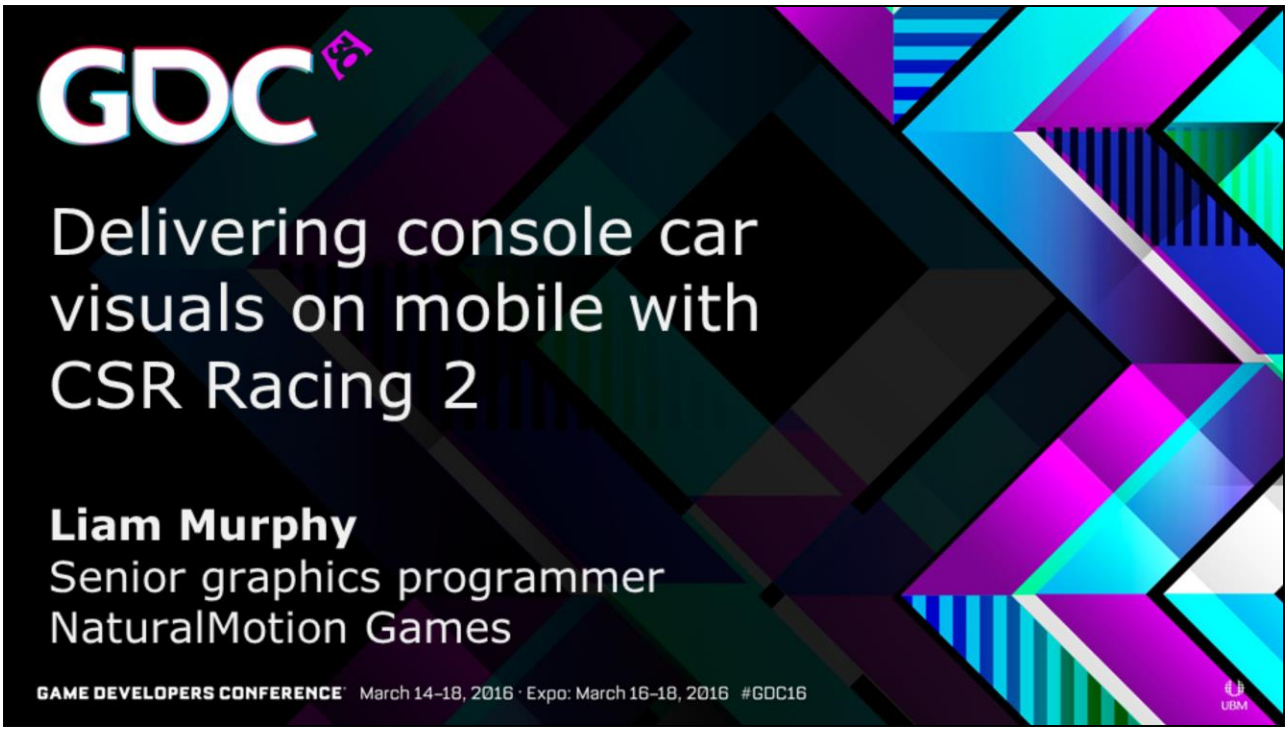


Good morning everybody.

I'm Liam Murphy, a senior graphics programmer at Natural Motion Games.

This talk, is one of two talks we're doing about our game CSR Racing 2, due to rollout worldwide later this year.

My colleague Scott Harber is doing his talk on the Environment Pipeline of CSR2 tomorrow at 2pm in room 2001, I recommend checking it out.

Today I'm going to tell you our story of how we overcame many hurdles, picked the right strategies and put console car visuals into our mobile game.

I'll show you how we've done this not just on new mobile devices, but old ones too, and not just across a few devices but thousands.

So lets get started.

CONTENTS

Here's a little preview of what I plan to be talking about.

I'll be starting with brief introduction.

Then cover a visual technical overview of the project.

Follow this with the evolution of our image based lighting solution.

And how we went about adding in the impression of motion.

Then finally wrapping up with our paint tech and some conclusions.

Oh, and the screenshot you see there, like almost all shown in this presentation is not a marketing shot but one captured in our game.

1. Introduction

I'll start with a bit about the history of CSR2.

CSR Racing

Here's an image from our predecessor, CSR Racing 1.

Weeks after it launched in June 2012, it soared to the top grossing app of 150 countries worldwide.

Since then it's been downloaded more than 150 million times with users averaging more than 800 races per second.

BUILDING CSR2

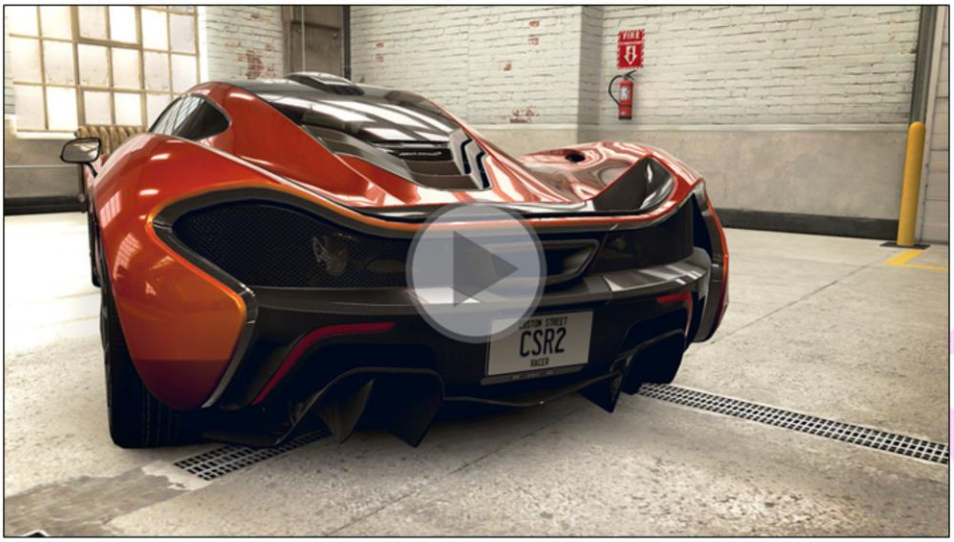
Following the huge success of CSR Racing, NaturalMotion grew a team with mobile and console backgrounds to create CSR2.

Like myself, many of us on the team have experience in developing AAA racing console titles.

Here's a photo of some of the people in our London Studio who have helped make CSR2 possible.

And now I'll show a quick teaser video of CSR2…

CSR Racing 2



[Teaser video plays of CSR Racing 2 … see https://www.youtube.com/watch?v=8G0RFAc5S_U]

As the trailer mentions, all the footage seen here was captured in-game on an iphone 6.

2. TECHNICAL OVERVIEW

I’m now going to cover a visual technical overview of CSR2.

PLANNED VISUAL LEAP FROM CSR1

- CSR1 & CSR2 using Unity

First off, like CSR1, CSR2 was built in variants of Unity 4.

PLANNED VISUAL LEAP FROM CSR1

- CSR1 & CSR2 using Unity
- BlinnPhong on CSR1

Some time after CSR Racing 1's entry to market, the CSR2 team started to create VT to aim for.

CSR Racing 1 was built using BlinnPhong shaders with an ambient and a single directional light – a good choice for the time.

PLANNED VISUAL LEAP FROM CSR1

- CSR1 & CSR2 using Unity
- BlinnPhong on CSR1
- New console visual targets & PBS

But bolstered with the confidence of big improvements to the average user's device performance, we made some ambitious console VT.

In order to deliver these we crafted a collection of physically based shaders fine tuned to the car team's requirements and adopted a top down approach for scalability.

KEY RENDERING CHOICES

- Forward rendering

Early on we had to make some key rendering choices.

We looked at number of different core rendering strategies that could match our gameplay and art requirements.

And due to memory and bandwidth limitations we opted to go with forward rendering.

KEY RENDERING CHOICES

- Forward rendering
- Hybrid static IBL

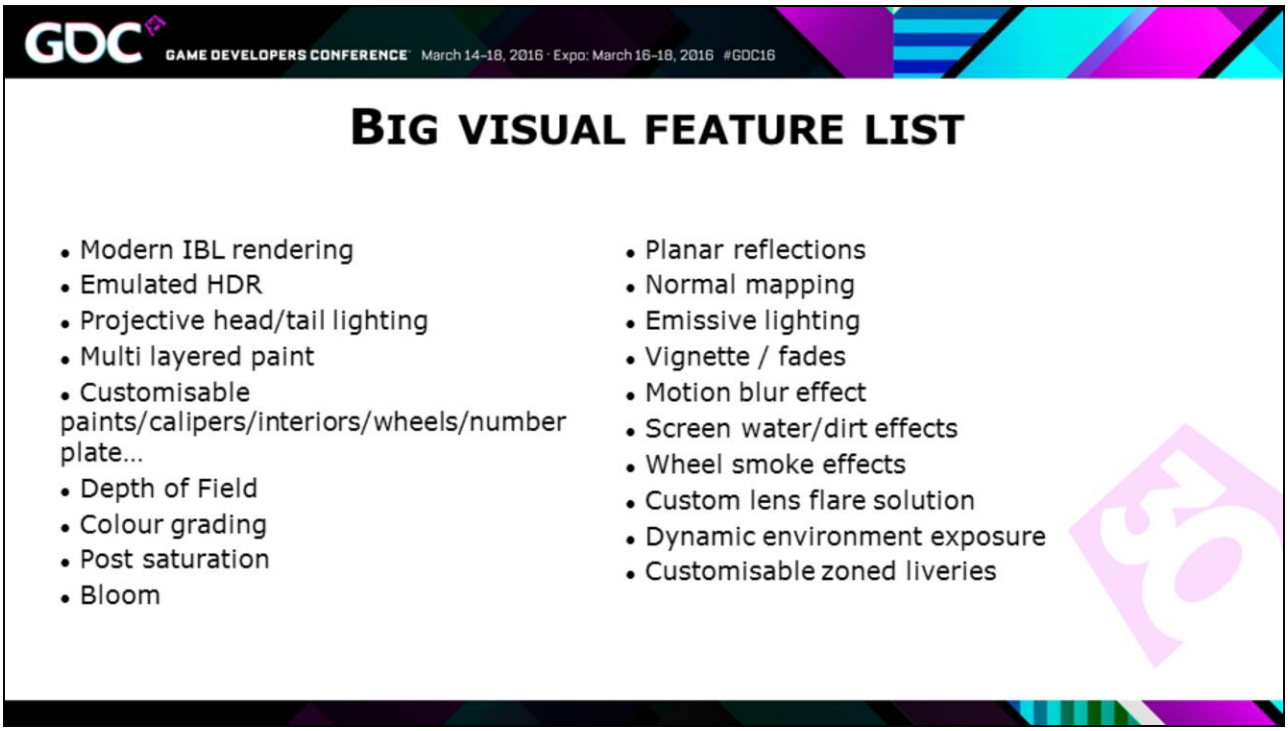
We tried out a number of lighting ideas, and thought thanks to the camera constraints in our game a hybrid static IBL setup could be a good fit.

