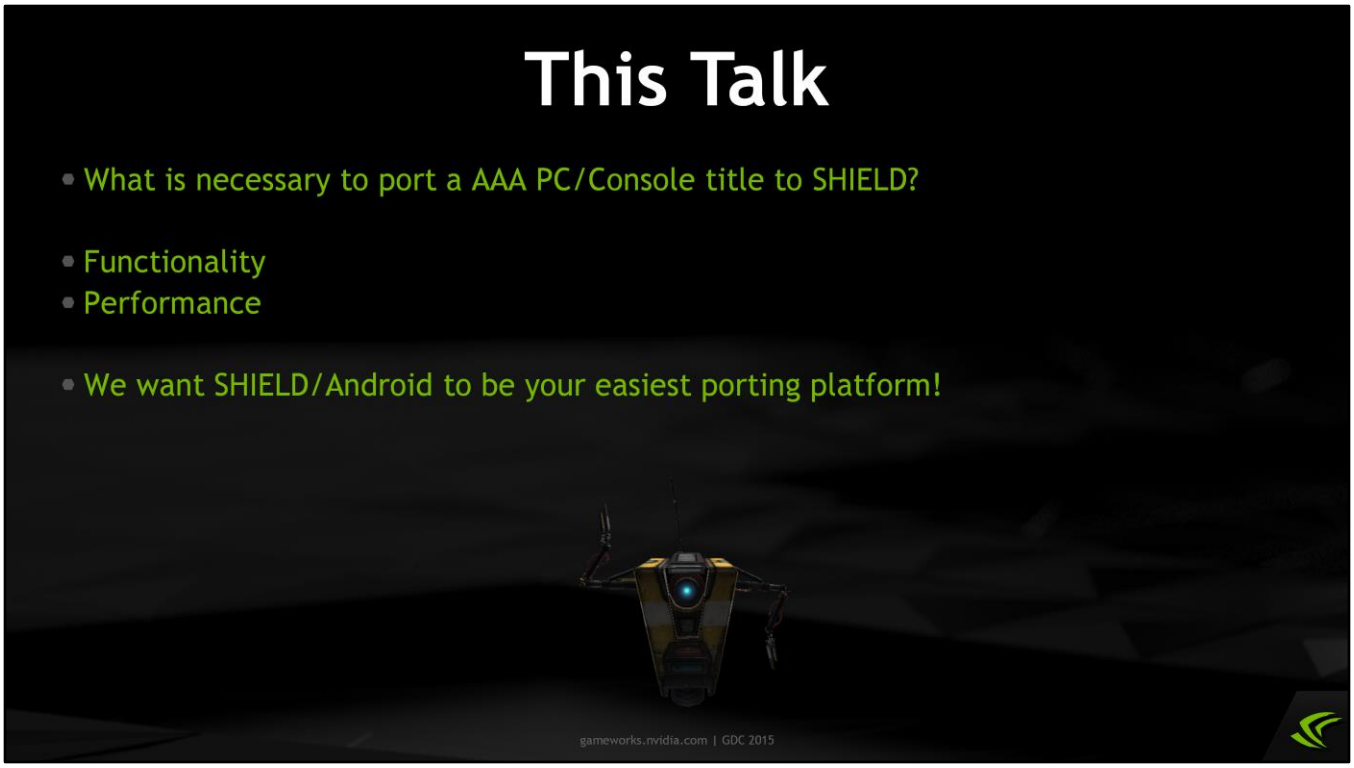
Expected talk length: 30 minutes

Welcome to “Bringing Borderlands 2 and Pre-Sequel to The SHIELD Platform”. My name is Justin Kim, and I am an engineer with NVIDIA’ s Developer Technology group. Our group works on technologies and optimizations to help developers on all NVIDIA platforms, including Geforce and now, SHIELD.

