Interactive Light Map and Irradiance Volume Preview in Frostbite

Diede Apers, Petter Edblom, Charles de Rousiers, and Sébastien Hillaire
Electronic Arts

Abstract

This chapter presents the real-time global illumination (GI) preview system available in the Frostbite engine. Our approach is based on Monte Carlo path tracing running on the GPU, built using the DirectX Raytracing (DXR) API. We present an approach to updating light maps and irradiance volumes in real time according to elements constituting a scene. Methods to accelerate these updates, such as view prioritization and irradiance caching, are also described. A light map denoiser is used to always present a pleasing image on screen. This solution allows artists to visualize the result of their edits, progressively refined on screen, rather than waiting minutes to hours for the final result using the previous CPU-based GI solver. Even if the GI solution being refined in real time on screen has not converged after a few seconds, it is enough for artists to get an idea of the final look and assess the scene quality. It enables them to iterate faster and so achieve a higher-quality scene lighting setup.

1 Introduction

Precomputed lighting using light maps has been used in games since *Quake* in 1996. From there, light map representations have evolved to reach higher visual fidelity [1, 8]. However, their use in production is still constrained by long baking times, making the lighting workflow inefficient for artists and difficult for engineers

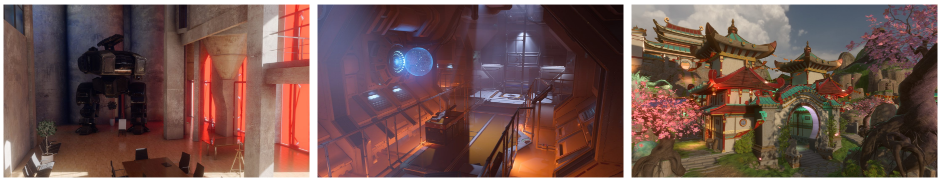


Figure 1: Three final shots from different environments, rendered by Frostbite using our GI preview system. Left: Granary. (Courtesy of Evermotion.) Center: SciFi test scene. (Courtesy of Frostbite, © 2018 Electronic Arts Inc.) Right: Zen Peak level from *Plants vs. Zombies Garden Warfare 2*. (Courtesy of Popcap Games, © 2018 Electronic Arts Inc.)

to debug. Our goal is to provide a real-time preview of diffuse GI within the Frostbite editor.

Electronic Arts produces various types of games relying on a wide range of lighting complexities: static lighting such as in *Star Wars Battlefront 2*, dynamic sunlight for time of day simulation such as in *Need for Speed*, and even destruction requiring dynamic updates of the GI solution from dynamic lights such as in the *Battlefield* series. This chapter focuses on the static GI case, i.e., unchanging GI during gameplay, for which Frostbite's own GI solver can be used [9]. See Figure 1. Static GI relying on baked light maps and probes will always be a good value: it enables high-quality GI baking, resulting in high-fidelity visuals on screen without needing much processing power. In contrast, dynamic GI, e.g., from animated lights, requires extensive offline data baking, has coarse approximations, and has costly runtime updates.

Light map generation is a conveniently parallel problem where each texel can be evaluated separately [3, 9]. This can be achieved using path tracing and integrated using Monte Carlo integration [5], where each path contribution can also be evaluated independently. The recent real-time ray tracing additions to DXR [11] and Vulkan Ray Tracing [15] make it convenient to leverage the GPU's massively parallel architecture to handle all the necessary computations such as ray/triangle intersection, surface evaluation, recursive tracing, and shading. The Frostbite GI solver is built on the DXR API.

This chapter describes the Frostbite path tracing solution built to achieve real-time GI preview [3]. Section 2 gives details about the path tracing solution used to generate light maps and irradiance volumes. Section 3 presents the acceleration techniques used to reduce the GI preview cost, e.g., using view/texel prioritization or direct irradiance caching. Section 4 describes when GI data are generated and invalidated based on artist interactions with the scene input, e.g., lights, materials, and meshes. Finally, Section 5 discusses the impact on accuracy and presents performance of the different acceleration components.

2 GI Solver Pipeline

This section discusses our GI solver for previewing light maps and irradiance volumes. First, Section 2.1 describes the input (scene geometry, materials, and lights) and output (light map data) of the GI solver and how they are stored on the GPU. Then, Section 2.2 gives an overview of all parts of the pipeline. Finally, Section 2.3 describes how the lighting computation is handled.

2.1 Input and Output

2.1.1 Input

- *Geometry:* The scene geometry is represented with triangular meshes. A unique UV parameterization is attached to each mesh for mapping them to light map textures. These meshes are usually simplified geometry, called *proxy meshes*, as compared to the in-game meshes, as shown in Figure 2. Using proxy meshes alleviates self-intersection issues caused by coarse light map resolution, a common situation due to memory footprint constraints.

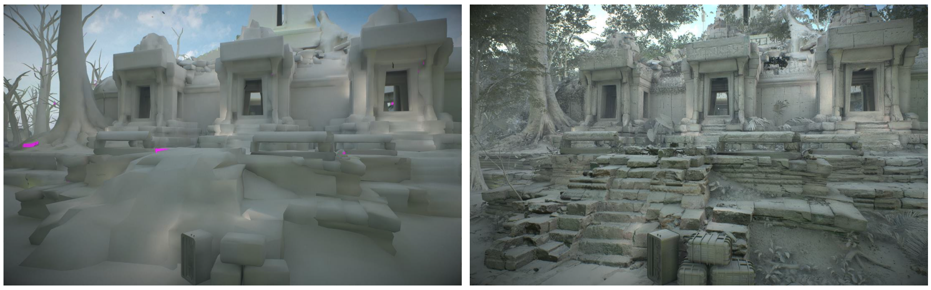


Figure 2: Light map applied to a scene in *Star Wars Battlefront II.* Left: light map applied to proxy meshes, against which GI is traced. Right: light map applied to final in-game meshes by projecting proxy UV coordinates and using normal mapping. (Courtesy of DICE, © 2018 Electronic Arts Inc.)

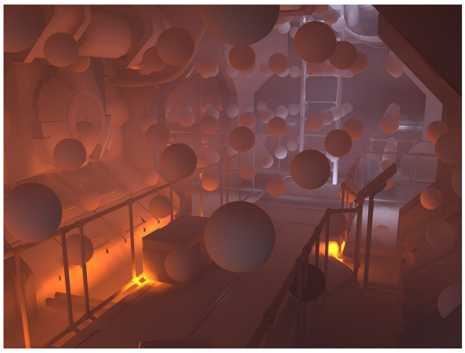
The UV parameterization can be done automatically through some proprietary algorithms or done manually by artists. In both case, the parameterization tries to mitigate texel stretching and texel crossing geometry, which can result in light leaks. Non-manifold meshes are divided into several charts, which are padded to avoid light bleeding across charts when the light map is bilinearly sampled at runtime. Multiple instances of a mesh share the same UV parameterization but cover different areas of the light map texture. If instances are scaled, by default their light map coverage will increase. Doing so helps to keep texel size relatively constant over a scene. Optionally, artists can disable this scaling per instance to conserve light map space.

- *Materials:* Each scene geometry instance has certain material properties: diffuse albedo, emissive color, and backface behavior. Albedo is used for simulating correct interreflections, while face orientation is used to determine how a ray should behave when intersecting the backface of a triangle. Since we are interested in only diffuse interreflections, the usage of a simple diffuse material model such as Lambert [10] is enough. Surfaces with metallic material are handled as if they were covered with a dielectric material. In such a case, the albedo can be estimated based on its reflectance and its roughness [7]. As a result, no caustics will be generated by our GI solver.

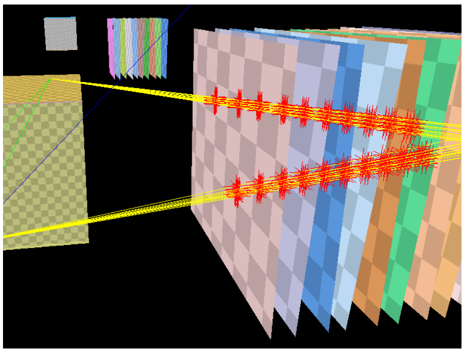
- *Light sources:* A scene can contain various types of light sources: local point lights, area lights, directional lights, and a sky dome [7]. Each light should behave as its real-time counterpart in order to have consistent behavior between its baked and runtime versions. The sky dome is stored in a low-resolution cube map.

- *Irradiance volumes:* In addition to light maps, the GI solver allows us to pre-visualize lighting for dynamic objects. They are lit by irradiance volumes placed into levels. See Figure 3(a). Each irradiance volume stores a three-dimensional grid of spherical harmonics coefficients.

The input geometry is preprocessed to produce *sample locations* for each light map texel. These world-space locations are generated over the entire scene's



(a) Irradiance volume visualization.



(b) Ray intersection visualization.

Figure 3: Debug visualizations inside the editor. (a) An irradiance volume placed into a futuristic corridor scene. This irradiance volume is used for lighting dynamic objects. (b) Visualization of shadow rays and intersections with transparent primitives. Yellow lines represent shadow rays, and red crosses represent any-hit shader invocations to account for transmittance.

