

# Graphics and Animation in Android

Evolve Advanced Track, Chapter 5

## Overview

---

It's possible to turn a good application into a great application with some well-placed graphics and animations. The creators of Android recognize this, and provided many different ways to create graphics and animate them.

In many situations, embedding PNG or JPEG files in an application will satisfy the drawing requirements for Android applications. However, sometimes they don't. Android has a number of ways to provide custom drawable resources to an application. There are two primary methods for creating 2D graphics in Android:

- **Drawable resources** – These are used to create custom graphics either programmatically or, more typically, by embedding special instructions in XML files.
- **Canvas** – this is a low level API that involves drawing directly on an underlying bitmap. Android will then render this underlying bitmap to the screen.

Along with 2D graphics, Android also provides several different ways to create animations:

- Android also supports frame-by-frame animations known as *Drawable Animation*. This is the simplest animation API - Android will sequentially load and display drawable resources in sequence, much like a cartoon.
- *View Animations* are the original animation API's in Android. They are simpler and easier to setup but are limited in that they can only work on Views.
- In 2011, Android 3.0 introduced a new set of animation API's known as *Property Animations*. These new API's introduced an extensible and flexible system that can be used to animate the properties of any object, not just View objects.

All of these frameworks are viable options, however where possible preference should be given to Property Animations, as it is an easier and more flexible API to work with.

## Graphics

---

The Android framework provides several systems for displaying 2D graphics in existing views. There are three commonly used techniques for providing graphics and artwork to an Android application:

- **Resources and Views** – this involves adding images such as PNGs or JPEGs to an application and then displaying them using widgets such as the `ImageView`.
- **Drawables** – In Android, drawables represent a general abstraction of “a thing that can be drawn”. Drawable resources are typically defined as XML files that contain instructions or actions for Android to render a 2D graphic.

- **Drawing with a Canvas** – this is a low-level API that involves customizing a view by overriding its `onDraw` method. It allows for very fine-grained control over what is displayed.

The first of these, embedding PNG or JPEG is the simplest and most common way of embedding graphics in an Android application. This was covered in previous chapters and will not be dealt with here.

The second of these is a very common and popular technique in Android applications. It allows graphics to be created in XML. As with other resources, this approach allows for a clean separation of code from resources. This can simplify development and maintenance because it is not necessary to change code to update or change the graphics in an Android application. The well-defined instructions can be more soluble to developers.


The third and final of these techniques involves using a Canvas object. This is similar to the iOS API. Using the Canvas object provides a low-level and the most control of how 2D graphics are created. It is appropriate for situations where a drawable resource will not work or will be difficult to work with. For example, it may be necessary to draw a custom slider control whose appearance will change based on calculations related to the value of the slider.

## Drawable Resources

Because drawable resources are an important part of Android application development, let's take a deeper look. Drawable resources are stored as an XML file in the directory `/Resources/drawable`. It is not necessary to provide density-specific

Android defines several different types of drawable resources. These are listed in the following table with a brief description:

Drawable Resource	Description
<a href="#">ShapeDrawable</a>	This is a drawable object that draws a primitive geometric shape and applying a limited set of graphical effects on that shape. They are very handy for things such as customizing Buttons or setting the background of TextViews.
<a href="#">StateListDrawable</a>	This is a drawable resource that will change appearance based on the state of a widget. For example, a button may change its appearance depending on if it is pressed or not. The <code>StateListDrawable</code> was discussed in some of the previous chapters.

<a href="#">LayerDrawable</a>	<p>This drawable resource that will stack several other drawables one on top of another. An example of a Layer is shown in the following screenshot:</p> 
<a href="#">TransitionDrawable</a>	<p>This is a LayerDrawable but with one difference. A TransitionDrawable is able to animation one layer showing up overtop another.</p>
<a href="#">LevelListDrawable</a>	<p>This is very similar to a StateListDrawable in that it will display a image based on certain conditions. However, unlike a StateListDrawable, the LevelListDrawable displays an image based on an integer value.</p>
<a href="#">ScaleDrawable</a> <a href="#">ClipDrawable</a>	<p>/ These drawables do pretty much what their name implies. The ScaleDrawable will scale another drawable, while the ClipDrawable will clip another drawable.</p>
<a href="#">InsetDrawable</a>	<p>This drawable will apply insets on the sides of another drawable resource. It is used when a View needs a background that is smaller than the View's actual bounds.</p>
XML <a href="#">BitmapDrawable</a>	<p>This file is a set of instructions, in XML, that are to be performed on an actual bitmap. Some actions that Android can perform are tiling, dithering, and anti-aliasing.</p>

Let's look at a quick example of how to use one a drawable resource using a ShapeDrawable. A ShapeDrawable can define one of the four basic shapes: rectangle, oval, line, and ring. It is also possible to apply basic effects, such as gradient, colour, and size. An XML file is added to the /Resources/drawable, and may contain the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android=http://schemas.android.com/apk/res/android
    android:shape="oval">

    <gradient
        android:type="sweep"
        android:startColor="#77990099"
        android:endColor="#22990099"/>

    <stroke
        android:width="1dp"
        android:color="#aa990099" />

    <padding
        android:left="10dp"
```

```

        android:right="10dp"
        android:top="10dp"
        android:bottom="10dp" />
    </shape>

```

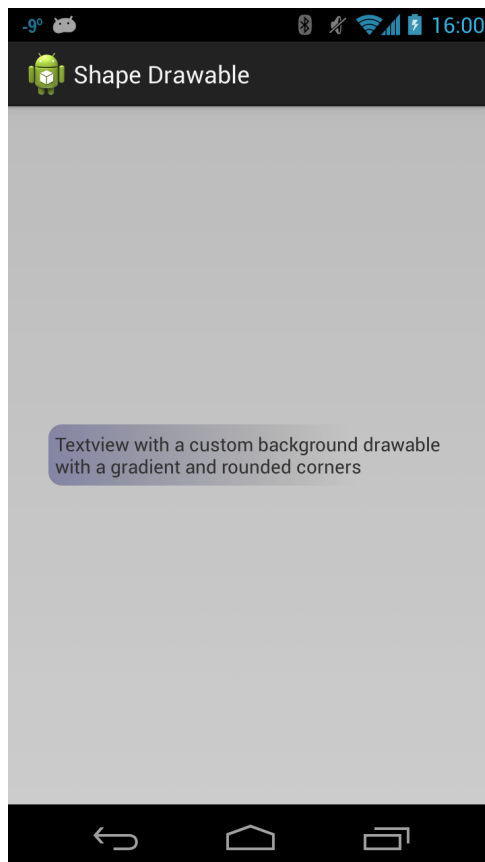
This defines an oval layout that has a purple gradient background. One way to use this shape drawable is to set the background property on a View, as shown in the following XML layout file:

```

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#33000000">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:background="@drawable/shape_rounded_blue_rect"
        android:text="@string/message_shapedrawable" />
</RelativeLayout>

```

This following screen shot shows the result of applying this shape drawable to a TextView:



For more details about the XML elements and syntax of the shape drawable XML file, consult [Google's documentation](http://developer.android.com/guide/topics/ui/visual-effects.html#shapedrawable).

## Drawing with a Canvas

This is a low-level approach that allows for maximum control and flexibility. At the center of these API's is the Canvas class. Canvas objects cannot work in isolation. They will take processing instructions and data from other graphics related classes and use those to draw to a bitmap that can be drawn on the screen.

The Canvas object can be thought of as a container of instructions on how to draw 2D graphics. It exposes methods that allow you to pass on these drawing instructions to the Canvas object. The Canvas will perform all of these drawing instructions on an underlying bitmap, which is ultimately displayed on the screen.

There are two ways to obtain a Canvas object. The first way involves defining a Bitmap object, and then instantiate a Canvas object with it. For example, the following code snippet creates a new canvas with an underlying bitmap:

```
Bitmap bitmap = Bitmap.CreateBitmap(100, 100, Bitmap.Config.Argb8888);
Canvas canvas = new Canvas(bitmap);
```

The other way to obtain a Canvas object is by the OnDraw callback method that is provided by the View base class. Android calls this method when it decides a View needs to draw itself, and passing in a Canvas object for the View to work with.

One class that is commonly used with the Canvas is the Paint class. This class holds colour and style information about how to draw on the class. It is used to provide things such as color and transparency.

The Canvas class exposes methods to programmatically provide the draw instructions. These methods are named Draw...(), for example:

- Canvas.DrawPaint(Paint p) – fills the entire canvas's bitmap with the specified paint.
- Canvas.DrawPath(Path path, Paint paint) – draws the specified geometric shape using the specified paint.
- Canvas.DrawText(String text, float x, float y, Paint paint) – draw the text on the canvas, with the specified colour, with the at x,y.

The following code snippet shows how to draw view:

```
public class MyView : View
{
    protected override void OnDraw(Canvas canvas)
    {
        base.OnDraw(canvas);
        Paint green = new Paint
        {
            AntiAlias = true,
            Color = Color.Rgb(0x99, 0xcc, 0),
        };
        green.SetStyle(Paint.Style.FillAndStroke);

        Paint red = new Paint
        {
```

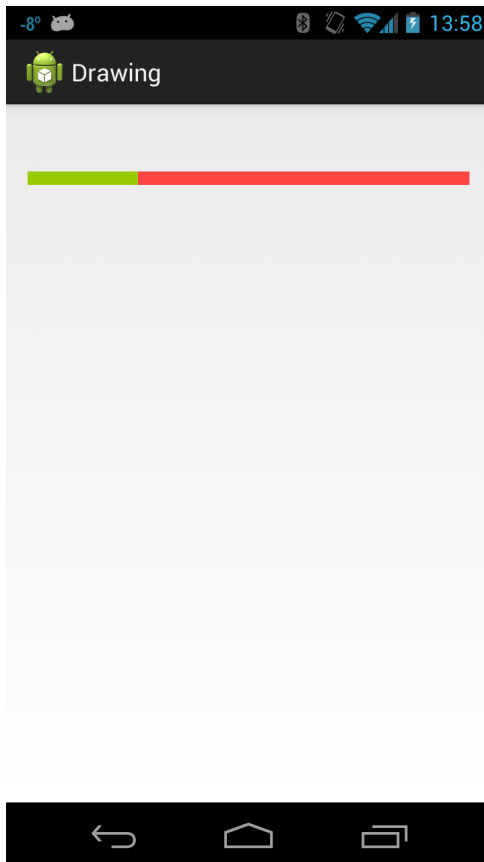
```

        AntiAlias = true,
        Color = Color.Rgb(0xff, 0x44, 0x44)
    };
    red.SetStyle(Paint.Style.FillAndStroke);

