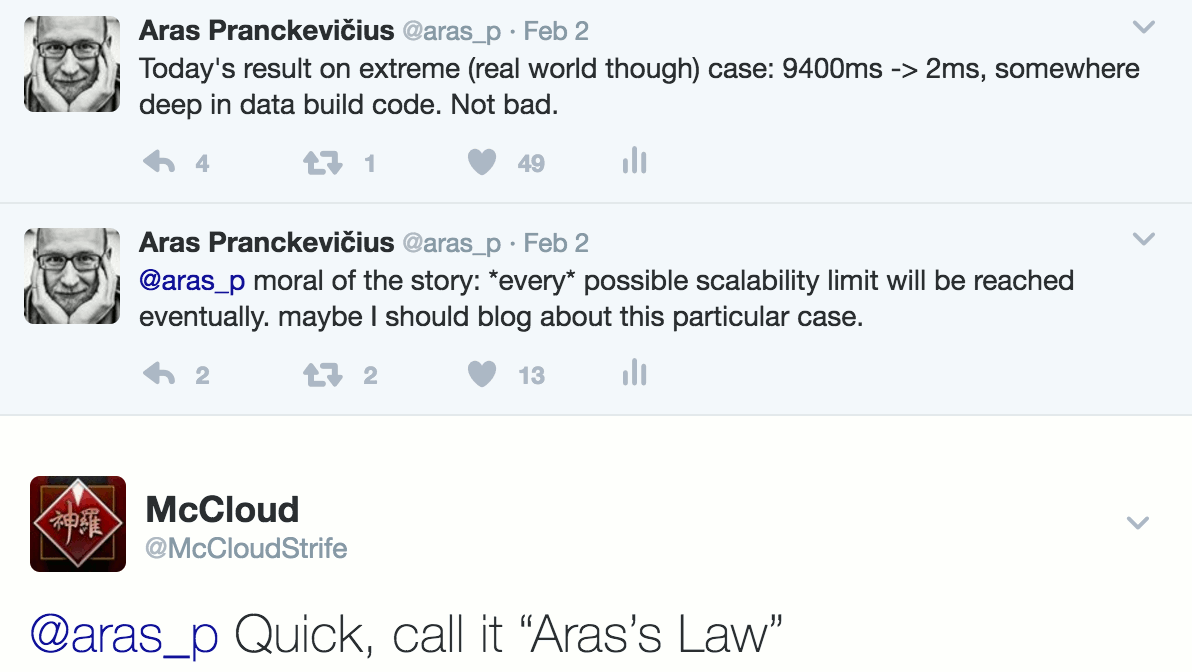
**Every Possible Scalability Limit Will Be Reached**

Posted on Feb 5, 2017

[#code](https://aras-p.info/tags/code) [#work](https://aras-p.info/tags/work) [#unity](https://aras-p.info/tags/unity) [#performance](https://aras-p.info/tags/performance)

[](https://aras-p.info/img/blog/2017-02/scalability-tweet.png)I wrote this the other day, and @McCloudStrife [suggested](https://twitter.com/McCloudStrife/status/827266539137011715) I should call it “Aras’s law”. Ok! here it is:

***Every possible scalability limit will be reached eventually****.*

Here’s a concrete example that I happened to work on a bit during the years: shader “combinatorial variant explosion” and dealing with it.

In retrospect, I should have skipped a few of these steps and recognized that each “oh we can do 10x more now” improvement won’t be enough when people will start doing 100x more. Oh well, live and learn. So here’s the boring story.

**Background: shader variants**

GPU programming models up to this day still have not solved “how to compose pieces together” problem. In CPU land, you have function calls, and function pointers, and goto, and virtual functions, and more elaborate ways of “do this or that, based on this or that”. In shaders, most of that either does not exist at all, or is cumbersome to use, or is not terribly performant.

So many times, people resort to writing many slightly different “variants” of some shader, and pick one or another to use depending on what is being processed. This is called “ubershaders” or “megashaders”, and often is done by stiching pieces of source code together, or by using a C-like preprocessor.

Things are slowly improving to move away from this madness (e.g. specialization constants in Vulkan, function constants in Metal), but it will take some time to get there.

So while we have the “shaders variants” as a thing, it can end up being a problem, especially if number of variants is large. Turns out, it *can* get large really easily!

