Diffuse Global Illumination

Darius Bouma

Figure 14.1. Example of diffuse global illumination.

14.1 Introduction

Global illumination is a key component in creating a realistic and immersive gaming experience. The techniques described in this article are directly used in our in-house rendering framework called Breda, which we use in our upcoming benchmark called *Evolve*, as well as the recent demo for Samsung on their latest XClipse 940 GPU, powering the S24. Trying to accurately capture all of the scene's bounce light from numerous light sources in real time is a difficult task. Typically the frame budget only allows a very limited amount of rays to be traced, which would require heavy denoising in order to somewhat reliably estimate indirect lighting in a scene. Conserving micro detail while keeping a stable and responsive image has proven to be particularly challenging. In order to address these challenges, ReSTIR [Bitterli et al. 20, Ouyang et al. 21, Wyman and Panteleev 22] is used for both the secondary rays and irradiance cache, amortizing the workload both spatially and temporally. Some trade-offs are made that sacrifice correctness, however in return we get highly responsive and detailed indirect diffuse lighting. To cache indirect lighting in world space, we chose to go for a ReSTIR-based clipmap irradiance cache [Stachowiak 22].

14.2 Clipmap Irradiance Cache

The main purpose of an irradiance cache is to answer queries that require indirect lighting information at a given intersection. These contributions are cached, saving a lot of computation time for any system that needs to sample indirect lighting. Clipmap cascades are used to keep the entry cell size roughly constant in screen space regardless of the distance to the camera. The cache is not temporally stable and lacks resolution for an accurate direct representation of irradiance in the scene, however it performs very well when sampled indirectly from paths at a relatively low cost. Entries are allocated on demand and trace four candidate paths along with four additional validation paths every frame. (See Table 14.1.) With an upper bound of 65,535 entries, this would net a total of 524,280 rays in the worst-case scenario. In reality, the cache barely reaches this upper bound, typically running around 20,000 active allocations.

	Timings	**Bounce Rays**	**Shadow Rays**
Update Lifetimes	14 µs		
Reproject Cascades	110 µs		
Trace Candidates	232 µs	4	4
Trace Validate	315 µs	4	4
Build SH	46 µs		
Total	717 µs	8 per entry	8 per entry

Table 14.1. Clipmap irradiance cache running in Breda on an RTX 3060.

14.2.1 Initialization

The clipmap consists of 12 cascades, each containing $32 \times 32 \times 32$ cells that represent an indirection to an irradiance cache allocation. Each cell utilizes a 16-bit index to the underlying irradiance cache entry along with an additional 2 bits for flagging the cell as occupied and available for sampling. The remaining 12 bits are unused, allowing for a potential increase in the number of irradiance cache allocations if so desired. The underlying irradiance cache can hold up to 65,535 entries, where each entry has a lifetime, trace origin, ReSTIR sample data, and L1 spherical harmonics (SH) to store the total irradiance contribution. (See Listing 14.1.)

- **Clipmap cells:** This is the only buffer that needs double buffering. The clipmap relies on history data that's potentially indexed with different positional data.
- **Entry pool:** Available allocation entries are stored in the form of a FIFO queue. The pool is initialized by setting the value to the index. An atomic counter is used to fetch and recycle allocation indices.

```
struct IrradianceCacheSample {
    Reservoir reservoir;
    float3 incomingRadiance;
    float hitT;
    float3 sampleDirection;
};

struct IrradianceCacheEntryData {
    // Samples for 4 x 4 octahedron mapping
    IrradianceCacheSample samples[16];
};
```

Listing 14.1. Irradiance cache data per entry.

- **Entry lifetimes:** Each entry has a lifetime that is used to recycle the entries back to the entry pool. A special value is assigned here to mark entries as deallocated, which is also used to initialize the entire buffer.
- **Entry origins:** Each irradiance cache entry traces rays from a probe origin; to avoid self-intersection, the origin is moved back along the ray direction from which it was allocated. The surface normal is used here in order to later scale down samples that would result self-lighting.
- **Irradiance samples:** Entries are represented in a 4×4 octahedral mapping, where all of the samples hold a ReSTIR reservoir, incoming radiance, hit distance, and sample direction.
- **Spherical Harmonics:** A single L1 SH is stored per entry.

To address the memory footprint, the data is compressed from 640 bytes to 256 bytes. This is done using octahedron encoding [Cigolle et al. 14] for the sample directions and *R9G9B9E5* is used for the incoming radiance. Finally the reservoir is compressed to 32 bits: 16 bits for the sample count (M) and 16 bits for the contribution weight (W). The lighting contribution and sample direction are already known, thus the full reservoir can be reconstructed using the reservoir M and W. (See Listing 14.2.)

```
Reservoir reconstructReservoir(float selectedWeight, uint
packedReservoir) {
    Reservoir result;
    result.M = f16tof32(packedReservoir & 0xffff);
    result.W = f16tof32(packedReservoir >> 16u);
    // W was initially constructed from
    // wSum / (M * selectedWeight), so the logic can be
    // reversed to reconstruct wSum. This assumes
    // `W` is > 0.
    result.wSum = selectedWeight * M / W;
    return result;
}
```

Listing 14.2. Reservoir reconstruction from 32 bits.

```
uint computeCascadeIndex(float3 cameraPositionWs, float baseCellSize
, uint cascadeResolution, float3 samplePositionWs) {
    float3 relativePosition = samplePositionWs - cameraPositionWs;
    float greatestDistance = max(max(abs(relativePosition.x),
          abs(relativePosition.y)), abs(relativePosition.z));

    float cascadeCenterToEdgeDistance = (baseCellSize * (float)(
cascadeResolution - 1)) * .5;
    uint cascadeIndex = 0;
    if (greatestDistance >= cascadeCenterToEdgeDistance) {
        cascadeIndex = (uint)log2(greatestDistance /
cascadeCenterToEdgeDistance) + 1;
    }

    return clamp(cascadeIndex, 0, kMaxNumberOfClipmapCascades - 1);
}
```

Listing 14.3. Cascade index.

14.2.2 Cache Lookup

When querying the clipmap, the associated cell needs to be loaded in order to determine if the indirection points to a valid allocation. This is done by converting the query position in world space to a local coordinate within the clipmap. First off, the relative distance to the clipmap cell is measured, which is used to calculate the cascade index (Listing 14.3). The resolution of the clipmap cascade is reduced by 1 during sampling for a very specific reason: the origins of the cascades are based on the cell size, but don't always align properly. By conservatively sampling from a higher cascade, sampling outside of the cascade is prevented.

