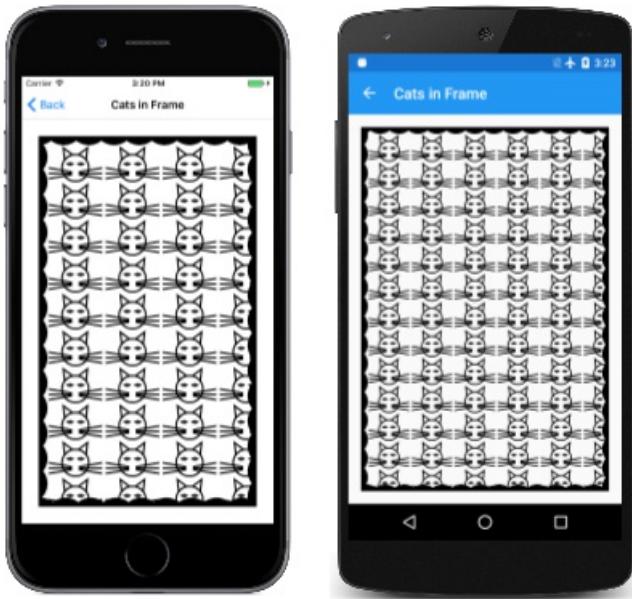
The two final static creation methods of `SKPathEffect` are `SKPathEffect.CreateSum` and `SKPathEffect.CreateCompose`:

```
public static SKPathEffect CreateSum (SKPathEffect first, SKPathEffect second)  
  
public static SKPathEffect CreateCompose (SKPathEffect outer, SKPathEffect inner)
```

Both these methods combine two path effects to create a composite path effect. The `CreateSum` method creates a path effect that is similar to the two path effects applied separately, while `CreateCompose` applies one path effect (the `inner`) and then applies the `outer` to that.

You've already seen how the `GetFillPath` method of `SKPaint` can convert one path to another path based on `SKPaint` properties (including `PathEffect`) so it shouldn't be *too* mysterious how an `SKPaint` object can perform that operation twice with the two path effects specified in the `CreateSum` or `CreateCompose` methods.

One obvious use of `CreateSum` is to define an `SKPaint` object that fills a path with one path effect, and strokes the path with another path effect. This is demonstrated in the **Cats in Frame** sample, which displays an array of cats within a frame with scalloped edges:



The `CatsInFramePage` class begins by defining several fields. You might recognize the first field from the `PathDataCatPage` class from the **SVG Path Data** article. The second path is based on a line and arc for the scallop pattern of the frame:

```

public class CatsInFramePage : ContentPage
{
    // From PathDataCatPage.cs
    SKPath catPath = SKPath.ParseSvgPathData(
        "M 160 140 L 150 50 220 103" +           // Left ear
        "M 320 140 L 330 50 260 103" +           // Right ear
        "M 215 230 L 40 200" +                   // Left whiskers
        "M 215 240 L 40 240" +
        "M 215 250 L 40 280" +
        "M 265 230 L 440 200" +                 // Right whiskers
        "M 265 240 L 440 240" +
        "M 265 250 L 440 280" +
        "M 240 100" +                           // Head
        "A 100 100 0 0 1 240 300" +
        "A 100 100 0 0 1 240 100 Z" +
        "M 180 170" +                           // Left eye
        "A 40 40 0 0 1 220 170" +
        "A 40 40 0 0 1 180 170 Z" +
        "M 300 170" +                           // Right eye
        "A 40 40 0 0 1 260 170" +
        "A 40 40 0 0 1 300 170 Z");
}

SKPaint catStroke = new SKPaint
{
    Style = SKPaintStyle.Stroke,
    StrokeWidth = 5
};

SKPath scallopPath =
    SKPath.ParseSvgPathData("M 0 0 L 50 0 A 60 60 0 0 1 -50 0 Z");

SKPaint framePaint = new SKPaint
{
    Color = SKColors.Black
};
...
}

```

The `catPath` could be used in the `SKPathEffect.Create2DPath` method if the `SKPaint` object `Style` property is set to `Stroke`. However, if the `catPath` is used directly in this program, then the entire head of the cat will be filled, and the whiskers won't even be visible. (Try it!) It's necessary to obtain the outline of that path and use that outline in the `SKPathEffect.Create2DPath` method.

The constructor does this job. It first applies two transforms to `catPath` to move the $(0, 0)$ point to the center and scale it down in size. `GetFillPath` obtains all the outlines of the contours in `outlinedCatPath`, and that object is used in the `SKPathEffect.Create2DPath` call. The scaling factors in the `SKMatrix` value are slightly larger than the horizontal and vertical size of the cat to provide a little buffer between the tiles, while the translation factors were derived somewhat empirically so that a full cat is visible in the upper-left corner of the frame:

```

public class CatsInFramePage : ContentPage
{
    ...
    public CatsInFramePage()
    {
        Title = "Cats in Frame";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;

        // Move (0, 0) point to center of cat path
        catPath.Transform(SKMatrix.MakeTranslation(-240, -175));

        // Now catPath is 400 by 250
        // Scale it down to 160 by 100
        catPath.Transform(SKMatrix.MakeScale(0.40f, 0.40f));

        // Get the outlines of the contours of the cat path
        SKPath outlinedCatPath = new SKPath();
        catStroke.GetFillPath(catPath, outlinedCatPath);

        // Create a 2D path effect from those outlines
        SKPathEffect fillEffect = SKPathEffect.Create2DPath(
            new SKMatrix { ScaleX = 170, ScaleY = 110,
                TransX = 75, TransY = 80,
                Persp2 = 1 },
            outlinedCatPath);

        // Create a 1D path effect from the scallop path
        SKPathEffect strokeEffect =
            SKPathEffect.Create1DPath(scallopPath, 75, 0, SKPath1DPathEffectStyle.Rotate);

        // Set the sum the effects to frame paint
        framePaint.PathEffect = SKPathEffect.CreateSum(fillEffect, strokeEffect);
    }
    ...
}

```

The constructor then calls `SKPathEffect.Create1DPath` for the scalloped frame. Notice that the width of the path is 100 pixels, but the advance is 75 pixels so that the replicated path is overlapped around the frame. The final statement of the constructor calls `SKPathEffect.CreateSum` to combine the two path effects and set the result to the `SKPaint` object.

All this work allows the `PaintSurface` handler to be quite simple. It only needs to define a rectangle and draw it using `framePaint`:

```

public class CatsInFramePage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        SKRect rect = new SKRect(50, 50, info.Width - 50, info.Height - 50);
        canvas.ClipRect(rect);
        canvas.DrawRect(rect, framePaint);
    }
}

```

The algorithms behind the path effects always cause the whole path used for stroking or filling to be displayed, which can cause some visuals to appear outside the rectangle. The `ClipRect` call prior to the `DrawRect` call allows the visuals to be considerably cleaner. (Try it without clipping!)

It is common to use `SKPathEffect.CreateCompose` to add some jitter to another path effect. You can certainly experiment on your own, but here's a somewhat different example:

The **Dashed Hatch Lines** fills an ellipse with hatch lines that are dashed. Most of the work in the `DashedHatchLinesPage` class is performed right in the field definitions. These fields define a dash effect and a hatch effect. They are defined as `static` because they are then referenced in an `SKPathEffect.CreateCompose` call in the `SKPaint` definition:

```
public class DashedHatchLinesPage : ContentPage
{
    static SKPathEffect dashEffect =
        SKPathEffect.CreateDash(new float[] { 30, 30 }, 0);

    static SKPathEffect hatchEffect = SKPathEffect.Create2DLine(20,
        Multiply(SKMatrix.MakeScale(60, 60),
            SKMatrix.MakeRotationDegrees(45)));

    SKPaint paint = new SKPaint()
    {
        PathEffect = SKPathEffect.CreateCompose(dashEffect, hatchEffect),
        StrokeCap = SKStrokeCap.Round,
        Color = SKColors.Blue
    };
    ...

    static SKMatrix Multiply(SKMatrix first, SKMatrix second)
    {
        SKMatrix target = SKMatrix.MakeIdentity();
        SKMatrix.Concat(ref target, first, second);
        return target;
    }
}
```

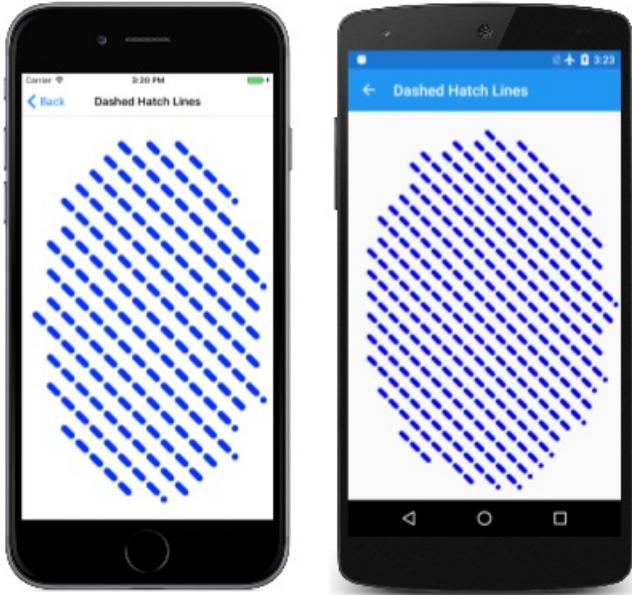
The `PaintSurface` handler needs to contain only the standard overhead plus one call to `DrawOval`:

```
public class DashedHatchLinesPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        canvas.DrawOval(info.Width / 2, info.Height / 2,
            0.45f * info.Width, 0.45f * info.Height,
            paint);
    }
    ...
}
```

As you've already discovered, the hatch lines aren't precisely restricted to the interior of the area, and in this example, they always begin at the left with a whole dash:



Now that you've seen path effects that range from simple dots and dashes to strange combinations, use your imagination and see what you can create.

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Paths and Text in SkiaSharp

3/5/2021 • 9 minutes to read • [Edit Online](#)

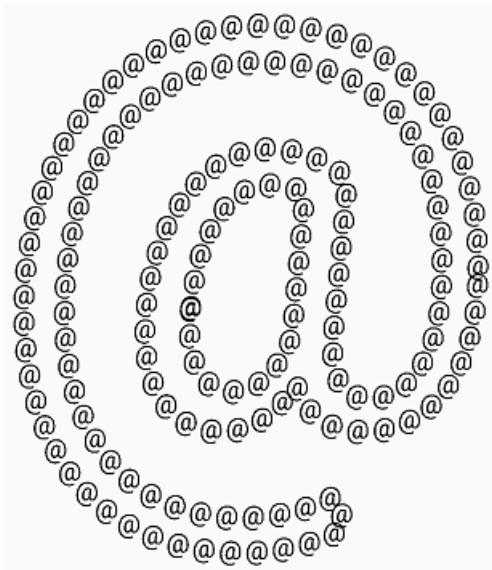
 [Download the sample](#)

Explore the intersection of paths and text

In modern graphics systems, text fonts are collections of character outlines, usually defined by quadratic Bézier curves. Consequently, many modern graphics systems include a facility to convert text characters into a graphics path.

You've already seen that you can stroke the outlines of text characters as well as fill them. This allows you to display these character outlines with a particular stroke width and even a path effect as described in the [Path Effects](#) article. But it is also possible to convert a character string into an `SKPath` object. This means that text outlines can be used for clipping with techniques that were described in the [Clipping with Paths and Regions](#) article.

Besides using a path effect to stroke a character outline, you can also create path effects that are based on a path that is derived from a character string, and you can even combine the two effects:



In the previous article on [Path Effects](#), you saw how the `GetFillPath` method of `SKPaint` can obtain an outline of a stroked path. You can also use this method with paths derived from character outlines.

Finally, this article demonstrates another intersection of paths and text: The `DrawTextOnPath` method of `SKCanvas` allows you to display a text string so that the baseline of the text follows a curved path.

Text to Path Conversion

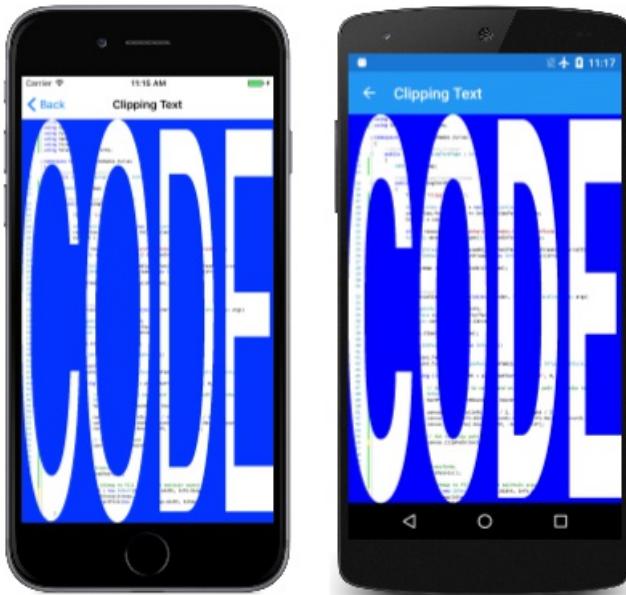
The `GetTextPath` method of `SKPaint` converts a character string to an `SKPath` object:

```
public SKPath GetTextPath (String text, Single x, Single y)
```

The `x` and `y` arguments indicate the starting point of the baseline of the left side of the text. They play the same role here as in the `DrawText` method of `SKCanvas`. Within the path, the baseline of the left side of the text will have the coordinates (x, y) .

The `GetTextPath` method is overkill if you merely want to fill or stroke the resultant path. The normal `DrawText` method allows you to do that. The `GetTextPath` method is more useful for other tasks involving paths.

One of these tasks is clipping. The **Clipping Text** page creates a clipping path based on the character outlines of the word "CODE." This path is stretched to the size of the page to clip a bitmap that contains an image of the **Clipping Text** source code:



The `clippingTextPage` class constructor loads the bitmap that is stored as an embedded resource in the **Media** folder of the solution:

```
public class ClippingTextPage : ContentPage
{
    SKBitmap bitmap;

    public ClippingTextPage()
    {
        Title = "Clipping Text";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;

        string resourceId = "SkiaSharpFormsDemos.Media.PageOfCode.png";
        Assembly assembly = GetType().GetTypeInfo().Assembly;

        using (Stream stream = assembly.GetManifestResourceStream(resourceId))
        {
            bitmap = SKBitmap.Decode(stream);
        }
    }
}
```

The `PaintSurface` handler begins by creating an `SKPaint` object suitable for text. The `Typeface` property is set as well as the `TextSize`, although for this particular application the `TextSize` property is purely arbitrary. Also notice there is no `Style` setting.

The `TextSize` and `Style` property settings are not necessary because this `SKPaint` object is used solely for the `GetTextPath` call using the text string "CODE". The handler then measures the resultant `SKPath` object and applies three transforms to center it and scale it to the size of the page. The path can then be set as the clipping path:

```

public class ClippingTextPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear(SKColors.Blue);

        using (SKPaint paint = new SKPaint())
        {
            paint.Typeface = SKTypeface.FromFamilyName(null, SKTypefaceStyle.Bold);
            paint.TextSize = 10;

            using (SKPath textPath = paint.GetTextPath("CODE", 0, 0))
            {
                // Set transform to center and enlarge clip path to window height
                SKRect bounds;
                textPath.GetTightBounds(out bounds);

                canvas.Translate(info.Width / 2, info.Height / 2);
                canvas.Scale(info.Width / bounds.Width, info.Height / bounds.Height);
                canvas.Translate(-bounds.MidX, -bounds.MidY);

                // Set the clip path
                canvas.ClipPath(textPath);
            }
        }

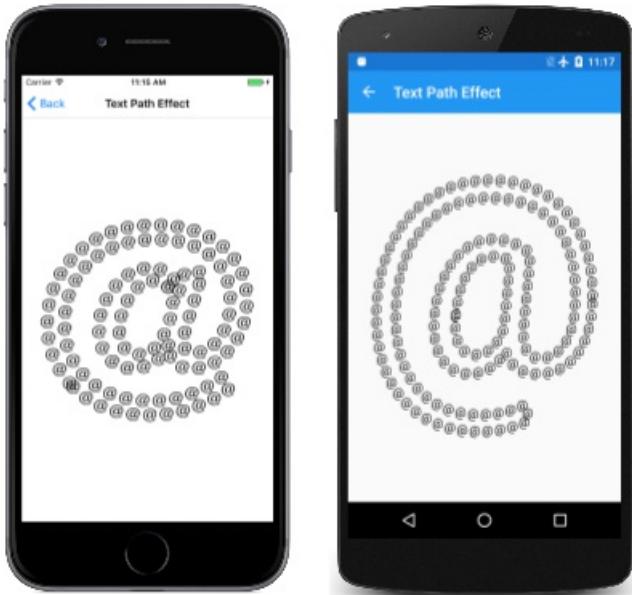
        // Reset transforms
        canvas.ResetMatrix();

        // Display bitmap to fill window but maintain aspect ratio
        SKRect rect = new SKRect(0, 0, info.Width, info.Height);
        canvas.DrawBitmap(bitmap,
            rect.AAspectFill(new SKSize(bitmap.Width, bitmap.Height)));
    }
}

```

Once the clipping path is set, the bitmap can be displayed, and it will be clipped to the character outlines. Notice the use of the `AspectFill` method of `SKRect` that calculates a rectangle for filling the page while preserving the aspect ratio.

The **Text Path Effect** page converts a single ampersand character to a path to create a 1D path effect. A paint object with this path effect is then used to stroke the outline of a larger version of that same character:



Much of the work in the `TextPathEffectPath` class occurs in the fields and constructor. The two `SKPaint` objects defined as fields are used for two different purposes: The first (named `textPathPaint`) is used to convert the ampersand with a `TextSize` of 50 to a path for the 1D path effect. The second (`textPaint`) is used to display the larger version of the ampersand with that path effect. For that reason, the `style` of this second paint object is set to `Stroke`, but the `StrokeWidth` property is not set because that property isn't necessary when using a 1D path effect:

```

public class TextPathEffectPage : ContentPage
{
    const string character = "@";
    const float littleSize = 50;

    SKPathEffect pathEffect;

    SKPaint textPathPaint = new SKPaint
    {
        TextSize = littleSize
    };

    SKPaint textPaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Black
    };

    public TextPathEffectPage()
    {
        Title = "Text Path Effect";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;

        // Get the bounds of textPathPaint
        SKRect textPathPaintBounds = new SKRect();
        textPathPaint.MeasureText(character, ref textPathPaintBounds);

        // Create textPath centered around (0, 0)
        SKPath textPath = textPathPaint.GetTextPath(character,
            -textPathPaintBounds.MidX,
            -textPathPaintBounds.MidY);

        // Create the path effect
        pathEffect = SKPathEffect.Create1DPath(textPath, littleSize, 0,
            SKPath1DPathEffectStyle.Translate);
    }
    ...
}

```

The constructor first uses the `textPathPaint` object to measure the ampersand with a `TextSize` of 50. The negatives of the center coordinates of that rectangle are then passed to the `GetTextPath` method to convert the text to a path. The resultant path has the (0, 0) point in the center of the character, which is ideal for a 1D path effect.

You might think that the `skPathEffect` object created at the end of the constructor could be set to the `PathEffect` property of `textPaint` rather than saved as a field. But this turned out not to work very well because it distorted the results of the `MeasureText` call in the `PaintSurface` handler:

```

public class TextPathEffectPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Set textPaint TextSize based on screen size
        textPaint.TextSize = Math.Min(info.Width, info.Height);

        // Do not measure the text with PathEffect set!
        SKRect textBounds = new SKRect();
        textPaint.MeasureText(character, ref textBounds);

        // Coordinates to center text on screen
        float xText = info.Width / 2 - textBounds.MidX;
        float yText = info.Height / 2 - textBounds.MidY;

        // Set the PathEffect property and display text
        textPaint.PathEffect = pathEffect;
        canvas.DrawText(character, xText, yText, textPaint);
    }
}

```

That `MeasureText` call is used to center the character on the page. To avoid problems, the `PathEffect` property is set to the paint object after the text has been measured but before it is displayed.

Outlines of Character Outlines

Normally the `GetFillPath` method of `SKPaint` converts one path to another by applying paint properties, most notably the stroke width and path effect. When used without path effects, `GetFillPath` effectively creates a path that outlines another path. This was demonstrated in the [Tap to Outline the Path](#) page in the [Path Effects](#) article.

You can also call `GetFillPath` on the path returned from `GetTextPath` but at first you might not be entirely sure what that would look like.

The [Character Outline Outlines](#) page demonstrates the technique. All the relevant code is in the `PaintSurface` handler of the [CharacterOutlineOutlinesPage](#) class.

The constructor begins by creating an `SKPaint` object named `textPaint` with a `TextSize` property based on the size of the page. This is converted to a path using the `GetTextPath` method. The coordinate arguments to `GetTextPath` effectively center the path on the screen:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    using (SKPaint textPaint = new SKPaint())
    {
        // Set Style for the character outlines
        textPaint.Style = SKPaintStyle.Stroke;

        // Set TextSize based on screen size
        textPaint.TextSize = Math.Min(info.Width, info.Height);

        // Measure the text
        SKRect textBounds = new SKRect();
        textPaint.MeasureText("@", ref textBounds);

        // Coordinates to center text on screen
        float xText = info.Width / 2 - textBounds.MidX;
        float yText = info.Height / 2 - textBounds.MidY;

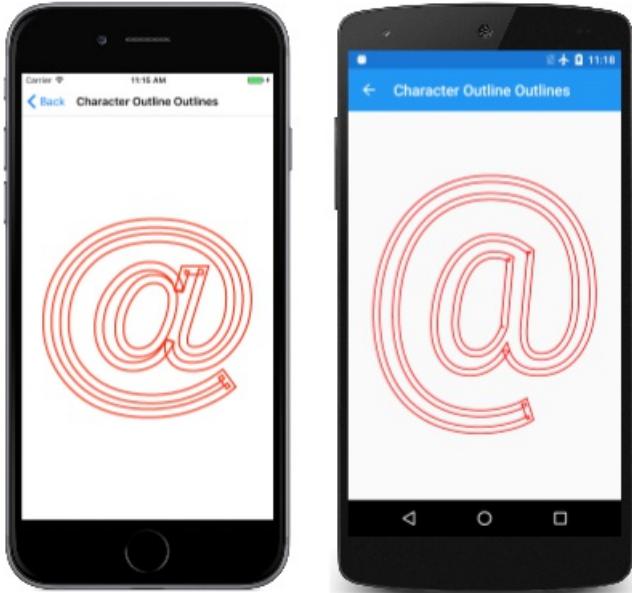
        // Get the path for the character outlines
        using (SKPath textPath = textPaint.GetTextPath("@", xText, yText))
        {
            // Create a new path for the outlines of the path
            using (SKPath outlinePath = new SKPath())
            {
                // Convert the path to the outlines of the stroked path
                textPaint.StrokeWidth = 25;
                textPaint.GetFillPath(textPath, outlinePath);

                // Stroke that new path
                using (SKPaint outlinePaint = new SKPaint())
                {
                    outlinePaint.Style = SKPaintStyle.Stroke;
                    outlinePaint.StrokeWidth = 5;
                    outlinePaint.Color = SKColors.Red;

                    canvas.DrawPath(outlinePath, outlinePaint);
                }
            }
        }
    }
}

```

The `PaintSurface` handler then creates a new path named `outlinePath`. This becomes the destination path in the call to `GetFillPath`. The `StrokeWidth` property of 25 causes `outlinePath` to describe the outline of a 25-pixel-wide path stroking the text characters. This path is then displayed in red with a stroke width of 5:



Look closely and you'll see overlaps where the path outline makes a sharp corner. These are normal artifacts of this process.

Text Along a Path

Text is normally displayed on a horizontal baseline. Text can be rotated to run vertically or diagonally, but the baseline is still a straight line.

There are times, however, when you want text to run along a curve. This is the purpose of the [DrawTextOnPath](#) method of [SKCanvas](#) :

```
public Void DrawTextOnPath (String text, SKPath path, Single hOffset, Single vOffset, SKPaint paint)
```

The text specified in the first argument is made to run along the path specified as the second argument. You can begin the text at an offset from the beginning of the path with the [hOffset](#) argument. Normally the path forms the baseline of the text: Text ascenders are on one side of the path, and text descenders are on the other. But you can offset the text baseline from the path with the [vOffset](#) argument.

This method has no facility to provide guidance on setting the [TextSize](#) property of [SKPaint](#) to make the text sized perfectly to run from the beginning of the path to the end. Sometimes you can figure out that text size on your own. Other times you'll need to use path-measuring functions to be described in the next article on [Path Information and Enumeration](#).

The [Circular Text](#) program wraps text around a circle. It's easy to determine the circumference of a circle, so it's easy to size the text to fit exactly. The [PaintSurface](#) handler of the [CircularTextPage](#) class calculates a radius of a circle based on the size of the page. That circle becomes [circularPath](#) :

```

public class CircularTextPage : ContentPage
{
    const string text = "xt in a circle that shapes the te";
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

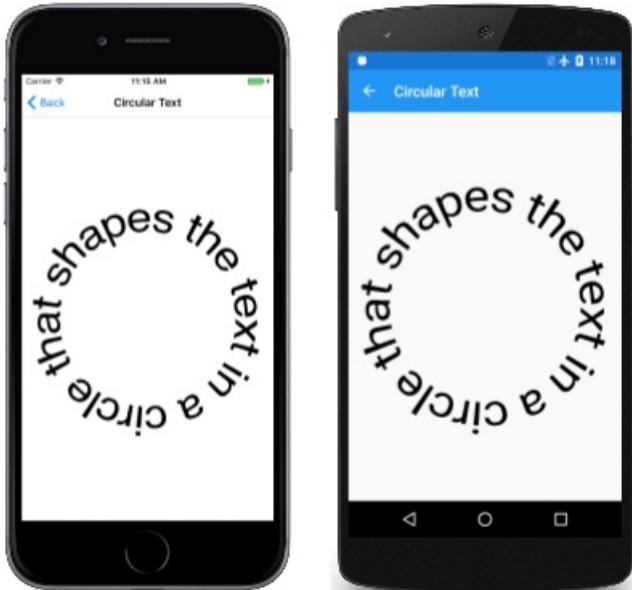
        using (SKPath circularPath = new SKPath())
        {
            float radius = 0.35f * Math.Min(info.Width, info.Height);
            circularPath.AddCircle(info.Width / 2, info.Height / 2, radius);

            using (SKPaint textPaint = new SKPaint())
            {
                textPaint.TextSize = 100;
                float textWidth = textPaint.MeasureText(text);
                textPaint.TextSize *= 2 * 3.14f * radius / textWidth;

                canvas.DrawTextOnPath(text, circularPath, 0, 0, textPaint);
            }
        }
    }
}

```

The `TextSize` property of `textPaint` is then adjusted so that the text width matches the circumference of the circle:



The text itself was chosen to be somewhat circular as well: The word "circle" is both the subject of the sentence and the object of a prepositional phrase.

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Path Information and Enumeration

3/5/2021 • 14 minutes to read • [Edit Online](#)

 [Download the sample](#)

Get information about paths and enumerate the contents

The `SKPath` class defines several properties and methods that allow you to obtain information about the path.

The `Bounds` and `TightBounds` properties (and related methods) obtain the metrical dimensions of a path. The `Contains` method lets you determine if a particular point is within a path.

It is sometimes useful to determine the total length of all the lines and curves that make up a path. Calculating this length is not an algorithmically simple task, so an entire class named `PathMeasure` is devoted to it.

It is also sometimes useful to obtain all the drawing operations and points that make up a path. At first, this facility might seem unnecessary: If your program has created the path, the program already knows the contents. However, you've seen that paths can also be created by [path effects](#) and by converting [text strings into paths](#). You can also obtain all the drawing operations and points that make up these paths. One possibility is to apply an algorithmic transform to all the points, for example, to wrap text around a hemisphere:



Getting the Path Length

In the article [Paths and Text](#) you saw how to use the `DrawTextOnPath` method to draw a text string whose baseline follows the course of a path. But what if you want to size the text so that it fits the path precisely? Drawing text around a circle is easy because the circumference of a circle is simple to calculate. But the circumference of an ellipse or the length of a Bézier curve is not so simple.

The `SKPathMeasure` class can help. The [constructor](#) accepts an `SKPath` argument, and the `Length` property reveals its length.

This class is demonstrated in the [Path Length](#) sample, which is based on the [Bezier Curve](#) page. The `PathLengthPage.xaml` file derives from `InteractivePage` and includes a touch interface:

```

<local:InteractivePage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:SkiaSharpFormsDemos"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    xmlns:tt="clr-namespace:TouchTracking"
    x:Class="SkiaSharpFormsDemos.Curves.PathLengthPage"
    Title="Path Length">
    <Grid BackgroundColor="White">
        <skia:SKCanvasView x:Name="canvasView"
            PaintSurface="OnCanvasViewPaintSurface" />
        <Grid.Effects>
            <tt:TouchEffect Capture="True"
                TouchAction="OnTouchEffectAction" />
        </Grid.Effects>
    </Grid>
</local:InteractivePage>

```

The [PathLengthPage.xaml.cs](#) code-behind file allows you to move four touch points to define the end points and control points of a cubic Bézier curve. Three fields define a text string, an `SKPaint` object, and a calculated width of the text:

```

public partial class PathLengthPage : InteractivePage
{
    const string text = "Compute length of path";

    static SKPaint textPaint = new SKPaint
    {
        Style = SKPaintStyle.Fill,
        Color = SKColors.Black,
        TextSize = 10,
    };

    static readonly float baseTextWidth = textPaint.MeasureText(text);
    ...
}

```

The `baseTextWidth` field is the width of the text based on a `TextSize` setting of 10.

The `PaintSurface` handler draws the Bézier curve and then sizes the text to fit along its full length:

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    // Draw path with cubic Bezier curve
    using (SKPath path = new SKPath())
    {
        path.MoveTo(touchPoints[0].Center);
        path.CubicTo(touchPoints[1].Center,
                     touchPoints[2].Center,
                     touchPoints[3].Center);

        canvas.DrawPath(path, strokePaint);

        // Get path length
        SKPathMeasure pathMeasure = new SKPathMeasure(path, false, 1);

        // Find new text size
        textPaint.TextSize = pathMeasure.Length / baseTextWidth * 10;

        // Draw text on path
        canvas.DrawTextOnPath(text, path, 0, 0, textPaint);
    }
    ...
}

```

The `Length` property of the newly created `SKPathMeasure` object obtains the length of the path. The path length is divided by the `baseTextWidth` value (which is the width of the text based on a text size of 10) and then multiplied by the base text size of 10. The result is a new text size for displaying the text along that path:



As the Bézier curve gets longer or shorter, you can see the text size change.

Traversing the Path

`SKPathMeasure` can do more than just measure the length of the path. For any value between zero and the path length, an `SKPathMeasure` object can obtain the position on the path, and the tangent to the path curve at that point. The tangent is available as a vector in the form of an `SKPoint` object, or as a rotation encapsulated in an `SKMatrix` object. Here are the methods of `SKPathMeasure` that obtain this information in varied and flexible

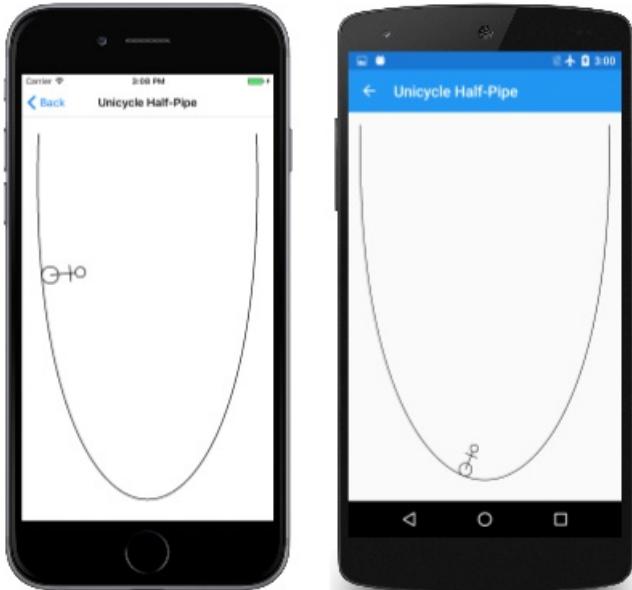
ways:

```
Boolean GetPosition (Single distance, out SKPoint position)  
Boolean GetTangent (Single distance, out SKPoint tangent)  
Boolean GetPositionAndTangent (Single distance, out SKPoint position, out SKPoint tangent)  
Boolean GetMatrix (Single distance, out SKMatrix matrix, SKPathMeasureMatrixFlags flag)
```

The members of the `SKPathMeasureMatrixFlags` enumeration are:

- `GetPosition`
- `GetTangent`
- `GetPositionAndTangent`

The **Unicycle Half-Pipe** page animates a stick figure on a unicycle that seems to ride back and forth along a cubic Bézier curve:



The `skPaint` object used for stroking both the half-pipe and the unicycle is defined as a field in the `UnicycleHalfPipePage` class. Also defined is the `skPath` object for the unicycle:

```
public class UnicycleHalfPipePage : ContentPage
{
    ...
    SKPaint strokePaint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        StrokeWidth = 3,
        Color = SKColors.Black
    };

    SKPath unicyclePath = SKPath.ParseSvgPathData(
        "M 0 0" +
        "A 25 25 0 0 0 0 -50" +
        "A 25 25 0 0 0 0 0 Z" +
        "M 0 -25 L 0 -100" +
        "A 15 15 0 0 0 0 -130" +
        "A 15 15 0 0 0 0 -100 Z" +
        "M -25 -85 L 25 -85");
    ...
}
```

The class contains the standard overrides of the `OnAppearing` and `OnDisappearing` methods for animation. The `PaintSurface` handler creates the path for the half-pipe and then draws it. An `SKPathMeasure` object is then created based on this path:

```

public class UnicycleHalfPipePage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        using (SKPath pipePath = new SKPath())
        {
            pipePath.MoveTo(50, 50);
            pipePath.CubicTo(0, 1.25f * info.Height,
                info.Width - 0, 1.25f * info.Height,
                info.Width - 50, 50);

            canvas.DrawPath(pipePath, strokePaint);

            using (SKPathMeasure pathMeasure = new SKPathMeasure(pipePath))
            {
                float length = pathMeasure.Length;

                // Animate t from 0 to 1 every three seconds
                TimeSpan timeSpan = new TimeSpan(DateTime.Now.Ticks);
                float t = (float)(timeSpan.TotalSeconds % 5 / 5);

                // t from 0 to 1 to 0 but slower at beginning and end
                t = (float)((1 - Math.Cos(t * 2 * Math.PI)) / 2);

                SKMatrix matrix;
                pathMeasure.GetMatrix(t * length, out matrix,
                    SKPathMeasureMatrixFlagsGetPositionAndTangent);

                canvas.SetMatrix(matrix);
                canvas.DrawPath(unicyclePath, strokePaint);
            }
        }
    }
}

```

The `PaintSurface` handler calculates a value of `t` that goes from 0 to 1 every five seconds. It then uses the `Math.Cos` function to convert that to a value of `t` that ranges from 0 to 1 and back to 0, where 0 corresponds to the unicycle at the beginning on the top left, while 1 corresponds to the unicycle at the top right. The cosine function causes the speed to be slowest at the top of the pipe and fastest at the bottom.

Notice that this value of `t` must be multiplied by the path length for the first argument to `GetMatrix`. The matrix is then applied to the `SKCanvas` object for drawing the unicycle path.

Enumerating the Path

Two embedded classes of `SKPath` allow you to enumerate the contents of path. These classes are `SKPath.Iterator` and `SKPath.RawIterator`. The two classes are very similar, but `SKPath.Iterator` can eliminate elements in the path with a zero length, or close to a zero length. The `RawIterator` is used in the example below.

You can obtain an object of type `SKPath.RawIterator` by calling the `CreateRawIterator` method of `SKPath`. Enumerating through the path is accomplished by repeatedly calling the `Next` method. Pass to it an array of four `SKPoint` values:

```

SKPoint[] points = new SKPoint[4];
...
SKPathVerb pathVerb = rawIterator.Next(points);

```

The `Next` method returns a member of the `SKPathVerb` enumeration type. These values indicate the particular drawing command in the path. The number of valid points inserted in the array depends on this verb:

- `Move` with a single point
- `Line` with two points
- `Cubic` with four points
- `Quad` with three points
- `Conic` with three points (and also call the `ConicWeight` method for the weight)
- `Close` with one point
- `Done`

The `Done` verb indicates that the path enumeration is complete.

Notice that there are no `Arc` verbs. This indicates that all arcs are converted into Bézier curves when added to the path.

Some of the information in the `SKPoint` array is redundant. For example, if a `Move` verb is followed by a `Line` verb, then the first of the two points that accompany the `Line` is the same as the `Move` point. In practice, this redundancy is very helpful. When you get a `Cubic` verb, it is accompanied by all four points that define the cubic Bézier curve. You don't need to retain the current position established by the previous verb.

The problematic verb, however, is `close`. This command draws a straight line from the current position to the beginning of the contour established earlier by the `Move` command. Ideally, the `Close` verb should provide these two points rather than just one point. What's worse is that the point accompanying the `close` verb is always $(0, 0)$. When you enumerate through a path, you'll probably need to retain the `Move` point and the current position.

Enumerating, Flattening, and Malforming

It is sometimes desirable to apply an algorithmic transform to a path to malform it in some way:



Most of these letters consist of straight lines, yet these straight lines have apparently been twisted into curves. How is this possible?

The key is that the original straight lines are broken into a series of smaller straight lines. These individual smaller straight lines can then be manipulated in different ways to form a curve.

To help with this process, the [SkiaSharpFormsDemos](#) sample contains a static `PathExtensions` class with an `Interpolate` method that breaks down a straight line into numerous short lines that are only one unit in length. In addition, the class contains several methods that convert the three types of Bézier curves into a series of tiny straight lines that approximate the curve. (The parametric formulas were presented in the article [Three Types of Bézier Curves](#).) This process is called *flattening* the curve:

```
static class PathExtensions
{
    ...
    static SKPoint[] Interpolate(SKPoint pt0, SKPoint pt1)
    {
        int count = (int)Math.Max(1, Length(pt0, pt1));
        SKPoint[] points = new SKPoint[count];

        for (int i = 0; i < count; i++)
        {
            float t = (i + 1f) / count;
            float x = (1 - t) * pt0.X + t * pt1.X;
            float y = (1 - t) * pt0.Y + t * pt1.Y;
            points[i] = new SKPoint(x, y);
        }

        return points;
    }

    static SKPoint[] FlattenCubic(SKPoint pt0, SKPoint pt1, SKPoint pt2, SKPoint pt3)
    {
        int count = (int)Math.Max(1, Length(pt0, pt1) + Length(pt1, pt2) + Length(pt2, pt3));
        SKPoint[] points = new SKPoint[count];

        for (int i = 0; i < count; i++)
        {
            float t = (i + 1f) / count;
            float x = (1 - t) * (1 - t) * (1 - t) * pt0.X +
                      3 * t * (1 - t) * (1 - t) * pt1.X +
                      3 * t * t * (1 - t) * pt2.X +
                      t * t * t * pt3.X;
            float y = (1 - t) * (1 - t) * (1 - t) * pt0.Y +
                      3 * t * (1 - t) * (1 - t) * pt1.Y +
                      3 * t * t * (1 - t) * pt2.Y +
                      t * t * t * pt3.Y;
            points[i] = new SKPoint(x, y);
        }

        return points;
    }

    static SKPoint[] FlattenQuadratic(SKPoint pt0, SKPoint pt1, SKPoint pt2)
    {
        int count = (int)Math.Max(1, Length(pt0, pt1) + Length(pt1, pt2));
        SKPoint[] points = new SKPoint[count];

        for (int i = 0; i < count; i++)
        {
            float t = (i + 1f) / count;
            float x = (1 - t) * (1 - t) * pt0.X + 2 * t * (1 - t) * pt1.X + t * t * pt2.X;
            float y = (1 - t) * (1 - t) * pt0.Y + 2 * t * (1 - t) * pt1.Y + t * t * pt2.Y;
            points[i] = new SKPoint(x, y);
        }

        return points;
    }

    static SKPoint[] FlattenConic(SKPoint pt0, SKPoint pt1, SKPoint pt2, float weight)
    {
        int count = (int)Math.Max(1, Length(pt0, pt1) + Length(pt1, pt2));
        SKPoint[] points = new SKPoint[count];
    }
}
```

```

SKPoint[] points = new SKPoint[count];

for (int i = 0; i < count; i++)
{
    float t = (i + 1f) / count;
    float denominator = (1 - t) * (1 - t) + 2 * weight * t * (1 - t) + t * t;
    float x = (1 - t) * (1 - t) * pt0.X + 2 * weight * t * (1 - t) * pt1.X + t * t * pt2.X;
    float y = (1 - t) * (1 - t) * pt0.Y + 2 * weight * t * (1 - t) * pt1.Y + t * t * pt2.Y;
    x /= denominator;
    y /= denominator;
    points[i] = new SKPoint(x, y);
}

return points;
}

static double Length(SKPoint pt0, SKPoint pt1)
{
    return Math.Sqrt(Math.Pow(pt1.X - pt0.X, 2) + Math.Pow(pt1.Y - pt0.Y, 2));
}
}

```

All these methods are referenced from the extension method `CloneWithTransform` also included in this class and shown below. This method clones a path by enumerating the path commands and constructing a new path based on the data. However, the new path consists only of `MoveTo` and `LineTo` calls. All the curves and straight lines are reduced to a series of tiny lines.

When calling `CloneWithTransform`, you pass to the method a `Func<SKPoint, SKPoint>`, which is a function with an `SKPaint` parameter that returns an `SKPoint` value. This function is called for every point to apply a custom algorithmic transform:

```

static class PathExtensions
{
    public static SKPath CloneWithTransform(this SKPath pathIn, Func<SKPoint, SKPoint> transform)
    {
        SKPath pathOut = new SKPath();

        using (SKPath.RawIterator iterator = pathIn.CreateRawIterator())
        {
            SKPoint[] points = new SKPoint[4];
            SKPathVerb pathVerb = SKPathVerb.Move;
            SKPoint firstPoint = new SKPoint();
            SKPoint lastPoint = new SKPoint();

            while ((pathVerb = iterator.Next(points)) != SKPathVerb.Done)
            {
                switch (pathVerb)
                {
                    case SKPathVerb.Move:
                        pathOut.MoveTo(transform(points[0]));
                        firstPoint = lastPoint = points[0];
                        break;

                    case SKPathVerb.Line:
                        SKPoint[] linePoints = Interpolate(points[0], points[1]);

                        foreach (SKPoint pt in linePoints)
                        {
                            pathOut.LineTo(transform(pt));
                        }

                        lastPoint = points[1];
                        break;

                    case SKPathVerb.Cubic:
                        SKPoint[] cubicPoints = FlattenCubic(points[0], points[1], points[2], points[3]);

```

```

        SKPoint[] cubicPoints = FlattenCubic(points[0], points[1], points[2], points[3]);
        foreach (SKPoint pt in cubicPoints)
        {
            pathOut.LineTo(transform(pt));
        }

        lastPoint = points[3];
        break;

    case SKPathVerb.Quad:
        SKPoint[] quadPoints = FlattenQuadratic(points[0], points[1], points[2]);

        foreach (SKPoint pt in quadPoints)
        {
            pathOut.LineTo(transform(pt));
        }

        lastPoint = points[2];
        break;

    case SKPathVerb.Conic:
        SKPoint[] conicPoints = FlattenConic(points[0], points[1], points[2],
iterator.ConicWeight());
        foreach (SKPoint pt in conicPoints)
        {
            pathOut.LineTo(transform(pt));
        }

        lastPoint = points[2];
        break;

    case SKPathVerb.Close:
        SKPoint[] closePoints = Interpolate(lastPoint, firstPoint);

        foreach (SKPoint pt in closePoints)
        {
            pathOut.LineTo(transform(pt));
        }

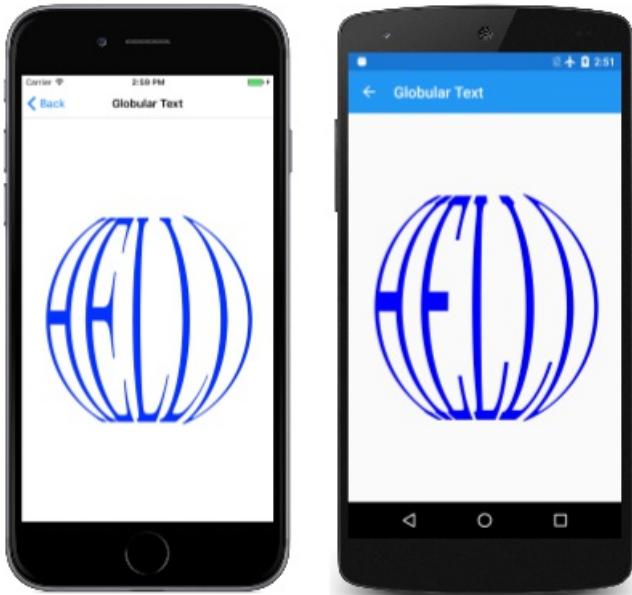
        firstPoint = lastPoint = new SKPoint(0, 0);
        pathOut.Close();
        break;
    }
}
return pathOut;
}
...
}

```

Because the cloned path is reduced to tiny straight lines, the transform function has the capability of converting straight lines to curves.

Notice that the method retains the first point of each contour in the variable called `firstPoint` and the current position after each drawing command in the variable `lastPoint`. These variables are necessary to construct the final closing line when a `Close` verb is encountered.

The `GlobularText` sample uses this extension method to seemingly wrap text around a hemisphere in a 3D effect:



The `GlobularTextPage` class constructor performs this transform. It creates an `SKPaint` object for the text, and then obtains an `SKPath` object from the `GetTextPath` method. This is the path passed to the `CloneWithTransform` extension method along with a transform function:

```

public class GlobularTextPage : ContentPage
{
    SKPath globePath;

    public GlobularTextPage()
    {
        Title = "Globular Text";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;

        using (SKPaint textPaint = new SKPaint())
        {
            textPaint.Typeface = SKTypeface.FromFamilyName("Times New Roman");
            textPaint.TextSize = 100;

            using (SKPath textPath = textPaint.GetTextPath("HELLO", 0, 0))
            {
                SKRect textPathBounds;
                textPath.GetBounds(out textPathBounds);

                globePath = textPath.CloneWithTransform((SKPoint pt) =>
                {
                    double longitude = (Math.PI / textPathBounds.Width) *
                        (pt.X - textPathBounds.Left) - Math.PI / 2;
                    double latitude = (Math.PI / textPathBounds.Height) *
                        (pt.Y - textPathBounds.Top) - Math.PI / 2;

                    longitude *= 0.75;
                    latitude *= 0.75;

                    float x = (float)(Math.Cos(latitude) * Math.Sin(longitude));
                    float y = (float)Math.Sin(latitude);

                    return new SKPoint(x, y);
                });
            }
        }
    }
}

```

The transform function first calculates two values named `longitude` and `latitude` that range from $-\pi/2$ at the top and left of the text, to $\pi/2$ at the right and bottom of the text. The range of these values isn't visually satisfactory, so they are reduced by multiplying by 0.75. (Try the code without those adjustments. The text becomes too obscure at the north and south poles, and too thin at the sides.) These three-dimensional spherical coordinates are converted to two-dimensional `x` and `y` coordinates by standard formulas.

The new path is stored as a field. The `PaintSurface` handler then merely needs to center and scale the path to display it on the screen:

```
public class GlobularTextPage : ContentPage
{
    SKPath globePath;
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        using (SKPaint pathPaint = new SKPaint())
        {
            pathPaint.Style = SKPaintStyle.Fill;
            pathPaint.Color = SKColors.Blue;
            pathPaint.StrokeWidth = 3;
            pathPaint.IsAntialias = true;

            canvas.Translate(info.Width / 2, info.Height / 2);
            canvas.Scale(0.45f * Math.Min(info.Width, info.Height));      // radius
            canvas.DrawPath(globePath, pathPaint);
        }
    }
}
```

This is a very versatile technique. If the array of path effects described in the [Path Effects](#) article doesn't quite encompass something you felt should be included, this is a way to fill in the gaps.

Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

SkiaSharp bitmaps

11/2/2020 • 2 minutes to read • [Edit Online](#)

A bitmap is a rectangular array of data corresponding to the pixels of a display device. The area of graphics programming associated with bitmaps is sometimes called *raster graphics* (named after the scan lines of early video displays) in contrast to the *vector graphics* of lines and curves.

The article [Bitmap Basics in SkiaSharp](#) described some of the fundamentals of loading and displaying bitmaps. These articles expand on the concepts in that introductory article:

Displaying SkiaSharp bitmaps

Learn how to display SkiaSharp bitmaps in their native pixel sizes or expanded to fill to rectangles while preserving the aspect ratio.

Creating and drawing on SkiaSharp bitmaps

Learn how to create SkiaSharp bitmaps and then draw on these bitmaps by creating a canvas using the bitmap as a drawing surface.

Cropping SkiaSharp bitmaps

Learn how to use SkiaSharp to design a user interface for interactively describing a cropping rectangle.

Segmented display of SkiaSharp bitmaps

Discover the specialized nine-patch and lattice options for rendering bitmaps.

Saving SkiaSharp bitmaps to files

Save bitmaps in the user's photo library.

Accessing SkiaSharp bitmap pixel bits

Discover the various techniques for accessing and modifying the pixel bits of SkiaSharp bitmaps.

Animating SkiaSharp bitmaps

Learn how to perform bitmap animation by sequentially displaying different bitmaps, and rendering animated GIF bitmaps.

Displaying SkiaSharp bitmaps

3/5/2021 • 13 minutes to read • [Edit Online](#)



[Download the sample](#)

The subject of SkiaSharp bitmaps was introduced in the article [Bitmap Basics in SkiaSharp](#). That article showed three ways to load bitmaps and three ways to display bitmaps. This article reviews the techniques to load bitmaps and goes deeper into the use of the `DrawBitmap` methods of `SKCanvas`.



The `DrawBitmapLattice` and `DrawBitmapNinePatch` methods are discussed in the article [Segmented display of SkiaSharp bitmaps](#).

Samples on this page are from the [SkiaSharpFormsDemos](#) application. From the home page of that application, choose **SkiaSharp Bitmaps**, and then go to the **Displaying Bitmaps** section.

Loading a bitmap

A bitmap used by a SkiaSharp application generally comes from one of three different sources:

- From over the Internet
- From a resource embedded in the executable
- From the user's photo library

It is also possible for a SkiaSharp application to create a new bitmap, and then draw on it or set the bitmap bits algorithmically. Those techniques are discussed in the articles [Creating and Drawing on SkiaSharp Bitmaps](#) and [Accessing SkiaSharp Bitmap Pixels](#).

In the following three code examples of loading a bitmap, the class is assumed to contain a field of type

`SKBitmap`:

```
SKBitmap bitmap;
```

As the article [Bitmap Basics in SkiaSharp](#) stated, the best way to load a bitmap over the Internet is with the `HttpClient` class. A single instance of the class can be defined as a field:

```
HttpClient httpClient = new HttpClient();
```

When using `HttpClient` with iOS and Android applications, you'll want to set project properties as described in the documents on [Transport Layer Security \(TLS\) 1.2](#).

Code that uses `HttpClient` often involves the `await` operator, so it must reside in an `async` method:

```
try
{
    using (Stream stream = await httpClient.GetStreamAsync("https:// ... "))
    using (MemoryStream memStream = new MemoryStream())
    {
        await stream.CopyToAsync(memStream);
        memStream.Seek(0, SeekOrigin.Begin);

        bitmap = SKBitmap.Decode(stream);
        ...
    };
}
catch
{
    ...
}
```

Notice that the `Stream` object obtained from `GetStreamAsync` is copied into a `MemoryStream`. Android does not allow the `Stream` from `HttpClient` to be processed by the main thread except in asynchronous methods.

The `SKBitmap.Decode` does a lot of work: The `stream` object passed to it references a block of memory containing an entire bitmap in one of the common bitmap file formats, generally JPEG, PNG, or GIF. The `Decode` method must determine the format, and then decode the bitmap file into SkiaSharp's own internal bitmap format.

After your code calls `SKBitmap.Decode`, it will probably invalidate the `CanvasView` so that the `PaintSurface` handler can display the newly loaded bitmap.

The second way to load a bitmap is by including the bitmap as an embedded resource in the .NET Standard library referenced by the individual platform projects. A resource ID is passed to the `GetManifestResourceStream` method. This resource ID consists of the assembly name, folder name, and filename of the resource separated by periods:

```
string resourceId = "assemblyName.folderName.fileName";
Assembly assembly = GetType().GetTypeInfo().Assembly;

using (Stream stream = assembly.GetManifestResourceStream(resourceId))
{
    bitmap = SKBitmap.Decode(stream);
    ...
}
```

Bitmap files can also be stored as resources in the individual platform project for iOS, Android, and the Universal Windows Platform (UWP). However, loading those bitmaps requires code that is located in the platform project.

A third approach to obtaining a bitmap is from the user's picture library. The following code uses a dependency service that is included in the `SkiaSharpFormsDemos` application. The `SkiaSharpFormsDemo` .NET Standard Library includes the `IPhotoLibrary` interface, while each of the platform projects contains a `PhotoLibrary` class that implements that interface.

```
IPhotoLibrary picturePicker = DependencyService.Get<IPhotoLibrary>();

using (Stream stream = await picturePicker.GetImageStreamAsync())
{
    if (stream != null)
    {
        bitmap = SKBitmap.Decode(stream);
        ...
    }
}
```

Generally, such code also invalidates the `CanvasView` so that the `PaintSurface` handler can display the new bitmap.

The `SKBitmap` class defines several useful properties, including `Width` and `Height`, that reveal the pixel dimensions of the bitmap, as well as many methods, including methods to create bitmaps, to copy them, and to expose the pixel bits.

Displaying in pixel dimensions

The SkiaSharp `Canvas` class defines four `DrawBitmap` methods. These methods allow bitmaps to be displayed in two fundamentally different ways:

- Specifying an `SKPoint` value (or separate `x` and `y` values) displays the bitmap in its pixel dimensions. The pixels of the bitmap are mapped directly to pixels of the video display.
- Specifying a rectangle causes the bitmap to be stretched to the size and shape of the rectangle.

You display a bitmap in its pixel dimensions using `DrawBitmap` with an `SKPoint` parameter or `DrawBitmap` with separate `x` and `y` parameters:

```
DrawBitmap(SKBitmap bitmap, SKPoint pt, SKPaint paint = null)

DrawBitmap(SKBitmap bitmap, float x, float y, SKPaint paint = null)
```

These two methods are functionally identical. The specified point indicates the location of the upper-left corner of the bitmap relative to the canvas. Because the pixel resolution of mobile devices is so high, smaller bitmaps usually appear quite tiny on these devices.

The optional `SKPaint` parameter allows you to display the bitmap using transparency. To do this, create an `SKPaint` object and set the `Color` property to any `SKColor` value with an alpha channel less than 1. For example:

```
paint.Color = new SKColor(0, 0, 0, 0x80);
```

The `0x80` passed as the last argument indicates 50% transparency. You can also set an alpha channel on one of the pre-defined colors:

```
paint.Color = SKColors.Red.WithAlpha(0x80);
```

However, the color itself is irrelevant. Only the alpha channel is examined when you use the `SKPaint` object in a `DrawBitmap` call.

The `SKPaint` object also plays a role when displaying bitmaps using blend modes or filter effects. These are demonstrated in the articles [SkiaSharp compositing and blend modes](#) and [SkiaSharp image filters](#).

The Pixel Dimensions page in the [SkiaSharpFormsDemos](#) sample program displays a bitmap resource that is 320 pixels wide by 240 pixels high:

```
public class PixelDimensionsPage : ContentPage
{
    SKBitmap bitmap;

    public PixelDimensionsPage()
    {
        Title = "Pixel Dimensions";

        // Load the bitmap from a resource
        string resourceId = "SkiaSharpFormsDemos.Media.Banana.jpg";
        Assembly assembly = GetType().GetTypeInfo().Assembly;

        using (Stream stream = assembly.GetManifestResourceStream(resourceId))
        {
            bitmap = SKBitmap.Decode(stream);
        }

        // Create the SKCanvasView and set the PaintSurface handler
        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

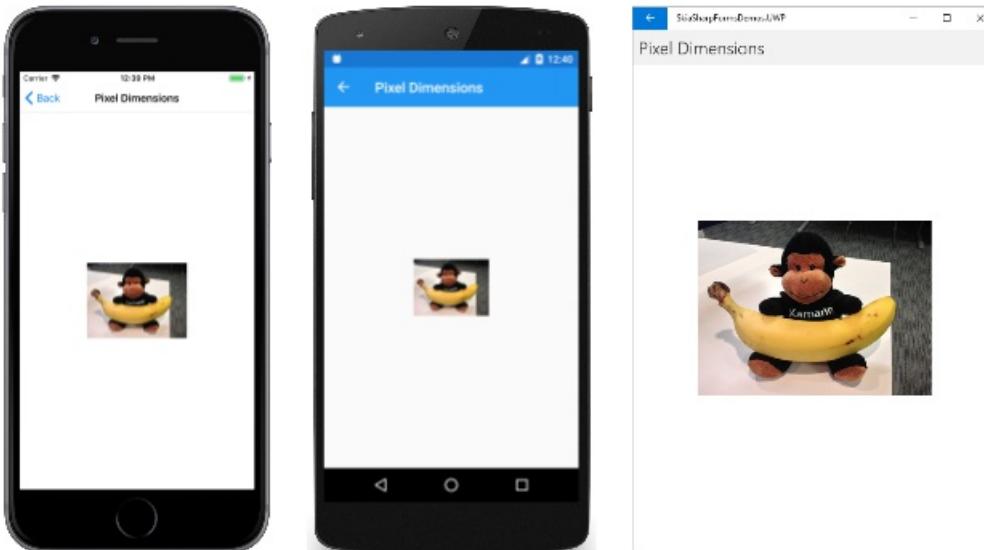
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        float x = (info.Width - bitmap.Width) / 2;
        float y = (info.Height - bitmap.Height) / 2;

        canvas.DrawBitmap(bitmap, x, y);
    }
}
```

The `PaintSurface` handler centers the bitmap by calculating `x` and `y` values based on the pixel dimensions of the display surface and the pixel dimensions of the bitmap:



If the application wishes to display the bitmap in its upper-left corner, it would simply pass coordinates of (0, 0).

A method for loading resource bitmaps

Many of the samples coming up will need to load bitmap resources. The static `BitmapExtensions` class in the `SkiaSharpFormsDemos` solution contains a method to help out:

```
static class BitmapExtensions
{
    public static SKBitmap LoadBitmapResource(Type type, string resourceId)
    {
        Assembly assembly = type.GetTypeInfo().Assembly;

        using (Stream stream = assembly.GetManifestResourceStream(resourceId))
        {
            return SKBitmap.Decode(stream);
        }
    }
    ...
}
```

Notice the `Type` parameter. This can be the `Type` object associated with any type in the assembly that stores the bitmap resource.

This `LoadBitmapResource` method will be used in all subsequent samples that require bitmap resources.

Stretching to fill a rectangle

The `SKCanvas` class also defines a `DrawBitmap` method that renders the bitmap to a rectangle, and another `DrawBitmap` method that renders a rectangular subset of the bitmap to a rectangle:

```
DrawBitmap(SKBitmap bitmap, SKRect dest, SKPaint paint = null)

DrawBitmap(SKBitmap bitmap, SKRect source, SKRect dest, SKPaint paint = null)
```

In both cases, the bitmap is stretched to fill the rectangle named `dest`. In the second method, the `source` rectangle allows you to select a subset of the bitmap. The `dest` rectangle is relative to the output device; the `source` rectangle is relative to the bitmap.

The [Fill Rectangle](#) page demonstrates the first of these two methods by displaying the same bitmap used in the earlier example in a rectangle the same size as the canvas:

```

public class FillRectanglePage : ContentPage
{
    SKBitmap bitmap =
        BitmapExtensions.LoadBitmapResource(typeof(FillRectanglePage),
                                              "SkiaSharpFormsDemos.Media.Banana.jpg");

    public FillRectanglePage ()
    {
        Title = "Fill Rectangle";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

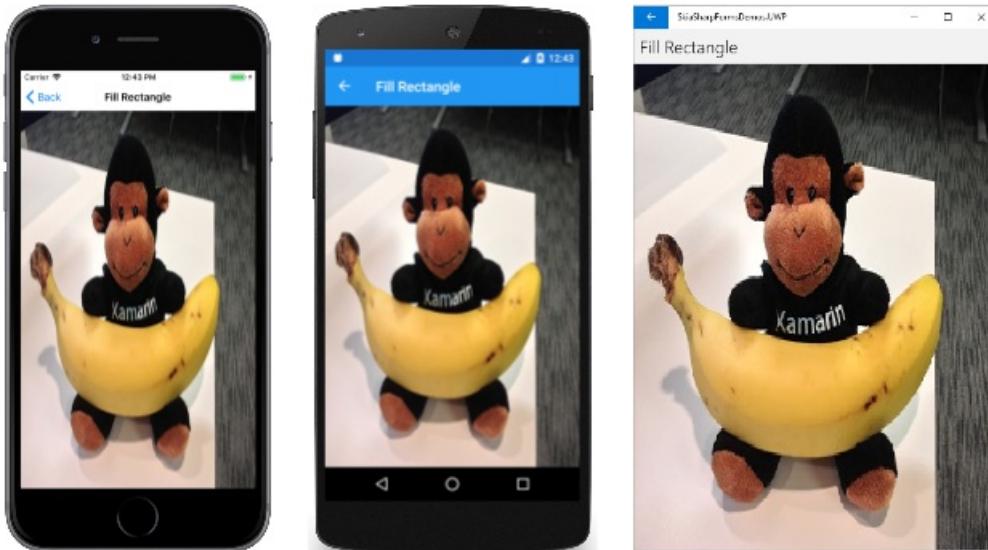
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        canvas.DrawBitmap(bitmap, info.Rect);
    }
}

```

Notice the use of the new `BitmapExtensions.LoadBitmapResource` method to set the `SKBitmap` field. The destination rectangle is obtained from the `Rect` property of `SKImageInfo`, which describes the size of the display surface:



This is usually *not* what you want. The image is distorted by being stretched differently in the horizontal and vertical directions. When displaying a bitmap in something other than its pixel size, usually you want to preserve the bitmap's original aspect ratio.

Stretching while preserving the aspect ratio

Stretching a bitmap while preserving the aspect ratio is a process also known as *uniform scaling*. That term suggests an algorithmic approach. One possible solution is shown in the **Uniform Scaling** page:

```

public class UniformScalingPage : ContentPage
{
    SKBitmap bitmap =
        BitmapExtensions.LoadBitmapResource(typeof(UniformScalingPage),
            "SkiaSharpFormsDemos.Media.Banana.jpg");

    public UniformScalingPage()
    {
        Title = "Uniform Scaling";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

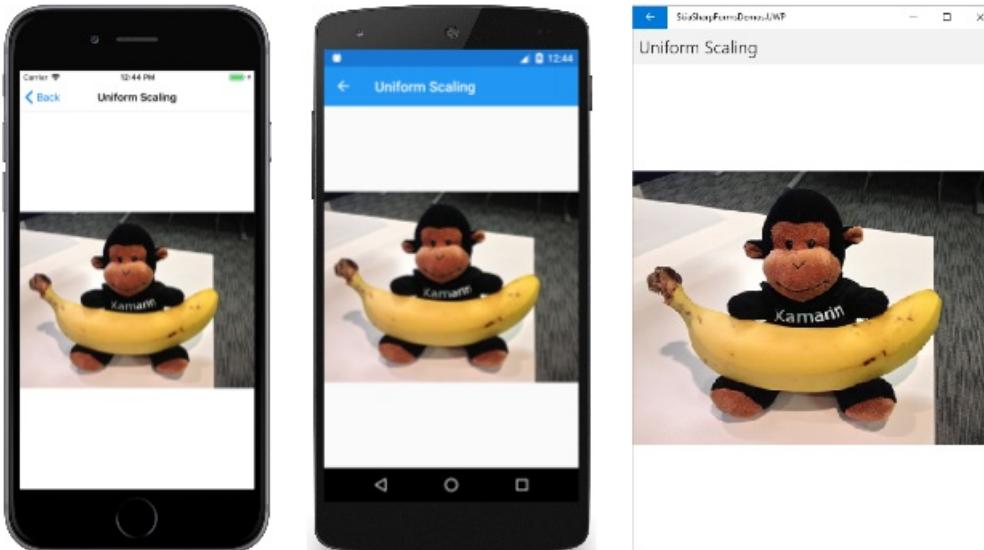
        canvas.Clear();

        float scale = Math.Min((float)info.Width / bitmap.Width,
            (float)info.Height / bitmap.Height);
        float x = (info.Width - scale * bitmap.Width) / 2;
        float y = (info.Height - scale * bitmap.Height) / 2;
        SKRect destRect = new SKRect(x, y, x + scale * bitmap.Width,
            y + scale * bitmap.Height);

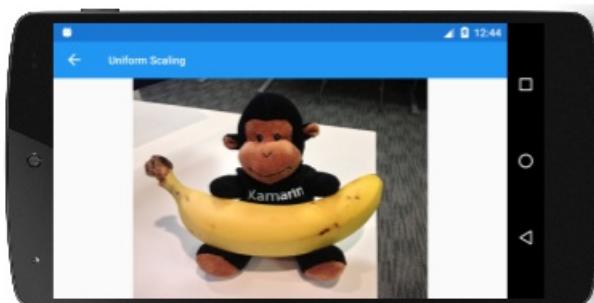
        canvas.DrawBitmap(bitmap, destRect);
    }
}

```

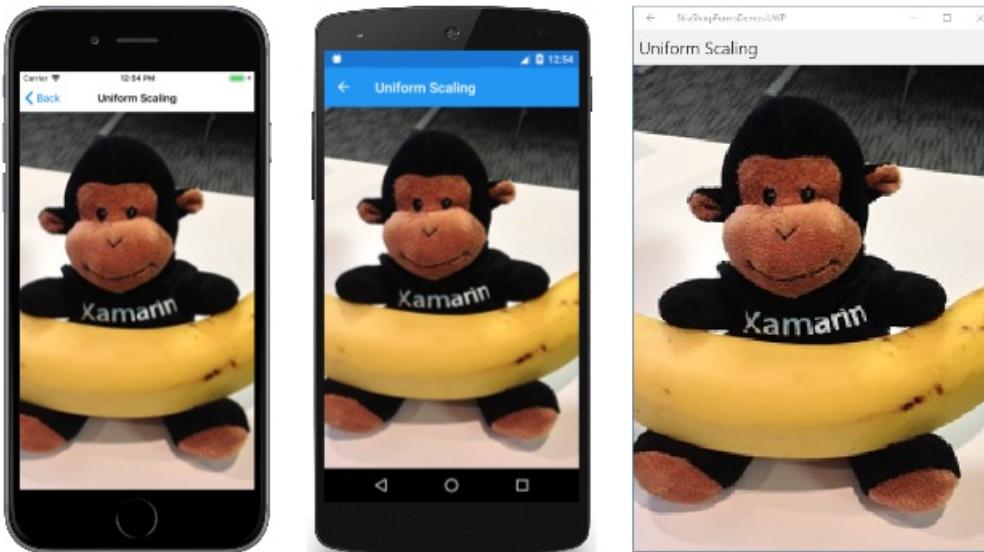
The `PaintSurface` handler calculates a `scale` factor that is the minimum of the ratio of the display width and height to the bitmap width and height. The `x` and `y` values can then be calculated for centering the scaled bitmap within the display width and height. The destination rectangle has an upper-left corner of `x` and `y` and a lower-right corner of those values plus the scaled width and height of the bitmap:



Turn the phone sideways to see the bitmap stretched to that area:



The advantage of using this `scale` factor becomes obvious when you want to implement a slightly different algorithm. Suppose you want to preserve the bitmap's aspect ratio but also fill the destination rectangle. The only way this is possible is by cropping part of the image, but you can implement that algorithm simply by changing `Math.Min` to `Math.Max` in the above code. Here's the result:



The bitmap's aspect ratio is preserved but areas on the left and right of the bitmap are cropped.

A versatile bitmap display function

XAML-based programming environments (such as UWP and Xamarin.Forms) have a facility to expand or shrink the size of bitmaps while preserving their aspect ratios. Although SkiaSharp does not include this feature, you

can implement it yourself. The `BitmapExtensions` class included in the `SkiaSharpFormsDemos` application shows how. The class defines two new `DrawBitmap` methods that perform the aspect ratio calculation. These new methods are extension methods of `SKCanvas`.

The new `DrawBitmap` methods include a parameter of type `BitmapStretch`, an enumeration defined in the `BitmapExtensions.cs` file:

```
public enum BitmapStretch
{
    None,
    Fill,
    Uniform,
    UniformToFill,
    AspectFit = Uniform,
    AspectFill = UniformToFill
}
```

The `None`, `Fill`, `Uniform`, and `UniformToFill` members are the same as those in the UWP `Stretch` enumeration. The similar Xamarin.Forms `Aspect` enumeration defines members `Fill`, `AspectFit`, and `AspectFill`.

The **Uniform Scaling** page shown above centers the bitmap within the rectangle, but you might want other options, such as positioning the bitmap at the left or right side of the rectangle, or the top or bottom. That's the purpose of the `BitmapAlignment` enumeration:

```
public enum BitmapAlignment
{
    Start,
    Center,
    End
}
```

Alignment settings have no effect when used with `BitmapStretch.Fill`.

