SIGGRAPH2015

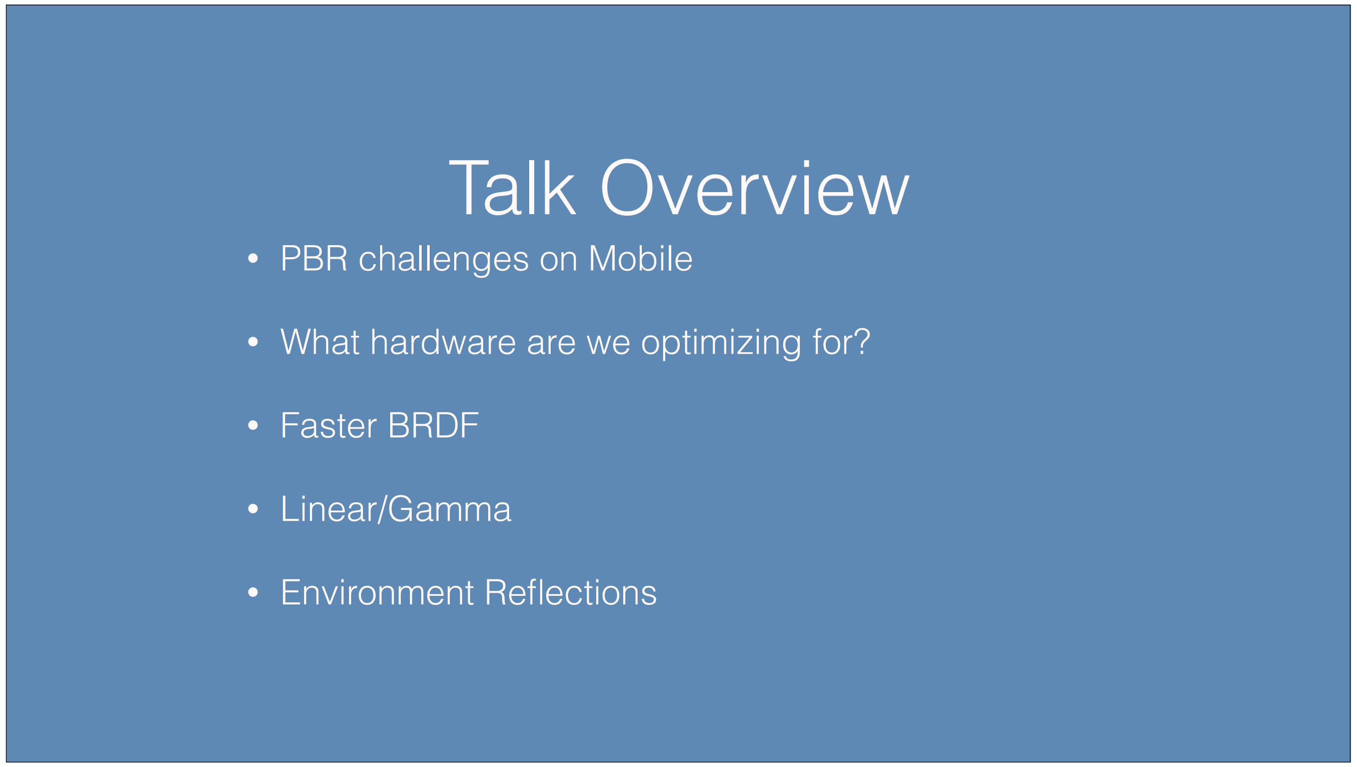
Xroads of Discovery

SIGGRAPH 2015
Xroads of Discovery

The 42nd International Conference and Exhibition on Computer Graphics and Interactive Techniques

Optimizing PBR

Renaldas Zioma
Unity Technologies



Talk Overview

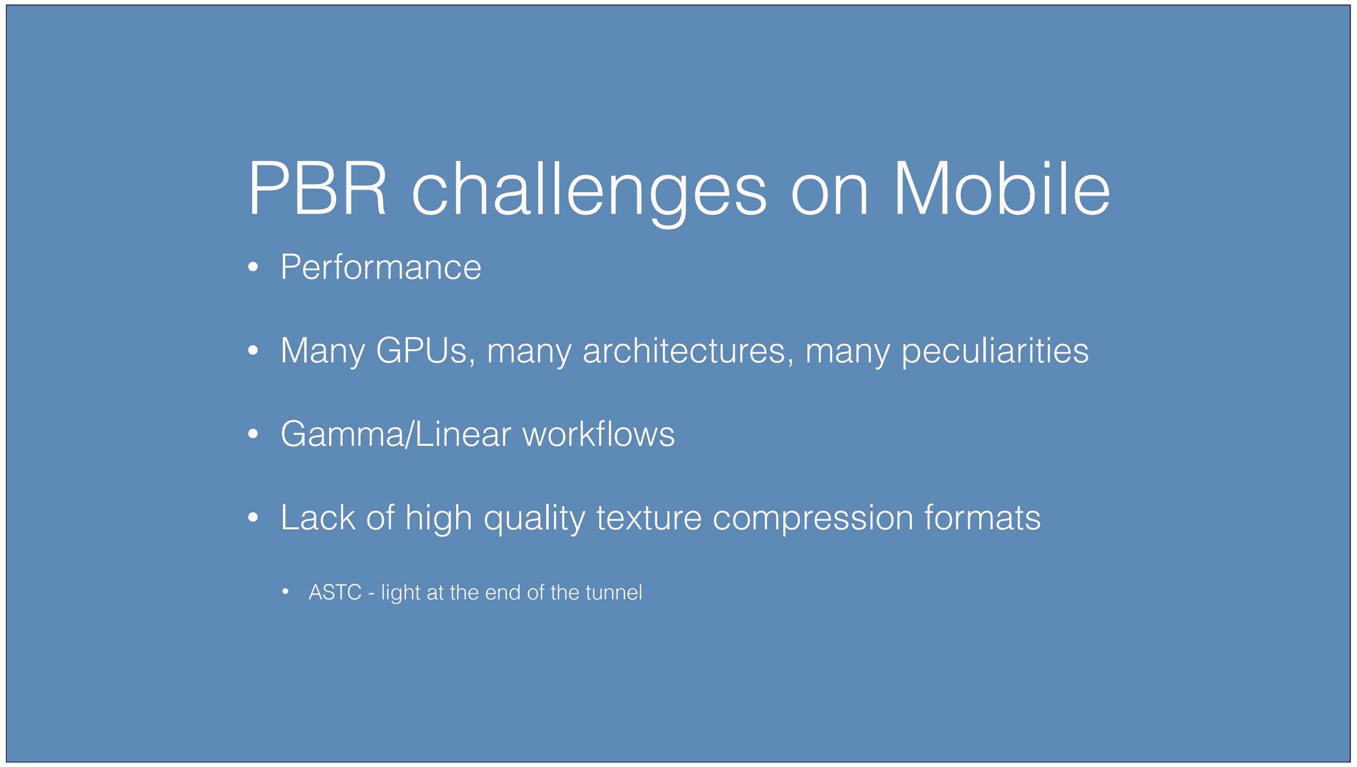
- PBR challenges on Mobile
- What hardware are we optimizing for?
- Faster BRDF
- Linear/Gamma
- Environment Reflections

What is this talk about?

The main focus of this talk is about bringing content developed with Physically Based Shading and Rendering in mind to Mobiles and optimizations to make it happen on a broad range of devices and GPUs.

I'll start with challenges and discuss the broad optimization target consisting of multitude of different GPU architectures.

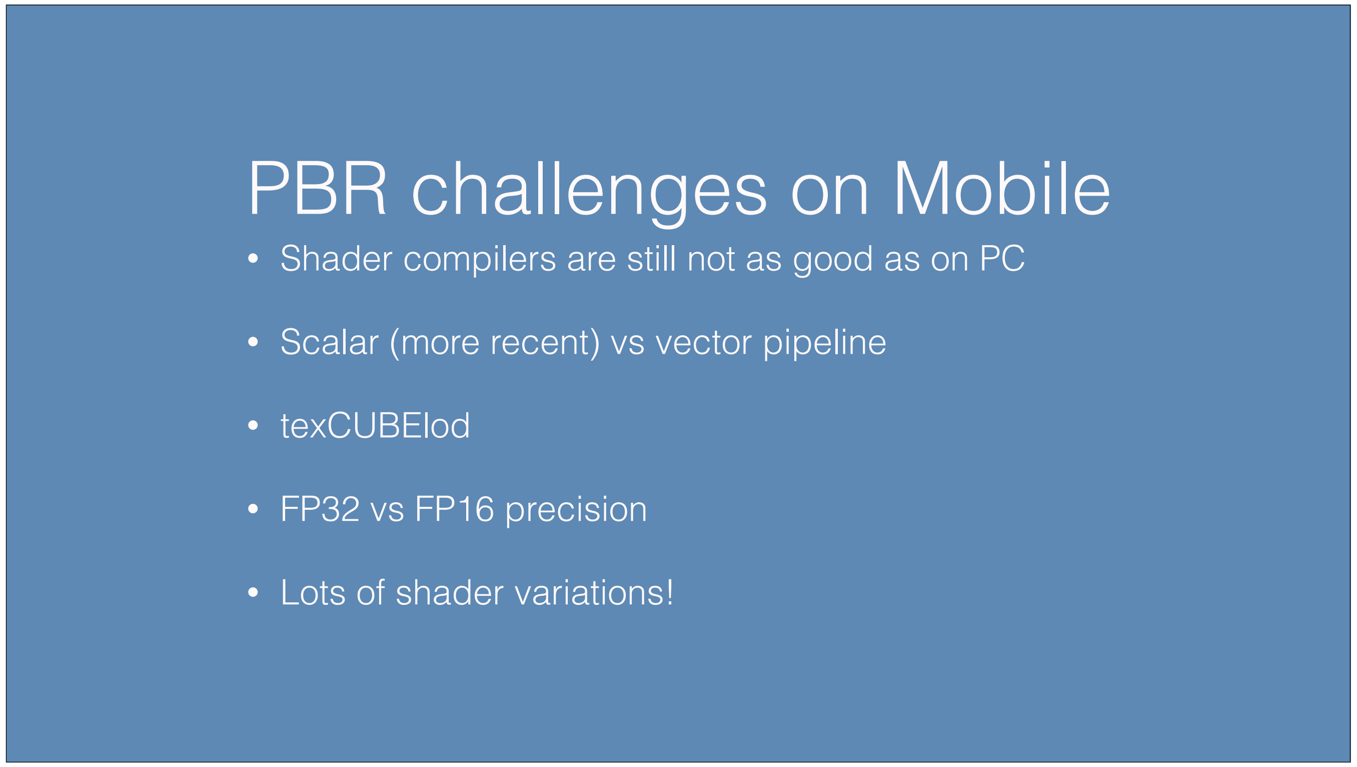
Then I'll discuss BRDF approximation that is suitable for Mobiles. Next I'll cover the ways to make Linear and Gamma workflows agree. For the closing I'll briefly cover some tricks for glossy reflections.



PBR challenges on Mobile

- Performance
- Many GPUs, many architectures, many peculiarities
- Gamma/Linear workflows
- Lack of high quality texture compression formats
  - ASTC - light at the end of the tunnel

Of course high-screen resolutions and tight power constraints makes complex shading a challenge on Mobile. To make it a harder challenge there is a multitude of different mobile GPU architectures and generations in the hands of the gamers. We can’t possibly optimize for every device, so we have to split our target into manageable chunks by analyzing similarities, differences and roughly grouping GPUs/devices by performance.
Linear lighting is almost an impossible tasks for many devices out there. However traditionally it is almost a requirement for PBR. Can we help our content creators to avoid redoing the content and have at least consistent look with non-Linear (aka Gamma) rendering mode?
Next challenge is lack of compressed uncorrelated 4-channel and HDR texture formats. Storing Roughness information and high quality glossy reflections require additional consideration.

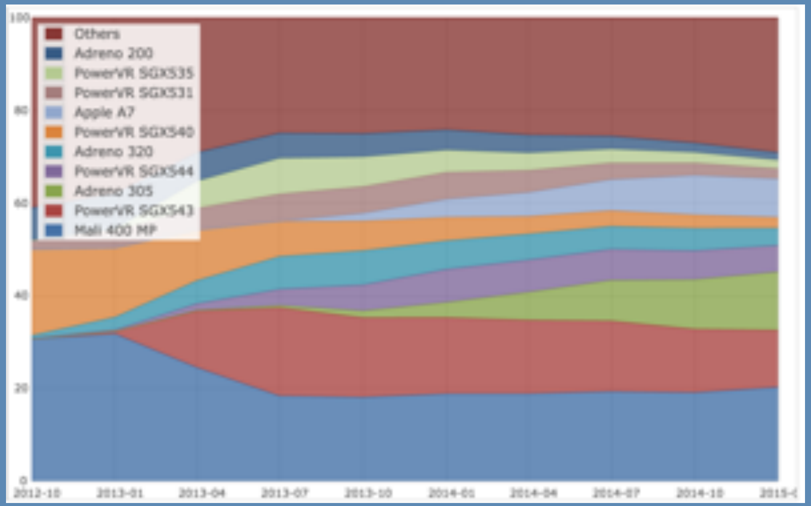


PBR challenges on Mobile

- Shader compilers are still not as good as on PC
- Scalar (more recent) vs vector pipeline
- texCUBElod
- FP32 vs FP16 precision
- Lots of shader variations!

On top of that:
Scalar and vector architectures of shading cores might require different optimizations resulting in more shader variations to maintain and distribute. What seems to be a useful approach at least in terms of optimizations is to make your shader good for scalar pipeline first - that usually makes work for shader compiler for vector architectures easier. If necessary start manual vectorization next.
On certain GPU architectures reading from particular mip-level (not literal constant) is a pretty expensive operation. On others it is OK.
There are GPUs that can perform ALU operations only with reduced precision. In the end you're bound to have a multitude of shader variations in order to take all these aspects into account. It becomes crucial to split problems into a manageable groups.

Optimization Target



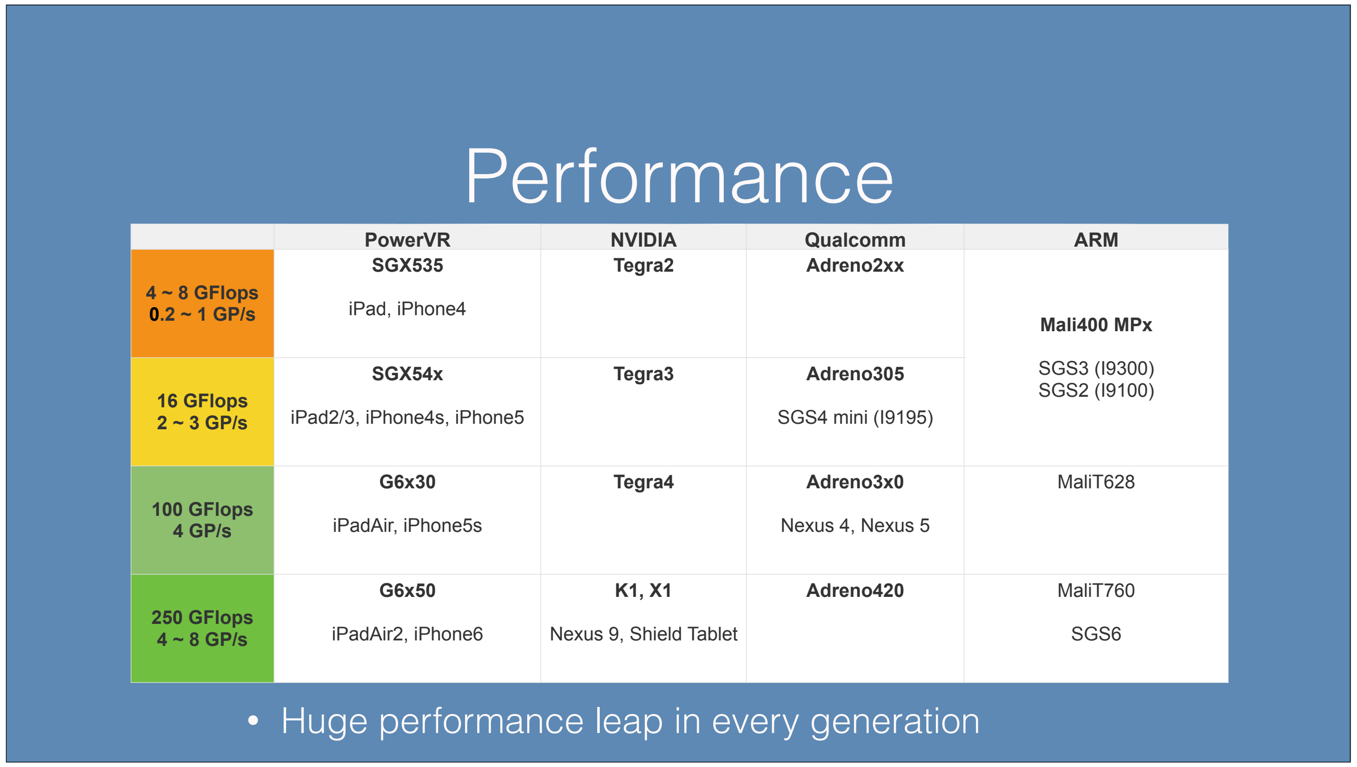
based on # of apps running Unity

What are out optimization targets?

Here you can see the market share by popular GPUs combined for both iOS, Android and WindowsPhone. Data is based on apps running Unity over last 3 years.

As you can clearly see there is no single winner you can concentrate on and although there are certain traits they're not straightforward.

For instance while more devices are produced with faster high-end GPUs, there is a strong influx of cheap last generation GPUs. Some not even supporting OpenGL ES3.0!



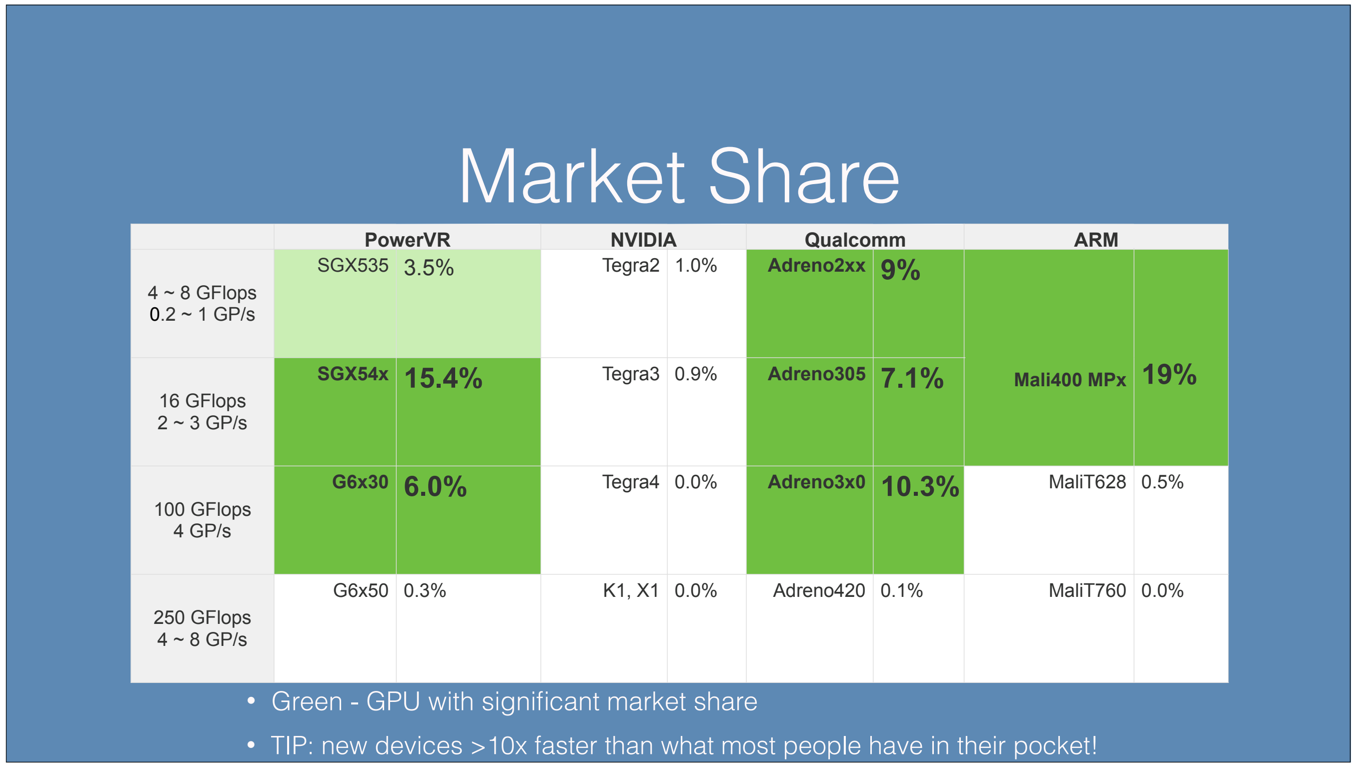
Performance

	PowerVR	NVIDIA	Qualcomm	ARM
4 ~ 8 GFlops 0.2 ~ 1 GP/s	**SGX535** iPad, iPhone4	**Tegra2**	**Adreno2xx**	**Mali400 MPx** SGS3 (I9300) SGS2 (I9100)
16 GFlops 2 ~ 3 GP/s	**SGX54x** iPad2/3, iPhone4s, iPhone5	**Tegra3**	**Adreno305** SGS4 mini (I9195)	
100 GFlops 4 GP/s	**G6x30** iPadAir, iPhone5s	**Tegra4**	**Adreno3x0** Nexus 4, Nexus 5	MaliT628
250 GFlops 4 ~ 8 GP/s	**G6x50** iPadAir2, iPhone6	**K1, X1** Nexus 9, Shield Tablet	**Adreno420**	MaliT760 SGS6

- Huge performance leap in every generation

This table groups most popular GPUs into performance groups and architectural families.

