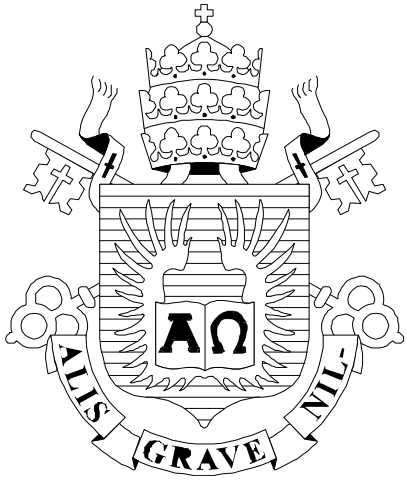


PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 14/09

GPU-Accelerated Uniform Grid Construction for Ray Tracing Dynamic Scenes

Paulo Ivson
Leonardo Duarte
Waldemar Celes

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900
RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 14/09 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena Junho, 2009

GPU-Accelerated Uniform Grid Construction for Ray Tracing Dynamic Scenes

Paulo Ivson, Leonardo Duarte and Waldemar Celes

{psantos,lduarte,celes}@inf.puc-rio.br

Abstract. We present a novel data-parallel algorithm for quickly rebuilding Uniform Grids on state of the art GPUs. The technique combines very fast scan and sorting procedures to classify scene primitives according to the spatial subdivision. Results demonstrate this routine is not only scalable with scene size, but achieves faster rebuild times than other state of the art implementations. In addition, we have developed a ray-tracing procedure that achieves interactive visualization rates, even when enabling shadows and reflection rays. Since the grid structure can be efficiently rebuilt each rendering frame, we can maintain performance with fully animated scenes containing unstructured movements. Overall performance achieved greatly improves upon Uniform Grids on the CPU, while remaining competitive to more adaptive structures such as the BVH and kd-tree.

Keywords: Uniform Grid, Programmable Graphics Hardware, Ray Tracing, Dynamic Scenes

Resumo. Apresentamos um novo algoritmo em paralelo para rapidamente reconstruir Grades Uniformes usando GPUs do estado da arte. A técnica combina procedimentos de "scan" e ordenação em paralelo para classificar primitivas de acordo com a subdivisão espacial. Resultados demonstram que essa rotina não somente é escalável com o tamanho da cena, mas atinge tempos de reconstrução mais rápidos do que outras implementações do estado da arte. Além disso, desenvolvemos um procedimento de traçado de raios que atinge taxas interativas de visualização, mesmo ao habilitar sombras e raios de reflexão. Como a estrutura da grade pode ser eficientemente reconstruída a cada quadro de renderização, podemos manter elevado desempenho para cenas totalmente animadas contendo movimentos não-estruturados. O desempenho final supera outras propostas de grades uniformes na CPU, enquanto permanece competitiva com outras estruturas mais adaptativas como BVH e kd-tree.

Palavras-chave: Grade Uniforme, Programação em GPU, Traçado de Raios, Cenas Dinâmicas

In charge of publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

1 Introduction

State of the art research have shown that ray tracing on the GPU can achieve similar, if not better, performance than the best known algorithms on the CPU. However, most of the established techniques are restricted to static scenes. Only a few recent proposals have been able to support moving and deformable objects. In this case, several acceleration structures and algorithms remain unexplored.

The goal of this research is to develop a ray tracing solution that is capable of harnessing the parallel processing power of the graphics hardware to render dynamic scenes. Similar to other proposals, we have focused on the strategy of rebuilding the acceleration structure from scratch in order to support scenes with any kind of unstructured movement. Our strategy consists in using an acceleration structure that is simpler to be rebuilt, while capable of maintaining fast rendering performance. The Uniform Grid simple construction and ray-traversal algorithms are very good candidates for implementation inside the graphics hardware.

The main contribution of this paper is a novel grid construction algorithm, best suited for parallel shared-memory architectures. Our proposed implementation minimizes inter-thread communication and required memory bandwidth, while at the same time avoiding concurrent writes to the same memory locations. The main idea is to build lists of primitives contained by each cell, storing each primitive in all cells it overlaps. This enables random access to any primitive list during ray traversal. We also present a ray-tracing procedure, also implemented inside the GPU, that fully exploits the grid acceleration structure. In addition, it is capable of efficiently tracing shadow rays as well as reflection ones. Together with our proposed grid rebuild, we can achieve fast rendering performance for static and dynamic scenes of varying sizes.

This document is organized as follows. The next section reviews related research in CPU and GPU ray tracing of dynamic scenes. Section 3 describes our proposed method for fully rebuilding the grid structure inside the GPU. In Section 4, we describe the ray traversal algorithm used to trace rays through the Uniform Grid on the graphics hardware. Results and performance numbers are evaluated in Section 5, where several test scenes identify the benefits and limitations of our approach. Finally, Section 6 concludes this research and introduces several future work that can further improve ray tracing dynamic scenes on the GPU.

Figure 1: Dynamic and CAD models entirely ray-traced on the GPU with our novel grid construction algorithm. From left to right: running character Ben (78K triangles, 13.3fps with shadows), animated wind-up toys (11k triangles, 28.9fps with shadows and 7.8fps with reflections), fairy dancing in a Forest scene (174k triangles, 2.8fps with shadows), static Boat model (50K triangles, 11.7fps with shadows and 4.1fps with reflections) and static MonoBR oil platform (112K triangles, 5.6fps with shadows and 1.4fps with reflections)

2 Related Work

Over the last few years, different spatial structures have been proposed to accelerate ray tracing. In order to support dynamic scenes, one can use a structure that is not so efficient for ray traversal, but that can be quickly modified or rebuilt during the rendering process. Moreover, information about object movement in the scene can aid in rebuilding a structure best suited for each situation.