    float middle = canvas.Width * 0.25f;
    canvas.DrawPaint(red);
    canvas.DrawRect(0, 0, middle, canvas.Height, green);
}
}

```

This code first creates a red paint and a green paint object. It fills the content of the canvas with red, and then instructs the canvas to draw a green rectangle that is 25% of the width of the canvas. The following screenshot shows the result of this code:



## Animation

---

Users like things that move in their applications. Animations are a great way to improve the user experience of an application and help it stand out. The best animations are the ones that users don't notice because they feel natural. Android provides the following three API's for:

- **View Animation** – This is the original API that Android provided. These animations are tied to a specific View and can perform simple transformations on the contents of the View. Because of its simplicity, this API is still useful for things like alpha animations, rotations, and so forth.
- **Property Animation** – Property animations were introduced in Android 3.0 and enable an application to animate almost anything. Property animations can be used to change any property of any object, even if that object is not visible on the screen.
- **Drawable Animation** – This is a special drawable resource that is used to apply a very simple animation effect to layouts.

In general, property animation is the preferred system to use as it is more flexible and offers more features.

## View Animations

View animations are limited to Views and can only perform animations on values such as start and end points, size, rotation, and transparency. These types of animations are typically referred to as tween animations. View animations can be defined two ways – programmatically in code or by using XML files. XML files are the preferred way to declare view animations, as they are more readable and easier to maintain.

The animation XML files will be stored in the `/Resources/anim` directory of a Xamarin.Android project. This file must have one of the following elements as the root element :

- `<alpha>` - This is a fade-in or fade-out animation.
- `<rotate>` - This is a rotation animation.
- `<scale>` - A resizing animation.
- `<translate>` - This is a horizontal and/or vertical motion.
- `<set>` - This is a container that may hold one or more of the other animation elements.

By default, all animations in an XML file will be applied simultaneously. To make animations occur sequentially, set the `android:startOffset` attribute on one of the elements defined above.

It is possible to affect the rate of change in an animation by using an *interpolator*. An interpolator allows animate effects to be accelerated, repeated, or decelerated. The Android framework provides several interpolators out of the box, such as (but not limited to):

- `AccelerateInterpolator` / `DecelerateInterpolator` – these interpolators increase or decrease the rate of change in an animation.
- `BounceInterpolator` – the change bounces at the end.
- `LinearInterpolator` – the rate of change is constant.



The following XML shows an example of an animation file that combines some of these elements:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android=http://schemas.android.com/apk/res/android
    android:shareInterpolator="false">

    <scale
        android:interpolator="@android:anim/accelerate_decelerate_interpolator"
        android:fromXScale="1.0"
        android:toXScale="1.4"
        android:fromYScale="1.0"
        android:toYScale="0.6"
        android:pivotX="50%"
        android:pivotY="50%"
        android:fillEnabled="true"
        android:fillAfter="false"
        android:duration="700" />

    <set android:interpolator="@android:anim/accelerate_interpolator">

        <scale
            android:fromXScale="1.4"
            android:toXScale="0.0"
            android:fromYScale="0.6"
            android:toYScale="0.0"
            android:pivotX="50%"
            android:pivotY="50%"
            android:fillEnabled="true"
            android:fillBefore="false"
            android:fillAfter="true"
            android:startOffset="700"
            android:duration="400" />

        <rotate
            android:fromDegrees="0"
            android:toDegrees="-45"
            android:toYScale="0.0"
            android:pivotX="50%"
            android:pivotY="50%"
            android:fillEnabled="true"
            android:fillBefore="false"
            android:fillAfter="true"
            android:startOffset="700"
            android:duration="400" />

    </set>

</set>
```

This animation will perform all the animations simultaneously – the first scale animation will stretch the image horizontally and shrink it vertically. Then the image will simultaneously be rotated 45 degrees counter-clockwise and shrunk as it is not on the screen.

Once the animation is applied, it can be programmatically applied to a View by inflating the animation and then applying it to a View. Android provides the helper class `Android.Views.Animations.AnimationUtils` that will inflate an animation resource and return an instance of `Android.Views.Animations.Animation`. This object is applied to a View by calling `StartAnimation` and passing the `Animation` object. The following code snippet shows an example of this:

```
ImageView myImage = FindViewById<ImageView>(Resource.Id.imageView1);
Animation myAnimation =
    AnimationUtils.LoadAnimation(Resource.Animation.MyAnimation);
myImage.StartAnimation(myAnimation);
```

Now that we have a fundamental understanding of how View Animations work, lets move on to Property Animations.

## Property Animations

Property animators are a new API that was introduced in Android 3.0. They are meant to be the

All property animations are created by instances of the `Animator` subclass. Applications do not directly use this class, instead they will use one of its subclasses:

- `ValueAnimator` – This class is the most important class in the entire property animation API. It calculates the values of properties that need to be changed. The `ViewAnimator` does not directly update those values however. Instead it will raise events that can be used to update animated objects.
- `ObjectAnimator` – This class is a subclass of `ValueAnimator`. It is meant to simplify the process of animating objects accepting a target object and property to update.
- `AnimationSet` – This class is responsible for orchestrating how animations run in relation to one another. Animations may run simultaneously, sequentially, or with a specified delay between them.

*Evaluators* are special classes that are used by animators to calculate the new values during an animation. Out of the box, Android provides the following evaluators:

- `IntEvaluator` – This will calculate values for integer properties.
- `FloatEvaluator` – This will calculate values for float properties.
- `ArgbEvaluator` – This will calculate values for colour properties.

If the property that is being animated is not a `float`, `int` or colour, applications may create their own evaluator by implementing the `ITypeEvaluator` interface. Implementing custom evaluators is beyond the scope of this document and will be covered in future documentation from Xamarin.

## Using the ValueAnimator

There are two parts to any animation: calculating animated values and the setting those values on properties on some object. `ValueAnimator` will only calculate the values, but it will not operate on objects directly. Instead, objects will be updated inside event handlers that will be invoked during the animation lifespan. This design allows several properties to be updated from one animated value.

You obtain an instance of `ValueAnimator` by calling one of the following factory methods:

- `ValueAnimator.OfInt`
- `ValueAnimator.OfFloat`
- `ValueAnimator.OfObject`

Once that is done, the `ValueAnimator` instance must have its duration set, and then it can be started. The following example shows how to animate a value from 0 to 1 over the span of 1000 milliseconds:

```
ValueAnimator animator = ValueAnimator.OfInt(0, 100);
animator.SetDuration(1000);
animator.Start();
```

But itself, the code snippet above is not very useful – the animator will run but there is no target for the updated value. The `Animator` class will raise the `Update` event when it decides it is necessary to inform listeners of a new value. Applications may provide an event handler to respond to this event as shown in the following code snippet:

```
MyCustomObject myObj = new MyCustomObject();
myObj.SomeIntegerValue = -1;

animator.Update += (object sender, ValueAnimator.AnimatorUpdateEventArgs
e) =>
{
    int newValue = (int) e.Animation.AnimatedValue;
    // Apply this new value to the object being animated.
    myObj.SomeIntegerValue = newValue;
};
```

Now that we have an understanding of `ValueAnimator`, lets learn more about the `ObjectAnimator`.

## Using the ObjectAnimator

`ObjectAnimator` is a subclass of `ValueAnimator` that combines the timing engine and value computation of the `ValueAnimator` with the logic required to wire up event handlers. The `ValueAnimator` required applications to explicitly wire up an event handler – `ObjectAnimator` will take care of this step for us.

