Multi-Scale Global Illumination in Quantum Break

Ari Silvennoinen
Remedy Entertainment
Aalto University

Ville Timonen
Remedy Entertainment

SIGGRAPH 2015
I work for Remedy Entertainment, which is an independent game studio based in Helsinki, Finland.

Remedy is best known for creating the Max Payne and Alan Wake franchises but today we are here to talk about Quantum Break.
Before we go any further, let's take a look at a video to find out what the game is all about.
Quantum Break is built on top of a custom, in-house engine, which is based on a physically based deferred renderer.
When we started to work on the renderer, we looked closely at the concept art to determine what kind of rendering and lighting features are needed.

Quantum Break is a game about time travel in the present day, and we wanted to have some clean, high-tech environments.
For example, looking at this concept art, it was clear that we needed a way to address specular reflections.
In addition to the clean, highly reflective scenes, we have these worn out industrial environments.
In this example, most of the scene is indirectly lit by the sun and the sky, so it became clear that we needed to have some form of global illumination.
Here is another example of an industrial environment.
Again, the scene is mostly lit by indirect illumination, but this time there is also participating media and light shafts in the background.

This meant that we also need to have the volume lighting affected by the global illumination.
One of our primary goals with the new renderer was consistency across the lighting.

We wanted to have the environments, dynamic objects, particles and volumetric lights all blend in together seamlessly.
Design Goals and Constraints

Consistency
Semi-dynamic environments and lighting

The second constraint was that we had to support large scale destruction events and, in some levels, dynamic time of day.
Finally, since we are a small team, we wanted to have a fully automatic system, which would require minimal amount of artist work.
When we started to design the renderer,
we knew that screen space lighting effects capture the small, high-frequency details quite well.
However, the screen space buffers don’t contain enough information about large scale lighting effects,
like these.

Looking at the individual components, it is clear, that in isolation, they do not provide enough detail to create a balanced image.

But, if we combine them together,
we get something that works quite well across multiple scales of detail.
So, inspired by this multiscale approach, the rest of the talk is divided into two parts.
In the first part, I'll present our approach to global illumination, and in the second part, Ville will present the screen space techniques, which are used to complement the large scale effects.
Possible Solutions for Global Illumination

Dynamic Approaches
- Virtual Point Lights (VPLs) [Keller97]
- Light Propagation Volumes [Kaplaynan10]
- Voxel Cone Tracing [Crassin11]
- Distance Field Tracing [Wright15]

Let's begin with global illumination.

Ideally, the solution would be fully dynamic, and, for every frame, we could compute the global illumination from scratch.
Possible Solutions for Global Illumination

**Dynamic Approaches**
- Virtual Point Lights (VPLs) [Keller97]
- Light Propagation Volumes [Kaplaynan10]
- Voxel Cone Tracing [Crassin11]
- Distance Field Tracing [Wright15]

Cost was too high for the **quality** we wanted

We experimented with voxel cone tracing and virtual point lights but it became clear that achieving the level of quality we wanted was too expensive with these techniques.
Most traditional rendering algorithms, such as path tracing, determine intensity by averaging many samples (MC sampling). This approach is not useful for finding derivatives.

Possible Solutions for Global Illumination

- **Mesh-based Precomputation**
  - Precomputed Radiance Transfer (PRT) [Sloan02]
  - Spherical Harmonic Light Maps

- **Meshless Precomputation**
  - Irradiance Volumes [Greger98]

So, during pre-production, it became clear that we needed to use some form of precomputation.
Most traditional rendering algorithms—path tracing, etc.—determine intensity by averaging many samples (MC sampling) —sample all possible paths light can take from source to sensor. Not useful for finding derivatives.

Possible Solutions for Global Illumination

Mesh-based Precomputation
- Precomputed Radiance Transfer (PRT) [Sloan02]
- Spherical Harmonic Light Maps ✗

Meshless Precomputation
- Irradiance Volumes [Greger98] ✓

We quickly ruled out the mesh-based approaches, because they don’t play well with dynamic objects, i.e., it’s difficult to achieve a consistent look. It’s not uncommon to see dynamic objects clearly standing out from the static background geometry: one of our key goals was to avoid this and we looked for a meshless solution instead.
Irradiance volumes, as introduced by Greger in 1998, is a technique where you take a scene, like this box,
Irradiance Volumes

and fill it with irradiance probes, which are placed in the empty space.
The method can be scaled to handle very large scenes by using an adaptive irradiance volume.
In order to reconstruct the irradiance at any point and direction in space, we need to traverse the volume structure and interpolate the irradiance at the query point.
We took irradiance volumes as a basis and augmented the structure with additional, precomputed light transport data.