**Unity 1.x: almost no shader variants**

Many years ago, shaders in Unity did not have many variants. They were only dealing with simple forward shading; you would write in your shader, and that would compile into 5 internal variants. And *that was it*.// autolight 7

Why a compile directive behind a C++ style comment? Why 7? Why 5 variants? *I don’t know, it was like that*. 7 was probably the bitmask of which light types (three bits: directional, spot, point) to support, but I’m not sure if any other values besides “7” worked. Five variants, because some lights needed more to support light cookies vs. no light cookies.

Back then Unity supported just one graphics API (OpenGL), and five variants of a shader was not a problem. You can count them on your hand! They were compiled at shader import time, and all five were always included into the game data build.

**Unity 2.x: add some more variants**

Unity 2.0 changed the syntax into a , so that it looks less of a commnt and more like a proper compile directive. And at some point it got the ability for users to add *their own* variants, which we called “[shader keywords](https://docs.unity3d.com/Manual/SL-MultipleProgramVariants.html)”. I forget which version exactly that happened in, but *I think* it was 2.x series.#pragma multi\_compile

Now people could make shaders do one or another thing of their choice (e.g. use a normalmap vs do not use a normal map), and control the behavior based on which “shader keywords” were set.

This was not a big problem, since:

* There was no way to have custom inspector UIs for materials, so doing complex data-dependent shader variants was not practical,
* We did not use the feature much in the Unity’s built-in shaders, so many thought of it as “something advanced, maybe not for me”,
* All shader variants for all graphics APIs (at this time: OpenGL & Direct3D 9) were always compiled at shader import time, and always included into game build data.
* I think there was a limit of maximum 32 shader keywords being used.

In any case, “crazy amount of shader variants” was not happening just yet.

**Unity 3.x: add some more variants**

Unity 3 added built-in lightmapping support (which meant more shader variants: with & without lightmaps), and added deferred lighting too (again more shader variants). The game build pipeline got ability to not include some of the “well this surely won’t be needed” shader variants into the game data. But compilation of all variants present in the shader was still happening at shader import time, making it impractical to go above couple dozen variants. *Maybe* up to a hundred, if they are simple enough each.

**Unity 4.x: things are getting out of hand! New import pipeline**

Little by little, people started adding more and more shader variants. I *think* it was [Marmoset Skyshop](https://www.marmoset.co/skyshop/) that gave us the “wow something needs to be done” realization, either in 2012 or 2013.

The thing with many shader-variant based systems is: number of *possible* shader variants is always much, much higher than the number of *actually used* shader variants. Imagine a simple shader that has these things in a multi-variant fashion:

* Normal map: off / on
* Specular map: off / on
* Emission map: off / on
* Detail map: off / on
* Alpha cutout: off / on

Now, each of the features above essentially is a bit with two states; there are 5 features so in total there are 32 possible shader variants (). How many will actually be used? Likely a lot less; a particular production usually settles for some standard way of authoring their materials. For example, most materials will end up using a normal map and specular map; with occasional one also putting in either an emission map or alpha cutout feature. That’s a handful of shader variants that are needed, instead of full set of 32.2^5

But up to this point, we were compiling *each and every* possible shader variant at shader import time! It’s not terribad if there’s 32 of them, but some people wanted to have ten or more of these “toggleable features”. gets to a *really large number*, really fast.2^N

It also did not help that by then, we did not only have OpenGL & Direct3D 9 graphics APIs; there also was Direct3D 11, OpenGL ES, PS3, Xbox360, Flash Stage3D etc. We were compiling all variants of a shader into all these backends at import time! Even if you never ever needed the result of that :(

