Selective ray tracing for interactive high-quality shadows

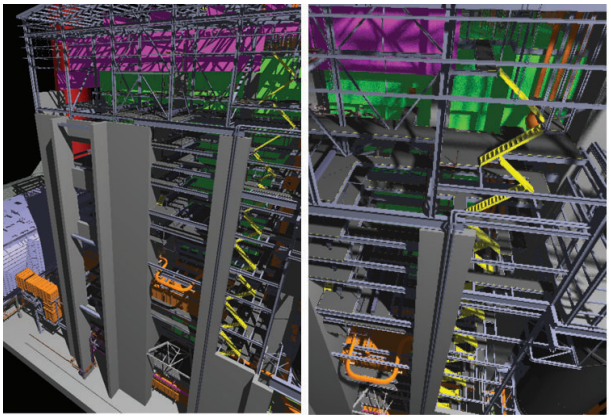

Figure 1: Soft shadows: *Our selective ray tracing algorithm can render the 12.7M triangle Powerplant model at 16 fps with hard shadows (left) and over 2 fps with soft shadows with 16 light samples (right) running on a NVIDIA GTX 280 GPU.*

Abstract

We present novel algorithms to achieve interactive high-quality illumination with hard and soft shadows in complex models. Our approach combines the efficiency of rasterization-based approaches with the accuracy of a ray tracer. We use conservative image-space bounds to identify only a small subset of the pixels in the rasterized image and perform selective ray tracing on those pixels. The algorithm can handle moving light sources as well as dynamic scenes. In practice, our approach is able to generate high quality hard and soft shadows on complex models with millions of triangles at almost interactive rates on current high-end GPUs.

1 Introduction

The state of the art in interactive rendering is constantly moving towards greater physical realism and detail. This is primarily driven by the rapid increase in performance provided by commodity GPUs. The underlying rasterization algorithms utilize parallelism along with high coherence in memory accesses and computations. As a result, current GPUs are able to render models with millions of triangles at interactive rates. However, there are still two main challenges for generating high-quality images using rasterization: first, visibility and occlusion computations used for shadows can suffer from accuracy and aliasing problems. Second, secondary effects such as reflections, refractions and full global illumination that are recursive do not seem to fit well into the rasterization framework.

It is well known that ray tracing offers a simple and elegant solution to many of the problems highlighted above and generates high quality images. Over the last decade, many fast algorithms have been proposed for ray tracing that utilize the parallel capabilities of current CPUs and GPUs. In practice, current ray tracing systems can achieve close to interactive performance for primary rays on a single desktop system. However, for many applications they are still one or two orders magnitude slower than GPU-accelerated rasterization algorithms. As a result, there is limited use of full ray tracers in interactive applications such as computer games or virtual environments.

In this paper, we address the first challenge faced by rasterization algorithms in the area of shadow generation. The main problems in these algorithms is aliasing error and many techniques have been proposed to alleviate these problems. In practice, these approaches either do not scale well to complex, general models composed of millions of triangles or do not fully address all of the sources of error.

Main results: In this paper, we present a *selective ray tracing* algorithm to generate high quality hard and soft shadows on current many-core GPUs. Our approach uses a hybrid strategy that first rasterizes the scene using shadow mapping techniques. Next, we identify the regions or the pixels of the frame that are potentially inaccurate and perform selective ray tracing to compute correct shading information only for those pixels. In practice, the subset of inaccurate pixels is small and our selective approach shoots relatively few rays as compared to a full ray tracer, resulting in a significant speedup. We use conservative techniques to identify all potentially inaccurate pixels (PIP) and present an efficient technique for coupling with a GPU ray tracer to provide the missing information for those pixels. Therefore, the overall rendering algorithm is able to combine the efficiency of the rasterization algorithms with the accuracy of ray tracers.

Overall, our approach provides several key advantages:

- High-quality shadows: In the benchmark scenes our hard and soft shadow generation algorithms result in almost no perspective or projective aliasing artifacts and the image quality is very close to a full ray tracer.
- Efficiency: We use compact hierarchical representations to accelerate the ray intersections and allow ray tracing on GPUs with low memory overhead. As compared to pure rasterization, our hybrid algorithms are 30-70% slower and about 4-10 times faster as compared to full ray tracing.
- Generality: We place no restriction on models or light positions and can handle all data sets that fit into GPU memory.

We demonstrate our implementation on several models ranging from game-like scenes with dynamic objects to massive CAD models with millions of triangles at interactive rates on a high-end GPU.

2 Previous work

There is considerable literature on fast ray tracing and shadow algorithms. Due to space limitations, we only give a brief overview of techniques that address shadow aliasing and real-time ray tracing.

