
SIGGRAPH2010
The People Behind the Pixels

A Real Time Radiosity Architecture for Video Games

Sam Martin, Per Einarsson
Geomerics, EA DICE

Hello various rendering dudes.

"Hello, I'm Per."

"Hello, I'm Sam."

Etc.

We (DICE and Geomerics) have been working together to incorporate Enlighten2 into Frostbite engine. The experience shaped the development of both Enlighten and Frostbite, and the resulting architecture and workflows are what we intend to discuss today.

Radiosity Architecture

- Hot topic: real time radiosity
  - Research focus on *algorithms*
  - Several popular "categories" of algorithm
- Architecture
  - Structure surrounding the algorithm
  - Use case: Integration in Frostbite 2

INTRODUCTION (Sam)

Radiosity has always been an exciting topic! There are a *massive* number of papers on the subject. The sheer number of papers that exist gives a near-continuum of algorithmic options, in which we can see some common categories of algorithms emerging as clusters. (eg. Photon-mapping, VPLs, voxel-grid based methods). If you have an idea for an awesome radiosity algorithm, you can probably find a paper somewhere that does something similar and learn the lessons from them. Or another good approach is to start by gluing together ideas from different papers.

In recent years, it's probably fair to say most focus has been on finding **new novel algorithms**, particularly ones focusing on fully dynamic "real time" solutions, and again particularly ones primarily based on the GPU (e.g. Crytech, LPB2 (Evans), VPL-based, "instant radiosity", etc). I won't be adding to this stack in this talk.

This talk is less about demonstrating an algorithm and more about discussing the surrounding architecture and its use in practice. I am also going to claim that the architecture is the more important element to get right.

So what do I even mean by radiosity architecture? It's the structure within which the algorithms operate.

This is orthogonal to algorithmic category in some sense. You should be able to change your algorithm independently of your architecture. This is very true of Enlighten which fairly regularly undergoes algorithmic changes, while it's overall structure remains static. For example, the first version of Enlighten ran on a GPU. Enlighten 2 runs on a CPU. Enlighten 3 might go back the other way again. We may remove the precompute, or we may add Wii support. I'm not sure yet, but these are things that only affect the algorithm, not the architecture.

The key point for this talk is that **the architecture is itself a very power tool**, which I will illustrate with examples. If you bear in mind that if you have an awesome idea for a new radiosity algorithm that will blow everyone away, do consider the architecture as well. And I would expect that if you implemented the 4 architectural features I will cover, you'll get a reasonable and workable solution for even fairly basic algorithms.

DICE and Geomerics have been working together since close to the birth of Enlighten. The integration into Frostbite showcases the use of real time radiosity. We will describe how our pipeline & runtime have been set up to work with Enlighten, and show the workflows we have created to use dynamic radiosity.

Agenda

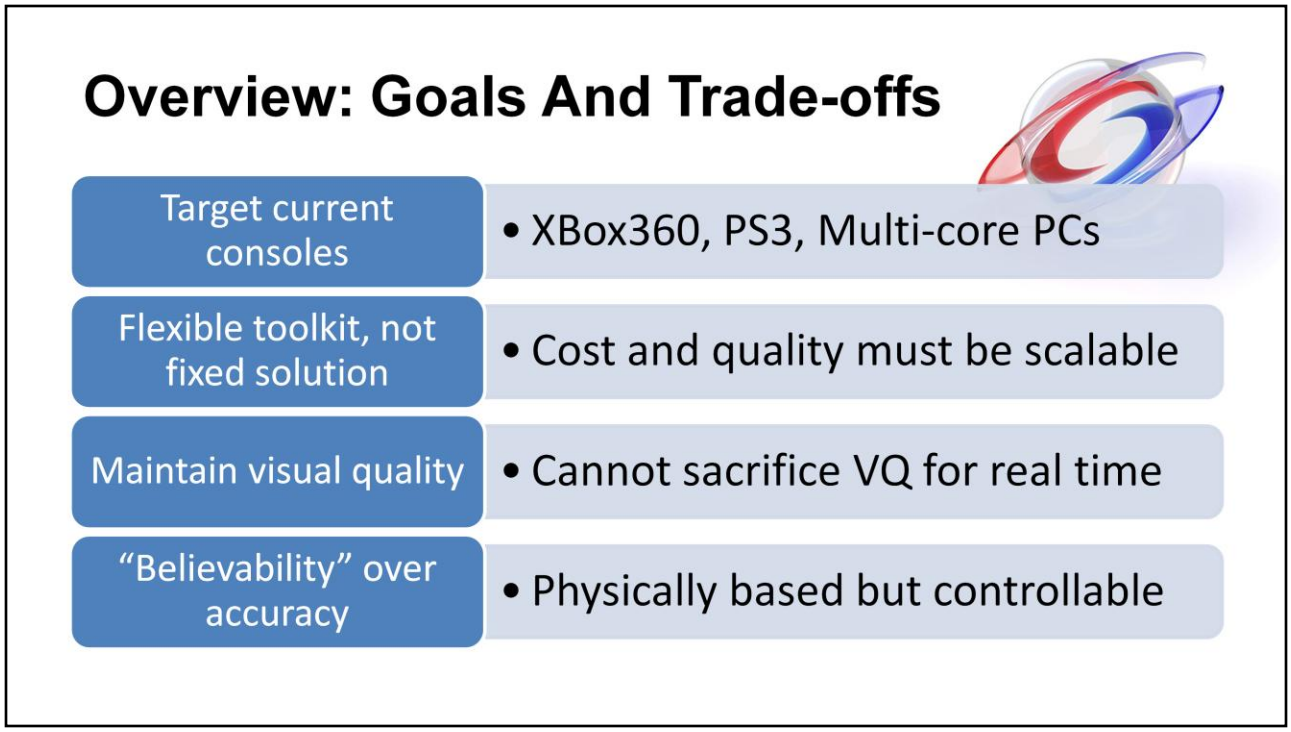
- Enlighten
  - Overview
  - Architectural Features
- Frostbite
  - Overview
  - Pipelines
  - Demo
- Summary / Questions

AGENDA

This talk is essentially me then Per.

I'll do an overview of Enlighten then talk about its architecture.

Per will then demonstrate how and why Enlighten was integrated into Frostbite, and how it shows up in the art workflows.



ENLIGHTEN OVERVIEW 1/2

All good projects need a solid direction, and this is a summary of what Enlighten is trying to achieve which is ultimately reflected in the architectural decisions we made. You can expect the architecture we use to change if you alter these working assumptions.

Current console cycle - constrained by hardware

The main feedback we got from pre-release versions of Enlighten running on GPU was that GPU wasn't a viable resource to use. It's already over-utilised on the consoles, isn't actually that powerful. In addition, the DX9-class hardware the consoles uses constrains our algorithmic options. Memory also very limited. So multi-core is clearly the best target for us. Another point to note is the huge range of abilities between the 3 targets – so scalability of our solution is vital (both up and down).

Always trading off quality with flexibility

There is some imaginary line with two poles. At one end you have offline lighting – great quality, terrible iteration time, not real time. And at the other you have "real time without precomputation" –great iteration and gameplay, but low quality lighting. Note you don't get to have your cake and eat it. Real time costs visual quality, but so does poor iteration time. So **Enlighten2 is a midpoint.** We do some pre-processing to keep visual quality up, so our lighting does not fully reflect moving geometry without the addition of some smoke and mirrors. But we do provide fast iteration through real time lighting. In short, we focus on good quality, scalability and flexibility, not fully dynamic at all costs.

Frostbite wanted a lighting solution with fast iteration times and support for dynamic environments. Previous static techniques DICE used gave great results, but were painful to wait for lightmaps to build. They wanted to give artist more creative freedom.

Art controls

There are many points at which artists can add value. The end goal is always to create beautiful, emotive, believable images, not physical realism. So where ever possible we allow artists control over all aspects of indirect lighting. For example, there are per-light indirect scaling/colouring controls so they can tweak the radiosity effect from each light source. There are also global direct/indirect balance controls and radiosity tonemapping controls, amongst many others.

