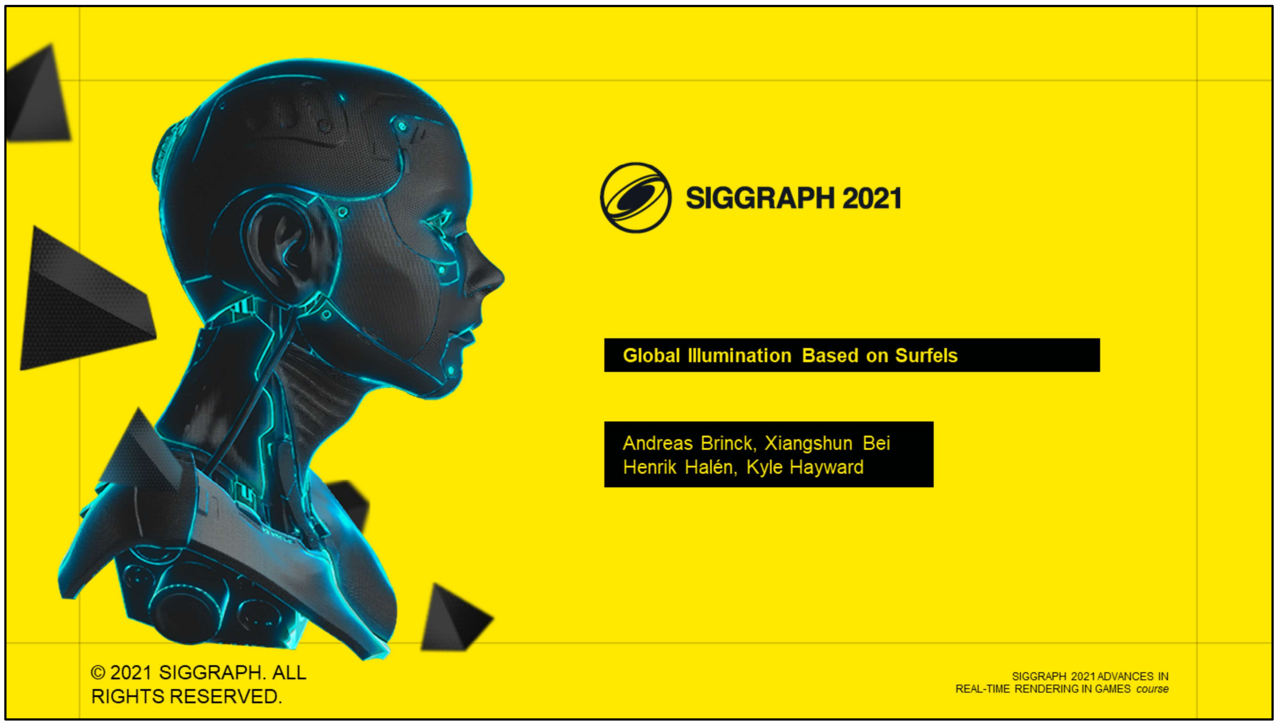
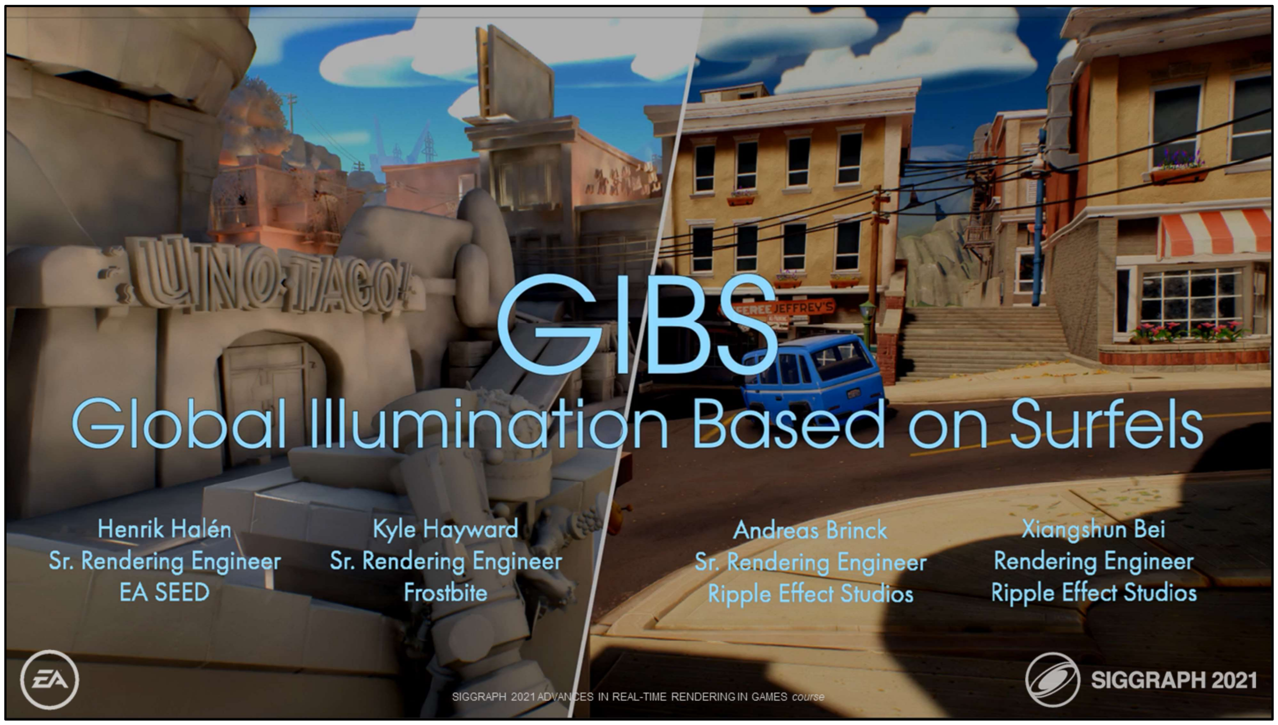


SIGGRAPH 2021

Global Illumination Based on Surfels

Andreas Brinck, Xiangshun Bei
Henrik Halén, Kyle Hayward



GIBS
Global Illumination Based on Surfels

Henrik Halén
Sr. Rendering Engineer
EA SEED

Kyle Hayward
Sr. Rendering Engineer
Frostbite

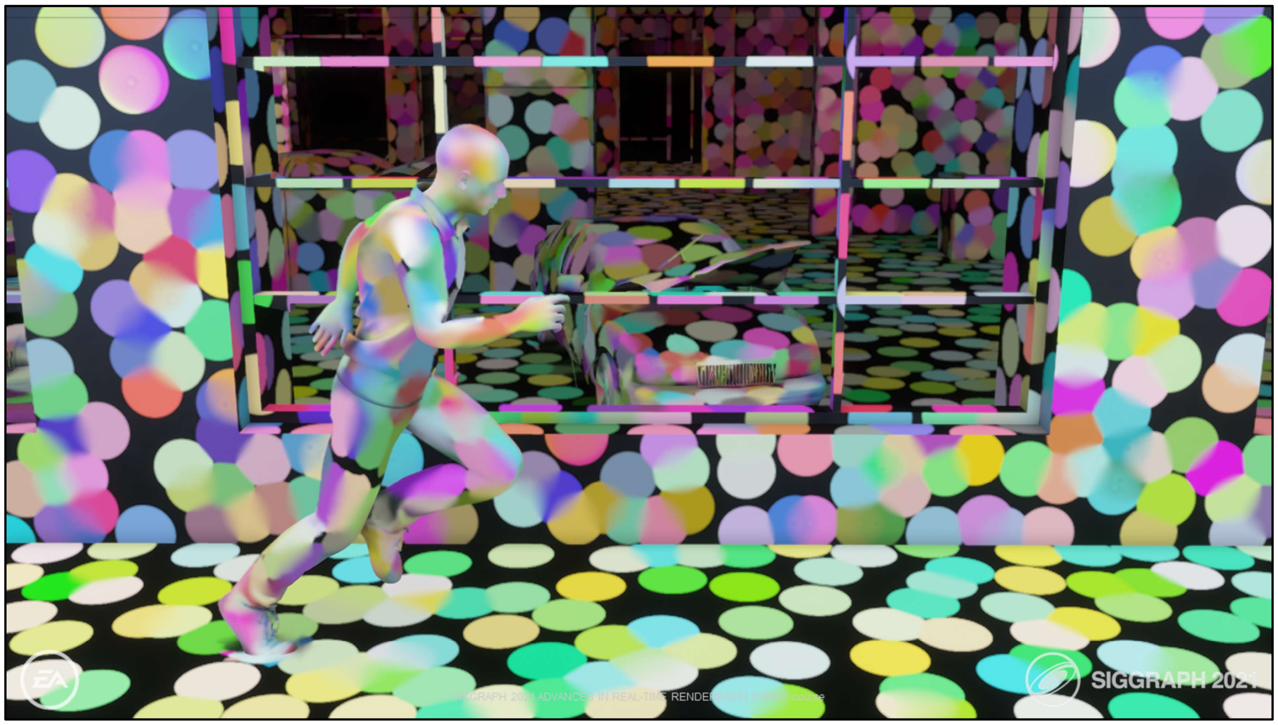
Andreas Brinck
Sr. Rendering Engineer
Ripple Effect Studios

Xiangshun Bei
Rendering Engineer
Ripple Effect Studios

SIGGRAPH 2021

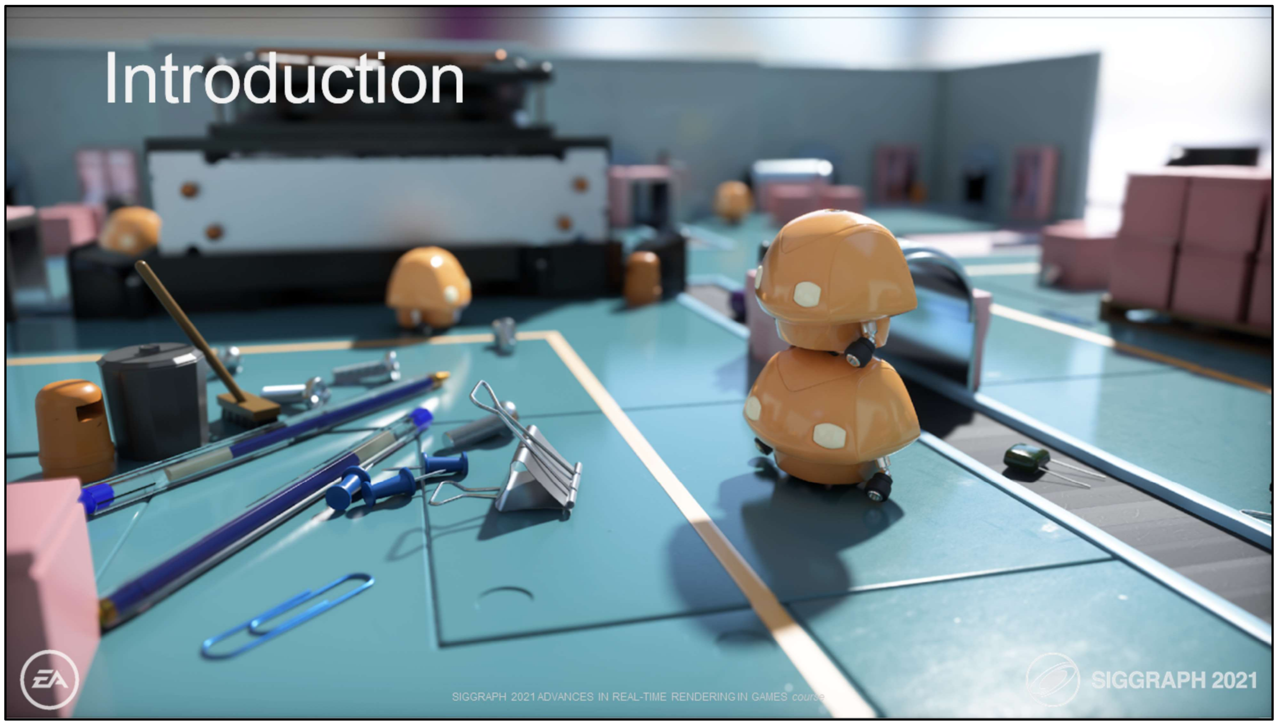
I am Henrik Halen from Electronic Art’s SEED R&D group. Today I will be presenting with Kyle Hayward from Frostbite.

This presentation and much of the technology behind it, has been put together by myself, Kyle Hayward from Electronic Art’s Frostbite, Andreas Brinck and Xhiangshun Bei from EA’Ripple Effect studios in Los Angeles.



“Global Illumination based on Surfels (GIBS) is a solution for calculating indirect diffuse illumination in real-time. The solution combines hardware ray tracing with a discretization of scene geometry, to cache and amortize lighting calculations across time and space. It requires no pre-computation, no special meshes, and no special UV sets. GIBS supports high fidelity lighting while accommodating content of arbitrary scale.“

菓子
お土産
魚市場
SAITO
ていしょく
EASPORTS MAGAZINE
Left and Right Hemispheres
A Guide to your Brain

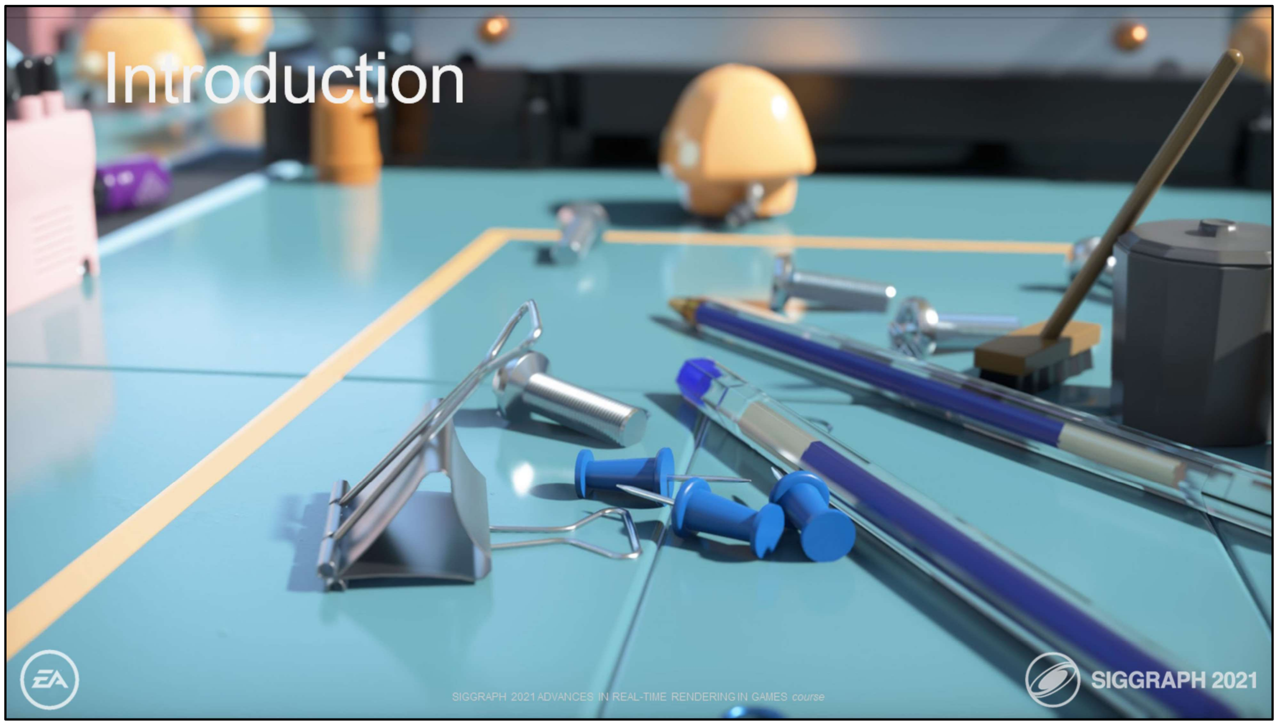


Introduction

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES course

SIGGRAPH 2021

The basic GIBS algorithm was implemented in 2018, as part of EA SEED's PICA PICA ray tracing showcase. It was developed for that demo primarily by Tomasz Stachowiak.

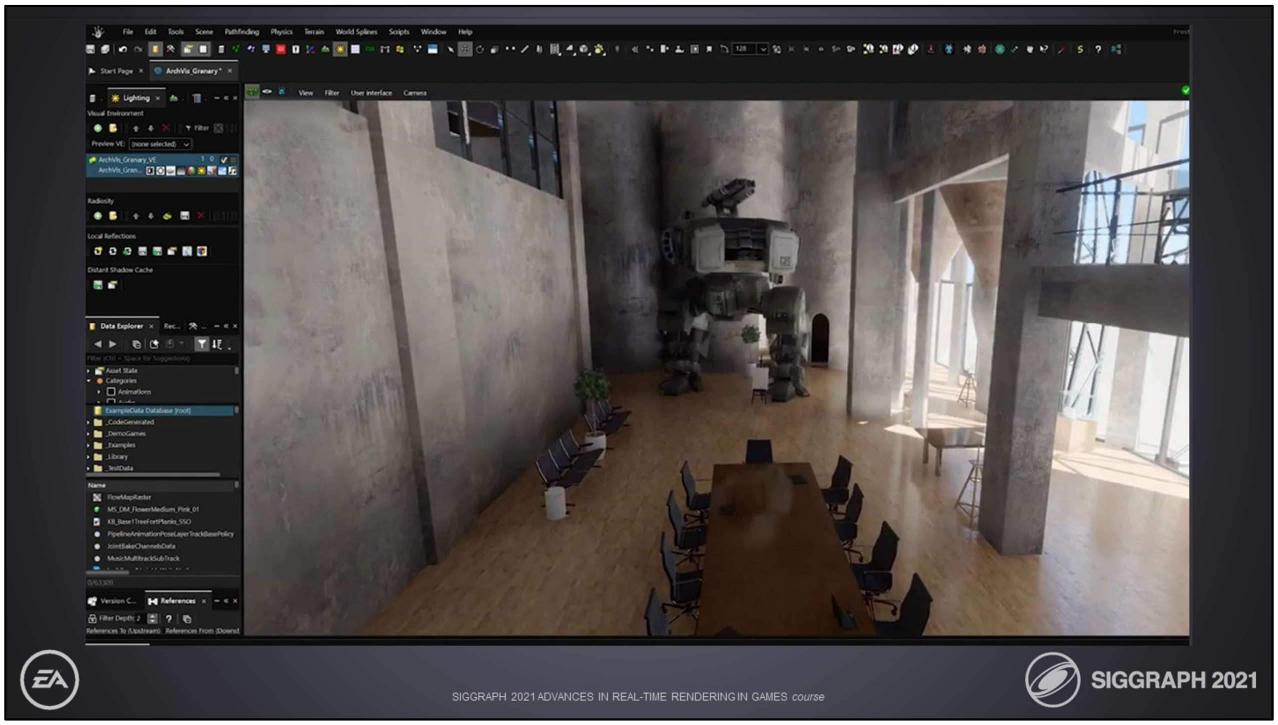


Introduction

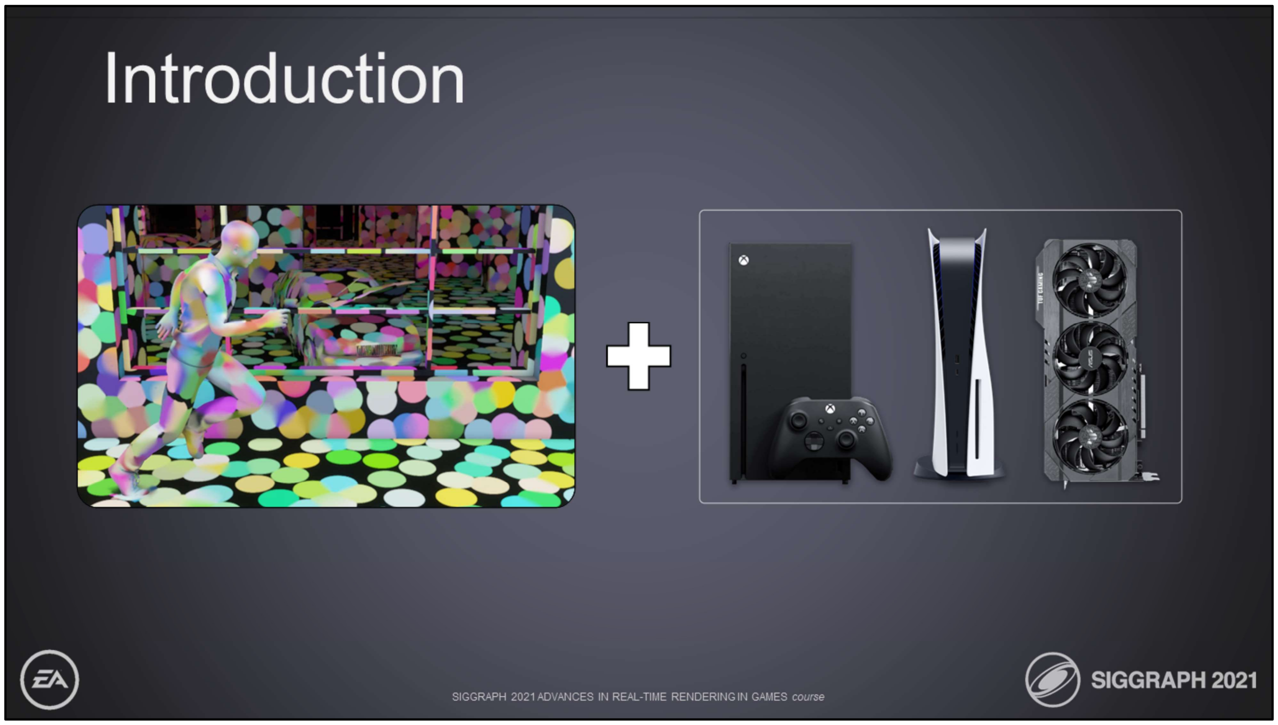
SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

Since then the algorithm has been significantly extended and optimized to handle any geometry, including skinned characters and large environments, while improving convergence times and quality.


File Edit Tools Scene Pathfinding Physics Terrain World Splines Scripts Window Help
Start Page
ArchVis_Granary*
Lighting
Visual Environment
Radiosity
Local Reflections
Distant Shadow Cache
Data Explorer
View Filter User interface Camera
SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES course
SIGGRAPH 2021

The algorithm is part of the suite of tools available to developers and teams throughout EA as part of the Frostbite engine.



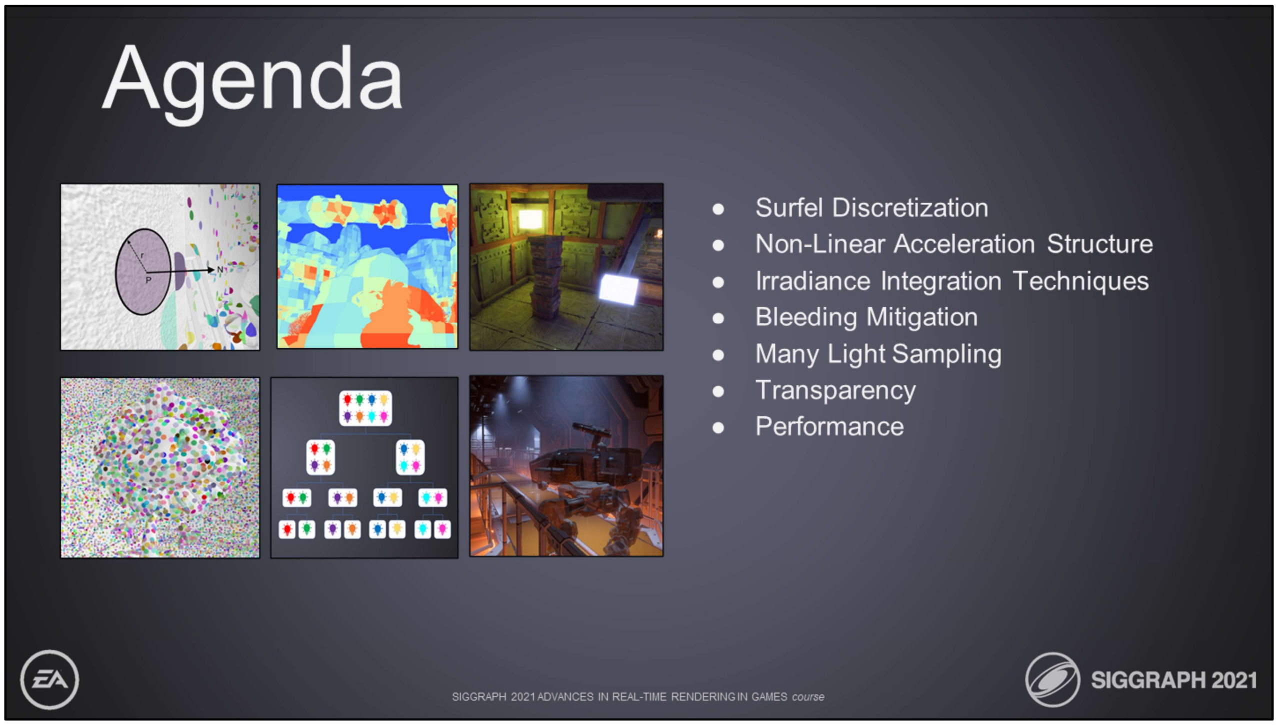
Introduction

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

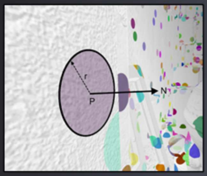
SIGGRAPH 2021

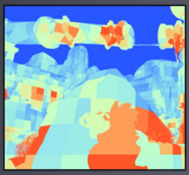
While hardware accelerated ray tracing is widely available on consoles and PC graphics hardware, shooting rays is still one of the most costly operations for that hardware. Recent work has shown that diffuse global illumination is solvable in real-time using high-end hardware at a significant performance penalty.

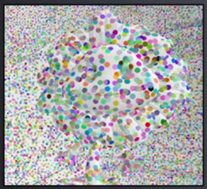
By minimizing where and when rays are sent, we hope to make real-time global illumination possible for mainstream hardware and titles.



Agenda



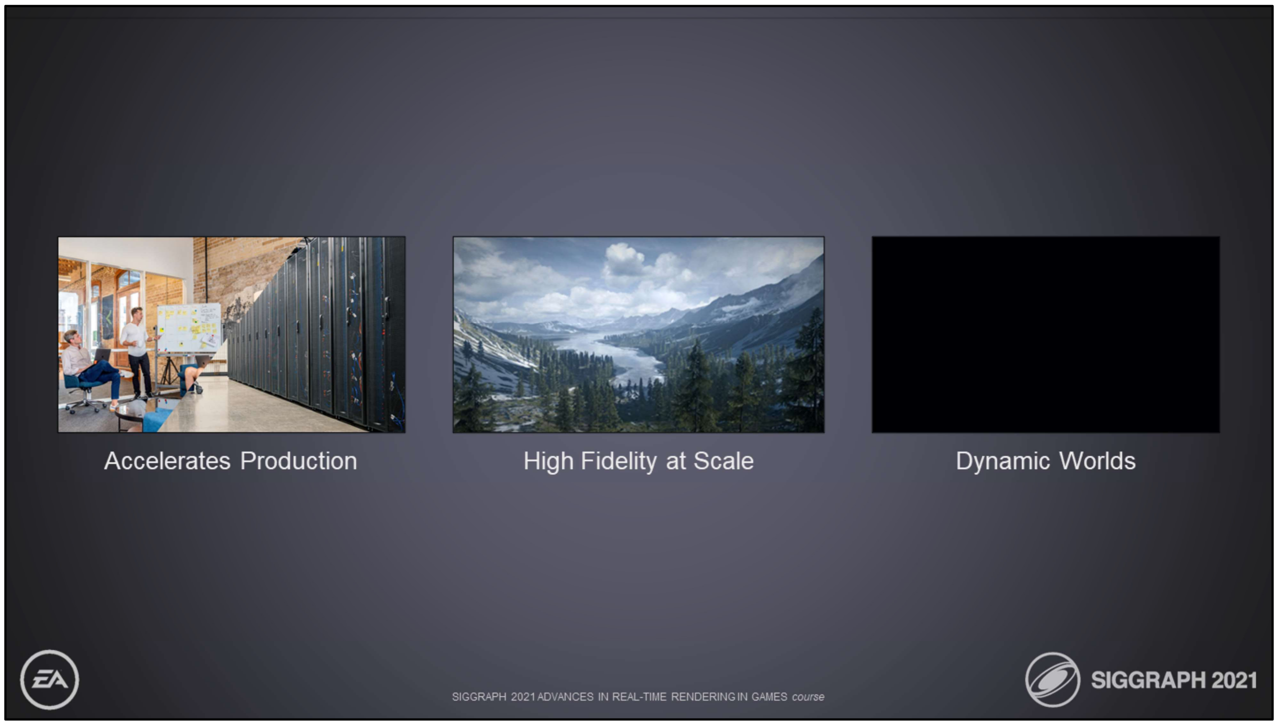


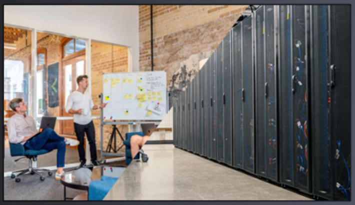


- Surfel Discretization
- Non-Linear Acceleration Structure
- Irradiance Integration Techniques
- Bleeding Mitigation
- Many Light Sampling
- Transparency
- Performance

Today we will talk about how we discretize the scene into surfels.
We will describe our non-linear acceleration structure that makes this possible.
We describe how we integrate irradiance, how we mitigate artifacts, sample scenes with many lights, and handle transparency.
Finally we will look at performance numbers.
Before we begin, we should mention that this technique is still very much in development, and as such we expect to see significant improvements as development continues.

So let’s get into the algorithm. In GIBS, surfels discretize the scene on the fly, as geometry that needs indirect lighting comes into view. Once surfelized, the scene efficiently accumulates and caches irradiance. We believe this is a good fit for global illumination. In contrast to probe grid solutions, we perform and cache ray tracing operations exactly where they are needed. In contrast to screen space filtering techniques, the surfel cache is persistent as surfels go in and out of view, saving us from doing the work again, Surfels are resolution independent, allowing for a scalable solution in terms of performance and quality. It is also fully dynamic, allowing complex lighting interaction in scenes where everything can change.




Accelerates Production

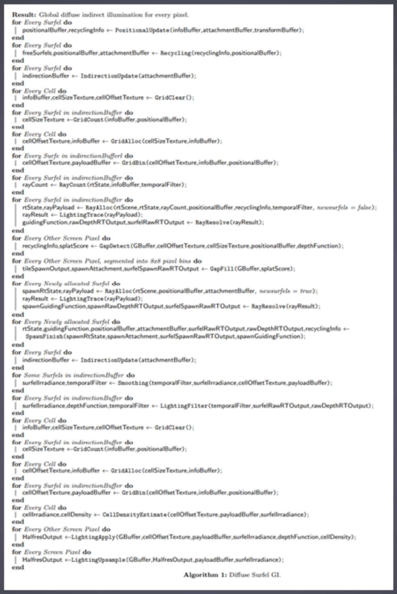
High Fidelity at Scale

Dynamic Worlds

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

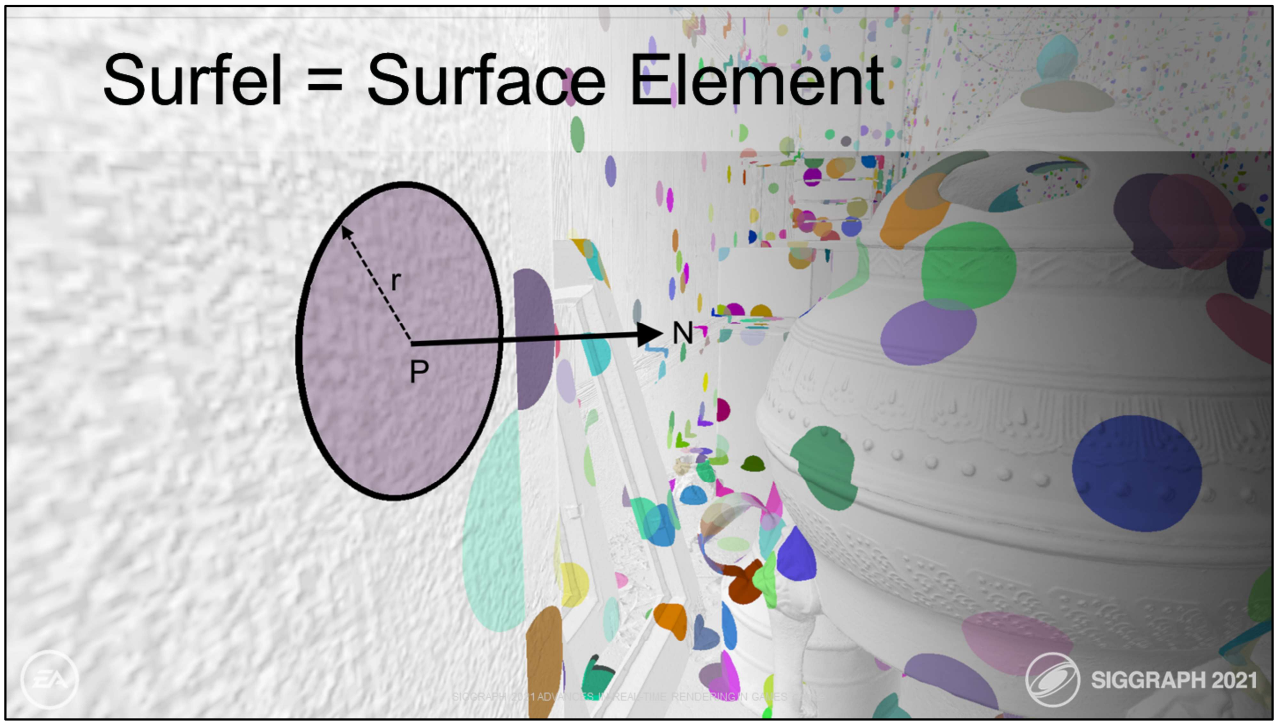
SIGGRAPH 2021

GIBS accelerates production by removing the need for time consuming bakes, and the setup of traditional lighting techniques, such as the creation of special meshes or UV set.
We support environments of arbitrary scale, and fully dynamic geometry and worlds.



```
Result: Global diffuse indirect illumination for every pixel.
for Every Surfel do
    positionalBuffer,recyclingInfo ← PositionalUpdate(infoBuffer,attachmentBuffer,transformBuffer);
end
for Every Surfel do
    freeSurfels,positionalBuffer,attachmentBuffer ← Recycling(recyclingInfo,positionalBuffer);
end
for Every Surfel do
    indirectionBuffer ← IndirectionUpdate(attachmentBuffer);
end
for Every Cell do
    infoBuffer,cellSizeTexture,cellOffsetTexture ← GridClear();
end
for Every Surfel in indirectionBuffer do
    cellSizeTexture ←GridCount(infoBuffer,positionalBuffer);
end
for Every Cell do
    cellOffsetTexture,infoBuffer ← GridAlloc(cellSizeTexture,infoBuffer);
end
for Every Surfe in indirectionBufferl do
    cellOffsetTexture,payloadBuffer ← GridBin(cellOffsetTexture,infoBuffer,positionalBuffer);
end
for Every Surfel in indirectionBuffer do
    rayCount ← RayCount(rtState,infoBuffer,temporalFilter);
end
for Every Surfel in indirectionBuffer do
    rtState,rayPayload ← RayAlloc(rtScene,rtState,rayCount,positionalBuffer,recyclingInfo,temporalFilter, newsurfels = false);
    rayResult ← LightingTrace(rayPayload);
    guidingFunction,rawDepthRTOutput,surfelRawRTOutput ← RayResolve(rayResult);
end
for Every Other Screen Pixel do
    recyclingInfo,splatScore ← GapDetect(GBuffer,cellOffsetTexture,cellSizeTexture,positionalBuffer,depthFunction);
end
for Every Other Screen Pixel, segmented into 8x8 pixel bins do
    tileSpawnOutput,spawnAttachment,surfelSpawnRawRTOutput ← GapFill(GBuffer,splatScore);
end
for Every Newly allocated Surfel do
    spawnRtState,rayPayload ← RayAlloc(rtScene,positionalBuffer,attachmentBuffer, newsurfels = true);
    rayResult ← LightingTrace(rayPayload);
    spawnGuidingFunction,spawnRawDepthRTOutput,surfelSpawnRawRTOutput ← RayResolve(rayResult);
end
for Every Newly allocated Surfel do
    rtState,guidingFunction,positionalBuffer,attachmentBuffer,surfelRawRTOutput,rawDepthRTOutput,recyclingInfo ←
     SpawnFinish(spawnRtState,spawnAttachment,surfelSpawnRawRTOutput,spawnGuidingFunction);
end
for Every Surfel do
    indirectionBuffer ← IndirectionUpdate(attachmentBuffer);
end
for Some Surfels in indirectionBuffer do
    surfelIrradiance,temporalFilter ← Smoothing(temporalFilter,surfelIrradiance,cellOffsetTexture,payloadBuffer);
end
for Every Surfel in indirectionBuffer do
    surfelIrradiance,depthFunction,temporalFilter ← LightingFilter(temporalFilter,surfelRawRTOutput,rawDepthRTOutput);
end
for Every Cell do
    infoBuffer,cellSizeTexture,cellOffsetTexture ← GridClear();
end
for Every Surfel in indirectionBuffer do
    cellSizeTexture ←GridCount(infoBuffer,positionalBuffer);
end
for Every Cell do
    cellOffsetTexture,infoBuffer ← GridAlloc(cellSizeTexture,infoBuffer);
end
for Every Surfel in indirectionBuffer do
    cellOffsetTexture,payloadBuffer ← GridBin(cellOffsetTexture,infoBuffer,positionalBuffer);
end
for Every Cell do
    cellIrradiance,cellDensity ← CellDensityEstimate(cellOffsetTexture,payloadBuffer,surfelIrradiance);
end
for Every Other Screen Pixel do
    HalfresOutput ←LightingApply(GBuffer,cellOffsetTexture,payloadBuffer,surfelIrradiance,depthFunction,cellDensity);
end
for Every Screen Pixel do
    HalfresOutput ←LightingUpsample(GBuffer,HalfresOutput,payloadBuffer,surfelIrradiance);
end
```

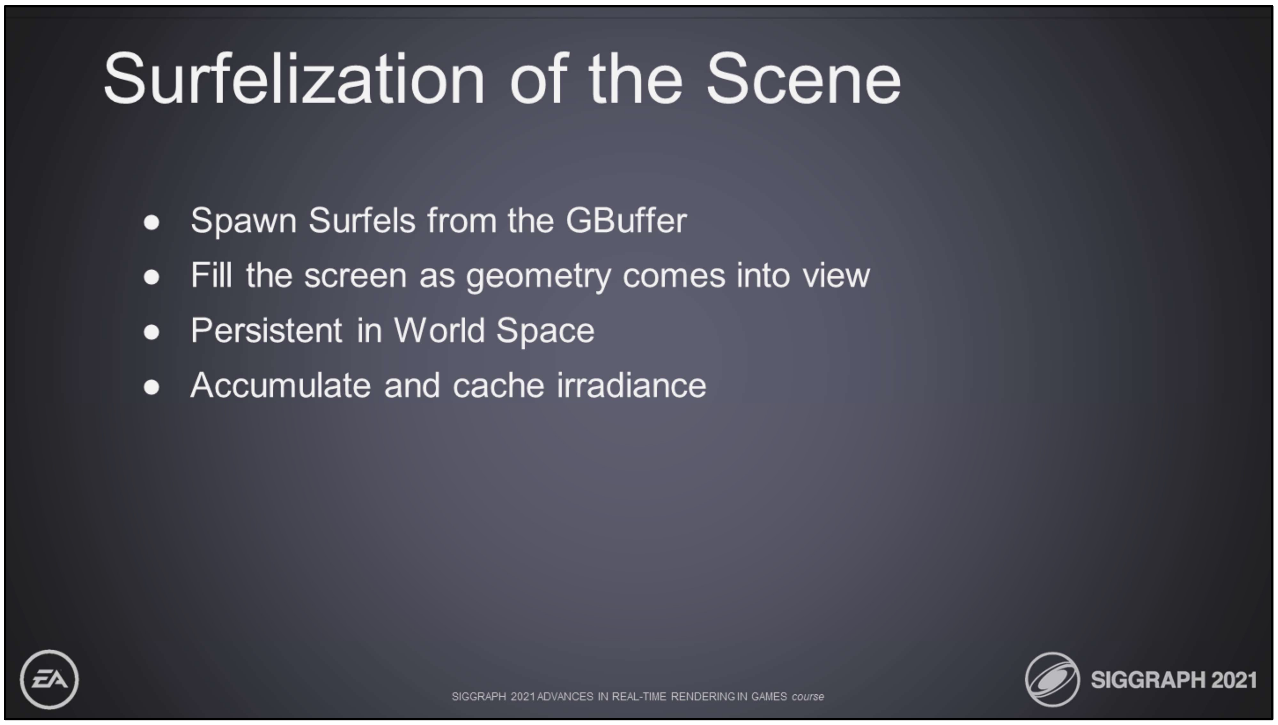
Algorithm 1: Diffuse Surfel GI.



Surfel = Surface Element

In the same way an image can be discretized by dividing it into pixels, we can also discretize a geometric surface.

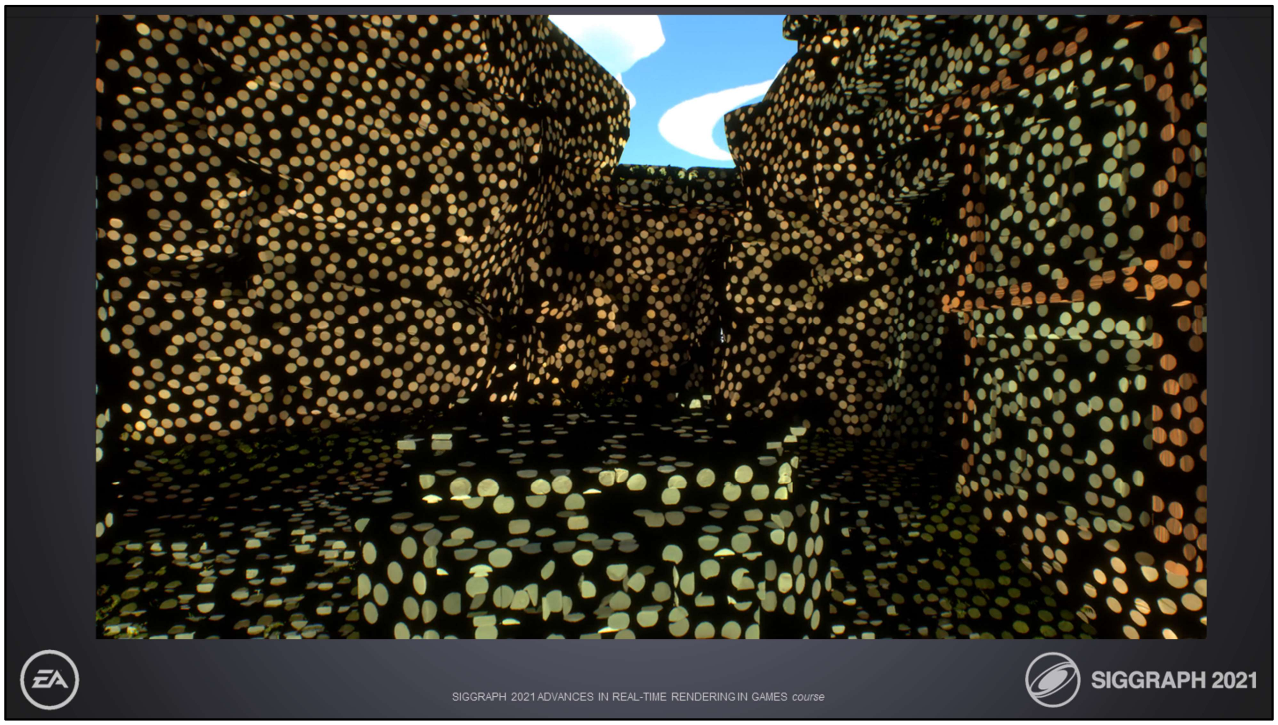
One way of doing this is by using surfels; shorthand for surface elements. A surfel is defined by a position, a radius, and a normal, and approximates a small neighborhood of a surface near the given position.



Surfelization of the Scene

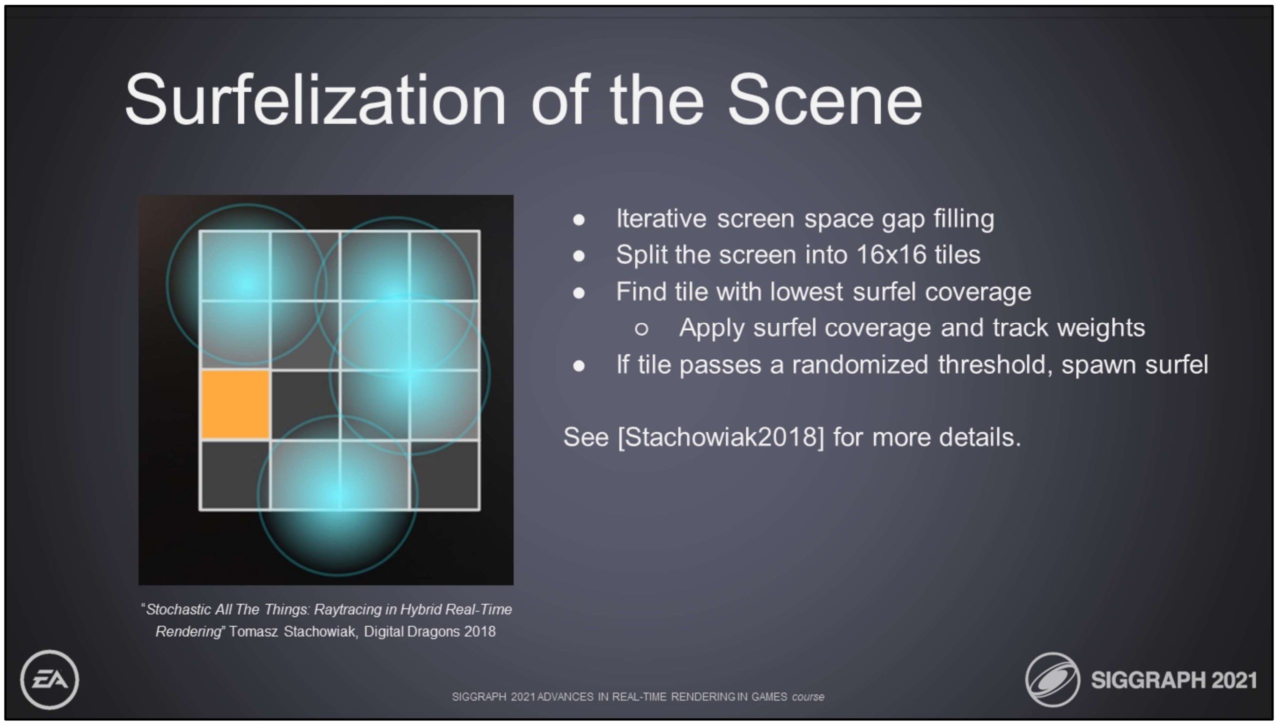
- Spawn Surfels from the GBuffer
- Fill the screen as geometry comes into view
- Persistent in World Space
- Accumulate and cache irradiance

So how do we discretize the scene with surfels? We spawn them from the GBuffer as geometry comes into view.
After that, surfels are persistent, which allows us to efficiently accumulate radiance over time, and to cache those costly operations without throwing the work away.

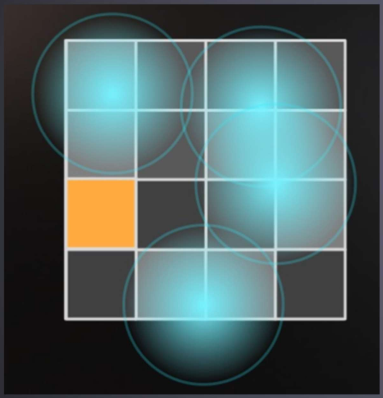


Here we have slowed down this surfelization from the gbuffer so you can see it happen in real time. As geometry comes into view the spawning algorithm fills any gaps in coverage that appears.

Normally this is fast enough to appear immediate to the user, you never really see any gaps in coverage.



Surfelization of the Scene



"*Stochastic All The Things: Raytracing in Hybrid Real-Time Rendering*" Tomasz Stachowiak, Digital Dragons 2018

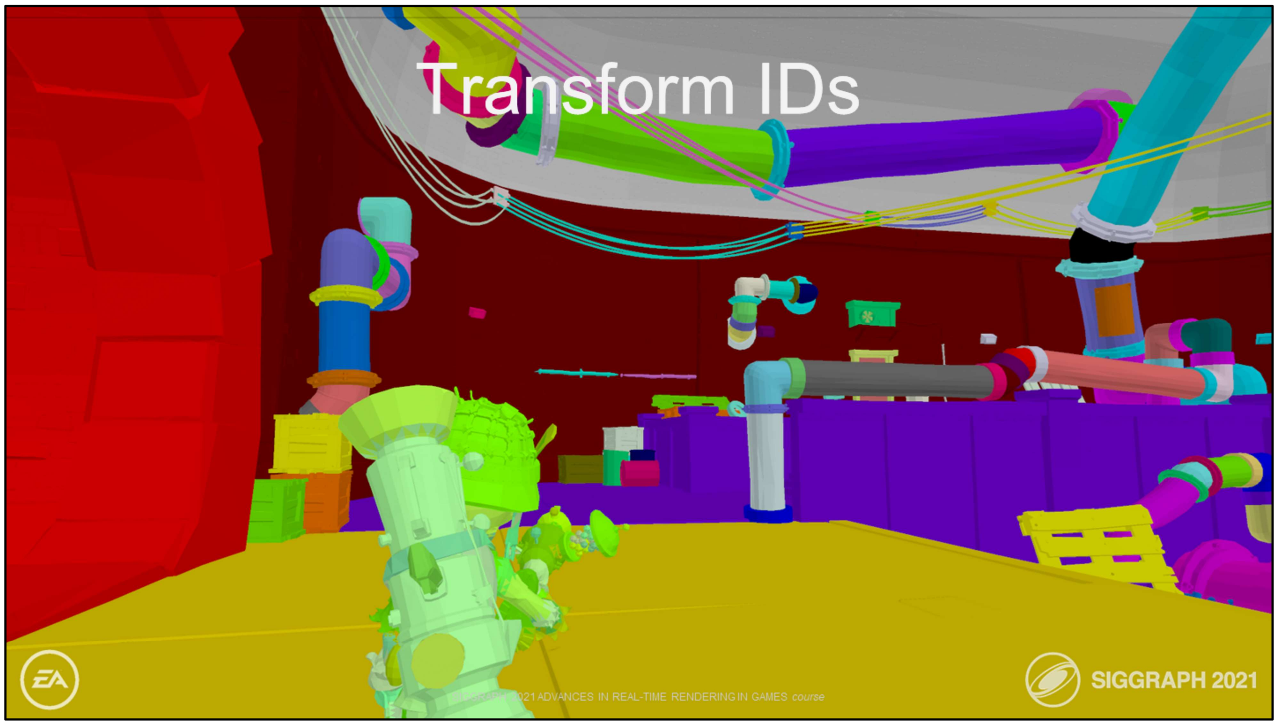
- Iterative screen space gap filling
- Split the screen into 16x16 tiles
- Find tile with lowest surfel coverage
  - Apply surfel coverage and track weights
- If tile passes a randomized threshold, spawn surfel

See [Stachowiak2018] for more details.

SIGGRAPH 2021

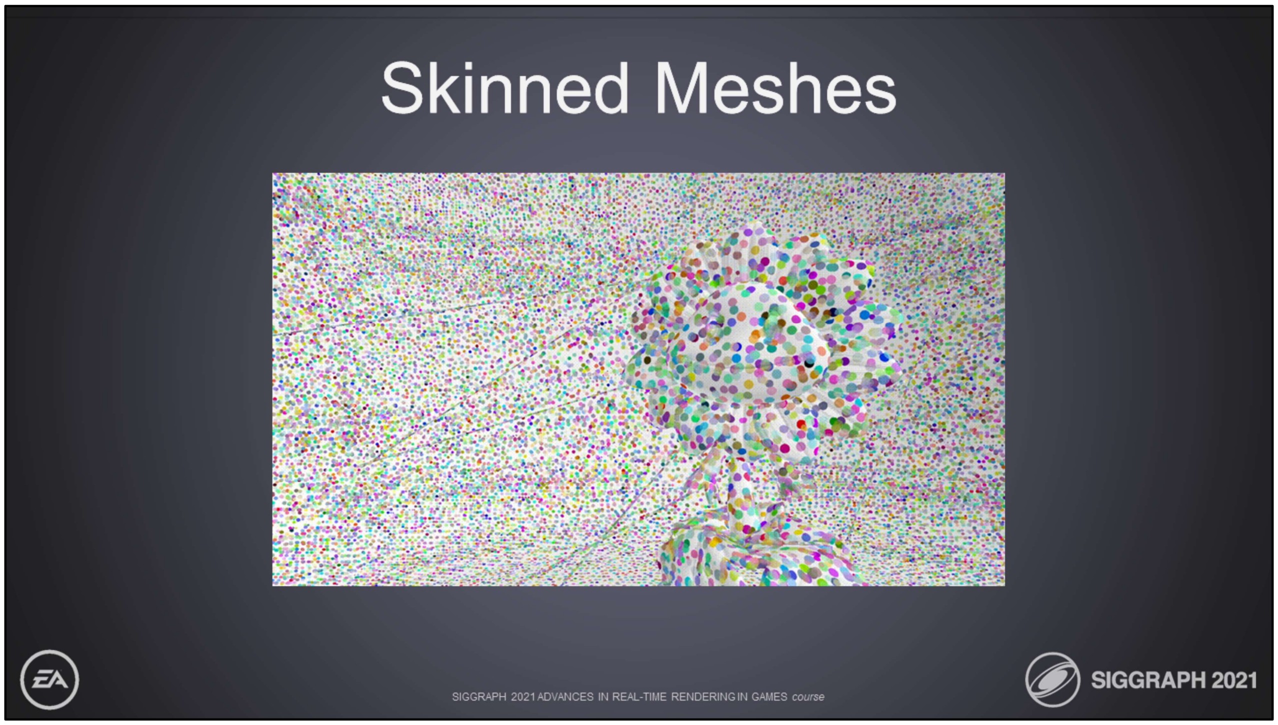
The actual spawning algorithm works by filling gaps as they are seen in screen space. The screen is split into 16x16 texel tiles, each tile finds the texel which has the least coverage currently. If this coverage passes a randomized threshold we spawn a new surfel using the geometric information from the GBuffer. Once a certain amount of coverage is achieved, no more surfels are spawned for that tile. For more details on spawning itself, I recommend Thomasz talk.

While surfels are persistent once they spawn, they do update their positions every frame in order to follow the surface they spawned on.
To accomplish this, surfels track a unique transform identifier of whatever geometry they are attached to.

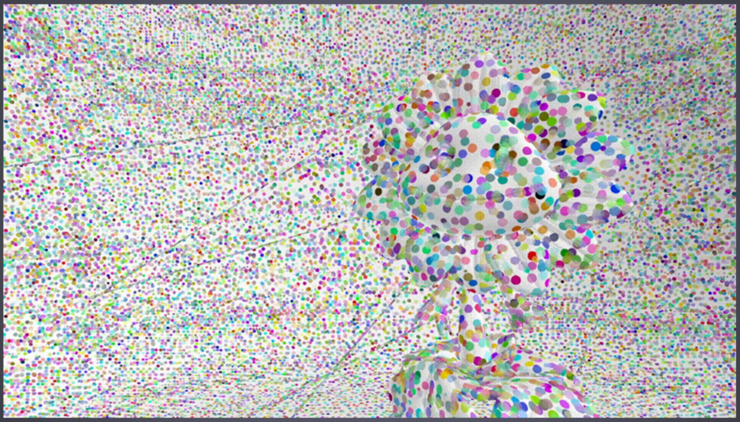


Transform IDs

This identifier is written to the gbuffer and picked up by the surfels during spawn. This image is a visualization of these IDs. In Frostbite, a global transform buffer is maintained, for surfels and other purposes, that contains the transforms of any geometry in the scene, including skinned bones.
The surfels store their local position in the transform we've identified for them.
Each frame we use the surfel's relative position together with the transform ID and global transform buffer to compute a new world space position.

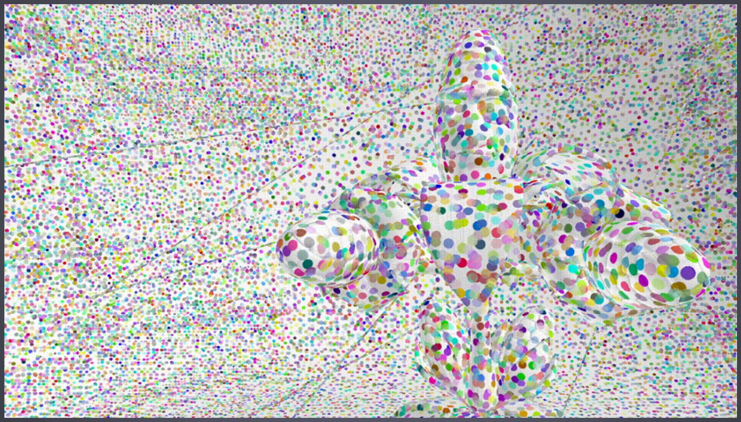


Skinned Meshes



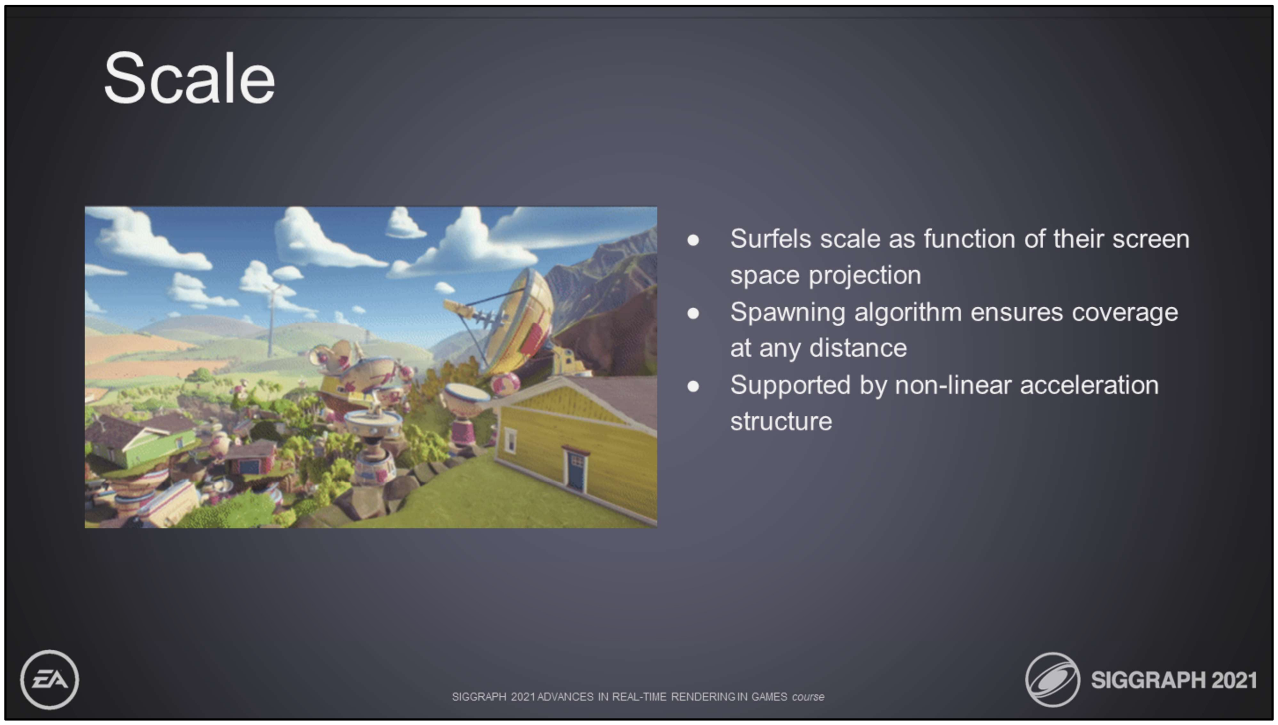
The obvious use case for this is for rigid geometry, but it also works for skinned meshes.
When skinned geometry writes their transform IDs to the GBuffer, they write the ID of the bone with the highest skinning weight.
This allows for essentially one bone skinning of surfels to geometry skinned with any number of weights. Because we only use one bone there are situations where the surfels don't follow exactly, but the surfel lighting algorithm is fairly forgiving in those circumstances.

Skinned Meshes



SIGGRAPH 2021

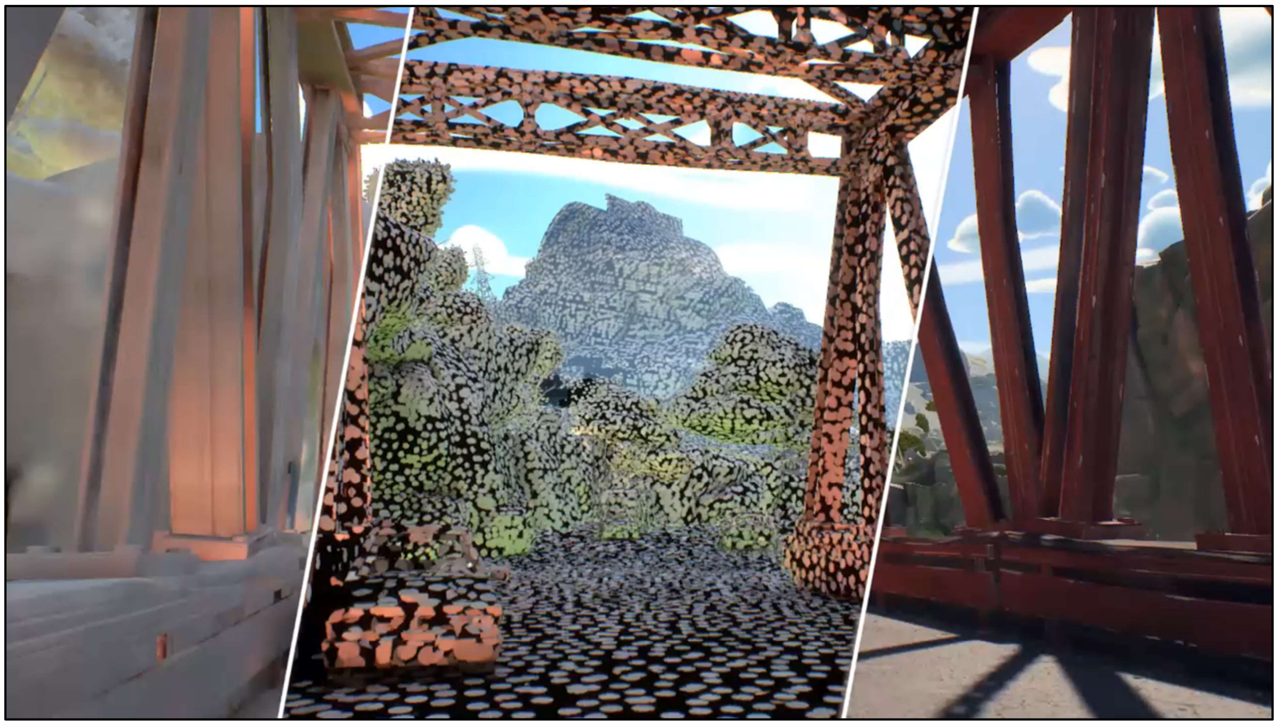
Since everything is assumed to be dynamic with GIBS, skinned and moving geometry both interact with the rest of the solution just like static geometry does.



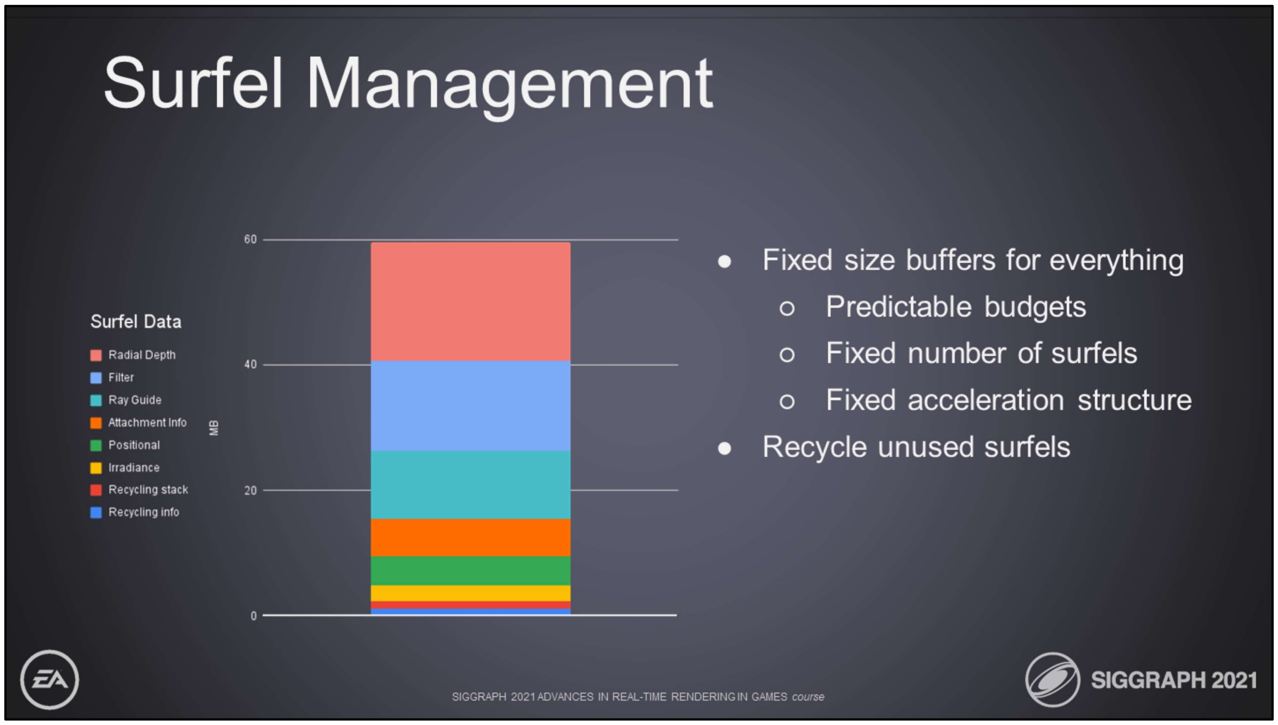
Scale

- Surfels scale as function of their screen space projection
- Spawning algorithm ensures coverage at any distance
- Supported by non-linear acceleration structure

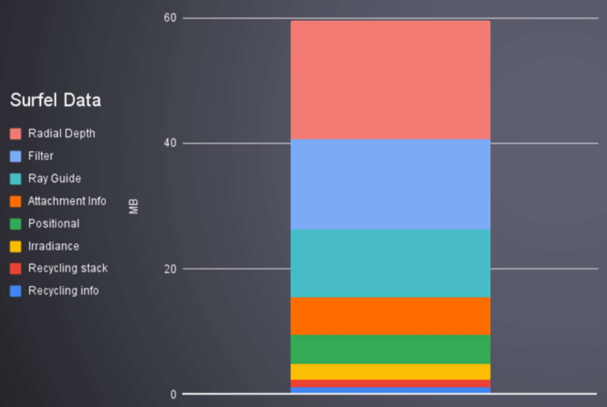
The surfelization of the scene works at any distance. Surfels are scaled so that their projection in screen space is roughly constant. This is true both when they spawn as well as when we move through the environment.



In this view, as we approach the top of the hill, you can see that the surfels shrink, and as they do we spawn more surfels to maintain the same ratio of surfels per screen space area. The reverse is true when we back away, surfels grow, and we can remove surfels where coverage is too high. The obvious impact of this is a constant level of quality at all distances, but the implication is the same for performance as well.



Surfel Management

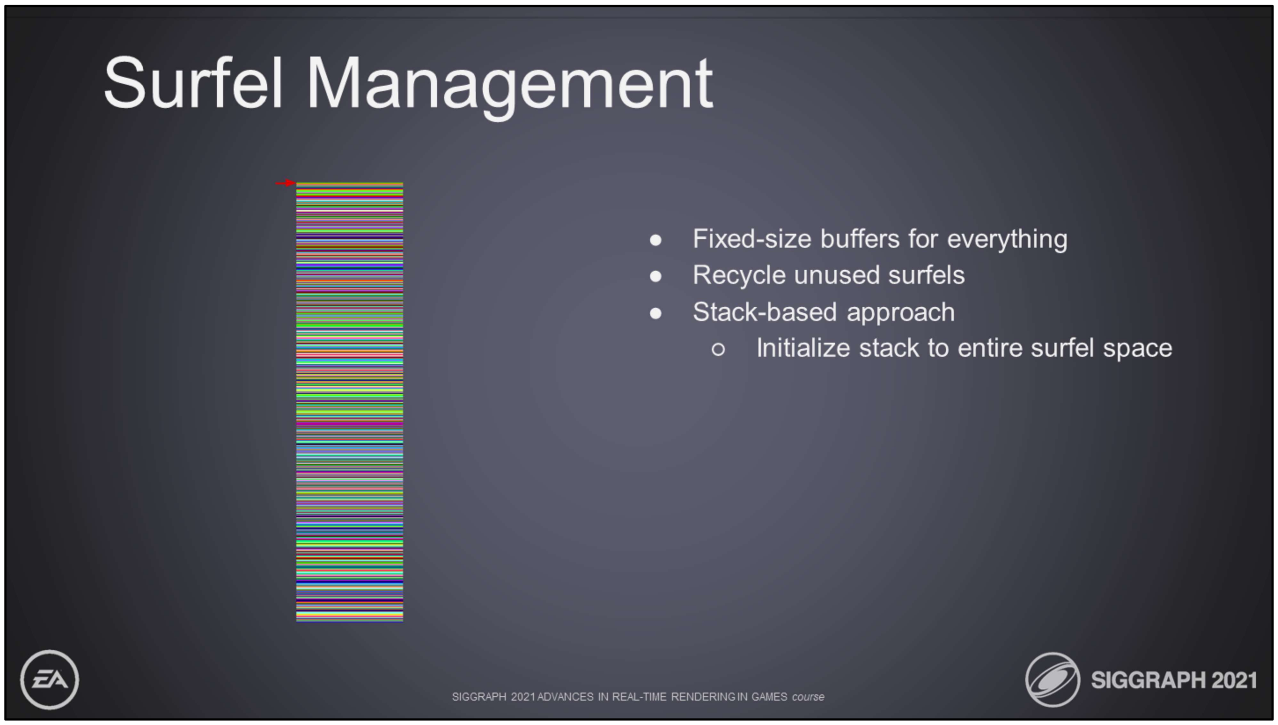


- Fixed size buffers for everything
  - Predictable budgets
  - Fixed number of surfels
  - Fixed acceleration structure
- Recycle unused surfels

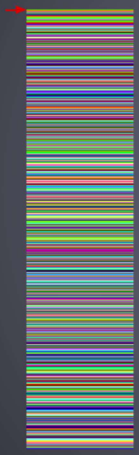
SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

Since we are dynamically spawning surfels as we move through the level, the question may be asked how we keep memory, and to some degree performance, consistent. The surfel algorithm allocates all the resources required for surfel management up front. This means that there are limits to the number of surfels that are available.

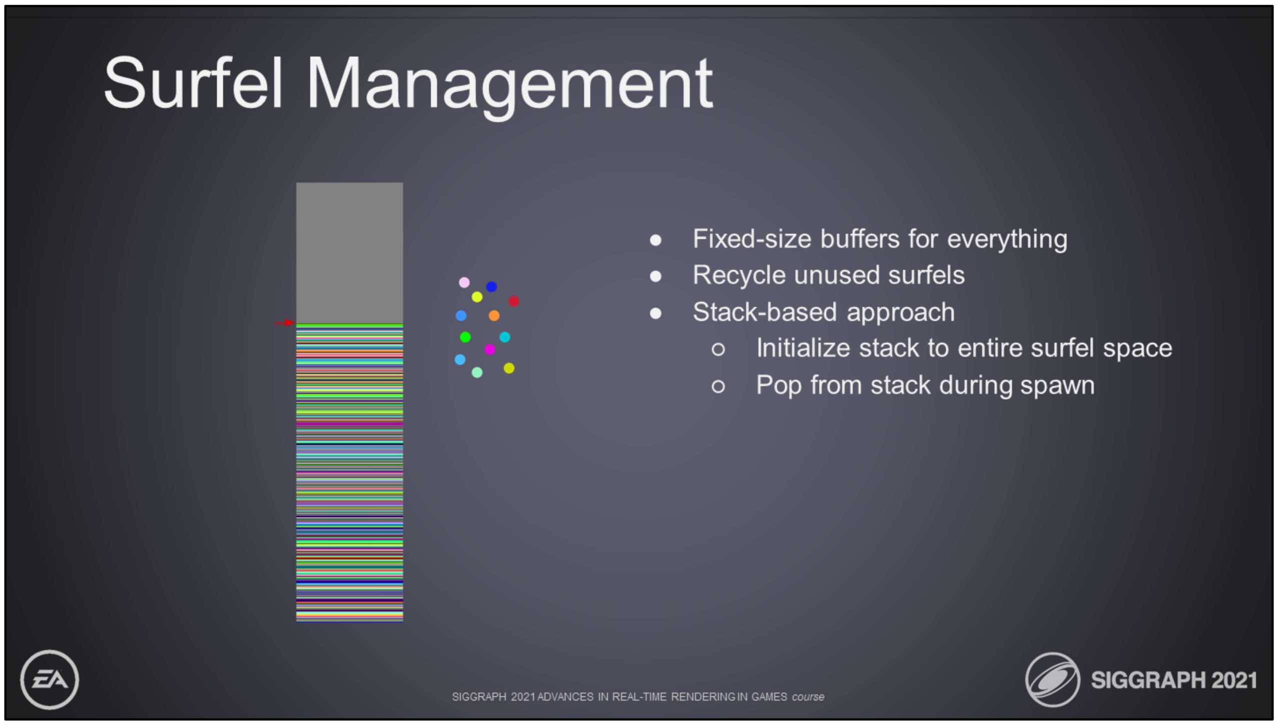


Surfel Management

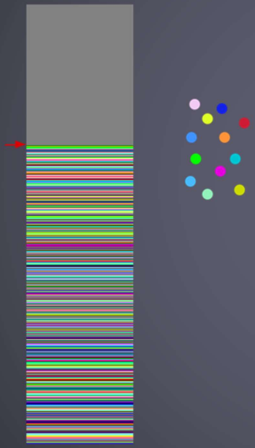


- Fixed-size buffers for everything
- Recycle unused surfels
- Stack-based approach
  - Initialize stack to entire surfel space

To effectively make use of the available resources, we employ a surfel recycling algorithm that removes surfels that are deemed to no longer be relevant. The way this works is that a stack of available surfels is maintained at all points. The stack is initialized to contain indirections to the entire surfel space.



Surfel Management

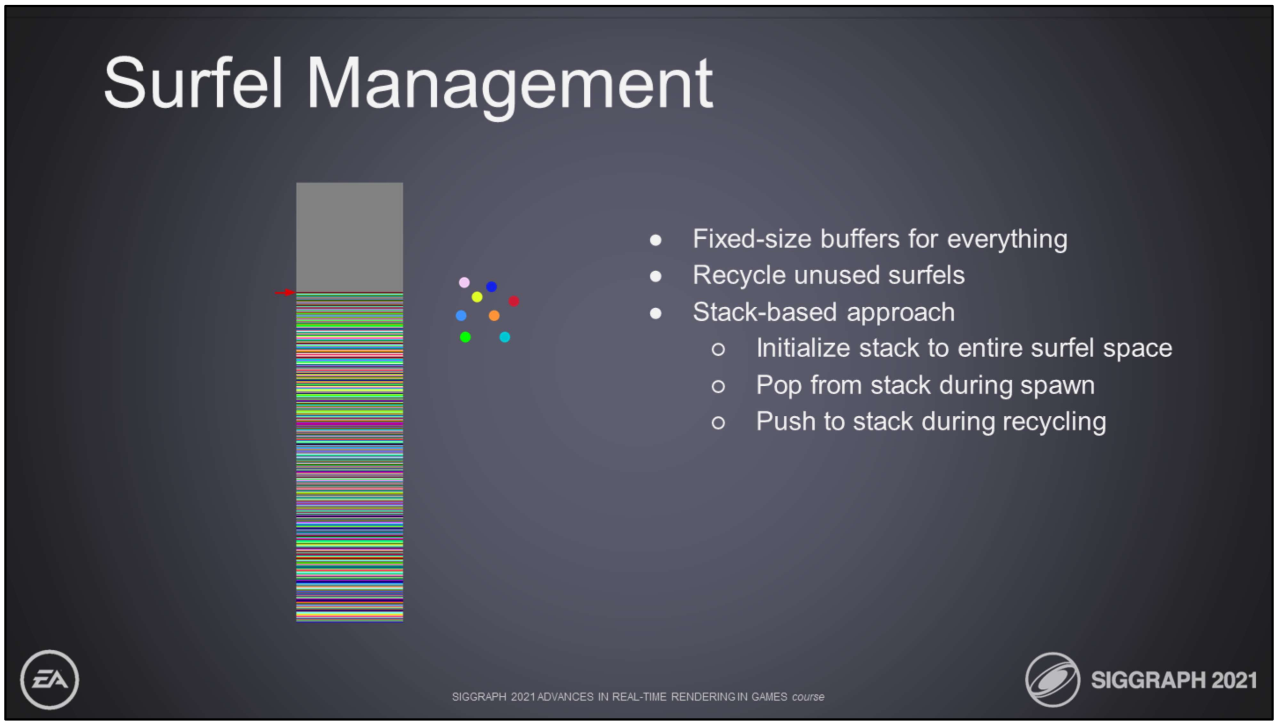


- Fixed-size buffers for everything
- Recycle unused surfels
- Stack-based approach
  - Initialize stack to entire surfel space
  - Pop from stack during spawn

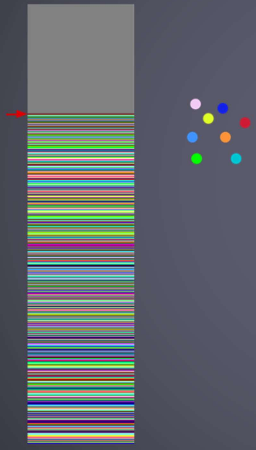
SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

When a surfels spawn, the stack is decremented, on the GPU, by an atomic operation on the stack counter, and the ID of that surfel is retrieved from the stack.



