GI-1.0: A Fast Scalable Two-Level Radiance Caching Scheme for Real-Time Global Illumination

Guillaume Boissé
Advanced Micro Devices, Inc.
France

Sylvain Meunier
Advanced Micro Devices, Inc.
France

Heloise de Dinechin
Advanced Micro Devices, Inc.
France

Matthew Oliver
Advanced Micro Devices, Inc.
Australia

Pieterjan Bartels
Advanced Micro Devices, Inc.
Belgium

Alexander Veselov
Advanced Micro Devices, Inc.
Germany

Kenta Eto
Advanced Micro Devices, Inc.
Japan

Takahiro Harada
Advanced Micro Devices, Inc.
USA

Figure 1: Kitchen and Sponza scenes rendered with direct and indirect lighting calculated using our GI-1.0 pipeline in 3.5ms and 4.2ms respectively at 1080p on Radeon™ RX 6900 XT.

ABSTRACT

Real-time global illumination is key to enabling more dynamic and physically realistic worlds in performance-critical applications such as games or any other applications with real-time constraints. Hardware-accelerated ray tracing in modern GPUs allows arbitrary intersection queries against the geometry, making it possible to evaluate indirect lighting entirely at runtime. However, only a small number of rays can be traced at each pixel to maintain high framerates at ever-increasing image resolutions.

Existing solutions, such as probe-based techniques, approximate the irradiance signal at the cost of a few rays per frame but suffer from a lack of details and slow response times to changes in lighting. On the other hand, reservoir-based resampling techniques capture much more details but typically suffer from poorer performance and increased amounts of noise, making them impractical for the current generation of hardware and gaming consoles.

To find a balance that achieves high lighting fidelity while maintaining a low runtime cost, we propose a solution that dynamically estimates global illumination without needing any content preprocessing, thus enabling easy integration into existing real-time rendering pipelines.

KEYWORDS

global illumination, ray tracing, radiance caching, spatial hashing

email: { Guillaume.Boisse, Sylvain.Meunier, Heloise.Dupontdedinechin, Matthew.Oliver, Pieterjan.Bartels, Kenta.Eto, Takahiro.Harada }@amd.com
Advanced Micro Devices, Inc. Technical Report No. 22-10-9831, October 18, 2022.

1 INTRODUCTION

Probe-based techniques are often used in applications where a high framerate is required [Greger et al. 1998]. Light probes were precomputed in the past, but real-time hardware ray tracing makes it possible to compute them dynamically at runtime. Majercik *et al.* proposed Dynamic Diffuse Global Illumination to cache the irradiance field into a set of dynamically ray-traced probes organized in world-space grids [Majercik et al. 2019]. The per-pixel irradiance value can then be estimated by interpolating from the eight neighboring probes, taking the visibility information into account in the form of a Chebychev inequality test [Donnelly and Lauritzen 2006]. While this reduces the light leaking issue that plagued previous probe systems, the visuals often end up looking flat as the irradiance near occluders is typically of higher frequency than what the spatial resolution of the probe field can capture.

Techniques using resampled importance sampling have been actively explored recently [Talbot et al. 2005; Tokuyoshi and Harada 2016]. Reservoir-based Spatiotemporal Importance Resampling (ReSTIR) heavily relies on reservoir resampling to get high-quality samples, which results in a higher quality sampling for direct illumination [Bitterli et al. 2020]. It was further extended to indirect illumination [Boissé 2021]. ReSTIR Global Illumination (ReSTIR GI) [Ouyang et al. 2021] and, more recently, ReSTIR Path Tracing (ReSTIR PT) [Lin et al. 2022] propose to trace rays per pixel and rely on reservoir-based resampling to efficiently share path sampling information across neighboring pixels and frames. Such approaches yield promising visual results but are too expensive for performance-critical applications such as games or any other

applications with real-time constraints. Furthermore, tracing per pixel as opposed to interpolating between probes introduces significant amounts of noise in the image that must be filtered [Schied et al. 2017]. Aggressive filtering of noisy signals can lead to a loss in quality and increased difficulty in keeping up with lighting changes. Calculating a temporal gradient from the radiance signal [Schied et al. 2018] can help inform the denoising in such cases, but it is not trivial to compute, nor does it help when the input signal is overly noisy in the first place.

Screen Space Radiance Caching (SSRC) [Wright 2021] introduces a novel method that caches the radiance in probes spawned directly onto primary visible surfaces. The approach has advantages similar to the probe system and shows little noise after interpolating the lighting for every pixel. However, unlike world probes, screen probes do not suffer from light and occlusion leaks as they are always placed precisely on the geometry. Furthermore, they offer a significantly higher density radiance representation, which leads to higher-fidelity visuals.

In this paper, we build on the SSRC approach and introduce several key contributions to improve performance and visual quality. Most importantly, we introduce a novel caching algorithm to achieve temporally stable lighting without needing an additional world-space structure as required in [Wright 2021]. We will show how we connect the screen probe rays to a secondary level of radiance caching based on spatial hashing [Binder et al. 2019] to achieve fast, high-fidelity, and leak-free dynamic global illumination as shown in Figure 1.

2 GI-1.0

For real-time purposes, only a few samples can be used for each pixel, even with today's high-end GPUs, to remain practical. It is, therefore, essential to try and ensure that most, if not all, samples contribute to the lighting estimate in a meaningful way to keep the variance to a minimum. Indeed, any ray or path not hitting a light source can potentially severely impact the quality of the rendered animation by introducing noise that would require excessive amounts of filtering.

We design our global illumination pipeline to make the most of every sample by reusing the lighting information across space and time for both sampling and filtering. We do so by persisting the scene illumination inside two distinct levels of radiance caching, as illustrated in Figure 2:

- The screen cache caches the incoming radiance for primary path vertices inside probes placed directly onto primary visible surfaces, and offers a detailed lighting representation thanks to a large number of probes.
- The world cache caches the outgoing radiance for secondary path vertices and, despite being less detailed than the screen cache, has the advantage of being stable and persistent.

We will demonstrate how this setup allows to compute high-fidelity and temporally responsive direct and indirect lighting using sampling rates as low as ¼ sample per pixel.

2.1 Screen Cache

This section describes how screen probes are spawned sparsely directly onto pixels, how we manage the integrity of the caching data

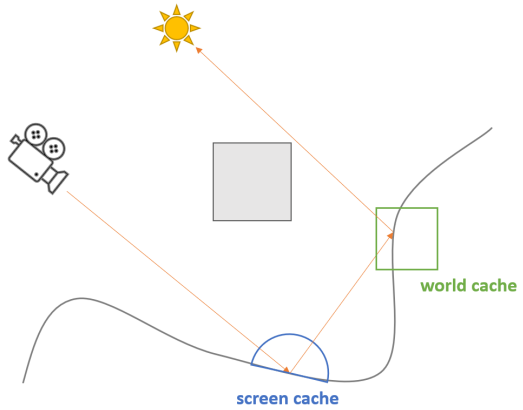


Figure 2: GI-1.0 two-level radiance caching scheme.

structure across frames, and how we adapt the filtering heuristics based on depth to ensure temporally stable lighting at any distance.

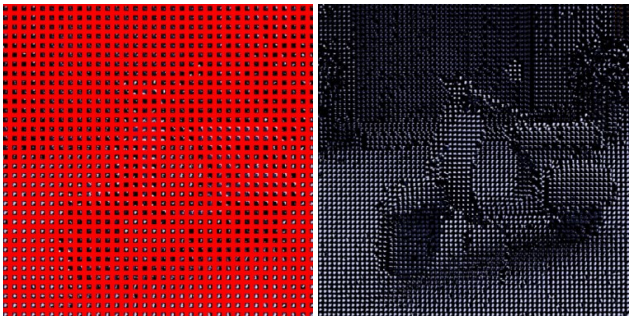
Similarly to [Wright 2021], we encode the incoming radiance across the oriented hemisphere into an 8x8 atlas using an octahedral projection mapping [Cigolle et al. 2014].

2.1.1 Temporal Upscale. In our method, the probe grid is upscaled to full resolution over multiple frames. The amount of upscaling is directly related to the overall per-pixel sample count and can be tweaked as a trade-off between performance and quality. In our implementation, we store up to one 8x8 screen probe, encoding the hemispherical radiance of a random pixel inside each 8x8 tile on the screen. A consequence of this setup is that the resulting probe grid can be stored inside a 2D texture of the size of a render target, with dimensions aligned to the next multiple of 8. This choice is arbitrary, and other configurations could be explored.

