

University of Southern California
School of Engineering
USC Viterbi

Mobile Media API – JSR 135

Murali Annavaram & Bhaskar Krishnamachari
Ming Hsieh Department of Electrical Engineering
USC

Lecture notes based in part on slides created by Vidya Settur, Thanks also to Mikko H. Lipasti for course suggestions.

Mobile Multi-Media - Hardware

- Mobile devices increasingly support advanced multimedia features
- Hardware advances driven by the Moore's law
 - > Broadcom's *Cellairity* mobile media platform supports 8MP camera, MP-4 video at 35 mW, audio at 15mW
 - > AMD Imageon Media processor supports 12MP camera, DVD recording with image stabilization, echo cancelling video telephony (~20mW)
 - > Freescale i.MX31*
- So the hardware is already there!
- What next?

Mobile Multi-Media - Software

- Roughly 20% of today's mobile phone cost is software
- Multimedia software expected increase the share of the software to 30-40% of cost
 - > Music downloads, mobile online gaming, mobile video downloading (Did you see the quality of SpyderMan -3 on N95?), YouTube Mobile,.....
- Mobile entertainment is poised for software explosion
 - > Enabling services, DRM and content protection,...

JSR 135 MMAPI V1.2

- Count the number multimedia types and formats?
 - > MP3, MP4, MIDI, WAV, WMA... (good luck!)
- Count the number of storage and delivery methods
 - > Hard drives, Flash drives, HTTP, UDP, WAP
- Mobile device capabilities are just as diverse
 - > Simple ring tone playback to recoding DVD video with image stabilization
- JSR 135 is a set of high-level API introduced to accommodate diverse configurations and multimedia processing capabilities

Salient Features of JSR 135

- The API supports any time-based audio and video content by offering tools to control the flow of the media stream
- Mainly targets CLDC, but CDC is also OK!
- Is independent of content (MIDI/WAV/?) and protocol (HTTP/WAP/?)
- A device may support only a subset of features (Audio only)
- API is extensible in future (without breaking the existing functionality)

Basics of Multimedia Processing

- Two parts to processing media
 - > Protocol Handling: Specifying the source of media data (file/capture device/streaming)
 - > Content Handling: Parsing the bits that are read from source
- DataSource is an abstract class for protocol handling
- DataSource hands data to Player to process and render
- Manager is used to create player from datasource
- Player player = Manager.createPlayer(String url)
 - > The url specifies the protocol and the content, using the format <protocol>:<content location>
 - > createPlayer("capture://audio")
 - > createPlayer("capture://video")
 - > createPlayer("capture://radio?f=93.3&st=stereo")
 - > createPlayer("http(rtp)://mymusicserver:port/type")
 - > createPlayer("device://midi")

<http://developers.sun.com/mobility/midi/articles/mmap-overview94.pdf>

Querying Supported Content Types

- MMAPI doesn't require any media type or protocol support
 - So how do you know what the device supports?
- `Manager.getSupportedContentTypes(string)`
 - Set parameter to NULL to return an array of strings that show supported content types

Audio/midi	Audio/mid	audio/x-tone-seq	audio/mpeg
audio/mp4	audio/mp4a-latm	audio/3gpp	audio/3gpp2
video/3gpp	video/mp4	video/x-pmd	

 - Set parameter to String to filter
 - `Manager.getSupportedContentTypes("http")` returns only content types that are supported by http protocol



Querying Supported Protocols

- `Manager.getSupportedProtocols(string)`
 - Set parameter to NULL to return an array of strings that show supported protocols

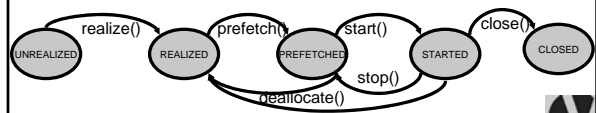
device	http	https	rtp
Rtsp	File	Capture	

 - Set parameter to String to filter
 - `Manager.getSupportedProtocols("audio/midi")` returns only protocols that supported midi content type
- Another approach (unreliable, use with Caution!) `System.getProperty(str)`
 - "supports.video.capture", "streamable.contents, supports"
- Application portability:
 - Never assume that something is supported by default
 - Check the content and protocol type support
 - Your Midlet should gracefully handle all exceptions



Player States

- You created a player, got the supported content and protocol types figured out
 - How do you actually play the content?
- Every player has a 5 state life-cycle
 - UNREALIZED, REALIZED, PREFETCHED, STARTED, and CLOSED
- Every player supports 6 methods to change state
 - `realize()`, `prefetch()`, `start()`, `stop()`, `deallocate()`, `close()`



Player States

- UNREALIZED : All players start here
 - Can not call methods such as `getContentType()`
 - Can never transition to UNREALIZED, except when `deallocate` is called during `REALIZE()` function
- REALIZED: obtained the information required to acquire the media resources
 - Can be a resource and time consuming process
 - Contact the server, read file etc.,
- PREFETCHED: Start getting the streaming data into buffers
 - Good practice to always prefetch before starting player to reduce user perceived delay
- STARTED: Player is processing the data (recording or playback)
 - Posts events when the processing is done (`END_OF_MEDIA`)
 - Events can be used for processing call back functions
- CLOSED: All resources are released



