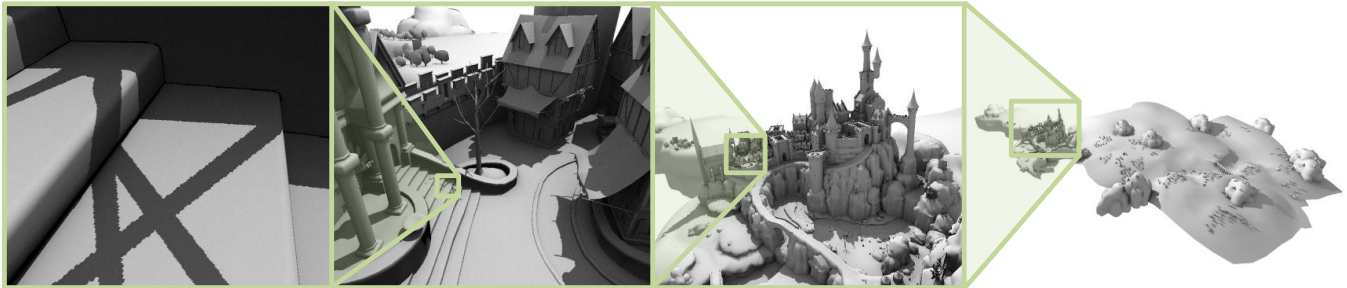# High Resolution Sparse Voxel DAGs

Viktor Kämpe   Erik Sintorn   Ulf Assarsson
Chalmers University of Technology

Figure 1: *The* EPICCITADEL *scene voxelized to a* $128K^3$ *(131 072*^{3}*) resolution and stored as a Sparse Voxel DAG. Total voxel count is 19 billion, which requires 945MB of GPU memory. A sparse voxel octree would require 5.1GB without counting pointers. Primary shading is from triangle rasterization, while ambient occlusion and shadows are raytraced in the sparse voxel DAG at 170 MRays/sec and 240 MRays/sec respectively, on an NVIDIA GTX680.*

Abstract

We show that a binary voxel grid can be represented orders of magnitude more efficiently than using a sparse voxel octree (SVO) by generalising the tree to a directed acyclic graph (DAG). While the SVO allows for efficient encoding of empty regions of space, the DAG additionally allows for efficient encoding of *identical* regions of space, as nodes are allowed to share pointers to identical subtrees. We present an efficient bottom-up algorithm that reduces an SVO to a minimal DAG, which can be applied even in cases where the complete SVO would not fit in memory. In all tested scenes, even the highly irregular ones, the number of nodes is reduced by one to three orders of magnitude. While the DAG requires more pointers per node, the memory cost for these is quickly amortized and the memory consumption of the DAG is considerably smaller, even when compared to an ideal SVO without pointers. Meanwhile, our sparse voxel DAG requires no decompression and can be traversed very efficiently. We demonstrate this by ray tracing hard and soft shadows, ambient occlusion, and primary rays in extremely high resolution DAGs at speeds that are on par with, or even faster than, state-of-the-art voxel and triangle GPU ray tracing.

CR Categories: I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Display Algorithms I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

Keywords: octree, sparse, directed acyclic graph, geometry, GPU, ray tracing

Links: ◆DL ⬇PDF

1 Introduction

The standard approach to rendering in real-time computer graphics is to rasterize a stream of triangles and evaluate primary visibility using a depth-buffer. This technique requires no acceleration structure for the geometry and has proved well suited for hardware acceleration. There is currently an increased interest in the video-game industry in evaluating secondary rays for effects such as reflections, shadows, and indirect illumination. Due to the incoherence of these secondary rays, most such algorithms require a secondary scene representation in which ray tracing can be accelerated. Since GPU memory is limited, it is important that these data structures can be kept within a strict memory budget.

Recent work has shown extremely fast ray tracing of triangle-meshes, with pre-built acceleration structures, on modern GPUs [Aila et al. 2012]. However, the acceleration structure has to be resident in GPU RAM for efficient access. This is particularly problematic when triangle meshes are augmented with displacement maps as they might become infeasibly large when fully tessellated to millions of polygons. This has triggered a search for simpler scene representations that can provide a sufficient approximation with a reasonable memory footprint.

Recently, sparse voxel octrees (SVO) have started to show promise as a secondary scene representation, since they provide an implicit LOD mechanism and can be efficiently ray traced. Both construction and traversal speed has reached impressive heights, which has enabled ray traced effects on top of a rasterized image in real time [Crassin et al. 2011]. At high resolutions, SVOs are still much too memory expensive, however, and therefore their applicability has been limited to effects such as rough reflections and ambient occlusion where a low resolution is less noticeable, except at contact. Equally, or even more importantly, scene sizes are also significantly restricted, often prohibiting practical usage. The data structure described in this paper allows for extremely high resolutions enabling us to improve image quality, decrease discretization artifacts, and explore high-frequency effects like sharp shadows, all in very large scenes (see e.g. Figure 1).

In this paper, we present an efficient technique for reducing the size of a sparse voxel octree. We search the tree for common subtrees and only reference unique instances. This transforms the tree structure

into a directed acyclic graph (DAG), resulting in a reduction of nodes without loss of information. We show that this simple modification of the data structure can dramatically reduce memory demands in real-world scenes and that it scales even better with increasing resolutions while remaining competitive with state of the art ray tracing methods.

Our reduction technique is based on the assumption that the original SVO contains a large amount of redundant subtrees, and we show that this is the case for all scenes that we have tried, when considering only geometry information at moderate to high resolutions. We do not attempt to compress material or reflectance properties of voxels in this paper, and thus, we focus mainly on using the structure for visibility queries. We have implemented and evaluated algorithms for calculating hard shadows, soft shadows, and ambient occlusion by ray tracing in our DAGs and show that it is by no means more difficult or expensive than to ray trace in a straight SVO.