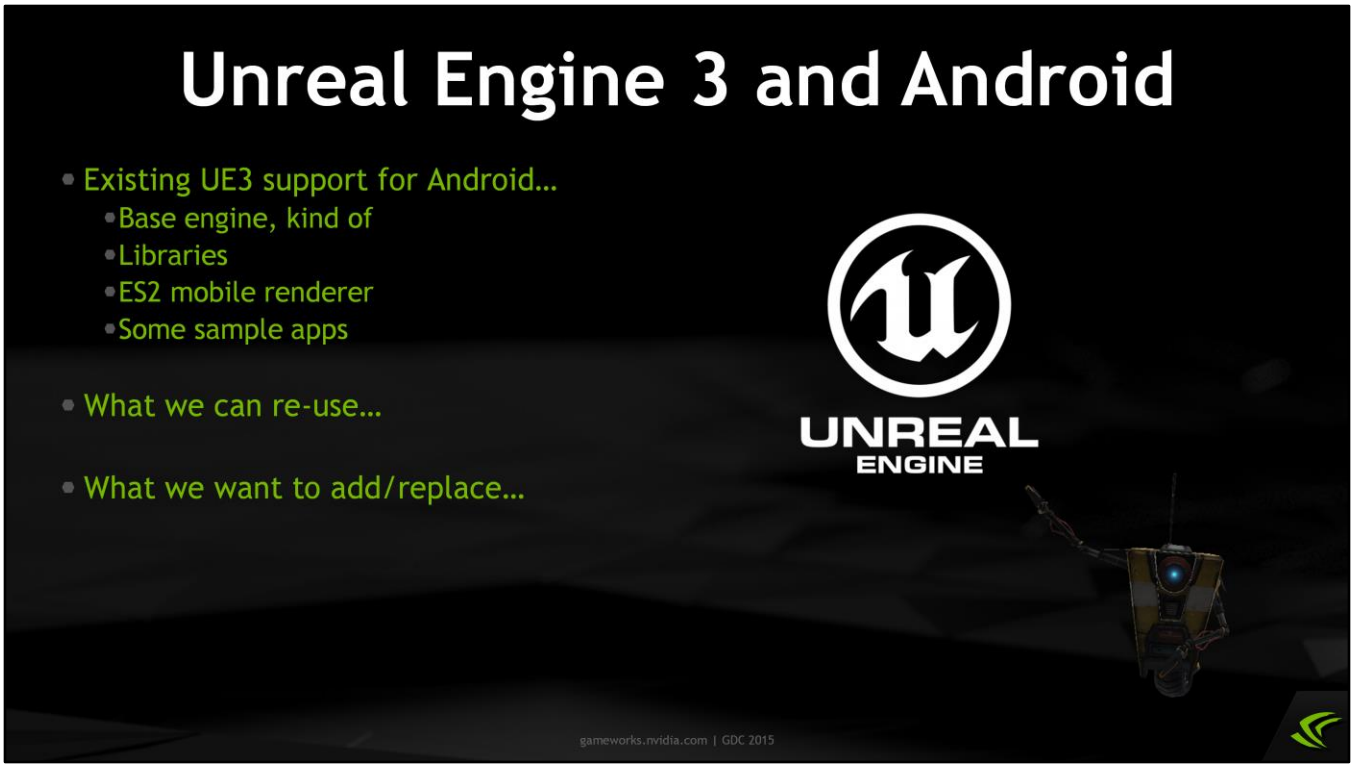


This talk is for AAA developers who either are currently making a game or have already made a game and are interested in bringing it to this new platform. In particular, I will be going into specifics for Unreal Engine 3 games, since that is what these two titles used as their foundation, but hopefully some of the lessons will be applicable elsewhere. I will cover what is necessary to get your game built and running on SHIELD, and some strategies to improve and optimize performance for the platform. Whereas before, mobile/Android versions of games were either very limited ports or their own standalone games, with the ever-increasing capabilities of devices like SHIELD, we want to demonstrate that it is feasible to port a full AAA game with minimal changes to this platform. The hope is that simplifying the porting process will make it more appealing to developers wanting maximum platform exposure for minimal commitment. Ideally, with our tools and ecosystem, we would like for Android to be the easiest porting platform for a AAA developer already working in a normal PC environment.

Introducing our cast of characters, first is BL2 and Pre-Sequel!

Here's just some details on the games themselves. The only notable detail is that for our purposes, we started with the D3D9 PC build with no built-in GL support. It may very well be the case that your game is also D3D only with no Linux support. If so, you would also be in a similar situation.

Next is our collection of SHIELD devices. Specifically, we are targeting the SHIELD Tablet and SHIELD Console with this game, running on Tegra K1 and X1, respectively.



UE3 does already have some level of Android support. However, it is not as fully featured as we would like, and as of starting, the Android version of the engine did not have all of the capabilities necessary to run this game at the level we wanted.

There are many parts of the existing Android code that we did reuse and found very helpful. Specifically, Android integration into the Unreal build toolchain is good, and saves a lot of headaches related to Android-specific build processes and the NDK. This point was particularly helpful to someone like myself, who had made a grand total of one Android NDK application before starting this project. The good integration prevents lots of lost man-hours wrestling with Android, and those who are already familiar with the Unreal build framework will find it easy to configure Android compilation. Also, the java side of the application, that is, app creation, input handling, event callbacks, etcetera is well implemented and can be used mostly as-is, with minor modifications as you add more features that you might want.

As for things we want to add or replace, we needed to add Android support for all relevant libraries (because not all had it out of the box), and had to replace the OpenGL ES2 mobile renderer, more on that later.

Unreal Engine 3 comes with support for many useful libraries, and you may also have added a few of your own during development. However, it certainly is not a guarantee that those libraries will have Android support out of the box. Many commonly used game libraries added Android support only in recent years, and so for UE3's built-in libraries, your mileage will vary depending on the specific build of your engine. With a newer build, you are more likely to have Android versions of libraries you need (although the binaries may not necessarily be built). Not only does it save you time spent integrating a newer version with possible API changes, it will also save you the trouble of having to possibly re-license a newer version to replace an incompatible one.

This list is of the libraries that we specifically had to upgrade to newer versions due to missing Android support. Scaleform alone was one of our biggest problem-causers. If you already have an Android compatible version of Scaleform in your version of UE3, it will save you a lot of trouble. For us, hooking in a new version with Android support had a different API, which required a number of changes, and then on top of that, didn't behave the same way as the old version that we had (either due to library changes or developer-specific changes).

