Good afternoon – my name is Lars Bishop, and I'm an engineer in NVIDIA's Tegra Developer Technologies group. Today, I'd like to present some topics in Android game development that seem to come up time and again with developers moving to the Android platform and working to get the most out of their games on Tegra superphones and tablets. This talk covers a lot of ground in small sections, and it's intended to have a mix of content for beginning and advanced Android app developers.

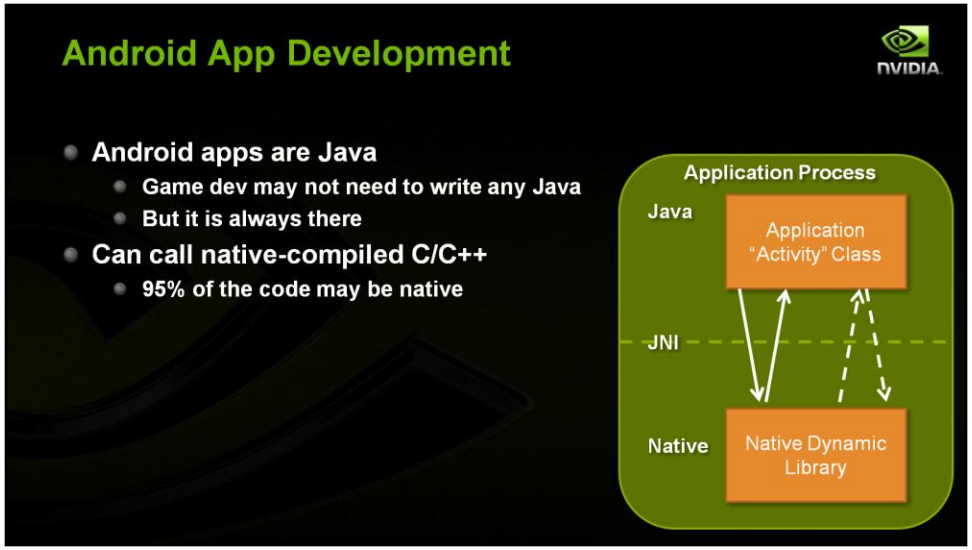
I'll start by discussing some of the top development challenges on Android. We'll define and discuss so-called "Pure" native game development on Android, along with the benefits of writing _some_ Java into your app. We'll cover programming tips for performance and power, as well as device interaction. I'll end my technical sections with a discussion of developing mobile games for big-screen, console-style play. Then, we'll show what these tips can lead to, with a multi-player, big-screen demo of Madfinger Games' triple A shooter, Shadowgun!

For developers just starting to plan out an Android game development or porting project, the questions I see are almost universal. Setting up the full development tools environment is an early concern. Once it is possible to build and deploy AN app, developers ask about how they should architect THEIR app. Most focus on the question of how much Java they need to write versus reusing their existing C++.

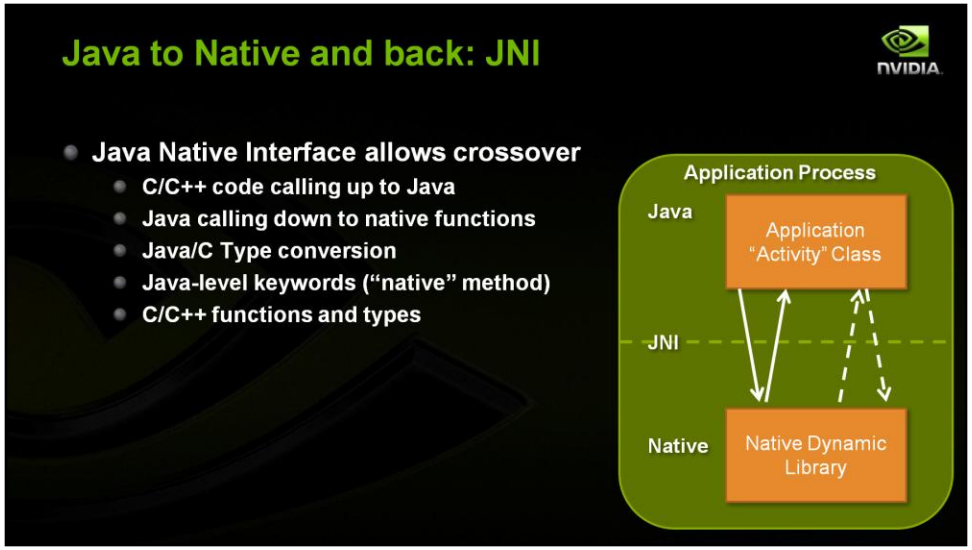
When developers get deep into their project, the questions change somewhat. Accessing Java-only Android features from native code is a concern. Maximizing multi-core performance comes up during tuning, as well as tuning across varying screen sizes. Input, especially sensors and game controllers are a common source of issues. But the number one question area by far tends to come late in the game during multi-handset QA, and it's the handling of application lifecycle. Reacting well to incoming calls, notifications, suspend/resume, and device orientation changes can be a challenge. These questions are best asked at the _beginning_ of development, so I like to mention them even to new developers.

The first step is setting up an Android development machine with all of the right components. NVIDIA makes this easy on all of the Android development platforms via the Tegra Android Development Pack, or TAD-P! This is a simple installer that downloads, installs and configures the numerous Google components for development, native code compilation and IDE integration, as well as the IDE itself, Eclipse. It includes some NVIDIA Android samples so that the developer has working apps to build and run right out of the box. It even installs some of NVIDIA's tools. However, the pack can be used to develop for any Android platform. It's a great way to get started, and it's available today at the NVIDIA developer website.

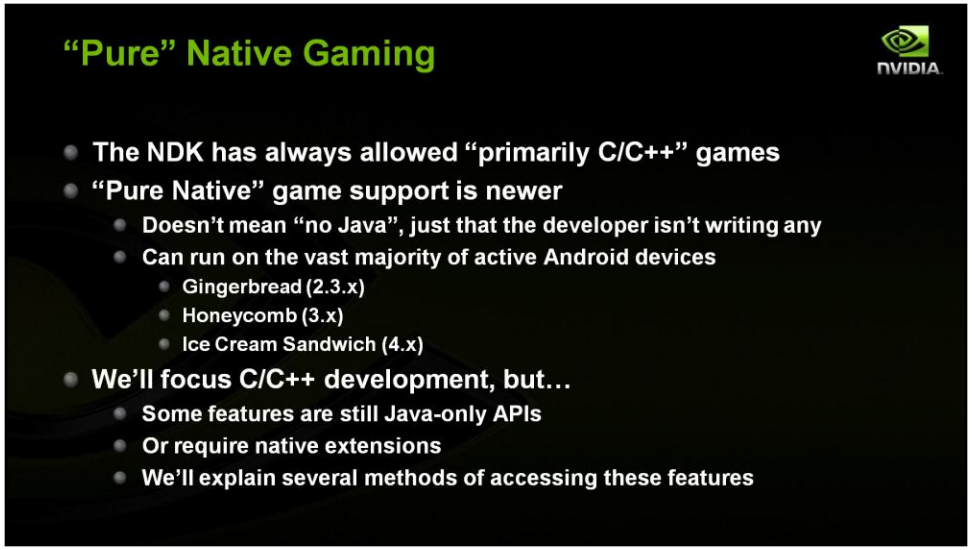
New developers often want guidance on basic Android app development. Google's documentation is really great, but some game developers find they want more specific game and native programming content. NVIDIA assists with samples, some of which are installed with TADP. However, we have JUST posted a Developers Preview of a NEW Samples Pack, specifically targeting the exact topics we'll be discussing in this session. TODAY, these new samples are a standalone download that you unzip ON TOP OF TADP, but soon they'll be a PART of TADP. NVIDIA's developer site also has a wealth of supporting docs and slides on the topics we'll cover today.



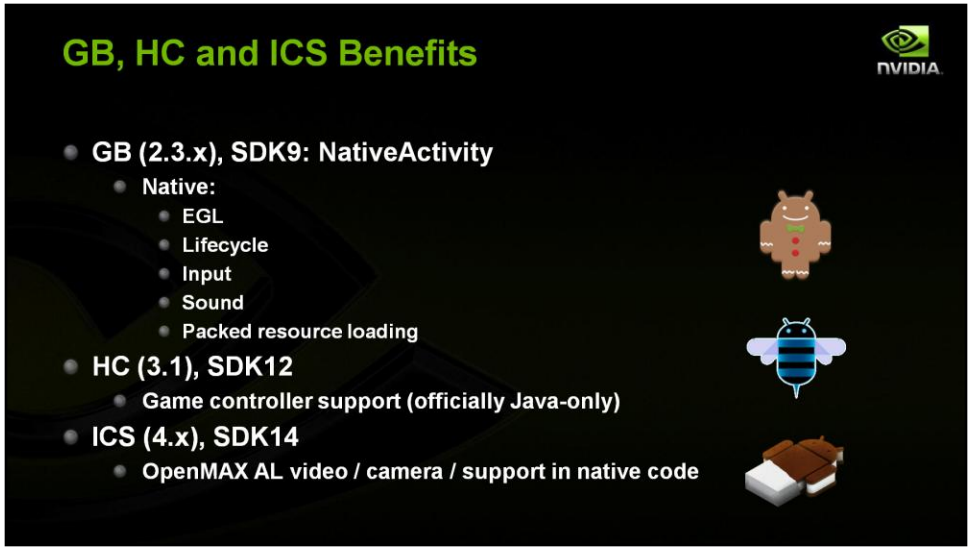
A quick sketch of the core of an Android application. All Android market apps are Java-based. A game developer may or may not write any Java code for their app, but the top level application is still a Java class. The most common class that represents an application is Android's Activity class. Core Android apps have numerous Activities, but most games use just one to represent their entire app in all of its modes. The Activity class is Java, but in practice, large 3D Android games will probably be 95% native code, and spend almost all of their cycles in native code. But the lifecycle of an app from launch, to being the focused app to being hidden and destroyed happens at the Java level. So we can't completely ignore it.



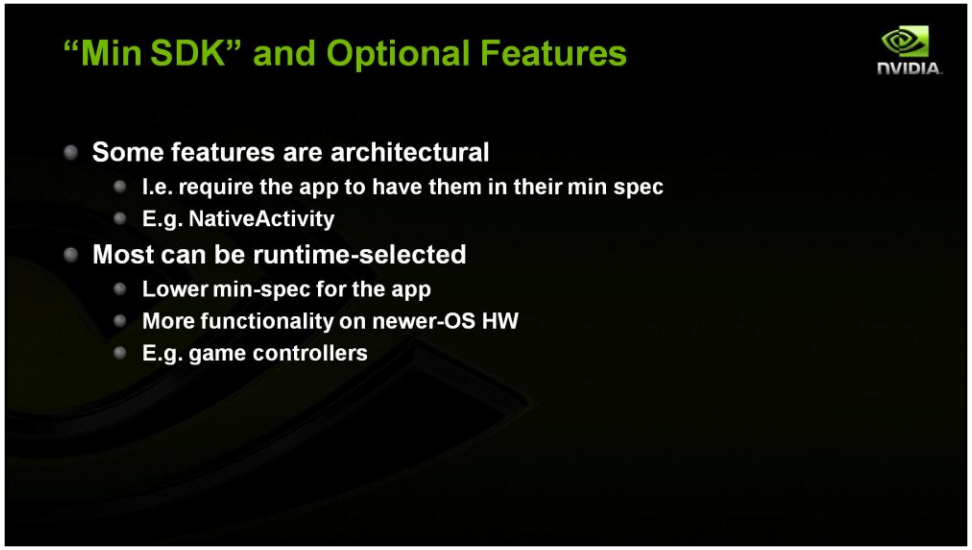
The method of jumping between Java and native code is Java's JNI or Java Native Interface. There are Java keywords and a set of native headers and libs that allow native code to be called from Java and allow native code to access Java objects and call their methods. We'll talk about these in more detail later.



