CHAPTER 23

RENDERING MANY LIGHTS WITH GRID-BASED RESERVOIRS

Jakub Boksansky, Paula Jukarainen, and Chris Wyman
NVIDIA

ABSTRACT

Efficient rendering of scenes with many lights is a longstanding problem in graphics. Sampling a light to shade from the pool of all lights, e.g., using next event estimation, is a nontrivial task. Sampling must be computationally efficient, must select lights contributing significantly to the shaded point, and must produce low noise while introducing little or no bias. Typically, the light pool is preprocessed to create a data structure that accelerates sampling queries; this may be complex to implement, build, and update.

This chapter builds a new sampling algorithm, *ReGIR*, based on a simple uniform grid structure and the recent screen-space resampling algorithm ReSTIR, which we extend to sample secondary rays in world space.

23.1 INTRODUCTION

Real-time ray tracing enables us to render more realistic images in games than was possible before. Accurate shadows, indirect illumination, and reflections have already been implemented in recent games. In combination with suitable denoisers, we can use ray tracing to render scenes lit by many more lights, without precomputing the static lighting, and with support for dynamic worlds. This is an attractive goal that can differentiate ray traced games from rasterized games in a significant way. Furthermore, we do not have to rely only on analytic lights, as we can also use emissive geometry for lighting.

This is possible by using shadow rays instead of shadow mapping to resolve visibility, avoiding the performance cost of preparing a shadow map for every light source, but also introducing the cost of a denoiser to filter the results. Though shadow rays are a much more flexible approach than shadow mapping, rendered images will be inherently noisy when lit by many different lights.

A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_23

Figure 23-1. *The Amazon Lumberyard Bistro [1], containing 65,535 emissive triangles, rendered with our ReGIR method using Falcor's [2] path tracer and denoised.*

In this chapter, we focus on a problem of selecting the best light for shading in a way to reduce noise and ensure consistent results.

23.2 PROBLEM STATEMENT

In virtual scenes, there are usually multiple lights simultaneously illuminating each point. (See, for example, Figure 23-1). But when scenes have hundreds or thousands of lights, identifying those actually contributing is a difficult task. Without evaluating the bidirectional reflectance distribution function (BRDF) and visibility of all lights, any one is a potential candidate.

An approach that simply selects a random light without considering visibility gives noisy images, as occluded or distant lights are as likely to be sampled as nearby, visible lights. Importance sampling improves the results (see the pseudocode in Figure 23-2), as the light selection probability varies based on its contribution, selecting lights with higher contribution more often.

Ideally, this probability varies based on the full BRDF and a visibility test; this gives outstanding samples, but is expensive to evaluate (i.e., as expensive as using all lights to shade). Good results are achievable by skipping the expensive visibility test and only evaluating the geometry term (cosine term and distance-based attenuation). This quickly discards backfacing lights and those that are dim or distant. Artists can also design scenes with this in mind,

```
function directLighting(Point p, Direction ω_o)
    result = 0;
    for i ∈ {1 . . . SAMPLE_COUNT} do
        light, pdfLight = SampleLight(p);
        if ShadowRay(p, light) then
            result += Brdf(light, p, ω_o) / pdfLight;
    return (result / SAMPLE_COUNT);
```

Figure 23-2. *Importance sampling loop taking the selected number of light samples at point p, observed from direction ω_o, and averaging them. A basic implementation might select a light randomly from the light pool and set* pdfLight = 1/LIGHTS_COUNT.

using lights with limited emission profiles for quick culling, e.g., spotlights. Because typical distance-based attenuation never reaches zero, modifying attenuation functions to quickly reach zero helps limit light range and further improves performance.

With importance sampling, we can reduce noise by selecting highly contributing lights more often, but basic implementations must still evaluate the probabilities of all the lights to determine which to sample.