Shadow algorithms: We refer the readers to [Hasenfratz et al. 2003; Lloyd 2007; Laine 2006] for recent surveys on shadow algorithms. At a broad level, prior techniques to alleviate aliasing artifacts using rasterization methods are based on shadow maps [Williams 1978] and shadow volumes [Crow 1977]. The latter is an exact object-space technique but may not scale well on complex models. Most current interactive applications use variants of shadow mapping, but may suffer from projective or perspective aliasing problems. Many practical algorithms have been proposed to alleviate perspective aliasing [Stamminger and Drettakis 2002; Wimmer et al. 2004; Lloyd 2007] as well as projective aliasing [Lefohn et al. 2007]. Other shadow mapping algorithms can eliminate blocking artifacts [Aila and Laine 2004; Johnson et al. 2005; Sintorn et al. 2008] by implementing a rasterizer that can process arbitrary samples on the image plane. Soft shadows can be implemented by sample-based methods such as using averaging visibility from multiple shadow maps to calculate visibility or ray tracing.

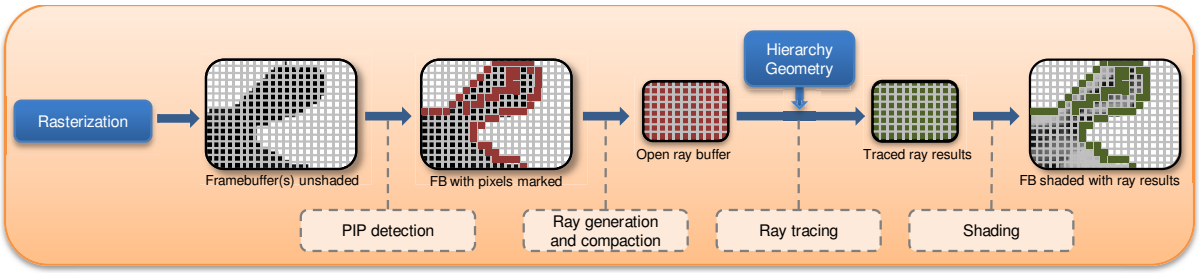

Figure 2: Overview: *Pipeline model of our hybrid rendering algorithm. After GPU-based rasterization is run, the PIP computation detects and marks pixels that need to be ray traced. The ray generation step generates a dense ray list from the sparse buffer of potentially incorrect pixels and then generates one or more rays per pixels as required. A selective ray tracer traces all the rays using the scene hierarchy and then applies the results to the original buffer. Finally, the pixels are shaded based on the ray results.*

Both these methods can be slow, so many approaches have been developed to generate plausible soft shadows with methods such as post-filtering shadow maps or special camera models [Mo et al. 2007], which produce correct results only for simple scenes. More accurate approaches evaluate the light source visibility from the image samples by backprojection [Assarsson and Akenine-Möller 2003; Schwarz and Stamminger 2007; Sintorn et al. 2008; Bavoil et al. 2008] or by generating shadows from environment lighting [Annen et al. 2008].

Hybrid rendering and ray tracing: Several approaches combine rasterization and ray tracing to generate higher quality images [Stamminger et al. 2000] or use GPU rasterization for visibility and shading [Foley and Sugerman 2005; Horn et al. 2007]. Animated scenes can be used with ray hierarchies [Roger et al. 2007]. Other work includes scattering in participating media [Zhou et al. 2008] or adding secondary effects such as refractions [Sun et al. 2008].

3 Selective ray tracing

In this section we give a brief overview of our hybrid rendering algorithm that performs selective ray tracing. The main idea behind selective ray tracing is to only shoot rays corresponding to a small subset of the pixels in the final image in order to accelerate the overall rendering.

Our assumption for the selective ray tracing approach is that we have underlying fast rasterization algorithms that compute the correct result for most of the frame, but may include localized error such as aliasing artifacts in parts of the image. We try to identify these regions of potentially incorrect pixels (PIP) in a conservative manner, since any additionally selected pixels will not change the image whereas missed ones may result in artifacts. As Fig. 2 illustrates, this is a multi-step process that starts with the results of a GPU rasterization algorithm that provides a first approximation to the desired result (e.g. shadows). As a next step, we test the accuracy of each pixel and classify it accordingly, marking each pixel in a buffer, e.g. as pass/fail. The buffer with all marked and unmarked pixels is then passed into the ray tracer where the first step filters out all non-marked pixels and keeps just the potentially incorrect pixels. For each pixel in the PIP set, we shoot one or more rays to compute the correct visibility or shading information for the underlying problem (e.g. shadows.) All rays are stored in a dense list that can be used as input for any data parallel, many-core GPU ray tracer. After the rays have been evaluated, the results are then written back to the original pixel in the PIP set. Back in the rasterizer, a shading kernel is used to compute colors for all the pixels.

