

## Shipping Dynamic Global Illumination in Frostbite

Diede Apers  
Rendering Engineer  
EA Frostbite



© 2024 SIGGRAPH ADVANCES IN REAL-TIME RENDERING IN GAMES course. ALL RIGHTS RESERVED.

My name is Diede (dee-duh) Apers, I'm a Rendering Engineer at Frostbite.

And in this presentation, I will provide an update on Global Illumination Based on Surfels which is being used in several game productions powered by Frostbite.

**Agenda**

- **Introduction**
  - GIBS & Productions
- **Improvements**
  - Probes & Surfels
- **Optimizations**
  - Compute & Ray Tracing
- **Results**
  - Quality & Performance

© 2024 SIGGRAPH ADVANCES IN REAL-TIME RENDERING IN GAMES course. ALL RIGHTS RESERVED.

SIGGRAPH 2024  
DENVER+ 28 JUL — 1 AUG

2

The agenda for today covers a quick introduction to GIBS and the productions that are using it.

We'll go over some improvements that we have made to our probe and surfel systems, as well as optimizations to our compute and ray tracing pipeline.

Finally, I'll show some visual results as well as a performance breakdown, and there should be some time left for questions after.



# Introduction

GIBS & Productions

© 2024 SIGGRAPH ADVANCES IN REAL-TIME RENDERING IN GAMES course. ALL RIGHTS RESERVED.

3

## Global Illumination in Frostbite

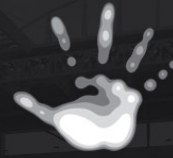
Frostbite ships a **variety of games**

**Different** diffuse GI **systems** available

- Flux [1, 2] – Precomputed
- Enlighten [3] – Partially precomputed
- GiGrid [4] – Streaming irradiance volumes

**Reaching the limit** in cases such as

- Large and open worlds
- Dynamic environments and destruction
- Procedural and user-generated content



Frostbite

BATTLEFIELD DRAGONAGE™

DEAD SPACE

NES UNBOUND

skate.™



COLLEGE FOOTBALL

Frostbite ships a lot of games, each with their own unique requirements. For this reason, Frostbite has several GI systems available depending on the game's needs.

Flux provides high quality but fully precomputed diffuse global illumination while Enlighten provides partial dynamic updates. GiGrid is used to stream irradiance volumes in the form of probes for both systems.

However, using these systems can be very costly to author during production. We are also limited in capability in the cases of; large and open worlds, dynamic environments with destruction, and procedural or user generated content.

We introduced Global Illumination Based on Surfels as an option to lift these restrictions.

## Dynamic (Diffuse) Global Illumination

 SIGGRAPH 2024  
DENVER+ 28 JUL — 1 AUG



### Global Illumination Based on Surfels

- All inputs **fully dynamic**
- No authoring required to get running
- **High quality** indirect diffuse lighting
- Uses **hardware ray tracing**

Collaboration with EA SEED [5]

Presented within Frostbite in 2021 [6]



SIGGRAPH 2021 Advances in Real-Time  
VIRTUAL 9-13 AUGUST Rendering in Games course



© 2024 SIGGRAPH ADVANCES IN REAL-TIME RENDERING IN GAMES course. ALL RIGHTS RESERVED.

5

All inputs to our GI system are fully dynamic, we can move meshes, change materials and add lights at runtime. No authoring is required to run GIBS initially, of course authoring can still be done to improve quality and/or performance where needed.

Surfel GI started as a research project at EA SEED, and we have been collaborating with them ever since we introduced GIBS to Frostbite, which we presented at the Advances in 2021.

## GIBS @ Advances '21

Thorough description of the system


- Fully functional dynamic GI
- Runs on current **generation consoles**

Two separate systems

- Surfels **decouple** ray tracing rate
- Probes as irradiance field
  - e.g., transparent

Did not ship in a game just yet

- Total avg gibs gpu time **-6ms**



© 2024 SIGGRAPH ADVANCES IN REAL-TIME RENDERING IN GAMES course. ALL RIGHTS RESERVED.

We gave a thorough description of the entire GIBS system then, so I will not be going into the details already described there. We showed that the system is fully functional and dynamic and runs on on the current generation consoles. We introduced two separate systems; there's the surfels which decouple the ray tracing rate from the shading rate, which is quite important for performance. We also introduced probes used to sample irradiance at arbitrary locations in the scene. This allows sampling of indirect lighting for draws that are not going through our deferred lighting pipeline, such as transparent objects.

However, back then we used content from a game that had

already been released, and as such GIBS had not yet shipped. One of the problems we faced was performance with a total average gpu time for GIBS of 6-7ms. Frostbite games usually ship at 60fps and don't have 6ms to spare generally.



## GIBS @ Advances '24

### Did not introduce any new systems

- Incremental **improvements**
  - Surfels & Probes systems
  - Ray tracing infrastructure
- Significant **optimizations**
  - Compute shader efficiency
  - Optional bias for better perf

### Used in multiple game productions

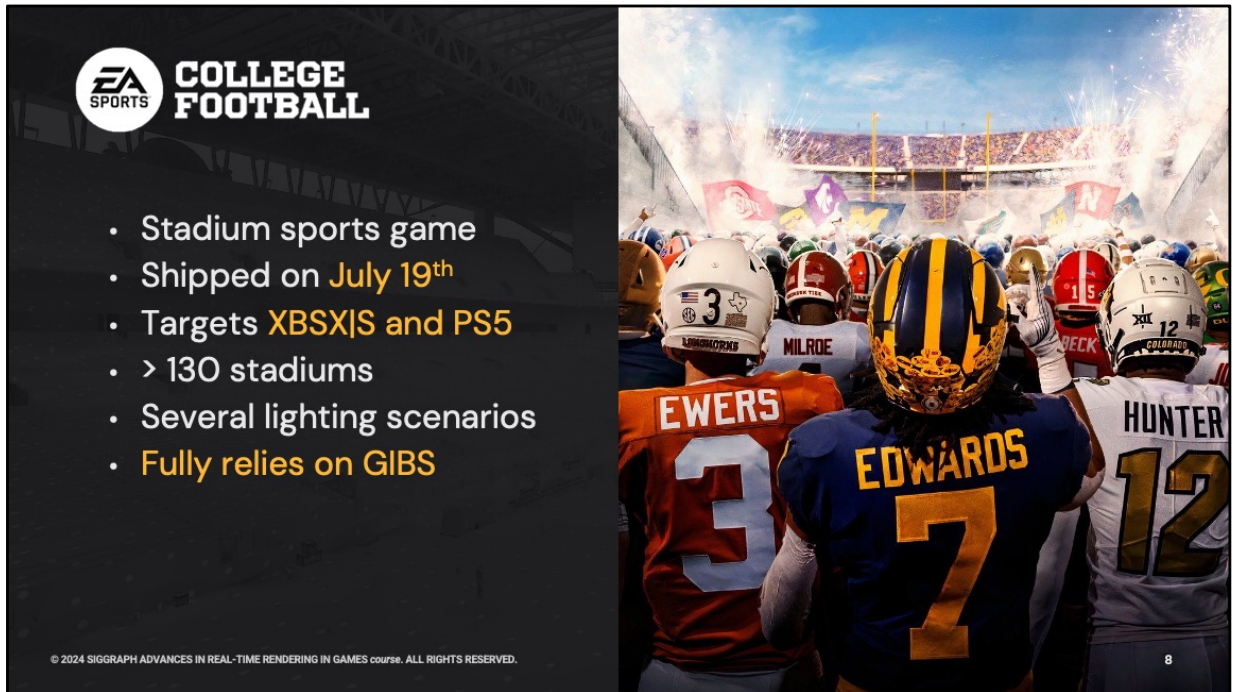
- **Shipped** in a stadium sports title
- Production in open world action title

© 2024 SIGGRAPH ADVANCES IN REAL-TIME RENDERING IN GAMES course. ALL RIGHTS RESERVED.