For example, in this scene, the lighting data which we store in the global illumination volume, can be broken down to three separate components:
The irradiance from local light sources,
indirect sun light transport,
and sky light transport.
These three components are then combined with the direct lighting, volumetric lighting and screen-space effects.
The upside is that GI volumes don’t require UVs, they work right out of the box with LOD-models and they can be easily integrated with dynamic objects and volumetric lighting.
The only downside is that specular PRT is not really feasible, because having a decent angular resolution would require too much data.

Next, I'll talk about how we handle specular reflections.
Let's take a look at a simple scene.
When we are shading this pixel
we want to find out the incoming radiance from the reflection direction.
In case we have screen space information available, we can use screen space ray marching to get the reflected color and Ville will tell us more about this in the second part of the talk.
However, in cases where screen space information is not available,
we fall back to local reflection probes, like many other recent titles.
The question is, how should we blend between the reflection probes.

In this case, we have two local light probes with overlapping supports, but we have no idea which one we should use.
It seems clear that we should use the probe on the left, because it contains information about the reflection hit point, whereas the probe on the right does not.
Based on this observation, we would like to somehow encode the fact,
that for this particular point,
we should fall back to this reflection probe,
Reflection Probe Visibility

because it contains most information about the reflection environment around the point.
And we would like to do this for all points.
In order to do this, our main idea was to extend the global illumination volumes to store reflection probe visibility.
The way we do this, is, that for each voxel in space,
we generate a bunch of reflection ray samples,
Main idea: extend global illumination volumes to store reflection probe visibility.

and then trace visibility rays back to the reflection probes.
Reflection Probe Visibility

Main idea: extend global illumination volumes to store reflection probe visibility

Like this.
Main idea: extend global illumination volumes to store reflection probe visibility.
Main idea: extend global illumination volumes to store reflection probe visibility.
In this case we find out, that the right probe doesn’t contain any useful information.
Now we have a way to pick the best reflection probes for each voxel by taking the visibility into account.

The reflection probes are then looked up and interpolated per-pixel to provide smoothly varying specular reflections at runtime.
Here is an example using the precomputed reflection probe visibility.
And this is without the specular probes, using screen-space and ambient probe only.
Here you can see a side-by-side comparison between the specular probes and screen-space and ambient only.
Here you can see how the specular probes help ground the objects better in the scene.
Now that we know how to blend between the reflection probes, the next question we need to address, is where to place the probes.
It seems clear that we should avoid placing the probes too near to surfaces, because in that case, they don’t provide much information about the reflections around them.
Where to Place Reflection Probes?

Not too far from geometry

On the other hand, we should avoid placing the probes too far from the geometry, because, even though the probes can see almost all the surfaces, they would have terrible angular resolution.
Observation

Maximise visible surface area
Minimize distance to surface

Based on these observations, it seems that a good probe location will maximise the visible surface area, and minimise the distance to the visible surfaces.
This leads to a simple two step algorithm to place the probes.

First, we voxelize the scene and generate candidate probes in the empty space.
Then, we compute the visible surface area from the probes and rank the candidates.

Finally, we pick the K best probes based on these rankings according to our budget.
Here is an example of the automatic probe placement.

We typically have somewhere around one thousand specular probes per level.
To recap: for the global illumination, we have the local irradiance, indirect sun light transport, sky light transport, specular probe visibility, and the specular probe data.

The lighting is reconstructed per-pixel at runtime and combined with normal maps to obtain the large-scale lighting.
The specular probes are stored in a global atlas, but we need to put the rest of the data in some kind of volume data structure, that we can use at runtime the reconstruct the lighting.

Next, we are going to find out how we store and interpolate this GI data at runtime.
Possibly the easiest approach would have been to use volume textures or the newly available sparse texture support on the GPUs.

However, our index based compression ruled out volume textures and we didn’t want to commit to a GPU-feature which might not be available.
Related Work

Adaptive Volumetric Data Structures
- Irradiance Volumes [Greger98, Tatarchuk05]
- GigaVoxels [Crassin09]
- Sparse Voxel Octrees [Laine and Karras 2010]
- Tetrahedralization, e.g., [Cupisz12], [Bentley14], [Valient14]
- Sparse Voxel DAGs [Kämpe13]
- Open VDB [Museth13]

Hardware solution was a no-go, so we started to look at existing software solutions in the field.