Renderer

- Our building blocks
  - D3D9 Renderer
  - OpenGL 3.2 PC Renderer
  - OpenGL ES2 Renderer
- Addition: OpenGL ES3.1 backend

	PC	SHIELD	Other Android
D3D9	YES	NO	NO
GL 3.2	YES	YES	NO
ES 2.0	YES*	YES	YES
ES 3.1	YES*	YES	YES

Stock Unreal Engine 3 comes with support for D3D9, OpenGL 3.2, and OpenGL ES (as well as D3D11 and various console renderers). However, our version of Borderlands 2 and Pre-Sequel! only had D3D9 support at the beginning. Certainly, many developers eschew OpenGL support for PC. To maximize compatibility across Android devices, we decided to run the game on an OpenGL ES 3.1 renderer. This version of OpenGL ES supports nearly all of the OpenGL features that the UE3 renderer calls for, and so was a good starting point for development.

This matrix shows what is supported for these different rendering APIs across the relevant platforms. Most notably, SHIELD is unique among Android platforms in that it supports full desktop-class OpenGL. This feature proved to be a big help in development, since it allowed us to make easy apples-to-apples comparisons between PC and Android while maintaining the biggest features set. Also, being able to easily toggle between GL 3.2 and ES 3.1 on SHIELD helped quickly track down ES-specific issues. OpenGL ES support on PC is technically emulated, but it is a subset of desktop GL, so shouldn't have many problems if you decide to use it. Still, we found it easier and more useful to develop mainly using full OpenGL and moving to ES later. If you are developing exclusively for SHIELD, there really isn't a reason to use ES instead of full OpenGL. You may even prefer full OpenGL since you could upgrade to a 4.x context and use some of the more advanced features. However, the Android Extension Pack and other extensions will also allow some of these features on OpenGL ES if you do want to use them there as well.

You may be wondering why we didn't just use the stock UE3 mobile OpenGL ES 2.0 renderer. This is the renderer used in mobile UE3 games to date, but we found that it uses a much reduced set of functionality compared to the desktop version, and significantly diverges in terms of codepath and content. These limitations didn't align with our goals of simplifying porting to be as close to PC development as possible, and bringing full AAA experiences to SHIELD. We've emphasized the fact that our Tegra K1 and X1 processors use the exact same graphics architecture as our Geforce desktop cards, which has proven to be an advantage both because of the raw performance and because it makes fully featured driver support easier. We have also put significant effort into providing a high quality OpenGL driver which is now available on both platforms. With so much effort put into bringing all of these features to both platforms, we wanted to be able to treat them as similarly as we could. This meant that we wanted to bring the full PC features to SHIELD and see what it could do, rather than going the usual route of cutting things down for mobile from the get-go.

This part of the talk is essentially “How to Port Your Game to OpenGL”. If you already have OpenGL support (maybe due to a Linux port or just because), you’ll save yourself a big chunk of work. Otherwise, if you stripped OpenGL out of your original project, you will now need to add it back in.

Copy over the OpenGLDrv project from the corresponding stock UE3 build, hook it in, recook, add –opengl to command line, and done!

OpenGL Correctness

- Inversions! Artifacts!
- Or worst of all…nothing…

- Stock UE3 effects
- Custom effects

- Common fixes
  - Texture space inversions (t = 1.0 - t)
    - Static versus dynamically generated versus render buffers
  - Clip space Z difference (z = 2z - 1.0)
  - NaN/inf behavior (abs/epsilon)

- NSight OpenGL will help you out here

Actually, chances are you won't be done. If you didn't originally develop your game with GL support, chances are your rendering will be incorrect in some way. The stock UE3 rendering code actually plays pretty well with the GL backend (taking care of inversions, etc.), so chances are most issues are due to developer-specific code that you added for custom graphics and effects. The most common problems are texture space vertical inversions and clip space z differences. Texture space problems can again be split into three categories, static, dynamically generated, and full screen render buffers. Static textures should more or less already work since it is unlikely that you as a developer messed with that code. However, it is much more likely that you have custom code for generating certain dynamic textures for your custom rendering that assumes a D3D texture space, which should be flipped at creation time. For fullscreen effects (ambient occlusion, for example), look for string "ScaleBias" to find the scaling/bias factors used to transform from screen space to texture space for full screen effects that appear to output inverted results (ambient occlusion, for example). Also, check for any functions that draw a fullscreen quad with texture coordinates as vertex attributes and make sure those are flipped. For clip space, look to multiply all projection matrices by a correction factor (times 2, minus 1) in the Z axis. Finally, we found that OpenGL and D3D had slightly different behavior with NaNs and infs in the shader. Just to be safe, make sure you are taking the proper precautions to avoid generating these values (abs before square root, add epsilon to divisions).

One example of runtime texture inversion, our item card was upside down because scaleform rendered to texture right side up (assume D3D), but sampled it as if it was upside down (OpenGL). For render buffer inversion, we saw ambient occlusion being flipped in the final render. With NaNs/infs, we saw nearly single pixels of such values being created somewhere in the render pipeline and then being smeared all across the screen by various postprocess effects.

Remember, two inversions can add up to make it look normal at times.

NSight is an excellent tool for tracking down the sources of rendering artifacts, particularly if you have a black screen, it will let you know where in the frame it is introduced (chances are not the whole thing will be black).

