

Hi – I’m Lars Bishop, and I’m an engineer in NVIDIA’s Tegra Developer Technologies group. Today, I’d like to share some experiences we’ve had while helping developers bring high-end game content to Android and Tegra

Intended Audience



- **Programmers looking to get the most out of Android for high-quality, high-end games**
 - Likely a C/C++-centric game developer
 - Wants to see some real technical strategies and tips
- **Assuming some basic familiarity with Android**
- **Disclaimer/Background**
 - Java was not a core language when I went to school
 - Java is/was not my first or native language
 - This frames many of these issues

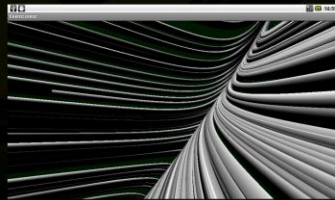
NVIDIA Confidential

My intended audience today is programmers, specifically those looking to move existing 3D game content to Android, or developing new 3D game content for Android. The target programmer here is likely a C/C++ programmer at their core, and I'll try to provide some real, tangible tips for them. I'm assuming a basic familiarity with Android, but not with native coding on Android. Note that much of this talk comes from the fact that my background and that of the developers I've seen are not expert Java or Java plus native programmers, and so some of the items I bring up may be less of a shock for seasoned Java hackers. Maybe.

What is Driving this Presentation?



- “Hello World!” met “The Real World”
- Initial simple sample apps worked well
- But cracks began to show in real-world game development
- Fixes to these issues became a part of our NVIDIA Android Samples Pack



So what specifically drove this presentation? Well, NVIDIA DevTech had created some basic “Hello World” samples for Android and OpenGL ES 2.0

We then began to directly assist developers in bringing large, complex, high-end content to Android

And we quickly learned that there are some challenges in making the larger systems work well

We fed these lessons into our NVIDIA Android Samples Pack and into this presentation

Agenda



- Elements of a compelling mobile experience
- Android Native (C/C++) gaming tips
- Graphics content tips
- Performance tools

NVIDIA Confidential

The highest level agenda is that we'll describe a few aspects of compelling mobile experiences, and then spiral down into some Android cases relating to them. We'll focus on native code structure in Android. We'll spend some time on specific graphics performance tips, and close with a discussion of some of NVIDIA's publicly available performance tools.

Android as a Gaming Platform



- Android has a lot going for it as a gaming platform
- Shader-based 3D APIs (OpenGL ES 2.0)
- Java and Native (C/C++) code supported
- Expressive input devices
- Very open platform
- Wide range of Android devices (also a challenge...)



NVIDIA Confidential

Even from a purely technical view alone, Android as a gaming platform has a lot going for it. Latest-generation mobile 3D with shaders, Java and C/C++ code supported and the ability to use multitouch, accelerometer and other input devices. It is an open platform, and there are a wide range of exciting Android devices in the market and coming to market (which can also be a challenge for an app developer)

Keys to a Compelling Mobile Game



- **User Experience on a mobile device is complex**
 - **Power efficiency**
 - Don't waste the user's battery
 - Convergence devices are "shared functionality" devices
 - **Integration with and respect for the device's core function**
 - User should be confident that everything will "just work"
 - App and device responsiveness
 - **Rendering quality and performance**
 - User expectations are set by non-mobile devices!
 - Screen densities and sizes are rising

NVIDIA Confidential

User experience on a mobile device is many-faceted. We'll focus on three basic parts today. Power-awareness is important on mobile devices, where gaming is but one functionality and power is a shared resource. Integration with the parts of the OS that provide the device's core functionality, like telephony, must also be considered. App and device responsiveness is pivotal. And finally, rendering quality and performance must do its best to come up to the standards of non-mobile devices that consumers use for comparison. That's quite a challenge!

Power Management



- **Use cycles wisely!**
- **Apps that drain the battery won't be popular for long**
- **Don't spin needlessly**
 - Be event-driven
 - Throttle frame rate reasonably
- **Use the efficient subsystem for the job**
 - Vertex shaders instead of CPU for skinning
 - Video core instead of CPU for video decode

NVIDIA Confidential

Use cycles wisely; you are likely one part of a converged device experience, even if you are a full-screen game. If a game drains a user's battery, they'll be afraid to play it for long out of fear that they won't have enough battery left to do the other things they need to later; make calls, surf the web, send email. If your game isn't being played, it can't self-advertise nearly as well!

In addition, just because you wrote a great engine and can run at 60fps does not mean you should. On mobile, an efficient game engine is just as much about saving power as it is about framerate.

Also, put the work you are going to do onto the most efficient core for the task. Offload the general CPU core by using the GPU and the video hardware.

Power Management: Android



- **Example: Keeping the screen on and bright**
- **Android includes multiple methods:**
 - Activity timers: fine if the app is interaction-focused
 - “Wake Locks”: very aggressive, not recommended
 - Window flag: more integrated with app focus
- **Use the right one; the least invasive for your needs**
- **Don’t keep brightness on all the time in your game**
 - Clear brightness flags between levels, in menus, etc
 - Or put these modes in windows with no power hints
 - Consider the needs of the game in each interaction phase
- **But don’t ignore them or skip them**

NVIDIA Confidential

A quick look at power management in Android by looking at the 600 milliwatt gorrilla: screen and backlight. Android has aggressive power management as you’d expect, and the screen and backlight are core targets for power management

Where possible, given no user input, user settings or app hints, it’ll turn off the screen as often as it can

But Android adds several methods to keep the screen on

User input just naturally keeps the screen on as you’d expect

But games and other media apps often want the screen on during input inactivity (for example hiding still in an alcove in a shooter game)

One item that caught my eye initially were Android’s Wake Locks. They allow apps to manually force the screen brightness be up even if the app is not focused. But this requires that the app manually manage it.

An Android window can also set a window flag that indicates the window wants the screen on whenever the app is focused

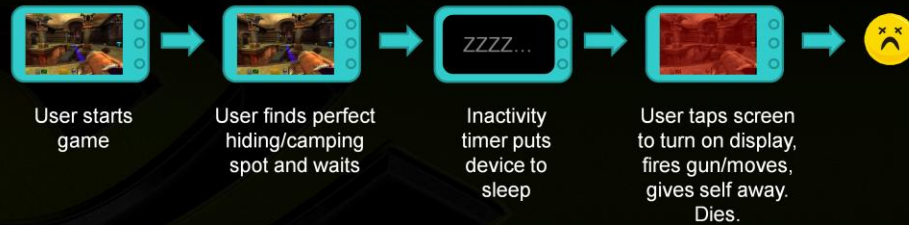
Even 20s of no backlight here or there in a casual game can make a big difference to the user’s battery experience

failure to use them can lead to annoying timeouts and screen darkening during non-input phases of the game; cutscenes, etc

Activity Timeout Only



- Relying on input is risky, since the user may have aggressive timeouts (15s)



- (Yes, this is extreme and requires some other incorrect coding, but you get the point...)

NVIDIA Confidential

Activity timeouts alone may work well for a game that is entirely based on input coming in constantly, but say the user starts a first person shooter, and then they find a perfect hiding spot. The wait there unmoving, not daring to touch the screen, waiting for their enemy to come. With no user input, the activity timer goes off and darkens the screen. In a panic, the user taps the screen a few times to wake it up. His character shoots his gun, gives away his location and he is killed. Unhappy user...

