SIGGRAPH2015

Xroads of Discovery

Technology Driving Angry Birds:Go!

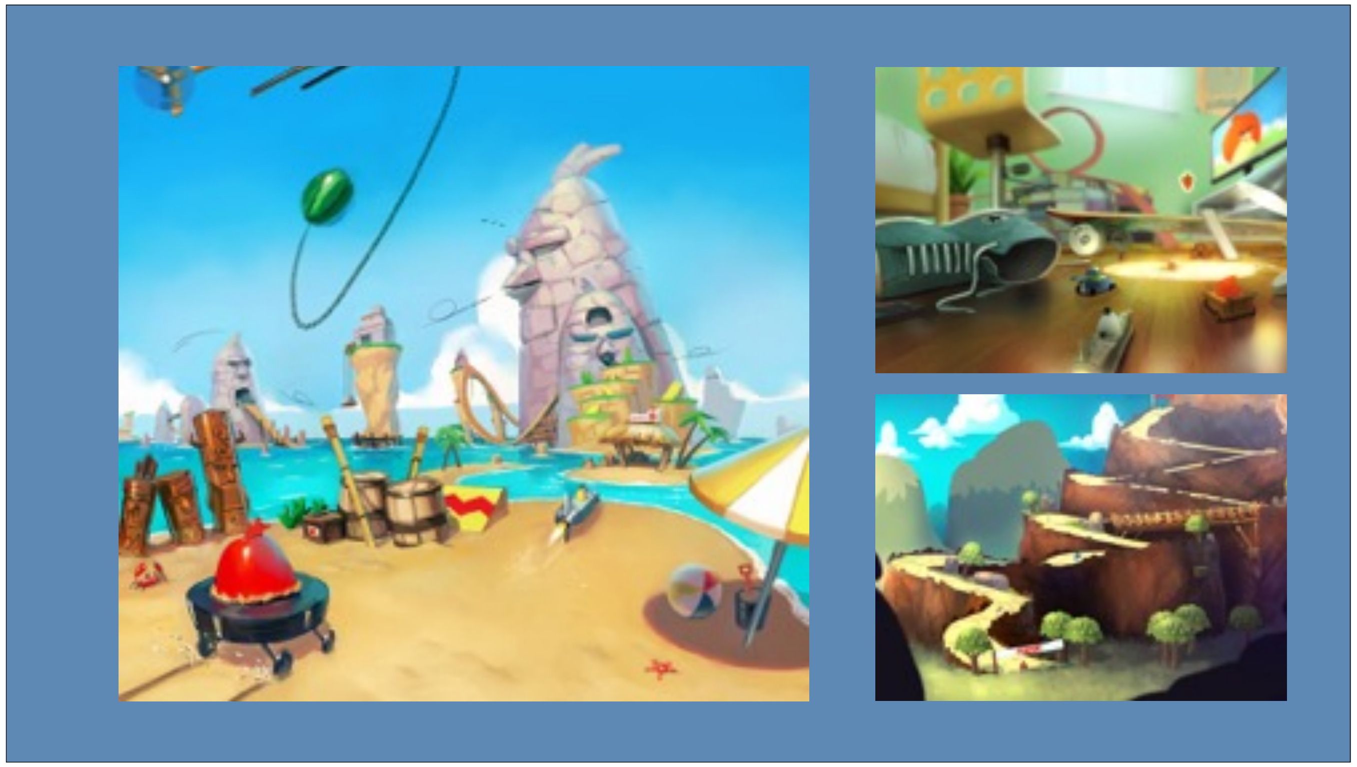
Simon Benge
Exient Ltd

Who am I?
Supervising technical artist at Exient Ltd and part of the development team that worked on Angry Birds:Go!
Exient are a game developer with a wealth experience on handheld and console games spanning some 15 years.
Based in Oxford in the UK, we have three studios, two in mainland UK and one in the island of Malta, each focused on a different genre of game.

We typically take on projects that I would consider to be the more technically challenging game titles.

Whats My role?
To make art production efficient for the art team.
To lubricate the process of getting assets from the art department into the product.
and as a shader artist, to balance visual quality with product performance.
….Thats the job spec, but in reality my job is about gaining trust from those I work with. Artists need to trust that I will handle their vision with care, and programmers need to trust that I won't do anything too performance crazy that will make their lives a misery!

What is this talk about?
During this presentation I'm going to share some of the challenges the technical team encountered developing Angry Birds Go. Specifically I'm going to talk about the environment and render optimisation techniques we employed and how we were able to achieve them through good teamwork and a simple approach.
There should be something for everyone at the very least you'll have a good understanding of what it takes to be a mobile game developer.
Game development can get a bit…. McGuyver ….sometimes, so please don't judge us too harshly !



What is Angry Birds Go ?

Angry Birds Go is a free to play mobile app featuring the Angry Birds and their mortal enemies, the pigs. It puts them together to compete in a downhill kart racing game.

To date, Go has been downloaded 140 million times and typically has around 1.5 million players a day.

History

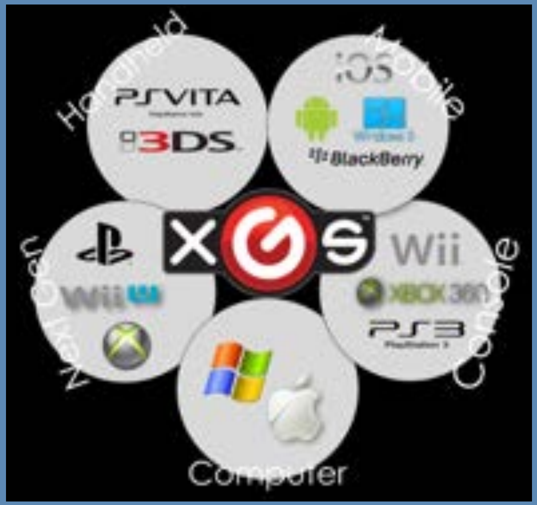
In the spring of 2013, having completed a port of Angry Birds trilogy to nintendo 3DS, we approached Rovio Entertainment with the idea of combining our existing racing game tech with the Angry Birds licence. Initially it was envisioned as a kart racer along the lines of Mario Kart, but it quickly evolved into a downhill racer.

The Angry birds themselves take prime focus and it was the first time they had been rendered in three dimensions and featured in a game outside of the traditional physics based gameplay established by the IP.

The technical challenge was turning into an enormous mountain we would have to climb.

Engine

- XGS - Exient Gaming Software
- Multiplatform modern game engine
- Fast data builds
- 200 million users



At the core
Angry birds Go uses Exient's own game engine, XGS - which stands for Exient gaming system

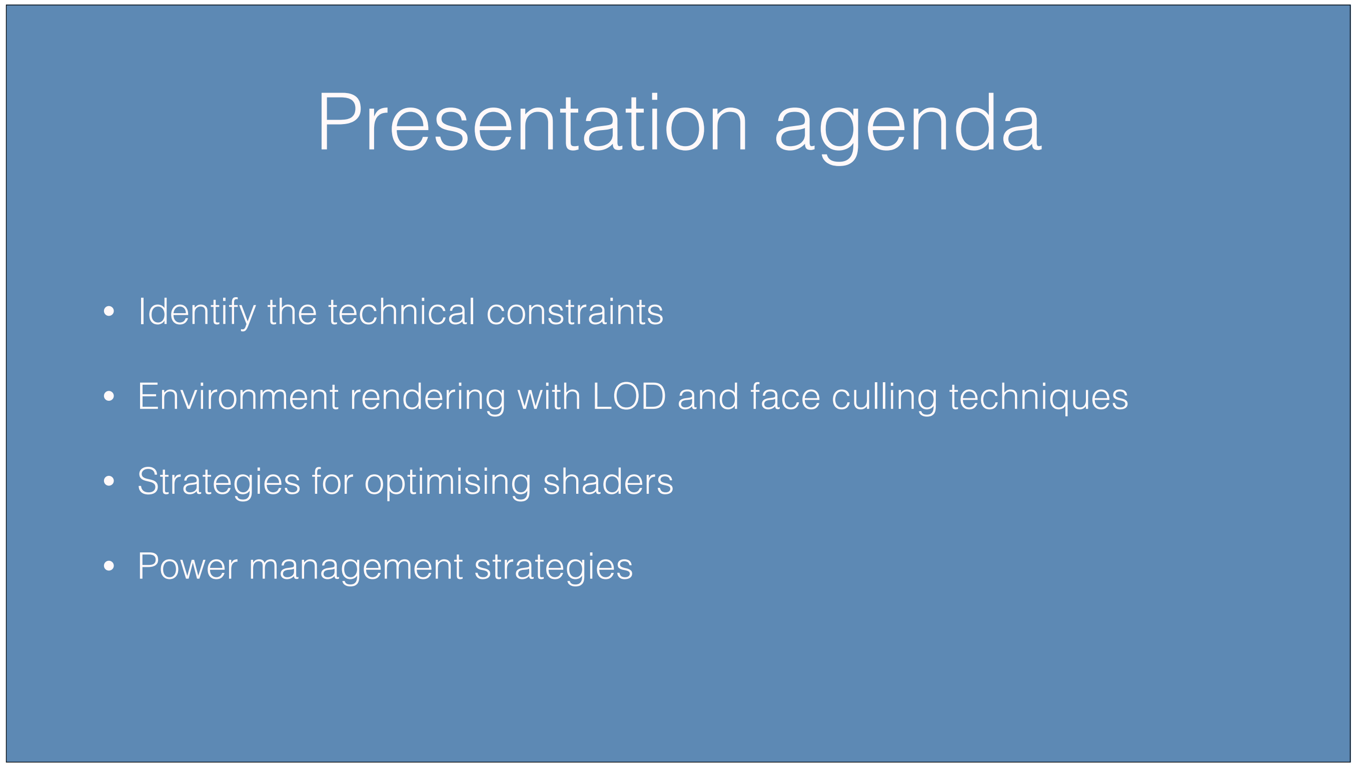
A multi-platform game engine using a centralised build system.

