Interactive Stable Ray Tracing

Alessandro Dal Corso
NVIDIA
Technical University of Denmark
alcor@dtu.dk

Marco Salvi
NVIDIA

Craig Kolb
NVIDIA

Jeppe Revall Frisvad
Technical University of Denmark

Aaron Lefohn
NVIDIA

David Luebke
NVIDIA

ABSTRACT

Interactive ray tracing applications running on commodity hardware can suffer from objectionable temporal artifacts due to a low sample count. We introduce stable ray tracing, a technique that improves temporal stability without the over-blurring and ghosting artifacts typical of temporal post-processing filters. Our technique is based on sample reprojection and explicit hole filling, rather than relying on hole-filling heuristics that can compromise image quality. We make reprojection practical in an interactive ray tracing context through the use of a super-resolution bitmask to estimate screen space sample density. We show significantly improved temporal stability as compared with supersampling and an existing reprojection techniques. We also investigate the performance and image quality differences between our technique and temporal antialiasing, which typically incurs a significant amount of blur. Finally, we demonstrate the benefits of stable ray tracing by combining it with progressive path tracing of indirect illumination.

CCS CONCEPTS

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

KEYWORDS

Reprojection, dynamic scene, caching, temporal stability, GPU

ACM Reference format:
Alessandro Dal Corso, Marco Salvi, Craig Kolb, Jeppe Revall Frisvad, Aaron Lefohn, and David Luebke. 2017. Interactive Stable Ray Tracing. In *Proceedings of HPG '17, Los Angeles, CA, USA, July 28-30, 2017,* 10 pages.
DOI: 10.1145/3105762.3105769

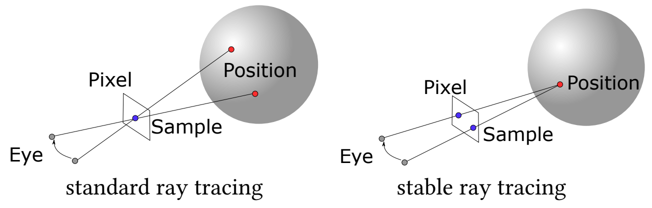


Figure 1: In standard ray tracing, sampled screen space locations are kept fixed and shading locations vary. In stable ray tracing, shading locations are kept static while screen space locations vary.

1 INTRODUCTION

A rendered image will contain aliasing artifacts in regions where the underlying signal carries higher frequency content than the local sampling rate can capture. For example, light reflected from a highly specular surface can lead to aliasing if not sampled at sufficiently high rate. In addition, such aliasing artifacts will be perceived as particularly objectionable if high-frequency details are inconsistently sampled, causing sample values to change rapidly in time. To eliminate these artifacts, the underlying signal should ideally be bandlimited to remove frequencies beyond the local Nyquist limit. In general, however, robustly bandlimiting reflectance functions, visibility, and programmable shaders are open problems.

Stable shading is one strategy that can successfully mitigate aliasing artifacts in practice. In stable shading, shading calculations are performed in an object-local parametrization space, such as at the vertices of an underlying mesh, and the resulting values are interpolated across image pixels. These same object-local vertices are typically shaded again in subsequent frames, improving temporal stability in the presence of aliasing. Gouraud shading [1971] and the REYES rendering algorithm [1987], for example, use this approach to improve temporal image quality. However, these stable shading techniques do not work well with approaches such as ray tracing, wherein shading locations are determined independently of and without regard to any underlying local surface parametrization.

Stable ray tracing is a general technique that draws inspiration from previous stable shading approaches to improve the visual quality and/or reduce the computational cost of generating a sequence of images using ray tracing. Rather than using independent rays to sample the screen, shading locations from previous frames are re-used when possible, as shown in Figure 1. The fact that the points being shaded are temporally coherent results in fewer objectionable artifacts, even though the resulting images are still aliased. Furthermore, intermediate shading values can be cached along with the shading location, providing an additional performance benefit.

Our stable ray tracing is based on sample reprojection [Adelson and Hodges 1995; Badt 1988; Martin et al. 2002]. The main challenges in reprojection are verifying visibility of reprojected samples and avoiding large holes in the resulting screen space sampling pattern. We deal with the first issue by tracing visibility rays from the camera to the reprojected samples. For the second issue, we generate new samples on demand, where the demand is determined

HPG '17, Los Angeles, CA, USA
© 2017 ACM. 978-1-4503-5101-0/17/07…$15.00
DOI: 10.1145/3105762.3105769

using screen space sample density estimation. We perform this density estimation efficiently using a super-resolution bitmask that maps subpixel sample locations. This bitmask is also useful for removing samples to keep a uniform sample distribution. As an example application of stable ray tracing, we use amortized sampling to add progressively path traced indirect illumination to an image. We demonstrate how our stable ray tracing significantly improves temporal stability as compared with supersampling and as compared with an existing reprojection technique [Martin et al. 2002]. In addition, we use an image sharpness metric to verify that our technique avoids the blur of post-process filtering techniques.

2 RELATED WORK

The use of sample reprojection to exploit the temporal coherence of ray traced frames was first suggested by Badt [Badt 1988]. His technique is limited to viewpoint changes only, but he identifies the key issues of bad pixels and missed pixels. Bad pixels occur in regions where the colors of reprojected samples are no longer valid. Missed pixels are pixels that are not hit by reprojected samples. Badt suggests the interesting notion of a "recast mat", a one-bit-per-pixel mask pointing out the pixels for which we need new samples. We reverse this concept and use a super-resolution bitmask pointing out the subpixels that were hit by a reprojected sample.

Chapman et al. [1991] map out the spatio-temporal coherence of a predefined animation sequence by tracking sample trajectories across scene geometry. This is similar to sample reprojection and also works for moving objects. A reprojection based technique exploiting coherence between frames in a predefined animation sequence is also available for variance reduction in Monte Carlo ray tracing [Zhou and Chen 2015]. Gröller and Purgathofer [1991] present a spatial data structure for techniques like these that assume a predefined animation sequence. A more progressive approach is however required in interactive ray tracing, where the future scene dynamics are unknown. Murakami and Hirota [1992] present such an incremental approach, but only for a fixed viewpoint. They connect ray paths with objects using a hash index so that it is only necessary to recompute paths that interact with dynamic objects. We also connect samples to objects using an index.

Adelson and Hodges [1995] present a fully general reprojection technique for ray tracing with a screen space data structure containing one sample per pixel. We enhance this data structure by enabling a nonintegral number of samples per pixel. Adelson and Hodges [1995] also provide a careful description of the verification phase including the need for shadow and visibility rays to check for occlusion. We adopt their verification phase and make it practical for an interactive ray tracer running on graphics hardware.

The render cache concept [Walter et al. 2002, 1999; Zhu et al. 2005] achieves interactive frame rates through reprojection with different heuristics for handling bad and missed pixels. While the heuristics significantly improve performance, they also lead to objectionable visual artifacts.