23.2.1 RESAMPLED IMPORTANCE SAMPLING

To reduce this cost, we use *resampled importance sampling* (RIS) [5], a powerful technique to numerically sample distributions that are difficult to analytically sample. RIS is a two-step process. First, we select M candidates from the pool of all N samples using a cheap-to-evaluate *source* probability density function (PDF), e.g., uniformly. Then we resample from the M-element subset according to a more expensive *target* PDF. The target PDF may be the ideal distribution discussed previously, or it could include some mixture of BRDF, visibility, geometry, and light contributions. Note that the target PDF is only evaluated for M samples, which is significantly less than N. Resampling can be efficiently implemented using *weighted reservoir sampling* (WRS), described in Chapter 22, and is summarized in Figure 23-3.

Resampling is a key building block of the recently introduced ReSTIR (reservoir spatiotemporal importance resampling) algorithm [3], designed to render scenes with many lights without maintaining a complex data structure. ReSTIR maintains a subset of light samples (called a reservoir) per pixel. The algorithm starts by sampling lights in an inexpensive way, then uses

```
Struct LightSample
    uint lightID;
Struct Reservoir
    LightSample sample;
    uint M;
    float totalWeight;
    float sampleTargetPdf;
function sampleLightRIS(Point p)
    Reservoir r;
    for i ∈ {1 . . . M} do
        candidate, sourcePdf = sampleFromSourcePool();
        targetPdf = PartialBrdf(candidate, p);
        risWeight = targetPdf / sourcePdf;
        r.totalWeight += risWeight;
        r.M++;
        if rand() < (risWeight / r.wSum) then
            r.sample = candidate;
            r.sampleTargetPdf = targetPdf;
    resultWeight = (r.totalWeight / M) / r.sampleTargetPdf;
    return r.sample, resultWeight;
```

Figure 23-3. *Basic RIS using weighted reservoir sampling to sample lights. This example calls* sampleFromSourcePool *to draw samples using an inexpensive method and resamples according to* PartialBrdf *for the shaded point p. The weight of the selected sample* resultWeight *is used as the inverse PDF in importance sampling, during shading. The reservoir structure stores the selected sample and metadata about its construction as explained in Section 23.2.2. The* LightSample *structure simply references a sampled light by its index.*

resampling to spatially and temporally reuse neighbor samples. ReSTIR further improves samples by combining existing reservoirs into new ones, a key component to continuously improve the reservoir in any pixel over many frames.

23.2.2 RESERVOIR

A reservoir, as defined by Bitterli et al. [3], is a data structure that holds one or more samples selected from a larger set (our implementation uses reservoirs of one light sample). A reservoir also stores metadata about how it was constructed, specifically the number of candidates evaluated during its construction M, their total weight, and the target PDF of the selected sample

(see the pseudocode in Figure 23-3). This data is needed for using the reservoir to perform unbiased sampling.

There are multiple ways to construct a reservoir; the first that we will cover is based on the RIS algorithm. Going back to our description of RIS, we can see how they are related: a reservoir is a structure holding a subset of a larger set, and RIS is the algorithm producing such a subset. The reservoir metadata are a byproduct of the resampling process; i.e., the number of candidates *M*, their total weight, and the target PDF of selected samples are all used in the code in Figure 23-3 when running a RIS algorithm, and they can be stored in the reservoir directly.

Another way to create a reservoir is by merging multiple reservoirs into one using the procedure shown in Figure 23-4. As mentioned in the problem statement of this section, this is a key component of ReSTIR and a powerful tool enabling us to construct large numbers of independent reservoirs in parallel using RIS and to merge them to obtain even higher quality sample sets.

The original ReSTIR algorithm works in screen space, meaning it samples lights for pixels on screen and hence only works for primary rays. Our method adapts the ideas in ReSTIR to work in world space. Instead of maintaining a reservoir per pixel, we create a coarse world-space grid and maintain a number of independent reservoirs in each voxel.

```
function updateReservoir(Reservoir result, Reservoir input)
    result.totalWeight += input.weight;
    result.M++;
    if rand() < (input.weight/result.totalWeight) then
        result.sample = input.sample;
        result.sampleTargetPdf = input.sampleTargetPdf;
    return result;
function mergeReservoirs(Reservoir r1, Reservoir r2)
    Reservoir merged;
    updateReservoir(merged, r1);
    updateReservoir(merged, r2);
    merged.M = r1.M + r2.M;
    return merged;
```