In these past 3 years we did not introduce any new systems. We still use ray tracing, surfels and probes. Instead, we've done incremental improvements to our surfels and probes systems, as well as our infrastructure for ray tracing. A significant number of optimizations went in. Most of these optimizations have had no impact on the quality of the indirect light itself, but there are a few which introduce bias to allow for fewer ray intersection tests. I'll provide numbers on these later and talk about those optimizations separately.

GIBS is now used in multiple game productions, one of which is a stadium sports title and has recently shipped, as well as an open world action game which is still in production.

The image is a promotional graphic for EA Sports College Football 25. On the left, a dark grey panel contains the EA Sports logo and the game title 'COLLEGE FOOTBALL'. Below this is a bulleted list of features: 'Stadium sports game', 'Shipped on July 19th', 'Targets XBOX|S and PS5', '> 130 stadiums', 'Several lighting scenarios', and 'Fully relies on GIBS'. At the bottom left of this panel is a small copyright notice: '© 2024 SIGGRAPH ADVANCES IN REAL-TIME RENDERING IN GAMES course. ALL RIGHTS RESERVED.' To the right of the text panel is a vibrant, high-resolution screenshot from the game. It shows a group of football players from different teams huddled together on a field. The player in the center foreground is wearing a blue jersey with the name 'EDWARDS' and the number '7'. To his left is a player in an orange jersey with 'EWERS' and '3'. To his right is a player in a white jersey with 'HUNTER' and '12'. The background is filled with other players, flags, and a large stadium filled with spectators under a bright sky with some smoke or fireworks in the air.

Our first shipping release is College Football 25, which came out earlier this month. They ship exclusively on current generation consoles, which support hardware accelerated ray tracing. The game allows you to play in over 130 stadiums, and each of those can run at a different time of day. CFB25 fully relies on GIBS for all indirect diffuse lighting.

**skate.**™

FULL CIRCLE

- Open world action game
- Still in development
- Targets mobile, console, PC
- Enlighten[3] on low-end
- GIBS on XBOX|S, PS5 & PC

© 2024 SIGGRAPH ADVANCES IN REAL-TIME RENDERING IN GAMES course. ALL RIGHTS RESERVED.

Pre-Pre-Alpha

9

The slide features a dark background on the left with the 'skate.' logo and a list of features. On the right, a person in a black hoodie and tan pants holds a skateboard with a red and teal deck. The word 'chocolate' is written on the red section of the skateboard. The background shows an indoor skate park setting with a person sitting on the floor in the distance.

The second production is Skate, which is still in development. They intend to target a large range of devices, from mobile up to consoles and PC. On lower-end devices they use Enlighten in combination with GiGrid for their indirect diffuse lighting. However, GIBS allows them to scale up and stay competitive on high-end devices, as well as provide consistency in lighting with user generated content.

## System Overview

### Inputs

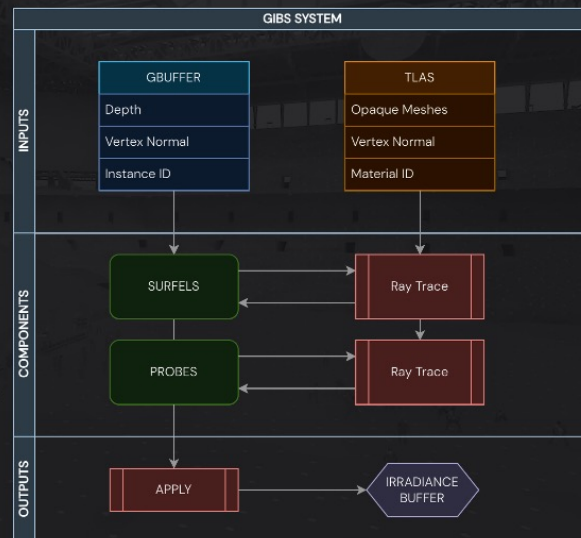
- **Gbuffer** used to spawn surfels
- **TLAS** used for ray tracing

### Components

- **Surfels** persistent & spawn ray trace
- **Probes** clipmap amortized ray trace

### Outputs

- **Irradiance buffer** for deferred lighting
- **Probes** for all other lighting



© 2024 SIGGRAPH ADVANCES IN REAL-TIME RENDERING IN GAMES course. ALL RIGHTS RESERVED.

Here's a quick overview of the system. Every frame, GIBS takes in our GBuffer, as seen on the top left of the diagram, to spawn surfels. We also build a Top-Level Acceleration Structure (TLAS) that can be used for ray tracing to estimate indirect lighting.

Both our Surfel and Probe systems use ray tracing to update their irradiance estimates each frame. We update our persistent surfels, which have been spawned in previous frames, as well as any newly spawned frames to make sure they get a good initial estimate.

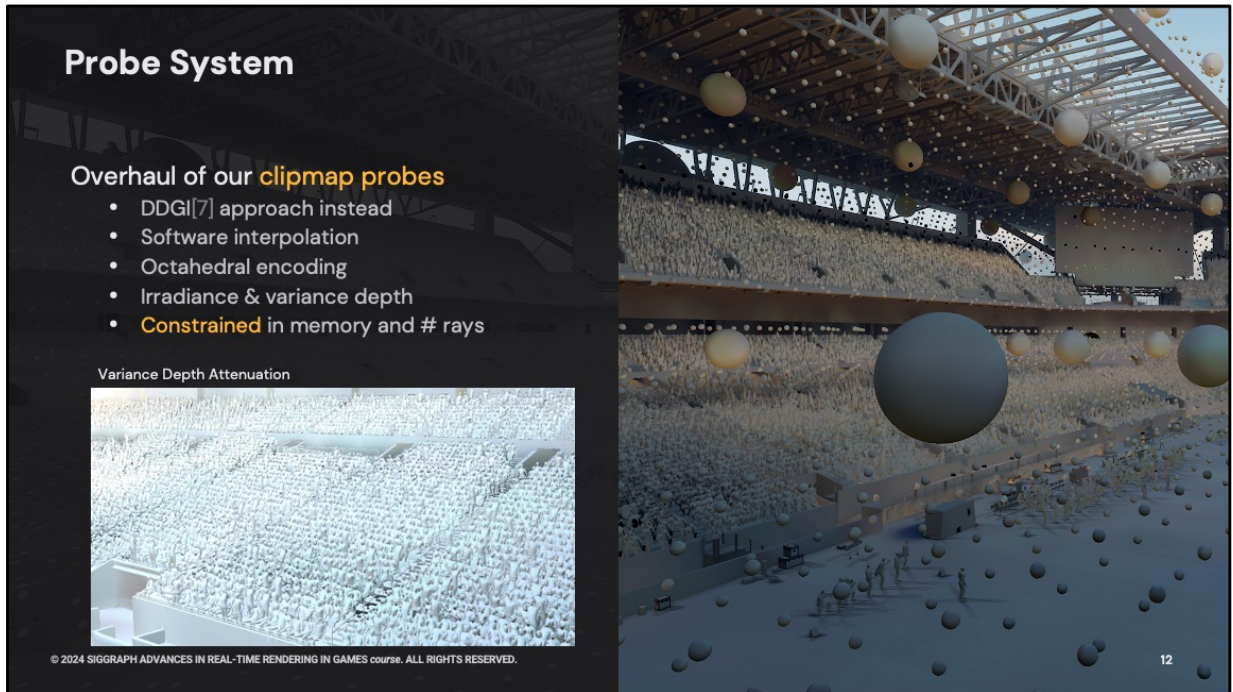
And finally, we apply both the surfel and the probe lighting (for certain probe lit objects) to the screen by producing

an indirect light buffer that we plug into our Deferred Tiled Lighting system.