Getting Player Controls

- Gives fine grain control of media players
 - `ctrl = player.getControl(str)`
 - Parameter can be `VideoControl`, `VolumeControl`, `TempoControl`, `RecordControl`, ...
 - Player has to be in `REALIZED`, `PREFETCHED`, `STARTED` state
 - Returns control class
- Control class returned supports implementation specific methods
 - `Videocontrol.setNidisplay()`
 - `Volumecontrol.setLevel()`
 - `Recordcontrol.setlocation()`



MMAPI application cycle

1. Create UI to give controls, create a new thread to handle player
2. Create player in new thread and realize the player: `Manager.createPlayer()`, `player.realize()`
3. Register listener: `player.addPlayerListener(class C)`
 - `playerUpdate()` function in class C is called when player encounters an event, such as `END_OF_MEDIA`
4. Get player controls: `ctrl = player.getControl(str)`
 - Parameter can be `VideoControl`, `VolumeControl`, ...
5. Use controls to control media specific issues (such as volume, are where the video can be displayed)
6. Use `playerUpdate()` function to handle events
 - For instance, if you break up the video into chunks at the sever you can stitch it on client at playtime using this call back



Example of Playing Video from a Webserver

```
Player myplayer;
VideoControl ctrl;
try {
    myplayer = Manager.createPlayer("http://java.sun.com/products/java-media/mma/media/test-wav.wav");
    myplayer.realize();

    // Grab the video control and set it to the current display.
    ctrl = (VideoControl) p.getControl("VideoControl");
    if (vc != null) {
        Form myform = new Form("Video Playback");
        myform.append(vc.initDisplayMode(vc.USE_GUI_PRIMITIVE, null));
        Display.getDisplay(midlet).setCurrent(myform);
    }
    myplayer.start();
} catch (IOException ioe) {
    //Handle IO exceptions
} catch (MediaException me) {
    //Handle Media Exceptions
}
```



Review

- Manager creates a player using a given protocol and content type
- Applications can query the supported protocols and content type
 - Applications need to handle exceptions gracefully to be portable
- Applications can control how the media is played/recorded
- All player state changes are delivered to a player listener



Looking to Future of MMAPi


- JSR 234 Advanced Multimedia Supplements
- Better support for camera and radio
 - Brightness, contrast, flash-control, zoom...
- Access to advanced audio processing
 - Equalizers (treble/bass settings), audio outputs
 - Audio output direction (headset/device)
- More to come to support future devices...



Thinking of Project#3

- Assume the media content is split into small chunks at the server
 - Server can be device itself that records its own surroundings in sound clips
- Client can download the media content in chunks
- Play the chunks back as if the end user does not know the difference
 - Again, cache the chunks if the user wants to playback





University of Southern California
School of Engineering


Introduction to Mobile Gaming

Murali Annavaram & Bhaskar
Krishnamachari

Ming Hsieh Department of Electrical
Engineering

USC

Lecture notes based in part on slides created by Vidya Setlur,
Thanks also to Mikko H. Lipasti for course suggestions.



Mobile Gaming in MIDP

- MIDP 2.0 introduced Game API using the package `javax.microedition.lcdui.game`
- The package has five classes
 - GameCanvas
 - LayerManger
 - Layer
 - Sprite
 - TiledLayer
- Game API provides two important functionality
 - GameCanvas allows user to paint a screen and respond to user input in the body of a game loop
 - In MIDP 1.0 these two functions were separated causing several glitches
 - Layer API makes it easy to build complex scenes efficiently



GameCanvas Vs Canvas

- The run() method updates the game once each time step
 - > Typical tasks would be to update the game object (animation)
 - > Repaint() updates the screen
- User inputs are delivered to keyPressed(), which updates the game state appropriately
- Asynchronous handling of events makes it impossible to predict when the repaint() actually occurs or when the keys are actually handled

```
public class MyGameCanvas extends Canvas
implements Runnable {
    public void run() {
        while (true) {
            // Update the game state
            repaint();
            // Delay one time step.
        }
    }
    public void paint(Graphics g) {
        // Painting code goes here.
    }
    protected void keyPressed(int keyCode) {
        // Respond to key presses here.
    }
}
```

<http://developers.sun.com/mobility/mdp/articles/game/>



GameCanvas Vs Canvas

- GameCanvas unifies all the components of gaming in one class
 - > getGraphics() gives you pointer to a screen offscreen buffer
 - > All manipulations to graphics happen offscreen
 - > flushGraphics() copies buffer to device screen
 - flushGraphics() is blocking
 - Repaint() is non-blocking and hence no control on when the graphics are displayed
 - > getKeyStates() is polling based
 - All keys pressed are immediately returned in one bit vector

```
public class MyGameCanvas extends
GameCanvas implements Runnable {
    public void run() {
        Graphics g = getGraphics();
        while (true) {
            // Update the game state.
            int keyState = getKeyStates();
            // Respond to key presses here.
            // Painting code goes here.
            flushGraphics();
            // Delay one time step.
        }
    }
}
```