In general, this hybrid approach is one of the ways to augment the weaknesses of current GPU-based rasterization algorithms. The most important issue that governs its performance is the fraction of pixels that are in the PIP set. If only a small number of pixels are correct then the overhead of the hybrid approach may outweigh the savings compared to full ray tracing. This also means that the PIP computation algorithm should try to limit the number of pixels it marks while still being conservative. Also, any rasterization algorithm that is used as input for the first step should be efficient so that it is able to compute visibility and shading information for the initial image quickly even on complex models. Otherwise, the typically logarithmic scalability of ray tracing may make a full ray tracing solution faster. The next section describe our algorithms for shadow generation that are based on these characteristics.

4 Shadows

In this section, we present our algorithms for generating high quality shadows based on hybrid rendering and selective ray tracing. We first describe our approach for hard shadows and then extend the approach to soft shadows.

4.1 Hard shadows

Shadow mapping is one of the most widely used algorithms for generating hard shadows for interactive applications. It works on general, complex 3D static and dynamic scenes and maps well onto current GPUs. However, the shadow maps may need high resolution to avoid aliasing artifacts. These errors can be classified into *perspective* and *projective* aliasing [Stamminger and Drettakis 2002]. Perspective error occurs due to the position of the surface with respect to the light and viewpoint, can result in the 'blockiness' of shadows in the algorithm. Projective error stems from the orientation of the receiver to light and viewpoint. Geometric errors from under-sampling can also include missing contributions from objects that are too small or thin in light space, e.g. surfaces that are oriented close to coplanar with the view direction. Artifacts from this error result in missing and interrupted shadows for these objects. Finally, shadow map self-shadowing error occurs from inaccuracies in the depth values computed in the light view and the camera view. It stems both from depth buffer precision (numerical error) as well as orientation of the surface (geometric error). We consider this mainly an artifact that can be minimized by increasing depth precision and using slope-dependent bias and do not address it directly in our algorithm.

Pixel classification: We now discuss our method for estimating the set of pixels in the image that can potentially be incorrect (PIP) due to the error described above. We first note that most of the error in shadow mapping appears at shadow boundaries while the shadow interiors (as well as the interiors of lit regions) tend to be accurate. Thus, we can assume that when we look up the corresponding sample in the shadow map for a given image sample, it should not be considered accurate if it is adjacent to an edge in

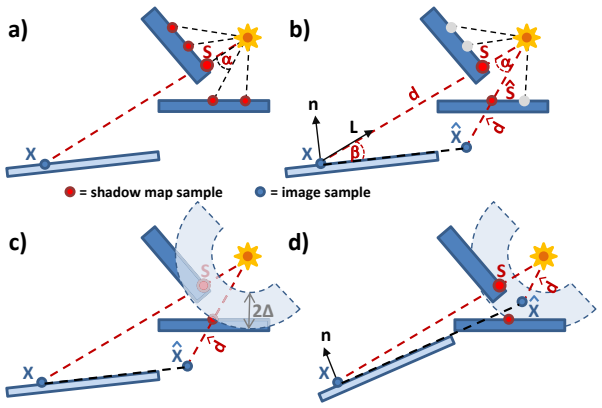

Figure 3: PIP computation: *a) For a given image sample **X** we project back to the shadow map and find the corresponding sample **S**. We then test the depths of the surrounding shadow map samples and select the one with the maximum difference Δ in depth value to **S** and label it as $\hat{\mathbf{S}}$. b) We now determine whether the surface at **X** can be affected by an edge at **S** and $\hat{\mathbf{S}}$ by finding the closest point $\hat{\mathbf{X}}$ on the surface within the angle α of one shadow map pixel and find its depth $\hat{d}$. c) If $\hat{d}$ is within the shaded region of depth Δ around **S** or on the other side of the region as seen from **X**, then it needs to be ray traced. In this case, the pixel can be classified as shadowed. d) Counter-example: $\hat{\mathbf{X}}$ is in the region and thus the pixel is ray traced.*

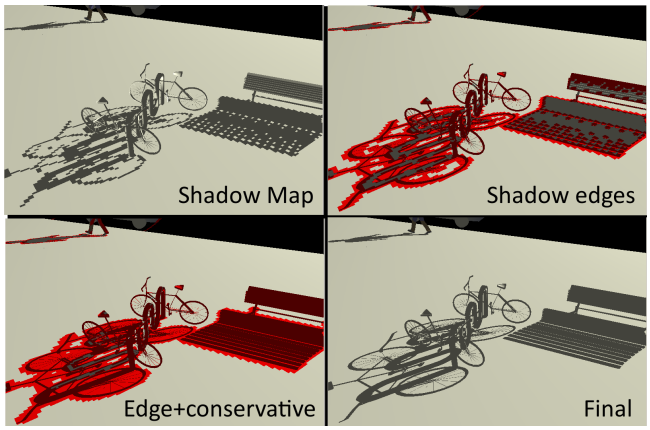

Figure 4: Detecting shadow artifacts: *Shadow on City model. Top left: shadows with shadow mapping at 2048^2 resolution. Top Right: pixels marked for ray tracing in red. Bottom left: pixels marked by conservative rendering. Bottom right: final result. The image is identical to the fully ray-traced result.*