Lighting calculations that don't go through our deferred pipeline will simply sample the probe's clipmap volume.

# Improvements

Probes & Surfels



Our clipmap probe system has been rehailed, where we used to do hardware trilinear interpolation of Spherical Harmonic coefficients, we decided to use an octahedral representation and software interpolation of the probes, very similar to DDGI. However, since we already store and trace surfels, we are restricted on memory and ray budgets, especially on console.

In this image you can see the crowd in CFB which samples our clipmap probes. Without the variance depth attenuation, we get dark probes leaking through the stands from below. And this is what it looks like with attenuation enabled.

## Probe System

### Target maximum of 100k probes

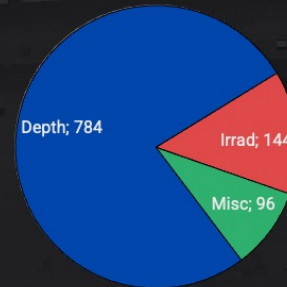
- We need up to 3 clips
- 32x32x32 resolution each

### Match Surfel memory = 100mb

- Means ~1kb per probe
- 4 bytes per component
- One atlas of 16x16 (incl. border)
- Need both irradiance & depth
- Could do 12x12 and 10x10
- Depth resolution more impactful

Depth: (14x14x4) = 784 bytes  
Irrad.: (6x6x4) = 144 bytes  
Misc.: filter data = 96 bytes

1024 Bytes Per Probe



When it comes to storage for our probes, we found that we need up to 3 clips to have coverage everywhere in the level, and to have enough density in the clip, we need about 100k probes, which is what we target as a maximum. Any gameteam can tweak the number of levels and the density, but our maximum target is 100k.

To match the memory budget we have for our Surfel system, we need to fit our probes in 100mb storage, which means we have about 1kb available per probe. Note that we have about 4 bytes per component for both irradiance and depth. If we had just one atlas we could fit 16x16 which includes the 2x2 border that is required for bilinear filtering.



However, we need an atlas for both irradiance and depth, so we could use a 12x12 and 10x10 atlas but we found that the depth resolution is more significant.

For these reasons we settled on a 14x14 resolution for the variance depth, including the 2x2 border. And a 6x6 resolution for irradiance, this gives us 4x4 with 16 directions.

The remaining 96 bytes we have are used for filtering data and other misc. data.

## Probe System

### Match Surfel rays = 100k rays

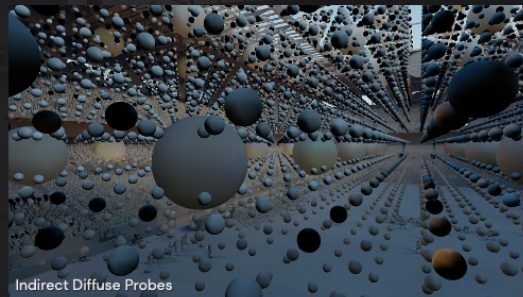
- 1 ray per probe per frame?
- 4x4 irradiance is 16 directions

### Update ~6k probes per frame

- One clipmap per frame
- Assign priorities per probe
  - + new probe locations
  - - probes with no nearby geo
- Update highest priority ones

### Improve responsiveness

- Drive hysteresis with MSME[8]



When it comes to our ray budget, we have 100k rays that we can dispatch, which means we can only afford 1 ray per probe per frame. However, since we do inline convolution of our octahedral irradiance, we need at least 16 rays for this to work properly.

This leaves us with having to amortize updates over the total probe count, which comes down to about 6k probes we can raytrace each frame.

We achieve this by considering one clipmap level per frame, and assigning priorities per probe therein. Probes that have updated locations, for example when the player moves, get higher prio. Probes that do not see any nearby

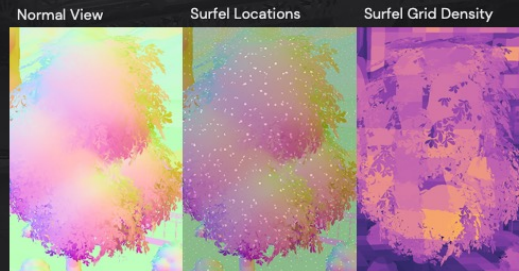
geometry, get lower priority. Finally, we simply update the 6k probes with highest priority which allows us to stay within our ray budget.

This image comparison shows the full set of probes versus the prioritized probes per clipmap level.

Doing so introduces some update latency, so, in order to improve the responsiveness when lighting changes, we drive the probes hysteresis (blend factor with historical data) using the same Multi-Scale Mean Estimator we have for our surfels. Note that we have 1 set of statistics per probe, which means we lack directional information, but this works out fine for us.

## Surfel Issues

- Visual inconsistencies
  - Incorrect normal/tangent basis
- Performance degradations
  - Excessive spawning of surfels



Accurate reconstruction from depth insufficient  
Export **vertex interpolated normal** during gbuffer

We did not overhaul our surfel system, but we did run into some interesting problems I would like to cover.

We had visual inconsistencies as well as random performance degradations which can both be attributed to the inconsistency of the normal tangent basis between our surfels and the actual surface they represent.

On this row of images you can see a tree that has a lot of variation in both depth and shading normals, which causes our coverage heuristic to excessively spawn surfels, resulting in high densities in our acceleration grid. We tried reconstructing the geometric normal from depth, but in cases where the depth plane cannot be faithfully

reconstructed, we would either end up spawning a surfel with a bogus normal, or reject spawning surfels at all. Since neither of these approaches worked out well enough, we decided to export the vertex interpolated normal during gbuffer laydown.

As you can see, this results in fewer surfels being spawned while maintaining the same coverage as well as fewer high-density entries in our acceleration grid.

## Surfel Issues

- Visual inconsistencies
  - Incorrect normal/tangent basis
- Performance degradations
  - Excessive spawning of surfels
- **Self-intersections and leaking**
  - Precision of surfel ray origin



Bias based on depth-buffer precision during spawn  
Store **unique bias** per surfel

When tracing a ray from a surfel origin, we need to add a small bias (or offset) to avoid intersection with the geometry the surfel spawned upon due to limited numerical precision.

In this underground garage, some surfels have positions that are very close to the ceiling. Our previous ray offset behaved too conservatively and allowed some rays to escape to the outside, which leads to light leaking, as you can see in this image.

We realized that determining this numerical error bound is non-trivial since we reconstruct surfel positions using the view-matrix and depth-buffer which are only known

during the frame in which each surfel is spawned.

Since the depth-buffer is not linear with respect to world-space units, we end up with a reconstruction error that is unique per surfel. Therefore, we need store this error per surfel, and as such avoid skipping too much space when tracing rays from their surfel origin, as you can see in this fixed version.

## Surfel Issues

- Visual inconsistencies
  - Incorrect normal/tangent basis
- Performance degradations
  - Excessive spawning of surfels
- Self-intersections and leaking
  - Precision of surfel ray origin
- **Dark results and popping**
  - Missing surfel coverage



Coverage is amortized over frames  
Sample probes when below a certain threshold

The last issue I'd like to discuss shows up when surfel coverage is missing. This is a bit of an extreme example, where the camera moved and discovered new surfaces on the left side on the image. Since this image represents the first few frames of discovering these surfaces, we have only spawned a select number of surfels since we amortize this process over multiple frames.