2 Previous Work

Sparse Voxel Octrees   Description of geometry is a broad field and we will focus on recent advancements of using sparse voxel trees, in particular octrees. There are methods to use the voxels as a level of detail primitive to replace the base primitive when viewed from far away, e.g with the base primitives in the leaf nodes being triangles [Gobbetti and Marton 2005] or points [Elseberg et al. 2012]. The previous work most related to our geometric description are those that also use voxels as the base primitive. We do not consider methods that use octrees purely as an acceleration structure to cull primitives.

Laine and Karras [2010a] present *efficient sparse voxel octrees* (ESVO) for ray tracing primary visibility. They avoid building the octree to its maximal depth by providing each cubical voxel with a contour. If the contours are determined to be a good approximation to the original geometry, the subdivision to deeper levels is stopped. The pruning of children allows them to save memory in scenes with many flat surfaces, but in certain tricky scenes, where the scene does not resemble flat surfaces, the result is instead problematic stitching of contours without memory savings. The highest octree resolutions, possible to fit into 4GB memory, ranged from $1\mathrm{K}^3$ to $32\mathrm{K}^3$ for the tested scenes. Approximately 40% of the memory consumption was due to geometry encoding and the rest due to material, i.e. color and normal.

Crassin et al. [2011] compute ambient occlusion and indirect lighting by cone tracing in a sparse voxel octree. The cone tracing is performed by taking steps along the cone axis, with a step length corresponding to the cone radius, and accumulating quadrilinearly filtered voxel data. The interpolation requires duplication of data to neighbouring voxels, which result in a memory consumption of nearly 1024 MB for a 512^3 sparse voxel octree with materials.

Crassin et al. [2009] mention voxel octree instancing and use the Sierpinski sponge fractal to illustrate the authoring possibilities of octrees but do not consider an algorithmic conversion from a tree to these graphs.

Schnabel and Klein [2006] focus on decreasing the memory consumption of a point cloud of geometry discretized into an octree with $4\mathrm{K}^3$ resolution. By sorting the tree in an implicit order, e.g. breadth-first order, they can store it without pointers between nodes. The implicit order cannot be efficiently traversed but is suitable to reduce the size in streaming or storage.

Common subtree merging   Highly related to our algorithm is the work by Webber and Dillencourt [1989], where quadtrees representing binary cartographic images are compressed by merging common subtrees. The authors show a significant decrease (up to $10\times$) in the number of nodes required to represent a 512×512 binary image. Additionally, they derive an upper bound on the number of nodes of a compressed tree for arbitrary images as well as images containing a single straight line, a half-space or a convex object. Parker et al. [2003] extend this to three dimensions, in order to compress voxel data instead. However, in their approach, the voxel content (e.g. surface properties) is not decoupled from geometry, and thus they report successful compression only of axis-aligned and highly regular data sets (flat electrical circuits). Also related is the work by Parsons [1986], who suggests a way of representing straight lines (in 2D with a rational slope) as cyclic quadgraphs.

Alternative approaches to compressing or compactly representing trees, and applications thereof, are presented and surveyed in a paper by Katajainen et al. [1990].

Shadows   GPU accelerated shadow rendering is a well researched subject, and we refer the reader to the book by Eisemann et al. [2011] for a complete survey. We will only attempt to briefly overview recent and relevant work concerning real-time or interactive shadows.

There is a large class of *image based* methods originating from the seminal work by Williams [1978]. The occluders are somehow recorded as seen from the light source in an image (a *shadow map*) in a first pass, and this image is later used to determine light visibility for fragments when the scene is rendered from the camera's viewpoint. While these algorithms suffer from an inherent problem with aliasing, several methods exist to alleviate artifacts by blurring the shadow edges [Reeves et al. 1987; Annen et al. 2007; Donnelly and Lauritzen 2006]. Additionally, several researchers have attempted to mimic soft shadows from area light sources by adapting the filter kernel size during lookup into the shadow map [Fernando 2005; Annen et al. 2008; Dong and Yang 2010]. These techniques are extremely fast but, generally, cannot capture the true visibility function from an area light source.

Another class of algorithms are the *geometry based* methods, based on the idea of generating *shadow volumes* [Crow 1977] that enclose shadowed space. For point-light sources, these algorithms can often produce correct light visibility per view sample with hardware acceleration and at high frame rates [Heidmann 1991], and recent work by Sintorn et al. [2011] present a very robust variation of the idea that performs well for arbitrary input geometry. Area light sources can be handled using Soft Shadow Volumes [Assarsson and Akenine-Möller 2003], where the visibility of view samples that lie in the penumbra is integrated in screen space. This algorithm has been extended to produce correctly sampled light visibility with very fast execution times [Laine et al. 2005].

A third approach that has been considered for hard shadows is the idea of *view-sample mapping*. Here, the view samples are projected into light space and are used as the sample positions in a subsequent shadow map rendering pass [Aila and Laine 2004; Johnson et al. 2005]. The same idea has been applied to soft shadows, where the shadow casting geometry is enlarged to cover all potentially occluded samples [Sintorn et al. 2008; Johnson et al. 2009]. This approach can lower the fill-rate demands, compared to shadow-volume based methods, but rendering speeds are still highly dependent on the amount of shadow casting geometry that needs to be processed.

