Google™ 09 IO

Writing Real Time Games For Android

Chris Pruett
May 2009

Did You See That Awesome Fade?

- Holy Crap!
  - The text was all sliding before that too.
  - How do they do that?!
- Right, this slide is about me, the presenter. Chris Pruett. That’s me.
- I’m a **Developer Advocate** for Android.
  - That means I advocate development for Android. Please make something. Maybe I can help you.
  - I work in Japan. 宜しくお願いします。
- Before that, I wrote code for **Lively.**
- Prior to working at Google, I made video games.
  - I shipped about 10 titles for lots of platforms: GBA, PS2, PSP, Wii.

Making Games on Android

- Writing games is awesome, and Android is awesome, so writing games on Android must be awesome^2.
- This theory required testing. So I made a game. Many lessons learned.
- Topics to cover today:
  - Why games? Why Android? I mean, besides awesome^2.
  - Game engine architecture.
  - Writing Java code that is fast. For serious.
  - Drawing stuff on the screen efficiently.
  - Tips, tricks, and pitfalls.

Who Cares About Games on Mobile Devices?

- Dude, what rock have you been living under?
  - **iPhone**: 79% of users have downloaded at least one game. (According to a report by Compete, Inc.)
  - There are more than *100 million* **Nintendo DS** devices throughout the world. (According to Nintendo, see http://www.nintendo.co.jp/ir/pdf/2009/090507e.pdf)
  - Sony's **Playstation Portable** has just passed *50 million* devices (see: http://www.computerandvideogames.com/article.php?id=208211%3fcid)
  - The **Nintendo Game Boy** and **Game Boy Advance** together account for about *200 million* devices (see http://www.nintendo.co.jp/ir/library/historical_data/pdf/consolidated_sales_e0806.pdf).
- Portable games appeal to a huge audience, but traditional phones have not been good game devices.
- Game tech is extremely specific.
  - If your platform can support good video games, other apps should be a walk in the park.

Why Games on Android?

- Traditional PC and console game markets have been come so high-risk that only a few companies can even compete.
- Smaller games on non-traditional platforms are steadily gaining popularity with both traditional gamers and folks new to the medium.
  - See also: Nintendo Wii, iPhone, Flash, XBLA, etc.
  - Lower risk = more interesting and diverse content!
- Android provides an avenue for innovative games across a wide, internet-savvy audience.

Why *This* Game for Android?

- My goal is three-fold:
  - To produce a fun game for Android.
  - To produce a reusable, open source game engine to allow others to make fun games for Android.
  - To stress test our platform with regards to games; only publically-available code and tools are to be used.
- I went for an orthodox 2D side-scroller.
  - Parallax layers, tile-based worlds, animated sprites, etc.
  - Pushes all the right hardware buttons: input systems, OpenGL ES, sound, etc.
  - Proper game is feasible with one 20% engineer (that's me) for six months and 1 full time artist for four months.
  - Tools are pretty easy to write.
  - Popular and recently under-served genre.

Agenda 2: The Return

- Topics to cover today:
  - ~~Why games? Why Android?~~
  - **Game engine architecture.**
  - Writing Java code that is fast.
  - Drawing stuff on the screen efficiently.
  - Tips, tricks, and pitfalls.

Insert Here: Picture of man holding giant gun that is also a chainsaw.

Quick Demo

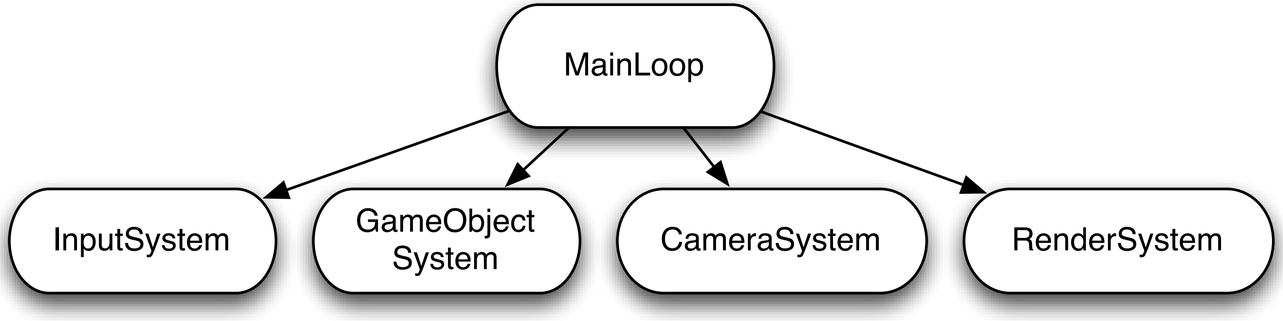
(video goes here)

Note that this is a work in progress. All bugs are mine.

