

INTERESTING CULTURAL ARTEFACTS	THREADS	SYNCHRONISING THREADS	SURFACE VIEWS	DISCUSSION
--------------------------------	---------	-----------------------	---------------	------------

Threading and Surface Views

CE881: Mobile and Social Application Programming

Spyros Samothrakis

February 15, 2016

1 / 40

INTERESTING CULTURAL ARTEFACTS	THREADS	SYNCHRONISING THREADS	SURFACE VIEWS	DISCUSSION
--------------------------------	---------	-----------------------	---------------	------------

Interesting Cultural Artefacts

Threads

Synchronising threads

Surface Views

Discussion

2 / 40

INTERESTING CULTURAL ARTEFACTS	THREADS	SYNCHRONISING THREADS	SURFACE VIEWS	DISCUSSION
--------------------------------	---------	-----------------------	---------------	------------

THEME: “MULTI-THREADING”

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

3 / 40

INTERESTING CULTURAL ARTEFACTS	THREADS	SYNCHRONISING THREADS	SURFACE VIEWS	DISCUSSION
--------------------------------	---------	-----------------------	---------------	------------

WEEKLY PROPAGANDA: IDE SHORTCUTS (IDEA)

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

4 / 40

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

5 / 40

ANDROID AND THREADING

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

6 / 40

PROCESS LIFECYCLE

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

7 / 40

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

8 / 40

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 threads 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, };
```

9 / 40

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

10 / 40

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

11 / 40

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())
```

12 / 40

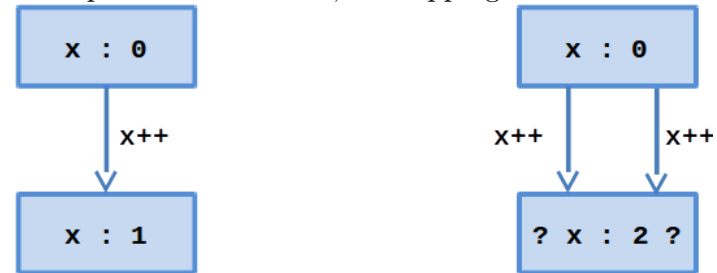
STOPPING THREADS

- ▶ `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()`

13 / 40

SINGLE TO MULTI THREADED

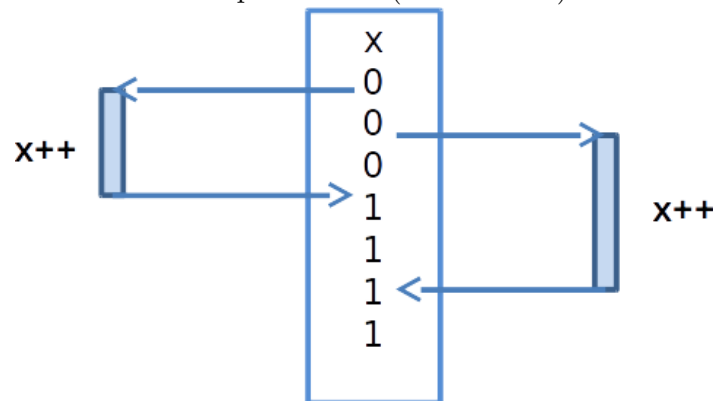
- ▶ Multiple flows of control, overlapping code AND data



14 / 40

THREAD INTERFERENCE

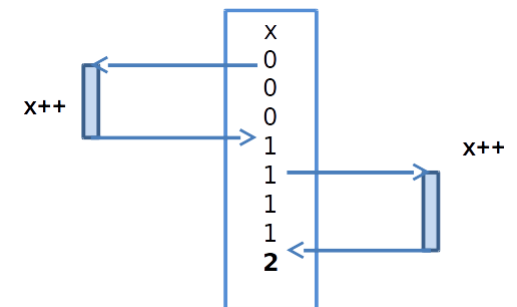
- ▶ Threads may interfere when modifying the same data in an uncontrolled way
- ▶ Result can be unpredictable (think `x+=1`)



15 / 40

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



16 / 40

JAVA EXAMPLE

```
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();
        }
    }
}
```