Finally, recent advances in real-time ray tracing performance has opened up for alternative approaches. Given a scene with a precomputed acceleration structure, casting a few shadow rays per pixel is now within reach for interactive applications. A number of recent papers discuss the possibility of efficiently sharing samples between pixels [Lehtinen et al. 2011; Egan et al. 2011; Hachisuka et al. 2008]. These methods significantly reduce noise in images with very low
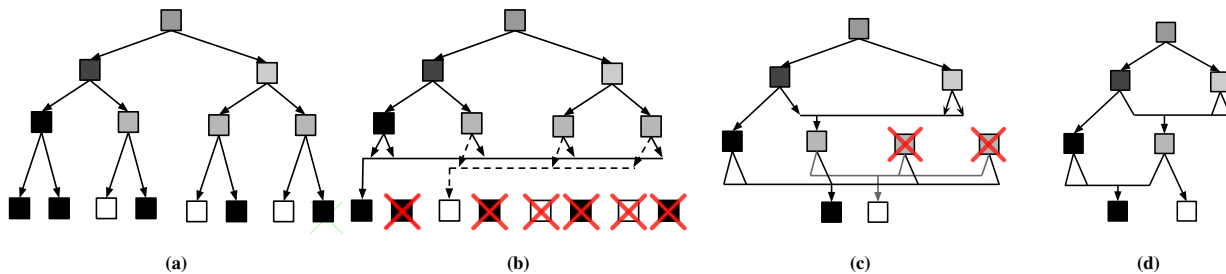
Figure 2: *Reducing a sparse voxel tree, illustrated using a binary tree, instead of an octree, for clarity. a) The original tree. b) Non-unique leaves are reduced. c) Now, there are non-unique nodes in the level above the leaves. These are reduced, creating non-unique nodes in the level above this. d) The process proceeds until we obtain our final directed acyclic graph.*

sampling rates, but the reconstruction algorithms still take too long to be used in an interactive framework. Based on the same frequency analysis reasoning, a much faster reconstruction algorithm is presented by Mehta et al. [2012]. These algorithms are orthogonal to the work presented in our paper and could potentially be used to extend our approach.

Ambient Occlusion   Ambient occlusion (AO) is a well known and frequently used approximation to global illumination for diffuse surfaces. A more comprehensive review of algorithms to compute AO can be found in the survey by Méndez-Feliu and Mateu Sbert [2009]. The cheapest and perhaps most used approach to achieving AO in real-time applications today is to perform all calculations in screen space, based on recorded depth values (the original idea is described by e.g. Akenine-Möller et al. [2008]). This is an extremely fast method but the information available in the depth buffer can be insufficient in difficult cases.

An alternative approach is to attach a volume of influence to the object or primitives that occlude the scene. View samples within this volume perform computations or look-ups to evaluate the occlusion caused by the object. There are several variants on this idea where occlusion is either pre-computed into a texture [Kontkanen and Laine 2005; Malmer et al. 2007], or a volume is generated per primitive with occlusion being computed at runtime [McGuire 2010; Laine and Karras 2010b]. In the first approach, the resolution of the pre-computed map will limit the frequency of occlusion effects, and in the second, rendering performance will drop severely if too many or too large occlusion volumes have to be considered. Laine and Karras [2010b] suggest an alternative method, where the hemisphere (bounded by occlusion radius) of each receiver point is instead tested against the scene BVH. For each potentially occluding triangle, all AO rays are then simultaneously tested, which can dramatically improve performance, especially when using many rays and a small maximum occlusion radius.

Ambient occlusion computations, like shadow computations, can benefit greatly from sharing rays between pixels, and there are several papers suggesting reconstruction algorithms for sparsely sampled ray traced images [Egan et al. 2011].

3   Reducing a Sparse Voxel Tree to a DAG

In this section, we will explain how a sparse voxel octree encodes geometry and how we can transform it into a DAG using a bottom-up algorithm. We begin with a quick recap to motivate our work and to establish the terminology to be used throughout the paper.

Terminology   Consider an N^3 voxel grid as a scene representation, where each cell can be represented by a bit: 0 if empty and 1

if it contains geometry. This is a compact representation at low resolutions, but representing a large and sparse grid is infeasible. For instance, Figure 1 shows a scene voxelized at a resolution of $128K^3$ and would require 2^{51} bits or 256 Terabytes.

The sparse voxel octree is a hierarchical representation of the grid which efficiently encodes regions of empty space. An octree recursively divided L times gives us a hierarchical representation of an N^3 voxel grid, where $N = 2^L$ and L is called the *max level*. The *levels* will range from 0 to L, and the *depth* of the octree equals the number of levels below the root.

The sparse representation is achieved by culling away nodes that correspond to empty space. This can be implemented by having eight pointers per node, each pointing to a child node, and interpreting an invalid pointer (e.g. null) as empty space. More commonly, a node is implemented with a *childmask*, a bitmask of eight bits where bit i tells us if child i contains geometry, and a pointer to the first of the non-empty children. These child nodes are then stored consecutively in memory.

The unique traversal *path*, from the root to a specific node in the SVO, defines the voxel without storing the spatial information explicitly in the node. Thus, voxels are decoupled from particular nodes and in the next paragraph, we will show how this fundamentally changes how we can encode geometry.

The Sparse Voxel DAG   Since it is not the nodes, but the paths, that define the voxels, rearranging the nodes will result in the same geometry *if* the childmasks encountered during traversal are the same as if we traversed the original tree. Consider a node with a childmask and eight pointers, one for each child. If two subtrees of the SVO have an identical configuration of childmasks, then they would offer the same continuation of paths, and the pointers to the subtrees could point to one and the same instance instead. Since two nodes then point to the same child, the structure is no longer a tree but its generalization, a *directed acyclic graph* (DAG).

To transform an SVO to a DAG, we could simply start at the root node and test whether its children correspond to identical subtrees, and proceed down recursively. This would be very expensive, however, and instead we suggest a bottom up approach. The leaf nodes are uniquely defined by their childmasks (which describe eight voxels), and so there can at most be $2^8 = 256$ unique leaf nodes in an SVO. The first step of transforming the SVO into a DAG is then to merge the identical leaves. The child-pointers in the level above are updated to reference these new and unique leafs (see Figure 2b).

Proceeding to the level above, we now find all nodes that have identical childmasks *and* identical pointers. Such nodes are roots of identical subtrees and can be merged (see Figure 2c). Note that when compacting the subtrees at this level of the tree, we need

only consider their root-nodes. We iteratively perform this merging operation, merging larger and larger subtrees by only considering their root-nodes, until we find no more merging opportunities. We have then found the smallest possible DAG, and we are guaranteed to find it within L iterations. See Figure 2d for the final DAG of the tree in Figure 2a.