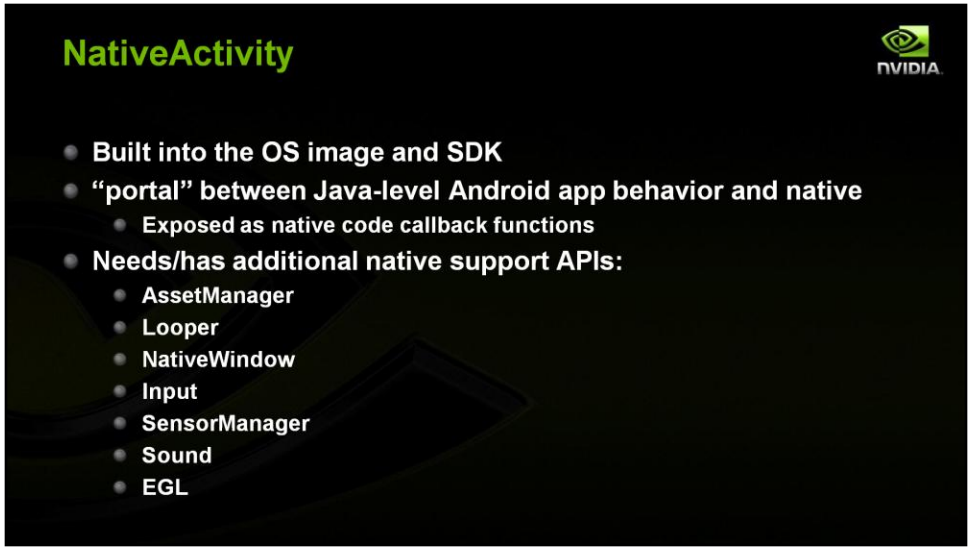
Developers often ask whether they can avoid writing any Java code at all. While the Android Native Development Kit or N D K has allowed Android games to be written MAINLY in C for about two years, originally, developers using native still had to write Java that called their C code. Developing "Pure" Native android games without writing *any* Java is newer. Today, Android games with no developer-written Java code are possible and will run on the vast majority of active Android devices. Specifically, any devices running Gingerbread or newer. We'll focus on native C/C++ development today, but some functionality is still only available from the Java APIs. We'll highlight those as we go along.



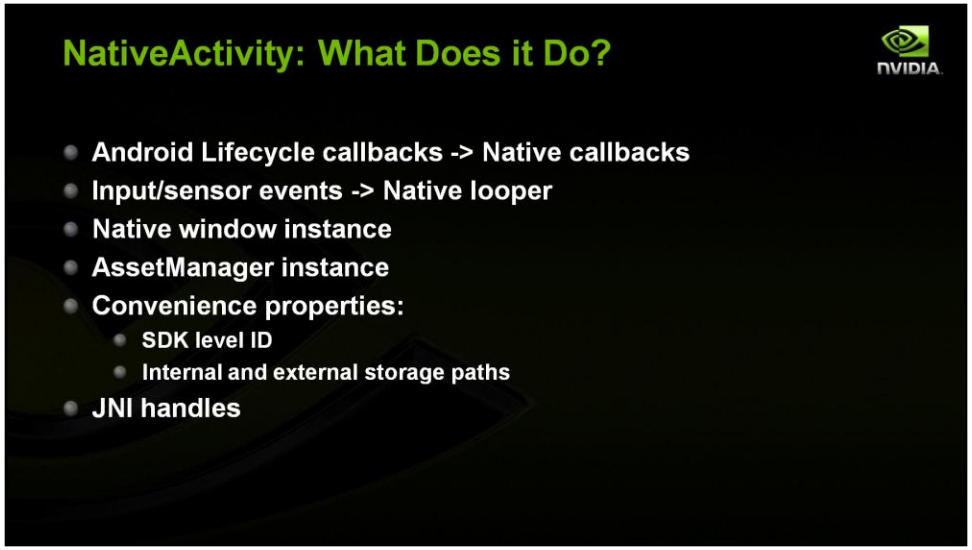
Each new version of Android has added new native or gaming features and a new SDK level to target those features. Gingerbread added the true mother lode: The NativeActivity game framework and related native APIs for core game behavior. Honeycomb added game controller support, although officially only from Java code. Ice Cream Sandwich and its APIs added video support in native code. Each of these new sets of features are only available on devices running the related OS or newer. Using a feature could mean setting your min spec higher, or it could mean optionally using the feature at runtime. It depends on the particular feature.



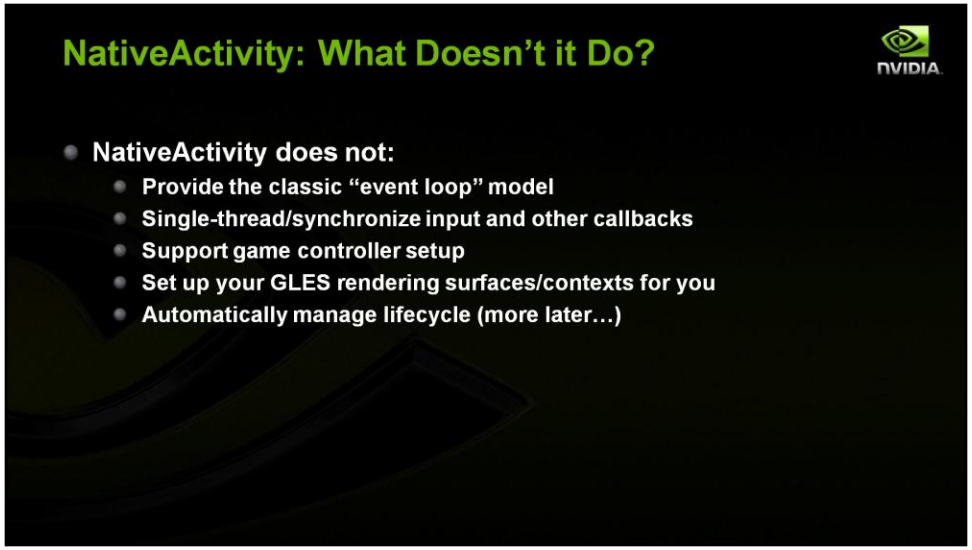
The simplest way to use a given feature is to set the "Min SDK" in your manifest to a level that includes the desired features. Of course, this makes your game appear in the Android market only on devices running the newer OSes. Some features like NativeActivity and thus pure native gaming are architectural and require the min spec method. So, to write a pure native game, Gingerbread is absolute min spec. But other features are optional and can be runtime-selected. If you support conditional features where possible, you may be able to keep a lower min-spec while supporting more features on the growing set of newer-OS hardware. Game controllers fall into this latter optional category.



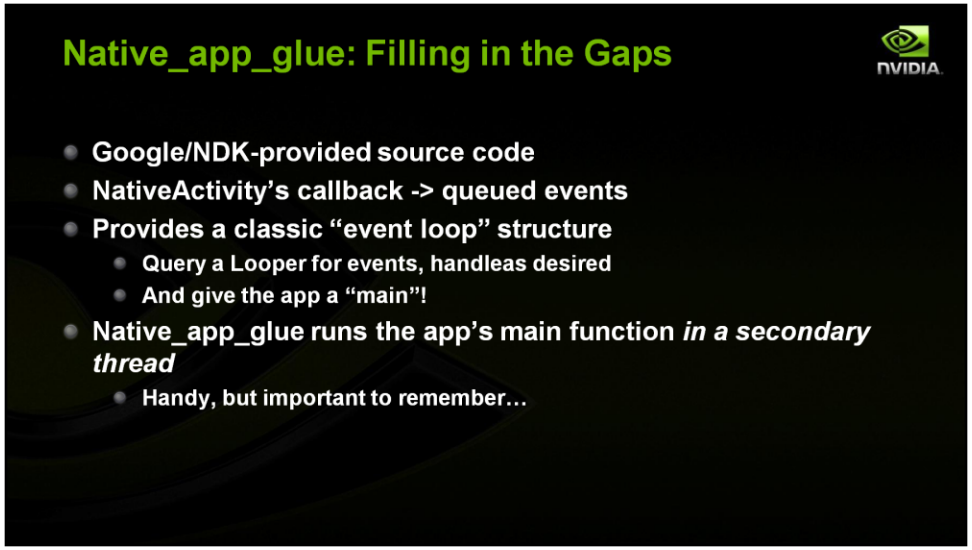
So what is NativeActivity, how did it enable pure native gaming? It's a system class, built into the OS image itself and exposed by the Gingerbread and newer SDK and OS. It serves as a pre-built "portal" of important Android callbacks and input messages down to native code. Your native code registers callback functions for these events, rather than implementing your own pass-through layer in Java and JNI. But pure native games need more than just input and lifecycle. So NativeActivity is bolstered by other NDK APIs. These APIs add data loading, input, audio and rendering setup.



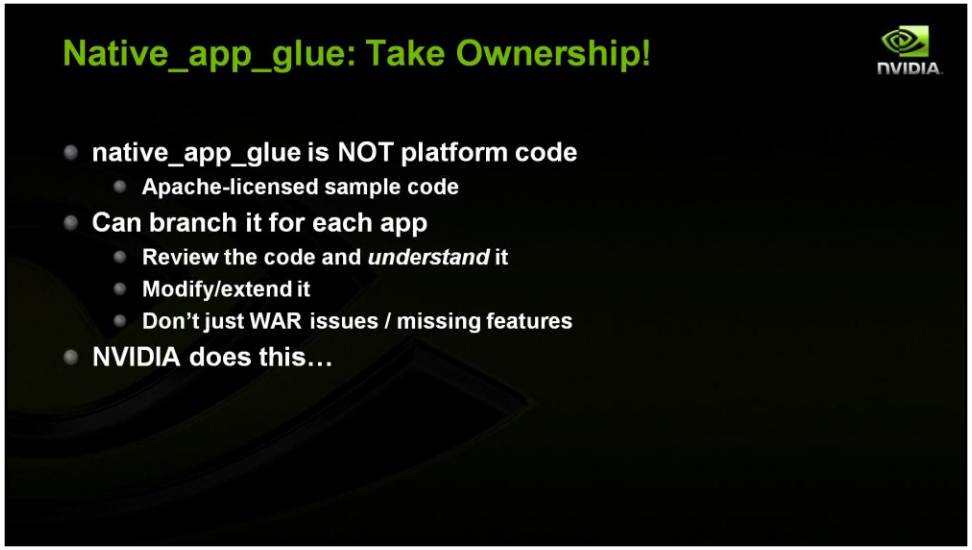
What does NativeActivity do for you? Well, it generates native callbacks for Java level Android lifecycle events. It fills a native “looper” queue with input events. It provides the native code with the window for CPU or GPU rendering. It provides an AssetManager for loading game data that has been packed into the app’s installable APK pack. Finally, for apps that still need to call up to Java, it exposes the required JNI handles



It is important to understand what NativeActivity doesn't do! It does not provide a single "main" function entry point. Nor does it provide a persistent native game thread or single "event queue" for all interaction and lifecycle. It is a mix of events and callbacks, calling native code when it sees fit. If you spend too long in any one native callback, Android will assume you've hung and will tell the user your Application is Not Responding. And most importantly, it does NOT automatically manage application lifecycle. More on that later…

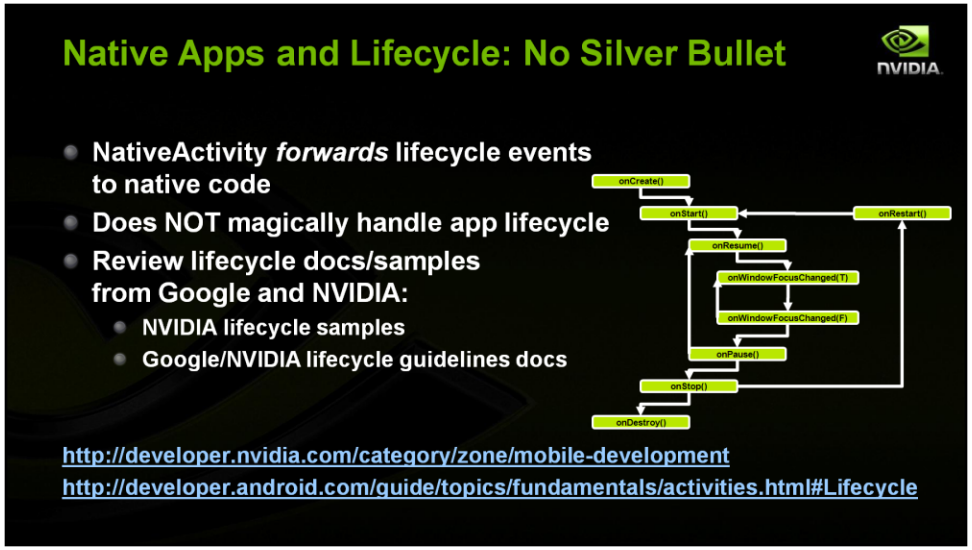


Google saw several of these lackings in nativeActivity as important to game developers and provided Native_app_glue to Fill in the Gaps. Native_app_glue is example source code in the NDK for a "wrapper" that turns NativeActivity's callback functions into queued events. It runs the app's main function in its own persistent thread. This transforms NativeActivity's on-demand callback flow into the classic "event loop and queue" structure. Apps do not provide callbacks for each event; they simply query a Looper for events and handle them as desired. The good news is that this thread decoupling allows your main loop to block on events without Android declaring your application unresponsive. But it also means that the app's code is not running in a Java thread, so you have to take care if you use JNI to call up to Java.

