
NVIDIA SHIELD
Android Dev-Diaries
with EA Firemonkeys
and TickTock Games

NVIDIA

# Welcome!

- 2 amazing game companies
- Each with efficient workflows for Android
- Very different stories

- Workflows evolved
- Tales from the trenches
- How they use NVIDIA tools and hardware to
  - Develop faster
  - Raise the performance bar
  - Increase visual fidelity



# Speakers

- Your host
  - Daniel Horowitz
  - Engineering Manager, Mobile Developer Tools
- Lewis Strudwick
  - Technical Director
  - Electronic Arts, Firemonkeys
- Arden Aspinall
  - Founder & CTO
  - TickTock Games




Battery woes? Some devices can consume more power in heavy load than the 0.5 amp USB port can restore.

That's not an issue for SHIELD.

And with the USB hub, you can even keep the gamepad charging.


This is a session about some of the cool tools NVIDIA has developed for android developers – specifically as it relates to our most recent title, Need for Speed: No Limits


Some background:

- I'm a technical director at EA firemonkeys, in the central technology group

- Firemonkeys is an EA studio formed from two Australian independent mobile games developers

- We specialize in triple A mobile games, some of which you may have heard of, such as Need for Speed, Real Racing and the Sims Freeplay

# In the beginning…

- I joined mobile at the cusp of the revolution we've seen around us.
- The studio had *just* migrated from a predominantly Java + Brew background.
- We needed to take advantage of the new possibilities the technology gave us.
- But this required the studio to adapt our workflow.


Need for Speed: Shift (iOS, 2009)



We're going to start his talk with a little bit of background on how we develop games at Firemonkeys

- I joined mobile games in 2009, just after the release of the iphone 3GS

- The studio had just migrated from building simple java/brew titles to first generation smartphone games

- The new generation of devices allowed us to create richer and more compelling titles

- As a result, we needed to change the way we built our games

# Developing for Mobile: Actually quite hard.

As the games got more complex, we began to run up against some serious issues.

- Game sizes were doubling year on year
- Code complexity was doing the same thing

Need for Speed: Shift (iOS, 2009)


Need for Speed: Hot Pursuit (iOS, 2010)


Need for Speed: Most Wanted (iOS, 2012)

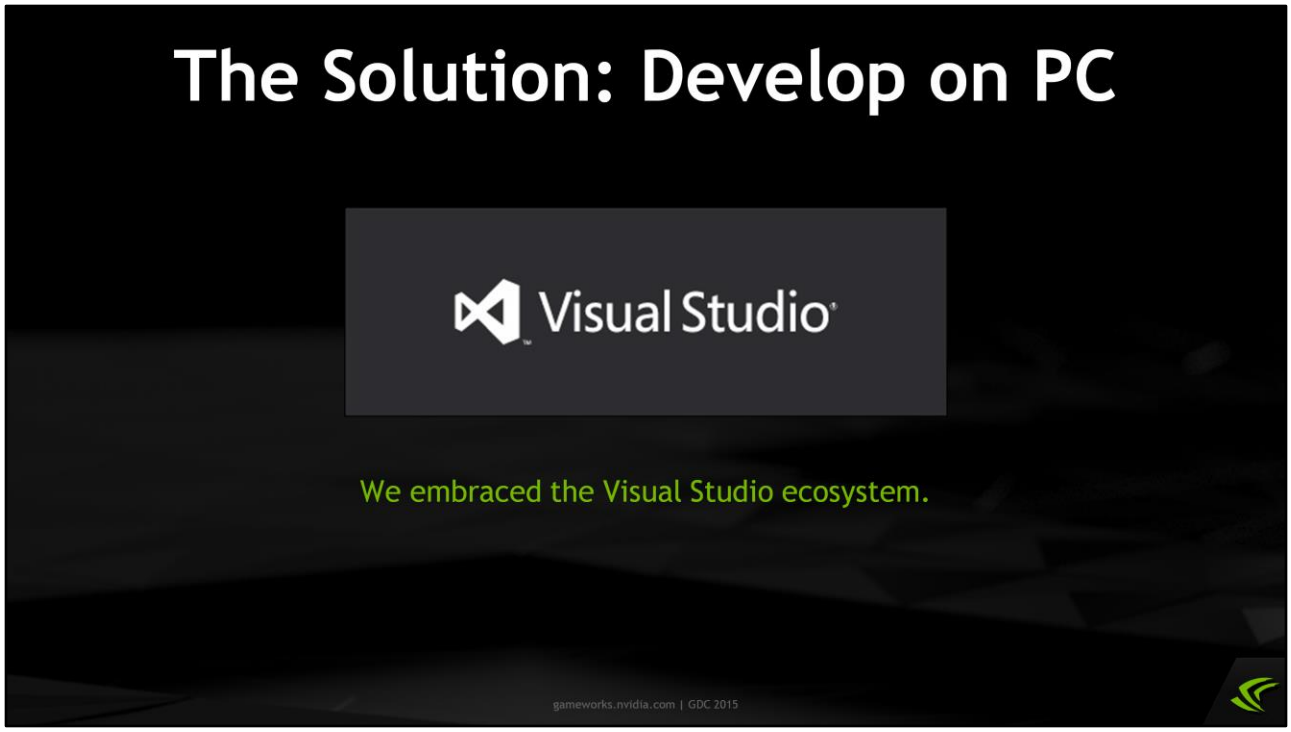
- Debugging and iteration were real limiting factors



Our titles have been doubling in complexity year on year – both in terms of code and assets

NFS Undercover (the title made before I joined) shipped at 90mb, NFS Shift at 250, and a year later, Hot pursuit was nearly 500mb. We now are close to 2GB for RR3.

The process of dealing with these more complex projects made the difficulties of mobile development, such as inconsistent debugging issues and incomprehensible deployment errors, stand out



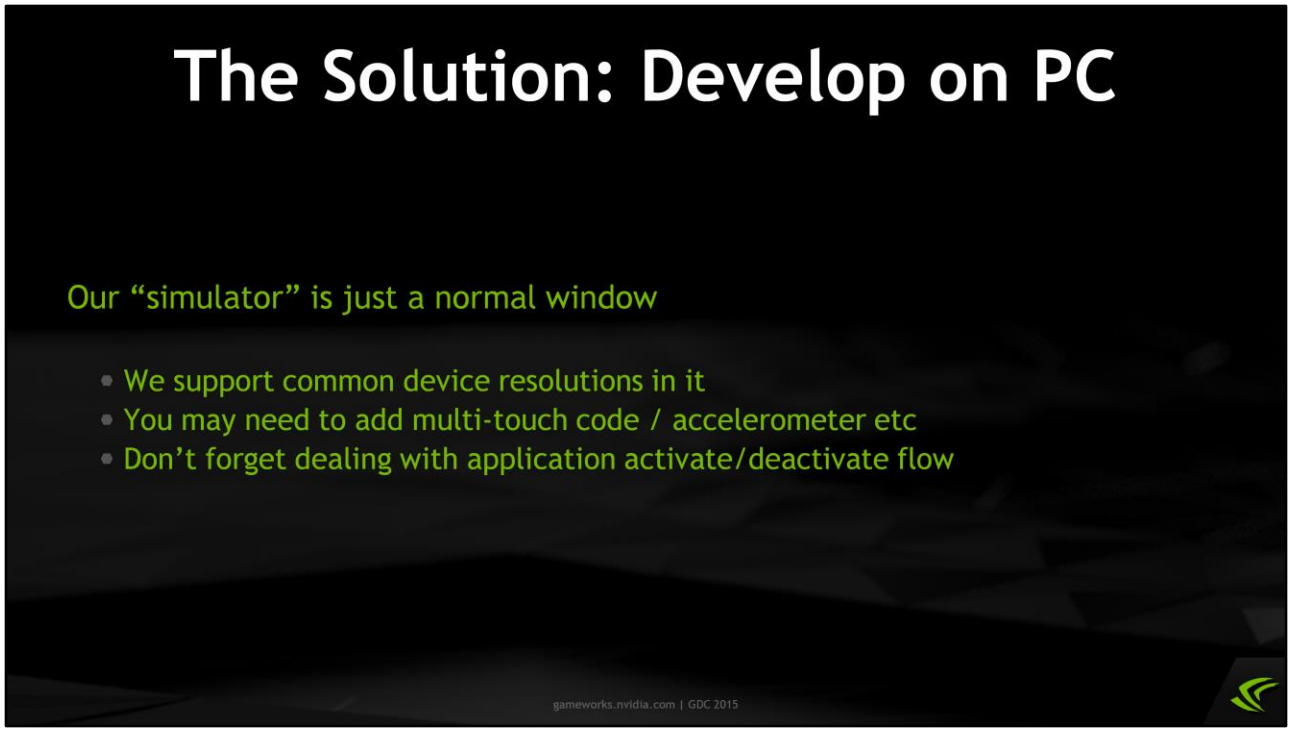

Our solution to these issues was to do the majority of the development on PC

We embraced the visual studio ecosystem


The necessitated some changes; we removed any platform specific code in the game, making it all device-agnostic C++

By having your game code clean of these dependencies, you can open up future porting options – which came in handy when we moved to android.

Any OS/graphics code was moved into platform abstraction layers in the engine




Building a working “simulator” for mobile titles is quite straightforward; ours is just a regular window

- We have pre-programmed it with common device resolutions, but support dynamic resizing for uncommon devices
- Extra inputs may need to be coded in to support multi-touch and features like the accelerometer
- It’s also worth being able to simulate app deactivation and reactivation to test those code paths

# What about the graphics?

## OpenGLES ~= OpenGL

We have a shim DLL that emulates EGL and OpenGLES specific APIs

- It also #defines out lowp, mediump, and highp and adds #version 120 to the top of shader source.
- We insert decompression routines for mobile specific texture formats like PVRTC

Interestingly, this first led us down the path of developing a relationship with NV.



You might be wondering what we do to emulate OpenGLES on Windows –

I'll let you into a secret - OpenGL is basically the same as its embedded counterpart

We have built a shim DLL that we link to which provides OpenGLES functions

For shader compatibility, we have added three #defines that allow our mobile shaders to function un-modified

In addition, we have embedded decompression routines for mobile texture formats that let us use assets we would ship with

Driver OpenGL compatibility became a big issue, so we switched to being a solely NV shop – we never aimed to ship on PCs

# (Why) I hadn't (yet) mentioned Android

- Android was originally dealt with by other teams in EA
- We tentatively looked at it and decided it was too hard
- Our games would be branched at (or near) release and sent to another EA team for porting.

== sub-par Android experience



For a long time, Android was not dealt with within the studio – we relied on external EA porting teams

At various times, we considered supporting it locally, but lacked the bandwidth to add support

This meant that our games would be branched near release and sent to another team for finalizing

Unfortunately this led to titles that gave players a sub-par experience on Android devices

# My First Date with Android ☹

- We decided to make a clean break with NFS Most Wanted in 2010, which would be the first title on our new engine.
- We were going to try to close the gap with Android
- New technology, new platform
  What could *possibly* go wrong??
- We sure were in for a shock!



With the development of our new engine – unfortunately named Isis – we decided to try adding first party android support

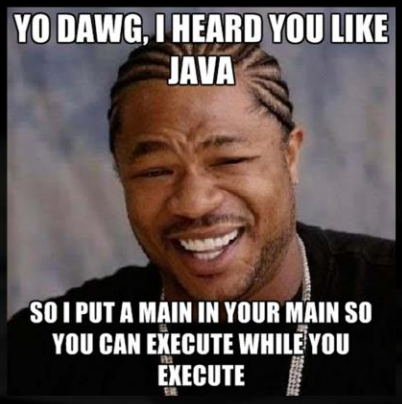
This first party support would allow external teams to only deal with device specific issues, closing the quality gap for our titles

It was a new codebase built on new technology, targeting a platform we had never dealt with

What could possibly go wrong?

# The forgotten child: the NDK

- It turns out Android is pretty keen on the whole Java thing
- Java is not the *ideal* language for our mobile games
  - Especially ones written in C++
  - …with an existing C++ engine
  - …that need to be highly efficient
- Solution(?) The NDK!

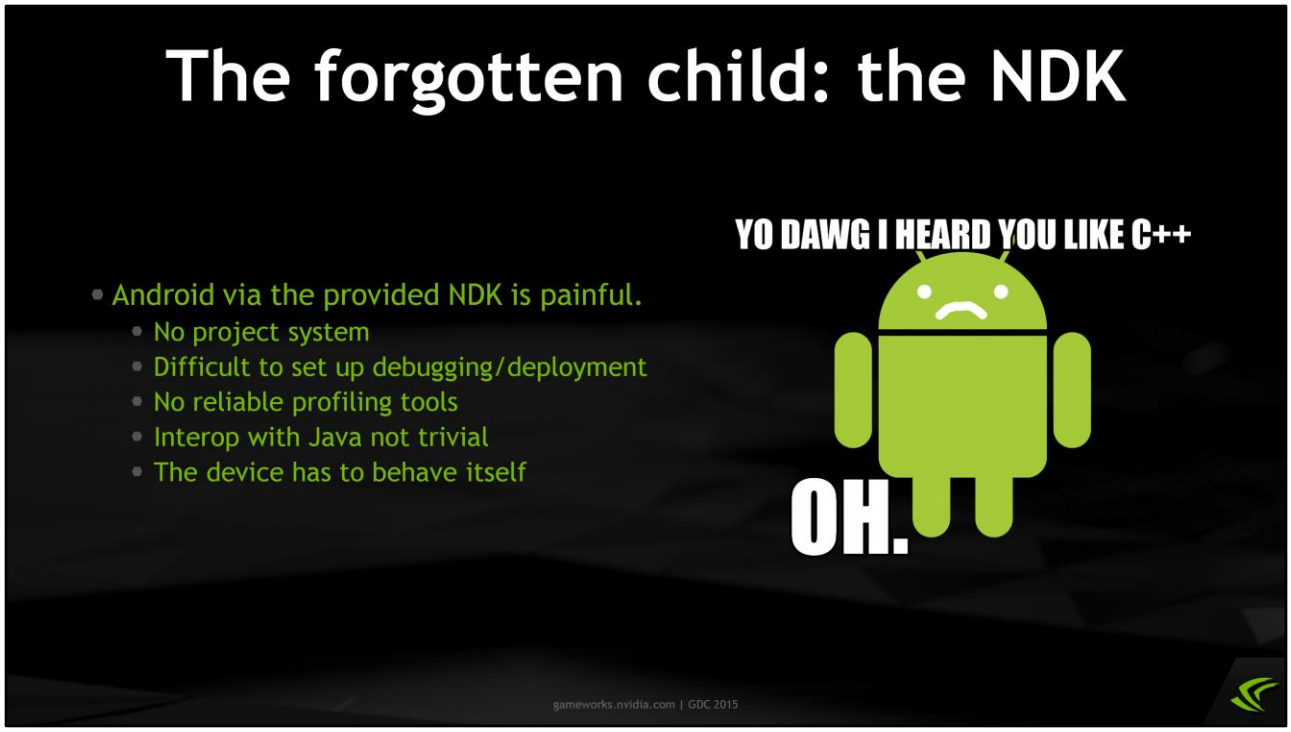





So it turns out that Android is pretty keen on the whole Java thing

Java isn't really suited for the development  of our titles, especially given our reliance on C++ and multiplatform engines

I have heard that the Java side of Android development is pretty decent (although I'll believe that when I see it), but given that we can't use the typical SDK, we are reliant on the NDK




The NDK, unfortunately, has all the trappings of being developed in a short timeframe by several interns colluding

It is extremely difficult to use it for a production workflow, lacking a project system other than make,