In this context, we define the spawn tile as a 2D region with dimensions of $8 \cdot (upscale\_x, upscale\_y)$, where $upscale\_x$ and $upscale\_y$ represent the amount of temporal upscaling performed along the X and Y axis respectively. We then generate a 2D jitter value inside the spawn tile using Halton's low-discrepancy sequence [Halton 1964] and use this value to select the pixel on which to place our new probes for every spawn tile on the screen. This leads to sparsely populated probe grids on first frames, which get resolved into fully populated grids after multiple frames. Figure 3 illustrates this process: on the very first frame, ¼ of all the probes are filled. The red regions in the figure represent the tiles that do not have probe data. We fill another ¼ in the next frame resulting in all probes being filled after four frames.

For each probe, we reconstruct the world-space position from the depth buffer and retrieve the surface orientation by decoding the normal vector from the rasterized depth and normal textures, usually called G-buffers [Saito and Takahashi 1990]. Additionally, our technique requires a set of motion vectors to be calculated for every pixel to reproject the probe grid from the previous frame into the current frame.

Probe reprojection is an essential step of the pipeline, as temporally reused probes are likely to persist over multiple frames, making accurate placement of previous probes onto the new frame's



(a) Sparse probe grid on 1st frame (b) Resolved grid after 4 frames

Figure 3: Temporal upscaling in probe space. (a) One out of four probe grids are filled. Red pixels show the empty probes. In each frame, we fill one out of four. After four frames, all the probe grids are filled (b).

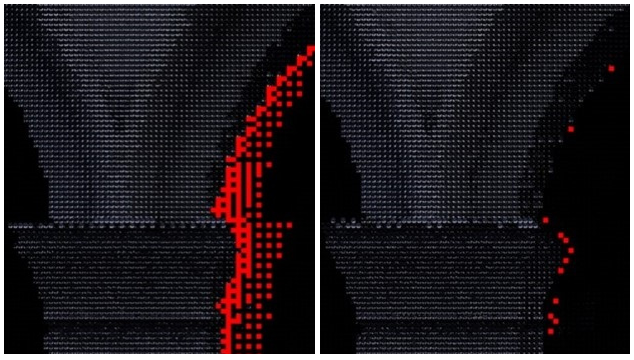
pixels a requirement to prevent degrading the overall image quality. Algorithm 1 outlines how every lane in an 8x8 dispatch group collaborates to find the best pixel for probe reuse. An interesting input to the algorithm is the *cell_size* heuristic, which controls how much spatial error is allowed when reusing the probes information temporally. As we will see in the later sections, this same heuristic is also used to guide the sampling and filtering of the screen probes, making it a key factor to get right.

```
__kernel reproject_screen_probes(pixel p)
    __local uint reprojection_score ←
      (pack_half(65504.0) << 16)|0xFFFFu
    barrier() // sync the threads
    if p isn't a sky pixel
        q ← p in previous frame
        if probe_q is valid
            plane_dist ←
              abs(dot(world_probe − world_p, normal_p))
            normal_check ← dot(normal_probe, normal_p)
            if plane_dist < cell_size and
            normal_check > 0.95
                dist ← distance(world_probe, world_p)
                uint probe_score ←
                  (pack_half(dist) << 16)|local_lane
                atom_min(reprojection_score, probe_score)
    barrier() // sync the threads
    // decode and use local_lane as destination pixel
```

Algorithm 1: Screen probes reprojection.

2.1.2 Adaptive Sampling. As a result of the sparse spawning of the screen probes, moving animations can exhibit grid regions where information is missing, as illustrated in Figure 4. These regions, usually called disocclusions or holes, are detrimental to the image quality as they introduce additional noise in newly appearing parts of the scene.

One possible solution to this problem is to allocate a fixed number of additional rays to the empty tiles to "fill the holes" [Swoboda 2012]. However, such an approach requires increasing the per-pixel ray budget, which we aim to keep low and constant throughout an animated sequence. Instead, we propose de-allocating rays randomly away from screen tiles that have succeeded the temporal reprojection and assigning them to the frame's empty tiles.



(a) Fixed probe spawning pattern (b) Stochastic ray re-balancing

Figure 4: Hole filling under motion with adaptive sampling.

For this purpose, we generate two separate queues:

- The *empty_tiles* buffer stores the list of tiles that have failed reprojection and are not filled with any newly spawned probe.
- The *override_tiles* buffer stores the list of newly spawned tiles that have succeeded the temporal reprojection.

We can patch our list of spawned tiles, referred to as *spawn_tiles* in Algorithm 2, by iterating over the empty tiles, and picking a random override tile to fill an empty tile with. This approach enables significant improvements in the visual quality of disoccluded regions, as shown in Figure 4, while maintaining a constant ray count over time. In effect, we simply redistribute a fixed ray budget to help fill our temporal holes.

```
__kernel patch_screen_probes(uint global_id)
    tile ← empty_tiles[global_id]
    index ← random(0, override_tile_count − 1)
    atom_xchg(spawn_tiles[override_tiles[index]], tile)
```

Algorithm 2: Temporal adaptive sampling.

2.1.3 Ray Guiding. At this point in the pipeline, we know what probes we will be calculating. We will be using raytracing for this but have yet to decide how to distribute our rays against the octahedral cells of each probe. Multiple options are possible here, such as assigning one ray for every cell and randomly jittering within the cell, otherwise known as uniform sampling.

However, we can do better than this by leveraging the information of our reprojected probe grid, when available, and guiding the sampling of the new rays or importance sampling [Wright 2021]. We implement this efficiently by writing the luminance of the reprojected radiance to local memory, or Local Data Share (LDS); the values are then scanned in parallel [Harris et al. 2007] and normalized into a Cumulative Distribution Function (CDF). We use the CDF to pick a random cell proportional to its estimated intensity

(a) Uniform sampling (b) Temporal ray guiding

Figure 5: Using reconstructed hemisphere for sampling.

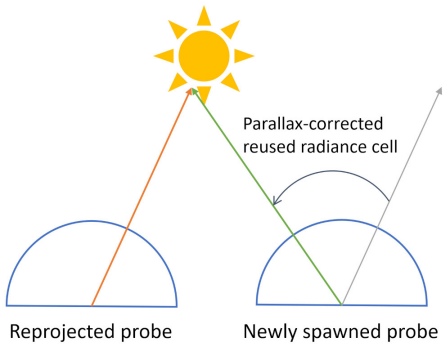


Figure 6: Parallax-corrected radiance reuse.

and recover the ray direction by generating a random 2D sample on the selected cell. We obtain a cleaner and more temporally stable image without increasing the ray count as illustrated in Figure 5.

To make the guiding precise, we want the hemisphere reconstruction to be as faithful as possible. Reconstruction here is iterating over the reprojected probes in a 3x3 tile neighborhood and accumulating the radiance values into the best corresponding cells of the newly spawned yet uncalculated probes. Reusing the cell indexing across probes leads to issues with nearby light sources having a different parallax in relation to the shading site, degrading the quality of the sampling. Therefore, we perform a parallax correction before scattering the reprojected radiance estimate into the storage of the new probe as shown in Figure 6. This is made possible by storing the traveled ray distance in the alpha channel of the probe grid texture, allowing recovery of the hit point position while iterating the reprojected cells. The scattering of radiance values is implemented efficiently by allocating the 8x8 octahedral map in LDS to perform the reconstruction and sampling. Finally, we use the same *cell_size* heuristic mentioned in section 2.1.1 for rejecting far away probes.

Now that all the sampled directions are generated, rays can be intersected against the scene using a closest hit query. If a ray misses all geometry, we consider it has reached the sky and add the corresponding incoming environment contribution to the ray payload. If the ray hits, however, we must calculate the lighting at the hit point. This is the role of our hash cells data structure, and we, therefore, defer the details of the implementation to section 2.2. For now, we assume that, similarly to the environment lighting, we somehow get a radiance contribution back that we add to the ray payload.

2.1.4 Radiance Blending. Now that we have a radiance estimate for each of our rays, we can resolve the payload into the new probes. As multiple rays can be assigned to the same cell, we need a way to accumulate into the destination storage efficiently. We again leverage the LDS and allocate the 8x8 probe in local memory for accumulating the contributions prior to normalizing.

