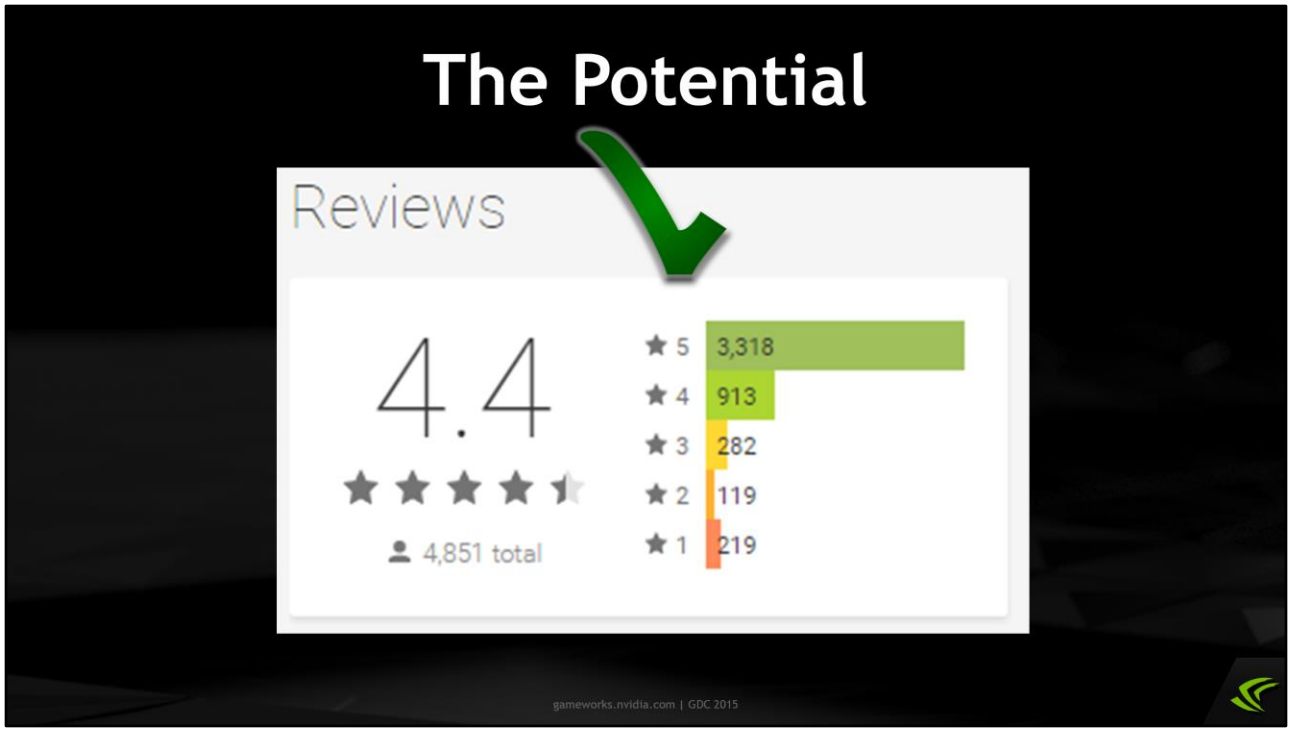
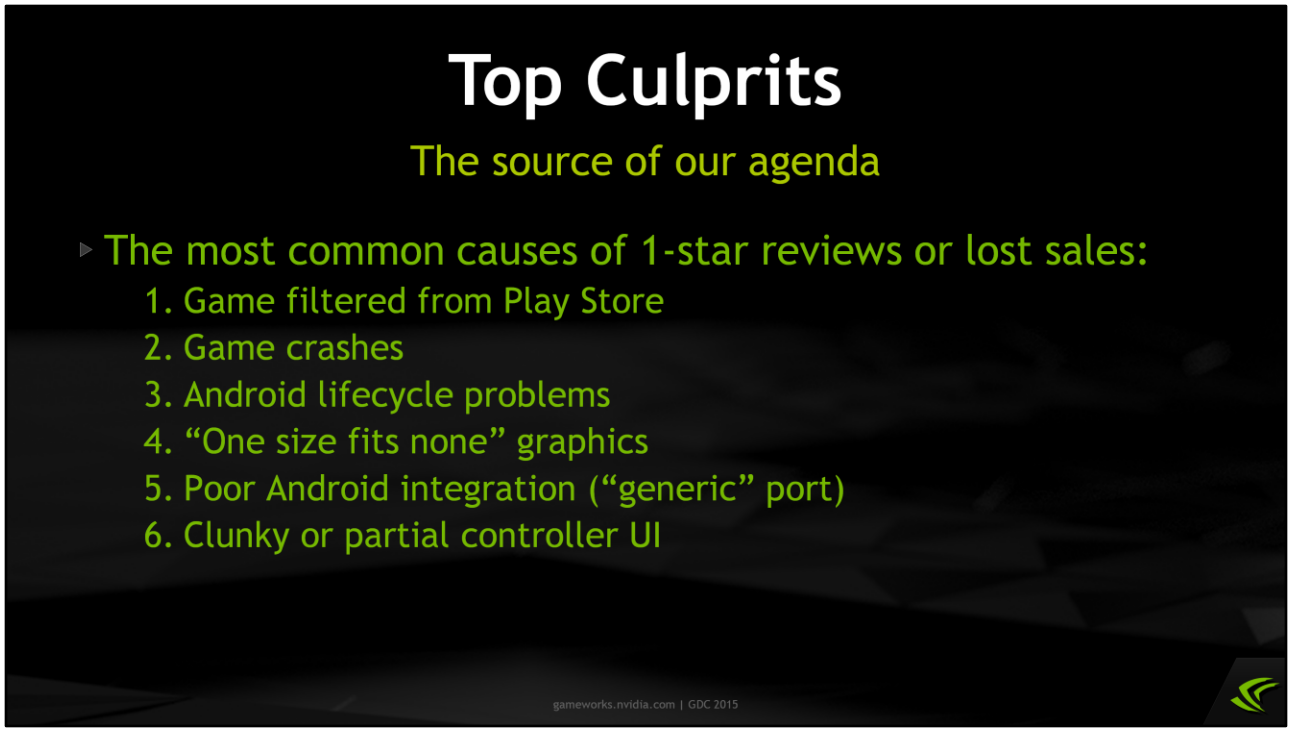
Good morning; My name is Lars Bishop, and I'm a developer technologies engineer on the Tegra team at NVIDIA. I've been at NVIDIA just about ten years, and I've spent my entire time there supporting mobile developers on a number of platforms. For the past few years, my focus has been Android, from drivers to OS core, to the app level. Joining me on-stage today is Krispy Uccello; he's a Developer Advocate at Google who focuses on Gaming. And today, we'd like to spend this session covering a mixture of specific engineering recommendations for Android and Android TV game development, based on our experiences supporting commercial game developers.



A great game has every chance to be a great gaming experience on Android, getting positive reviews and big sales. Android is a really powerful, diverse gaming platform. This is the potential for any game on Android. But our experience has also shown us some heartbreakers – great games that ***could*** have been 5-star experiences on Android, but got hammered with 1-star reviews for things that could have been avoided. They're the not-so-little things that snatch defeat from the jaws of victory.

This isn't about major changes to the way you develop your games. This is about things you can add to your already-working development pipeline that are specific to SHIELD, to Android TV, and Android. So, the official name of the session is 'building 5-star games for SHIELD and Android TV'. To be honest, this wasn't the name that we the engineers wanted. We wanted to call it 'Techniques for Avoiding Negative Reviews on NVIDIA SHIELD and Android TV'. But marketing said we couldn't have a negative-sounding title. Now, we have some pretty important reasons for wanting to call it 'Avoiding negative reviews' – we've worked with enough game developers to know that **we can't** make a game be a 5-star game. That's up to the developers, the designers and the market. It would be foolish and frankly arrogant to think we could do otherwise. 5-star games come from developers like those represented here at the conference.



So what are some of these biggest, top-level items that we see that causing issues? Well, games not even appearing in the market for a user’s device; getting filtered out is a sort of “silent killer” for sales.