KEY RENDERING CHOICES

- Forward rendering
- Hybrid static IBL
- Specular intensity

And initially our typical car shaders utilised an AlbedoOpacity texture and a SpecularColourRoughness texture.

But with specular colour frequently underutilised we decided to replace it with a single channel SpecularIntensity and added Ambient and Specular occlusion to the empty channels giving even more detail to our cars.



Here's a growing list of our key visual features.

Supporting 2500 Android and 19 iOS devices while having the rich tech list noted above needed to hit our VT has been challenging.

Primarily we've achieved this with our hybrid top down approach.

Traditionally titles aim their visuals at the bottom end device and bolt on visual improvements.

Like newer titles, the approach we've taken works by aiming visual targets at the top end device and then gradually stripping to core visuals for less performant devices.

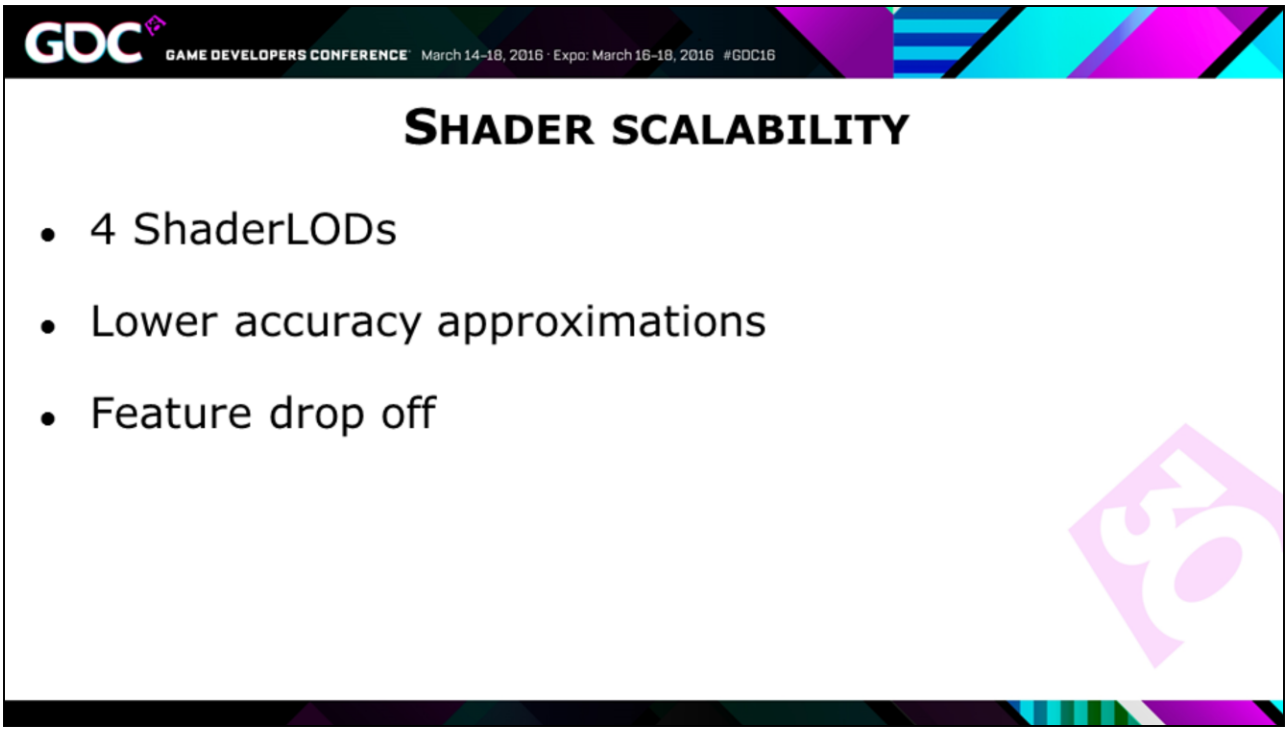
But the bottom end devices need special attention as they are particularly memory lacking and so we made super efficient assets for them.

SHADER SCALABILITY

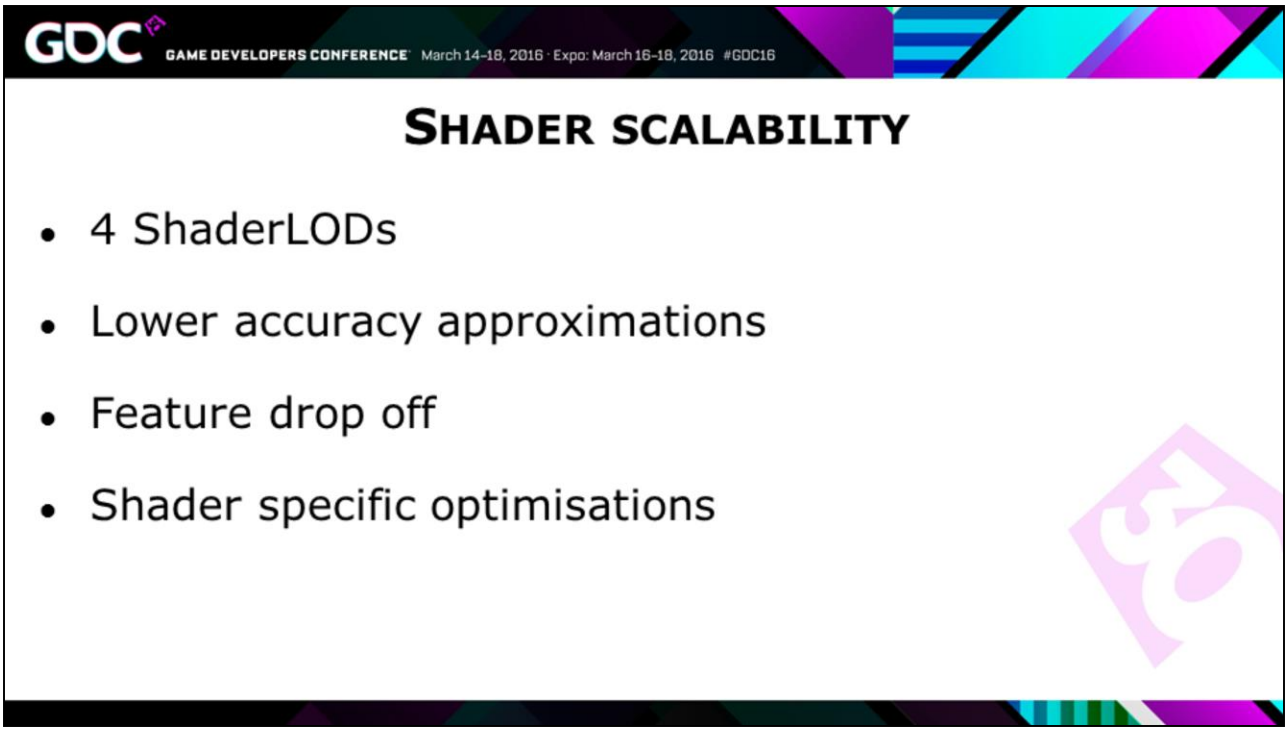
- 4 ShaderLODs

The performance gulf between the top and bottom devices was vast from the beginning, it gets even larger over time.

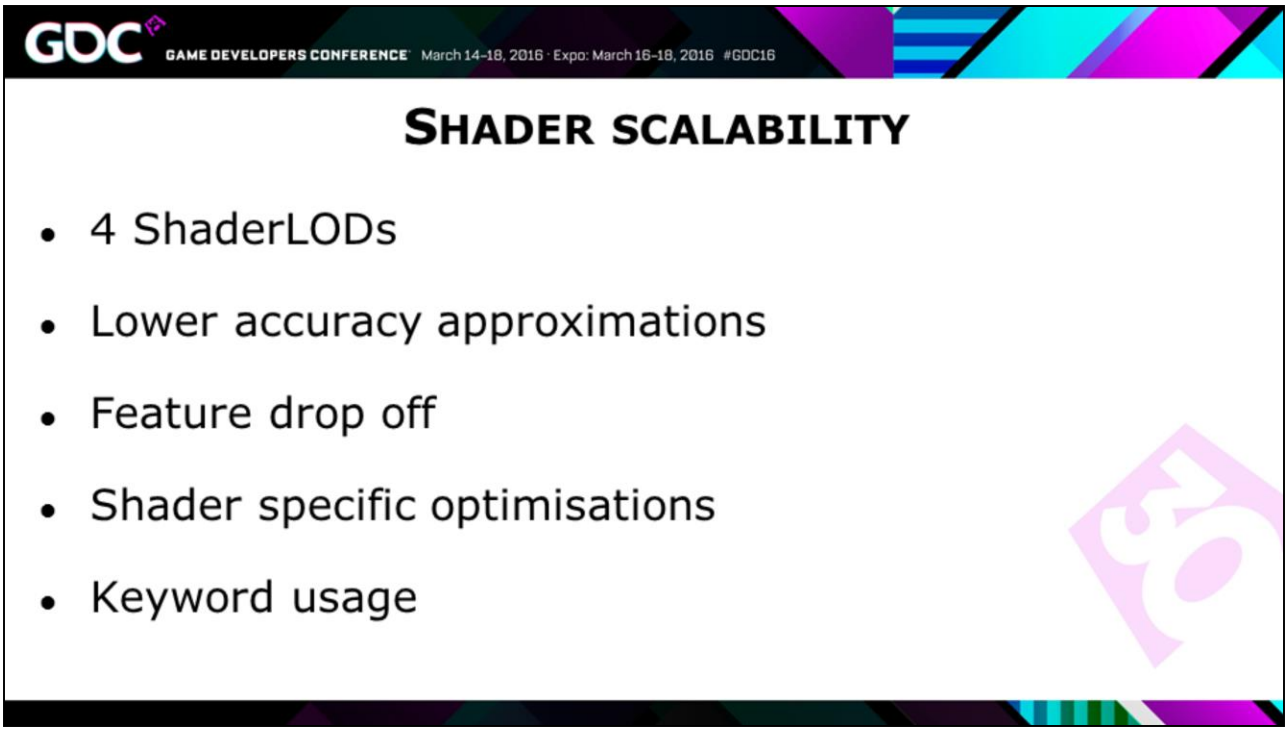
We addressed the scalability challenge of GPU performance with four different Shader LODs.