We emphasise fast data builds for quick iteration. The engine is build around a core framework that is very flexible which means we can create bespoke solutions for games.

We have roughly 200 million players on XGS games. These include, Diggs:Nightcrawler for Sony Wonderbook, AngryBirds trilogy, Starwars and Transformers and several Need for Speed and Fifa titles.

One great advantage XGS gives us is that it is the master of it’s own destiny, it’s not bound to 3rd party solutions.

“XGS allows us to provide specific solutions to specific problems.”



Presentation agenda

- Identify the technical constraints
- Environment rendering with LOD and face culling techniques
- Strategies for optimising shaders
- Power management strategies

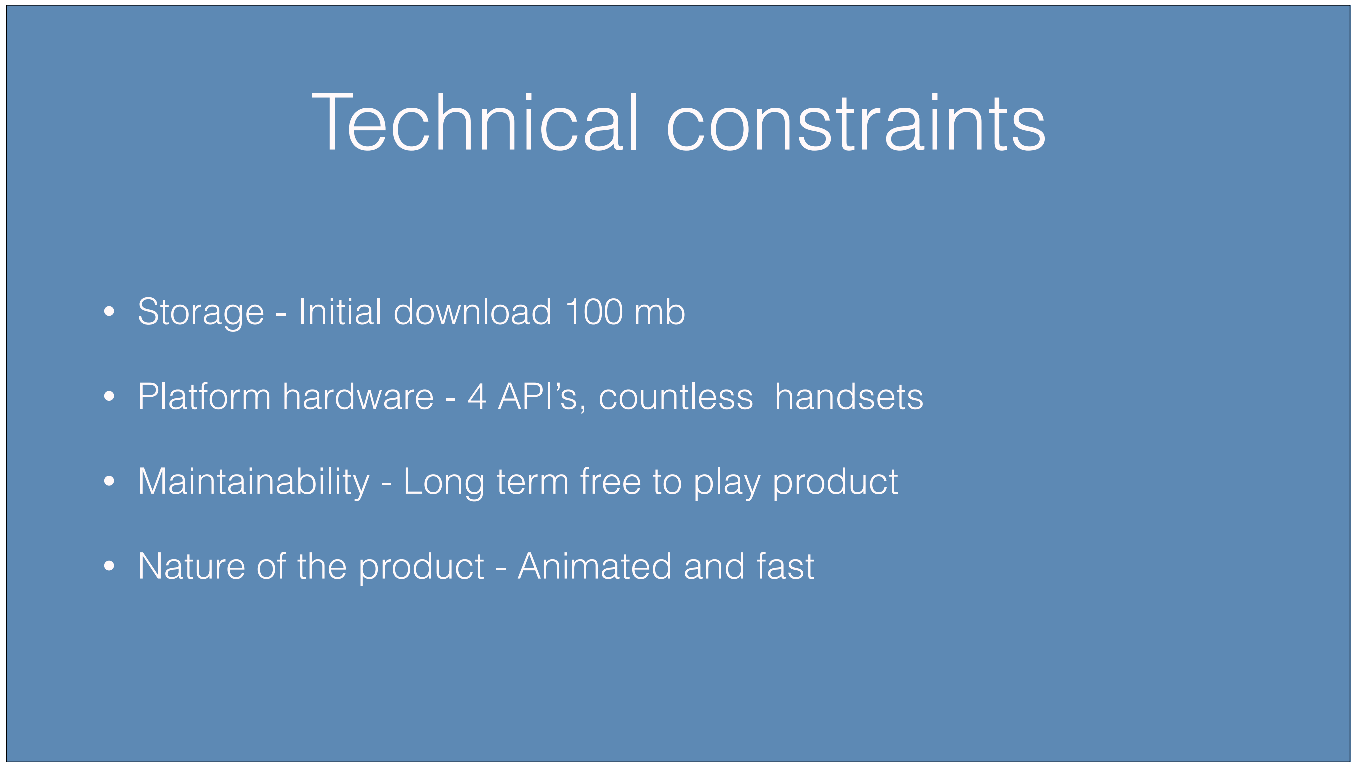
So thats covered a description of the game and summarised the core technology used to create it.

What does this presentation deal with?
I am going to begin by taking a look at the technical challenges that faced the development team and identify the key areas where most technical expertise was required and the reasons why.

The next section focuses on environment rendering where we explore the technical challenges of rendering fast, fluid race tracks on mobile devices and give an explanation of our LOD and geometry culling techniques.

Following this I'll explain the shader optimisations we implemented for normal and post processing shaders

Finally I'll end with a brief look at some of the power management strategies we employ in our mobile games.



Technical constraints

- Storage - Initial download 100 mb
- Platform hardware - 4 API's, countless handsets
- Maintainability - Long term free to play product
- Nature of the product - Animated and fast

To begin, I'd like to share what I consider to be the most important technical constraints that underpinned the development of the title.

Storage

- Free to play product - more downloads means more players which means more potential revenue.
- The initial package size has to be small in order to encourage downloads on mobile data.
- Apple originally specified a 50mb maximum size, later expanding to 100mb

Platform hardware

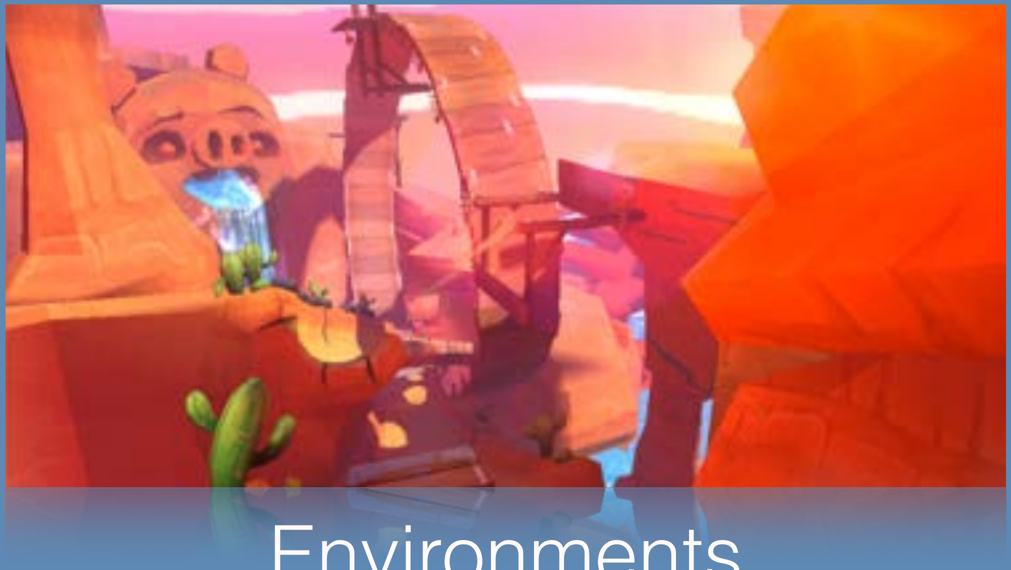
- We released on IOS, Blackberry, windows Phone 8 and Android
- Based on their experience, our client predicted that 50% of the projected revenue would come from devices using Android.
- We needed to release on as many Android handsets as possible, therefore the differences in device specification varied enormously.
- iPhone 5s and the iPad air was due for release as we finished the game, so these were initially our top spec devices.

Maintainability

- As a free to play title, revenue is directly linked to its lifespan. We are financed through people playing the game, if they stop playing they stop paying.
- We need to be able to react to changes in player demand.
- Its important not only to be able to update the game with new content but also to react to improvements in operating systems and hardware. So that every user feels they are playing a quality product.
- We were able to leverage Apple's Metal API to provide HDR visuals.