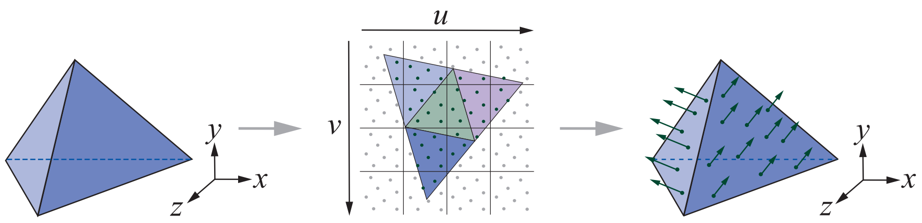


Figure 4: Left: geometry in three-dimensional space. Center: the same geometry unwrapped in UV space. Sample positions are generated in UV space using a low-discrepancy sample set covering the entire texel. Right: only valid samples intersecting the geometry are kept.

geometric surface. Each is used as the first vertex of a path when path tracing. Valid sample locations are produced by generating points within each texel's boundary in the light map. These points are then tested for intersection with the unwrapped geometry and transformed into world space using the instance transformation corresponding to the intersected primitive, as illustrated in Figure 4. Points without any intersections are discarded, and all valid sample locations are uploaded to the GPU for later use by the path tracing kernel. The algorithm proceeds in a greedy fashion, generating samples until (say) eight valid sample locations in a texel are found. The UV space is sampled using a low-discrepancy Halton sequence, whose sample enumeration covers the entire domain. In the case that the sequence does not contain any points that produce valid sample locations, e.g., a small triangle residing between points, we generate a sample by clipping the triangle to the pixel's boundary and by using its centroid to generate a new valid sample location. Additionally, using this algorithm, it is also possible that one point in UV space produces multiple sample locations. This can happen when the light-mapped geometry is overlapping in UV space, which is undesirable. The algorithm is resilient and allows for this.

The geometry in the scene is stored in a two-level bounding volume hierarchy (BVH) (DXR's acceleration structure [11]). The bottom level contains a BVH for each unique mesh. The top level contains all instances, each of which has a unique spatial transform and references a bottom-level BVH node. While this structure is less efficient during traversal than a single-level BVH, it simplifies scene update, which is a frequent operation during level editing. For instance, moving a mesh requires updating only the top-level instance transform matrix, instead of transforming the entire triangle soup stored in the bottom level.

2.1.2 Output

The GI solver produces several outputs, structured either into light maps or irradiance volumes. For light map data, instances are packed into one or several light map atlases, which are coarsely packed on the fly[1]:

- *Irradiance:* This is the main output of the GI solver. It describes the directional irradiance[2] for light maps or irradiance volumes. Usually, the runtime geometry is finer than the geometry used for baking and often contains detailed normals, e.g., with normal mapping, which are not taken into account at baking time. See Figure 2. At runtime, the directional irradiance allows us to compute the actual incoming irradiance for detailed normals. Several representations are supported, such as the average value, principal direction, and spherical harmonics [9].

- *Sky visibility:* This describes the portion of the sky visible from a given light map texel or irradiance volume point [7]. This value is used at runtime for various purposes, such as reflection blending or material effects.

- *Ambient occlusion:* This describes the surrounding occlusion for a given light map texel or irradiance volume point [7]. It is used at runtime for reflection occlusion.

2.2 GI Solver Pipeline Overview

The proposed pipeline aims to preview the final outputs as quickly as possible. Since computing fully converged outputs would likely take several seconds or minutes for a production-size level, the proposed pipeline refines the outputs iteratively, as seen in Figure 5. At each solving iteration, the following operations are done:

- *Update scene:* All scene modifications since the last iteration are applied, e.g., moving a mesh or changing a light's color. These inputs are translated and uploaded to the GPU. See Section 4.

[1]When atlasing the different instances' charts, some padding is added between charts to avoid interpolating between texels that are not adjacent in world space.

[2]Directional irradiance stores incident lighting in a way that the irradiance can be evaluated for a variety of directions.

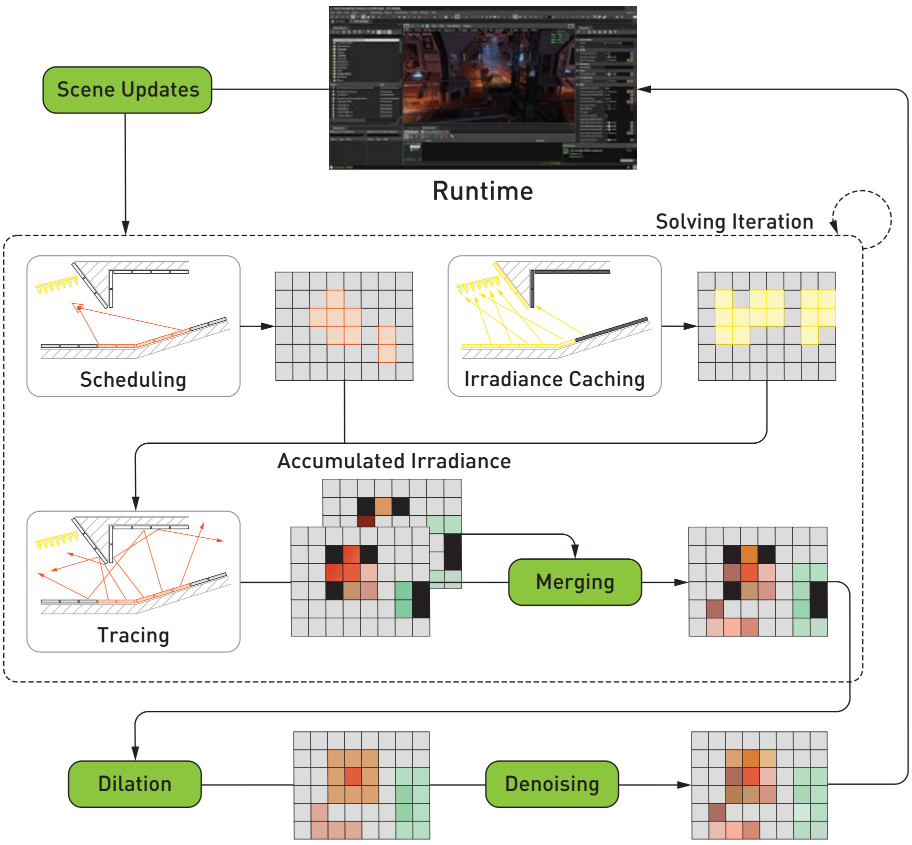


Figure 5: Overview of the GI solver pipeline. Light maps and irradiance volumes are updated iteratively. The camera viewpoint is used to prioritize texels that need to be scheduled for the tracing steps. Using an irradiance cache, the tracing step refines the GI data. The traced results are merged with those of previous frames, before being post-processed (dilated and denoised) and sent back to the runtime.

- *Update caches:* If invalidated or incomplete, the irradiance caches are refined by tracing additional rays for estimating the incident direct irradiance. These caches are used for accelerating the tracing step. See Section 3.3.
- *Schedule texels:* Based on the camera's view frustum, the most relevant visible light map texels and visible irradiance volumes are identified and scheduled for the tracing step. See Section 3.1.
- *Trace texels:* Each scheduled texel and irradiance point is refined by tracing a set of paths. These paths allow one to compute the incoming irradiance, as well as sky visibility and ambient occlusion. See Section 2.3.
- *Merge texels:* The newly computed irradiance samples are accumulated into persistent output resources. See Section 2.6.
- *Post-process outputs:* Dilation and denoising post-process passes are applied to the outputs, giving users a noise-free estimate of the converged output. See Section 2.8.

2.3 Lighting Integration and Path Construction

To compute the irradiance E reaching each texel, we need to integrate the radiance L incident to the upper hemisphere Ω weighted by its projected solid angle $\omega^{\perp}$:

$$E = \int_{\Omega_p} L d\omega^{\perp}. \tag{1}$$

Computing the incoming radiance L requires us to solve the light transport equation, which computes the outgoing radiance L based on an incoming radiance L_i and interacts with the surface material properties. In our case, we are interested in only the diffuse interreflection. For diffuse materials, with albedo ρ and emission L_e, this equation is

$$L(\omega) = L_e + \frac{\rho}{\pi} \int_{\Omega} L_i(\omega) d\omega^{\perp}. \tag{2}$$

Due to its high dimensionality, Equation 1 can be difficult to solve. Relying on stochastic methods, such as Monte Carlo, has proven to be a good fit for several reasons. First, the result is unbiased, meaning it will converge to the correct value $\mathbb{E}(E)$ with enough samples. Second, the end result can be computed in an iterative fashion, which perfectly suits our needs to display incremental refinement to artists. Finally, refinements for a given light map texel are independent and thus can run in parallel. Solving Equation 1 with a Monte Carlo estimator,

$$\mathbb{E}(E) \approx \frac{1}{n} \sum_{\zeta=0}^{n} \frac{L_\zeta}{p_{L_\zeta}}, \tag{3}$$

simply means that averaging n random evaluations L_ζ of this integral, weighted by its probability distribution function (PDF) p_{L_ζ}, will converge to the correct result. This is an extremely convenient property.