Four Key Architectural Features

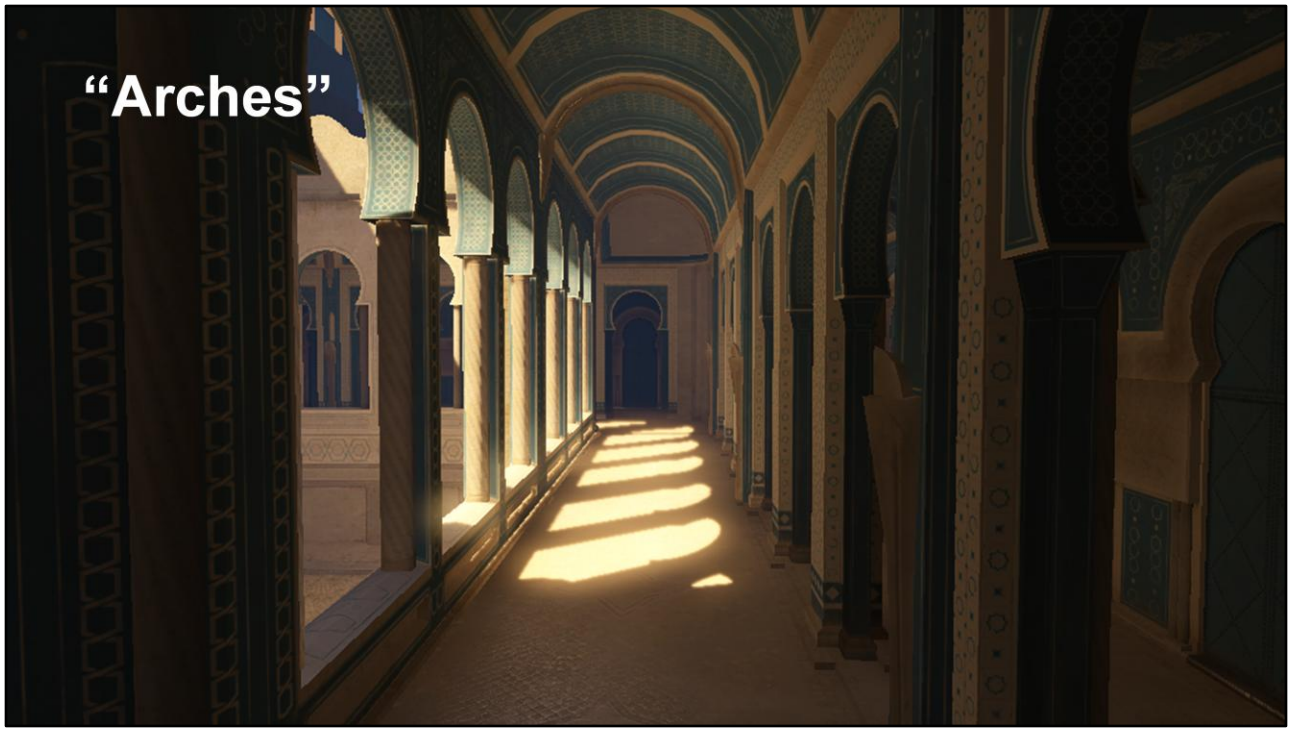
1. Separate lighting pipeline
2. Single bounce with feedback
3. Lightmap output
4. Relighting from target geometry

ENLIGHTEN DECISIONS

These are the 4 key architectural features I'd like to dig into further today.

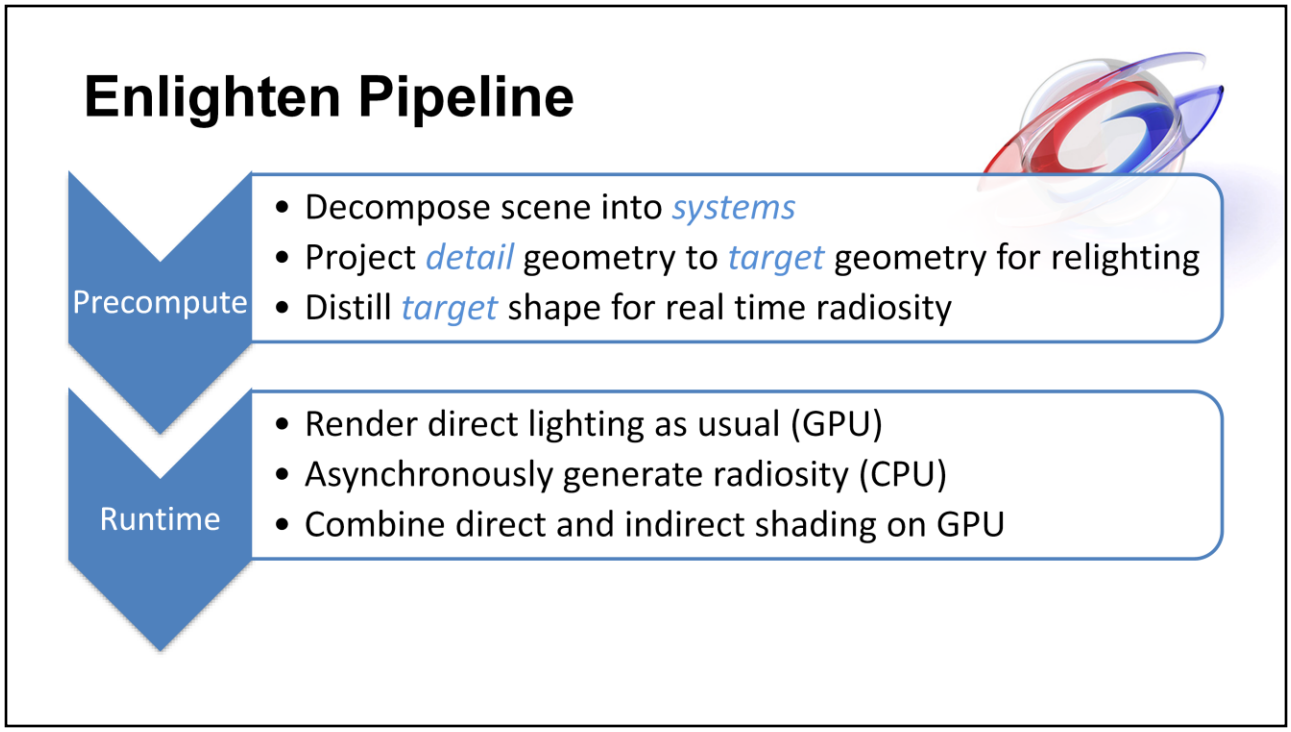
Point to bear in mind is that if you were to implement some algorithm that had these architectural features, it'd most likely either be realtime or very close to realtime.

Will now walk through each in turn.



This is the asset I will use as an example.

It’s similar to the sponza atrium. We built it internally and ship it in our SDK.



ENLIGHTEN Pipeline

This is a high level summary of the Enlighten pipeline.

Precompute

There are a number of new terms – I will attempt to unpack them as we proceed. The precompute in Enlighten is essentially a problem-distillation stage. We attempt to put as much magic into this as possible. Some details to be aware of now:

We break up the scene into "systems" – a system is a locally solveable problem but taken together they still provide a global solution. Each system outputs a small lightmap for part of the scene.