The API for `ObjectAnimator` is very similar to the API for `ValueAnimator`, but requires you provide the object and the name of the property to update. The following example shows an example of using `ObjectAnimator`:

```
MyCustomObject myObj = new MyCustomObject();
myObj.SomeIntegerValue = -1;
```

```

ObjectAnimator animator = ObjectAnimator.OfFloat(myObj, "SomeIntegerValue",
0, 100);
animator.SetDuration(1000);
animator.Start();

```

As you can see from the previous code snippet `ObjectAnimator` can reduce and simplify the code the code that is necessary to animate an object.

## Drawable Animations

The final animation API is the drawable animation. Drawable animations load a series of drawable resources one after the other and display them sequentially, similar to a flip-it cartoon.

Drawable resources are defined in an XML file that has an `<animation-list>` element as the root element and a series of `<item>` elements that define each frame in the animation. This XML file is stored in the `/Resource/drawable` folder of the application. The following XML is an example of a drawable animation:

```

<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
>
    <item android:drawable="@drawable/asteroid01"
android:duration="100" />
    <item android:drawable="@drawable/asteroid02"
android:duration="100" />
    <item android:drawable="@drawable/asteroid03"
android:duration="100" />
    <item android:drawable="@drawable/asteroid04"
android:duration="100" />
    <item android:drawable="@drawable/asteroid05"
android:duration="100" />
    <item android:drawable="@drawable/asteroid06"
android:duration="100" />
</animation-list>

```

This animation will run through six frames. The `android:duration` attribute declares how long each frame will be displayed. The next code snippet shows an example of creating a drawable animation and starting it when the user clicks a button on the screen:

```

AnimationDrawable _asteroidDrawable;

protected override void onCreate(Bundle bundle)
{
    base.OnCreate(bundle);
    SetContentView(Resource.Layout.Main);

    _asteroidDrawable = (Android.Graphics.Drawables.AnimationDrawable)
Resources.GetDrawable(Resource.Drawable.spinning_asteroid);

    ImageView asteroidImage =
FindViewById<ImageView>(Resource.Id.imageView2);
    asteroidImage.SetImageDrawable((Android.Graphics.Drawables.Drawable
) _asteroidDrawable);

    Button asteroidButton =
FindViewById<Button>(Resource.Id.spinAsteroid);
    asteroidButton.Click += (sender, e) =>

```

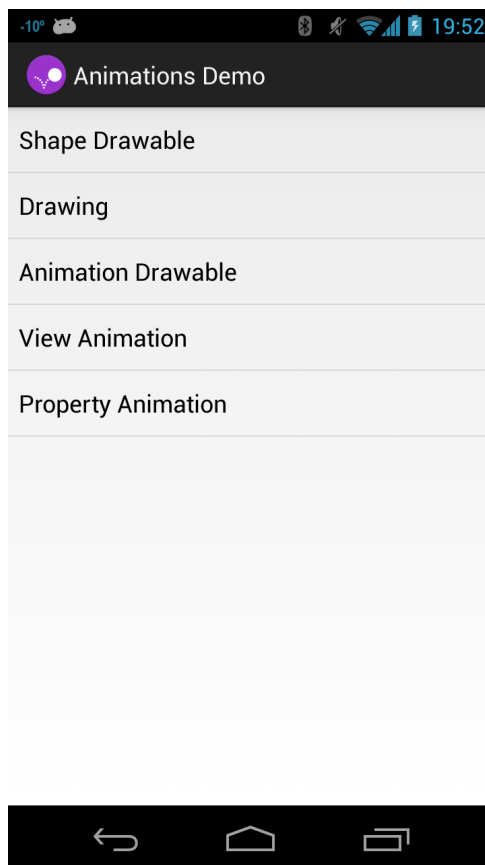
```
        {  
            _asteroidDrawable.Start();  
        };  
    }
```

At this point we have an understanding of graphics and animations. Let's move on to a walkthrough that will demonstrate some of these concepts in action.

## Walkthrough

---

With the basics out of the way, let's move on and see some of these concepts at work. Rather than create many little applications, we will create one application with an Activity that demonstrates each concept. The following screenshot shows the main activity:

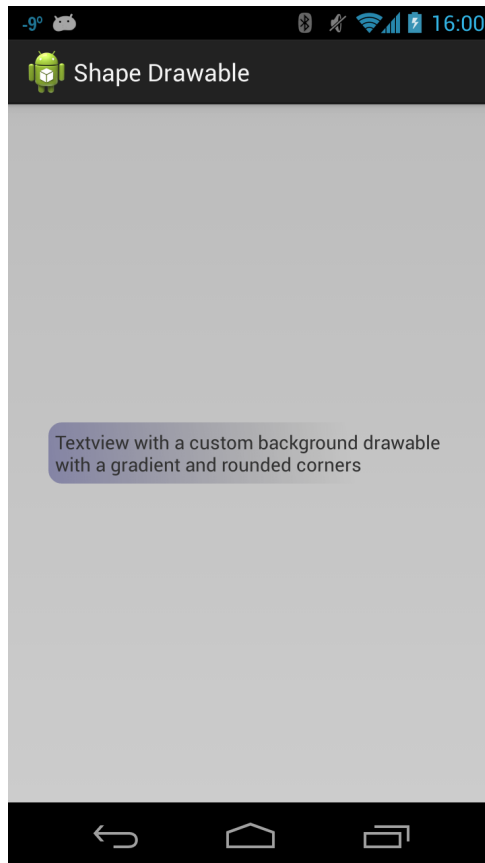


The logic for this screen has already been created, and the activities have been stubbed. The appropriate string resources have also been added to the project, so you don't have to worry about those either.

We will go through each section and implement the functionality for each screen. Open the solution `AnimationsDemo.sln`, and let's get started.

# Shape Drawable

This activity will demonstrate how create a ShapeDrawable that will look something like the screenshot below.