Nature of the product

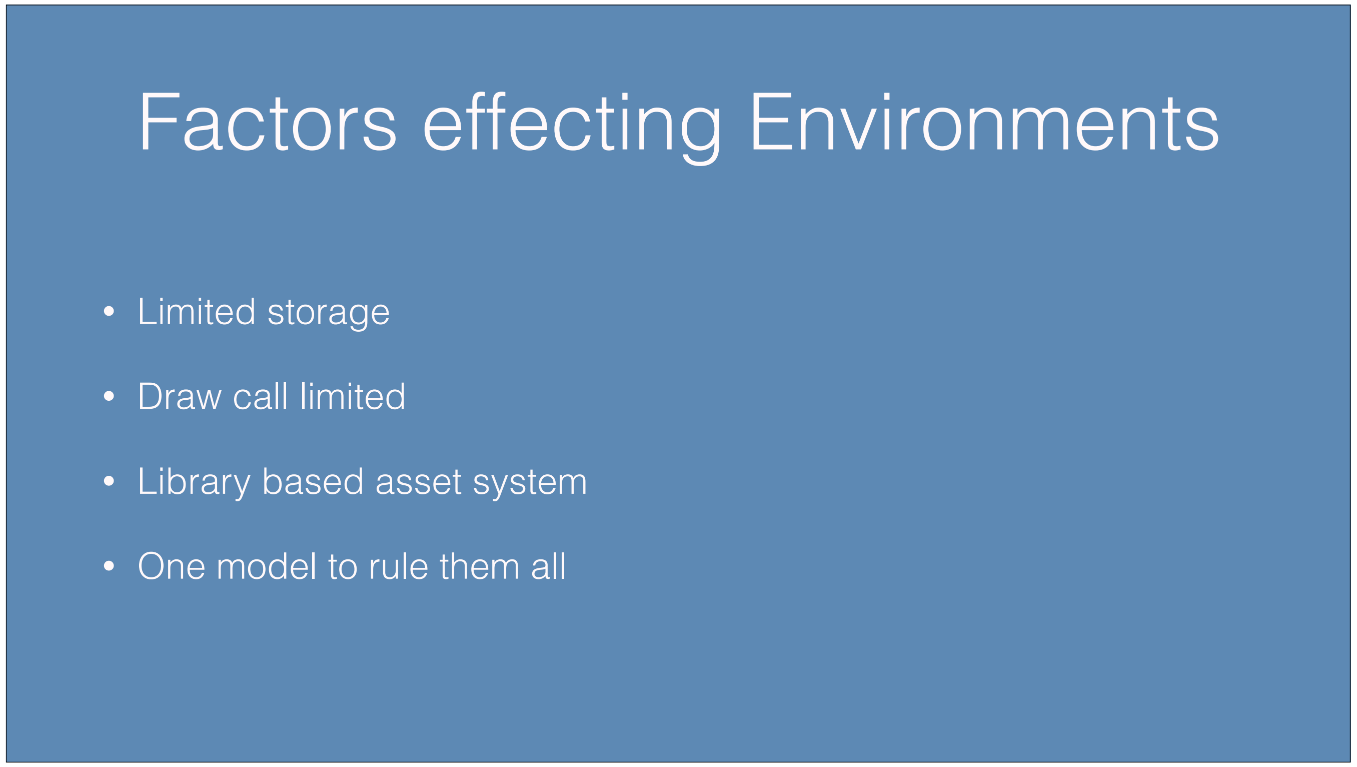
- The game is character based, meaning we absolutely have to portray them in the most sympathetic way, the characters ARE the IP.
- We introduce them in an environment that is arguably the worst case for graphical performance: i.e Multiple skinned characters in destructible vehicles racing extremely fast in environments that offer few occlusion opportunities on a mobile platform.
- Lots of animation and extremely fast rendering are prerequisites.



Environments

Bearing these constraints in mind, lets take a look at environment rendering:

The trouble with racing game levels is that the player can get through races pretty quickly.
We need the initial download package to provide, say, an hours worth of entertainment.
Angry Birds Go's download package included 2 themes each with 3 different tracks, this provided a reasonable amount of gameplay before new tracks need to be downloaded.
Environment rendering needs to be fast and slick. Lets now take a look at some of the factors effecting Go's tracks.



Factors effecting Environments

- Limited storage
- Draw call limited
- Library based asset system
- One model to rule them all

Storage
Typically, environments take up the most space in the package. A single race includes the level geometry, LOD's, collision geometry, Ai paths, texture sets and a whole host of other bits of data. We have to take steps to minimise the environments storage footprint.

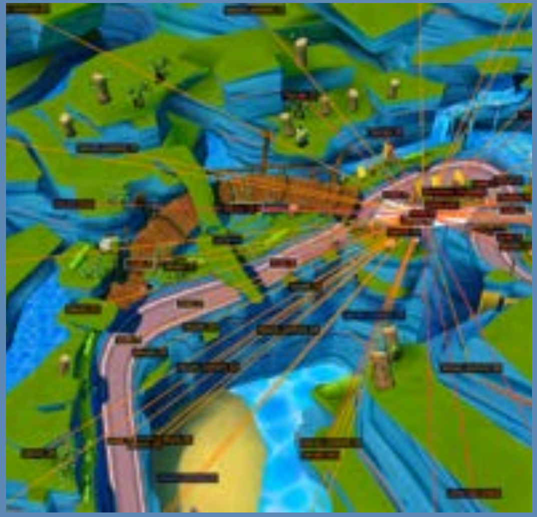
Draw call limited
Based on the nature of mobile rendering, we know we need to batch draw calls to save performance wherever possible.

Asset Library
Assets are placed by the artists from a library in order to accelerate the environment building process. This means that the mesh geometry is not always optimal for performance. We need a method to cull any non-visible geometry before it gets rendered without significantly impacting the art team.

Performance across the device range
We wanted to use one environment model that will work on all devices so that artists could focus their creative efforts. Per device optimisation can be controlled through shader LODs.

Existing Env Pipeline

- PVS visibility system
- 3d tools allow artists to place assets from a library.
- Baked vertex colour lighting with perspective shadow maps

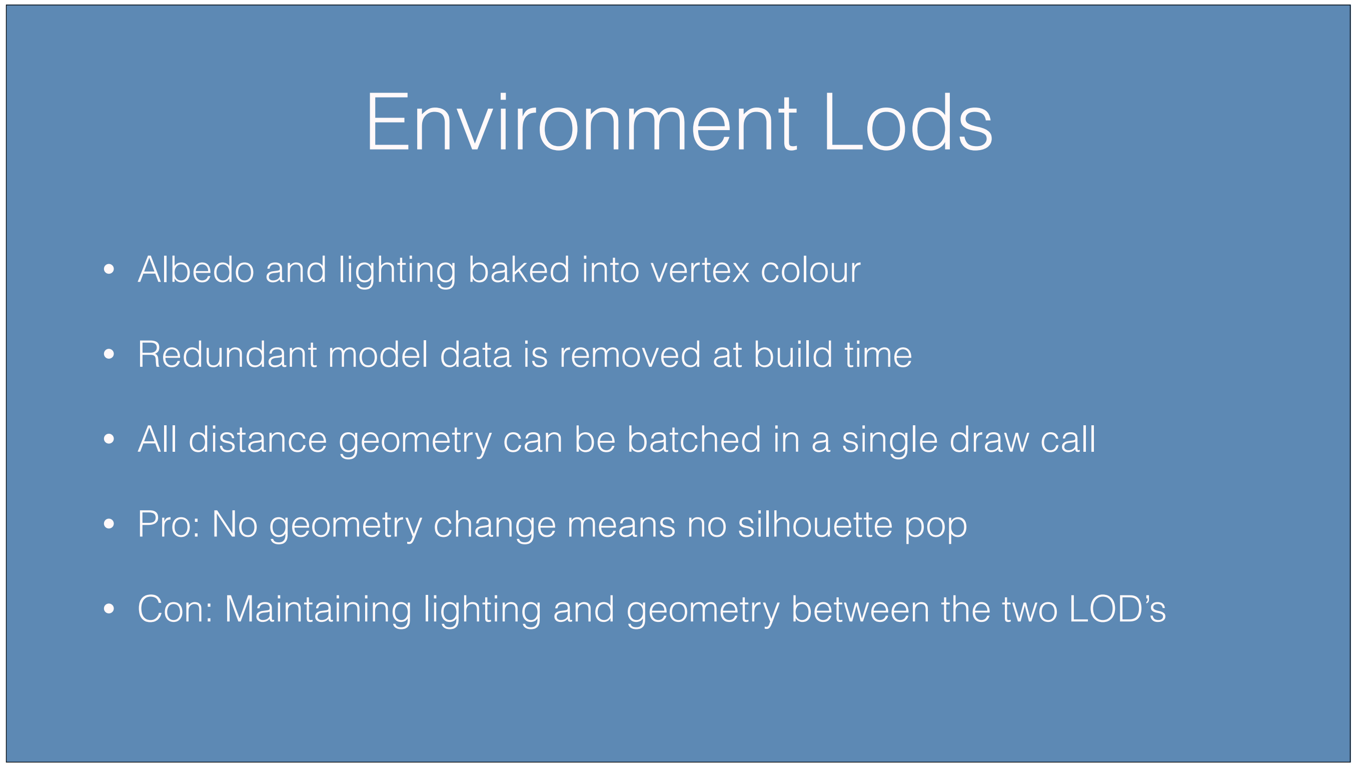


Visibility system
Our environment pipeline utilises a PVS system (potential visible set) managed by a tool within the build system.
- The environment is manually split into sections. For each section we store the visibility and level of detail of all the other sections in a database.
- This results in a fast system that gives artists control over the visibility of objects.