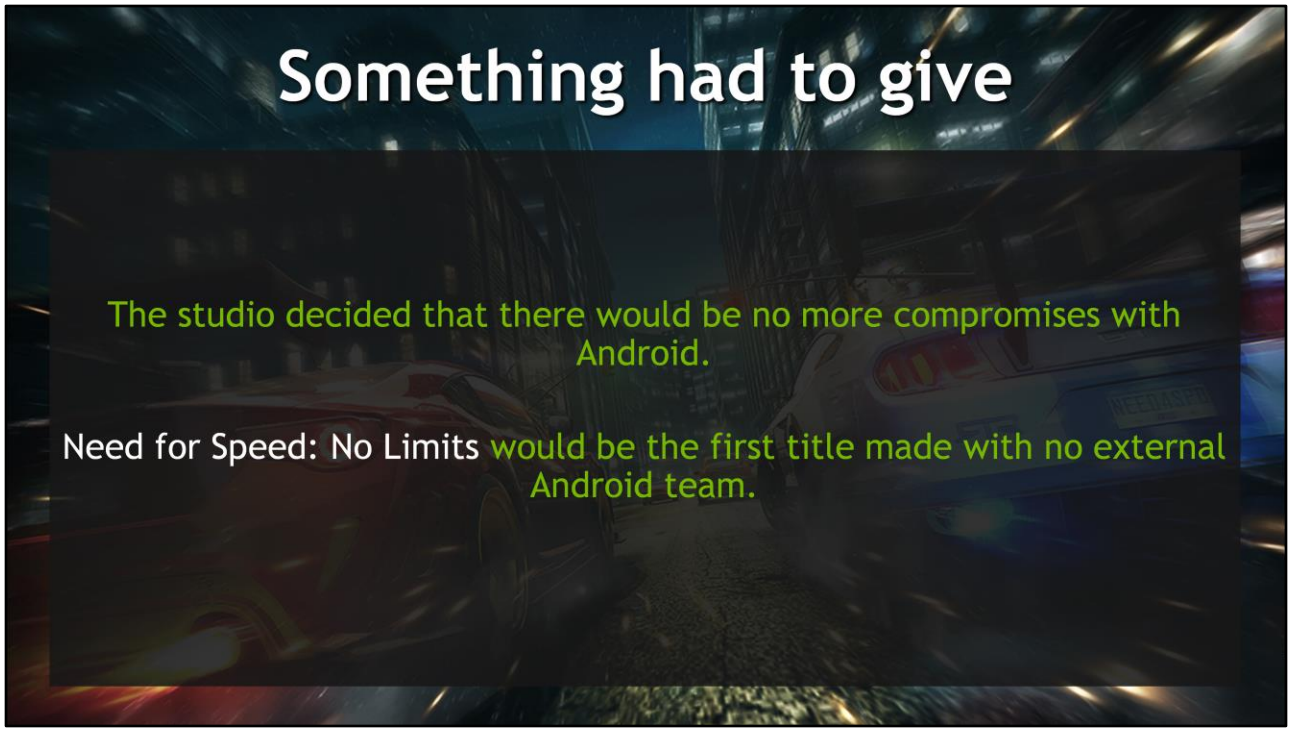
no debugging integration with IDEs (although it is ~~~possible~~~ to use the GDB transport layer with Eclipse),

no integrated profiling tools,

and it is reliant on interoperating with the Java-based OS

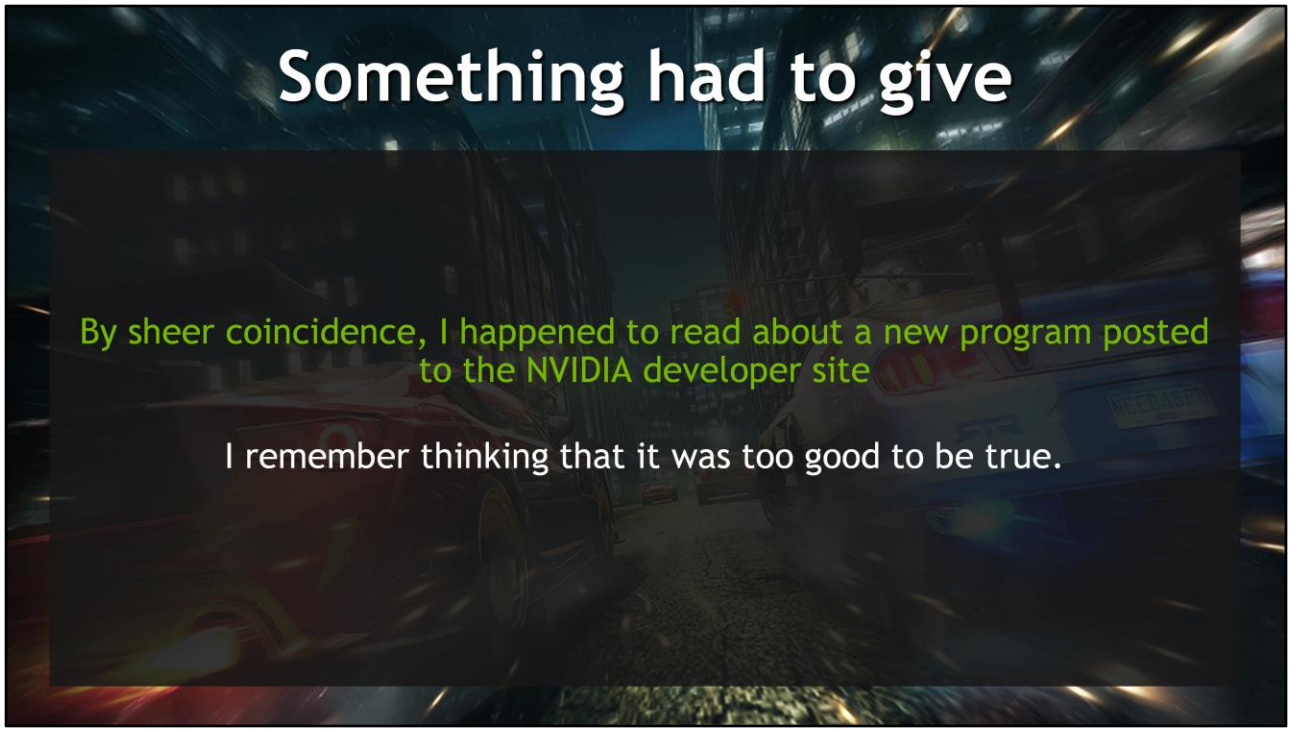
(as well as) having a device that is in a good mood.


So, let’s talk about Need for Speed: No Limits




When we started this project, the shifting market demographics meant that we could have no more compromises with our Android release

NFS would be the first title where we didn't have a dedicated android team – we had only one set of developers for both platforms




Shortly after we made that decision, I was reading the latest news from the NVIDIA developer portal, where a new program with an extremely long name was offering to make the android development “experience” more straightforward

I remember thinking that what they offered was too good to be true


The verbosely named Nsight Tegra Visual Studio Edition and the Tegra Android Development Pack slotted directly into our existing windows and visual studio workflow

- We could build and deploy within Visual Studio without requiring specialized hardware and while sharing the same project/solution files as our windows simulator
- It also supported native debugging over ADB and integrated with Visual Studio
- And – much to our surprise – even came with debugging support for Java code
- The streamlined setup process from the custom installer managed to remove most of the arcane knowledge required to build and deploy to android devices


We were already using visual studio and windows for the majority of our feature development

The way that this toolchain integrated not only allowed us to do meaningful work on android within the studio, but we could make every member of the team an “android developer”


For a long time, we had felt like the NDK had been overlooked by its creators – and our suggestions for improving it fell on deaf ears

But now we had NVIDIA really investing in the developer experience, and building it into a viable gaming platform

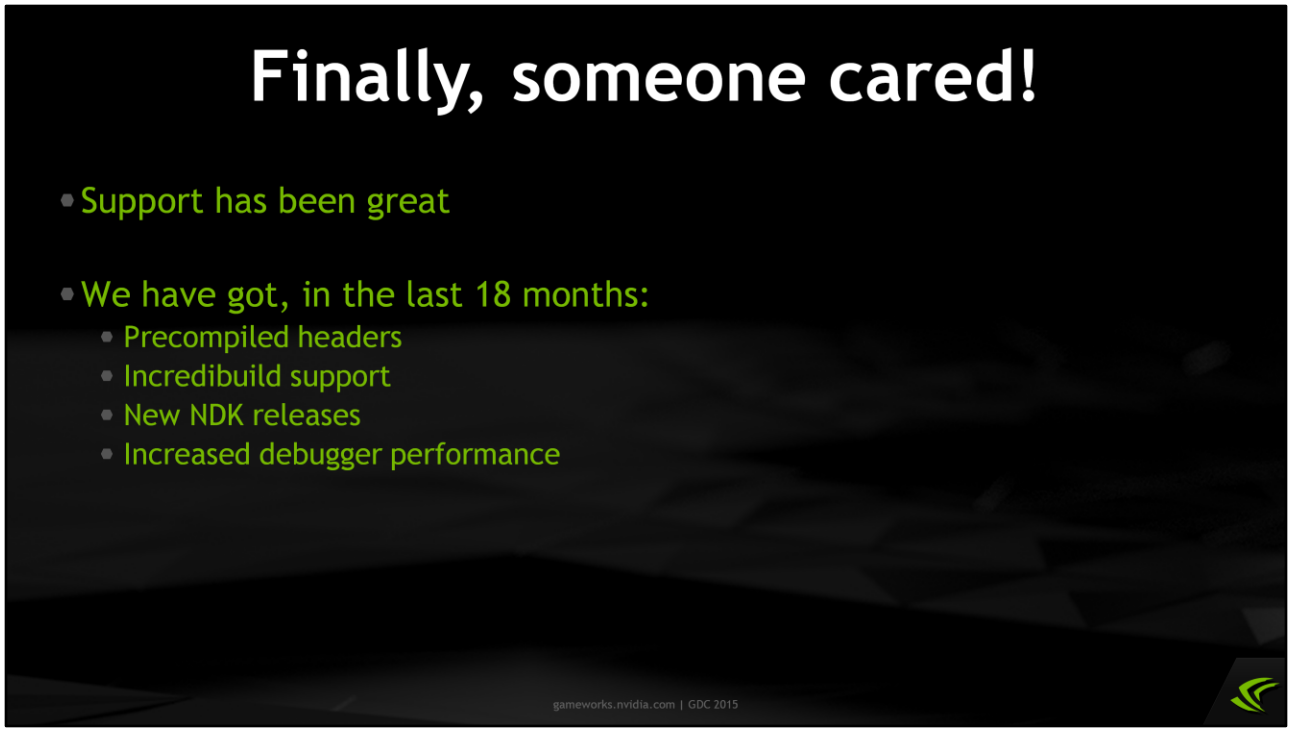
Turn into a viable gaming platform

Precompiled headers which reduced our full build time from nearly an hour to a fraction of that

Incredibuild support so now none of my programmers can complain

They've added new SDK releases and dealt with our crazy requests to use the latest possible compilers

As well as a host of other features that make the experience better




The initial releases lacked a few features we wanted, but we engaged in an email campaign to get our way, and NVIDIA responded by showing their commitment in the platform.

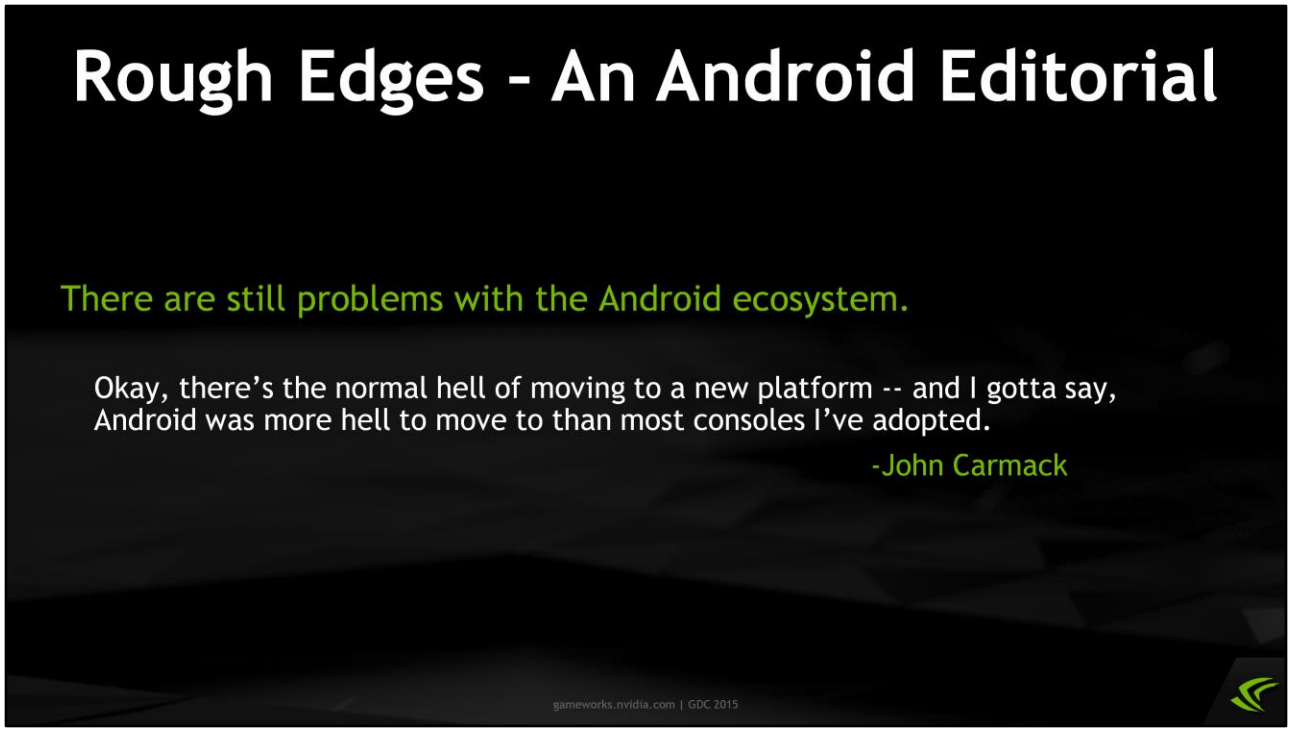
In the last 18 months during our development cycle, several major features were added – hopefully as a direct result of our requests.

We have received precompiled header support, which cut our build times by 80%

Incredibuild has recently been integrated, to improve them even further

They update the pack rapidly for each NDK release, something we are keen on as we always use the latest compiler available

And have improved debugger performance and reliability

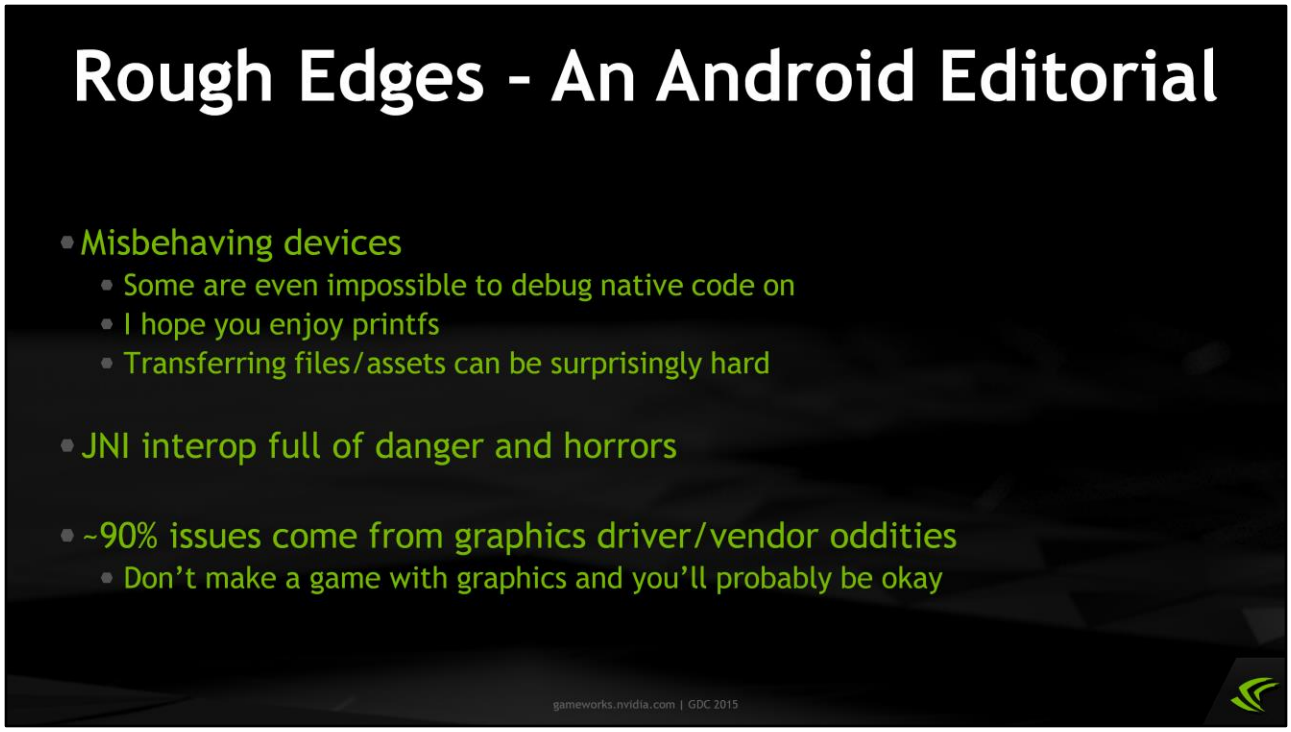



Unfortunately, there are still some problems with the Android ecosystem

John Carmack recently said:

<read quote>

Which I think sums up some of the sentiment surrounding the platform



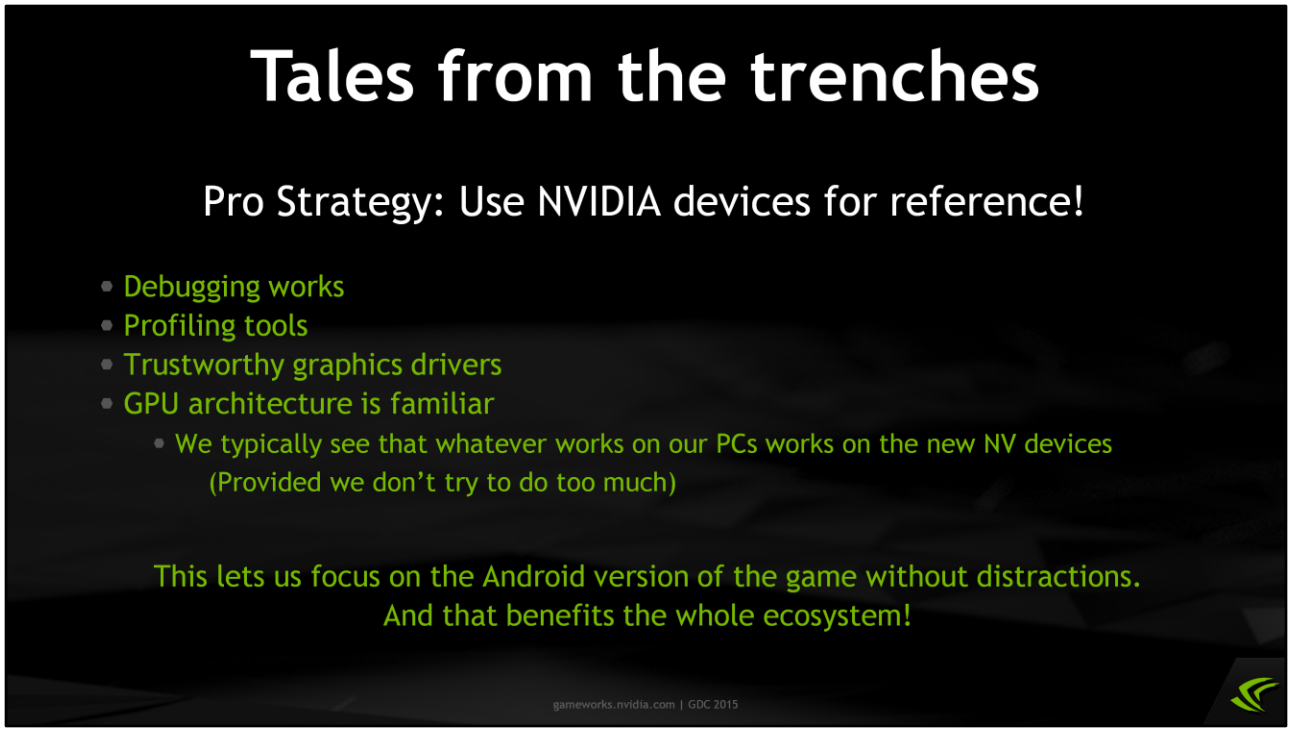

There are some core issues with building titles for Android, and sadly NVIDIA cannot fix them all.

