

Android TV Gaming:

Designing (and Programming)
for Success on Marshmallow

Lars M. Bishop

Senior Developer Technologies Engineer



Good Afternoon. My name is Lars Bishop, and I'm an engineer in NVIDIA's Developer Technologies team. I focus on our Android application developers and on 3D rendering, especially on mobile platforms like NVIDIA's SHIELD.

Background

- Battles in the trenches
 - NVIDIA Devtech directly supports many dozens of AndroidTV (ATV) titles a year
 - NVIDIA SQA tests hundreds (or more) a year
- We've seen the good, the bad, the ugly and the crashing
- Getting the *basics* right is key to good reviews

"If you don't eat your meat, you can't have any pudding! How can you have any pudding if you don't eat your meat?"
R. Waters

- Getting the basics right early leaves time for standout features
- So We'll cover some of both here

The basis of this session is my team's own experiences working with two groups of people; first, our direct support interactions with Android TV and Android Mobile game and app developers. Many of these topics originally came in as developer questions. These formed the depth of the session. On the other hand, the session is also based on many discussions with and bug reports filed by our own NVIDIA software QA team, who test hundreds upon hundreds of game titles and apps on NVIDIA's various mobile platforms. These interactions formed the breadth of this talk; frequent issues we see in a wide range of titles. We're going to cover a mix of the basics and some more advanced Android feature topics. We've found getting the basics right from the start is the best way to have time for the advanced bits.



Android Marshmallow: New Features & New Gotchas

First, we want to discuss a few of the new features that Android Marshmallow brings to both mobile and Android TV developers. Along with that, there are some behavior changes that have caught quite a few developers off-guard; we'll discuss them as well.

Android Marshmallow new Features

- Core 4K support
- Adoptable Storage
- Low-latency Audio

Android M's new features cover a wide range. One of the most important to Android TV developers will be core support for 4K display. While a lot of the 4K talk is around streaming media apps, it is even easier for games to take advantage of this. Also, M adds adoptable storage, a feature that makes SD cards and USB drives even more useful on Android, both mobile and TV. Low-latency audio improves the impact of game sounds.

Building Against API23

- Cool features, but...
- Switching to a new target API is like signing a “contract”
- Existing behaviors change
- In practice, this broke many games
- Rule 1: If you build against API23, *Test on Marshmallow!*



gameworks.nvidia.com



The first thing many folks do when they consider supporting Android M is to target M's API level, 23 in order to gain access to the Java APIs for the new features. But this is a double-edged sword. Android works hard to avoid breaking or changing existing app behavior when an older-API app is installed on a new OS. But all bets are off if the app builds against the new OS's API level. That rebuilding is a form of implicit “contract”; the developer gets access to the new APIs in their app, but they also agree to any documented behavior changes in the new platform. This caused issues in a number of games we tested. In a few cases, the developer seemed to have upgraded to the new API level as a matter of course when the SDK shipped, but did not actually test on Android M devices. As more and more devices shipped with M, the hidden issues that were lurking in their apps came out. The rule here is simple; if you build on an API level, test on devices of that API level. If you also export that your app will run on older OSes, then test on those and make sure your use of any newer APIs is conditional.

4K Support

- `Display.getSupportedModes()`
 - Returns the available modes, including 4K if supported
 - Includes IDs representing the modes
- `WindowManager.LayoutParams.preferredDisplayModeId`
 - Requests a desired mode
 - Affects `SurfaceViews` in the window
 - But `SurfaceViews` are used for 3D content
- But...
 - This API does not always return 4K support on devices that can support it

Android M adds a new Java API for supported display modes. Using these APIs, an application can determine if 4K resolutions are supported on the platform, grab the ID of a 4K mode and use it. Note that this only affects Surface Views, not core Android UI views. But Surface Views are the basis upon which all 3D rendering is done on Android, so games are covered here. There's only two problems, really. One is a practical and likely temporary one. Some devices that can support 4K and are attached to 4K displays do not yet export the 4K modes via this API. So if 4K support is not indicated by these APIs, it may still be available to a game... The other issue is `NativeActivity`.

4K Native Activity Support

- But what about NativeActivity apps?

- 4K may still be available

- If the user has selected 4K in the device's settings

`com.android.tv.settings/.device.display.hdmi.HdmiActivity`

- Test for the availability of 4K via the property:

`sys.display-size`

- Set the backbuffer to 4K via the NDK call

- `ANativeWindow_setBuffersGeometry`

NativeActivity apps do not have direct access to the window parameters before creation, so the Java APIs for 4K aren't applicable. Luckily, there is another method for native apps. The app can test the `sys display-size` property; if the display lists a 4K resolution, then there are native APIs that will let you create a 4K rendering surface. If the 4K mode is available, the app can simply use the set buffers geometry call they already use in their native activity setup code to set the desired resolution. And they will be able to render and display to 4K. Do not set the buffer size to 4K if the property indicates that the display is still 1080 P. If you do that, you'll pay the price of rendering to 4K, but the scaler will just filter it to 1080 P and you will have wasted the perf. Note that this does require user intervention on their device right now. The user must enable 4K display on their device via the Android display HDMI settings panel. In fact, the app `_could_` even redirect the user to that resolution control panel page directly with an intent; the package is listed on this slide.

Permissions

- **Marshmallow Alert!** (Apps that build to API23)
- Permissions are not just listed in the manifest
- Some must be requested at runtime in app code!
 - Yes, this likely means “from Java”
 - So-called “Dangerous Permissions”
 - Includes: `READ/WRITE_EXTERNAL_STORAGE` !
 - But - all is not as it seems here...