Library based object placement
Within the 3d authoring software we provide an environment building tool that allows artists to place models from a library. The tool joins the assets to the PVS sections at export from the 3d package.
The model converter in the build pipeline optimises these sections based on device - removing any unnecessary model data (tangents, UV's) in order to streamline the storage space.

Vertex colour lighting with shadow mapping from karts and dynamic objects.
We used baked lighting for the environments, the decision to use vertex colour was driven by two factors.
- Low storage footprint
- It suited the angular look of the cartoon art style
- The closest environment vertex provides an inexpensive method to get the in-direct lighting value for a kart.
- Light baking was a final stage environment process.

Perspective shadow maps provide grounding shadows for the Karts and physics objects on high end devices, lower spec devices use a simpler method.



Environment Lods

- Albedo and lighting baked into vertex colour
- Redundant model data is removed at build time
- All distance geometry can be batched in a single draw call
- Pro: No geometry change means no silhouette pop
- Con: Maintaining lighting and geometry between the two LOD's

Environment Lod's are necessary to achieve the desired performance for a fluid racing experience
We have to take into account the increased storage requirement for extra model geometry.
For best performance the lod system needs to minimise the draw calls in order to provide a benefit on all devices. A common method would be to bake the terrain sections textures so it can be drawn in one call.

Our solution was to use one set of LOD's with vertex colour providing lighting and diffuse colour.

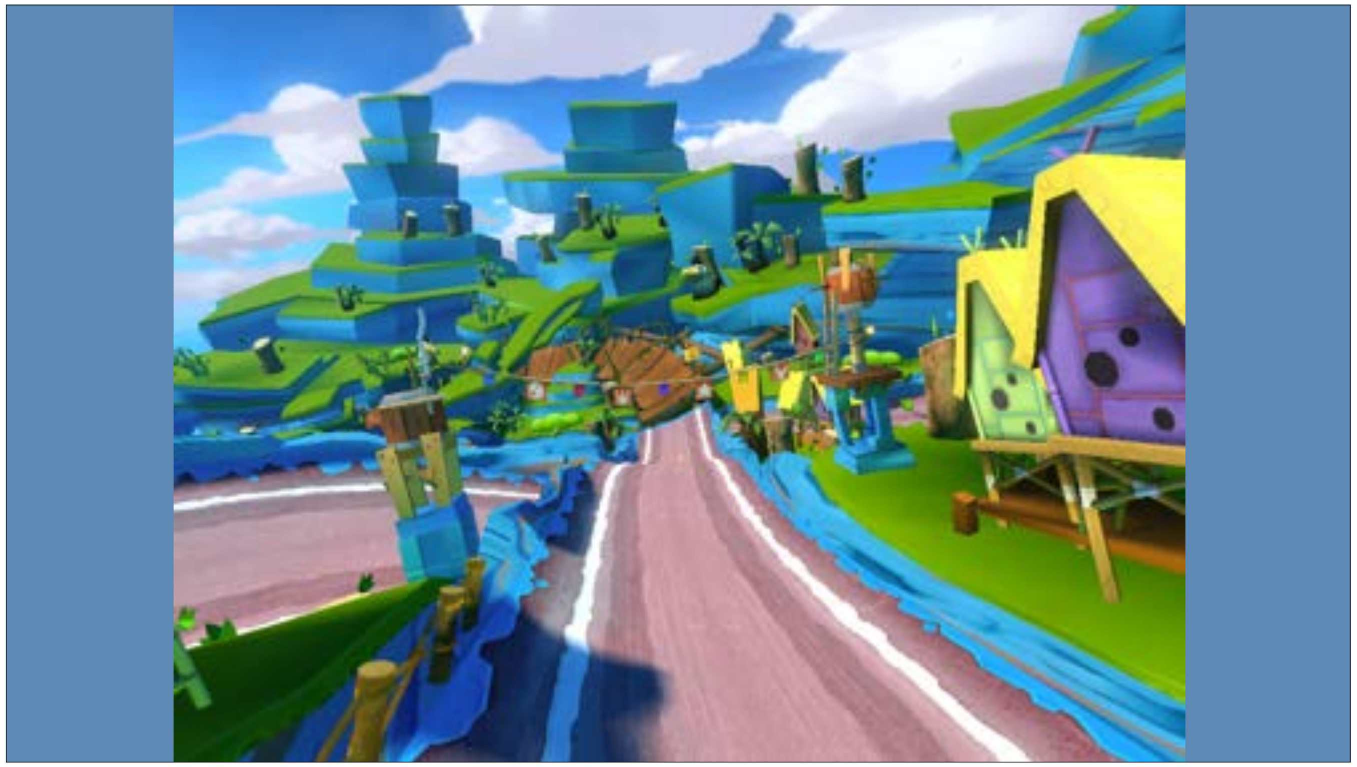
Copy the model and combine lighting and texture and bake to vertex colour
We take a copy of each PVS chunk and combine its lighting and texture colour and bake it to vertex colour in the 3d software. The rendering is achieved with a simple shader that requires no texture samples, just a fog algorithm.

Remove redundant data
Any redundant model data, such as smoothing and UV's is removed from the asset at the build stage. This is done to reduce the storage footprint for the LODS

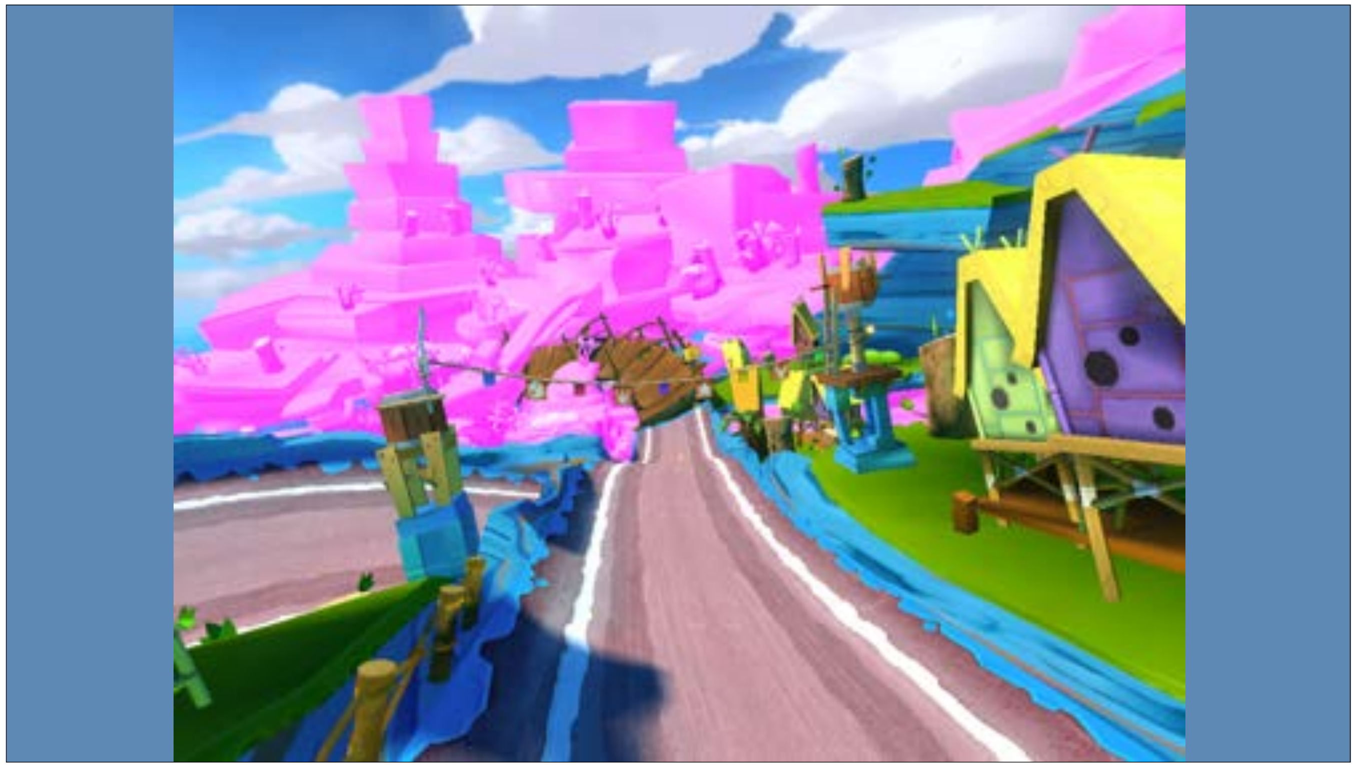
Result
The result is a fast rendering Lod that can be batched meaning all low lods can be rendered with a single draw call.