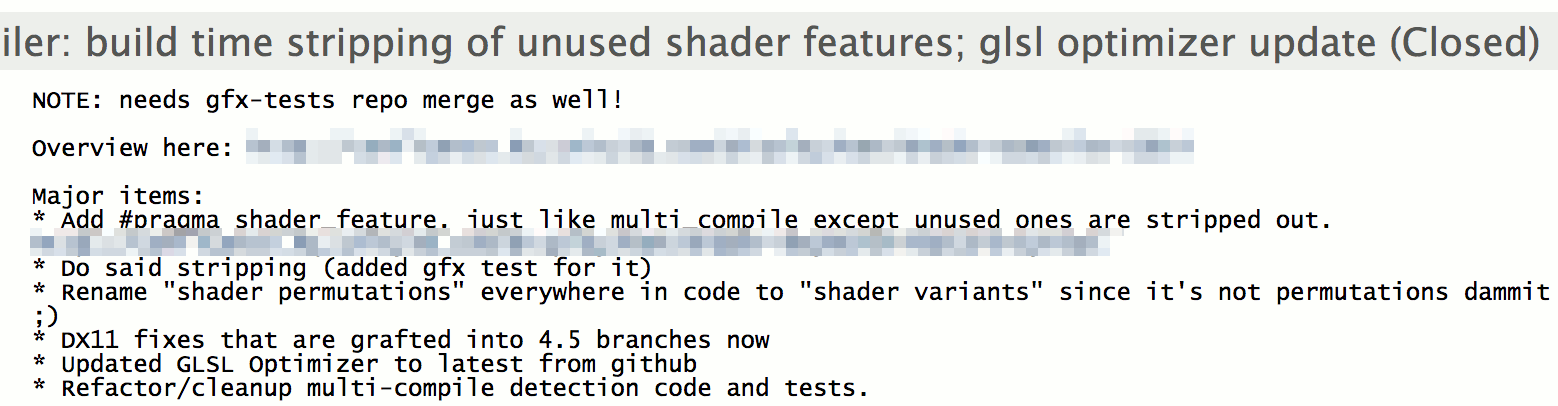
So [@robertcupisz](https://twitter.com/robertcupisz) and myself started rewriting the shader compilation pipeline. I have written about it before: [Shader compilation in Unity 4.5](https://aras-p.info/blog/2014/05/05/shader-compilation-in-unity-4-dot-5/). It basically changed several things:

* Shader variants would be compiled on-demand (whenever needed by editor itself; or for game data build) and cached,
* Shader compiler was split into a separate process per CPU core, to work around global mutexes in some shader compiler backends; these were preventing effictive multithreading.

This was a big improvement compared to previous state. *But are we done yet? Far from it.*

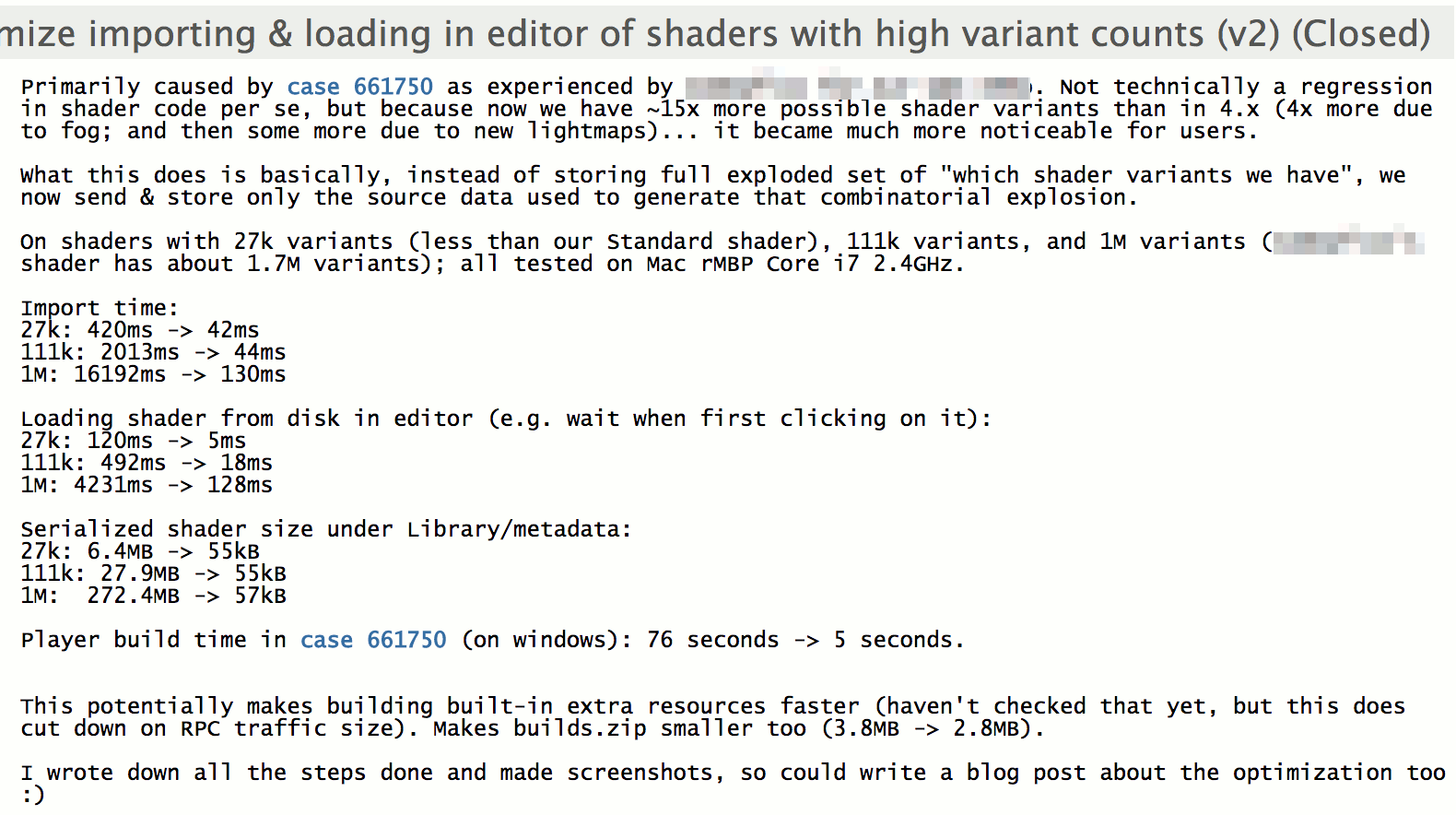
**Unity 5.0: need to make some variants optional**