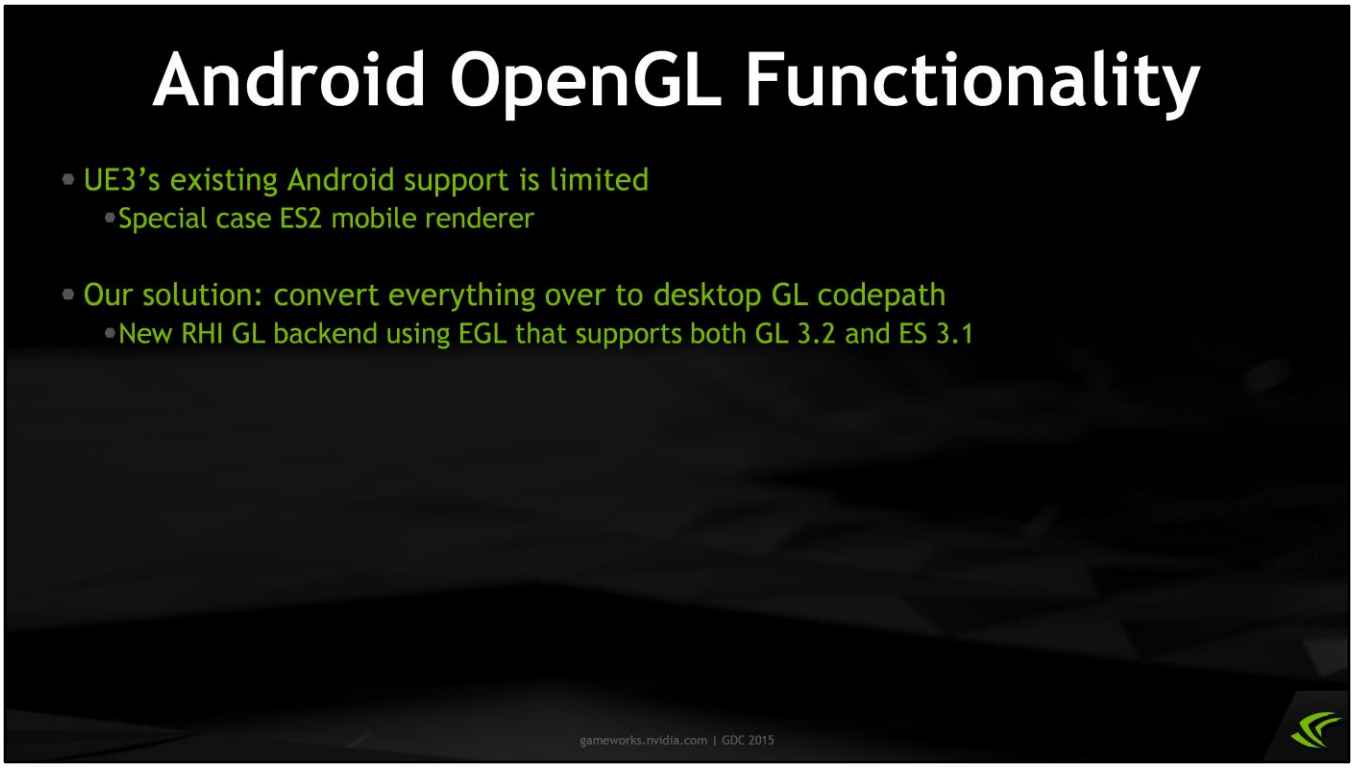
Some more example bugs we ran into for possibly common issues:

Top left: NaN value introduced during lighting calculation propagated into large bar across screen

Bottom left: NaN value introduced during HDR calculation made certain objects pure white