Hard WakeLock

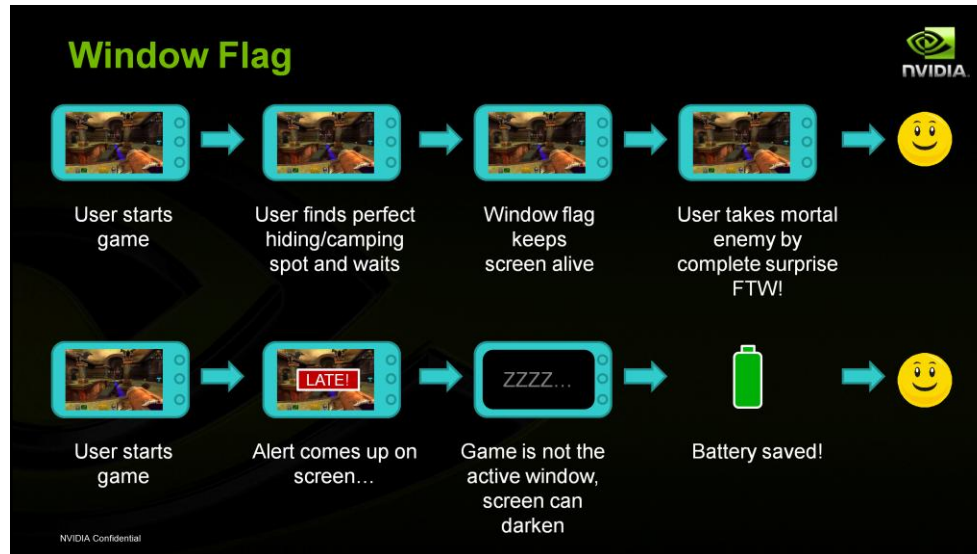


- Hard wakelocks need constant (manual) management (at best) to avoid keeping the screen bright when it should not be



NVIDIA Confidential

Hard wake locks are system level items. You lock the screen's brightness up and it stays there until your game releases it manually. But say the user is playing a game when they see a warning dialog that they are late for their important meeting. They drop the device in their pocket as-is and they run for it. Even though the active window is just a calendar dialog, your game still kept the wake lock. So the device warms their pocket with the brightness fully on for the entire meeting. At the end of the meeting, the user's battery is flat. Unhappy user. Most apps should avoid this big hammer as it can degenerate into my app equals bright.



Window flags can be used to indicate that a particular window (like your game’s main action gameplay window) should keep the screen bright. This allows the screen to go dark if your window is not the focused one for any reason. So, see these use cases. The users starts a first person shooter again, they camp out, and the game window stays the focused window, so the screen stays bright long enough for the user to see their mortal enemy, take them by surprise and kill them for the win! On the other hand, the user again sees a warning dialog that they are late for their important meeting and they run for it. Because the warning or the game’s pause window is active and not the game’s main window, the screen is allowed to dim and the user’s battery is saved.

Platform Integration



- **Know the platform events to be handled**
 - Do not want a user trying to pause their game manually when a call comes in!
 - Don't want user to lose their progress!
- **Be prepared to be swapped out or shut down**
 - Mobile OSes can be very aggressive on managing memory
 - Know if and how your OS gives warnings before using The Hammer

NVIDIA Confidential

Most modern mobile device Oses are better and better at ensuring that apps can't break the basics, like completely ignoring incoming calls

But that's just level zero of user experience

Modern mobile Oses tend to go steps further and actively manage apps out of the way for system events. The bigger challenge is for the app to ensure that the user does not lose app state when the app gets managed out

The user should be confident that your app/game will do the right thing no matter what

E.g. when a call comes in, put the game in pause mode. When the call ends, don't just unpause and leave the user scrambling to catch up – show a pause menu. That's perhaps a basic, trivial example, but shows the point.

Platform Integration: Android



- **Android applications have complex lifecycles**
- **Android tries to keep apps resident**
 - Better responsiveness
 - Not visible/accessible != process exited
- **Can kill some processes at surprising times**
 - In order to free resources
 - But clearly documented and completely handle-able
- **Android components have a carefully laid-out state graph**

NVIDIA Confidential

Android has a rather complex, but well-documented state graph for the lifecycle of application components. This is important for resource management, power management and system integration. Android tries to keep parts of inactive apps resident in memory and process space if it can to make things more responsive. Also, if it needs to, it can kill applications, even partially visible ones (although that's very rare) if it needs resources. Lets look at a part of that state graph.



Android defines numerous Activity states between which it will transition Activities as appropriate

The states shown here are merely an interesting subset


Android includes a detailed spec of what an Activity can expect to have happen to it in each state; care should be taken to understand these expectations

Note the red arrows in the diagram – these in particular can cause confusion for developers. Lets look at the paused to killed case now

<http://developer.android.com/guide/topics/fundamentals.html>

Android and “Paused” Activities



- Note the direct Paused→Killed arrow!
 - A paused Activity can be killed!
“at any time without another line of its code being executed”
 - Would you see this in testing? Maybe not
 - But the spec allows it
 - This is why `onPause` documentation recommends saving persistent state
 - If you don’t, a user could lose their data
 - E.g. game progress
 - Implement important system callbacks in your apps
- 

NVIDIA Confidential

The important bits that some new developers miss in this spec is the fact that a “Paused” Activity is “killable” – note the direct arrow in the previous state graph from paused to killed.

While the spec says that it tries to avoid doing this, the fact is that once the Activity has acknowledged being placed in a paused state (by returning from the `onPause`) callback, the OS reserves the right to kill the Activity’s process *at any time* without another line of its code being executed.

No additional warning.

This is why the `onPause` callback is supposed to save persistent state of the Activity, like game progress.

If your game’s `onPause` callback it does not save game progress and the Activity that interrupted your game suddenly needs a lot of resources,

the OS may kill off your Activity,

and your user could lose their game progress with no chance to have avoided the loss.

Which makes it a user experience issue. Activities should know and implement the callbacks like `onCreate`, `onPause` and `onResume` for best experience.

Paused Activities: the Flip Side



- You should save your application's persistent state in the `onPause` callback
- But once that returns, don't waste more cycles rendering or doing game logic
- You're almost certainly not visible
- Even if you are, the window manager will use your last-rendered frame
 - So don't bother rendering again until you are Active

NVIDIA Confidential

The flip side to the need to have code in the `onPause` callback to handle saving context. Once you are paused, do not continue rendering and running game logic. The game should not be running anyway, since the user cannot interact with it. Also, there's no need to be concerned about redrawing for any reason – the window manager is composited, and has a copy of your last-rendered frame to keep re-using until you are made Active again. So rendering or spinning on other game work is likely doing nothing but stealing cycles from the active app and wasting battery.

Performance and Porting Existing Content



- Existing set-top/desktop content is within the reach of Tegra
 - See today's developer sessions!
- This content is generally C/C++
- Core Android apps are Java-launched
- Device manufacturers can build native apps into a device's OS
 - But that's not a solution for ISVs
- But Android provides an official method of using native C/C++ code across multiple devices: Java's JNI and Android's NDK

NVIDIA Confidential

As you've already seen and will see continually throughout the day, high-end content is within reach with Android on Tegra. Most of this source content is C/C++ or based on C/C++ engines. But installable Android apps are Java-based. Well, device manufacturers can choose to build native apps into their devices, but that's not an option for most ISVs. To assist, Android provides a way for ISVs to use native code across multiple devices and OS versions; Java's JNI and Android's NDK

Android and JNI



- **Android does not allow for 100% native apps**
 - All apps are still Java-launched
 - From the Android UI PoV, apps are Java classes
 - Native code resides in shared libraries (.so's)
 - Native code is called from Java via the JNI standard
- **JNI == Java/Native Interface**
 - Java-specific, not Android-specific
 - Defines the call protocol and data interchange between Java and Native
 - Can feel very fragile at times: take care to match code
 - Is not "free" to call "across" the boundary; batch your native code

