WORLD-SPACE SPATIOTEMPORAL RESERVOIR REUSE FOR RAY-TRACED GLOBAL ILLUMINATION

Guillaume Boissé
Guillaume.Boisse@amd.com
Advanced Micro Devices, Inc.
France

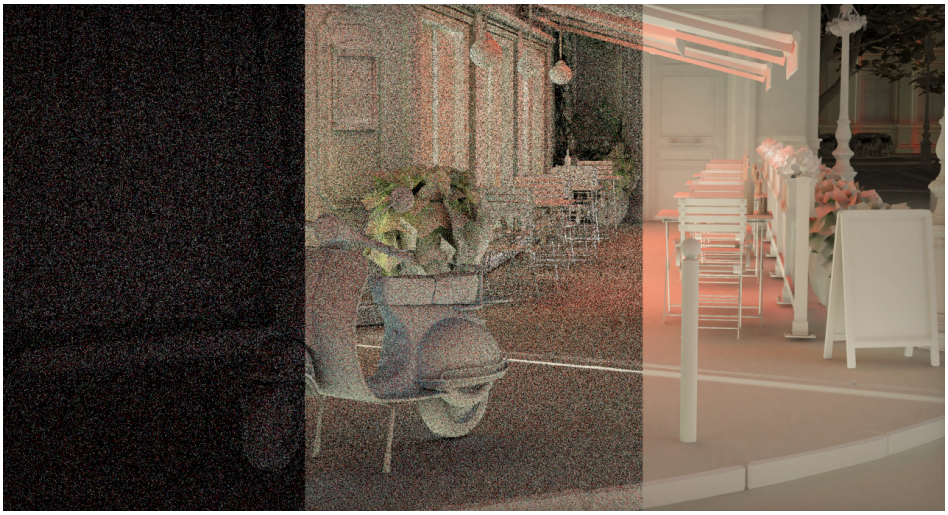


Figure 1: Bistro scene with ~30,000 area lights rendered with indirect lighting only, at 2560x1440 and 1 sample per pixel (spp): diffuse global illumination with next event estimation (NEE) (left), with our world-space reservoir caching (middle), and a spatiotemporal denoiser (right).

ABSTRACT

Path-traced global illumination of scenes with complex lighting remains particularly challenging at real-time framerates. Reservoir-based resampling methods for light sampling allow for significant noise reduction at the cost of very few shadow rays per pixel. However, current image-space approaches to reservoir reuse do not scale to sample lighting at further bounces, as is required for efficiently evaluating indirect illumination.

We present a novel approach to performing reservoir-based spatiotemporal importance resampling in world space, allowing for efficient light sampling at arbitrary vertices along the eye path. Our approach caches the reservoirs of the path vertices into the cells of a hash grid built entirely on the GPU. Such a structure allows for stochastic reuse of neighboring reservoirs across space and time for efficient spatiotemporal reservoir resampling at any point in space.

SA '21 Technical Communications, December 14–17, 2021, Tokyo, Japan

ACM ISBN 978-1-4503-9073-6/21/12…$15.00
https://doi.org/10.1145/3478512.3488613

CCS CONCEPTS

• **Computing methodologies** → **Rendering**; **Ray tracing**.

KEYWORDS

global illumination, hash grid, ray tracing, reservoir resampling

ACM Reference Format:
Guillaume Boissé. 2021. WORLD-SPACE SPATIOTEMPORAL RESERVOIR REUSE FOR RAY-TRACED GLOBAL ILLUMINATION. In *SIGGRAPH Asia 2021 Technical Communications (SA '21 Technical Communications), December 14–17, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3478512.3488613

1 INTRODUCTION

Reservoir-based Spatiotemporal Importance Resampling (ReSTIR) [Bitterli et al. 2020] recently introduced some exciting improvements for sampling complex scene lighting for ray-traced direct illumination, which Rearchitecting Spatiotemporal Resampling for Production [Wyman and Panteleev 2021] further optimizes, making the technique suitable for use in real-time applications such as games.