Each ShaderLOD adds in more approximations and phases out less noticeable visual features.



On the most expensive shaders, we created custom approximations for the bottom end ShaderLOD (such as the paint shader)



To reduce instruction waste, shader keywords were used, although we had to keep an eye on the increase in permutations as it created memory bloat.

I'm now going to show a demo of the results of our hard work on scalability…

DEVICE QUALITY LEVEL ACHIEVEMENTS

[Quality comparison video plays showing four different scenes using two different devices … see http://www.lfmurphy.com/public/gdc2016/4__3__techoverview__qualityachievements.mov]

We've engineered our game to play great across a huge number of devices spanning the iOS and Android platforms.

This demo shows the quality levels we've achieved, on the left a modern iPad Air 2 from 2014, on the right, many models older an iPad2 from 2011.

The biggest challenges come from keeping quality and performance up on older devices, a good portion of which our audience are still using.

We feel we achieved these goals with:

- Global savings across the board, AND
- Low end specific optimisations such as:
  - Geometry LOD,
  - Shader LOD,
  - And retaining only the visual effects to match the core of the visual targets.

Here are two charts showing average framerate readings from a mix of top and bottom devices.

We've taken readings from a variety of areas in the game, and as you can see we are close to maxing the framerates on not just the top end devices but the bottom end too, a big achievement given the quality levels on those devices.

We're still hard at work optimising so you can look forward to even better performance and quality in the future.

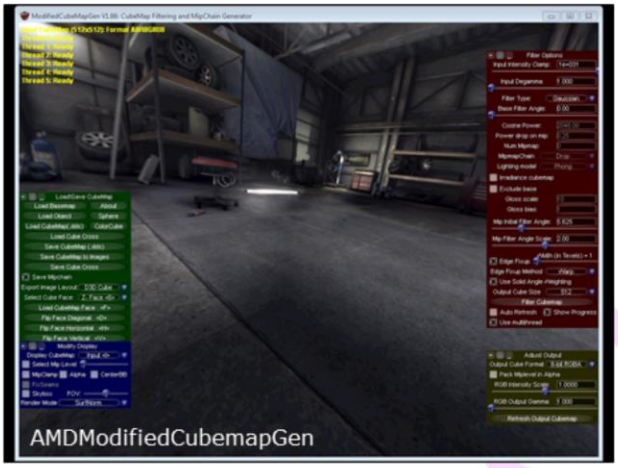
3. Evolution of IBL

As the last section showed, we've been able to get our game running and looking great.

The looking great part was mostly thanks to our IBL usage. I'll now cover the story of that.

PMREM

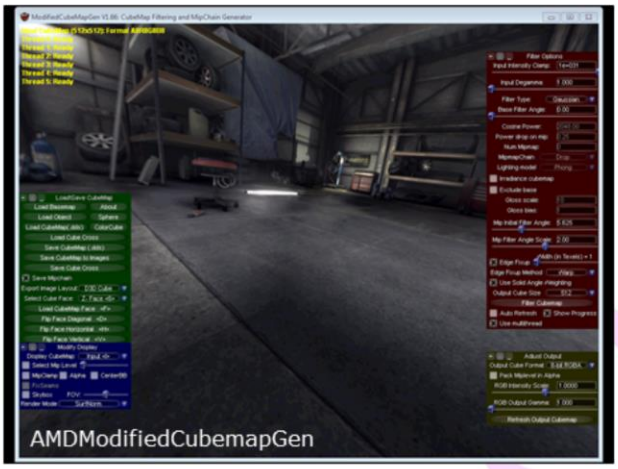
- Prefiltered Mipmapped Radiance Environment Maps



Very early on we experimented with PMREMs (Prefiltered Mipmapped Radiance Environment Maps) We were impressed with results.

PMREM

- Prefiltered Mipmapped Radiance Environment Maps
- Cubemaps from AMDModifiedCubemapGen



The maps are created by feeding a tool, such as AMDModifiedCubemapGen with a cubemap render.

The tool spits two cubemaps out:

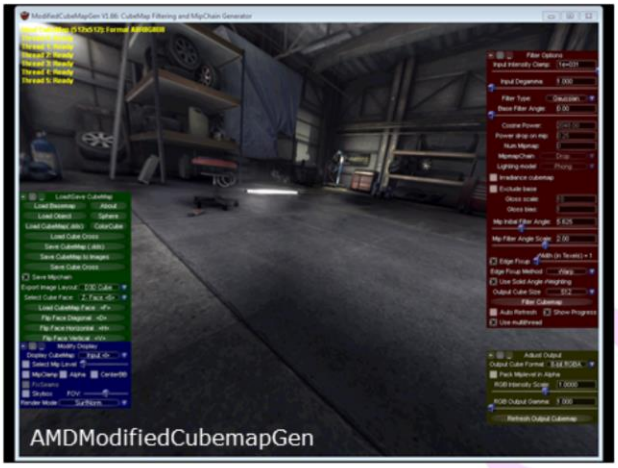
- A diffuse cubemap for indirect lighting.
- And a specular cubemap for direct lighting.

The specular cubemap hijacks the mipchain with progressively blurrier reflections.

Your shader code can then pick a mip through a read of a roughness map using the texCUBElod instruction to give realistic and diverse looking materials.

PMREM

- Prefiltered Mipmapped Radiance Environment Maps
- Cubemaps from AMDModifiedCubemapGen
- texCubeLOD via extension or GLES3+

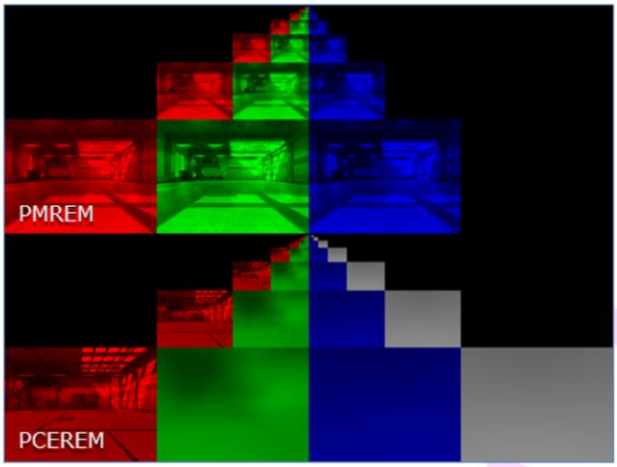


Blurry specular reflection relies on the texCUBElod instruction, only available with GLES3 or hardware with the correct extension support.

Too few target devices supported GLES3 at that time, we needed an alternative.

PCEREM

- Prefiltered Colour Encoded Radiance Environment Maps
- Store desaturated blurred reflections in colours
- Issues, but portable!



We had the idea to store desaturated specular blur reflections in the colour channels instead of the mip chain.

This approach works on GLES2, gives you back the mips to resolve distance sample aliasing, and there's a boost to the resolution in the higher blur levels.

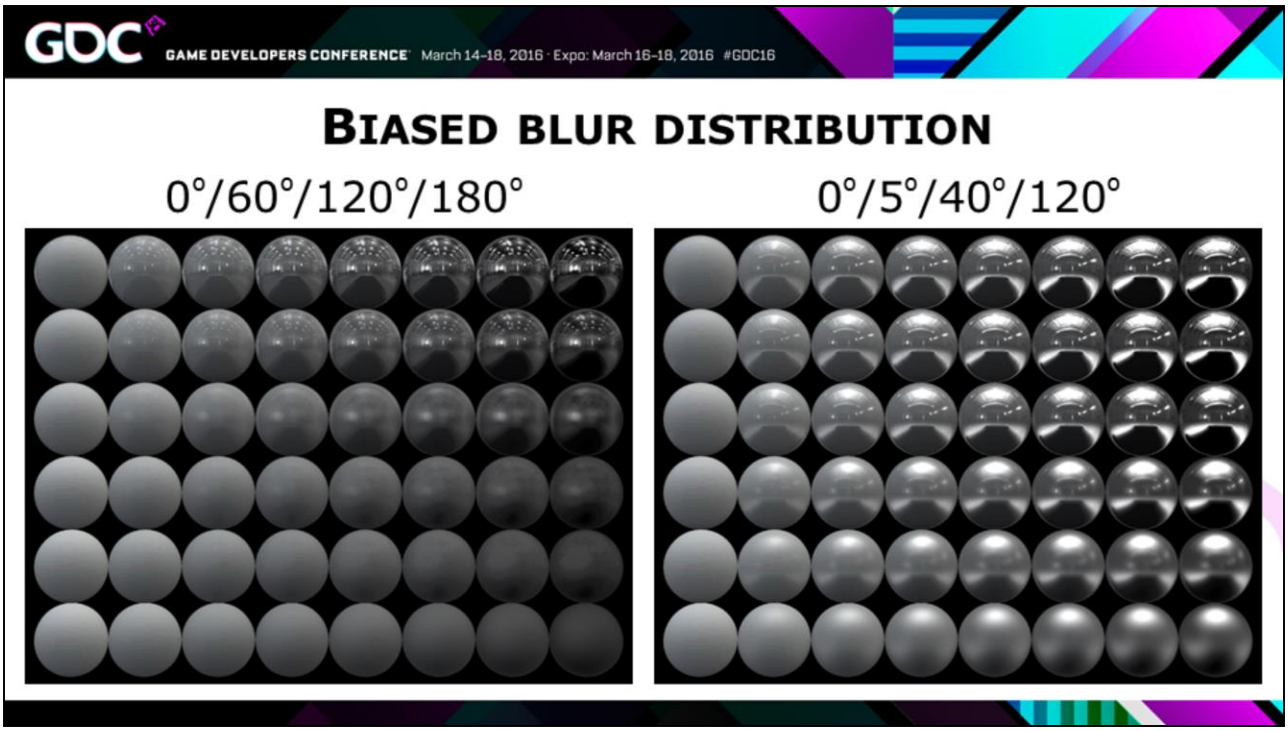
But it requires shader branching, has less blur levels, and there's a loss of specular colour although in practice it's not as noticeable as you think and you can author around it.

PCEREM AUTHORING

- 4 cubemaps for 1 PCEREM
- 1 cubemap for 1 PMREM

Building a full resolution PCEREM map took hours because it needed four large specular cubemaps with increasing blur level.

PMREM builds are much quicker as they only need 1 cubemap and the resolution drops with the blur level. This massively reduces the calculations required.



PCEREM started with equal 60º steps in blur level per channel.

We found that the transition between 0º and 60º was disproportionately discretised when compared to other levels.

Also, we found artists were mostly using the low end of the blur spectrum.

Knowing this we implemented a bias blur distribution of 0º/5º/40º/120º.

This choice dropped the 120º+ range but artists felt that was a worthy compromise.

