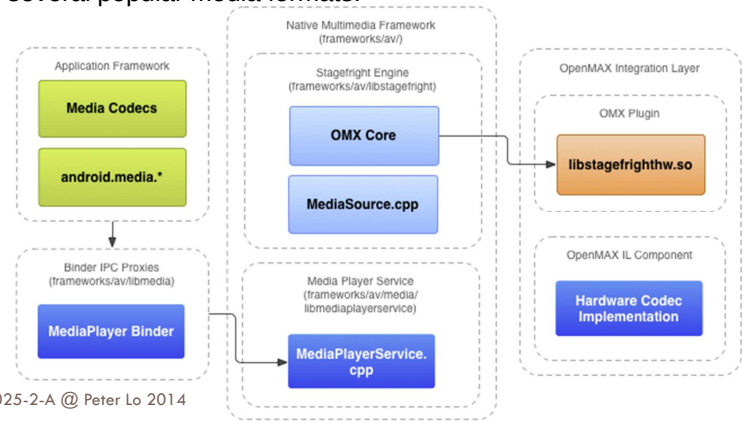
ANDROID APPS DEVELOPMENT FOR MOBILE AND TABLET DEVICE (LEVEL II)

Lecture 4: Multimedia and Open GL

Peter Lo

Media Playback Engine

- Android provides a media playback engine at the native level called Stagefright that comes built-in with software-based codecs for several popular media formats.



Playing Sound

- Android provides two main API's for playing sounds. The first one via the *SoundPool* class and the other one via the *MediaPlayer* class.
  - **SoundPool** can be used for small audio clips (<1MB). It can repeat sounds and play several sounds simultaneously.
  - **MediaPlayer** is better suited for longer music and movies.

SoundPool

- A *SoundPool* is a collection of samples that can be loaded into memory from a resource inside the APK or from a file in the file system.
- The *SoundPool* library uses the MediaPlayer service to decode the audio into a raw 16-bit PCM mono or stereo stream.
- This allows applications to ship with compressed streams without having to suffer the CPU load and latency of decompressing during playback.

Using SoundPool

Load the sound from the specified resource. You would specify "R.raw.explosion" if you want to load a sound “sample.mp3". Note that this means you cannot have both “sample.wav" and “samplen.mp3" in the res/raw directory.

Constructs a SoundPool with the maximum number of simultaneous streams and Audio stream type.

```
soundID = soundPool.load(this, R.raw.sample, 1);
soundPool = new SoundPool(10, AudioManager.STREAM_MUSIC, 0);
soundPool.setOnLoadCompleteListener(new OnLoadCompleteListener() {
    @Override
    public void onLoadComplete(SoundPool soundPool, int sampleId, int status) {
        soundPool.play(soundID, LeftVolume, RightVolume, priority, loop, rate);
    }
});
```

Define the action when complete loading.

Play the music clip
- *soundID* – Sound ID returned by the load() function
- *leftVolume* – Left volume value (0.0 ~ 1.0)
- *rightVolume* – right volume value (0.0 ~ 1.0)
- *priority* – stream priority (0 = lowest priority)
- *loop* – loop mode (0 = no loop, -1 = loop forever)
- *rate* – playback rate (0.5 ~ 2.0, 1.0 = normal playback)

MediaPlayer

- One of the most important components of the media framework is the *MediaPlayer* class.
- An object of this class can fetch, decode, and play both audio and video with minimal setup.
- It supports several different media sources such as:
  - Local resources (**raw** resources)
  - Internal URIs, such as one you might obtain from a Content Resolver
  - External URLs (streaming)

Using MediaPlayer

Constructs a MeidaPlayer

Sets the audio stream type for this MediaPlayer

Sets the data source (file-path or http/rtsp URL) of the stream you want to play

Prepares the player for playback, synchronously. After setting the datasource and the display surface, you need to either call prepare() or prepareAsync().
For files, it is OK to call prepare(), which blocks until MediaPlayer is ready for playback.