A simple way for evaluating Equation 1 is to construct *paths* composed of *vertices* connecting a target texel to a light source. Each vertex lies on a geometric surface, whose material properties, e.g., albedo, reduce the path's *throughput*. This throughput determines the quantity of light carried. We construct these paths iteratively from the texel to the light sources, as described by the kernel code in Listing 1.

```
Ray r = initRay(texelOrigin, randomDirection);

float3 outRadiance = 0;
float3 pathThroughput = 1;
while (pathVertexCount++ < maxDepth) {
    PrimaryRayData rayData;
    TraceRay(r, rayData);

    if (!rayData.hasHitAnything) {
        outRadiance += pathThroughput * getSkyDome(r.Direction);
        break;
    }

    outRadiance += pathThroughput * rayData.emissive;

    r.Origin = r.Origin + r.Direction * rayData.hitT;
    r.Direction = sampleHemisphere(rayData.Normal);
    pathThroughput *= rayData.albedo * dot(r.Direction,rayData.Normal);
}

return outRadiance;
```

Listing 1: Kernel code describing a simple light integration.

The algorithm in Listing 1 outlines a way of integrating the irradiance for each texel. This simple solution is rather slow and does not scale well. In the following, we describe a few traditional techniques that can be used to improve performance:

- *Importance sampling:* It is more effective to importance-sample the upper hemisphere according to the projected solid angle instead of a uniform distribution, as grazing directions have little contribution compared to more vertical ones [10].

- *Path construction with random numbers:* The initial two vertices of each path are carefully built to reduce the variance of the estimated irradiance. First, *spatial* sample locations, which map texel sub-samples onto meshes, are pre-generated using a low-discrepancy Halton sequence as described in Section 2.1. This ensures that the full domain is uniformly sampled. Second, the *directional* samples are also sampled using a low-discrepancy Halton sequence. However, to avoid correlation issues between directional samples of adjacent spatial samples, a random jitter is added to offset directions. This construction ensures the full four-dimensional domain, spatial and angular. Using an actual four-dimensional sequence, rather than two independent two-dimensional sequences, will sample this space more efficiently, but was omitted from our first implementation for simplicity. Subsequent path vertices are built using uniform random values for constructing directional samples. Sample positions are determined by the ray intersection.

- *Next event estimation:* Building a path until it reaches a light source is inefficient. The likelihood of reaching a light source becomes smaller as the number of lights (or their sizes) decreases. In the limit, when a scene has only local point lights, it is impossible to sample them with a random direction. One simple approach to solve this issue is to explicitly connect each vertex of a path to light sources and evaluate their contribution. This is known as *next event estimation*. By doing so, we artificially build multiple paths using existing sub-paths. This simple, yet efficient, scheme improves convergence drastically. To avoid double contribution, light sources are not part of the same structure as regular scene geometry.

All the above techniques can be summarized in the simplified kernel code in Listing 2.

```
Ray r = initRay(texelOrigin, randomDirection);

float3 outRadiance = 0;
float3 pathThroughput = 1;
while (pathVertexCount++ < maxDepth) {
    PrimaryRayData rayData;
    TraceRay(r, rayData);

    if (!rayData.hasHitAnything) {
        outRadiance += pathThroughput * getSkyDome(r.Direction);
        break;
    }

    float3 Pos = r.Origin + r.Direction * rayData.hitT;
    float3 L = sampleLocalLighting(Pos, rayData.Normal);

    pathThroughput *= rayData.albedo;
    outRadiance += pathThroughput * (L + rayData.emissive);

    r.Origin = Pos;
    r.Direction = sampleCosineHemisphere(rayData.Normal);
}

return outRadiance;
```

Listing 2: Kernel code describing the lighting integration.

2.4 Light Sources

A scene can contain a set of point and area light sources. When a path is constructed, surrounding local lights, directional lights (e.g., sun), and any sky dome are evaluated at each vertex of the path (next event estimation):

- *Local point lights:* The irradiance evaluation is trivial. Its intensity is computed based on its distance to the shading point and its angular falloff [7]. While point light intensity decays inversely to the square distance, artists can reduce their influence by tuning the light bounding volume. The received intensity increases as a light gets closer to a shaded surface and can approach infinity. To avoid this problem, we use a minimal distance, set to one centimeter, between the shading point and the light.

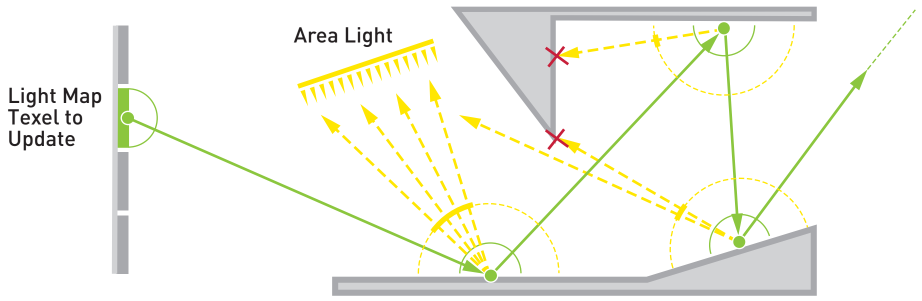


Figure 6: Irradiance evaluation of a texel. For this purpose, a path is constructed in green. At each vertex of this path, the direct lighting is evaluated by casting rays toward the area light (a.k.a. next event estimation). The number of samples for each light is proportional to its subtended solid angle. If no geometry is intersected, the sky dome radiance is evaluated.

- *Area lights:* The irradiance evaluation implies integrating the visible surface for each light. To do so, samples are generated onto the visible part of the light sources [10] and connected to the current path vertex. Sampling an area light source can require many samples for resolving not only its irradiance contribution but also its visibility, which creates soft shadows. To amortize the path construction, multiple samples are cast for each area light source, proportionally to their subtended solid angle. These samples are stratified over the integration domain to build a good estimate of their contribution. See Figure 6. Sample positions on light sources are generated with a low-discrepancy Hammersley sequence because the number of samples, based on the solid angle, is known up front. This sequence is randomly offset at each path's vertex to avoid spatial correlation, which could result in shadow replicates.
- *Directional lights:* Irradiance is sampled at each vertex. Even if at runtime the directional light is evaluated as a small disk area light, the coarse light map resolution makes the small disk evaluation unnecessary.
- *Sky dome:* Irradiance is sampled when the generated direction does not hit any geometry. For efficient evaluation one could importance-sample for sky lighting at each vertex of a path, for instance, with the alias method [18].

2.5 Special Materials

In addition to regular diffuse albedo, materials can emit light or let the light pass through them:

- *Emissive surfaces:* Geometry instances with *emissive surfaces* can emit light, making any regular geometry a potential light source. During path construction the surface emission is evaluated at each vertex. While this method produces the correct result on average, it requires many samples, especially for small emissive surfaces. To address this issue, emissive triangles can be added to our light acceleration structure; see Section 3.2. These emissive surfaces would then be part of the regular direct lighting evaluation.

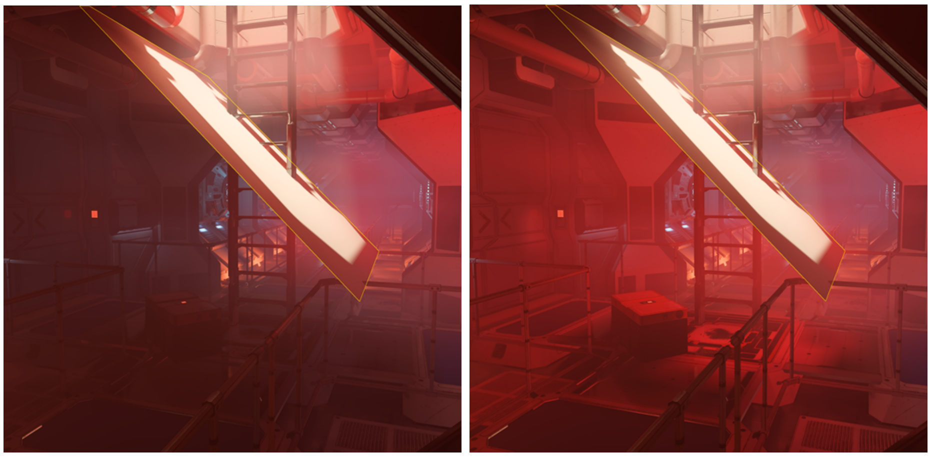


Figure 7: Scene containing a plane whose material is translucent. Left: translucency is disabled. No light is diffused through the surface. Right: translucency is enabled, allowing the light to be diffused into the scene. The transmitted light gets a red tint in this case.

- *Translucency:* Instance material properties can describe translucent surfaces by specifying their backface behavior. For such surfaces, light will be diffusely transmitted from the other side of the geometry, as shown in Figure 7. The quantity of light transmitted is driven by the surface albedo and a translucency factor. Based on this quantity, we stochastically select if the path should be transmitted or reflected when it hits such a surface. Direct light evaluation is done on the selected side. Due to this process, during the direct lighting evaluation, if a translucent surface lies between a light and a path's vertex, no light will be transmitted. This path's vertex will be shaded only if the path can be extended to connect it with the translucent surface.

- *Transparency:* During next event estimation (see Section 2.3), a ray is traced toward the light source. Intersecting geometry with transparent materials will attenuate the visibility. Using DXR, this effect is realized by multiplying visibility by transmittance in an any-hit shader. Geometry that does not contain any transparent materials can be flagged as `D3D12_RAYTRACING_GEOMETRY_FLAG_OPAQUE`. When a ray encounters this type of geometry, the ray is terminated. For geometry that does contain transparent materials, `D3D12_RAYTRACING_GEOMETRY_FLAG_NO_DUPLICATE_ANYHIT_INVOCATION` must be used to avoid any double contributions. Figure 3(b) shows how these any-hit shader invocations are triggered for only transparent geometry.