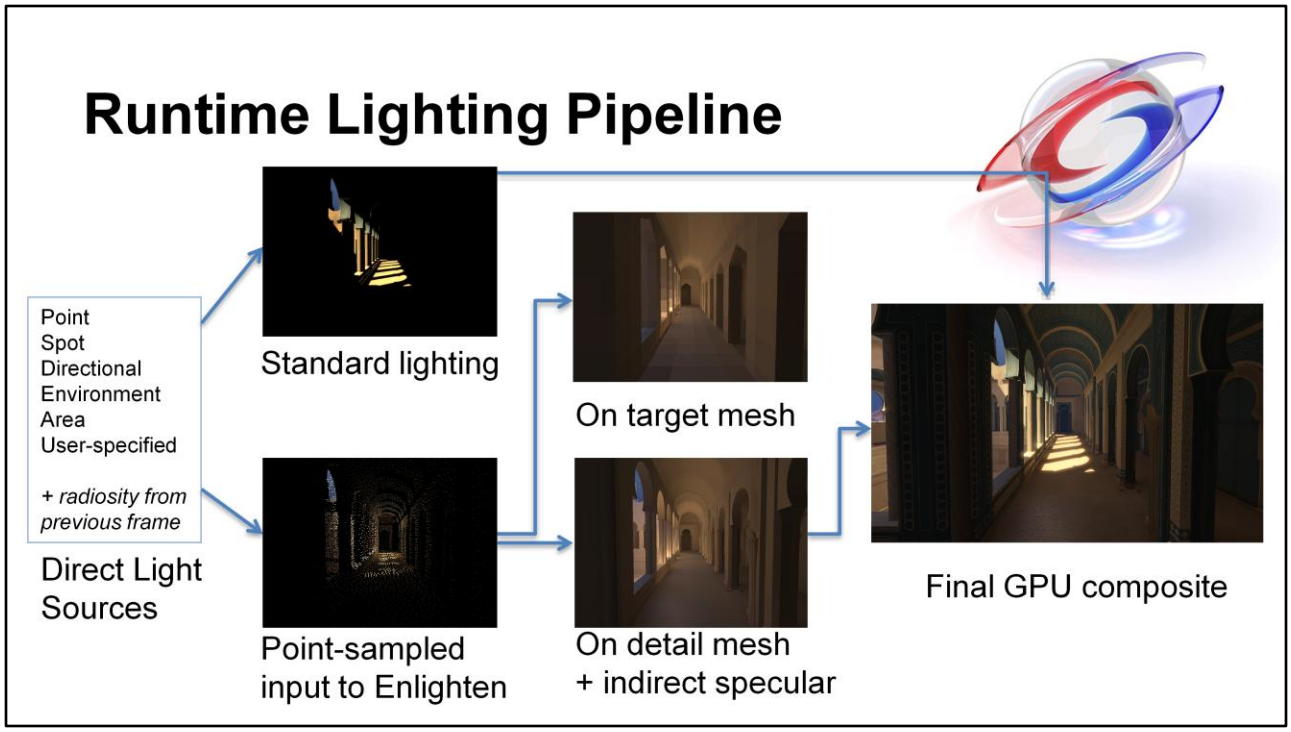
The precompute also setup "relighting" information – I will cover this in more detail later on.

Precomputing information is a tough compromise to make, but very important to getting near-to-offline lighting quality for our target platforms.

Runtime

The key point (and one of my 4 architectural features) is that our **runtime separates indirect lighting work from direct lighting work**. The direct lighting is done on the GPU as usual, while all indirect lighting is done on the CPU (the current best place for target platforms). Both can run asyncronously, so you get good parallelism. And most significantly, the previous output of Enlighten is always available. So our output is **cacheable**. The GPU just composites with the latest results off the carousel. This is a massive optimisation, largely independent of algorithm that simply comes from using the right architecture. I'll describe this further in a few slides time.

A final point: The separation is not just an optimisation – also allows a lot of creative freedom different integration options. Per will talk a bit more about this in his half of the talk.



LIGHTMAP OUTPUT

We shall walk through an example shot in our runtime pipeline.

Note the split of direct and indirect lighting paths, with the dependence on a common description of the lighting in the scene. We have all the usual lighting types, plus some new ones.

“Environment” is essentially a very low detail cube map. It gives you the same effect a simple image-based light would: Soft shadows, directional lighting. Nothing sharp.

“Area” lights are easy in Enlighten. Our tech essentially allows you to make surfaces glow and compute the bounce result. So an area light is pretty much the simplest form of input lighting for us.

Enlighten is careful to not restrict the API to predefined light sources. If you can describe the lighting incident on your geometry surface and point sample it, you can put it into Enlighten. For example, anything you can shade in a deferred renderer on the GPU can immediately be put into Enlighten.

This is the traditional direct lighting in this shot. It's a simple cascade shadow map based directional light, computed on the GPU. This bit is nothing to do with Enlighten.

The choice of light and its implementation is essentially arbitrary.

Note the lack of direct *environment* lighting. Environment lighting (and other area lights) are handled entirely within Enlighten.

This is the corresponding input to Enlighten. In this example, the two sets of data are computed entirely independently.

The main runtime operation in Enlighten is to map a point sampled description of the lighting over the target mesh surface (shown above), to a lightmap or lightprobe output. This is the bounce. Anything you can express in these terms is valid input to Enlighten.

When generating the point sampled input, we provide several fast paths for common light types. Directional lights with precomputed visibility to static geometry is one of these options, which we use in this example. This provides a fast efficient method of generating the input lighting that does not require any interaction with the GPU-rendered sunlight, although we could have chosen to point sample that lighting instead.

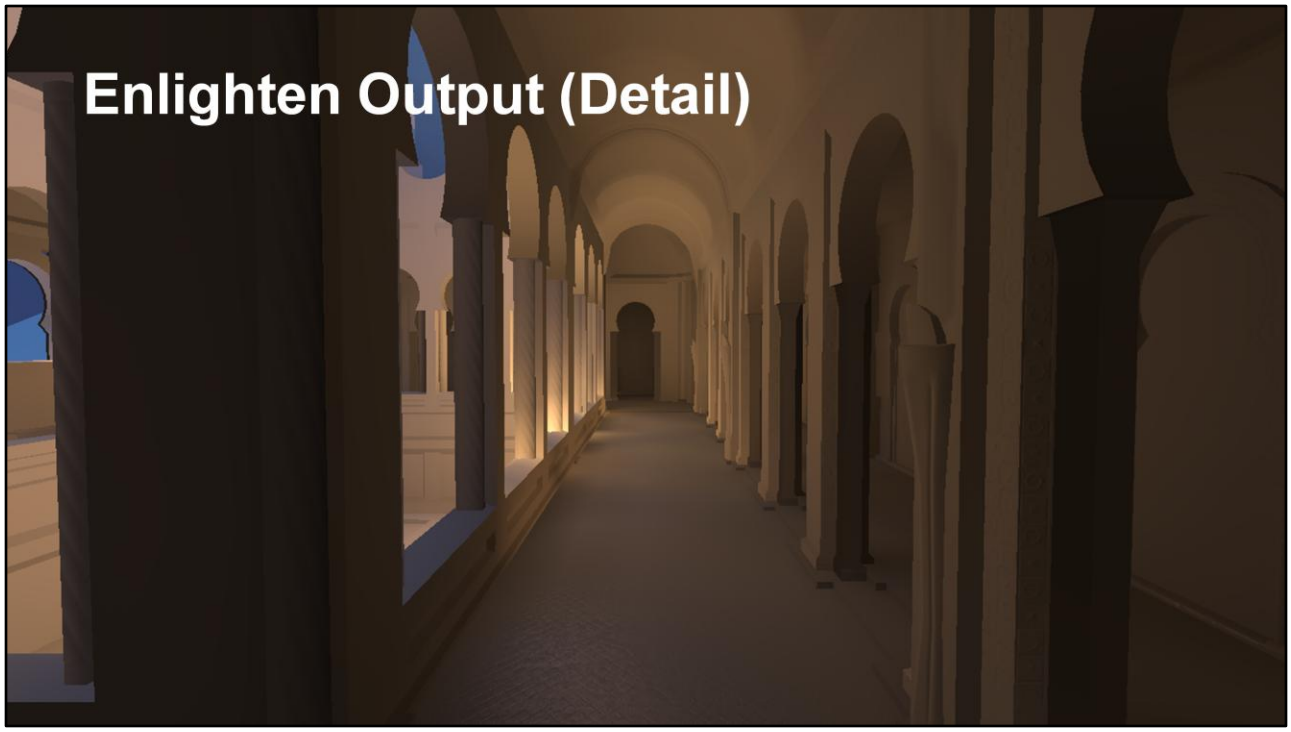
You may note that as well as the directional light, there is also radiosity lighting in the point sampled data. This is the previous lightmap output being fed to enlighten as an input light to generate a second bounce. This is how we generate multiple bounces. I'll return to this later on.

This is what the previous point sampled data is mapped to. In some sense, this is the raw lightmap output from Enlighten, before any interpolation, albedo modulation or directional relighting is applied.

The shot shows the generated lightmap over the “target” mesh for this scene.

Note how low resolution the output is. You can see the effect of the skylight (blue) and the bounced directional light.

Although low detail this resolution captures the essence of the indirect lighting. Significant soft shadows are present, and the lighting gradients can be seen quite clearly.



This shows exactly the **same** lighting data, but applied to the (white untextured) detailed geometry, together with normal maps and indirect specular.

