Animation and Graphics

Ref

http://developer.android.com/

http://www.tutorialspoint.com/android/android_mediaplayer.htm

Animation

- Android provides a variety of powerful APIs for applying animation to UI elements and drawing custom 2D and 3D graphics.
- The Android framework provides two animation systems:
  - property animation (introduced in Android 3.0)
  - view animation.
- Both animation systems are viable options, but the property animation system, in general, is the preferred method to use, because it is more flexible and offers more features.
- In addition to these two systems, you can utilize Drawable animation
  - it allows you to load drawable resources and display them one frame after another.

Animation Types in Android

- **Property Animation**
  - Introduced in Android 3.0 (API level 11), the property animation system lets you animate properties of any object, including ones that are not rendered to the screen. The system is extensible and lets you animate properties of custom types as well.
- **View Animation**
  - View Animation is the older system and can only be used for Views. It is relatively easy to setup and offers enough capabilities to meet many application's needs.
- **Drawable Animation**
  - Drawable animation involves displaying Drawable resources one after another, like a roll of film. This method of animation is useful if you want to animate things that are easier to represent with Drawable resources, such as a progression of bitmaps.

2D and 3D Graphics

- When writing an application, it's important to consider exactly what your graphical demands will be.
- graphics and animations for a rather static application should be implemented much differently than graphics and animations for an interactive game.
- Here, we'll discuss a few of the options you have for drawing graphics on Android and which tasks they're best suited for.

Canvas and Drawables

- Android provides a set of View widgets that provide general functionality for a wide array of user interfaces.
- You can also extend these widgets to modify the way they look or behave. In addition, you can do your own custom 2D rendering using the various drawing methods contained in the Canvas class or create Drawable objects for things such as textured buttons or frame-by-frame animations.

2D and 3D Graphics

Hardware Acceleration

- Beginning in Android 3.0, you can hardware accelerate the majority of the drawing done by the Canvas APIs to further increase their performance.

OpenGL

- Android supports OpenGL ES 1.0 and 2.0, with Android framework APIs as well as natively with the Native Development Kit (NDK).
- Using the framework APIs is desireable when you want to add a few graphical enhancements to your application that are not supported with the Canvas APIs, or if you desire platform independence and don't demand high performance.
- There is a performance hit in using the framework APIs compared to the NDK, so for many graphic intensive applications such as games, using the NDK is beneficial

Property Animation

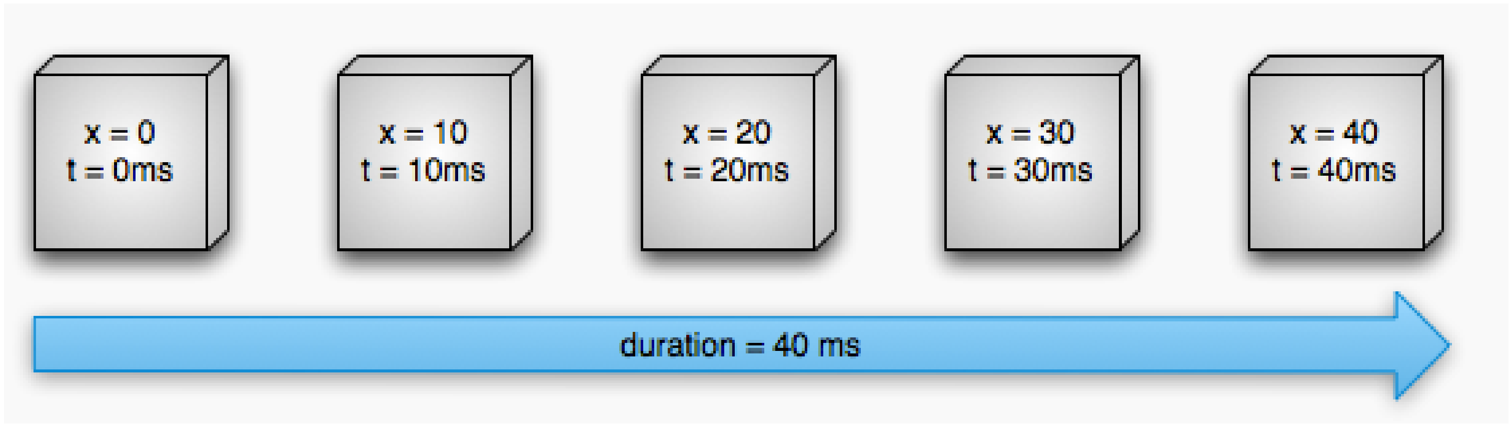
- The property animation system is a robust framework that allows you to animate almost anything. You can de
- fine an animation to change any object property over time, regardless of whether it draws to the screen or not.
- A property animation changes a property's (a field in an object) value over a specified length of time.
- To animate something, you specify the object property that you want to animate, such as an object's position on the screen, how long you want to animate it for, and what values you want to animate between.