Since all of the clipmap cells are located linearly in a buffer, a 3D index within the current cascade needs to be converted to a linear buffer index. We know that all of the cascades are centered around the camera, however to find the cell index within the cascade, the cascade origin needs to be known, as well as the cascade bounds (Listing 14.4).

```
float3 calculateCascadeOrigin(float3 cameraPositionWs, float
baseCellSize, uint cascadeIndex) {
    float cascadeCellSize = (float)(1U << cascadeIndex) *
baseCellSize;
    return floor(cameraPositionWs / cascadeCellSize) *
cascadeCellSize;
}

Aabb calculateCascadeBounds(float3 cameraPositionWs, float
baseCellSize, uint cascadeResolution, uint cascadeIndex) {
    float3 cascadeOrigin = calculateCascadeOrigin(cameraPositionWs,
baseCellSize, cascadeIndex);
    float cascadeSize = baseCellSize * (float)cascadeResolution * (
float)(1U << cascadeIndex);

    Aabb result;
    result.minBounds = -cascadeSize * 0.5 + cascadeOrigin;
    result.maxBounds = cascadeSize * 0.5 + cascadeOrigin;

    return result;
}
```

Listing 14.4. Cascade origin and bounds.

The sampling position can then be converted to a relative index within the cascade. After this, all that's left to do is to linearize the relative index and account for the size of all cascades that come before the current cascade, which gives the global index. (See Listing 14.5.)

```
uint computeBufferLocation(float3 cameraPositionWs, float
baseCellSize, uint cascadeResolution, float3 samplePositionWs) {
    uint cascadeIndex = computeCascadeIndex(cameraPositionWs,
baseCellSize, cascadeResolution, samplePositionWs);
    // Note that the bounds are in world space.
    Aabb bounds = calculateCascadeBounds(cameraPositionWs,
           baseCellSize,  cascadeResolution, cascadeIndex);

    if (any(samplePositionWs < bounds.minBounds) || any(
samplePositionWs > bounds.maxBounds)) {
        return ~0u; // Position is out of bounds.
    }

    float3 cascadeSize = bounds.maxBounds - bounds.minBounds;
    uint3 relativeIndex =
        uint3(((samplePositionWs - bounds.minBounds) / cascadeSize)
* cascadeResolution);

    // Linearize to global buffer index.
    uint location = cascadeIndex * cascadeResolution *
cascadeResolution * cascadeResolution;
    location += relativeIndex.x;
    location += relativeIndex.y * cascadeResolution;
    location += relativeIndex.z * cascadeResolution *
cascadeResolution;
    return location;
}
```

Listing 14.5. Calculating buffer location.

With the clipmap cell index known, the metadata is loaded to determine if the cell is free or occupied. If the cell is free, an `InterlockedOr` is performed to see if the query is the first one to touch the cell. This is extremely important as multiple sources might be trying to allocate an irradiance cache entry; not being careful here can result in oversubscribing the irradiance cache. If the result is not yet flagged as occupied, the query can safely proceed with allocating an entry. If the allocation has succeeded, a new probe origin and hit normal is stored. (See Listing 14.6.)

In a scenario where most of the queries are trying to sample/allocate the same clipmap cell, contention may become a very big problem. This is especially true when the irradiance cache has a low resolution, increasing the odds that queries are performed on the same cell. To mitigate this problem, the cell metadata is loaded up front. If the cell is marked as occupied, the atomic will be skipped, which will drastically reduce contention. This does however come at a slight cost when cells are unoccupied, as both a memory load and dependent atomic are performed.

When the cell both is occupied and contains valid sampling data, the entry SH is resolved based on the hit normal [Hazel et al. 15]. Additionally, a new probe origin is also stored for the cell, which serves as a way to prevent

```
uint clipmapCellLocation = computeBufferLocation(cameraPositionWs,
baseCellSize, cascadeResolution, samplePositionWs);
uint metadata = clipmapCellBuffer.Load<uint>(clipmapCellLocation *
sizeof(uint));

bool isOccupied = (metadata & kAllocatedBit) == kAllocatedBit;
// Only perform the InterlockedOr if the cell is free to reduce
// contention.
if(!isOccupied) {
    clipmapCellBuffer.InterlockedOr(clipmapCellLocation * sizeof(
uint), kAllocatedBit | kJustAllocatedBit, metadata);
    bool wasFirstToAllocate = (metadata & kAllocatedBit) !=
kAllocatedBit;
    if(wasFirstToAllocate) {
        const uint counterOffsetInBytes = 0u;
        const uint dataOffsetInBytes = sizeof(uint);
        // First request an available pool entry.
        uint freeIndex;
        entryPool.InterlockedAdd(counterOffsetInBytes, 1u, freeIndex
);
        // Retrieve the actual allocation index.
        uint allocationIndex = entryPool.Load<uint>(
dataOffsetInBytes + freeIndex * sizeof(uint));

        // Pack the new allocation index and update the cell
        // metadata.
        metadata |= allocationIndex << 16u;
        clipmapCellBuffer.Store<uint>(clipmapCellLocation * sizeof(
uint), metadata);
        .. // Store associated data using the allocationIndex.
    }
}

bool justAllocated = (metadata & kJustAllocatedBit) ==
kJustAllocatedBit;
// Early out if no sampling data is available.
if(justAllocated) {
    return 0;
}

uint allocationIndex = metadata >> 16u;
// Proceed to sample SH data at allocationIndex.
..
```

Listing 14.6. Cache lookup and allocation.

potentially bad probe origins from persisting throughout many frames. Note that atomically allocating probe origins using a "first come first served" principle is *not* deterministic. If determinism is desired, an alternative probe origin allocation scheme is needed.

Lifetimes are updated for allocation indices each frame, ensuring no stale allocations survive longer than necessary. During this process we proceed to deallocate any entries that no longer have a valid lifetime. All associated data related to the allocation is cleared upon deallocating an entry, which includes all ReSTIR reservoir data, tracing origins, and spherical harmonics. For simplicity we avoid directly updating the clipmap references here, as the reprojection pass will already take care of invalid references by validating the associated lifetimes.

14.2.3 Reprojecting Cascades

In order to use history lighting information, we need to reproject the clipmap cascades. Each frame we calculate how many cells each cascade has moved, which is then used to scroll each cascade individually. If a clipmap entry in the cascade is about to go out of the cascade bounds, the underlying allocation needs to be flagged for recycling, which is done by setting the lifetime to 0. It's important that this happens during the reprojection, as letting the cells decay on their own will cause massive spikes in allocation counts when moving the camera around. Additionally, if a reprojected cell originates from a different cascade, it's discarded as the tracing origin may not fall within the bounds of the new cascade.