The new compilation pipeline meant that editing a shader with a thousand potential variants was no longer a coffee break. However, all 1000 variants were still always included into the build. For Unity 5 launch we were developing a new built-in [Standard shader](https://docs.unity3d.com/Manual/shader-StandardShader.html) with 20000 or so possible variants, and always including all of them was a problem.

[](https://aras-p.info/img/blog/2017-02/scalability-50-stripping.png)So we added a way to indicate that “you know, only include these variants if some materials use them” when authoring a shader. Variants that were never used by anything were:

1. never even compiled and
2. not included into the game build either.

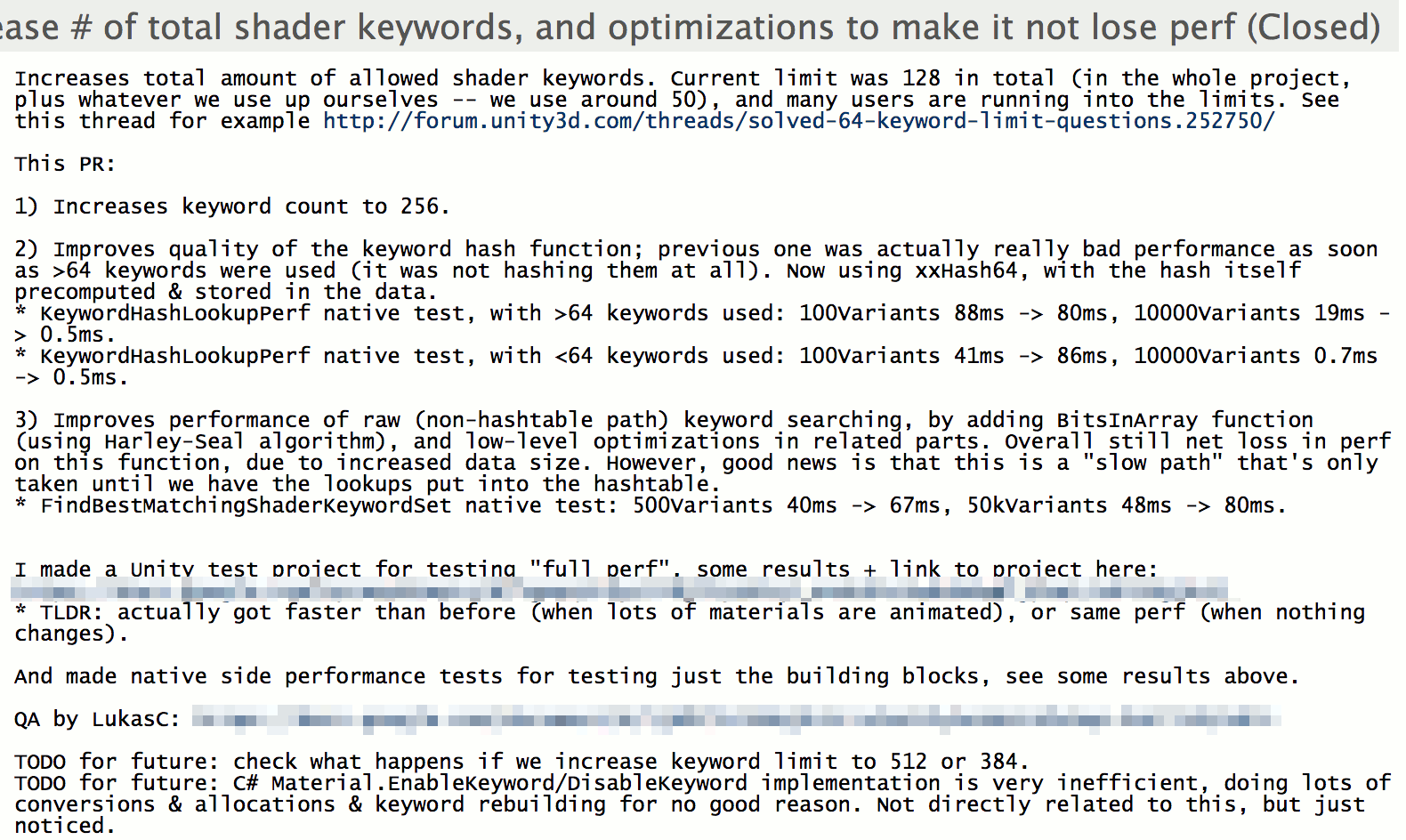
That was done in 2013 December, with plenty of time to ship in Unity 5.0.