Figure 23-4. *A process that merges two reservoirs into one. Note that by calling* updateReservoir *multiple times, we can merge any number of reservoirs.*

23.3 GRID-BASED RESERVOIRS

Our main idea is to split light sampling into two steps. First, we create a pool of samples likely to contribute to a certain area (i.e., the region in a grid cell). In this step, we resample the initial samples according to the grid cell position by estimating light intensity at its area. Here, we can also cull lights that have no contribution to the grid cell. In the second step, we resample according to the BRDF contribution at the shaded point. Both steps produce high-quality samples using RIS to quickly draw samples using a cheap source PDF, but in the second step we sample from the smaller pool. This chains resampling steps, evaluating a more expensive target PDF in the second step but working from the smaller sample count provided by the first step.

23.3.1 SELECTING LIGHT SAMPLES FOR THE GRID

The first step draws samples uniformly from the pool of all lights, a constant time operation, and then resamples according to the target probability based on the light intensity at the grid cell's position (attenuated due to squared distance). It is important to clamp the light distance to the cell borders, otherwise lights inside the cell would be incorrectly prioritized. Note that any method for drawing samples can be used to select initial candidates, e.g., the efficient *alias method* discussed in Chapter 21. Our method works for any light type, as we only evaluate the light intensity for the first step and the BRDF in the second step.

23.3.2 SAMPLING THE LIGHT FOR SHADING

Now, the selected samples in each grid cell can be seen as a pool of single-light reservoirs created by RIS, as shown in Figure 23-5. We can merge these into a single reservoir for shading. Looping over all reservoirs

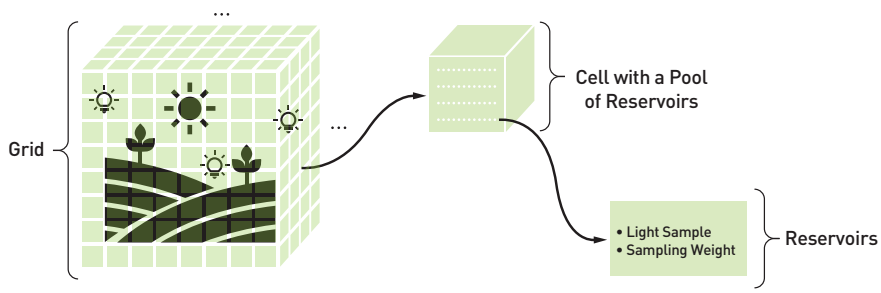

Figure 23-5. *A depiction of grid cells, each storing reservoirs of light samples.*

exhaustively, however, results in poor performance when the number of reservoirs per grid cell is large (we use 512 as a default), and we still need to resample these lights again to account for the BRDF.

To solve both issues, we use the resampling procedure from Figure 23-3 again to merge reservoirs and resample according to the BRDF at the same time. Here, the BRDF is also multiplied by light intensity at the shaded point when calculating the target PDF for RIS. Reservoir merging selects one best light from many reservoirs, whereas resampling implements this merging efficiently, without the need to iterate over all input reservoirs. This time, the resampling target probability is set to the partial BRDF for a given shaded point. The source pool is the set of reservoirs we want to merge. The source PDF is more complex and is described in Section 23.4.2.

Note that the first and second steps are decoupled and do not depend on the screen resolution, world complexity, or number of lights. We can draw any number of lights from the pool in the grid cell once it is built, and we can build pools of lights corresponding to any area in the scene, as we will discuss in Section 23.4.1.

This concludes the high-level description of our algorithm; we cover important implementation details in the next section.

23.4 IMPLEMENTATION

23.4.1 CONSTRUCTION OF THE GRID

To construct our grid, we must first decide on the grid dimensions and how many light samples to store per cell. There will be multiple reservoirs in each grid cell, which we call *light slots*, each storing one light sample. Parameters can be established by experimentation, potentially varying with scene light count, scene extent, visibility range, and desired performance. Our default allocates a grid with 16^3 cells, giving 4096 cells in total. We also default to 512 light slots per grid cell, but depending on the scene lighting complexity, values from 64 to 1024 give good results. As virtual worlds are often flat, allocating a grid with fewer cells along the vertical axis is often advisable.

