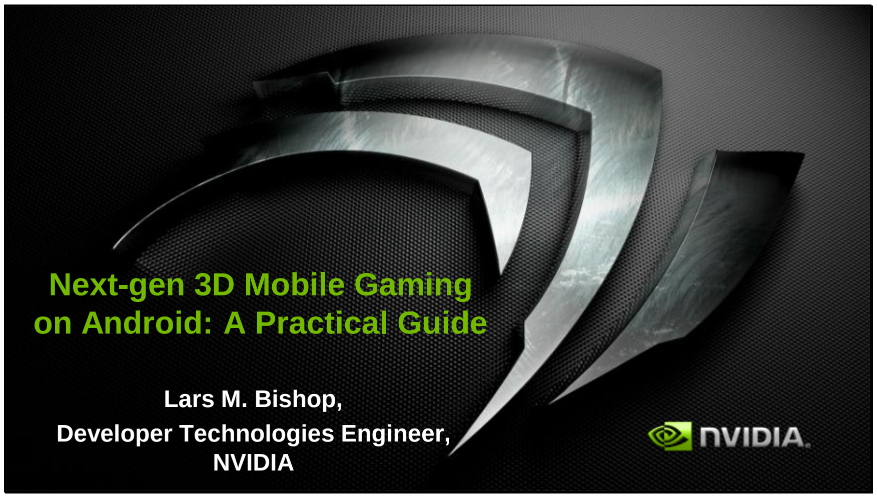
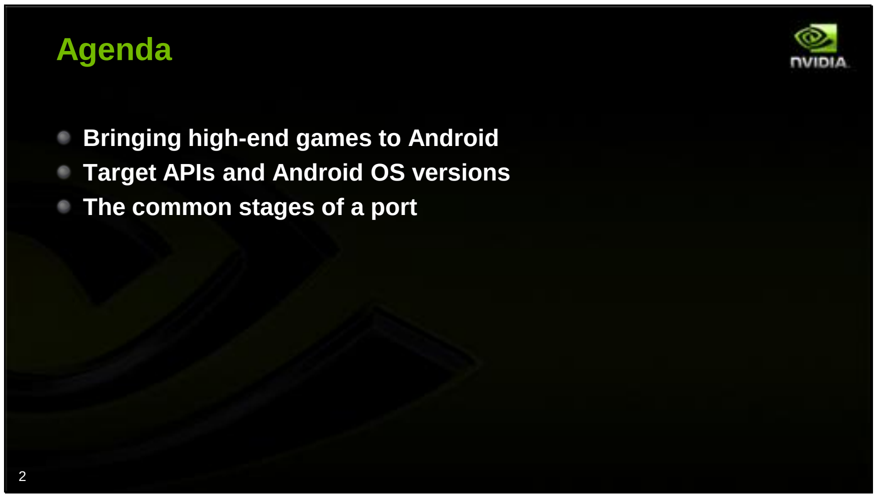
Slide 1



Next-gen 3D Mobile Gaming on Android: A Practical Guide

Lars M. Bishop,
Developer Technologies Engineer,
NVIDIA

Slide 2



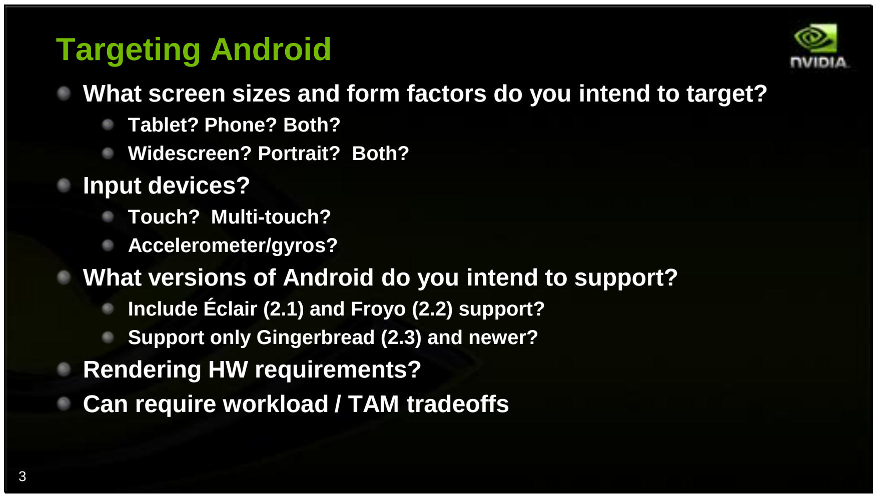
Agenda

- Bringing high-end games to Android
- Target APIs and Android OS versions
- The common stages of a port

2

This presentation is about next gen 3D gaming on Android and Tegra. We’ve got a lot to cover here today, but at the top level, we’re going to discuss the most common paths we’ve seen for bringing high-end 3D games to Android. We’ll discuss some of the market share and feature tradeoffs developers must make, and we’ll detail some issues we see developers facing in each of the three common stages of an Android port or development project.

Slide 3



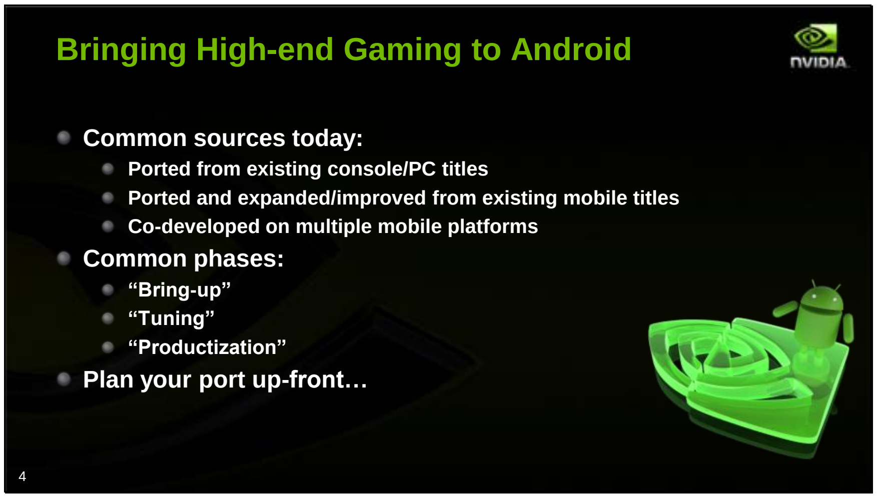
Targeting Android

- What screen sizes and form factors do you intend to target?
  - Tablet? Phone? Both?
  - Widescreen? Portrait? Both?
- Input devices?
  - Touch? Multi-touch?
  - Accelerometer/gyros?
- What versions of Android do you intend to support?
  - Include Éclair (2.1) and Froyo (2.2) support?
  - Support only Gingerbread (2.3) and newer?
- Rendering HW requirements?
- Can require workload / TAM tradeoffs

3

Android devices are not all cut from the same cloth. There are a lot of decisions to make regarding what you mean when you say “porting to Android”. For example, Android’s support of both phones and tablets of multiple sizes requires you to consider the screen sizes, form factors and orientations you intend to target.
As for input, how many touch points do you need? Do you want to require accelerometer or gyros?
Most importantly from a market perspective, what versions of Android do you need to support?
These decisions will affect your total available market size and your development workload

Slide 4



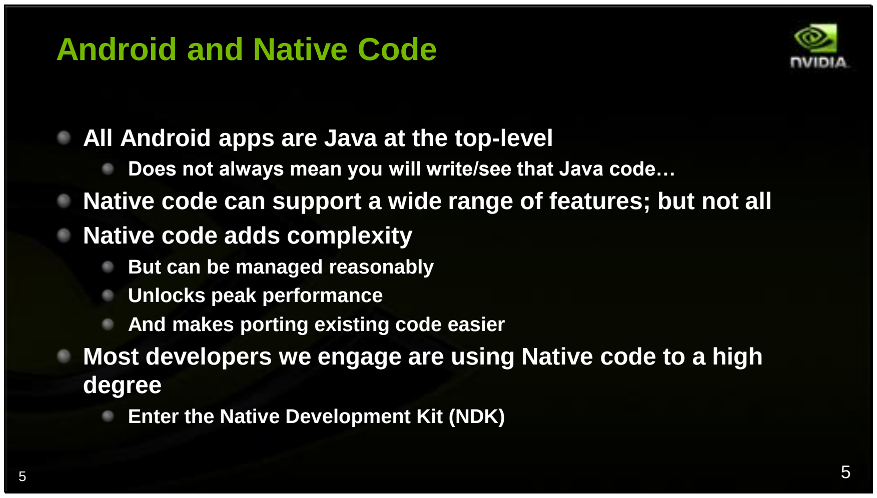
Bringing High-end Gaming to Android

- Common sources today:
  - Ported from existing console/PC titles
  - Ported and expanded/improved from existing mobile titles
  - Co-developed on multiple mobile platforms
- Common phases:
  - “Bring-up”
  - “Tuning”
  - “Productization”
- Plan your port up-front…

4

Currently, most high-end 3D gaming content that we see coming to Android is either existing console and PC titles porting to Android as their first mobile platform, existing mobile titles being ported and improved for Tegra and Android, or new purpose-built mobile titles being developed on multiple mobile platforms simultaneously. We’ve seen porting normally include three distinct phases: An initial “Bring-up” phase where the game is quickly brought to “first playable”, then a “Tuning” phase where the performance and playability are improved, and then a “Productization” phase where the game moves from demo to marketable property. Note that some decisions made or ignored early on in the process can have consequences later…

Slide 5



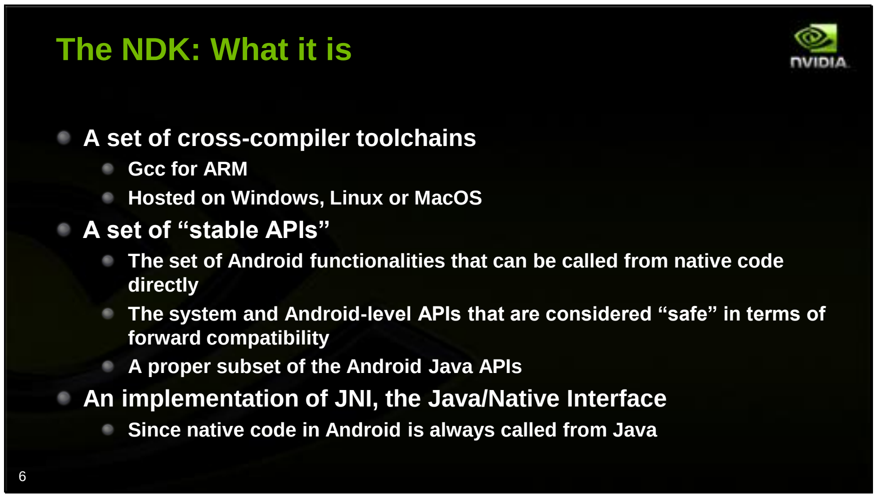
Android and Native Code

- All Android apps are Java at the top-level
  - Does not always mean you will write/see that Java code…
- Native code can support a wide range of features; but not all
- Native code adds complexity
  - But can be managed reasonably
  - Unlocks peak performance
  - And makes porting existing code easier
- Most developers we engage are using Native code to a high degree
  - Enter the Native Development Kit (NDK)

5 5

The first issues that most games face is the questions of Java versus C code. The question of Java versus native code has several facets on Android which we'll detail in this talk, but for now, an overview. The question is oversimplified: ***all*** Android apps are Java at the top level, although in the latest versions of Android, you the developer may not have to write any of that Java code.
Java-level APIs give access to all Android platform features
Native (C/C++)-level APIs give access to many important game platform features, but not all of them
Java-only apps can be simpler to code and manage
But Native apps unlock both peak system/app performance and allow for reuse of existing console and PC engine source code
The choice is per-game, but for high-end, differentiated content, we mainly see Native-based games and engines
Native C++ code is handled via the Native Development Kit and generally called from a Java-based app class. We'll be spending a good bit of quality time with the NDK, so let me introduce it.

Slide 6



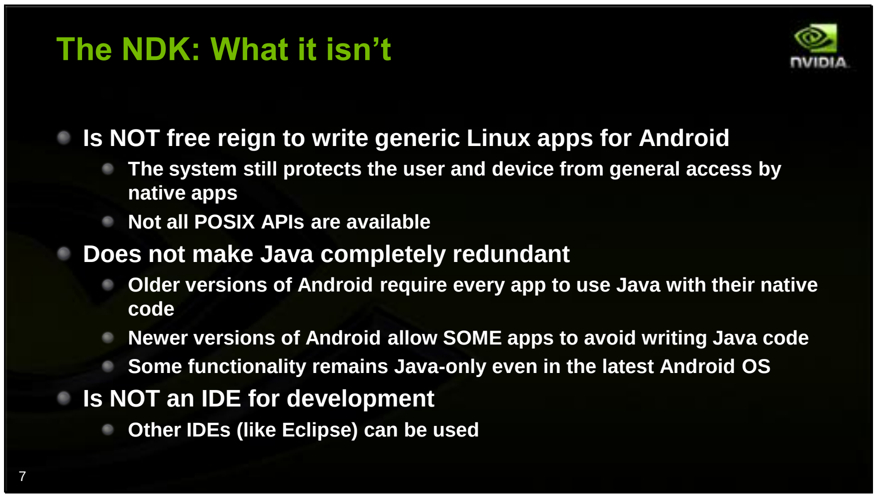
The NDK: What it is

- A set of cross-compiler toolchains
  - Gcc for ARM
  - Hosted on Windows, Linux or MacOS
- A set of “stable APIs”
  - The set of Android functionalities that can be called from native code directly
  - The system and Android-level APIs that are considered “safe” in terms of forward compatibility
  - A proper subset of the Android Java APIs
- An implementation of JNI, the Java/Native Interface
  - Since native code in Android is always called from Java

6

Put simply, the NDK is a set of cross-compiler toolchains that build your C and C++ code for Android devices, along with a set of headers and libs for the APIs that Android allows native code to call directly. These APIs are known as “stable” APIs, not in the sense of being stable at runtime, but in the sense that the APIs themselves won’t require changes moving forward, and thus an app built on them can be forward-compatible with new Android versions. Finally, the NDK includes as a part of its stable APIs the Java-to-Native code bindings known as JNI, or the Java Native Interface. All of this serves to generate shared library code that can interop with the Java level app.