14.2.4 Updating the Cache

Candidate Paths Due to the dynamic nature of the irradiance cache, the assumption can be made that these entries will never fully converge as they are frequently updated with a new tracing origin. For that reason it's very important to gather lighting contributions uniformly, especially when an irradiance cache entry was just allocated. To ensure a uniform distribution, each frame four candidate paths are traced from the entry's origin, which update the octahedron faces using checkerboard indexing. Doing so prevents only a single side of the octahedron receiving new samples in a single frame by chance. Additionally, an R2 sequence [Roberts 18] is used per octahedron face to further improve a uniform distribution.

Paths originating from an entry query the irradiance cache recursively, which over time results in "infinite" light bounces. This is also where the probe's associated normal is used to scale down contributions that point toward this specific normal. The resulting data is used to create a ReSTIR reservoir, which is then merged with the history reservoir.

Validation Paths In order to keep the cache responsive to lighting changes, four additional previous paths are validated each frame [Stachowiak 22]. A total of four frames (Figure 14.2) is needed to provide all the samples with data. The validation is performed on the last frame to receive new candidates, which would be two frames ahead of/behind the current frame. Using this validation scheme, we ensure that no set of samples will be stale for longer than one frame. In the scenario of no history data being present, canonical paths are traced instead.

The paths are replayed using the random number of that given frame, which is then validated based on the difference in hit distance or incoming radiance compared to the previously stored data. (See Listing 14.7.) If the difference is large enough, we will need to inform the reservoir of the new lighting conditions. The strategy here is to determine the maximum distance between the history contribution and the newly measured contribution, and use this delta to scale the reservoir's sample count down.

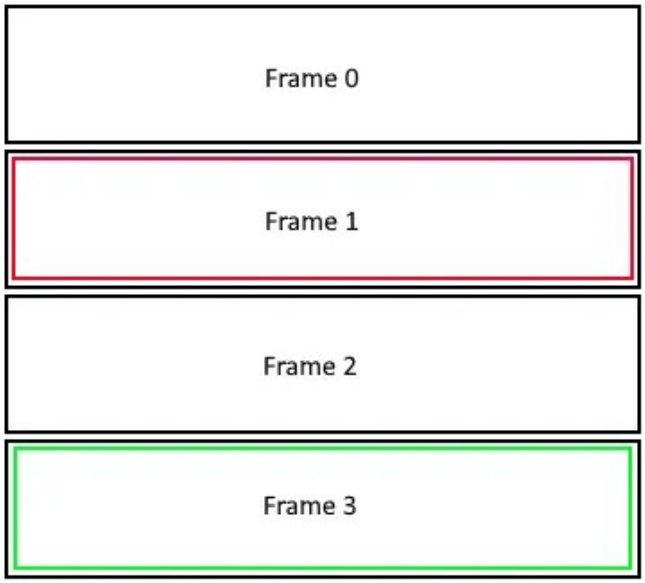


Figure 14.2. Validation is performed on samples that originate two frames ago, marked with red. Canonical paths are traced for the current frame, marked with green.

```
float dist = max3(abs(validationContribution - previousContribution)
 / (validationContribution + previousContribution));
const float minDist = 0.1;
const float maxDist = 0.6;
float validity = 1.0 - smoothstep(minDist, maxDist, dist);
```

Listing 14.7. Generating a validity factor.

The validity could then be used to linearly scale down the reservoir sample count (Listing 14.8); this improves responsiveness to changes to lighting conditions greatly, but does not respond quite well to smaller changes. Instead, a nonlinear curve can be used to respond well to both small and large changes, while still preserving temporal stability.

```
const float reservoirClamp = 20;
float suggestedSampleCount = exp2(log2(reservoirClamp) * invalidity)
;
```

Listing 14.8. Determining validation sample count.

Additionally, when validating history samples, the stored sample is directly updated with the re-evaluated lighting contribution. Ideally, the contribution weight is adjusted as well to reflect the change of the selected sample, however in practice this makes close to no difference as the reservoir quickly picks up newer samples during subsequent resampling stages.

14.2.5 Resolving Data

To reduce the memory footprint for cache entries, spherical harmonics are used, which nicely avoids having to resolve 16 path results each time the irradiance cache is queried. The SH coefficients can be compressed using FP16; optionally, this can be compressed even further using YCoCg spherical harmonics [Shyshkovtsov et al. 19], which can be compressed to 12 bytes at acceptable quality levels. Packing the coefficients using R11G11B10 or R9G9B9E5 turned out to differ too much visually from the expected result.

Only valid samples are considered when building the SH; if only one frame's worth of samples is traced for an entry, only those samples should be added to the SH. Furthermore, to increase temporal stability, the spherical harmonics coefficients are interpolated each frame.

14.2.6 Modifying Lookup

Clipmap cells are essentially axis-aligned bounding boxes (AABBs), as seen in Figure 14.3, which is problematic when axis-aligned geometry is located exactly on the boundary of two cells. Due to floating point imprecision, artifacts similar to Z-fighting start to appear. To mitigate the issue, we can modify the lookup

Figure 14.3. Clipmap visualized.

Figure 14.4. Applying periodic shift on clipmap lookup [Gautron 22].

function by applying a periodic shift [Gautron 22]; this increases the quantity of intersections with geometry but nearly always prevents full overlap with geometry. (See Figure 14.4.)

14.2.7 Ranking System

With a clipmap recursively allocating new entries, we run into potential query collisions when geometry is small enough to fit within the bounds of one of the axes of a clipmap cell. In such a scenario, we need a way to avoid light leaking through the geometry. (See Figure 14.5.)

A way to improve this is to use a ranking system [Stachowiak 22]. Rays that originate from the camera will have a rank of 1, entries that were allocated from this ray will trace their rays with their allocated rank + 1, and so on. A clipmap tracing the origin will only be considered if the query rank is less than or equal to the rank it currently has. (See Figure 14.6.)

