Geometry Caching for Ray-Tracing Displacement Maps

Matt Pharr Pat Hanrahan

Computer Science Department
Stanford University *

Abstract

We present a technique for rendering displacement mapped geometry in a ray-tracing renderer. Displacement mapping is an important technique for adding detail to surface geometry in rendering systems. It allows complex geometric variation to be added to simpler geometry, without the cost in geometric complexity of completely describing the nuances of the geometry at modeling time and with the advantage that the detail can be added adaptively at rendering time.

The cost of displacement mapping is geometric complexity. Renderers that provide it must be able to efficiently render scenes that have effectively millions of geometric primitives. Scan-line renderers process primitives one at a time, so this complexity doesn't tax them, but traditional ray-tracing algorithms require random access to the entire scene database, so any part of the scene geometry may need to be available at any point during rendering. If the displaced geometry is fully instantiated in memory, it is straightforward to intersect rays with it, but displacement mapping has not yet been practical in ray-tracers due to the memory cost of holding this much geometry.

We introduce the use of a *geometry cache* in order to handle the large amounts of geometry created by displacement mapping. By caching a subset of the geometry created and rendering the image in a coherent manner, we are able to take advantage of the fact that the rays spawned by traditional ray-tracing algorithms are spatially coherent. Using our algorithm, we have efficiently rendered highly complex scenes while using a limited amount of memory.

1 Introduction

Displacement mapping is a valuable technique for modeling variation in surface geometry for computer graphics[2]. It allows the user to describe surface detail with either displacement maps (analogous to texture maps, except they define an offset from the base surface), or displacement shaders, which algorithmically describe the detail being added to the surface[6]. Displacement mapping is unusual in that it is essentially a modeling operation that is performed at rendering-time; this has a number of advantages, including easy inclusion of adaptive detail and reduced computation for geometry that is not visible.

The appearance of rough surfaces (which are frequently created by displacement mapping) is strongly affected by self-shadowing. However, scan-line algorithms typically do not include shadows, or if shadows are rendered, approximate techniques such as shadow maps[20][14] are used. Unfortunately, these techniques break down on geometry with fine detail. However, ray-tracing algorithms can compute accurate shadows, and are desirable because of their support for complex luminaires, realistic surface reflection models, and global illumination algorithms.

*{mmp,hanrahan}@graphics.stanford.edu
URL: http://www-graphics.stanford.edu

The first implementation of displacement mapping was in the REYES rendering algorithm[3]. REYES subdivides the original geometry into pixel-sized polygons whose vertices can then be displaced. Because of the huge amounts of geometry that this process produces, it is only feasible in rendering systems based on scan-line techniques, since they can easily render the geometry in the scene one primitive at a time and discard each primitive after processing it. Traditional implementations of the ray-tracing algorithm need random access to the entire scene database, however, since the rays that are traced can potentially pass through any part of the scene.

Bump mapping is a technique that perturbs the normal across a smooth surface to give the appearance of greater detail[1]. It is often used as a substitute for displacement mapping since it is easy to implement and gives similar effects without creating large amounts of geometry. However, it suffers from a number of disadvantages: since the surface of a bump mapped object remains smooth the silhouettes are still smooth, and the object cannot cast shadows on itself. (Techniques for improving the accuracy of shadows onto bump mapped surfaces exist[21][11], but they still do not provide the full flexibility or accuracy of ray-tracing displacement maps.)

In order to implement displacement mapping in a ray-tracing system, it is necessary to develop techniques for managing large amounts of geometry. Though there are techniques for ray-tracing complex scenes[4], and although work has been done to manage large databases for walkthroughs[18] and flight simulators, none of this work examined methods for ray-tracing scenes with more geometry than could fit in memory at once. (Snyder and Barr rendered scenes with billions of primitives, but this geometric complexity was a result of instancing a few simple models many times[15].)