Pro:
There is no visible geometry change and therefore none of the silhouette pop that's traditionally associated with LOD switching. Many people have been surprised when the LOD's are pointed out, the change from high frequency colour information to low frequency is much less perceptible to the human eye than a change in form.

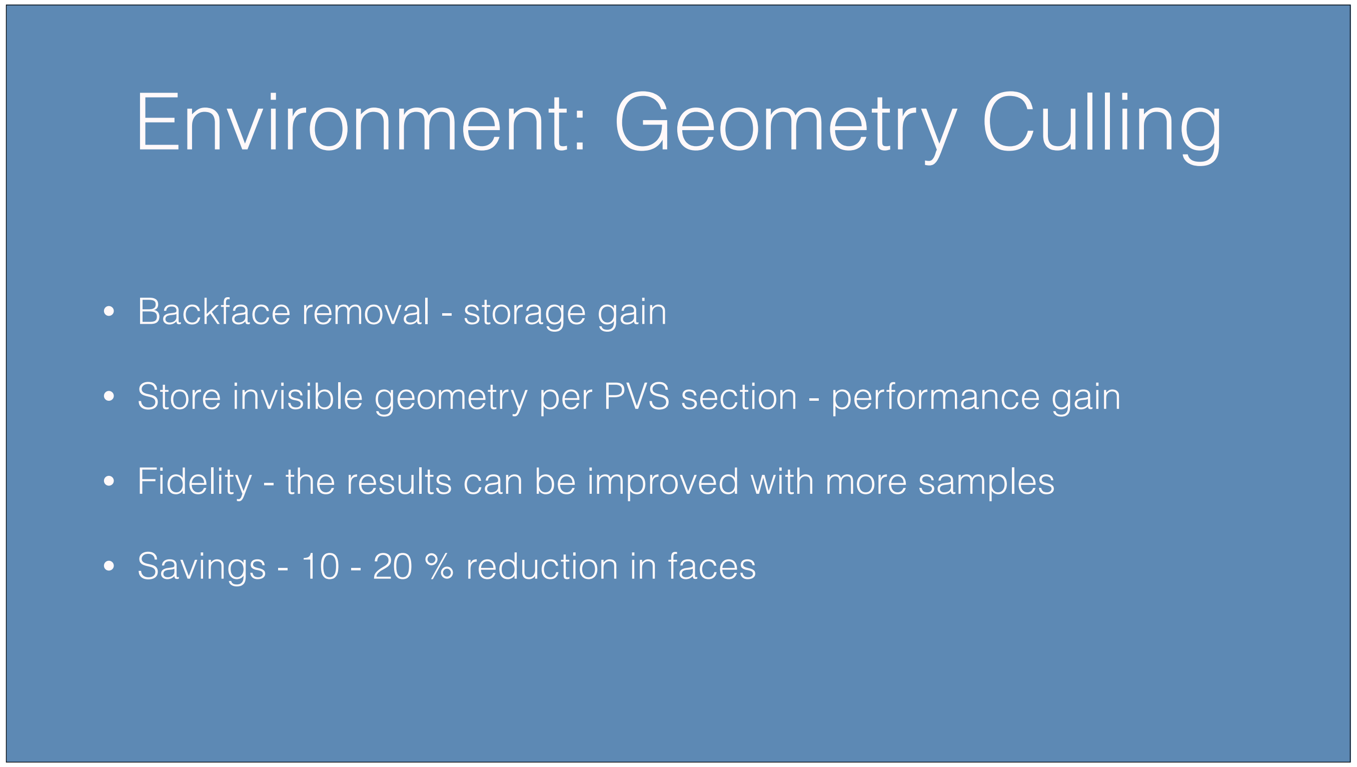
Con:
The downside to this is that the process is dependent on the level geometry and lighting being completed.
In practice we made this negligible by creating a script that creates all the levels lods in a matter of seconds. It became a simple matter for the artists to manually re-generate LOD's when required.



Heres a typical Angry Birds:Go! Environment showing both high and low LOD’s…..



.. here are the low lods highlighted.
Once pointed out you can see it clearly. But in motion, the players vision is focussed on the road in the middle distance and is oblivious to the effect. We do take some extra steps to hide the the switch, such as when the player goes under a bridge or round a sharp bend.



Environment: Geometry Culling

- Backface removal - storage gain
- Store invisible geometry per PVS section - performance gain
- Fidelity - the results can be improved with more samples
- Savings - 10 - 20 % reduction in faces

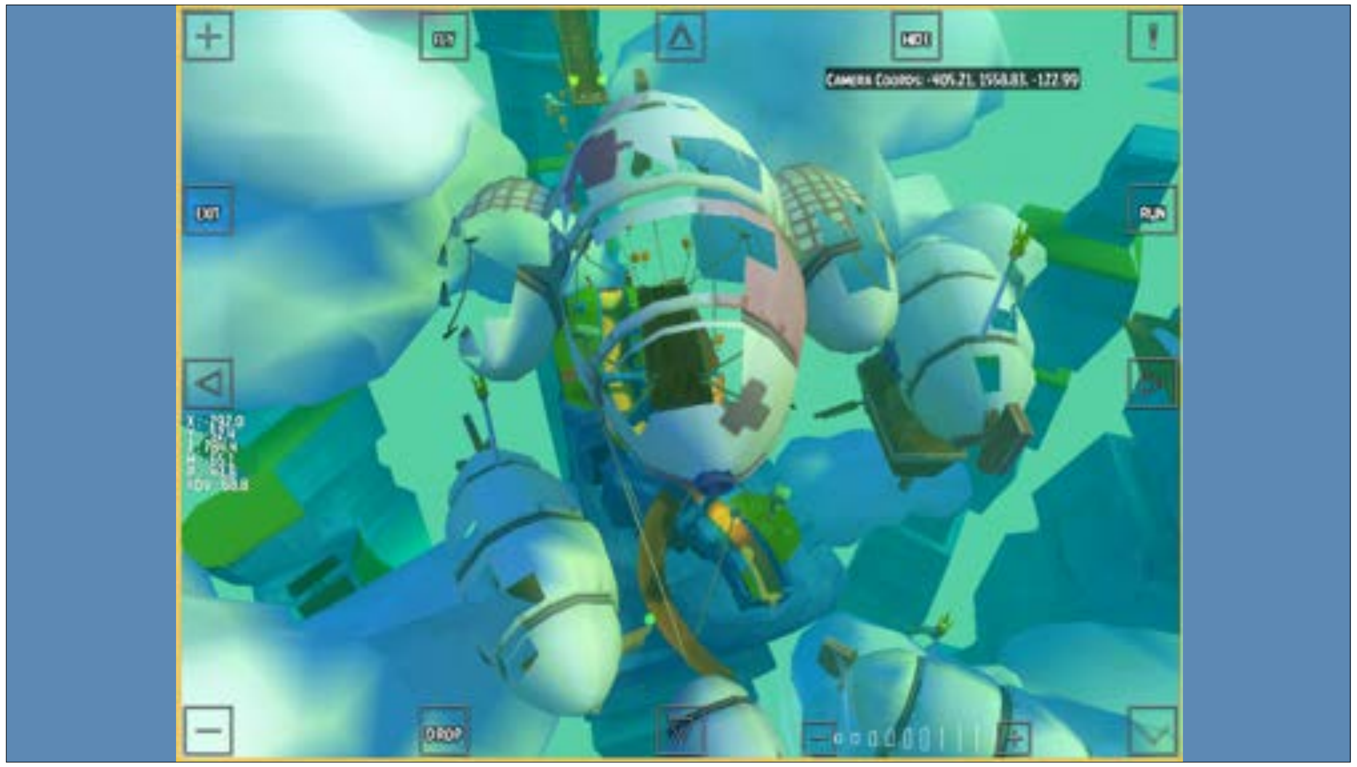
PVS provides a real way to control visibility manually but at the expense of time. The environment is Octree processed so some culling can be done at run time, frustum culling for example.
Since the environments are assembled using objects from a library it makes sense to cull non visible geometry before it gets into the package and thus reduce the storage requirement for the level.
We could do this before export from the 3d package, but its much faster as a process in the build pipeline.

Backface removal
The first stage culling is backface removal - A virtual camera is flown along the route of the track, any faces pointing away from the camera are removed from the built model. Surprisingly the storage savings are quite significant and since there is no real-time processing involved this optimisation is applied to every level.

Per PVS backface removal
The second stage involves storing each sections occluded face-list on a per PVS section basis within the PVS database. This significantly improves the rendering performance but with a cost to storage. The model remains unchanged but the PVS database increases in size. There is some cost in removing the face-lists from the assets before they are passed to the GPU.