Property Animation...

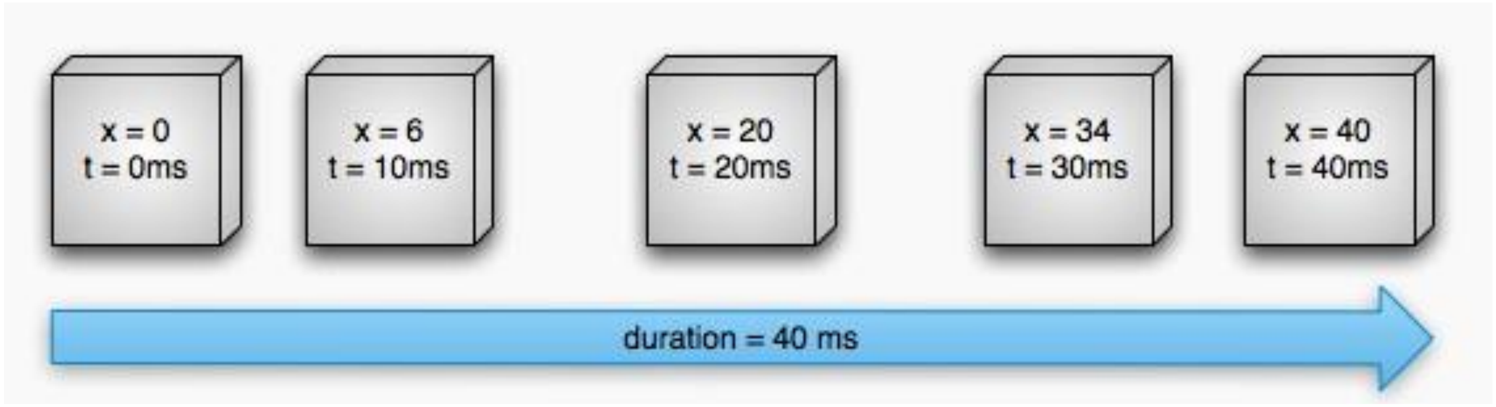
The property animation system lets you define the following characteristics of an animation:

- **Duration:** You can specify the duration of an animation. The default length is 300 ms.
- **Time interpolation:** You can specify how the values for the property are calculated as a function of the animation's current elapsed time.
- **Repeat count and behavior:** You can specify whether or not to have an animation repeat when it reaches the end of a duration and how many times to repeat the animation. You can also specify whether you want the animation to play back in reverse. Setting it to reverse plays the animation forwards then backwards repeatedly, until the number of repeats is reached.
- **Animator sets**: You can group animations into logical sets that play together or sequentially or after specified delays.
- **Frame refresh delay:** You can specify how often to refresh frames of your animation. The default is set to refresh every 10 ms, but the speed in which your application can refresh frames is ultimately dependent on how busy the system is overall and how fast the system can service the underlying timer.