Previously we already addressed this issue by having our surfel grid provide an average fallback per gridcell, but this still produces missing irradiance for grid cells that don't have a surfel yet either. Instead, we now rely on the probes to give us an initial estimate when surfel coverage is low. This works well since probes have coverage



effectively everywhere, and we don't need extra storage or gpu cycles for a fallback anymore.

# Optimizations

Compute & Ray Tracing

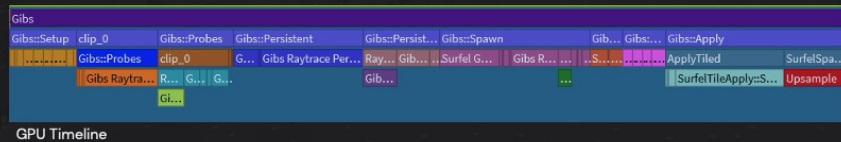
## Compute Performance

### GPU driven pipeline

- > 50 compute dispatches
- Including 6 ray tracing

### General optimizations

- RDNA compute (see [9])
- Surfel loop scalarization (see [10])
- Improved binning efficiency (bonus)
- Tile based surfel/probe apply



When it comes to performance, we are mostly concerned with compute and hardware rt workloads since GIBS is GPU driven. We dispatch over 50 compute shaders each frame, and only 6 of those are issuing ray intersections. While RT dispatches are certainly spending most of our GPU cycles, we did spend a significant amount of time making sure all our compute shaders performed well on RDNA, which is the gpu architecture used on our target console platforms.

One such optimization that saves us gpu time in several dispatches each frame is scalarization of our surfel iteration loops. Evaluating surfels involves loading a lot of structured data, for which we get a significant reduction in

data traffic by utilizing the scalar memory path where possible.

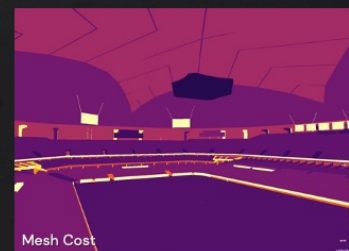
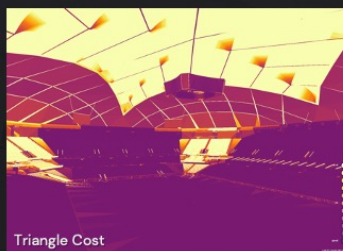
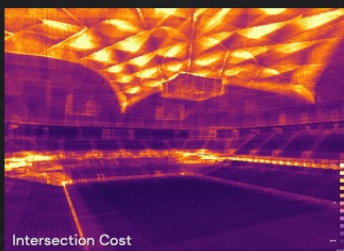
We also need to bin surfels into our 3d acceleration structure twice per frame, once for the transformed persistent surfels, and once more to include any newly spawned surfels.

And finally, we classify screen tiles to determine which texels need surfel lighting and which ones need probe lighting, allowing us to only run the respective shader code.

## Ray Tracing – Performance

### Acceleration structure quality

- Tools and visualizations
- Geometry splitting and merging
- Axis-aligning meshes
- Offline BLAS -15%
- Rebraiding -5%



© 2024 SIGGRAPH ADVANCES IN REAL-TIME RENDERING IN GAMES course. ALL RIGHTS RESERVED.

20

When it comes to performance for ray tracing specifically, we start by looking at the quality of our acceleration structures. We added debug tools and visualizations to help diagnose cases that are particularly bad for the efficiency of ray traversal. On the left you can see a debug view of the inside of a stadium where the heatmap encodes the relative cost of ray traversal per pixel, observe that the cost of evaluating intersections on the ceiling is more costly compared to other regions in the stadium. To help pinpoint the exact asset that causes this degradation, we have two additional viewmodes that show the estimated cost per triangle and per mesh.

With the ability the diagnose issues and validate changes, we

introduced several optimization stages to our offline mesh pipeline. We can automatically split large geometry and merge multiple small geometries together to improve the quality of our ray tracing acceleration structures. Similarly, mesh data can be rotated to improve alignment with the local axes, further improving BLAS quality.

Some platforms allow us to precompute the Bottom Level Acceleration Structure offline, we observe a 15% improvement to ray traversal when using this for static geometry.

Similarly, rebraiding can be used to improve ray traversal further, we compute a set of heuristics for each mesh offline, and based on an adjustable threshold we allow additional, or fewer instances of those meshes to be rebraided. Doing this selectively adds another 5% improvement on top.

## Ray Tracing – Scene Complexity

### Reduce scene complexity

- Only opaque geometry
- No alpha-tested materials
- **Simplified** material albedo

*Radiosity material still manual, we have a system for automatic diffuse which we intend to use in the future.*



Ok, that said, important to note is that for the sake of estimating irradiance, we only consider opaque, and non alpha-tested geometry in our acceleration structures for ray tracing. Lastly, we use a simplified diffuse albedo instead of evaluating the entire material graph for each primary ray intersection.

To give you an idea, in this image we can see what the rasterizer produces while the view of the ray tracer looks like this. As you can see, not all geometry is accounted for, for example, the vegetation is all missing from the ray tracing view, and materials are limited to just diffuse albedo, simplified down to one color per mesh-material subset.

Note that this workflow is still manual, but we have developed a system to help automate this which we intend to use in the future.



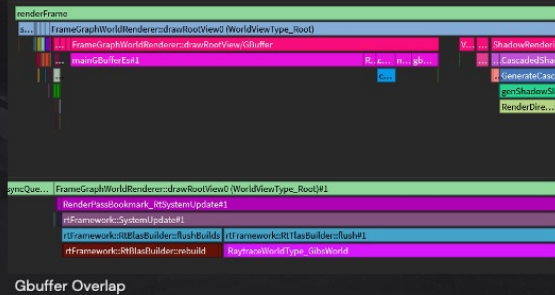
## Ray Tracing – Shader Efficiency

### Shader improvements

- Inline tracing in compute
- Async tracing during shadows
- **Overlap** AS build with Gbuffer

### HW-utilization is still a challenge

- Split primary and secondary rays
- Indirect dispatch & compaction
- **Reduce ray iteration #**



Next, we look at ways to improve the efficiency when traversing rays. We moved our ray tracing calls to compute and use inline tracing much like dxr 1.1. Each ray tracing dispatch uses indirect arguments which allows us to compact our workload while allowing to saturate the gpu's resources. The async queue may be used for tracing when the frame allows for overlap with a geometry stage.

We always overlap our BLAS and TLAS building in async with our gbuffer laydown, after which we can spawn surfels and trace them immediately.

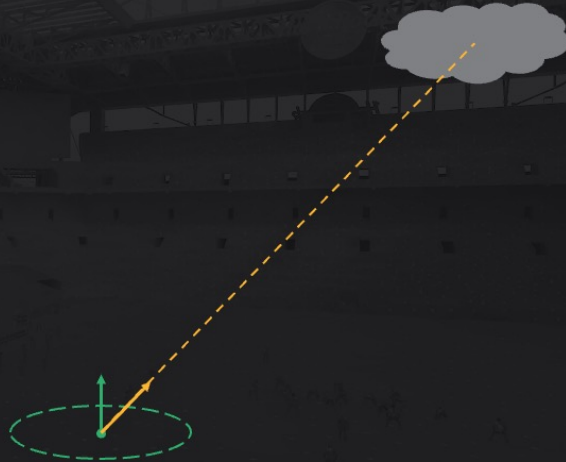
All this considered, it is still a challenge to keep the gpu utilization high for the entire duration of our ray tracing

workloads.