1. The first thing we need to add is an XML file. Create the file `/Resources/Drawable/shape_rounded_blue_rect.xml`. Edit the file so that it contains the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android=http://schemas.android.com/apk/res/android
    android:shape="rectangle">

    <!-- Specify a gradient for the background -->
    <gradient
        android:angle="45"
        android:startColor="#55000066"
        android:centerColor="#00000000"
        android:endColor="#00000000"
        android:centerX="0.75"/>

    <padding
        android:left="5dp"
        android:right="5dp"
        android:top="5dp"
        android:bottom="5dp" />

    <corners
```

```

        android:topLeftRadius="10dp"
        android:topRightRadius="10dp"
        android:bottomLeftRadius="10dp"
        android:bottomRightRadius="10dp" />
    </shape>

```

This drawable setups the gradient for the background, the internal margins, and specifies rounded corners.

2. The next thing that we need is the layout file for `ShapeDrawableActivity`. Create the layout file `Resources/Layout/activity_shapedrawable.xml` with the following contents:

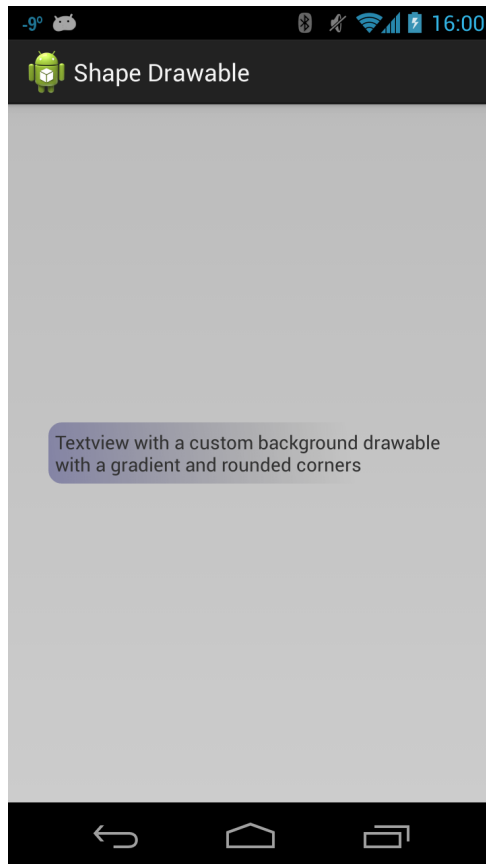
```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#33000000">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:background="@drawable/shape_rounded_blue_rect"
        android:text="@string/message_shapedrawable" />
</RelativeLayout>

```

This layout file uses the shape drawable that we created in the previous step as the background for the `TextView`. When Android inflates this layout, it will create a bitmap and set that as the background.

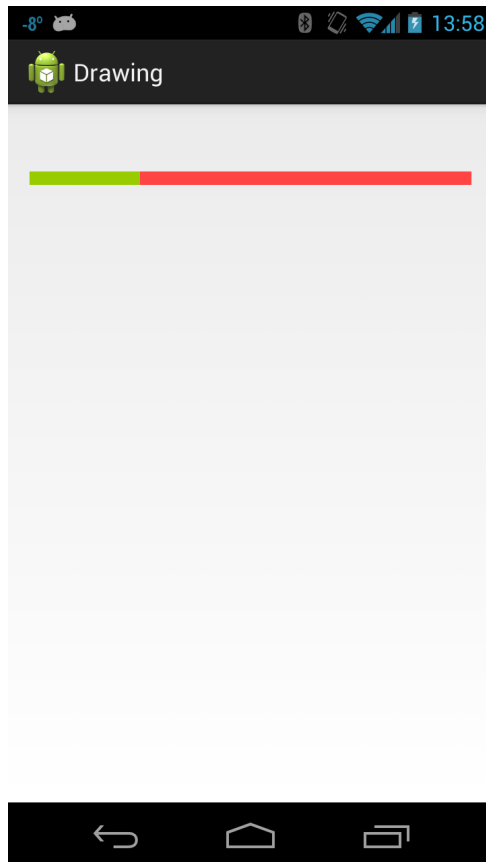
3. At this point we have successfully implemented the first Activity. If we run the application, and select Shape Drawable from the the main activity, we should see the following screenshot:



## Drawing

This next example will show an example of how to use the Canvas object to draw a progress bar on the screen. It will display a value in a horizontal bar in green, and fill the unused portion of the horizontal bar in red. Think of this bar as displaying karma – the amount of good karma displayed is in green. The value will be a value from 0 to 1.0. The following screenshot is an example of what we will build:





This screenshot shows our karma with a value of 0.25 – displayed on the left in green. The “unused” part of our bar is displayed on the right and in red.

1. For this next screen, let's begin by adding a new class to the project called KarmaMeter. Edit the class so it resembles the following code:

```
public class KarmaMeter : View
{
    private const int DefaultHeight = 20;
    private const int DefaultWidth = 120;

    private Paint _negativePaint;
    private double _position = 0.5;
    private Paint _positivePaint;

    public KarmaMeter(Context context, IAttributeSet attrs)
        : this(context, attrs, 0)
    {
        Initialize();
    }

    public KarmaMeter(Context context, IAttributeSet attrs, int
defStyle)
        : base(context, attrs, defStyle)
    {
        Initialize();
    }
}
```

```

public double KarmaValue
{
    get { return _position; }
    set
    {
        _position = Math.Max(0f, Math.Min(value, 1f));
        Invalidate();
    }
}

protected override void OnDraw(Canvas canvas)
{
    base.OnDraw(canvas);
    float middle = canvas.Width * (float)_position;
    canvas.DrawPaint(_negativePaint);
    canvas.DrawRect(0, 0, middle, canvas.Height,
        _positivePaint);
}

protected override void OnMeasure(int widthMeasureSpec, int
heightMeasureSpec)
{
    int width = MeasureSpec.GetSize(widthMeasureSpec);
    SetMeasuredDimension(width < DefaultWidth ? DefaultWidth :
width, DefaultHeight);
}

private void Initialize()
{
    _positivePaint = new Paint
    {
        AntiAlias = true,
        Color = Color.Rgb(0x99, 0xcc, 0),
    };
    _positivePaint.SetStyle(Paint.Style.FillAndStroke);