<http://developers.sun.com/mobility/mdp/articles/game/>



GameCanvas

- Calling the method getKeyStates() returns a bitwise representation of all of the physical game keys, expressed as 1 for pressed and 0 for unpressed, *since the last time the method was called*.
- The following game states are identified: **DOWN_PRESSED, UP_PRESSED, RIGHT_PRESSED, LEFT_PRESSED, FIRE_PRESSED, GAME_A_PRESSED, GAME_B_PRESSED, GAME_C_PRESSED, and GAME_D_PRESSED.**



Layer API

- Build game scenes using layers
 - > Draw one layer as background
 - > Draw the animated character as second layer on top
- LayerManager controls all the layers in your scene
- Layer is an abstract parent class of all layers
 - > Supports isVisible, size, position methods
 - > Two important sub classes TiledLayer, Sprite



TiledLayer

- TiledLayer: Takes an image and splits it into tiles
 - > Image image = Image.createImage("imagename");
 - > TiledLayer tiledLayer = new TiledLayer(ROWS, COLS, image, X-PIXELS, Y-PIXELS)
- These tiles can then be independently used in a scene creation
 - > tiledLayer.setCell(4,2,5) → Replace tile5 tile in cell 5, row 3
- You can create image masks and then use setCell to create a new scene from an existing image
 - > image = Image.createImage("myimage.png");
 - TiledLayer tiledLayer = new TiledLayer(2, 2, image, 16, 16)
 - int[] map = { 1, 3, 2, 0};
 - for (int i = 0; i < map.length; i++) {
 - int column = i % 10; int row = (i - column) / 10;
 - tiledLayer.setCell(column, row, map[i]);
 - return tiledLayer; }



Sprite

- Sprite is a collection of images that are primarily meant for moving around on the screen (animation)
- Traditionally, TiledLayer forms the background and Sprite forms the action objects
- Sprite class provides methods to animate the sprite from a handful of images, similar to the way backgrounds are created using the TiledLayer class.
- It also provides methods to check collisions with other game elements, including images, sprites, or tiled layers.



Sprite

- `mySprite = new Sprite(IMAGE, x-dim, y-dim)`
 - The IMAGE is then subdivided into smaller images each of size x-dim, y-dim and then appended to form a frame series
- Defaultly, the frames are appended top-left towards bottom-right (scanning the image left to right)
 - `nextFrame()` gives you the frame that comes after the current image
 - Use `setFrame` (frame number) to change the default
- **Note that you can create many Sprite layers**
 - But pay attention to clarity on a small screen size



LayerManager

- A game may contain at least one `TiledLayer` for background and several `Sprite` classes for individual movable game pieces.
- The `LayerManager` provides methods to add, remove, or insert layers from a game, and also provides a single method to paint all of these layers to the underlying `Graphics` object. So you really don't have to call `paint()` method of each of the layers of a game.
- An instance of `LayerManager` is created using its zero argument constructor. Layers are then added, removed, or inserted into it by using the methods `append(Layer layer)`, `remove(Layer layer)`, and `insert(Layer l, int index)`.
- The layer at index 0 is painted on top of all the other layers, and hence, is *closest* to the user.



LayerManager Usage

- Create a layer manager
 - `lm = new layerManager()`
- Add layers in reverse order you want them to be painted
 - `lm.append(sprite) → Layer 0`
 - `lm.append(tiledlayer) → Layer 1`
- Layer 0 is closest to user



Collisions

- `Sprite` provides API to detect collisions
 - `collidesWith()` returns TRUE when collision has occurred, i.e pixel overlap
- ```
sprite1=new Sprite(img,95,35);
sprite2=new Sprite(img,95,35);
if(sprite1.collidesWith(sprite2,true)) { // do something}
```



## Review

- Mobile Gaming is a hot topic
- MIDP 2.0 provides several API for 2D gaming
  - Layermanager -> Tiledlayer -> Sprites
- Game applications should watch for the mobile device constraints
  - Small screen size, ADD (Attention Deficit Disorder), Network costs, connection limitations
- 3D gaming API also recently introduced



## Thinking of Project#3-B

- We already know how to do Bluetooth based communication
- Now imagine a simple two player car racing game where one device becomes a master game console
- At the start of the game the master device starts the game and player 1 car1 (`Sprite`) is displayed on screen
  - Master then communicates the tiledlayer information, Car 1 screen coordinates
- Player 2 receives the information over bluetooth and uses that information to display the screen background (from tiledlayer) and where Car 1 is located
- Player 2 then places car2 (`Sprite`) and sends it screen coordinates to Player 1
- Now let the two players move on the screen without collisions