None of the existing approaches was directly usable for our purposes, but we drew a lot of inspiration and ideas from each of them.
Our approach is heavily inspired by Open VDB and is based on implicit spatial partitioning and a large branching factor.
Adaptive Voxel Tree

*Implicit* spatial partitioning
Branching factor of 64
Multi-scale data

To see how it works, let's take a look at an example scene.
Given a set of triangles and their bounding box, we build the tree by subdividing the AABB into a regular 4x4x4 grid of children.
Adaptive Voxel Tree

Implicit spatial partitioning
Branching factor of 64
Multi-scale data

For each child node, we mark it as solid if it intersects any of the triangles.
Adaptive Voxel Tree

**Implicit** spatial partitioning
Branching factor of 64
Multi-scale data

This process is continued
Adaptive Voxel Tree

**Implicit** spatial partitioning
Branching factor of 64
Multi-scale data

until we reach the maximum level of the tree.
The resulting voxel tree is stored as a single linear array of voxel tree nodes in memory.
Each node consists of a 64-bit child mask and an offset to its children.

The nodes can have a variable number of children, which are tightly packed together at the child block offset.
The child mask contains one bit for each voxel in the world space voxel grid of the node.
If the voxel is solid, then the corresponding child mask bit is set.
To traverse the tree, we need to access the children of each node. To see how this works, let’s take a look at an example.

Let’s say we want to access the orange child with index seven.
To locate the child node position in the array, we make use of the fact that the children are tightly packed together.
To locate a particular child in the child block, we need to count the number of set bits in the child mask before it.

In this case, there are three solid children before the orange one.
Now we can find the index of the child in the array by adding the bit count to the child block offset.
The interesting observation here is, that we can use exactly the same child index as the payload index.

This means that we can attach different types of payload data to a voxel without storing any extra links.
For example, we store the light transport matrices, local light irradiance and specular probe visibility in the payload arrays.

Furthermore, it is easy to add or remove payload data without modifying the tree structure.
What About Leaf Nodes?

Leaf nodes are **implicit**: they only show up in the child masks of their parent voxels

**Compact** trees encoding only the **topology**

Only a **few hundred kilobytes** for an entire level

It is worth noting that the leaf nodes are not explicitly stored anywhere: they are fully described by the child mask of their parent nodes.

The implicit encoding of the payload indices lead to compact trees, which take only a few hundred kilobytes per level.
First Level Lookup

First level of the tree can have **arbitrary** dimensions

We use a **dense** grid of 8x8x8 meter cells to guarantee coverage for large dynamic objects

Similar to OpenVDB, the first level of the tree can have arbitrary dimensions. Instead of using a spatial hash, we store the first level of the tree in a dense lookup grid.
And this is how the voxel tree looks when overlaid on top of a scene.
Seamless Interpolation

Dynamic and static objects lit by same data
Need seamless interpolation everywhere

We use the same GI data to light both dynamic and static objects, and this is the key to achieving a consistent look across the board.

I'll skip the details for brevity, but you can find more information in the downloadable slides.
Let's take a look at an example.

In this case the query point is in a solid leaf voxel.
All the data points in the trilinear neighbourhood of the query point reside on a single hierarchy level and we can perform the usual trilinear interpolation.
Seamless Interpolation

However, the general case, when query point is in an empty leaf, is more interesting.
A trivial solution to this case would be to dilate the voxel tree and precompute the interpolated data values between the hierarchy levels in the dilated voxels.
Dilated tree has the best runtime performance and is a perfectly valid option to use if you can.

However, doing a dilation at the leaf level can create a lot of new voxels, and in our case, the data size almost doubled so we had to find another way.
In order to perform trilinear interpolation at the query point, we must look at the trilinear neighborhood around the point.

In the general case, the trilinear neighborhood may contain points from multiple hierarchy levels.

For example, in this case, the orange points in the trilinear neighborhood can be directly obtained from the neighboring leaf voxels but the what about the red one?
The red data point must be interpolated from the next hierarchy level.

In general, this process could lead to a long recursion, possibly all the way up to the root node of the tree.
To prevent this, we impose a special structure on our voxel trees by performing a partial dilation only in the upper levels of the tree. This increases the total memory usage only by a few percent.

After the partial dilation, each trilinear neighbourhood contains data points from exactly one or two hierarchy levels, which basically avoids the costly recursion.

So, to do seamless interpolation, we construct the trilinear neighbourhood of the query point, look up each data point in the voxel tree and, if necessary, interpolate the missing data from the parent node.
And this is how the seamless interpolation looks in game.
Here you can see how the dynamic character and the dynamic barrel blend in seamlessly with the static environment.
On the other side you can see a larger scale transition, that we typically use to control the level of detail.
If we look at the final image, it is not completely obvious that the barrel and the character are dynamic objects.
To avoid light leaking, we multiply the trilinear interpolation weight with a geometry term, which takes the surface normal into account.