Although reprojection started out as a way of exploiting temporal coherence to save computations, Martin et al. [2002] recognize it as an important technique for avoiding temporal aliasing. They find that reprojection achieves temporal stability similar to supersampling at a significantly lower computational cost. Their system only accounts for viewpoint changes and they apply temporal filtering using a box filter spanning three frames. Apart from this, their technique seems quite similar to that of Adelson and Hodges [1995]. Martin et al. [2002] also use one sample per pixel and pick the closest sample when multiple samples land in one pixel. This one-sample-per-pixel policy easily leads to scintillation artifacts due to insertion or removal of samples as objects rotate or move relative to the camera. Missed pixels and multiple samples in one pixel occur frequently when samples move across pixel boundaries (especially in perspective view) even if the local sample density is not changing much. We successfully mitigate this issue by estimating sample density in a 2-by-2 pixels area centered in every pixel. Our super-resolution bitmask strategy enables us to perform this density estimation efficiently.

In rasterization, the use of reprojection seems to be introduced in the context of warping one rendered image to the next [Chen and Williams 1993; Mark et al. 1997]. Rasterization-based techniques like the edge and point image [Bala et al. 2003; Velázquez-Armendáriz et al. 2006] achieve good results by adding edge information to the render cache information. However, this requires precomputation of an edge-based data structure [2003] or an additional edge rendering of the image [2006]. This becomes expensive in geometry-rich scenes where several edges may land in a pixel.

Inspired by the offline techniques [Adelson and Hodges 1995; Walter et al. 1999], reprojection finds an efficient implementation in a rasterization context with the reverse reprojection cache [Nehab et al. 2007] (also discovered by Scherzer et al. [2007] in a shadow mapping context and optimized by Sitthi-amorn et al. [2008a,b]). We keep forward reprojection, as this is better suited for ray tracing. As an add-on, these techniques [Nehab et al. 2007; Scherzer et al. 2007] introduce amortized sampling where pixel values are progressively updated over time. We use such amortized sampling for progressive sampling of indirect illumination.

Reprojection has also been used together with Monte Carlo ray tracing techniques like bidirectional path tracing and photon mapping [Havran et al. 2003; Tawara et al. 2004]. These techniques rely on stored sample points in any case, so no additional data structure is needed for the reprojection. In our case, we add a screen space data structure to support stable ray tracing. Our approach is thus well-suited for unidirectional Monte Carlo techniques.

In rasterization, Herzog et al. [2010] find that temporal finite differences are useful for amortized upsampling of images rendered with real-time global illumination techniques. They investigate screen-space ambient occlusion and indirect illumination from virtual point lights. In addition to better performance, they also find that their reprojection cache improves temporal stability.

On the side of temporal stability, recently introduced postprocessing filters like temporal supersampling [Karis 2014; Patney et al. 2016] efficiently hide temporal aliasing at the cost of introducing blur in the final image. Reprojection helps avoid excessive blurring and is effective in combination with sampling and filtering techniques from antialiasing [Jimenez et al. 2012] and from denoising [Iglesias-Guitian et al. 2016]. We set out to confirm that forward reprojection also has this ability to reduce temporal aliasing while preserving image sharpness. In addition, we exemplify the benefits of having stable samples in interactive ray tracing.

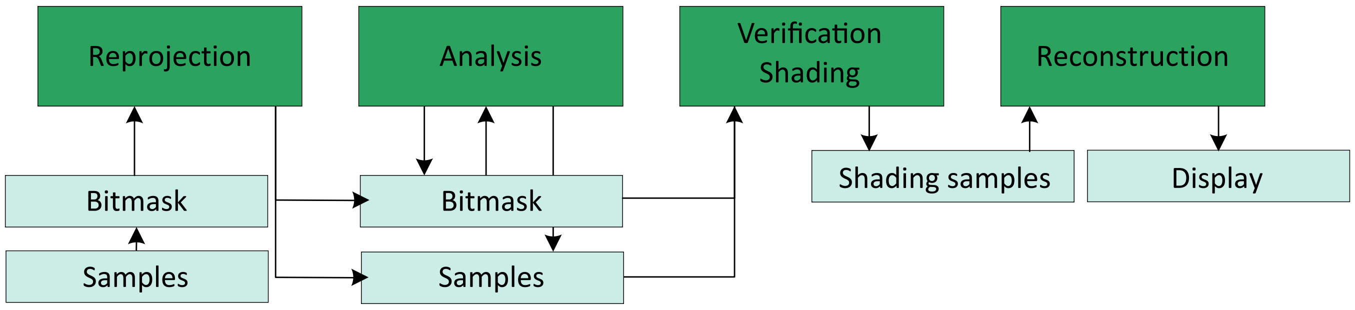


Figure 2: Main building blocks of our algorithm. Data are in light green, compute phases are in dark green.

3 OVERVIEW

In its most straight-forward implementation, stable ray tracing consists of four phases. *Reprojection* projects cached shading locations from the previous frame into screen space of the current frame, accounting for camera and object motion/deformation, to create a set of screen space sample locations. *Verification* constructs and traces primary visibility rays through the screen space sample locations to determine which of the reprojected shading positions are visible from the camera. Visible locations are then shaded, optionally caching intermediate results of the shading computation for later reuse. *Hole filling* generates screen space samples in regions where the density of visible reprojected points is low, and traces, shades, and caches hitpoint/shading information. Finally, *reconstruction* generates the final image for the current frame from the set of shaded samples.

The basic version of stable ray tracing improves temporal stability through the reuse of shading points across frames. However, there are a number of practical challenges to achieving interactive performance. In this section, we discuss these issues and associated tradeoffs, and briefly describe the choices we made in our system.

3.1 Sampling Rate and Uniformity

Sampling rate is the primary means of trading image quality for performance. Unlike conventional ray tracing, wherein screen sample locations are essentially independent of objects in the scene, in stable ray tracing screen space sampling density can be highly non-uniform due to the effects of camera and object movement on reprojected samples. Reprojection can lead to oversampling due to many points being reprojected to the same region of the screen, for example when an object moves away from the camera, or the camera zooms out. In such cases, maintaining performance requires that we ensure oversampling is kept to a minimum. Conversely, reprojection can also lead to undersampling due to disocclusions, or when sample density decreases due to a surface moving closer to the camera. In such cases, maintaining image quality requires that we ensure that enough samples are used. Highly non-uniform sampling can also lead to issues with resource contention (for example, multiple threads attempting to write to same cache location during reprojection) and load balancing. In addition, nonuniform sampling can produce artifacts when the sampling rate is very low compared to the reconstruction rate, as discussed in Section 7.

In order to ensure appropriate sampling rate and uniformity, our implementation adds an *analysis phase* prior to verification. The analysis phase efficiently estimates local sampling density and adds or removes samples to ensure the sampling rate falls within a specified range. As described in Section 4.2, the analysis phase makes use of a bitmask that encodes a quantized representation of the sampling pattern in each pixel, which allows us to estimate sampling density without having to read or recompute exact screen space locations for each sample.

3.2 Caching

Key to the efficiency and effectiveness of our implementation is the sample cache, which allows temporal re-use of shading locations and intermediate values. However, stable ray tracing's computational and memory overhead is proportional to the number of entries that are reprojected and potentially verified and shaded. As such, a cache eviction policy is needed that allows trading performance and memory use for temporal stability.

The simplest policy would be to evict points that are occluded or otherwise not used in the current frame. However, stability in the face of high-frequency visibility changes can be improved if occluded points remain in cache long enough to be re-used when they become visible again. As a result, there is a tradeoff between the space and reprojection cost of keeping occluded points in the cache and the temporal stability improvements to which such points may contribute in the future.

