Hello,

My name is ben sugden and I am the lead programmer at lionhead studios on the incubation team,

and I'd like to talk about some of the work we did when developing the engine technology for milo and kate.

Today

- Milo & Kate Art Direction
- Modelling tool and pipeline
- Build pipeline
- Compression and streaming
- Virtual Texturing
- Real-time GI solution

First of all today, I'm going to talk a little about the art direction for the game and discuss the pipeline we have created to address some of the modelling challenges this threw up and the associated build process for it.

Then I'll talk about how we use the data generated by this in the runtime code, talking a little about our compression scheme and then I'll talk for a short while about our virtual texturing system.

Finally, Michal Iwanicki is going to present some material on our lighting system and introduce a new realtime global illumination algorithm we have developed.

Art Direction

Milo & Kate

Environment: Shipwreck up against cliff-top with overgrown deck
RD_MaK056_EnvironmentSketches_01.psd

LIONHEAD STUDIOS

So, for milo and kate,
we were presented with a challenging art direction;

To render a world seen through the eyes of a young boy.
We decided the world needed to have a hand sculpted feel
but have the softness
and subtlety of realistic lighting.



So, as you can see from these pictures, we wanted bold shapes with areas of fine detail.

And we wanted all the elements of the world to feel as if they flowed into each other.

One thing in particular that we wanted to try to avoid were hard polygon edges when objects are obviously placed onto of other objects,
or when two objects are intersecting
rather than nestling together naturally.

These obvious sharp edges give a very unnatural interface between two surfaces, and break the illusion of the world being real, and telegraph that this is a world made up of triangles.

Instead we really wanted to try to make the world to appear be a coherent, single whole,
like one huge painting or sculpture.

And its no good having amazing geometry if you can't light it well, so we obviously needed some great lighting to really show it off.
And it had to run on an xbox 360.

So let's talk about how we achieved this.

Conventional Environment Modelling

- Generally done in 3 phases
  - Build mesh in DCC of choice – max / maya / xsi
  - UV unwrapping step
  - Texture it in Photoshop
- Result is artists think in 3D then 2D
  - Not playing to their strengths
  - Image space texturing poor way of approximating medium frequency surface details
  - Models often look like wallpapered tri meshes

So, first of all I'd like to recall how we conventionally model environment meshes.
It's generally done in three phases and usually across two or more packages
First you build a mesh in the modelling tool of your choice, be it max, maya or xsi
Then you UV unwrap everything.
And then you move to photoshop or something for authoring textures, bump maps etc.
We're all pretty familiar with this, but there's a problem.
The problem with this, is that it means artists have to work first in 3D and then in 2D.

And we felt this wasn't playing to their strengths, nor was it the best way to get the best models.
When we model something, we're really representing an object's attributes (position, colour etc) at various sample points on it's surface.
So when modelling in XSI (or whatever) we are sampling at a fairy low frequency, just at vertex locations.
Then, when we begin drawing the texture map, we jump from the vertex granularity to micro facets in one leap. Clearly this is missing lots of bands.
Moving from 3D to image space means the artists jump from thinking about the shape of an object, to thinking about very fine surface details (such as "lets apply a wood grain" here) in one step.
We felt this could easily result in models which look like triangle meshes which have textures plastered onto them,
rather than a good representation of the geometry you're trying to model with details at all frequencies.

Character Modelling

- Character artists typically have more freedom
  - Tend to get bigger budgets for normal maps etc
- Using mesh sculpting/painting tools:
  - Zbrush / Mudbox / Silo
  - Paint displacements, colours directly onto vertices
  - Subdivide mesh to get more detail
  - Incremental - all frequency bands get coverage
  - Artists can concentrate on what they are modelling not how to model it

By contrast, character artists have more freedom to overcome this.
Since we usually allocate relatively large budgets to character models, sculpting tools become viable.
I'm sure everyone here is familiar with zbrush, mudbox or silo.

Here, the artists can work at any level of detail they wish, and can paint directly onto vertices.
Normal and albedo maps are generally derived from the high polygon model, rather than painted independantly.

And when artists are modelling, if they want to increase the resolution they need for painting, they just subdivide the geometry.

This is a more incremental approach to modelling, and results in artists painting, without thinking about it, at all frequencies.

What we especially liked about this was that artists can concentrate on what they are modelling, not how to model it, and we tended to find really high quality models would result.

Benefits

- Would be nice to sculpt and paint whole world
  - Artist centric approach
  - Helps us deliver on art direction
  - Modelling entirely in 3D at all detail levels
- Allow removal of artist-unfriendly stages
  - Remove concept of texture mapping altogether from authoring

So we decided this would be great for environments if we could get it to work, as Sculpting is a very freeform approach and very artist centric.

Since our world required a painted look we felt that to paint and sculpt our whole world would allow us to deliver on our art ambitions.

And since we'd be painting directly onto vertices, we wouldn't need to explicitly texture map anything either.

Which sounded great.

So, There Must Be A Catch, Right?

- Yes.
- Current crop of sculpting tools only really well suited for discrete models
  - Limited to editing ~10 million quad models
- Does not scale up well for worlds
  - Environment datasets are huge
    - At ~5mm resolution we need nearly 10 billion quads per level(!)
  - Need to support multiple users and version control

So, it seems pretty obvious that *we* should be using a sculpting tool – so why isn't everyone already?
Well, there's a problem with scale.
The current crop of sculpting tools are great for discrete models, but start to hit a wall when you hit very large polygon counts.

For example Zbrush works brilliantly for characters, but as soon as you start to push beyond 10 million polygons or so, things really start to slow down.
To give you an idea of scale, a typical level in our game (if we worked at roughly a 5mm texel resolution) would require around 10 billion quads(!)
There's also workflow concerns –for large environments, we'd need multiple users to work on assets simultaneously and we'd like version control too.

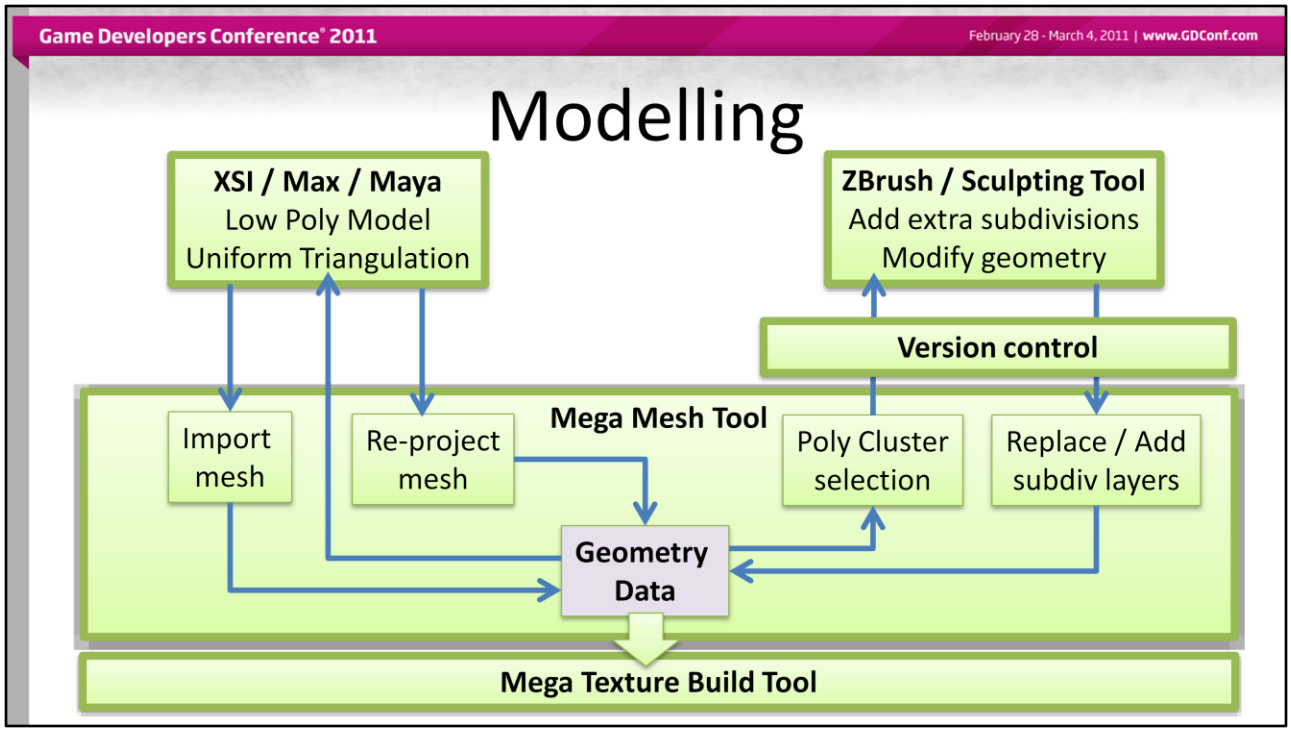
Our Solution

- Mega Mesh Tool
  - Manages models with 10's of billions of polys
  - Supports multiple users
  - Sits between DCC tool, VC & sculpting tool
  - Windowed editing at variable detail levels
  - Responsible for coordinating build process

So we've developed a tool which allows us to overcome these problems enabling us to manage multi billion poly models.
It's the core of our pipeline and is called the mega mesh tool.
The tool sits between the content creation tool producing the rough low poly model and the sculpting package
and enables multiple users to work simultaneously through integration with version control.

We work around the 10 million poly limit by allowing the user to select regions of the world using a window,
So with a large selection window, you edit the world at a lower resolution
and with a smaller window you have finer detail available.