the shadow map. We therefore modify the standard depth buffer look-up to test the 8 texels around the sample value with depth s in the shadow map and find the maximum absolute difference $\Delta = \max(\|s - s'\|, s' \in \text{depth around S})$ in depth. If Δ exceeds a threshold value, e.g. a fraction of the possible scene depth as determined by z-buffer near and far planes, we assume that there is a shadow edge at this image pixel. In this case, we need to make sure this edge can affect the shadow generation at the current shading sample **X** that is at distance d from the light source (see Fig. 4.1 for illustration.) This is to prevent that a relatively small shadow discontinuity at one side of the model does still affect pixels far away. To achieve this, we find the closest distance to the light $\hat{d}$ that the receiver surface can reach within the angle α of one texel of the shadow map. Intuitively, the closer to parallel the receiver surface is to the light direction, the larger the difference between $\hat{d}$ and d becomes. If $\hat{d}$ overlaps the interval $[s-\Delta, s+\Delta]$ or d and $\hat{d}$ are on different sides of **S** then we conservatively mark the pixel as part of the PIP set.

The actual computation of $\hat{d}$ for a local point light source follows from trigonometry such that $\hat{d} = d \sin\beta / \sin\alpha + \beta$. Given the normal vector **n** and normalized light direction **L**, then this is easily computed by using $\sin\beta = \mathbf{n} \cdot \mathbf{L}$. A similar calculation for directional lights is relatively straightforward. Note that is also possible to precompute Δ for the shadow map and store it for each pixel in addition to the depth value. This may be useful if the light source is mostly static since it reduces the memory bandwidth needed during the lighting pass of the rasterization algorithm.

One case that the method above does not detect is the geometric aliasing problem when a primitive is too small (e.g. a thin wire) as seen from the light view and thus not even drawn during scanline rendering. Some of the pixels that are actually covered by the object may not get rasterized and thus not detected during edge filtering. This can cause missing shadows regions in the actual image. Our solution is to identify these small objects during rasterization from the light view and employ conservative rendering techniques to assure that they are actually part of the PIP set. We use geometry shaders to implement the conservative triangle rendering method described in [Hasselgren et al. 2005] and modified by [Sintorn et al. 2008]. In essence, each triangle is transformed into a polygon with 6 corners by extruding at the original vertices. The new extruded primitive is then guaranteed to cover the center point of each pixel that it touches. Since extending all the triangles in the scene could be extremely costly, the shader also tests the area and aspect ratio of each triangle in image space and only uses conservative rendering for those that are thin and thus likely to be missed. At the same time, this technique automatically avoids very small but regular triangles (e.g. as in scanned models) where conservative rendering is not needed. Note that an even more efficient culling method would be needed to detect whether the triangle also has a silhouette edge, but we found that this adds more constraints on the rendering pipeline. e.g. by having to provide adjacency information. Figure 4 illustrates the effect of our conservative rendering approach on the very thin spokes of the bicycle wheels in the model, as well as the bench.

4.2 Soft shadows

We now describe an extension of the approach presented above that can also be used to render high quality soft shadows. Ray tracing approaches stochastically generate a number of samples on the area light source for each hit point, evaluate their visibility using shadow rays and then average the results to find an estimate of how much of the light source is visible from the hit point. We can emulate this approach by rendering multiple shadow maps, each from one of the light source samples and then averaging the visibility results as well. However, this approach has two disadvantages: first, it is equivalent to testing the same light samples at each pixel which cause visible sampling artifacts. Second, and more importantly, a high number of samples are needed to avoid high frequency aliasing error, which means that the scene has to be scanline rendered numerous times from multiple viewpoints on the light to get a high-quality result. This could be too slow for interactive applications.

Our approach for soft shadows modifies the naive approach described above to reduce the number of shadow maps needed. The main idea is that it is sufficient if we can accurately classify the pixels in terms of whether they are in full shadow (umbra), fully lit or in the penumbra region. The exact computation of umbra and penumbra region for area light sources are expensive. In terms of sampled-based approximation, a high number of light samples may be needed to correctly evaluate the penumbra region. We use

this classification of penumbra-region pixels only to mark them for evaluation with a high number of samples using ray tracing. The problem then simplifies to finding the pixels in penumbra regions. We generate a fixed number of light samples (e.g. four to six) and then use the shadow mapping approach above to compute the lighting information for all pixels. If the results from all these fixed light samples match then we assume the pixel is fully lit or fully shadowed. All other pixels as well as those marked by the previous shadow mapping step are sent to the ray tracer for stochastic visibility evaluation such that the penumbra regions can be lit with a higher number of samples per pixel.