## Ray Tracing – Separate Ray Types

### Primary ray

- Originating from surfel/probe center
- Sample sky on ray miss

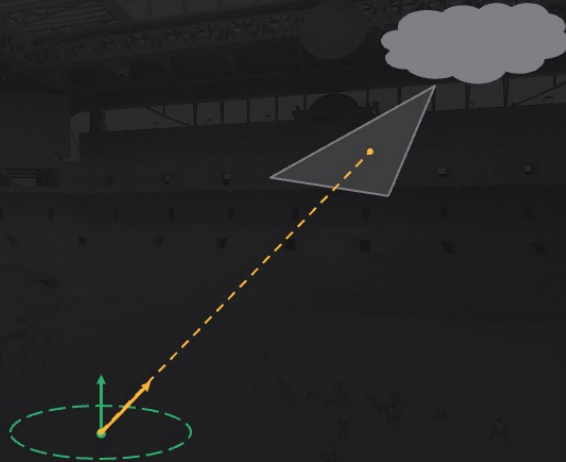


When it comes to ray traversal specifically, we looked into ways of reducing the number of iterations that need to be evaluated for each of our two distinct ray types. We have primary rays that originate from a surfel or probe center and are traced into the scene. If no geometry is intersected, we simply sample our environment lighting.

## Ray Tracing – Separate Ray Types

### Primary ray

- Originating from surfel/probe center
- Sample sky on ray miss
- Otherwise, find **closest hit**
- Return distance, normal and material id



Otherwise, if we did find an intersection along our ray, we find the closest hit and the following 3 attributes; there's the distance to the hit, the geometric normal of the intersected triangle (which we get from from an intrinsic on console) and the associated material id used to later look up the simplified diffuse albedo.

## Ray Tracing – Shader Efficiency

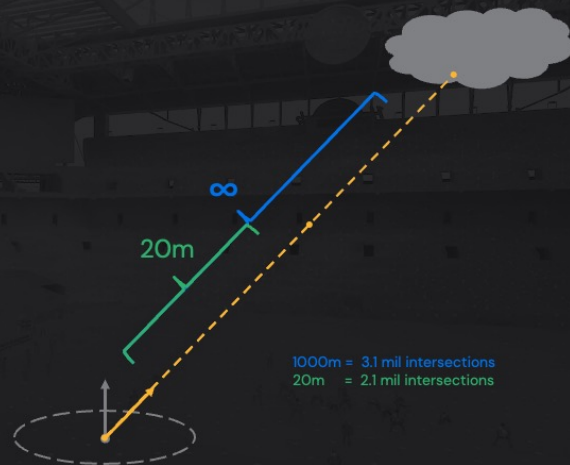
### Primary ray dispatch

- Technically “infinite” length
- Ray intersection has **no upper-bound**
- **Long-running** wavefronts

### Enforce an upper-bound

- Ray iteration limiting
- Reduced ray length (20m)
- Combine both

→ terminating a ray requires **fallback**



The problem with these primary rays is that their length is technically infinite. We limit the length to 1000 meters, but we still don't know the amount of ray intersections that will have to be evaluated for any of the rays in our list. Even when binning based on ray properties, we do not know which rays will be most expensive and thus cannot make sure to dispatch them early. This results in bubbles of execution by single rays causing their wavefront to be long-running while all other lanes might be inactive meanwhile, resulting in poor utilization of the hardware units.

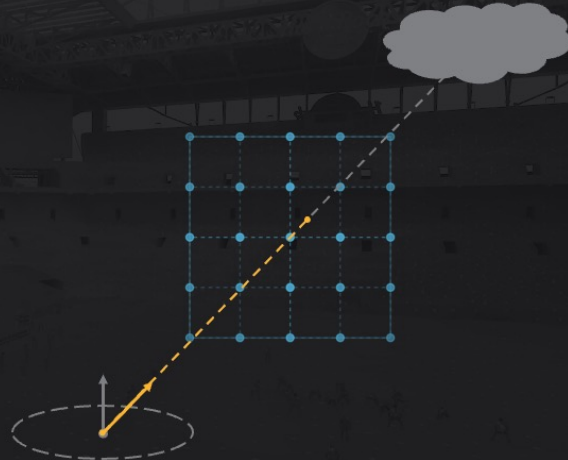
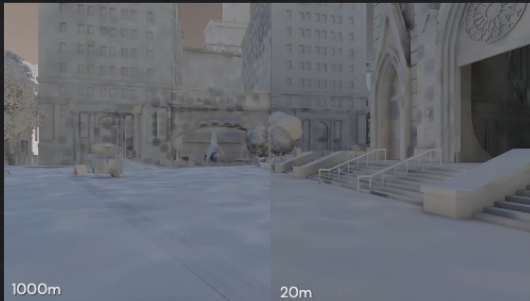
For these reasons, we allow to enforce an upper bound on the number of iterations by setting a conservative

maximum as well as shortening our ray length to e.g. 20 meters. Doing so, however, requires us to rely on a fallback to provide an estimate of the radiance that our ray carries.

## Ray Tracing – Shader Efficiency

Instead of sampling sky after tracing,  
sample probes after a short distance

→ fewer ray iterations, improved quality\*



© 2024 SIGGRAPH ADVANCES IN REAL-TIME RENDERING IN GAMES course. ALL RIGHTS RESERVED.

26

We actually have an irradiance volume available for us to fallback onto when tracing from surfels. When we trace a short ray and reach the endpoint without finding an intersection, we can lookup the nearest clipmap probe and sample its irradiance using the direction of our ray.

Doing this is certainly biased since our probes do not have infinite resolution (and are storing irradiance at this point), but we actually get a significant improvement to our convergence rate on top of limiting the number of ray intersection we need to evaluate.

This can be observed in the video, where the left side shows surfels tracing with a 1000-meter ray length and thus no

fallback, compared to just 20 meters with fallback on the right side.

The inside of this building is particularly challenging for convergence since most of the indirect diffuse lighting is coming from the outside. Our clipmap volume already has coverage inside of this building, long before any surfels are spawned.

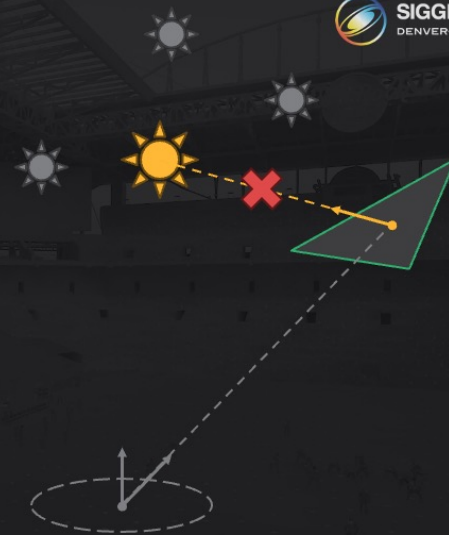


## Ray Tracing – Shader Efficiency

### Secondary ray

- Originates from **primary** intersection
- Estimate direct light **visibility**
  - Select N candidate samples (reservoir)
  - Trace 1 shadow ray
  - Sample shadow map instead

