A Visibility Field for Ray Tracing

Jesper Mortensen[1] Pankaj Khanna[1] Insu Yu[1] Mel Slater[1,2]
[1]Department of Computer Science, University College London, London, UK
[2]ICREA-Universitat Politècnica de Catalunya, Department de LSI, Barcelona, Spain
j.mortensen | p.khanna | i.yu | m.slater @cs.ucl.ac.uk

Abstract

This paper presents a type of visibility data structure for accelerated ray tracing. The visibility field is constructed by choosing a regular point subdivision over a hemisphere to obtain a set of directions. Corresponding to each direction there is then a rectangular grid of parallel beams, with each beam referencing a set of identifiers corresponding to objects that intersect it. Objects lying along a beam are sorted using a 1D BSP along the beam direction. The beam corresponding to any ray can be looked up in small constant time and the set of objects corresponding to the beam can then be searched for intersection with the ray using an optimised traversal strategy. This approach trades off rendering speed for memory usage and pre-processing time. The data structure is also very suitable for hemisphere integration tasks due to its spherical nature and results for one such task - Ambient Occlusion - are also presented. Results for several scenes with various rendering methods are presented and compare favourably with a well established approach, the single-ray Coherent Ray Tracing approach of Wald and Slusallek et al.

1. Introduction

There have been significant advances towards real-time ray tracing in recent years, through the exploitation of algorithms that are tailor made to perform well on today's graphics hardware [28, 6, 39, 29]. Every ray tracing algorithm has to deal with the problem of ray-object traversal – that is, to find for any ray the nearest surface that it intersects, and this problem has received a great deal of attention since the introduction of ray tracing into computer graphics [2, 42]. All successful methods rely on a data structure that when traversed by a ray, delivers a set of objects, the candidate set, or the potentially visible set, that the ray may intersect. In this paper we present a modification of this standard approach. We exploit a 4-dimensional data structure, which is similar to a light field that instead of storing radiance stores object identifiers. The data structure is a special instance of a 'virtual light field' (VLF) [34], that we call VLF-RT in this paper. A ray is used as an array look-up index into the VLF-RT data structure, and immediately delivers a set of candidate objects for ray-object intersection testing. That set of objects, a tiny fraction of the original number in the scene, may be traversed linearly or by any other method.

We discuss the background literature and state-of-the art in Section 2. The VLF-RT data structure for ray tracing is presented in Section 3. In Section 4 we motivate and describe the use of a uniform depth partition for the final traversal of the potentially visible set (PVS) returned from the ray lookup into the VLF-RT. Implementation details are given in Section 5. Results in the form of comparisons between Coherent Ray Tracing (CRT)[37, 36] and the VLF-RT method are presented in Section 6. Section 7 presents application to Ambient Occlusion and conclusions in Section 8, including a discussion of how dynamic scene changes are very simple in this method. In this paper we concentrate only on 'classical' ray tracing as described by Whitted [42]. For compatibility with CRT all our examples are limited to polygons, and in our main results, triangles. However, the method is not bound to polygonal objects.

2. Background

Ray tracing was the first type of global illumination algorithm introduced into computer graphics [42]. It very simply and elegantly supports shadows, specular reflection and transmission, and also solves the problem of visibility. It does not correctly handle light paths that involve diffuse or glossy reflections, and these will not be considered in this paper. The overall benefits of ray tracing have been discussed many times, for example [13] for a general overview and standard algorithms, and [37] for potential benefits as compared to the standard graphics pipeline. In the original paper Whitted pointed out that the vast amount of the time to produce a ray traced image is taken up by ray-object intersection calculations. Many techniques have been developed to try to reduce this time. These can be classified into