HDR

- Paint and glass looked flat
- Limited support & high cost
- What to do?

With PBS and IBL active, the game was looking good but certain materials were looking flat.

We believed the primary reason for this was a our lack of high dynamic range.

Originally we looked into adding a full HDR solution, while feasible for the latest devices we had low end concerns such as limited support for floating point buffers, memory bandwidth and performance problems.

We needed something cheaper to achieve the same effect.

PCEREM WITH EMULATED HDR?

- HR from LR initially, HR rig later on
- Drive optimisations with visual target / prototype

PCEREM emulated HDR prototype (paint only)

Before we could prototype emulated HDR, we needed high range and low range assets.

Initially we generated the high range from low range in Photoshop.

Later on we created high range rigs for each environment.

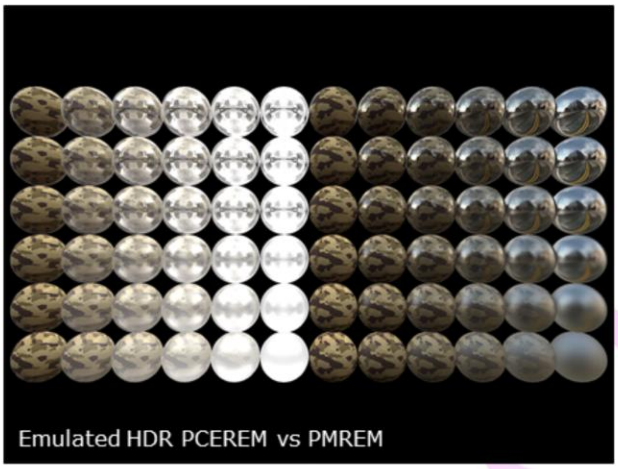
With the high range and low range cubemaps made, we started investigating ideas for efficient HDR emulation.

We found ourselves going round in circles trying to pre-optimise without a specification or prototype.

So instead we worked with the artists and put together a prototype that met their requirements and then optimised the solution.

END OF THE ROAD FOR PCEREM?

- Prototype great, but too expensive
- Need high resolution
- Bit encoding too much of a visual compromise



Emulated HDR PCEREM vs PMREM

Our HDR emulation prototype gave us impressive results.

But the setup needed optimising though as it used 2 large cubemaps, 2 tex reads and the HR cubemap was particularly wasteful as we only needed one channel.

We considered reducing the cubemap resolution, but the quality drop was unacceptable and that still left us with 2 tex reads.

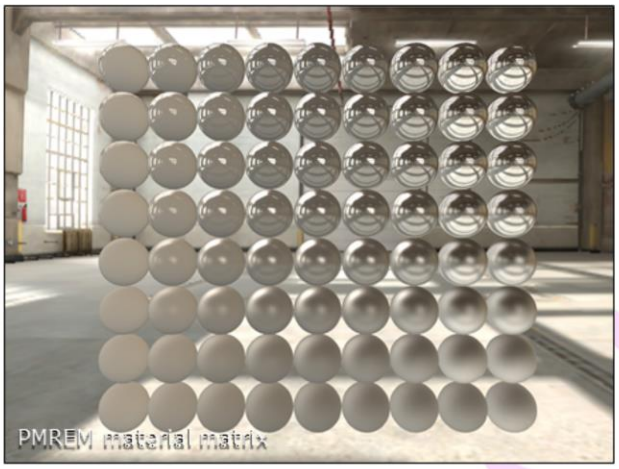
We experimented with bit encoding HR and LR into the same cubemap.

But any distribution resulted in banding artefacts in one of the ranges, particularly in high blur channels where gradients were common place.

Dithering wasn't helpful either as it introduced obvious grainy artefacts.

BACK TO PMREM!

- texCubeLOD support on all target iOS devices



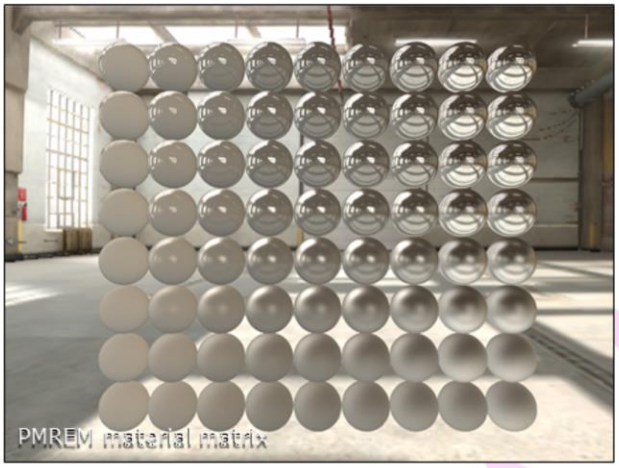
With HDR like effects essential to hit VT, and PCEREM looking like a dead end, we needed alternatives.

Knowing PMREM could in theory be great, we did a deeper investigation in the hope we'd been too hasty to abandon it.

Sure enough we discovered that although our bottom iOS devices don't support GLES3+, they do have extension support for texCubeLOD!

BACK TO PMREM!

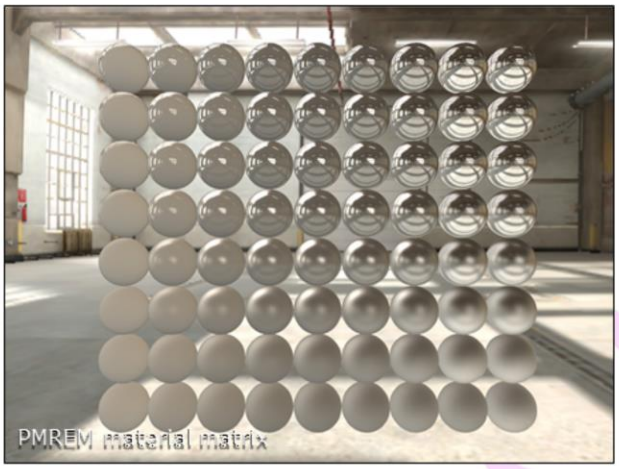
- texCubeLOD support on all target iOS devices
- texCubeLOD emulation viable



We also uncovered evidence of a number of developers using texCubeLOD emulation methods in the wild, potentially solving our issues on bottom end Android.

BACK TO PMREM!

- texCubeLOD support on all target iOS devices
- texCubeLOD emulation viable
- HR & LR fits in 1 cubemap



This was fantastic news as it would allow us to collapse both cubemaps into one by storing the HR cubemap into the LR cubemap's alpha channel.

As a bonus, it uses no branching, has more blur levels, quicker build times, and even put colour back into our specular reflection.

PMREM WITH HR IN UNITY

- LR/HR combination

After committing to move to PMREM we looked at what the next step was.

We started by combining the LR and HR cubemaps into one cubemap, initially in Photoshop later in script.

With a combined LRHR asset made, we then encountered three stumbling blocks to getting PMREM into the game…

PMREM WITH HR IN UNITY

- LR/HR combination
- Mips overwriten on import

The first:

- Was we found Unity overwrites the PMREM mipchain on import!
- We overcame this with a custom import script.

PMREM WITH HR IN UNITY

- LR/HR combination
- Mips overwriten on import
- Distance sample aliasing

The second hurdle:

- Was that we found PMREM's hijacking of the mipchain for blur levels introduces distance sample aliasing.
- We mitigated this with a well known trick: we encoded the mip level into the diffuse cubemaps alpha and then biased reads of roughness upwards based on reads of that value.

PMREM WITH HR IN UNITY

- LR/HR combination
- Mips overwriten on import
- Distance sample aliasing
- Resolution disparities

The third hurdle encountered

- Was that we have different resolutions for diffuse and specular cubemaps and so mip level reads on diffuse do not translate to the specular properly.
- As a work around we added an offset which achieved roughly the desired effect.

PMREM WITH HR IN UNITY

- LR/HR combination
- Mips overwriten on import
- Distance sample aliasing
- Resolution disparities
- Linear response

As an extra tip:

- We reconfigured our specular cubemaps to grow linearly and end at 180º allowing for intuitive roughness map authoring.

TEXCUBELOD EMULATION

- All target iOS devices had texCubeLOD support
- Number of target Android devices with GLES2 and no or questionable support

By now PMREM was working nicely on all our target iOS devices and almost all our Android devices.

But we needed a working solution for our bottom end Android devices – without it the game looked as above.

Many of the bottom end Android devices only had GLES2 support but some have texCubeLOD extension.

Unfortunately we found runtime polling for texCubeLOD support not an option, but we did have the option to distribute different builds per device on Android.

So knowing this we created texCubeLOD emulation builds for devices with no or questionable texCubeLOD support.
I say questionable support because some devices such as the Samsung S3 have different chipsets depending on region, some with texCubeLOD support others without.

TEXCUBEBIAS

- Requires trilinear filtering

We experimented with texCubeLOD emulation strategies.

A method from an old AMD presentation using texCubeBias looked the most promising.

The results give us perceptibly identical visuals at the cost of some memory and extra GPU instructions.

But I wont cover the implementation here, I have a good link in the references for that.

Instead I'll mention two interesting points:

Firstly, the algorithm falls apart without trilinear filtering:

- By default Unity sets cubemaps to use bilinear filtering and there's no UI to change this.
- But fortunately you can override at script level as we did.

TEXCUBEBIAS

- Requires trilinear filtering
- Mip level calculation must be accurate

The second issue encountered was that the mip level calculation must be accurate:

- Earlier I mentioned a trade-off we make to keep the diffuse cubemap memory usage down by using an offset trick to approximate miplevel.
- This approximation isn't good enough for texCubeLOD emulation, and so we matched the cubemap resolutions on those builds.

4. Motion IBL

So as you can see, static IBL worked out great for us.

But we needed something else for when the cars are in motion. Here's what we did…

MOTION PRMEM

- Race looked odd with static reflections
- Desire to keep support for rough materials
- Illusion of motion good enough?

After our first stage introductions of PBS and PCEREM, our attention turned to the race visuals.

As expected with static IBL, the race looked odd, with cars accelerating through the environment with lighting and reflections staying still.

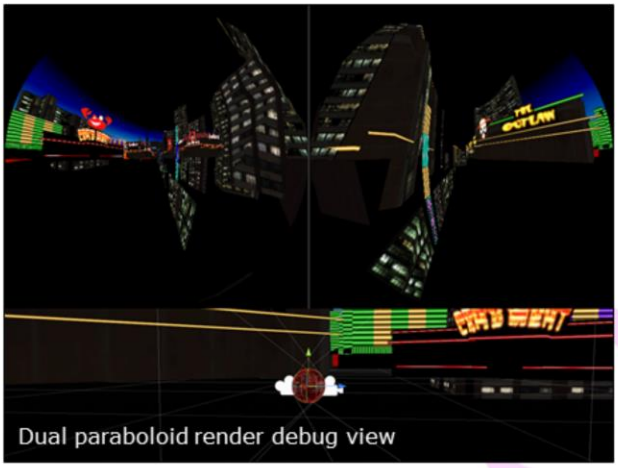
We considered switching to a real-time reflection solution, but there was a strong desire to preserve rough material support allowing pearlescent and gunmetal paints.

