Eurographics Italian Chapter Conference (2008)
V. Scarano, R. De Chiara, and U. Erra (Editors)

A Survey on Exploiting Grids for Ray Tracing

Biagio Cosenza

ISISLab, Dipartimento di Informatica ed Applicazioni "R.M. Capocelli"
Università degli Studi di Salerno, Italy
cosenza@dia.unisa.it

Abstract

Grid is one of the first data structure introduced at the very beginning of computer graphics. Grids are used in several applications of computer graphics, especially in rendering algorithms. Lately, in ray tracing dynamic scenes, grid has received attention for its appealing linear time building time. In this paper, we aim to survey several aspects behind the use of grids in ray tracing. In particular we investigate grid traversal algorithms, building techniques and several approaches for hierarchical grids.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques: Graphics data structures and data types I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism: Ray tracing.

1 Introduction

Several works (e.g., [Hav00]) have showed the significance of data structures for rendering algorithms and in particular for ray tracing [Whi80]. Ray tracing is a widely used algorithm for rendering images aiming at an high realism. Ray tracing requires a big amount of computational power and therefore, a lot of research has been proposed to speed up ray tracing algorithm by using additional acceleration data structures, such as grids, octrees, bounding volume hierarchies, or kd-trees (see [Hav00]). In particular it has been showed that, in a general kd-tree implementation for static scenes, about 60% of time in ray tracing algorithm is spent in tree traversal [Wal06].

Although, ray tracing used to be considered not suitable for interactive applications, recently, several works [Wal06, RSH05, BSW06] have showed that it is possible to achieve interactive performance, at least for static scene, by using several traversal optimization, such as packet and frustum traversal.

The choice for a efficient data structure become quite intricate if one has to consider dynamic scenes. In that case not only traversal time, but also building and updating time, have to be considered in the choice of the acceleration data structure. Indeed, as showed in [WMG*07], ray racing of dynamic scenes has shifted the attention from traversal time to build and update time.

In this context, grid data structures [Gla84] becomes an appealing data structure: although kd-trees outperforms grids with respect traversal time, they are not suitable for dynamic scenes, due to their high cost of rebuilding/updating. As an example, the surface area heuristics, required to build fast-traversal kd-trees [WH06], may require seconds to minutes to build a kd-tree for quite complex scenes. Therefore kd-trees can be efficiently used only for static scenes. This restriction limits the utility of kd-tree, as data structure for ray tracing, for many interactive scenarios, such as visual simulation, animations, and interactive games. While some efforts have focused on extending kd-trees to dynamic scenes [WMG*07], from the best of our knowledge, they are limited to hierarchical motion or require advance knowledge of the scene, and therefore they are unsuitable for pure dynamic animations that require unstructured motion.

Grid data structures, introduced by [Gla84], divides the scene-space in uniform voxel spaces. In contrast with kd-trees and other adaptive data structures, due to their linear time build algorithm, grids can be created from scratch and updated at every frame with interactive performance [RSH00] (at least for moderate sized scenes). Consequently, even if grids provide higher traversal time than kd-trees, they have been considered attractive for dynamic scenes because of theirs faster building time.

Grid data structures are popular not only for rendering analytical objects of a scene, but also for different scenarios such as particle-based simulations, computational fluids dynamics and volume rendering .

Assumptions on grid. Formally, a grid is a spatial data structure that divides the scene-space in constant size voxels, named cells. We assume to have a cubic main voxel that represents the whole scene. A grid equally subdivides the main voxel along each dimension. Therefore, a grid can be seen as

a collection of $M = m^3$ cubic grid cells, where m is the number of grid cells in one dimension. Whereas the main scene voxel is not cubic, we can still subdivide the main voxel in cubic voxels using a different number of grid cell for each dimension. We have $M = m_x \cdot m_y \cdot m_z$ where m_x, m_y and m_z represents the number of grid cells for each dimension.
Each cell is associated to a subset of the primitives that describes the scene. Of course, a primitive can fill more than a cell.

2 Traversal algorithms

In ray tracing, a traversal algorithm for a grid return all the voxels (cells) traversed by the ray. Formally, starting from a parametric representation of a ray $f(t) = \vec{o} + t \cdot \vec{d}$ where $\vec{o}$ and $\vec{d}$ are the origin and direction vector, a traversal algorithm returns all the cells traversed by the ray. An interval $[t_{min}, t_{max})$ (usually $[0, +\infty)$) is associated to a ray, and only intersections within this interval are considered. Since we are looking for the closest primitive intersection, grid traversal stops as soon as an intersection is found.
Traversal algorithms in grid are similar to rasterization. Bresenham 2D line rastering algorithm [Bre65], for example, produce visually acceptable rasterized line in the shortest time. However, it does not find all the pixels traversed by the line segment. Therefore a naive 3D extension of Bresenham algorithm is not suitable as a grid traversal algorithm.