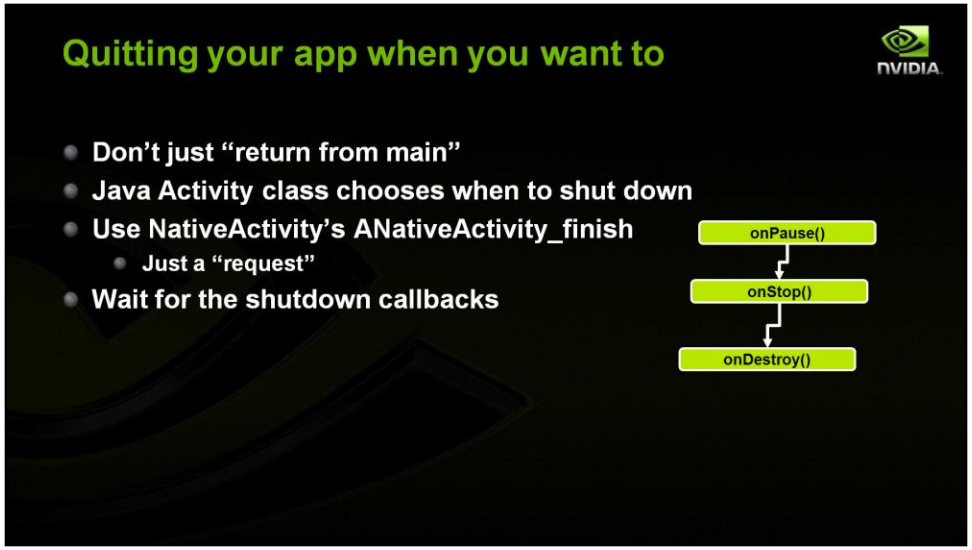


One recommendation for those using native_app_glue is to Take Ownership of it. Unlike NativeActivity, native_app_glue is NOT platform code. It is compiled individually into each app. Google provides it as Apache-licensed sample code. Apps can (and should) branch it into their own source base. You should review the code and try to *understand* it. That way, you can modify and extend it, rather than working around anything missing. In fact, we do this ourselves…

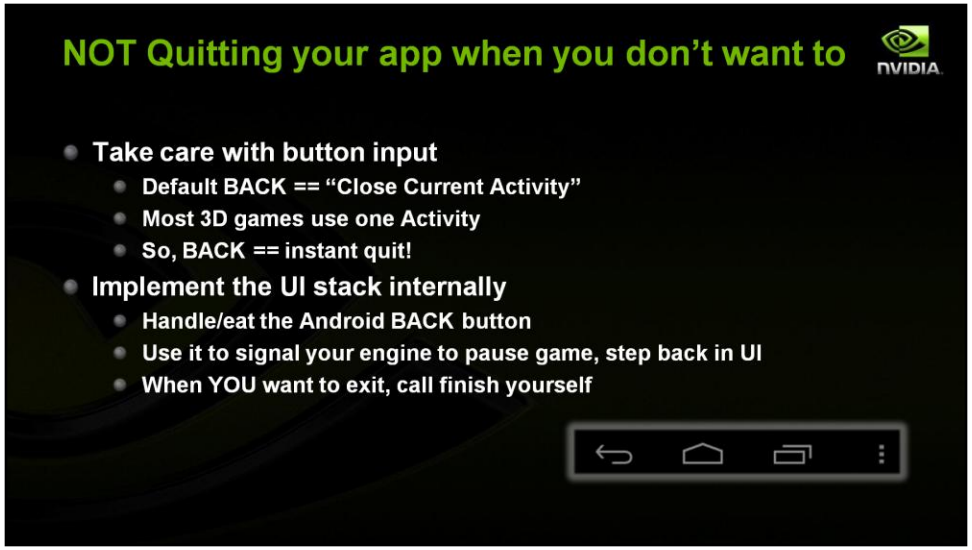
The Tegra Android Samples Pack includes a modified and extended native_app_glue, which we build into a helper library. It adds some missing events for NativeActivity callbacks and some state-tracking convenience functions, as well as registering the app's mainloop thread with Java. This made it easier for us to create clean, lifecycle-friendly apps. It also made it easier for us to add JNI calls up to Java. And we add more features to our version of native app glue as we find need. You should likely consider the same.



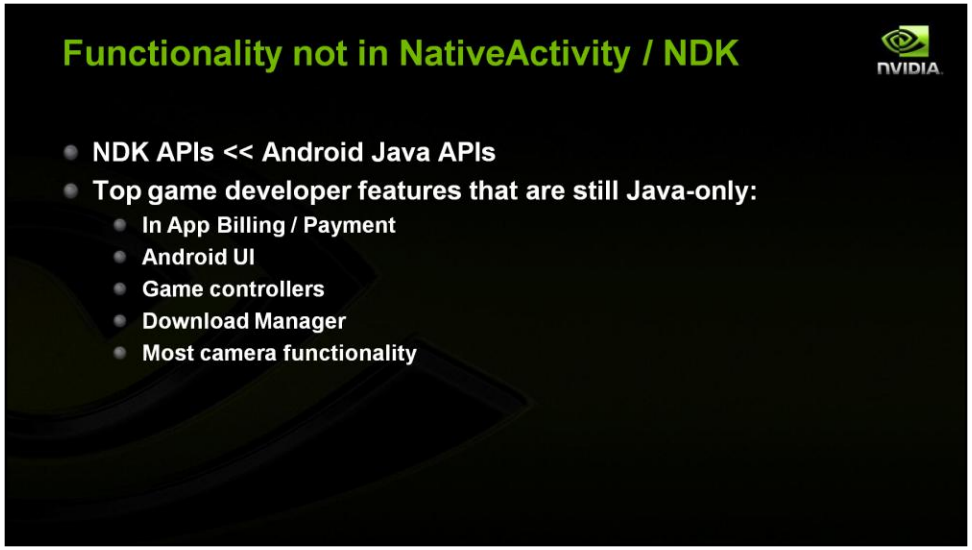
NativeActivity and native_app_glue make the Java app lifecycle callbacks AVAILABLE to native code. They do NOT magically handle app lifecycle – they are merely conduits for the same information you used to have to get in java. Pure native apps have the same lifecycle responsibilities as Java apps with respect to starting and stopping game rendering, sound, auto-pausing gameplay, and the like. So, developers must review the lifecycle docs and samples from Google and NVIDIA and test frequently to have the best chance of pleasing their users. NVIDIA's new samples pack includes specific pure native app lifecycle demos that show this. We also have detailed documentation on handling app lifecycle on multiple Android versions.



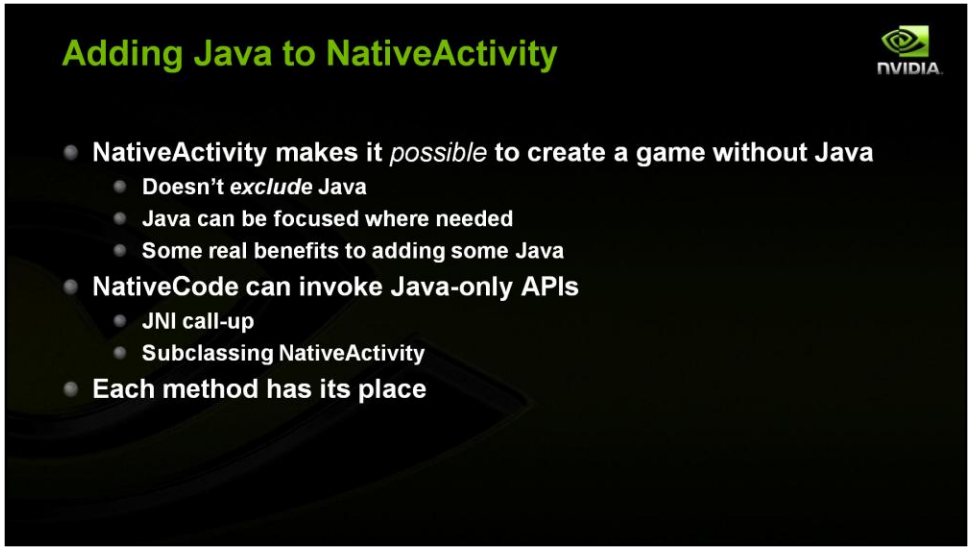
One surprising thing for many new Android developers is that Quitting your App on your Schedule is not trivial. Native code (in NativeActivity or native_app_glue) cannot simply "return from main" and expect the app to shut down. Java still rules the roost. When the Java activity chooses to shut down or "finish", it sends callbacks to that effect to the native code. NativeActivity allows native code to explicitly request that the app finish using Native Activity finish. But it is still a request. The native code must then handle the lifecycle callbacks indicating the app is exiting before shutting down.



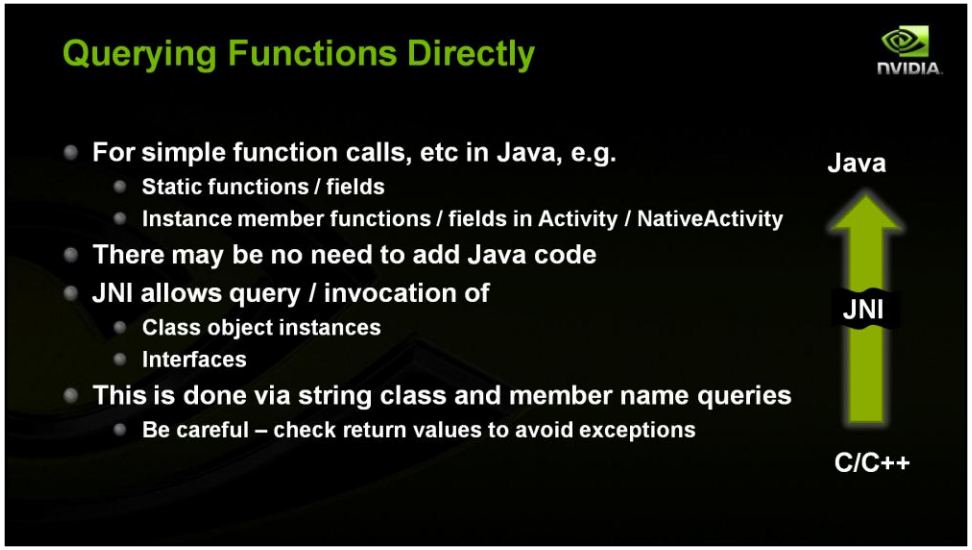
On the other end, the app needs to take care with input events. If an Android BACK button event is not handled and explicitly consumed by the app's native code, the Java code will default to closing the current activity. Since 3D games do not generally use multiple Android Activities to represent their UI "stacks", they tend to have one Activity for the entire app. So the default Android BACK button behavior would exit the app with no warning or confirmation! Games should add their own handler for the BACK button to pause the game as needed, step backwards in their menu stack, and so on.



There are still plenty of Android APIs that are not accessible directly from NDK-exposed native interfaces even today. Some of the top game developer features that are still Java-only include: In App Billing and Payment, Android UI, such as dialogs, game controllers and the download manager. We'll cover several of these specifically, along with some general techniques.