object-space subdivision and ray-space subdivision methods. The former constructs a scene space subdivision, such that each cell in the subdivision references a relatively small set of objects, and ray traversal through this subdivision is relatively simple and fast. For any given ray the vast majority of objects are therefore never tested for intersection - only those that are picked up by the ray traversal scheme are considered as candidates for intersection. Examples of this method include bounding extent hierarchies [22, 14], direct space subdivision methods - such as uniform space subdivision [11, 1, 8], oc-trees [12], and BSP trees [21, 20, 35]. It was argued in [35] that of these the BSP subdivision scheme results in the fastest ray traversal, with logarithmic time in the number of polygons. Ray classification schemes on the other hand exploit coherence amongst rays. One example of this was the light buffer [16] which efficiently computed intersections for 'shadow feeler' rays. However, a general ray classification approach that applied to the entire ray tracing process was provided by [3]. Rays were represented as points in 5D space and a 32-tree of ray space was lazily built as each successive ray was encountered (in fact six 32-trees each representing one of the six faces of a bounding box around the scene). Each cell of a 32-tree represents a set of similar rays, and corresponding to each cell is a candidate set of objects. Every object in the candidate set is such that at least one of the rays in the cell intersects it. In other words a cell of the 32-tree corresponds to a beam in 3D space that intersects a set of objects - the candidate set for the cell. The size of the tree depends on the maximum permitted size of the candidate object set. Now given any new ray, it is filtered down the tree, its candidate set identified, and intersections carried out with these. The ray classification scheme can also be enhanced by employing an adaptive subdivision method [32]. Another variation on this approach is presented in [25] which removes one of the dimensions of ray space by using ray coherence. As many rays can lie on the same line, duplication of rays is reduced by classifying lines instead of rays. These variations use a binary search tree to find intersections with a list of candidate objects. The VLF-RT algorithm presented in this paper may be thought of as a much more efficient representation of this same idea - since in this case similar rays are also grouped together and each such group of rays has a candidate object set. However, the data structure is much simpler than the 32-tree, and ray-candidate set retrieval is look-up rather than a tree traversal. Within each of these two broad categories there have been many proposals for further and substantial improvement in ray-object traversal speed. For example, building on the idea of a BSP representation Havran [19, 18] introduced rope trees to further accelerate ray-BSP tree traversal, and in [30] the cost of BSP traversal is reduced by computing BSP tree entry points for collections of rays. In addition caching schemes have been introduced to reuse elements of a solution across several views, exploiting a kind of ray-view coherence [41, 40].

With the introduction of programmable graphics hardware replacing the fixed function pipeline and the rapid growth of performance of graphics hardware a number of attempts have been made to map ray tracing to graphics processing units (GPUs) [29, 6]. Raw ray-triangle intersection on the GPU have been shown to outperform CPU implementations. Nevertheless, kd-tree space subdivision data structures map poorly to streaming architectures and necessitates the use of uniform grids; a suboptimal acceleration structure. Attempts have been made to map kd-tree data structures to the GPU [10], but so far CPU ray tracing algorithms perform better than GPU counterparts.

Advances in processor power and network bandwidth have supported a massive speed up in ray tracing so that today it is possible to attain interactive speed for millions of polygons on clusters of consumer PCs [39]. This research has relied on space subdivision schemes for fast ray-intersection solutions, in particular BSP trees, together with precise organisation of the overall algorithm to fit the needs of the hardware, and parallel implementation over PC clusters [38, 4]. The evidence to date suggests that one scheme in particular; coherent ray tracing (CRT) [37] is the fastest implementation of ray tracing. This uses a BSP tree space subdivision. The implementation is organised so that most memory accesses fall within the first two processor caches, which itself results in a speed-up by half an order of magnitude as reported in the original paper. Moreover, packets of 4 rays are SIMD traced in parallel. We have also implemented the single ray CRT scheme, and it is with the results of this that we compare our new approach in Sections 6 and 7.

3. Visibility Field for Ray Tracing

In the following sections we discuss the data structure, its construction and how to perform ray queries against it.

3.1. Data Structure

The virtual light field data structure was originally inspired by the light field [27, 15] and the type of representation used is similar to that in [5] and also to a data structure used for visibility culling in [7]. Whereas light fields typically only store radiance at the first intersection of a ray with an object, Layered Depth Images [31] maintain radiance information about each of the surfaces that rays intersect rather than just the first surface, and in that sense the data structure is also similar to LDI. A general VLF data structure for a view independent global illumination solution has been previously used where radiance information

was stored [34]. However, in VLF-RT we never store radiance, only object (in fact polygon) identifiers. The VLF-RT thus uses a modification of the VLF data structure as originally introduced but oriented towards ray tracing, and therefore the solution is view-dependent. It also requires an order of magnitude less memory.