For instance, if all animation key-frames are known prior to rendering, it is possible to build a Bounding Volume Hierarchy (BVH) that can be adapted (deformed) while objects move in the scene (Wald et al. 2007). The main idea is to keep the hierarchy topology while only deforming the node bounding volumes. However, if extensive or unpredicted movements occur, the hierarchy topology can become invalid and must be entirely reconstructed, slowing down rendering performance.

In order to support the general case of fully dynamic scenes, there have been extensive research in quickly rebuilding the acceleration structure each frame. One technique uses a Uniform Grid (Wald et al. 2006) since its simplicity equates to a fast rebuild scheme. A second proposal implements a kd-tree construction procedure in parallel, tapping into the processing power of modern multi-core CPUs (Shevtsov et al. 2007). A third research, also implemented on a many-core CPU, quickly rebuilds a BVH structure each frame (Wald 2007). These three approaches are able to achieve interactive rendering rates even for unstructured animations.

With the increasing programmability of modern GPUs, recent research have focused on exploiting its parallel processing power to accelerate ray tracing of dynamic scenes. One of the first techniques was based on an efficient kd-tree reconstruction algorithm, fully implemented inside the graphics hardware (Zhou et al. 2008). The entire tree structure was rebuilt each frame, if necessary. The tree traversal procedure used a small per-ray stack, implemented inside the GPU using the CUDA programming model (Nvidia 2008). When ray tracing dynamic scenes, performance obtained surpassed the best CPU implementations. Another related research builds upon the work in (Wald 2007) and presents three different BVH construction algorithms inside the GPU (Lauterbach et al. 2008). Their best results achieve fast rebuild times and matches ray-tracing performance of similar CPU implementations.

In our work, we propose a technique with a similar goal. Our procedure fully rebuilds the entire acceleration structure inside the GPU, while performing ray-traversal and shading computations. However, we have taken a fundamentally different approach. These latest state of the art results use hierarchies, which are highly efficient for ray-traversal but also require more complex construction procedures. Our strategy consists in using an acceleration structure that is simpler to be rebuilt inside the graphics hardware. The work presented here will investigate whether it is more effective to trade-off ray-traversal performance for a faster structure rebuild.

3 Uniform Grid Construction

The Uniform Grid is a regular spatial subdivision structure. The axis aligned bounding box of the entire scene is subdivided into equally sized cells along each of the three main axis X, Y and Z. To obtain best ray-tracing performance, the deciding factor is to choose

a good grid resolution. The typical heuristic to determine the number of cells in each dimension attempts to consider both the complexity of the scene and the size it occupies in space (Wald et al. 2006). In other words, a fairly complex scene would demand a more refined grid. Similarly, a scene that occupies a very large space could be represented by a more sparse grid. This leads to Equation 1, as follows:

$$N_x = d_x \sqrt[3]{\frac{kP}{V}} \qquad N_y = d_y \sqrt[3]{\frac{kP}{V}} \qquad N_z = d_z \sqrt[3]{\frac{kP}{V}} \tag{1}$$

N_x, N_y and N_z are the number of grid cells in each dimension

P is the total number of primitives in the scene

V is the total volume of the grid

d_x, d_y and d_z form the diagonal of the grid

k is a user-defined constant to determine a more dense or sparse grid

A typical grid representation stores a list of primitives contained by each cell. Consecutive lists are arranged contiguously in memory. In practical use, the grid structure must support random access during ray traversal. This means that it is necessary to determine all primitives that overlap a given cell using only its ID. Thus, in addition to the actual grid data we need to build an index to translate the cell ID into the beginning of its corresponding primitive list, as shown in Figure 2.

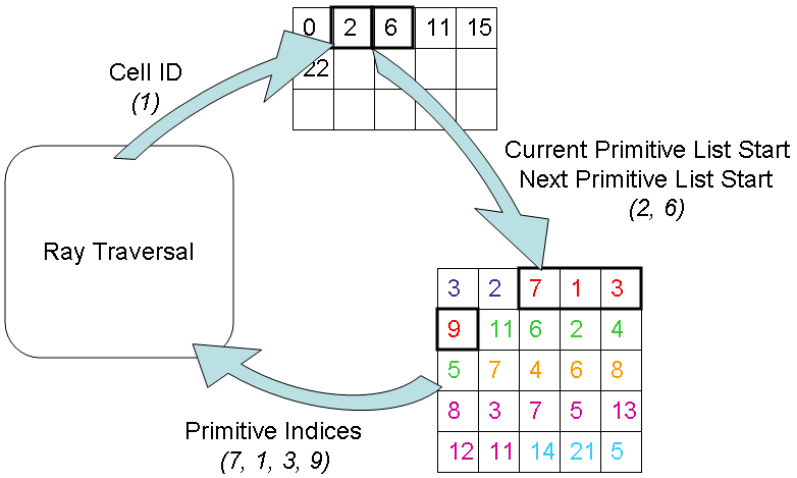


Figure 2: The ray traversal procedure uses the current cell ID to obtain its corresponding list of primitives, to be tested for intersection next. Different colors identify separate lists.