Slide 7



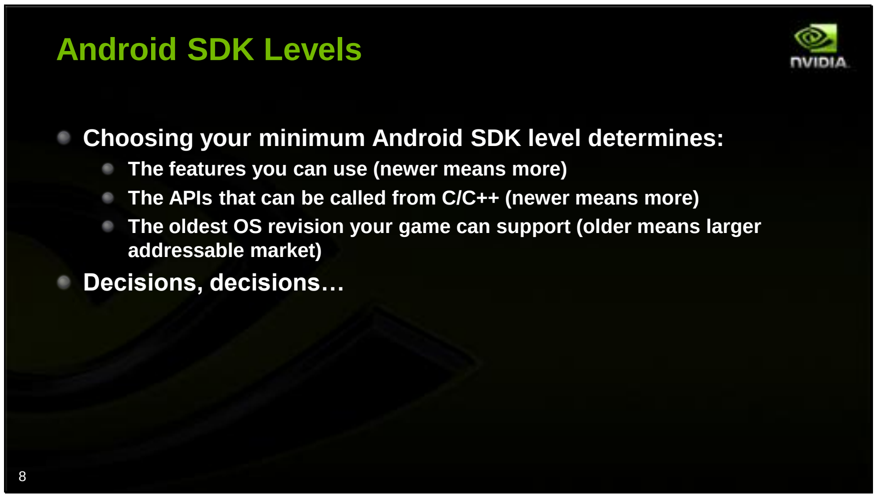
The NDK: What it isn't

- Is NOT free reign to write generic Linux apps for Android
  - The system still protects the user and device from general access by native apps
  - Not all POSIX APIs are available
- Does not make Java completely redundant
  - Older versions of Android require every app to use Java with their native code
  - Newer versions of Android allow SOME apps to avoid writing Java code
  - Some functionality remains Java-only even in the latest Android OS
- Is NOT an IDE for development
  - Other IDEs (like Eclipse) can be used

7

The NDK is not a free hand to start trapping into the kernel and treating the Android device as an open Linux system. The NDK APIs still protect the user's device from various forms of access. Not all POSIX APIs are available, and not all Android APIs are available. Using APIs exposed in Android's source code but not in the NDK can lead to issues on later OS versions. Until the most recent versions of Android, the NDK did not make Java redundant at all, since every app still had to write Java code. While that is changing as more and more functionality if made available to the NDK, many apps will still end up writing Java code for one feature or another. Finally, the NDK does not include a specific IDE like Visual Studio. It can be integrated into any number of other IDEs, Eclipse being the most common.

Slide 8



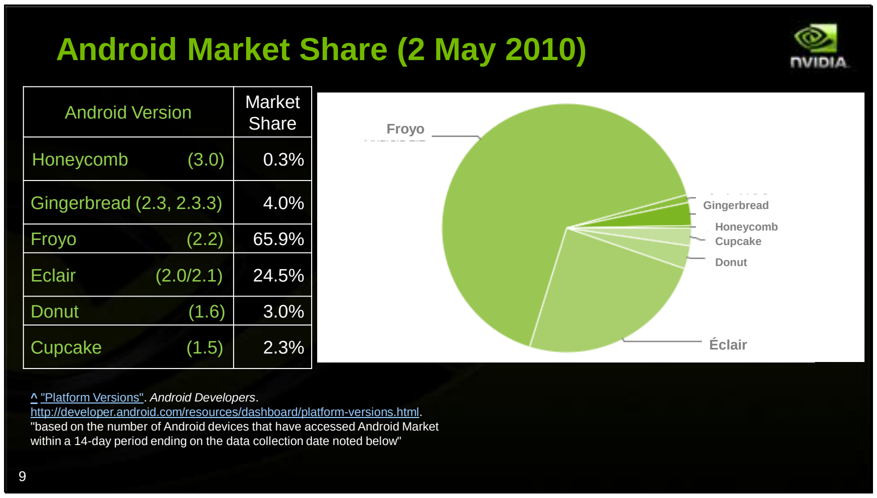
Android SDK Levels

- Choosing your minimum Android SDK level determines:
  - The features you can use (newer means more)
  - The APIs that can be called from C/C++ (newer means more)
  - The oldest OS revision your game can support (older means larger addressable market)
- Decisions, decisions…

8

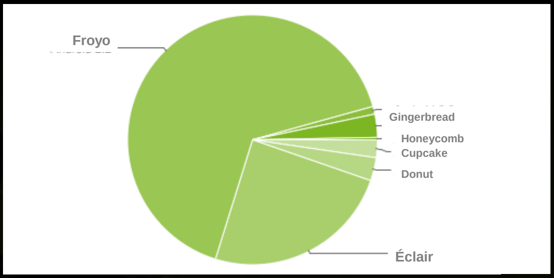
The specific features available in native code (as opposed to Java) depend on the SDK level you target. Basically, the higher the minimum SDK level, the newer the OS required to run the application, and the greater the number of APIs that can be called directly from native code. But that also means that when you select your minimum SDK level, you select the size of your target market. Apps requiring a min SDK level newer than the device's OS will be filtered out of the Market for that device. Choosing your SDK level is a tradeoff between native code feature-set and range of supported devices

Slide 9



Android Market Share (2 May 2010)

Android Version		Market Share
Honeycomb	(3.0)	0.3%
Gingerbread (2.3, 2.3.3)		4.0%
Froyo	(2.2)	65.9%
Eclair	(2.0/2.1)	24.5%
Donut	(1.6)	3.0%
Cupcake	(1.5)	2.3%

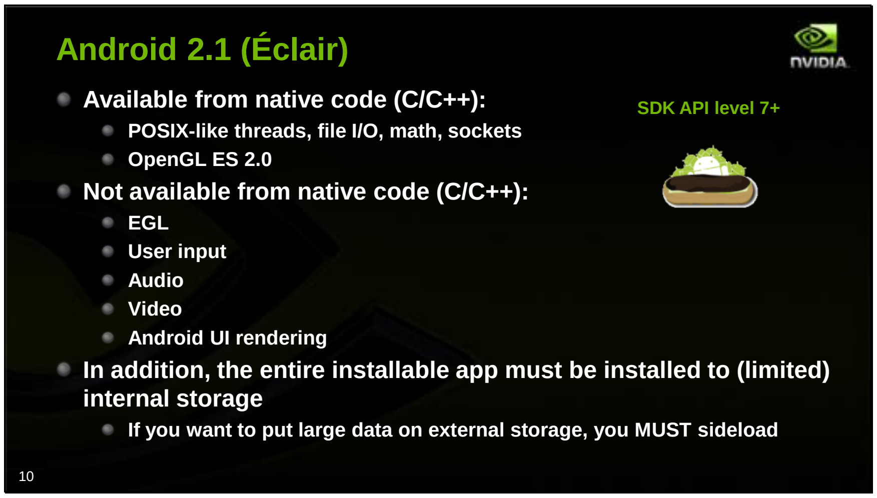


^ "Platform Versions". *Android Developers*. http://developer.android.com/resources/dashboard/platform-versions.html. "based on the number of Android devices that have accessed Android Market within a 14-day period ending on the data collection date noted below"

9

Let's look at some Android OS version market share numbers released at the beginning of May. This most recent public chart gives the basic idea of the breakdown of Android versions across all Android devices. If you support Éclair and newer, you will cover about 95% of the active devices. If you make Froyo your minspec, you'll still net about two-thirds and rising. If you choose to make Gingerbread or Honeycomb your minspec, you'll be waiting at least a while for your total addressable market to expand. So why would you select a newer OS version as a minspec? The main answer is required features: let's discuss them in more detail.

Slide 10



Android 2.1 (Éclair)

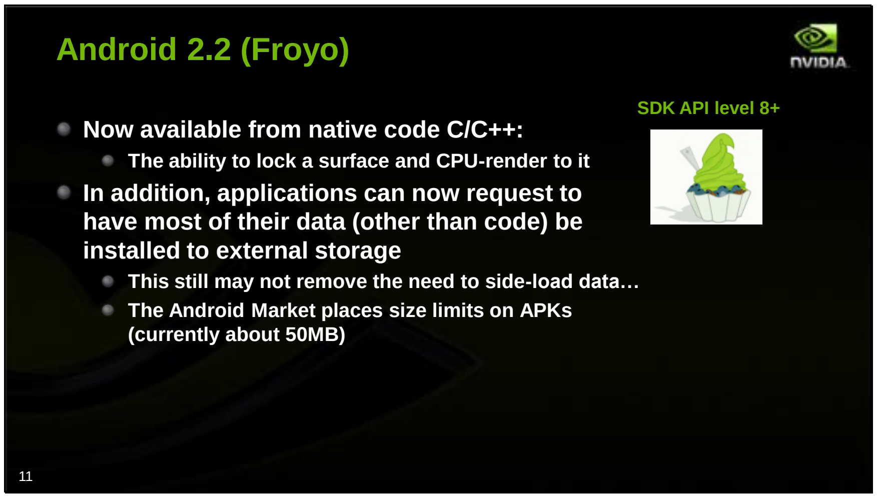
- Available from native code (C/C++):
  - POSIX-like threads, file I/O, math, sockets
  - OpenGL ES 2.0
- Not available from native code (C/C++):
  - EGL
  - User input
  - Audio
  - Video
  - Android UI rendering
- In addition, the entire installable app must be installed to (limited) internal storage
  - If you want to put large data on external storage, you MUST sideload

SDK API level 7+

10

Since going back as far as Éclair as the minimum platform opens up the market to cover over 95% of the currently active Android devices, we'll use it as a baseline. As of Éclair (AKA Android 2.1 or API level 7), you can do basic system-level work like file I/O, math, networking and threads from native code. You can also render 3D content with OpenGL ES 2.0 from native code, but you still have to use Java to initialize the buffers and contexts. Furthermore, in order to flesh out your game with user input events and sound, you have use some Java at this API level. Finally, apps built for Éclair had to be installed to the device's internal storage, rather than removable SD cards. So this limited the practical size of the app's installer package. Most apps had to side-load their data from the network and place it onto external storage themselves. So game development is definitely possible on API level 7, but there are programming and logistics challenges.

Slide 11



Android 2.2 (Froyo)

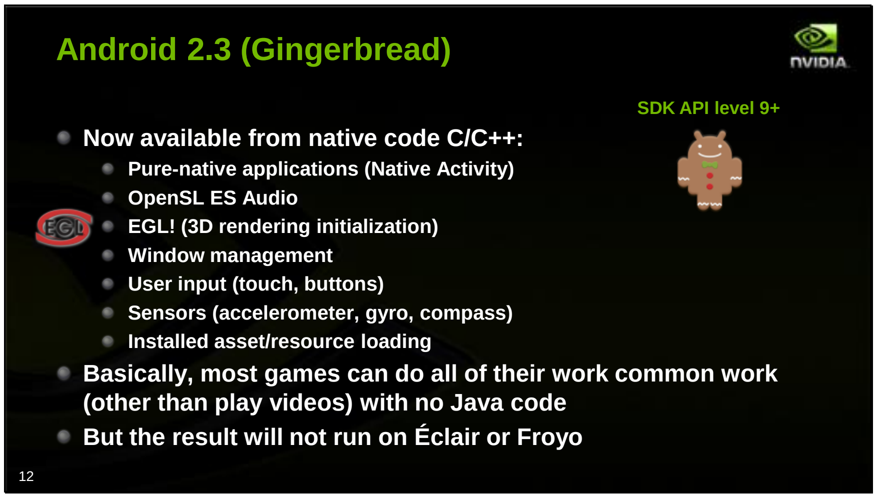
SDK API level 8+

- Now available from native code C/C++:
  - The ability to lock a surface and CPU-render to it
- In addition, applications can now request to have most of their data (other than code) be installed to external storage
  - This still may not remove the need to side-load data…
  - The Android Market places size limits on APKs (currently about 50MB)

11

Switching to Froyo as the minspec currently equates to about two-thirds of the active installed base. Froyo didn't add any interesting new APIs to the NDK. However, building an application with Froyo as the minspec DOES make it possible to specify that the app can (or even should) be installed to external storage like a large SD card, rather than the limited user data partition. However, this is not carte blanche to pack huge assets into your apps installer (or APK). Side loading will still be required for the largest games, as the Android Market enforces APK size limits independent of OS version. Currently, that limit is about 50 megabytes per APK.