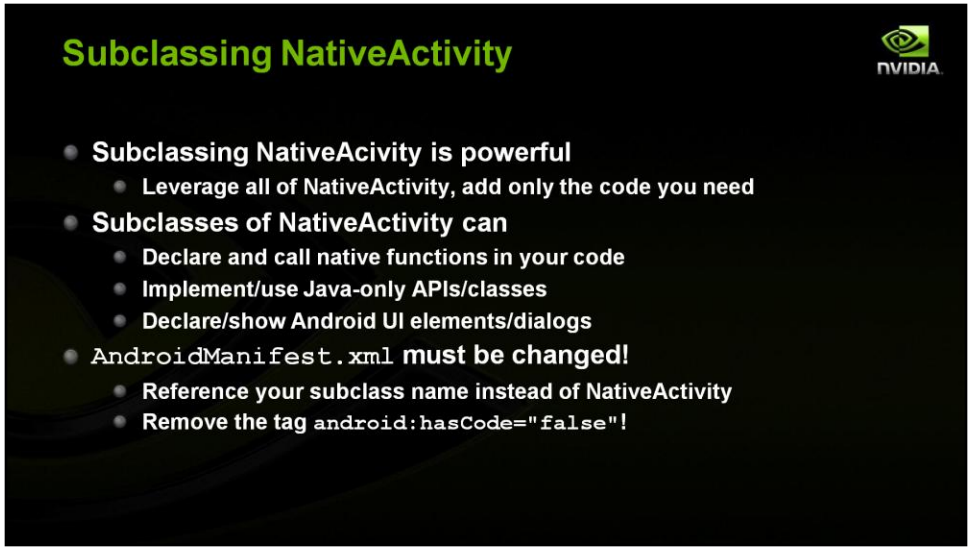


NativeActivity makes it possible to create a game without writing any Java code, but it doesn't rule out adding Java code to your app or calling Java from your app. Instead, it allows a developer to focus the Java on the more Android-specific features. Native code can "call up" to Java methods via JNI, or NativeActivity's Java class can be derived and extended. The choice between the two methods depends on what you need to do.

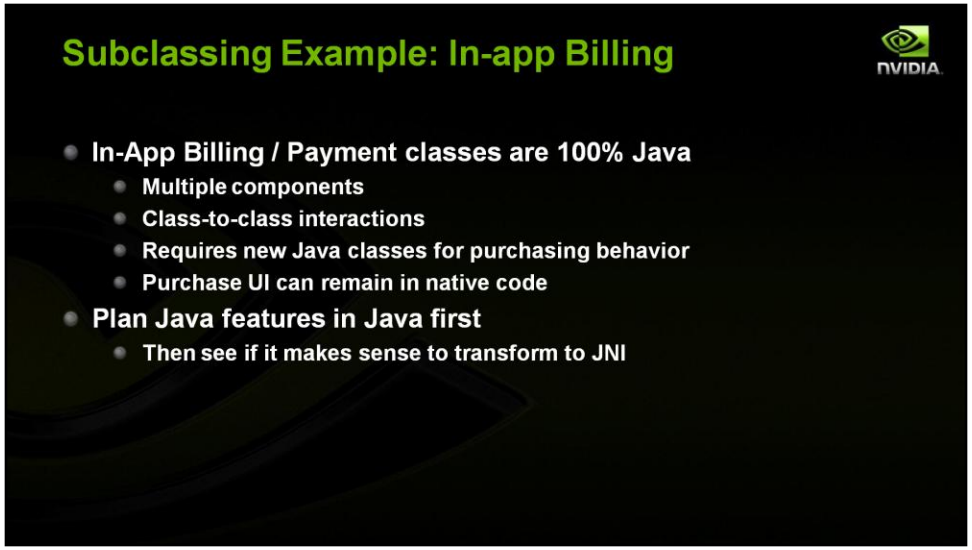


For simple Java-related code, such as reading member data in an Android Java class or calling one or two Java functions, be they static or instance, there may be no need to write any explicit Java source code. JNI allows native code to query handles to class instances and even to named member data and methods. Those handles can be used to read the member data or invoke the methods. However, be careful to catch and clear errors at each stage to avoid Java exceptions, which can kill the app.

Here's an example of getting data from a Java class via JNI. First, we query the interfaces for the Android-standard class OS dot Build. Then, we get the handle to the static field DEVICE , which is a Java string. We retrieve the static object's field value and convert it from a Java string to a C string, and print it. This is a static data member, so we had no need of any class instance to reference it. But JNI does allow you to query class instances and call their non-static members, too. Please note this code is illustrative only – in a real app, the return value and exception status should be checked after each call. Invoking a JNI function if a previous exception is pending can kill the app.



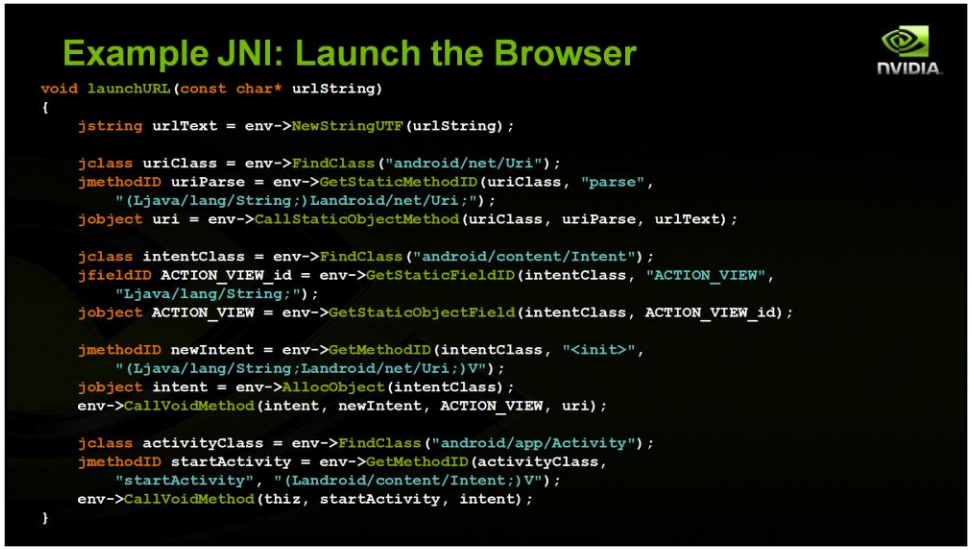
Even more advanced functionality can be enabled by subclassing the app from NativeAcivity. The subclass can be very simple, adding a function or two. But you can leverage all the existing functionality of NativeActivity and add only the code you need. Subclasses of NativeActivity can call your native code via JNI. But you can also use new Java to call any Android functionality you want. You can even show and handle Android dialogs if you need to. The sky's the limit here, but one tip is for those just starting: Don't forget to change your app's manifest file to point to your subclass instead of the base NativeActivity. And, remove the false has Code tag or else Android will completely ignore your newly-written Java. This trips up a lot of first-time users.



In-App Billing and Payment APIs are heavily Java-based and involve creating derived Java classes to handle response messages from the market. This means explicit Java is the only choice for in-app Billing. However, if you HAVE a choice whether to use pure JNI or a mix of Java and some JNI, my guideline is to plan out the feature in Java FIRST; the analyze. Replacing lines of Java with lines of JNI can be a 10x expansion of source code. Pure JNI starts to feel like building a ship in a bottle. When that happens, its time to just subclass NativeActivity and write that longer piece of code in Java. Then you can make ONE JNI call up from native to invoke it. Much cleaner.

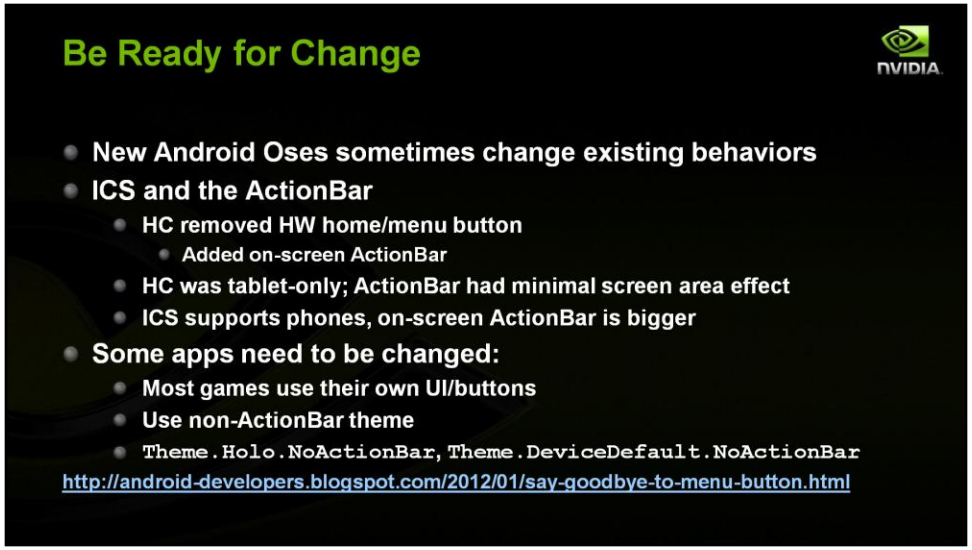
As an example, here's the code in Java to launch a browser in Android. Simple – you create a URL from the string, you create an intent that "intends" to launch the browser, and you launch it. You could easily call this Java function from native using three lines of JNI code.

Example JNI: Launch the Browser



```
void launchURL(const char* urlString)
{
    jstring urlText = env->NewStringUTF(urlString);

    jclass uriClass = env->FindClass("android/net/Uri");
    jmethodID uriParse = env->GetStaticMethodID(uriClass, "parse",
        "(Ljava/lang/String;)Landroid/net/Uri;");
    jobject uri = env->CallStaticObjectMethod(uriClass, uriParse, urlText);

    jclass intentClass = env->FindClass("android/content/Intent");
    jfieldID ACTION_VIEW_id = env->GetStaticFieldID(intentClass, "ACTION_VIEW",
        "Ljava/lang/String;");
    jobject ACTION_VIEW = env->GetStaticObjectField(intentClass, ACTION_VIEW_id);

    jmethodID newIntent = env->GetMethodID(intentClass, "<init>",
        "(Ljava/lang/String;Landroid/net/Uri;)V");
    jobject intent = env->AllocObject(intentClass);
    env->CallVoidMethod(intent, newIntent, ACTION_VIEW, uri);

    jclass activityClass = env->FindClass("android/app/Activity");
    jmethodID startActivity = env->GetMethodID(activityClass,
        "startActivity", "(Landroid/content/Intent;)V");
    env->CallVoidMethod(thiz, startActivity, intent);
}
```

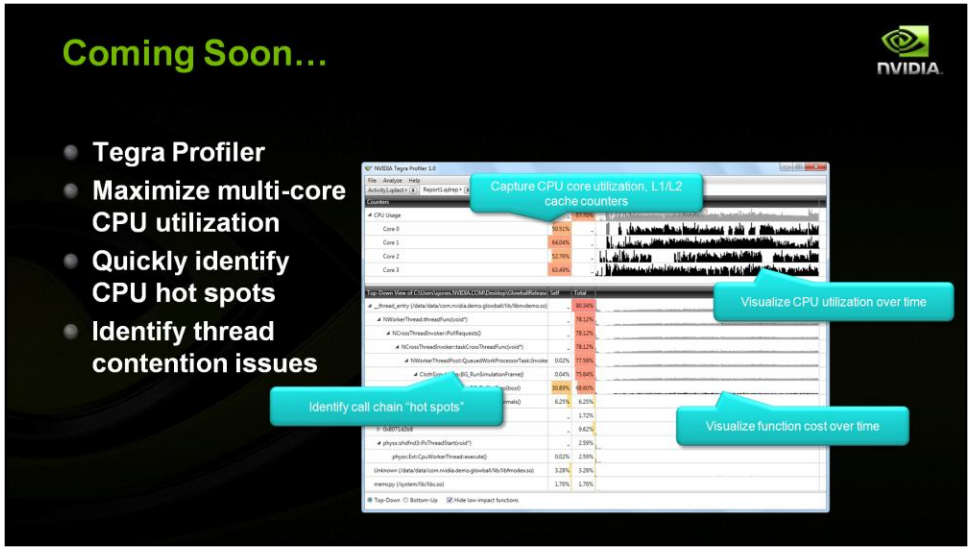
And here's what the same operation looks like doing it ALL in native using JNI. Note that I cheated here in a way that makes the JNI look _better_ than it might by skipping error detection and using C++ instead of C. So consider this before you decide to skip Java entirely.