When using the GPU, the main challenge in this construction procedure is to develop an algorithm that is capable of writing into several primitive lists, all arranged contiguously in memory, using hundreds of concurrent threads. Solving this data-parallel problem means to avoid concurrent writes to the same memory position. For instance, if more than one thread is about to write to the same buffer, it is necessary to serialize write operations, slowing down overall performance.

Another question is how to avoid consecutive buffers to overlap one another. One option is to build each buffer in sequence, so that the end of the previous list is known before starting to write the current one. However, buffer sizes tend to be in the order of tens of primitives. This would restrict the amount of parallelism that could be achieved. Another more efficient approach is to compute all buffer sizes beforehand, thereby reserving sufficient memory space for each one.

3.1 Conceptual Algorithm

We therefore propose a multi-pass algorithm to overcome the above challenges. The main idea is to maintain separate buffers, avoiding write serialization, while at the same time assigning computations from different primitives to separate threads. As mentioned in the previous section, the output of the grid construction procedure is the actual grid data, made of a set of primitive lists, accompanied by an index to translate a given cell ID into a buffer start position.

In order to build the actual grid data, suppose each thread computes all cells a single primitive overlaps, and writes these values as contiguous lists of (cell ID, primitive ID) pairs. This avoids concurrent write operations but means that cell-primitive pairs are sorted according to their primitive IDs. The desired layout requires all pairs that have the same cell ID to be arranged contiguously. Our proposed solution is to perform a key-value sorting operation to change the order of these primitive lists, as can be seen in Figure 3. As it turns out, this procedure can be efficiently performed in parallel, inside the GPU (Satish et al. 2008).

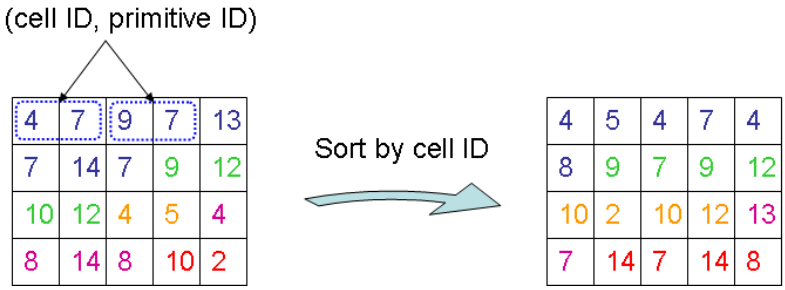


Figure 3: Cell-primitive pairs on the left are sorted by primitive ID. The desired result, on the right, with pairs sorted by cell ID. Colors highlight elements which belong to the same list.

Since in our procedure all threads would perform simultaneous writes to different positions, it is necessary to avoid overlapping primitive lists. We will adopt the strategy of computing buffer sizes beforehand, reserving sufficient memory space for each one. Observe that the start address of buffer N is the sum of all buffer sizes from 0 to N-1. Given this reasoning, we use the following parallel implementation: each thread (primitive) simply counts the number of cells it overlaps. Afterwards, a parallel-prefix-sum procedure (scan) is used to accumulate all values and generate the buffer start indices.

Finally, all that remains is to build the index to translate a given cell ID into a buffer start position. It is possible to use a similar solution as the previous one: compute the number of primitives contained by each cell, in parallel, and then accumulate these values.

If each thread performs the work of a single primitive, concurrent threads are required to serialize increments to the same cell counter. The other option would be to use each thread to perform the work of a single cell, reading all primitive data and incrementing separate counters.

There is another approach that avoids concurrencies and has lower memory bandwidth requirements. Observe that, after the grid sort operation, cell-primitive pairs are arranged in increasing cell ID order. It is therefore possible to use a binary search for a given cell ID, finding the list of primitives it contains (Fernando 2004). All that remains is to perform a linear search for the buffer start index (to the left) and the buffer end index (to the right). To avoid performing this operation during ray traversal, we will pre-compute buffer start indices as well as their sizes beforehand.

Gathering all the previous ideas, we can summarize the multi-pass algorithm described. Figure 4 further details the data generated by each consecutive step.

1. For each primitive, count the number of cells it overlaps.
2. Accumulate the values in Step 1 to compute buffer start indices. These are used in Step 3 to write several buffers in parallel.
3. Write all cells each primitive overlaps using pairs (cell ID, primitive ID).
4. Sort the pairs in Step 3 according to their cell IDs.
5. For each cell ID, perform a binary search in the sorted grid data to find buffer start indices and their sizes.

3.2 Implementation Details

The algorithm for rebuilding the Uniform Grid is implemented entirely on the GPU. Its main input is a list of vertices that makeup the triangles in the scene. These vertices are stored in a texture, which is accessed during grid construction and ray tracing. The total number of triangles in the scene determines the grid resolution, according to Equation 1. After the grid resolution is chosen, we use a multi-pass construction algorithm based on the discussion in the previous section.

We have combined both GLSL and CUDA implementations where each has performed best. Specifically, the parallel-prefix-sum in Step 2 and the key-value sort operation in Step 4 are performed by CUDA procedures, while the other steps can be done efficiently by a single fragment shader each. Figure 5 illustrates an example of our reconstruction procedure. The following subsections present additional details of each step during grid rebuild.