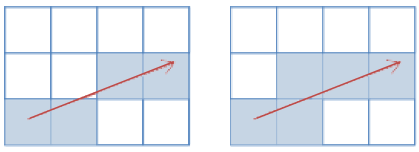

Figure 1: (left) Bresenham cells. (right) Traversed cells.

Traversal algorithm can be classified with respect of several aspects.

Discrete integer traversal. A traversal algorithm generate a sequence of cell traversed by a ray directly. We can distinguish integer based algorithms, which assume that each ray has a start and an end point, referred with integer coordinates, (which corresponds to the first and the last cell to visit). On the other hand, there are algorithms where rays have a starting point and direction expressed with floating point variables [Sra95]. In the latter case one has to pay attention to the precision, reducing the numerical errors to the minimum.

Ray aggregation strategies: exploiting spatial coherence. Ray tracing methods usually use a geometric representation of the 3D scene. Each geometric primitive supported by a ray tracer implements a ray-primitive intersection algorithm, and the purpose of the acceleration data structure (such as grid) is to minimize the number of intersection tests required.
Recently, an important class of techniques exploit spatial coherence to provide fast traversal performance. For example in grids, close rays go through similar path of cells. This coherence is effective especially for primary eye rays, hard shadow rays, soft shadow rays and many other kind of secondary rays. There are two main strategies to exploit this coherence:
Ray aggregation, where a group of rays is traced as a single unit (a packet), and
Beam tracing, where rays are considered only in a final step.
The first strategy is the most used by current interactive systems. It is used with several data structures (i.e., kd-tree [Wal06]), allows the use of register SIMD to process four rays in bundle, and can be combined with frustum- and interval- arithmetic techniques to avoid traversal steps and intersection tests by exploiting conservative bounds of the group of rays [RSH05].
Beam strategies does not represents explicitly rays until a final sampling step. Overbeck et al. showed that *beam tracing* is competitive with frustum based tracers, at least for eyes and shadow rays [ORM07]. However, this work focus on a kd-tree based data structure and, from the best of out knowledge, there are no known implementations of grid based beam tracer.

Traversal time. Traversal algorithm should be fast. A common cost model used for grid supposes that is possible to identify, for each algorithm, two constants [ISP07]:
T_{setup} : time to setup traversal algorithm;
T_{step} : time spent to advance from a cell to its neighbor (without checking intersection).
The performance of a traversal algorithm depends directly from this two conflictual values. For example, an algorithm having a low T_{setup} usually performs better for low resolution grids, whereas an high T_{step} provide bad performance for larger grids.

2.1 Single ray traversal

Several grid traversal algorithms have been historically developed to determinate the cells traversed by a ray in a grid. Algorithms for grid traversal are quite similar to rasterization algorithms (such as Bresenham line draw algorithm [Bre65]). Many three dimensional traversal algorithms can be directly derived from an equivalent line drawing algorithm. Indeed, most of them are variants of the Digital Differential Analyzer (DDA).
We are going to give a brief outline of the most important grid traversal algorithms. Our analysis will focus in the two dimensions case. Anyway, extensions to three dimensional space is usually simple and straight.

Cleary and Wyvill algorithm [CW88]. Cleary and Wywill showed that the next cell calculations can be done in two or three integer operations. We henceforth indicate with vertical (resp. horizontal) walls the vertical (resp. horizontal) axis belonging to the grid. When a ray enters in a new cell, it can traverse an horizontal wall (i.e., from the bottom to the top) or a vertical wall (i.e from left to right). Considering only the passage through vertical walls, let δx the distance along the ray between such crossings. Similarly, δy is the constant distance between successive crossings of horizontal walls. In order to find the cell traversal order, we need to determinate the sequence of crossing type (horizontal or vertical). This can be done by keeping two variables, dx and dy, which record the total distance along the ray, from the origin to the next crossing of a vertical or horizontal wall respectively. If $dx < dy$, then the next crossing is over a vertical wall, and the next cell is the horizontal neighbor cell (see Figure 2). Determinated the next cross type, traversal algorithm update dx ($dx = dx - \delta x$). Similar operations are done on dy if $dx > dy$. Extending the algorithm to the three dimensions simply requires the addition of the appropriate dz and δz variables, and finding the minimum of dx, dy and dz at each step.