Using these parameters, we can allocate a GPU buffer of the necessary size. Each light is represented by the `Reservoir` structure, which can use 16-bit precision numbers for all fields except `totalWeight` to reduce memory requirements. As an optimization, we need not explicitly store the number of

Figure 23-6. *Left: our Burger Restaurant scene. Right: an illustration of the world-space grid cell placement. Individual cells are highlighted by random colors.*

seen candidates M in the reservoir, as it is only used to normalize `totalWeight`, which can be precalculated. This also prevents M from growing to infinity as the number of seen samples grows over time. With these optimizations, the memory requirements for the default setting are only 16 MB.

POSITIONING THE GRID

Deciding how to place our grid in the scene depends largely on the rendered content. The simplest approach stretches the grid so that it spans all geometry (see, e.g., the Figure 23-6). This works well for relatively small scenes of known size, but many games feature open-ended worlds with dynamically loaded content, where scene size constantly changes.

For these cases, we recommend one of two approaches. A scrolling clipmap can be used, which ensures that individual grid cells seemingly stay in place and their positions are given by the window into a clipmap centered around the camera. In this case, the range of the grid (or size of the cell) has to be determined, which prevents our light sampling from being used outside of this range.

Another possible approach is to implement a sparse grid, where the positions of individual cells are determined using a hash map. In this case, we map each point in world space to a grid cell in the hypothetical infinite uniform grid as:

```
1     return int3(worldCoords / gridCellSize);
```

Once we try to sample from a grid cell at given coordinates, we store these coordinates into the map, at a position determined by the hash. Note that conflicts must be resolved at this point. When we construct the grid, we first

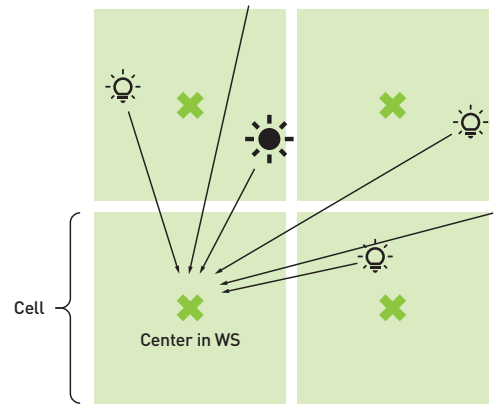

Figure 23-7. *Construction of the grid. RIS selects a number of candidates and resamples them according to the light intensity at the cell center.*

read the map to determine which cells in the world space are used, map them back to world space, and create reservoirs only for them.

BUILDING CELL RESERVOIRS

To fill the grid with light samples, we apply the RIS algorithm (see Figure 23-3) for each entry (light slot) in a grid. Remember that each grid cell contains multiple light slots, i.e., multiple light samples that were created by running the RIS algorithm.

Because selecting a per-slot light sample is done independently of other slots, even from the same cell, sampling can be parallelized, making grid construction very fast. This procedure first maps thread ID to the corresponding grid cell (see Figure 23-7), which is in turn used to map the cell position to world space:

```
// Calculate the grid cell coordinates.
int gridCellLinearIndex = threadID / gGridLightsPerCell;
int3 cellCenterCoords = linearToCoords3D(gridCellLinearIndex);

// Calculate the grid cell center in world coordinates.
float3 gridCellCenter = mapGridToWorld(cellCenterCoords);

// Select a light to be stored in evaluated light slot.
Reservoir lightSlotReservoir = sampleLightRIS(gridCellCenter);
```

By default, the number of initial RIS candidates (M) we use is eight. Increasing it quickly encounters the law of diminishing returns. As mentioned in

Section 23.3.1, the target PDF in `sampleLightUsingRIS` only depends on the intensity of the light at the grid center, as follows. Note the clamping that ensures that lights inside the cell have the same probability:

```
float3 lightVector = candidate.position - gridCellCenter;
float lightDistanceSquared = max(gMinDistanceSquared, dot(lightVector,
    lightVector));
float sourcePdf = <Source PDF as described, e.g., uniform sampling>;
float targetPdf = sample.intensity / lightDistanceSquared;
```

This code assumes a typical distance-based attenuation function for all lights, i.e., attenuating intensity with the distance squared, but any type of light can be supported. Most notably, directional lighting from the sun has the same intensity everywhere in the scene.

