CHAPTER 13

Ray Traced Shadows: Maintaining Real-Time Frame Rates

Jakub Boksansky,[1] Michael Wimmer,[2] and Jiri Bittner[1]
[1]Czech Technical University in Prague
[2]Technische Universität Wien

ABSTRACT

Efficient and accurate shadow computation is a long-standing problem in computer graphics. In real-time applications, shadows have traditionally been computed using the rasterization-based pipeline. With recent advances of graphics hardware, it is now possible to use ray tracing in real-time applications, making ray traced shadows a viable alternative to rasterization. While ray traced shadows avoid many problems inherent in rasterized shadows, tracing every shadow ray independently can become a bottleneck if the number of required rays rises, e.g., for high-resolution rendering, for scenes with multiple lights, or for area lights. Therefore, the computation should focus on image regions where shadows actually appear, in particular on the shadow boundaries.

We present a practical method for ray traced shadows in real-time applications. Our method uses the standard rasterization pipeline for resolving primary-ray visibility and ray tracing for resolving visibility of light sources. We propose an adaptive sampling algorithm for shadow rays combined with an adaptive shadow-filtering method. These two techniques allow computing high-quality shadows with a limited number of shadow rays per pixel. We evaluated our method using a recent real-time ray tracing API (DirectX Raytracing) and compare the results with shadow mapping using cascaded shadow maps.

13.1 INTRODUCTION

Shadows contribute significantly to realistic scene perception. Due to the importance of shadows, many techniques have been designed for shadow computation in the past. While offline rendering applications use ray tracing for shadow evaluation [20], real-time applications typically use shadow maps [21]. Shadow mapping is highly flexible in terms of scene geometry, but it has several important issues:

> Perspective aliasing, which shows as jaggy shadows, due to insufficient shadow-map resolution or poor use of its area.

> Self-shadowing artifacts (shadow acne) and disconnected shadows (Peter Panning).

> Lack of penumbras (soft shadows).

> Lack of support for semitransparent occluders.

A number of techniques have been developed to address these issues [7, 6]. Usually, a combination of several of them and manual fine-tuning by the scene designer are required to achieve good results. This makes an efficient implementation of shadow mapping complicated, and different solutions are usually required for different scenes.

Ray tracing [20] is a flexible rendering paradigm that can compute accurate shadows with a simple algorithm and is able to handle complex lighting (area lights, semitransparent occluders) in an intuitive and scalable way. However, it has been difficult to achieve ray tracing performance that is sufficient for real-time applications. This was due to limited hardware resources as well as implementation complexity of the underlying algorithms required for real-time ray tracing, such as fast construction and maintenance of spatial data structures. There was also no explicit ray tracing support in popular graphics APIs used for real-time applications.

With the introduction of NVIDIA RTX and DirectX Raytracing (DXR), it is now straightforward to exploit ray tracing using DirectX and Vulkan APIs. The recent NVIDIA Turing graphics architecture provides hardware support for DXR using the dedicated *RT Cores*, which greatly improve ray tracing performance. These new features combine well with emerging *hybrid rendering* methods [11] that use rasterization to resolve primary-ray visibility and ray tracing to compute shadows, reflections, and other illuminations effects.

However, even with the new powerful hardware support, we have to use our resources wisely when rendering high-quality shadows using ray tracing. A naive algorithm might easily cast too many rays to sample shadows from multiple light sources and/or area light sources, leading to low frame rates. See Figure 13-1 for an example.

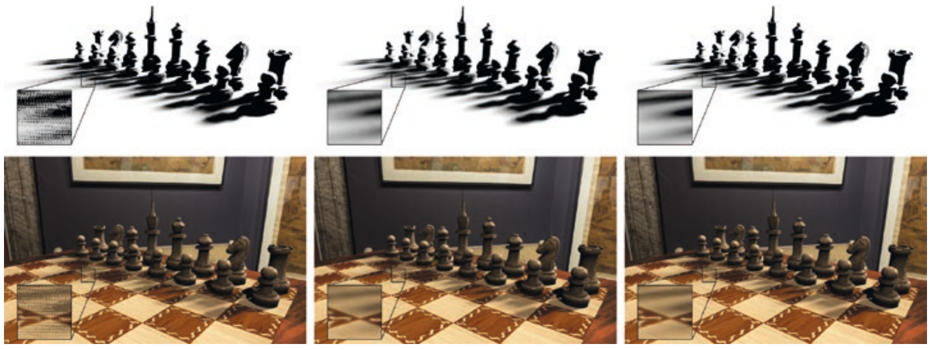

Figure 13-1. *Left: soft shadows rendered using naive ray traced shadows with 4 samples per pixel running at 3.6 ms per frame. Center: soft shadows rendered using our adaptive method with 0 to 5 samples per pixel running at 2.7 ms per frame. Right: naive ray traced shadows using 256 samples per pixel running at 200 ms per frame. Times measured using a GeForce RTX 2080 Ti GPU. Top: visibility buffers. Bottom: final images.*

In this chapter, we introduce a method that follows the hybrid rendering paradigm. Our method optimizes the evaluation of ray traced shadows by using adaptive shadow sampling and adaptive shadow filtering based on a spatiotemporal analysis of light-source visibility. We evaluate our method using the Falcor [3] framework and compare it with cascaded shadow maps [8] and naive ray traced shadows [20].

13.2 RELATED WORK

Shadows have been a focus of computer graphics research since the very beginning. They are a native element of Whitted-style ray tracing [20], where for each hit point a shadow ray is cast to each light source to determine mutual visibility. Soft shadows were introduced through distributed ray tracing [4], where the shadow term is calculated as an average of multiple shadow rays cast to an area light source. This principle is still the basis for many soft shadow algorithms today.

Interactive shadows were made possible through the shadow mapping [21] and shadow volume [5] algorithms. Due to its simplicity and speed, most interactive applications nowadays use shadow mapping, despite a number of disadvantages and artifacts caused by its discrete nature. Several algorithms for soft shadows are based on shadow mapping, most notably percentage closer soft shadows [9]. However, despite the many approaches and improvements to the original algorithms (for a comprehensive overview, see the book and course by Eisemann et al. [6, 7]), robust and fast soft shadows are still an elusive goal.

Inspired by advances in interactive ray tracing [18], researchers recently went back to investigating the use of ray tracing for hard and soft shadows. However, instead of performing a full ray tracing pass, a key idea was to use rasterization for the primary rays and to use ray tracing for only shadow rays [1, 19], leading to a hybrid rendering pipeline. To make this viable for soft shadows, the industry is experimenting with temporal accumulation in various ways [2].

The NVIDIA Turing architecture finally introduced fully hardware-accelerated ray tracing to the consumer market, and easy integration with the rasterization pipeline exists in the DirectX (DXR) and Vulkan APIs. Still, soft shadows for multiple light sources pose a challenge and require intelligent adaptive sampling and temporal reprojection approaches, as we will describe in this chapter.

The advent of real-time ray tracing also opens the door for other hybrid rendering techniques, for example adaptive temporal antialiasing, where pixels that cannot be rendered through reprojection are ray traced [14]. Temporal coherence has been used specifically for soft shadows before [17], but here we introduce a much simpler temporal coherence scheme based on a novel variation measure to estimate the required sample count.