3.2.1 Counting the Number of Cells Each Primitive Overlaps

In the first step, the 2D fragment coordinates of a full-screen quadrilateral are converted to a primitive ID that is used to access the corresponding triangle information. Afterwards, the shader computes the triangle's axis-aligned bounding box (AABB) and counts how many grid cells this box overlaps. This value is then written in the framebuffer. An alternative method would use a more precise, but computationally expensive, triangle-box

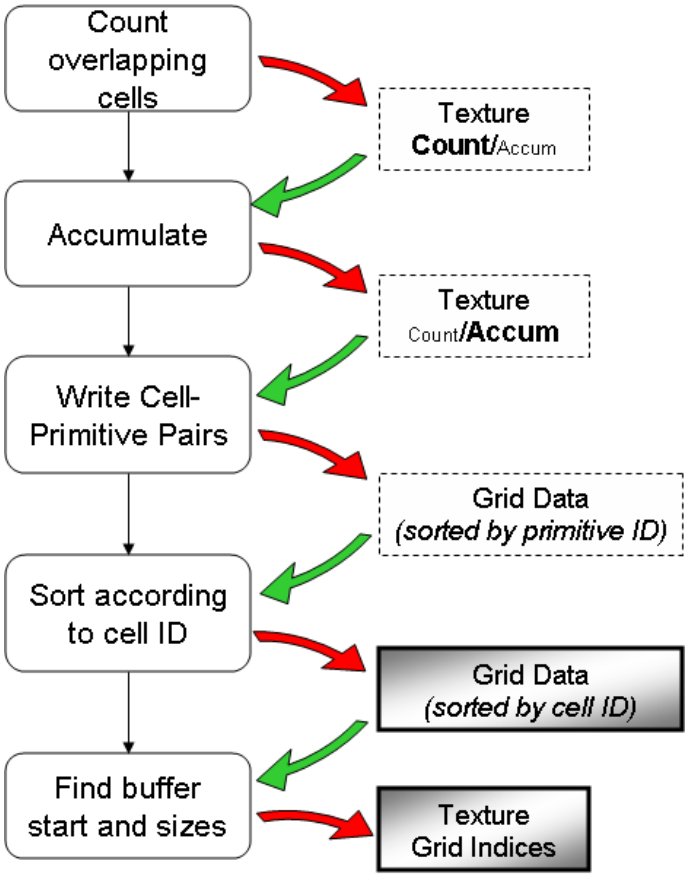


Figure 4: Data flow of the proposed grid rebuild algorithm. Red and green arrows indicate write and read operations, respectively. Grey boxes highlight the main output of the procedure.

overlap algorithm (Möller 2001). This option, however, has performed poorly during our tests. Also, the use of AABB is by far more adequate for the third (and most challenging) step of the algorithm.

3.2.2 Computing Primitive Buffer Indices

The output values of the previous step are then processed by a scan procedure implemented in the CUDA Data Parallel Primitives Library or CUDPP (Sengupta et al. 2007). The resulting accumulated values are read back to an OpenGL texture. This texture now contains, given a primitive ID, the index where its buffer starts in the final grid texture. We store an additional accumulated value at the end, which represents where the last primitive list terminates. This number also represents the required size of the final grid texture.

3	2	4	1

(a) Output from Step 1: the number of cells each of the 4 primitives overlaps.

0	3	5	9	10

(b) Step 2 accumulates the values from Step 1, indicating where each primitive list begins. Notice an additional accumulated value at the end, indicating the required grid size.

3	0	4	0	6	0	0	1	3	1	3	2	4	2	5	2	7	2	1	3

(c) The (cell ID, primitive ID) pairs from Step 3, initially sorted by primitive ID.

0	1	1	3	3	0	3	1	3	2	4	0	4	2	5	2	6	0	7	2

(d) Step 4 output: pairs are sorted by cell ID.

0	1	1	1	0	0	2	3	5	2	7	1	8	1	9	1	0	0

(e) Results from step 5: the cell start index and the number of primitives it contains.

Figure 5: Example of our grid construction algorithm using 4 primitives and 9 cells. Colors indicate elements that belong to the same list.

3.2.3 Writing Cell-Primitive Pairs

The third step is to write the (cell ID, primitive ID) pairs using the indices computed in Step 2. At first sight, this would require each thread (primitive) to perform several write operations in a single pass. It is possible to implement this algorithm either using CUDA or GLSL. In order to achieve best performance using the CUDA programming model, it is necessary to arrange sequential threads to perform sequential write operations, thus enabling the hardware to amortize memory latency by coalescing several small write requests into one large memory operation. This optimization cannot be achieved when writing several pairs in different buffers.

An alternative is to use GLSL shaders instead. To write several values per primitive, there are two options: using a vertex shader or a geometry shader. Our experiments have shown that both options achieve poor performance. Outputting more than a dozen primitives per geometry shader results in severe performance degradation, while current GPUs limit vertex output to about 340 vertices. Additionally, using a vertex shader would require multiple draw calls totalling the largest primitive buffer size. That is, the maximum number of cells a single triangle overlaps. This means that if the scene contains even one large triangle, it would severely slow down the entire grid construction.