The advantage of this approach is that it can produce a correct sampling for the penumbra regions via ray tracing for general models where methods using back-projections are more limited. However, since we only use four initial samples to estimate penumbra status for a pixel, it is possible that pixels are misclassified as fully occluded when the four samples hit separate objects but miss the unoccluded parts. This mainly occurs when the sample density of the shadow maps is far lower than locally needed.

5 Implementation and Results

In this section, we describe our implementation and highlight the performance on many complex models.

5.1 Implementation

We have implemented the algorithms described above on a NVIDIA GPU. We use OpenGL with Cg as the rendering interface and CUDA for general-purpose programming. In practice, there are many possible variations to implement our approach. First of all, the selective ray tracing model is very similar to a manual implementation of allowing a `trace()` call in the original shader, but in a more limited form. In particular, the shader cannot explicitly generate rays, but only mark pixels such that they are then generated by the ray tracer itself. In practice, it is harder to store an arbitrary number of rays per pixel in a rasterization framework compared to a general purpose GPU programming environment because of limitations in the rasterization interface. Another issue for an implementation is when to perform the shading for the pixels. Our implementation assumes that shading for all pixels is deferred until the very last step such that ray traced pixels are shaded in the same way as the rasterized samples. As an alternative, the shading could also be performed at the end of the ray tracer and just written to the frame buffer instead. While this might be easier to implement, it would require duplication of all shading kernels both in the pixel shading as well as GPU computation language.

In our system, we make the simplifying assumption that the rays are evaluated for visibility only, which sidesteps the problem of how to handle shading inside the ray tracer. Our ray tracer uses a BVH built on the GPU as the acceleration structure with a simple stack-based traversal similar to the one described in [Günther et al. 2007]. This also allows us to handle dynamic scenes by rebuilding the hierarchy each frame. A compaction step groups all the rays generated for marked pixels into a dense buffer that is then distributed into small packets, each of which is handled independently. For rendering massive models, memory for storing the geometry and hierarchy on the GPU becomes an issue. We use a variant of the ReduceM representation [Lauterbach et al. 2008], but we modify the strip representation such that it is possible to also directly rasterize strips via OpenGL in order to use the same representation both for rasterization and ray tracing. In addition, we also use the top levels of the scene BVH for view frustum culling and occlusion culling based on occlusion queries both from the light as well as the camera view. These culling methods accelerate the performance of the rasterization algorithm. In our current implementation,view frustum culling is currently performed on the CPU, but could also be easily implemented in a CUDA algorithm. For soft shadows, we use a simple stratified sampling scheme for the shadow ray samples that is computed for each pixel with a simple random number generator inside the ray generation kernel. We base the random seed on sample location which allows us to compare our results for selective and full ray tracing without having to isolate variance in the estimate.

Benchmarks	Tris	Hard			Soft		
		SM	SRT	FRT	SM	SRT	FRT
City	58K	256	103 (3%)	21	200	19 (9.8%)	3.7
Sibenik	82K	150	66 (4%)	13	47	7.3 (6%)	0.64
Buddha	1M	42	29 (1.4%)	8	34	9 (7.7%)	2.5
Powerplant	12M	25	16 (7.1%)	4	4.4	2.1 (11%)	0.8

Figure 5: Performance: *Performance of our selective ray tracing (SRT) approach on our benchmark models, compared to the simple shadow mapping algorithm (SM) with one point light for hard and 4 point lights for hard shadows, as well as a fully ray traced solution (FRT). The percentages show the fraction of pixels marked as incorrect. All numbers are frames per second (FPS) at* 1024^2 *screen resolution.*

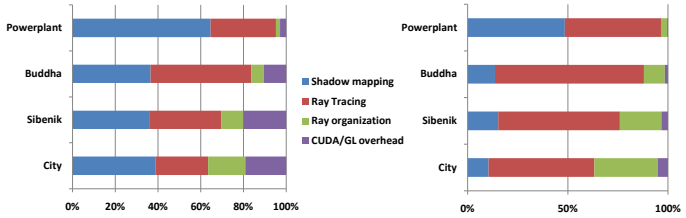

Figure 6: Timings: *Time spent in different parts of the algorithm for hard (left) and soft shadows (right). Rasterization includes both shadow map generation, frame buffer rendering and shading. Ray tracing includes time spent in the ray tracing kernel only, while ray generation is the time for generating the compact ray buffer from sparse frame buffer and writing back at the end. CUDA/GL times represent the cost for buffer transfers and similar as the overhead of switching between OpenGL and CUDA.*

5.2 Results

We now present results from our implementation running on a Intel Core2 Duo system at 2.83 GHz using a NVIDIA GTX 280 GPU running Windows XP. Table 5 summarizes the timings for hard and soft shadows at 1024×1024 screen resolution, including comparison timings for GPU algorithm only, selective ray tracing and full ray tracing. The data for selective ray tracing also shows the percentage of pixels in the PIP set. We selected a wide range of benchmark models from relatively low-complexity game-like environments to high-complex scanned, CAD and architectural models to highlight the scalability of the algorithm. For shadows, all timings are for a moving light source, i.e. the shadow map is generated per frame, and soft shadows are generated by using 16 samples/pixel and four original samples for the penumbra estimation Note that our algorithm is typically 5 times faster than full ray tracing.