13.3 RAY TRACED SHADOWS

Shadows appear when a scene object—a shadow caster—blocks light that would otherwise contribute to illumination at another scene object—the shadow receiver. Shadows can appear due to direct or indirect illumination. Direct illumination shadows are induced when the visibility of primary light sources is blocked, indirect illumination shadows are induced when strong reflections or refractions of light at scene surfaces are blocked. In this chapter, we focus on the case of direct illumination—indirect illumination can be evaluated independently using some standard global-illumination technique such as path tracing or many-light methods.

The outgoing radiance $L(p, \omega_o)$ at a point P in direction ω_o is defined by the rendering equation [12]:

$$L(P,\omega_o) = L_e(\omega_o) + \int_\Omega f(P,\omega_i,\omega_o)\, L_i(P,\omega_i)(\omega_i \cdot \hat{\mathbf{n}}_P)\, d\omega_i, \tag{1}$$

where $L_e(\omega_o)$ is the self-emitted radiance, $f(P, \omega_i, \omega_o)$ is the BRDF, $L_i(P, \omega_i)$ is the incoming radiance from direction ω_i, and $\hat{\mathbf{n}}_P$ is the normalized surface normal at point P.

For the case of direct illumination with a set of point light sources, the direct illumination component of L can be written as a sum of contributions from individual light sources:

$$L_d(P, \omega_o) = \sum_l f(P, \omega_l, \omega_o) L_l(P_l, \omega_l) v(P, P_l) \frac{\omega_l \cdot \hat{\mathbf{n}}_P}{\|P - P_l\|^2}, \tag{2}$$

where P_l is the position of light l, the light direction is $\omega_l = (P_l - P)/\|P_l - P\|$, $L_l(P_l, \omega_l)$ is the radiance emitted from light source l in direction ω_l, and $v(P, P_l)$ is the visibility term, which equals 1 if the point P_l is visible from P and 0 if it is not.

The evaluation of $v(p, p_l)$ can easily be performed by shooting a ray from P toward P_l and checking if the corresponding line segment is unoccluded. Care must be taken near the endpoints of the line segment not to include the self-intersection of the geometry of the shaded point or the light source. This is usually resolved by shrinking the parametric range for valid intersection by a small ε-threshold.

The L_d due to an area light source a is given by

$$L_d(P, \omega_o) = \int_{X \in A} f(P, \omega_X, \omega_o) L_a(X, \omega_X) v(P, X) \frac{(\omega_X \cdot \hat{\mathbf{n}}_P)(-\omega_X \cdot \hat{\mathbf{n}}_X)}{\|P - X\|^2} dA, \tag{3}$$

where A is the surface of light a, $\hat{\mathbf{n}}_X$ is the normal of the light source surface at point X, $\omega_X = (X - P)/\|X - P\|$ is the direction from point P toward point X on the light source, $L_a(X, \omega_X)$ is the radiance emitted from point X in direction ω_X, and $v(P, X)$ is the visibility term that equals 1 if the point X is visible from P and 0 if it is not.

This integral is commonly evaluated by Monte Carlo integration using a set of well-distributed samples S on the light source:

$$L_d(P, \omega_o) \approx \frac{1}{|S|} \sum_{X \in S} f(P, \omega_X, \omega_o) L_a(X, \omega_X) v(P, X) \frac{(\omega_X \cdot \hat{\mathbf{n}}_P)(-\omega_X \cdot \hat{\mathbf{n}}_X)}{\|P - X\|^2}, \tag{4}$$

where $|S|$ is the number of light samples. In our work we separate the shading and visibility terms, and for shading we approximate the area light source with a centroid C of the light source:

$$L_d(P, \omega_o) \approx f(P, \omega_C, \omega_o) L_a(C, \omega_C) \frac{(\omega_C \cdot \hat{\mathbf{n}}_P)(-\omega_C \cdot \hat{\mathbf{n}}_C)}{\|P - C\|^2} \frac{1}{|S|} \sum_{X \in S} v(P, X). \tag{5}$$

This allows us to accumulate the results of visibility tests for each light within a given frame and store them in a dedicated *visibility buffer* for each light. The visibility buffer is a screen-sized texture that holds visibility terms for each pixel. A more elaborate method of shading and visibility separation was recently proposed by Heitz et al. [11], which might be used for light sources with large areas or more complicated BRDFs. Separating visibility allows us to decouple visibility computation from shading as well as analyzing and using the temporal coherence of visibility. An illustration of visibility evaluation for a point light source and an area light source is shown in Figure 13-2. The difference between the resulting shadows is shown in Figure 13-3.

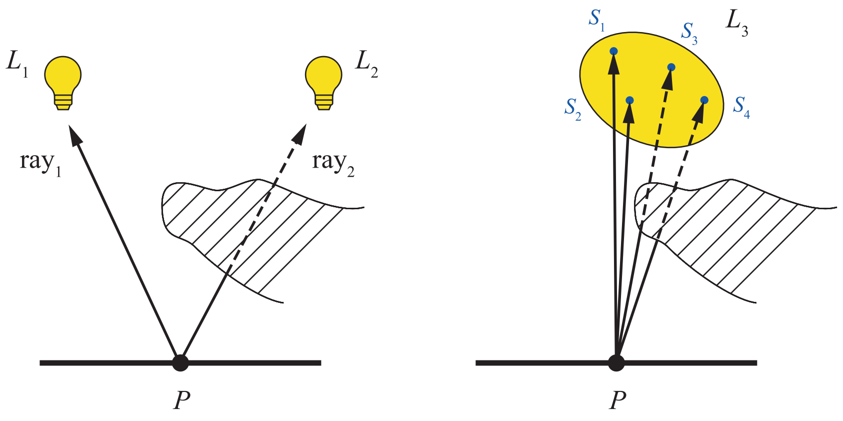

Figure 13-2. *Left: for point light sources, a single shadow ray is cast toward each light source from the shaded point P. The ray toward light source L_2 is blocked by an occluder, resulting in $v(P, L_2) = 0$. The ray toward L_1 is unoccluded, thus $v(P, L_1) = 1$. Right: the visibility of a disk light source is evaluated by sampling using several shadow rays.*

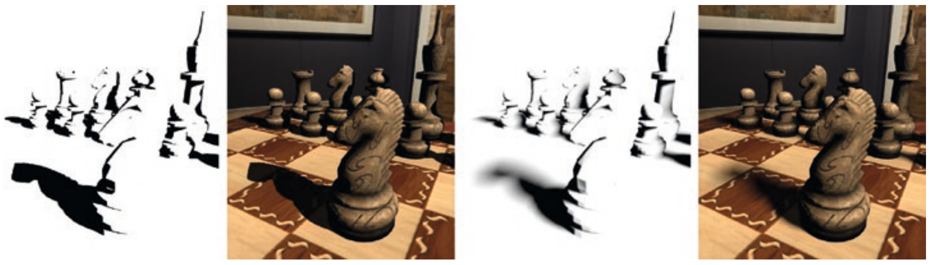

Figure 13-3. *An example of hard shadows (left) and soft shadows (right) computed by ray tracing, showing both visibility buffer and shaded image.*

13.4 ADAPTIVE SAMPLING

A naive implementation of shadow computation using ray tracing requires a high number of rays to achieve a desired shadow quality, especially for larger area lights, as shown in Figure 13-1. This would decrease performance considerably with an increase in the number and/or size of lights. Because a high number of rays is required only in penumbra areas of an image, we base our method on identifying these areas and then using more rays to sample them effectively. The fully lit and fully occluded areas are sampled sparsely, and the saved computational resources can be used for other ray tracing tasks such as reflections.