The first `DrawBitmap` extension function contains a destination rectangle but no source rectangle. Defaults are defined so that if you want the bitmap centered, you need only specify a `BitmapStretch` member:

```

static class BitmapExtensions
{
    ...
    public static void DrawBitmap(this SKCanvas canvas, SKBitmap bitmap, SKRect dest,
        BitmapStretch stretch,
        BitmapAlignment horizontal = BitmapAlignment.Center,
        BitmapAlignment vertical = BitmapAlignment.Center,
        SKPaint paint = null)
    {
        if (stretch == BitmapStretch.Fill)
        {
            canvas.DrawBitmap(bitmap, dest, paint);
        }
        else
        {
            float scale = 1;

            switch (stretch)
            {
                case BitmapStretch.None:
                    break;

                case BitmapStretch.Uniform:
                    scale = Math.Min(dest.Width / bitmap.Width, dest.Height / bitmap.Height);
                    break;

                case BitmapStretch.UniformToFill:
                    scale = Math.Max(dest.Width / bitmap.Width, dest.Height / bitmap.Height);
                    break;
            }

            SKRect display = CalculateDisplayRect(dest, scale * bitmap.Width, scale * bitmap.Height,
                horizontal, vertical);

            canvas.DrawBitmap(bitmap, display, paint);
        }
    }
    ...
}

```

The primary purpose of this method is to calculate a scaling factor named `scale` that is then applied to the bitmap width and height when calling the `CalculateDisplayRect` method. This is the method that calculates a rectangle for displaying the bitmap based on the horizontal and vertical alignment:

```

static class BitmapExtensions
{
    ...
    static SKRect CalculateDisplayRect(SKRect dest, float bmpWidth, float bmpHeight,
                                      BitmapAlignment horizontal, BitmapAlignment vertical)
    {
        float x = 0;
        float y = 0;

        switch (horizontal)
        {
            case BitmapAlignment.Center:
                x = (dest.Width - bmpWidth) / 2;
                break;

            case BitmapAlignment.Start:
                break;

            case BitmapAlignment.End:
                x = dest.Width - bmpWidth;
                break;
        }

        switch (vertical)
        {
            case BitmapAlignment.Center:
                y = (dest.Height - bmpHeight) / 2;
                break;

            case BitmapAlignment.Start:
                break;

            case BitmapAlignment.End:
                y = dest.Height - bmpHeight;
                break;
        }

        x += dest.Left;
        y += dest.Top;

        return new SKRect(x, y, x + bmpWidth, y + bmpHeight);
    }
}

```

The `BitmapExtensions` class contains an additional `DrawBitmap` method with a source rectangle for specifying a subset of the bitmap. This method is similar to the first one except that the scaling factor is calculated based on the `source` rectangle, and then applied to the `source` rectangle in the call to `CalculateDisplayRect`:

```

static class BitmapExtensions
{
    ...
    public static void DrawBitmap(this SKCanvas canvas, SKBitmap bitmap, SKRect source, SKRect dest,
        BitmapStretch stretch,
        BitmapAlignment horizontal = BitmapAlignment.Center,
        BitmapAlignment vertical = BitmapAlignment.Center,
        SKPaint paint = null)
    {
        if (stretch == BitmapStretch.Fill)
        {
            canvas.DrawBitmap(bitmap, source, dest, paint);
        }
        else
        {
            float scale = 1;

            switch (stretch)
            {
                case BitmapStretch.None:
                    break;

                case BitmapStretch.Uniform:
                    scale = Math.Min(dest.Width / source.Width, dest.Height / source.Height);
                    break;

                case BitmapStretch.UniformToFill:
                    scale = Math.Max(dest.Width / source.Width, dest.Height / source.Height);
                    break;
            }

            SKRect display = CalculateDisplayRect(dest, scale * source.Width, scale * source.Height,
                horizontal, vertical);

            canvas.DrawBitmap(bitmap, source, display, paint);
        }
    }
}
...
}

```

The first of these two new `DrawBitmap` methods is demonstrated in the [Scaling Modes](#) page. The XAML file contains three `Picker` elements that let you select members of the `BitmapStretch` and `BitmapAlignment` enumerations:

```

<?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:SkiaSharpFormsDemos"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Banners.ScalingModesPage"
    Title="Scaling Modes">

    <Grid Padding="10">
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

```

```

<skia:SKCanvasView x:Name="canvasView"
    Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2"
    PaintSurface="OnCanvasViewPaintSurface" />

<Label Text="Stretch:"
    Grid.Row="1" Grid.Column="0"
    VerticalOptions="Center" />

<Picker x:Name="stretchPicker"
    Grid.Row="1" Grid.Column="1"
    SelectedIndexChanged="OnPickerSelectedIndexChanged">
    <Picker.ItemsSource>
        <x:Array Type="{x:Type local:BitmapStretch}">
            <x:Static Member="local:BitmapStretch.None" />
            <x:Static Member="local:BitmapStretch.Fill" />
            <x:Static Member="local:BitmapStretch.Uniform" />
            <x:Static Member="local:BitmapStretch.UniformToFill" />
        </x:Array>
    </Picker.ItemsSource>
    <Picker.SelectedIndex>
        0
    </Picker.SelectedIndex>
</Picker>

<Label Text="Horizontal Alignment:"*
    Grid.Row="2" Grid.Column="0"
    VerticalOptions="Center" />

<Picker x:Name="horizontalPicker"
    Grid.Row="2" Grid.Column="1"
    SelectedIndexChanged="OnPickerSelectedIndexChanged">
    <Picker.ItemsSource>
        <x:Array Type="{x:Type local:BitmapAlignment}">
            <x:Static Member="local:BitmapAlignment.Start" />
            <x:Static Member="local:BitmapAlignment.Center" />
            <x:Static Member="local:BitmapAlignment.End" />
        </x:Array>
    </Picker.ItemsSource>
    <Picker.SelectedIndex>
        0
    </Picker.SelectedIndex>
</Picker>

<Label Text="Vertical Alignment:"*
    Grid.Row="3" Grid.Column="0"
    VerticalOptions="Center" />

<Picker x:Name="verticalPicker"
    Grid.Row="3" Grid.Column="1"
    SelectedIndexChanged="OnPickerSelectedIndexChanged">
    <Picker.ItemsSource>
        <x:Array Type="{x:Type local:BitmapAlignment}">
            <x:Static Member="local:BitmapAlignment.Start" />
            <x:Static Member="local:BitmapAlignment.Center" />
            <x:Static Member="local:BitmapAlignment.End" />
        </x:Array>
    </Picker.ItemsSource>
    <Picker.SelectedIndex>
        0
    </Picker.SelectedIndex>
</Picker>
</Grid>
</ContentPage>

```

The code-behind file simply invalidates the `CanvasView` when any `Picker` item has changed. The `PaintSurface`

handler accesses the three `Picker` views for calling the `DrawBitmap` extension method:

```
public partial class ScalingModesPage : ContentPage
{
    SKBitmap bitmap =
        BitmapExtensions.LoadBitmapResource(typeof(ScalingModesPage),
                                              "SkiaSharpFormsDemos.Media.Banana.jpg");
    public ScalingModesPage()
    {
        InitializeComponent();
    }

    private void OnPickerSelectedIndexChanged(object sender, EventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

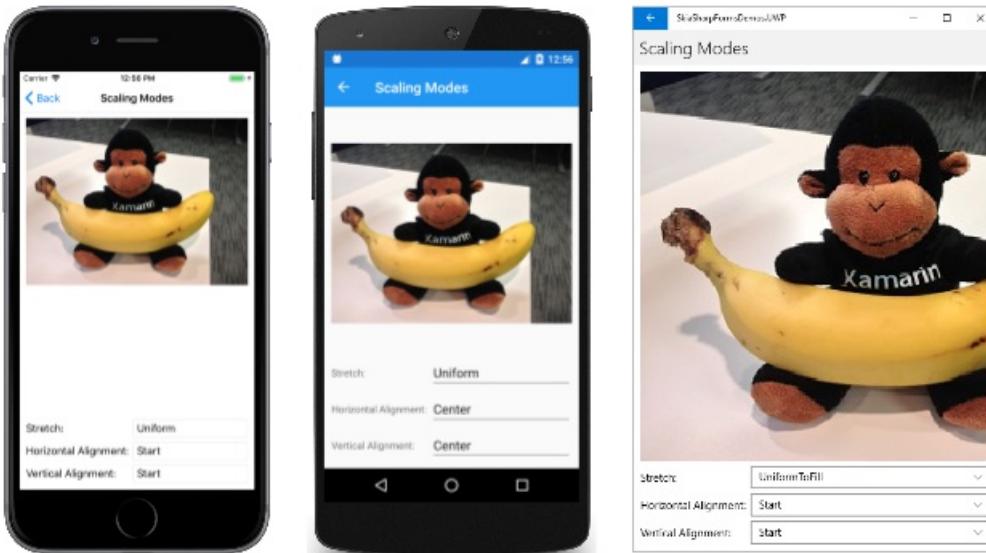
        canvas.Clear();

        SKRect dest = new SKRect(0, 0, info.Width, info.Height);

        BitmapStretch stretch = (BitmapStretch)stretchPicker.SelectedItem;
        BitmapAlignment horizontal = (BitmapAlignment)horizontalPicker.SelectedItem;
        BitmapAlignment vertical = (BitmapAlignment)verticalPicker.SelectedItem;

        canvas.DrawBitmap(bitmap, dest, stretch, horizontal, vertical);
    }
}
```

Here are some combinations of options:



The **Rectangle Subset** page has virtually the same XAML file as **Scaling Modes**, but the code-behind file defines a rectangular subset of the bitmap given by the `SOURCE` field:

```

public partial class ScalingModesPage : ContentPage
{
    SKBitmap bitmap =
        BitmapExtensions.LoadBitmapResource(typeof(ScalingModesPage),
            "SkiaSharpFormsDemos.Media.Banana.jpg");

    static readonly SKRect SOURCE = new SKRect(94, 12, 212, 118);

    public RectangleSubsetPage()
    {
        InitializeComponent();
    }

    private void OnPickerSelectedIndexChanged(object sender, EventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

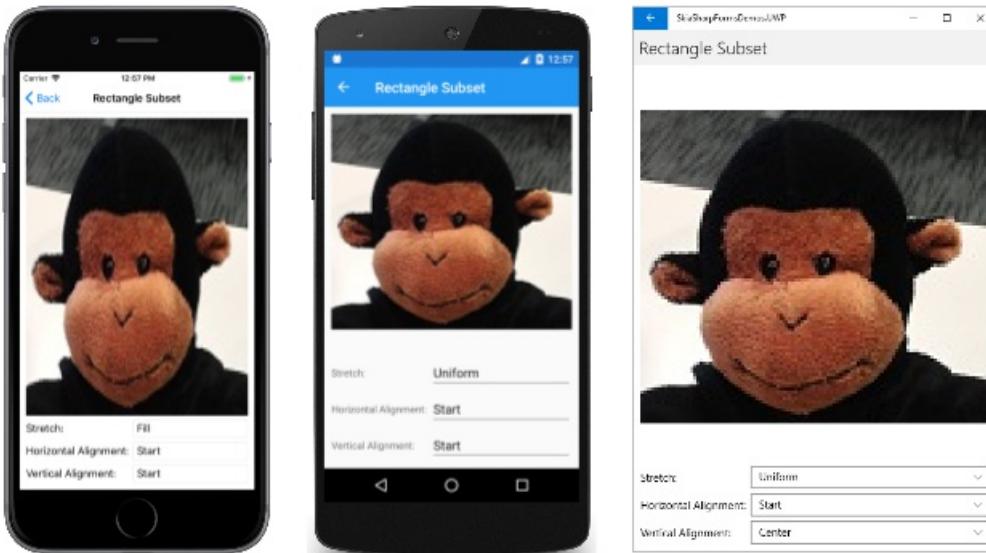
        SKRect dest = new SKRect(0, 0, info.Width, info.Height);

        BitmapStretch stretch = (BitmapStretch)stretchPicker.SelectedItem;
        BitmapAlignment horizontal = (BitmapAlignment)horizontalPicker.SelectedItem;
        BitmapAlignment vertical = (BitmapAlignment)verticalPicker.SelectedItem;

        canvas.DrawBitmap(bitmap, SOURCE, dest, stretch, horizontal, vertical);
    }
}

```

This rectangle source isolates the monkey's head, as shown in these screenshots:



Related links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Creating and drawing on SkiaSharp bitmaps

3/5/2021 • 16 minutes to read • [Edit Online](#)

 [Download the sample](#)

You've seen how an application can load bitmaps from the Web, from application resources, and from the user's photo library. It's also possible to create new bitmaps within your application. The simplest approach involves one of the constructors of [SKBitmap](#):

```
SKBitmap bitmap = new SKBitmap(width, height);
```

The `width` and `height` parameters are integers and specify the pixel dimensions of the bitmap. This constructor creates a full-color bitmap with four bytes per pixel: one byte each for the red, green, blue, and alpha (opacity) components.

After you've created a new bitmap, you need to get something on the surface of the bitmap. You generally do this in one of two ways:

- Draw on the bitmap using standard [Canvas](#) drawing methods.
- Access the pixel bits directly.

This article demonstrates the first approach:



The second approach is discussed in the article [Accessing SkiaSharp Bitmap Pixels](#).

Drawing on the bitmap

Drawing on the surface of a bitmap is the same as drawing on a video display. To draw on a video display, you obtain an [SKCanvas](#) object from the [PaintSurface](#) event arguments. To draw on a bitmap, you create an [SKCanvas](#) object using the [SKCanvas](#) constructor:

```
SKCanvas canvas = new SKCanvas(bitmap);
```

When you're finished drawing on the bitmap, you can dispose of the [SKCanvas](#) object. For this reason, the [SKCanvas](#) constructor is generally called in a [using](#) statement:

```
using (SKCanvas canvas = new SKCanvas(bitmap))
{
    ... // call drawing function
}
```

The bitmap can then be displayed. At a later time, the program can create a new `SKCanvas` object based on that same bitmap, and draw on it some more.

The [Hello Bitmap](#) page in the [SkiaSharpFormsDemos](#) application writes the text "Hello, Bitmap!" on a bitmap and then displays that bitmap multiple times.

The constructor of the `HelloBitmapPage` begins by creating an `SKPaint` object for displaying text. It determines the dimensions of a text string and creates a bitmap with those dimensions. It then creates an `SKCanvas` object based on that bitmap, calls `Clear`, and then calls `DrawText`. It's always a good idea to call `Clear` with a new bitmap because a newly created bitmap might contain random data.

The constructor concludes by creating an `SKCanvasView` object to display the bitmap:

```

public partial class HelloBitmapPage : ContentPage
{
    const string TEXT = "Hello, Bitmap!";
    SKBitmap helloBitmap;

    public HelloBitmapPage()
    {
        Title = TEXT;

        // Create bitmap and draw on it
        using (SKPaint textPaint = new SKPaint { TextSize = 48 })
        {
            SKRect bounds = new SKRect();
            textPaint.MeasureText(TEXT, ref bounds);

            helloBitmap = new SKBitmap((int)bounds.Right,
                                      (int)bounds.Height);

            using (SKCanvas bitmapCanvas = new SKCanvas(helloBitmap))
            {
                bitmapCanvas.Clear();
                bitmapCanvas.DrawText(TEXT, 0, -bounds.Top, textPaint);
            }
        }

        // Create SKCanvasView to view result
        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

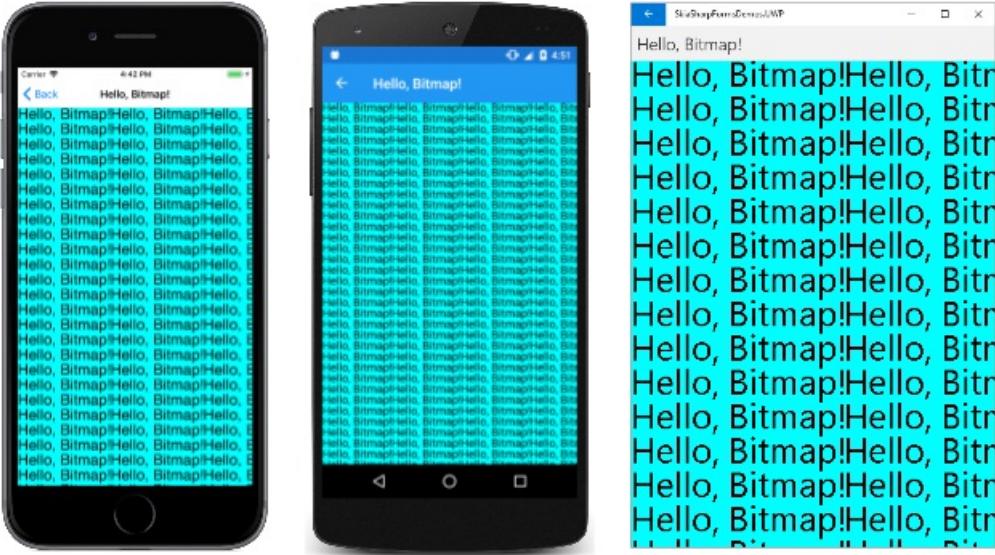
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear(SKColors.Aqua);

        for (float y = 0; y < info.Height; y += helloBitmap.Height)
            for (float x = 0; x < info.Width; x += helloBitmap.Width)
            {
                canvas.DrawBitmap(helloBitmap, x, y);
            }
    }
}

```

The `PaintSurface` handler renders the bitmap multiple times in rows and columns of the display. Notice that the `Clear` method in the `PaintSurface` handler has an argument of `SKColors.Aqua`, which colors the background of the display surface:



The appearance of the aqua background reveals that the bitmap is transparent except for the text.

Clearing and transparency

The display of the Hello Bitmap page demonstrates that the bitmap the program created is transparent except for the black text. That's why the aqua color of the display surface shows through.

The documentation of the `Clear` methods of `SKCanvas` describes them with the statement: "Replaces all the pixels in the canvas' current clip." The use of the word "replaces" reveals an important characteristic of these methods: All the drawing methods of `SKCanvas` add something to the existing display surface. The `Clear` methods *replace* what's already there.

`Clear` exists in two different versions:

- The `Clear` method with an `SKColor` parameter replaces the pixels of the display surface with pixels of that color.
- The `Clear` method with no parameters replaces the pixels with the `SKColors.Empty` color, which is a color in which all the components (red, green, blue, and alpha) are set to zero. This color is sometimes referred to as "transparent black."

Calling `Clear` with no arguments on a new bitmap initializes the entire bitmap to be entirely transparent. Anything subsequently drawn on the bitmap will usually be opaque or partially opaque.

Here's something to try: In the Hello Bitmap page, replace the `clear` method applied to the `bitmapCanvas` with this one:

```
bitmapCanvas.Clear(new SKColor(255, 0, 0, 128));
```

The order of the `SKColor` constructor parameters is red, green, blue, and alpha, where each value can range from 0 to 255. Keep in mind that an alpha value of 0 is transparent, while an alpha value of 255 is opaque.

The value (255, 0, 0, 128) clears the bitmap pixels to red pixels with a 50% opacity. This means that the bitmap background is semi-transparent. The semi-transparent red background of the bitmap combines with the aqua background of the display surface to create a gray background.

Try setting the color of the text to transparent black by putting the following assignment in the `SKPaint` initializer:

```
Color = new SKColor(0, 0, 0, 0)
```

You might think that this transparent text would create fully transparent areas of the bitmap through which you'd see the aqua background of the display surface. But that is not so. The text is drawn on top of what's already on the bitmap. The transparent text will not be visible at all.

No `Draw` method ever makes a bitmap more transparent. Only `Clear` can do that.

Bitmap color types

The simplest `SKBitmap` constructor allows you to specify an integer pixel width and height for the bitmap. Other `SKBitmap` constructors are more complex. These constructors require arguments of two enumeration types: `SKColorType` and `SKAlphaType`. Other constructors use the `SKImageInfo` structure, which consolidates this information.

The `SKColorType` enumeration has 9 members. Each of these members describes a particular way of storing the bitmap pixels:

- `Unknown`
- `Alpha8` — each pixel is 8 bits, representing an alpha value from fully transparent to fully opaque
- `Rgb565` — each pixel is 16 bits, 5 bits for red and blue, and 6 for green
- `Argb4444` — each pixel is 16 bits, 4 each for alpha, red, green, and blue
- `Rgba8888` — each pixel is 32 bits, 8 each for red, green, blue, and alpha
- `Bgra8888` — each pixel is 32 bits, 8 each for blue, green, red, and alpha
- `Index8` — each pixel is 8 bits and represents an index into an `SKColorTable`
- `Gray8` — each pixel is 8 bits representing a gray shade from black to white
- `RgbaF16` — each pixel is 64 bits, with red, green, blue, and alpha in a 16-bit floating-point format

The two formats where each pixel is 32 pixels (4 bytes) are often called *full-color* formats. Many of the other formats date from a time when video displays themselves were not capable of full color. Bitmaps of limited color were adequate for these displays and allowed bitmaps to occupy less space in memory.

These days, programmers almost always use full-color bitmaps and don't bother with other formats. The exception is the `RgbaF16` format, which allows greater color resolution than even the full-color formats. However, this format is used for specialized purposes, such as medical imaging, and doesn't make much sense when used with standard full-color displays.

This series of articles will restrict itself to the `SKBitmap` color formats used by default when no `SKColorType` member is specified. This default format is based on the underlying platform. For the platforms supported by Xamarin.Forms, the default color type is:

- `Rgba8888` for iOS and Android
- `Bgra8888` for the UWP

The only difference is the order of the 4 bytes in memory, and this only becomes an issue when you directly access the pixel bits. This won't become important until you get to the article [Accessing SkiaSharp Bitmap Pixels](#).

The `SKAlphaType` enumeration has four members:

- `Unknown`
- `Opaque` — the bitmap has no transparency
- `Premul` — color components are pre-multiplied by the alpha component

- `Unpremul` — color components are not pre-multiplied by the alpha component

Here's a 4-byte red bitmap pixel with 50% transparency, with the bytes shown in the order red, green, blue, alpha:

0xFF 0x00 0x00 0x80

When a bitmap containing semi-transparent pixels is rendered on a display surface, the color components of each bitmap pixel must be multiplied by that pixel's alpha value, and the color components of the corresponding pixel of the display surface must be multiplied by 255 minus the alpha value. The two pixels can then be combined. The bitmap can be rendered faster if the color components in the bitmap pixels have already been pre-multiplied by the alpha value. That same red pixel would be stored like this in a pre-multiplied format:

0x80 0x00 0x00 0x80

This performance improvement is why `SkiaSharp` bitmaps by default are created with a `Premul` format. But again, it becomes necessary to know this only when you access and manipulate pixel bits.

Drawing on existing bitmaps

It is not necessary to create a new bitmap to draw on it. You can also draw on an existing bitmap.

The [Monkey Moustache](#) page uses its constructor to load the `MonkeyFace.png` image. It then creates an `SKCanvas` object based on that bitmap, and uses `SKPaint` and `SKPath` objects to draw a moustache on it:

```

public partial class MonkeyMoustachePage : ContentPage
{
    SKBitmap monkeyBitmap;

    public MonkeyMoustachePage()
    {
        Title = "Monkey Moustache";

        monkeyBitmap = BitmapExtensions.LoadBitmapResource(GetType(),
            "SkiaSharpFormsDemos.Media.MonkeyFace.png");

        // Create canvas based on bitmap
        using (SKCanvas canvas = new SKCanvas(monkeyBitmap))
        {
            using (SKPaint paint = new SKPaint())
            {
                paint.Style = SKPaintStyle.Stroke;
                paint.Color = SKColors.Black;
                paint.StrokeWidth = 24;
                paint.StrokeCap = SKStrokeCap.Round;

                using (SKPath path = new SKPath())
                {
                    path.MoveTo(380, 390);
                    path.CubicTo(560, 390, 560, 280, 500, 280);

                    path.MoveTo(320, 390);
                    path.CubicTo(140, 390, 140, 280, 200, 280);

                    canvas.DrawPath(path, paint);
                }
            }
        }

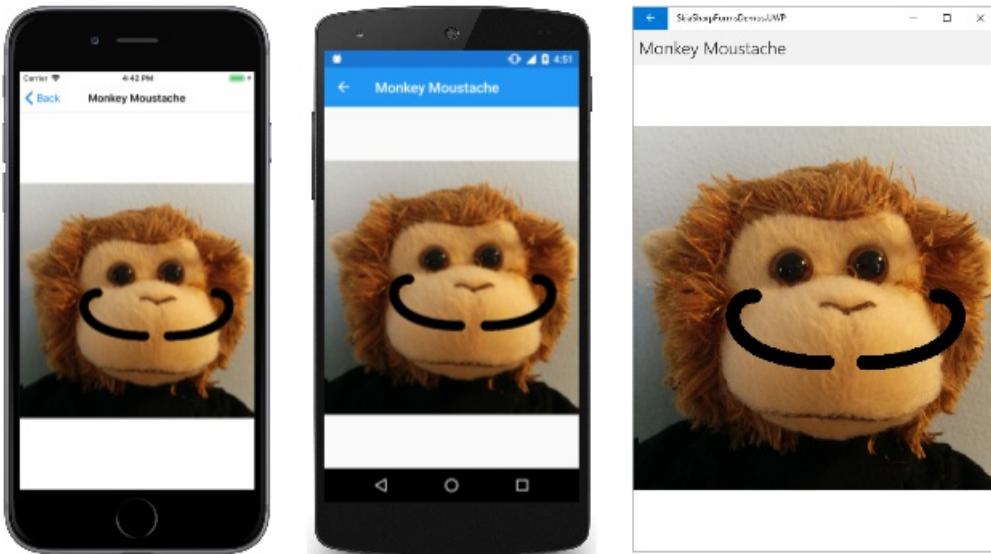
        // Create SKCanvasView to view result
        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();
        canvas.DrawBitmap(monkeyBitmap, info.Rect, BitmapStretch.Uniform);
    }
}

```

The constructor concludes by creating an `SKCanvasView` whose `PaintSurface` handler simply displays the result:



Copying and modifying bitmaps

The methods of `SKCanvas` that you can use to draw on a bitmap include `DrawBitmap`. This means that you can draw one bitmap on another, usually modifying it in some way.

The most versatile way to modify a bitmap is through accessing the actual pixel bits, a subject covered in the article [Accessing SkiaSharp bitmap pixels](#). But there are many other techniques to modify bitmaps that don't require accessing the pixel bits.

The following bitmap included with the [SkiaSharpFormsDemos](#) application is 360 pixels wide and 480 pixels in height:



Suppose you haven't received permission from the monkey on the left to publish this photograph. One solution is to obscure the monkey's face using a technique called *pixelization*. The pixels of the face are replaced with blocks of color so you can't make out the features. The blocks of color are usually derived from the original image by averaging the colors of the pixels corresponding to these blocks. But you don't need to perform this averaging yourself. It happens automatically when you copy a bitmap into a smaller pixel dimension.

The left monkey's face occupies approximately a 72-pixel square area with an upper-left corner at the point (112, 238). Let's replace that 72-pixel square area with a 9-by-9 array of colored blocks, each of which is 8-by-8 pixels square.

The **Pixelize Image** page loads in that bitmap and first creates a tiny 9-pixel square bitmap called `faceBitmap`. This is a destination for copying just the monkey's face. The destination rectangle is just 9-pixels square but the source rectangle is 72-pixels square. Every 8-by-8 block of source pixels is consolidated down to just one pixel by averaging the colors.

The next step is to copy the original bitmap into a new bitmap of the same size called `pixelizedBitmap`. The tiny `faceBitmap` is then copied on top of that with a 72-pixel square destination rectangle so that each pixel of `faceBitmap` is expanded to 8 times its size:

```

public class PixelizedImagePage : ContentPage
{
    SKBitmap pixelizedBitmap;

    public PixelizedImagePage ()
    {
        Title = "Pixelize Image";

        SKBitmap originalBitmap = BitmapExtensions.LoadBitmapResource(GetType(),
            "SkiaSharpFormsDemos.Media.MountainClimbers.jpg");

        // Create tiny bitmap for pixelized face
        SKBitmap faceBitmap = new SKBitmap(9, 9);

        // Copy subset of original bitmap to that
        using (SKCanvas canvas = new SKCanvas(faceBitmap))
        {
            canvas.Clear();
            canvas.DrawBitmap(originalBitmap,
                new SKRect(112, 238, 184, 310), // source
                new SKRect(0, 0, 9, 9)); // destination
        }

        // Create full-sized bitmap for copy
        pixelizedBitmap = new SKBitmap(originalBitmap.Width, originalBitmap.Height);

        using (SKCanvas canvas = new SKCanvas(pixelizedBitmap))
        {
            canvas.Clear();

            // Draw original in full size
            canvas.DrawBitmap(originalBitmap, new SKPoint());

            // Draw tiny bitmap to cover face
            canvas.DrawBitmap(faceBitmap,
                new SKRect(112, 238, 184, 310)); // destination
        }

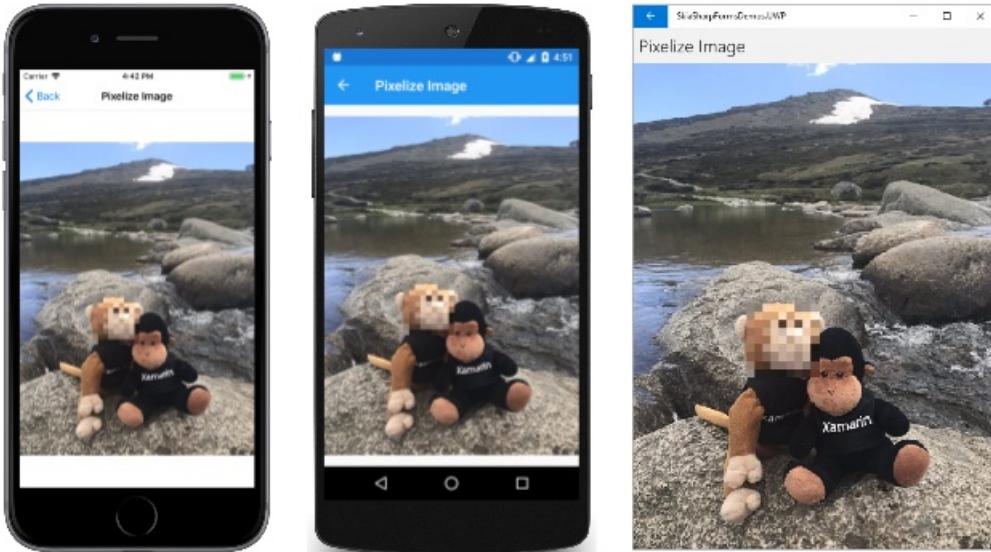
        // Create SKCanvasView to view result
        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();
        canvas.DrawBitmap(pixelizedBitmap, info.Rect, BitmapStretch.Uniform);
    }
}

```

The constructor concludes by creating an `SKCanvasView` to display the result:



Rotating bitmaps

Another common task is rotating bitmaps. This is particularly useful when retrieving bitmaps from an iPhone or iPad photo library. Unless the device was held in a particular orientation when the photo was taken, the picture is likely to be upside-down or sideways.

Turning a bitmap upside-down requires creating another bitmap the same size as the first, and then setting a transform to rotate by 180 degrees while copying the first to the second. In all of the examples in this section, `bitmap` is the `SKBitmap` object that you need to rotate:

```
SKBitmap rotatedBitmap = new SKBitmap(bitmap.Width, bitmap.Height);

using (SKCanvas canvas = new SKCanvas(rotatedBitmap))
{
    canvas.Clear();
    canvas.RotateDegrees(180, bitmap.Width / 2, bitmap.Height / 2);
    canvas.DrawBitmap(bitmap, new SKPoint());
}
```

When rotating by 90 degrees, you need to create a bitmap that is a different size than the original by swapping the height and width. For example, if the original bitmap is 1200 pixels wide and 800 pixels high, the rotated bitmap is 800 pixels wide and 1200 pixels high. Set translation and rotation so that the bitmap is rotated around its upper-left corner and then shifted into view. (Keep in mind that the `Translate` and `RotateDegrees` methods are called in the opposite order of the way that they are applied.) Here's the code for rotating 90 degrees clockwise:

```
SKBitmap rotatedBitmap = new SKBitmap(bitmap.Height, bitmap.Width);

using (SKCanvas canvas = new SKCanvas(rotatedBitmap))
{
    canvas.Clear();
    canvas.Translate(bitmap.Height, 0);
    canvas.RotateDegrees(90);
    canvas.DrawBitmap(bitmap, new SKPoint());
```

And here's a similar function for rotating 90 degrees counter-clockwise:

```

SKBitmap rotatedBitmap = new SKBitmap(bitmap.Height, bitmap.Width);

using (SKCanvas canvas = new SKCanvas(rotatedBitmap))
{
    canvas.Clear();
    canvas.Translate(0, bitmap.Width);
    canvas.RotateDegrees(-90);
    canvas.DrawBitmap(bitmap, new SKPoint());
}

```

These two methods are used in the **Photo Puzzle** pages described in the article [Cropping SkiaSharp Bitmaps](#).

A program that allows the user to rotate a bitmap in 90-degree increments needs only implement one function for rotating by 90 degrees. The user can then rotate in any increment of 90 degrees by repeated execution of this one function.

A program can also rotate a bitmap by any amount. One simple approach is to modify the function that rotates by 180 degrees by replacing 180 with a generalized `angle` variable:

```

SKBitmap rotatedBitmap = new SKBitmap(bitmap.Width, bitmap.Height);

using (SKCanvas canvas = new SKCanvas(rotatedBitmap))
{
    canvas.Clear();
    canvas.RotateDegrees(angle, bitmap.Width / 2, bitmap.Height / 2);
    canvas.DrawBitmap(bitmap, new SKPoint());
}

```

However, in the general case, this logic will crop off the corners of the rotated bitmap. A better approach is to calculate the size of the rotated bitmap using trigonometry to include those corners.

This trigonometry is shown in the **Bitmap Rotator** page. The XAML file instantiates an `SKCanvasView` and a `Slider` that can range from 0 through 360 degrees with a `Label` showing the current value:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos Bitmaps BitmapRotatorPage"
    Title="Bitmap Rotator">

    <StackLayout>
        <skia:SKCanvasView x:Name="canvasView"
            VerticalOptions="FillAndExpand"
            PaintSurface="OnCanvasViewPaintSurface" />

        <Slider x:Name="slider"
            Maximum="360"
            Margin="10, 0"
            ValueChanged="OnSliderValueChanged" />

        <Label Text="{Binding Source={x:Reference slider},
            Path=Value,
            StringFormat='Rotate by {0:F0}°'}"
            HorizontalTextAlignment="Center" />
    </StackLayout>
</ContentPage>

```

The code-behind file loads a bitmap resource and saves it as a static read-only field named `originalBitmap`. The bitmap displayed in the `PaintSurface` handler is `rotatedBitmap`, which is initially set to `originalBitmap`:

```

public partial class BitmapRotatorPage : ContentPage
{
    static readonly SKBitmap originalBitmap =
        BitmapExtensions.LoadBitmapResource(typeof(BitmapRotatorPage),
            "SkiaSharpFormsDemos.Media.Banana.jpg");

    SKBitmap rotatedBitmap = originalBitmap;

    public BitmapRotatorPage ()
    {
        InitializeComponent ();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();
        canvas.DrawBitmap(rotatedBitmap, info.Rect, BitmapStretch.Uniform);
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        double angle = args.NewValue;
        double radians = Math.PI * angle / 180;
        float sine = (float)Math.Abs(Math.Sin(radians));
        float cosine = (float)Math.Abs(Math.Cos(radians));
        int originalWidth = originalBitmap.Width;
        int originalHeight = originalBitmap.Height;
        int rotatedWidth = (int)(cosine * originalWidth + sine * originalHeight);
        int rotatedHeight = (int)(cosine * originalHeight + sine * originalWidth);

        rotatedBitmap = new SKBitmap(rotatedWidth, rotatedHeight);

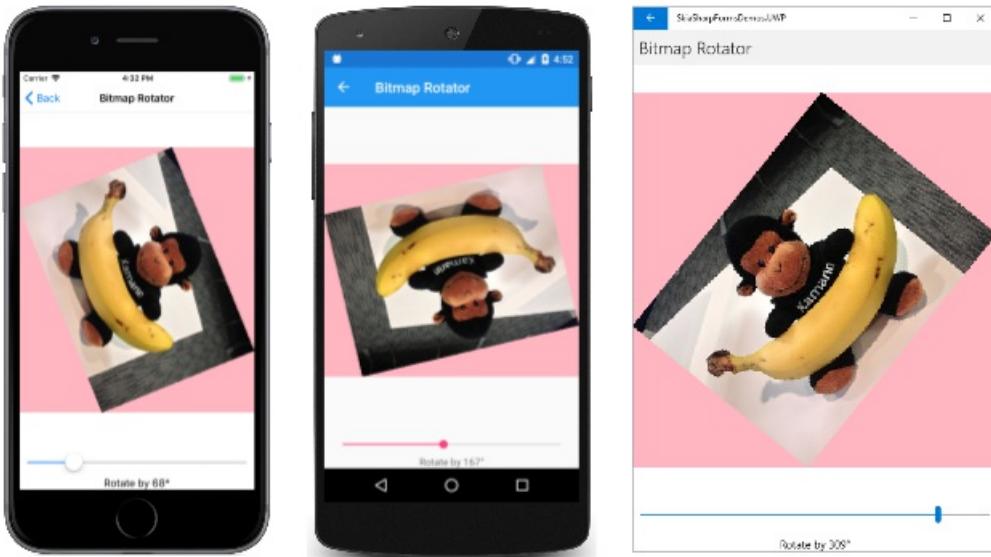
        using (SKCanvas canvas = new SKCanvas(rotatedBitmap))
        {
            canvas.Clear(SKColors.LightPink);
            canvas.Translate(rotatedWidth / 2, rotatedHeight / 2);
            canvas.RotateDegrees((float)angle);
            canvas.Translate(-originalWidth / 2, -originalHeight / 2);
            canvas.DrawBitmap(originalBitmap, new SKPoint());
        }

        canvasView.InvalidateSurface();
    }
}

```

The `ValueChanged` handler of the `Slider` performs the operations that create a new `rotatedBitmap` based on the rotation angle. The new width and height are based on absolute values of sines and cosines of the original widths and heights. The transforms used to draw the original bitmap on the rotated bitmap move the original bitmap center to the origin, then rotate it by the specified number of degrees, and then translate that center to the center of the rotated bitmap. (The `Translate` and `RotateDegrees` methods are called in the opposite order than how they are applied.)

Notice the use of the `Clear` method to make the background of `rotatedBitmap` a light pink. This is solely to illustrate the size of `rotatedBitmap` on the display:



The rotated bitmap is just large enough to include the entire original bitmap, but no larger.

Flipping bitmaps

Another operation commonly performed on bitmaps is called *flipping*. Conceptually, the bitmap is rotated in three dimensions around a vertical axis or horizontal axis through the center of the bitmap. Vertical flipping creates a mirror image.

The [Bitmap Flipper](#) page in the [SkiaSharpFormsDemos](#) application demonstrates these processes. The XAML file contains an `SKCanvasView` and two buttons for flipping vertically and horizontally:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Bitsmaps.BitmapFlipperPage"
    Title="Bitmap Flipper">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <skia:SKCanvasView x:Name="canvasView"
            Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2"
            PaintSurface="OnCanvasViewPaintSurface" />

        <Button Text="Flip Vertical"
            Grid.Row="1" Grid.Column="0"
            Margin="0, 10"
            Clicked="OnFlipVerticalClicked" />

        <Button Text="Flip Horizontal"
            Grid.Row="1" Grid.Column="1"
            Margin="0, 10"
            Clicked="OnFlipHorizontalClicked" />
    </Grid>
</ContentPage>
```

The code-behind file implements these two operations in the `Clicked` handlers for the buttons:

```

public partial class BitmapFlipperPage : ContentPage
{
    SKBitmap bitmap =
        BitmapExtensions.LoadBitmapResource(typeof(BitmapRotatorPage),
            "SkiaSharpFormsDemos.Media.SeatedMonkey.jpg");

    public BitmapFlipperPage()
    {
        InitializeComponent();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();
        canvas.DrawBitmap(bitmap, info.Rect, BitmapStretch.Uniform);
    }

    void OnFlipVerticalClicked(object sender, ValueChangedEventArgs args)
    {
        SKBitmap flippedBitmap = new SKBitmap(bitmap.Width, bitmap.Height);

        using (SKCanvas canvas = new SKCanvas(flippedBitmap))
        {
            canvas.Clear();
            canvas.Scale(-1, 1, bitmap.Width / 2, 0);
            canvas.DrawBitmap(bitmap, new SKPoint());
        }

        bitmap = flippedBitmap;
        canvasView.InvalidateSurface();
    }

    void OnFlipHorizontalClicked(object sender, ValueChangedEventArgs args)
    {
        SKBitmap flippedBitmap = new SKBitmap(bitmap.Width, bitmap.Height);

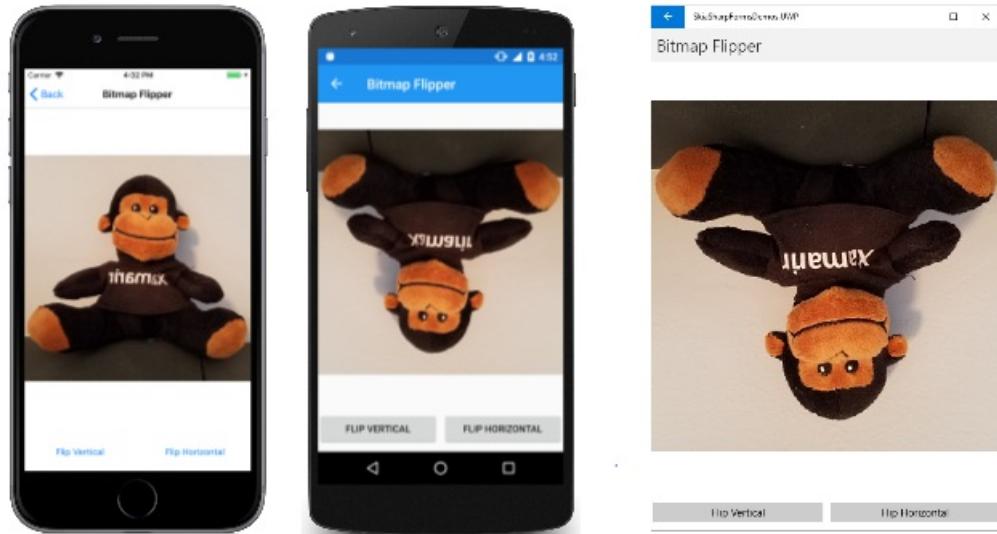
        using (SKCanvas canvas = new SKCanvas(flippedBitmap))
        {
            canvas.Clear();
            canvas.Scale(1, -1, 0, bitmap.Height / 2);
            canvas.DrawBitmap(bitmap, new SKPoint());
        }

        bitmap = flippedBitmap;
        canvasView.InvalidateSurface();
    }
}

```

The vertical flip is accomplished by a scaling transform with a horizontal scaling factor of -1 . The scaling center is the vertical center of the bitmap. The horizontal flip is a scaling transform with a vertical scaling factor of -1 .

As you can see from the reversed lettering on the monkey's shirt, flipping is not the same as rotation. But as the UWP screenshot on the right demonstrates, flipping both horizontally and vertically is the same as rotating 180 degrees:



Another common task that can be handled using similar techniques is cropping a bitmap to a rectangular subset. This is described in the next article [Cropping SkiaSharp Bitmaps](#).

Related links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Cropping SkiaSharp bitmaps

3/5/2021 • 14 minutes to read • [Edit Online](#)

 [Download the sample](#)

The [Creating and Drawing SkiaSharp Bitmaps](#) article described how an `SKBitmap` object can be passed to an `skCanvas` constructor. Any drawing method called on that canvas causes graphics to be rendered on the bitmap. These drawing methods include `DrawBitmap`, which means that this technique allows transferring part or all of one bitmap to another bitmap, perhaps with transforms applied.

You can use that technique for cropping a bitmap by calling the `DrawBitmap` method with source and destination rectangles:

```
canvas.DrawBitmap(bitmap, sourceRect, destRect);
```

However, applications that implement cropping often provide an interface for the user to interactively select the cropping rectangle:



This article focuses on that interface.

Encapsulating the cropping rectangle

It's helpful to isolate some of the cropping logic in a class named `CroppingRectangle`. The constructor parameters include a maximum rectangle, which is generally the size of the bitmap being cropped, and an optional aspect ratio. The constructor first defines an initial cropping rectangle, which it makes public in the `Rect` property of type `SKRect`. This initial cropping rectangle is 80% of the width and height of the bitmap rectangle, but it is then adjusted if an aspect ratio is specified:

```

class CroppingRectangle
{
    ...
    SKRect maxRect;           // generally the size of the bitmap
    float? aspectRatio;

    public CroppingRectangle(SKRect maxRect, float? aspectRatio = null)
    {
        this.maxRect = maxRect;
        this.aspectRatio = aspectRatio;

        // Set initial cropping rectangle
        Rect = new SKRect(0.9f * maxRect.Left + 0.1f * maxRect.Right,
                          0.9f * maxRect.Top + 0.1f * maxRect.Bottom,
                          0.1f * maxRect.Left + 0.9f * maxRect.Right,
                          0.1f * maxRect.Top + 0.9f * maxRect.Bottom);

        // Adjust for aspect ratio
        if (aspectRatio.HasValue)
        {
            SKRect rect = Rect;
            float aspect = aspectRatio.Value;

            if (rect.Width > aspect * rect.Height)
            {
                float width = aspect * rect.Height;
                rect.Left = (maxRect.Width - width) / 2;
                rect.Right = rect.Left + width;
            }
            else
            {
                float height = rect.Width / aspect;
                rect.Top = (maxRect.Height - height) / 2;
                rect.Bottom = rect.Top + height;
            }

            Rect = rect;
        }
    }

    public SKRect Rect { set; get; }
    ...
}

```

One useful piece of information that `CroppingRectangle` also makes available is an array of `SKPoint` values corresponding to the four corners of the cropping rectangle in the order upper-left, upper-right, lower-right, and lower-left:

```

class CroppingRectangle
{
    ...
    public SKPoint[] Corners
    {
        get
        {
            return new SKPoint[]
            {
                new SKPoint(Rect.Left, Rect.Top),
                new SKPoint(Rect.Right, Rect.Top),
                new SKPoint(Rect.Right, Rect.Bottom),
                new SKPoint(Rect.Left, Rect.Bottom)
            };
        }
    }
    ...
}

```

This array is used in the following method, which is called `HitTest`. The `SKPoint` parameter is a point corresponding to a finger touch or a mouse click. The method returns an index (0, 1, 2, or 3) corresponding to the corner that the finger or mouse pointer touched, within a distance given by the `radius` parameter:

```

class CroppingRectangle
{
    ...
    public int HitTest(SKPoint point, float radius)
    {
        SKPoint[] corners = Corners;

        for (int index = 0; index < corners.Length; index++)
        {
            SKPoint diff = point - corners[index];

            if ((float)Math.Sqrt(diff.X * diff.X + diff.Y * diff.Y) < radius)
            {
                return index;
            }
        }

        return -1;
    }
    ...
}

```

If the touch or mouse point was not within `radius` units of any corner, the method returns `-1`.

The final method in `CroppingRectangle` is called `MoveCorner`, which is called in response to touch or mouse movement. The two parameters indicate the index of the corner being moved, and the new location of that corner. The first half of the method adjusts the cropping rectangle based on the new location of the corner, but always within the bounds of `maxRect`, which is the size of the bitmap. This logic also takes account of the `MINIMUM` field to avoid collapsing the cropping rectangle into nothing:

```

class CroppingRectangle
{
    const float MINIMUM = 10; // pixels width or height
    ...
    public void MoveCorner(int index, SKPoint point)
    {
        SKRect rect = Rect;

        switch (index)
        {
            case 0: // upper-left
                rect.Left = Math.Min(Math.Max(point.X, maxRect.Left), rect.Right - MINIMUM);
                rect.Top = Math.Min(Math.Max(point.Y, maxRect.Top), rect.Bottom - MINIMUM);
                break;

            case 1: // upper-right
                rect.Right = Math.Max(Math.Min(point.X, maxRect.Right), rect.Left + MINIMUM);
                rect.Top = Math.Min(Math.Max(point.Y, maxRect.Top), rect.Bottom - MINIMUM);
                break;

            case 2: // lower-right
                rect.Right = Math.Max(Math.Min(point.X, maxRect.Right), rect.Left + MINIMUM);
                rect.Bottom = Math.Max(Math.Min(point.Y, maxRect.Bottom), rect.Top + MINIMUM);
                break;

            case 3: // lower-left
                rect.Left = Math.Min(Math.Max(point.X, maxRect.Left), rect.Right - MINIMUM);
                rect.Bottom = Math.Max(Math.Min(point.Y, maxRect.Bottom), rect.Top + MINIMUM);
                break;
        }

        // Adjust for aspect ratio
        if (aspectRatio.HasValue)
        {
            float aspect = aspectRatio.Value;

            if (rect.Width > aspect * rect.Height)
            {
                float width = aspect * rect.Height;

                switch (index)
                {
                    case 0:
                    case 3: rect.Left = rect.Right - width; break;
                    case 1:
                    case 2: rect.Right = rect.Left + width; break;
                }
            }
            else
            {
                float height = rect.Width / aspect;

                switch (index)
                {
                    case 0:
                    case 1: rect.Top = rect.Bottom - height; break;
                    case 2:
                    case 3: rect.Bottom = rect.Top + height; break;
                }
            }
        }

        Rect = rect;
    }
}

```

The second half of the method adjusts for the optional aspect ratio.

Keep in mind that everything in this class is in units of pixels.

A canvas view just for cropping

The `CroppingRectangle` class you've just seen is used by the `PhotoCropperCanvasView` class, which derives from `SKCanvasView`. This class is responsible for displaying the bitmap and the cropping rectangle, as well as handling touch or mouse events for changing the cropping rectangle.

The `PhotoCropperCanvasView` constructor requires a bitmap. An aspect ratio is optional. The constructor instantiates an object of type `CroppingRectangle` based on this bitmap and aspect ratio and saves it as a field:

```
class PhotoCropperCanvasView : SKCanvasView
{
    ...
    SKBitmap bitmap;
    CroppingRectangle croppingRect;
    ...
    public PhotoCropperCanvasView(SKBitmap bitmap, float? aspectRatio = null)
    {
        this.bitmap = bitmap;

        SKRect bitmapRect = new SKRect(0, 0, bitmap.Width, bitmap.Height);
        croppingRect = new CroppingRectangle(bitmapRect, aspectRatio);
        ...
    }
    ...
}
```

Because this class derives from `SKCanvasView`, it doesn't need to install a handler for the `PaintSurface` event. It can instead override its `OnPaintSurface` method. The method displays the bitmap and uses a couple of `SKPaint` objects saved as fields to draw the current cropping rectangle:

```
class PhotoCropperCanvasView : SKCanvasView
{
    const int CORNER = 50;      // pixel length of cropper corner
    ...
    SKBitmap bitmap;
    CroppingRectangle croppingRect;
    SKMatrix inverseBitmapMatrix;
    ...
    // Drawing objects
    SKPaint cornerStroke = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.White,
        StrokeWidth = 10
    };

    SKPaint edgeStroke = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.White,
        StrokeWidth = 2
    };
    ...
    protected override void OnPaintSurface(SKPaintSurfaceEventArgs args)
    {
        base.OnPaintSurface(args);

        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
```

```

SKCanvas canvas = surface.Canvas;

canvas.Clear(SKColors.Gray);

// Calculate rectangle for displaying bitmap
float scale = Math.Min((float)info.Width / bitmap.Width, (float)info.Height / bitmap.Height);
float x = (info.Width - scale * bitmap.Width) / 2;
float y = (info.Height - scale * bitmap.Height) / 2;
SKRect bitmapRect = new SKRect(x, y, x + scale * bitmap.Width, y + scale * bitmap.Height);
canvas.DrawBitmap(bitmap, bitmapRect);

// Calculate a matrix transform for displaying the cropping rectangle
SKMatrix bitmapScaleMatrix = SKMatrix.MakeIdentity();
bitmapScaleMatrix.SetScaleTranslate(scale, scale, x, y);

// Display rectangle
SKRect scaledCropRect = bitmapScaleMatrix.MapRect(croppingRect.Rect);
canvas.DrawRect(scaledCropRect, edgeStroke);

// Display heavier corners
using (SKPath path = new SKPath())
{
    path.MoveTo(scaledCropRect.Left, scaledCropRect.Top + CORNER);
    path.LineTo(scaledCropRect.Left, scaledCropRect.Top);
    path.LineTo(scaledCropRect.Left + CORNER, scaledCropRect.Top);

    path.MoveTo(scaledCropRect.Right - CORNER, scaledCropRect.Top);
    path.LineTo(scaledCropRect.Right, scaledCropRect.Top);
    path.LineTo(scaledCropRect.Right, scaledCropRect.Top + CORNER);

    path.MoveTo(scaledCropRect.Right, scaledCropRect.Bottom - CORNER);
    path.LineTo(scaledCropRect.Right, scaledCropRect.Bottom);
    path.LineTo(scaledCropRect.Right - CORNER, scaledCropRect.Bottom);

    path.MoveTo(scaledCropRect.Left + CORNER, scaledCropRect.Bottom);
    path.LineTo(scaledCropRect.Left, scaledCropRect.Bottom);
    path.LineTo(scaledCropRect.Left, scaledCropRect.Bottom - CORNER);

    canvas.DrawPath(path, cornerStroke);
}

// Invert the transform for touch tracking
bitmapScaleMatrix.TryInvert(out inverseBitmapMatrix);
}
...
}

```

The code in the `CroppingRectangle` class bases the cropping rectangle on the pixel size of the bitmap. However, the display of the bitmap by the `PhotoCropperCanvasView` class is scaled based on the size of the display area. The `bitmapScaleMatrix` calculated in the `OnPaintSurface` override maps from the bitmap pixels to the size and position of the bitmap as it is displayed. This matrix is then used to transform the cropping rectangle so that it can be displayed relative to the bitmap.

The last line of the `OnPaintSurface` override takes the inverse of the `bitmapScaleMatrix` and saves it as the `inverseBitmapMatrix` field. This is used for touch processing.

A `TouchEvent` object is instantiated as a field, and the constructor attaches a handler to the `TouchAction` event, but the `TouchEvent` needs to be added to the `Effects` collection of the *parent* of the `SKCanvasView` derivative, so that's done in the `OnParentSet` override:

```

class PhotoCropperCanvasView : SKCanvasView
{
    ...
    const int RADIUS = 100;      // pixel radius of touch hit-test
}

```

```

...
CroppingRectangle croppingRect;
SKMatrix inverseBitmapMatrix;

// Touch tracking
TouchEffect touchEffect = new TouchEffect();
struct TouchPoint
{
    public int CornerIndex { set; get; }
    public SKPoint Offset { set; get; }
}

Dictionary<long, TouchPoint> touchPoints = new Dictionary<long, TouchPoint>();
...

public PhotoCropperCanvasView(SKBitmap bitmap, float? aspectRatio = null)
{
    ...
    touchEffect.TouchAction += OnTouchEffectTouchAction;
}
...

protected override void OnParentSet()
{
    base.OnParentSet();

    // Attach TouchEffect to parent view
    Parent.Effects.Add(touchEffect);
}
...

void OnTouchEffectTouchAction(object sender, TouchEventArgs args)
{
    SKPoint pixelLocation = ConvertToPixel(args.Location);
    SKPoint bitmapLocation = inverseBitmapMatrix.MapPoint(pixelLocation);

    switch (args.Type)
    {
        case Touch ActionType.Pressed:
            // Convert radius to bitmap/cropping scale
            float radius = inverseBitmapMatrix.ScaleX * RADIUS;

            // Find corner that the finger is touching
            int cornerIndex = croppingRect.HitTest(bitmapLocation, radius);

            if (cornerIndex != -1 && !touchPoints.ContainsKey(args.Id))
            {
                TouchPoint touchPoint = new TouchPoint
                {
                    CornerIndex = cornerIndex,
                    Offset = bitmapLocation - croppingRect.Corners[cornerIndex]
                };

                touchPoints.Add(args.Id, touchPoint);
            }
            break;

        case Touch ActionType.Moved:
            if (touchPoints.ContainsKey(args.Id))
            {
                TouchPoint touchPoint = touchPoints[args.Id];
                croppingRect.MoveCorner(touchPoint.CornerIndex,
                    bitmapLocation - touchPoint.Offset);
                InvalidateSurface();
            }
            break;

        case Touch ActionType.Released:
        case Touch ActionType.Cancelled:
            if (touchPoints.ContainsKey(args.Id))
            {
                touchPoints.Remove(args.Id);
            }
    }
}

```

```

        }
        break;
    }

    SKPoint ConvertToPixel(Xamarin.Forms.Point pt)
    {
        return new SKPoint((float)(CanvasSize.Width * pt.X / Width),
                           (float)(CanvasSize.Height * pt.Y / Height));
    }
}

```

The touch events processed by the `TouchAction` handler are in device-independent units. These first need to be converted to pixels using the `ConvertToPixel` method at the bottom of the class, and then converted to `CroppingRectangle` units using `inverseBitmapMatrix`.

For `Pressed` events, the `TouchAction` handler calls the `HitTest` method of `CroppingRectangle`. If this returns an index other than -1, then one of the corners of the cropping rectangle is being manipulated. That index and an offset of the actual touch point from the corner is stored in a `TouchPoint` object and added to the `touchPoints` dictionary.

For the `Moved` event, the `MoveCorner` method of `CroppingRectangle` is called to move the corner, with possible adjustments for the aspect ratio.

At any time, a program using `PhotoCopperCanvasView` can access the `CroppedBitmap` property. This property uses the `Rect` property of the `CroppingRectangle` to create a new bitmap of the cropped size. The version of `DrawBitmap` with destination and source rectangles then extracts a subset of the original bitmap:

```

class PhotoCopperCanvasView : SKCanvasView
{
    ...
    SKBitmap bitmap;
    CroppingRectangle croppingRect;
    ...

    public SKBitmap CroppedBitmap
    {
        get
        {
            SKRect cropRect = croppingRect.Rect;
            SKBitmap croppedBitmap = new SKBitmap((int)cropRect.Width,
                                                   (int)cropRect.Height);
            SKRect dest = new SKRect(0, 0, cropRect.Width, cropRect.Height);
            SKRect source = new SKRect(cropRect.Left, cropRect.Top,
                                       cropRect.Right, cropRect.Bottom);

            using (SKCanvas canvas = new SKCanvas(croppedBitmap))
            {
                canvas.DrawBitmap(bitmap, source, dest);
            }

            return croppedBitmap;
        }
    }
    ...
}

```

Hosting the photo copper canvas view

With those two classes handling the cropping logic, the **Photo Cropping** page in the **SkiaSharpFormsDemos** application has very little work to do. The XAML file instantiates a `Grid` to host the `PhotoCopperCanvasView` and a `Done` button:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="SkiaSharpFormsDemos Bitmaps PhotoCroppingPage"
    Title="Photo Cropping">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Grid x:Name="canvasViewHost"
            Grid.Row="0"
            BackgroundColor="Gray"
            Padding="5" />

        <Button Text="Done"
            Grid.Row="1"
            HorizontalOptions="Center"
            Margin="5"
            Clicked="OnDoneButtonClicked" />
    </Grid>
</ContentPage>

```

The `PhotoCropperCanvasView` cannot be instantiated in the XAML file because it requires a parameter of type `SKBitmap`.

Instead, the `PhotoCropperCanvasView` is instantiated in the constructor of the code-behind file using one of the resource bitmaps:

```

public partial class PhotoCroppingPage : ContentPage
{
    PhotoCropperCanvasView photoCropper;
    SKBitmap croppedBitmap;

    public PhotoCroppingPage ()
    {
        InitializeComponent ();

        SKBitmap bitmap = BitmapExtensions.LoadBitmapResource(GetType(),
            "SkiaSharpFormsDemos.Media.MountainClimbers.jpg");

        photoCropper = new PhotoCropperCanvasView(bitmap);
        canvasViewHost.Children.Add(photoCropper);
    }

    void OnDoneButtonClicked(object sender, EventArgs args)
    {
        croppedBitmap = photoCropper.CroppedBitmap;

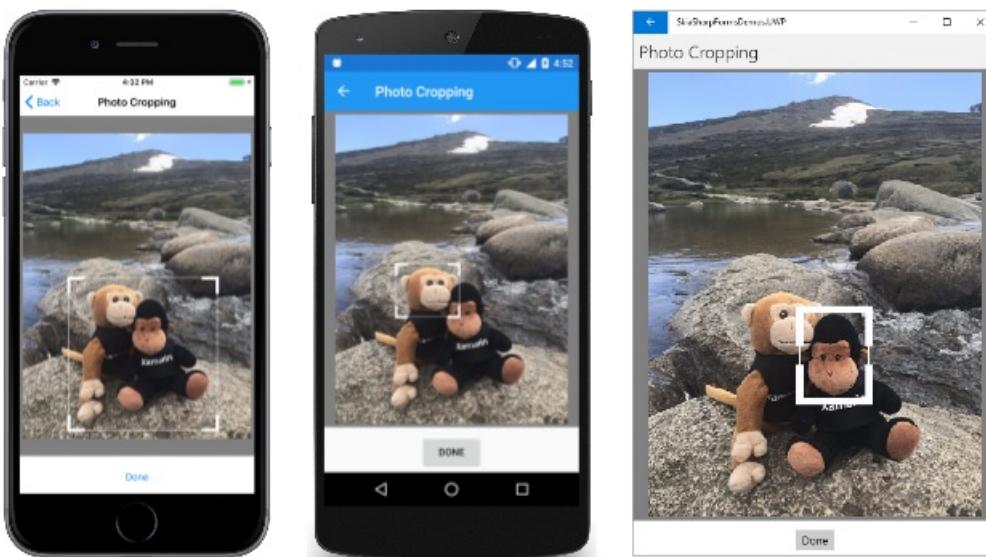
        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

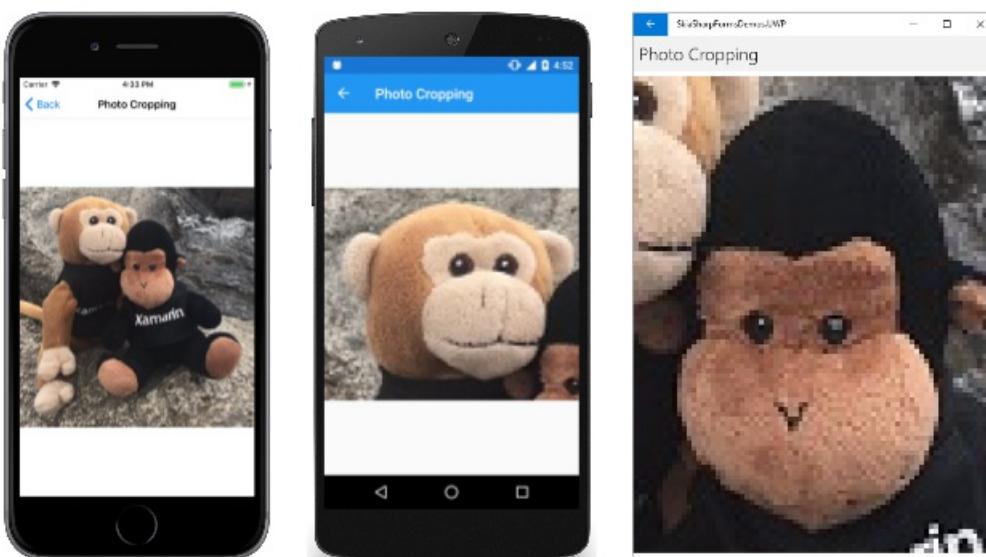
        canvas.Clear();
        canvas.DrawBitmap(croppedBitmap, info.Rect, BitmapStretch.Uniform);
    }
}

```

The user can then manipulate the cropping rectangle:



When a good cropping rectangle has been defined, click the **Done** button. The `Clicked` handler obtains the cropped bitmap from the `CroppedBitmap` property of `PhotoCropperCanvasView`, and replaces all the content of the page with a new `SKCanvasView` object that displays this cropped bitmap:



Try setting the second argument of `PhotoCropperCanvasView` to 1.78f (for example):

```
photoCropper = new PhotoCropperCanvasView(bitmap, 1.78f);
```

You'll see the cropping rectangle restricted to a 16-to-9 aspect ratio characteristic of high-definition television.

Dividing a bitmap into tiles

A Xamarin.Forms version of the famous 14-15 puzzle appeared in Chapter 22 of the book *Creating Mobile Apps with Xamarin.Forms* and can be downloaded as [XamagonXuzzle](#). However, the puzzle becomes more fun (and often more challenging) when it is based on an image from your own photo library.

This version of the 14-15 puzzle is part of the [SkiaSharpFormsDemos](#) application, and consists of a series of pages titled **Photo Puzzle**.

The `PhotoPuzzlePage1.xaml` file consists of a `Button`:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="SkiaSharpFormsDemos Bitmaps PhotoPuzzlePage1"
    Title="Photo Puzzle">

    <Button Text="Pick a photo from your library"
        VerticalOptions="CenterAndExpand"
        HorizontalOptions="CenterAndExpand"
        Clicked="OnPickButtonClicked"/>

</ContentPage>

```

The code-behind file implements a `Clicked` handler that uses the `IPhotoLibrary` dependency service to let the user pick a photo from the photo library:

```

public partial class PhotoPuzzlePage1 : ContentPage
{
    public PhotoPuzzlePage1 ()
    {
        InitializeComponent ();
    }

    async void OnPickButtonClicked(object sender, EventArgs args)
    {
        IPhotoLibrary photoLibrary = DependencyService.Get<IPhotoLibrary>();
        using (Stream stream = await photoLibrary.PickPhotoAsync())
        {
            if (stream != null)
            {
                SKBitmap bitmap = SKBitmap.Decode(stream);

                await Navigation.PushAsync(new PhotoPuzzlePage2(bitmap));
            }
        }
    }
}

```

The method then navigates to `PhotoPuzzlePage2`, passing to the constructor the selected bitmap.

It's possible that the photo selected from the library is not oriented as it appeared in the photo library, but is rotated or upside-down. (This is particularly a problem with iOS devices.) For that reason, `PhotoPuzzlePage2` allows you to rotate the image to a desired orientation. The XAML file contains three buttons labeled **90° Right** (meaning clockwise), **90° Left** (counterclockwise), and **Done**.

The code-behind file implements the bitmap-rotation logic shown in the article [Creating and Drawing on SkiaSharp Bitmaps](#). The user can rotate the image 90 degrees clockwise or counter-clockwise any number of times:

```

public partial class PhotoPuzzlePage2 : ContentPage
{
    SKBitmap bitmap;

    public PhotoPuzzlePage2 (SKBitmap bitmap)
    {
        this.bitmap = bitmap;

        InitializeComponent ();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();
        canvas.DrawBitmap(bitmap, info.Rect, BitmapStretch.Uniform);
    }

    void OnRotateRightButtonClicked(object sender, EventArgs args)
    {
        SKBitmap rotatedBitmap = new SKBitmap(bitmap.Height, bitmap.Width);

        using (SKCanvas canvas = new SKCanvas(rotatedBitmap))
        {
            canvas.Clear();
            canvas.Translate(bitmap.Height, 0);
            canvas.RotateDegrees(90);
            canvas.DrawBitmap(bitmap, new SKPoint());
        }

        bitmap = rotatedBitmap;
        canvasView.InvalidateSurface();
    }

    void OnRotateLeftButtonClicked(object sender, EventArgs args)
    {
        SKBitmap rotatedBitmap = new SKBitmap(bitmap.Height, bitmap.Width);

        using (SKCanvas canvas = new SKCanvas(rotatedBitmap))
        {
            canvas.Clear();
            canvas.Translate(0, bitmap.Width);
            canvas.RotateDegrees(-90);
            canvas.DrawBitmap(bitmap, new SKPoint());
        }

        bitmap = rotatedBitmap;
        canvasView.InvalidateSurface();
    }

    async void OnDoneButtonClicked(object sender, EventArgs args)
    {
        await Navigation.PushAsync(new PhotoPuzzlePage3(bitmap));
    }
}

```

When the user clicks the **Done** button, the `Clicked` handler navigates to `PhotoPuzzlePage3`, passing the final rotated bitmap in the page's constructor.

`PhotoPuzzlePage3` allows the photo to be cropped. The program requires a square bitmap to divide into a 4-by-4 grid of tiles.