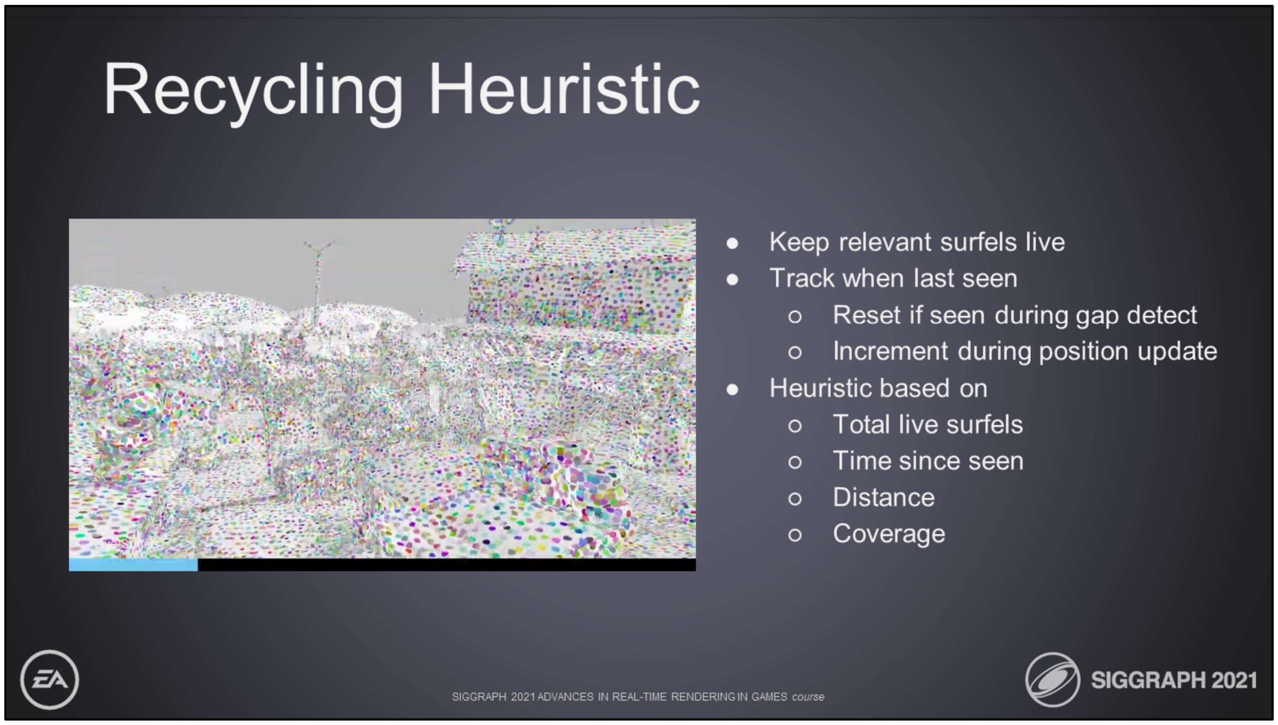
Surfel Management



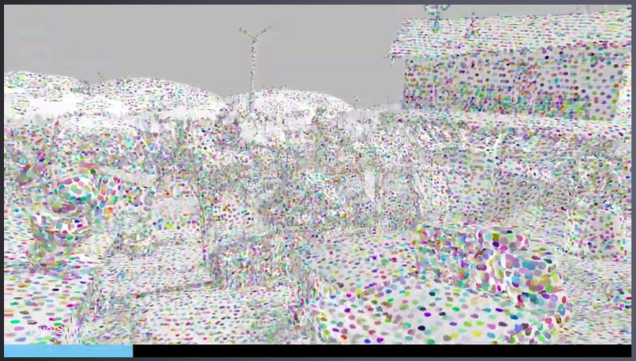
- Fixed-size buffers for everything
- Recycle unused surfels
- Stack-based approach
    - Initialize stack to entire surfel space
    - Pop from stack during spawn
    - Push to stack during recycling

SIGGRAPH 2021

When a surfel is recycled, the stack counter is incremented, and the surfel writes its ID to the pre-allocated stack buffer. As with everything else in this algorithm, this is all done on the GPU, in this case with atomics and indirections



Recycling Heuristic



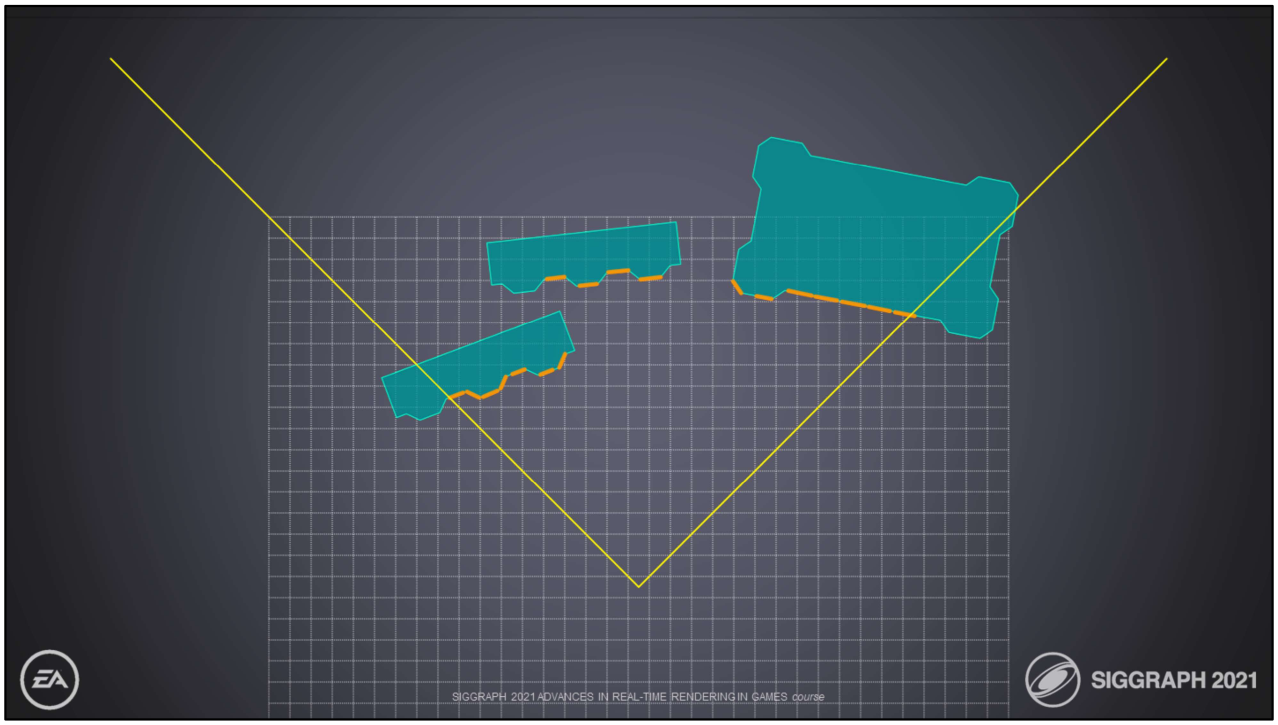
- Keep relevant surfels live
- Track when last seen
  - Reset if seen during gap detect
  - Increment during position update
- Heuristic based on
  - Total live surfels
  - Time since seen
  - Distance
  - Coverage

While the recycling algorithm keeps the number of live surfels in check, we don't want to recycle surfels unnecessarily. That would throw away the all the hard work done to calculate lighting for that surfel. For this reason, we employ a recycling heuristic based on some factors. These factors include: 1. How many surfels are currently live. 2. When the surfel last contributed to the lighting of the scene. 3. How far away the surfel is. These factors are combined and compared to a random number pulled from the uniform distribution, giving surfels that are less relevant a higher probability to be recycled.

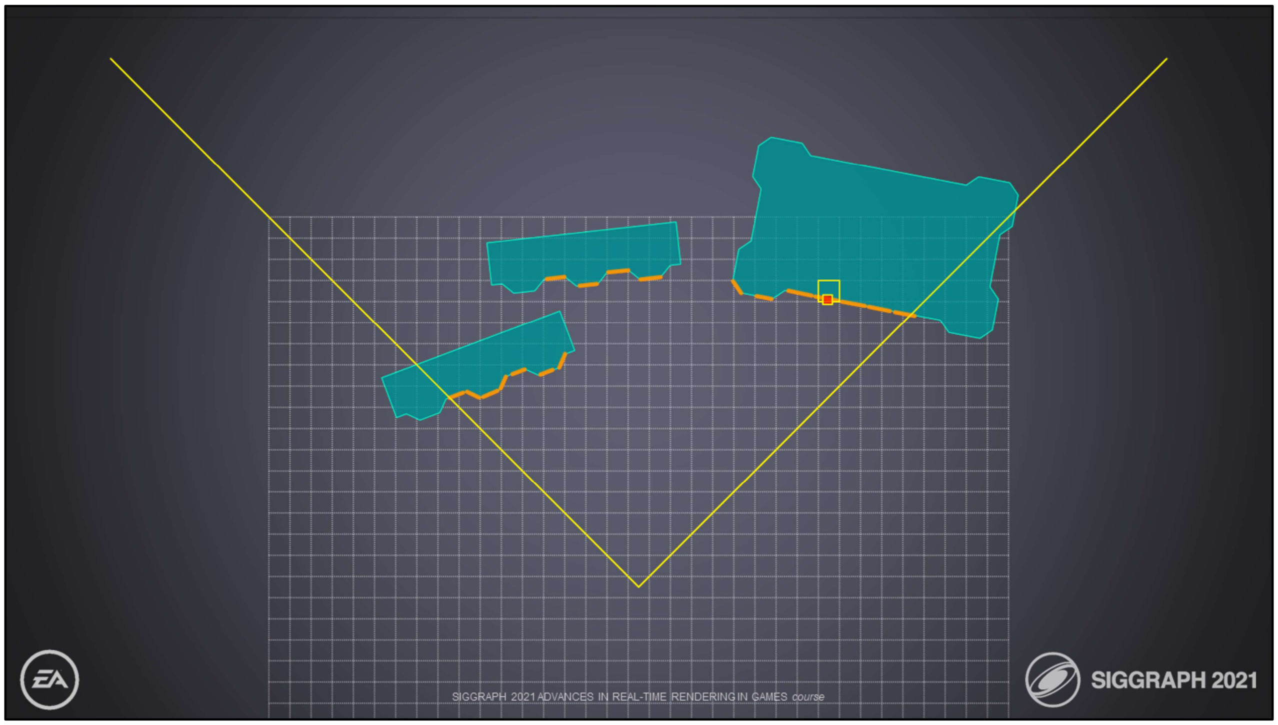
Acceleration Structure

Many parts of the algorithm require quickly finding which surfels are in the vicinity of a given position.

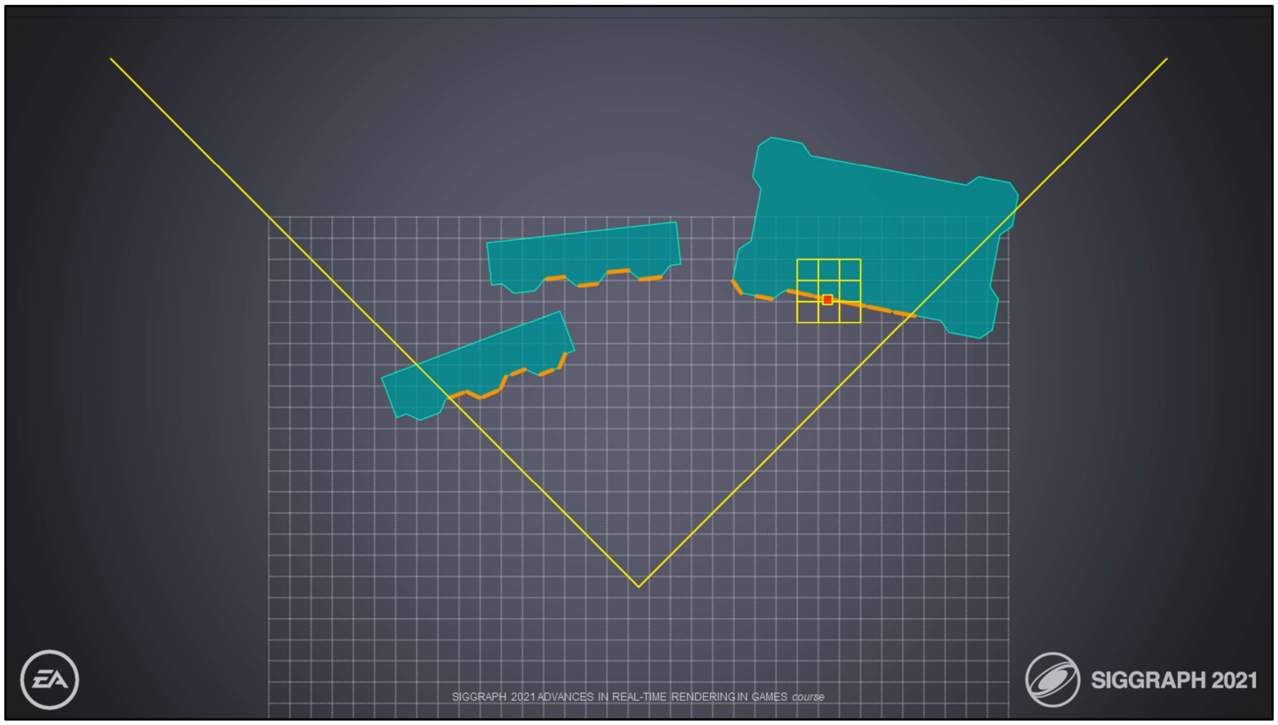
In the original Pica Pica version, a uniform grid was used to accelerate these queries.



Here is a visualization of of that. We have a view frustum in yellow, and some geometry in the scene with surfels attached to it.

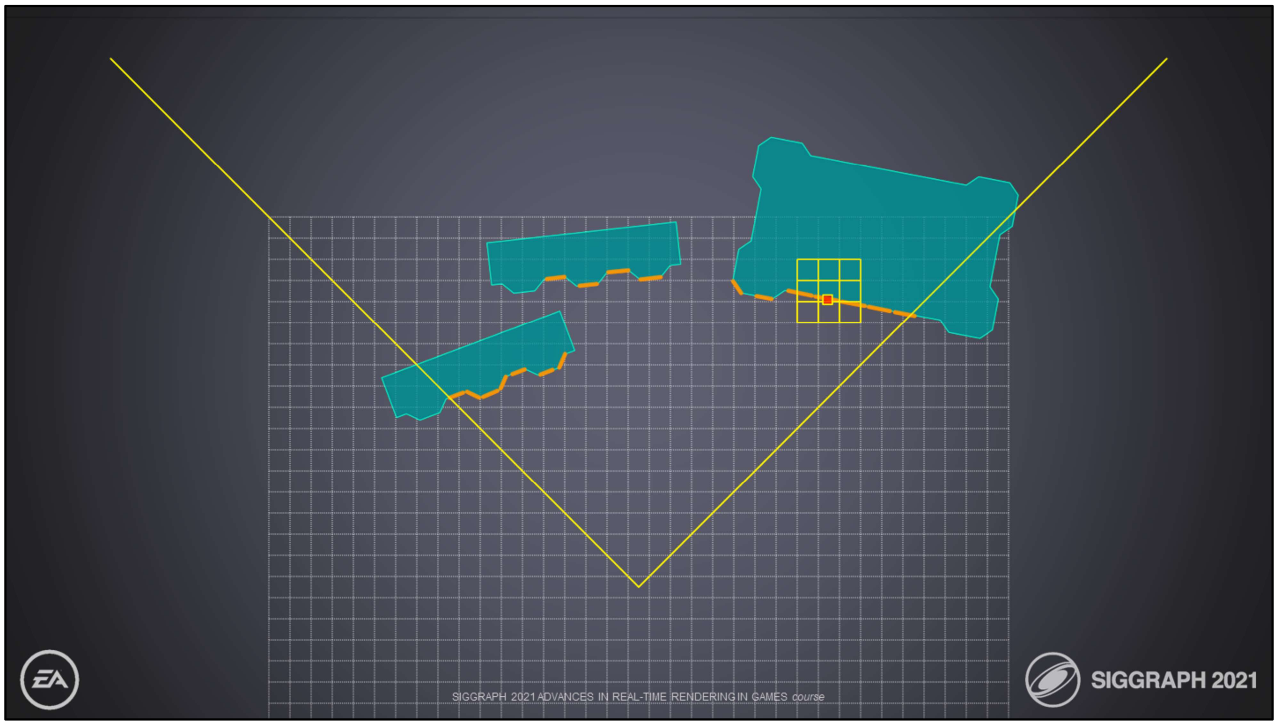

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*
SIGGRAPH 2021

We insert all surfels into this structure every frame. First a surfel finds its position in the grid, which is a simple operation, and inserts itself there.

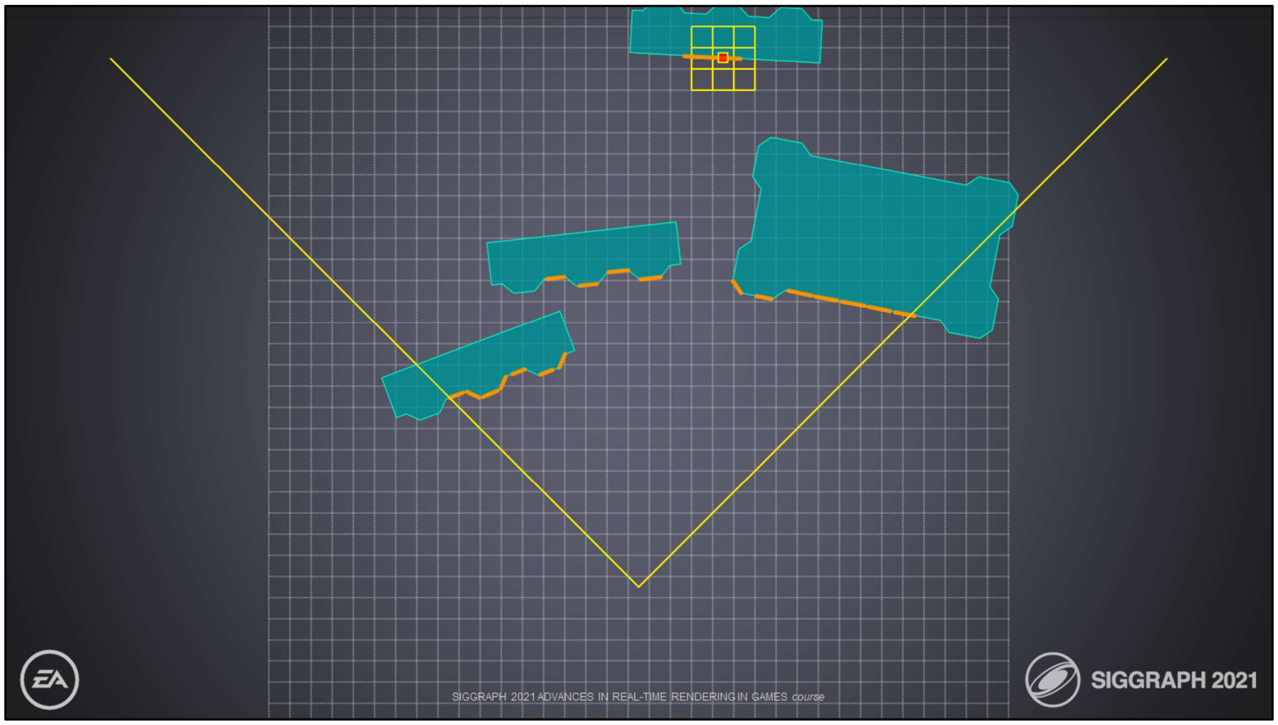


SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*
SIGGRAPH 2021

But surfels have a radius, so we check to see if the surfel overlaps any of the neighboring cells as well, and insert it there as well.
We guarantee that the radius of a surfel is never larger than the side of a grid cell.


SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*
SIGGRAPH 2021

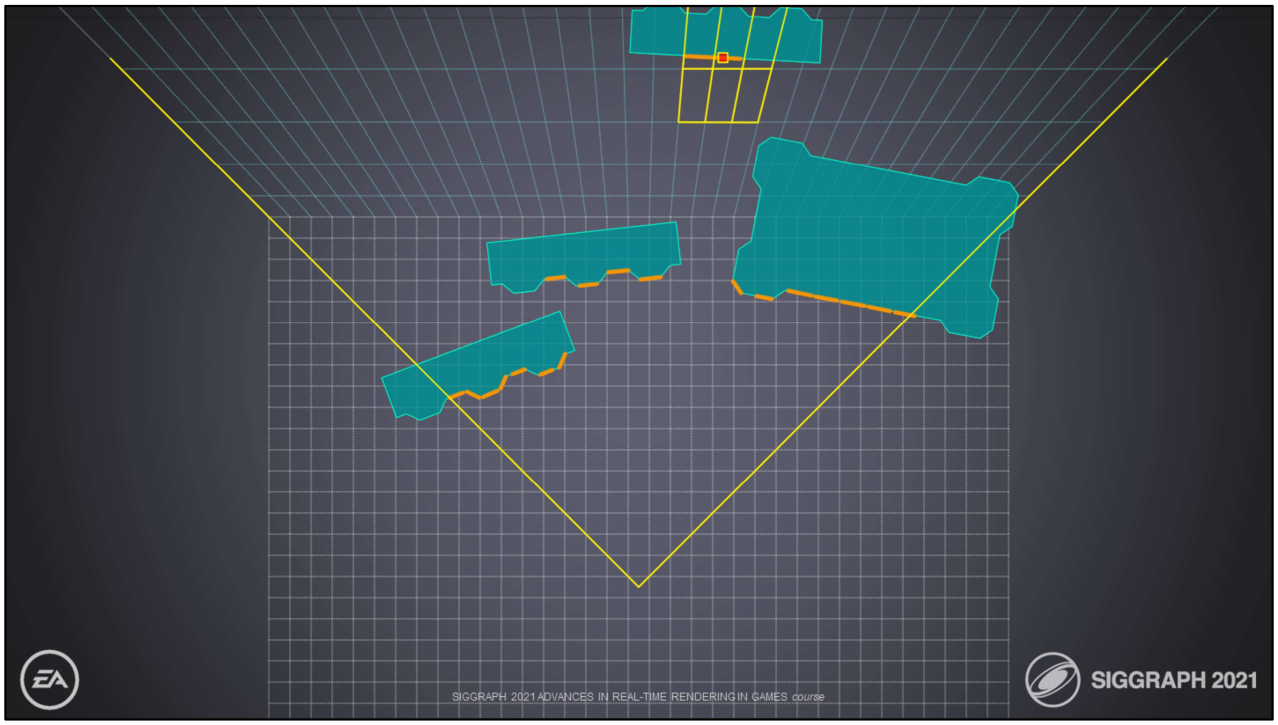
But we also insert it into its immediate neighbors, if the surfel covers those at all. This to allow ease of access later on, and to avoid discontinuities in the lighting.


SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES course
SIGGRAPH 2021

The uniform grid worked well for the small level of Pica Pica, but presented a problem when we tried to move to levels of a size used in modern games.

As we mentioned earlier, the radius of surfels are adjusted to be roughly constant in screen space, which means distant surfels will have a large radius.

To avoid the aforementioned discontinuities we would have had to use a large cell size as well, which would have negated a lot of the benefit of using a grid.

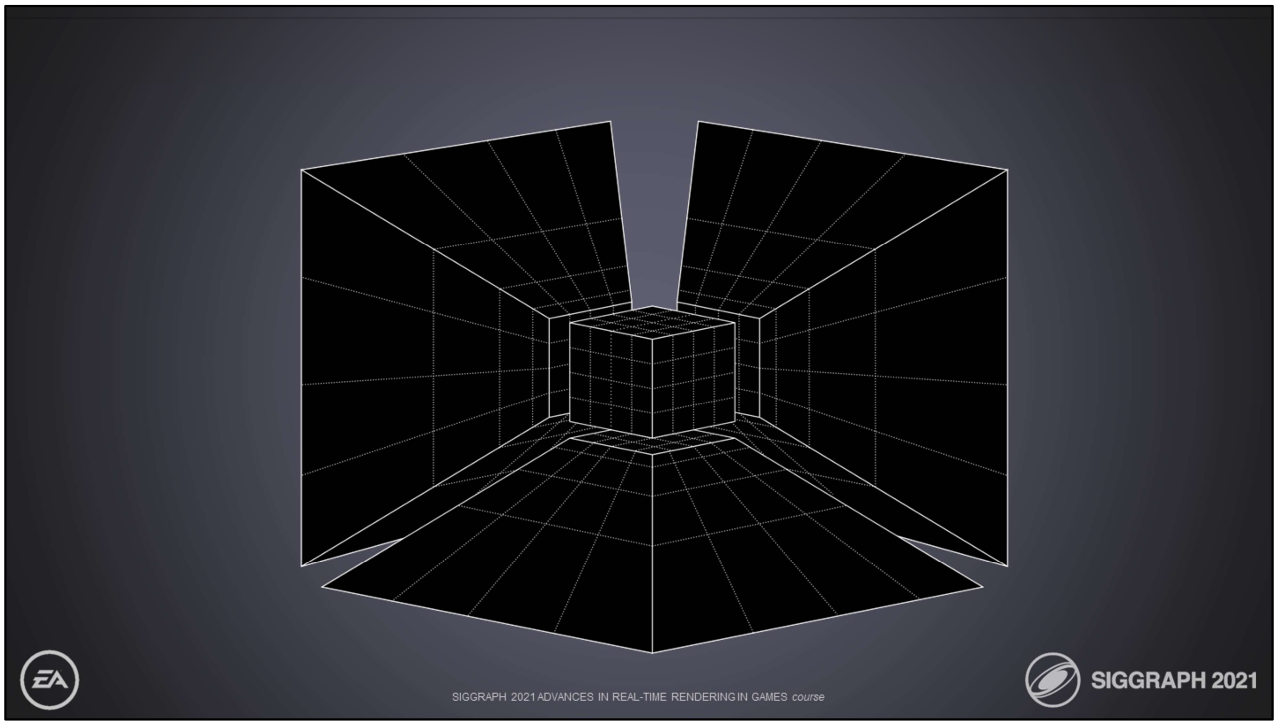

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*
SIGGRAPH 2021

We wanted an acceleration structure with properties similar to a projection transform, where we had more resolution close to the camera.

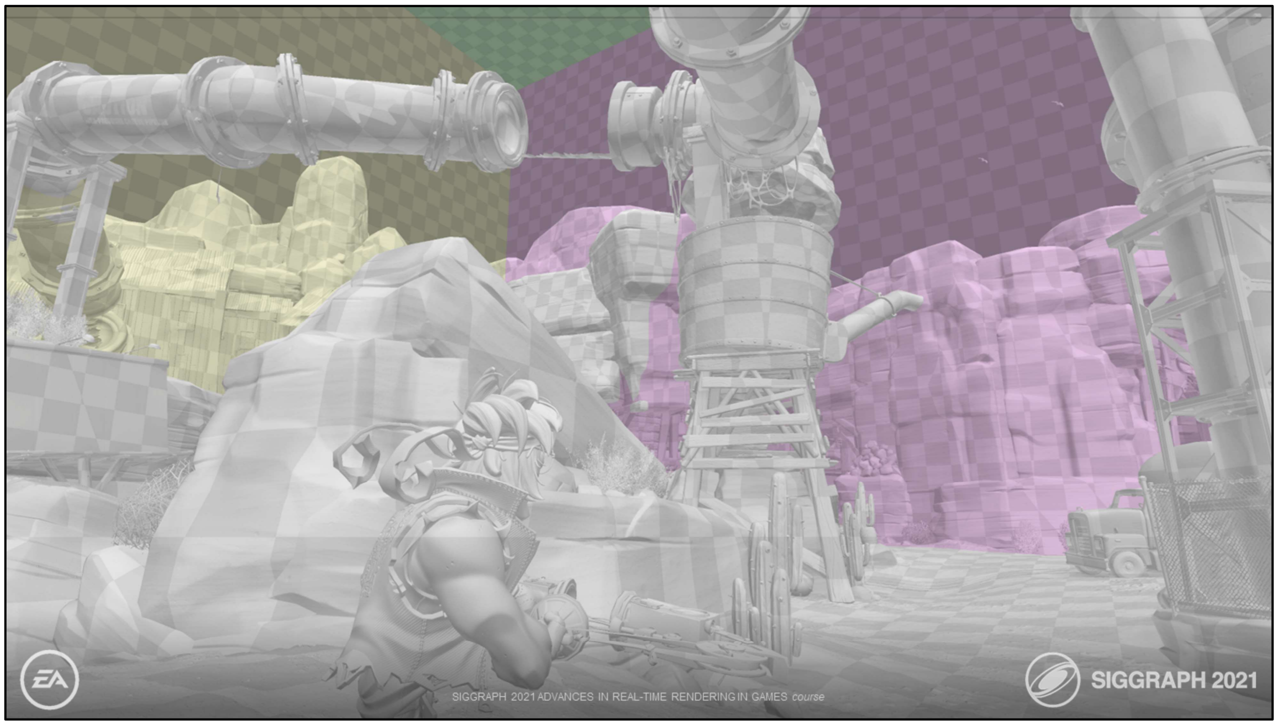
We also wanted something reasonably simple to make sure lookups remained fast.

After some experimentation, we settled on a structure where we keep a uniform grid for a small area close to the camera, combined with a trapezoidal grid along each principal axis.

The thicknesses of the slices of the trapezoidal grids increase with distance from the center of the structure.

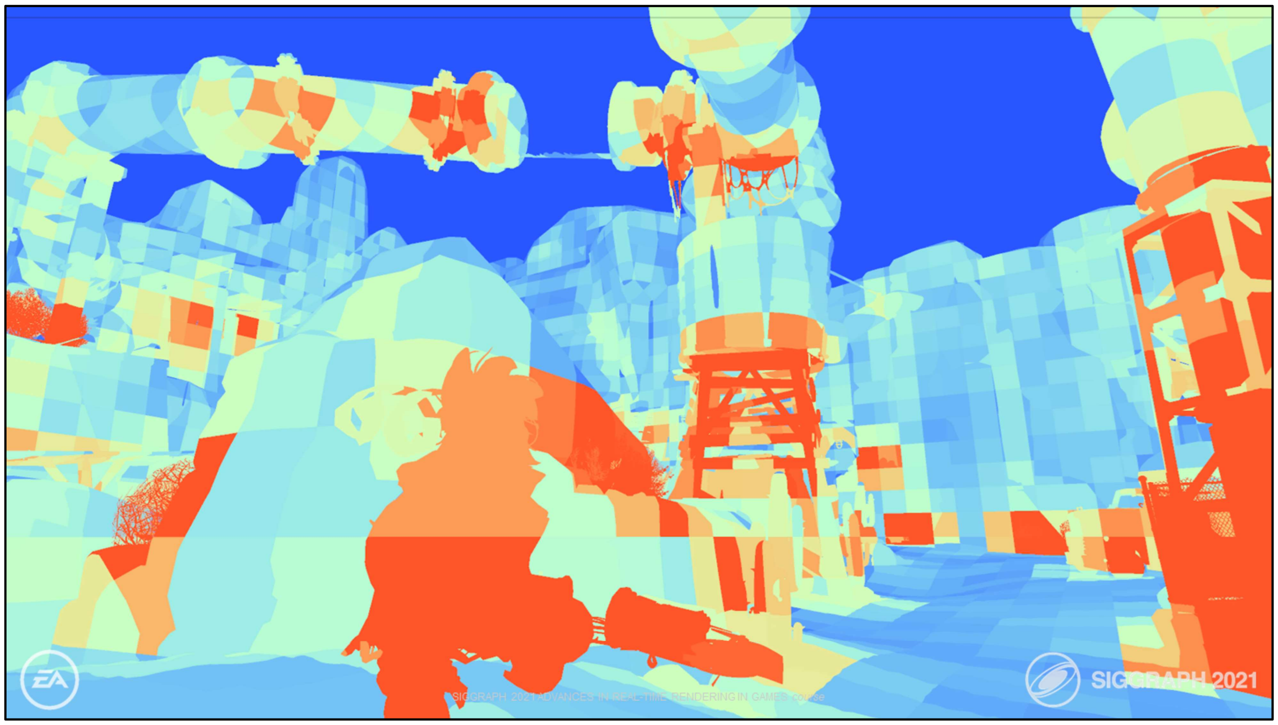


Here is an image of what the structure looks like in 3D. This gives us an acceleration structure that perfectly fits the way surfels grow with distance. And since the trapezoids are essentially regular grids, just with a non-linear transformation, lookups and insertions are still very fast.


SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES course
SIGGRAPH 2021

This is a debug view from showing the cells of the uniform center grid in grey, as well as the surrounding non uniform grids in other colors.

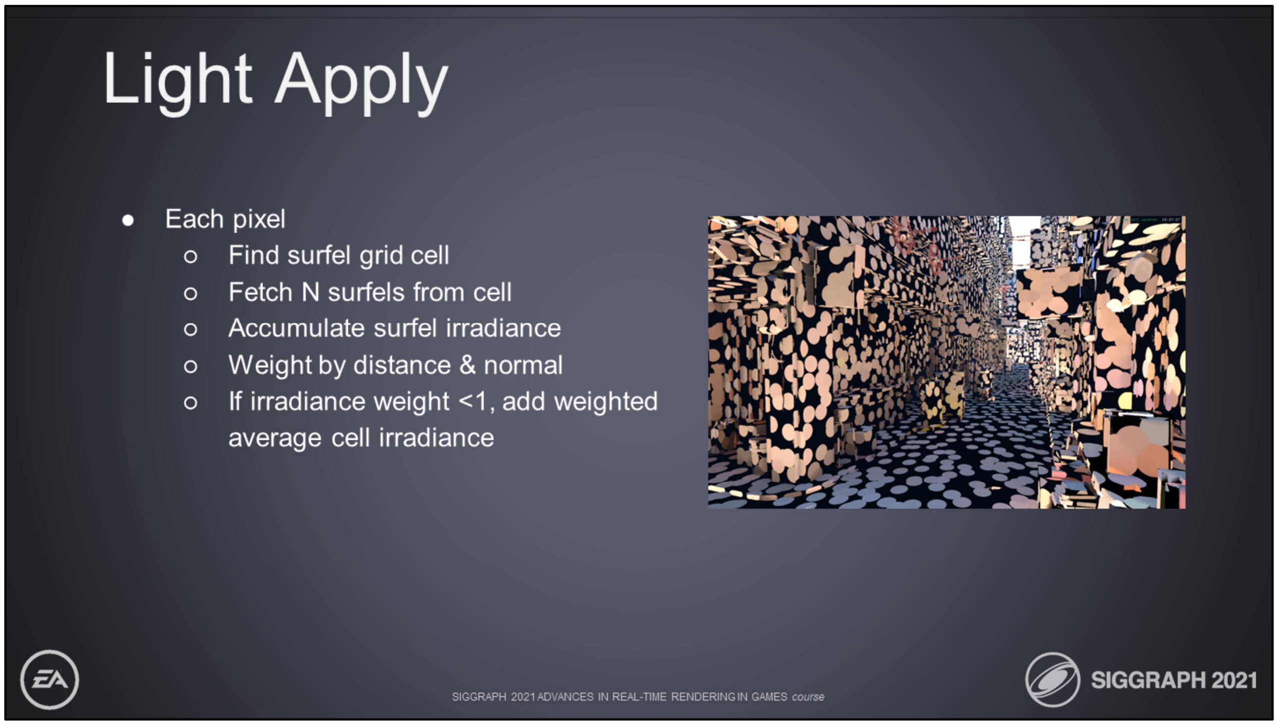
As you can see the trapezoidal grids maintain a constant grid cell size regardless of distance.


SIGGRAPH 2021
© SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES course

This view shows a heat map of the number of surfels in each cell.

For areas with comparable geometric complexity, the number of surfels in each cell stays constant in screen space.

Cells that contain a lot of geometric complexity naturally contain more surfels, in order to approximate the underlying geometry. You can see this on the character at the center of the screen, as well as the structures underneath the cheese tanks.



Light Apply

- Each pixel
  - Find surfel grid cell
  - Fetch N surfels from cell
  - Accumulate surfel irradiance
  - Weight by distance & normal
  - If irradiance weight <1, add weighted average cell irradiance

In the final pass of our surfel frame, we reconstruct the world space position of each pixel, find which cell the position is inside, and then fetch the first N surfels in the cell.

We loop over these surfels and accumulate their irradiance, weighted by their orientation and distance to the source pixel's position and orientation.

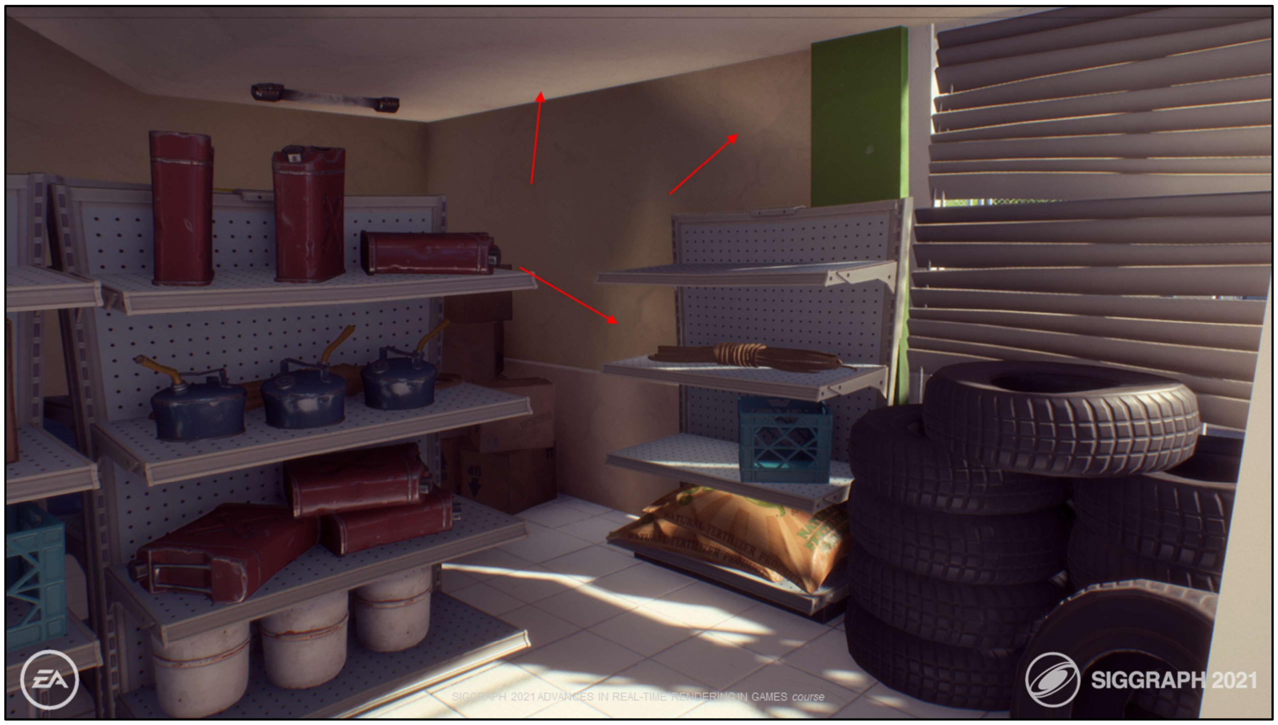
Once we've looped over the surfels, if we find that the contribution of the surfels is less than one, we then add in the weighted average irradiance of the grid cell.

Light Bleeding

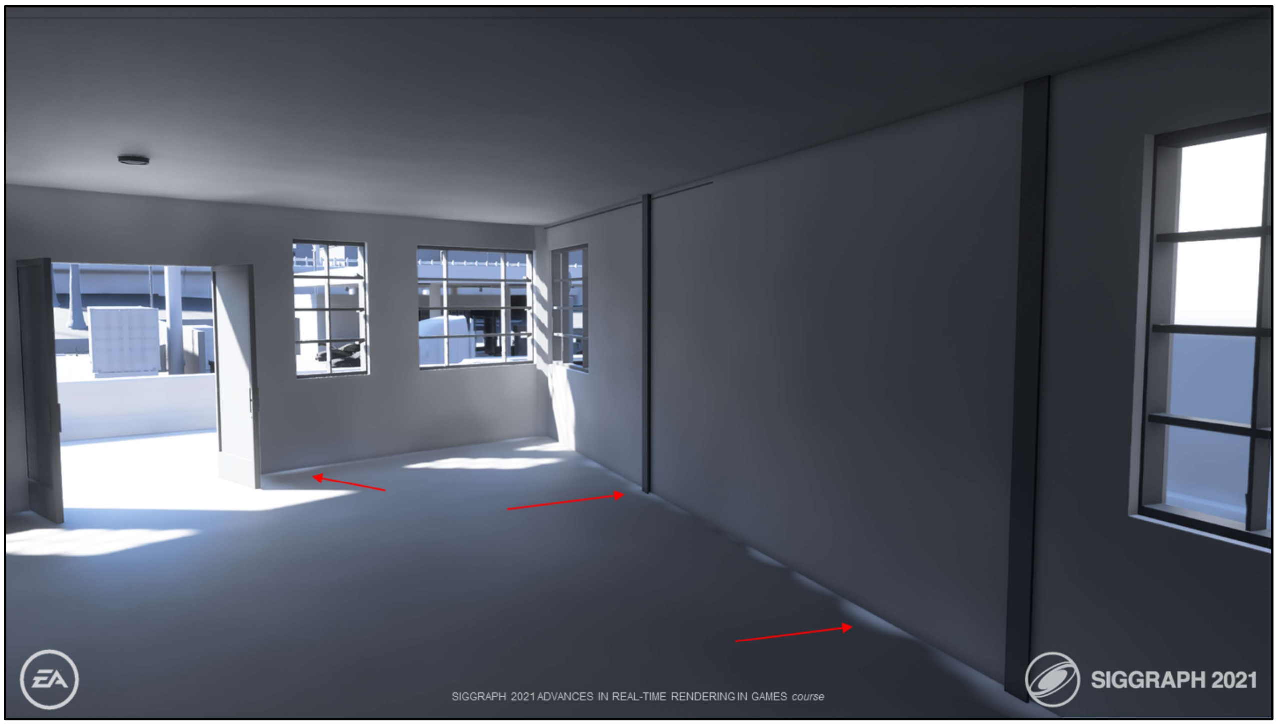
SIGGRAPH 2021

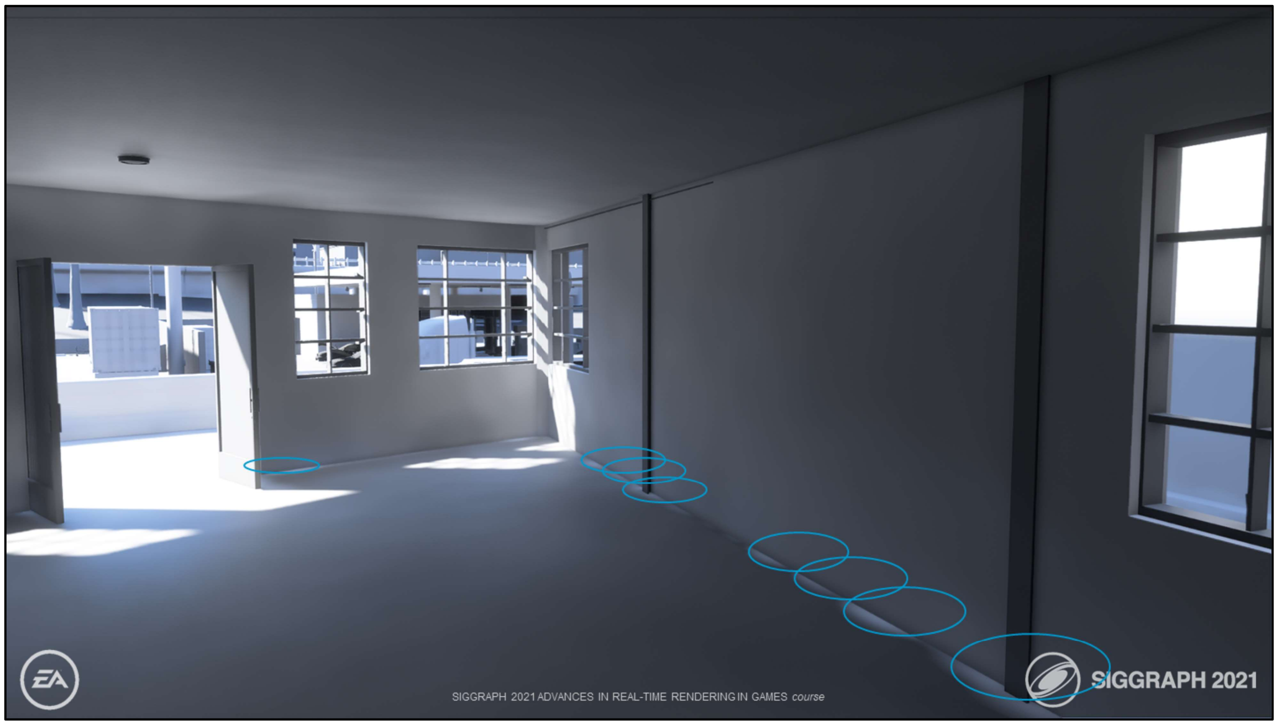
As we started to use surfels on game content, we noticed some some issues.

Even though surfels in a way represents the geometry in the scene, they do not inherently know what’s in their neighborhood. This in combination with the sometimes large size of surfels relative to the underlying scene can cause some unfortunate artifacts. One such artifact is bleeding of light through geometry. Here are a couple of examples of that.

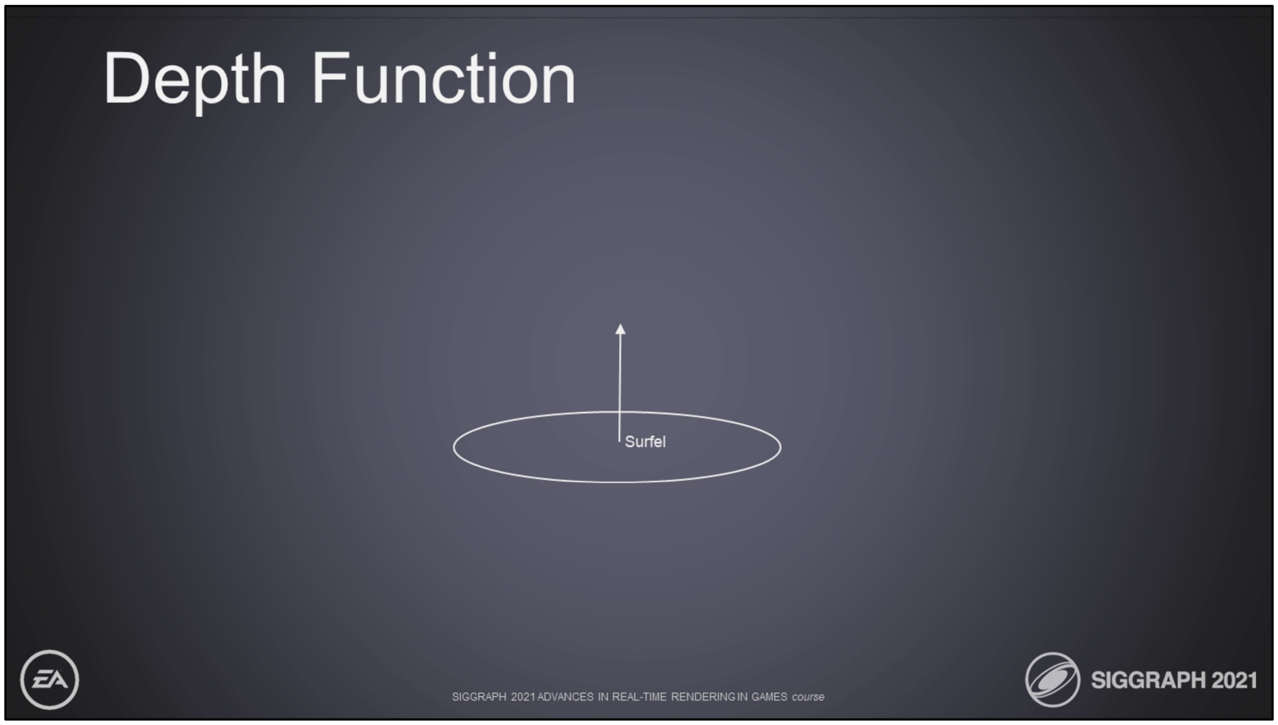

SIGGRAPH 2021
SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES course

Even though surfels in a way represents the geometry in the scene, they do not inherently know what's in their neighborhood. This in combination with the sometimes large size of surfels relative to the underlying scene can cause some unfortunate artifacts. One such artifact is bleeding of light through geometry. Here are a couple of examples of that.

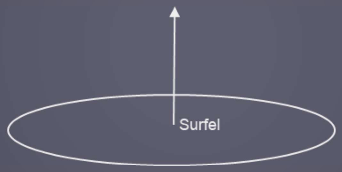

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*
SIGGRAPH 2021

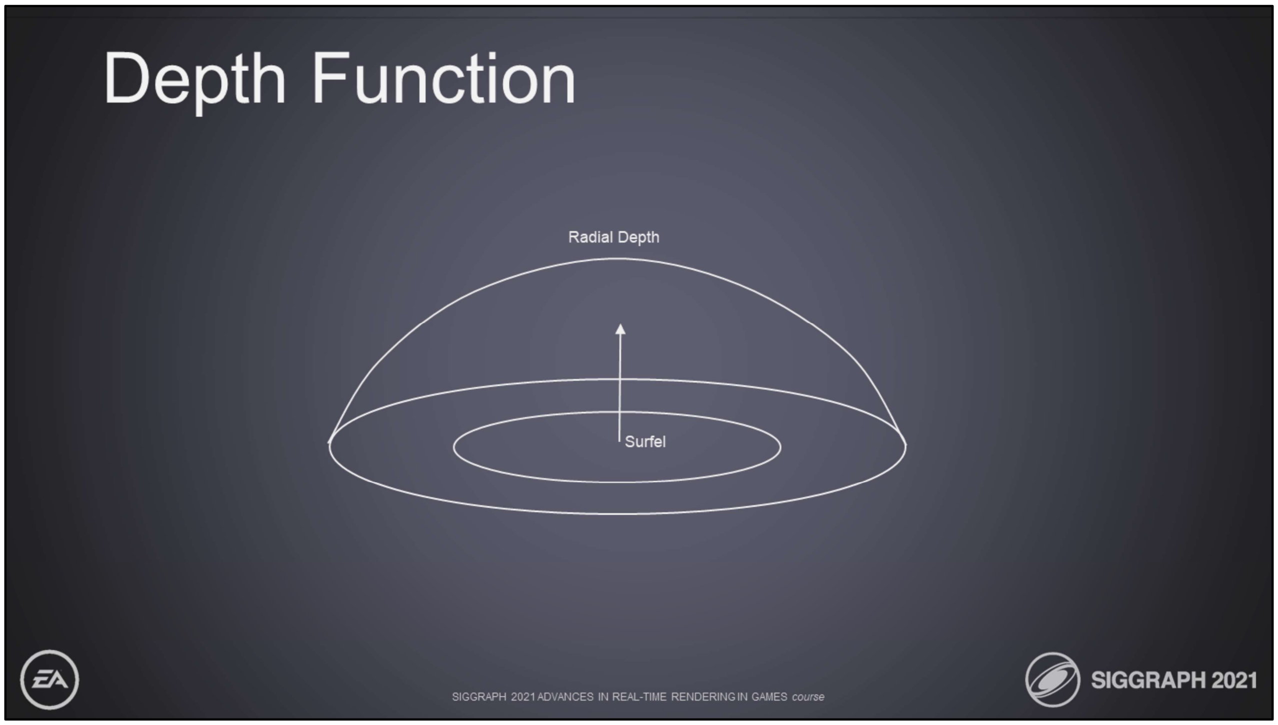

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*
SIGGRAPH 2021

What's happening here is that surfels outside of the wall, in a much brighter area, apply lighting through the geometry. The surfels simply don't know that the wall is there.

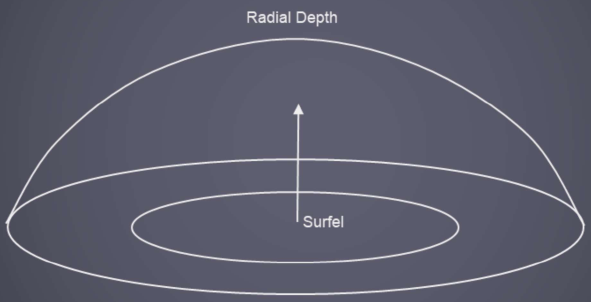


Depth Function



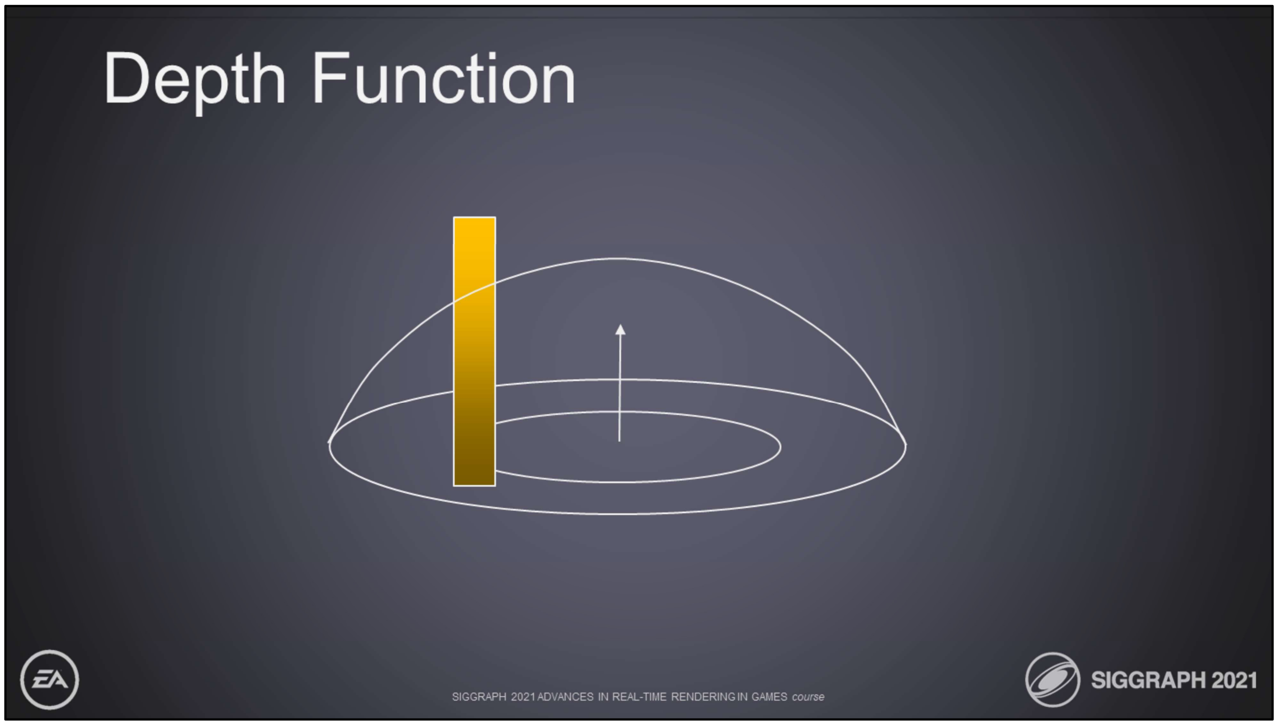


Depth Function

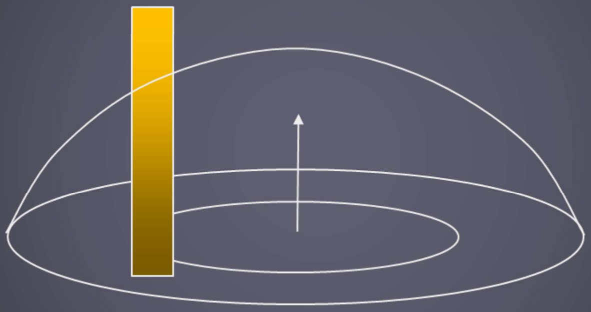


SIGGRAPH 2021

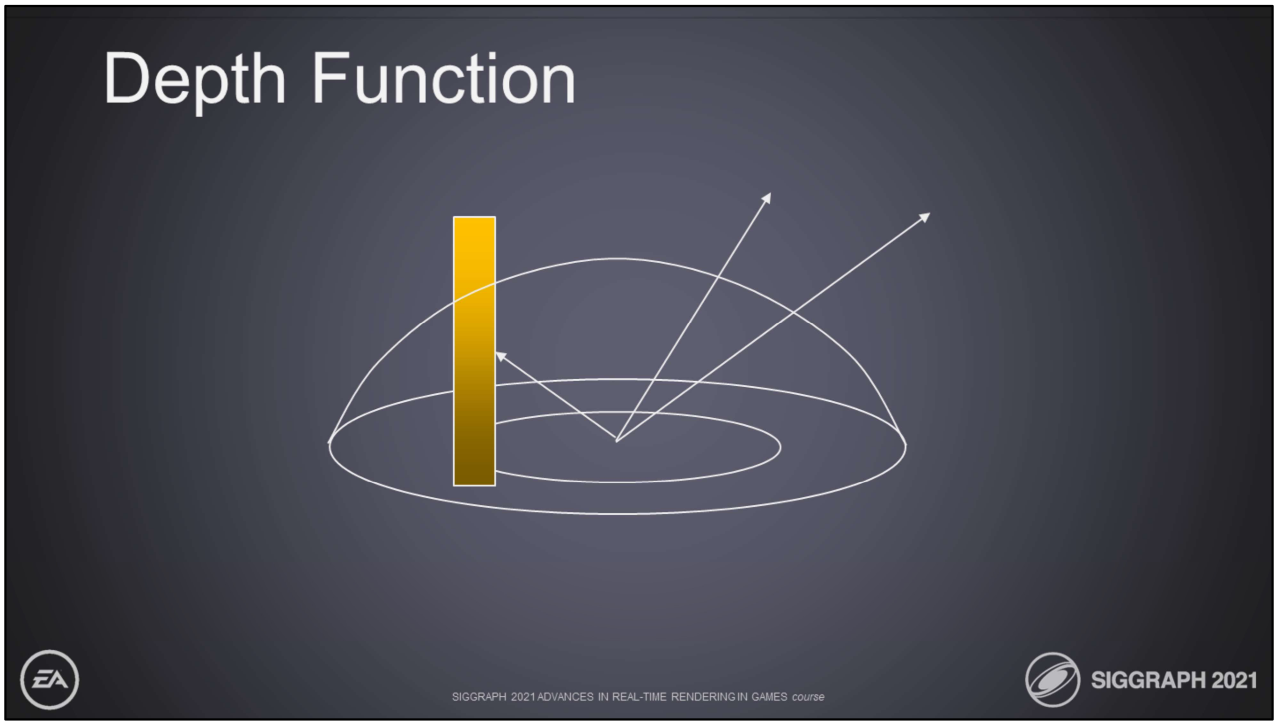
To solve this problem we use a radial depth function. We initialize this to the diameter of a surfel.



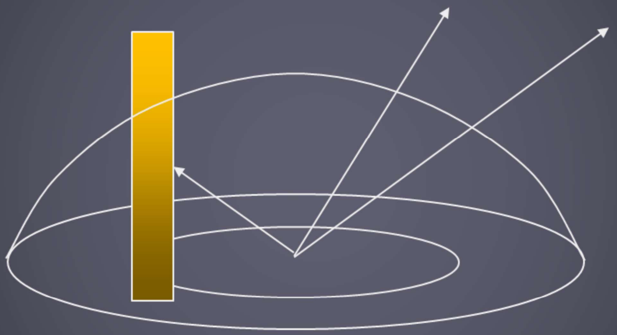
Depth Function



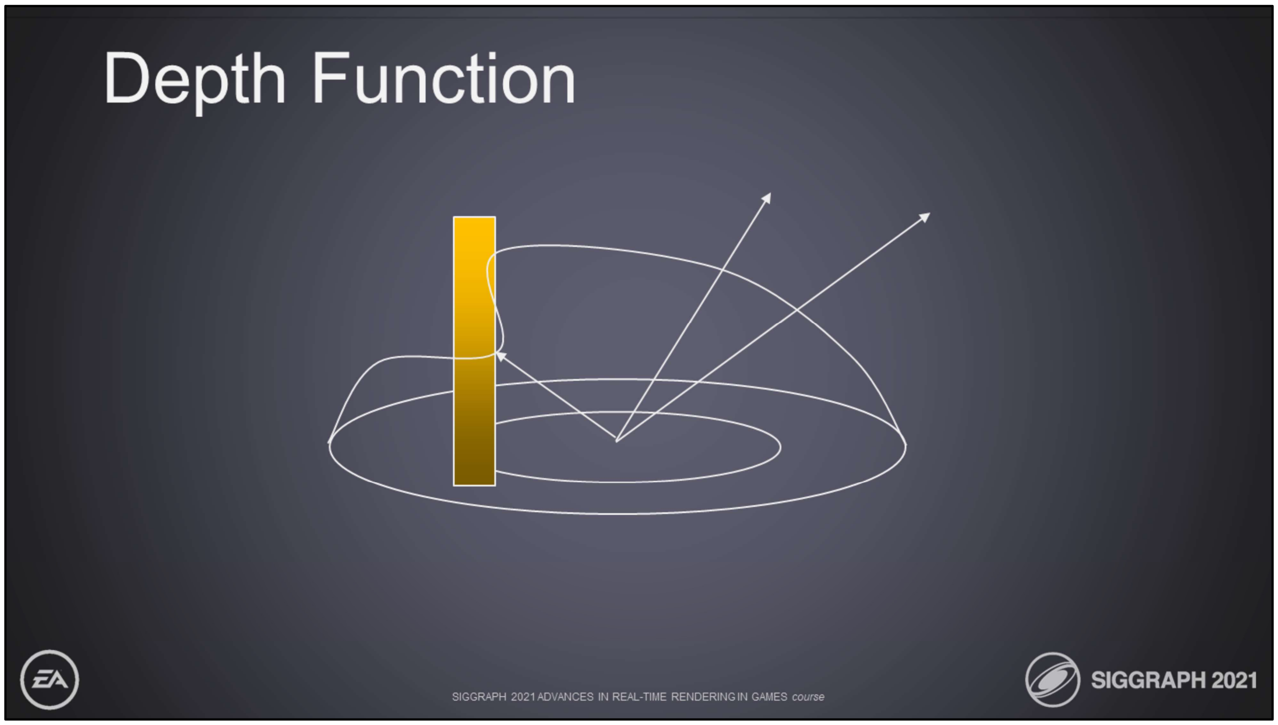
SIGGRAPH 2021



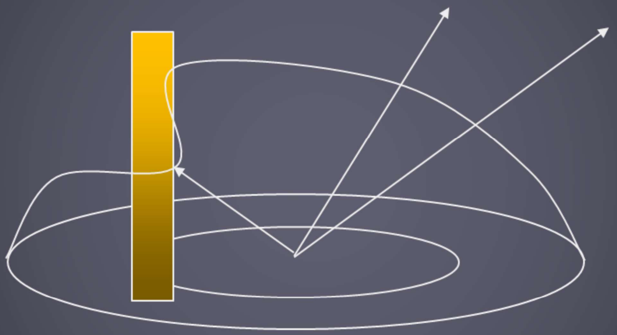
Depth Function

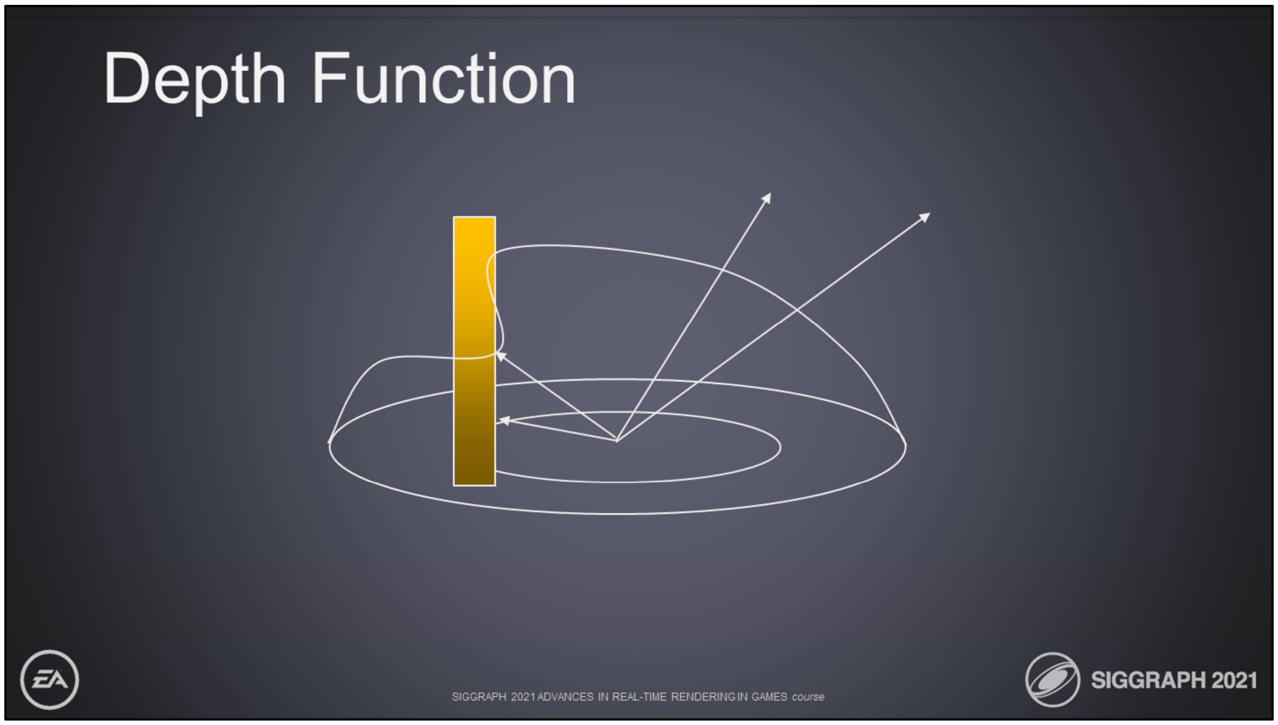


As we trace rays for accumulating irradiance, and we hit geometry within the surfel diameter, we update the depth function.

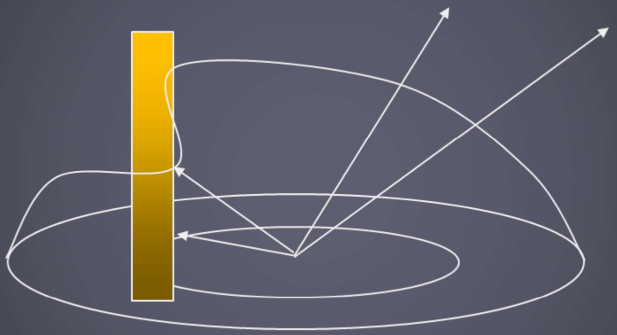


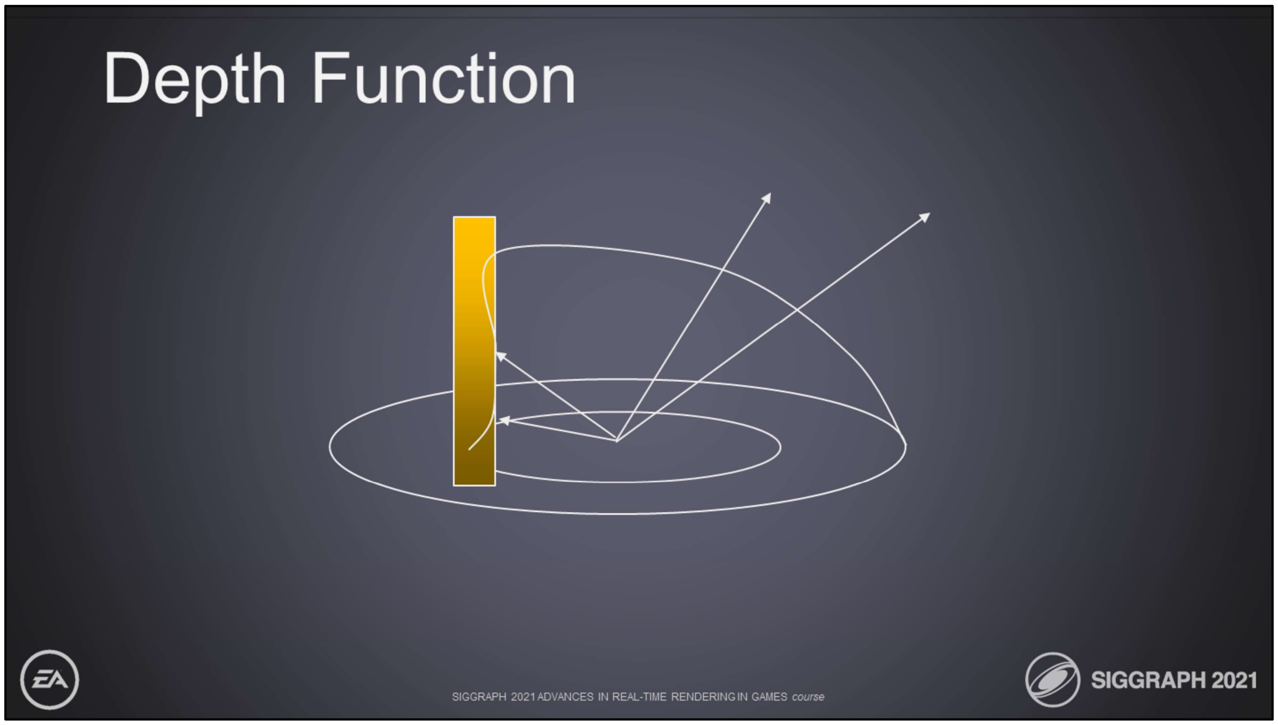
Depth Function





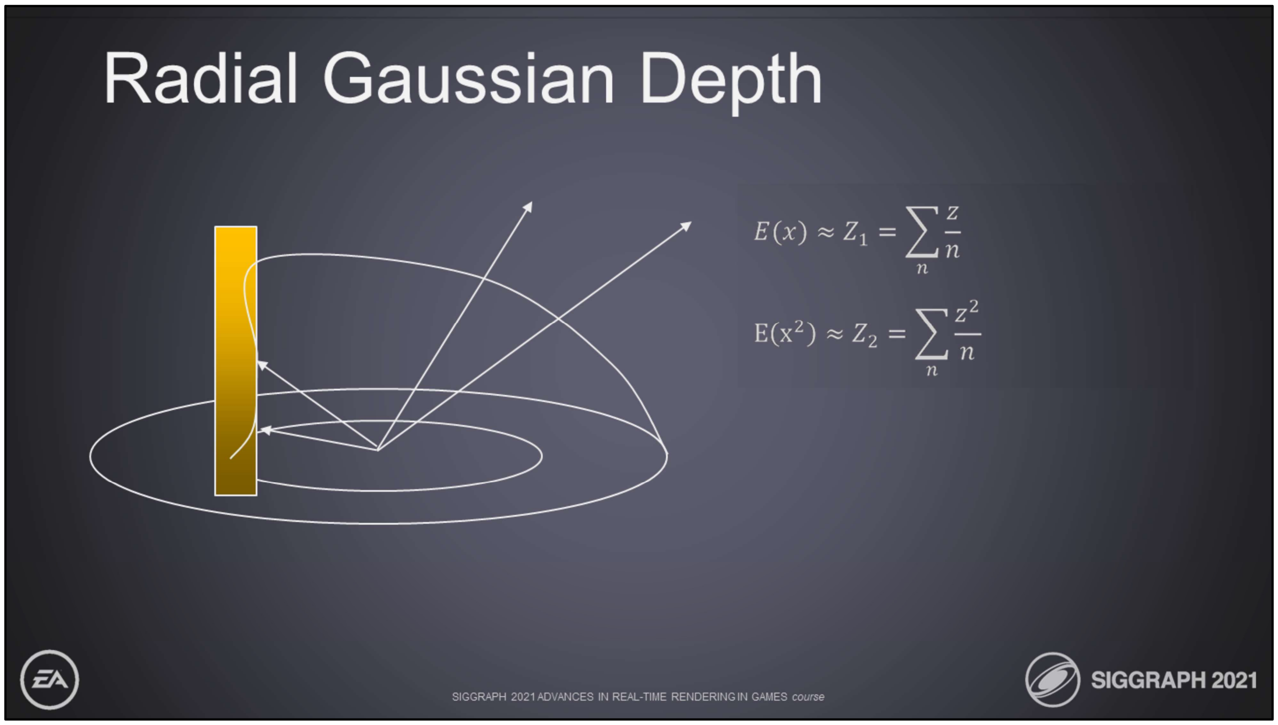
Depth Function



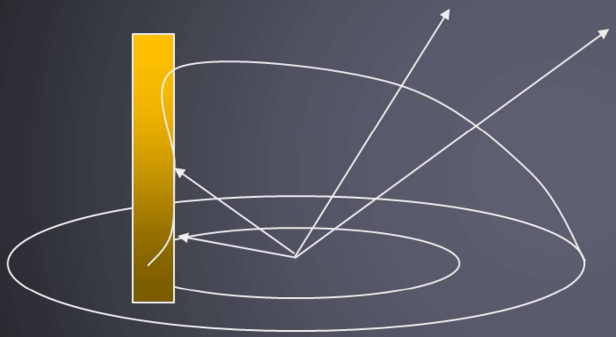


Depth Function

As we send more rays the depth function estimate gets better.



Radial Gaussian Depth



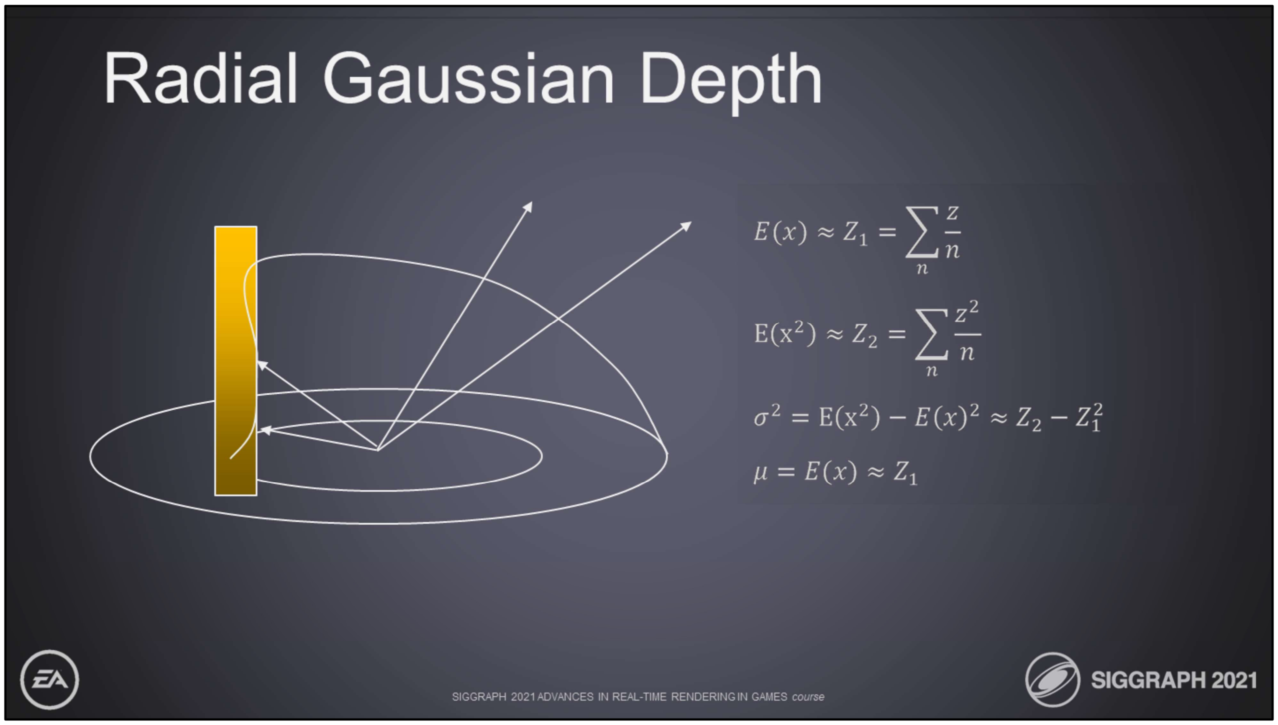
$$E(x) \approx Z_1 = \sum_n \frac{z}{n}$$

$$\mathrm{E}(\mathrm{x}^2) \approx Z_2 = \sum_n \frac{z^2}{n}$$

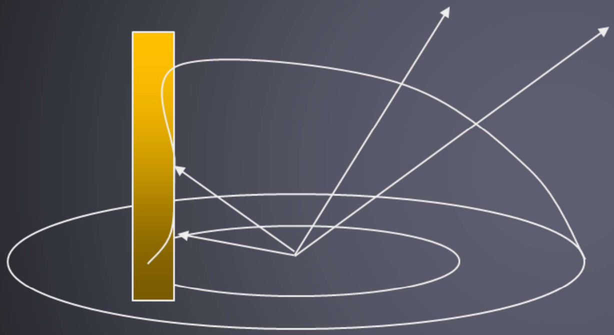
SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

But we don’t just store depth. We store a moving average of depth and depth square.



Radial Gaussian Depth



$$E(x) \approx Z_1 = \sum_n \frac{z}{n}$$

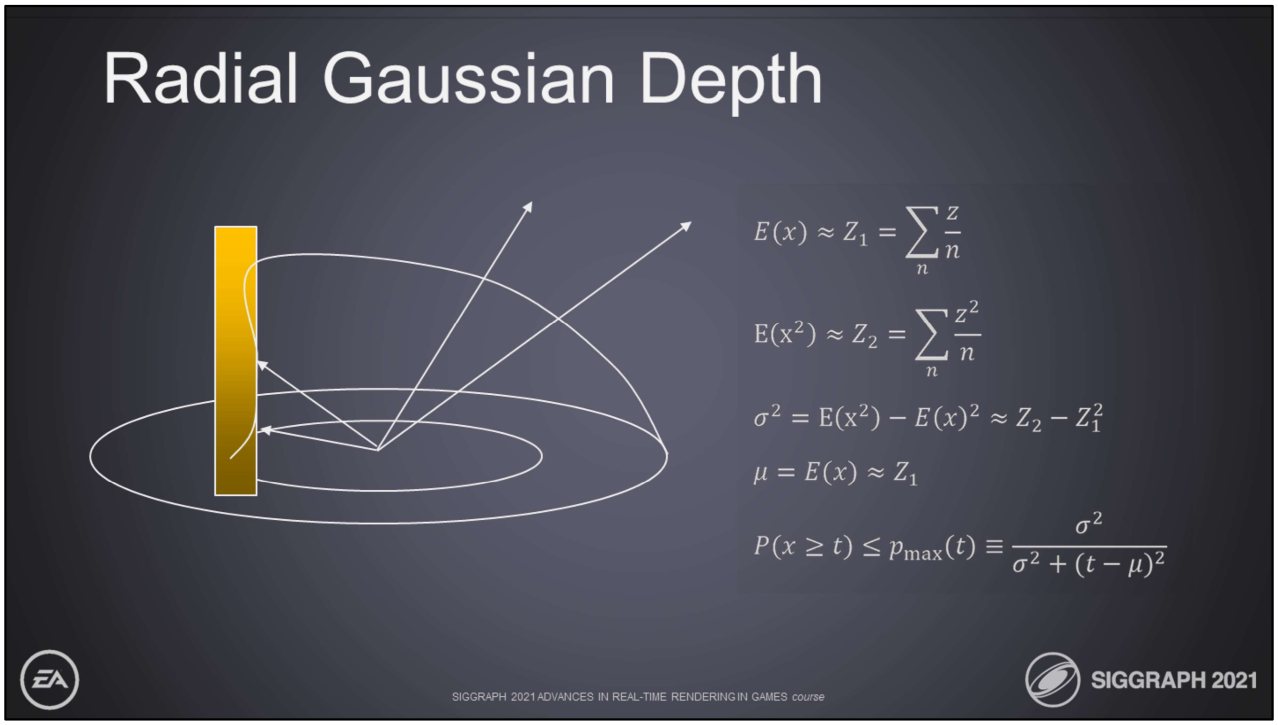
$$\mathrm{E}(\mathrm{x}^2) \approx Z_2 = \sum_n \frac{z^2}{n}$$

$$\sigma^2 = \mathrm{E}(\mathrm{x}^2) - E(x)^2 \approx Z_2 - Z_1^2$$

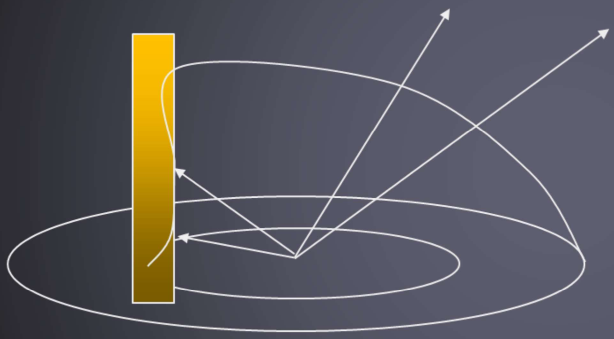
$$\mu = E(x) \approx Z_1$$

SIGGRAPH 2021

This allows us to re-create not just an estimate of the mean, but an estimate of the variance as well.