How property Animation Works

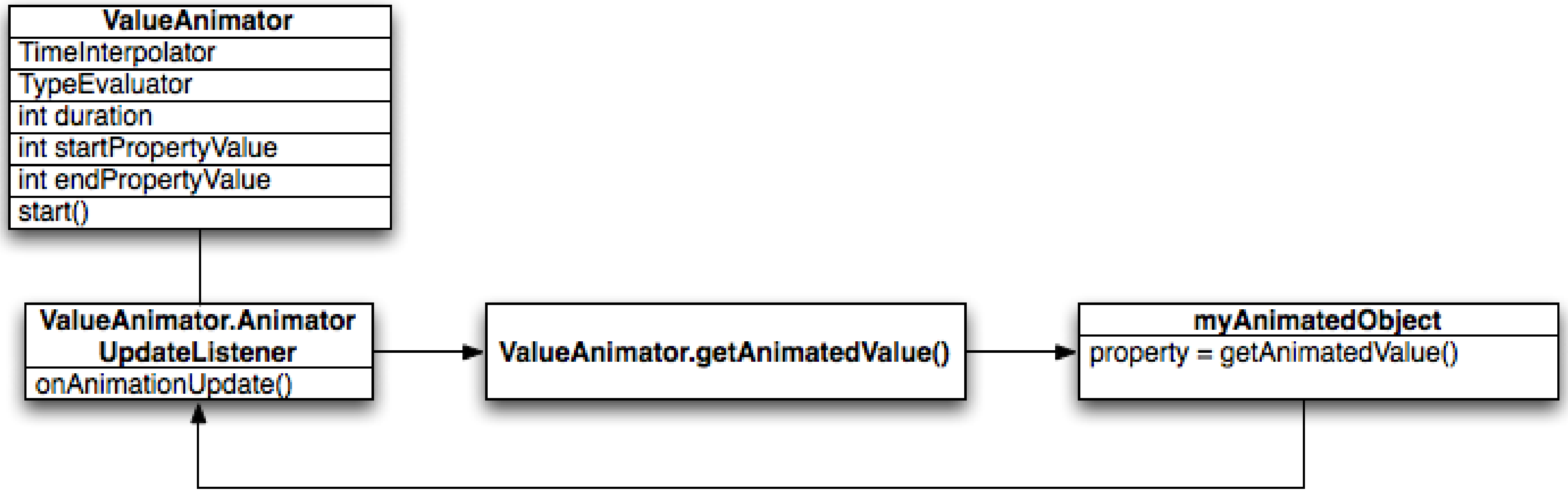


- Linear Animation



- Non- Linear Animation

How Animation Are Calculated in Android



How Animation Are Calculated in Android

- The **ValueAnimator** object keeps track of your animation's timing, such as how long the animation has been running, and the current value of the property that it is animating.
- The **ValueAnimator** encapsulates a
  - **Time Interpolator,** which defines animation interpolation
    - For example, in Figure 2, the **TimeInterpolator** used would be **AccelerateDecelerateInterpolator**
  - **TypeEvaluator,** which defines how to calculate values for the property being animated.
    - For example, in Figure 2,the **TypeEvaluator** would be **IntEvaluator.**

- To start an animation, create a ValueAnimator and give it the starting and ending values for the property that you want to animate, along with the duration of the animation. When you call start() the animation begins. During the whole animation, the ValueAnimator calculates an *elapsed fraction* between 0 and 1, based on the duration of the animation and how much time has elapsed. The elapsed fraction represents the percentage of time that the animation has completed, 0 meaning 0% and 1 meaning 100%. For example, in Figure 1, the elapsed fraction at t = 10 ms would be .25 because the total duration is t = 40 ms.
- When the ValueAnimator is done calculating an elapsed fraction, it calls the TimeInterpolator that is currently set, to calculate an *interpolated fraction*. An interpolated fraction maps the elapsed fraction to a new fraction that takes into account the time interpolation that is set. For example, in Figure 2, because the animation slowly accelerates, the interpolated fraction, about .15, is less than the elapsed fraction, .25, at t = 10 ms. In Figure 1, the interpolated fraction is always the same as the elapsed fraction.
- When the interpolated fraction is calculated, ValueAnimator calls the appropriate TypeEvaluator, to calculate the value of the property that you are animating, based on the interpolated fraction, the starting value, and the ending value of the animation. For example, in Figure 2, the interpolated fraction was .15 at t = 10 ms, so the value for the property at that time would be .15 X (40 - 0), or 6.

View Animation

- You can use the view animation system to perform tweened animation on Views.
- Tween animation calculates the animation with information such as the start point, end point, size, rotation, and other common aspects of an animation.
- A tween animation can perform a series of simple transformations (position, size, rotation, and transparency) on the contents of a View object.
  - Example 1: if you have a TextView object, you can move, rotate, grow, or shrink the text.
  - Example 2: If it has a background image, the background image will be transformed along with the text.
- A sequence of animation instructions defines the tween animation, defined by either XML or Android code.
  - an XML file is recommended because it's more readable, reusable, and swappable than hard-coding the animation.

View Animation

- The animation XML file belongs in the res/anim/ directory of your Android project.
- The file must have a single root element: this will be either a single <alpha>, <scale>, <translate>, <rotate>, interpolator element, or <set>element that holds groups of these elements (which may include another <set>).
- By default, all animation instructions are applied simultaneously.
- To make them occur sequentially, you must specify the **startOffsetattribute**, as shown in the example below.