The PhotoPuzzlePage3.xaml file contains a `Label`, a `Grid` to host the `PhotoCropperCanvasView`, and another `Done` button:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="SkiaSharpFormsDemos Bitmaps PhotoPuzzlePage3"
    Title="Photo Puzzle">
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <Label Text="Crop the photo to a square"
        Grid.Row="0"
        FontSize="Large"
        HorizontalTextAlignment="Center"
        Margin="5" />

    <Grid x:Name="canvasViewHost"
        Grid.Row="1"
        BackgroundColor="Gray"
        Padding="5" />

    <Button Text="Done"
        Grid.Row="2"
        HorizontalOptions="Center"
        Margin="5"
        Clicked="OnDoneButtonClicked" />
</Grid>
</ContentPage>
```

The code-behind file instantiates the `PhotoCropperCanvasView` with the bitmap passed to its constructor. Notice that a 1 is passed as the second argument to `PhotoCropperCanvasView`. This aspect ratio of 1 forces the cropping rectangle to be a square:

```

public partial class PhotoPuzzlePage3 : ContentPage
{
    PhotoCropperCanvasView photoCropper;

    public PhotoPuzzlePage3(SKBitmap bitmap)
    {
        InitializeComponent ();

        photoCropper = new PhotoCropperCanvasView(bitmap, 1f);
        canvasViewHost.Children.Add(photoCropper);
    }

    async void OnDoneButtonClicked(object sender, EventArgs args)
    {
        SKBitmap croppedBitmap = photoCropper.CroppedBitmap;
        int width = croppedBitmap.Width / 4;
        int height = croppedBitmap.Height / 4;

        ImageSource[] imgSources = new ImageSource[15];

        for (int row = 0; row < 4; row++)
        {
            for (int col = 0; col < 4; col++)
            {
                // Skip the last one!
                if (row == 3 && col == 3)
                    break;

                // Create a bitmap 1/4 the width and height of the original
                SKBitmap bitmap = new SKBitmap(width, height);
                SKRect dest = new SKRect(0, 0, width, height);
                SKRect source = new SKRect(col * width, row * height, (col + 1) * width, (row + 1) * height);

                // Copy 1/16 of the original into that bitmap
                using (SKCanvas canvas = new SKCanvas(bitmap))
                {
                    canvas.DrawBitmap(croppedBitmap, source, dest);
                }

                imgSources[4 * row + col] = (SKBitmapImageSource)bitmap;
            }
        }

        await Navigation.PushAsync(new PhotoPuzzlePage4(imgSources));
    }
}

```

The **Done** button handler obtains the width and height of the cropped bitmap (these two values should be the same) and then divides it into 15 separate bitmaps, each of which is 1/4 the width and height of the original. (The last of the possible 16 bitmaps is not created.) The `DrawBitmap` method with source and destination rectangle allows a bitmap to be created based on subset of a larger bitmap.

Converting to Xamarin.Forms bitmaps

In the `OnDoneButtonClicked` method, the array created for the 15 bitmaps is of type `ImageSource`:

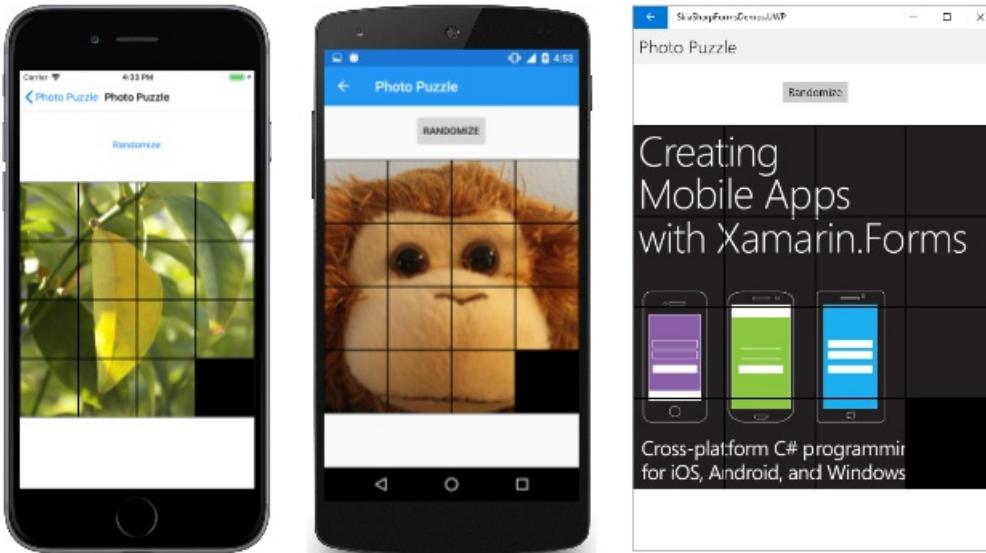
```
ImageSource[] imgSources = new ImageSource[15];
```

`ImageSource` is the Xamarin.Forms base type that encapsulates a bitmap. Fortunately, SkiaSharp allows converting from SkiaSharp bitmaps to Xamarin.Forms bitmaps. The `SkiaSharp.Views.Forms` assembly defines an `SKBitmapImageSource` class that derives from `ImageSource` but can be created based on a SkiaSharp `SKBitmap`.

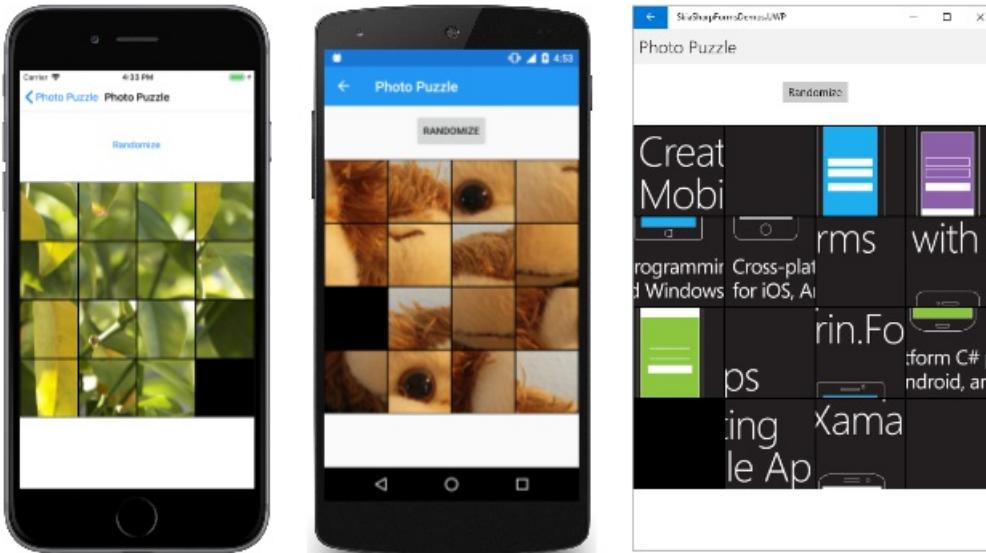
object. `SKBitmapImageSource` even defines conversions between `SKBitmapImageSource` and `SKBitmap`, and that's how `SKBitmap` objects are stored in an array as Xamarin.Forms bitmaps:

```
imgSources[4 * row + col] = (SKBitmapImageSource)bitmap;
```

This array of bitmaps is passed as a constructor to `PhotoPuzzlePage4`. That page is entirely Xamarin.Forms and doesn't use any SkiaSharp. It is very similar to [XamagonXuzzle](#), so it won't be described here, but it displays your selected photo divided into 15 square tiles:



Pressing the **Randomize** button mixes up all the tiles:



Now you can put them back in the correct order. Any tiles in the same row or column as the blank square can be tapped to move them into the blank square.

Related links

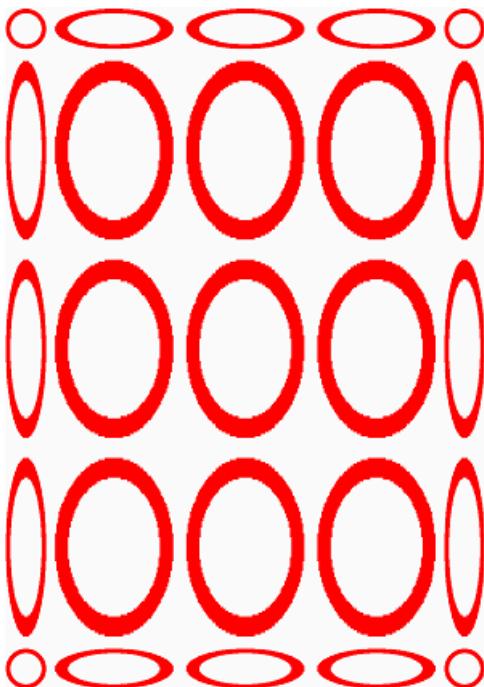
- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Segmented display of SkiaSharp bitmaps

3/5/2021 • 6 minutes to read • [Edit Online](#)

 [Download the sample](#)

The SkiaSharp `SKCanvas` object defines a method named `DrawBitmapNinePatch` and two methods named `DrawBitmapLattice` that are very similar. Both these methods render a bitmap to the size of a destination rectangle, but instead of stretching the bitmap uniformly, they display portions of the bitmap in its pixel dimensions and stretch other parts of the bitmap so that it fits the rectangle:

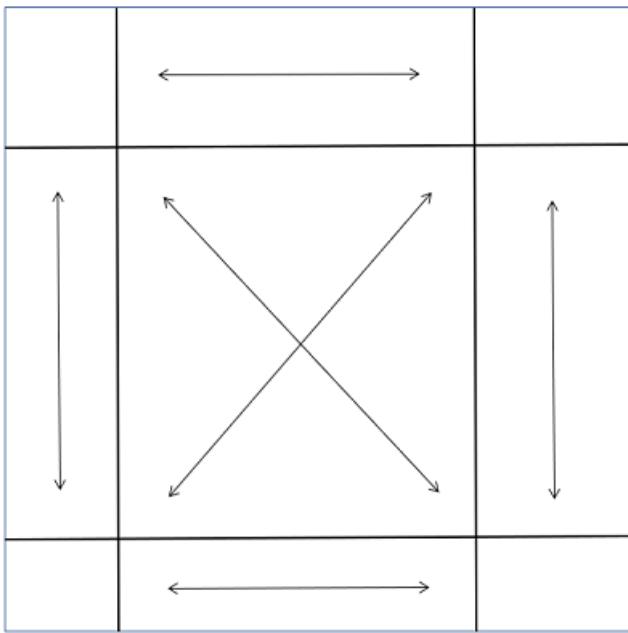


These methods are generally used for rendering bitmaps that form part of user-interface objects such as buttons. When designing a button, generally you want the size of a button to be based on the content of the button, but you probably want the button's border to be the same width regardless of the button's content. That's an ideal application of `DrawBitmapNinePatch`.

`DrawBitmapNinePatch` is a special case of `DrawBitmapLattice` but it is the easier of the two methods to use and understand.

The nine-patch display

Conceptually, `DrawBitmapNinePatch` divides a bitmap into nine rectangles:



The rectangles at the four corners are displayed in their pixel sizes. As the arrows indicate, the other areas on the edges of the bitmap are stretched horizontally or vertically to the area of the destination rectangle. The rectangle in the center is stretched both horizontally and vertically.

If there is not enough space in the destination rectangle to display even the four corners in their pixel dimensions, then they are scaled down to the available size and nothing but the four corners are displayed.

To divide a bitmap into these nine rectangles, it is only necessary to specify the rectangle in the center. This is the syntax of the `DrawBitmapNinePatch` method:

```
canvas.DrawBitmapNinePatch(bitmap, centerRectangle, destRectangle, paint);
```

The center rectangle is relative to the bitmap. It is an `SKRectI` value (the integer version of `SKRect`) and all the coordinates and sizes are in units of pixels. The destination rectangle is relative to the display surface. The `paint` argument is optional.

The [Nine Patch Display](#) page in the [SkiaSharpFormsDemos](#) sample first uses a static constructor to create a public static property of type `SKBitmap`:

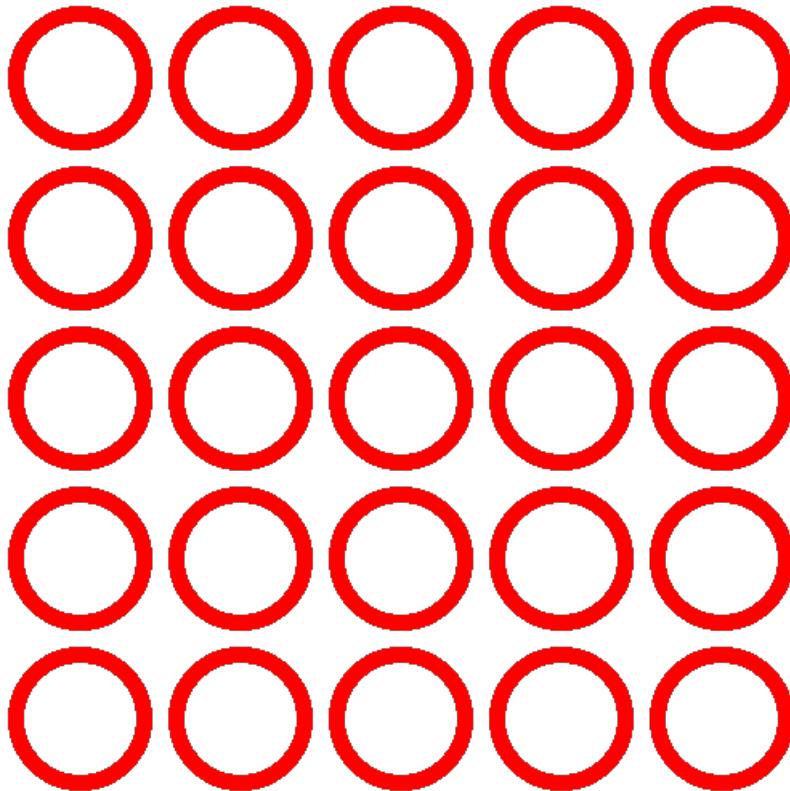
```

public partial class NinePatchDisplayPage : ContentPage
{
    static NinePatchDisplayPage()
    {
        using (SKCanvas canvas = new SKCanvas(FiveByFiveBitmap))
        using (SKPaint paint = new SKPaint
        {
            Style = SKPaintStyle.Stroke,
            Color = SKColors.Red,
            StrokeWidth = 10
        })
        {
            for (int x = 50; x < 500; x += 100)
                for (int y = 50; y < 500; y += 100)
                {
                    canvas.DrawCircle(x, y, 40, paint);
                }
        }
    }

    public static SKBitmap FiveByFiveBitmap { get; } = new SKBitmap(500, 500);
    ...
}

```

Two other pages in this article use that same bitmap. The bitmap is 500 pixels square, and consists of an array of 25 circles, all the same size, each occupying a 100-pixel square area:



The program's instance constructor creates an `SKCanvasView` with a `PaintSurface` handler that uses `DrawBitmapNinePatch` to display the bitmap stretched to its entire display surface:

```

public class NinePatchDisplayPage : ContentPage
{
    ...
    public NinePatchDisplayPage()
    {
        Title = "Nine-Patch Display";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

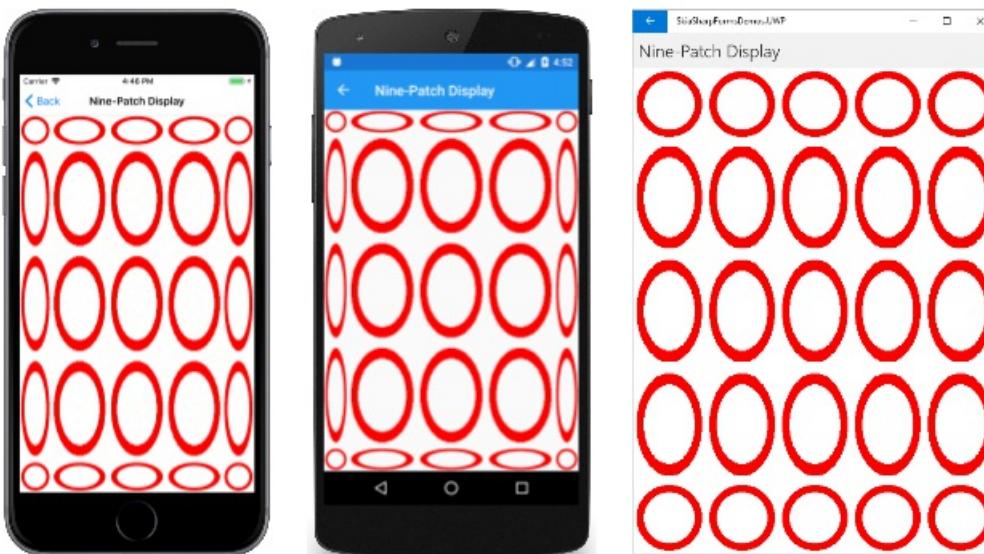
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        SKRectI centerRect = new SKRectI(100, 100, 400, 400);
        canvas.DrawBitmapNinePatch(FiveByFiveBitmap, centerRect, info.Rect);
    }
}

```

The `centerRect` rectangle encompasses the central array of 16 circles. The circles in the corners are displayed in their pixel dimensions, and everything else is stretched accordingly:



The UWP page happens to be 500 pixels wide, and hence displays the top and bottom rows as a series of circles of the same size. Otherwise, all the circles that are not in the corners are stretched to form ellipses.

For a strange display of objects consisting of a combination of circles and ellipses, try defining the center rectangle so that it overlaps rows and columns of circles:

```
SKRectI centerRect = new SKRectI(150, 150, 350, 350);
```

The lattice display

The two `DrawBitmapLattice` methods are similar to `DrawBitmapNinePatch`, but they are generalized for any number of horizontal or vertical divisions. These divisions are defined by arrays of integers corresponding to pixels.

The `DrawBitmapLattice` method with parameters for these arrays of integers does not seem to work. The `DrawBitmapLattice` method with a parameter of type `SKLattice` does work, and that's the one used in the samples shown below.

The `SKLattice` structure defines four properties:

- `XDivs`, an array of integers
- `YDivs`, an array of integers
- `Flags`, an array of `SKLatticeFlags`, an enumeration type
- `Bounds` of type `Nullable<SKRectI>` to specify an optional source rectangle within the bitmap

The `xdivs` array divides the width of the bitmap into vertical strips. The first strip extends from pixel 0 at the left to `xdivs[0]`. This strip is rendered in its pixel width. The second strip extends from `xdivs[0]` to `xdivs[1]`, and is stretched. The third strip extends from `xdivs[1]` to `xdivs[2]` and is rendered in its pixel width. The last strip extends from the last element of the array to the right edge of the bitmap. If the array has an even number of elements, then it's displayed in its pixel width. Otherwise, it's stretched. The total number of vertical strips is one more than the number of elements in the array.

The `ydivs` array is similar. It divides the height of the array into horizontal strips.

Together, the `xdivs` and `ydivs` array divide the bitmap into rectangles. The number of rectangles is equal to the product of the number of horizontal strips and the number of vertical strips.

According to Skia documentation, the `Flags` array contains one element for each rectangle, first the top row of rectangles, then the second row, and so forth. The `Flags` array is of type `SKLatticeFlags`, an enumeration with the following members:

- `Default` with value 0
- `Transparent` with value 1

However, these flags don't seem to work as they are supposed to, and it's best to ignore them. But don't set the `Flags` property to `null`. Set it to an array of `SKLatticeFlags` values large enough to encompass the total number of rectangles.

The **Lattice Nine Patch** page uses `DrawBitmapLattice` to mimic `DrawBitmapNinePatch`. It uses the same bitmap created in `NinePatchDisplayPage`:

```

public class LatticeNinePatchPage : ContentPage
{
    SKBitmap bitmap = NinePatchDisplayPage.FiveByFiveBitmap;

    public LatticeNinePatchPage ()
    {
        Title = "Lattice Nine-Patch";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

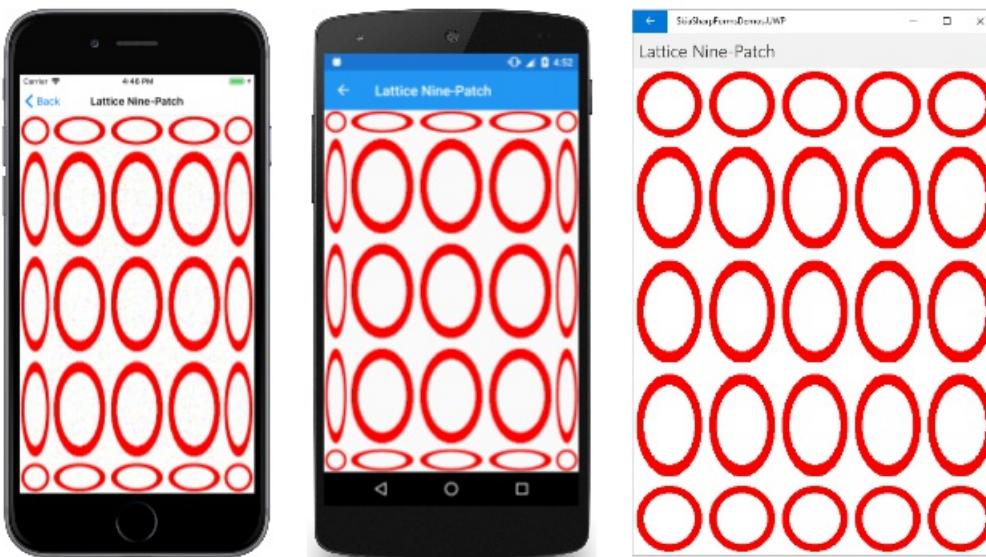
        SKLattice lattice = new SKLattice();
        lattice.XDivs = new int[] { 100, 400 };
        lattice.YDivs = new int[] { 100, 400 };
        lattice.Flags = new SKLatticeFlags[9];

        canvas.DrawBitmapLattice(bitmap, lattice, info.Rect);
    }
}

```

Both the `XDivs` and `YDivs` properties are set to arrays of just two integers, dividing the bitmap into three strips both horizontally and vertically: from pixel 0 to pixel 100 (rendered in the pixel size), from pixel 100 to pixel 400 (stretched), and from pixel 400 to pixel 500 (pixel size). Together, `XDivs` and `YDivs` define a total of 9 rectangles, which is the size of the `Flags` array. Simply creating the array is sufficient to create an array of `SKLatticeFlags.Default` values.

The display is identical to the previous program:



The Lattice Display page divides the bitmap into 16 rectangles:

```

public class LatticeDisplayPage : ContentPage
{
    SKBitmap bitmap = NinePatchDisplayPage.FiveByFiveBitmap;

    public LatticeDisplayPage()
    {
        Title = "Lattice Display";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

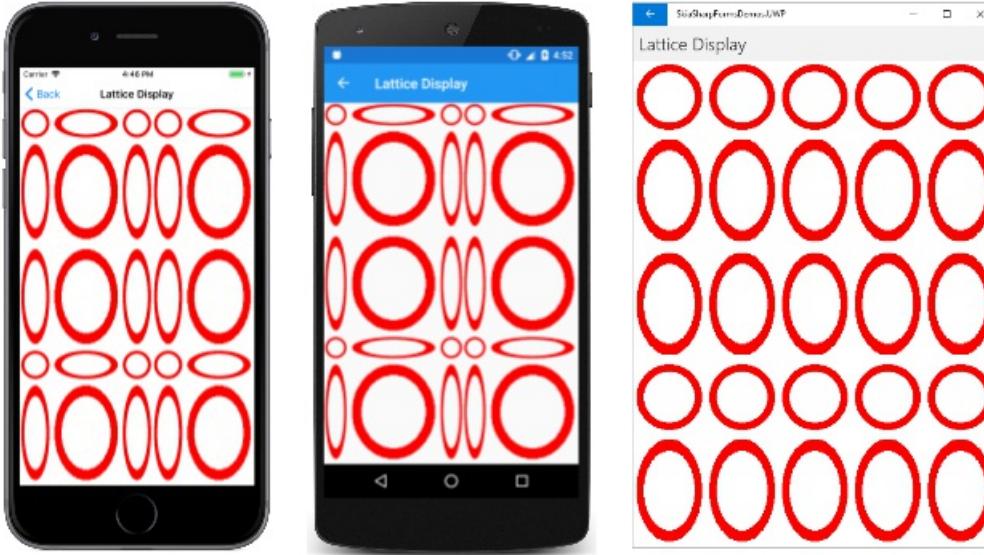
        SKLattice lattice = new SKLattice();
        lattice.XDivs = new int[] { 100, 200, 400 };
        lattice.YDivs = new int[] { 100, 300, 400 };

        int count = (lattice.XDivs.Length + 1) * (lattice.YDivs.Length + 1);
        lattice.Flags = new SKLatticeFlags[count];

        canvas.DrawBitmapLattice(bitmap, lattice, info.Rect);
    }
}

```

The `XDivs` and `YDivs` arrays are somewhat different, causing the display to be not quite as symmetrical as the previous examples:



In the iOS and Android images on the left, only the smaller circles are rendered in their pixel sizes. Everything else is stretched.

The **Lattice Display** page generalizes the creation of the `Flags` array, allowing you to experiment with `XDivs` and `YDivs` more easily. In particular, you'll want to see what happens when you set the first element of the `XDivs` or `YDivs` array to 0.

Related links

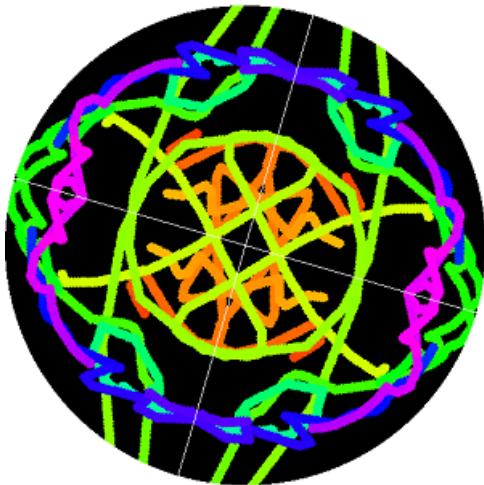
- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Saving SkiaSharp bitmaps to files

3/5/2021 • 15 minutes to read • [Edit Online](#)

 [Download the sample](#)

After a SkiaSharp application has created or modified a bitmap, the application might want to save the bitmap to the user's photo library:



This task encompasses two steps:

- Converting the SkiaSharp bitmap to data in a particular file format, such as JPEG or PNG.
- Saving the result to the photo library using platform-specific code.

File formats and codecs

Most of today's popular bitmap file formats use compression to reduce storage space. The two broad categories of compression techniques are called *lossy* and *lossless*. These terms indicate whether or not the compression algorithm results in the loss of data.

The most popular lossy format was developed by the Joint Photographic Experts Group and is called JPEG. The JPEG compression algorithm analyzes the image using a mathematical tool called the *discrete cosine transform*, and attempts to remove data that is not crucial to preserving the image's visual fidelity. The degree of compression can be controlled with a setting generally referred to as *quality*. Higher quality settings result in larger files.

In contrast, a lossless compression algorithm analyzes the image for repetition and patterns of pixels that can be encoded in a way that reduces the data but does not result in the loss of any information. The original bitmap data can be restored entirely from the compressed file. The primary lossless compressed file format in use today is Portable Network Graphics (PNG).

Generally, JPEG is used for photographs, while PNG is used for images that have been manually or algorithmically generated. Any lossless compression algorithm that reduces the size of some files must necessarily increase the size of others. Fortunately, this increase in size generally only occurs for data that contains a lot of random (or seemingly random) information.

The compression algorithms are complex enough to warrant two terms that describe the compression and decompression processes:

- *decode*— read a bitmap file format and decompress it

- *encode*—compress the bitmap and write to a bitmap file format

The `SKBitmap` class contains several methods named `Decode` that create an `SKBitmap` from a compressed source. All that's required is to supply a filename, stream, or array of bytes. The decoder can determine the file format and hand it off to the proper internal decoding function.

In addition, the `SKCodec` class has two methods named `Create` that can create an `SKcodec` object from a compressed source and allow an application to get more involved in the decoding process. (The `SKCodec` class is shown in the article [Animating SkiaSharp Bitmaps](#) in connection with decoding an animated GIF file.)

When encoding a bitmap, more information is required: The encoder must know the particular file format the application wants to use (JPEG or PNG or something else). If a lossy format is desired, the encode must also know the desired level of quality.

The `SKBitmap` class defines one `Encode` method with the following syntax:

```
public Boolean Encode (SKWStream dst, SKEncodedImageFormat format, Int32 quality)
```

This method is described in more detail shortly. The encoded bitmap is written to a writable stream. (The 'W' in `SKWStream` stands for "writable".) The second and third arguments specify the file format and (for lossy formats) the desired quality ranging from 0 to 100.

In addition, the `SKImage` and `SKPixmap` classes also define `Encode` methods that are somewhat more versatile, and which you might prefer. You can easily create an `SKImage` object from an `SKBitmap` object using the static `SKImage.FromBitmap` method. You can obtain an `SKPixmap` object from an `SKBitmap` object using the `PeekPixels` method.

One of the `Encode` methods defined by `SKImage` has no parameters and automatically saves to a PNG format. That parameterless method is very easy to use.

Platform-specific code for saving bitmap files

When you encode an `SKBitmap` object into a particular file format, generally you'll be left with a stream object of some sort, or an array of data. Some of the `Encode` methods (including the one with no parameters defined by `SKImage`) return an `SKData` object, which can be converted to an array of bytes using the `ToByteArray` method. This data must then be saved to a file.

Saving to a file in application local storage is quite easy because you can use standard `System.IO` classes and methods for this task. This technique is demonstrated in the article [Animating SkiaSharp Bitmaps](#) in connection with animating a series of bitmaps of the Mandelbrot set.

If you want the file to be shared by other applications, it must be saved to the user's photo library. This task requires platform-specific code and the use of the `Xamarin.Forms DependencyService`.

The `SkiaSharpFormsDemo` project in the `SkiaSharpFormsDemos` application defines an `IPhotoLibrary` interface used with the `DependencyService` class. This defines the syntax of a `SavePhotoAsync` method:

```
public interface IPhotoLibrary
{
    Task<Stream> PickPhotoAsync();

    Task<bool> SavePhotoAsync(byte[] data, string folder, string filename);
}
```

This interface also defines the `PickPhotoAsync` method, which is used to open the platform-specific file-picker for the device's photo library.

For `SavePhotoAsync`, the first argument is an array of bytes that contains the bitmap already encoded into a particular file format, such as JPEG or PNG. It's possible that an application might want to isolate all the bitmaps it creates into a particular folder, which is specified in the next parameter, followed by the file name. The method returns a Boolean indicating success or not.

The following sections discuss how `SavePhotoAsync` is implemented on each platform.

The iOS implementation

The iOS implementation of `SavePhotoAsync` uses the `SaveToPhotosAlbum` method of `UIImage`:

```
public class PhotoLibrary : IPhotoLibrary
{
    ...
    public Task<bool> SavePhotoAsync(byte[] data, string folder, string filename)
    {
        NSData nsData = NSData.FromArray(data);
        UIImage image = new UIImage(nsData);
        TaskCompletionSource<bool> taskCompletionSource = new TaskCompletionSource<bool>();

        image.SaveToPhotosAlbum((UIImage img, NSError error) =>
        {
            taskCompletionSource.SetResult(error == null);
        });

        return taskCompletionSource.Task;
    }
}
```

Unfortunately, there is no way to specify a file name or folder for the image.

The `Info.plist` file in the iOS project requires a key indicating that it adds images to the photo library:

```
<key>NSPhotoLibraryAddUsageDescription</key>
<string>SkiaSharp Forms Demos adds images to your photo library</string>
```

Watch out! The permission key for simply accessing the photo library is very similar but not the same:

```
<key>NSPhotoLibraryUsageDescription</key>
<string>SkiaSharp Forms Demos accesses your photo library</string>
```

The Android implementation

The Android implementation of `SavePhotoAsync` first checks if the `folder` argument is `null` or an empty string. If so, then the bitmap is saved in the root directory of the photo library. Otherwise, the folder is obtained, and if it doesn't exist, it is created:

```

public class PhotoLibrary : IPhotoLibrary
{
    ...
    public async Task<bool> SavePhotoAsync(byte[] data, string folder, string filename)
    {
        try
        {
            File picturesDirectory =
Environment.GetExternalStoragePublicDirectory(Environment.DirectoryPictures);
            File folderDirectory = picturesDirectory;

            if (!string.IsNullOrEmpty(folder))
            {
                folderDirectory = new File(picturesDirectory, folder);
                folderDirectory.Mkdirs();
            }

            using (File bitmapFile = new File(folderDirectory, filename))
            {
                bitmapFile.CreateNewFile();

                using (FileOutputStream outputStream = new FileOutputStream(bitmapFile))
                {
                    await outputStream.WriteAsync(data);
                }

                // Make sure it shows up in the Photos gallery promptly.
                MediaScannerConnection.ScanFile(MainActivity.Instance,
                    new string[] { bitmapFile.Path },
                    new string[] { "image/png", "image/jpeg" }, null);
            }
        }
        catch
        {
            return false;
        }

        return true;
    }
}

```

The call to `MediaScannerConnection.ScanFile` isn't strictly required, but if you're testing your program by immediately checking the photo library, it helps a lot by updating the library gallery view.

The `AndroidManifest.xml` file requires the following permission tag:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

The UWP implementation

The UWP implementation of `SavePhotoAsync` is very similar in structure to the Android implementation:

```

public class PhotoLibrary : IPhotoLibrary
{
    ...
    public async Task<bool> SavePhotoAsync(byte[] data, string folder, string filename)
    {
        StorageFolder picturesDirectory = KnownFolders.PicturesLibrary;
        StorageFolder folderDirectory = picturesDirectory;

        // Get the folder or create it if necessary
        if (!string.IsNullOrEmpty(folder))
        {
            try
            {
                folderDirectory = await picturesDirectory.GetFolderAsync(folder);
            }
            catch
            { }

            if (folderDirectory == null)
            {
                try
                {
                    folderDirectory = await picturesDirectory.CreateFolderAsync(folder);
                }
                catch
                {
                    return false;
                }
            }
        }

        try
        {
            // Create the file.
            StorageFile storageFile = await folderDirectory.CreateFileAsync(filename,
                CreationCollisionOption.GenerateUniqueName);

            // Convert byte[] to Windows buffer and write it out.
            IBuffer buffer = WindowsRuntimeBuffer.Create(data, 0, data.Length, data.Length);
            await FileIO.WriteBufferAsync(storageFile, buffer);
        }
        catch
        {
            return false;
        }

        return true;
    }
}

```

The Capabilities section of the `Package.appxmanifest` file requires **Pictures Library**.

Exploring the image formats

Here's the `Encode` method of `SKImage` again:

```
public Boolean Encode (SKWStream dst, SKEncodedImageFormat format, Int32 quality)
```

`SKEncodedImageFormat` is an enumeration with members that refer to eleven bitmap file formats, some of which are rather obscure:

- `Astc` — Adaptive Scalable Texture Compression

- `Bmp` — Windows Bitmap
- `Dng` — Adobe Digital Negative
- `Gif` — Graphics Interchange Format
- `Ico` — Windows icon images
- `Jpeg` — Joint Photographic Experts Group
- `Ktx` — Khronos texture format for OpenGL
- `Pkm` — Pokémon save file
- `Png` — Portable Network Graphics
- `Wbmp` — Wireless Application Protocol Bitmap Format (1 bit per pixel)
- `Webp` — Google WebP format

As you'll see shortly, only three of these file formats (`Jpeg`, `Png`, and `Webp`) are actually supported by SkiaSharp.

To save an `SKBitmap` object named `bitmap` to the user's photo library, you also need a member of the `SKEncodedImageFormat` enumeration named `imageFormat` and (for lossy formats) an integer `quality` variable. You can use the following code to save that bitmap to a file with the name `filename` in the `folder` folder:

```
using (MemoryStream memStream = new MemoryStream())
using (SKManagedWStream wstream = new SKManagedWStream(memStream))
{
    bitmap.Encode(wstream, imageFormat, quality);
    byte[] data = memStream.ToArray();

    // Check the data array for content!

    bool success = await DependencyService.Get<IPhotoLibrary>().SavePhotoAsync(data, folder, filename);

    // Check return value for success!
}
```

The `SKManagedWStream` class derives from `skwstream` (which stands for "writable stream"). The `Encode` method writes the encoded bitmap file into that stream. The comments in that code refer to some error checking you might need to perform.

The [Save File Formats](#) page in the [SkiaSharpFormsDemos](#) application uses similar code to allow you to experiment with saving a bitmap in the various formats.

The XAML file contains an `SKCanvasView` that displays a bitmap, while the rest of the page contains everything the application needs to call the `Encode` method of `SKBitmap`. It has a `Picker` for a member of the `SKEncodedImageFormat` enumeration, a `Slider` for the quality argument for lossy bitmap formats, two `Entry` views for a filename and folder name, and a `Button` for saving the file.

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:skia="clr-namespace:SkiaSharp;assembly=SkiaSharp"
             xmlns:skiaforms="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
             x:Class="SkiaSharpFormsDemos.Banners.SaveFileFormatsPage"
             Title="Save Bitmap Formats">

    <StackLayout Margin="10">
        <skiaforms:SKCanvasView PaintSurface="OnCanvasViewPaintSurface"
                                  VerticalOptions="FillAndExpand" />

        <Picker x:Name="formatPicker"
               Title="image format"
               SelectedIndexChanged="OnFormatPickerChanged">
            <Picker.ItemsSource>
```

```

    <Picker x:Name="formatPicker">
        <x:Array Type="{x:Type skia:SKEncodedImageFormat}">
            <x:Static Member="skia:SKEncodedImageFormat.Astc" />
            <x:Static Member="skia:SKEncodedImageFormat.Bmp" />
            <x:Static Member="skia:SKEncodedImageFormat.Dng" />
            <x:Static Member="skia:SKEncodedImageFormat.Gif" />
            <x:Static Member="skia:SKEncodedImageFormat.Ico" />
            <x:Static Member="skia:SKEncodedImageFormat.Jpeg" />
            <x:Static Member="skia:SKEncodedImageFormat.Ktx" />
            <x:Static Member="skia:SKEncodedImageFormat.Pkm" />
            <x:Static Member="skia:SKEncodedImageFormat.Png" />
            <x:Static Member="skia:SKEncodedImageFormat.Wbmp" />
            <x:Static Member="skia:SKEncodedImageFormat.Webp" />
        </x:Array>
    </Picker.ItemsSource>
</Picker>

<Slider x:Name="qualitySlider"
        Maximum="100"
        Value="50" />

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

<StackLayout Orientation="Horizontal">
    <Label Text="Folder Name: "
          VerticalOptions="Center" />

    <Entry x:Name="folderNameEntry"
           Text="SaveFileFormats"
           HorizontalOptions="FillAndExpand" />
</StackLayout>

<StackLayout Orientation="Horizontal">
    <Label Text="File Name: "
          VerticalOptions="Center" />

    <Entry x:Name="fileNameEntry"
           Text="Sample.xxx"
           HorizontalOptions="FillAndExpand" />
</StackLayout>

<Button Text="Save"
        Clicked="OnButtonClicked">
    <Button.Triggers>
        <DataTrigger TargetType="Button"
                     Binding="{Binding Source={x:Reference formatPicker},
                                     Path=SelectedIndex}"
                     Value="-1">
            <Setter Property="IsEnabled" Value="False" />
        </DataTrigger>
        <DataTrigger TargetType="Button"
                     Binding="{Binding Source={x:Reference fileNameEntry},
                                     Path=Text.Length}"
                     Value="0">
            <Setter Property="IsEnabled" Value="False" />
        </DataTrigger>
    </Button.Triggers>
</Button>

<Label x:Name="statusLabel"
       Text="OK"
       Margin="10, 0" />
</StackLayout>
</ContentPage>

```

The code-behind file loads a bitmap resource and uses the `SKCanvasView` to display it. That bitmap never changes. The `SelectedIndexChanged` handler for the `Picker` modifies the filename with an extension that is the same as the enumeration member:

```
public partial class SaveFileFormatsPage : ContentPage
{
    SKBitmap bitmap = BitmapExtensions.LoadBitmapResource(typeof(SaveFileFormatsPage),
        "SkiaSharpFormsDemos.Media.MonkeyFace.png");

    public SaveFileFormatsPage ()
    {
        InitializeComponent ();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        args.Surface.Canvas.DrawBitmap(bitmap, args.Info.Rect, BitmapStretch.Uniform);
    }

    void OnFormatPickerChanged(object sender, EventArgs args)
    {
        if (formatPicker.SelectedIndex != -1)
        {
            SKEncodedImageFormat imageFormat = (SKEncodedImageFormat)formatPicker.SelectedItem;
            fileNameEntry.Text = Path.ChangeExtension(fileNameEntry.Text, imageFormat.ToString());
            statusLabel.Text = "OK";
        }
    }

    async void OnButtonClicked(object sender, EventArgs args)
    {
        SKEncodedImageFormat imageFormat = (SKEncodedImageFormat)formatPicker.SelectedItem;
        int quality = (int)qualitySlider.Value;

        using (MemoryStream memStream = new MemoryStream())
        using (SKManagedWStream wstream = new SKManagedWStream(memStream))
        {
            bitmap.Encode(wstream, imageFormat, quality);
            byte[] data = memStream.ToArray();

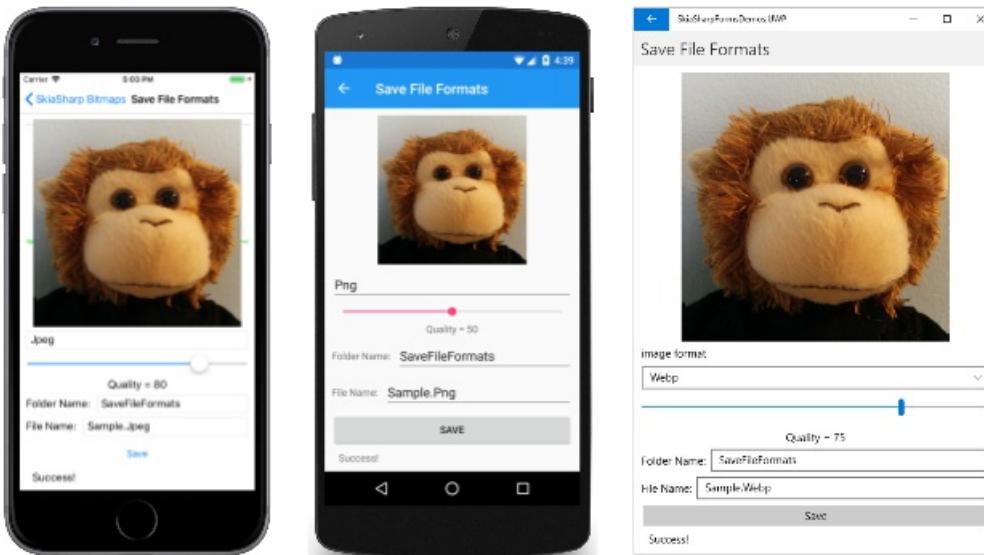
            if (data == null)
            {
                statusLabel.Text = "Encode returned null";
            }
            else if (data.Length == 0)
            {
                statusLabel.Text = "Encode returned empty array";
            }
            else
            {
                bool success = await DependencyService.Get<IPhotoLibrary>().
                    SavePhotoAsync(data, folderNameEntry.Text, fileNameEntry.Text);

                if (!success)
                {
                    statusLabel.Text = "SavePhotoAsync return false";
                }
                else
                {
                    statusLabel.Text = "Success!";
                }
            }
        }
    }
}
```

The `clicked` handler for the `Button` does all the real work. It obtains two arguments for `Encode` from the `Picker` and `Slider`, and then uses the code shown earlier to create an `SKManagedWStream` for the `Encode` method. The two `Entry` views furnish folder and file names for the `SavePhotoAsync` method.

Most of this method is devoted to handling problems or errors. If `Encode` creates an empty array, it means that the particular file format isn't supported. If `SavePhotoAsync` returns `false`, then the file wasn't successfully saved.

Here is the program running:



That screenshot shows the only three formats that are supported on these platforms:

- JPEG
- PNG
- WebP

For all the other formats, the `Encode` method writes nothing into the stream and the resultant byte array is empty.

The bitmap that the **Save File Formats** page saves is 600-pixels square. With 4 bytes per pixel, that's a total of 1,440,000 bytes in memory. The following table shows the file size for various combinations of file format and quality:

FORMAT	QUALITY	SIZE
PNG	N/A	492K
JPEG	0	2.95K
	50	22.1K
	100	206K
WebP	0	2.71K
	50	11.9K
	100	101K

You can experiment with various quality settings and examine the results.

Saving finger-paint art

One common use of a bitmap is in drawing programs, where it functions as something called a *shadow bitmap*. All the drawing is retained on the bitmap, which is then displayed by the program. The bitmap also comes in handy for saving the drawing.

The [Finger Painting in SkiaSharp](#) article demonstrated how to use touch tracking to implement a primitive finger-painting program. The program supported only one color and only one stroke width, but it retained the entire drawing in a collection of `SKPath` objects.

The [Finger Paint with Save](#) page in the [SkiaSharpFormsDemos](#) sample also retains the entire drawing in a collection of `SKPath` objects, but it also renders the drawing on a bitmap, which it can save to your photo library.

Much of this program is similar to the original [Finger Paint](#) program. One enhancement is that the XAML file now instantiates buttons labeled **Clear** and **Save**:

```
<?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:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    xmlns:tt="clr-namespace:TouchTracking"
    x:Class="SkiaSharpFormsDemos.Bitsmaps.FingerPaintSavePage"
    Title="Finger Paint Save">

    <StackLayout>
        <Grid BackgroundColor="White"
            VerticalOptions="FillAndExpand">
            <skia:SKCanvasView x:Name="canvasView"
                PaintSurface="OnCanvasViewPaintSurface" />
            <Grid.Effects>
                <tt:TouchEffect Capture="True"
                    TouchAction="OnTouchEffectAction" />
            </Grid.Effects>
        </Grid>
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
        </Grid>
        <Button Text="Clear"
            Grid.Row="0"
            Margin="50, 5"
            Clicked="OnClearButtonClicked" />
        <Button Text="Save"
            Grid.Row="1"
            Margin="50, 5"
            Clicked="OnSaveButtonClicked" />
    </StackLayout>
</ContentPage>
```

The code-behind file maintains a field of type `SKBitmap` named `saveBitmap`. This bitmap is created or recreated in the `PaintSurface` handler whenever the size of the display surface changes. If the bitmap needs to be recreated, the contents of the existing bitmap are copied to the new bitmap so that everything is retained no matter how the display surface changes in size:

```

public partial class FingerPaintSavePage : ContentPage
{
    ...
    SKBitmap saveBitmap;

    public FingerPaintSavePage ()
    {
        InitializeComponent ();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        // Create bitmap the size of the display surface
        if (saveBitmap == null)
        {
            saveBitmap = new SKBitmap(info.Width, info.Height);
        }
        // Or create new bitmap for a new size of display surface
        else if (saveBitmap.Width < info.Width || saveBitmap.Height < info.Height)
        {
            SKBitmap newBitmap = new SKBitmap(Math.Max(saveBitmap.Width, info.Width),
                                              Math.Max(saveBitmap.Height, info.Height));

            using (SKCanvas newCanvas = new SKCanvas(newBitmap))
            {
                newCanvas.Clear();
                newCanvas.DrawBitmap(saveBitmap, 0, 0);
            }

            saveBitmap = newBitmap;
        }

        // Render the bitmap
        canvas.Clear();
        canvas.DrawBitmap(saveBitmap, 0, 0);
    }
    ...
}

```

The drawing done by the `PaintSurface` handler occurs at the very end, and consists solely of rendering the bitmap.

The touch processing is similar to the earlier program. The program maintains two collections, `inProgressPaths` and `completedPaths`, that contain everything the user has drawn since the last time the display was cleared. For each touch event, the `OnTouchEffectAction` handler calls `UpdateBitmap`:

```

public partial class FingerPaintSavePage : ContentPage
{
    Dictionary<long, SKPath> inProgressPaths = new Dictionary<long, SKPath>();
    List<SKPath> completedPaths = new List<SKPath>();

    SKPaint paint = new SKPaint
    {
        Style = SKPaintStyle.Stroke,
        Color = SKColors.Blue,
        StrokeWidth = 10,
        StrokeCap = SKStrokeCap.Round,
        StrokeJoin = SKStrokeJoin.Round
    };
    ...

    void OnTouchEffectAction(object sender, TouchEventArgs args)

```

```

    {
        switch (args.Type)
        {
            case TouchActionType.Pressed:
                if (!inProgressPaths.ContainsKey(args.Id))
                {
                    SKPath path = new SKPath();
                    path.MoveTo(ConvertToPixel(args.Location));
                    inProgressPaths.Add(args.Id, path);
                    UpdateBitmap();
                }
                break;

            case TouchActionType.Moved:
                if (inProgressPaths.ContainsKey(args.Id))
                {
                    SKPath path = inProgressPaths[args.Id];
                    path.LineTo(ConvertToPixel(args.Location));
                    UpdateBitmap();
                }
                break;

            case TouchActionType.Released:
                if (inProgressPaths.ContainsKey(args.Id))
                {
                    completedPaths.Add(inProgressPaths[args.Id]);
                    inProgressPaths.Remove(args.Id);
                    UpdateBitmap();
                }
                break;

            case TouchActionType.Cancelled:
                if (inProgressPaths.ContainsKey(args.Id))
                {
                    inProgressPaths.Remove(args.Id);
                    UpdateBitmap();
                }
                break;
        }
    }

    SKPoint ConvertToPixel(Point pt)
    {
        return new SKPoint((float)(canvasView.CanvasSize.Width * pt.X / canvasView.Width),
                           (float)(canvasView.CanvasSize.Height * pt.Y / canvasView.Height));
    }

    void UpdateBitmap()
    {
        using (SKCanvas saveBitmapCanvas = new SKCanvas(saveBitmap))
        {
            saveBitmapCanvas.Clear();

            foreach (SKPath path in completedPaths)
            {
                saveBitmapCanvas.DrawPath(path, paint);
            }

            foreach (SKPath path in inProgressPaths.Values)
            {
                saveBitmapCanvas.DrawPath(path, paint);
            }
        }

        canvasView.InvalidateSurface();
    }
    ...
}

```

The `UpdateBitmap` method redraws `saveBitmap` by creating a new `SKCanvas`, clearing it, and then rendering all the paths on the bitmap. It concludes by invalidating `canvasView` so that the bitmap can be drawn on the display.

Here are the handlers for the two buttons. The **Clear** button clears both path collections, updates `saveBitmap` (which results in clearing the bitmap), and invalidates the `SKCanvasView`:

```
public partial class FingerPaintSavePage : ContentPage
{
    ...
    void OnClearButtonClicked(object sender, EventArgs args)
    {
        completedPaths.Clear();
        inProgressPaths.Clear();
        UpdateBitmap();
        canvasView.InvalidateSurface();
    }

    async void OnSaveButtonClicked(object sender, EventArgs args)
    {
        using (SKImage image = SKImage.FromBitmap(saveBitmap))
        {
            SKData data = image.Encode();
            DateTime dt = DateTime.Now;
            string filename = String.Format("FingerPaint-{0:D4}{1:D2}{2:D2}-{3:D2}{4:D2}{5:D2}{6:D3}.png",
                dt.Year, dt.Month, dt.Day, dt.Hour, dt.Minute, dt.Second,
                dt.Millisecond);

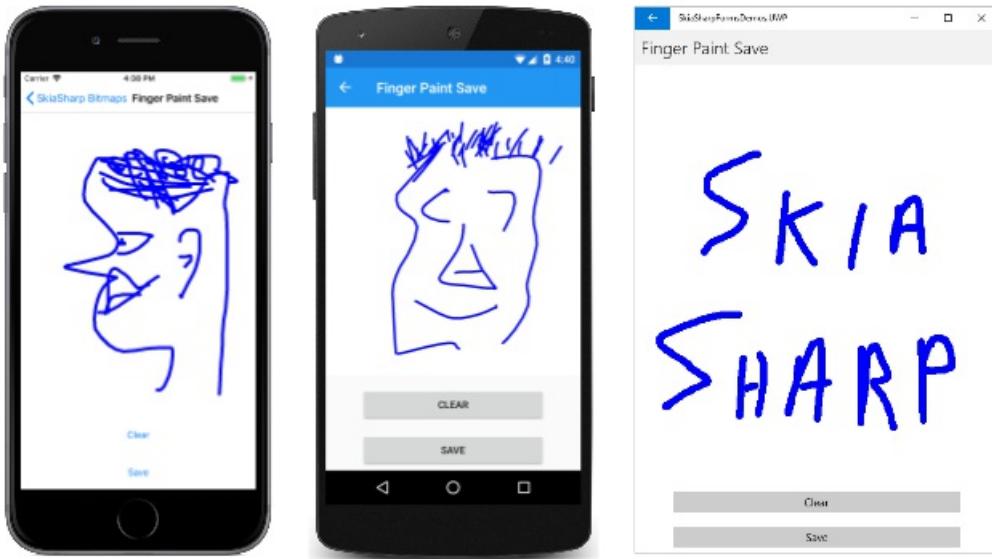
            IPhotoLibrary photoLibrary = DependencyService.Get<IPhotoLibrary>();
            bool result = await photoLibrary.SavePhotoAsync(data.ToArray(), "FingerPaint", filename);

            if (!result)
            {
                await DisplayAlert("FingerPaint", "Artwork could not be saved. Sorry!", "OK");
            }
        }
    }
}
```

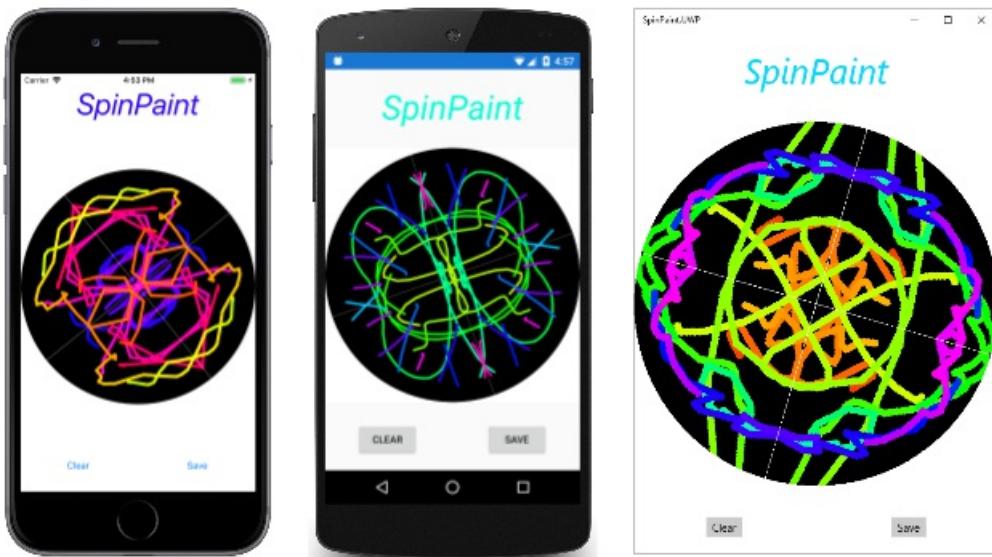
The **Save** button handler uses the simplified `Encode` method from `SKImage`. This method encodes using the PNG format. The `SKImage` object is created based on `saveBitmap`, and the `SKData` object contains the encoded PNG file.

The `ToArray` method of `SKData` obtains an array of bytes. This is what is passed to the `SavePhotoAsync` method, along with a fixed folder name, and a unique filename constructed from the current date and time.

Here's the program in action:



A very similar technique is used in the [SpinPaint](#) sample. This is also a finger-painting program except that the user paints on a spinning disk that then reproduces the designs on its other four quadrants. The color of the finger paint changes as the disk is spinning:



The `Save` button of [SpinPaint](#) class is similar to [Finger Paint](#) in that it saves the image to a fixed folder name (`SpainPaint`) and a filename constructed from the date and time.

Related links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)
- [SpinPaint \(sample\)](#)

Accessing SkiaSharp bitmap pixel bits

3/5/2021 • 24 minutes to read • [Edit Online](#)

 [Download the sample](#)

As you saw in the article [Saving SkiaSharp bitmaps to files](#), bitmaps are generally stored in files in a compressed format, such as JPEG or PNG. In contrast, a SkiaSharp bitmap stored in memory is not compressed. It is stored as a sequential series of pixels. This uncompressed format facilitates the transfer of bitmaps to a display surface.

The memory block occupied by a SkiaSharp bitmap is organized in a very straightforward manner: It begins with the first row of pixels, from left to right, and then continues with the second row. For full-color bitmaps, each pixel consists of four bytes, which means that the total memory space required by the bitmap is four times the product of its width and height.

This article describes how an application can get access to those pixels, either directly by accessing the bitmap's pixel memory block, or indirectly. In some instances, a program might want to analyze the pixels of an image and construct a histogram of some sort. More commonly, applications can construct unique images by algorithmically creating the pixels that make up the bitmap:



The techniques

SkiaSharp provides several techniques for accessing a bitmap's pixel bits. Which one you choose is usually a compromise between coding convenience (which is related to maintenance and ease of debugging) and performance. In most cases, you'll use one of the following methods and properties of `SKBitmap` for accessing the bitmap's pixels:

- The `GetPixel` and `SetPixel` methods allow you to obtain or set the color of a single pixel.
- The `Pixels` property obtains an array of pixel colors for the entire bitmap, or sets the array of colors.
- `GetPixels` returns the address of the pixel memory used by the bitmap.
- `SetPixels` replaces the address of the pixel memory used by the bitmap.

You can think of the first two techniques as "high level" and the second two as "low level." There are some other

methods and properties that you can use, but these are the most valuable.

To allow you to see the performance differences between these techniques, the [SkiaSharpFormsDemos](#) application contains a page named **Gradient Bitmap** that creates a bitmap with pixels that combine red and blue shades to create a gradient. The program creates eight different copies of this bitmap, all using different techniques for setting the bitmap pixels. Each of these eight bitmaps is created in a separate method that also sets a brief text description of the technique and calculates the time required to set all the pixels. Each method loops through the pixel-setting logic 100 times to get a better estimate of the performance.

The SetPixel method

If you only need to set or get several individual pixels, the `SetPixel` and `GetPixel` methods are ideal. For each of these two methods, you specify the integer column and row. Regardless of the pixel format, these two methods let you obtain or set the pixel as an `SKColor` value:

```
bitmap.SetPixel(col, row, color);

SKColor color = bitmap.GetPixel(col, row);
```

The `col` argument must range from 0 to one less than the `Width` property of the bitmap, and `row` ranges from 0 to one less than the `Height` property.

Here's the method in **Gradient Bitmap** that sets the contents of a bitmap using the `SetPixel` method. The bitmap is 256 by 256 pixels, and the `for` loops are hard-coded with the range of values:

```
public class GradientBitmapPage : ContentPage
{
    const int REPS = 100;

    Stopwatch stopwatch = new Stopwatch();
    ...

    SKBitmap FillBitmapSetPixel(out string description, out int milliseconds)
    {
        description = "SetPixel";
        SKBitmap bitmap = new SKBitmap(256, 256);

        stopwatch.Restart();

        for (int rep = 0; rep < REPS; rep++)
            for (int row = 0; row < 256; row++)
                for (int col = 0; col < 256; col++)
                {
                    bitmap.SetPixel(col, row, new SKColor((byte)col, 0, (byte)row));
                }

        milliseconds = (int)stopwatch.ElapsedMilliseconds;
        return bitmap;
    }
    ...
}
```

The color set for each pixel has a red component equal to the bitmap column, and a blue component equal to the row. The resultant bitmap is black at the upper-left, red at the upper-right, blue at the lower-left, and magenta at the lower-right, with gradients elsewhere.

The `SetPixel` method is called 65,536 times, and regardless how efficient this method might be, it's generally not a good idea to make that many API calls if an alternative is available. Fortunately, there are several alternatives.

The Pixels property

`SKBitmap` defines a `Pixels` property that returns an array of `SKColor` values for the entire bitmap. You can also use `Pixels` to set an array of color values for the bitmap:

```
SKColor[] pixels = bitmap.Pixels;  
  
bitmap.Pixels = pixels;
```

The pixels are arranged in the array starting with the first row, from left to right, then the second row, and so forth. The total number of colors in the array is equal to the product of the bitmap width and height.

Although this property appears to be efficient, keep in mind that the pixels are being copied from the bitmap into the array, and from the array back into the bitmap, and the pixels are converted from and to `SKColor` values.

Here's the method in the `GradientBitmapPage` class that sets the bitmap using the `Pixels` property. The method allocates an `SKColor` array of the required size, but it could have used the `Pixels` property to create that array:

```
SKBitmap FillBitmapPixelsProp(out string description, out int milliseconds)  
{  
    description = "Pixels property";  
    SKBitmap bitmap = new SKBitmap(256, 256);  
  
    stopwatch.Restart();  
  
    SKColor[] pixels = new SKColor[256 * 256];  
  
    for (int rep = 0; rep < REPS; rep++)  
        for (int row = 0; row < 256; row++)  
            for (int col = 0; col < 256; col++)  
            {  
                pixels[256 * row + col] = new SKColor((byte)col, 0, (byte)row);  
            }  
  
    bitmap.Pixels = pixels;  
  
    milliseconds = (int)stopwatch.ElapsedMilliseconds;  
    return bitmap;  
}
```

Notice that the index of the `pixels` array needs to be calculated from the `row` and `col` variables. The row is multiplied by the number of pixels in each row (256 in this case), and then the column is added.

`SKBitmap` also defines a similar `Bytes` property, which returns a byte array for the entire bitmap, but it is more cumbersome for full-color bitmaps.

The `GetPixels` pointer

Potentially the most powerful technique to access the bitmap pixels is `GetPixels`, not to be confused with the `GetPixel` method or the `Pixels` property. You'll immediately notice a difference with `GetPixels` in that it returns something not very common in C# programming:

```
IntPtr pixelsAddr = bitmap.GetPixels();
```

The .NET `IntPtr` type represents a pointer. It is called `IntPtr` because it is the length of an integer on the native processor of the machine on which the program is run, generally either 32 bits or 64 bits in length. The `IntPtr` that `GetPixels` returns is the address of the actual block of memory that the bitmap object is using to store its pixels.

You can convert the `IntPtr` into a C# pointer type using the `ToPointer` method. The C# pointer syntax is the same as C and C++:

```
byte* ptr = (byte*)pixelsAddr.ToPointer();
```

The `ptr` variable is of type *byte pointer*. This `ptr` variable allows you to access the individual bytes of memory that are used to store the bitmap's pixels. You use code like this to read a byte from this memory or write a byte to the memory:

```
byte pixelComponent = *ptr;  
  
*ptr = pixelComponent;
```

In this context, the asterisk is the C# *indirection operator* and is used to reference the contents of the memory pointed to by `ptr`. Initially, `ptr` points to the first byte of the first pixel of the first row of the bitmap, but you can perform arithmetic on the `ptr` variable to move it to other locations within the bitmap.

One drawback is that you can use this `ptr` variable only in a code block marked with the `unsafe` keyword. In addition, the assembly must be flagged as allowing unsafe blocks. This is done in the project's properties.

Using pointers in C# is very powerful, but also very dangerous. You need to be careful that you don't access memory beyond what the pointer is supposed to reference. This is why pointer use is associated with the word "unsafe."

Here's the method in the `GradientBitmapPage` class that uses the `GetPixels` method. Notice the `unsafe` block that encompasses all the code using the byte pointer:

```
SKBitmap FillBitmapBytePtr(out string description, out int milliseconds)  
{  
    description = "GetPixels byte ptr";  
    SKBitmap bitmap = new SKBitmap(256, 256);  
  
    stopwatch.Restart();  
  
    IntPtr pixelsAddr = bitmap.GetPixels();  
  
    unsafe  
    {  
        for (int rep = 0; rep < REPS; rep++)  
        {  
            byte* ptr = (byte*)pixelsAddr.ToPointer();  
  
            for (int row = 0; row < 256; row++)  
                for (int col = 0; col < 256; col++)  
                {  
                    *ptr++ = (byte)(col); // red  
                    *ptr++ = 0; // green  
                    *ptr++ = (byte)(row); // blue  
                    *ptr++ = 0xFF; // alpha  
                }  
        }  
    }  
  
    milliseconds = (int)stopwatch.ElapsedMilliseconds;  
    return bitmap;  
}
```

When the `ptr` variable is first obtained from the `ToPointer` method, it points to the first byte of the leftmost pixel of the first row of the bitmap. The `for` loops for `row` and `col` are set up so that `ptr` can be incremented

with the `++` operator after each byte of each pixel is set. For the other 99 loops through the pixels, the `ptr` must be set back to the beginning of the bitmap.

Each pixel is four bytes of memory, so each byte has to be set separately. The code here assumes that the bytes are in the order red, green, blue, and alpha, which is consistent with the `SKColorType.Rgba8888` color type. You might recall that this is the default color type for iOS and Android, but not for the Universal Windows Platform. By default, the UWP creates bitmaps with the `SKColorType.Bgra8888` color type. For this reason, expect to see some different results on that platform!

It's possible to cast the value returned from `ToPointer` to a `uint` pointer rather than a `byte` pointer. This allows an entire pixel to be accessed in one statement. Applying the `++` operator to that pointer increments it by four bytes to point to the next pixel:

```
public class GradientBitmapPage : ContentPage
{
    ...
    SKBitmap FillBitmapUIntPtr(out string description, out int milliseconds)
    {
        description = "GetPixels uint ptr";
        SKBitmap bitmap = new SKBitmap(256, 256);

        stopwatch.Restart();

        IntPtr pixelsAddr = bitmap.GetPixels();

        unsafe
        {
            for (int rep = 0; rep < REPS; rep++)
            {
                uint* ptr = (uint*)pixelsAddr.ToPointer();

                for (int row = 0; row < 256; row++)
                    for (int col = 0; col < 256; col++)
                    {
                        *ptr++ = MakePixel((byte)col, 0, (byte)row, 0xFF);
                    }
            }
        }

        milliseconds = (int)stopwatch.ElapsedMilliseconds;
        return bitmap;
    }
    ...
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    uint MakePixel(byte red, byte green, byte blue, byte alpha) =>
        (uint)((alpha << 24) | (blue << 16) | (green << 8) | red);
    ...
}
```

The pixel is set using the `MakePixel` method, which constructs an integer pixel from red, green, blue, and alpha components. Keep in mind that the `SKColorType.Rgba8888` format has a pixel byte ordering like this:

RR GG BB AA

But the integer corresponding to those bytes is:

AABBGGRR

The least significant byte of the integer is stored first in accordance with little-endian architecture. This `MakePixel` method will not work correctly for bitmaps with the `Bgra8888` color type.

The `MakePixel` method is flagged with the `MethodImplOptions.AggressiveInlining` option to encourage the

compiler to avoid making this a separate method, but instead to compile the code where the method is called. This should improve performance.

Interestingly, the `SKColor` structure defines an explicit conversion from `skcolor` to an unsigned integer, which means that an `SKColor` value can be created, and a conversion to `uint` can be used instead of `MakePixel`:

```
SKBitmap FillBitmapUIntPtrColor(out string description, out int milliseconds)
{
    description = "GetPixels SKColor";
    SKBitmap bitmap = new SKBitmap(256, 256);

    stopwatch.Restart();

    IntPtr pixelsAddr = bitmap.GetPixels();

    unsafe
    {
        for (int rep = 0; rep < REPS; rep++)
        {
            uint* ptr = (uint*)pixelsAddr.ToPointer();

            for (int row = 0; row < 256; row++)
                for (int col = 0; col < 256; col++)
                {
                    *ptr++ = (uint)new SKColor((byte)col, 0, (byte)row);
                }
        }
    }

    milliseconds = (int)stopwatch.ElapsedMilliseconds;
    return bitmap;
}
```

The only question is this: Is the integer format of the `skcolor` value in the order of the `SKColorType.Rgba8888` color type, or the `SKColorType.Bgra8888` color type, or is it something else entirely? The answer to that question shall be revealed shortly.

The SetPixels Method

`SKBitmap` also defines a method named `SetPixels`, which you call like this:

```
bitmap.SetPixels(intPtr);
```

Recall that `GetPixels` obtains an `IntPtr` referencing the memory block used by the bitmap to store its pixels. The `SetPixels` call *replaces* that memory block with the memory block referenced by the `IntPtr` specified as the `SetPixels` argument. The bitmap then frees up the memory block it was using previously. The next time `GetPixels` is called, it obtains the memory block set with `SetPixels`.

At first, it seems as if `SetPixels` gives you no more power and performance than `GetPixels` while being less convenient. With `GetPixels` you obtain the bitmap memory block and access it. With `SetPixels` you allocate and access some memory, and then set that as the bitmap memory block.

But using `SetPixels` offers a distinct syntactic advantage: It allows you to access the bitmap pixel bits using an array. Here's the method in `GradientBitmapPage` that demonstrates this technique. The method first defines a multi-dimensional byte array corresponding to the bytes of the bitmap's pixels. The first dimension is the row, the second dimension is the column, and the third dimension corresponds to the four components of each pixel:

```

SKBitmap FillBitmapByteBuffer(out string description, out int milliseconds)
{
    description = "SetPixels byte buffer";
    SKBitmap bitmap = new SKBitmap(256, 256);

    stopwatch.Restart();

    byte[,] buffer = new byte[256, 256, 4];

    for (int rep = 0; rep < REPS; rep++)
        for (int row = 0; row < 256; row++)
            for (int col = 0; col < 256; col++)
            {
                buffer[row, col, 0] = (byte)col; // red
                buffer[row, col, 1] = 0; // green
                buffer[row, col, 2] = (byte)row; // blue
                buffer[row, col, 3] = 0xFF; // alpha
            }

    unsafe
    {
        fixed (byte* ptr = buffer)
        {
            bitmap.SetPixels((IntPtr)ptr);
        }
    }

    milliseconds = (int)stopwatch.ElapsedMilliseconds;
    return bitmap;
}

```

Then, after the array has been filled with pixels, an `unsafe` block and a `fixed` statement is used to obtain a byte pointer that points to this array. That byte pointer can then be cast to an `IntPtr` to pass to `SetPixels`.

The array that you create doesn't have to be a byte array. It can be an integer array with only two dimensions for the row and column:

```

SKBitmap FillBitmapUIntBuffer(out string description, out int milliseconds)
{
    description = "SetPixels uint buffer";
    SKBitmap bitmap = new SKBitmap(256, 256);

    stopwatch.Restart();

    uint[,] buffer = new uint[256, 256];

    for (int rep = 0; rep < REPS; rep++)
        for (int row = 0; row < 256; row++)
            for (int col = 0; col < 256; col++)
            {
                buffer[row, col] = MakePixel((byte)col, 0, (byte)row, 0xFF);
            }

    unsafe
    {
        fixed (uint* ptr = buffer)
        {
            bitmap.SetPixels((IntPtr)ptr);
        }
    }

    milliseconds = (int)stopwatch.ElapsedMilliseconds;
    return bitmap;
}

```

The `MakePixel` method is again used to combine the color components into a 32-bit pixel.