We have developed an efficient algorithm (both in running time and memory use) for ray-tracing displacement maps. The input geometry is lazily triangulated and adaptively subdivided and displaced, and then stored in a *geometry cache*. The cache keeps a fixed number of displaced triangles in memory, and discards geometry that has not been recently referenced when the cache fills up. We have used this scheme to render scenes with millions of polygons created by displacement mapping while only having a small subset of the total geometry in memory at any time. One of our test scenes contains over a million unique displaced primitives, and was rendered quickly using only 22MB of memory.

In the remainder of this paper, we will discuss previous implementations of displacement mapping and other related work, and discuss desirable qualities in displacement mapping algorithms. We will then describe the algorithm that we have implemented, report on its performance, and discuss some of the trade-offs we made. Finally, we will summarize our results and suggest directions for future work.

2 Background and Previous Work

Displacement mapping was first introduced by Cook[2], and later described in detail as part of the REYES algorithm[3]. In REYES, all geometric primitives are subdivided into grids of pixel-sized micropolygons. The vertices of these micropolygons can be displaced before the polygons are shaded. Once the micropolygons have been shaded, they are stochastically sampled and the samples are filtered to generate image pixels. Since the memory requirements to create and store the micropolygons and samples for all the objects in an entire scene simultaneously is prohibitive, REYES divides the screen into rectangular buckets that are rendered individually. For each bucket, all primitives that are possibly visible in that bucket (taking into account the maximum amount that the primitive could be displaced), are turned into grids, possibly displaced, shaded, and broken up into micropolygons. The micropolygons are then placed into the buckets that they overlap, and are stored in memory until their buckets have been processed.

Though displacement mapping is naturally incorporated into the REYES algorithm, this approach suffers from a number of disadvantages that our algorithm addresses:

- The resolution of the grid of micropolygons used for shading is the same as the resolution used for displacements. This means, for example, that much higher grid resolutions are often necessary for acceptable looking displacements than are necessary for acceptable looking shading, particularly when displacements are large or abrupt and the shading is smooth.

  Our system separates the displacement mapping process from the shading process, by virtue of being in a ray-tracing renderer. Geometry is subdivided and displaced before rays are intersected with it, and the subdivision is adaptive, so that the displaced surface is accurately represented by the displaced polygons. Shading occurs at the points where rays intersect the displaced surface. We are also able to greatly reduce geometric complexity by simplifying the displaced geometry before rendering it.

- If a displaced object is obscured by a closer object, REYES still performs displacement (and shading) calculations, since no visibility checks are performed until after the micropolygons have been created and shaded. (However, hierarchical visibility techniques can reduce the impact of this problem in scan-line systems[5].)

  This is one of the classic trade-offs between ray-tracers and scan-line renderers. Since we do not create the displaced triangles until a ray approaches the geometry being displaced, we defer creation until the geometry is needed. For scenes with high depth complexity, this greatly reduces processing time since much of the geometry is not visible.

- If shadow maps or environment maps are to be used for rendering the scene, then REYES re-renders the scene when creating these maps. All of the calculations necessary for displacement mapping must be redone for each of these renderings. In addition, these techniques are prone to aliasing, particularly when used on detailed self-shadowing geometry.

  Since we are able to trace rays to compute shadows, reflections, and transmission, these effects are much more accurately rendered in our system. We make use of a different form of coherence than REYES: nearby eye rays will generally intersect surfaces in nearby locations, and the additional rays traced from these nearby intersections will follow similar paths. Since our algorithm recalculates geometry only if it is needed again after being discarded from the cache, we can potentially only calculate the displaced geometry once.

Other work in displacement mapping has focused on techniques for directly intersecting rays with displacement mapped geometry without subdividing it into small polygons[16][13]. Given a primitive and a precomputed displacement map, these techniques first compute a mapping that flattens out the original primitive. By applying the inverse of this mapping to a ray, the (now curved) ray can be intersected with the displacement map, which is treated as a heightfield. Though this technique is attractive since the memory requirements are modest, in practice it has two problems: first, finding the intersection of a curved ray with a heightfield is an expensive operation, typically requiring iterative calculations to find an intersection. Second, this technique only handles displacements along the surface normal.