All previous approaches have the same paradigm of trying to use each thread to perform several write operations. Let us think backwards then. Given a cell-primitive pair slot in the resulting grid, we need to obtain which cell ID and which primitive ID need to be written. As it turns out, we can use the accumulated values computed in Step 2 to determine both these IDs. Recall that these values represent the start index of each primitive list, while an additional accumulated value at the end represents the size of the

resulting grid. It is safe to say, then, that all memory positions in the resulting grid are contained by a lower and an upper bound in this accumulated texture.

Note that the lower bound value represents the start of the primitive list where any given cell-primitive pair belongs. In other words, all memory positions in the resulting grid with the same lower bound in the accumulated texture belong to the same primitive list. Therefore, they have the same primitive ID. This ID is simply the memory index of the lower bound value in the accumulated texture. Since the accumulated values are, by definition, sorted in increasing order, we can perform a modified binary search, inside a simple fragment shader, to find the required lower bound for any memory position in the resulting grid.

The cell ID value, on the other hand, can be computed by using a local offset: the difference between the memory position in the resulting grid and the start position of the primitive list (the lower bound value). This local offset indicates the *nth* overlapped cell ID must be written in the *nth* position in the primitive list. We establish a convention of ordering grid cells in ascending linear IDs in X, Y and then Z order. This allows for converting linear cell IDs to 3D cell IDs.

3.2.4 Sorting the Grid Data

After that, a sort operation is performed to arrange the pairs from Step 3 according to their cell IDs. We use a very fast key-value *radix-sort*, recently available in the latest CUDA SDK (Satish et al. 2008). The resulting texture now contains the final constructed Uniform Grid. All that remains is to determine how to access this data structure.

3.2.5 Computing Indices to Access Grid Data

In Step 5, each thread performs a binary search using the cell ID to find its corresponding primitive list. Afterwards, a linear search is used to compute the buffer start index (to the left) and the buffer end index (to the right). All these operations can be performed by a single fragment shader. The shader writes both the buffer start index and its size in the same pixel, allowing a single texture fetch operation, to recover both these values during ray traversal.

4 Ray Tracing Implementation

The regular subdivision of a Uniform Grid greatly simplifies the task of determining which cell must be visited along a given ray. We use a classic grid traversal algorithm based on a 3D digital differential analyzer, or 3D-DDA, to step the ray along successive cells (Amanatides and Woo 1987). The main advantage of this technique is that it only requires a few pre-computed values per ray, and its main loop can be done efficiently on the graphics hardware. In contrast to other acceleration structures, such as a kd-tree, there is no need to maintain a per-ray traversal stack. These characteristics make the 3D-DDA grid traversal a favorable candidate for implementation inside the GPU.

We have implemented our entire ray-tracing procedure using GLSL shaders. There are fundamentally two approaches to implementing the ray tracing kernel. The first is to use a single fragment shader to encode the entire ray setup, traversal, intersection and shading routines. The other is to break the procedure into several rendering passes. We

have verified experimentally that separating the traversal and intersection from the shading routine can perform up to two times faster than a monolithic shader approach.

Therefore, our proposed ray tracing implementation on the GPU can be summarized in three main steps, each performed by a different fragment shader:

1. Primary ray traversal and intersection
2. Shadow ray traversal and intersection
3. Shading computations

After initializing the primary rays, the shader in Step 1 first checks for intersection against the grid bounding box. If none is found, the shader outputs invalid values. In the other case, the shader uses the 3D-DDA traversal to find the nearest triangle intersection, which is then stored in an off-screen buffer. This output contains, for each texel, the following hit information: triangle ID, barycentric coordinates (u, v) and the hit distance. If no hit was found, the shader writes invalid values.

The shader in Step 2 uses this hit information to compute the origin and direction of the shadow rays, tracing them using an optimized procedure that only computes minimal information to determine if a point is in shadow. Shadow rays are cast from the primary hit position towards a global point light. The output is the same hit information from Step 1, with shadowed hits identified by negative triangle IDs.

Finally, Step 3 reads the hit information from Step 2. The triangle ID is used to access the necessary triangle information (normals, texture coordinates and materials) to perform all shading computations. The barycentric coordinates (u, v) are used to interpolate per-vertex attributes such as normals and texture coordinates. Finally, the hit distance is used to evaluate the hit position in space and compute the final Phong illumination model.

4.0.6 Enabling Reflections

We can re-use the aforementioned algorithm to shade reflective materials. Our strategy consists in another rendering pass, performing the same Steps 1, 2 and 3 but now for reflection rays. The final color values are modulated with the original ones, using the OpenGL blend operation.

In this case, Step 1 does not initialize the rays using the view information, but reads the primary hits previously computed in the same frame. The secondary hits are then processed by Steps 2 and 3 as usual, generating the final color for the reflection rays. Clearly, this iterative process could be repeated, each time shading another level of reflections. For testing purposes, we have limited our implementation to a single level.

5 Results and Discussion

In this section we present a performance evaluation of our proposed grid-construction and ray-tracing solutions. Section 5.1 presents a synthetic analysis of GPU grid reconstruction performance, comparing to an equivalent CPU implementation. Following that, Section 5.2 evaluates our ray-tracing algorithm using static scenes, with no grid rebuild. In this case, we are also interested in measuring ray-tracing scalability and flexibility.

Afterwards, in Section 5.3, we perform a full evaluation of the integrated ray-tracing system, using dynamic scenes to identify the possible benefits and limitations of our method. Finally, Section 5.4 presents a detailed comparison of our results with other state of the art research. In all our tests, we have used a Core 2 Duo 3.0GHz CPU with an Nvidia 8800 Ultra graphics card. All scenes were rendered at 1024 x 1024 screen resolution.

