

MULTI-THREADING & ANIMATION PROGRAMMING IN JAVA (SE, ME, ANDROID)

Mobile
Application
Development
Barend
Scholtus
17/11/2011

CONTENT

- Multi-threading in Java
- Using multiple threads in a game
- Input handling
- Updates vs. Rendering
- Real-time animation loop

MULTI-THREADING IN JAVA

- Multiple tasks at the same time
 - Handle keyboard/touchscreen input
 - Render and display a frame
 - Receive data from network
 - Read/write data to local storage
- Some tasks are inherently slow, and other tasks cannot wait for it to complete
 - Network vs. Rendering

MULTI-THREADING IN JAVA (CONT.)

- Multi-tasking

- Multiple processes loaded and running
- Managed by the OS
- Have their own address space
- May interoperate via OS, files, network sockets, ...
- High overhead

MULTI-THREADING IN JAVA (CONT.)

- Multi-threading
 - One thread is a (normal) sequence of instructions
 - Multiple threads within a program execute independently
 - Threads share data
 - Synchronisation mechanisms to ensure data integrity and to prevent locks
 - “Low” overhead
- May be the basis of utilising multiple CPU cores
- GPUs use multi-threading for massive parallelisation

MULTI-THREADING IN JAVA (CONT.)

- Swing (and Java ME) programs use multiple threads
 - Thread(s) to handle system calls for **Frame** instances
 - Event handler thread
 - Timer classes for doing background tasks or repetitive tasks
 - Programmer-defined threads

EXAMPLE CODE

How to
create and
use a
Thread in
Java (ME)

MULTI-THREADING IN JAVA (CONT.)

- The essence of creating and starting a new (background) Thread in Java

```
// create and start a new background Thread that runs
// the run() method in an anonymous Runnable class
new Thread(new Runnable() {
    public void run() {
        // do this
    }
}).start();
```


USING MULTIPLE THREADS IN A GAME

- Discussion:

Given that fewer threads are better (faster), what is a minimal set of threads we need for an interactive, animated, networked game with AI?

USING MULTIPLE THREADS (CONT.)

- You may have concluded that:
 - interactive → there is input, influencing game state/physics
 - animated → the game state/physics is updated at a fixed rate
→ render as many frames as possible
 - networked → we need to get other players' state/physics,
which may be slow and unpredictable
 - AI → AI' state/physics is updated at a fixed rate

USING MULTIPLE THREADS (CONT.)

- Animation requires a thread that updates game state and AI state at a fixed rate, and considers player's input before each update
- AI may run in a separate thread to promote multi-core usage, but this requires additional synchronisation
- Networking requires a separate thread that updates local game state whenever data is received (requires synchronisation)

INPUT HANDLING

- Windowing toolkits often provide high-level APIs for event-handling
 - keyboard, mouse, touchscreen
- Ideal for implementing the MVC pattern
- Impose unacceptable performance penalties for animated games, especially for limited devices
 - object creation, garbage collection, memory fragmentation
- Use low-level APIs to use only what you need with minimal overhead

UPDATES VS. RENDERING

- Assume a simple main gameloop:

```
repeat forever:
```

```
    handle input
```

```
    update game state
```

```
    render a frame
```




- The trouble is:
 - How to synchronise game updates to time?
 - How to guarantee game updates will happen even if rendering is slow?
 - How often to update the game state (per second) to make animation smooth?

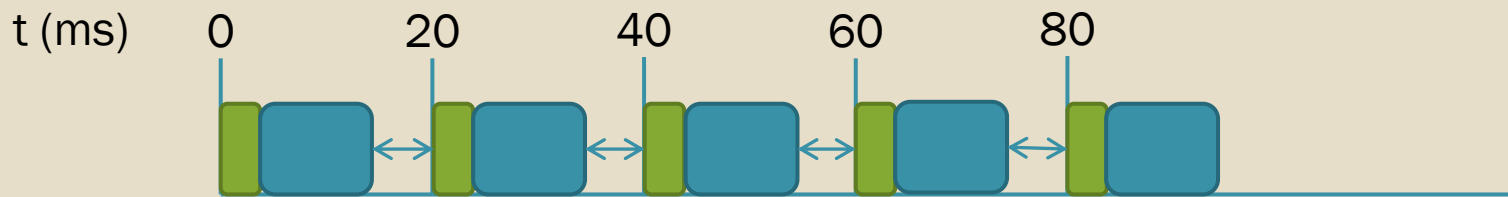
UPDATES VS. RENDERING

- Updates to the game state/physics need to happen at a fixed rate
 - Update methods assume a fixed interval → simpler, faster
 - If rendering is slow, do multiple updates to “catch up”
 - 25-50 times per second
- The (complicated) alternative is to determine how much time has passed since the last update and extrapolate how much the game state should change
- Rendering should occur immediately after the game state has been updated
- Divide time in fixed-length “slots” to synchronise to time
- In every slot, perform each task once

