Intermediate Android

CSCI 4448/5448: Object-Oriented Analysis & Design
Lecture 18 — 10/25/2012

Goals of the Lecture

- Dig deeper into the Android Framework
  - Screen Orientation
  - Animation
  - Dialogs
  - Playing Sounds
  - (Simple) Networking

Android Development Philosophy

- As I learned more about Android development, I came to understand the Android Development Philosophy
  - “Everything is a Resource”
- or
  - “It’s resources all the way down…”
- Many of the steps in Android programming depend on creating resources and then loading them or referencing them (in XML files) at the right time

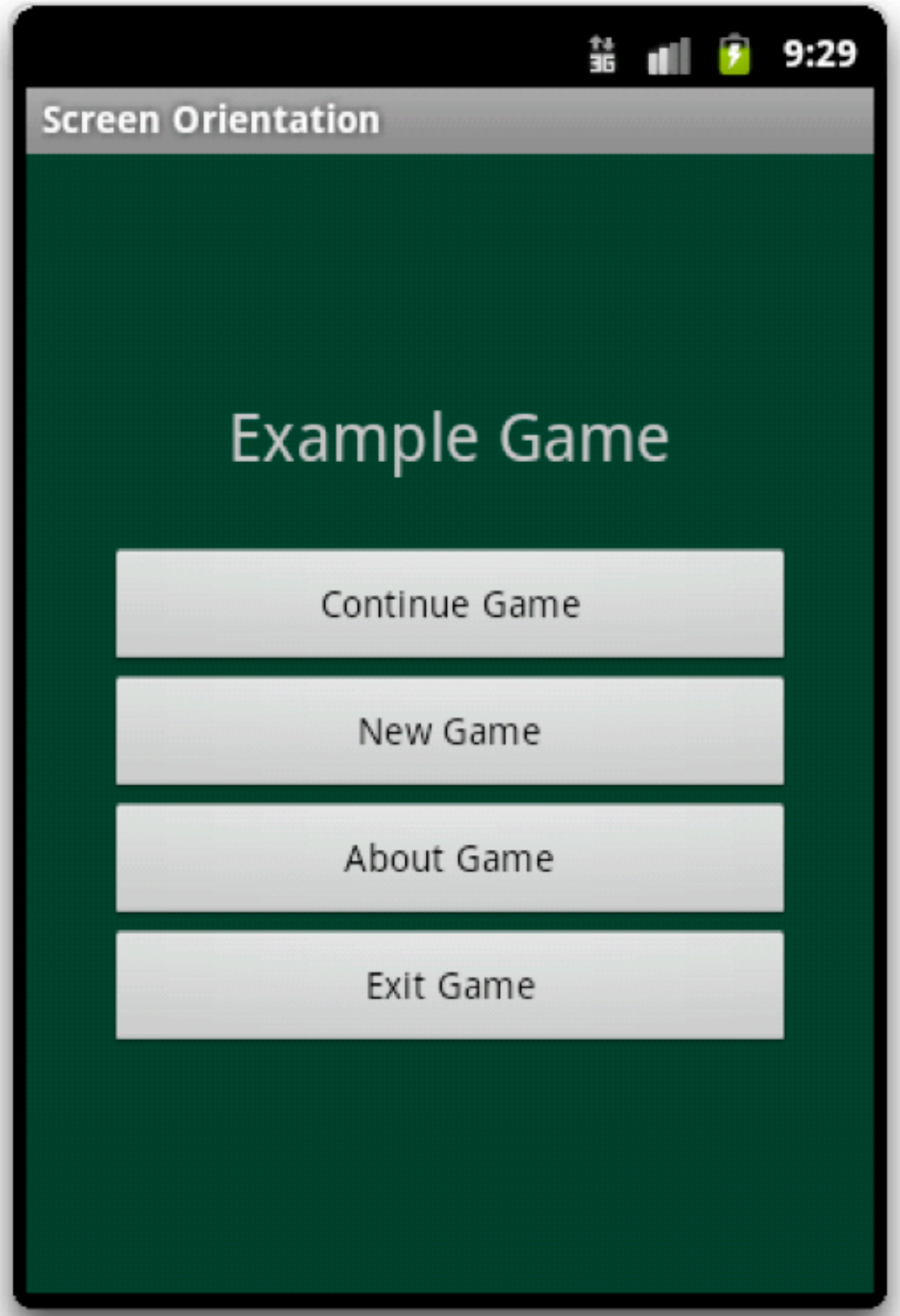
Screen Orientation

- People can easily change the orientation by which they hold their mobile devices
  - Mobile apps have to deal with changes in orientation frequently
  - We saw iOS automatic support for multiple orientations in our last lecture
  - Let's see how Android deals with this issue (hint: resources)

Start with Portrait Orientation

- It is natural to start by designing the UI of your main activity in portrait orientation