5.1 Grid Construction

The Uniform Grid reconstruction procedure has been evaluated against a similar CPU implementation. The test scene consists of several triangles randomly distributed inside a box with dimensions [-50, -50, -50] x [50, 50, 50]. Each triangle has a randomly determined size, obtained by varying the radius of its enclosing bounding sphere from 0.2 to 1.0. Grid resolution is determined using Equation 1. Therefore, increasing the number of triangles not only increases the amount of data to be stored, but also the number of cells in the resulting grid.

#Tri	$5K$	$10K$	$50K$	$100K$	$200K$	$300K$	$400K$	$500K$
CPU	0.8	1.5	10.2	33.8	94.1	167.4	253.0	353.6
Ours	1.7	2.2	6.9	14.5	32.9	55.2	63.8	87.4

Table 1: Time in milliseconds to rebuild the entire grid structure.

As can be seen in Table 1, the GPU solution suffers from the overhead of the graphics API for small scenes (10K triangles or less). However, this procedure scales better than its CPU counterpart, obtaining faster grid rebuild times for scenes with more than 10k triangles. We are capable of achieving near-linear scaling, even though increasing the number of triangles in the scene also increases the total number of cells in the grid.

5.2 Static Scenes

In order to measure ray-tracing performance independently of grid rebuild, we have devised a number of static scene tests. In these, the Uniform Grid is built only once during initialization. The first test measures performance of primary rays only, by using a simple grey-scale shader with no additional texture accesses. In the second test, we use additional material information obtained from several textures to perform lighting computations, and includes tracing shadow rays from a single point light. Finally, the third test case further enables reflection rays for the entire scene.

5.2.1 CAD Models

The first CAD model, called “Boat", is made of 50K triangles with no textures. The second model is an oil platform called “MonoBR", made of 112K triangles, including a few textured materials. The third model is a complex section of the “P-40" oil platform, with more than 470K triangles (see Figures 1 and 6). Table 2 summarizes ray-tracing performance for all test cases.

An evident result is the scalability of the ray-tracing procedure according to scene size. Even the “P-40" model with about half a million triangles can be rendered at almost the

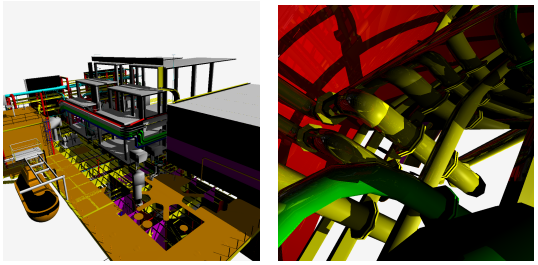


Figure 6: On the left, P-40 CAD model with over 470K triangles, ray-traced with shadows. On the right, a close-up of an equipment inside the platform, including shadows and reflections.

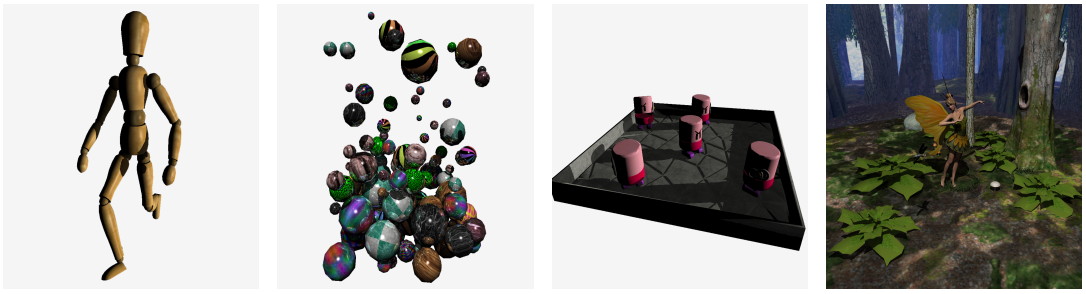
Scene	# Tris	Simple	+One Light	+Reflections
Boat	50K	21.3	11.7	4.1
MonoBR	112K	11.6	5.6	1.4
P40	470K	14.1	7.8	1.5

Table 2: Performance in frames per second (fps) for different static scenes.

same performance as the smaller “Boat". In fact, using the point of view in Figure 6 (a), it even outperforms the “MonoBR" scene with the point of view in Figure 1.

5.2.2 Benchmarks

The second batch of tests consist in a known benchmark for ray tracing dynamic scenes. All models contain detailed material information, including several high-resolution textures, as can be seen in Figures 1 and 7. In this case we seek to evaluate the pure ray-tracing performance, disregarding the acceleration structure rebuild. We use one animation key-frame from each model, building the grid structure only once. The results in Table 3 will help identify the bottleneck during the final rendering of the entire animation.



(a) Wood-doll model with shadows.

(b) Several marbles rendered with shadows and reflections.

(c) Toys scene with shadows.

(d) Fairy dancing in the Forest scene.

Figure 7: Deformable meshes and scenes with unstructured movement used for static and dynamic tests.

The first model is a simple “Wood-doll", made of 5K triangles. The second one is a

“Hand", modeled with 16K triangles. A runner character of 78K triangles is the model called “Ben". Another model consists of several “Marbles" that add up to almost 9K triangles. The fourth scene is made of five wind-up “Toys", totalling 11K triangles. Finally, the largest scene is a fairy model inside a “Forest", consisting in about 174K triangles.