And finally the tool ties the pipeline together by coordinating the build process.



Let's talk about the work flow we've developed here and how the mega mesh tool fits into it.
<click> So typically we begin by importing low poly mesh into the mega mesh tool, these are the meshes from our white box world (which we've used to prototype gameplay).
Incidentally the shape of the mesh, other than the topology, isn't actually that important at this stage, as its going to change once we start sculpting it.
<click>
Since meshes can be of arbitrary complexity and size, and since we don't want to perform version control at the polygon level, the artists group polys together into clusters.
We create files containing each group of polys and manage them in our version control software.
So, typically you'd select a few of these clusters around an area you wish to work in, and the tool will check out the corresponding files and collapse the geometry into a flat mesh for sculpting in zbrush.
Next, the artists do their painting or sculpting and adding extra subdivision layers where necessary.
<click> When we are done painting or sculpting the mesh, we can re-import the model back into the tool and propagate changes back onto the mega mesh model.
<click>
If we need to modify a meshes topology, we have another path which lets us do this.
I'll discuss this step in a little more detail later.
<click> Once we're happy with our changes we build the geometry and deploy to the game.

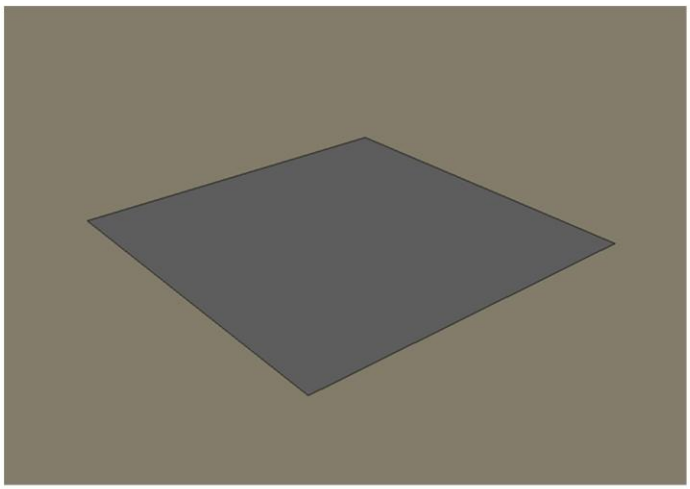
Hierarchical Modelling

- Impractical to store such large datasets in physical memory
- Geometry data is stored hierarchically
  - Successive subdivision levels stored as deltas
  - Levels loaded on demand

One of the problems with absolutely massive data sets such as ours is that it's impractical to store the whole model in physical memory at once.
Instead we store the data hierarchically, with successive subdivision levels stored as delta layers, which are loaded or unloaded on demand.

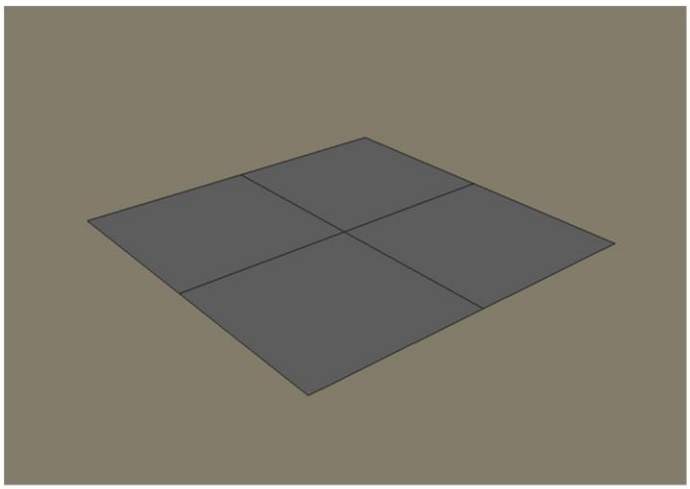
So...

Subdivision – Level 0



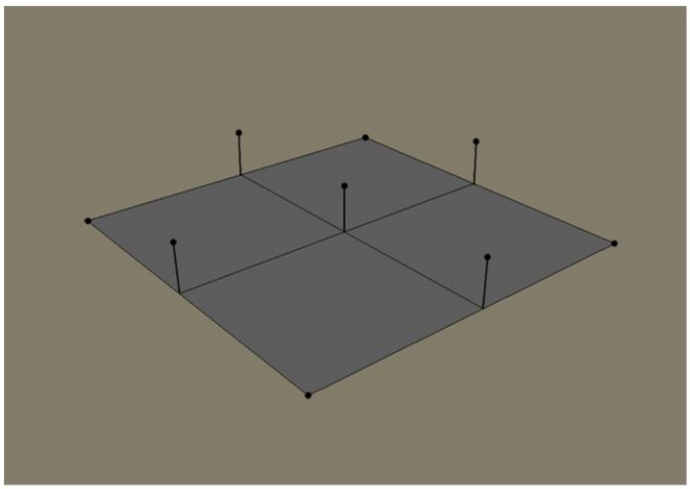
Here we have a cluster, at level 0,
In practice a cluster would contain probably hundreds or even thousands of quads.
But for illustration, this one consists of a single quad.

Subdivision – Level 1



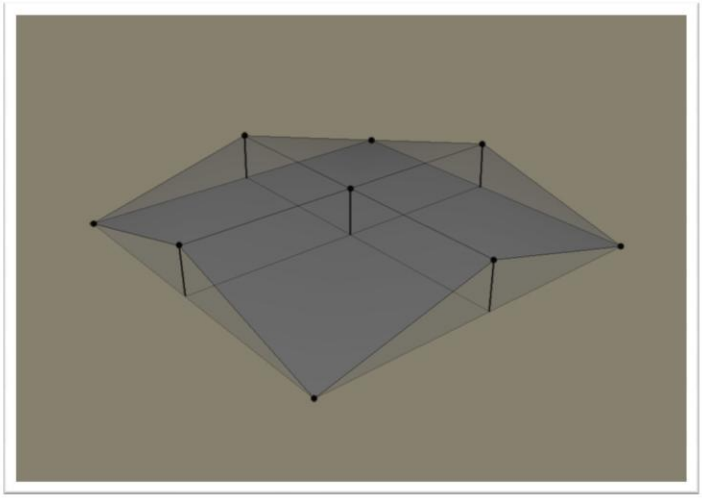
When we load in level 1,
we divide each quad in the current layer into four new quads

Subdivision – Vertex Displacement



and offset the 5 new vertices by some delta transforms
Note, the original 4 vertices retain their position

Subdivision – Level 1 Final

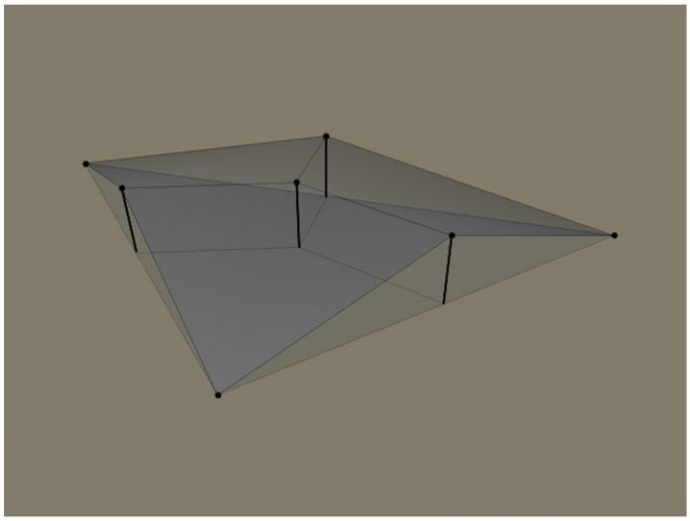


The deltas are stored relative to the coordinate frame of the parent quad.

So, if the 4 vertices in the base were rotated or translated, the 5 child vertices would move with it.

This is a really useful property that I'll get onto later.

Triangle Subdivided Into Quads



And here you can see triangle subdivision.

Note how this produces three new quads and 4 new vertices
so whilst we can import triangle models into the mega mesh tool for the low poly cage,
all successive subdivisions from 1 onwards operate on quads.

This was done just to simplify the datastructures.

Benefits of Hierarchical Storage

- Can edit regions of world at low subdivision level retaining high frequency details
  - Can change macro topography of terrain without losing all details
  - Just modify lower subdivision layers
  - Can make large scale tonal adjustments

Sadly, It's not a perfect world and at Lionhead we have what is known as an "organic" development model.
Which is shorthand for saying we make lots of last minute changes.

And this is where having a hierarchical storage model really helps us...

Storing the deltas relative to the coordinate frame of the parent quad means that we can modify regions of the world at a coarse resolution, whilst still retaining high frequency details.

So one thing you can do is slide out just the lower few subdivision levels for a part of the world.
The exported mesh could cover a whole zone, for instance, but at a low level of detail.
You then sculpt this low resolution model, changing the macro surface shape or colours.
And when you're finished, you just slide the layers back in.
Because the high resolution geometry is stored as deltas, they are effectively transformed relative to it.
So, if you've painted some nice details such as cracks or pebbles or something, then these details will be preserved and effectively superimposed over the modified geometry,
You can do this for any channel too, so it works just as well for colour correction.