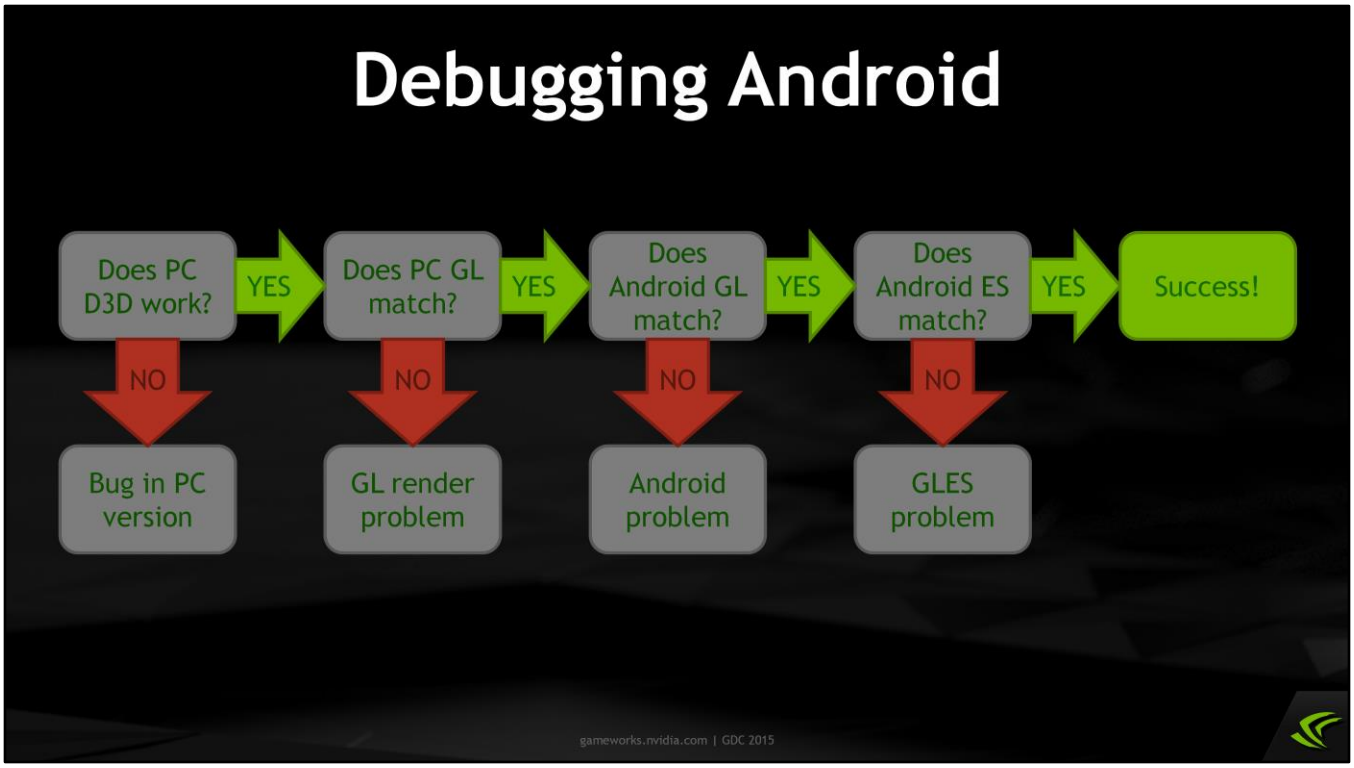
Top right: Changes in Scaleform library resulted in bad behavior

Bottom right: A collection of broken post-process, inversions, and broken instanced rendering for foliage



Like I mentioned earlier, UE3's existing Android support is significantly limited, mainly due to the cut-down OpenGL ES 2.0 renderer. These limitations are mostly due to the constraints of the Android platform at the time, many of which are no longer as significant with more advanced devices like SHIELD. For developers, it would be easiest to just use the same code and shaders for both PC and Android, and that is the capability that we want to offer. Our approach was to implement a whole new RHI (rendering hardware interface) using EGL for Android which supports both OpenGL 3.2 (or theoretically any version) and OpenGL ES 3.1. Switching is as simple as changing some flags at context creation. This is a non-trivial amount of work, but the good news is you only need to do it once and then can reuse it for all UE3 Android games in the future. The end result is that we can render the exact same thing on both PC and Android using the same textures and the same shaders (the shader cache files are identical).

We would certainly be willing to share this code with interested developers, so please get in contact if that is the case.



As per our chart on slide 8, we can debug visual errors on Android by comparing similar platforms to isolate the source of the problem to a specific area (usually either Android or OpenGL). Since we've implemented an Android renderer with full PC-level functionality, it is much easier to make an equal comparison.

Does PC D3D work? If not, it's probably some overall bug that your main release should fix normally.

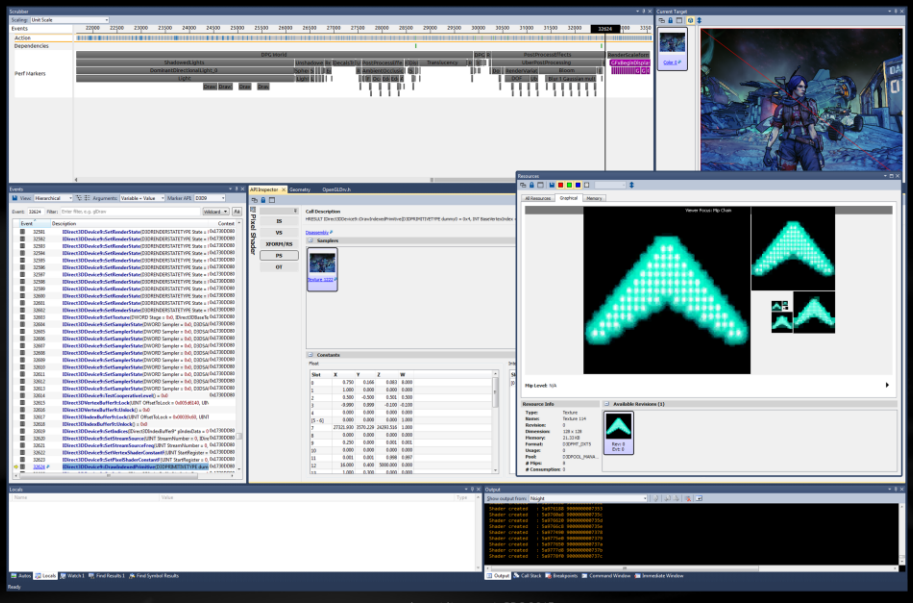
Does PC GL match? If not, it's an OpenGL-specific error, likely one of the ones mentioned on the previous slides.

Does Android GL match? If not, the Android codepath has a problem, check for #if (!)ANDROID, #if (!)CONSOLE, etcetera.

Does Android ES match? If not, it's some ES-specific issue, likely something is stricter for ES that is not a problem for GL.