Games crashing on a users’ devices is the “NOT so silent killer”. Poor handling of suspend and resume and app switching is a common complaint. Unsatisfying, “old” visuals on new Android systems can lead to vocal user disappointment. Lack of Android feature integration is a sign of a “generic” or “turn the crank” port and it definitely turns gamers off. Finally, clunky or partial controller UI is a major issue for Android TV and console-style gaming on all devices. We’ll discuss these and a few more.

So why does NVIDIA engage game developers on overall Android gaming and TV experiences? Well, the answer is, of course, SHIELD. From the original SHIELD portable, a purpose-built Android gaming handheld, to the gaming-focused SHIELD tablet and its bundled SHIELD controller, all the way to the just-announced SHIELD, the Android TV-based living room gaming experience, NVIDIA has been focusing on Android gaming for several years.

Looking at these platforms, I think you can see why we care about android compatibility and features support. We've created a handheld, a tablet, and an Android TV entertainment system, all powered by NVIDIA's Tegra processors. And all of them focused on a great gaming experience.

However, because of the wide range of form factors here, our recommendations in this talk will apply to games on a wide range of Android and Android TV devices, SHIELD or not.

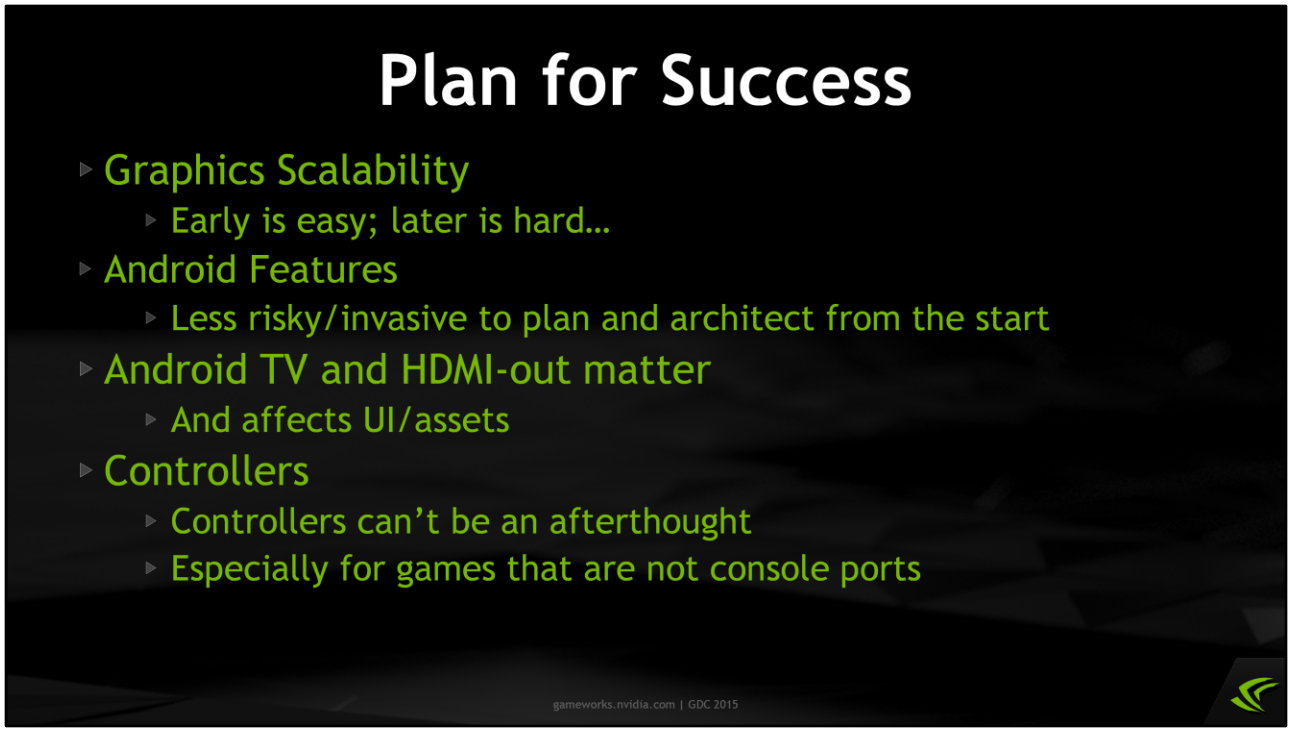
Running Example: Oddworld

A great example of an Android value-added port done well

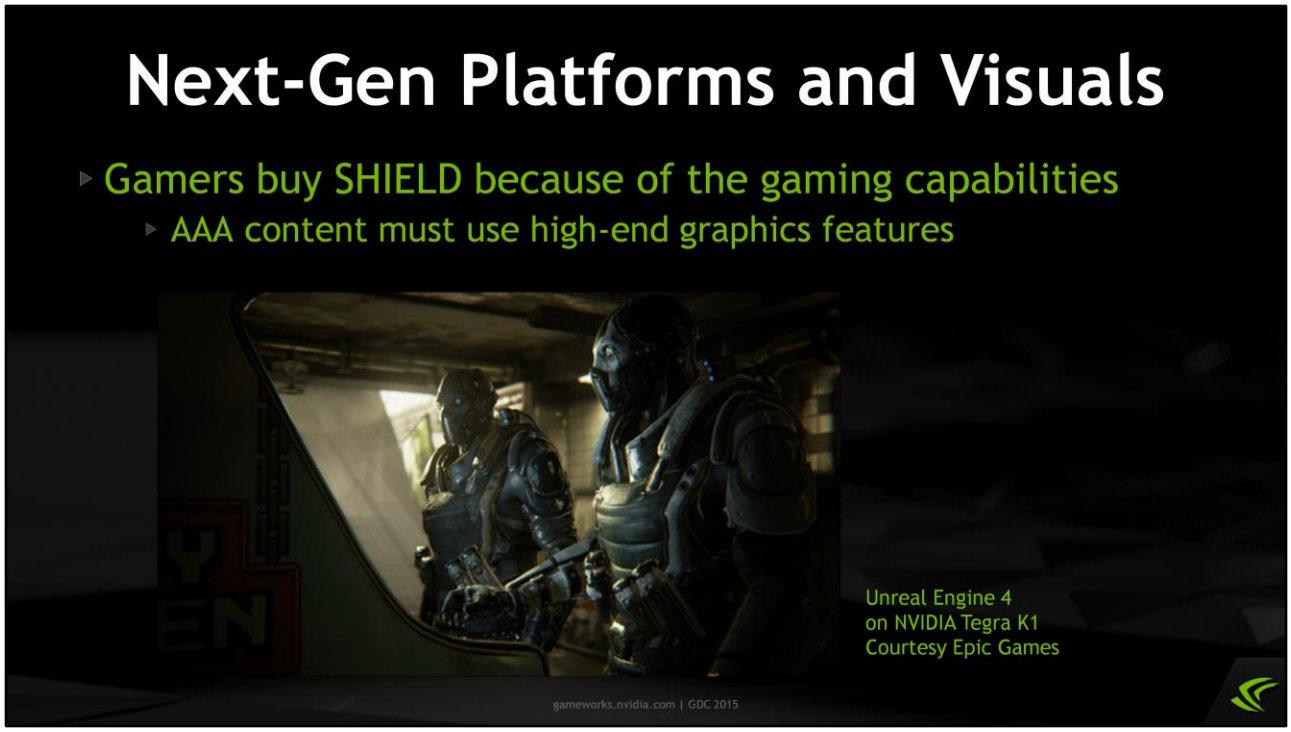
- Oddworld: Stranger's Wrath
  - Critically acclaimed 2005 Action/Adventure title
- Android developed/supported by Square One Games
  - http://www.s1games.com/
- Android TV, tablet and phone support on day 1
- Runs well on four generations of Tegra!
  - And a wide range of other devices

Throughout the talk, we'll use some examples from an Android title that we think really did things the right way. Oddworld: Stranger's Wrath. The game likely needs no introduction; it's a great episode of a very popular franchise. The studio that did the Android version is Square One Games in West Vancouver, run by Stephane Jacoby and Roger Freddi. In full disclosure, I've supported Square One and Stephane on several titles, and I'm a big fan; they're a joy to support. They do their titles well. Oddworld had game controller support, Google Play integration and Android TV support all at launch. And, as you may have guessed, the well-reviewed game on slide two was *Oddworld*. They really hit all of the items we'll discuss today.