We consistently have issues with certain device/firmware combinations

- Such as ones which have misconfigured security settings that disallow debugging
- The difficulties in debugging these devices means you often rely on printfs (not that printfs will actually appear in the output, you need a different method for that!)
- Often these devices have incomplete or suspect USB file handling, making reliable deployment difficult

Interacting with the Java system libraries via JNI can also expose you to pitfalls in the JNI system

But the vast majority of issues come from driver or vendor problems with the graphics API – as long as your game has no graphics, you are likely to be okay.

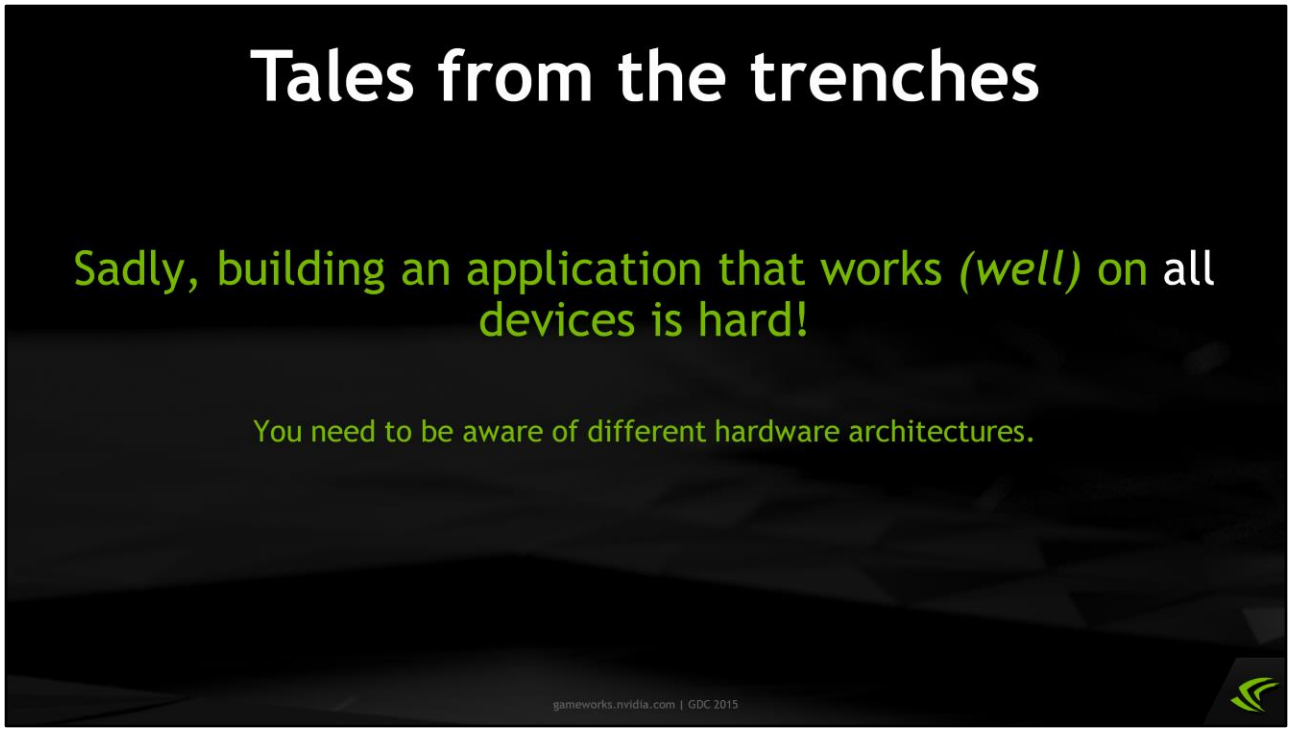



We mitigate these issues by using NVIDIA android devices as reference hardware.

These devices, such as the shield tablet, function correctly when debugging, have profiling tools, and have trustworthy graphics drivers without unexpected pitfalls

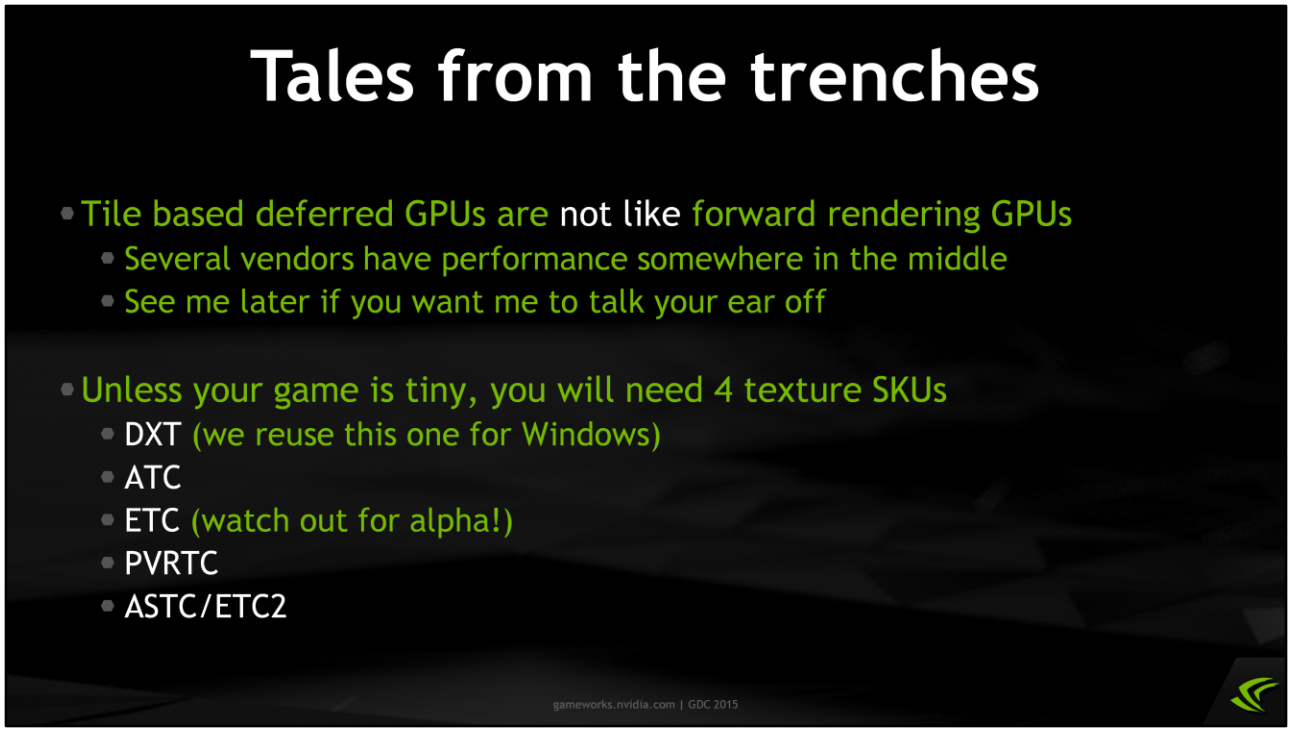
Because we are already using NVIDIA hardware on our development machines, we can typically see visual effects immediately running on device

Working on a known good device allows us to concentrate on features and deal with device-specific corner cases later




The fact of the matter is, building an application that works well on all devices is quite hard.

Notwithstanding driver or vendor issues, you need to be aware of different architectural configurations




The first thing to note is that the tile-based deferred renderers are not at all similar to the forward rendering chips in terms of their performance characteristics. While we can mitigate issues by sorting geometry differently, there are major discrepancies in fill rate and throughput.

In addition, these differing GPU architectures require that typical android titles ship with four texture compression options;

DXT, which we can reuse for our windows simulator so it takes less baking time to move to NVIDIA Shield and Tegra hardware

ATC, ETC and PVRTC

Watch out for the lack of alpha support in ETC1 devices

There is now also ASTC and ETC2 but those are not popular yet and those devices already support one of the other formats.

# Tales from the trenches

- Graphics performance surprisingly hard to predict
  - Power/thermals a big issue
  - Fill rate vs insane resolutions
  - Very little documentation on mobile GPUs
  - Unpredictable shader costs



The graphics performance of a particular device can be surprisingly hard to predict.

Unlike consoles, we must consider issues of power management and thermal characteristics.

The screen resolutions offered by android vendors vary wildly – and can often be many times larger than we would see on traditional HD platforms

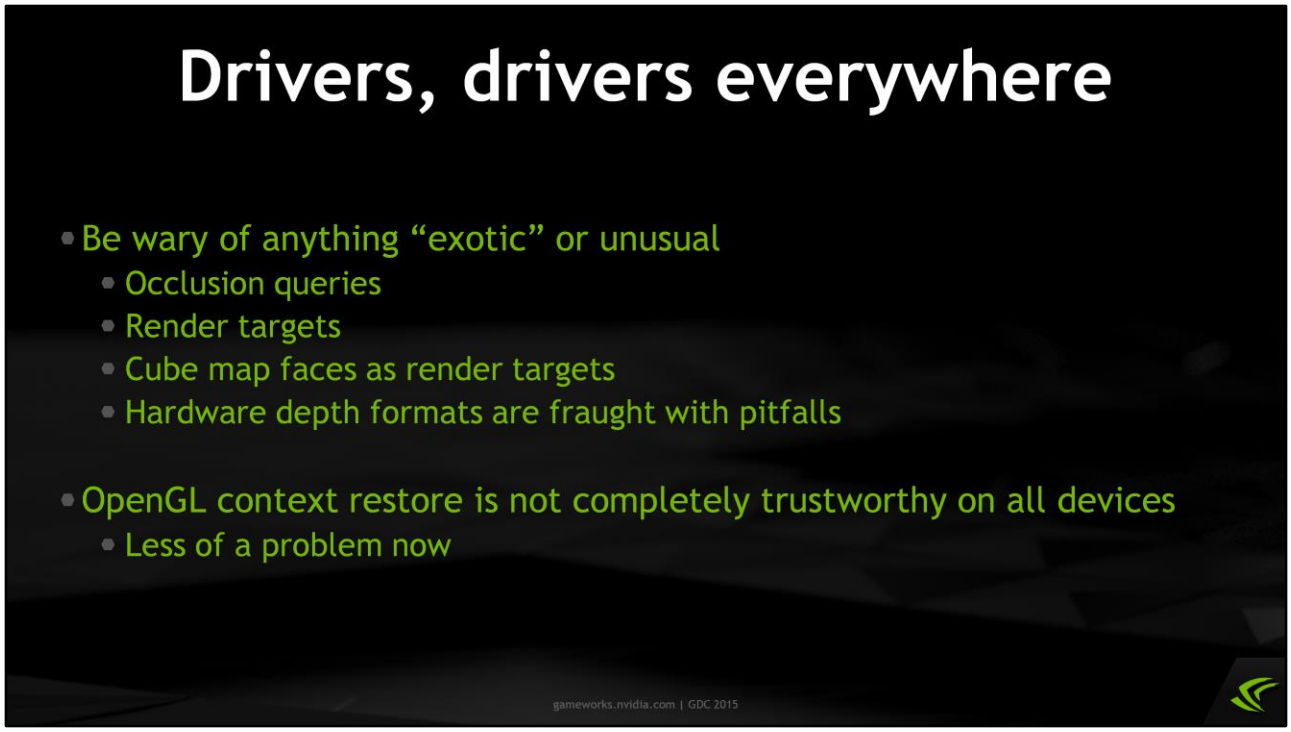
Another concern is the deep lack of documentation for mobile GPUs, which can lead to

Highly variable shading costs for what may appear to be similarly performing hardware


Not all GPU vendors have as good a driver team as the one at NVIDIA, and many GPU drivers for Android devices are lacking

- It is a common mistake to assume that all OpenGLES features will function as the specification suggests

- Even when your API call of choice functions on one device, there is no guarantee that it will perform similar or repeatable on other devices
- When considering this, remember that vendors do replace the drivers in chipsets and not all instances of identical hardware will be running the same code




Typically, we find that avoiding anything that could be construed as “exotic” helps significantly

- Need for speed had severe issues with occlusion query performance on one brand of hardware
- Rendering to a texture can also behave unpredictably, especially if you have a lot of off-screen buffers. In that particular instance, we fixed the issue by adding a readpixels function to force the hardware to function
- Binding cube map faces as render targets appears to be highly unusual for many common android vendors and has been associated with immediate device resets for completely valid code
- It is also worth ensuring you understand the various depth formats and restrictions for each type of device

Finally, there is a mechanism for restoring lost GPU resources when resuming an android activity – we have found that if you are supporting older devices in particular, you cannot rely on this to function; even when the device informs you that it is capable.


In addition to differing GPU architectures, there are many proprietary SoCs, and several distinct CPU architectures

- ARM supports an additional instruction set called "thumb" which is optimized for small binaries: it is not designed to be fast. We sometimes see libraries compiled for Thumb; please stop making them
- Most hardware we care about uses the ARMV7 instruction set with a SIMD system called Neon, with one notable exception where the vendor apparently decided that nobody needed the neon unit. That vendor has learnt its lesson and promises to be good from now on
- It's also worth remembering that there are x86 devices out there running Android, and two of them are in the top 50. x86 devices run arm code through some kind of emulation layer, but that has serious performance ramifications
- And finally, ARM64 is looming – now that iOS has switched over we'll see how fast Android apps start to convert

All in all, CPU issues seem to be blessedly rare - or maybe we just haven't noticed them

# Tales from the trenches


## Fragmentation is NO JOKE



Some of you may have seen this before – it's a visualization of the relative popularity of Android devices that logged on to the UK-based OpenSignal network. There were 4,000 different models that connected – we typically target the top 50 and hope that's a big enough range to hit most of the rest.

This has been repeated many times, but fragmentation is no joke for Android

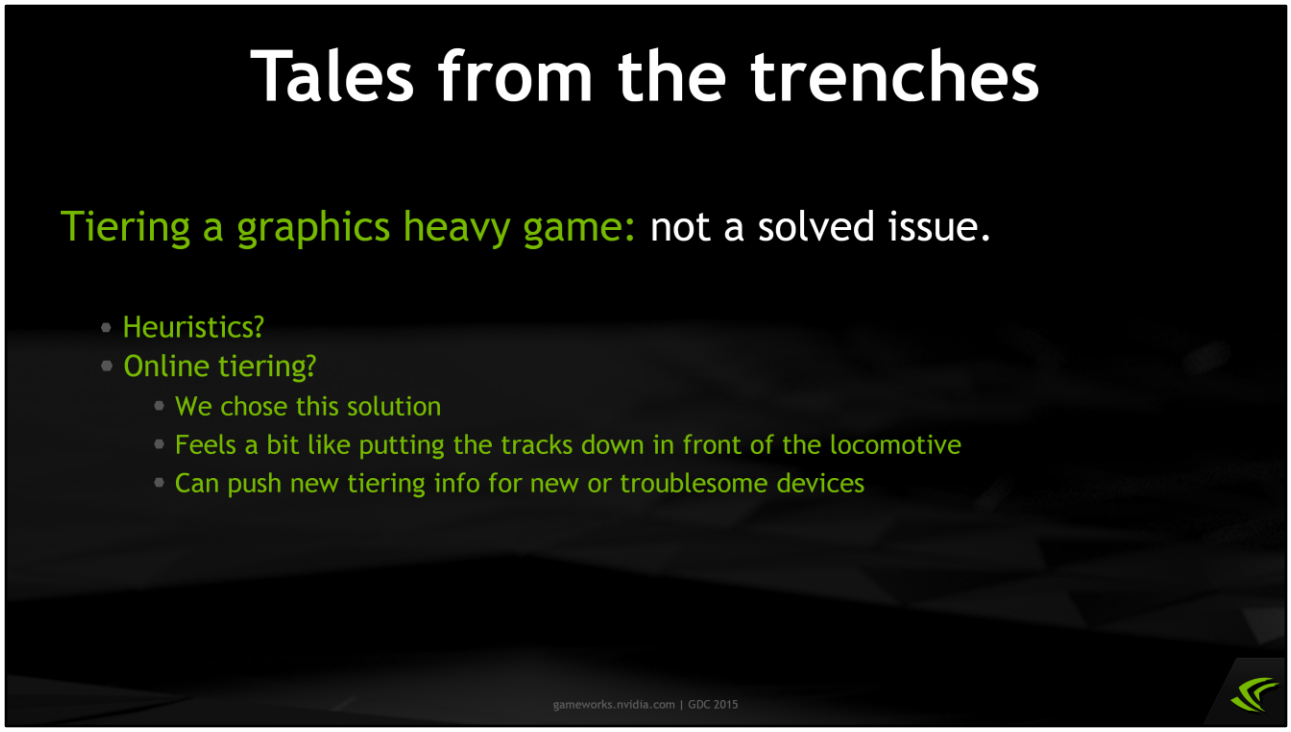
# Tales from the trenches

There are *thousands* of devices

Lots of devices are called the same thing but are *not related at all*.

