Mobile Information Device Programming (17)

Lecturer: Alireza Mousavi
School of Engineering & Design
www.brunel.ac.uk/~emstaam

Topic

- Game API
- Designing a simple game canvas
- Multiple layers

Reading Sources:

Riggs, R. et al (2003), Programming wireless devices with J2ME, second edition, Sun Micro Systems

Knudsen J. (2003), Creating 2D Action Games with Game API – Sun Micro - Systems Developer notes

Datasheet – Games on the Java Platform for Mobile Information Device Profile – Sun Micro Systems Developer notes

Developing Mobile Phone Applications with J2ME Technology (2004), Sun Micro Systems, Educational Services

Sing Li and Knudsen, J. (2005), Beginning J2ME from novice to professional, 3rd edition, Apress.

Game API

- The game API is located in the javax.microedition.lcdui.game package

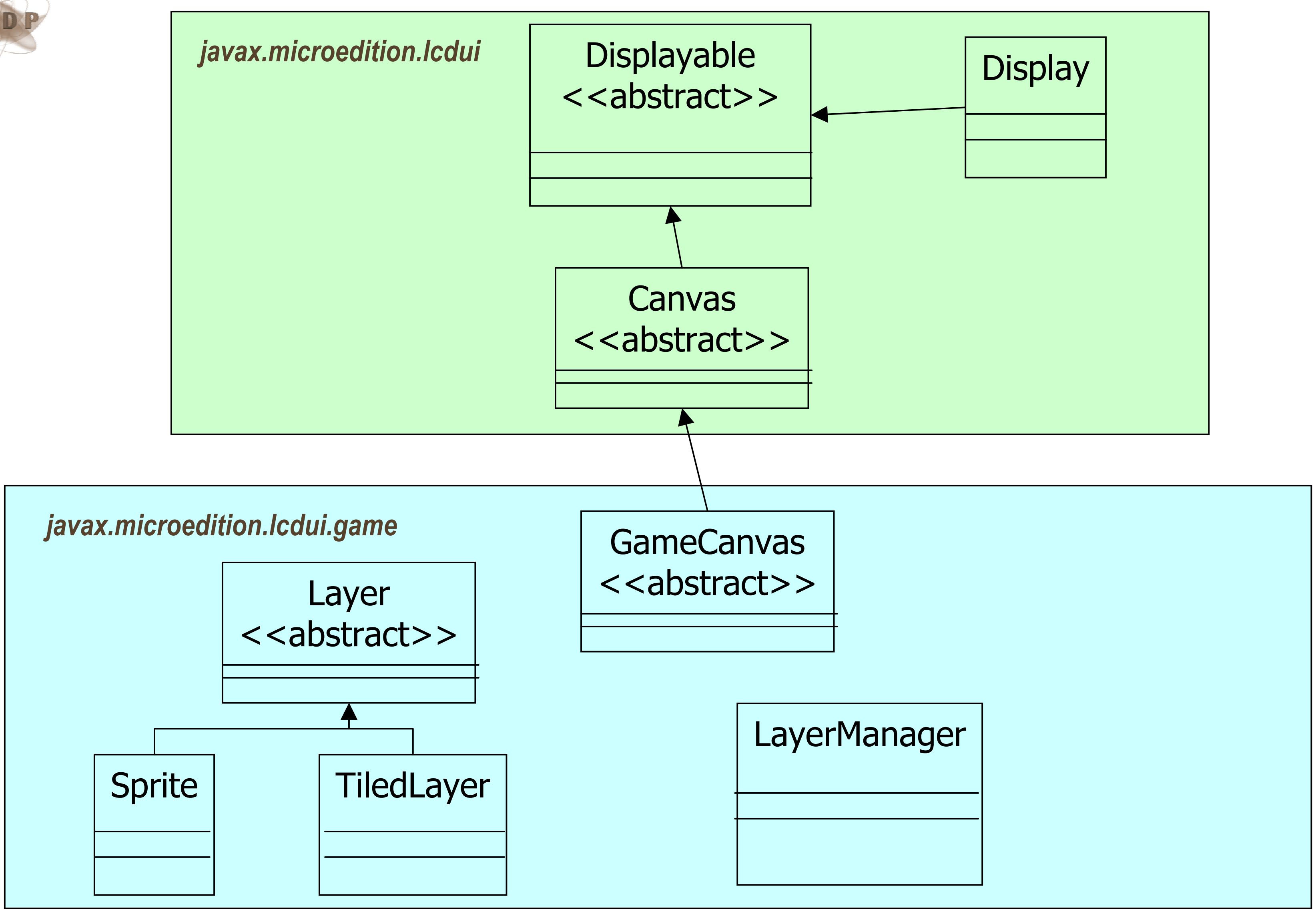
- It consists of 6 classes *GameCanvas, GameDeviceCaps, Layer, LayerManager, Sprite, TiledLayer*