We're going to break down the issues not just by category, but by stage of the common game development pipeline. Often, we end up engaging game developers who already in alpha testing. And that late in the process, we find that some of our recommendations are much harder to integrate than they would have been early in the development cycle.



Here are the four biggest items that we see getting pushed late in the production pipeline that can really benefit from earlier planning on Android. Android devices in the market have a wide range of graphical horsepower; graphics scalability should be planned out from the start; it can be a big pain to have to tune the game at the end without enough knobs at hand. Android features like in-app payment can be added much more easily if you architect to include them. Android TV and HDMI out affect not only gameplay but UI assets. And controller support needs to be considered early, especially if the game is entirely new or the original game did not support consoles. So, now the details.



Be sure you set the graphics “bar” high enough at the high end. Gamers buy SHIELD platforms for gaming, and they know what their systems are capable of doing. So triple-A mobile content needs to take advantage of that. As an example, see the screenshot here – this is U E 4 running on NVIDIA’s SHIELD tablet, using Google’s OpenGL ES 3.1 Android Extension Pack.

Android Extension Pack (AEP)

One extension; many features

- GLES 3.1 AEP adds great 3D features under one roof:
- https://www.khronos.org/registry/gles/extensions/ANDROID/ANDROID_extension_pack_es31a.txt
- https://developer.nvidia.com/astc-texture-compression-for-game-assets

- Tessellation and Geometry shaders
- ASTC texture compression
- Shader atomics
- Per-sample shading and variables
- Per-render target blending modes

Google's G L E S Android Extension Pack, or A E P adds a whole group of desktop or console-level 3D features, including: Tessellation and Geometry shaders for tunable geometry amplification. ASTC adds a cross-platform texture compression format with alpha support and variable bitrates. Atomic variables are supported in shaders for advanced compute shaders. Per-sample shading and shader variables are available for complex AA. And independent blending modes per render target are supported for effects like deferred rendering.

Using the Android Extension Pack

- Batched set of next-gen extensions
  - Defines an OpenGL ES “profile” to target
  - Has a dedicated Play Store filter:

```
<uses-feature android:name=“android.hardware.opengles.aep” android:required="true" />
```

- Session TODAY, in this room:
  - 15:00-15:30
  - Cutting Edge Graphics for Android
  - Mathias Schott, DevTech Engineer

gameworks.nvidia.com | GDC 2015

The great thing about A E P is that it’s a single extension with all of these features. So it defines a sort of “Open GL feature level” that a developer can target, rather than having to plan for some combinatorial explosion of missing or supported extensions. And to make it even easier to use on Android, you can filter on it in the Google Play Store by adding a feature tag to your app’s manifest. Add this, and you know that any platform where you’ll be installed will have all of these features!

For a ton more details on the possibilities unlocked by A E P, there is a session by NVIDIA’s own Mathias Schott at 3 P M today, in this room.

And a quick demo of a few A E P features running on the new NVIDIA SHIELD.

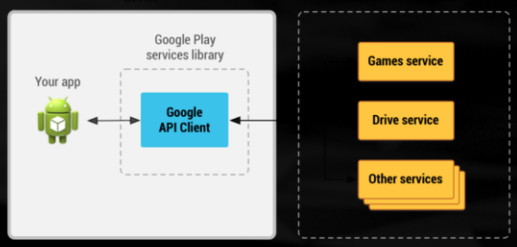
So much for the high bar… As early as you can, figure out your best optional and tunable features and expose these in user settings. As we know, gamers like the option to adjust these things. However, to ensure the best chance of having the game "just work" out of the box, if possible your app should either do some form of micro-benchmark on the first launch or else use some form of device detection and pre-set the options for optimal frame rate. Oddworld does this, and it works pretty well out of the box. Some good scalable items are particles, shadows, and image-based post-processing passes. If you're using a commercial engine, this is likely available AND scriptable. See the options for U E 4 and Unity listed here.

In terms of resolution, all of your rendering should be scalable. This is important not only because you'll need to handle a large range of screen resolutions, but also because it may be necessary to render below the device's native resolution to hit your desired frame rate. Most high-end games already do a post-process "resolve" pass for their post-processing filters. In this common architecture, the 3D scene is rendered to an offscreen buffer, and then a final 2D rendering pass is done for effects like motion blur and depth-of-field. This already decouples the main rendering resolution (the offscreen buffer) from the screen resolution, that makes it easy to let the user adjust the pre-resolve resolution. Smart default options are particularly important for this. The major rendering engines make this easy, too – see the references. The engines can also scale the 3D rendering while doing the UI at full res for quality.

Also, screen variance is a *design* concern! Android gaming device screens go from less than five inches diagonal to more than sixty inches for Android TV. So pay attention to screen size information and consider a different UI or at least UI assets for different systems.

Java is not the Enemy

- Important Android integration features are Java-only
- External Displays
- License Verification
- Monetization
  - In-app payments (IAP)
  - Adverts



One question I often get from folks starting out in Android game development is “how do I avoid writing any Java?”. Depending on the developer, this may be out of performance concerns, a desire to avoid complicating their development, or in rare cases, just fear, uncertainty and doubt. I tend to try to redirect this question – what I think we *should* be thinking of is “how do we create the best Android gaming experience while keeping the app as simple, small and stable as possible”. The fact is that numerous, important Android integration features like externa;l displays, In App Payment and advertisements are Java-only. So some Java is going to be required; the key is to make the *integration* of Java and C++ as clean and tight as possible. Used well, explicit Java code plus a LITTLE J N I (or Java Native Integration) C code can help make a great Android game.

Using Java and C++

NativeActivity and NativeAppGlue are key

- **NativeActivity/AndroidNativeAppGlue** the most common setup
  - Event-loop-based
- **NativeAppGlue** is not platform code
  - Make your own copy; learn it, extend it
- **NativeActivity** can be subclassed in Java
  - The best of both worlds

The most common way that Android apps stay native is to use NativeActivity. Actually, what most apps use is a combo of NativeActivity and AndroidNativeAppGlue. Together, these interfaces allow an Android application to run their main loop in C++, in their own thread and receive input and system information via an event loop. One important thing that many developers do not realize is that unlike NativeActivity, which is implemented in the OS itself, NativeAppGlue is not platform code. It's fully open, app-level support code. It's *designed* to be branched into the app and extended. So, you *should* make your own copy of it; learn it, and extend it for what you need. We've done this with our samples for things like rendering at the refresh rate.

In addition, while NativeActivity itself is a part of the OS, a developer should *subclass* the NativeActivity object in Java for their game. Initially, this subclass can be completely empty. This is the best of both worlds; you can add only the Java you need, when you need it.

By basing your app off of a NativeActivity subclass and your own copy of the C AndroidNativeAppGlue code, you're ready for anything.

Down the road, you can add things like J N I messaging for in-app purchases, or passing just the data you need from native to Java for cloud saves. You can even directly leverage Android UI dialog boxes and pre-built Google UI for things like Google Play sign-in. All with isolated Java code, very little J N I code and extremely few changes to your native code.

Oddworld used this subclassing method with great success. And as we mentioned, NVIDIA's own GameWorks OpenGL Samples include examples of a minimal subclassing of NativeActivity

If your game started as a touch-only mobile or PC game, you'll likely need to add some schedule time for creating a controller UI. *Don't* try to get away with controller-driven mouse pointer – you should code for the best platform experience.

Also, plan extra time to test the *full* game experience on controller only. We still see pre-release games with partial controller support, especially deep in the settings menus. ANY required touch interaction is a showstopper on Android TV. If possible, test on the common controllers; SHIELD controller, Android TV gamepad and Android TV remote if you can support it

Develop effectively; use good tools

- Tegra Android Development Pack
  - One-stop installer for Android development
  - Includes GameWorks Samples
  - Includes NVIDIA developer tools
- Nsight Tegra Visual Studio Edition
  - Seamless Android development integration inside of MS Visual Studio
  - Free!
- Session TODAY, in this room:
  - 17:30-18:30
  - Android Dev-Diaries with EA Firemonkeys and Tick Tock Games
  - Daniel Horowitz, NVIDIA

Another key in the planning phase is to plan the build, debugging, and profiling of your application. To make this easier, NVIDIA ships the Tegra Android Development Pack. It's designed to be a one-stop installer for Android build tools, NVIDIA's advanced CPU / GPU debugging and profiling tools, and Android-related sample code. Key among these tools is N-sight Tegra Visual Studio Edition. It's a free Microsoft Visual Studio plugin that adds full Android development and debugging within Microsoft's I D E. You can even include Windows and Android builds in the same project.