As you can see every generation comes with a significant performance leap! Modern mobile GPUs rivaling specs of the last generation consoles.



Market Share

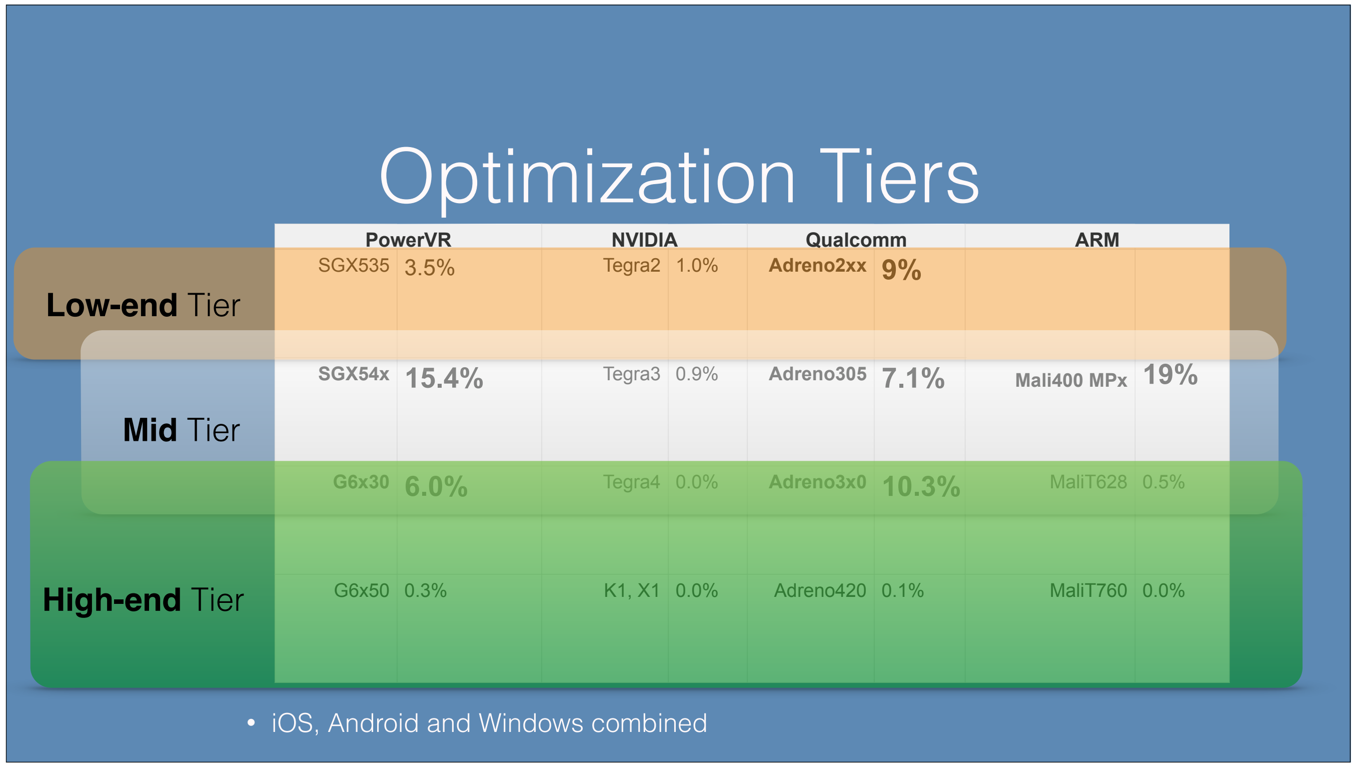
	PowerVR		NVIDIA		Qualcomm		ARM	
4 ~ 8 GFlops 0.2 ~ 1 GP/s	SGX535	3.5%	Tegra2	1.0%	**Adreno2xx**	**9%**	**Mali400 MPx**	**19%**
16 GFlops 2 ~ 3 GP/s	**SGX54x**	**15.4%**	Tegra3	0.9%	**Adreno305**	**7.1%**		
100 GFlops 4 GP/s	**G6x30**	**6.0%**	Tegra4	0.0%	**Adreno3x0**	**10.3%**	MaliT628	0.5%
250 GFlops 4 ~ 8 GP/s	G6x50	0.3%	K1, X1	0.0%	Adreno420	0.1%	MaliT760	0.0%

- Green - GPU with significant market share
- TIP: new devices >10x faster than what most people have in their pocket!

Let's add market share into the picture. As you see that gravity of the market pulls towards lower and medium ends. Not something graphics programmer usually wants to see :)

As well you can see that addressing all architectures is important if you want to support your user base well - no clear winner.

It is worth noticing that high-end device will be most likely 10 times faster than the average one!

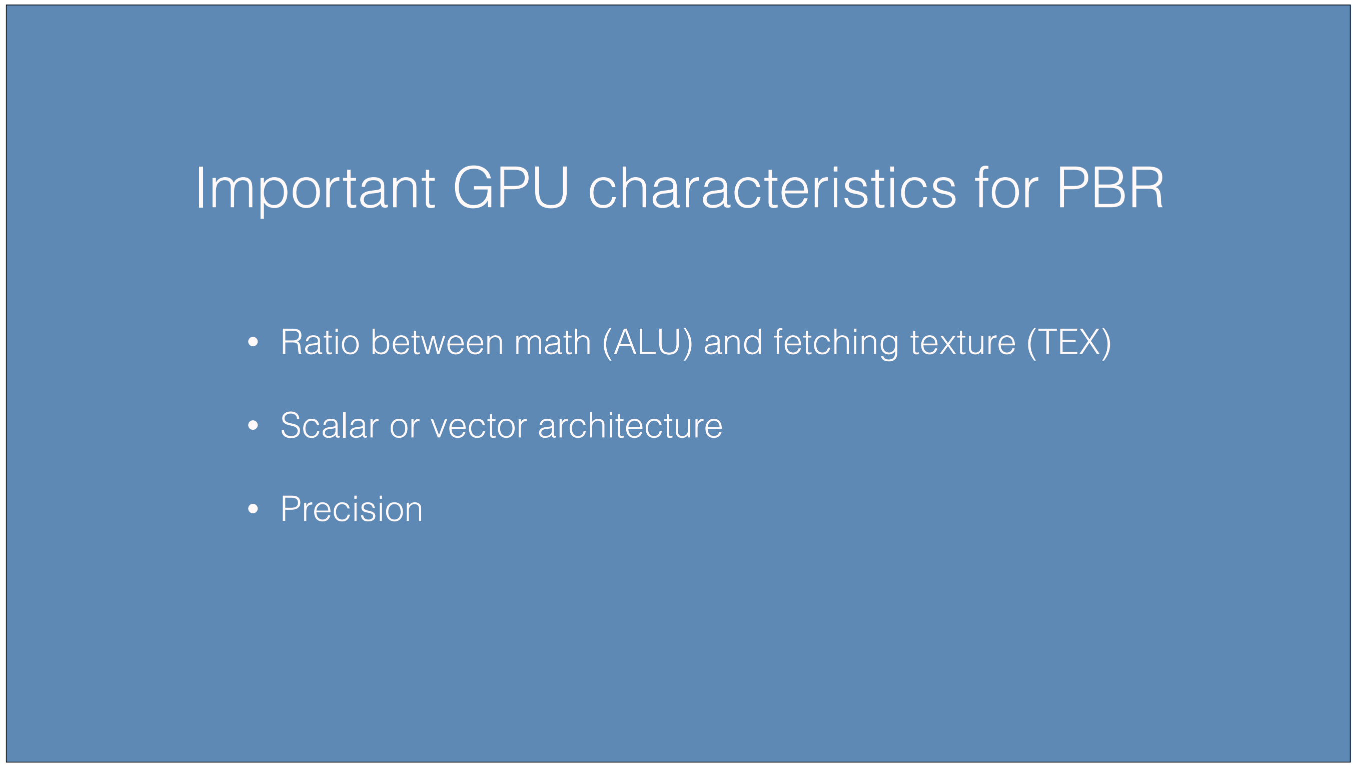


Optimization Tiers

	PowerVR		NVIDIA		Qualcomm		ARM	
Low-end Tier	SGX535	3.5%	Tegra2	1.0%	Adreno2xx	9%		
Mid Tier	SGX54x	15.4%	Tegra3	0.9%	Adreno305	7.1%	Mali400 MPx	19%
	G6x30	6.0%	Tegra4	0.0%	Adreno3x0	10.3%	MaliT628	0.5%
High-end Tier	G6x50	0.3%	K1, X1	0.0%	Adreno420	0.1%	MaliT760	0.0%

- iOS, Android and Windows combined

Let’s split devices into performance tiers and see what optimization we can do for each tier.

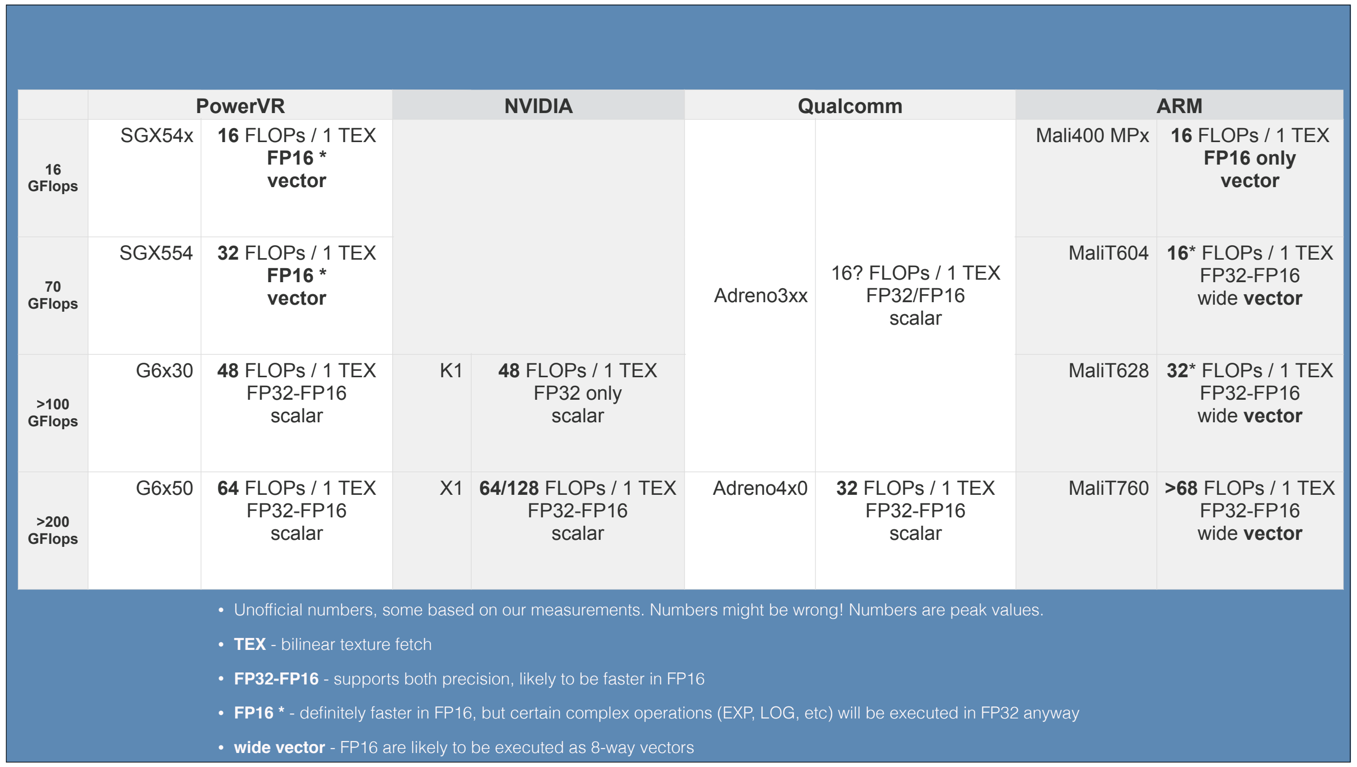

Important GPU characteristics for PBR

- Ratio between math (ALU) and fetching texture (TEX)
- Scalar or vector architecture
- Precision

For each tier it is important to understand how much math operation can we do for each single texture being read.

It is useful to know what type of shading core architecture dominates. Is it mostly scalar or vector?

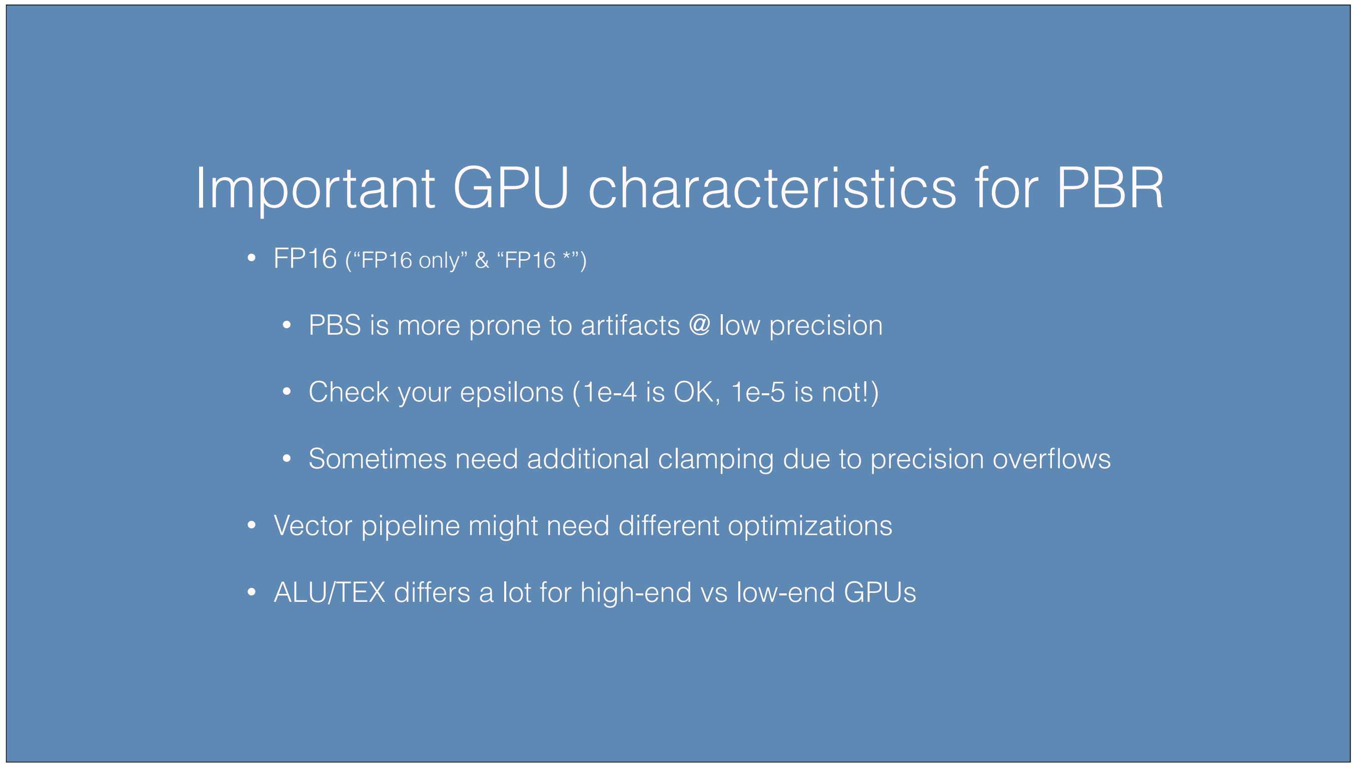
Next, how well it operates at full precision? Can we gain performance by using reduced precision? Is it worth in terms of quality?