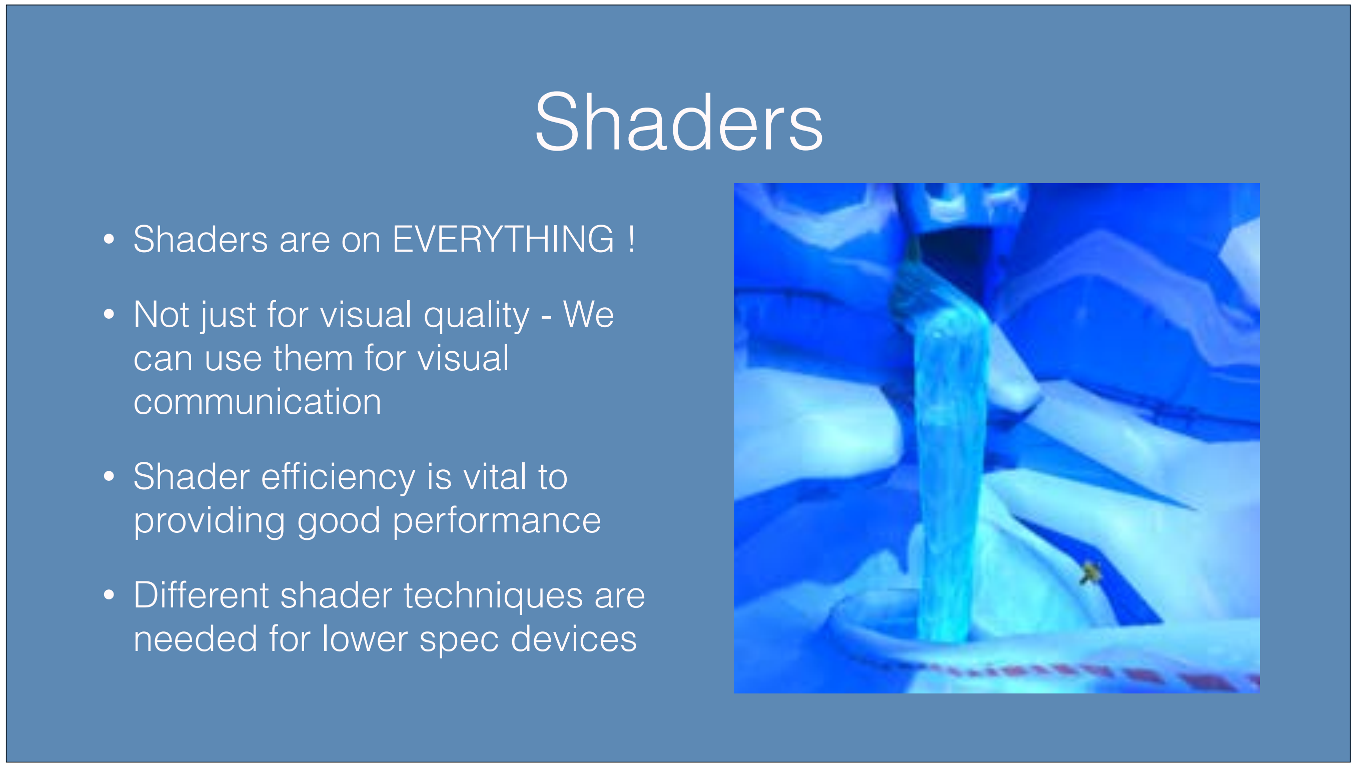
Fidelity
Both methods can use the existing AI paths as a route for the virtual camera , alternatively artists can specify multiple paths. For example, to define the bounds of a corridor through the levels.

Savings
The savings are dependent on the design of the levels. On average simple culling provides a 10 - 20% reduction in faces, whilst per PVS culling provides the same again but at a 10% storage cost.
In practice the two techniques used together negate the storage benefit but provide an overall performance gain.



This image shows several of the same airship assets, with different faces that have been culled. This optimisation is in the model data so it has no performance overhead in the game.
The culling is re-calculated in the build pipeline every time the environment model is changed. The artists can continue to work with the original library assets as required.

Angry Birds Go! relies on a flexible shader system to provide some fast environment rendering. In the next section I cover some of the methods employed to manage the shader database and our solution for a practical shader LOD system.



Shaders

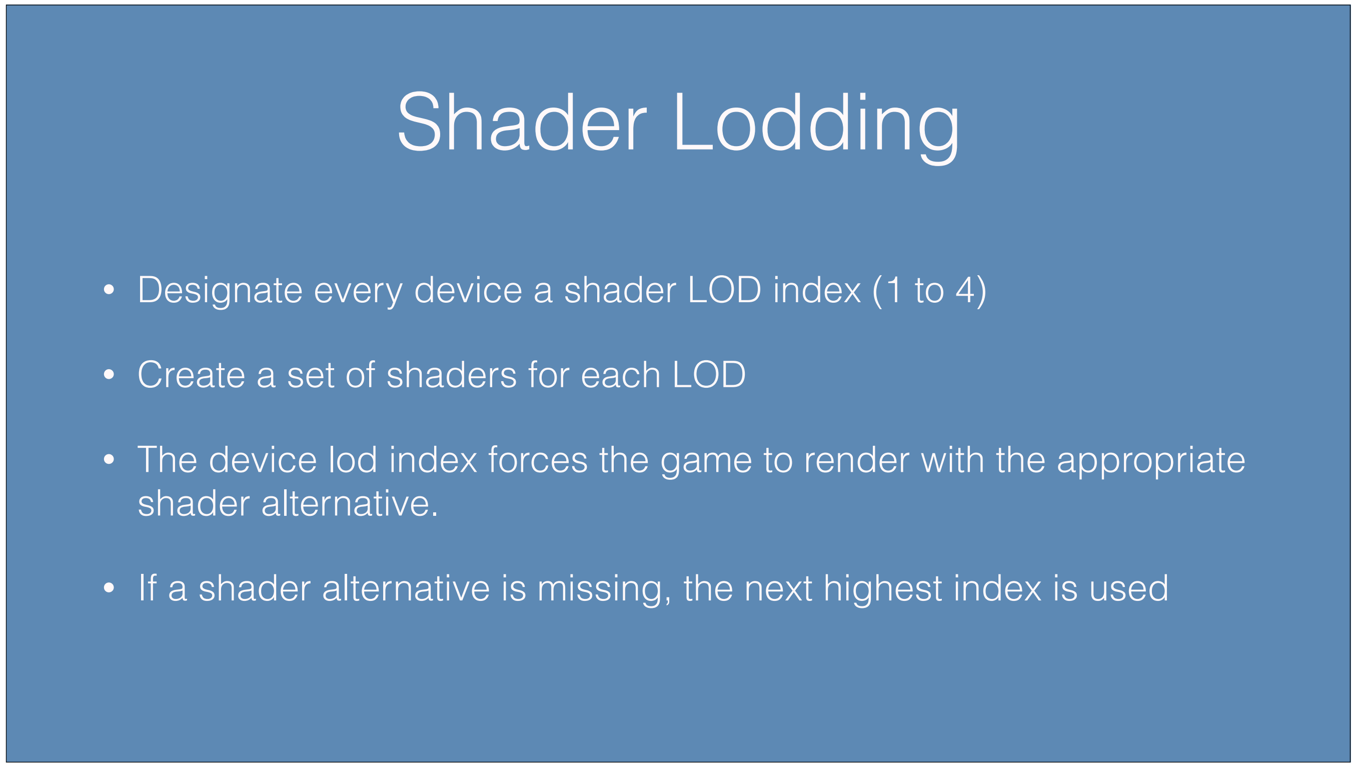
- Shaders are on EVERYTHING !
- Not just for visual quality - We can use them for visual communication
- Shader efficiency is vital to providing good performance
- Different shader techniques are needed for lower spec devices

What are we talking about?
Pretty much everything seen on screen is processed by a shader of some description.
….which means as a shader artist, I can point to any pixel in game and say “hey!, I did that!”

What are they used for?
Apart from providing visual quality we also use shaders to solve some visual communication issues. For instance, highlighting interactive objects to the player through a glow effect.

Whats the problem?
The more complex a shader is, the longer it takes to render.
Shader performance is often dependent on outside influences. Shader uniforms and variables can effect efficiency.

Shader optimisation won’t help with all situations.
Whilst efficient shading gets some way towards a smooth framerate, not all devices are equal; we can’t guarantee a particular rendering method is practical for all devices. Shader lodding is necessary to support the lower spec devices within the range.



Shader Lodding

- Designate every device a shader LOD index (1 to 4)
- Create a set of shaders for each LOD
- The device lod index forces the game to render with the appropriate shader alternative.
- If a shader alternative is missing, the next highest index is used

We could use defines for each device throughout the shader code to cover all eventualities, but this is not a very maintainable system when completely different shader techniques are required.