We thought that such a solution might work on top end devices but would be too demanding of the lower ones.

Perhaps we could create a hybrid of the existing setup based on the illusion of motion?

How did CSR1 do it?

- Top spec: Dual paraboloid reflections



Dual paraboloid render debug view

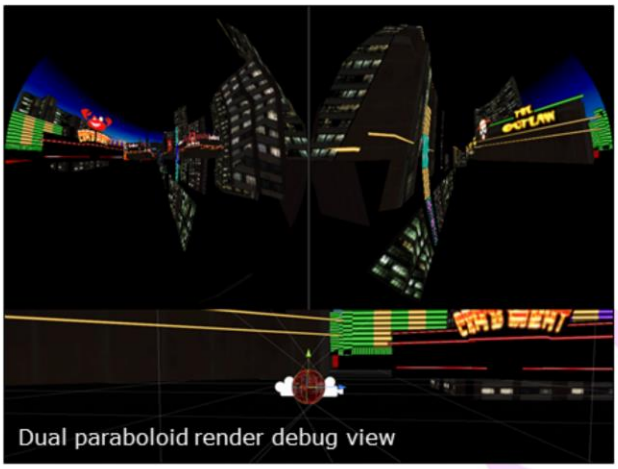
Before delving into our solution, we thought we would see what CSR Racing 1 did.

On the top spec, CSR1 used a technique known as dual paraboloid rendering:

- This technique renders a low LOD version of the environment onto two planes warped into an overlapping hemisphere rig, achieving a form of real time reflections.
- This is performant with good results, but we wanted to keep rough material support, and felt it too expensive to convolve blur levels on the fly.

How did CSR1 do it?

- Top spec: Dual paraboloid reflections
- Rest: Specular tumble + horizontal texture projection



Dual paraboloid render debug view

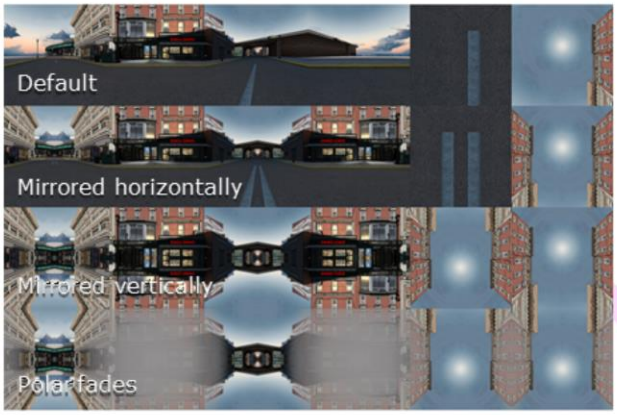
On remaining specs, CSR1 exploits the drag racing constraints of the game and uses a two stage effect:

- It rotates a specular cubemap about the Z axis based on the car's speed.
- It adds a horizontal scrolling projected texture on the side of the car to eliminate polar reflection artefacts.

We thought the tumbling specular cubemap would be a good starting place, but the scrolling texture could be difficult to integrate.

TUMBLING SPECULAR REFLECTIONS

- Mirror XY for missing assets



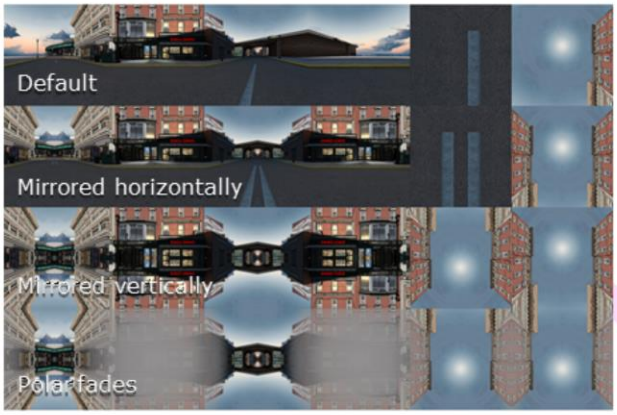
Early on, cars only moved in the race environment, so we targeted the technique to race only.

The first problem we encountered:

- Was that our environments had only been authored in the direction of the camera, but race reflections needed both sides.
- Initially we mirrored into the missing reflections as a temporary workaround (seen in the image above)
- Later on artists created special areas for cubemap baking.

TUMBLING SPECULAR REFLECTIONS

- Mirror XY for missing assets
- Mirror ZX for reflection artefacts

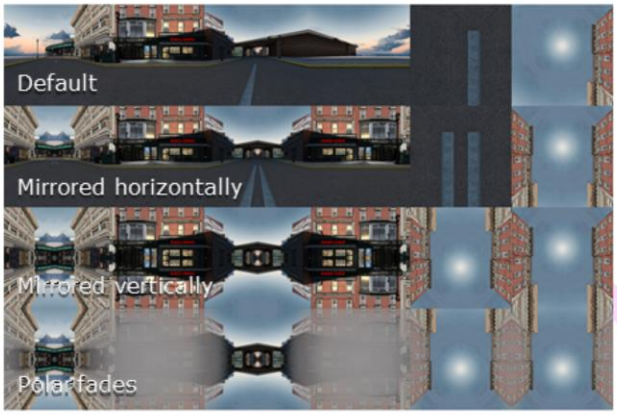


The second problem we found:

- Was there was an unsightly strobe effect in the reflections, as the floor would reflect on the roof of the car.
- We countered this by mirroring the top hemisphere downwards.
- This created an up-lighting artefact but was a good stop gap solution.

TUMBLING SPECULAR REFLECTIONS

- Mirror XY for missing assets
- Mirror ZX for reflection artefacts
- Fade at poles for swirl artefacts



Finally we found that:

- Swirling artefacts would occur around the doors, along the axis of rotation.
- CSR1 countered this by adding in a texture projection but we avoided that.
- Instead we baked a fade around the poles in the LR/HR cubemaps and found this worked nicely.

BLEND FROM STATIC

- Start line visuals matter
- Full res static specular
- Half res tumble specular

The tumbling specular solution lasted for some time, but with the quality bar up a few notches, we soon needed to improve on our solution.

We noticed that there was a lot of player attention on the start line visuals.

These get most scrutiny because of close up cut scenes and start line idling.

We thought a quick win would be to introduce a new start line specular, and blend into the tumble specular map.

We opted for this approach as:

- We were able to half the resolution of the tumble specular, as detail when the car is in motion is less noticeable.
- But also because we'd made memory savings.

LOWER HEMISPHERE MASK

- Mask out up-lighting
- Tumble on up vectors
- Static on down vectors

The static blend technique was a good step in quality.

But motion reflections were still left with up-lighting artefacts.

We addressed this by adding in a bias blend:

- Where the shader would prefer tumble specular on upwards vectors.
- And would prefer static specular on downwards vectors.

PARALLAX CORRECTED PMREM

- Some setup work
- Best with "boxy" environments

With the previous changes, artefacts in race reflections had been heavily cleaned up.

We then wanted to try something a bit more ambitious now we'd banked a good fallback solution.

We found examples on the Internet of Parallax Corrected PMREM.

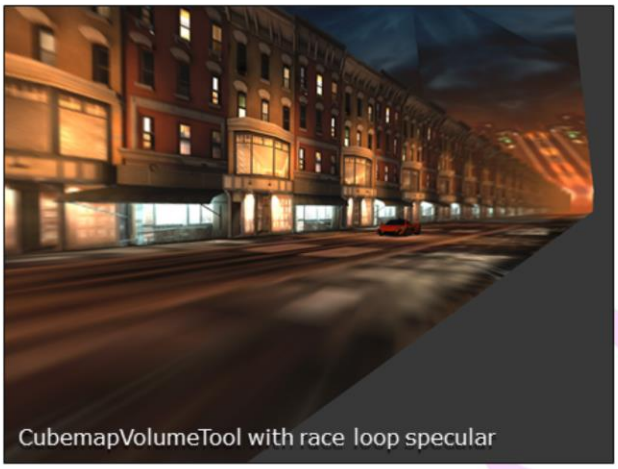
The tech works by bending reflection vectors to give the impression of a render position.

The demos looked great, but we realised we would need a lot of up front work to get the tech into a previewable state.

We also learnt early on that the effect works best with environments that are boxed shaped.

CubemapVolumeTool

- Tool to help fit AABB to scene
- Defined constants for Parallax Correction
- Useful for debugging PMREM



We started the prototype by making a tool to project cubemaps onto an AABB.

This would make fitting that box to the scene easy and effective.

The tool ended up doing other things:

- Such as populating shader constants.
- Allowing portal and looping positions.
- And later on, Mip and HR visualisations were added to make debugging cubemaps super easy.

I'm now going to demo the CubemapVolumeTool…

CUBEMAPVOLUMETOOL IN ACTION

[Demo plays showing how the tool helps achieve PCSR … see http://www.lfmurphy.com/public/gdc2016/16_9__motionibl__cubemapvolumetool.mp4]

In the demo, a CubemapVolumeTool's AABB has been fitted to one of our scenes – in this instance, the garage.

I toggle between the debug mode, to simple cubemap view mode and finally to the cubemap projection mode.

The projection looks unusual here because the cubemap rendering position doesn't match AABB centre. Here I use offset correction to fix this. You can see flying around the projection looks pretty convincing.

The hide toggle displays the similarity to the scene render. With a toggle we can slide through the mip blurry levels of the cubemap. The same applies for the alpha HR channel. Both can be used in conjunction.

Next, I hide the CubemapVolumeTool and move some spheres around the scene which make use of the values from the CubemapVolumeTool to apply parallax corrected specular reflection.

FEEDING THE SHADERS

All of our signed off cubemap renders were not made at the AABB centres to begin with, so we made strong use of offset correction.

Offset correction results in subpar texel distribution, and a minor hit to performance, but it gave us a great bridge until we could author cubemaps at the correct positions.

With the tool done, we wrote test shaders as seen in the sphere demo and then ported the support into the car shaders.