	PowerVR		NVIDIA		Qualcomm		ARM	
16 GFlops	SGX54x	**16** FLOPs / 1 TEX **FP16 *** **vector**					Mali400 MPx	**16** FLOPs / 1 TEX **FP16 only** **vector**
70 GFlops	SGX554	**32** FLOPs / 1 TEX **FP16 *** **vector**			Adreno3xx	16? FLOPs / 1 TEX FP32/FP16 scalar	MaliT604	**16*** FLOPs / 1 TEX FP32-FP16 wide **vector**
>100 GFlops	G6x30	**48** FLOPs / 1 TEX FP32-FP16 scalar	K1	**48** FLOPs / 1 TEX FP32 only scalar			MaliT628	**32*** FLOPs / 1 TEX FP32-FP16 wide **vector**
>200 GFlops	G6x50	**64** FLOPs / 1 TEX FP32-FP16 scalar	X1	**64/128** FLOPs / 1 TEX FP32-FP16 scalar	Adreno4x0	**32** FLOPs / 1 TEX FP32-FP16 scalar	MaliT760	**>68** FLOPs / 1 TEX FP32-FP16 wide **vector**

- Unofficial numbers, some based on our measurements. Numbers might be wrong! Numbers are peak values.
- **TEX** - bilinear texture fetch
- **FP32-FP16** - supports both precision, likely to be faster in FP16
- **FP16 *** - definitely faster in FP16, but certain complex operations (EXP, LOG, etc) will be executed in FP32 anyway
- **wide vector** - FP16 are likely to be executed as 8-way vectors

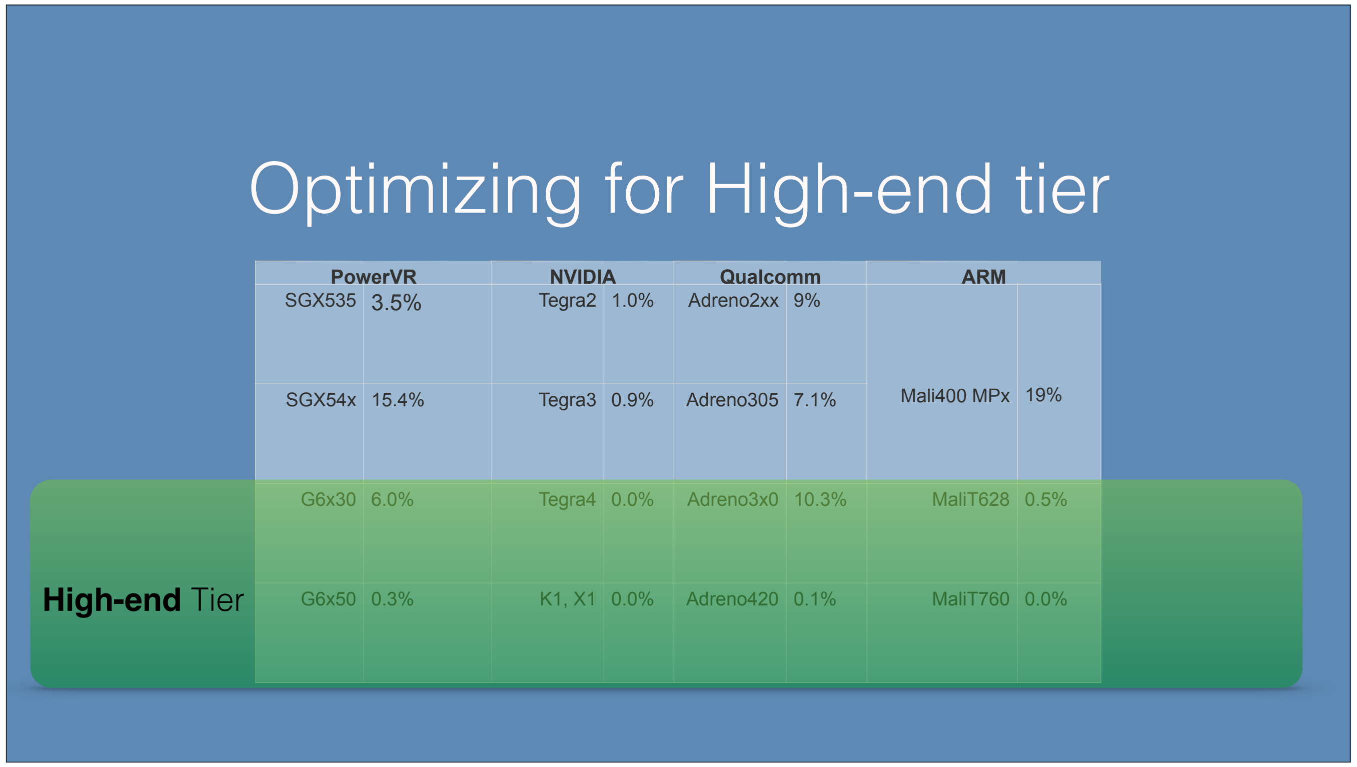
First number show how many simple ALU operations can GPU execute in parallel with reading single bilinear interpolated texture. Note that most often trilinear interpolation and cubemaps will double or quadruple cost of the fetch. As well this numbers only apply when reading textures doesn’t exceed memory bandwidth. Many modern architectures/SoCs are made in mind that you’re barely limited by memory bandwidth when reading uncompressed 32bpp textures, but unfortunately not all of them. Some optimize for compressed texture reads.
Next you will notice that there are some GPU that require you to worry about FP16 precision. Most modern GPUs are fine executing at full (FP32) precision, but most often you get better pipelining, if you use FP16 whenever you can. That does not guarantee 100% performance gain, more like 30-50%.
Mobile GPUs used to have predominantly 4 component vector shading cores. That’s not the case anymore, most are fine with scalar. So avoid multiplying two 4 component vectors, if you going to use only 3 components in the end.



Important GPU characteristics for PBR

- FP16 (“FP16 only” & “FP16 *”)
  - PBS is more prone to artifacts @ low precision
  - Check your epsilons (1e-4 is OK, 1e-5 is not!)
  - Sometimes need additional clamping due to precision overflows
- Vector pipeline might need different optimizations
- ALU/TEX differs a lot for high-end vs low-end GPUs

Watch out for epsilons and constants. They can easily overflow or not have enough precision on FP16 only GPUs.
Keep in mind ALU-to-TEX ratio changes drastically between generations. Predictably we have much more ALU power nowadays with ratio matching desktop GPU nowadays.

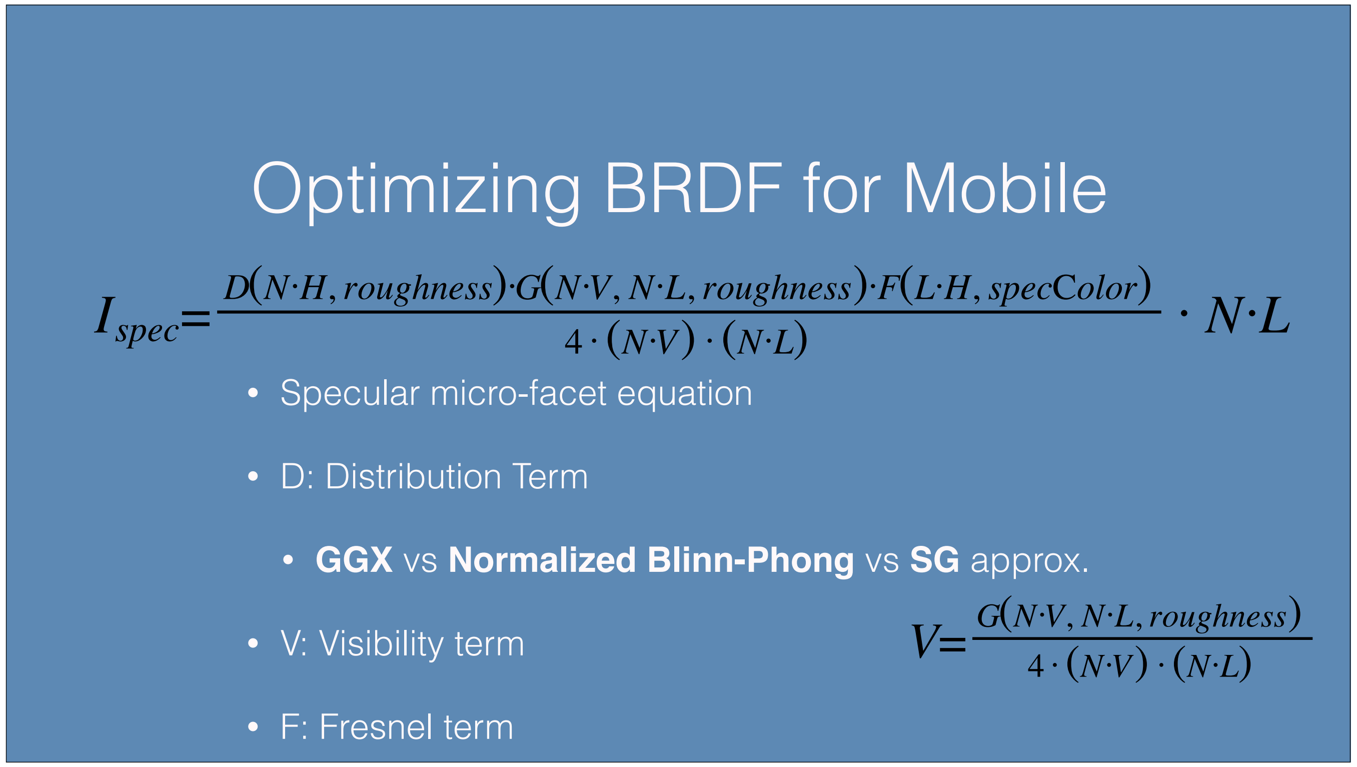


Optimizing for High-end tier

PowerVR		NVIDIA		Qualcomm		ARM	
SGX535	3.5%	Tegra2	1.0%	Adreno2xx	9%		
SGX54x	15.4%	Tegra3	0.9%	Adreno305	7.1%	Mali400 MPx	19%
G6x30	6.0%	Tegra4	0.0%	Adreno3x0	10.3%	MaliT628	0.5%
G6x50	0.3%	K1, X1	0.0%	Adreno420	0.1%	MaliT760	0.0%

High-end Tier

Given high-performance Tier with mostly scalar shading cores lets see how far can we go optimizing the most performance hungry part of the shader - specular BRDF.



Optimizing BRDF for Mobile

$$I_{spec} = \frac{D(N \cdot H, roughness) \cdot G(N \cdot V, N \cdot L, roughness) \cdot F(L \cdot H, specColor)}{4 \cdot (N \cdot V) \cdot (N \cdot L)} \cdot N \cdot L$$

- Specular micro-facet equation
- D: Distribution Term
  - **GGX** vs **Normalized Blinn-Phong** vs **SG** approx.
- V: Visibility term
- F: Fresnel term

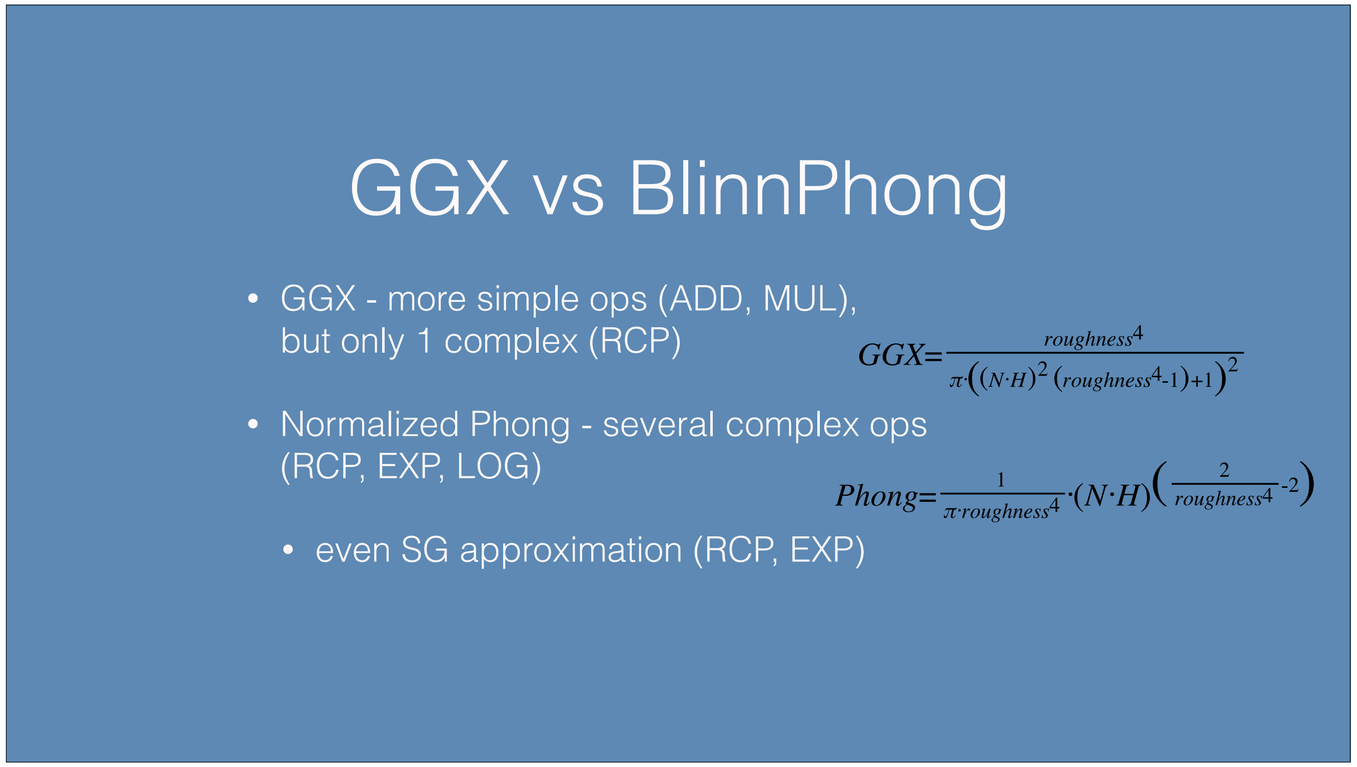
$$V = \frac{G(N \cdot V, N \cdot L, roughness)}{4 \cdot (N \cdot V) \cdot (N \cdot L)}$$

Classical form of micro-facet specular BRDF.

D - defines general shape of the highlight

V - shapes highlight in the vicinity of light terminator. Primary source for off-peak specular in case of metals - they have virtually no Fresnel curve.

F - defines color of highlight (classically defined by index of refraction parameter, simplified with specular color in games), creates off-peak and intensity falloff of highlight in case of diellectrics.



GGX vs BlinnPhong

- GGX - more simple ops (ADD, MUL), but only 1 complex (RCP)

$$GGX=\frac{roughness^4}{\pi\cdot\left((N\cdot H)^2\left(roughness^4-1\right)+1\right)^2}$$

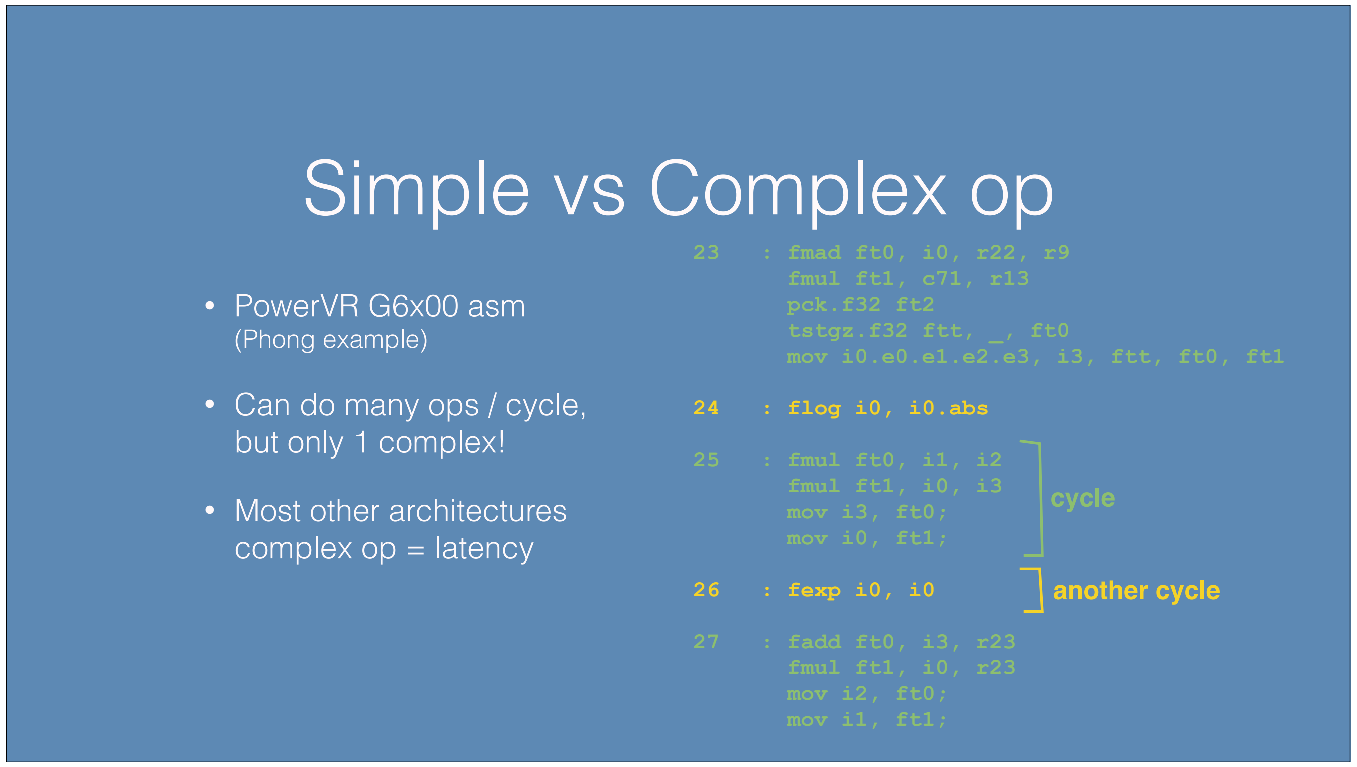
- Normalized Phong - several complex ops (RCP, EXP, LOG)

$$Phong=\frac{1}{\pi\cdot roughness^4}\cdot(N\cdot H)\left(\frac{2}{roughness^4}-2\right)$$

  - even SG approximation (RCP, EXP)

Common expectation is that Phong should be faster to shade. However in practice results between normalize Phong and GGX may vary depending on the architecture of GPU shading cores.

It should be noted that GGX in terms of complex operations relies only on division to form highlight. This is an important trait of GGX that we can leverage since cost of such operation can be shared with other terms of BRDF that require division too.



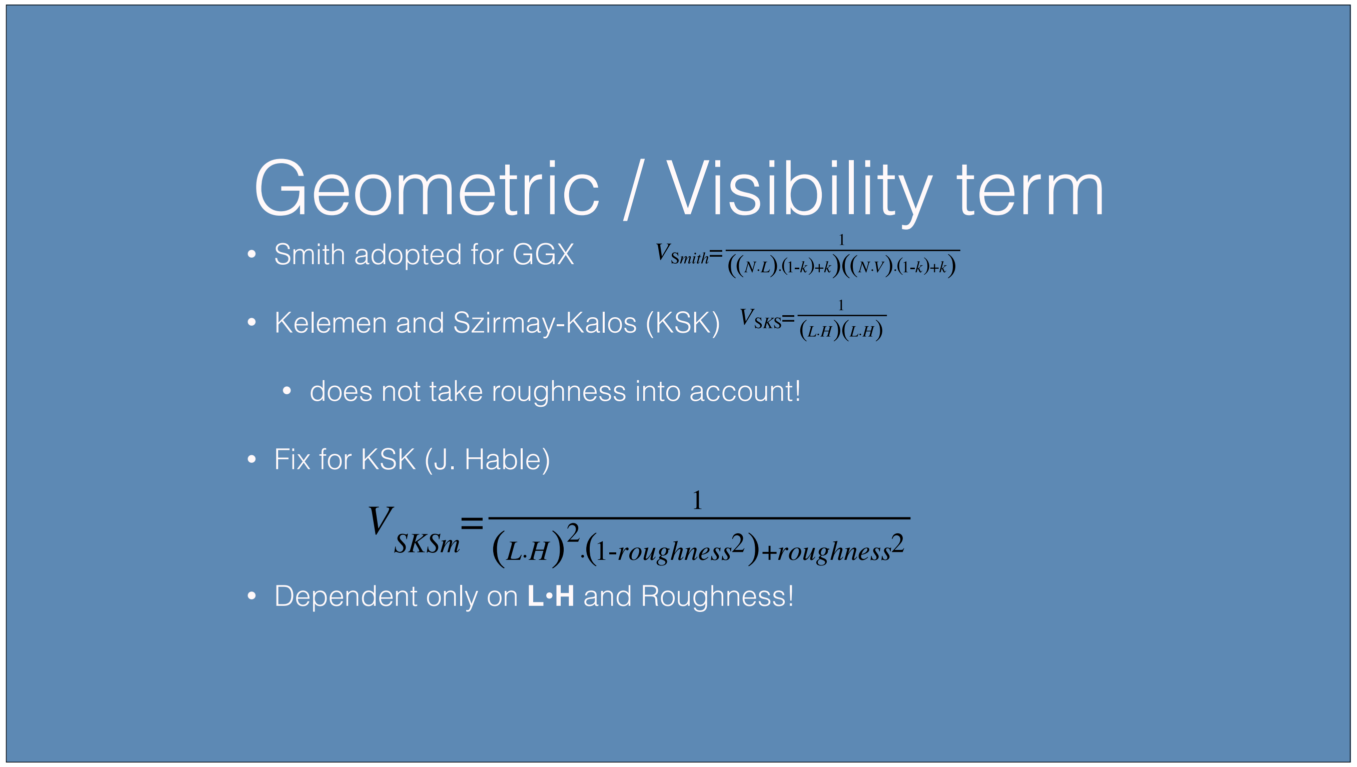
Simple vs Complex op

- PowerVR G6x00 asm
  (Phong example)
- Can do many ops / cycle, but only 1 complex!
- Most other architectures complex op = latency

```
23   : fmad ft0, i0, r22, r9
       fmul ft1, c71, r13
       pck.f32 ft2
       tstgz.f32 ftt, _, ft0
       mov i0.e0.e1.e2.e3, i3, ftt, ft0, ft1

24   : flog i0, i0.abs

25   : fmul ft0, i1, i2
       fmul ft1, i0, i3
       mov i3, ft0;
       mov i0, ft1;

26   : fexp i0, i0

27   : fadd ft0, i3, r23
       fmul ft1, i0, r23
       mov i2, ft0;
       mov i1, ft1;
```

cycle
another cycle

Why bother about complex ops?