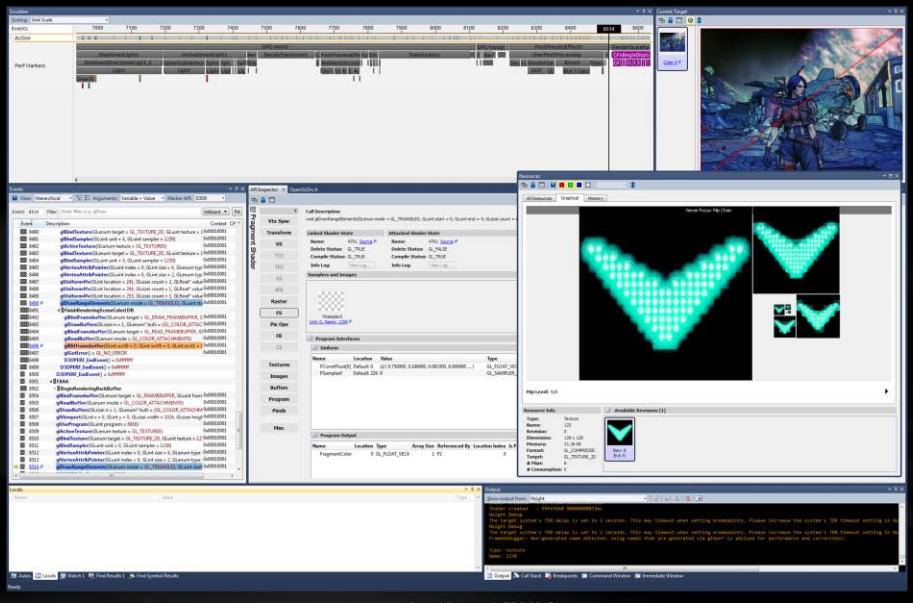
If you have a fairly bug-free PC release, in order of frequency, GL problems > Android problems > ES problems.

Debug View (D3D)

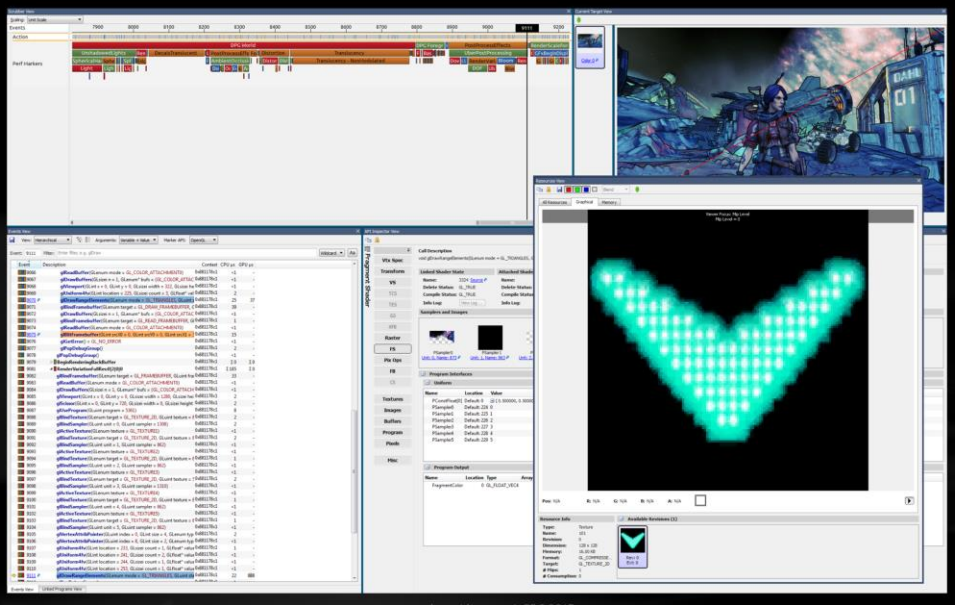


These three slides show a comparison between the various NSight debugger views for PC D3D, PC GL, and SHIELD. The interfaces are all identical and make it easy to do specific comparisons between platforms. Also shown is an example texture to demonstrate how we should expect textures to be flipped on GL versus D3D.

Debug View (PC GL)



Debug View (SHIELD)



If you've made it here, let's take a moment to appreciate the fact that we have your full, uncut game running on an Android device with a mobile processor. Performance will most certainly not be as good as PC, but now we can work to optimize and tune the game to make it perform better on this specific platform.

Initial Performance Notes

- A mix of CPU and GPU bound cases in the render thread
- More hitching on Android versus PC

- Example scene: main menu
  - 15-30FPS, GPU bound on TX1 1080p
- Example scene: first level
  - 26-28FPS, CPU/GPU bound on TX1 810p
  - Null GPU runs at 40FPS

- Game thread 10-12ms on TX1 during play

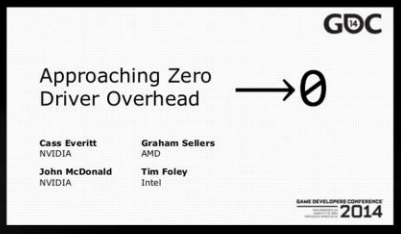
Once we got our game working, we had pretty low framerates and noticeable hitching during play. In particular, we noticed the render thread spending a lot of time in the OpenGL driver. By running with a Null GPU, which does nothing for render commands and returns instantly, we found that the best-case scenario for a zero-overhead driver and unbounded GPU would be around 40FPS with no further changes to the engine. This means that the difference between the Null GPU case and the CPU bound case was all coming from the driver overhead.