- These classes:
    - Make it possible to paint a screen within the body of a game, instead of relying on the system's input thread and painting
    - Provide an efficient, capable and flexible layer API to facilitate the build of complex screens
    - Improved application performance
    - Decreased application size

Game API Offers

1. Simplifies game development (familiar environment)
2. Reduction in size and complexity
3. Improve performance by using frequently used game routines

javax.microedition.lcdui.game **Package**



Developing Mobile Phone Applications with J2ME Technology (2004), Sun Micro Systems, Educational Services

GameCanvas **Class**

- The *GameCanvas* class is the backbone of *lcdui.game* package
- It is similar to the *Canvas* class
- Acts as the basic screen for a typical game application
- It contains methods to manage graphics (painting) and key actions (state)
- It enables users to draw on display using *paint( )* method
- With the methods in this class you can manage game functionalities much more efficiently

GameCanvas **Methods Functionality**

The GameCanvas methods provide with the following functionalities:

1. Game querying functions

2. Synchronous graphics functioning

GameCanvas Methods Features

GameCanvas bypasses the normal painting and key-event mechanism – allowing the game applications to be contained in one loop.

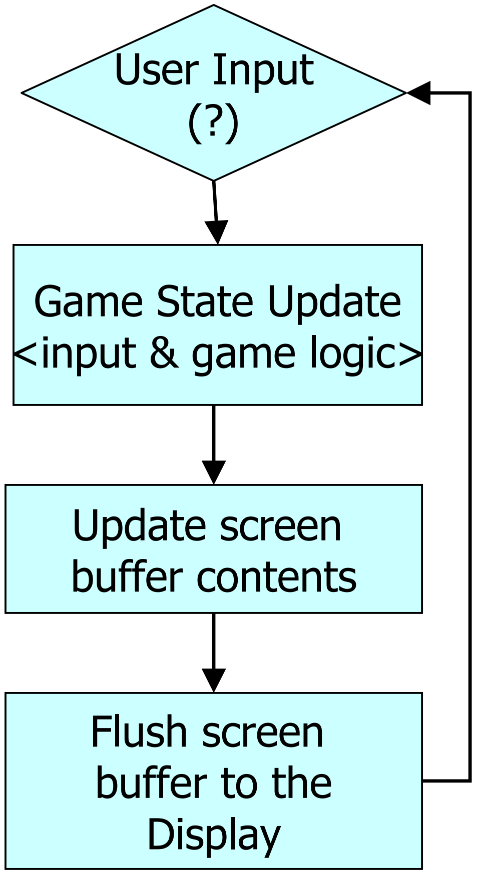
It allows you to do this by:

1. Using *GetGraphics ( )* method to directly access the *Graphics* object
2. Maintaining an internal off-screen buffer for the *Graphics* object
3. Updating the screen using *flushGraphics( )* method – it is better way than the *repaint ( )* method
4. Containing a better method to determine the key state (polling) *getKeyStates ( )* instead of *keyPressed ( )* method

Canvas

GameCanvas

Game Loop Flowchart


User Input (?)
Game State Update <input & game logic>
Update screen buffer contents
Flush screen buffer to the Display

Source: Riggs, R. et al (2003), Programming wireless devices with J2ME, second edition, Sun Micro Systems