17 / 40

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);
}
```

18 / 40

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)

19 / 40

FIXED

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

20 / 40

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

21 / 40

ANDROID: SURFACE VIEW

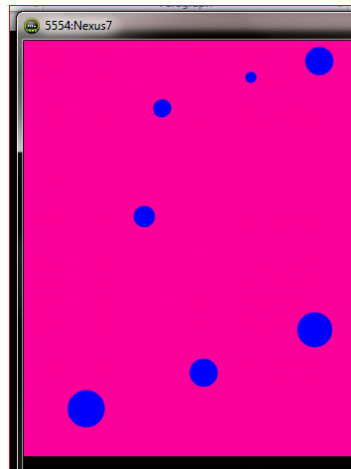
- ▶ We've seen how improper management of multi-threaded access to shared resources can cause problems
- ▶ If you do this when using a SurfaceView in Android:
- ▶ The App may crash
- ▶ Disaster!
- ▶ Five seconds of unresponsiveness will...

22 / 40

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



23 / 40

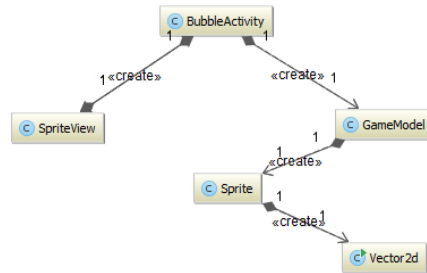
MODEL-VIEW-{CONTROLLER, PRESENTER}

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

24 / 40

OVERVIEW OF CLASSES - SHOWING DEPENDENCIES

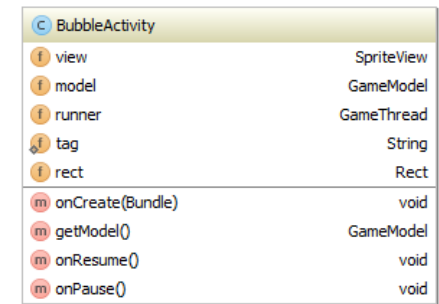
- ▶ At this stage no inheritance in App classes
- ▶ Though some inherit from appropriate Android classes
 - ▶ Which ones?
- ▶ Let's look at each in turn
- ▶ Is a class missing from the diagram?



25 / 40

BUBBLEACTIVITY EXTENDS ACTIVITY

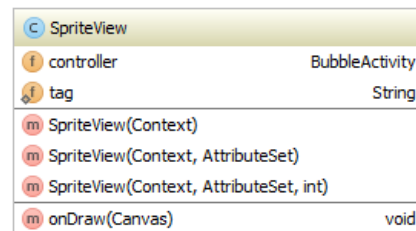
- ▶ Standard entry point for app
- ▶ Overrides onCreate()
- ▶ Creates a new SpriteView object
- ▶ Sets the current layout to that object
- ▶ Starts and stops thread in onPause and onResume



26 / 40

SPRITEVIEW EXTENDS SURFACEVIEW

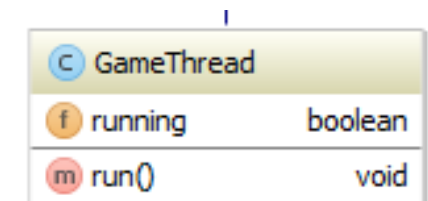
- ▶ Draws the sprites in the model
- ▶ Also handles onTouch and onClick events
- ▶ Some strange things happen if you only override one of these!
- ▶ I had to override both to get them working!



27 / 40

GAMETHREAD EXTENDS THREAD

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



28 / 40

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

29 / 40

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)

Sprite	
i s	Vector2d
i v	Vector2d
i rad	float
f random	Random
<hr/>	
m Sprite()	
m update(Rect)	void
m draw(Canvas, Paint)	void

30 / 40

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

Vector2d	
f x	float
f y	float
<hr/>	
m Vector2d()	
m Vector2d(float, float)	
m Vector2d(Vector2d)	

31 / 40

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

32 / 40

OLD GAME THREAD (USES POSTINVALIDATE)

```
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();
            }
        }
    }
}
```

33 / 40

THE NEW VERSION: SPOT THE DIFFERENCE!

```
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();
            }
        }
    }
}
```

34 / 40

AND THE DRAW METHOD ...

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

```
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);
    }
}
```

35 / 40

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);
    }
}
```

36 / 40

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

37 / 40

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?

38 / 40

WRITING YOUR 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, ...)

39 / 40

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

40 / 40