2.6 Scheduling Texels

This section details the first and third part of our pipeline, as depicted in Figure 5. Texels are scheduled in a separate pass, prior to being traced. This scheduling

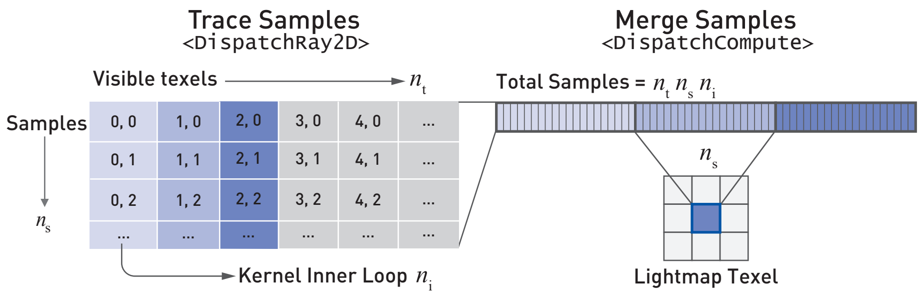


Figure 8: Left: dispatching strategy for the tracing kernel. Right: samples stored in one large buffer being accumulated and merged into a single light map texel.

pass runs multiple heuristics to determine if a certain texel should be processed. Examples of such heuristics are view prioritization (see Section 3.1) and convergence culling (see Listing 4). *Convergence culling* analyzes the texel's convergence and avoids scheduling texels that are already converged enough. Once a texel is determined to be scheduled, we select a sample on its surface and append it to a buffer.

Once all sample locations are appended to that buffer, they are consumed by the second part of the pipeline. Each sample location can be evaluated multiple times, depending on our performance budgeting system (see Section 2.7). Figure 8 illustrates our dispatching strategy. To ensure a large enough number of threads to fully saturate the available hardware resources, we schedule each sample location multiple times, n_s = number of samples. Their contributions are deposited in one large buffer organized in buckets belonging to each texel, n_t = number of texels. Additionally, we have an inner loop in our kernel that allows us to trace multiple primary rays from each dispatched thread, n_i = number of iterations. The total number of samples is then $n_\mathrm{t} \times n_\mathrm{s} \times n_\mathrm{i}$. The value of n_t depends on the result of view prioritization and is currently bound by 1 sample per 16 pixels in screen space. Both n_s and n_i are scaled by the `sampleRatio` in Listing 3.

Merging of multiple samples into one texel happens in a compute shader. Note that we do not have to worry about the same texel being scheduled multiple times, as view prioritization ensures that each visible texel is scheduled only once. Finally, the output is combined with that from previous frames.

2.7 Performance Budgeting

Path tracing performance can be unpredictable and cause hitches in frame rate. To provide artists with a smooth workflow, we implemented a *performance budgeting* system that tracks the time spent path tracing on the GPU. Based on a target frame budget (in milliseconds), the system will adaptively scale the number of samples being traced to align with the performance budget. See Listing 3.

```
const float tracingBudgetInMs = 16.0f;
const float dampingFactor = 0.9f;                    // 90% (empirical)
const float stableArea = tracingBudgetInMs*0.15f;    // 15% of the budget

float sampleRatio = getLastFrameRatio();
```

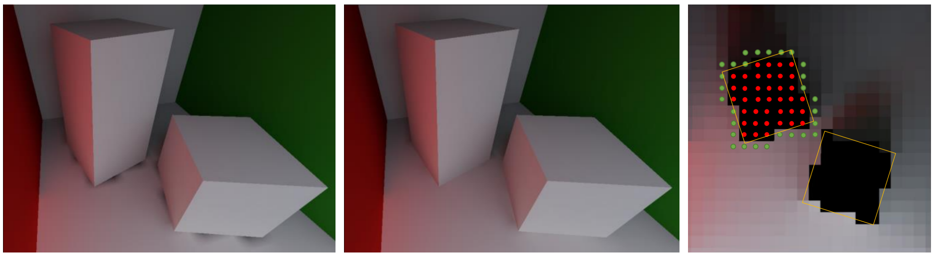


Figure 9: Dilation filter applied to a light-mapped Cornell box. Left: texels covered by geometries have invalid irradiance, due to paths hitting inner geometry, which is flagged as invalid. Middle: a dilation filter is applied for removing invalid texels. Right: top view of the scene showing the valid texels (green) and the invalid texels (red) for one of the objects.

```
float timeSpentTracing = getGPUTracingTime();
float boostFactor =
        clamp(0.25f, 1.0f, tracingBudgetInMs / timeSpentTracing);

if (abs(timeSpentTracing - tracingBudgetInMs) > stableArea)
    if (traceTime > tracingBudgetInMs)
        sampleRatio *= dampingFactor * boostFactor;
    else
        sampleRatio /= dampingFactor;

sampleRatio = clamp(0.001f, 1.0f, sampleRatio);
```

Listing 3: Host code for computing the sample ratio used by the performance budgeting system.

2.8 Post-Process

The output of the GI solver is built progressively. Therefore, the final result is likely not completed before several seconds have passed. However, to give a sense of instant control to the artist, the presented output needs to be representative of its final version as soon as possible. Three types of issues need to be addressed:

- *Black texels:* These happen when either a given texel has not received any irradiance sample yet or samples hit backface surfaces marked as invalid, as shown in Figure 9. To alleviate both cases, a dilation filter is applied to the data presented to the user, but not to the progressively built version so as to not alter the final output. This dilation filter ensures that all texels are valid for bilinear lookup at runtime. A partially covered texel, i.e, one in which certain sample locations have not received any lighting, do not need any dilation, because their final irradiance value is computed by averaging only valid sample locations. Doing so avoid a darkening effect at geometry junctions.

- *Noisy texels:* These are a manifestation of undersampling when integrating values with a stochastic method. The amount of noise reduces over time, because with additional samples the average value converges to the expected mean. To present meaningful values to the user, we use a denoiser algorithm whose goal is to predict an estimate of the converged mean value. See Figure 10. To do so, we use a variance-guided filter [12]. The main idea is to track texels' variance and use this information for adapting the strength of neighborhood filtering. This filter is applied in light map space and uses instances' chart IDs to act like an edge-stopping function. Doing so avoids filtering across nonadjacent geometry in world space. See Figure 11. This hierarchical filter looks sparsely at surrounding texels with increasing distance in multiple passes. Doing so allows it to extract the mean value, even in the presence of several frequencies of noise. Since this filter runs in light map space, it is preferable to have a relatively consistent texel density over the scene, and more filtering passes will be needed if the texel density increases. As the variance reduces, the luminance filter will shrink, as well as the spacing between samples, converging smoothly to the actual average value.

The variance of the mean[3] of each texel is computed using Welford's online variance algorithm [19]. The variance is not updated at each iteration, but after tracing a certain bucket of samples per texel. The size of this bucket increases at each update due to the quadratic convergence of Monte Carlo integration. Our bucket size is initially set to 12 samples and is doubled for each successive iteration. Using this variance information, the standard error [20] indicates when the mean has reached a certain confidence interval. We use a confidence interval of 95%, at which point a texel is considered as fully converged. For example code, see Listing 4.

- *Chart seams:* Seams can arise between light map charts, as the lighting can be different on two texels adjacent in world space but distant in light map space. This is a known issue with light maps. Our current GPU GI solver tool does not address this issue yet, and we instead rely on our existing CPU-based stitcher [4].

```
float quantile = 1.959964f; // 95% confidence interval
float stdError = sqrt(float(varianceOfMean / sampleCount));
bool hasConverged =
        (stdError * quantile) <= (convergenceErrorThres * mean);
```

Listing 4: Kernel code describing variance tracking.

3 Acceleration Techniques

The GI solver relies on several acceleration techniques to reduce the cost of each refinement step. The goal when using these techniques is to converge faster to the final GI solution with a minimum of approximations, while presenting a coherent result on screen to the user (see Section 5.2).

[3]The variance of the mean is not the same as the variance of accumulated samples within a texel. The latter is related to the subpixel information, while the former is related to the convergence of the estimated value.

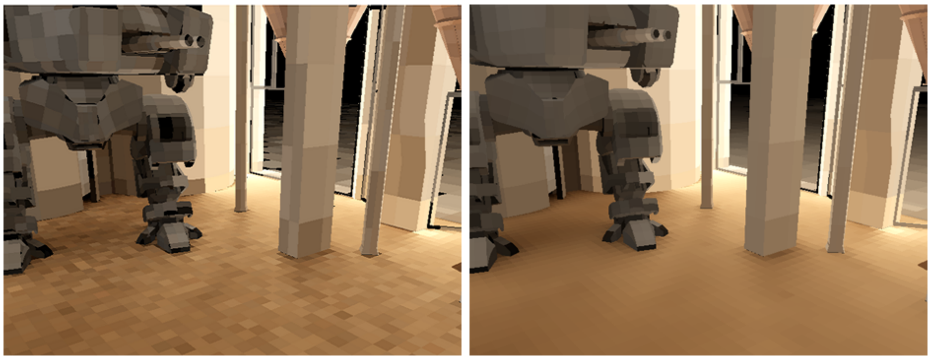


Figure 10: Left: light map data visualized after the merge and dilation operations. Right: light map data visualized after the denoiser step. In both pictures bilinear filtering is disabled to emphasize the noise reduction.