The combined weight is still continuous, but, as you can see on the right, it fixes a lot of the typical light leaking issues with volume interpolation.
So far we have talked about a single voxel tree structure.

In order to support large levels and streaming, we divide the world space into 128-by-128 meter cells.

Each world space cell contains its own voxel tree and a full set of GI data.
On the GPU, we have global arrays per data type and the arrays are streamed in and defragmented on the fly.
This is the final image with all the global illumination features enabled.
And this is without the GI, using only screen space effects and ambient lighting.
Here you can see a side-by-side comparison between global illumination and screen-space plus ambient.

In the bottom row you can see a lighting only image.
Here you can see some of the large-scale features that come from the GI.
For each 128-meter cell, we store a maximum of 65K diffuse probes, which, in terms of data, is roughly comparable to a 256 by 256 light map.

In total, all the diffuse GI data takes around 30Mb-50Mb per level.

The current, unoptimised implementation takes more than 3ms to evaluate the precomputed light transport, look up and uncompress the data and perform the seamless hierarchical interpolation per-pixel.
Part of this cost is offset by the fact that we can precompute soft area lighting by placing reflectors in the scene instead of using dynamic fill lights.

For example, the image on the right is lit only by the local, reflected spot light.

In order to validate and compute the GI, we use a path tracer.
Here you can see a real-time view of the global illumination resulting from a single spot light.

The bottom row contains a side-by-side comparison between the ground truth reference and the real-time indirect illumination. As you can see, the real-time indirect illumination is visually quite close approximation to the path traced reference indirect.

We also use the GI to do volumetric lighting.
Here you can see participating media, which is lit by indirect illumination.
Here is a side-by-side comparison between global illumination and constant ambient.
Here you can see how the volumetric GI makes the image work better.
To recap, I presented a unified approach to GI based on a sparse voxel structure and a fully automatic specular probe system.

And with that, I’ll let Ville to talk about how we use screen-space effects to complement the large-scale lighting features.
Talk Outline

**Part I:** Large-scale lighting

**Part II:** Screen space lighting

SIGGRAPH 2015: Advances in Real-Time Rendering course
We prefer screen-space lighting when possible: fully dynamic, and finer scale-detail. Geometry outside of the screen and behind the first depth layer is unknown to screen-space methods, which is where we fall back to GI. Therefore we need the screen-space methods to detect when they can reliably supply screen-space lighting: produce occlusion for GI. For the occluded parts, lighting is complemented from screen-space.
We treat diffuse and specular separately, and first present our diffuse solution.
Diffuse GI data is in principle multiplied with the values of the image on the left hand side. The screen-space color, shown to the right, is then added.
Screen-Space Ambient Occlusion

Based on Line-Sweep Ambient Obscurance [Timonen2013]:
LSAO locates most contributing occluders

For occlusion, we have SSAO which is based on LSAO.
Refer to the LSAO paper for full description, which is outside of this presentation’s scope.
In summary, LSAO gives you dominant occluders along a set of discrete directions.
The occluders are found from the whole depth buffer, and the AO effect can therefore span the entire screen.
Screen-Space Ambient Occlusion

We scan in 36 directions, long steps (~10px) and short line spacing (~2px apart)

— Scheduling friendly for the GPU
— Scan is 0.75ms on Xbox One at 720p

These are our LSAO settings. Despite the long steps, average distance to the nearest step is less than 3 px.
As opposed to the original regular sampling in LSAO, we jitter steps along each line for roughly even sample distribution.
To fill in the “gaps” of the longer LSAO samples, we take one traditional near field sample per pixel per direction which is roughly half-way from the receiver to the nearest sample position of the sweep.

When we evaluate AO, we sample normal to clamp occluders to the visible hemisphere. This way we get AO that respects fine-scale normal variations.
Screen-Space Ambient Occlusion

36 directions too expensive to gather per pixel

- Interleave on a 3x3 neighborhood (4 dirs/pixel)
- Gather using a depth and normal aware 3x3 box filter

We cycle through the 36 directions on 9 pixel neighborhoods.
Geometry is interpreted much the same way as in Horizon-Based Ambient Occlusion and is therefore a more correct approximation than random sampling based SSAO methods which don’t account for sample inter-occlusion. As opposed to HBAO, our geometry scan takes care of the exhaustive occluder search and covers unbounded range in screen-space.