The GPU was certainly working hard as well, so there was some tuning to be done there. We believe the hitching to be caused by caching of shaders and assets.

The game thread was running fast enough to be of little to no concern.

State of the UE3 GL RHI

- OpenGL 3.2, mostly core
- Pros
  - Feature parity with D3D9
  - Ease of use
  - API idiosyncrasies
- Cons
  - Performance and efficiency
- There are many known ways to increase GL API efficiency
  - See “Approaching Zero Driver Overhead” from GDC 2014

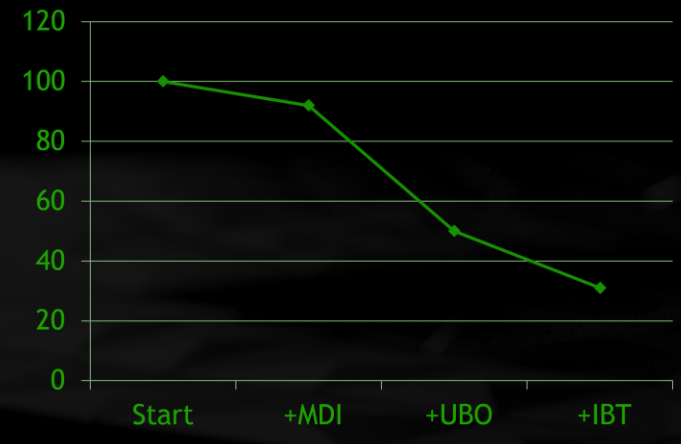


To improve the render thread performance, we had to revisit the OpenGL renderer. The renderer is actually very good in terms of functional parity with the D3D9 renderer, ease of understanding and debugging, and handling API idiosyncrasies, but it does not have particularly good performance. Due to many of the advances that we have made in improving CPU-side GL performance, there’s a lot of opportunities to make things faster. The same approaches you can take to improve GL performance on PC will carry over to Android, and there is a large existing body of knowledge on how to optimize OpenGL driver overhead.

The slow GL renderer is actually still a great tool for bringing your game up to functional correctness, as many of these optimizations will make your application more difficult to debug. It is a good idea to make all API-level optimizations switchable at compile time.

OpenGL API Optimizations

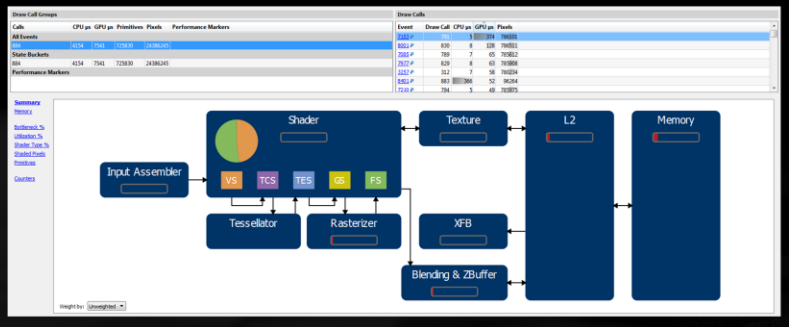
- First level test scene
  - 10K+ events reported by debugger
  - 1636 draw calls
- MultiDrawIndirect
  - 1494 draw calls (-8% from start)
- Ongoing work…
- One large UBO indexed by vertex
  - 809 draw calls (-50%)
  - Fewer uniform-related binds
- Indexed bindless textures
  - 502 draw calls (-69%)



Here is our plan of attack for adding these OpenGL optimizations. We've already added MultiDrawIndirect support, which on its own has a small gain in performance, but also is a key component in following improvements. We plan to then change uniforms from individual glUniform calls to one large uniform buffer object into which the shaders can index by vertex ID (since gl_DrawID is not accessible on OpenGL ES). This is probably the biggest potential performance gain, both due to reduction and draw calls and elimination of glUniform binds. Finally, we can leverage these changes to index bindless textures in a similar way, giving additional performance benefit. These optimizations are significant work, to be sure, but they are just implementations of a handful of known techniques.

GPU Performance

- Ongoing work on GPU bottlenecks
- Possible techniques
  - Z-prepass
  - Post-process quality
  - Anti-aliasing options
  - Resolutions
  - Shader optimization
- Depends on specific needs



We also want to consider GPU performance bottlenecks. There are a number of techniques to improve GPU performance, and their effectiveness is determined by which parts of the GPU tend to be most bottlenecked in your game.

For example, if your main draw pass is taking a long time in Fragment Shader and ROP, you may want to consider a Z-prepass to reduce that workload. If post-process passes are taking a long time due to complicated shaders, we can try reducing the quality of those effects. If Fragment Shader, Raster, and ROP are struggling in general, we could try reducing resolution. If textures are taking up too much memory or the unit is bottlenecking, reduce texture resolution. If SM is taking a long time on complex shaders (used often), look into optimizing those individual shaders close-up.