  - That is the default orientation in the Eclipse plug-in
  - Here's a typical layout for the "main screen" of a game

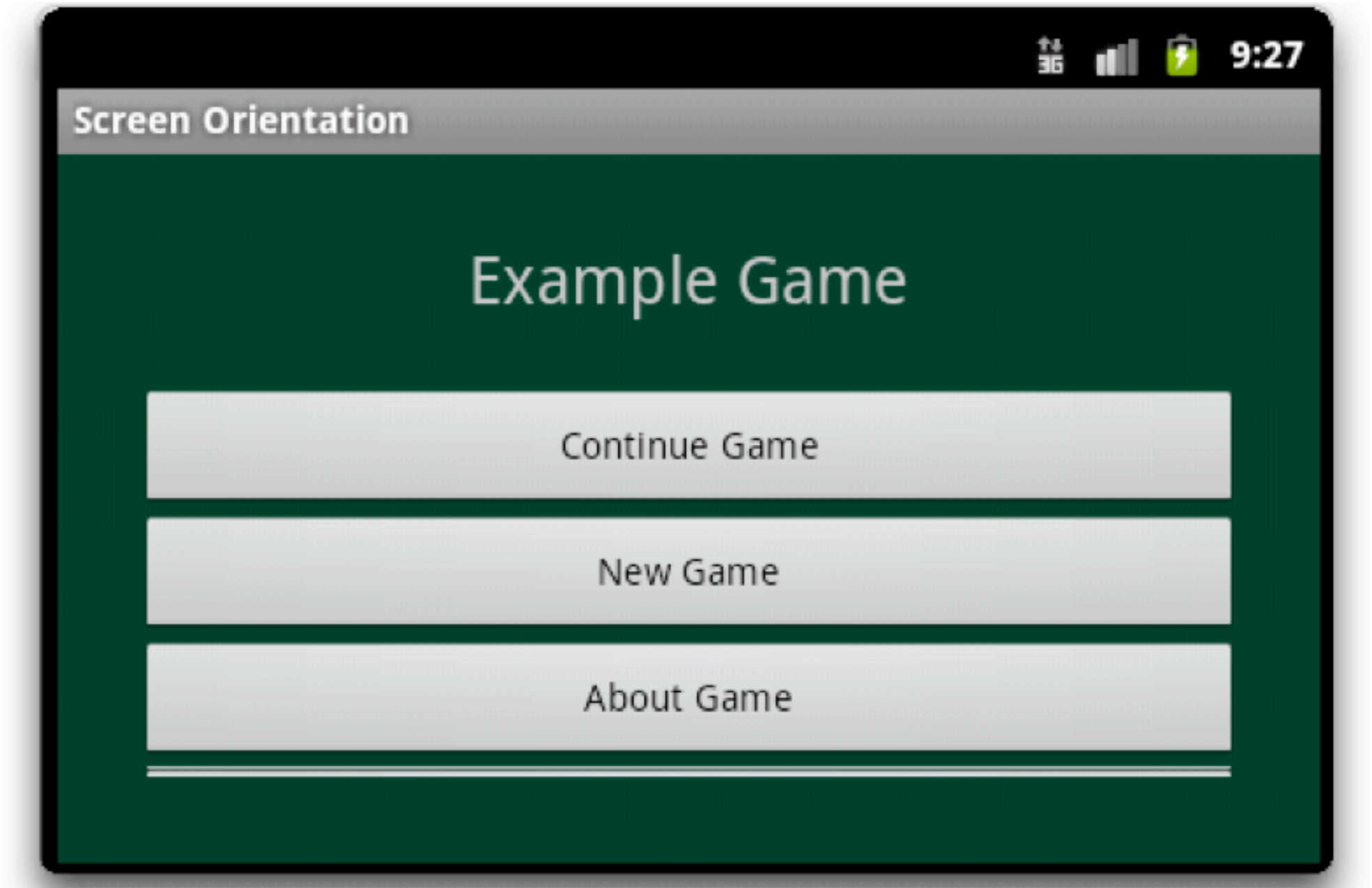
```xml
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:background="@color/background"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_gravity="center"
    android:padding="30dp">
        <TextView
            android:text="@string/main_title"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:layout_marginBottom="25dp"
            android:textSize="24.5sp" />
        <Button
            android:id="@+id/continue_button"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="@string/continue_label" />
        <Button
            android:id="@+id/new_button"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="@string/new_game_label" />
        <Button
            android:id="@+id/about_button"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="@string/about_label" />
        <Button
            android:id="@+id/exit_button"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="@string/exit_label" />
</LinearLayout>
```