The algorithm is applicable to sparse voxel trees of non-uniform depths, as well as partially reduced DAGs. We can use this latter property to avoid constructing the whole SVO at once, since the whole SVO may be orders of magnitude larger than the final DAG. We construct a few top levels of the tree, and for each leaf node (in the top) we construct a subtree to the desired depth. Each subtree is reduced to a DAG before we insert it into the top and continue with the next subtree. This allows us to conveniently build DAGs from SVOs too large to reside in RAM or on disk.

Implementation details   In our implementation, when building a DAG of max level $L > 10$, we choose to construct the top $L - 10$ levels by triangle/cube intersection tests in a top-down fashion. Then, for each generated leaf node (in the top tree), we build a subtree to the maximum depth, using a straightforward, CPU assisted algorithm based on depth-peeling. There are several papers that describe more efficient ways of generating SVOs that could be used instead (e.g. the method by Crassin and Green [2012]). The subtree is then reduced to a DAG before we proceed to the next leaf.

After all sub-DAGs have been constructed, we have a top-SVO with many separate DAGs in its leaves (i.e., it is a partially reduced DAG). The final DAG is generated by applying the reduction one last time. In the end, the result is the same as if we had processed the whole SVO at once, but the memory consumption during construction is considerably lower.

Thus, reducing an SVO to a DAG simply becomes a matter of iteratively merging the nodes of one level at a time and updating the pointers of the level above. The implementation of these two steps can be done in several ways. We choose to find merging opportunities by first sorting the nodes of a level (using the child-pointers as a 256bit key and a pointer to the node as value). Identical nodes become adjacent in the sorted list, and the list can then trivially be compacted to contain only unique nodes. While compacting, we maintain a table of indirections with an entry, for each original node, containing that node's position in the compacted list. To improve the performance of sorting, we employ a simple but efficient optimization to the encoding of the two lowest levels in the SVO (where the majority of the nodes are). Here, we store subtrees of resolution 4^3 without pointers, in a 64bit integer.

Our implementation of the tree reduction is run on a multi-core CPU. Sorting a list is done with tbb::parallel_sort [Intel 2013], and the remaining operations are performed serially on a single core. Our focus has not been on dynamic scenes which would require real-time compression, but if performance is essential, all parts of the compaction could be performed in parallel (e.g. on the GPU). Sorting on the GPU is well researched and we would suggest using a comparison based algorithm due to the large keys. The compaction step could be parallelized by first filling an array where, for each node in the sorted list, we store a 1 if it differs from its left neighbor and 0 otherwise. A parallel prefix sum over this array will generate the list of indirections from which the parent nodes can be updated in parallel. Finally a stream compaction pass generates the new list of unique nodes [Billeter et al. 2009].

We strip the final DAG of unused pointers by introducing an 8bit childmask that encodes the existence of child pointers and store the pointers consecutively in memory after it. The memory consumption of a node becomes between 8 and 36 bytes (see Figure 3).
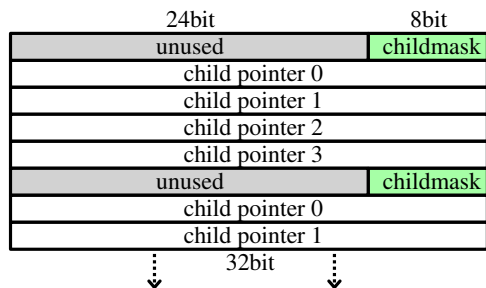
Figure 3: *In memory, both mask and pointers are 4 bytes, and only the pointers to non-empty children are stored consecutively after the childmask. The size of a node is dependent on the number of children.*

4   Ray tracing a sparse voxel DAG

We have implemented a GPU-based raytracer in CUDA that efficiently traverses our scene representation. We use the raytracer primarily for visibility queries to evaluate hard shadows, soft shadows and ambient occlusion for the view samples of a deferred rendering target. Additionally, we have implemented tracing of primary rays (requiring the closest intersection), and to evaluate this implementation, we query a traditional SVO structure, containing material information, with the hit points obtained from tracing in our structure.

The main traversal loop closely resembles that of Laine and Karras [2010a], with a few simplifications regarding intersection tests and termination criteria. Notably, since we do not need to perform any intersection tests with contours, and voxels can be traversed in ray order, we do not need to maintain the distance travelled along the ray. The most significant change, the DAG data structure, requires only minor changes in code.

We use the beam optimization described by Laine and Karras [2010a], with a size corresponding to 8×8 pixels, to improve the performance of primary rays. We also extend the optimization to soft shadows by shooting a beam (i.e. four rays) per pixel from the view-sample position to the area light. We can then find the conservative furthest distance along the rays where we can be certain that no geometry is intersected, and shorten all shadow rays accordingly. By simply tracing the same beam rays in the reverse direction (from the light source), we can find a conservative *interval* of the beam where geometry may be intersected. This improves traversal speed of the shadow rays further. This optimization frequently allows us to find pixels whose entire beams have no intersection with geometry, and for these pixels, no additional shadow rays are required.