We now describe VLF-RT. A scene can be enclosed, for example, by a regular cuboid. Suppose this is a cuboid bound by $(-1,-1,-1)$ to $(1,1,1)$. Consider the lower face $(z=-1)$ bounded by $(-1,-1,-1)$ and $(1,1,-1)$. This is subdivided into $n \times n$ square tiles. Each tile is the base of a beam parallel to the z-axis that extends infinitely (though only the finite part that intersects the scene is of interest). This set of $n \times n$ parallel beams along the z-axis is called the canonical *parallel subfield* (PSF). If l points with spherical coordinates $\omega_i = (\theta_i, \phi_i)$ are chosen on the unit hemisphere, then PSFs are defined as rotations of the canonical PSF by rotating the $(0,0,1)$ direction into the corresponding spherical point. The rotation can be achieved in any manner that is consistent throughout. Consider any beam in the canonical PSF. This will intersect a number of surfaces in the scene. The tile corresponding to that beam stores this set of surface identifiers. The process of finding all the intersections of surfaces with the tiles of the canonical PSF is straightforward. If we consider the special case that all surfaces are polygons, then this is similar to polygon rasterisation and can be implemented very efficiently as it is effectively only to a resolution of $n \times n$. Given any other PSF, corresponding to direction ω_i the entire scene can be rotated such that ω_i is transformed to lie along $(0,0,1)$ - the z-axis. Rasterisation is then done in the canonical space. It is critical to choose a parameterisation over the hemisphere so that no searching is required in order to find the closest PSF direction to any arbitrary direction - since such ray lookup is a critical operation during ray tracing. The method for placing points on the hemisphere uses a recursive subdivision of a regular tetrahedron, which partitions the hemisphere into triangles. Fast constant time lookup is attained for any arbitrary point on the hemisphere in order to find the closest stored point to any given point on the hemisphere by a method described in [33]. The (finite) set of given PSF directions is denoted Ω_l and $\omega_i \in \Omega_l$ refers to a particular direction. The tiling coordinate system is referenced by (s,t), where $s,t = 0,1,\ldots,n-1$. Hence a tile is referenced as (ω_i, s, t). The set of identifiers associated with a tile is denoted by $S(\omega_i, s, t)$.

3.2. Constructing the Data Structure

The application of this data structure to ray tracing is very straightforward. First the data structure as discussed above is constructed. For each PSF_{ω_i} the scene is transformed to the canonical space, and each polygon is orthographically projected to the base of the PSF, and the tiles that it covers computed. This can be achieved by traversing each polygon edge through the tiling to compute the tiles of all the polygon edges, and then filling in the non-edge tiles that lie inside the polygon. It is important that the edge-tiling traversal algorithm allow for an 8-connected path, rather than follow a traditional DDA-style algorithm. The difference is shown in Figure 1. Such algorithms are discussed with reference to a 3D context in [9] but are easily adapted to 2D. When a tile (s,t) is found to be covered by a polygon, its polygon identifier is written into $S(\omega_i, s, t)$. By the end of this process for all PSFs the data structure is complete.

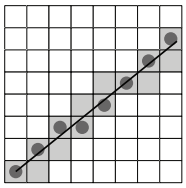

Figure 1. Finding the tiles corresponding to a polygon edge - the correct ones are the shaded tiles, a DDA algorithm would produce only the marked tiles.

3.3. Ray-Object Intersections

Now suppose that all polygon identifiers for all tiles in all PSFs have been computed, and that we require the candidate set of objects for a ray with direction ω and origin (x,y,z). Find the direction in Ω_l that is closest to ω and suppose that this is ω_j. There will be a rotation matrix M_j pre-calculated and stored with PSF_{ω_j} that rotates direction ω_j into $(0,0,1)$. Then $(x,y,z) \times M_j = (x_q, y_q, z_q)$ will be the point in the canonical space of PSF_{ω_j} that corresponds to (x,y,z) in world space. In particular the projection $(x_q, y_q, -1)$ will belong to a particular tile. The set of identifiers in that tile forms a potentially visible set for that ray.