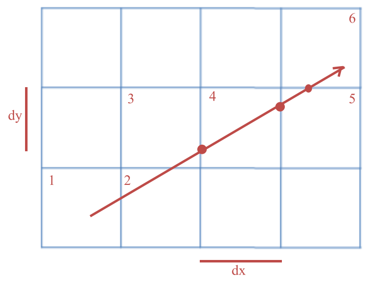

Figure 2: An example of Amanatides and Woo traversal algorithm with the corresponding cell traversal order. The Figure shows dx and dy after tr

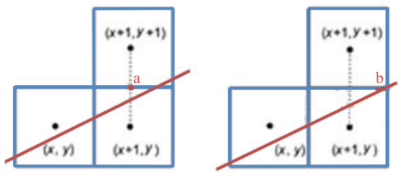

Figure 3: Bresenham (left) and Liu (right) thresholds.

Amanatides and Woo algorithm [AW87]. Amanatides and Woo traversal algorithm differs from Cleary and Wywill's algorithm because it is based on floating point values. Starting from the parametric equation of the ray, with $t \geq 0$, the algorithm breaks down the ray into intervals of t, each of which spans one cell. The floating point variable dx (dy) represents the value of the parameter t where the ray hit the horizontal (vertical) wall. If the ray direction is normalized, dx (dy) is also the distance from origin to the wall hit point. the three dimensional version of this traversal algorithm requires, in a traversal step, only two comparisons and one addition (floating point operations).

Modified Bresenham algorithms. As discussed above, standard Bresenham line drawing does not consider all the cells traversed by a ray [Bre65]. Indeed, chosen the driving axis, e.g., the x axis and starting from the cell (x,y), Bresenham's algorithm choices only one cell between $(x+1,y)$ and $(x+1,y+1)$, never both.
Zalik et al. solve this problem proposing a code-based traversal algorithm [ZCO97]. This algorithm has two phases: first it determines the cells traversed by Bresenham's algorithm; then it examines the relationship between two successive cell to determinate the remaining cells. However, this approach has some drawbacks: a higher traversal time and an elaborate implementation.
A more efficient extension of Bresenham algorithm is proposed by Liu et al. [LZY04]. Bresenham's algorithm, chosen the driving axis, at each step selects a cell to traverse. The error e tells the difference between the chosen integer cell and the real coordinate of the line. The error e is compared against a threshold ($1/2$ in Bresenham) and then a cumulative value of the line segment slope is computed($e = e + k$ and $k = dx/dy$).
Liu extension, instead, determines the intersection point of the line with the two walls, and then compare the error e against a threshold $= \frac{1}{2} - \frac{dy}{2dx}$. If e is greater than the threshold, both $(x+1,y)$ and $(x+1,y+1)$ are visited. Thereafter, following the Bresenham approach, the algorithm is adjusted to integer arithmetic. The three dimensional case is quite similar, and identify two possible paths from the cell (x,y,z) to $(x+1,y+1,z+1)$ (cfr. [LZY04]). Liu algorithm is faster even on modern processors, where floating point arithmetic is almost as efficient as integer arithmetic.

Other approaches. Line drawing algorithms raised historically a lot of interest by the Computer Graphics research community, and several approaches and solutions have been proposed.
For example, in *symmetric* algorithms (see [Wyv90]): since lines are symmetric around the center, it is possible to draw two (symmetric) pixel at time. Notice that, in symmetric algorithms, the traversal algorithms starts from the midpoint (pixel) and go trough the two outermost points, or viceversa, starts from the outermost pixels and go trough the midpoint, whereas, in general, traversal algorithm start from the origin and eventually stop when an intersection is found.
Other line drawing algorithms use several patterns to draw more pixels at once. For example, instead of draw a pixel at once, they draw a pattern of three pixels, taken by four different precomputed patterns [Wyv90]. From the best of our knowledge, uses and benefits of symmetric and pattern-based algorithms, in the context of grid traversal, are not investigated yet.
3D Digital Differential Analyzer (3DDDA) traversal is applicable for several data structure traversal algorithms. An interesting view of 3DDDA issue is done by Fujimoto et al. [FTI]. They provides an environment for ray tracing named SEADS. The paper analyzes uniform grids, but it also discusses the traversal algorithm for octrees. They argue that 3DDDA traversal algorithm for octrees is not as efficient as expected. They discuss the 3DDDA octree traversal in details and compare it with the uniform grid 3DDDA traversal. They also provides experimental results that show that uniform grid 3DDDA traversal is faster than Glassner octree traversal [Gla84].