If you build against API 23 for Android M, you implicitly sign on to a BIG change; permissions. Unlike the older system where users were shown a laundry list of every permission your app requested in a play store dialog, most of which they didn't understand anyway, API 23 actually requires some permissions to be explicitly requested of and accepted by the user at run time! If you don't do this, the APIs related to the permission will fail. This runtime request requirement is now true of all permissions that Android has declared as “dangerous”. As an example, Read Write external storage is now considered dangerous. That sounds pretty extreme, since historically a huge number of games listed this permission. But there's more here in the details.

File Access

- As of API19, Android does not require READ/WRITE_EXTERNAL_STORAGE for:
`getExternalFilesDir(String)`
`getExternalCacheDir()`
- But generic access to other paths requires READ/WRITE_EXTERNAL_STORAGE
- And as of API23, this is “**Dangerous**”
 - I.e. runtime check/request
- Games really should NOT need this. Just use the above Files/Cache dir

As of API level 19 (which seems so far ago now...) Android stopped requiring the read write external storage permission for most app. Apps that only accessed paths returned by the get external directories functions no longer needed those permissions. Those directories were considered “safe”. Only apps accessing non-app-specific directories in external storage had to request the permissions. Well, API 23 took this a step further; apps needing this wide-ranging access to external storage were now declaring a dangerous permission and have to make the request directly to the user via the request UI APIs. However, in practice, exceedingly few games need external storage beyond the per-app directories. So apps using those can simply stop requiring the external storage permission at all

Adoptable Storage

- Adoptable Storage was added to make SD cards more useful in Android
- Allows more types of data onto expandable storage
 - But also encrypts the cards
- More app storage paths change dynamically
- This broke a lot of applications that were getting away with out-of-spec code
- Apps must use the Context and ApplicationInfo functions to get data paths
 - And should not cache the paths

Related to this permissions change is the fact that Android M now makes it possible to “adopt” external storage like SD cards and USB drives. The benefit of this is that the expandable storage can be used for a wider range of application data than before. For users, the complication is that Android encrypts the storage device as a part of the adoption process. For developers, the main complication is that storage paths change. So apps that cached fixed paths to resources found that under Android M, they could no longer find their data, and they often simply crashed. Basically, the fix here is to store and load application resource data and cache data from the paths exported via Context and Application Info and to do so on each access, and not cache expected paths for later use.

Security Changes

- In M, Google changed the results of some key functions:

```
WifiInfo.getMacAddress()
```

```
BluetoothAdapter.getAddress()
```

- In M, they return **02:00:00:00:00:00!!**
 - Your app must not rely on these for DRM, identification, etc!
- Some have used **Settings.Secure.ANDROID_ID**, but is disallowed for some cases:
 - Google disallows it for advertising use:
<https://support.google.com/googleplay/android-developer/answer/6048248?hl=en>
- Must use Google Play Advertising ID (which the user can reset)

M was designed to improve user identity security, too. The biggest change that many games will see is that if they rolled their own DRM via bluetooth or Wifi MAC address, the functions returning those MAC addresses will return the same ID for all devices on M. In other words, those functions will be useless for identifying a device. In addition, apps that used the property “settings secure android ID” as a unique ID have another issue on their hands - Google specifically disallows using that ID to identify a user or device for targeted ads. They require that apps use the Google Play Advertising ID, because the user can choose regenerate that ID and wipe their advertising identity at any time. Using any other ID for targeted ads will get your app pulled from the play store.

Low Latency Audio



- Android audio latency has been improving since Android L
 - Even greater strides on Android M with Tegra systems like Nexus 9 and SHIELD
- Fast paths used to be narrow (correct format, settings, etc)
 - But are getting wider with M
- Google has detailed examples and info:
 - <http://googlesamples.github.io/android-audio-high-performance/>
 - Designed for Pro-audio devs, but the output latency sections are great for games
- There is a feature tag: `android.hardware.audio.low_latency`
 - But is mainly designed for apps that REQUIRE low-latency as a core feature
 - I.e. professional audio tools, instrument simulators



gameworks.nvidia.com



Historically, Android had a bad reputation with respect to audio latency. In the early days, this was, frankly, deserved. However, as of Android L, and especially on Android M with NVIDIA Tegra-based systems, this latency has fallen dramatically. The fast paths used to be quite narrow, and using the wrong audio format, sampling rate, et cetera would quickly have you back on the road to high lag. However, with M, those fast paths are wider. The details are beyond what we have time for here, but Google has an excellent set of resources and samples that detail these new paths. The documentation there is focused on pro-audio developers, such as those writing guitar effect emulators and the like, but the output latency sections are absolutely appropriate for game developers. There is a feature tag for low-latency audio, but games should not require it. It is designed to be used with apps that are unusable when audio latency is not perfect. Games can usually handle either case; the sound is just more effective when it is low latency.



NVIDIA.

AndroidTV: Core User Experience

Next, lets cover a little bit on developing Android TV apps focusing on the core User TV experience.

Keys to Great Android TV Titles

- Make the UI feel native to ATV
- Ensure that basic Android functionalities are solid
- Create better-than-expected visual/audio experiences
- Think of your game as an ATV *app*, *not* a game port

There are several high-level keys we've found in common for great Android TV games. First, make the UI feel native to Android TV; don't do the minimum and have it feel like a quickie port from a touch-based mobile game. This includes support for gamepad(s), remote control, and good visual feedback for the UI.

Also, ensure that basic Android functionalities are solid, including the manifest properties, application lifecycle, and permission handling. You want to create better-than-expected audiovisual experiences - this will be on a big TV. Finally, think of your game as an Android TV *app*, *not* a game port. So what do we mean by each of these?

User Interfaces

- Android TV requires a very different UI than touch gaming
- “Different”?
 - Remote-based (do not assume gamepad!)
 - 10’ visuals
 - Clear focus indication
 - Voice input common/expected
- All pieces of the UI need to be tested on AndroidTV!