[](https://aras-p.info/img/blog/2017-02/scalability-50-opt.png)During that time other people started “going crazy” with shader variant counts too – e.g. [Alloy shaders](https://alloy.rustltd.com/) were about 2 million variants. So we needed some more optimizations that I [wrote about before](https://aras-p.info/blog/2015/01/14/optimizing-shader-info-loading/), that managed to get just in time for Unity 5 launch.

So we went from “five variants” to “two million possible variants” by now… Is that the limit? *Are we there yet? Not quite!*

**Unity 5.4: people need more shader keywords**

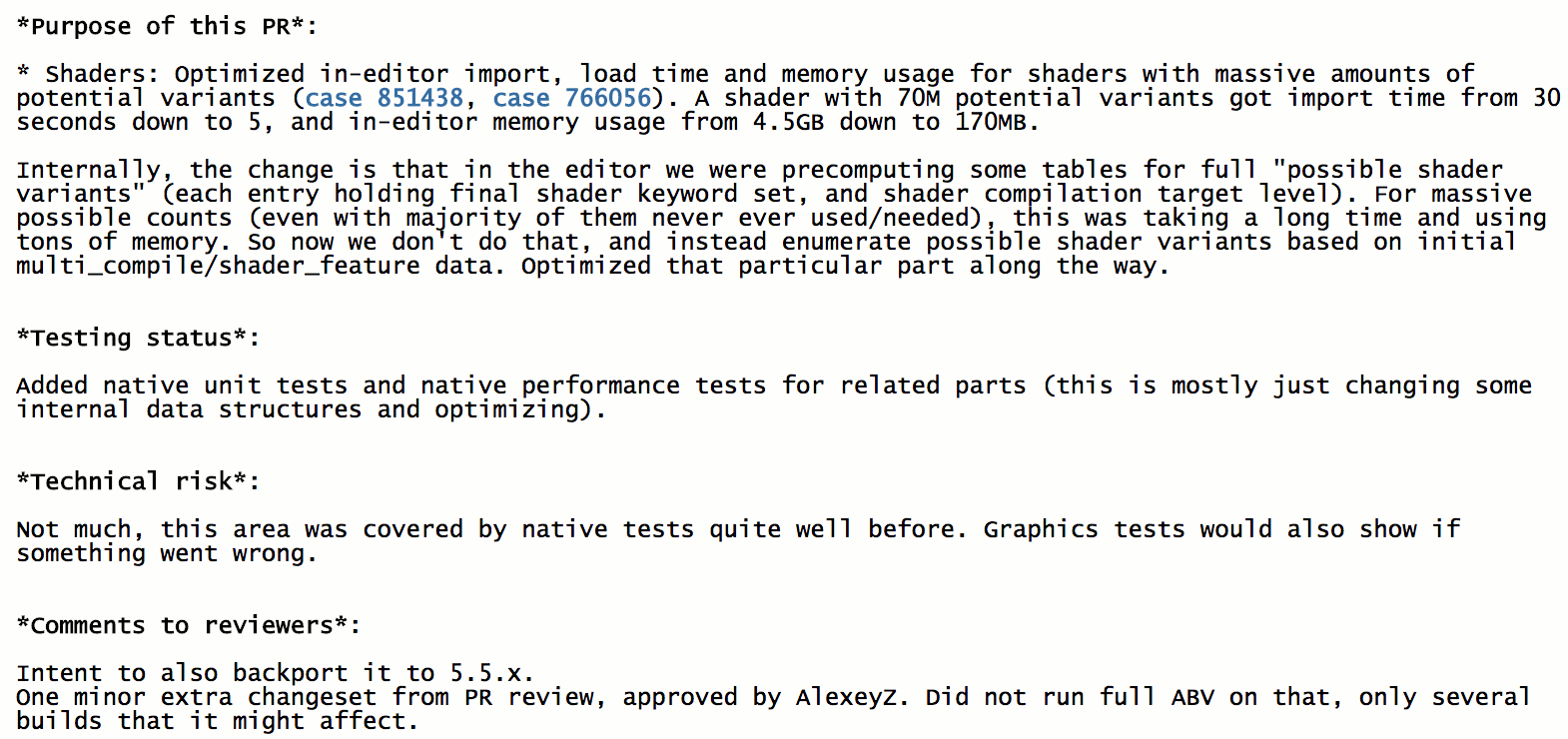
Sometime along the way amount of shader keywords (the “toggleable shader features that control which variant is picked”) that we support went from 32 up to 64, then up to 128. That was still not enough, as you can see from this [long forum thread](https://forum.unity3d.com/threads/solved-64-keyword-limit-questions.252750/).