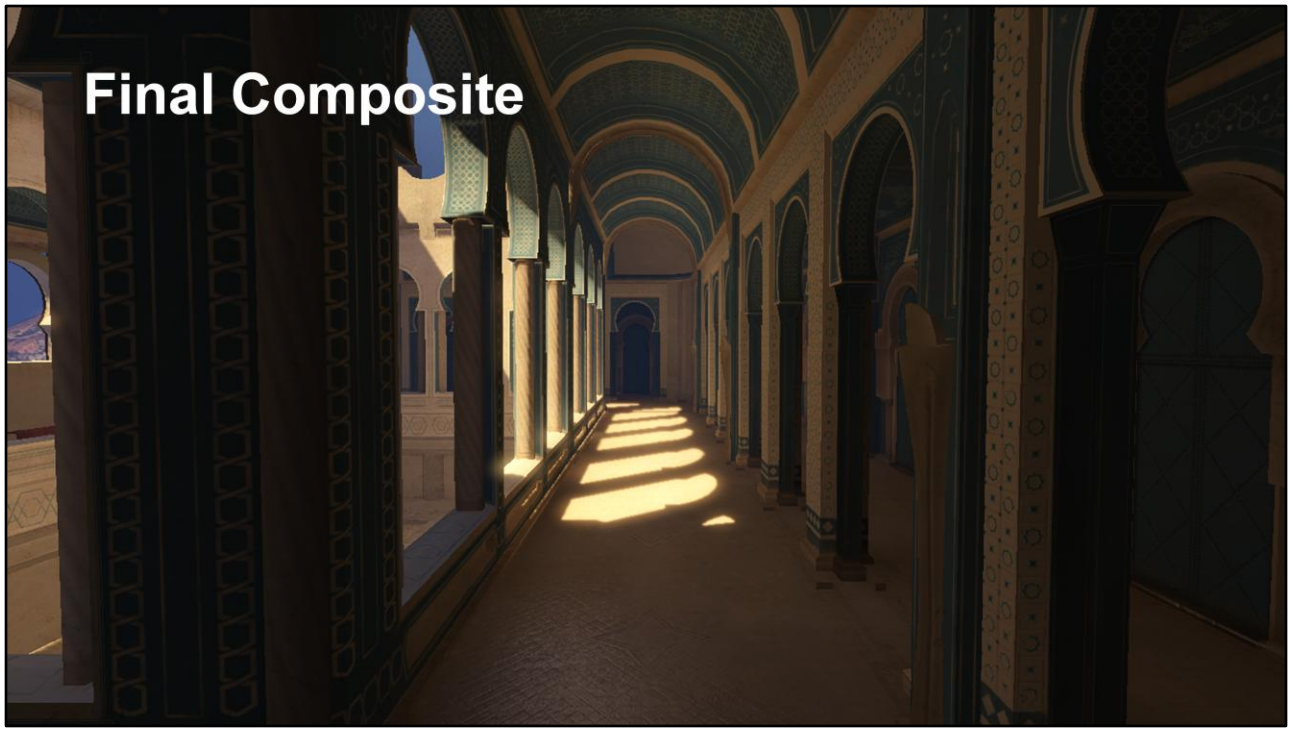
Note how much the basic lightmap output gives you when taken together with these additions.

In particular, hardware interpolation gives you a lot as long as your output lightmap is very 'clean'. This cleanness is important - we scale linearly with output texels, so each texel really has to work for it's place in the lightmap. If you want a very dense output format, as we have, you can't afford noise.

Much of the detail is filled in by the off-axis relighting. Simple variation in normals gives you a lot of lighting variation.

There are multiple output formats for Enlighten. This particular screenshot is using our "directional irradiance" technique which is a mid point between a full spherical output (e.g. Spherical harmonics – complete spherical lighting data) and direction-less irradiance. We support all 3, but prefer the directional irradiance as a good balance point in practice.

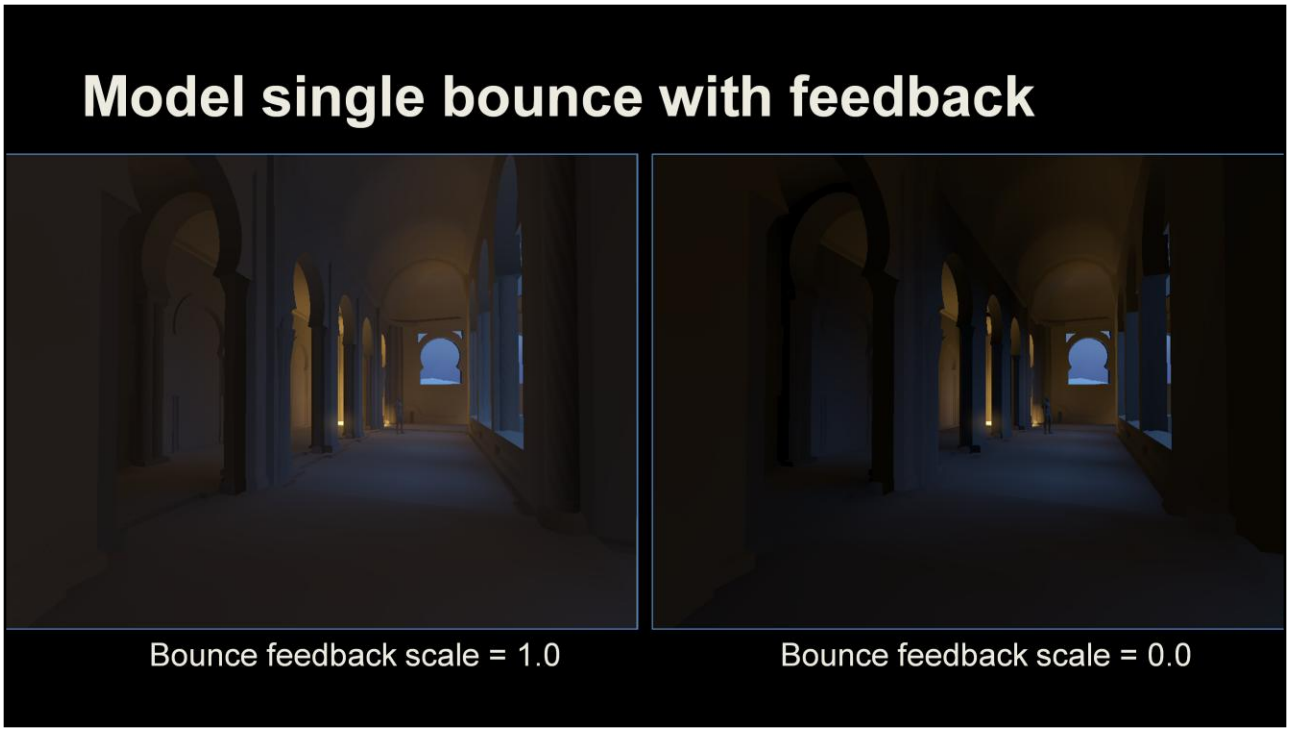
The specular effect is a simple 'imaginary' specular effect generated in the final shader. Think of it as a phong specular model parametised by the strongest lighting direction. You only get one specular highlight, but your eye is very intolerant to errors in specular term. Believable specular results are far easier to obtain than "accurate" ones. So we prefer to give as much control to the user/artist as possible here.



And here we see the final composite. I've just added the textures and direct lighting back.

This was a brief introduction to our runtime radiosity pipeline. Per will talk through its use in Frostbite later on.

But now I want to move to the next architectural point: the modelling of a single bounce...