Slide 12



Android 2.3 (Gingerbread)

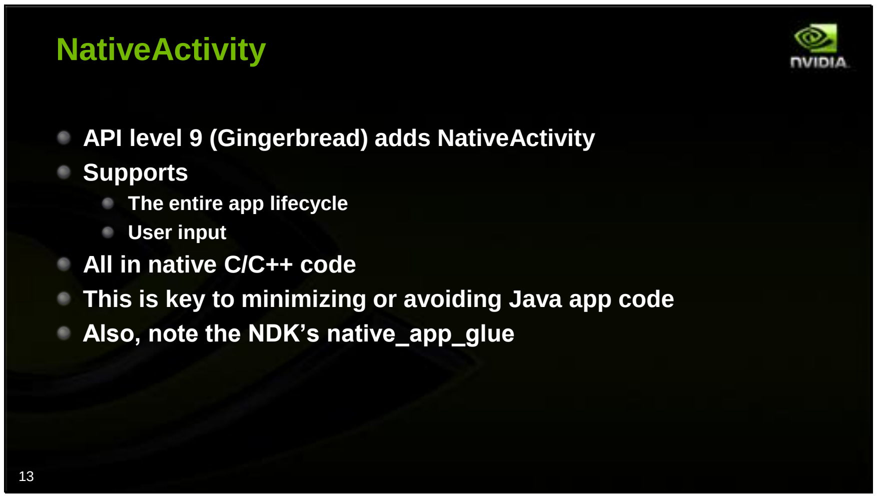
SDK API level 9+

- Now available from native code C/C++:
  - Pure-native applications (Native Activity)
  - OpenSL ES Audio
  - EGL! (3D rendering initialization)
  - Window management
  - User input (touch, buttons)
  - Sensors (accelerometer, gyro, compass)
  - Installed asset/resource loading
- Basically, most games can do all of their work common work (other than play videos) with no Java code
- But the result will not run on Éclair or Froyo

12

Keeping in mind that Gingerbread's current installed market share is small but obviously growing, its SDK level adds pivotal functionality, known as NativeActivity. For some 3D games, NativeActivity may obviate the need for any app-specific Java code. Other than video playback, which is still Java-only, the core functionality is all available in native code on Gingerbread and beyond. This includes setup and management of 3D contexts, application lifecycle events, audio and user input devices. While almost all of this can be done in earlier SDK levels via Java-to-Native JNI wrappers, the new native interfaces are undeniably enticing.

Slide 13



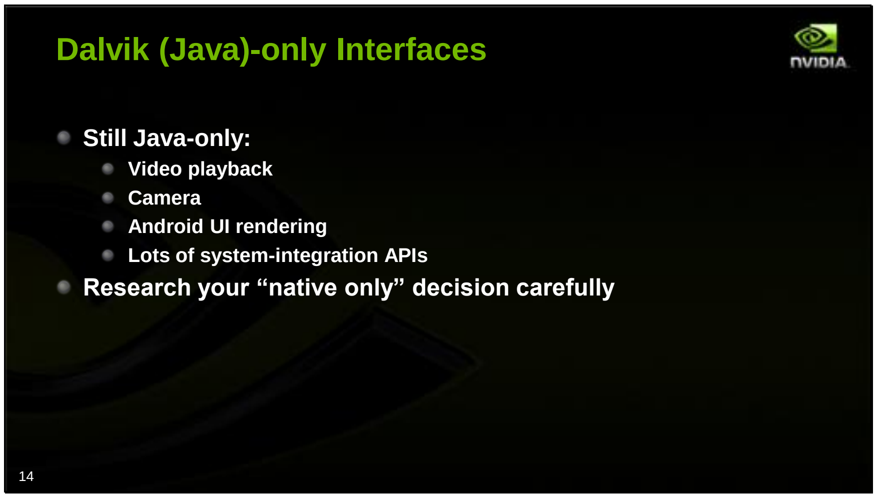
NativeActivity

- API level 9 (Gingerbread) adds NativeActivity
- Supports
  - The entire app lifecycle
  - User input
- All in native C/C++ code
- This is key to minimizing or avoiding Java app code
- Also, note the NDK's native_app_glue

13

With the advent of Gingerbread's NativeActivity, applications using Gingerbread as a minspec can be written entirely in native C++ code. This was specifically designed to make it easier to port native content to Android, and from the buzz it has generated with developers, it seems to have succeeded. While NativeActivity can be used alone, the Android NDK samples also include a very important sample framework called native_app_glue that does an excellent job of making NativeActivity easier to use in practice.

Slide 14



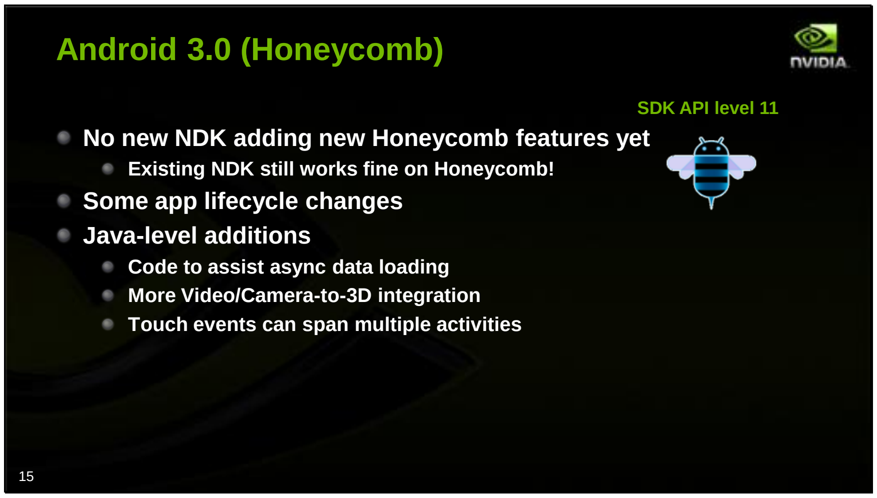
Dalvik (Java)-only Interfaces

- Still Java-only:
  - Video playback
  - Camera
  - Android UI rendering
  - Lots of system-integration APIs
- Research your "native only" decision carefully

14

Some features have yet to be exposed to the NDK, even in the latest SDK level. These include Video playback, camera, Android UI rendering, and some system-integration APIs. While these may not be required by most games, you should be prepared for the idea that you may not be able to completely avoid Java code by moving to Gingerbread as a min spec. So if you believe that using Gingerbread as a minspec will allow you to completely avoid Java, research carefully to ensure that this is indeed true, and that it is actually important. Plenty of native games have done well with small amounts of Java code and a lower minspec.

Slide 15



Android 3.0 (Honeycomb)

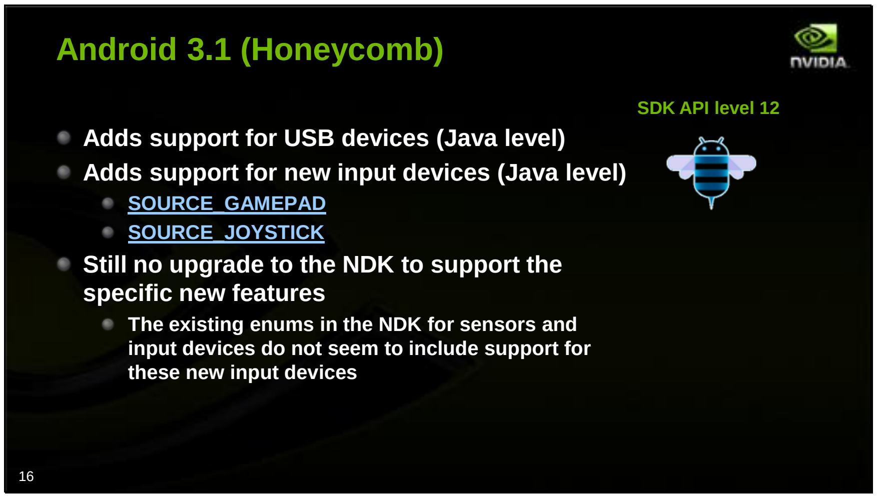
SDK API level 11

- No new NDK adding new Honeycomb features yet
  - Existing NDK still works fine on Honeycomb!
- Some app lifecycle changes
- Java-level additions
  - Code to assist async data loading
  - More Video/Camera-to-3D integration
  - Touch events can span multiple activities

15

Of course, there's also Android 3.0, or Honeycomb, the OS running on the Motorola Xoom tablet. That is API level 11, and as of this morning, there was no new NDK version that adds specific API level 11 functionality in native code. So new features are at the Java level; I list a few here, such as standardized video texture support and some new classes to assist with background data loading.

Slide 16



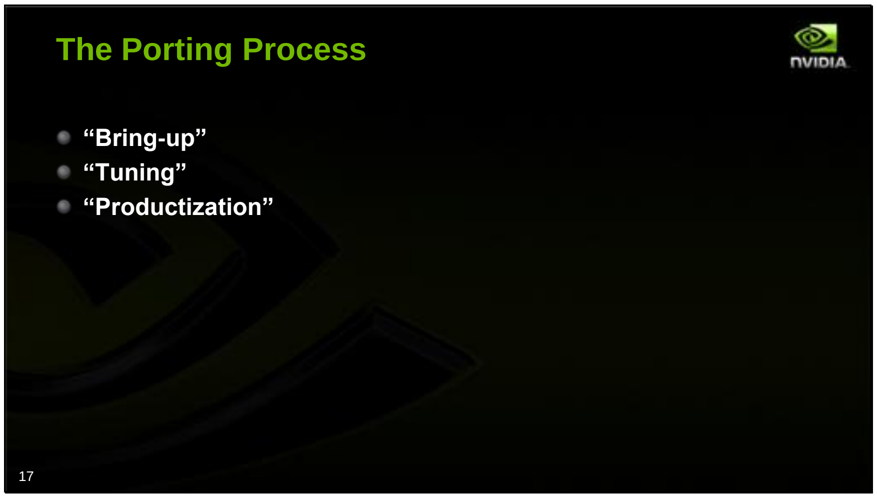
Android 3.1 (Honeycomb)

SDK API level 12

- Adds support for USB devices (Java level)
- Adds support for new input devices (Java level)
  - SOURCE_GAMEPAD
  - SOURCE_JOYSTICK
- Still no upgrade to the NDK to support the specific new features
  - The existing enums in the NDK for sensors and input devices do not seem to include support for these new input devices

16

Slide 17



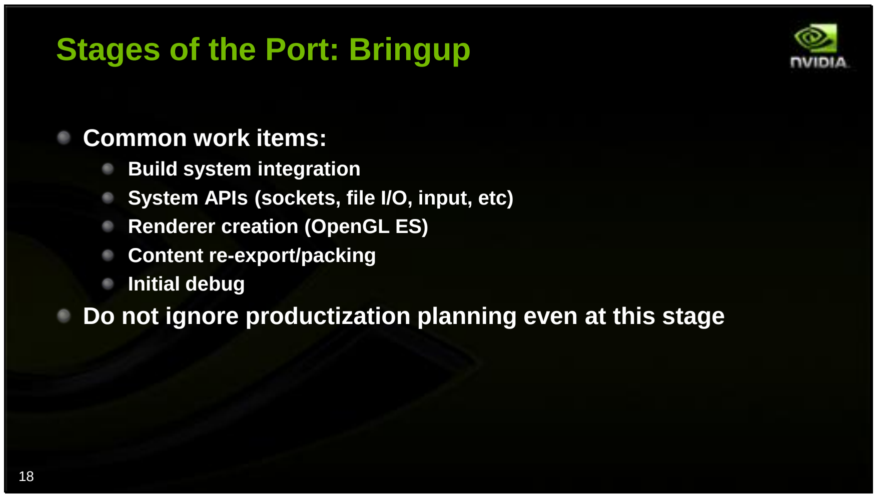
The Porting Process

- “Bring-up”
- “Tuning”
- “Productization”

17

Now, we’ll spend the rest of this presentation spiraling into some details on each of my three steps of the porting process, in the order that an app would attack them. Most of the high-end games we see coming to Android right now are existing titles of one form or another, so we’ll refer to this most frequently as the “porting” process, rather than the development process, and we’ll assume that game code and assets exist in one form or another.

Slide 18



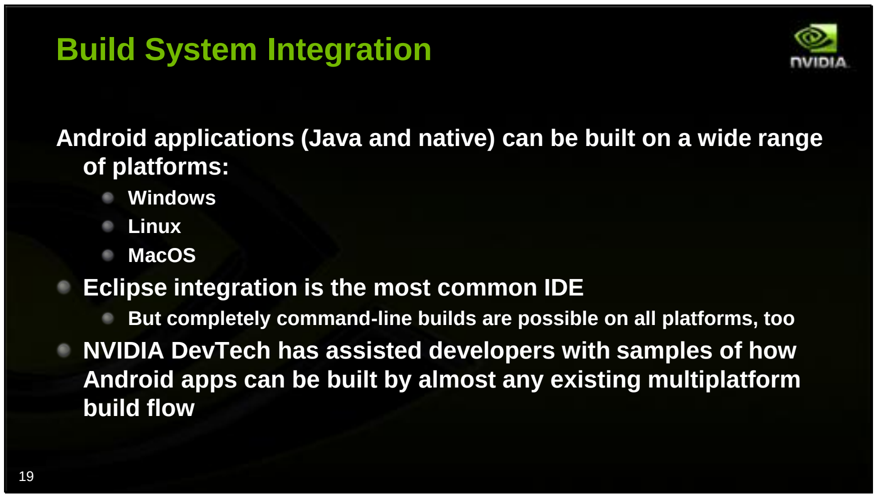
Stages of the Port: Bringup

- Common work items:
  - Build system integration
  - System APIs (sockets, file I/O, input, etc)
  - Renderer creation (OpenGL ES)
  - Content re-export/packing
  - Initial debug
- Do not ignore productization planning even at this stage

18

Bringup is often done as a quick (and dirty) pass to see things running. The most common initial work item in bringup is to integrate the app into the Android build system. Then basic system features like I/O need to be ported. You'll need an OpenGL ES renderer. Also, there's content. Depending on your art toolchain, content re-export and packing can be anything from trivial to challenging. And there's always the joys of knocking down the load-time bugs one by one until the first rendered screen pops up. Depending on the tools you have at your disposal, this can be a challenge on Android.