- The following XML from one of the ApiDemos is used to stretch, then simultaneously spin and rotate a View object.

```
<set 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:fillAfter="false"
        android:duration="700" />
    <set android:interpolator="@android:anim/decelerate_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:startOffset="700"
           android:duration="400"
           android:fillBefore="false" />
        <rotate
           android:fromDegrees="0"
           android:toDegrees="-45"
           android:toYScale="0.0"
           android:pivotX="50%"
           android:pivotY="50%"
           android:startOffset="700"
           android:duration="400" />
    </set>
</set>
```

Drawable Animation

- Drawable animation lets you load a series of Drawable resources one after another to create an animation.
- This is a traditional animation in the sense that it is created with a sequence of different images, played in order, like a roll of film.
- The AnimationDrawable class is the basis for Drawable animations.
- While you can define the frames of an animation in your code, using the AnimationDrawable class API, it's more simply accomplished with a single XML file that lists the frames that compose the animation.
- The XML file for this kind of animation belongs in the res/drawable/ directory of your Android project.
- The XML file consists of an <animation-list> element as the root node and a series of child <item> nodes that each define a frame: a drawable resource for the frame and the frame duration.

Drawable Animation

- Here's an example XML file for a Drawable animation:

```
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="true">
    <item android:drawable="@drawable/rocket_thrust1" android:duration="200"  />
    <item android:drawable="@drawable/rocket_thrust2" android:duration="200"  />
    <item android:drawable="@drawable/rocket_thrust3" android:duration="200"  />
</animation-list>
```

- This animation runs for just three frames.
- By setting the android:oneshot attribute of the list to true, it will cycle just once then stop and hold on the last frame. If it is set false then the animation will loop.
- With this XML saved as rocket_thrust.xml in the res/drawable/ directory of the project, it can be added as the background image to a View and then called to play.

Drawable Animation

- Here's an example Activity, in which the animation is added to an ImageView and then animated when the screen is touched:
-
```
AnimationDrawable rocketAnimation;

public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  ImageView rocketImage = (ImageView) findViewById(R.id.rocket_image);
  rocketImage.setBackgroundResource(R.drawable.rocket_thrust);
  rocketAnimation = (AnimationDrawable) rocketImage.getBackground();
}

public boolean onTouchEvent(MotionEvent event) {
  if (event.getAction() == MotionEvent.ACTION_DOWN) {
    rocketAnimation.start();
    return true;
  }
  return super.onTouchEvent(event);
}
```

Drawable Animation

- It's important to note that the start() method called on the AnimationDrawable cannot be called during theonCreate() method of your Activity, because the AnimationDrawable is not yet fully attached to the window.
- If you want to play the animation immediately, without requiring interaction, then you might want to call it from the onWindowFocusChanged() method in your Activity, which will get called when Android brings your window into focus.

Canvas and Drawables

- The Android framework APIs provides a set of 2D-drawing APIs that allow you to render your own custom graphics onto a canvas or to modify existing Views to customize their look and feel. When drawing 2D graphics, you'll typically do so in one of two ways:
  - Draw your graphics or animations into a View object from your layout. In this manner, the drawing of your graphics is handled by the system's normal View hierarchy drawing process — you simply define the graphics to go inside the View.
  - Draw your graphics directly to a Canvas. This way, you personally call the appropriate class's onDraw() method (passing it your Canvas), or one of the Canvas draw...() methods (like drawPicture()). In doing so, you are also in control of any animation.

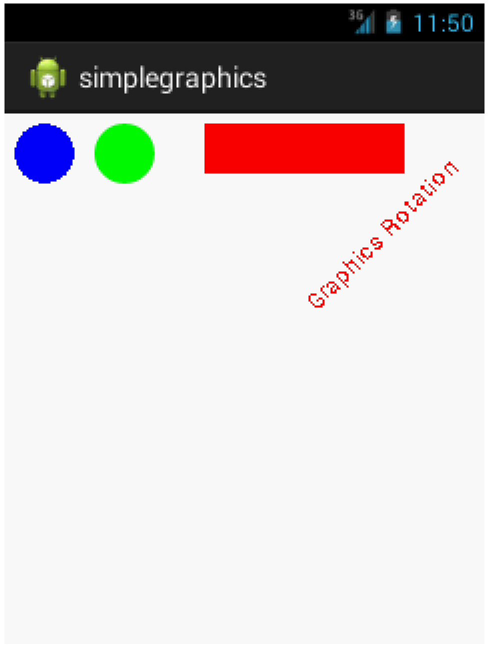
Canvas and Drawables

- Option "a," drawing to a View, is your best choice when you want to draw simple graphics that do not need to change dynamically and are not part of a performance-intensive game. For example, you should draw your graphics into a View when you want to display a static graphic or predefined animation, within an otherwise static application.
- Option "b," drawing to a Canvas, is better when your application needs to regularly re-draw itself. Applications such as video games should be drawing to the Canvas on its own. However, there's more than one way to do this:
  - In the same thread as your UI Activity, wherein you create a custom View component in your layout, callinvalidate() and then handle the onDraw() callback.
  - Or, in a separate thread, wherein you manage a SurfaceView and perform draws to the Canvas as fast as your thread is capable (you do not need to request invalidate()).