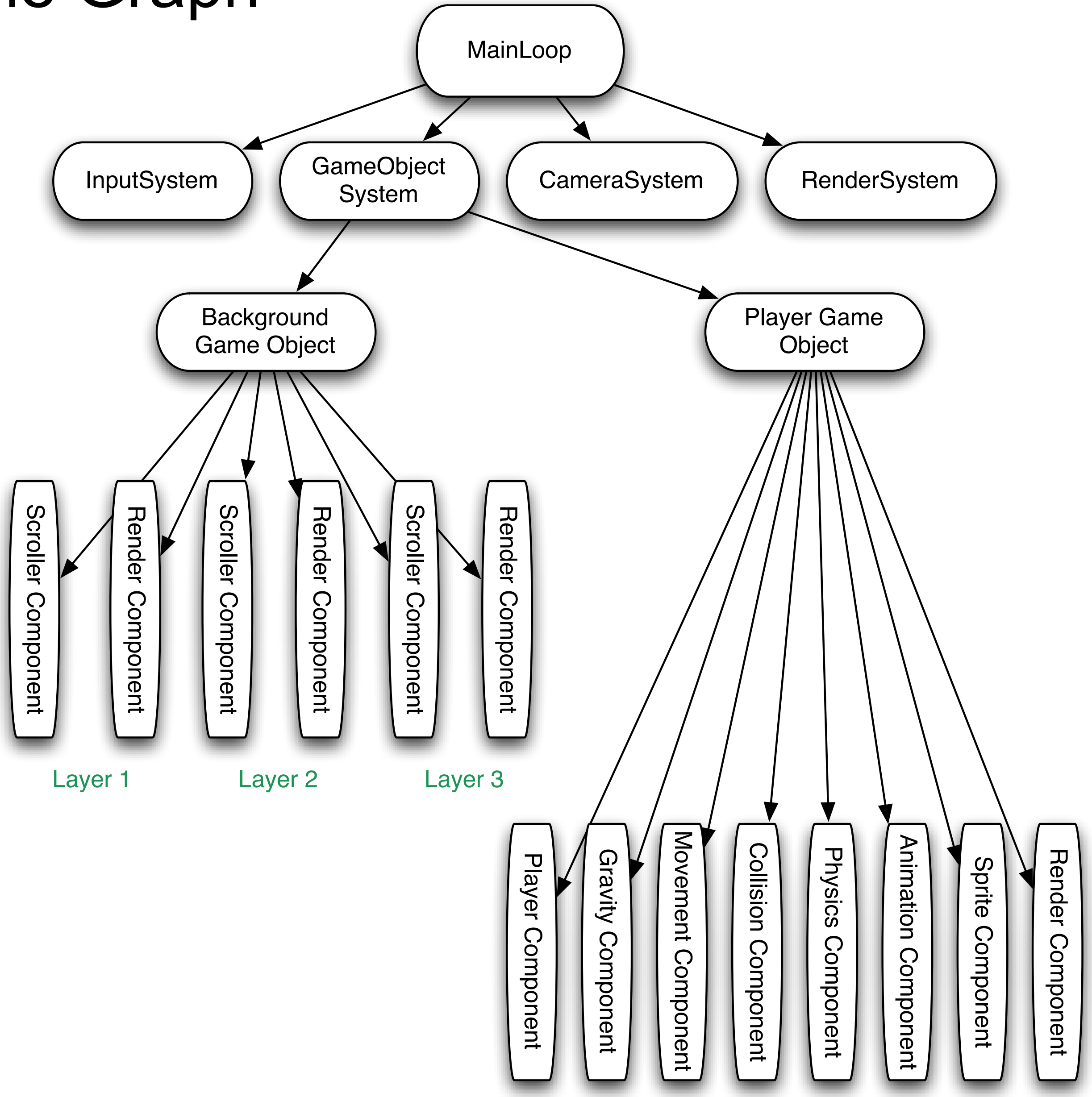
Game Engine Architecture

- Lots of approaches, but the basic problems are similar.
- My approach is a “game graph” that can be traversed every frame, taking time and motion events as input and resulting in a list of things to draw to the screen.
  - The root of the graph is the “main loop.”
  - Children of the main loop get called once per frame.
  - Children further down the tree might get called once per frame, depending on their parent.
  - “Game objects” are children of a “game manager” node, which only visits children within a certain activity bubble around the camera.
  - Game objects themselves are sub-graphs of “game components,” each implementing a single characteristic or feature of the object.

Game Graph


MainLoop
InputSystem
GameObject System
CameraSystem
RenderSystem

Game Graph

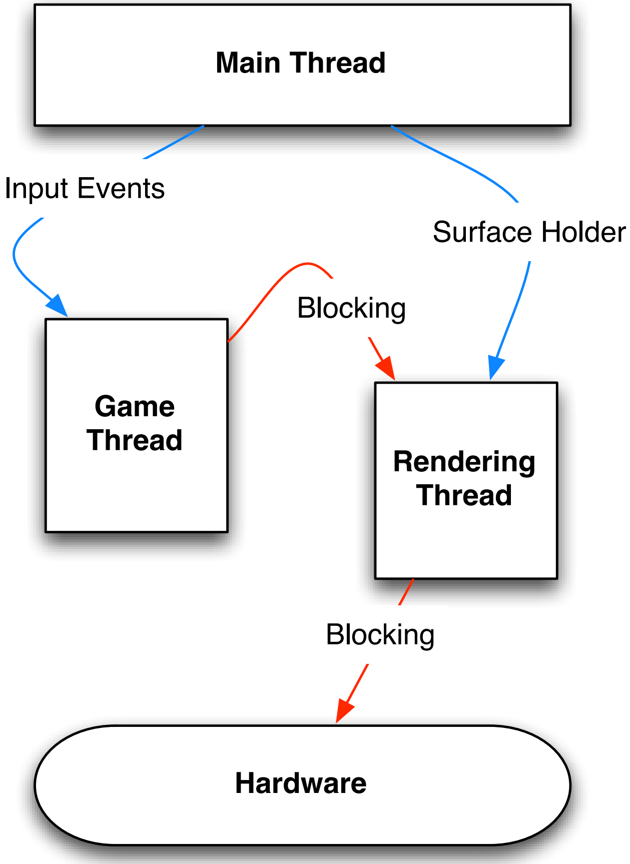

MainLoop
InputSystem
GameObject System
CameraSystem
RenderSystem
Background Game Object
Player Game Object
Scroller Component
Render Component
Scroller Component
Render Component
Scroller Component
Render Component
Layer 1
Layer 2
Layer 3
Player Component
Gravity Component
Movement Component
Collision Component
Physics Component
Animation Component
Sprite Component
Render Component

Game Engine Architecture

- At least, that's how I do it.
- Important point: time is passed to each node in the graph so that *framerate independent motion* is possible.
- Second important point: this system collects things to draw in a draw list each frame, *but it doesn't actually draw anything to the screen.*

Game Engine Architecture - Nice Threads, Yo

- I have three threads:
  - The main thread spawned by the Android activity.
    - Responsible for bootstrapping the game and receiving input events.
    - Mostly dormant.
  - The game thread.
    - Handles all non-rendering parts of the game: physics, AI, collision detection, animation, etc.
    - Owns the game graph.
  - The rendering thread.
    - Controlled by a SurfaceHolder.
    - Just runs through its draw list and fires off commands to OpenGL every frame--knows nothing about the game content.