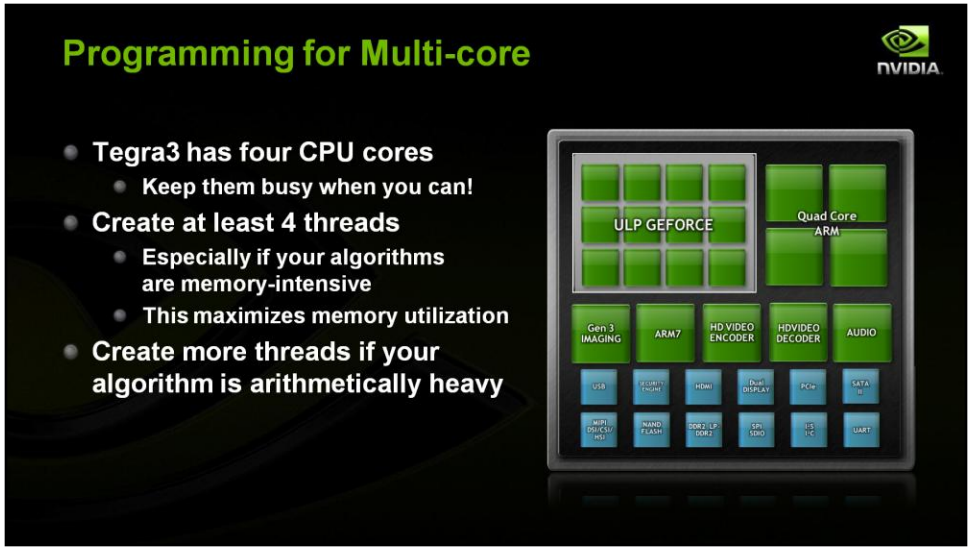
Android does sometimes change existing behaviors. Keep an eye on the Android developer blogs and SDK site. An important example of this is ICS and the ActionBar. Honeycomb removed physical buttons for home and back and replaced them with an on-screen ActionBar. Because Honeycomb was tablet-only, the new ActionBar had minimal screen area impact. Then, ICS added phone support. With ICS phones, the on-screen ActionBar is much bigger in relative terms. So, some apps needed to be changed. Most games use their own UI system, so they don't need or want the ActionBar. The Android developer blog posted an article on how to easily change the app's UI theme to one with no ActionBar. Keep an eye on the official developer blogs and SDK site to avoid getting blindsided.

A quick mention of speed and power in game programming. This is not specifically an optimization presentation, but it bears mentioning, since it comes up in most of the game projects I see. Programming for performance is about both top speed and knowing how to maximize your use of device power, which is just as limited a resource as device performance. Obviously, optimizing a game involves optimizing the GPU and CPU performance. Both vary depending on the exact device. Apps should dynamically scale their rendering as needed. Multi-core and NEON CPUs mean optimizing for multiple threads and vectorization. Power, on the other hand is mainly about knowing how to avoid needless work or causing a system component to do needless work for you.

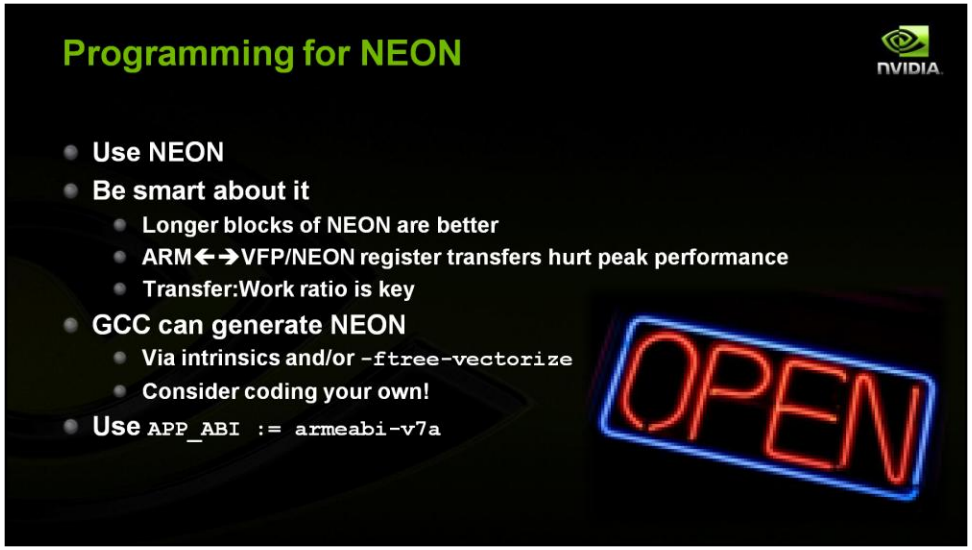
Don't bother with optimizations without profiling first! Few things are more demoralizing than a complex optimization that buys nothing. Fortunately, NVIDIA makes several tools available on its developer site for profiling both the CPU and GPU. NVIDIA PerfHUD ES is available for graphical GPU profiling and debugging. The Linux PERF and OProfile tools are available for system and app CPU profiling. All of these are available from the NVIDIA developer site.



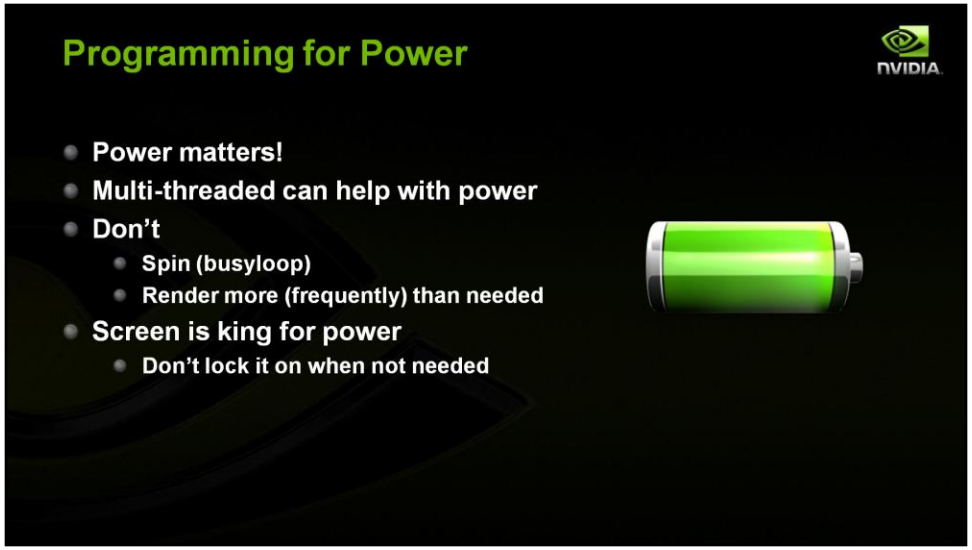
Also, to be released in the coming weeks is the graphical Tegra Profiler tool, an NVIDIA tool for hierarchical, multi-core, multi-threaded visual profiling. It will really help with multi-CPU optimization on Tegra.



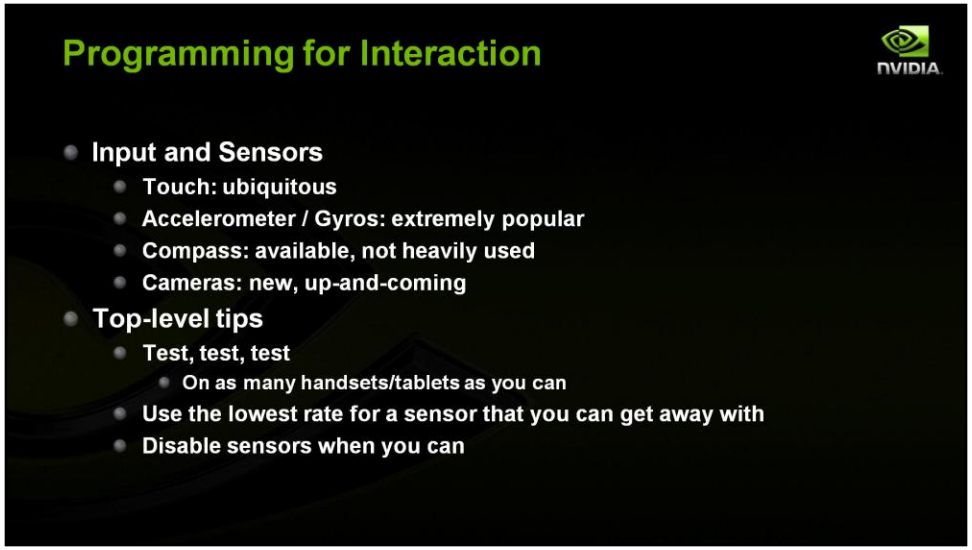
Tegra3 has four CPU cores, and you should look to keep them busy when you can. For most modern games, this is not a difficult thing to do, between physics, gameplay, rendering, and animation. If an algorithm is particularly memory-intensive and parallelizable, you should look to using at least four threads. Doing so will have the best chance of using the memory transaction slots of all 4 cores. If your algorithm is arithmetically heavy, you may want to create even more threads to maximize all units. Note that threads can be created in Java or native code, and even multiple Java threads can call into native code at once.



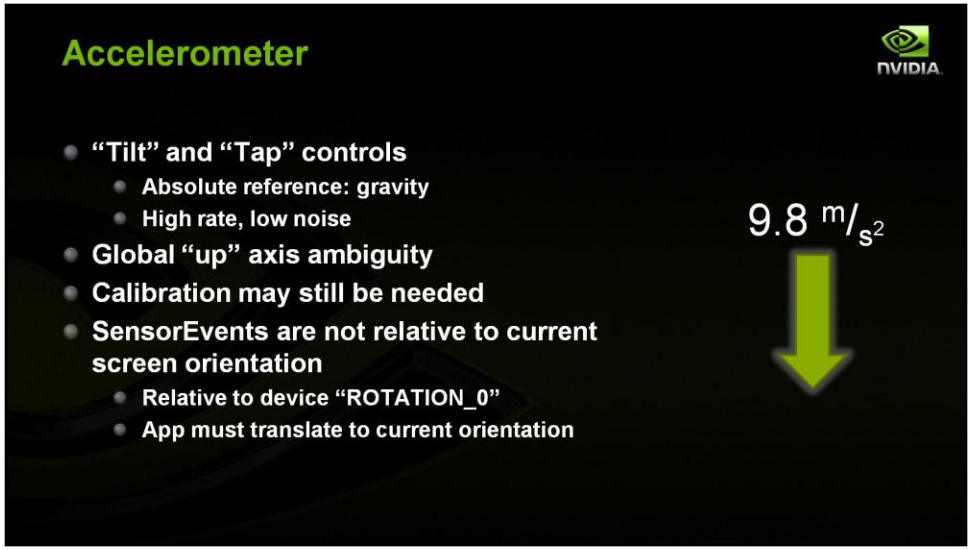
Use NEON, but do so in a smart way by converting longer sections of code to NEON. Avoid having a lot of small NEON functions requiring lots of ARM-to-NEON transitions. Transferring data back and forth between ARM registers and VFP/NEON registers can lower the benefits of NEON usage. Always consider the transfer-to-work ratio. Note that while GCC can generate NEON using intrinsics or the `vectorize option, if tight NEON code matters, you likely want to consider writing your own. Also, NEON or not,` don't forget to set the build system's or compiler settings `to target ARM V7A, not generic ARM, as this will generate tighter code.`