- On manufacturer has 20 devices with the same name
  - Some with entirely different chipset architectures





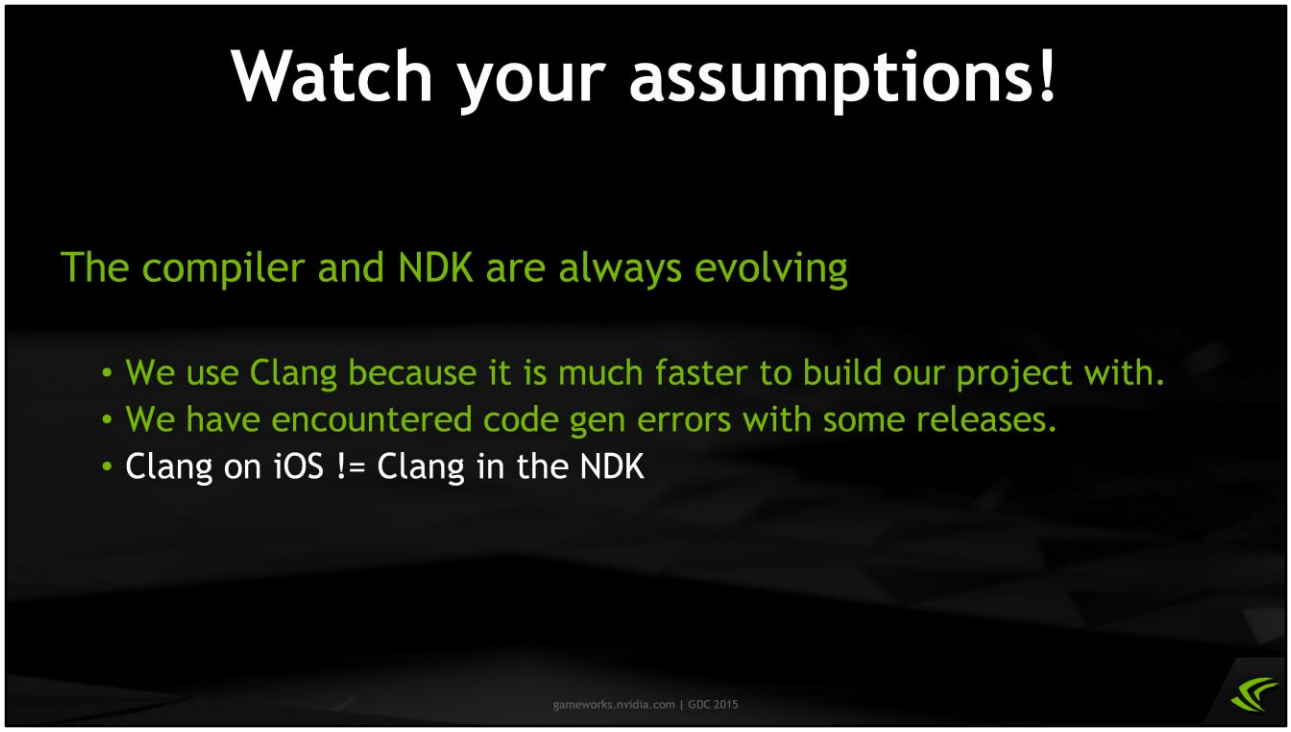

This adds together to make a difficult problem: tiering graphics heavy titles, which is unfortunately not a solved issue.

Before we decided to take android in-house, we used a heuristic to guess what the device was capable of, with varying degrees of success

Now, we've transitioned to using an online database of tiering information – when the game starts it will download the settings for that particular device and firmware

It's worked reasonably well, but feels like a scary failure point and still requires a huge amount of effort to populate the database.

It does let us push out new info for new devices, as well as solve issues that only appear in the wild or escape testing

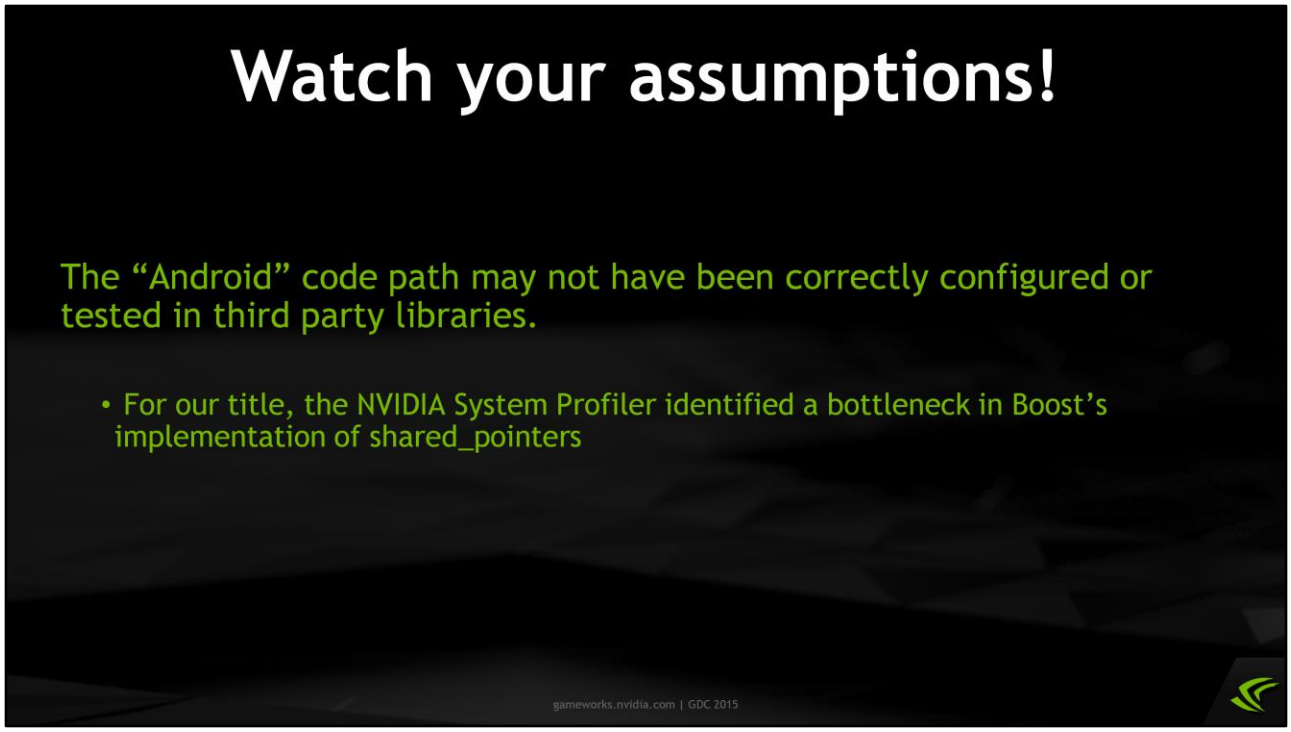



I always warn developers who are starting on Android not to get complacent about the toolchain and the language.

We often find ourselves on the bleeding edge of the NDK as we have adopted Clang for our Android titles (due to the better precompiled header system)

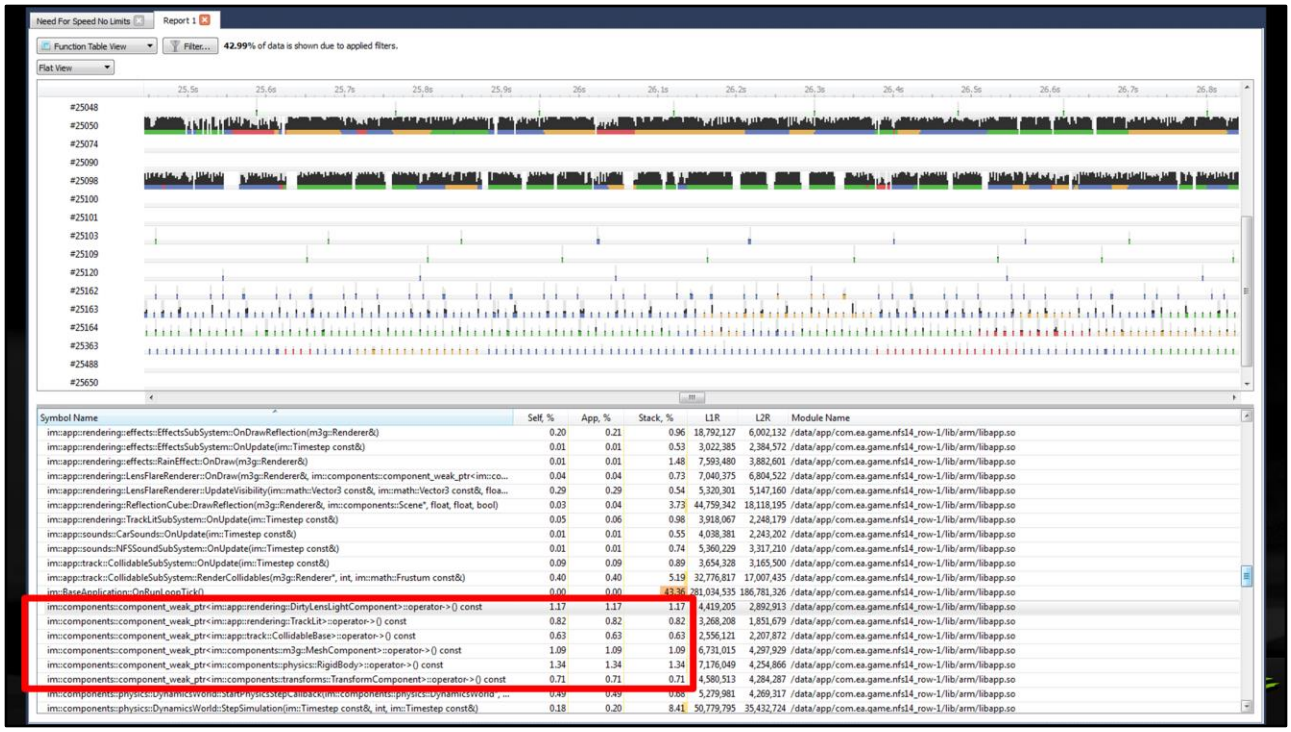
Unfortunately, we have also encountered 100% issues with code generation that have appeared in one NDK release and disappeared in the next

Another thing to bear in mind is that Clang on iOS and within the NDK are not the same – beware of codegen differences




Another concern with android is that many libraries have been poorly configured for the Android platform

- We used the Tegra System Profiler to identify an unusually large amount of time in the boost library




Everything in the red box is the boost problem.

There are even more calls that are filtered out.

That is around 6+ % combined!

Removing bottlenecks can make the overall speed sometimes go even higher but in this case it was more like a straight recoup and a far reduced % overall.

# Watch your assumptions!

The “Android” code path may not have been correctly configured or tested in third party libraries.

- For our title, the NVIDIA System Profiler identified a bottleneck in Boost’s implementation of shared_pointers
- It turned out that it was incorrectly using a slow path for multithreaded reference counts



- It turned out that the android configuration was not using atomic operations and fell back to a slow path for multithreaded primitives


Remember how I said that we typically don't use Java? Well unfortunately we can often find ourselves with no choice.

# Java and Native Apps

JNI may not be your *friend*, but it isn't out to harm you ☺
e.g., useful for Google Play Services

Interop with native does have pitfalls:

- Each native thread needs to initialize JNI
- The classloader from the main thread should be reused everwhere
- Watch out for exceptions!
- Be aware of the process model
  - Native threads may survive termination of the Java activity

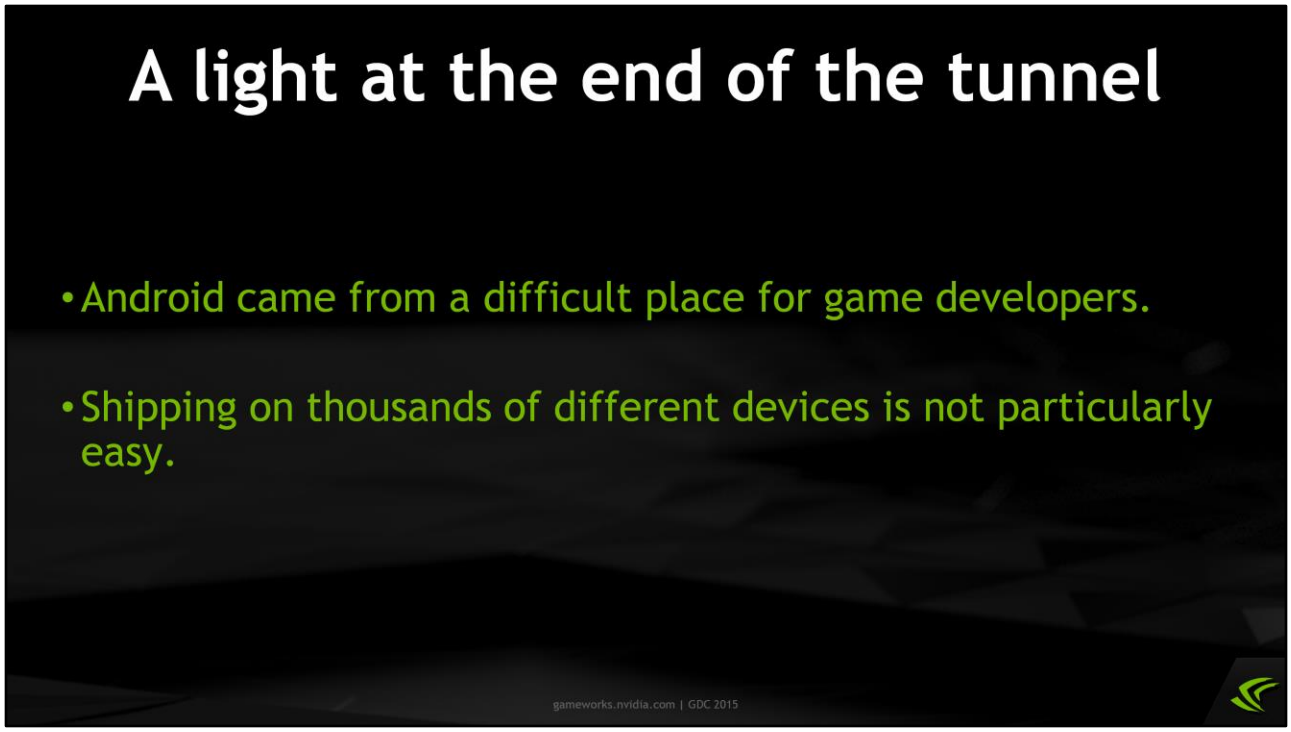



Java APIs are the primary way that some core features are made available, such as the google play services, many third party engagement or ad libraries, and new OS features

But it's not all bad, we can use JNI, the java-native interface to communicate between native code and java code.

There are a few pitfalls to avoid; hopefully some of these are already well known

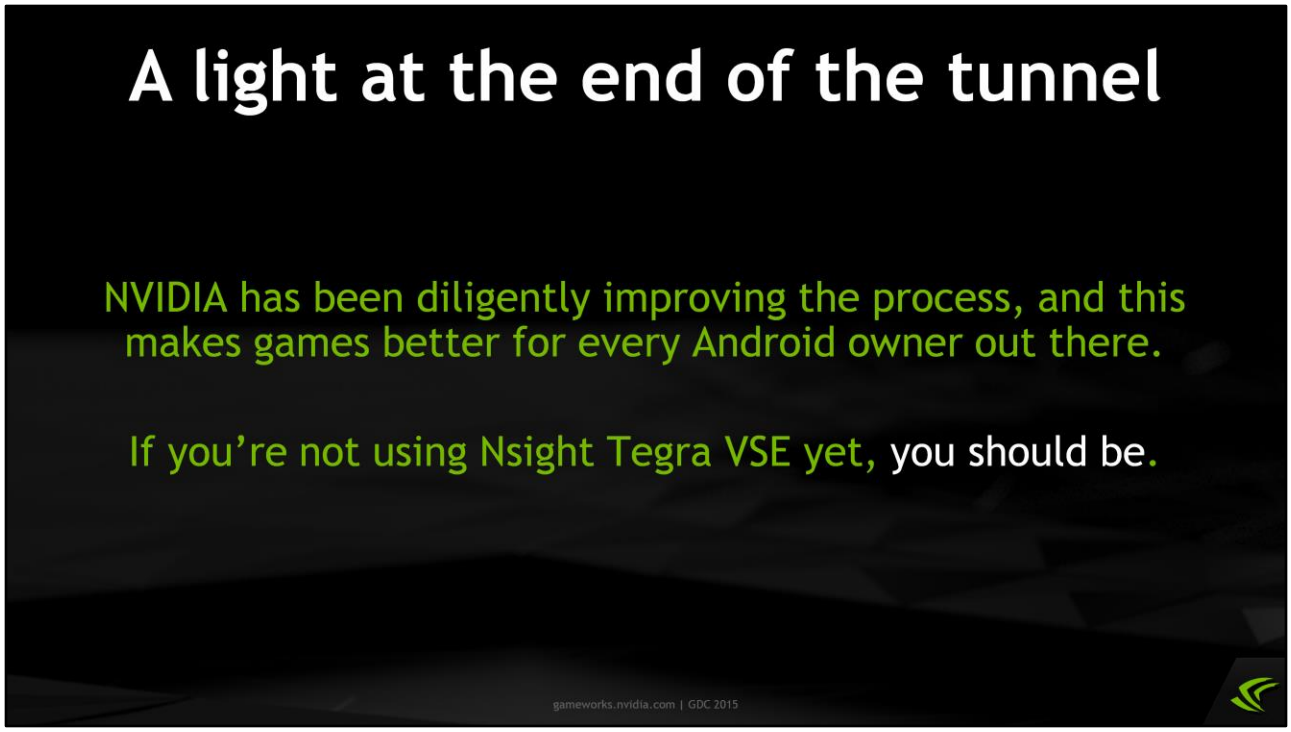
- JNI needs to be initialized per native thread created
- And the classloader from the initial thread is the one you should be using, so you need to share it between your native threads
- It's also important to remember that if an exception is raised in Java and you don't pick it up, it'll cause a crash the next time you invoke a JNI method
- Finally, the Android process model can get a bit confused with mixed Java and native applications – it is possible to end up with your Java application terminated while your native code continues to run. This is especially confusing as the native threads get can be re-attached to a new application instance, which is highly unlikely to be what you wanted




So where does all this lead us? Well, Android was – I would be the first to admit this – not a great platform for game developers.