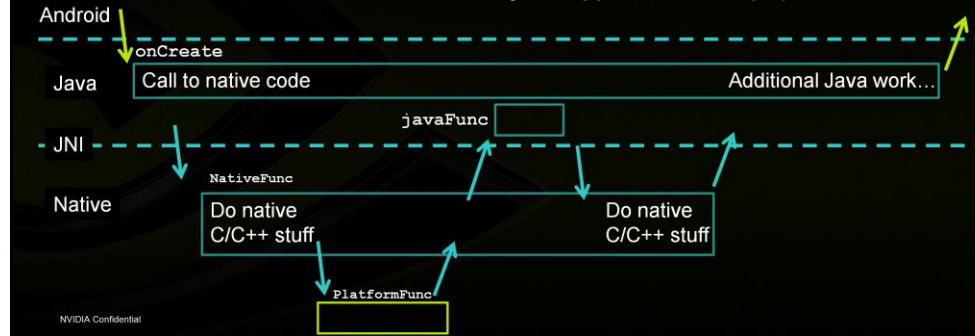
NVIDIA Confidential

All installable apps on Android are Java-launched. In terms of the Android UI and launcher, apps are Java components. But your app can install and call native code stored in shared libraries. This is done via Java's JNI or Java/Native Interface. This is a generic Java standard for calling back and forth between Java and compiled C/C++ code. It defines a calling protocol and a set of functions for native code. Sadly, it is a bit fragile, and can be pretty easy to get wrong initially. And the calls across the so-called JNI boundary are not free. But it is extremely useful

Android, Java and JNI



- We'll think in three levels:
 - Android, which manages and calls the apps/activities/etc
 - "Java", the Java class(es) that form(s) the application
 - "Native", the native code called by the app's Java class(es)

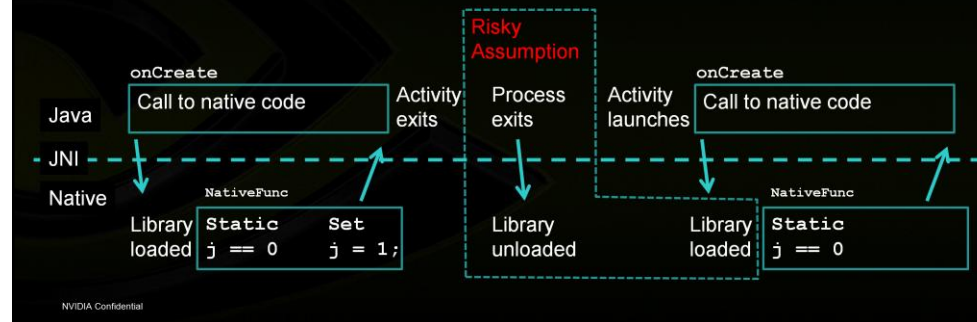


We'll use these kinds of diagrams for the rest of this presentation. Time roughly flows left to right on the diagram. We think in three levels; the android system that calls down into our Java-based Android app. The Java-based android app that can call other Java methods or can call down into the native level via JNI. Note that Java can call native code, Native code can call native system calls and Native code can call Java methods, too (we refer to that as "Calling up")!

Case Study: Android Lifespan + Native Code



- Android activity lifespan also affects native code
- The “classic” app exit/restart model is not safe to assume

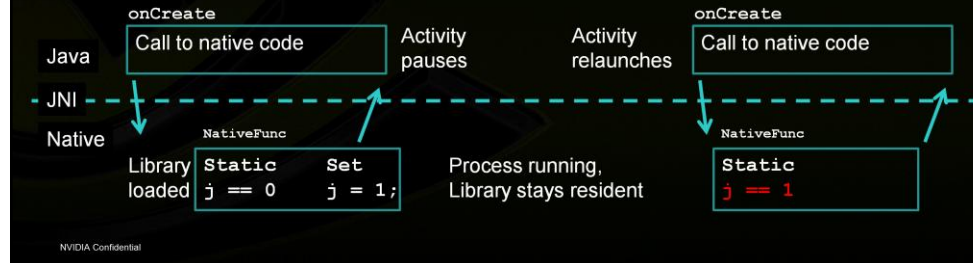


So let's see how the Android app lifecycle we introduced can affect native code. Process management in Android and other mobile OSes isn't just about killing things, but also about leaving things resident when possible for responsiveness. This can be a bit of a shock to developers if they do not pay attention to the spec and other docs. Basically, assuming that your native code's shared library will be reloaded (and thus reinitialized) on each launch is dangerous. Here, we show that assumption graphically; if we assume that when the activity is stopped, the process exits and our library is unloaded, then when we are launched again, we can assume the we'll be reloaded, and static init will get called again. This is a risky assumption.

Android Lifespan and Native Code (2)



- When the Activity restarts, static variables may *or* may not be re-initialized
- Here's what often happens



When your app's activity stops and then gets launched again, Android may have kept your process and thus your native code library resident. So, when you activity gets launched again, static initialization does not happen. See the diagram. First time through, static is zero. We set it to one and then we exit. But when we get re-launched, the library is still resident, so the data is still one! If we were assuming that the data would be zero at activity startup, bad things will happen

Android Lifespan and Native Code Conclusion



- **Best option is to avoid initializing static data automatically**
 - Don't assume that at start up, all C-style statics are 0
 - Don't use static class instances with default constructors
- **Include a function to specifically init all static data, so you can call it on each (re-) initialization of the activity**
- **Avoid in-function statics for "first-time" stuff**

```
static int firstCallToThisFunction = 1;
if (firstCallToThisFunction)
    // Do critical operation such as
    // resetting/initializing or loading..
```

NVIDIA Confidential

The best option is NOT to base your code on pre-main or static initialization. For all static or global data members, include manually-called initialization and shutdown functions, so you can manage your own lifespan independent of library loading and unloading. And be careful of "first time" statics hidden in functions. They can catch you out later.

Alternative: Android Native Code “Thunk”

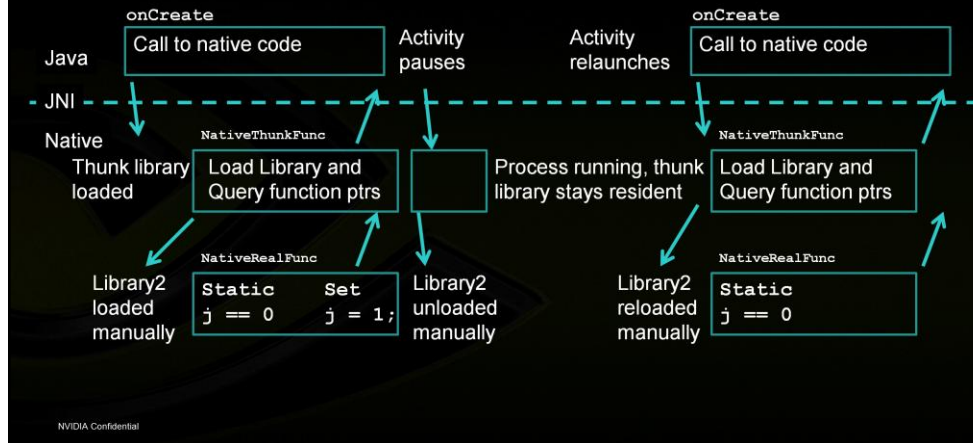


- **Android+JNI does not include Java functions to unload a shared library**
- **So you cannot manually unload the Java-referenced code**
- **But you can make the Java-referenced library a little “Thunk”**
- **Put the bulk of your code (including static inits) in another library**
- **Pack that library in your APK**
- **Do not load it from Java**
- **Load it using `dlopen`**
- **Manage its lifespan manually**

NVIDIA Confidential

One of our developers suggested another trick. They had too much code that was based on static initialization for them to even consider reworking all of it for a content port. Instead, they recognized that the number of calls into their code was very minimal. Thus, adding another indirection would not be an issue. So they placed all of their application code in an additional shared library that was in the APK but NOT loaded by Java. Then, their Java-loaded shared library would use `dlopen` and `dlclose` whenever they wanted to completely shutdown and reinitialize their statics.