The situation is in fact slightly more complicated. Figure 2 shows a 2D analogue of a condition where a ray would project to more than one tile. In the case of Ray_A if we only project the origin of the ray to the base of the PSF it would pick up tile 4. However, clearly the appropriate candidate set would be the union of those of tiles 2, 3 and 4. Therefore two points on each ray should be projected - the origin, and an end-point. In the case of shadow feeler rays the end-point is given by the light source position. In the case of primary or secondary rays the point on the ray that intersects with the boundary of the scene (represented by the circle in Figure 2) may be used. If the projections of these two points

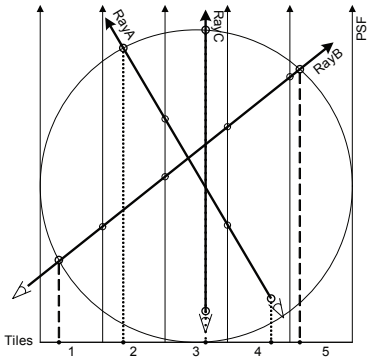

Figure 2. The rays overlap more than one tile.

are not in the same tile then the appropriate set of candidate objects is the union of all tiles that the ray traverses. For a large enough number of PSFs (l), the direction of the ray will be more closely represented and both end-points of the ray will project to the same tile.

4. Ray-Tile Traversal

Once a PSF and tile have been identified for a ray the set of polygons referenced must be searched to find the nearest ray-polygon intersection (if any). This is a critical operation. For a sufficient large number of polygons the approach presented thus far suffers considerably and it is faster to instead use a CRT-type BSP tree to traverse the entire set of polygons for every ray compared to a linear search of the polygons within the tiles. The reason is due to the logarithmic performance of the BSP tree and the linear performance in the average number of polygons per tile of the VLF-RT approach. On the other hand the VLF-RT approach does have the advantage that without any ray traversal, and simply with a lookup it is possible to reduce the search space to a very small fraction of its total size. In order to reduce the linear dependence of the timing on the mean number of polygons per tile the depth complexity along the tile (along the PSF direction) is reduced by binning the tile's polygons into a 1D BSP partition. Splitting planes are orthogonal to the beam direction and selected using a simple median split. This BSP partition along the tile can be efficiently represented by a standard 1D array. Polygons that straddle multiple BSP leaves are assigned to each of those leaves. During intersection testing, tile-BSP traversal involves recursive splitting of the ray segment corresponding to the tile across the BSP's partitioning planes and visiting nodes/leaves from near to far along the ray. This near-to-far traversal permits early termination when an intersection is found. Performance of this process is further enhanced by using a stack rather than explicit recursion code. Performance of the algorithm with the inclusion of tile-BSPs is significantly faster than using the single BSP partition of the CRT scheme. The logarithmic search time of the tile's BSP

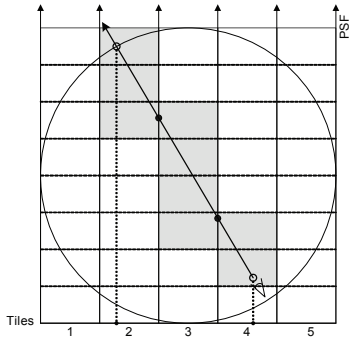

Figure 3. Ray/Tile intersection with linear traversal of the 1D tile BSP-trees. The marked BSP leaves are traversed from near to far along the ray.

can be further reduced and recursion avoided completely by traversing the BSP leaves linearly. The nearest and farthest BSP leaves can be determined directly from the parametric t-intersection values of the ray with the tile's boundaries. Given the 1D array representation of the BSP, once start and end leaves are determined, the intervening leaves can be visited near to far in a simple linear traversal. This is possible as the leaves are already correctly aligned in the required order along the ray's direction. Early termination of the process is again possible due to near to far traversal. Figure 3 shows linear intersection of tile-BSP leaves during ray-traversal of several tiles - only polygons lying in grey leaves are tested for intersection.

5. Implementation Issues

Linear traversal of the tile-BSP is very efficient as the traversal can be realised by a very simple implementation that requires no searching working order to compute the correct order in which to visit the BSP leaves. Entry and exit parametric t-values at tile-boundaries are computed incrementally as the traversal crosses into a new tile. While constructing the tile-BSP, polygons are clipped to the tile's boundary before comparing the polygon's depth range with the splitting planes - this prevents unnecessary excesses in the assigned polygon lists. When intersecting a given ray, the nearest PSF is used and the ray is projected onto this PSF using a line rasterisation algorithm similar to the one used in tile rasterisation during initialisation (see Figure 1). Also, the ray is truncated to its intersections with the scene's

boundary. The intersected tile list (e.g. tiles 2 to 4 in Figure 2) is traversed in front to back order and the ray segment for each tile is limited by the intersection of the ray with that tile's boundary. Within a tile, the ray segment is intersected using linear traversal of the tile's BSP. Note that the directional discrepancy between rays and PSF direction has been exaggerated in Figures 2 and 3 to illustrate the issues more clearly; in practice only a small number of tiles are intersected depending on tile resolution (n) and the number of PSFs (l). The ideal situation is when the ray direction matches the PSF direction very closely - only one tile need be considered in that case (Ray_C). There are obviously no tile crossings in this case and a simple linear traversal of the tile-BSP for the entire ray-segment is all that is required.