Radial Gaussian Depth



$$E(x) \approx Z_1 = \sum_n \frac{z}{n}$$

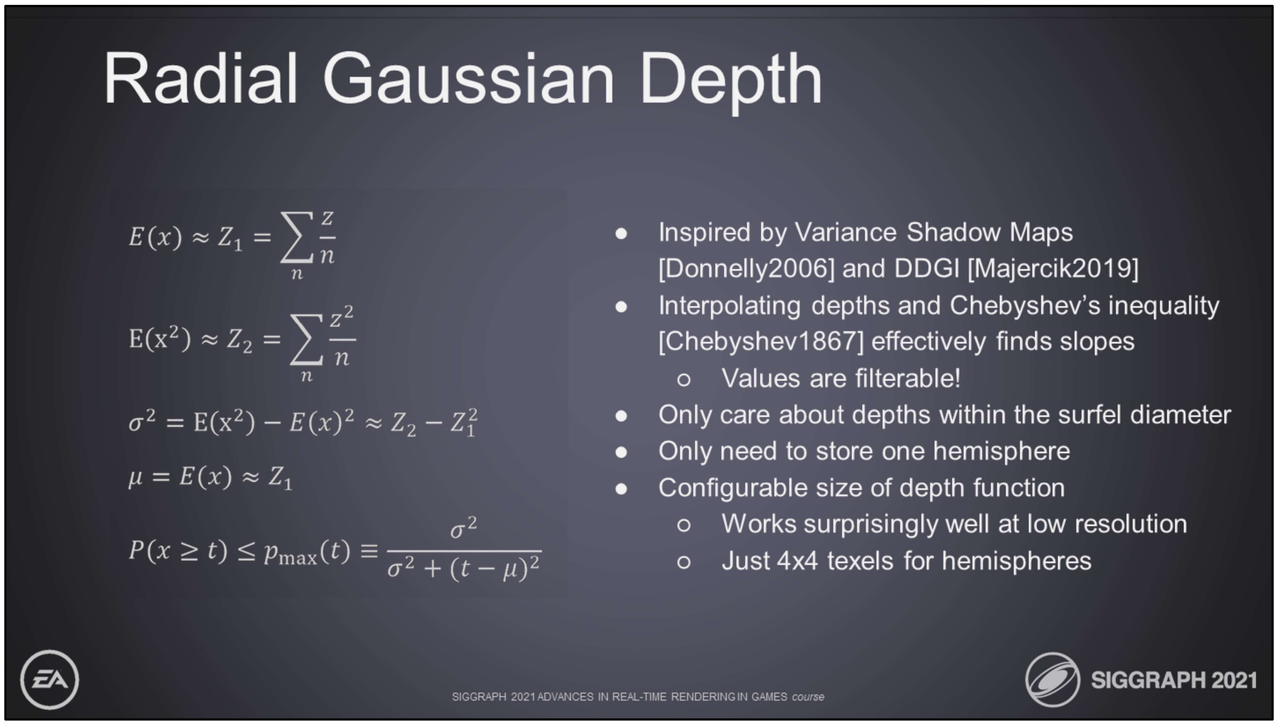
$$\mathrm{E}(\mathrm{x}^2) \approx Z_2 = \sum_n \frac{z^2}{n}$$

$$\sigma^2 = \mathrm{E}(\mathrm{x}^2) - E(x)^2 \approx Z_2 - Z_1^2$$

$$\mu = E(x) \approx Z_1$$

$$P(x \geq t) \leq p_{\max}(t) \equiv \frac{\sigma^2}{\sigma^2 + (t-\mu)^2}$$

We can use the mean and variance to use Chebychev's inequality to do depth testing. This provides a smooth depth test, and works great for re-creating slopes in depth space.



Radial Gaussian Depth

$$E(x) \approx Z_1 = \sum_n \frac{z}{n}$$

$$\mathrm{E}(\mathrm{x}^2) \approx Z_2 = \sum_n \frac{z^2}{n}$$

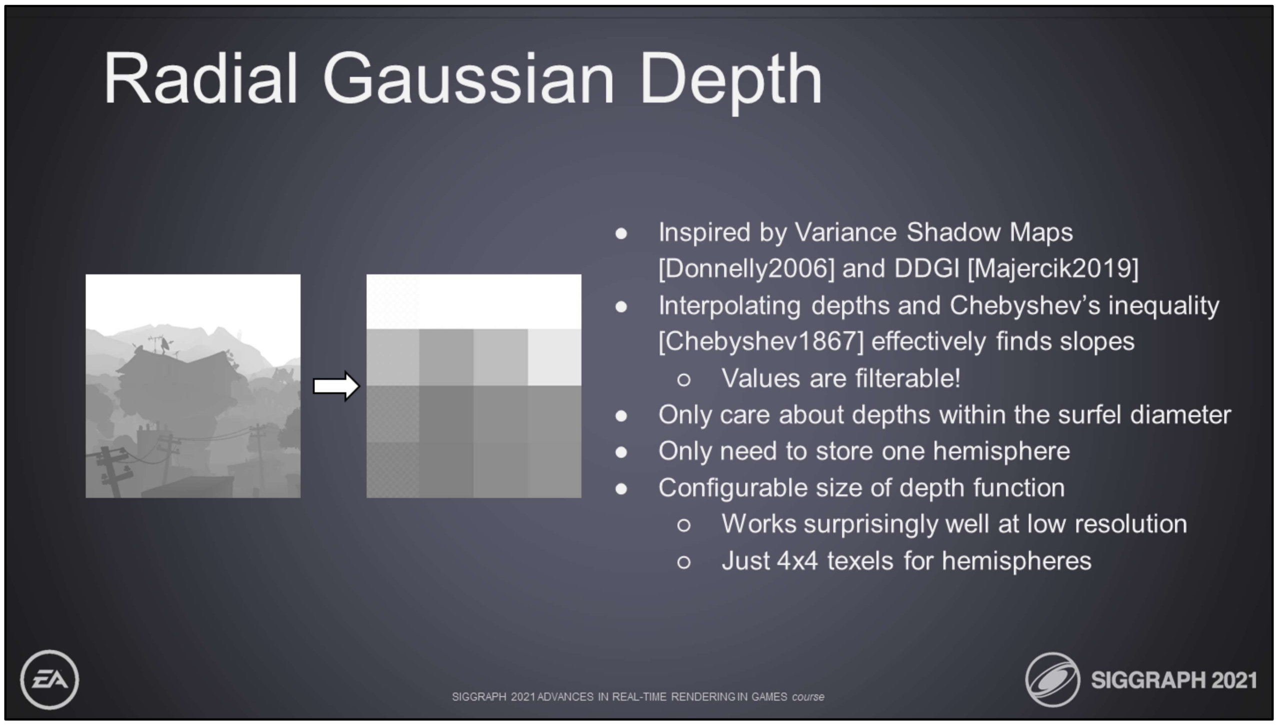
$$\sigma^2 = \mathrm{E}(\mathrm{x}^2) - E(x)^2 \approx Z_2 - Z_1^2$$

$$\mu = E(x) \approx Z_1$$

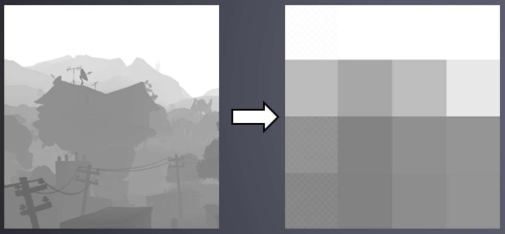
$$P(x \geq t) \leq p_{\max}(t) \equiv \frac{\sigma^2}{\sigma^2 + (t-\mu)^2}$$

- Inspired by Variance Shadow Maps [Donnelly2006] and DDGI [Majercik2019]
- Interpolating depths and Chebyshev's inequality [Chebyshev1867] effectively finds slopes
  - Values are filterable!
- Only care about depths within the surfel diameter
- Only need to store one hemisphere
- Configurable size of depth function
  - Works surprisingly well at low resolution
  - Just 4x4 texels for hemispheres

This technique is inspired by Variance Shadow Maps and DDGI. In contrast to the DDGI implementation, we only care about depth within the surfel diameter, and what's in the hemisphere.



Radial Gaussian Depth

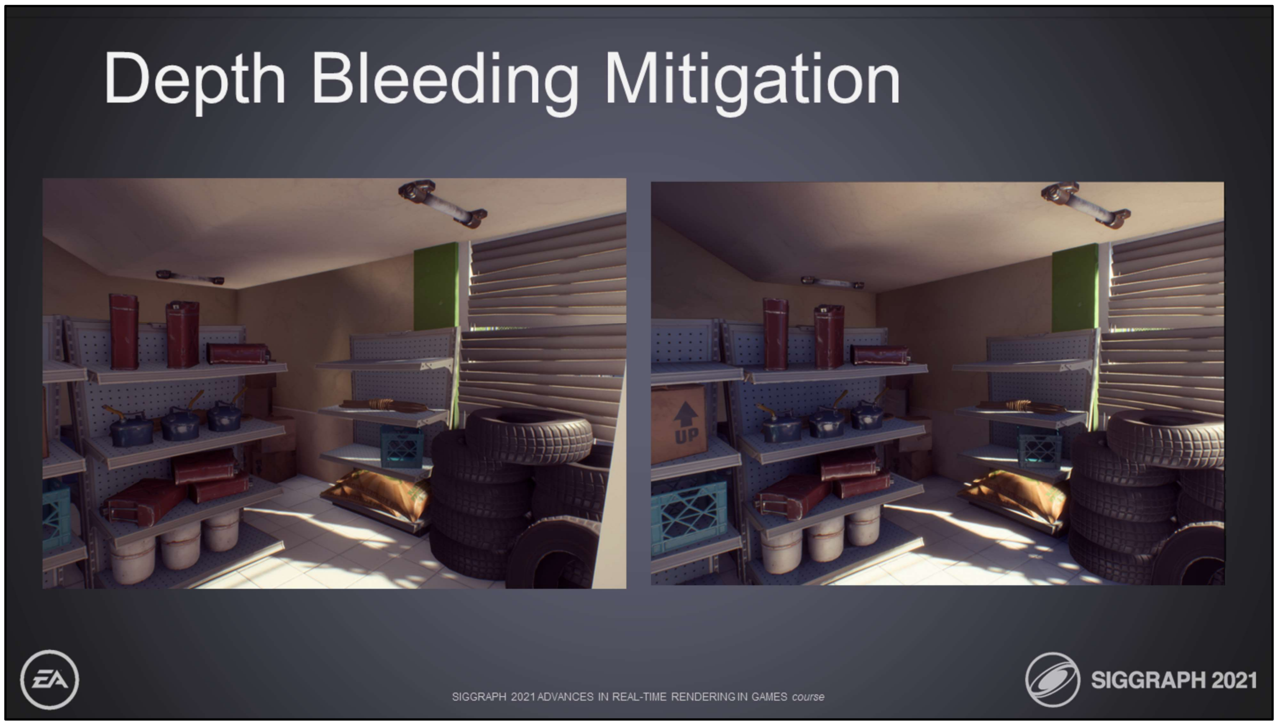


- Inspired by Variance Shadow Maps [Donnelly2006] and DDGI [Majercik2019]
- Interpolating depths and Chebyshev's inequality [Chebyshev1867] effectively finds slopes
    - Values are filterable!
- Only care about depths within the surfel diameter
- Only need to store one hemisphere
- Configurable size of depth function
    - Works surprisingly well at low resolution
    - Just 4x4 texels for hemispheres

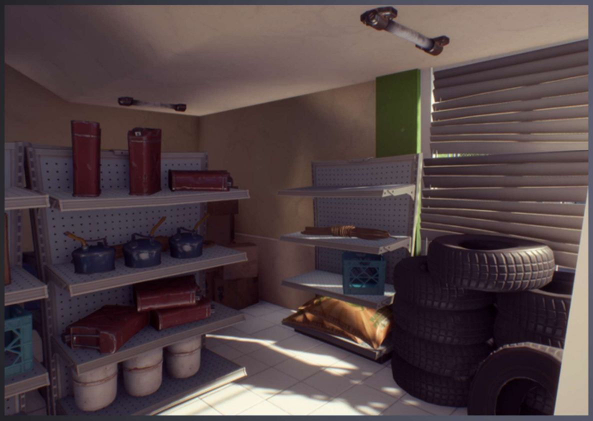
SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

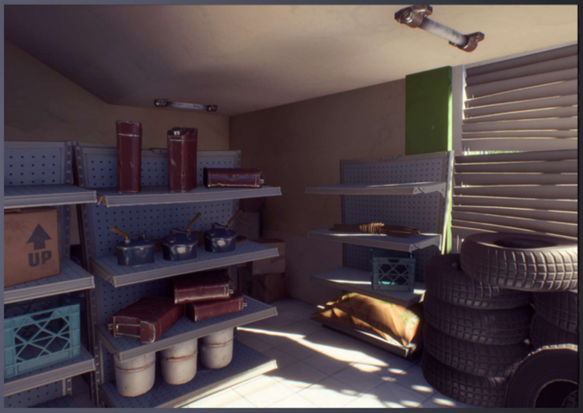
SIGGRAPH 2021

This works amazingly well, with very low resolution depth functions. The size is configurable, but our default is just 4x4 texels per surfel hemisphere.



Depth Bleeding Mitigation





SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

And this is enough for the vast majority of cases. Here is the same scene with the radial gaussian depth function enabled on the right. As you can see, all of those artifacts on the wall and in the ceiling are gone.

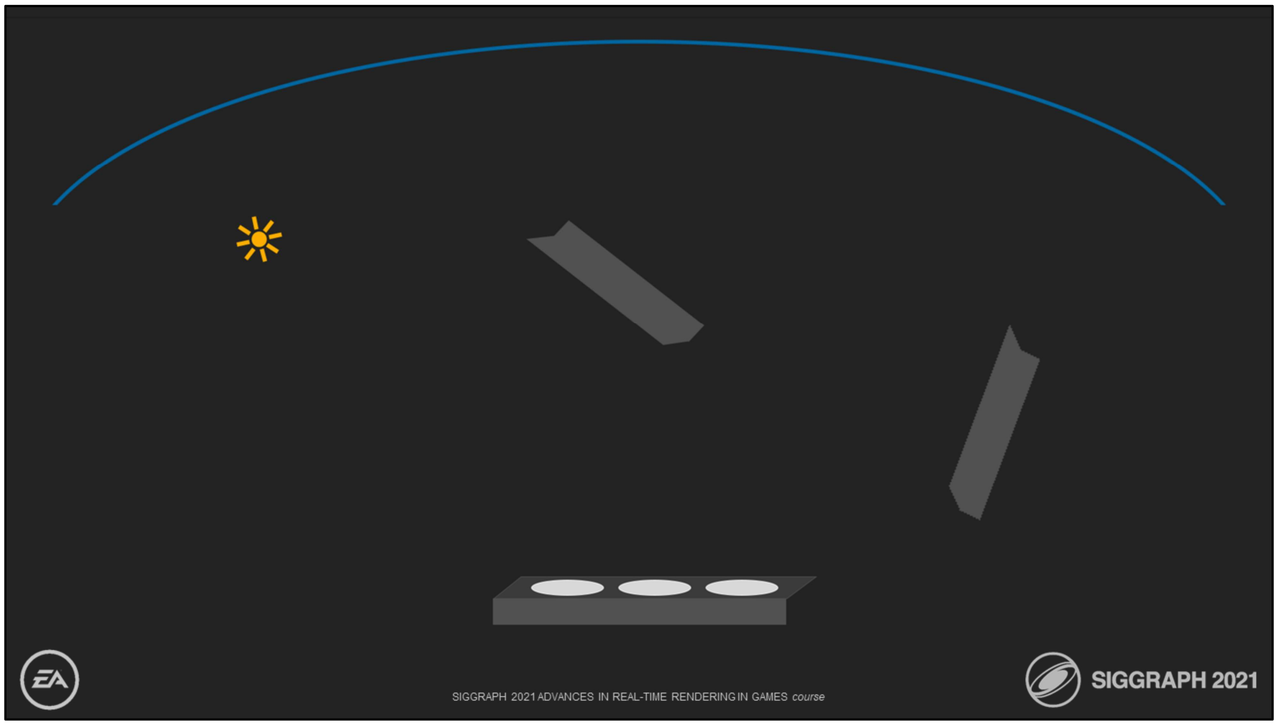
SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*
SIGGRAPH 2021

And the same is true for this scene as well.

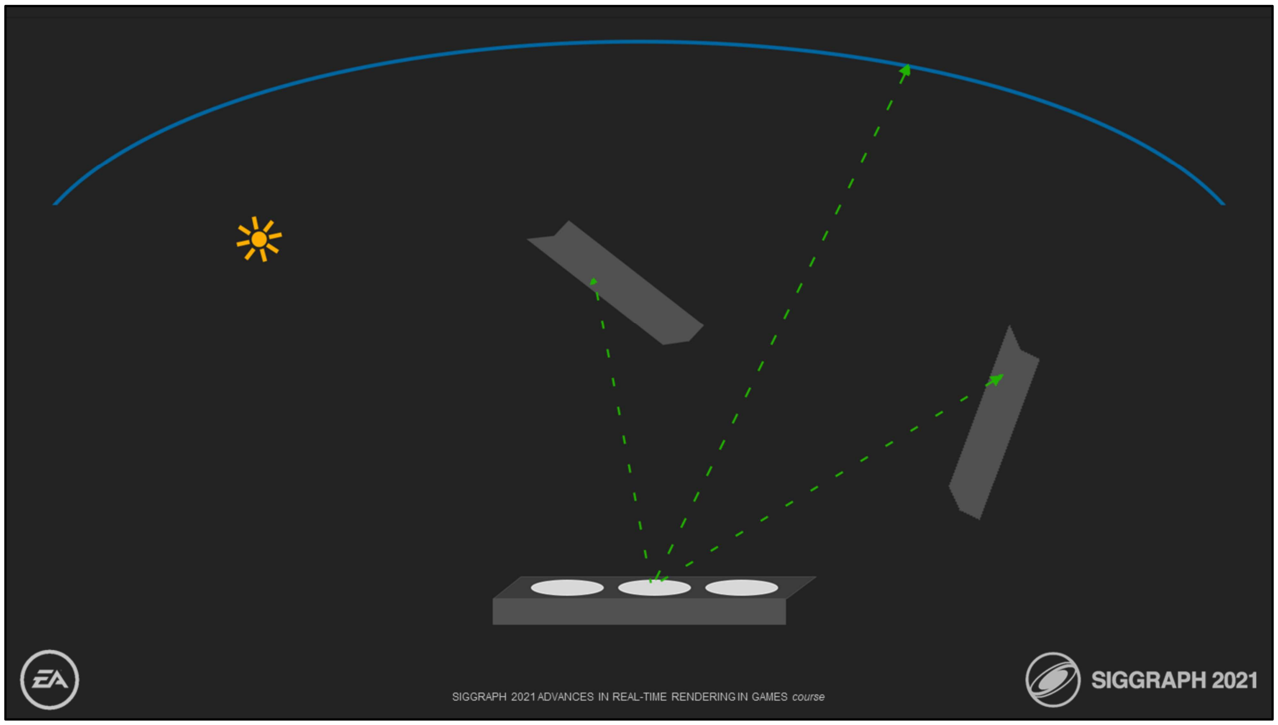
Integrating Irradiance

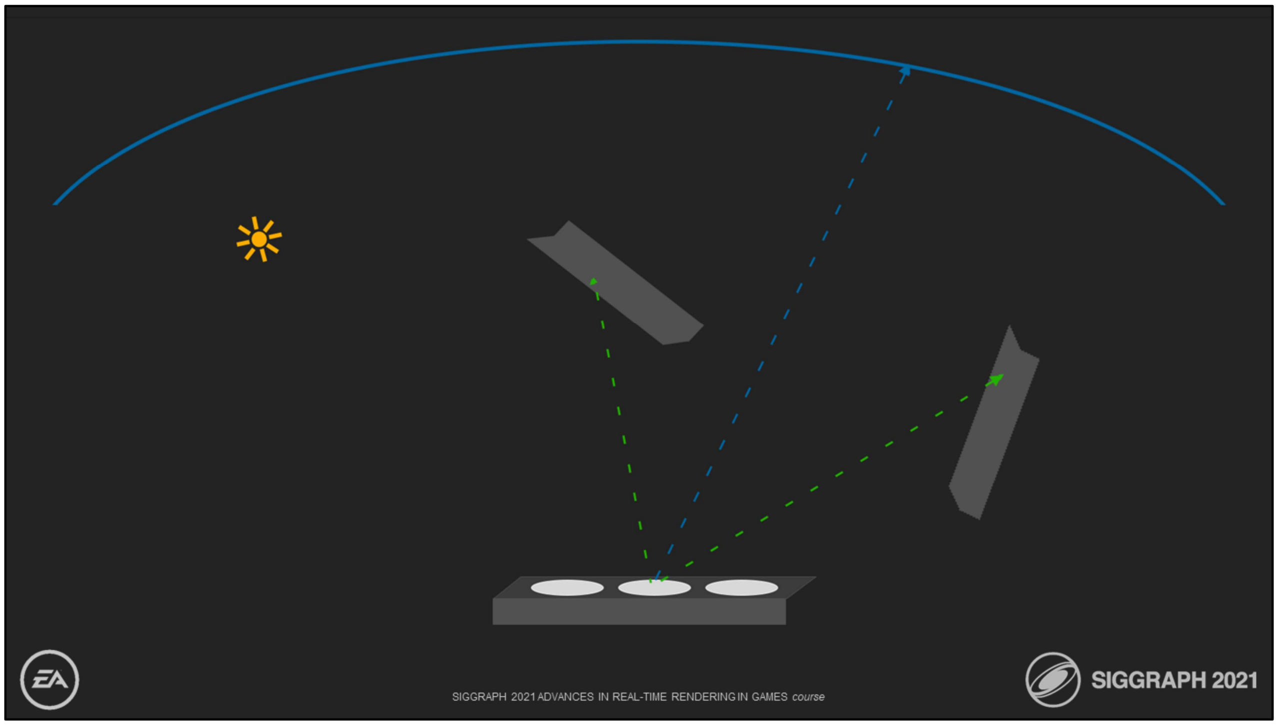
So how do we actually calculate lighting for surfels? We shoot rays into the scene, but we also employ a number of techniques to make sure we use the limited ray budget we have in the best way possible.

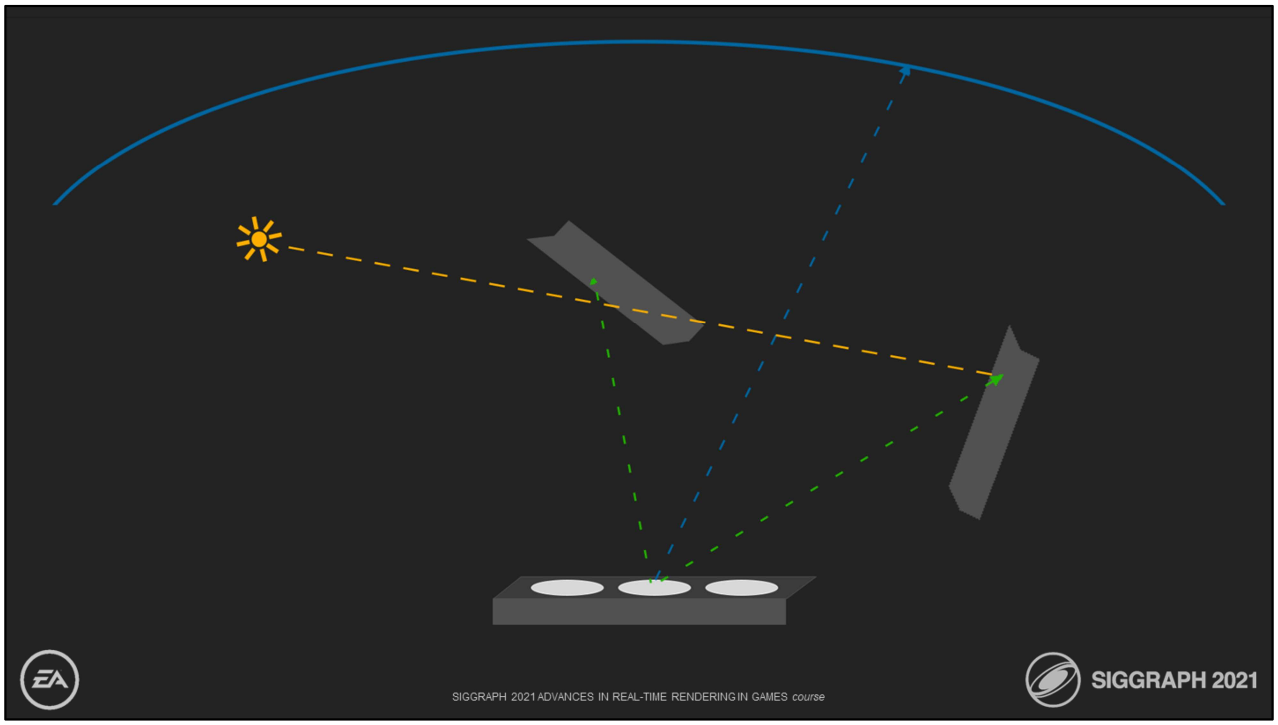
Since everything is dynamic within GIBS, we can support emissive surfaces and materials, as well as destructible and dynamic environments. We need to be able to effectively integrate irradiance in these situations and react to changes in the scene.

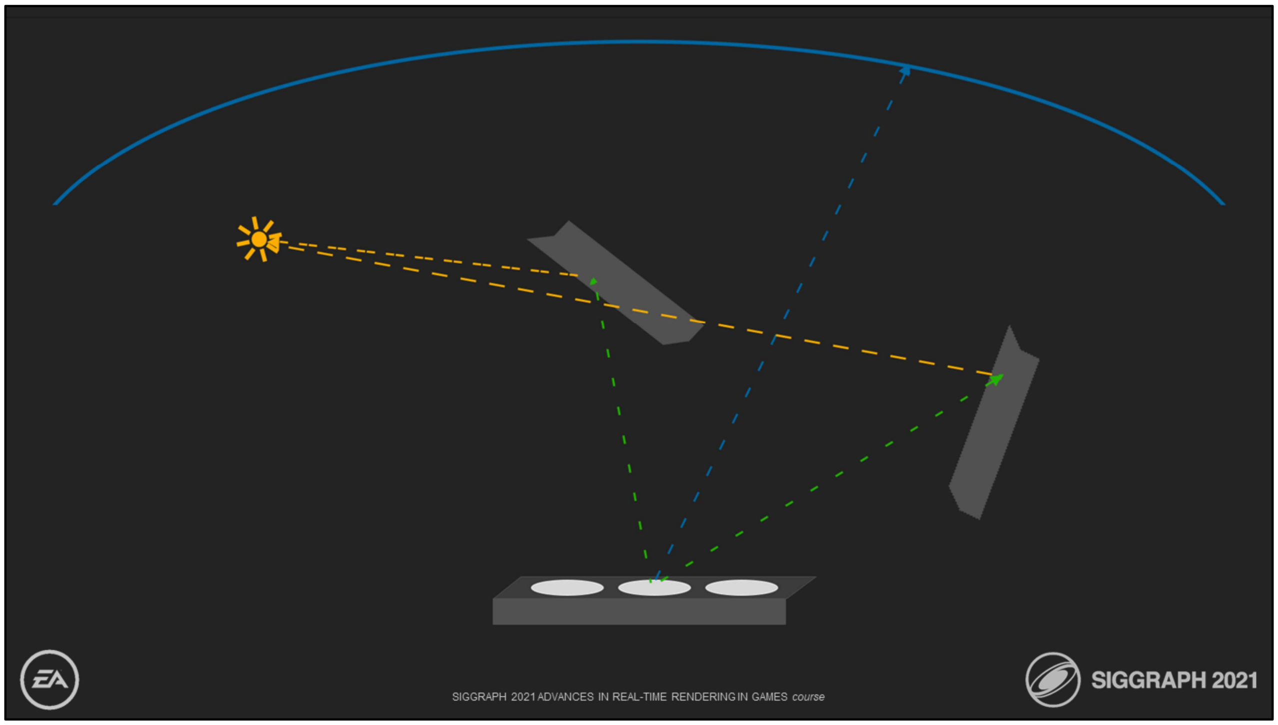

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*
SIGGRAPH 2021

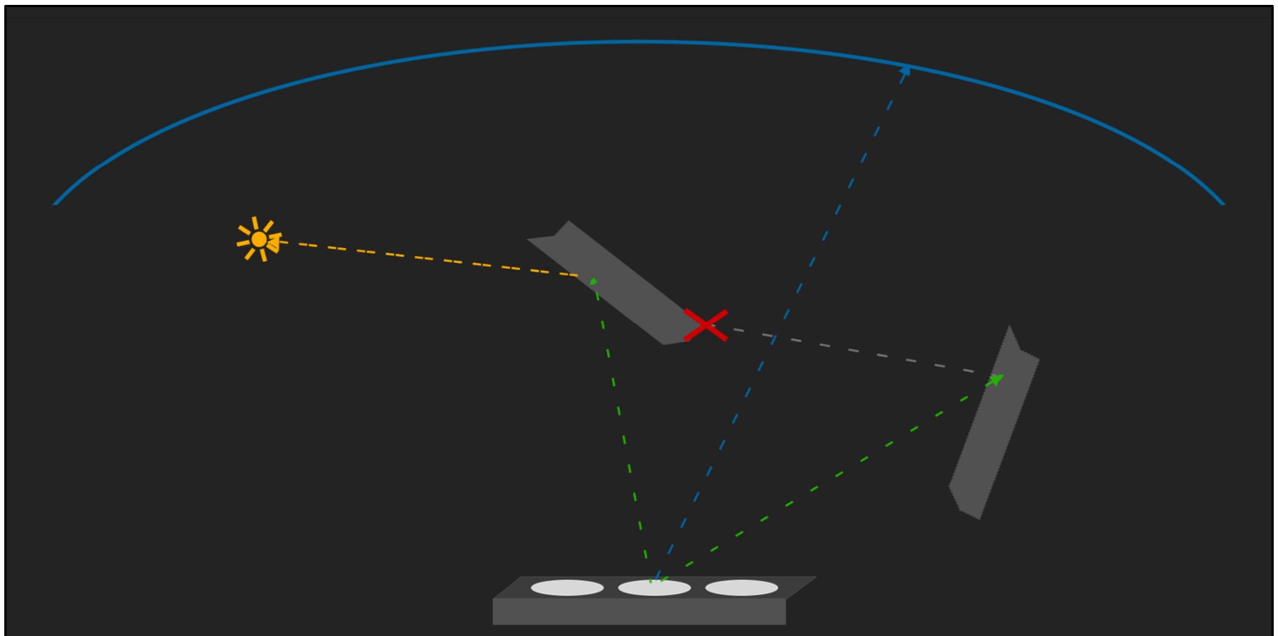
This is where ray tracing comes in. Surfels shoot rays into the scene. These rays hit geometry, any geometry that is in the ray tracing acceleration structure, including dynamic and skinned geometry. On a hit point, we evaluate direct diffuse lighting and shadowing by tracing the light sources in the scene. We will talk about how we do that efficiently a bit later. Additionally at the hit locations we also evaluate the existing surfel lighting at those locations. This gives us effectively infinite bounce over time, as long as there is surfel coverage in that area.



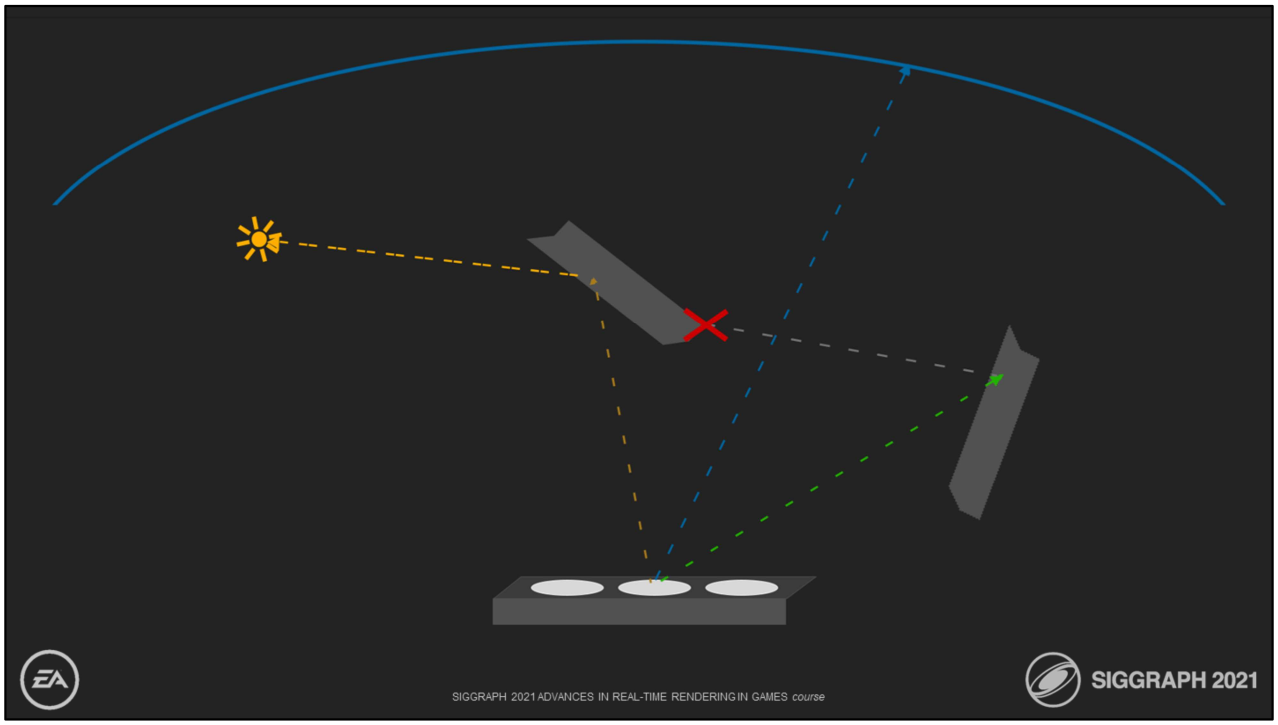


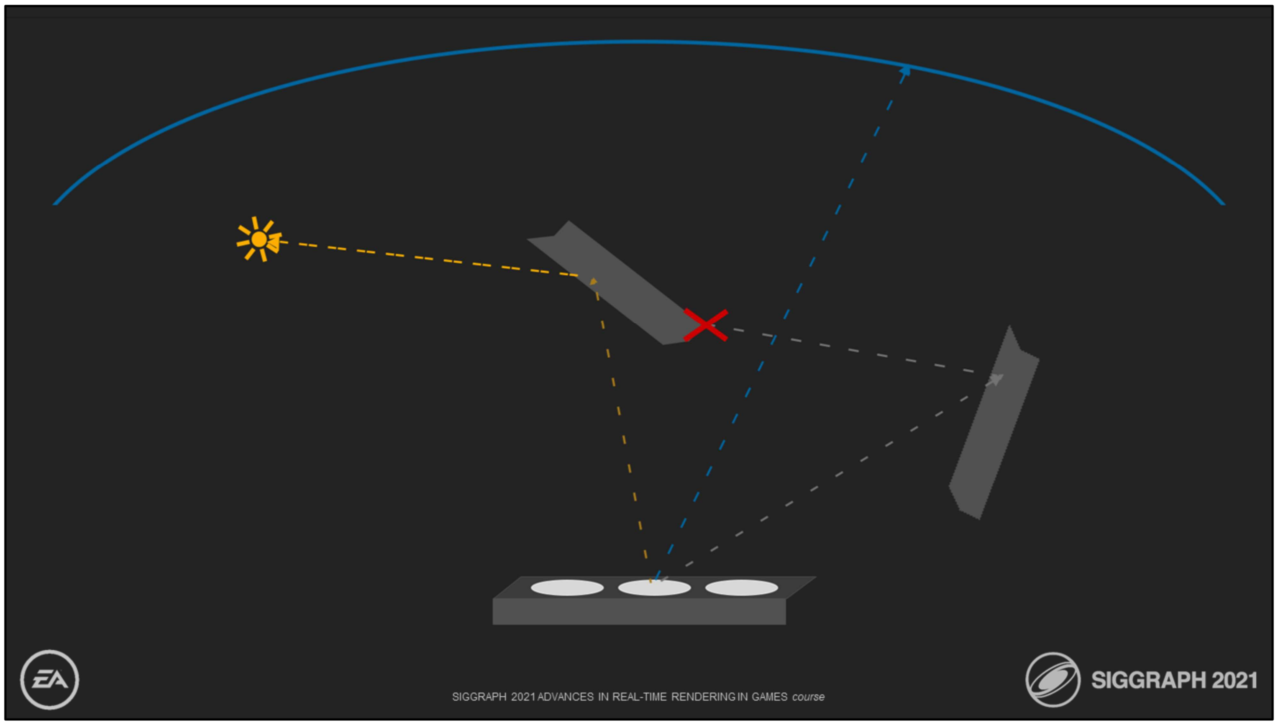


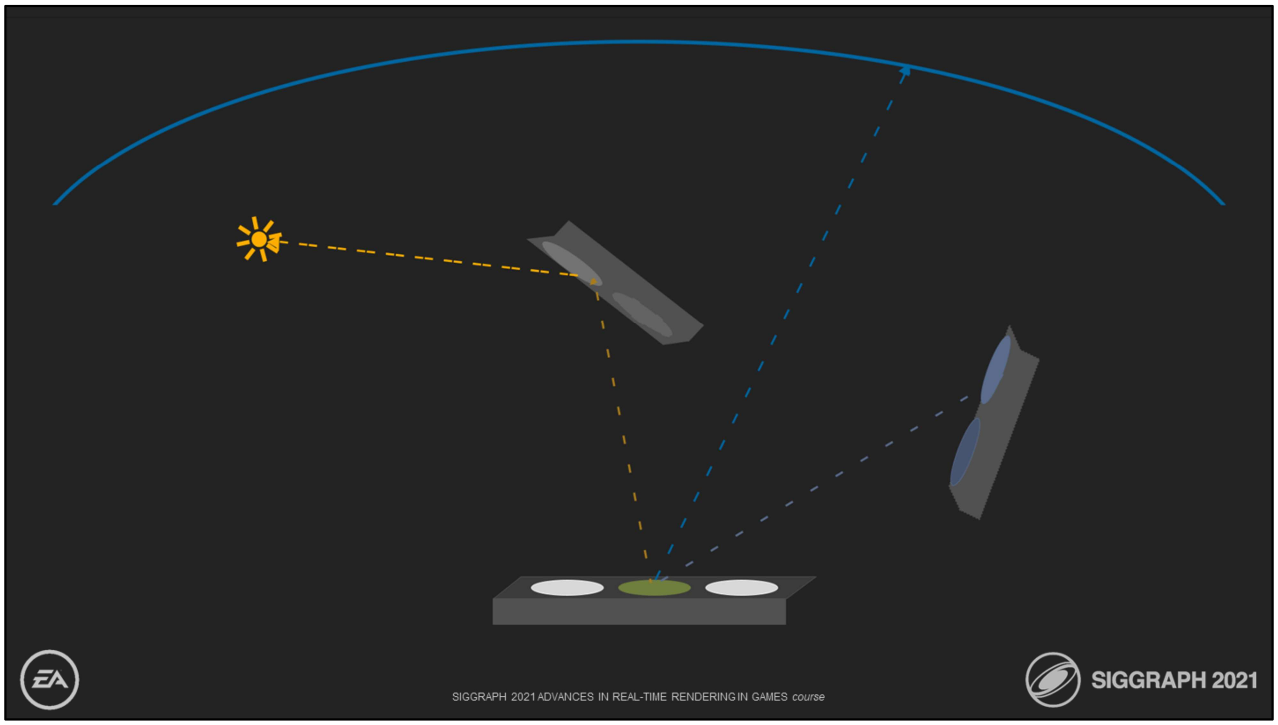


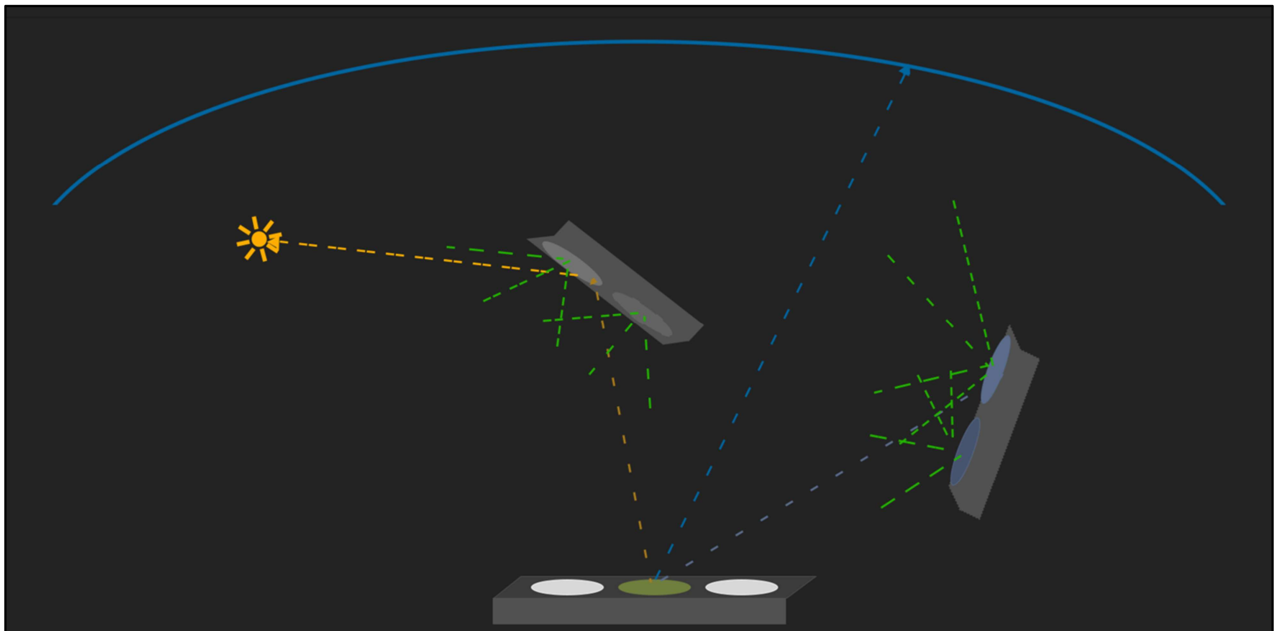


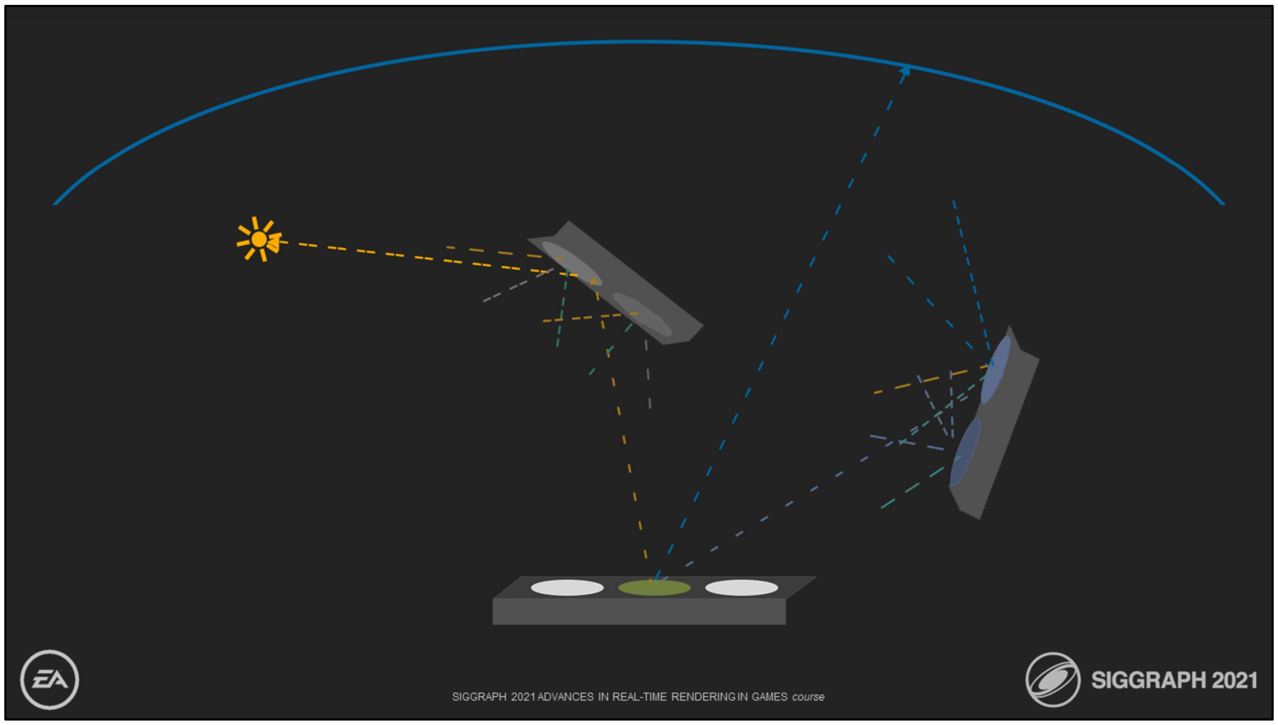
SIGGRAPH 2021

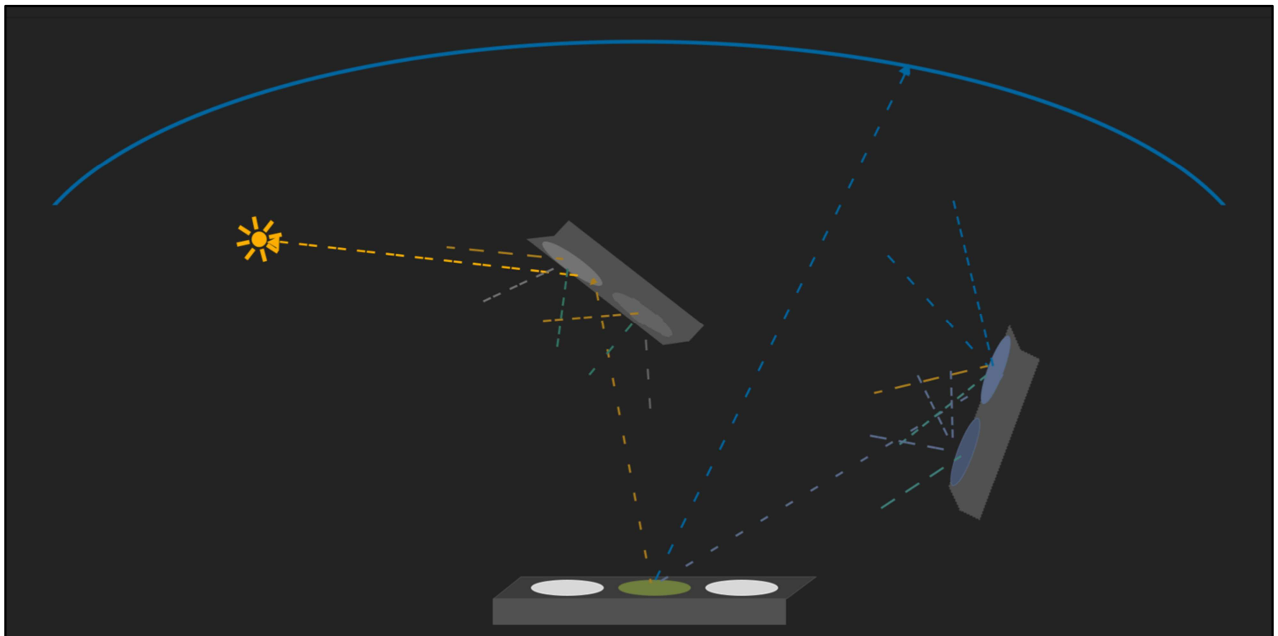




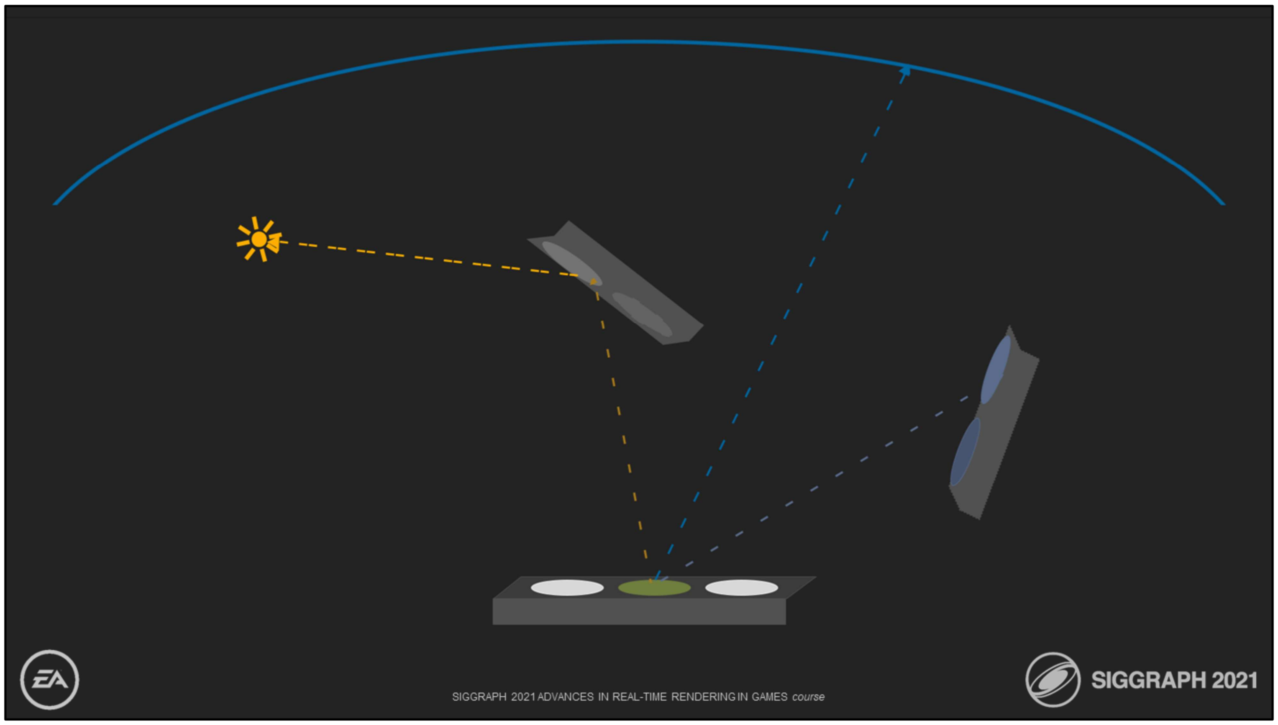


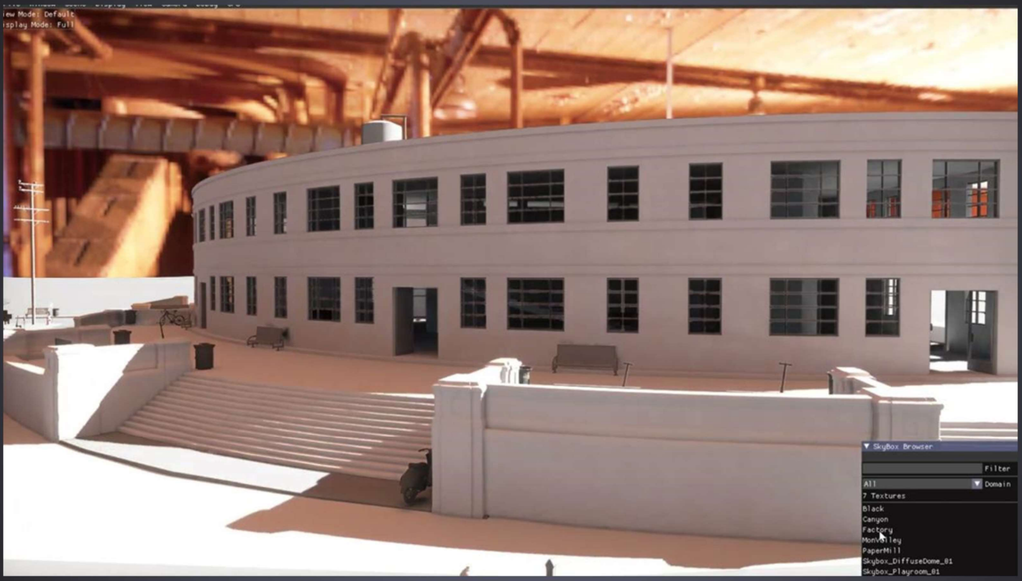




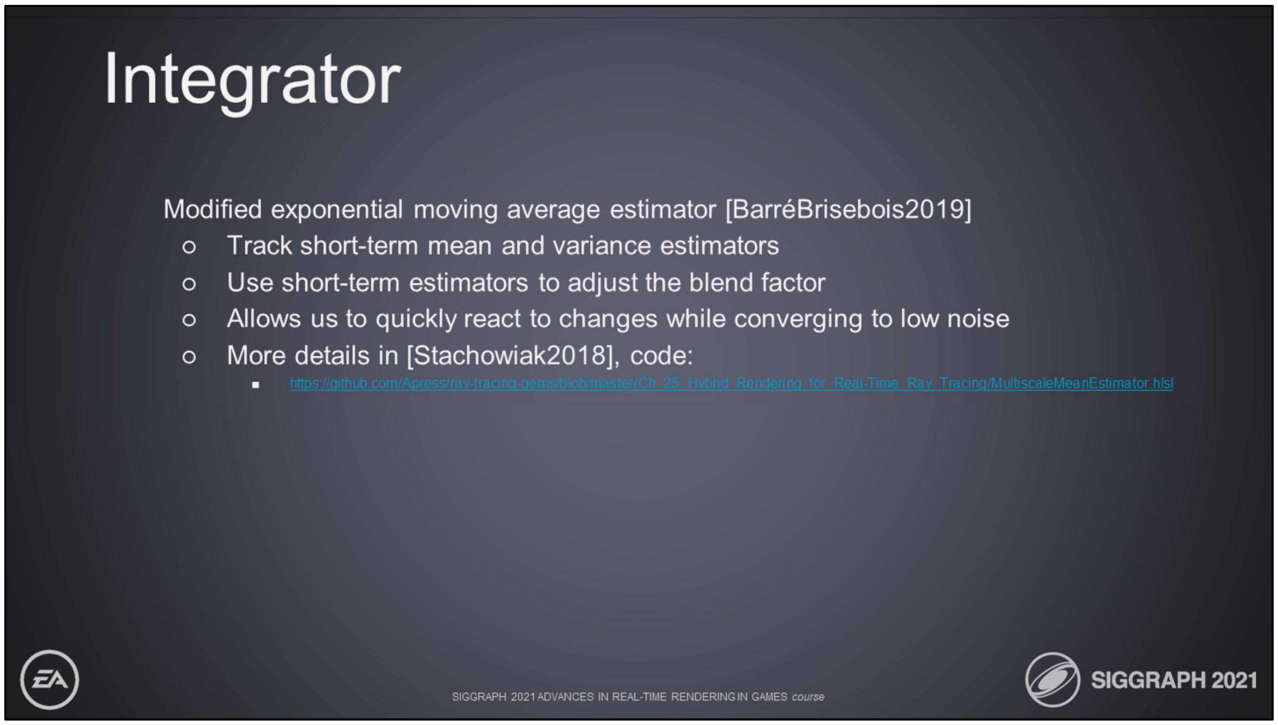


SIGGRAPH 2021





Since we want to support dynamic environments and reduce the cost of ray tracing, we accumulate lighting over many frames.



Integrator

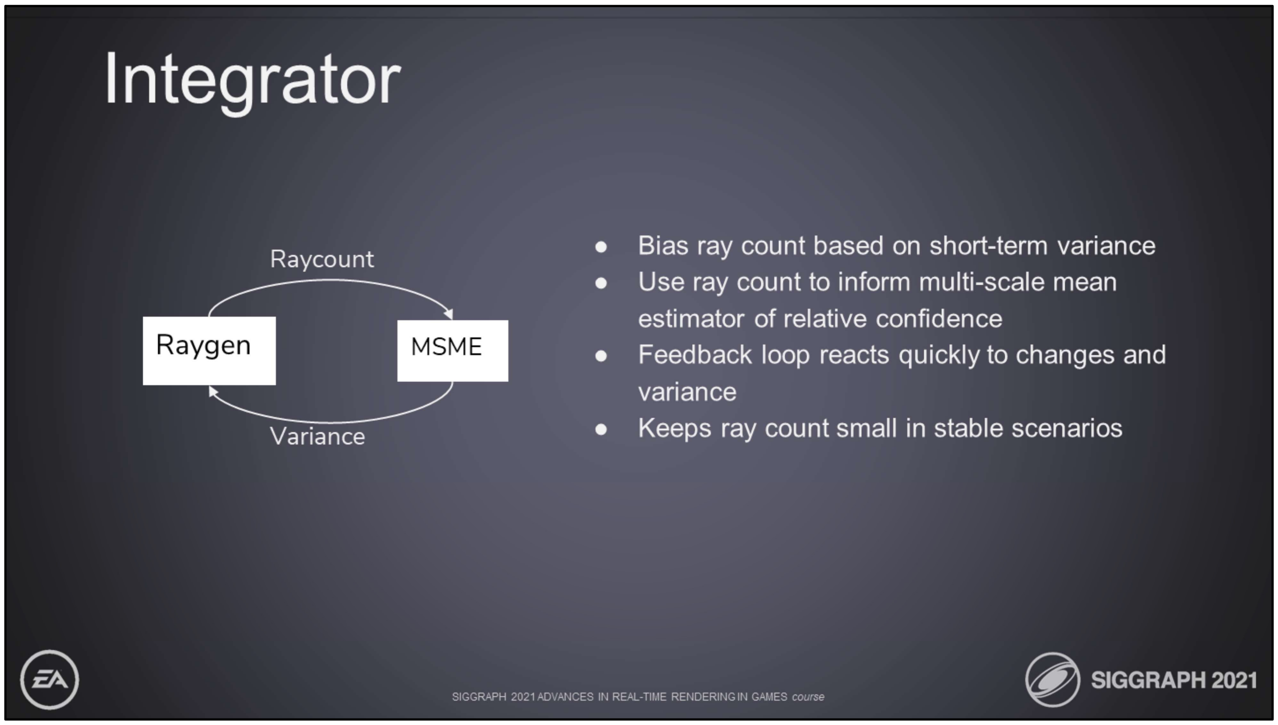
Modified exponential moving average estimator [BarréBrisebois2019]

- Track short-term mean and variance estimators
- Use short-term estimators to adjust the blend factor
- Allows us to quickly react to changes while converging to low noise
- More details in [Stachowiak2018], code:
  - https://github.com/Apress/ray-tracing-gems/blob/master/Ch_25_Hybrid_Rendering_for_Real-Time_Ray_Tracing/MultiscaleMeanEstimator.hlsl

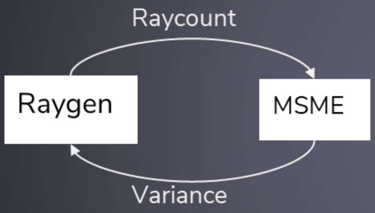
SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

To do this, we use a modified version of a moving average estimator.
With a regular moving average estimator you have to choose a blend factor for each sample. This means you have the choice of either something that's reactive to changes in the scene, *or* something that converges well over time.
What we really want is the best of both worlds, and we want it to be automatic.
So in addition to accumulating a longer term moving average, we also track a shorter term mean and a short term variance estimator.
These *short-term* estimators are used to adjust the blend factor for the long term average.
This allows us to quickly react to changes in the scene, while also converging to a noise free result.
I will again recommend you check out Thomasz's presentation for slightly more details there.

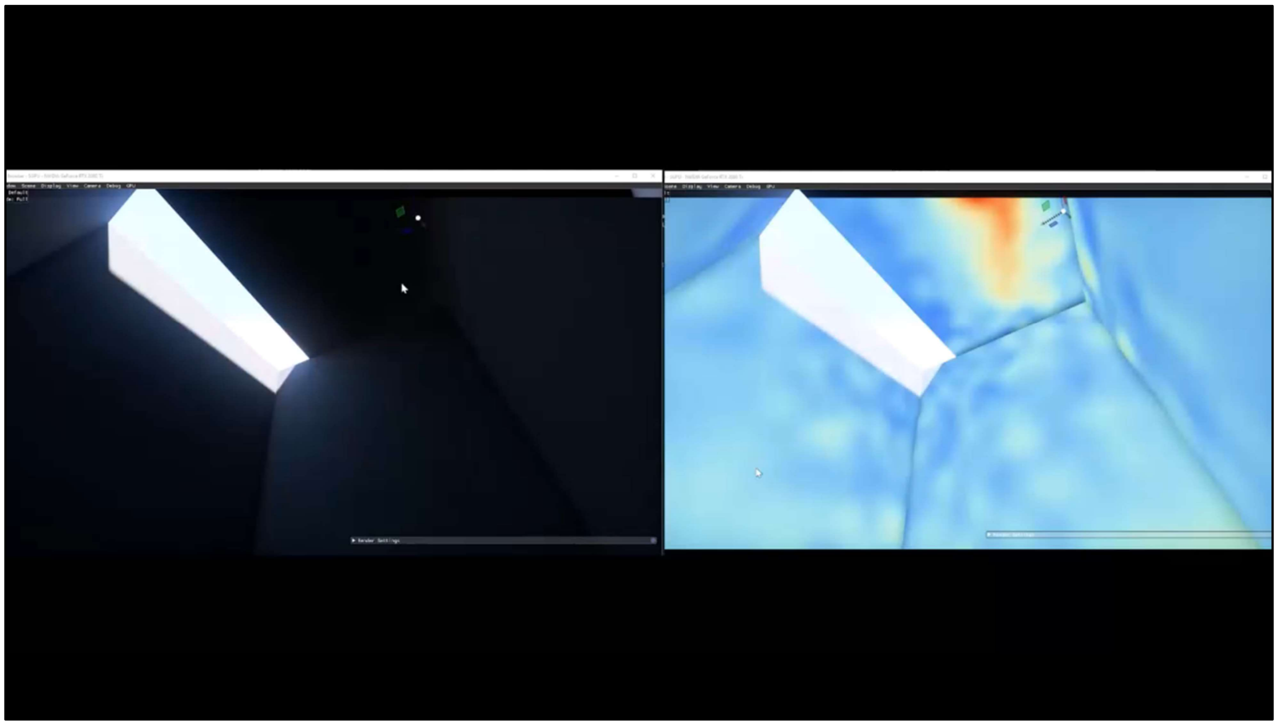


Integrator

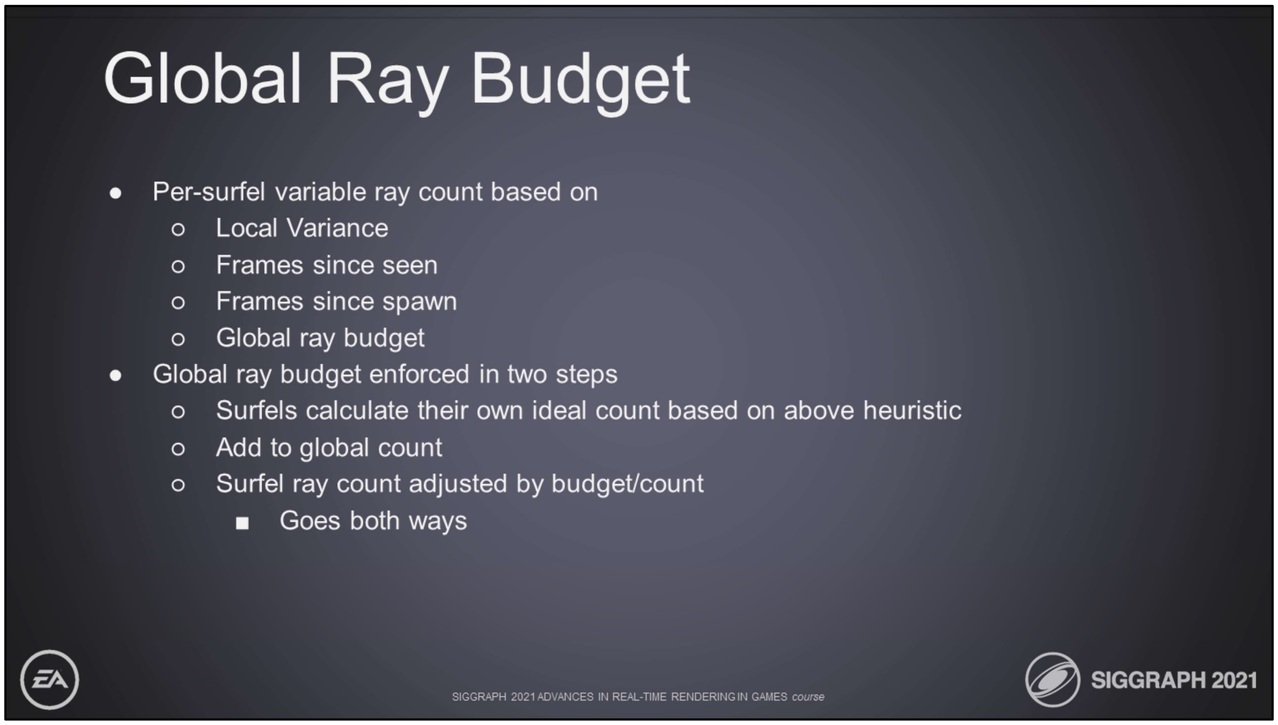


- Bias ray count based on short-term variance
- Use ray count to inform multi-scale mean estimator of relative confidence
- Feedback loop reacts quickly to changes and variance
- Keeps ray count small in stable scenarios

Since we are tracking variance per surfel, we can make good use of this for other purposes as well.
The number of rays sent from each surfel every frame depends on a few different factors, one of which is the variance the surfel sees.
If a surfel is seeing high variance, it requests more rays in order to converge faster.
In contrast, if a surfel sees low variance, it can send much fewer rays.
We use this to quickly react to changes in the scene, while reducing the overall ray count when things converge. In fact, surfels can go into an almost dormant state, just sending enough rays to detect that there is a change.



As a visualization of this working, in this video we visualize the variance. Blue is very low variance and red is high variance. As the scene starts, variance is low, and consequently surfels send very few rays. As the sun moves, surfels start seeing more variance because the current mean is different from the previous mean. So the surfels quickly up the amount of rays sent and again converge to a stable low variance state. This is also true if we exit this area and spawn completely new surfels. When surfels first spawn variance is high because it takes some time to converge, but as we do converge variance drops and surfels can go down to their sort of dormant low ray counts. For this reason we allow more rays for newly spawned surfels.

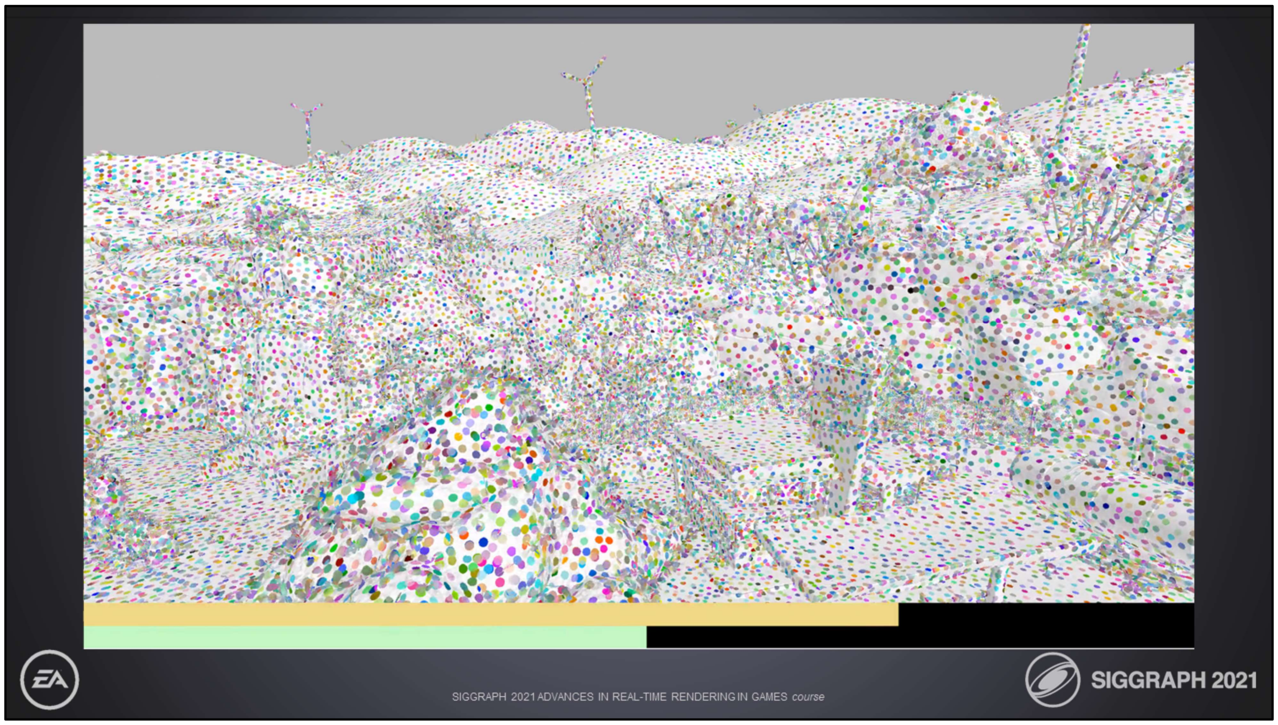


Global Ray Budget

- Per-surfel variable ray count based on
  - Local Variance
  - Frames since seen
  - Frames since spawn
  - Global ray budget
- Global ray budget enforced in two steps
  - Surfels calculate their own ideal count based on above heuristic
  - Add to global count
  - Surfel ray count adjusted by budget/count
    - Goes both ways

Surfels use a number of pieces of information to determine how many rays they would like to send each frame. These factors include how many frames have gone by since a surfel actually contributed to the lighting the camera sees. Surfels that are actively contributing lighting to what the camera sees get more rays. We allow surfels that just spawn to have a higher budget, so that we can quickly reduce variance. We also use variance to bias the ray counts.

To keep performance consistent across many areas we allow content creators to set a total ray budget for their game. The ray budget is enforced by a two step process. First a ray count pass is run, where each surfel increments a global ray counter according to the local surfel ray request as we just described. Second, as rays are allocated in the binning pass, which we will describe later, each surfel gets a ray count proportional to its requested part of the total budget. This allows us to balance performance with unpredictable lighting scenarios, and allows content creators to influence that balance.

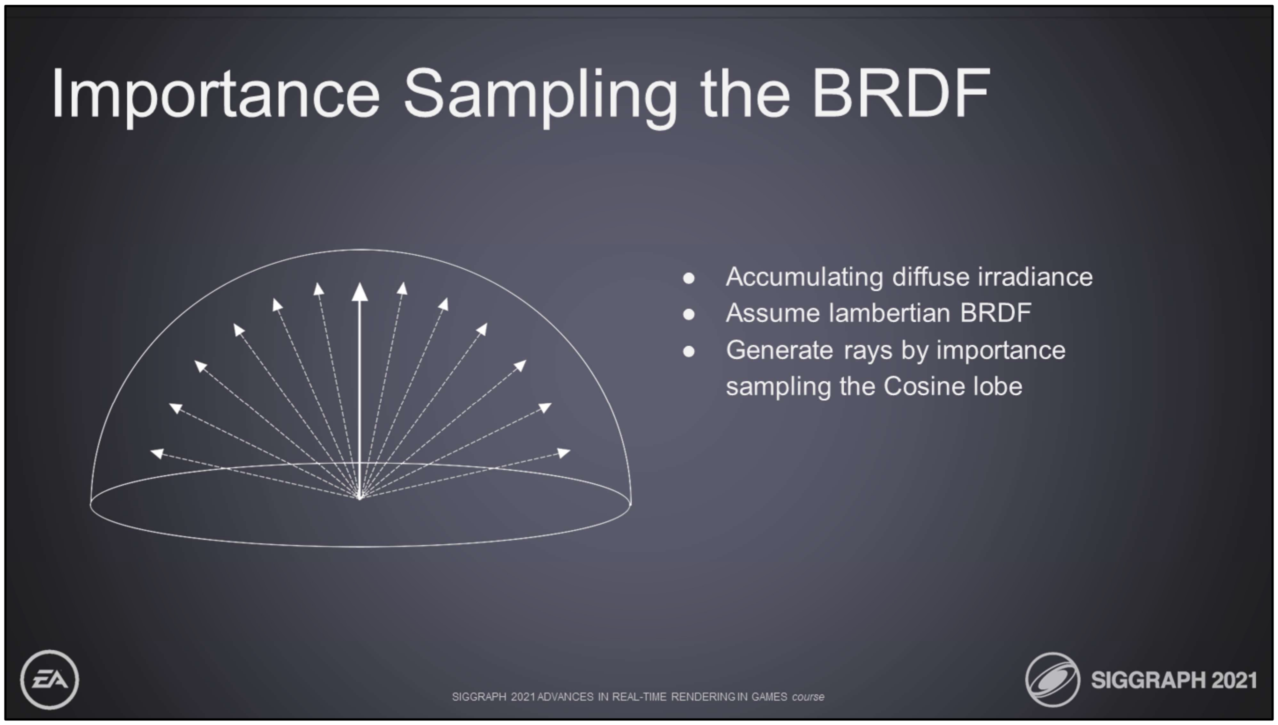

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

Here you can see an example of this in action. On the the right is a visualization of variance, blue indicates low variance, and red indicates high variance. As surfels detect a change in the scene, they ramp up their ray counts in order to converge to the new solution. When variance drops they go back to a lower almost dormant state, where they send just enough rays to detect a change.

The same is true in this scene. When the light changes, variance goes up, surfels increase their ray counts and converge to the new solution, and then go back to their almost dormant state.

This is true when we spawn new surfels as well, like here. The new surfels see a lot of variance because they have not converged yet, but quickly do so and reduce their ray counts.



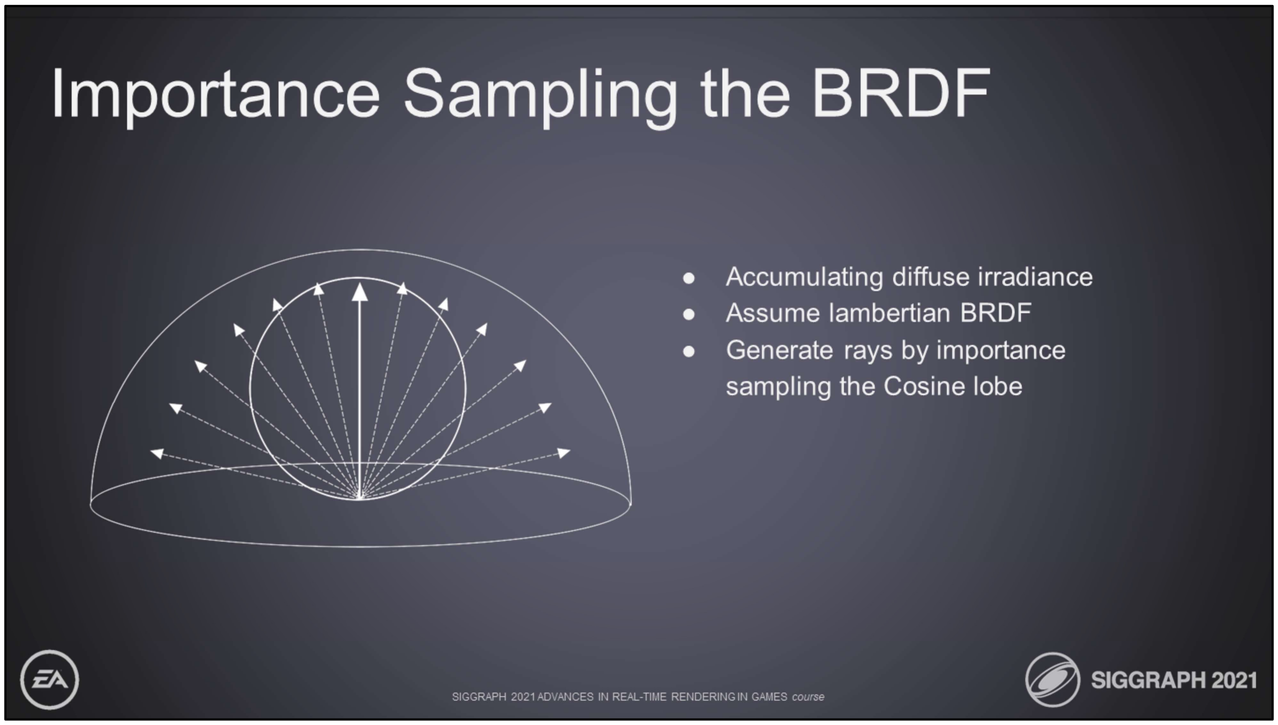
Importance Sampling the BRDF

- Accumulating diffuse irradiance
- Assume lambertian BRDF
- Generate rays by importance sampling the Cosine lobe

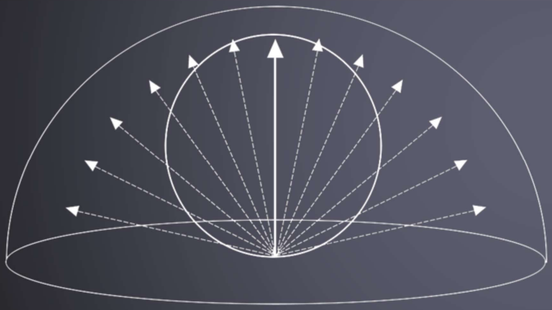
SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

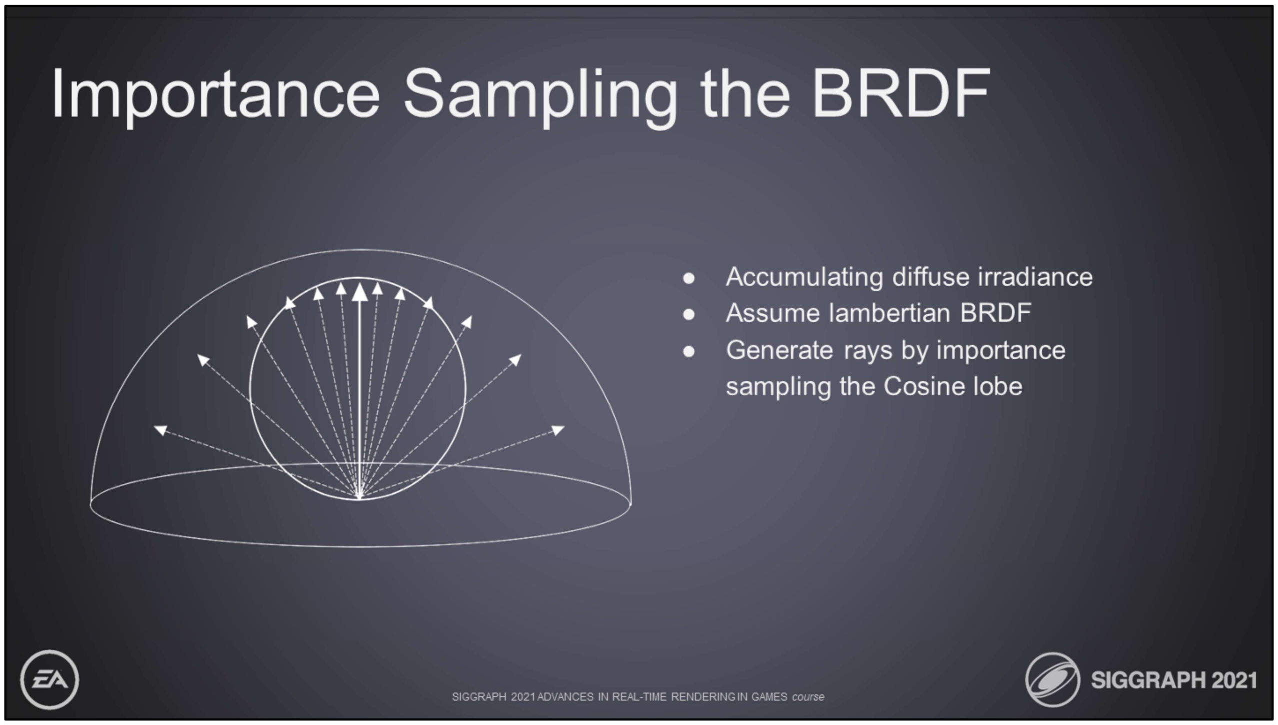
But we can do more in terms of where we are sending the rays as well. Since we are dealing with diffuse indirect lighting, the BRDF we are dealing with is essentially the cosine lobe. The most obvious thing to do is to importance sample that lobe, which we do individually for each surfel. But there are situations where the cosine lobe is not the most efficient direction to send your rays.