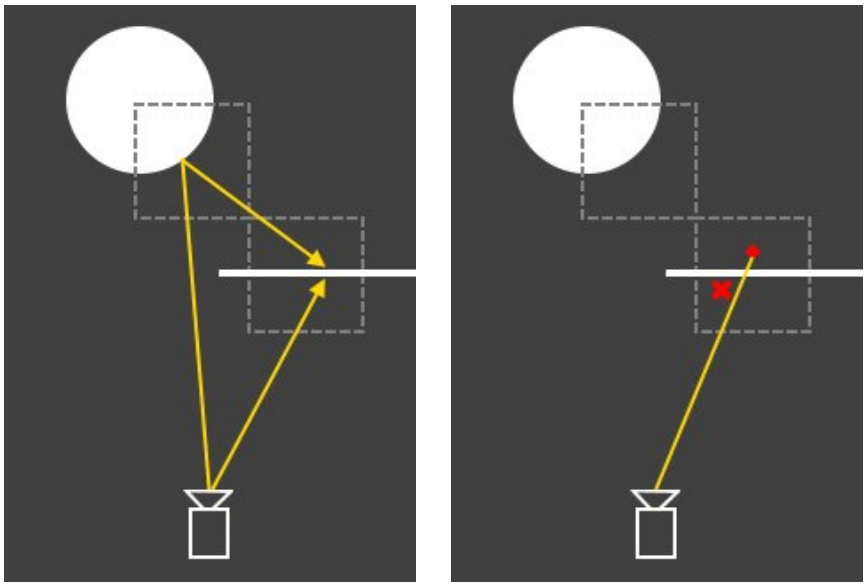


Figure 14.5. Light leak caused by tracing the origin, potentially originating from either side of the geometry.

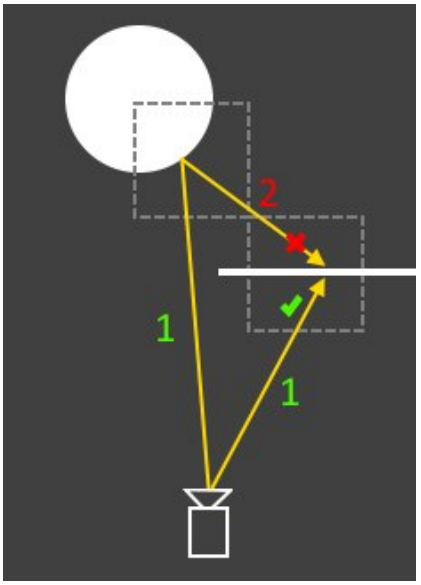


Figure 14.6. Rejecting trace origin candidates that have a greater rank compared to the entry rank.

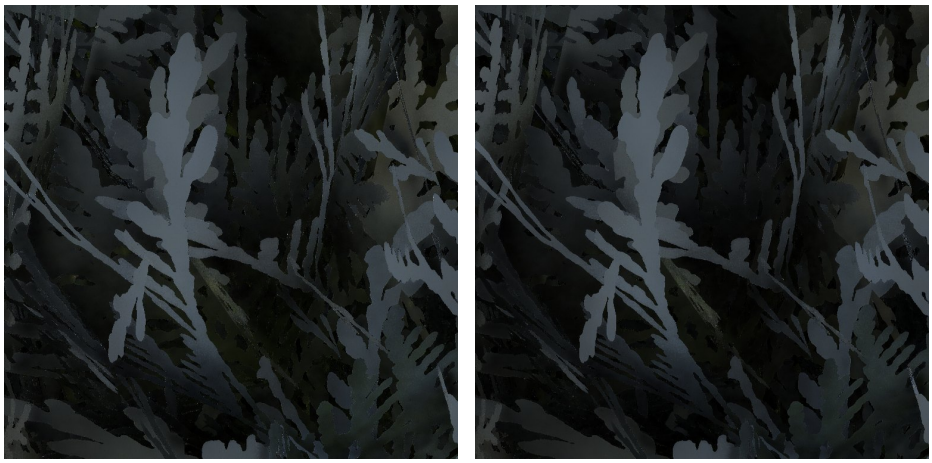


Figure 14.7. Self-lighting showing the irradiance cache in dense geometry in Amazon Lumberyard Bistro [Amazon Lumberyard 17].

14.2.8 Rejecting Queries from the Same Cell

In the scenario where a query that starts within a clipmap cell, hits the geometry within that same cell, invalid lighting is potentially sampled. (See Figure 14.7.) Optionally, queries that have a hit distance smaller than the size of a cell can be rejected. This preserves occlusion detail, at the cost of some darkening. However, it does prevent "boiling" artifacts from the temporally unstable irradiance cache from showing in such scenarios.

14.2.9 Sampling Cache from Mirror Reflections

Sampling the irradiance cache from mirror or almost mirror-like reflections will show the individual clipmap cells. There are two workarounds to this problem;

- **Add a jitter:** Adding a jitter to the cell indexing based on the tangent plane of the sampling location drastically helps here, but is prone to some initial light leaks. The ranking system should automatically take care of these situations, however it may produce unwanted results in more complex geometry situations.
- **Modify periodic shift:** Modifying the lookup periodic shift in a way that it'll cover the entire cell, along with a tiny bit of jitter, hides these artifacts. However, just like jittering the cell by a larger number, this is prone to initial light leaks.

Both of these workarounds heavily rely on the ranking system as stochastically sampling neighbor cells may cause indices that leak through geometry. In

	Timings	Bounce Rays	Shadow Rays
Trace candidates	823 µs	0.25 rpp	0.25 rpp
Temporal resampling	143 µs		
First spatial resampling	210 µs		
Second spatial resampling	200 µs		
Validate sample visibility	110 µs		0.25 rpp
Patch reservoirs	55 µs		
Resolve	1103 µs		
Total	2.64 ms	0.25 rpp	0.5 rpp

Table 14.2. ReSTIR GI running at half-resolution (native 1440p) on an RTX 3060, showing resulting rays per pixel (rpp).

these cases the neighbor cells will be assigned with origins that fall outside their cell position, essentially acting as additional cells to sample from in areas that matter the most.

14.3 Diffuse ReSTIR GI

All of the tracing and resampling kernels are run at half-resolution, where each candidate traces a single bounce path. (See Table 14.2.) In the case of a path not hitting a light source, the irradiance cache described in Section 14.2 is queried to measure the irradiance at the hit point. Once every three frames, the temporal reservoirs are re-evaluated by replaying the chosen paths [Stachowiak 22]. The temporal reservoir M is scaled down based on the difference in lighting contribution and hit distance. During spatial resampling, we trade correctness for speed by not testing visibility. Instead, the visibility is tested after two spatial resampling steps, which recovers detail at the cost of some bias. Finally, a resolve pass shades the final contribution and upscales it to the native resolution. A denoiser is still needed after the resolve pass, however relatively basic temporal reprojection in combination with a bilateral filter is already nearly enough.