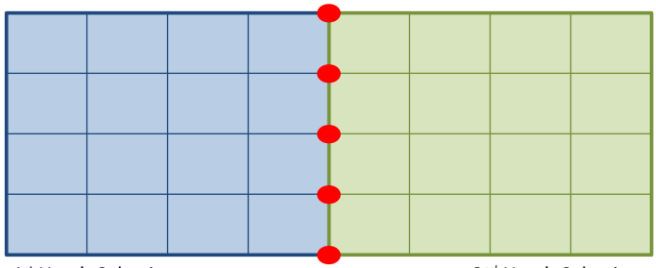
Topological Changes

- Frequently necessary to re-topologize geometry.
  - Moving objects around
  - Adding new features
- Alignment tool used for matching source and target geometries
  - Reimport new triangulation from DCC tool to MM tool
  - Select pair of matching faces from source and target
- Re-project previous highest resolution subdivision level onto new model

Another typical change we needed to support was
physically moving parts of the scenery around or add new topological features.
Given that a lot of our world is welded together, this is potentially a major problem.
We address this with our alignment tool.
If we are updating a mesh, say we've moved a tree, for example, we don't want to lose all our lovely tree trunk modelling work.
So, when we reimport a mesh, we select two correlating faces from the original and updated geometry – to give us a transformation matrix, and reproject the geometry from the original onto the modified geometry.
Since much of our geometry is welded together, this technique isn't 100% perfect as there will bound to be new geometry introduced which wasn't in the original model, but works pretty well for the geometry which was relocated and can mitigate a lot of asset reworking.

Multiple Users and Seams

- Multiple users can work simultaneously on adjacent clusters
  - Reject modifications to vertices on shared edges
  - Must check out both adjacent clusters to guarantee smooth shared edge

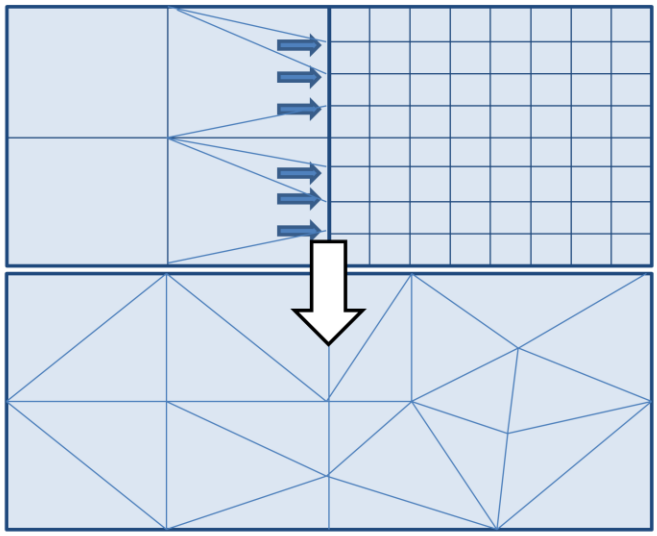


Another problem is encountered when two people are working simultaneously on two adjacent clusters.
Since the edge vertices are shared, we don't want to reimport edge vertices as there will be confliciting versions of the vertex locations, and we didn't want to write a merge tool.

In the diagram we have two clusters checked out to different people, the vertices in red will retain their original values when re-imported.

If you want to work up any edge vertices, to make a smooth transition between two clusters – you have to check out adjacent clusters and work on the shared edge in the sculpting tool.

Generating the Runtime Mesh



- Don't want seams between clusters stored at different resolutions
  - High order vertices on edges with no matching pair on other cluster are clamped to interpolated value from lower resolutions and quads re-tesselated
  - Optimised using modified version of Hughes Hoppes' mesh optimisation

So, how do we generate the final game mesh and how we prevent cracks or seams between clusters in it?
Well, since each cluster can be stored at different resolutions, and to potential cover cracks,
we re-triangulate the low resolution quads.
So we don't introduce nasty ridges,
if an adjacent mesh is at a higher resolution,
the high order edge vertices (like the ones with arrows on in the diagram)
are snapped to match interpolated positions on the edges of the lower resolution mesh.
We then take this sealed mesh and optimise it using a modified version of Hughes Hoppes' poly reduction scheme.

Game Representation

- We bake albedo, specular luminance and specular exponent onto texture charts
  - Project onto low poly mesh derived from high subdiv data
  - Normals / Ambient occlusion calculated algorithmically
- UV unwrapping done automatically
  - Chart size based on actual mip usage data from game engine not spatial dimensions
  - No more 1k x 1k eyeball textures

Next we bake out albedo, specular luminance and specular exponent terms onto texture charts.
Each channel is managed as a separate hierarchy in the tool and can be edited independently.
Normals and AO terms are derived algorithmically from the high poly geometry and baked out too.
So 5 channels.
As should be apparent by now, at no point in the pipeline do the artists had to bother with unwrapping their models as we've not needed to do any work in image space.
We found lifting this constraint saved the artists a load of time and removed a stage of their work which did not really map well onto their primary skill set.

So we have to generate UV charts.
When we do this, we govern the size of the chart not by it's spatial dimensions or it's entropy as you do with lightmaps,
but instead by the maximum resolution we'll ever need when rendering. If we've painted stacks of detail somewhere but our runtime analysis says we don't need it, we just don't bake it out.
So we only ever store on DVD the maximum mip levels we'll potentially ever see.
So no more 1k x 1k eyeball textures.

Sparse Virtual Textures

- High memory footprint with so many unique channels
  - Need good streaming / rendering system
- Virtual texturing good candidate
  - Virtualises texture space
- Sean Barrett's GDC paper:
  http://www.silverspaceship.com/src/svt/

Even for a moderate sized world, the amount of texture data generated by this system is huge, so clearly we need a good way of streaming and rendering it.

Around 3 years ago, when I was first thinking of using sculpting tools to model worlds, I was toying with this problem.
During this time, I happened to attend a lecture by Sean Barrett, here at GDC, where he explained his algorithm for virtualising texture space.
It was a great talk and provided me with a very elegant way of simplifying the problem of managing large amounts of texture data.

I've attached a link to Sean's paper here, it's a really great talk and covers lots of implementation details.

I'm going to spend a little time now talking about some of the hurdles we've had to overcome getting it working in our engine.

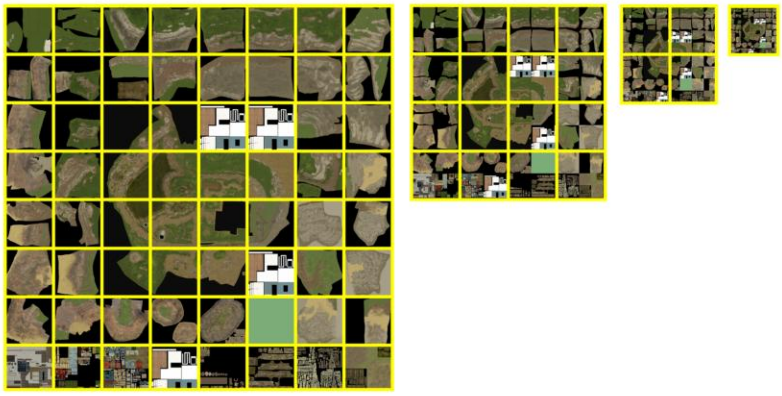
Virtual Textures

An early Mega Texture page from Milo & Kate

First though, here's a brief recap on virtual textures.

Conceptually, the scheme is very similar to virtual memory on an MMU.
We start off by using a global UV parameterisation for our world
So everything gets atlased into a single massive texture.
We call this a "very big" or sometimes a "mega" texture
This parameterization is completely transparent to the artists.

Tiles



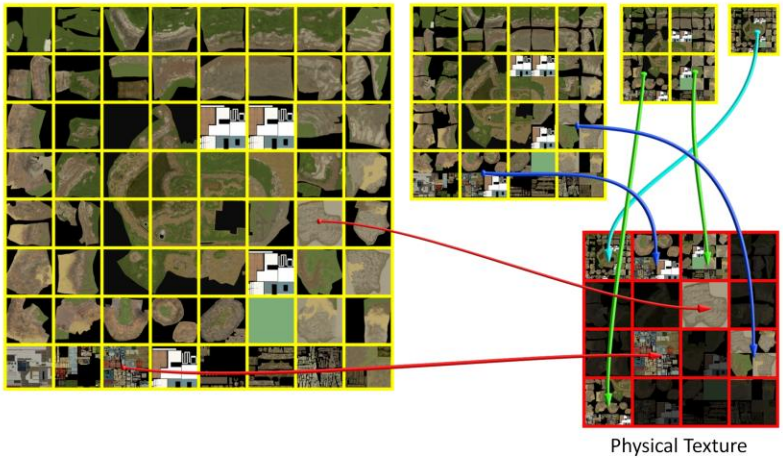
Mega Texture and mipmaps split into tiles

Since this is too big for the GPU to render from directly, we virtualise it by splitting the texture space into square pages or tiles.

Since we want to support mip mapping,
we segment each mip level of the virtual texture into tiles too.

As you can see here,
the tiles are always the same physical size irrespective of the mip level.
So lower mip levels have fewer tiles covering them as you can see here
The lowest mip level is the dimension of a single tile

Physical Memory



For any given viewpoint, we perform some analysis on the scene and decide which tiles we need.
We then populate a smaller texture with these tiles and it's this we use for rendering.
To continue the virtual memory analogy, this texture can be considered our physical memory
Since we need to know how to map from our virtual address space to the physical one,
we maintain a virtual address lookup table which maps virtual tile indices to physical tile indices.
These are illustrated by the arrows in the slide.