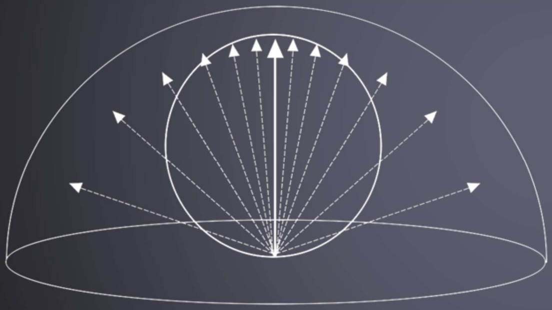
Importance Sampling the BRDF



- Accumulating diffuse irradiance
- Assume lambertian BRDF
- Generate rays by importance sampling the Cosine lobe

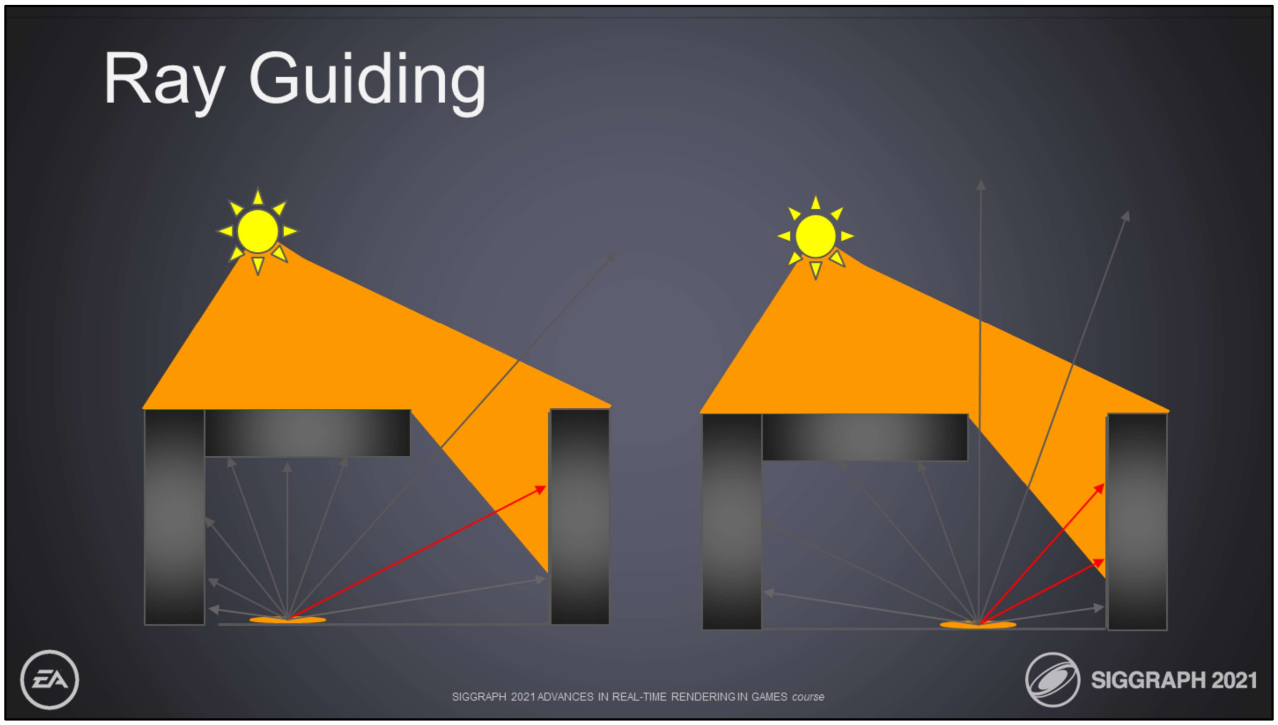


Importance Sampling the BRDF

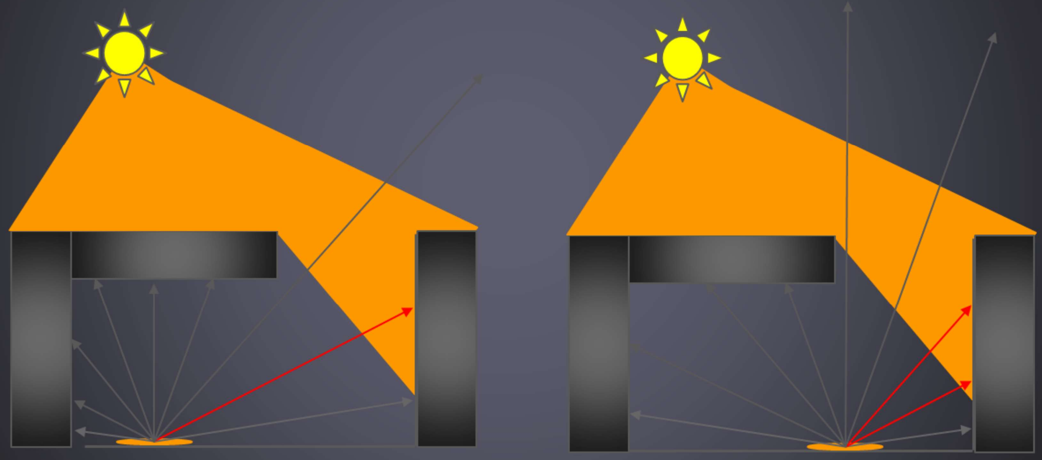


- Accumulating diffuse irradiance
- Assume lambertian BRDF
- Generate rays by importance sampling the Cosine lobe

SIGGRAPH 2021

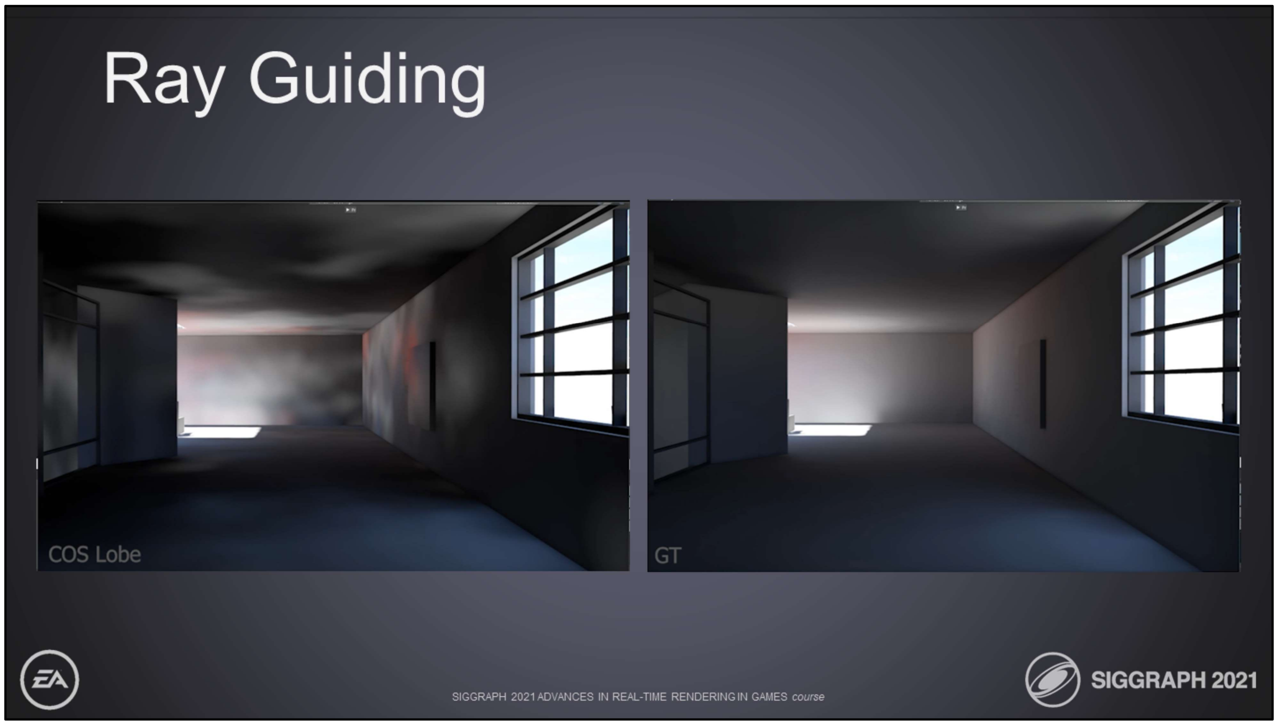


Ray Guiding

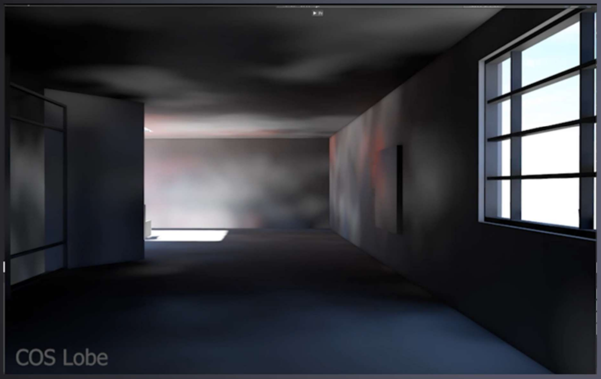


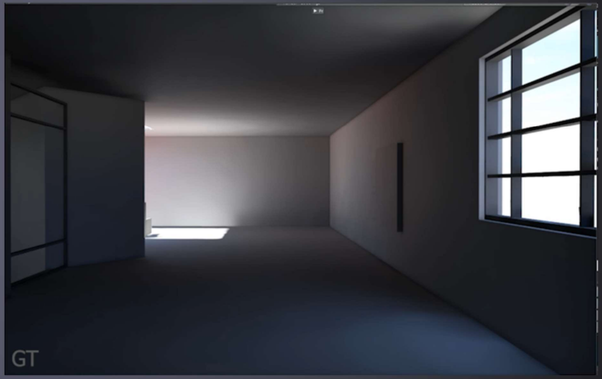
SIGGRAPH 2021

While importance sampling the cosine lobe works well in many scenes, this is not always the case. Here we have two surfels side-by-side in a room where most of the light comes from the wall on the right, which is hit directly by the sunlight. These surfels send the same number of rays in the same directions, but the one of the right ends up hitting this bright area twice as often as the one on the left. So both of these surfels have a bad estimate of what the irradiance is.



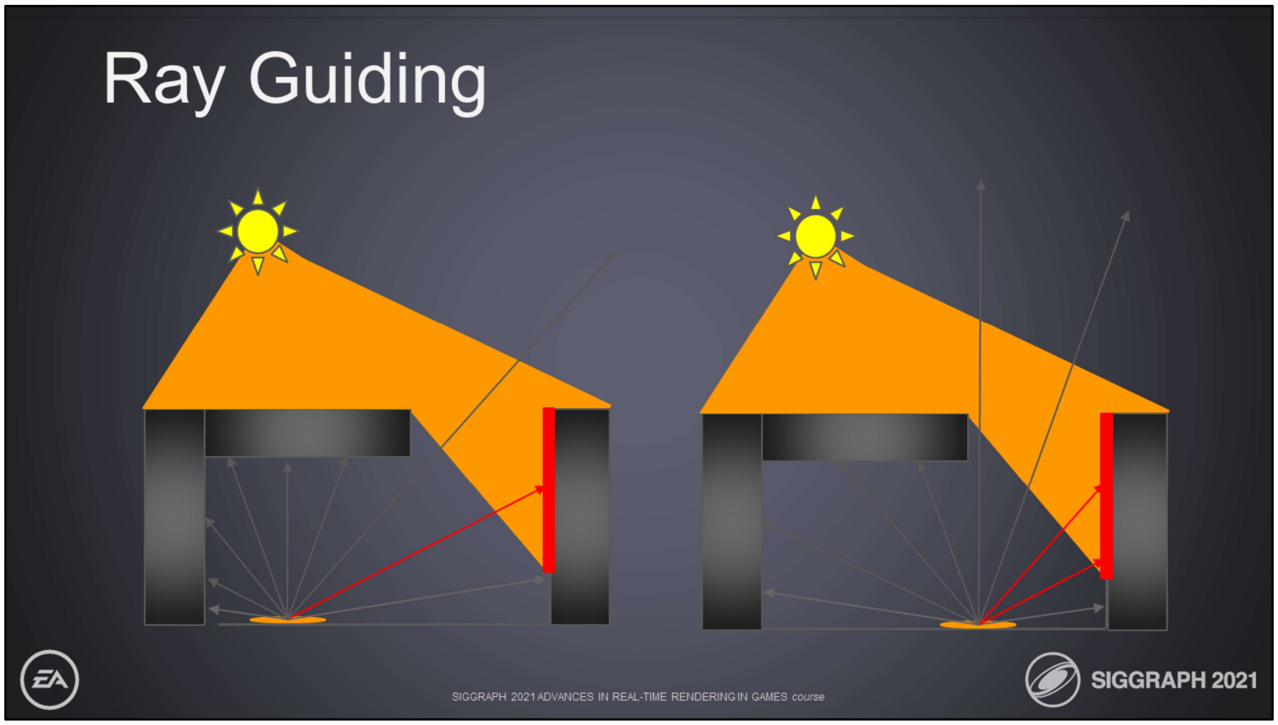
Ray Guiding



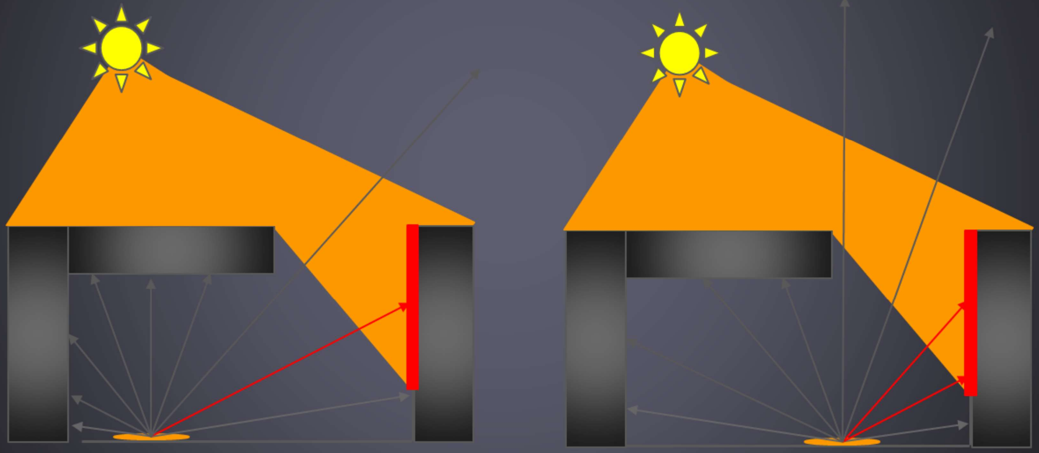


SIGGRAPH 2021

What this looks like in practice, as you can see on the left, is noice, or variance across surfels.



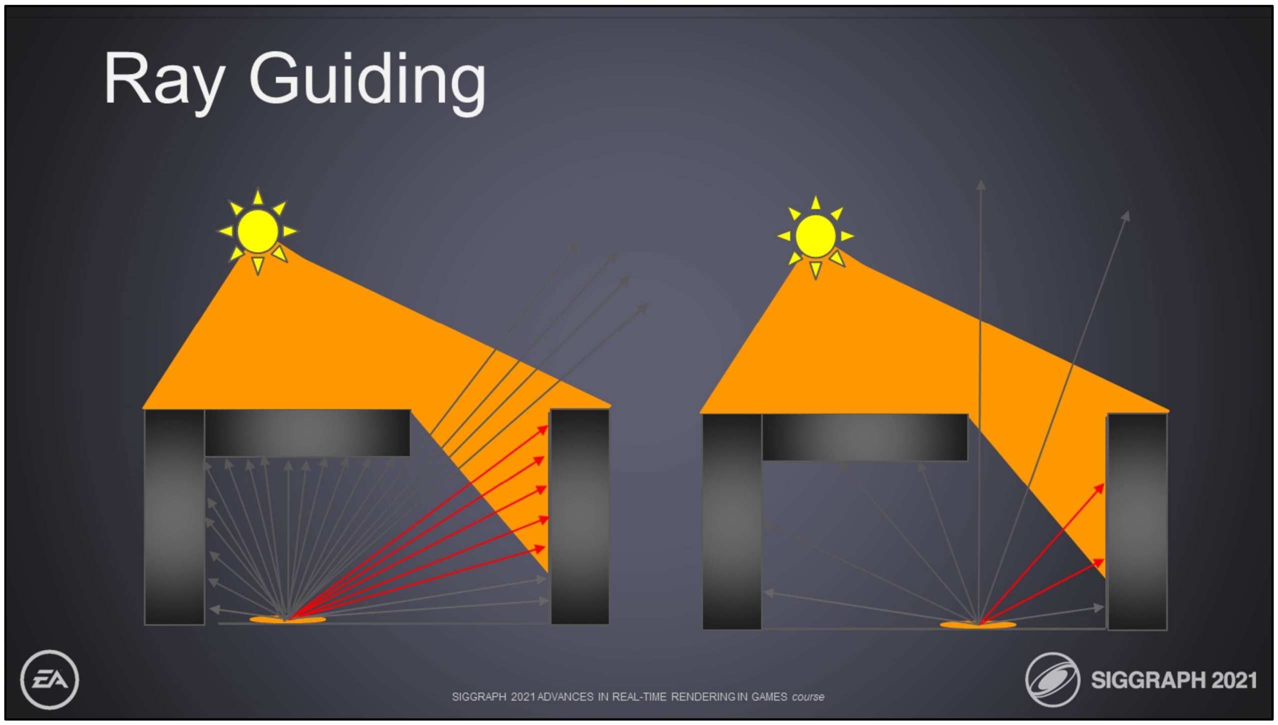
Ray Guiding



SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

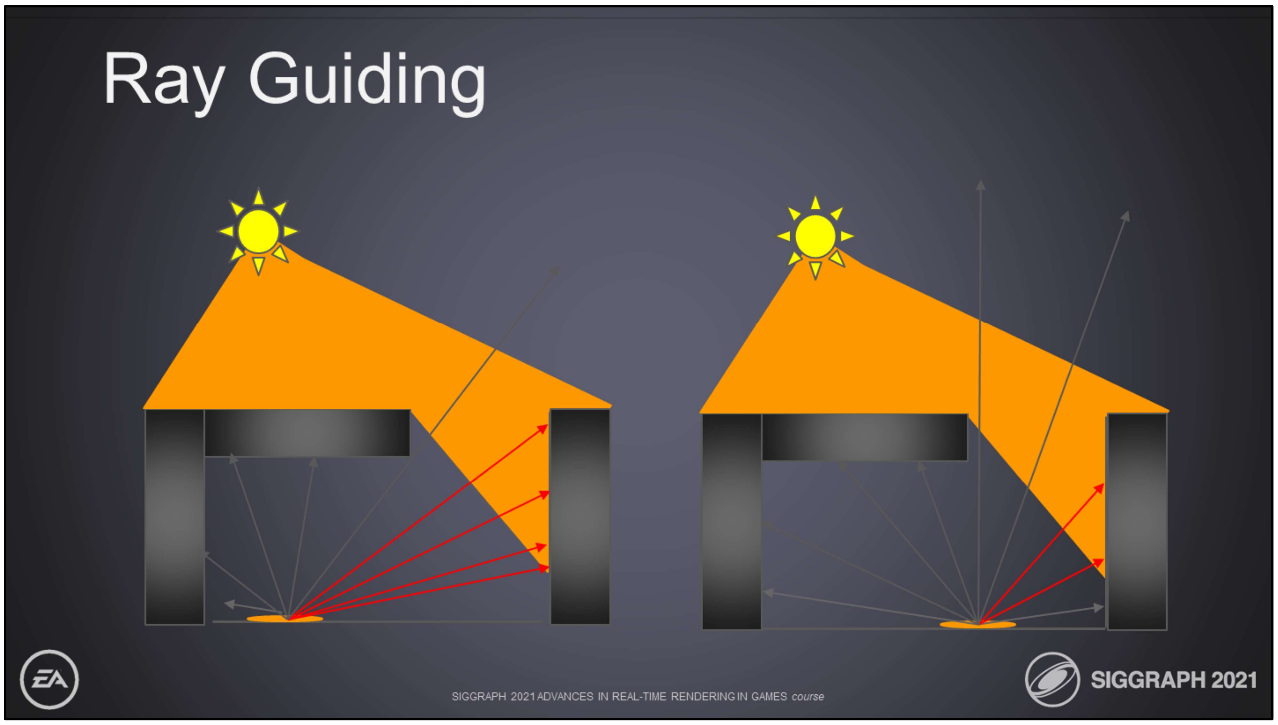
SIGGRAPH 2021

What's really important in this particular scene is to get a good estimate of exactly how much light is coming from the wall on the left. The rest of the room is very dark and does not really matter.

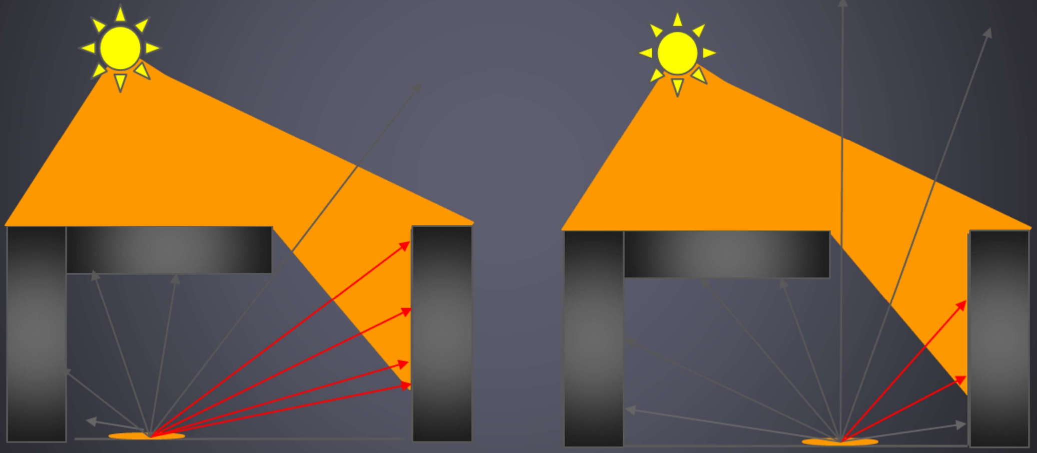


Ray Guiding

One way to get a better estimate is to send more rays. But that’s expensive, so we don’t want to do that.



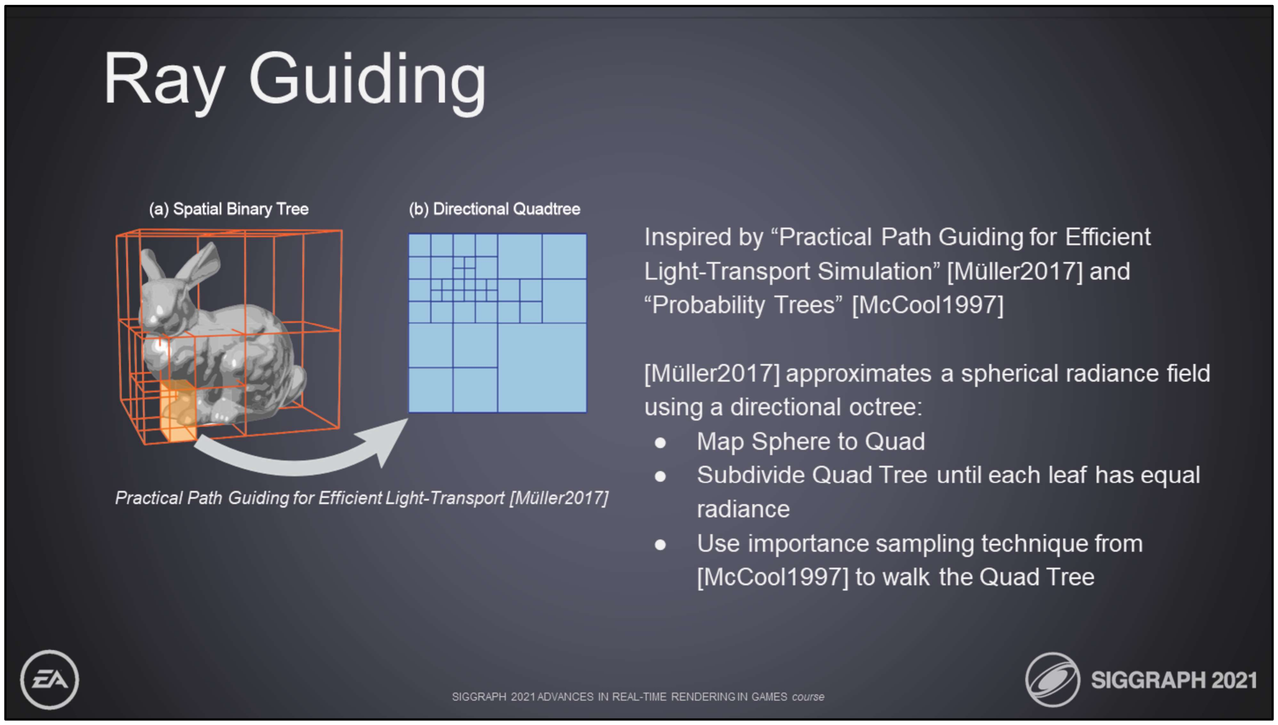
Ray Guiding



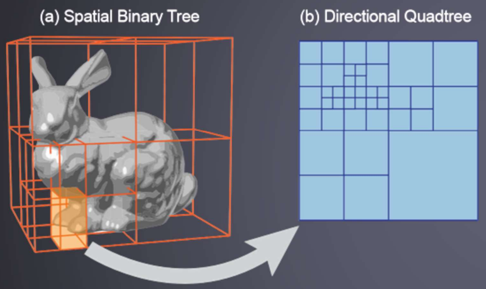
SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

But what if we could use the same number of rays, and just send more of them in directions that are important. That is what ray guiding does.



Ray Guiding



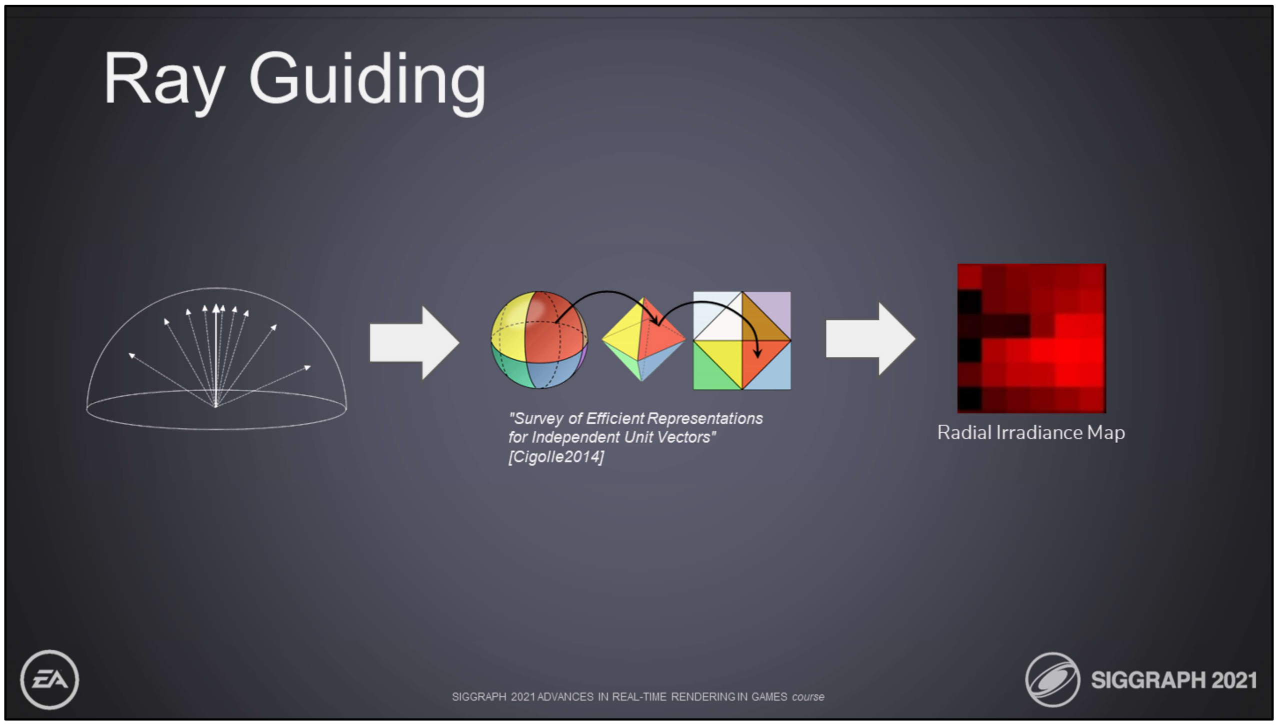
Practical Path Guiding for Efficient Light-Transport [Müller2017]

Inspired by “Practical Path Guiding for Efficient Light-Transport Simulation” [Müller2017] and “Probability Trees” [McCool1997]

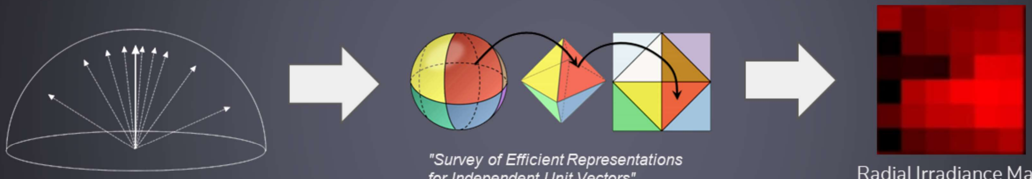
[Müller2017] approximates a spherical radiance field using a directional octree:

- Map Sphere to Quad
- Subdivide Quad Tree until each leaf has equal radiance
- Use importance sampling technique from [McCool1997] to walk the Quad Tree

We were inspired by <references>. Muller creates a quad tree of the sphere, where each leaf node is of equal radiance. This is then importance sampled by employing the technique of McCool, and generates rays in direction proportional to the radiance.



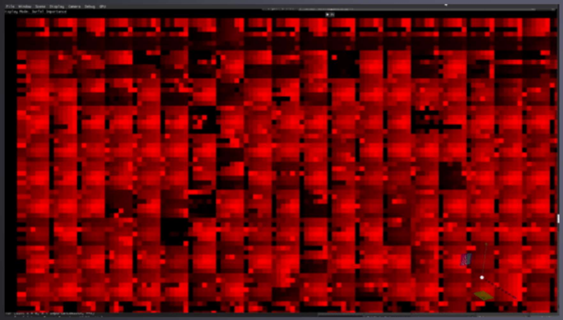
Ray Guiding



"Survey of Efficient Representations for Independent Unit Vectors" [Cigolle2014]

But we don't want to construct a quad tree every frame, for every surfel. We map the hemisphere to a quad where we track radiance.

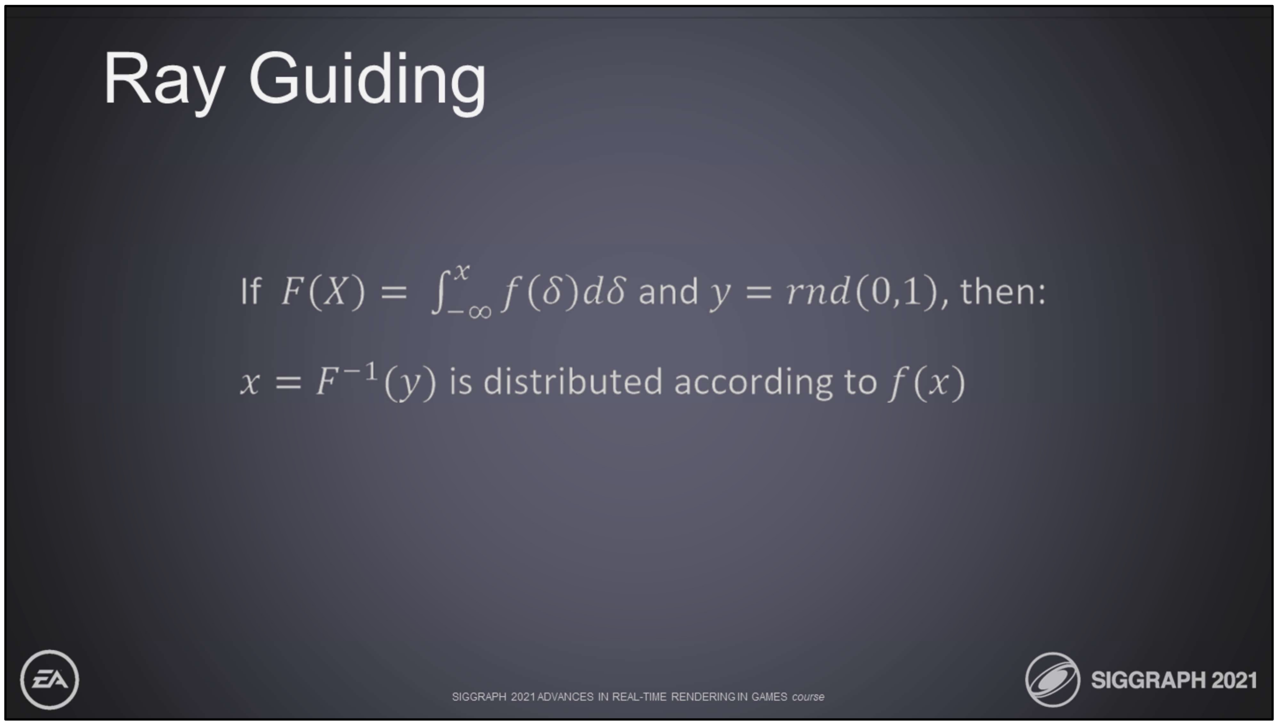
Ray Guiding



- Every surfel produces a moving average 6x6 luminance map over its hemisphere
- Stored in single 4K texture (could support 7x7 for all surfels)
- 8 bits per texel + Single 16 bit scaling per surfel
- Normalize function every frame

SIGGRAPH 2021

The size of the map is configurable, but for most applications we use 6x6 texels, with 8 bits per component, plus a single 16 bit scaling value. We normalize the entire function after each iteration, so we essentially track relative radiance, and maximize precision. As surfels integrate radiance we populate the guiding function. Once the function is somewhat populated we start using that to guide our rays.



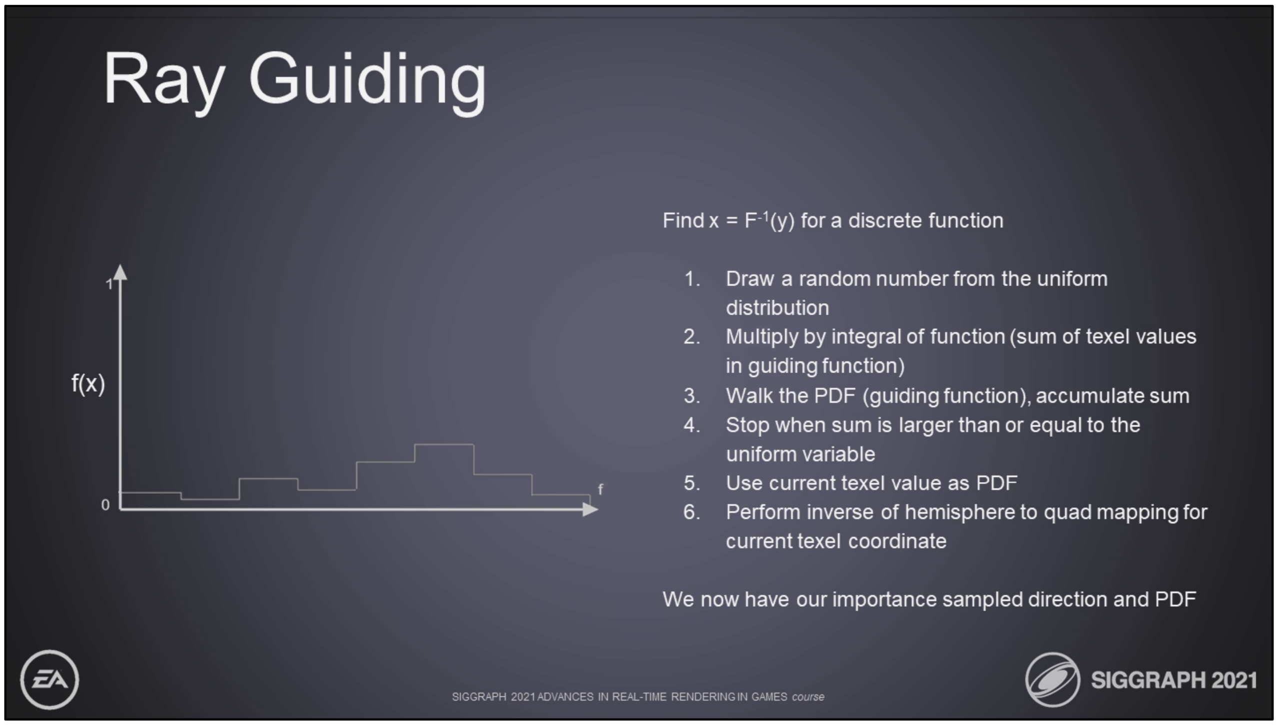
Ray Guiding

If $F(X) = \int_{-\infty}^{x} f(\delta)d\delta$ and $y = rnd(0,1)$, then:

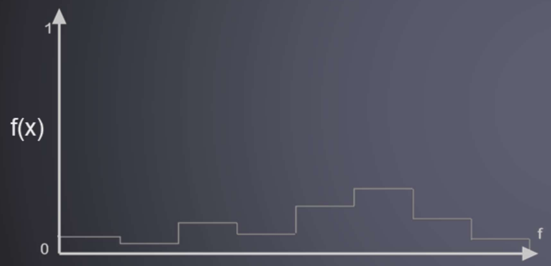
$x = F^{-1}(y)$ is distributed according to $f(x)$

SIGGRAPH 2021

Inverse of the Cumulative Distribution Function (CDF) can be used to generate samples according to the PDF.



Ray Guiding

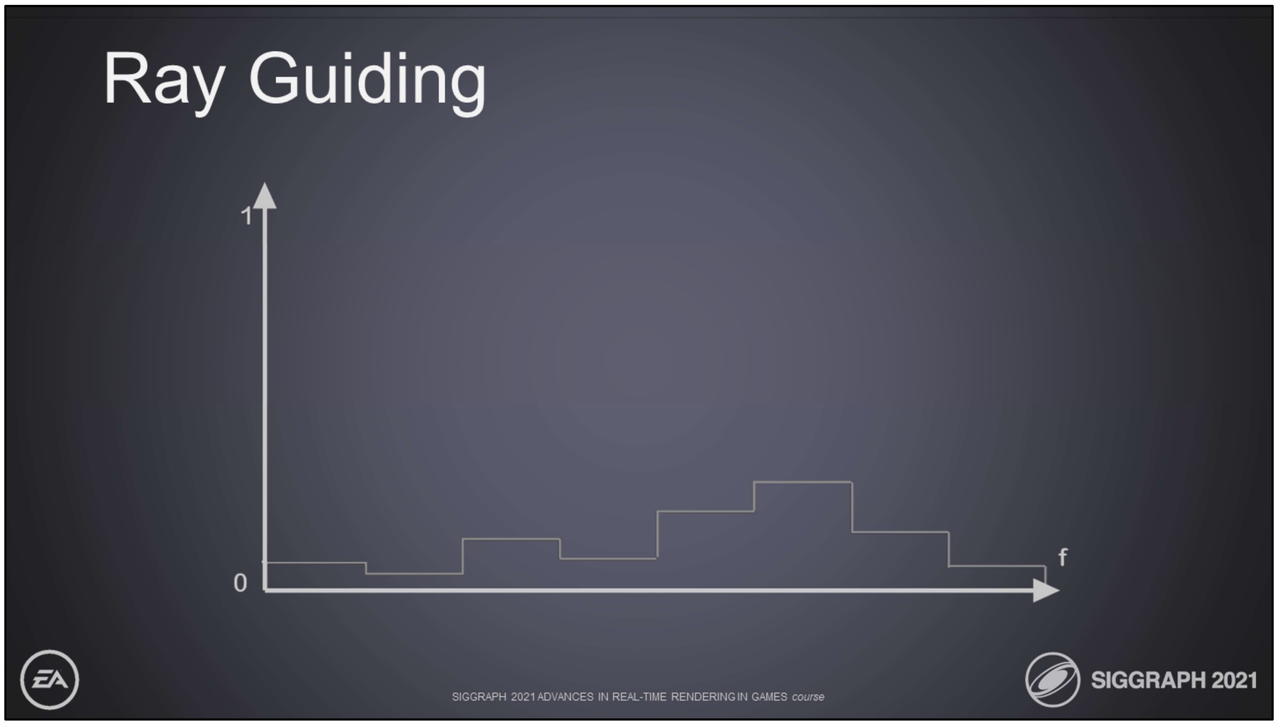


Find x = $F^{-1}(y)$ for a discrete function

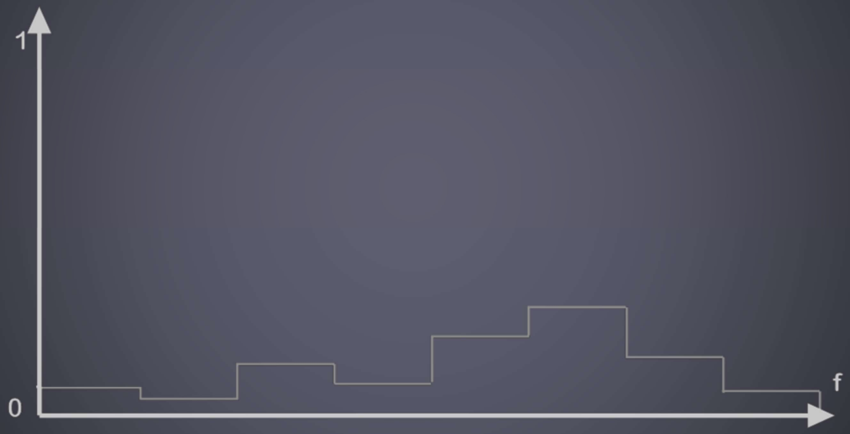
1. Draw a random number from the uniform distribution
2. Multiply by integral of function (sum of texel values in guiding function)
3. Walk the PDF (guiding function), accumulate sum
4. Stop when sum is larger than or equal to the uniform variable
5. Use current texel value as PDF
6. Perform inverse of hemisphere to quad mapping for current texel coordinate

We now have our importance sampled direction and PDF

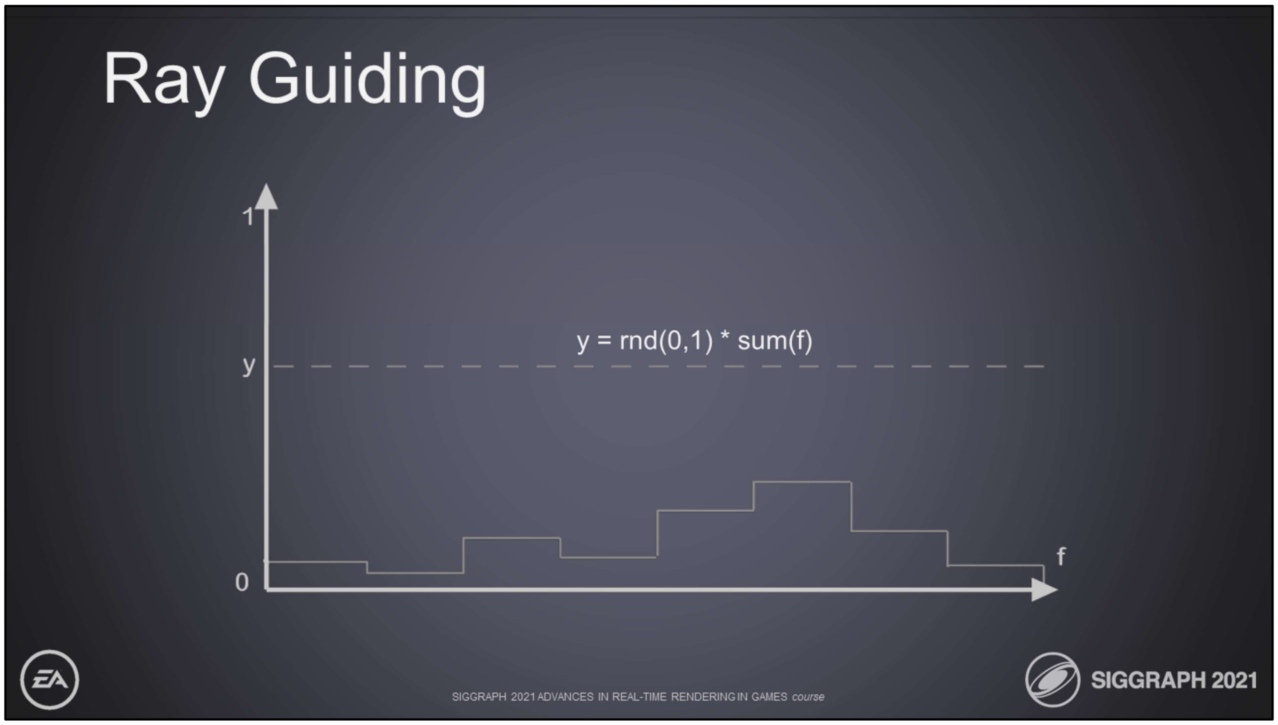
I will illustrate how we importance sample this, by first showing how this is done for a 1D discrete function.



Ray Guiding



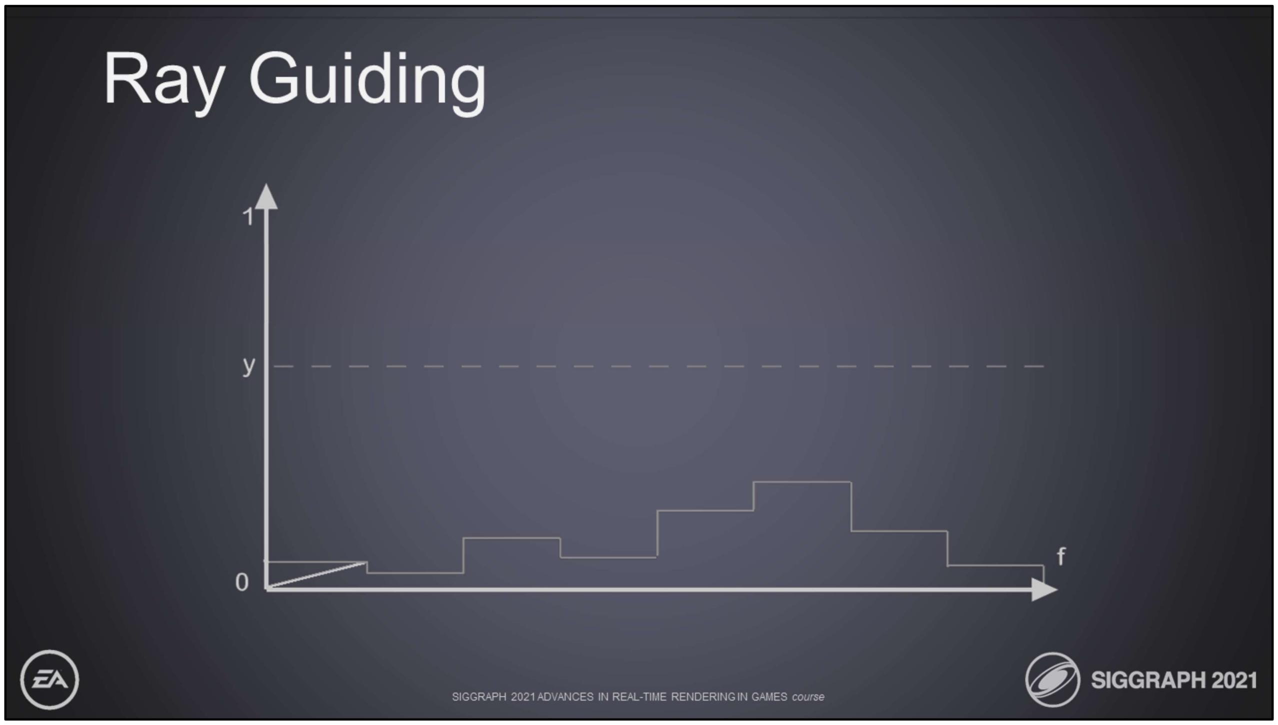
SIGGRAPH 2021



Ray Guiding

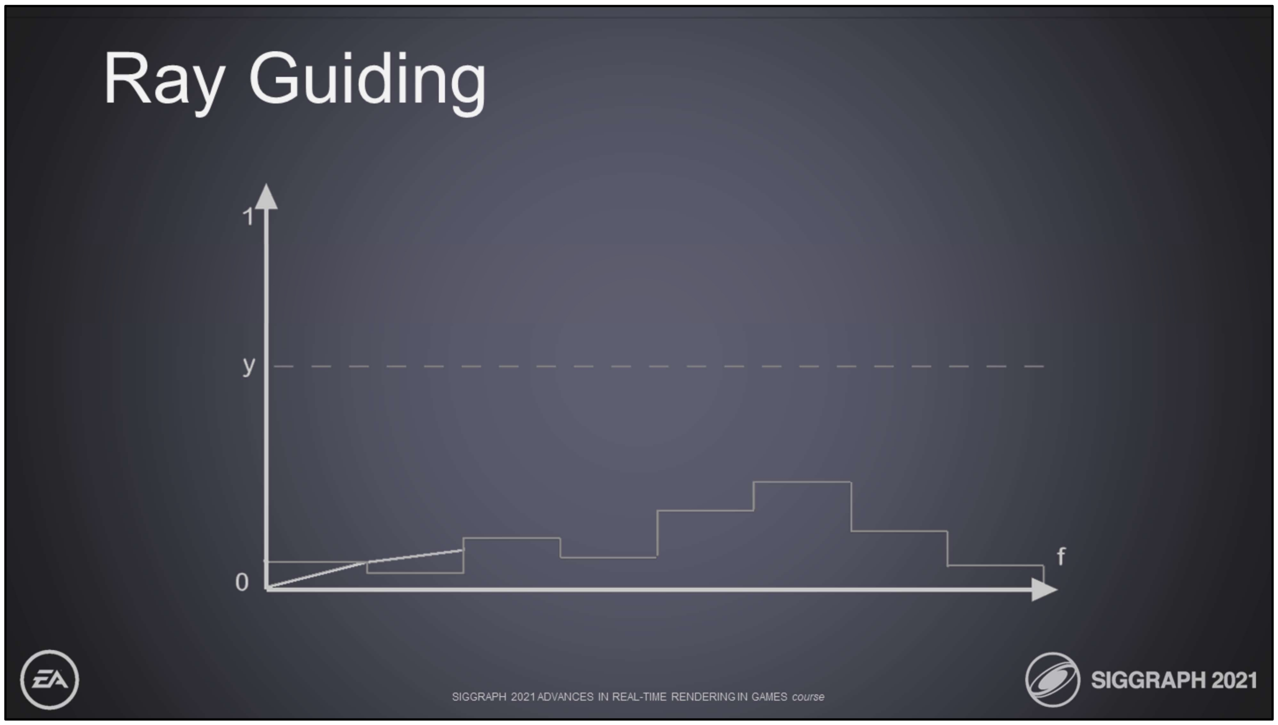
SIGGRAPH 2021

First we generate a random number from the uniform distribution, and multiply that by the sum of the discrete function.

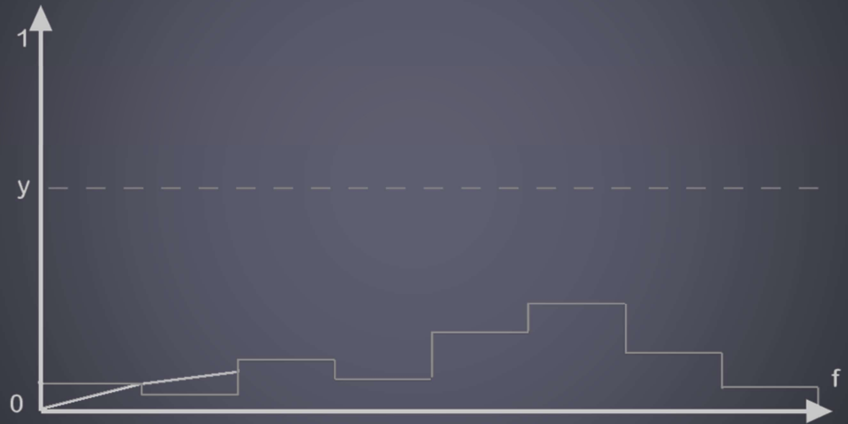


Ray Guiding

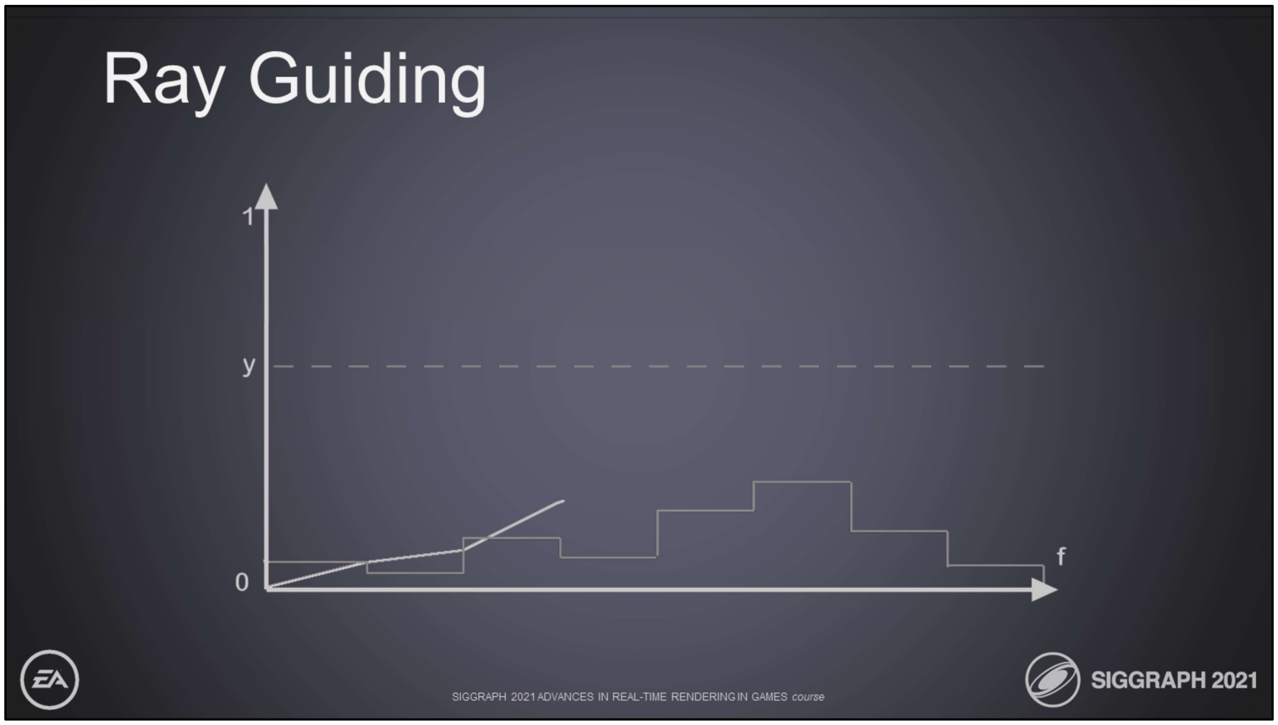
Then we start walking the function, and it doesn’t really matter which order you walk in, as long as you are consistent.



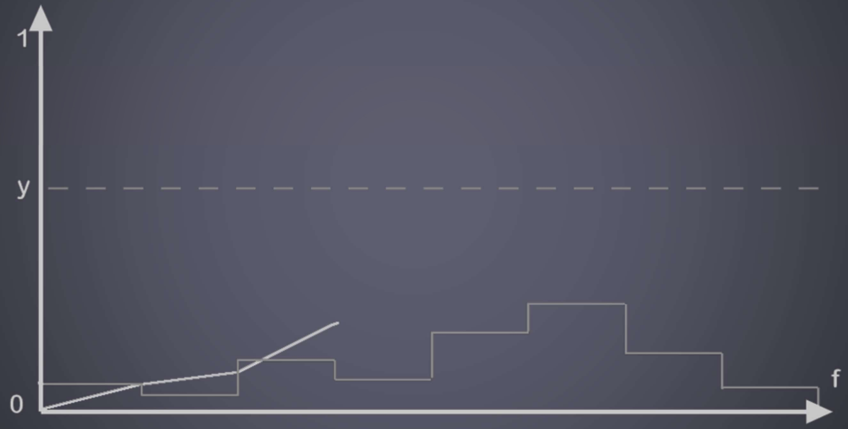
Ray Guiding



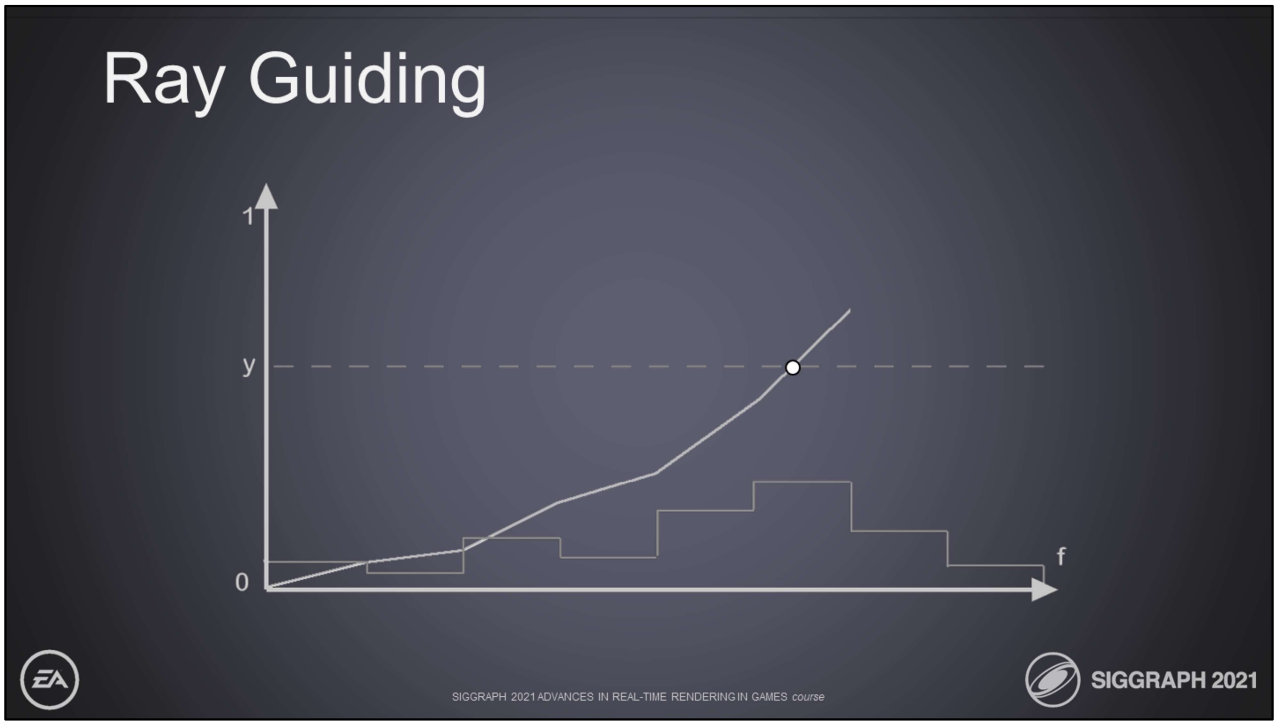
As we walk the function we sum the discrete parts.



Ray Guiding

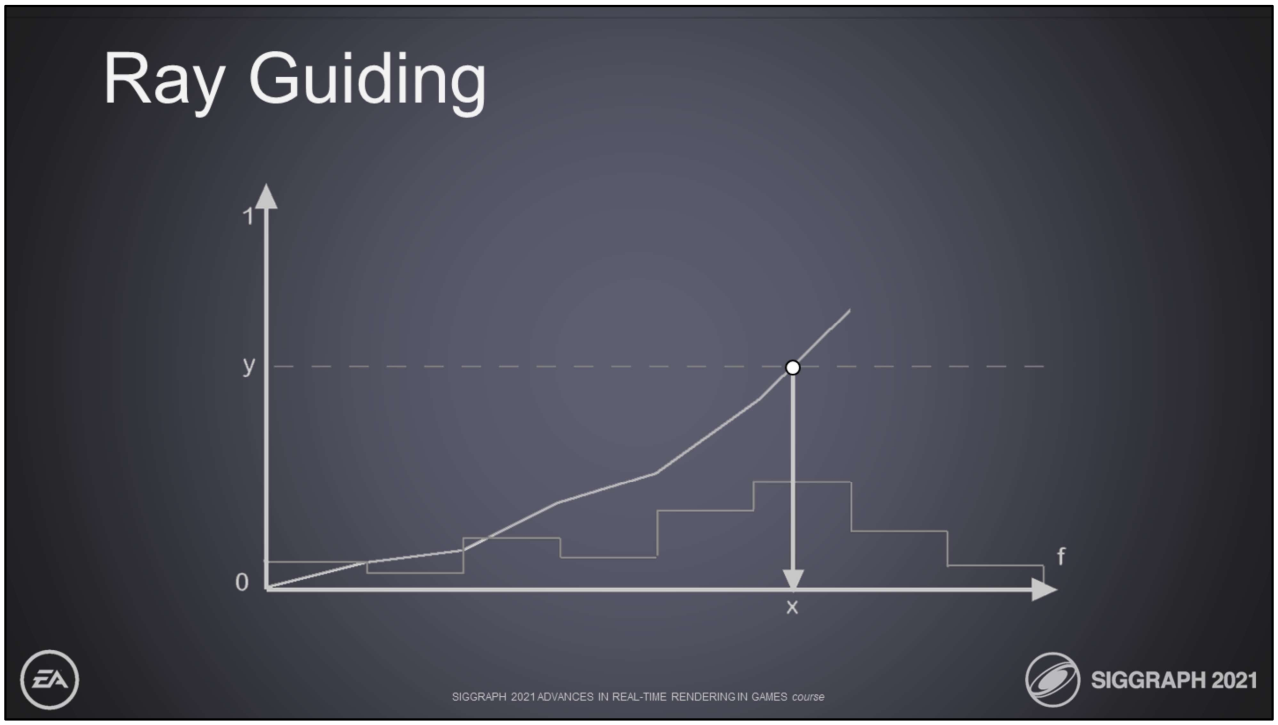


SIGGRAPH 2021

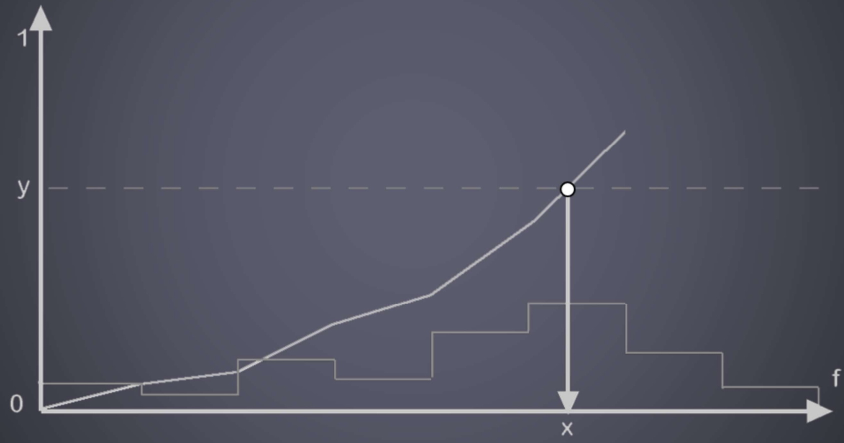


Ray Guiding

When the sum reaches the random uniform variable, we stop.



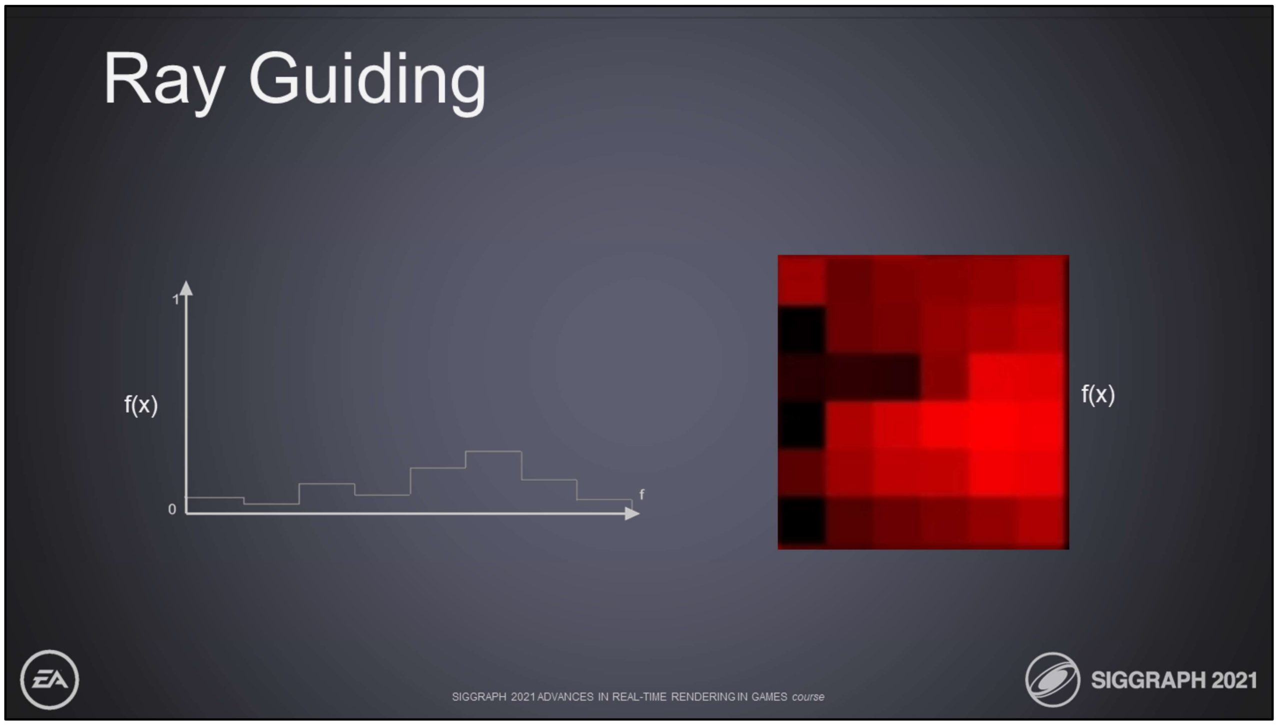
Ray Guiding



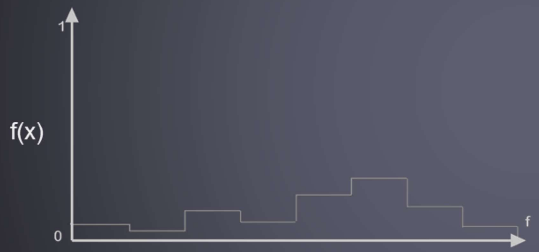
SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

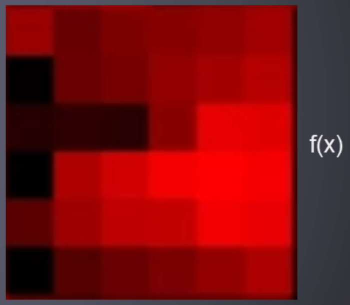
SIGGRAPH 2021

We now have our importance sampled variable. Each discrete part of the function will be picked proportionally to its value. And we also have it’s Probability Density function, which is the value of the function at that position.



Ray Guiding

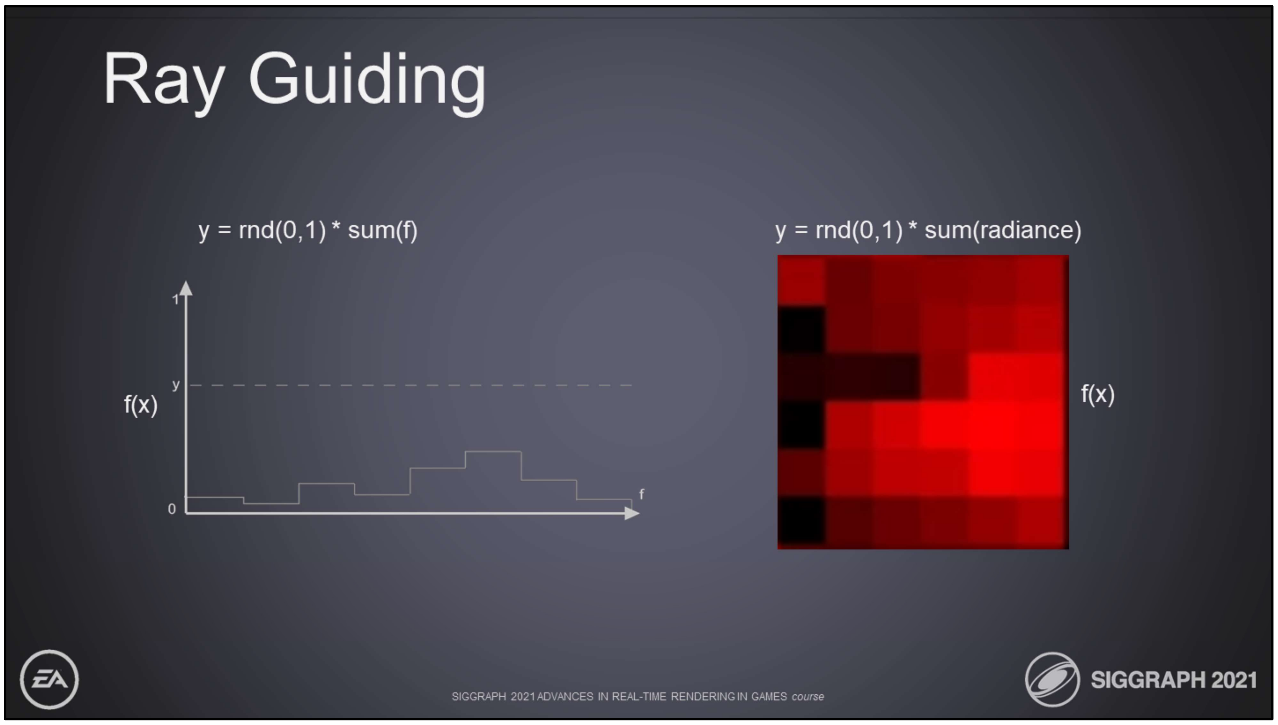




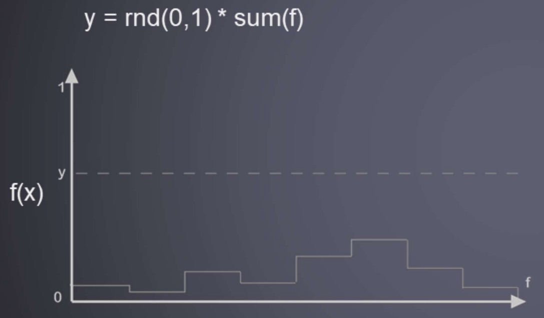
SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

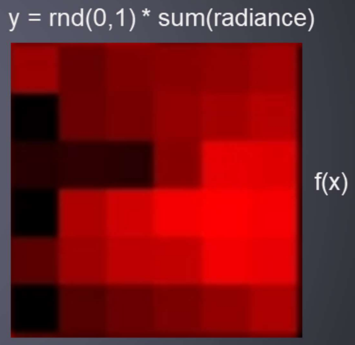
SIGGRAPH 2021

While our hemispherical radiance map is two dimensional, we can do the same thing there.

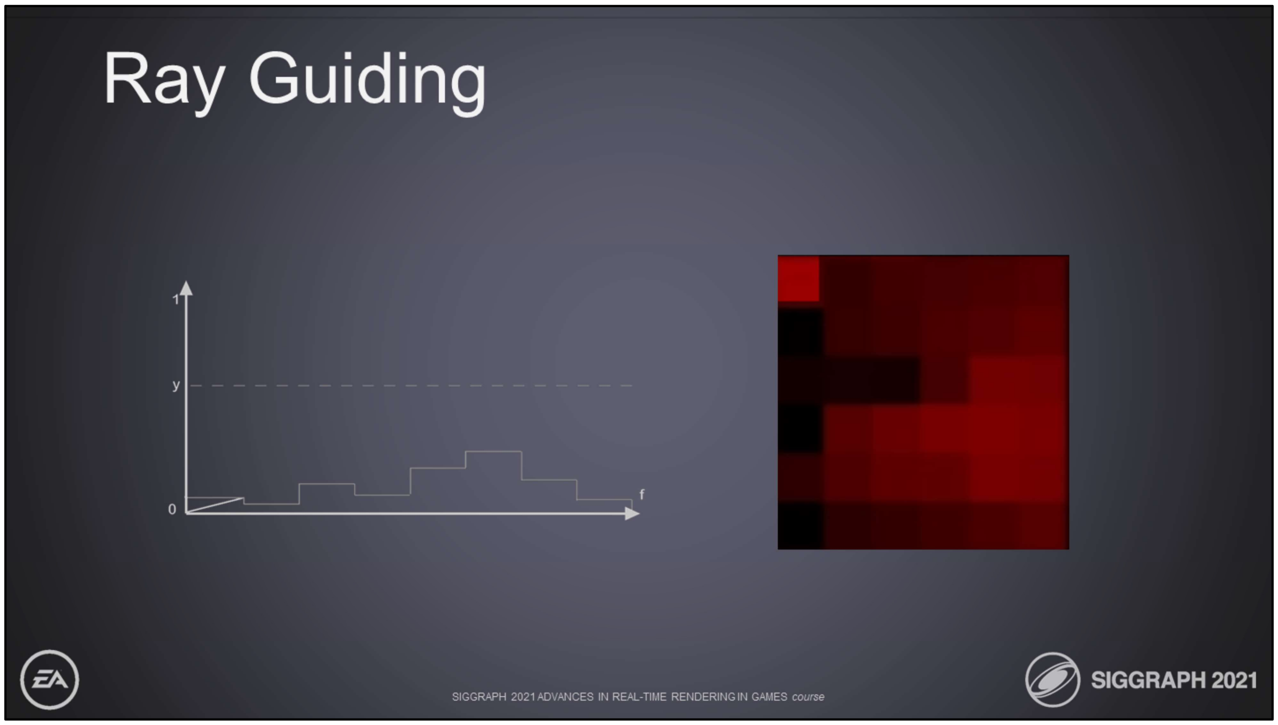


Ray Guiding

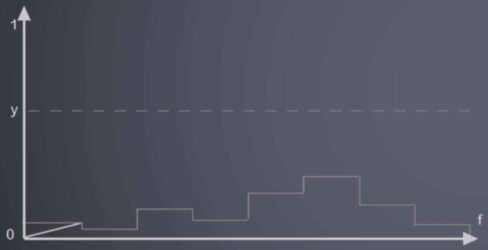


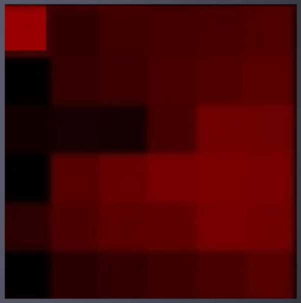


We multiply the random distribution with the sum of radiance.