3.1 View Prioritization

Rendering every texel in the light map is unnecessary when the scene is being observed from only one point of view. By prioritizing texels that are directly in view, we can achieve a higher convergence rate where it matters most for artists. This is referred to as *view prioritization* and is evaluated during our texel scheduling pass, as described in Section 2.6.

To schedule each texel in view at least once, we compute the visibility over multiple frames, as depicted later in Figure 19. Each frame we trace n_{v} rays from the camera into the scene as described in the following pseudocode. When multiple visibility queries schedule the same texel, care needs to be taken when merging this texel. We use atomic operations to ensure that a texel is only scheduled once each frame.

```
Search for visible texel from camera:
Stratify near plane
for each stratum do
    Generate random point in stratum
    Construct ray through point on near plane
    if Intersect light-mapped geometry then
        Load geometry attributes
        Interpolate light map UV coordinates using barycentrics
        Determine texel index from UV coordinates
        Schedule visible texel
```

Since the variance of each texel is tracked during the lighting integration (see Section 2.8), this information is used for scheduling only unconverged texels, i.e., texels whose variance is higher than a certain threshold.

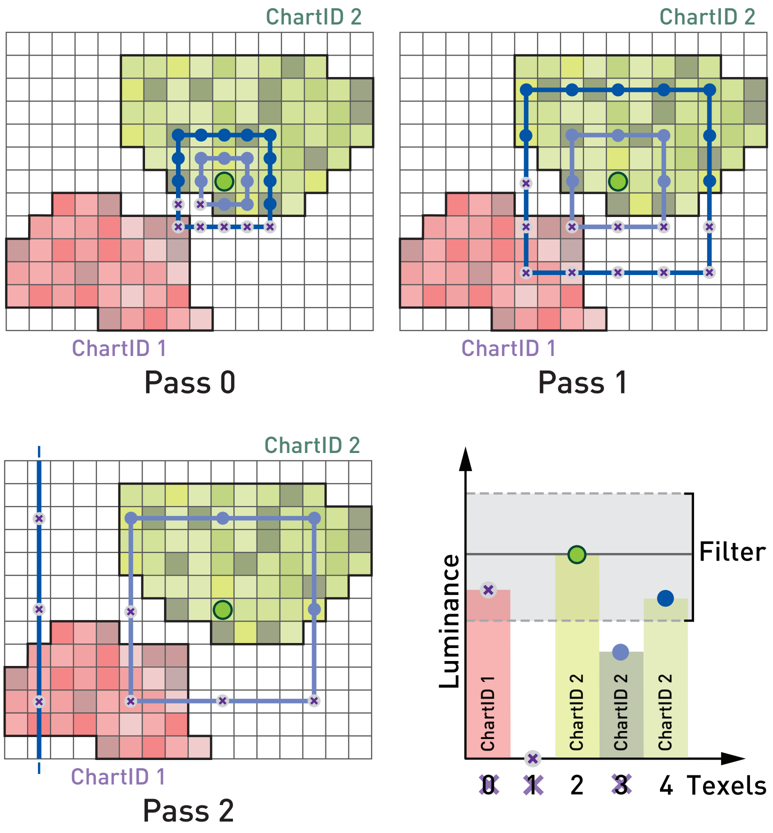


Figure 11: Three passes of the Á-Trous (i.e., with holes) denoiser [12] working in light map space. At each pass, a 5×5 kernel is evaluated (green, light blue, and blue bands). The gathered samples are spread farther and farther apart to filter the noise across multiple frequencies. The luminance value and chart ID are used as edge-stopping functions to avoid overblurring and light bleeding across geometries. In the texel histogram on the bottom right, texel 2 (green dot) is currently being filtered. Only texel 4 (blue dot) contributes to the filtered value, as other texels either have a different chart ID than texel 2 or are out of the valid luminance range.

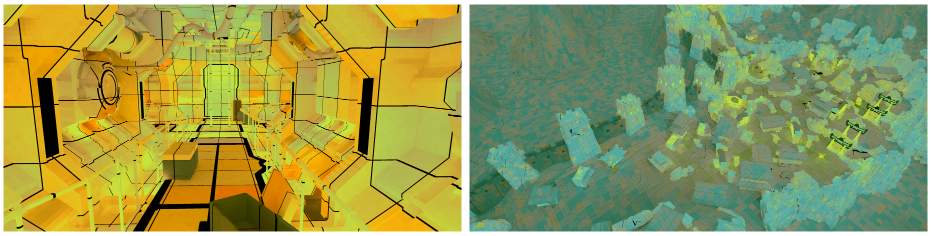


Figure 12: Number of local light queries per texel: blue is none, orange is most dense. Left: many local lights in a corridor. Right: sparse distribution of local lights with some hash collisions in an outdoor environment.

3.2 Light Acceleration Structure

A level can contain a large number of lights. Casting shadow rays toward each of them for the purpose of next event estimation is costly (see Section 2.3). This needs to be done for each vertex of each path. To counter this issue, an acceleration structure is used to evaluate only lights that potentially interact with a given world-space position.

The acceleration structure is a spatial hash function [16]. The world space is divided in an infinite uniform grid of axis-aligned bounding volumes of size $S_{3\mathrm{D}}$. Each of these boxes maps to a single entry of the one-dimensional hash function of entry count n_{e}. When a level is loaded, the table is built once, accounting for all the lights, and uploaded to GPU memory. Each bounding volume's hash entry contains a list of indices, one for each light intersecting with it. Thankfully, each volume does not contain all lights. This is made possible by the fact that, for the sake of performance, lights in Frostbite are bounded in space according to their intensity [7].

The bounding volume of size $S_{3\mathrm{D}}$ is computed based on the average size of the lights' bounding volumes, divided by a constant factor. This constant factor, 8 by default, can be used to reduce the cells's size. The hash function is created with n_{e} set to a large prime number, e.g., 524,287. In the case of a hash collision, it is possible for multiple volumes, far from each other in the world, to map to a single entry of the table. This case can create false-positive lights; however, this has not been observed as a problem so far. Results using this light acceleration structure are visible in Figure 12. Please refer to the original paper [16] for more details about this topic.

3.3 Irradiance Caching

Section 2 presents how path tracing is used to estimate light maps and volumes storing irradiance [9]. For each vertex of a traced path, the surrounding local light sources are sampled, resulting in an estimate of direct lighting. For each of these events, a ray is traced to assess the visibility of each light. However, a scene can have many lights, making this process, which is run for each vertex of a path, expensive. Furthermore, paths are built independently, so there is a high chance that paths will diverge quickly. Divergence can result in a higher overall cost of the process due to incoherent spatial structure queries and rays causing scattered

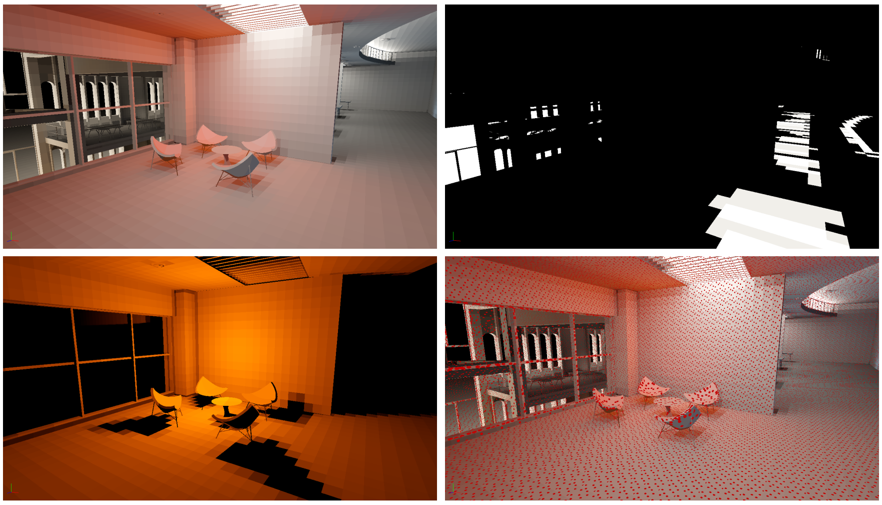


Figure 13: Visualization of the irradiance caches on the Granary scene, lit by local lights, a direction light, and a sky dome. Top left: indirect lighting stored into a light map. Top right: the directional light irradiance cache. Bottom left: the local light irradiance cache. Bottom right: red dots showing where the irradiance cache is computed.

memory access with higher latency. To accelerate this step, we use irradiance caching.

3.3.1 Direct Irradiance Cache Light Maps

The idea behind *irradiance caching* is to store the incoming light on a surface patch (irradiance) into a structure that is fast to query. A complete description of irradiance caching is available from Křivánek et al. [6]. Frostbite's GI solver stores *direct irradiance* in light map space according to a one-to-one mapping with the GI light map parameterization of the scene. See Figure 13. A cache is built for each type of light source: local lights, the sun, and the sky dome. This separation is important, as it avoids rebuilding the local lights cache when only the sky has changed (see Section 4.2). Once computed, these direct irradiance cache textures can then be fetched at each vertex of a path to accumulate the direct irradiance instead of explicitly sampling lights (see Section 2.3).