→ **significant reduction** in # shadow rays



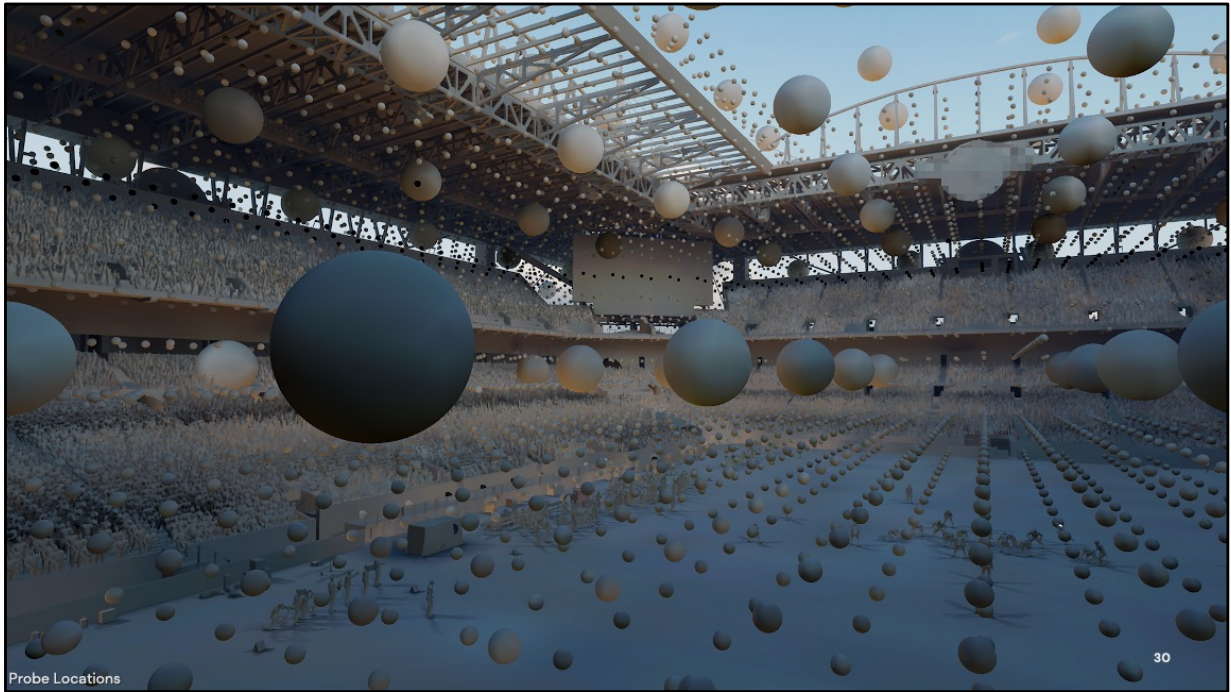
For secondary rays, which happen when such primary ray did not miss and instead found the closest surface. We a trace a ray from that point to estimate the direct light contribution. To do this efficiently, we go over N number of candidates using reservoir sampling, and we select one light sample to shoot a shadow ray towards. Instead of reducing the number of iterations for these secondary “shadow” rays, we instead look to reduce numbers of rays need to be traced. When the selected light source has a shadow map available, we can tap that instead of tracing a ray, resulting in a significant reduction of ray intersection requests.

# Results

Quality & Performance



Here's a set of screenshots captured in a stadium in College Football 25. This view shows the indirect diffuse lighting produced by GIBS, dynamically during runtime.



These are the probe locals which are used to sample indirect diffuse lighting on the crowd as well as the players on the pitch.



Surfels are used to provide indirect diffuse lighting on all static geometry such as the stands and stadium geometry.



Direct diffuse lighting goes on top.

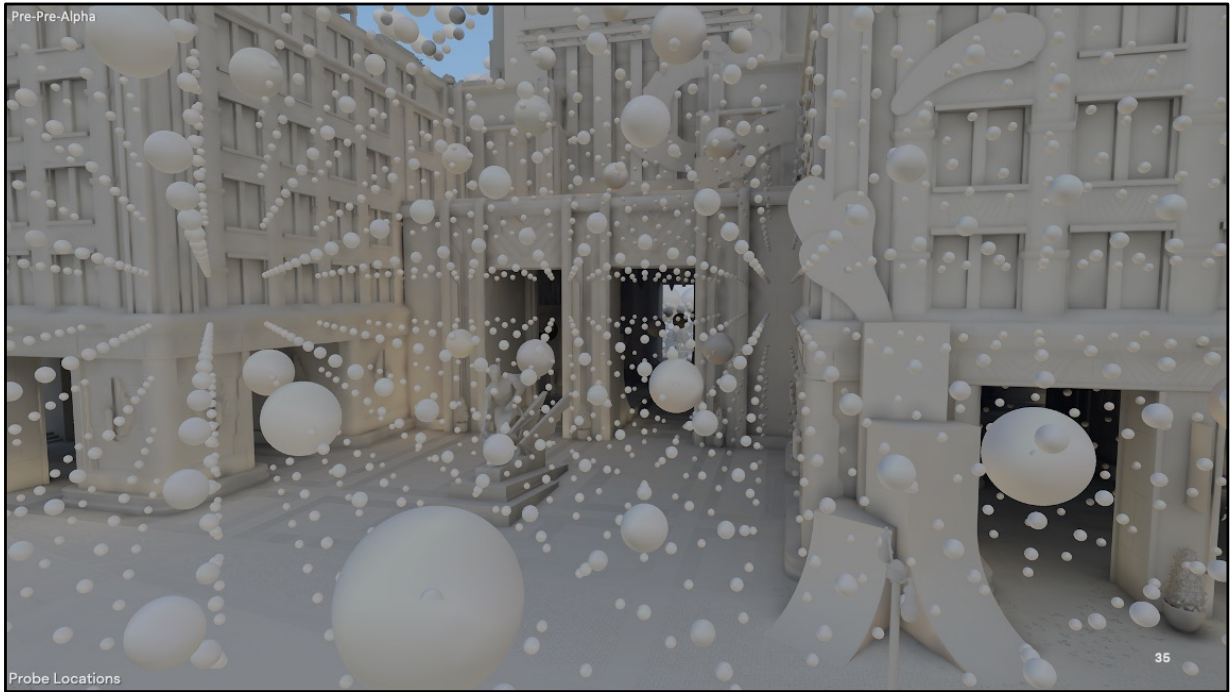


And this is the final image.



And this shows the same term in Skate





Probes here are generally not used much directly, and behave more as a cache for surfel rays and surfel coverage fallback.