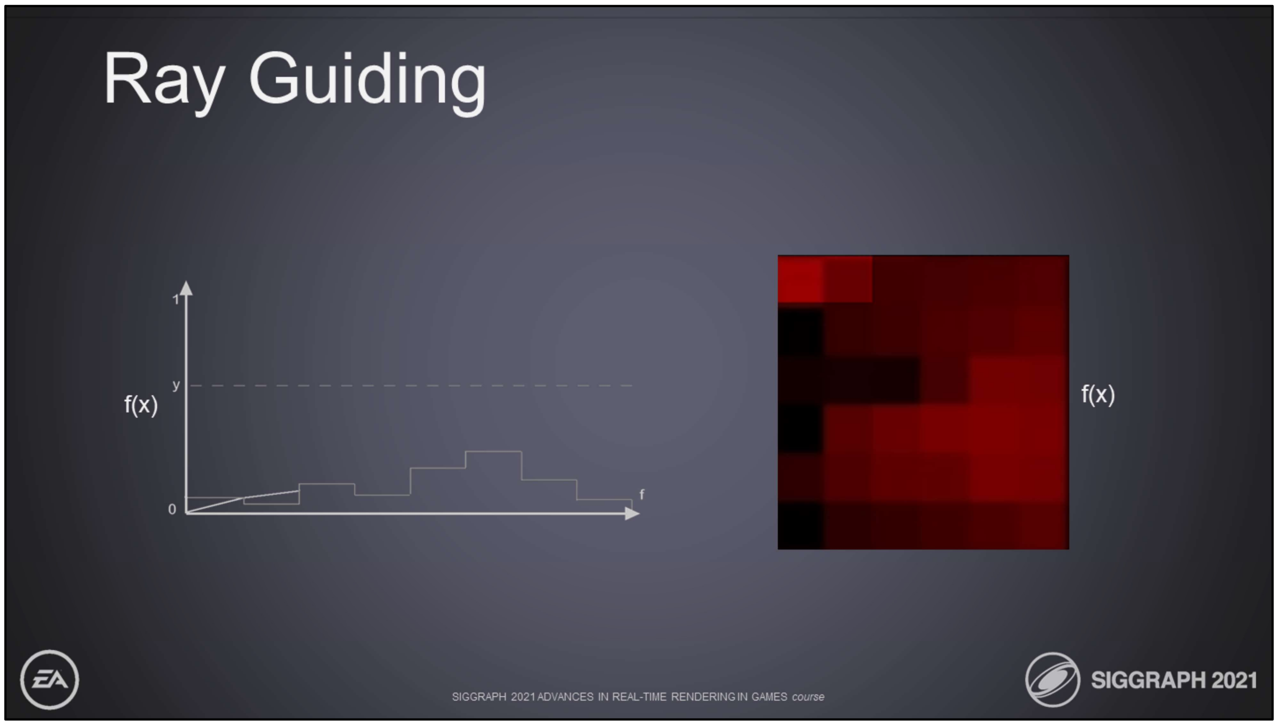
Ray Guiding



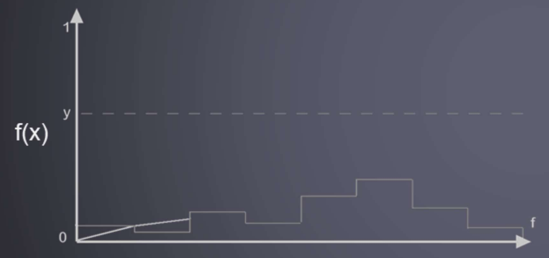


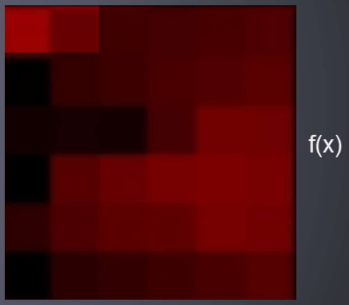
SIGGRAPH 2021

Then we walk the function and accumulate.

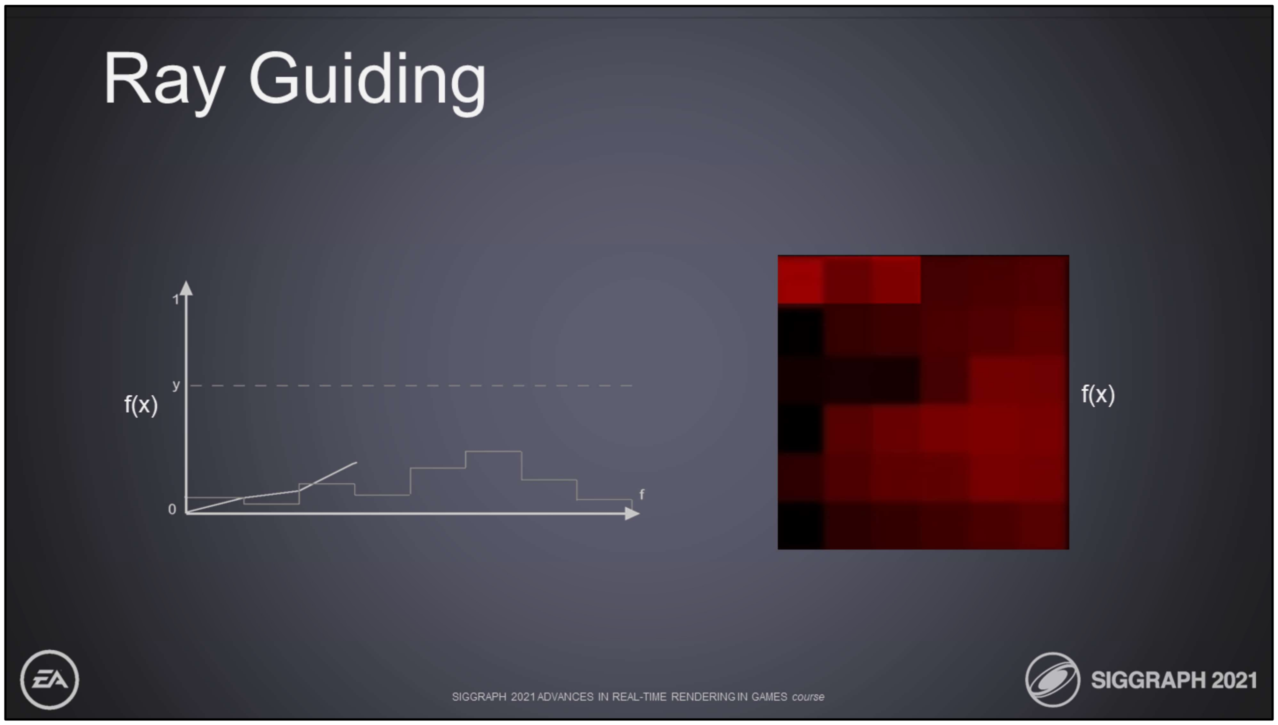


Ray Guiding

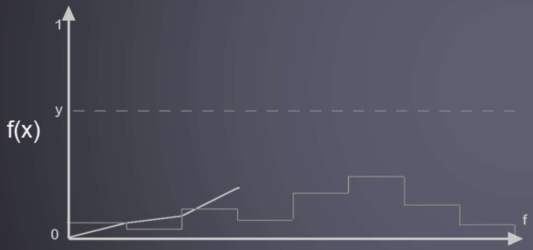


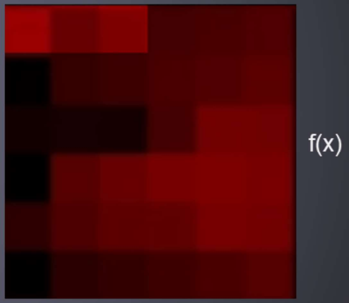


In practice we don’t use a quad tree, we use a fixed 6x6 texel function. We find higher resolutions give better results, but we want to keep memory in check, and not make sampling the function too expensive. We currently just walk the entire function when we select ray directions, but a 6x6 function does allow for a hierarchical traversal similar to the octree case, where we can use a linear sampler and effectively walk a higher 3x3 mip level through the first iteration.

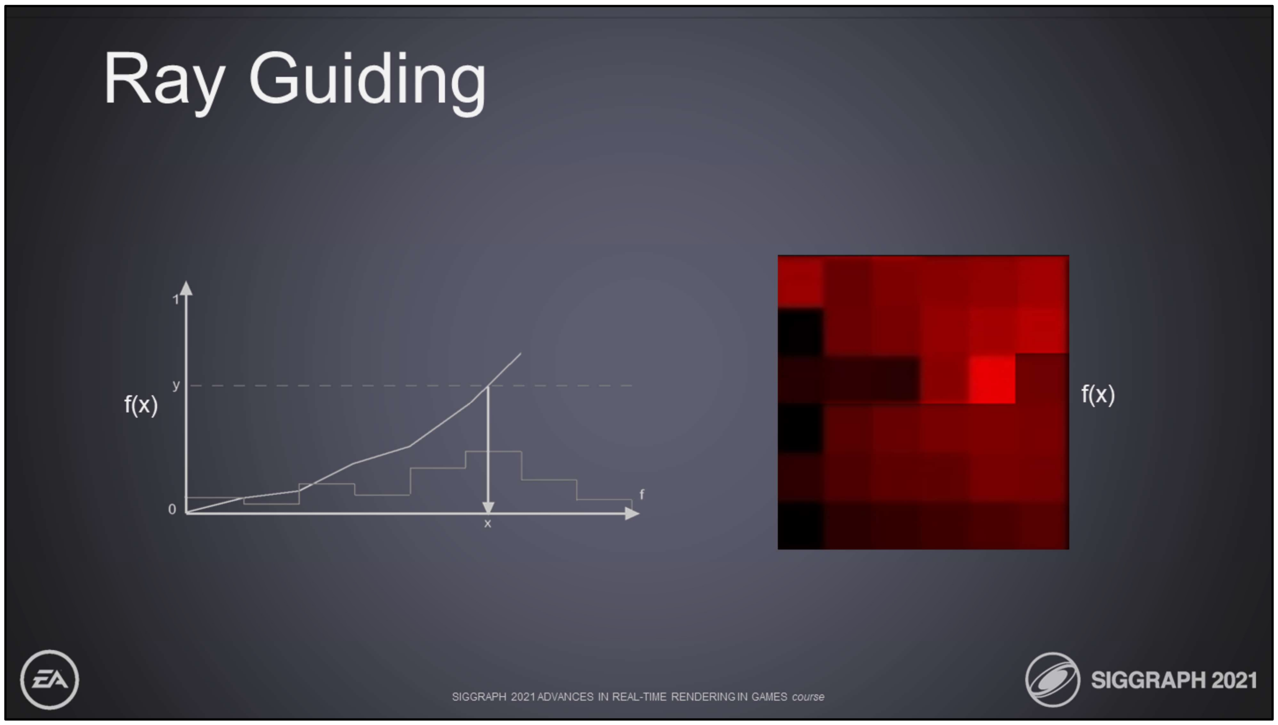


Ray Guiding

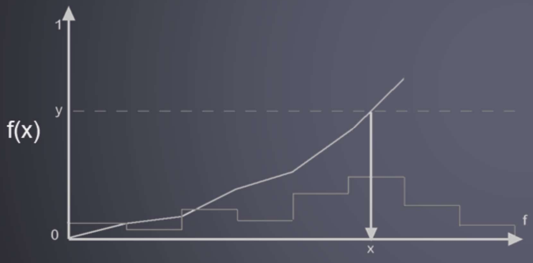


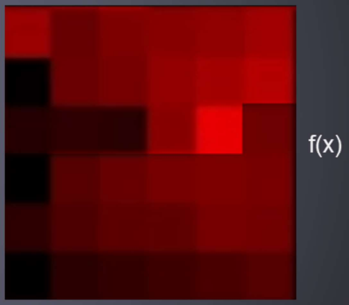


SIGGRAPH 2021

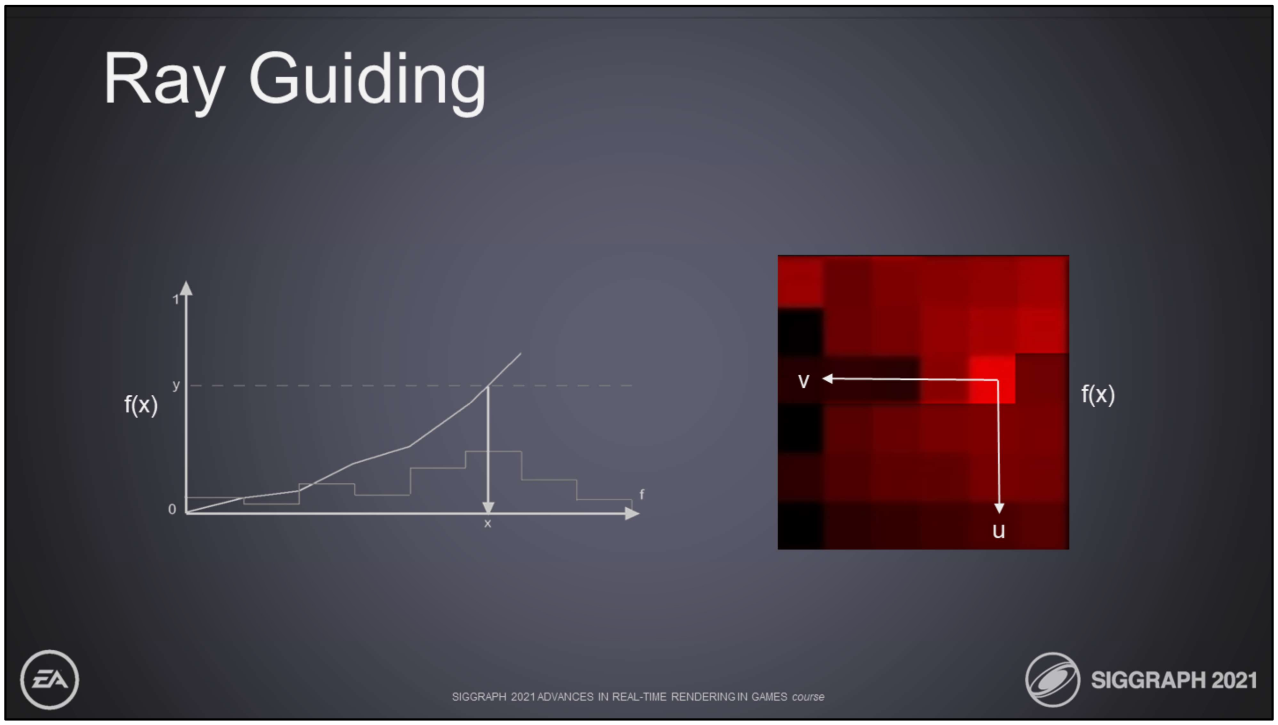


Ray Guiding

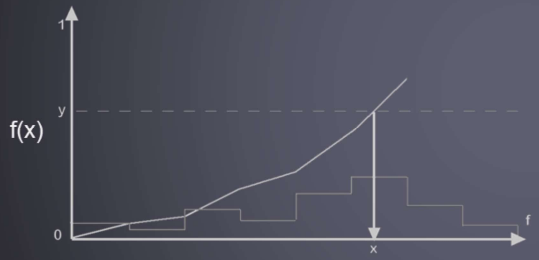


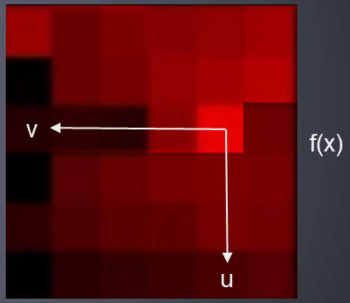


When the sum reaches the random variable, we stop.

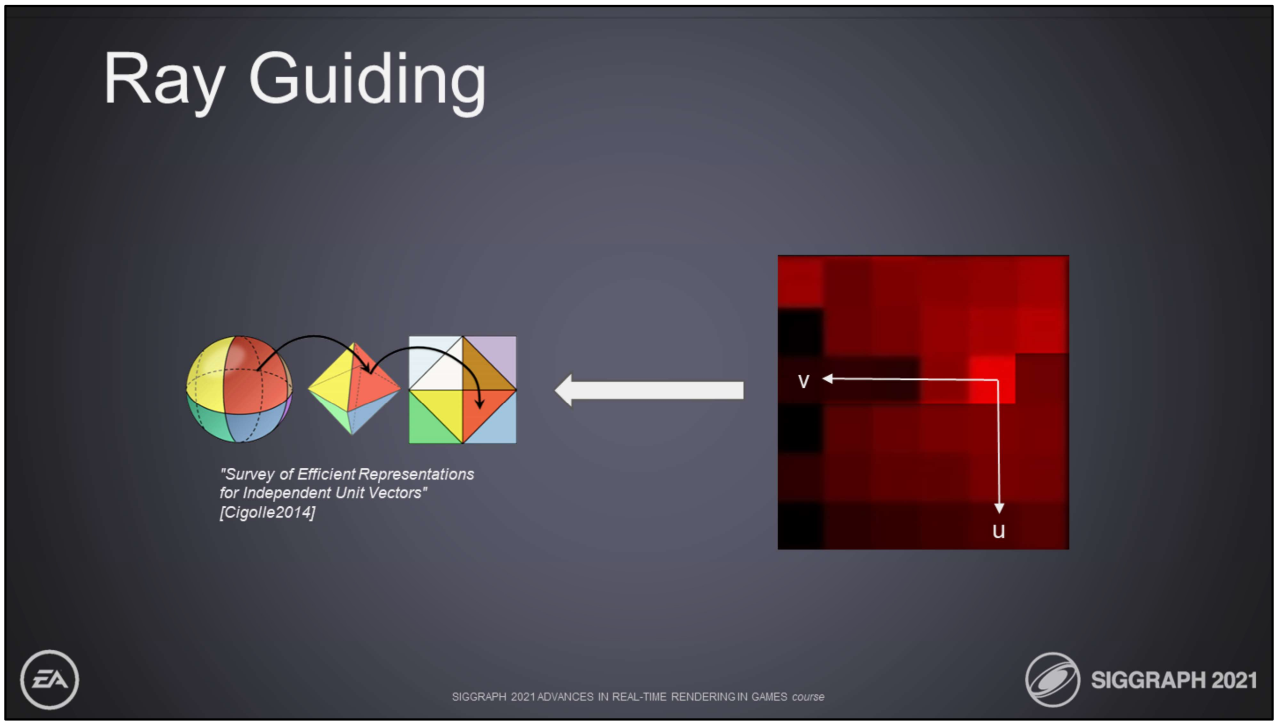


Ray Guiding

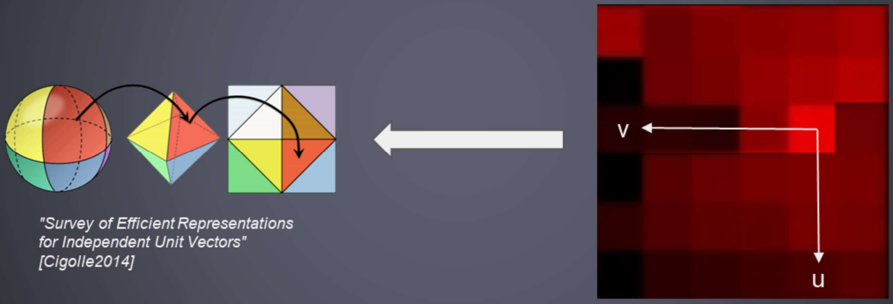




And we have our importance sampled variable, in UV-space.

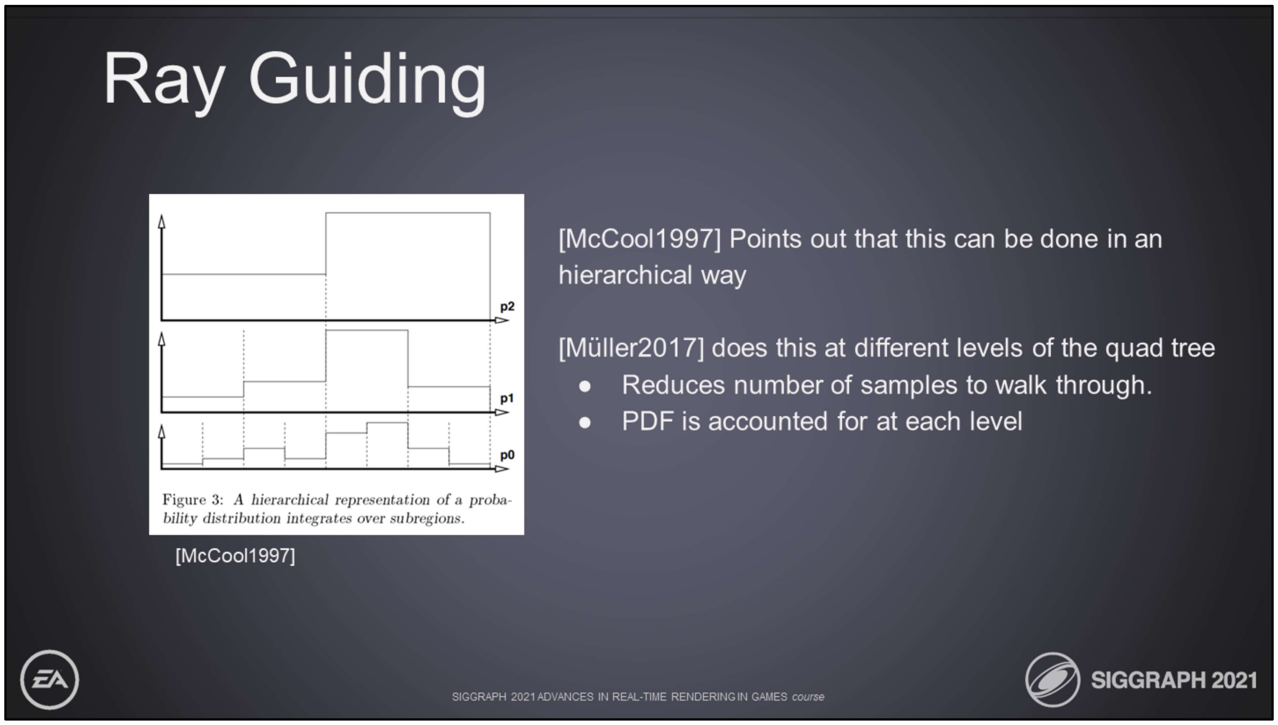


Ray Guiding



"Survey of Efficient Representations for Independent Unit Vectors" [Cigolle2014]

We map this UV variable back to the hemisphere, and we now have an importance sampled ray direction, and its PDF.



Ray Guiding

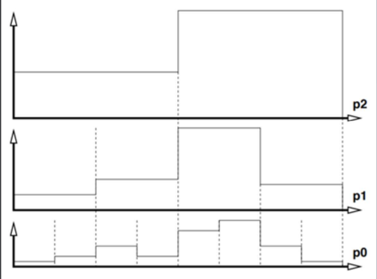


Figure 3: *A hierarchical representation of a probability distribution integrates over subregions.*

[McCool1997]

[McCool1997] Points out that this can be done in an hierarchical way

[Müller2017] does this at different levels of the quad tree

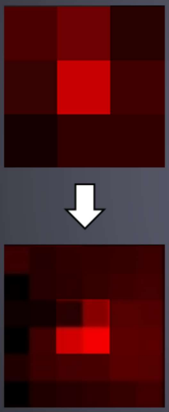
- Reduces number of samples to walk through.
- PDF is accounted for at each level

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

In practice we don’t use a quad tree, we use a fixed 6x6 texel function. We find higher resolutions give better results, but we want to keep memory in check, and not make sampling the function too expensive. We currently just walk the entire function when we select ray directions, but a 6x6 function does allow for a hierarchical traversal similar to the octree case, where we can use a linear sampler and effectively walk a higher 3x3 mip level through the first iteration.

Ray Guiding



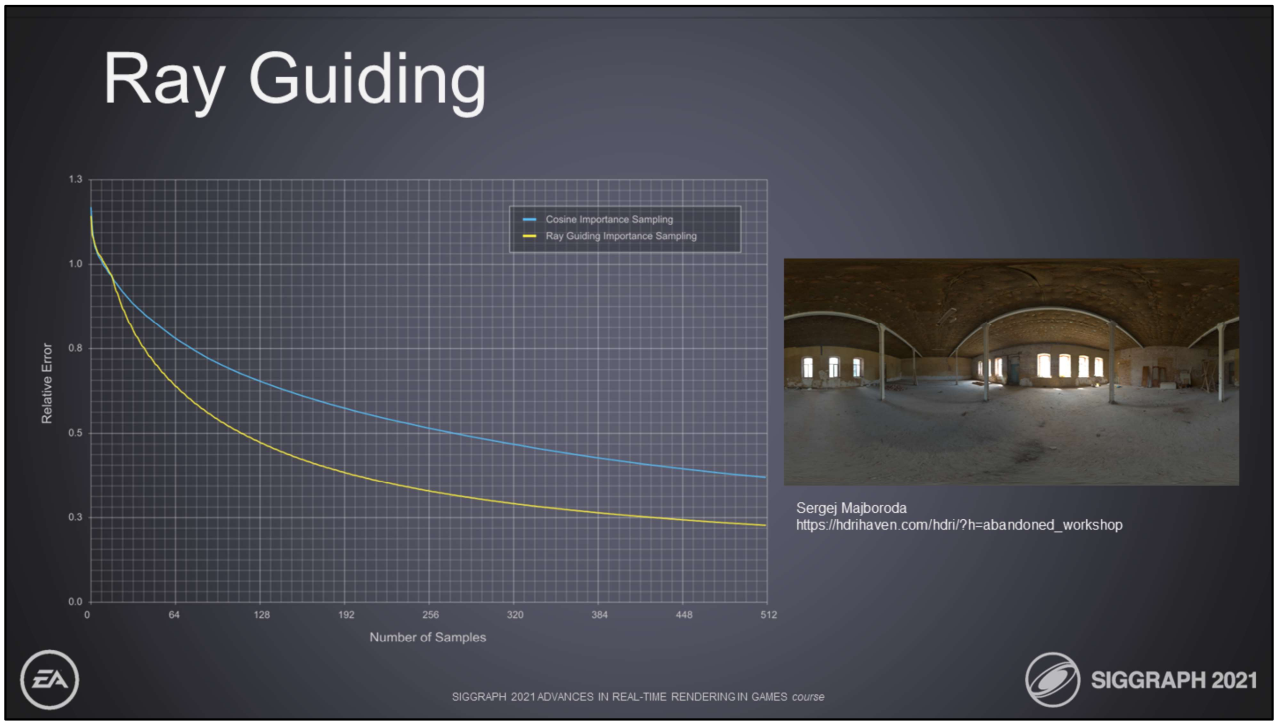
[McCool1997] Points out that this can be done in an hierarchical way

[Müller2017] does this at different levels of the quad tree

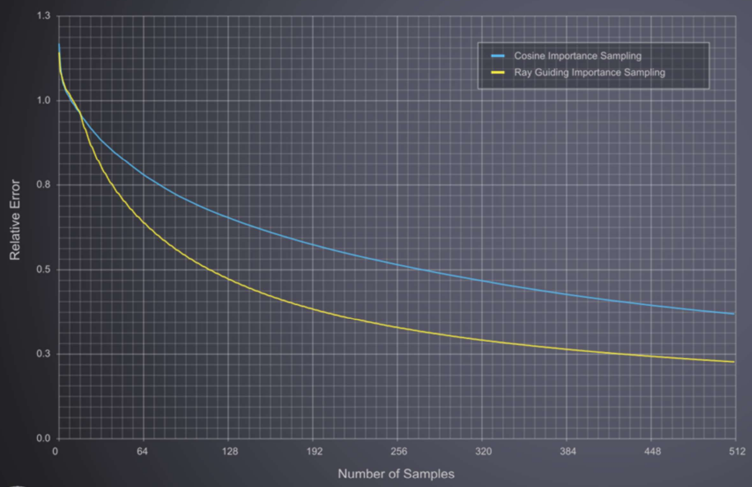
- Reduces number of samples to walk through.
- PDF is accounted for at each level

We could do this as well

- Use linear sample at texel connections



Ray Guiding



Sergej Majboroda
https://hdrihaven.com/hdri/?h=abandoned_workshop

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

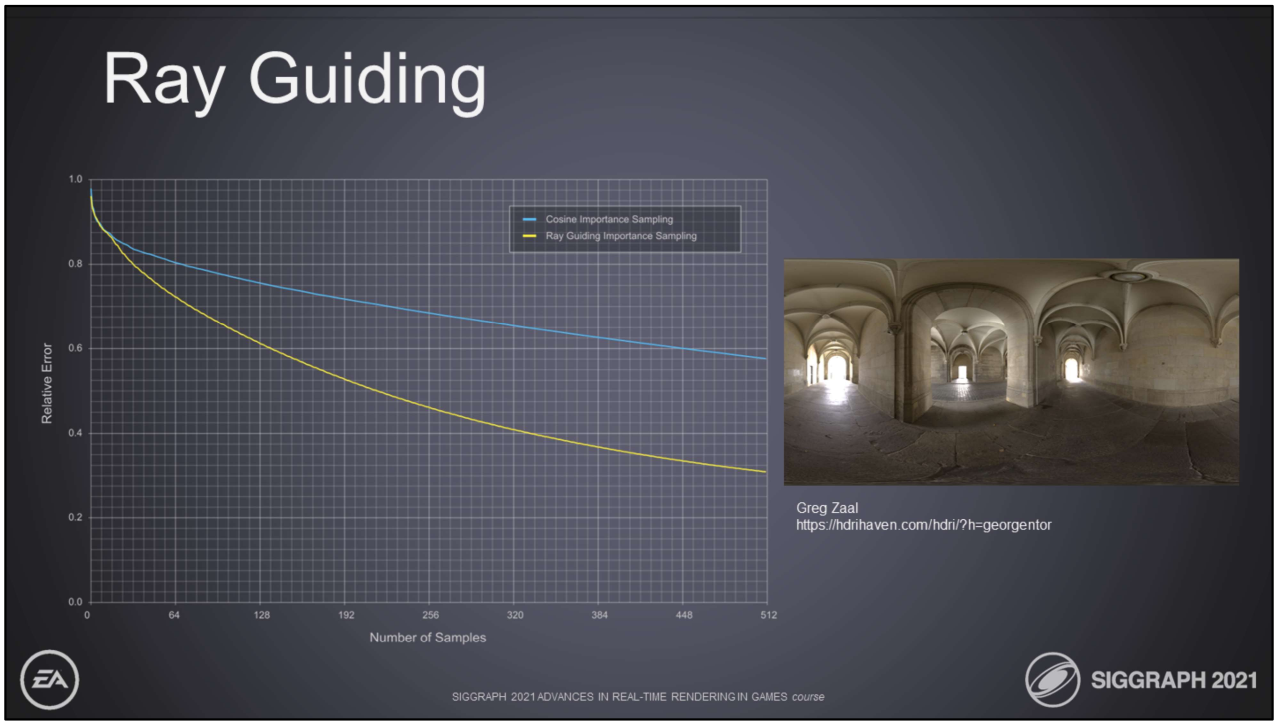
To verify that this works, where it works well and where it doesn't, we created a test bed where we can integrate across irradiance probes using the different sampling techniques.

I will now show some examples. In this graph we have the number of rays on the horizontal axis and the relative error on the vertical axis.

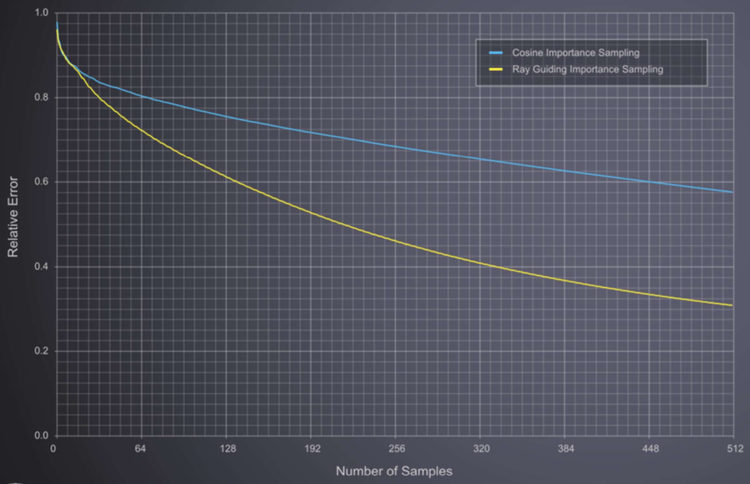
The blue curve shows the error when using cosine lobe importance sampling and the yellow one shows the error when also using our ray guiding texture.

The ray guiding works best when the guiding texture is a reasonable approximation of the incoming irradiance, and much of it is coming from a direction which would have been sampled rarely by the cosine lobe.

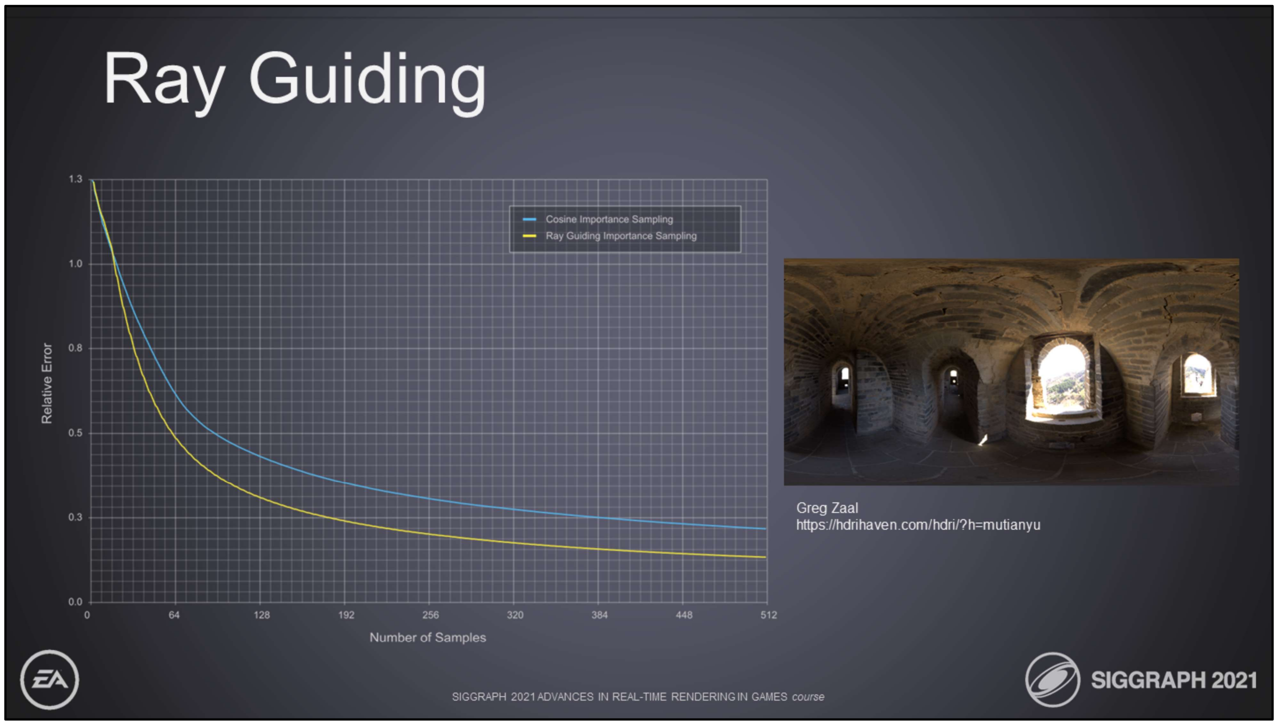
An example of a scenario like this is shown in the current graph. We have light coming in through big windows covering the entire horizon.



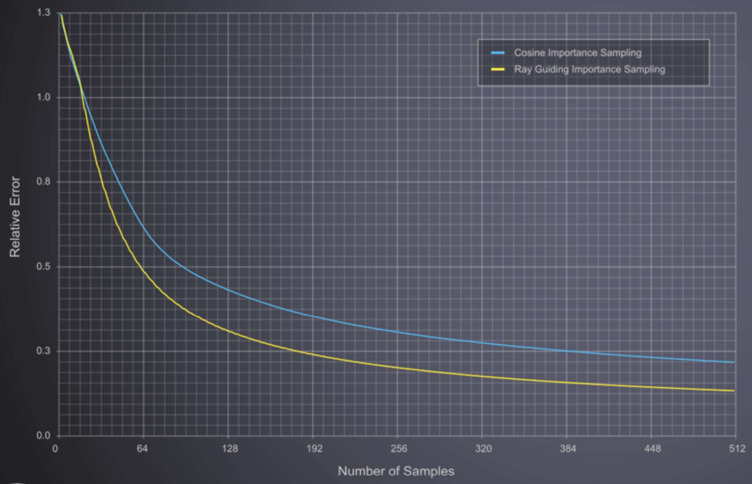
Ray Guiding



Greg Zaal
https://hdrihaven.com/hdri/?h=georgentor

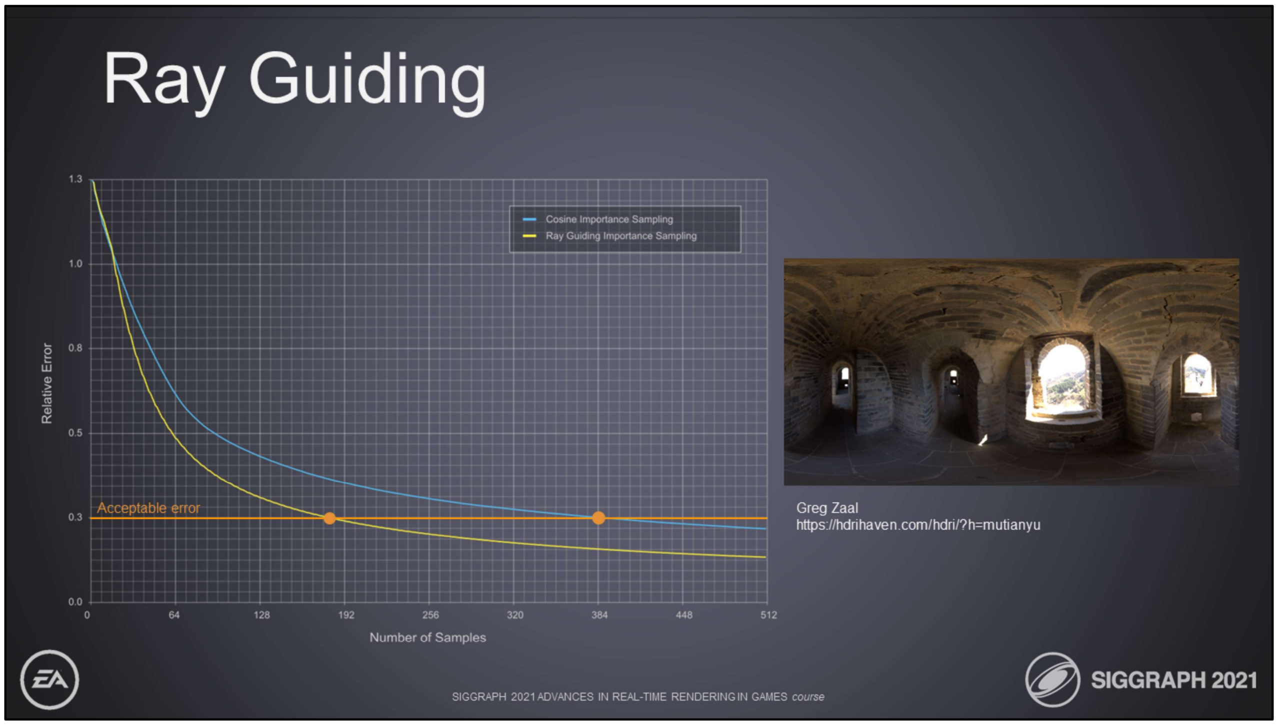


Ray Guiding

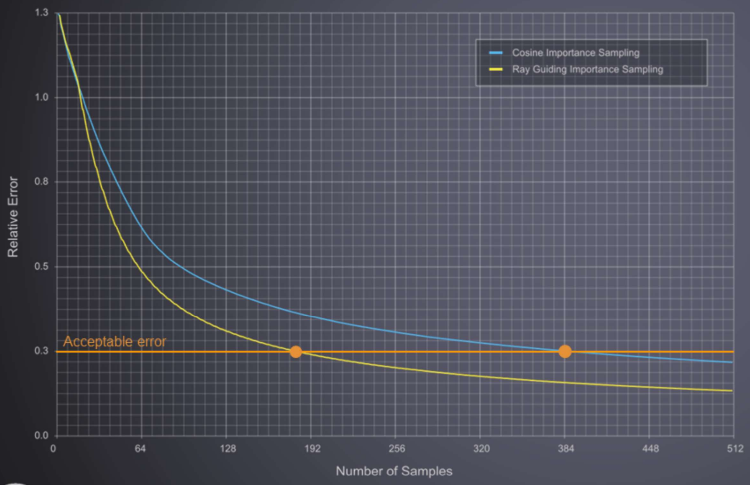


Greg Zaal
https://hdrihaven.com/hdri/?h=mutianyu

SIGGRAPH 2021



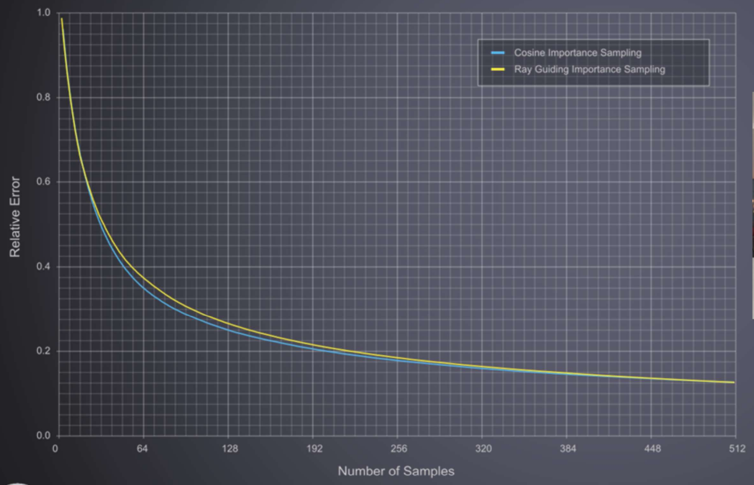
Ray Guiding



Greg Zaal
https://hdrihaven.com/hdri/?h=mutianyu

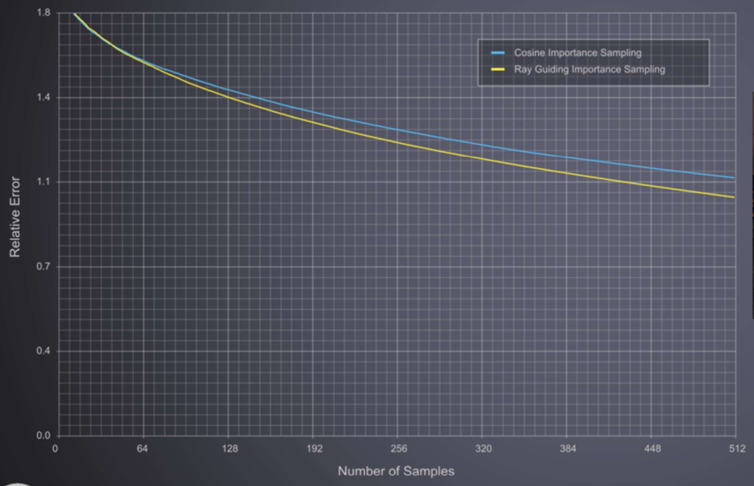
SIGGRAPH 2021

Ray Guiding

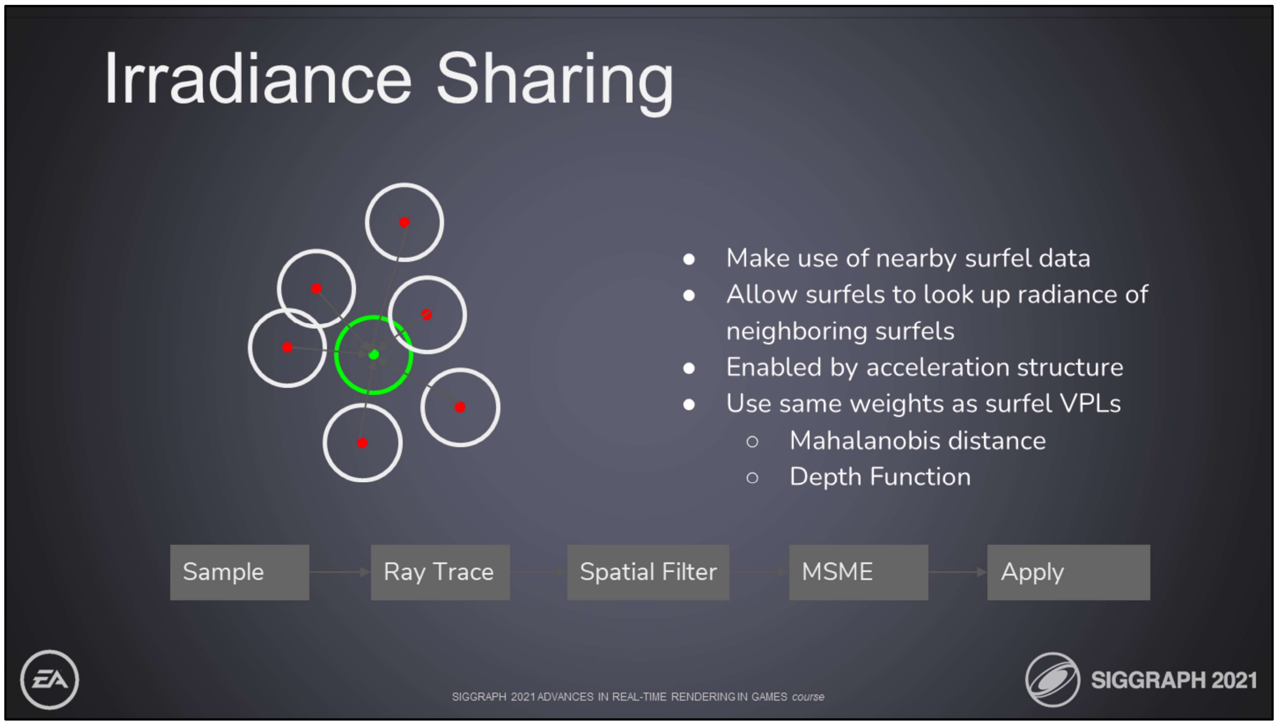


Sergej Majboroda
https://hdrihaven.com/hdri/?h=christmas_photo_studio_04

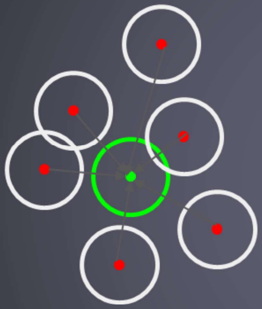
Ray Guiding



Sergej Majboroda
https://hdrihaven.com/hdri/?h=colorful_studio



Irradiance Sharing



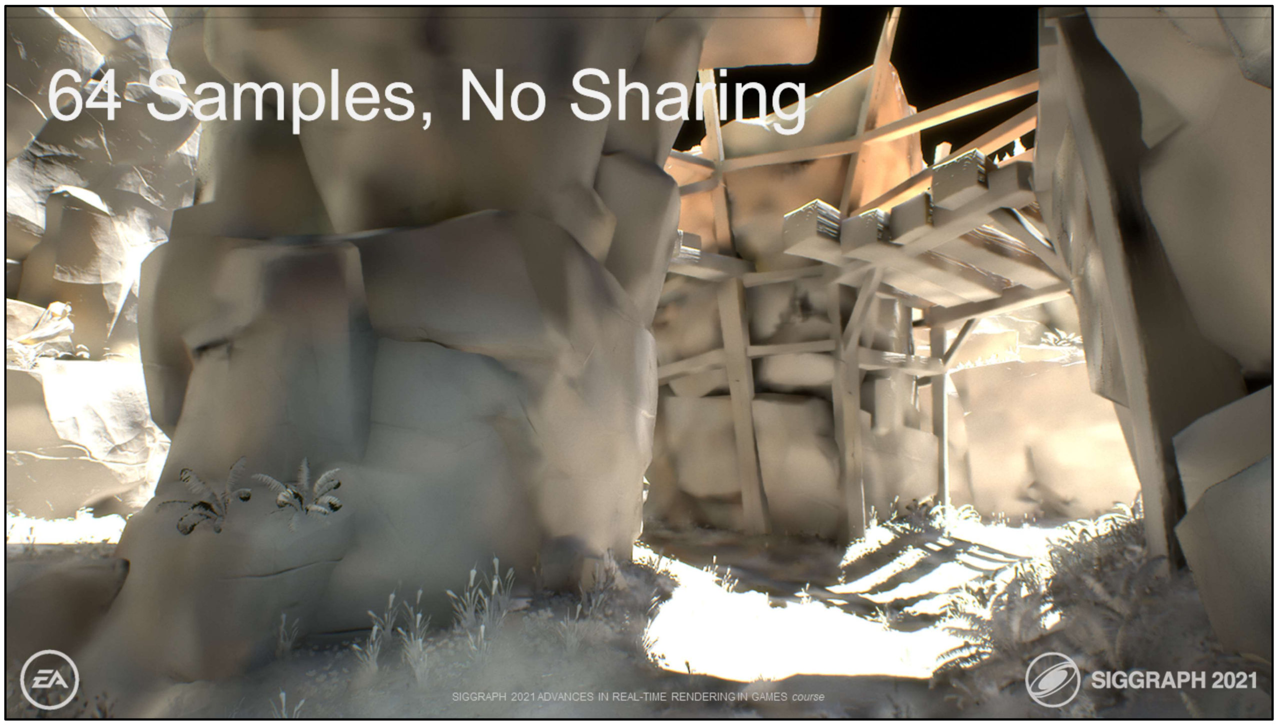
- Make use of nearby surfel data
- Allow surfels to look up radiance of neighboring surfels
- Enabled by acceleration structure
- Use same weights as surfel VPLs
  - Mahalanobis distance
  - Depth Function

$$\begin{tikzcd} \text{Sample} \arrow[r] & \text{Ray Trace} \arrow[r] & \text{Spatial Filter} \arrow[r] & \text{MSME} \arrow[r] & \text{Apply} \end{tikzcd}$$

In addition to the ray tracing optimizations we have just described, we also reduce the perceived noise by sharing irradiance between surfels. Since surfels are independent, it’s less straightforward to do this than say, with a regular grid, or a voxelized discretization. Luckily our acceleration structure contains all the surfels in the same region. This allows us to share some information between neighboring surfels. We utilize this to share irradiance when variance is high, which significantly reduces the perceived noise.

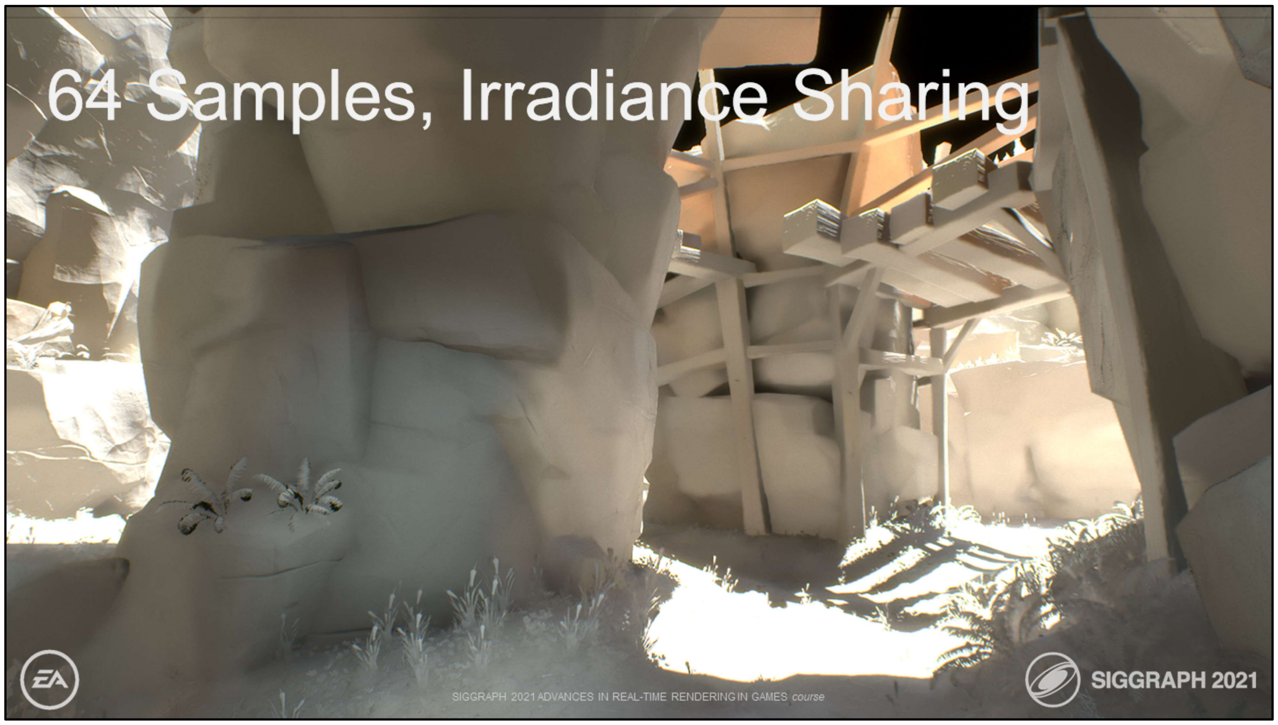
SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES course
SIGGRAPH 2021

Here are some examples of irradiance sharing in action.



64 Samples, No Sharing

Here we have a scene after 64 samples, with no irradiance sharing. As can be seen, the results are pretty blotchy.

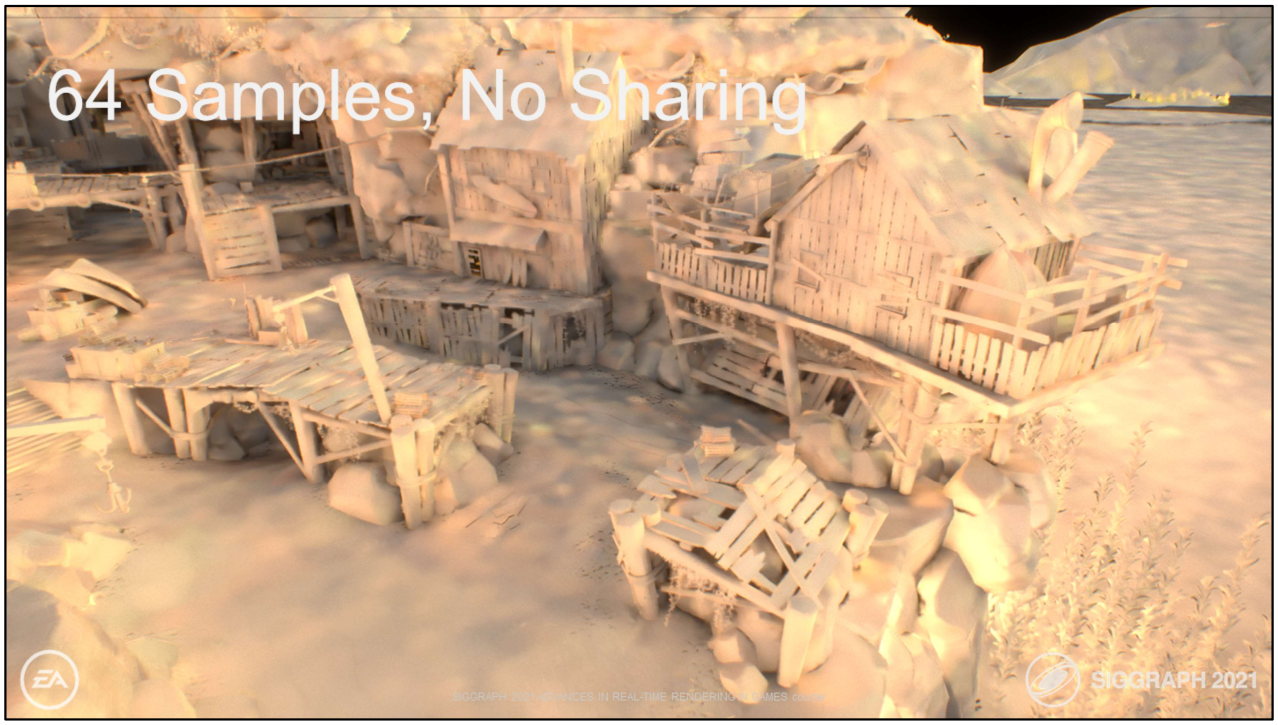


64 Samples, Irradiance Sharing

And this is the result with irradiance sharing after the same 64 samples, which looks much better!

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES course
SIGGRAPH 2021

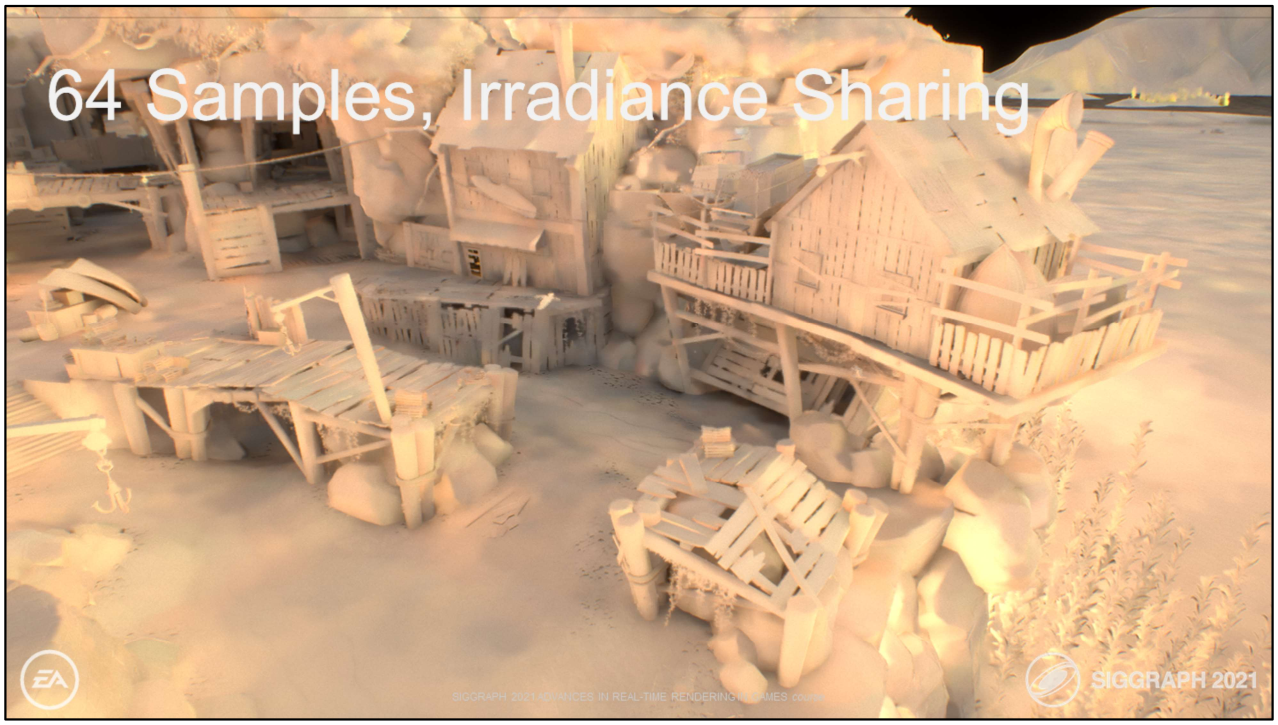
Here is another scene.



64 Samples, No Sharing

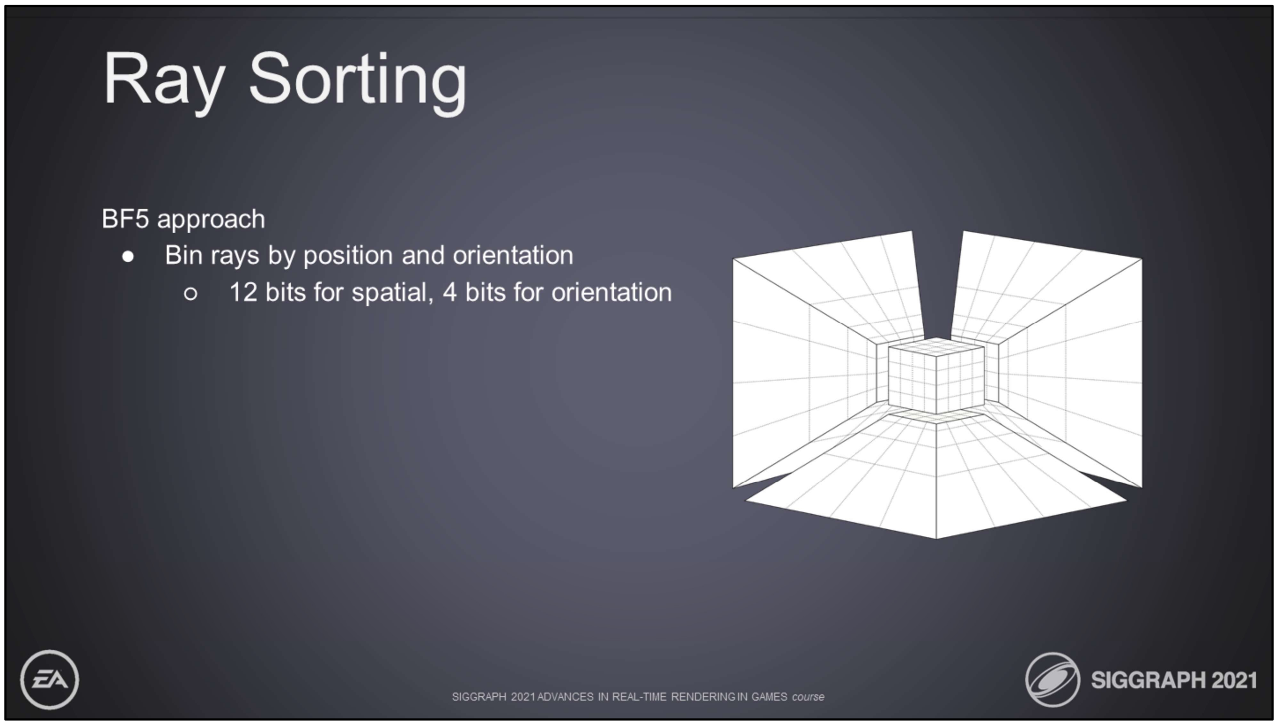
SIGGRAPH 2021

Again this is after 64 samples, and without irradiance sharing. This again looks pretty noisy.



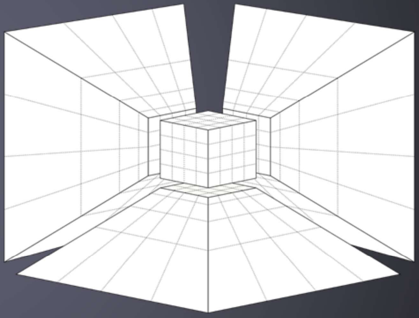
64 Samples, Irradiance Sharing

And with irradiance sharing the results are not perfect, but pretty good after just 64 samples. We should mention that ray guiding is being used in these examples as well.



Ray Sorting

BF5 approach
- Bin rays by position and orientation
  - 12 bits for spatial, 4 bits for orientation



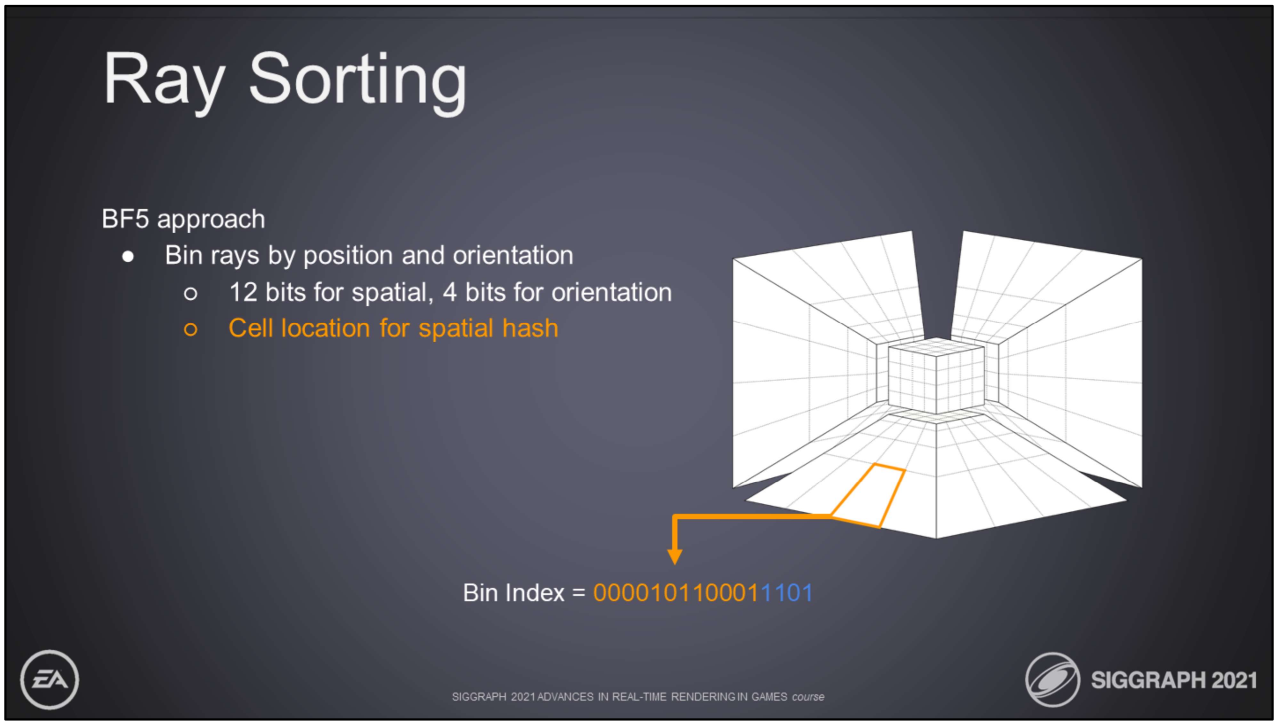
I'll now go over our many-light sampling research, transparency support, and some performance numbers. But first, let's quickly cover how we sort our rays.

Shooting rays with little spatial coherence causes poor performance on most platforms.

Executing threads tend to go down very different parts of the acceleration structure. Which results in poor cache utilization.

So to alleviate this, we've opted to use a sorting strategy similarly to Battlefield 5, known as ray binning.

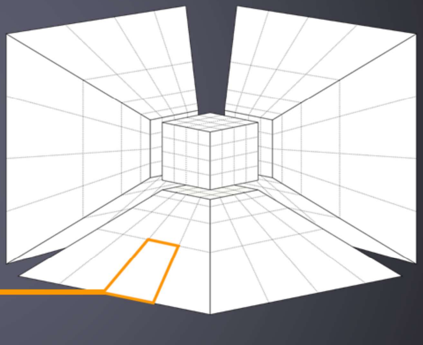
We bin rays based on their position and orientation.



Ray Sorting

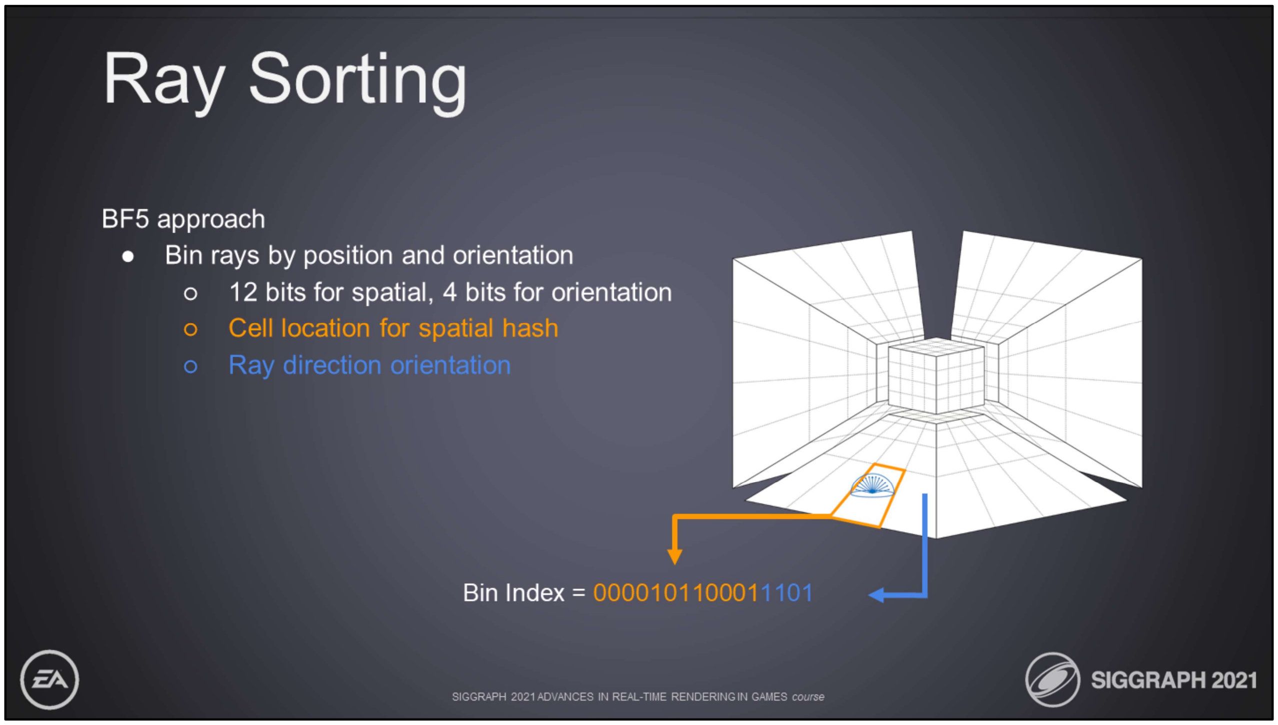
BF5 approach

- Bin rays by position and orientation
  - 12 bits for spatial, 4 bits for orientation
  - Cell location for spatial hash



Bin Index = 0000101100011101

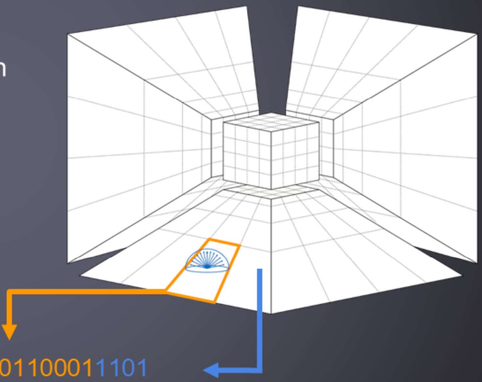
We simply use the surfel's cell coordinate converted to 1D, for the spatial hash. This is the dominant value when calculating the bin index.



Ray Sorting

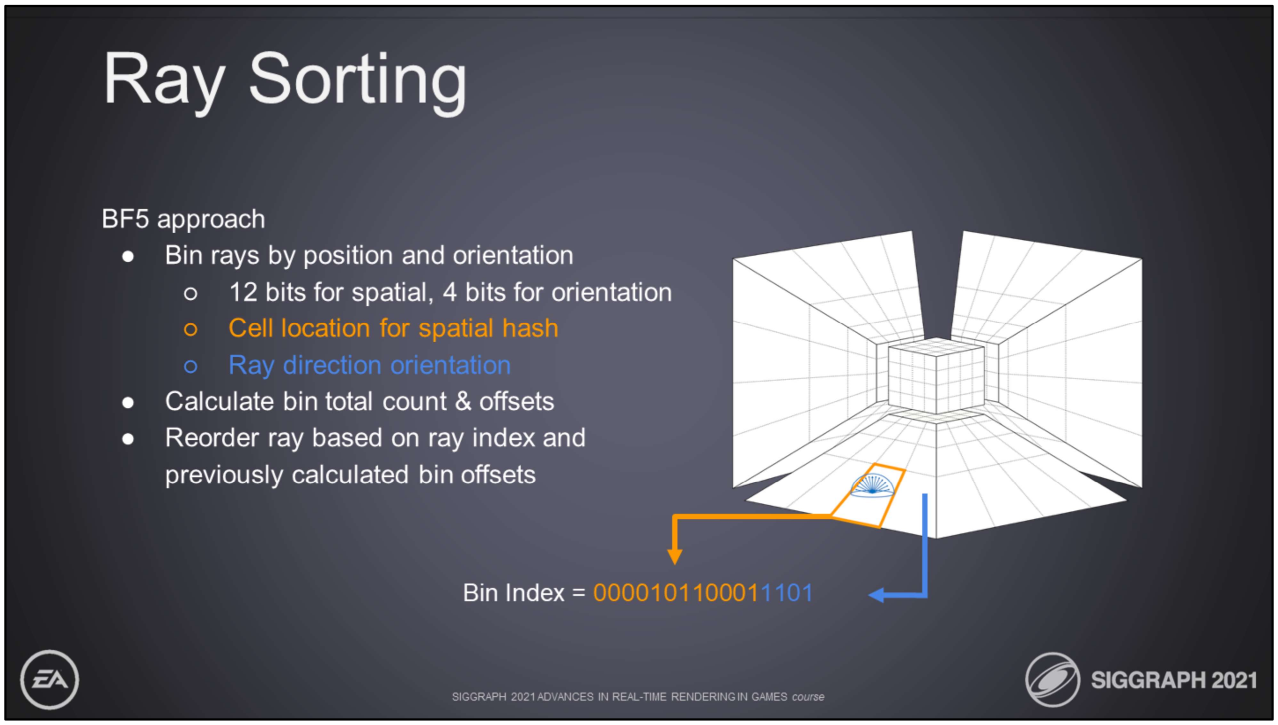
BF5 approach

- Bin rays by position and orientation
    - 12 bits for spatial, 4 bits for orientation
    - Cell location for spatial hash
    - Ray direction orientation



Bin Index = 0000101100011101

And then we use the ray shot from the surfel, as the other part of the bin index.



Ray Sorting

BF5 approach

- Bin rays by position and orientation
  - 12 bits for spatial, 4 bits for orientation
  - Cell location for spatial hash
  - Ray direction orientation
- Calculate bin total count & offsets
- Reorder ray based on ray index and previously calculated bin offsets

Bin Index = 0000101100011101

Then we run 2 passes that generate the re-ordering information we need to sort the rays.

Followed by a final pass that re-orders the rays based on the previously calculated bin counts and offsets.

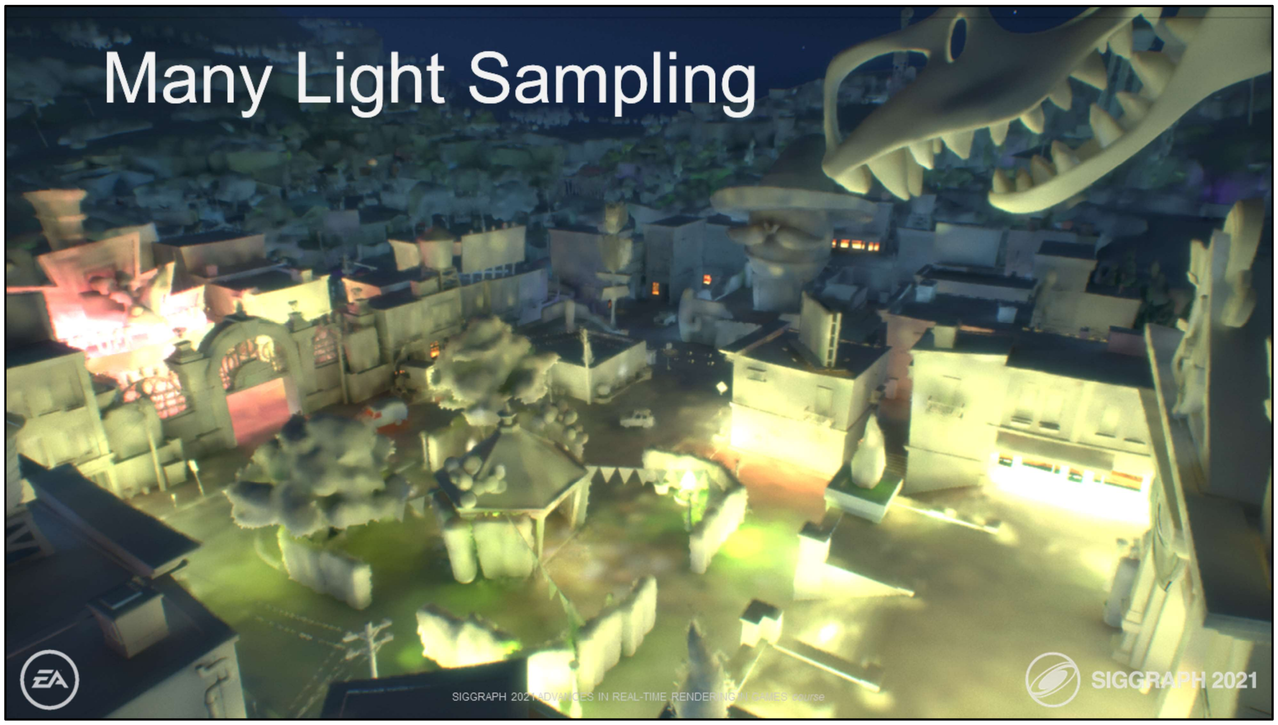
You can see the Battlefield 5 presentation for more information.

Many Light Sampling
SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES course
SIGGRAPH 2021

So far we've discussed how we integrate lighting. But how do we light the ray intersections? Let's talk about our research into many-light, sampling techniques.

Many-light sampling is an ongoing area of research within offline and online computer graphics. But, there have been recent advancements in the last 2 years that have been promising. And we'll discuss our application of those techniques.

This scene is part of our stress test and contains 1400 lights inside the camera frustum.



Many Light Sampling

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES course

SIGGRAPH 2021

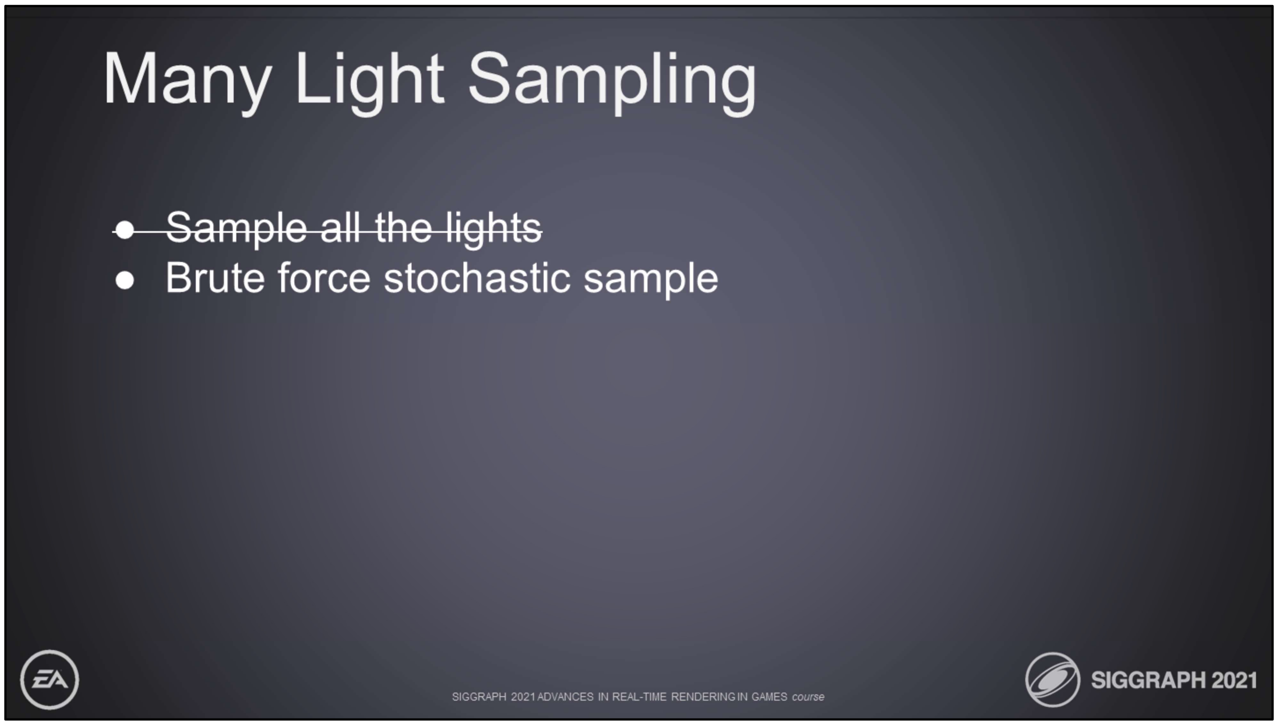
And here’s the indirect diffuse for this scene.