UPDATES VS. RENDERING

- Target 50 frames/sec: $1000\text{ms} / 50 \text{ fps} = 20 \text{ ms/frame}$

- Game update: 
- Rendering: 
- Time left: 



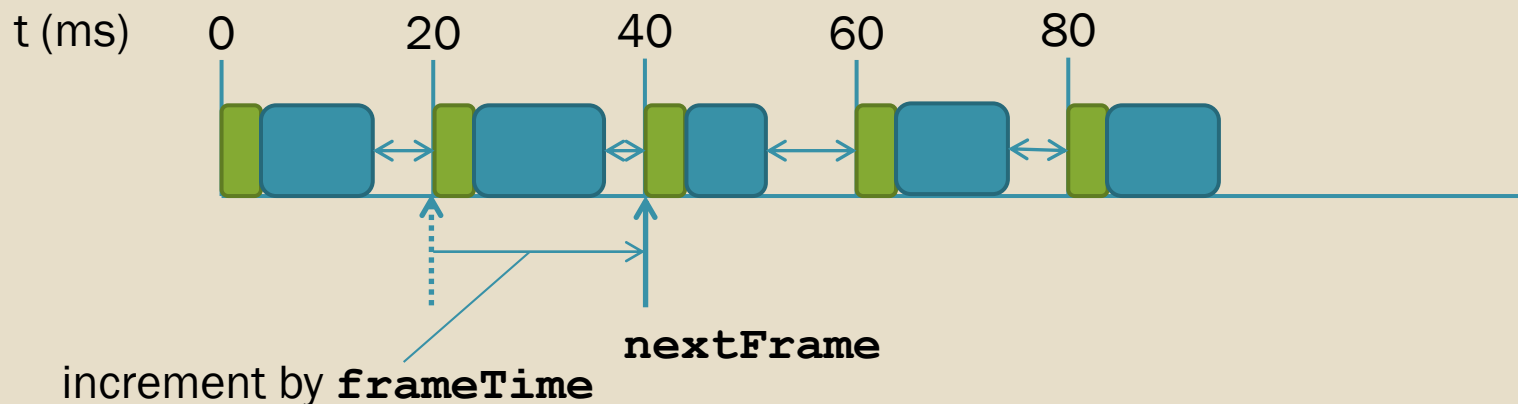
- The animation loop sleeps during the “time left”, e.g. $\sim 15 \text{ ms}$