In some circumstances, we may not have a tile ready in physical memory to copy over.
If, for example, it hasn't been streamed in from disk yet
So, we temporarily point to a tile at a lower mip level instead.
Also if a particular tile isn't deemed to be required, we point to whatever mip we have available.
This allows us to have soft page faults
– so instead of throwing an exception or stalling while we load the appropriate tile, we always have something to render, albeit at a sub optimal resolution.
Address translation happens in the pixel shader, from virtual to physical address.
Having a single virtual texture space for all our assets makes life really easy, and simplifies a lot of our material and shader code.

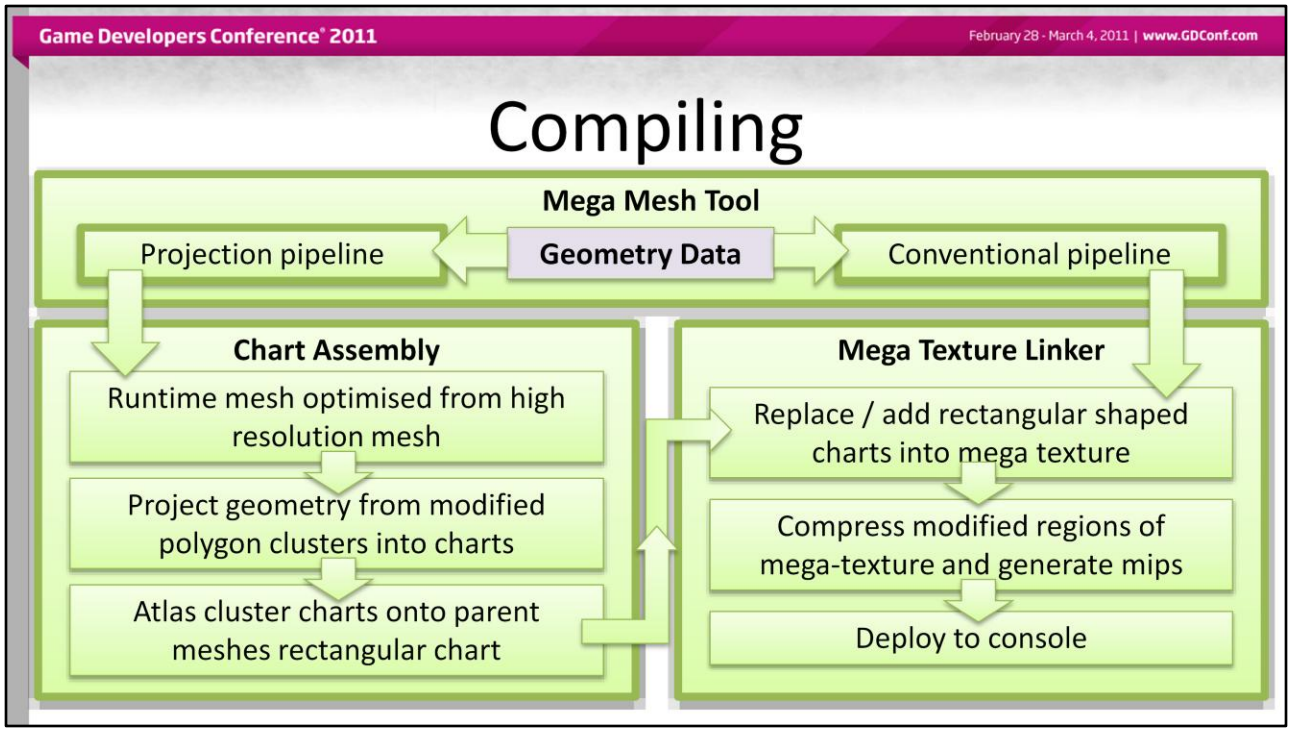
Virtual Texture Details

- Multiple virtual textures in an array
  - 16 x 32k x 32k address space
  - 128 x 128 tiles
  - 8:8 bit UV tile addresses
  - Some virtual textures for characters / props
  - The rest to separate sections of the world
  - Can unbind unused ones at level boundaries
  - Allows continuous streaming world
  - Physical pages at 2048 x 4096 (28mb)

For our implementation, instead of a single massive virtual texture, we found it useful to use an array of smaller ones.
So, we have an array of 16 32k x 32k textures active at once.
We use 128x128 pixel tiles, we found this a good sweet spot for us.
Any smaller than this and the management overhead and virtual address lookup texture became too big,
and much bigger and there's too much redundancy in the physical page.
It also means that we can use 8 bit addresses for the tile UVs, which is always a useful property.

Some texture slices need to be permanently resident, for props and characters, for example, but others need only be active if we are near the geometry they are mapped to.
So, when we cross a zone boundary, for example, we can unbind the virtual texture slice for that region and bind another one to that slot.
This gives us the ability to stream continuously without any load points.

We also experimented with different sizes for the physical pages for each channel and settled on 2048 x 4096.
Much smaller than this, we found we got quite a lot of thrashing particularly on complex scenes, and particularly if the camera is rotating rapidly.
Ideally, as always with caches, the bigger the better, but this performed pretty well, and to double it to 4k x 4k would cost us another 28meg which we just couldn't afford.



So let's talk about how we compile our mega textures
A key goal for our tools is to reduce iteration times and when we are dealing with billions of polygons and huge textures, this is potentially a challenge. We address this by rebuilding as little as possible when iterating geometry.
Consequently, we employed a two tier approach and incremental builds.
The first stage is chart assembly and the second stage is the mega texture linker.
<click>
The first stage generates optimised meshes from any modified polygon groups.
It then creates a UV chart for each and projects the high res geometry onto the runtime models.
These are then arranged into a rectangular chart for the mesh they belong to.
<click>
The linker then takes all the new or modified rectangular charts and reallocates space for them in the megatexture.
We have a kind of 2D spatial allocator for this.
We then compress any modified regions of the megatexture and generate mip maps,
Before deploying to the console for testing

We got turnaround on this to under 2 minutes for a smallish change, so it's a relatively quick to iterate.
<click>
Now early on, we made a decision to allow for the sculpting pipeline to be entirely circumvented.
This may seem strange, but in some cases it's just not profitable to use sculpting tools.
For example, when modelling foliage, artists may want fine grain control over individual triangles.
In this case we found it's actually just better to let them model it traditionally in XSI.
However, we'd still like to unify everything into our runtime virtual-texture system.
So, we simply treat the artists' custom UV sets as a rectangular mesh chart, pass it to the linker, and just copy the hand drawn textures into the mega-texture.

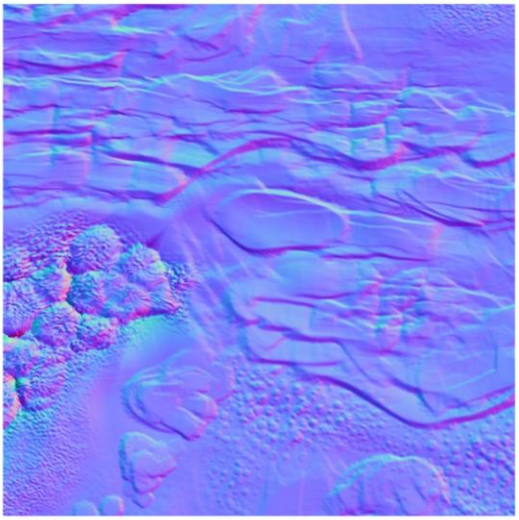
Compression

- DXT compression not sufficient
  - Need normal, albedo, AO and specular channels
  - Virtual texture sizes too big
    - 32k x 32k with mips ~ 2.5Gb
    - 16 x 32k x 32k with mips ~ 40 gig
  - Also, space for other things; audio, anims etc.
  - Throughput of DVD not sufficient either

Ok, a few quick words on compression now.
When I first tried this scheme I used DXT compression for the texture storage on disk
It quickly became apparent this wasn't going to be sufficient
to give you an example, we have normal, albedo, AO, and 2 specular channels,
For a single 32k x 32k virtual texture, this would already take 2.5 gig which is bad.
And for 16 slices, that's 40gig!
– not forgetting that we also have audio and animation and other stuff like that to fit on DVD.
The last nail in the coffin is DVD throughput, which is not adequate for this density of traffic anyway.
So it was clear we needed better compression that we get out of the box.

Typical Normal Map

- Heterogeneous frequency distribution
- Good for entropy encoding codecs such as Jpeg or HDPhoto



It's worth pointing out here that we're not storing typical normal maps or textures here, as it's all baked from unique geometry.

To illustrate this, here's a typical normal map from our demo
As you can see here, our textures have quite a heterogeneous distribution of frequencies

Meaning that we have some smooth sections and some areas of fine detail

Codecs which use entropy encoding, such as JPEG work well on mixed frequency images such as these.
They offer variable compression ratios, so we can balance quality against storage and offer good savings above and beyond that which we can get with the GPU fixed bitrate formats.

Our Compression

- Proprietary lossy compression algorithm
  - Jpeg output was too blocky
  - HDPhoto gives high compression ratios
    - Dithering artefacts make it unsuitable for normal maps
  - Ours based on PTC scheme by Rico Malvar (MSR)
    - Comparable decompression speed / ratios to HDPhoto
    - http://research.microsoft.com/pubs/101991/PTC.pdf
  - Normal compression up to 40:1 vs 3:1 DXN
  - Albedo compression up to 60:1 vs 6:1 DXT

So with this in mind, I wrote a JPEG decompressor.
To my disappointment though, the blocking artefacts were pretty obvious and pretty nasty at higher compression ratios, and they looked particularly bad on normal maps.

So then I tried a HD photo codec.