Native Code Thunk in Practice



Here's how it works. The library loaded by Java is nothing more than a thunk that manually loads the main library, here called Library2. It loads that implementation library and queries the required function pointers. Now, the thunk library can call the real code via the function pointers. When the app throws the `onPause` callback, the thunk can unload the implementation library, causing post-main shutdown. The thunk stays resident. When the app is re-launched, the still-resident thunk can reload the library and cause the pre-main static initialization to run again.

Android and Threading



- **Threads are key to:**
 - Keeping Android responsive with native code
 - Unleashing the power of high-end multi-core CPUs like Tegra
- **Use threads!**
- **Current console and PC engines are already well-threaded**
- **Our terms:**
 - “JNI Thread”: a thread that enters native code by calling from the JVM
 - “Native Thread”: a thread that is entirely created and managed in native

NVIDIA Confidential

Threads are very important in Android games to keep the app and OS responsive and smooth and for getting the most out of powerful CPUs like the multicore CortexA9 in Tegra. So use them! Luckily, most modern console and PC engines not only use them, they depend on them to run well. For the rest of the presentation, we'll use the term JNI thread to refer to a thread that called down from Java through JNI to native code. The term native thread will refer to threads that are directly created and managed in native code.

Native Threads and JNI



- **Threads can and should be created as needed in native code**
- **You cannot automatically make JNI calls in native-created threads**
 - Each thread needs a JNI Environment object (JNIEnv)
- **Threads not calling down from Java must be:**
 - Registered before calling JNI or you'll crash
 - Un-registered before exit, or else the JVM may throw an exception
- **Our sample code does this in a thin wrapper for thread creation, `nv_thread`**

NVIDIA Confidential

JNI does allow the native code to create and manage threads. But you need a little more if you want to call back up to Java from a native thread. All JNI functions require you to pass a JNI Environment object. For JNI threads, that's easy, the JNIEnv is passed down in the native function call itself as the first parameter. But a native thread has none by default. And calling a JNI function with another thread's JNIEnv doesn't work. TRUST ME, I learned the hard way. So, native-created threads must be manually registered with JNI to get a JNIEnv, and must be un-registered before exiting or you'll throw an exception. Our sample code has a wrapper to do this for you...

nv_thread



- **NVIDIA DevTech sample code that wraps pthread creation so that the resulting thread is registered with the JVM**
- **Caches the all-important JNIEnv object in thread-local storage**
 - Provides function to return the calling thread's JNIEnv
- **Still has limitations:**
 - Some JNI code can only be called from JNI threads, not native
- **And can lead to some surprises for those new to JNI (like me...)**

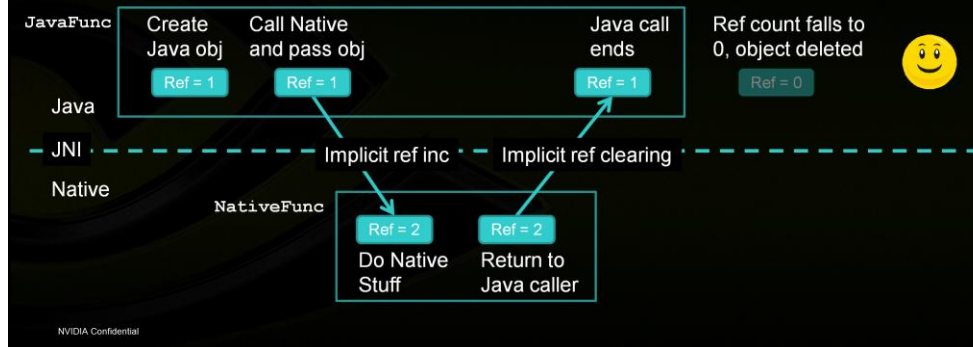
NVIDIA Confidential

Nv_thread allows you to call a pthread create-like function that automatically registers your thread, stores the JNIEnv in thread-local storage, makes it easy for you to query that JNIEnv, and un-registers your thread when you return from it. We found it to be pretty handy. It has some limitations that are really JNI limitations, and as we soon found, JNI from native threads can lead to some surprises...

Sidebar: How to Leak a Lot of Memory



- **JNI scopes based on call life Java->Native**
 - Java objects passed to Native are de-ref'ed on return from the Java-Native call



A quick sidebar on how we were able to leak a lot of memory really quickly.

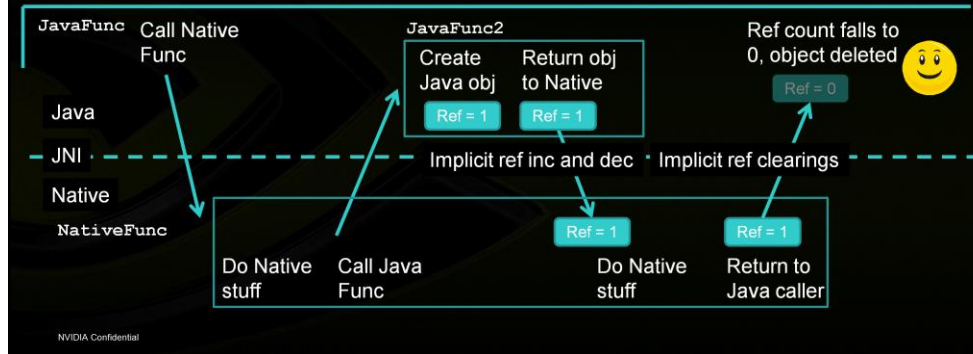
Java Objects passed via JNI to native code or queried from Java get a reference added to them to ensure they do not get deleted by the JVM during the native call. When the thread that called down from Java to Native returns to Java, all of those “local references” are released

So in this case, we see that the references all work out at the end of the java call and we reclaim the storage

How to Leak a Lot of Memory (2)



- We often call from Native->Java
 - Some Native-Java calls return Java objects
 - Those refs are reclaimed at the end of the enclosing Native call

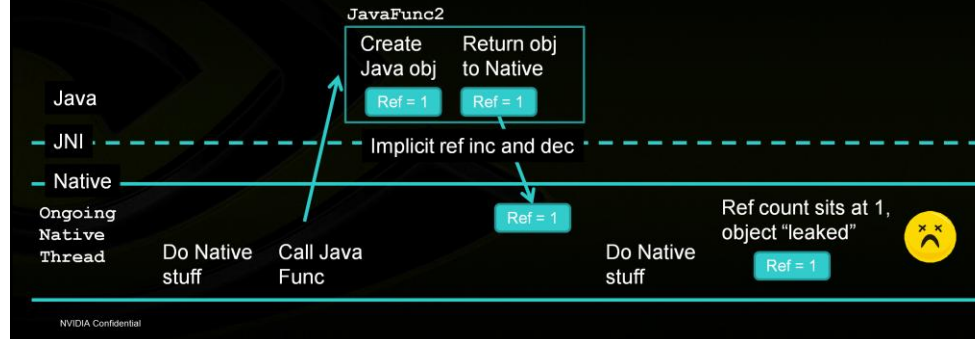


In our case, we also have helper functions in Java that we call up to from native. In the case in question, **JavaFunc** calls down to **NativeFunc**, which then calls back up into **JavaFunc2**. **JavaFunc2** is a helper function that loads data for us. The helper function allocates a big block of data in Java and passes it back to native. The implicit reference is there again, and when the enclosing **NativeFunc** returns to Java, that implicit reference is deleted. The reference count is zero because there are no Java references, and the block of memory is reclaimed.

How to Leak a Lot of Memory (3)



- “Ah, forgotten, but not gone.” George S. Kaufman
- If the Native-Java call is made in an ongoing Native thread, there is no “return to Java” that clears the local refs!



But we moved the code calling up from Native to Java into a native thread.