14.3.1 Candidate Paths

Candidate paths are traced using a uniform hemisphere distribution [Ouyang et al. 21]. When using a cosine weighted direction, the likelihood that grazing angles are sampled is low, even if they provide the biggest lighting contribution. It's important to use a low-discrepancy sequence here such as blue noise; the higher quality the initial candidates are, the better ReSTIR performs in subsequent passes. These paths do not require many bounces as we are just trying to connect a single bounce diffuse path to one of the irradiance cache entries, which will inform the path of how much indirect lighting contributes to the hit

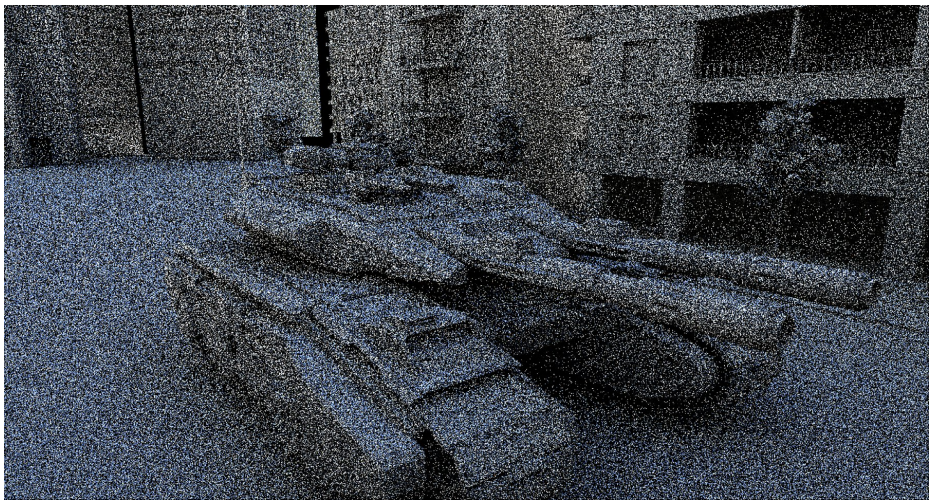


Figure 14.8. Candidate diffuse paths.

point. Since the irradiance cache only gives information for any indirect lighting, an explicit light sample needs to be traced as well from the hit point. (See Figure 14.8.)

The BRDF and sampling PDF are not applied during the candidate generation as we would need to demodulate the samples every time when trying to resample reservoirs. Instead, we need to store all the information needed to resample and shade the final chosen reservoir:

- sample direction,
- hit distance,
- hit normal,
- lighting contribution.

In order to reduce the memory footprint, this data is compressed to a total of 16 bytes using octahedron encoding [Cigolle et al. 14] for both the sample direction and hit normals, along with R9G9B9E5 for the lighting contribution.

The sample direction and hit distance can be reused to construct ray-traced ambient occlusion. Though this is not directly used during resampling, it may prove very useful during denoising passes.

14.3.2 Temporal Resampling

Applying the BRDF weight here would make the temporal samples cosine distributed, resulting in small or complex geometry to potentially have low-quality

samples available. To ensure these objects have high-quality samples, only the incoming lighting is used as the target function during temporal resampling, resulting in a uniform distribution. Some care is needed with reprojecting history data into the current frame, as artifacts produced by invalid reprojection are very difficult to filter. To avoid bad samples from propagating throughout multiple frames, depth and geometric normal rejection heuristics are used. The depth rejection needs to be very tight, however being too strict with normal rejection may decrease temporal stability with a moving camera. (See Listing 14.9.)

```
float relativeDepthDiff = abs(historyDepth / currentDepth - 1.0f);
bool rejectDepth = relativeDepthDiff > 0.03;

float normalSimilarity = dot(historyNormal, centerNormal);
// ReSTIR GI suggests rejecting angles outside of 25 degrees,
// which would be 0.86. Using a rejection value of 0.3 trades
// some accuracy for temporal stability.
bool rejectNormal = normalSimilarity < 0.3;
```

Listing 14.9. Rejection heuristics.

A Jacobian needs to be applied here to account for geometric differences [Ouyang et al. 21], as the resampling origin in world space does not stay the same throughout frames. A form of permutation sampling [Panteleev 21] is applied, which improves the temporal characteristics when sampling the history samples. The idea behind permutation sampling is to avoid correlations between pixels by using a per-frame jitter and afterward performing an ordered spatial exchange for each history pixel. Randomly sampling the history may by chance cause a sample to be reused between multiple pixels simultaneously. It's suggested to use an XOR value of 3, however this does not seem to perform very well on lower-resolution passes. Since the entire ReSTIR pipeline is ran at half-resolution, a slightly more complex sequence is needed to avoid block artifacts. (See Listing 14.10.)

```
int2 permutationSample(int2 samplePos, int2 jitter, uint sampleIndex
, uint frameIndex) {
    int2 result = samplePos;
    const uint sampleSeed = (frameIndex + sampleIndex) & 3u;
    const uint2 sequenceA = uint2(3u - sampleSeed, 1u + sampleSeed);
    const uint2 sequenceB = uint2(sequenceA.x ^ sampleSeed,
sequenceA.y ^ sampleSeed) & 3u;
    if (sampleIndex > 0) {
        result = (result + jitter) ^ sequenceA;
        result = (result + jitter) ^ sequenceB;
        result -= jitter;
    }
    return result;
}
```

Listing 14.10. Permutation sampling sequence.

To boost convergence, the temporal reuse pass also gathers additional reservoirs using permutation sampling in the scenario that a reservoir has a low sample

Figure 14.9. Permutation sampling.

count. (See Figure 14.9.) This is done until either the upper bound of permutations has been reached (5) or the reservoir M has reached or exceeded the M clamp value. Allowing a high M clamp during temporal reuse will reduce the workload for a denoiser, however this comes at the increased risk of temporal artifacts and responsiveness. Even when validating the temporal reservoirs, a higher M clamp tends to still be significantly less responsive and prone to correlation artifacts. ReSTIR GI suggests an M clamp of 20, however we found that an M clamp of 12 seems to be the sweet spot for responsiveness and convergence, bringing the best of both worlds.

With permutation sampling and gathering additional reservoirs, it's unknown if the spatial sample is actually visible from the origin of the pixel into which we are trying to merge. Ignoring this visibility term will result in loss of contact shadows along with blurry indirect shadows, both of which are unacceptable. Testing visibility for all of these samples is too expensive for a real-time implementation. As a workaround, a visibility guiding factor generated by the previous frame (covered in Section 14.3.5) is used as an additional rejection heuristic, which follows a fitted curve. (See Listing 14.11.)

```
    float validityDiff = exp2(-20 * abs(centerValidity - historyValidity
    ));
    // Only apply this validity rejection when gathering spatial
    // reservoirs (permutation sampling).
    if (validityDiff < 0.3 && sampleIndex > 0) {
        continue;
    }
```

Listing 14.11. Rejecting validity.

Figure 14.10. Permutation sampling with guiding.