It also has challenges of its own that are unique to having such an open ecosystem.

I think in a lot of cases, the difficulties we, the development community, had with Android was reflected in the quality of the porting efforts.




There really is a light at the end of the tunnel, however. NVIDIA is investing in making Android better for every Android owner and game developer out there, and they're really trying to make it a fully viable gaming platform.

I'd like to end with a single suggestion: if you're a game developer, and you're not using Nsight Tegra VSE yet, you really should be.

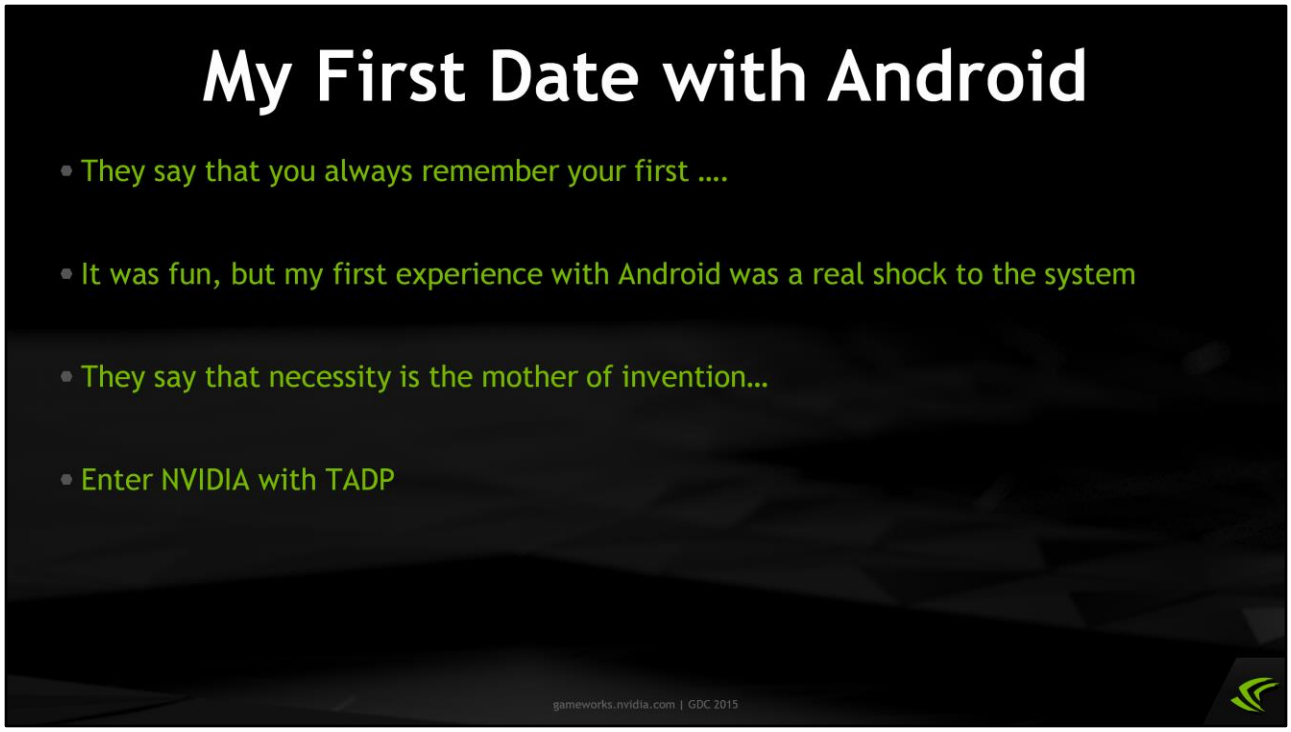

This is a session about some of the cool tools NVIDIA has developed for android developers – specifically as it relates to our most recent title, Need for Speed: No Limits


Introduce yourself

Who am I? **Say a bit about myself – keeping it brief.**

Mention Jonathan Seymour – Senior Producer – attending GameConnection this week


We work on our own projects and we consult and develop games with partners in the industry




- **Started developing games when in teens**
- **Dev tool chains improved over time and now dev teams are spoilt**

- **Asked to port game from iOS to Android in late 2009 – not using Visual Studio was like losing old friend (sad face)**
- **Could have tried to make it work – could have been so beautiful!! – could have had great relationship with Eclipse and Android NDK, but…**

- **We missed Visual Studio too much! Hacked together workable toolchain – not perfect and cracks started to show**
- **Treasured the rare moments when we could single step debug**

- **By chance we were introduced to our now good friends at NVIDIA**
- **Discovered TADP**

- **Debugging and developing using TADP has been a match made in heaven – we've never looked back**
- **If you are developing for Android you need TADP**

# Nsight Tegra

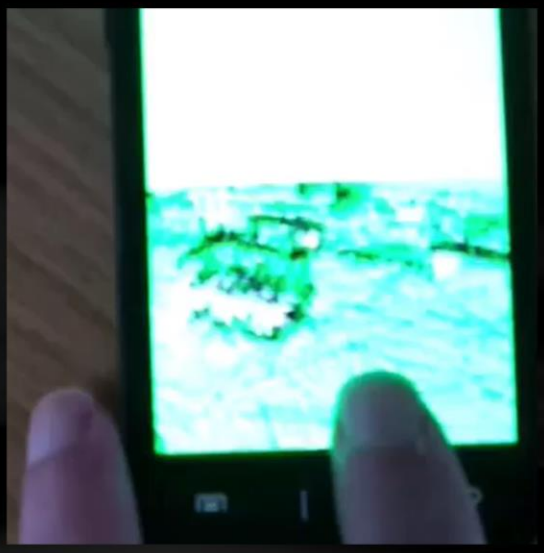
- Awesome!  That's what we use.
- But Lewis covered it.
- So on to other tools...

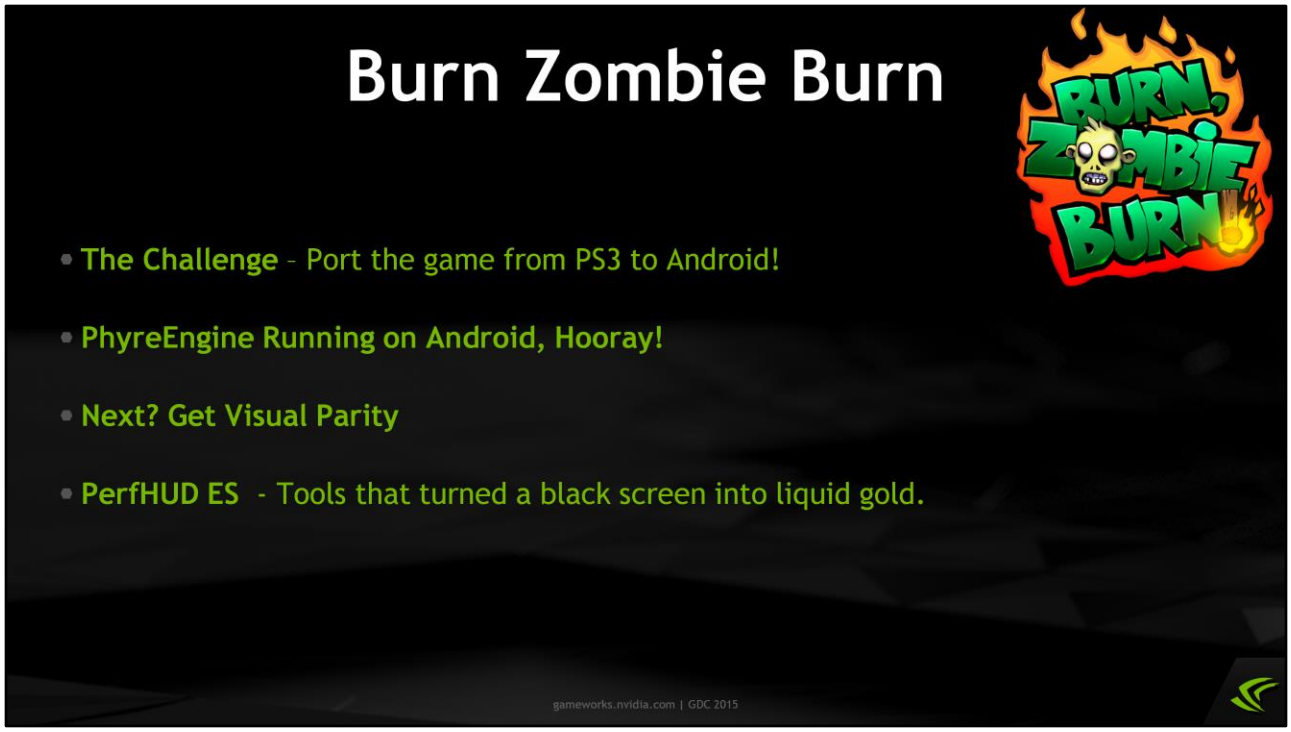



- **BZB – PS3 game by Double6 developed in 2009**

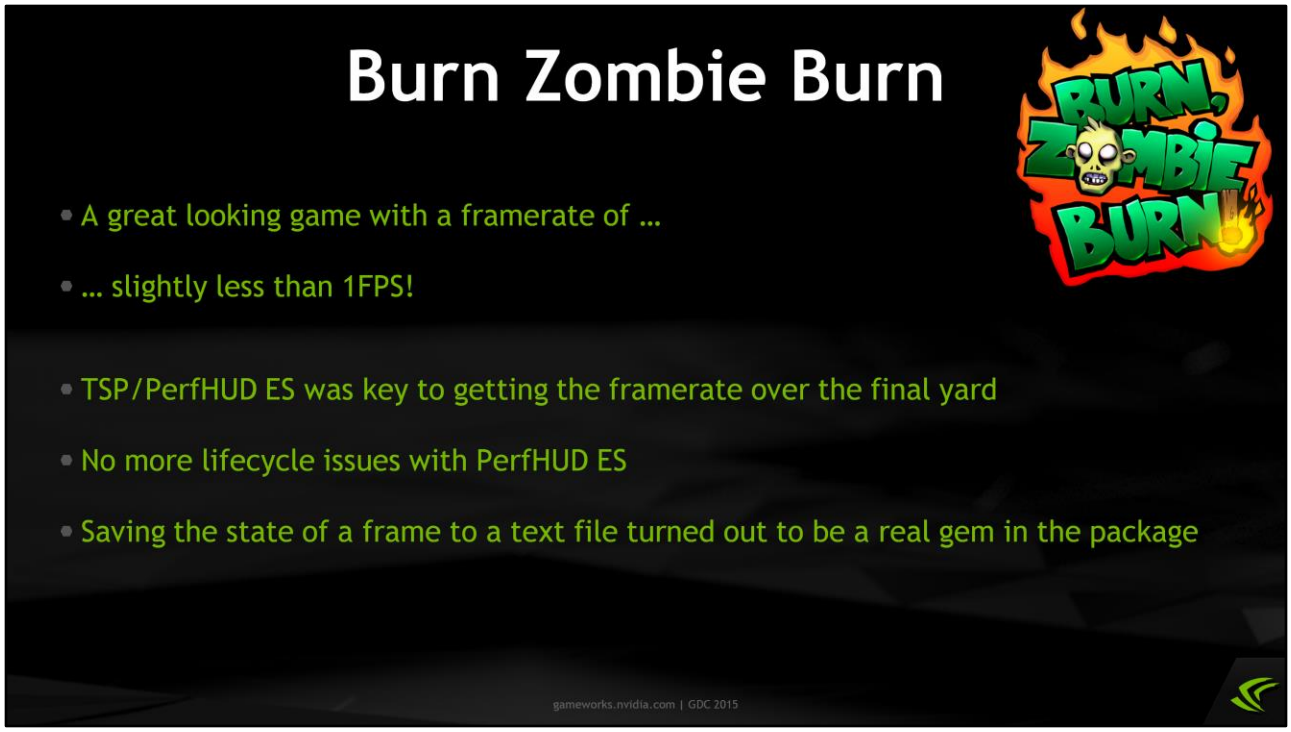
- **Compiled without TADP, game used Sony Phyre Engine we'd ported to Android. Game was running, could hear sound, but only saw black screen**

# Burn Zombie Burn






- **Plan to get parity with PS3 version, then worry about framerate**
- **PerfHUD tools meant we could analyse GL call stack – took us from black screen to almost perfect visual parity in only a few weeks**
- **Time consuming process – 100's of shaders embedded in assets needed to be converted from CG and hand optimised to GLSL**
- **PerfHUD made process much faster and less complex to visualise**




** So, we had a great looking game running at ….

- * slightly under 1fps!
- CPU bound, quick sweep timing main loop = Death by 1000 cuts
- Tegra System Profiler
- Phyre engine was sorting geometry for depth order ☺ but taking 80ms to do so ☹
- Solution = pre-sorting; rinse and repeat. Remove string compares for hash comparisons, rinse and repeat
- Moving skinning from CPU to GPU was not in scope of project
- Using profiler load on CPU was reduced by almost 35 times the original code with minimal loss of precision
- Rapidly became GPU bound and CPU issues gone ☺
- Cool trick with PerfHUD – how to resolve loss of GL context – lots of problems in BZB with this
- Phyre Engine had no need to handle loss of graphics context. Needed to add this in – quickly worked out saving state of a frame to a text file (feature in PerfHUD) then doing same after a restore context and comparing text files allowed us to see what we had not properly restored.
- Did NVIDIA have this in mind when they created the tools – who cares! It saved us loads of time, and we've been using this technique ever since. Neat!

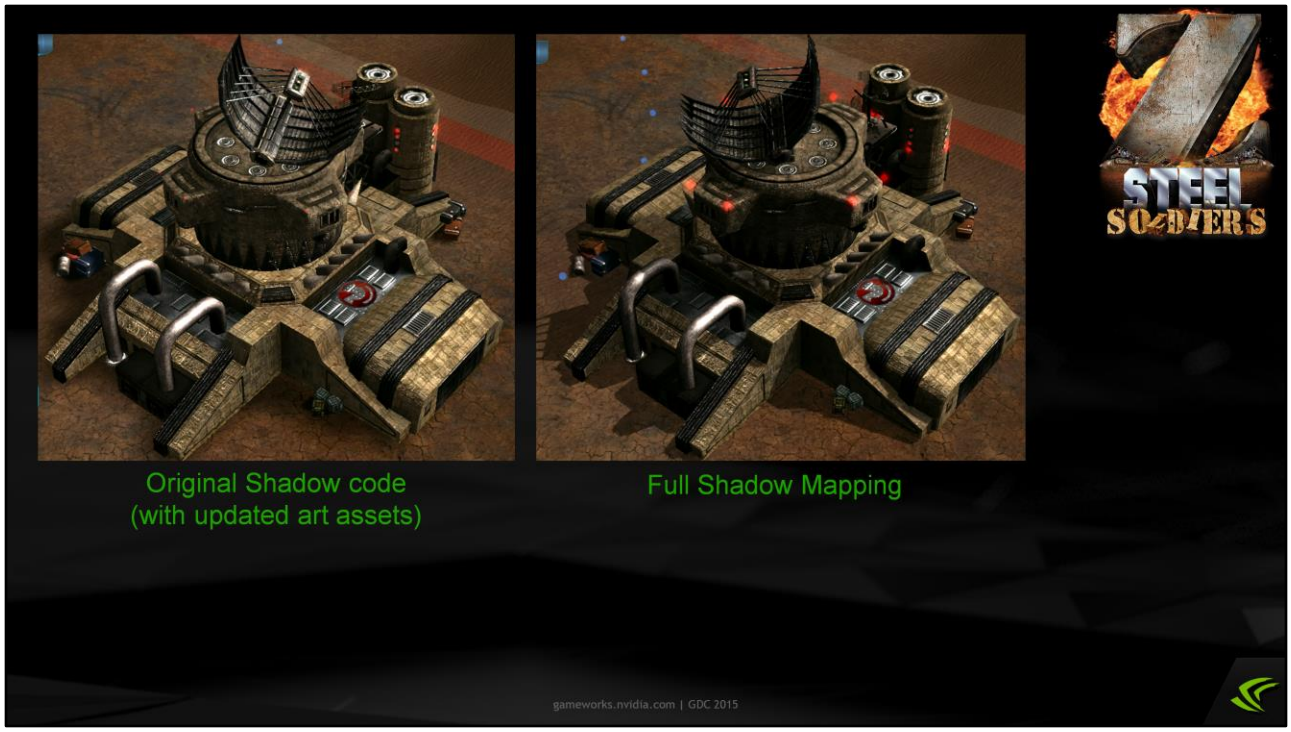
# Burn Zombie Burn


- **Z: The Game – released in 1996, competing with Command & Conquer – huge success, but Bit Bros wanted sequel to be first 3D RTS – Z Steel Soldiers released in 2001 – See screenshots**