5.1. Coherent Ray Tracing Implementation

The single ray CRT approach implemented follows [37]. It uses the spatial median for BSP splitting planes, and the optimised triangle layout described in [36]. The BSP traversal's recursion is rolled out using an explicit stack, and the intersection kernel is directly inlined using a macro. All other framework code such as ray creation, shading, etc. is shared between the two implementations.

6. Results

We compared performance of VLF-RT with the CRT implementation described above. For each method we used parameters that were fastest for that method. In the case of the BSP tree for CRT, a parameterisation must be chosen - specifically the maximum depth of the tree, and the ideal maximum number of polygons allowed per leaf-node (subject to the maximum depth). In order to determine these parameters we ran a series of pre-test experiments with the scenes described in following sections, in order to determine the best combination of depth and leaf size - these were then used in the comparative performance tests. Similarly, the ideal depth of the tile-BSP was determined for VLF-RT using the same strategy. We could also vary the number of PSFs and tile resolutions - already knowing that higher resolution would result in better performance. All timings were obtained on a 2.8Ghz Pentium 4 with 2GB of system memory.

6.1. Performance Comparison

We are interested in ray-traversal speed as the number of polygons increases. For this purpose we use an artificial scene with uniformly distributed random triangles (Figure 4). Figure 5 shows the frame time, averaged over a few frames for increasing numbers of triangles viewed from a short camera path through the scene. Again, results for several parameters were obtained and the results from the best parameterisation presented.

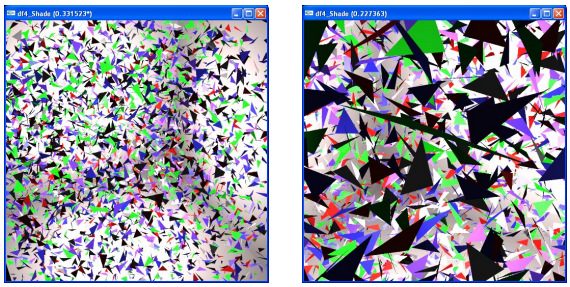

Figure 4. Images of the random scenes. The 9K(left) and 16K(right) scenes are viewed from two of the viewpoints used to determine performance.

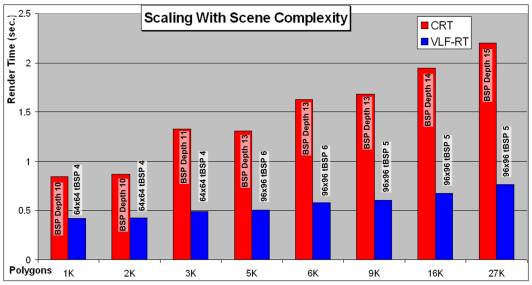

Figure 5. Average time to render a 512×512 image of scenes with increasing numbers of uniformly distributed random triangles.

6.2. Walkthrough

In the previous section we discussed performance for scalability of the VLF-RT for a scene composed of random polygons. We now consider a walkthrough of a more realistic 'Classroom' scene (Figure 7). The scene consists of 51,208 polygons and 4 point light-sources. Two shading methods were used: a simple OpenGL like method that casts only primary rays and shades points based on local diffuse texture and distance/direction to the light sources; and Whitted-type shading that uses secondary reflection and shadow rays. Comparative frame render times for a walkthrough of the classroom scene along a fixed camera path comprising 420 frames for the CRT and VLF-RT methods are shown in Figure 6. These results are for optimum parameters for the CRT (BSP depth 23) and VLF-RT (128×128 tiling, tile-BSP depth 5) - the number of PSFs

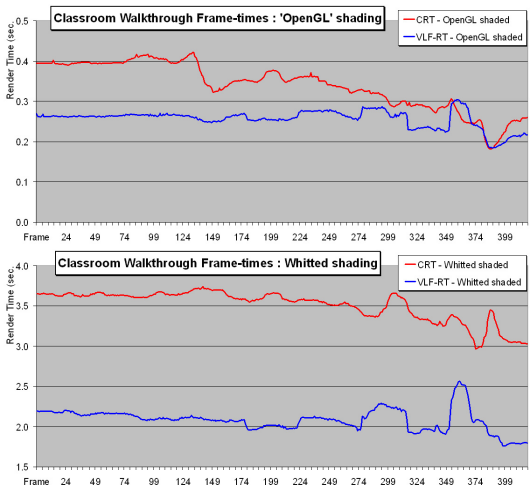

Figure 6. Frame render times for the Classroom scene.

was however kept fixed at 513. Image size rendered per frame was 256×256.