In this pass, we blend the newly calculated radiance with the estimate reconstructed in 2.1.3. We found that using a regular exponential moving average [Karis 2014] led to a significant loss in visual fidelity. Indeed, the low-resolution nature of the probes leads to cells covering a relatively large area of the oriented hemisphere, making averaging uniformly across the whole region undesirable. Instead, inspired by [Lottes 2015], we adapt the temporal blending amount as a factor of the normalized difference between the newly estimated radiance and the reconstructed one. We propose a biased temporal hysteresis detailed in Algorithm 3, which we design to better preserve occlusion and shadows at the expense of some image darkening.

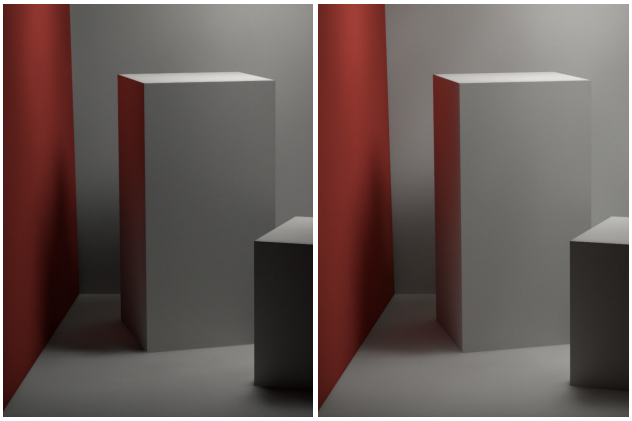
Further to preserving shadow details, our biased temporal hysteresis also acts as a firefly removal technique by effectively filtering out bright signals that are significantly smaller than the cone described by the cell's solid angle, making the responsible surface unlikely to be hit often. We found this property particularly important for ensuring the temporal stability of scenes featuring many emissive surfaces or mesh lights.

function temporal_blend($curr\_radiance, prev\_radiance$)
  $l1 \leftarrow dot(curr\_radiance, \frac{1.0}{3.0})$
  $l2 \leftarrow dot(prev\_radiance, \frac{1.0}{3.0})$
  $\alpha \leftarrow max(l1 - l2 - min(l1, l2), 0.0) / max(max(l1, l2), 1e-4)$
  $\alpha \leftarrow clamp(alpha, 0.0, 0.95)^2$ `// clamp and remap`
  return $lerp(curr\_radiance, prev\_radiance, \alpha)$

Algorithm 3: Biased shadow-preserving temporal hysteresis.

Due to our ray guiding strategy, we can end up with cells for which no rays were traced. Re-using the results of the temporal reconstruction aggressively in such cases leads to strong visual artifacts, while leaving the cell's content as black or empty leads to undesirable over-darkening caused by the missing energy. Instead, we average the radiance from the populated cells and distribute the result uniformly across the untraced cells. This allows recovering some of the energy loss through approximating the missing samples as illustrated in Figure 7. In practice, the approximation is only used for low-probability cells, and we have not found this to cause objectionable visual artifacts on any of the tested content.

2.1.5 Probe Masking. As probes can be placed on any random pixel within each of the 8x8 screen tiles, a mechanism is needed to retrieve that sub-tile position. We therefore encode the precise pixel coordinates of each probe inside a 32-bit integer that we store in a 2D texture, referred to as *probe_mask*. Empty or invalid tiles,



(a) Darkening due to empty cells (b) Fixed using radiance backup

Figure 7: Approximating empty cells with radiance average.

Figure 8: Skipping reprojection holes w/ mip-based masking.

that is, tiles containing no probe, are flagged using a sentinel value, shown as red in Figure 8.

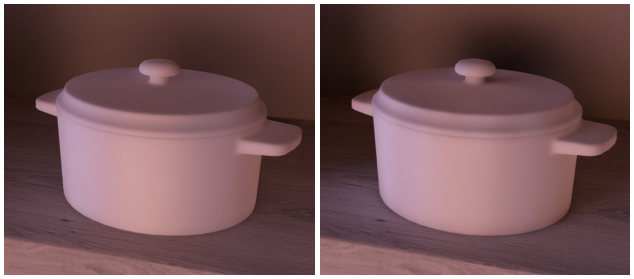
As we can see in this figure, areas for which probe information is missing, can be relatively large under motion. This is an issue as we often need to find the closest neighbor probe to a given pixel in a specific direction. Our solution is to generate a MIP chain of the probe mask, keeping the first valid probe found within the 2x2 upper values for each successive level. Leveraging this structure, we can efficiently perform large searches in screen space to retrieve neighbor probes in any direction, as shown in Algorithm 4.

```
function find_closest_probe(int2 pixel, int2 offset)
    pixel /= 8 // transform to probe space
    foreach mip ∈ 0..mip_count − 1 do
        int2 pos ← pixel + offset
        if pos is out of bounds
            break
        uint probe ← probe_mask.Load(pos, mip)
        if probe isn't sentinel
            return probe // found a probe :)
        pixel /= 2
    return sentinel // couldn't find any probe :(
```

Algorithm 4: Sparse directional search in probe space.

2.1.6 Probe Filtering. As we jitter the rays inside the cells every frame, the resulting probes can be fairly noisy. Furthermore, the radiance returned by our hash cells may also be noisy, especially when discovering a new scene area.

We leverage our large-scale probe search function to implement an efficient separable 7x7 sparse blur of the probes' radiance, as



(a) Naive probe-space filtering (b) With angle-based rejection

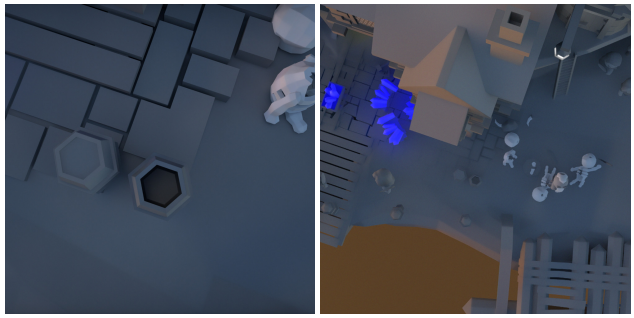
Figure 9: 7x7 separable sparse probe-space filtering.

shown in Algorithm 5. The *cell_size* heuristic is again used to avoid filtering across far away probes and prevent light leaking. Finally, we use a similar angle error detection to [Wright 2021] for preserving small-scale occlusion details as illustrated in Figure 9.

```
__kernel filter_screen_probes(global_id, local_id, group_id)
    p ← decode_probe_mask(spawn_tiles[group_id])
    dir ← calculate_cell_direction(local_id, normal_p)
    radiance ← probe_buffer[global_id]
    hit_dist ← radiance.a
    weight ← 1.0
    foreach i ∈ 0..5 do
        step ← (((i & 1) << 1) − 1) · ((i >> 1) + 1)
        probe ← find_closest_probe(p, step · blur_direction)
        if probe is sentinel
            continue
        q ← decode_probe_mask(probe)
        plane_dist ← abs(dot(world_q − world_p, normal_p))
        normal_check ← dot(direction, normal_q)
        if plane_dist > cell_size or normal_check < 0.0
            continue
        hit_dist_clamped ← min(hit_dist_q, hit_dist)
        hit_point ← world_q + dir · hit_dist_clamped
        angle_error ←
          dot(dir, normalize(hit_point − world_p))
        if angle_error < cos(π / 50.0)
            continue
        depth_weight ←
          calculate_depth_weight(depth_p, depth_q)
        radiance += 
          depth_weight · float4(radiance_q, hit_dist_clamped)
        weight += depth_weight
        hit_dist ← radiance.a / weight
    // store radiance / weight
```

Algorithm 5: Radiance filtering in probe space.

2.1.7 Adaptive Cell Size. Throughout the previous sections, we have used the same heuristic referred to as *cell_size* for guiding the reprojection, sampling, and filtering of our screen probes. This quantity directly relates to how permissive the radiance reuse is between neighboring probes; a low value leads to better detail preservation at the cost of degraded temporal stability, while a



(a) Details are captured up-close (b) Gracefully degraded far out

Figure 10: Relaxing radiance reuse heuristics at a distance.

high value helps maintain temporal stability but at the cost of decreased lighting quality. Therefore, we choose to use an adaptive value relaxed for objects located further away from the camera, as illustrated in Figure 10.