To ensure that each reservoir's running count *M* of the light candidates that it incorporates does not grow to infinity, we normalize it before storing the reservoir in the grid. This also enables removing *M* from the reservoir, as it is always one at this point:

```
lightSlotReservoir.totalWeight = lightSlotReservoir.totalWeight /
    lightSlotReservoir.M;
lightSlotReservoir.M = 1;
```

Note that using this process, a light can end up in *any* grid cell where it has a nonzero probability of contribution. Thus, our grid is not a spatial subdivision structure, but rather a data structure to store samples and their probabilities.

Finally, an important optimization is to only update grid cells that are used. Cells covering empty space will never be used for shading, so we can skip filling their light slots. We implement a cache where each cell stores a frame number for when it was last accessed. This information is cached by the sampling routine. During per-frame construction, we first check whether each cell has been recently accessed (e.g., in the last eight frames) and only update cells in active use.

TEMPORAL REUSE

The construction process just described is repeated each frame, rebuilding the grid from scratch. However, we can use reservoirs of lights from previous frames to continuously improve the grid in the most recent frame using the reservoir merging process described in Section 23.3.2. This is an important technique also used in the original ReSTIR implementation [3] that achieves much more stable results.

To handle temporal reuse, we also retain grids from previous frames (eight by default). During grid construction, we compute each reservoir as previously described, but before storing it into the grid, we merge this new reservoir (see Figure 23-4) with the reservoirs from prior frames (corresponding to the same light slot):

```
// Merge new reservoir with reservoirs from previous frames.
for (int i = 0; i < GRIDS_HISTORY_LENGTH; i++) {
    lightSlotReservoir = mergeReservoirs(lightSlotReservoir,
        gLightGridHistory[i][lightSlotIndex]);

    // Divide wSum by M after each combining "round."
    lightSlotReservoir.totalWeight /= lightSlotReservoir.M;
    lightSlotReservoir.M = 1;
}
```

Because good light samples have higher weight, it is likely that we keep reusing them until even better samples are found. This helps to continuously improve our samples.

DYNAMIC LIGHTS

Our method supports dynamic lights out of the box. We have intentionally used a light ID in the reservoir structure to reference sampled lights. This requires indirection to access light properties, but also ensures that every time we use a light, its current properties are used for shading. However, a problem can occur with temporal reuse, which does not modify a light's weight even if it changes. This can introduce excessive noise if the lights change significantly. As a workaround, we can prevent dynamic lights from participating in temporal reuse, so they will be replaced by a new sample. Alternatively, a more expensive reservoir merge can re-weigh dynamic lights during grid construction.

23.4.2 SAMPLING FROM THE GRID

When sampling lights, we start by finding the grid cell corresponding to the shaded point (see Figure 23-8). To reduce artifacts from nearest neighbor lookups (i.e., visible cell boundaries), we first randomize the lookup with an offset proportional to the cell size. This essentially performs stochastic trilinear filtering, sampling neighboring cells proportionally to their distance from the lookup:

```
// Jitter hit point position within the size of grid cell.
float3 gridLoadPosition = pointPosition+(float3(rand(), rand(), rand()) *
     2.0f - 1.0f) * gHalfGridCellSize;