Slide 19



Build System Integration

Android applications (Java and native) can be built on a wide range of platforms:

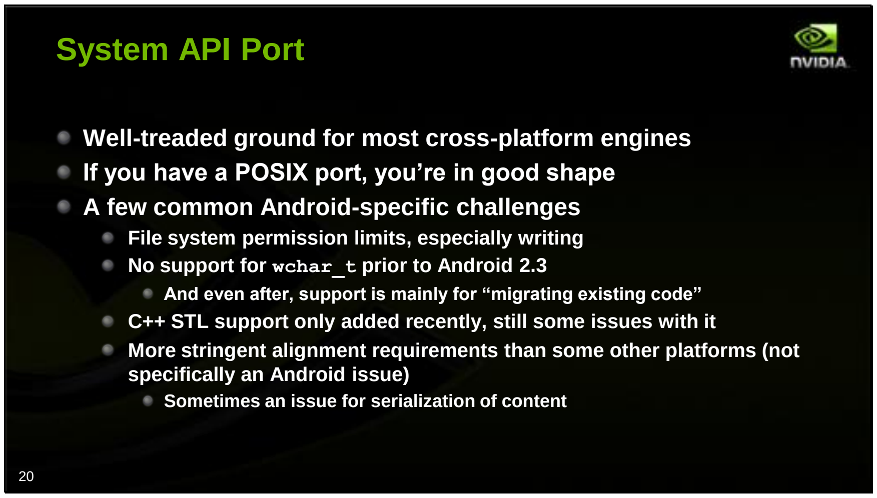
- Windows
- Linux
- MacOS

- Eclipse integration is the most common IDE
  - But completely command-line builds are possible on all platforms, too
- NVIDIA DevTech has assisted developers with samples of how Android apps can be built by almost any existing multiplatform build flow

19

The first hurdle that almost any existing project hits is “how do I compile my code?” Building an Android application involves compiling and linking your native code, building any Java code and then packing the results into an installable pack, known as an APK. The APK can also include game art and assets. All of the stages of building code and packing the app can be accomplished on Windows, Linux or Mac OS. And while Eclipse is the most common IDE, no specific high-level build system like make is required by Android. So most build systems for a given cross-platform game can pretty easily handle building for Android, too. If you game already builds under GCC, then you’re really in good shape. NVIDIA DevTech has helped developers move all manner of build systems over to Android, and in general we’ve found it can be done cleanly.

Slide 20



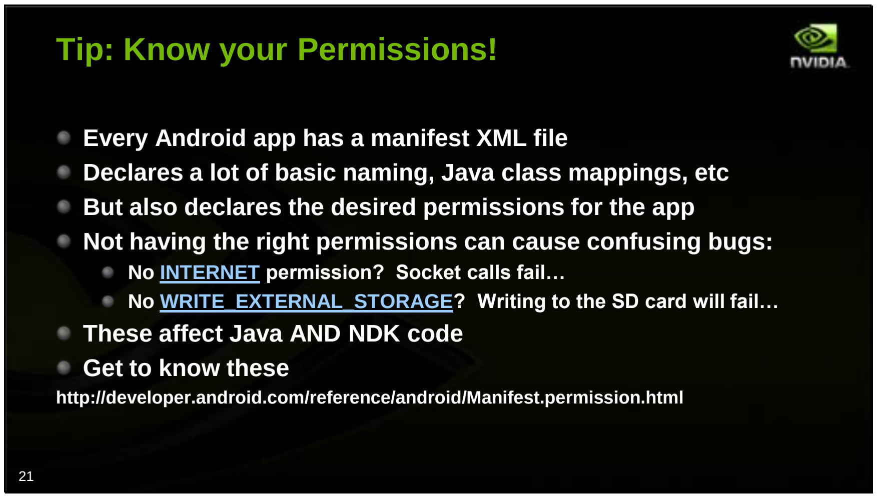
System API Port

- Well-treaded ground for most cross-platform engines
- If you have a POSIX port, you're in good shape
- A few common Android-specific challenges
  - File system permission limits, especially writing
  - No support for `wchar_t` prior to Android 2.3
    - And even after, support is mainly for "migrating existing code"
  - C++ STL support only added recently, still some issues with it
  - More stringent alignment requirements than some other platforms (not specifically an Android issue)
    - Sometimes an issue for serialization of content

20

Once you can compile your code, you'll likely be looking at porting the system level functionalities next. If you have a Linux port of your game, that's the best starting point for your Android version. Pay particular attention to file I/O, since Android enforces some access path limitations. Also, wide character support is problematic and sizes can differ per OS version, so watch out for tricky memory bugs there. STL was only recently added to the NDK and there are features missing, so users of STL may need to pay attention there. Finally, there are often more stringent data access alignment requirements on this platform, so keep that in mind if you use direct binary data loads in your serialization code.

Slide 21



Tip: Know your Permissions!

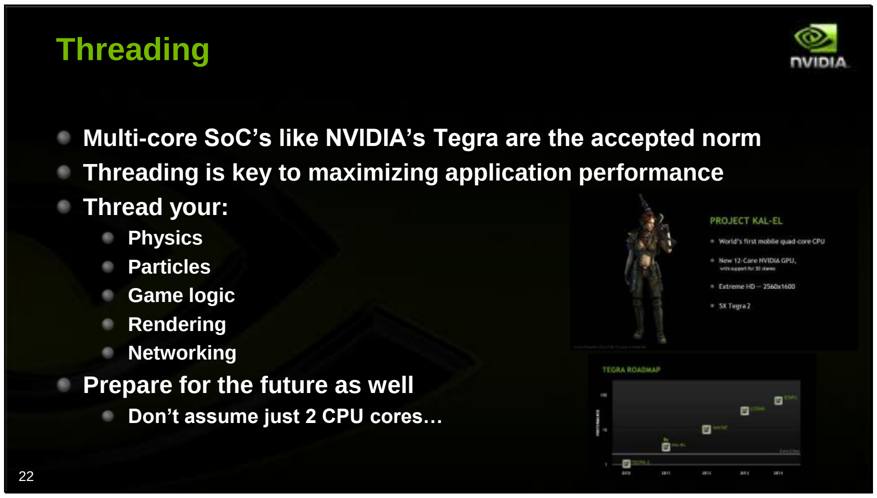
- Every Android app has a manifest XML file
- Declares a lot of basic naming, Java class mappings, etc
- But also declares the desired permissions for the app
- Not having the right permissions can cause confusing bugs:
  - No INTERNET permission? Socket calls fail…
  - No WRITE_EXTERNAL_STORAGE? Writing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

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

21

Android apps all include a manifest XML file that declares a number of application settings. But the manifest also includes a very important list of requested app permissions pertaining to both Java and native code. Not having the right permissions can lead to runtime failures. For example, not having the INTERNET permission declared will cause your native POSIX socket functions to fail. There's also a permission flag for writing to external storage cards. Check out the Manifest permission docs on the Android developer website for a list of all permissions. Request the permissions you need, but keep in mind that users are often suspicious of apps requesting many of them, so request what you need only.

Slide 22



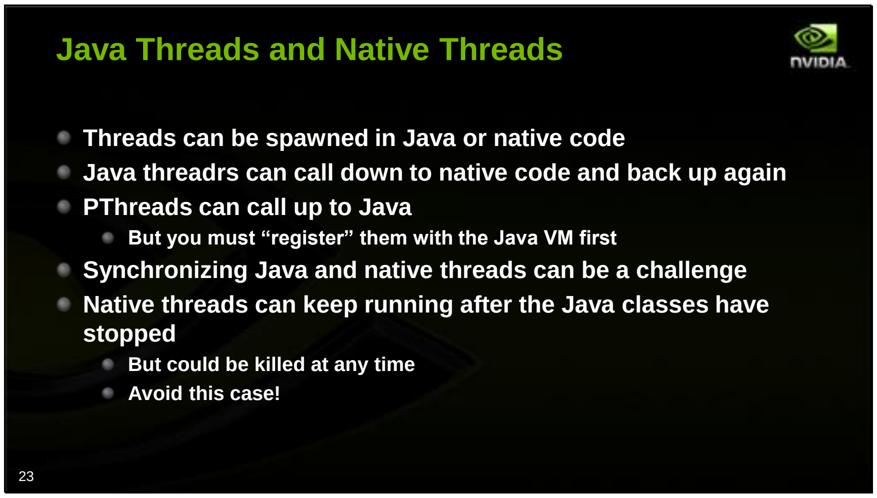
Threading

- Multi-core SoC’s like NVIDIA’s Tegra are the accepted norm
- Threading is key to maximizing application performance
- Thread your:
  - Physics
  - Particles
  - Game logic
  - Rendering
  - Networking
- Prepare for the future as well
  - Don’t assume just 2 CPU cores…

PROJECT KAL-EL
World's first mobile quad-core CPU
New 12-Core NVIDIA GPU,
Extreme HD – 2560x1600
5X Tegra 2
TEGRA ROADMAP

22

Multi-core systems like NVIDIA’s Tegra are really de facto requirements in quality Android devices today. So. application threading is key to maximizing performance. This tends to mesh well with current-generation console and PC titles, which were designed for multi-core platforms from the ground up. Note that on Android, blocking I/O and networking calls almost ***have*** to be moved to a secondary thread. For example, the upcoming Android API levels may ***require*** networking calls be made from a secondary thread or else they will immediately fail. Prepare for the near future. As you may have seen in the news, NVIDIA has been showing the quad-core Project Kal-El since February and other major vendors have announced that they will have quad core solutions by next year. Don’t assume just 2 cores…



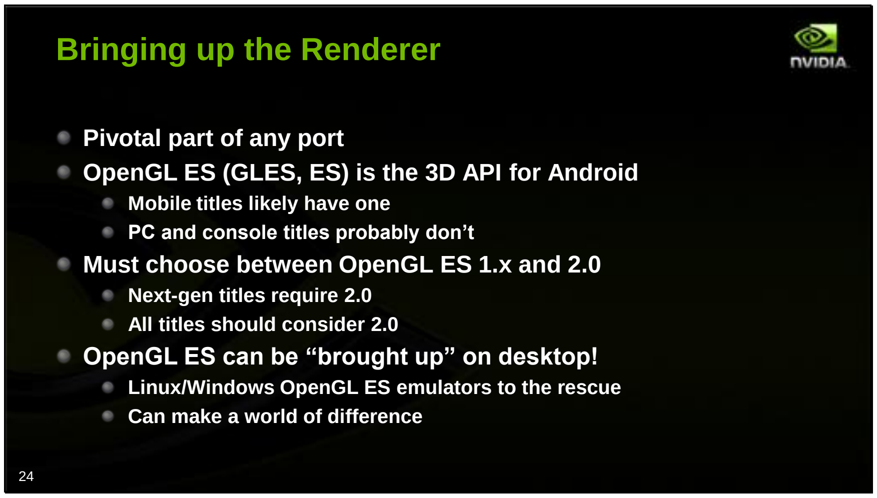
Java Threads and Native Threads

- Threads can be spawned in Java or native code
- Java threadrs can call down to native code and back up again
- PThreads can call up to Java
  - But you must “register” them with the Java VM first
- Synchronizing Java and native threads can be a challenge
- Native threads can keep running after the Java classes have stopped
  - But could be killed at any time
  - Avoid this case!

23

Threads can be spawned from Java or native code. Threads created in Java can call down to native code and back up again using JNI. Native spawned threads can call up to Java code, but you must “register” those threads with the Java VM first or else you’ll crash. Be careful with thread lifespan: Android keeps an app’s process running after the Java class is destroyed. But once the Java class is destroyed, android can choose to kill the process at any time. So while Native threads ***can*** keep running after the Java finishes, don’t risk it.