This uses a different transform function which eliminates theses blocking artefacts, instead making the image more blurry at higher compression ratios.

And this was working great, until we noticed it also introduced some subtle dither patterns which were fine on the albedo maps, but were very distracting for normals when we added specular highlights to our materials, as it gave everything a corrugated feel.

On a bit of a punt, we decided to try the scheme which the HD Photo was derived from, a codec called PTC, and found it gave comparable quality at similar compression ratios to HD photo, but crucially eliminated these noise artefacts.

As you'd expect the compression ratio varies a lot based on the entropy of the input images – but as I demonstrated in a previous slide, our textures suit this type of compression very well.
So as you can see, typical compression rates we see for our textures, compare very favourably with the GPU based alternatives.

Albedo Channel

- YCoCg colour space for Albedo
  - Trick used by Jpeg compressors
    - Y channel at full scale, CoCg at ½ scale
  - Very fast transcode to DXN textures
  - Decode direct to DXN blocks
    - No intermediate stages
  - Transformed to RGB in pixel shader

When we compress the albedo channel, we first use a trick commonly employed by Jpeg codecs.
It's based on the observation that the eye is far more receptive to changes in luminance than colour,
so we store the chroma channels at half resolution, which saves us 50% before we've even begun to do anything remotely clever.
So, to begin, we convert to another colour space (called YCoCg) – which is luminance, orange and green. Then we crunch the orange and green channels down to half size.
This is entropy encoded using the PTC encoder.
At decompression time we keep everything in Y Co Cg space, and transcode directly to two DXN textures which we render from,
one full size for the luminance and one half size for the chrominance.
Now, instead of encoding to an flat image then transcoding that, we do it one 4x4 pixel block at a time as we are decompressing the data, so we get very good cache performance
Then finally, conversion to RGB is done n the pixel shader, which is just 3 dot products, so it's very quick.

Albedo Compression

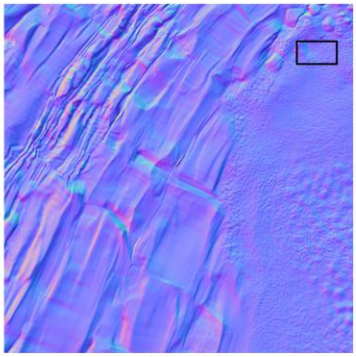
Original 769k

Compressed 37k

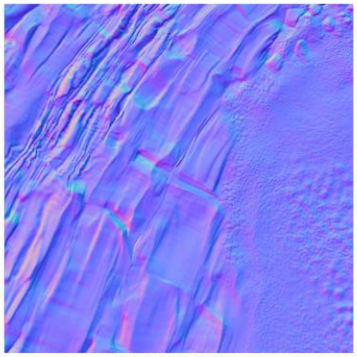
So, even though keeping this as one full size texture and one half size textures costs us an additional 25% extra storage over DXT1, on balance, its a pretty big win for us because the reduction in transcoding errors we get from not having to encode a DXT1 texture from RGB space, is really significant.

Here's an example of our the compression.
I feel it's pretty good given the ratios.

Normal Compression



Original 736k



Compressed 43k

Some artefacts on high frequency details, but generally, good feature preservation.

And here's a detail on the normal map compression, as you can see there are some smoothing artefacts on the very high frequency details, but generally there's pretty good feature preservation.

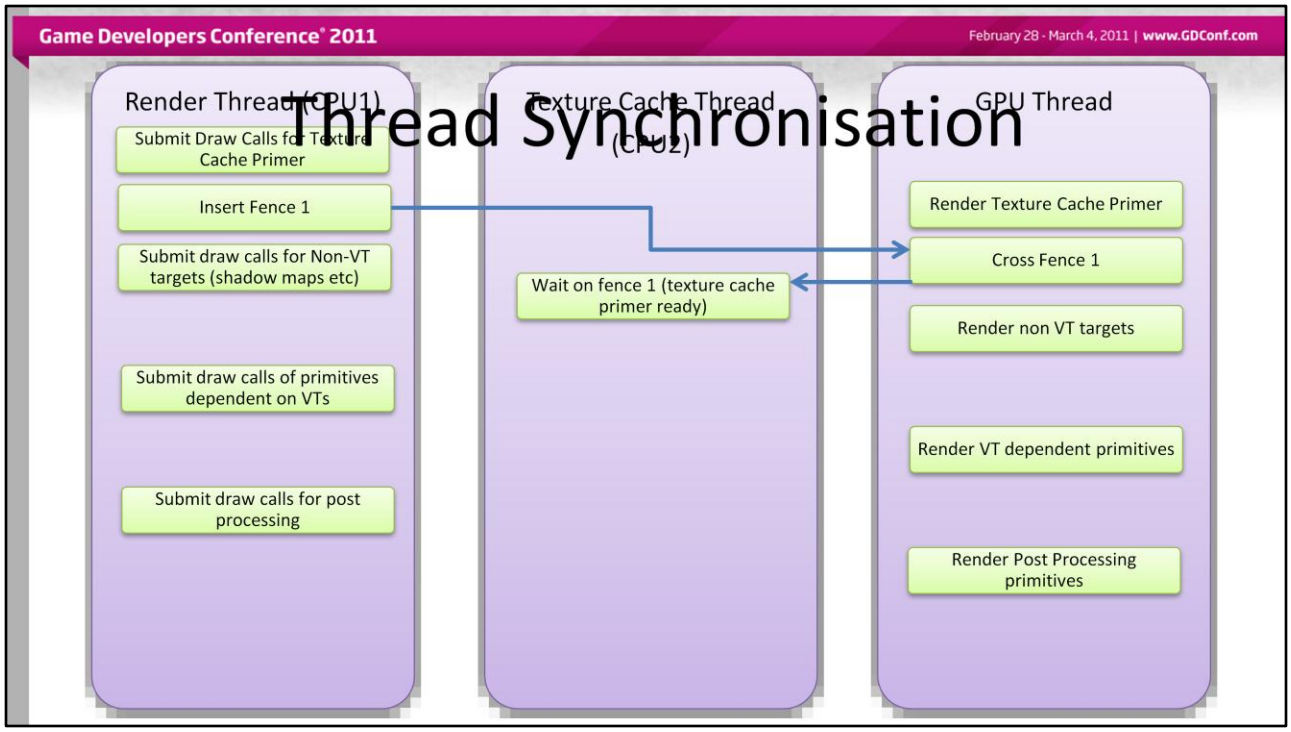
Frame Analysis

So, in practice, how do we know which tiles we will need for a given frame?
There are a number of solutions to this problem
The basis of ours is very simple, but works well in most scenarios

Texture Cache Primer

At the start of the frame we render the scene to a special render target called the texture cache primer – which we can see above for the last frame
For every fragment, we output the tile uv, slice and mip level it will reference.
Then, synchronised to the GPU thread is a CPU worker thread which waits for the target to be rendered and then analyses it.
From this we can work out any page faults we will encounter and we fill the physical page accordingly.

As it happens, this is really useful information,
not just for populating the virtual texture,
but for compiling information for the pipeline as a whole.
Since for any given viewport we know exactly which tiles are used, we can build up a history of global tile usage, which we then feed back to the mega mesh builder to optimise chart resolution.
We also use this information in the mega mesh tool to indicate where more detail would be beneficial or where surplus exists.
We highlight regions in red where we need more subdivision
and green where there is already enough.
This is particularly helpful when painting vistas or backdrops where it's often really hard to estimate appropriate texture resolution.

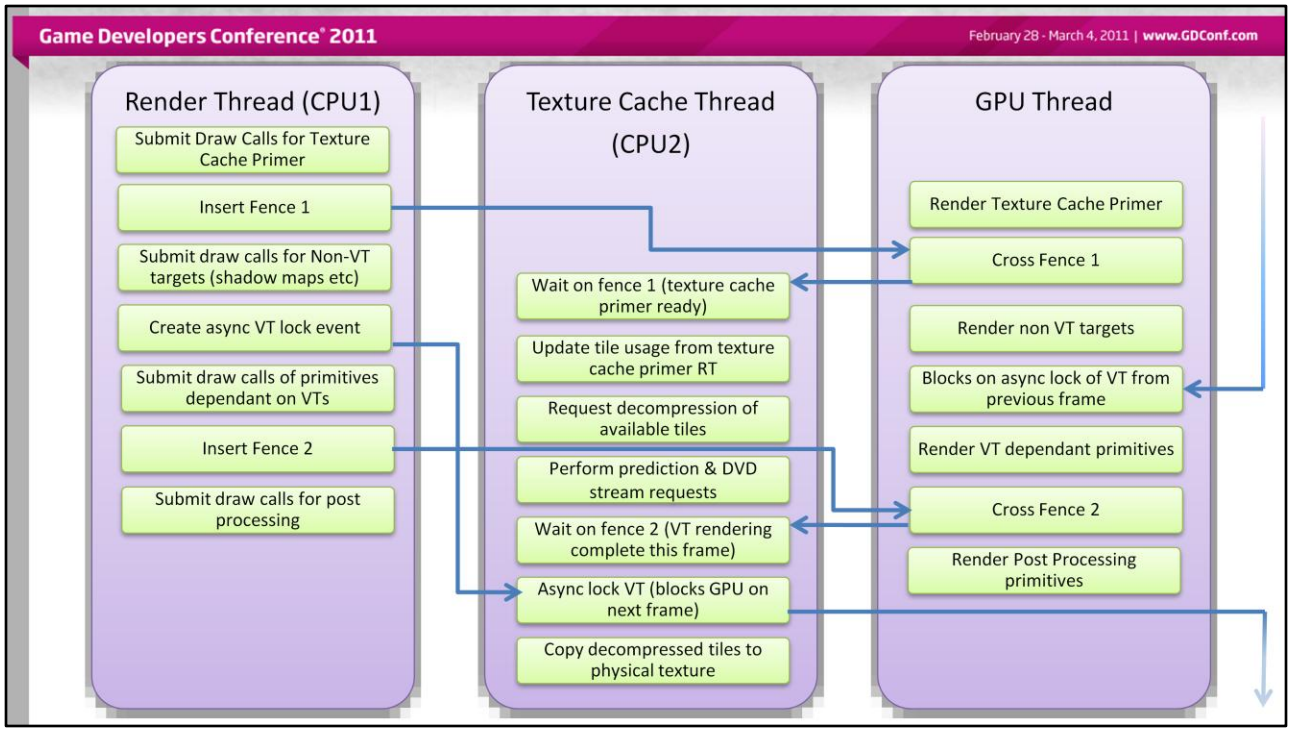


To save memory and reduce latency, we don't double buffer anything, which means we have to guarantee some synchronisation between various threads, which I'll explain a little bit about.