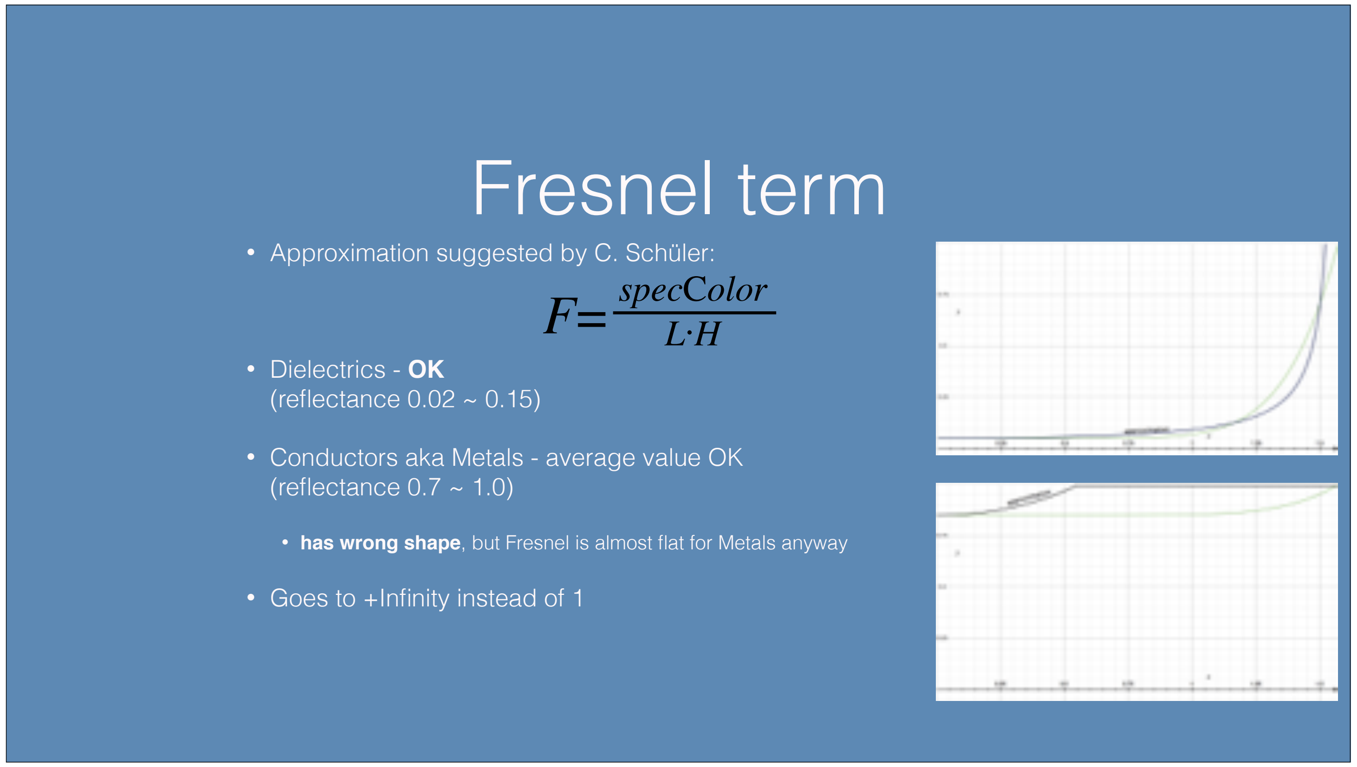
Geometric / Visibility term

- Smith adopted for GGX $V_{Smith}=\frac{1}{((N\cdot L)\cdot(1-k)+k)((N\cdot V)\cdot(1-k)+k)}$
- Kelemen and Szirmay-Kalos (KSK) $V_{SKS}=\frac{1}{(L\cdot H)(L\cdot H)}$
  - does not take roughness into account!
- Fix for KSK (J. Hable)

$$V_{SKSm}=\frac{1}{(L\cdot H)^2\cdot(1-roughness^2)+roughness^2}$$

- Dependent only on **L•H** and Roughness!

Smith is becoming de-facto standard of Visibility term. Another common and very cheap approximation is KSK. However main downside of KSK - it does not take roughness into account.
However by comparing Smith and KSK formula we can see that both follow the same “pattern” and KSK optimization lies in averaging N•L and N•V contributions with L•H. It is intuitive that KSK can be extended with roughness parameter.

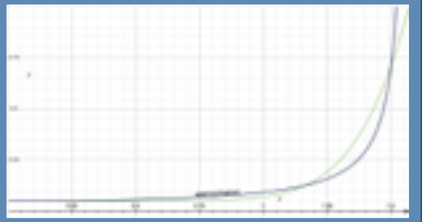


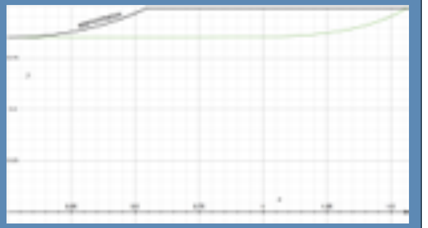
Fresnel term

- Approximation suggested by C. Schüler:

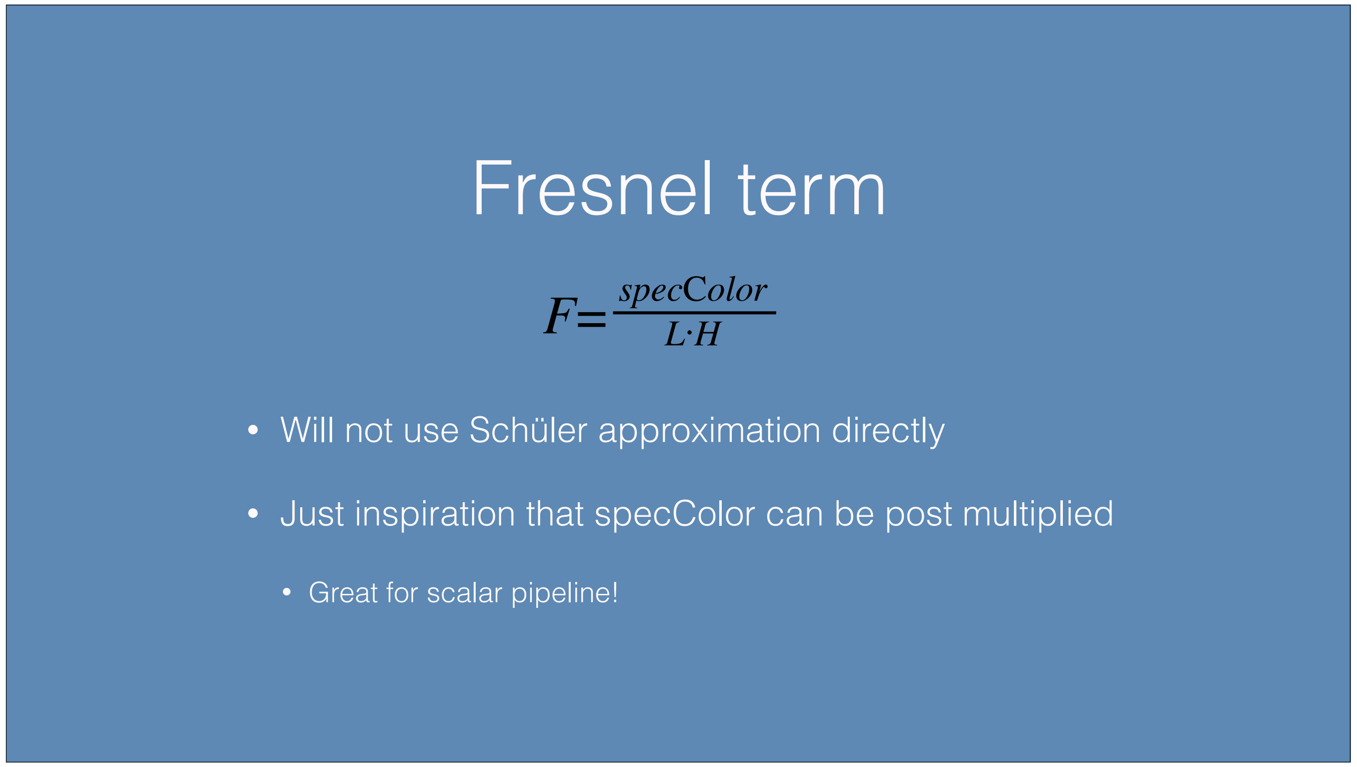
$$F = \frac{specColor}{L \cdot H}$$

- Dielectrics - **OK**
  (reflectance 0.02 ~ 0.15)
- Conductors aka Metals - average value OK
  (reflectance 0.7 ~ 1.0)
  - **has wrong shape**, but Fresnel is almost flat for Metals anyway
- Goes to +Infinity instead of 1





There is an interesting Fresnel approximation that was suggested by Christian Schüler. This approximation is a bit horrible in terms of following actual Fresnel curve for metals, but it gives an interesting insight.

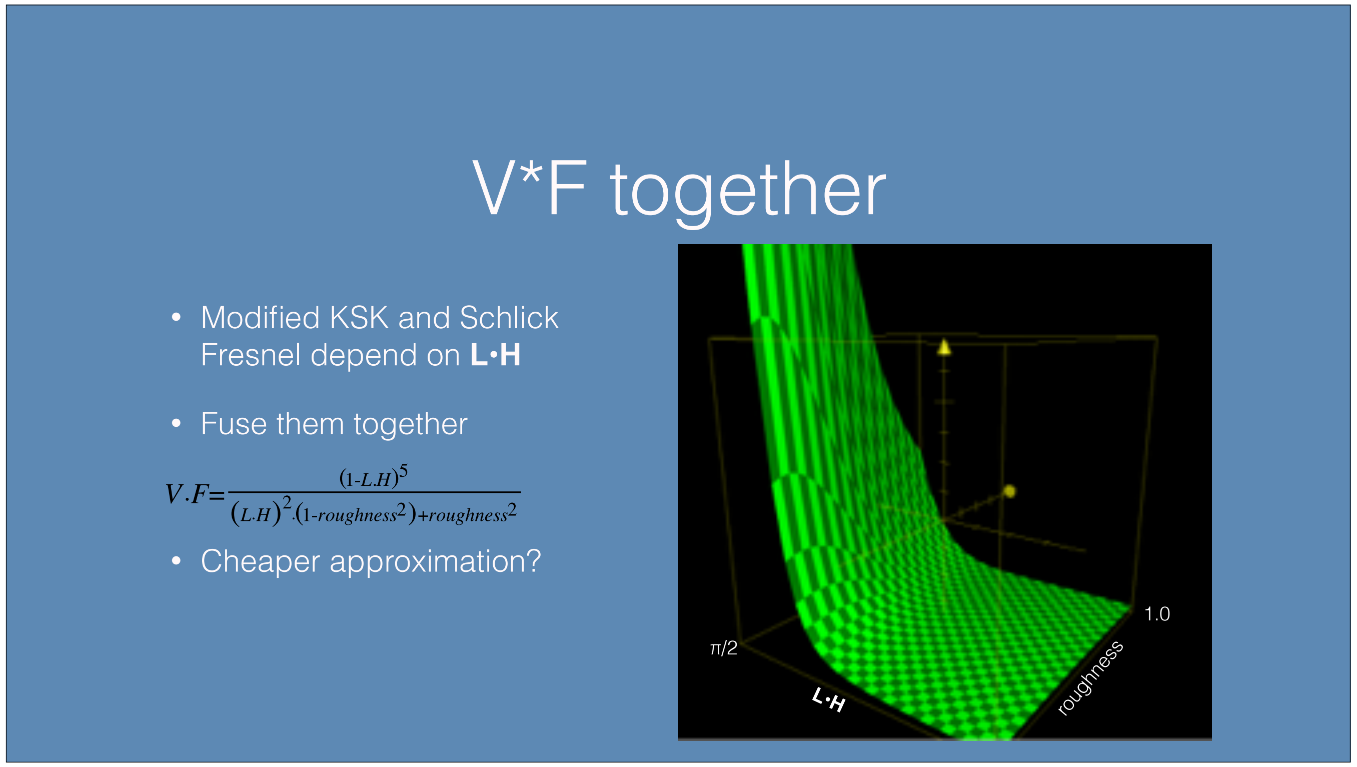


Fresnel term

$$F = \frac{specColor}{L \cdot H}$$

- Will not use Schüler approximation directly
- Just inspiration that specColor can be post multiplied
  - Great for scalar pipeline!

That we could get away evaluating whole specular BRDF disregarding index of refraction (specular color), but apply it at the end. This is brilliant for scalar architecture. Following steps will not use Schüler approximation directly, but just leverage this insight.

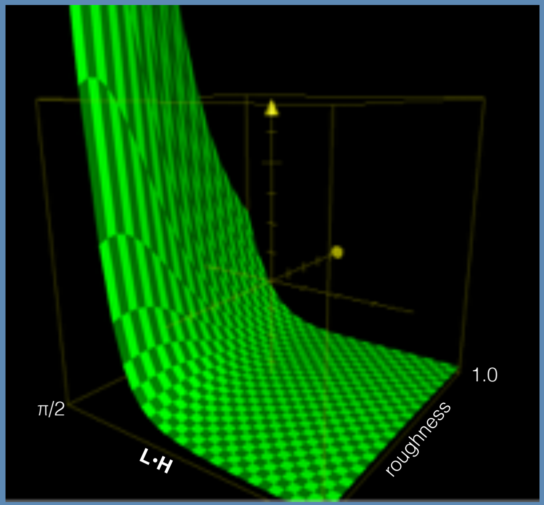


V*F together

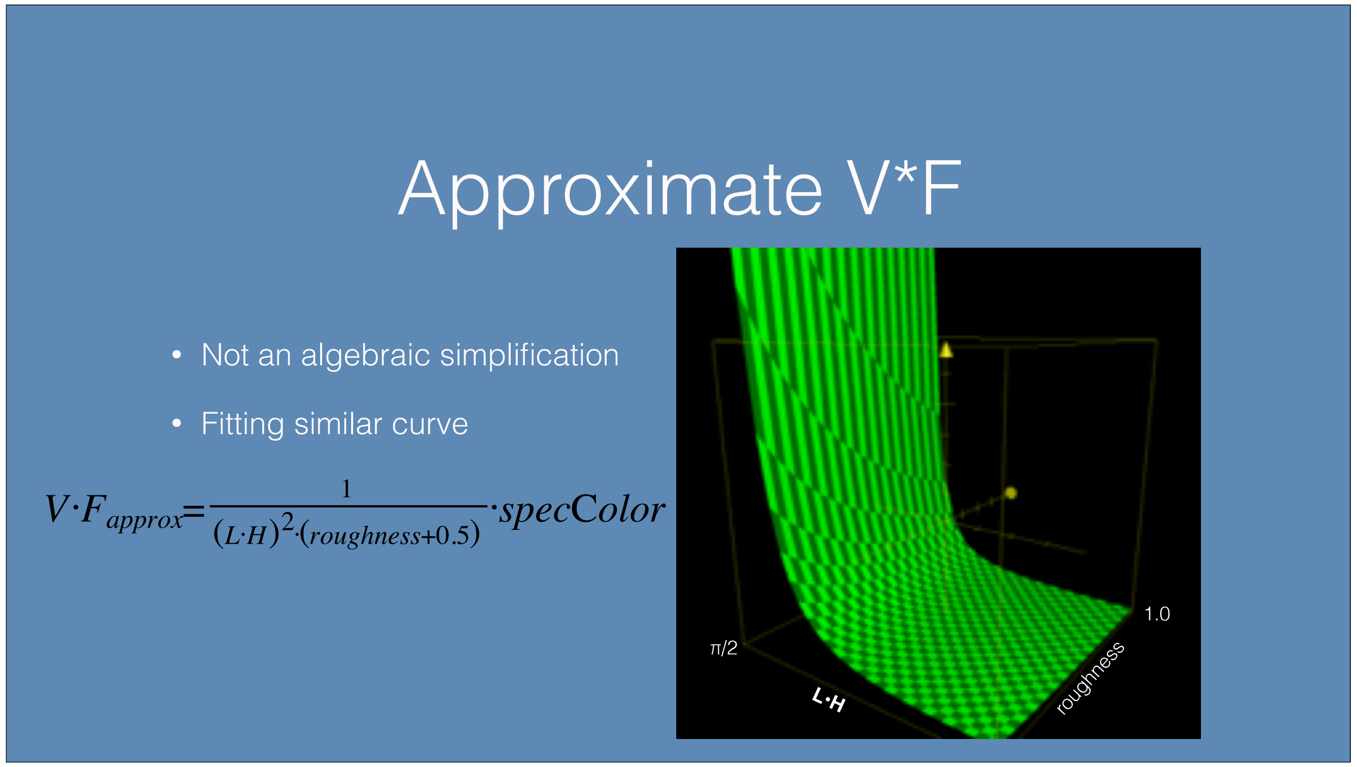
- Modified KSK and Schlick Fresnel depend on **L•H**
- Fuse them together

$$V.F=\frac{(1\text{-}L.H)^5}{(L.H)^2.(1\text{-}roughness^2)+roughness^2}$$

- Cheaper approximation?



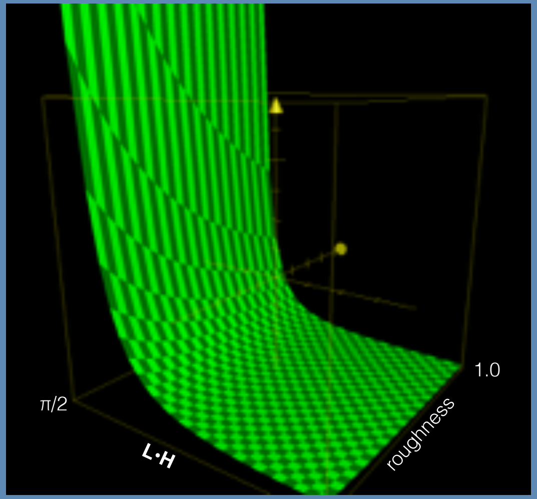
Both Visibility and Fresnel are responsible for shaping highlight at grazing angles to light and off-peak response. In approximate forms both depend on the L•H. Thus it is logical to combine them and seek for cheaper approximation for this combination.
Lets graph both modified KSK V and Schlick F terms (fixed for particular index of refraction) as arguments of Roughness and L•H to have a better visual intuition, if we could approximate it cheaper. Shape doesn’t sound too complex.