And that's when the leak was created

Since the thread was created in native code, it NEVER returned to Java

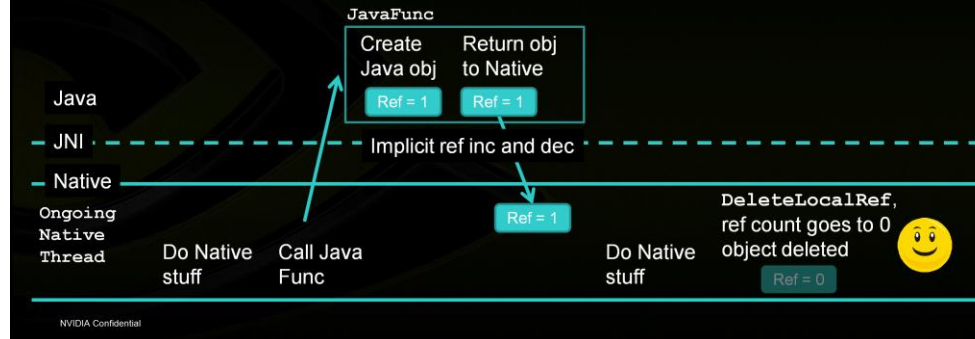
When we returned from the particular native function that called up into Java, we stayed in native

So the objects never get deleted!

How to (NOT) Leak a Lot of Memory



- Manually manage the implicit reference on the returned object
- Use a manual `DeleteLocalRef` to avoid leaks!
 - Good to do at the end of the native sub-function that uses the queried object



In these cases, you must explicitly release the object references when you are done with them in native code using `DeleteLocalRef`

We added those calls, and the leak went away.

Native code and Android



- **Android + JNI does NOT mean you just throw together code with any set of Linux headers and toolchains**
- **Developers want to know that their code will “just work” across as many Android versions and hardware devices as possible**
- **Android needs to innovate moving forward while continuing to run existing content seamlessly**
- **Android’s NDK makes this easy**

NVIDIA Confidential

The JNI provided with Android is NOT carte blanche to mix and match Linux kernel headers and random favored toolchains. Developers want confidence that the app they are building today will run on the widest range of Android devices and will continue working just as well with upcoming OS releases. Android ensures this with the NDK...

The Android NDK



- **Android's Native Development Kit**
- **Designed to allow applications to mix native and Java code while ensuring wide device compatibility**
- **Provides a set of toolchains, supported compiler/linker options, and a list of "stable APIs"**
- **Currently, not all major functionalities are available from the NDK**
 - These can be done in the app's Java class directly
 - Or exposed to native from Java via JNI

NVIDIA Confidential

Google provides the Android NDK, or native development kit. This is a set of toolchains, makefiles, and headers and libs that expose a set of "stable" APIs. The concept of "stable" here is not runtime stability, as much as API and ABI stability across Android versions and hardware. This does mean that not all Android or Linux functionalities are available via the NDK. So what did and didn't make the current cut?

NDK – What's There (Currently)



- **POSIX stuff**

- PThreads and related objects
- File I/O
- Sockets
- Memory management
- Math
- Basic C++ support



- **Multimedia**

- OpenGL ES 1.x
- OpenGL ES 2.0



- **Error / message logging**

NVIDIA Confidential

What's in the NDK? Well, a lot of pivotal functionalities you need to implement a game engine in native code: POSIX threads and sockets, math, 3D rendering, basic C++

NDK – What's NOT There (Currently)



- C++ Exceptions and RTTI, most of STL
- Multimedia
 - OpenMAX IL or any other video or audio support 
 - EGL 
- User input devices
- Android UI integration/functionality 

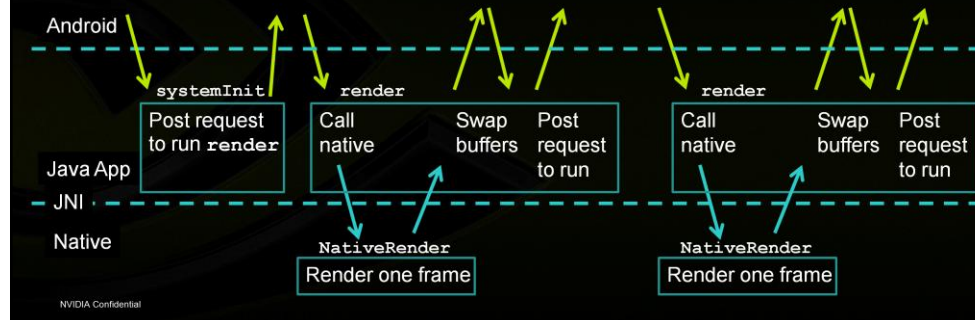
NVIDIA Confidential

What's missing? Well, to be honest, some pretty important things you need to COMPLETE that native game engine. Currently, you cannot use RTTI or exception handling in C++. There are no NDK audio or video APIs. And no user input devices or Android UI functionalities are available in native right now. But as you'll see, NVIDIA's samples assist with some of this.

First-shot: How we used the NDK for 3D



- We posted a threadable function that would call native to render one frame and then re-post itself to be run again “soon”
- After each frame, we returned from native to Java app to Android

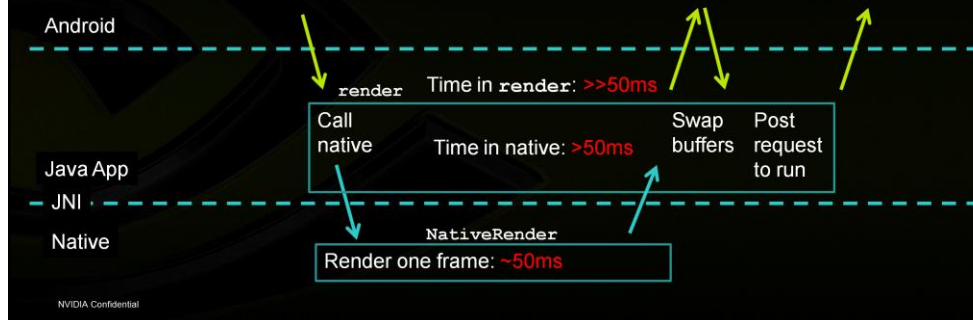


So how did our original samples render 3D? Quite simply. The diagram here is pretty complex, but the basic idea is simple. In Java, we posted a request to call our Java frame rendering function on the main app thread. In that function, we then called down to Native to render the 3D and manage the animation. Then we returned to Java to swap buffers. Before we returned from the Java render function, if we weren't exiting or paused, we posted another request to run the rendering function. And so on...

Rendering and Responsiveness



- As rendering took more time (complex data), issues appeared
- Longer rendering times meant that “render” was taking longer



That worked fine for HelloWorld, but as we began working in the RealWorld, the longer frame times meant we spent more time per frame in Native. That meant our Java render function was taking more time on the main app thread. Which lead to...

ANRs and Threads



- **During an engine port, rendering in a JNI thread caused issues**
 - Overall system UI slowed, input queues backed up!
 - Could even emulate this in our smaller demos
- **Spending “too much time” in app code in Java OR in native on the main app thread can make Android less responsive**
- **At the limit, Android declares the “app not responding”**
 - An “ANR” dialog is shown to the user
- **This can be solved by moving the main loop into its own thread**

NVIDIA Confidential

An interesting set of behaviors – our app was getting less responsive, and the input queues were backing up. Turns out that spending too much time doing Java or Native work in that app thread makes a lot of things less responsive! In fact, at the limit, if we put long sleeps in the render function, we got ANR's. ANRs, or Application Not Responding errors are thrown by Java when the app spends too much time in a function on the main UI thread. It shows the user a warning and lets them choose to KILL the app. Not good. The solution is to move the main rendering loop into its own thread. There are a few ways to do this.