As illustrated in Figure 14, the direct irradiance evaluation from lights, previously achieved using many rays (Figure 6), is now replaced by a few simple texture fetches, leveraging hardware-accelerated bilinear filtering. Thus, it has less impact on performance than incoherent tracing toward many lights interacting with each vertex of a path. These cache textures can easily have their resolution scaled up to increase accuracy. Otherwise, large texels will miss finer details resulting from complex occlusions and the final GI will look blurry. Increasing resolution is also a way to reduce the light map blurring resulting from the texture bilinear filtering used when sampling the cache.

Using direct irradiance caches results in large performance wins, which are presented in Section 5.1. Timings are shown in Table 3. Convergence is greatly

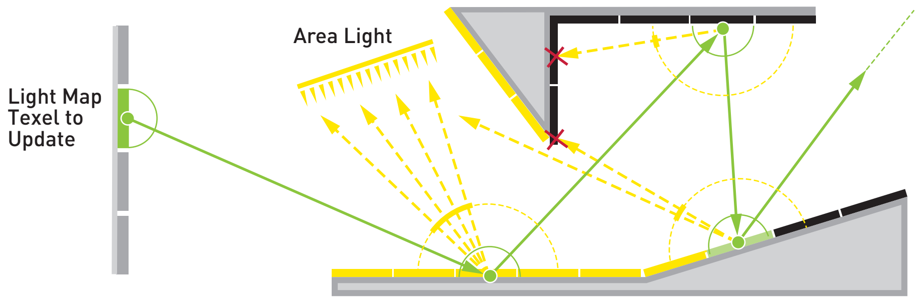


Figure 14: When using path tracing for next event estimation, to accumulate direct irradiance, a visibility test ray (dashed yellow lines) must be cast toward each light source for each vertex of a path (green dots). When using direct irradiance caching, a single texture fetch for each vertex can give the direct irradiance result for every light in the scene. Caching removes all the traced yellow rays toward each light source and thus accelerates the path tracing kernel. Direct irradiance cache texels are represented as small rectangles on the surfaces, in yellow for irradiance $E > 0$ or black if $E = 0$.

improved, as depicted in Figure 17, depending on the scene, light setup, and viewpoint.

3.3.2 Cache Update Process

Every cache is invalidated when a scene is opened in the editor. When the sun is modified, only the direct sunlight irradiance cache is invalidated, with similar limitations for the sky dome and local light caches. When a cache is invalidated, its update process starts. Taking the local light cache as an example, the following process is followed for each update round:

1. A number n_{ic} of samples are chosen for each texel of the light map cache.
2. For each sample, the direct irradiance is evaluated using ray tracing toward every light source.
3. For area lights, uniform samples are chosen over the light surface. The number of samples is adapted for each area light as a function of its subtended solid angle [10]: the larger it is, the more samples it gets to properly resolve its complex visibility.
4. The samples are accumulated in the direct irradiance light map.

The same process is applied for the sun and sky dome lights. Both are also uniformly sampled by distributing samples on the sun's disk and the sky dome hemisphere. Since the number of samples that are going to be taken for each source is known, low-discrepancy Hammersley sampling is used. In the case of the sun, only $n_{\mathrm{ic}}^{\mathrm{sun}} = 8$ total samples are used, assuming sharp shadows. In the case of local lights, up to $n_{\mathrm{ic}}^{\mathrm{L}} = 128$ samples are used; more samples are needed to integrate area light irradiance and soft shadows. In the case of the sky dome, $n_{\mathrm{ic}}^{\mathrm{sky}} = 128$ samples are taken. A high number is needed, since a physically based sky simulation can result in high-frequency variations, especially when the sun is at the horizon [2]. During these steps, we ignore translucent surfaces (see Section 2), because for this interaction to happen, a path has to traverse a surface. However, the direct irradiance cache stores only non-occluded, i.e., directly visible, contributions to irradiance. Since each update round uses $n_{\mathrm{ic}} = 8$ samples per cache, each irradiance cache update is considered done after 1, 16, and 16 iterations, respectively, for the sun, local lights, and sky dome caches. These values are settings available for users to tweak according to their preference and the game on which they are working. For instance, some games or levels could rely mostly on sun and sky dome lighting instead of local lights, thus more samples would need to be allocated for sun and sky sampling.

3.3.3 Future Improvements

Indirect irradiance cache In addition to direct light caching, accumulated indirect irradiance can be used to shorten path sampling. As described in Section 2.8, each texel convergence is tracked. When a texel is fully converged, its value can be used as an estimate of incident indirect lighting. By doing so, the path can end there, reducing the amount of computation.

Emissive surfaces As of today, emissive surfaces are not taken into account in any of the direct irradiance caches. In Section 2.4, we mention that such surfaces could be converted to triangle area lights. Such area lights could be sampled to populate a direct irradiance cache light map dedicated to emissive surfaces. This cache would then only be updated when a mesh is moved or a material property affecting emitted surface radiance is modified by an artist.

4 Live Update

4.1 Lighting Artist Workflow in Production

Frostbite-based game teams have dedicated artists responsible for designing the lighting and thereby setting the mood of a scene. There are several editing operations that lighting artists use to accomplish this. The most common include placing lights sources, adjusting them, moving objects, and changing their materials. Other operations include modifying light map resolution for individual objects and switching objects from being lit by a light map or by irradiance volumes. These two operations mainly aim to adjust memory usage to fit within a certain light map budget.

As stated in Section 1, the offline Frostbite GI solver uses CPU-based ray tracing. Historically, getting feedback on the work performed in the game editor with regard to global illumination takes minutes to hours when using a CPU-based solver. The transition to GPU-based ray tracing allows the time range for this process to become seconds to minutes.

The raw performance when taking a scene from nothing to one with reasonable global illumination is important in this context. The general acceleration techniques used are described in Section 3. The next section focuses on

		Irradiance Cache		
Invalidates:	Light Map	Sky	Sun	Light
Mesh	×	×	×	×
Sky	×	×		
Sun	×		×	
Light	×			×
Material	×			
Resolution		×	×	×

Table 1: When an input is changed (left column), GI states may need to be reset (upper row). Those states are marked with a cross.

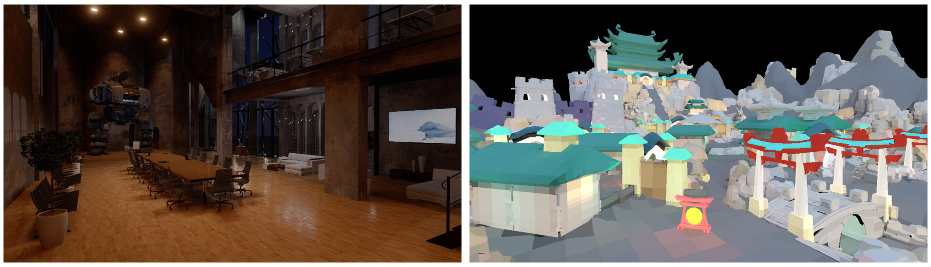
optimizations made possible when dealing with a specific modification done to the scene by the artist.

4.2 Scene Manipulation and Data Invalidation

For a scene, light maps, irradiance volumes, and irradiance caches are considered states, which are updated after each iteration of the GI preview. When a light, mesh, or material is updated, the current GI states become invalid. However, it is vital to not present misleading results to the artist. A straightforward solution is to clear all the states and restart the lighting solution integration. At the same time, the user experience relies on not doing excessive invalidation that would cause unnecessary computations while restarting the lighting solution, which results in noisy visuals. The goal is to invalidate the smallest possible data set while still presenting valid results.

Table 1 presents what state is reset after an input has been updated. It shows that the light map (or irradiance volumes) are invalidated in all cases, except when scaling the resolution of a mesh light map. Scaling resolution without invalidation is possible because, during GI preview, each mesh has a dedicated light map texture. We use per-mesh instance textures together with bindless techniques to sample all these textures when rendering a scene. Material changes will also affect indirect light, but at least here irradiance caches are still valid since they include only direct light, before it is affected by a surface material. One can also notice that irradiance caches need to be invalidated only when the associated light source type is modified.

The real bottleneck come from meshes. Each time a mesh is added, updated, transformed, or removed, light maps and irradiance caches all need to be reset. It is wasteful to invalidate the light maps across the whole scene just for the movement of one mesh. A task for future work would be to experiment with the possibility of selectively invalidating mesh light maps based on their relative distance and lighting intensity. Furthermore, only texels within an area around the considered mesh could be invalidated by setting their sample count to 0. This might be sufficient to improve the lighting artists' workflow in this case, without misleading them or presenting wrong information on screen.



(a) Granary by Night (rendered). **(b)** Zen Peak (light maps).

Figure 15: Test scenes from the selected viewpoint. (a) Granary by Night. (Courtesy of Evermotion.) (b) Zen Peak is a level from *Plants vs. Zombies Garden Warfare 2*. (Courtesy of Popcap Games, © 2018 Electronic Arts Inc.)