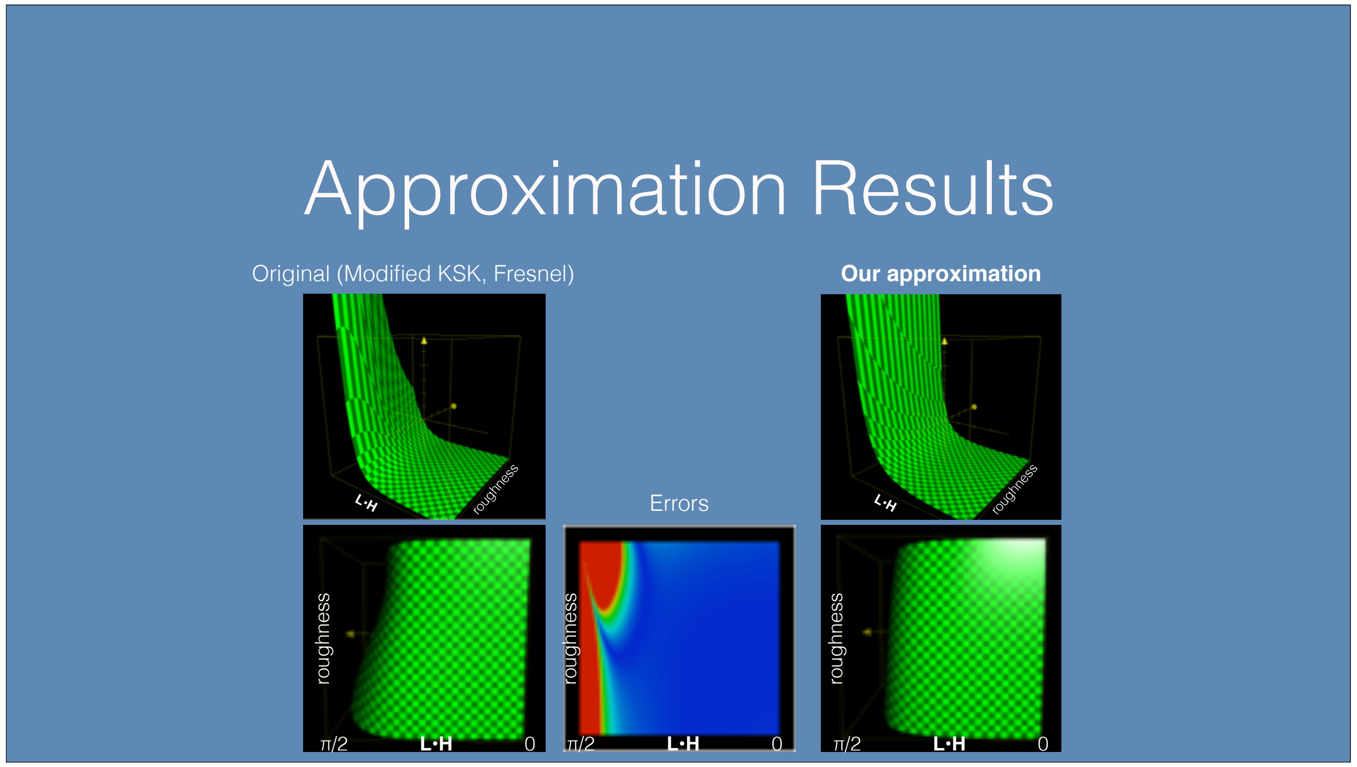
Approximate V*F

- Not an algebraic simplification
- Fitting similar curve

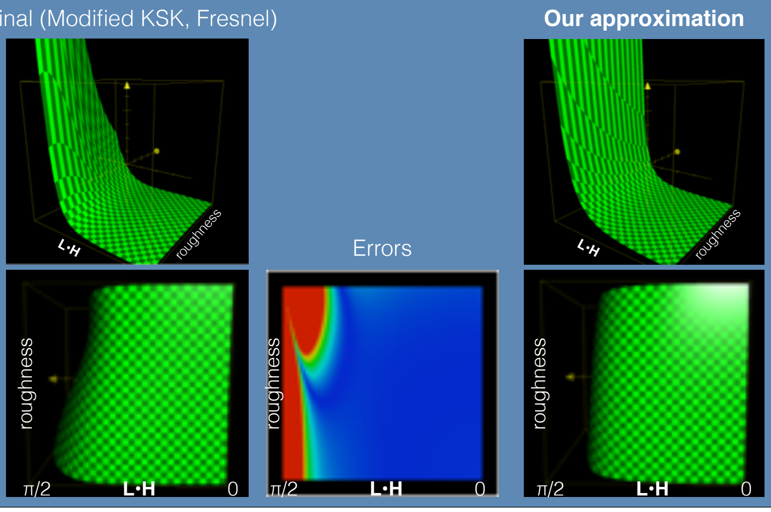
$$V \cdot F_{approx} = \frac{1}{(L \cdot H)^2 \cdot (roughness + 0.5)} \cdot specColor$$



As it turns out it is possible to find a simpler formulae which provides us with similar features as the one on the previous slide - it gives plateau when L•H is close to 0 and slopes rapidly at grazing angles when angle between L and H vectors are close to 90 degrees - light is grazing the surface and shines almost opposite of camera direction.



Approximation Results



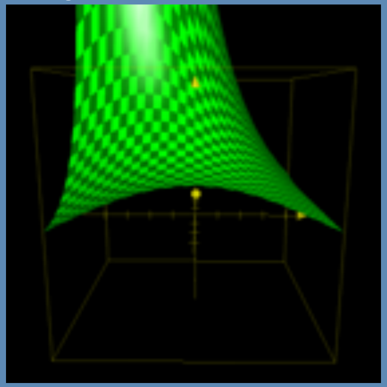
Approximation formula captures another feature that sloping depends on the Roughness. It is not faithful to original slope, providing more linear dependency. You can see from error plot that it is towards very rough materials and grazing angles were errors are significant (red). Blue means no errors.
But on overall approximation does the job - it maintains off-peak specular and it is stronger for rough surfaces. Despite numerical differences, approximation has enough slope to compensate N•L term outside BRDF to keep specular strong close to terminator unlike sp called “implicit” G term.

How did we come up with approximation equation?
Both modified KSK and Schlick Fresnel equations are pretty tight, so it’s hard to do any straightforward mathematical simplifications. Instead several equations that intuitively could fit the shape were tried and parameters were chosen with simple offline MachineLearning algorithm.
I think there is still room for improvement here. I’d like to encourage for further search of simple approximation and tighter parameter fitting to reduce error of combined V and F terms.

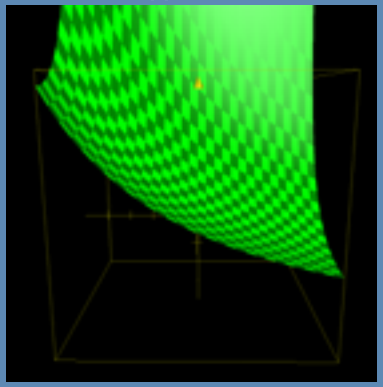
Approximate V*F

- Good for Dielectrics, but diverge for Metals

Original (Modified KSK, Fresnel)



Our approximation

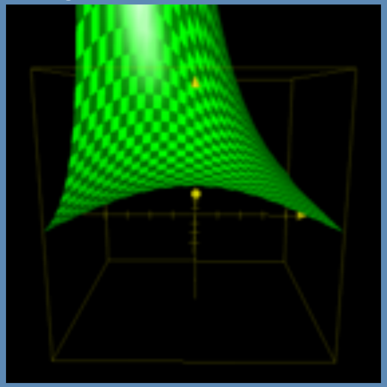


Because of our hack with applying specular color at the end results diverge from ground truth for Metals.

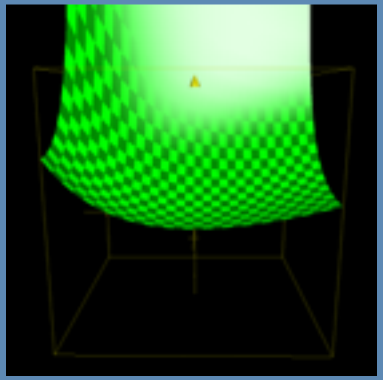
Approximate V*F

- Can be improved with couple more ops, but does not matter in practice

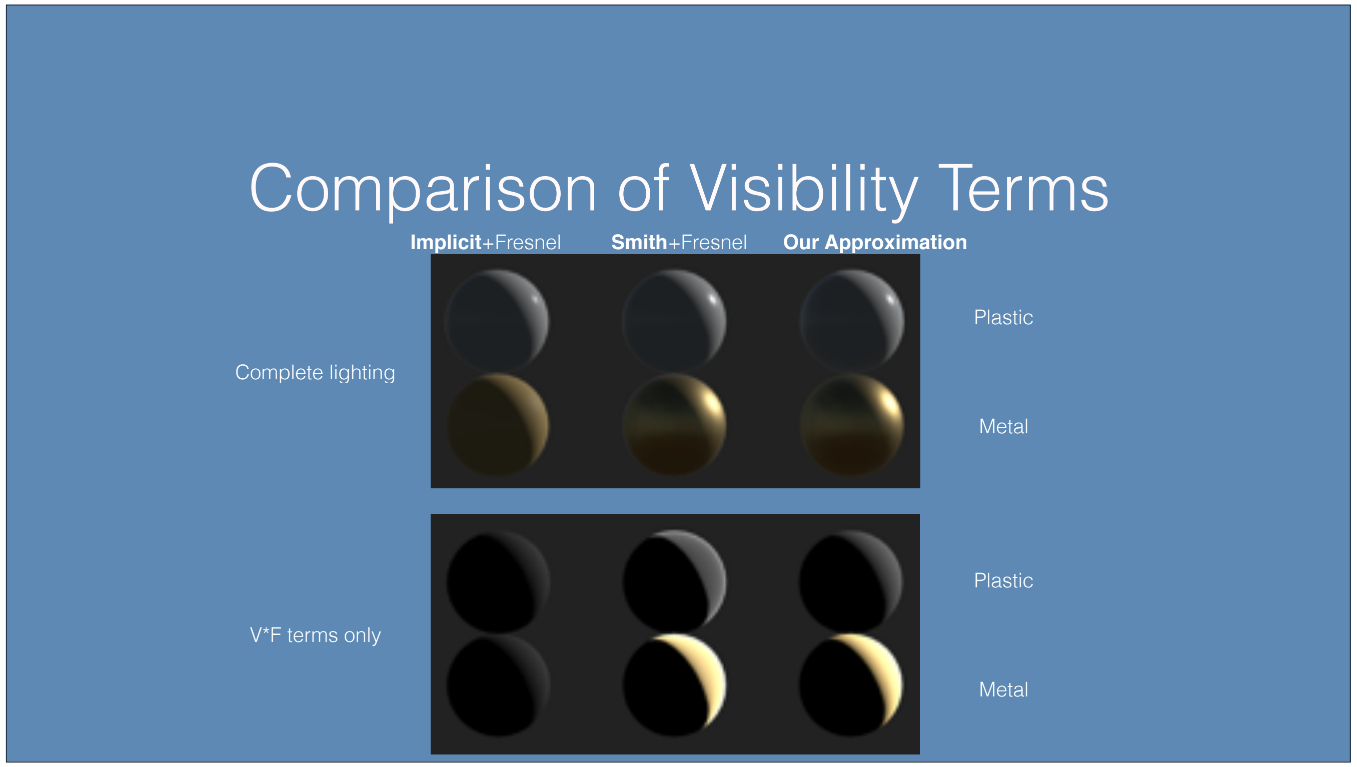
Original (Modified KSK, Fresnel)



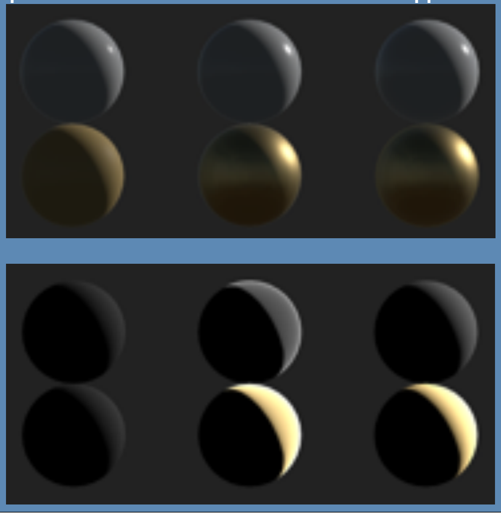
Our approximation*



Shape of the approximation can be improved with just additional linear transformation, but in practice it is not worth it. Visual results are satisfying for what I'd expect on Mobile ;)

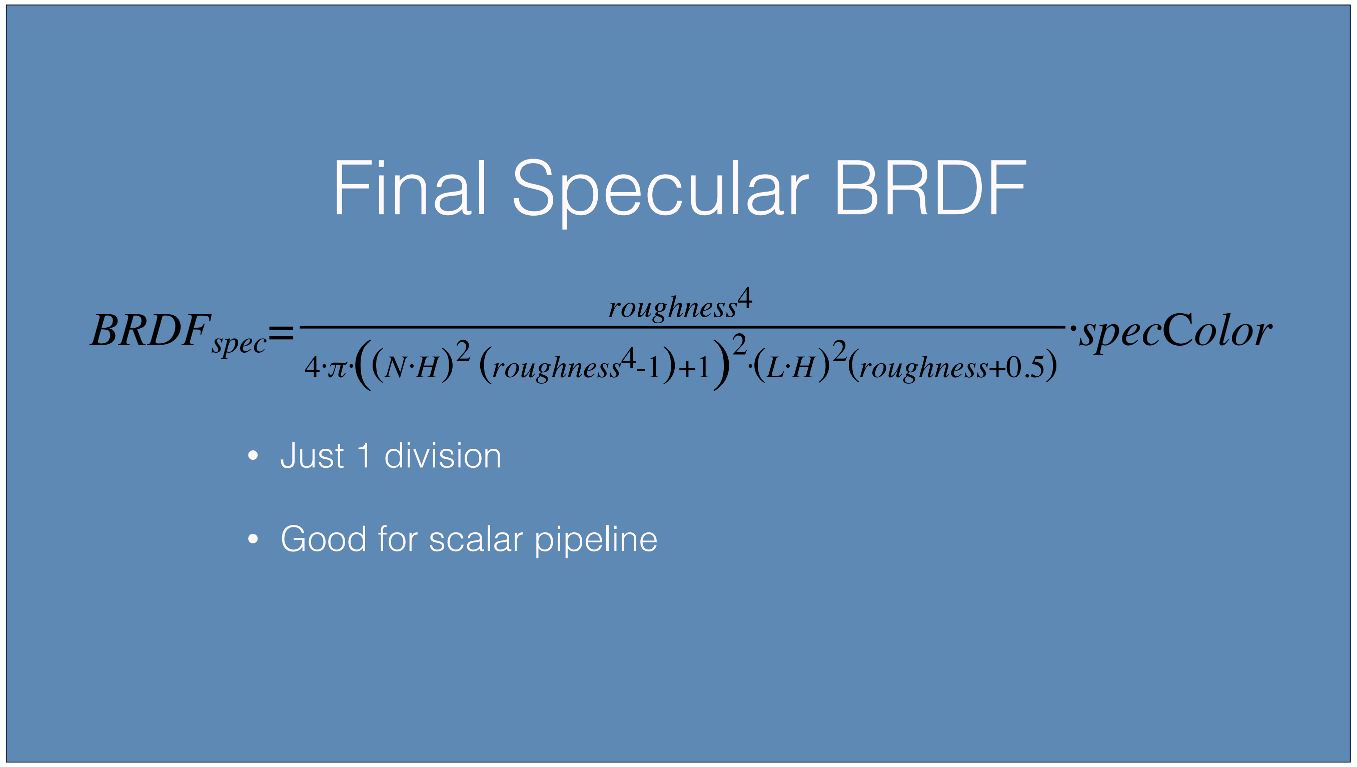


Comparison of Visibility Terms



Here is the visual comparison of our approximation versus Implicit (cheapest possible) V term and Smith V term which is commonly used in high-end PBS shaders. As you can see our approximation is significantly better at capturing important visual cues of the specular than Implicit for both dielectric and conducting materials. It is hardly distinguishable from combination of “proper” V and F terms for untrained eye.

To understand how our approximation behaves under the hood bottom image shows debug visualization of just V and F terms. For example, we can see that very strong amplification of Metalic highlight is present and that highlight can come closer to the terminator for non-Metals. There are some subtle features of Smith and Fresnel missing, but it’s hardly noticeable in the final image.



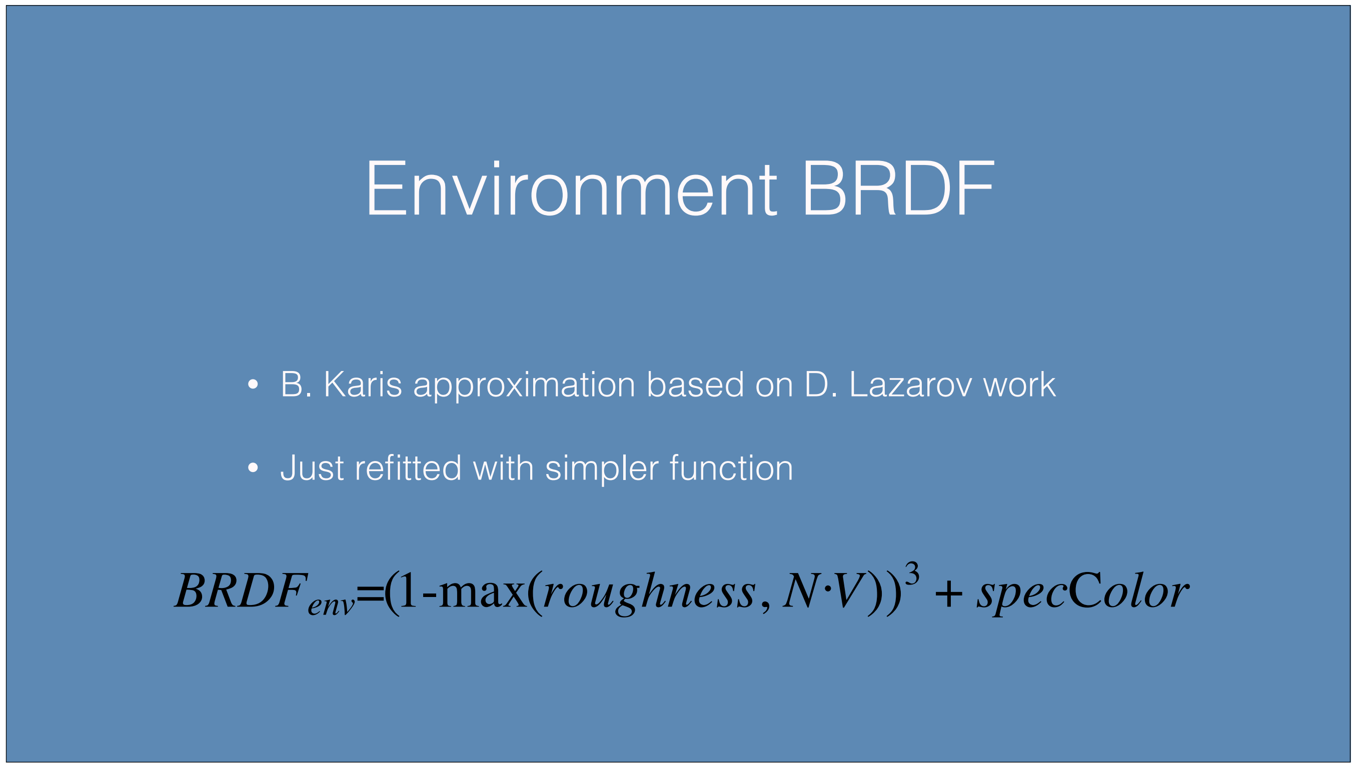
Final Specular BRDF

$$BRDF_{spec} = \frac{roughness^4}{4 \cdot \pi \cdot \left((N \cdot H)^2 \, (roughness^4 - 1) + 1 \right)^2 \cdot (L \cdot H)^2 (roughness + 0.5)} \cdot specColor$$

- Just 1 division
- Good for scalar pipeline

Lets bring all our terms of specular BRDF together. Proposed V*F approximation fits perfectly with GGX and all terms share cost of the same division. Whole specular BRDF can be evaluated as operations on one scalar followed by vector multiplication at the very end.

