Threading and Surface Views

CE881: Mobile and Social Application Programming

Spyros Samothrakis

Febrary 15, 2016

Theme: "Multi-threading"

- Matrix Trilogy
- Neuromancer
- Shadowrun (tabletop game and computer game)

Weekly propaganda: IDE Shortcuts (IDEA)

- Ctrl + Shift + A (Meta key)
- Alt + Insert (Generate)
- Ctrl + left click

Background

- Most of the Android apps we've covered so far have been single threaded
  - Event driven
  - An exception is the BubbleGame studied in the lab
- In event driven apps all the methods were invoked either directly or indirectly by:
  - Lifecycle events (e.g. onCreate(), onPause())
  - Or user-actions
- onTouch(), onClick()
- The recommended way to implement RT games:
  - Use a SurfaceView
  - And a separate animation Thread

Android and threading

- Each app runs by default in its own thread
- Single process
- UI-Thread

Process lifecycle

- Process
  - Foreground process
  - Visible process
  - Service process
  - Background process
  - Empty process

Priorities (0)

- android.os.Process.setThreadPriority(int priority)
- -20 to 19 (lowest is highest priority)
- Same as linux “nice” command
- java.lang.Thread.setPriority(int priority)
- 0 to 10
- Java thread priorities map to process (linux) priorities

Priorities (1)

```
enum {  ANDROID_PRIORITY_LOWEST         =  19,
        /* use for background tasks */
        ANDROID_PRIORITY_BACKGROUND     =  10,
        /* most threads run at normal priority */
        ANDROID_PRIORITY_NORMAL         =   0,
        /* threads currently running a UI that the user is interacting with */
        ANDROID_PRIORITY_FOREGROUND     =  -2,
        /* the main UI thread has a slightly more favorable priority */
        ANDROID_PRIORITY_DISPLAY        =  -4,
        /* ui service treads might want to run at a urgent display (uncommon) */
        ANDROID_PRIORITY_URGENT_DISPLAY =  -8,
        /* all normal audio threads */
        ANDROID_PRIORITY_AUDIO          = -16,
        /* service audio threads (uncommon) */
        ANDROID_PRIORITY_URGENT_AUDIO   = -19,
        /* should never be used in practice. regular process might not
         * be allowed to use this level */
        ANDROID_PRIORITY_HIGHEST        = -20,
        ANDROID_PRIORITY_DEFAULT        = ANDROID_PRIORITY_NORMAL,
        ANDROID_PRIORITY_MORE_FAVORABLE = -1,
        ANDROID_PRIORITY_LESS_FAVORABLE = +1, };
```

Priorities (2)

```
static const int kNiceValues[10] = {
  ANDROID_PRIORITY_LOWEST, /* 1 (MIN_PRIORITY) */
  ANDROID_PRIORITY_BACKGROUND + 6,
  ANDROID_PRIORITY_BACKGROUND + 3,
  ANDROID_PRIORITY_BACKGROUND,
  ANDROID_PRIORITY_NORMAL, /* 5 (NORM_PRIORITY) */
  ANDROID_PRIORITY_NORMAL - 2,
  ANDROID_PRIORITY_NORMAL - 4,
  ANDROID_PRIORITY_URGENT_DISPLAY + 3,
  ANDROID_PRIORITY_URGENT_DISPLAY + 2,
  ANDROID_PRIORITY_URGENT_DISPLAY /* 10 (MAX_PRIORITY) */
};
```

- From 19 to -8
- Default priority is 0

Threads

- Multi-threaded programs: multiple flows of control (easy-ish)
- But problems arise when multiple threads need write-access to the same data
- Synchronisation is necessary to ensure proper behaviour

Example

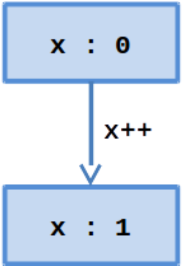
```
// get number of available cores
n_cores = Runtime.getRuntime().availableProcessors();

// create queue
blockQueue = new LinkedBlockingQueue<Runnable>();
// create executor
threadPool = new ThreadPoolExecutor(
               n_cores,      // initial pool size
               n_cores,      // maximum pool size
               5, // idle threads die after 5
               TimeUnit.SECONDS, // seconds
               blockQueue);

// Execute one or more runnables
threadPool.execute(SomeRunnable())
```