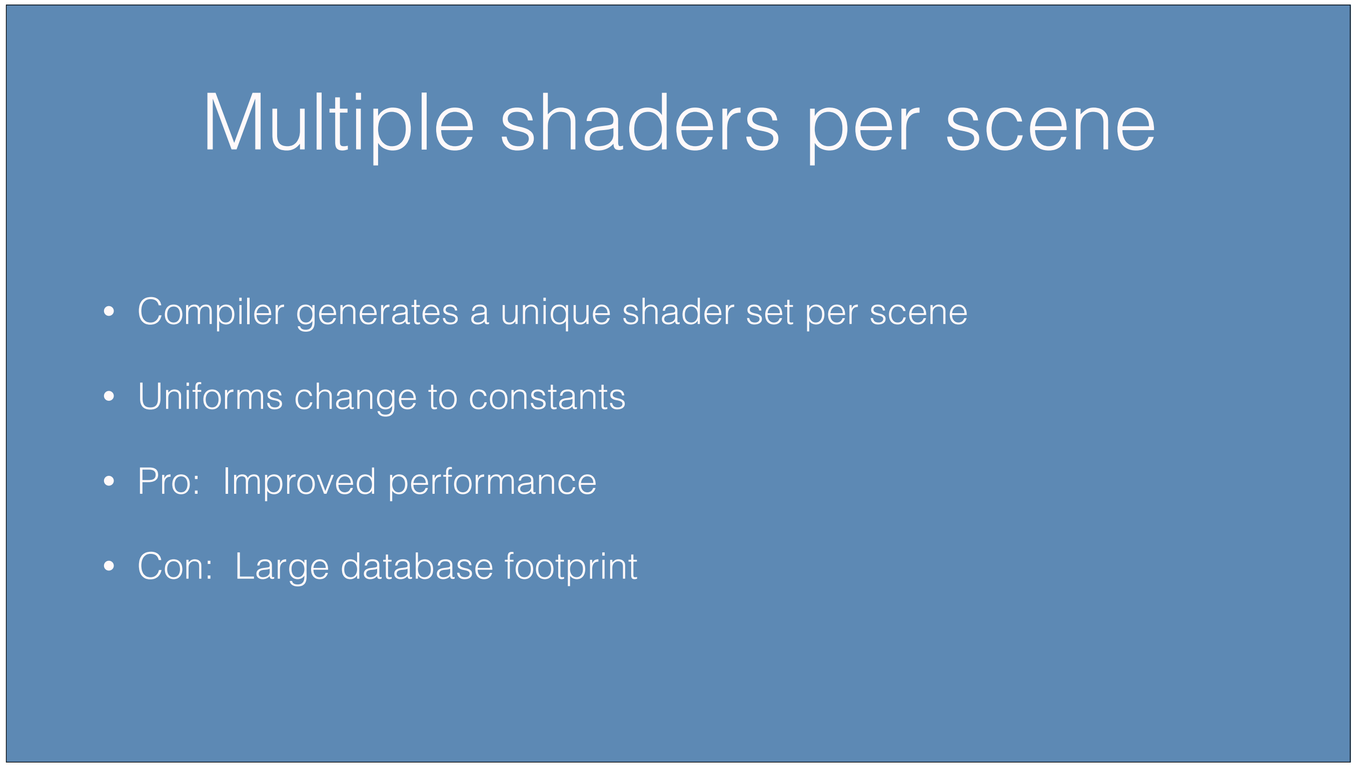
Designate the devices
We designate every device a shader LOD index, ranging from 1 to 4, based on it's capabilities.

Create alternatives
We create a set of shaders for each LOD index. The alternative LOD shader names are stored in a list within the high lod shader.

Alternative shaders are drawn based on index
High lod shaders are applied to geometry and referenced in code as normal - however if the device shader LOD index is not the highest the code overrides the high lod shader with the appropriate alternative.

Simplifying the database
To avoid duplicating shaders that are already low overhead, we don't create an alternative. If a shader alternative is missing, the system uses the next highest LOD. The system also means that several high lod shaders can refer to a single alternative, further simplifying the shader database management.

Not all devices could support shadow mapping. Only the high lod shader includes the necessary shader code, lower lods have this removed, and a CPU based technique is used. The game code is aware of the LOD index and will not render the depth buffer.



Multiple shaders per scene

- Compiler generates a unique shader set per scene
- Uniforms change to constants
- Pro: Improved performance
- Con: Large database footprint

So now we have a per device specification for shaders….
What ways can we improve shader performance without effecting flexibility?
We know that shaders will use some common variables on a per theme basis. Such as ambient lighting colour, fog ranges and so on.
If we created a unique shader set for each theme we could hard code the variables into the shader - but at the expense of flexibility. Instead of managing one shader, we are managing one shader per scene.
The answer to this problem is to let the shader compiler do the work behind the scenes.

Compile shaders against scene database
We store per scene uniforms in a scene database and tell the compiler to generate a unique shader set for each scene. Scenes can be seen as a storage area for shader uniforms and can be edited by artists.

Uniforms to constants
Uniforms are changed to constants, hard coded values within each shader.

Pro: Improved performance
Shader performance is improved and we retain the flexibility to change these variables on the fly by recompiling and reloading the database to give nearly instant feedback.

Con: Larger database footprint
The only tradeoff is a larger storage footprint for the shader database, the original size multiplied by the number of scenes.

On the whole its fairly inconsequential when compared to the storage footprint for other asset streams. When I last checked, the compiled shader database for Go! was around 9mb, this includes the device lods and shaders compiled for multiple scenes.

Post Processing

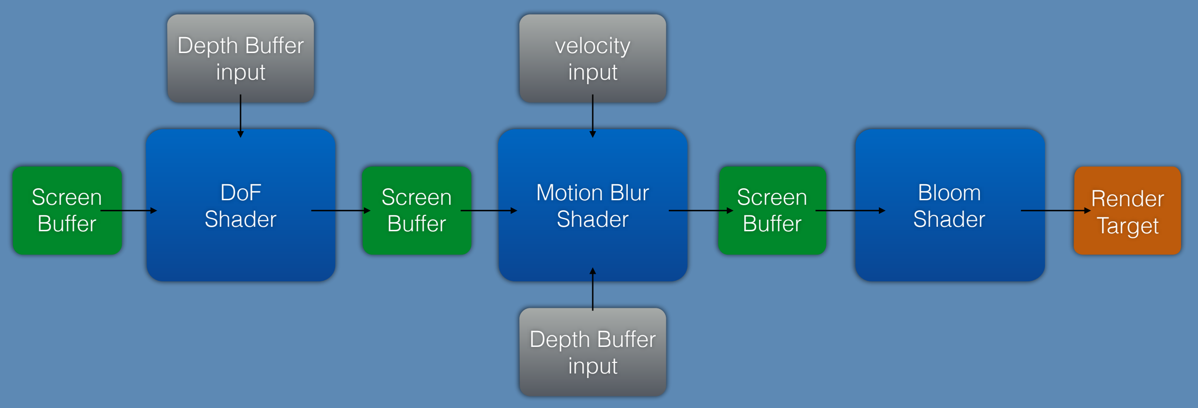
- Bloom, Depth of Field, Motion Blur.
- Provide final-stage visual styling per scene.
- Full screen per pixel effect

What are we talking about?
Post process shaders such scene Bloom , Depth of field effects , motion blur

What do they do in AB:Go! ?
They provide final stage visual styling for the art director and lighting artist, different scenes will use different settings.

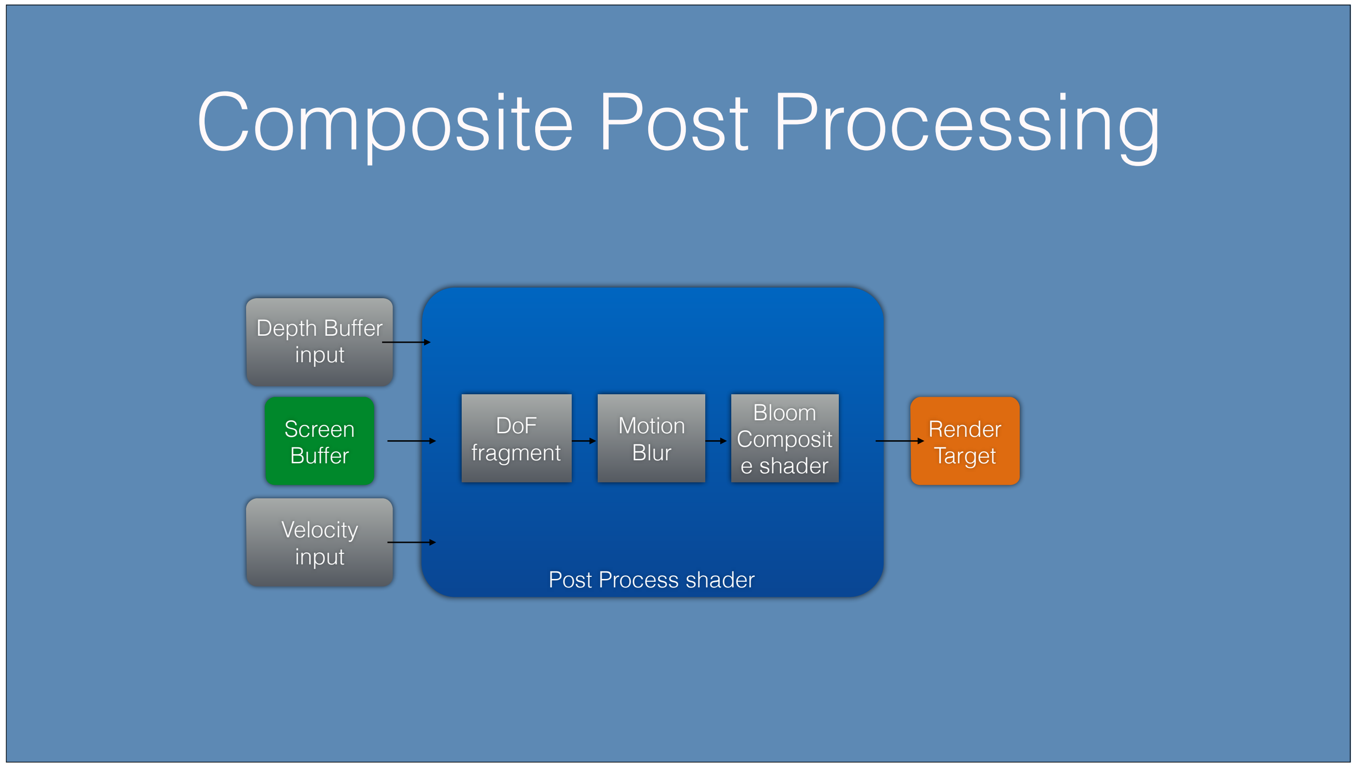
Why is it a problem?
Expensive to render as the operation is done per screen pixel. More techniques equate to slower performance.