Ambient occlusion is usually approximated by casting a number of rays over the hemisphere and averaging the occlusion of each ray. This occlusion is evaluated by mapping the first-hit distance to some falloff function. Laine and Karras [2010b] suggest a way to approximate the same integral using only shadow rays, by incorporating the falloff function into the shadow-ray sampling scheme. We found this method to be faster while providing similar quality. In our implementation, we shoot 64 shadow rays per pixel. Additionally, we demonstrate a useful property of the SVO and DAG scene representation. We can choose to stop traversal when a node (projected on the unit hemisphere) subtends a certain solid angle. This essentially corresponds to using a lower level-of-detail representation the further we are from the point to be shaded and results in images that are darker than ground-truth but maintains high quality near-contact occlusion (see Figure 8b).
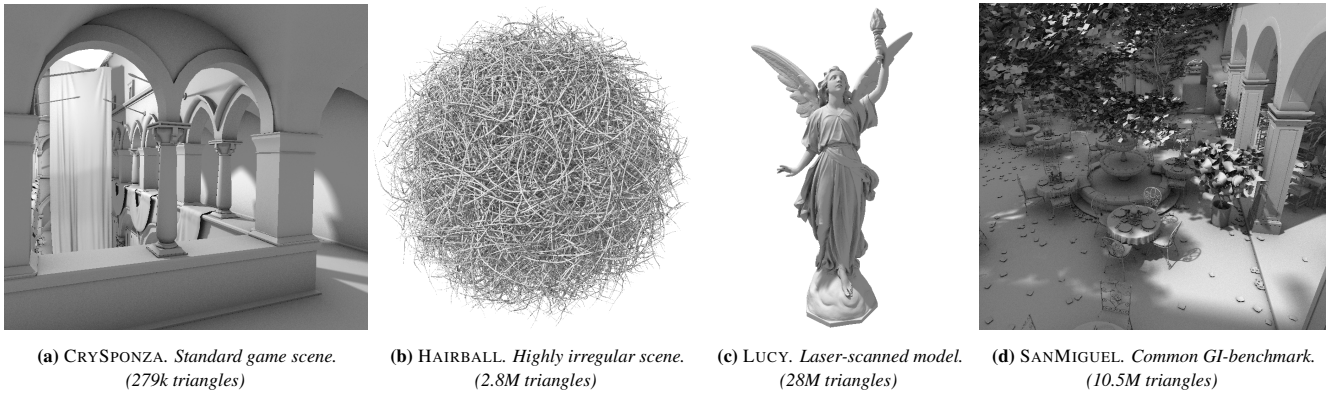
(a) CRYSPONZA. *Standard game scene.* *(279k triangles)*   **(b)** HAIRBALL. *Highly irregular scene.* *(2.8M triangles)*   **(c)** LUCY. *Laser-scanned model.* *(28M triangles)*   **(d)** SANMIGUEL. *Common GI-benchmark.* *(10.5M triangles)*

Figure 4: *The scenes used in our experiments. All images are rendered using our algorithms, with soft shadows and ambient occlusion.*

5 Evaluation

We will evaluate the DAG representation with respect to reduction speed, memory consumption and traversal speed and compare to relevant previous work. We use five test scenes with different characteristics (see Figure 4 and 1).

5.1 Reduction Speed

The time taken to reduce an SVO to a DAG using our algorithm (on an Intel Core i7 3930K) is presented in Table 1. As detailed in Section 3, for resolutions over $1K^3$ we generate sub-DAGs of that size first, connect these with a top tree and finally reduce the entire DAG. As a comparison, Crassin et al. [2012] report a building time of 7.34ms for the CRYSPONZA SVO at resolution 512^3 on an NVIDIA GTX680 GPU. Laine and Karras [2010a] build the ESVO (with contours) for HAIRBALL at resolution $1K^3$ in 628 seconds (on an Intel Q9300).

Table 1: *Time taken to reduce the SVOs to DAGs. The first row is the total time and the second row is the part of that spent on sorting. There is a computational overhead when merging sub-DAGs (see jump between $1K^3$ and $2K^3$), but scaling to higher resolutions is unaffected.*

scene	512^3	$1K^3$	$2K^3$	$4K^3$	$8K^3$
Crysponza	8.4ms	33ms	0.30s	1.0s	4.5s
	1.9ms	5.5ms	0.05s	0.14s	0.6s
EpicCitadel	1.5ms	5.0ms	0.05s	0.20s	0.80s
	0.7ms	1.2ms	0.006s	0.024s	0.10s
SanMiguel	2.6ms	8.3ms	0.06s	0.23s	0.95s
	1.3ms	1.7ms	0.008s	0.03s	0.14s
Hairball	43ms	202ms	2.19s	9.7s	40.6s
	5.1ms	29ms	0.23s	1.0s	4.5s
Lucy	2.6ms	8.4ms	0.08s	0.31s	1.3s
	0.7ms	1.6ms	0.009s	0.04s	0.14s

5.2 Memory Consumption

Comparing the number of nodes of our sparse voxel DAG with the SVO (see Table 2) clearly shows that all tested scenes have a lot of merging opportunities. The CRYSPONZA scene has the largest reduction of nodes with a decrease of $576\times$ (at highest built resolution), which means that each DAG node on average represents 576 SVO nodes. This is probably due to the many planar

and axis aligned walls that generate a lot of identical nodes. The HAIRBALL scene, however, contains no obvious regularities, and still the reduction algorithm manages to decrease the node count by $28\times$, which indicates that scenes can have large amounts of identical subvolumes without appearing regular.

The efficient sparse voxel octree proposed by Laine and Karras [2010a] has voxels with contours designed to approximate planar surfaces especially well. We measured their node count by building ESVOs with their own open source implementation with its provided default settings. The ESVO managed to describe three out of five scenes with significantly fewer nodes than a plain SVO. Still, our DAG required less nodes in all scenes and at all resolutions compared to the ESVO, and by a great margin in HAIRBALL and SANMIGUEL. This further strengthens our claim that our algorithm can efficiently find identical subvolumes in difficult scenes.

The propagation of merging opportunities from leaf nodes and upwards, described in Section 3, can be quantified by the distribution of nodes per level in the resulting DAG (see Figure 5). The figure also shows that the number of merging opportunities increases faster than the total number of nodes as we increase the resolution.
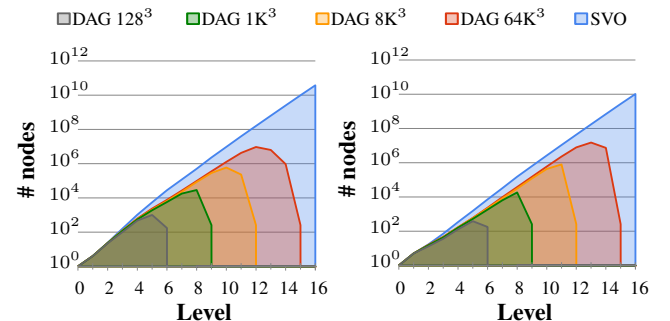