In addition to storing in the cache sufficient information to reconstruct world space position, we can also use the cache to avoid recomputation of expensive intermediate values required during shading (e.g., visibility or normals). Taken together, these values can cause each cache entry to be rather large. As such, minimizing overall size is important to performance, as is minimizing cache reads due to memory bandwidth constraints.

We use a two-phase cache eviction scheme that strives to strike a balance between overall performance and temporal stability. The first set of evictions occur in the reprojection phase (Section 4.1) and the second in the analysis phase (Section 4.2).

3.3 Ray Tracing

The basic stable ray tracing algorithm has two distinct ray tracing phases: verification and hole filling. The number of holes to be filled is typically small compared to the number of verification rays, and as a result the overhead associated with launching a separate hole-filling ray tracing pass can be non-trivial. As such, performance

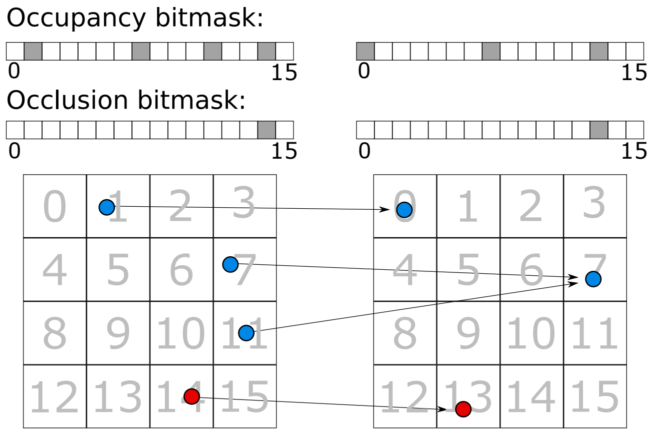


Figure 3: Reprojection phase for an M=4 set of subpixels from one frame, left, to the next, right, and evolution of the bitmasks. Both occluded and unoccluded samples are recorded in the occupancy bitmask. Occluded samples (red) also have a bit set in the occlusion bitmask. In the case of a collision between two samples (subpixel 7, right), the first unoccluded sample written to the subpixel is kept.

```
input : pixelDestination and subpixelDestination for a sample
        and associated data that isOccluded or not.
subpixel ⟵ flatten (subpixelDestination);
bitOccupancy ⟵ 1 ≪ subpixel;
bitOcclusion ⟵ 1 ≪ (subpixel + M·M);
bitMask ⟵ bitOccupancy ∨ (isOccluded? bitOcclusion: 0);
originalBitmask ⟵ AtomicOr (pixelDestination, bitMask);
originalIsOccluded ⟵ (bitOcclusion ∧ originalBitmask) ==
    bitOcclusion;
replace ⟵ not isOccluded ∧ originalIsOccluded;
if not (isOccluded ∧ originalIsOccluded) then
    AtomicAnd (pixelDestination, ¬bitOcclusion)
end
originalExists ⟵ (bitOccupancy ∧ originalBitmask) ==
    bitOccupancy;
if replace ∨ not originalExists then
    writeData( pixelDestination,data);
end
```

Algorithm 1: Pseudocode for sample reprojection storage.

could benefit if it were possible to combine the two ray tracing passes into one.

In our implementation, we fill verification-failure holes by using the occluding hitpoints discovered in the verification phase. This optimization improves performance over the naive implementation, at the cost of some sampling bias and an increase in sampling rate variance. However, the instability added is typically spatially incoherent and persists for a single frame, and as such is not usually objectionable.

4 METHOD

In this section, we discuss the details of our implementation, the design decisions we faced, and the choices we made. Our implementation is illustrated in Figure 2.

We store samples in two screen space data buffers, which serve as caches for the previous and current frame. At the beginning of each frame, samples are reprojected from the previous buffer to the current to account for object and camera motion. We analyze the outcome of the reprojection process and adaptively add or remove samples in the reprojection buffer in order to achieve a uniform sample distribution. The location samples are then verified, and finally shaded. The resulting color information is stored in a shading buffer, which is used by the reconstruction phase to resolve color.

4.1 Reprojection

Stable ray tracing requires that cached samples are updated to reflect scene dynamics such as camera motion and object motion and deformation. The data to be stored per sample in the reprojection buffers should thus be chosen according to the scene dynamics that one would like to support. We store a 3D position in object space coordinates and a transform ID to support affine transformations. More data would likely be required to support arbitrary object deformation. The ID we store is used to access an object-to-world transformation matrix for the current frame. This matrix is in turn used to transform the sample position to world space. We then project the world space position onto the screen using the current camera transformation, and we clip away samples that fall outside the screen area.

During reprojection, we take steps to ensure that not too many samples reproject to the same screen location in order to reduce resource contention, improve load balancing, and manage size of the cache. We also strive to preferentially keep samples that are visible over those that are occluded.

To do so, we divide each pixel in the reprojection buffer into $M \times M$ subpixels, as illustrated in Figure 3. We maintain a corresponding occupancy bitmask representing the occupancy state of each subpixel, which is cleared at the start of each frame. The occupancy bitmasks are also used during the analysis phase to determine approximate sample location and local sample density. We similarly maintain with each pixel an $M \times M$ bitmask that indicates if the sample in each subpixel is occluded; values in this occlusion bitmask are written during the verification phase. Storing these bitmasks separately from the cache values themselves allows us to reduce bandwidth required by the reprojection phase.

When a source sample reprojects into a given destination subpixel, we check the destination subpixel's corresponding occupancy bit in the bitmask. If the destination subpixel occupancy bit is zero, the sample is written to the destination location, the destination occupancy bit is set to one, and the destination subpixel occlusion bit is copied from the source bitmask. If the destination subpixel occupancy bit is one, we examine the destination subpixel occlusion bit. If the destination subpixel occlusion bit is one and the source occlusion bit is zero, the source sample is written to the destination, and the destination occlusion bit set to zero. Otherwise the source sample is not written to the destination buffer, effectively evicting

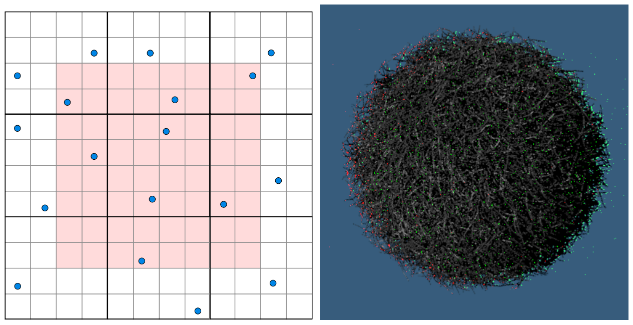


Figure 4: Left: the spatial extent (light red) of our local density analysis. Right: Update pattern that arises from our analysis scheme. Example: hairball sequence with a right-rotating camera and $d_{\text{target}} = d_{\text{tolerance}} = 1$. Green pixels indicate areas where new samples are added, while red samples indicate where samples are evicted.

it from the cache. A pseudocode outline of our eviction scheme is in Algorithm 1.