First, user interfaces. Touch is a direct, immediate UI; Android TV is, by its nature much closer to console gaming. What’s different? Well, for one, you need your UI to be based on very simple inputs; left, right, up, down, select and back. Your UI will be viewed from a large distance, and it will be seen “publicly” in the sense that a TV is more public than a personal screen. You need to make sure that UI focus indication is extremely clear at all times. Also, voice input is far more common and expected if it can cut down on complex interactions. Finally, you need to be sure to actually test on Android TV, not just an Android device with a controller and maybe TV out. Why?

UI Testing

- 24" monitor @24" distance \neq ATV User Experience!
- Test your UI on a consumer TV
 - From "couch distance"
- Consider user annoyance and fatigue
- Lifecycle is a UI issue (more later)

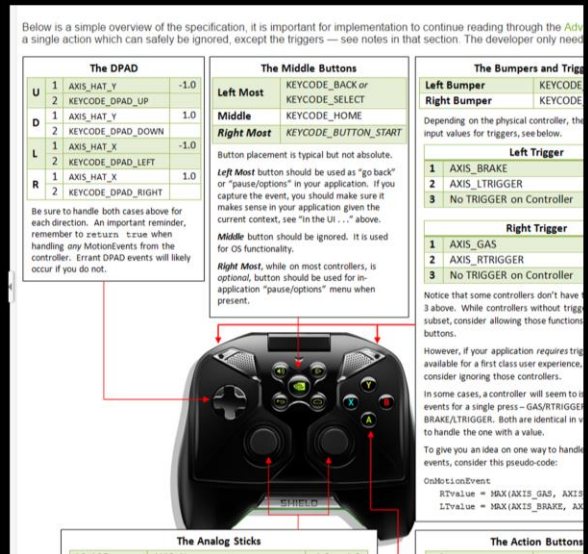


Well, for one thing, using an HDMI-compatible 24 inch monitor on your desk does not accurately represent what most Android TV users will see. You should do as much testing as possible on a consumer TV, from a couch distance, and if possible in less than perfect lighting conditions. Use the UI for an extended period of time and note where interactions get annoying. And remember that at a higher level, lifecycle issues like the device going to sleep or the user jumping to the home screen are, in a sense, UI issues.

Gamepads

- Not all gamepads are the same!
- Test for available axes/buttons
- Show controller tips on-screen
- Disable any touch tips/prompts!
- If you know the controller connected, Show the controls on an image of that controller (e.g. SHIELD Controller...)
- NVIDIA has a guide for this:

http://docs.nvidia.com/gameworks/index.html#technologies/mobile/game_controller_spec.htm



Gamepads are pivotal to most Android TV games. You should be prepared for a range of available axes and buttons, and should ensure that your UI shows controller diagrams where possible to help the user. Also, make sure the UI text replaces words like “touch”, and “tap”; it’s a little thing but it gives the user confidence that the app is going to work smoothly on Android TV. If you happen to recognize the exact make and model of a common controller, say a SHIELD controller, include images of that controller in the UI to add to the feeling of UI immersion. NVIDIA developer technologies has a detailed document on handling Android TV and controllers on our developer site - I recommend it highly.

Multiple/Dynamic Controllers

- Multiple controllers are the norm
 - Of different capabilities...
- Controller disconnect is common
- Handle controller hot-plug



GDC

gameworks.nvidia.com



Your game should be ready for multiple controllers and be prepared if those controllers are different brands and have different capabilities. This is important even for single-player games. Be ready to handle the controller the user wants to use... within reason. You should allow the base menus and UI to work with any of the active controllers. Don't lock global menus to first controller you find. Obviously, you can and should lock per-player selections to specific player's controller. An important case is to handle controller disconnect. Since these are wireless controllers in most cases, they will go to sleep with inactivity, so be ready. Android exports the `InputDevice.getDescriptor()` function to get a UID to avoid controller confusion. It is very reliable at creating unique, persistent controller IDs. Also, handle controller hot-plug. In the case of second controller hot-plugging, consider game repercussions and handle them on the fly, for example, pausing and asking if they want to add a new player.

Gamepad-related UI

~~Analog stick mouse~~

~~On-screen keyboard~~

• Selection Highlights

```
android:nextFocusLeft="@+id/myLeftNeighbor"
android:nextFocusRight="@+id/myRightNeighbor"
android:nextFocusUp="@+id/myTopNeighbor"
android:nextFocusDown="@+id/myBottomNeighbor"
```

Do not use “analog stick-based mouse” for menus! It may make sense in some forms of gameplay, but never in menus. In addition, the Android TV on-screen keyboard is NOT fun to use, so avoid making users re-enter items, for example with username password failure. Avoid long text entry at all costs - consider alternative “sign-in” and entry methods. Make your selection highlights obvious, and avoid doing it purely by color changes. On Android TV, Java-based Android UI may just work with a game controller. But test it and correct surprising left/right/up/down navigations by editing the UI XML files to specify the appropriate “neighboring” widgets in each direction. And most importantly when reusing Android Java UIs, ensure that there are no inaccessible UI components. We’ve run into apps where there were entire sections of the UI that were either impossible to access with the d-pad or else required a non-intuitive path to the item.

Remote Controls

- Support remote control in your menus/UI
 - Even if your game is a controller game!
 - Can still support gamepad shortcuts (e.g. L1/R1 paging)
- **EXTRA CREDIT:** User may launch game from remote initially
- I.e. only expect/require:
 - DPAD L/R/U/D
 - DPAD CENTER / BUTTON A
 - BACK / BUTTON B



Users may launch your game from their remote control. Even if you require a gamepad for most gameplay, consider supporting the remote control in your menus. You can still support shortcuts like paging and the like from a full game controller, but users may launch your game from the remote and not bother picking up the controller until it is time to play. Heck, if you have modes of gameplay that will work fine with just the remote, consider supporting that.