This scheme gives us a best case of 1 frame latency between a tile being requested and it becoming available for rendering.
So, There are 3 primary threads involved in updating the texture cache
<click>
The CPU Render thread which submits draw calls,
The GPU Thread which services these
And the texture cache thread. This is responsible for maintaining virtual texture.
So lets look at how they communicate with each other.
<click>
First we split our rendering into 4 stages as you can see here;
The first thing we do is render the texture cache primer at the start of the frame

Then we draw as many render targets as possible which don't rely on the virtual texture, such as shadow maps, z prepass etc.
We're trying here to delay the rendering of anything which is dependant on the virtual texture (such as the normal buffer) as late in the frame as possible, to give the other thread as much time as possible to update it
And then we finish up with any post processing.
<click>
And here we have the GPU rendering each stage.

To synchronise the texture cache with the GPU thread, we use fences on the xbox.
<click>



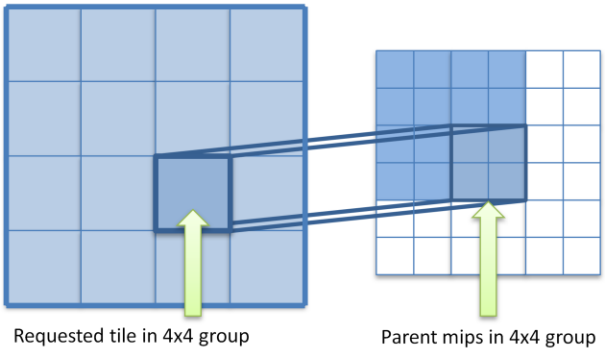
So we wait till the GPU has finished rendering the texture cache primer
<click>
And now we can scan it and build up a list of tile dependencies.
Each time we mark a tile, we increment a counter for it and it's mip chain.
Then we kick off a bunch of decompression requests for missing tiles and
tell our so-called L2 cache to perform prediction for streaming.
Finally, we want to update the virtual texture, but we must wait till the GPU is finished rendering from it, so we use another fence

<click>
Only then can we update it

<click>
Now, if the GPU is running ahead, it may attempt to access the virtual texture in the next frame before we are finished updating it.
So, I use an asynchronous lock to guarantee thread safety with the GPU.

<click>

L2 cache



- Services tile requests (or page faults) from L1 cache
- Don't want hundreds of small requests to disk
  - If tile requested, can assume neighbours needed soon too
  - And probably the higher resolution mip level too
  - Group tiles in 4x4 blocks
  - Stream this and parent group together

So as I mentioned, the L2 cache is responsible for servicing requests for tiles (or page faults) from the L1 cache
and for driving the streaming system.

As we know, streaming systems tend to work best when they are handling larger chunks, so ideally we don't want to be handling hundreds of incoherent tile requests a frame.

Now, since most of our tiles are arranged spatially in the virtual texture page, it's reasonable to assume that if we require a tile, then it's neighbours may be requested in the near future.

Similarly, it 's reasonable to assume that if a tile is increasing in resolution, it will continue to do so.

With this in mind, we group tiles in blocks of 4x4.
For each tile we want, if's not already loaded into the L2 cache, we request the tile block it's contained in, as well as one for tile group containing it's parent mip level.

L2 Cache (2)

- All tile groups stored compressed
  - Decompressed to GPU format on demand
- Prioritise tile decompression by:
  - Number of pixels referencing tile
- Prioritise L2 fetches based on:
  - Mip level – lower mip levels get priority

All tile groups in the L2 cache are stored in our PTC compressed format, and then decompressed on demand to the GPU friendly format, before being copied to the physical texture page.

Because there's a finite number of tiles we can actually decompress per frame, our texture decompression thread runs a priority queue.

Requests are prioritized in order of the number of pixels that require that tile.
As I mentioned, we calculated this when we scanned the texture cache primer frame buffer.
We prioritise L2 fetches from the DVD on the mip level – for the soft page faults to work, we always need the lower mips before higher ones, so we fetch these first.

Hint Frames

- Using current camera position is “just too late”
- Solution: “Hint Frames”
- Pre-empts possible future L1/L2 cache misses
- Periodically we render to tile usage render target
- Fallback - explicitly preload tiles from the Virtual Texture to L2 cache

One of the problems inherent in using the camera position for rendering the texture cache primer is that it’s just too late,
There’s a load of potential fail cases when we can surprise the system, for example
If the camera suddenly turns 180 degrees...Or if a door opens.....Or if an object appears on screen.....Or if we teleport somewhere.

So we try to mitigate some of the unexpected camera movement using hint frames,
which help us pre-empt possible future cache misses.
So what we do is, as we move around the world, we periodically render not from the cameras point of view but from another.
The game passes these to the rendering engine as hints.
Examples of hints would be: predicted future camera positions,
or the current camera position facing backwards, for example.
We can also turn off dynamic occluders too, such as doors, for a frame.
One nice trick is to drop points of interest around the world,
So for example, we may know up front that a picture on a desk is something that the player is going to interact with and we know it has a lot of extra surface detail
So we drop a interest point near it, And as the camera moves near these interest points, we automatically add them to the hint list,
and the engine will render from this location during the hint pass.

Finally, the last tool we have is to explicitly preload up regions of virtual texture the L2 cache. This is most useful for objects that pop into existence which cannot be predicted by simply moving the camera to a new location.

Virtual Texturing Gotchas

- Transparency
  - Render dither pattern to Texture Cache Primer
  - Increases load on physical memory
- Physical texture over population
  - Automatically lowers/raises mip LOD Bias
  - Auto balancing; adapts to smaller physical textures
- Texture Popping
  - Adding bias texture in shader adds another level of indirection
  - CPU solution working but it's pretty heavyweight

Some gotchas now.
To handle transparency, we use a 2x2 dither pattern to render out to the texture cache primer, this gives us up to 4 levels of overdraw.
However, transparency is not great for us as more levels of transparency means more tile utilisation, and so it increases the load on the physical page.
This can lead to over population.
We do have a fix for this, though.
If the physical texture is starting to get full, we automatically lower the mip LOD bias when we render the texture cache primer.
This tricks the system into thinking that everything is at a slightly lower mip and allows the marginal pages to become free.
Then, once pressure lifts from the system the LOD bias reverts back giving us full resolution again.
Since this is auto balancing, you can give the system a smaller physical texture and it will adapt accordingly – although naturally you'll get a more blurry image as a result..
One issue we are still not happy with is texture popping –when a new tile becomes available at a higher resolution, it just pops in.
We tried adding an additional prelookup to the shader to use the mip mapping hardware to blend in the new tile, but this adds another level of indirection and just wasn't a cost we could afford.
We've got a solution working on the CPU to do the blending but it's pretty slow and pretty heavyweight, so we're still thinking of ideas for this.

Mega Meshes - Summary

- Artist centric approach to modelling
- Detail only where you need it
- Tolerant to late changes
- Maps onto current hardware
- Good prospect for future pipelines – displacement mapping or voxel-octrees.

So, in summary.

Mega meshes give a more artist centric approach to modelling
Detail only where you need it
It's pretty tolerant to late changes
The representation we have here maps really well onto surface parameterisations, and therefore current console hardware.
Ideally, I would have liked to have used displacement maps, but I never got it running quick enough on the xbox to be viable.
Recently, however, I've been doing some tests using displacement maps in direct X 11 and got some very exciting results here,
so I think the tech has good prospects for future hardware.
Even for voxel octrees, the pipeline we've developed is one way of addressing the modelling requirements.

Video

So, that's enough of me blathering on about how great it is,
now I'd like to show you a video of the rendering engine in action.

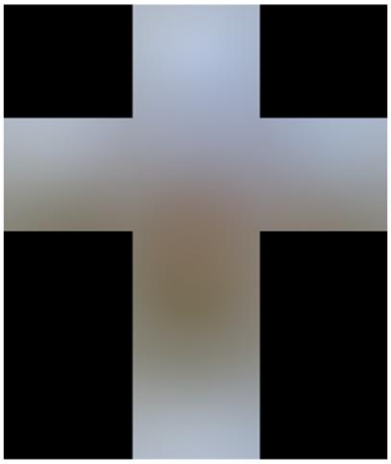
Lighting Requirements

- We needed a lighting solution that fits an organic world – simple direct lighting was just not enough
- What we wanted:
  - Soft and subtle lighting
  - Preserve details in shadows
  - Dynamically changing lighting
  - Avoid common artifacts
  - And, of course, low perfomance impact

Hi everyone, I'm Michał and I'll tell you more about the way we light this world.
We always knew that for such organic world, we would need a matching lighting solution. We couldn't just rely on the direct diffuse term and a constant ambient. So one day we listed all the requirements for our perfect lighting system
The most important thing was the look. We were after that subtle, soft feeling of a Pixar movies. Artists wanted to preserve all the details of our geometry, especially they wanted to see their normals in shadows. The gameplay team required the ability to change lighting, to show the passing of time.
I personally wanted the solution to be as correct as possible, without some distracting artifacts.
And of course we all wanted it to be fast – especially on the GPU side, as today this is a very limited resource.