Data races due to competing threads working on the same sample can be avoided by atomically updating the per-sample data, potentially causing a large performance impact. We note instead that as we only perform atomic updates of the bitmasks a data race can only occur when a first occluded sample lands on a sample and second unoccluded one tries to overwrite it. In this rare case, we would simply store the occluded sample over the unoccluded, leading to reduced temporal stability. In practice we found these events to be rare and to have small impact on the final image quality.

Our sample rejection policy ensures that we cache at most $M \times M$ samples in any pixel, enforcing an upper bound on storage and subsequent processing costs, while maintaining a good screen-space distribution of samples, unlike, for example, simply keeping the first $M \times M$ samples that reproject into a given pixel would. The mechanism also ensures that unoccluded samples are preferentially cached over occluded samples.

4.2 Sample Analysis

In regions that are oversampled, analysis chooses which samples to remove, and adds new samples in undersampled regions to meet the desired sampling rate.

To help ensure a good spatial distribution of samples, we divide each pixel in a number of strata (in our implementation, 4). For each stratum, we count the number of samples. To remove samples, we choose from the substratum with the most number samples, selecting randomly in the case of a tie. Similarly, we progressively add samples to the substratum with the fewest samples. This process allows us to stratify the samples across the pixel. Within a substratum, new samples are placed in the center, with a small random offset in order to avoid correlation in the screen space location of the samples.

To minimize the overall performance impact of analysis, we use the occupancy and occlusion bitmasks to determine whether samples should be added or removed. To determine how many to add or remove, we analyze the local sample density $d = N/A$, where N is the number of unoccluded samples in an area of $A = 2 \times 2$ pixels around the current pixel. The user can then specify two parameters, d_{target} and $d_{\text{tolerance}}$. The algorithm will not add or remove samples if the density is within $[d_{\text{target}} - d_{\text{tolerance}}, d_{\text{target}} + d_{\text{tolerance}}]$. Otherwise, we add or remove enough unoccluded samples ΔN to bring the density within limits:

$$\Delta N = \begin{cases} \operatorname{sgn}(d_{\text{target}} - d) \left\lceil \left| d_{\text{target}} - d \right| \right\rceil & \text{if } \left| d_{\text{target}} - d \right| \geq d_{\text{tolerance}} \\ 0 & \text{otherwise,} \end{cases} \tag{1}$$

where the sign of ΔN tells us whether we need to add or remove samples. Figure 4 illustrates a typical pattern of sample addition and removal for a dynamic scene.

It is necessary to modify the cache when we add a new sample, since in the next phase we need to distinguish between new and cached samples. To remove a sample, we simply set the corresponding occupancy bit to zero. For a new sample, we write $(\text{NaN}, \mathbf{p}^x, \mathbf{p}^y)$ instead of its object space position. The NaN marks the sample as new. Since we have to store the new sample in memory, we also store the chosen screen space coordinates for the sample $(\mathbf{p}^x, \mathbf{p}^y)$.

4.3 Verification and Shading

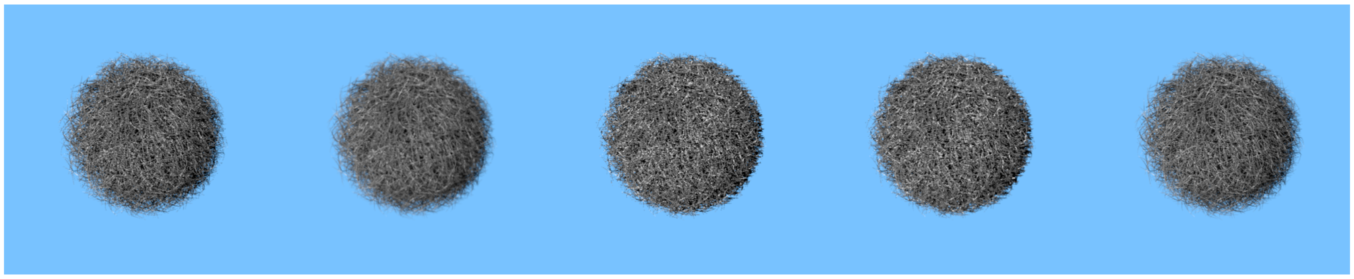
The verification phase processes the location samples to generate shading samples for the reconstruction phase. Our algorithm works on top of any ray tracing framework that provides programmable camera and closest hit stages. We define a standard ray as a tuple $\mathbf{r} = (\mathbf{o}, \vec{d}, t_{\min}, t_{\max})$, where the quantities represent origin, direction, and minimum and maximum intersection distances, respectively.

In this step, we distinguish between cached samples and newly generated samples with screen space coordinates $(\text{NaN}, \mathbf{p}^x, \mathbf{p}^y)$ in the cache. We trace these new samples with a closest hit ray, using the stored screen space position to generate a corresponding world space direction $\vec{d}$ according to our camera model. Given the camera position $\mathbf{c}$, our ray becomes $\mathbf{r} = (\mathbf{c}, \vec{d}, \epsilon, +\infty)$. Once the ray tracing operation terminates, we store the hitpoint object space position and transform ID in the reprojection cache, and the corresponding shade in the shading cache.

For existing samples with cached position $\mathbf{x}_{\text{object}}$, we first compute its corresponding world space position $\mathbf{x}_{\text{world}}$. Then, we cast a closest hit ray $\mathbf{r}_{\text{cached}} = (\mathbf{c}, (\mathbf{x}_{\text{world}} - \mathbf{c}) / \|\mathbf{x}_{\text{world}} - \mathbf{c}\|, \epsilon, \|\mathbf{x}_{\text{world}} - \mathbf{c}\| + \epsilon)$. When we hit the closest surface, we verify that the sample is still visible in the current frame. If the sample is still visible, the intersected t should match the cached $t = \|\mathbf{x}_{\text{world}} - \mathbf{c}\|$.

Occluded samples can cause numerical instability in the shading distribution, in particular around geometric edges. In our implementation, we normally mark such samples as occluded and keep them in the cache. However, if an occluded sample is the last one remaining in a pixel, we replace its hitpoint with the one from the occluding surface. This allows us to maintain a minimum sample density without requiring a new ray to be traced, as discussed in Section 3.3.

Once a sample is verified, or if it is new, we shade it according to our rendering algorithm, and store the results in the shading buffer, alongside its subpixel position.



Supersampling, 4 spp	Supersampling, 4 spp + temporal antialiasing	Stable ray tracing, 2 spp	Stable ray tracing, 2 spp + temporal integration	Supersampling, 32 spp
sharpness: 0.8142	sharpness: 0.6610	sharpness: 0.8056	sharpness: 0.7783	sharpness: 0.8054

Figure 5: Comparison of frames rendered for the hairball video. For each technique, we report the number of samples per pixel (spp) and the CPBD-based image sharpness. Stable ray tracing strikes a compromise between sharpness and temporal stability at the price of added spatial aliasing.

