Graphics Support in Android
Android and Graphics

• The Android platform supports both 2D and 3D graphics:
  – 2D graphics is powered by a custom library
  – High performance 3D graphics is based on the OpenGL ES 1.0 library

• Choice of the graphics approach within your application is driven by the need:
  – Graphics and animation for static applications
  – Graphics and animation for interactive games or 3D rendering
2D Graphics

• 2D graphics can be approached in one of two ways:
  – Using a View object: drawing of graphics handled by the system’s normal view hierarchy drawing process
    • Suitable for simple graphics that do not need to change dynamically
    • Draw to the background of a view or to an ImageView in the layout
  – Canvas: use the appropriate class’ draw() methods and passing them the canvas
    • Suitable for applications where your application needs to frequently re-draw the screen
2D Graphics with Drawables

• Make use of Drawables for drawing shapes and images to a View
  – Drawable class
  – Subclasses: BitmapDrawable, ShapeDrawable, PictureDrawable, LayerDrawable etc.
• Drawables can be defined and instantiated in three ways:
  – Using an image saved in your project resources
  – Using an XML file that defines Drawable properties
  – Using normal class constructors in code
• Drawable resources usually stored in res/drawable folder of your project
Drawables

• Drawables may take a variety of forms:
  – **Bitmap**: the simplest Drawable, a PNG or JPEG image.
  – **Nine Patch**: an extension to the PNG format allows it to specify information about how to stretch it and place things inside of it.
  – **Shape**: contains simple drawing commands instead of a raw bitmap, allowing it to resize better in some cases.
  – **Layers**: a compound drawable, which draws multiple underlying drawables on top of each other.
  – **States**: a compound drawable that selects one of a set of drawables based on its state.
  – **Levels**: a compound drawable that selects one of a set of drawables based on its level.
  – **Scale**: a compound drawable with a single child drawable, whose overall size is modified based on the current level.

• Let us examine some of these next
• Detailed information and examples can be found in Android documentation
Bitmap Drawables

• We have already seen the use of ImageView to include images into the layouts
• Also can use XML to include bitmaps with modifications into the layouts
• Example:

```xml
<?xml version="1.0" encoding="utf-8"?>
<bitmap
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:src="@drawable/icon"
    android:gravity="center"
    android:dither="true"
    android:tileMode="repeat" />
```
Shape Drawables

• Especially useful for defining various properties of some standard shapes to be included in a view background
• For example, to achieve a gradient background to a textview, define the following in a XML file (back_gradient.xml) in the res/drawable:

```xml
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle">
    <gradient
        android:startColor="#003366"
        android:endColor="#006699"
        android:angle="270"/>
</shape>

• Then set it as the background to the textview
```

```xml
<textview
    android:background="@drawable/back_gradient"
...
/>
```
StateList Drawables

• Can use several different images to represent the same graphics, depending on the state of the object
• For example, can change the background of a textview based on whether it is selected, focused, normal ...
• For example, define an XML file in the res/drawable and then use it as background of the view:

```xml
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:state_pressed="true"
        android:drawable="@drawable/highlighted" />
  <!-- pressed -->
  <item android:state_selected="true"
        android:drawable="@drawable/highlighted" />
  <!-- pressed -->
  <item android:state_focused="true"
        android:drawable="@drawable/highlighted" />
  <!-- pressed -->
  <item android:drawable="@drawable/back_gradient" />
  <!-- default -->
</selector>
```
View Animation

• View animation can be used in any View object
• Two kinds available:
  – Tween animation: animation performed based on information such as the starting point, end point, size, rotation, transparency etc..
  – Frame by frame animation: allows loading a series of Drawable resources one after another to create the animation
• Frame by frame animation is more straightforward
• Look for animation demos in the Api demos sample code and Lunar Lander sample code
View Animation: Tween Animation

- Series of simple transformations of position, size, rotation and transparency
- For example, to fade in text in a TextView object, define the following XML (fadein.xml) file in res/anim:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<set
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:shareInterpolator="false">
<alpha
 android:fromAlpha="0.0"
 android:toAlpha="1.0"
 android:duration="5000">
</alpha>
</set>
```
View Animation: Tween Animation

• Then manipulate the textview in code as follows:

```
Animation fadein = AnimationUtils.loadAnimation(this, R.anim.fade_in);
TextView text = (TextView) innerlayout.getChildAt(i);
text.setAnimation(fadein);
```
2D Drawing With Canvas

• Suited for applications which require specialized drawing and/or control of the animation of graphics
• Canvas provides the interface to the actual surface upon which graphics can be drawn
  – Drawn to a bitmap which is placed into the window
• Canvas class offers a set of drawing methods:
  – drawBitmap(), drawRect(), drawText() …
• For applications not requiring significant amount of processing or frame-rate speed, create a custom view component and draw on it with a Canvas in View.onDraw()
• SurfaceView, a subclass of View, offers a dedicated drawing surface within the View hierarchy
2D Canvas on a View

• Extend the View class and define the `onDraw()` callback method
  – Use the `onDraw()` method to perform all calls to draw on the canvas
  – Android framework calls `onDraw()` to update the screen
  – Use `invalidate()` to trigger your view to be drawn
  – Inside your View component's `onDraw()`, use the Canvas given to you for all your drawing, using various `Canvas.draw...()` methods, or other class `draw()` methods that take your Canvas as an argument
  – Once your `onDraw()` is complete, the Android framework will use your Canvas to draw a Bitmap handled by the system.
2D Canvas on SurfaceView

• SurfaceView is a special subclass of View that offers a dedicated drawing surface within the View hierarchy
  – Offers this drawing surface to an application's secondary thread, so that the application isn't required to wait until the system's View hierarchy is ready to draw
  – Instead, a secondary thread that has reference to a SurfaceView can draw to its own Canvas at its own pace.
• In your code extend the SurfaceView class
• Need to implement the SurfaceHolder.Callback
• Also the interface requires three additional methods to be implemented: surfaceCreated(), surfaceChanged(), and surfaceDestroyed()
  – Important so that you know when you can start drawing, whether you need to make adjustments based on new surface properties, and when to stop drawing and potentially kill some tasks.
2D Canvas on SurfaceView

• The surface object is handled via a SurfaceHolder which you get when you initialize the SurfaceView, by calling getHolder().

• Inform the SurfaceHolder that you will handle the callbacks by calling holder.addCallback(this). Then override the Callback methods within your SurfaceView: surfaceCreated(), surfaceChanged(), and surfaceDestroyed().

• Create a secondary thread that will control all the drawings to the surface, and pass it the SurfaceHolder:
  – Within the thread get hold of the canvas: surfaceHolder.lockCanvas()
  – Then make modifications to the canvas
  – Finally, surfaceHolder.unlockCanvasAndPost(canvas)
3D Graphics in Android and Open GL ES
3D Graphics Support in Android

• 3D graphics support in Android is provided through:
  – Open GL ES, an specialized version of the Open GL library, optimized for embedded devices
  – Renderscript providing native level high-performance 3D support
3D with Open GL ES

• Basic approach to using Open GL ES in Android:
  1. Write a custom View subclass.
  2. Obtain a handle to an OpenGLContext, which provides access to the OpenGL functionality.
  3. In your View's onDraw() method, get a handle to a GL object, and use its methods to perform GL operations.

• Basic approach is similar to the use of SurfaceView in 2D graphics, but we use the 3D Open GL library
3D with GLSurfaceView

• GLSurfaceView class makes it easier to use OpenGL ES rendering in your applications:
  – Provides the glue code to connect OpenGL ES to the Android View system
  – Provides the glue code to make OpenGL Es work with the Activity life-cycle of Android applications
  – Makes it easy to select an appropriate frame buffer pixel format
  – Creates and manages a separate rendering thread, to enable smooth animation
  – Provides easy-to-use debugging tools for tracing OpenGL Es API calls and checking for errors
The GLSurfaceView.Renderer interface has three methods:

- The onSurfaceCreated() method is called at the start of rendering, and whenever the OpenGL ES drawing context has to be recreated. (The drawing context is typically lost and recreated when the activity is paused and resumed.) OnSurfaceCreated() is a good place to create long-lived OpenGL resources such as textures.
- The onSurfaceChanged() method is called when the surface changes size. It's a good place to set your OpenGL viewport. You may also want to set your camera here, if it's a fixed camera that doesn't move around the scene.
- The onDrawFrame() method is called every frame, and is responsible for drawing the scene. You would typically start by calling glClear to clear the framebuffer, followed by other OpenGL ES calls to draw the current scene.