Agenda III: The Series Continues

- Topics to cover today:
  - ~~Why games? Why Android?~~
  - ~~Game engine architecture.~~
  - **Writing Java code that is fast.**
  - Drawing stuff on the screen efficiently.
  - Tips, tricks, and pitfalls.

I Love Coffee, I Love Tea

- I am pretty much a C++ engineer.
  - In fact, I wrote my first line of Java ever for this project.
  - So you should take my advice on the topic of Java-specific optimization with a grain of salt.
  - Still, I have done a lot of optimization work in the last six months, and maybe at a level that most Java apps do not require, so maybe I can offer some useful tidbits.
- Writing real-time games is an exercise in finding the perfect balance between flexibility and performance.
- My (non-language-specific) approach is:
  - Start with the simplest possible implementation, but design for future rework.
  - Choose flexibility over speed every day of the week... until the gameplay is damaged.
  - Profile early and constantly.

Step One: Memory Management

- Never allocate memory. Or release it.
  - Well, never allocate during gameplay.
  - The GC will stop your game for **100 ~ 300 ms**. That's death for most real-time games.
- Revised: Allocate as much as possible up front, don't release things until you have natural pause time. Invoke the GC manually when you know it to be safe.
- Use DDMS to track allocations.
  - Hey, Java allocates memory CONSTANTLY. Ugh!
  - Hidden allocations in things like enum.values(), Class.getClassName(), Iterator, HashMap, Arrays.sort() etc etc etc.
  - Some of these are not really avoidable.

Allocation-Related Java Language Contortions

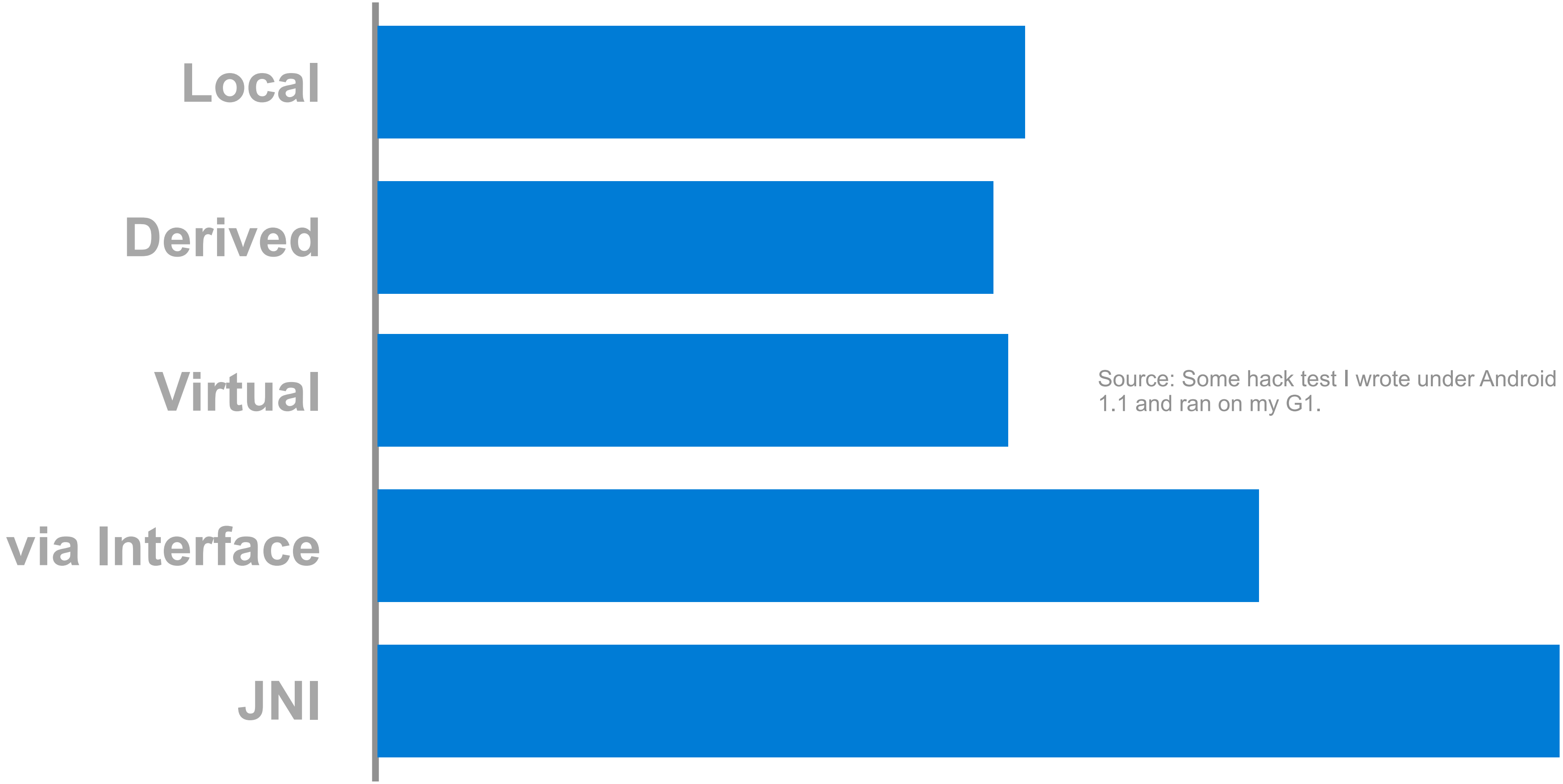
- Treat Java like C++
- Lots of the standard Java utility objects allocate memory.
  - Collections are out, as are iterators.
  - Forget about enums (they are really heavy-weight anyway).
  - Arrays.sort() and similar functions
  - Anything returning a String that needs to be read-only (like Class.getXXX(); man, I miss me some const).
- DDMS is your tool to name and blame.
- Better Java engineers than I might be able to supplement existing frameworks with non-allocating implementations.