Traditional post processing



What do we know?
Each shader takes an input image, does some stuff to it, and writes an output image.
Each shader does a lot of the same things e.g decoding a buffer, loading a render target.

What can we do to improve performance?
It's obvious really, lets combine the individual shaders into one.
>>>NEXT



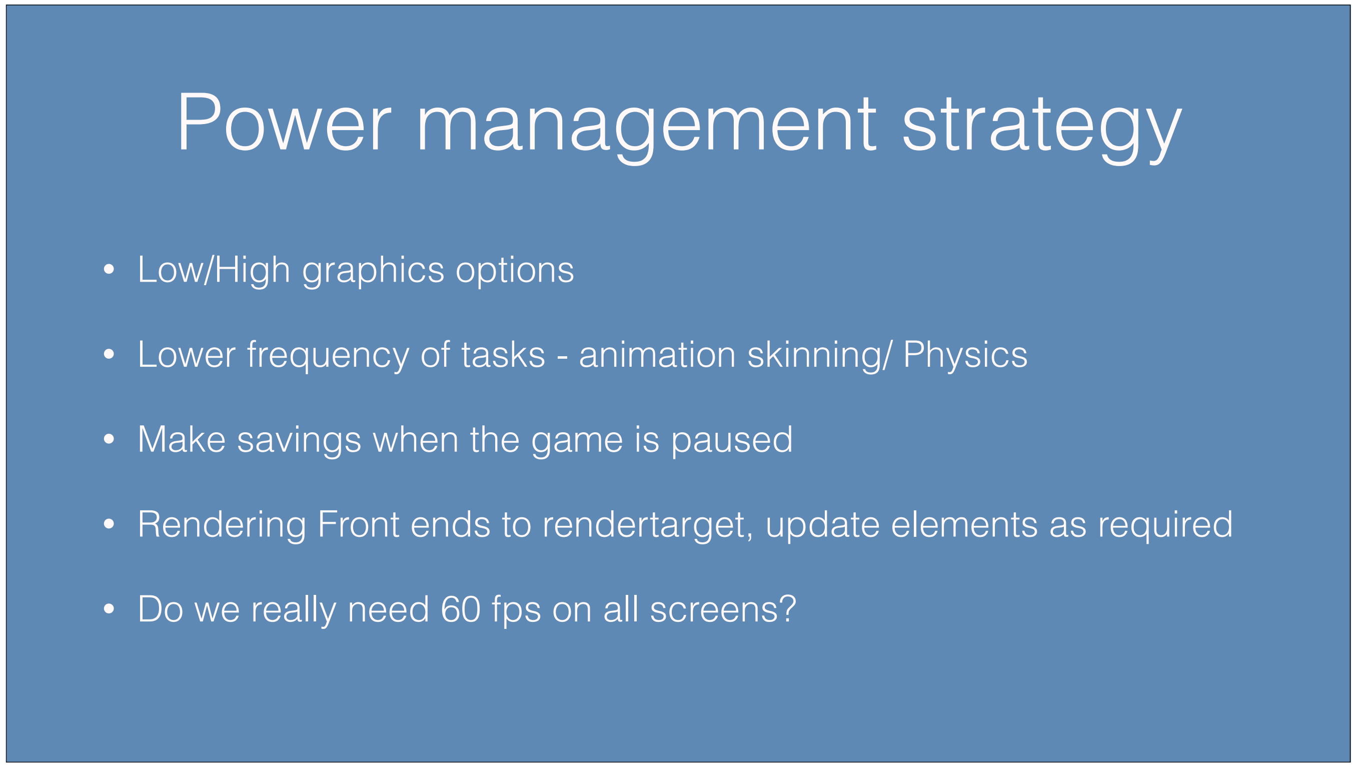
Composite Post Processing

This combines the operations into one shader reading one buffer and writing the render target.

Each post process effect is now a composite shader in our editor - a shader within a shader - in practical terms each effect is seen as a fragment, but for workflow purposes it enables the shader artist to still work on a specific effect in isolation.And there's no impact to storage, if anything this shader is smaller in size than the other three combined.
As far as the GPU is concerned, it's not using multiple render targets and the pixels are only drawn twice (once in the main render pass and once in the post process pass)

Within the compiler, if any shader thread results in zero, it's removed from the shader at compile time. An artist can simply disable an effect through a scene constant….
…and using the per Device shader LOD technique we can opt to remove or change processing passes dependent on device capabilities.

Ultimately, the techniques I've presented for shader management provide small but worthwhile gains to performance. A hidden benefit is that most are designed to give a performance gain with the least disruption to workflows and pipelines.



Power management strategy

- Low/High graphics options
- Lower frequency of tasks - animation skinning/ Physics
- Make savings when the game is paused
- Rendering Front ends to rendertarget, update elements as required
- Do we really need 60 fps on all screens?

As a final slide, I'd like to briefly share some power management strategies we use in some of our games.
It wasn't until I was researching this talk that I became fully aware of how much of an issue it actually is.
From the point of view of a developer making a mobile game, a power hungry game creates a situation where the user avoids playing when mobile - defeating the object of mobile gameplay. We should avoid forcing people to cripple a mobile device by tethering it to a power supply.

Of course, others can speak with more authority about power management, the following techniques mostly intend to make a saving by reducing the load on the GPU.

Low/High Graphics options
Seems like a good idea, but in reality isn't as practical as it seems. Most users will default to high regardless.
Lower frequency of CPU tasks
Reducing the frequency of tasks such as animation skinning or Physics can lower the load on the CPU and improve battery life.
Game pause savings
When paused, Go's screen is de-focused using a post process shader. When the process is complete we render to texture and write it to the screen buffer. Ending a power hungry process.
Render FE to rendertarget
We can render the Front end to a render target and choose to only update the parts required.
Do we really need 60fps on all screens?
Users will accept metagame map screens and front ends running at a much lower framerate than the core game loop.

Before I finish, I'd like to take a moment to summarise everything I've covered:

Initially I identified the technical constraints that underpinned the development and showed how storage, maintainability and design can influence a mobile product.

You were shown our environment LOD technique and the face culling methods we employed to make racing environments more efficient for mobile platforms.

We looked at how storing shader uniforms as constants can improve shader efficiency by letting the compiler create duplicates per scene. After this I explained a shader LOD technique that enables a single environment to work with a diverse range of devices.

I demonstrated how post processing shaders can be combined in a single composite shader, saving render targets and optimising performance by not duplicating shader tasks.

…and finally I shared some of the strategies we employ to save battery life on mobiles.

Take aways

- Taking a straightforward approach to rendering
- Find ways to optimise that will give the biggest impact for the least disruption of the pipeline
- Breaking tradition - current rendering trends aren't always the best
- Trust and teamwork

Taking a straightforward approach to rendering.
When the art style leads the technology it can be a good thing. In our case, vertex colour lods would be inconceivable for a title with realistic graphics. Ultimately, simplicity is the key to success.

Find ways to optimise that will give the biggest impact for the least disruption of the pipeline.
The environment LOD's for example, provide excellent performance with little sacrifice to visual quality as well as saving the art team a lot of mundane polygon reduction work.

Breaking tradition
Quite often technical requests are based on current practices and trends rather than what is really needed for the product. It's important to get to the reason behind a request, quite often what's asked for isn't actually what's needed.

Trust and teamwork
Code, art and design focussed on creating the product as a whole. These days its common to see departments producing incredible work, but often in isolation of each other without awareness of how the other departments view the product.
A technical artist is a lynchpin between the teams. As they grow to trust you, barriers begin to break down and they begin to trust each other and thats when it starts to get really exciting.

Thank you!

Visit us at www.exient.com

simon@exient.com