Our final image is 1080p but screen-space lighting is evaluated at 1280x720.
Works nicely across different scales; does not overdarken nor have halos
Temporal filtering is normally used to mitigate noise, but in our case we alleviate banding: We cycle through 4 sets of different 36 directions and effectively get 144 directions.
It's not necessary, but it helps. Roughly 0.2ms
Screen-Space Diffuse Lighting

Now that the occlusion is covered, moving onto lighting
We reproject the points from which we calculate occlusion to the previous frame, and sample its color with a MIP that roughly corresponds to the scan sector’s width at the occluder’s distance.
Screen-space diffuse lighting contribution to the left, final image to the right.
Notice self-occlusion below the pallet and next to the pile of wire.
Without color albedos
Without color albedos
Without color albedos
Without color albedos
Without color albedos
Without color albedos
Screen-Space Ambient Occlusion OFF
Screen-Space Diffuse Lighting OFF
Screen-Space Ambient Occlusion ON
Screen-Space Diffuse Lighting ON
Screen-Space Reflections and Occlusion
The same high-level logic for specular light as well: produce occlusion to cull specular GI (shown to the left) and provide screen-space lighting for the occluded areas (their contribution to the right). Both diffuse and specular are evaluated for all surfaces. We have a single material shader with parameterization for diffuse and specular albedo, roughness, etc.
Screen-Space Reflections

1 ray per pixel from GGX distribution, evaluated for all surfaces

- Linear search (7 steps)
- Step distances form a geometric series

Based on roughness, we draw rays from GGX distribution (1 per pixel).
We only perform linear search as it’s most important not to miss occluders.
Bilinear refinement not that necessary, accuracy is sufficient with proper occluder interpolation.
Steps scaled to always end at the screen edge, and have denser sampling near the receiver.
Screen-Space Reflections

Treating the depth buffer samples

Need to support varying roughness
— Calculate cone coverage

Need to suit both occlusion and color sampling
— Also find a single color sample location

Now that the sample locations have been decided, the two important remaining aspects are choosing a proper depth field thickness, calculating reflection cone coverage, and find a sample location for color.
Screen-Space Reflections

Depth thickness =
\[ a + b \times \text{(distance along the ray)} \]

Depth field extends to/from camera, not along view z!

Constant and a linear term in thickness.
Remember that depth field extends along camera direction, otherwise you'll get issues especially near the view frustum edges.
As opposed to matching the depth thickness to how thick objects on screen might be, we feel it's more important to match the linear term to step sizes as to avoid rays slipping through solid geometry.
Although, depending on surface orientation, there will always be some gaps
Screen-Space Reflections

For occlusion, calculate max coverage of the cone

Clamp the cone’s lower bound to surface tangent!

Finally just iterate over the samples and calculate how much of the cone is occluded.
Notice that you’ll get self-occlusion unless you clamp the cone’s lower bound.
We use temporal filtering here also.
Screen-Space Reflections

For **color**, we need a single sample location.

First, we pick the sample that covered most of the cone.

We use the same samples for color sampling.
Screen-Space Reflections

Aim the reflection ray towards the center of the coverage

And intersect with the line between the last 2 samples
Screen-Space Reflections

Low sample density: interpolate towards camera direction (in blue)

We're using 7 samples per pixel (low sample density), and the line between 2 consecutive samples will usually be too gently sloping. On average we can get a better match to geometry if we take the half vector between camera and 2 previous samples instead.
A fail case is when the previous depth sample was above (on the other side) the reflection ray; don’t interpolate, use camera direction instead.
Reproject the intersection point to previous frame and sample
(MIP level = roughness * hit distance)
Total render time for SSRO (0.8ms) and SSR (0.5ms) is 1.3ms
Screen-Space Ambient Occlusion
1.4ms @ 720p
Screen-Space Reflection Occlusion OFF
Screen-Space Reflections OFF
SIGGRAPH 2015: Advances in Real-Time Rendering course
Screen-Space Reflection Occlusion ON
Screen-Space Reflections OFF
As described before, we support varying smoothness
Often the largest issue is not taking enough samples and therefore skipping over geometry or interpolating an inaccurate intersection point. If you use compute shaders, you can figure out if the reflection ray direction/origin within the pixel's neighborhood is similar enough to have the neighborhood collaborate on the search. If so, Interleave sample distances, and take the nearest found hit distance from neighborhood.

You don't get box artifacts if you use original ray directions/origins; only replace hit distance.
Seems to do good, perhaps this can be generalized further.
Thank You!

Acknowledgments

Tatu Aalto
Janne Pulkkinen
Laurent Harduin
Natalya Tatarchuk
Jaakko Lehtinen
References