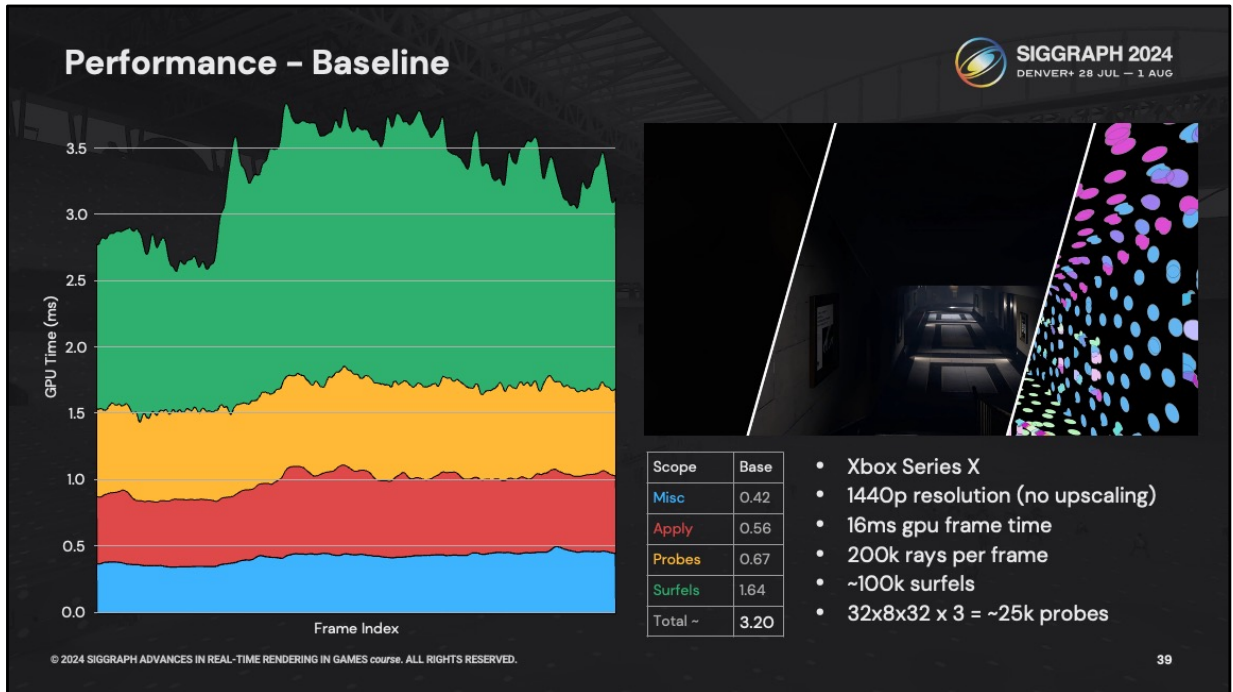
These are the surfel locations



Direct diffuse ontop



And the final image.



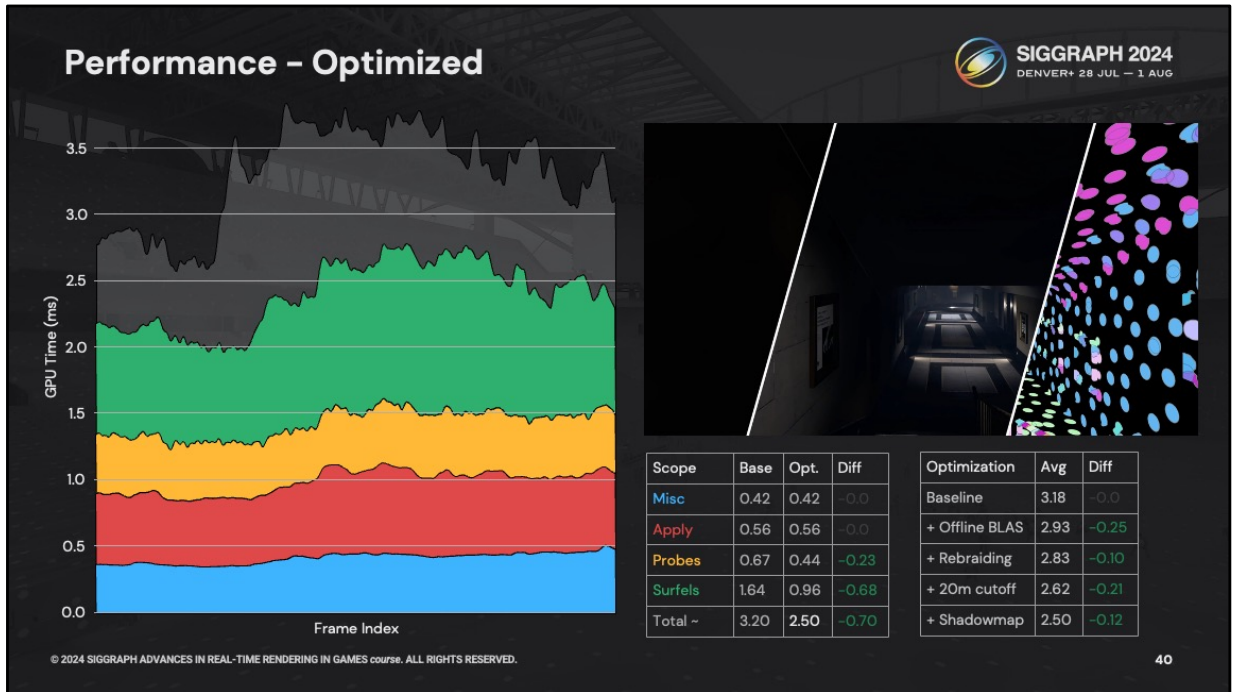
When it comes to performance, we have a camera flythrough on Skate, which we ran on XBOX at 1440p resolution without any upscaling. We target 60fps and allow for a maximum of 200k rays per frame. During this run we have about 100k surfels alive at any one frame, we constantly spawn and recycle surfels. When it comes to probes, we used 3 clipmaps with reduced density in height, totaling around 25k probes.

This graph shows a portion of this flythrough with some of our optimization disabled as a baseline. We spend about 0.5ms on compute work that is not directly related to ray tracing. Another 0.5ms is spent each frame to apply all surfels to the main view at quarter rate and upscaling to

1440p. Each frame we update a portion of our probes, which takes 0.7ms with ray tracing inclusive. And finally, the biggest cost is related to spawning and raytracing surfels at 1.64ms. Surfel timings tend to vary a lot depending on how many surfels are active, and how many new surfels are spawned in any given frame.

All of GIBS' gpu time combined takes about 3.2ms on average in this run, with the highest mark hitting 4ms in this run.

Note that this is already a significant improvement over the 6ms average we had in 2021, although those timing were on PS5 and in a different game.



Looking at some of the optimizations I talked about, this table shows our baseline and the cumulative improvement of each of our additional optimizations.

Offline BLAS and selective rebraiding shave off 0.35ms together. Adding our ray limiting and shadowmap sampling shaves off another 0.33ms on top.

This brings us down to a total average gpu time of 2.5ms, without going over the 3ms mark during the entire run.

Looking at the graph again, we see that Misc. and Apply are unaffected by these optimizations, but Probes spend 34% less time tracing due to the shadowmap sampling, and

surfels about 60% less due to the ray shortening and fallback to probes. Notice also that there is much less variability in the surfel duration across frames.





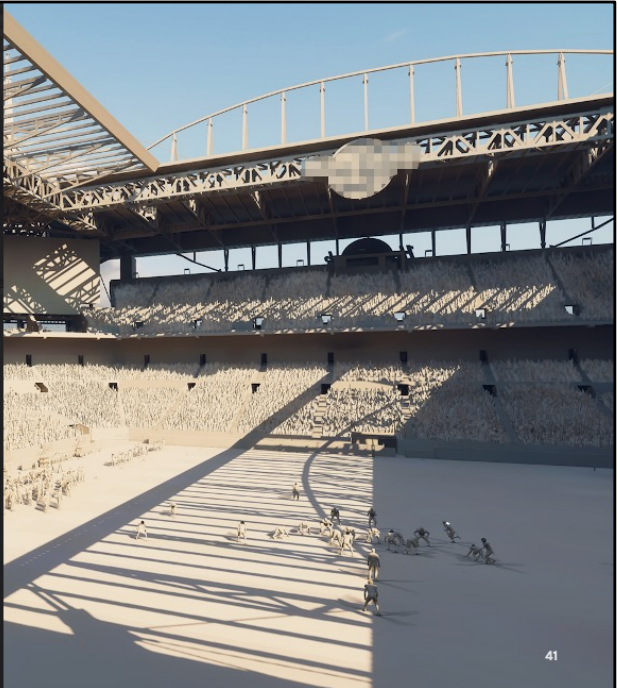
"Teapot in a stadium"

- Overlapping instances
- Few LOD's to fallback on
- Every ray is **expensive**

< 2ms during gameplay

- Geometry and lighting **fixed**
- **Halt irradiance integration**
- Only apply surfels & probes

© 2024 SIGGRAPH ADVANCES IN REAL-TIME RENDERING IN GAMES course. ALL RIGHTS RESERVED.