Starts or resumes playback.
- If playback had previously been paused, playback will continue from where it was paused.
- If playback had been stopped, or never started before, playback will start at the beginning.

Stops playback after playback has been stopped or paused

```
MediaPlayer mPlayer = new MediaPlayer();
mPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
mPlayer.setDataSource(url);
mPlayer.prepare();
mPlayer.start();
mPlayer.stop();
```


Volume Control

- Android supports different audio streams for different purposes.
- The phone volume button can be configured to control a specific audio stream,
  - E.g. during a call the volume button allow increase / decrease the caller volume.
- To set the button to control the sound media stream set the audio type in your application.

```
context.setVolumeControlStream(AudioManager.STREAM_MUSIC);
```

Network Protocols

- The following network protocols are supported for audio and video playback:
  - RTSP (RTP, SDP)
  - HTTP/HTTPS progressive streaming
  - HTTP/HTTPS live streaming draft protocol:
    - MPEG-2 TS media files only
    - Protocol version 3 (Android 4.0 and above)
    - Protocol version 2 (Android 3.x)
    - Not supported before Android 3.0

Manifest Declarations

- Before starting development using MediaPlayer, manifest has the appropriate declarations to allow use of related features.
  - **Internet Permission** – For using MediaPlayer to stream network-based content, the application must request network access.

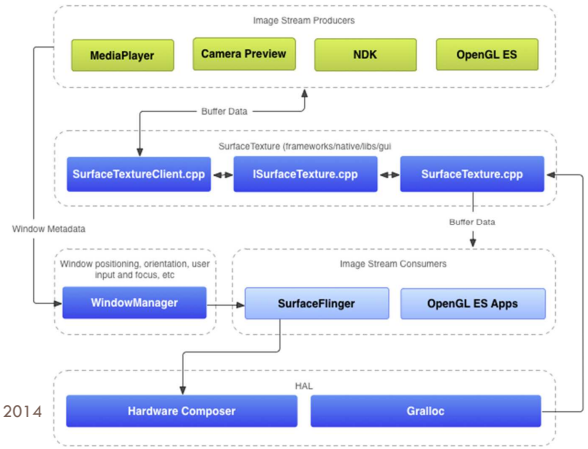
```
<uses-permission android:name="android.permission.INTERNET" />
```

  - **Wake Lock Permission** – If player application needs to keep the screen from dimming or the processor from sleeping, must request this permission.

```
<uses-permission android:name="android.permission.WAKE_LOCK" />
```


Graphics

- There are two general ways that app developers can draw things to the screen: Canvas or OpenGL.



2D Drawing

- Android provides a set 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. Typically, there are two ways:
  - Draw your graphics or animations into a View object from your layout.
    - 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.
    - Call the appropriate class's draw method such as *onDraw()*, *drawPicture()*. You are also in control of any animation.

Drawing on a View

- If your application does not require a significant amount of processing or frame-rate speed, you should create a custom View component and draw with a Canvas in *View.onDraw()*.
- The *onDraw()* method will be called by the Android framework to request that your View draw itself.
- Android only call *onDraw()* as necessary.
  - Each time that your application is prepared to be drawn, you must request your View be invalidated by calling *invalidate()*. This indicates that you'd like your View to be drawn and Android will then call your *onDraw()* method.
- Inside your View component's *onDraw()*, use the Canvas given to you for all your drawing using various draw methods
- Once your onDraw() is complete, the Android framework will use your Canvas to draw a Bitmap handled by the system

Create a Custom View

- Extend the View class and define the *onDraw()* callback method.

```
public class MyView extends View {
    private RectF mOval;
    private Paint mPaint;

    public MyView(Context context) {
        super(context);
        mOval = new RectF();
        mPaint = new Paint();
    }

    @Override
    protected void onDraw(Canvas canvas) {
        mOval.set(20, 40, 180, 200);
        mPaint.setColor(Color.BLUE);
        canvas.drawOval(mOval, mPaint);
    }
}
```

The custom view can extend View directly for saving time

Constructor for the custom view

Define the four coordinates for a rectangle's corner

Create a new paint with default settings

An overridden onDraw() method to draw the label onto the provided canvas.