Algorithm 6 describes the calculations for the adaptive cell size, where fov_y corresponds to the vertical field of view in radians, and $proj\_size$ is the targeted cell size in pixels after projection. We choose $proj\_size$ to be set to a value of 8.0, which is roughly the amount of pixel spacing between neighbor probes. Note that $distance\_scale$ may be pre-calculated on the CPU as it is constant across the screen.

function calculate_cell_size($distance\_to\_camera$)
  $distance\_scale \leftarrow$
    $tan(fov\_y \cdot proj\_size \cdot max(\frac{1.0}{view\_height}, \frac{view\_height}{view\_width^2}))$
  return $distance\_scale \cdot distance\_to\_camera$

Algorithm 6: Calculating an adaptive cell size.

2.1.8 Persistent Least-Recently Used (LRU) Side Cache. Thanks to the temporal upscale, our screen cache is relatively dense across the screen over throughout multiple frames. However, despite many probes, we are still heavily undersampled with only one screen probe available for each 8x8 pixel tile. This is not an issue in itself as probes are meant to be interpolated from in a process where each pixel locates its four closest neighbors and performs some form of the average of the irradiance signal. We will cover the details of our probe interpolation and irradiance estimation in section 2.4.

This section focuses on on the temporal instability issues introduced by thin geometrical features. We have mentioned that the *cell_size* heuristic is used to choose whether to keep or reject temporal probe history and how we adapt this value at a distance to account for far away regions having a lower ratio of sample count to geometrical detail. However, there exist situations where the spatiotemporal radiance reuse between probes keeps failing, causing temporal wobbling in the illumination.

Figure 11 describes such a scenario:

- Probe 1 is initially spawned on the thin geometrical feature and calculated.
- Probe 2 is then spawned but fails to reuse the temporal information from 1 as the *cell_size* test fails. It is therefore re-calculated from scratch or, in other words, no ray guiding nor radiance blending can be performed. It is worth noting that, as mentioned in section 2.1.3, we perform the reconstruction against a 3x3 tile neighborhood in probe space rather than a single tile. We will ignore this property here however, for the purpose of simplifying the explanations.
- Finally, probe 3 is spawned but cannot reuse information from 2 as it is back on the geometrical feature.

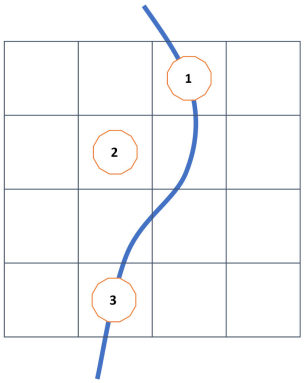


Figure 11: Temporal reuse failure caused by thin geometry.

The net result of this highlighted scenario is that our temporal reprojection, ray guiding and radiance blending keep failing each frame, leading to an unstable lighting response. What we can note however, is that we had valid temporal information for reconstructing probe 3 just two frames ago. We therefore propose to push the "evicted probes" onto a persistent queue, so such probes may be reused an arbitrary number of frames later.

In this context, evicted probes are detected when newly spawned probes fail to reuse data from the reprojected probe within the same tile during reconstruction. The queue itself is implemented as a simple array of probe indices while the radiance data is stored in another 2D texture with identical dimensions to our regular probe grid. This choice is arbitrary and other configurations could be explored.

In order to use what we will call the "cached probes" as part of the reconstruction, we need a mechanism for efficiently looking up the nearby probes for a given pixel. We implement a lookup structure similar to that used in tile-based deferred rendering pipelines [Andersson 2011] by iterating over the list of cached probes prior to the reconstruction, projecting them onto the screen, and scattering the probe indices into the dedicated list of the corresponding 8x8 tile. Since cached probes can persist over many frames, we cannot rely on the depth buffer to recover the world-space position, so we instead store the single-precision floating point position and packed world-space normal into a 128-bit integer for each cache entry.

We update our hemisphere reconstruction routine to search for cached probes in a 3x3 tile neighborhood after having completed the 3x3 tile search on the reprojected grid. All radiance contributions are still parallax-corrected and accumulated into LDS as before. Once the search is completed, we end up with three possible scenarios with regards to updating our side cache:

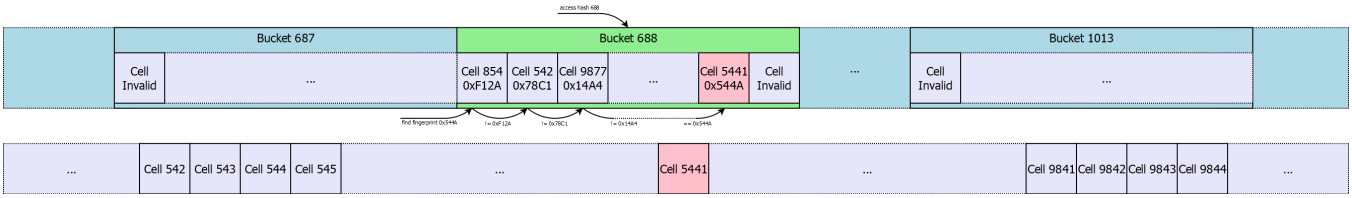


Figure 12: Hash map with open addressing and linear probing.

- The previous probe is evicted by the new one and no matching cache entry was found; we create a new cached probe and schedule the reprojected radiance to be written out to the cache during the blending pass.
- The previous probe is evicted by the new one but a matching cache entry could be found; we schedule that the reprojected radiance be written out to the cache during the blending pass and push the entry onto a most-recently used (MRU) queue.
- Regardless of whether the previous probe was evicted, we identify the best-matching cache entry that participated in the reconstruction, if any, and schedule that the new radiance estimate be written out to the cache during the blending pass. This step is important so the cached radiance does not lag behind under changing lighting conditions.

We mentioned an MRU queue in the case of finding a match for an evicted probe. We indeed perform a re-ordering pass of the indices of the cached probes every frame, keeping the MRU elements ahead while the LRU entries naturally fall behind. This setup allows to avoid evicting cache entries that would be actively participating in the illumination of the current viewpoint.

Finally, it is worth mentioning that we perform an atomic compare and swap operation prior to the scheduling of the cache updates mentioned in the above scenarios. This is to avoid a race condition where multiple work groups may try and update the same cached content during the radiance blending. Additionally, care must be taken as entries scheduled for update may have been overwritten by newly created cache entries. This can be verified trivially by checking the index of the entry inside the LRU queue against the allocation cursor for the newly created cache entries.

2.2 World Cache

We had previously skipped over the details of how to calculate and cache the outgoing radiance at each of the secondary path vertices. This is the role of our world cache, stored in hash cells, where we accumulate and filter the radiance being returned to the screen cache queries. We describe in this section how we build on the work of [Binder et al. 2019], using spatial hashing to create our data structure, which we further extend to allow fast filtering across neighbor cells using a novel tiling approach.

2.2.1 Caching Outgoing Radiance for Secondary Path Vertices. We aim at minimising the memory footprint while requiring little to no pre-processing of the scene geometry. The latter is indeed an important requirement for supporting dynamic worlds as well as facilitating the integration of our solution inside an existing real-time rendering pipeline. [Binder et al. 2019] proposes to cache the radiance into cells using spatial hashing. Their approach provides interesting properties that we want to build on:

- Their use of spatial hashing adapts to any kind of geometrical input as well as moving content.
- Filtering can be performed at no additional cost inside each cell thanks to the scattering logic used when writing out the radiance samples.
- The resulting grid is built entirely on the fly, as traversal occurs, therefore only requiring to allocate memory sparsely where information is needed.

Our world cache addresses the radiance cells by hashing a descriptor for each vertex. The descriptor, which we will describe next, is hashed using a first hash function in order to retrieve the index of a "bucket" as shown in Figure 12. Then, it is hashed a second time using another hash function in order to calculate a "fingerprint", which we use to perform linear probing inside the bucket and locate our cell in memory. Linear probing is an important step to solving the collisions that arise when two distinct descriptors wrongly resolve to the same bucket after only applying the initial hashing. Finally, we leverage the study by [Jarzynski and Olano 2020] and pick two fast hash functions that we found to produce little to no collision between one another.

As we want to cache the outgoing radiance inside the cells of our structure, we define our descriptors as the world space position of each vertex along with the direction of the ray or, in other words, the input parameters to the rendering equation [Kajiya 1986]. In practice, both attributes are quantized in order to enable reuse and filtering across neighboring vertices. Additionally, similar to the adaptive size described in section 2.1.7, we adapt the amount of quantization applied to the position attribute at a distance so as to ensure a roughly constant number of samples per cell. The descriptor is therefore updated to include the quantization level, effectively creating radiance level of details (LODs).