Render Thread: The Java Way



- **GLSurfaceView and GLSurfaceView.Render solve this issue**
- **Runs the rendering in its own thread**
- **Can completely solve the render-based ANR issues**
- **No need for native code**
- **We chose to move the main loop to native code at the same time**
- **This lead us to another way to avoid the rendering ANRs**
 - **The nv_event sample library...**

NVIDIA Confidential

The Android class GLSurfaceView includes the member Renderer, which makes it possible to specify a rendering function that will be run in its own thread. This avoids the issue we saw with posting rendering to the main UI thread. In our case, when we started our samples pack, this API did not allow us to use OpenGL ES 2.0, so it was not a solution for us. But that's historical. We could now do this in Java, but we have chosen to move it all into native and provide an event queue to make porting easier. We created the nv_event sample library.

nv_event



- **NVIDIA DevTech helper library that eases the porting of native event-loop apps into the NDK**
- **Provides a “main”-like entrypoint for apps**
 - Allows for more common in-app render looping
 - Avoids doing any significant native app work in JNI threads
- **Remaps Android’s callback-based events to a message queue**
- **App just provides its native entrypoint (NVEventMain)**

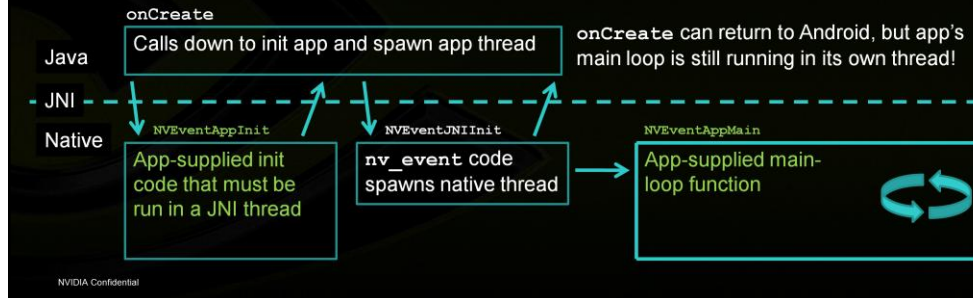
NVIDIA Confidential

Our nv_event helper library provides a main-like entrypoint for native apps and adds event-queue querying in native code to allow for easy porting or creation of native event/render-loop applications. It provides Java code to remap Android event callbacks into queued events.

Main Loop in Native Code



- Avoids ANRs by launching a native thread in a wrapper function
 - Wrapper function does initialization and calls the app's "main"
 - App can run as an event/render main-loop without blocking JNI threads
- Requires that events be available as a polled/waited queue...



NV_event avoids ANRs by launching a native thread on creation and running the app's main function in that thread. The Java `onCreate` function returns, and the app is happily looping on rendering and events in its own thread. Of course, we still need those events.

Event Queues



- `nv_event` includes event handling code
- Turns the various Android input callbacks into a stream of events
- Native app code can block or block-with-timeout on the queue
 - Similar to other platforms like Win32 and OpenKODE
- Java code does event culling/coalescing

NVIDIA Confidential

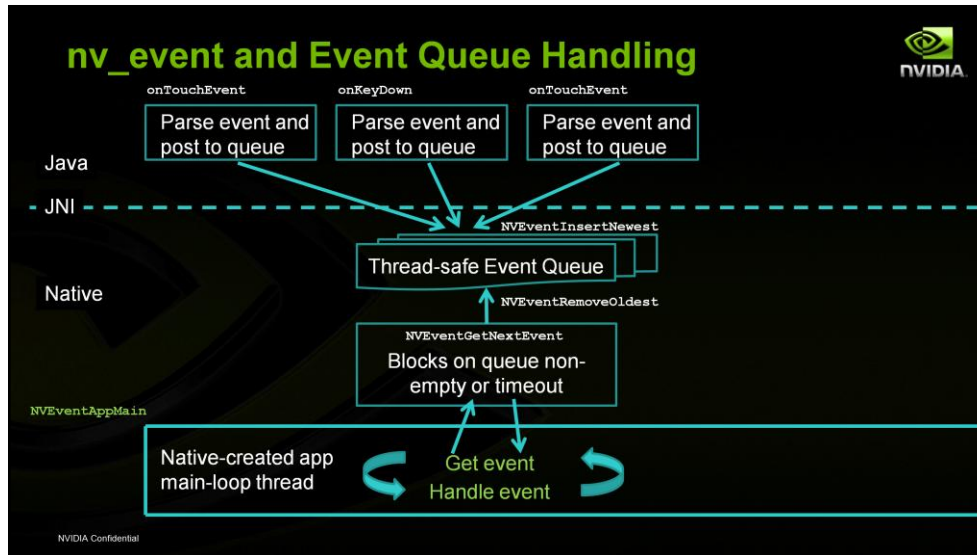
`nv_event` includes Java and native code to fill a thread-safe event queue with input and system events from the Android input callbacks

Java code does event culling/coalescing

Some touchscreens generate streams of unchanging events...

Avoids redundant JNI calls, app-level handling

That queue is available to the native app code



Here's what it looks like. Up top, the Android event callbacks are implemented by the helper lib and push the events into the queue. The app sits in its own thread, looping on the event queue, blocking or timing out as it chooses. Which lets the native app work a lot more like Win32 or OpenKODE apps, and eases porting while solving the ANR issue.

Nv_event: Looking Back



- **Nv_event is a powerful framework, especially for large native engine codebases that are multithreaded**
- **But it can feel complex, and may not be needed for some games**
- **And calling JNI from native threads adds to complexity**
- **If you can rework your main loop into Java, you might want to consider doing so**
- **But nv_event can still serve as an example of how many of these apparently “Java only” operations can be done in Native**

NVIDIA Confidential

Nv_event is a powerful framework, especially for large native engine codebases that are already multithreaded to their core. But it can feel complex, and may not be needed for some games – calling JNI from native threads adds to complexity, and complexity adds some risk.

If you can rework your main loop into Java, you might want to consider doing so

But nv_event can still serve as an example of how many of these apparently “Java only” operations can be done in Native. It can also serve as a great “quick port” layer to get your app up on Android quickly. From there, you can migrate it into a more Java-centric architecture as time permits to better handle pause, resume, etc.

Keep in mind that nv_event is a work in progress. For example, we do not currently generate events or callbacks into the native code for pause – we’re looking into options for doing this smoothly.

Content... Lots of Content...



- High-end games have a lot of media and data
- Android can store data in the application's installer pack (.APK)
- Or you can install it to removable storage (SD card)
- Each option has pros/cons

NVIDIA Confidential

High-end games tend to contain a lot of data for their worlds, sounds and the like. Android offers two ways to deploy this data. It can be packed into the application's installer pack (called the APK), or it can be stored on removable storage, like an SD card. There are pros and cons to each.

In-APK Data



- **Pros:**

- Data packed into APK automatically during build
- Installs automatically with the app
- Easy to load the files into the app (from Java)
- Automatic compression/expansion; aligned for fast streaming

- **Cons**

- Large files make each recompile/repack/install take a long time
- Pre-Froyo, devices often had small partitions to hold installed APKs
 - Froyo allows apps to specify that they can be installed to SD card
- Read-only

NVIDIA Confidential

If the data is packed into the APK, then the packaging and deployment is automatic along with the app. There's nice code in Java to load files from the pack, and it even gets compressed and expanded for you. The data is aligned during packing for fast streaming. But, the code to load APK-based data in native is tricky and involves offsetting tricks. Also, packing in large data bloats the APK, often by tens of megs. This, in turn makes debug iteration slow because rebuilding and redeploying take longer. The partition sizes on older devices for app install were designed for some code and a little data. Not a full-sized 3D game. Prior to Froyo, these partitions were the only allowed install locations. Froyo allows apps to include a flag in their manifest that specifies the app can be installed to local storage partitions, or to removable storage, making this less of an issue. And the data is read-only.