OpenGL ES

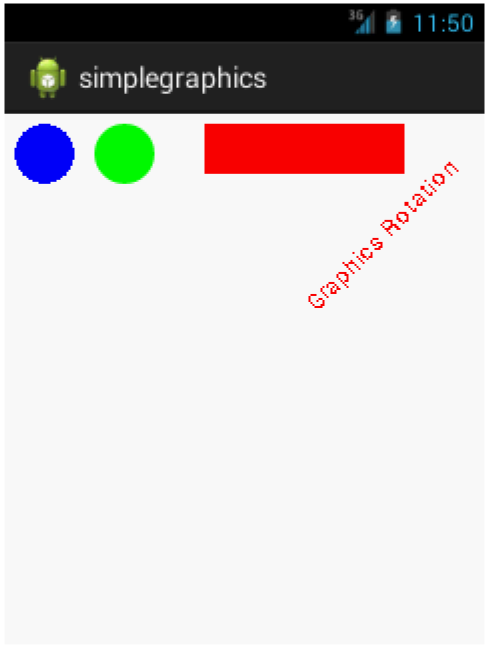
- Android includes support for high performance 2D and 3D graphics with the Open Graphics Library (OpenGL®), specifically, the OpenGL ES API.
- OpenGL is a cross-platform graphics API that specifies a standard software interface for 3D graphics processing hardware.
- OpenGL ES is a flavor of the OpenGL specification intended for embedded devices.

Problem



Using Canvas draw the following(check fig at left)

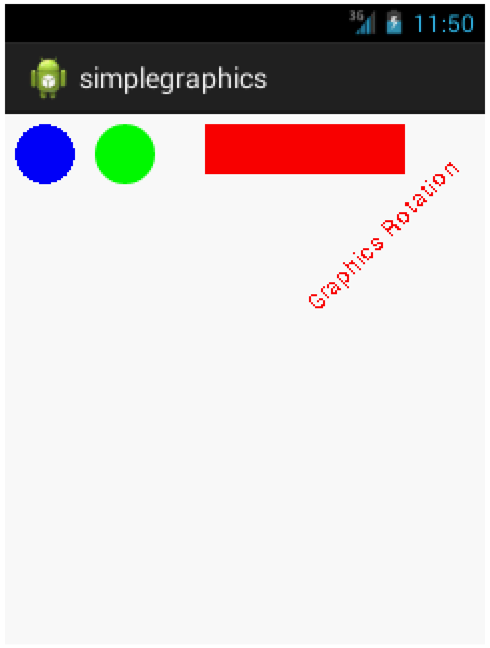
```
public class MainActivity extends Activity {
        DemoView demoview;
        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                demoview = new DemoView(this);
                setContentView(demoview);
        }
        private class DemoView extends View{
                public DemoView(Context context){
                        super(context);
                }
        protected void onDraw(Canvas canvas) {
                        super.onDraw(canvas);
                        Paint paint = new Paint();
                        paint.setStyle(Paint.Style.FILL);
                        paint.setColor(Color.WHITE);
                        canvas.drawPaint(paint);
                        paint.setAntiAlias(false);
                        paint.setColor(Color.BLUE);
                        canvas.drawCircle(20, 20, 15, paint);
                        paint.setAntiAlias(true);
                        paint.setColor(Color.GREEN);
                        canvas.drawCircle(60, 20, 15, paint);
                        paint.setAntiAlias(false);
                        paint.setColor(Color.RED);
                        canvas.drawRect(100, 5, 200, 30, paint);
                        canvas.rotate(-45);
                        paint.setStyle(Paint.Style.FILL);
                        canvas.drawText("Graphics Rotation", 40, 180, paint);
                        canvas.restore();
                }
        }
        }
```



activity_main.xml

```
<RelativeLayoutxmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:paddingBottom="@dimen/activity_vertical_margin"
android:paddingLeft="@dimen/activity_horizontal_margin"
android:paddingRight="@dimen/activity_horizontal_margin"
android:paddingTop="@dimen/activity_vertical_margin"
tools:context=".MainActivity">
<TextView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="@string/hello_world"/>

</RelativeLayout>
```



Reference

- http://developer.android.com
- http://www.tutorialspoint.com/android/android_mediaplayer.htm