ReSTIR GI [Ouyang et al. 2021] is an extension of the algorithm for global illumination that resamples the paths themselves to achieve interesting noise reduction for indirect lighting. However, the approach does not allow sampling of the lighting distribution for secondary vertices as effectively as ReSTIR does for direct lighting, which is desirable to reduce the noise even further.

Fast path-space filtering [Binder et al. 2018] proposes to use spatial hashing to efficiently find all neighboring path vertices within a region of world space in a single grid lookup.

Finally, [Jarzynski and Olano 2020] published a study comparing the performance of many hash functions executing on the GPU allowing the fastest hash function at a given quality level to be selected.

We propose combining the path-space filtering approach from [Binder et al. 2018] with the light sampling of ReSTIR to perform reservoir reuse at secondary path vertices. We describe how to adapt the hash-grid structure to achieve efficient reuse in a single cache lookup resulting in significant noise reduction in the indirect illumination of complex scenes.

2 PRELIMINARIES

The ReSTIR technique achieves important noise reduction for direct lighting of complex scenes by sharing sampling decisions across neighboring pixels via Resampled Importance Sampling (RIS) [Talbot et al. 2005] and Weighted Reservoir Sampling (WRS) [Chao 1982].

Direct lighting samples are generated for every pixel on the screen by drawing M samples x_i from a poor-quality distribution p, which is resampled against a higher-quality distribution $\hat{p}$. [Bitterli et al. 2020] proposes to use power sampling for p as an easy-to-sample distribution, but it is interesting to note that any other light sampling technique can be used to improve the quality of the initial sample generation. For $\hat{p}$ we typically want the distribution to be as close as possible to the lighting function f used for shading. However, calculating f accurately requires tracing a ray to evaluate the visibility term, which is prohibitively expensive. Instead, we can define $\hat{p}$ as the full lighting term without visibility, otherwise known as the unshadowed illumination. This process allows the generation of an initial reservoir for each pixel, as shown in Algorithm 1.

class *Reservoir*
  $y \leftarrow 0$ // The currently best light sample
  $w_{\text{sum}} \leftarrow 0$ // The sum of resampling weights
  $M \leftarrow 0$ // The number of streamed samples
  $W \leftarrow 0$ // The final resampling weight

  function *WRS*(x_i, w_i)
    $w_{\text{sum}} \leftarrow w_{\text{sum}} + w_i$
    $M \leftarrow M + 1$
    if $rand() < \frac{w_i}{w_{\text{sum}}}$
      $y \leftarrow x_i$

function *RIS*($pixel\ q$)
  $Reservoir\ r$
  for $i \leftarrow 1$ **to** M **do**
    generate $x_i \sim p$
    $r.WRS(x_i, \frac{\hat{p}(x_i)}{p(x_i)})$
  $r.W \leftarrow \frac{1}{\hat{p}(r.y)} \cdot (\frac{1}{r.M} \cdot r.w_{\text{sum}})$
  return r

foreach $pixel\ q \in Image$ **do**
  $Reservoir\ r \leftarrow RIS(q)$
  $Image[q] \leftarrow f(r.y) \cdot r.W$

Algorithm 1: Initial reservoir generation and shading

An interesting property of reservoirs is that they can be combined with one another in a single WRS operation that is mathematically equivalent to having resampled the two combined input streams while remaining computationally efficient. This allows the correlations between neighboring pixels to be efficiently leveraged by reusing reservoirs across space and time in image space.

3 WORLD-SPACE RESERVOIR CACHING

To extend the reservoir reuse to further path vertices, we require a mechanism to find neighboring vertices efficiently in world space. Path-space filtering introduces the use of a hash grid to perform filtering of radiance samples by averaging all the values within a cell [Binder et al. 2018]. This approach is particularly well suited to the massively parallel nature of GPUs, making it a good candidate for our reservoir cache. For reservoir reuse however, we want to recombine multiple reservoirs within the vertex neighborhood to achieve high counts of streamed samples and reduce noise appropriately.