    _negativePaint = new Paint
    {
        AntiAlias = true,
        Color = Color.Rgb(0xff, 0x44, 0x44)
    };
    _negativePaint.SetStyle(Paint.Style.FillAndStroke);
}
}

```

There is a lot going on with this class, so let's explain what is going here. The first thing is that this new class extends `Android.Views.View`. We created the necessary constructors. The constructors will initialize two `Paint` objects: `_positivePaint` (green in colour) and `_negativePaint` (red in colour).

This class provides a property called `KarmaValue`. This allows users of our view to set the karma to be displayed. After the karma value is updated, our class will call `Invalidate()` – this signals Android to redraw the view when it can.

When Android determines it is time to redraw the view, it will invoke `OnDraw` and pass in a `Canvas`. `OnDraw` will first call `canvas.DrawPaint()`, which will fill the canvas with red (the colour of `_negativePaint`). After figuring out the value of `middle`, `canvas.DrawRect` is called which will draw a green rectangle to display the amount of good karma.

The final thing to note is that the `OnMeasure` method is overridden as well. This is done to ensure that the View is not narrower than the minimum width as specified by the value of `DefaultWidth` and that the View is always the size of the value `DefaultHeight`.

2. The next thing we need is to create a layout file to display the `KarmaMeter`. Add a new layout file to the project called `Resources/Layout/activity_drawing.xml` with the following contents:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <com.xamarin.evolve2013.animationsdemo.KarmaMeter
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="16dp"
        android:layout_marginRight="16dp"
        android:id="@+id/karmaMeter"
        android:layout_gravity="center"
        android:layout_marginTop="50dp"
        android:layout_marginBottom="25dp" />
</LinearLayout>
```

The important thing to note in this layout is how we declare the view in XML – we used the fully qualified package name (which is also the namespace of our View subclass) as the name of the element.

3. Next update the `OnCreate` method of the `DrawingActivity` class so that it resembles the following code snippet:

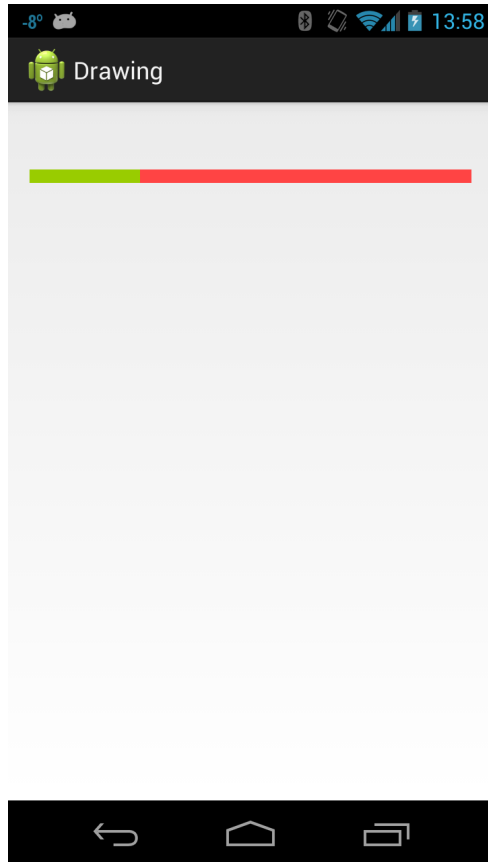
```
public class DrawingActivity : Activity
{
    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);
        SetContentView(Resource.Layout.activity_drawing);

        KarmaMeter meter =
            FindViewById<KarmaMeter>(Resource.Id.karmaMeter);

        meter.KarmaValue = 0.25;
    }
}
```

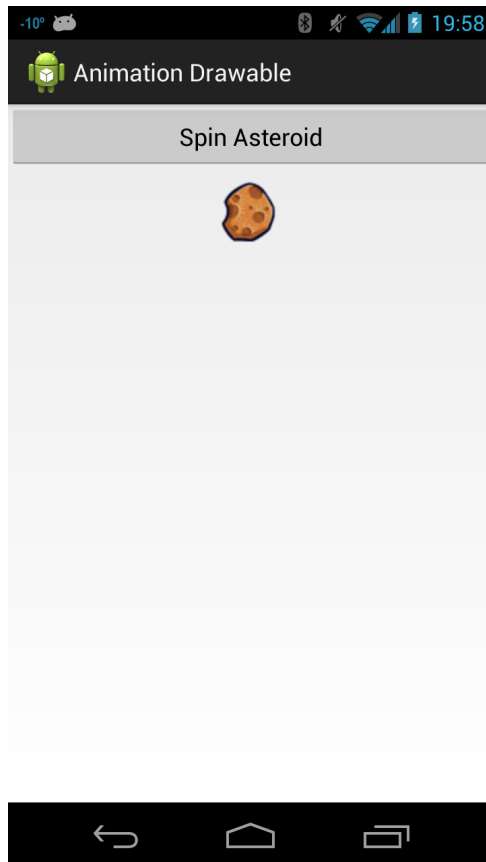
This will cause `DrawingActivity` to inflate the layout file we created in the previous step, and then set the value of the `KarmaMeter` to 0.25

4. Run the application, and select launch the **Drawing** item in the main menu. You should see a screen similar to the following screen shot:



## Animation Drawable

This screen will show how to create an animation drawable. The following screenshot shows the activity before the animation starts. When the user clicks on a button, it will kick off an animation that rotates an asteroid.