Quick Interjection: Unit Sizes

- Android supports a wide variety of unit sizes for specifying UI layouts; here are all but two
    - px (device pixel), in, mm, pt (1/72nd of an inch)
- All of these have problems creating UIs that work across multiple types of devices
    - Google recommends using resolution-independent units
        - dp (or dip): density-independent pixels
        - sp: scale-independent pixels
- In particular, use sp for font sizes and dp for everything else

But switch to landscape mode (in the emulator Ctrl+F12) and a problem becomes evident

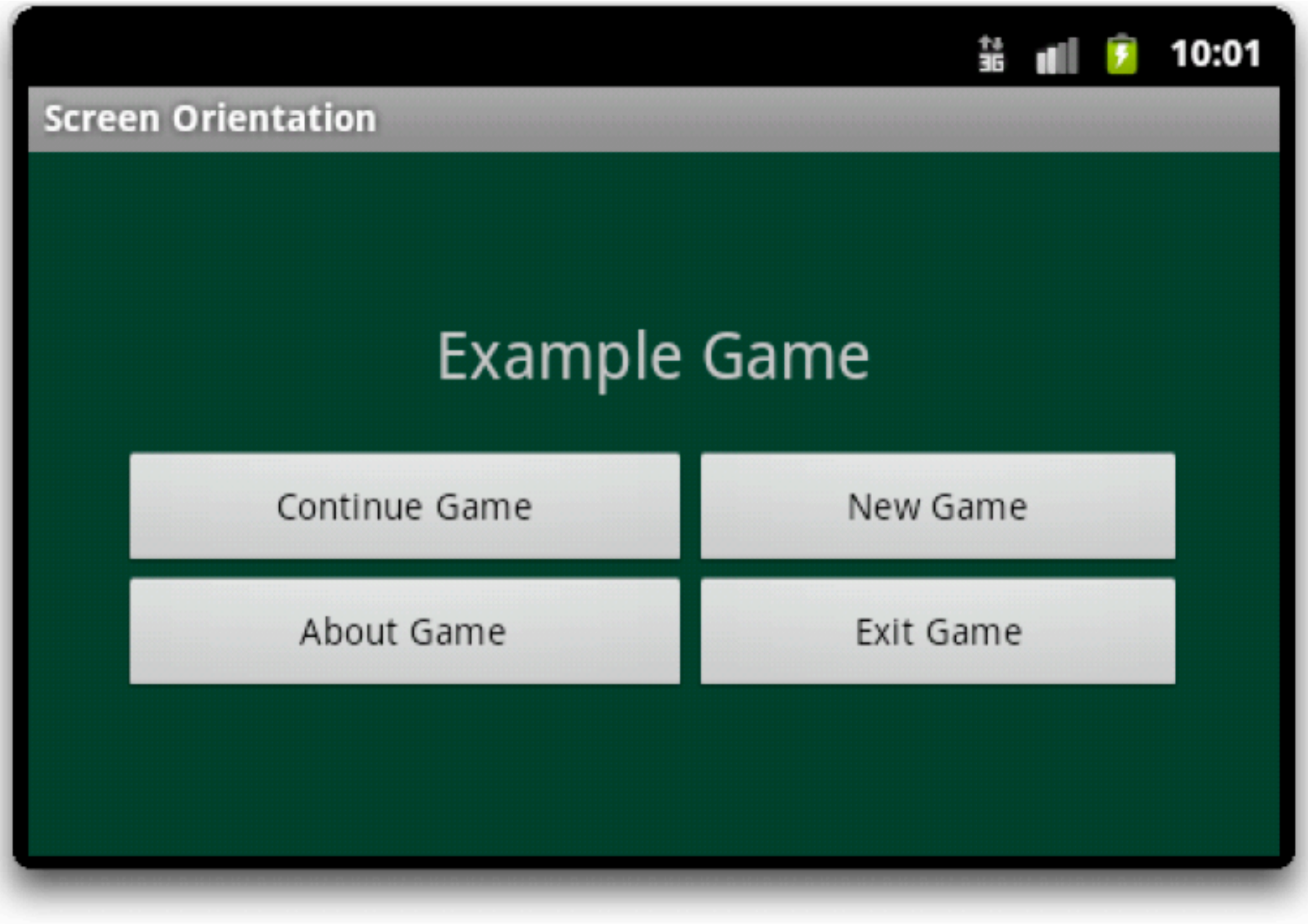


Resources to the Rescue!

- To solve this problem, we create a new activity_main.xml file that has been created specifically for landscape orientation
- This file will live in a new subfolder in the res folder of our Android project: res/layout-land/
- This folder is not created by default; right click on the res folder and select New ⇒ Folder
- Then you can right click on the existing activity_main.xml and select copy and then right click on layout-land and select paste; Finally, you can edit the file for the new orientation

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:background="@color/background"
    android:layout_height="fill_parent"
    android:layout_width="fill_parent"
    android:layout_gravity="center"
    android:paddingLeft="20dp"
    android:paddingRight="20dp"
    android:orientation="vertical" >
        <TextView
            android:text="@string/main_title"
            android:layout_height="wrap_content"
            android:layout_width="wrap_content"
            android:layout_gravity="center"
            android:layout_marginBottom="20dip"
            android:textSize="24.5sp" />
        <TableLayout
            android:layout_height="wrap_content"
            android:layout_width="wrap_content"
            android:layout_gravity="center"
            android:stretchColumns="*" >
            <TableRow>
                <Button android:id="@+id/continue_button" android:text="@string/continue_label" />
                <Button android:id="@+id/new_button" android:text="@string/new_game_label" />
            </TableRow>
            <TableRow>
                <Button android:id="@+id/about_button" android:text="@string/about_label" />
                <Button android:id="@+id/exit_button" android:text="@string/exit_label" />
            </TableRow>
        </TableLayout>
</LinearLayout>
```

This layout arranges the buttons into two rows and two columns using a TableLayout