Perfect!



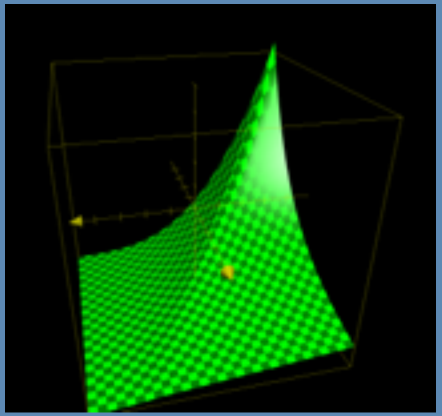
Environment BRDF

- B. Karis approximation based on D. Lazarov work
- Just refitted with simpler function

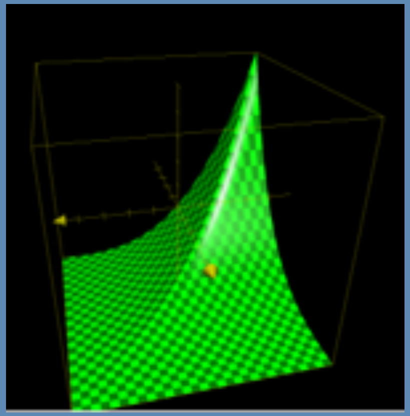
$$BRDF_{env} = (1 - \max(roughness, N \cdot V))^3 + specColor$$

Next relatively expensive part of the shader is specular BRDF applied to glossy environment reflections (Image Based Lighting).
Since proper evaluation of such BRDF requires integration across all potential reflected directions, common solution for high-end realtime graphics is to precompute results and store the in the lookup-texture.
Luckily Brian Karis from Epic already did a great job at devising numerical approximation for such integral based on Lazarov's work analyzing Geometrical term which allows to obtain results cheaper than with lookup-texture (remember ALU/TEX ratio). However that approximation requires complex exponent that ideally we would like to avoid on Mobile.

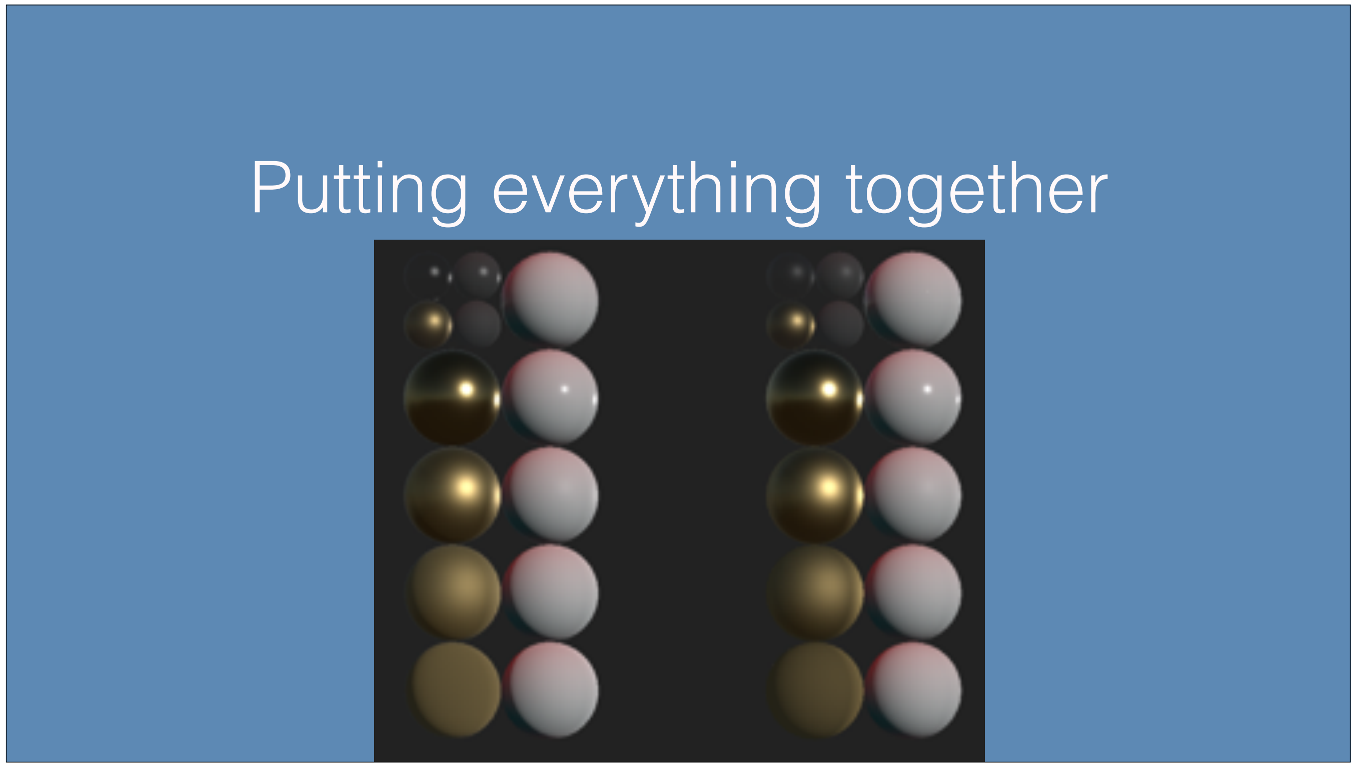
Environment BRDF



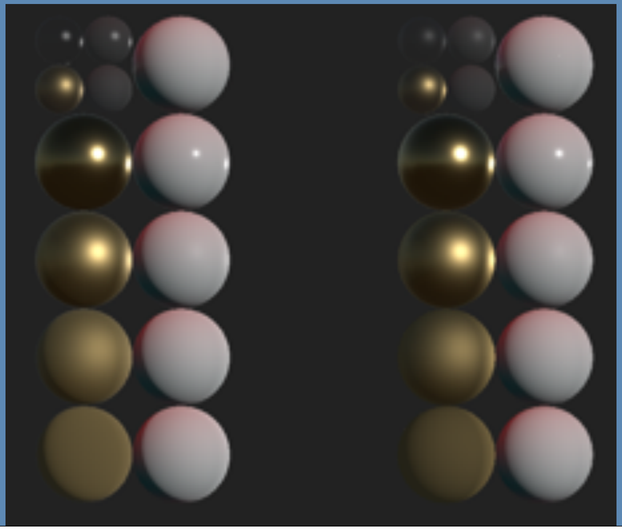
$(1\text{-}\max(\mathit{roughness}, N\cdot V))^3$



Again by graphing approximated environment BRDF it is easy to see that we can come up with even simpler equation with minimal errors at the grazing angles of very rough materials.
Our approximation has less simple ops and completely avoids any complex operations such as evaluation of exponent.



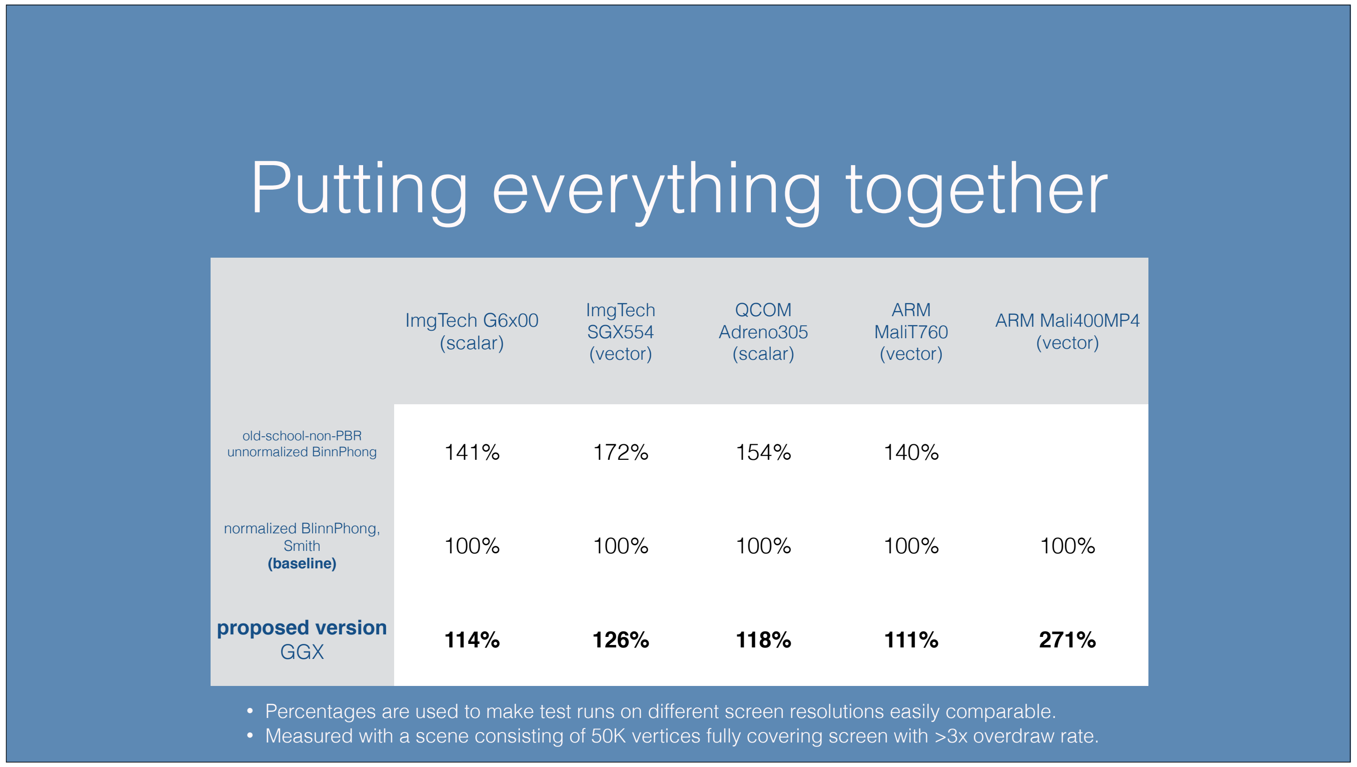
Putting everything together



Finally we can compare our approximations against the shader used for desktop GPUs.

On the left side you see GGX specular with Smith visibility term, proper Fresnel and nice glossy reflections. On the right side is our approximation that never even evaluates Fresnel in the full form.

You can see that very rough metals don’t look as good and specula reflection is slightly dimmer at grazing angles in case of non-metals, but overall result is not too bad for Mobiles.



Putting everything together

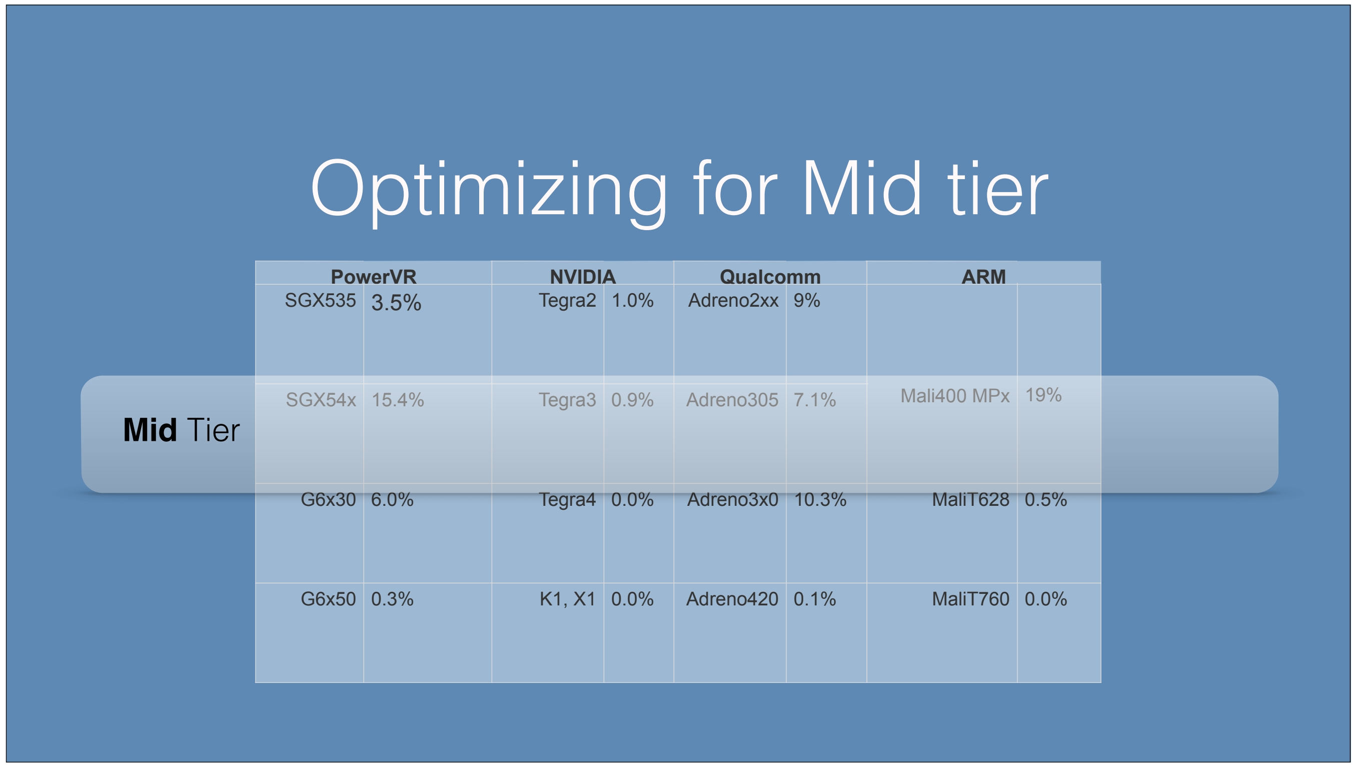
	ImgTech G6x00 (scalar)	ImgTech SGX554 (vector)	QCOM Adreno305 (scalar)	ARM MaliT760 (vector)	ARM Mali400MP4 (vector)
old-school-non-PBR unnormalized BinnPhong	141%	172%	154%	140%	
normalized BlinnPhong, Smith **(baseline)**	100%	100%	100%	100%	100%
proposed version GGX	**114%**	**126%**	**118%**	**111%**	**271%**

- Percentages are used to make test runs on different screen resolutions easily comparable.
- Measured with a scene consisting of 50K vertices fully covering screen with >3x overdraw rate.

Let’s look at performance numbers comparing our approach on multiple high-end and medium-end GPUs across several architectures. Data was gathered on a scene consisting of 50K vertices that fully cover whole screen with average 3x overdraw rate - rather ordinary “house” object at close up with full set of PBS textures including detailed albedo and ambient occlusion. Numbers are presented as percentage of baseline PBS implementation. Percentages were used to make comparison easier regardless of device screen resolution.

You can see that noticeable speed up is achieved and performance is somewhere between horrible old-school unnormalized Phong and full albeit already reasonably optimized by many contributors solution for PBS. It is worth noticing outlier in this table - Mali400. As far as I understand the reasons for such significant performance improvement is that our approximations significantly reduced number of temporary registers being used on this architecture, which was significant bottleneck before.

Given high ALU/TEX ratios of modern mobile GPUs further optimizations should concentrate on reducing number of textures (if possible) used in the shading.

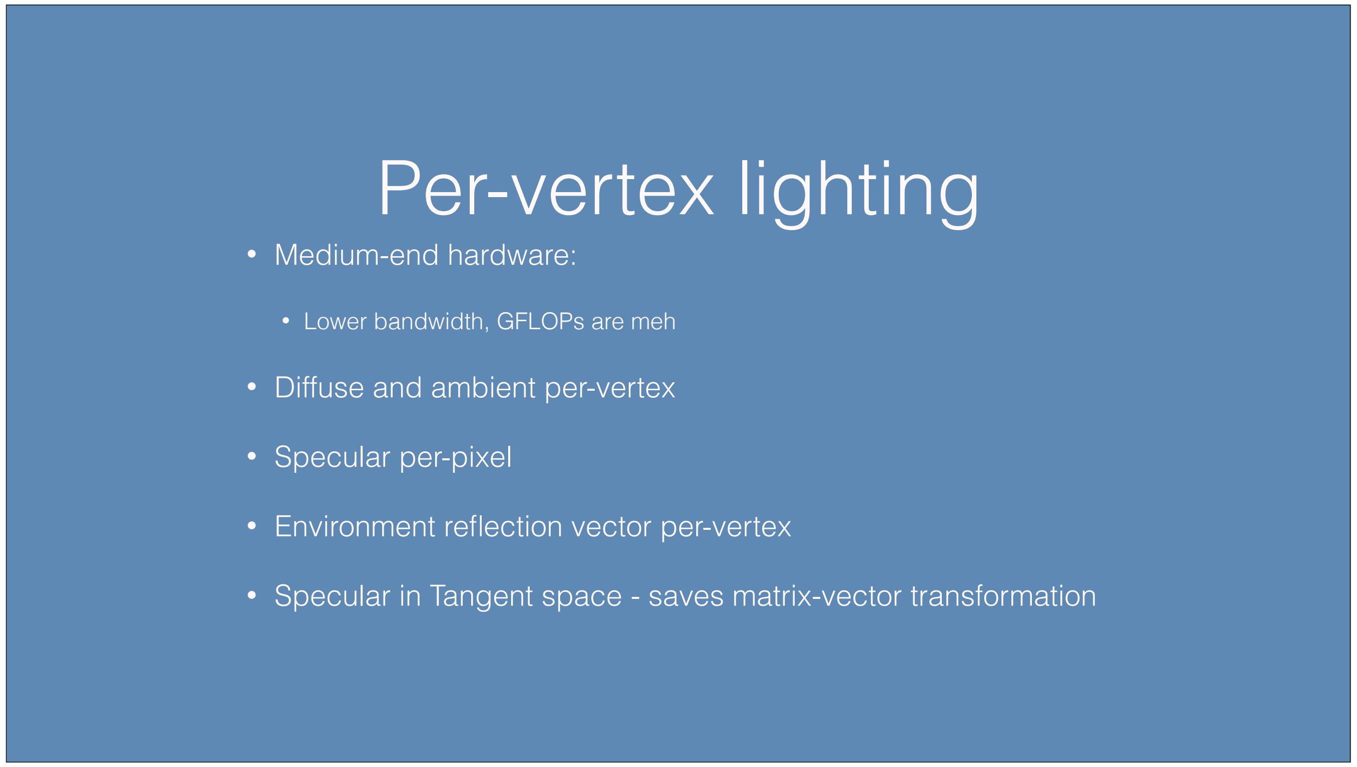


Optimizing for Mid tier

Mid Tier

PowerVR		NVIDIA		Qualcomm		ARM	
SGX535	3.5%	Tegra2	1.0%	Adreno2xx	9%		
SGX54x	15.4%	Tegra3	0.9%	Adreno305	7.1%	Mali400 MPx	19%
G6x30	6.0%	Tegra4	0.0%	Adreno3x0	10.3%	MaliT628	0.5%
G6x50	0.3%	K1, X1	0.0%	Adreno420	0.1%	MaliT760	0.0%

But is it enough for medium-end devices?