Each cell is associated a decay value, which is reset upon access, or left to decay towards zero otherwise. As a cell's decay reaches zero, we de-allocate the entry so its memory can be later reused for another region of world space. The main bottleneck of a global hash map, such as the one we are describing, is the extensive use of atomic operations for each insertion into the cache. While these operations remain relatively fast, we want to limit their use to a minimum for efficiency reasons. In practice, we only perform a single lookup for each vertex and cache the resulting index out to memory.

2.2.2 Eliminating Light Leaks. The quantization of the descriptor attributes mentioned in the previous section allows for efficient filtering of the radiance signal across neighbor vertices, while the

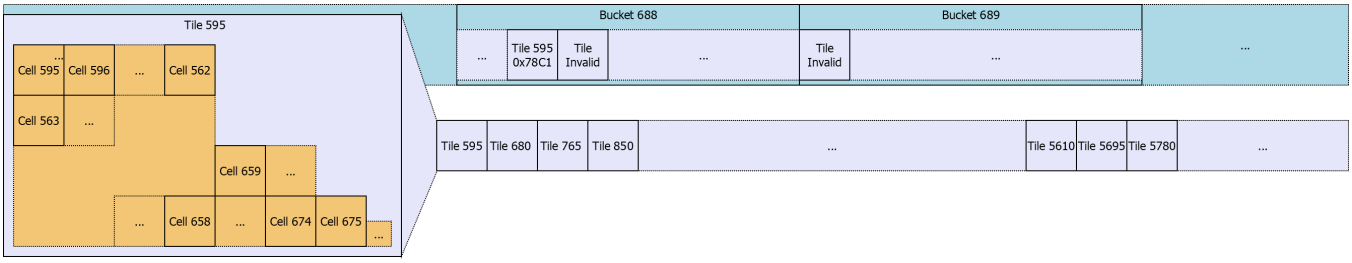


Figure 13: Two-level hash map. Each tile is a mipmapped 8x8 region of cells. Tiles and cells follow a linear layout in memory.

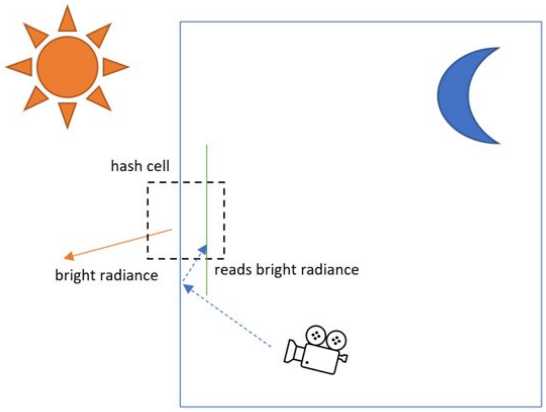


Figure 14: Illustrating a possible light leaking scenario.

(a) Without light leak fixup **(b) With light leak heuristic**

Figure 15: Fixing light leaks caused by thin occluders.

adaptivity of the quantization amount ensures a roughly constant number of samples per cell at any distance. However, this same quantization may be responsible for light leaking artefacts under certain conditions. Figure 14 illustrates such a scenario. Here, the inner "green wall" is reached by our secondary ray after bouncing off the outer wall. The hit point belongs to an enclosed dark region of the scene, and the estimated radiance should therefore be dark too. However, we also see a "bright radiance" event happening on the outside of the wall; here, the environment is bright and sunny. In this setup, both the bright and dark radiance events are close enough in world space that their positions are equal after quantization. We have seen however that our descriptors also include the ray direction, which can help break apart two nearby vertices into different cells. In this scenario though, both the secondary ray and bright radiance direction are parallel enough that their directions are also equal after quantization. Our secondary ray hit therefore shares the same cell after hashing as the bright radiance event, resulting in a light leak as illustrated in Figure 15.

We found that such light leaking situations mostly occur when the length of the secondary ray is less than the size of the cell being looked up, or, in other words, the ray hasn't travelled for more than a cell. We therefore enhance our descriptor by additionally hashing the boolean result of the $(ray\_length < cell\_size)$ inequality test. This helps breaking the two events mentioned above into two distinct cells fixing our leaking issue as seen in Figure 15.

2.2.3 Prefiltering Radiance. Filtering across cells requires to quickly access neighbors; performing additional lookups in even just a 3x3x3 hash-space neighborhood ends up being prohibitively expensive, so we instead propose to implement a two-level data structure. In this context, cells are not directly indexed by the buckets, but rather grouped into tiles, as illustrated in Figure 13. Indexing a cell requires its tile index and relative position inside the tile. Each tile contains a fixed number of cells that we can store in memory using a linear layout. This layout also includes MIP levels, which we use for prefiltering the radiance, thus enabling fast spatial filtering across cells within a tile. Tiling doesn't help however for filtering the radiance between tiles, which remains expensive. Instead we rely on our screen cache to hide the resulting structured artefacts at no additional cost.

Each tile represents a fixed volume of the scene, however, a 3D encoding of the cells isn't necessary for caching and filtering the radiance. Indeed, surfaces tend to form a plane when seen at a small enough scale. Thus, a 2D tile of cells is sufficient for faithfully describing the radiance of a local region of space. We build our tiles by projecting along the main axis for indexing the cell within its tile. The largest component of the outgoing direction defines the main axis and the projection is done by discarding the corresponding coordinate when computing the cell offset within its parent tile. This 2D rather than 3D tiling approach helps maximising the tile

(a) Using only shadow rays (b) With radiance feedback

Figure 16: Approximate multi bounce w/ temporal feedback.

occupancy, or, in other words, the ratio of used versus unused cells. This ultimately helps with both reducing the overall memory budget and keeping the filtering fast.

Reducing the variance to an acceptable level typically requires several frames worth of contributions inside a cell. The outgoing radiance therefore needs to be filtered before returning to the screen cache queries, especially for cells that have been recently created. Contributions are temporally accumulated into the cells at the first MIP level using an exponential moving average [Karis 2014]. Other MIP levels are then iteratively generated using box filtering of radiance samples from the upper level in a single pass. In practice, we found that the best performance was achieved with a tile size of 8x8. Finally, we accumulate the radiance into the screen cache queries using the first finest level for which enough samples have been accumulated.

2.2.4 Evaluating Lighting at Secondary Path Vertices. We have described so far how we cached and filtered the direct lighting at secondary path vertices inside a two-level hash grid data structure. However, the direct lighting still needs to be evaluated. First, we reuse the illumination from the last frame using temporal reprojection. If the reprojection succeeds, we can reuse the radiance sample and forego any further, potentially expensive calculations. If the reprojection fails however, we need a reliable way of estimating the lighting at the vertex site. This is the role of our light sampling implementation, which we describe in section 2.3. It is worth noting that when reprojecting last frame's lighting, the reused radiance sample encodes the direct lighting as well as an indirect lighting estimate. This is an interesting property, which leads to what we refer as "temporal radiance feedback", and allows to approximate an infinite number of light bounces as seen in Figure 16. This approximative multi bounce illumination is estimated at no additional cost over multiple frames. Finally, it is worth noting that the result of the reprojection is filtered, therefore injecting a noise-free radiance sample into our world cache, leading to an interesting reduction in the variance of the resulting indirect illumination.

2.3 Light Sampling

We have seen that populating the radiance cache requires evaluating the direct lighting at each of our hit points. Our implementation supports traditional real-time visibility techniques such as shadow maps, however, such techniques typically do not scale to large numbers of lights. We therefore propose a light sampling strategy, using raytracing, to stochastically sample the lighting distribution and fill the radiance cache.

(a) Blotches from hash cells noise (b) Fixed w/ world-space ReSTIR

Figure 17: Using ReSTIR for efficient light sampling.

2.3.1 Reservoir-based Resampling. Sampling the direct lighting distribution can lead to large variance when a significant proportion of sampled shadow rays do not hit the light source therefore failing to contribute meaningfully to the estimate. This can have a strong impact on the world cache variance which flows through to the screen cache, and ultimately results in increased onscreen noise as illustrated in Figure 17. Efficient importance sampling of the lighting distribution is therefore key to keep the noise to a minimum while casting as few rays as possible.