Problem solved. Android automatically switches the layout behind the scenes when the orientation of the device changes.

Types of Layouts?

- **LinearLayout**: Each child view is placed after the previous one in a single row or column
- **RelativeLayout**: Each child view is placed in relation to other views in the layout or relative to its parent's layout
- **FrameLayout**: Each child view is stacked within a frame, relative to the top-left corner. Child views may overlap.
- **TableLayout**: Each child view is a cell in a grid of rows and columns

Specifying the Size of a View

- We've previously discussed the use of resolution-independent measurements for specifying the size of a view
  - These values go in the XML attributes
    - **android:layout_width** and **android:layout_height**
- But, you get more flexibility with
  - **fill_parent**: the child scales to the size of its parent
  - **wrap_content**: the parent shrinks to the size of the child

Animating Views

- Android offers four different ways of performing animation
  - Support for Animated GIF images
  - Frame-by-Frame animation: developer supplies images and specifies transitions between them
  - Tweened animation: simple animation effects that can be programmatically applied to views
  - OpenGL ES: advanced 3D drawing, animation, etc.

Tweened Animation

- Tweened animations are specified (unsurprisingly) via resources
- The basic process involves doing the following in the onCreate() method of the Activity
  - get a handle to the view
  - load the animation resource: such as fade
  - apply it to the view: view.startAnimation(fade)
- Android provides animation support for **alpha**, **rotation**, **scaling** and **translating**
  - the first deals with transparency; the third deals with a view's size; the last deals with moving views around

Our Plan

- We'll apply animations to the buttons defined on the portrait layout of the previous example
- We'll make one fade in, one rotate, one scale, and one that does all three at once!
    - We'll also have each animation happen one after the other
    - In a real application, this would get tedious, but as an example, it's fine

The Process (I)

- Step One: Use the New Folder command to create a folder called anim in the res folder of our project
- Step Two: Create a new Android XML File in the anim subfolder, call it fade.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="1000"
    </alpha>
</set>
```