[](https://aras-p.info/img/blog/2017-02/scalability-54-keywords.png)So I looked at increasing the keyword count to 256. Turns out, it was doable, especially after fiddling around with [some hash functions](https://aras-p.info/blog/2016/08/09/More-Hash-Function-Tests/). Side effect of investigating various hash functions: replaced almost all hash functions used across the whole codebase \o/.

Ok this by itself does neither improve scalability of shader variant parts, nor makes it worse… Except that with more shader keywords, people started adding *even more* potential shader variants! Give someone a thing, and they will start using it in both expected and unexpected ways.

*Are we there yet? Nope, still a few possible scalability cliffs in near future.*

**Unity 5.5: we are up to 70 million variants now, or “don’t expand that data”**

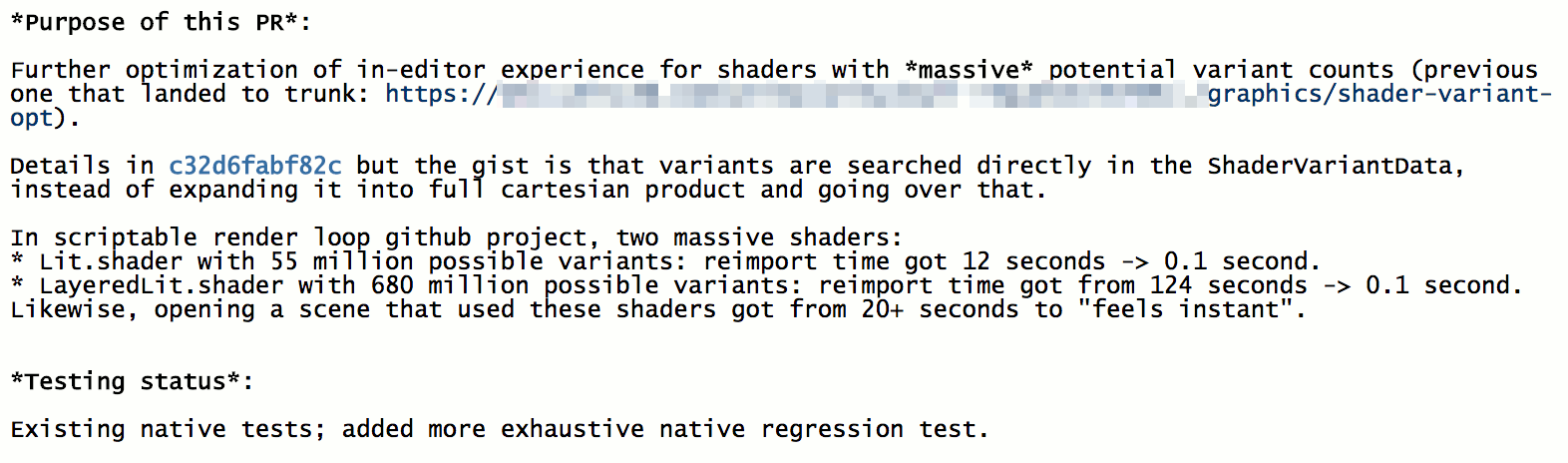
[](https://aras-p.info/img/blog/2017-02/scalability-55-import-opt.png)Our own team working on the new “[HD Render Pipeline](https://docs.google.com/document/d/1e2jkr_-v5iaZRuHdnMrSv978LuJKYZhsIYnrDkNAuvQ/edit#heading=h.gu5i27hc83ei)” had shaders with about 70 million of *possible* variants by now.