Technique	Reprojection	Analysis	Verification / Shading	Reconstruction	Total
Stable ray tracing, $d_{\text{target}} = 1$	1.05 ms	0.28 ms	18.91 ms	0.71 ms	**20.94 ms**
Stable ray tracing, $d_{\text{target}} = 2$	1.23 ms	0.38 ms	28.88 ms	0.82 ms	**31.31 ms**
Stable ray tracing, $d_{\text{target}} = 4$	1.73 ms	0.62 ms	47.48 ms	0.90 ms	**50.73 ms**
Supersampling, 1 spp	-	-	13.35 ms	0.21 ms	**13.56 ms**
Supersampling, 2 spp	-	-	20.94 ms	0.38 ms	**21.32 ms**
Supersampling, 3 spp	-	-	28.36 ms	0.54 ms	**28.90 ms**
Supersampling, 4 spp	-	-	35.86 ms	0.71 ms	**36.57 ms**
Supersampling, 5 spp	-	-	43.40 ms	0.88 ms	**44.28 ms**
Supersampling, 6 spp	-	-	50.91 ms	1.04 ms	**51.95 ms**

Table 1: Average time spent per frame in the hairball video for each phase of the different techniques. All results use GPU timers. The additional price for stable ray tracing is a slowdown of the overall rendering time between 1.4x and 1.5x. Temporal integration is performed on the resulting image, at an additional cost of 0.67 ms.

4.4 Reconstruction

Each color sample stored in the previous step carries an RGB color and subpixel position. We then filter our resulting color using a 3×3 truncated spatial Gaussian filter. Our algorithm does not guarantee uniform sampling rate, since it trades off a uniform rate for temporal stability. A nonuniform sampling density can lead to challenges in reconstruction, such as pixels with no samples. At low sampling densities, the use of this simple reconstruction filter can lead to blurring and apparent thickening of edges. We discuss the artifacts resulting from trading spatial uniformity for temporal coherence in Section 7.

After reconstruction, an additional post processing step may be performed. In Section 6, we discuss how our method fares with a temporal reconstruction scheme on top, namely temporal integration. When performing this additional step, we calculate and store motion vectors in the shading cache, picking the one with maximum length during reconstruction.

5 IMPLEMENTATION DETAILS

Our reprojection and analysis phases are implemented as OpenGL compute shaders. The reprojection shader transfers data between two identically deep screen sized buffers. The verification and shading step is implemented on the GPU in the camera program using the NVIDIA OptiX ray tracing engine [2010]. The programmable ray tracing pipeline of OptiX allows us to insert our cache management. The reconstruction and post processing were implemented as full screen passes in OpenGL shaders.

We compress our samples as 16-bytes elements of which 12 are reserved for 3 floating point elements defining position in object space. Due to OpenGL-OptiX interoperability limitations, we were not able to write the occlusion bit in the bitmask in the verification and shading phase directly. So we use one of the remaining 32 bits to store occlusion for the sample. Note that this does not change performance, since we have to fetch the sample anyways in the reprojection phase. The remaining 31 bits are reserved for a transform ID to allow affine transformations. The existence and occlusion statuses of the samples are stored in the bitmasks, for which we use $M = 4$. We use the two halves of a 32 bits unsigned integer to store both 16 bits bitmasks. The shading samples are stored as 8 bytes elements: 3 bytes for the tone-mapped color, 4 bytes for a motion vector (16 bits per component) and 1 byte for the subpixel position and flags (3+3 bits for position in a 8x8 grid, plus 1 bit for an existence flag).

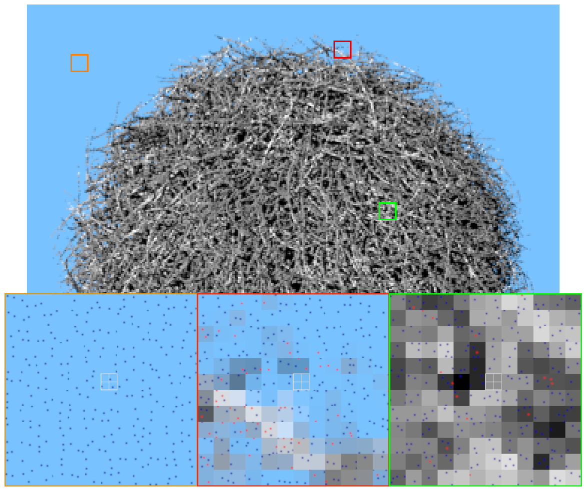


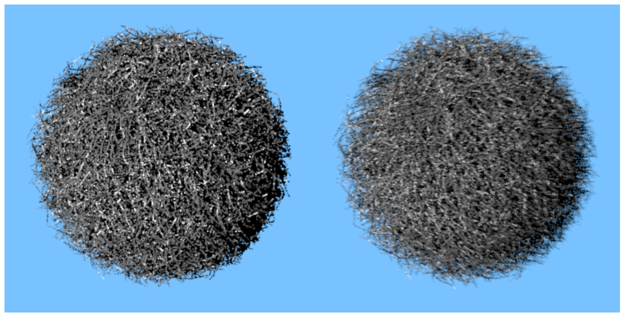
Figure 6: Three examples of sample distributions generated by our algorithm in the areas marked by the colored squares. Blue circles represent visible samples, red circles represent occluded samples.

6 RESULTS

Given the dynamic nature of our algorithm, we provide some of our results in a video (hairball.mp4) of a static hairball [McGuire 2011] captured with a moving camera. The hairball has a standard glossy material applied, and is illuminated by a single point light to which a shadow ray is traced per shading evaluation. The frames of the video were captured individually and then assembled to create a video of 60 frames per second. All our results were generated using an NVIDIA GeForce GTX 1080 graphics card. We report rendering times for a 1080x1080 image frame.

The hairball video compares stable ray tracing with supersampling of similar performance. In addition, to measure the impact of a recent temporal noise reduction scheme, we apply temporal integration with color clamping in the variant proposed by Patney et al. [2016]. For stable ray tracing, samples are not jittered and we choose an integration factor of $\alpha = 0.25$. For supersampling, we use $\alpha = 0.1$ and do full temporal antialiasing by including sample jittering in the temporal integration. The larger α used for stable ray tracing incurs a smaller amount of blur, which we can get away with because our input values are more stable. If we use $\alpha > 0.1$ with supersampling, the temporal antialiasing cannot hide the underlying temporal instability. A single frame of the hairball video is provided in Figure 5. Here, we compare image sharpness using a CPBD-based sharpness metric [Narvekar and Karam 2011]. The sharpness score measures the percentage of edges at which blur (probably) cannot be detected. The video shows a reference rendering, rendered as 32 samples per pixels.

Comparing supersampling and stable ray tracing, we first observe that while stable ray tracing does not completely remove temporal artifacts (in particular around the strands of the hairball), the final result perceptually improves in temporal stability. This is especially true at the beginning of the video, where the camera is only rotating. Sharpness of stable ray tracing and supersampling is similar to that of the reference, with supersampling being slightly



Stable ray tracing, 1 spp
sharpness: 0.8182

[Martin et al. 2002]
sharpness: 0.6957

Figure 7: Quality comparison with Martin et al. [Martin et al. 2002] for frame 526 of the martin_comparison.mp4 video. Their technique produces a blurrier result and is also more temporally unstable.

greater than reference. Once we apply temporal integration to both results, the situation reverses. Supersampling with temporal integration is more temporally stable (although some underlying noise is still present), but it is also significantly more blurry. Temporal integration applied to stable ray tracing reduces some of the higher frequency noise, but it also better preserves sharpness while retaining temporal stability.