13.4.1 TEMPORAL REPROJECTION

To effectively increase the sample count used per pixel, we use temporal reprojection, which allows us to accumulate visibility values for visible scene surfaces over time. Temporal reprojection is becoming a standard tool in many recent real-time rendering methods [15], and in many cases it is already implemented within the application rasterization pipeline. We use the accumulated values for two purposes: first, estimating visibility variation to derive the required sample count, and second, determining the kernel size for filtering the sampled visibility.

We store the results of visibility calculations from previous frames in a cache containing four frames. To ensure correct results for dynamic scenes, we use reverse reprojection [15], which handles the camera movement. When starting an evaluation of a new frame, we perform reverse reprojection of three previous frames, stored in the cache, to the current frame. Thus, we always have a four-tuple of values from four consequent frames aligned with the image corresponding to the current frame.

Given a point P_t in clip space in frame t, the reprojection finds the corresponding clip-space coordinates $\bar{P}_{t-1}$ in frame $t - 1$ as

$$\bar{P}_{t-1} = \mathbf{C}_{t-1}\mathbf{V}_{t-1}\mathbf{V}_t^{-1}\mathbf{C}_t^{-1}P_t, \tag{6}$$

where $\mathbf{C}_t$ and $\mathbf{C}_{t-1}$ are the camera projection matrices and $\mathbf{V}_t$ and $\mathbf{V}_{t-1}$ are the camera viewing matrices. After reprojection we check for depth discontinuities and discard invalid correspondences (mostly disocclusions). Depth discontinuities are detected using a relative depth difference condition, i.e., the point is successfully reprojected if the following condition holds:

$$\left|1 - \frac{\bar{P}_{t-1}^z}{P_{t-1}^z}\right| < \varepsilon, \quad \varepsilon = c_1 + c_2\left|\hat{n}_z\right|, \tag{7}$$

where ε is an adaptive depth similarity threshold, $\hat{n}_z$ is a z-coordinate of the view-space normal of the corresponding pixel, and c_1 and c_2 are user-specified constants of linear interpolation (we used $c_1 = 0.003$ and $c_2 = 0.017$). The adaptive threshold ε allows for greater depth differences of valid samples on sloped surfaces.

For successfully reprojected points, we store image-space coordinates in the range 0 to 1. If the reprojection fails, we store negative values to indicate the reprojection failure for subsequent computations. Note that, as all previous frames have already been aligned during the previous reprojection steps, only one cache entry for storing the depth values P^z_{t-1} is sufficient.

13.4.2 IDENTIFYING PENUMBRA REGIONS

The number of samples (rays) required for a given combination of shaded point and light source generally depends on the light size, its distance to the shaded point, and the complexity of occluding geometry. Because this complexity would be difficult to analyze, we base our method on using the *temporal visibility variation measure* $\Delta v(x)$:

$$\Delta v_t(x) = \max\left(v_{t-1}(x) \ldots v_{t-4}(x)\right) - \min\left(v_{t-1}(x) \ldots v_{t-4}(x)\right), \quad (8)$$

where $v_{t-1}(x) \ldots v_{t-4}(x)$ are the cached visibility values for a pixel x in the four previous frames. Note that these visibility values are cached in a single four-component texture per light.

The described measure corresponds to the range variation measure, which is highly sensitive to extreme values of the visibility function. Therefore, this measure is more likely to detect penumbra regions than other, smoother variation measures such as the variance.

The variation is zero for either fully lit or fully occluded areas and is usually greater than zero in penumbra areas. Our sample sets are generated with regard to the fact that we use four frames for variation computation, so they repeat only after these four frames. See Section 13.5.1.

To make results more temporally stable, we apply a spatial filter on the variation measure followed by a temporal filter. The spatial filter is expressed as

$$\widetilde{\Delta v_t} = M_{5\times5}\left(\Delta v_t\right) * T_{13\times13}, \quad (9)$$

where $M_{5\times5}$ is a nonlinear maximum filter using a 5 × 5 neighborhood followed by a convolution with a low-pass tent filter $T_{13\times13}$ with a 13 × 13 neighborhood. The maximum filter makes sure that a high variation detected in a single pixel will cause a higher number of samples to be used in surrounding pixels too. This is important for dynamic scenes to make the method more temporally stable and for cases where the penumbra is completely missed in nearby pixels. The tent filter prevents abrupt changes in variation values to avoid flickering. Both filters are separable, therefore we execute them in two passes to reduce the computational effort.

Finally, we combine the spatially filtered variation measure $\widetilde{\Delta v_t}$ with temporally filtered values Δv_t from the four previous frames. For the temporal filtering, we use a simple box filter, and we intentionally use the raw Δv_t values that are cached prior to spatial filtering:

$$\overline{\Delta v_t} = \frac{1}{2}\left(\widetilde{\Delta v_t} + \frac{1}{4}\left(\Delta v_{t-1} + \Delta v_{t-2} + \Delta v_{t-3} + \Delta v_{t-4}\right)\right). \tag{10}$$

Such a combination of filters proved efficient in our tests as it is able to propagate the variation over larger regions (using maximum and tent filters). At the same time, it does not miss small regions with large variation by combining the spatially filtered variation with the temporally filtered variation values from the previous frames.

13.4.3 COMPUTING THE NUMBER OF SAMPLES

The decision on the number of samples to be used for a given point is based on the number of samples used in the previous frame and the current filtered variation $\overline{\Delta v_t}$. We use a threshold δ on the variation measure to decide whether to increase or decrease sampling density at the corresponding pixel. In particular, we maintain the sample counts $s(x)$ for each pixel and use the following algorithm to update $s(x)$ in the given frame:

1. If $\overline{\Delta v_t}(x) > \delta$ and $s_{t-1}(x) < s_{\max}$, increase the number of samples by one ($s_t(x) = s_{t-1}(x) + 1$).
2. If $\overline{\Delta v_t}(x) < \delta$ and the number of samples has been stable in the four previous frames, decrease the number of samples ($s_t(x) = s_{t-1}(x) - 1$).

The maximum number of samples per light $s_{\max}$ ensures a limited ray budget for each light per frame (we use $s_{\max} = 5$ for standard settings and $s_{\max} = 8$ for high-quality settings). The constraint of stability in the four previous frames used in step (2) induces a hysteresis into the algorithm and aims to prevent oscillations in the number of samples caused by a feedback loop between the number of samples and the variation. The described technique works with sufficient temporal stability and provides better results than directly computing $s(x)$ from $\overline{\Delta v_t}(x)$.

For pixels where reverse reprojection fails, we use $s_{\max}$ samples and replace all cached visibility values with the current result. When a reverse reprojection fails for all pixels on the screen, e.g., when the camera pose changes dramatically, a sudden performance drop occurs due to the high number of samples used in each pixel. To prevent the performance drop, we can detect large changes of camera pose on the CPU, and we can reduce the maximum number of samples ($s_{\max}$) for several subsequent frames. This will momentarily cause noisier results, but it will prevent frame-rate stuttering, which is usually more disturbing.