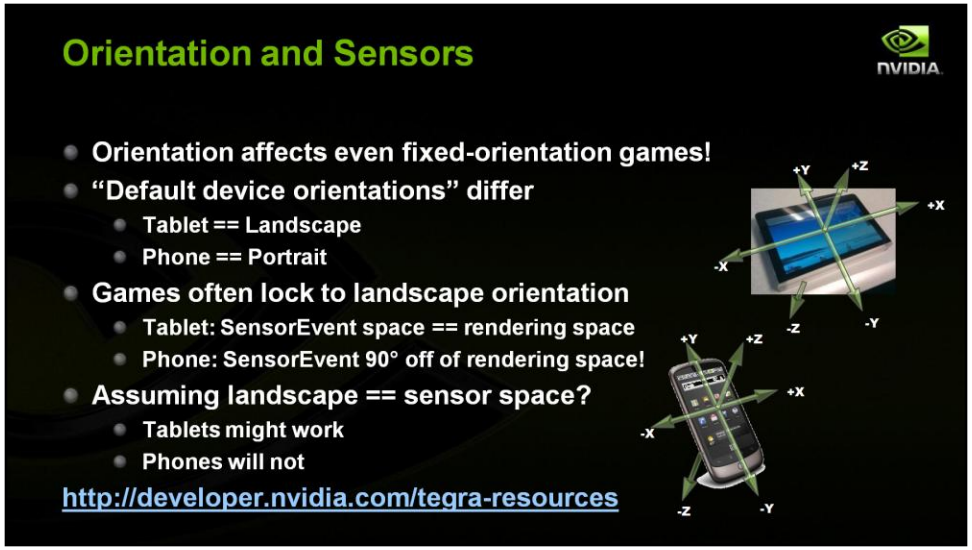
While we generally think of top speed in games, power matters, too! Multi-threaded programming can help reduce clock speeds and thus lower power. Don't render at full bore for no reason; for example, throttle rendering of pause screens or render them on-demand. But the biggest power hog can be wakelocks and keeping the screen bright. In turn-based games there probably should be no need to force the screen to stay on. In non-time-based adventure games, auto-pause when the user is clearly inactive for some time. In shooters, this can be harder owing to camping or sniping, but be smart about it.



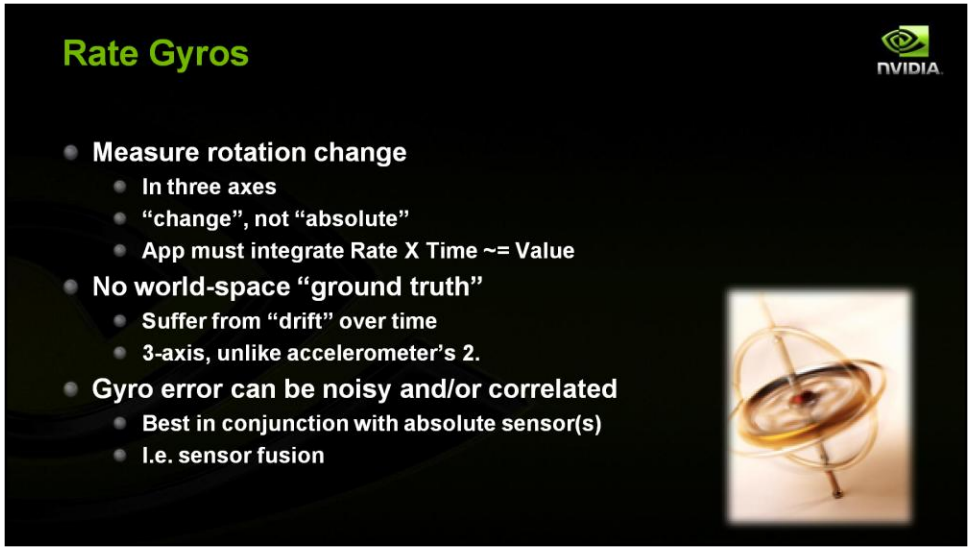
Interaction is pivotal, but also the source of last-minute bugs. Touch is certainly the ubiquitous, “every game has it” input device, but accelerometer and gyros are also used by a large fraction of action-based games. Compass is exported by many devices, but is one axis and noisy. It isn’t very popular in most games; we’ll not discuss it in detail here. And even cameras can be used as compelling input devices. As for top-level tips, test on as many devices as you can, as early as you can. Performance, noise and accuracy differ significantly. Also, many sensors let you set the rate of incoming data. Use the lowest sensor rate you can in your app. And turn off a sensor when your game is paused or loses focus. This avoids spamming your app’s event queue and can improve your power utilization



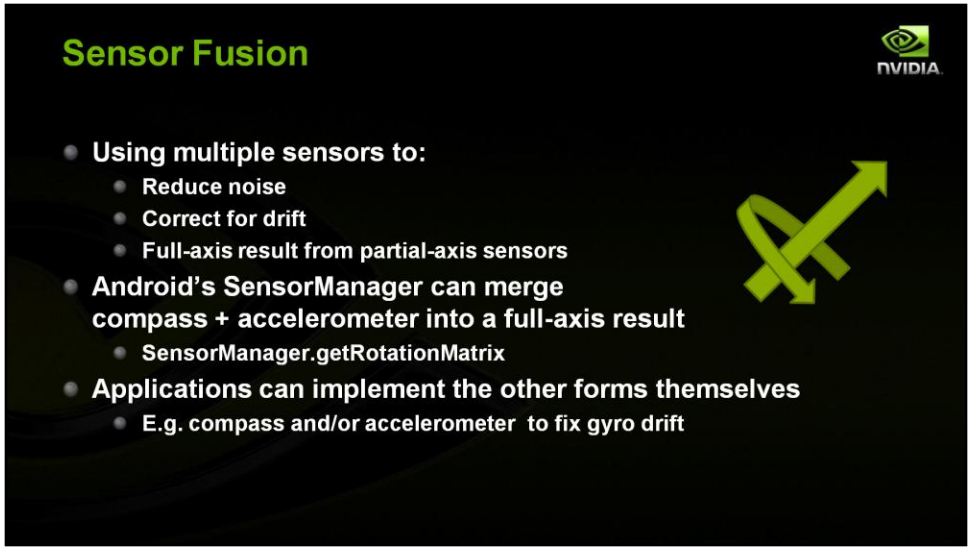
Accelerometers are great for "tilt" and "tap" controls. The tend to support high rates and have generally low noise. They are always relative to a great absolute reference: gravity. But they cannot differentiate rotation about the global "up" axis. So they do not provide three-axis orientation. While accelerometers have a global reference, don't rely on exactly one "g" at rest. We find devices report magnitudes that are three to seven percent off. Consider a calibration user option. One pivotal point about accelerometer and other SensorEvents is that they are not relative to current screen orientation. They are relative to the "default device orientation" or Android's ROTATION_0. The app must compute the transform to the current rendering orientation. What does this mean?



Some developers think that because they lock their game to landscape orientation, they don't need to worry about this sensor-space transformation. And if they only test on tablets, they might think they're right. However tablets tend to have landscape as their default device orientation. So a landscape-locked game happens to need no transform on most tablets. But, phones tend to have portrait as their default device orientation. So a landscape-locked game on a phone needs to rotate the sensor events 90 degrees. Apps that have assumed sensor equals device space have gotten nasty shocks when people run their games on phones and the controls are all wrong. NVIDIA includes docs and sample code on the developer site detailing this transformation. It's pretty easy, but you do need to do it.



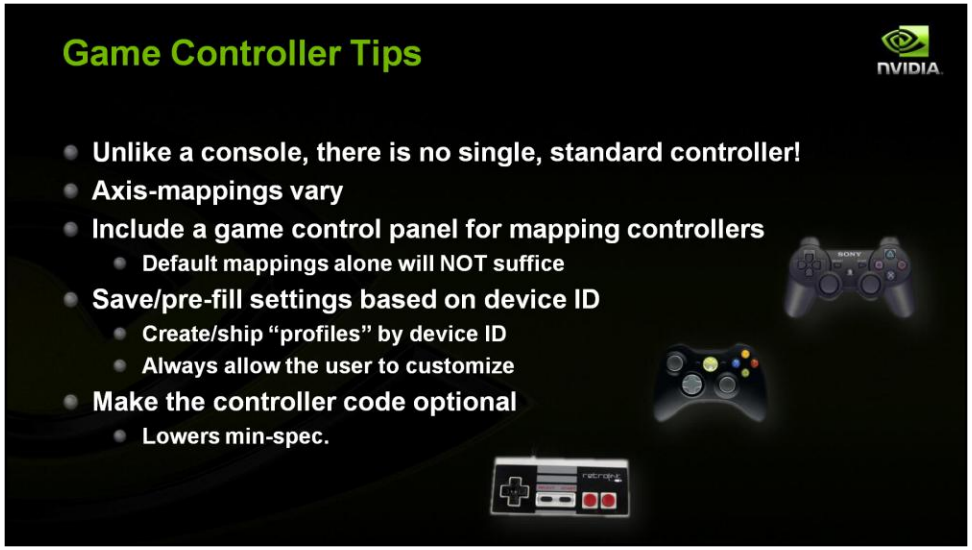
Rate Gyros measure rotation change in all three axes of orientation. The key here is rotation "change", not "absolute" rotation. The app must integrate the rate of change over time to estimate current orientation. Rate Gyros have no world-space "ground truth" in and of themselves. They are relative to the "initial position" when sampling started. So they can (and DO) suffer from "drift" when integrated. But they can track all axes of rotation, unlike accelerometers. We have found that the noise or error in rate gyros is often correlated in one direction; if you average the results, you may still drift in one direction or the other. It doesn't always cancel out. Gyros are best when used in conjunction with an absolute sensor in a process more generally called "sensor fusion"



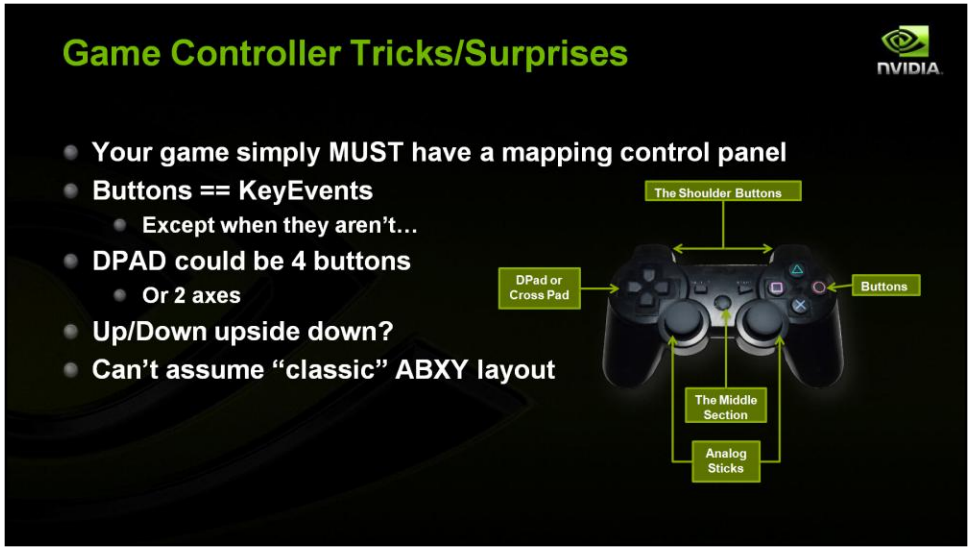
Sensor Fusion is the general term for using multiple sensors to; Reduce noise in the results, reduce drift in relative sensors like gyros and compile results from multiple partial-axis sensors into a full-axis result. Android's SensorManager has some functions that can assist with the most basic cases, like merging the one-axis compass and two-axis accelerometer into a full three-axis orientation. Applications may choose to implement the other forms themselves, with one of the most common being to counteract gyro drift with regular checks against compass or accelerometer

Console-style, “lean back” gaming is possible today on commercially-available Tegra-based tablets! Console-style gaming on a large-screen TV involves several Tegra features: Wired or wireless game controller support, HDMI-out-based HD rendering and 3D Vision support for stereo 3D rendering. Each of these involves some amount of application work, which we’ll cover in the next few slides.