Just for completeness, here's the same code but with an `SKColor` value cast to an unsigned integer:

```
SKBitmap FillBitmapUintBufferColor(out string description, out int milliseconds)
{
    description = "SetPixels SKColor";
    SKBitmap bitmap = new SKBitmap(256, 256);

    stopwatch.Restart();

    uint[,] buffer = new uint[256, 256];

    for (int rep = 0; rep < REPS; rep++)
        for (int row = 0; row < 256; row++)
            for (int col = 0; col < 256; col++)
            {
                buffer[row, col] = (uint)new SKColor((byte)col, 0, (byte)row);
            }

    unsafe
    {
        fixed (uint* ptr = buffer)
        {
            bitmap.SetPixels((IntPtr)ptr);
        }
    }

    milliseconds = (int)stopwatch.ElapsedMilliseconds;
    return bitmap;
}
```

Comparing the techniques

The constructor of the [Gradient Color](#) page calls all eight of the methods shown above, and saves the results:

```
public class GradientBitmapPage : ContentPage
{
    ...
    string[] descriptions = new string[8];
    SKBitmap[] bitmaps = new SKBitmap[8];
    int[] elapsedTimes = new int[8];

    SKCanvasView canvasView;

    public GradientBitmapPage ()
    {
        Title = "Gradient Bitmap";

        bitmaps[0] = FillBitmapSetPixel(out descriptions[0], out elapsedTimes[0]);
        bitmaps[1] = FillBitmapPixelsProp(out descriptions[1], out elapsedTimes[1]);
        bitmaps[2] = FillBitmapBytePtr(out descriptions[2], out elapsedTimes[2]);
        bitmaps[4] = FillBitmapIntPtr(out descriptions[4], out elapsedTimes[4]);
        bitmaps[6] = FillBitmapIntPtrColor(out descriptions[6], out elapsedTimes[6]);
        bitmaps[3] = FillBitmapByteBuffer(out descriptions[3], out elapsedTimes[3]);
        bitmaps[5] = FillBitmapUintBuffer(out descriptions[5], out elapsedTimes[5]);
        bitmaps[7] = FillBitmapUintBufferColor(out descriptions[7], out elapsedTimes[7]);

        canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }
    ...
}
```

The constructor concludes by creating an `SKCanvasView` to display the resultant bitmaps. The `PaintSurface` handler divides its surface into eight rectangles and calls `Display` to display each one:

```
public class GradientBitmapPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        int width = info.Width;
        int height = info.Height;

        canvas.Clear();

        Display(canvas, 0, new SKRect(0, 0, width / 2, height / 4));
        Display(canvas, 1, new SKRect(width / 2, 0, width, height / 4));
        Display(canvas, 2, new SKRect(0, height / 4, width / 2, 2 * height / 4));
        Display(canvas, 3, new SKRect(width / 2, height / 4, width, 2 * height / 4));
        Display(canvas, 4, new SKRect(0, 2 * height / 4, width / 2, 3 * height / 4));
        Display(canvas, 5, new SKRect(width / 2, 2 * height / 4, width, 3 * height / 4));
        Display(canvas, 6, new SKRect(0, 3 * height / 4, width / 2, height));
        Display(canvas, 7, new SKRect(width / 2, 3 * height / 4, width, height));
    }

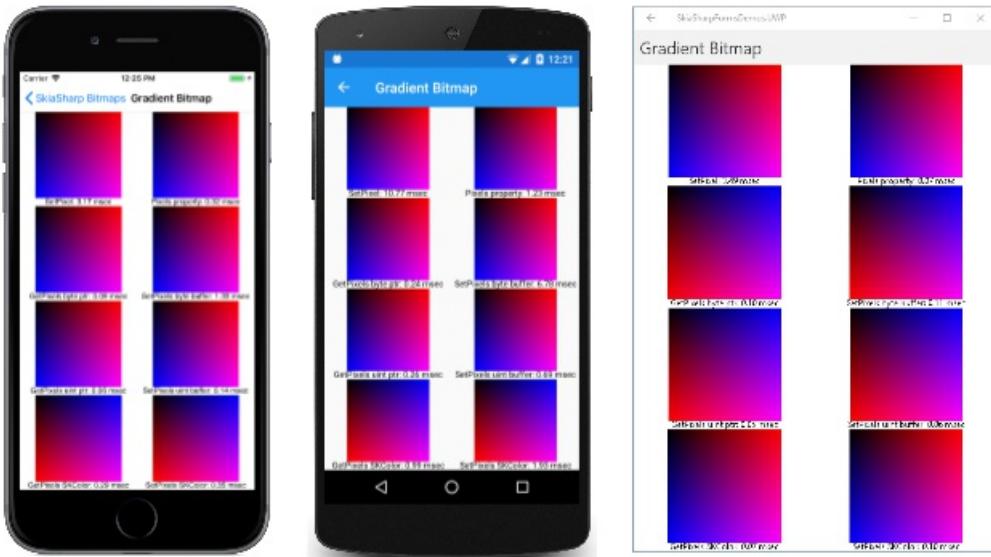
    void Display(SKCanvas canvas, int index, SKRect rect)
    {
        string text = String.Format("{0}: {1:F1} msec",
            descriptions[index],
            (double)elapsedTimes[index] / REPS);

        SKRect bounds = new SKRect();

        using (SKPaint textPaint = new SKPaint())
        {
            textPaint.TextSize = (float)(12 * canvasView.CanvasSize.Width / canvasView.Width);
            textPaint.TextAlign = SKTextAlign.Center;
            textPaint.MeasureText("Tly", ref bounds);

            canvas.DrawText(text, new SKPoint(rect.MidX, rect.Bottom - bounds.Bottom), textPaint);
            rect.Bottom -= bounds.Height;
            canvas.DrawBitmap(bitmaps[index], rect, BitmapStretch.Uniform);
        }
    }
}
```

To allow the compiler to optimize the code, this page was run in **Release** mode. Here's that page running on an iPhone 8 simulator on a MacBook Pro, a Nexus 5 Android phone, and Surface Pro 3 running Windows 10. Because of the hardware differences, avoid comparing the performance times between the devices, but instead look at the relative times on each device:



Here is a table that consolidates the execution times in milliseconds:

API	DATA TYPE	IOS	ANDROID	UWP
SetPixel		3.17	10.77	3.49
Pixels		0.32	1.23	0.07
GetPixels	byte	0.09	0.24	0.10
	uint	0.06	0.26	0.05
	SKColor	0.29	0.99	0.07
SetPixels	byte	1.33	6.78	0.11
	uint	0.14	0.69	0.06
	SKColor	0.35	1.93	0.10

As expected, calling `SetPixel` 65,536 times is the least efficient way to set a bitmap's pixels. Filling an `skcolor` array and setting the `Pixels` property is much better, and even compares favorably with some of the `GetPixels` and `SetPixels` techniques. Working with `uint` pixel values is generally faster than setting separate `byte` components, and converting the `skcolor` value to an unsigned integer adds some overhead to the process.

It's also interesting to compare the various gradients: The top rows of each platform are the same, and show the gradient as it was intended. This means that the `SetPixel` method and the `Pixels` property correctly create pixels from colors regardless of the underlying pixel format.

The next two rows of the iOS and Android screenshots are also the same, which confirms that the little `MakePixel` method is correctly defined for the default `Rgba8888` pixel format for these platforms.

The bottom row of the iOS and Android screenshots is backwards, which indicates that the unsigned integer obtained by casting an `skcolor` value is in the form:

AARRGGBB

The bytes are in the order:

BB GG RR AA

This is the `Bgra8888` ordering rather than the `Rgba8888` ordering. The `Bgra8888` format is the default for the Universal Windows platform, which is why the gradients on the last row of that screenshot are the same as the first row. But the middle two rows are incorrect because the code creating those bitmaps assumed an `Rgba8888` ordering.

If you want to use the same code for accessing pixel bits on each platform, you can explicitly create an `SKBitmap` using either the `Rgba8888` or `Bgra8888` format. If you want to cast `SKColor` values to bitmap pixels, use `Bgra8888`.

Random access of pixels

The `FillBitmapBytePtr` and `FillBitmapUintPtr` methods in the [Gradient Bitmap](#) page benefited from `for` loops designed to fill the bitmap sequentially, from top row to bottom row, and in each row from left to right. The pixel could be set with the same statement that incremented the pointer.

Sometimes it's necessary to access the pixels randomly rather than sequentially. If you're using the `GetPixels` approach, you'll need to calculate pointers based on the row and column. This is demonstrated in the [Rainbow Sine](#) page, which creates a bitmap showing a rainbow in the form of one cycle of a sine curve.

The colors of the rainbow are easiest to create using the HSL (hue, saturation, luminosity) color model. The `SKColor.FromHsl` method creates an `SKColor` value using hue values that range from 0 to 360 (like the angles of a circle but going from red, through green and blue, and back to red), and saturation and luminosity values ranging from 0 to 100. For the colors of a rainbow, the saturation should be set to a maximum of 100, and the luminosity to a midpoint of 50.

[Rainbow Sine](#) creates this image by looping through the rows of the bitmap, and then looping through 360 hue values. From each hue value, it calculates a bitmap column that is also based on a sine value:

```

public class RainbowSinePage : ContentPage
{
    SKBitmap bitmap;

    public RainbowSinePage()
    {
        Title = "Rainbow Sine";

        bitmap = new SKBitmap(360 * 3, 1024, SKColorType.Bgra8888, SKAlphaType.Unpremul);

        unsafe
        {
            // Pointer to first pixel of bitmap
            uint* basePtr = (uint*)bitmap.GetPixels().ToPointer();

            // Loop through the rows
            for (int row = 0; row < bitmap.Height; row++)
            {
                // Calculate the sine curve angle and the sine value
                double angle = 2 * Math.PI * row / bitmap.Height;
                double sine = Math.Sin(angle);

                // Loop through the hues
                for (int hue = 0; hue < 360; hue++)
                {
                    // Calculate the column
                    int col = (int)(360 + 360 * sine + hue);

                    // Calculate the address
                    uint* ptr = basePtr + bitmap.Width * row + col;

                    // Store the color value
                    *ptr = (uint)SKColor.FromHsl(hue, 100, 50);
                }
            }
        }

        // Create the SKCanvasView
        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();
        canvas.DrawBitmap(bitmap, info.Rect);
    }
}

```

Notice that the constructor creates the bitmap based on the `SKColorType.Bgra8888` format:

```
bitmap = new SKBitmap(360 * 3, 1024, SKColorType.Bgra8888, SKAlphaType.Unpremul);
```

This allows the program to use the conversion of `SKColor` values into `uint` pixels without worry. Although it doesn't play a role in this particular program, whenever you use the `SKColor` conversion to set pixels, you should also specify `SKAlphaType.Unpremul` because `SKColor` doesn't premultiply its color components by the alpha value.

The constructor then uses the `GetPixels` method to obtain a pointer to the first pixel of the bitmap:

```
uint* basePtr = (uint*)bitmap.GetPixels().ToPointer();
```

For any particular row and column, an offset value must be added to `basePtr`. This offset is the row times the bitmap width, plus the column:

```
uint* ptr = basePtr + bitmap.Width * row + col;
```

The `skColor` value is stored in memory using this pointer:

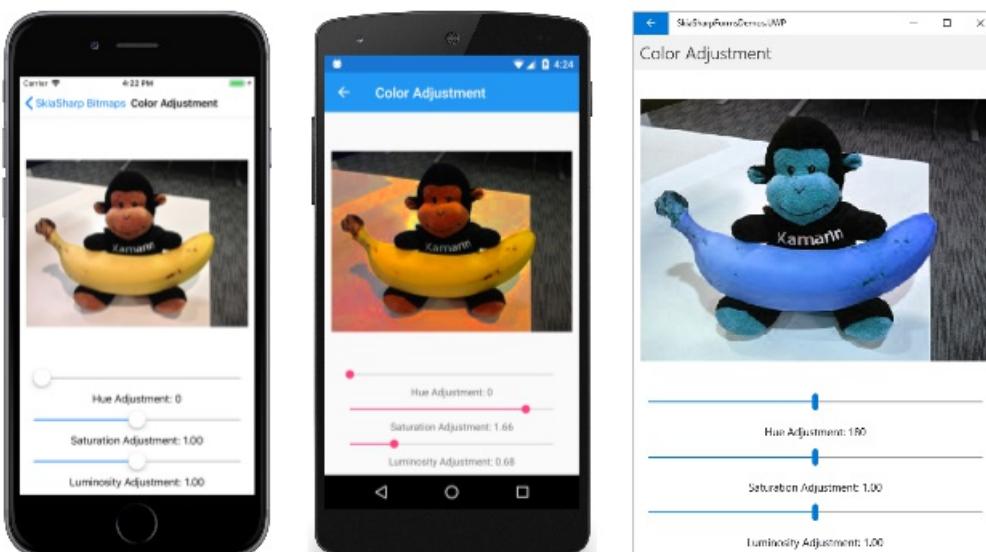
```
*ptr = (uint)SKColor.FromHsl(hue, 100, 50);
```

In the `PaintSurface` handler of the `SKCanvasView`, the bitmap is stretched to fill the display area:



From one bitmap to another

Very many image-processing tasks involve modifying pixels as they are transferred from one bitmap to another. This technique is demonstrated in the **Color Adjustment** page. The page loads one of the bitmap resources and then allows you to modify the image using three `Slider` views:



For each pixel color, the first `Slider` adds a value from 0 to 360 to the hue, but then uses the modulo operator to keep the result between 0 and 360, effectively shifting the colors along the spectrum (as the UWP screenshot demonstrates). The second `Slider` lets you select a multiplicative factor between 0.5 and 2 to apply to the saturation, and the third `Slider` does the same for the luminosity, as demonstrated in the Android screenshot.

The program maintains two bitmaps, the original source bitmap named `srcBitmap` and the adjusted destination bitmap named `dstBitmap`. Each time a `Slider` is moved, the program calculates all new pixels in `dstBitmap`. Of course, users will experiment by moving the `Slider` views very quickly, so you want the best performance you can manage. This involves the `GetPixels` method for both the source and destination bitmaps.

The **Color Adjustment** page doesn't control the color format of the source and destination bitmaps. Instead, it contains slightly different logic for `SKColorType.Rgba8888` and `SKColorType.Bgra8888` formats. The source and destination can be different formats, and the program will still work.

Here's the program except for the crucial `TransferPixels` method that transfers the pixels from the source to the destination. The constructor sets `dstBitmap` equal to `srcBitmap`. The `PaintSurface` handler displays `dstBitmap`:

```
public partial class ColorAdjustmentPage : ContentPage
{
    SKBitmap srcBitmap =
        BitmapExtensions.LoadBitmapResource(typeof(FillRectanglePage),
                                              "SkiaSharpFormsDemos.Media.Banana.jpg");
    SKBitmap dstBitmap;

    public ColorAdjustmentPage()
    {
        InitializeComponent();

        dstBitmap = new SKBitmap(srcBitmap.Width, srcBitmap.Height);
        OnSliderValueChanged(null, null);
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        float hueAdjust = (float)hueSlider.Value;
        hueLabel.Text = $"Hue Adjustment: {hueAdjust:F0}";

        float saturationAdjust = (float)Math.Pow(2, saturationSlider.Value);
        saturationLabel.Text = $"Saturation Adjustment: {saturationAdjust:F2}";

        float luminosityAdjust = (float)Math.Pow(2, luminositySlider.Value);
        luminosityLabel.Text = $"Luminosity Adjustment: {luminosityAdjust:F2}";

        TransferPixels(hueAdjust, saturationAdjust, luminosityAdjust);
        canvasView.InvalidateSurface();
    }

    ...
}

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();
    canvas.DrawBitmap(dstBitmap, info.Rect, BitmapStretch.Uniform);
}
```

The `valueChanged` handler for the `Slider` views calculates the adjustment values and calls `TransferPixels`.

The entire `TransferPixels` method is marked as `unsafe`. It begins by obtaining byte pointers to the pixel bits of both bitmaps, and then loops through all the rows and columns. From the source bitmap, the method obtains

four bytes for each pixel. These could be in either the `Rgba8888` or `Bgra8888` order. Checking for the color type allows an `SKColor` value to be created. The HSL components are then extracted, adjusted, and used to recreate the `SKColor` value. Depending on whether the destination bitmap is `Rgba8888` or `Bgra8888`, the bytes are stored in the destination bitmp:

```

public partial class ColorAdjustmentPage : ContentPage
{
    ...
    unsafe void TransferPixels(float hueAdjust, float saturationAdjust, float luminosityAdjust)
    {
        byte* srcPtr = (byte*)srcBitmap.GetPixels().ToPointer();
        byte* dstPtr = (byte*)dstBitmap.GetPixels().ToPointer();

        int width = srcBitmap.Width;           // same for both bitmaps
        int height = srcBitmap.Height;

        SKColorType typeOrg = srcBitmap.ColorType;
        SKColorType typeAdj = dstBitmap.ColorType;

        for (int row = 0; row < height; row++)
        {
            for (int col = 0; col < width; col++)
            {
                // Get color from original bitmap
                byte byte1 = *srcPtr++;           // red or blue
                byte byte2 = *srcPtr++;           // green
                byte byte3 = *srcPtr++;           // blue or red
                byte byte4 = *srcPtr++;           // alpha

                SKColor color = new SKColor();

                if (typeOrg == SKColorType.Rgba8888)
                {
                    color = new SKColor(byte1, byte2, byte3, byte4);
                }
                else if (typeOrg == SKColorType.Bgra8888)
                {
                    color = new SKColor(byte3, byte2, byte1, byte4);
                }

                // Get HSL components
                color.ToHsl(out float hue, out float saturation, out float luminosity);

                // Adjust HSL components based on adjustments
                hue = (hue + hueAdjust) % 360;
                saturation = Math.Max(0, Math.Min(100, saturationAdjust * saturation));
                luminosity = Math.Max(0, Math.Min(100, luminosityAdjust * luminosity));

                // Recreate color from HSL components
                color = SKColor.FromHsl(hue, saturation, luminosity);

                // Store the bytes in the adjusted bitmap
                if (typeAdj == SKColorType.Rgba8888)
                {
                    *dstPtr++ = color.Red;
                    *dstPtr++ = color.Green;
                    *dstPtr++ = color.Blue;
                    *dstPtr++ = color.Alpha;
                }
                else if (typeAdj == SKColorType.Bgra8888)
                {
                    *dstPtr++ = color.Blue;
                    *dstPtr++ = color.Green;
                    *dstPtr++ = color.Red;
                    *dstPtr++ = color.Alpha;
                }
            }
        }
    }
}

```

It's likely that the performance of this method could be improved even more by creating separate methods for the various combinations of color types of the source and destination bitmaps, and avoid checking the type for every pixel. Another option is to have multiple `for` loops for the `col` variable based on the color type.

Posterization

Another common job that involves accessing pixel bits is *posterization*. The number of colors encoded in a bitmap's pixels is reduced so that the result resembles a hand-drawn poster using a limited color palette.

The **Posterize** page performs this process on one of the monkey images:

```
public class PosterizePage : ContentPage
{
    SKBitmap bitmap =
        BitmapExtensions.LoadBitmapResource(typeof(FillRectanglePage),
                                              "SkiaSharpFormsDemos.Media.Banana.jpg");
    public PosterizePage()
    {
        Title = "Posterize";

        unsafe
        {
            uint* ptr = (uint*)bitmap.GetPixels().ToPointer();
            int pixelCount = bitmap.Width * bitmap.Height;

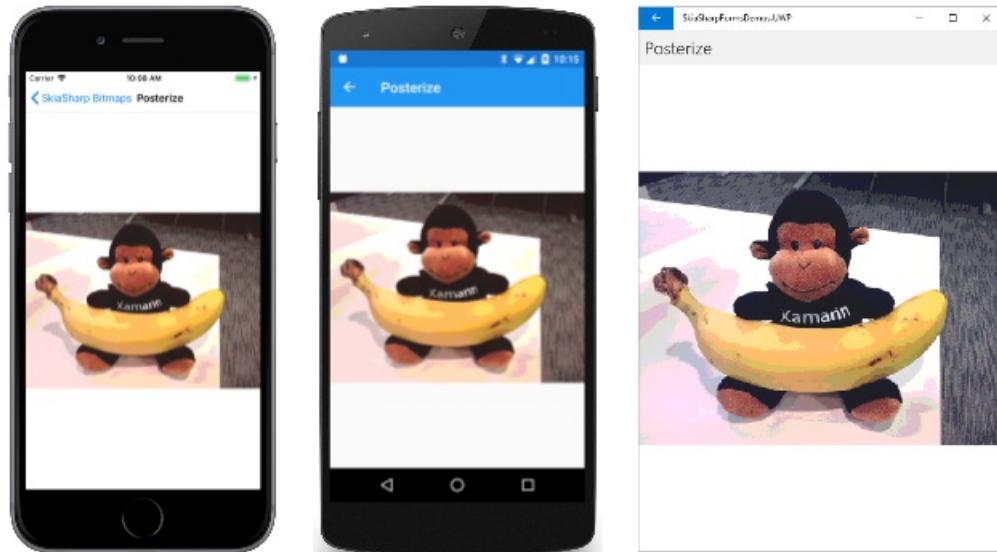
            for (int i = 0; i < pixelCount; i++)
            {
                *ptr++ &= 0xE0E0E0FF;
            }
        }

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();
        canvas.DrawBitmap(bitmap, info.Rect, BitmapStretch.Uniform);
    }
}
```

The code in the constructor accesses each pixel, performs a bitwise AND operation with the value `0xE0E0E0FF`, and then stores the result back in the bitmap. The values `0xE0E0E0FF` keeps the high 3 bits of each color component and sets the lower 5 bits to 0. Rather than 2^{24} or 16,777,216 colors, the bitmap is reduced to 2^9 or 512 colors:



Related links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Animating SkiaSharp bitmaps

3/5/2021 • 16 minutes to read • [Edit Online](#)

 [Download the sample](#)

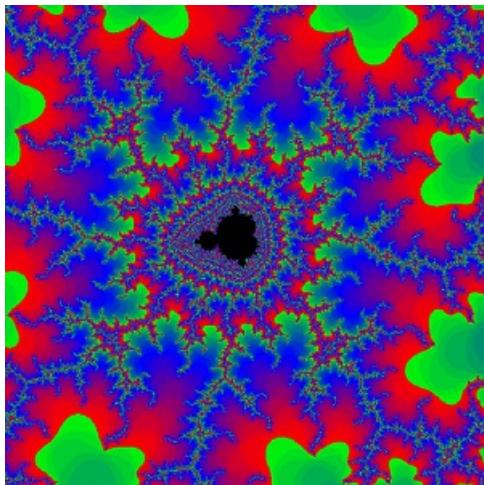
Applications that animate SkiaSharp graphics generally call `InvalidateSurface` on the `SKCanvasView` at a fixed rate, often every 16 milliseconds. Invalidating the surface triggers a call to the `PaintSurface` handler to redraw the display. As the visuals are redrawn 60 times a second, they appear to be smoothly animated.

However, if the graphics are too complex to be rendered in 16 milliseconds, the animation can become jittery. The programmer might choose to reduce the refresh rate to 30 times or 15 times a second, but sometimes even that's not enough. Sometimes graphics are so complex that they simply can't be rendered in real time.

One solution is to prepare for the animation beforehand by rendering the individual frames of the animation on a series of bitmaps. To display the animation, it's only necessary to display these bitmaps sequentially 60 times a second.

Of course, that's potentially a lot of bitmaps, but that is how big-budget 3D animated movies are made. The 3D graphics are much too complex to be rendered in real time. A lot of processing time is required to render each frame. What you see when you watch the movie is essentially a series of bitmaps.

You can do something similar in SkiaSharp. This article demonstrates two types of bitmap animation. The first example is an animation of the Mandelbrot Set:



The second example shows how to use SkiaSharp to render an animated GIF file.

Bitmap animation

The Mandelbrot Set is visually fascinating but computationally lengthy. (For a discussion of the Mandelbrot Set and the mathematics used here, see [Chapter 20 of *Creating Mobile Apps with Xamarin.Forms*](#) starting on page 666. The following description assumes that background knowledge.)

The [Mandelbrot Animation](#) sample uses bitmap animation to simulate a continuous zoom of a fixed point in the Mandelbrot Set. Zooming in is followed by zooming out, and then the cycle repeats forever or until you end the program.

The program prepares for this animation by creating up to 50 bitmaps that it stores in application local storage. Each bitmap encompasses half the width and height of the complex plane as the previous bitmap. (In the program, these bitmaps are said to represent integral *zoom levels*.) The bitmaps are then displayed in sequence.

The scaling of each bitmap is animated to provide a smooth progression from one bitmap to another.

Like the final program described in Chapter 20 of *Creating Mobile Apps with Xamarin.Forms*, the calculation of the Mandelbrot Set in **Mandelbrot Animation** is an asynchronous method with eight parameters. The parameters include a complex center point, and a width and height of the complex plane surrounding that center point. The next three parameters are the pixel width and height of the bitmap to be created, and a maximum number of iterations for the recursive calculation. The `progress` parameter is used to display the progress of this calculation. The `cancelToken` parameter is not used in this program:

```

static class Mandelbrot
{
    public static Task<BitmapInfo> CalculateAsync(Complex center,
                                                double width, double height,
                                                int pixelWidth, int pixelHeight,
                                                int iterations,
                                                IProgress<double> progress,
                                                CancellationToken cancellationToken)
    {
        return Task.Run(() =>
        {
            int[] iterationCounts = new int[pixelWidth * pixelHeight];
            int index = 0;

            for (int row = 0; row < pixelHeight; row++)
            {
                progress.Report((double)row / pixelHeight);
                cancellationToken.ThrowIfCancellationRequested();

                double y = center.Imaginary + height / 2 - row * height / pixelHeight;

                for (int col = 0; col < pixelWidth; col++)
                {
                    double x = center.Real - width / 2 + col * width / pixelWidth;
                    Complex c = new Complex(x, y);

                    if ((c - new Complex(-1, 0)).Magnitude < 1.0 / 4)
                    {
                        iterationCounts[index++] = -1;
                    }
                    // http://www.reenigne.org/blog/algorithm-for-mandelbrot-cardioid/
                    else if (c.Magnitude * c.Magnitude * (8 * c.Magnitude * c.Magnitude - 3) < 3.0 / 32 -
c.Real)
                    {
                        iterationCounts[index++] = -1;
                    }
                    else
                    {
                        Complex z = 0;
                        int iteration = 0;

                        do
                        {
                            z = z * z + c;
                            iteration++;
                        }
                        while (iteration < iterations && z.Magnitude < 2);

                        if (iteration == iterations)
                        {
                            iterationCounts[index++] = -1;
                        }
                        else
                        {
                            iterationCounts[index++] = iteration;
                        }
                    }
                }
            }
            return new BitmapInfo(pixelWidth, pixelHeight, iterationCounts);
        }, cancellationToken);
    }
}

```

The method returns an object of type `BitmapInfo` that provides information for creating a bitmap:

```

class BitmapInfo
{
    public BitmapInfo(int pixelWidth, int pixelHeight, int[] iterationCounts)
    {
        PixelWidth = pixelWidth;
        PixelHeight = pixelHeight;
        IterationCounts = iterationCounts;
    }

    public int PixelWidth { private set; get; }

    public int PixelHeight { private set; get; }

    public int[] IterationCounts { private set; get; }
}

```

The **Mandelbrot Animation** XAML file includes two `Label` views, a `ProgressBar`, and a `Button` as well as the `SKCanvasView`:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="MandelAnima.MainPage"
    Title="Mandelbrot Animation">

    <StackLayout>
        <Label x:Name="statusLabel"
            HorizontalTextAlignment="Center" />
        <ProgressBar x:Name="progressBar" />

        <skia:SKCanvasView x:Name="canvasView"
            VerticalOptions="FillAndExpand"
            PaintSurface="OnCanvasViewPaintSurface" />

        <StackLayout Orientation="Horizontal"
            Padding="5">
            <Label x:Name="storageLabel"
                VerticalOptions="Center" />

            <Button x:Name="deleteButton"
                Text="Delete All"
                HorizontalOptions="EndAndExpand"
                Clicked="onDeleteButtonClicked" />
        </StackLayout>
    </StackLayout>
</ContentPage>

```

The code-behind file begins by defining three crucial constants and an array of bitmaps:

```

public partial class MainPage : ContentPage
{
    const int COUNT = 10;           // The number of bitmaps in the animation.
                                   // This can go up to 50!

    const int BITMAP_SIZE = 1000;   // Program uses square bitmaps exclusively

    // Uncomment just one of these, or define your own
    static readonly Complex center = new Complex(-1.17651152924355, 0.298520986549558);
    // static readonly Complex center = new Complex(-0.774693089457127, 0.124226621261617);
    // static readonly Complex center = new Complex(-0.556624880053304, 0.634696788141351);

    SKBitmap[] bitmaps = new SKBitmap[COUNT]; // array of bitmaps
    ...
}

```

At some point, you'll probably want to change the `COUNT` value to 50 to see the full range of the animation. Values above 50 are not useful. Around a zoom level of 48 or so, the resolution of double-precision floating-point numbers becomes insufficient for the Mandelbrot Set calculation. This problem is discussed on page 684 of *Creating Mobile Apps with Xamarin.Forms*.

The `center` value is very important. This is the focus of the animation zoom. The three values in the file are those used in the three final screenshots in Chapter 20 of *Creating Mobile Apps with Xamarin.Forms* on page 684, but you can experiment with the program in that chapter to come up with one of your own values.

The **Mandelbrot Animation** sample stores these `COUNT` bitmaps in local application storage. Fifty bitmaps require over 20 megabytes of storage on your device, so you might want to know how much storage these bitmaps are occupying, and at some point you might want to delete them all. That's the purpose of these two methods at the bottom of the `MainPage` class:

```

public partial class MainPage : ContentPage
{
    ...
    void TallyBitmapSizes()
    {
        long fileSize = 0;

        foreach (string filename in Directory.EnumerateFiles(FolderPath()))
        {
            fileSize += new FileInfo(filename).Length;
        }

        storageLabel.Text = $"Total storage: {fileSize:N0} bytes";
    }

    void OnDeleteButtonClicked(object sender, EventArgs args)
    {
        foreach (string filepath in Directory.EnumerateFiles(FolderPath()))
        {
            File.Delete(filepath);
        }

        TallyBitmapSizes();
    }
}

```

You can delete the bitmaps in local storage while the program is animating those same bitmaps because the program retains them in memory. But the next time you run the program, it will need to recreate the bitmaps.

The bitmaps stored in local application storage incorporate the `center` value in their filenames, so if you change the `center` setting, the existing bitmaps will not be replaced in storage, and will continue to occupy space.

Here are the methods that `MainPage` uses for constructing the filenames, as well as a `MakePixel` method for defining a pixel value based on color components:

```
public partial class MainPage : ContentPage
{
    ...
    // File path for storing each bitmap in local storage
    string FolderPath() =>
        Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData);

    string FilePath(int zoomLevel) =>
        Path.Combine(FolderPath(),
            String.Format("R{0}I{1}Z{2:D2}.png", center.Real, center.Imaginary, zoomLevel));

    // Form bitmap pixel for Rgba8888 format
    uint MakePixel(byte alpha, byte red, byte green, byte blue) =>
        (uint)((alpha << 24) | (blue << 16) | (green << 8) | red);
    ...
}
```

The `zoomLevel` parameter to `FilePath` ranges from 0 to the `COUNT` constant minus 1.

The `MainPage` constructor calls the `LoadAndStartAnimation` method:

```
public partial class MainPage : ContentPage
{
    ...
    public MainPage()
    {
        InitializeComponent();

        LoadAndStartAnimation();
    }
    ...
}
```

The `LoadAndStartAnimation` method is responsible for accessing application local storage to load any bitmaps that might have been created when the program was run previously. It loops through `zoomLevel` values from 0 to `COUNT`. If the file exists, it loads it into the `bitmaps` array. Otherwise, it needs to create a bitmap for the particular `center` and `zoomLevel` values by calling `Mandelbrot.CalculateAsync`. That method obtains the iteration counts for each pixel, which this method converts into colors:

```
public partial class MainPage : ContentPage
{
    ...
    async void LoadAndStartAnimation()
    {
        // Show total bitmap storage
        TallyBitmapSizes();

        // Create progressReporter for async operation
        Progress<double> progressReporter =
            new Progress<double>((double progress) => progressBar.Progress = progress);

        // Create (unused) CancellationTokenSource for async operation
        CancellationTokenSource cancelTokenSource = new CancellationTokenSource();

        // Loop through all the zoom levels
        for (int zoomLevel = 0; zoomLevel < COUNT; zoomLevel++)
        {
            // If the file exists, load it
            if (File.Exists(FilePath(zoomLevel)))

```

```

{
    statusLabel.Text = $"Loading bitmap for zoom level {zoomLevel}";

    using (Stream stream = File.OpenRead(filePath(zoomLevel)))
    {
        bitmaps[zoomLevel] = SKBitmap.Decode(stream);
    }
}

// Otherwise, create a new bitmap
else
{
    statusLabel.Text = $"Creating bitmap for zoom level {zoomLevel}";

    CancellationToken cancellationToken = cancelTokenSource.Token;

    // Do the (generally lengthy) Mandelbrot calculation
    BitmapInfo bitmapInfo =
        await Mandelbrot.CalculateAsync(center,
            4 / Math.Pow(2, zoomLevel),
            4 / Math.Pow(2, zoomLevel),
            BITMAP_SIZE, BITMAP_SIZE,
            (int)Math.Pow(2, 10), progressReporter, cancellationToken);

    // Create bitmap & get pointer to the pixel bits
    SKBitmap bitmap = new SKBitmap(BITMAP_SIZE, BITMAP_SIZE, SKColorType.Rgba8888,
SKAlphaType.Opaque);
    IntPtr basePtr = bitmap.GetPixels();

    // Set pixel bits to color based on iteration count
    for (int row = 0; row < bitmap.Width; row++)
        for (int col = 0; col < bitmap.Height; col++)
    {
        int iterationCount = bitmapInfo.IterationCounts[row * bitmap.Width + col];
        uint pixel = 0xFF000000; // black

        if (iterationCount != -1)
        {
            double proportion = (iterationCount / 32.0) % 1;
            byte red = 0, green = 0, blue = 0;

            if (proportion < 0.5)
            {
                red = (byte)(255 * (1 - 2 * proportion));
                blue = (byte)(255 * 2 * proportion);
            }
            else
            {
                proportion = 2 * (proportion - 0.5);
                green = (byte)(255 * proportion);
                blue = (byte)(255 * (1 - proportion));
            }

            pixel = MakePixel(0xFF, red, green, blue);
        }

        // Calculate pointer to pixel
        IntPtr pixelPtr = basePtr + 4 * (row * bitmap.Width + col);

        unsafe // requires compiling with unsafe flag
        {
            *(uint*)pixelPtr.ToPointer() = pixel;
        }
    }

    // Save as PNG file
    SKData data = SKImage.FromBitmap(bitmap).Encode();

    try
    {

```

```

        File.WriteAllBytes(FilePath(zoomLevel), data.ToArray());
    }
    catch
    {
        // Probably out of space, but just ignore
    }

    // Store in array
    bitmaps[zoomLevel] = bitmap;

    // Show new bitmap sizes
    TallyBitmapSizes();
}

// Display the bitmap
bitmapIndex = zoomLevel;
canvasView.InvalidateSurface();
}

// Now start the animation
stopwatch.Start();
Device.StartTimer(TimeSpan.FromMilliseconds(16), OnTimerTick);
}
...
}

```

Notice that the program stores these bitmaps in local application storage rather than in the device's photo library. The .NET Standard 2.0 library allows using the familiar `File.OpenRead` and `File.WriteAllBytes` methods for this task.

After all the bitmaps have been created or loaded into memory, the method starts a `Stopwatch` object and calls `Device.StartTimer`. The `OnTimerTick` method is called every 16 milliseconds.

`OnTimerTick` calculates a `time` value in milliseconds that ranges from 0 to 6000 times `COUNT`, which apportions six seconds for the display of each bitmap. The `progress` value uses the `Math.Sin` value to create a sinusoidal animation that will be slower at the beginning of the cycle, and slower at the end as it reverses direction.

The `progress` value ranges from 0 to `COUNT`. This means that the integer part of `progress` is an index into the `bitmaps` array, while the fractional part of `progress` indicates a zoom level for that particular bitmap. These values are stored in the `bitmapIndex` and `bitmapProgress` fields, and are displayed by the `Label` and `Slider` in the XAML file. The `SKCanvasView` is invalidated to update the bitmap display:

```

public partial class MainPage : ContentPage
{
    ...
    Stopwatch stopwatch = new Stopwatch();           // for the animation
    int bitmapIndex;
    double bitmapProgress = 0;
    ...
    bool OnTimerTick()
    {
        int cycle = 6000 * COUNT;          // total cycle length in milliseconds

        // Time in milliseconds from 0 to cycle
        int time = (int)(stopwatch.ElapsedMilliseconds % cycle);

        // Make it sinusoidal, including bitmap index and gradation between bitmaps
        double progress = COUNT * 0.5 * (1 + Math.Sin(2 * Math.PI * time / cycle - Math.PI / 2));

        // These are the field values that the PaintSurface handler uses
        bitmapIndex = (int)progress;
        bitmapProgress = progress - bitmapIndex;

        // It doesn't often happen that we get up to COUNT, but an exception would be raised
        if (bitmapIndex < COUNT)
        {
            // Show progress in UI
            statusLabel.Text = $"Displaying bitmap for zoom level {bitmapIndex}";
            progressBar.Progress = bitmapProgress;

            // Update the canvas
            canvasView.InvalidateSurface();
        }

        return true;
    }
    ...
}

```

Finally, the `PaintSurface` handler of the `SKCanvasView` calculates a destination rectangle to display the bitmap as large as possible while maintaining the aspect ratio. A source rectangle is based on the `bitmapProgress` value. The `fraction` value calculated here ranges from 0 when `bitmapProgress` is 0 to display the entire bitmap, to 0.25 when `bitmapProgress` is 1 to display half the width and height of the bitmap, effectively zooming in:

```

public partial class MainPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

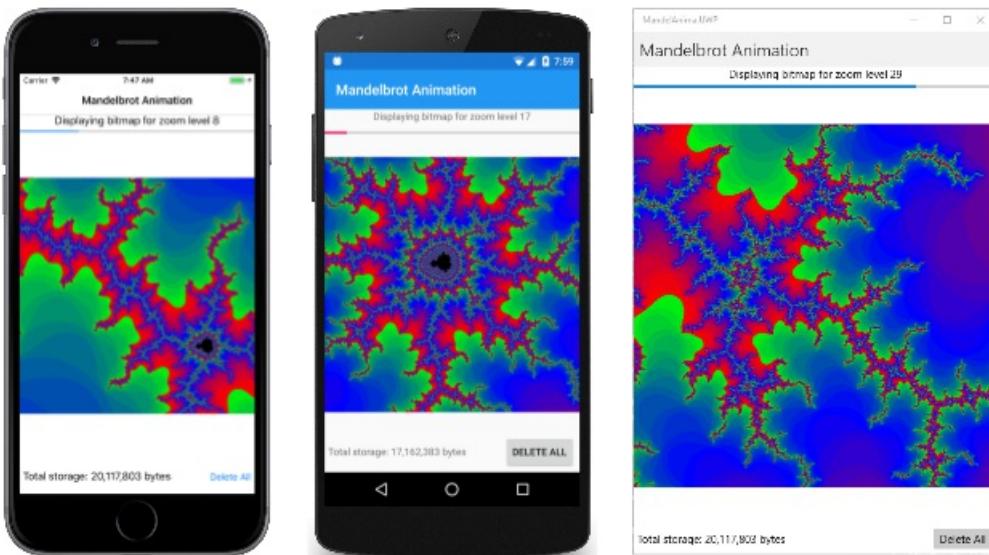
        if (bitmaps[bitmapIndex] != null)
        {
            // Determine destination rect as square in canvas
            int dimension = Math.Min(info.Width, info.Height);
            float x = (info.Width - dimension) / 2;
            float y = (info.Height - dimension) / 2;
            SKRect destRect = new SKRect(x, y, x + dimension, y + dimension);

            // Calculate source rectangle based on fraction:
            // bitmapProgress == 0: full bitmap
            // bitmapProgress == 1: half of length and width of bitmap
            float fraction = 0.5f * (1 - (float)Math.Pow(2, -bitmapProgress));
            SKBitmap bitmap = bitmaps[bitmapIndex];
            int width = bitmap.Width;
            int height = bitmap.Height;
            SKRect sourceRect = new SKRect(fraction * width, fraction * height,
                (1 - fraction) * width, (1 - fraction) * height);

            // Display the bitmap
            canvas.DrawBitmap(bitmap, sourceRect, destRect);
        }
    }
    ...
}

```

Here's the program running:



GIF animation

The Graphics Interchange Format (GIF) specification includes a feature that allows a single GIF file to contain multiple sequential frames of a scene that can be displayed in succession, often in a loop. These files are known as *animated GIFs*. Web browsers can play animated GIFs, and SkiaSharp allows an application to extract the frames from an animated GIF file and to display them sequentially.

The [SkiaSharpFormsDemos](#) sample includes an animated GIF resource named `Newtons_cradle_animation_book_2.gif` created by DemonDeLuxe and downloaded from the [Newton's Cradle](#) page in Wikipedia. The [Animated GIF](#) page includes a XAML file that provides that information and instantiates an `SKCanvasView`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos Bitmaps AnimatedGifPage"
    Title="Animated GIF">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <skia:SKCanvasView x:Name="canvasView"
            Grid.Row="0"
            PaintSurface="OnCanvasViewPaintSurface" />

        <Label Text="GIF file by DemonDeLuxe from Wikipedia Newton's Cradle page"
            Grid.Row="1"
            Margin="0, 5"
            HorizontalTextAlignment="Center" />
    </Grid>
</ContentPage>
```

The code-behind file is not generalized to play any animated GIF file. It ignores some of the information that is available, in particular a repetition count, and simply plays the animated GIF in a loop.

The use of SkiaSharp to extract the frames of an animated GIF file does not seem to be documented anywhere, so the description of the code that follows is more detailed than usual:

The decoding of the animated GIF file occurs in the page's constructor, and requires that the `Stream` object referencing the bitmap be used to create an `SKManagedStream` object and then an `SKCodec` object. The `FrameCount` property indicates the number of frames that make up the animation.

These frames are eventually saved as individual bitmaps, so the constructor uses `FrameCount` to allocate an array of type `SKBitmap` as well as two `int` arrays for the duration of each frame and (to ease the animation logic) the accumulated durations.

The `FrameInfo` property of `SKCodec` class is an array of `SKCodecFrameInfo` values, one for each frame, but the only thing this program takes from that structure is the `Duration` of the frame in milliseconds.

`SKCodec` defines a property named `Info` of type `SKImageInfo`, but that `SKImageInfo` value indicates (at least for this image) that the color type is `skColorType.Index8`, which means that each pixel is an index into a color type. To avoid bothering with color tables, the program uses the `Width` and `Height` information from that structure to construct its own full-color `ImageInfo` value. Each `SKBitmap` is created from that.

The `GetPixels` method of `SKBitmap` returns an `IntPtr` referencing the pixel bits of that bitmap. These pixel bits have not been set yet. That `IntPtr` is passed to one of the `GetPixels` methods of `SKCodec`. That method copies the frame from the GIF file into the memory space referenced by the `IntPtr`. The `SKCodecOptions` constructor indicates the frame number:

```

public partial class AnimatedGifPage : ContentPage
{
    SKBitmap[] bitmaps;
    int[] durations;
    int[] accumulatedDurations;
    int totalDuration;
    ...

    public AnimatedGifPage ()
    {
        InitializeComponent ();

        string resourceId = "SkiaSharpFormsDemos.Media.Newtons_cradle_animation_book_2.gif";
        Assembly assembly = GetType().GetTypeInfo().Assembly;

        using (Stream stream = assembly.GetManifestResourceStream(resourceId))
        using (SKManagedStream skStream = new SKManagedStream(stream))
        using (SKCodec codec = SKCodec.Create(skStream))
        {
            // Get frame count and allocate bitmaps
            int frameCount = codec.FrameCount;
            bitmaps = new SKBitmap[frameCount];
            durations = new int[frameCount];
            accumulatedDurations = new int[frameCount];

            // Note: There's also a RepetitionCount property of SKCodec not used here

            // Loop through the frames
            for (int frame = 0; frame < frameCount; frame++)
            {
                // From the FrameInfo collection, get the duration of each frame
                durations[frame] = codec.FrameInfo[frame].Duration;

                // Create a full-color bitmap for each frame
                SKImageInfo imageInfo = codec.new SKImageInfo(codec.Info.Width, codec.Info.Height);
                bitmaps[frame] = new SKBitmap(imageInfo);

                // Get the address of the pixels in that bitmap
                IntPtr pointer = bitmaps[frame].GetPixels();

                // Create an SKCodecOptions value to specify the frame
                SKCodecOptions codecOptions = new SKCodecOptions(frame, false);

                // Copy pixels from the frame into the bitmap
                codec.GetPixels(imageInfo, pointer, codecOptions);
            }

            // Sum up the total duration
            for (int frame = 0; frame < durations.Length; frame++)
            {
                totalDuration += durations[frame];
            }

            // Calculate the accumulated durations
            for (int frame = 0; frame < durations.Length; frame++)
            {
                accumulatedDurations[frame] = durations[frame] +
                    (frame == 0 ? 0 : accumulatedDurations[frame - 1]);
            }
        }
    ...
}

```

Despite the `IntPtr` value, no `unsafe` code is required because the `IntPtr` is never converted to a C# pointer value.

After each frame has been extracted, the constructor totals up the durations of all the frames, and then initializes another array with the accumulated durations.

The remainder of the code-behind file is dedicated to animation. The `Device.StartTimer` method is used to start a timer going, and the `OnTimerTick` callback uses a `Stopwatch` object to determine the elapsed time in milliseconds. Looping through the accumulated durations array is sufficient to find the current frame:

```

public partial class AnimatedGifPage : ContentPage
{
    SKBitmap[] bitmaps;
    int[] durations;
    int[] accumulatedDurations;
    int totalDuration;

    Stopwatch stopwatch = new Stopwatch();
    bool isAnimating;

    int currentFrame;
    ...
    protected override void OnAppearing()
    {
        base.OnAppearing();

        isAnimating = true;
        stopwatch.Start();
        Device.StartTimer(TimeSpan.FromMilliseconds(16), OnTimerTick);
    }

    protected override void OnDisappearing()
    {
        base.OnDisappearing();

        stopwatch.Stop();
        isAnimating = false;
    }

    bool OnTimerTick()
    {
        int msec = (int)(stopwatch.ElapsedMilliseconds % totalDuration);
        int frame = 0;

        // Find the frame based on the elapsed time
        for (frame = 0; frame < accumulatedDurations.Length; frame++)
        {
            if (msec < accumulatedDurations[frame])
            {
                break;
            }
        }

        // Save in a field and invalidate the SKCanvasView.
        if (currentFrame != frame)
        {
            currentFrame = frame;
            canvasView.InvalidateSurface();
        }
    }

    return isAnimating;
}

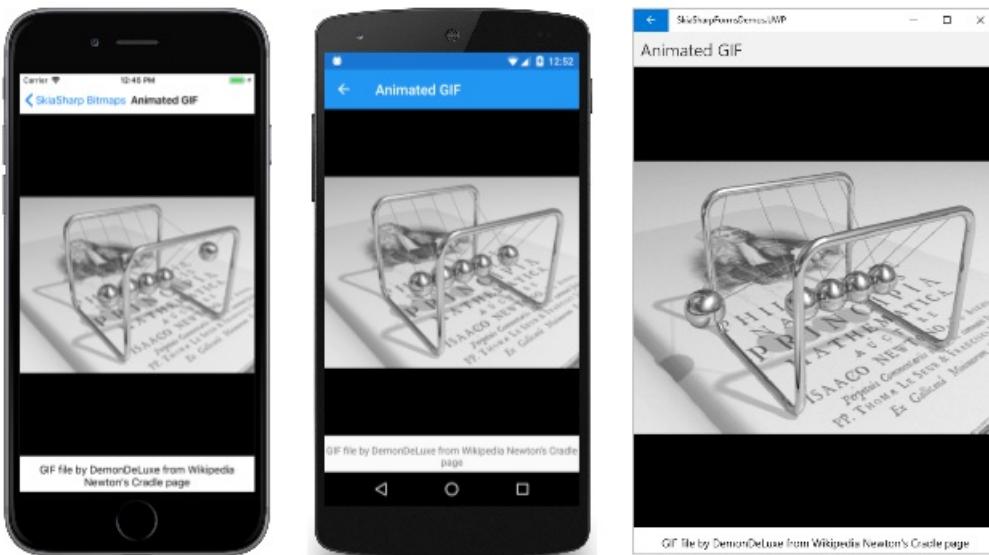
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear(SKColors.Black);

    // Get the bitmap and center it
    SKBitmap bitmap = bitmaps[currentFrame];
    canvas.DrawBitmap(bitmap, info.Rect, BitmapStretch.Uniform);
}
}

```

Each time the `currentframe` variable changes, the `SKCanvasView` is invalidated and the new frame is displayed:



Of course, you'll want to run the program yourself to see the animation.

Related links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)
- [Mandelbrot Animation \(sample\)](#)

SkiaSharp effects

3/5/2021 • 2 minutes to read • [Edit Online](#)

 [Download the sample](#)

The SkiaSharp `SKPaint` class defines six properties that can be classified under the general term of *effects*. These are properties that alter the normal display of graphics in some way. The SkiaSharp effects fall into six categories:

Path Effects

Set the `PathEffect` property of `SKPaint` to an object of type `SKPathEffect` to display dashed lines, or to stroke or fill an area with a pattern created from paths. The path effect was covered earlier in this series in the article [Path Effects in SkiaSharp](#).

Shaders

Set the `Shader` property of `SKPaint` to an object of type `SKShader` to display linear or circular gradients, tiled bitmaps, and Perlin noise patterns.

Blend Modes

Set the `BlendMode` property of `SKPaint` to a member of the `SKBlendMode` enumeration to govern what happens when a source graphic is displayed on a destination. SkiaSharp supports all the CSS compositing and blend modes, including the Porter-Duff modes, separable blend modes, and non-separable blend modes.

Mask Filters

Set the `MaskFilter` property of `SKPaint` to an object of type `SKMaskFilter` for blurs and other alpha effects.

Image Filters

Set the `ImageFilter` property of `SKPaint` to an object of type `SKImageFilter` for blurring bitmaps and creating drop shadows, embossing, or engraving effects.

Color Filters

Set the `ColorFilter` property of `SKPaint` to an object of type `SKColorFilter` to alter colors using tables or matrix transforms.

All the sample code for these articles are in the [SkiaSharpFormsDemos](#). From the home page, select [SkiaSharp Effects](#).

Related links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

SkiaSharp shaders

3/5/2021 • 2 minutes to read • [Edit Online](#)

 [Download the sample](#)

You can set the `Shader` property of `SKPaint` to an object of type `SKShader` to create several types of gradients, a tiled bitmap pattern, or Perlin noise.

The SkiaSharp linear gradient

Discover how to stroke lines or fill areas with gradients composed of a gradual blend of two colors.

SkiaSharp circular gradients

Learn about the different types of gradients based on circles, and use them for masks or specular highlights.

SkiaSharp bitmap tiling

Tile an area using bitmaps repeated horizontally and vertically.

SkiaSharp noise and composing

Generate Perlin noise shaders and combine with other shaders.

Related links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

The SkiaSharp linear gradient

3/5/2021 • 23 minutes to read • [Edit Online](#)



[Download the sample](#)

The `SKPaint` class defines a `Color` property that is used to stroke lines or fill areas with a solid color. You can alternatively stroke lines or fill areas with *gradients*, which are gradual blends of colors:



The most basic type of gradient is a *linear* gradient. The blend of colors occurs on a line (called the *gradient line*) from one point to another. Lines that are perpendicular to the gradient line have the same color. You create a linear gradient using one of the two static `SKShader.CreateLinearGradient` methods. The difference between the two overloads is that one includes a matrix transform and the other does not.

These methods return an object of type `SKShader` that you set to the `Shader` property of `SKPaint`. If the `Shader` property is non-null, it overrides the `Color` property. Any line that is stroked or any area that is filled using this `SKPaint` object is based on the gradient rather than the solid color.

NOTE

The `Shader` property is ignored when you include an `SKPaint` object in a `DrawBitmap` call. You can use the `Color` property of `SKPaint` to set a transparency level for displaying a bitmap (as described in the article [Displaying SkiaSharp bitmaps](#)), but you can't use the `Shader` property for displaying a bitmap with a gradient transparency. Other techniques are available for displaying bitmaps with gradient transparencies: These are described in the articles [SkiaSharp circular gradients](#) and [SkiaSharp compositing and blend modes](#).

Corner-to-corner gradients

Often a linear gradient extends from one corner of a rectangle to another. If the start point is the upper-left corner of the rectangle, the gradient can extend:

- vertically to the lower-left corner
- horizontally to the upper-right corner
- diagonally to the lower-right corner

The diagonal linear gradient is demonstrated in the first page in the [SkiaSharp Shaders and Other Effects](#) section of the [SkiaSharpFormsDemos](#) sample. The [Corner-to-Corner Gradient](#) page creates an `SKCanvasView` in its constructor. The `PaintSurface` handler creates an `SKPaint` object in a `using` statement and then defines a 300-pixel square rectangle centered in the canvas:

```

public class CornerToCornerGradientPage : ContentPage
{
    ...
    public CornerToCornerGradientPage ()
    {
        Title = "Corner-to-Corner Gradient";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
        ...
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        using (SKPaint paint = new SKPaint())
        {
            // Create 300-pixel square centered rectangle
            float x = (info.Width - 300) / 2;
            float y = (info.Height - 300) / 2;
            SKRect rect = new SKRect(x, y, x + 300, y + 300);

            // Create linear gradient from upper-left to lower-right
            paint.Shader = SKShader.CreateLinearGradient(
                new SKPoint(rect.Left, rect.Top),
                new SKPoint(rect.Right, rect.Bottom),
                new SKColor[] { SKColors.Red, SKColors.Blue },
                new float[] { 0, 1 },
                SKShaderTileMode.Repeat);

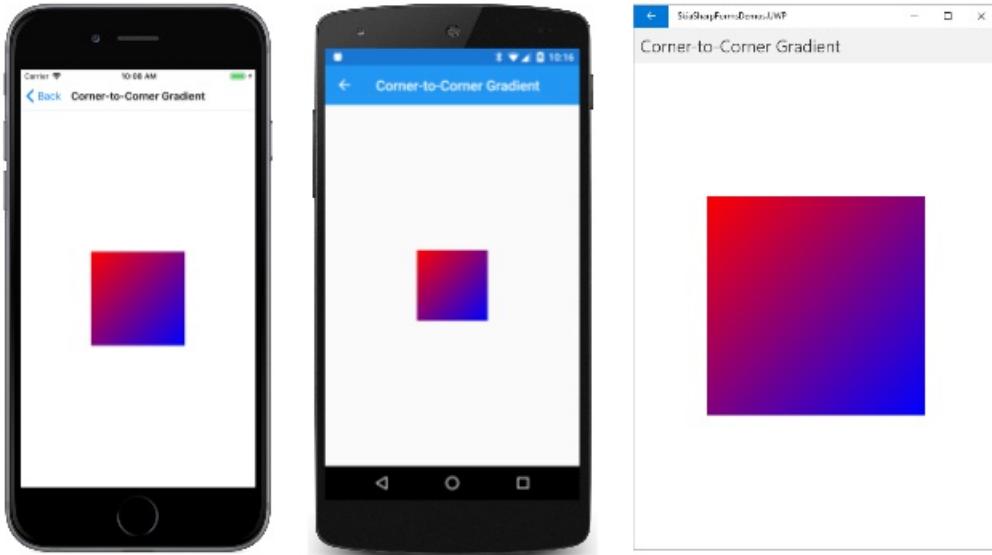
            // Draw the gradient on the rectangle
            canvas.DrawRect(rect, paint);
            ...
        }
    }
}

```

The `shader` property of `SKPaint` is assigned the `SKShader` return value from the static `SKShader.CreateLinearGradient` method. The five arguments are as follows:

- The start point of the gradient, set here to the upper-left corner of the rectangle
- The end point of the gradient, set here to the lower-right corner of the rectangle
- An array of two or more colors that contribute to the gradient
- An array of `float` values indicating the relative position of the colors within the gradient line
- A member of the `SKShaderTileMode` enumeration indicating how the gradient behaves beyond the ends of the gradient line

After the gradient object is created, the `DrawRect` method draws the 300-pixel square rectangle using the `SKPaint` object that includes the shader. Here it is running on iOS, Android, and the Universal Windows Platform (UWP):



The gradient line is defined by the two points specified as the first two arguments. Notice that these points are relative to the *canvas* and *not* to the graphical object displayed with the gradient. Along the gradient line, the color gradually transitions from red at the upper left to blue at the lower right. Any line that is perpendicular to the gradient line has a constant color.

The array of `float` values specified as the fourth argument have a one-to-one correspondence with the array of colors. The values indicate the relative position along the gradient line where those colors occur. Here, the 0 means that `Red` occurs at the start of the gradient line, and 1 means that `Blue` occurs at the end of the line. The numbers must be ascending, and should be in the range of 0 to 1. If they aren't in that range, they will be adjusted to be in that range.

The two values in the array can be set to something other than 0 and 1. Try this:

```
new float[] { 0.25f, 0.75f }
```

Now the whole first quarter of the gradient line is pure red, and the last quarter is pure blue. The mix of red and blue is restricted to the central half of the gradient line.

Generally, you'll want to space these position values equally from 0 to 1. If that is the case, you can simply supply `null` as the fourth argument to `CreateLinearGradient`.

Although this gradient is defined between two corners of the 300-pixel square rectangle, it isn't restricted to filling that rectangle. The **Corner-to-Corner Gradient** page includes some extra code that responds to taps or mouse clicks on the page. The `drawBackground` field is toggled between `true` and `false` with each tap. If the value is `true`, then the `PaintSurface` handler uses the same `SKPaint` object to fill the entire canvas, and then draws a black rectangle indicating the smaller rectangle:

```

public class CornerToCornerGradientPage : ContentPage
{
    bool drawBackground;

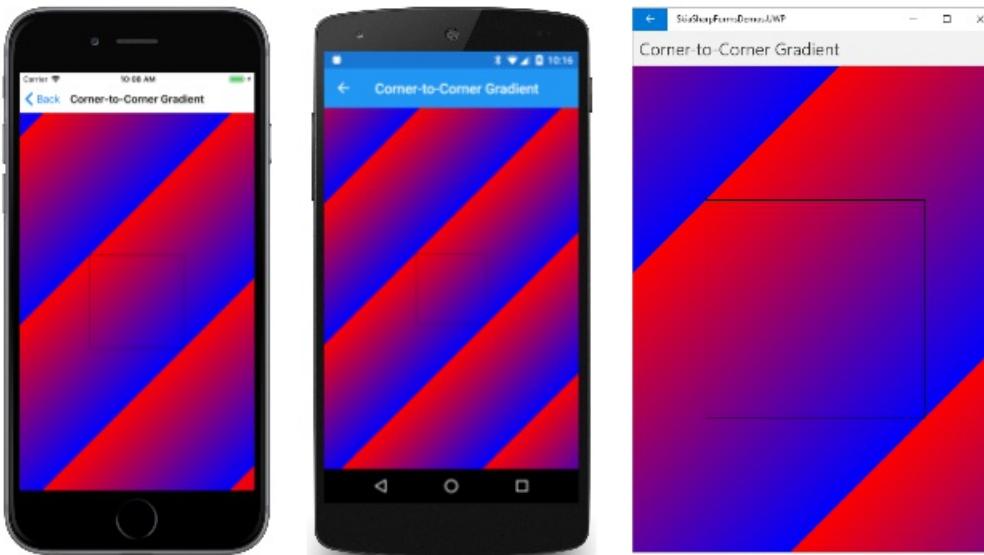
    public CornerToCornerGradientPage ()
    {
        ...
        TapGestureRecognizer tap = new TapGestureRecognizer();
        tap.Tapped += (sender, args) =>
        {
            drawBackground ^= true;
            canvasView.InvalidateSurface();
        };
        canvasView.GestureRecognizers.Add(tap);
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        ...
        using (SKPaint paint = new SKPaint())
        {
            ...
            if (drawBackground)
            {
                // Draw the gradient on the whole canvas
                canvas.DrawRect(info.Rect, paint);

                // Outline the smaller rectangle
                paint.Shader = null;
                paint.Style = SKPaintStyle.Stroke;
                paint.Color = SKColors.Black;
                canvas.DrawRect(rect, paint);
            }
        }
    }
}

```

Here's what you'll see after tapping the screen:



Notice that the gradient repeats itself in the same pattern beyond the points defining the gradient line. This repetition occurs because the last argument to `CreateLinearGradient` is `SKShaderTileMode.Repeat`. (You'll see the other options shortly.)

Also notice that the points that you use to specify the gradient line aren't unique. Lines that are perpendicular to the gradient line have the same color, so there are an infinite number of gradient lines that you can specify for

the same effect. For example, when filling a rectangle with a horizontal gradient, you can specify the upper-left and upper-right corners, or the lower-left and lower-right corners, or any two points that are even with and parallel to those lines.

Interactively experiment

You can interactively experiment with linear gradients with the **Interactive Linear Gradient** page. This page uses the `InteractivePage` class introduced in the article [Three ways to draw an arc](#). `InteractivePage` handles `TouchEvent` events to maintain a collection of `TouchPoint` objects that you can move with your fingers or the mouse.

The XAML file attaches the `TouchEvent` to a parent of the `SKCanvasView` and also includes a `Picker` that allows you to select one of the three members of the `SKShaderTileMode` enumeration:

```
<local:InteractivePage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:SkiaSharpFormsDemos"
    xmlns:skia="clr-namespace:SkiaSharp;assembly=SkiaSharp"
    xmlns:skiaforms="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    xmlns:tt="clr-namespace:TouchTracking"
    x:Class="SkiaSharpFormsDemos.Effects.InteractiveLinearGradientPage"
    Title="Interactive Linear Gradient">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Grid BackgroundColor="White"
            Grid.Row="0">
            <skiaforms:SKCanvasView x:Name="canvasView"
                PaintSurface="OnCanvasViewPaintSurface" />
            <Grid.Effects>
                <tt:TouchEvent Capture="True"
                    TouchAction="OnTouchEventAction" />
            </Grid.Effects>
        </Grid>

        <Picker x:Name="tileModePicker"
            Grid.Row="1"
            Title="Shader Tile Mode"
            Margin="10"
            SelectedIndexChanged="OnPickerSelectedIndexChanged">
            <Picker.ItemsSource>
                <x:Array Type="{x:Type skia:SKShaderTileMode}">
                    <x:Static Member="skia:SKShaderTileMode.Clamp" />
                    <x:Static Member="skia:SKShaderTileMode.Repeat" />
                    <x:Static Member="skia:SKShaderTileMode.Mirror" />
                </x:Array>
            </Picker.ItemsSource>

            <Picker.SelectedIndex>
                0
            </Picker.SelectedIndex>
        </Picker>
    </Grid>
</local:InteractivePage>
```

The constructor in the code-behind file creates two `TouchPoint` objects for the start and end points of the linear gradient. The `PaintSurface` handler defines an array of three colors (for a gradient from red to green to blue) and obtains the current `SKShaderTileMode` from the `Picker`:

```

public partial class InteractiveLinearGradientPage : InteractivePage
{
    public InteractiveLinearGradientPage ()
    {
        InitializeComponent ();

        touchPoints = new TouchPoint[2];

        for (int i = 0; i < 2; i++)
        {
            touchPoints[i] = new TouchPoint
            {
                Center = new SKPoint(100 + i * 200, 100 + i * 200)
            };
        }

        InitializeComponent();
        baseCanvasView = canvasView;
    }

    void OnPickerSelectedIndexChanged(object sender, EventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        SKColor[] colors = { SKColors.Red, SKColors.Green, SKColors.Blue };
        SKShaderTileMode tileMode =
            (SKShaderTileMode)(tileModePicker.SelectedIndex == -1 ?
                0 : tileModePicker.SelectedItem);

        using (SKPaint paint = new SKPaint())
        {
            paint.Shader = SKShader.CreateLinearGradient(touchPoints[0].Center,
                touchPoints[1].Center,
                colors,
                null,
                tileMode);
            canvas.DrawRect(info.Rect, paint);
        }
        ...
    }
}

```

The `PaintSurface` handler creates the `skShader` object from all that information, and uses it to color the entire canvas. The array of `float` values is set to `null`. Otherwise, to equally space three colors, you'd set that parameter to an array with the values 0, 0.5, and 1.

The bulk of the `PaintSurface` handler is devoted to displaying several objects: the touch points as outline circles, the gradient line, and the lines perpendicular to the gradient lines at the touch points:

```

public partial class InteractiveLinearGradientPage : InteractivePage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        ...
        // Display the touch points here rather than by TouchPoint
        using (SKPaint paint = new SKPaint())
        {
            paint.Style = SKPaintStyle.Stroke;
            paint.Color = SKColors.Black;
            paint.StrokeWidth = 3;

            foreach (TouchPoint touchPoint in touchPoints)
            {
                canvas.DrawCircle(touchPoint.Center, touchPoint.Radius, paint);
            }

            // Draw gradient line connecting touchpoints
            canvas.DrawLine(touchPoints[0].Center, touchPoints[1].Center, paint);

            // Draw lines perpendicular to the gradient line
            SKPoint vector = touchPoints[1].Center - touchPoints[0].Center;
            float length = (float)Math.Sqrt(Math.Pow(vector.X, 2) +
                Math.Pow(vector.Y, 2));
            vector.X /= length;
            vector.Y /= length;
            SKPoint rotate90 = new SKPoint(-vector.Y, vector.X);
            rotate90.X *= 200;
            rotate90.Y *= 200;

            canvas.DrawLine(touchPoints[0].Center,
                touchPoints[0].Center + rotate90,
                paint);

            canvas.DrawLine(touchPoints[0].Center,
                touchPoints[0].Center - rotate90,
                paint);

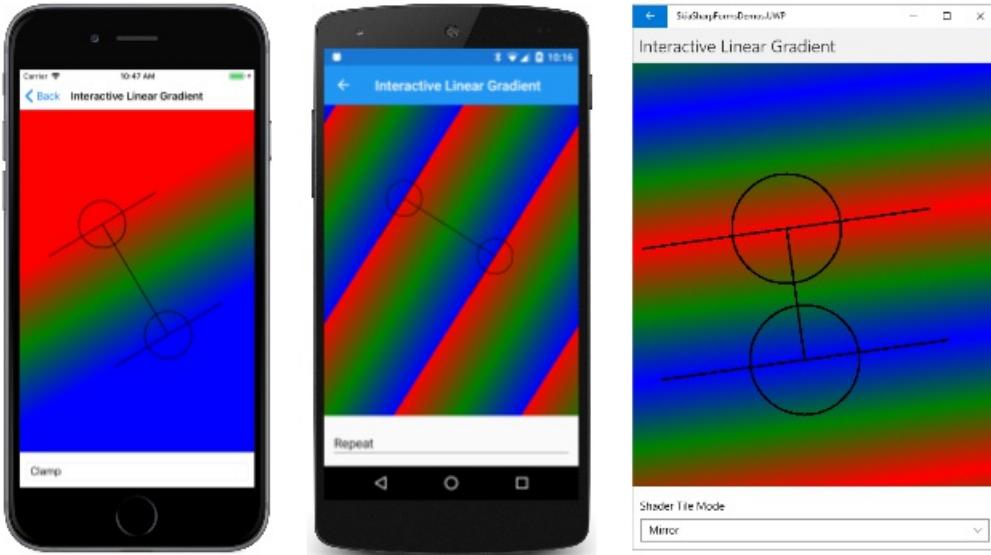
            canvas.DrawLine(touchPoints[1].Center,
                touchPoints[1].Center + rotate90,
                paint);

            canvas.DrawLine(touchPoints[1].Center,
                touchPoints[1].Center - rotate90,
                paint);
        }
    }
}

```

The gradient line connecting the two touchpoints is easy to draw, but the perpendicular lines require some more work. The gradient line is converted to a vector, normalized to have a length of one unit, and then rotated by 90 degrees. That vector is then given a length of 200 pixels. It's used to draw four lines that extend from the touch points to be perpendicular to the gradient line.

The perpendicular lines coincide with the beginning and end of the gradient. What happens beyond those lines depends on the setting of the `SKShaderTileMode` enumeration:

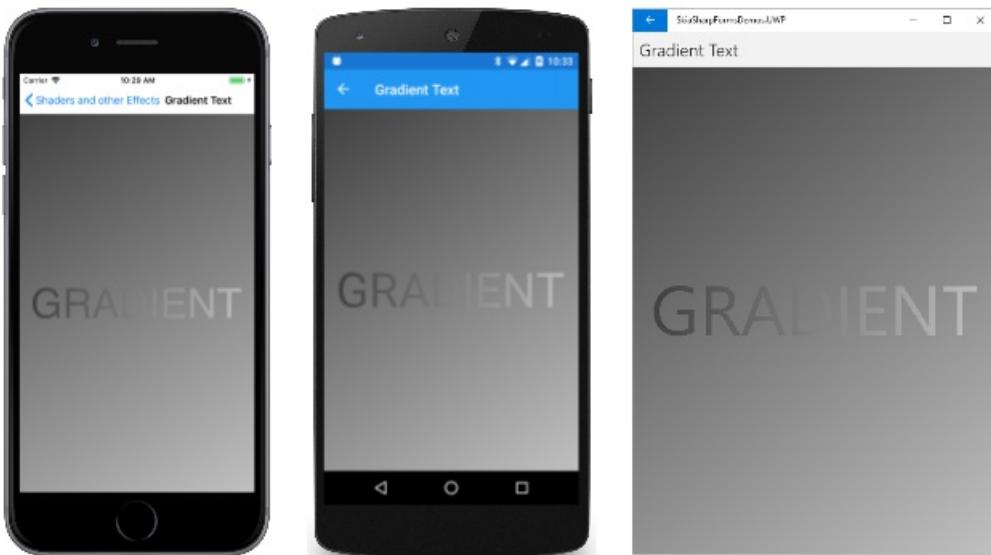


The three screenshots show the results of the three different values of `SKShaderTileMode`. The iOS screenshot shows `SKShaderTileMode.Clamp`, which just extends the colors on the border of the gradient. The `SKShaderTileMode.Repeat` option in the Android screenshot shows how the gradient pattern is repeated. The `SKShaderTileMode.Mirror` option in the UWP screenshot also repeats the pattern, but the pattern is reversed each time, resulting in no color discontinuities.

Gradients on gradients

The `skShader` class defines no public properties or methods except for `Dispose`. The `skShader` objects that created by its static methods are therefore immutable. Even if you use the same gradient for two different objects, it's likely you'll want to vary the gradient slightly. To do that, you'll need to create a new `skShader` object.

The [Gradient Text](#) page displays text and a background that are both colored with similar gradients:



The only differences in the gradients are the start and end points. The gradient used for displaying text is based on two points on the corners of the bounding rectangle for the text. For the background, the two points are based on the entire canvas. Here's the code:

```

public class GradientTextPage : ContentPage
{
    const string TEXT = "GRADIENT";

    public GradientTextPage ()
    {
        Title = "Gradient Text";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        using (SKPaint paint = new SKPaint())
        {
            // Create gradient for background
            paint.Shader = SKShader.CreateLinearGradient(
                new SKPoint(0, 0),
                new SKPoint(info.Width, info.Height),
                new SKColor[] { new SKColor(0x40, 0x40, 0x40),
                               new SKColor(0xC0, 0xC0, 0xC0) },
                null,
                SKShaderTileMode.Clamp);

            // Draw background
            canvas.DrawRect(info.Rect, paint);

            // Set TextSize to fill 90% of width
            paint.TextSize = 100;
            float width = paint.MeasureText(TEXT);
            float scale = 0.9f * info.Width / width;
            paint.TextSize *= scale;

            // Get text bounds
            SKRect textBounds = new SKRect();
            paint.MeasureText(TEXT, ref textBounds);

            // Calculate offsets to center the text on the screen
            float xText = info.Width / 2 - textBounds.MidX;
            float yText = info.Height / 2 - textBounds.MidY;

            // Shift textBounds by that amount
            textBounds.Offset(xText, yText);

            // Create gradient for text
            paint.Shader = SKShader.CreateLinearGradient(
                new SKPoint(textBounds.Left, textBounds.Top),
                new SKPoint(textBounds.Right, textBounds.Bottom),
                new SKColor[] { new SKColor(0x40, 0x40, 0x40),
                               new SKColor(0xC0, 0xC0, 0xC0) },
                null,
                SKShaderTileMode.Clamp);

            // Draw text
            canvas.DrawText(TEXT, xText, yText, paint);
        }
    }
}

```

The `shader` property of the `SKPaint` object is set first to display a gradient to cover the background. The gradient points are set to the upper-left and lower-right corners of the canvas.