ENLIGHTEN SINGLE BOUNCE

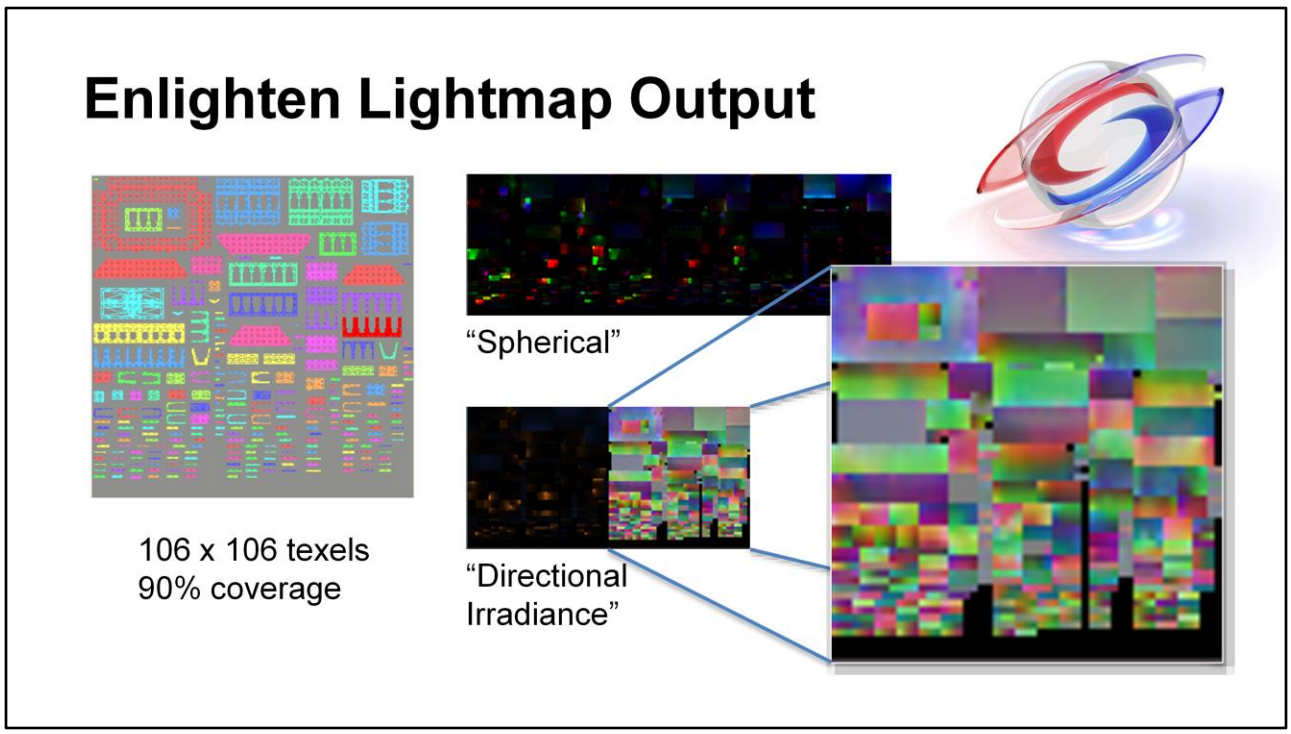
The key point here is that this is a big algorithmic simplification, and actually adds rather than takes away. Capturing the converged radiosity output is important – a single bounce is not enough. But I'd claim that modelling them through a feedback loop is just "The Right Way".

Why is it a big win algorithmically?

1. Easy dynamic albedo. This is now modelled by modulation of the input lighting and independent of everything else.
2. Visibility/light transport much simpler to compute and compress.

Both of these points are consequences of avoiding the cross-terms that occur when you attempt to compute bounce terms beyond the first, or the converged integral. There are some very elegant methods for computing the fully converged integral (based around the geometric series solution S = (1-T)^-1), and if you never intended to update some aspect of that solution (including the lighting!) maybe this would start to look more attractive. But in practice I don't expect this to be important enough.

It's worth noting that convergence for radiosity is quick. We typically see convergence in 4 frames or less. So, if you were to update one bounce a frame at 60fps you'd get a lag of 4*1000/60 = 66 ms. You can't see this in practice. The decoupled direct lighting also helps you here, as this is correct even if the indirect lighting is still changing. Even if there is a lag, your direct lights are always lag-free. It's harder to perceive the lag in indirect lighting in this setting, and only becomes apparent with momentary light sources (grenades, muzzle flash, etc).



The word "Lightmap" does tend to conjure up images of large pages of memory-hogging textures. Enlighten lightmaps are actually really very small and very densely packed – there is very little wastage. This is important as memory footprint and bandwidth usage for our output are low. The smallness primarily comes from only storing indirect lighting, and secondly, from using target geometry to simplify the uv surface area. The target mesh lightmap is not a 1-1 mapping of the original detailed surface and the low numbers of charts allow us to get very high coverage.

We also support a number of different output texture formats that all share the same layout. So you can swap algorithms easily.

As well as cheap interpolation, and as I mentioned earlier, the **key property** lightmaps give us a **cacheable** output that allows us to **amortise or completely avoid** the cost of computing the indirect lighting. This is in contrast to many gpu solutions that are temporary or view-frustum only and incur a fixed cost regardless. This property gives us a wide range of scalability options. This is the real point of using lightmaps.

If you consider the different kinds of lighting environment you might encounter in a video game it becomes more apparent why this is helpful:

Modelling outdoor time-of-day lighting

Moving the sun motion triggers global lighting update (expensive), but the sun moves slowly so we can stagger updates across frames. Essentially, lots of work but we can do it gradually. This makes large outdoor works feasible.

Intimate indoor lighting

In a more detailed enclosed environment with rapidly updating lighting you will have to update more detailed radiosity more rapidly, but the restricted visibility helps you. You only need update visible areas. Streaming also becomes viable.

A zero-order integration

Another use to only use Enlighten offline. Turn all the quality dials up in your editor, and run everything in realtime there, then simply only store the output lightmaps for use at runtime. Enlighten effectively becomes a real time light baking solution. This also allows you to target any platform that supports textures (wii, etc).

"Parametised" lighting descriptions

It's fairly common to re-use the same level, but with different lighting (time of day, weather changes, etc). Particularly in MMOs. Here you only need re-compute the radiosity on level load.

Target Geometry

Has simple UV surface area

Tri count not important

Various authoring options

Lets take a closer look at the target/detail mesh projection. This is the target geometry authored for our arches scene.

Unfortunately, I don't have time to cover the algorithm used in any detail, but it's quite a simple algorithm. In this area we've found that the simple approach is often the best. Artists need to be able to predict the behaviour, so complex rocket-science algorithms full of awesomeness that sometimes have unpredictable behaviour can actually make your tools harder to use!

The target geometry does two things:

1. It allows us to control and capture lighting at the resolution we choose, and not be bound by the geometry we actually want to relight.
2. It allows us to reducing the surface area and chart count which is a major optimisation.

You can see that the mesh itself is very basic. The key thing is to get a mesh that has a simple uv surface area (low chart counts). This mesh happens to have a low tri count, but this is not important. Collision geometry or a low LOD is usually a good starting point, but they have slightly different requirements.

Detail Geometry

UVs generated by projection

No additional lighting data

“Off-axis” lighting comes from directional data in lightmap

Does not interact with radiosity

This is the detail geometry with the lighting from the target mesh lifted onto it.

The lifting operation is actually an offline mesh-to-mesh projection, which as I said, is deliberately simple.

A big plus of this approach is that there is no need to author uvs for the detail geometry. These are generated for you during the projection. This is a rather cool property that may have other uses: by modelling easy to author target geometry you can skip uv authoring on complex detail geometry.

The downside of target geometry is that, despite being simple, is still an additional authoring overhead. We already have a set of UV authoring tools to simplify the process and are investigating automated approaches to generating this information.

Recap: Architectural Features

1. Separate lighting pipeline
2. Single bounce with feedback
3. Lightmap output
4. Relighting from target geometry

To recap, these are the 4 architectural features we've just covered.

Separate lighting pipeline: Radiosity calculated independently of and asyncronously to rendering engine.

Single bounce: Big algorithmic simplification

Lightmaps: Compact, controllable and cacheable indirect lighting representation

Target geometry: Separate lighting resolution from geometric detail

I'll now hand over Per to discuss their use in Frostbite...

Agenda

- Enlighten
  - Overview, Architectural Features
- Frostbite
  - Motivation
  - Pipeline
  - Runtime
  - Demo
- QA?

So I'm going to go through how we've set up the Frostbite Engine to run with Enlighten and how our pipeline and runtime support this architecture. But first I'm going to talk about why we started to work with Geomerics, and why we decided to go for a real-time radiosity solution in Frostbite.

Motivation

- Why real-time radiosity in Frostbite?
  - Workflows and iteration times
  - Dynamic environments
  - Flexible architecture

So why do we want real-time radiosity in a game engine? For us the main argument was to improve the workflows and iteration times.

We've been working with traditional lightmap techniques at Dice, and even if the results can look amazing in the end, it was just a very painful way of creating content . It's not unusual that artists spend hours waiting for lightmap renders to finish. So we thought, if artist can spend their time working on actually lighting the game instead of waiting for lightmaps to compute, then perhaps the end results will look more interesting?

Another main argument is to support for dynamic environments. Video games are becoming more dynamic, so if we change the lighting in the game, we should also be able to update the bounce light dynamically.

And finally, the architecture that came out of integrating Enlighten into Frostbite turned out to be pretty flexible. The direct and indirect lighting pipeline is completely separate, so the architecture is pretty robust to general changes to the rendering engine.