Turns out, there was a step where in the editor, we were still expanding some data for all possible variants into some in-memory structures. At 70 million potential variants, that was taking gobs of memory, and a lot of time was spent searching through that.

Time to fix that! Stop expanding that data, instead search directly from a fairly compact “unexpanded” data. That unblocked the team; import time from doing a minor shader edit went from “a minute” to “a couple seconds”.

Yay! *For a while*.

**Unity 5.6: half a billion variants anyone? or “don’t search that data”**

[](https://aras-p.info/img/blog/2017-02/scalability-56-seach-opt.png)Of course they went up to half a billion possible variants, and another problem surfaced: in some paths in the editor, when it was looking for “okay so which shader variant I should use right now?”, the code was enumerating all possible variants and checking which one is “closest to what we want”. In Unity, shader keywords do not have to exactly match some variant present in the shader, for better or worse… Previous step made it so that the table of “all possible variants” is not expanded in memory. But purely enumerating half a billion variants and doing fairly simple checks for each was still taking a long time! “Half a billion” turns out to be a big number.

Now of course, doing a search like that is fairly stupid. If we know we are searching for a shader variant with a keyword “NORMALMAP\_ON” in it, there’s very little use in enumerating all the ones that do *not* have it. Each keyword cuts the search space in half! So that optimization was done, and nicely got some timings from “dozens of seconds” to “feels instant”. For that case when you have half a billion shader variants, that is :)