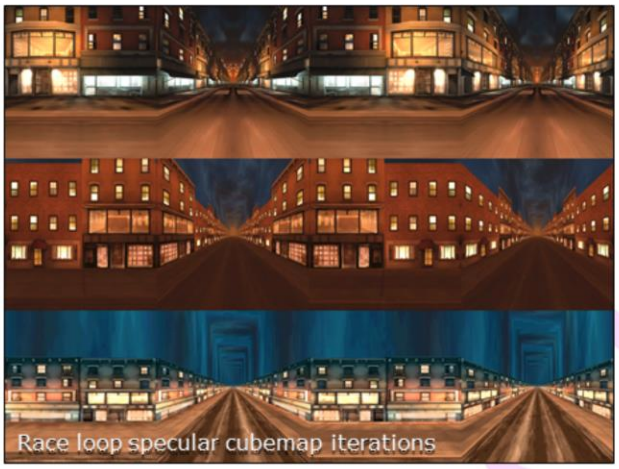
Originally we planned to roll the tech out to all environments but eventually we opted to keep it just for race.

While this may change in the future, we did this mainly because the effect is costly…

As it'd be difficult to find the budget to have the effect on low end devices while inside a full garage.

AUTHORING LOOP SPECULAR

- Bake tunnel to cubemap

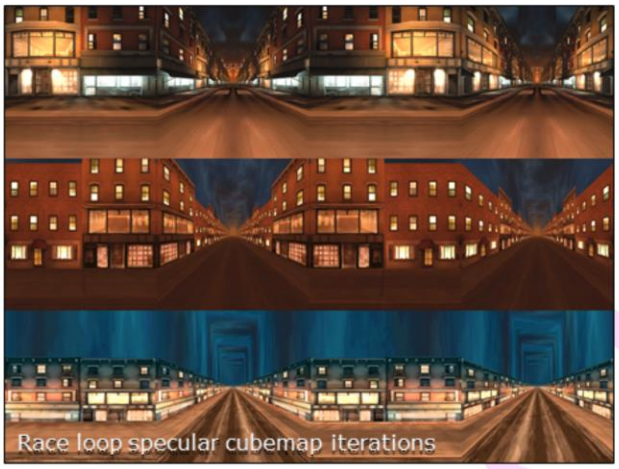


With parallax correction working in standard scenes, we moved to prototype an idea we'd originally had early on, called loop specular reflection.
The idea behind this is to create a seemingly infinitely long tunnel with a repeating pattern on, and render it to a cubemap.

We'd then fit that cubemap to a CubemapVolumeTool and have our cars pick reflections as if they were moving through a portaled section of that environment.
Portaling is necessary because texel density within the reflections decreases the further you move from the original render position.

AUTHORING LOOP SPECULAR

- Bake tunnel to cubemap
- Balance loop distance



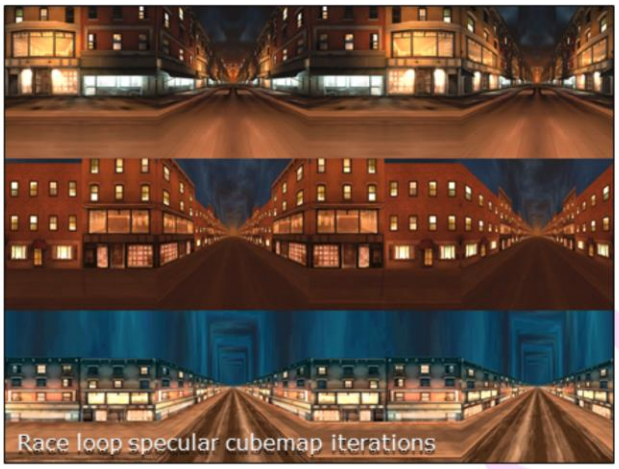
After getting the tech working we had a steep learning curve getting the artwork right.

It's important to balance portal loop distances:

- If they're too close you get overly repetitive reflections.
- Too far and you get resolution strobe artefacts.

AUTHORING LOOP SPECULAR

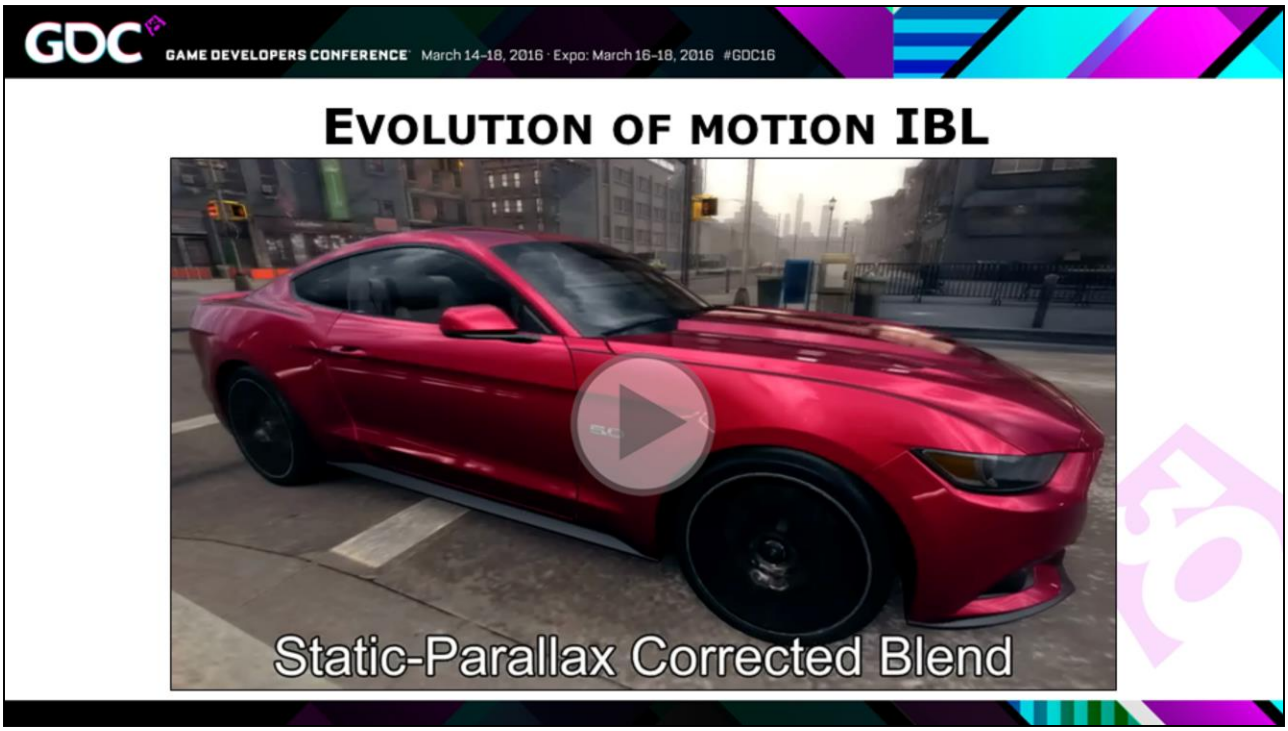
- Bake tunnel to cubemap
- Balance loop distance
- Protrusions break the effect



Also, orthographic environments look best:

- Environment protrusions like chairs, tables and awnings create loop artefacts.
- And interestingly, in the above image it looks like progress is bottom up, but actually its top down.

I will now demo the evolution of our motion IBL…



[Evolution of motion IBL plays showing all previous quality steps one after another … see http://www.lfmurphy.com/public/gdc2016/16_9__motionibl__racereflections.mov]

We start with the initial static specular reflection, only spinning wheels and shadow cues help with perception of motion here.

We then move to tumble specular reflection, this helps a lot but there are a number of visual issues left.

Next we add in a static blend which improves the start-line visuals significantly.

And then add in a mask to help eliminate uplighting artefacts when the car is in motion.

Finally we replace the reflection tech with our static blend to loop specular version, a big step forward.

Here we toggle the CubemapVolumeTool and you see what's actually being reflected, and now we hide it.

5. PAINT

As you've seen, our IBL and motion IBL were essential in getting us to console car visuals.

The other big thing though, was achieving realistic looking car paints. Here's a little insight into that…

Early paint & Emulated HDR

- Early paint similar to other materials
- Emulated HDR helped deliver glassy sheen appearance of clear coat

Our early paint shader had a material diffuse tint, f0 and shininess.

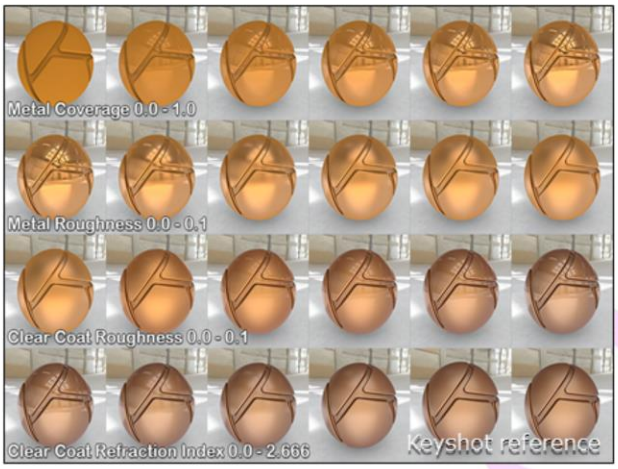
The shader started off very similar to other car shaders.

The introduction of emulated HDR covered earlier added big realism to our car paints.

It helped do this by allowing bright lights to ”punch through” a low fresnel multiplier at facing angles delivering that glassy sheen appearance we’d been looking for.

APPROXIMATING PAINT

- Used offline renderers & reference images
- Approximated rather than simulated



Simulating all the light interacting complexities of multi-layered car paint is far too expensive on mobile so approximations are required.

One alternative idea is to model the primary responses seen with combination of reference footage and offline renderers (for example we used Keyshot)

We tried to model the energy conservation of real paint by balancing contributions across three layers, those are: base coat, clear-coat and metallic coat.

In reality the metallic coat is composed of countless randomly orientated reflective flecks.

We simulated this effect using a metallic tint contribution mimicking the observation that:

- The tint contribution varies according to the view angle and intensity of the reflection.