Since our algorithm trades temporal stability for an irregular spatial sampling pattern, we want to validate the aliasing artifacts that are generated by the algorithm. An example of the kind of distribution of samples we achieve with our algorithm is shown in Figure 6, for three different areas of a single frame of the hairball video. We compare the quality for different target densities of our algorithm in Figure 8, for three different scenes (hairball, plane with text and ogre). The images were taken after 25 frames of an animated video, to allow stable ray tracing to set into a nonstandard sampling pattern. We provide closeups to better show the artifacts generated at a pixel level. For the lowest sample count (1 spp averages), we can see that stable ray tracing introduces artifacts. In the hairball frames, we can see that this manifests as thickened edges. In the plane with text frame, the artifacts manifest as broken edges and letters. In the ogre scene, they manifest as weirdly shaped specular highlights. For averages of 2 spp, the differences reduce and it almost disappears with averages of 4 spp.

We compare the performance of stable ray tracing against supersampling in Table 1. All results were obtained using OpenGL GPU timers, averaging the milliseconds spent in each phase over the whole sequence in the hairball video. From the totals in the table, we can see that stable ray tracing generally performs 1.4 to 1.5 times slower than the equivalent supersampling. This is similar to the performance cost of a factor of around 1.35 reported by Martin et al. [2002]. The overhead of reprojection and analysis phases is between 1 and 3 milliseconds. We note that the reconstruction phase for stable ray tracing has a higher impact than the one in supersampling, given that we need to adapt it to the irregular number of samples we have per pixel.

We made a comparison with Martin et al. [2002], tweaking the algorithm to fit modern GPU pipelines. For each sample, we generate

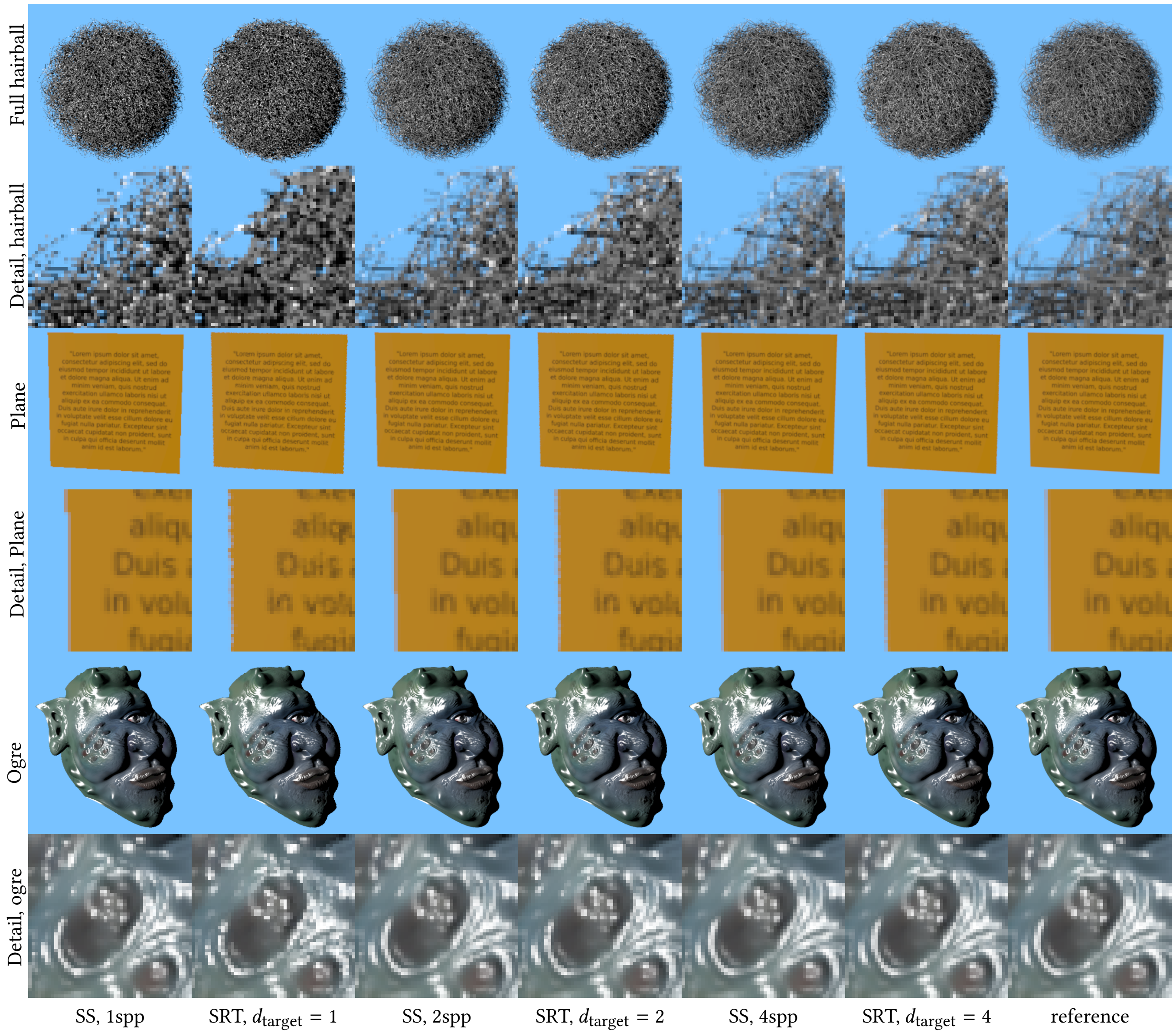


Figure 8: Quality comparisons between standard supersampling (SS) and stable ray tracing (SRT), for different number of samples. We use the same Gaussian reconstruction filter for all images. For low sample counts, stable ray tracing gives a result that is more temporally stable, at the price of introducing spatial aliasing artifacts.

a single vertex, rendered as a 1x1 pixel splat in the final destination pixel. This allows us to use the depth buffer to find the closest sample in the case of multiple samples landing in one pixel. If a sample does not exist, we simply generate a vertex outside of the view frustum. Then, a ray tracing step generates a sample in the middle of the pixel if it does not find one, and traces the ray. Relatively, our implementation is a bit faster than the original method, being only 1.2 times slower than the equivalent supersampling. The results are in a video (martin_comparison.mp4) and in Figure 7, where we provide a comparison with our method for similar sample counts. On the left-hand side of the video, we compare the two techniques for a panning view of a bump mapped plane. In this case, the quality of the two techniques is similar, except for the blurring due to the temporal filter employed by Martin et al. [2002]. If we consider the hairball (right-hand side of the video), our method is significantly more temporally stable. In addition, since we do not use an averaging temporal filter, our method produces sharper images (see Figure 7).