The code sets the `TextSize` property of the `SKPaint` object so that the text is displayed at 90% of the width of the canvas. The text bounds are used to calculate `xText` and `yText` values to pass to the `DrawText` method to center the text.

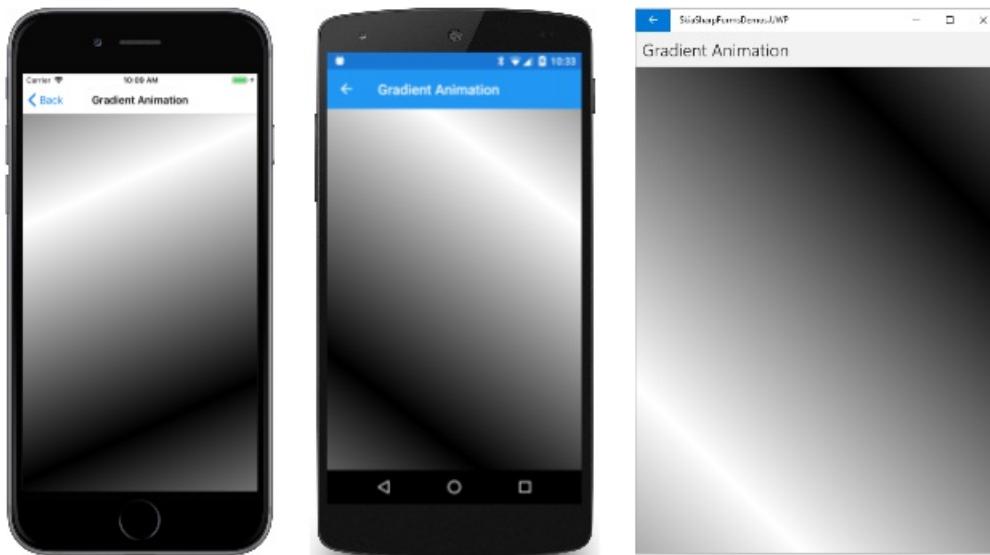
However, the gradient points for the second `createLinearGradient` call must refer to the upper-left and lower-right corner of the text relative to the canvas when it's displayed. This is accomplished by shifting the `textBounds` rectangle by the same `xText` and `yText` values:

```
textBounds.Offset(xText, yText);
```

Now the upper-left and lower-right corners of the rectangle can be used to set the start and end points of the gradient.

Animating a gradient

There are several ways to animate a gradient. One approach is to animate the start and end points. The **Gradient Animation** page moves the two points around in a circle that is centered on the canvas. The radius of this circle is half the width or height of the canvas, whichever is smaller. The start and end points are opposite each other on this circle, and the gradient goes from white to black with a `Mirror` tile mode:



The constructor creates the `SKCanvasView`. The `OnAppearing` and `OnDisappearing` methods handle the animation logic:

```

public class GradientAnimationPage : ContentPage
{
    SKCanvasView canvasView;
    bool isAnimating;
    double angle;
    Stopwatch stopwatch = new Stopwatch();

    public GradientAnimationPage()
    {
        Title = "Gradient Animation";

        canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    protected override void OnAppearing()
    {
        base.OnAppearing();

        isAnimating = true;
        stopwatch.Start();
        Device.StartTimer(TimeSpan.FromMilliseconds(16), OnTimerTick);
    }

    protected override void OnDisappearing()
    {
        base.OnDisappearing();

        stopwatch.Stop();
        isAnimating = false;
    }

    bool OnTimerTick()
    {
        const int duration = 3000;
        angle = 2 * Math.PI * (stopwatch.ElapsedMilliseconds % duration) / duration;
        canvasView.InvalidateSurface();

        return isAnimating;
    }
    ...
}

```

The `OnTimerTick` method calculates an `angle` value that is animated from 0 to 2π every 3 seconds.

Here's one way to calculate the two gradient points. An `SKPoint` value named `vector` is calculated to extend from the center of the canvas to a point on the radius of the circle. The direction of this vector is based on the sine and cosine values of the angle. The two opposite gradient points are then calculated: One point is calculated by subtracting that vector from the center point, and other point is calculated by adding the vector to the center point:

```

public class GradientAnimationPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        using (SKPaint paint = new SKPaint())
        {
            SKPoint center = new SKPoint(info.Rect.MidX, info.Rect.MidY);
            int radius = Math.Min(info.Width, info.Height) / 2;
            SKPoint vector = new SKPoint((float)(radius * Math.Cos(angle)),
                                         (float)(radius * Math.Sin(angle)));

            paint.Shader = SKShader.CreateLinearGradient(
                center - vector,
                center + vector,
                new SKColor[] { SKColors.White, SKColors.Black },
                null,
                SKShaderTileMode.Mirror);

            canvas.DrawRect(info.Rect, paint);
        }
    }
}

```

A somewhat different approach requires less code. This approach makes use of the [SKShader.CreateLinearGradient](#) overload method with a matrix transform as the last argument. This approach is the version in the [SkiaSharpFormsDemos](#) sample:

```

public class GradientAnimationPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        using (SKPaint paint = new SKPaint())
        {
            paint.Shader = SKShader.CreateLinearGradient(
                new SKPoint(0, 0),
                info.Width < info.Height ? new SKPoint(info.Width, 0) :
                                            new SKPoint(0, info.Height),
                new SKColor[] { SKColors.White, SKColors.Black },
                new float[] { 0, 1 },
                SKShaderTileMode.Mirror,
                SKMatrix.MakeRotation((float)angle, info.Rect.MidX, info.Rect.MidY));

            canvas.DrawRect(info.Rect, paint);
        }
    }
}

```

If the width of the canvas is less than the height, then the two gradient points are set to (0, 0) and (`info.Width`, 0). The rotation transform passed as the last argument to [CreateLinearGradient](#) effectively rotates those two

points around the center of the screen.

Note that if the angle is 0, there's no rotation, and the two gradient points are the upper-left and upper-right corners of the canvas. Those points aren't the same gradient points calculated as shown in the previous `CreateLinearGradient` call. But these points are *parallel* to the horizontal gradient line that bisects the center of the canvas, and they result in an identical gradient.

Rainbow Gradient

The **Rainbow Gradient** page draws a rainbow from the upper-left corner of the canvas to the lower-right corner. But this rainbow gradient isn't like a real rainbow. It's straight rather than curved, but it's based on eight HSL (hue-saturation-luminosity) colors that are determined by cycling through hue values from 0 to 360:

```
SKColor[] colors = new SKColor[8];

for (int i = 0; i < colors.Length; i++)
{
    colors[i] = SKColor.FromHsl(i * 360f / (colors.Length - 1), 100, 50);
}
```

That code is part of the `PaintSurface` handler shown below. The handler begins by creating a path that defines a six-sided polygon that extends from the upper-left corner of the canvas to the lower-right corner:

```

public class RainbowGradientPage : ContentPage
{
    public RainbowGradientPage ()
    {
        Title = "Rainbow Gradient";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        using (SKPath path = new SKPath())
        {
            float rainbowWidth = Math.Min(info.Width, info.Height) / 2f;

            // Create path from upper-left to lower-right corner
            path.MoveTo(0, 0);
            path.LineTo(rainbowWidth / 2, 0);
            path.LineTo(info.Width, info.Height - rainbowWidth / 2);
            path.LineTo(info.Width, info.Height);
            path.LineTo(info.Width - rainbowWidth / 2, info.Height);
            path.LineTo(0, rainbowWidth / 2);
            path.Close();

            using (SKPaint paint = new SKPaint())
            {
                SKColor[] colors = new SKColor[8];

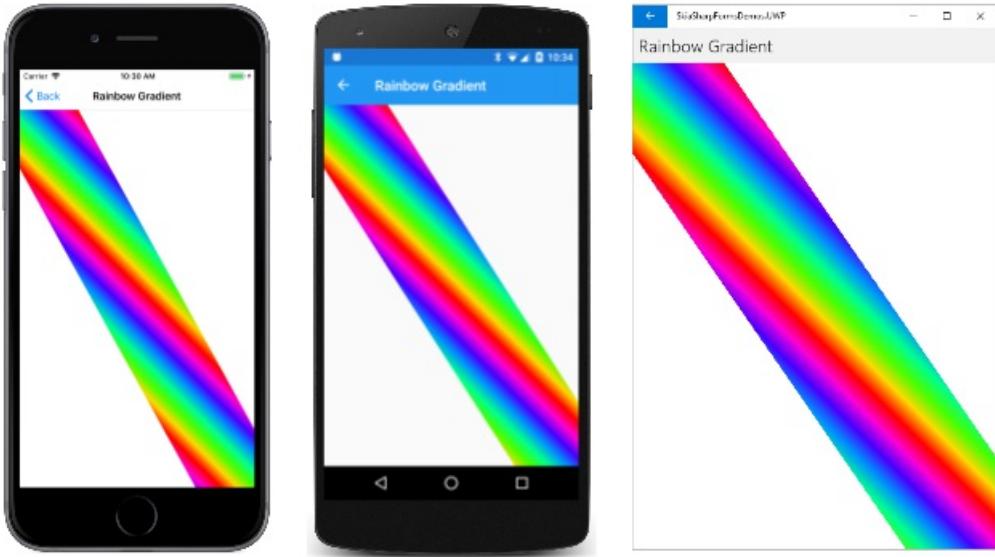
                for (int i = 0; i < colors.Length; i++)
                {
                    colors[i] = SKColor.FromHsl(i * 360f / (colors.Length - 1), 100, 50);
                }

                paint.Shader = SKShader.CreateLinearGradient(
                    new SKPoint(0, rainbowWidth / 2),
                    new SKPoint(rainbowWidth / 2, 0),
                    colors,
                    null,
                    SKShaderTileMode.Repeat);

                canvas.DrawPath(path, paint);
            }
        }
    }
}

```

The two gradient points in the `CreateLinearGradient` method are based on two of the points that define this path: Both points are close to the upper-left corner. The first is on the upper edge of the canvas and the second is on the left edge of the canvas. Here's the result:



This is an interesting image, but it's not quite the intent. The problem is that when creating a linear gradient, the lines of constant color are perpendicular to the gradient line. The gradient line is based on the points where the figure touches the top and left sides, and that line is generally not perpendicular to the edges of the figure that extend to the bottom-right corner. This approach would work only if the canvas were square.

To create a proper rainbow gradient, the gradient line must be perpendicular to the edge of the rainbow. That's a more involved calculation. A vector must be defined that is parallel to the long side of the figure. The vector is rotated 90 degrees so that it's perpendicular to that side. It is then lengthened to be the width of the figure by multiplying by `rainbowWidth`. The two gradient points are calculated based on a point on the side of the figure, and that point plus the vector. Here is the code that appears in the **Rainbow Gradient** page in the [SkiaSharpFormsDemos](#) sample:

```

public class RainbowGradientPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        ...
        using (SKPath path = new SKPath())
        {
            ...
            using (SKPaint paint = new SKPaint())
            {
                ...
                // Vector on lower-left edge, from top to bottom
                SKPoint edgeVector = new SKPoint(info.Width - rainbowWidth / 2, info.Height) -
                    new SKPoint(0, rainbowWidth / 2);

                // Rotate 90 degrees counter-clockwise:
                SKPoint gradientVector = new SKPoint(edgeVector.Y, -edgeVector.X);

                // Normalize
                float length = (float)Math.Sqrt(Math.Pow(gradientVector.X, 2) +
                    Math.Pow(gradientVector.Y, 2));
                gradientVector.X /= length;
                gradientVector.Y /= length;

                // Make it the width of the rainbow
                gradientVector.X *= rainbowWidth;
                gradientVector.Y *= rainbowWidth;

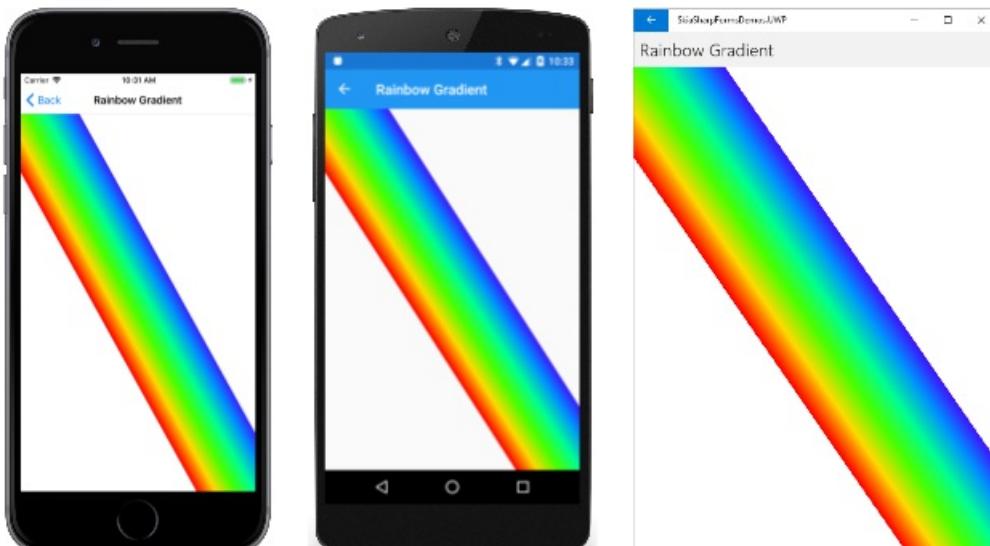
                // Calculate the two points
                SKPoint point1 = new SKPoint(0, rainbowWidth / 2);
                SKPoint point2 = point1 + gradientVector;

                paint.Shader = SKShader.CreateLinearGradient(point1,
                    point2,
                    colors,
                    null,
                    SKShaderTileMode.Repeat);

                canvas.DrawPath(path, paint);
            }
        }
    }
}

```

Now the rainbow colors are aligned with the figure:



Infinity Colors

A rainbow gradient is also used in the [Infinity Colors](#) page. This page draws an infinity sign using a path object described in the article [Three Types of Bézier Curves](#). The image is then colored with an animated rainbow gradient that continuously sweeps across the image.

The constructor creates the `SKPath` object describing the infinity sign. After the path is created, the constructor can also obtain the rectangular bounds of the path. It then calculates a value called `gradientCycleLength`. If a gradient is based on the upper-left and lower-right corners of the `pathBounds` rectangle, this `gradientCycleLength` value is the total horizontal width of the gradient pattern:

```
public class InfinityColorsPage : ContentPage
{
    ...
    SKCanvasView canvasView;

    // Path information
    SKPath infinityPath;
    SKRect pathBounds;
    float gradientCycleLength;

    // Gradient information
    SKColor[] colors = new SKColor[8];
    ...

    public InfinityColorsPage ()
    {
        Title = "Infinity Colors";

        // Create path for infinity sign
        infinityPath = new SKPath();
        infinityPath.MoveTo(0, 0);                                // Center
        infinityPath.CubicTo( 50, -50,  95, -100, 150, -100); // To top of right loop
        infinityPath.CubicTo( 205, -100, 250, -55, 250,     0); // To far right of right loop
        infinityPath.CubicTo( 250,  55, 205,  100, 150,  100); // To bottom of right loop
        infinityPath.CubicTo( 95,  100, 50,  50, 0, 0);           // Back to center
        infinityPath.CubicTo( -50, -50, -95, -100, -150, -100); // To top of left loop
        infinityPath.CubicTo(-205, -100, -250, -55, -250,     0); // To far left of left loop
        infinityPath.CubicTo(-250,  55, -205,  100, -150,  100); // To bottom of left loop
        infinityPath.CubicTo( -95,  100, -50,  50, 0, 0);          // Back to center
        infinityPath.Close();

        // Calculate path information
        pathBounds = infinityPath.Bounds;
        gradientCycleLength = pathBounds.Width +
            pathBounds.Height * pathBounds.Height / pathBounds.Width;

        // Create SKColor array for gradient
        for (int i = 0; i < colors.Length; i++)
        {
            colors[i] = SKColor.FromHsl(i * 360f / (colors.Length - 1), 100, 50);
        }

        canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }
    ...
}
```

The constructor also creates the `colors` array for the rainbow, and the `SKCanvasView` object.

Overrides of the `OnAppearing` and `OnDisappearing` methods perform the overhead for the animation. The `OnTimerTick` method animates the `offset` field from 0 to `gradientCycleLength` every two seconds:

```

public class InfinityColorsPage : ContentPage
{
    ...
    // For animation
    bool isAnimating;
    float offset;
    Stopwatch stopwatch = new Stopwatch();
    ...

    protected override void OnAppearing()
    {
        base.OnAppearing();

        isAnimating = true;
        stopwatch.Start();
        Device.StartTimer(TimeSpan.FromMilliseconds(16), OnTimerTick);
    }

    protected override void OnDisappearing()
    {
        base.OnDisappearing();

        stopwatch.Stop();
        isAnimating = false;
    }

    bool OnTimerTick()
    {
        const int duration = 2;      // seconds
        double progress = stopwatch.Elapsed.TotalSeconds % duration / duration;
        offset = (float)(gradientCycleLength * progress);
        canvasView.InvalidateSurface();

        return isAnimating;
    }
    ...
}

```

Finally, the `PaintSurface` handler renders the infinity sign. Because the path contains negative and positive coordinates surrounding a center point of (0, 0), a `Translate` transform on the canvas is used to shift it to the center. The translate transform is followed by a `Scale` transform that applies a scaling factor that makes the infinity sign as large as possible while still staying within 95% of the width and height of the canvas.

Notice that the `STROKE_WIDTH` constant is added to the width and height of the path bounding rectangle. The path will be stroked with a line of this width, so the size of the rendered infinity size is increased by half that width on all four sides:

```

public class InfinityColorsPage : ContentPage
{
    const int STROKE_WIDTH = 50;
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Set transforms to shift path to center and scale to canvas size
        canvas.Translate(info.Width / 2, info.Height / 2);
        canvas.Scale(0.95f *
            Math.Min(info.Width / (pathBounds.Width + STROKE_WIDTH),
                     info.Height / (pathBounds.Height + STROKE_WIDTH)));

        using (SKPaint paint = new SKPaint())
        {
            paint.Style = SKPaintStyle.Stroke;
            paint.StrokeWidth = STROKE_WIDTH;
            paint.Shader = SKShader.CreateLinearGradient(
                new SKPoint(pathBounds.Left, pathBounds.Top),
                new SKPoint(pathBounds.Right, pathBounds.Bottom),
                colors,
                null,
                SKShaderTileMode.Repeat,
                SKMatrix.MakeTranslation(offset, 0));

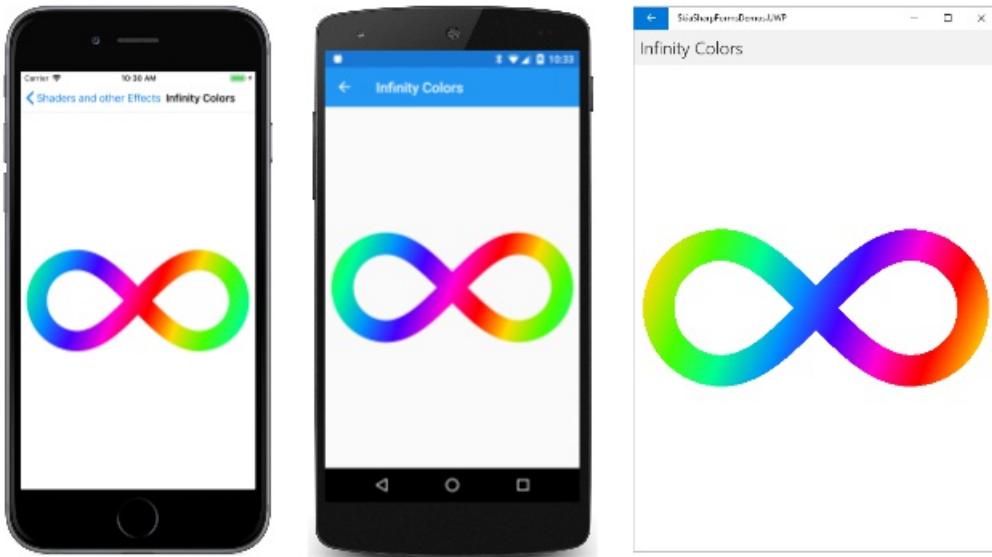
            canvas.DrawPath(infinityPath, paint);
        }
    }
}

```

Look at the points passed as the first two arguments of `SKShader.CreateLinearGradient`. Those points are based on the original path bounding rectangle. The first point is $(-250, -100)$ and the second is $(250, 100)$. Internal to SkiaSharp, those points are subjected to the current canvas transform so they align correctly with the displayed infinity sign.

Without the last argument to `CreateLinearGradient`, you'd see a rainbow gradient that extends from the upper left of the infinity sign to the lower right. (Actually, the gradient extends from the upper-left corner to the lower-right corner of the bounding rectangle. The rendered infinity sign is greater than the bounding rectangle by half the `STROKE_WIDTH` value on all sides. Because the gradient is red at both the beginning and end, and the gradient is created with `SKShaderTileMode.Repeat`, the difference isn't noticeable.)

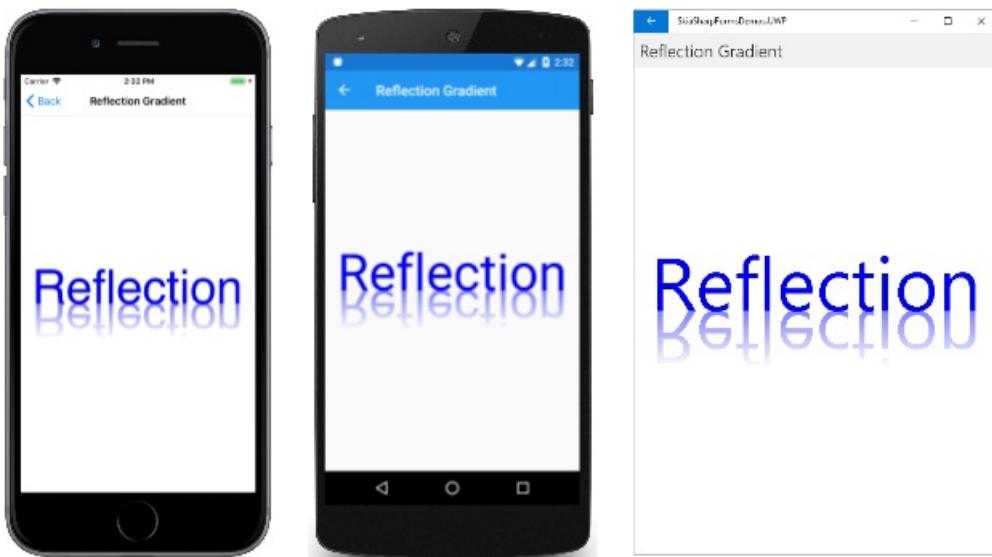
With that last argument to `CreateLinearGradient`, the gradient pattern continuously sweeps across the image:



Transparency and gradients

The colors that contribute to a gradient can incorporate transparency. Instead of a gradient that fades from one color to another, the gradient can fade from a color to transparent.

You can use this technique for some interesting effects. One of the classic examples shows a graphical object with its reflection:



The text that is upside-down is colored with a gradient that is 50% transparent at the top to fully transparent at the bottom. These levels of transparency are associated with alpha values of 0x80 and 0.

The `PaintSurface` handler in the **Reflection Gradient** page scales the size of the text to 90% of the width of the canvas. It then calculates `xText` and `yText` values to position the text to be horizontally centered but sitting on a baseline corresponding to the vertical center of the page:

```

public class ReflectionGradientPage : ContentPage
{
    const string TEXT = "Reflection";

    public ReflectionGradientPage ()
    {
        Title = "Reflection Gradient";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        using (SKPaint paint = new SKPaint())
        {
            // Set text color to blue
            paint.Color = SKColors.Blue;

            // Set text size to fill 90% of width
            paint.TextSize = 100;
            float width = paint.MeasureText(TEXT);
            float scale = 0.9f * info.Width / width;
            paint.TextSize *= scale;

            // Get text bounds
            SKRect textBounds = new SKRect();
            paint.MeasureText(TEXT, ref textBounds);

            // Calculate offsets to position text above center
            float xText = info.Width / 2 - textBounds.MidX;
            float yText = info.Height / 2;

            // Draw unreflected text
            canvas.DrawText(TEXT, xText, yText, paint);

            // Shift textBounds to match displayed text
            textBounds.Offset(xText, yText);

            // Use those offsets to create a gradient for the reflected text
            paint.Shader = SKShader.CreateLinearGradient(
                new SKPoint(0, textBounds.Top),
                new SKPoint(0, textBounds.Bottom),
                new SKColor[] { paint.Color.WithAlpha(0),
                               paint.Color.WithAlpha(0x80) },
                null,
                SKShaderTileMode.Clamp);

            // Scale the canvas to flip upside-down around the vertical center
            canvas.Scale(1, -1, 0, yText);

            // Draw reflected text
            canvas.DrawText(TEXT, xText, yText, paint);
        }
    }
}

```

Those `xText` and `yText` values are the same values used to display the reflected text in the `DrawText` call at the bottom of the `PaintSurface` handler. Just before that code, however, you'll see a call to the `Scale` method of

`SKCanvas`. This `Scale` method scales horizontally by 1 (which does nothing) but vertically by -1 , which effectively flips everything upside-down. The center of rotation is set to the point $(0, \text{yText})$, where `yText` is the vertical center of the canvas, originally calculated as `info.Height` divided by 2.

Keep in mind that Skia uses the gradient to color graphical objects prior to the canvas transforms. After the unreflected text is drawn, the `textBounds` rectangle is shifted so it corresponds to the displayed text:

```
textBounds.Offset(xText, yText);
```

The `CreateLinearGradient` call defines a gradient from the top of that rectangle to the bottom. The gradient is from a completely transparent blue (`paint.Color.WithAlpha(0)`) to a 50% transparent blue (`paint.Color.WithAlpha(0x80)`). The canvas transform flips the text upside-down, so the 50% transparent blue starts at the baseline, and becomes transparent at the top of the text.

Related links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

The SkiaSharp circular gradients

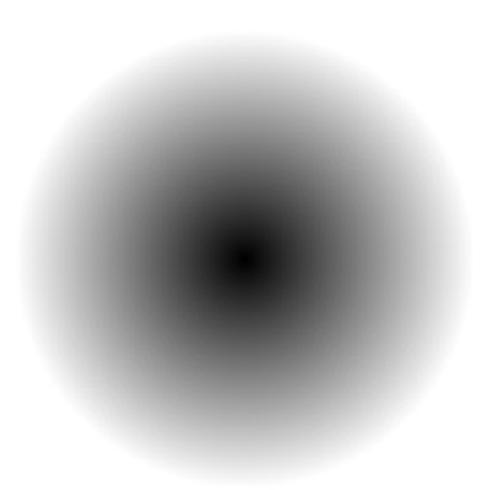
3/5/2021 • 15 minutes to read • [Edit Online](#)



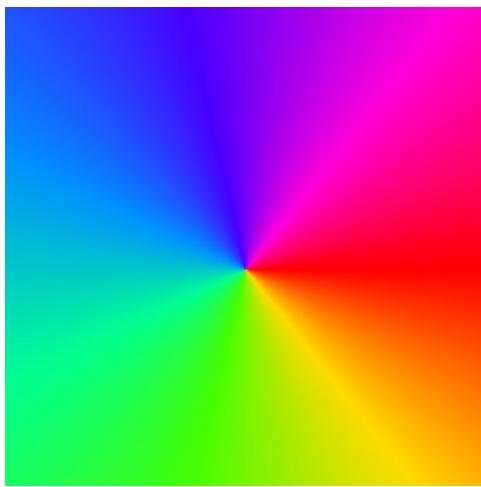
[Download the sample](#)

The `SKShader` class defines static methods to create four different types of gradients. The [SkiaSharp linear gradient](#) article discusses the `CreateLinearGradient` method. This article covers the other three types of gradients, all of which are based on circles.

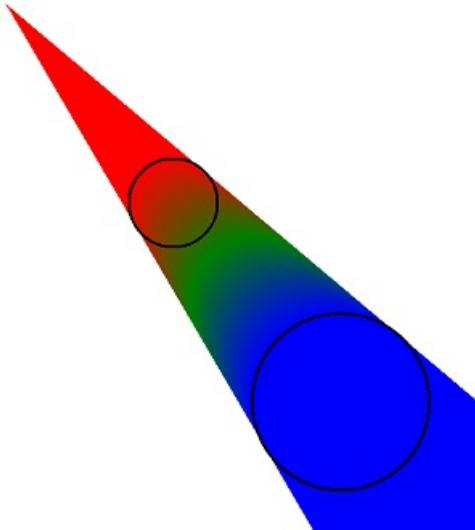
The `CreateRadialGradient` method creates a gradient that emanates from the center of a circle:



The `CreateSweepGradient` method creates a gradient that sweeps around the center of a circle:



The third type of gradient is quite unusual. It is called the two-point conical gradient and is defined by the `CreateTwoPointConicalGradient` method. The gradient extends from one circle to another:



If the two circles are different sizes, then the gradient takes the form of a cone.

This article explores these gradients in more detail.

The radial gradient

The `CreateRadialGradient` method has the following syntax:

```
public static SKShader CreateRadialGradient (SKPoint center,
                                             Single radius,
                                             SKColor[] colors,
                                             Single[] colorPos,
                                             SKShaderTileMode mode)
```

A `CreateRadialGradient` overload also includes a transform matrix parameter.

The first two arguments specify the center of a circle and a radius. The gradient begins at that center and extends outward for `radius` pixels. What happens beyond `radius` depends on the `SKShaderTileMode` argument. The `colors` parameter is an array of two or more colors (just as in the linear gradient methods), and `colorPos` is an array of integers in the range of 0 to 1. These integers indicate the relative positions of the colors along that `radius` line. You can set that argument to `null` to equally space the colors.

If you use `CreateRadialGradient` to fill a circle, you can set the center of the gradient to the center of the circle, and the radius of the gradient to the radius of the circle. In that case, the `skShaderTileMode` argument has no effect on the rendering of the gradient. But if the area filled by the gradient is larger than the circle defined by the gradient, then the `skShaderTileMode` argument has a profound effect on what happens outside the circle.

The effect of `skShaderMode` is demonstrated in the **Radial Gradient** page in the [SkiaSharpFormsDemos](#) sample. The XAML file for this page instantiates a `Picker` that allows you to select one of the three members of the `SKShaderTileMode` enumeration:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp;assembly=SkiaSharp"
    xmlns:skiaforms="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Effects.RadialGradientPage"
    Title="Radial Gradient">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <skiaforms:SKCanvasView x:Name="canvasView"
            Grid.Row="0"
            PaintSurface="OnCanvasViewPaintSurface" />

        <Picker x:Name="tileModePicker"
            Grid.Row="1"
            Title="Shader Tile Mode"
            Margin="10"
            SelectedIndexChanged="OnPickerSelectedIndexChanged">
            <Picker.ItemsSource>
                <x:Array Type="{x:Type skia:SKShaderTileMode}">
                    <x:Static Member="skia:SKShaderTileMode.Clamp" />
                    <x:Static Member="skia:SKShaderTileMode.Repeat" />
                    <x:Static Member="skia:SKShaderTileMode.Mirror" />
                </x:Array>
            </Picker.ItemsSource>
            <Picker.SelectedIndex>
                0
            </Picker.SelectedIndex>
        </Picker>
    </Grid>
</ContentPage>

```

The code-behind file colors the entire canvas with a radial gradient. The center of the gradient is set to the center of the canvas, and the radius is set to 100 pixels. The gradient consists of just two colors, black and white:

```

public partial class RadialGradientPage : ContentPage
{
    public RadialGradientPage ()
    {
        InitializeComponent ();
    }

    void OnPickerSelectedIndexChanged(object sender, EventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

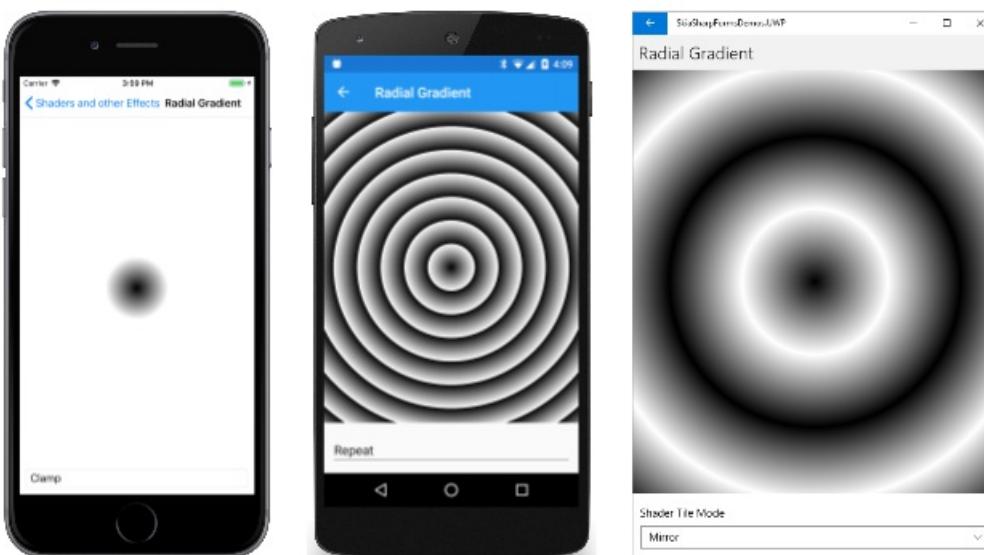
        SKShaderTileMode tileMode =
            (SKShaderTileMode)(tileModePicker.SelectedIndex == -1 ?
                0 : tileModePicker.SelectedItem);

        using (SKPaint paint = new SKPaint())
        {
            paint.Shader = SKShader.CreateRadialGradient(
                new SKPoint(info.Rect.MidX, info.Rect.MidY),
                100,
                new SKColor[] { SKColors.Black, SKColors.White },
                null,
                tileMode);

            canvas.DrawRect(info.Rect, paint);
        }
    }
}

```

This code creates a gradient with black at the center, gradually fading to white 100 pixels from the center. What happens beyond that radius depends on the `skShaderTileMode` argument:



In all three cases, the gradient fills the canvas. On the iOS screen at the left, the gradient beyond the radius continues with the last color, which is white. That's the result of `skShaderTileMode.Clamp`. The Android screen shows the effect of `SKShaderTileMode.Repeat`: At 100 pixels from the center, the gradient begins again with the first color, which is black. The gradient repeats every 100 pixels of radius.

The Universal Windows Platform screen at the right shows how `SKShader.TileMode.Mirror` causes the gradients to alternate directions. The first gradient is from black at the center to white at a radius of 100 pixels. The next is white from the 100-pixel radius to black at a 200-pixel radius, and the next gradient is reversed again.

You can use more than two colors in a radial gradient. The **Rainbow Arc Gradient** sample creates an array of eight colors corresponding to the colors of the rainbow and ending with red, and also an array of eight position values:

```
public class RainbowArcGradientPage : ContentPage
{
    public RainbowArcGradientPage ()
    {
        Title = "Rainbow Arc Gradient";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        using (SKPaint paint = new SKPaint())
        {
            float rainbowWidth = Math.Min(info.Width, info.Height) / 4f;

            // Center of arc and gradient is lower-right corner
            SKPoint center = new SKPoint(info.Width, info.Height);

            // Find outer, inner, and middle radius
            float outerRadius = Math.Min(info.Width, info.Height);
            float innerRadius = outerRadius - rainbowWidth;
            float radius = outerRadius - rainbowWidth / 2;

            // Calculate the colors and positions
            SKColor[] colors = new SKColor[8];
            float[] positions = new float[8];

            for (int i = 0; i < colors.Length; i++)
            {
                colors[i] = SKColor.FromHsl(i * 360f / 7, 100, 50);
                positions[i] = (i + (7f - i) * innerRadius / outerRadius) / 7f;
            }

            // Create sweep gradient based on center and outer radius
            paint.Shader = SKShader.CreateRadialGradient(center,
                outerRadius,
                colors,
                positions,
                SKShaderTileMode.Clamp);

            // Draw a circle with a wide line
            paint.Style = SKPaintStyle.Stroke;
            paint.StrokeWidth = rainbowWidth;

            canvas.DrawCircle(center, radius, paint);
        }
    }
}
```

Suppose the minimum of the width and height of the canvas is 1000, which means that the `rainbowWidth` value

is 250. The `outerRadius` and `innerRadius` values are set to 1000 and 750, respectively. These values are used for calculating the `positions` array; the eight values range from 0.75f to 1. The `radius` value is used for stroking the circle. The value of 875 means that the 250-pixel stroke width extends between the radius of 750 pixels and the radius of 1000 pixels:



If you filled the whole canvas with this gradient, you'd see that it's red within the inner radius. This is because the `positions` array doesn't start with 0. The first color is used for offsets of 0 through the first array value. The gradient is also red beyond the outer radius. That's the result of the `Clamp` tile mode. Because the gradient is used for stroking a thick line, these red areas aren't visible.

Radial gradients for masking

Like linear gradients, radial gradients can incorporate transparent or partially transparent colors. This feature is useful for a process called *masking*, which hides part of an image to accentuate another part of the image.

The [Radial Gradient Mask](#) page shows an example. The program loads one of the resource bitmaps. The `CENTER` and `RADIUS` fields were determined from an examination of the bitmap and reference an area that should be highlighted. The `PaintSurface` handler begins by calculating a rectangle to display the bitmap and then displays it in that rectangle:

```

public class RadialGradientMaskPage : ContentPage
{
    SKBitmap bitmap = BitmapExtensions.LoadBitmapResource(
        typeof(RadialGradientMaskPage),
        "SkiaSharpFormsDemos.Media.MountainClimbers.jpg");

    static readonly SKPoint CENTER = new SKPoint(180, 300);
    static readonly float RADIUS = 120;

    public RadialGradientMaskPage ()
    {
        Title = "Radial Gradient Mask";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Find rectangle to display bitmap
        float scale = Math.Min((float)info.Width / bitmap.Width,
                               (float)info.Height / bitmap.Height);

        SKRect rect = SKRect.Create(scale * bitmap.Width, scale * bitmap.Height);

        float x = (info.Width - rect.Width) / 2;
        float y = (info.Height - rect.Height) / 2;
        rect.Offset(x, y);

        // Display bitmap in rectangle
        canvas.DrawBitmap(bitmap, rect);

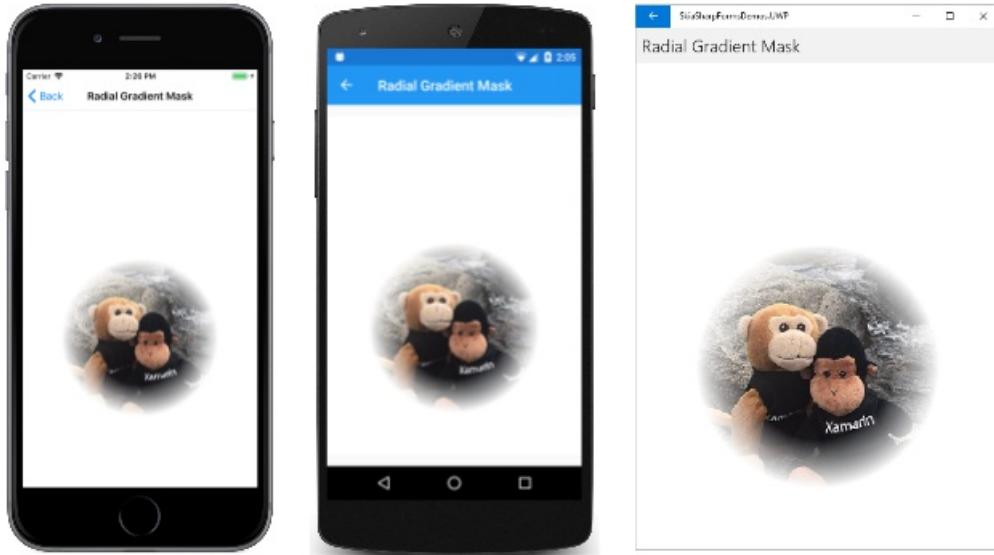
        // Adjust center and radius for scaled and offset bitmap
        SKPoint center = new SKPoint(scale * CENTER.X + x,
                                      scale * CENTER.Y + y);
        float radius = scale * RADIUS;

        using (SKPaint paint = new SKPaint())
        {
            paint.Shader = SKShader.CreateRadialGradient(
                center,
                radius,
                new SKColor[] { SKColors.Transparent,
                               SKColors.White },
                new float[] { 0.6f, 1 },
                SKShaderTileMode.Clamp);

            // Display rectangle using that gradient
            canvas.DrawRect(rect, paint);
        }
    }
}

```

After drawing the bitmap, some simple code converts `CENTER` and `RADIUS` to `center` and `radius`, which refer to the highlighted area in the bitmap that has been scaled and shifted for display. These values are used to create a radial gradient with that center and radius. The two colors begin at transparent in the center and for the first 60% of the radius. The gradient then fades to white:



This approach is not the best way to mask a bitmap. The problem is that the mask mostly has a color of white, which was chosen to match the background of the canvas. If the background is some other color — or perhaps a gradient itself — it won't match. A better approach to masking is shown in the article [SkiaSharp Porter-Duff blend modes](#).

Radial gradients for specular highlights

When a light strikes a rounded surface, it reflects light in many directions, but some of the light bounces directly into the viewer's eye. This often creates the appearance of a fuzzy white area on the surface called a *specular highlight*.

In three-dimensional graphics, specular highlights often result from the algorithms used to determine light paths and shading. In two-dimensional graphics, specular highlights are sometimes added to suggest the appearance of a 3D surface. A specular highlight can transform a flat red circle into a round red ball.

The [Radial Specular Highlight](#) page uses a radial gradient to do precisely that. The `PaintSurface` handler begins by calculating a radius for the circle, and two `SKPoint` values — a `center` and an `offCenter` that is halfway between the center and the upper-left edge of the circle:

```

public class RadialSpecularHighlightPage : ContentPage
{
    public RadialSpecularHighlightPage()
    {
        Title = "Radial Specular Highlight";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

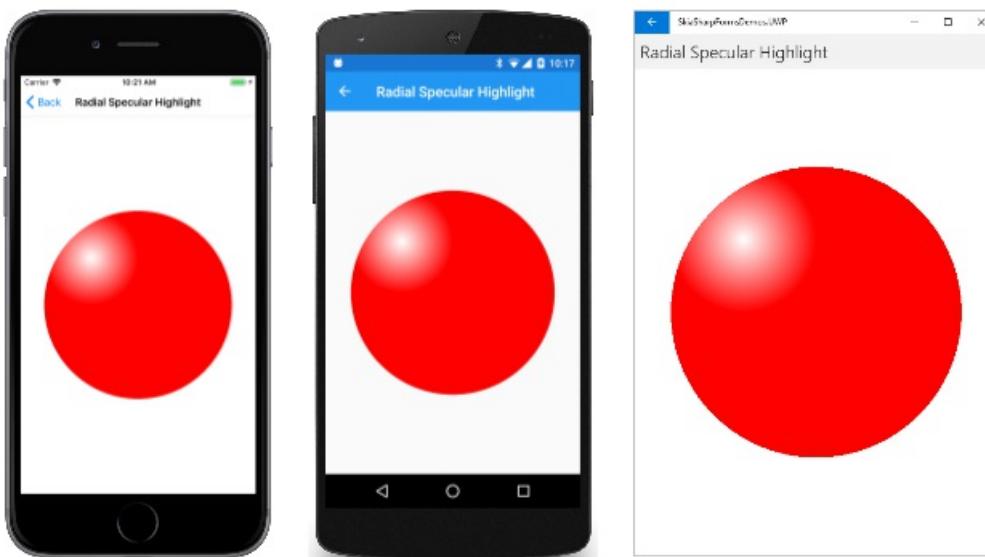
        float radius = 0.4f * Math.Min(info.Width, info.Height);
        SKPoint center = new SKPoint(info.Rect.MidX, info.Rect.MidY);
        SKPoint offCenter = center - new SKPoint(radius / 2, radius / 2);

        using (SKPaint paint = new SKPaint())
        {
            paint.Shader = SKShader.CreateRadialGradient(
                offCenter,
                radius / 2,
                new SKColor[] { SKColors.White, SKColors.Red },
                null,
                SKShaderTileMode.Clamp);

            canvas.DrawCircle(center, radius, paint);
        }
    }
}

```

The `CreateRadialGradient` call creates a gradient that begins at that `offCenter` point with white and ends with red at a distance of half the radius. Here's what it looks like:



If you look closely at this gradient, you might decide that it is flawed. The gradient is centered around a particular point, and you might wish it were a little less symmetrical to reflect the rounded surface. In that case, you might prefer the specular highlight shown below in the section [Conical gradients for specular highlights](#).

The sweep gradient

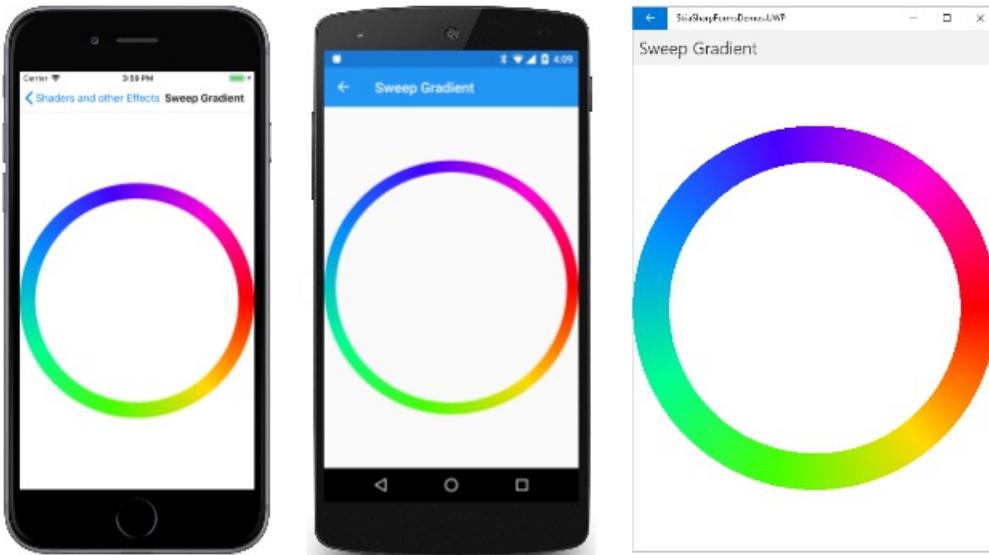
The `CreateSweepGradient` method has the simplest syntax of all the gradient-creation methods:

```
public static SKShader CreateSweepGradient (SKPoint center,
                                            SKColor[] colors,
                                            Single[] colorPos)
```

It's just a center, an array of colors, and the color positions. The gradient begins at the right of the center point and sweeps 360 degrees clockwise around the center. Notice that there's no `SKShaderTileMode` parameter.

A `CreateSweepGradient` overload with a matrix transform parameter is also available. You can apply a rotation transform to the gradient to change the starting point. You can also apply a scale transform to change the direction from clockwise to counter-clockwise.

The [Sweep Gradient](#) page uses a sweep gradient to color a circle with a stroke width of 50 pixels:



The `SweepGradientPage` class defines an array of eight colors with different hue values. Notice that the array begins and ends with red (a hue value of 0 or 360), which appears at the far right in the screenshots:

```

public class SweepGradientPage : ContentPage
{
    bool drawBackground;

    public SweepGradientPage ()
    {
        Title = "Sweep Gradient";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;

        TapGestureRecognizer tap = new TapGestureRecognizer();
        tap.Tapped += (sender, args) =>
        {
            drawBackground ^= true;
            canvasView.InvalidateSurface();
        };
        canvasView.GestureRecognizers.Add(tap);
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        using (SKPaint paint = new SKPaint())
        {
            // Define an array of rainbow colors
            SKColor[] colors = new SKColor[8];

            for (int i = 0; i < colors.Length; i++)
            {
                colors[i] = SKColor.FromHsl(i * 360f / 7, 100, 50);
            }

            SKPoint center = new SKPoint(info.Rect.MidX, info.Rect.MidY);

            // Create sweep gradient based on center of canvas
            paint.Shader = SKShader.CreateSweepGradient(center, colors, null);

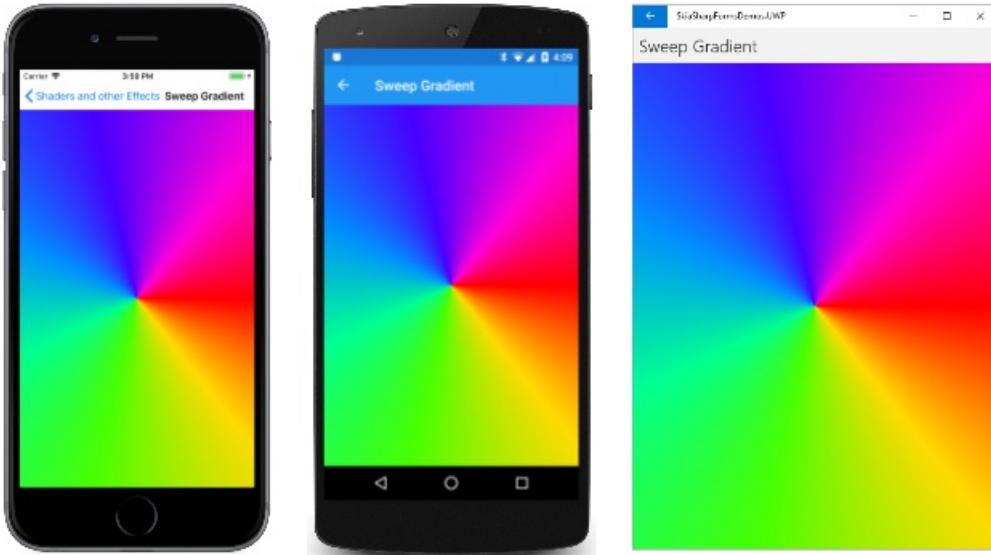
            // Draw a circle with a wide line
            const int strokeWidth = 50;
            paint.Style = SKPaintStyle.Stroke;
            paint.StrokeWidth = strokeWidth;

            float radius = (Math.Min(info.Width, info.Height) - strokeWidth) / 2;
            canvas.DrawCircle(center, radius, paint);

            if (drawBackground)
            {
                // Draw the gradient on the whole canvas
                paint.Style = SKPaintStyle.Fill;
                canvas.DrawRect(info.Rect, paint);
            }
        }
    }
}

```

The program also implements a `TapGestureRecognizer` that enables some code at the end of the `PaintSurface` handler. This code uses the same gradient to fill the canvas:



These screenshots demonstrate that the gradient fills whatever area is colored by it. If the gradient does not begin and end with the same color, there will be a discontinuity to the right of the center point.

The two-point conical gradient

The `CreateTwoPointConicalGradient` method has the following syntax:

```
public static SKShader CreateTwoPointConicalGradient (SKPoint startCenter,  
                                                    Single startRadius,  
                                                    SKPoint endCenter,  
                                                    Single endRadius,  
                                                    SKColor[] colors,  
                                                    Single[] colorPos,  
                                                    SKShaderTileMode mode)
```

The parameters begin with center points and radii for two circles, referred to as the *start* circle and *end* circle.

The remaining three parameters are the same as for `CreateLinearGradient` and `CreateRadialGradient`. A

`CreateTwoPointConicalGradient` overload includes a matrix transform.

The gradient begins at the start circle and ends at the end circle. The `SKShaderTileMode` parameter governs what happens beyond the two circles. The two-point conical gradient is the only gradient that doesn't entirely fill an area. If the two circles have the same radius, the gradient is restricted to a rectangle with a width that is the same as the diameter of the circles. If the two circles have different radii, the gradient forms a cone.

It's likely you'll want to experiment with the two-point conical gradient, so the **Conical Gradient** page derives from `InteractivePage` to allow two touch points to be moved around for the two circle radii:

```

<local:InteractivePage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:SkiaSharpFormsDemos"
    xmlns:skia="clr-namespace:SkiaSharp;assembly=SkiaSharp"
    xmlns:skiaforms="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    xmlns:tt="clr-namespace:TouchTracking"
    x:Class="SkiaSharpFormsDemos.Effects.ConicalGradientPage"
    Title="Conical Gradient">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Grid BackgroundColor="White"
            Grid.Row="0">
            <skiaforms:SKCanvasView x:Name="canvasView"
                PaintSurface="OnCanvasViewPaintSurface" />
            <Grid.Effects>
                <tt:TouchEffect Capture="True"
                    TouchAction="OnTouchEffectAction" />
            </Grid.Effects>
        </Grid>

        <Picker x:Name="tileModePicker"
            Grid.Row="1"
            Title="Shader Tile Mode"
            Margin="10"
            SelectedIndexChanged="OnPickerSelectedIndexChanged">
            <Picker.ItemsSource>
                <x:Array Type="{x:Type skia:SKShaderTileMode}">
                    <x:Static Member="skia:SKShaderTileMode.Clamp" />
                    <x:Static Member="skia:SKShaderTileMode.Repeat" />
                    <x:Static Member="skia:SKShaderTileMode.Mirror" />
                </x:Array>
            </Picker.ItemsSource>

            <Picker.SelectedIndex>
                0
            </Picker.SelectedIndex>
        </Picker>
    </Grid>
</local:InteractivePage>

```

The code-behind file defines the two `TouchPoint` objects with fixed radii of 50 and 100:

```

public partial class ConicalGradientPage : InteractivePage
{
    const int RADIUS1 = 50;
    const int RADIUS2 = 100;

    public ConicalGradientPage ()
    {
        touchPoints = new TouchPoint[2];

        touchPoints[0] = new TouchPoint
        {
            Center = new SKPoint(100, 100),
            Radius = RADIUS1
        };

        touchPoints[1] = new TouchPoint
        {
            Center = new SKPoint(300, 300),
            Radius = RADIUS2
        };
    }
}

```

```

        InitializeComponent();
        baseCanvasView = canvasView;
    }

    void OnPickerSelectedIndexChanged(object sender, EventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        SKColor[] colors = { SKColors.Red, SKColors.Green, SKColors.Blue };
        SKShaderTileMode tileMode =
            (SKShaderTileMode)(tileModePicker.SelectedIndex == -1 ?
                0 : tileModePicker.SelectedItem);

        using (SKPaint paint = new SKPaint())
        {
            paint.Shader = SKShader.CreateTwoPointConicalGradient(touchPoints[0].Center,
                RADIUS1,
                touchPoints[1].Center,
                RADIUS2,
                colors,
                null,
                tileMode);
            canvas.DrawRect(info.Rect, paint);
        }

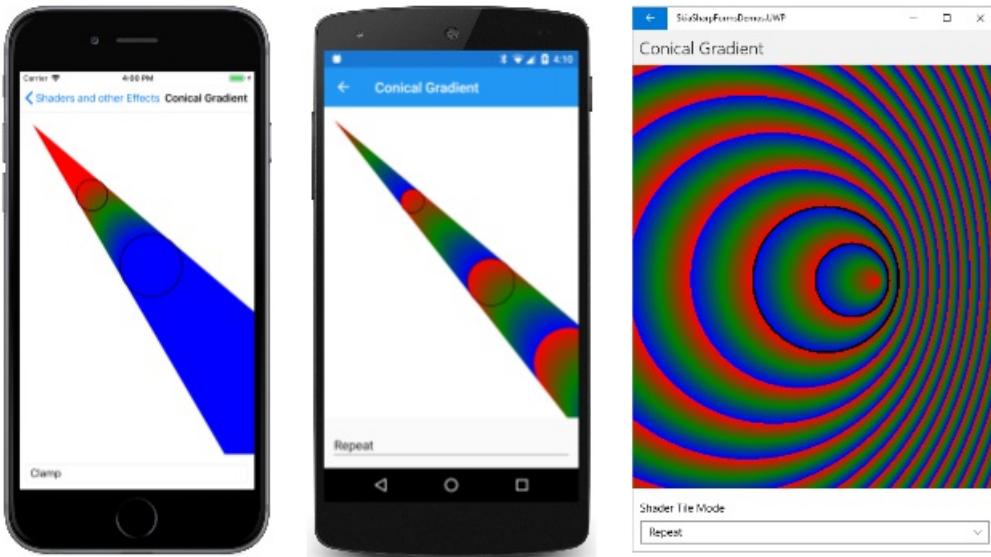
        // Display the touch points here rather than by TouchPoint
        using (SKPaint paint = new SKPaint())
        {
            paint.Style = SKPaintStyle.Stroke;
            paint.Color = SKColors.Black;
            paint.StrokeWidth = 3;

            foreach (TouchPoint touchPoint in touchPoints)
            {
                canvas.DrawCircle(touchPoint.Center, touchPoint.Radius, paint);
            }
        }
    }
}

```

The `colors` array is red, green, and blue. The code towards the bottom of the `PaintSurface` handler draws the two touch points as black circles so that they don't obstruct the gradient.

Notice that `DrawRect` call uses the gradient to color the entire canvas. In the general case, however, much of the canvas remains uncolored by the gradient. Here's the program showing three possible configurations:



The iOS screen at the left shows the effect of the `SKShaderTileMode` setting of `Clamp`. The gradient begins with red inside the edge of the smaller circle that is opposite the side closest to the second circle. The `Clamp` value also causes red to continue to the point of the cone. The gradient ends with blue at the outer edge of the larger circle that is closest to the first circle, but continues with blue within that circle and beyond.

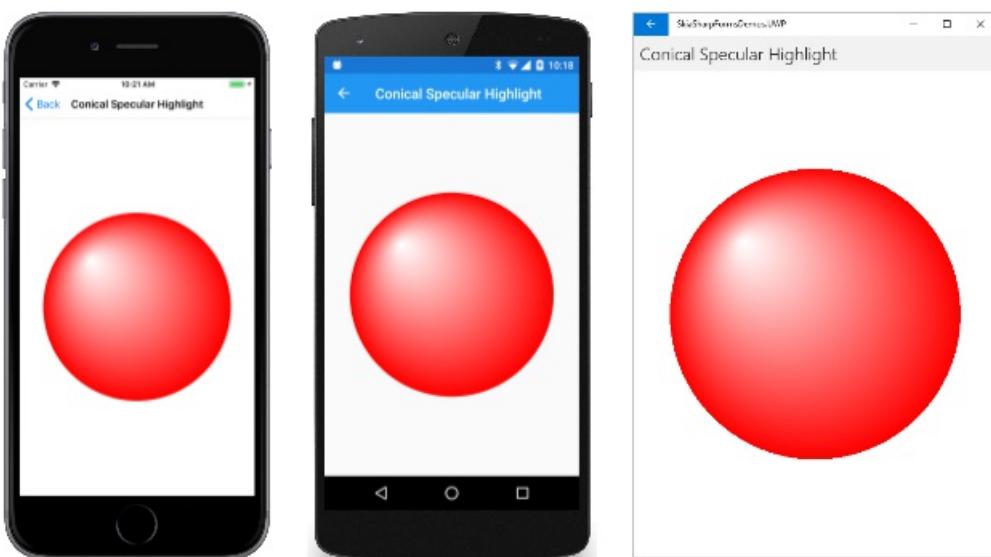
The Android screen is similar but with an `SKShaderTileMode` of `Repeat`. Now it's clearer that the gradient begins inside the first circle and ends outside the second circle. The `Repeat` setting causes the gradient to repeat again with red inside the larger circle.

The UWP screen shows what happens when the smaller circle is moved entirely inside the larger circle. The gradient stops being a cone and instead fills the whole area. The effect is similar to the radial gradient, but it's asymmetrical if the smaller circle is not exactly centered within the larger circle.

You might doubt the practical usefulness of the gradient when one circle is nested in another, but it's ideal for a specular highlight.

Conical gradients for specular highlights

Earlier in this article you saw how to use a radial gradient to create a specular highlight. You can also use the two-point conical gradient for this purpose, and you might prefer how it looks:



The asymmetrical appearance better suggests the rounded surface of the object.

The drawing code in the [Conical Specular Highlight](#) page is the same as the [Radial Specular Highlight](#)

page except for the shader:

```
public class ConicalSpecularHighlightPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        ...
        using (SKPaint paint = new SKPaint())
        {
            paint.Shader = SKShader.CreateTwoPointConicalGradient(
                offCenter,
                1,
                center,
                radius,
                new SKColor[] { SKColors.White, SKColors.Red },
                null,
                SKShaderTileMode.Clamp);

            canvas.DrawCircle(center, radius, paint);
        }
    }
}
```

The two circles have centers of `offCenter` and `center`. The circle centered at `center` is associated with a radius that encompasses the entire ball, but the circle centered at `offCenter` has a radius of just one pixel. The gradient effectively begins at that point and ends at the edge of the ball.

Related links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

SkiaSharp bitmap tiling

3/5/2021 • 15 minutes to read • [Edit Online](#)

 [Download the sample](#)

 [Download the sample](#)

As you've seen in the two previous articles, the `SKShader` class can create linear or circular gradients. This article focuses on the `SKShader` object that uses a bitmap to tile an area. The bitmap can be repeated horizontally and vertically, either in its original orientation or alternately flipped horizontally and vertically. The flipping avoids discontinuities between the tiles:



The static `SKShader.CreateBitmap` method that creates this shader has an `SKBitmap` parameter and two members of the `SKShaderTileMode` enumeration:

```
public static SKShader CreateBitmap (SKBitmap src, SKShaderTileMode tmx, SKShaderTileMode tmy)
```

The two parameters indicate the modes used for horizontal tiling and vertical tiling. This is the same `SKShaderTileMode` enumeration that is also used with the gradient methods.

A `CreateBitmap` overload includes an `SKMatrix` argument to perform a transform on the tiled bitmaps:

```
public static SKShader CreateBitmap (SKBitmap src, SKShaderTileMode tmx, SKShaderTileMode tmy, SKMatrix localMatrix)
```

This article contains several examples of using this matrix transform with tiled bitmaps.

Exploring the tile modes

The first program in the **Bitmap Tiling** section of the **Shaders and other Effects** page of the [SkiaSharpFormsDemos](#) sample demonstrates the effects of the two `SKShaderTileMode` arguments. The **Bitmap Tile Flip Modes XAML** file instantiates an `SKCanvasView` and two `Picker` views that allow you to select an `SKShaderTilerMode` value for horizontal and vertical tiling. Notice that an array of the `SKShaderTileMode` members is defined in the `Resources` section:

```

<?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:skia="clr-namespace:SkiaSharp;assembly=SkiaSharp"
    xmlns:skiaforms="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Effects.BitmapTileFlipModesPage"
    Title="Bitmap Tile Flip Modes">

    <ContentPage.Resources>
        <x:Array x:Key="tileModes"
            Type="{x:Type skia:SKShaderTileMode}">
            <x:Static Member="skia:SKShaderTileMode.Clamp" />
            <x:Static Member="skia:SKShaderTileMode.Repeat" />
            <x:Static Member="skia:SKShaderTileMode.Mirror" />
        </x:Array>
    </ContentPage.Resources>

    <StackLayout>
        <skiaforms:SKCanvasView x:Name="canvasView"
            VerticalOptions="FillAndExpand"
            PaintSurface="OnCanvasViewPaintSurface" />

        <Picker x:Name="xModePicker"
            Title="Tile X Mode"
            Margin="10, 0"
            ItemsSource="{StaticResource tileModes}"
            SelectedIndex="0"
            SelectedIndexChanged="OnPickerSelectedIndexChanged" />

        <Picker x:Name="yModePicker"
            Title="Tile Y Mode"
            Margin="10, 10"
            ItemsSource="{StaticResource tileModes}"
            SelectedIndex="0"
            SelectedIndexChanged="OnPickerSelectedIndexChanged" />

    </StackLayout>
</ContentPage>

```

The constructor of the code-behind file loads in the bitmap resource that shows a monkey sitting. It first crops the image using the `ExtractSubset` method of `SKBitmap` so that the head and feet are touching the edges of the bitmap. The constructor then uses the `Resize` method to create another bitmap of half the size. These changes make the bitmap a little more suitable for tiling:

```

public partial class BitmapTileFlipModesPage : ContentPage
{
    SKBitmap bitmap;

    public BitmapTileFlipModesPage ()
    {
        InitializeComponent ();

        SKBitmap origBitmap = BitmapExtensions.LoadBitmapResource(
            GetType(), "SkiaSharpFormsDemos.Media.SeatedMonkey.jpg");

        // Define cropping rect
        SKRectI cropRect = new SKRectI(5, 27, 296, 260);

        // Get the cropped bitmap
        SKBitmap croppedBitmap = new SKBitmap(cropRect.Width, cropRect.Height);
        origBitmap.ExtractSubset(croppedBitmap, cropRect);

        // Resize to half the width and height
        SKImageInfo info = new SKImageInfo(cropRect.Width / 2, cropRect.Height / 2);
        bitmap = croppedBitmap.Resize(info, SKBitmapResizeMethod.Box);
    }

    void OnPickerSelectedIndexChanged(object sender, EventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

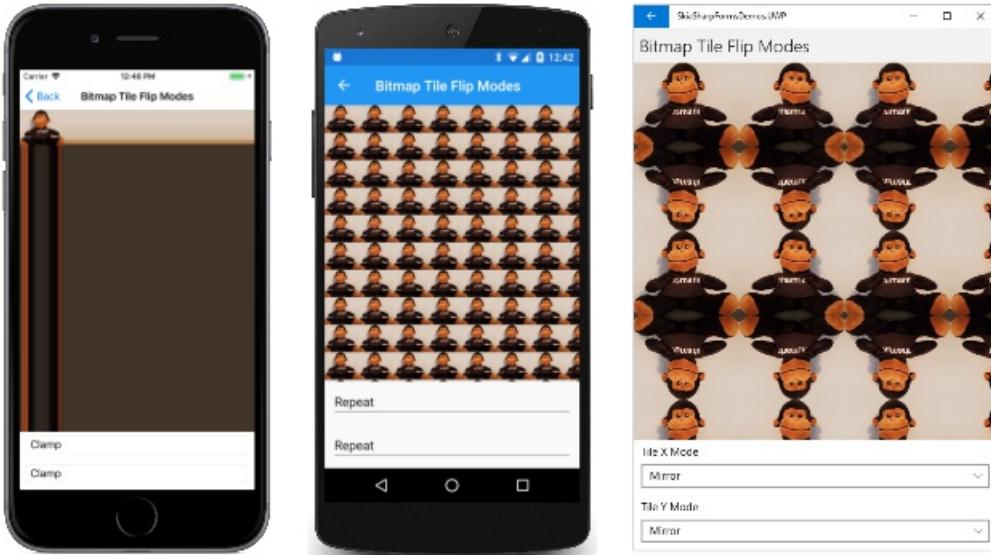
        canvas.Clear();

        // Get tile modes from Pickers
        SKShaderTileMode xTileMode =
            (SKShaderTileMode)(xModePicker.SelectedIndex == -1 ?
                0 : xModePicker.SelectedItem);
        SKShaderTileMode yTileMode =
            (SKShaderTileMode)(yModePicker.SelectedIndex == -1 ?
                0 : yModePicker.SelectedItem);

        using (SKPaint paint = new SKPaint())
        {
            paint.Shader = SKShader.CreateBitmap(bitmap, xTileMode, yTileMode);
            canvas.DrawRect(info.Rect, paint);
        }
    }
}

```

The `PaintSurface` handler obtains the `SKShaderTileMode` settings from the two `Picker` views and creates an `SKShader` object based on the bitmap and those two values. This shader is used to fill the canvas:



The iOS screen at the left shows the effect of the default values of `SKShaderTileMode.Clamp`. The bitmap sits in the upper-left corner. Below the bitmap, the bottom row of pixels is repeated all the way down. To the right of the bitmap, the rightmost column of pixels is repeated all the way across. The remainder of the canvas is colored by the dark brown pixel in the bitmap's lower-right corner. It should be obvious that the `clamp` option is almost never used with bitmap tiling!

The Android screen in the center shows the result of `skShaderTileMode.Repeat` for both arguments. The tile is repeated horizontally and vertically. The Universal Windows Platform screen shows `skShaderTileMode.Mirror`. The tiles are repeated but alternately flipped horizontally and vertically. The advantage of this option is that there are no discontinuities between the tiles.

Keep in mind that you can use different options for the horizontal and vertical repetition. You can specify `SKShaderTileMode.Mirror` as the second argument to `CreateBitmap` but `skShaderTileMode.Repeat` as the third argument. On each row, the monkeys still alternate between the normal image and the mirror image, but none of the monkeys are upside-down.

Patterned backgrounds

Bitmap tiling is commonly used to create a patterned background from a relatively small bitmap. The classic example is a brick wall.

The [Algorithmic Brick Wall](#) page creates a small bitmap that resembles a whole brick and two halves of a brick separated by mortar. Because this brick is used in the next sample as well, it's created by a static constructor and made public with a static property:

```

public class AlgorithmicBrickWallPage : ContentPage
{
    static AlgorithmicBrickWallPage()
    {
        const int brickWidth = 64;
        const int brickHeight = 24;
        const int morterThickness = 6;
        const int bitmapWidth = brickWidth + morterThickness;
        const int bitmapHeight = 2 * (brickHeight + morterThickness);

        SKBitmap bitmap = new SKBitmap(bitmapWidth, bitmapHeight);

        using (SKCanvas canvas = new SKCanvas(bitmap))
        using (SKPaint brickPaint = new SKPaint())
        {
            brickPaint.Color = new SKColor(0xB2, 0x22, 0x22);

            canvas.Clear(new SKColor(0xF0, 0xEA, 0xD6));
            canvas.DrawRect(new SKRect(morterThickness / 2,
                                      morterThickness / 2,
                                      morterThickness / 2 + brickWidth,
                                      morterThickness / 2 + brickHeight),
                           brickPaint);

            int ySecondBrick = 3 * morterThickness / 2 + brickHeight;

            canvas.DrawRect(new SKRect(0,
                                      ySecondBrick,
                                      bitmapWidth / 2 - morterThickness / 2,
                                      ySecondBrick + brickHeight),
                           brickPaint);

            canvas.DrawRect(new SKRect(bitmapWidth / 2 + morterThickness / 2,
                                      ySecondBrick,
                                      bitmapWidth,
                                      ySecondBrick + brickHeight),
                           brickPaint);
        }