External Data



- **Pros:**
 - Files can be much larger (limited only by user's SD card)
 - Custom compression, etc and native loading can make loading faster
 - Does not bloat the size of the installed APK
 - Makes debug iteration much faster
 - Data can be updated independently of app
- **Cons:**
 - App-managed (but Froyo makes it less so)
 - App must figure out how to install the data
 - App could download the content from a website on first run
 - Must carefully query the device's external storage path

NVIDIA Confidential

Saving your data to external storage allows your data to be limited only by the user's SD card. You can do your own compression, and you can use native file calls to load efficiently. And debug iteration is fast because the APK only has your code in it. Of course, you have to manage it yourself, specifically you must install the data to the card yourself. As of Froyo, however, if you install your data to a specific location on the storage card as per the Android spec, Android will automatically delete it when the app is uninstalled. But you will still need to install the data from somewhere. You could have the app download the data from the game's servers on the first run. That takes more work. But the sheer size of the datasets in the apps we saw meant external storage was the way to go for our developers.

Tip: Know your Permissions!



- Every Android app has a manifest XML file
- Declares a lot of basic naming, Java class mappings, etc
- But also declares the desired permissions for the app
- Not having the right permissions can cause confusing bugs:
 - No INTERNET permission? Socket calls fail...
 - No WRITE_EXTERNAL_STORAGE? Writing the SD card will fail...
- These affect Java AND NDK code
- Get to know these

<http://developer.android.com/reference/android/Manifest.permission.html>

NVIDIA Confidential

Android apps all include a manifest XML file that describes things like the Java code to be used, the startup information, and the icon and strings. But it also includes a very important list of requested app permissions. The list is long, but not having the right permissions can lead to API failures that can waste time. For example, not having the INTERNET permission declared will cause your native POSIX socket functions to fail. Given how hard sockets are to debug already, that's not a problem you need. So check out the Manifest permission docs on the Android website for a list.

Graphics Tuning



- **Several focii:**
- **App-level feeding of the APIs**
- **Vertex/Geometry tuning**
- **Pixel/Fragment tuning**
- **Tools**

NVIDIA Confidential

Next, we'd like to talk about tuning 3D content. We'll focus on app-level API calls, vertices and geometry, fragments and shading, and then discuss some performance tools.

OpenGL ES 2.0



- **Becoming widely supported on major smartphone OSes / platforms and other mobile platforms**
- **Current and next-generation mobile 3D hardware is generally built for ES 2.0**
- **Availability of powerful vertex and pixel shaders are an important upgrade:**
 - *Performance:* avoid per-vertex CPU work that was common when shoehorning modern content into ES 1.x
 - *Differentiation:* huge range of effects now possible
 - *Power:* Use the right core for the job; dedicated vertex units avoid lighting up CPU's FPU as much

NVIDIA Confidential

OpenGL ES 2 is becoming widely supported on major mobile platforms

Current and next-generation mobile 3D hardware is generally built for ES 2 support

The addition of programmable vertex and pixel shaders which is core to ES 2 has important ramifications on numerous user experience factors.

Performance and content differentiation of shaders versus fixed-function are well known

But there are also potential power improvements by offloading vertex-related work from a general CPU core to a vertex-focused shader core.

App Behavior Tuning



- **Within reason, minimize the number of GL ES calls**
 - Cull state changes based on high-level app knowledge
 - Driver has to handle anything in the spec. Your engine knows better!
 - Don't over-batch if you can cull, but do avoid small batches
- **Shaders can be "heavy state"**
 - Uniforms are shader state; when changing from one shader program to another, all of the uniforms from the new shader must be set up
 - Avoid shader thrashing; group rendering calls by shaders



NVIDIA Confidential

Overall, you want to spend as little time in the driver code as possible. So, the best calls for performance are the ones you don't make. Use your high-level engine information to avoid redundant render state changes. Batch your geometry reasonably. Don't draw one to 10 tris at a time. Do not over-batch if it hurts the ability to cull invisible geometry, but be reasonable.

Shaders are particularly heavy state changes because all of the shader constants in a shader are swapped out with the shader. So every shader change causes a lot of constants to be shuffled around.

The diagram at the bottom shows two orderings for the same content. One changes shader on every draw call. The second sorts the geometry by shader and avoids changing shaders nearly as often. The latter can be a big performance win in some cases.

Vertex/Geometry Tuning



- **Good, compelling content tends to be large**
 - Memory bandwidth can be tight
- **Optimize all aspects of memory bandwidth usage**
 - Vertex formats
 - Vertex layout (interleaving)
 - Normals are particularly ripe for smaller formats like 8-bit signed
- **Maximize use of static VBOs/IBOs**
 - Use indexed primitives, sort for vertex caching
 - IBOs+VBOs to allow for maximal GPU parallelism

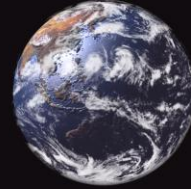
NVIDIA Confidential

Geometry can generate a lot of memory bandwidth, so while good datasets can and should be geometry-heavy, they should choose their vertex component formats well. Interleave vertices for good memory behavior. Select the smallest vertex component type you can get away with. For example, Tegra can handle byte components, which can be good for normals. Use indexed primitives and `glDrawElements` so that you can take advantage of vertex re-use. And use VBOs and IBOs for as much of your geometry as possible to maximize the ability of the driver to get out of the way of the GPU.

Fragment/Pixel Tuning



- Screen densities on newer mobile devices are high
- Lots more pixels to fill now (854x480, 1024x600, etc)
- Content optimization is likely to focus on the fragment pipeline
- Common fill limitations:
 - Memory bandwidth (texture reads, alpha blending, buffer writes)
 - GPU cycles (long shaders)



NVIDIA Confidential

I've joked that devices are making up for having larger screens by having denser pixels, but it really is true. The number of pixels to be filled continues to rise, and thus optimization most frequently focuses on the pixel or fragment pipeline. The most common fill limitations are memory bandwidth from textures and alpha blending, and GPU cycles from the longer and longer shaders that we see with high-end content.

Fragment Shaders



- **Do work in the (likely under-utilized) vertex unit if possible**
 - Longer vertex shaders, shorter fragment shaders...
- **How much shader precision do you need?**
 - Use lowp and midp where possible
 - Important for varyings and locals
- **Fragment bottlenecks can be well upstream of the shaders!**
 - Investigate your depth complexity
 - Investigate your culling
- **Alpha testing via `discard` is often a performance loss, not a gain**

NVIDIA Confidential

One way to optimize fragment shaders is to offload them! Be on the lookout for data that is linear or close to linear across a triangle. In those cases, consider moving computation of those from the fragment shader to the normally-underutilized vertex shader. Consider shader precision. If you do not need full floating-point precision (and you rarely do in a fragment shader) then declare a lower precision on the variable. Keep in mind that what appear to be fragment bottlenecks may not be the shaders themselves. Keep your depth complexity in mind and consider culling options. Finally, keep in mind that for various reasons, using the shader instruction `discard` like the classic fixed-function alpha testing is often a performance loss, not a gain in modern hardware.

Textures



- Texture format selection matters
- Compression is (still) king
- Use deep textures, but only where they're needed
- Use 1- and 2-component textures where possible
- Use mipmapping to improve performance AND visual quality
 - But beware trilinear and especially anisotropic filtering, which often have a performance penalty
- No standard compression+Alpha formats exist for GLES

RGBX32==2MB



DXT1==256KB



NVIDIA Confidential

Textures are the major component of data read bandwidth in 3D rendering. And with screen sizes getting larger, using smaller textures is less and less of an option. Instead, use compressed texture formats to reduce the “depth” dimension of your texture sizes. Also, consider using 1 and 2 component luma and luma alpha textures for grayscale and 2 vector textures. Mipmapping can be a big performance win if you do not over-use trilinear filtering and avoid anisotropic filtering. Note however, that there is currently no completely standard OpenGL ES compressed texture format with alpha. The GL ES standard compression format (ETC1) has no alpha support. So proprietary formats will almost certainly be required right now. This is a good reason to consider the post-install data downloads, as the device’s texture format support can be queried and a matching pack of nicely compressed texture data downloaded.