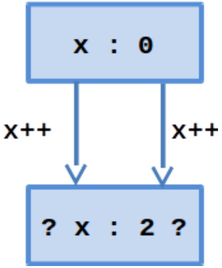
Stopping threads

- Thread.interrupt()
- Only stops threads that are sleeping/waiting
- Thus you might get stuck in doing CPU/IO intensive tasks
- Check Thread.interrupted() inside run()

Single to Multi Threaded

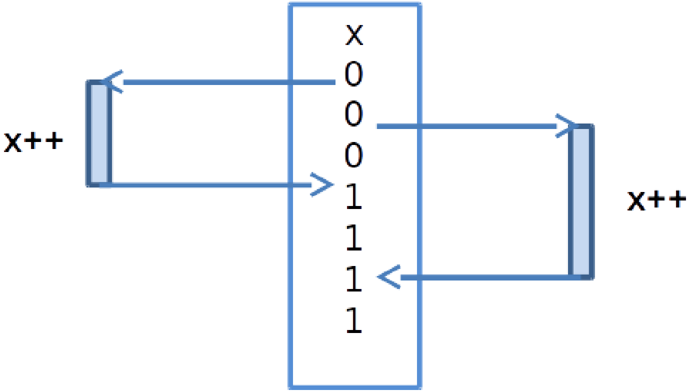
- Multiple flows of control, overlapping code AND data





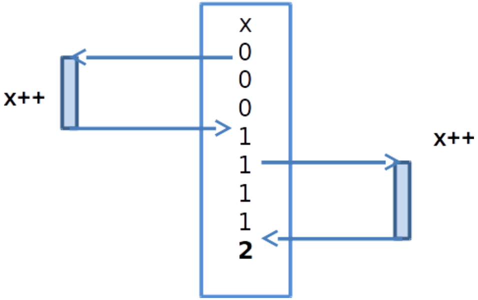
Thread Interference

- Threads may interfere when modifying the same data in an uncontrolled way
- Result can be unpredictable (think x+=1)



Solution: protect access via a lock

- In Java we use *synchronized* blocks/methods, or *Semaphore* class, or *volatile* keyword
- Each thread has to wait for access to protected area
- We are now guaranteed the correct result



Java Example

```java
public class ThreadTest extends Thread {
    static int x;
    int n;

    public void inc() {
        x++;
    }

    public ThreadTest(int n) {
        this.n = n;
        // run method called in this new Thread
        start();
    }

    public void run() {
        while (n-- > 0) {
            inc();
        }
    }
```

Broken

```
public static void main(String[] args)
        throws Exception {
    int n = 5000000;
    ThreadTest tt1 = new ThreadTest(n);
    ThreadTest tt2 = new ThreadTest(n);
    tt1.join();
    tt2.join();
    System.out.println("x = " + x);
  }
```

Solution

- Use synchronized keyword
- Restrict access to inc() method (or use volatile keyword)
- But note:
- Method must be declared static as well as synchronized
- Each lock is associated with an object
- Without the static modifier independent locks will be used, one for each object (and hence for each thread)

Fixed

```
public static synchronized void inc() {
    x++;
 }
```

Deadlocks

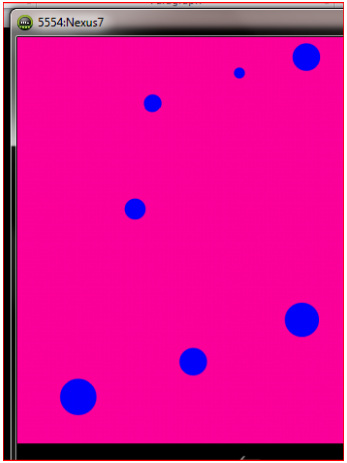
- Deadlock can occur when multiple threads compete for multiple locks
- Thread 1 holds lock that Thread 2 needs to proceed
- And vice versa
- Simplest solution
- Use a single lock (may be enough for game-type apps)
- More sophisticated
- Always ensure shared locks are requested in the same order

Android: Surface View

- We've seen how improper management of multi-threaded access to shared resources can cause problems
- If you do this when using a SurfaceView in Android:
- The App may crash
- Disaster!
- Five seconds of unresponsiveness will…

Hello Surface View

Some movable sprites . . .