```

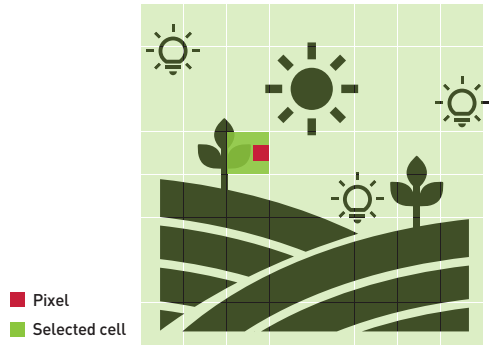

Figure 23-8. *Sampling from the grid. First, a cell in the neighborhood of the pixel is selected, then RIS is used to resample lights stored in the cell at the pixel. Light sources indicated in the image are an example of lights stored in sampled grid cells.*

```
// Figure out which grid cell to sample from (gridCellStartIndex) based on
    the jittered hit position.
int3 gridCellCoords = mapWorldToGrid(gridLoadPosition);
uint gridCellIndex = coords3DToLinear(gridCellCoords);
uint gridCellStartIndex = gridCellIndex * gGridLightsPerCell;
```

This gives us `gridCellStartIndex`, a position in the grid buffer where the grid cell's pool of samples starts. Next, as discussed in Section 23.3.2, we apply RIS again to merge reservoirs in the pool into a final light sample for shading, and also we resample according to the target PDF (which is based on the BRDF at the shaded point). Here, RIS draws source samples directly from the reservoirs of the selected grid cell:

```
float sourcePdf = candidate.sampleTargetPdf / reservoirAverageWeight;
float targetPdf = PartialBrdf(candidate, shadedPoint);
```

We calculate the source probability as the target PDF of candidates from the first RIS pass (during grid construction) divided by the average weight of all reservoirs (light slots) in the grid cell. This is an iterative application of RIS, similar to the approach described in the original ReSTIR article [3]. This average may be used multiple times, so we precalculate it for each cell in a separate pass. The function `PartialBrdf` depends on renderer-specific BRDFs and light parameters and can be a full BRDF if the cost is reasonable.

Finally, a pixel can be shaded using the light sample and the RIS weight:

```
Reservoir risReservoir = sampleLightUsingRIS(shadedPoint);
float lightSampleWeight = (risReservoir.totalWeight / M) / risReservoir.
    sampleTargetPdf;
LightSample light = loadLight(risReservoir.lightSample);
return light.intensity * Brdf(light, shadedPoint) * lightSampleWeight;
```

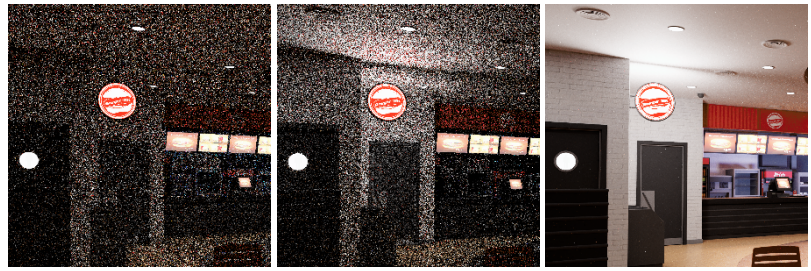

Figure 23-9. *A single frame rendered using naive uniform sampling (left) and ReGIR (center) and the accumulated result (right) of the Burger Restaurant scene. Note the significantly better sampling around the light source in top left corner using our method.*

23.5 RESULTS

We implemented and evaluated ReGIR (our method) in the Falcor rendering framework [2]. Compared to naive uniform sampling and basic RIS, we achieved superior results (see Figures 23-9 and 23-10) with only a small per-frame performance cost of about 1.5 ms using our default settings. Performance was measured with a RTX 3090 GPU and 1920 $\times$ 1080 resolution