Android TV: Core Requirements

Probably the number one cause of Android TV games getting poor reviews is getting the basic Android and Android TV requirements wrong. For a game to be popular, it has to be solid on the platform. Let's discuss a few places where we commonly see issues in core Android functionality.

Manifest

- The manifest remains a source of issues
- Basic ATV items are well-understood now:
 - Banners, leanback launcher activities, etc
- But feature tags are still critical:
 - Gamepad (required or optional)
 - Touch optional
 - No requirement of non-ATV features (e.g. fine location)

```
<?xml version="1.0" encoding="utf-8"
<!-- BEGIN_INCLUDE(manifest) -->
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.nvidia.thru"
    android:versionCode="1"
    android:versionName="1.0"

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
    <uses-feature android:name="android.hardware.gamepad" android:required="true" />
    <uses-feature android:name="android.hardware.touchscreen" android:required="false" />
    <uses-feature android:name="android.hardware.location.fine" android:required="false" />
```

First, the Android manifest xml file. Such a basic thing. Just an XML file. And yet so many app issues, some taking surprisingly long to debug have ended up being one line in this pivotal file. The manifest controls much of how your app launches, how it behaves in conjunction with other apps, and even whether or not a potential customer will ever see your app in the play store on their device. Sure, the basics like setting up the required leanback launcher activity and banner images for Android TV are really not an issue anymore in the games we test. But we still see issues with apps not declaring required and optional features correctly, among other problems.

Manifest and Lifecycle...

- Something as simple as a manifest tag can change *lifecycle*
 - Seen in a real title
 - Caused last-minute pre-launch crashing!
- Triggered by launching an app from different parts of ATV UI
 - Frequency: low, but severity: high (ATV had to be rebooted to run the game again)

In fact, we had one case in a real-world title that we supported where the manifest caused a complex lifecycle error. During NVIDIA SQA pre-release testing of this third party title, the testers noticed that if they launched the app from the global settings panel “apps” (the one that also has uninstall), played the game, and then pressed the home button to go back to the homescreen, and THEN launched the app from the app’s home screen tile, the app stopped working. Worse yet, there was no way to make the app work again without rebooting. Even force-closing the app was not sufficient. While this was a rare use case, the severity was quite bad, so it was a pretty high priority issue.

So what Happened?

- The two launches trigger different Intents
 - App ended up with two running copies of the Java Activity!
- Solution was simple (once we found it...)

`android:launchMode="singleInstance"`

After a good bit of debugging in the game and in a small test app, we realized <click> that the two different ways of launching the app in the UI generated similar but unique Intents in Android. As a result, <click> TWO copies of the app's Java class were being created, something the app had no intention of seeing or handling. In the end, <click> it was a simple manifest key - the app wasn't designed to handle two instances of itself, but had not declared the <click> single instance requirement in the manifest. One line, so much pain.

Lifecycle

- Test on ATV - we still see differences from tablet
- Leanback Launcher is different...
 - E.g. apps can be visible behind the homescreen (AKA “visibleBehind” - more later)
- Audio focus is (more) important
 - Grab audio focus (when you need it!)
 - Register for notifications to know when you lose it

In general, we can't stress this enough - if your app supports tablet and Android TV, test as many lifecycle cases as you can on both systems. <click>The Leanback launcher is quite different from the tablet launcher. Apps can actually be visible behind the homescreen in some cases. It's a cool feature, and one we'll discuss later in this session, but it does mean that your app's lifecycle could be even more complex. Also, keep in mind that since Android TV is a media device, <click>Audio Focus is very important. Audio focus in Android can be independent of visual focus, to allow things like a music player in the background. So your app should be aware of audio focus, request focus when it wants to own all audio, and be a good citizen in terms of releasing it when not needed.

Lifecycle Specific Cases

- Exiting the app

- We found that on ATV, we saw a different sequence of events on exit
- Had to keep our native event loop running longer into the exit sequence for clean exit

- Daydream

- Active, visual “sleep” / screensaver
- More common on ATV because there’s no battery to save
- First, don’t keep any wakelocks when your game is at all “inactive” (don’t block daydream)
- Be ready to dream - Daydream drops the app all the way to onStop
- And then wakes it right back up on action; test this case!

Two cases are specific to Android TV and should be tested and handled in your app. First, we found that the lifecycle events when exiting on Android TV can differ from those on tablet. Test this, and be sure to keep your event loop spinning until you get that final Destroy message, so your app isn’t sitting in Java/native limbo. Also, <click>Android TV has a daydream mode - it is an active, visual form of “sleep” like a screensaver. Your app should be sure to avoid holding wakelocks when paused or at a menu, so that the user’s desired daydream mode can launch during periods of inactivity. Also, test daydream, as the lifecycle messages may come in a different order than you’ve seen when a tablet goes to sleep or wakes up.

Prepare for Success

- Best to review all of the ATV requirements up-front
- NVIDIA SQA and DevTech have a guide to help

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

- Guide covers the details of many of today's topics
- And many more we didn't have time for!



There's a good bit more to consider on this topic; it's good to know what's coming up-front. NVIDIA Dev tech and SQA have collaborated to create a development guide based on years of experience testing and supporting these apps. It covers a wide range of topics that developers should consider when planning, coding and testing their Android TV games.



AndroidTV: Standout Features

Having gotten the basics right, the way to really stand out on Android TV is go the extra mile specifically for the TV experience. The more a game feels tailor-made for TV and less like a quick touch port, the more appreciative Android TV gamers are likely to be.