6.1 Application: Progressive Path Tracing

Our screen space sample data structure serves a double purpose: nearby samples in the data structure are close in world space, and

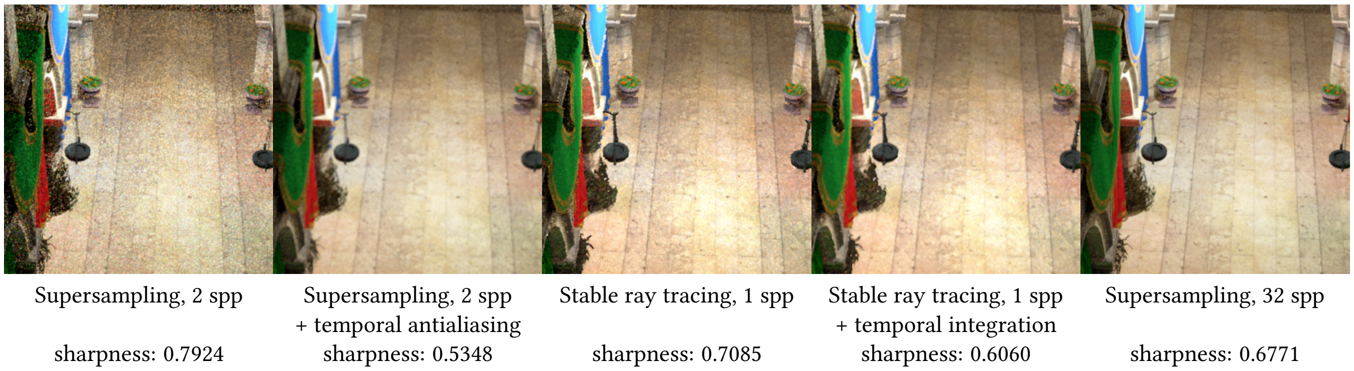


Figure 9: Including indirect illumination, the different techniques are here applied to a frame in the Sponza video.

the majority of samples are consistent in world space across frames. These properties make stable ray tracing suitable for accumulating view-independent but time-dependent information, such as diffuse indirect illumination.

As a proof of concept, we apply our technique on top of standard unidirectional path tracing to cache diffuse indirect illumination in a dynamic scene. For performance reasons, our path tracing has a fixed maximum trace depth. For each frame, we choose a random direction, trace a new path in that direction, and accumulate the final result. Directions are sampled using a cosine-weighted hemispherical distribution. For a completely static scene, we could give equal importance to all frames. Since we want to be able to react to dynamic content in the scene, we use a simple exponential moving average [Nehab et al. 2007; Scherzer et al. 2007] with integration factor 0.1. Our focus is here to illustrate the virtues of stable ray tracing in accumulation. More complicated sampling schemes are possible, such as accumulating indirect illumination to allow convergence when camera and scene are static, or from literature [Herzog et al. 2010; Yang et al. 2009].

Like the hairball video, we provide a similar video comparison for our global illumination method. In this video (sponza.mp4), we compare our progressive path tracing to a similar performance supersampling with 2 samples per pixel. As in the previous section, we provide comparisons with and without the temporal integration schemes. In this comparison, we observe that stable ray tracing improves temporal stability for a scene with a dynamic moving light. Some noise is still present, mostly due to fireflies generated by the new shading directions chosen for each sample. Since we use a screen space data structure, results must be re-generated upon disocclusion. This is why the flagpoles in the video leave trails of higher variance content. Figure 9 compares a cutout of a still frame of the Sponza video. One should note that the blur incurred by the temporal post-processing filters is good at hiding the stochastic noise of the path tracing. However, as is clear from visual comparison and the CPBD-based image sharpness measurements, the blurring of temporal antialiasing on top of supersampling is too much. We also note how the reference rendering in this case also has a lower sharpness score than results without temporal post-processing filters. This is mainly due to the noise in these two images being considered as sharpness.

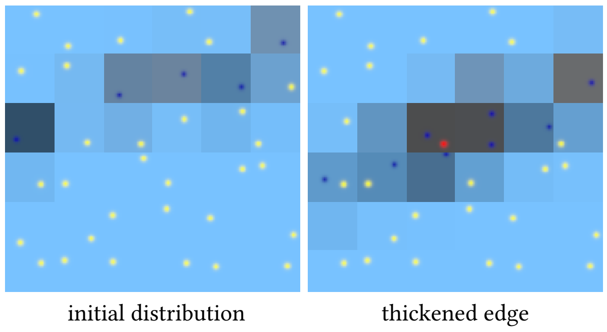


Figure 10: Edge thickening. Blue samples belong to one of the hairball strands, yellow samples belong to the background, and red samples are occluded. In the right image, the camera has moved upwards, so that the apparent motion in screen space of the edge is downwards. The low local density in the area above the strand causes the thickening.

7 DISCUSSION

Stable ray tracing improves temporal stability while retaining sharpness (hairball video and Figure 5). Our algorithm offers an intermediate solution between supersampling, which is sharp but temporally unstable, and temporal antialiasing, which is too blurry. The reason for this excessive blurriness is the high temporal instability in the input from supersampling. Since we do not have this temporal instability, we can apply a more relaxed temporal filtering (larger α) and thus strike a compromise between stability and sharpness. On the other hand, we cannot use jittering and therefore pay the price of spatial aliasing artifacts. These artifacts are particularly evident at lower sampling rates, resulting in broken or thickened edges and changed highlight patterns (Figure 8).

Spatial aliasing artifacts arise from the fact that we do not estimate the screen space coverage of each sample, but rather give them the same weight in the reconstruction phase. As we illustrate in Figure 10, this causes edge thickening. The distribution of samples changes a bit, but not enough to change the density. New samples are therefore added. The small gap introduced by the change in distribution is filled as possible by the reconstruction algorithm, causing the edge to thicken. A lower $d_{\text{tolerance}}$ could mitigate this problem by fixing the distribution more quickly wherever necessary. However, lowering this parameter would cause samples to get recycled more often, leading as well to temporal instability. This screen space coverage problem is partly to blame for the residual temporal instability of stable ray tracing, since each sample would have a different estimated coverage every frame.

As previously noted, the overhead of our technique is similar to that of Martin et al. [2002]. In our video comparison, we see how we reduce temporal artifacts, by allowing an irregular number of samples per pixel in our technique. This allow us to remove the originally proposed scene-based temporal filter, increasing the sharpness of the final image in the process. Although the overhead added by the reprojection and analysis phases are relatively low, there is an additional verification overhead when comparing on an iso-sample-rate basis. This penalty is due to load balancing issues resulting from the nonuniform screen-space sampling patterns, and subsequent varying amount of per-pixel work, generated by reprojection. We expect that the ray tracing overhead can be reduced by performing a load balancing step prior to tracing rays.

Our Sponza video exemplifies the potential of stable ray tracing as a technique for caching indirect light. In this example, due to the nature of our accumulation scheme, the fireflies generated by the path tracing procedure cause an additional level of temporal instability. However, our algorithm still retains its qualities, retaining a higher temporal stability (at least when temporal filtering is not used to hide it) and better image sharpness (Figure 9).

8 CONCLUSION

We presented a new practical technique for stable shading in interactive ray tracing. Our technique is based on sample reprojection and introduces low cost sample analysis for generating and evicting samples in the reprojection cache. The stable ray tracing that we propose is useful for striking a balance between temporal stability and image sharpness in interactive ray tracing applications. This comes at the cost of spatial aliasing and around a factor 1.5 hit to the performance. If the rendering budget allows a target sample density of just 4 samples per pixel, our technique can eliminate most spatial aliasing artifacts and provide a visually pleasing (sharp, antialiased) and fairly temporally stable result. Since we have stable shading in a ray tracing context, we can use our shading cache to add global illumination effects such as progressively path traced indirect illumination. In general, our algorithm eases the use of progressive techniques when a scene is dynamic.