Kajiya has developed a technique for rendering fractal terrains that creates triangles lazily[8]. This technique is similar to our method for displacement mapping, though it did not cache the geometry it created and does not try to remove unnecessary geometry from the triangle mesh. Musgrave *et al.* also discuss lazy creation of fractal terrain data[12], but do not provide details or statistics about this part of their algorithm.

3 Basic Algorithm

Our algorithm has three phases (Figure 1). Starting with an input triangle mesh, we insert the triangles into a regular grid of voxels[1]. We use a bound on the maximum distance any input triangle will be displaced to determine which voxels could receive geometry from each primitive when it is subdivided and displaced. The grid that holds this information is called the *contributor grid*.

The second phase starts when a ray intersects the contributor grid. Starting with the first voxel that the ray intersects, we subdivide and displace all the input triangles inside that voxel, and store the displaced triangles in a second voxel grid (with the same dimensions and resolution as the contributor grid). This second grid is called the *geometry cache*. We then perform ray intersection tests with the displaced geometry in the voxel. The ray continues stepping through the voxels until we find an intersection with the displaced triangles or it leaves the grid.

The third phase is a recycling phase that allows us to strictly limit the memory used to store the displaced triangles. If we determine that more than some fixed number of displaced triangles is in the geometry cache, we discard the least recently used entries.

3.1 Subdivision and Displacement

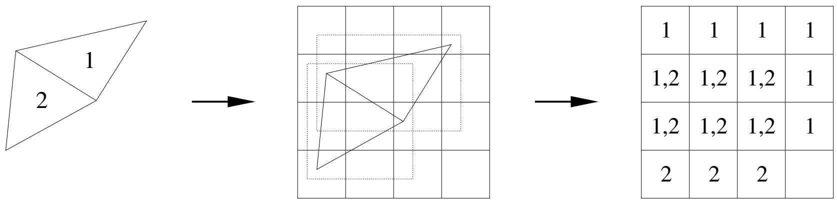
Our system allows the user to write displacement shaders in a C-like language similar to the RenderMan shading language[6]. The shader can access information about the local differential geometry at the point where it is being applied; it can displace this point anywhere in space.

Given an input triangle, we execute the shader on its three vertices. If the resulting triangle requires further subdivision (based on the subdivision criteria developed below), we recursively subdivide it by splitting each of its edges in half and computing positions for the new vertices created by the subdivision, thus creating four new subtriangles. Each of these is checked to see if it needs to be subdivided further, *etc.*

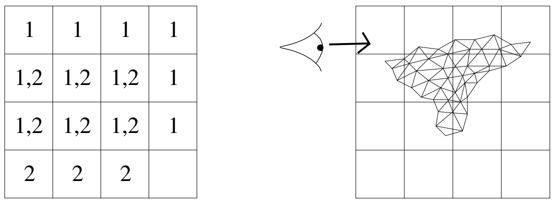
We subdivide based on triangle edge length in world space. The user can set a desired edge length, and all displaced geometry will have triangle edge lengths less than that length. Given a triangle to possibly subdivide, we check the length of each edge against the maximum edge length. If all of the edges are shorter than the maximum edge length, we terminate the recursion and add the triangle to the geometry cache. If all of the edges are longer, then we split them all and use the displacement shader to calculate positions for the midpoints of the edges. If only some of the edges need to be subdivided, we split all of them, but only evaluate the displacement shader for the midpoints of the edges that wanted to subdivide; for the rest of the midpoints, we just interpolate between the positions of the vertices at the ends of their edges (Figure 2). This technique is frequently used in algorithms that subdivide patches into triangles for rendering. It ensures that if an triangle that shares an edge with a subdivided triangle does not need to be subdivided itself, then the triangle that was subdivided will meet up along their shared edge.