2.2 Group of rays traversal

Unfortunately, 3DDDA like algorithms used for single ray grid traversal can not be easily improved using standard

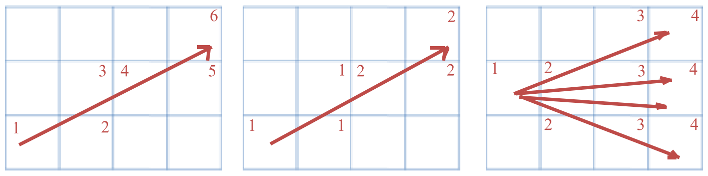

Figure 4: Traversal technique strategies: (left) single step , (center) pattern based , and (right) slice by slice.

traversal optimization techniques (such as packet and frustum traversal). The main issue of extending traversal from a single ray to a group of rays concerns about the cell traversed by a group of rays. If the path of two close rays is quite similar, they can differs in a restricted number of cell that a ray traverses and another does not. Single ray algorithm can chose only one cell at a time to step into, whereas different rays can disagree on the next cell to be traversed. A naive solution to this problem is to split rays in different sub-packets with the same traversal decision. However, rays that have diverged in a cell (and then are split in different sub-packets) may traverse other common cells later on. This loose of coherence can be avoided by re-merging sub-packets. Nevertheless, splitting and merging packets of rays are expensive operations and thus, this can not be a practical solution.

Slice by slice algorithm. Ize. et al. [WIK*06] have solved this problem by abandoning 3DDDA like algorithm in favor of a new *slice by slice* traversal algorithm for group of rays. The algorithm, known as CGT (Coherent Grid Traversal), traverse the grid slice by slice rather than cell by cell, avoiding expensive merge and split operations.

The algorithm, given a set of coherent rays (rays that spans an angle of less than $\pi/2$), computes the packet's bounding frustum that is traversed through one slice at a time. The major dominant axis is taken by selecting the dominant component of the direction of the first ray, and the remaining two dimensions determinates the slices. For each slice, a incrementally frustum's overlap with the slice is performed, which determinates the cells actually overlapped by the frustum. Each ray packet have four corner rays which defines the frustum boundaries. An important feature is that frustum traversal step can be done efficiently by using SIMD instructions.

Although the number of overall ray-primitive intersection test can be higher, because the packet can traverse cells that some rays does not intersect, in practice ray coherence easily compensate this overhead.

Ize et al. also extends this algorithm to hierarchical grids. They show significant test results in dynamic scenes, and competitive with Intel MLRT System based on kd-tree for static scene (actually the best known for static scenes) [RSH05]. An interesting consideration about CGT concerns about scalability with screen resolution. Since higher resolutions enable larger packets, we generally see sublinear scaling in screen resolution.

3 Building algorithms

Ize et al. [ISP07] defines a general approach to build flat grids: based on some assumption on geometry distribution they provides good estimation for grid resolution. Furthermore, the also shows a theoretical analysis for hierarchical grids. The analysis focus on two kind of scene models, the first one assumes that compact triangles are comparable to points while the second considers long-skinny triangles as lines. An interesting property of grids is that the building process is easy to parallelize [IWRP06].

In the following we consider a cubic main voxel containing N geometric primitives describing the scene, and a given value for m. Two building algorithms have been proposed [IWRP06]:

Sort-last building algorithms: for each primitive p, the algorithm determinates the grid cells which contains p or a portion of p;

Sort-first building algorithms: for each grid cell g, the algorithm determinates the primitives which have a non-empty intersection with g.

In this papers we refers to sort-last building algorithms as general grid building algorithm. The algorithm can use an exact test or a faster but imprecise axis aligned bounding box test. The latter is preferred in the context of interactive real time environments.

Algorithm 1 A general grid building algorithm

```
1: $m \leftarrow determinatem()$
2: split grid in $m^3$ cells
3: for all primitive $p \in scene$ do
4:     $box \leftarrow p.boundingBox$
5:     insert $p$ in grid cells from $box.min$ to $box.max$
6: end for
```

A general method, to build grids, is shortly summarized in the algorithm *determinatem* (see Algorithm 1).

If we suppose that each primitive covers a constant number of grid cells and assuming that *determinatem* is $O(N)$, then one can easily verify that the building algorithm has linear time complexity.

3.1 Grid building: preliminaries

In the following we will give some basic assumptions: We assume that rays are uniformly distributed in the space, hence each cell has the same probability to be reached by a ray. A similar assumption is done by Surface Area Heuristic (SAH) heuristics; we assume to have a single ray 3DDDA like traversal algorithm, and that it is possible to determinate the time required by atomic operations:

T_{setup}: time to do the initial intersection with main voxel bounding box and setup traversal;
T_{step}: time spent to advance from a cell to its neighbor (without checking intersection);
T_{inter}: time required to check a ray-primitive intersection.