We present a detailed analysis of the timing breakdown in selective ray tracing (see Table 6) for several of our benchmark scenes. There is a relatively constant overhead associated with PIP detection, ray compaction and parts of the implementation such as the overhead of CUDA/OpenGL communication in the current programming environment. Note that the actual tracing of the selected ray samples only makes up a fraction of the time spent. While we do not explicitly show the overhead in rasterization introduced by conservative rendering in the graph above, we have found that in practice it slows

down the rasterization step by about 10% since only a relatively small set of primitives are rendered conservatively.

6 Analysis and Comparison

Comparison: A very important aspect for evaluation of our algorithm is the difference in image quality compared to the full ray tracing solution. We present a detailed comparison for several benchmark scenes in the appendix that show the original image generated with shadow mapping at 1024^2 resolution, selective ray tracing results and reference full ray tracing (with a difference image). In practice, for hard shadows we have observed that our algorithm yields images that are almost error free and virtually identical to fully ray traced results for all our benchmark scenes, with differences arising from small biasing errors. Unfortunately, we cannot guarantee full correctness since some features such as very small holes inside a solid object could theoretically be missed due to the regular sampling in light space (i.e. geometric aliasing errors). However, despite the theoretical problem we have not run into this case on any of our benchmark models even for very complex geometric situations and any remaining differences stem from self-shadowing artifacts.

It is hard to directly compare the quality of our results with only rasterization-based approaches. The fastest rasterization methods for hard shadows based on warping and partitioning map well to current GPUs [Stamminger and Drettakis 2002; Wimmer et al. 2004]. However, they only alleviate perspective aliasing and the performance can vary based on the relative position of the light source with respect to the viewpoint. Many techniques to handle projective aliasing [Lefohn et al. 2007] and alias-free shadow maps [Aila and Laine 2004; Johnson et al. 2005] can be implemented on current GPUs. These approaches can generate high quality shadows, but it is not clear whether they can scale well to massive models. For soft shadows, most of the accurate methods may not be able to handle complex models at interactive rates on current processors. Recently, Sintorn et al. [2008] presented a soft shadow algorithm that can handle models with at most tens of thousands of triangles at interactive rates. It uses a more accurate method for penumbra computation, but its scalability on large models with moving light sources is not clear.

Performance analysis: On current highly-parallel architectures, algorithms should avoid being limited by memory bandwidth rather than computation. This is mainly because the growth rate for compute power far exceeds that of memory bandwidth. Previous work has already shown that ray tracing using hierarchies on GPUs is in fact limited by memory bandwidth. In contrast, the streaming model of computation used in the rasterization pipeline has been shown to be very successful in this regard with memory bandwidth being mostly used for depth and frame buffer accesses. We analyze our selective ray tracing approach compared to the full ray tracing solution to investigate its efficiency.

We look at the memory bandwidth requirements when running both selective and full ray tracing for two of our benchmark models, in particular the memory bandwidth used by the actual ray tracing kernel. Since current GPU architectures have limited cache sizes, i.e. only a texture cache, such analysis is simpler than for CPUs. We implemented a simple software simulator that emulates the behavior of the memory unit for global memory accesses in CUDA in device emulation mode running on the host CPU. Care has to be taken mainly to correctly account for the behavior of the memory unit in combining data parallel accesses to contiguous memory. Our results (see Fig. 7) indicate that selective ray tracing consistently only uses about an order of magnitude less memory bandwidth per frame than full ray tracing. The bandwidth for selective ray tracing is still slightly higher than expected from the number of rays compared to full ray tracing. This is due to the fact that the ray groups in selective ray tracing are more incoherent and thus will access more memory. In order to perform a complete analysis, we also need to compute the memory bandwidth needed by the rasterizer for scanline rendering of the shadow map. However, this is not possible for hardware accelerated rasterization (i.e. current GPUs) since the implementation details are not publicly available. However, we provide an estimate for the bandwidth used for rasterizing the shadow map by multiplying the peak bandwidth on the GPU by the time taken for rasterization, thus providing us a conservative upper bound. We list the summed memory bandwidth for shadow mapping plus selective ray tracing in the last column. Note that the combined bandwidth for this is still significantly lower than full ray tracing.