It would be preferable to base subdivision on edge length in image space, rather than world space, but there are a number of difficulties with this. Not only do realistic lens systems introduce non-linear projections that make determining an object's image space

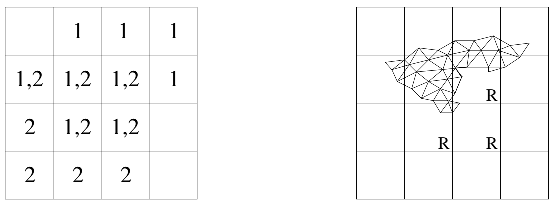
[1] Our current implementation only operates on triangle meshes; it would be straightforward to extend our algorithm to operate on curved surfaces as well if the geometry subdivision step accounted for the true curved geometry.



Phase 1: The input triangles (left) are added to the contributor grid (middle). The contributor grid stores a list of input triangles at each voxel, recording which input triangles could contribute geometry to that voxel after being subdivided and displaced. To create these lists, a bounding box is computed for each triangle. The bounding box is expanded by the maximum displacement distance, and the triangle is added to the lists in each voxel that its bounding box overlaps.



Phase 2: A ray approaches and is stepped through the voxels in the geometry cache (right). When the ray enters a voxel, we test whether all the displaced triangles are present in the geometry cache. To do this, we compare the list of potential input triangles in the corresponding voxel in the contributor grid to the list of input triangles accounted for in the geometry cache. In this example, we must subdivide and displace triangle 1 for this ray.



Phase 3: When the geometry cache fills up, the recycling phase begins. The contents of voxels that have not been referenced recently are discarded (here these voxels are marked with the letter 'R'.) The reason that the recycled voxels don't appear to be completely empty is that the triangles that overlap multiple voxels are individually stored in each of them. Note that after triangle 1 was subdivided and displaced in phase 2, we determined that some voxels in the geometry cache received no displaced triangles from it, even though those voxels had Triangle 1 in their list of potential contributors. We remove the triangle from these lists in the contributor grid.

Figure 1: The three phases of the algorithm.

size difficult[10], but we would also like to be able to account for geometry between the viewer and the displaced object that magnifies or reduces the size of the object. Light sources introduce another complication: when displaced geometry casts a shadow, the part of the shadow cast by a single displaced triangle should be approximately the size of a pixel in the final image. How to do all of this well is still an open research issue.

Before we add the displaced geometry to the cache, we can reduce the amount of geometry by removing geometric detail that is too subtle to be visible. Given a triangle that has been subdivided once, we have four triangles defined by six vertices. We test if these four triangles are nearly coplanar by checking if the vertices at the midpoints

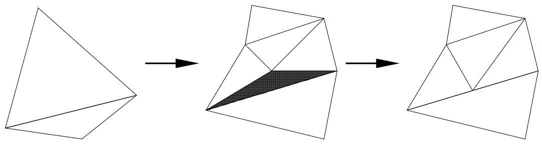


Figure 2: The cracking problem. Two input triangles (left) are subdivided to different levels. If the vertex along the shared edge is not constrained to lie along the edge on the lower triangle, then a crack will result (center). This can be fixed by interpolating its location between the positions at the ends of the edge (right).

of the edges of the parent triangle are nearly on the edges (within some user-set tolerance). If so, we merge the four triangles into one, defined by the three corner vertices. If only some of the edges are nearly flat, then we move the midpoints of the nearly flat edges to lie on the edges; this prevents cracking in the mesh from adjacent triangles that are flat and are simplified. This process starts at the finest level of subdivision from the displacement step, and continues recursively until a non-flat region is found or all of the displaced triangles from a single input triangle have been merged into a single displaced triangle. This technique can save valuable space in the geometry cache. It is surprisingly effective: in our experience 30% to 90% of the geometry can be merged.