REFERENCES

Stephen J. Adelson and Larry F. Hodges. 1995. Generating exact ray-traced animation frames by reprojection. *IEEE Computer Graphics and Applications* 15, 3 (1995), 43–52.

Sig Badt, Jr. 1988. Two algorithms for taking advantage of temporal coherence in ray tracing. *The Visual Computer* 4, 3 (1988), 123–132.

Kavita Bala, Bruce Walter, and Donald P. Greenberg. 2003. Combining edges and points for interactive high-quality rendering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003)* 22, 3 (July 2003), 631–640.

John Chapman, Thomas W. Calvert, and John Dill. 1991. Spatio-temporal coherence in ray tracing. In *Proceedings of Graphics Interface (GI '91)*. 101–108.

Shenchang Eric Chen and Lance Williams. 1993. View interpolation for image synthesis. In *Proceedings of SIGGRAPH 93*. ACM, 279–288.

Robert L. Cook, Loren Carpenter, and Edwin Catmull. 1987. The Reyes image rendering architecture. *Computer Graphics (Proceedings of SIGGRAPH 87)* 21, 4 (July 1987), 95–102.

Henri Gouraud. 1971. Continuous shading of curved surfaces. *IEEE Trans. Comput.* 20, 6 (June 1971), 623–629.

Eduard Gröller and Werner Purgathofer. 1991. Using temporal and spatial coherence for accelerating the calculation of animation sequences. In *Proceedings of Eurographics (EG '91)*, Vol. 91. 103–113.

Vlastimil Havran, Cyrille Damez, Karol Myszkowski, and Hans-Peter Seidel. 2003. An efficient spatio-temporal architecture for animation rendering. In *Rendering Techniques 2003 (Proceedings of EGSR 2003)*, Per H. Christensen and Daniel Cohen-Or (Eds.). Eurographics Association, 106–117.

Robert Herzog, Elmar Eisemann, Karol Myszkowski, and Hans-Peter Seidel. 2010. Spatio-temporal upsampling on the GPU. In *Proceedings of Interactive 3D Graphics and Games (I3D '10)*. ACM, 91–98.

Jose A. Iglesias-Guitian, Bochang Moon, Charalampos Koniaris, Eric Smolikowski, and Kenny Mitchell. 2016. Pixel history linear models for real-time temporal filtering. *Computer Graphics Forum (Proceedings of Pacific Graphics 2016)* 35, 7 (October 2016), 363–372.

Jorge Jimenez, Jose I. Echevarria, Tiago Sousa, and Diego Gutierrez. 2012. SMAA: enhanced subpixel morphological antialiasing. *Computer Graphics Forum (Proceedings of Eurographics 2012)* 31, 2pt1 (May 2012), 355–364.

Brian Karis. 2014. High-quality temporal supersampling. In *Advances in Real-Time Rendering in Games, Part I*. Number 10 in ACM SIGGRAPH 2014 Courses. http://advances.realtimerendering.com/s2014/

William R. Mark, Leonard McMillan, and Gary Bishop. 1997. Post-rendering 3D warping. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics (I3D '97)*. ACM, 7–16.

William Martin, Peter Shirley, Steven Parker, William Thompson, and Erik Reinhard. 2002. Temporally coherent interactive ray tracing. *Journal of Graphics Tools* 7, 2 (2002), 41–48.

Morgan McGuire. 2011. Computer Graphics Archive. (August 2011). http://graphics.cs.williams.edu/data

Koichi Murakami and Katsuhiko Hirota. 1992. Incremental ray tracing. In *Photorealism in Computer Graphics (Proceedings of EGWR 1990)*, K. Bouatouch and C. Bouville (Eds.). Springer, 17–32.

N. D. Narvekar and L. J. Karam. 2011. A no-reference image blur metric based on the cumulative probability of blur detection (CPBD). *IEEE Transactions on Image Processing* 20, 9 (September 2011), 2678–2683.

Diego Nehab, Pedro V. Sander, Jason Lawrence, Natalya Tatarchuk, and John R. Isidoro. 2007. Accelerating real-time shading with reverse reprojection caching. In *Proceedings of Graphics Hardware (GH 2007)*. 25–36.

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: a general purpose ray tracing engine. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010)* 29, 4 (July 2010), 66:1–66:13.

Anjul Patney, Marco Salvi, Joohwan Kim, Anton Kaplanyan, Chris Wyman, Nir Benty, David Luebke, and Aaron Lefohn. 2016. Towards foveated rendering for gaze-tracked virtual reality. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2016)* 35, 6 (November 2016), 179:1–179:12.

Daniel Scherzer, Stefan Jeschke, and Michael Wimmer. 2007. Pixel-correct shadow maps with temporal reprojection and shadow test confidence. In *Rendering Techniques 2007 (Proceedings of EGSR 2007)*. Eurographics Association, 45–50.

Pitchaya Sitthi-amorn, Jason Lawrence, Lei Yang, Pedro V Sander, and Diego Nehab. 2008a. An improved shading cache for modern GPUs. In *Proceedings of Graphics Hardware (GH 2008)*. Eurographics Association, 95–101.

Pitchaya Sitthi-amorn, Jason Lawrence, Lei Yang, Pedro V Sander, Diego Nehab, and Jiahe Xi. 2008b. Automated reprojection-based pixel shader optimization. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2008)* 27, 5 (December 2008), 127:1–127:11.

Takehiro Tawara, Karol Myszkowski, Kirill Dmitriev, Vlastimil Havran, Cyrille Damez, and Hans-Peter Seidel. 2004. Exploiting temporal coherence in global illumination. In *Proceedings of Spring Conference on Computer Graphics (SCCG 2004)*. ACM, 23–33.

Edgar Velázquez-Armendáriz, Eugene Lee, Kavita Bala, and Bruce Walter. 2006. Implementing the render cache and the edge-and-point image on graphics hardware. In *Proceedings of Graphics Interface 2006 (GI '06)*. Canadian Information Processing Society, 211–217.

Bruce Walter, George Drettakis, and Donald P. Greenberg. 2002. Enhancing and optimizing the render cache. In *Proceedings of the Eurographics Workshop on Rendering (EGWR 2002)*. ACM Press, 37–42.

Bruce Walter, George Drettakis, and Steven Parker. 1999. Interactive rendering using the render cache. In *Rendering techniques '99 (Proceedings of EGWR 1999)*. Springer, 19–30.

Lei Yang, Diego Nehab, Pedro V. Sander, Pitchaya Sitthi-amorn, Jason Lawrence, and Hugues Hoppe. 2009. Amortized supersampling. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2009)* 28, 5 (December 2009), 135:1–135:12.

Peng Zhou and Yanyun Chen. 2015. Variance reduction using interframe coherence for animated scenes. *Computational Visual Media* 1, 4 (December 2015), 343–349.

Tenghui Zhu, Rui Wang, and David Luebke. 2005. A GPU accelerated render cache. In *Proceedings of Pacific Graphics 2005 (short paper)*.