Figure 5: *Distribution of nodes over levels built to different depths. Level zero corresponds to the root, and level 16 corresponds to resolution $64K^3$. CRYSPONZA (left). SANMIGUEL (right).*

The memory consumption depends on the node size (see Table 2). Our DAG consumes 8 to 36 bytes per node (see Figure 3). The ESVO consumes 8 bytes per node if we disregard memory devoted to material properties and only count the childmask, contour and pointer, which describe the geometry. We also compare against the pointer-less SVO described by Schnabel and Klein [2006], where each node consumes one byte. That structure cannot, however,

Table 2: *Comparison of the sparse voxel DAG, ESVO and SVO. The resolutions are stated in the top row. On the left, we show the total node count to allow comparison of merging or pruning opportunities. On the right, we show the total memory consumption of the nodes when the node size has been taken into account. Cases where the DAG has the smallest memory consumption are highlighted in green. The last column states memory consumption per described cubical voxel in the highest built resolutions. ESVO was left out of the comparisons of bits/voxel since it does not encode the same voxels. The* HAIRBALL *was not built in the two highest resolutions due to limited hard drive capacity.*

scene		Total number of nodes in millions						Memory consumption in MB						bit/vox
		$2K^3$	$4K^3$	$8K^3$	$16K^3$	$32K^3$	$64K^3$	$2K^3$	$4K^3$	$8K^3$	$16K^3$	$32K^3$	$64K^3$	
Cryponza	DAG	1	1	2	4	9	23	4	11	27	71	184	476	0.08
	ESVO	5	12	32	94	226	521	38	88	243	715	1 721	3 970	-
	SVO	12	51	205	822	3 290	13 169	12	48	195	784	3 138	12 559	2.07
EpicCitadel	DAG	1	1	1	3	7	18	3	7	19	53	142	371	0.65
	ESVO	1	3	7	17	38	81	7	20	53	124	287	613	-
	SVO	2	6	21	85	340	1 364	2	5	20	81	324	1 301	2.29
SanMiguel	DAG	1	1	2	5	13	34	3	10	31	93	270	742	0.45
	ESVO	4	14	57	229	922	3 698	26	107	433	1 747	7 030	28 212	-
	SVO	4	14	56	224	903	3 628	4	13	53	214	861	3 460	2.08
Hairball	DAG	5	15	44	115	-	-	117	339	996	2 629	-	-	2.24
	ESVO	53	224	924	3 649	-	-	401	1 709	7 049	27 837	-	-	-
	SVO	45	188	781	3 191	-	-	43	180	745	3 044	-	-	2.59
Lucy	DAG	1	1	2	6	16	46	3	10	32	110	348	983	1.54
	ESVO	2	8	26	76	160	255	15	56	198	576	1 219	1 941	-
	SVO	2	7	27	108	430	1 720	2	7	26	103	410	1 640	2.57

be traversed without being unpacked by adding pointers. But it is useful in for instance off-line disk storage. Note that for the largest resolutions used in our tests, four-byte pointers would not be sufficient for an SVO.

A pointer-less SVO structure and the ESVO with only geometry information are two extremely memory-efficient SVO representations. Even so, the sparse voxel DAG requires less memory than the ESVO at all scenes and resolutions. The DAG also outperforms the pointer-less SVO consistently at all but the lower resolutions.
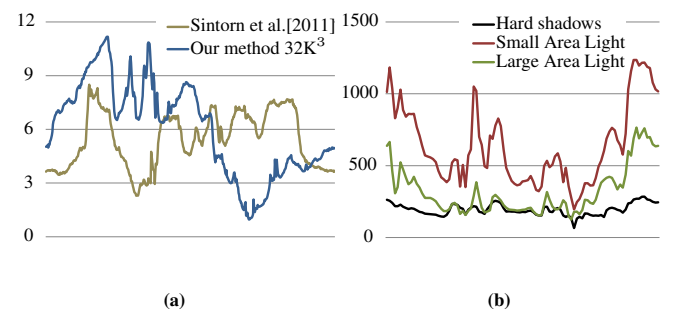
5.3 Traversal Speed

In this section, we will show that the reduction in memory consumption does not come at the cost of increased render times. We have implemented a raytracer that can handle primary rays, ambient occlusion, and hard and soft shadows, by tracing through the DAG structures. The performance of each type of ray is presented, as they result in different memory access patterns. We compare execution times (in MRays/second) against relevant previous work. To be able to discuss the relative performance of the different algorithms more confidently, we have recorded execution times for each frame in a fly-through animation of each scene (shown in the supplementary video).

(a) **(b)**
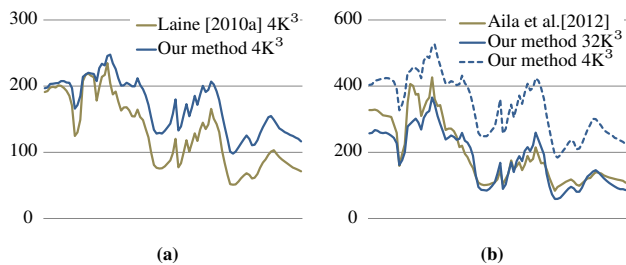
Figure 6: *Traversal speeds in MRays/second for primary rays in a fly-through of* SANMIGUEL. *a) Our DAG and the ESVO on the GTX480. Both use the beam optimization. b) Our DAG vs. triangle ray tracing on the GTX680.*

Primary rays Primary rays are cast from the camera and through each pixel of the image plane. These rays are extremely coherent and differ from the other types of rays in that we must search for the closest intersection instead of aborting as soon as some intersection is found. We have compared our implementation against that presented by Laine and Karras [2010a], on the SANMIGUEL scene on an NVIDIA GTX480 GPU (as their code currently does not perform well on the Kepler series of cards) at a voxel resolution of 4096^3 (as this is the largest tree their method can fit in memory).