The Process (II)

- Step 3: Add the following code to the Main activity's onCreate() method

```
Button continue_button = (Button) findViewById(R.id.continue_button);

Animation fade = AnimationUtils.loadAnimation(this, R.anim.fade);

continue_button.startAnimation(fade);
```

- You will need these import statements

```
import android.view.animation.Animation;

import android.view.animation.AnimationUtils;

import android.widget.Button;
```

The Process (IV)

- There are no additional steps… just run the program!
  - Demo of “Fun With Animation”
- As you saw from the code, we used the attribute
  - android:startOffset
- to control when particular animations start
- As you can see, Android makes it straightforward to perform simple animations within Android apps

Getting input from the user

- Android provides several types of default dialog boxes
    - and provides a way to create custom dialogs as well
- The dialog types
    - Dialog
        - the base class for all dialogs; you subclass this class to create custom dialogs
- AlertDialog: a dialog with 1-3 buttons
- DatePicker and TimePicker
- ProgressDialog (both determinate and indeterminate)

Dialog Life Cycle (I)

- Each activity manages the life cycle of the dialog boxes it displays to its users
  - It calls showDialog()to display a dialog
    - That dialog gets added to its dialog window cache
  - It calls dismissDialog() to
    - remove a dialog window
    - but keep it in the cache
    - subsequent display of the dialog is faster
- It calls removeDialog() to remove the dialog from the cache

Dialog Life Cycle (II)

- Each dialog has an associated id; you pass that id to showDialog()
  - This causes the method onCreateDialog() to be called with that id. You then use a switch statement to create the appropriate dialog based on the id
    - onCreateDialog() is typically called once; thereafter the dialog is retrieved from the cache
- The next method called is onPrepareDialog()
  - this method is called whenever the dialog is about to be shown

Example

- Let’s create an app that shows how to use
  - AlertDialog
  - DatePicker
  - TimePicker
- We’ll see the use of a ProgressDialog a little bit later
- Demo of “Fun With Dialogs”

Discussion (I)

- Code looks more complex than it actually is
  - In the onCreateDialog() method, we simultaneously create the dialogs that we need PLUS the methods that act as the dialog's event handlers
  - In the onPrepareDialog() method, we either reuse the previously set value (stored in attributes) or we set the dialog to a default value (current day and current time)

Discussion (II)