To importance sample the light candidates, we use a form of ReSTIR but implement world-space reservoir reuse described by Boissé instead of screen space [Boissé 2021; Ouyang et al. 2021]. Indeed, screen-space reuse is not amenable to sampling the lighting at secondary path vertices, as neighbor vertices in world space are not necessarily close, or even available, in screen space. We start out by generating a reservoir for each of the hit positions using Resampled Importance Sampling (RIS) [Talbot et al. 2005] and Weighted Reservoir Sampling (WRS) [Chao 1982]. We will describe in the next section how we try and select the most important lights for a given point in space in order to better initialize the reservoirs.

The generated reservoirs are then stored inside a hash grid, using a very similar approach to the spatial hashing technique used for the world cache. In this case however, we only supply the adaptively quantized position of the shaded vertex as input to our hash functions. Still, we do store the surface normal into an additional data stream, which allows for bilateral thresholding during the reuse, helping to reduce the darkening bias introduced by ReSTIR [Boissé 2021].

To improve performance for real-time applications, our approach deviates from a typical ReSTIR implementation and aims at reducing the number of shadow rays generated per reservoir as well as the number of resampling passes. Our approach skips the visibility ray altogether and only uses a single shadow ray after the resampling has completed. In effect, this reduces the required number of shadow rays down to one per reservoir, but it increases the bias in the form of darkening, as highlighted in Figure 17. We also forego the spatial resampling passes and instead rely on a single temporal reuse pass. This approach reduces the noise within a fixed raytracing budget, which allows for the world cache to be less aggressive with how the radiance is retained temporally. This is indeed an important requirement for achieving low latency response to lighting changes.

2.3.2 Light Grid Lookup Structure. Reservoir resampling helps in improving the quality of the light sampling by effectively sharing the sampling effort across neighbor vertices. However, resampling can only be as good as the initial state it is being fed. We therefore look to improve this initial state by generating a global light grid structure storing the most important lights for each spatial cell prior to the reservoir generation. This helps producing better candidates for the reservoir generation phase, therefore greatly improving on the resampled output produced by our reservoir pipeline.

We generate the light grid by first determining its bounds from the positions of every hit point. From there, we can create an axis-aligned bounding box (AABB) around the scene extents containing all these intersection positions. Within these bounds, we then create a uniform voxel grid. Each grid axis dimension is calculated to ensure grid cells remain square and no single axis has more than a user configurable maximum cell count. Each grid cell finally holds a fixed-size list of the most important lights for that region of the scene. The size of this list is determined to be the minimum between a user configurable value and the number of lights in the scene. To weight each lights importance to the grid cell, we calculate the relative luminance of the total deposited light from each light source over the total volume of the cell. The total deposited light is calculated from the radiance L at each point p within the cells volume V using:

$$\iiint_V L(p)\, dx\, dy\, dz \tag{1}$$

Equation (1) must be solved for all supported light types which are, point, spot, directional, area, and environment in our implementation. However, it is not possible to integrate this quantity analytically. One numerical solution is to generate multiple samples within the volume and combine them in order to approximate the integral [Pharr and Humphreys 2010]. This has the downside of requiring many samples to properly converge to the correct value and ensure all lighting is correctly detected when lights only partially cover the cell. For instance, a spotlight whose cone only passes through a small region of the volume may be entirely skipped if none of the generated samples fall within the light cone. Instead, we use a fast, cheap approximation to Equation (1) by sampling the light radiance at each corner of the grid cell and tri-linearly interpolating across the entire volume. During this step we ignore the cone angle of spotlights and instead treat them as point lights. We then prepend an additional step to cull lights quickly by clamping the area of effect of each light. We then perform an intersection test between the lights area of effect and the grid cell and ignore lights that do not intersect. For point and spotlights this area of effect is based on the light's user supplied maximum distance attribute. Area lights however have an infinite area of effect, so we calculate a maximum distance value by determining the range in which the lights contribution falls below a user supplied threshold value. This threshold value can thus be used to trade performance for light sampling bias. For point lights, the area of effect becomes a sphere, for spotlights, it is their outer cone, and for area lights, it is the hemisphere above the surface. We further optimise this test by treating each cell's AABB as a bounding sphere as this significantly simplifies the calculations while only introducing minimal error.

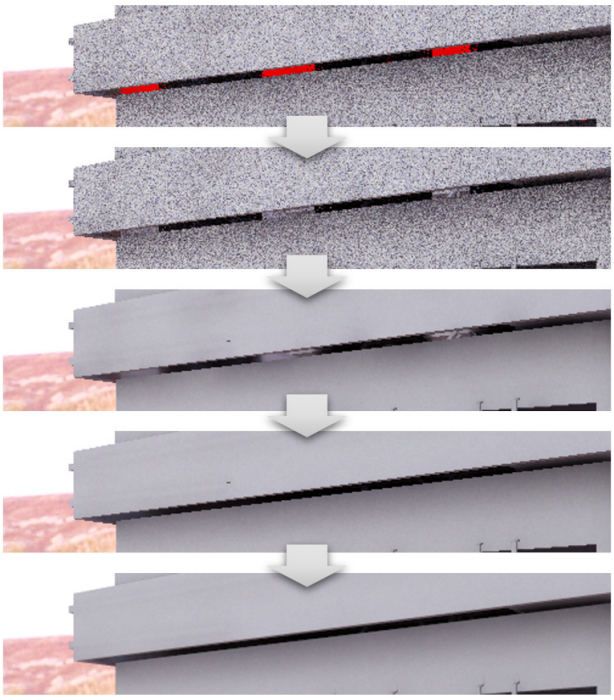


Figure 18: Fixing interpolation failures using denoiser hint.

The above algorithm trades accuracy for performance in multiple locations but these inaccuracies are hidden by the subsequent reservoir resampling passes.

The light grid structure is finally used to feed samples as batches into each of the generated reservoirs. Samples are selected by generating a random index into the light list of the grid cell overlapping the shaded vertex, which are then resampled into a reservoir using the procedure described in section 2.3.1.

2.4 Irradiance Estimation

We have seen that our screen cache encodes the incoming radiance in every direction over the primary visible surfaces. However, we are still left to estimate the irradiance. In this section, we see how to leverage our probe grid and evaluate the irradiance for every pixel on the screen.

2.4.1 Per-Pixel Interpolation. As probes are placed sparsely across the screen, we compute a weighted average of the radiance from the neighbor probes to reconstruct the lighting signal. This process, usually referred to as interpolation, is performed for every pixel at target resolution. If we reflect on the rest of the pipeline presented so far, this is the only pass running at full rate, with regards to target resolution, along with the probe reprojection pass described in 2.1.1.

We start out by finding the neighboring probes in all 4 directions using the $find\_closest\_probe()$ function defined in 2.1.5. The contribution for each probe is weighted using an edge-aware function based on the surface depth and normal. In this process, we also ensure that nearby probes, lying further away than $cell\_size$ units from the pixel's plane, are discarded by assigning a weight of 0. If

(a) Disocclusion mask (b) Dilated blur mask (c) Filtered irradiance

Figure 19: Spatial filtering guided by dilated blur mask.

(a) GI-1.0 irradiance estimation (b) Combined w/ screen-space GI

Figure 20: Using short screen traces to recover small details.

all probes get assigned a null weight then interpolation fails, which can lead to light leaking artefacts as highlighted in Figure 18. In such cases, we fall back to setting equal weights for all neighbor probes, which we refer to as "relaxed interpolation". We flag pixels calculated using relaxed interpolation in the alpha channel of the resolved texture; this information will be used later as a hint for the denoiser to try and discard the evaluated sample.

Finally, we jitter the pixel's position prior to performing the search for neighbor probes in order to break up the structured artefacts resulting from all neighboring pixels interpolating from the exact same set of probes. It is worth noting however that we cancel the jitter if the resulting position lands outside the original pixel's plane [Wright 2021]. This cancellation is important to prevent increasing the number of pixels having to rely on relaxed interpolation.

2.4.2 Spherical Harmonics. To accurately evaluate the irradiance encoded by a given probe, we need to fetch nearly all the stored radiance samples, that is, all the mapped directions across the oriented hemisphere. We instead project the screen probes to spherical harmonics (SH) prior to the interpolation pass.