Per-vertex lighting

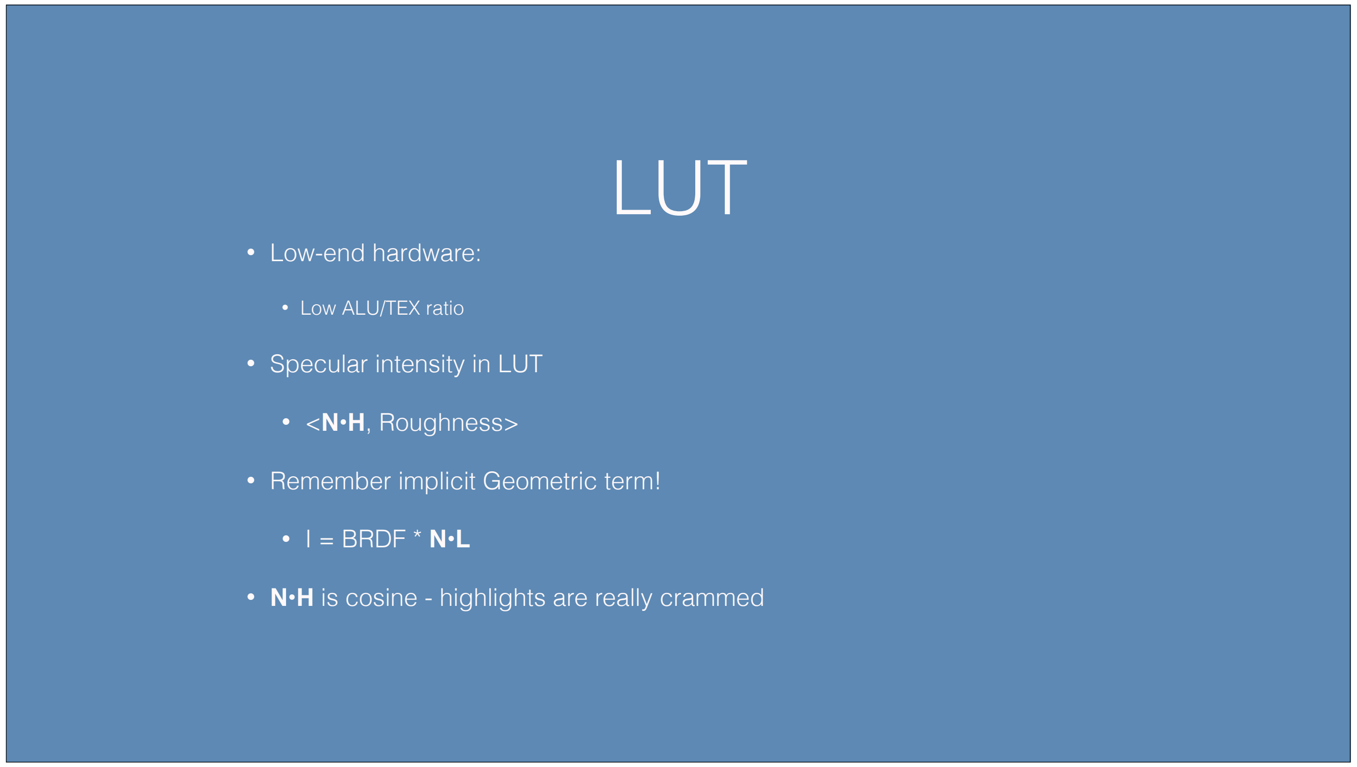
- Medium-end hardware:
  - Lower bandwidth, GFLOPs are meh
- Diffuse and ambient per-vertex
- Specular per-pixel
- Environment reflection vector per-vertex
- Specular in Tangent space - saves matrix-vector transformation

For further performance gains we need to employ more drastic approaches - moving as much as possible to vertex shader leaving only specular BRDF part per-pixel. Glossy reflections and evaluating ambient stored as Spherical Harmonics require pixel shader to do lighting in world space. However if we're satisfied with lower quality of them per-vertex, we can go back to evaluating specular directly in tangent space saving matrix-vector transformation per-pixel.

Optimizing for Low-end

Low-end Tier

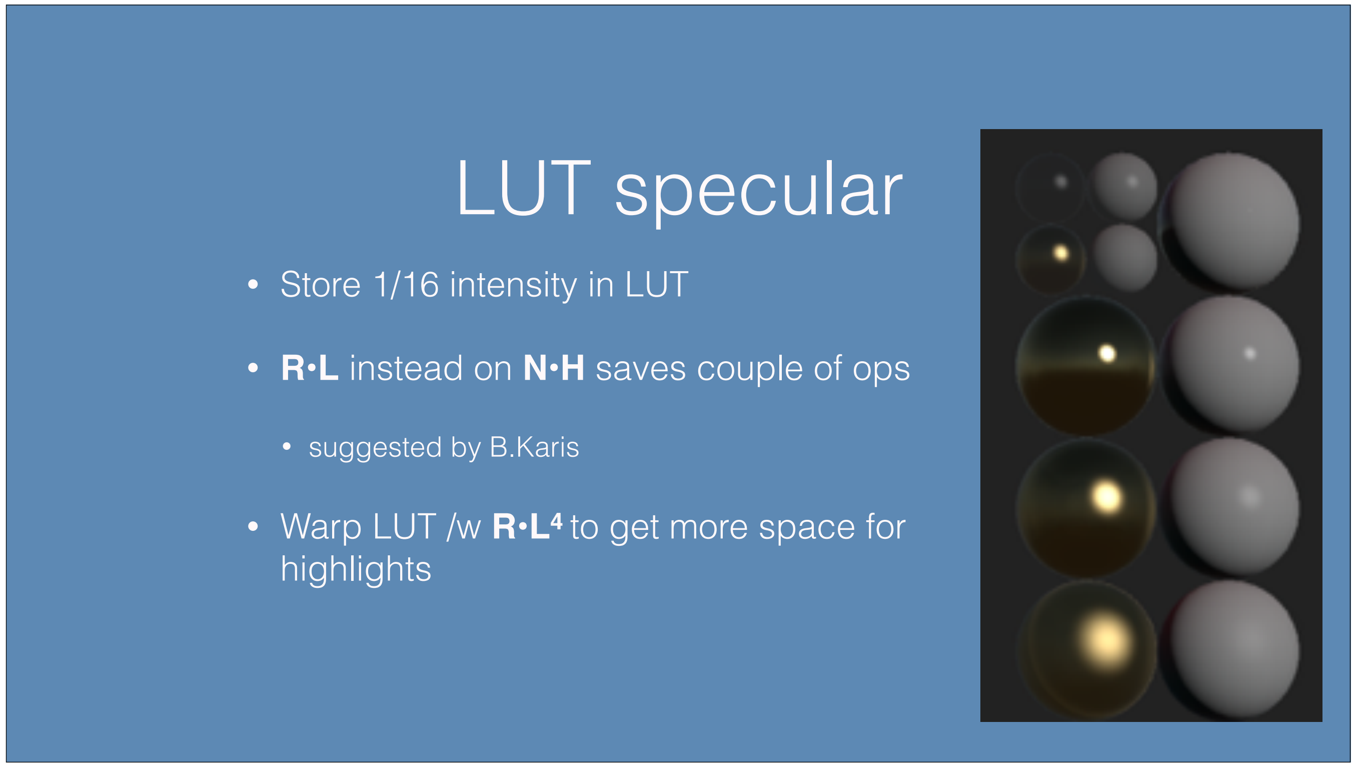
PowerVR		NVIDIA		Qualcomm		ARM	
SGX535	3.5%	Tegra2	1.0%	Adreno2xx	9%		
SGX54x	15.4%	Tegra3	0.9%	Adreno305	7.1%	Mali400 MPx	19%
G6x30	6.0%	Tegra4	0.0%	Adreno3x0	10.3%	MaliT628	0.5%
G6x50	0.3%	K1, X1	0.0%	Adreno420	0.1%	MaliT760	0.0%



LUT

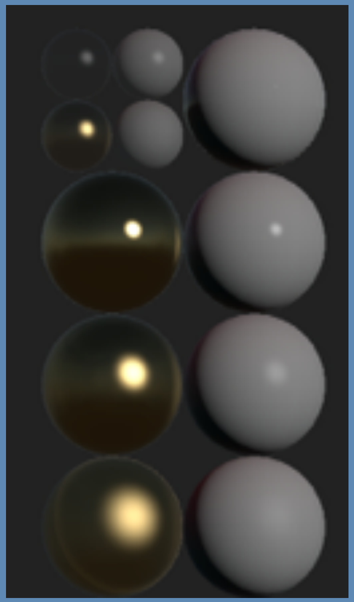
- Low-end hardware:
  - Low ALU/TEX ratio
- Specular intensity in LUT
  - <**N•H**, Roughness>
- Remember implicit Geometric term!
  - I = BRDF * **N•L**
- **N•H** is cosine - highlights are really crammed

For oldest GPUs we have pretty low computational power, therefore it makes sense to fallback to evaluating specular BRSD via lookup textures. Of course that means no Fresnel , no nice specular at grazing angles and no tight small highlights when N•H is close to 0.
Nice thing that we can have nice GGX falloff, normalized specular intensity basically and even gamma adjustments “for free”.



LUT specular

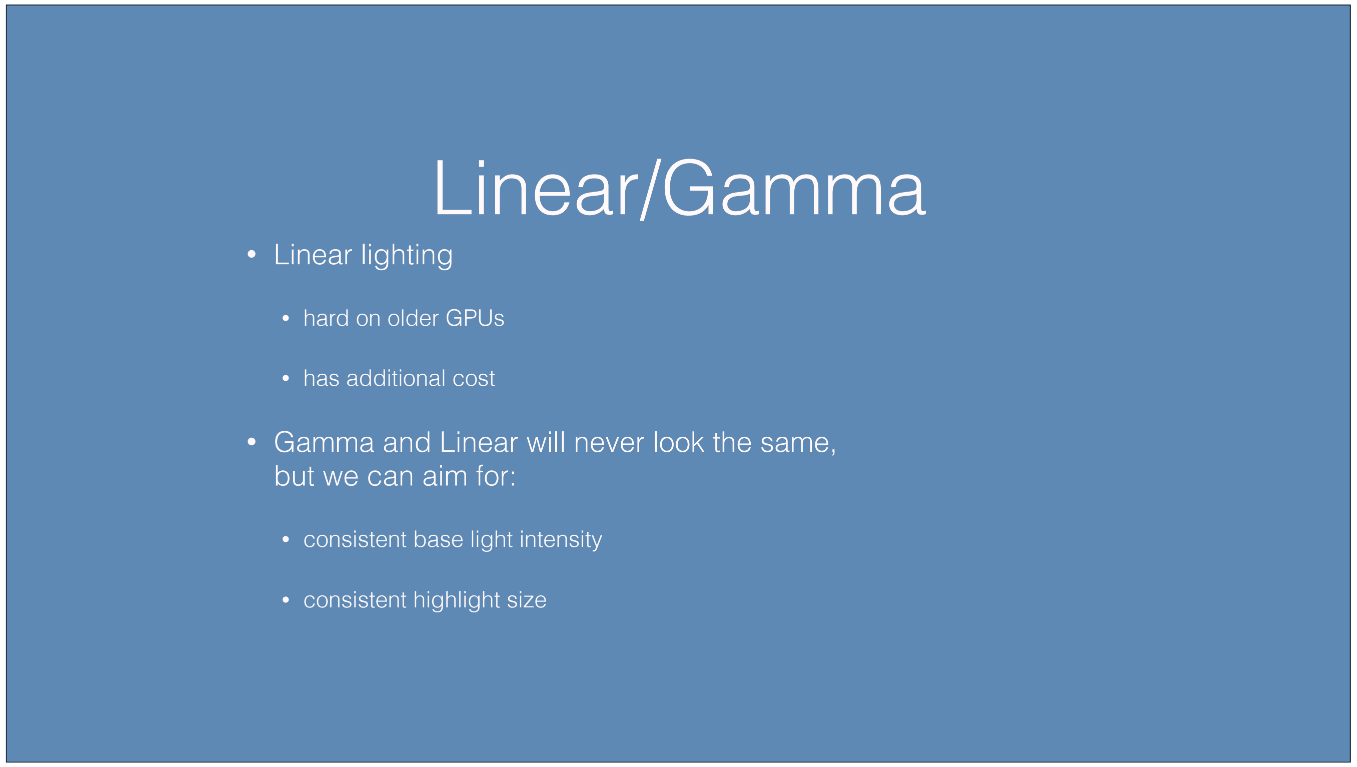
- Store 1/16 intensity in LUT
- **R•L** instead on **N•H** saves couple of ops
  - suggested by B.Karis
- Warp LUT /w **R•L**^{4} to get more space for highlights



It is important to keep in mind however that results of BRDF have to be multiplied by N•L which means 0..1 range is not enough. To simulate higher range we have to store results divided by certain constant and multiply back to full range in the shader. Given limited 8bit precision this comes at quality tradeoff. We found 16 to be reasonable tradeoff for quality.

We have to warp LUT on one axis to get a bit more precision for sharp highlights. Simply multiplying N•H or R•L with itself couple of times gives bearable quality.

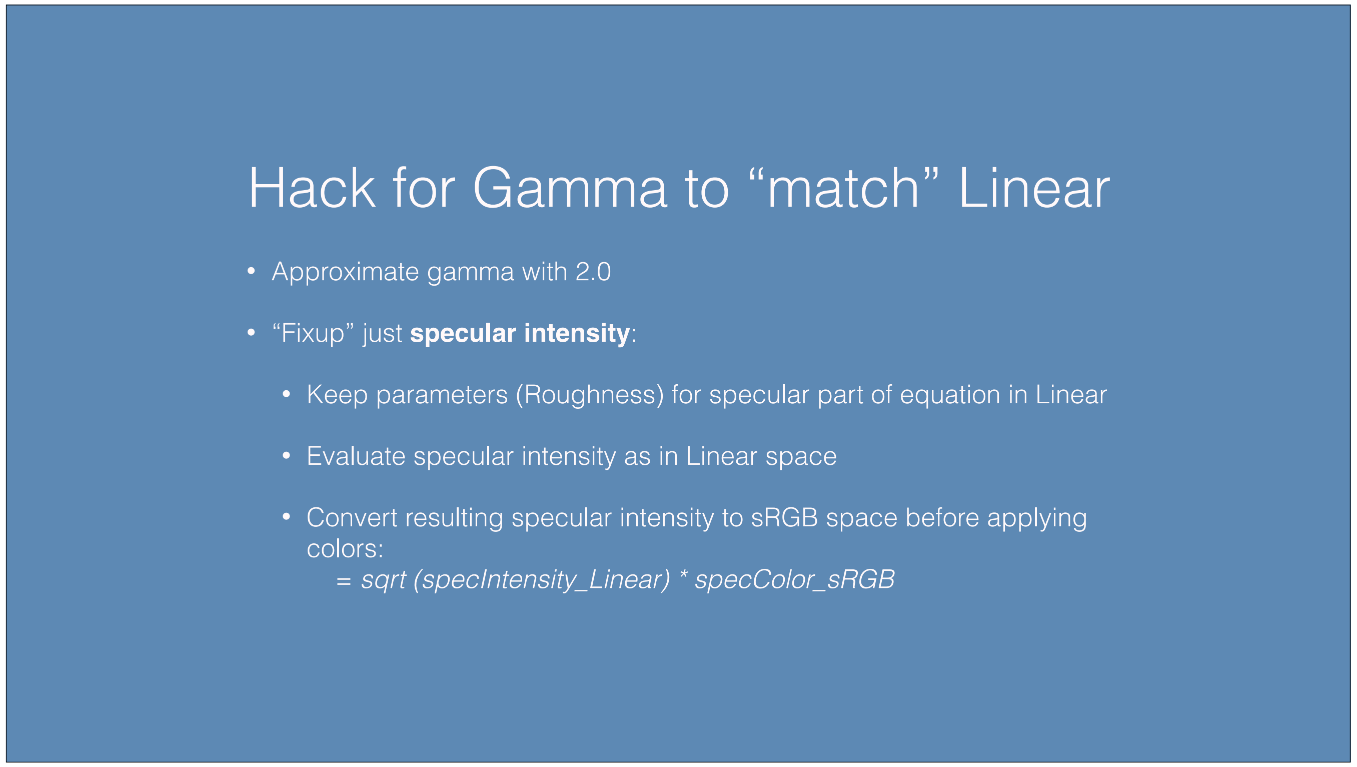
- PBR challenges on Mobile
- What hardware are we optimizing for?
- Faster BRDF
- **Linear/Gamma**
- **Environment Reflections**



Linear/Gamma

- Linear lighting
  - hard on older GPUs
  - has additional cost
- Gamma and Linear will never look the same, but we can aim for:
  - consistent base light intensity
  - consistent highlight size

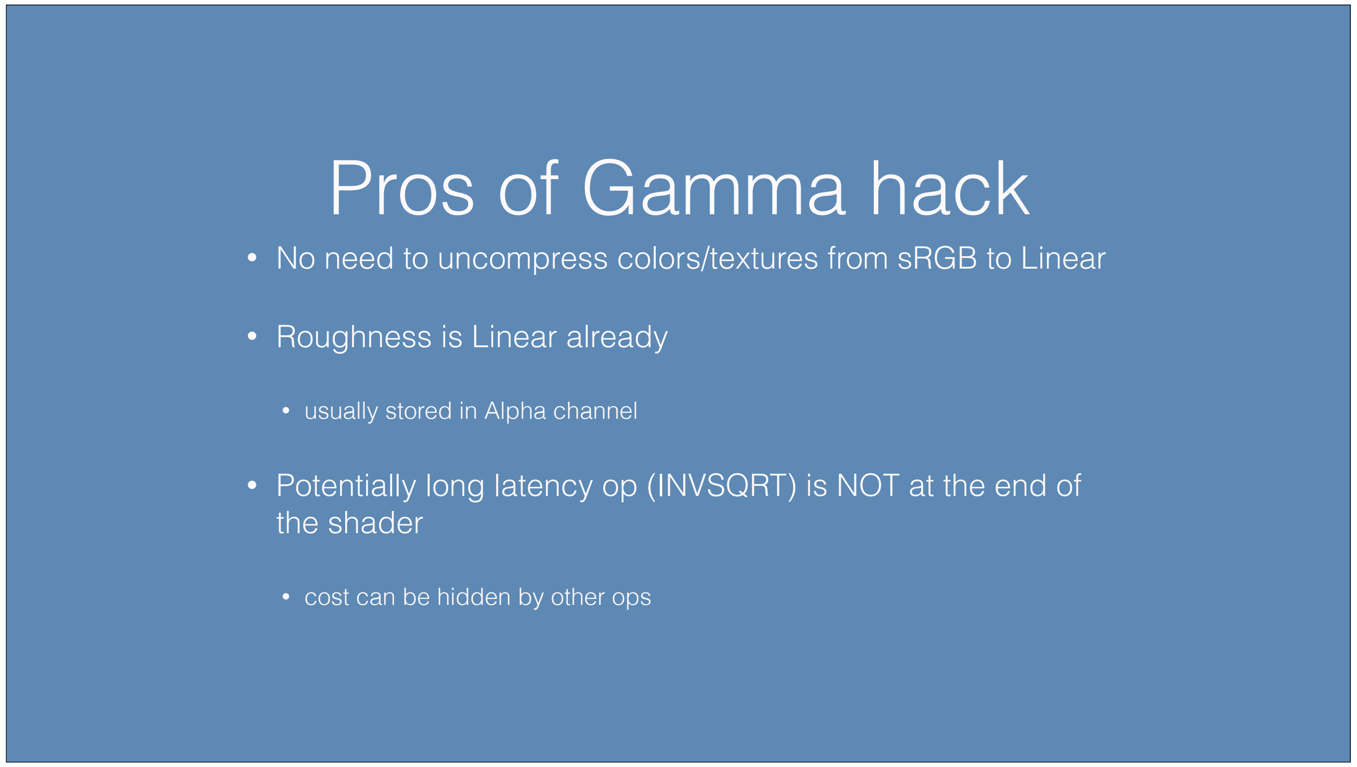
Linear lighting is out of reach on many GPUs because of performance or lack of features. And even on high-end Mobile GPUs Linear lighting comes at additional cost that sometimes is desirable to avoid.