13.4.4 SAMPLING MASK

Pixels for which our algorithm computes sample counts equal to zero indicate a region with no temporal and spatial variation. This is mostly the case for fully lit and fully shadowed regions in an image. For these pixels we might skip the calculation of visibility completely and use value from the previous frame. However, this may lead to an accumulation of errors over time in these regions, for example when a light is moving fast or the camera is zooming slowly (in both these cases the reprojection succeeds, but visibility can change). Therefore, we use a mask that enforces regular sampling for at least one fourth of the pixels. We enforce sampling of individual blocks of pixels on the screen as performance tests have shown that shooting a single ray for one pixel out of four in a close neighborhood yields similar performance as shooting rays for each of these pixels (probably due to warp dependencies). Therefore, we enforce sampling of a block of $n_b \times n_b$ pixels on the screen (we get the best performance increase for $n_b = 8$).

To ensure that every pixel is sampled at least once in four frames, we use a matrix that checks if the sampling should be enforced in the current frame. We find an entry in a mask of size 4 × 4 repeated over the screen that corresponds to the location of the block. If the entry is equal to the current frame's sequence number modulo four, all pixels in blocks with zero sample counts are sampled with one shadow ray per pixel per light. The mask is set up so that in each quad of neighboring blocks, only one block will be evaluated. Furthermore, every pixel will be evaluated once in four consecutive frames to make sure that new shadows are detected. This is illustrated in Figure 13-4. An example of the sample distribution using the adaptive sampling is shown in Figure 13-5.

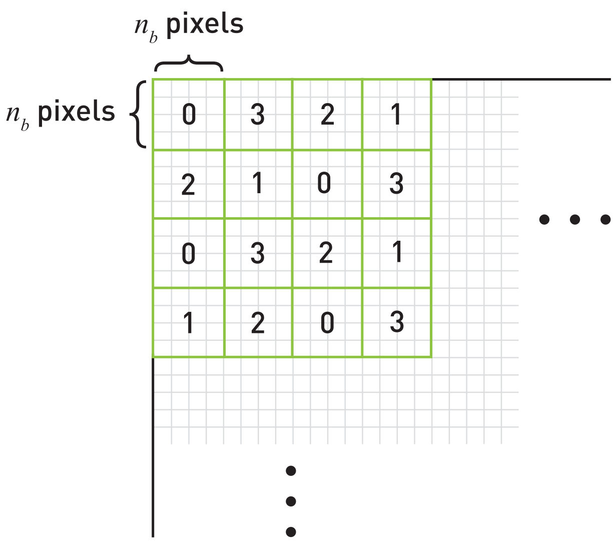

Figure 13-4. *An example of the sampling mask matrix. In each sequence of four consecutive frames, the shadow rays are enforced even for pixels with low visibility variation.*

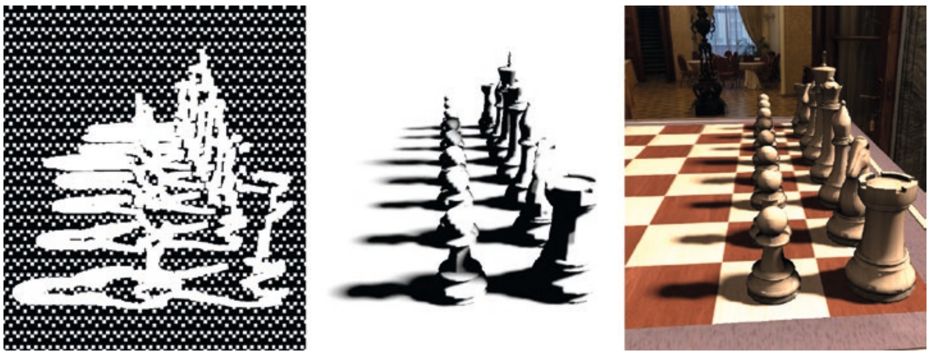

Figure 13-5. *Left: image showing the pixels with nonzero sample counts. Note the sampling of the penumbra regions and the pattern enforced by the sampling matrix. Center: visibility buffer. Right: final image.*

13.4.5 COMPUTING VISIBILITY VALUES

As a final step in our algorithm, we employ two filtering techniques on the visibility values themselves (as opposed to the visibility variation measure): temporal filtering, which makes use of results from previous frames, and spatial filtering, which applies a low-pass filter over visibility values and removes the remaining noise.

Recent denoising methods for global illumination, such as spatiotemporal variance-guided filtering (SVGF) by Schied et al. [16] and AI-based denoisers, can produce noise-free results from sequences of stochastically sampled images with as little as one sample per pixel. These methods take care to preserve edge sharpness after denoising (especially on textured materials), typically by using information from noise-free albedo and normal buffers. We use a simpler solution that is specifically tailored toward shadow computation and combines well with our adaptive sampling strategy for shadow rays.

13.4.5.1 TEMPORAL FILTERING

To apply temporal accumulation of visibility values, we calculate an average visibility value, effectively applying a temporal box filter on the cached reprojected visibility values:

$$\tilde{v}_t = \frac{1}{4}\left(v_t + v_{t-1} + v_{t-2} + v_{t-3}\right). \tag{11}$$

Using a temporal box filter leads to the best visual results, since our sample sets are generated to be interleaved over the last four frames. Note that our approach does not explicitly account for the movement of lights. Our results indicate that for interactive frame rates (>30 FPS) and caching only four previous frames, the artifacts introduced by this simplification are quite minor.

13.4.5.2 SPATIAL FILTERING

The spatial filter operates on the visibility buffer that was already processed by the temporal filtering step. We use a traditional cross bilateral filter with a variable-sized Gaussian kernel to filter the visibility. The size of the filter kernel is chosen between 1 × 1 and 9 × 9 pixels and is given by the variation measure $\widetilde{\Delta v_t}$—more variation in a given area results in more aggressive denoising. The filter size is scaled linearly in dependence on $\widetilde{\Delta v_t}$, while the maximum kernel size is achieved for a predefined variation of η (we used $\eta = 0.4$). To prevent popping when switching from one kernel size to the other, we store precalculated Gaussian kernels for each size and linearly interpolate the corresponding entries between the two closest kernels. This is especially important for blending with the smallest kernel size to preserve hard edges where needed.

We make use of depth and normal information to prevent shadows leaking over geometry discontinuities. This makes the filter nonseparable, but we apply it as if it was with reasonably good results, as can be seen in Figure 13-6. Samples whose depths do not satisfy Equation 7 are not taken into account. Additionally, we discard all samples for which the corresponding normals do not satisfy the normal similarity test:

$$\hat{\mathbf{n}}_p \cdot \hat{\mathbf{n}}_q > \zeta, \tag{12}$$

where $\hat{\mathbf{n}}_p$ is a normal at a pixel p, $\hat{\mathbf{n}}_q$ is a normal of the pixel q from the neighborhood of p, and ζ is a normal similarity threshold (we used $\zeta = 0.9$).