Spherical harmonics indeed provide several key benefits to estimating the irradiance:

- High frequency noise can be filtered at no additional cost by limiting the projection to the first three bands.
- Irradiance can be estimated faithfully while reducing the requirements on the amount of memory to be fetched.
- Finally, we update our reprojection pass to also reproject the SH representation of each probe. We therefore only need to perform the projection for the newly spawned probes, which drastically reduces the cost of the operation.

The irradiance is then computed using the dot product of the projected cosine lobe with the projected radiance. The projected radiance is fetched from the cache while the cosine lobe is projected analytically [Ramamoorthi and Hanrahan 2001].

2.4.3 Denoising. The interpolated irradiance typically exhibits low-frequency noise from the probes, as well as high-frequency noise caused by the pixel jittering mentioned in 2.4.1. The low-frequency noise manifests itself as boiling in areas where the incoming radiance is most noisy, while the high-frequency noise is similar to that of a dithering pattern. These issues can be solved using a simple denoiser based on temporal accumulation and an adaptive spatial filter, where we compute the spatial filter radius depending on the number of samples accumulated in the history. As not all pixels have history, such as disoccluded pixels, we adapt the filtering radius to the number of accumulated samples as illustrated in Figure 19. The spatial filter uses an edge-aware weight based on depth and normal. When interpolation fails and there is no history available, we use the relaxed irradiance interpolation instead of outputting no irradiance.

3 SCREEN-SPACE GLOBAL ILLUMINATION

The techniques described so far can typically lack details as small-scale occlusion and color bleeding, smaller than the space between screen probes, cannot be captured by the grid. We propose a hybrid approach to mitigate this issue and implement short-range screen-space ambient occlusion and global illumination to fill in the missing details. In this section, we explain how we combine screen-space global illumination with the pipeline we have described so far.

3.1 Combining Near and Far Fields

As the screen cache captures the low-frequency global illumination well, we only add global illumination from screen-space techniques where the high-frequency information is missing. Following [Jimenez et al. 2016] and [Mayaux 2018], we create a mask using a bent cone, which is a cone centered on the bent normal with an aperture angle derived from the ambient occlusion. Instead of integrating the clamped cosine lobe with the incoming radiance, we multiply the bent cone and the clamped cosine lobe, then integrate the product with the incoming radiance. Ringing artefacts can occur in darker areas. This issue can be mitigated by using windowing with a small factor as done in [Sloan 2008].

3.2 Horizon-Based Occlusion

While most screen-space methods could be used or extended, we focus on horizon-based methods [Jimenez et al. 2016] [Mayaux 2018] which offer a good trade-off between performance and quality for near-field global illumination (i.e., using a short radius). We use temporal accumulation for removing interpolation artefacts after the irradiance integration, so we only compute one slice per frame using a few steps. The bent normal and ambient occlusion estimation leads to a noisy mask approximation, which we denoise using the spatiotemporal loop described in 2.4.3.

Table 1: GI-1.0 performance results (in ms)

	Raytracing (DXR-1.1)		Caching & sampling		Interpolate & denoise		**Total time**	
	AMD Radeon™ RX 6900 XT	NVIDIA GeForce RTX™ 3080	AMD Radeon™ RX 6900 XT	NVIDIA GeForce RTX™ 3080	AMD Radeon™ RX 6900 XT	NVIDIA GeForce RTX™ 3080	AMD Radeon™ RX 6900 XT	NVIDIA GeForce RTX™ 3080
Gas station (99k triangles) 	0.235	0.132	1.032	1.220	0.665	0.554	**1.932**	**1.906**
Flying world (267k triangles) 	0.532	0.292	1.226	1.460	0.705	0.574	**2.463**	**2.326**
Kitchen #1 (1.4M triangles) 	0.336	0.198	1.633	2.122	0.765	0.656	**2.734**	**2.976**
Kitchen #2 (9.0M triangles) 	0.676	0.343	1.693	2.327	0.755	0.756	**3.124**	**3.426**

4 RAY TRAVERSAL

The method described in this paper requires ray casting operations which we implement using hardware accelerated ray tracing using a traditional Bounding Volume Hierarchy (BVH) to accelerate intersection calculations. However, the global illumination algorithm can use other methods for ray casting as long as the method returns the hit point of a ray. We also implemented ray casting using a hybrid of hardware accelerated BVH ray tracing and distance field tracing [Bartels and Harada 2022]. We improved the ray tracing quality by using BVH ray tracing at the place where distance field tracing only produces large error. Although it was implemented, the speed up we observed was not big enough to employ it as the default option as our technique is already optimized to trace a reduced number of rays. The method would pay off if the algorithm requires more ray casting.

5 IMPLEMENTATION AND RESULTS

We implemented our GI-1.0 pipeline inside our Direct3D12 research framework using DXR-1.1 for accelerating the ray intersection queries. We prepared some test scenes shown in Figure 22 and used them to evaluate the runtime performance of our technique estimating both the direct and indirect lighting at ¼ samples per pixel at 1080p.

We have broken down the timings into 3 categories:

- Raytracing represents approximately the time spent in ray traversal. We mention approximately here as other bits of logic are run in both the closest-hit kernel (i.e., screen cache rays) and any-hit kernel (i.e., world cache rays), although the cost is vastly dominated by ray traversal.
- Caching & sampling represents the time spent maintaining the screen cache and world cache representations. This includes all guiding, sampling, reconstruction, filtering, prefiltering, reprojection, etc.
- Interpolate & denoise represents the time spent in the per-pixel interpolation pass followed by the spatiotemporal denoiser. This includes projecting the screen probes to spherical harmonics as well as estimating the disocclusion mask, further dilated into a blur mask for filtering.

Table 1 shows the breakdown and the total time spent executing our GI-1.0 pipeline [1]. We can see that the total time ranges from 1.932ms to 3.124ms at most on an AMD Radeon™ RX 6900 XT GPU. We achieve this performance with small amounts of noise and at a low raytracing cost thanks to our caching scheme. A recorded video of our technique is also available at GPUOpen.com.

We also implemented our global illumination pipeline as a plugin to Unreal Engine 5 (UE5). This allowed us to validate our technique and choices across a wider variety of environments and lighting scenarios. It further enabled the ability to switch between our solution and UE5's Lumen renderer for performance and quality comparisons. We present such a comparison, considering only the indirect lighting signal, using the sample UE5 Archviz scene in Figure 21. These images are screen captures of the UE5 engine running Lumen and GI-1.0.

It is worth noting however that our UE5 integration remains incomplete resulting in materials being approximated in our plugin. This is not a limitation of our GI-1.0 pipeline but rather a consequence of the important engineering effort required to accurately evaluate UE5's material system.

6 LIMITATIONS AND FUTURE WORK

We have presented a complete raytraced global illumination pipeline aimed primarily at estimating the indirect diffuse lighting of a scene dynamically at runtime. Direct lighting from environment maps and emissive surfaces is also supported, at no additional cost, thanks to the robustness of our ray guiding implementation.

6.1 Limitations

We have mentioned in section 2.2.4 that we could estimate an infinite number of light bounces over multiple frames thanks to our radiance feedback mechanism. While this technique provides some interesting performance advantage, on top of the visual improvements, it only really works when the contributing reflector is visible inside the previous frame. This can be detrimental to the visual fidelity of interior scenes in particular, where bounced lighting tends to dominate. In some future work, we want to look at efficient ways of approximating path continuation directly in hash space in order to achieve fast and reliable multi-bounced lighting everywhere in world space.

6.2 Glossy Reflections

Glossy reflections can be rendered with stochastic methods similar to [Stachowiak 2015]. Although our implementation of glossy reflections is still in progress, we briefly describe here the algorithm we are implementing. We generate importance-sampled directions

[1] Based on AMD internal testing, September 2022, using a desktop system configured with a Radeon™ RX 6900 XT GPU, Ryzen™ 7 5800X CPU, 32GB RAM, and Windows 10 vs. a similarly configured system with an NVIDIA RTX™ 3080 GPU to measure the performance results (in ms) of both systems in DXR-1.1 raytracing, caching & sampling, and interpolation & denoising. Results will vary.

(a) Lumen's output using SW traversal (2.6ms)

(b) GI-1.0's output using HW traversal (2.4ms)

Figure 21: Running our GI-1.0 pipeline inside Unreal Engine 5.