Multi-channel Sound

- AndroidTV is a living room platform
- Surround sound systems will be common
- Games that use multi-channel sound are more immersive and compelling
 - Ubiquitous in console games
 - But uncommon in mobile games
- Android supports multi-channel audio output
 - SHIELD supports it with low latency on Android M



As we've said by way of admonishment when talking about UI, Android TV is a living room, lean-back platform. But there's also serious benefit to that. Surround-sound systems are pretty common on today's living room TVs, and this is even more likely to be the case with TVs connected to high-end sources like SHIELD. So games have a really good reason to take advantage of this and use multi-channel sound. Android has supported multi-channel surround sound for a while, and as of Android M, SHIELD can support surround sound as a part of its low-latency path discussed earlier. So consider taking advantage of this in your game.


SHIELD Controller 2-way Audio

- SHIELD's bundled controller includes a connector for a two way headset
 - Likely unexpected on a non-handheld device
 - Excellent for 2-way in-game chat
- This can be accessed as a 2-way headset using normal Android APIs
- Currently, a headset plugged in to the SHIELD controller will appear as two devices:
 - A source of type `TYPE_DOCK`
 - A sink of type `TYPE_DOCK`



GDC

gameworks.nvidia.com

 NVIDIA

The SHIELD controller and SHIELD remote include a headset jack - this is NOT just a headphone jack. It supports two-way headsets with a microphone. This is likely not something an Android TV game would expect, but it can be extremely useful, especially if your game supports two-way network-based voice chat on other platforms. The trick here is that the headset is exposed as being connected to a dock. In this sense, the wireless controller appears as a dock and exports two Audio devices to the Android APIs; one sink and one source. Consider hooking these up if your game is designed to include chat.

Home Screen Integration

- The ATV homescreen is transparent!
- Apps can be visible (and animated!) behind the homescreen
- API is simple:
 - `requestVisibleBehind()`
 - `onVisibleBehindCanceled()`
- But the feature should be used carefully/judiciously
- Best for cases where the game's action **MUST** continue with or without the player

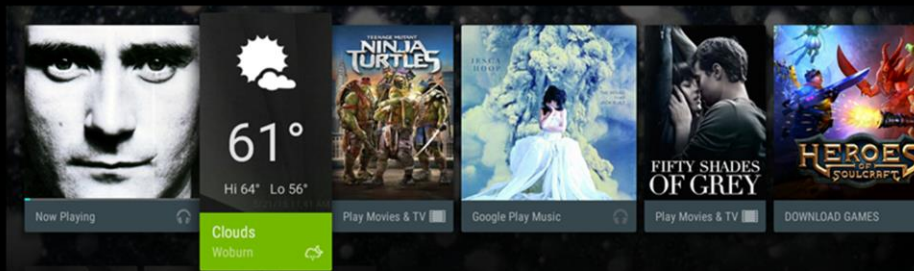
For those of us who spent years on Android tablet apps and games before Android TV, this next feature was surprising. The Android TV homescreen allows an app to declare that it should still be visible when the user presses the home button. That's right, you can request that your app be visible and animated behind the homescreen tiles! The API is simple, almost trivial - you request to be visible behind the homescreen, and then you listen for events that tell you when you are visible behind and when you are no longer visible. Not every game should use this. But in cases where the user dropping to the homescreen will not actually end or pause the game, say if they are a spectator in someone else's game, or they are a part of a game that does not pause or end just because they step back, it can be quite a cool feature. Most importantly, it continues to keep your game in the user's consciousness and can help pull them back in.

Android Recommendations “Tiles”

- Apps can generate their own recommendation tiles!
- Java-only feature, but easy to create

<https://developer.nvidia.com/content/android-tv-recommendations>

- Be considerate and do not spam the recommendations row...
- Lots of cases could be useful for games



GDC

gameworks.nvidia.com



The tiles across the top of the homescreen are not just the domain of Google’s applications and services. ANY app can provide recommendations, including games. It is a Java-only feature, but the APIs are quite simple. NVIDIA DevTech has a guide for how apps can create , publish and update these tiles dynamically, referenced here. Games can take great advantage of this as a way to remind users of their game. Recommendations could include announcements of new in-game DLC, or updates for free to play games that a new feature or resource is available. In some ways, they can be used like tablet notifications. And just like notifications, be responsible and considerate of the user when deciding what to show and how frequently. Add value for the user, lest they get annoyed and uninstall your app.

Deep Search Integration

- Allows your app to show up as a [voice] search result from the homescreen
 - Can even link to some specific behavior or location in your app
- A bit “niche” for games
- But can be a useful integration for games where you want players to keep returning



Deep search integration allows your app to provide potential search results in the global Android search. Not only can those results link to your app, they can link quite deeply into your app. While making it possible to say your app’s name and launch its main menu is fine, you can go further than this. For example, you could provide search shortcuts that launch quick games, common matchups or other actions. If your app supports information that changes over time like rankings, making it possible to quickly voice search into the leadboard could be a nice shortcut for a user. Once users learn about these search options, it makes it even easier for them to jump back into your game.

TV Channel Integration

- AndroidTV's "Live TV Channels" support tuners, but...
- Also collections of streaming videos!
- Games can create their own TV Input Framework (TIF) service
- Your game's popular streams, related videos, etc can be a channel!

<http://developer.android.com/training/tv/tif/index.html>

<http://developer.android.com/training/tv/tif/channel.html>

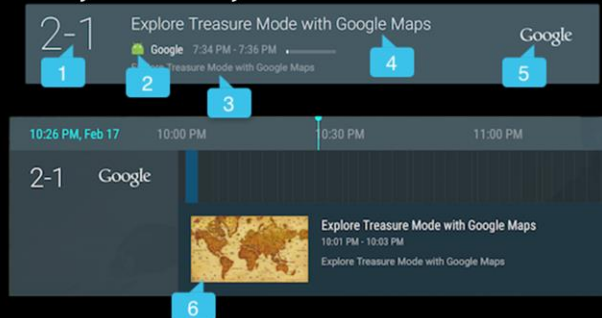
So, this next one is a little... Out there, I'll admit. But in terms of making your game's overall experience feel truly integrated into the Android TV experience, it's also pretty hard to beat. Android TV includes so-called "Live TV Channels"; while they CAN be an interface for a physical tuning device like a cablecard tuner, they can also be an interface for collections of streaming videos. So why would your game want this? Well, if anyone streams your game or creates video content surrounding your game, this is a novel and VERY nicely integrated way of getting this in front of your users.