EXAMPLE CODE

Simple
animation
loop
&
How to
handle
input

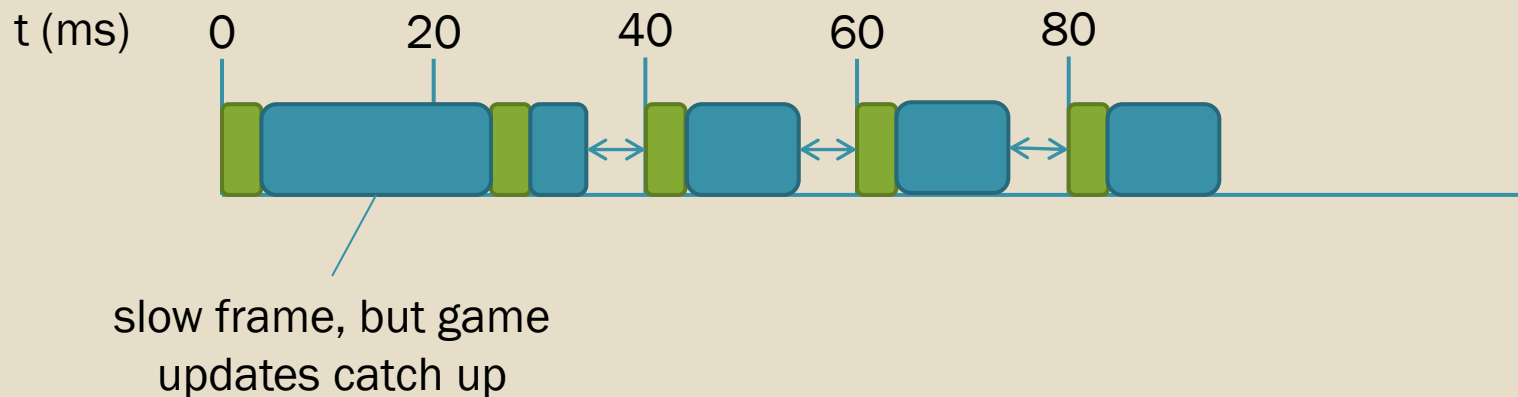
REAL-TIME ANIMATION LOOP

- Sleeping for a fixed amount of time is a waste of time
- It doesn't synchronise the loop to time
 - No guaranteed number of game state updates
- Faster/slower devices have different results
- Solution:
 - Keep track of the **nextFrame** time
 - Determine **sleepTime** based on **nextFrame** and current time
 - Update **nextFrame** time by adding the duration of a frame



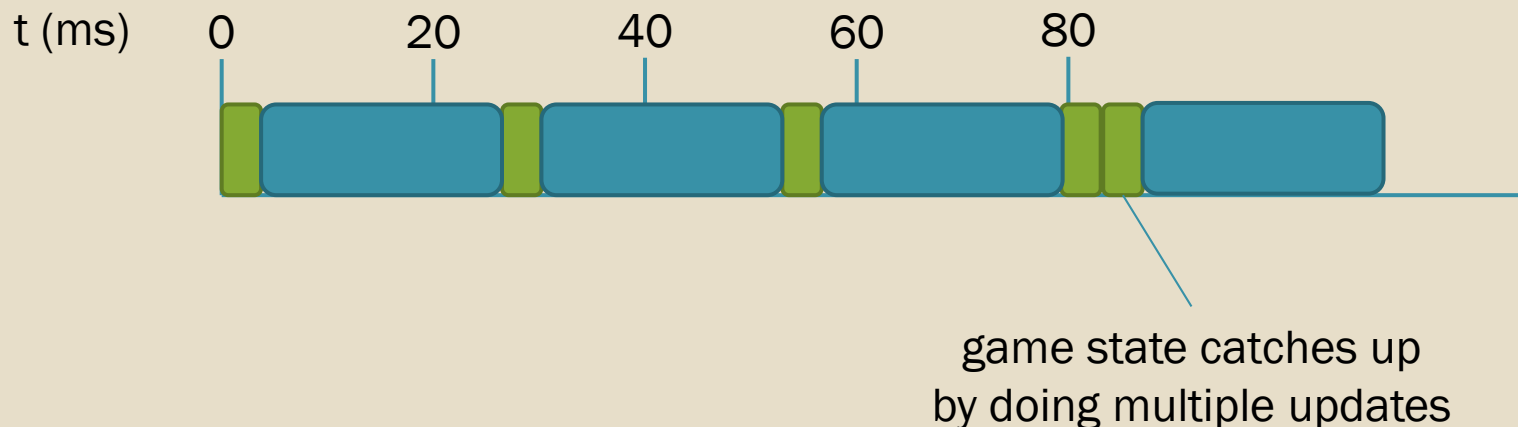
REAL-TIME ANIMATION LOOP

- If updates or rendering are slow occasionally, **sleepTime** ≤ 0, so animation will “catch up” and is (almost) independent of device speed



REAL-TIME ANIMATION LOOP

- If updates or rendering are even slower, running the animation without any sleep time will still not help, and the game state/physics will fall behind.
- Solution:
 - Do multiple updates until the game state has “caught up”
 - In code: do update until nextFrame is in the future
 - Always at least 1 update
 - The frame rate will be lower than the “update rate”



EXAMPLE CODE

Real-time
animation
loop

FINAL THOUGHTS

- **sleepTime** and **nextFrame** are not accurate
 ➔ use **double** for accuracy
- Add statistics to check updates/sec and frames/sec
- Your lecturer has a **mad.lcdui.game** package with an animated **GameCanvas** base class
- Any questions?
- Now apply this to your game!