Lets get started on the next

1. To get started, we first need to create the Animation Drawable. The images that we will use to create our animation have all ready been added to the project. Create a new XML file called `Resources/Drawable/spinning_asteroid.xml` with the following contents:

```
<?xml version="1.0" encoding="UTF-8" ?>
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
>
    <item android:drawable="@drawable/asteroid01"
android:duration="100" />
    <item android:drawable="@drawable/asteroid02"
android:duration="100" />
    <item android:drawable="@drawable/asteroid03"
android:duration="100" />
    <item android:drawable="@drawable/asteroid04"
android:duration="100" />
    <item android:drawable="@drawable/asteroid05"
android:duration="100" />
    <item android:drawable="@drawable/asteroid06"
android:duration="100" />
</animation-list>
```

2. The next change we need to make is to create the layout file that will be inflated by `AnimationDrawableActivity`. Add the file `Resources/Drawable/activity_imageandbutton.axml` to the project with the following contents:

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <Button
        android:layout_gravity="center_horizontal"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/button1" />
    <ImageView
        android:layout_gravity="center_horizontal"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/imageView1" />
</LinearLayout>

```

3. Now lets update `AnimationDrawableActivity` so that it will load the drawable that we just created, and apply it to the `ImageView` in the layout to start the animation. Edit the method `AnimationDrawableActivity.onCreate` so that it matches the following code snippet:

```

protected override void onCreate(Bundle bundle)
{
    base.onCreate(bundle);
    setContentView(Resource.Layout.activity_imageandbutton);

    // Load the animation from resources
    _asteroidDrawable =
    (AnimationDrawable)Resources.getDrawable(Resource.Drawable.spinning_astero
    id);

    ImageView imageView =
    findViewById<ImageView>(Resource.Id.imageView1);
    imageView.setImageDrawable(_asteroidDrawable);

    Button spinAsteroidButton =
    findViewById<Button>(Resource.Id.button1);
    spinAsteroidButton.Text =
    Resources.getString(Resource.String.title_spinasteroid);
    spinAsteroidButton.Click += (sender, args) =>
    _asteroidDrawable.Start();
}

```

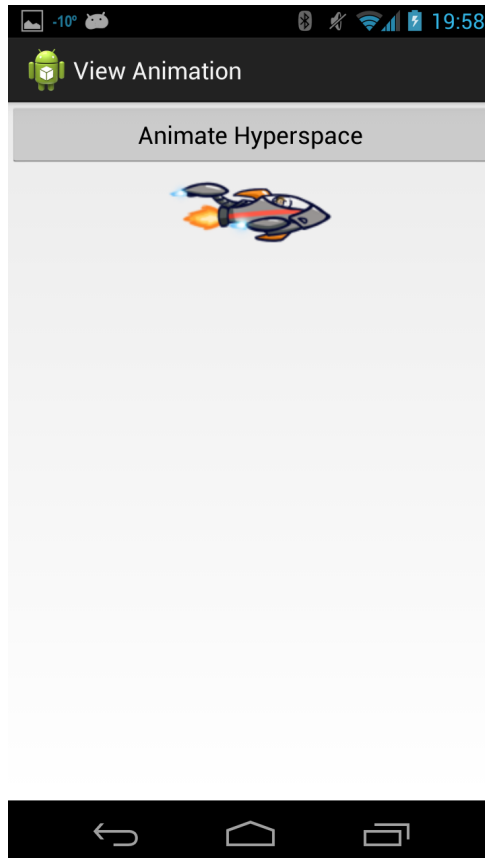
To begin with, we load the drawable animation, and cast it to an `AnimationDrawable` instance. When the user clicks the button, we will invoke the `start()` method on the resource, which will start the animation.

4. Run the application, and run the **Drawable Animation** activity from the main menu. When you click the button, you should see the asteroid spinning.

At this point we've covered a lot of ground – but we've saved the best for last. Lets move on to the final sections on the animation API's.

## View Animation

Next let's take a look at the older animation API's and learn about view animations. In this example, the user will click a button to start the animation. The animation will distort a spaceship graphic to make it look like it is starting some faster-than-light travel via hyperspace. The following screenshot shows the screen before the button is clicked.



1. For this example we will store the view animation in an XML file. Create a new file called `/Resource/anim/hyperspace.xml` with the following contents:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android=http://schemas.android.com/apk/res/android
    android:shareInterpolator="false">

    <scale
        android:interpolator="@android:anim/accelerate_decelerate_i
nterpolator"
        android:fromXScale="1.0"
        android:toXScale="1.4"
        android:fromYScale="1.0"
        android:toYScale="0.6"
        android:pivotX="50%"
        android:pivotY="50%"
        android:fillEnabled="true"
        android:fillAfter="false">
```

```

        android:duration="700" />

<set android:interpolator="@android:anim/accelerate_interpolator">
    <scale
        android:fromXScale="1.4"
        android:toXScale="0.0"
        android:fromYScale="0.6"
        android:toYScale="0.0"
        android:pivotX="50%"
        android:pivotY="50%"
        android:fillEnabled="true"
        android:fillBefore="false"
        android:fillAfter="true"
        android:startOffset="700"
        android:duration="400" />

    <rotate
        android:fromDegrees="0"
        android:toDegrees="-45"
        android:toYScale="0.0"
        android:pivotX="50%"
        android:pivotY="50%"
        android:fillEnabled="true"
        android:fillBefore="false"
        android:fillAfter="true"
        android:startOffset="700"
        android:duration="400" />

</set>
</set>

```

This file describes the view animations that Android must perform on the `ImageView`. The first animation will stretch the `ImageView`, making it long and skinny. The second animation is in a set and will do two things simultaneously – it will shrink the image into nothing while rotating it 45 degrees counter-clockwise.