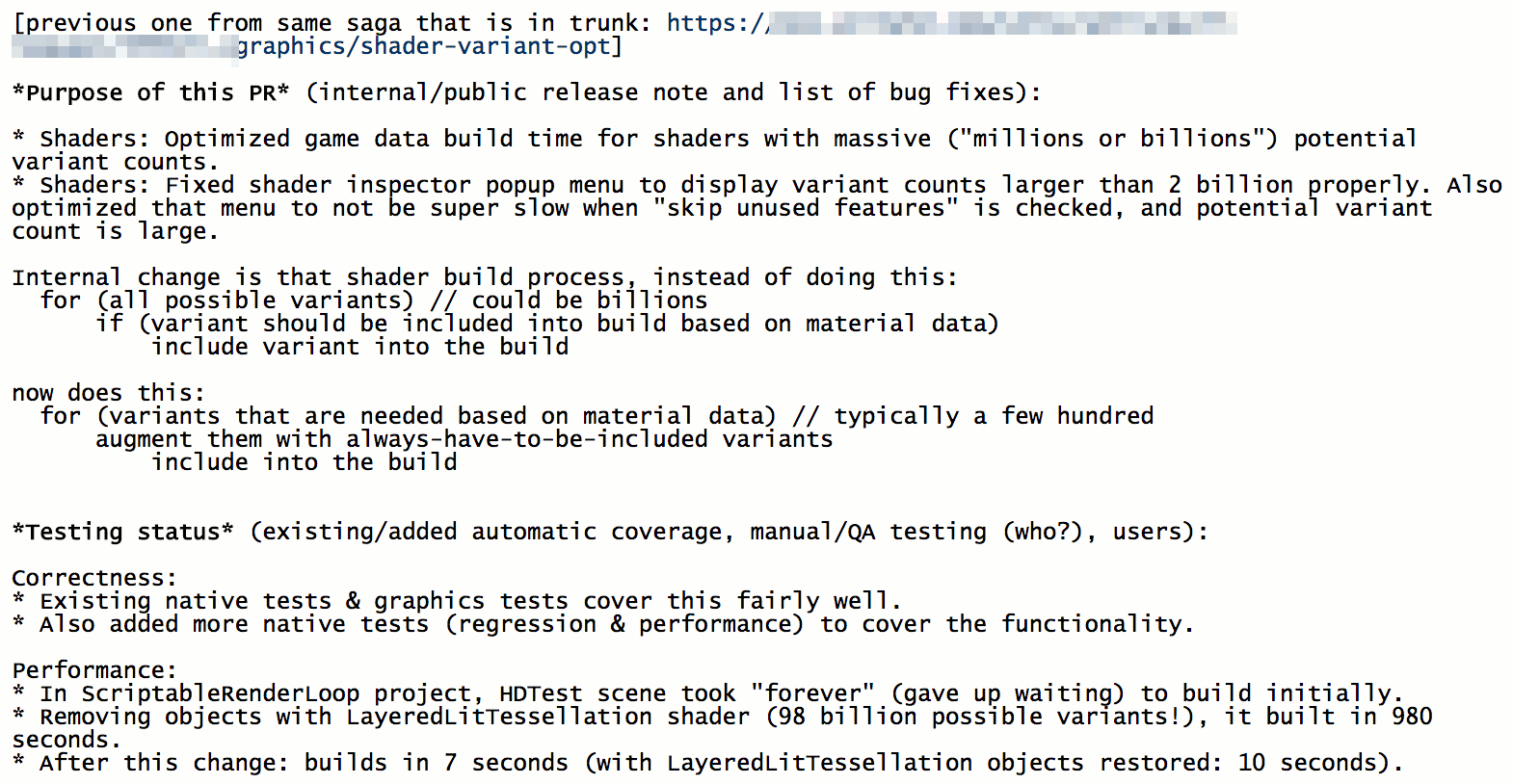
*We are done now, right?*

**Now: can I have hundred billion variants? or “dont' search that *other* data”**

Somehow the team ended up with a shader that has almost a hundred billion of possible variants. How? Don’t ask; my guess is by adding everything and the kitchen sink to it. From a quick look, it is “layered lit + tessellation” shader, and it seems to have:

* Usual optional textures: normal map, specular map, emissive map, detail map, detail mask map.
* One, two, three or four “layers” of the maps, mixed together.
* Mixing based on vertex colors, or height, or something else.
* Tessellation: displacement, Phong + displacement, parallax occlusion mapping.
* Several double sided lighting modes.
* Transparency and alpha cutout options.
* Lightmapping options.
* A few other oddball things.

The thing is, you “only” need about 36 of individually toggleable features to get to a hundred billion variant range (). 36 features is a lot of features, but imaginable.2^36=69B

[](https://aras-p.info/img/blog/2017-02/scalability-57-build-opt.png)The problem they ran into, was that at game data build time, the code was, similar to the previous case, looping over all possible shader variants and deciding whether each one should be included into the data file or not. Looping over a hundred billion simple things is a long process! So they were like “oh we wanted to do a build to check performance on a console, but gave up waiting”. Not good!

And of course it’s a stupid thing to do. The loop should be inverted, since we already have the info about which materials are included into the game build, and from there know which shader variants are needed. We just need to augment that set with variants that “always have to be in the build”, and that’s it. That got the build time from “forever” down to ten seconds.

*Are we done yet?* I don’t know. We’ll see what the future will bring :)

**Moral of the story is**: code that used to do something with five things years ago might turn out to be problematic when it has to deal with a hundred. And then a thousand. And a million. And a hundred billion. Kinda obvious, isn’t it?

* [← Older](https://aras-p.info/blog/2017/01/30/UI-is-hard-and-other-Typical-Work-Stories/)
* [Newer →](https://aras-p.info/blog/2017/02/15/Font-Rendering-is-Getting-Interesting/)

**Possibly Related Posts**

* [Optimizing Unity Renderer Part 3: Fixed Function Removal](https://aras-p.info/blog/2015/04/27/optimizing-unity-renderer-3-fixed-function-removal/), from 2015 April
* [Optimizing Unity Renderer Part 2: Cleanups](https://aras-p.info/blog/2015/04/04/optimizing-unity-renderer-2-cleanups/), from 2015 April
* [Optimizing Unity Renderer Part 1: Intro](https://aras-p.info/blog/2015/04/01/optimizing-unity-renderer-1-intro/), from 2015 April