We propose adapting the data structure and, instead, storing lists of reservoirs in each cell of the hash grid. As we will demonstrate, this allows for efficient sharing of reservoirs across path neighbors in world space, thus achieving high numbers of streamed samples with a low performance overhead. To build this structure efficiently on the GPU, we have broken the process down into steps as shown in Algorithm 2.

foreach $Vertex\ v \in Paths$ **do**
  $Reservoir\ r \leftarrow RIS(v)$
  $Paths[v] \leftarrow r$ // Store initial reservoir
  $cell\_index \leftarrow FindOrInsertCell(v)$ // See Section 5
  $index\_in\_cell \leftarrow atom\_inc(cell\_counters[cell\_index])$
  $append\_buffer.Append(< v, cell\_index, index\_in\_cell >)$

$index\_buffer \leftarrow ParallelPrefixSum(cell\_counters)$

foreach $triplet \in append\_buffer$ **do**
  $base\_offset \leftarrow index\_buffer[triplet[1]]$
  $scatter\_offset \leftarrow base\_offset + triplet[2]$
  $cell\_storage[scatter\_offset] \leftarrow triplet[0]$

Algorithm 2: Building a reservoir hash grid on the GPU

Incrementing the per-cell counters. For each path vertex v, we resolve the *cell_index* within the hash grid and atomically increment the number of reservoirs in the cell. Additionally, we save out the return value from the atomic operation; this is the index at which to scatter the reservoir within the cell storage, which we call *index_in_cell*. We store all the required information by appending the triplet $< v, cell\_index, index\_in\_cell >$ to an *append_buffer*.

Parallel prefix sum. We perform a parallel prefix sum operation over the cell counters [Harris et al. 2007] and write the results to an *index_buffer* which effectively stores the base offset for each cell.

Scattering the reservoirs into a compacted stream. We scatter the reservoirs into their respective locations within the *cell_storage* buffer. This is done by loading the triplets stored in *append_buffer*; we retrieve the cell's *base_offset* by reading from *index_buffer* at *cell_index* to which we add *index_in_cell* to get the memory location at which to scatter the reservoir. Note that we only scatter the index of the reservoir to save on the memory bandwidth.

4 WORLD-SPACE SPATIOTEMPORAL RESERVOIR REUSE

We perform spatiotemporal reservoir reuse similarly to [Bitterli et al. 2020] but use our hash grid rather than an image-space kernel for finding the neighboring reservoirs. For temporal reuse, we simply keep the hash grid from the last frame to reuse previous reservoirs.

The most obvious way to reuse reservoirs from the hash-grid cache is to resolve the cell index of the current path vertex and perform weighted reservoir sampling on all reservoirs within the cell. However, this yields strong visual artefacts due to all neighboring vertices reusing the same set of reservoirs as shown in Figure 2. These tiling artefacts are particularly undesirable in the context of rendering as they tend to produce structured noise that is typically very difficult to filter with a denoiser. Additionally, cells can contain varying numbers of reservoirs depending on the number of path vertices landing in a given region of space resulting in potentially poor performance during reuse.

(a) Naïve reuse **(b) Stochastic selection** **(c) Jittered descriptors**

Figure 2: World-space reservoir resampling used for direct lighting

Stochastic reservoir reuse. Instead, we opt for a stochastic selection of reservoirs within the cell therefore resampling only a random subset of *max_count* elements from the reservoir list. This is made possible by the flattened nature of the lists inside each grid cell allowing for random access to any element. Stochastic reservoir reuse helps break down the tiling artefacts, as shown in Figure 2.

foreach $Vertex\ v \in Paths$ **do**
  $Reservoir\ s \leftarrow Paths[v]$
  $M \leftarrow s.M$
  $< cell_{\text{start}}, cell_{\text{end}} > \leftarrow FindCell(v)$ // See Section 5
  $count \leftarrow cell_{\text{end}} - cell_{\text{start}}$
  $increment \leftarrow (count + max\_count - 1)/max\_count$
  $offset \leftarrow Rand(0, increment - 1)$
  for $i \leftarrow 0$ **to** $count - 1$ **by** $increment$ **do**
    $Reservoir\ r \leftarrow cell\_storage[cell_{\text{start}} + ((i + offset)\%count)]$
    $s.WRS(r.y, \hat{p}(r.y) \cdot r.W \cdot r.M)$
    $M \leftarrow M + r.M$
  $s.M \leftarrow M$
  $s.W \leftarrow \frac{1}{\hat{p}(s.y)} \cdot (\frac{1}{s.M} \cdot s.w_{\text{sum}})$
  $Paths[v] \leftarrow s$ // Store resampled reservoir