Depth and Scene Complexity



- **Consider rough depth sort or a depth pre-pass to optimize later color pass for expensive shaders**
 - Can reduce the expense in the fragment unit
- **Additionally, consider extensions like Occlusion Query to really allow for app-level culling of hidden items**
 - Can reduce the expense in all units

NVIDIA Confidential

As mentioned earlier, being fragment limited can be caused further up the pipeline. Consider a rough front-to-back sorting of your opaque geometry, or (if you have heavy fragment shaders), consider a depth-only pre-pass. Also, consider powerful extensions like Occlusion Query to avoid even vertex and CPU-level work for obscured objects.

EGL Config Confusions



- EGLConfigs define the pixel depth, aux buffer formats, API support, etc for surfaces and contexts
- Querying and selecting a config can be confusing:
`eglChooseConfig(disp, attribs, &config, 1, &count);`
- Don't be tempted to just grab first matching config
 - See the EGL spec – the sorting method required by the spec ended up being confusing to some developers
 - E.g. requesting 16bpp RGB can return 32bpp RGB *FIRST* even if an exact 16bpp config existed
 - Spec requires that the deeper config be returned first!
 - Other surprises in there, too
- Request a long array of matches, and sort in the app
- Note that GLSurfaceView already does a “closest match” sort

NVIDIA Confidential

When selecting an EGLConfiguration upon which to base their rendering, apps generally set up the list of desired color channel bits, depth bits, stencil bits and supported APIs.

Then, they call `eglChooseConfig` to return a match.

Applications often rely implicitly upon EGL's configuration sorting algorithm to do what they want.

They request only the single “best match” config and use it blindly.

The problem here is that the EGL spec's sorting method can return surprising results;

requesting a 565, 16 bit rendering config can return a 32-bit rendering config as the best match, even if 565 is supported.

The spec requires the deeper config to be returned first.

For best results, applications should request a longer list of configs that match and sort and cull them manually using their own sorting metric.

If you are using `GLSurfaceView`, it already does a form of “closest match” sorting. It may “just work” for you. But if you have specific needs, sorting configs yourself is still safest. The `GLSurfaceView` sort may have a different idea of “goodness” than your app.

Losing your EGL Context



- **You can lose your EGLContext while paused**
 - In fact, if you use GLSurfaceView, you WILL lose it, as GLSurfaceView attempts to be aggressive and consistent on this front
- **Be ready for this case!**
 - Look out for `eglSwapBuffers` or `eglMakeCurrent` returning `FALSE` and an error code of `EGL_CONTEXT_LOST`
 - Be ready to reload all textures, VBOs, etc following the event
 - This is best done by compartmentalizing/grouping your GLES content loading as much as possible so it can be re-run
- **This requires careful planning and good testing!**
 - But will really help you run well

NVIDIA Confidential

Be ready to lose your EGLContext. And in so losing your context, losing ALL of your OpenGL ES resources. This can be a painful rework of your code if you don't consider it early on. Split out your GLES resource loading as cleanly as possible so you can call it as needed.

Window Size and Composition



- **Android is a composited OS**
 - Your window is drawn to the screen by a GLES-based compositor
- **If you're really stuck on fill rate issues on a high-resolution screen...**
 - `SurfaceHolder.setFixedSize` may be your friend
 - Sets a constant, fixed size for your window
 - Window manager will automatically scale your backbuffer to the full screen size during composition
 - Filter-magnifies if needed
- **Not great, but can work well on a high-density screen**

NVIDIA Confidential

Since Android is already a composited OS, it has decoupled the size of a window with the size of its presentation on screen. So, if you find that you need to lower your 3D fill rate on a large-screen device, you can fix your window size to be small, and Android will scale up while compositing. There's no free lunch, though – your game will look blurrier than one that runs at the native resolution.

Performance Tuning Tools



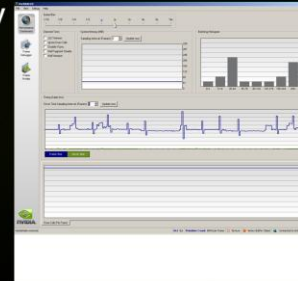
- Use performance tools early and often
- Technical artists can be excellent users of performance tools
- Know where your CPU and GPU cycles are going
- Prove your bottlenecks

NVIDIA Confidential

PerfHUD ES



- **Mobile-centric NV PerfHUD**
 - Works exclusively with Tegra-based development kits
- **Renders stats and graphs on a separate host PC**
 - Minimizes overhead on mobile device
 - Allows for more screen real estate for feedback
 - Most mobile dev is done with a host PC, anyway
 - Currently, host PC must be MS Windows-based
- **Publicly available on Android!**
- **Includes/Supports**
 - Stats graphs
 - Directed tests
 - 2x2 textures, ignore draw/all calls, etc
 - Frame profiling/debugging



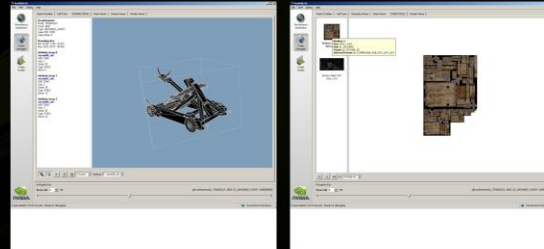
NVIDIA Confidential

NV Perf HUD ES is the mobile version of NVIDIA's popular perf hud performance tool exclusively for Tegra-based development kits. Unlike PC perf hud, which generally renders stats to the application's window, perf hud ES renders detailed performance graphs on a connected Windows-based host PC while the application runs live and interactively on the device. Using the host PC avoids trying to render the graphs on the smaller device screen and offloads the additional work. Perf HUD ES is now publicly available for Tegra Android on NVIDIA's developer website. In addition to the live graphs, it supports making changes to the rendering to diagnose performance issues and even has an object-by-object, call by call frame profiler and debugger

Handy PerfHUD ES Features (2)



- Call Trace / Frame Debugger Mode
- Full list of state calls in frame (redundancy checking)
- Frame “scrubbing”
- Partial-frame (frame-to-call) views including FBOs
 - Color buffer
 - Depth buffer



NVIDIA Confidential

The call trace mode in Perf HUD ES allows you to capture every GL ES call in an entire frame and “scrub” through the frame, seeing the contents of all buffers after each draw call, and even viewing the shaders, textures and geometry used to render each object.

Handy PerfHUD ES Features



- **Performance Dashboard Mode**
- **Ultra-fast top-level performance triage**
 - Hit the common, top-level issues quickly
 1. Ignore all calls (are we just app-limited?)
 2. Null fragment shader (are we shader-heavy?)
 3. 2x2 textures (are we memory-bound?)
 4. Disable primitive batches by histogram
- **Break-on-GL-error**
 - For those rare (ahem) cases where your code isn't checking each call

NVIDIA Confidential

Performance triage can be done very quickly in perf hud using the directed test options. These options tell the driver to null out all GL ES calls, use a NULL fragment shader, replace all textures with 2x2, or disable all small or large draw calls. Finally, the mode also allows for basic debugging by being able to halt rendering on the first GL error

Tegra Developers' Zone



<http://developer.nvidia.com/tegra>

- OS Support packs
 - Android
 - Linux
 - Windows CE
- SDK's, demos, apps
- Docs
- Development Tools
- Public support forums/community
- Access to the Tegra board store

NVIDIA Confidential



In closing, I'll invite everyone to visit NVIDIA's Tegra developer zone. It includes access to the developer kit hardware store, OS images, tools, samples and documentation, as well as developer forums.

Conclusion



- **Android is a powerful gaming platform, especially on high-performance systems like Tegra**
- **Leverage the Android NDK for high-performance and lower-effort conversion of high-end content**
- **Use OpenGL ES 2.0 to get the most out of modern devices**
- **Visit NVIDIA's Tegra developers zone for devkits, tools, and sample code**
- **Keep pushing the content envelope!**

NVIDIA Confidential