        // Save as public property for other programs
        BrickWallTile = bitmap;
    }

    public static SKBitmap BrickWallTile { private set; get; }
    ...
}

```

The resultant bitmap is 70 pixels wide and 60 pixels high:



The rest of the **Algorithmic Brick Wall** page creates an `SKShader` object that repeats this image horizontally and vertically:

```

public class AlgorithmicBrickWallPage : ContentPage
{
    ...
    public AlgorithmicBrickWallPage ()
    {
        Title = "Algorithmic Brick Wall";

        // Create SKCanvasView
        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

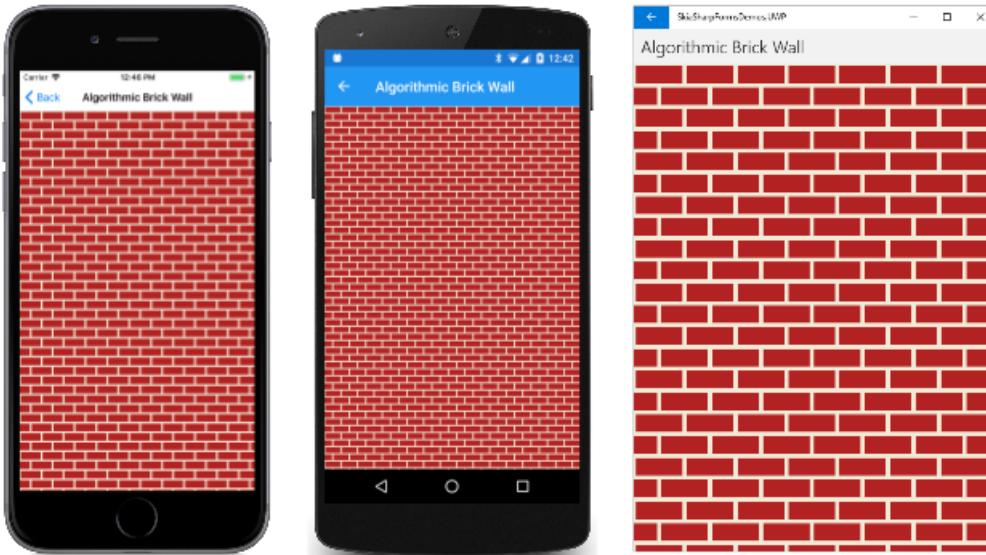
        canvas.Clear();

        using (SKPaint paint = new SKPaint())
        {
            // Create bitmap tiling
            paint.Shader = SKShader.CreateBitmap(BrickWallTile,
                SKShaderTileMode.Repeat,
                SKShaderTileMode.Repeat);

            // Draw background
            canvas.DrawRect(info.Rect, paint);
        }
    }
}

```

Here's the result:



You might prefer something a little more realistic. In that case, you can take a photograph of an actual brick wall and then crop it. This bitmap is 300 pixels wide and 150 pixels high:



This bitmap is used in the **Photographic Brick Wall** page:

```
public class PhotographicBrickWallPage : ContentPage
{
    SKBitmap bitmap = BitmapExtensions.LoadBitmapResource(
        typeof(PhotographicBrickWallPage),
        "SkiaSharpFormsDemos.Media.BrickWallTile.jpg");

    public PhotographicBrickWallPage()
    {
        Title = "Photographic Brick Wall";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

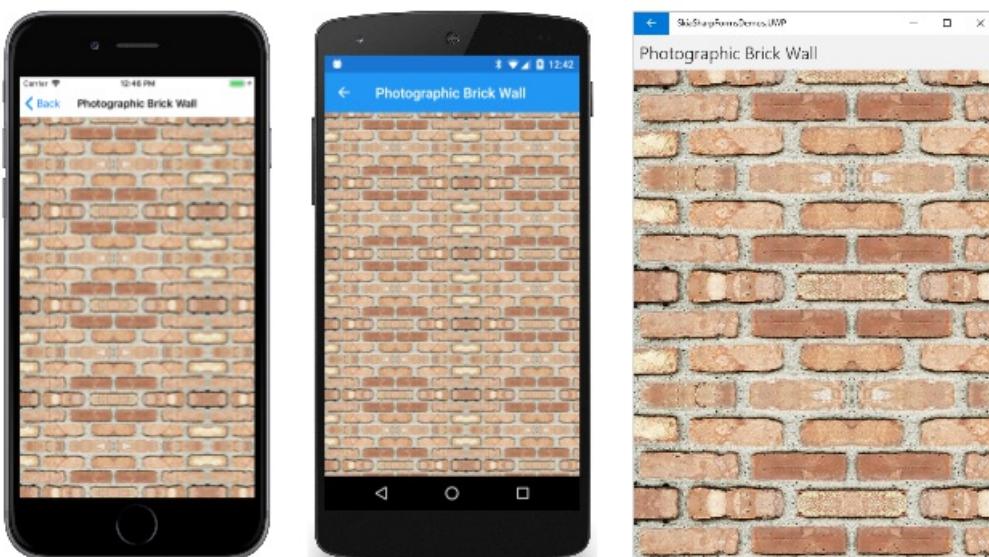
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        using (SKPaint paint = new SKPaint())
        {
            // Create bitmap tiling
            paint.Shader = SKShader.CreateBitmap(bitmap,
                SKShaderTileMode.Mirror,
                SKShaderTileMode.Mirror);

            // Draw background
            canvas.DrawRect(info.Rect, paint);
        }
    }
}
```

Notice that the `SKShaderTileMode` arguments to `CreateBitmap` are both `Mirror`. This option is usually necessary when you use tiles created from real-world images. Mirroring the tiles avoids discontinuities:



Some work is required to get a suitable bitmap for the tile. This one doesn't work very well because the darker brick stands out too much. It appears regularly within the repeated images, revealing the fact that this brick wall was constructed from a smaller bitmap.

The **Media** folder of the [SkiaSharpFormsDemos](#) sample also includes this image of a stone wall:



However, the original bitmap is a little too large for a tile. It could be resized, but the `SKShader.CreateBitmap` method can also resize the tile by applying a transform to it. This option is demonstrated in the [Stone Wall](#) page:

```
public class StoneWallPage : ContentPage
{
    SKBitmap bitmap = BitmapExtensions.LoadBitmapResource(
        typeof(StoneWallPage),
        "SkiaSharpFormsDemos.Media.StoneWallTile.jpg");

    public StoneWallPage()
    {
        Title = "Stone Wall";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

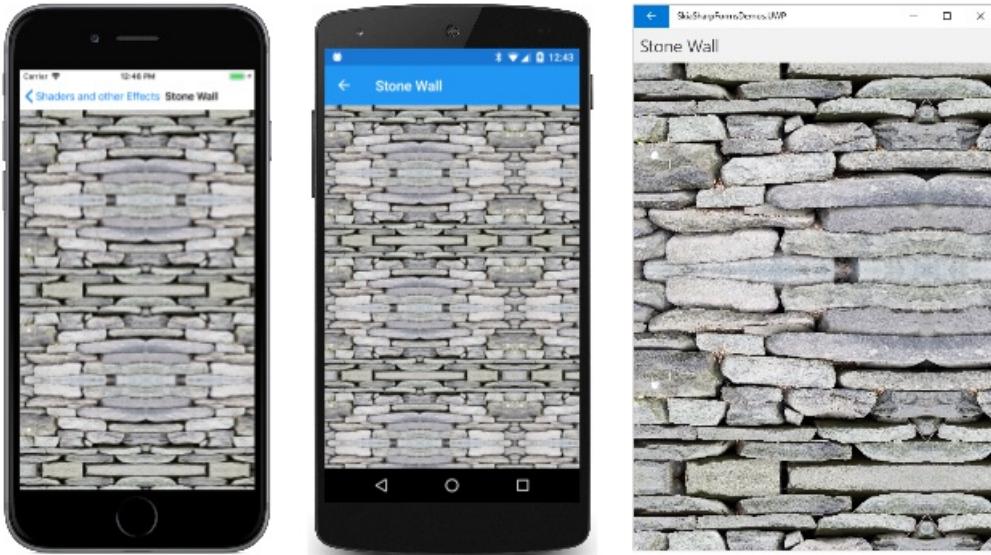
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        using (SKPaint paint = new SKPaint())
        {
            // Create scale transform
            SKMatrix matrix = SKMatrix.MakeScale(0.5f, 0.5f);

            // Create bitmap tiling
            paint.Shader = SKShader.CreateBitmap(bitmap,
                SKShaderTileMode.Mirror,
                SKShaderTileMode.Mirror,
                matrix);
            // Draw background
            canvas.DrawRect(info.Rect, paint);
        }
    }
}
```

An `SKMatrix` value is created to scale the image to half its original size:

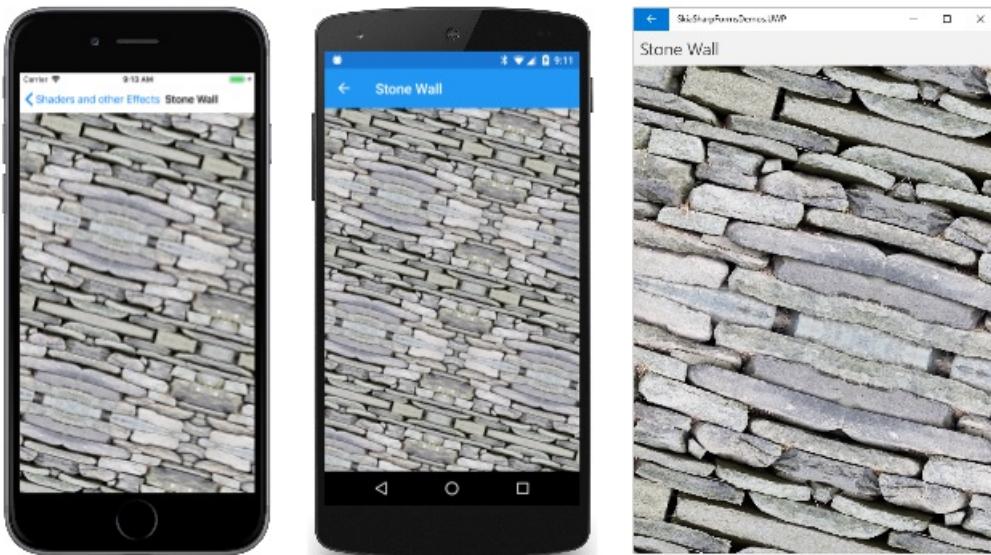


Does the transform operate on the original bitmap used in the `CreateBitmap` method? Or does it transform the resultant array of tiles?

An easy way to answer this question is to include a rotation as part of the transform:

```
SKMatrix matrix = SKMatrix.MakeScale(0.5f, 0.5f);
SKMatrix.PostConcat(ref matrix, SKMatrix.MakeRotationDegrees(15));
```

If the transform is applied to the individual tile, then each repeated image of the tile should be rotated, and the result would contain many discontinuities. But it's obvious from this screenshot that the composite array of tiles is transformed:

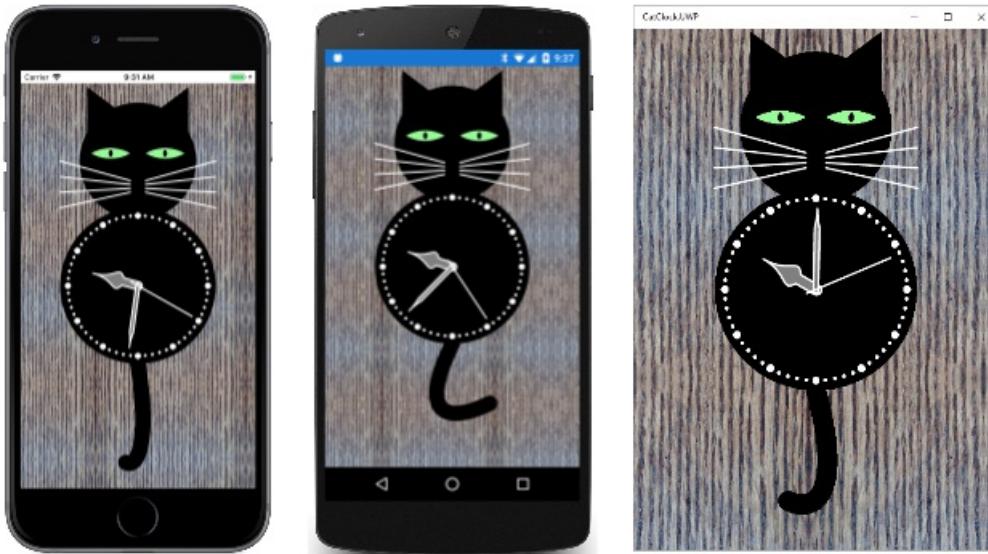


In the section [Tile alignment](#), you'll see an example a translate transform applied to the shader.

The standalone [Cat Clock](#) sample (not part of `SkiaSharpFormsDemos`) simulates a wood-grain background using bitmap tiling based on this 240-pixel square bitmap:

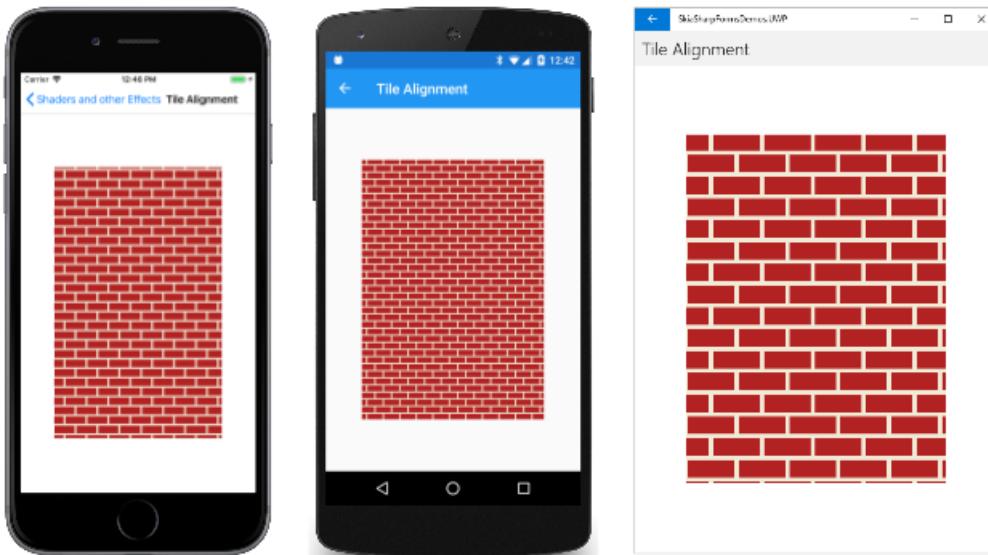


That is a photograph of a wood floor. The `skShaderTileMode.Mirror` option allows it to appear as a much larger area of wood:



Tile alignment

All the examples shown so far have used the shader created by `skShader.CreateBitmap` to cover the entire canvas. In most cases, you'll be using bitmap tiling for filling smaller areas or (more rarely) for filling the interiors of thick lines. Here's the photographic brick-wall tile used for a smaller rectangle:



This might look fine to you, or maybe not. Perhaps you're disturbed that the tiling pattern doesn't begin with a full brick in the upper-left corner of the rectangle. That's because shaders are aligned with the canvas and not the graphical object that they adorn.

The fix is simple. Create an `SKMatrix` value based on a translation transform. The transform effectively shifts the tiled pattern to the point where you want the upper-left corner of the tile to be aligned. This approach is demonstrated in the [Tile Alignment](#) page, which created the image of the unaligned tiles shown above:

```

public class TileAlignmentPage : ContentPage
{
    bool isAligned;

    public TileAlignmentPage()
    {
        Title = "Tile Alignment";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;

        // Add tap handler
        TapGestureRecognizer tap = new TapGestureRecognizer();
        tap.Tapped += (sender, args) =>
        {
            isAligned ^= true;
            canvasView.InvalidateSurface();
        };
        canvasView.GestureRecognizers.Add(tap);

        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        using (SKPaint paint = new SKPaint())
        {
            SKRect rect = new SKRect(info.Width / 7,
                                    info.Height / 7,
                                    6 * info.Width / 7,
                                    6 * info.Height / 7);

            // Get bitmap from other program
            SKBitmap bitmap = AlgorithmicBrickWallPage.BrickWallTile;

            // Create bitmap tiling
            if (!isAligned)
            {
                paint.Shader = SKShader.CreateBitmap(bitmap,
                                                    SKShaderTileMode.Repeat,
                                                    SKShaderTileMode.Repeat);
            }
            else
            {
                SKMatrix matrix = SKMatrix.MakeTranslation(rect.Left, rect.Top);

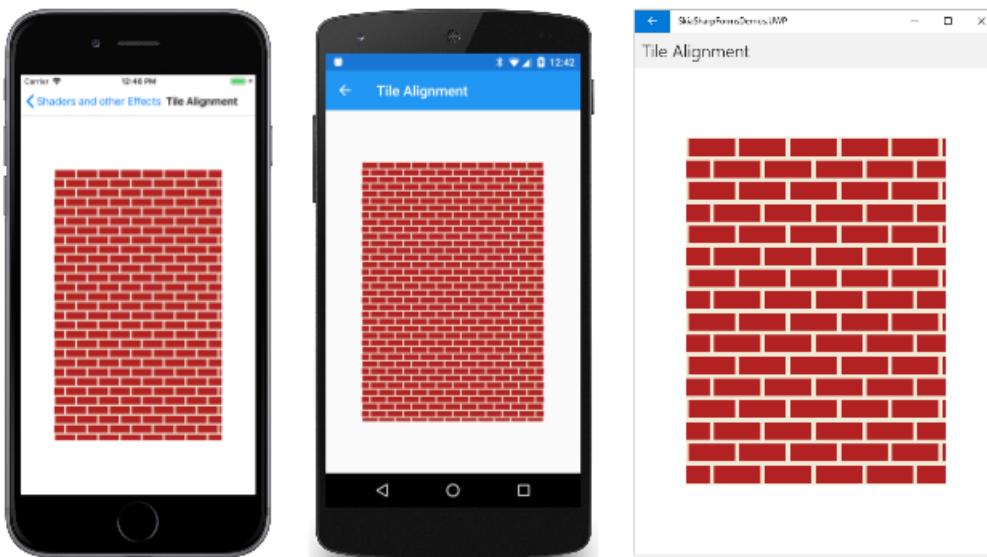
                paint.Shader = SKShader.CreateBitmap(bitmap,
                                                    SKShaderTileMode.Repeat,
                                                    SKShaderTileMode.Repeat,
                                                    matrix);
            }

            // Draw rectangle
            canvas.DrawRect(rect, paint);
        }
    }
}

```

The **Tile Alignment** page includes a `TapGestureRecognizer`. Tap or click the screen, and the program switches to the `SKShader.CreateBitmap` method with an `SKMatrix` argument. This transform shifts the pattern so that the

upper-left corner contains a full brick:



You can also use this technique to ensure that the tiled bitmap pattern is centered within the area that it paints. In the **Centered Tiles** page, the `PaintSurface` handler first calculates coordinates as if it's going to display the single bitmap in the center of the canvas. It then uses those coordinates to create a translate transform for `SKShader.CreateBitmap`. This transform shifts the entire pattern so that a tile is centered:

```

public class CenteredTilesPage : ContentPage
{
    SKBitmap bitmap = BitmapExtensions.LoadBitmapResource(
        typeof(CenteredTilesPage),
        "SkiaSharpFormsDemos.Media.monkey.png");

    public CenteredTilesPage ()
    {
        Title = "Centered Tiles";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

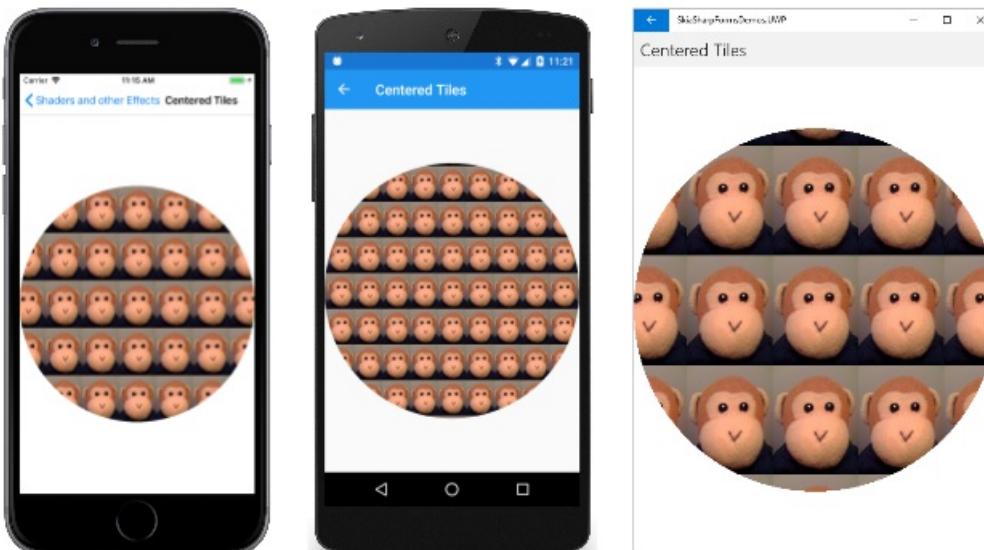
        // Find coordinates to center bitmap in canvas...
        float x = (info.Width - bitmap.Width) / 2f;
        float y = (info.Height - bitmap.Height) / 2f;

        using (SKPaint paint = new SKPaint())
        {
            // ... but use them to create a translate transform
            SKMatrix matrix = SKMatrix.MakeTranslation(x, y);
            paint.Shader = SKShader.CreateBitmap(bitmap,
                SKShaderTileMode.Repeat,
                SKShaderTileMode.Repeat,
                matrix);

            // Use that tiled bitmap pattern to fill a circle
            canvas.DrawCircle(info.Rect.MidX, info.Rect.MidY,
                Math.Min(info.Width, info.Height) / 2,
                paint);
        }
    }
}

```

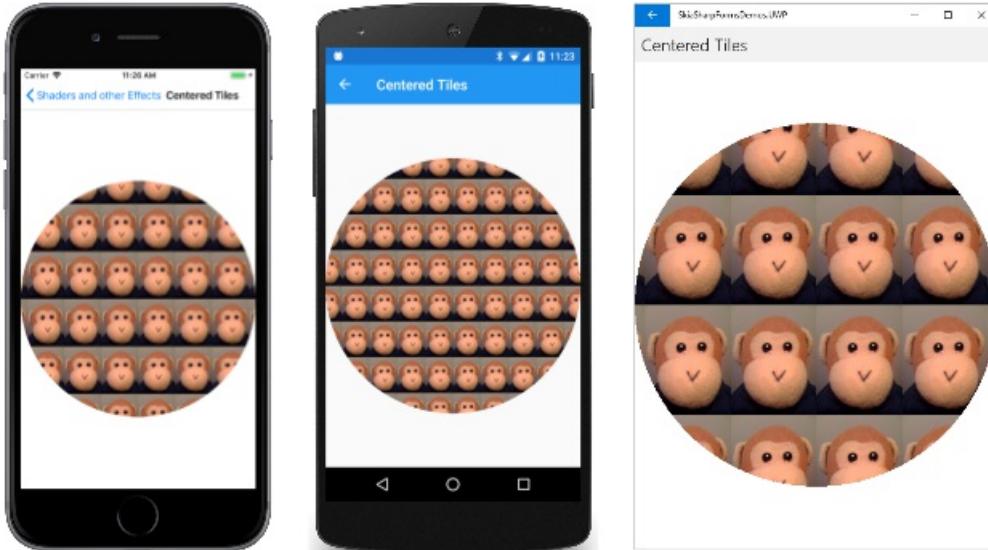
The `PaintSurface` handler concludes by drawing a circle in the center of the canvas. Sure enough, one of the tiles is exactly in the center of the circle, and the others are arranged in a symmetric pattern:



Another centering approach is actually a bit easier. Rather than construct a translate transform that puts a tile in the center, you can center a corner of the tiled pattern. In the `SKMatrix.MakeTranslation` call, use arguments for the center of the canvas:

```
SKMatrix matrix = SKMatrix.MakeTranslation(info.Rect.MidX, info.Rect.MidY);
```

The pattern is still centered and symmetrical, but no tile is in the center:



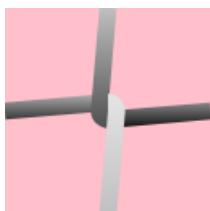
Simplification through rotation

Sometimes using a rotate transform in the `skShader.CreateBitmap` method can simplify the bitmap tile. This becomes evident when attempting to define a tile for a chain-link fence. The `ChainLinkTile.cs` file creates the tile shown here (with a pink background for purposes of clarity):



The tile needs to include two links, so that the code divides the tile into four quadrants. The upper-left and lower-right quadrants are the same, but they are not complete. The wires have little notches that must be handled with some additional drawing in the upper-right and lower-left quadrants. The file that does all this work is 174 lines long.

It turns out to be much easier to create this tile:



If the bitmap-tile shader is rotated 90 degrees, the visuals are nearly the same.

The code to create the easier chain-link tile is part of the [Chain-Link Tile](#) page. The constructor determines a tile size based on the type of device that the program is running on, and then calls `CreateChainLinkTile`, which

draws on the bitmap using lines, paths, and gradient shaders:

```
public class ChainLinkFencePage : ContentPage
{
    ...
    SKBitmap tileBitmap;

    public ChainLinkFencePage ()
    {
        Title = "Chain-Link Fence";

        // Create bitmap for chain-link tiling
        int tileSize = Device.Idiom == TargetIdiom.Desktop ? 64 : 128;
        tileBitmap = CreateChainLinkTile(tileSize);

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    SKBitmap CreateChainLinkTile(int tileSize)
    {
        tileBitmap = new SKBitmap(tileSize, tileSize);
        float wireThickness = tileSize / 12f;

        using (SKCanvas canvas = new SKCanvas(tileBitmap))
        using (SKPaint paint = new SKPaint())
        {
            canvas.Clear();
            paint.Style = SKPaintStyle.Stroke;
            paint.StrokeWidth = wireThickness;
            paint.IsAntialias = true;

            // Draw straight wires first
            paint.Shader = SKShader.CreateLinearGradient(new SKPoint(0, 0),
                new SKPoint(0, tileSize),
                new SKColor[] { SKColors.Silver, SKColors.Black },
                new float[] { 0.4f, 0.6f },
                SKShaderTileMode.Clamp);

            canvas.DrawLine(0, tileSize / 2,
                tileSize / 2, tileSize / 2 - wireThickness / 2, paint);

            canvas.DrawLine(tileSize, tileSize / 2,
                tileSize / 2, tileSize / 2 + wireThickness / 2, paint);

            // Draw curved wires
            using (SKPath path = new SKPath())
            {
                path.MoveTo(tileSize / 2, 0);
                path.LineTo(tileSize / 2 - wireThickness / 2, tileSize / 2);
                path.ArcTo(wireThickness / 2, wireThickness / 2,
                    0,
                    SKPathArcSize.Small,
                    SKPathDirection.CounterClockwise,
                    tileSize / 2, tileSize / 2 + wireThickness / 2);

                paint.Shader = SKShader.CreateLinearGradient(new SKPoint(0, 0),
                    new SKPoint(0, tileSize),
                    new SKColor[] { SKColors.Silver, SKColors.Black
                },
                    null,
                    SKShaderTileMode.Clamp);
                canvas.DrawPath(path, paint);

                path.Reset();
                path.MoveTo(tileSize / 2, tileSize);
                path.LineTo(tileSize / 2 + wireThickness / 2, tileSize / 2);
            }
        }
    }
}
```

```

        path.ArcTo(wireThickness / 2, wireThickness / 2,
                    0,
                    SKPathArcSize.Small,
                    SKPathDirection.CounterClockwise,
                    tileSize / 2, tileSize / 2 - wireThickness / 2);

        paint.Shader = SKShader.CreateLinearGradient(new SKPoint(0, 0),
                                                      new SKPoint(0, tileSize),
                                                      new SKColor[] { SKColors.White, SKColors.Silver
},
                                                      null,
                                                      SKShaderTileMode.Clamp);
        canvas.DrawPath(path, paint);
    }
    return tileBitmap;
}
}
...
}

```

Except for the wires, the tile is transparent, which means that you can display it on top of something else. The program loads in one of the bitmap resources, displays it to fill the canvas, and then draws the shader on top:

```

public class ChainLinkFencePage : ContentPage
{
    SKBitmap monkeyBitmap = BitmapExtensions.LoadBitmapResource(
        typeof(ChainLinkFencePage), "SkiaSharpFormsDemos.Media.SeatedMonkey.jpg");
    ...

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

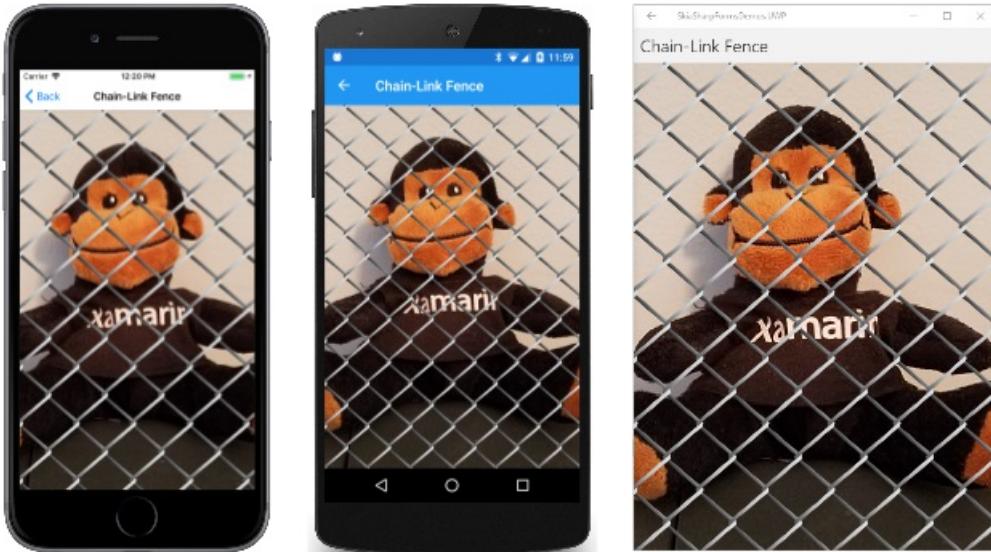
        canvas.Clear();

        canvas.DrawBitmap(monkeyBitmap, info.Rect, BitmapStretch.UniformToFill,
                         BitmapAlignment.Center, BitmapAlignment.Start);

        using (SKPaint paint = new SKPaint())
        {
            paint.Shader = SKShader.CreateBitmap(tileBitmap,
                                                SKShaderTileMode.Repeat,
                                                SKShaderTileMode.Repeat,
                                                SKMatrix.MakeRotationDegrees(45));
            canvas.DrawRect(info.Rect, paint);
        }
    }
}

```

Notice that the shader is rotated 45 degrees so it is oriented like a real chain-link fence:



Animating bitmap tiles

You can animate an entire bitmap-tile pattern by animating the matrix transform. Perhaps you want the pattern to move horizontally or vertically or both. You can do that by creating a translation transform based on the shifting coordinates.

It's also possible to draw on a small bitmap, or to manipulate the bitmap's pixel bits at the rate of 60 times a second. That bitmap can then be used for tiling, and the entire tiled pattern can seem to be animated.

The [Animated Bitmap Tile](#) page demonstrates this approach. A bitmap is instantiated as a field to be 64-pixels square. The constructor calls `DrawBitmap` to give it an initial appearance. If the `angle` field is zero (as it is when the method is first called), then the bitmap contains two lines crossed as an X. The lines are made long enough to always reach to the edge of the bitmap regardless of the `angle` value:

```

public class AnimatedBitmapTilePage : ContentPage
{
    const int SIZE = 64;

    SKCanvasView canvasView;
    SKBitmap bitmap = new SKBitmap(SIZE, SIZE);
    float angle;
    ...

    public AnimatedBitmapTilePage ()
    {
        Title = "Animated Bitmap Tile";

        // Initialize bitmap prior to animation
        DrawBitmap();

        // Create SKCanvasView
        canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }
    ...
    void DrawBitmap()
    {
        using (SKCanvas canvas = new SKCanvas(bitmap))
        using (SKPaint paint = new SKPaint())
        {
            paint.Style = SKPaintStyle.Stroke;
            paint.Color = SKColors.Blue;
            paint.StrokeWidth = SIZE / 8;

            canvas.Clear();
            canvas.Translate(SIZE / 2, SIZE / 2);
            canvas.RotateDegrees(angle);
            canvas.DrawLine(-SIZE, -SIZE, SIZE, SIZE, paint);
            canvas.DrawLine(-SIZE, SIZE, SIZE, -SIZE, paint);
        }
    }
    ...
}

```

The animation overhead occurs in the `OnAppearing` and `OnDisappearing` overrides. The `OnTimerTick` method animates the `angle` value from 0 degrees to 360 degrees every 10 seconds to rotate the X figure within the bitmap:

```

public class AnimatedBitmapTilePage : ContentPage
{
    ...
    // For animation
    bool isAnimating;
    Stopwatch stopwatch = new Stopwatch();
    ...

    protected override void OnAppearing()
    {
        base.OnAppearing();

        isAnimating = true;
        stopwatch.Start();
        Device.StartTimer(TimeSpan.FromMilliseconds(16), OnTimerTick);
    }

    protected override void OnDisappearing()
    {
        base.OnDisappearing();

        stopwatch.Stop();
        isAnimating = false;
    }

    bool OnTimerTick()
    {
        const int duration = 10;      // seconds
        angle = (float)(360f * (stopwatch.Elapsed.TotalSeconds % duration) / duration);
        DrawBitmap();
        canvasView.InvalidateSurface();

        return isAnimating;
    }
    ...
}

```

Because of the symmetry of the X figure, this is the same as rotating the `angle` value from 0 degrees to 90 degrees every 2.5 seconds.

The `PaintSurface` handler creates a shader from the bitmap and uses the `paint` object to color the entire canvas:

```

public class AnimatedBitmapTilePage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

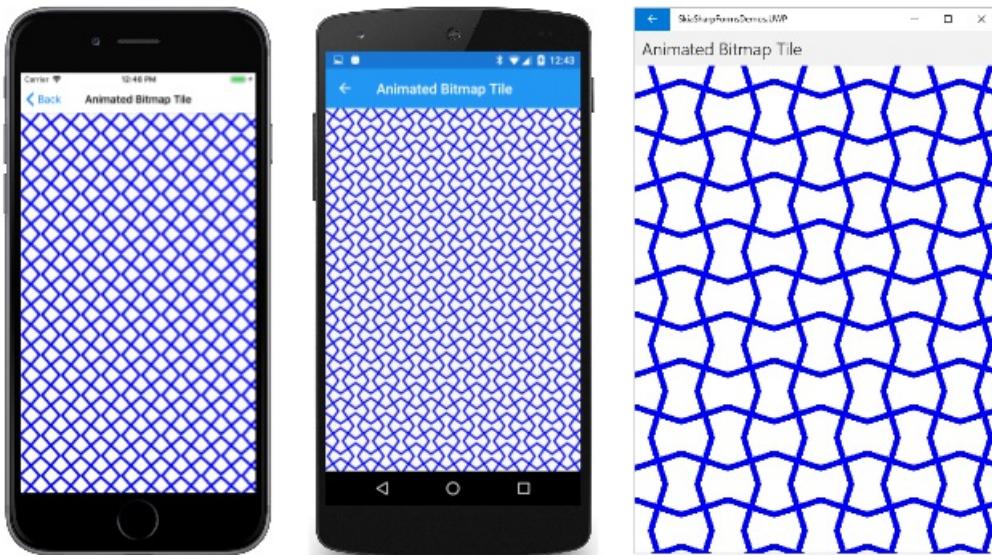
        canvas.Clear();

        using (SKPaint paint = new SKPaint())
        {
            paint.Shader = SKShader.CreateBitmap(bitmap,
                SKShaderTileMode.Mirror,
                SKShaderTileMode.Mirror);
            canvas.DrawRect(info.Rect, paint);
        }
    }
}

```

The `SKShaderTileMode.Mirror` options ensure that the arms of the X in each bitmap join with the X in the

adjacent bitmaps to create an overall animated pattern that seems much more complex than the simple animation would suggest:



Related links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)
- [CatClock \(sample\)](#)

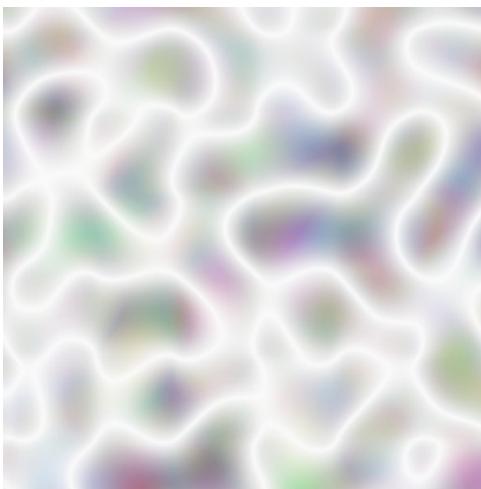
SkiaSharp noise and composing

3/5/2021 • 8 minutes to read • [Edit Online](#)



[Download the sample](#)

Simple vector graphics tend to look unnatural. The straight lines, smooth curves, and solid colors don't resemble the imperfections of real-world objects. While working on the computer-generated graphics for the 1982 movie *Tron*, computer scientist Ken Perlin began developing algorithms that used random processes to give these images more realistic textures. In 1997, Ken Perlin won an Academy Award for Technical Achievement. His work has come to be known as Perlin noise, and it is supported in SkiaSharp. Here's an example:



As you can see, each pixel is not a random color value. The continuity from pixel to pixel results in random shapes.

The support of Perlin noise in Skia is based on a W3C specification for CSS and SVG. Section 8.20 of [Filter Effects Module Level 1](#) includes the underlying Perlin noise algorithms in C code.

Exploring Perlin noise

The `SKShader` class defines two different static methods for generating Perlin noise:

`CreatePerlinNoiseFractalNoise` and `CreatePerlinNoiseTurbulence`. The parameters are identical:

```
public static SkiaSharp CreatePerlinNoiseFractalNoise (float baseFrequencyX, float baseFrequencyY, int numOctaves, float seed);  
  
public static SkiaSharp.SKShader CreatePerlinNoiseTurbulence (float baseFrequencyX, float baseFrequencyY, int numOctaves, float seed);
```

Both methods also exist in overloaded versions with an additional `SKPointI` parameter. The section [Tiling Perlin noise](#) discusses these overloads.

The two `baseFrequency` arguments are positive values defined in the SkiaSharp documentation as ranging from 0 to 1, but they can be set to higher values as well. The higher the value, the greater the change in the random image in the horizontal and vertical directions.

The `numOctaves` value is an integer of 1 or higher. It relates to an iteration factor in the algorithms. Each additional octave contributes an effect that is half of the previous octave, so the effect decreases with higher octave values.

The `seed` parameter is the starting point for the random-number generator. Although specified as a floating-point value, the fraction is truncated before it's used, and 0 is the same as 1.

The **Perlin Noise** page in the [SkiaSharpFormsDemos](#) sample allows you experiment with various values of the `baseFrequency` and `numOctaves` arguments. Here's the XAML file:

```
<?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:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Effects.PerlinNoisePage"
    Title="Perlin Noise">

    <StackLayout>
        <skia:SKCanvasView x:Name="canvasView"
            VerticalOptions="FillAndExpand"
            PaintSurface="OnCanvasViewPaintSurface" />

        <Slider x:Name="baseFrequencyXSlider"
            Maximum="4"
            Margin="10, 0"
            ValueChanged="OnSliderValueChanged" />

        <Label x:Name="baseFrequencyXText"
            HorizontalTextAlignment="Center" />

        <Slider x:Name="baseFrequencyYSlider"
            Maximum="4"
            Margin="10, 0"
            ValueChanged="OnSliderValueChanged" />

        <Label x:Name="baseFrequencyYText"
            HorizontalTextAlignment="Center" />

        <StackLayout Orientation="Horizontal"
            HorizontalOptions="Center"
            Margin="10">

            <Label Text="{Binding Source={x:Reference octavesStepper},
                Path=Value,
                StringFormat='Number of Octaves: {0:F0}'}"
                VerticalOptions="Center" />

            <Stepper x:Name="octavesStepper"
                Minimum="1"
                ValueChanged="OnStepperValueChanged" />
        </StackLayout>
    </StackLayout>
</ContentPage>
```

It uses two `Slider` views for the two `baseFrequency` arguments. To expand the range of the lower values, the sliders are logarithmic. The code-behind file calculates the arguments to the `SKShader` methods from powers of the `Slider` values. The `Label` views display the calculated values:

```
float baseFreqX = (float)Math.Pow(10, baseFrequencyXSlider.Value - 4);
baseFrequencyXText.Text = String.Format("Base Frequency X = {0:F4}", baseFreqX);

float baseFreqY = (float)Math.Pow(10, baseFrequencyYSlider.Value - 4);
baseFrequencyYText.Text = String.Format("Base Frequency Y = {0:F4}", baseFreqY);
```

A `Slider` value of 1 corresponds to 0.001, a `Slider` value of 2 corresponds to 0.01, a `Slider` value of 3 corresponds to 0.1, and a `Slider` value of 4 corresponds to 1.

Here's the code-behind file that includes that code:

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

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnStepperValueChanged(object sender, ValueChangedEventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Get values from sliders and stepper
        float baseFreqX = (float)Math.Pow(10, baseFrequencyXSlider.Value - 4);
        baseFrequencyXText.Text = String.Format("Base Frequency X = {0:F4}", baseFreqX);

        float baseFreqY = (float)Math.Pow(10, baseFrequencyYSlider.Value - 4);
        baseFrequencyYText.Text = String.Format("Base Frequency Y = {0:F4}", baseFreqY);

        int numOctaves = (int)octavesStepper.Value;

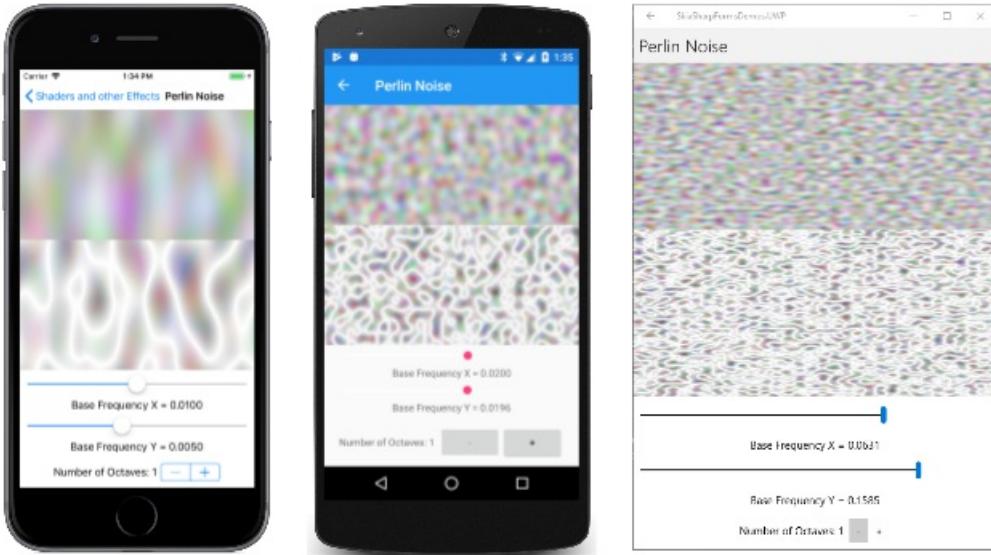
        using (SKPaint paint = new SKPaint())
        {
            paint.Shader =
                SKShader.CreatePerlinNoiseFractalNoise(baseFreqX,
                                                       baseFreqY,
                                                       numOctaves,
                                                       0);

            SKRect rect = new SKRect(0, 0, info.Width, info.Height / 2);
            canvas.DrawRect(rect, paint);

            paint.Shader =
                SKShader.CreatePerlinNoiseTurbulence(baseFreqX,
                                                     baseFreqY,
                                                     numOctaves,
                                                     0);

            rect = new SKRect(0, info.Height / 2, info.Width, info.Height);
            canvas.DrawRect(rect, paint);
        }
    }
}
```

Here's the program running on iOS, Android, and Universal Windows Platform (UWP) devices. The fractal noise is shown in the upper half of the canvas. The turbulence noise is in the bottom half:



The same arguments always produce the same pattern that begins at the upper-left corner. This consistency is obvious when you adjust the width and height of the UWP window. As Windows 10 redraws the screen, the pattern in the upper half of the canvas remains the same.

The noise pattern incorporates various degrees of transparency. The transparency becomes obvious if you set a color in the `canvas.Clear()` call. That color becomes prominent in the pattern. You'll also see this effect in the section [Combining multiple shaders](#).

These Perlin noise patterns are rarely used by themselves. Often they are subjected to blend modes and color filters discussed in later articles.

Tiling Perlin noise

The two static `SKShader` methods for creating Perlin noise also exist in overload versions. The

`CreatePerlinNoiseFractalNoise` and `CreatePerlinNoiseTurbulence` overloads have an additional `SKPointI` parameter:

```
public static SKShader CreatePerlinNoiseFractalNoise (float baseFrequencyX, float baseFrequencyY, int numOctaves, float seed, SKPointI tileSize);

public static SKShader CreatePerlinNoiseTurbulence (float baseFrequencyX, float baseFrequencyY, int numOctaves, float seed, SKPointI tileSize);
```

The `SKPointI` structure is the integer version of the familiar `SKPoint` structure. `SKPointI` defines `X` and `Y` properties of type `int` rather than `float`.

These methods create a repeating pattern of the specified size. In each tile, the right edge is the same as the left edge, and the top edge is the same as the bottom edge. This characteristic is demonstrated in the [Tiled Perlin Noise](#) page. The XAML file is similar to the previous sample, but it only has a `Stepper` view for changing the `seed` argument:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Effects.TiledPerlinNoisePage"
    Title="Tiled Perlin Noise">

    <StackLayout>
        <skia:SKCanvasView x:Name="canvasView"
            VerticalOptions="FillAndExpand"
            PaintSurface="OnCanvasViewPaintSurface" />

        <StackLayout Orientation="Horizontal"
            HorizontalOptions="Center"
            Margin="10">

            <Label Text="{Binding Source={x:Reference seedStepper},
                Path=Value,
                StringFormat='Seed: {0:F0}'}"
                VerticalOptions="Center" />

            <Stepper x:Name="seedStepper"
                Minimum="1"
                ValueChanged="OnStepperValueChanged" />

        </StackLayout>
    </StackLayout>
</ContentPage>

```

The code-behind file defines a constant for the tile size. The `PaintSurface` handler creates a bitmap of that size and an `SKCanvas` for drawing into that bitmap. The `SKShader.CreatePerlinNoiseTurbulence` method creates a shader with that tile size. This shader is drawn on the bitmap:

```

public partial class TiledPerlinNoisePage : ContentPage
{
    const int TILE_SIZE = 200;

    public TiledPerlinNoisePage()
    {
        InitializeComponent();
    }

    void OnStepperValueChanged(object sender, ValueChangedEventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Get seed value from stepper
        float seed = (float)seedStepper.Value;

        SKRect tileRect = new SKRect(0, 0, TILE_SIZE, TILE_SIZE);

        using (SKBitmap bitmap = new SKBitmap(TILE_SIZE, TILE_SIZE))
        {
            using (SKCanvas bitmapCanvas = new SKCanvas(bitmap))
            {
                bitmapCanvas.Clear();

                // Draw tiled turbulence noise on bitmap
                using (SKPaint paint = new SKPaint())
                {
                    paint.Shader = SKShader.CreatePerlinNoiseTurbulence(
                        0.02f, 0.02f, 1, seed,
                        new SKPointI(TILE_SIZE, TILE_SIZE));

                    bitmapCanvas.DrawRect(tileRect, paint);
                }
            }

            // Draw tiled bitmap shader on canvas
            using (SKPaint paint = new SKPaint())
            {
                paint.Shader = SKShader.CreateBitmap(bitmap,
                    SKShaderTileMode.Repeat,
                    SKShaderTileMode.Repeat);
                canvas.DrawRect(info.Rect, paint);
            }

            // Draw rectangle showing tile
            using (SKPaint paint = new SKPaint())
            {
                paint.Style = SKPaintStyle.Stroke;
                paint.Color = SKColors.Black;
                paint.StrokeWidth = 2;

                canvas.DrawRect(tileRect, paint);
            }
        }
    }
}

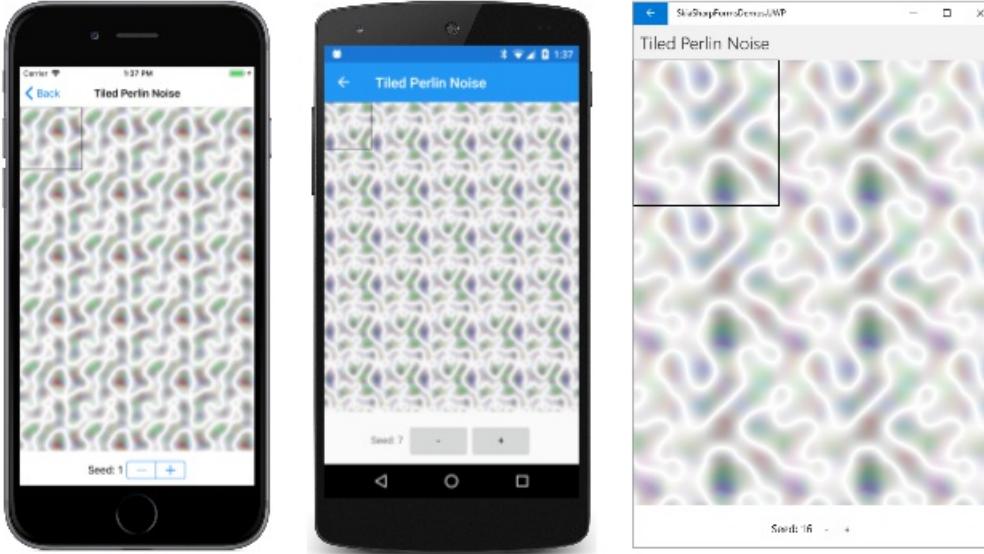
```

After the bitmap has been created, another `SKPaint` object is used to create a tiled bitmap pattern by calling `SKShader.CreateBitmap`. Notice the two arguments of `skShaderTileMode.Repeat`:

```
paint.Shader = SKShader.CreateBitmap(bitmap,
                                      SKShaderTileMode.Repeat,
                                      SKShaderTileMode.Repeat);
```

This shader is used to cover the canvas. Finally, another `SKPaint` object is used to stroke a rectangle showing the size of the original bitmap.

Only the `seed` parameter is selectable from the user interface. If the same `seed` pattern is used on each platform, they would show the same pattern. Different `seed` values result in different patterns:



The 200-pixel square pattern in the upper-left corner flows seamlessly into the other tiles.

Combining multiple shaders

The `SKShader` class includes a `CreateColor` method that creates a shader with a specified solid color. This shader is not very useful by itself because you can simply set that color to the `color` property of the `SKPaint` object and set the `Shader` property to null.

This `CreateColor` method becomes useful in another method that `SKShader` defines. This method is `CreateCompose`, which combines two shaders. Here's the syntax:

```
public static SKShader CreateCompose (SKShader dstShader, SKShader srcShader);
```

The `srcShader` (source shader) is effectively drawn on top of the `dstShader` (destination shader). If the source shader is a solid color or a gradient without transparency, the destination shader will be completely obscured.

A Perlin noise shader contains transparency. If that shader is the source, the destination shader will show through the transparent areas.

The **Composed Perlin Noise** page has a XAML file that is virtually identical to the first **Perlin Noise** page. The code-behind file is also similar. But the original **Perlin Noise** page sets the `shader` property of `SKPaint` to the shader returned from the static `CreatePerlinNoiseFractalNoise` and `CreatePerlinNoiseTurbulence` methods. This **Composed Perlin Noise** page calls `CreateCompose` for a combination shader. The destination is a solid blue shader created using `CreateColor`. The source is a Perlin noise shader:

```

public partial class ComposedPerlinNoisePage : ContentPage
{
    public ComposedPerlinNoisePage()
    {
        InitializeComponent();
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnStepperValueChanged(object sender, ValueChangedEventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Get values from sliders and stepper
        float baseFreqX = (float)Math.Pow(10, baseFrequencyXSlider.Value - 4);
        baseFrequencyXText.Text = String.Format("Base Frequency X = {0:F4}", baseFreqX);

        float baseFreqY = (float)Math.Pow(10, baseFrequencyYSlider.Value - 4);
        baseFrequencyYText.Text = String.Format("Base Frequency Y = {0:F4}", baseFreqY);

        int numOctaves = (int)octavesStepper.Value;

        using (SKPaint paint = new SKPaint())
        {
            paint.Shader = SKShader.CreateCompose(
                SKShader.CreateColor(SKColors.Blue),
                SKShader.CreatePerlinNoiseFractalNoise(baseFreqX,
                                                       baseFreqY,
                                                       numOctaves,
                                                       0));

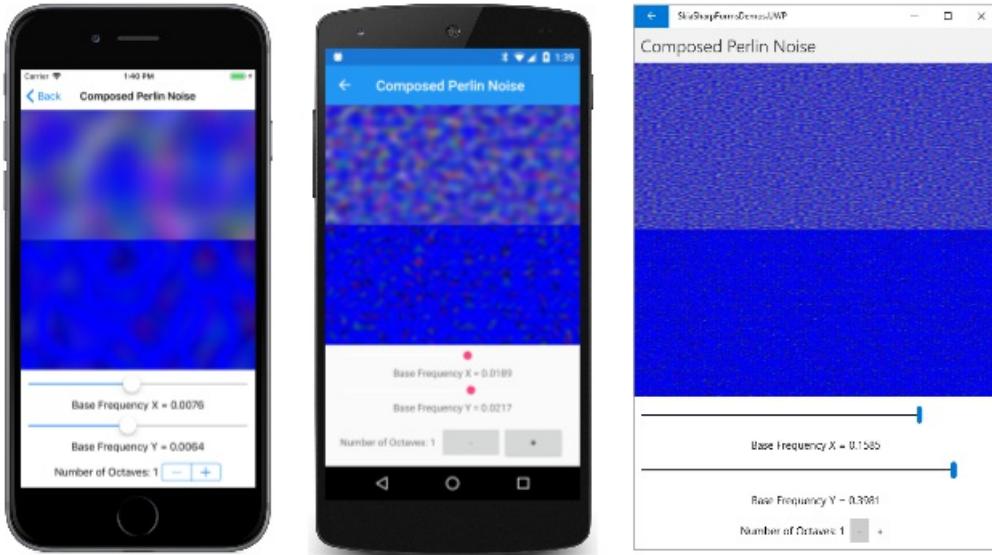
            SKRect rect = new SKRect(0, 0, info.Width, info.Height / 2);
            canvas.DrawRect(rect, paint);

            paint.Shader = SKShader.CreateCompose(
                SKShader.CreateColor(SKColors.Blue),
                SKShader.CreatePerlinNoiseTurbulence(baseFreqX,
                                                      baseFreqY,
                                                      numOctaves,
                                                      0));

            rect = new SKRect(0, info.Height / 2, info.Width, info.Height);
            canvas.DrawRect(rect, paint);
        }
    }
}

```

The fractal noise shader is on the top; the turbulence shader is on the bottom:



Notice how much bluer these shaders are than the ones displayed by the [Perlin Noise](#) page. The difference illustrates the amount of transparency in the noise shaders.

There's also an overload of the [CreateCompose](#) method:

```
public static SKShader CreateCompose (SKShader dstShader, SKShader srcShader, SKBlendMode blendMode);
```

The final parameter is a member of the [SKBlendMode](#) enumeration, an enumeration with 29 members that is discussed in the next series of articles on [SkiaSharp compositing and blend modes](#).

Related links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

SkiaSharp blend modes

3/5/2021 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

These articles focus on the `BlendMode` property of `SKPaint`. The `BlendMode` property is of type `SKBlendMode`, an enumeration with 29 members.

The `BlendMode` property determines what happens when a graphical object (often called the *source*) is rendered on top of existing graphical objects (called the *destination*). Normally, we expect the new graphical object to obscure the objects underneath it. But that happens only because the default blend mode is `SKBlendMode.SrcOver`, which means that the source is drawn *over* the destination. The other 28 members of `SKBlendMode` cause other effects. In graphics programming, the technique of combining graphical objects in various ways is known as *compositing*.

The SKBlendModes enumeration

The SkiaSharp blend modes correspond closely to those described in the W3C [Compositing and Blending Level 1](#) specification. The Skia [SkBlendMode Overview](#) also provides helpful background information. For a general introduction to blend modes, the [Blend modes](#) article in Wikipedia is a good start. Blend modes are supported in Adobe Photoshop, so there is much additional online information about blend modes in that context.

The 29 members of the `SKBlendMode` enumeration can be divided into three categories:

PORTR-DUFF	SEPARABLE	NON-SEPARABLE
<code>Clear</code>	<code>Modulate</code>	<code>Hue</code>
<code>Src</code>	<code>Screen</code>	<code>Saturation</code>
<code>Dst</code>	<code>Overlay</code>	<code>Color</code>
<code>SrcOver</code>	<code>Darken</code>	<code>Luminosity</code>
<code>DstOver</code>	<code>Lighten</code>	
<code>SrcIn</code>	<code>ColorDodge</code>	
<code>DstIn</code>	<code>ColorBurn</code>	
<code>SrcOut</code>	<code>HardLight</code>	
<code>DstOut</code>	<code>SoftLight</code>	
<code>SrcATop</code>	<code>Difference</code>	
<code>DstATop</code>	<code>Exclusion</code>	

PORTER-DUFF	SEPARABLE	NON-SEPARABLE
Xor	Multiply	
Plus		

The names of these three categories will take on more meaning in the discussions that follow. The order that the members are listed here is the same as in the definition of the `SKBlendMode` enumeration. The 13 enumeration members in the first column have the integer values 0 to 12. The second column are enumeration members that correspond to integers 13 to 24, and the members in the third column have values of 25 to 28.

These blend modes are discussed in *approximately* the same order in the [W3C Compositing and Blending Level 1](#) document, but there are a few differences: The `Src` mode is called *Copy* in the W3C document, and `Plus` is called *Lighter*. The W3C document defines a *Normal*/blend mode that isn't included in `SKBlendModes` because it would be the same as `SrcOver`. The `Modulate` blend mode (at the top of the first column) isn't included in the W3C document, and discussion of the `Multiply` mode precedes `Screen`.

Because the `Modulate` blend mode is unique to Skia, it will be discussed as an additional Porter-Duff mode and as a separable mode.

The importance of transparency

Historically, compositing was developed in conjunction with the concept of the *alpha channel*. In a display surface such as the `SKCanvas` object and a full-color bitmap, each pixel consists of 4 bytes: 1 byte each for the red, green, and blue components, and an additional byte for transparency. This alpha component is 0 for full transparency and 0xFF for full opacity, with different levels of transparency between those values.

Many of the blend modes rely on transparency. Usually, when an `SKCanvas` is first obtained in a `PaintSurface` handler, or when an `SKCanvas` is created to draw on a bitmap, the first step is this call:

```
canvas.Clear();
```

This method replaces all the pixels of the canvas with transparent black pixels, equivalent to `new SKColor(0, 0, 0, 0)` or the integer 0x00000000. All the bytes of all the pixels are initialized to zero.

The drawing surface of an `SKCanvas` that is obtained in a `PaintSurface` handler might appear to have a white background, but that's only because the `SKCanvasView` itself has a transparent background, and the page has a white background. You can demonstrate this fact to yourself by setting the `Xamarin.Forms` `BackgroundColor` property of `SKCanvasView` to a `Xamarin.Forms` color:

```
canvasView.BackgroundColor = Color.Red;
```

Or, in a class that derives from `ContentPage`, you can set the page background color:

```
BackgroundColor = Color.Red;
```

You'll see this red background behind your SkiaSharp graphics because the SkiaSharp canvas itself is transparent.

The article [SkiaSharp Transparency](#) showed some basic techniques in using transparency to arrange multiple graphics in a composite image. The blend modes go beyond that, but transparency remains crucial to the blend modes.

SkiaSharp Porter-Duff blend modes

Use the Porter-Duff blend modes to compose scenes based on source and destination images.

SkiaSharp separable blend modes

Use the separable blend modes to alter red, green, and blue colors.

SkiaSharp non-separable blend modes

Use the non-separable blend modes to alter hue, saturation, or luminosity.

Related links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

Porter-Duff blend modes

3/5/2021 • 22 minutes to read • [Edit Online](#)



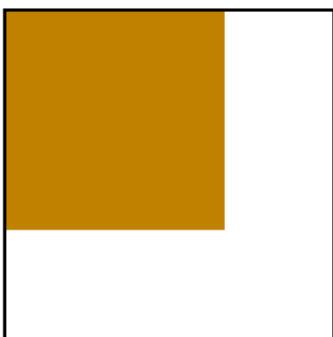
[Download the sample](#)

The Porter-Duff blend modes are named after Thomas Porter and Tom Duff, who developed an algebra of compositing while working for Lucasfilm. Their paper [Compositing Digital Images](#) was published in the July 1984 issue of *Computer Graphics*, pages 253 to 259. These blend modes are essential for compositing, which is assembling various images into a composite scene:



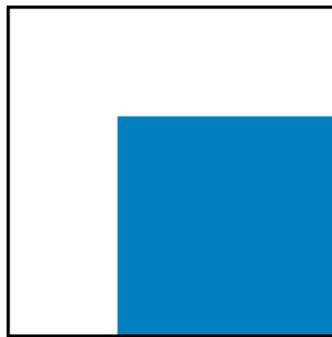
Porter-Duff concepts

Suppose a brownish rectangle occupies the left and top two-thirds of your display surface:



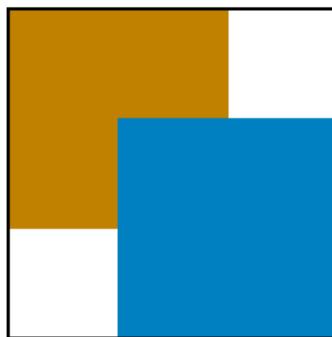
This area is called the *destination* or sometimes the *background* or *backdrop*.

You wish to draw the following rectangle, which is the same size of the destination. The rectangle is transparent except for a bluish area that occupies the right and bottom two-thirds:



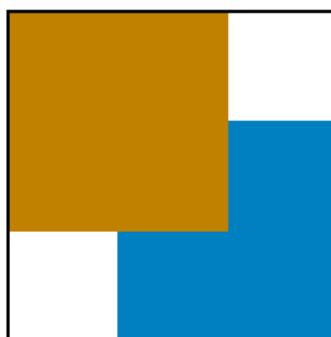
This is called the *source* or sometimes the *foreground*.

When you display the source on the destination, here's what you expect:

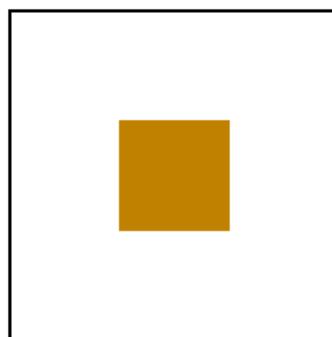


The transparent pixels of the source allow the background to show through, while the bluish source pixels obscure the background. That's the normal case, and it is referred to in SkiaSharp as `SKBlendMode.SrcOver`. That value is the default setting of the `BlendMode` property when an `SKPaint` object is first instantiated.

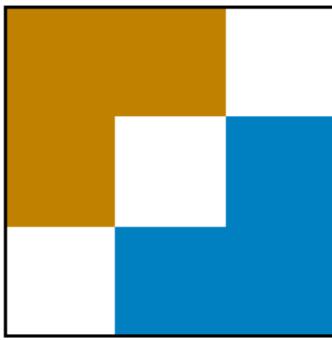
However, it's possible to specify a different blend mode for a different effect. If you specify `SKBlendMode.DstOver`, then in the area where the source and destination intersect, the destination appears instead of the source:



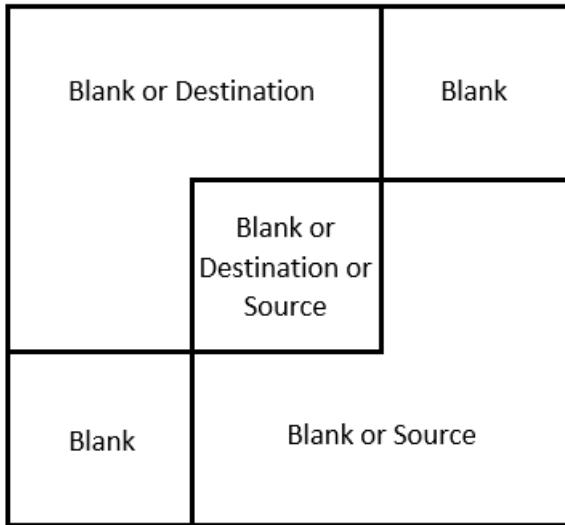
The `SKBlendMode.DstIn` blend mode displays only the area where the destination and source intersect using the destination color:



The blend mode of `SKBlendMode.Xor` (exclusive OR) causes nothing to appear where the two areas overlap:



The colored destination and source rectangles effectively divide the display surface into four unique areas that can be colored in various ways corresponding to the presence of the destination and source rectangles:



The upper-right and lower-left rectangles are always blank because both the destination and source are transparent in those areas. The destination color occupies the upper-left area, so that area can either be colored with the destination color or not at all. Similarly, the source color occupies the lower-right area, so that area can be colored with the source color or not at all. The intersection of the destination and source in the middle can be colored with the destination color, the source color, or not at all.

The total number of combinations is 2 (for the upper-left) times 2 (for the lower-right) times 3 (for the center), or 12. These are the 12 basic Porter-Duff compositing modes.

Towards the end of *Compositing Digital Images* (page 256), Porter and Duff add a 13th mode called *plus* (corresponding to the SkiaSharp `SKBlendMode.Plus` member and the W3C *Lighter* mode (which is not to be confused with the W3C *Lighten* mode.) This `Plus` mode adds the destination and source colors, a process that will be described in more detail shortly.

Skia adds a 14th mode called `Modulate` that is very similar to `Plus` except that the destination and source colors are multiplied. It can be treated as an additional Porter-Duff blend mode.

Here are the 14 Porter-Duff modes as defined in SkiaSharp. The table shows how they color each of the three non-blank areas in the diagram above:

MODE	DESTINATION	INTERSECTION	SOURCE
<code>Clear</code>			
<code>Src</code>		Source	X

MODE	DESTINATION	INTERSECTION	SOURCE
Dst	X	Destination	
SrcOver	X	Source	X
DstOver	X	Destination	X
SrcIn		Source	
DstIn		Destination	
SrcOut			X
DstOut	X		
SrcATop	X	Source	
DstATop		Destination	X
Xor	X		X
Plus	X	Sum	X
Modulate		Product	

These blend modes are symmetrical. The source and destination can be exchanged and all the modes are still available.

The naming convention of the modes follows a few simple rules:

- **Src** or **Dst** by itself means that only the source or destination pixels are visible.
- The **Over** suffix indicates what is visible in the intersection. Either the source or destination is drawn "over" the other.
- The **In** suffix means that only the intersection is colored. The output is restricted to only the part of the source or destination that is "in" the other.
- The **Out** suffix means that the intersection is not colored. The output is only the part of the source or destination that is "out" of the intersection.
- The **ATop** suffix is the union of **In** and **Out**. It includes the area where the source or destination is "atop" of the other.

Notice the difference with the **Plus** and **Modulate** modes. These modes are performing a different type of calculation on the source and destination pixels. They are described in more detail shortly.

The **Porter-Duff Grid** page shows all 14 modes on one screen in the form of a grid. Each mode is a separate instance of `SKCanvasView`. For that reason, a class is derived from `SKCanvasView` named `PorterDuffCanvasView`. The static constructor creates two bitmaps of the same size, one with a brownish rectangle in its upper-left area and another with a bluish rectangle:

```
class PorterDuffCanvasView : SKCanvasView
{
    static SKBitmap srcBitmap, dstBitmap;

    static PorterDuffCanvasView()
    {
        dstBitmap = new SKBitmap(300, 300);
        srcBitmap = new SKBitmap(300, 300);

        using (SKPaint paint = new SKPaint())
        {
            using (SKCanvas canvas = new SKCanvas(dstBitmap))
            {
                canvas.Clear();
                paint.Color = new SKColor(0xC0, 0x80, 0x00);
                canvas.DrawRect(new SKRect(0, 0, 200, 200), paint);
            }
            using (SKCanvas canvas = new SKCanvas(srcBitmap))
            {
                canvas.Clear();
                paint.Color = new SKColor(0x00, 0x80, 0xC0);
                canvas.DrawRect(new SKRect(100, 100, 300, 300), paint);
            }
        }
    ...
}
```

The instance constructor has a parameter of type `SKBlendMode`. It saves this parameter in a field.