If the guiding factor differs too much, the sample is rejected. This essentially creates a split between indirectly shadowed and non-shadowed pixels, which recovers contact shadows and preserves indirect shadows. (See Figure 14.10.)

14.3.3 Validating Reservoirs

Samples chosen by temporal reservoirs may change due to changes in the scene or in lighting conditions. If we don't update these reservoirs that potentially have stale data, we end up with very unresponsive indirect lighting. A good way to increase responsiveness is to re-trace the selected samples from the temporal reservoirs, however shooting an additional path per pixel just for the purpose of making the reservoirs responsive is very costly. Ideally, the validation work would be spread out over multiple frames; however, due to the nature of permutation sampling, this may not be possible as stale samples may survive validation if they are shared spatiotemporally. Instead, the reservoir validation is performed once every three frames. It's important to only validate history reservoirs that actually end up being reprojected into the new frame. (See Figure 14.11.)

For the pixels on screen that don't have a history available due to disocclusion etc., canonical paths are traced instead. This way all of the pixels during a validation frame will either have a history or candidates available, preventing any gaps in samples.

When the light sample is re-measured, the sample count is adjusted based on the lighting contribution and hit distance; if any of these factors show a big difference, we clamp the sample count of the reservoir down and update the

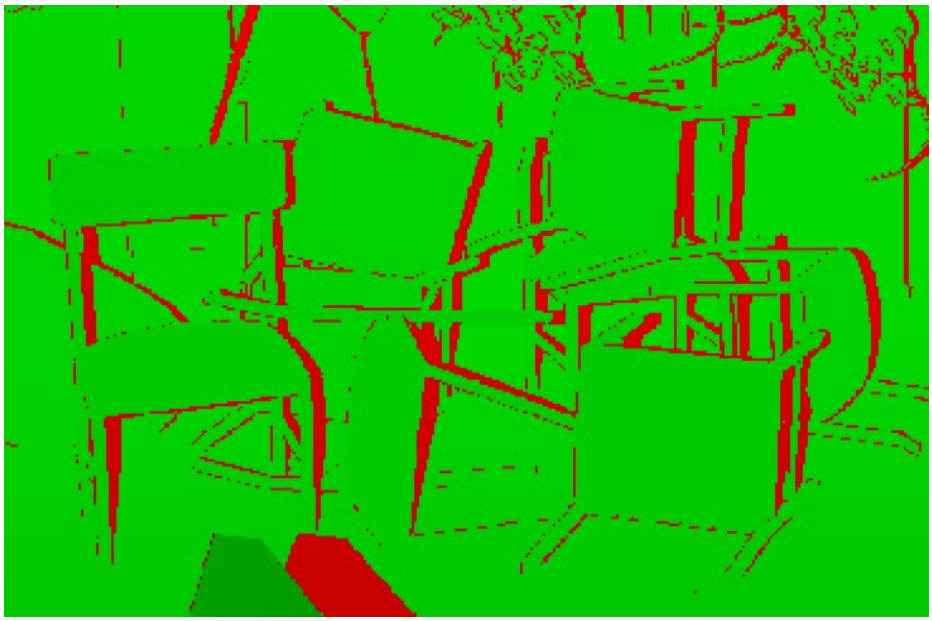


Figure 14.11. Tracing selection with a moving camera: green indicates that validation logic should be executed, and red indicates candidates generation.

stored selected contribution with the validated contribution value. This will make the temporal reservoirs very responsive to changes.

14.3.4 Spatial Resampling

Two spatial reuse passes are used in order to share as many unique reservoirs between pixels as possible. The first pass is aimed at less-complex geometry and resamples six neighbor reservoirs in a radius up to 42 pixels. The second pass is aimed toward scenarios where geometry is slightly more complex, such as vegetation or thin objects, and resamples up to five neighbor reservoirs in a radius up to 20 pixels. This second pass relies on the results of the first spatial reuse pass, effectively extending the search radius from the first pass while taking care of smaller details. The sampling radius is based on a fitted curve for each of the spatial reuse passes:

$$\text{first pass radius} = \left(\frac{\texttt{sampleIndex}}{\texttt{numSamples}}\right)^{0.5} \cdot \texttt{maxRadius},$$

$$\text{second pass radius} = \left(\frac{\texttt{sampleIndex}}{\texttt{numSamples}}\right)^{1.5} \cdot \texttt{maxRadius}.$$

Figure 14.12. Spatial reuse without visibility testing.

The sampling angle starts with a random offset and is incremented by the golden ratio for every iteration. Rejection heuristics are mostly identical to temporal resampling, with the exception of the validity factor. Additionally, unlike temporal reuse, the BRDF weight is used here in the target function when resampling, making the distribution cosine weighted again. It's very important to reconnect spatial samples to the target reservoir's origin rather than directly reusing the reservoir's sample direction. The use of a Jacobian is needed as well to account for geometric differences.

We chose to not test visibility in any of the spatial reuse passes (as in Figure 14.12), and instead we test the visibility of the final surviving sample. The contribution weight (W) of the reservoir is set to 0 if the visibility test failed. Applying a single deferred visibility test (as in Figure 14.13) is biased, as the sum of weights gathered in spatial resampling may contain contributions of samples that were not visible. That being said, in practice this provides a very nice trade-off as all contact shadows and indirect shadows have been recovered.

Alternatively, there are several different ways to approach visibility testing, such as ray-marching visibility in screen space [Stachowiak 22] or actually tracing visibility every time a reservoir is considered for reuse. Ray-marching the visibility is relatively cheap, but primarily suffers from screen-space limitations and lack of accuracy. Tracing a ray for each reuse is accurate but currently way too expensive for a real-time implementation.

The biggest performance win with resampling lies in coherent memory access and reducing the memory footprint. Since spatial resampling can only resample with information that originated from the temporal reuse pass, only a pixel

Figure 14.13. Spatial reuse with deferred visibility testing.

reference is stored for the selected reservoir sample data, which greatly reduces the memory footprint.

14.3.5 Visibility Guiding

The deferred spatial reservoir visibility test does provide some information that will be useful during the resolve and filtering passes. We know that in general samples with a very high contribution will be prioritized during resampling, meaning that most visible indirect shadows are likely going to be caused by these selected high-contribution samples. Treating the visibility result as a shadow factor gives us a result similar to ray-traced ambient occlusion, however this takes light contribution directionality into account, highlighting most of the contact shadow and indirect shadowing regions of the scene. (See Figure 14.14.)