The same scene was also used to compare our implementation against the triangle raytracer by Aila et al. [2012]. This test was performed on a GTX680 GPU, and then, the voxel resolution for our method was set to $32K^3$. As shown in Figure 6, our method performs slightly better than the other voxel method and similarly to the triangle raytracer even at extremely high resolutions.


(a) **(b)**

Figure 7: *Comparison of traversal speed for shadow rays in a fly-through of the* CITADEL *scene. a) Comparing frame times (in ms) b) Performace in MRays/sec for our method with varying light source sizes.*

Shadow rays In Figure 7a, we compare the performance of our implementation against the recent, robust shadow algorithm presented by Sintorn et al. [2011]. The time taken to evaluate shadows is measured in milliseconds for a fly-through of the CITADEL scene (a small subset of EPICCITADEL) on a GTX480 GPU. The exact

resolutions differ, but both correspond to approximately one million pixels. The performance of our algorithm is on par with that of Sintorn et al. [2011] for this scene. Their algorithm will perform proportionally to the number of shadow volumes on screen, while ours is largely independent of polygonal complexity and can easily handle scenes with very dense polygons (e.g. SANMIGUEL and LUCY). Figure 7b shows how performance is affected by our beam optimization for varying light source sizes.

Ambient occlusion   Finally, we ray trace our DAG to generate images with ambient occlusion. The ray tracing performance is compared to that of Aila et al. [2012] in the same fly through of the SANMIGUEL scene as for primary rays, and at the same resolution ($32K^3$). Additionally, timings are presented of an experiment where rays are terminated as soon as they intersect a node that subtends a specific solid angle (as explained in Section 4). We shoot 64 rays per pixel, and the maximum ray length is the same in all experiments. The results are outlined in Figure 8a.
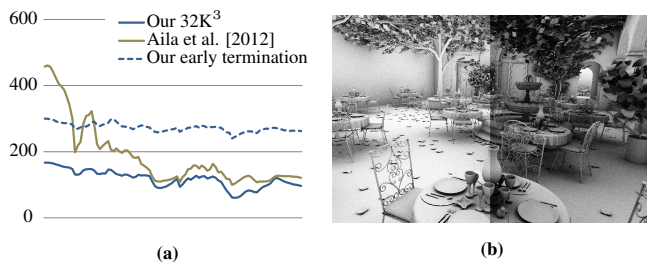
Figure 8: *a) Comparison of traversal speed for AO rays (MRays/sec), in a fly through of* SANMIGUEL. *b) Comparing image quality for 64 AO rays with and without early termination of rays. Rays are terminated at a cone-angle of 9 degrees (0.07sr)*

6   Conclusions

We have shown that sparse voxel DAGs, an evolution of sparse voxel octrees, allow for an efficient encoding of identical regions of space. An algorithm has been presented that finds such regions and encodes them in the smallest possible DAG. We have shown that our algorithm reduces the node count significantly even in seemingly irregular scenes. The decrease in nodes needed to represent the octree ranged from $28\times$ for the highly irregular HAIRBALL to $576\times$ for CRYSPONZA, compared to a tree.

The increased node size is quickly amortized by the reduction in node count, and the DAG representation decreases the total memory consumption by up to $38\times$ compared to the ESVO and up to $26\times$ compared to a minimal pointer-less SVO.

Our algorithm can be used to construct extremely high resolution DAGs without ever having to store the complete SVO in memory.

Despite the greatly reduced memory footprint, our data structure can be efficiently ray traced, allowing, for the first time, high-quality secondary-ray effects to be evaluated in a voxelized scene at very high resolutions.

7   Future Work

The memory layout of the DAG used in this paper could potentially be improved. A more sophisticated layout could for instance enable mixing of tree nodes (with one pointer) and DAG nodes (with up to eight pointers). There are also many identical pointers in the DAG, since nodes can have multiple parents, and by rearranging nodes,

they could potentially share pointers and amortize their cost. Finding an algorithm for efficient rearrangement, optimal or heuristic, would be an interesting direction of future work.

Furthermore, we would like to add a material representation. This could reside in a separate tree or DAG with a completely different connectivity, which is traversed upon finding an intersection between a ray and the geometry.

Even though we have shown extremely high resolution in this paper, the voxels will still look like blocks when viewed closed up. Future work could introduce cycles in the graph to fake unlimited resolution, and it would be particularly interesting to see cyclic endings of a graph that mimic micro geometry of materials, e.g. rocks, plants or trees.

References

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

AILA, T., LAINE, S., AND KARRAS, T. 2012. Understanding the efficiency of ray traversal on GPUs – Kepler and Fermi addendum. NVIDIA Technical Report NVR-2012-02, NVIDIA Corporation, June.

AKENINE-MÖLLER, T., HAINES, E., AND HOFFMAN, N. 2008. *Real-Time Rendering*, 3rd ed. A K Peters.

ANNEN, T., MERTENS, T., BEKAERT, P., SEIDEL, H.-P., AND KAUTZ, J. 2007. Convolution shadow maps. In *Proceedings of Eurographics Symposium on Rendering 2007*, 51–60.

ANNEN, T., DONG, Z., MERTENS, T., BEKAERT, P., SEIDEL, H.-P., AND KAUTZ, J. 2008. Real-time, all-frequency shadows in dynamic scenes. *ACM Transactions on Graphics 27*, 3 (Proceedings of ACM SIGGRAPH 2008) (Aug.), 34:1–34:8.

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

BILLETER, M., OLSSON, O., AND ASSARSSON, U. 2009. Efficient stream compaction on wide simd many-core architectures. In *Proceedings of the Conference on High Performance Graphics 2009*, ACM, New York, NY, USA, HPG '09, 159–166.

CRASSIN, C., AND GREEN, S. 2012. Octree-based sparse voxelization using the gpu hardware rasterizer. In *OpenGL Insights*. CRC Press, Patrick Cozzi and Christophe Riccio.

CRASSIN, C., NEYRET, F., LEFEBVRE, S., AND EISEMANN, E. 2009. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*.