Scene	# Tris	Simple	+One Light	+Reflections
Wood-doll	5K	71.5	54.6	14.6
Hand	16K	42.4	26.5	6.8
Ben	78K	19.6	16.5	4.4
Marbles	9K	108.9	88.1	16.5
Toys	11K	53.7	33.4	8.2
Forest	174K	6.3	3.4	0.7

Table 3: Performance in frames per second (fps) for a single key-frame of the benchmark scenes.

As shown in Table 3, the two simplest scenes, “Wood-doll" and “Marbles", can be rendered with shading and shadows at speeds above 50 fps. With the other less simple models, “Hand" and “Toys", our implementation can still achieve rendering rates of around 30 frames per second. The more complex “Ben" model can be rendered at about the same speed as the “Boat" model, similar in size, evaluated in the previous section.

The “Forest" scene is a classic worst-case scenario for the Uniform Grid: a complex object (the fairy) at the center of a larger but simple scene (the background). This is commonly known as the *teapot in a stadium* problem. Nevertheless, our implementation is capable of achieving interactive rendering rates, except when enabling reflections in the more complex models.

5.2.3 Discussion

The performance results from the CAD models and the benchmark scenes are consistent with the following observations:

1. Activating shading computations as well as shadow rays decreases rendering performance by at most a factor of two.

2. Tracing reflection rays further reduces these values by a factor of four.

The first observation is not only due to the cost of tracing additional shadow rays, which effectively doubles the number of rays being traced per frame, but also due to additional memory operations required for shading computations.

Considering these two factors, performance is above what one would expect. We have further verified that using optimized traversal and intersection routines for shadow rays have significantly reduced their overall impact in rendering speed.

Furthermore, performance with one level of reflection is interactive for scenes with less than 100K triangles. A simple explanation is that each additional reflection ray performs the entire shading computations once more, while also spawning additional shadow rays. In effect, this test has twice the number of rays being traced per frame and also twice the number of shading procedures being performed.

5.3 Dynamic Scenes

The main goal of this work is to ray-trace dynamic scenes, including illumination effects, at interactive rendering rates. We have evaluated the same six benchmarks, from Section 5.2.2, but this time we render their entire animation sequences.

The “Wood-doll", “Hand" and “Ben" models consist of deformable meshes that do not move along the scene. Meanwhile, the “Marbles", “Toys" and “Forest" scenes combine mesh deformation with unstructured movements, comprising real-world test scenarios.

Scene	# Tris	Simple	+One Light	+Reflections
Wood-doll	5K	68.6	52.6	13.1
Hand	16K	41.2	25.2	6.4
Ben	78K	17.5	13.3	3.1
Marbles	9K	103.2	84.3	15.3
Toys	11K	52.7	28.9	7.8
Forest	174K	3.1	2.8	0.4

Table 4: Performance in frames per second (fps) for the entire animation of each benchmark scene.

When rendering the entire animation, it is necessary to fully rebuild the grid structure each new key-frame. Comparing the values in Table 4 with Table 3, it is clear that this reconstruction procedure has little to no impact in rendering performance. The frame-rate with full animations is up to 10% smaller than when rendering a single key-frame. Only the “Forest" scene has suffered a greater slowdown, from 30% to 50% depending on the test case.

5.4 Comparison with Related Work

In this section, we seek to evaluate our work in relation to state of the art research. We have chosen the test case with fully animated scenes including shading, textures and shadows. Table 5 includes performance figures from our technique, as well as other four related work.

Scene	Our Method	CPU Grid	CPU BVH	CPU kd-tree	GPU kd-tree
Wood-doll	52.6	35.1	60.0	*n/a*	*n/a*
Hand	25.2	15.9	48.0	*n/a*	*n/a*
Ben	13.3	8.9	*n/a*	*n/a*	*n/a*
Marbles	84.3	19.6	56.0	*n/a*	*n/a*
Toys	28.9	9.4	36.0	23.5	32.0
Forest	2.8	1.3	11.5	5.8	6.4

Table 5: Frames-per-second comparison between our method and four state of the art research, using complex shading and shadow rays.

The first work is a CPU implementation that uses a Uniform Grid to trace packets of rays (Wald et al. 2006). The second quickly rebuilds a bounding volume hierarchy

using an 8-core CPU (Wald 2007). Another proposal uses a 4-core CPU to build a kd-tree structure in parallel (Shevtsov et al. 2007). Finally, the fourth related research is an equivalent solution to ours, but one that uses a kd-tree as acceleration structure inside the GPU (Zhou et al. 2008).

Values in Table 5 demonstrate the efficiency of our grid rebuild and ray-tracing algorithms. Related to the work in (Wald et al. 2006), we are capable of achieving up to four times faster rendering speeds. When compared against more optimized structures, such as the BVH used in (Wald 2007) and the kd-tree used in (Shevtsov et al. 2007; Zhou et al. 2008), our method still remains competitive, even achieving faster rendering speeds for the "Marbles" scene.

In the more sparse "Toys" model, we are able to surpass the work in (Shevtsov et al. 2007), but still achieve inferior performance than results in (Zhou et al. 2008; Wald 2007). In the "Forest" scene, this situation is aggravated: our implementation achieves at most half the performance values of these three related research. This once more indicates a limitation in the Uniform Grid traversal (as discussed in Section 5.2.2).