Slide 24

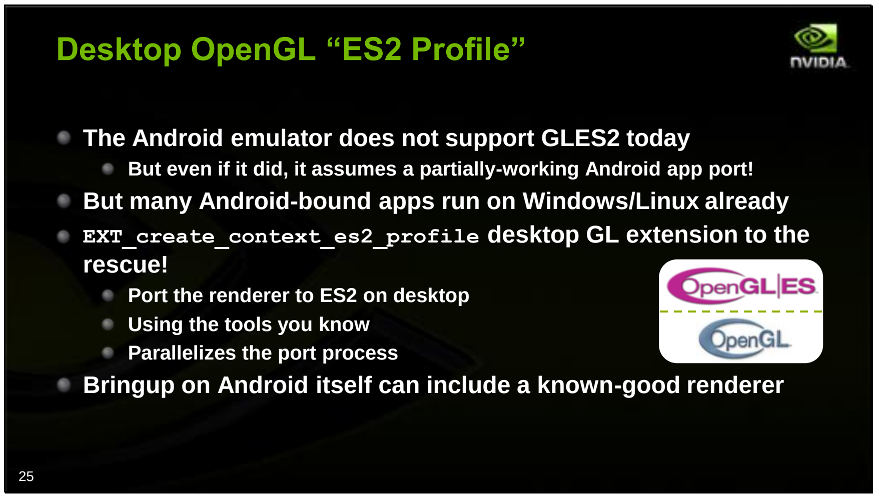


Bringing up the Renderer

- Pivotal part of any port
- OpenGL ES (GLES, ES) is the 3D API for Android
  - Mobile titles likely have one
  - PC and console titles probably don't
- Must choose between OpenGL ES 1.x and 2.0
  - Next-gen titles require 2.0
  - All titles should consider 2.0
- OpenGL ES can be "brought up" on desktop!
  - Linux/Windows OpenGL ES emulators to the rescue
  - Can make a world of difference

24

Bringing up your 3D renderer is a pivotal part of any port. On Android, this means an OpenGL ES (or GLES or ES) renderer. Mobile titles will likely have a ES renderer of some sort already, but PC and console titles may require much more work. You must choose between ES 1.x (fixed function) and ES 2.0 (shaders). Most next-gen, high end titles will require 2.0, but 2.0 should be considered for any title, as it is the "native language" of current and future mobile 3D hardware. You do not need to have your app compiling and loading on Android to start your renderer port, and you don't have to debug your ES renderer on Android. If you have a Linux or Windows build of your codebase, you can use a desktop ES emulator to bring up the OpenGL ES renderer on desktop before running on Android.



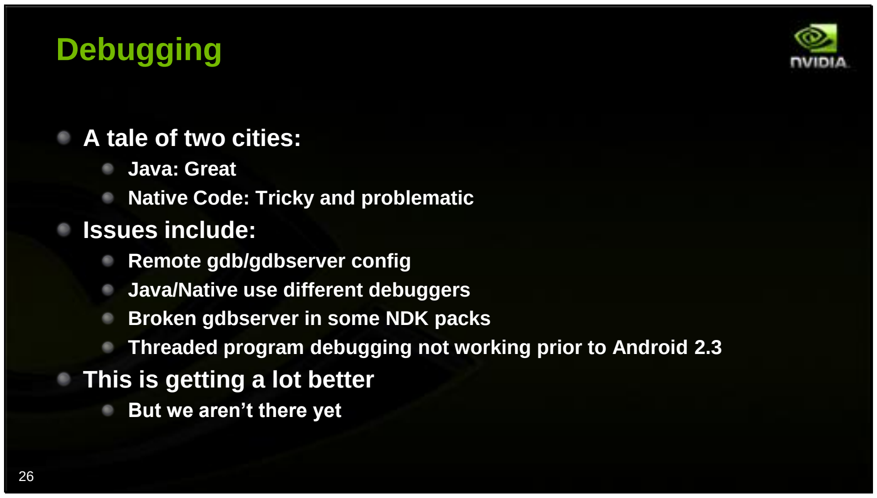
Desktop OpenGL “ES2 Profile”

- The Android emulator does not support GLES2 today
  - But even if it did, it assumes a partially-working Android app port!
- But many Android-bound apps run on Windows/Linux already
- `EXT_create_context_es2_profile` desktop GL extension to the rescue!
  - Port the renderer to ES2 on desktop
  - Using the tools you know
  - Parallelizes the port process
- Bringup on Android itself can include a known-good renderer

25

As of today, the Android emulator on PC cannot support GLES 2. But using an Android device or emulator to debug your renderer assumes that the rest of your Android port is working. Many applications looking to port to Android are already running on Windows or Linux. Since the renderer can be one of the larger parts of a port, it would be nice to be able to bring up an ES 2 renderer in parallel with or even before the Android port. NVIDIA’s desktop drivers support an ES2 profile extension that makes the API act like GLES 2. An ES2-compatible rendering path can be developed on PC without first having to port to Android and can be done using familiar desktop tools. Some of the quickest and best ports we’ve seen used PC-based ES2 renderer development

Slide 26



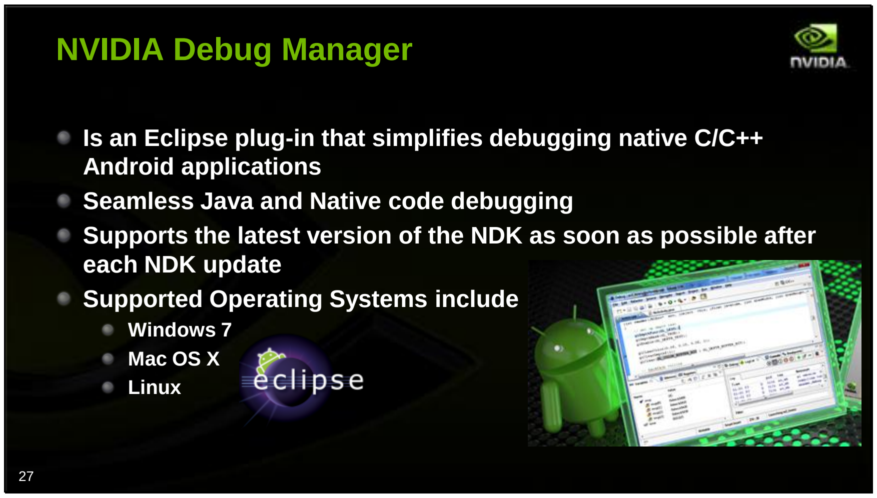
Debugging

- A tale of two cities:
  - Java: Great
  - Native Code: Tricky and problematic
- Issues include:
  - Remote gdb/gdbserver config
  - Java/Native use different debuggers
  - Broken gdbserver in some NDK packs
  - Threaded program debugging not working prior to Android 2.3
- This is getting a lot better
  - But we aren’t there yet

26

Debugging on Android has been a tale of two cities; Java debugging from Eclipse has been solid for Android really since day one. Native code debugging has been more problematic and longer in coming. At the best of times, remote debugging with gdb/gdbserver is, well, “exciting”. But the Java/Native situation with Android complicates things further. For a while, various basic tool issues caused troubles. Threaded program debugging was not working prior to Android 2.3. This is getting a lot better, and we regularly debug large game engines in Eclipse. But native code debugging is still not at the level to which Windows/PC developers are accustomed.

Slide 27



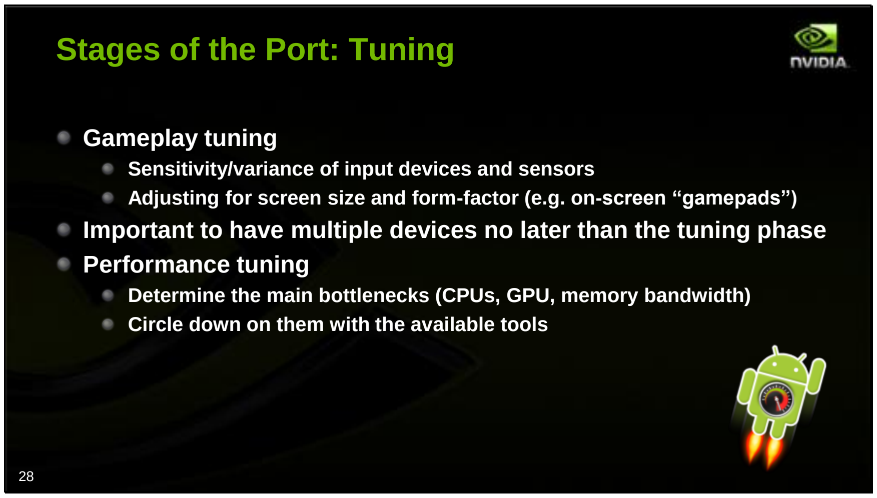
NVIDIA Debug Manager

- Is an Eclipse plug-in that simplifies debugging native C/C++ Android applications
- Seamless Java and Native code debugging
- Supports the latest version of the NDK as soon as possible after each NDK update
- Supported Operating Systems include
  - Windows 7
  - Mac OS X
  - Linux

27

NVIDIA's Debug Manager NVDM is an Eclipse plug-in that simplifies debugging hybrid Java / C++ Android applications on Tegra. Note that this tool works not only with apps that use Eclipse as their build system, but also with applications that are built by custom build systems. It supports seamless Java and Native code debugging in the same project and works on multiple Android OS versions. And because it is based on Eclipse and the Android SDK, it supports debugging from development host PCs that run Windows 7, Mac OS or Linux

Slide 28

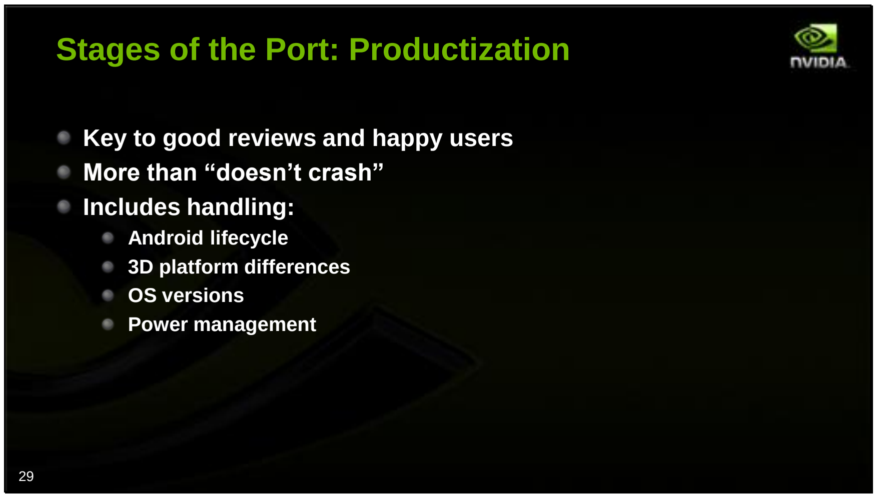


Stages of the Port: Tuning

- Gameplay tuning
  - Sensitivity/variance of input devices and sensors
  - Adjusting for screen size and form-factor (e.g. on-screen “gamepads”)
- Important to have multiple devices no later than the tuning phase
- Performance tuning
  - Determine the main bottlenecks (CPUs, GPU, memory bandwidth)
  - Circle down on them with the available tools

The tuning phase generally involves initial performance optimization and then a mix of performance tuning and user input tuning. Input varies across Android devices. Tuning is often very device-specific, and is generally the first time that developers start testing on more than one device. It can actually be valuable to have a different device type per engineer. Engineers then become natural “advocates” for the device they have to use! They have a vested interest in it not being broken. It can improve the quality of the initial port. The important points of the performance tuning are the same as you’d expect on any platform. Other NVIDIA presentations will cover performance tuning and tools in detail, so we’ll not discuss them in this presentation.

Slide 29



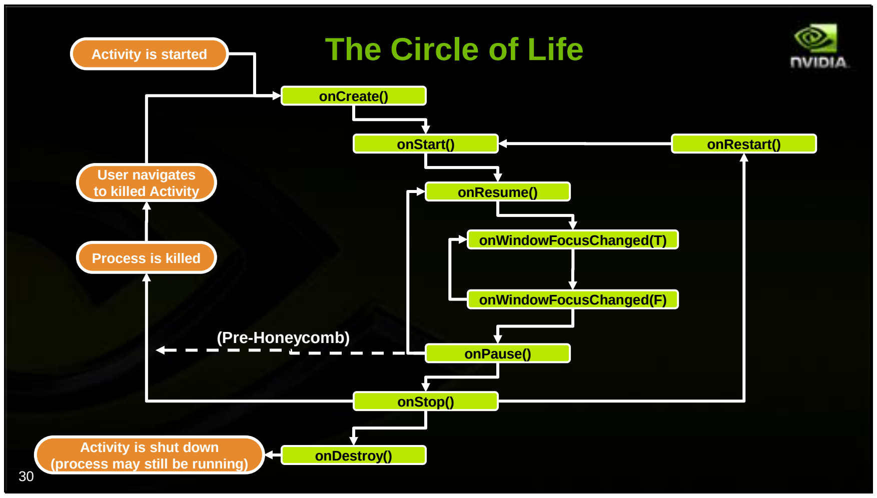
Stages of the Port: Productization

- Key to good reviews and happy users
- More than "doesn't crash"
- Includes handling:
  - Android lifecycle
  - 3D platform differences
  - OS versions
  - Power management

29

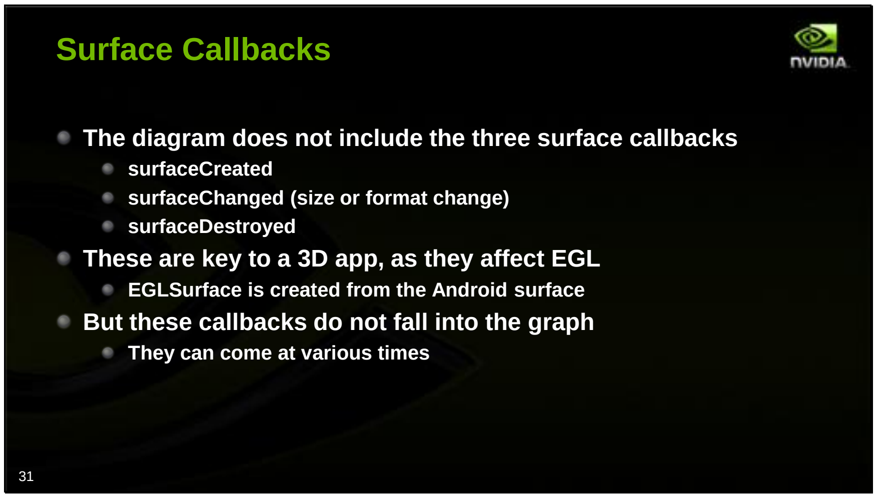
Productization is perhaps the most important stage of your game port because it can make the difference between a 1-star review and a 5-star review, even if the core game is good. Productization may also involve some of the most "foreign" steps for games coming from PC or console, since games on Android are like any other app; they are considered interruptible and lightweight. Note that productization is not just basic stabilization; it includes handling Android's ideas of app-swapping, different 3D HW features and different Android OS variants.