TV Channel Code

- While not trivial, Google's example is a great place to start:

<https://github.com/googlesamples/androidtv-sample-inputs>

- The guide data is key - defines your channel



- Once the code is in place, you can serve up a dynamic set of channel information and update your "programming"

Any app can create an implementation of the TV Input Framework and start providing "channels" of content. Often, these channels are nothing more than lists of streaming video links. The linked videos appear in the Live Channels UI as a new TV channel, one video after another. You can serve the information that generates the channel "guide" and content from your game's website and update it as desired. Google even has a full sample of exactly this. It isn't a one-liner, but once the code is integrated, the ability to update the "guide" and content for your channel (or channels) is a gift that keeps giving.

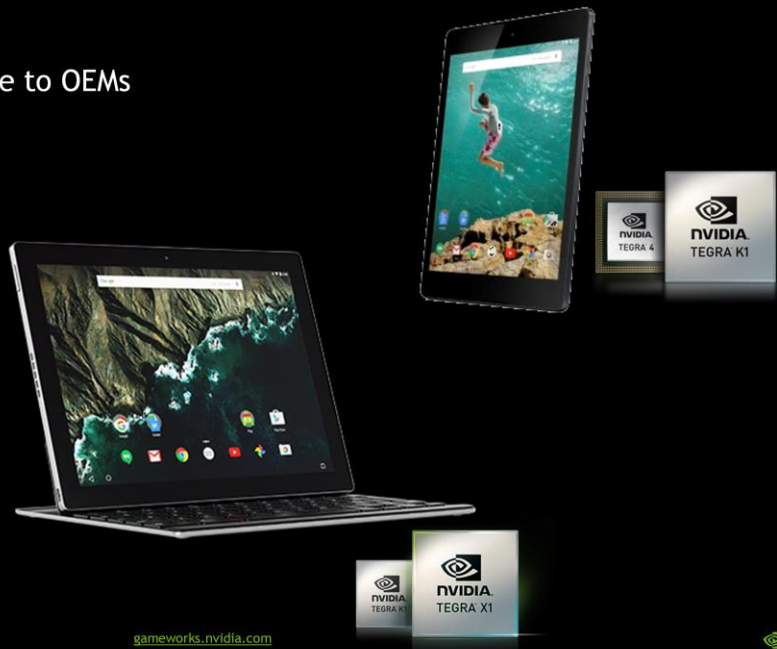


Android N: Coming Attractions

While the title of this talk focuses on Android M, Google recently made some of the coming features of Android N public, and several of them are quite interesting and worth mentioning here. Let's discuss a few of the key visual features in N.

Android N

- Slated for Q3 2016 release to OEMs
- Preview 1 Available Now:
 - Nexus 9
 - Pixel C
 - Nexus Player
 - Emulator



GDC

gameworks.nvidia.com



Android N is currently slated by Google to be released to OEMs and as open source in Q3 of this year. But the first preview SDK and preview OS images are already available. Supported platforms include the NVIDIA Tegra K1-based Nexus 9 tablet, the Tegra X1-based Pixel-C tablet/crossover, the Nexus Player Android TV and the Android emulator images. So plenty of interesting target devices to try already.

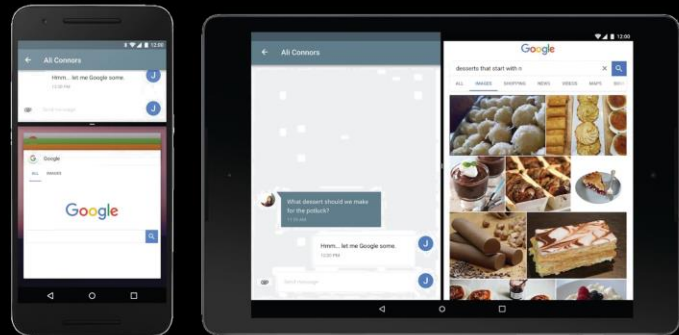
New Features for Android N

- Split Screen
- Freeform
- Picture-in-Picture
- Vulkan

The most interesting new features to us consist of two forms of tablet multi-app support, split screen and freeform, as well as the Android TV-based Picture in Picture app mode. Finally, official Vulkan support arrives in Android N as well.

Split Screen

- Multiple apps at once
- Draggable split point
- App must be able to resize
- Cannot hide System UI
- Only one is “active”
- Other is “Paused”



Split screen is a form of multi-app support on Android. It allows two apps to be shown at once on a phone or tablet screen, with an adjustable split-point between them. As a result, any app that supports split screen must support pretty generic dynamic resizing. Also, the app cannot hide the system UI in this mode. Note that there is still a single application stack that is active at once. Whichever app is the last one to receive user interaction is the focused and active app. The other visible app is in “paused” mode in terms of Android lifecycle. This is actually within even the original scope of the paused lifecycle level; paused indicates that an app is not interactable, but may be partially visible. Thus, the app can still render when it is not the active application. So video apps should likely only pause video playback on “stop” events. As for games, this is more complex, but likely they should detect split-screen and not pause the game entirely.