Precompute pipeline

1. Classify static and dynamic objects
2. Generate radiosity systems
3. Parametrize static geometry
4. Generate runtime data

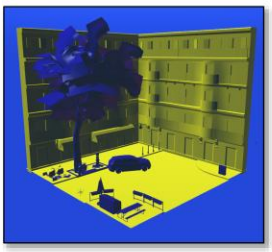
Before we can run the game with dynamic radiosity, we have to do a few things in our precompute pipeline.

I'll go through these steps one by one.

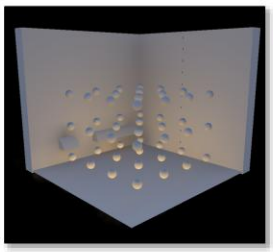
1. Static & dynamic geometry

- Static objects receive and bounce light
  - Uses dynamic lightmaps
- Dynamic object only receive light
  - Samples lighting from lightprobes

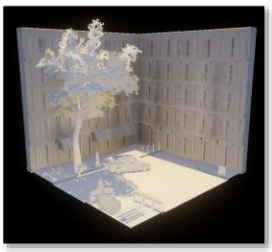
Input scene



Mesh classification



Underlying geometry



Transferred lighting

Enlighten provides two ways of representing the bounce light. Either via lightmaps, or via lightprobes. The first thing we do is to decide how each object should be lit.

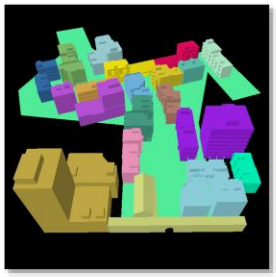
Static geometry is parameterized and lit with dynamic lightmaps. This geometry can bounce light, and is typically large objects that don't move.

Dynamic objects can only receive bounce light by sampling lightprobes, so they are typically moving around in the scene or just small and don't affect the lighting to much themselves.

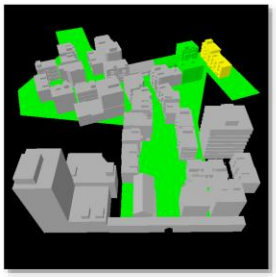
In this scene you can see how we've separated static and dynamic objects. The underlying geometry is used to bounce light, which is then transferred to all objects in the scene.

2. Radiosity systems

- Processed and updated in parallel
- Input dependencies control light transport
- Used for radiosity granularity



Systems



Input dependencies

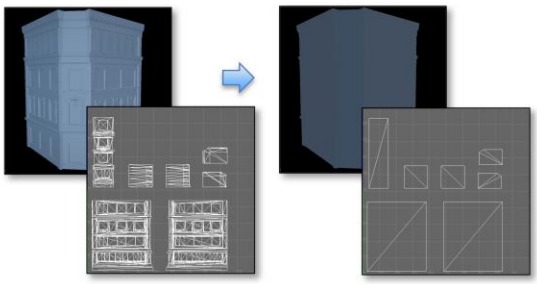
One of the key features in Enlighten is to group objects into systems. A system is a collection of meshes that can be processed independently, and this really makes the radiosity a more local problem. Each system can be processed in parallel, the precompute can be distributed and runtime updates can be separated.

We automatically define input dependencies to each system, which is a way to put restrictions on the light transport. So when we update the yellow system here, we only read bounce light from the green systems and we can forget about the rest.

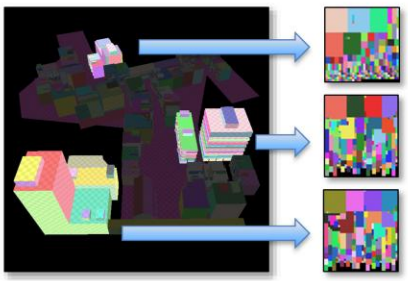
We also use systems to control update performance. Large systems will have many pixels and it will take longer to compute, so by creating many small systems, we can spread out radiosity updates on several frames if we like. We typically update one system per CPU core every frame.

3. Parametrization

- Static meshes uses target geometry
  - Target geometry is used to compute radiosity
  - Project detail mesh onto target mesh to get uvs
- Systems packed into separate uv atlases



Automatic uv projection



System atlases

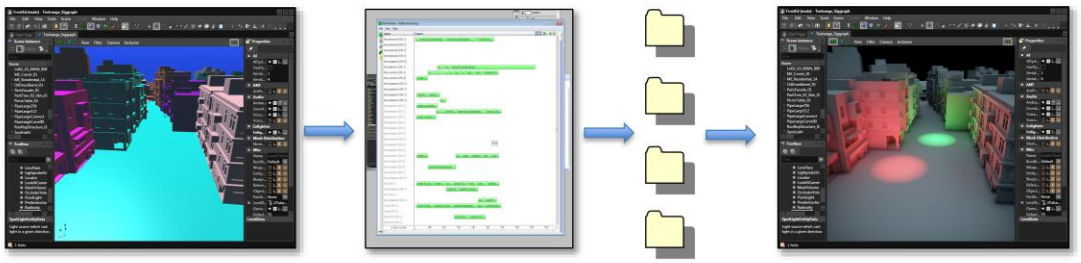
We need to parametrize the world to get lightmaps uvs, and we do this semi-automatically.

For each static mesh we also have a low-poly target mesh that we use for lighting. The target mesh is manually parametrized, and we project the detail mesh onto the target mesh to generate uvs for the detail mesh.

We also pack a uv atlas for each system. Each system atlas is independent of all other systems, so we end up with one lightmap per system.

4. Runtime data generation

- One dataset per system (streaming friendly)
- Distributed precompute with Incredibuild's XGI
- Data dependent on geometry only (not light or albedo)



Distributed precompute pipeline generates runtime datasets for dynamic radiosity updates

When we have generated systems and parametrized the geometry, we can generate runtime data in our precompute pipeline.

The runtime data has information about geometry and form factors that we need to update the radiosity in real time.

There's one data set per system, which is very nice if we want to stream data from disk.

All systems can be processed in parallel, so we use Incredibuild's XGI to distribute this build step. This is the only time consuming step of the precompute, but it scales pretty well with incredibuild. A typical final game level takes about 10 – 30 min to precompute.

Since this data only contains geometry information, so we only have to regenerate it when we change the geometry, not the lighting or the colors in the scene.

Rendering

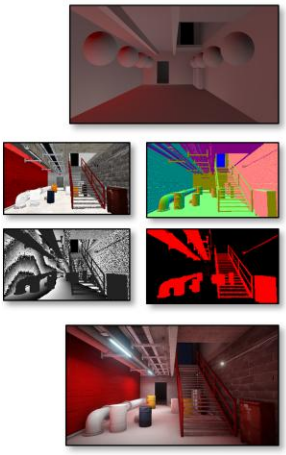
- Separate direct light / radiosity pipeline
  - CPU: radiosity
  - GPU: direct light & compositing
- Frostbite uses deferred rendering
  - All lights can bounce dynamic radiosity
- Separate lightmap / lightprobe rendering
  - Lightmaps rendered in forward pass
  - Lightprobes added to 3D textures and rendered deferred

Let's take a look at the runtime. A key thing with this architecture is that we have a separate render pipeline for direct and indirect radiosity. In fact, we update the radiosity on the CPU and we do the direct light & compositing on the GPU.

Frostbite uses deferred rendering, so can we can render many light sources every frame. Each light source is fed to Enlighten and part of the radiosity bounce light.

Another thing we do is to separate the rendering of lightmaps and lightprobes. Lightmaps are rendered in the forward pass, but lightprobes are added to 3D textures so we can render them deferred in screen. The reason we do this is so we don't have to upload a unique lightprobe for every object we render, which tends to quite a few objects if you consider foliage, vegetation, particle effect and decals, so adding lightprobes deferred in screenspace is just simpler for us.

