Optimization Strategies for Mobile Game Development

“Premature optimization is the root of all evil (or at least most of it) in programming.”
- Donald E. Knuth (1974)

A presentation by Maya Posch

Defining 'optimization'

- Steps of optimization:
  - architecture
  - design
  - implementation
  - testing/debugging
  - maintenance
- Preventing bottlenecks and flaws

Premature Optimization?

- Optimizing is part of a good design process.
- Optimization is about understanding the platform.
- Balancing performance with ease of maintenance and room for extending.
- Optimize the appropriate things in the right phase.
- Avoid careless 'optimizations'.

Optimize appropriately

- *“We* should *forget about small efficiencies, say about 97% of the time: premature optimization the root of all evil. Yet we should not pass up o opportunities in that crucial 3%*” Donald E. Knuth (1974)
- Focus on the easiest changes with the biggest performance pay-offs.
- Preventing bottlenecks is also optimizing.

Optimal architecture

- Architecture has to fit the use case(s) and scenarios within the constraints of the platform
- Minimize communication between architectura elements.
- Minimize dependencies.
- Document the intentions, assumptions and motivations behind the architecture.

Optimal design

- Translate the architecture blocks into design components.
- Minimize communication between design components.
- Minimize dependencies.
- Document the intentions, assumptions and motivations behind the design.

Optimal implementation

- Minimize external dependencies.
- Evaluate external dependencies for resource usage/performance.
- Minimize communication between modules (methods, classes, tiers, etc.).
- Document intentions, assumptions and motivations behind the implementation.

Optimal testing/debugging

- Hot spot testing: idle and varying loads.
- Resource usage monitoring:
  - CPU
  - memory
  - network
  - storage
- Testing across varying hardware configuration

Optimal maintenance

- Document changes to the architecture, design and/or implementation: document intentions, assumptions and motivations.
- Don't make optimizations without first properly testing the intended change.
- Use hot spot testing and resource monitoring t track down performance issues.

Optimizing is understanding

- How can one optimize that which one doesn't understand?
- Blindly applying third-party libraries without understanding how they function: the new root all evil?
- Understand dependencies: avoid dependency hell.
- Always ask 'How does it work?'.

Mobile platforms

- Limited resources.
- High resource usage not appreciated (i.e. battery life, mobile data charges).
- Restrictive frameworks and APIs.
- Optimization has high priority.

Mobile gaming optimization

- Minimize use of GPU, RAM, CPU, network.
- Optimize concurrency.
- Make power usage part of the usage scenario
- Use dynamic generating of resources.
- Explore use of efficient compression algorithms like LZMA.