We also suppose that each primitive covers a bounded number of grid cells. In the following we denote by N_m the number of intersections between primitives and cells for a given subdivision factor m. The number of cells covered by a primitive depends by the scene model considered. For instance, if the model is based on lines, a primitive covers roughly m cells ($N_m \approx N \cdot m$), where in a point-based model a primitive covers exactly 1 cell ($N_m \approx N$).

3.2 A traversal time cost-based model

In this section we shift out attention in the choice of the subdivision resolution. We will show that this problem is not trivial and it is strongly influenced by the scene model. The choice of the subdivision factor (i.e., m) affects traversal time and memory space occupation.

Defining a traversal time cost based approach, during the building phase, is a common approach in ray tracing.

In particular, in the context of the kd-tree building, split planes are chosen by using a Surface Area Heuristic (SAH) in order to evaluate possible candidate planes. However, exact SAH build algorithms have $O(n \log n)$ time complexity, which is considered too high for interactive purpose. A first complete cost model for grid has been introduced in [CW88].

Ize et al. [ISP07] introduced a clear and simple cost model for grid. Following the above assumptions, a ray reaches m cells on average. The average traversal time T, in a flat grid is:

$$T = T_{setup} + mT_{step} + \mu T_{inter}$$

where $\mu = N_m/m^2$ is the average number of primitives contained (or partially contained) in m cells.

The value of T strongly depends on the scene model, on the value of the subdivision factor m and on primitives cell occupation (N_m).

In point based models, each primitive is contained in exactly one cell (i.e.,$N_m = N$), and therefore each cell contains N/m^3 primitives on average. Since, we are looking for the number of primitives contained in m cells, we have $\mu_m = N/m^2$. On the other hand, considering line based models, each primitive is partially contained into m cells ($N_m = N \cdot m$), hence $\mu_m = N/m$.

If we have enough information about the scene model, it is possible to determinate the value of m which minimizes T [ISP07]. In particular, if we consider point based models, then[†]

$$M = N \cdot \frac{2T_{inter}}{T_{step}} = O(N). \qquad (1)$$

† We recall that T_{inter} and T_{step} are constant.

For line based models

$$M = \left(N \cdot \frac{T_{inter}}{T_{step}}\right)^{\frac{3}{2}} = O(N^{1.5}). \qquad (2)$$

Commonly used models behave between this two opposite ones. Anyway, there is a remarkable asymptotic difference between the two models. Nevertheless, the use of these theoretical results requires a knowledge of scene model type.

Mailboxing. Mailboxing [AW87] is a common technique that avoid multiple identical ray-primitive intersection tests. A revised approach that include mailboxing requires to redefine N_m as the average number of *distinct* primitives contained (or partially contained) in m cells. For sake of simplicity we do not consider this approach in this cost-based model.

3.3 Choosing the resolution

The choice of m is fundamental in traversal performance. Ize et al. [ISP07] work requires an *a priori* estimate of models occupation.
It is possible to determinate an *exact* value of N_m for a given m by performing a primitive counting during the building. This means that, for a given m, it is possible to evaluate N_m in linear time. A guess and check building algorithm can test several value of m in linear time and then select the one which minimize the cost function T. The theoretical analysis give us a bounded interval where look for a good value of m. As an example, assuming that $T_{setup} \approx 1.0, T_{step} \approx 0.1$ and $T_{inter} \approx 0.5$, then $m \in \left[(10N)^{\frac{1}{3}}, 5N^{\frac{3}{2}}\right]$.

Sampling-based building algorithm. The idea of guess and check algorithms is to sample the cost function in order to build a efficient grid. This idea is not new. For example, in the context of SAH kd-tree build [WH06], Popov et al. [PGSS06] provide theoretical and practical results regarding conservatively sub-sampling of the SAH cost function in kd tree. Hunt et al. [HMS06] define an adaptive error-bounded heuristic based on a scanning-based algorithm for choosing kd-tree split planes that are close to optimal with respect to the SAH criteria. Similar works exploit SAH build in other data structures such as BVH.

Notice that our cost based grid building is a SAH algorithm, where a grid resolution is weighted using the number of primitive contained in a cell. Because grid cells are voxels having all the same surface area, the probability that a ray hits a cell is equal for all the cells. Where in kd tree SAH heuristic evaluates the cost of a plane candidate, instead in grid we evaluate a cost of a grid resolution (our m).

In this context we can use theoretical results to bound sampling and evaluate the cost function for several values of m (in our case $m \in \left[(10N)^{\frac{1}{3}}, 5N^{\frac{3}{2}}\right]$). A naive approach can sample this interval in a constant number of points, selecting the resolution in linear time. Hence we have a linear overall building time.