Slide 30



Games on a platform have to conform to the basic application model of the OS. In Android, that's generally known as the Android Activity lifecycle. The Activity lifecycle expects games to be subservient to the overall device's functionalities. The top level structure of this graph is shown on this slide. The orange ovals on the left show external operations or states, and the green rectangles show application callbacks that are called by Android on the app as it moves the app from state to state according to the arrows. Note that while most of the functions in this sequence only get called via the transitions listed, that's not always the case. Over the next few slides, we'll cover common actions the user might take, and how they progress through the graph.

Slide 31



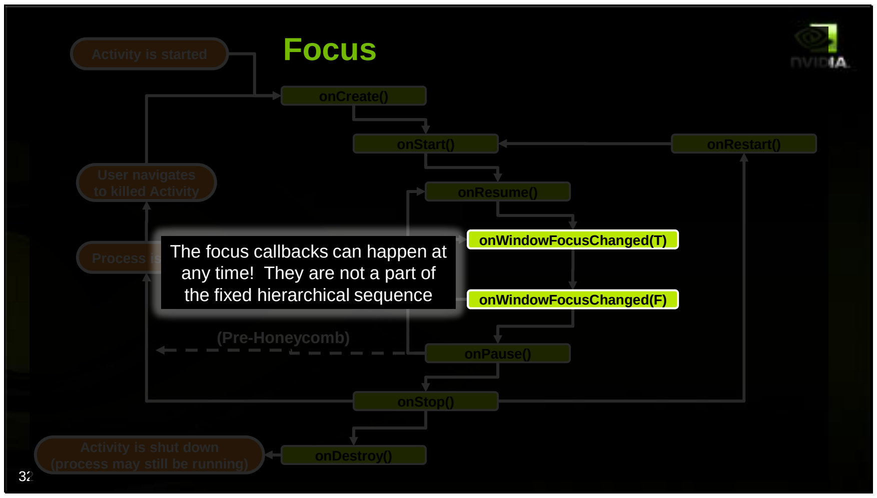
Surface Callbacks

- The diagram does not include the three surface callbacks
  - surfaceCreated
  - surfaceChanged (size or format change)
  - surfaceDestroyed
- These are key to a 3D app, as they affect EGL
  - EGLSurface is created from the Android surface
- But these callbacks do not fall into the graph
  - They can come at various times

31

Despite showing an already complex set of sequences, the previous graph actually leaves out another set of callbacks that are pivotal to application lifecycle, especially 3D applications: the Surface callbacks. The indicate the lifespan of the Android surface to which an application's EGLSurface is attached, and thus imply the lifespan and size changes of the 3D rendering surface. These callbacks are not fixed in the sequence of the previous slide's graph. They can come during a wider range of application states. So applications should be ready to track the state of the surface themselves in the larger context of app state. These surface callbacks themselves cannot make too many assumptions about the application's state. For example, we've seen surfaceDestroyed come after onStop and even after onDestroy.

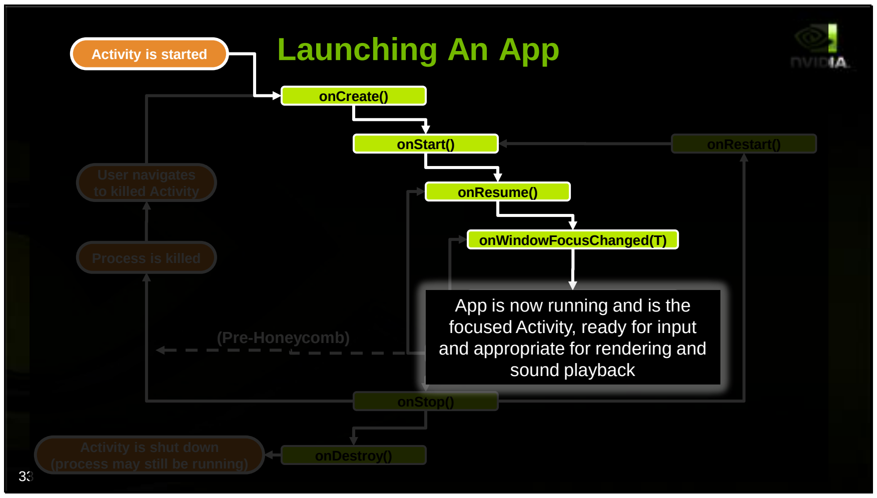
Slide 32


Focus

The focus callbacks can happen at any time! They are not a part of the fixed hierarchical sequence

In addition, while the diagram shows the onWindowFocusChanged callbacks as being a part of the fixed sequence, this is not guaranteed. While this is a COMMON result, this function is not a part of the fixed graph of calling sequences. Window focus changes can be sent to an app that is not between calls to onResume and onPause. The ordering will vary from OS version to OS version. So be ready to catch these cases.

Slide 33

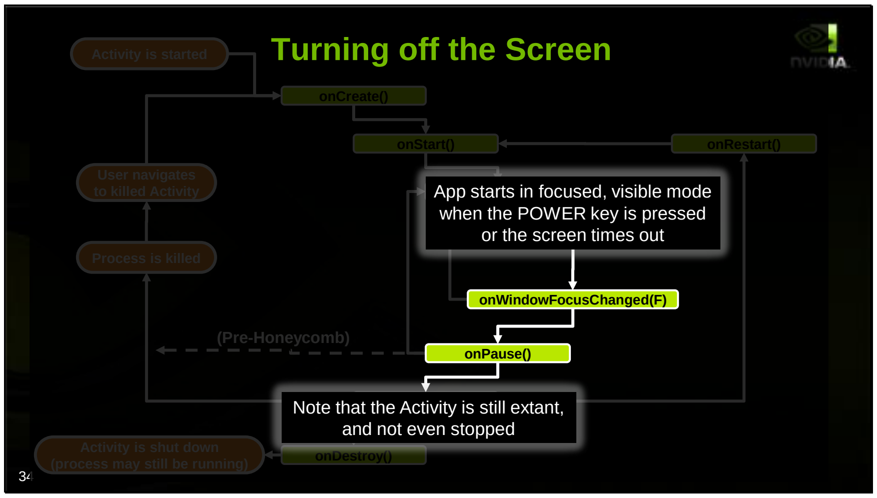


Launching An App

App is now running and is the focused Activity, ready for input and appropriate for rendering and sound playback

When the user launches an app that is not already running, then the Activity's Java class is created, the onCreate function is called on it to initialize the Activity's global state. Then, it transitions to a visible state with a call to onStart. It transitions to being the top-most app with a call to the Activity's onResume. And finally, the app's window is given the focus callback to indicate that the window is not only the top-most app, but is definitely visible to the user. That last distinction will become important in another case down the road. At this point, we should be rendering, playing our sounds, and accepting input.

Slide 34

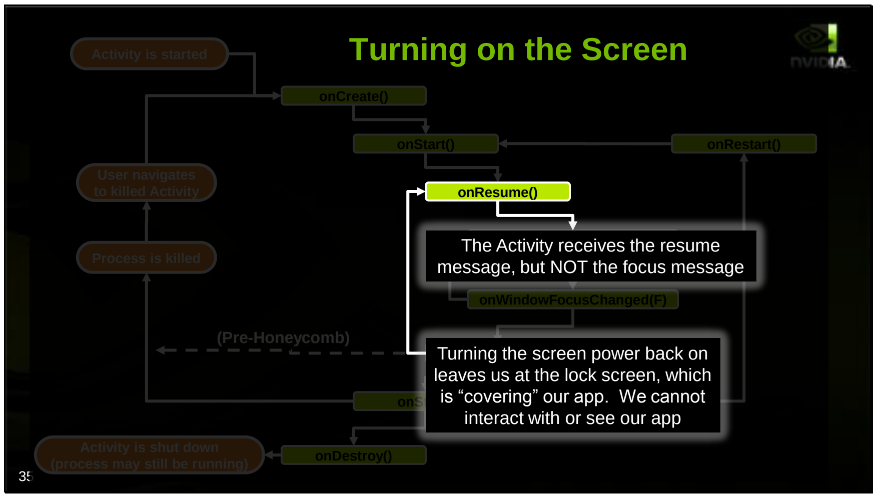

Turning off the Screen

App starts in focused, visible mode when the POWER key is pressed or the screen times out

Note that the Activity is still extant, and not even stopped

If the user taps the power button to turn off the screen, or the screen-saver timeout goes off, we lose our focus, which is an indication to pause our gameplay to some auto-pause screen, start to turn off our sounds and slow or stop our rendering to save battery. But we then get onPause because we're not fully visible. We need to stop rendering and stop doing most processing. Note that how we handle onPause depends on whether we have to run on all versions of Android or only on Honeycomb and newer. On versions of Android prior to Honeycomb, Android was allowed to kill our process silently any time after we return from onPause. So, you had to save all of your app's state before returning from onPause or risk the user losing game progress. On Honeycomb and newer, the activity is not silently killable until it is stopped and the onStop callback returns. So in this case, on Honeycomb, we would not be required to save state.

Slide 35

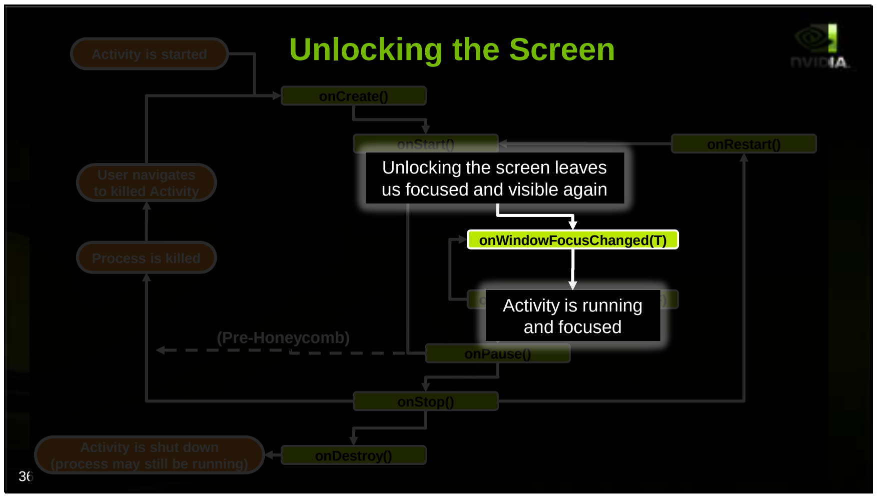

Turning on the Screen

The Activity receives the resume message, but NOT the focus message

Turning the screen power back on leaves us at the lock screen, which is "covering" our app. We cannot interact with or see our app

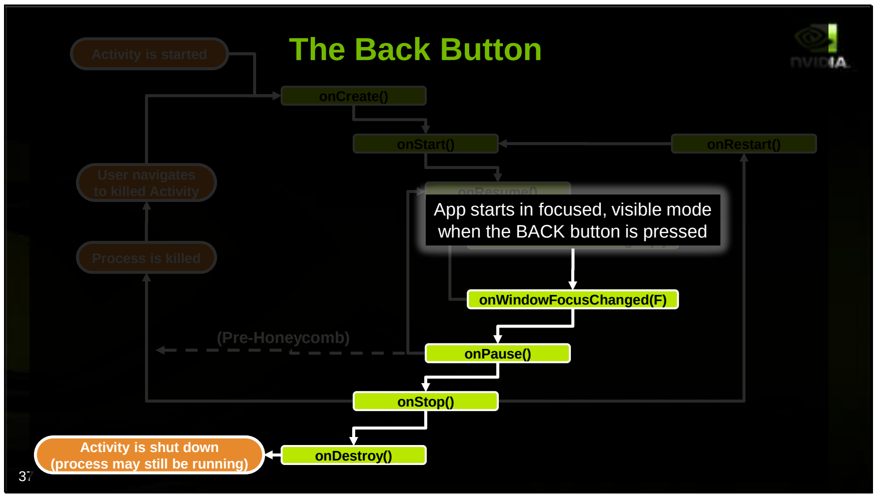
When we turn the screen back on, we get an onResume to indicate that we are the top-most activity. HOWEVER, we are not visible. The lock screen is covering us! So we must not begin playing music or rendering yet. We prepare to render, but we stay dormant for the moment.