Figure 7. The classroom scene with Whitted and OpenGL-like shading.

7. Application to Ambient Occlusion

When illuminating a scene/image the computation of diffuse inter-reflection is often very expensive. Typically only a few iterations of diffuse transfer are computed – possibly leaving some fragments of the scene completely dark. To counter this an ambient term (usually a constant) is added to the shading computation to account for later iterations of diffuse transfer – this constant ambient term can however make the results appear artificial. Ambient Occlusion [26] is a method used to make images look more physically plausible. Instead of using a constant ambient term, the visibility at a point is sampled and the point is shaded based on the number of samples not immediately occluded. Illumination is usually from a skylight or sky-dome over the object and results in shading that simulates soft shadows. Ambient Occlusion at a point can be computed by casting shadow rays into the scene to integrate visibility over the hemisphere around the normal at that point. Directions selected for this integration are uniformly sampled over the hemisphere to avoid bias and a large number of rays are cast to reduce noise. The VLF-RT data structure has several

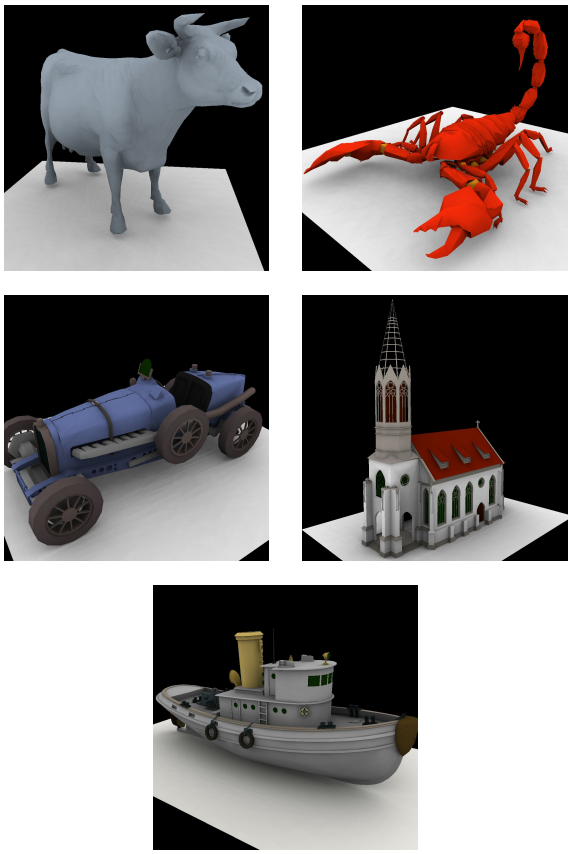

Figure 8. Ray traced images with Ambient Occlusion shading.

advantages in the computation of Ambient Occlusion at a point. As mentioned, the directions selected for the rays cast from a point have to be uniformly sampled over the hemisphere over that point. These directions could equally be selected from the set of PSF directions as these represent a uniform distribution of directions. When a shadow ray is cast into the scene from a point and lies along a PSF direction, both endpoints of that ray fall on the same tile (see Figure 2, Ray_C). In this case, the ray would not cross any tile boundaries and would simply need to traverse the associated tile BSP leaves from near-to-far. This is a very simple process as the data structure is optimal for a traversal of this nature. Image render times for various scenes (Figure 8) were obtained using optimum parameters for BSP-

Scene	CRT	VLF-RT	Relative Speedup
Cow (5.8K polys)	38.01 secs (BSP depth 16)	15.15 secs (tBSP depth 3)	2.51x
Scorpion (10K polys)	32.19 secs (BSP depth 19)	17.88 secs (tBSP depth 3)	1.80x
Bugatti (11K polys)	43.54 secs (BSP depth 17)	21.63 secs (tBSP depth 4)	2.01x
Church (16K polys)	25.50 secs (BSP depth 19)	14.07 secs (tBSP depth 4)	1.81x
Tugboat (34K polys)	44.36 secs (BSP depth 23)	30.65 secs (tBSP depth 4)	1.45x

Table 1. Comparative performance for Ambient Occlusion shading.

tree depth in CRT and tile-BSP depth in the VLF-RT. The number of PSF directions and tile resolution per PSF were however kept fixed at 513 and 128×128 respectively. Table 1 shows the time taken to render Ambient Occlusion shaded images of several scenes for both methods.