Set the rectangle's coordinates to the specified values.

Set the paint's color.

Draw the specified oval using the specified paint.

Using the Custom View

- All of the view classes defined in the Android framework extend View. Your custom view can also extend View directly, or you can save time by extending one of the existing view subclasses

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        View MyView = new MyView(this);
        setContentView(MyView);
    }
}
```

Create a custom view instance and set it as the ContentView for this Activity.

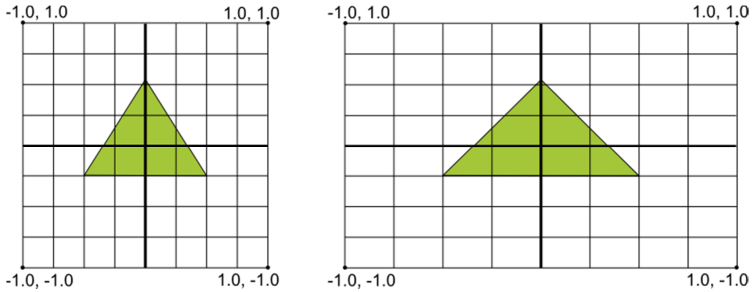
OpenGL ES

- Android includes support for high performance 2D and 3D graphics with the OpenGL ES API.
- OpenGL is a cross-platform graphics API that specifies a standard software interface for 3D graphics processing hardware.
- Android supports several versions of the OpenGL ES API:
  - OpenGL ES 1.0 and 1.1 – This API specification is supported by Android 1.0 and higher.
  - OpenGL ES 2.0 – This API specification is supported by Android 2.2 (API level 8) and higher.
  - OpenGL ES 3.0 – This API specification is supported by Android 4.3 (API level 18) and higher.

Support of the OpenGL ES 3.0 API on a device requires an implementation of this graphics pipeline provided by the device manufacturer. A device running Android 4.3 or higher may not support the OpenGL ES 3.0 API.

Mapping Coordinates for Drawn Objects

- One of the basic problems in displaying graphics on Android devices is that their screens can vary in size and shape.
- OpenGL assumes a square, uniform coordinate system and, by default, happily draws those coordinates onto your typically non-square screen as if it is perfectly square.



OpenGL ES Version Requirements

- If your application uses OpenGL features that are not available on all devices, you must include these requirements in your *AndroidManifest.xml* file.
  - If your application only supports OpenGL ES 2.0, you must declare that requirement by adding the following settings to your manifest as shown below.

```
<!-- Tell the system this app requires OpenGL ES 2.0. -->
<uses-feature android:glEsVersion="0x00020000" android:required="true" />
```

  - If your application uses texture compression, you must declare which compression formats you support so that devices that do not support theses formats do not try to run your application:

```
<supports-gl-texture android:name="GL_OES_compressed_ETC1_RGB8_texture" />
<supports-gl-texture android:name="GL_OES_compressed_paletted_texture" />
```


OpenGL ES API

- There are two foundational classes in the Android framework that let you create and manipulate graphics with the OpenGL ES API:
  - GLSurfaceView
    - This class is a View where you can draw and manipulate objects using OpenGL API calls and is similar in function to a *SurfaceView*
  - GLSurfaceView.Renderer
    - This interface defines the methods required for drawing graphics in a *GLSurfaceView*.

Create an Activity for OpenGL ES Graphics

- Android applications that use OpenGL ES have activities just like any other application that has a user interface. In an OpenGL ES app that, *GLSurfaceView* is used.