So, what are our options when it comes to sampling hundreds or thousands of lights?

Many Light Sampling

- Sample all the lights

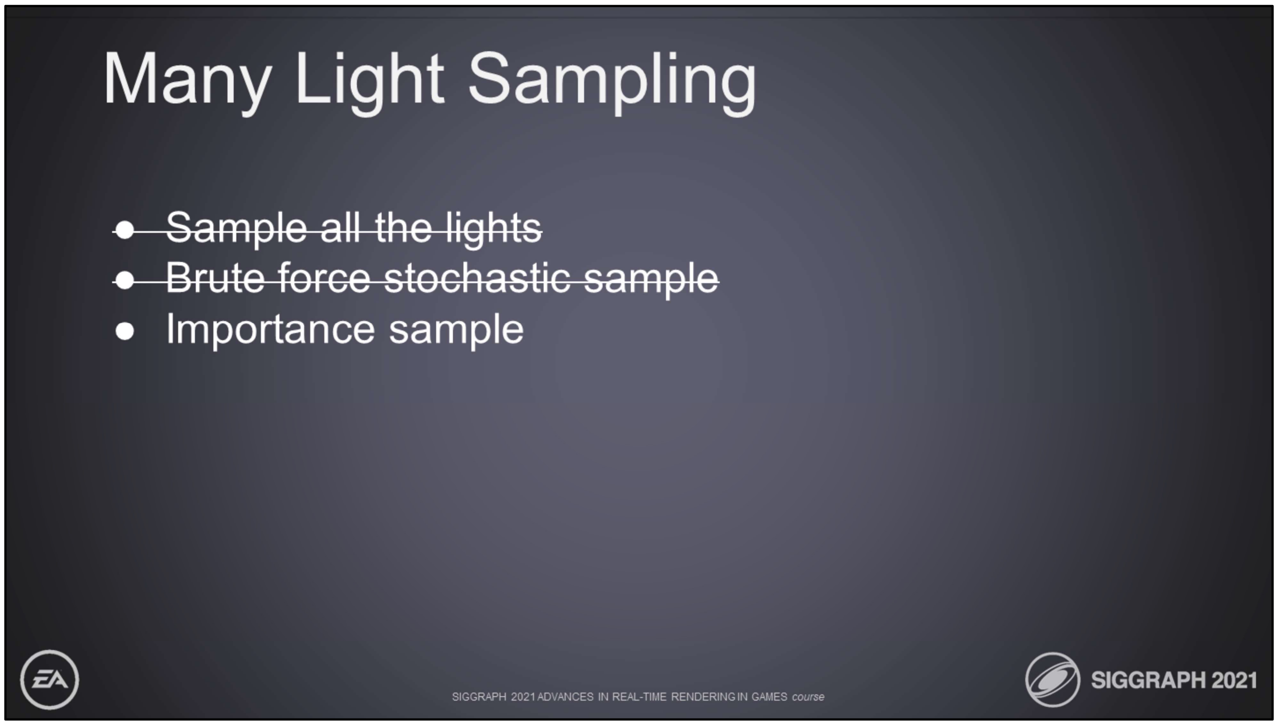
Once you move beyond a few lights, it is no longer feasible to sample all the lights in the scene.



Many Light Sampling

- ~~Sample all the lights~~
- Brute force stochastic sample

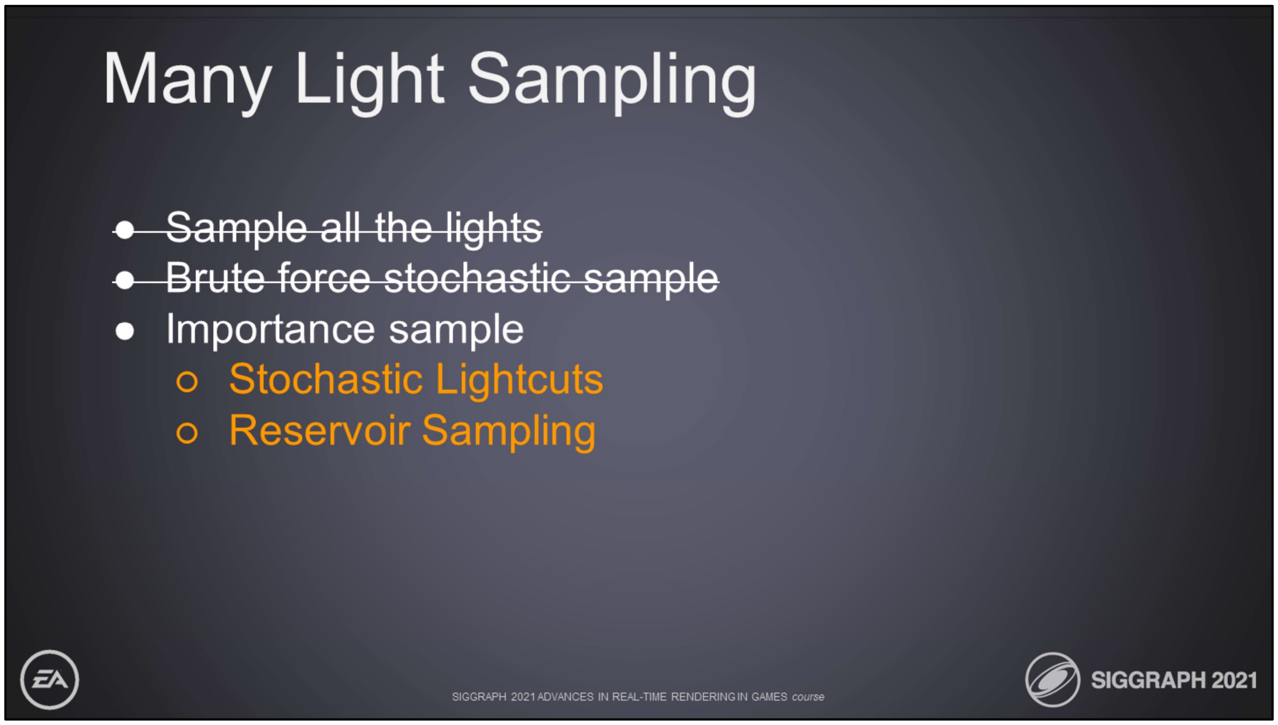
A first step solution is to stochastically sample N lights. This might work well when there are 10s of lights in an area, and is also dirt cheap performance wise if N is small. However, once you move to hundreds or thousands of lights, this approach starts taking too long to converge.



Many Light Sampling

- ~~Sample all the lights~~
- ~~Brute force stochastic sample~~
- Importance sample

So, we need some sort of importance sampling to combat the convergence problem.

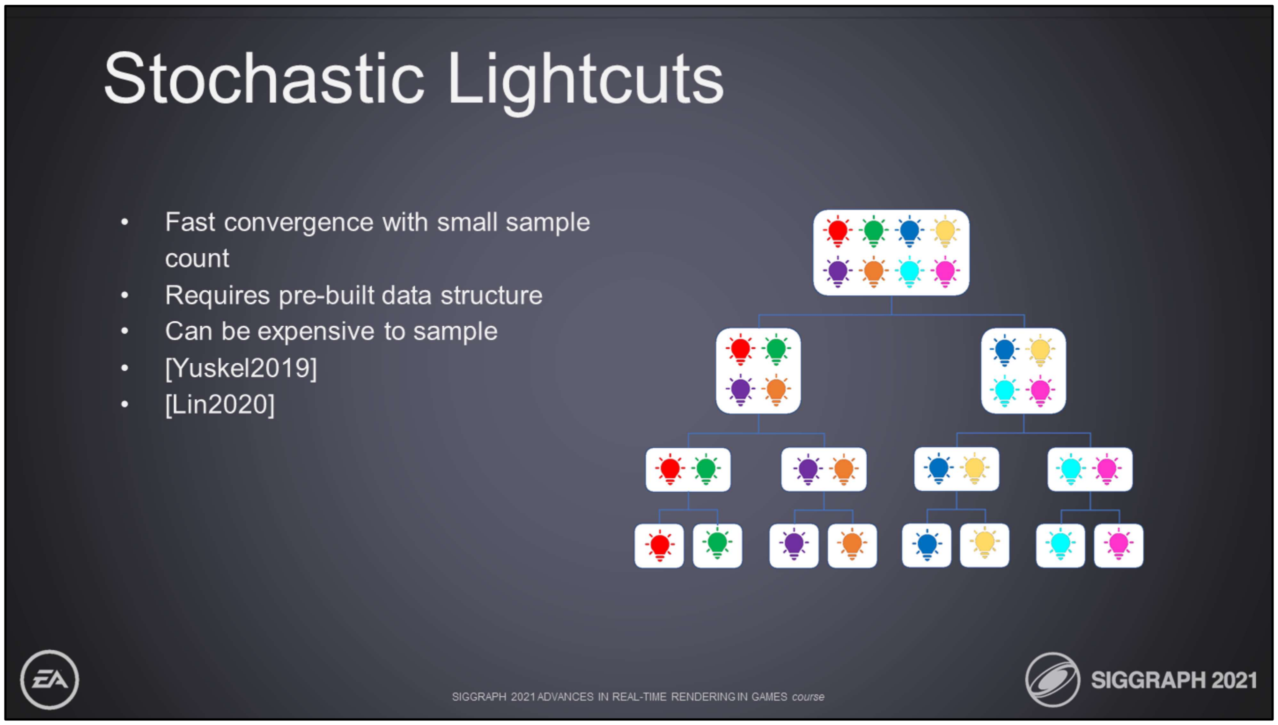


Many Light Sampling

- ~~Sample all the lights~~
- ~~Brute force stochastic sample~~
- Importance sample
  - Stochastic Lightcuts
  - Reservoir Sampling

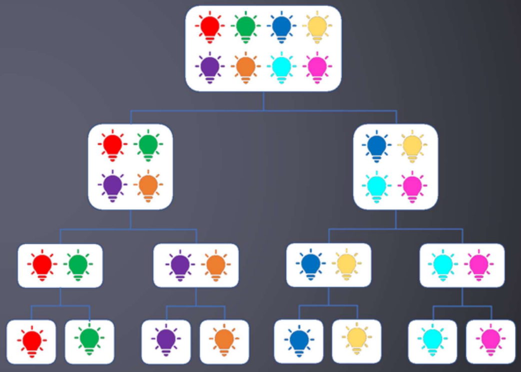
SIGGRAPH 2021

We've investigated a few solutions and will present the two most promising avenues: stochastic light-cus, and reservoir sampling.



Stochastic Lightcuts

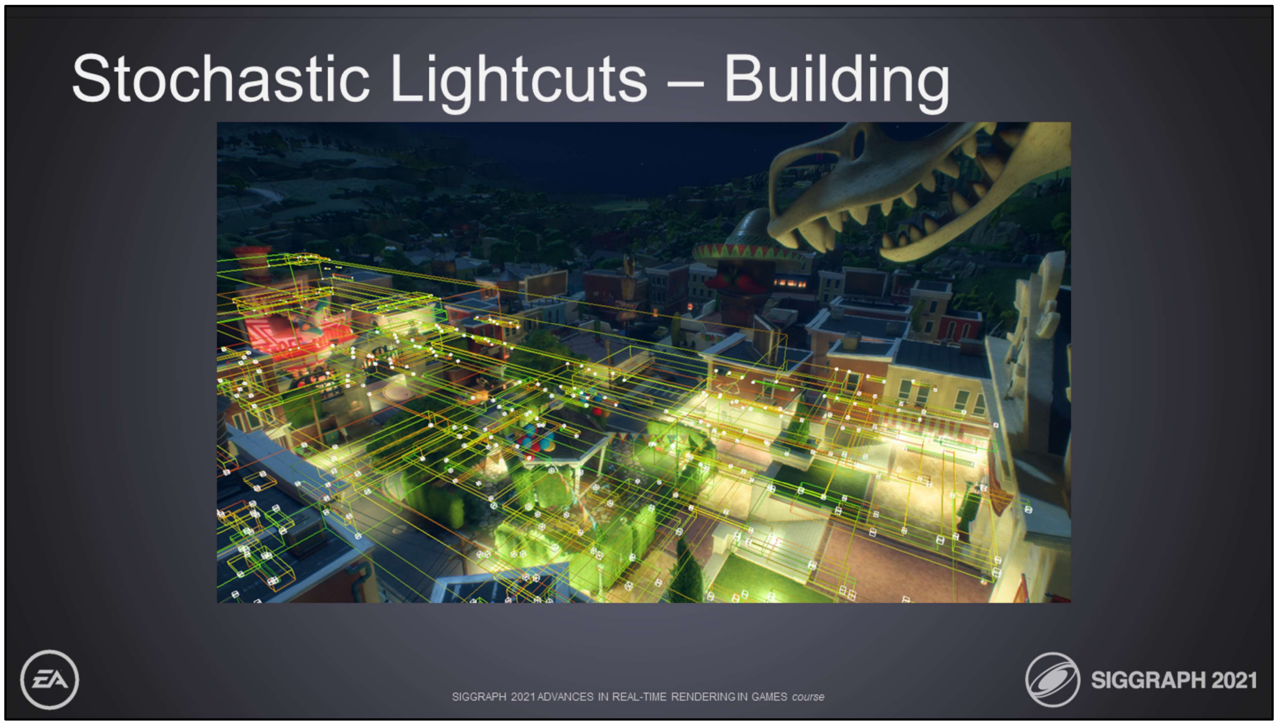
- Fast convergence with small sample count
- Requires pre-built data structure
- Can be expensive to sample
- [Yuskel2019]
- [Lin2020]



SIGGRAPH 2021

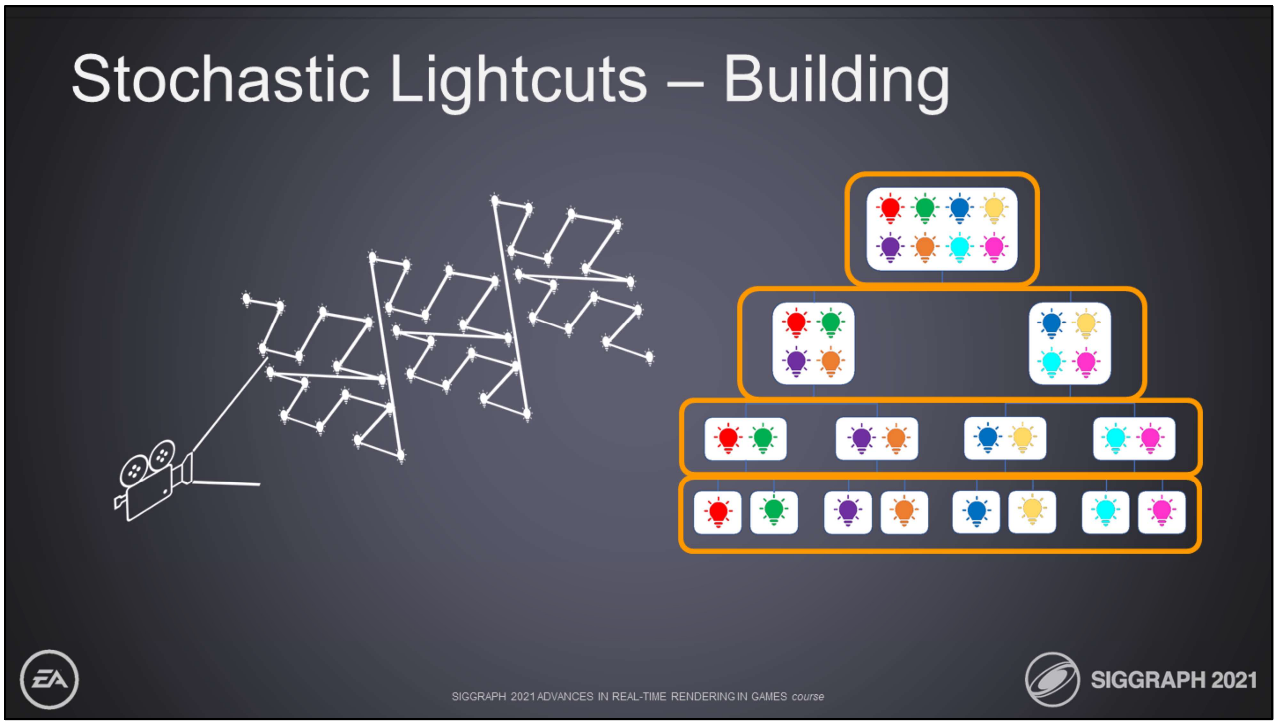
The first solution I'll discuss is stochastic lightcuts. This solution is based on the work of Cem Yuksel from HPG 2019. Stochastic-light cuts builds upon lightcuts by minimizing biased sampling of the light tree and improving sampling efficiency.

This research is attractive, because it does not need a large amount of samples to converge, and requires no spatial or temporary storage beyond the light data structure; unlike resampled reservoir sampling which we'll cover in a few slides.



Stochastic Lightcuts – Building

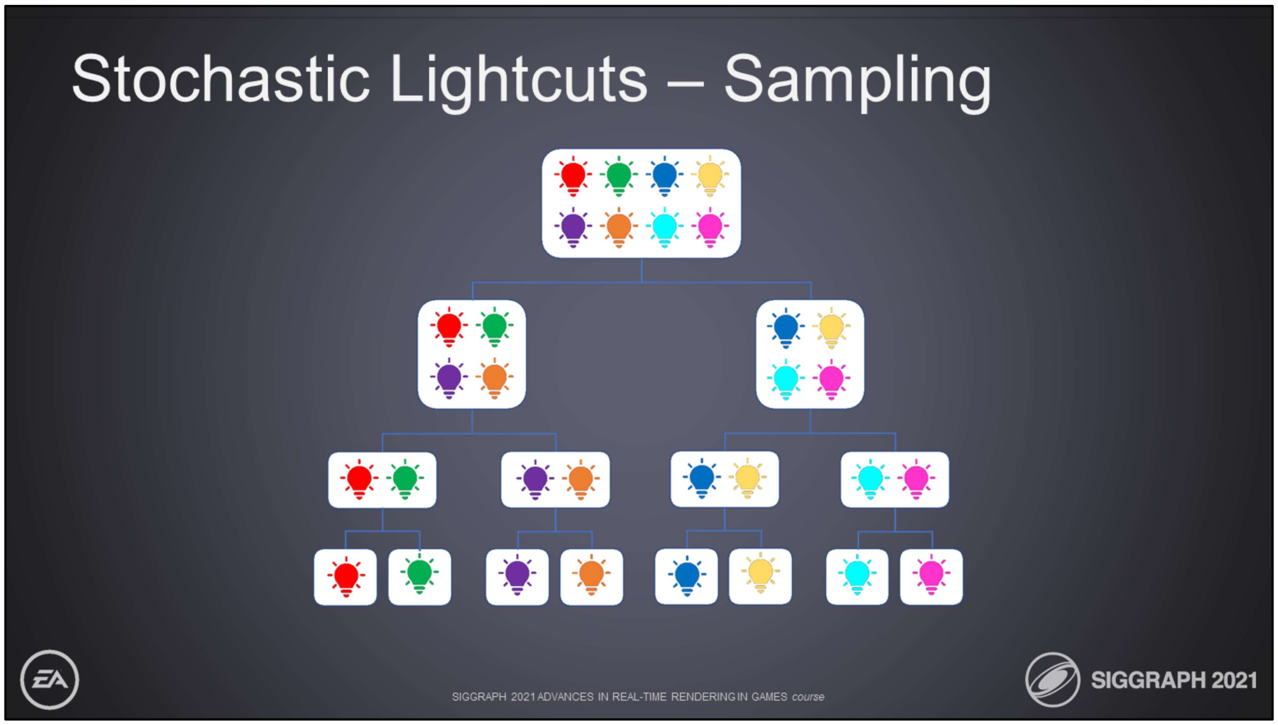
So, how do we build our light tree?



Stochastic Lightcuts – Building

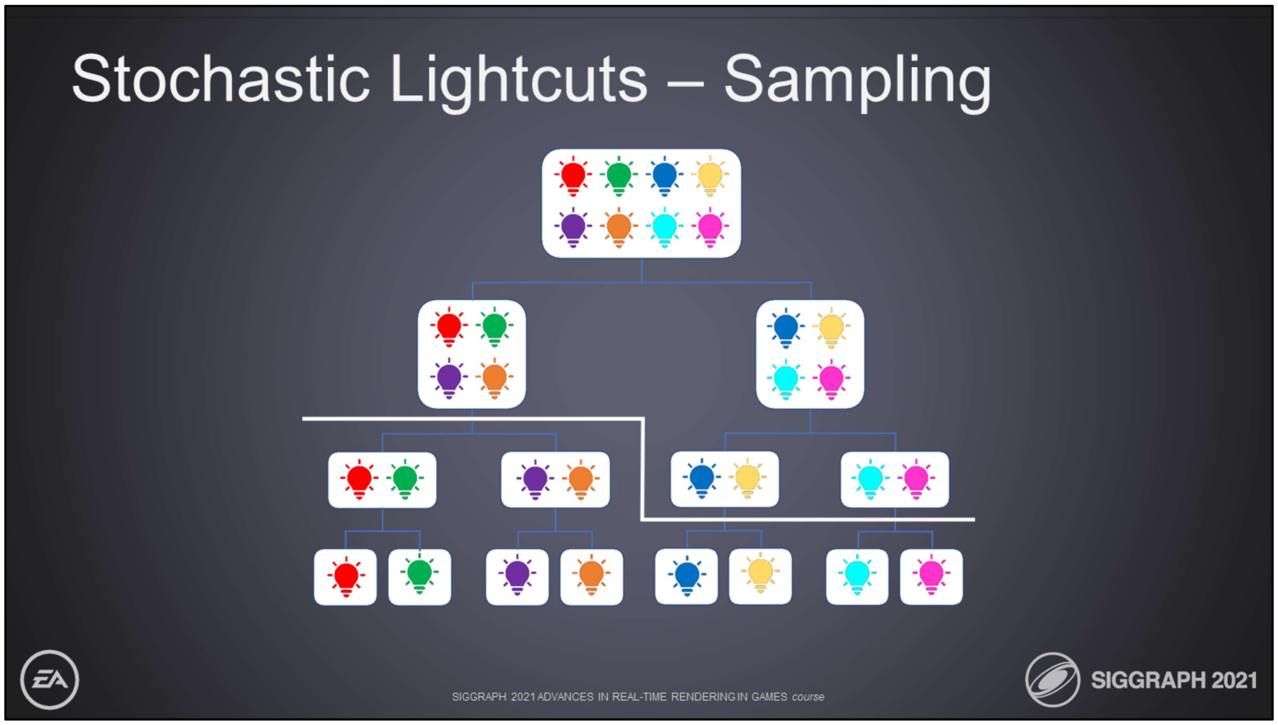
We store light positions in view space for precision, and sort the lights based on a morton encoding of their position. This sorting helps group lights spatially, so that the tree has implicit spatial correlation. And we don't need to do multiple sorts during construction as with BVH

We then build the tree bottom up, where each internal node represents the combined bounds, and intensities, of the children below it.



Stochastic Lightcuts – Sampling

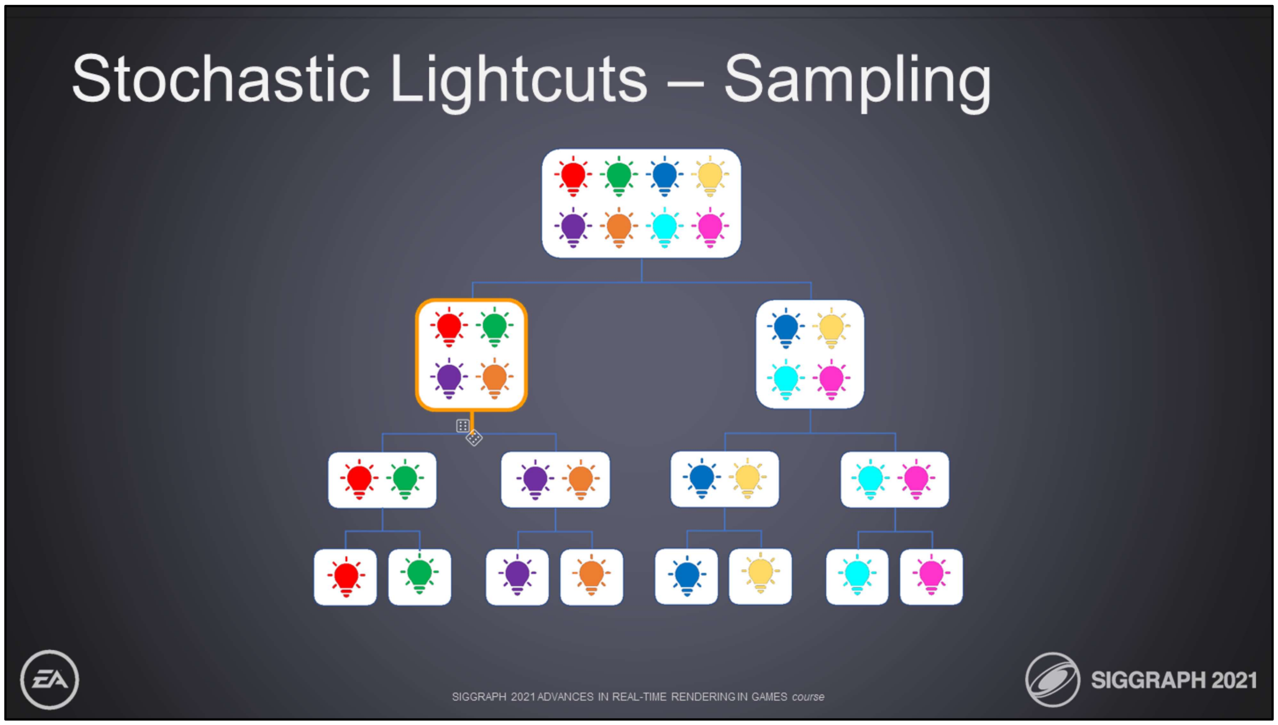
Now that we know how we build the tree, let’s walk through the process of sampling from the tree.



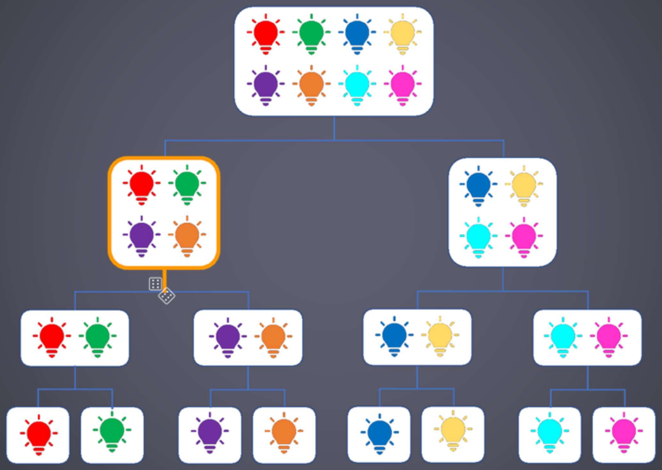
Stochastic Lightcuts – Sampling

To sample the tree we first choose a cut through the tree, based on a user defined node limit. This is typically 2-8 nodes, depending on quality level and platform. This count corresponds to the number of lights that we will eventually shoot rays towards to compute visibility.

We compute the cut through the tree by choosing nodes that minimize the lighting error based on the sampling position compared to the nodes in the tree.



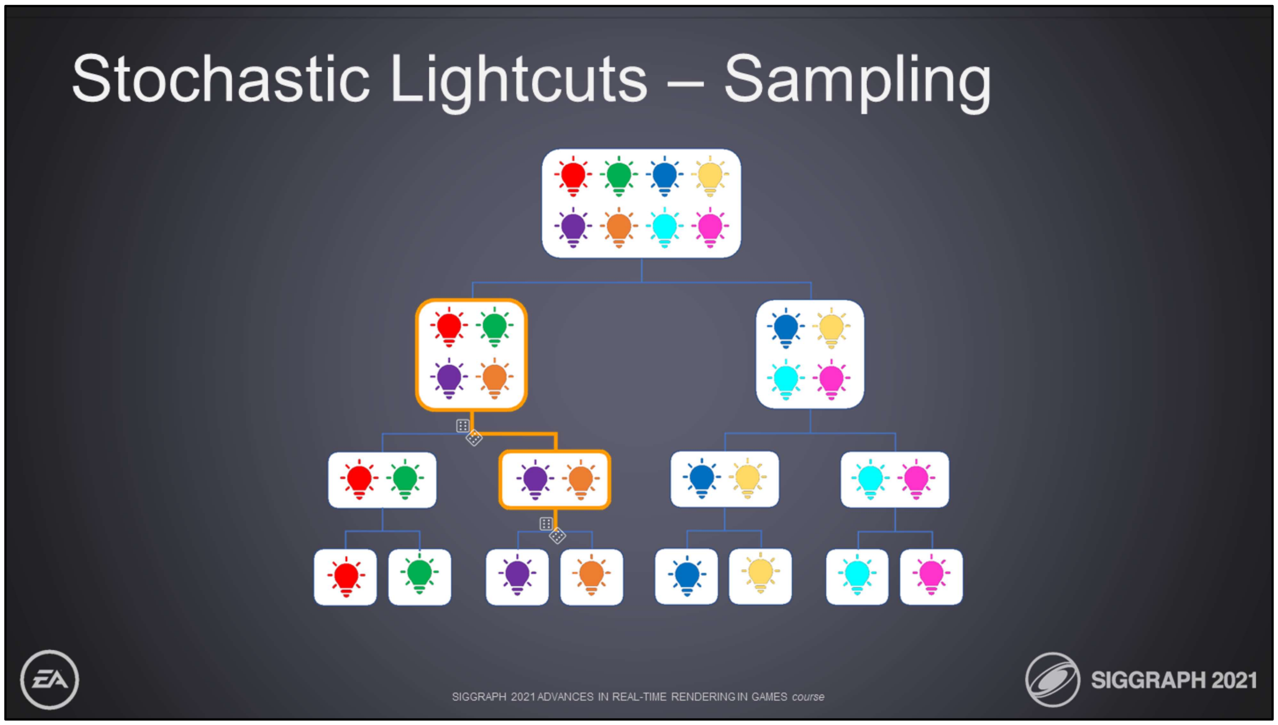
Stochastic Lightcuts – Sampling



SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

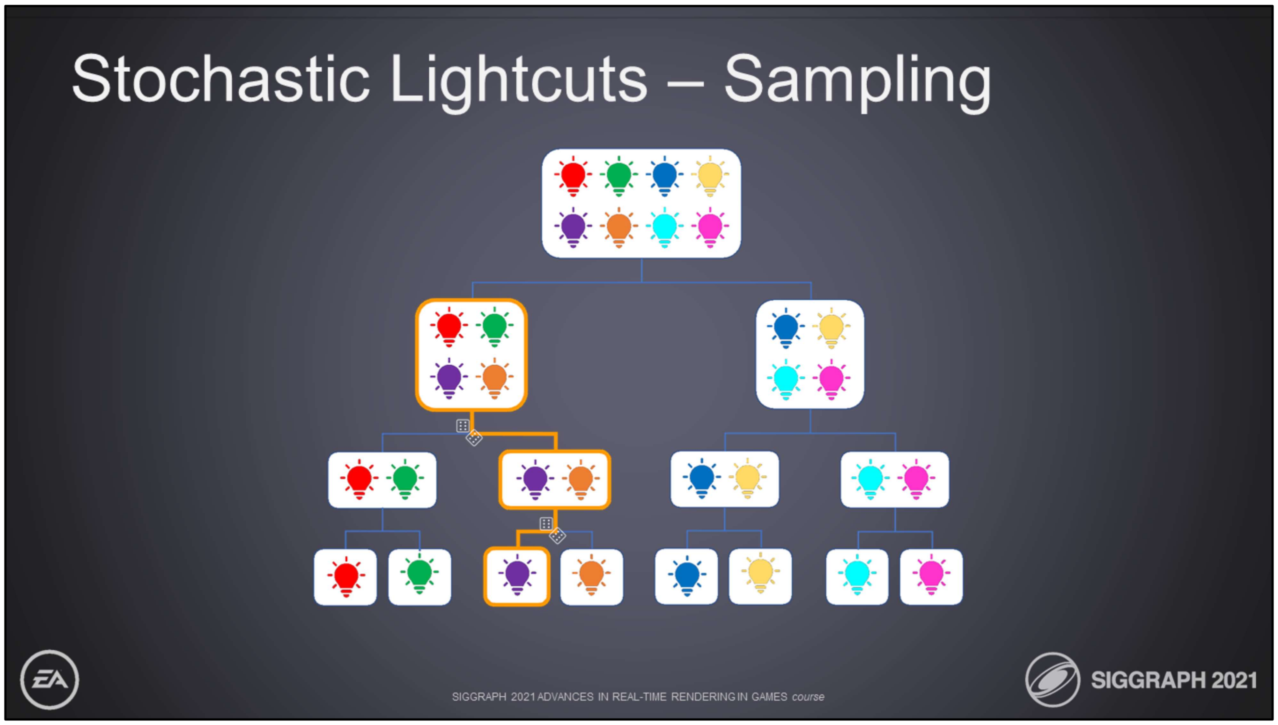
SIGGRAPH 2021

Once we’ve computed the cut, we traverse, stochastically, down the tree from each root node in the cut.

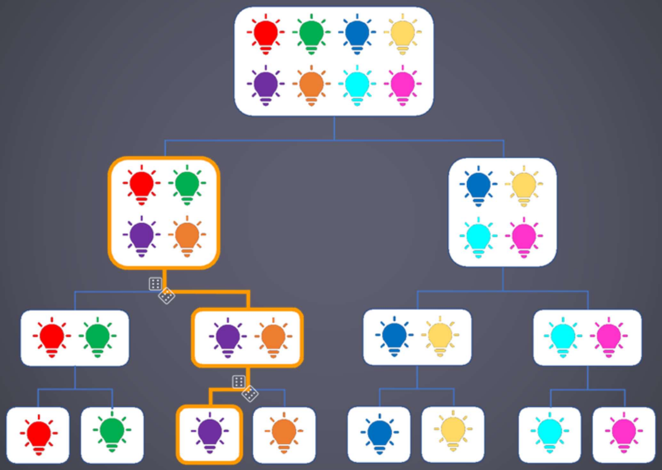


Stochastic Lightcuts – Sampling

For each internal node, we assign probabilities based on importance weights to its children, **and** then randomly select the left or right child based on their probability.

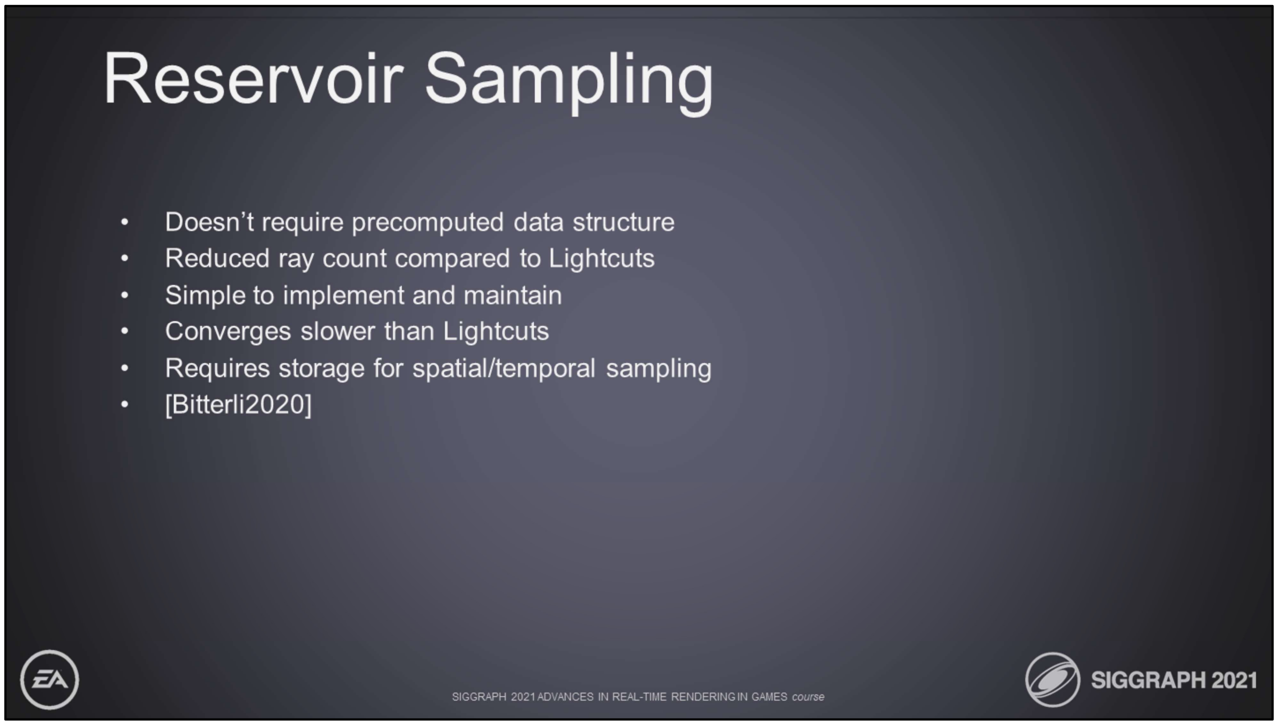


Stochastic Lightcuts – Sampling



SIGGRAPH 2021

For each internal node, we assign probabilities based on importance weights to its children, and then randomly select the left or right child based on its probability.



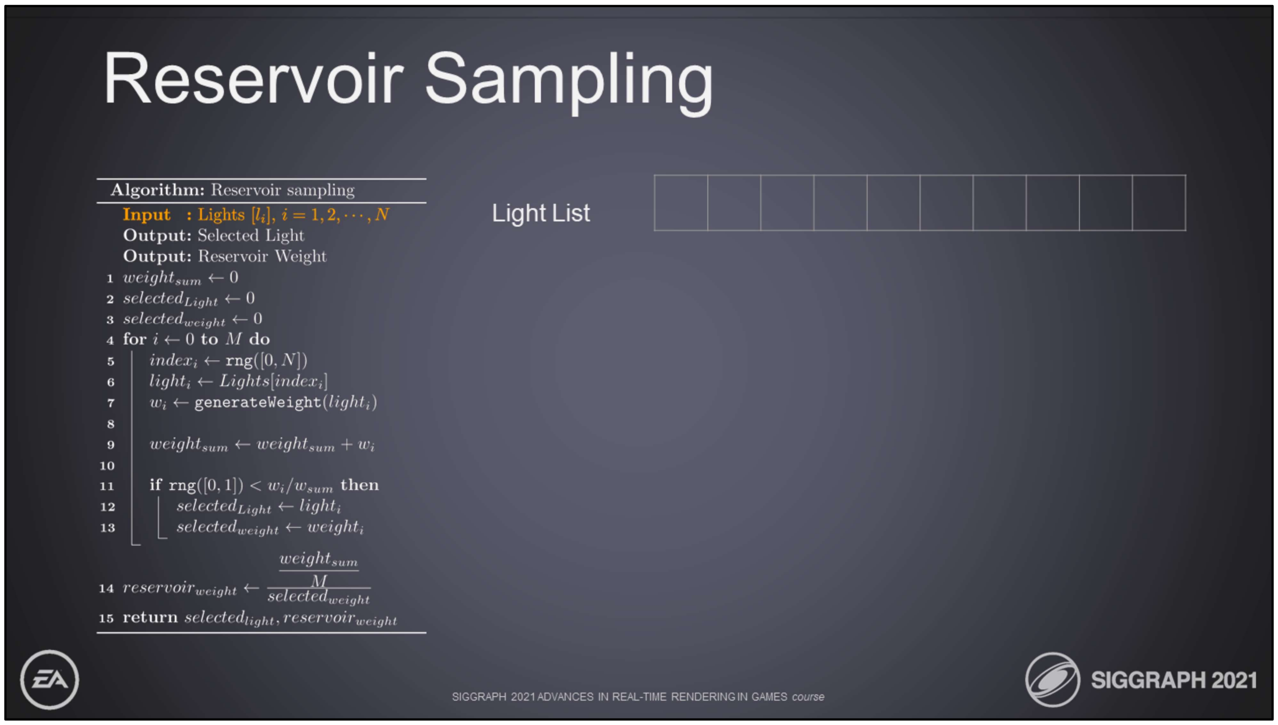
Reservoir Sampling

- Doesn't require precomputed data structure
- Reduced ray count compared to Lightcuts
- Simple to implement and maintain
- Converges slower than Lightcuts
- Requires storage for spatial/temporal sampling
- [Bitterli2020]

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

Another light sampling solution we've investigated is reservoir sampling, based on the ReStir research by Nvidia. Unlike lightcuts, reservoir sampling requires no precomputed data structure for the lights. Also, another benefit, is that it only requires one ray per chosen sample, which makes it's cheaper on consoles compared to our implementation of stochastic lightcuts.



Reservoir Sampling

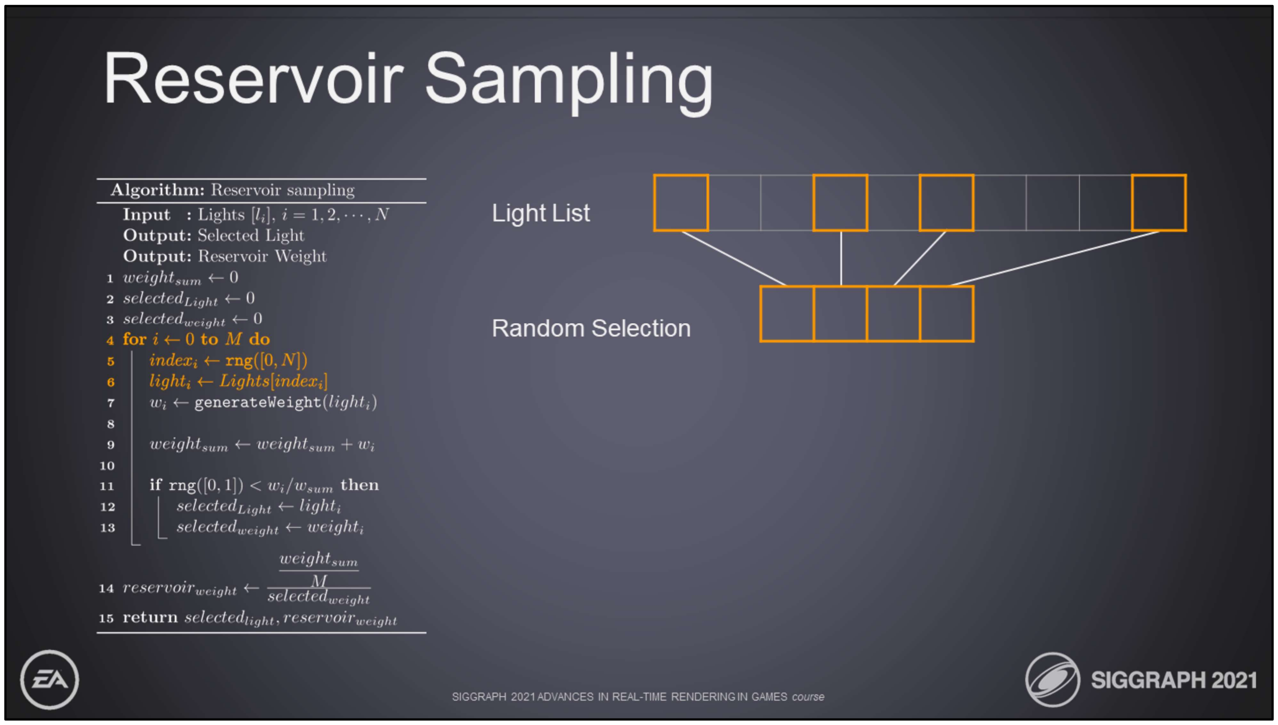
```
Algorithm: Reservoir sampling
    Input  : Lights $[l_i]$, $i = 1, 2, \cdots, N$
    Output: Selected Light
    Output: Reservoir Weight
$weight_{sum} \leftarrow 0$
$selected_{Light} \leftarrow 0$
$selected_{weight} \leftarrow 0$
for $i \leftarrow 0$ to $M$ do
    $index_i \leftarrow \mathtt{rng}([0, N])$
    $light_i \leftarrow Lights[index_i]$
    $w_i \leftarrow \mathtt{generateWeight}(light_i)$

    $weight_{sum} \leftarrow weight_{sum} + w_i$

    if $\mathtt{rng}([0, 1]) < w_i / w_{sum}$ then
        $selected_{Light} \leftarrow light_i$
        $selected_{weight} \leftarrow weight_i$
$reservoir_{weight} \leftarrow \frac{\frac{weight_{sum}}{M}}{selected_{weight}}$
return $selected_{light}, reservoir_{weight}$
```

Light List

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

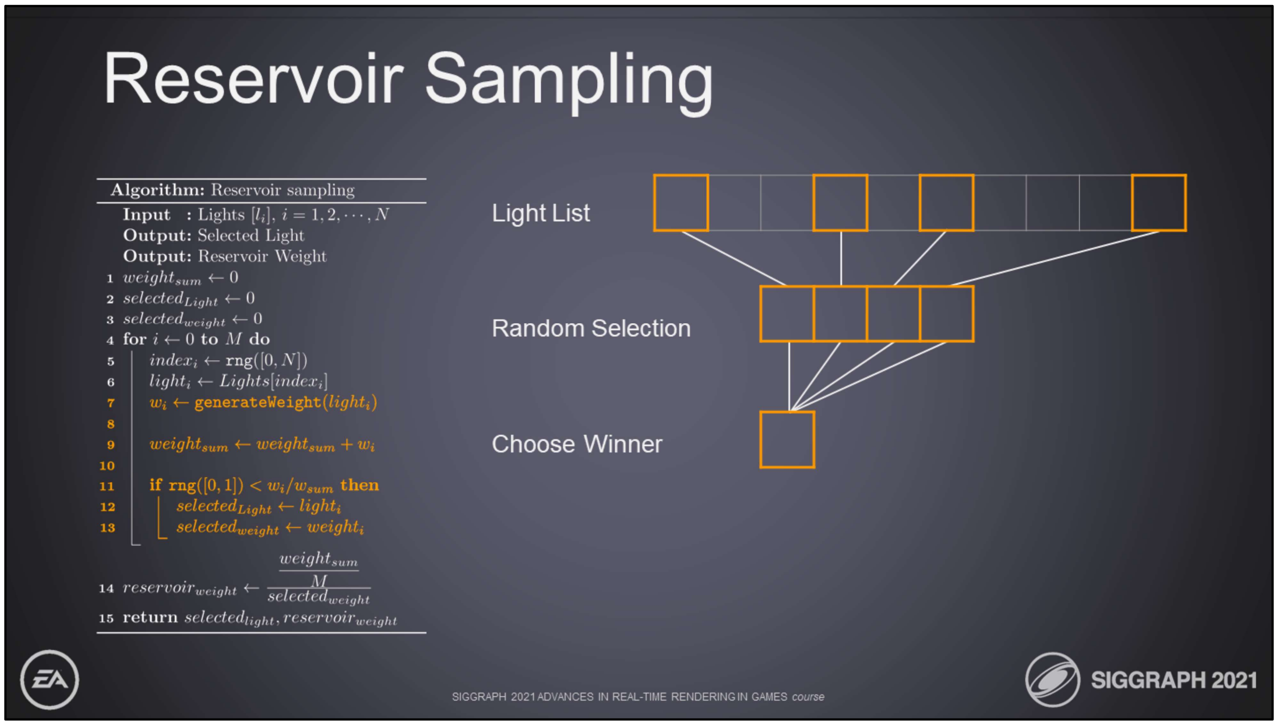
The essential idea behind reservoir sampling is that you stochastically evaluate N lights, but only choose M winners to shoot rays against.



Reservoir Sampling

```
Algorithm: Reservoir sampling
  Input  : Lights $[l_i]$, $i = 1, 2, \cdots, N$
  Output: Selected Light
  Output: Reservoir Weight
$weight_{sum} \leftarrow 0$
$selected_{Light} \leftarrow 0$
$selected_{weight} \leftarrow 0$
for $i \leftarrow 0$ to $M$ do
    $index_i \leftarrow \texttt{rng}([0, N])$
    $light_i \leftarrow Lights[index_i]$
    $w_i \leftarrow \texttt{generateWeight}(light_i)$

    $weight_{sum} \leftarrow weight_{sum} + w_i$

    if $\texttt{rng}([0, 1]) < w_i / w_{sum}$ then
        $selected_{Light} \leftarrow light_i$
        $selected_{weight} \leftarrow weight_i$
$reservoir_{weight} \leftarrow \frac{\frac{weight_{sum}}{M}}{selected_{weight}}$
return $selected_{light}, reservoir_{weight}$
```

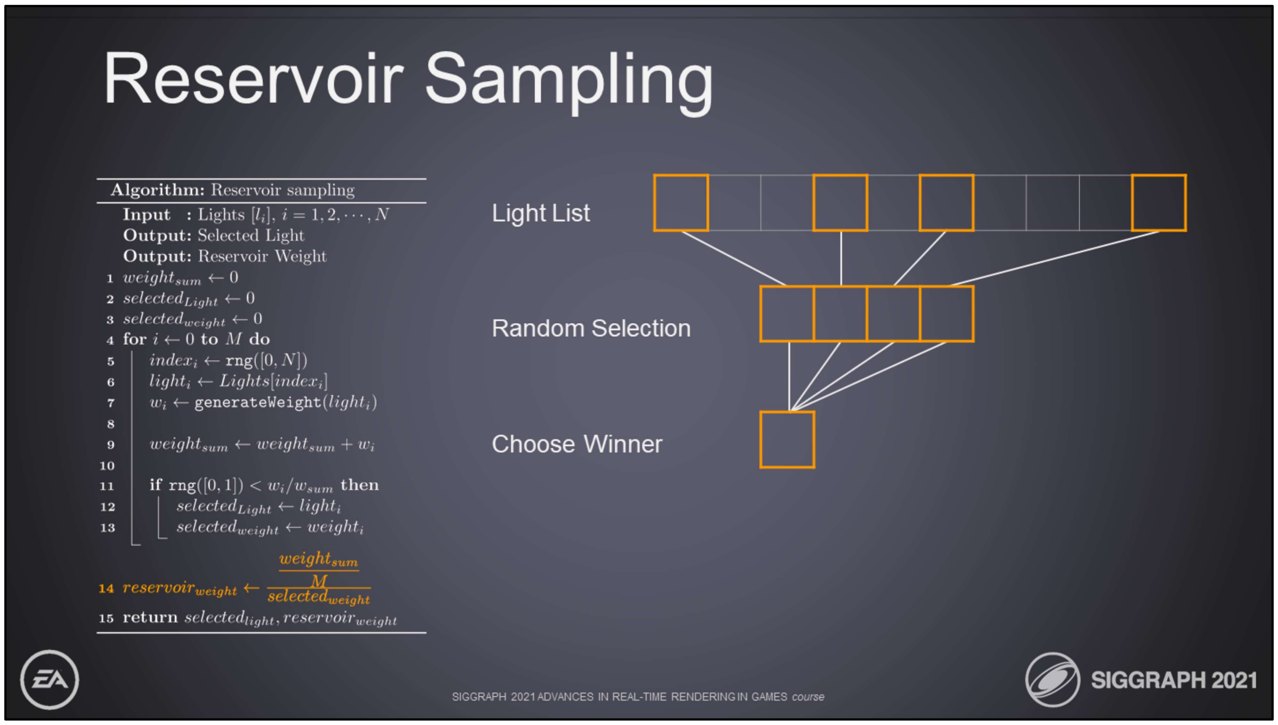
In our case, we randomly sample four to eight lights, and choose one winner.



Reservoir Sampling

```
Algorithm: Reservoir sampling
  Input  : Lights $[l_i]$, $i = 1, 2, \cdots, N$
  Output: Selected Light
  Output: Reservoir Weight
  $weight_{sum} \leftarrow 0$
  $selected_{Light} \leftarrow 0$
  $selected_{weight} \leftarrow 0$
  for $i \leftarrow 0$ to $M$ do
      $index_i \leftarrow \mathrm{rng}([0, N])$
      $light_i \leftarrow Lights[index_i]$
      $w_i \leftarrow \texttt{generateWeight}(light_i)$

      $weight_{sum} \leftarrow weight_{sum} + w_i$

      if $\mathrm{rng}([0, 1]) < w_i / w_{sum}$ then
          $selected_{Light} \leftarrow light_i$
          $selected_{weight} \leftarrow weight_i$
  $reservoir_{weight} \leftarrow \frac{\frac{weight_{sum}}{M}}{selected_{weight}}$
  return $selected_{light}, reservoir_{weight}$
```

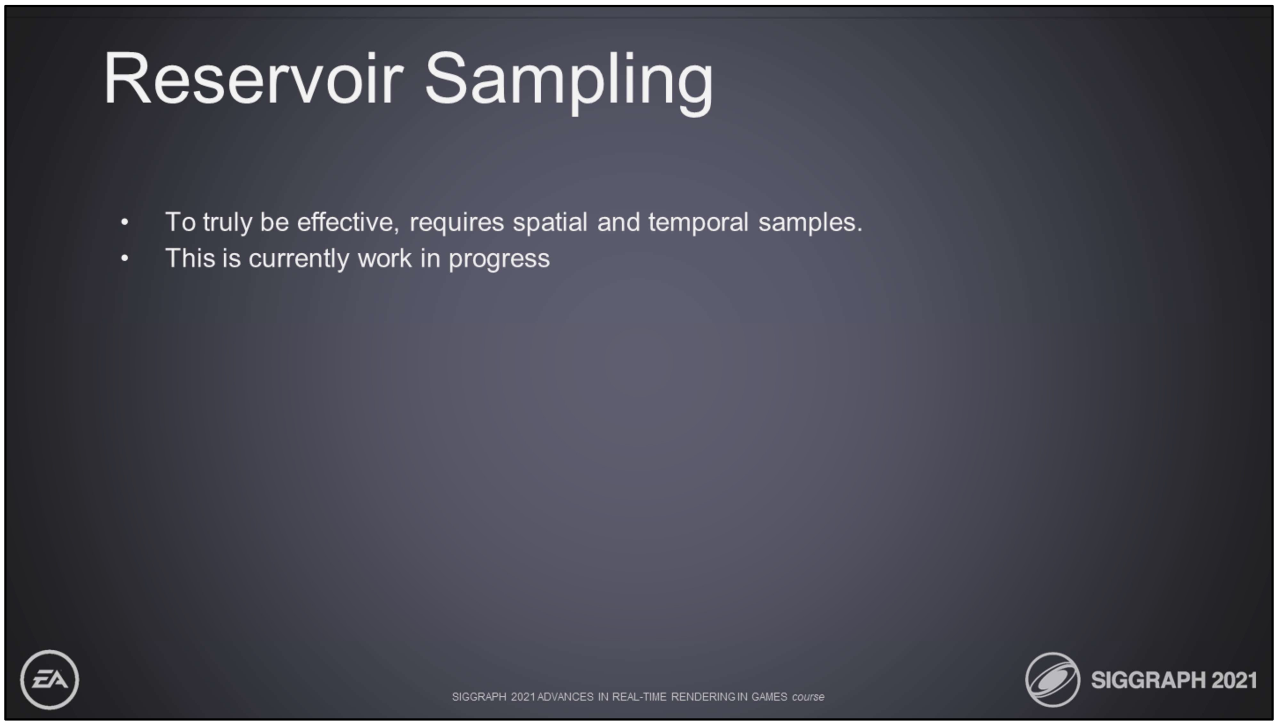
The winner is chosen by calculating the weight of the light, based on its distance and intensity. We then generate a random number, and if that number is less than the weight of the light divided by the total weight, we select that light.



Reservoir Sampling

```
Algorithm: Reservoir sampling
  Input  : Lights $[l_i]$, $i = 1, 2, \cdots, N$
  Output: Selected Light
  Output: Reservoir Weight
$weight_{sum} \leftarrow 0$
$selected_{Light} \leftarrow 0$
$selected_{weight} \leftarrow 0$
for $i \leftarrow 0$ to $M$ do
    $index_i \leftarrow$ rng$([0, N])$
    $light_i \leftarrow Lights[index_i]$
    $w_i \leftarrow$ generateWeight$(light_i)$

    $weight_{sum} \leftarrow weight_{sum} + w_i$

    if rng$([0, 1]) < w_i / w_{sum}$ then
        $selected_{Light} \leftarrow light_i$
        $selected_{weight} \leftarrow weight_i$
$reservoir_{weight} \leftarrow \frac{\frac{weight_{sum}}{M}}{selected_{weight}}$
return $selected_{light}, reservoir_{weight}$
```

And finally, we normalize the resulting pdf by the total weight of all sampled lights.



Reservoir Sampling

- To truly be effective, requires spatial and temporal samples.
- This is currently work in progress

SIGGRAPH 2021

The downside here is that in-order to get the most out of reservoir sampling, you really want to compare spatial and temporal samples to your currently-chosen reservoir sample.

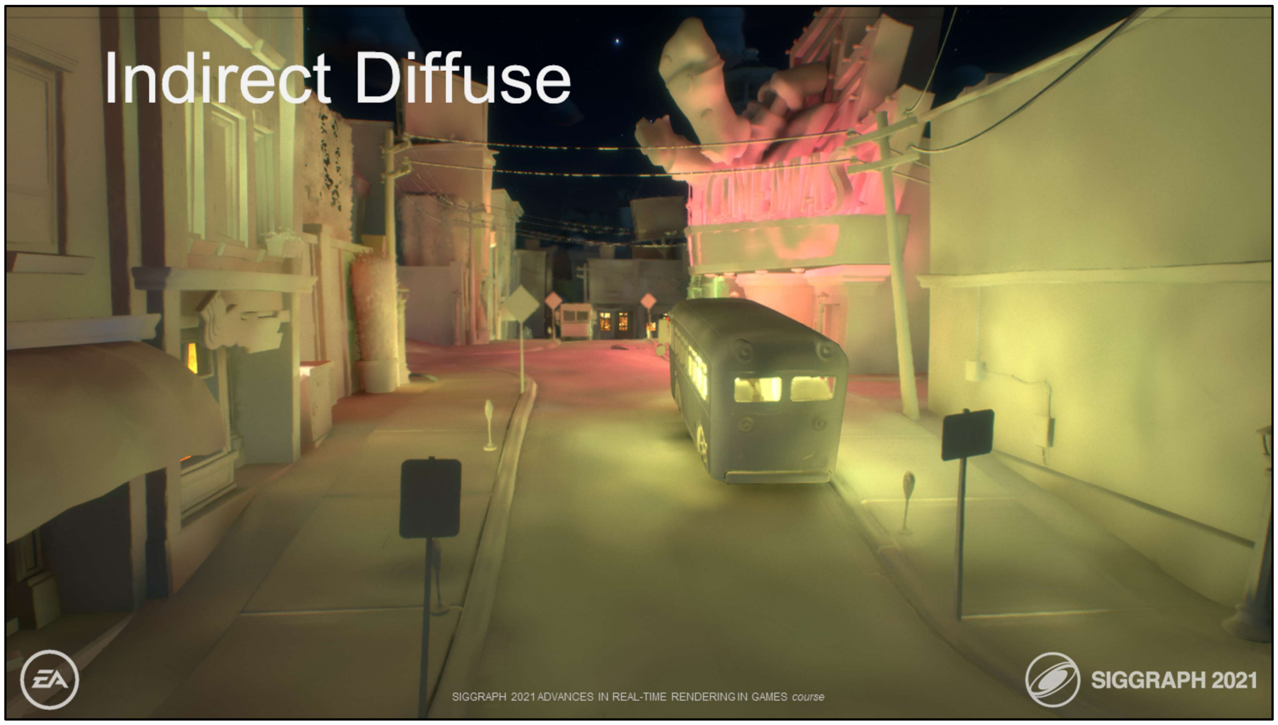
This is called reservoir resampling. This is a bit complicated to do for indirect lighting, because it requires extra storage to save your chosen sample, to then later compare against other temporal and spatial samples. And this is an area we're currently working on supporting.

Final Lighting

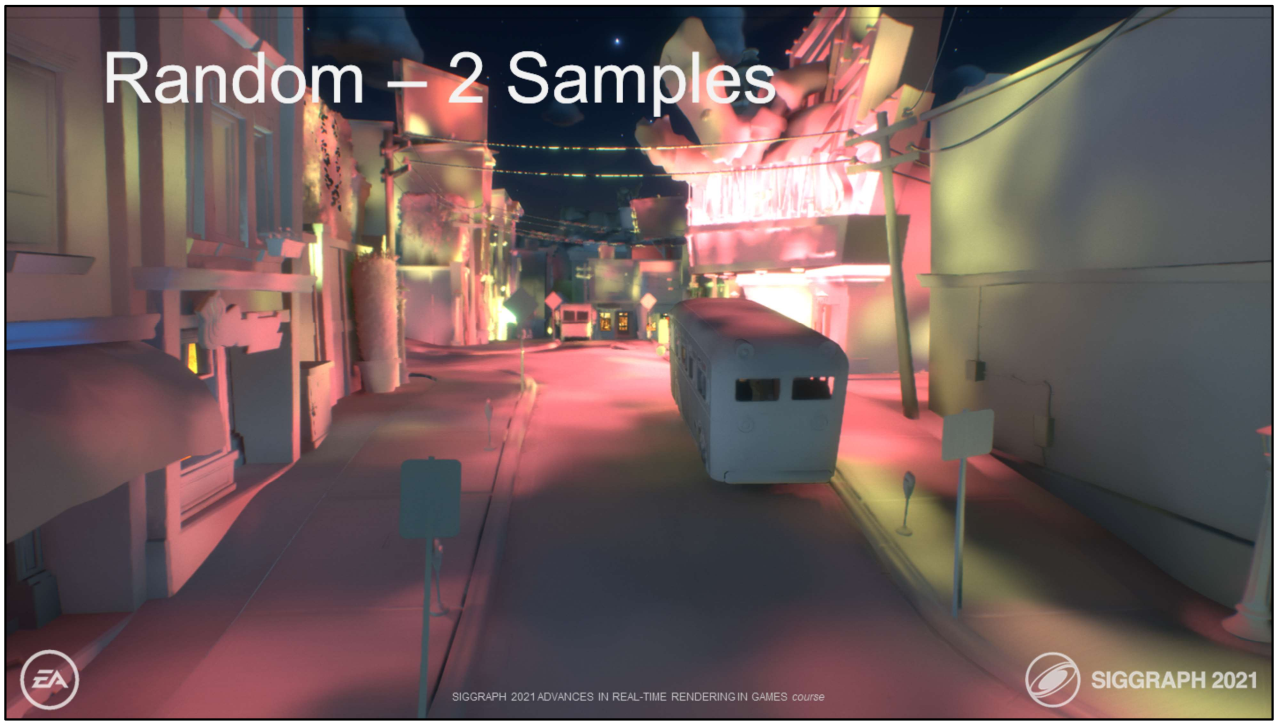
COMING SOON
SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES course
SIGGRAPH 2021

Now I’ll show some examples of each technique. In this shot we have around 700 lights in view, with another 700 outside the frustum.

Indirect Diffuse


SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*
SIGGRAPH 2021

And here's the Indirect diffuse, after it has converged to acceptable quality in about five seconds. In the following slides I'll show each technique after solving for only 15 frames.

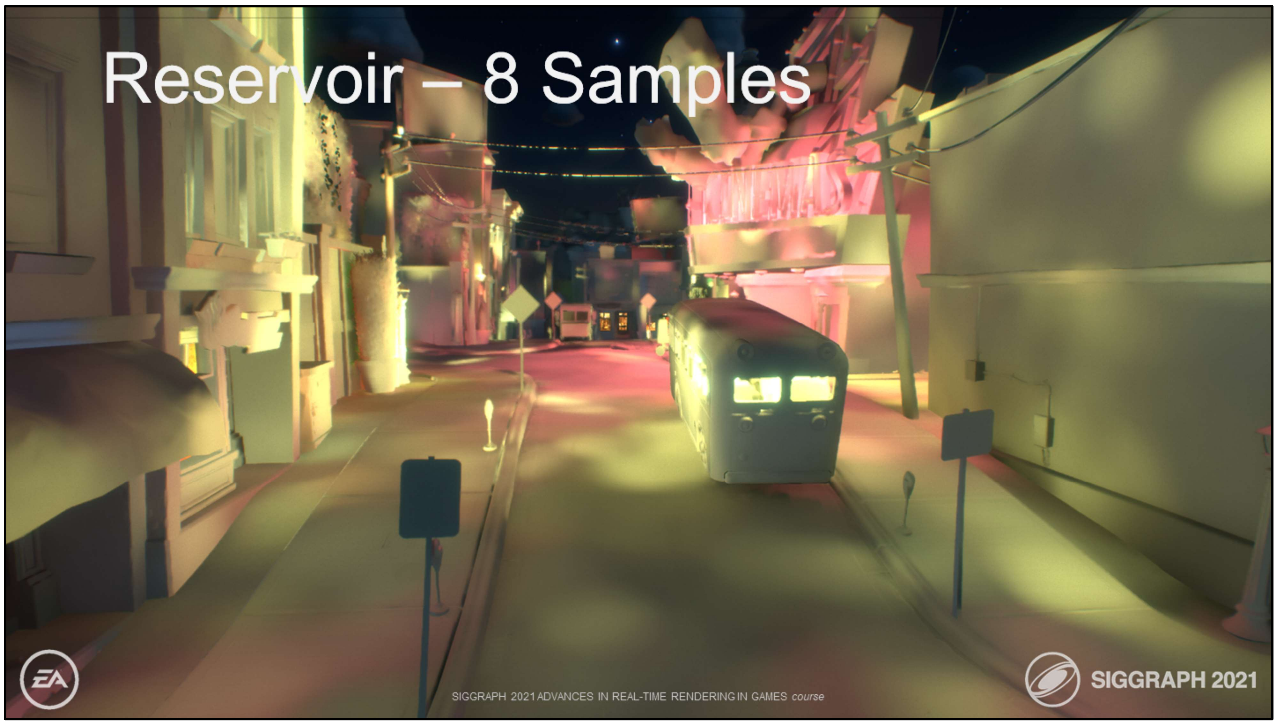


Random – 2 Samples

The 1st example here is 2 sample brute-force random sampling, which means we will shoot 2 shadow rays.

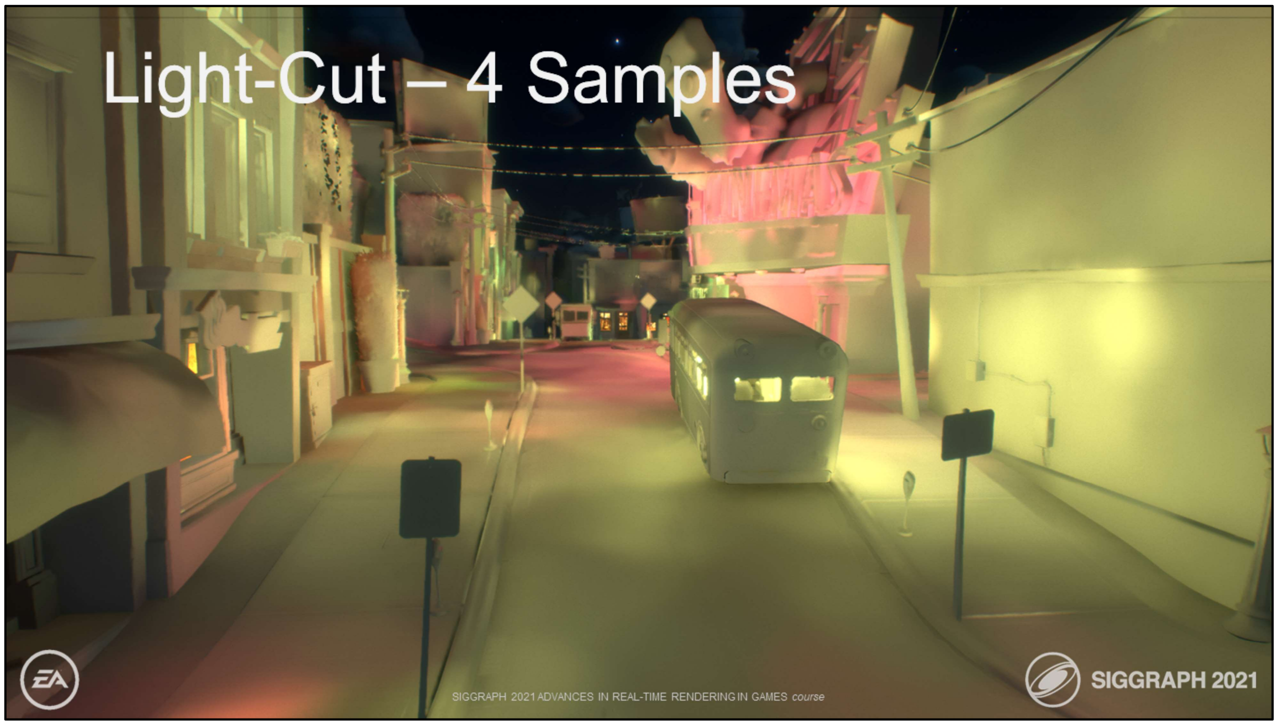
We can see this is very noisy. And we’re missing a ton of light contribution. We’re getting a lot of emissve contribution from the theater sign, but hardly any contribution from lights.

Reservoir – 8 Samples



SIGGRAPH 2021

Now we have 8 sample reservoir sampling. It’s getting much closer to our converged example, far outpacing random sampling. But it’s still quite noisy.



Light-Cut – 4 Samples

And finally, we have 4 sample light-cut sampling after 15 frames. This is much closer to our converged example, but also much more costly on console.

Converged

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*
SIGGRAPH 2021

And again here is the converged indirect diffuse. And I'll cycle between lightcuts after solving for only 15 frames and the converged example. We can see that it gets very close to the converged result after only 15 frames.

Transparency

So far we've describe our solution for opaque surfaces. Now, let's talk about how we support transparency.

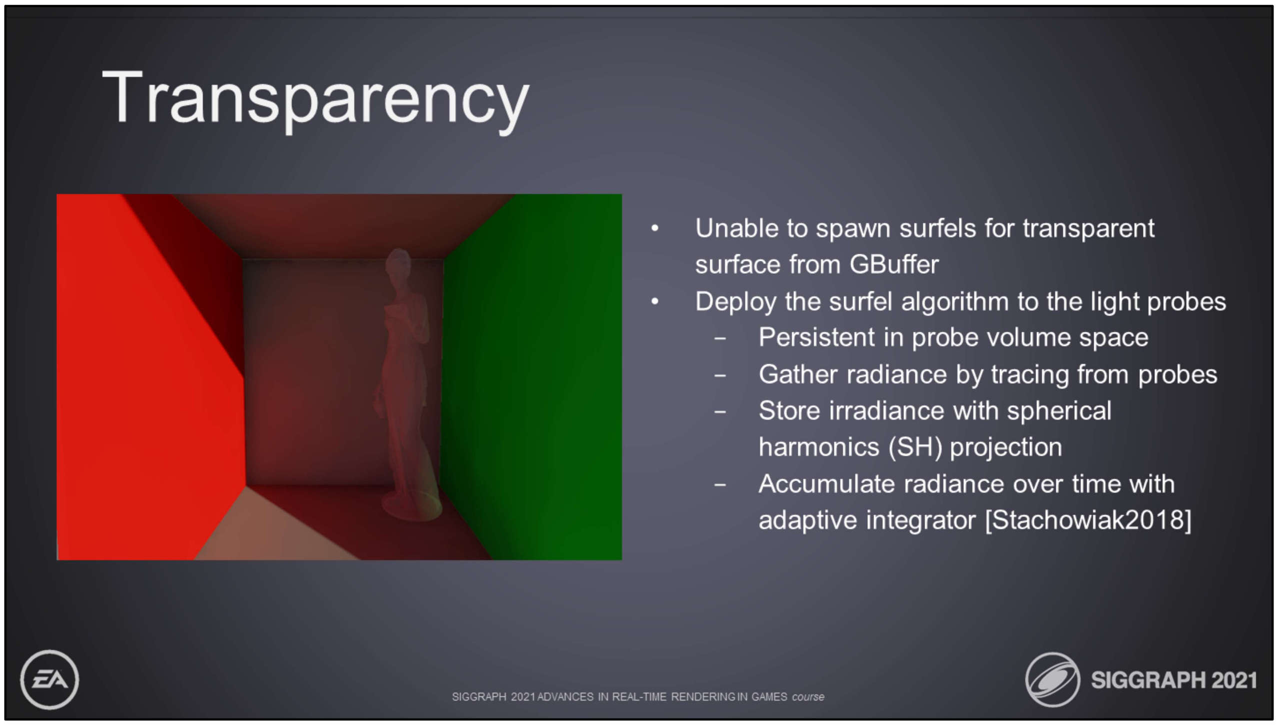
Lightcuts

SIGGRAPH 2021

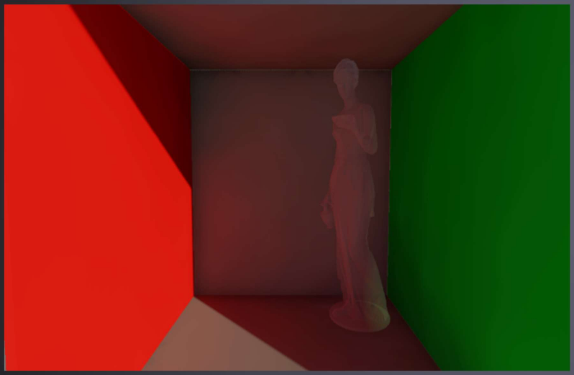
And again here is the converged indirect diffuse. And I'll cycle between lightcuts after solving for only 15 frames and the converged example. We can see that it gets very close to the converged result even after only 15 frames.

Transparency

But first, a gratuitous lighting shot showing off lighting for transparents… :-)



Transparency

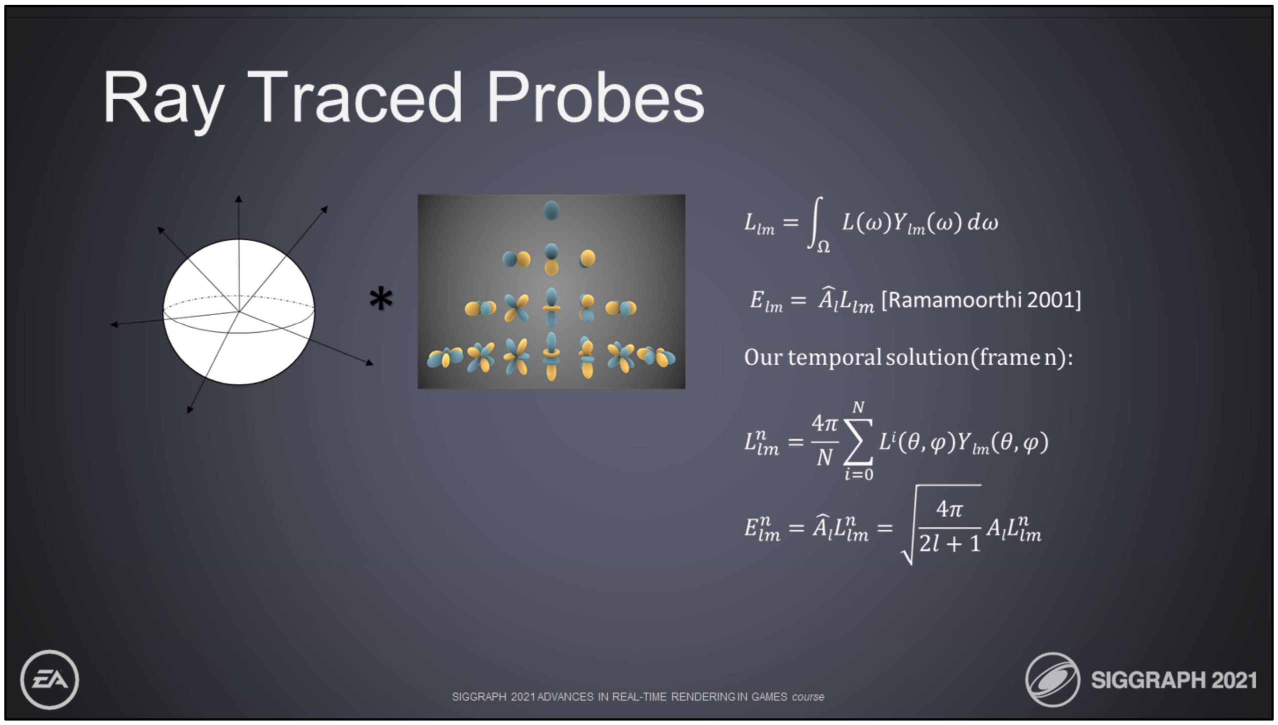


- Unable to spawn surfels for transparent surface from GBuffer
- Deploy the surfel algorithm to the light probes
  - Persistent in probe volume space
  - Gather radiance by tracing from probes
  - Store irradiance with spherical harmonics (SH) projection
  - Accumulate radiance over time with adaptive integrator [Stachowiak2018]

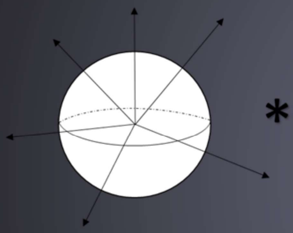
While the Surfels are perfectly fit for caching irradiance for opaque surfaces, it’s hard to port the same method to transparents because we rely on spawning surfels from the gbuffer.

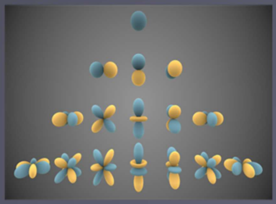
To address this issue, we apply the surfel algorithm for light probes. The ray traced probes will persist in a probe volume and gather radiance similarly to surfels. Because we are gathering the diffuse radiance, it gives us the opportunity to project probes to SH space.

In addition, we use the same adaptive integrator that we have discussed earlier with surfel integration, to accumulate probe samples across frames.