Honeycomb 3.1 added support for game controllers in Java. The classes for some game input have existed since Gingerbread, but gamepad and joystick functions were added for Honeycomb. The APIs allow for discovery and value query of controllers, axes and buttons. Note that while the NDK supports motion events, the corresponding native functions to query analog game controller values are NOT officially exposed in the NDK headers or stub libs. They have to be queried from the Android system, and that is technically not safe across all platform versions. NVIDIA provides documentation and samples on how to best handle game controllers, but a few highlights…

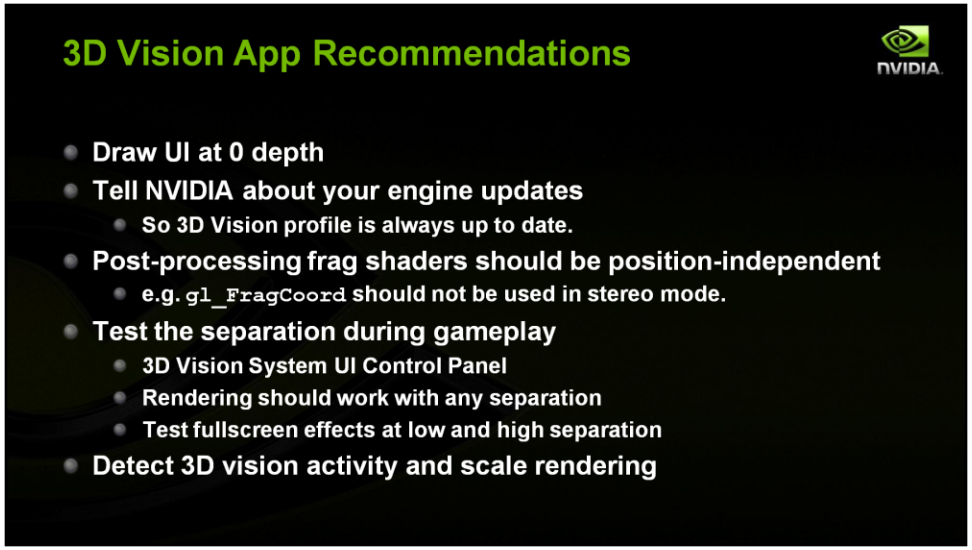


Users can and will connect just about any controller they might have. There is no single, standard controller. Despite NVIDIA's work on Tegra to ensure that most common controllers are normalized, controller axis mappings can be wildly different from device to device. It is imperative for user experience to include an in-game control panel for mapping axes and buttons to game actions. Default mappings alone will NOT be sufficient. To make things easier, create pre-built "profiles" based on common controller names and ship those with the game. Pre-fill those settings into the control panel for a detected controller. But always allow the user to customize. Make your game controller code optional at runtime, since without game controllers, your game might still be able to support Gingerbread as minspec.

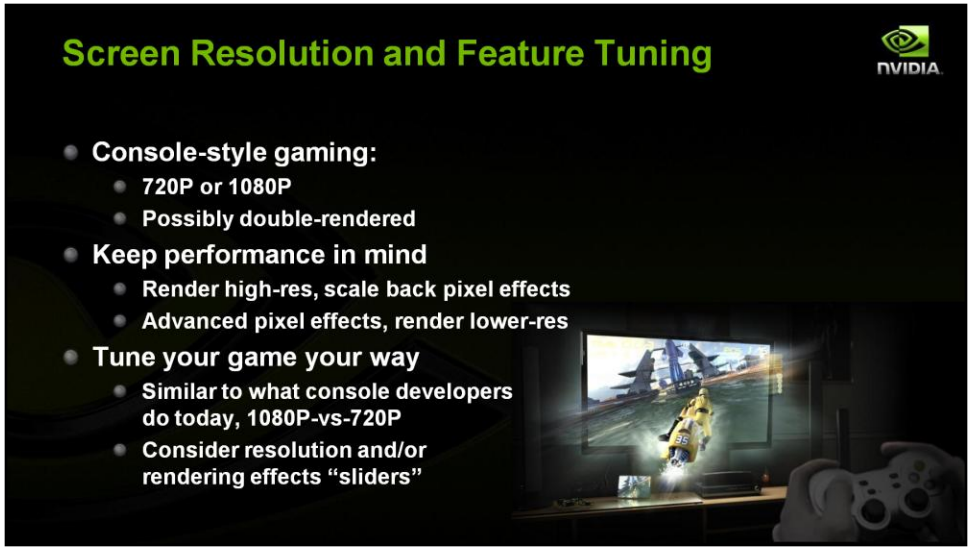


Buttons can come in as individual KeyEvents, or sometimes they can be paired into an axis (D pads are great examples of this). Axes and buttons can map very differently on different controllers. It may physically be button, but it may be reported as an axis with zero and one as the only returned values. It may look like an axis, but it may show up as two buttons. And Up/Down could map to negative 1 and 1 in either direction. So despite layout normalization efforts, you can't assume standard layouts. Your game simply MUST have a mapping control panel.

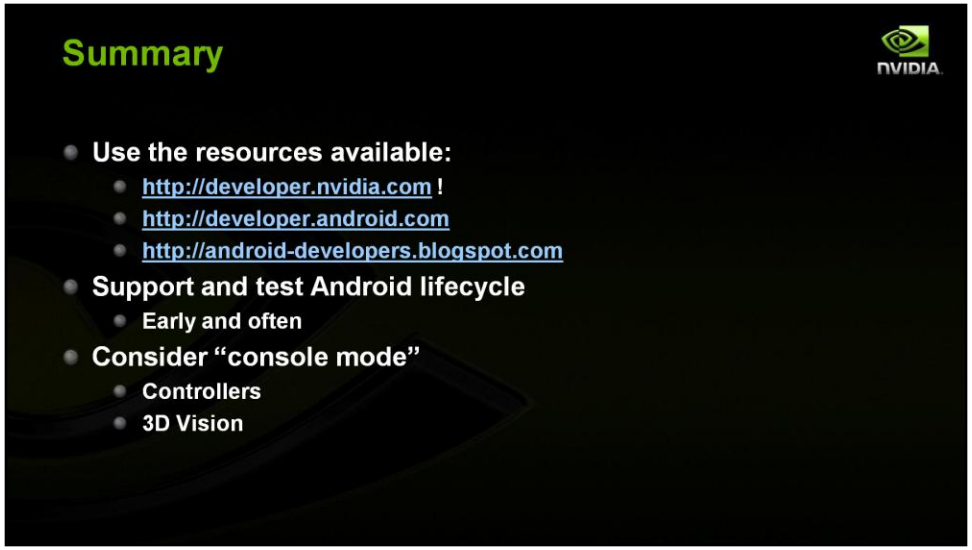
Stereo 3D rendering through HDMI is possible on consumer Tegra devices today! 3D Vision allows Tegra applications to be automatically rendered in 3D stereo at runtime. Apps can also detect whether 3D Vision-based stereo is active at runtime. This can be useful for several reasons, such as adjusting effects to counter the cost of doubled rendering for stereo and modifying any errant effects that break the illusion of stereo. While 3D Vision is automatic, there are some recommendations for app developers…



A few recommendations for apps wanting to optimize for 3D Vision. First, render your UI elements at 0 depth to avoid visual artifacts. If you make changes to your engine's rendering pipeline, talk to your NVIDIA Developer Relations contact, so we can ensure that any 3D Vision profile for the game is up to date. Post-processing fragment shaders should *not* have any position dependencies when running in stereo mode. So for example, `gl_FragCoord` should not be used in stereo mode. When testing, adjust and test at different separation levels during gameplay using the stereo separation slider in the 3D Vision System UI Control Panel. In general, rendering should work for the entire gamut of stereo separation values. In particular, test fullscreen effects at both low and high separation levels. Also, see the NVIDIA developer site and developer relations for sample code of how to detect 3D vision activity, so you can scale your rendering in stereo.



Console-style gaming is HD and may be stereo. It'll be at least 720P or 1080P native, and possibly double-rendered for 3D Vision. That may be a much larger rendering load than the on-device screen, especially for a phone. So keep performance in mind and trade off. If high resolution is more important to your engine than lots of pixel effects, then scale back the per-pixel effects. If you have per-pixel or postprocessing effects that are pivotal to the look of the game, consider rendering your effects or the game to a smaller-than-screen buffer. Basically, tune your game your way, similar to what console developers do today when choosing 1080P versus 720P. In addition, you might consider sliders or options for rendering resolution and effect levels as you might in a PC game, so users can trade off based on their device and preferences.



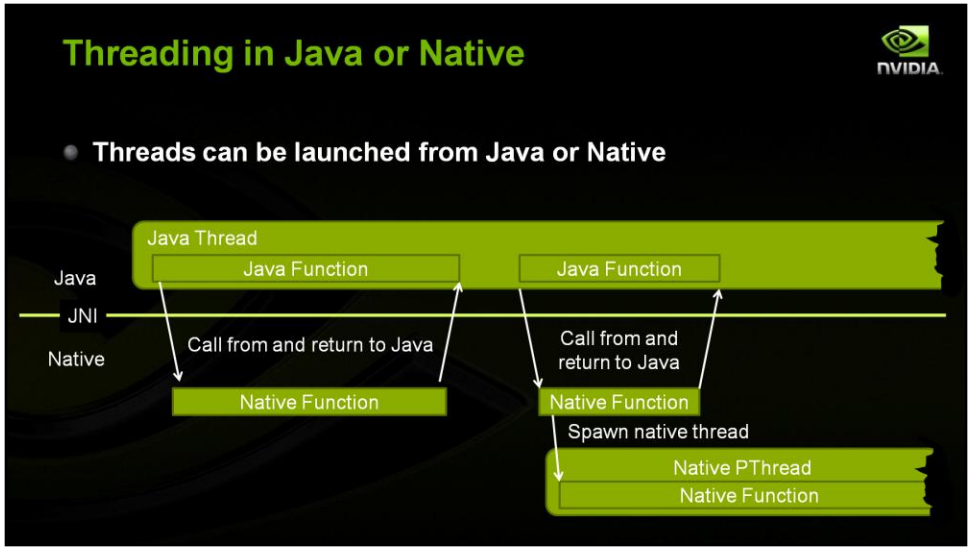
In closing this section, I'd advise you to visit NVIDIA's mobile developers zone and make use of the wealth of resources available. These are based on a ton of experience helping Android game developers ship commercial titles. Keep up with the latest on Android from Google. Support and test Android lifecycle in your games to keep users happy. And consider adding console-like gaming support in your game for differentiation.

So to close out the session, we'd like to present a multi-player, big-screen demo of Shadowgun THD by Madfinger Games. Shadowgun has been extensively optimized for Tegra and console-style mobile gaming, and it just looks amazing. So I'll hand over the microphone to our folks here to give you a feeling for what's possible.

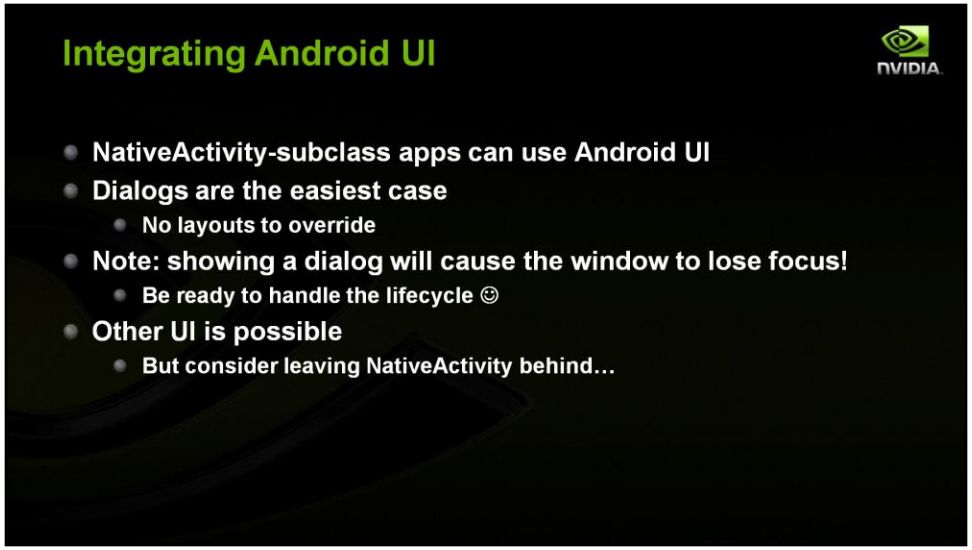
The Tegra 3-optimized version of Shadowgun features console-quality water, enhanced rag-doll physics, particle effects, enhanced shaders and dynamic textures.