Game Loop Example

```
Public Class FlyingObject  extends GameCanvas  implements Runnable {

      public void run ( ) {
        Graphics g = getGraphics( );
        while (true) {
        // while run is true
       // update the game
      int keystate = getKeyStates ( );
      // implement the function associated with the key
      // update the graphics
      flushGraphics( ); // flush the buffer
      // other functions
      try {
      Thread.sleep(100); // delay time
      }
      catch (InterruptedException ie) {  }
      }
    }
}
```

Layer Classes

In order to improve the performance of mobile games and use the resources more efficiently 4 classes have been included in the *javax.microedition.lcdui.game* package:

1. *Layer* class
2. *LayerManager* class
3. *TiledLayer* class
4. *Sprite* class

The *Layer* class

The *Layer* class:

- represents one visual layer in a game application (visual entity)
- is the abstract parent of all layers
- it defines the basic attributes i.e. size, position and visibility
- each subclass of *Layer* defines a *paint ( )* method
- the subclasses are *TiledLayer* and *Sprite*

The *LayerManager* class

The LayerManager class:

- automates the rendering of multiple *Layer* classes making the game pieces more manageable
- keeps track of all the layers on the screen

Example:

```
private  LayerManager  Mylayermanager;   // declaring a layer manager
...
Mylayermanager = new  LayerManager ( );  // creating a LayerManager object
Mylayermanager.append(backgrnsImage);  // append the image to the LayerManager object
```

The *TiledLayer* class

- *TiledLayer* is suitable for designing game backgrounds
- It helps you achieve different looks by combining images
- Imagine the screen to be divided into rows and columns that are covered with tiles containing a part of an image
- You need to create a separate image file that contains the image or a combination of images (save in "*res*" directory)
- This image can then be loaded to create a *TiledLayer,*

Example:

```
TiledLayer (int column, int rows, Image img, int tileWidth, int tileHeight)

Image img = Image.createImage("/backgrnd.png);
TiledLayer bcktile = new TiledLayer(8, 8, img, 16, 16);
```

The *Sprite* class

- Subclass of *Layer* class
- It is used to represent individual game pieces
- *Sprite* uses a sequence source image frames to assimilate animation (it points a single image file)

Creating *Sprite* object:

1. An image object needs to be associated with *Sprite* object
2. First define an image object and then passing the parameter to the *Sprite* object constructor

Sprite **Object**

```
// Create a Sprite Object
Image  SubmarineImage = Image.createImage("/sub.png"); // an image instantiated
submarine =  new Sprite(SubmarineImage); // parameters passed to a Sprite object
submarine.setPosition (50, 150); // the position of the object on the screen

// add this to the LayerManager object

Mylayermanager.insert(submarine, 0);
```

Collision between *Sprite objects*

- Sometimes you may want to assimilate collision
- Basically two objects overlapping with one another
- You can create *Sprite* objects to detect collision on display

Using *Sprite* objects you can detect the elements collisions on the display:

1. *Sprite* objects
2. *Tiledlayer* objects
3. *Image* objects

How to define collision and the methods

```
/* the area defined by the collision rectangle --- pixel detection
The setting improves game performance  → Why? */
public void defineCollisionRectangle(int x, int y, int width, int height)


// check if two Sprite/Tiledlayer/Image objects have collided
public final boolean collidesWith(Sprite s, boolean pixelLevel)
public final boolean collidesWith(TiledLayer t, boolean pixelLevel)
public final boolean collidesWith(Image img, boolean pixelLevel)
```

Polling Example

```
public void run( ){
 Graphics g = getGraphics( );
int  keystroke = 0; // initialisation
int current x = 0; // initialisation
while (true) {
keystroke = getKeyStates ( ) ; // find out which key is pressed (current state of the
      keys called polling)
…
if ((keystroke & Left_PRESSED) != 0) {
Img.move(-5, 0) //  if keystroke value is not zero and Left-Pressed move Img 5 pixles
      to the left
}
…
```

Features of Game API

- Use of animation loop in *GameCanvas*
- Polling for key states through *GamecCanvas*
- Using *LayerManager* to create and maintain multiple layer
- Creation of *Sprite* and *TileLayer* objects
- Animation of *Sprite*, including changing frame sequences and transformations
- Use of an animated tile in a *TileLayer*

Source: Sing Li et al 2005

Exercise 12-1 & 12-2