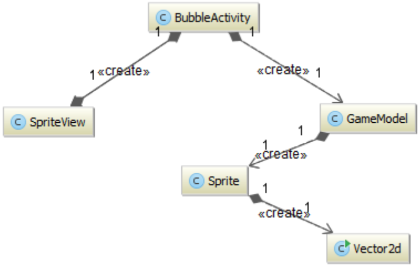
- We'll now study a "simple" surface view app
- In these notes we'll just show an overview of the classes involved
- Complete source code is in associated lab

Model-View-{Controller, Presenter}

- Controller - receives actions
  - Controller updates Model
    - Model deals with app logic
    - Model Updates View
- Presenter - receives actions
  - Updates model
  - Updates view

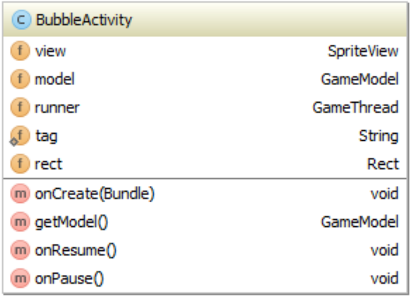
Overview of Classes - Showing dependencies

- At this stage no inheritance in App classes
- Though some inherit from appropriate Android classes
  - Which ones?
- Let's look at each in turn
- Is a class missing from the diagram?



BubbleActivity extends Activity

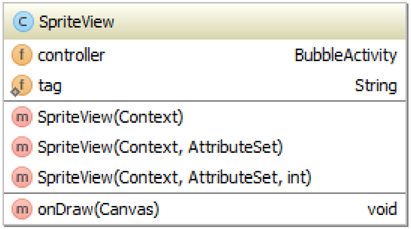
- Standard entry point for app
- Overrides onCreate()
- Creates a new SpriteView object
- Sets the current layout to that object
- Starts and stops thread in onPause and onResume



BubbleActivity	
view	SpriteView
model	GameModel
runner	GameThread
tag	String
rect	Rect
onCreate(Bundle)	void
getModel()	GameModel
onResume()	void
onPause()	void

SpriteView extends SurfaceView

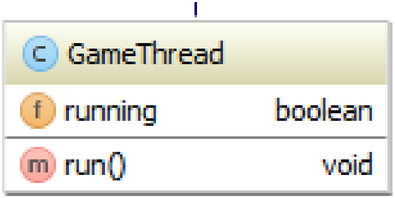
- Draws the sprites in the model
- Also handles onTouch and onClick events
- Some strange things happen if you only override one of these!
- I had to override both to get them working!



SpriteView	
controller	BubbleActivity
tag	String
SpriteView(Context)	
SpriteView(Context, AttributeSet)	
SpriteView(Context, AttributeSet, int)	
onDraw(Canvas)	void

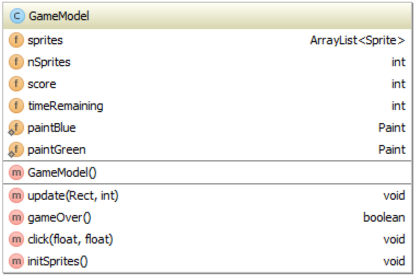
GameThread extends Thread

- Controls the running of the app
- Most work is orchestrated in the run method
- This calls:
  - model.update()
  - view.draw()
  - sleep()



GameModel

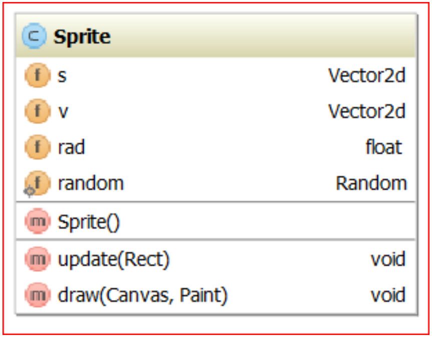
- Stores the sprites
- Provides a method for updating them
- Also implements the action for when the view is clicked
- Checks whether a bubble needs popping
- Anything out of place?