As we add displaced triangles to the geometry cache, we can improve the accuracy of the contributor grid. Since the voxels in the contributor grid start out by recording which input triangles *could* contribute geometry to them, we can remove triangles from the lists in the voxels that they did not actually contribute geometry to. Should we later need to recompute the displaced triangles for a voxel (after it has been recycled, for example), this reduces the number of input triangles that must be subdivided and displaced to the ones that actually do contribute geometry.

As a final optimization, we ensure that voxels in the geometry cache that contain large numbers of triangles do not excessively slow down intersection calculations. We put another voxel grid inside any voxel that has more than a hundred or so displaced triangles in it. The resolution of this voxel grid varies depending on how many triangles are in the voxel. This is an important time/memory tradeoff; adding these sub-voxel grids sped up rendering of our test scenes by a factor of three, with a modest cost in memory. If we were too aggressive in adding these sub-voxel grids or if they were excessively high-resolution, total memory use would be unacceptable. This technique is similar to a technique for accelerating ray-tracing proposed by Jevans[7].

3.2 Cache Management

Good performance from the geometry cache is central to our algorithm. Three factors can be adjusted to improve cache performance: capacity, access coherence, and replacement strategy. The more the cache can hold, the more likely it will have the geometry in it that we need; the more coherently we access it, the more likely geometry that was created for a previous access will still be present; and the better we are at picking voxels to be replaced that will not be accessed again for a long time (if at all), the less computation will later be necessary to fill voxels.

By minimizing the amount of memory we use to store each displaced triangle, we are able to fit a larger cache in a given amount of memory. Each voxel in the geometry cache holds an array of vertex locations and an array of structures that hold vertex indices used by each triangle in the voxel. The recursive subdivision done to compute the displaced triangles maintains information about shared vertices, so we can store a compact points-polygons representation of the displaced triangles in the voxel. Also, since the displaced triangles are pixel-sized, we do not store per-vertex normals for them. Instead, we use the triangle normal for shading; this saves both time and storage space.

We have tried to make the geometry cache access patterns as coherent as possible by carefully choosing the screen sampling pattern. This makes the eye rays spatially coherent and in practice, the rays spawned from nearby eye rays are also spatially coherent. Most ray-tracers render the image one scan-line at a time, rendering pixels sequentially in each scan line. This is not a very coherent sampling pattern. To improve coherence, a Hilbert curve can be used, which processes the pixels in a much more coherent manner[19]. Another effective sampling strategy is to divide the screen into rectangular buckets and render them sequentially.

Our replacement strategy is to discard the geometry stored in the least recently accessed voxel when the cache is full. This strategy performs well and is widely used to manage cache replacement; we have not performed extensive tests to evaluate the effectiveness of other replacement strategies.

4 Analysis and Results

To evaluate the system, we gathered statistics for two test scenes.

- A displacement mapped sphere and a displacement mapped box, sitting on a table in a room (Plate 1). The displacement shader for the sphere uses two sinusoidal functions to determine the displacement, and the box is displaced with a shader that bevels its edges. The geometric simplification step greatly reduced the amount of geometry used to represent the box and somewhat reduced the geometry needed for the sphere. Although this is a simple scene, it demonstrates the calculation of accurate transmission and shadows in a scene with displacement maps.

- The facade of a sandstone building, viewed in late afternoon (Plate 2). Approximately 90% of the pixels in this scene have geometry that is displacement mapped; the stonework on the front of the building is done with two displacement shaders, one for the grooved top part, and one for the craggy bottom part. Additionally, the window pane is displaced and transmissive; the edges of it are beveled, and the rest of it is slightly bumpy, simulating low-quality glass.

  The shadows cast by the stonework are critical to the success of this image. This is particularly evident in comparison to Plate 3 (where the facade scene is rendered without computing any shadows). Not only are the shadows that the stone casts on the windowsill important, but the self-shadowing brings out the ridges of the stone, giving it a depth that is lacking without them. The glass windowpane created by displacement mapping is also distinctive and was easily included in the scene since we could trace rays to compute the specular transmission.