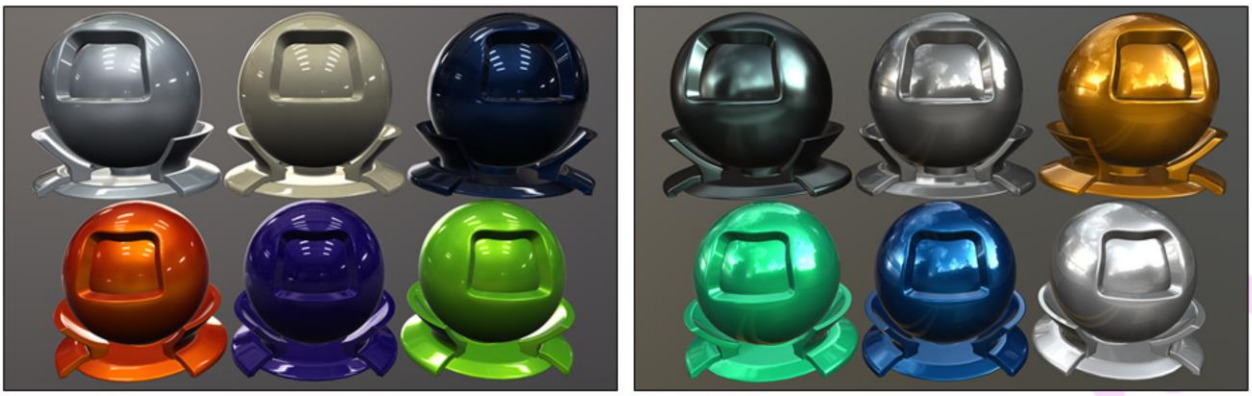
INPUTS

- diffuseTint
- metallicTint
- metallicShininess
- metallicCoverage
- clearcoatF0
- clearcoatShininess
- metallicBoost
- fresnelPower

After many iterations of improvements, our car paint shader ended up with quite some complexity to it.

Case in point in the number of inputs that go into each and every paint.

RESULTS



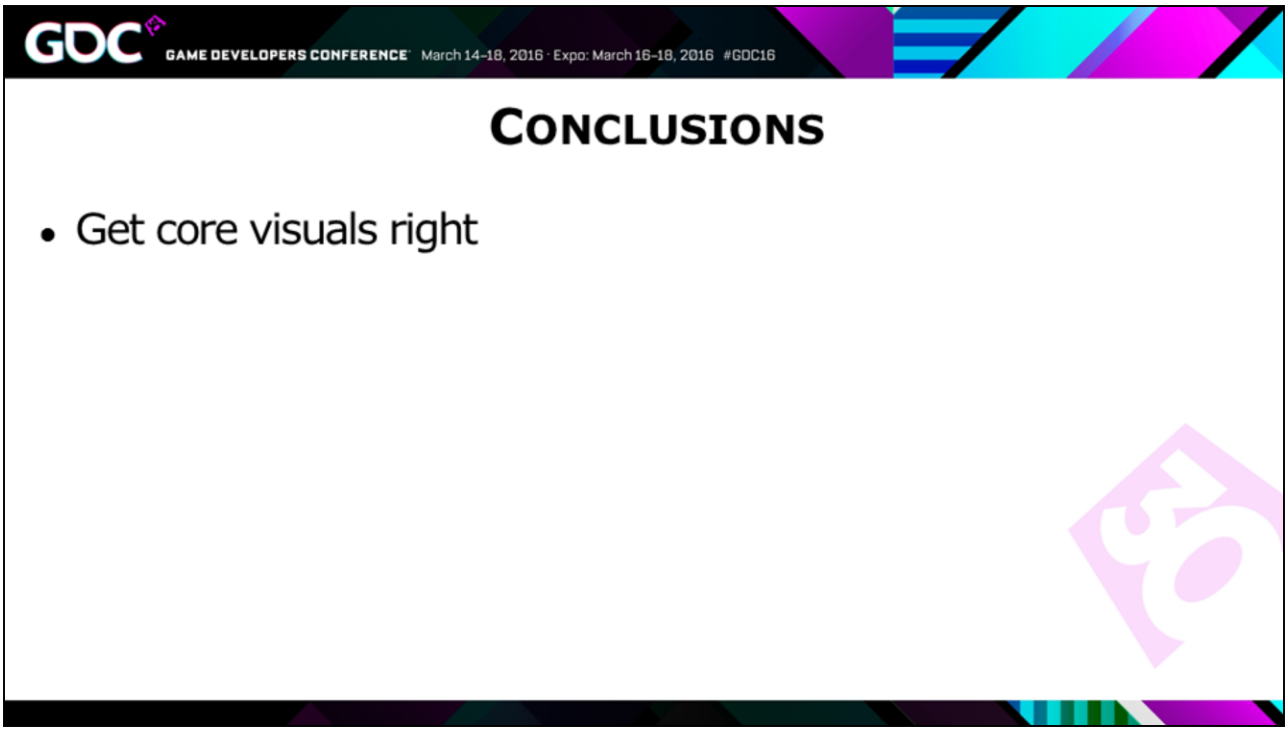
Here's a very small selection demonstrating the variety of the paints we use in the game.

It's been very important to us to get our paints to look great.

While our game supports a range of aftermarket paints we designed, all our cars feature real world manufacturer paints that have been painstakingly matched by a professional hired specifically for the job.

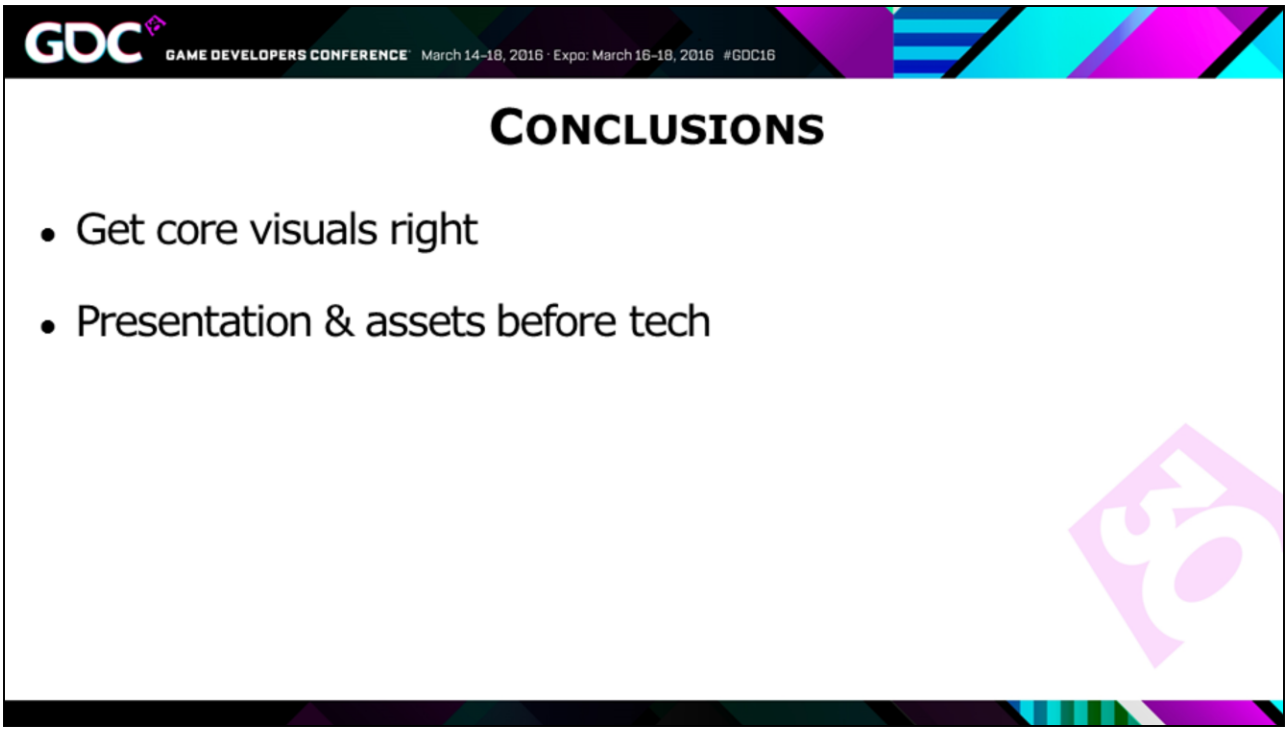
6. Conclusions

I will now wrap up!



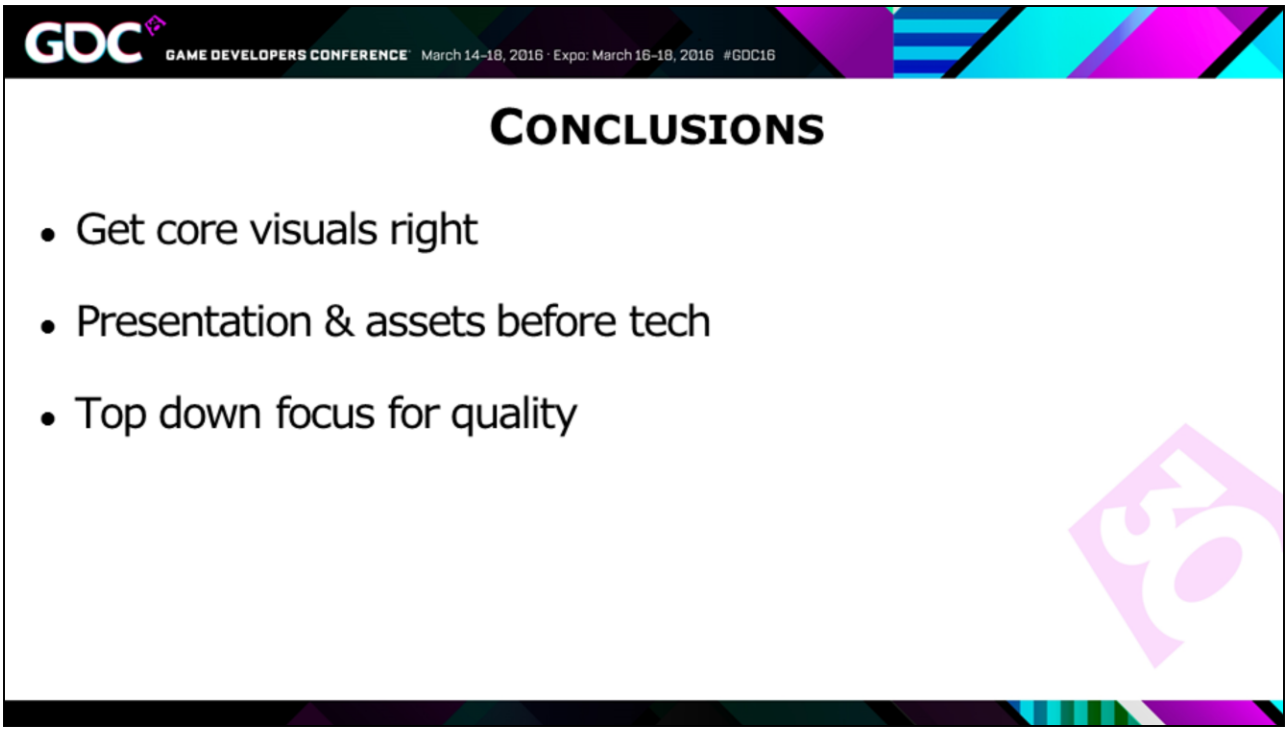
Invest in and agree visual targets early.