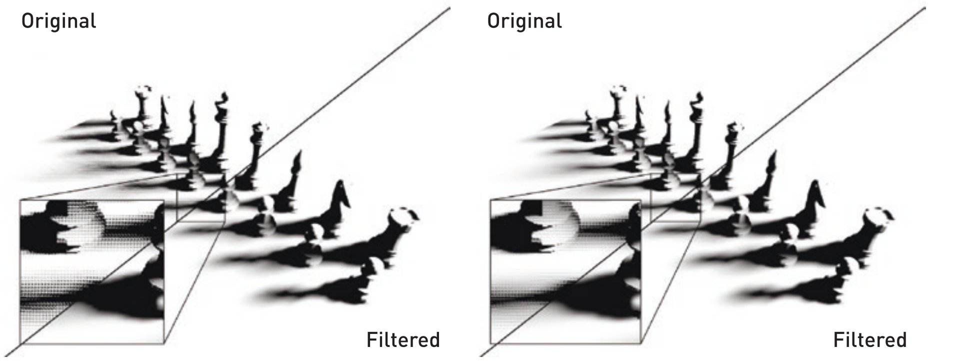

Figure 13-6. *Difference between raw visibility values and filtered result. Left: using naive shadow-ray tests with 8 samples per pixel (4.25 ms per frame). Right: our method using 1 to 8 samples per pixel and the sampling mask (2.94 ms per frame).*

The temporal filtering step packs the filtered visibility buffers for four lights into single four-component texture. Then, each spatial filtering pass operates on two of these textures at the same time, effectively denoising eight visibility buffers at once.

13.5 IMPLEMENTATION

This section describes details regarding the implementation of our algorithm.

13.5.1 SAMPLE-SET GENERATION

Our adaptive sampling method assumes that we work with samples that are interleaved over four frames. As the method uses different sample counts for each pixel, we generate an optimized set of samples for each size used in our implementation (1 to 8). In our implementation, we used two different quality settings: the *standard-quality* setting with $s_{\max} = 5$, and the *high-quality* setting with $s_{\max} = 8$.

Considering that we aim to interleave the samples over four frames and that the smallest effective spatial filter size is 3 × 3 (for spatial filtering), our sets contain $s_{\max} \times 4 \times 3 \times 3$ samples. This yields sample counts effectively used for a single pixel of 36 for 1 sample per pixel, 72 for 2 samples per pixel, and up to 288 for 8 samples per pixel.

In each of four consecutive frames, a different subset consisting of a quarter of these samples is used. Furthermore, in each pixel we use a different ninth of this subset. The choice of which ninth to use is given by pixel position within a block of 3 × 3 pixels repeated over the screen.

We optimize the direct output of a Poisson distribution generator to decrease the discrepancy of the whole sample set also for the four subsets used in consecutive frames and nine of their subsets used for different pixels. This procedure optimizes sample sets with respect to their usage in temporal and spatial filtering and reduces visual artifacts. An example sample set is shown in Figure 13-7.

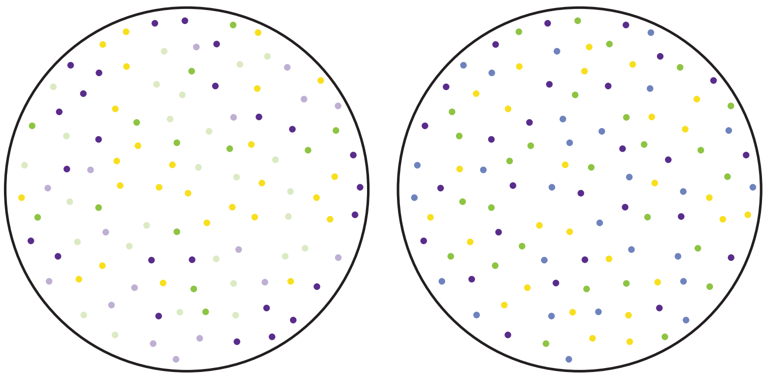

Figure 13-7. *Left: samples colored by position on the screen—similar colors will be evaluated in pixels close to each other. Right: samples colored by the frame number—samples with the same color will be used in the same frame. Samples are well distributed in both temporal and spatial domains. The figure shows a sample set for three samples per pixel.*

13.5.2 DISTANCE-BASED LIGHT CULLING

Even before casting the shadow rays, we can cull distant and low-intensity lights to increase performance. To do this, we calculate the range of each light—this is the distance where the intensity of a light becomes negligible due to its attenuation function. Before evaluating visibility, we compare the distance of the light to its range and simply store zero for non-contributing lights. Typical attenuation functions (inverse of squared distance) never reach zero, and thus it is practical to modify this function so that it reaches zero eventually, e.g., by implementing a linear drop-off below a certain threshold. This will decrease light ranges, making the culling more efficient while preventing popping when a light starts contributing again after being culled.

13.5.3 LIMITING THE TOTAL SAMPLE COUNT

Because our adaptive algorithm puts more samples in penumbras, a significant performance decrease can occur when the penumbra covers a large portion of the screen. For dynamic scenes, this could display as disturbingly high variations of frame rate.

We provide a method to limit the sample count globally based on computing the sum of the variation measures $\overline{\Delta v}$ over the whole image (we compute the sum using hierarchical reduction with mipmaps). If the sum rises above a certain threshold, we progressively limit the number of samples that can be used in each pixel. This threshold and the value at which a single sample per pixel should be used must be fine-tuned to the desired performance-to-visual-quality ratio. This will result in a momentary decrease in visual quality, but it can be preferable to stuttering caused by longer shadow calculation.

13.5.4 FORWARD RENDERING PIPELINE INTEGRATION

We implemented our algorithm within a forward rendering pipeline. Compared to deferred rendering, this pipeline provides advantages such as simpler transparency handling, support for more complex materials, hardware antialiasing (MSAA), and lower memory requirements.

Our implementation builds on top of the Forward+ pipeline introduced by Harada et al. [10], which makes use of a depth prepass and adds a light-culling stage to solve problems with overdraw and many lights. DXR makes integration of ray tracing into existing renderers straightforward, and considerable investment made into materials, special effects, etc. is therefore preserved when adding ray traced features such as shadows.

An overview of our method is shown in Figure 13-8. First, we perform the depth prepass to fill a depth buffer with no color buffer attached. After the depth prepass, we generate motion vectors based on camera movement and the normal buffer, which will be used later during denoising. The normal buffer is generated from depth values. Because it is not used for shading but denoising, this approximation works reasonably well.

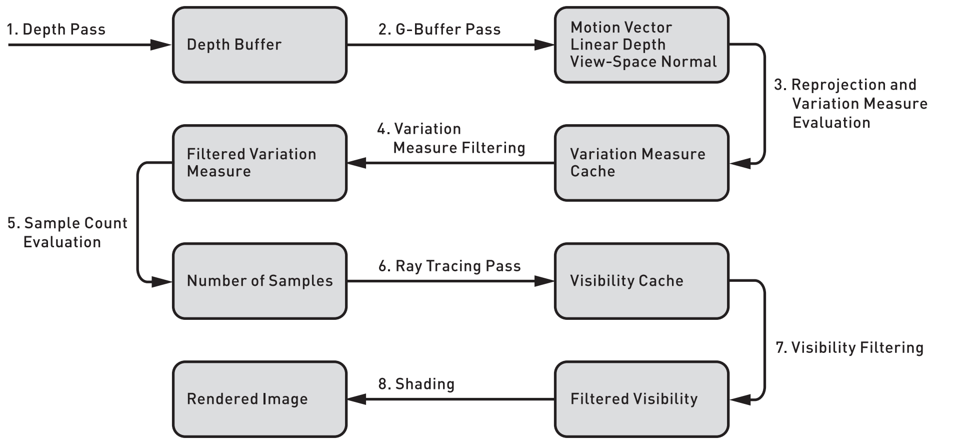

Figure 13-8. *Overview of our ray tracing shadow algorithm.*

The layout of the buffers used in our method is shown in Figure 13-9. The visibility cache, the variance measures, and the sample counts are cached over the last four frames for each light. The filtered visibility buffers and the filtered variation measure buffers are stored for only the last frame for each light. Note that the sample counts and the variation measures are packed into the same buffer.