Runtime pipeline

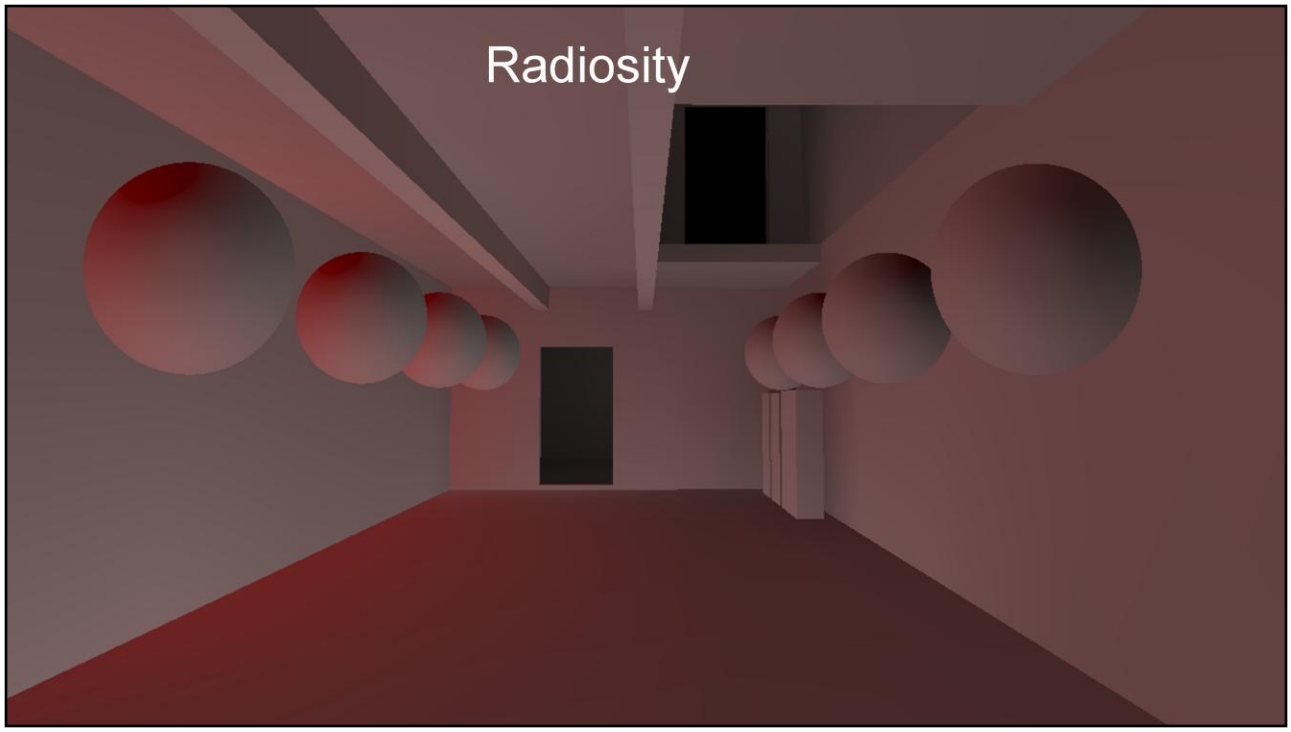


1) Radiosity pass (CPU)
   - Update indirect lightmaps & lightprobes
   - Lift lightprobes into 3D textures
2) Geometry pass (GPU)
   - Add indirect lightmaps to separate g-buffer
   - Use stencil buffer to mask out dynamic objects
3) Light pass (GPU)
   - Render deferred light sources
   - Add lightmaps from g-buffer
   - Add lightprobes from 3D textures

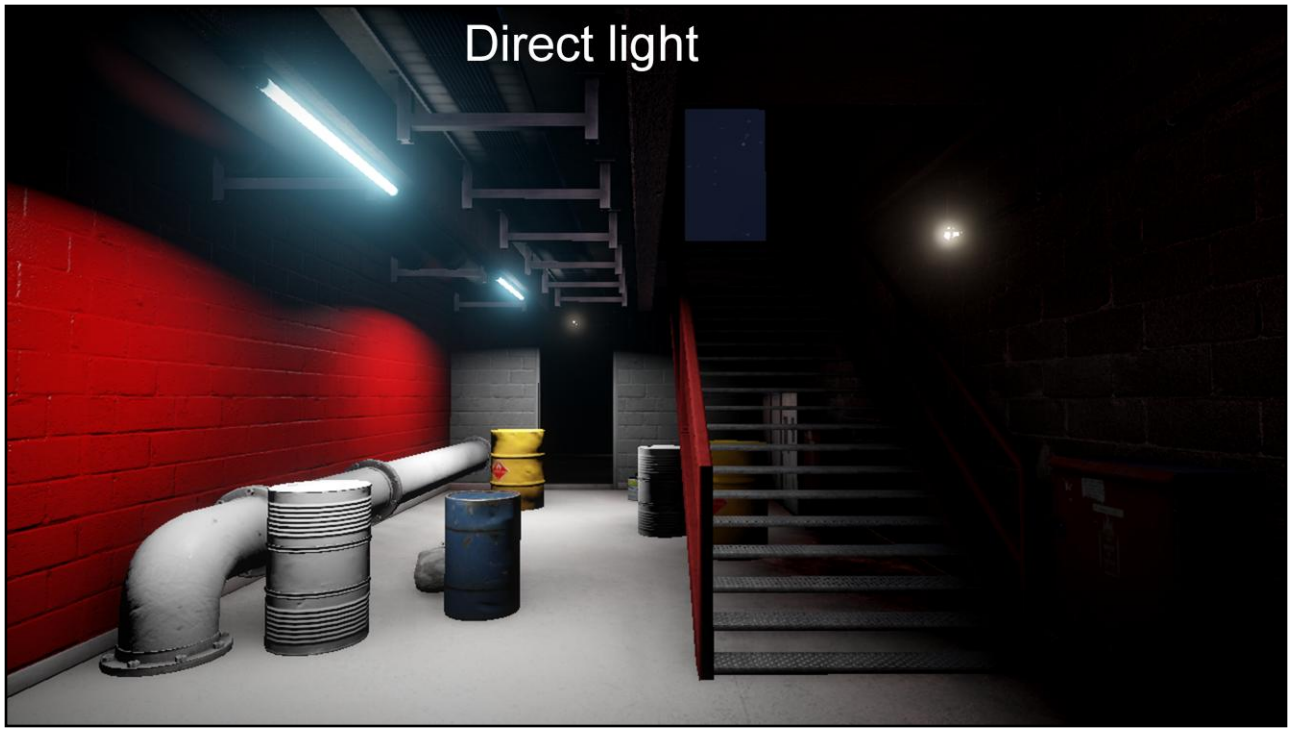
The complete render pipeline looks like this. First, we update lightmaps & lightprobes on the CPU, and we lift lightprobes into 3d textures. These 3d textures cover the entire scene.

Next, we run the geometry pass, where we add bounce light from the lightmaps to a separate g-buffer which we LogLuv encode. We also use the stencil buffer to mask out all dynamic objects, so we know what to light with lightprobes later.

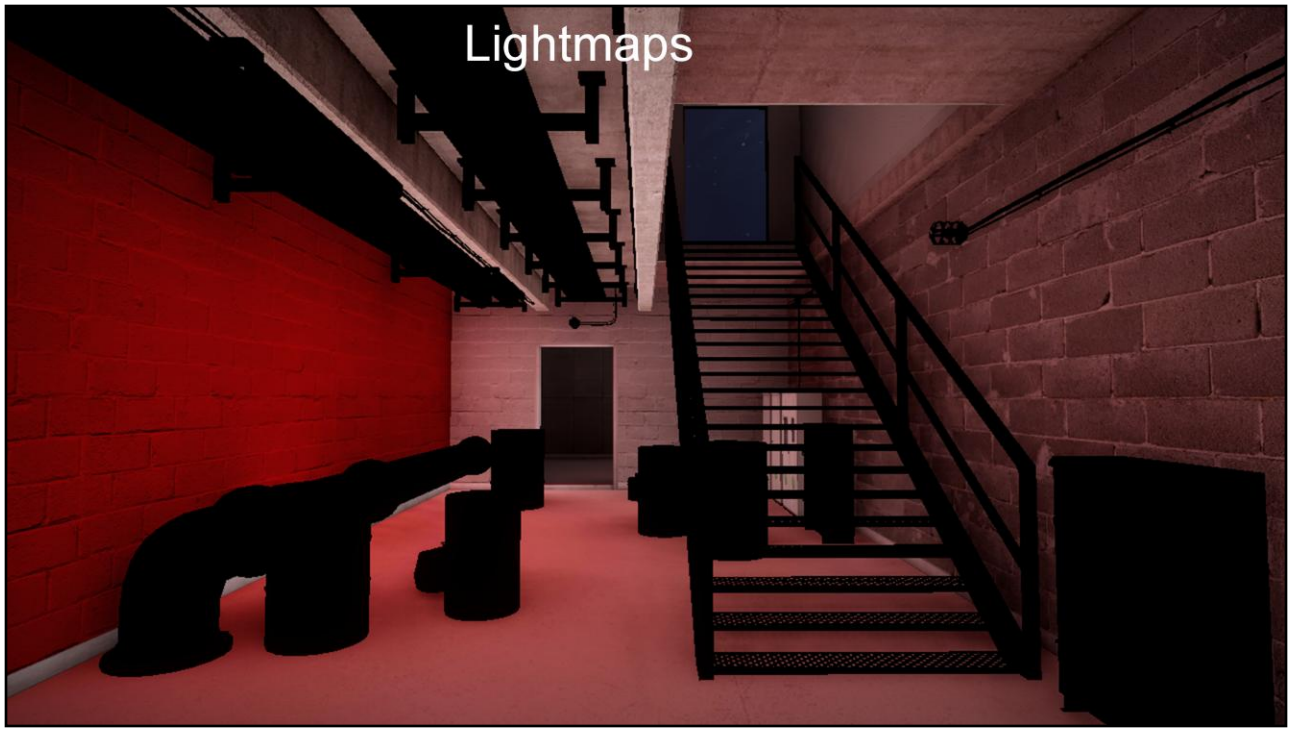
Finally, in the light pass, we first render all deferred lights, we then add the lightmaps from the g-buffer, and finally we add the lightprobe 3d textures deferred in screen space.