Scenarios where indirect shadows and contact shadows are the sharpest/most visible mainly occur when the validated hit distance is substantially smaller compared to the target distance, while thin objects primarily have shadows when the light source has a relatively shallow angle with respect to the object. Based on these two observations, the Boolean hit value is modified to be a scalar by increasing the weight for shallow angles and very large hit differences. Since the guiding factor is exclusively used to reject resampling, we primarily care about the contrast between contact/indirect shadow regions and regularly indirectly lit surfaces; the factor does not need to be perfectly denoised to be effective. (See Figure 14.15.)

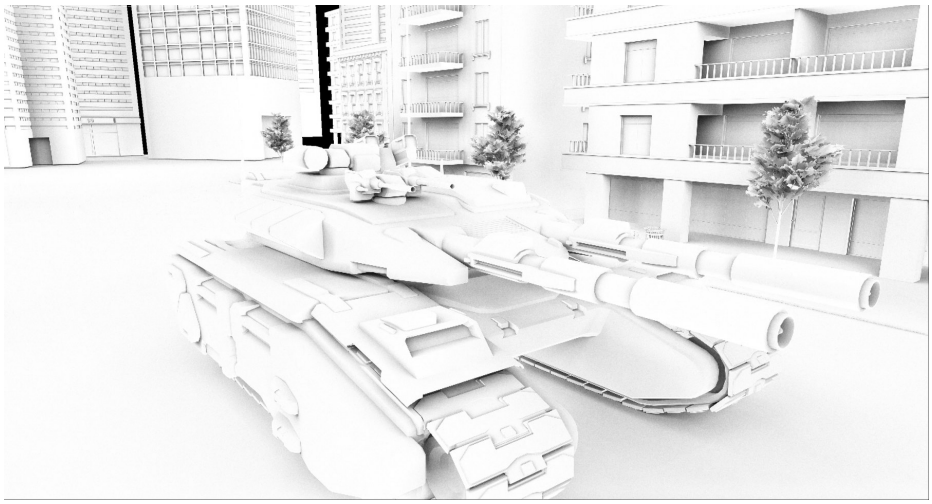


Figure 14.14. Accumulated visibility tests.

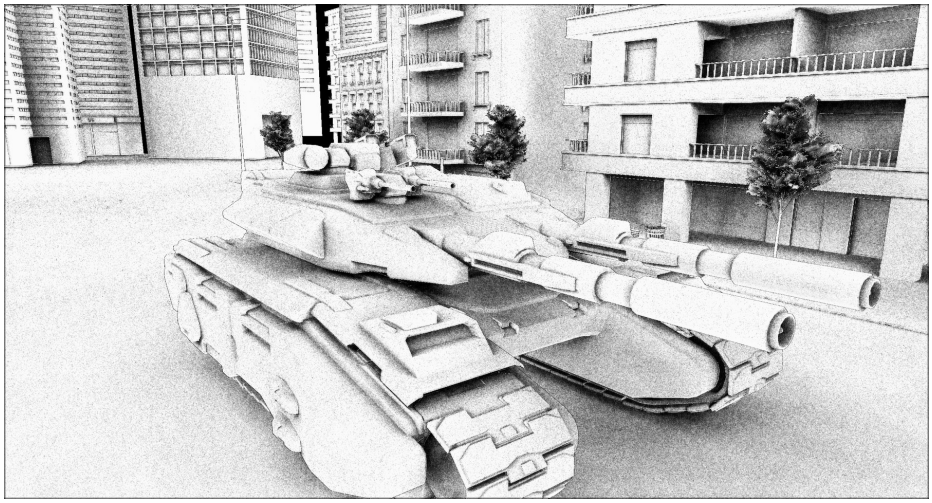
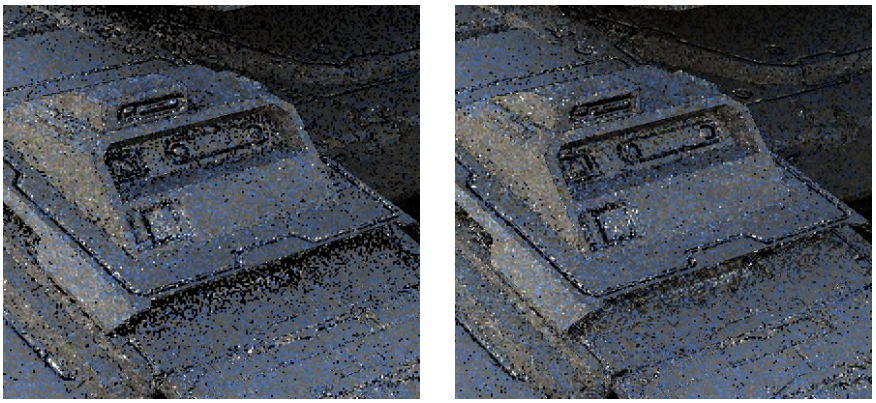


Figure 14.15. Visibility guiding factor.

14.3.6 Post-Validation Cleanup

After validating the spatial reuse reservoirs, the invalidated reservoirs will have zero contribution as we have no way to accurately represent these reservoirs, resulting in an increase of variance before the final resolve. Falling back to candidate or temporal reservoirs for these invalidated reservoirs is not a great idea either as this adds even more variance and potential outliers. Instead, a



(a) Validated output. (b) Patched reservoirs.

Figure 14.16. (a) Validated output showing excessive darkening due to invalidated reservoirs. (b) Patching invalidated reservoirs results in an overall higher-quality result.

tiny 2×2 pixel search is performed with very strict depth and normal rejection in order to find the closest valid reservoir (as in Figure 14.16). If a reservoir is found, its merged into the center pixel's reservoir. No additional rays are traced here to test visibility. If the reservoir survived the rejection heuristics, the neighbor sample is assumed to be visible as the origin is close enough to the target pixel.

14.3.7 Resolve

A final resolve pass is performed to upscale the half-resolution reservoirs to full resolution. (See Figure 14.17.) The history visibility guiding factor described in Section 14.3.5 is reprojected and updated with the new factor that was generated from the current frame. In order to reduce the workload for the denoising steps, multiple samples are gathered in a radius of up to 10 pixels, which are weighed based on visibility, normal, and depth similarity. In order to get away with aggressive neighbor sampling without losing too much detail, each sample is scaled by its weight and added to the L1 SH. After gathering all samples, the L1 SH is scaled by one over the weight sum and resolved to the shading normal of the pixel.

Resolving multiple pixels in a bigger area greatly improves temporal stability for the subsequent denoising passes, almost enough to remove all noise with temporal reprojection (Figure 14.18). Even though basic temporal reprojection already gets us very far, a more specialized denoiser is needed to get a completely artifact-free image under very fast camera movement.