- It's wrong to pass up quick VFX wins.
- BUT, the bulk of your time should be spent on building and polishing key visual features that align with your visual target.
- And get your prototypes looking right first and then approximate, optimise and fine tune.



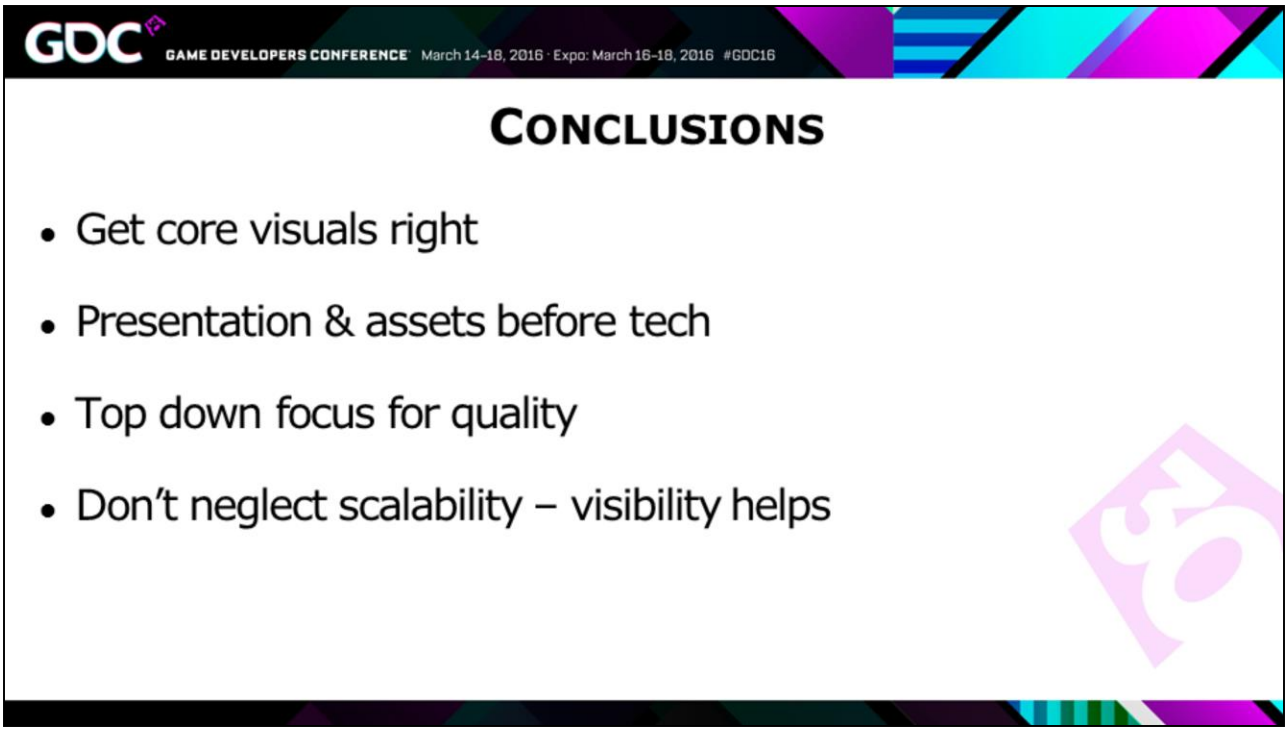
Powerful tech can really help push realism.

- However, I believe it's far more important to ensure you have a base of top notch assets and then ensure they're presented in a realistic and interesting way.
- With well authored environments, post and cameras you are less reliant on tech and so can scale back on lower end devices while still looking great.



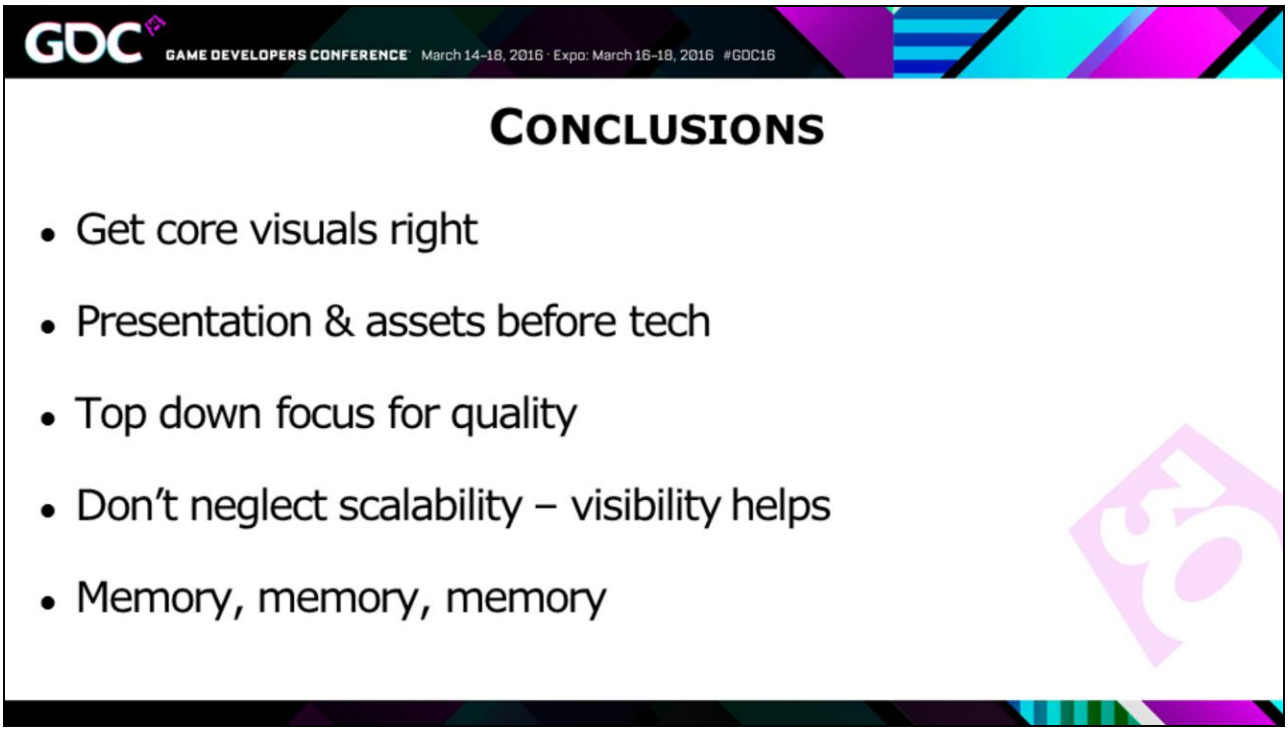
A top down focus is great for quality:

- Because all visual features have a clear purpose towards forming the VT.
- And because it future proofs the game, making it easier to shift the VT upwards to cater for new devices coming to market.
- Also, more users will hit the original VT overtime as the average user's device performance will increase.



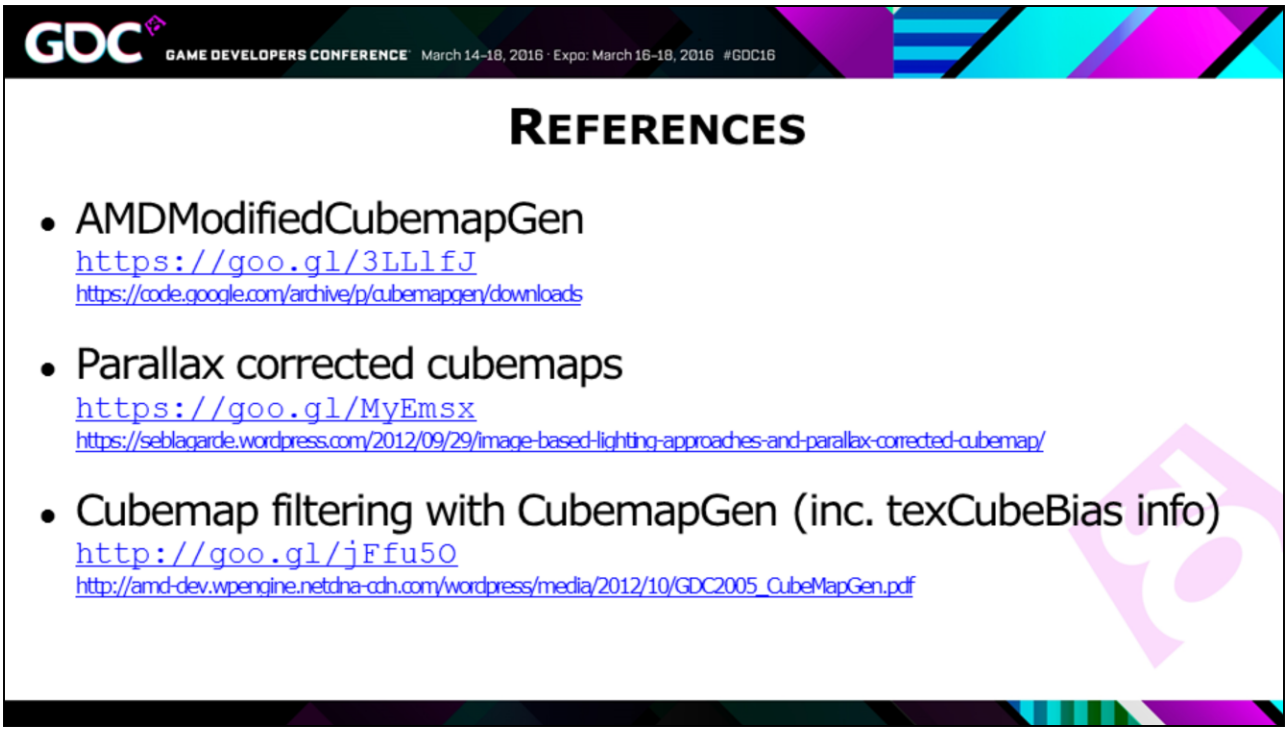
BUT, be careful not to neglect scalability.

- Avoid having too many visual features that must exist all the way down to the bottom spec.
- And avoid having decision makers only playing the game exclusively on top end devices.
- Low end devices are not just for QA and your graphics team.



Memory and memory bandwidth are real pains particularly on low end devices:

- Keep low end devices stable:
  - Doing this keeps devices reviewable and thus visible to decision makers.
  - Plus it's painful to have to try to make big memory savings late in development.
- Have a widespread LOD strategy.
- And if you can, avoid pushing the problem onto another area, such as load times and quality, or at least be aware you're doing it.



Here's some good references relating to a few of the things I've spoke about today.

Please see the published slides or see me after for details on these.

GOOD NEWS, WE'RE HIRING!

Thanks for listening!