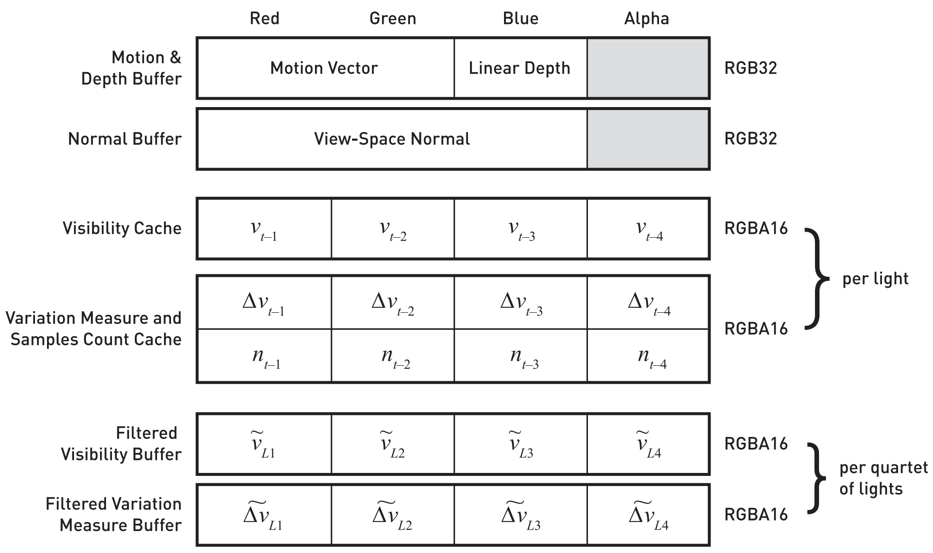

Figure 13-9. *Buffer layout used by our algorithm.*

Then, we generate visibility buffers for all lights using ray tracing. We use the depth-buffer values to reconstruct the world-space positions of visible pixels using inverse projection. World-space pixel positions can also be read directly from a G-buffer (if available) or evaluated by casting primary rays for greater precision. From these positions, we shoot shadow rays toward light sources to evaluate their visibility using our adaptive sampling algorithm. Results are denoised and stored in visibility buffers, which are passed to the final lighting stage. Visualizations of variation measures, sample counts, and filtering kernel sizes used by our shadow calculation are shown in Figure 13-10 for a single frame.

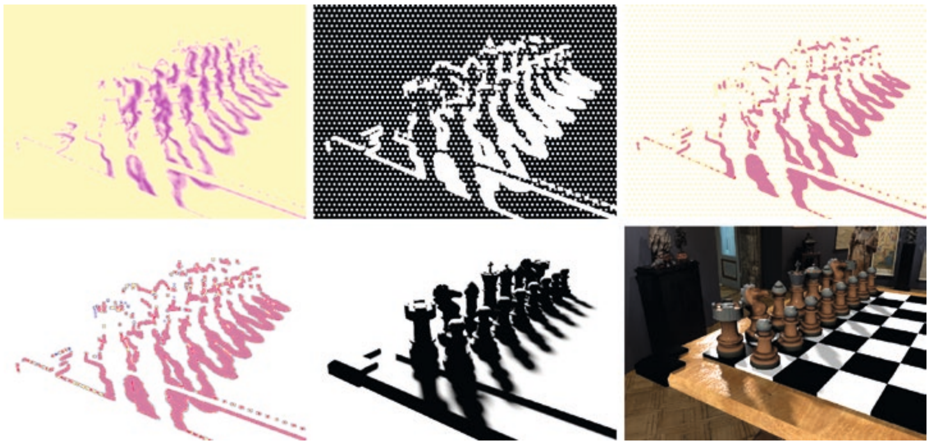

Figure 13-10. *Top left: filtered variation measure* $\widetilde{\Delta v_t}$. *Top center: areas with sample counts evaluated to zero shown in black. Top right: sample counts mapped to yellow-to-pink spectrum. Bottom left: spatial filtering kernel size levels mapped to different colors. Bottom center: filtered visibility buffer. Bottom right: final result.*

The lighting stage uses a single rasterization pass during which all scene lights are evaluated. A rasterized point is lit by all scene lights in a loop and the results are accumulated. Note that the visibility buffer of each light is queried before shading, which in turn is done only for visible lights—this provides implicit light culling to increase performance.

13.6 RESULTS

We evaluated our method for computing both hard and soft shadows and compared it with a reference shadow-mapping implementation. We used three test scenes of 20-second animation sequences with a moving camera. The Pub and Resort scenes have similar geometric complexity, but the Pub scene contains much larger area lights. The Breakfast scene has a significantly larger triangle count. The Pub and Breakfast scenes represent interiors, and thus they use point lights, and the exterior Resort scene uses directional lights. For computing soft shadows, these lights are treated as disk lights. We used the shadow-mapping implementation of the Falcor framework, which uses cascaded shadow map (CSM) and exponential variance shadow map (EVSM) [13] filtering. We used four CSM cascades for directional lights and one cascade for point lights, with the largest level using a shadow map of size 2048 × 2048. The screen resolution for all tests was 1920 × 1080.

We evaluated four shadow-computation methods: hard shadows computed using shadow mapping (SM hard), hard shadows computed using our method (RT hard), soft shadows computed using ray tracing with $s_{max} = 5$ (RT soft SQ), and soft shadows computed using ray tracing with $s_{max} = 8$ (RT soft HQ). The measurements are summarized in Table 13-1.

Table 13-1. *Overview of the measured results. The table shows the shadow-computation GPU times (in ms) for the tested methods when using one and four light sources. The measurements were performed on a GeForce RTX 2080 Ti GPU.*

	Pub		Resort		Breakfast	
	281k triangles		376k triangles		1.4M triangles	
	1 light	4 lights	1 light	4 lights	1 light	4 lights
SM hard	0.7	2.6	2.3	9.1	0.9	5.9
RT hard	1.4	4.8	1.3	3.4	1.6	3.7
RT soft SQ	3.2	13.5	2.7	8.3	4.7	11.0
RT soft HQ	3.5	19.9	2.9	12.0	6.5	16.2

13.6.1 COMPARISON WITH SHADOW MAPPING

The measurements in Table 13-1 show that for the Breakfast and Resort scenes with four lights, ray traced hard shadows (RT hard) outperform shadow mapping (SM hard) by about 40% and 60%, respectively. For the Breakfast scene, we attribute this to its large number of triangles. Increasing the number of triangles seems to slow down the rasterization pipeline used by shadow mapping more quickly than the RT Cores. The exterior Resort scene requires all four CSM cascades to be generated and filtered, causing significantly longer execution times for shadow mapping.