In addition, we have a session TODAY at five thirty, in this room. Daniel Horowitz from our developer tools team will present a session with E A Firemonkeys and Tick Tock games on using NVIDIA tools for Android development.

Resources

- NVIDIA GameWorks
  - https://developer.nvidia.com/gameworks
  - Tools, samples, documentation, whitepapers

- GameWorks OpenGL Samples
  - Support GLES 2.0 through GLES 3.1 AEP and even OpenGL 4.3
  - Single-source samples for Android, Linux and Windows
  - Supports SHIELD / Android TV ***TODAY***

- Google Android Developer Site
  - https://developer.android.com/
  - http://android-developers.blogspot.com/

There are a lot of resources available to you: NVIDIA has created GameWorks, with tools, samples, documentation, and whitepapers for Android game development. The GameWorks OpenGL Samples specifically show a wide range of OpenGL and G L E S features, including A E P. The samples run from a single-source base on Android, Linux and Windows. And they support SHIELD and Android TV ***TODAY***

While many developers use the Google Android Developer Site for reference docs, the guides to development and publishing on the site are great, too. As are the Google Android developer blogs with articles from Krispy and others covering common development questions in detail.

Finally, plan on working with partners for success… We at NVIDIA can help amplify your team. Square One Games have worked with NVIDIA on several titles. NVIDIA DevTech have provided testing and profiling feedback well in advance of release. For example, NVIDIA QA tested their games on SHIELD before SHIELD was even announced. And in return, working with Sqaure One on pre-release devices helped us understand where there were issues in our own drivers. So it has helped improve the SHIELD as well.

Next, let’s cover some items that don’t tend to be an issue until development itself.

Android 5.0 “Lollipop”

Common pitfalls

- **Pros:** Lollipop adds great new features and Android TV
- **Challenges:** Many existing applications had to be fixed
  - NVIDIA worked with scores of developers to help them with this transition
- Almost **all** of these issues fell into one of a few cases
- https://developer.nvidia.com/content/why-does-my-app-crash-android-50-lollipop

Android Lollipop added great new features to Android, and most importantly, it added support for Android TV. However, it’s also been the first Android update in years that required fixes to a lot of existing applications. For reasons of security and performance, Android L is stricter than previous versions. We worked with lots of developers to help them with this transition, and we learned a bunch doing it. We found that almost all of these issues fell into one of a few cases.

ART is the new Java runtime in Android, and over half of the crashes in existing applications were related to ART being tighter to spec. Specifically, JNI checks are tighter in ART than they were in Dalvik. So apps that got away with “JNI warnings” before would throw an exception on Android L. Also, ART shares its thread stacks between native and Java. So apps that used to create native threads with explicit stack sizes suddenly found these stacks were no longer deep enough on L.

One of our recommendations to assist with these JNI exceptions are to *avoid* doing complex operations in JNI itself. Instead, write the code in Java (in your NativeActivity subclass) and make a single JNI call up to that code. In general, prefer making one JNI call up to Java over dozens of JNI calls to do the same work without explicit Java. Basically, JNI code is fragile and JNI calls can be slow. We find that what you could do in five Lines of Java can take 50 lines of C++-based JNI. So while you often COULD avoid most explicit Java code by writing a TON of J-N-I C++ code, this is not the best option.

JNI vs JNI+Java: Launch Browser

Pure C++ JNI

```
jstring urlText = mApp->appThreadEnv->NewStringUTF(urlString);
EXCEPTION_RETURN(mApp->appThreadEnv);
jclass uriClass = mApp->appThreadEnv->FindClass("android/net/Uri");
EXCEPTION_RETURN(mApp->appThreadEnv);
jmethodID uriParse = mApp->appThreadEnv->GetStaticMethodID(uriClass, "parse",
"(Ljava/lang/String;)Landroid/net/Uri;");
EXCEPTION_RETURN(mApp->appThreadEnv);
jobject uri = mApp->appThreadEnv->CallStaticObjectMethod(uriClass, uriParse, urlText);
EXCEPTION_RETURN(mApp->appThreadEnv);
jclass intentClass = mApp->appThreadEnv->FindClass("android/content/Intent");
EXCEPTION_RETURN(mApp->appThreadEnv);
jfieldID ACTION_VIEW_id = mApp->appThreadEnv->GetStaticFieldID(intentClass, "ACTION_VIEW",
"Ljava/lang/String;");
EXCEPTION_RETURN(mApp->appThreadEnv);
jobject ACTION_VIEW = mApp->appThreadEnv->GetStaticObjectField(intentClass, ACTION_VIEW_id);
EXCEPTION_RETURN(mApp->appThreadEnv);
jmethodID newIntent = mApp->appThreadEnv->GetMethodID(intentClass, "<init>",
"()V");
EXCEPTION_RETURN(mApp->appThreadEnv);
jobject intent = mApp->appThreadEnv->AllocObject(intentClass);
EXCEPTION_RETURN(mApp->appThreadEnv);
mApp->appThreadEnv->CallVoidMethod(intent, newIntent);
EXCEPTION_RETURN(mApp->appThreadEnv);
jmethodID setURI = mApp->appThreadEnv->GetMethodID(intentClass, "setData",
"(Landroid/net/Uri;)Landroid/content/Intent;");
EXCEPTION_RETURN(mApp->appThreadEnv);
mApp->appThreadEnv->CallObjectMethod(intent, setURI, uri);
EXCEPTION_RETURN(mApp->appThreadEnv);
jmethodID setAction = mApp->appThreadEnv->GetMethodID(intentClass, "setAction",
"(Ljava/lang/String;)Landroid/content/Intent;");
EXCEPTION_RETURN(mApp->appThreadEnv);
mApp->appThreadEnv->CallObjectMethod(intent, setAction, ACTION_VIEW);
EXCEPTION_RETURN(mApp->appThreadEnv);
jclass activityClass = mApp->appThreadEnv->FindClass("android/app/Activity");
EXCEPTION_RETURN(mApp->appThreadEnv);
jmethodID startActivity = mApp->appThreadEnv->GetMethodID(activityClass,
"startActivity", "(Landroid/content/Intent;)V");
EXCEPTION_RETURN(mApp->appThreadEnv);
mApp->appThreadEnv->CallVoidMethod(mApp->appThreadThis, startActivity, intent);
EXCEPTION_RETURN(mApp->appThreadEnv);
```

C++ JNI (for calling up to Java)

```
jstring jniText = mApp->appThreadEnv->NewStringUTF(url);
EXCEPTION_RETURN(mApp->appThreadEnv);
jclass thisClass = mApp->appThreadEnv->GetObjectClass(mApp->appThreadThis);
EXCEPTION_RETURN(mApp->appThreadEnv);
jmethodID launchURL = mApp->appThreadEnv->GetMethodID(thisClass,
"launchURL", "(Ljava/lang/String;)V");
EXCEPTION_RETURN(mApp->appThreadEnv);
mApp->appThreadEnv->CallVoidMethod(mApp->appThreadThis, launchURL, jniText);
EXCEPTION_RETURN(mApp->appThreadEnv);
```

Java

```
public void launchURL(String urlString)
{
    Uri uri = Uri.parse(urlString);
    Intent launchBrowser = new Intent(Intent.ACTION_VIEW, uri);
    startActivity(launchBrowser);
}
```


Here's a quick example of launching a web browser to a given URL from native code. On the left it is done purely in native code calling JNI. 34 lines of C++ code. On the right, we write most of it in 6 lines of explicit Java, and then call that Java with 9 lines of native JNI code. And this is, obviously, a simple example. The difference between all native JNI and just using a little native JNI to call explicit Java code can be arbitrarily large. We prefer the simpler version on the right.