A total of 513 rays were cast into the scene from each surface point to integrate visibility over the hemisphere. An image size of 256×256 was rendered for each scene. These results show a significant advantage for the VLF-RT method for Ambient Occlusion type shading that requires integration of visibility over the hemisphere. For the *Church* scene the CRT traces $\sim$1.3 Mrays/sec whereas the VLF-RT traces $\sim$2.4 Mrays/sec.

8. Conclusions

In this paper we have introduced a new ray-traversal method, and illustrated its application in classical ray tracing and applications of ray tracing to efficient ambient occlusion shading. The method relies on a very fast lookup to obtain a candidate set of polygons for any ray, and then these polygons may be traversed by a BSP tree with partitioning along only one axis. The results suggest that this method performs better than the single ray CRT method with a spatial median BSP tree for a variety of scenes.

Obviously the CRT implementation used is far from optimal. Performance can be significantly increased by using a different splitting plane heuristic [17] and ray bundling [37], and it would be interesting to see how the approach presented here fares if these optimisations were introduced to both algorithms; this is an area of ongoing research.

In this paper we have only discussed walkthrough applications. However, real-time ray tracing also demands the possibility of dynamic changes to objects. This is easily achievable with the VLF-RT method [24, 23]. When an object is transformed it must be first deleted from the data structure, then its geometry transformed and inserted back into the data structure. Once these operations have been carried out the ray tracing can be used to render the next frame as usual. In order to delete an object from the VLF-RT data structure, all PSFs are visited, and the object rasterised into the tiling coordinate system as usual, except that in this case the identifiers of the object are removed rather than added. Then the polygon's geometry is transformed, and reinserted into both the tiling and BSP structures.

Acknowledgements

This research was funded by EPSRC GR/R13685/01. Thanks to Ingo Wald and Carsten Benthin for helpful suggestions on real time ray tracing.

References

[1] J. Amanatides. Ray tracing with cones. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 129–135, 1984.

[2] A. Appel. Some techniques for shading machine renderings of solids. *AFIPS 1968 Spring Joint Computer Conference*, 32:37–45, 1968.

[3] J. Arvo and D. Kirk. Fast ray tracing by ray classification. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, volume 21, pages 55–64, July 1987.

[4] C. Benthin, I. Wald, and P. Slusallek. A scalable approach to interactive global illumination. *Computer Graphics Forum (Proc.of Eurographics)*, 22(3):621–630, 2003.

[5] E. Camahort, A. Lerios, and D. Fussell. Uniformly sampled light fields. In *Rendering Techniques '98 (Proceedings of Eurographics Rendering Workshop '98)*, pages 117–130, 1998.

[6] N. A. Carr, J. D. Hall, and J. C. Hart. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, 2002.

[7] Y. Chrysanthou, D. Cohen-Or, and D. Lischinki. Fast approximate quantitative visibility for complex scenes. In *Pro-*

ceedings of Computer Graphics International '98 (CGI '98), pages 220–227, 1998.

[8] J. G. Cleary and G. Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer*, 4(2):65–83, 1988.

[9] D. Cohen and A. Kaufman. Scan conversion algorithms for linear and quadratic objects. In A. Kaufman, editor, *Volume Visualization*, pages 280–300. IEEE Computer Society Press, Los Alamitos, CA, USA, 1990.

[10] T. Foley and J. Sugerman. KD-tree acceleration structures for a GPU raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, 2005.

[11] A. Fujimoto, T. Tanaka, and K. Iwata. ARTS: Accelerated ray-tracing system. *IEEE Computer Graphics & Applications*, 6(4):16–26, Apr. 1986.

[12] A. S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics & Applications*, 4(10):15–22, 1984.

[13] A. S. Glassner. *An Introduction to Ray tracing*. Academic Press, San Diego, CA, USA, Jan. 1989.