GameModel	
sprites	ArrayList<Sprite>
nSprites	int
score	int
timeRemaining	int
paintBlue	Paint
paintGreen	Paint
GameModel()	
update(Rect, int)	void
gameOver()	boolean
click(float, float)	void
initSprites()	void

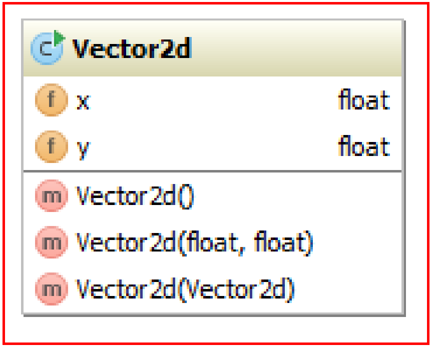
Sprite

- Stores position (s) and velocity (v) of a 2d object
- These are modelled with Vector2d objects
- Responsible for:
- Updating position
- Drawing
- Also holds size of sprite (rad)



Vector2D

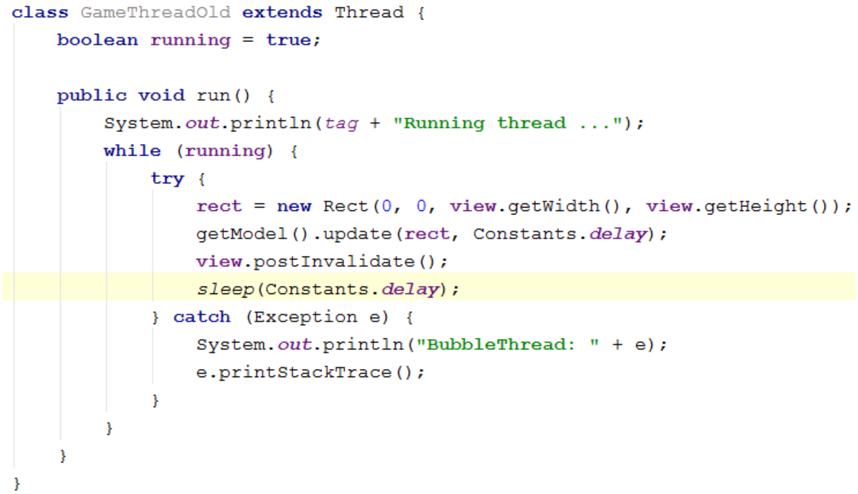
- Useful in any 2d app that deals with a number of movable objects
- Can then think in terms of positions and velocities directly
- Methods not shown on diagram, but include
  - Addition
  - Subtraction
  - Distance
  - Rotation
  - Scalar Product



From View -> SurfaceView

- Recall from the lab that using postInvalidate causes a problem: what is the problem and why is it caused?
- Interestingly, remarkably little needs to change in going from a view to a surface view
- First we'll cover the essentials
- And then look at some optional extras

Old Game Thread (uses postInvalidate)



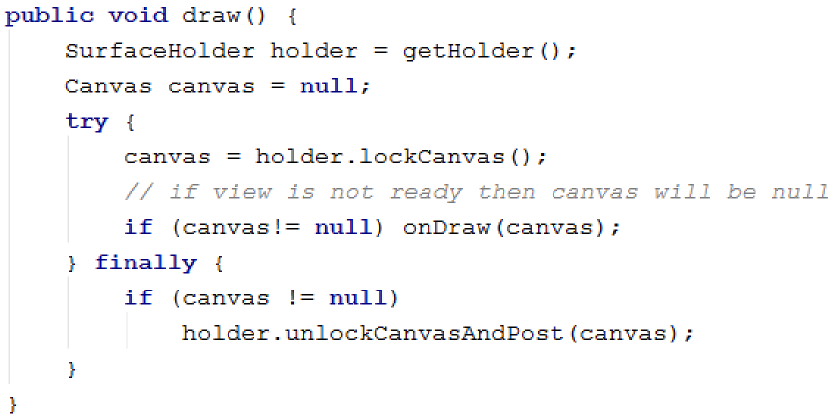
```java
class GameThreadOld extends Thread {
    boolean running = true;

    public void run() {
        System.out.println(tag + "Running thread ...");
        while (running) {
            try {
                rect = new Rect(0, 0, view.getWidth(), view.getHeight());
                getModel().update(rect, Constants.delay);
                view.postInvalidate();
                sleep(Constants.delay);
            } catch (Exception e) {
                System.out.println("BubbleThread: " + e);
                e.printStackTrace();
            }
        }
    }
}
```