Step Two: Don’t Call Functions

- Ok, that’s extreme. But function calls are not cheap and you can’t rely on inlining.
- Use static functions whenever possible.
- Don’t call functions through an interface. 30% slower than regular virtual functions!
- Accessors and Mutators are my bestest friends in C++, but they have no place in your Java inner loop.
- Be wary of JNI functions.
  - In particular: lots of gl.glXX() functions.

Don’t Call Functions: A Graph

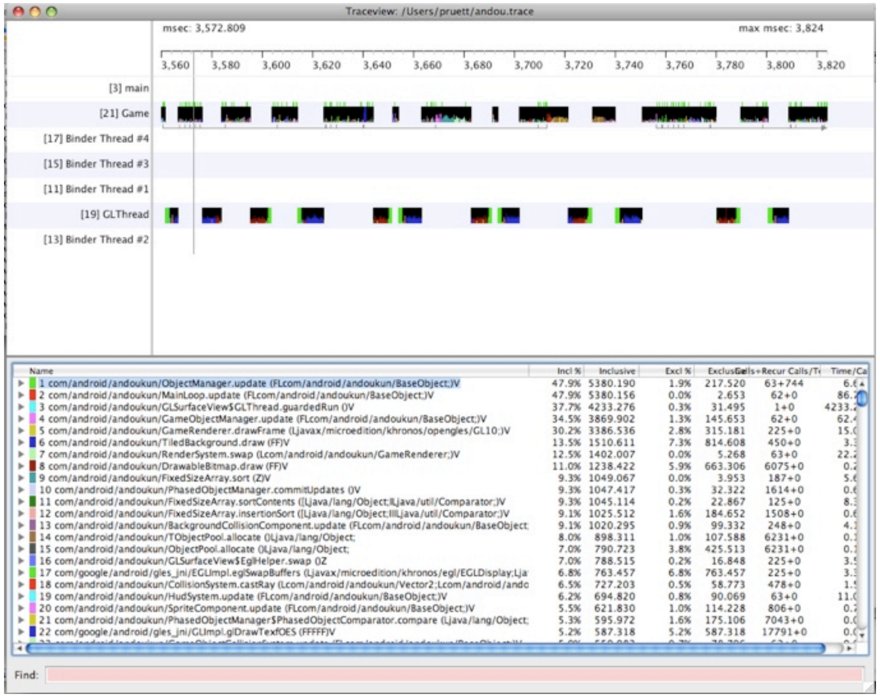
- Take this with a grain of salt, not a very scientific test.



Source: Some hack test I wrote under Android 1.1 and ran on my G1.

Step Three: Other Tips

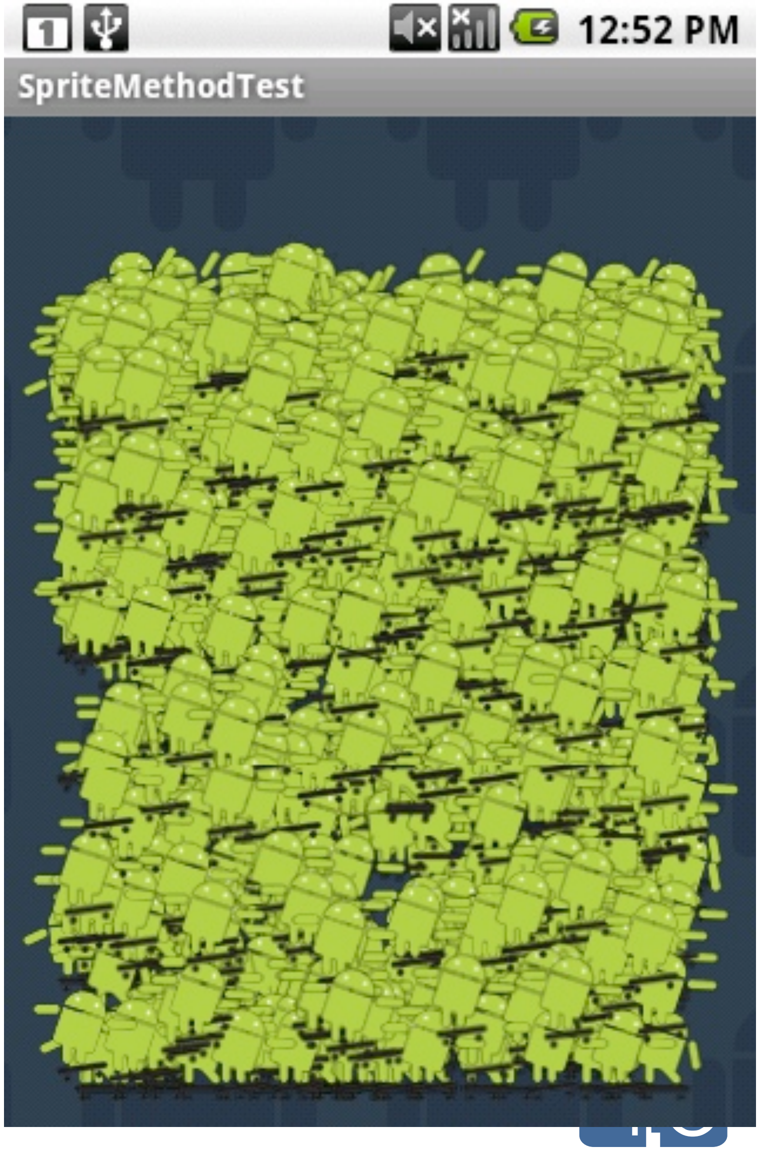
- Use local variables, especially in inner loops.
- Use the final keyword on fields whenever you possibly can.
- Some hardware (like the G1) has no FPU, so avoid float math.
- Always use Log.d() or similar rather than System.out.print(). Printing takes time!
- Use Traceview!