Common Snags

Some changes to previous behaviors

- Implicit Service Intents
  - E.g. LVL license-checking code
- NuPlayer (Audio player)
  - New, stricter AwesomePlayer replacement
  - Settings -> Developer Options -> Media -> Use NuPlayer (uncheck)
- `clock()` changes
  - Use `clock_gettime(CLOCK_MONOTONIC, &now)` for timing
- Don't forget about 3rd-party code!
  - Especially if it is closed-source

Another security change in Android L affected more advanced application that used services for things like content download or License-checking. In L, services in your app that launch another system via an intent must specify the package that will handle the Intent. Previously, many apps left out the package and relied on the system's package manager to implicitly resolve it to an app for them. You have to name the receiving app explicitly now for security. For audio, NuPlayer is the new Android audio playback system. It tends to be stricter and throws errors in more cases than AwesomePlayer did. To test if NuPlayer is related to an issue in your app, you can revert to AwesomePlayer via the developer options control panel as shown here. Calls to **`clock()`** may now return non-linear results, as apps were relying on non-spec behavior. So just use the replacement shown here for timing instead. And for *all* of these Android L recommendations, don't forget to consider issues with L in your 3rd-party code. This is especially pivotal to consider early in the process if the library is closed-source and you cannot control your own destiny.

Lifecycle

Housekeeping: Life-or-death edition

- Android games have gotten much better about lifecycle
- Still two common showstoppers

- Audio/Music:
  - Watch for focus-loss and kill audio
  - `AudioManager.OnAudioFocusChangeListener` is handy

http://android-developers.blogspot.com/2013/08/respecting-audio-focus.html

- Background CPU load
  - Don't render or burn CPU when you are paused/not-focused

Application Lifecycle. Looking back on my old GDC slides, I used to have to present 20 minutes of my hour-long Android sessions on it. But thankfully, developers are doing a much better job handling suspend, resume, and app switching. But there are still two common, major issues in a lot of Android games

The first is playing Audio or Music on the user's lock screen. So the user is playing a game, and then suspends their device for a meeting. During the meeting, they resume their device to look at the time on the lockscreen, and "boom" out comes loud game music. It's important to kill your app's audio, especially music when you are not the focused window. The Android Audio Focus Change system is handy here. And Krispy has an detailed blog posting on this at the location below.

The second remaining issue is background CPU load. When you are paused and even when you are not focused, be sure to drop your CPU utilization to as near to zero as possible. Look at for background threads in your app, and be sure to block them from spinning when the device is suspended or you aren't focused.

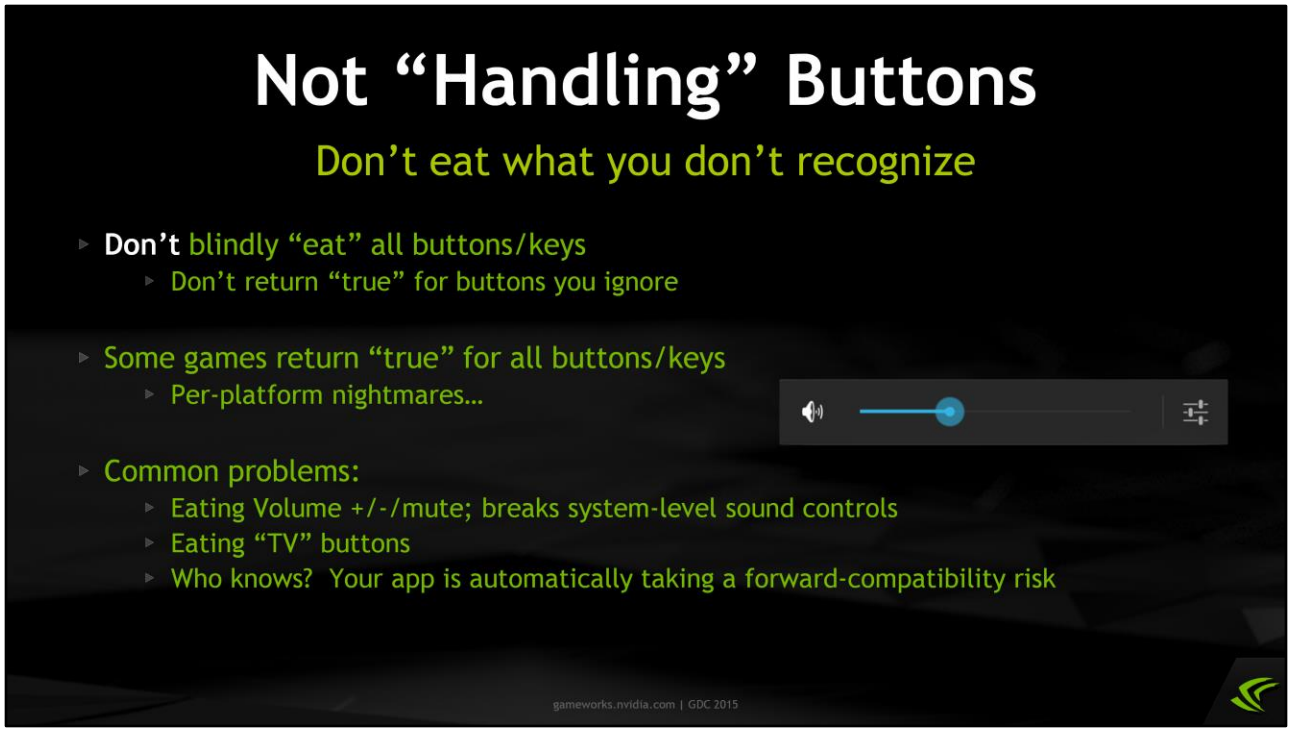
Handling Buttons

How to maximize compatibility

- Return “true” from the callbacks for buttons you recognize and handle
  - “Fallbacks” differ on controllers
- E.g. **BUTTON_B** falls back to **BUTTON_BACK**
- Explicitly handle the **BUTTON_BACK** to avoid confusion

Button handling can be a little confusing; we still see apps that have problems here. With all input callbacks or events, you want to return “true” (or handled) for buttons you recognize and process. Failing to do so can result in your app getting *another* callback for the same button press - a so-called “fallback”. As a specific example, on many systems with a game controller, an unhandled B button event will fall back to a back button event. Say the user presses the B button. Maybe your app steps back one menu level on B. Then, say your app returns “false” from the B event. Well, then the system assumes you did not know about the B button and “falls back”; it sends a separate BACK BUTTON callback. And your app likely steps back a second menu level, leaving the user confused. If you had returned “true” when you handled the B button event, you’d have never gotten the second BACK event, and all would be fine.

Also, you should always explicitly handle the BACK button to avoid confusion. If you don’t explicitly handle the back button, then ANY press of the back button will exit your app instantly.



Now, while returning “false” for every event is a problem, returning “true” for every event can be an even bigger problem. We find that some apps decide to “fix” the “doubled events” issue with a blanket “true”, implicitly saying they handle all events. The problem here is that in some cases you may be eating events that you *should* have let fall back to the system. Only return “true” for buttons you actually handle. With very few exceptions, input events go to the app FIRST. You handle them and the system does not get them. As an example, one common problem stemming from returning “true” on all buttons is that you can break the behavior of the volume rocker while your app is active, because the system never gets a chance to see those volume events. Also, you may be eating buttons you’ve never even heard of, or didn’t even exist when you shipped your app. That makes your app fragile over time. Eat only the events you handle.

Touch and other Controls

Design touch controls carefully

- Avoid touch-points right at the bottom of the screen
  - Especially swiping!
- Use immersive fullscreen mode
- Accelerometer-based controls are controversial
  - **Always** make them a user option
  - Oddworld has aim-with-gyro in touch mode; can be turned off in settings

Game touch controls can be problematic and can interact with Android UI elements. If possible, use immersive fullscreen mode in Android to hide the Navigation Bar. For example, the common “dual touch” first person shooter controls at the bottom corners of the screen can be a problem. We recommend avoiding touch-points right at the bottom of the screen. This is especially true if the touch points in question involve swiping; swiping up from bottom could deploy the navigation bar and mess up the game!