```

class PorterDuffCanvasView : SKCanvasView
{
    ...
    SKBlendMode blendMode;

    public PorterDuffCanvasView(SKBlendMode blendMode)
    {
        this.blendMode = blendMode;
    }

    protected override void OnPaintSurface(SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Find largest square that fits
        float rectSize = Math.Min(info.Width, info.Height);
        float x = (info.Width - rectSize) / 2;
        float y = (info.Height - rectSize) / 2;
        SKRect rect = new SKRect(x, y, x + rectSize, y + rectSize);

        // Draw destination bitmap
        canvas.DrawBitmap(dstBitmap, rect);

        // Draw source bitmap
        using (SKPaint paint = new SKPaint())
        {
            paint.BlendMode = blendMode;
            canvas.DrawBitmap(srcBitmap, rect, paint);
        }

        // Draw outline
        using (SKPaint paint = new SKPaint())
        {
            paint.Style = SKPaintStyle.Stroke;
            paint.Color = SKColors.Black;
            paint.StrokeWidth = 2;
            rect.Inflate(-1, -1);
            canvas.DrawRect(rect, paint);
        }
    }
}

```

The `OnPaintSurface` override draws the two bitmaps. The first is drawn normally:

```
canvas.DrawBitmap(dstBitmap, rect);
```

The second is drawn with an `SKPaint` object where the `BlendMode` property has been set to the constructor argument:

```

using (SKPaint paint = new SKPaint())
{
    paint.BlendMode = blendMode;
    canvas.DrawBitmap(srcBitmap, rect, paint);
}

```

The remainder of the `OnPaintSurface` override draws a rectangle around the bitmap to indicate their sizes.

The `PorterDuffGridPage` class creates fourteen instances of `PorterDuffCanvasView`, one for each member of the

`blendModes` array. The order of the `SKBlendModes` members in the array is a little different than the table in order to position similar modes adjacent to each other. The 14 instances of `PorterDuffCanvasView` are organized along with labels in a `Grid`:

```
public class PorterDuffGridPage : ContentPage
{
    public PorterDuffGridPage()
    {
        Title = "Porter-Duff Grid";

        SKBlendMode[] blendModes =
        {
            SKBlendMode.Src, SKBlendMode.Dst, SKBlendMode.SrcOver, SKBlendMode.DstOver,
            SKBlendMode.SrcIn, SKBlendMode.DstIn, SKBlendMode.SrcOut, SKBlendMode.DstOut,
            SKBlendMode.SrcATop, SKBlendMode.DstATop, SKBlendMode.Xor, SKBlendMode.Plus,
            SKBlendMode.Modulate, SKBlendMode.Clear
        };

        Grid grid = new Grid
        {
            Margin = new Thickness(5)
        };

        for (int row = 0; row < 4; row++)
        {
            grid.RowDefinitions.Add(new RowDefinition { Height = GridLength.Auto });
            grid.RowDefinitions.Add(new RowDefinition { Height = GridLength.Star });
        }

        for (int col = 0; col < 3; col++)
        {
            grid.ColumnDefinitions.Add(new ColumnDefinition { Width = GridLength.Star });
        }

        for (int i = 0; i < blendModes.Length; i++)
        {
            SKBlendMode blendMode = blendModes[i];
            int row = 2 * (i / 4);
            int col = i % 4;

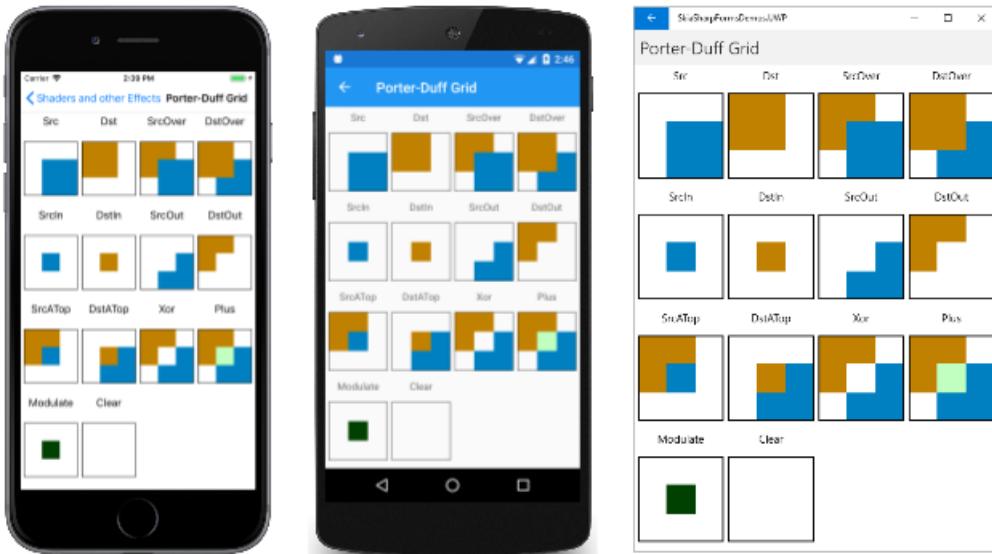
            Label label = new Label
            {
                Text = blendMode.ToString(),
                HorizontalTextAlignment = TextAlignment.Center
            };
            Grid.SetRow(label, row);
            Grid.SetColumn(label, col);
            grid.Children.Add(label);

            PorterDuffCanvasView canvasView = new PorterDuffCanvasView(blendMode);

            Grid.SetRow(canvasView, row + 1);
            Grid.SetColumn(canvasView, col);
            grid.Children.Add(canvasView);
        }

        Content = grid;
    }
}
```

Here's the result:



You'll want to convince yourself that transparency is crucial to the proper functioning of the Porter-Duff blend modes. The `PorterDuffCanvasView` class contains a total of three calls to the `Canvas.Clear` method. All of them use the parameterless method, which sets all the pixels to transparent:

```
canvas.Clear();
```

Try changing any of those calls so that the pixels are set to opaque white:

```
canvas.Clear(SKColors.White);
```

Following that change, some of the blend modes will seem to work, but others will not. If you set the background of the source bitmap to white, then the `SrcOver` mode doesn't work because there's no transparent pixels in the source bitmap to let the destination show through. If you set the background of the destination bitmap or the canvas to white, then `DstOver` doesn't work because the destination doesn't have any transparent pixels.

There might be a temptation to replace the bitmaps in the **Porter-Duff Grid** page with simpler `DrawRect` calls. That will work for the destination rectangle but not for the source rectangle. The source rectangle must encompass more than just the bluish-colored area. The source rectangle must include a transparent area that corresponds to the colored area of the destination. Only then will these blend modes work.

Using mattes with Porter-Duff

The **Brick-Wall Compositing** page shows an example of a classic compositing task: A picture needs to be assembled from several pieces, including a bitmap with a background that needs to be eliminated. Here's the `SeatedMonkey.jpg` bitmap with the problematic background:



In preparation for compositing, a corresponding *matte* was created, which is another bitmap that is black where you want the image to appear and transparent otherwise. This file is named **SeatedMonkeyMatte.png** and is among the resources in the **Media** folder in the [SkiaSharpFormsDemos](#) sample:



This is *not* an expertly created matte. Optimally, the matte should include partially transparent pixels around the edge of the black pixels, and this matte does not.

The XAML file for the **Brick-Wall Compositing** page instantiates an `SKCanvasView` and a `Button` that guides the user through the process of composing the final image:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Effects.BrickWallCompositingPage"
    Title="Brick-Wall Compositing">

    <StackLayout>
        <skia:SKCanvasView x:Name="canvasView"
            VerticalOptions="FillAndExpand"
            PaintSurface="OnCanvasViewPaintSurface" />

        <Button Text="Show sitting monkey"
            HorizontalOptions="Center"
            Margin="0, 10"
            Clicked="OnButtonClicked" />

    </StackLayout>
</ContentPage>
```

The code-behind file loads the two bitmaps that it needs and handles the `Clicked` event of the `Button`. For every `Button` click, the `step` field is incremented and a new `Text` property is set for the `Button`. When `step`

reaches 5, it is set back to 0:

```
public partial class BrickWallCompositingPage : ContentPage
{
    SKBitmap monkeyBitmap = BitmapExtensions.LoadBitmapResource(
        typeof(BrickWallCompositingPage),
        "SkiaSharpFormsDemos.Media.SeatedMonkey.jpg");

    SKBitmap matteBitmap = BitmapExtensions.LoadBitmapResource(
        typeof(BrickWallCompositingPage),
        "SkiaSharpFormsDemos.Media.SeatedMonkeyMatte.png");

    int step = 0;

    public BrickWallCompositingPage ()
    {
        InitializeComponent ();
    }

    void OnButtonClicked(object sender, EventArgs args)
    {
        Button btn = (Button)sender;
        step = (step + 1) % 5;

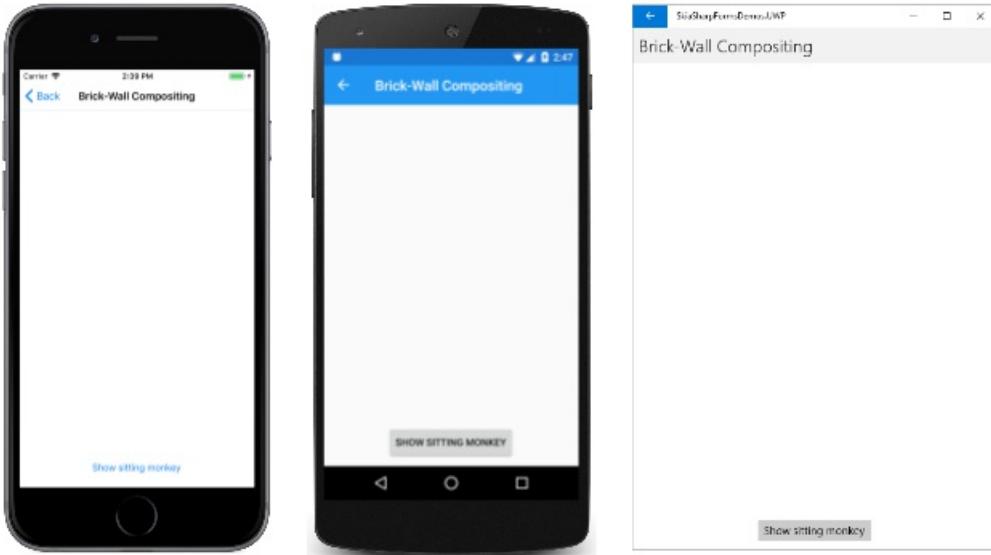
        switch (step)
        {
            case 0: btn.Text = "Show sitting monkey"; break;
            case 1: btn.Text = "Draw matte with DstIn"; break;
            case 2: btn.Text = "Draw sidewalk with DstOver"; break;
            case 3: btn.Text = "Draw brick wall with DstOver"; break;
            case 4: btn.Text = "Reset"; break;
        }

        canvasView.InvalidateSurface();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();
        ...
    }
}
```

When the program first runs, nothing is visible except the `Button`:

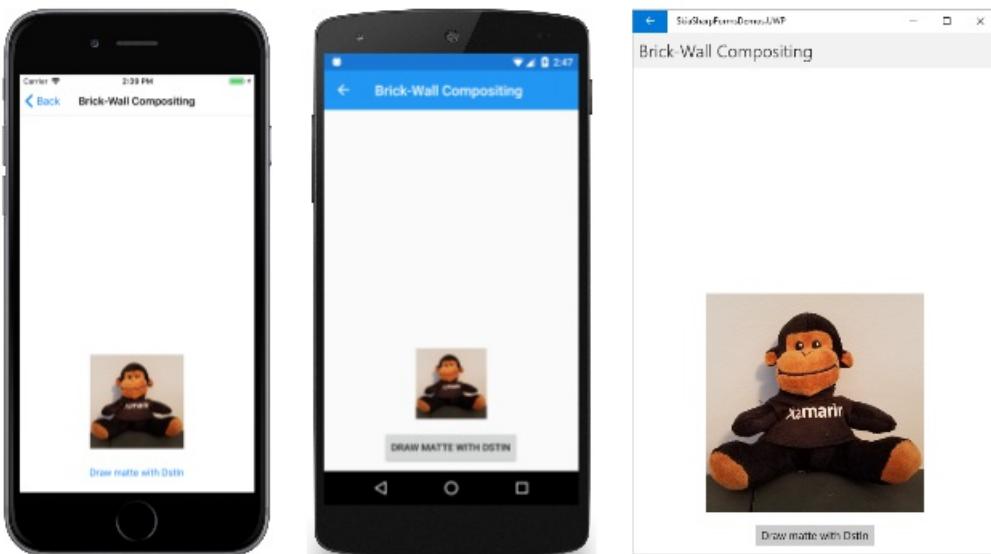


Pressing the `Button` once causes `step` to increment to 1, and the `PaintSurface` handler now displays `SeatedMonkey.jpg`:

```
public partial class BrickWallCompositingPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        ...
        float x = (info.Width - monkeyBitmap.Width) / 2;
        float y = info.Height - monkeyBitmap.Height;

        // Draw monkey bitmap
        if (step >= 1)
        {
            canvas.DrawBitmap(monkeyBitmap, x, y);
        }
        ...
    }
}
```

There's no `SKPaint` object and hence no blend mode. The bitmap appears at the bottom of the screen:



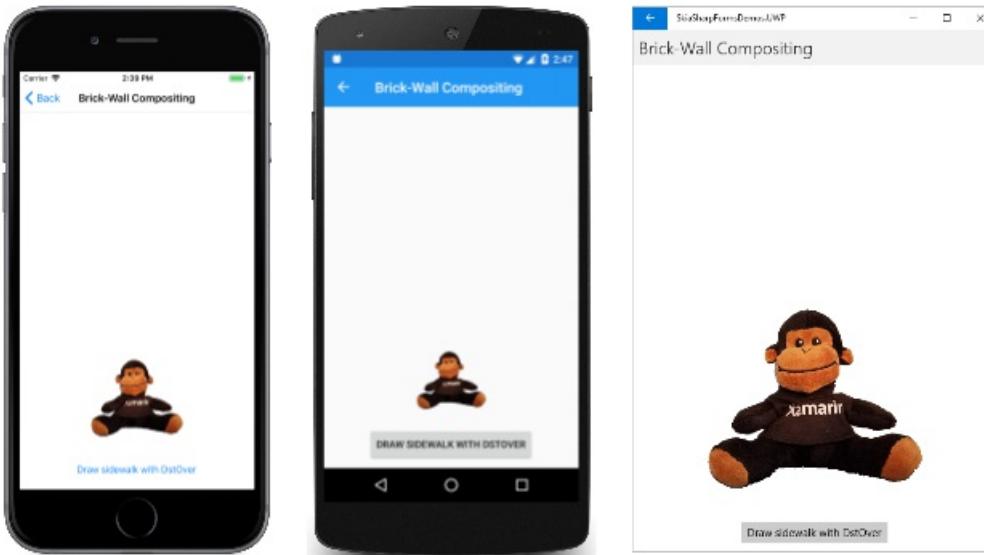
Press the `Button` again and `step` increments to 2. This is the crucial step of displaying the `SeatedMonkeyMatte.png` file:

```

public partial class BrickWallCompositingPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        ...
        // Draw matte to exclude monkey's surroundings
        if (step >= 2)
        {
            using (SKPaint paint = new SKPaint())
            {
                paint.BlendMode = SKBlendMode.DstIn;
                canvas.DrawBitmap(matteBitmap, x, y, paint);
            }
        }
        ...
    }
}

```

The blend mode is `SKBlendMode.DstIn`, which means that the destination will be preserved in areas corresponding to non-transparent areas of the source. The remainder of the destination rectangle corresponding to the original bitmap becomes transparent:



The background has been removed.

The next step is to draw a rectangle that resembles a sidewalk that the monkey is sitting on. The appearance of this sidewalk is based on a composition of two shaders: a solid color shader and a Perlin noise shader:

```

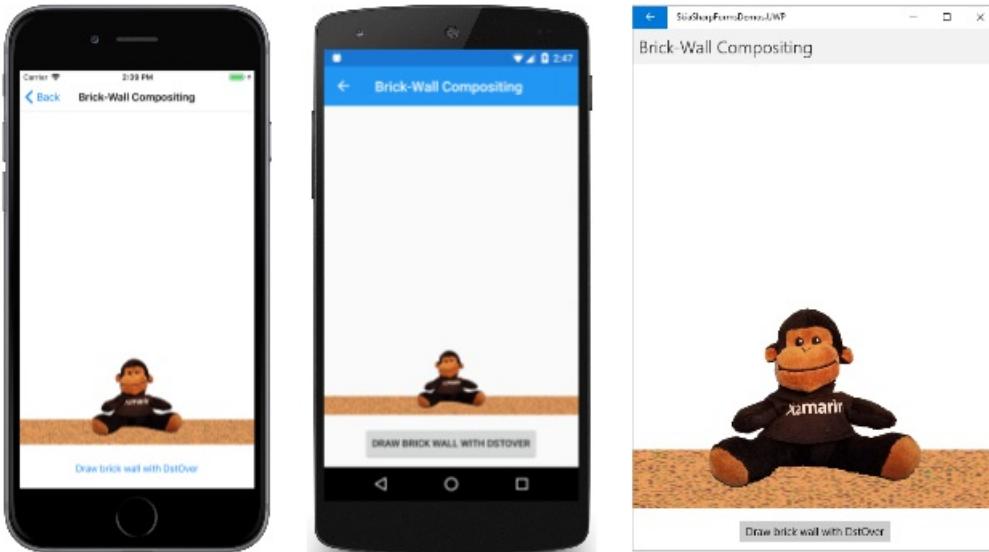
public partial class BrickWallCompositingPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        ...
        const float sidewalkHeight = 80;
        SKRect rect = new SKRect(info.Rect.Left, info.Rect.Bottom - sidewalkHeight,
                               info.Rect.Right, info.Rect.Bottom);

        // Draw gravel sidewalk for monkey to sit on
        if (step >= 3)
        {
            using (SKPaint paint = new SKPaint())
            {
                paint.Shader = SKShader.CreateCompose(
                    SKShader.CreateColor(SKColors.SandyBrown),
                    SKShader.CreatePerlinNoiseTurbulence(0.1f, 0.3f, 1, 9));

                paint.BlendMode = SKBlendMode.DstOver;
                canvas.DrawRect(rect, paint);
            }
        }
        ...
    }
}

```

Because this sidewalk must go behind the monkey, the blend mode is `DstOver`. The destination appears only where the background is transparent:



The final step is adding a brick wall. The program uses the brick-wall bitmap tile available as the static property `BrickWallTile` in the `AlgorithmicBrickWallPage` class. A translation transform is added to the `SKShader.CreateBitmap` call to shift the tiles so that the bottom row is a full tile:

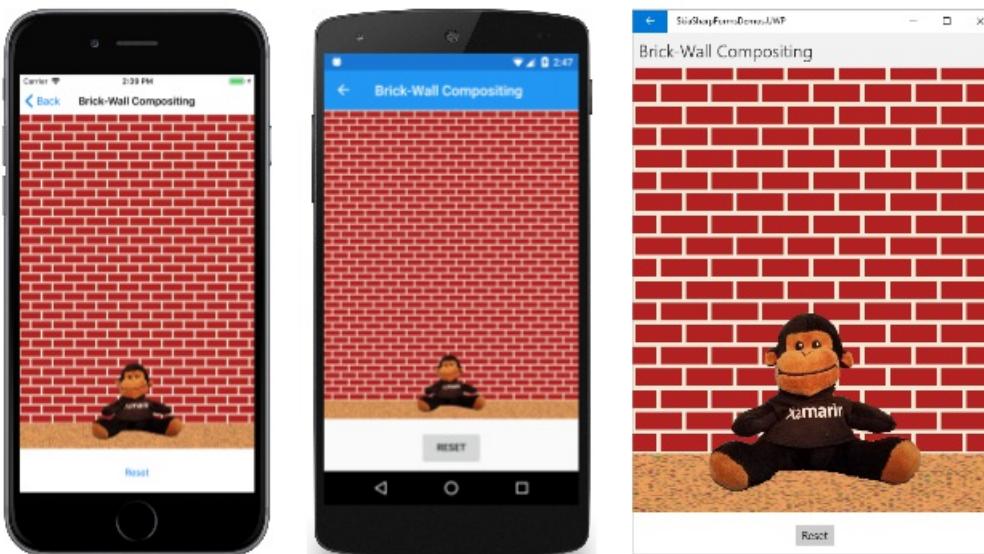
```

public partial class BrickWallCompositingPage : ContentPage
{
    ...
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        ...
        // Draw bitmap tiled brick wall behind monkey
        if (step >= 4)
        {
            using (SKPaint paint = new SKPaint())
            {
                SKBitmap bitmap = AlgorithmicBrickWallPage.BrickWallTile;
                float yAdjust = (info.Height - sidewalkHeight) % bitmap.Height;

                paint.Shader = SKShader.CreateBitmap(bitmap,
                    SKShaderTileMode.Repeat,
                    SKShaderTileMode.Repeat,
                    SKMatrix.MakeTranslation(0, yAdjust));
                paint.BlendMode = SKBlendMode.DstOver;
                canvas.DrawRect(info.Rect, paint);
            }
        }
    }
}

```

For convenience, the `DrawRect` call displays this shader over the entire canvas, but the `DstOver` mode limits the output to only the area of the canvas that is still transparent:



Obviously there are other ways to compose this scene. It could be built up starting at the background and progressing to the foreground. But using the blend modes gives you more flexibility. In particular, the use of the matte allows the background of a bitmap to be excluded from the composed scene.

As you learned in the article [Clipping with Paths and Regions](#), the `SKCanvas` class defines three types of clipping, corresponding to the `ClipRect`, `ClipPath`, and `ClipRegion` methods. The Porter-Duff blend modes add another type of clipping, which allows restricting an image to anything that you can draw, including bitmaps. The matte used in **Brick-Wall Compositing** essentially defines a clipping area.

Gradient transparency and transitions

The examples of the Porter-Duff blend modes shown earlier in this article have all involved images that consisted of opaque pixels and transparent pixels, but not partially transparent pixels. The blend-mode functions are defined for those pixels as well. The following table is a more formal definition of the Porter-Duff blend

modes that uses notation found in the Skia [SkBlendMode Reference](#). (Because `SkBlendMode Reference` is a Skia reference, C++ syntax is used.)

Conceptually, the red, green, blue, and alpha components of each pixel are converted from bytes to floating-point numbers in the range of 0 to 1. For the alpha channel, 0 is fully transparent and 1 is fully opaque

The notation in the table below uses the following abbreviations:

- `Da` is the destination alpha channel
- `Dc` is the destination RGB color
- `Sa` is the source alpha channel
- `Sc` is the source RGB color

The RGB colors are pre-multiplied by the alpha value. For example, if `Sc` represents pure red but `Sa` is 0x80, then the RGB color is (0x80, 0, 0). If `Sa` is 0, then all the RGB components are also zero.

The result is shown in brackets with the alpha channel and the RGB color separated by a comma: [`alpha, color`]. For the color, the calculation is performed separately for the red, green, and blue components:

MODE	OPERATION
<code>Clear</code>	[0, 0]
<code>Src</code>	[<code>Sa</code> , <code>Sc</code>]
<code>Dst</code>	[<code>Da</code> , <code>Dc</code>]
<code>SrcOver</code>	[$Sa + Da \cdot (1 - Sa)$, $Sc + Dc \cdot (1 - Sa)$]
<code>DstOver</code>	[$Da + Sa \cdot (1 - Da)$, $Dc + Sc \cdot (1 - Da)$]
<code>SrcIn</code>	[$Sa \cdot Da$, $Sc \cdot Da$]
<code>DstIn</code>	[$Da \cdot Sa$, $Dc \cdot Sa$]
<code>SrcOut</code>	[$Sa \cdot (1 - Da)$, $Sc \cdot (1 - Da)$]
<code>DstOut</code>	[$Da \cdot (1 - Sa)$, $Dc \cdot (1 - Sa)$]
<code>SrcATop</code>	[Da , $Sc \cdot Da + Dc \cdot (1 - Sa)$]
<code>DstATop</code>	[Sa , $Dc \cdot Sa + Sc \cdot (1 - Da)$]
<code>Xor</code>	[$Sa + Da - 2 \cdot Sa \cdot Da$, $Sc \cdot (1 - Da) + Dc \cdot (1 - Sa)$]
<code>Plus</code>	[$Sa + Da$, $Sc + Dc$]
<code>Modulate</code>	[$Sa \cdot Da$, $Sc \cdot Dc$]

These operations are easier to analyze when `Da` and `Sa` are either 0 or 1. For example, for the default `SrcOver` mode, if `Sa` is 0, then `Sc` is also 0, and the result is [`Da`, `Dc`], the destination alpha and color. If `Sa` is 1, then the result is [`Sa`, `Sc`], the source alpha and color, or [1, `Sc`].

The `Plus` and `Modulate` modes are a little different from the others in that new colors can result from the

combination of the source and the destination. The `Plus` mode can be interpreted either with byte components or floating-point components. In the **Porter-Duff Grid** page shown earlier, the destination color is `(0xC0, 0x80, 0x00)` and the source color is `(0x00, 0x80, 0xC0)`. Each pair of components is added but the sum is clamped at `0xFF`. The result is the color `(0xC0, 0xFF, 0xC0)`. That's the color shown in the intersection.

For the `Modulate` mode, the RGB values must be converted to floating-point. The destination color is `(0.75, 0.5, 0)` and the source is `(0, 0.5, 0.75)`. The RGB components are each multiplied together, and the result is `(0, 0.25, 0)`. That's the color shown in the intersection in the **Porter-Duff Grid** page for this mode.

The **Porter-Duff Transparency** page allows you to examine how the Porter-Duff blend modes operate on graphical objects that are partially transparent. The XAML file includes a `Picker` with the Porter-Duff modes:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp;assembly=SkiaSharp"
    xmlns:skiaviews="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Effects.PorterDuffTransparencyPage"
    Title="Porter-Duff Transparency">

    <StackLayout>
        <skiaviews:SKCanvasView x:Name="canvasView"
            VerticalOptions="FillAndExpand"
            PaintSurface="OnCanvasViewPaintSurface" />

        <Picker x:Name="blendModePicker"
            Title="Blend Mode"
            Margin="10"
            SelectedIndexChanged="OnPickerSelectedIndexChanged">
            <Picker.ItemsSource>
                <x:Array Type="{x:Type skia:SKBlendMode}">
                    <x:Static Member="skia:SKBlendMode.Clear" />
                    <x:Static Member="skia:SKBlendMode.Src" />
                    <x:Static Member="skia:SKBlendMode.Dst" />
                    <x:Static Member="skia:SKBlendMode.SrcOver" />
                    <x:Static Member="skia:SKBlendMode.DstOver" />
                    <x:Static Member="skia:SKBlendMode.SrcIn" />
                    <x:Static Member="skia:SKBlendMode.DstIn" />
                    <x:Static Member="skia:SKBlendMode.SrcOut" />
                    <x:Static Member="skia:SKBlendMode.DstOut" />
                    <x:Static Member="skia:SKBlendMode.SrcATop" />
                    <x:Static Member="skia:SKBlendMode.DstATop" />
                    <x:Static Member="skia:SKBlendMode.Xor" />
                    <x:Static Member="skia:SKBlendMode.Plus" />
                    <x:Static Member="skia:SKBlendMode.Modulate" />
                </x:Array>
            </Picker.ItemsSource>
            <Picker.SelectedIndex>
                3
            </Picker.SelectedIndex>
        </Picker>
    </StackLayout>
</ContentPage>
```

The code-behind file fills two rectangles of the same size using a linear gradient. The destination gradient is from the upper right to the lower left. It is brownish in the upper-right corner but then towards the center begins fading to transparent, and is transparent in the lower-left corner.

The source rectangle has a gradient from the upper left to the lower right. The upper-left corner is bluish but again fades to transparent, and is transparent in the lower-right corner.

```

public partial class PorterDuffTransparencyPage : ContentPage
{
    public PorterDuffTransparencyPage()
    {
        InitializeComponent();
    }

    void OnPickerSelectedIndexChanged(object sender, EventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Make square display rectangle smaller than canvas
        float size = 0.9f * Math.Min(info.Width, info.Height);
        float x = (info.Width - size) / 2;
        float y = (info.Height - size) / 2;
        SKRect rect = new SKRect(x, y, x + size, y + size);

        using (SKPaint paint = new SKPaint())
        {
            // Draw destination
            paint.Shader = SKShader.CreateLinearGradient(
                new SKPoint(rect.Right, rect.Top),
                new SKPoint(rect.Left, rect.Bottom),
                new SKColor[] { new SKColor(0xC0, 0x80, 0x00),
                               new SKColor(0xC0, 0x80, 0x00, 0) },
                new float[] { 0.4f, 0.6f },
                SKShaderTileMode.Clamp);

            canvas.DrawRect(rect, paint);

            // Draw source
            paint.Shader = SKShader.CreateLinearGradient(
                new SKPoint(rect.Left, rect.Top),
                new SKPoint(rect.Right, rect.Bottom),
                new SKColor[] { new SKColor(0x00, 0x80, 0xC0),
                               new SKColor(0x00, 0x80, 0xC0, 0) },
                new float[] { 0.4f, 0.6f },
                SKShaderTileMode.Clamp);

            // Get the blend mode from the picker
            paint.BlendMode = blendModePicker.SelectedIndex == -1 ? 0 :
                (SKBlendMode)blendModePicker.SelectedItem;

            canvas.DrawRect(rect, paint);

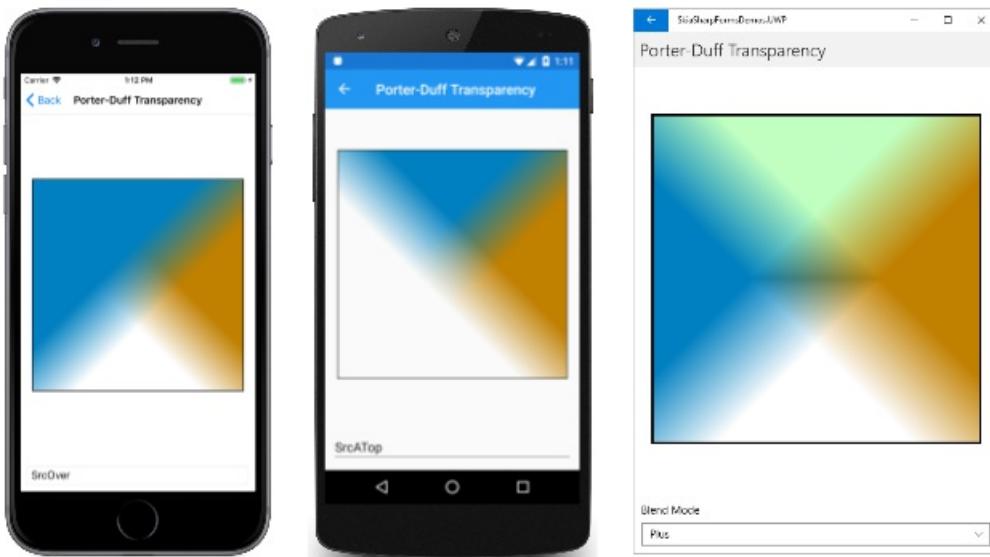
            // Stroke surrounding rectangle
            paint.Shader = null;
            paint.BlendMode = SKBlendMode.SrcOver;
            paint.Style = SKPaintStyle.Stroke;
            paint.Color = SKColors.Black;
            paint.StrokeWidth = 3;
            canvas.DrawRect(rect, paint);
        }
    }
}

```

This program demonstrates that the Porter-Duff blend modes can be used with graphic objects other than

bitmaps. However, the source must include a transparent area. This is the case here because the gradient fills the rectangle, but part of the gradient is transparent.

Here are three examples:



The configuration of the destination and source is very similar to the diagrams shown in page 255 of the original Porter-Duff [Compositing Digital Images](#) paper, but this page demonstrates that the blend modes are well-behaved for areas of partial transparency.

You can use transparent gradients for some different effects. One possibility is masking, which is similar to the technique shown in the [Radial gradients for masking](#) section of the [SkiaSharp circular gradients](#) page. Much of the [Compositing Mask](#) page is similar to that earlier program. It loads a bitmap resource and determines a rectangle in which to display it. A radial gradient is created based on a pre-determined center and radius:

```

public class CompositingMaskPage : ContentPage
{
    SKBitmap bitmap = BitmapExtensions.LoadBitmapResource(
        typeof(CompositingMaskPage),
        "SkiaSharpFormsDemos.Media.MountainClimbers.jpg");

    static readonly SKPoint CENTER = new SKPoint(180, 300);
    static readonly float RADIUS = 120;

    public CompositingMaskPage ()
    {
        Title = "Compositing Mask";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Find rectangle to display bitmap
        float scale = Math.Min((float)info.Width / bitmap.Width,
                               (float)info.Height / bitmap.Height);

        SKRect rect = SKRect.Create(scale * bitmap.Width, scale * bitmap.Height);

        float x = (info.Width - rect.Width) / 2;
        float y = (info.Height - rect.Height) / 2;
        rect.Offset(x, y);

        // Display bitmap in rectangle
        canvas.DrawBitmap(bitmap, rect);

        // Adjust center and radius for scaled and offset bitmap
        SKPoint center = new SKPoint(scale * CENTER.X + x,
                                      scale * CENTER.Y + y);
        float radius = scale * RADIUS;

        using (SKPaint paint = new SKPaint())
        {
            paint.Shader = SKShader.CreateRadialGradient(
                center,
                radius,
                new SKColor[] { SKColors.Black,
                               SKColors.Transparent },
                new float[] { 0.6f, 1 },
                SKShaderTileMode.Clamp);

            paint.BlendMode = SKBlendMode.DstIn;

            // Display rectangle using that gradient and blend mode
            canvas.DrawRect(rect, paint);
        }

        canvas.DrawColor(SKColors.Pink, SKBlendMode.DstOver);
    }
}

```

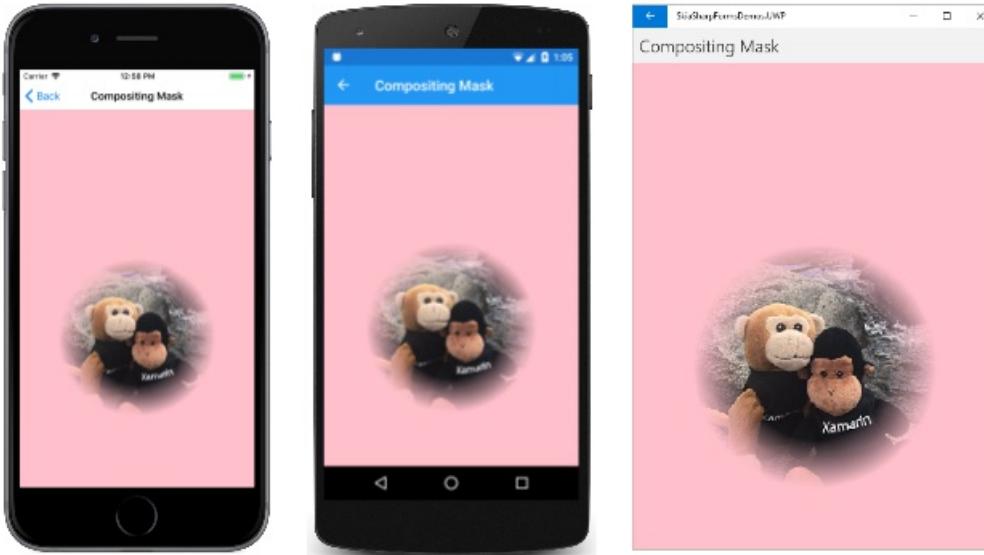
The difference with this program is that the gradient begins with black in the center and ends with transparency. It is displayed on the bitmap with a blend mode of `DstIn`, which shows the destination only in the areas of the

source that are not transparent.

After the `DrawRect` call, the entire surface of the canvas is transparent except for the circle defined by the radial gradient. A final call is made:

```
canvas.DrawColor(SKColors.Pink, SKBlendMode.DstOver);
```

All the transparent areas of the canvas are colored pink:



You can also use Porter-Duff modes and partially transparent gradients for transitions from one image to another. The **Gradient Transitions** page includes a `Slider` to indicate a progress level in the transition from 0 to 1, and a `Picker` to choose the type of transition you want:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Effects.GradientTransitionsPage"
    Title="Gradient Transitions">

    <StackLayout>
        <skia:SKCanvasView x:Name="canvasView"
            VerticalOptions="FillAndExpand"
            PaintSurface="OnCanvasViewPaintSurface" />

        <Slider x:Name="progressSlider"
            Margin="10, 0"
            ValueChanged="OnSliderValueChanged" />

        <Label Text="{Binding Source={x:Reference progressSlider},
            Path=Value,
            StringFormat='Progress = {0:F2}'}"
            HorizontalTextAlignment="Center" />

        <Picker x:Name="transitionPicker"
            Title="Transition"
            Margin="10"
            SelectedIndexChanged="OnPickerSelectedIndexChanged" />

    </StackLayout>
</ContentPage>
```

The code-behind file loads two bitmap resources to demonstrate the transition. These are the same two images used in the **Bitmap Dissolve** page earlier in this article. The code also defines an enumeration with three

members corresponding to three types of gradients — linear, radial, and sweep. These values are loaded into the

Picker :

```
public partial class GradientTransitionsPage : ContentPage
{
    SKBitmap bitmap1 = BitmapExtensions.LoadBitmapResource(
        typeof(GradientTransitionsPage),
        "SkiaSharpFormsDemos.Media.SeatedMonkey.jpg");

    SKBitmap bitmap2 = BitmapExtensions.LoadBitmapResource(
        typeof(GradientTransitionsPage),
        "SkiaSharpFormsDemos.Media.FacePalm.jpg");

    enum TransitionMode
    {
        Linear,
        Radial,
        Sweep
    };

    public GradientTransitionsPage ()
    {
        InitializeComponent ();

        foreach (TransitionMode mode in Enum.GetValues(typeof(TransitionMode)))
        {
            transitionPicker.Items.Add(mode.ToString());
        }

        transitionPicker.SelectedIndex = 0;
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnPickerSelectedIndexChanged(object sender, EventArgs args)
    {
        canvasView.InvalidateSurface();
    }
    ...
}
```

The code-behind file creates three `SKPaint` objects. The `paint0` object doesn't use a blend mode. This paint object is used to draw a rectangle with a gradient that goes from black to transparent as indicated in the `colors` array. The `positions` array is based on the position of the `Slider`, but adjusted somewhat. If the `Slider` is at its minimum or maximum, the `progress` values are 0 or 1, and one of the two bitmaps should be fully visible. The `positions` array must be set accordingly for those values.

If the `progress` value is 0, then the `positions` array contains the values -0.1 and 0. SkiaSharp will adjust that first value to be equal to 0, which means that the gradient is black only at 0 and transparent otherwise. When `progress` is 0.5, then the array contains the values 0.45 and 0.55. The gradient is black from 0 to 0.45, then transitions to transparent, and is fully transparent from 0.55 to 1. When `progress` is 1, the `positions` array is 1 and 1.1, which means the gradient is black from 0 to 1.

The `colors` and `position` arrays are both used in the three methods of `skshader` that create a gradient. Only one of these shaders is created based on the `Picker` selection:

```
public partial class GradientTransitionsPage : ContentPage
{
    ...
}
```

```

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    // Assume both bitmaps are square for display rectangle
    float size = Math.Min(info.Width, info.Height);
    SKRect rect = SKRect.Create(size, size);
    float x = (info.Width - size) / 2;
    float y = (info.Height - size) / 2;
    rect.Offset(x, y);

    using (SKPaint paint0 = new SKPaint())
    using (SKPaint paint1 = new SKPaint())
    using (SKPaint paint2 = new SKPaint())
    {
        SKColor[] colors = new SKColor[] { SKColors.Black,
                                           SKColors.Transparent };

        float progress = (float)progressSlider.Value;

        float[] positions = new float[]{ 1.1f * progress - 0.1f,
                                         1.1f * progress };

        switch ((TransitionMode)transitionPicker.SelectedIndex)
        {
            case TransitionMode.Linear:
                paint0.Shader = SKShader.CreateLinearGradient(
                    new SKPoint(rect.Left, 0),
                    new SKPoint(rect.Right, 0),
                    colors,
                    positions,
                    SKShaderTileMode.Clamp);
                break;

            case TransitionMode.Radial:
                paint0.Shader = SKShader.CreateRadialGradient(
                    new SKPoint(rect.MidX, rect.MidY),
                    (float)Math.Sqrt(Math.Pow(rect.Width / 2, 2) +
                        Math.Pow(rect.Height / 2, 2)),
                    colors,
                    positions,
                    SKShaderTileMode.Clamp);
                break;

            case TransitionMode.Sweep:
                paint0.Shader = SKShader.CreateSweepGradient(
                    new SKPoint(rect.MidX, rect.MidY),
                    colors,
                    positions);
                break;
        }

        canvas.DrawRect(rect, paint0);

        paint1.BlendMode = SKBlendMode.SrcOut;
        canvas.DrawBitmap(bitmap1, rect, paint1);

        paint2.BlendMode = SKBlendMode.DstOver;
        canvas.DrawBitmap(bitmap2, rect, paint2);
    }
}
}

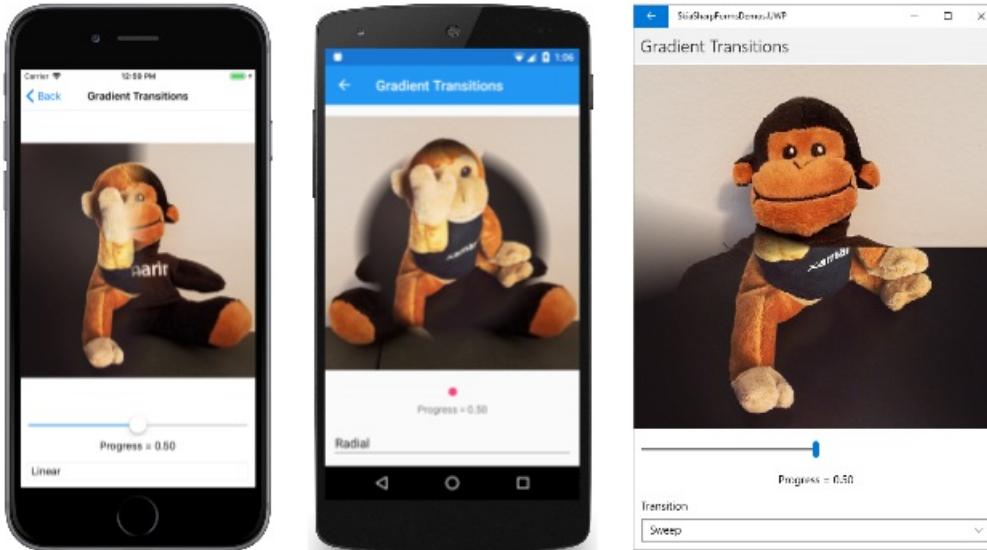
```

That gradient is displayed in the rectangle without a blend mode. After that `DrawRect` call, the canvas simply

contains a gradient from black to transparent. The amount of black increases with higher `Slider` values.

In the final four statements of the `PaintSurface` handler, the two bitmaps are displayed. The `SrcOut` blend mode means that the first bitmap is displayed only in the transparent areas of the background. The `DstOver` mode for the second bitmap means that the second bitmap is displayed only in those areas where the first bitmap is not displayed.

The following screenshots show the three different transitions types, each at the 50% mark:



Related links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

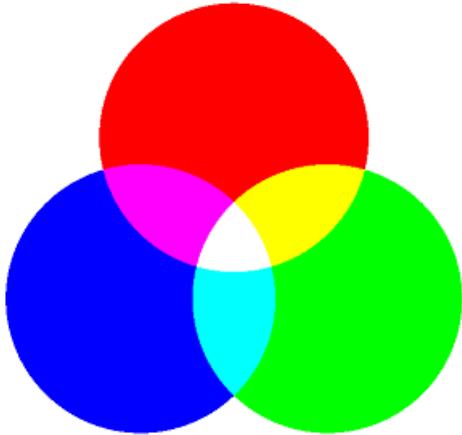
The separable blend modes

3/5/2021 • 9 minutes to read • [Edit Online](#)



[Download the sample](#)

As you saw in the article [SkiaSharp Porter-Duff blend modes](#), the Porter-Duff blend modes generally perform clipping operations. The separable blend modes are different. The separable modes alter the individual red, green, and blue color components of an image. Separable blend modes can mix color to demonstrate that the combination of red, green, and blue is indeed white:



Lighten and darken two ways

It is common to have a bitmap that is somewhat too dark or too light. You can use separable blend modes to lighten or darken the image. Indeed, two of the separable blend modes in the `SKBlendMode` enumeration are named `Lighten` and `Darken`.

These two modes are demonstrated in the [Lighten and Darken](#) page. The XAML file instantiates two `SKCanvasView` objects and two `Slider` views:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Effects.LightenAndDarkenPage"
    Title="Lighten and Darken">
    <StackLayout>
        <skia:SKCanvasView x:Name="lightenCanvasView"
            VerticalOptions="FillAndExpand"
            PaintSurface="OnCanvasViewPaintSurface" />

        <Slider x:Name="lightenSlider"
            Margin="10"
            ValueChanged="OnSliderValueChanged" />

        <skia:SKCanvasView x:Name="darkenCanvasView"
            VerticalOptions="FillAndExpand"
            PaintSurface="OnCanvasViewPaintSurface" />

        <Slider x:Name="darkenSlider"
            Margin="10"
            ValueChanged="OnSliderValueChanged" />
    </StackLayout>
</ContentPage>
```

The first `SKCanvasView` and `Slider` demonstrate `SKBlendMode.Lighten` and the second pair demonstrates `SKBlendMode.Darken`. The two `Slider` views share the same `ValueChanged` handler, and the two `SKCanvasView` share the same `PaintSurface` handler. Both event handlers check which object is firing the event:

```

public partial class LightenAndDarkenPage : ContentPage
{
    SKBitmap bitmap = BitmapExtensions.LoadBitmapResource(
        typeof(SeparableBlendModesPage),
        "SkiaSharpFormsDemos.Media.Banana.jpg");

    public LightenAndDarkenPage ()
    {
        InitializeComponent ();
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        if ((Slider)sender == lightenSlider)
        {
            lightenCanvasView.InvalidateSurface();
        }
        else
        {
            darkenCanvasView.InvalidateSurface();
        }
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Find largest size rectangle in canvas
        float scale = Math.Min((float)info.Width / bitmap.Width,
                               (float)info.Height / bitmap.Height);
        SKRect rect = SKRect.Create(scale * bitmap.Width, scale * bitmap.Height);
        float x = (info.Width - rect.Width) / 2;
        float y = (info.Height - rect.Height) / 2;
        rect.Offset(x, y);

        // Display bitmap
        canvas.DrawBitmap(bitmap, rect);

        // Display gray rectangle with blend mode
        using (SKPaint paint = new SKPaint())
        {
            if ((SKCanvasView)sender == lightenCanvasView)
            {
                byte value = (byte)(255 * lightenSlider.Value);
                paint.Color = new SKColor(value, value, value);
                paint.BlendMode = SKBlendMode.Lighten;
            }
            else
            {
                byte value = (byte)(255 * (1 - darkenSlider.Value));
                paint.Color = new SKColor(value, value, value);
                paint.BlendMode = SKBlendMode.Darken;
            }

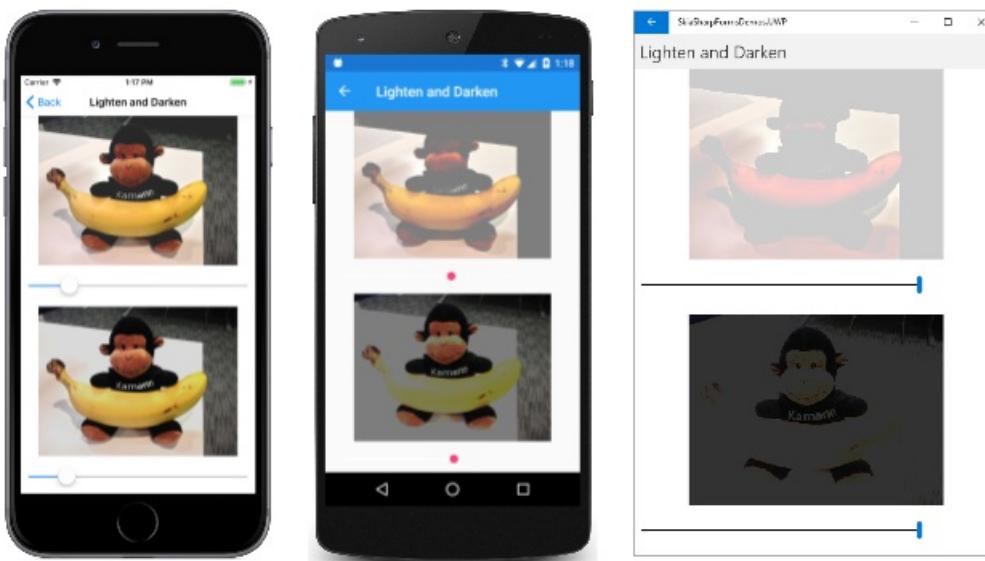
            canvas.DrawRect(rect, paint);
        }
    }
}

```

The `PaintSurface` handler calculates a rectangle suitable for the bitmap. The handler displays that bitmap and then displays a rectangle over the bitmap using an `SKPaint` object with its `BlendMode` property set to `SKBlendMode.Lighten` or `SKBlendMode.Darken`. The `Color` property is a gray shade based on the `slider`. For the

`Lighten` mode, the color ranges from black to white, but for the `Darken` mode it ranges from white to black.

The screenshots from left to right show increasingly larger `Slider` values as the top image gets lighter and the bottom image gets darker:



This program demonstrates the normal way in which the separable blend modes are used: The destination is an image of some sort, very often a bitmap. The source is a rectangle displayed using an `SKPaint` object with its `BlendMode` property set to a separable blend mode. The rectangle can be a solid color (as it is here) or a gradient. Transparency is *not* generally used with the separable blend modes.

As you experiment with this program, you'll discover that these two blend modes do not lighten and darken the image uniformly. Instead, the `Slider` seems to set a threshold of some sort. For example, as you increase the `Slider` for the `Lighten` mode, the darker areas of the image get light first while the lighter areas remain the same.

For the `Lighten` mode, if the destination pixel is the RGB color value (D_r, D_g, D_b), and the source pixel is the color (S_r, S_g, S_b), then the output is (O_r, O_g, O_b) calculated as follows:

$$O_r = \max(D_r, S_r) \quad O_g = \max(D_g, S_g) \quad O_b = \max(D_b, S_b)$$

For red, green, and blue separately, the result is the greater of the destination and source. This produces the effect of lightening the dark areas of the destination first.

The `Darken` mode is similar except that the result is the lesser of the destination and source:

$$O_r = \min(D_r, S_r) \quad O_g = \min(D_g, S_g) \quad O_b = \min(D_b, S_b)$$

The red, green, and blue components are each handled separately, which is why these blend modes are referred to as the *separable* blend modes. For this reason, the abbreviations `Dc` and `Sc` can be used for the destination and source colors, and it's understood that calculations apply to each of the red, green, and blue components separately.

The following table shows all the separable blend modes with brief explanations of what they do. The second column shows the source color that produces no change:

BLEND MODE	NO CHANGE	OPERATION
<code>Plus</code>	Black	Lightens by adding colors: $Sc + Dc$
<code>Modulate</code>	White	Darkens by multiplying colors: $Sc \cdot Dc$

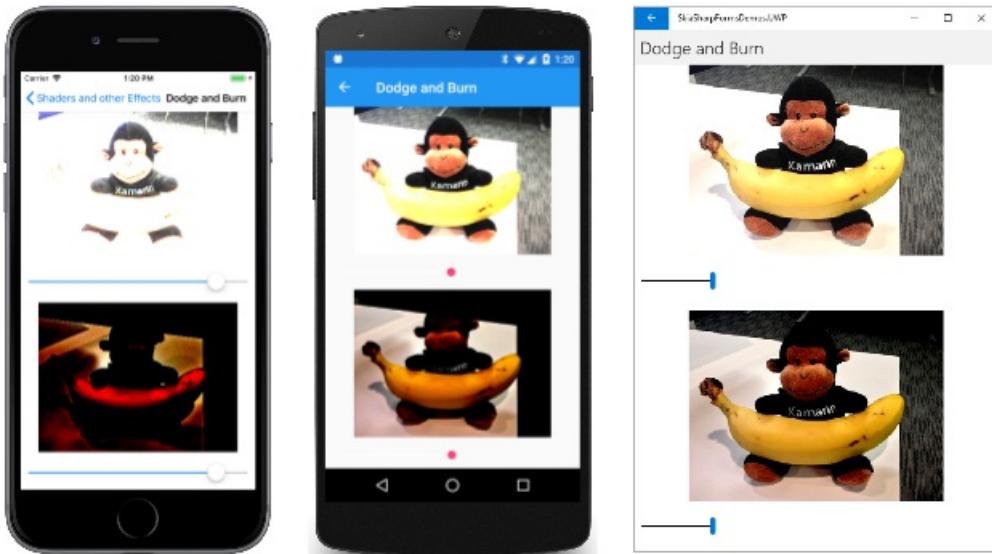
BLEND MODE	NO CHANGE	OPERATION
Screen	Black	Complements product of complements: $Sc + Dc - Sc \cdot Dc$
Overlay	Gray	Inverse of HardLight
Darken	White	Minimum of colors: $\min(Sc, Dc)$
Lighten	Black	Maximum of colors: $\max(Sc, Dc)$
ColorDodge	Black	Brightens destination based on source
ColorBurn	White	Darkens destination based on source
HardLight	Gray	Similar to effect of harsh spotlight
SoftLight	Gray	Similar to effect of soft spotlight
Difference	Black	Subtracts the darker from the lighter: $\text{Abs}(Dc - Sc)$
Exclusion	Black	Similar to Difference but lower contrast
Multiply	White	Darkens by multiplying colors: $Sc \cdot Dc$

More detailed algorithms can be found in the W3C [Compositing and Blending Level 1](#) specification and the Skia [SkBlendMode Reference](#), although the notation in these two sources is not the same. Keep in mind that Plus is commonly regarded as a Porter-Duff blend mode, and Modulate is not part of the W3C specification.

If the source is transparent, then for all the separable blend modes except Modulate, the blend mode has no effect. As you've seen earlier, the Modulate blend mode incorporates the alpha channel in the multiplication. Otherwise, Modulate has the same effect as Multiply.

Notice the two modes named ColorDodge and ColorBurn. The words *dodge* and *burn* originated in photographic darkroom practices. An enlarger makes a photographic print by shining light through a negative. With no light, the print is white. The print gets darker as more light falls on the print for a longer period of time. Print-makers often used a hand or small object to block some of the light from falling on a certain part of the print, making that area lighter. This is known as *dodging*. Conversely, opaque material with a hole in it (or hands blocking most of the light) could be used to direct more light in a particular spot to darken it, called *burning*.

The Dodge and Burn program is very similar to Lighten and Darken. The XAML file is structured the same but with different element names, and the code-behind file is likewise quite similar, but the effect of these two blend modes is quite different:



For small `Slider` values, the `Lighten` mode lightens dark areas first, while `ColorDodge` lightens more uniformly.

Image-processing application programs often allow dodging and burning to be restricted to specific areas, just like in a darkroom. This can be accomplished by gradients, or by a bitmap with varying shades of gray.

Exploring the separable blend modes

The **Separable Blend Modes** page allows you to examine all the separable blend modes. It displays a bitmap destination and a colored rectangle source using one of the blend modes.

The XAML file defines a `Picker` (to select the blend mode) and four sliders. The first three sliders let you set the red, green, and blue components of the source. The fourth slider is intended to override those values by setting a gray shade. The individual sliders are not identified, but colors indicate their function:

```
<?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:skia="clr-namespace:SkiaSharp;assembly=SkiaSharp"
    xmlns:skiviews="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Effects.SeparableBlendModesPage"
    Title="Separable Blend Modes">

    <StackLayout>
        <skiviews:SKCanvasView x:Name="canvasView"
            VerticalOptions="FillAndExpand"
            PaintSurface="OnCanvasViewPaintSurface" />

        <Picker x:Name="blendModePicker"
            Title="Blend Mode"
            Margin="10, 0"
            SelectedIndexChanged="OnPickerSelectedIndexChanged">
            <Picker.ItemsSource>
                <x:Array Type="{x:Type skia:SKBlendMode}">
                    <x:Static Member="skia:SKBlendMode.Plus" />
                    <x:Static Member="skia:SKBlendMode.Modulate" />
                    <x:Static Member="skia:SKBlendMode.Screen" />
                    <x:Static Member="skia:SKBlendMode.Overlay" />
                    <x:Static Member="skia:SKBlendMode.Darken" />
                    <x:Static Member="skia:SKBlendMode.Lighten" />
                    <x:Static Member="skia:SKBlendMode.ColorDodge" />
                    <x:Static Member="skia:SKBlendMode.ColorBurn" />
                    <x:Static Member="skia:SKBlendMode.HardLight" />
                    <x:Static Member="skia:SKBlendMode.SoftLight" />
                    <x:Static Member="skia:SKBlendMode.Difference" />
                </x:Array>
            </Picker.ItemsSource>
        </Picker>
    </StackLayout>

```

```

        <x:Static Member="skia:SKBlendMode.Exclusion" />
        <x:Static Member="skia:SKBlendMode.Multiply" />
    </x:Array>
</Picker.ItemsSource>

<Picker.SelectedIndex>
    0
</Picker.SelectedIndex>
</Picker>

<Slider x:Name="redSlider"
        MinimumTrackColor="Red"
        MaximumTrackColor="Red"
        Margin="10, 0"
        ValueChanged="OnSliderValueChanged" />

<Slider x:Name="greenSlider"
        MinimumTrackColor="Green"
        MaximumTrackColor="Green"
        Margin="10, 0"
        ValueChanged="OnSliderValueChanged" />

<Slider x:Name="blueSlider"
        MinimumTrackColor="Blue"
        MaximumTrackColor="Blue"
        Margin="10, 0"
        ValueChanged="OnSliderValueChanged" />

<Slider x:Name="graySlider"
        MinimumTrackColor="Gray"
        MaximumTrackColor="Gray"
        Margin="10, 0"
        ValueChanged="OnSliderValueChanged" />

<Label x:Name="colorLabel"
       HorizontalTextAlignment="Center" />

</StackLayout>
</ContentPage>

```

The code-behind file loads one of the bitmap resources and draws it twice, once in the top half of the canvas and again in the bottom half of the canvas:

```

public partial class SeparableBlendModesPage : ContentPage
{
    SKBitmap bitmap = BitmapExtensions.LoadBitmapResource(
        typeof(SeparableBlendModesPage),
        "SkiaSharpFormsDemos.Media.Banana.jpg");

    public SeparableBlendModesPage()
    {
        InitializeComponent();
    }

    void OnPickerSelectedIndexChanged(object sender, EventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs e)
    {
        if (sender == graySlider)
        {
            redSlider.Value = greenSlider.Value = blueSlider.Value = graySlider.Value;
        }

        colorLabel.Text = String.Format("Color = {0:X2} {1:X2} {2:X2}",
            (byte)(255 * redSlider.Value),
            (byte)(255 * greenSlider.Value),
            (byte)(255 * blueSlider.Value));

        canvasView.InvalidateSurface();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Draw bitmap in top half
        SKRect rect = new SKRect(0, 0, info.Width, info.Height / 2);
        canvas.DrawBitmap(bitmap, rect, BitmapStretch.Uniform);

        // Draw bitmap in bottom halr
        rect = new SKRect(0, info.Height / 2, info.Width, info.Height);
        canvas.DrawBitmap(bitmap, rect, BitmapStretch.Uniform);

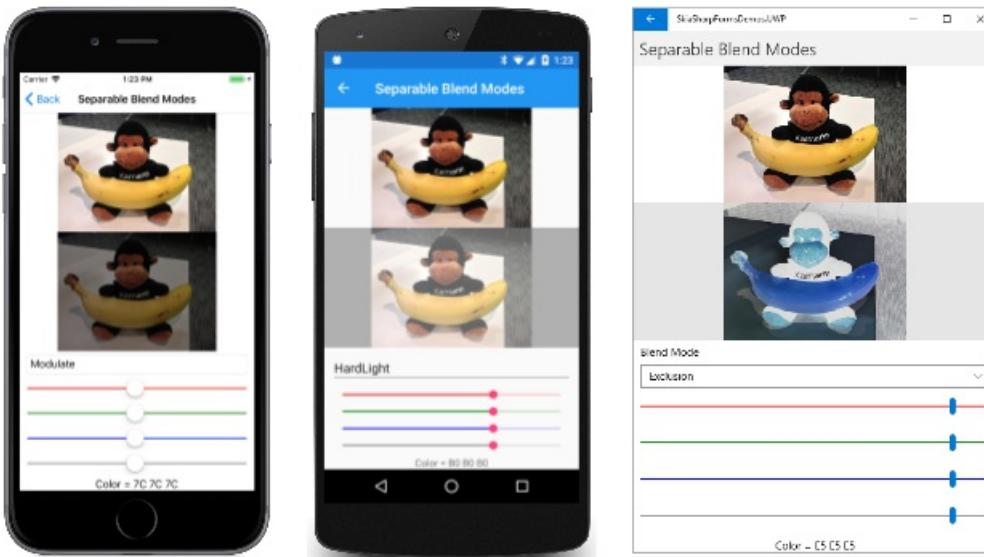
        // Get values from XAML controls
        SKBlendMode blendMode =
            (SKBlendMode)(blendModePicker.SelectedIndex == -1 ?
                0 : blendModePicker.SelectedItem);

        SKColor color = new SKColor((byte)(255 * redSlider.Value),
            (byte)(255 * greenSlider.Value),
            (byte)(255 * blueSlider.Value));

        // Draw rectangle with blend mode in bottom half
        using (SKPaint paint = new SKPaint())
        {
            paint.Color = color;
            paint.BlendMode = blendMode;
            canvas.DrawRect(rect, paint);
        }
    }
}

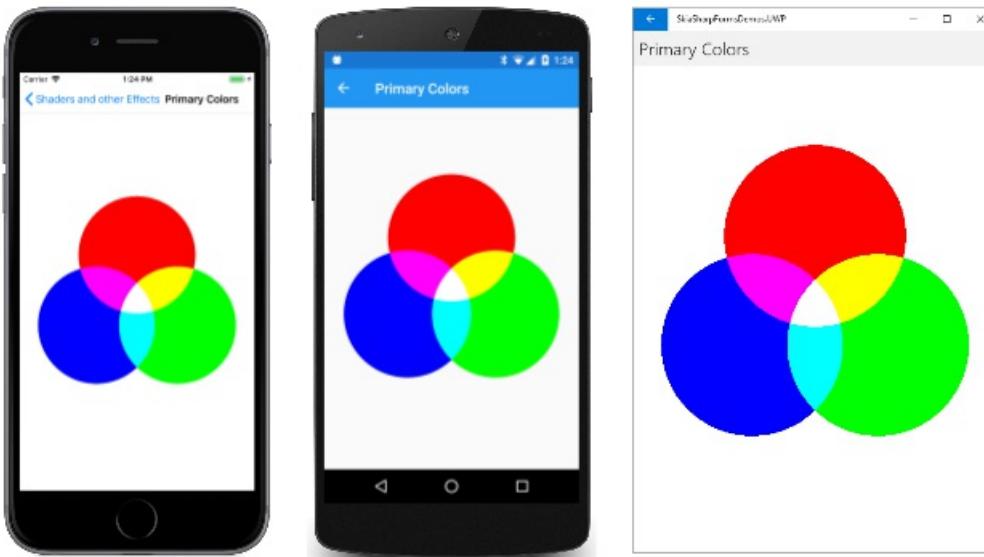
```

Towards the bottom of the `PaintSurface` handler, a rectangle is drawn over the second bitmap with the selected blend mode and the selected color. You can compare the modified bitmap at the bottom with the original bitmap at the top:



Additive and subtractive primary colors

The `Primary Colors` page draws three overlapping circles of red, green, and blue:



These are the additive primary colors. Combinations of any two produce cyan, magenta, and yellow, and a combination of all three is white.

These three circles are drawn with the `SKBlendMode.Plus` mode, but you can also use `Screen`, `Lighten`, or `Difference` for the same effect. Here's the program:

```
public class PrimaryColorsPage : ContentPage
{
    bool isSubtractive;

    public PrimaryColorsPage ()
    {
        Title = "Primary Colors";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
```

```

// SWITCN BETWEEN ADDITIVE AND SUBTRACTIVE PRIMARIES AT TAP
TapGestureRecognizer tap = new TapGestureRecognizer();
tap.Tapped += (sender, args) =>
{
    isSubtractive ^= true;
    canvasView.InvalidateSurface();
};

canvasView.GestureRecognizers.Add(tap);

Content = canvasView;
}

void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKImageInfo info = args.Info;
    SKSurface surface = args.Surface;
    SKCanvas canvas = surface.Canvas;

    canvas.Clear();

    SKPoint center = new SKPoint(info.Rect.MidX, info.Rect.MidY);
    float radius = Math.Min(info.Width, info.Height) / 4;
    float distance = 0.8f * radius;      // FROM CANVAS CENTER TO CIRCLE CENTER
    SKPoint center1 = center +
        new SKPoint(distance * (float)Math.Cos(9 * Math.PI / 6),
                    distance * (float)Math.Sin(9 * Math.PI / 6));
    SKPoint center2 = center +
        new SKPoint(distance * (float)Math.Cos(1 * Math.PI / 6),
                    distance * (float)Math.Sin(1 * Math.PI / 6));
    SKPoint center3 = center +
        new SKPoint(distance * (float)Math.Cos(5 * Math.PI / 6),
                    distance * (float)Math.Sin(5 * Math.PI / 6));

    using (SKPaint paint = new SKPaint())
    {
        if (!isSubtractive)
        {
            paint.BlendMode = SKBlendMode.Plus;
            System.Diagnostics.Debug.WriteLine(paint.BlendMode);

            paint.Color = SKColors.Red;
            canvas.DrawCircle(center1, radius, paint);

            paint.Color = SKColors.Lime;    // == (00, FF, 00)
            canvas.DrawCircle(center2, radius, paint);

            paint.Color = SKColors.Blue;
            canvas.DrawCircle(center3, radius, paint);
        }
        else
        {
            paint.BlendMode = SKBlendMode.Multiply
            System.Diagnostics.Debug.WriteLine(paint.BlendMode);

            paint.Color = SKColors.Cyan;
            canvas.DrawCircle(center1, radius, paint);

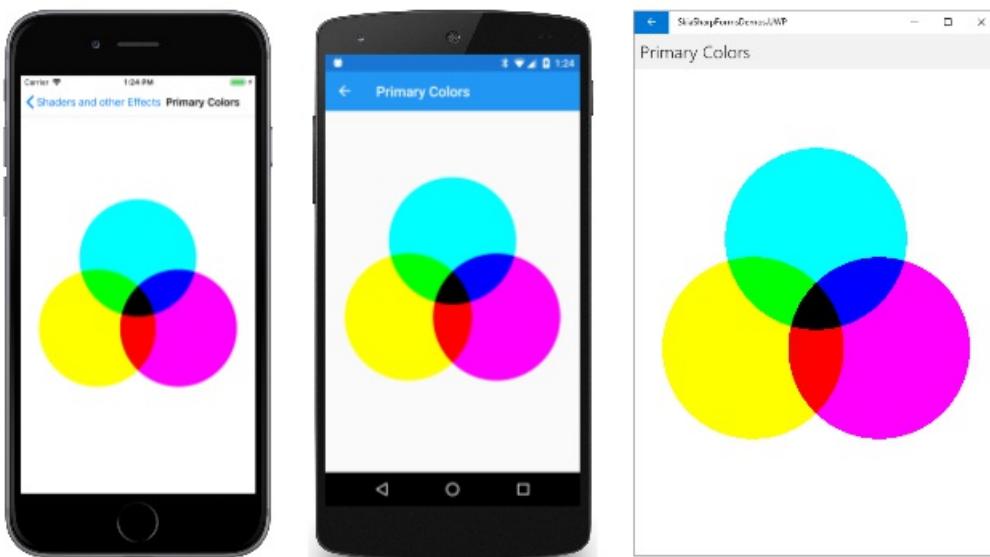
            paint.Color = SKColors.Magenta;
            canvas.DrawCircle(center2, radius, paint);

            paint.Color = SKColors.Yellow;
            canvas.DrawCircle(center3, radius, paint);
        }
    }
}
}

```

The program includes a `TapGestureRecognizer`. When you tap or click the screen, the program uses

`SKBlendMode.Multiply` to display the three subtractive primaries:



The `Darken` mode also works for this same effect.

Related links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

The non-separable blend modes

3/5/2021 • 9 minutes to read • [Edit Online](#)



[Download the sample](#)

As you saw in the article [SkiaSharp separable blend modes](#), the separable blend modes perform operations on the red, green, and blue channels separately. The non-separable blend modes do not. By operating upon the Hue, Saturation, and Luminosity levels of color, the non-separable blend modes can alter colors in interesting ways:



The Hue-Saturation-Luminosity model

To understand the non-separable blend modes, it is necessary to treat the destination and source pixels as colors in the Hue-Saturation-Luminosity model. (Luminosity is also referred to as Lightness.)

The HSL color model was discussed in the article [Integrating with Xamarin.Forms](#) and a sample program in that article allows experimentation with HSL colors. You can create an `SKColor` value using Hue, Saturation, and Luminosity values with the static `SKColor.FromHsl` method.

The Hue represents the dominant wavelength of the color. Hue values range from 0 to 360 and cycle through the additive and subtractive primaries: Red is the value 0, yellow is 60, green is 120, cyan is 180, blue is 240, magenta is 300, and the cycle goes back to red at 360.

If there is no dominant color — for example, the color is white or black or a gray shade — then the Hue is undefined and usually set to 0.

The Saturation values can range from 0 to 100 and indicate the purity of the color. A Saturation value of 100 is the purest color while values lower than 100 cause the color to become more grayish. A Saturation value of 0 results in a shade of gray.

The Luminosity (or Lightness) value indicates how bright the color is. A Luminosity value of 0 is black regardless of the other settings. Similarly, a Luminosity value of 100 is white.

The HSL value (0, 100, 50) is the RGB value (FF, 00, 00), which is pure red. The HSL value (180, 100, 50) is the RGB value (00, FF, FF), pure cyan. As the Saturation is decreased, the dominant color component is decreased and the other components are increased. At a Saturation level of 0, all the components are the same and the color is a gray shade. Decrease the Luminosity to go to black; increase the Luminosity to go to white.

The blend modes in detail

Like the other blend modes, the four non-separable blend modes involve a destination (which is often a bitmap image) and a source, which is often a single color or a gradient. The blend modes combine Hue, Saturation, and

Luminosity values from the destination and the source:

BLEND MODE	COMPONENTS FROM SOURCE	COMPONENTS FROM DESTINATION
Hue	Hue	Saturation and Luminosity
Saturation	Saturation	Hue and Luminosity
Color	Hue and Saturation	Luminosity
Luminosity	Luminosity	Hue and Saturation

See the W3C [Compositing and Blending Level 1](#) specification for the algorithms.

The Non-Separable Blend Modes page contains a [Picker](#) to select one of these blend modes and three [Slider](#) views to select an HSL color:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp;assembly=SkiaSharp"
    xmlns:skiaviews="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Effects.NonSeparableBlendModesPage"
    Title="Non-Separable Blend Modes">

    <StackLayout>
        <skiaviews:SKCanvasView x:Name="canvasView"
            VerticalOptions="FillAndExpand"
            PaintSurface="OnCanvasViewPaintSurface" />

        <Picker x:Name="blendModePicker"
            Title="Blend Mode"
            Margin="10, 0"
            SelectedIndexChanged="OnPickerSelectedIndexChanged">
            <Picker.ItemsSource>
                <x:Array Type="{x:Type skia:SKBlendMode}">
                    <x:Static Member="skia:SKBlendMode.Hue" />
                    <x:Static Member="skia:SKBlendMode.Saturation" />
                    <x:Static Member="skia:SKBlendMode.Color" />
                    <x:Static Member="skia:SKBlendMode.Luminosity" />
                </x:Array>
            </Picker.ItemsSource>
            <Picker.SelectedIndex>
                0
            </Picker.SelectedIndex>
        </Picker>

        <Slider x:Name="hueSlider"
            Maximum="360"
            Margin="10, 0"
            ValueChanged="OnSliderValueChanged" />

        <Slider x:Name="satSlider"
            Maximum="100"
            Margin="10, 0"
            ValueChanged="OnSliderValueChanged" />

        <Slider x:Name="lumSlider"
            Maximum="100"
            Margin="10, 0"
            ValueChanged="OnSliderValueChanged" />

        <StackLayout Orientation="Horizontal">
            <Label x:Name="hslLabel"
                HorizontalOptions="CenterAndExpand" />

            <Label x:Name="rgbLabel"
                HorizontalOptions="CenterAndExpand" />
        </StackLayout>
    </StackLayout>
</ContentPage>

```

To save space, the three `slider` views are not identified in the user interface of the program. You'll need to remember that the order is Hue, Saturation, and Luminosity. Two `Label` views at the bottom of the page show the HSL and RGB color values.