Algorithm 3: Stochastic world-space reservoir reuse

Jittered descriptors. Similarly to [Binder et al. 2018], we jitter the hash-grid descriptors during both insertions and lookups of the reservoir cache. We implement this by simply jittering the position of the path vertex in the tangent plane proportional to the cell size prior to inserting or searching the hash grid. This finishes breaking down the tiling artefacts, resulting in denoiser-friendly noise that can be filtered.

Adaptive cell size. At a distance, the projected size of a cell becomes smaller making it less likely to be hit by multiple rays, therefore reducing the resampling quality due to the lack of reservoir availability. We modify our cell definition and apply an adaptive cell size, aiming to obtain a roughly constant projected size in pixels.

We derive a formula listed in Algorithm 4 that calculates the optimal cell size for a given position in space v and a virtual camera positioned at *eye*, with a vertical field of view of $fovY$ radians and a sensor size of *width* by *height* pixels. We introduce two parameters; *min_cell_size* describing the minimum cell size in world space and *projected_size* the targeted cell size projected to pixels.

function *CalculateCellSize*($Vertex\ v$)
  $cell\_size\_step \leftarrow distance(v, eye) \cdot tan(projected\_size \cdot fovY \cdot max(\frac{1}{height}, \frac{height}{width^2}))$
  $log\_step \leftarrow floor(log2(\frac{cell\_size\_step}{min\_cell\_size}))$
  return $min\_cell\_size \cdot exp2(log\_step)$

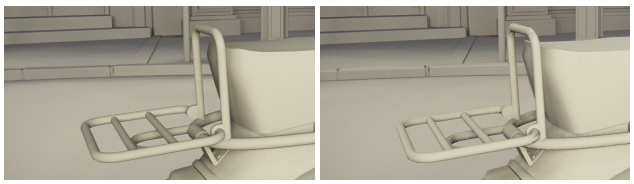
Algorithm 4: Calculating an adaptive cell size

Figure 3 shows a comparison with and without adaptive cell size, highlighting the additional noise at a distance due to insufficient reservoirs being available for effective reuse.

(a) Constant cell size **(b) Adaptive cell size**

Figure 3: Reservoir reuse with and without adaptive cell size

Bias reduction. The original ReSTIR algorithm proposes using the depth and normal information to filter neighboring reservoirs during reuse to reduce some of the over darkening caused by the bias. In the case of our hash grid, the spatial proximity is guaranteed due to sampling from cells on a grid. Therefore, we only store the world-space normal used to generate each reservoir and compare with the vertex normal during reuse. If the normals differ too much, we skip the reservoir and move on to the next one, helping to reduce the amount of bias introduced in the image as shown in Figure 4.



(a) No bilateral filtering **(b) Normal-based bilateral threshold**

Figure 4: Bias reduction via normal-based bilateral thresholding

5 IMPLEMENTATION AND RESULTS

We have implemented world-space ReSTIR for single-bounce diffuse global illumination (GI). We spawn GI rays from the primary visible surfaces by cosine-weighted sampling of the hemisphere using blue noise [Heitz et al. 2019]. We then generate reservoirs by resampling 16 initial samples and insert the outcome into the hash grid.

Inserting into the hash grid. Using spatial hashing may introduce conflicts where two distinct locations of world space that should resolve to different cells end up at the same index. To solve this issue, we use a secondary hash function and calculate a 32-bit checksum; this is compared with all entries within the cell to find the correct memory location, a process usually referred to as linear probing. In our implementation, we allocate 100,000 cells and allow for up to 32 entries in each cell. Additionally, we leverage the study by [Jarzynski and Olano 2020] and find that using *pcg*() as the primary hash function (referred to as $h1$) and *xxhash32*() as the secondary hash function (referred to as $h2$) gives good entry distribution.

```
function FindOrInsertCell(Vertex v)
    p_size ← CalculateCellSize(v)
    p ← floor(v/p_size)
    cell_index ← h1(p_size + h1(p.z + h1(p.y + h1(p.x))))%100,000
    checksum ← max(h2(p_size + h2(p.z + h2(p.y + h2(p.x)))), 1)
    for i ← 0 to 31 do
        idx ← i + cell_index * 32
        checksum_prev ←
          atom_cmpxchg(checksum_buffer[idx], 0, checksum)
        if checksum_prev = 0 or checksum_prev = checksum
            break // Found or inserted an entry :)
    if i = 32
        return −1 // Out of memory :(
    return i + cell_index * 32
```