Android game development

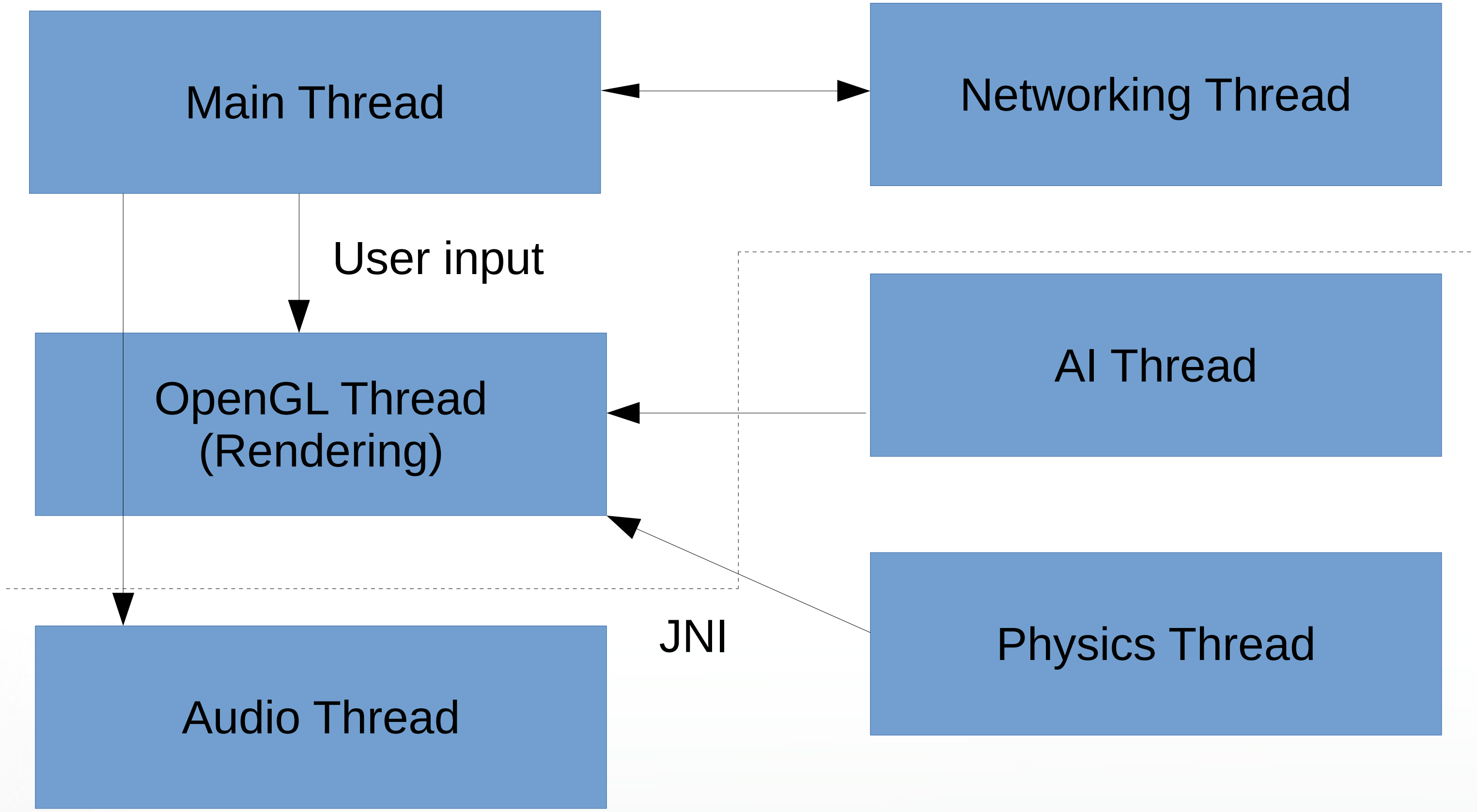
- Wide variety of devices:
  - Displays: 426x320 to 1920x1080.
  - RAM: 512 MB to 4+ GB.
  - OS APIs: Android 2.3.3 to 4.4.
  - All hardware sensors are optional; APIs differ per Android version.
  - Available codecs/decoding hardware differs per device.
  -

Android concurrency

- Threads for long-running tasks (AI, physics, etc.).
- Handlers for cross-thread communication (Messages and Runnables).
- AsyncTask for short async tasks.
- Main (UI) thread one should rarely touch.
- OpenGL context implicitly running on its own thread.

Game threading layout

- Activity:



Regular vs Native Activity

- NativeActivity: convenience class for pure nativ (C/C++) projects.
- Simplifies project setup and maintenance.
- No performance increase: still uses the JNI.
- Runs the native code in its own thread.

Java-less Android project

```
<!-- This .apk has no Java code itself, so set hasCode to false. -->
<application android:label="@string/app_name" android:hasCode="false">

    <!-- Our activity is the built-in NativeActivity framework class.
         This will take care of integrating with our NDK code. -->
    <activity android:name="android.app.NativeActivity"
            android:label="@string/app_name"
            android:configChanges="orientation|keyboardHidden">
        <!-- Tell NativeActivity the name of our native library -->
        <meta-data android:name="android.app.lib_name"
                android:value="native-activity" />
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
```

Java vs native threads

- Java threads are built upon native threads.
- Native threads don't have GC lag.
- Java threads are more convenient, native threads are more powerful.
- Android native threads API is incomplete pthreads implementation (Bionic).
- No pthreads read/write locks, pthread_cancel(), condition variables, etc.

Going native (API)

- Very complete API, also for Android-specific features.
- Lack of NDK documentation makes development and maintenance harder.
- Rapid development due to wealth of existing C/C++ code.
- Avoids locking code into Android, enables easy porting to other (mobile) platforms.