Split Screen Manifest

- App must handle dynamic resize, e.g.:

```
android:configChanges="orientation|screenLayout|  
    screenSize|smallestScreenSize|uiMode"
```

- App should declare resizable:

```
android:resizeableActivity="true"
```

- App should declare minimum width and height:

```
android:minimalSize
```

To fully support split screen, it is best for the app to add a few keys into their manifest. Specifically, the app should be able to resize without being restarted. Most games already have to handle configuration changes dynamically for changes like orientation, so the idea of flagging config changes for dynamic handling should not be new. I've included a likely overzealous set here, but it should be more than safe. Also, the app should declare itself as resizable. Technically, if you target Android N as your API level, this defaults to true, but always best to be explicit. And then to ensure that you never get a window too small for your rendering, you should declare your minimum size, which applies to width and height.

Split Screen APIs

```
boolean Activity.inMultiWindow()
```

```
void Activity.onMultiWindowChanged(boolean multiWindow)
```

There are few APIs for split screen, and they are both optional. The “in multi window” query lets an app detect whether it happens to be in a multi-window mode at any time. Alternatively, the activity can override the Activity “on multi window changed” function. This function will be called by the window manager with a boolean that indicates when the app is entering (true) or leaving (false) multi-window mode. These are mainly useful if the app has to change its UI or behavior when going to multi-window mode.

Freeform Mode

- Logical extension to split-screen mode
- Multiple-window, desktop-like
- Designed for larger-screen handheld devices
- Single focused window
- Resizing, window moving, etc.

Freeform mode is less of a different mode and more of a logical extension of split screen mode. It is Android's new mode for larger tablets and presents something closer to a classic desktop model. We only expect to see it used on larger screen tablets, convertibles and notebook form factor devices.

Freeform Manifest

- Default width and height when launched in freeform

`android:defaultWidthDefault`

`android:defaultHeightDefault`

Default screen location when launched in freeform mode

`android:gravity`

There are three additional manifest keys that are useful for freeform mode. They all define the desired defaults for your app. The keys set the desired default width and height, and then the “gravity” or desired location on the screen where your app would prefer to be launched when in freeform windowed mode. There are no new APIs of interest at this level for freeform mode, as it is really just a more generic version of split-screen mode.

Picture-in-Picture

- Android TV's multi-app support
- Really designed for video playback
- Use it for video-like cases

Picture-in-picture is the multi-window support mode for Android TV. The visual effect should be pretty familiar to most - the app is rendered to a small rectangle on the screen, and is (for all intents and purposes) non-interactive. It is designed rather explicitly for video playback, so games should use it for video-like cases; cutscenes and spectator mode are the best examples.

PiP Manifest / APIs

- Manifest key:

```
android:supportsPictureInPicture="true"
```

- Mode Change APIs:

```
boolean Activity.inPictureInPicture()
```

```
void Activity.onPictureInPictureChanged(boolean pip)
```

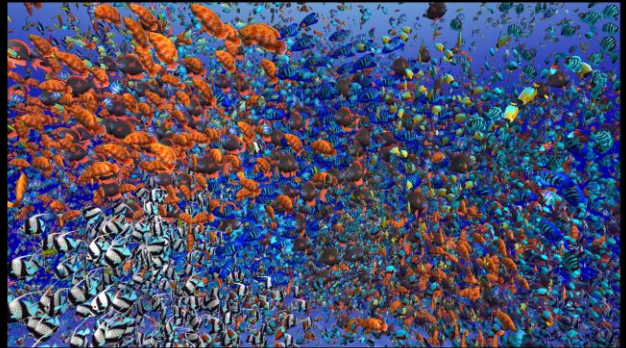
- Mode entry request:

```
Activity.enterPictureInPicture()
```

There is an explicit manifest key that declares an activity as able to support picture in picture. In addition, there are mode-change APIs for picture in picture that allow for polled or callback-based query of the current picture in picture status. Finally, unlike split screen and freeform modes, picture in picture mode is explicitly requested by the app using “enter picture in picture”. This is a request function, so use the query or callback functions to determine whether the app actually entered picture in picture mode.

Vulkan

- New, open standard for high-performance 3D rendering
 - Thread-friendly (command buffer-based, no bound contexts)
 - Efficient, low-overhead
 - Avoids runtime “hitching” e.g. precompiled shaders and state



In terms of standout new features, fewer could be more forward-looking than Vulkan. Vulkan (if you’ve somehow managed to miss it here at GDC...) is a new, open, cross-platform 3D rendering API from the Khronos group. Vulkan is specifically designed to be lightweight and thread-friendly. NVIDIA was deeply involved in the development of it, and is shipping Vulkan drivers for all of its platforms today.

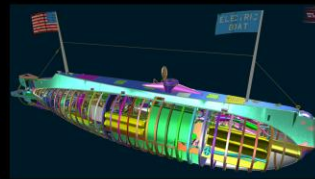
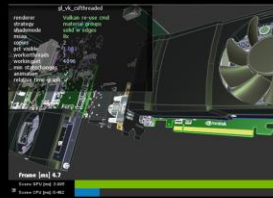
Vulkan on Android

- Available on SHIELD Android TV **TODAY**
- Android N NDK R11 includes Vulkan support:

<https://developer.android.com/ndk/downloads/index.html>

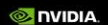
Lots of Vulkan developer resources for Android at:

<https://developer.nvidia.com/vulkan-android>



GDC

gameworks.nvidia.com



Vulkan is available for SHIELD Android TV on Android M today. The N-preview version of the NDK adds official support for the Vulkan headers and libs, which will make it even easier to use moving forward. The reason that Vulkan is of interest to Android TV developers is that it is designed from the ground up to be a low-overhead, thread-friendly rendering API. This is exactly the kind of API that can allow smart developers to use all of the CPU cores in their target platforms to feed the GPU efficiently. By lowering the overall CPU overhead, runtime hitching and expanding thread support, it becomes easier for Android TV games to get the most out of the powerful GPU in SHIELD and create really compelling visual experiences.