41

Some notes about performance in CFB25 specifically; due to the nature of their game, ray tracing instances have a lot of overlap and we generally do not have any lower-detail meshes available to fallback on to improve ray traversal efficiency. This means that every ray we trace is expensive.

CFB runs at 60 frames per second during gameplay, and 30fps outside of gameplay. GIBS is fully active and dynamic during 30hz mode and fits into budget snugly. However, during 60hz we only have a 2ms budget which means we need to compromise here.

The CFB team was able to play into some of their game's

constraints for this, were all stadium geometry and lighting is determined once during level load. This allowed them to expend more surfel and ray budget during the 30hz preload sequence and halt any irradiance updates when going into 60hz gameplay. During gameplay, GIBS is in a frozen state and we only apply the surfel and probe lighting to the screen each frame, which allows us to stay under 2ms of gpu time.

# Conclusion

## What did not work out

### Ray binning no longer a win

- Fewer rays per frame
- Overhead from sorting
- Compaction still important

### Single ray per probe

- Write into accumulation atlas
- Convolve into “frontbuffer” once all texels are valid
- Ultimately decided to go with inline convolution

### Stochastic surfel apply

- Attempt to improve apply performance
- Select n surfels per texel using restir
- Halfres and upscale still faster

Some things we already had or tried out the passed 3 years did not end up working out.

In our 2021 talk we showed how ray binning improved our ray traversal performance. This is no longer the case due to having fewer rays with less redundancy as well as more optimized traversal. Worth noting though, is that running a compaction pass based on ray properties (e.g. needs shadow ray), is still important to keep hardware utilization high.

Early on, when overhauling our probe system, we tried running with just 1 ray per probe per frame by accumulating the intermediate radiance in a “backbuffer”

atlas and later convolve into the “frontbuffer” atlas. However, we ultimately settled for inline convolution mostly due to the added memory cost of double-buffering of the depth atlas and the overhead copying probe data around every frame.

We also tried different approaches to apply our surfels to the screen, since this is a constant cost each frame. The complexity of doing so depends on the number of surfels as well as the screen resolution. One experiment we ended up discarding was a stochastic approach where we’d only evaluate a fixed number of surfels per texel using restir. However, in the end, the naïve approach is still faster at halfres (quarter rate) with a cheap spatial upscale, especially when using scalarized loads for all surfel data to reduce bandwidth.

## Summary

### Turn-key dynamic global illumination

- Skate: higher fidelity, consistency with user generated content
- CFB: no fallback, saves time authoring allowing for more content

### 60hz gameplay on console

- Skate: fully dynamic and  $< 4\text{ms}$  at 60hz
- CFB: halt updates during gameplay to stay  $< 2\text{ms}$  at 60hz

### Fully-featured under constrained budget

- Memory under 500mb at full functionality
- Maximum of 200k rays per frame

To conclude, we have shown that GIBS allows for fully dynamic diffuse global illumination. In Skate this allows higher fidelity lighting and consistency with user generated content. On College Football 25, GIBS saves a lot of time authoring, especially in their combinatorial explosion of many stadiums with multiple times of day.

Previously, GIBS was considered a 30hz feature on console, but after all our optimizations we managed to unlock 60hz, fully dynamic in Skate by staying under 4ms, and while CFB has only a 2ms budget, we can deactivate parts of the system during gameplay.

GIBS is fully featured on our lowest spec platforms by

keeping our surfel and probe system under 100mb and 100k rays per frame. Specifically, on XBSS we fit within 500mb for the entirety of GIBS including all raytracing resources.

## Special thanks

**Advances:** Natalya Tatarchuk & Peter-Pike Sloan

**GIBS, SEED, and the RT Teams:**

- Filipe Amim
- Colin Barré-Brisebois
- Petter Edblom
- William Donnelly
- Jon Greenberg
- Henrik Halén
- Kyle Hayward
- Alexander Polya
- Darrin Stewart
- Joel Svensson
- AJ Weeks
- Alan Wolfe

The Frostbite rendering team

**Skate:**

- Nathaniel Johnson
- Bob Peet
- Gary Steinke

**College Football:**

- Ken Ball
- Richard Burgess-Dawson
- Jay Goodman
- Travis Robbins
- Ishaan Singh
- Josh Verrier

The rest of EA Sports Rendering and Lighting

Thank you Natalya for hosting us a second time to talk about GIBS. Peter-Pike for reviewing our slides and providing valuable feedback.

This work involved a lot of talented engineers working on ray tracing across Frostbite and SEED. Thank you to our partners at Skate, as well as College Football, and many other individuals across EA Sports Rendering and Lighting.



See CFB's talk

 SIGGRAPH 2024  
DENVER+ 28 JUL — 1 AUG



**EA SPORTS College Football is Huge: Delivering a Triple-A Sports Game at Scale**

The Sporting Life, Sunday, 28 July 9:00am

© 2024 SIGGRAPH ADVANCES IN REAL-TIME RENDERING IN GAMES course. ALL RIGHTS RESERVED.

46

If you are interested in learning more about College Football 25's production, which includes details on their use of GIBS, make sure to check out the talk they gave here at SIGGRAPH 2024.

Questions?

That's the end of my talk, thank you very much for listening,  
I'm happy to answer any of your questions regarding GIBS.

## References



- [1] Yuriy O'Donnell. 2018. Precomputed Global Illumination in Frostbite. In Game Developers Conference.
- [2] Diederik Apers, Petter Edblom, Charles de Rousiers, and Sébastien Hillaire. 2019. Interactive Light Map and Irradiance Volume Preview in Frostbite. Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs (2019), 377–407.
- [3] Sam Martin. 2010. A Real Time Radiosity Architecture for Video Games. SIGGRAPH Advances in Real-Time Rendering (2010).
- [4] Kleber Garcia, Andreas Lindqvist, and Andreas Brinck. 2020. "Hustle by Day, Risk it all at Night" The Lighting of Need for Speed Heat in Frostbite. In Special interest group on computer graphics and interactive techniques conference talks, 1–2.
- [5] Johan Andersson and Colin Barré-Brisebois. 2018. Shiny pixels and beyond: Real-time raytracing at SEED. In Game Developers Conference.
- [6] Henrik Halen and Kyle Hayward. 2021. Global Illumination Based on Surfels. In Proc. ACM SIGGRAPH Symp. Interactive 3D Graph. Games, 1399–1405.
- [7] Zander Majercik, Jean-Philippe Guertin, Derek Nowrouzezahrai, and Morgan McGuire. 2019. Dynamic diffuse global illumination with ray-traced irradiance fields. Journal of Computer Graphics Techniques 8, 2 (2019).
- [8] Tomasz Stachowiak. 2018. Stochastic all the things: Raytracing in hybrid real-time rendering. SEED, Digital Dragons (2018).
- [9] Advanced Micro Devices. 2024. AMD GPU Open RDNA™ Performance Guide. Retrieved from <https://gpuopen.com/learn/rdna-performance-guide/>
- [10] Michal Drobot. 2017. Improved Culling for Tiled and Clustered Rendering. SIGGRAPH Advances in Real-Time Rendering in Games course (2017), 902–905.