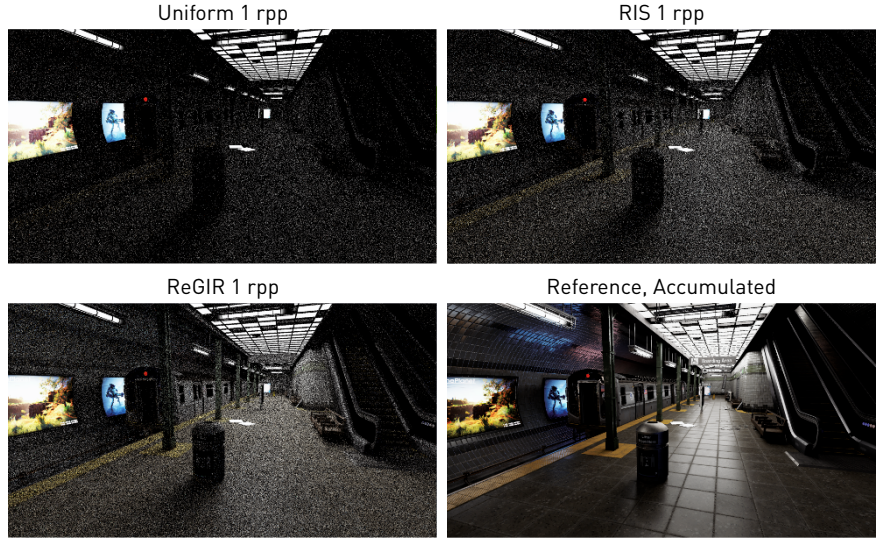

Figure 23-10. *Comparison of naive uniform sampling (top left) to only using RIS with 16 candidates (top right), our ReGIR method (bottom left), and a ground truth (bottom right). We use one ray per pixel (rpp).*

and is similar across most tested scenes. Of this cost, about 0.3 ms is spent on constructing the grid. The remaining cost comes from selecting lights from the grid for shading; this cost depends on the resolution and number of secondary rays accessing the grid. Our performance depends mostly on how many light samples we store in the grid and the number of nonempty cells due to the scene complexity.

Visual quality depends on the number of light samples that we can afford to store per grid cell, and on the size of the cells in world space. Using smaller cells can improve visual quality, but can also limit the range of the grid.

We show results of our method applied to primary rays; however, its expected use case is shading secondary rays and arbitrary points in the scene, whereas screen-space ReSTIR typically gives better quality on primary hits.

The slight bias of our method can be attributed to the discrete nature of the grid and the limited number of samples stored in each grid cell. Temporal reuse can also contribute to the bias. In real-time applications, this should not pose significant issues as we believe high performance is preferable, and the presence of a denoiser should smooth out any remaining artifacts.

23.6 CONCLUSIONS

In this chapter, we presented our new ReGIR algorithm for many light sampling, which can be used in real-time ray tracing for both primary and secondary rays. Because it uses a simple data structure, it is relatively easy to implement in game engines and provides high performance, although other, more costly methods can give superior results in terms of lower noise. However, when using a specialized denoiser, higher performance can be a more important benefit, assuming input noise is low enough to produce sharp and stable results after denoising.

Because our world-space method uses a larger granularity than screen-space ReSTIR, it is not as well suited for shading primary ray hits, except in scenes with only a few lights. However, it enables shading secondary hits, and combining these two methods might be a preferred way of using ReSTIR. ReGIR can also be combined with methods for calculating global lighting such as dynamic diffuse global illumination (DDGI) [4] or with a path tracer.

REFERENCES

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

[2] Benty, N., Yao, K.-H., Clarberg, P., Chen, L., Kallweit, S., Foley, T., Oakes, M., Lavelle, C., and Wyman, C. The Falcor real-time rendering framework. https://github.com/NVIDIAGameWorks/Falcor, 2020. Accessed August 2020.

[3] Bitterli, B., Wyman, C., Pharr, M., Shirley, P., Lefohn, A., and Jarosz, W. Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 39(4):148:1–148:17, July 2020. DOI: 10/gg8xc7.

[4] Majercik, Z., Guertin, J.-P., Nowrouzezahrai, D., and McGuire, M. Dynamic diffuse global illumination with ray-traced irradiance fields. *Journal of Computer Graphics Techniques (JCGT)*, 8(2):1–30, 2019. http://jcgt.org/published/0008/02/01/.

[5] Talbot, J., Cline, D., and Egbert, P. Importance resampling for global illumination. In *Eurographics Symposium on Rendering (2005)*, pages 139–146, 2005. DOI: 10.2312/EGWR/EGSR05/139-146.