2. Next, edit `ViewAnimationActivity` and change the `onCreate` method to look like the following code snippet:

```

protected override void onCreate(Bundle bundle)
{
    base.onCreate(bundle);
    setContentView(Resource.Layout.activity_imageandbutton);

    FindViewById<ImageView>(Resource.Id.imageView1).SetImageResource(Resource.Drawable.ship2_2);
    Button button = FindViewById<Button>(Resource.Id.button1);
    button.Text =
        Resources.GetString(Resource.String.title_hyperspace);
    button.Click += (sender, args) =>
    {
        Animation hyperspaceAnimation =
            AnimationUtils.LoadAnimation(this, Resource.Animation.hyperspace);
        FindViewById<ImageView>(Resource.Id.imageView1).StartAnimation(hyperspaceAnimation);
    };
}

```



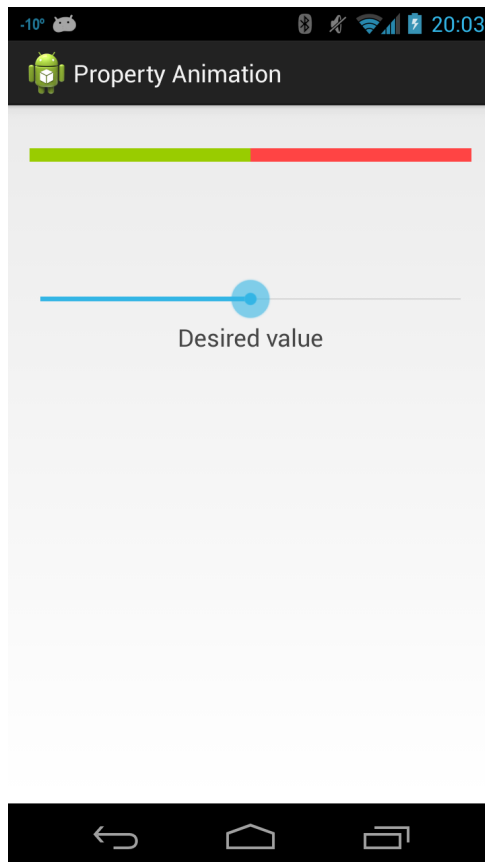
When the button is clicked, it will load the animation using the helper class `AnimationUtils`. The animation is then started with a call to `ImageView.StartAnimation` and providing the `Animation` that was loaded.

Notice that we didn't have to create a new layout for this activity – it is reusing the layout `activity_imageandbutton.xml` that we created in the previous section.

3. Run the application again, and select the **View Animation** item from the Main Activity. When you click the Animate Hyperspace button you should see the animation happening.

## Property Animation

This final screen will demonstrate how to use the newer animation API's in Android. This example will only work for API level 11 (Honeycomb) or higher. We will animate the property `KarmaMeter.KarmaValue` to demonstrate property animations. You can see what this activity will look like in the screenshot below:



When you move the **Desired value** SeekBar, the value displayed in the `KarmaMeter` will be updated in an animated fashion. Now that we have an understanding of what we are trying to do, let's get started with making the changes.

1. The first thing we need is a layout for our activity. Create a new layout file called `Resources/Layout/activity_propertyanimation.xml` with the following contents:

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <FrameLayout
        android:minWidth="25px"
        android:minHeight="160px"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/frameLayout1"
        android:layout_marginTop="32dp"
        android:layout_marginBottom="16dp">
        <com.xamarin.evolve2013.animationsdemo.KarmaMeter
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:layout_marginLeft="16dp"
            android:layout_marginRight="16dp"
            android:id="@+id/karmaMeter" />
    </FrameLayout>
    <SeekBar
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/karmaSeeker"
        android:layout_marginLeft="8dp"
        android:layout_marginRight="8dp"
        android:max="100"
        android:progress="50" />
    <TextView
        android:text="Desired value"
        android:textAppearance="?android:attr/textAppearanceMedium"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/textView1"
        android:gravity="center"
        android:textColor="#ff474747" />
</LinearLayout>

```

This layout file should be pretty straightforward. Notice the syntax for adding our custom `KarmaMeter` view to the layout.

- Design guidelines dictate that properties shouldn't perform any complicated logic (such as animation). So let's add a new method to `KarmaMeter` that will update the value of `KarmaMeter` and optionally animate the change. This will allow consumers of the `KarmaMeter` class to just update its value, or animate the change in value as they see fit. Open the file `KarmaMeter.cs`, and add the following method:

```

public void SetKarmaValue(double value, bool animate)
{
    if (!animate)
    {
        KarmaValue = value;
        return;
    }
}

```

```

    }

    ValueAnimator animator = ValueAnimator.OfFloat((float)_position,
(float)Math.Max(0f, Math.Min(value, 1f)));
    animator.SetDuration(500);

    animator.Update += (sender, e) => KarmaValue =
(double)e.Animation.AnimatedValue;
    animator.Start();
}

```

This method will create a new `ValueAnimator` object that will take 500 milliseconds to change the value of the `KarmaMeter` from its current value to the new value that is provided. When the `ValueAnimator` raises its `Update` event, the event handler will update the value of the `KarmaMeter`.

3. Next we want to update `PropertyAnimationActivity` so that it will display our layout and update the `KarmaMeter` with the value. Edit the class `PropertyAnimationActivity` so that it resembles the following code snippet:

```

public class PropertyAnimationActivity : Activity
{
    private KarmaMeter _karmaMeter;
    private SeekBar _seekBar;

    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);
        SetContentView(Resource.Layout.activity_propertyanimation);
        _seekBar = FindViewById<SeekBar>(Resource.Id.karmaSeeker);
        _seekBar.StopTrackingTouch += (sender, args) =>
        {
            double karmaValue = ((double)_seekBar.Progress) /
_seekBar.Max;
            _karmaMeter.SetKarmaValue(karmaValue, true);
        };
        _karmaMeter =
FindViewById<KarmaMeter>(Resource.Id.karmaMeter);
    }
}

```

The event handler assigned to the `SeekBar.StopTrackingTouch` event will invoke the method `KarmaMeter.SetKarmaValue`, which will animate the change in the value for the user.

4. With these changes in place, run the application again. Select the `Property Animation` activity from the `Main Menu`, and change the value of the `SeekBar`. The `KarmaBar` should now have a smooth animated change in the value it is displaying.

At this point, we are done the walkthrough. We covered a lot of ground in a short period of time, so take a break and come back to look things over again.

## Summary

---

This was a very dense chapter that introduced a lot of new concepts and API's to help add some graphic zip to an Android application. First it discussed the various graphics API's and demonstrated how Android allows applications to draw directly using a Canvas object. We also saw some alternate techniques that allow graphics to be declaratively created using XML files. Then this chapter moved on to discuss the old and new API's for creating animations in Android. Finally we finished up with a walk through that demonstrated some of the different ways to do graphics and animations in Android.