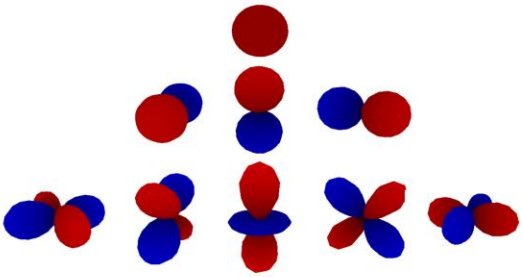
Irradiance Environment Maps

- For static, distant lighting we can precompute the diffuse response
- Don't need to use texture for storage – spherical harmonics work great!
- With SH we don't really need it to be stored pre-convolved – can do it in the shaders



We wanted to get some form of indirect lighting, and we felt that irradiance environment maps would be a good starting point.
The main idea is precompute the diffuse response of a material under a given lighting environement in every direction and store it in a cubemap. Now, the lighting is simply a lookup along the normal direction.
The cool thing is that this way we can model almost any light source we want. For example if we use an image of directly lit environemt as a source we will effectively get 1 bounce of indirect lighting this way.
What's important – irradiance environment maps can be very efficiently represented using spherical harmonics, as described by Ravi Ramamoorthi and Pat Hanrahan.

Spherical Harmonics



- Solution to Laplace's equations, analogic to Fourier transform but on sphere
- Linear
- Rotationaly invariant
- See Peter-Pike Sloan's or Robin Green's articles for details

Just a very quick reminder of what spherical harmonics are:
Basically it's just a way to encode a spherical function, analogous to a Fourier transform, so the function is described as a series of coefficients. Projection to SH is linear, rotationaly invariant and if you need further details see PPS or RG articles.

What We Tried

- Single SH environment map per zone:
  - No variance inside single zone
  - Problems on boundaries
- Precomputed SH lightmaps:
  - Costly to precompute
  - Static

So with that in mind we started our experiments.
The frst idea was to divide the world into a set of „zones” or „rooms” and prepare a single SH environment map for each one of them.
This was cheap and dynamic – because we could recreate those maps on the fly, but we were not getting any variation inside the zones and it was creating discontinuties on the zone boundaries
So we tried something more complex – we baked static lightmaps but containing SH coefficients, so we get directional information. This looked great, however was totally static, we couldn’t change anything. But we decided to investigate this option further.

SH Lightmaps

- A lightmap but storing a set of SH coefficients per texel
- Can be generated by rendering multiple cubemaps
- Let's refer to all points at which we render as **lighting probes**
- Can simulate multiple bounces – just re-iterate

Let's briefly describe how those lightmaps were generated.
We used exactly the same method as in the first approach, but instead of just one cubemap we rendered one for each texel on the lightmap. We call the points at which we render these cubemaps lighting probes.
This actually quite popular method to generate lightmaps offline – it's quite simple and can be easily extended to support multiple bounces of light.

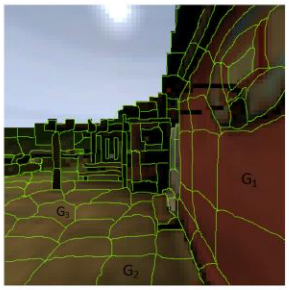
Realtime Global Illumination?

- Currently impractical so we have to sacrifice something
- We don't need fully dynamic solution but still want:
  - Changing illumination
  - Changing materials/colors
  - Proper occlusion for indirect light
  - It to be GPU friendly

But like I said, in this form it was impractical to use it in game. But since we really loved the look we started to think what we could do to make it run in real-time. This is the problem with real-time GI – at the moment you cannot afford everything, you have to sacrifice something. As we wanted to keep the quality maxed out we were prepared not to have fully dynamic solution, we were ok with some precomputed elements. But we definitely wanted to have the option to change the lighting, the materials, have proper occlusion for the indirect illumiunation.

From Envmap To SH Vector

- Why not just generate SH ceoffs directly?
- Describe envmap as a collection of groups of pixels sharing the same color, project each one separately and sum.
- Compute „unit" projection per group and multiply it by each component



$$G_1 = \{g_{1-0}, g_{1-1}, g_{1-2}, \ldots, g_{1-8}\}$$
$$G_2 = \{g_{2-0}, g_{2-1}, g_{2-2}, \ldots, g_{2-8}\}$$
$$G_3 = \{g_{3-0}, g_{3-1}, g_{3-2}, \ldots, g_{3-8}\}$$
$$\ldots$$
$$G_n = \{g_{n-0}, g_{n-1}, g_{n-2}, \ldots, g_{n-8}\}$$

$$G = \sum_i G_i$$

First let's make an important observation – the method renders a huge number of envmaps but we only need them to generate SH coefficients. If we could generate them directly, bypassing the rendering stage, the whole process should be much faster.
So our idea is to describe the envmap not as a collection of individual pixels, but rather as a collection of groups of pixel with the same color. Thanks to SH linearity we can project each such group separately and just add the resuting coefficients. Additionally we don't have to project each color channel on its own, but rather compute a „unit" projection for the whole group, and just multiply the result by each color component.

From Envmap To SH Vector (cont.)

- It's bounced diffuse lighting – just ignore high frequency details!
- Divide the world into patches of the same color, normal and small enough to ensure uniform lighting – **surface probes**
- Since the world is static we can precompute probes' visibility
- To generate SH representation we don't need to render envmap but just sum individual components

The good thing is that those input cubemaps are pretty blurry, and those groups of pixel actually exist. But even if they didn't were're calculating indirect lighting we dont really need any high freqency details anyway, we just care about the large-scale features.
To make things even more convinient, we actually divide our world into a number of small patches with uniform surface properties - we call them surface probes. And if a surface probe is rendered to a cube map it results in a group of pixels with the same color that we're after.
Now, if we assume the world is static we can precompute which surface probes are visible from each lighting probes and what are their SH projections.
So to get a irradiance environement map now we don't need to render it – we can just construct its SH representation directly by summing its individual compoents.

Our first approach

- Each surface probe is a disc, appoximated as a distant spherical light source
- Store binary visibility
- Quite inaccurate for surface probes near lighting probe
- A bit too costly to project each disc to SH

The first idea was to treat each surface probe as a disc, and store just the binary visiblity between the light probes and the surface probes. At runtime we were treating each disc as a spherical light source – as it's pretty easy to generate SH representation for this case.
And this was kind-of working but the results were a bit weird in some cases, especially when the surface probes were close to the lighting probes - moslty due to the fact that close up, a sphere is not really a good representation of a projected disc.
Additionally, it wasn't very fast – for each surface probe pair we had to regenerate its SH projection and for the numbers of probes we had it was just taking too much time

A Better Way

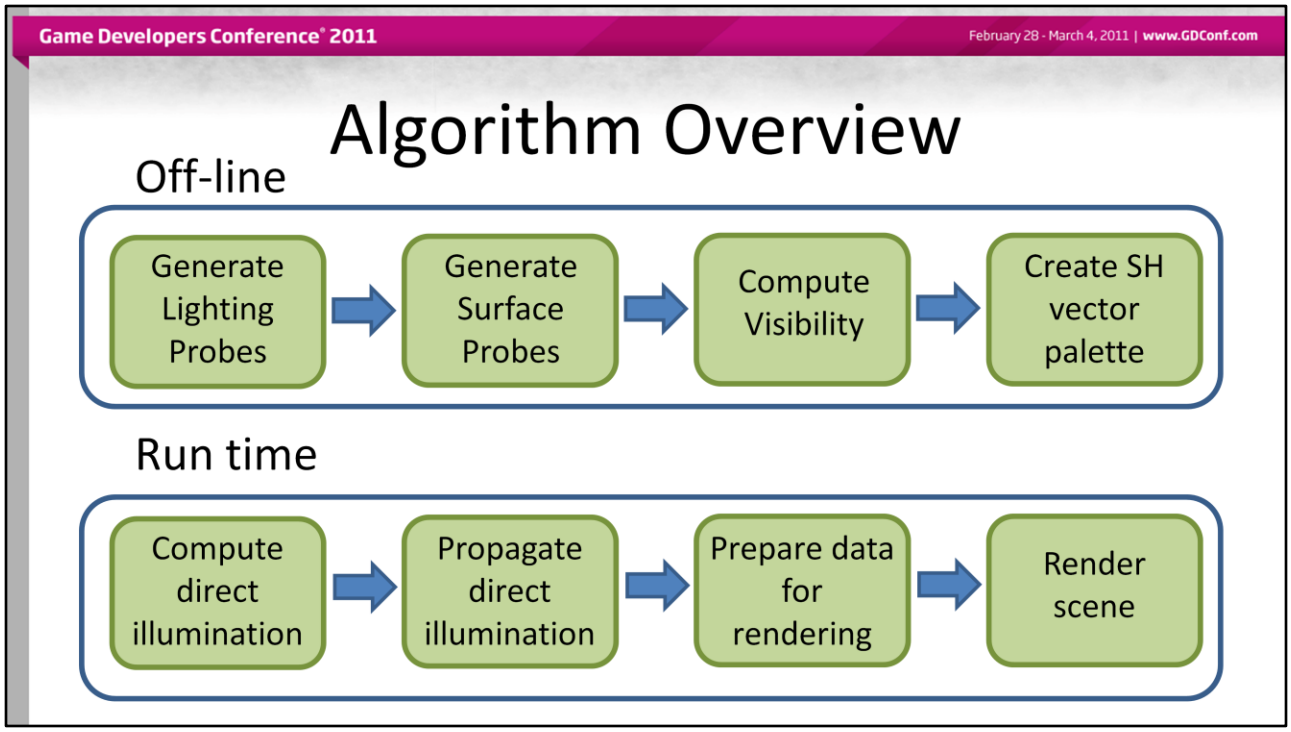
- If computing SH vectors is that costly why not just store them in the lookup table?
- Oh, and we can store anything! We're not bound to disc surface probes any longer!
- For each lighting probe we store a list of visible surface probes and indices to their coressponding SH projections