The code-behind file loads one of the bitmap resources, displays that as large as possible on the canvas, and then covers the canvas with a rectangle. The rectangle color is based on the three `Slider` views and the blend mode is the one selected in the `Picker`:

```

public partial class NonSeparableBlendModesPage : ContentPage
{
    SKBitmap bitmap = BitmapExtensions.LoadBitmapResource(
        typeof(NonSeparableBlendModesPage),
        "SkiaSharpFormsDemos.Media.Banana.jpg");
    SKColor color;

    public NonSeparableBlendModesPage()
    {
        InitializeComponent();
    }

    void OnPickerSelectedIndexChanged(object sender, EventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs e)
    {
        // Calculate new color based on sliders
        color = SKColor.FromHsl((float)hueSlider.Value,
                               (float)satSlider.Value,
                               (float)lumSlider.Value);

        // Use labels to display HSL and RGB color values
        color.ToHsl(out float hue, out float sat, out float lum);

        hslLabel.Text = String.Format("HSL = {0:F0} {1:F0} {2:F0}",
                                      hue, sat, lum);

        rgbLabel.Text = String.Format("RGB = {0:X2} {1:X2} {2:X2}",
                                     color.Red, color.Green, color.Blue);

        canvasView.InvalidateSurface();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();
        canvas.DrawBitmap(bitmap, info.Rect, BitmapStretch.Uniform);

        // Get blend mode from Picker
        SKBlendMode blendMode =
            (SKBlendMode)(blendModePicker.SelectedIndex == -1 ?
                         0 : blendModePicker.SelectedItem);

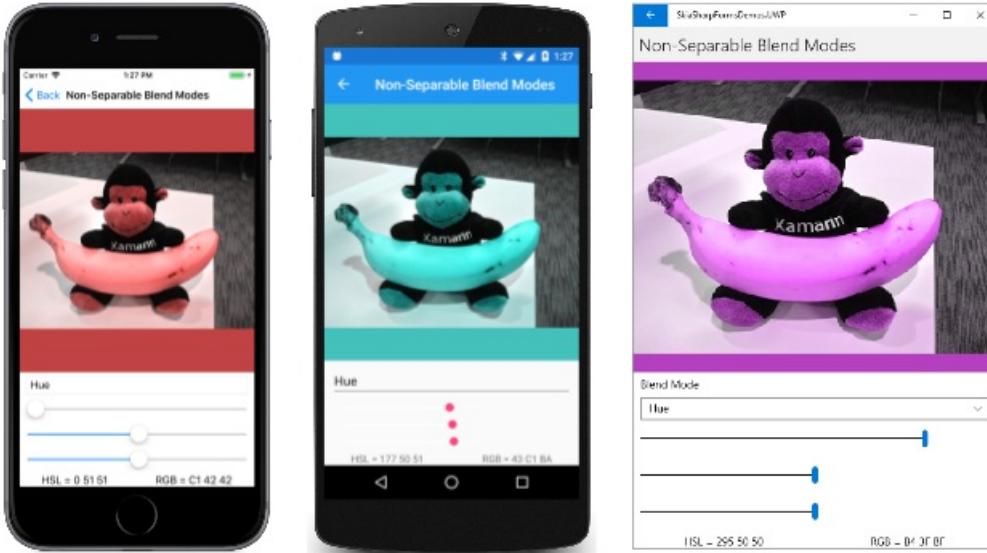
        using (SKPaint paint = new SKPaint())
        {
            paint.Color = color;
            paint.BlendMode = blendMode;
            canvas.DrawRect(info.Rect, paint);
        }
    }
}

```

Notice that the program does not display the HSL color value as selected by the three sliders. Instead, it creates a color value from those sliders and then uses the `ToHsl` method to obtain the Hue, Saturation, and Luminosity values. This is because the `FromHsl` method converts an HSL color to an RGB color, which is stored internally in the `SKColor` structure. The `ToHsl` method converts from RGB to HSL, but the result will not always be the original value.

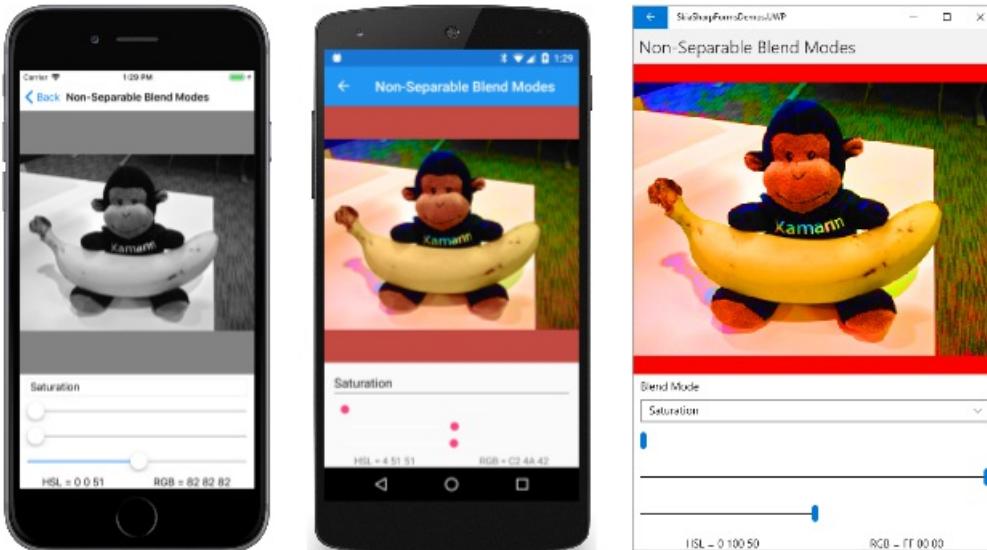
For example, `FromHsl` converts the HSL value (180, 50, 0) to the RGB color (0, 0, 0) because the `Luminosity` is zero. The `ToHsl` method converts the RGB color (0, 0, 0) to the HSL color (0, 0, 0) because the Hue and Saturation values are irrelevant. When using this program, it is better that you see the representation of the HSL color that the program is using rather than the one you specified with the sliders.

The `SKBlendModes.Hue` blend mode uses the Hue level of the source while retaining the Saturation and Luminosity levels of the destination. When you test this blend mode, the saturation and luminosity sliders must be set to something other than 0 or 100 because in those cases, the Hue is not uniquely defined.



When you use set the slider to 0 (as with the iOS screenshot at the left), everything turns reddish. But this does not mean that the image is entirely absent of green and blue. Obviously there are still gray shades present in the result. For example, the RGB color (40, 40, C0) is equivalent to the HSL color (240, 50, 50). The Hue is blue, but the saturation value of 50 indicates that there are red and green components as well. If the Hue is set to 0 with `SKBlendModes.Hue`, the HSL color is (0, 50, 50), which is the RGB color (C0, 40, 40). There are still blue and green components but now the dominant component is red.

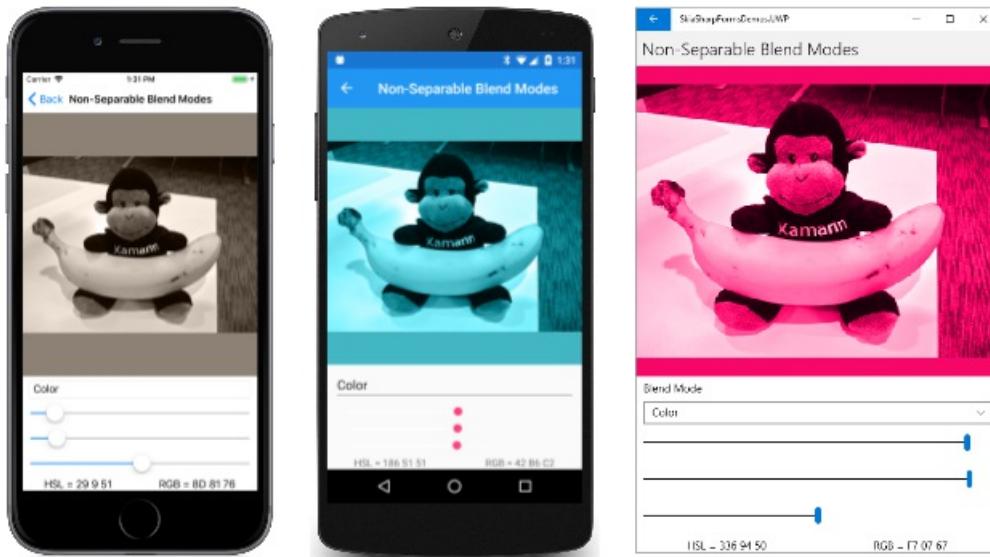
The `SKBlendModes.Saturation` blend mode combines the Saturation level of the source with the Hue and Luminosity of the destination. Like the Hue, the Saturation is not well defined if the Luminosity is 0 or 100. In theory, any Luminosity setting between those two extremes should work. However, the Luminosity setting seems to affect the result more than it should. Set the luminosity to 50, and you can see how you can set the Saturation level of the picture:



You can use this blend mode to increase the color Saturation of a dull image, or you can decrease the Saturation

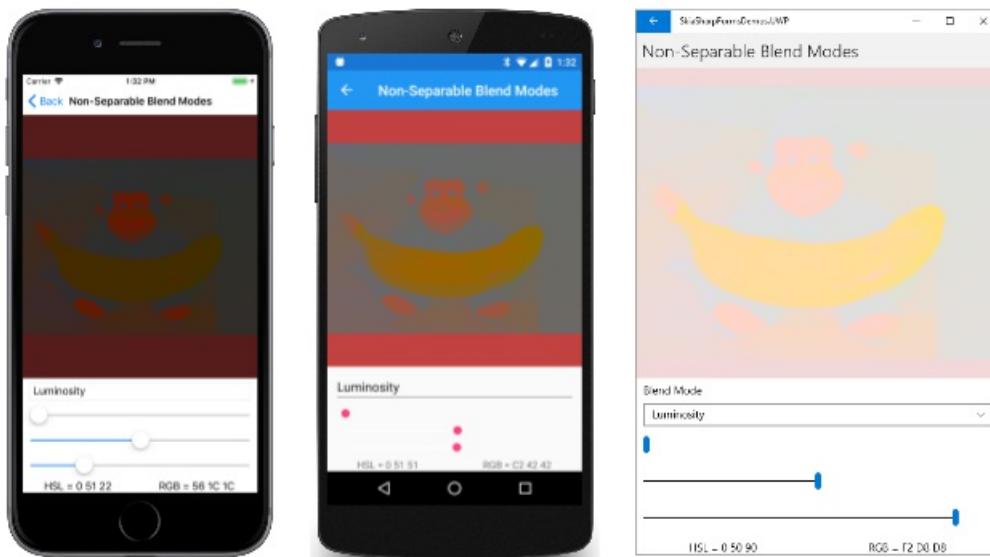
down to zero (as in the iOS screenshot at the left) for a resultant image composed of only gray shades.

The `SKBlendModes.Color` blend mode retains the Luminosity of the destination but uses the Hue and Saturation of the source. Again, that implies that any setting of the Luminosity slider somewhere between the extremes should work.



You'll see an application of this blend mode shortly.

Finally, the `SKBlendModes.Luminosity` blend mode is the opposite of `SKBlendModes.Color`. It retains the Hue and Saturation of the destination but uses the Luminosity of the source. The `Luminosity` blend mode is the most mysterious of the batch: The Hue and Saturation sliders affect the image, but even at medium Luminosity, the image is not distinct:



In theory, increasing or decreasing the Luminosity of an image should make it lighter or darker. Interestingly, the [Luminosity example in the Skia SkBlendMode Reference](#) is quite similar.

It is generally not the case that you'll want to use one of the non-separable blend modes with a source that consists of a single color applied to the entire destination image. The effect is just too great. You'll want to restrict the effect to one part of the image. In that case, the source will probably incorporate transparency, or perhaps the source will be limited to a smaller graphic.

A matte for a separable mode

Here's one of the bitmaps included as a resource in the [SkiaSharpFormsDemos](#) sample. The filename is

Banana.jpg:



It's possible to create a matte that encompasses just the banana. This is also a resource in the [SkiaSharpFormsDemos](#) sample. The filename is **BananaMatte.png**:



Aside from the black banana shape, the rest of the bitmap is transparent.

The [Blue Banana](#) page uses that matte to alter the Hue and Saturation of the banana that the monkey is holding, but to change nothing else in the image.

In the following `BlueBananaPage` class, the **Banana.jpg** bitmap is loaded as a field. The constructor loads the **BananaMatte.png** bitmap as the `matteBitmap` object, but it does not retain that object beyond the constructor. Instead, a third bitmap named `blueBananaBitmap` is created. The `matteBitmap` is drawn on `blueBananaBitmap` followed by an `SKPaint` with its `Color` set to blue and its `BlendMode` set to `SKBlendMode.SrcIn`. The `blueBananaBitmap` remains mostly transparent but with a solid pure blue image of the banana:

```

public class BlueBananaPage : ContentPage
{
    SKBitmap bitmap = BitmapExtensions.LoadBitmapResource(
        typeof(BlueBananaPage),
        "SkiaSharpFormsDemos.Media.Banana.jpg");

    SKBitmap blueBananaBitmap;

    public BlueBananaPage()
    {
        Title = "Blue Banana";

        // Load banana matte bitmap (black on transparent)
        SKBitmap matteBitmap = BitmapExtensions.LoadBitmapResource(
            typeof(BlueBananaPage),
            "SkiaSharpFormsDemos.Media.BananaMatte.png");

        // Create a bitmap with a solid blue banana and transparent otherwise
        blueBananaBitmap = new SKBitmap(matteBitmap.Width, matteBitmap.Height);

        using (SKCanvas canvas = new SKCanvas(blueBananaBitmap))
        {
            canvas.Clear();
            canvas.DrawBitmap(matteBitmap, new SKPoint(0, 0));

            using (SKPaint paint = new SKPaint())
            {
                paint.Color = SKColors.Blue;
                paint.BlendMode = SKBlendMode.SrcIn;
                canvas.DrawPaint(paint);
            }
        }

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

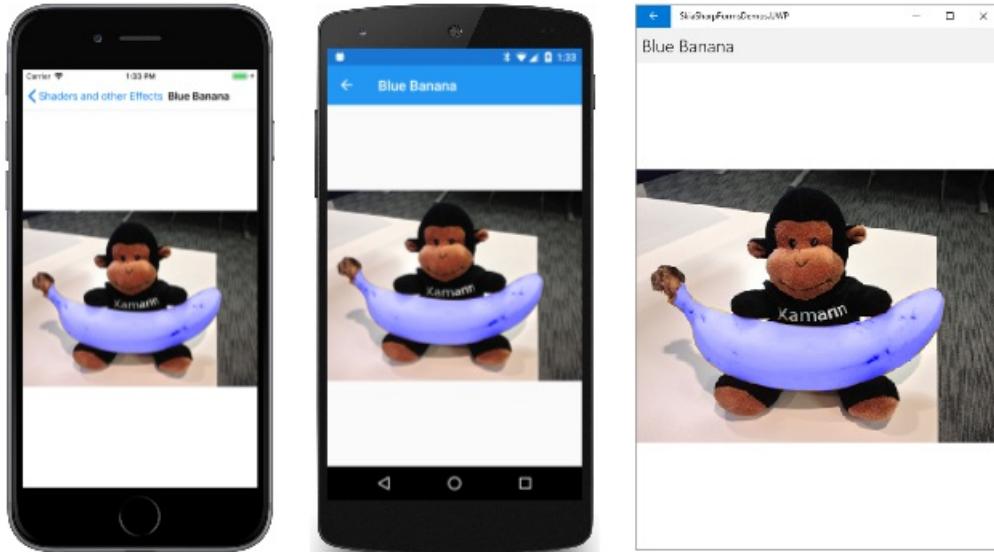
        canvas.Clear();

        canvas.DrawBitmap(bitmap, info.Rect, BitmapStretch.Uniform);

        using (SKPaint paint = new SKPaint())
        {
            paint.BlendMode = SKBlendMode.Color;
            canvas.DrawBitmap(blueBananaBitmap,
                info.Rect,
                BitmapStretch.Uniform,
                paint: paint);
        }
    }
}

```

The `PaintSurface` handler draws the bitmap with the monkey holding the banana. This code is followed by the display of `blueBananaBitmap` with `SKBlendMode.Color`. Over the surface of the banana, each pixel's Hue and Saturation is replaced by the solid blue, which corresponds to a hue value of 240 and a saturation value of 100. The Luminosity, however, remains the same, which means the banana continues to have a realistic texture despite its new color:



Try changing the blend mode to `SKBlendMode.Saturation`. The banana remains yellow but it's a more intense yellow. In a real-life application, this might be a more desirable effect than turning the banana blue.

Related links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

SkiaSharp mask filters

3/5/2021 • 5 minutes to read • [Edit Online](#)

 [Download the sample](#)

Mask filters are effects that manipulate the geometry and alpha channel of graphical objects. To use a mask filter, set the `MaskFilter` property of `SKPaint` to an object of type `SKMaskFilter` that you've created by calling one of the `SKMaskFilter` static methods.

The best way to become familiar with mask filters is by experimenting with these static methods. The most useful mask filter creates a blur:

Blur My Text

That's the only mask filter feature described in this article. The next article on [SkiaSharp image filters](#) also describes a blur effect that you might prefer to this one.

The static `SKMaskFilter.CreateBlur` method has the following syntax:

```
public static SKMaskFilter CreateBlur (SKBlurStyle blurStyle, float sigma);
```

Overloads allow specifying flags for the algorithm used to create the blur, and a rectangle to avoid blurring in areas that will be covered with other graphical objects.

`SKBlurStyle` is an enumeration with the following members:

- `Normal`
- `Solid`
- `Outer`
- `Inner`

The effects of these styles are shown in the examples below. The `sigma` parameter specifies the extent of the blur. In older versions of Skia, the extent of the blur was indicated with a radius value. If a radius value is preferable for your application, there is a static `SKMaskFilter.ConvertRadiusToSigma` method that can convert from one to the other. The method multiplies the radius by 0.57735 and adds 0.5.

The [Mask Blur Experiment](#) page in the [SkiaSharpFormsDemos](#) sample allows you to experiment with the blur styles and sigma values. The XAML file instantiates a `Picker` with the four `SKBlurStyle` enumeration members and a `Slider` for specifying the sigma value:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp;assembly=SkiaSharp"
    xmlns:skiaforms="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Effects.MaskBlurExperimentPage"
    Title="Mask Blur Experiment">

    <StackLayout>
        <skiaforms:SKCanvasView x:Name="canvasView"
            VerticalOptions="FillAndExpand"
            PaintSurface="OnCanvasViewPaintSurface" />

        <Picker x:Name="blurStylePicker"
            Title="Filter Blur Style"
            Margin="10, 0"
            SelectedIndexChanged="OnPickerSelectedIndexChanged">
            <Picker.ItemsSource>
                <x:Array Type="{x:Type skia:SKBlurStyle}">
                    <x:Static Member="skia:SKBlurStyle.Normal" />
                    <x:Static Member="skia:SKBlurStyle.Solid" />
                    <x:Static Member="skia:SKBlurStyle.Outer" />
                    <x:Static Member="skia:SKBlurStyle.Inner" />
                </x:Array>
            </Picker.ItemsSource>
            <Picker.SelectedIndex>
                0
            </Picker.SelectedIndex>
        </Picker>

        <Slider x:Name="sigmaSlider"
            Maximum="10"
            Margin="10, 0"
            ValueChanged="OnSliderValueChanged" />

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

```

The code-behind file uses those values to create an `SKMaskFilter` object and set it to the `MaskFilter` property of an `SKPaint` object. This `SKPaint` object is used to draw both a text string and a bitmap:

```

public partial class MaskBlurExperimentPage : ContentPage
{
    const string TEXT = "Blur My Text";

    SKBitmap bitmap = BitmapExtensions.LoadBitmapResource(
        typeof(MaskBlurExperimentPage),
        "SkiaSharpFormsDemos.Media.SeatedMonkey.jpg");

    public MaskBlurExperimentPage ()
    {
        InitializeComponent ();
    }

    void OnPickerSelectedIndexChanged(object sender, EventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear(SKColors.Pink);

        // Get values from XAML controls
        SKBlurStyle blurStyle =
            (SKBlurStyle)(blurStylePicker.SelectedIndex == -1 ?
                0 : blurStylePicker.SelectedItem);

        float sigma = (float)sigmaSlider.Value;

        using (SKPaint paint = new SKPaint())
        {
            // Set SKPaint properties
            paint.TextSize = (info.Width - 100) / (TEXT.Length / 2);
            paint.MaskFilter = SKMaskFilter.CreateBlur(blurStyle, sigma);

            // Get text bounds and calculate display rectangle
            SKRect textBounds = new SKRect();
            paint.MeasureText(TEXT, ref textBounds);
            SKRect textRect = new SKRect(0, 0, info.Width, textBounds.Height + 50);

            // Center the text in the display rectangle
            float xText = textRect.Width / 2 - textBounds.MidX;
            float yText = textRect.Height / 2 - textBounds.MidY;

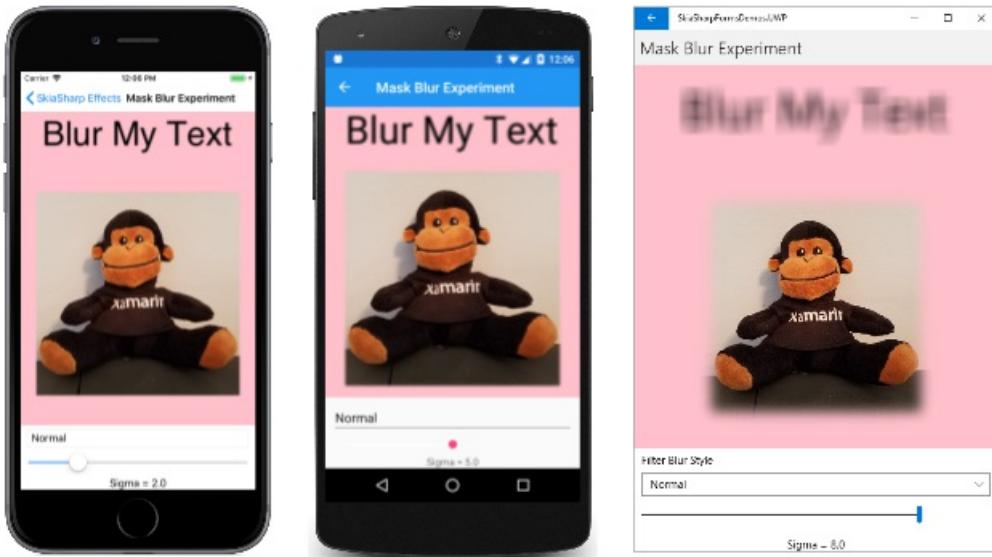
            canvas.DrawText(TEXT, xText, yText, paint);

            // Calculate rectangle for bitmap
            SKRect bitmapRect = new SKRect(0, textRect.Bottom, info.Width, info.Height);
            bitmapRect.Inflate(-50, -50);

            canvas.DrawBitmap(bitmap, bitmapRect, BitmapStretch.Uniform, paint: paint);
        }
    }
}

```

Here's the program running on iOS, Android, and the Universal Windows Platform (UWP) with the `Normal` blur style and increasing `sigma` levels:



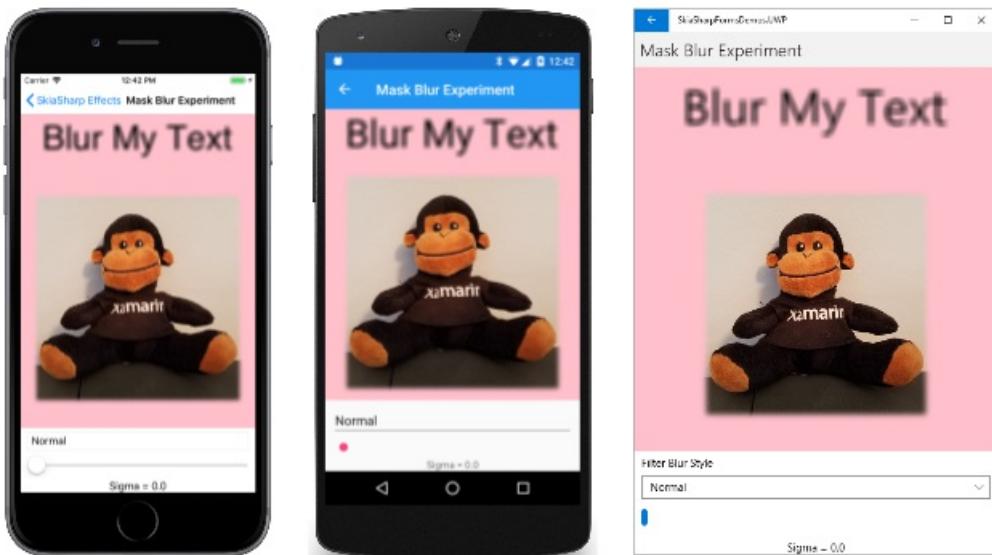
You'll notice that only the edges of the bitmap are affected by the blur. The `SKMaskFilter` class is not the correct effect to use if you want to blur an entire bitmap image. For that you'll want to use the `SKImageFilter` class as described in the next article on [SkiaSharp image filters](#).

The text is blurred more with increasing values of the `sigma` argument. In experimenting with this program, you'll notice that for a particular `sigma` value, the blur is more extreme on the Windows 10 desktop. This difference occurs because the pixel density is lower on a desktop monitor than on mobile devices, and hence the text height in pixels is lower. The `sigma` value is proportional to a blur extent in pixels, so for a given `sigma` value, the effect is more extreme on lower resolution displays. In a production application, you'll probably want to calculate a `sigma` value that is proportional to the size of the graphic.

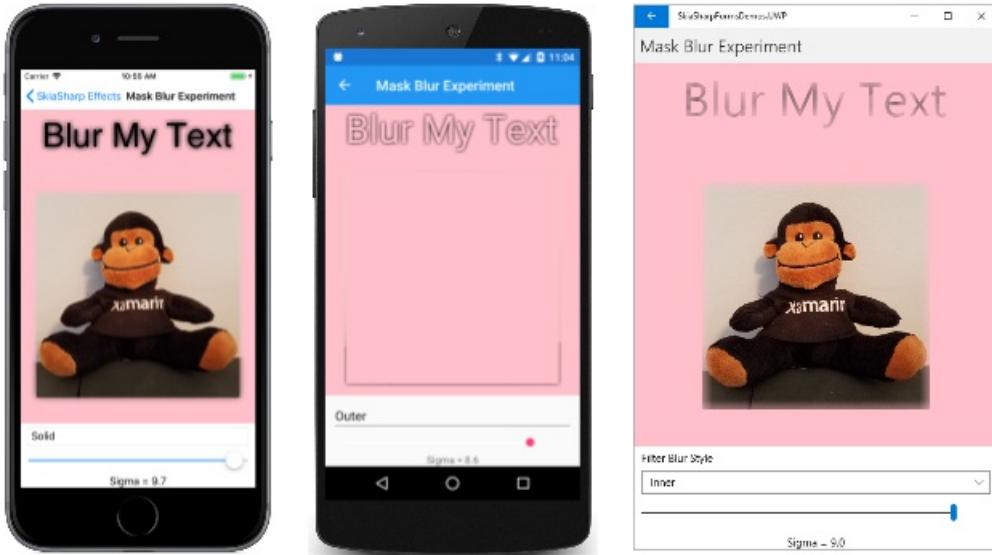
Try several values before settling on a blur level that looks the best for your application. For example, in the [Mask Blur Experiment](#) page, try setting `sigma` like this:

```
sigma = paint.TextSize / 18;
paint.MaskFilter = SKMaskFilter.CreateBlur(blurStyle, sigma);
```

Now the `Slider` has no effect, but the degree of blur is consistent among the platforms:



All the screenshots so far have shown blur created with the `SKBlurStyle.Normal` enumeration member. The following screenshots show the effects of the `Solid`, `Outer`, and `Inner` blur styles:

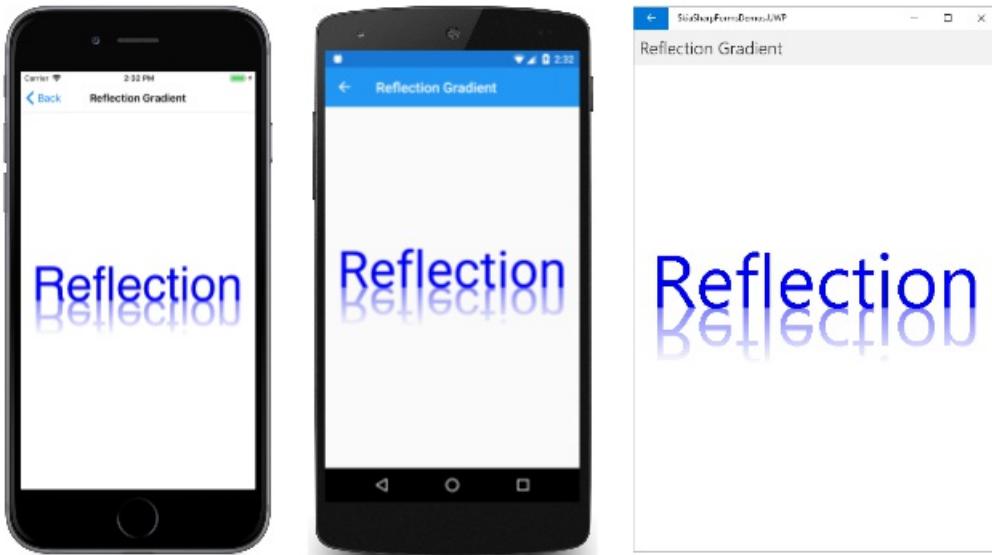


The iOS screenshot shows the `Solid` style: The text characters are still present as solid black strokes, and the blur is added to the outside of these text characters.

The Android screenshot in the middle shows the `Outer` style: The character strokes themselves are eliminated (as is the bitmap) and the blur surrounds the empty space where the text characters once appeared.

The UWP screenshot on the right shows the `Inner` style. The blur is restricted to the area normally occupied by the text characters.

The [SkiaSharp linear gradient](#) article described a **Reflection Gradient** program that used a linear gradient and a transform to mimic a reflection of a text string:



The **Blurry Reflection** page adds a single statement to that code:

```

public class BlurryReflectionPage : ContentPage
{
    const string TEXT = "Reflection";

    public BlurryReflectionPage()
    {
        Title = "Blurry Reflection";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        using (SKPaint paint = new SKPaint())
        {
            // Set text color to blue
            paint.Color = SKColors.Blue;

            // Set text size to fill 90% of width
            paint.TextSize = 100;
            float width = paint.MeasureText(TEXT);
            float scale = 0.9f * info.Width / width;
            paint.TextSize *= scale;

            // Get text bounds
            SKRect textBounds = new SKRect();
            paint.MeasureText(TEXT, ref textBounds);

            // Calculate offsets to position text above center
            float xText = info.Width / 2 - textBounds.MidX;
            float yText = info.Height / 2;

            // Draw unreflected text
            canvas.DrawText(TEXT, xText, yText, paint);

            // Shift textBounds to match displayed text
            textBounds.Offset(xText, yText);

            // Use those offsets to create a gradient for the reflected text
            paint.Shader = SKShader.CreateLinearGradient(
                new SKPoint(0, textBounds.Top),
                new SKPoint(0, textBounds.Bottom),
                new SKColor[] { paint.Color.WithAlpha(0),
                    paint.Color.WithAlpha(0x80) },
                null,
                SKShaderTileMode.Clamp);

            // Create a blur mask filter
            paint.MaskFilter = SKMaskFilter.CreateBlur(SKBlurStyle.Normal, paint.TextSize / 36);

            // Scale the canvas to flip upside-down around the vertical center
            canvas.Scale(1, -1, 0, yText);

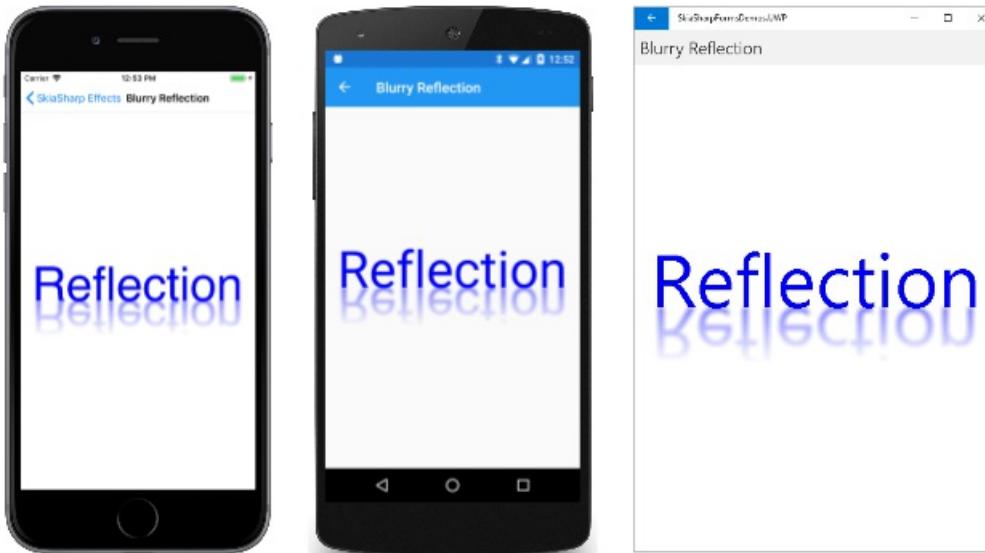
            // Draw reflected text
            canvas.DrawText(TEXT, xText, yText, paint);
        }
    }
}

```

The new statement adds a blur filter for the reflected text that is based on the text size:

```
paint.MaskFilter = SKMaskFilter.CreateBlur(SKBlurStyle.Normal, paint.TextSize / 36);
```

This blur filter causes the reflection to seem much more realistic:



Related links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

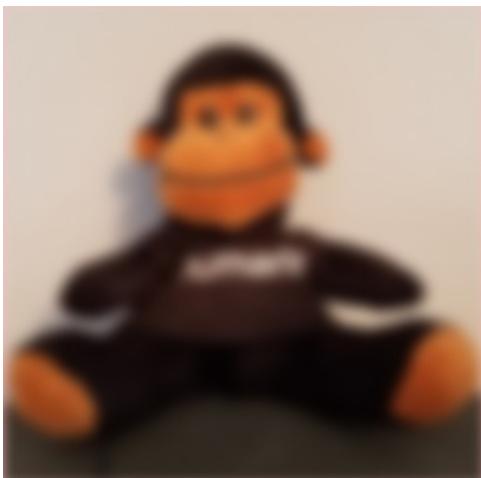
SkiaSharp image filters

3/5/2021 • 9 minutes to read • [Edit Online](#)

 [Download the sample](#)

Image filters are effects that operate on all the color bits of pixels that make up an image. They are more versatile than mask filters, which only operate on the alpha channel as described in the article [SkiaSharp mask filters](#). To use an image filter, set the `ImageFilter` property of `SKPaint` to an object of type `SKImageFilter` that you've created by calling one of the class's static methods.

The best way to become familiar with mask filters is by experimenting with these static methods. You can use a mask filter to blur an entire bitmap:



This article also demonstrates using an image filter to create a drop shadow, and for embossing and engraving effects.

Blurring vector graphics and bitmaps

The blur effect created by the `SKImageFilter.CreateBlur` static method has a significant advantage over the blur methods in the `SKMaskFilter` class: The image filter can blur an entire bitmap. The method has the following syntax:

```
public static SkiaSharp.SKImageFilter CreateBlur (float sigmaX, float sigmaY,  
                                                SKImageFilter input = null,  
                                                SKImageFilter.CropRect cropRect = null);
```

The method has two sigma values — the first for the blur extent in the horizontal direction and the second for the vertical direction. You can cascade image filters by specifying another image filter as the optional third argument. A cropping rectangle can also be specified.

The [Image Blur Experiment](#) page in the [SkiaSharpFormsDemos](#) includes two `Slider` views that let you experiment with setting various levels of blur:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Effects.ImageBlurExperimentPage"
    Title="Image Blur Experiment">

    <StackLayout>
        <skia:SKCanvasView x:Name="canvasView"
            VerticalOptions="FillAndExpand"
            PaintSurface="OnCanvasViewPaintSurface" />

        <Slider x:Name="sigmaXSlider"
            Maximum="10"
            Margin="10, 0"
            ValueChanged="OnSliderValueChanged" />

        <Label Text="{Binding Source={x:Reference sigmaXSlider},
            Path=Value,
            StringFormat='Sigma X = {0:F1}'}"
            HorizontalTextAlignment="Center" />

        <Slider x:Name="sigmaYSlider"
            Maximum="10"
            Margin="10, 0"
            ValueChanged="OnSliderValueChanged" />

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

```

The code-behind file uses the two `slider` values to call `SKImageFilter.CreateBlur` for the `SKPaint` object used to display both text and a bitmap:

```

public partial class ImageBlurExperimentPage : ContentPage
{
    const string TEXT = "Blur My Text";

    SKBitmap bitmap = BitmapExtensions.LoadBitmapResource(
        typeof(MaskBlurExperimentPage),
        "SkiaSharpFormsDemos.Media.SeatedMonkey.jpg");

    public ImageBlurExperimentPage ()
    {
        InitializeComponent ();
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear(SKColors.Pink);

        // Get values from sliders
        float sigmaX = (float)sigmaXSlider.Value;
        float sigmaY = (float)sigmaYSlider.Value;

        using (SKPaint paint = new SKPaint())
        {
            // Set SKPaint properties
            paint.TextSize = (info.Width - 100) / (TEXT.Length / 2);
            paint.ImageFilter = SKImageFilter.CreateBlur(sigmaX, sigmaY);

            // Get text bounds and calculate display rectangle
            SKRect textBounds = new SKRect();
            paint.MeasureText(TEXT, ref textBounds);
            SKRect textRect = new SKRect(0, 0, info.Width, textBounds.Height + 50);

            // Center the text in the display rectangle
            float xText = textRect.Width / 2 - textBounds.MidX;
            float yText = textRect.Height / 2 - textBounds.MidY;

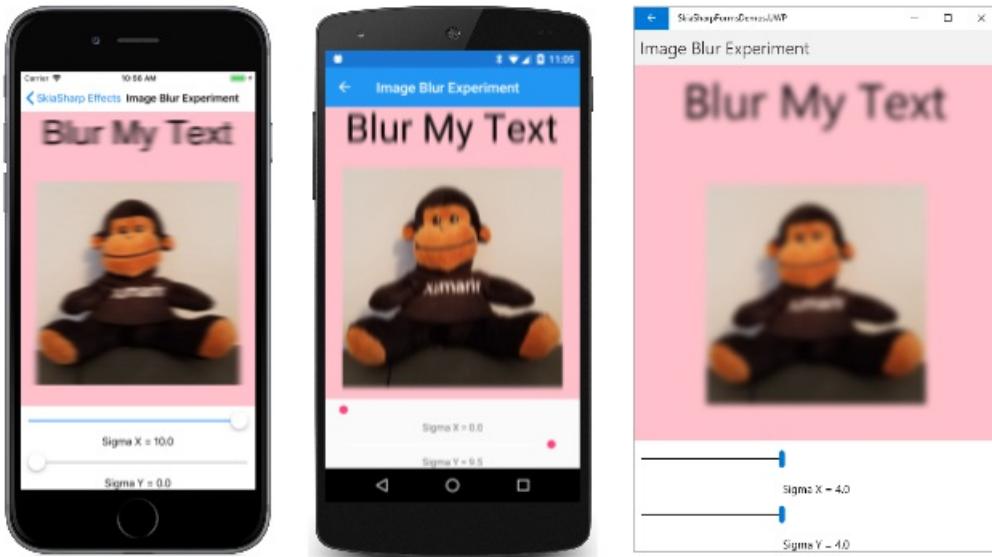
            canvas.DrawText(TEXT, xText, yText, paint);

            // Calculate rectangle for bitmap
            SKRect bitmapRect = new SKRect(0, textRect.Bottom, info.Width, info.Height);
            bitmapRect.Inflate(-50, -50);

            canvas.DrawBitmap(bitmap, bitmapRect, BitmapStretch.Uniform, paint: paint);
        }
    }
}

```

The three screenshots show various settings for the `sigmax` and `sigmaY` settings:



To keep the blur consistent among different display sizes and resolutions, set `sigmax` and `sigmay` to values that are proportional to the rendered pixel size of the image that the blur is applied to.

Drop shadow

The `SKImageFilter.CreateDropShadow` static method creates an `SKImageFilter` object for a drop shadow:

```
public static SKImageFilter CreateDropShadow (float dx, float dy,
                                             float sigmaX, float sigmaY,
                                             SKColor color,
                                             SKDropShadowImageFilterShadowMode shadowMode,
                                             SKImageFilter input = null,
                                             SKImageFilter.CropRect cropRect = null);
```

Set this object to the `ImageFilter` property of an `SKPaint` object, and anything you draw with that object will have a drop shadow behind it.

The `dx` and `dy` parameters indicate the horizontal and vertical offsets of the shadow in pixels from the graphical object. The convention in 2D graphics is to assume a light source coming from the upper left, which implies that both these arguments should be positive to position the shadow below and to the right of the graphical object.

The `sigmax` and `sigmay` parameters are blurring factors for the drop shadow.

The `color` parameter is the color of the drop shadow. This `SKColor` value can include transparency. One possibility is the color value `SKColors.Black.WithAlpha(0x80)` to darken any color background.

The final two parameters are optional.

The **Drop Shadow Experiment** program lets you experiment with values of `dx`, `dy`, `sigmax`, and `sigmay` to display a text string with a drop shadow. The XAML file instantiates four `Slider` views to set these values:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaSharpFormsDemos.Effects.DropShadowExperimentPage"
    Title="Drop Shadow Experiment">
<ContentPage.Resources>
    <Style TargetType="Slider">
        <Setter Property="Margin" Value="10, 0" />
    </Style>

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

<StackLayout>
    <skia:SKCanvasView x:Name="canvasView"
        VerticalOptions="FillAndExpand"
        PaintSurface="OnCanvasViewPaintSurface" />

    <Slider x:Name="dxSlider"
        Minimum="-20"
        Maximum="20"
        ValueChanged="OnSliderValueChanged" />

    <Label Text="{Binding Source={x:Reference dxSlider},
        Path=Value,
        StringFormat='Horizontal offset = {0:F1}'}" />

    <Slider x:Name="dySlider"
        Minimum="-20"
        Maximum="20"
        ValueChanged="OnSliderValueChanged" />

    <Label Text="{Binding Source={x:Reference dySlider},
        Path=Value,
        StringFormat='Vertical offset = {0:F1}'}" />

    <Slider x:Name="sigmaXSlider"
        Maximum="10"
        ValueChanged="OnSliderValueChanged" />

    <Label Text="{Binding Source={x:Reference sigmaXSlider},
        Path=Value,
        StringFormat='Sigma X = {0:F1}'}" />

    <Slider x:Name="sigmaYSlider"
        Maximum="10"
        ValueChanged="OnSliderValueChanged" />

    <Label Text="{Binding Source={x:Reference sigmaYSlider},
        Path=Value,
        StringFormat='Sigma Y = {0:F1}'}" />
</StackLayout>
</ContentPage>

```

The code-behind file uses those values to create a red drop shadow on a blue text string:

```

public partial class DropShadowExperimentPage : ContentPage
{
    const string TEXT = "Drop Shadow";

    public DropShadowExperimentPage ()
    {
        InitializeComponent ();
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        // Get values from sliders
        float dx = (float)dxSlider.Value;
        float dy = (float)dySlider.Value;
        float sigmaX = (float)sigmaXSlider.Value;
        float sigmaY = (float)sigmaYSlider.Value;

        using (SKPaint paint = new SKPaint())
        {
            // Set SKPaint properties
            paint.TextSize = info.Width / 7;
            paint.Color = SKColors.Blue;
            paint.ImageFilter = SKImageFilter.CreateDropShadow(
                dx,
                dy,
                sigmaX,
                sigmaY,
                SKColors.Red,
                SKDropShadowImageFilterShadowMode.DrawShadowAndForeground);

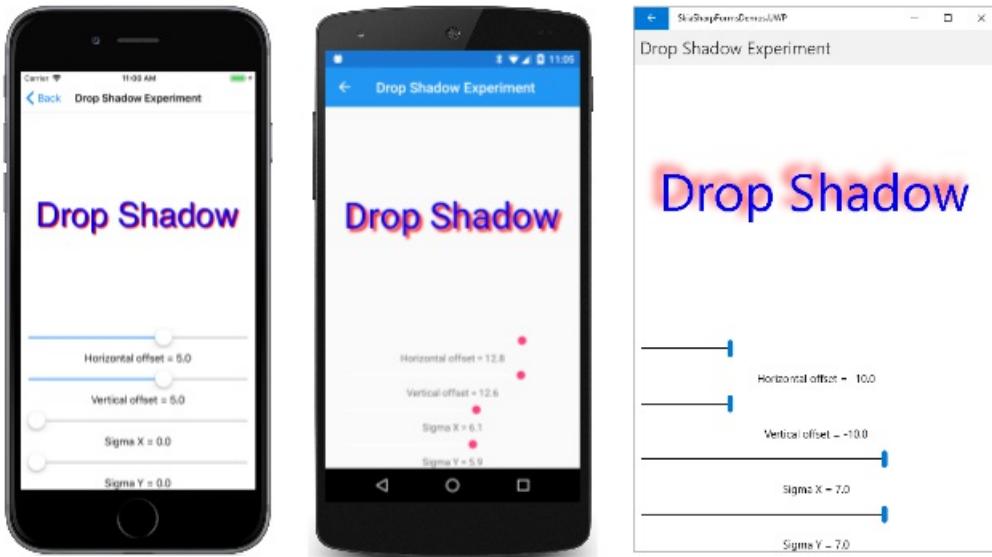
            SKRect textBounds = new SKRect();
            paint.MeasureText(TEXT, ref textBounds);

            // Center the text in the display rectangle
            float xText = info.Width / 2 - textBounds.MidX;
            float yText = info.Height / 2 - textBounds.MidY;

            canvas.DrawText(TEXT, xText, yText, paint);
        }
    }
}

```

Here's the program running:



The negative offset values in the Universal Windows Platform screenshot at the far right cause the shadow to appear above and to the left of the text. This suggests a light source in the lower right, which is not the convention for computer graphics. But it doesn't seem wrong in any way, perhaps because the shadow is also made very blurry and seems more ornamental than most drop shadows.

Lighting Effects

The `SKImageFilter` class defines six methods that have similar names and parameters, listed here in order of increasing complexity:

- [CreateDistantLitDiffuse](#)
- [CreateDistantLitSpecular](#)
- [CreatePointLitDiffuse](#)
- [CreatePointLitSpecular](#)
- [CreateSpotLitDiffuse](#)
- [CreateSpotLitSpecular](#)

These methods create image filters that mimic the effect of different kinds of light on three-dimensional surfaces. The resultant image filter illuminates two-dimensional objects as if they existed in 3D space, which can cause these objects to appear elevated or recessed, or with specular highlighting.

The `Distant` light methods assume that the light comes from a far distance. For the purpose of illuminating objects, the light is assumed to point in one consistent direction in 3D space, much like the Sun on a small area of the Earth. The `Point` light methods mimic a light bulb positioned in 3D space that emits light in all directions. The `spot` light has both a position and a direction, much like a flashlight.

Locations and directions in 3D space are both specified with values of the `SKPoint3` structure, which is similar to `SKPoint` but with three properties named `x`, `y`, and `z`.

The number and complexity of the parameters to these methods make experimentation with them difficult. To get you started, the **Distant Light Experiment** page lets you experiment with parameters to the `CreateDistantLightDiffuse` method:

```

public static SKImageFilter CreateDistantLitDiffuse (SKPoint3 direction,
                                                    SKColor lightColor,
                                                    float surfaceScale,
                                                    float kd,
                                                    SKImageFilter input = null,
                                                    SKImageFilter.CropRect cropRect = null);

```

The page doesn't use the last two optional parameters.

Three `Slider` views in the XAML file let you select the `z` coordinate of the `SKPoint3` value, the `surfaceScale` parameter, and the `kd` parameter, which is defined in the API documentation as the "diffuse lighting constant":

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:skia="clr-namespace:SkiaSharp.Views.Forms;assembly=SkiaSharp.Views.Forms"
    x:Class="SkiaLightExperiment.MainPage"
    Title="Distant Light Experiment">

    <StackLayout>

        <skia:SKCanvasView x:Name="canvasView"
            PaintSurface="OnCanvasViewPaintSurface"
            VerticalOptions="FillAndExpand" />

        <Slider x:Name="zSlider"
            Minimum="-10"
            Maximum="10"
            Margin="10, 0"
            ValueChanged="OnSliderValueChanged" />

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

        <Slider x:Name="surfaceScaleSlider"
            Minimum="-1"
            Maximum="1"
            Margin="10, 0"
            ValueChanged="OnSliderValueChanged" />

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

        <Slider x:Name="lightConstantSlider"
            Minimum="-1"
            Maximum="1"
            Margin="10, 0"
            ValueChanged="OnSliderValueChanged" />

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

```

The code-behind file obtains those three values and uses them to create an image filter to display a text string:

```

public partial class DistantLightExperimentPage : ContentPage
{
    const string TEXT = "Lighting";

    public DistantLightExperimentPage()
    {
        InitializeComponent();
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        canvasView.InvalidateSurface();
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        float z = (float)zSlider.Value;
        float surfaceScale = (float)surfaceScaleSlider.Value;
        float lightConstant = (float)lightConstantSlider.Value;

        using (SKPaint paint = new SKPaint())
        {
            paint.IsAntialias = true;

            // Size text to 90% of canvas width
            paint.TextSize = 100;
            float textWidth = paint.MeasureText(TEXT);
            paint.TextSize *= 0.9f * info.Width / textWidth;

            // Find coordinates to center text
            SKRect textBounds = new SKRect();
            paint.MeasureText(TEXT, ref textBounds);

            float xText = info.Rect.MidX - textBounds.MidX;
            float yText = info.Rect.MidY - textBounds.MidY;

            // Create distant light image filter
            paint.ImageFilter = SKImageFilter.CreateDistantLitDiffuse(
                new SKPoint3(2, 3, z),
                SKColors.White,
                surfaceScale,
                lightConstant);

            canvas.DrawText(TEXT, xText, yText, paint);
        }
    }
}

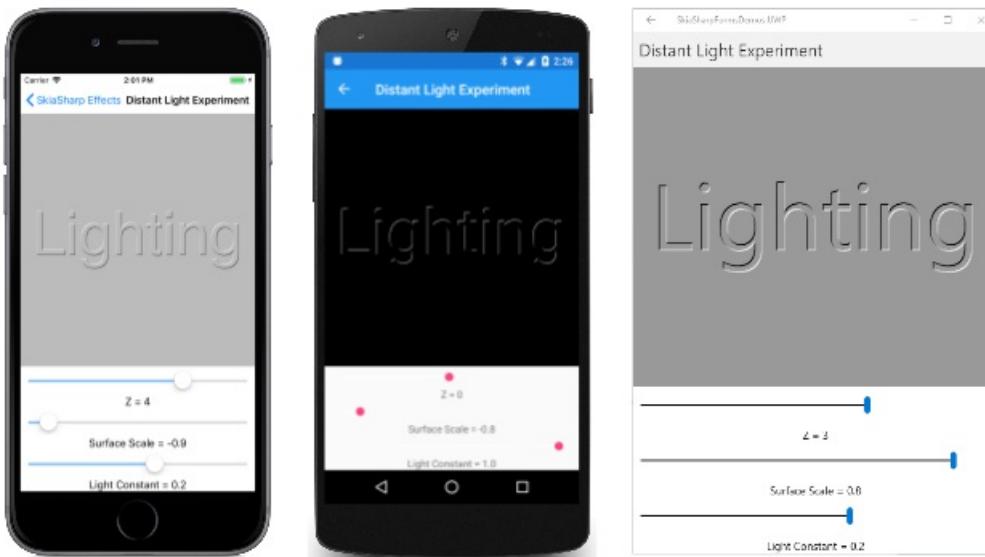
```

The first argument of `SKImageFilter.CreateDistantLitDiffuse` is the direction of the light. The positive X and Y coordinates indicate that the light is pointed to the right and down. Positive Z coordinates point into the screen. The XAML file allows you to select negative Z values, but that's only so you can see what happens: Conceptually, negative Z coordinates cause the light to point out of the screen. For anything other than small negative values, the lighting effect stops working.

The `surfaceScale` argument can range from -1 to 1 . (Higher or lower values have no further effect.) These are relative values in the Z axis that indicate the displacement of the graphical object (in this case, the text string) from the canvas surface. Use negative values to raise the text string above the surface of the canvas, and positive values to depress it into the canvas.

The `lightConstant` value should be positive. (The program allows negative values so you can see that they cause the effect to stop working.) Higher values cause more intense light.

These factors can be balanced to obtain an embossed effect when `surfaceScale` is negative (as with the iOS and Android screenshots) and an engraved effect when `surfaceScale` is positive, as with the UWP screenshot at the right:



The Android screenshot has a Z value of 0, which means that the light is only pointing down and to the right. The background isn't illuminated and the surface of the text string isn't illuminated either. The light only effects the edge of the text for a very subtle effect.

An alternative approach to embossed and engraved text was demonstrated in the article [The Translate Transform](#): The text string is displayed twice with different colors that are offset slightly from each other.

Related links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)

SkiaSharp color filters

3/5/2021 • 7 minutes to read • [Edit Online](#)

 [Download the sample](#)

Color filters can translate colors in a bitmap (or other image) to other colors for effects such as posterization:



To use a color filter, set the `ColorFilter` property of `SKPaint` to an object of type `SKColorFilter` created by one of the static methods in that class. This article demonstrates:

- a color transform created with the `CreateColorMatrix` method.
- a color table created with the `CreateTable` method.

The color transform

The color transform involves using a matrix to modify colors. Like most 2D graphics systems, SkiaSharp uses matrices mostly for transforming coordinate points as discussed in the article [Matrix Transforms in SkiaSharp](#). The `SKColorFilter` also supports matrix transforms, but the matrix transforms RGB colors. Some familiarity with matrix concepts is necessary to understand these color transforms.

The color-transform matrix has a dimension of four rows and five columns:

M ₁₁	M ₁₂	M ₁₃	M ₁₄	M ₁₅
M ₂₁	M ₂₂	M ₂₃	M ₂₄	M ₂₅
M ₃₁	M ₃₂	M ₃₃	M ₃₄	M ₃₅
M ₄₁	M ₄₂	M ₄₃	M ₄₄	M ₄₅

It transforms a RGB source color (R, G, B, A) to the destination color (R', G', B', A').

In preparation for the matrix multiplication, the source color is converted to a 5×1 matrix:

R
G
B
A
1

These R, G, B, and A values are the original bytes ranging from 0 to 255. They are *not* normalized to floating

point values in the range 0 to 1.

The extra cell is required for a translation factor. This is analogous to the use of a 3×3 matrix to transform two-dimensional coordinate points as described in the section [The Reason for the 3-by-3 Matrix](#) in the article on using matrices for transforming coordinate points.

The 4×5 matrix is multiplied by the 5×1 matrix, and the product is a 4×1 matrix with the transformed color:

$$\begin{vmatrix} M_{11} & M_{12} & M_{13} & M_{14} & M_{15} \\ M_{21} & M_{22} & M_{23} & M_{24} & M_{25} \\ M_{31} & M_{32} & M_{33} & M_{34} & M_{35} \\ M_{41} & M_{42} & M_{43} & M_{44} & M_{45} \end{vmatrix} \times \begin{vmatrix} R \\ G \\ B \\ A \\ 1 \end{vmatrix} = \begin{vmatrix} R' \\ G' \\ B' \\ A' \end{vmatrix}$$

Here are the separate formulas for R' , G' , B' , and A' :

$$R' = M_{11} \cdot R + M_{12} \cdot G + M_{13} \cdot B + M_{14} \cdot A + M_{15}$$

$$G' = M_{21} \cdot R + M_{22} \cdot G + M_{23} \cdot B + M_{24} \cdot A + M_{25}$$

$$B' = M_{31} \cdot R + M_{32} \cdot G + M_{33} \cdot B + M_{34} \cdot A + M_{35}$$

$$A' = M_{41} \cdot R + M_{42} \cdot G + M_{43} \cdot B + M_{44} \cdot A + M_{45}$$

Most of the matrix consists of multiplicative factors that are generally in the range of 0 to 2. However, the last column (M_{15} through M_{45}) contains values that are added in the formulas. These values generally range from 0 to 255. The results are clamped between the values of 0 and 255.

The identity matrix is:

$$\begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{vmatrix}$$

This causes no change to the colors. The transform formulas are:

$$R' = R$$

$$G' = G$$

$$B' = B$$

$$A' = A$$

The M_{44} cell is very important because it preserves opacity. It is generally the case that M_{41} , M_{42} , and M_{43} are all zero, because you probably don't want opacity to be based on the red, green, and blue values. But if M_{44} is zero, then A' will be zero, and nothing will be visible.

One of the most common uses of the color matrix is to convert a color bitmap to a gray-scale bitmap. This involves a formula for a weighted average of the red, green, and blue values. For video displays using the sRGB ("standard red green blue") color space, this formula is:

$$\text{gray-shade} = 0.2126 \cdot R + 0.7152 \cdot G + 0.0722 \cdot B$$

To convert a color bitmap to a gray-scale bitmap, the R' , G' , and B' results must all equal that same value. The matrix is:

```

| 0.21 0.72 0.07 0 0 |
| 0.21 0.72 0.07 0 0 |
| 0.21 0.72 0.07 0 0 |
| 0     0     0     1 0 |

```

There is no SkiaSharp data type that corresponds to this matrix. Instead you must represent the matrix as an array of 20 `float` values in row order: first row, then second row, and so forth.

The static `SKColorFilter.CreateColorMatrix` method has the following syntax:

```
public static SKColorFilter CreateColorMatrix (float[] matrix);
```

where `matrix` is an array of the 20 `float` values. When creating the array in C#, it is easy to format the numbers so they resemble the 4×5 matrix. This is demonstrated in the **Gray-Scale Matrix** page in the [SkiaSharpFormsDemos](#) sample:

```
public class GrayScaleMatrixPage : ContentPage
{
    SKBitmap bitmap = BitmapExtensions.LoadBitmapResource(
        typeof(CenteredTilesPage),
        "SkiaSharpFormsDemos.Media.Banana.jpg");

    public GrayScaleMatrixPage()
    {
        Title = "Gray-Scale Matrix";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

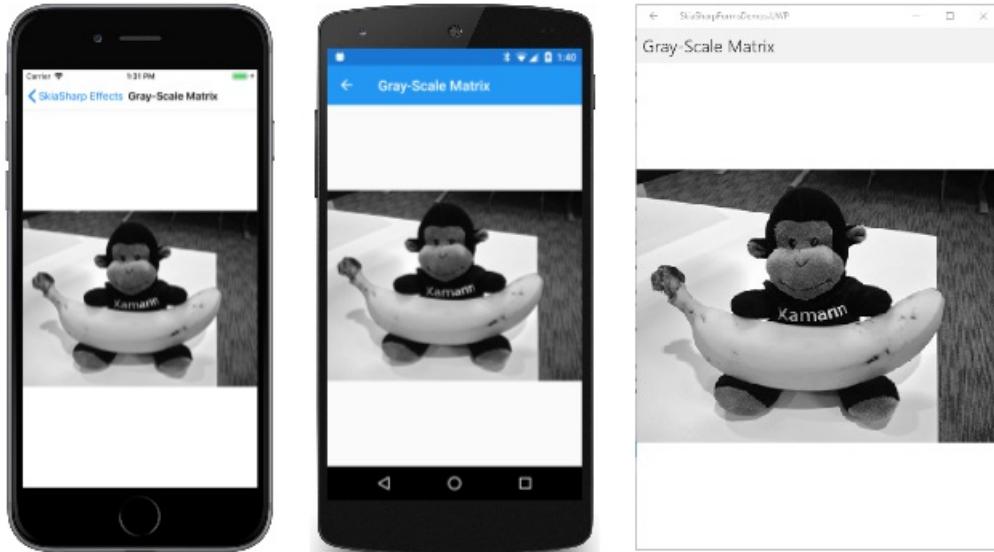
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        using (SKPaint paint = new SKPaint())
        {
            paint.ColorFilter =
                SKColorFilter.CreateColorMatrix(new float[]
                {
                    0.21f, 0.72f, 0.07f, 0, 0,
                    0.21f, 0.72f, 0.07f, 0, 0,
                    0.21f, 0.72f, 0.07f, 0, 0,
                    0,     0,     0,     1, 0
                });
            canvas.DrawBitmap(bitmap, info.Rect, BitmapStretch.Uniform, paint: paint);
        }
    }
}
```

The `DrawBitmap` method used in this code is from the `BitmapExtension.cs` file included with the [SkiaSharpFormsDemos](#) sample.

Here's the result running on iOS, Android, and Universal Windows Platform:



Watch out for the value in the fourth row and fourth column. That's the crucial factor that is multiplied by the A value of the original color for the A' value of the transformed color. If that cell is zero, nothing will be displayed and the problem might be difficult to locate.

When experimenting with color matrices, you can treat the transform either from the perspective of the source or the perspective of the destination. How should the red pixel of the source contribute to the red, green, and blue pixels of the destination? That's determined by the values in the first *column* of the matrix. Alternatively, how should the destination red pixel be affected by the red, green, and blue pixels of the source? That's determined by the first *row* of the matrix.

For some ideas on how to use color transforms, see the [Recoloring Images](#) pages. The discussion concerns Windows Forms, and the matrix is a different format, but the concepts are the same.

The **Pastel Matrix** calculates the destination red pixel by attenuating the source red pixel and slightly emphasizing the red and green pixels. This process occurs similarly for the green and blue pixels:

```

public class PastelMatrixPage : ContentPage
{
    SKBitmap bitmap = BitmapExtensions.LoadBitmapResource(
        typeof(PastelMatrixPage),
        "SkiaSharpFormsDemos.Media.MountainClimbers.jpg");

    public PastelMatrixPage()
    {
        Title = "Pastel Matrix";

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

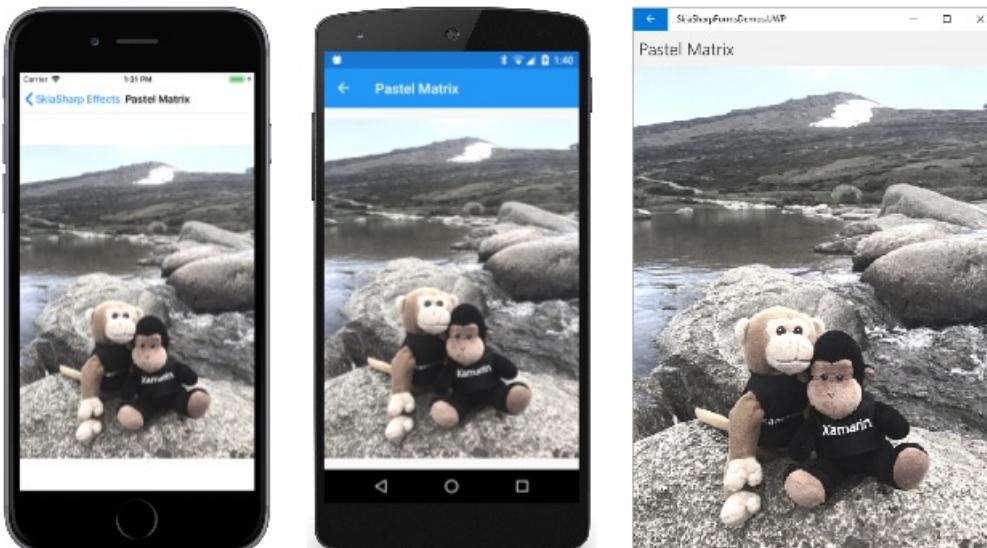
    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

        canvas.Clear();

        using (SKPaint paint = new SKPaint())
        {
            paint.ColorFilter =
                SKColorFilter.CreateColorMatrix(new float[]
                {
                    0.75f, 0.25f, 0.25f, 0, 0,
                    0.25f, 0.75f, 0.25f, 0, 0,
                    0.25f, 0.25f, 0.75f, 0, 0,
                    0, 0, 0, 1, 0
                });
            canvas.DrawBitmap(bitmap, info.Rect, BitmapStretch.Uniform, paint: paint);
        }
    }
}

```

The result is to mute the intensity of the colors as you can see here:



Color tables

The static `SKColorFilter.CreateTable` method comes in two versions:

```
public static SKColorFilter CreateTable (byte[] table);

public static SKColorFilter CreateTable (byte[] tableA, byte[] tableR, byte[] tableG, byte[] tableB);
```

The arrays always contain 256 entries. In the `CreateTable` method with one table, the same table is used for the red, green, and blue components. It's a simple look-up table: If the source color is (R, G, B), and the destination color is (R', B', G'), then the destination components are obtained by indexing `table` with the source components:

```
R' = table[R]
G' = table[G]
B' = table[B]
```

In the second method, each of four color components can have a separate color table, or the same color tables might be shared among two or more components.

If you want to set one of the arguments to the second `CreateTable` method to a color table that contains the values 0 through 255 in sequence, you can use `null` instead. Very often the `CreateTable` call has a `null` first argument for the alpha channel.

In the section on **Posterization** in the article on [Accessing SkiaSharp bitmap pixel bits](#), you saw how to modify the individual pixel bits of a bitmap to reduce its color resolution. This is a technique called *posterization*.

You can also posterize a bitmap with a color table. The constructor of the **Posterize Table** page creates a color table that maps its index to a byte with the bottom 6 bits set to zero:

```

public class PosterizeTablePage : ContentPage
{
    SKBitmap bitmap = BitmapExtensions.LoadBitmapResource(
        typeof(PosterizeTablePage),
        "SkiaSharpFormsDemos.Media.MonkeyFace.png");

    byte[] colorTable = new byte[256];

    public PosterizeTablePage()
    {
        Title = "Posterize Table";

        // Create color table
        for (int i = 0; i < 256; i++)
        {
            colorTable[i] = (byte)(0xC0 & i);
        }

        SKCanvasView canvasView = new SKCanvasView();
        canvasView.PaintSurface += OnCanvasViewPaintSurface;
        Content = canvasView;
    }

    void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
    {
        SKImageInfo info = args.Info;
        SKSurface surface = args.Surface;
        SKCanvas canvas = surface.Canvas;

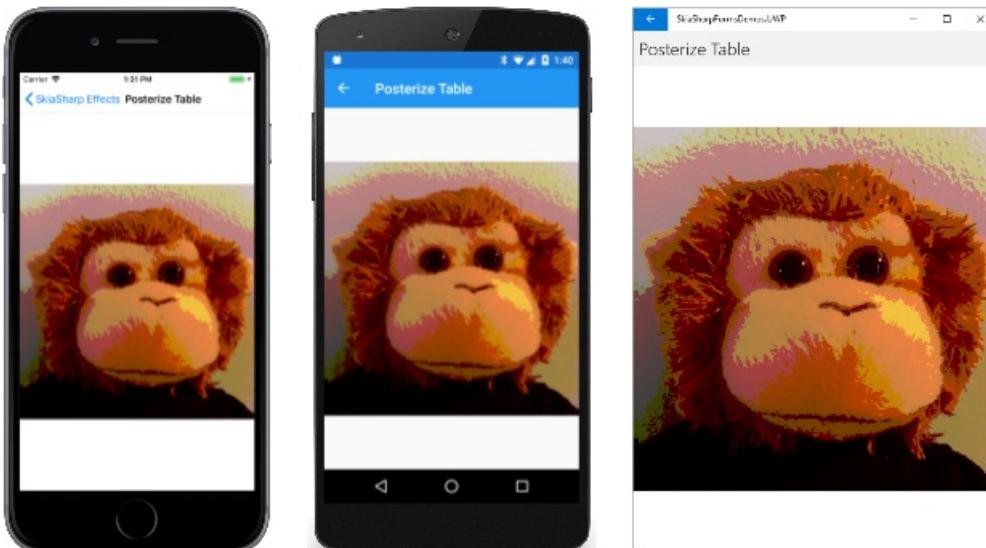
        canvas.Clear();

        using (SKPaint paint = new SKPaint())
        {
            paint.ColorFilter =
                SKColorFilter.CreateTable(null, null, colorTable, colorTable);

            canvas.DrawBitmap(bitmap, info.Rect, BitmapStretch.Uniform, paint: paint);
        }
    }
}

```

The program chooses to use this color table only for the green and blue channels. The red channel continues to have full resolution:



You can use various color tables for the different color channels for various effects.

Related links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)