Name	Incl %	Inclusive	Excl %	Exclus	Calls+Recur Calls/T	Time/Ca
1 com/android/andoukun/ObjectManager.update (FLcom/android/andoukun/BaseObject;)V	47.9%	5380.190	1.9%	217.520	63+744	6.6
2 com/android/andoukun/MainLoop.update (FLcom/android/andoukun/BaseObject;)V	47.9%	5380.156	0.0%	2.653	62+0	86.2
3 com/android/andoukun/GLSurfaceView$GLThread.guardedRun ()V	37.7%	4233.276	0.3%	31.495	1+0	4233.2
4 com/android/andoukun/GameObjectManager.update (FLcom/android/andoukun/BaseObject;)V	34.5%	3869.902	1.3%	145.653	62+0	62.4
5 com/android/andoukun/GameRenderer.drawFrame (Ljavax/microedition/khronos/opengles/GL10;)V	30.2%	3386.536	2.8%	315.181	225+0	15.0
6 com/android/andoukun/TiledBackground.draw (FF)V	13.5%	1510.611	7.3%	814.608	450+0	3.3
7 com/android/andoukun/RenderSystem.swap (Lcom/android/andoukun/GameRenderer;)V	12.5%	1402.007	0.0%	5.268	63+0	22.2
8 com/android/andoukun/DrawableBitmap.draw (FF)V	11.0%	1238.422	5.9%	663.306	6075+0	0.2
9 com/android/andoukun/FixedSizeArray.sort (Z)V	9.3%	1049.067	0.0%	3.953	187+0	5.6
10 com/android/andoukun/PhasedObjectManager.commitUpdates ()V	9.3%	1047.417	0.3%	32.322	1614+0	0.6
11 com/android/andoukun/FixedSizeArray.sortContents ([Ljava/lang/Object;ILjava/util/Comparator;)V	9.3%	1045.114	0.2%	22.867	125+0	8.3
12 com/android/andoukun/FixedSizeArray.insertionSort ([Ljava/lang/Object;IILjava/util/Comparator;)V	9.1%	1025.512	1.6%	184.652	1508+0	0.6
13 com/android/andoukun/BackgroundCollisionComponent.update (FLcom/android/andoukun/BaseObject;	9.1%	1020.295	0.9%	99.332	248+0	4.1
14 com/android/andoukun/TObjectPool.allocate ()Ljava/lang/Object;	8.0%	898.311	1.0%	107.588	6231+0	0.1
15 com/android/andoukun/ObjectPool.allocate ()Ljava/lang/Object;	7.0%	790.723	3.8%	425.513	6231+0	0.1
16 com/android/andoukun/GLSurfaceView$EglHelper.swap ()Z	7.0%	788.515	0.2%	16.848	225+0	3.5
17 com/google/android/gles_jni/EGLImpl.eglSwapBuffers (Ljavax/microedition/khronos/egl/EGLDisplay;Lja	6.8%	763.457	6.8%	763.457	225+0	3.3
18 com/android/andoukun/CollisionSystem.castRay (Lcom/android/andoukun/Vector2;Lcom/android/ando	6.5%	727.203	0.5%	58.773	478+0	1.5
19 com/android/andoukun/HudSystem.update (FLcom/android/andoukun/BaseObject;)V	6.2%	694.820	0.8%	90.069	63+0	11.0
20 com/android/andoukun/SpriteComponent.update (FLcom/android/andoukun/BaseObject;)V	5.5%	621.830	1.0%	114.228	806+0	0.7
21 com/android/andoukun/PhasedObjectManager$PhasedObjectComparator.compare (Ljava/lang/Object;	5.3%	595.972	1.6%	175.106	7043+0	0.0
22 com/google/android/gles_jni/GLImpl.glDrawTexfOES (FFFFF)V	5.2%	587.318	5.2%	587.318	17791+0	0.0

Agenda Part 4: Even More Agenda

- Topics to cover today:
  - ~~Why games? Why Android?~~
  - ~~Game engine architecture.~~
  - ~~Writing Java code that is fast.~~
  - **Drawing stuff on the screen efficiently.**
  - Tips, tricks, and pitfalls.