The final solution emerged from exeperiments with optimizing this process. I thought that there must be a fairly limited number of possible projections of disc visibility function onto SH – so maybe instead of generating them on the fly, we could just generate them offline and store in a lookup table. And then it struck me that with this approach we don't even need the surface probes to be discs – we can store any shape we want projected to SH in the lookup table and it will work equally well.
And now, things become even simpler – for each lighting probe we just store a list of visible surface probes, each with an index to its coressponding SH projection in the lookup table.

Palletizing SH Vectors

- Typical scene can generate a palette of over half million entries
- Reduce it to any desired size using some clustering algorithm
- We use k-means clustering executed on GPU (Compute Shaders)
- Average reconstruction error is 2-3%

To give you just an overview: if we take Milo's house and its surroundings and generate projections for all the lightprobes there it would give us about half a milion different SH projections. It's a lot.
To make this data set more resonable we shrink this list down to a desired size – 64k entries in our case, so that each reference fits into 2 bytes. We just palletize this set, just like a bitmap image – we use k-means clustering implemented on GPU, so it can efficiently deal with this massive dataset.
The reconstruction error caused by palettization is actually quite small ~around 2-3% on average - and totaly unnoticable in practice.

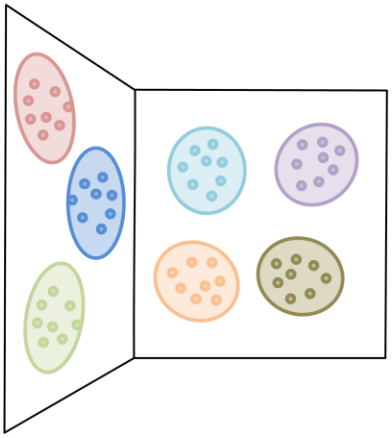


So before i move on to some implementation details let's briefly summarize:
- The algorithm is dividied into 2 stages: precomputation and run time
- In the offline stage we generate lighting and surface probes, calculate their visibility and create a palette of SH vectors
- At run time we calculate the direct illumination for the surface probes, then we propagate the light to lighting probes, and convert the results to some GPU-edible format and then render the scene with indirect lighting

Now, Let me describe some details of how we implemented this in our Megamesh pipeline.

Generating Surface Probes

- Surface probes generated by randomly generating points on geometry and recursively subdividing the point cloud
- Groups of surface probes can be marked with additional metadata to allow changes at run time



The first stage is generating surfaces probes.
Our method is pretty simple – we start by generating a number of points evenly distributed on all the meshes in the world. Then, we first group points that are lying on neighbouring triangles with similar normals, and subdivide those groups until they meet some predefined size criteria.
As at this point we have all the high resolution color data from Zbrush, we just use it to derive probe colors.
At this point we can also add some metadata to the surface probes. This way they can be for example groupped – and accessed at runtime to change their color or made to be emissive – which would effectively make them area light sources.

Generating Lighting Probes

- Two different ways of generating lighting probes:
  - Objects smaller then some threshold use a rectangular lattice of probes
  - Larger objects are uniquely unwrapped
- Together with lighting probes we create a **probemap** – metatexture holding index of a lighting probe for each texel
- A grid of probes created for dynamic objects

Generating lighting probes is a bit more complicated.
Since lighting probes data will be stored in lightmaps, our idea was to unwrap the geometry, put it into texture and create a lighting probe for each texel of such atlas.
Conventional unwrapping algorithms are a bit inefficient with the texel density we use, and create many discontinuieties, so we use a simplified approach.
So we use two different approaches depending on object size – if it's below certain threshold it uses just a 2d grid of points aligned to its bounding box. If it's larger – it's unwrapped traditionally.
At this time we also create what we call a probemap – a sort of metatexture describing which lighting probe corresponds to each texel. This is used at runtime to generate actual textures for rendering.
Together with lighting probes for the static meshes we generate a grid of lighting probes hanging „in space" – to be used for lighting dynamic objects.

Generating Connectivity

- Different ways to do this:
  - Can render probemaps to cubemaps
  - Can trace rays on CPU
  - Can trace rays on GPU
- Add the resulting SH vectors to palette, store list of indices in lighting probes
- Compress palette to required size

Once we have the lighting probes and the surface probes we generate connectivity information.
This is basically looking for surface probes visible from all the lighting probes – so we can do it in number of ways.
As we had a pretty decent kdTree in place, we just shoot rays on the CPU. It so it's pretty fast, a full rebuild of the visibility doesn't take more than 20-something minutes. And if it ever becomes too slow we could easily port it to CUDA or CS.
The projections of visible probes are added to the lookup table which is then shrunk to desired size.
Those stages are part of the export process from the MegaMesh tool and the GI data comes out together with the megatextures.

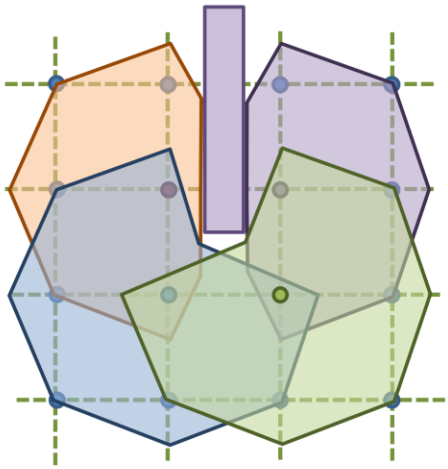
What We Do At Runtime

- Just like described above
  - Comput direct lighting for surface probes – can be done on GPU or CPU - we use CPU and SH visibility for shadowing from sky/sun
  - Propagate lighting to light probes – just add appropriate pallette entries multiplied by direct illumination
  - Generate textures based on probemap.
  - ... And then we just render the geometry

The runtime component is really simple now.
So to generate SH lightmaps we start by computing direct illumination for all the surface probes.
We just do it on CPU.
Then we perform propagation of light to lighting probes – this is just adding appropriate palette entries multiplied by the direct illumination. Once this is done, we use probemap to write this data to textures.
Every step is easily parallelizable, can be executed on multiple cores, and is a good candidate to offload to SPUs.
In our case whole process for for example Milo's bedroom takes about 15ms on a single XBOX CPU – so we can move lights around and have new results every frame (we're making 30Hz game ;-)

Dynamic Objects

- Act only as receivers of indirect illumination
- Use the grid of „floating” lighting probes – basically an irradiance volume
- Custom interpolation scheme – each probe stores its area of influence



The thing that we havn't covered yet are the dynamic objects.
We don't know about them at precomputation stage, so they only act as recivers of indirect illumination.
To light them we use a grid of probe, evenly distributed in space.
Using this approach often causes light bleeding through the wall so we developed a custom interpolation scheme to deal with that.
Each probe stores its area of influence – which in case of our 2d world is just a distance to the nearest occluder in every direction. So when we grab SH coefficients for an object, we check which of the four nearest probes can actually see the object and simply ignore the rest.

Optimizations

- This works fine, but with larger worlds becomes impractical
  - Divide the world into smaller, independent zones, solve zones separately
- Remember only n most important surface probes for each lighting probe
- Use some smarter encoding of SH coeffs in the textures
- Or just use lower SH order

Now for a bunch of optimization tricks.
First of all: we divide our world into smaller, independent zones that can be processed separately. This way, when for example light moves, you only need to update one of few zones instead of all of them.
To save a bit on the memory side you don't have to store all references between surface probes for lighting probes. We do, but we made some experiments, and you can get pleasing results when storing as little as 32 most infuential references for each lighting probe.
You may also consider using lower SH order – we use 2nd or 3rd order SH, but even 1st order SH – which is just a single color, with no directionality at all can give pretty impressive results.

Conclusions

- Lots of further options to experiment with
  - Make surface probes smaller and treat them as points when summing contributions
  - Hierarchy of surface probes
  - Truly seamless SH lightmaps
  - More memory-efficient ways to store surface probes list
- Just do it! It's possible!

Before I show you the video, I just wanted to say that this is not finished research, there is a number o things to try out, just to list a few.
By my general advice is: doi it! Real time GI is possible, is affordable and there is no reason not to have it!

Video

- <show video of the system in action>

Questions?

- Contact us
  - Ben Sugden – ben.sugden@microsoft.com
  - Michal Iwanicki – miciwan@gmail.com

Thanks for coming!