For the Pub scene (Figure 13-11) and the Breakfast scene (Figure 13-12) with one light, shadow mapping is about twice as fast as hard ray traced shadows. This is because only one CSM cascade is used for point lights, but it comes at the cost of visual artifacts. For the Pub scene, perspective aliasing occurs close to the camera (in the screen borders) and on the wall in the back. Also, shadows cast by chairs are disconnected from the ground. Trying to remedy these artifacts leads to shadow acne in other parts of the image. Ray traced shadows, on the other hand, do not suffer from these artifacts.

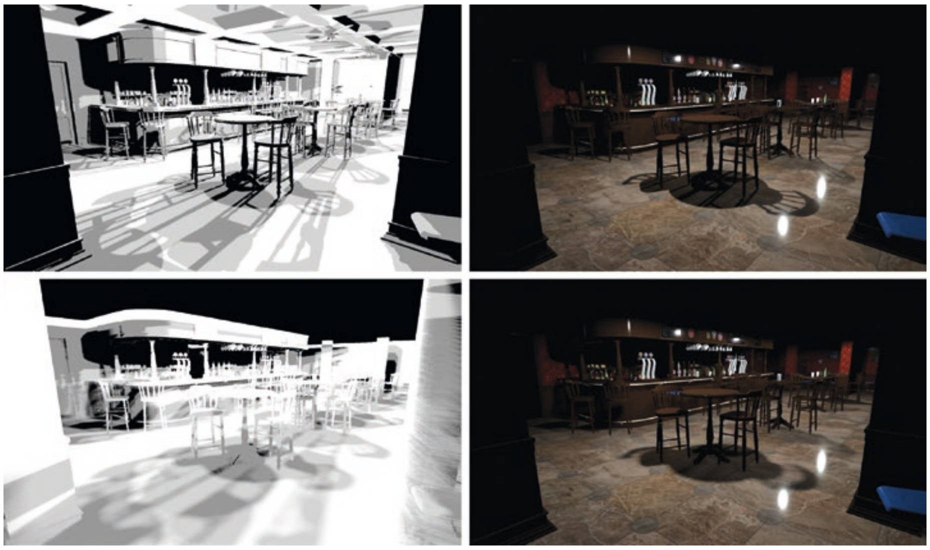

Figure 13-11. *Hard shadows comparison. Visibility buffers (left) and rendered image (right) for the Pub scene with four lights, showing hard shadows rendered using our method (top) and shadow mapping (bottom).*

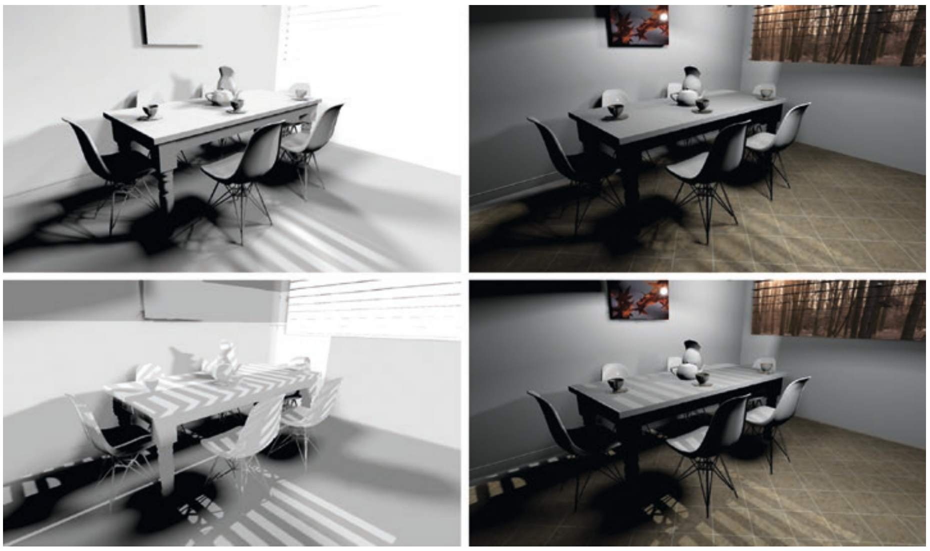

Figure 13-12. *Soft shadows comparison. Visibility buffers (left) and rendered image (right) for the Breakfast scene with four lights, showing soft shadows rendered using our method (top) and shadow mapping (bottom).*

For the Breakfast scene, EVSM filtering produces very soft and unfocused shadows under the table. This is likely due to the insufficient shadow-map resolution in this area, which is compensated for by stronger filtering. Using less aggressive filtering resulted in aliasing artifacts, which were more disturbing. For the Resort scene, the visual results of ray tracing and shadow mapping are quite similar; however, the ray traced shadows outperform shadow mapping in most tests.

13.6.2 SOFT SHADOWS VERSUS HARD SHADOWS

Comparing soft and hard ray traced shadows, in our tests it takes about 2–3 times longer to calculate soft shadows. This is, however, highly dependent on the size of the lights. For the Pub scene, which had lights set up to produce larger penumbras, calculation is up to 40% slower for four lights compared to the similarly complex Resort scene. This is because we are bound to use a high number of samples in larger areas. A visual comparison of the RT soft SQ and RT soft HQ methods is shown in Figure 13-13. Note that for the large Breakfast scene, the execution time did not increase linearly with the number of lights for the RT hard method. This indicates that the RT Cores were not yet fully occupied for the single light case.

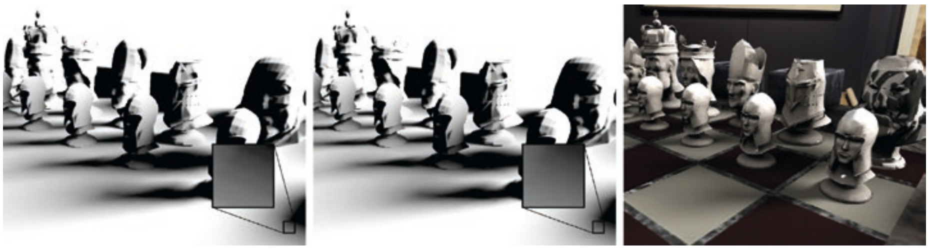

Figure 13-13. *Difference between standard and high-quality adaptive sampling. Left: normal quality (up to 5 samples per pixel). Center: high quality (up to 8 samples per pixel). Right: final render using the high-quality setting.*

Compared to the unoptimized calculation using 8 samples per pixel, our adaptive sampling method provides a combined speedup of about 40–50% for the tested scenes. Our method, however, achieves better visual quality thanks to the temporal accumulation.

13.6.3 LIMITATIONS

Our implementation of the proposed method currently has several limitations that might show as artifacts in fully dynamic scenes. In the current implementation, we do not consider motion vectors of moving objects, which reduces the success of reprojection for moving shadow receivers and can at the same time introduce false-positive reprojection successes for a particular combination of camera and shadow receiver movement (although this case should be quite rare).