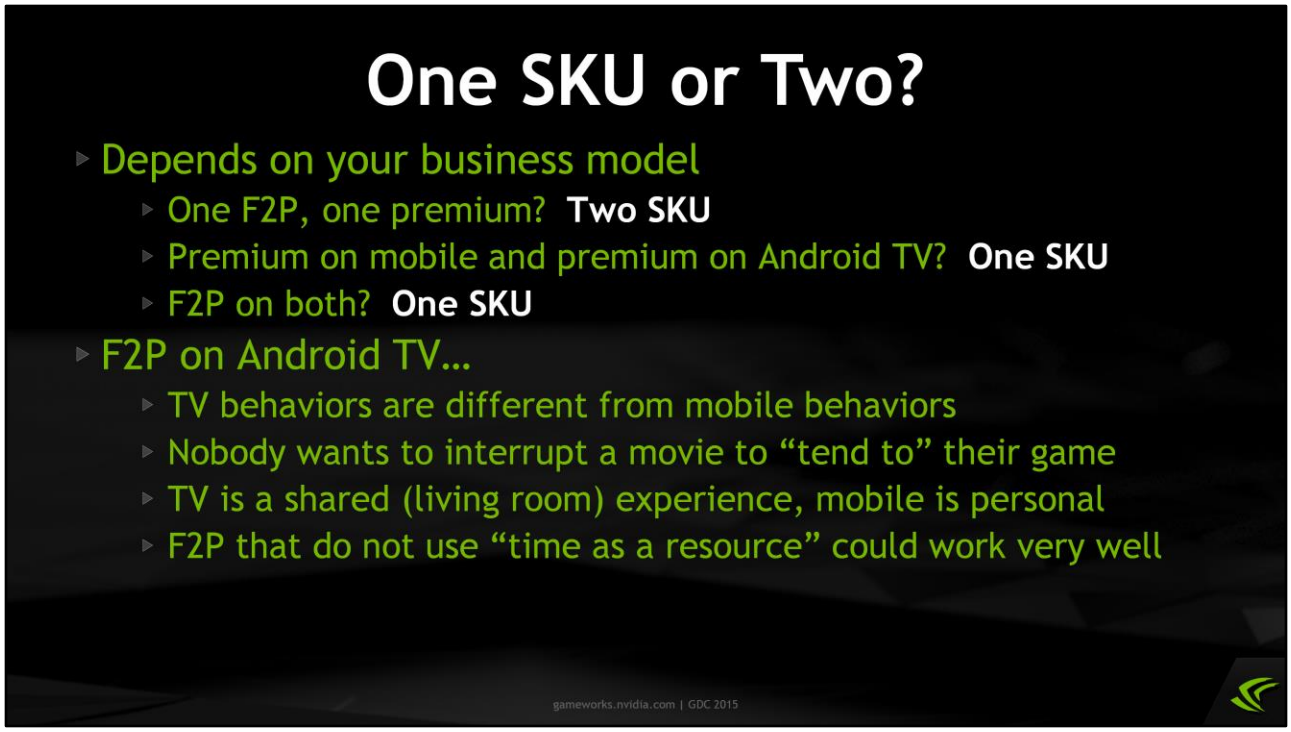
Also, accelerometer-based controls are controversial. Some people hate them, and they can feel uneven on some devices. Always make orientation-based controls a user option even if gyro hardware is available.

Keep in mind the set of axes and buttons that are likely to be available, and design your controller support accordingly. Not every game can support the Android TV remote, but if your gameplay allows it, why not? If your game can be played enjoyably with the remote, you open up your market to non-core gamers.

You may want to consider supporting controller hot-plugging, even if you're not a multiplayer game. Users may have more than one controller, or at least more than one input device. They may have launched your game from the Android TV remote, and then picked up their game controller. The game controller could come up post-launch as a new device, and you could miss it if you aren't watching for hot-plug. Also, you may want to handle hot-unplug of a controller, say in case of battery dying, at least by pausing the game…

Here are some specific examples of controllers. On the left, the SHIELD controller, with its full set of analog and DPAD controls, cross buttons, shoulder buttons, triggers, and "menu" button. It even has a headset plug for two-way audio. The standard Android TV gamepad is similar, but without a couple of the buttons, and the two-way audio. Finally, an Android TV remote like the SHIELD remote has a DPAD navigation, media controls and voice search.

So this covers the items through the development phase. So all that is left is to package and ship. Krispy, what are the important items for developers to consider when packaging and finalizing their game for Android, especially Android TV?

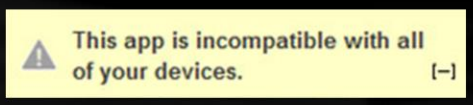


When targeting both Android TV and handheld devices, it isn’t always clear that the same monetization SKU works for both. Free to play might work very well for handheld platforms, but Android TV is more of a premium experience. So consider a full, premium, paid version of the game for Android TV. This requires a different SKU for the two platforms, but can be well worth it in terms of maximizing experience for the user and value to you the developer.

Manifest Destiny

Make sure Play Store customers see your game

- The app manifest determines who can see your app in the Play Store
- The app manifest is your responsibility
  - Even if you're using middleware
  - Know how to customize your manifest
- What might be missing or wrong?
  - Minimum supported OS version
  - Required device features
  - Requested permissions
  - Android TV resources



http://developer.android.com/guide/topics/manifest/manifest-intro.html

The Android Manifest file is the Google Play Store's main interface to your game. It exports a ton of information about your game to Google Play. Get it wrong, and you'll: Not even show up on a compatible device's Play Store and lose potential sales. Or, you'll show up as available on an incompatible device and end up with angry users when the app crashes on their device

The key components of the manifest include the minimum supported OS via the SDK version, the required and requested features (and what those imply), the requested permissions and the Android TV-required resources.

One note for users of middleware game engines: Using an engine does *not* let you ignore this manifest. You should know what your engine's generated manifest looks like! And you should know how to override that generated manifest. For example, we've found that some engines declare all apps as Android TV ready without regard to the app.