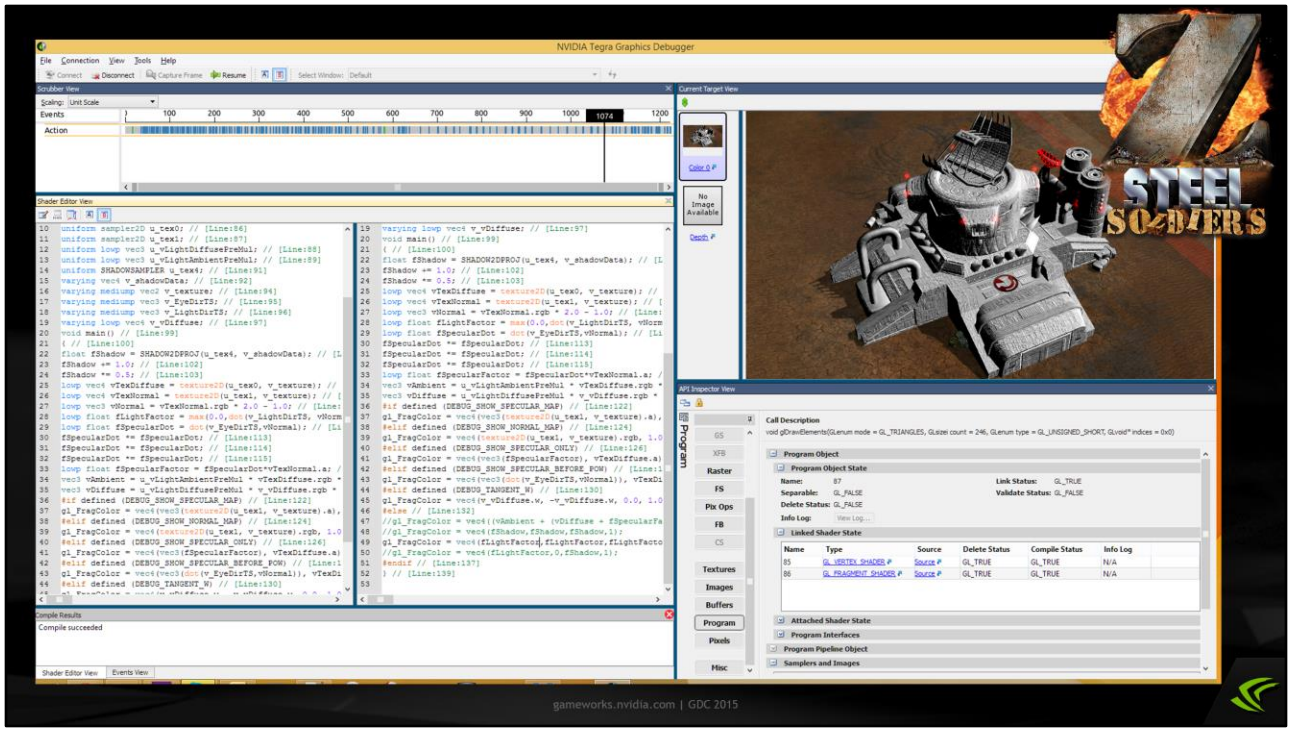
- We had the honour of porting the classic Z the Game. At the time, it was competing with Command and Conquer and while it was a huge success, the developers had huge ambitions and wanted the sequel to be the first 3D RTS. Remember we've gone back now to the year 2001 – it was no small thing to get this up and running. Of course, now the screenshots are vintage quality, but this was development that was taking place in the late 90's.


- **Z: The Game – released in 1996, competing with Command & Conquer – huge success, but Bit Bros wanted sequel to be first 3D RTS – Z Steel Soldiers released in 2001 – See screenshots**

- We had the honour of porting the classic Z the Game. At the time, it was competing with Command and Conquer and while it was a huge success, the developers had huge ambitions and wanted the sequel to be the first 3D RTS. Remember we've gone back now to the year 2001 – it was no small thing to get this up and running. Of course, now the screenshots are vintage quality, but this was development that was taking place in the late 90's.




- **Like with BZB we wanted visual parity – we knew that with TADP we had the tools to go the extra mile on framerate**
- **Programmable pipeline – we could consider normal maps, shadow mapping & post process effects**
- **Code that was 20 years old stood test of time as piece of software engineering**
- **TSP instrumental again in reducing the CPU load – could fix bottlenecks in code within hours rather than days.**
- **Experience with BZB/Sniper rifle accuracy at finding problems PLUS TSP saved us weeks of work**
- **Tegra Graphics Debugger – lead to irresistible urge to add further enhancements – TGD made this so easy.**
- **Boosted shadow quality repeatedly**
- ***Show image of old shadows, then image of new shadows "z2_screenshot_old_shadows.png", "z2_screenshot_old_shadows.png"***



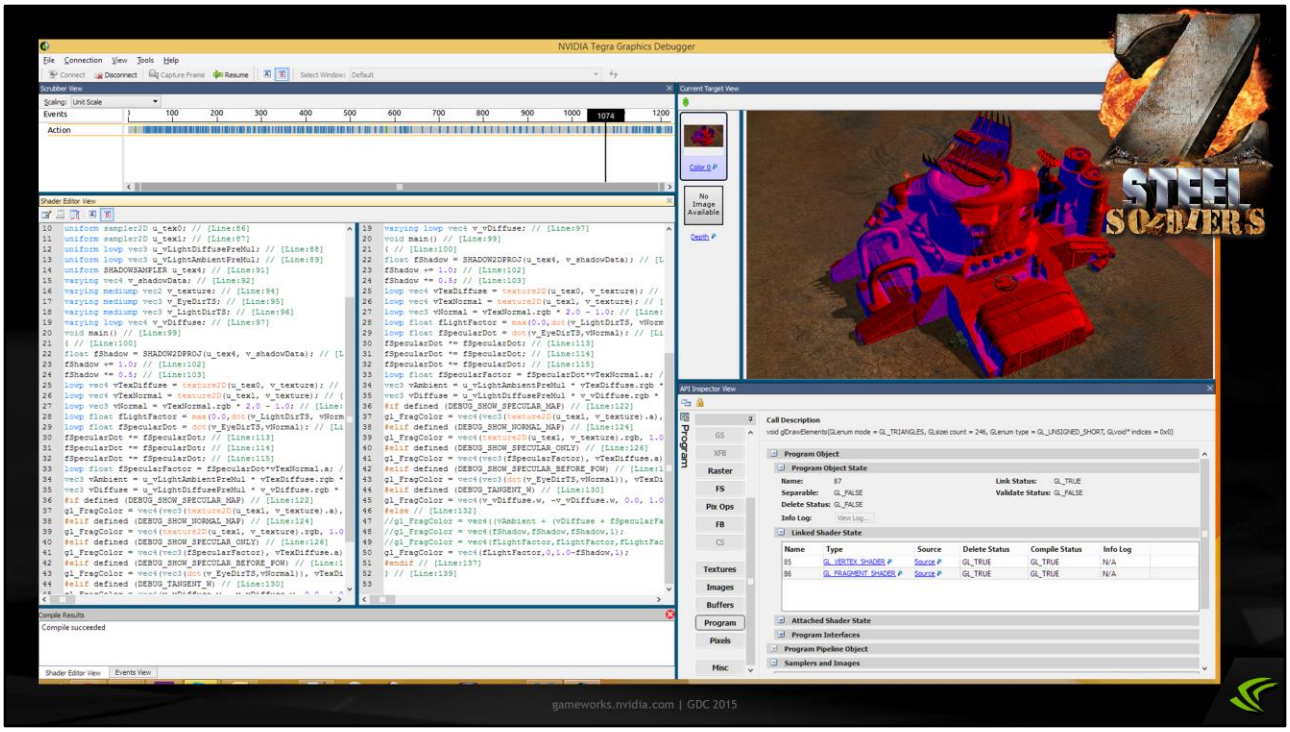

- **improved lighting**
- *z2_gfx_lighting.PNG*
- editing a shader to visualize lighting with normal mapping

- **added more particle effects and specular maps – Difficult to stop**
- **REALLY EXCITING TIME – I attribute this to TGD**

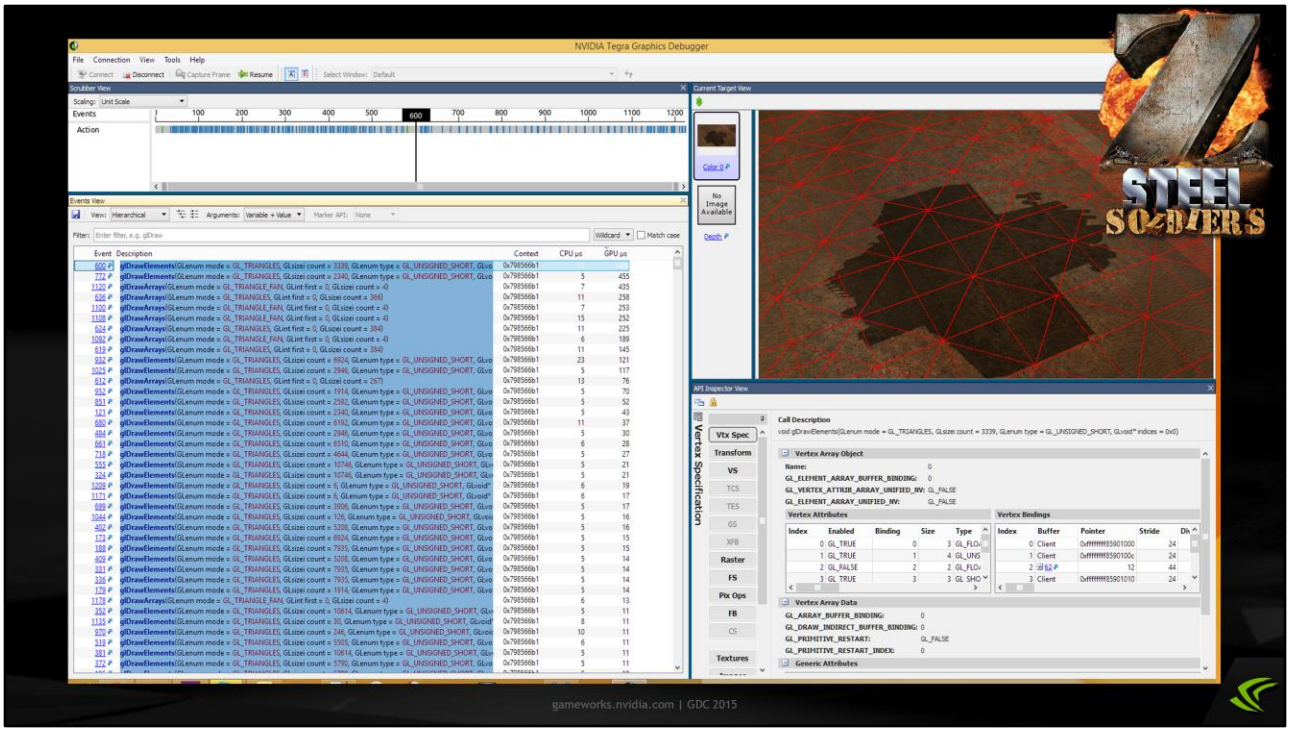
- (z2_gfx_most_expensive_draw_call.PNG, z2_gfx_second_most_expensive_draw_call.PNG)
- (z2_gfx_no_mipmaps.PNG)

*Talk about what we did to use the tools to give us better lighting effects.*

*Next slide shows how we used the tools to optimised our shaders for shadows and lighting*

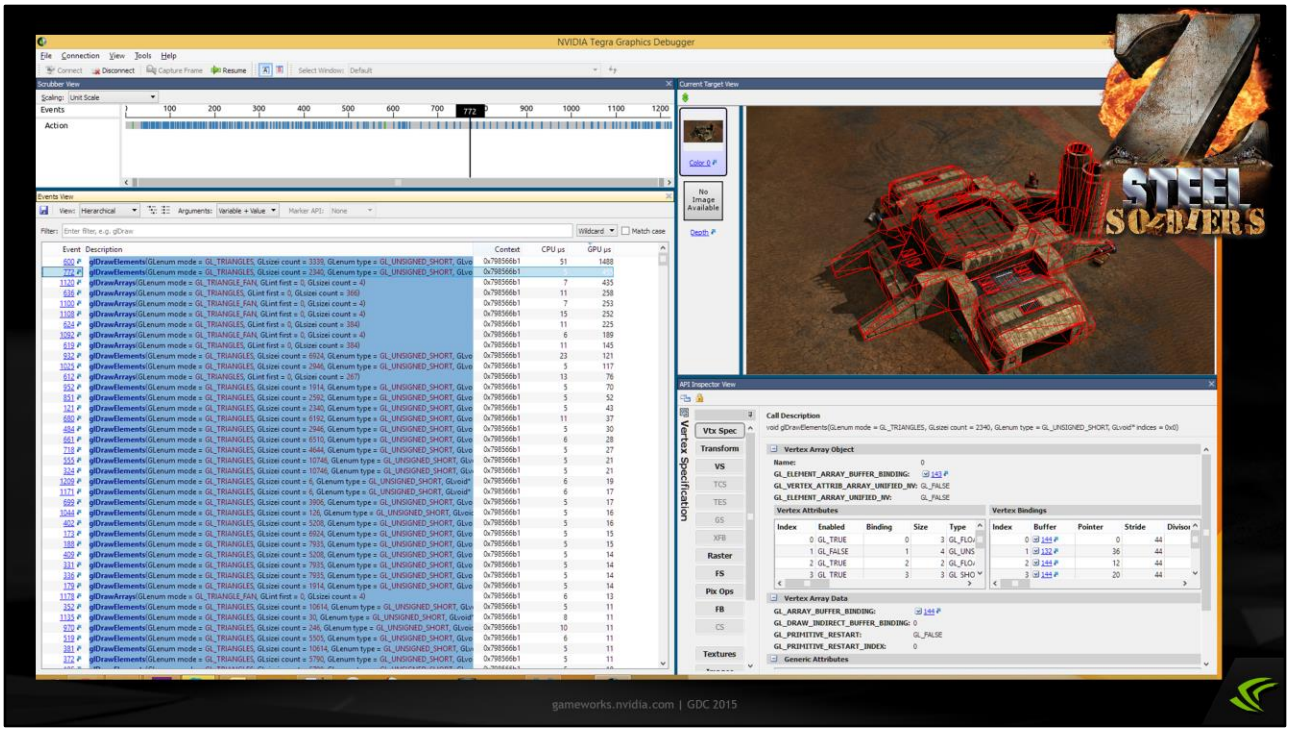



*This slide shows how we used the tools to use AB testing to visualise what was happening under the hood*



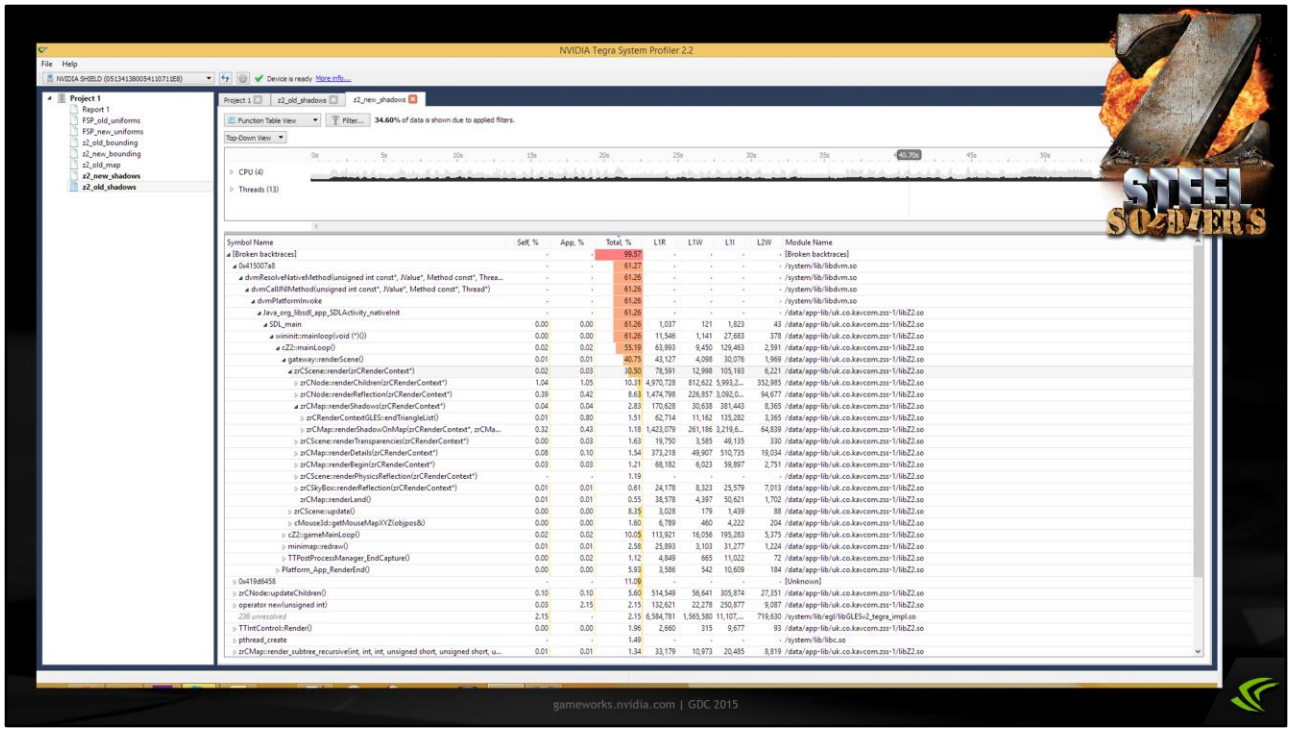

*z2_gfx_most_expensive_draw_call.PNG*

ordering draw calls to find the most expensive (terrain)