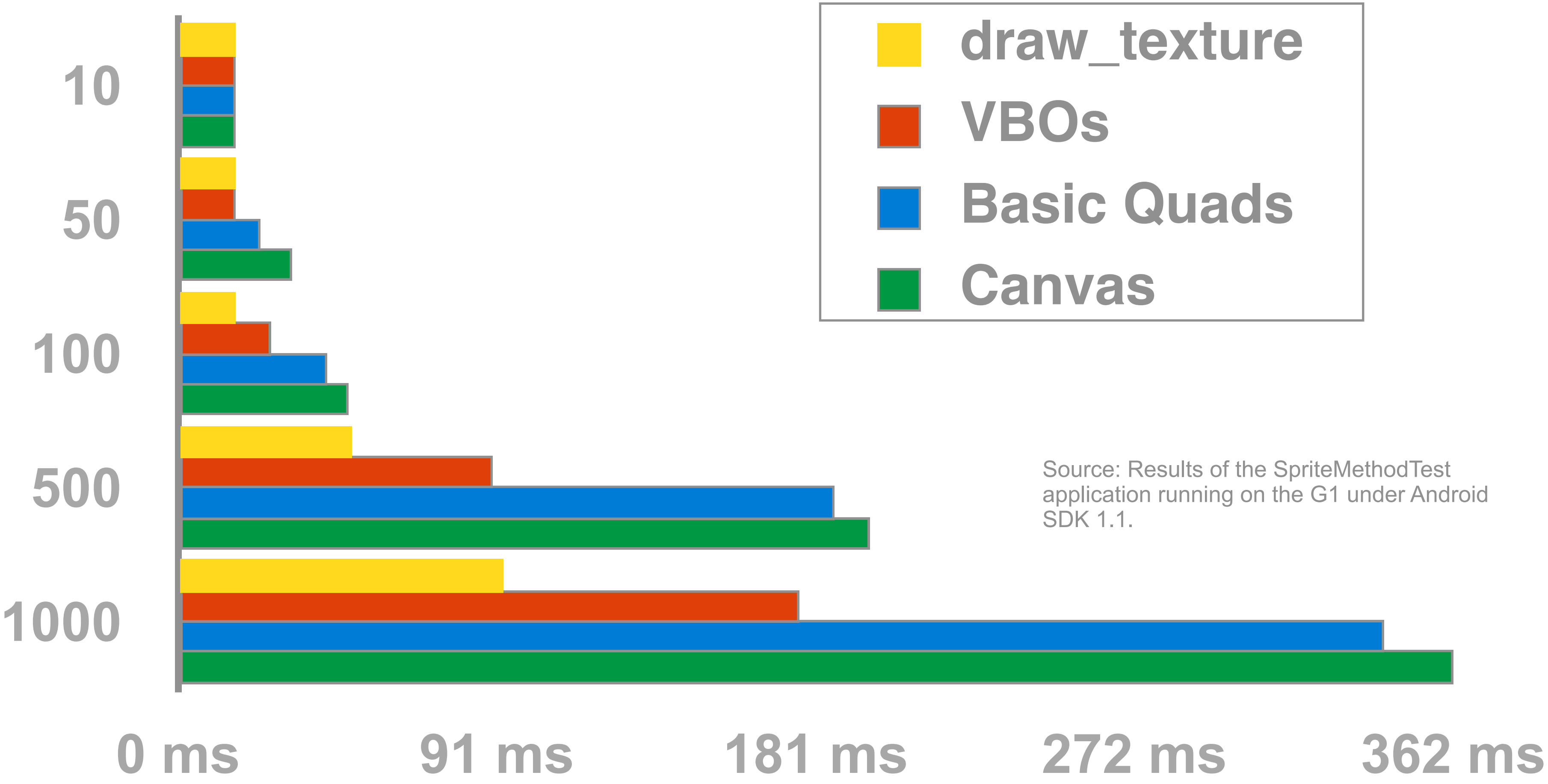


Screenshot from SpriteMethodTest, 1000 sprites, OpenGL DrawTexture extension. Runs at around ~10 fps on the G1.

Android Drawing Methods

- Canvas:
  - CPU-based 2D drawing. Used for most of the Android UI.
  - Fast for a small number of blits. (~ 10 sprites < 16 ms in my tests)
  - Very straightforward and easy to use.
- OpenGL ES
  - 2D and 3D drawing.
  - Hardware accelerated on some platforms (like the G1).
  - Scales to much more complex scenes than Canvas.
  - Various 2D drawing methods:
    - Quads with orthographic projection
    - VBO quads (on supported platforms)
    - draw_texture extension (on supported platforms)
  - Only OpenGL ES 1.0 is guaranteed.

OpenGL vs Canvas for 2D drawing (G1)



Source: Results of the SpriteMethodTest application running on the G1 under Android SDK 1.1.