So that we could have a baseline with which to compare the execution time and memory use of our algorithm, we rendered each scene without displacement mapping. Next, we rendered the scenes with a cache of unlimited size to determine how much storage would be necessary to store the whole displacement mapped model and to find a lower bound on running time (Table 1). We will present detailed results for only the facade scene in this paper; the results for the sphere and box scene were very similar.

Rendering the undisplaced scenes was quick and memory use was low. Displacement mapping caused geometric complexity to increase a thousandfold, running time to double, and memory use to increase ten times when there was no limit to the amount of geometry in memory. The fact that running time only doubled points to one of the

	Not Displaced	Unlimited Cache
Image resolution	480 x 720	480 x 720
Samples per pixel	1	1
Input triangles	1,254	1,254
Displaced triangles	0	1,126,661
Time to render	11m 5s	24m 38s
Memory used	10.9MB	94.9MB
Geometry saved by simplification	n/a	35%

Table 1: Statistics for the building facade. All tests were run on an SGI Onyx with a 250MHz R4400 processor and enough memory so that paging did not have an impact on running time.

main advantages of ray-tracing: complexity is handled well, since the time to intersect rays with geometry grows approximately logarithmically with geometric complexity.

Next, we turned to testing the performance of our geometry caching algorithm and comparing the effects of different screen sampling algorithms. We tried three different sampling patterns: scan-line, buckets, and Hilbert curve. The scan-line pattern was the standard top-to-bottom and left-to-right order that most ray-tracers use, the bucket pattern subdivided the screen into 16 by 16 pixel buckets and rendered those top-to-bottom and left-to-right, and the Hilbert curve sampling pattern generated a Hilbert curve over the screen and sampled pixels in the order that the curve passed over them.

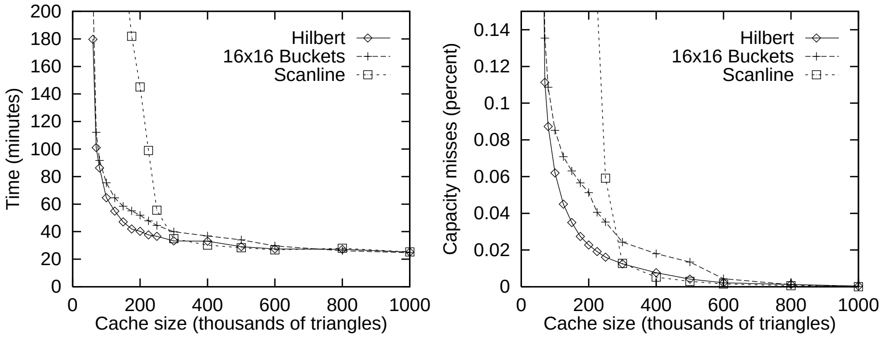


Figure 3: Effect of cache size versus running time for various screen sampling strategies in minutes (left), and capacity misses (right).

The cache was effective for all sampling patterns, though the Hilbert curve and the bucket pattern resulted in significantly better performance than scan-line sampling (Figure 3). In particular, using Hilbert curve sampling and a cache of 200,000 displaced triangles (less than 20% of the total geometric complexity) used only 20 more megabytes of memory than the undisplaced scene, and took only 60% more time than if we had used a cache of unlimited size. If the size of the cache was too small (less than 10% of total geometric complexity), performance decreased rapidly as the cache thrashed.

A standard measure of cache effectiveness is the miss percentage: the percentage of all cache accesses where the information of interest is not in the cache. Two causes of cache misses are relevant for our geometry cache: compulsory misses and capacity misses. Compulsory misses are unavoidable; the first time some piece of information is needed, it will not be in the cache. We strive to minimize capacity misses, which occur when some piece of information that was previously discarded from the cache is later needed again. We gathered data on the percentage of cache accesses that were capacity misses (Figure 3) and the percentage of all misses that were capacity misses (Table 2). As one would expect, capacity misses and running time are closely related, and the percentage of misses due to capacity decreases as cache size increases.