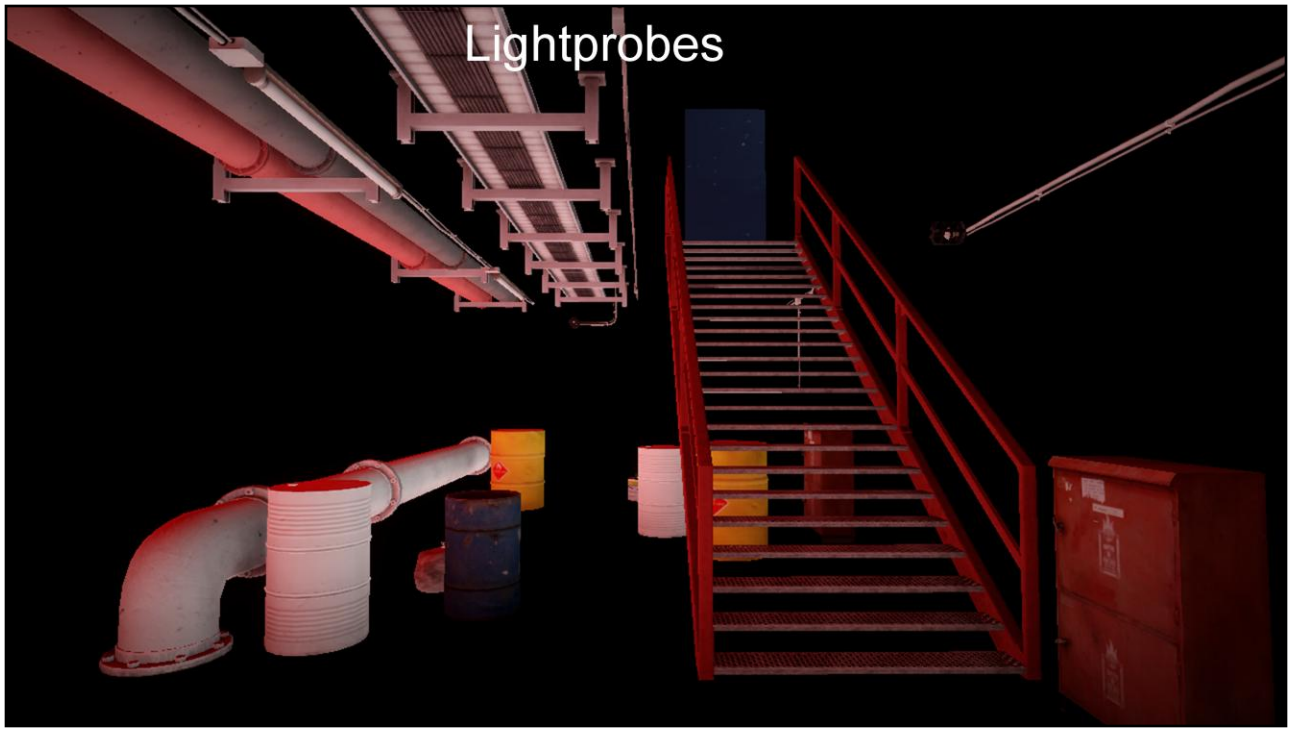
Let’s take a look at an example scene. This is the lightmaps and lightprobes generated on the CPU.



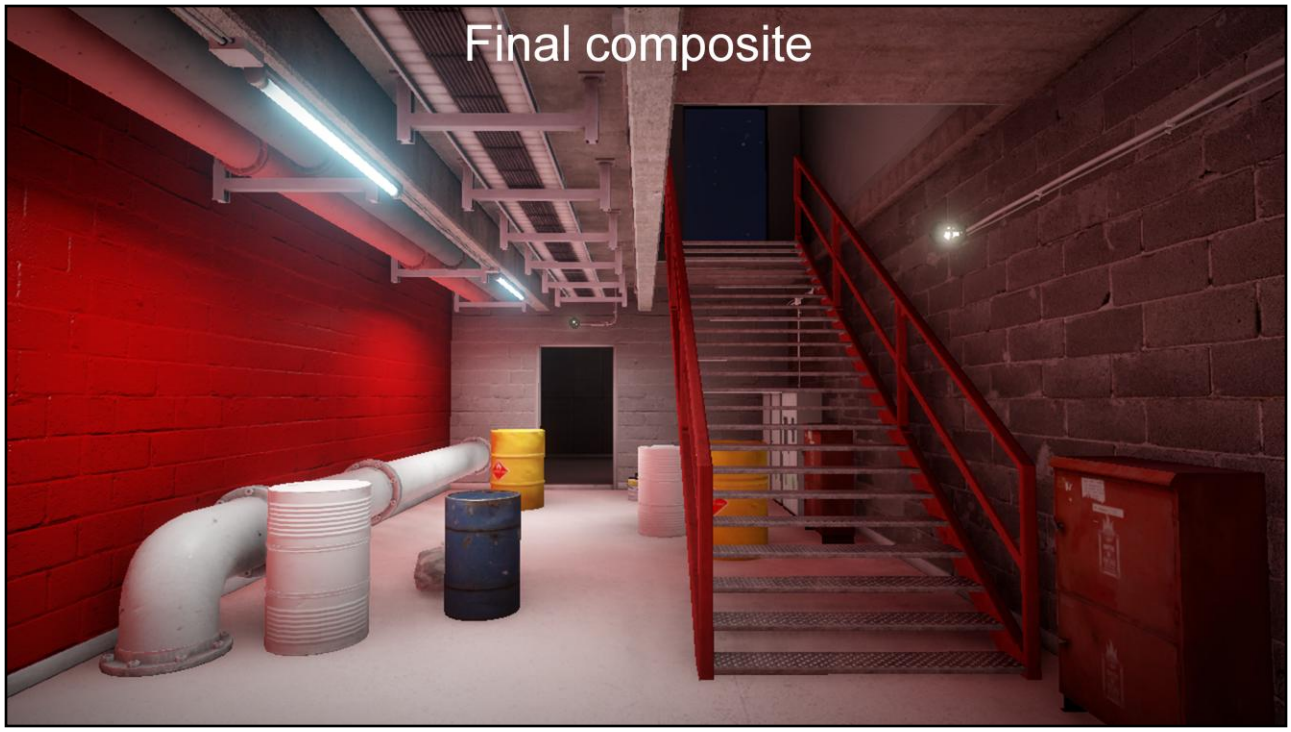
First we render the direct light sources deferred.



Then we add the lightmap bounce light.



Then we add bounce light from the lightprobes. We add them all together, to get the final compposite.



Final composite

Demo

(This is where we ran the video – Per gave a commentary over it during the talk).

Summary / Questions?

- Thanks!

- per.einarsson@dice.se
- sam.martin@geomerics.com

For people emailing: Please note the lack of the ‘t’ in Geomerics! Geometrics is a different company. I don’t know if they have a sam.martin there, but if they do they might be a bit confused by now ☺.