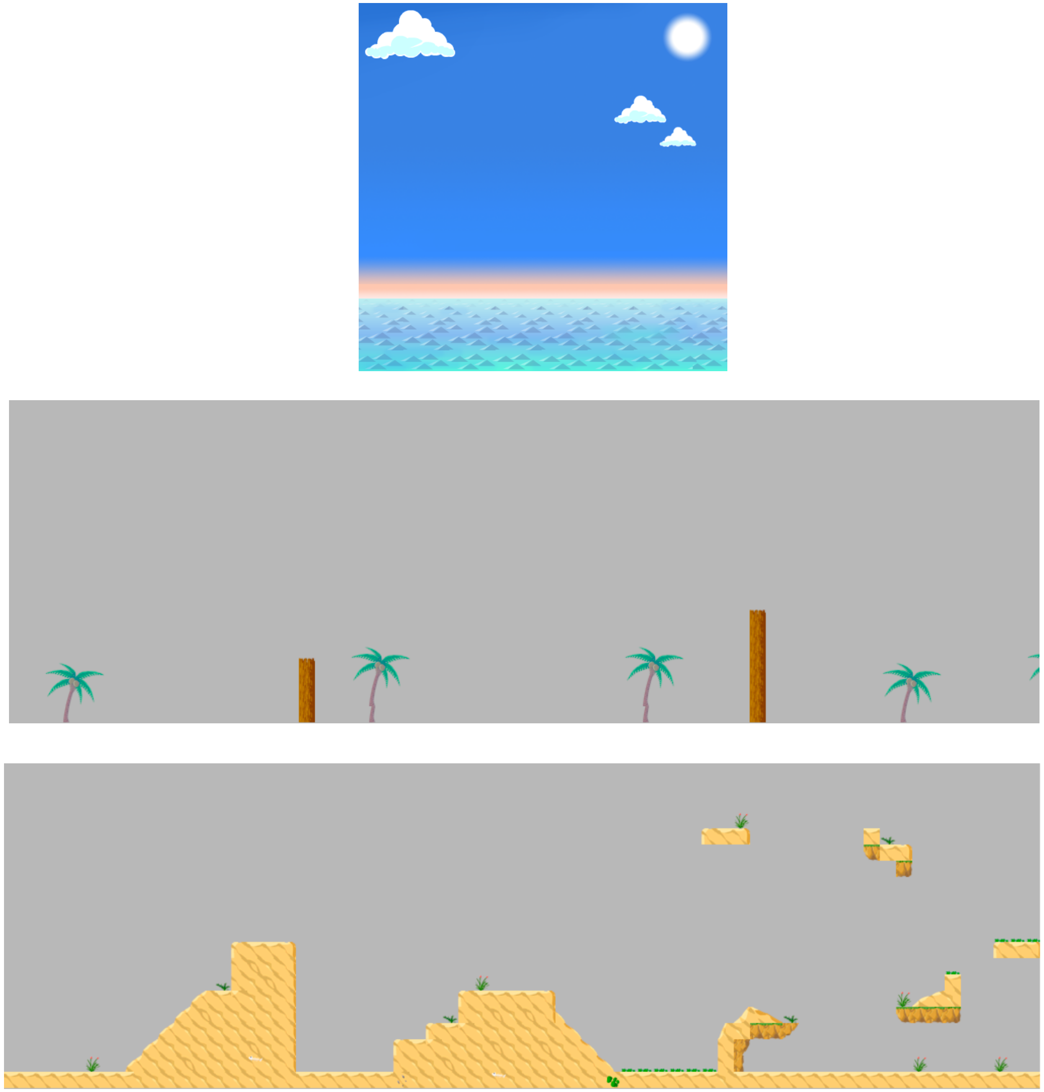
Which Method is Best?

- Clearly, OpenGL ES + the draw_texture extension is fastest for 2D drawing on the G1.
  - But that extension isn't guaranteed to be supported on all platforms. You MUST check glGetString(GL10.GL_EXTENSIONS) before using it.
- However, Canvas isn't bad if...
  - You have very few things to draw every frame, or
  - You don't have to draw every frame (puzzle games, etc).
- SpriteMethodTest provides a framework for swapping between drawing methods (and timing them) based on my game code.
  - http://code.google.com/p/apps-for-android/

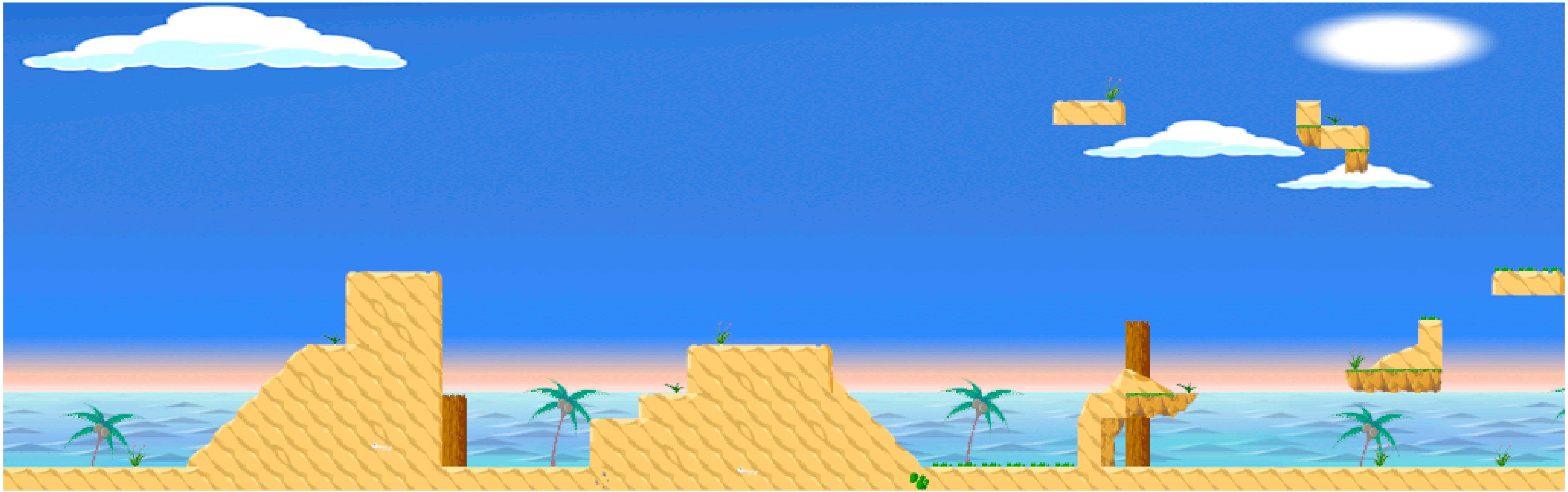
Case Study: Drawing Tiled Backgrounds

- Replica Island uses three background layers: one large static image, one mid ground tile-map, and the foreground tile map.
- The tile map layers are regular grids of 32x32 tiles. That means 150 tiles to draw per layer on any given frame (worst case).
- More layers would be nice, but drawing the background is the single most expensive operation in the Replica Island code.
  - Actually, the single static image is quick. It's just one 512x512 texture.
  - It's the tile maps that eat frame time, either on the CPU or the GPU, depending on the drawing method.