4 Hierarchical grids

One of the most issues on designing a grid is that it lacks to adapt on geometry distribution. Adaptive subdivision often works better for complex scenes with uneven geometry distribution, but generally are harder to build. Hierarchical grids (see e.g. [RSH00]) overcome this problem by using a recursive data structure. They provide a trade-off between traversal and building time. There are several ways to build hierarchical grids. The main idea is to subdivide some regions of space finer than others, and thus quickly traverse empty spaces

We are going to give a brief outline of the most important ways to build hierarchical grid.

Loosely nested grids. Cazals et al. [CDP95] proposes a hierarchical grid building which is able to handle very complex scene. In particular they propose a four step algorithm:
- organizes primitives of the scene in subset of similar size;
- for each group of primitives, group the neighbors into clusters;
- construct a grid for each cluster;
- construct a hierarchy of these grids.

The proposed data structure can be seen as a loosely nested recursive grid. Filtering and clustering steps effect a bottom-up construction, in strong constrast with all other methods that subdivide their structure adaptively in a top-down manner.

Klimaszewski and Sederberg [KS97] propose an adaptive grid data structure. They suggest to build *local* grids that acts as a bounding box in densely populated areas. They use a fast bottom-up building algorithm:
- for each primitive, surrounds it with a bounding box (unstructured grids);
- for all bounding boxes, merge close boxes (structured grid);
- for all remaining boxes, insert the box into tree using a minimum surface criterion;
- for all bounding boxes in hierarchy, build a local grid.

Again, this data structure can be seen as a loosely nested recursive grid.

Both works seem to have a greater build cost against general bottom-up strategies.

Multi-resolution grids. Jevans and Wyvill [JW89] illustrates a recursive hierarchical subdivision algorithm for grids. All the primitives are initially inserted into a single voxel. Thera are two kind of voxel: leaf voxel, which contains a list of objects inside that voxel and internal voxel, which maintains a voxel grid subdivision. They use an hashtable which maintains each non-empty internal voxel. They propose several build methods:
- octree like, setting the size of voxel grids to 2 on a side;
- setting a maximum depth of the subdivision tree;
- fixing the resolution of the voxel subgrids to a constant value;
- variable (adaptive) voxel resolution.

Notice that build approach is top-down This integration of regular and adaptive spatial subdivision methods allows images consisting of large regularly distributed objects and small dense objects to be ray traced efficiently. The parameters controlling the coarseness of the voxel grid, depth of adaptive subdivision trees, and maximum number of polygons per voxel are tailored and their effects on execution time, subdivision time, and memory use are measured.

Macrocell or multigrids. Parker et al. [PPL*99] use a simple hierarchical optimization to a base uniform grid, called macrocell. Macrocells superimpose a second, coarser grid over the original fine grid, such that each macrocell correspond to an $AxAxA$ block of original grid cells. Each macrocell stores a boolean flag specifying whether any of its corresponding grid cells are occupied. Parker also organizes cell in *bricks* to improve locality. Build process is top-down, similar to multi-resolution grids.

Other works address the problem of adaptivity with different solutions. For example, in proximity clouds [CS94], a ray traversing empty space is assisted by the distance values which permit to perform long skips along the ray direction.

Cost model suggests a way to build hierarchical grid and in particular to determinate a termination criteria to stop recursion. For example, if $T_{setup} + mT_{step} + \mu T_{inter} > NT_{inter}$, then recursion introduce a benefit in grid traversal time (see Subsection 3.2). However it is worth noting that, without using a good indexing strategy (such as Jevans and Wyvill hash table [JW89]) is not possible use to deep levels of recursion.

5 Hardware architectures considerations

The current trend in CPU and GPU hardware design is towards three concepts: a streaming compute model, vector-like SIMD units, and multi-core architectures. These new architectures combines more and more parallel computations, indeed fast local computations and slow memory access time. This leads to favor method that reduces memory accesses (i.e. compact data structure, compression, multiresolution ...), maximize local memory accesses, and can be easily parallelized, especially in SIMD/MIMD setting. For example memory layouts (i.e. bricking [PPL*99]) should have big and bigger importance in order to improve the performance. This is also more important for the streaming compute model supported by nowadays hardwares.

Moreover, the current trend is to implement rendering methods on GPUs. However, GPU presents some limitations. For example, fragment programs are not allowed to perform data depending branching and have a more restricted memory access (i.e. the number of level of dependent texture fetch is limited). Purcell et al. [PBMH02] showed that is possible to do ray tracing with programmable graphics hardware. Further works have been made by using different data structure such as kd-tree or BVH. Results showed that ray tracing so far does not fully utilize GPU.