CRASSIN, C., NEYRET, F., SAINZ, M., GREEN, S., AND EISEMANN, E. 2011. Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum (Proceedings of Pacific Graphics 2011) 30*, 7 (sep).

CROW, F. C. 1977. Shadow algorithms for computer graphics. *Computer Graphics 11*, 2 (Proceedings of ACM SIGGRAPH 77) (Aug.), 242–248.

DONG, Z., AND YANG, B. 2010. Variance soft shadow mapping. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2010: Posters*, 18:1.

DONNELLY, W., AND LAURITZEN, A. 2006. Variance shadow maps. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2006*, 161–165.

EGAN, K., HECHT, F., DURAND, F., AND RAMAMOORTHI, R. 2011. Frequency analysis and sheared filtering for shadow light fields of complex occluders. *ACM Transactions on Graphics 30*, 2 (Apr.), 9:1–9:13.

EISEMANN, E., SCHWARZ, M., ASSARSSON, U., AND WIMMER, M. 2011. *Real-Time Shadows*. A.K. Peters.

ELSEBERG, J., BORRMANN, D., AND NÜCHTER, A. 2012. One billion points in the cloud – an octree for efficient processing of 3d laser scans. *ISPRS Journal of Photogrammetry and Remote Sensing*.

FERNANDO, R. 2005. Percentage-closer soft shadows. In *ACM SIGGRAPH 2005 Sketches and Applications*, 35.

GOBBETTI, E., AND MARTON, F. 2005. Far Voxels – a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM Transactions on Graphics 24*, 3 (August), 878–885. Proc. SIGGRAPH 2005.

HACHISUKA, T., JAROSZ, W., WEISTROFFER, R. P., DALE, K., HUMPHREYS, G., ZWICKER, M., AND JENSEN, H. W. 2008. Multidimensional adaptive sampling and reconstruction for ray tracing. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2008) 27*, 3 (Aug.), 33:1–33:10.

HEIDMANN, T. 1991. Real shadows real time. *IRIS Universe 18* (Nov.), 28–31.

INTEL, 2013. Threading Building Blocks. Computer Software.

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

JOHNSON, G. S., HUNT, W. A., HUX, A., MARK, W. R., BURNS, C. A., AND JUNKINS, S. 2009. Soft irregular shadow mapping: Fast, high-quality, and robust soft shadows. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2009*, 57–66.

KATAJAINEN, J., AND MÄKINEN, E. 1990. Tree compression and optimization with applications. *International Journal of Foundations of Computer Science 1*, 4, 425–447.

KONTKANEN, J., AND LAINE, S. 2005. Ambient occlusion fields. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2005*, 41–48.

LAINE, S., AND KARRAS, T. 2010. Efficient sparse voxel octrees. In *Proceedings of ACM SIGGRAPH 2010 Symposium on Interactive 3D Graphics and Games*, ACM Press, 55–63.

LAINE, S., AND KARRAS, T. 2010. Two methods for fast ray-cast ambient occlusion. *Computer Graphics Forum (Proc. Eurographics Symposium on Rendering 2010) 29*, 4.

LAINE, S., AILA, T., ASSARSSON, U., LEHTINEN, J., AND AKENINE-MÖLLER, T. 2005. Soft shadow volumes for ray tracing. *ACM Transactions on Graphics 24*, 3 (Proceedings of ACM SIGGRAPH 2005) (July), 1156–1165.

LEHTINEN, J., AILA, T., CHEN, J., LAINE, S., AND DURAND, F. 2011. Temporal light field reconstruction for rendering distribution effects. *ACM Transactions on Graphics 30*, 4.

MALMER, M., MALMER, F., ASSARSSON, U., AND HOLZSCHUCH, N. 2007. Fast precomputed ambient occlusion for proximity shadows. *Journal of Graphics Tools 12*, 2, 59–71.

MCGUIRE, M. 2010. Ambient occlusion volumes. In *Proceedings of High Performance Graphics 2010*, 47–56.

MEHTA, S. U., WANG, B., AND RAMAMOORTHI, R. 2012. Axis-aligned filtering for interactive sampled soft shadows. *ACM Transactions on Graphics 31*, 6 (Nov.), 163:1–163:10.

MÉNDEZ-FELIU, À., AND SBERT, M. 2009. From obscurances to ambient occlusion: A survey. *The Visual Computer 25*, 2 (Feb.), 181–196.

PARKER, E., AND UDESHI, T. 2003. Exploiting self-similarity in geometry for voxel based solid modeling. In *Proceedings of the eighth ACM symposium on Solid modeling and applications*, ACM, New York, NY, USA, SM '03, 157–166.

PARSONS, M. S. 1986. Generating lines using quadgraph patterns. *Computer Graphics Forum 5*, 1, 33–39.

REEVES, W. T., SALESIN, D. H., AND COOK, R. L. 1987. Rendering antialiased shadows with depth maps. *Computer Graphics 21*, 4 (Proceedings of ACM SIGGRAPH 87) (July), 283–291.

SCHNABEL, R., AND KLEIN, R. 2006. Octree-based point-cloud compression. In *Symposium on Point-Based Graphics 2006*, Eurographics.

SINTORN, E., EISEMANN, E., AND ASSARSSON, U. 2008. Sample based visibility for soft shadows using alias-free shadow maps. *Computer Graphics Forum 27*, 4 (Proceedings of Eurographics Symposium on Rendering 2008) (June), 1285–1292.

SINTORN, E., OLSSON, O., AND ASSARSSON, U. 2011. An efficient alias-free shadow algorithm for opaque and transparent objects using per-triangle shadow volumes. *ACM Transactions on Graphics 30*, 6 (Dec.), 153:1–153:10.

WEBBER, R. E., AND DILLENCOURT, M. B. 1989. Compressing quadtrees via common subtree merging. *Pattern Recognition Letters 9*, 3, 193 – 200.

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. *Computer Graphics 12*, 3 (Proceedings of ACM SIGGRAPH 78) (Aug.), 270–274.