5 Performance and Hardware

5.1 Method

In recent ray tracing–related work, the performance metrics used are often technically oriented, such as giga-rays per second or a simple frames per second. In this project, the focus instead has been on the lighting artist's experience, and therefore our methods to assess performance reflect this. Performance manifests itself to the artist as refresh rate and convergence rate, i.e., how often and to what quality lighting conditions are presented to the user.

The quality of the GI solver's progressive output is assessed using a perceptual image difference measure, and the error is tracked over time. The L1 metric was simple and sufficient for our use case [13]. The L1 metric is applied to the results of the GI solver, i.e., light maps or irradiance volumes. Only texels in view that affect the viewed result are part of the calculation. A texel is considered significant if it is scheduled in at least 1% of all the update iterations. To limit the number of independent variables, the refresh rate has been fixed to 33 milliseconds in these tests.

To be able to calculate a perceptual image difference, an image representing ground truth is needed. It is computed using the base version of our GI solver pipeline, without optimizations or simplifications and without using time restrictions. Convergence is computed continuously. When each texel has reached a certain threshold of convergence (for details see Section 2), the resulting light maps and irradiance volumes are stored as a reference.

A specific acceleration technique can be assessed by adding it to the GI solver. The test starts by invalidating the state, depending on what user operation we want to simulate (as described in Section 4). As the process continues, after each iteration the results are compared with the reference and the metric gets applied.

The metric results in an error, or distance, from the ground truth. The error is logged and used to graph how it changes over time, and it is compared to a GI solver that does not include the acceleration technique.

Scene	Light Map texels		Mesh Triangles	Lights	
	Scene	View		Point	Area
Granary by Night	510k	25k	278k	89	18
Zen Peak	600k	25k	950k	297	0

Table 2: Test scene complexity. Local point lights include point, spot, and frustum lights.

	Granary by Night	Zen Peak
Irradiance Cache Building		
View Prioritization	0.2 ms	0.2 ms
Irradiance Cache	25.7 ms	3.4 ms
Trace	5.0 ms	27.3 ms
Denoise	0.7 ms	0.7 ms
Irradiance Cache Converged		
View Prioritization	0.2 ms	0.2 ms
Trace	32.7 ms	32.6 ms
Denoise	0.7 ms	0.7 ms

Table 3: Average time spent each iteration. As explained in Section 3.3.2, the irradiance cache cost will completely disappear once the cache has converged, i.e., after all the required samples have been resolved.

5.2 Results

Two different scenes with one viewpoint for each of them are used to measure performance. They are meant to represent typical use cases for our lighting artists. The first is an indoor scene mainly lit by local lights (Granary by Night) and the second is a large-scale outdoor scene from a production game (Zen Peak level from *Plants vs. Zombies Garden Warfare 2*). To give a sense of the complexity of each scene's view, both are presented visually in Figure 15 and their statistics are presented in Table 2. Apart from what is shown in the table, a relevant difference is the amount of overlapping local light. Zen Peak has lights distributed over the entire scene, while Granary by Night has clusters of 10 or more lights affecting the same area. The latter makes the light evaluation cost high, as shown by the time spent on the irradiance cache in Table 3.

All the graphs in Figures 16–18 show the normalized error over time, with a logarithmic y-axis scale. Notice that the outdoor scene converges faster, requiring a different interval on the same axis.

Intuitively, prioritizing texels that are visible in the scene is a substantial acceleration technique. As can be seen in Table 2, the different test scenes have a similar number of texels in view. The potential speedup should be relative to the fraction of texels in view. The results using view prioritization (described in Section 3.1) are shown in Figure 16. They are based on a full reset, and the error

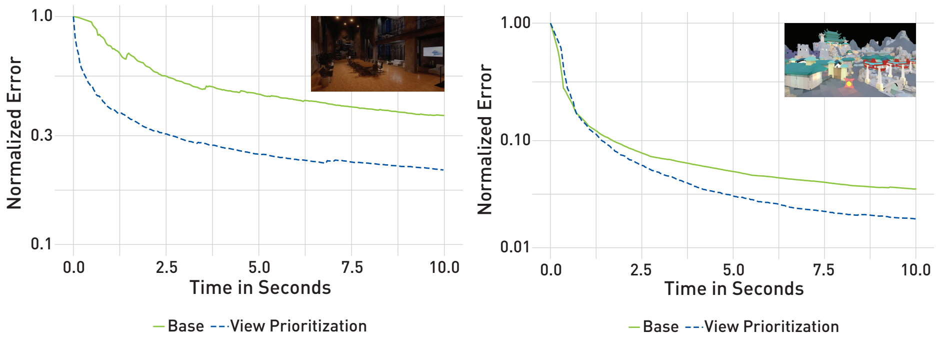


Figure 16: Convergence plots for demonstrating performance gain by using view prioritization on two scenes: Granary by Night (left) and Zen Peak (right). These plots show the error (L1) compared to a reference light map. View prioritization (dashed curve) allows for faster convergence, especially during the first few seconds, compared to scheduling all light map texels at once (red curve).

is plotted for the GI solver with and without view prioritization enabled. As seen in these graphs, the first test scene exhibits the significant speedup expected, but the second test case suffers from our measurement method, where every texel in the light map is equally important. The view prioritization algorithm will not hit tiny texels that cover less than a pixel in that view, thus the improved convergence rate is lower than expected.

The irradiance cache is the second major acceleration technique described in this chapter. Initially, it will use computing power to fill the cache with data, which is not directly used. It performs work on all texels in the scene, not only the ones hit by a ray during path traversal. This affects initial convergence, but, as shown in the first test case in Figure 17, after only a second the irradiance cache outperforms the base version of the GI solver. While the irradiance cache also converges quickly in the second test, it introduces an error bias of about 4.5%. This error is caused by a lack of resolution in areas where there are many tiny details or high frequencies in the lighting. See Section 3.3 for issues related to direct irradiance caching. Note that the irradiance cache is valuable long after being populated and needs to be reset only after certain user operations; see Section 4.

Denoising is primarily used to make the result look more pleasant to the user. The performance tests in Figure 18 show that it also improves the convergence rate. This is important, as denoising does not contribute to the total convergence. Instead, it only temporary affects what the user sees. Figure 19 summarizes visually the impact of all the described techniques.

5.3 Hardware Setup

When a single GPU is available, the Frostbite editor together with the GI solver will be scheduled concurrently on the same GPU. In this case, the operating system will try to schedule the workload evenly. However, by default, this setup will fail to provide a smooth experience, since there is no way for it to divide the work uniformly to target an even frame rate. Either a large ray tracing task is run

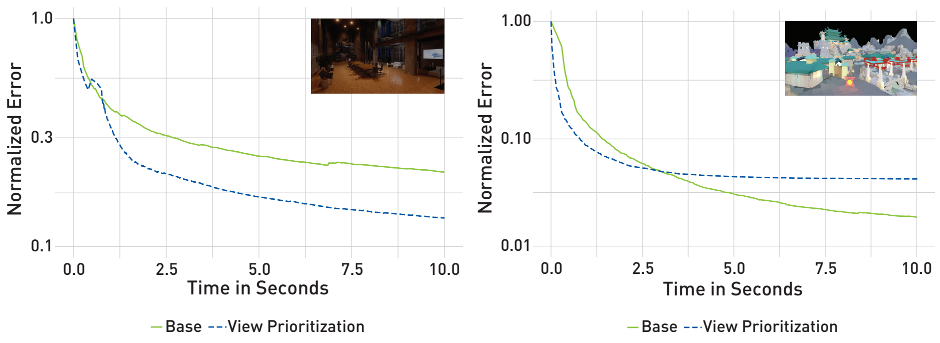


Figure 17: Convergence plots for demonstrating performance gain by using irradiance cache on two scenes: Granary by Night (left) and Zen Peak (right). These plots show the relative error (L1) compared to a reference light map. Using an irradiance cache (dashed curve) achieves a faster convergence compared to next event estimation at every path's vertex (red curve). This difference is especially visible on the Granary scene, which contains many local lights and so requires additional occlusion rays for estimating their visibility.

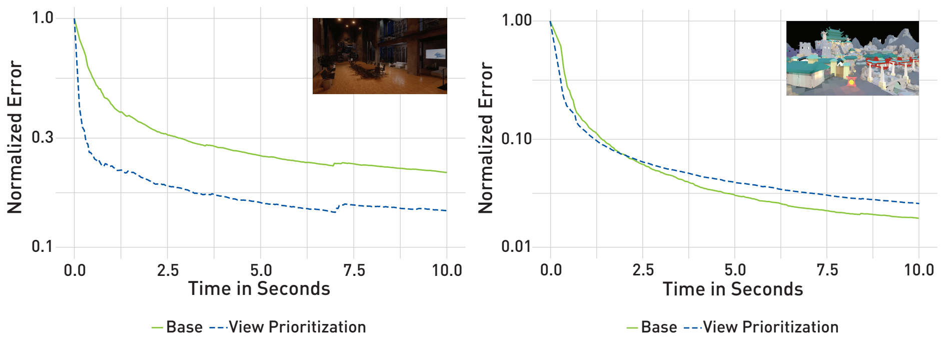


Figure 18: Plots illustrating the converge gain obtained by denoising the light map output on two scenes: Granary by Night (left) and Zen Peak (right). These plots show the relative error (L1) compared to a reference light map. This denoising step removes most of the high- and medium- frequency noise and allows us to quickly present a result similar to the converged output.