Algorithm 5: Inserting into the hash grid

For looking up the hash grid, we define a *FindCell*() variation of the function listed in Algorithm 5 which simply compares every entry with the desired checksum, without using *atom_cmpxchg*().

Results. We have tested our technique on diffuse global illumination rendered at half resolution from 1080p using one temporal and four spatial world-space reuse passes. We set the reuse *max_count* to 4 and achieve significant noise reduction in the scene indirect illumination as shown in Figure 5.

(a) Hangar scene #1 (7,832 area lights) **(b)** Bistro exterior #1 (30,672 area lights) **(c)** Bistro interior #1 (37,436 area lights)

Figure 5: Top: 1spp GI with NEE, bottom: world-space ReSTIR

We realize real-time levels of performance on complex scenes featuring many area lights using only ¼ sample per pixel and our world-space resampling technique. We report the overhead of world-space ReSTIR for a single frame of animation in Table 1.

Table 1: World-space ReSTIR performance results (in ms)

	Reservoir generation		Hash grid building		Temporal reuse (1x)		Spatial reuse (4x)	
	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
Hangar scene #1	0.298	0.429	0.252	0.265	0.318	0.509	1.051	1.698
Hangar scene #2	0.294	0.441	0.262	0.278	0.286	0.449	1.023	1.537
Bistro exterior #1	0.313	0.394	0.196	0.184	0.304	0.444	1.115	1.918
Bistro exterior #2	0.239	0.307	0.153	0.165	0.237	0.276	0.799	1.278
Bistro interior #1	0.285	0.356	0.126	0.164	0.342	0.523	1.303	2.553
Bistro interior #2	0.201	0.251	0.097	0.128	0.280	0.427	0.861	1.531

6 CONCLUSION

We have presented a novel way of caching reservoirs in world space using spatial hashing. We have shown that our technique allows for efficient reservoir-based spatiotemporal resampling for light sampling at secondary vertices, thus bringing significant noise reduction to indirect lighting calculations for a small performance overhead.

Limitations and future work. We have implemented world-space ReSTIR for single-bounce global illumination. In the future, we want to look at updating the hash grid as opposed to rebuilding it each frame, so that further vertices can be added as the paths bounce around. Eventually, we see using the hash-grid structure for direct lighting sampling as well, resulting in a single unified reservoir cache for all illumination. Finally, our approach should combine well with both the ReSTIR improvements brought by [Wyman and Panteleev 2021] and the path resampling technique presented by [Ouyang et al. 2021].

REFERENCES

Nikolaus Binder, Sascha Fricke, and Alexander Keller. 2018. Fast Path Space Filtering by Jittered Spatial Hashing. In *Proc. SIGGRAPH.* ACM, 1–2.

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 Transactions on Graphics (Proceedings of SIGGRAPH)* 39, 4 (July 2020).

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

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, Laurent Belcour, Victor Ostromoukhov, David Coeurjolly, and Jean-Claude Iehl. 2019. A Low-Discrepancy Sampler that Distributes Monte Carlo Errors as a Blue Noise in Screen Space. In *ACM SIGGRAPH Talk.*

Mark Jarzynski and Marc Olano. 2020. Hash Functions for GPU Rendering. *Journal of Computer Graphics Techniques (JCGT)* 9, 3 (17 October 2020), 20–38.

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).

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.

Chris Wyman and Alexey Panteleev. 2021. Rearchitecting Spatiotemporal Resampling for Production. In *High-Performance Graphics - Symposium Papers*, Nikolaus Binder and Tobias Ritschel (Eds.). The Eurographics Association.