Slide 36


Unlocking the Screen
Activity is started
onCreate()
onStart()
onRestart()
Unlocking the screen leaves us focused and visible again
User navigates to killed Activity
onWindowFocusChanged(T)
Process is killed
Activity is running and focused
(Pre-Honeycomb)
onPause()
onStop()
Activity is shut down (process may still be running)
onDestroy()
36

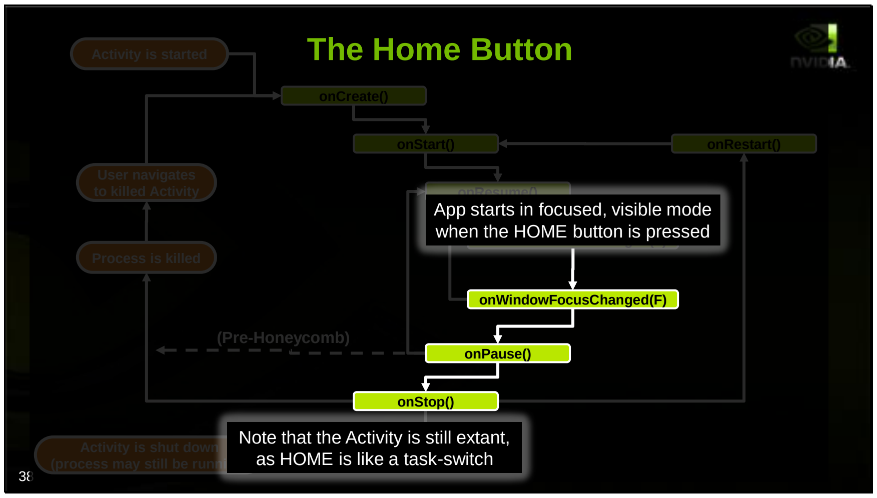
When the user unlocks their screen, we get the window focus change we've been waiting for, and we know that it is time to start rendering and playing sound again. However, we should go back to the game's autopause screen rather than just resuming the game to make sure the user isn't scrambling to catch up.

Slide 37



When your application receives a back-button event, you have two choices; you can handle it yourself and "eat" the event, or you can pass it on to Android. If you pass it to Android, it will "pop" your app's activity off of the UI stack. So, we'll lose focus, we'll get paused and then stopped because we are not visible. Finally, because we've been completely popped from the stack, we'll get our Java class destroyed. Note that to speed re-launch, our process will likely keep running. But our Java class is gone, and our native code had better stopped doing any work at all, because Android could kill the process at will in that state!

Slide 38

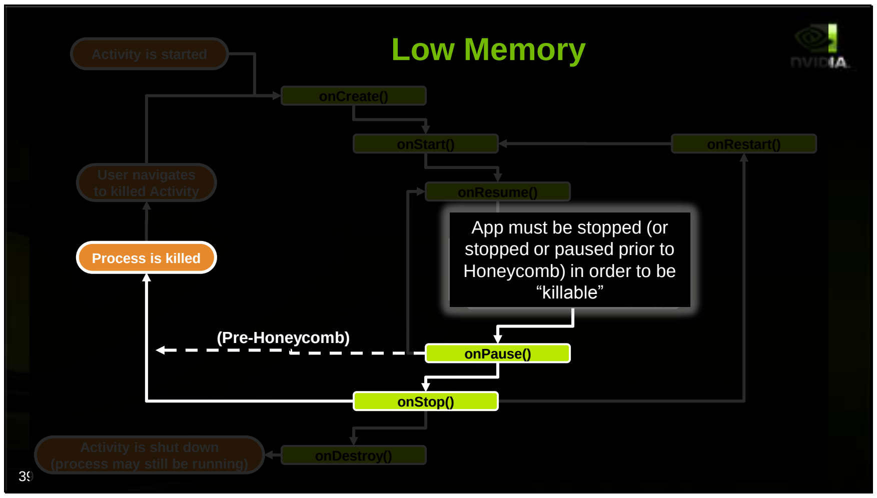

The Home Button

App starts in focused, visible mode when the HOME button is pressed

Note that the Activity is still extant, as HOME is like a task-switch

If we again start from our app being the focused, visible activity and hit the HOME button, we're indicating that we want to task-switch to our home screen to run other apps or swap to another app. We haven't been popped off of the stack, we've just been shuffled down. So, unlike BACK, we do not get destroyed. We're not focused, we're not visible, but our app still has all of its global state. After returning from onStop, we'll be killable, so we need to save game state before returning. We really should not be doing _ANYHTING_ at that point, because in the stopped state, we could get our process killed by Android at any moment

Slide 39

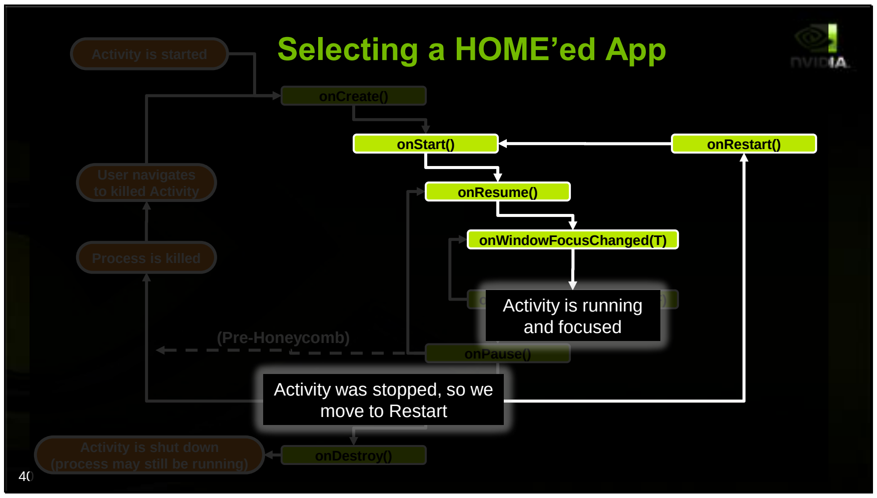


Low Memory

App must be stopped (or stopped or paused prior to Honeycomb) in order to be "killable"

As mentioned, if we've been given the onStop in any version of Android or if we've been given the onPause prior to Honeycomb, we need to save our state to persistent storage before returning from the appropriate callback, or else we could be killed before we have the time to save the user's progress or settings. Now, you may think you can keep yourself from being killable by simply never returning from the appropriate callback. No. If you do not return from those lifecycle callbacks in a some brief period, android will consider you unresponsive, show the user an error dialog and give them the option of killing you. It is called an Application Not Responding or ANR error, and various device makers and carriers consider it a showstopper for a game. While technically you will not get an ANR until you have failed to respond to user input for 5 seconds, in practice, you don't want to be unresponsive for anywhere near that long.

Slide 40

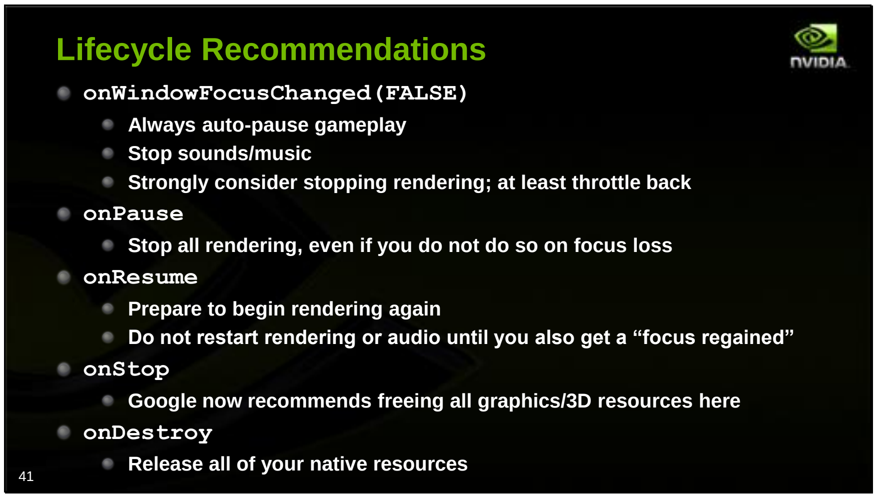


Selecting a HOME'ed App

Activity is started
onCreate()
onStart()
onRestart()
User navigates to killed Activity
onResume()
onWindowFocusChanged(T)
Process is killed
Activity is running and focused
(Pre-Honeycomb)
onPause()
Activity was stopped, so we move to Restart
Activity is shut down (process may still be running)
onDestroy()
40

Now, if we were sent away by the HOME button but then before Android found a need to kill us we got selected from the list of backgrounded apps, we get restarted. Rather than be sent straight to the onStart callback again, we first get our onRestart callback. This lets us deal with the case of restarting from a stopped object, which may be a different setup path than if we are being started from scratch like we were in the launch case.

Slide 41



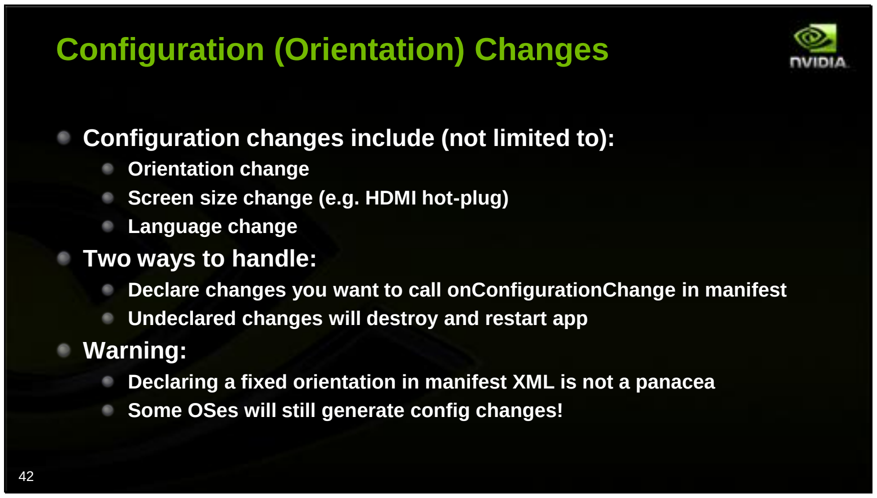
Lifecycle Recommendations

- `onWindowFocusChanged(FALSE)`
  - Always auto-pause gameplay
  - Stop sounds/music
  - Strongly consider stopping rendering; at least throttle back
- `onPause`
  - Stop all rendering, even if you do not do so on focus loss
- `onResume`
  - Prepare to begin rendering again
  - Do not restart rendering or audio until you also get a “focus regained”
- `onStop`
  - Google now recommends freeing all graphics/3D resources here
- `onDestroy`
  - Release all of your native resources

41

I recommend writing an app that prints to the log in each callback and playing for a while – that’s how I generated the previous graphs and compared them to the spec. It is very informative; we’ll be including several such apps in the next Tegra Android Samples Pack. So here are the top-level recommendations for what to do in callbacks. On loss of focus, always auto-pause gameplay and stop sounds. You probably want to stop rendering, but you should at least throttle back. `onPause`, stop all rendering, even if you kept rendering on focus loss. When you get `onResume`, _prepare_ to begin rendering again, but do not restart rendering or audio until you also get a “focus regained”. In `onStop`, Google now recommends freeing all graphics/3D resources. And, of course in `onDestroy` release all of your remaining resources and be sure to stop any native threads you might have left running.

Slide 42



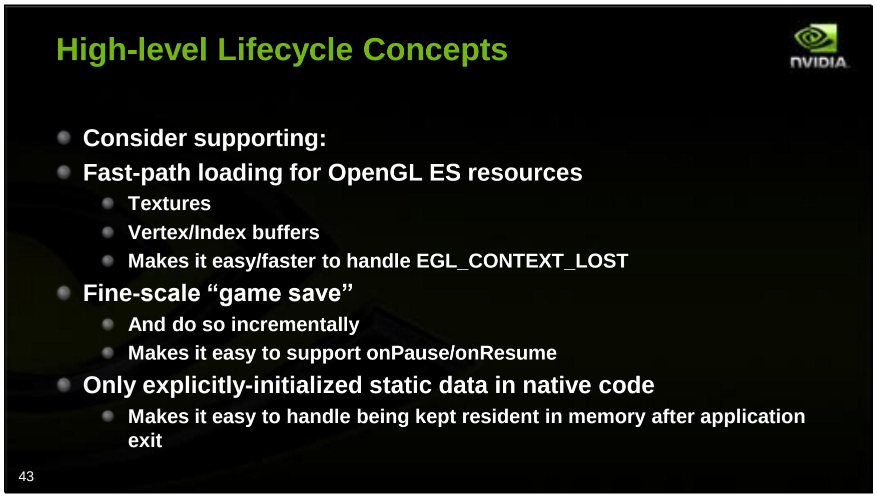
Configuration (Orientation) Changes

- Configuration changes include (not limited to):
  - Orientation change
  - Screen size change (e.g. HDMI hot-plug)
  - Language change
- Two ways to handle:
  - Declare changes you want to call onConfigurationChange in manifest
  - Undeclared changes will destroy and restart app
- Warning:
  - Declaring a fixed orientation in manifest XML is not a panacea
  - Some OSes will still generate config changes!

42

Configuration changes are a source of confusion and bugs in many apps. Android sends them for numerous different events like orientation and screen size change. For each possible configuration change, you can choose to handle it by receiving an onConfigurationChange callback, or you can default. For any change you default, when that change happens, Android will shut down your app completely and relaunch it in the new configuration. This can be expensive for big games. The most common configuration change by far is device rotation between portrait and landscape. Android includes the ability to request that your app be given a fixed orientation, but note that this does not ensure that you will never see an orientation config change. We see them sent to fixed-orientation apps on some devices at startup or at wakeup from sleep. So test your app with multiple devices and be sure to try launching your landscape-forced app with the device in portrait mode. You don’t want to crash.