Ray Traced Probes





$$L_{lm} = \int_{\Omega} L(\omega) Y_{lm}(\omega)\, d\omega$$

$$E_{lm} = \hat{A}_l L_{lm}$$ [Ramamoorthi 2001]

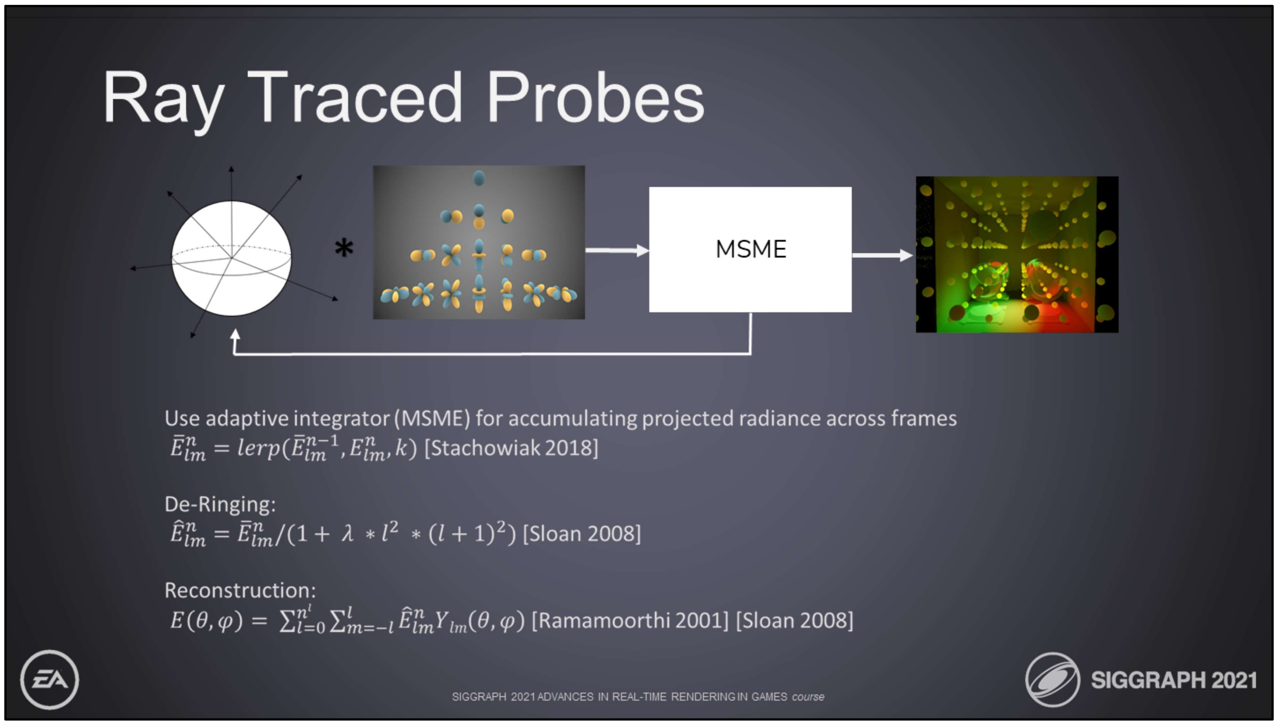
Our temporal solution(frame n):

$$L_{lm}^{n} = \frac{4\pi}{N} \sum_{i=0}^{N} L^{i}(\theta, \varphi) Y_{lm}(\theta, \varphi)$$

$$E_{lm}^{n} = \hat{A}_l L_{lm}^{n} = \sqrt{\frac{4\pi}{2l+1}} A_l L_{lm}^{n}$$

As we can see it’s hard to compute the integral for the SH coefficients in a single frame at real-time, which needs a large amount of samples to converge.

We amortize the calculation across frames: the irradiance SH coefficients on every frame, are calculated by sampling incident radiance with a couple of rays and projecting to SH space.



Ray Traced Probes

Use adaptive integrator (MSME) for accumulating projected radiance across frames
$\bar{E}^{n}_{lm} = lerp(\bar{E}^{n-1}_{lm}, E^{n}_{lm}, k)$ [Stachowiak 2018]

De-Ringing:
$\hat{E}^{n}_{lm} = \bar{E}^{n}_{lm}/(1 + \lambda * l^2 * (l+1)^2)$ [Sloan 2008]

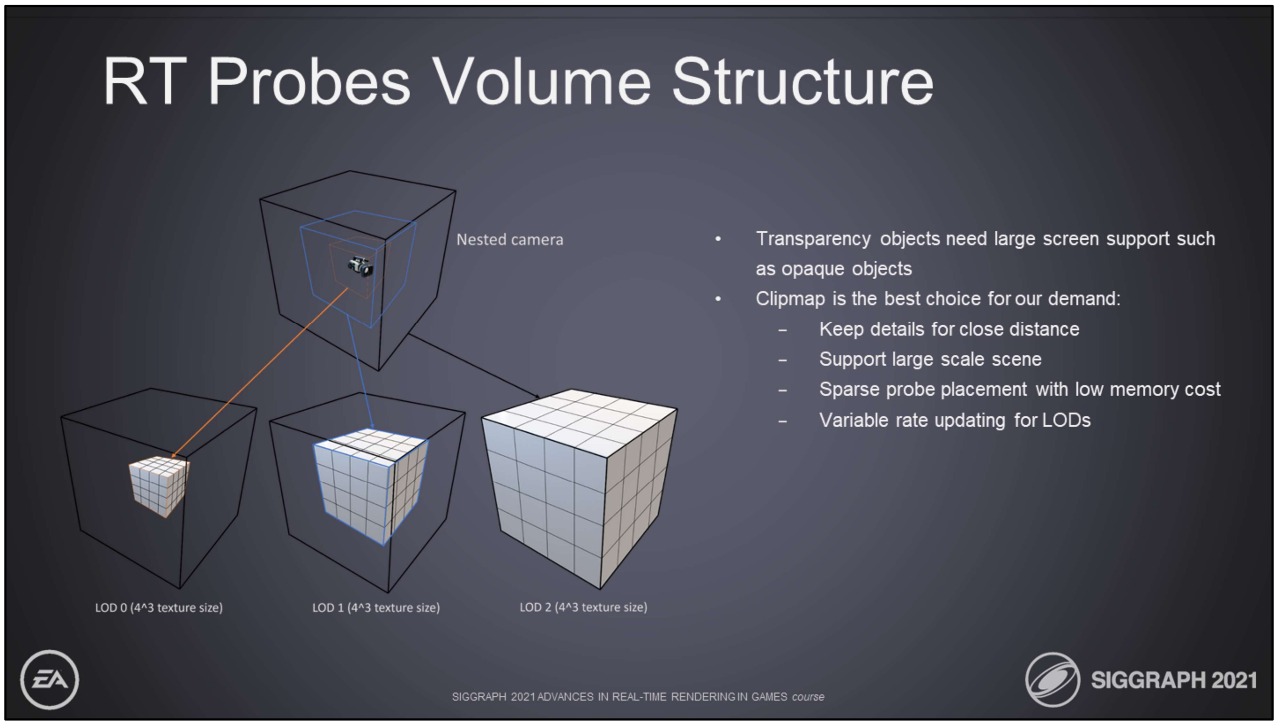
Reconstruction:
$E(\theta, \varphi) = \sum_{l=0}^{n^l} \sum_{m=-l}^{l} \hat{E}^{n}_{lm} Y_{lm}(\theta, \varphi)$ [Ramamoorthi 2001] [Sloan 2008]

Then we use the same adaptive MSME integrator, as Surfels to accumulate the projected radiance across frames.

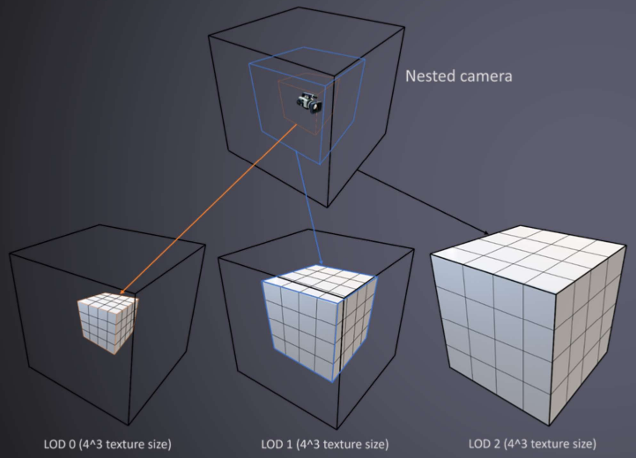
The integrator will also calculate the number of rays to shoot in the next frame, depending on variance.

We also employ Sloan’s de-ringing windowing function to avoid the unwanted dark and bright banding that occurs when there is high variance in lighting values.

And finally, we store the filtered results into a probe volume, from which we can sample and do irradiance reconstruction for shading.



RT Probes Volume Structure



- Transparency objects need large screen support such as opaque objects
- Clipmap is the best choice for our demand:
  - Keep details for close distance
  - Support large scale scene
  - Sparse probe placement with low memory cost
  - Variable rate updating for LODs

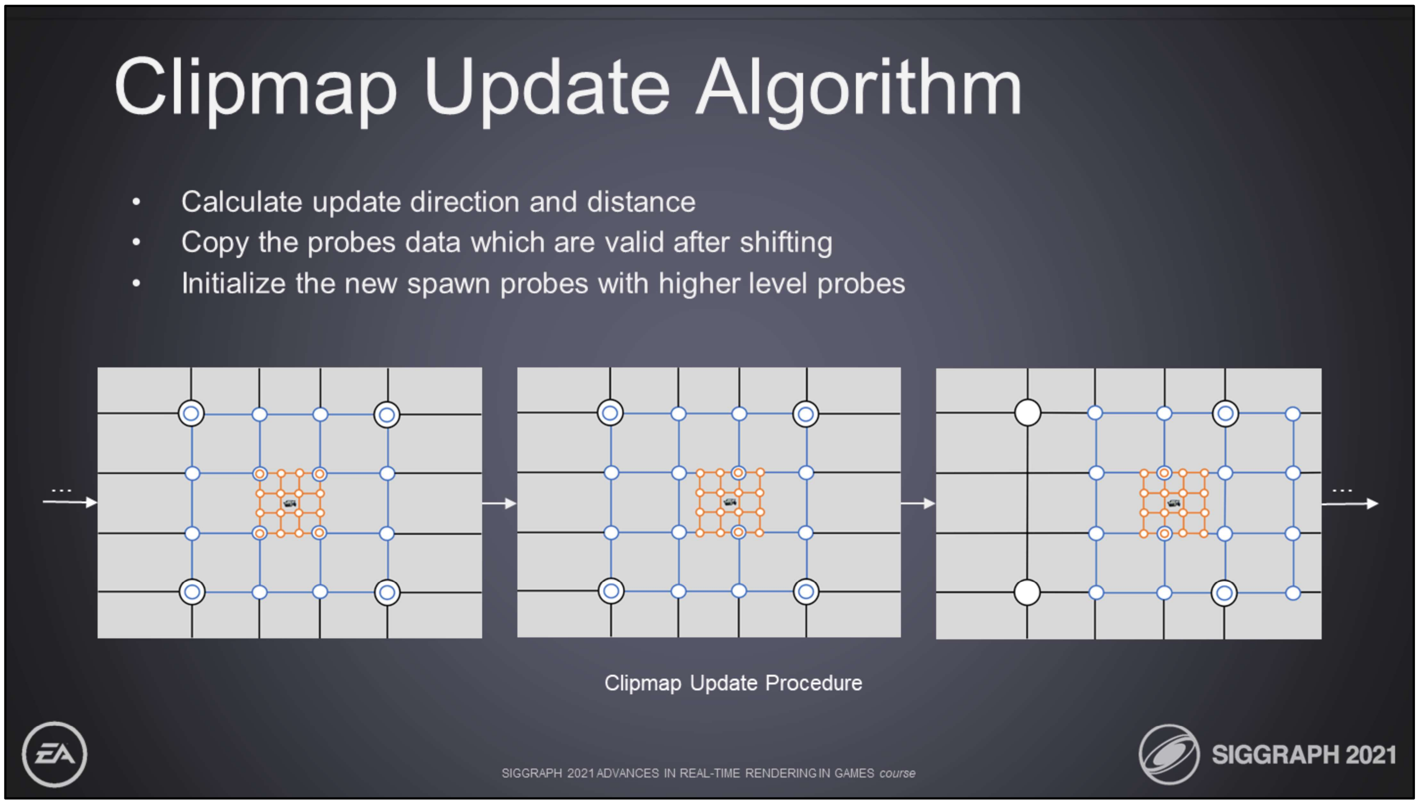
The issue of scale is a problem for light probes, just as it is for surfels. We investigated a few different schemes, but settled on volume clipmaps.

These are a set of nested volumes, where each level represents larger and larger area but less detail.

This structure enables relatively high detail close to the camera, while still covering a large amount of area, for a low memory cost.

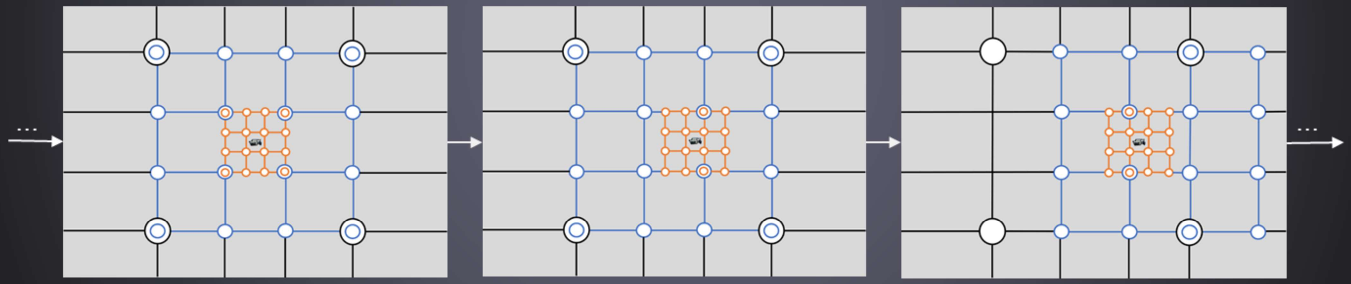
We can also update the structure out of sync, and use lower detail levels to prime higher detail levels.

Now, let’s take a closer look at how we update the clipmap when camera moves.



Clipmap Update Algorithm

- Calculate update direction and distance
- Copy the probes data which are valid after shifting
- Initialize the new spawn probes with higher level probes



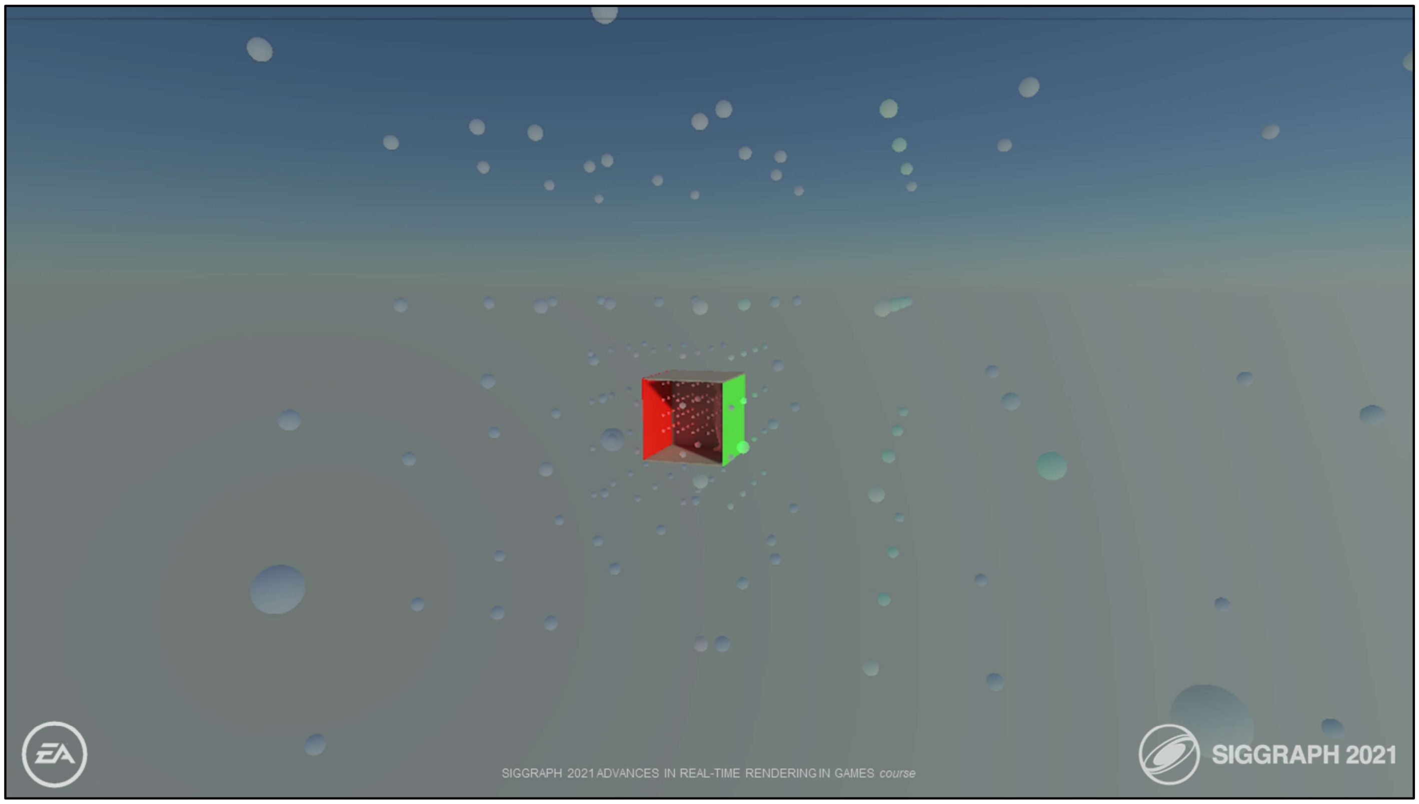
Clipmap Update Procedure

First, we check the world space position of the camera every frame. If the camera moves outside of the center grid of any levels, we shift the probes of these levels towards the camera to keep it in the level’s center. The distance of shifting depends on how many grid cells the camera has moved away from the center.

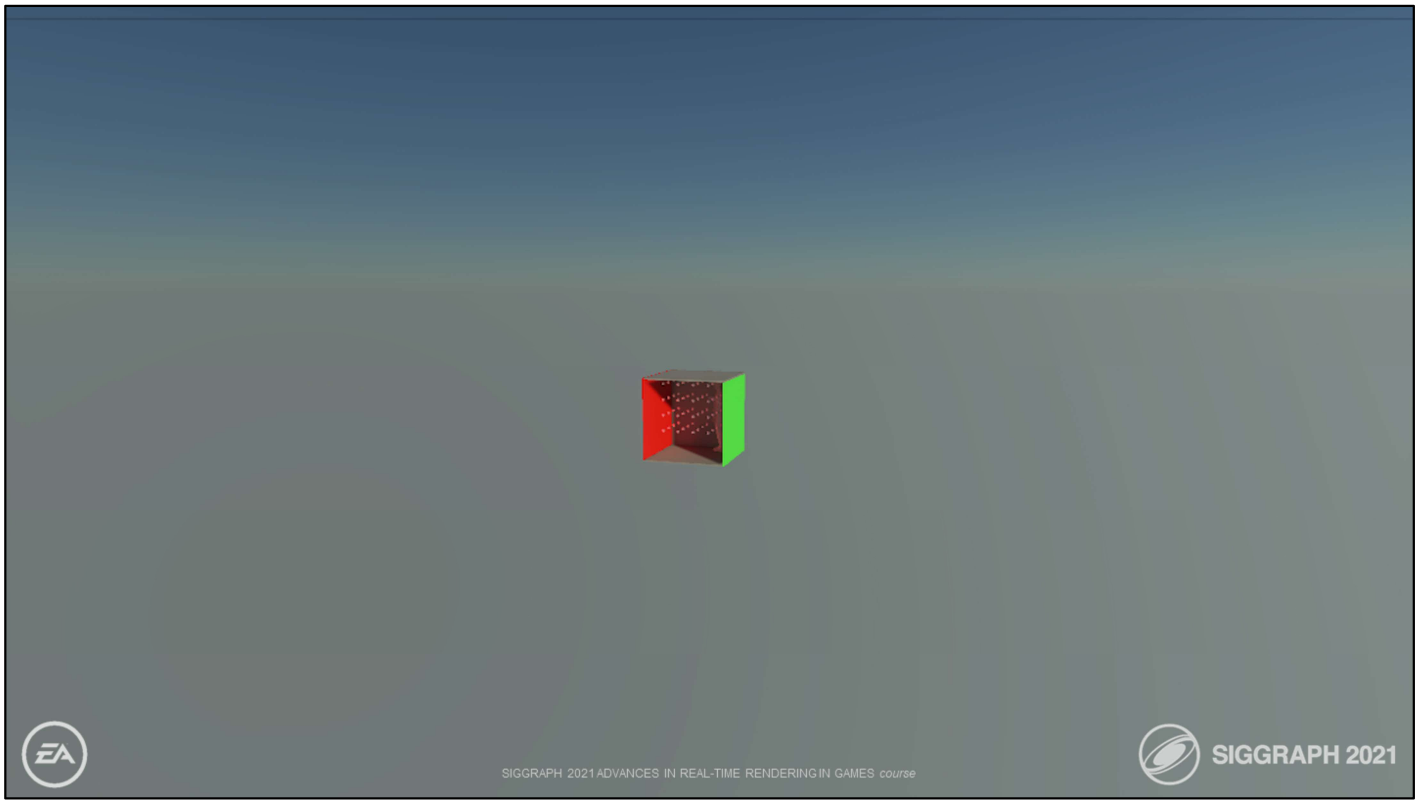
After the shifting, there will be new probes which are spawned in the updated clipmap, and they will be initialized with higher level probes using interpolation.

Because the hierarchical grid covers a large area in worldspace, most probes will still be valid after updating, which means they only need to be copied to their new coordinate in probe volume space, instead of wasteful discarding. This is the key to keep the stored irradiance valid for as long as possible and improve performance.

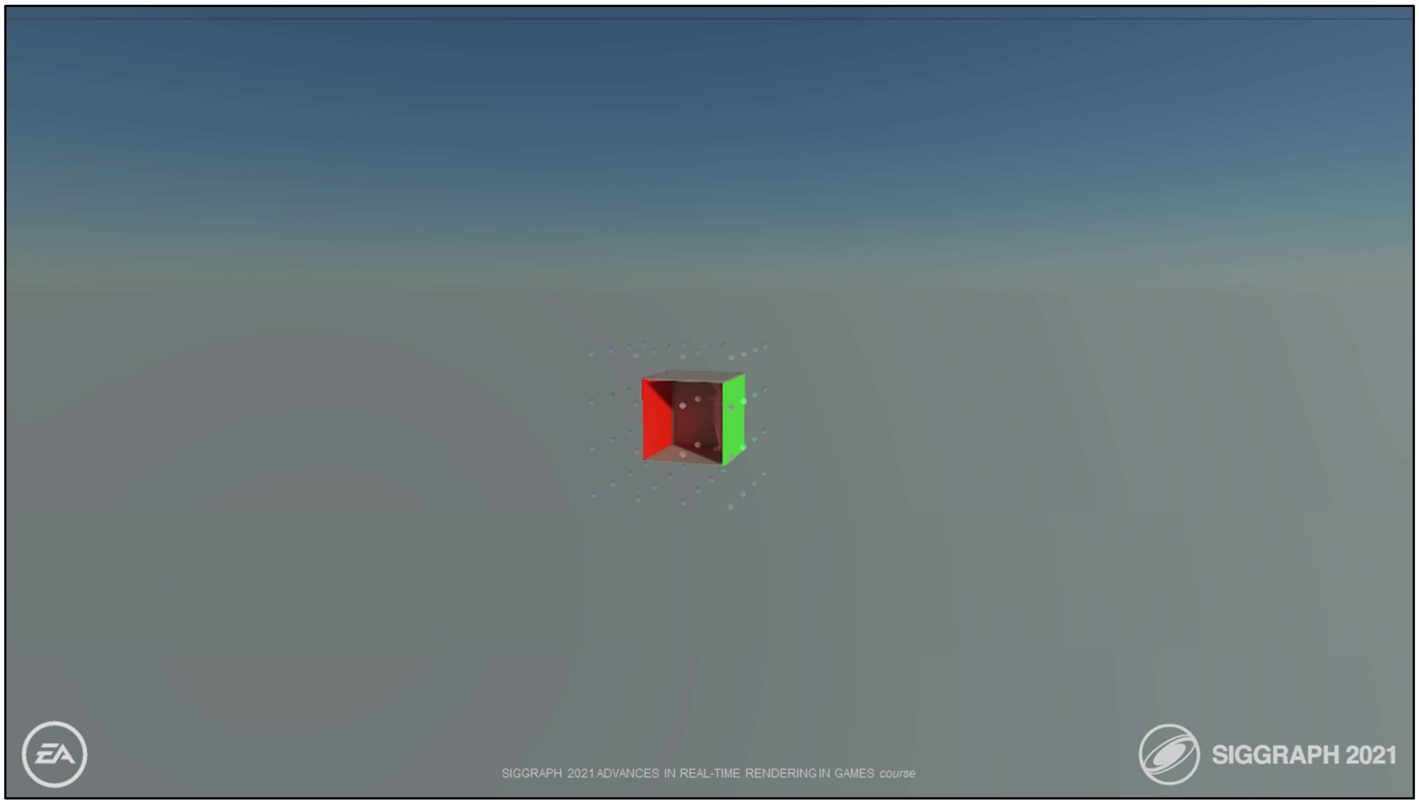
Here’s a demonstration of the probe volume updating its coordinates as it follows the camera


SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES course
SIGGRAPH 2021

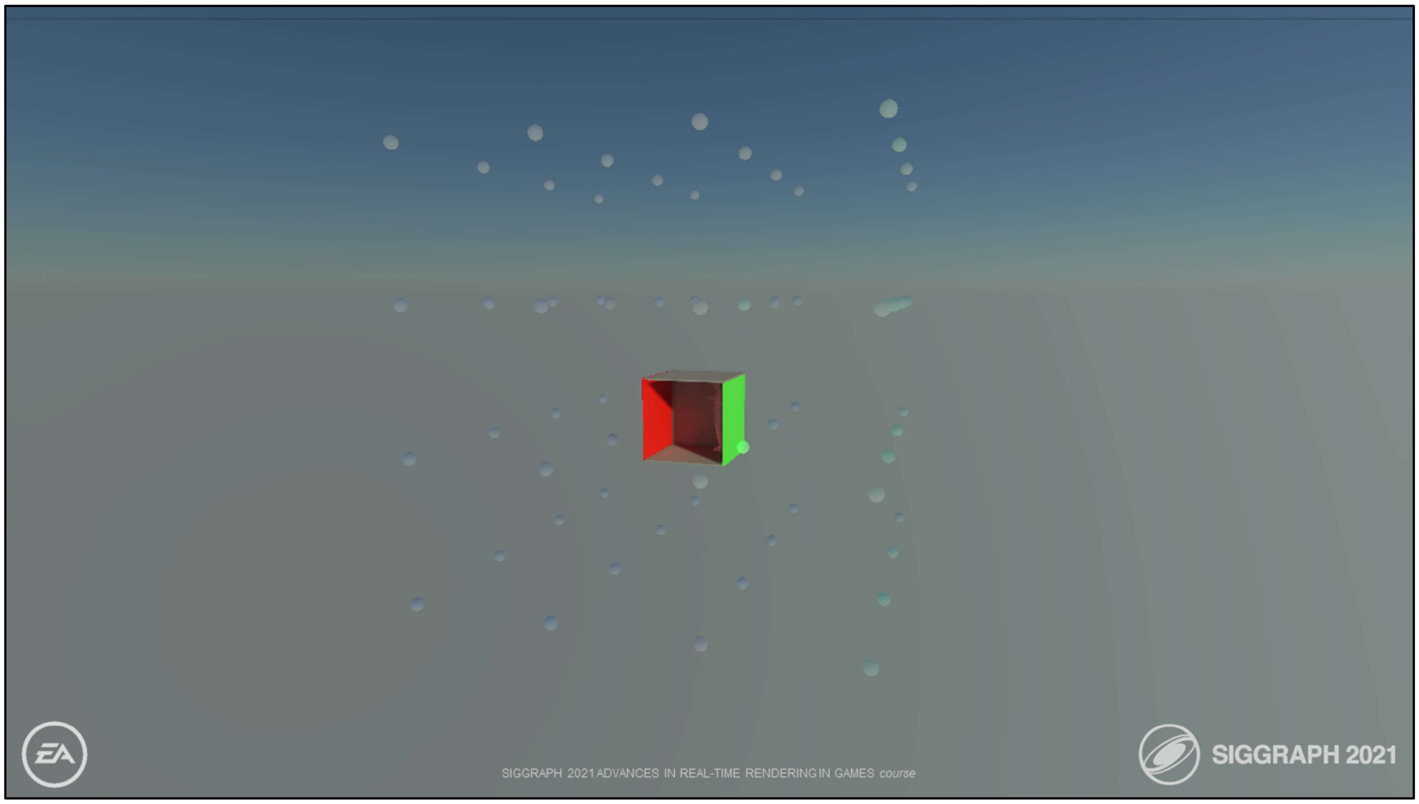
Now I'll show how the clipmaps are set up spatially. Here we have all 4 levels.



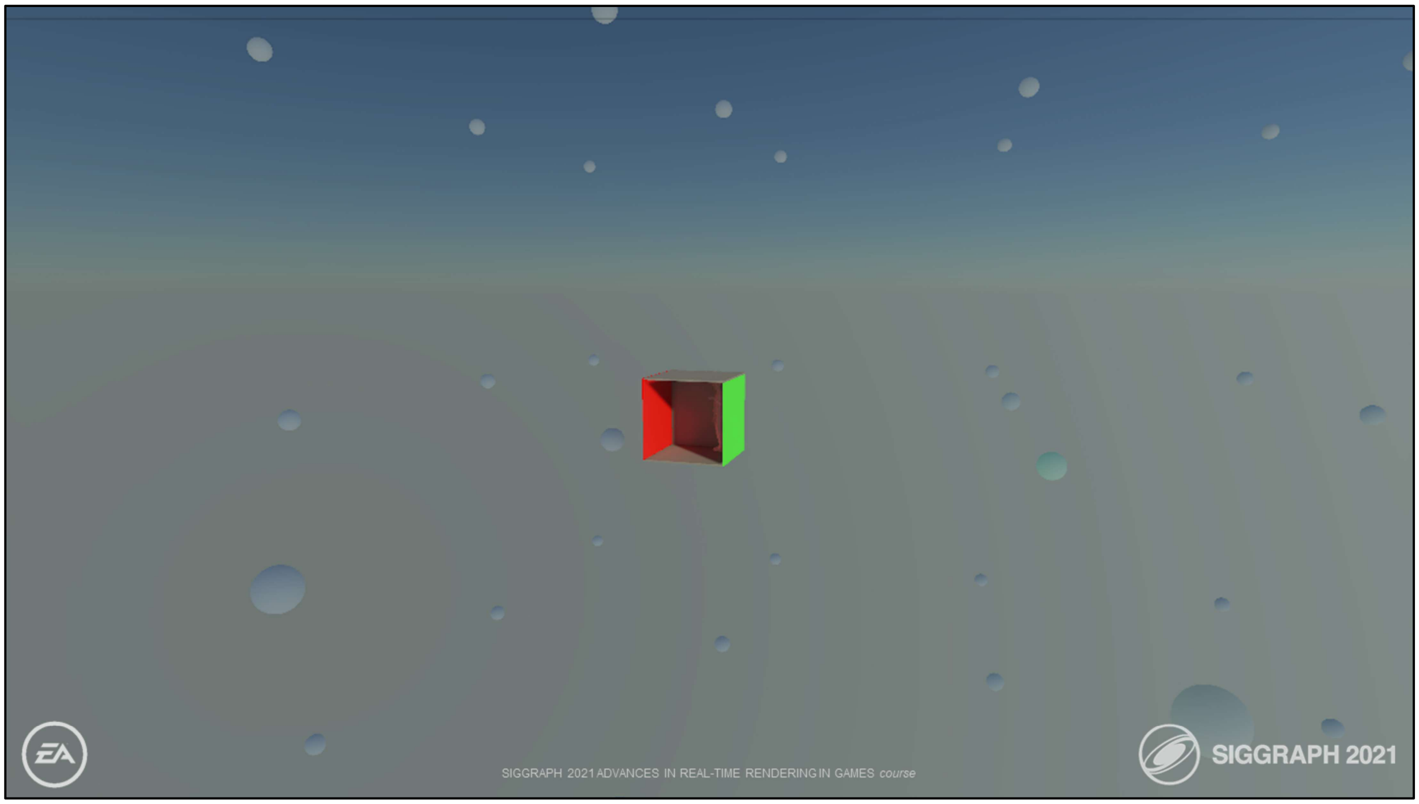
And now just the first level



The second level



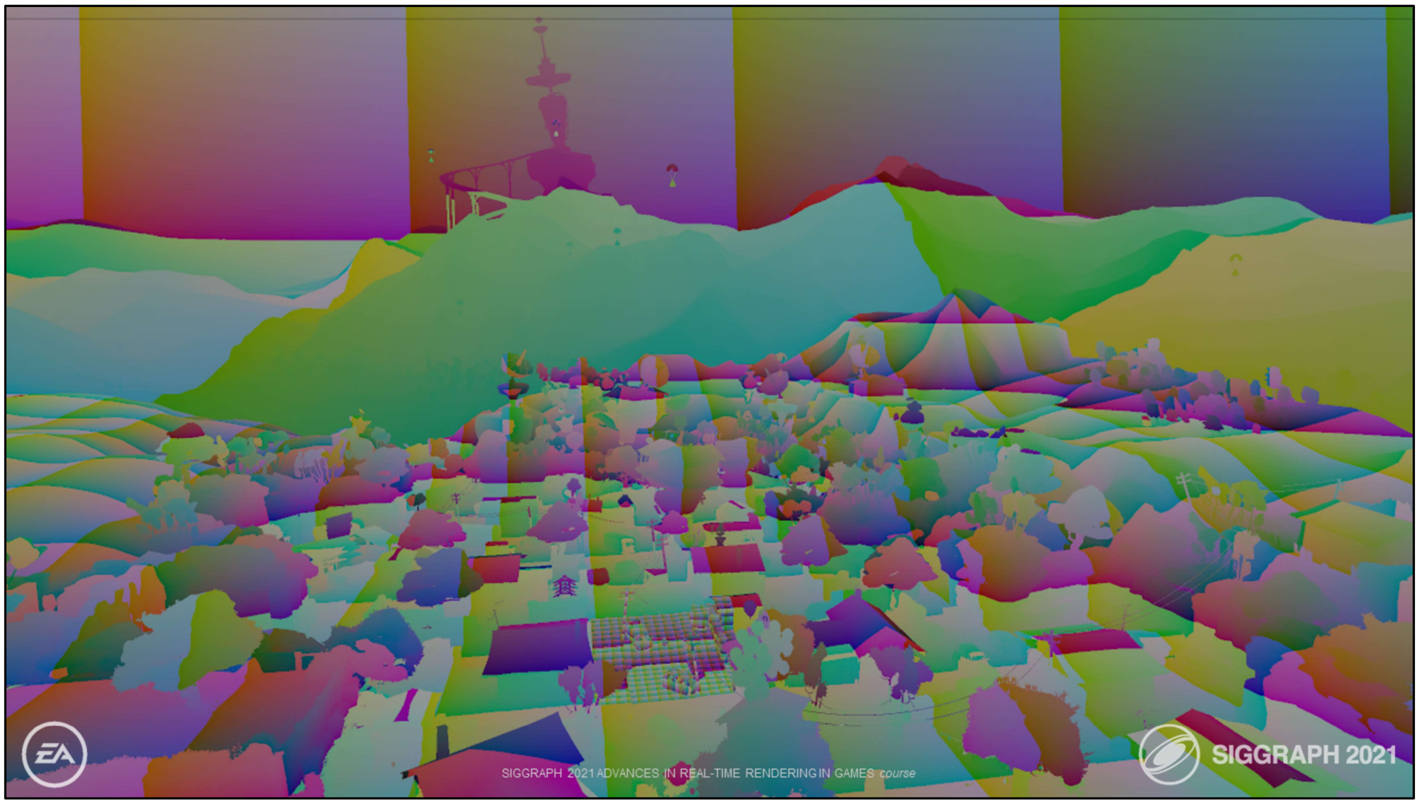
The third


SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES course
SIGGRAPH 2021

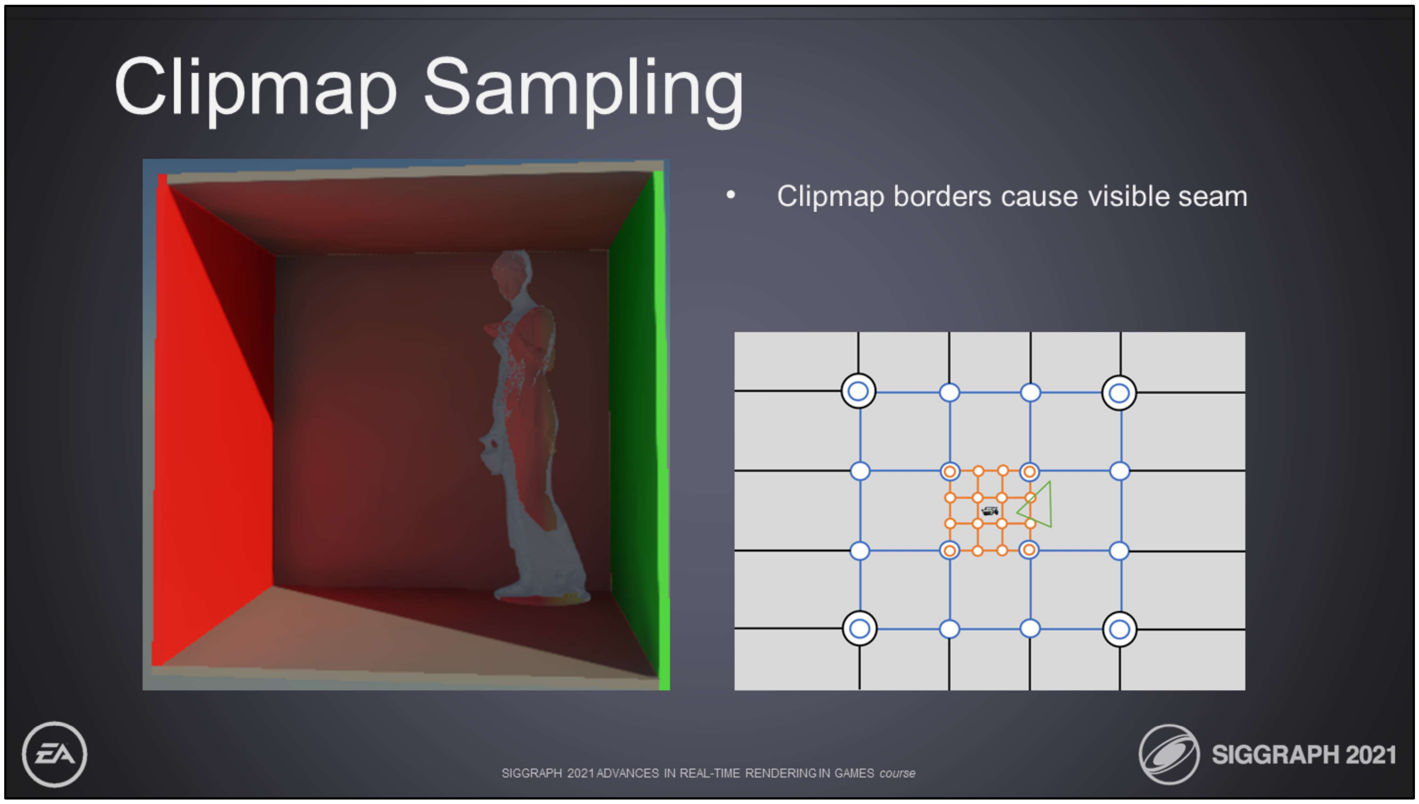
And, the fourth

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES course
SIGGRAPH 2021

Here's towncenter again, from PvZ. Where we'll show the debug view, showing which pixels are sampling which level.

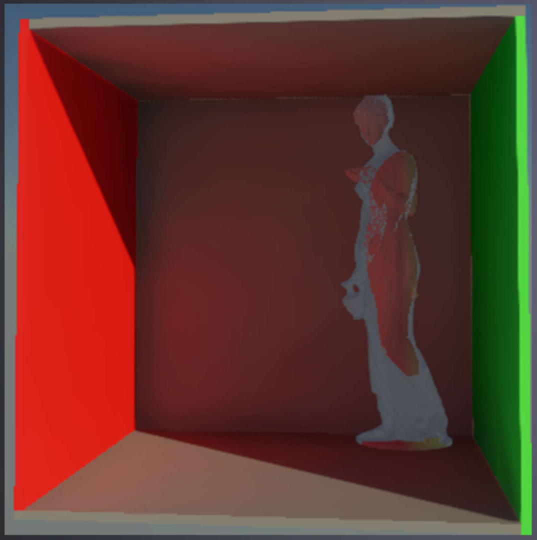

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES course
SIGGRAPH 2021

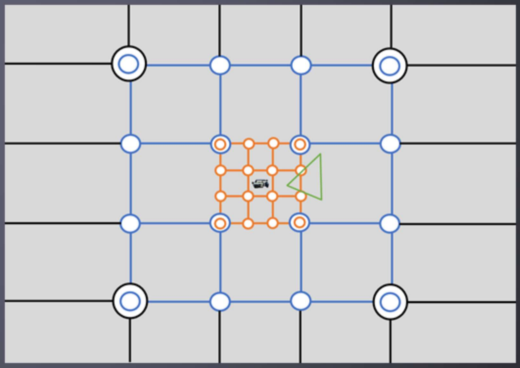
We've frozen the clip-map camera and we can see the clip-map hierarchy expanding outward from the bottom-center of the screen.



Clipmap Sampling

- Clipmap borders cause visible seam



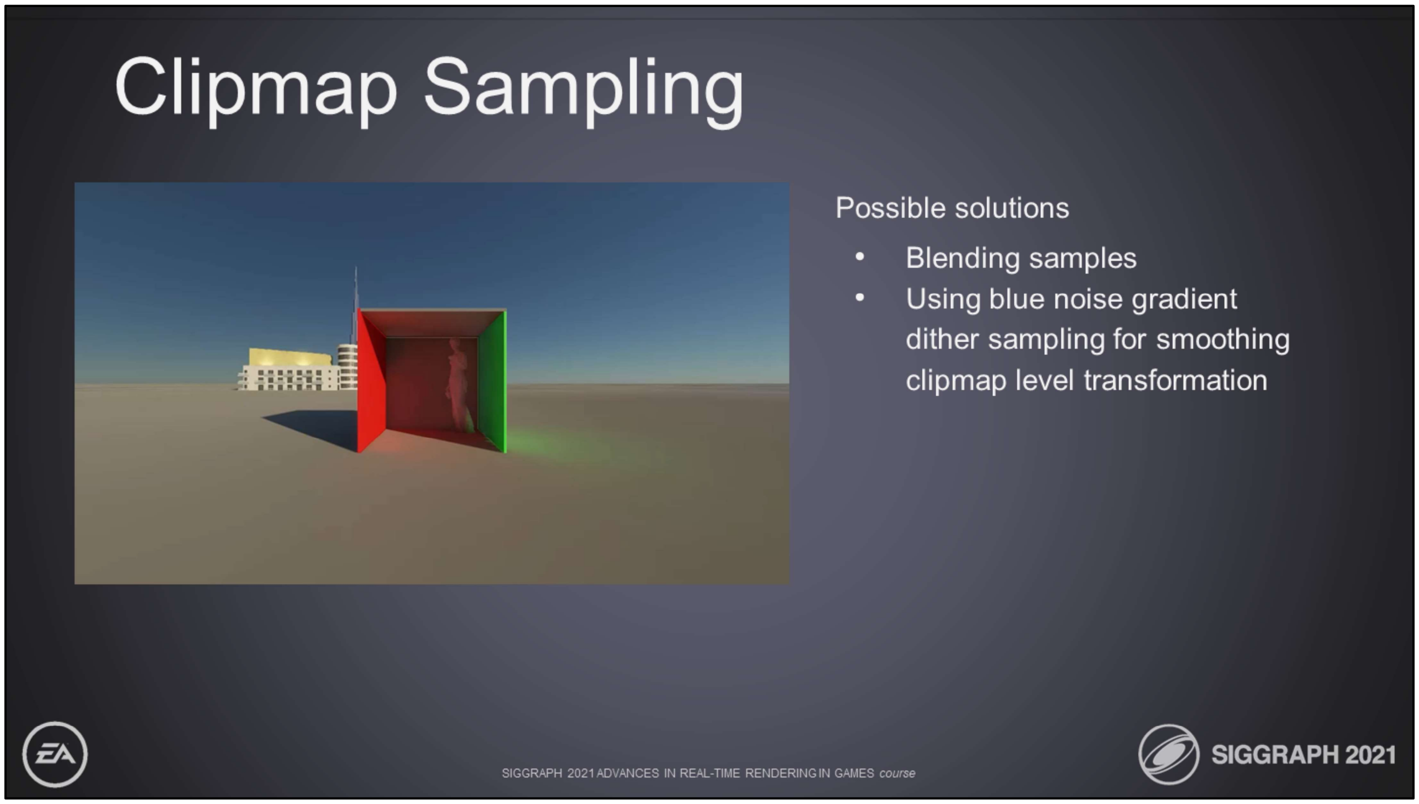


SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

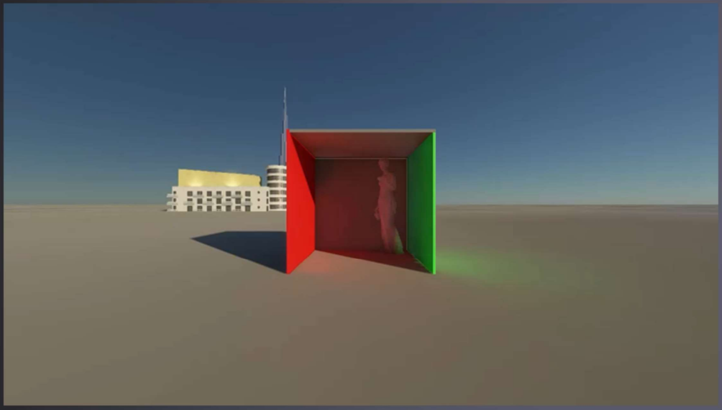
SIGGRAPH 2021

Sampling the clipmap is straightforward: find the highest detailed level that contains the shaded pixel, then sample the SH coefficients from the 3D clipmap texture and reconstruct irradiance.

However, this naive sampling strategy will cause visible discontinuities on borders. As we can see here with the statue.



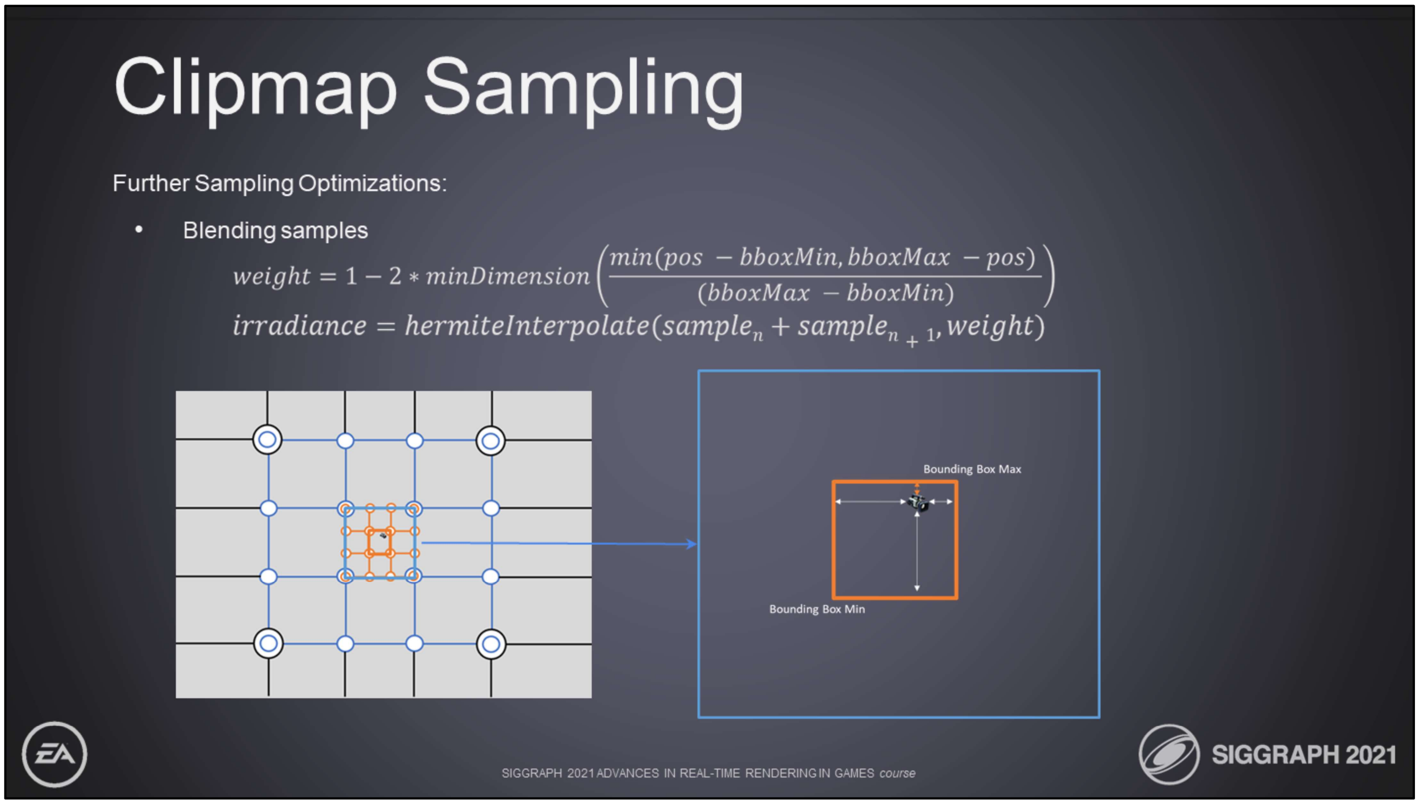
Clipmap Sampling



Possible solutions

- Blending samples
- Using blue noise gradient dither sampling for smoothing clipmap level transformation

This is perhaps even more of a problem for animated scenes. You can see the statue pop to the lower detail level



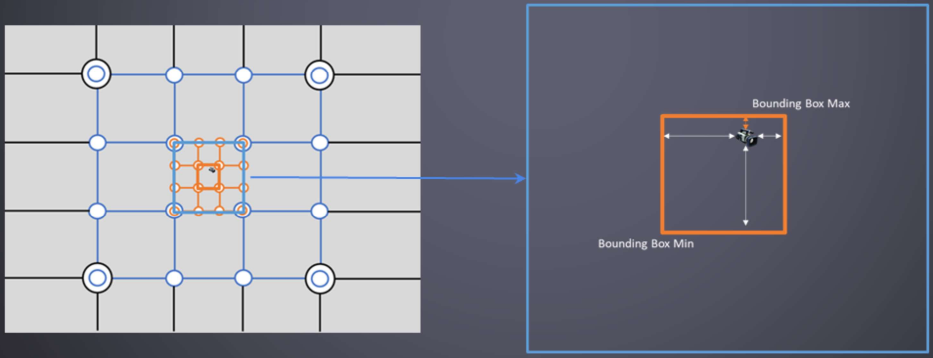
Clipmap Sampling

Further Sampling Optimizations:

- Blending samples

$$weight = 1 - 2 * minDimension\left(\frac{min(pos - bboxMin, bboxMax - pos)}{(bboxMax - bboxMin)}\right)$$

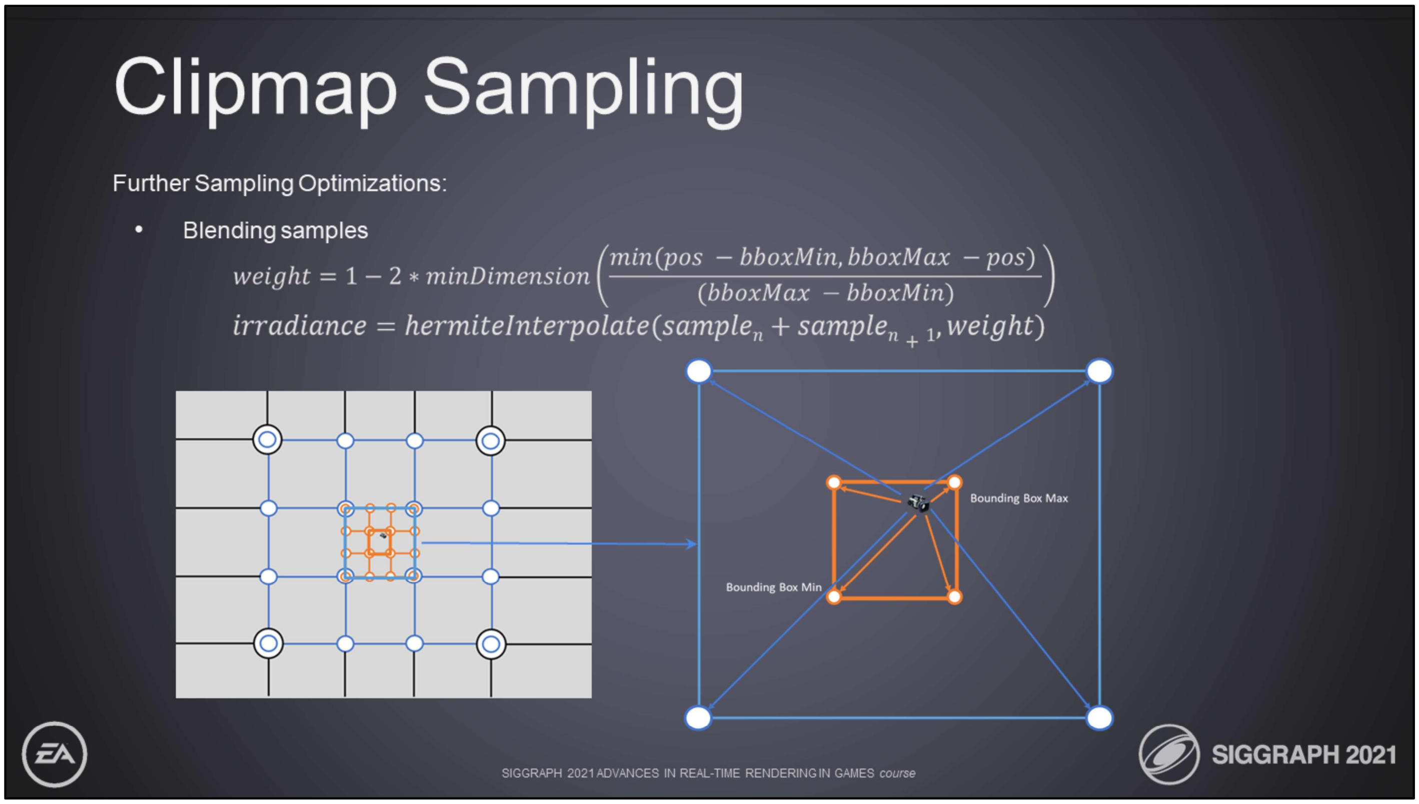
$$irradiance = hermiteInterpolate(sample_n + sample_{n+1}, weight)$$



SIGGRAPH 2021

A simple solution is to blend samples from the 2 closest levels.

We create a transition border for each level. Pixels lying in this border will sample from the current level and the next lower or higher detailed level. We calculate the blend weight based on the normalized distance to the border.



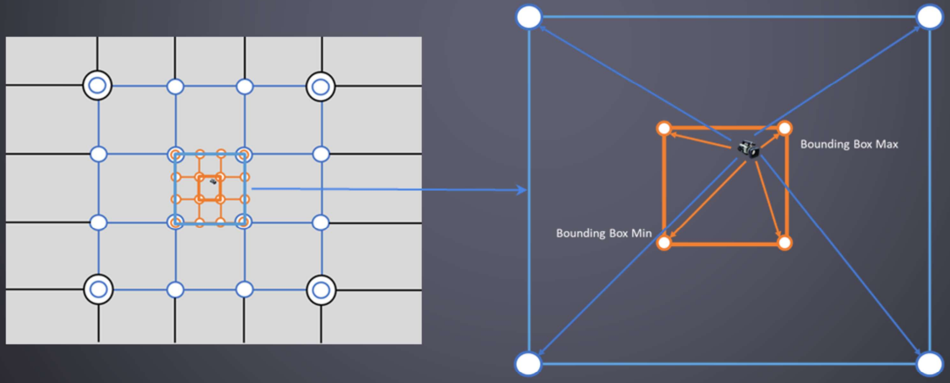
Clipmap Sampling

Further Sampling Optimizations:

- Blending samples

$$weight = 1 - 2 * minDimension\left(\frac{min(pos\ - bboxMin, bboxMax\ - pos)}{(bboxMax\ - bboxMin)}\right)$$

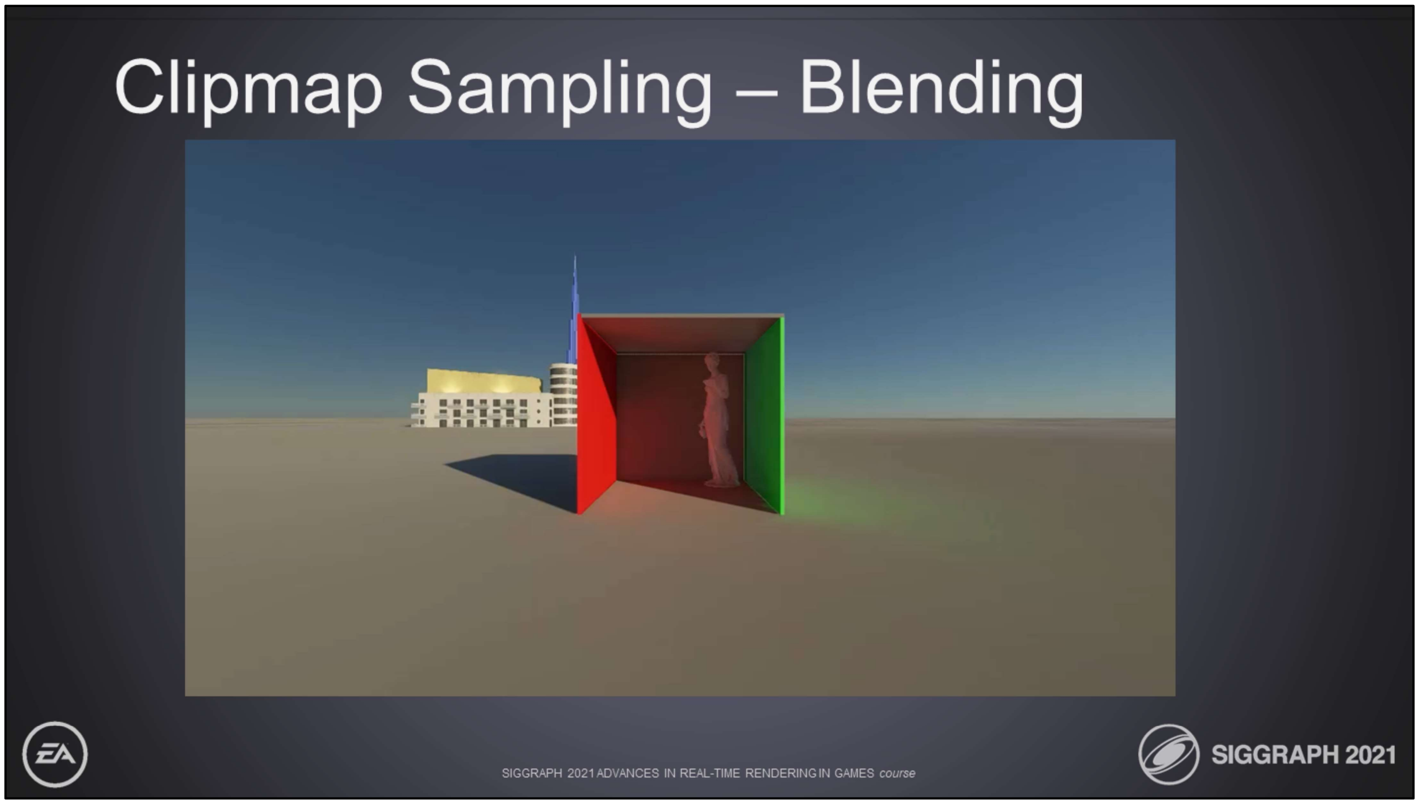
$$irradiance = hermiteInterpolate(sample_n + sample_{n\ +\ 1}, weight)$$



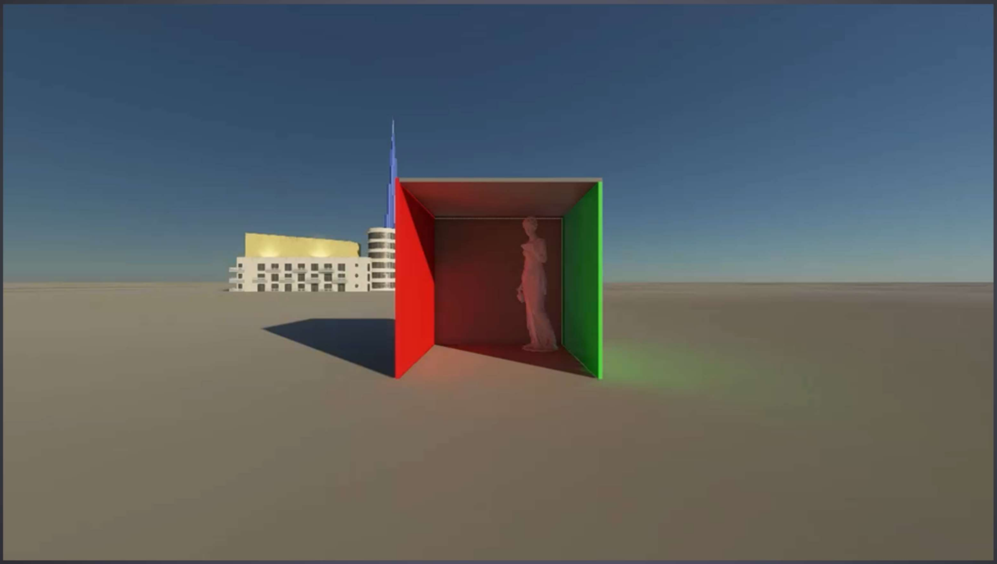
SIGGRAPH 2021

A simple solution is to blend samples from the 2 closest levels.

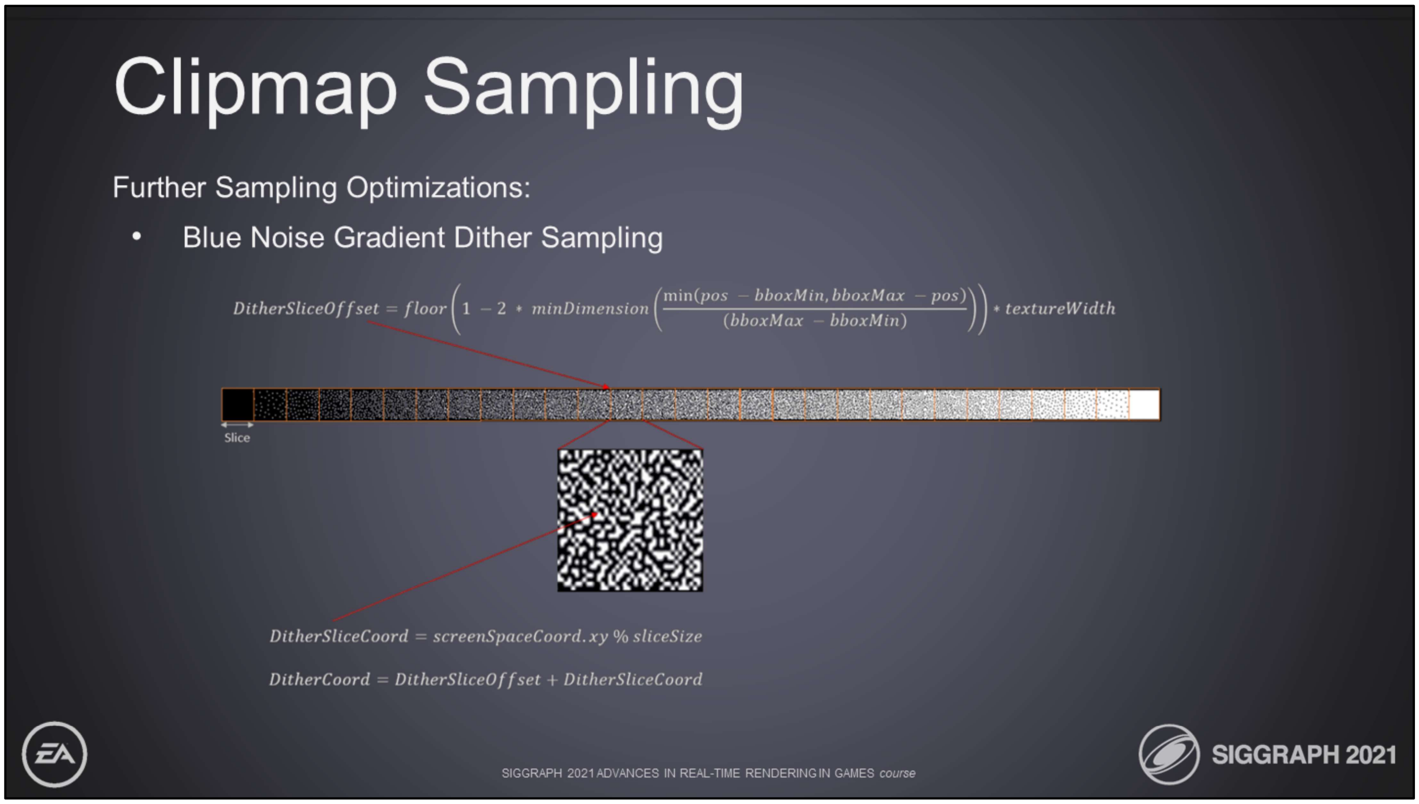
We create a transition border for each level. Pixels lying in this border will sample from the current level and the next lower or higher detailed level. We calculate the blend weight based on the normalized distance to the border.



Clipmap Sampling – Blending



Now with blending enabled, as we back the camera out we have a smooth transition.



Clipmap Sampling

Further Sampling Optimizations:

- Blue Noise Gradient Dither Sampling

$$DitherSliceOffset = floor\left(1 - 2 * minDimension\left(\frac{\min(pos - bboxMin, bboxMax - pos)}{(bboxMax - bboxMin)}\right)\right) * textureWidth$$

$$DitherSliceCoord = screenSpaceCoord.xy \,\%\, sliceSize$$

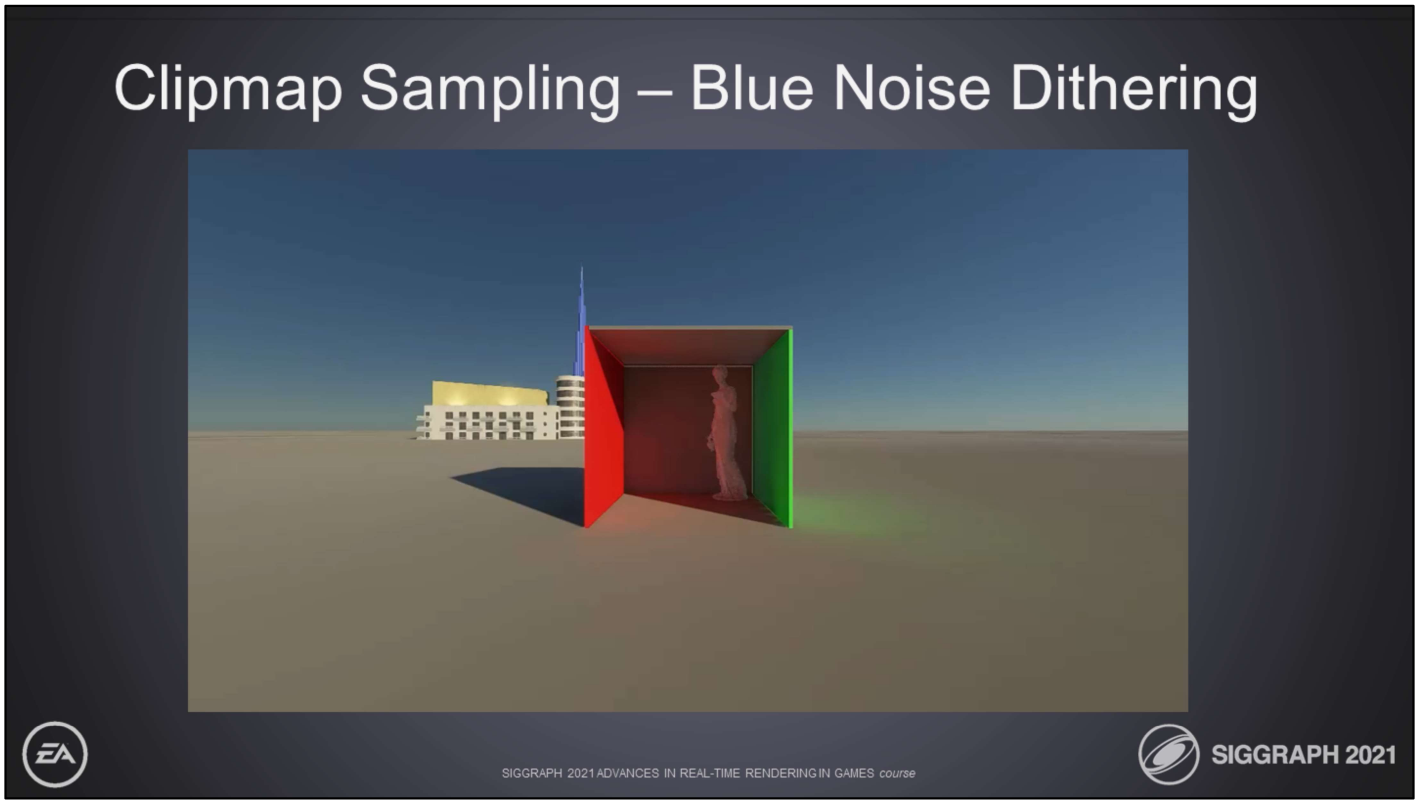
$$DitherCoord = DitherSliceOffset + DitherSliceCoord$$

But, blending has the extra cost of sampling 2 volumes per a shaded sample.

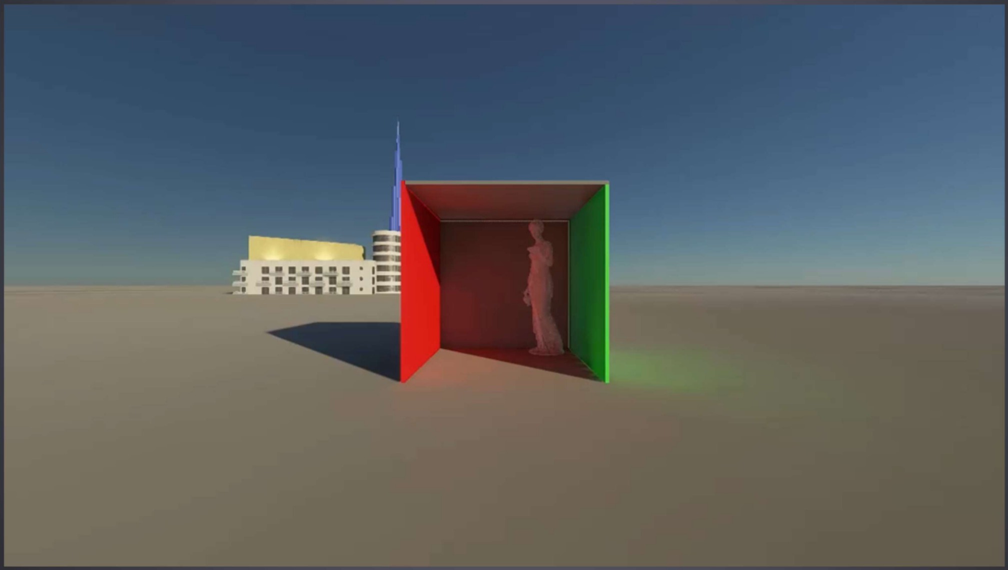
Our goal is to sample only one volume while maintaining a similar quality to blending. Introducing, blue noise dithered sampling.

Blue noise is a great, low-discrepancy distribution, which is easily filterable and works well with our TAA.

The blue noise dithering is similar to the blending solution, we sample from the current level or the next level based on the screen space sampled blue-noise value. Resulting in a quality similar to blending.

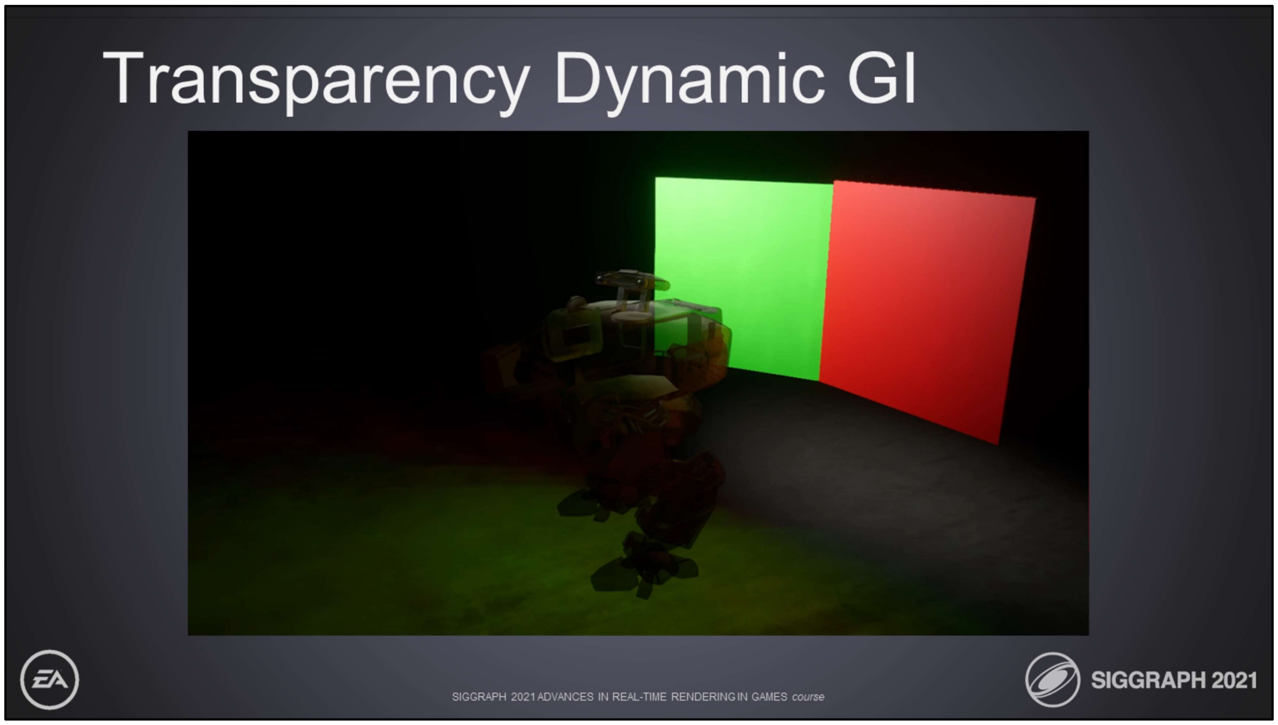


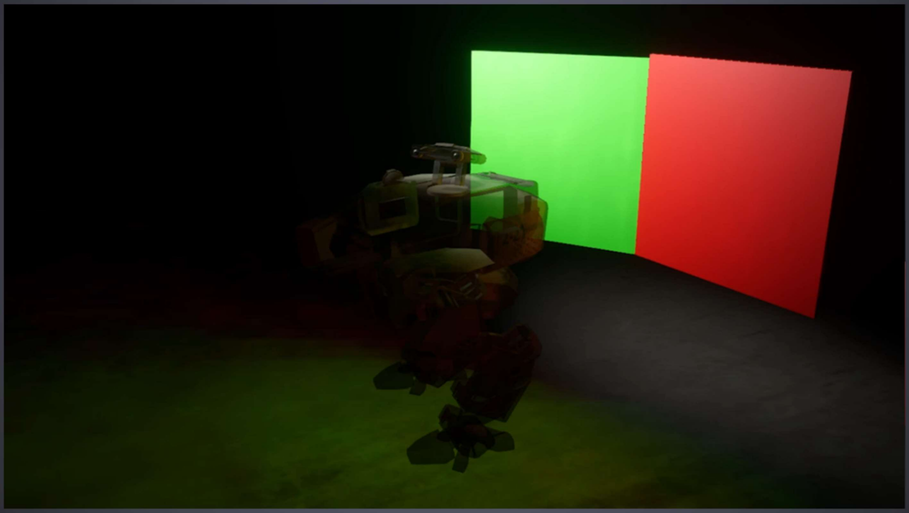
Clipmap Sampling – Blue Noise Dithering



SIGGRAPH 2021

So as we move the camera we can see a smooth transition, albeit with not quite as high quality as with blending.