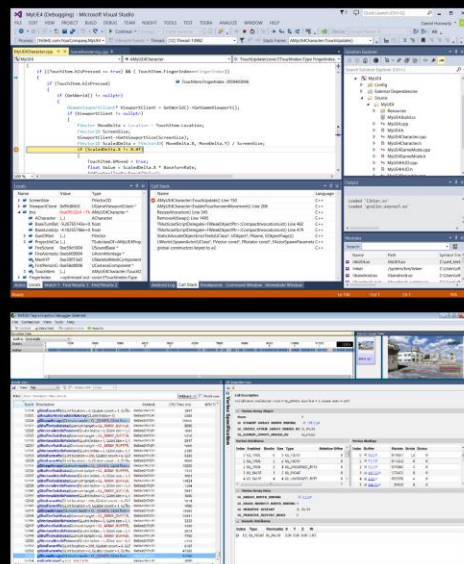
Development Resources

Finally, let's look at some resources that can make Android games easier to develop.

Tools for Success

<https://developer.nvidia.com/gameworks-tools-overview>

- Nsight Tegra, Visual Studio Edition
 - Great MSVC++ integration for Android dev and debugging
 - <https://developer.nvidia.com/nvidia-nsight-tegra>
- Tegra Graphics Debugger
 - Frame capture and review, detailed perf analysis, live shader edit
- Tegra System Profiler
 - Whole-system CPU-side profiling and analysis



Good tools are key to developing great games. Some of the individual NVIDIA tools for Android development include NSIGHT Tegra Visual Studio Edition, which includes top-notch development and debugging of Java and Native android code within Visual Studio, and the Tegra Graphics Debugger, which allows developers to capture, review and analyze OpenGL and OpenGL ES rendering in their Android app on a real device.

CodeWorks for Android

- One-stop shop for setting up an entire Android dev environment
 - Android SDK
 - Android NDK
 - NSIGHT Tegra, Visual Studio Edition
 - Tegra Graphics Debugger
 - Tegra System Profiler
 - Samples and resources



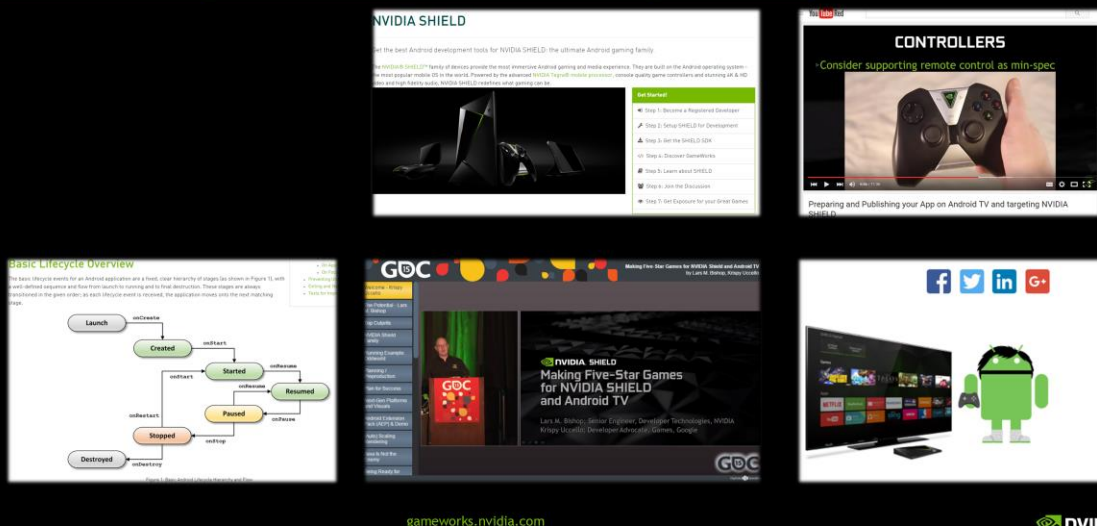
gameworks.nvidia.com



NVIDIA's developer tools include not only the individual development tools you need to create, debug and tune your games, but also makes setting up a new development machine a one-stop affair. Code Works for Android is a package that installs, manages and updates all of the tools you need for Android development, whether those tools come from Google, NVIDIA or other sources. Code Works makes it possible to take a PC with nothing more than Visual Studio and turn it into a fully-configured Android development machine automatically. In addition, it can help you manage and update those tools down the road. Visit NVIDIA's developers tools site for more details.

Blogs, Tutorials, Samples...

<https://developer.nvidia.com/develop4shield>



NVIDIA also provides tons of Android and SHIELD development resources at developer dot NVIDIA dot com. Developer guides, dev cast videos and tons of conference presentations covering Android, SHIELD, OpenGL and Vulkan development are all available there.

Summary

- Android TV:
 - Great TV gaming and entertainment experience
- Android M:
 - New features, new challenges
- NVIDIA SHIELD:
 - Top-notch living room gaming, world class gaming performance

In summary, to build a great game for NVIDIA SHIELD and Android TV on Marshmallow, the key is to know what is available and take full advantage of as much of the performance and as many of the features as possible. Android TV games should be written as Android apps that happen to be great games. SHIELD games should take advantage of the tremendous GPU and CPU power available and look and sound great on big-screen televisions and surround-sound audio systems. All while keeping in mind the basics of creating a solid, integrated Android application.

Android TV Gaming:

Designing (and Programming)
for Success on Marshmallow

Lars M. Bishop

Senior Developer Technologies Engineer



QUESTIONS?



Thanks for joining me today - please visit our booth at the Expo floor, and see what NVIDIA has to offer for mobile, TV, desktop and VR gaming.