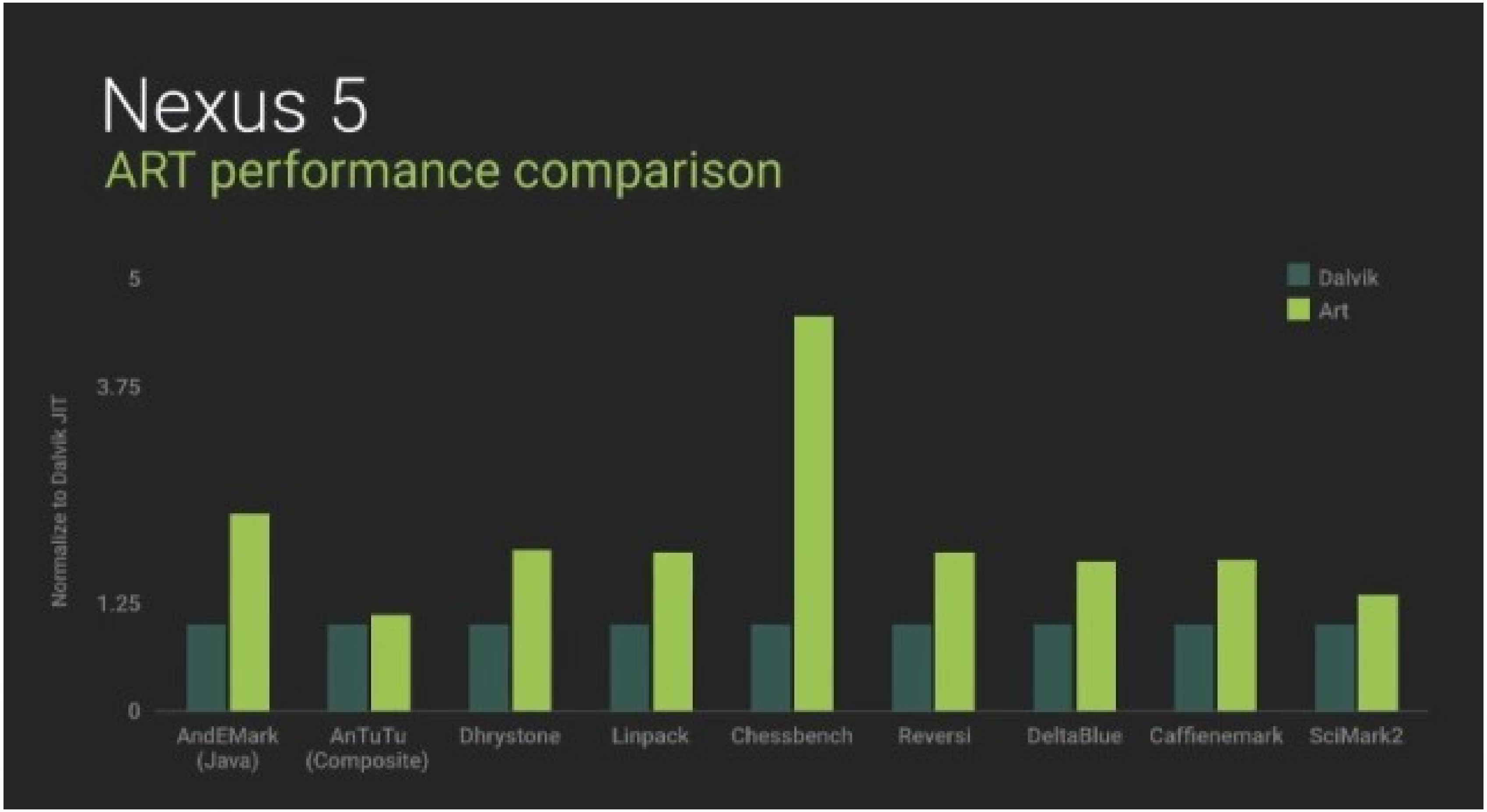
Android runtime challenges

- Java is slow. Code doing a lot of calculations (physics, AI, etc.) is often over 10x faster when implemented as native (NDK) code.
- JNI is slow. The overhead of passing objects to and from the VM is significant.
- JNI is complex. The glue logic required by the JNI is non-trivial.

ART replacing Dalvik

- Bytecode versus native code.
- Just In Time versus Ahead Of Time.
- ART is roughly 2 times faster.
- ART uses less memory and less CPU.
- Fewer garbage-collector (GC) runs.
- New allocator: ~10x boost.
- Default runtime on Android 5.0 (Lollipop).

Relative performance



ART GC performance

- One GC pause of ~3 ms (Dalvik ~54 ms).
- At 30 FPS, 1 frame lasts ~33 ms.
- After 10 GC runs, ~1 dropped frame (Dalvik 1
- More predictable performance.
- GC with Dalvik always leading to dropped frames.
- Minimizing RAM usage important with Dalvik.

ART allocator

- Introduction of 'Large Object Space' for bitmap etc. to prevent fragmentation.
- Fine-grained locking for MT applications.
- ~10x speed boost for memory allocations.
- Addition of background heap defragmentation

Optimization limits

- Usually not realistic to optimize a third-party framework or library.
- With a framework like AndEngine or Libgdx on has to understand its own optimizations.
- Extreme Cross-Platform (tm) approaches like Unity3D make optimization really hard.

Finding bottlenecks

- Android is a rather closed platform.
- Dalvik Debug Monitor System (DDMS) for profiling.
- ART runtime provides far more detailed output (sampling profiler).
- Traceview can visualize the traces produced by DDMS profiling.
- Logcat logging.

Thanks for listening

- Any questions?