Hack for Gamma to “match” Linear

- Approximate gamma with 2.0
- “Fixup” just **specular intensity**:
  - Keep parameters (Roughness) for specular part of equation in Linear
  - Evaluate specular intensity as in Linear space
  - Convert resulting specular intensity to sRGB space before applying colors:
    = *sqrt (specIntensity_Linear) * specColor_sRGB*

Non-linearized lighting significantly affects gradients. For some gradients such as slowly changing diffuse especially in the scene with multiple light sources - human eye can tolerate errors quite well. However very sharp gradients such as shapes of specular highlights are immediately noticeable.
Given that, we can do quality trade-off and gamma correct only what is most noticeable - specular highlights.

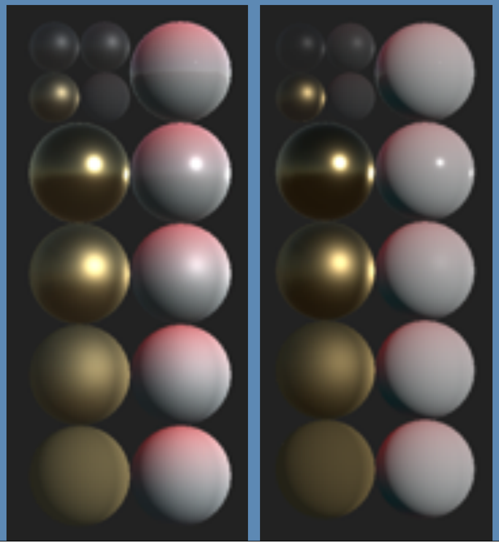


Pros of Gamma hack

- No need to uncompress colors/textures from sRGB to Linear
- Roughness is Linear already
  - usually stored in Alpha channel
- Potentially long latency op (INVSQRT) is NOT at the end of the shader
  - cost can be hidden by other ops

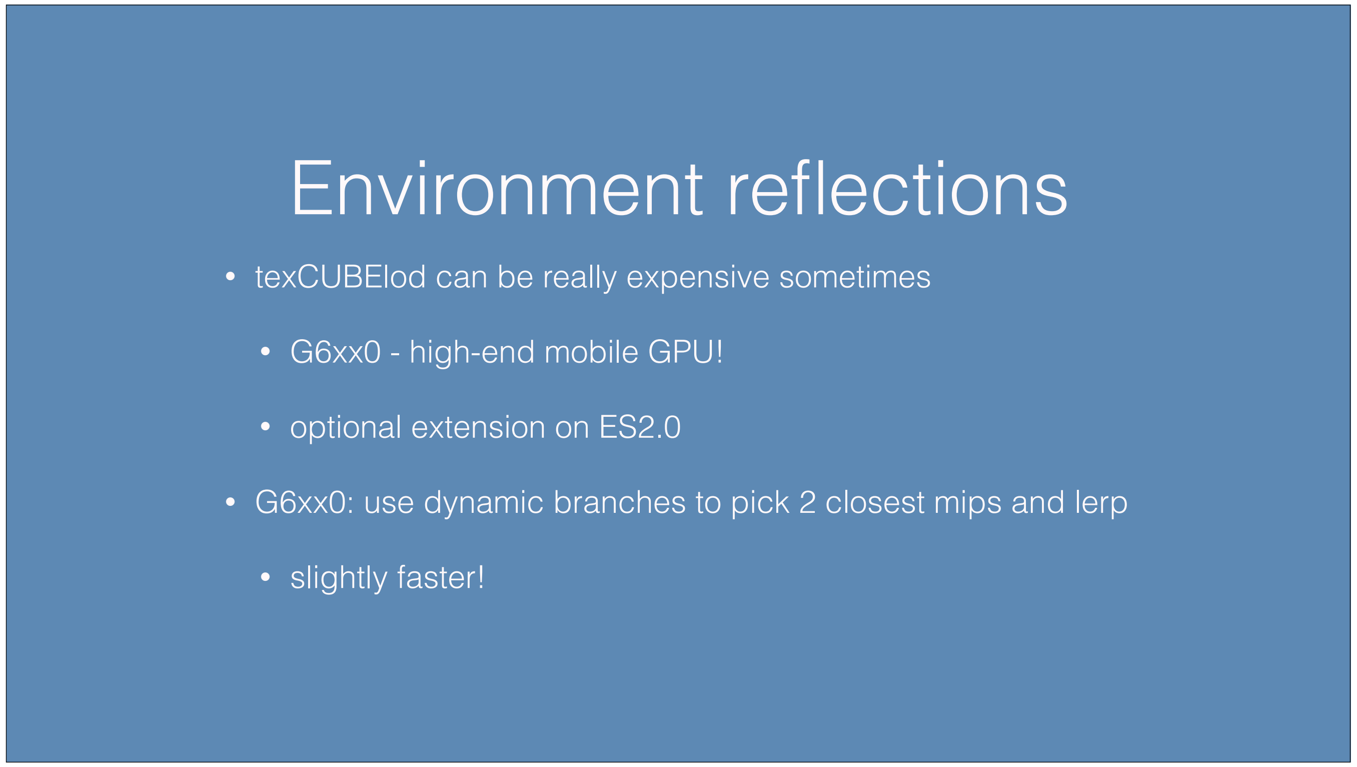
There are certain handy aspects for that matter: the way we traditionally store Roughness (as linear value in alpha channel) and the fact that, if we do relatively expensive gamma correction in the middle of the lighting equation instead of the final result, it is easier for GPU to hide its latency rather when that happens at the very end of the pixel shader.

Gamma vs Linear



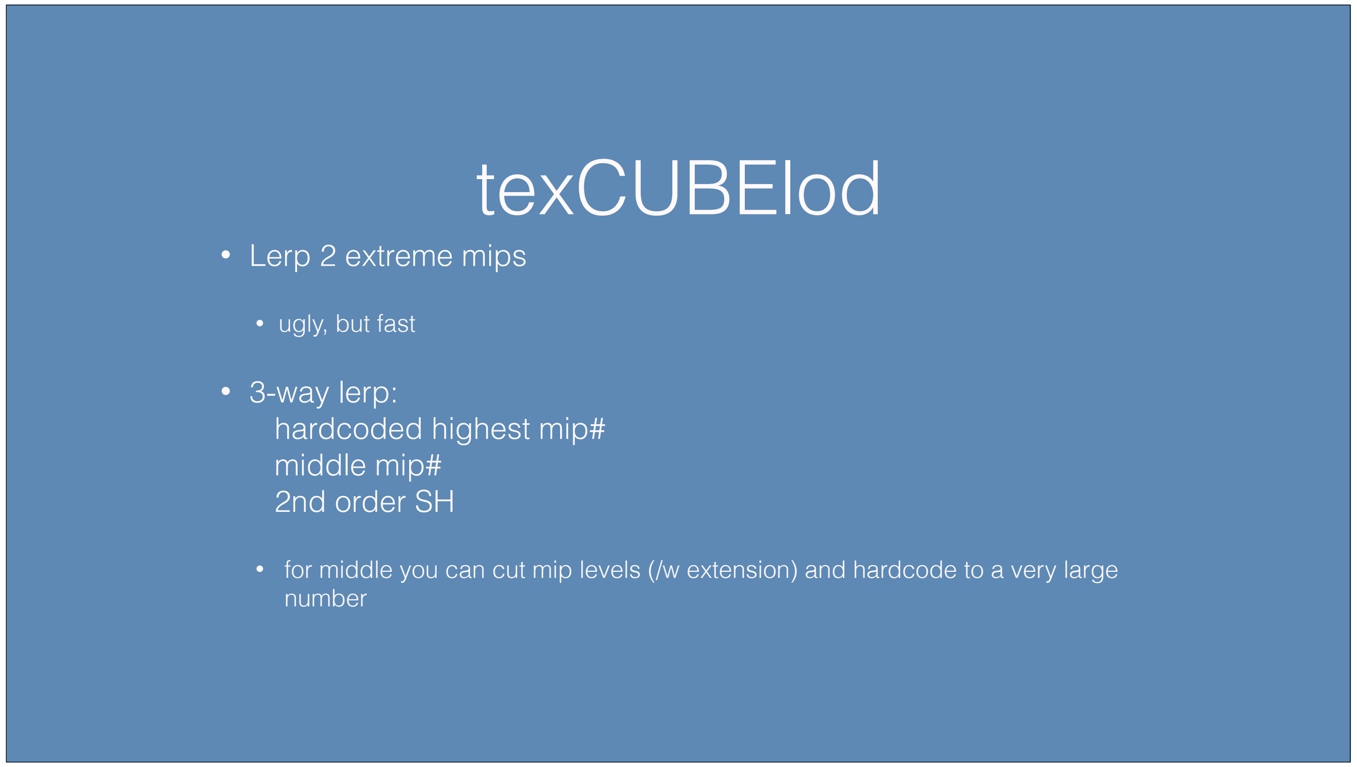
On the left we have most of the lighting evaluated in the wrong gamma space, except we fixup specular part with 2.0 gamma approximation. On the right lighting is correctly evaluated in linear space.

Results on the left are wrong, but bearable to human eye. Shape and size of the specular is close to ground truth. That means artists do not need to redo their content and game could potentially switch between different rendering modes on demand.


Environment reflections

- texCUBElod can be really expensive sometimes
  - G6xx0 - high-end mobile GPU!
  - optional extension on ES2.0
- G6xx0: use dynamic branches to pick 2 closest mips and lerp
  - slightly faster!

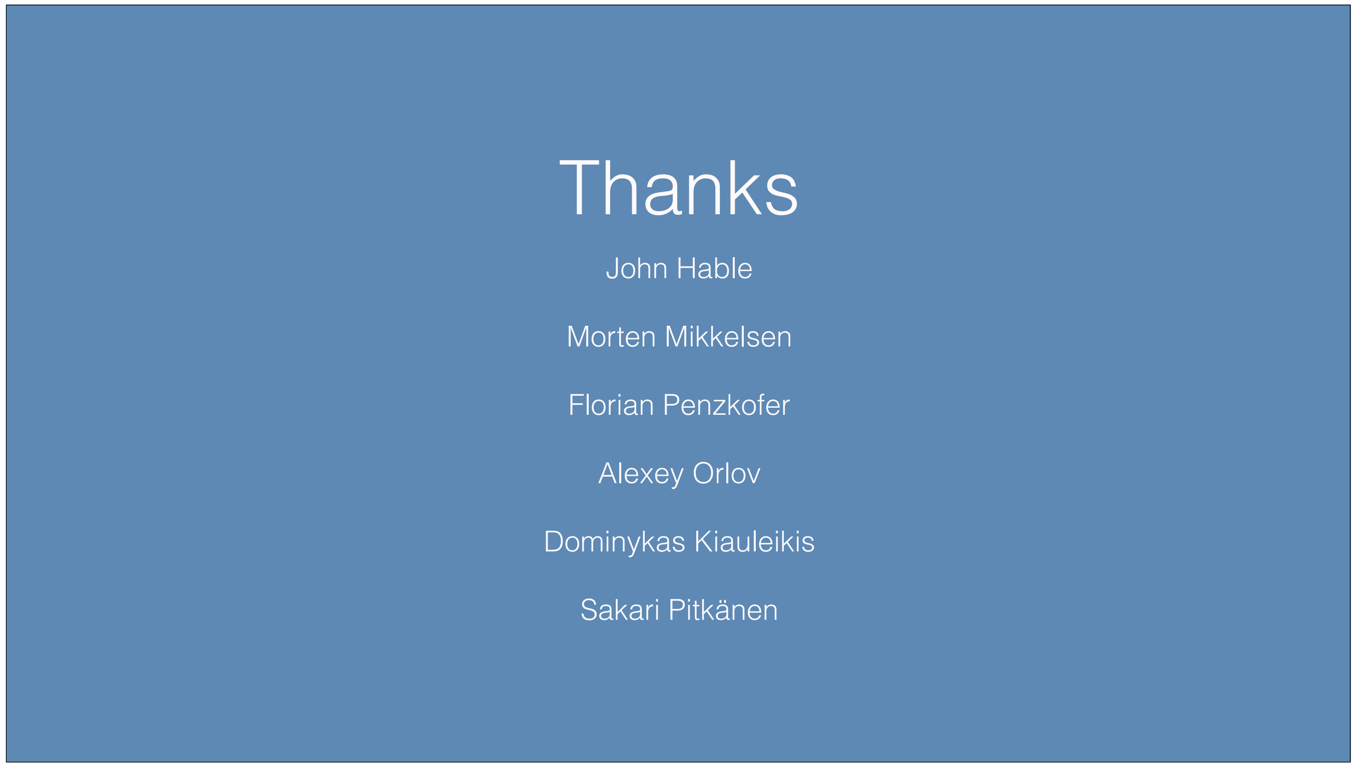
To end the talk lets quickly revisit glossy reflections. Even on some high-end Mobile GPU architectures texCUBElod operation can be expensive and on low end Mobile GPUs (OpenGLES2.0 not supporting proper extension) it can be missing altogether.



texCUBElod

- Lerp 2 extreme mips
  - ugly, but fast
- 3-way lerp:
  hardcoded highest mip#
  middle mip#
  2nd order SH
  - for middle you can cut mip levels (/w extension) and hardcode to a very large number

We can “wing it” by simply reading from two predefined extreme mips (or two separate textures) and linearly interpolating results. Super fast, but visuals are not satisfying.
Adding 3 steps instead of 2 significantly improves visual result. To save on one cubemap read we can store the most convoluted mip in shader constants in the form of 2nd order Spherical Harmonics. Relatively fast and gives good quality.



Thanks

John Hable

Morten Mikkelsen

Florian Penzkofer

Alexey Orlov

Dominykas Kiauleikis

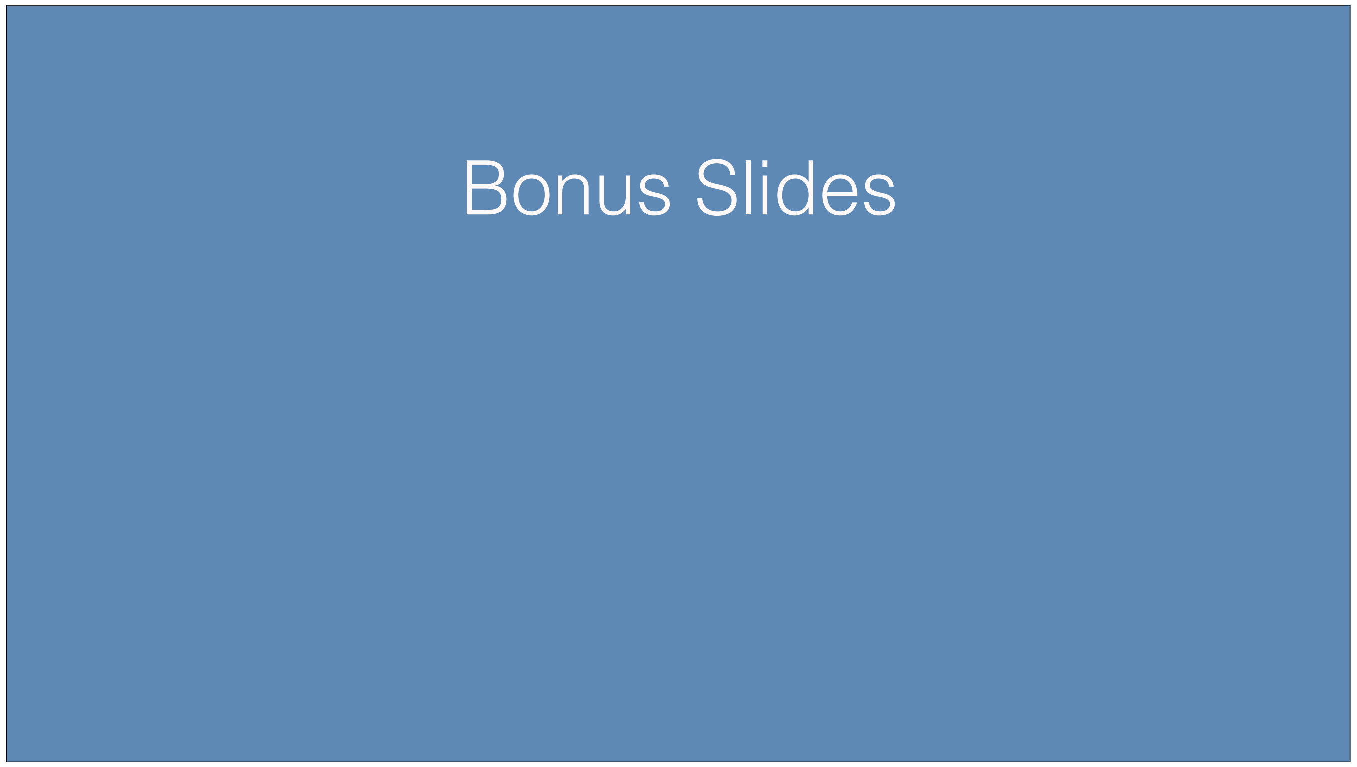
Sakari Pitkänen

I’d like to thank everyone who gave useful feedback, thought material and performance analysis for our optimizations.

Our optimizations will be available as open source shaders.

References

1. Morten Mikkelsen, “Microfacet Based Bidirectional Reflectance Distribution Function”, 2009
2. John Hable, “Optimizing GGX Shaders with dot(L,H)”, 2014, online
3. Christian Schüler, “An Efficient and Physically Plausible Real-Time Shading Model.” ShaderX 7, Chapter 2.5, pp. 175 – 187
4. Brian Karis, “Physically Based Shading on Mobile”, 2014, online
5. Sébastien Lagarde, “Spherical Gaussian approximation for Blinn-Phong, Phong and Fresnel”, 2012, online
6. Kelemen and Szirmay- Kalos, “A Microfacet Based Coupled Specular- Matte BRDF Model with Importance Sampling”, Eurographics 2001
7. Robert Cook and Kenneth Torrance, “A reflectance model for computer graphics”



Bonus Slides

More slides to bore yourself.

OpenGL ES3.0

	PowerVR	NVIDIA	Qualcomm	ARM
4 ~ 8 GFlops 0.2 ~ 1 GP/s	SGX535 iPad, iPhone4	Tegra2	Adreno2xx	Mali400 MPx SGS3 (I9300) SGS2 (I9100)
16 GFlops 2 ~ 3 GP/s	SGX54x iPad2/3, iPhone4s, iPhone5	Tegra3	**Adreno305** SGS4 mini (I9195)	
100 GFlops **4 GP/s**	**G6x30** iPadAir, iPhone5s	Tegra4	**Adreno3x0** Nexus 4, Nexus 5	**MaliT628**
250 GFlops **4 ~ 8 GP/s**	**G6x50** iPadAir2, iPhone6	**K1, X1** Nexus 9, Shield Tablet	**Adreno420**	**MaliT760** SGS6

- Green - GPU with ES3.0 support
- TIP: you can't just use ES2.0 / ES3.0 to determine performance of GPU

Low-end with large share

	PowerVR		NVIDIA		Qualcomm		ARM	
4 ~ 8 GFlops 0.2 ~ 1 GP/s	SGX535	3.5%	Tegra2	1.0%	**Adreno2xx**	**9%**	**Mali400 MPx**	**19%**
16 GFlops 2 ~ 3 GP/s	**SGX54x**	**15.4%**	Tegra3	0.9%	**Adreno305**	**7.1%**		
100 GFlops 4 GP/s	**G6x30**	**6.0%**	Tegra4	0.0%	**Adreno3x0**	**10.3%**	MaliT628	0.5%
250 GFlops 4 ~ 8 GP/s	G6x50	0.3%	K1, X1	0.0%	Adreno420	0.1%	MaliT760	0.0%

- Yellow - Low-end with large share, but most in APAC and Latin America
- And you still need to support iPhone4

Textures

- Lack of uncorrelated 4 channel compression
    - Consider Roughness in a separate texture
    - Pairing Roughness with Specular/Metal instead of Albedo or Normals since former is low frequency & low variance data

Textures

- Lack of HDR compression
  - IBL, Lightmaps: RGB*2 instead of RGBm/HDR
  - IBL: uncompressed HDR cubemaps
  - An awful tradeoff :(