Additionally, the guiding factor is used here to reject samples that differ too much in visibility, preventing the loss of contact/indirect shadows (Figure 14.19).

Figure 14.17. Resolving reservoirs to full resolution.

Figure 14.18. Temporal reprojection.

14.4 Future Work

A clipmap irradiance cache in combination with ReSTIR GI provides a good foundation for dynamic diffuse GI, however in highly dynamic and complex scenes there is still room for improvement. Temporal resampling tends to introduce pixel correlations that are very difficult to filter. Removing the temporal

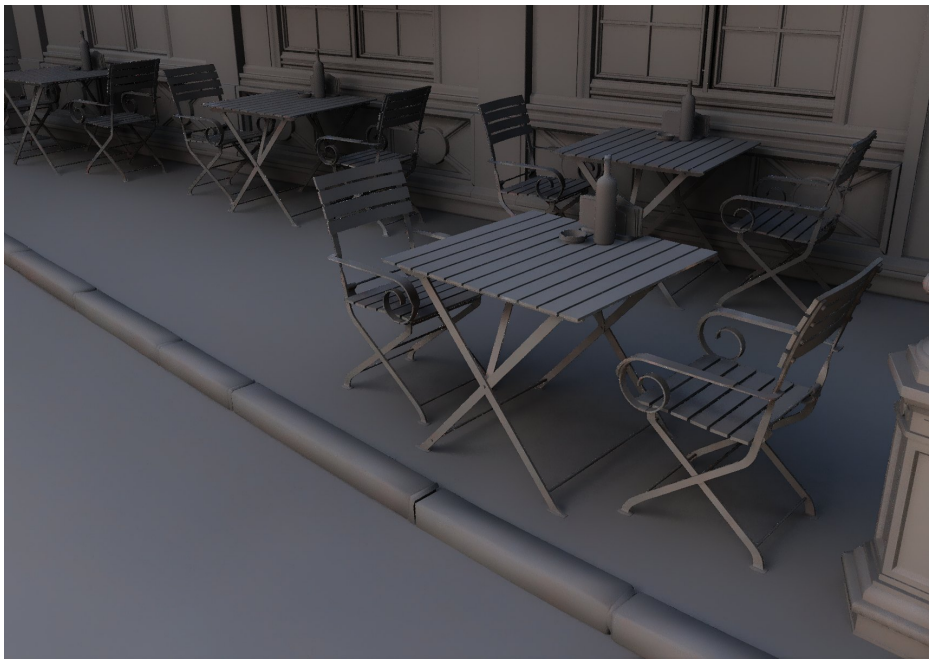


Figure 14.19. Preserving indirect and contact shadows with a visibility guiding factor in Amazon Lumberyard Bistro [Amazon Lumberyard 17].

part may solve these correlations, however this would greatly increase the workload for the denoiser. Additionally, there is still room for optimization. We chose to have a relatively expensive resolve pass in order to have the spatiotemporal denoiser do much less work, however this may not be necessary with a more involved denoising algorithm.

Finally, performing deferred ReSTIR visibility checks is biased, but works well enough for a believable GI solution. Tracing a visibility ray each time a reservoir is resampled is not practical for a game workload (yet). Alternatively, a better way to cache/reuse visibility could be explored, which would replace the visibility tests during the resampling of reservoirs.

Bibliography

[Amazon Lumberyard 17] Amazon Lumberyard. "Amazon Lumberyard Bistro." Open Research Content Archive (ORCA), 2017. http://developer.nvidia.com/orca/amazon-lumberyard-bistro.

[Bitterli et al. 20] Benedikt Bitterli, Chris Wyman, Matt Pharr, Peter Shirley, Aaron Lefohn, and Wojciech Jarosz. "Spatiotemporal Reservoir Resampling for Real-Time Ray Tracing with Dynamic Direct Lighting." *ACM Transactions on Graphics* 39:4 (2020), 148:1–148:17. https://cs.dartmouth.edu/wjarosz/publications/bitterli20spatiotemporal.html.

[Cigolle et al. 14] Zina H. Cigolle, Sam Donow, Daniel Evangelakos, Michael Mara, Morgan McGuire, and Quirin Meyer. "A Survey of Efficient Representations for Independent Unit Vectors." *Journal of Computer Graphics Techniques* 3:2 (2014), 1–30. https://jcgt.org/published/0003/02/01/.

[Gautron 22] Pascal Gautron. "Advances in Spatial Hashing: A Pragmatic Approach towards Robust, Real-Time Light Transport Simulation." In *SIGGRAPH '22: ACM SIGGRAPH 2022 Talks*, pp. 6:1–6:2. Association for Computing Machinery, 2022. https://doi.org/10.1145/3532836.3536239.

[Hazel et al. 15] Graham Hazel, Chris Doran, and William Joseph. "Geomerics Reconstructing Diffuse Lighting." Presented at Computer Entertainment Developers Conference, 2015. https://web.archive.org/web/20160313132301if_/http://www.geomerics.com/wp-content/uploads/2015/08/CEDEC_Geomerics_ReconstructingDiffuseLighting1.pdf.

[Ouyang et al. 21] Y. Ouyang, S. Liu, M. Kettunen, M. Pharr, and J. Pantaleoni. "ReSTIR GI: Path Resampling for Real-Time Path Tracing." *Compututer Graphics Forum* 40:8 (2021), 17–29. http://dx.doi.org/10.1111/cgf.14378.

[Panteleev 21] Alexey Panteleev. "Permutation Sampling." X (formerly Twitter), November 8, 2021. https://twitter.com/more_fps/status/1457749362025459715.

[Roberts 18] Martin Roberts. "The Unreasonable Effectiveness of Quasirandom Sequences." Extreme Learning, April 25, 2018. https://extremelearning.com.au/unreasonable-effectiveness-of-quasirandom-sequences/.

[Shyshkovtsov et al. 19] Oles Shyshkovtsov, Sergei Karmalsky, Benjamin Archard, and Dmitry Zhdan. "Exploring Raytraced Future in Metro Exodus." Presented at Game Developers Conference, 2019. https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9985-exploring-ray-traced-future-in-metro-exodus.pdf.

[Stachowiak 22] Tomasz Stachowiak. "Global Illumination Overview." GitHub, 2022. https://github.com/EmbarkStudios/kajiya/blob/main/docs/gi-overview.md.

[Wyman and Panteleev 22] Chris Wyman and Alexey Panteleev. "Rearchitecting Spatiotemporal Resampling for Production." In *High-Performance Graphics 2021—Symposium Papers*, pp. 23–41. Eurographics Association, 2022. https://doi.org/10.2312/hpg.20211281.