Android TV Items

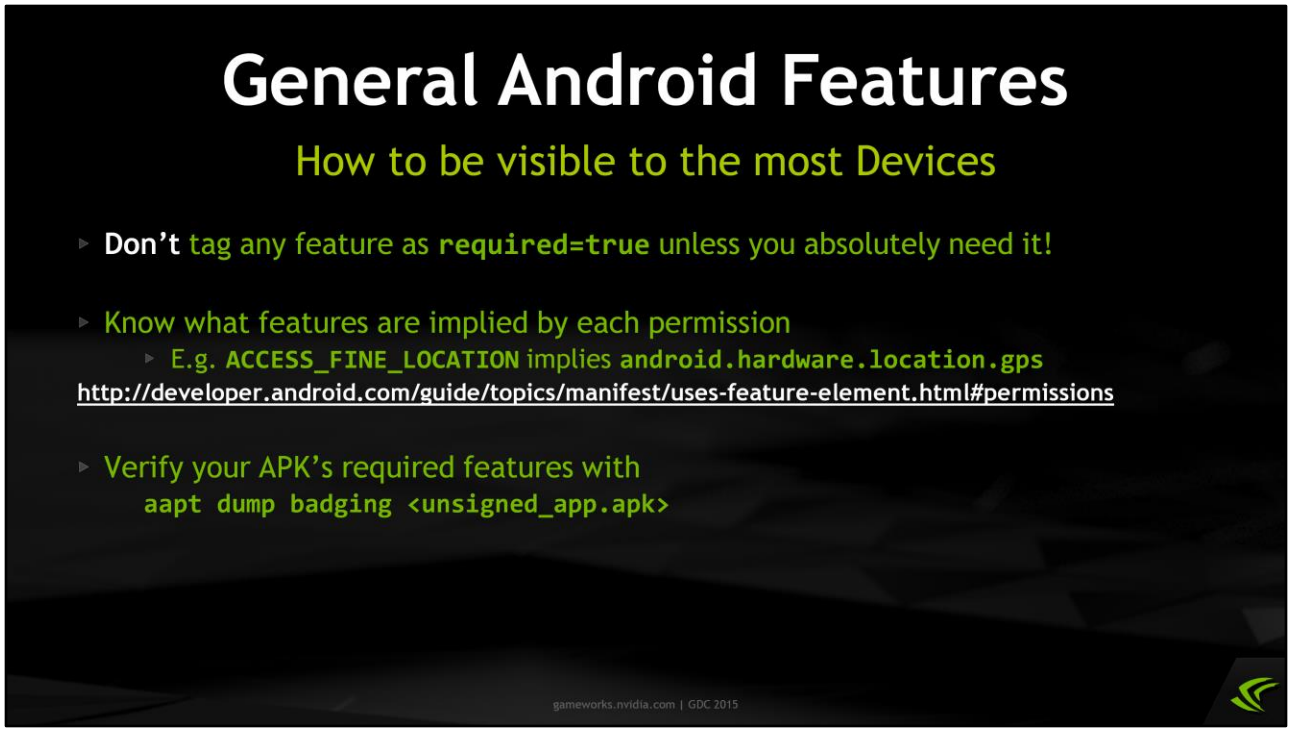
How to be visible on Android TV's Store

- Target API-21 or newer (min can be lower)
- "Leanback Launcher" support
  - Activity to launch on Android TV
  - Banner image for the Leanback Launcher

- What does Android TV require?
  - TV Activities run in landscape
  - Explicitly declare touchscreen not required
  - Declare gamepad feature (only if gamepad needed - not for "simple remote")
  - Don't require non-TV features, e.g. camera, GPS, telephony, accelerometer
  - Know your middleware!

https://developer.nvidia.com/android-tv-developer-guide

In order to be visible on the Android TV Play store, there are a few things you need to do above and beyond general Play Store items. Your app needs to target android SDK level 21 or newer. Note that this does NOT mean Android L is the min spec for your app in general – it can be much lower. You need to prepare for the "Leanback Launcher" so you appear on the Android TV home screen. You do this by designating the Activity that should be launched for Android TV. And you need a Banner image to represent your app in the Launcher. Furthermore, you need to ensure all Activities can run in landscape orientation. Touchscreen support must be declared as optional, not required. If you need a full gamepad, declare a gamepad as required. And don't require camera, GPS or other non-Android TV hardware features. And once again, don't forget any UI generated by third-party code you depend upon, like ad-ware middleware.



In terms of maximizing the number of devices you can support, don't request any feature with **`required equals true`** unless you absolutely need it to run! And for each permission you request, know if any hardware features are implied by that permission. The link below lists these.

To better understand what your manifest implies, you can test for your required features (both explicit and implicit) with the A A P T tool in the Android SDK.

Note that the Google Play Store's publishing dashboard automatically advises you on Android TV compatibility

Earn Your Users' Trust

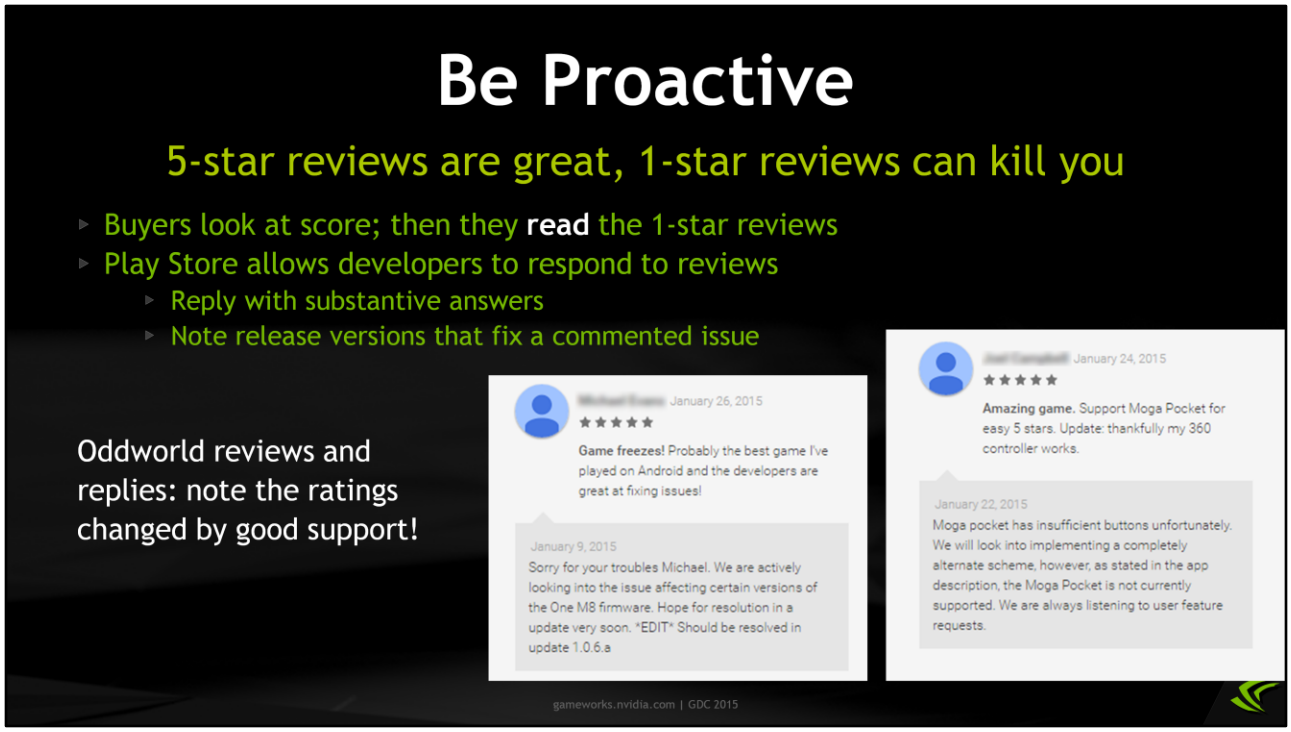
Users read permission requests

- Google's "featured games" editors read them, too
- Require only the permissions you need
- Explain why you need each permission
  - Do it right in the Google Play app description!

Users do *not* like seeing long lists of permission requests when installing apps.

Speaking of earning your users' trust, there's important work to do after your game is shipped. Lars?

Finally, let's discuss a little about what can help your game succeed after you release it upon a grateful world…



Let's face it – people look at games in the Play Store not that differently from how they look at buying anything online. They may see the average rating, but they really read the 1-star reviews. They want to know if these are real problems, or just people looking to complain about *something*. So, those 1-star reviews are the ones that you'll end up spending time with. The Play Store allows developers to respond to reviews, and replying with real, substantive answers shows you want to make your game experience better. "Substantive" is the key here. Try to track down and fix the issue if it's a real bug, and be sure to comment in the reviews that you believe you've fixed the issue in a given version. Look at these Oddworld reviews. These previously-low reviews got responses and were then edited by the reviewers based on great support. On the left, note what happened. A user on a specific handset reported an issue and gave a poor review. Rather than waiting for a groundswell, Square One tends to hit these issues very early. Reproducing and root-causing can take time, so triaging quickly is important. In this case, Square One found the issue and immediately noted publicly that there was an issue and that they were working on it. Then, when they fixed the issue and shipped the fix, they posted again and noted the exact version with the fix. And boom; the user edits the review. Not only is the review 5 stars now, the user notes it is because the developer is responsive.

Even if the issue can't be fixed exactly the way the user wants, Square one tends to respond with the background behind the issue. This gives potential buyers more

confidence.