The new version: spot the difference!

```java
class GameThread extends Thread {
    // have
    boolean running = true;

    public void run() {
        System.out.println(tag + "Running thread ...");
        while (running) {
            try {
                rect = new Rect(0, 0, view.getWidth(), view.getHeight());
                getModel().update(rect, Constants.delay);
                view.draw();
                sleep(Constants.delay);
            } catch (Exception e) {
                System.out.println("BubbleThread: " + e);
                e.printStackTrace();
            }
        }
    }
}
```

And the draw method ...

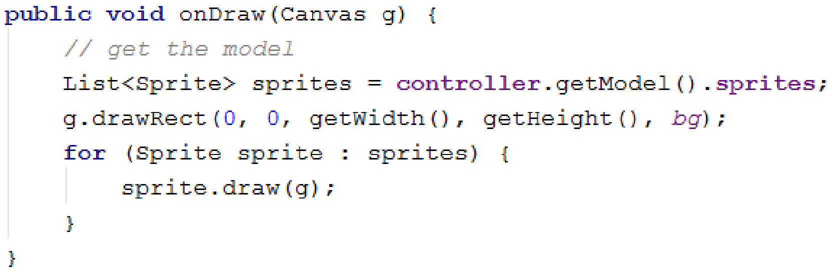
- Get a surface holder and lock the canvas
- Then use the same onDraw method



```java
public void draw() {
    SurfaceHolder holder = getHolder();
    Canvas canvas = null;
    try {
        canvas = holder.lockCanvas();
        // if view is not ready then canvas will be null
        if (canvas!= null) onDraw(canvas);
    } finally {
        if (canvas != null)
            holder.unlockCanvasAndPost(canvas);
    }
}
```

onDraw is the same as before . . .

- except now it is being called from the app thread



```
public void onDraw(Canvas g) {
    // get the model
    List<Sprite> sprites = controller.getModel().sprites;
    g.drawRect(0, 0, getWidth(), getHeight(), bg);
    for (Sprite sprite : sprites) {
        sprite.draw(g);
    }
}
```

Some more

- Note that we checked that the Canvas was not null before trying to use it
- This is because the call to holder.lockCanvas() will return null if the SurfaceView is not yet ready for drawing
- The approach I've taken in my code is to start the app thread (GameThread) before the surface is ready
- And then use the null test to avoid trying to draw on it if it is not ready

Using SurfaceHolder.Callback

- There is another way to do it
- Can use SurfaceView callbacks
- The interface SurfaceHolder.Callback has methods:
- surfaceCreated()
- surfaceDestroyed()
- Add an implementation of SurfaceHolder.Callback to the SurfaceView
- Could then start and stop the app thread within this
- However, I chose to start and stop it in the onResume and onPause methods of the main Activity
- Can you think of an advantage of this way?

Writing you own Real-Time Apps

- The simple bubble game demonstrates some important concepts
- However, it is missing an important feature:
- It has no proper model of internal game states – the game is always running until the time runs out at which point the model stops updating (though the thread keeps running)
- Discussion question: how would you model and transition between game states?
- (e.g. ready, playing, paused, gameOver, ...)

Summary: Key Android Concepts

- SurfaceView (View to extend to give independent threaded access for drawing)
- SurfaceHolder
- Provides convenient locked access to underlying view
- Use of threads for parallel execution
- Use of Threads and locking for smooth and efficient real-time apps such as games
- Simple app discussed above provides a useful base to build on
- Use helper classes such as Vector2d where appropriate
- Some slides/Code by Simon Lucas