(a) Gas Station

(b) Flying world

(c) Kitchen #1

(d) Kitchen #2

Figure 22: Test scenes.

from the GGX lobe [Heitz 2018] and evaluate the reflected radiance using our screen and world caches. As screen probes store low-frequency illumination, directly using this representation for low-roughness surfaces gives an overly blurry result. In such situations, we instead cast a ray in the sampled direction and evaluate the incoming radiance by accessing the environment map if the ray misses, or the world cache at the hit position. One advantage of using our caching hierarchy for reflections is that we can account for multi-bounce lighting solely by looking up our cached radiance in memory.

6.3 Future Work

The current light grid sampling approach provides a list of important lights for a given area of the scene; it is interesting to note that this also includes the visible region of the view frustum. As such it may be useful to leverage this structure for sampling the lighting at primary path vertices and enable general direct lighting support from many light sources. The metric used in calculating the light grid importance adds bias to the rendering as we clamp the light intensity. Adding a non-biased approach is future work [Tokuyoshi and Harada 2016].

ACKNOWLEDGMENTS

We would like to thank Bruno Stefanizzi and Prashanth Kannan for their support of the research, as well as Holger Gruen and Oleksandr Kupriyanchuk for their help in reviewing this paper. Gas station and Flying world were created by John Constantine and burunduk, respectively. AMD, AMD Radeon and the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

Johan Andersson. 2011. DirectX 11 Rendering in Battlefield 3. https://www.ea.com/frostbite/news/directx-11-rendering-in-battlefield-3

Pieterjan Bartels and Takahiro Harada. 2022. Combining GPU Tracing Methods within a Single Ray Query. In SIGGRAPH Asia 2022 Technical Communications, to appear (SA '22 Technical Communications). Association for Computing Machinery, New York, NY, USA.

Nikolaus Binder, Sascha Fricke, and Alexander Keller. 2019. Massively Parallel Path Space Filtering. https://doi.org/10.48550/ARXIV.1902.05942

Benedikt Bitterli, Chris Wyman, Matt Pharr, Peter Shirley, Aaron Lefohn, and Wojciech Jarosz. 2020. Spatiotemporal Reservoir Resampling for Real-Time Ray Tracing with Dynamic Direct Lighting. ACM Trans. Graph. 39, 4, Article 148 (jul 2020), 17 pages. https://doi.org/10.1145/3386569.3392481

Guillaume Boissé. 2021. WORLD-SPACE SPATIOTEMPORAL RESERVOIR REUSE FOR RAY-TRACED GLOBAL ILLUMINATION. In SIGGRAPH Asia 2021 Technical Communications (Tokyo, Japan) (SA '21 Technical Communications). Association for Computing Machinery, New York, NY, USA, Article 22, 4 pages. https://doi.org/10.1145/3478512.3488613

Min-Te Chao. 1982. A General Purpose Unequal Probability Sampling Plan. Biometrika 69, 3 (Dec 1982), 653–656.

Zina H. Cigolle, Sam Donow, Daniel Evangelakos, Michael Mara, Morgan McGuire, and Quirin Meyer. 2014. A Survey of Efficient Representations for Independent Unit Vectors. Journal of Computer Graphics Techniques (JCGT) 3, 2 (17 April 2014), 1–30. http://jcgt.org/published/0003/02/01/

William Donnelly and Andrew Lauritzen. 2006. Variance Shadow Maps. In Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games (Redwood City, California) (I3D '06). Association for Computing Machinery, New York, NY, USA, 161–165. https://doi.org/10.1145/1111411.1111440

Gene Greger, Peter Shirley, Philip M. Hubbard, and Donald P. Greenberg. 1998. The Irradiance Volume. IEEE Comput. Graph. Appl. 18, 2 (mar 1998), 32–43. https://doi.org/10.1109/38.656788

J. H. Halton. 1964. Algorithm 247: Radical-Inverse Quasi-Random Point Sequence. Commun. ACM 7, 12 (dec 1964), 701–702. https://doi.org/10.1145/355588.365104

Mark Harris, Shubhabrata Sengupta, and John D. Owens. 2007. Parallel Prefix Sum (Scan) with CUDA. In GPU Gems 3, Hubert Nguyen (Ed.). Addison Wesley, Chapter 39, 851–876.

Eric Heitz. 2018. Sampling the GGX Distribution of Visible Normals. Journal of Computer Graphics Techniques (JCGT) 7, 4 (30 November 2018), 1–13. http://jcgt.org/published/0007/04/01/

Mark Jarzynski and Marc Olano. 2020. Hash Functions for GPU Rendering. Journal of Computer Graphics Techniques (JCGT) 9, 3 (17 October 2020), 20–38. http://jcgt.org/published/0009/03/02/

Jorge Jimenez, Xian-Chun Wu, Angelo Pesce, and Adrián Jarabo. 2016. Practical Real-Time Strategies for Accurate Indirect Occlusion. Technical Report.

James T. Kajiya. 1986. The Rendering Equation. SIGGRAPH Comput. Graph. 20, 4 (aug 1986), 143–150. https://doi.org/10.1145/15886.15902

Brian Karis. 2014. HIGH-QUALITY TEMPORAL SUPERSAMPLING. http://advances.realtimerendering.com/s2014/#_HIGH-QUALITY_TEMPORAL_SUPERSAMPLING

Daqi Lin, Markus Kettunen, Benedikt Bitterli, Jacopo Pantaleoni, Cem Yuksel, and Chris Wyman. 2022. Generalized Resampled Importance Sampling: Foundations of ReSTIR. ACM Trans. Graph. 41, 4, Article 75 (jul 2022), 23 pages. https://doi.org/10.1145/3528223.3530158

Timothy Lottes. 2015. GPU Unchained. https://www.youtube.com/watch?v=WzpLWzGvFK4

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 (JCGT) 8, 2 (5 June 2019), 1–30. http://jcgt.org/published/0008/02/01/

Benoît Mayaux. 2018. Horizon Based Indirect Lighting (HBIL). Technical Report.

Yaobin Ouyang, Shiqiu Liu, Markus Kettunen, Matt Pharr, and Jacopo Pantaleoni. 2021. ReSTIR GI: Path Resampling for Real-Time Path Tracing. Computer Graphics Forum (2021).

Matt Pharr and Greg Humphreys. 2010. Physically Based Rendering, Second Edition: From Theory To Implementation (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Ravi Ramamoorthi and Pat Hanrahan. 2001. An Efficient Representation for Irradiance Environment Maps. In Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01). Association for Computing Machinery, New York, NY, USA, 497–500. https://doi.org/10.1145/383259.383317

Takafumi Saito and Tokiichiro Takahashi. 1990. Comprehensible Rendering of 3-D Shapes. In Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques (Dallas, TX, USA) (SIGGRAPH '90). Association for Computing Machinery, New York, NY, USA, 197–206. https://doi.org/10.1145/97879.97901

Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R. Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. 2017. Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination. In Proceedings of High Performance Graphics (Los Angeles, California) (HPG '17). Association for Computing Machinery, New York, NY, USA, Article 2, 12 pages. https://doi.org/10.1145/3105762.3105770

Christoph Schied, Christoph Peters, and Carsten Dachsbacher. 2018. Gradient Estimation for Real-Time Adaptive Temporal Filtering. Proc. ACM Comput. Graph. Interact. Tech. 1, 2, Article 24 (aug 2018), 16 pages. https://doi.org/10.1145/3233301

Peter-Pike Sloan. 2008. Stupid Spherical Harmonics (SH) Tricks. Game Developers Conference (01 2008).

Tomasz Stachowiak. 2015. Stochastic screen-space reflections. In ACM SIGGRAPH Courses 2015: Advances in Real-Time Rendering in Games.

Matt Swoboda. 2012. Advanced Procedural Rendering with DirectX 11. https://www.gdcvault.com/play/1015455/Advanced-Procedural-Rendering-with-DirectX

Justin Talbot, David Cline, and Parris Egbert. 2005. Importance Resampling for Global Illumination. In Eurographics Symposium on Rendering (2005), Kavita Bala and Philip Dutre (Eds.). The Eurographics Association. https://doi.org/10.2312/EGWR/EGSR05/139-146

Yusuke Tokuyoshi and Takahiro Harada. 2016. Stochastic Light Culling. Journal of Computer Graphics Techniques (JCGT) 5, 1 (2016), 35–60. https://jcgt.org/published/0005/01/02/

Daniel Wright. 2021. Radiance Caching for Real-Time Global Illumination. https://advances.realtimerendering.com/s2021/index.html#_mrnver3hf0ag