Model	Geometry	Full RT	SRT	SRT+SM	SRT % Rays
City	2 MB	1066 MB	113 MB	222 MB	5
Sibenik	2.2 MB	2280 MB	224 MB	859 MB	4
Buddha	27 MB	3148 MB	601 MB	1144 MB	12

Figure 7: Memory bandwidth: *Simulated memory bandwidth requirements for rendering one frame at 512^2 image resolution with selective (SRT) and full ray tracing (FRT) on several models (total storage for geometry and BVH is given in second column to show the total working set size.) The last column shows the time for SR plus a very conservative estimation of bandwidth needed by scanline rendering of the shadow map.*

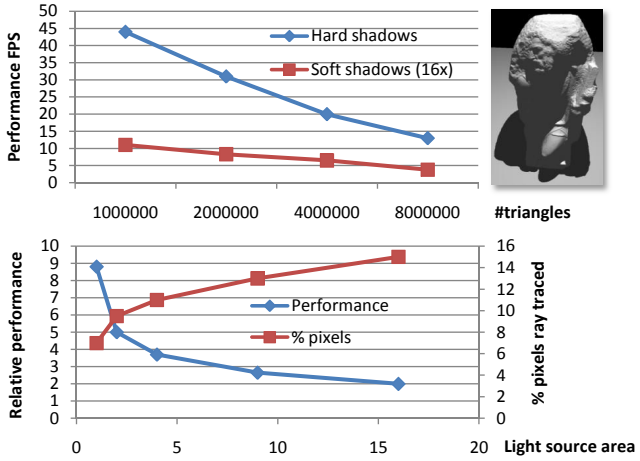

Figure 8: Scalability: *Top: Performance vs. model complexity on several simplification levels of the St. Matthew model. Bottom: Performance vs. light source size on Buddha model.*

We also demonstrate the scalability of our approach with model complexity. To eliminate bias introduced by different model characteristics, we use different simplification levels of the same model and then compare the time used for selective ray tracing for each of the levels. Our results in Fig. 8 show that we can achieve sub-linear scalability with complexity thanks to the use of ray tracing and occlusion culling techniques. We also look at the performance implications of increasing the area of the light source for soft shadow rendering (see Fig. 8 bottom). Similar to other approaches our performance decreases significantly with larger light sources mostly due to more of the model being in penumbra regions and subsequently being ray traced.

Limitations: Overall, the main determining factor for our algorithm is the size of the PIP set. If the set is too large, then our al-

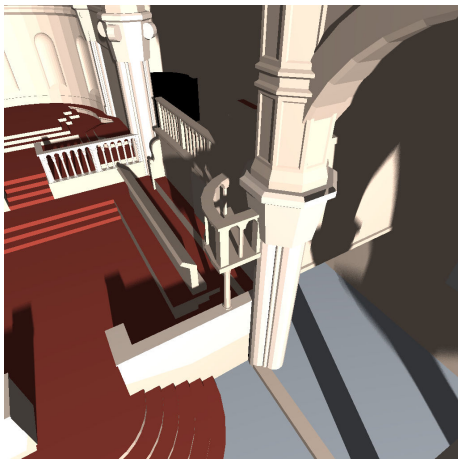

Figure 9: Benchmarks: *Left: soft shadows using 16 light samples in Sibenik cathedral model, running at 7 fps at* 1024^2 *resolution.*

gorithm cannot achieve a significant speedup over full ray tracing; however, at worst it can also only be slightly slower to the extent of shadow mapping overhead. In addition, the rays generated by selective ray tracing may exhibit less ray coherence than in full ray tracing which means that tracing these rays will be slightly more expensive on a per-ray metric. Since the rays still can access any part of the model, we also can only render models that fit into GPU memory and need to store an additional ray tracing hierarchy as well as update or rebuild it for deformable models. Our approach may also still carry over some of the geometric errors from rasterization such as depth buffer errors and resulting shadow map bias. The accuracy of our soft shadow algorithm is governed by the underlying sampling algorithm.

7 Conclusion and future work

We presented new algorithms for selective ray tracing that augment existing fast rasterization approaches for shadows. In practice, they can generate high-quality shadows, similar to full ray tracing and are about 5 times faster on current GPUs. Our approach is robust and scales in terms of handling complex models. We also analyzed its bandwidth requirements compared to full ray tracing and demonstrate that our approach maps well to current architectural trends.

There are many avenues for improvement. For one, the shadow accuracy detection could be made less conservative by using a better edge representation in shadow maps, such as silhouette maps [Sen et al. 2003]. However, in each case the trade-off between a more expensive rasterization shadow step and the resulting ray workload will need to be carefully evaluated. Our approach should be directly applicable to ambient occlusion, as screen-space ambient occlusion approaches can generate a good approximation of the local indirect illumination using the depth buffer. In general, our selective ray tracing approach could also be used to evaluate secondary effects such as reflections and refractions that require shading, but this would require more flexibility in the rendering pipeline than current GPUs allow. Upcoming processors such as Larrabee may provide this functionality. An important aspect in the pipeline is the implementation of the actual ray tracing algorithm. The traditional ray tracing paradigm using accelerating structures means that all the geometry needs to be stored for random access during ray tracing, which may be incompatible with the GPU streaming model. One interesting solution here is to use ray hierarchies, i.e. a hierarchy that is built on top of the total set of rays and is intersected with the scene.