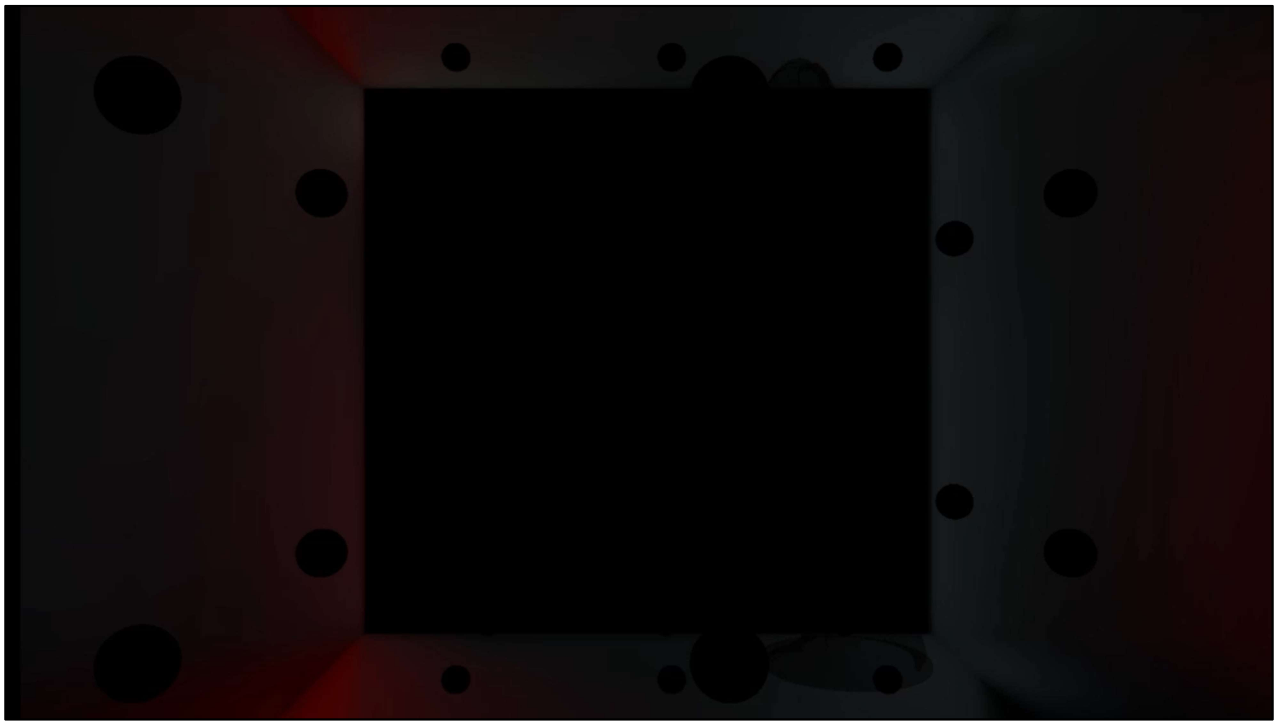

Transparency Dynamic GI



SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

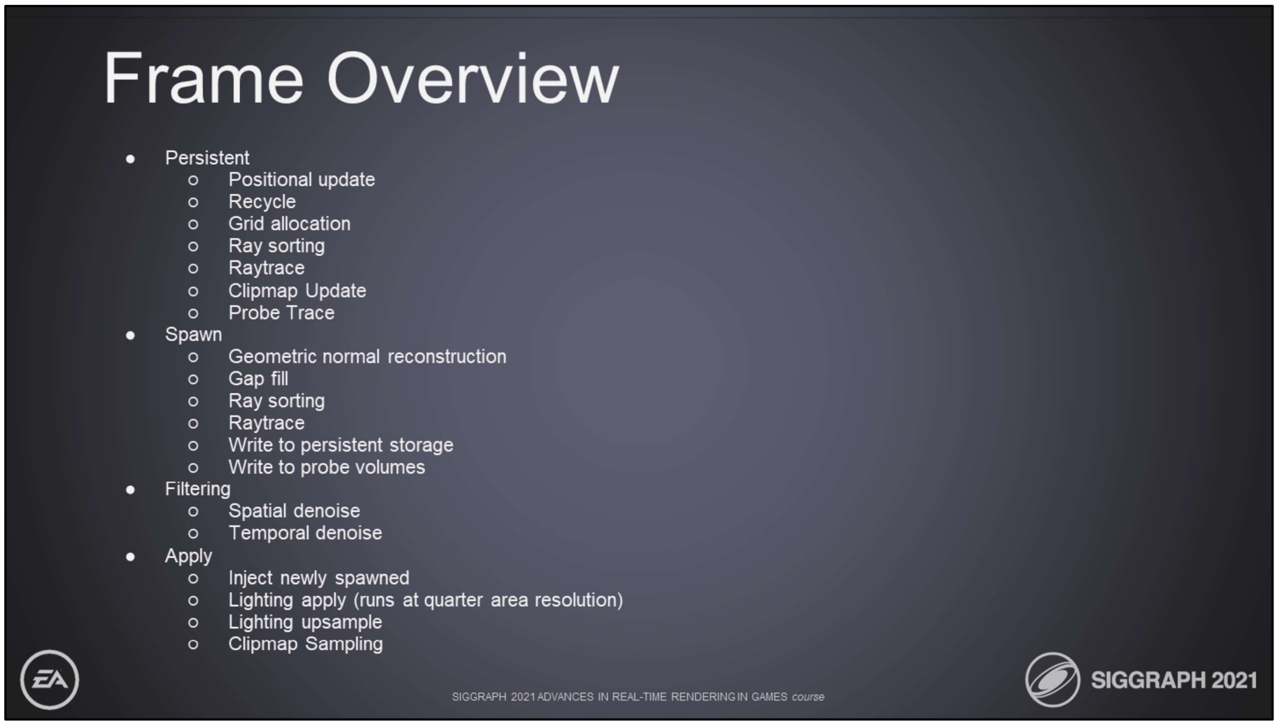
Here’s an example to show how our solution for transparent objects works well under a fully dynamic world. No pre-computations are needed, and the probes will automatically update when the scene geometry or lighting changes. But at the same time, we quickly converge to the new environment.



Performance

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES course
SIGGRAPH 2021

Let’s talk about where we’re at with performance of GIBS today. I’ll go over our general frame structure, and then give some performance examples of our worst case scenarios, followed by scenarios that are more representative of game content.

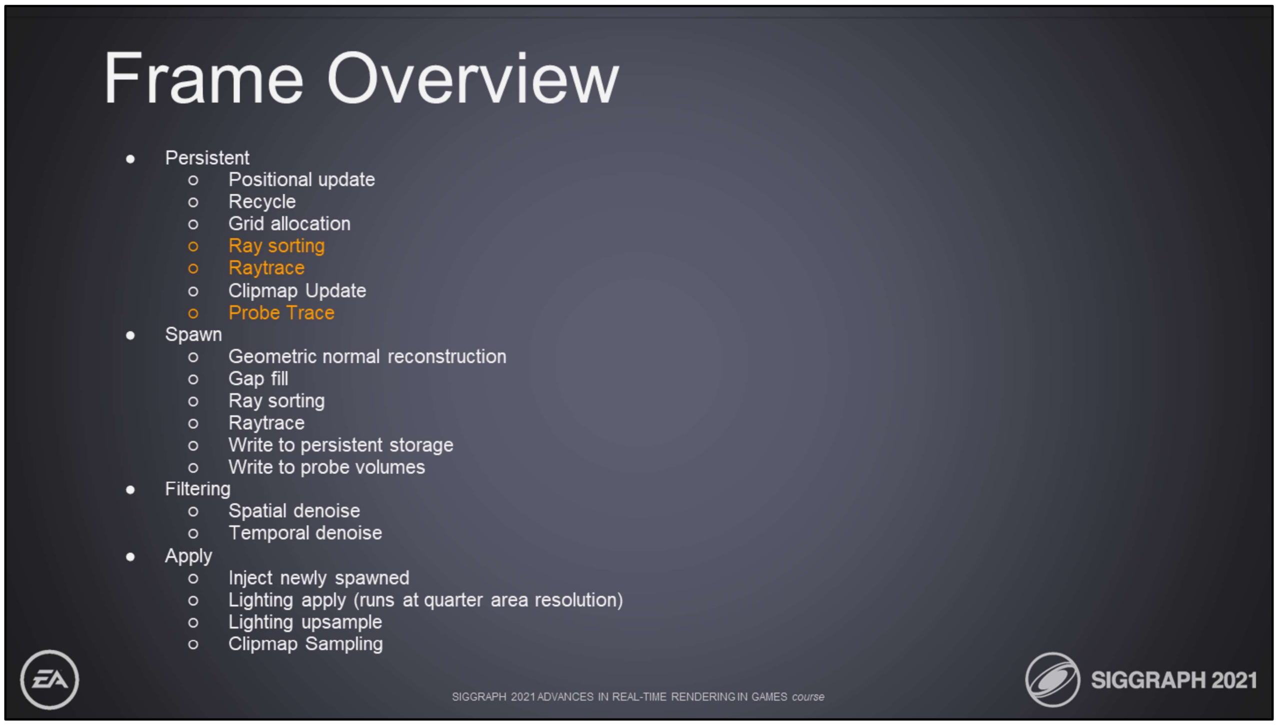


Frame Overview

- Persistent
  - Positional update
  - Recycle
  - Grid allocation
  - Ray sorting
  - Raytrace
  - Clipmap Update
  - Probe Trace
- Spawn
  - Geometric normal reconstruction
  - Gap fill
  - Ray sorting
  - Raytrace
  - Write to persistent storage
  - Write to probe volumes
- Filtering
  - Spatial denoise
  - Temporal denoise
- Apply
  - Inject newly spawned
  - Lighting apply (runs at quarter area resolution)
  - Lighting upsample
  - Clipmap Sampling

SIGGRAPH 2021

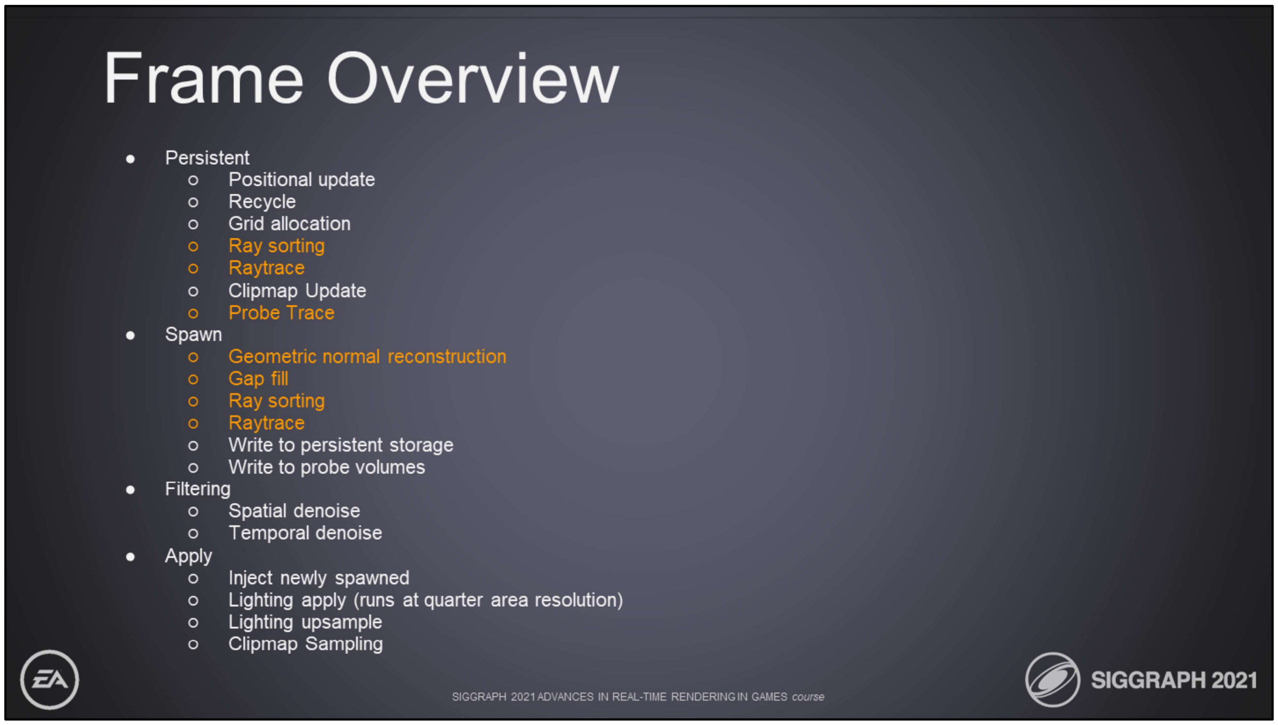
Here’s a general overview of our frame that we’ve detailed until now. There are 4 general sections, persistent surfel work, spawned surfel work, filtering surfels, and then finally applying the surfels to the screen.



Frame Overview

- Persistent
  - Positional update
  - Recycle
  - Grid allocation
  - Ray sorting
  - Raytrace
  - Clipmap Update
  - Probe Trace
- Spawn
  - Geometric normal reconstruction
  - Gap fill
  - Ray sorting
  - Raytrace
  - Write to persistent storage
  - Write to probe volumes
- Filtering
  - Spatial denoise
  - Temporal denoise
- Apply
  - Inject newly spawned
  - Lighting apply (runs at quarter area resolution)
  - Lighting upsample
  - Clipmap Sampling

Generally the raytracing work dominates the persistent work.



Frame Overview

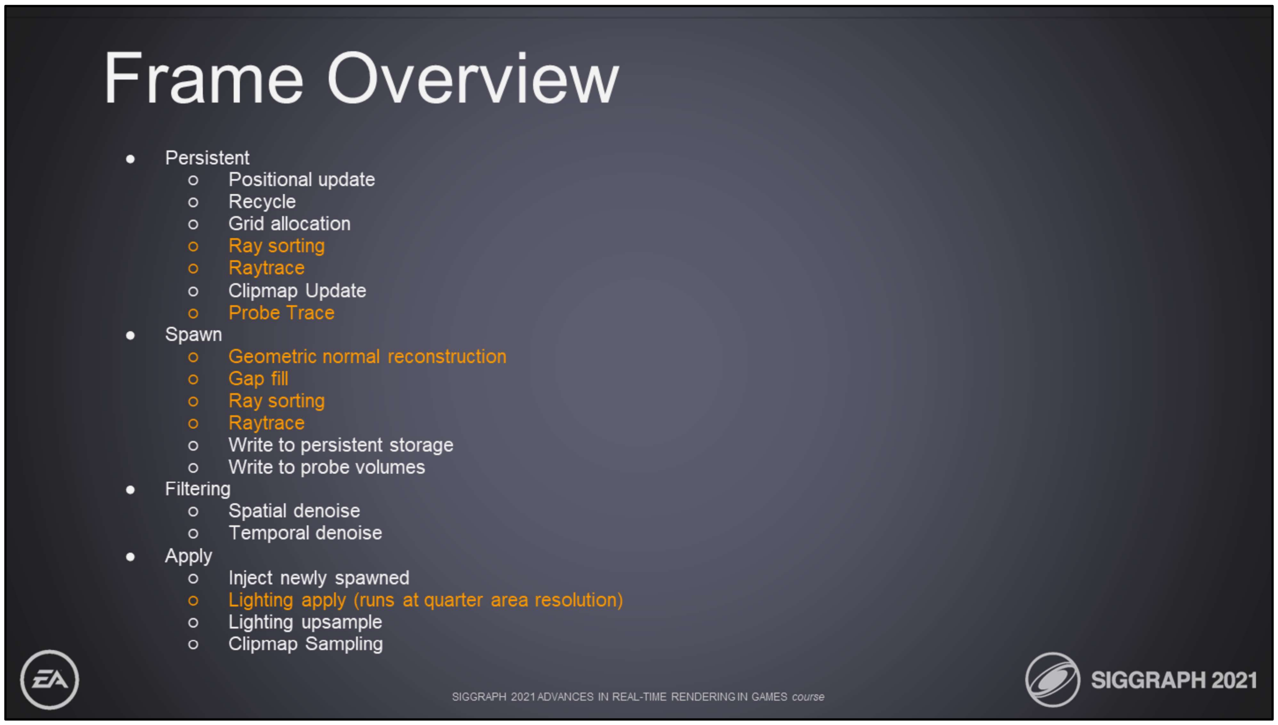
- Persistent
  - Positional update
  - Recycle
  - Grid allocation
  - Ray sorting
  - Raytrace
  - Clipmap Update
  - Probe Trace
- Spawn
  - Geometric normal reconstruction
  - Gap fill
  - Ray sorting
  - Raytrace
  - Write to persistent storage
  - Write to probe volumes
- Filtering
  - Spatial denoise
  - Temporal denoise
- Apply
  - Inject newly spawned
  - Lighting apply (runs at quarter area resolution)
  - Lighting upsample
  - Clipmap Sampling

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

For spawned surfels, depending on how many are spawning in a frame, the raytracing or the geometric normal reconstruction can dominate. Or, depending on resolution, the gap fill can dominate.

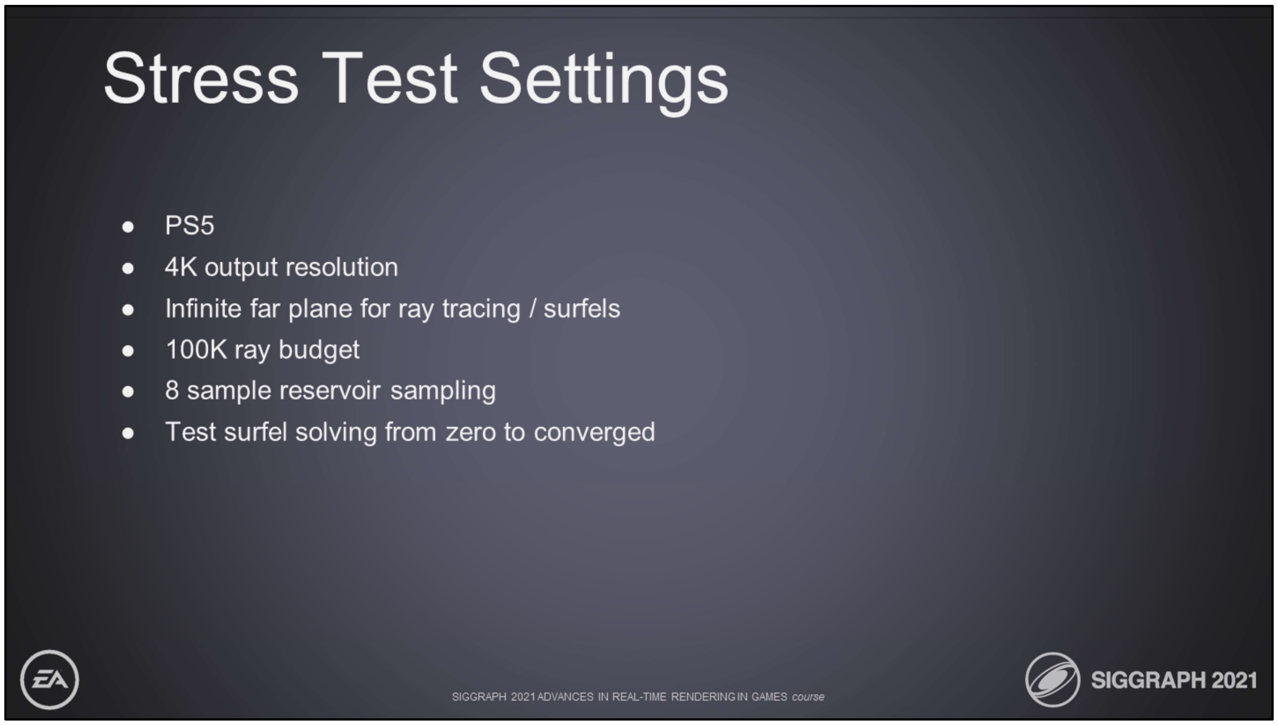
Filtering is generally very fast, averaging around .2ms



Frame Overview

- Persistent
  - Positional update
  - Recycle
  - Grid allocation
  - Ray sorting
  - Raytrace
  - Clipmap Update
  - Probe Trace
- Spawn
  - Geometric normal reconstruction
  - Gap fill
  - Ray sorting
  - Raytrace
  - Write to persistent storage
  - Write to probe volumes
- Filtering
  - Spatial denoise
  - Temporal denoise
- Apply
  - Inject newly spawned
  - Lighting apply (runs at quarter area resolution)
  - Lighting upsample
  - Clipmap Sampling

And finally, for surfel application. The Lighting apply pass dominates this group.



Stress Test Settings

- PS5
- 4K output resolution
- Infinite far plane for ray tracing / surfels
- 100K ray budget
- 8 sample reservoir sampling
- Test surfel solving from zero to converged

SIGGRAPH 2021

All the numbers and timings that follow have the following settings. We're testing on a PS5 with 4k resolution. Which means our screen-space passes run at 1080p.

We're not limiting how far the surfels are spawned or cutting off rays at a certain distance.

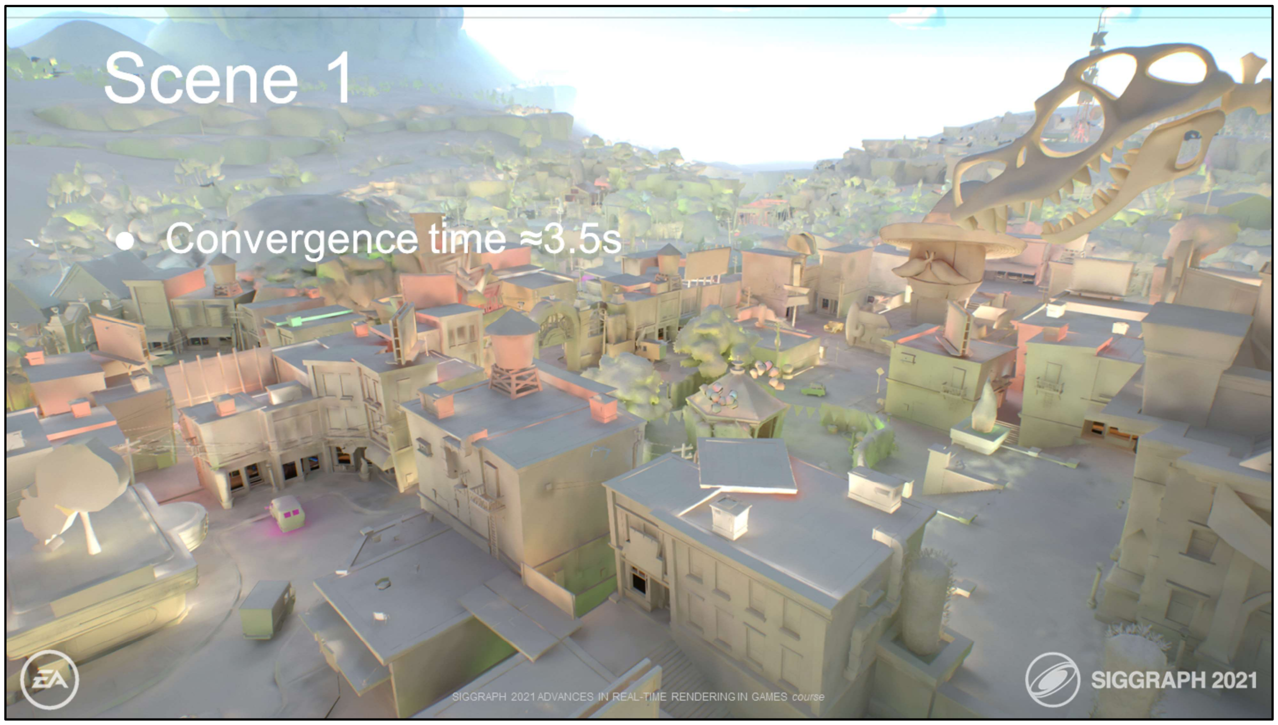
And, we're using 8 sample reservoir sampling, unless otherwise noted.

Lastly, we're testing the worst case for our solution: starting from an empty surfel scene, and then to converged to acceptable quality. Once we've converged, the costs trail off, so we'll focus on the expensive part in the next few performance examples.

Scene 1

- Convergence time ≈3.5s

Here's example scene 1, using Plants vs Zombies towncenter map. With sunlight only.



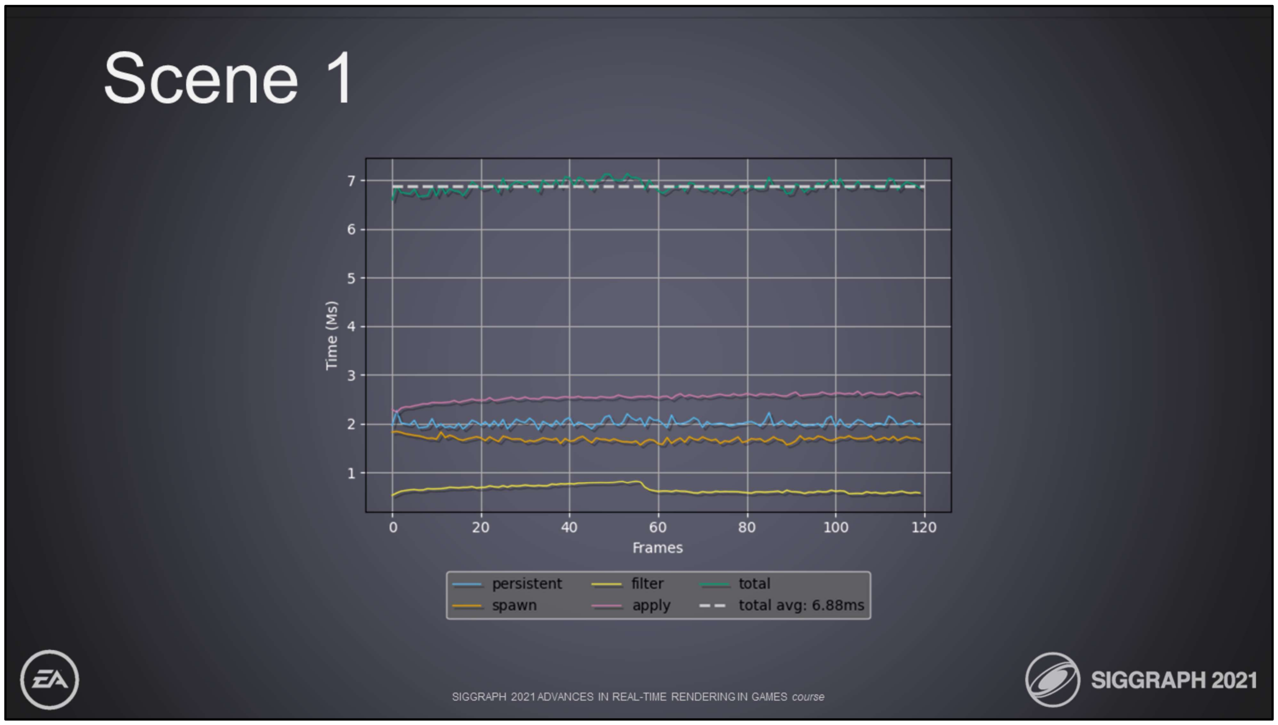
Scene 1

- Convergence time ≈3.5s

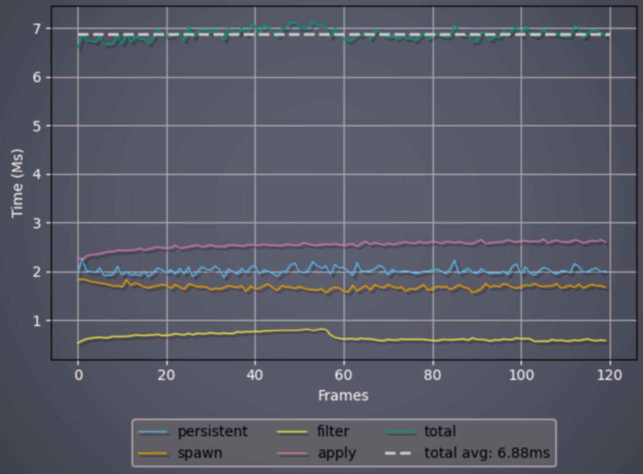
SIGGRAPH 2021

And here's the indirect diffuse after we've converged to acceptable quality.

(note some of the albedo colors used for GI are not 100% true to the albedo textures)



Scene 1



In this scene, due to the large draw distance, we average roughly 7ms total.

Scene 2a

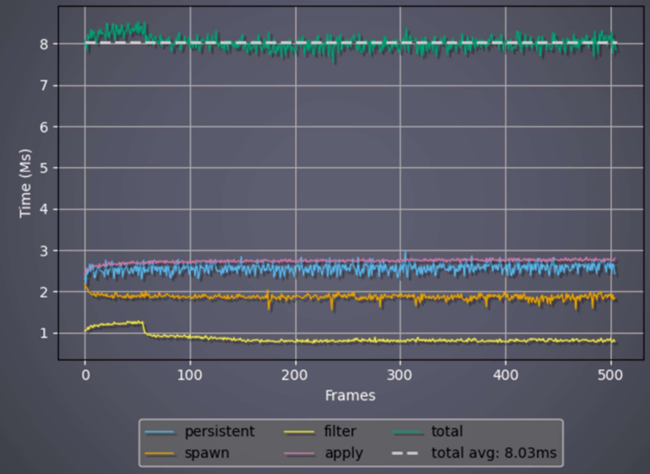
- Convergence time ≈ never

SIGGRAPH 2021

Here’s scene two. This is the same view we showed earlier when discussing our many-light solution.

This is our stress test scene with 1400 lights in view. It converges in…. About never.

Scene 2a



So we can see that there is a bit of a tax when it comes to sampling local lights compared to the daytime example. But, it takes ages to converge.

Scene 2b

- 1M target ray budget
- Convergence time ≈ 10s

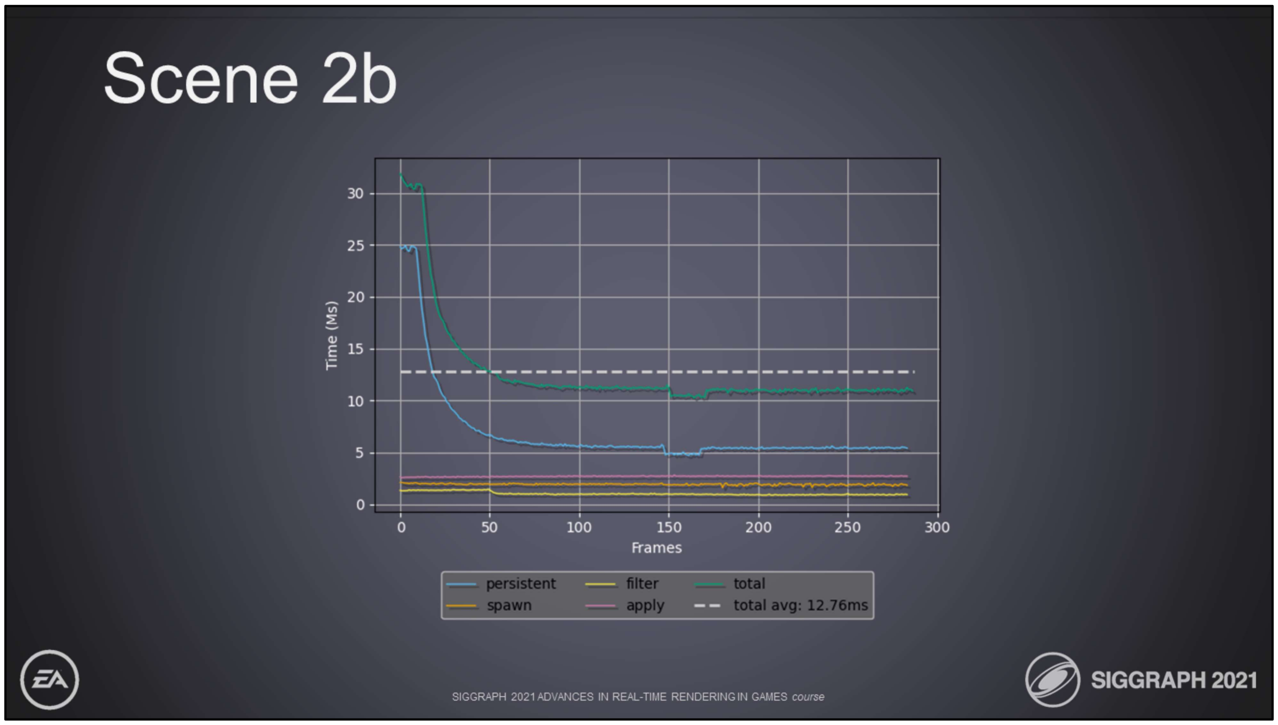
SIGGRAPH 2021

So what does it take to converge? Here’s the same scene, now with a 1 million ray budget

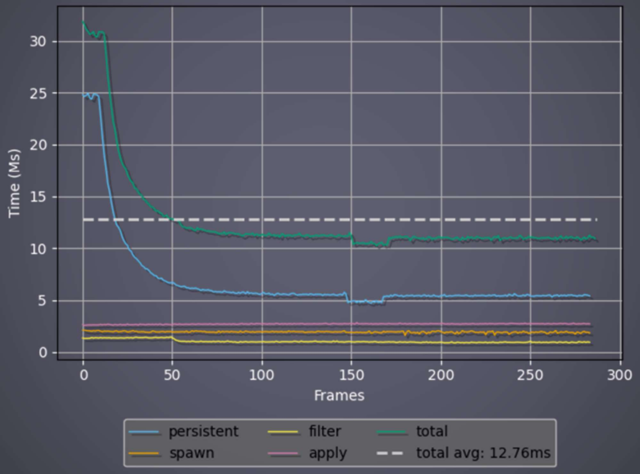
Scene 2b

- 1M target ray budget
- Convergence time ≈ 10s

And still quite noisy after ten seconds, but it’s getting there.



Scene 2b



SIGGRAPH 2021

So we spike quite high in the beginning, but level off quite fast. Though we're still pushing 11 milliseconds once we've started to converge. And this is one of the situations we want to accel at, so we still have some work to do!

Scene 3a

- Convergence time ≈ 7s

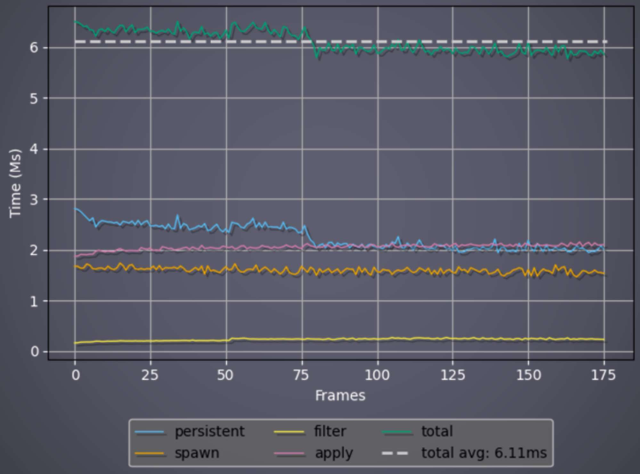
CINEMAS
COMING SOON
SAD THIRTY-SOMETHING MOVES BACK HOME
NO PARKING

Here’s scene three. This is the same scene 1400 light scene, but now a different view.

Scene 3a

- Convergence time ≈ 7s

Scene 3a



SIGGRAPH 2021

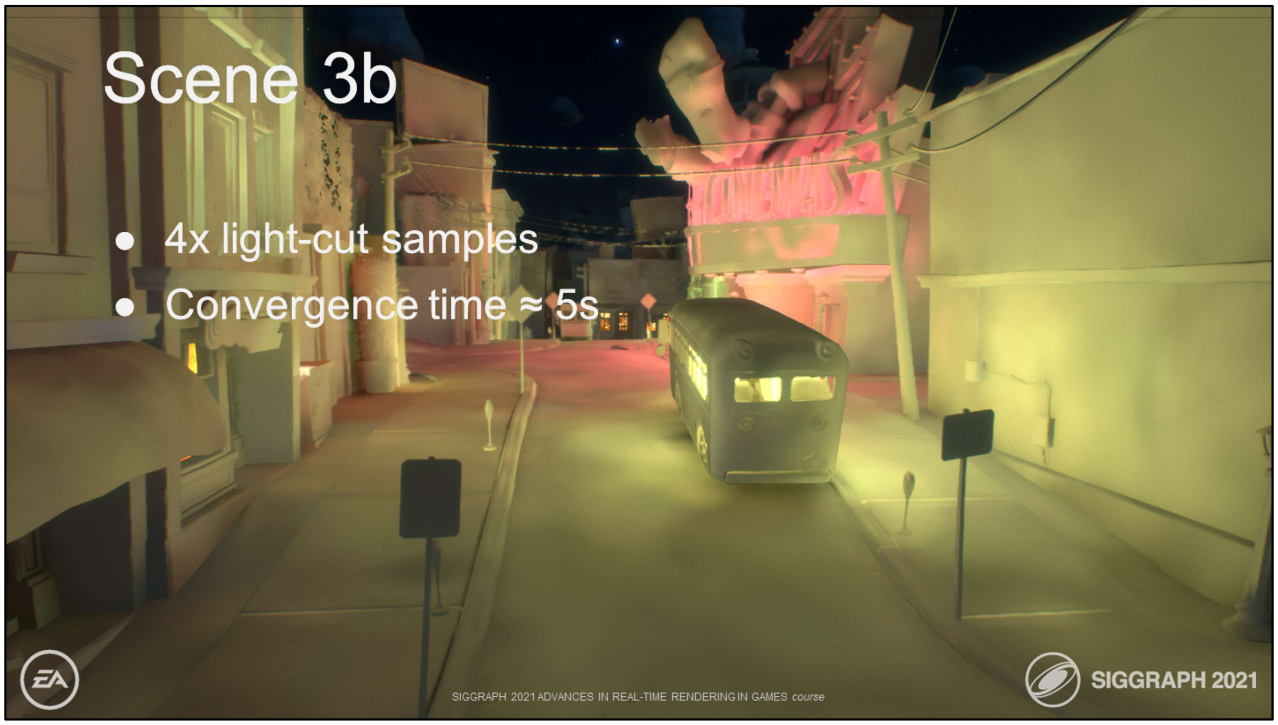
As you can see it’s quite a bit cheaper with our more constrained view. And it also converged much faster.

Scene 3b

- 4x light-cut samples
- Convergence time ≈ 5s

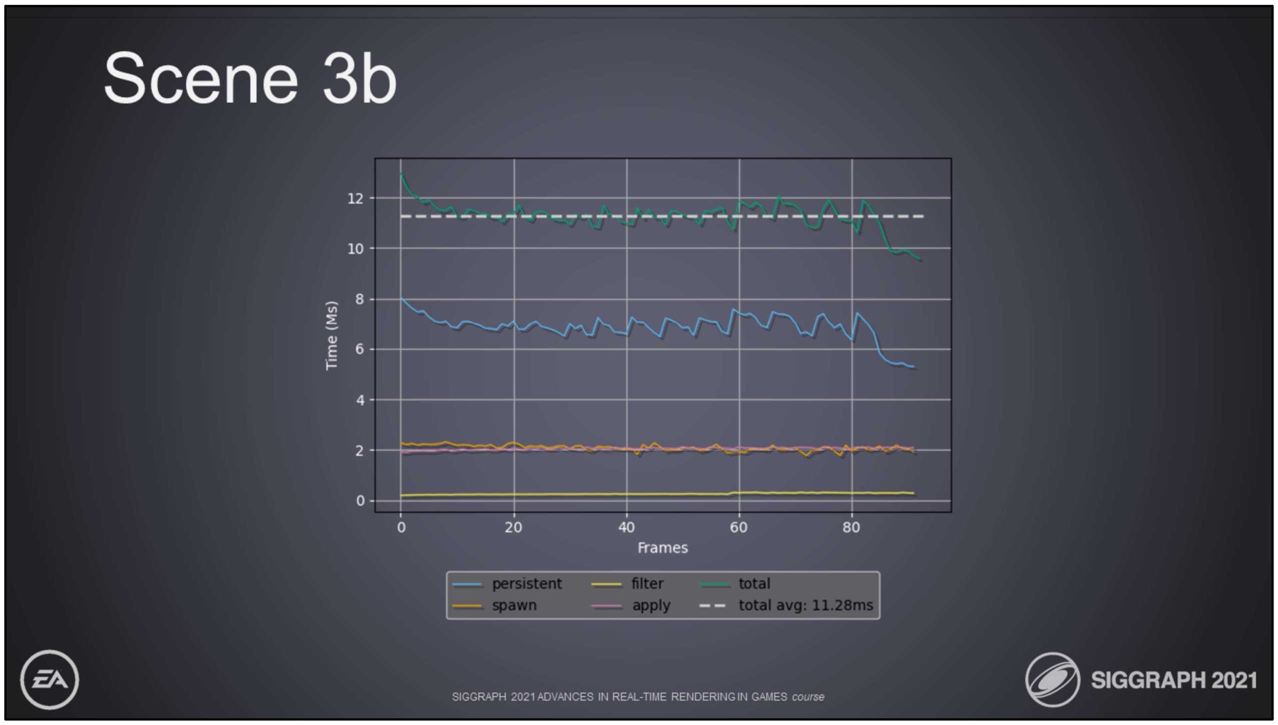
COMING SOON
SIGGRAPH 2021

But what if we want faster convergence? Here we do the same test but with 4 sample lightcuts.

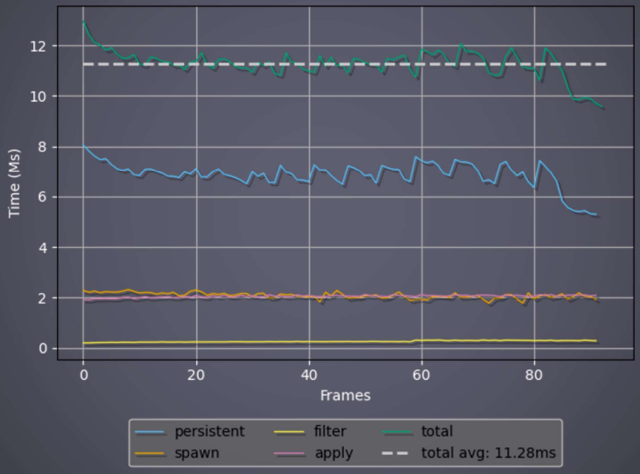


Scene 3b

- 4x light-cut samples
- Convergence time ≈ 5s



Scene 3b



So, pretty costly to gain the advantage in convergence time. Converging fast in scenes with many-lights is still actively under development, but I think we have a promising start here.

And as you can on the right of the graph, both persistent and spawn are trending downwards now that variance is stabilizing.

Scene 4

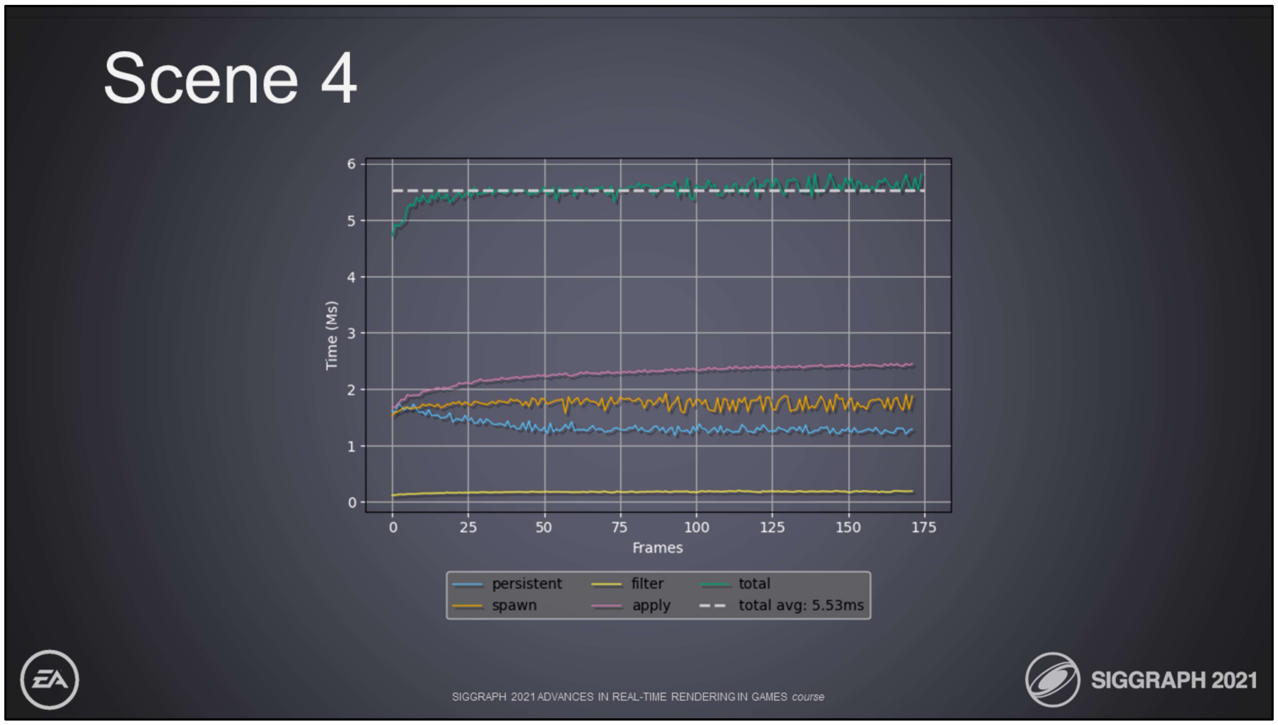
- Convergence time ≈ 2s

And finally our last scene from an internal demo. This level has 140 lights.

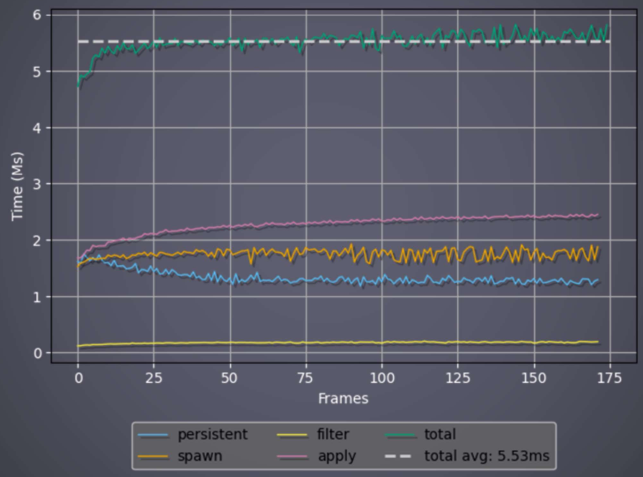
Scene 4

- Convergence time ≈ 2s

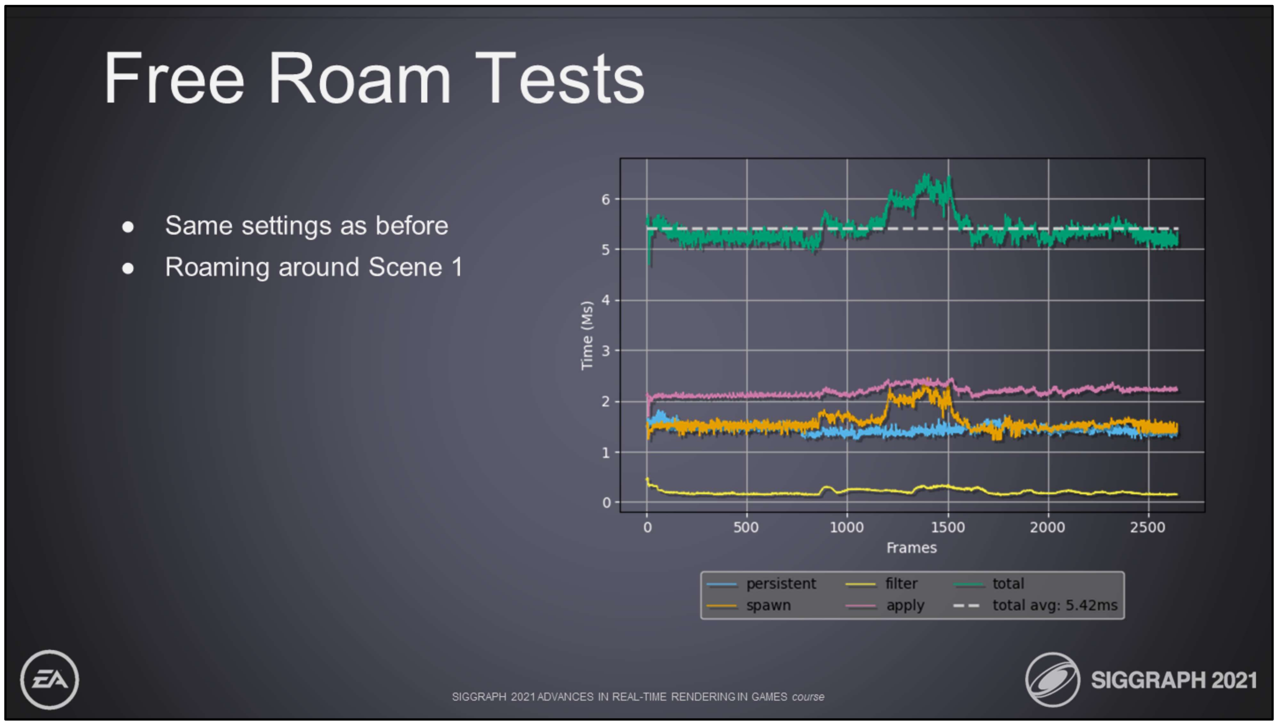
Convergence time is pretty fast here with a constrained view.



Scene 4

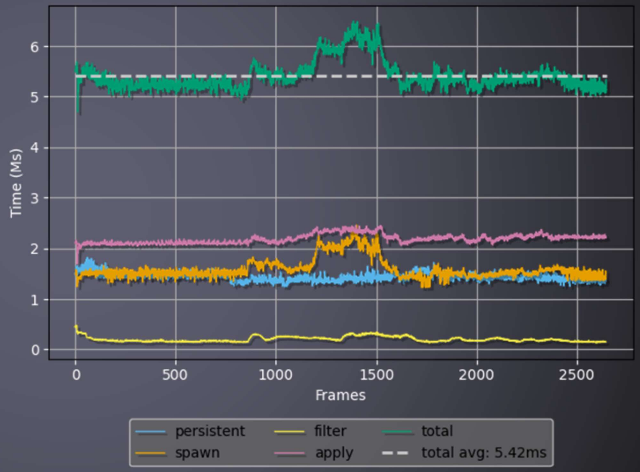


And performance is even better than the street view from PvZ.



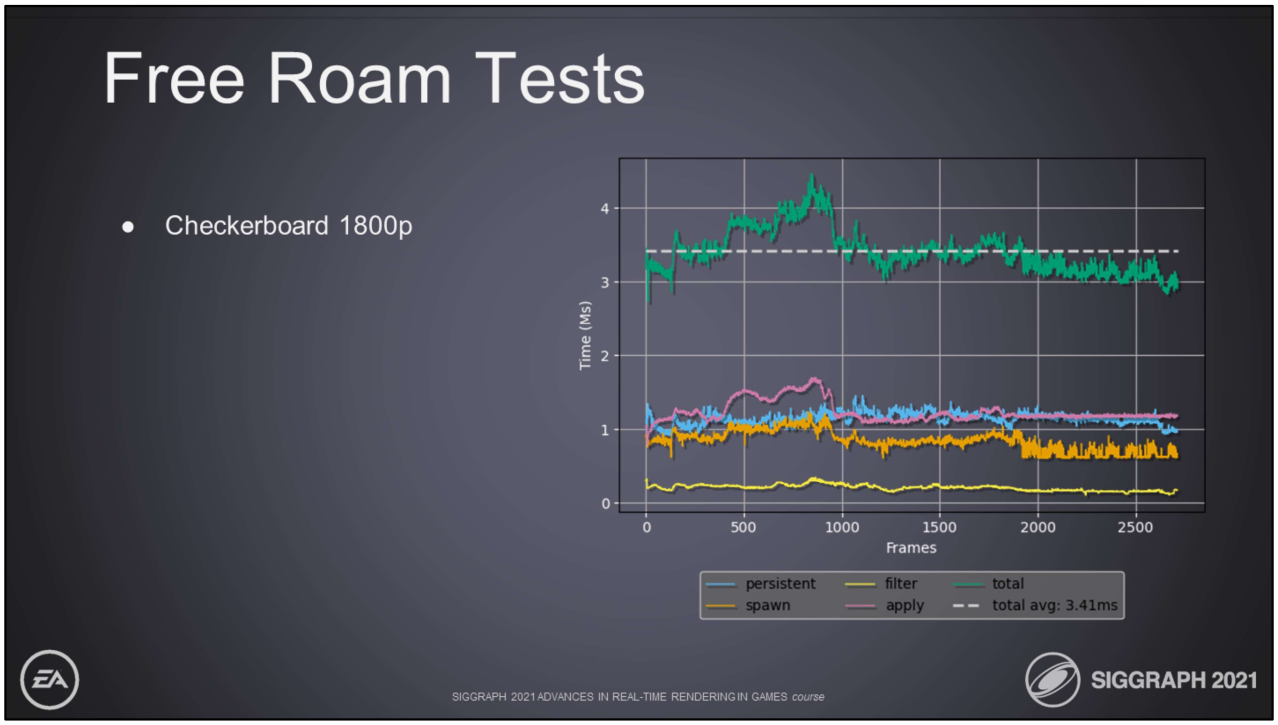
Free Roam Tests

- Same settings as before
- Roaming around Scene 1



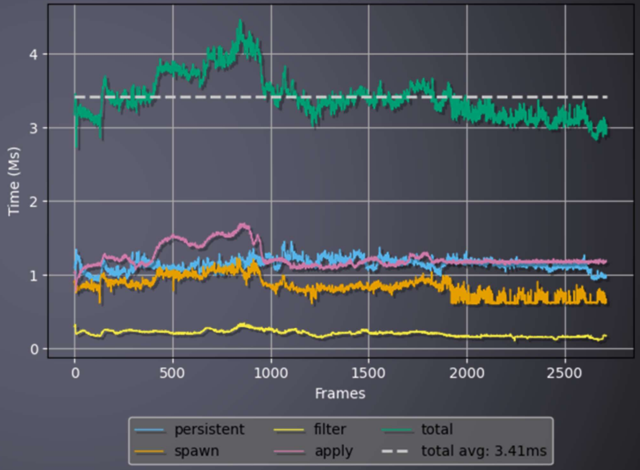
Now I'll show a couple of examples of roaming around scene one from PvZ. Something more representative of real-world performance, as surfels will constantly spawn and need solving while moving through the level.

Here we have the same settings as previously, and we can see performance is better than in the stress test.



Free Roam Tests

- Checkerboard 1800p

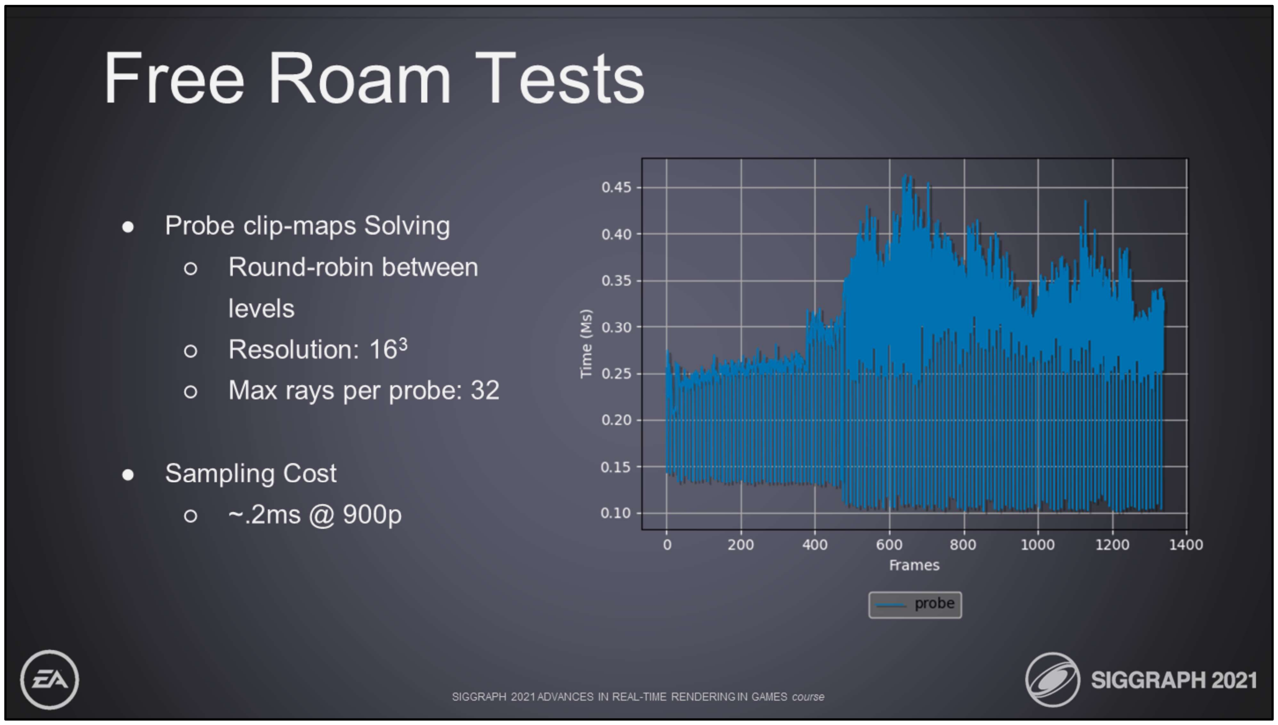


SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

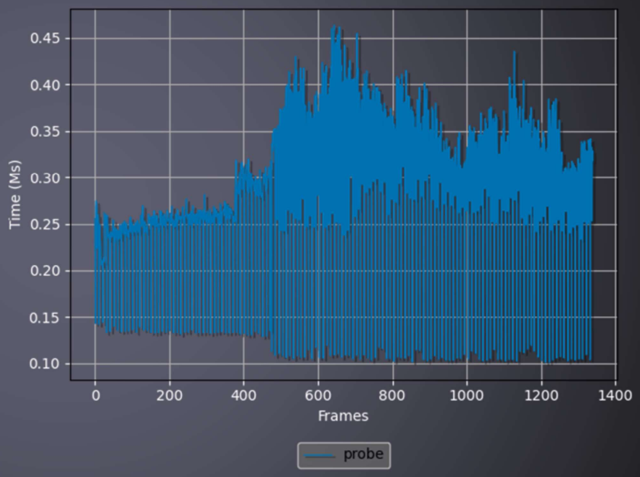
And here we have relatively the same path as the previous slide, but now we’re using an output resolution of 1800p with checkerboard rendering enabled.

This test is more representative of the settings we expect games to use. We can see we’re quite a bit faster here. We still have work to do, but it shows we’re in the ballpark in performance.



Free Roam Tests

- Probe clip-maps Solving
  - Round-robin between levels
  - Resolution: 16^3
  - Max rays per probe: 32
- Sampling Cost
  - ~.2ms @ 900p

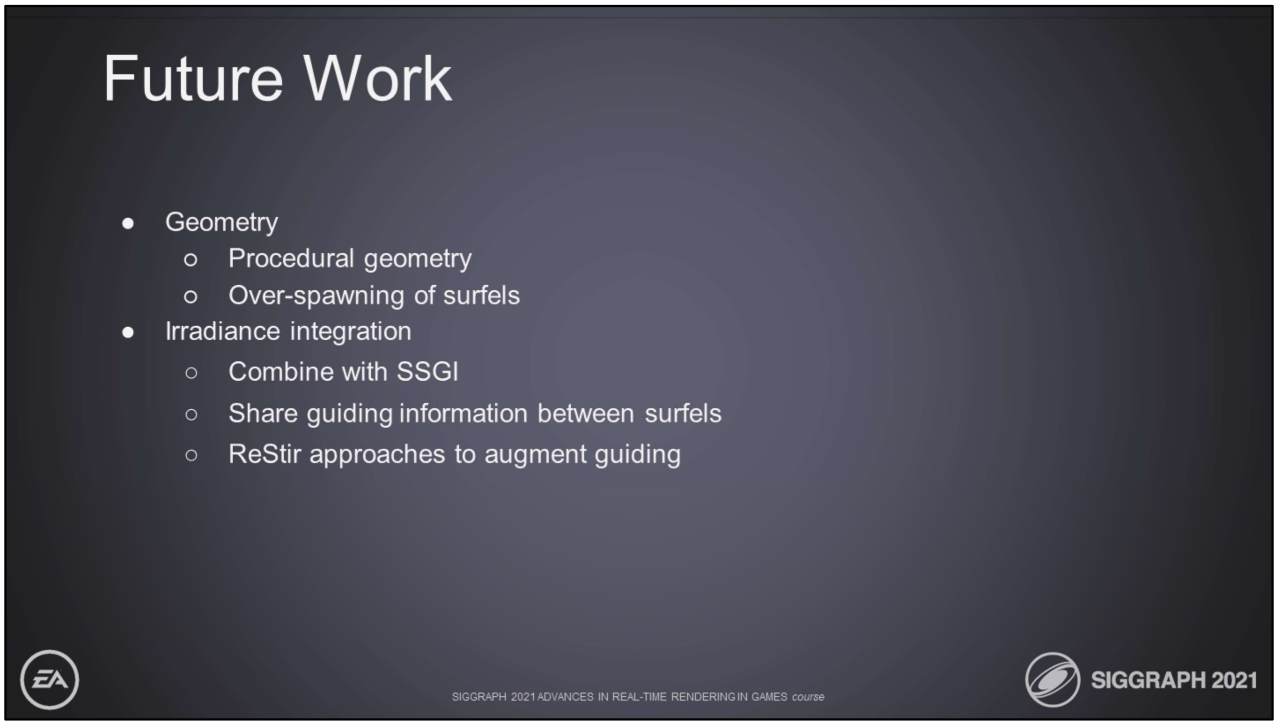


SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

And finally, here’s an example of the cost to solve our clip-map probes. This follows the same route in the previous free roam tests. We have a 3 level clip-map and the cost is very reasonable. We cycle which clip-map we solve every few frames. The third clipmap is the cheapest as most rays are misses, as can be seen in the graph.

The sampling cost for probes in a full res pass at 900p, is roughly .2ms



Future Work

- Geometry
  - Procedural geometry
  - Over-spawning of surfels
- Irradiance integration
  - Combine with SSGI
  - Share guiding information between surfels
  - ReStir approaches to augment guiding

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

So that about sums up where our GIBS technology is at today. Here are some details on where we’re headed:

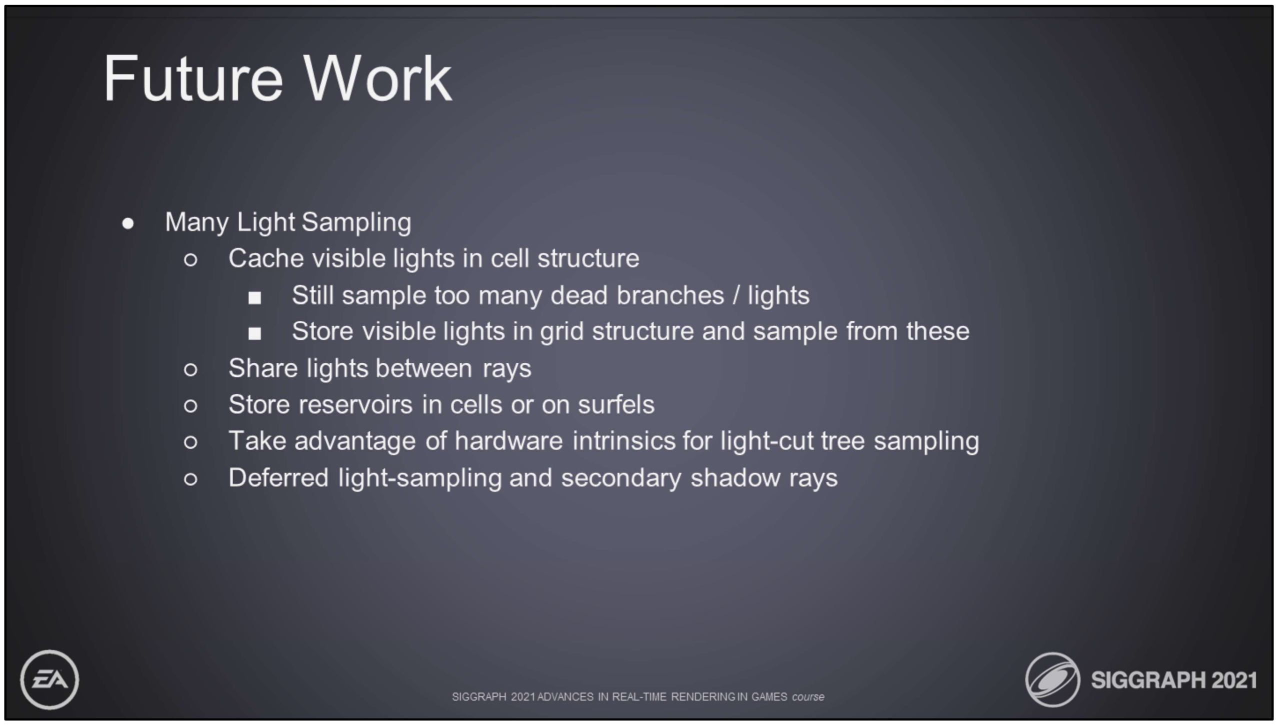
Currently we don’t support procedural geometry explicitly. Procedural geometry works, but will continuously spawn and recycle surfels.

We also have certain situations where high-detail geometry can over-spawn surfels. One of our options is to combine with screen space global illumination to minimize surfel spawning in areas where SSGI solves well.

Combining with SSGI will also help limit how many surfels we need to solve, and it will also help limit how far out we cull surfels and raytracing.

We are investigating sharing guiding information across surfels in order to get a better picture for where to send rays

And we are investigating using ReStir-like approaches to augment the guiding



Future Work

- Many Light Sampling
  - Cache visible lights in cell structure
    - Still sample too many dead branches / lights
    - Store visible lights in grid structure and sample from these
  - Share lights between rays
  - Store reservoirs in cells or on surfels
  - Take advantage of hardware intrinsics for light-cut tree sampling
  - Deferred light-sampling and secondary shadow rays

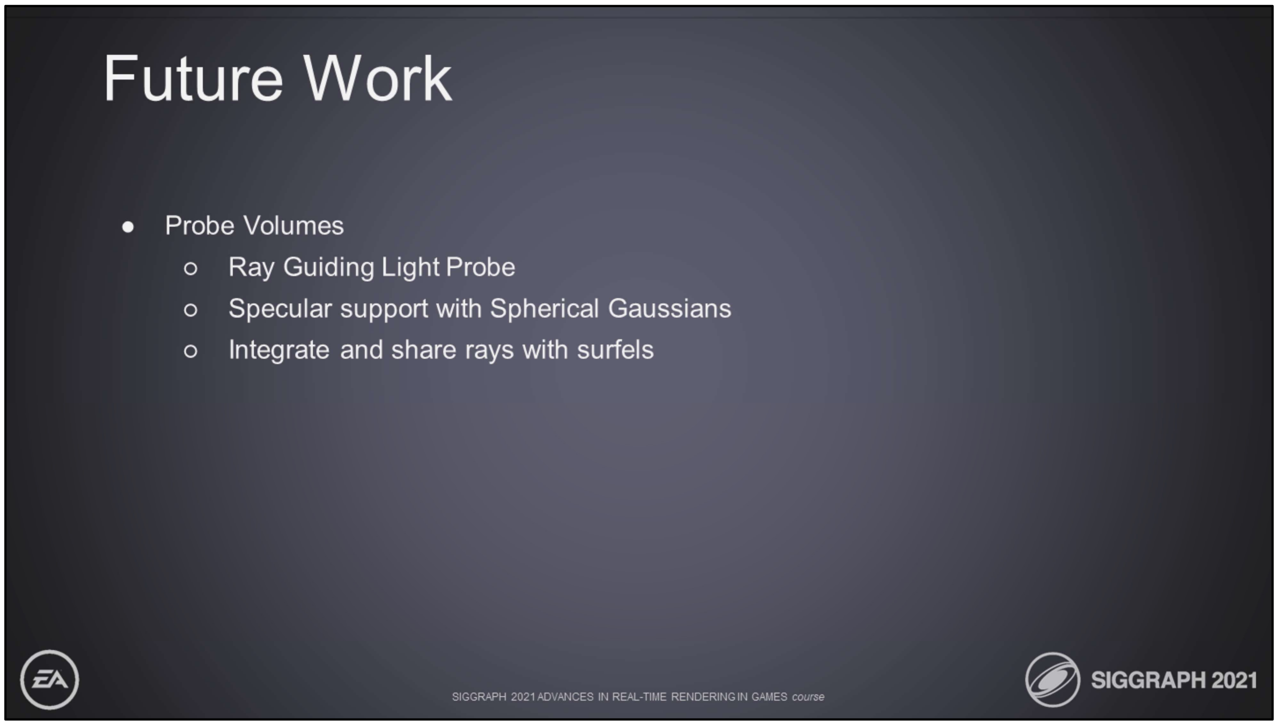
- We still end up selecting too many dead branches or lights. If we can store visibility of lights in our cell or voxel data structures, then we can vastly limit the lights we need to consider per ray-hit

- We bin and sort our rays, so it may be advantageous to share lights between these rays. For example, for reservoir sampling, we could do our spatial sample in this fashion.

- For temporal reservoir sampling, we are investigating storing reservoirs in the cell structure or on each surfel

- Our light-cut traversal is not well optimized yet, and could make use of the platform specific intrinsics

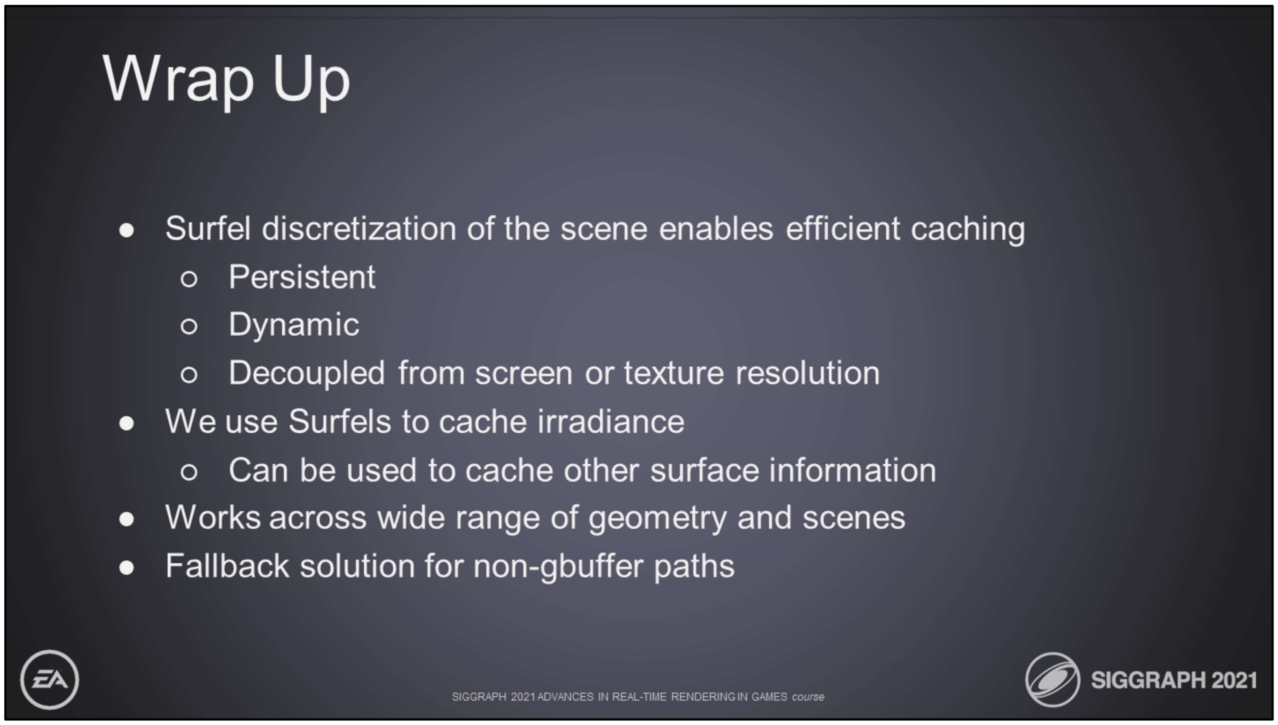
- And finally, we need to move our light-sampling out or our raytracing pass and schedule secondary shadow rays to a later pass.



Future Work

- Probe Volumes
  - Ray Guiding Light Probe
  - Specular support with Spherical Gaussians
  - Integrate and share rays with surfels

Our probe clipmap solution is still early days and we are looking to add ray-guiding and specular support in the future. And improve performance by integrating and sharing rays with surfel ray dispatch.



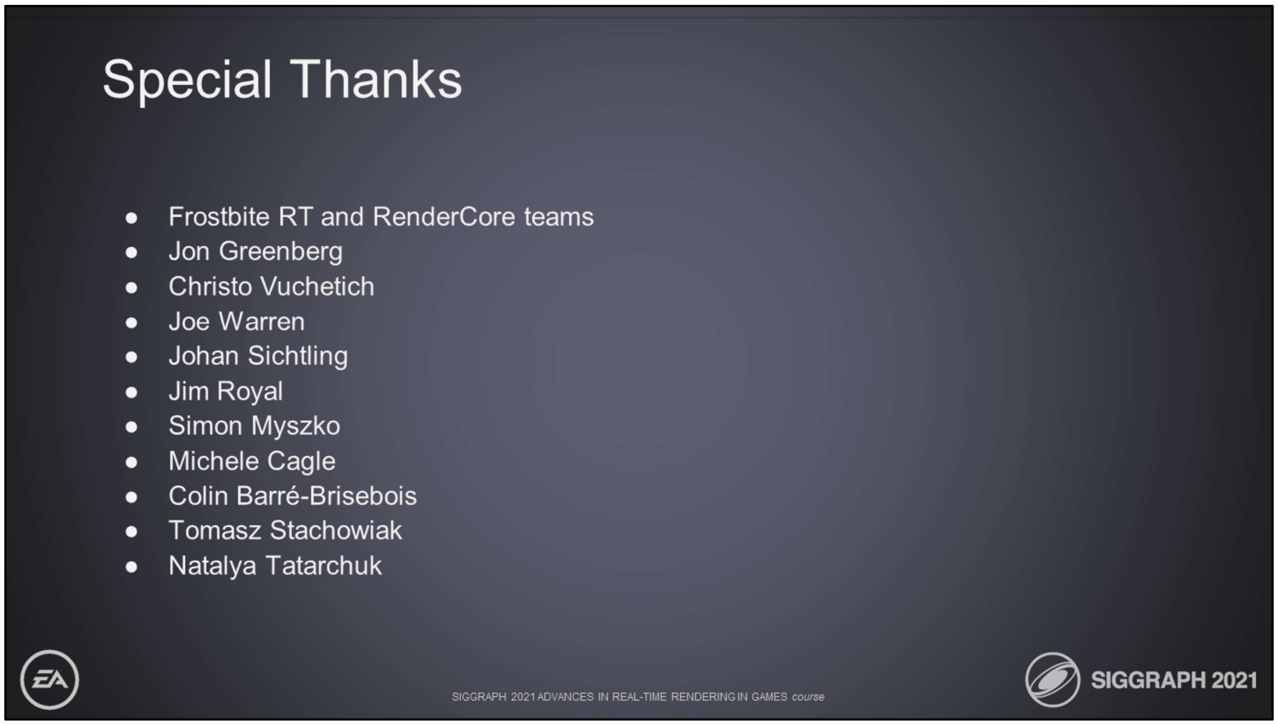
Wrap Up

- Surfel discretization of the scene enables efficient caching
  - Persistent
  - Dynamic
  - Decoupled from screen or texture resolution
- We use Surfels to cache irradiance
  - Can be used to cache other surface information
- Works across wide range of geometry and scenes
- Fallback solution for non-gbuffer paths

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

To wrap things up, we have seen that the opportunistic surfelization of the scene provides an efficient mechanism to cache information on surfaces of opaque geometry.
The caching mechanism is persistent, dynamic and decoupled from the output resolution.
We use surfels to cache irradiance on a wide range of geometry and scenes,
And provide a fallback solution for geometry that does not fit the requirements.



Special Thanks

- Frostbite RT and RenderCore teams
- Jon Greenberg
- Christo Vuchetich
- Joe Warren
- Johan Sichtling
- Jim Royal
- Simon Myszko
- Michele Cagle
- Colin Barré-Brisebois
- Tomasz Stachowiak
- Natalya Tatarchuk

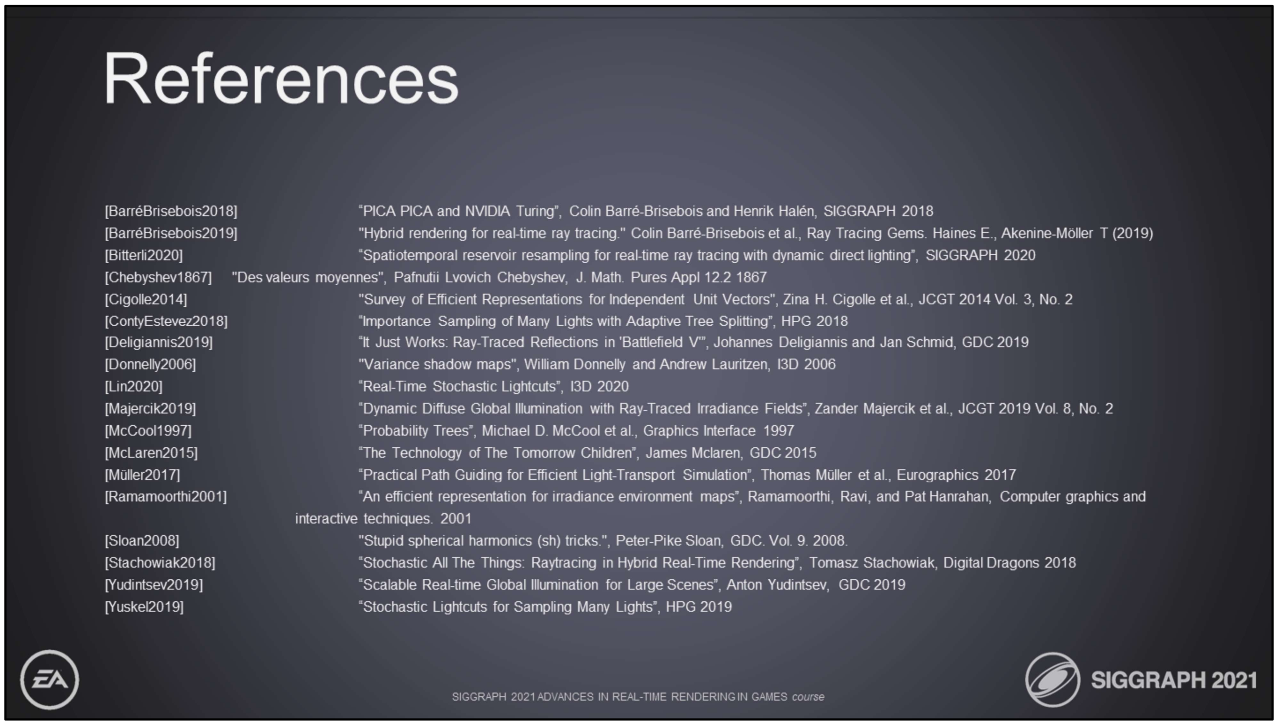
We would like to thank the Frostbite Ray Tracing and RenderCore teams, for bringing Ray tracing support to the engine.
We thank Jon Greenberg for his contributions and research on many light sampling.
Joe Warren, Johan Sichtling and Christo Vuchetich for providing high quality content for this presentation.
Jim Royal for proofreading and ensuring the presentation meets our quality standards.
And last but not least, Tomasz Stachowiak for the original implementation of surfel GI back in 2018.

Thank you!

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES *course*

SIGGRAPH 2021

Thank you for attending our talk, I hope you enjoyed the presentation.



References

[BarréBrisebois2018] "PICA PICA and NVIDIA Turing", Colin Barré-Brisebois and Henrik Halén, SIGGRAPH 2018

[BarréBrisebois2019] "Hybrid rendering for real-time ray tracing." Colin Barré-Brisebois et al., Ray Tracing Gems. Haines E., Akenine-Möller T (2019)

[Bitterli2020] "Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting", SIGGRAPH 2020

[Chebyshev1867] "Des valeurs moyennes", Pafnutii Lvovich Chebyshev, J. Math. Pures Appl 12.2 1867

[Cigolle2014] "Survey of Efficient Representations for Independent Unit Vectors", Zina H. Cigolle et al., JCGT 2014 Vol. 3, No. 2

[ContyEstevez2018] "Importance Sampling of Many Lights with Adaptive Tree Splitting", HPG 2018

[Deligiannis2019] "It Just Works: Ray-Traced Reflections in 'Battlefield V'", Johannes Deligiannis and Jan Schmid, GDC 2019

[Donnelly2006] "Variance shadow maps", William Donnelly and Andrew Lauritzen, I3D 2006

[Lin2020] "Real-Time Stochastic Lightcuts", I3D 2020

[Majercik2019] "Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields", Zander Majercik et al., JCGT 2019 Vol. 8, No. 2

[McCool1997] "Probability Trees", Michael D. McCool et al., Graphics Interface 1997

[McLaren2015] "The Technology of The Tomorrow Children", James Mclaren, GDC 2015

[Müller2017] "Practical Path Guiding for Efficient Light-Transport Simulation", Thomas Müller et al., Eurographics 2017

[Ramamoorthi2001] "An efficient representation for irradiance environment maps", Ramamoorthi, Ravi, and Pat Hanrahan, Computer graphics and interactive techniques. 2001

[Sloan2008] "Stupid spherical harmonics (sh) tricks.", Peter-Pike Sloan, GDC. Vol. 9. 2008.

[Stachowiak2018] "Stochastic All The Things: Raytracing in Hybrid Real-Time Rendering", Tomasz Stachowiak, Digital Dragons 2018

[Yudintsev2019] "Scalable Real-time Global Illumination for Large Scenes", Anton Yudintsev, GDC 2019

[Yuskel2019] "Stochastic Lightcuts for Sampling Many Lights", HPG 2019