The traversal algorithm is unsuited for GPU computation, and branching is unavoidable.Today, for ray tracing, CPUs seem to be more suitable than GPUs.

6 Conclusion

This paper presents a survey on algorithms used in ray tracing by using grid data structure.

Several aspects have been examined: how to efficient traverse a grid, building methodology and their impact in traversal algorithms, how it is possible to exploit adaptivity in grid by building nested hyerarchical grid.

We believe that two main trends will drive research in this data structure. The first is the need of rendering dynamic scenes, because faster grid build time. The latter is the upraise of parallel hardware, because of the easy parallelization.

References

[AW87] AMANATIDES J., WOO A.: A fast voxel traversal algorithm for ray tracing. In *Eurographics '87* (Aug. 1987), pp. 3–10.

[Bre65] BRESENHAM J.: Algorithm for computer control of a digital plotter. *IBM Systems Journal 4*, 1 (1965), 25–30.

[BSW06] BOULOS S., SHIRLEY P., WALD I.: *Geometric and Arithmetic Culling Methods for Entire Ray Packets.* Tech. rep., University of Utah, SCI Institute, 2006.

[CDP95] CAZALS F., DRETTAKIS G., PUECH C.: Filtering, clustering and hierarchy construction: a new solution for ray tracing complex scenes. *Computer Graphics Forum 14*, 3 (1995).

[CS94] COHEN D., SHEFFER Z.: Proximity clouds – an acceleration technique for 3d grid traversal. *Vis. Comput. 11*, 1 (1994), 27–38.

[CW88] CLEARY J. G., WYVILL G.: Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer 4*, 2 (July 1988), 65–83.

[FTI] FUJIMOTO A., TANAKA T., IWATA K.: Arts: Accelerated ray-tracing system. *IEEE Computer Graphics & Applications*.

[Gla84] GLASSNER A. S.: Space subdivision for fast ray tracing. *IEEE Computer Graphics & Applications 4*, 10 (Oct. 1984), 15–22.

[Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[HMS06] HUNT W., MARK W. R., STOLL G.: Fast kd-tree construction with an adaptive error-bounded heuristic. In *2006 IEEE Symposium on Interactive Ray Tracing* (Sept. 2006).

[ISP07] IZE T., SHIRLEY P., PARKER S. G.: Grid Creation Strategies for Efficient Ray Tracing. In *IEEE/EG Symposium on Interactive Ray Tracing* (Sept. 2007), pp. 27–32.

[IWRP06] IZE T., WALD I., ROBERTSON C., PARKER S. G.: An Evaluation of Parallel Grid Construction for Ray Tracing Dynamic Scenes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 27–55.

[JW89] JEVANS D., WYVILL B.: Adaptive voxel subdivision for ray tracing. In *Graphics Interface* (June 1989), pp. 164–172.

[KS97] KLIMASZEWSKI K. S., SEDERBERG T. W.: Faster ray tracing using adaptive grids. *IEEE Comput. Graph. Appl. 17*, 1 (1997), 42–51.

[LZY04] LIU Y. K., ZALIK B., YANG H.: An integer one-pass algorithm for voxel traversal. *Comput. Graph. Forum 23*, 2 (2004), 167–172.

[ORM07] OVERBECK1 R., RAMAMOORTHI1 R., MARK W. R.: A Real-time Beam Tracer with Application to Exact Soft Shadows. In *Eurographics Symposium on Rendering* (2007).

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*, 3 (July 2002), 703–712.

[PGSS06] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Experiences with streaming construction of SAH KD-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (Sept. 2006), pp. 89–94.

[PPL*99] PARKER S., PARKER M., LIVNAT Y., SLOAN P.-P., HANSEN C., SHIRLEY P. S.: Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics 5*, 3 (July/Sept. 1999), 238–250.

[RSH00] REINHARD E., SMITS B., HANSEN C.: Dynamic acceleration structures for interactive ray tracing. In *Eurographics Workshop on Rendering* (2000), pp. 299–306.

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. *ACM Transactions on Graphics 24*, 3 (Aug. 2005), 1176–1185.

[Sra95] SRAMEK M.: A comparison of some ray tracing generators for ray tracing volumetric data. In *The Third International Conference in Central Europe on Computer Graphics and Visualization 1995* (1995), pp. 466–475.

[Wal06] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, Saarbrücken, Germany, 2006.

[WH06] WALD I., HAVRAN V.: On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006).

[Whi80] WHITTED T.: An improved illumination model for shaded display. *Communications of the ACM 6*, 23 (1980), 343–349.

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics 25*, 3 (July 2006), 485–493.