[14] J. Goldsmith and J. K. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics & Applications*, 7(5):14–20, May 1987.

[15] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The lumigraph. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 43–54, 1996.

[16] E. A. Haines and D. P. Greenberg. The light buffer: A shadow-testing accelerator. *IEEE Computer Graphics & Applications*, 6(9):6–16, 1986.

[17] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University in Prague, Prague, Apr. 2001.

[18] V. Havran, J. Bittner, and J. Žára. Ray tracing with rope trees. In *Proceedings of 13th Spring Conference on Computer Graphics*, pages 130–139, 1998.

[19] V. Havran, T. Kopal, J. Bittner, and J. Žára. Fast robust bsp traversal algorithm for ray tracing. *Journal of Graphics Tools*, 2(4):15–23, 1997.

[20] F. W. Jansen. Data structures for ray tracing. In *Proceedings of a workshop (Eurographics Seminars on Data structures for raster graphics)*, pages 57–73. Springer-Verlag, Inc., NY, USA, 1986.

[21] M. R. Kaplan. Space-tracing: A constant time ray tracer. In *SIGGRAPH '85 State of the Art in Image Synthesis seminar notes*, volume 19, pages 149–158, July 1985.

[22] T. L. Kay and J. T. Kajiya. Ray tracing complex scenes. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, volume 20, pages 269–278, NY, USA, 1986.

[23] P. Khanna, J. Mortensen, I. Yu, and M. Slater. Fast ray tracing of scenes with unstructured motion. Technical report, University College London, 2004.

[24] P. Khanna, J. Mortensen, I. Yu, and M. Slater. A visibility field for dynamic ray tracing. Technical report, University College London, 2004.

[25] B. Kwon, D. S. Kim, K.-Y. Chwa, and S. Y. Shin. Memory-efficient ray classification for visibility operations. *IEEE Transactions on Visualization and Computer Graphics*, 4(3):193–201, July 1998.

[26] H. Landis. Production-ready global illumination. In *SIGGRAPH 2002 Course Notes*, 2002.

[27] M. Levoy and P. Hanrahan. Light field rendering. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 31–42, 1996.

[28] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 101–108, Aug. 1997.

[29] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002.

[30] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. *ACM Transactions on Graphics*, 24(3):1176–1185, 2005.

[31] J. Shade, S. Gortler, L. wei He, and R. Szeliski. Layered depth images. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 231–242, 1998.

[32] G. Simiakakis and A. M. Day. Five-dimensional adaptive subdivision for ray tracing. *Computer Graphics Forum (Proc.of Eurographics)*, 13(2):133–140, 1994.

[33] M. Slater. Constant time queries on uniformly distributed points on a hemisphere. *Journal of Graphics Tools*, 7(1):33–44, 2002.

[34] M. Slater, J. Mortensen, P. Khanna, and I. Yu. A virtual light field approach to global illumination. In *Proceedings of Computer Graphics International (CGI 2004)*, pages 102–109. IEEE Computer Society Press, June 16-19 2004.

[35] K. Sung and P. Shirley. Ray tracing with the bsp tree. In D. Kirk, editor, *Graphics Gems III*, volume IBM version, pages 271–274. Morgan Kaufmann Publishers, 1992.

[36] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

[37] I. Wald, C. Benthin, M. Wagner, and P. Slusallek. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001)*, volume 20, pages 153–164, 2001.

[38] I. Wald, T. Kollig, C. Benthin, A. Keller, and P. Slusallek. Interactive global illumination using fast ray tracing. In *Rendering Techniques 2002 (Proceedings of the 13th Eurographics workshop on Rendering)*, 2002.

[39] I. Wald, T. J. Purcell, J. Schmittler, C. Benthin, and P. Slusallek. Realtime ray tracing and its use for interactive global illumination. *Eurographics 2003 STAR Report*, 22(3), 2003.

[40] B. Walter, G. Drettakis, and S. Parker. Interactive rendering using the render cache. In *Rendering techniques '99 (Proceedings of the 10th Eurographics Workshop on Rendering)*, volume 10, pages 235–246, Jun 1999.

[41] G. Ward and M. Simmons. The holodeck ray cache: an interactive rendering system for global illumination in nondiffuse environments. *ACM Transactions on Graphics*, 18(4):361–368, 1999.

[42] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.