One interesting result is how scan-line sampling leads to a catastrophic drop in performance when the cache is too small, while the more coherent sampling patterns degrade more gracefully. We believe that this is a result of our replacement strategy: when a cache that has a LRU replacement strategy is too small to hold all of the data being accessed and when the access pattern goes sequentially through the data, cache performance is terrible. Another replacement strategy, like randomly picking a cache entry for replacement, would probably make scan-line sampling degrade more gracefully.

Cache statistics for the facade scene				
Geometry cache size	100,000	200,000	400,000	600,000
Cache size as percentage of total geometric complexity	8.9%	17.7%	35.5%	53.2%
Total memory use	22.3MB	31.0MB	46.9MB	61.7MB
Increase in memory use compared to undisplaced scene	11.4MB	20.1MB	36.0MB	50.8MB
Cache hit percentage	99.93%	99.97%	99.98%	99.99%
Percentage of total misses that are capacity misses	87.2%	72.7%	48.6%	22.6%
Running time	1h 4m 36s	40m 16s	33m 8s	27m 26s

Table 2: The effect of cache size on memory use, cache hit percentage, capacity misses, and running time. (Hilbert sampling was used for these tests.)

5 Conclusion and Future Work

We have presented an algorithm that allows for the efficient rendering of displacement maps in a ray-tracer. This is the first such algorithm that has been shown to be effective in realistic, highly complex scenes. Previous implementations of displacement mapping in non ray-tracing systems were not able to trace rays to compute shadows, reflections, and other effects; as a result, they did not robustly handle self-shadowing geometric detail and were unable to accurately compute specular light transport. By providing for efficient ray-tracing of displacement maps, our algorithm makes these rendering techniques possible. We store a subset of all of the displaced triangles in a geometry cache that is lazily filled with displaced triangles as the rays approach the displaced geometry; when the cache fills up, old geometry is discarded from it. Pixels are rendered in a spatially coherent pattern, and because of the spatial coherence among rays spawned by nearby eye-rays, access patterns to the cache are coherent enough that small caches suffice to render scenes quickly.

Some rendering algorithms access the scene in a very incoherent manner (traditional implementations of path tracing[9], for example); our algorithm performs poorly when these algorithms are used unless the cache is big enough to hold almost all of the geometry created. Similarly, a scene with a hundreds of light sources in the scene would have poor access patterns. Teller *et al.*[17] were able to perform radiosity computations on enormous models in small amounts of memory by reordering the computation so that it accessed the scene database in a coherent pattern. We expect that by similarly reordering the order in which the rays other than the eye-rays are traced, cache behavior should improve further.

In comparison to other rendering techniques, ray-tracing has the best performance with scenes with huge amounts of geometry, since the time to intersect a ray with a collection of geometry is approximately logarithmic in the amount of geometry. Until now, the amount of detail in scenes that are ray-traced has been limited by how much geometry can fit into memory; the algorithms presented in this paper have made it possible to ray-trace scenes with more geometric complexity than can fit into memory at once, without requiring extensive modifications to the internals of the rendering system. We have used these techniques to make displacement mapping available in our rendering system, and have been able to make images with new effects and previously impractical amounts of geometric complexity in our ray-tracer.

6 Acknowledgments

Craig Kolb and Reid Gershbein helped develop the rendering toolkit where this work was done and offered many helpful suggestions. Eric Veach suggested how to merge almost flat triangles without causing holes in the mesh, and Gordon Stoll was the source of many interesting conversations about caching and rendering. Thanks also to Julie Dorsey and Hans Pedersen for asking for this feature in *toro* and for enduring early versions of the implementation. This research was supported by NSF contract CCR-9508579 and by equipment grants from Silicon Graphics, Inc.

References

[1] BLINN, J. F. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)* (Aug. 1978), vol. 12, pp. 286–292.