[WMG*07] WALD I., MARK W. R., GÜNTHER J., BOULOS S., IZE T., HUNT W., PARKER S. G., SHIRLEY P.: State of the Art in Ray Tracing Animated Scenes. In *Eurographics 2007 State of the Art Reports* (2007).

[Wyv90] WYVILL B.: Symmetric double step line algorithm. *Graphics gems* (1990), 101–104.

[ZCO97] ZALIK B., CLAPWORTHY G., OBLONSEK C.: An efficient code-based voxel-traversing algorithm. *Comput. Graph. Forum 16*, 2 (1997), 119–128.

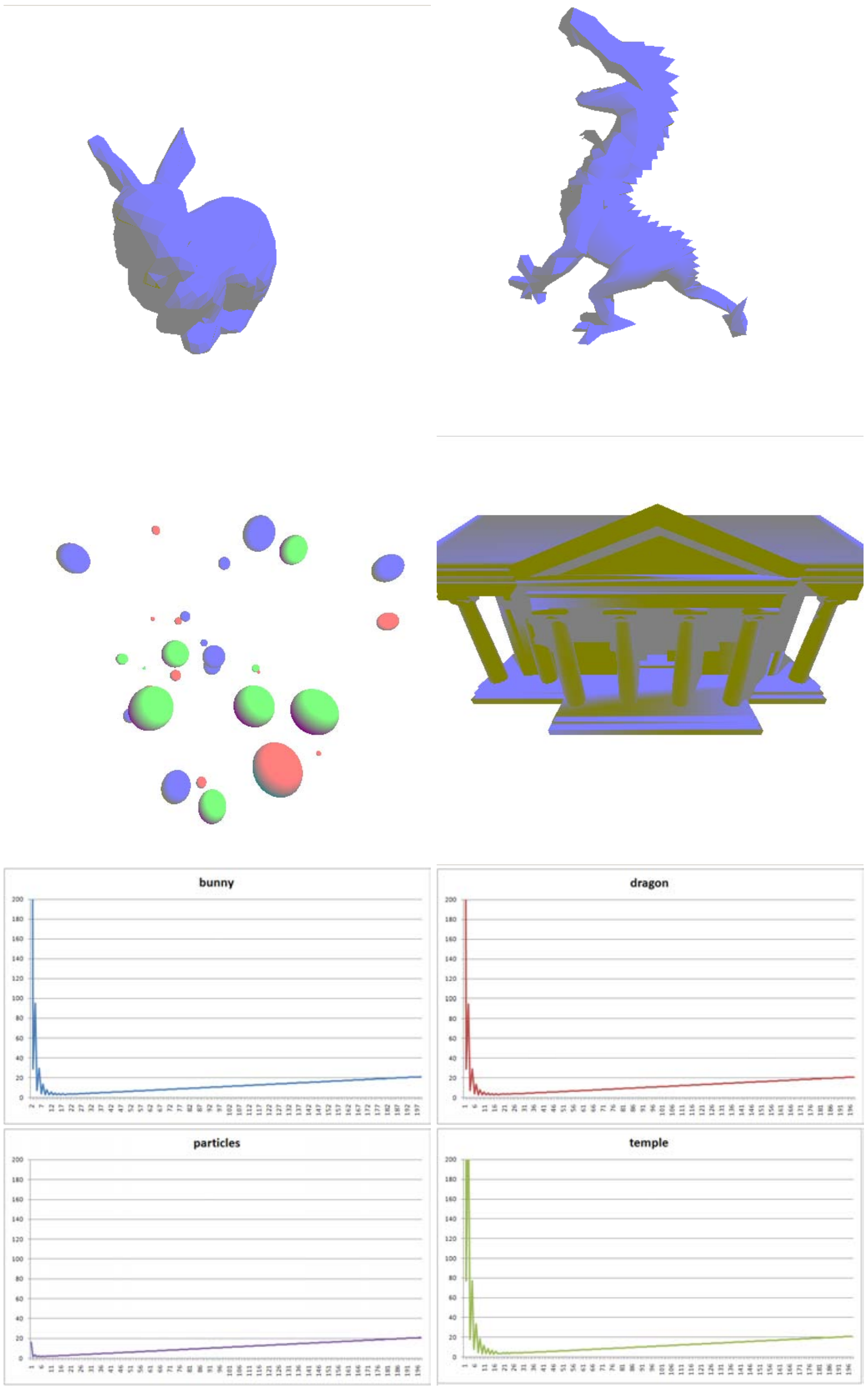

Figure 5: Impact of resolution in four test scenes: two scanned models, an architectural model, a particle-based model. Of the two theoretical model, our test suggest that real world scene are closer to point instead line model. In particular, particle based scenes are less sensitive to the choice of resolution. As expected, architectural model have an higher traversal cost.