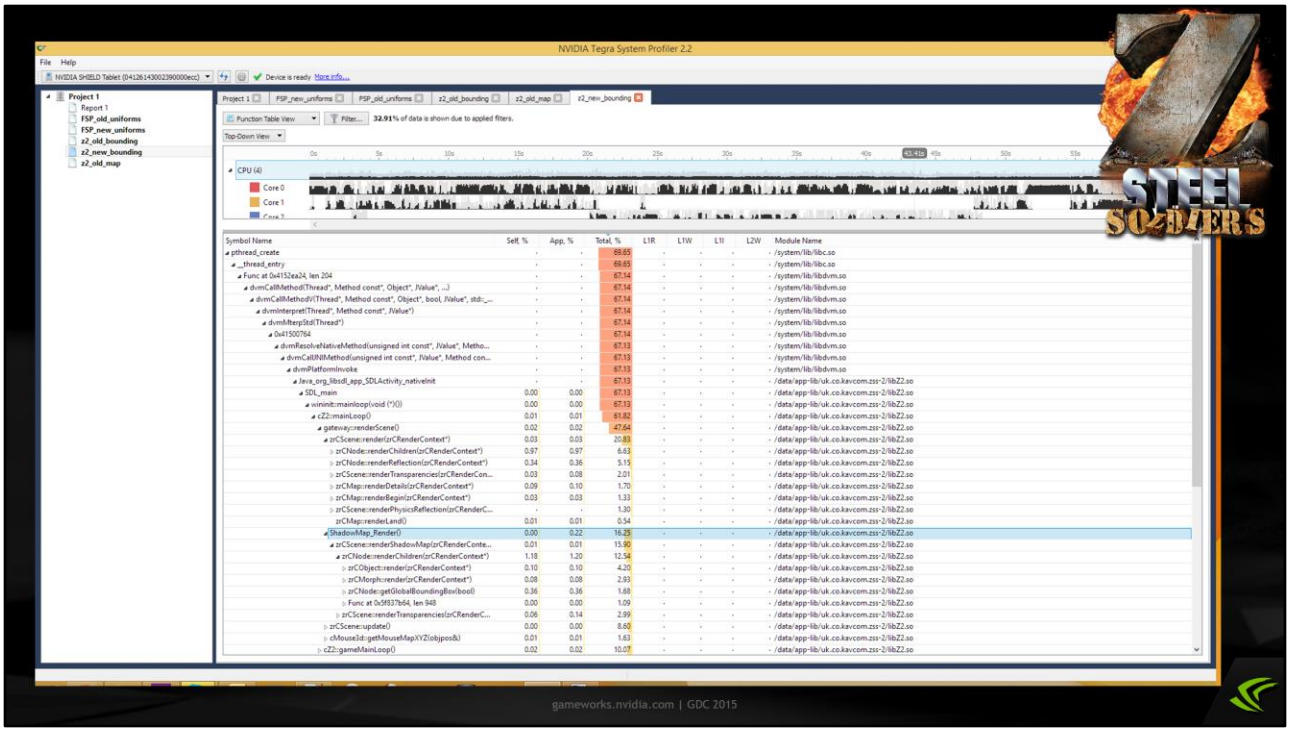

*z2_gfx_second_most_expensive_draw_call.PNG*

second most expensive draw call (command centre)



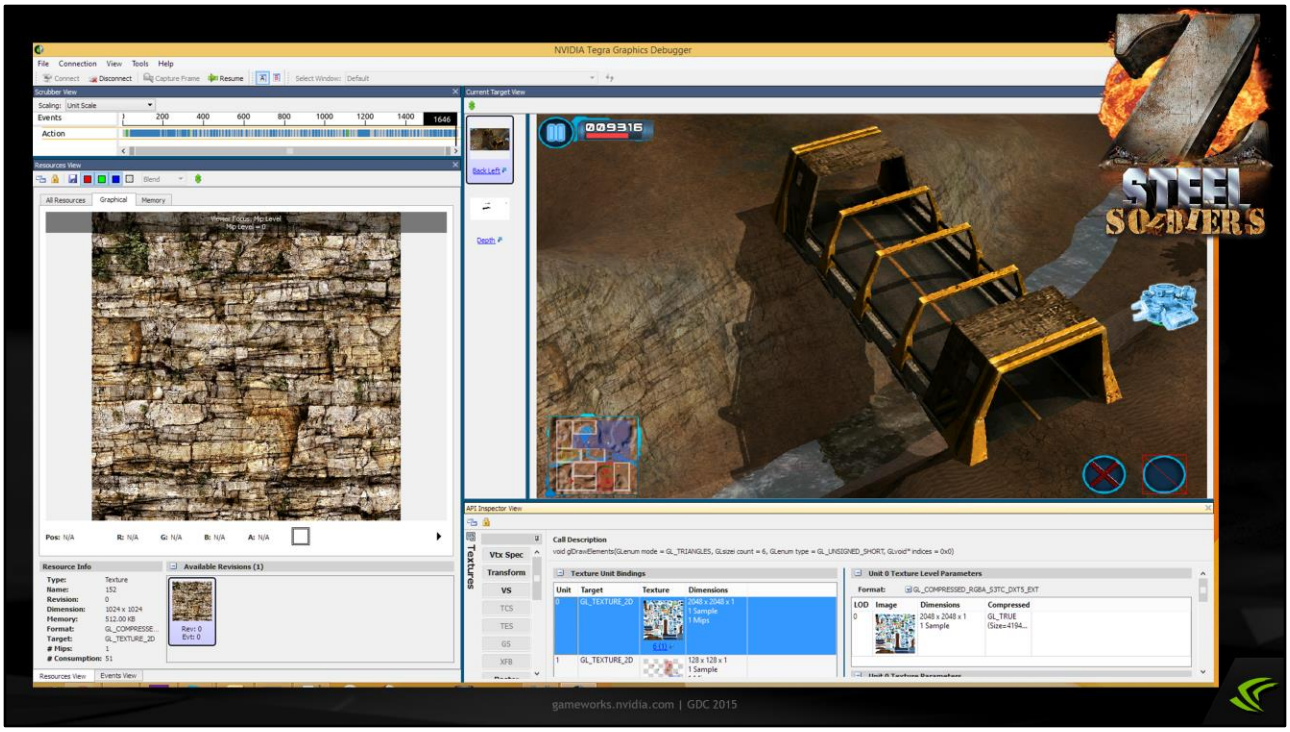

*z2_sys_old_shadows.PNG*

System profile of old shadow system enabled.




*z2_sys_new_shadows.PNG*

System profile of new shadows being rendered




*z2_gfx_no_mipmaps.PNG*

Showing that we noticed a texture with no mipmaps using the tool

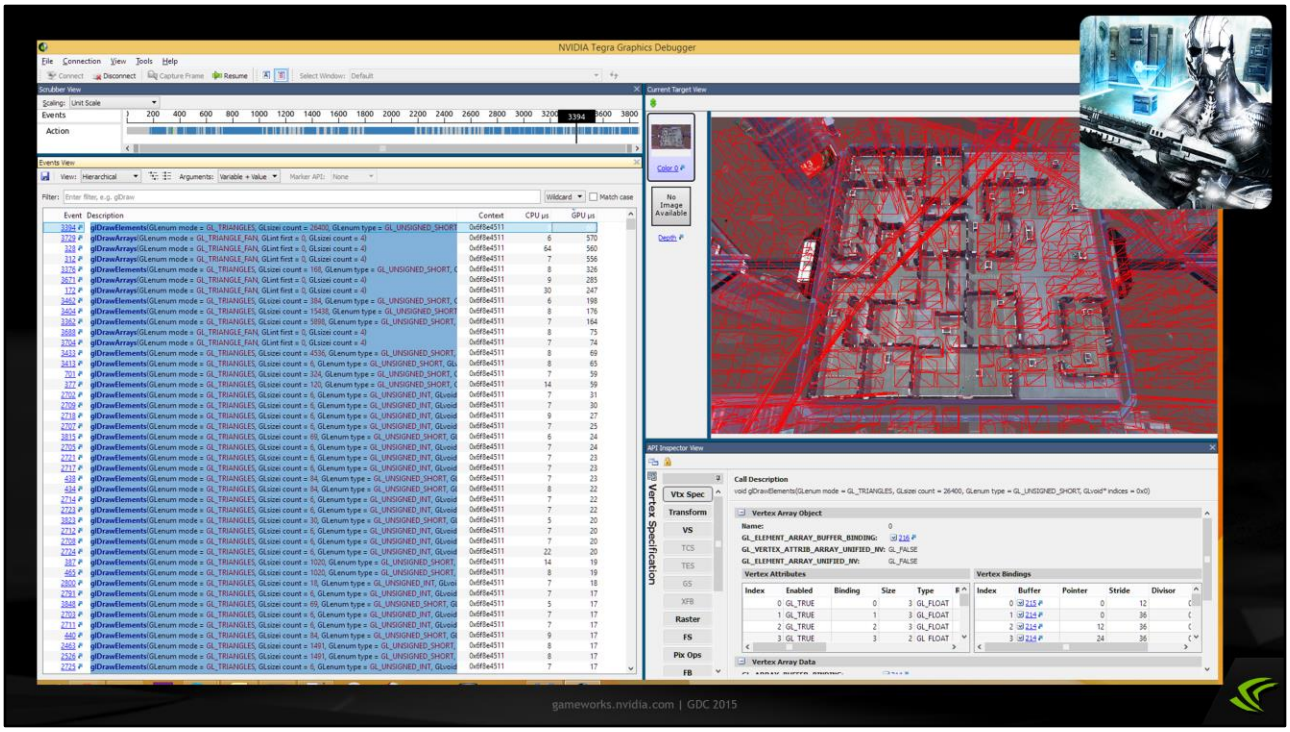
# Z: Steel Soldiers

# Frozen Synapse Prime

- Frozen Synapse Prime was a different beast – a real head on test of the TADP tools
- TGD/TSP – a match made in heaven
- A/B Shader Testing Vital again!
- PerfHUD ES compared to TGD
- Understanding the rendering pipeline and where are immediate issue were

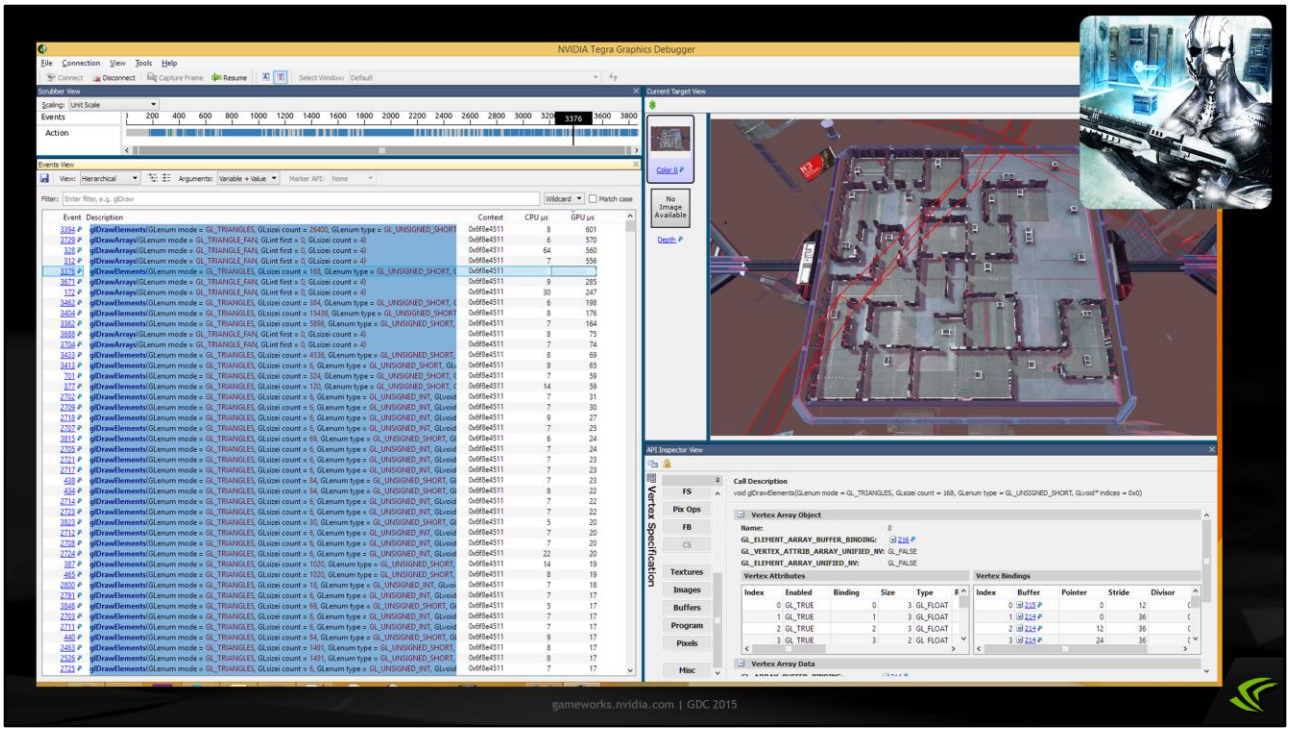


# Frozen Synapse Prime



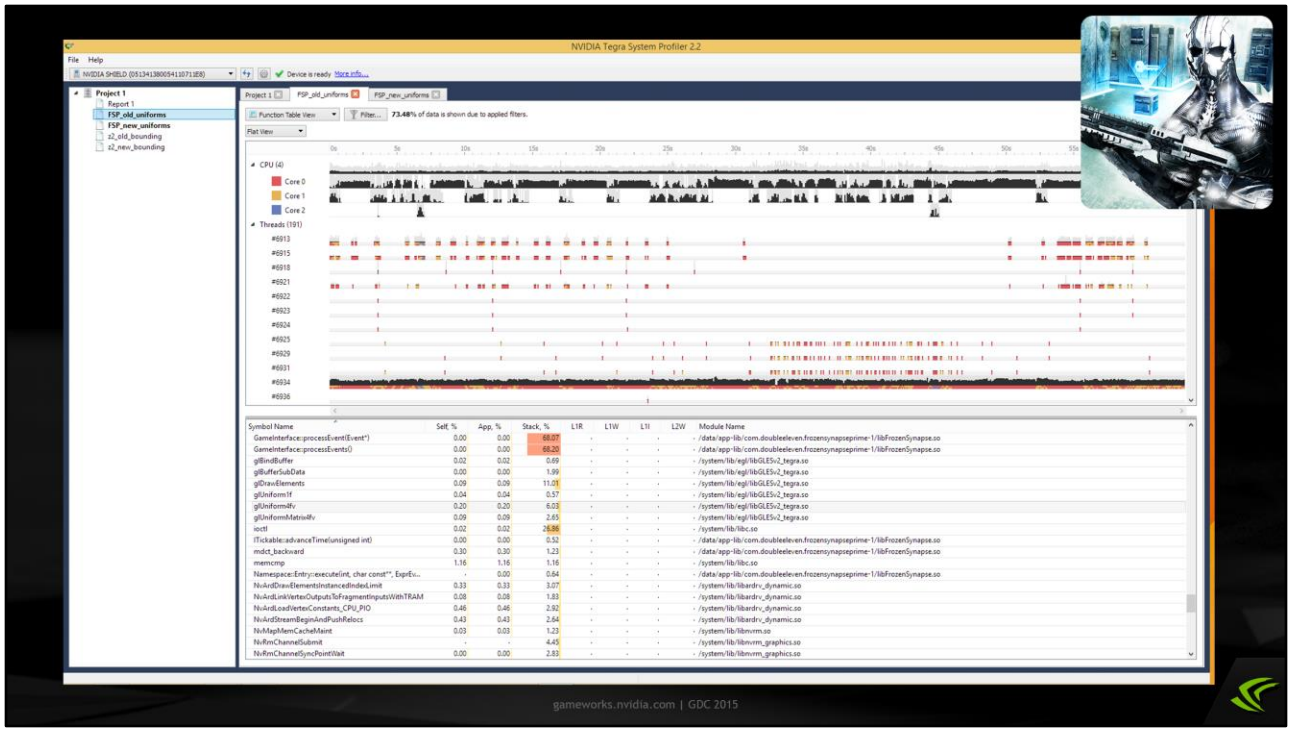

*fsp_gfx_slowest_draw_call.PNG*

Showing slowest draw call is the cityscape beneath the playfield (Don't forget this is after change)