References

AILA, T., AND LAINE, S. 2004. Alias-free shadow maps. In *Proceedings of Eurographics Symposium on Rendering 2004*, Eurographics Association, 161–166.

ANNEN, T., DONG, Z., MERTENS, T., BEKAERT, P., SEIDEL, H.-P., AND KAUTZ, J. 2008. Real-time, all-frequency shadows in dynamic scenes. *ACM Trans. Graph. 27*, 3, 1–8.

ASSARSSON, U., AND AKENINE-MÖLLER, T. 2003. A geometry-based soft shadow algorithm using graphics hardware. *ACM Transactions on Graphics 22*, 3, 511–520.

BAVOIL, L., CALLAHAN, S. P., AND SILVA, C. T. 2008. Robust soft shadow mapping with backprojection and depth peeling. *Journal of Graphics Tools 13(1)*.

CROW, F. C. 1977. Shadow algorithms for computer graphics. *ACM Computer Graphics 11*, 3, 242–248.

FOLEY, T., AND SUGERMAN, J. 2005. KD-tree acceleration structures for a GPU raytracer. In *Proc. ACM SIGGRAPH/EG Conf. on Graphics Hardware*, 15–22.

GÜNTHER, J., POPOV, S., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In *Proc. IEEE/EG Symposium on Interactive Ray Tracing 2007*, 113–118.

HASENFRATZ, J.-M., LAPIERRE, M., HOLZSCHUCH, N., AND SILLION, F. 2003. A survey of real-time soft shadows algorithms. *Computer Graphics Forum 22*, 4 (dec), 753–774.

HASSELGREN, J., AKENINE-MÖLLER, T., AND OHLSSON, L. 2005. Conservative rasterization on the gpu. *GPU Gems 2*, 677–690.

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-d tree GPU raytracing. In *Proc. I3D '07*, 167–174.

JOHNSON, G. S., LEE, J., BURNS, C. A., AND MARK, W. R. 2005. The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. Graph. 24*, 4, 1462–1482.

LAINE, S. 2006. *Efficient Physically-Based Shadow Algorithms*. PhD thesis, Helsinki University of Technology.

LAUTERBACH, C., YOON, S.-E., TANG, M., AND MANOCHA, D. 2008. ReduceM: Interactive and memory efficient ray tracing of large models. *Computer Graphics Forum 27*, 4, 1313–1321.

LEFOHN, A. E., SENGUPTA, S., AND OWENS, J. D. 2007. Resolution-matched shadow maps. *ACM Trans. Graph. 26*, 4, 20.

LLOYD, B. 2007. *Logarithmic Perspective Shadow Maps*. PhD thesis, University of North Carolina at Chapel Hill.

MO, Q., POPESCU, V., AND WYMAN, C. 2007. The soft shadow occlusion camera. *Proc. Pacific Graphics 2007*, 189–198.

ROGER, D., ASSARSSON, U., AND HOLZSCHUCH, N. 2007. Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU. In *Rendering Techniques 2007 (Proc. Eurographics Symposium on Rendering)*, 99–110.

SCHWARZ, M., AND STAMMINGER, M. 2007. Bitmask soft shadows. *Comput. Graph. Forum 26*, 3, 515–524.

SEN, P., CAMMARANO, M., AND HANRAHAN, P. 2003. Shadow silhouette maps. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003) 22*, 3 (July), 521–526.

SINTORN, E., EISEMANN, E., AND ASSARSSON, U. 2008. Sample-based visibility for soft shadows using alias-free shadow maps. *Computer Graphics Forum (Proc. EGSR '07) 27*, 4, 1285–1292.

STAMMINGER, M., AND DRETTAKIS, G. 2002. Perspective shadow maps. In *Proc. SIGGRAPH '02*, 557–562.

STAMMINGER, M., HABER, J., SCHIRMACHER, H., AND SEIDEL, H.-P. 2000. Walkthroughs with corrective textures. *Proc. Eurographics Workshop on Rendering '00*, 377–388.

SUN, X., ZHOU, K., STOLLNITZ, E., SHI, J., AND GUO, B. 2008. Interactive relighting of dynamic refractive objects. *ACM Trans. Graph. 27*, 3, 1–9.

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, vol. 12, 270–274.

WIMMER, M., SCHERZER, D., AND PURGATHOFER, W. 2004. Light space perspective shadow maps. In *Proc. of the Eurographics Symposium on Rendering*, 143–152.

ZHOU, K., REN, Z., LIN, S., BAO, H., GUO, B., AND SHUM, H.-Y. 2008. Real-time smoke rendering using compensated ray marching. *Proc. of SIGGRAPH*, 1–12.