Slide 43



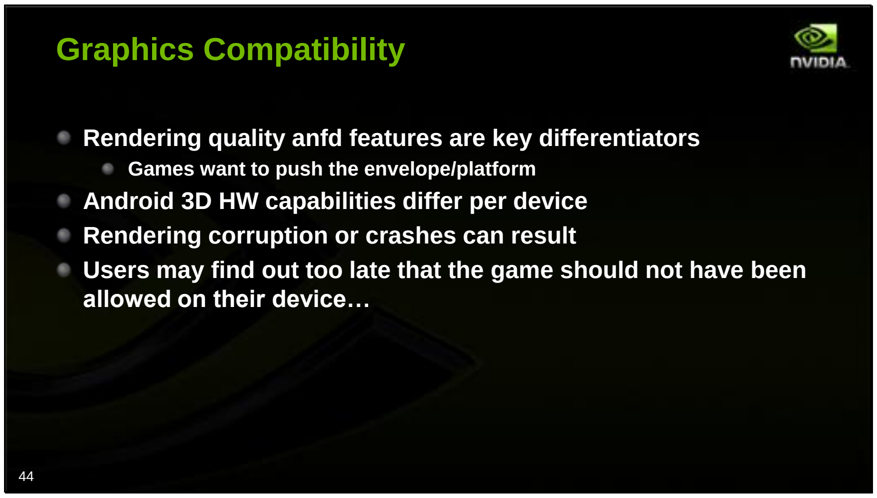
High-level Lifecycle Concepts

- Consider supporting:
- Fast-path loading for OpenGL ES resources
  - Textures
  - Vertex/Index buffers
  - Makes it easy/faster to handle EGL_CONTEXT_LOST
- Fine-scale "game save"
  - And do so incrementally
  - Makes it easy to support onPause/onResume
- Only explicitly-initialized static data in native code
  - Makes it easy to handle being kept resident in memory after application exit

43

There are some other things you can do to make the lifecycle support cleaner. If possible, make it easy to reload OpenGL ES resources like textures and buffer objects without restarting the engine from scratch. Try to support fine-grained "game save" rather than widely-spaced "save points". If swapped out or killed, games should pick up right where the user left off rather than some distant save point. In a perfect world, the user shouldn't know when they navigate back to a backgrounded game whether the app was restarted or merely resumed. Finally, avoid implicitly-initialized static data in your code. This is a coding detail that really matters: Your native library proably won't be unloaded and reloaded every time the game is exited and restarted, so your implicit static data initializers may not be called when you expect them to be. Initialize all data explicitly in onCreate.

Slide 44



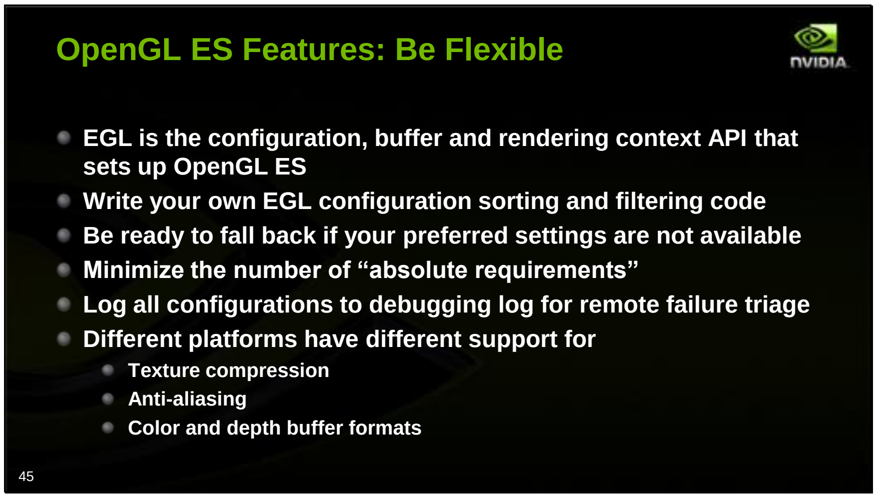
Graphics Compatibility

- Rendering quality anfd features are key differentiators
  - Games want to push the envelope/platform
- Android 3D HW capabilities differ per device
- Rendering corruption or crashes can result
- Users may find out too late that the game should not have been allowed on their device…

44

Graphics rendering is a key feature in high-end games; Games want to push every feature they can to stand out in the market. But Android HW platforms differ quite a lot with respect to 3D features. Graphics features and renderer setup are common reasons for apps to fail on a given device. This can lead to unhappy customers, sspecially if the issue arises only after they buy the game and wait for the app's side-loaded content to download. It gets even worse if that issue happens only after the Android Marketplace's 15 minute refund timeout has expired.

Slide 45



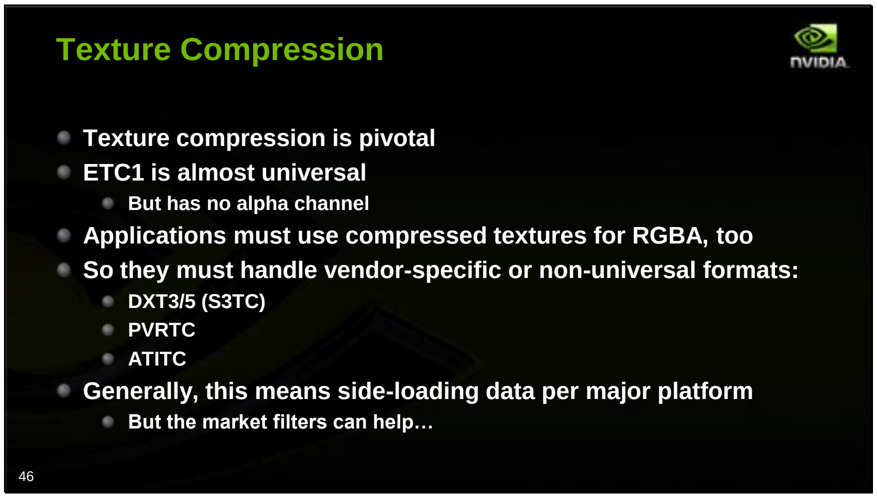
OpenGL ES Features: Be Flexible

- EGL is the configuration, buffer and rendering context API that sets up OpenGL ES
- Write your own EGL configuration sorting and filtering code
- Be ready to fall back if your preferred settings are not available
- Minimize the number of “absolute requirements”
- Log all configurations to debugging log for remote failure triage
- Different platforms have different support for
  - Texture compression
  - Anti-aliasing
  - Color and depth buffer formats

45

When setting up your application’s rendering, ask for what you want, but be ready to fall back if you can’t find a perfect match. You need to select the best feature and buffer configuration for your app ***as available on the device***. We recommend writing your own, custom algorithm for finding the best config. Existing, black-box filtering code may fall back to software rendering or return no matching configs at all. Keep in mind that different Android hardware platforms have different support for texture compression, antialiasing, and color and depth buffer formats. Test on as wide a variety of hardware as possible. If possible, include a debug mode that logs the available configs to stdout. That way, if a beta tester has a problem on a new device, they can send you a log of all configurations, so you can debug why your filtering failed.

Slide 46



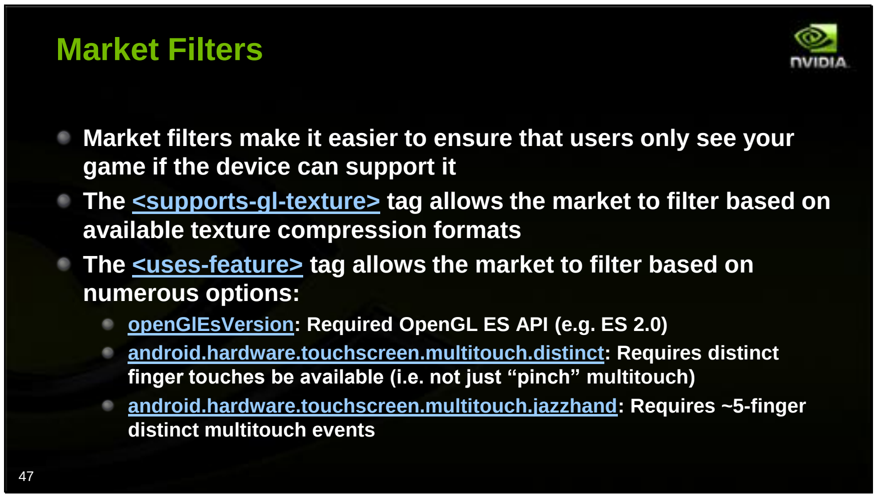
Texture Compression

- Texture compression is pivotal
- ETC1 is almost universal
  - But has no alpha channel
- Applications must use compressed textures for RGBA, too
- So they must handle vendor-specific or non-universal formats:
  - DXT3/5 (S3TC)
  - PVRTC
  - ATITC
- Generally, this means side-loading data per major platform
  - But the market filters can help…

46

Texture compression is pivotal for high-quality mobile games owing to the limited memory and memory bandwidth on these devices. But there is no universally supported compressed format with alpha. So, applications wishing to maximize their market must handle multiple vendor-specific or non-universal formats. Generally, this means either multiple versions of the game or a single version that side-loads data per platform on first the run of the app. You can improve user experience on this front by giving the Android market information in your manifest about texture formats and other features you require…

Slide 47



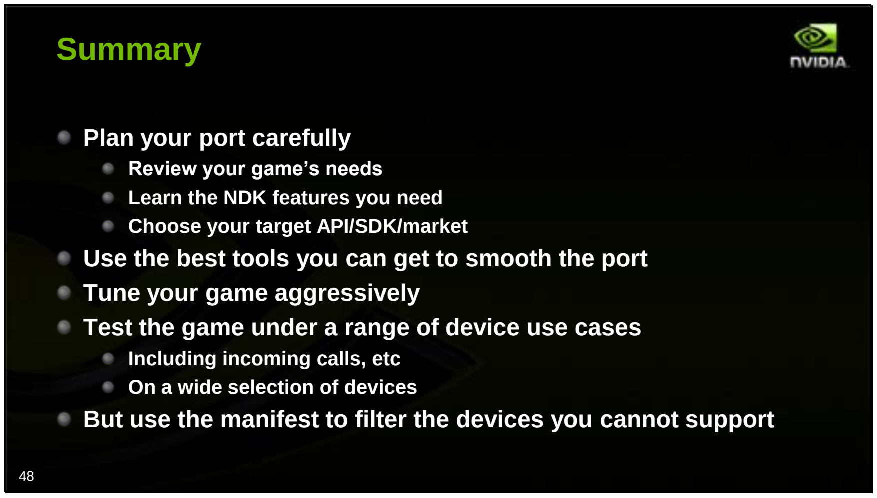
Market Filters

- Market filters make it easier to ensure that users only see your game if the device can support it
- The <supports-gl-texture> tag allows the market to filter based on available texture compression formats
- The <uses-feature> tag allows the market to filter based on numerous options:
  - openGlEsVersion: Required OpenGL ES API (e.g. ES 2.0)
  - android.hardware.touchscreen.multitouch.distinct: Requires distinct finger touches be available (i.e. not just "pinch" multitouch)
  - android.hardware.touchscreen.multitouch.jazzhand: Requires ~5-finger distinct multitouch events

47

The Android Market supports filtering of apps based on tags in your manifest file. They just added app filters based on supported texture compression formats. These can ensure that your game appears only on devices that will run it and run it well. There are a wide range of other filters that can ensure that users do not download your game to a device that will not run it; I list a few key ones here. This can avoid the dreaded 1-star flame reviews. See the android developer site for details of the supported filters.

Slide 48



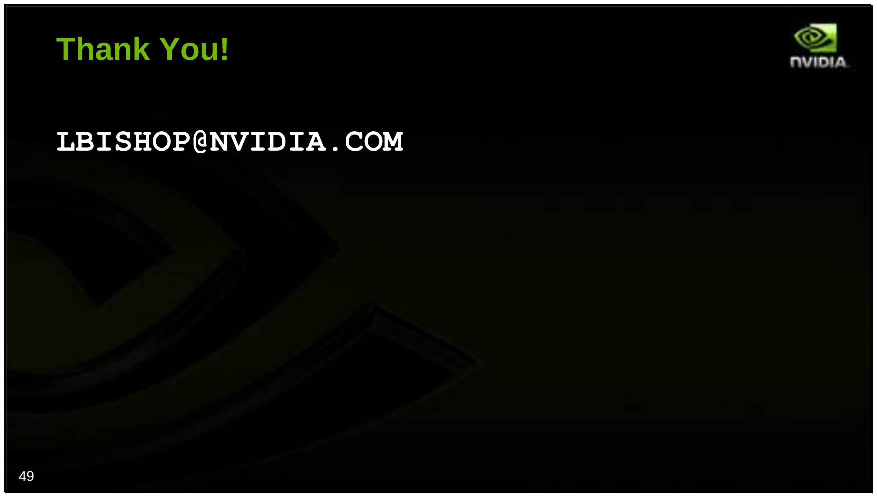
Summary

- Plan your port carefully
  - Review your game's needs
  - Learn the NDK features you need
  - Choose your target API/SDK/market
- Use the best tools you can get to smooth the port
- Tune your game aggressively
- Test the game under a range of device use cases
  - Including incoming calls, etc
  - On a wide selection of devices
- But use the manifest to filter the devices you cannot support

48

In summary, it is important to consider all stages of the port when you plan it. Planning ahead for things like your targetted OS version and for Android lifecycle issues can save time and hassle later. Use the best tools at your disposal for renderer porting, debugging and tuning. Aggressively tune for performance when you're active, and for battery when you're not. And test on every device you can, testing both how the game plays and how it behaves in the larger context of device use. Finally, be sure to try to indicate accurately in you manifest what devices you can and can't support.

Slide 49



Thank You!

LBISHOP@NVIDIA.COM