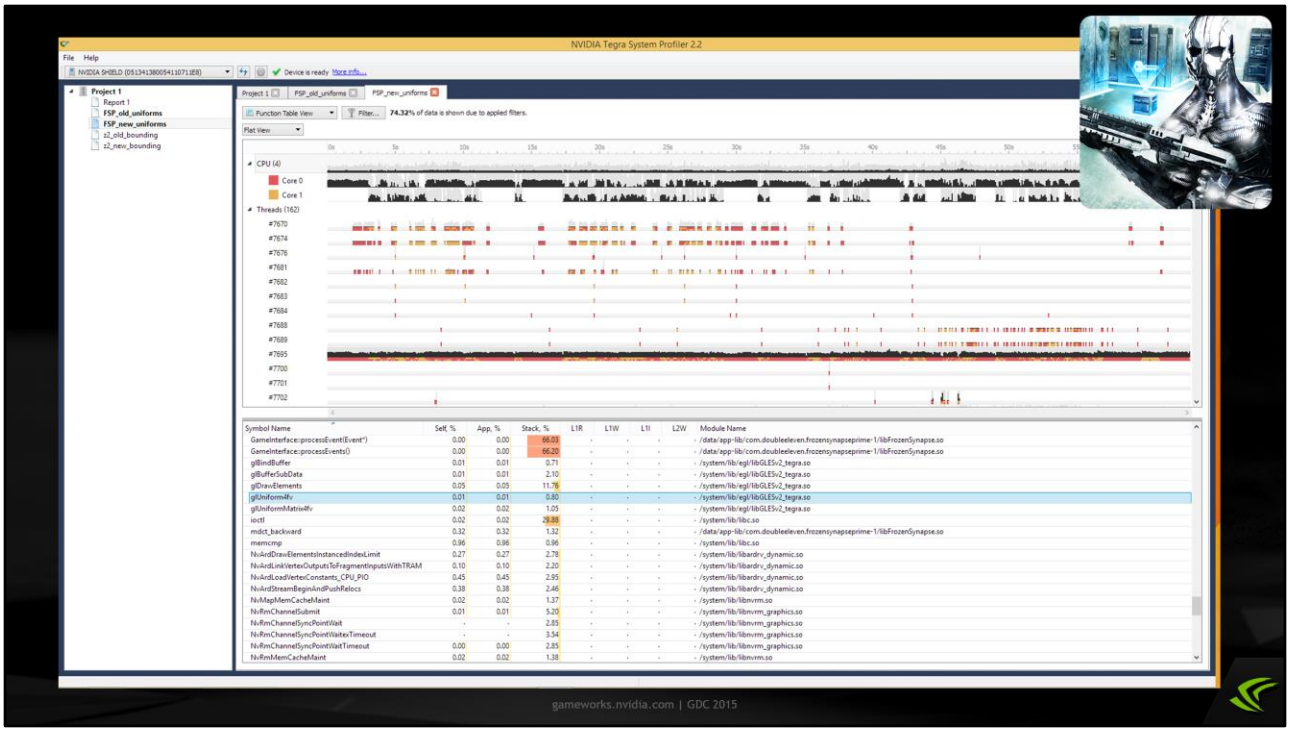

*fsp_gfx_surprisingly_slow_call.PNG*

A draw call drawing over the roads shows up as being quite slowest



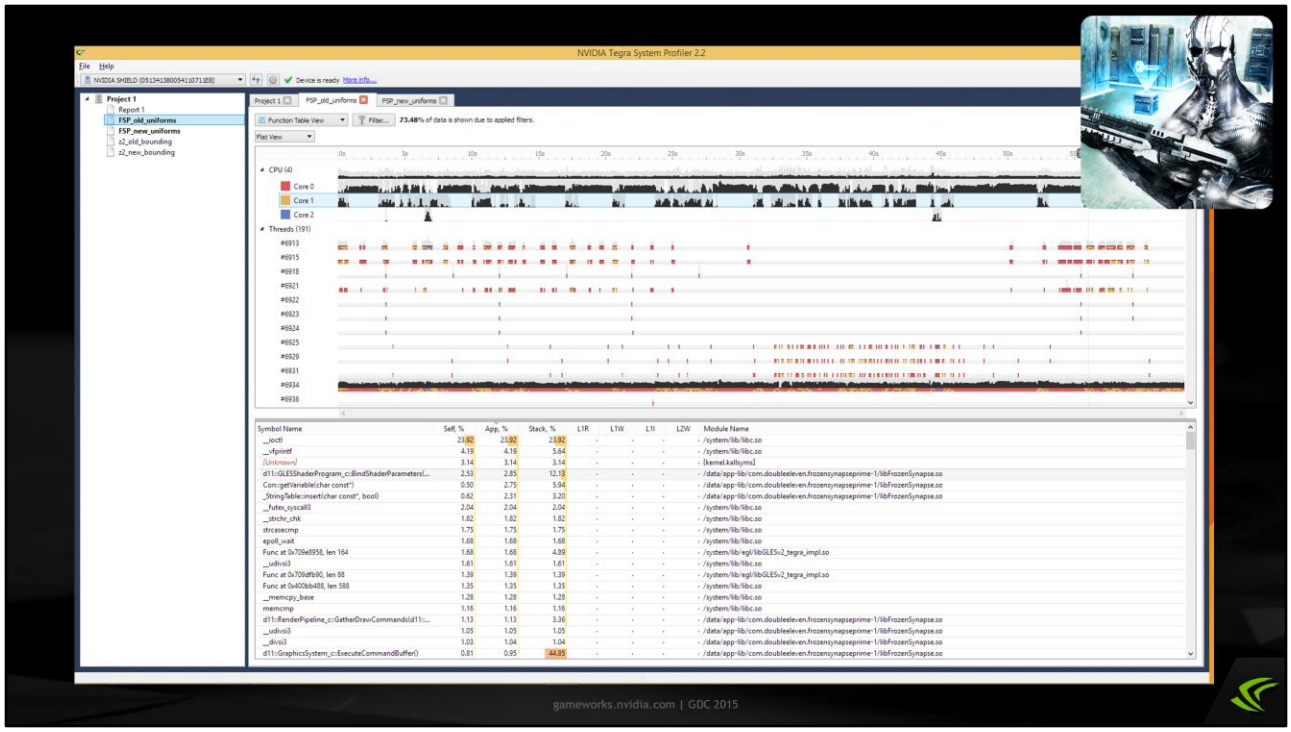

*fsp_sys_old_uniforms1.PNG*

Flat view showing gl calls with no uniform optimized



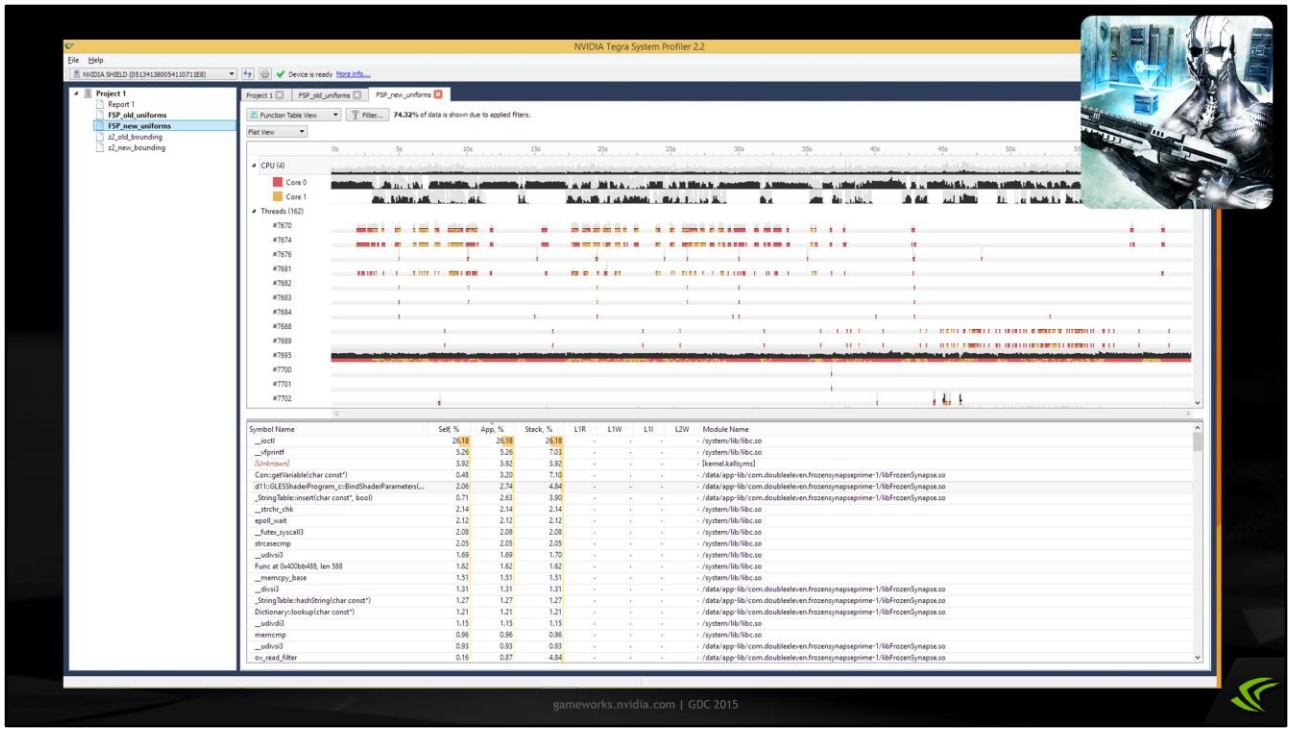

fsp_sys_new_uniforms1.PNG

Flat view showing gl calls with new uniform optimization on




fsp_sys_old_uniforms2.PNG

flat view ordered by cpu time showing constant setting time before optimizations




fsp_sys_new_uniforms2.PNG

flat view ordered by cpu time showing how new uniform optimization improved constant setting time

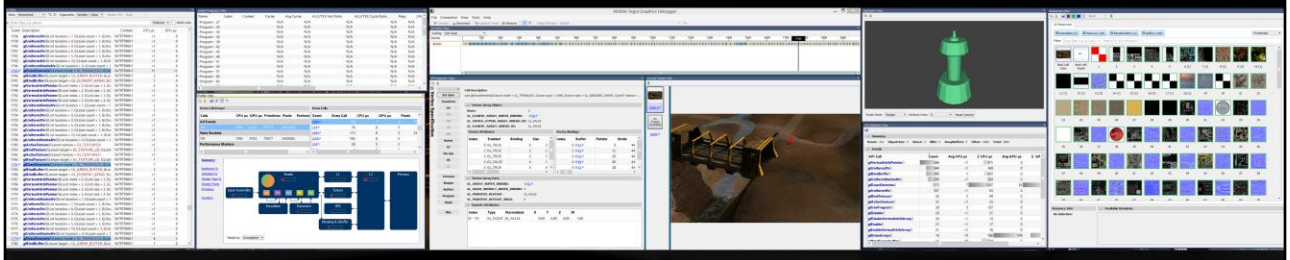
# TSP Favourite Things

- Quick to find if the bottleneck is in our code or in the system
- A great way of measuring whether our experiments have had an impact
- Precise visualisation of thread utilisation.



# TGD Favourite Things

- The UI gave me an excuse to move from two screens to three.  Makes you feel like a NASA Commander. Epic!



# TGD Favourite Things

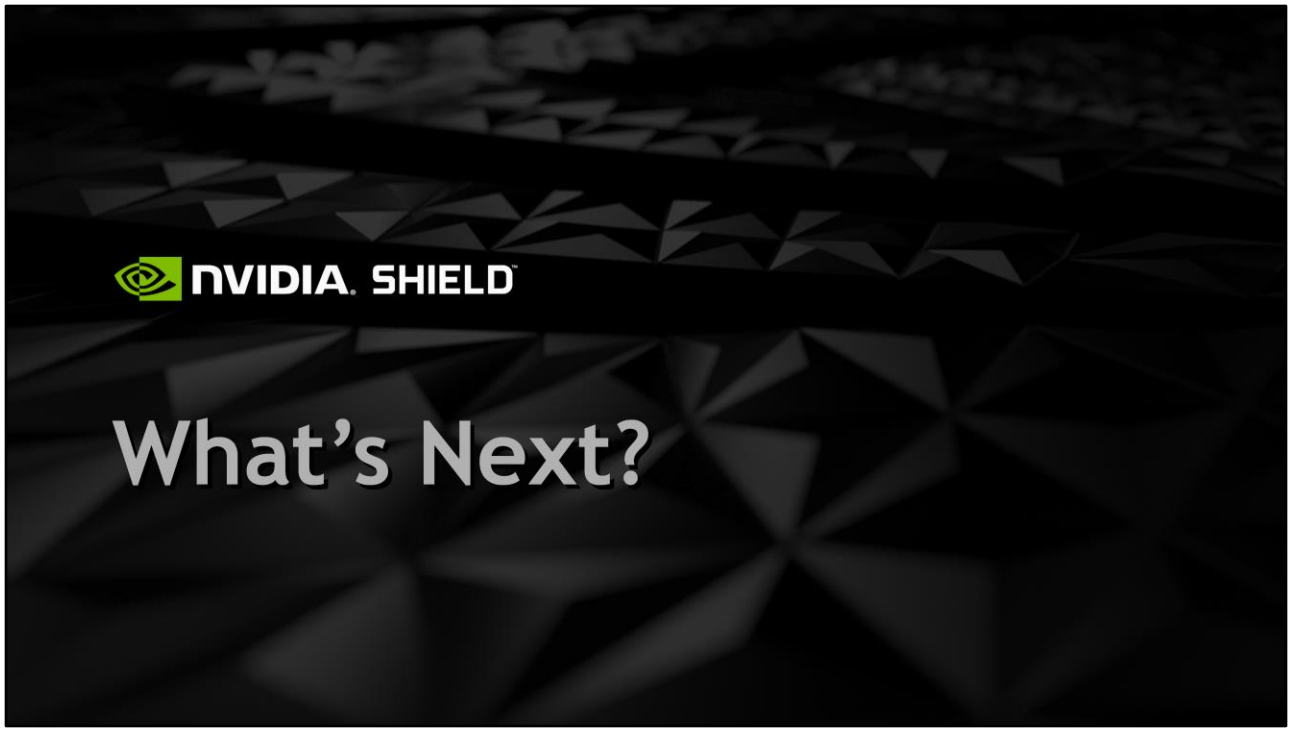
- The UI gave me an excuse to move from two screens to three.  Makes you feel like a NASA Commander. Epic!
- Great tools for A/B Shader testing, with a fast turn around on ideas, bug fixing and optimisation.
- Gives you a really quick way to find most the expensive draw calls.
- The more you use it, the more features you discover. Simple to work with but with enough depth to tackle more complex problems.
- Quick Tests… Quick Decisions



# TGD Getting up to speed

- Check out Daniel's and Jeff's YouTube video from GDC 2014 for getting up to speed fast on these tools.
  - http://goo.gl/PgFlCP

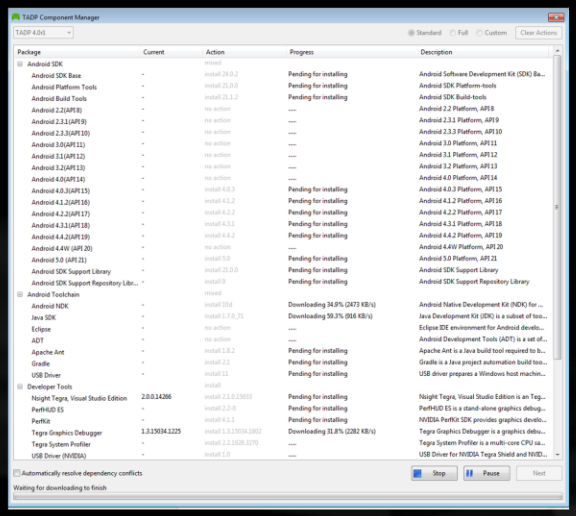


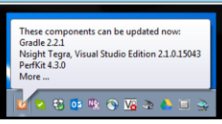



NVIDIA

# TADP 4r1

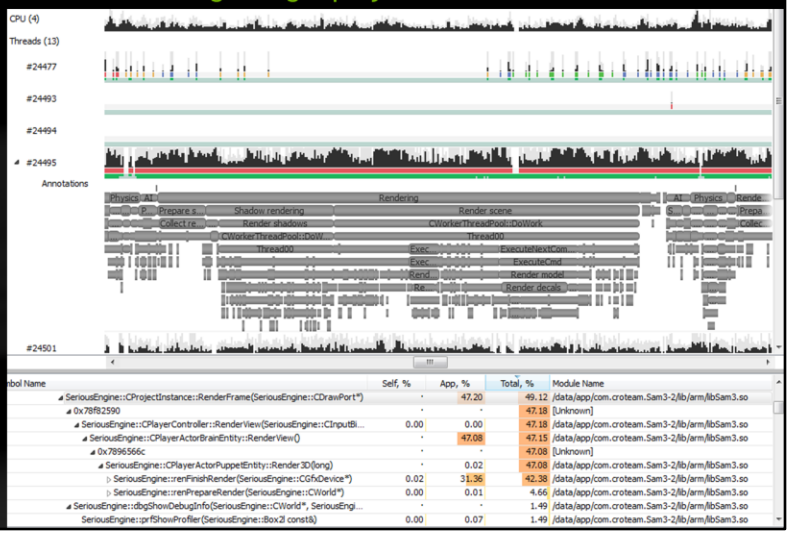
- Tegra X1 and SHIELD support
- Component based installer
  - Upgrade individual parts
  - Concurrent downloads
  - Update notification
- Nsight Tegra
  - Multi-platform APKs
  - Compile x86/x64 and debug on virtual devices
- Tegra System Profiler
  - Improved backtracing
- Tegra Graphics Debugger
  - Significant performance enhancements



# TADP 4r2

- Tegra Graphic Debugger
  - Frame captures can generate source code and Nsight Tegra projects
  - OpenGL 4.5
- Tegra System Profiler
  - Tracing via NVTX APIs
  - CPU Frequency
  - Thread States
- Nsight Tegra
  - Performance




The picture is from Serious Sam 3, showing their internal tracing profiler redirected to Tegra System Profiler via NVTX so that it can be visualized and correlated back to CPU activity.

I would like to thank Croteam.

# Questions?

NVIDIA Developer Tools
devtools-support@nvidia.com
https://developer.nvidia.com/gameworks-tools-overview
Come see our tools!  North hall, business center, back right

Lewis Strudwick
lstrudwick@ea.com
http://www.firemonkeys.com.au

Arden Aspinall
aaspinall@ticktockgames.com
http://www.ticktockgames.com



# GameWorks

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