The best approach for you will depend on how your game specifically stresses the graphics hardware, which is dependent on many design aspects (art style, geometric complexity, etcetera).

NSight frame profiler is a great tool for finding GPU bottlenecks. It will tell you specifically which units are struggling on particular draw calls, and you can intuit your solutions from there.

Miscellaneous Notes and Pitfalls

- Debug context
- BGRA support on OpenGL ES
- PF_V8U8 bump maps
- ARM NEON support
- Array cookie size
- Memory alignment

Some miscellaneous notes for minor items that may save you some time.

Both OpenGL and OpenGL ES support the ARB_debug_output extension, which will help greatly in tracking down GL-specific incompatibilities and errors. It is particularly useful in debugging errors due to OpenGL ES's increased strictness compared to regular GL.

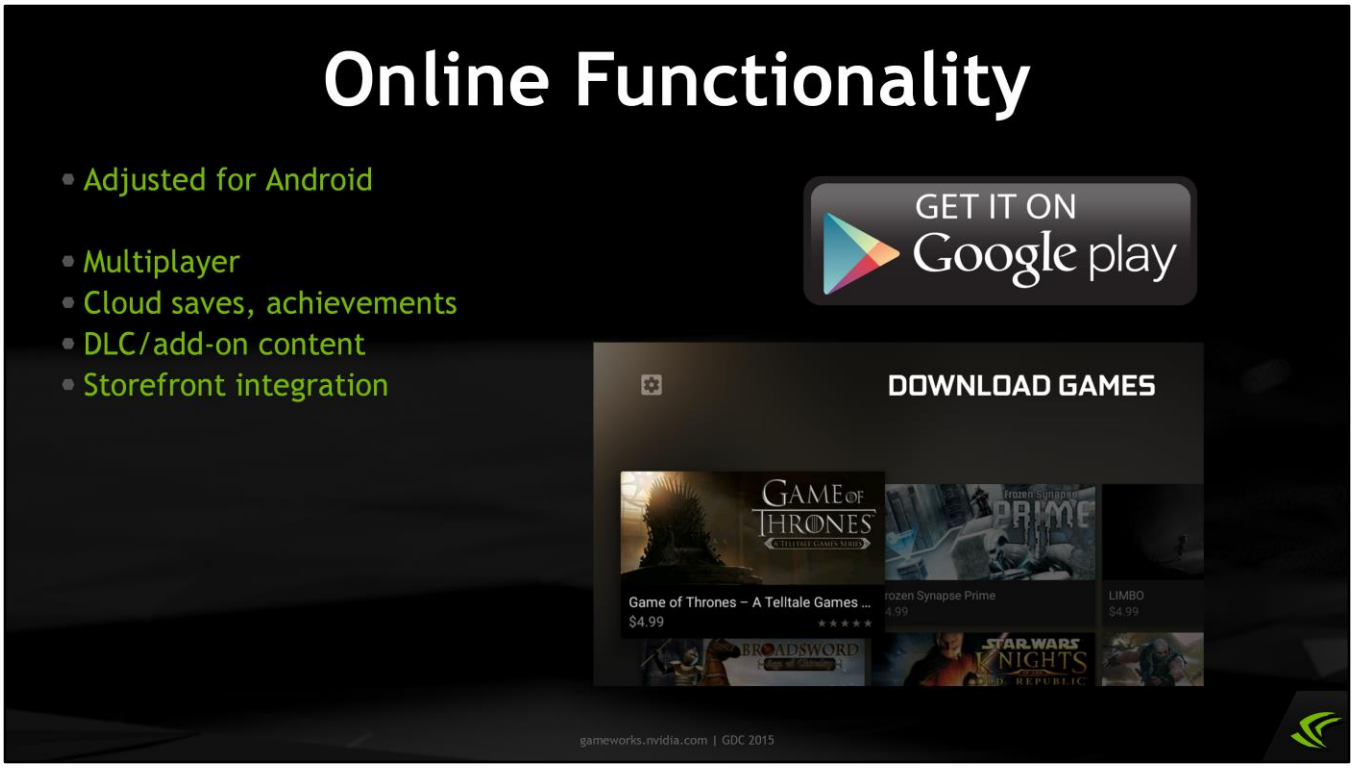
OpenGL ES does not support BGRA pixel formats out of the box, though it is a fairly common extension. Unreal Engine 3 uses this format for many of its uncompressed textures, so you will need to handle it accordingly, either by supporting the extension or by swizzling your data manually (we recommend at cook time).

OpenGL doesn't have a dedicated bump map texture format like D3D9 does (PF_V8U8), use GL_RG8_SNORM which does the same thing.

ARM NEON allows for SSE-style vectorization for CPU work, which can give you a small boost in CPU performance, but requires some wrangling to integrate it into the UE3 math headers.

Array cookie size (size added to the raw data of an array with new[]) is 8 bytes on ARM, but 4 on other platforms.

Android memory alignment is only guaranteed up to BIGGEST_ALIGNMENT of 8 bytes, but UE3 originally tries to align to 16 byte boundaries.



You will need to re-implement or modify most of your online integration code for Android. It's more or less the same thing you would need to do for Xbox Live or Playstation Network. Google Play and the SHIELD store will have their own processes for handling these different online features.

End

GameWorks

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