Questions?
NVIDIA

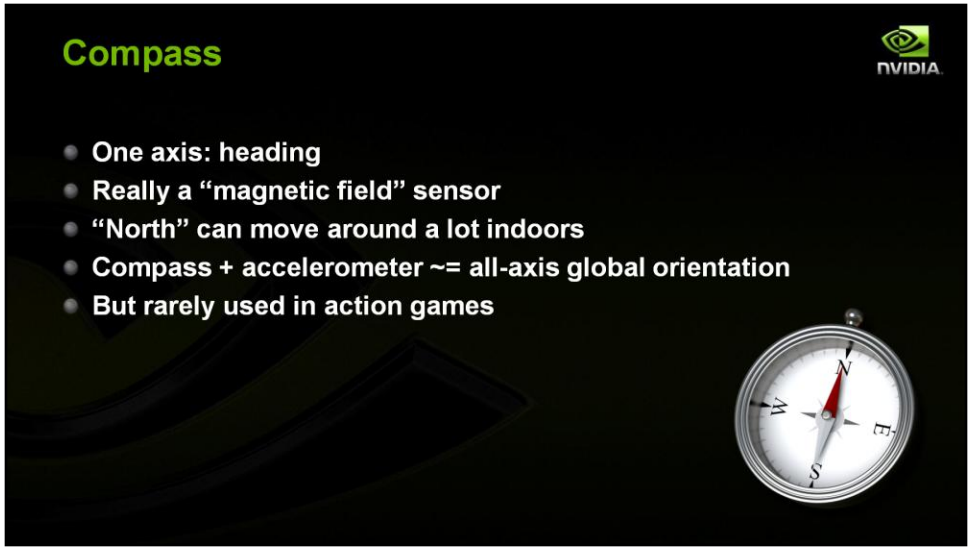

Backup
NVIDIA



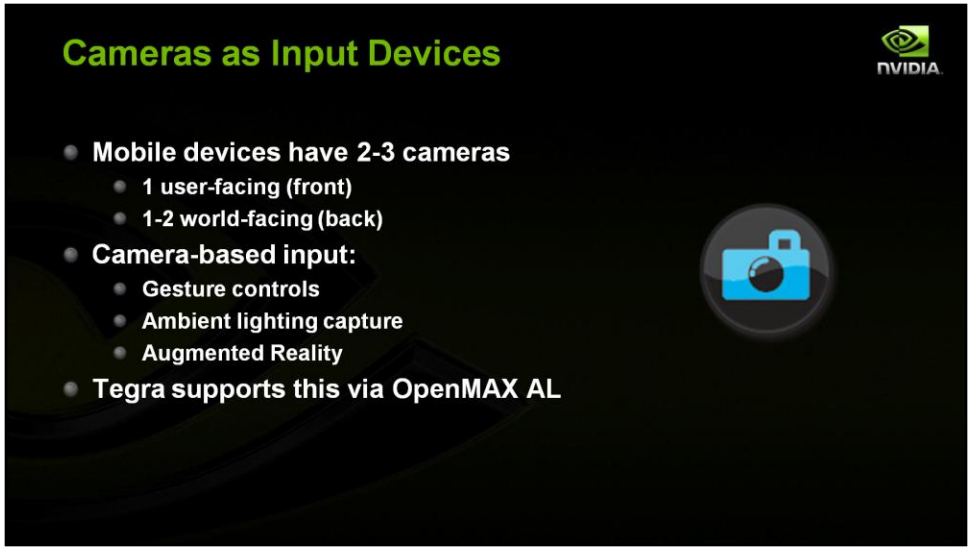
Threads running native code can be launched from Java or Native. Java threads can call into native code, which then runs in the context of the Java thread. From Java's point of view, that thread is blocked until the native call returns. This is handy for threading compartmentalized operations, as the Java threads are easy to launch. JNI is always available in native calls from Java threads, as the Java thread implicitly has a JNI context. Native code can launch Pthreads with no relation to Java. Handy when native algorithms need to communicate between threads all in native code. But JNI is NOT available unless the thread manually registers itself with the JVM



Apps that subclass from NativeActivity can use Android UI elements; they can even create other activities for truly complex UI. However, dialog boxes and alerts are the easiest cases. The Java code is simple, only needing to write simple strings for prompts and attach some simple result callbacks. Response-handling can be done via JNI calls down to native, but keep in mind that these response callbacks will come to native in a different thread than main! And remember that showing a dialog on top of your window will cause your 3D window to lose focus. More lifecycle fun! Other UI cases are possible, but if you REALLY want to use a lot of Android UI, you might consider replacing NativeActivity with your own Java-to-native “wrapper” class straight off of the base Activity.

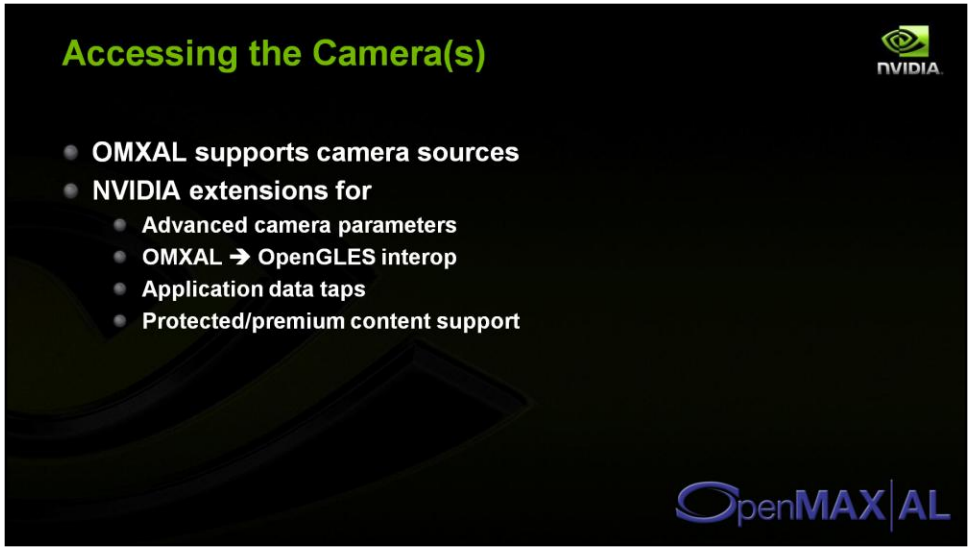


Compass only handles one axis: global heading. It should really be thought of as what it is: a “magnetic field” sensor. “North” can move around or be noisy, especially indoors. Compass is sometimes used along with accelerometer to provide an all-axis global orientation value. For example, this is useful in games that involve placing multiple phones/tablets on a table and using them as a virtual display. But the noise and often slow response means compass is not ubiquitous in action games.

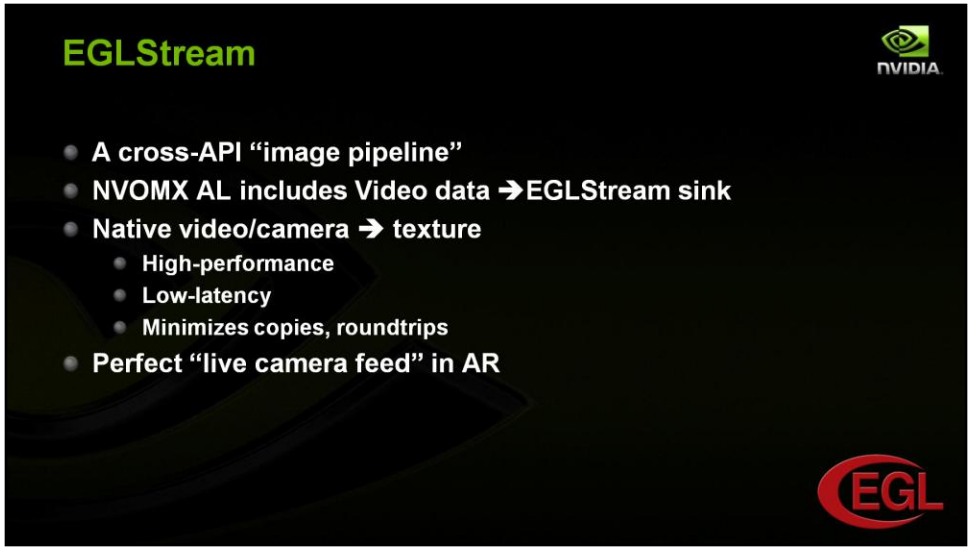


Current and coming mobile devices have 2-3 cameras. Generally, a single user-facing or front camera, and one or two (stereo) world-facing or back cameras. With these, camera-based input becomes an interesting option. Gesture based controls on either camera, ambient/environment-lighting images for contextual rendering and gameplay, and of course, Augmented Reality. Tegra supports cameras in native Android code via OpenMAX AL

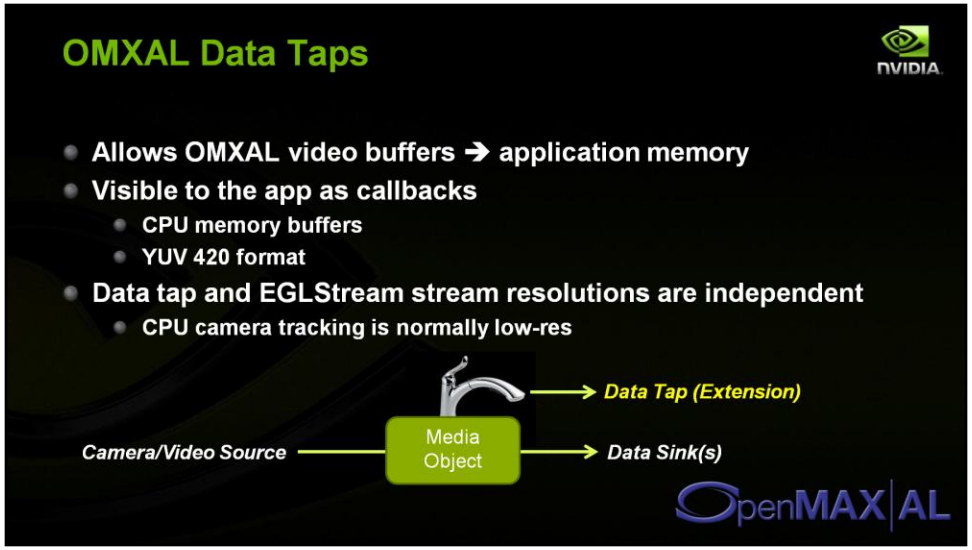
OpenMAX AL is a native C-based media API using COM-like objects. It is designed for high-level application-level use (unlike its brother, OpenMAX IL). But like OpenMAX IL, it is an extensible Khronos standard. It ships today in base form in the NDK for API level 14 and beyond, which equates to Ice Cream Sandwich. It includes the concepts of sources, sinks and processors for video playback sources, camera(s), etc.



OpenMAX AL supports the creation of camera sources in its base spec, along with the ability to attach them to preview windows and video encoders for the videocamera use case. NVIDIA adds extensions for: Querying and setting additional advanced camera parameters, doing OpenMAX AL to OpenGL ES video pipelining, passing live video buffers to an app as data taps, and support for premium or protected video content.



EGLStream is a form of cross-API “image pipeline”. NVIDIA’s OMXAL includes an extension that allows EGLStream to be a sink for video data. This creates a high-performance, low-latency native video/camera-to-texture path. It minimizes copies and avoids the need to upload each frame as a texture from system memory. It is useful for creating the “live camera feed” on screen that is used in most AR applications



Data taps allow OMXAL video buffers to be routed into application/CPU-accessible memory buffers. The data is visible to the app as callbacks bearing buffer pointers. The image data is currently in YUV 4:2:0 format. An important feature is that the resolution of the data tap buffers is independent of any EGLStream stream from the same camera. This is useful for low-res, CPU camera tracking computations while rendering the color image at screen resolution.