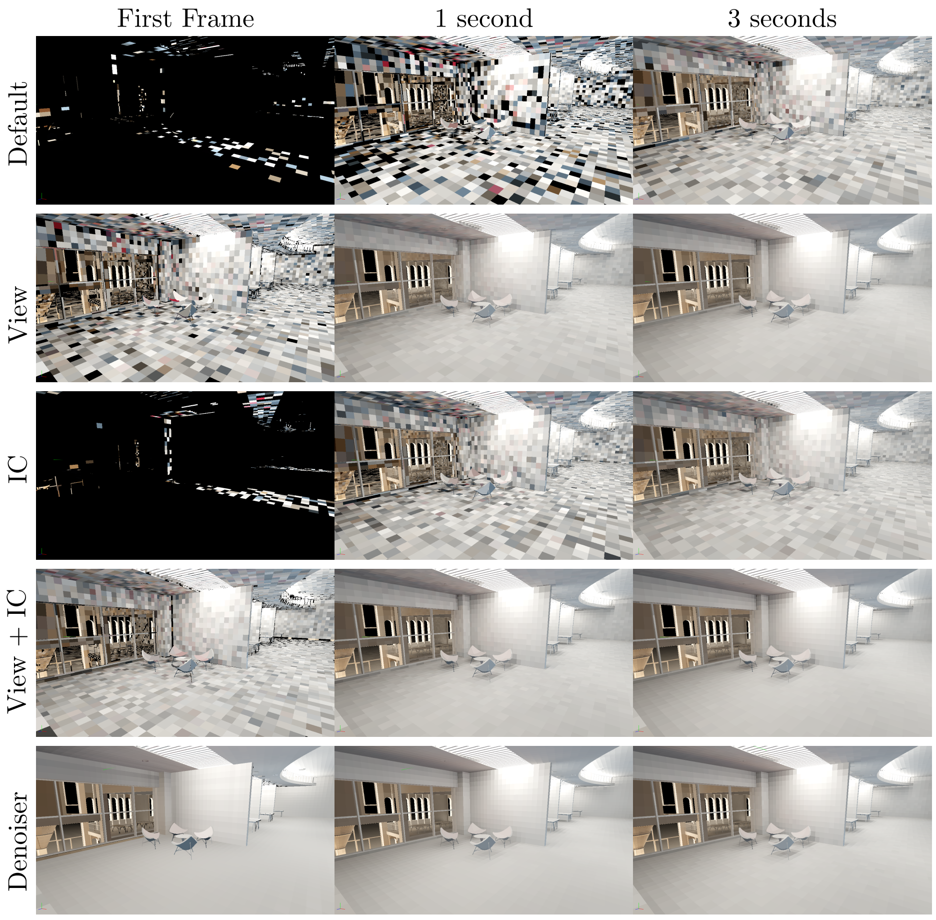


Figure 19: Visual comparison of convergence rate between different acceleration techniques. First row: all texels are scheduled equally, and next event estimation is done at every vertex of all paths. Second row: view prioritization (View) is used for scheduling only visible texels. Third row: irradiance cache (IC) is used for avoiding next event estimation. Fourth row: combination of view prioritization and irradiance caching. Fifth row: combination of view prioritization, irradiance caching, and denoiser.

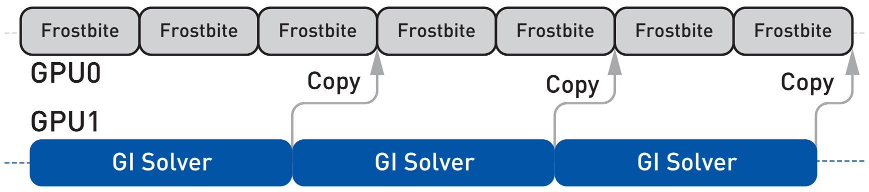


Figure 20: The Frostbite editor renders the game while the GI solver runs asynchronously on a side GPU. This enables a stutter-free experience, with both processes each leveraging the full power of a GPU.

and the Frostbite editor slows down considerably, or the ray tracing work is scheduled less often and the time to convergence is longer and updates are less frequent. A dual-GPU setup is recommended to avoid our GI path solver and Frostbite competing for the same GPU resources.

As shown in Figure 20, when two GPUs are used, the first can be used to render the editor and the game, while the second handles the GI solver doing ray tracing. Once an update cycle is done, the light map and light probe volumes are copied over to the GPU running the editor view for visualization. To this aim, the DirectX 12 multi-adapter mode is used, where each GPU is controlled explicitly and independently [14]. The most-capable GPU is selected to run the path tracing work, while the Frostbite editor requires at least a DirectX 11 compatible GPU.

In the future, the GI solver could be extended to handle $n + 1$ GPUs, n doing path tracing and one presenting the game editor to artists. The current dual-GPU approach is already a good fit for artists, providing a smooth, stutter-free experience for Frostbite games currently in production.

6 Conclusion

This chapter describes the real-time global illumination preview system used in production going forward by Electronic Arts titles running on Frostbite. It allows lighting artists to preview what the final global illumination baking process will produce while editing a level, in a matter of seconds rather than waiting minutes to hours for the final result. We strongly believe that it will make artists more efficient while allowing them to focus on what is important: art and their creative process. As a result they will have more time to iterate and polish each scene, and thus they will be more likely to produce higher-quality content.

The acceleration techniques put in place, such as dynamic light map texel scheduling and irradiance caching, enable reaching a higher convergence rate with minimal impact on quality. A light map denoising technique is also put in place to make sure the result is pleasing to the eye, even for a low sample count, for instance when the light map evaluation is restarted.

Going forward, more advanced caching representations could be investigated, such as path guiding for long paths and bidirectional path tracing [17]. For a smooth user experience, a dual-GPU local machine setup is recommended, allowing lockless asynchronous GI updates. Going further, a farm of DXR-enabled GPUs could be installed around the world, providing global illumination previewing and high-quality baking as a service for everyone in each of Electronic Arts' studios.

References

[1] Hao, C., and Xinguo, L. Lighting and Material of Halo 3. Game Developers Conference, 2008.

[2] Hillaire, S. Physically Based Sky, Atmosphere and Cloud Rendering in Frostbite. Physically Based Shading in Theory and Practice, SIGGRAPH Courses, 2016.

[3] Hillaire, S., de Rousiers, C., and Apers, D. Real-Time Raytracing for Interactive Global Illumination Workflows in Frostbite. Game Developers Conference, 2018.

[4] Iwanicki, M. Lighting Technology of 'The Last of Us'. In *ACM SIGGRAPH Talks* (2013), p. 20:1.

[5] Kajiya, J. T. The Rendering Equation. *Computer Graphics (SIGGRAPH)* (1986), 143–150.

[6] Křivánek, J., Gautron, P., Ward, G., Jensen, H. W., Christensen, P. H., and Tabellion, E. Practical Global Illumination with Irradiance Caching. In *ACM SIGGRAPH Courses* (2007), p. 1:7.

[7] Lagarde, S., and de Rousiers, C. Moving Frostbite to Physically Based Rendering. Advanced Real-Time Rendering in 3D Graphics and Games, SIGGRAPH Courses, 2014.

[8] Mitchell, J., McTaggart, G., and Green, C. Shading in Valve's Source Engine. Advanced Real-Time Rendering in 3D Graphics and Games, SIGGRAPH Courses, 2006.

[9] O'Donnell, Y. Precomputed Global Illumination in Frostbite. Game Developers Conference, 2018.

[10] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*, third ed. Morgan Kaufmann, 2016.

[11] Sandy, M., Andersson, J., and Barré-Brisebois, C. DirectX: Evolving Microsoft's Graphics Platform. Game Developers Conference, 2018.

[12] Schied, C., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C. R. A., Burgess, J., Liu, S., Dachsbacher, C., Lefohn, A., and Salvi, M. Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination. In *Proceedings of High-Performance Graphics* (2017), pp. 1–12.

[13] Sinha, P., and Russell, R. A Perceptually Based Comparison of Image Similarity Metrics. *Perception 40*, 11 (January 2011), 1269–1281.

[14] Sjöholm, J. Explicit Multi-GPU with DirectX 12—Control, Freedom, New Possibilities. https://developer.nvidia.com/explicit-multi-gpu-programming-directx-12, February 2017.

[15] Subtil, N. Introduction to Real-Time Ray Tracing with Vulkan. NVIDIA Developer Blog, https://devblogs.nvidia.com/vulkan-raytracing/, Oct. 2018.

[16] Teschner, M., Heidelberger, B., Müller, M., Pomeranets, D., and Markus, G. Optimized Spatial Hashing for Collision Detection of Deformable Objects. In *Proceedings of Vision, Modeling, Visualization Conference* (2003), pp. 47–54.

[17] Vorba, J., Karlík, O., Šik, M., Ritschel, T., and Křivánek, J. On-line Learning of Parametric Mixture Models for Light Transport Simulation. *ACM Transactions on Graphics 33*, 4 (July 2014), 1–11.

[18] Walker, A.J. New Fast Method for Generating Discrete Random Numbers with Arbitrary Frequency Distributions. *Electronics Letters 10*, 8 (February 1974), 127–128.

[19] Wikipedia. Online Variance Calculation Algorithm (Knuth/Welford). Accessed 2018-12-10.

[20] Wikipedia. Standard Error. Accessed 2018-12-10.