Layered Parallax Backgrounds

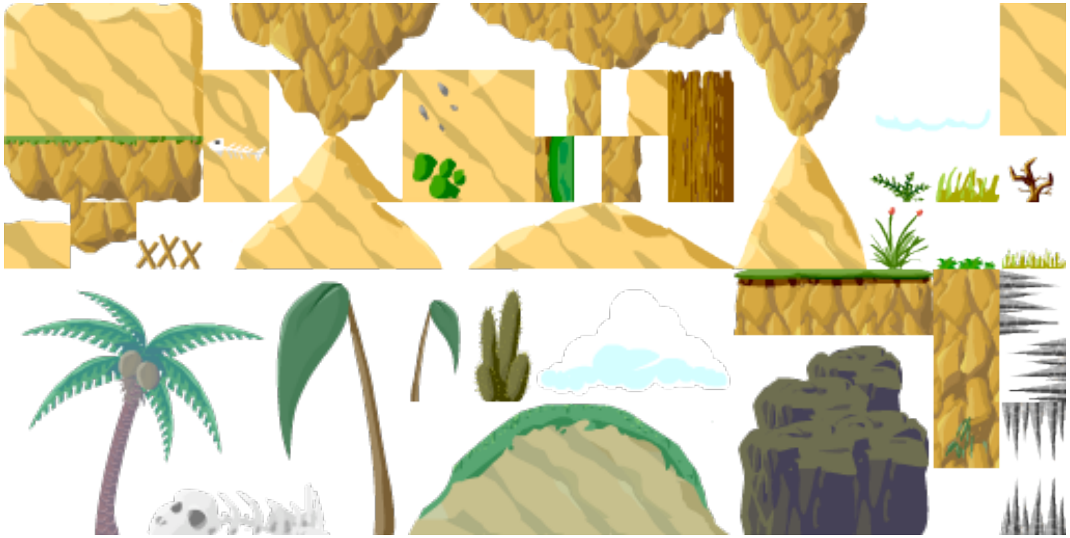


Layered Backgrounds Composited



Case Study: Drawing Tiled Backgrounds

- First idea: use a single atlas texture as the tile map, use draw_texture to draw, and adjust cropping of the texture to select individual tiles.
  - Only one glBindTexture() call needed, which is good.
  - Otherwise, this was a terrible idea. glTexParameteriv() is expensive.
  - Remember kids, state change is costly on fixed-function hardware!



Case Study: Drawing Tiled Backgrounds

- Second idea: draw the tiles individually with draw_texture extension using a bunch of really small textures.
  - This actually works pretty well once some easy optimizations are made (RLE the tile map, etc).
  - But it’s still a lot of calls to OpenGL. 400 calls to glDrawTexfOES() in the worst case, plus lots of superfluous glBindTexture calls.
  - Average case is good: less than 16 ms for the hardware to draw everything. But the actual submission of tiles to OpenGL is variable depending on the sparseness of the layer: 2 - 10 ms.
  - Worst case is bad: 9 - 13 ms to make GL calls and 19 - 23 ms to draw. Way unacceptable.

Case Study: Drawing Tiled Backgrounds

- Third idea: Make “meta tiles” out of vertex arrays, uv them to the atlas texture, and project them using orthographic projection.
  - Initial results were in line with the “basic vert quads” test in SpriteMethodTest: terrible.
  - Switching to VBOs sped things up a lot.
  - Lots of advantages to this approach: only one glBindTexture() call, very few total calls to other GL commands (only four meta tiles to draw per layer per frame).
  - Worst case situation (two layers with no empty tiles) is much faster than the draw_texture approach. CPU spends only 3 - 5 ms submitting to GL and drawing takes less than 16 ms.
  - Average case is the same! This makes it slightly slower than draw_texture in the average case (most maps are very sparse).

Case Study: Drawing Tiled Backgrounds

- Last ditch idea: pre-render the tile map and cut it up into meta tile textures, which can be drawn with VBO quads or draw_texture.
  - I haven't actually implemented this yet.
  - Level size will be restricted by total amount of VRAM if I use this method (unless I dynamically load textures).
  - High main memory cost too.
  - But, given all available information, drawing this way should be blazing fast.
  - I'm close enough to 60hz now that this probably won't be necessary.
- Future improvements to Android's GL interface (or the G1 GL driver) might render these optimizations unnecessary.

Requiem for Agenda #5

- Topics to cover today:
  - ~~Why games? Why Android?~~
  - ~~Game engine architecture.~~
  - ~~Writing Java code that is fast.~~
  - ~~Drawing stuff on the screen efficiently.~~
  - **Tips, tricks, and pitfalls.**

Performance Tips

- Touching the screen causes your app to be flooded with MotionEvents.
  - This will kill your framerate.
  - Sleep in the onTouchEvent callback to slow the flood. 16 ms is a good place to start.
- The mechanics of pausing and resuming are complicated when it comes to OpenGL, as the contents of VRAM are not always maintained. GLSurfaceView solves this for you.
  - GLSurfaceView handles the state machine correctly.
- ATITC texture compression (supported on the G1) can be a big win if you are bus-bound.
- Android 1.0 - 1.5 failed to throw errors if you try to use VBOs with indirect buffers. Result: unpredictable crashes.
  - Solution is to just use direct buffers.

Game Design Tips

- Keep your application as small as possible. 2 ~ 3 mb is ideal.
- Now is the time for competent games. Customers are hungry for them and the platform is capable!
- As of right now there are already Android devices without a hardware keyboard (HTC Magic) or trackball (Samsung i7500). Don’t rely on these input devices for game play (or support them all).
- The key to success Android Market is *quality*. Making a high-quality game is the way to be considered for the Featured Apps section too. Polish polish polish!
- You have always-on internet at your disposal. Use it!

Questions?

Google™ 09 IO