```
public class OpenGLES20Activity extends Activity {

    private GLSurfaceView mGLView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Create a GLSurfaceView instance and set it
        // as the ContentView for this Activity.
        mGLView = new MyGLSurfaceView(this);
        setContentView(mGLView);
    }
}
```

Build a GLSurfaceView Object

- A *GLSurfaceView* is a specialized view where you can draw OpenGL ES graphics.
- The actual drawing of objects is controlled in the *GLSurfaceView.Renderer* that you set on this view.

```
class MyGLSurfaceView extends GLSurfaceView {

    public MyGLSurfaceView(Context context){
        super(context);

        // Set the Renderer for drawing on the GLSurfaceView
        setRenderer(new MyRenderer());
    }
}
```


Build a Renderer Class

- *GLSurfaceView.Renderer* class controls what gets drawn on the *GLSurfaceView* with which it is associated.

```
public class MyGLRenderer implements GLSurfaceView.Renderer {

    public void onSurfaceCreated(GL10 unused, EGLConfig config) {
        // Set the background frame color
        GLES20.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    }

    public void onDrawFrame(GL10 unused) {
        // Redraw background color
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);
    }

    public void onSurfaceChanged(GL10 unused, int width, int height) {
        GLES20.glViewport(0, 0, width, height);
    }
}
```

The custom view can extend View directly for saving time

Called for each redraw of the view.

Called if the geometry of the view changes, for example when the device's screen orientation changes.

Defining Shapes

- OpenGL ES allows you to define drawn objects using coordinates in three-dimensional space.
- The typical way to do this is to define a vertex array of floating point numbers for the coordinates.
- For maximum efficiency, you write these coordinates into a *ByteBuffer*, that is passed into the OpenGL ES graphics pipeline for processing.

```
public class Triangle {

    private FloatBuffer vertexBuffer;

    // number of coordinates per vertex in this array
    static final int COORDS_PER_VERTEX = 3;
    static float triangleCoords[] = {   // in counterclockwise order:
             0.0f,  0.622008459f, 0.0f, // top
            -0.5f, -0.311004243f, 0.0f, // bottom left
             0.5f, -0.311004243f, 0.0f  // bottom right
    };

    // Set color with red, green, blue and alpha (opacity) values
    float color[] = { 0.63671875f, 0.76953125f, 0.22265625f, 1.0f };

    public Triangle() {
        // initialize vertex byte buffer for shape coordinates
        ByteBuffer bb = ByteBuffer.allocateDirect(
                // (number of coordinate values * 4 bytes per float)
                triangleCoords.length * 4);
        // use the device hardware's native byte order
        bb.order(ByteOrder.nativeOrder());

        // create a floating point buffer from the ByteBuffer
        vertexBuffer = bb.asFloatBuffer();
        // add the coordinates to the FloatBuffer
        vertexBuffer.put(triangleCoords);
        // set the buffer to read the first coordinate
        vertexBuffer.position(0);
    }
}
```


Drawing Sharp

- You must initialize and load the shapes you plan to draw.
- You should initialize them in the *onSurfaceCreated()* method of your renderer for memory and processing efficiency.
- Then create a *draw()* method for drawing the shape.

```
public void draw() {
    // Add program to OpenGL ES environment
    GLES20.glUseProgram(mProgram);

    // get handle to vertex shader's vPosition member
    mPositionHandle = GLES20.glGetAttribLocation(mProgram, "vPosition");

    // Enable a handle to the triangle vertices
    GLES20.glEnableVertexAttribArray(mPositionHandle);

    // Prepare the triangle coordinate data
    GLES20.glVertexAttribPointer(mPositionHandle, COORDS_PER_VERTEX,
                                 GLES20.GL_FLOAT, false,
                                 vertexStride, vertexBuffer);

    // get handle to fragment shader's vColor member
    mColorHandle = GLES20.glGetUniformLocation(mProgram, "vColor");

    // Set color for drawing the triangle
    GLES20.glUniform4fv(mColorHandle, 1, color, 0);

    // Draw the triangle
    GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vertexCount);

    // Disable vertex array
    GLES20.glDisableVertexAttribArray(mPositionHandle);
}
```