[2] COOK, R. L. Shade trees. In *Computer Graphics (SIGGRAPH '84 Proceedings)* (July 1984), H. Christiansen, Ed., vol. 18, pp. 223–231.

[3] COOK, R. L., CARPENTER, L., AND CATMULL, E. The Reyes image rendering architecture. In *Computer Graphics (SIGGRAPH '87 Proceedings)* (July 1987), M. C. Stone, Ed., pp. 95–102.

[4] GLASSNER, A. E. *An Introduction to Ray Tracing*. Academic Press, 1989.

[5] GREENE, N., AND KASS, M. Hierarchical Z-buffer visibility. In *Computer Graphics Proceedings, Annual Conference Series, 1993* (1993), pp. 231–240.

[6] HANRAHAN, P., AND LAWSON, J. A language for shading and lighting calculations. In *Computer Graphics (SIGGRAPH '90 Proceedings)* (Aug. 1990), F. Baskett, Ed., vol. 24, pp. 289–298.

[7] JEVANS, D., AND WYVILL, B. Adaptive voxel subdivision for ray tracing. In *Proceedings of Graphics Interface '89* (Toronto, Ontario, June 1989), Canadian Information Processing Society, pp. 164–72.

[8] KAJIYA, J. T. New techniques for ray tracing procedurally defined objects. In *Computer Graphics (SIGGRAPH '83 Proceedings)* (July 1983), vol. 17, pp. 91–102.

[9] KAJIYA, J. T. The rendering equation. In *Computer Graphics (SIGGRAPH '86 Proceedings)* (Aug. 1986), D. C. Evans and R. J. Athay, Eds., vol. 20, pp. 143–150.

[10] KOLB, C., HANRAHAN, P., AND MITCHELL, D. A realistic camera model for computer graphics. In *SIGGRAPH 95 Conference Proceedings*, R. Cook, Ed., pp. 317–324.

[11] MAX, N. L. Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer 4*, 2 (July 1988), 109–117.

[12] MUSGRAVE, F. K., KOLB, C. E., AND MACE, R. S. The synthesis and rendering of eroded fractal terrains. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, J. Lane, Ed., vol. 23, pp. 41–50.

[13] PATTERSON, J. W., HOGGAR, S. G., AND LOGIE, J. R. Inverse displacement mapping. *Computer Graphics Forum 10*, 2 (June 1991), 129–139.

[14] REEVES, W. T., SALESIN, D. H., AND COOK, R. L. Rendering antialiased shadows with depth maps. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, M. C. Stone, Ed., vol. 21, pp. 283–291.

[15] SNYDER, J. M., AND BARR, A. H. Ray tracing complex models containing surface tessellations. In *Computer Graphics (SIGGRAPH '87 Proceedings)* (July 1987), M. C. Stone, Ed., vol. 21, pp. 119–128.

[16] TAILLEFER, F. Fast inverse displacement mapping and shading in shadow. In *Graphics Interface '92 Workshop on Local Illumination* (May 1992), pp. 53–60.

[17] TELLER, S., FOWLER, C., FUNKHOUSER, T., AND HANRAHAN, P. Partitioning and ordering large radiosity computations. In *Proceedings of SIGGRAPH '94* (July 1994), A. Glassner, Ed., pp. 443–450.

[18] TELLER, S. J., AND SÉQUIN, C. H. Visibility preprocessing for interactive walkthroughs. In *Computer Graphics (SIGGRAPH '91 Proceedings)* (July 1991), T. W. Sederberg, Ed., vol. 25, pp. 61–69.

[19] VOORHIES, D. Space-filling curves and a measure of coherence. In *Graphics Gems II*, J. Arvo, Ed. Academic Press, 1991, pp. 26–30.

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

[21] WOO, A., POULIN, P., AND FOURNIER, A. A survey of shadow algorithms. *IEEE Computer Graphics and Applications 10*, 6 (Nov. 1990), 13–32.