- The approach demonstrated by this code works but it is **deprecated**
- The new approach recommended by Google is documented here:
  - http://developer.android.com/guide/topics/ui/dialogs.html
- The basic difference is that you now need to create a custom subclass of DialogFragment and then use the AlertDialog.Builder and DatePickerDialog as shown in my example code
  - The reason for this change is a need to unify the user interface paradigm across phones and tablets
    - In table interfaces, you can create “fragments” of UI that appear embedded in the larger space of a table UI
      - On a phone, these same UI elements would appear as dialogs

Playing Sounds

- Android makes it very easy to play sounds
  - You copy supported sound files to res/raw
    - Just copy the file to the right place on the file system and then right click on res/raw in Eclipse and select “Refresh”
  - You create an instance of MediaPlayer
    - When you want the sound to play, you call start() and pass in the id of the sound you want; Call stop() want the sound to stop
- Demo of SoundPlayer
- Note: The included sound is public domain; I downloaded it from here:
  - http://www.mediacollege.com/downloads/sound-effects/space/

Networking (I)

- Mobile apps will often need to access a web service or web page to retrieve information that it then displays to its user
- In Android, accessing network resources must always occur in a thread that is separate from the GUI thread
    - Otherwise, the GUI thread can be blocked waiting for a remote server to respond and the user will think that the application has crashed

Networking (II)

- There is nothing magic about Android's networking
  - Your program can use any of Java's IO packages to access the internet
  - The trick is that you must run that code in a thread
- Android offers two ways of running tasks asynchronously
  - AsyncTask and Thread/Handler
  - The latter requires the developer to do all the work, so we will look at the former

Networking (III)

- AsyncTask is an abstract class that makes it straightforward to run a task in the background that also updates the GUI
- To use, you create a subclass of AsyncTask and override the following methods
  - onPreExecute() - runs on the GUI thread before the background process is started
  - doInBackground() - contains the code for the background process

Networking (IV)

- To use, you create a subclass of AsyncTask and override
  - onProgressUpdate() - runs on the GUI thread and contains information passed from the background thread; to do this, the background thread, passes information to a method called publishProgress()
  - onPostExecute() - runs on the GUI thread, once the background process is done

Networking (V)

- So, for a standard hit on a web service, you would
  - set up a progress bar in onPreExecute()
  - call the web service in doInBackground()
    - when you receive a result, loop over the contents and call publishProgress() with info
  - in onProgressUpdate() update the progress bar or update the GUI with information from the web service or both
  - make the progress bar go away in onPostExecute()

Java Feature: varargs

- The AsyncTask class makes use of Java's version of sending a method a variable number of arguments
- The syntax looks like this
  - public void process(String… args);
- Inside the method, args acts just like a Java array but defining it this way allows you to pass in any number of strings to process, be it as an array or as individual string arguments

The progress indicator

- We'll create an instance of ProgressDialog to let our user know that data is being downloaded and processed
    - Since we don't know how long the download will take, we will use an indeterminate progress indicator
        - This type of progress bar displays a spinning image to let the user know that the program hasn't crashed

Demonstration

- Let's write a simple Android client that uses AsyncTask to hit the Twitter Search API to retrieve tweets that contain the word "Android"
  - We will hit a URL that returns a list of tweets in JSON format
  - We'll parse the JSON to get the text of the tweets
  - We'll display the tweets in a list
  - We'll demonstrate the use of AsyncTask along the way
  - Note: must set android.permission.INTERNET to access the network

Discussion

- Straightforward example
  - AsyncTask works as advertised
    - creating, displaying, and dismissing progress dialog was a snap
    - very easy to send results from background thread to GUI thread
  - Makes use of some advanced Java constructs to allow a private class to access attributes and methods of its surrounding class

Wrapping Up

- Learned more about the Android framework
  - How to handle multiple orientations
  - How to handle simple animations
  - How to handle simple dialogs
  - How to play sounds
  - How to handle a simple network request (with progress bars!)

Coming Up Next

- Homework 5: Released on Monday; Due in Two Weeks
  - Need to form teams **now**, if you haven't already!