More significantly, moving shadow casters are not handled by the method, which might introduce temporal shadow artifacts. On the positive side, our method uses a limited-size temporal buffer (only the last four frames are considered), and in combination with the aggressive variability measure, it will usually enforce dense sampling of the dynamic penumbras. Another problematic case is moving light sources, which we do not address explicitly at the moment. The situation is similar to moving shadow casters: a quickly moving light source causes severe changes in shadows that reduce the potential of adaptive sampling and can cause ghosting artifacts.

The current algorithm for maintaining a per-frame ray budget is relatively simple, and it would be desirable to use a technique that would directly relate the variation measure to the number of samples while aiming to minimize the perceived error (including shading). In that case it would be easier to guarantee frame rates while obtaining shadows of highest possible quality.

13.7 CONCLUSION AND FUTURE WORK

In this chapter, we have presented a method for calculating ray traced shadows using the modern DXR API within a rasterization forward-rendering pipeline. We proposed an adaptive shadow-sampling method that is based on estimating the variation of the visibility function over surfaces seen by the camera. Our method produces hard shadows as well as soft shadows using lights of various sizes. We have evaluated various configurations of light-sampling and shadow-filtering techniques and provided recommendations for best results.

We compared our method to a state-of-the-art shadow-mapping implementation in terms of visual quality and performance. In general, we conclude that the higher visual quality, simpler implementation, and high performance of ray traced shadows makes them preferable over shadow mapping on DXR-capable hardware. This will also move the burden of calculating shadows from rasterization to ray tracing hardware units, making more performance available for rasterization tasks. Using AI-based denoisers running on dedicated GPU cores can help even more in this respect.

With shadow mapping, scene designers are often challenged with minimizing the technique's artifacts by setting up technical parameters such as near/far planes, shadow-map resolutions, and bias and penumbra sizes not related to physical lighting. With ray traced shadows, there is still a burden on designers to make shadow calculation efficient and noise-free by using reasonable light sizes, ranges, and placement. We believe, however, that these parameters are more intuitive and closer to physically based lighting.

13.7.1 FUTURE WORK

Our method does not explicitly handle movement of lights, which can lead to ghosting artifacts from rapid light movement. A correct approach would be to discard cached visibility from previous frames when it is no longer valid after light movement between the frames.

Shadow mapping is not view-dependent, and a common optimization is to calculate shadow maps only when either a light or the scene changes. This optimization is not applicable for ray tracing, as ray traced visibility buffers need to be recalculated after every camera movement. Because of this, shadow mapping can still be preferable for scenarios where the shadow map is rarely updated. Therefore, a combination of high-quality ray traced shadows for significant light sources and shadow mapping for mostly static parts of the scene and/or less contributing lights can be desirable.

As mentioned in Section 13.3, an improved approach to combining shadows evaluated using our method with the analytic direct illumination, such as the one introduced by Heitz et al. [11], can be used to improve the correctness of the rendered images.

ACKNOWLEDGEMENTS

We thank Tomas Akenine-Möller for his feedback and help with the performance measurements, Nir Benty for assistance with the Falcor framework, and David Sedlacek for providing the environment maps. This research was supported by the Czech Science Foundation under project number GA18-20374S and by the MŠMT under the identification code 7AMB17AT021 within the activity MOBILITY (MSMT-539/2017-1).

REFERENCES

[1] Anagnostou, K. Hybrid Ray Traced Shadows and Reflections. Interplay of Light Blog, `https://interplayoflight.wordpress.com/2018/07/04/hybrid-raytraced-shadows-and-reflections/`, July 2018.

[2] Barré-Brisebois, Colin. Halén, H. PICA PICA & NVIDIA Turing. Real-Time Ray Tracing Sponsored Session, SIGGRAPH, 2018.

[3] Benty, N., Yao, K.-H., Foley, T., Kaplanyan, A. S., Lavelle, C., Wyman, C., and Vijay, A. The Falcor Rendering Framework. `https://github.com/NVIDIAGameWorks/Falcor`, July 2017.

[4] Cook, R. L., Porter, T., and Carpenter, L. Distributed Ray Tracing. *Computer Graphics (SIGGRAPH) 18*, 3 (July 1984), 137–145.

[5] Crow, F. C. Shadow Algorithms for Computer Graphics. *Computer Graphics (SIGGRAPH) 11*, 2 (August 1977), 242–248.

[6] Eisemann, E., Assarsson, U., Schwarz, M., Valient, M., and Wimmer, M. Efficient Real-Time Shadows. In *ACM SIGGRAPH Courses* (2013), pp. 18:1–18:54.

[7] Eisemann, E., Schwarz, M., Assarsson, U., and Wimmer, M. *Real-Time Shadows*, first ed. A K Peters Ltd., 2011.

[8] Engel, W. Cascaded Shadow Maps. In *ShaderX^5: Advanced Rendering Techniques*, W. Engel, Ed. Charles River Media, 2006, pp. 197–206.

[9] Fernando, R. Percentage-Closer Soft Shadows. In *ACM SIGGRAPH Sketches and Applications* (July 2005), p. 35.

[10] Harada, T., McKee, J., and Yang, J. C. Forward+: Bringing Deferred Lighting to the Next Level. In *Eurographics Short Papers* (2012), pp. 5–8.

[11] Heitz, E., Hill, S., and McGuire, M. Combining Analytic Direct Illumination and Stochastic Shadows. In *Symposium on Interactive 3D Graphics and Games* (2018), pp. 2:1–2:11.

[12] Kajiya, J. T. The Rendering Equation. *Computer Graphics (SIGGRAPH) 20*, 4 (August 1986), 143–150.

[13] Lauritzen, A. T. Rendering Antialiased Shadows using Warped Variance Shadow Maps. Master's thesis, University of Waterloo, 2008.

[14] Marrs, A., Spjut, J., Gruen, H., Sathe, R., and McGuire, M. Adaptive Temporal Antialiasing. In *Proceedings of High-Performance Graphics* (2018), pp. 1:1–1:4.

[15] Scherzer, D., Yang, L., Mattausch, O., Nehab, D., Sander, P. V., Wimmer, M., and Eisemann, E. A Survey on Temporal Coherence Methods in Real-Time Rendering. In *Eurographics State of the Art Reports* (2011), pp. 101–126.

[16] Schied, C., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C. R. A., Burgess, J., Liu, S., Dachsbacher, C., Lefohn, A., and Salvi, M. Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination. In *Proceedings of High-Performance Graphics* (2017), pp. 2:1–2:12.

[17] Schwärzler, M., Luksch, C., Scherzer, D., and Wimmer, M. Fast Percentage Closer Soft Shadows Using Temporal Coherence. In *Symposium on Interactive 3D Graphics and Games* (March 2013), pp. 79–86.

[18] Shirley, P., and Slusallek, P. State of the Art in Interactive Ray Tracing. ACM SIGGRAPH Courses, 2006.

[19] Story, J. Hybrid Ray Traced Shadows, `https://developer.nvidia.com/content/hybrid-ray-traced-shadows`. NVIDIA Gameworks Blog, June 2015.

[20] Whitted, T. An Improved Illumination Model for Shaded Display. Communications of the ACM 23, 6 (June 1980), 343–349.

[21] Williams, L. Casting Curved Shadows on Curved Surfaces. *Computer Graphics SIGGRAPH() 12*, 3 (August 1978), 270–274.