Device Coverage

Shipping on lots of devices can be a challenge

- Many of your users use devices you do not have and will never see
- Plan for this
- Make it easier to get useful debugging info from your customers

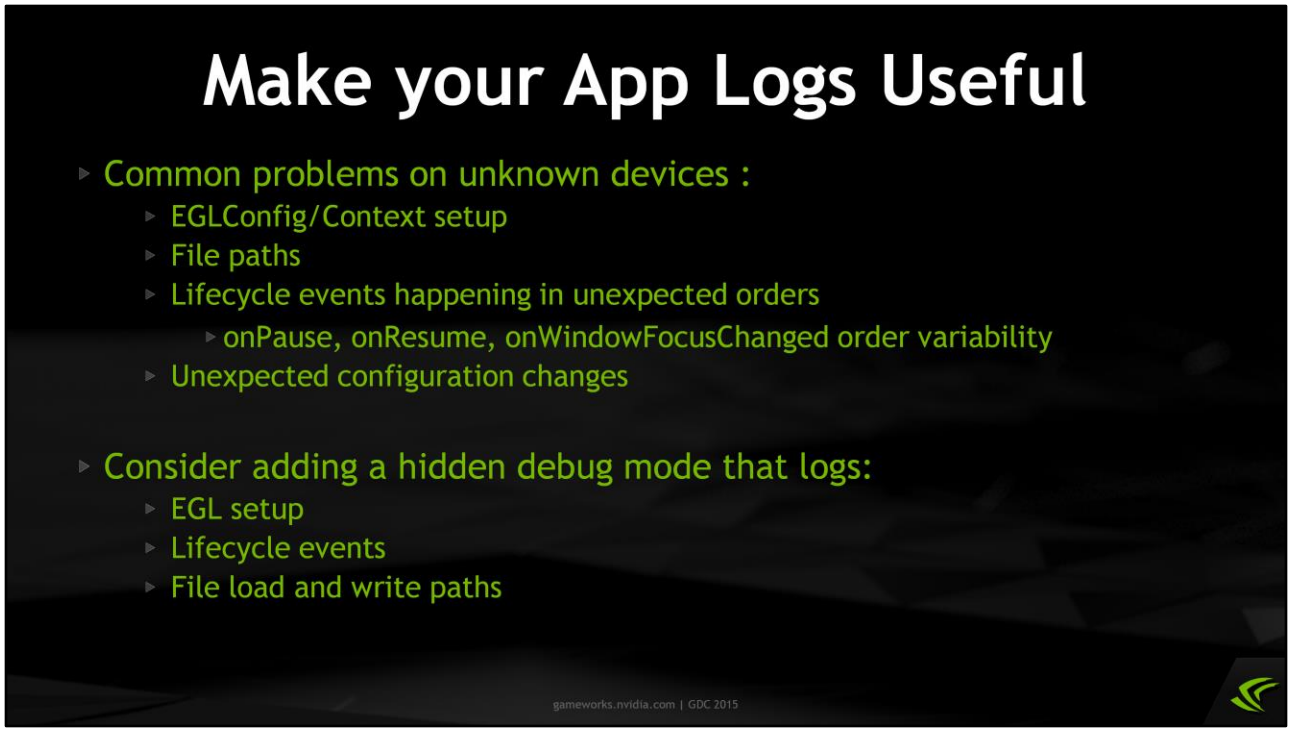
If you are available on a ton of devices, many, even most of your users will be on devices you do not have and have never seen. So it is important to make it easier to get useful debugging info from your customers remotely. This way, you can avoid relying on *their* vague description of the problem and get real data.

Make your App Traceable

Make it possible for end users to provide useful logs

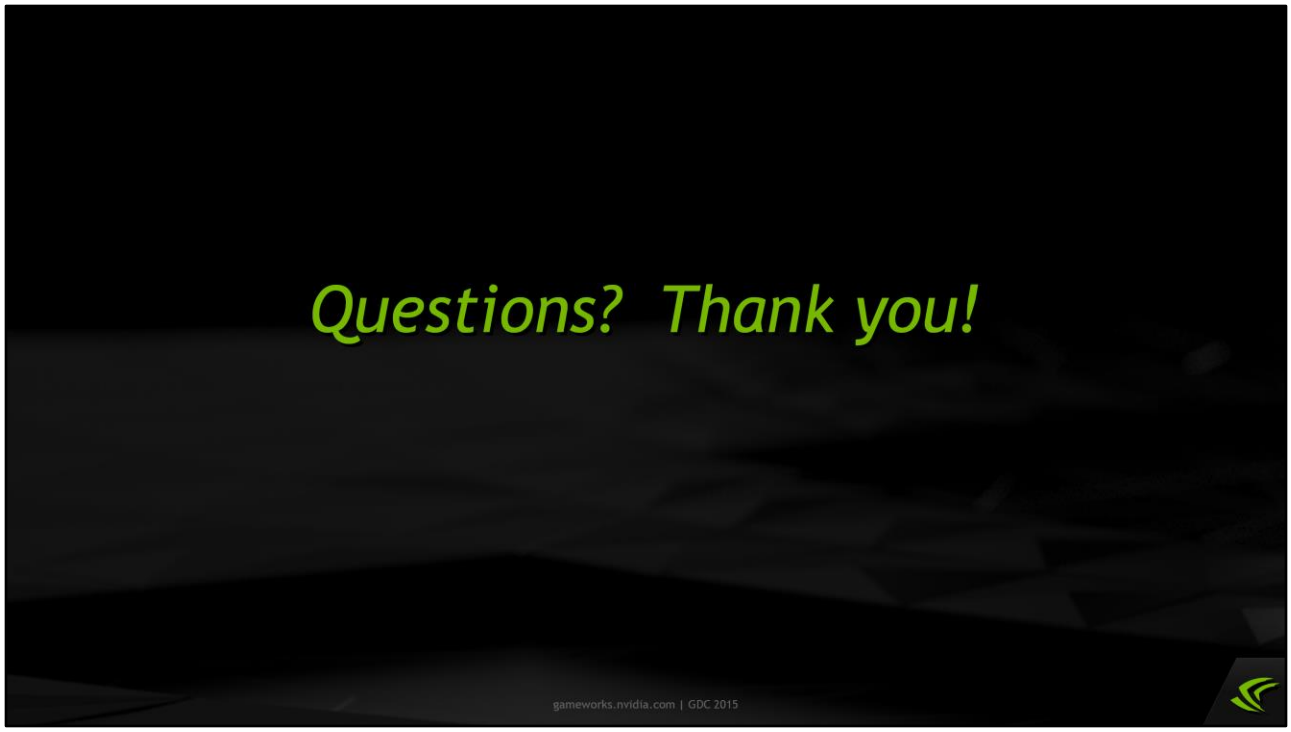
- Consumer Android L prints no native crash stack unless the app's:
  - Calls native `prctl(PR_SET_DUMPABLE, 1)`
  - Calls Java `Os.prctl`
  - Manifest declares `debuggable=true` (Not recommended for shipping apps)
- Consider adding a hidden/PW-protected option to set these
- Can be enabled remotely for users seeing problems

Due to security upgrades, Android L no longer logs detailed native stacks for app crashes on USER OS images unless the app calls process control APIs from native or Java code, or the manifest declares the app as **`debuggable`**. Of these, the lattermost is a BAD idea in a shipping app; it opens other potential holes. So the process control calls are the best option. Developers may want to consider adding a hidden option or password-protected key to unlock setting these debug flags. That way, the feature can be enabled remotely for users seeing problems.



Speaking of a debugging mode, you can also use this mode to unlock detailed logging. In practice, many common problems on unknown devices fall into one of a few cases: Graphics Config or Context setups you've never seen before, app data file path defaults you've never seen before, and lifecycle events related to suspend, resume and unlock happening in a different order than on your test devices. So consider adding a hidden debug mode that logs EGL setup (like all of the available configs), lifecycle events, file load and write paths, etcetera, so you can remotely debug what you weren't expecting to see.

Finally, a plug for NVIDIA Developer Technologies. NVIDIA Devtech engineers have years or decades of experience working with developers to help make their games better. Many of us came from “first careers” in studios or with game middleware vendors; I came from a decade in the latter. We see hundreds of games every year, and we’ve seen a lot of what can go right and wrong on Android. We also have extensive QA groups at NVIDIA that can help amplify your own QA organization. So please contact us via your NVIDIA Developer Relations rep or via our developer website.



In the end, these recommendations, strategies, resources and tools are all designed to help you ensure that your game has the best chance of being a 5-star experience on SHIELD, Android TV and Android. So please stop by NVIDIA's booth in the expo hall – try out Android TV gaming on SHIELD, and talk to our Devtech and Devtools engineers about bringing your own titles to SHIELD. Thanks for joining us today!

GAMEWORKS

- Get the latest information for developers from NVIDIA and continue the discussion
- gameworks.nvidia.com