To try and identify this possible bottleneck in our implementation, we have split the total frame time of our solution into: key-frame upload time, grid rebuild time and ray-tracing time. Table 6 summarizes our findings with the complete shading algorithm and shadow rays. For comparison purposes, we have included the structure rebuild times from both (Wald et al. 2006) and (Zhou et al. 2008).

Scene	Upload	Rebuild	Ray-Trace	CPU Grid	GPU kd-tree
Wood-doll	1.1	2.3	15.5	1.0	*n/a*
Hand	2.7	4.1	36.2	5.0	*n/a*
Ben	12.4	10.4	81.1	14.0	*n/a*
Marbles	1.6	1.8	8.6	2.0	*n/a*
Toys	1.8	3.0	34.9	4.0	12.0
Forest	27.1	38.4	317.4	68.0	77.0

Table 6: Analysis of times in milliseconds from our proposed implementation, compared to rebuild times from related work.

From these results, we can conclude that uploading new key-frame data to the GPU is not the current bottleneck. Even though in the "Forest" scene a time of 27 ms starts to detriment overall performance, it is still not the most time consuming step: the ray-tracing procedure is the major factor to slowing down rendering rates. Comparing our grid rebuild times with the ones from (Wald et al. 2006), we find that our method is considerably faster except for the very simple "Wood-doll" scene. This can be easily explained as the overhead of the GPU implementation, which has already become evident in Section 3.

As expected, the kd-tree rebuild times from (Zhou et al. 2008) are higher than our Uniform Grid implementation. Not included in Table 6, the parallel BVH construction time from (Wald 2007) for the "Forest" scene is 83 ms, more than double our rebuild time. Since our overall performance is worse in the "Toys" and "Forest" scenes, we can conclude that our ray-tracing procedure is taking most of the rendering times. Indeed, Table 6 show that for all the other test scenes our current bottleneck is the ray-tracing step, which can take up to ten times longer than the grid structure rebuild.

6 Conclusion and Future Work

In order to explore the processing power of current GPUs, we have successfully developed a data-parallel Uniform Grid construction algorithm, capable of obtaining fast and scalable rebuild times. Additionally, we have presented optimized ray-traversal, intersection and shading routines inside the graphics hardware. Together, we have demonstrated a complete ray-tracing solution capable of interactively rendering fully dynamic scenes including illumination effects such as shadows and reflections.

Our proposed GPU implementation of the Uniform Grid rebuild has performed significantly faster than a similar state of the art result on the CPU (Wald et al. 2006), while clearly surpassing more complex structure rebuild times from other research (Wald 2007; Shevtsov et al. 2007; Zhou et al. 2008). Performance figures demonstrate that our current bottleneck is the ray-tracing step.

Nevertheless, our proposed ray-tracing solution greatly improves upon related work using Uniform Grids on the CPU (Wald et al. 2006). On the other hand, our technique has presented similar or inferior performance than both BVH (Wald 2007) and kd-tree (Shevtsov et al. 2007; Zhou et al. 2008) state of the art implementations. However, our ray-tracing algorithm only performed the worst in more sparse scenes ("Toys" and "Forest"). These usually represent a worst-case scenario for the Uniform Grid, naturally requiring a more adaptive structure to obtain optimal ray-traversal performance.

A major improvement to our work would be to use a more adaptive acceleration structure. For instance, our current grid construction algorithm can be modified to build multi-level Uniform Grids. There are several ways to organize grids hierarchically, including loosely nested grids (Cazals et al. 1995; Klimaszewski and Sederberg 1997), recursive or multiresolution grids (Jevans and Wyvill 1989), and macrocells or multigrids (Parker et al. 2005). With knowledge of the behavior of each scene object, it is also possible to assign independent Uniform Grids to each moving and deformable object. This would guarantee tightly packed structures, with a minimal number of empty cells. In this case, rigid body movement could be simply treated by transforming the ray into local object space, as in (Wald 2004). Another structure modification that could improve traversal of empty space would be to use a flag to skip empty cells along the ray (Baboud and Décoret 2006).

In addition, it should be noted that our ray-traversal implementation could be further optimized to trace ray bundles, as in (Wald et al. 2006). It is possible to trace packets of rays through a hierarchical Uniform Grid using several optimizations not investigated in our work. A modified GPU ray-tracing implementation with these optimizations could achieve an order of magnitude in performance gains.

Finally, our Uniform Grid construction procedure is not tied to triangle representations. Our proposed algorithm is generic and flexible, becoming an excellent tool for accelerating other applications, such as photon mapping or particle system simulations.

Acknowledgements

The wood-doll, ben, fairy, toys, marbles, and hand models are from the Utah Animation Repository. The boat, monoBR and p40 models are used with permission from Petróleo Brasileiro S/A - PETROBRAS, by the means of Tecgraf, PUC-Rio.

References

Amanatides, J., and Woo, A. 1987. A fast voxel traversal algorithm for ray tracing. In *In Eurographics '87*, 3–10.

Baboud, L., and Décoret, X. 2006. Rendering geometry with relief textures. In *Graphics Interface '06*.

Cazals, F., Drettakis, G., and Puech, C. 1995. Filtering, clustering and hierarchy construction: a new solution for ray tracing very complex environments. In *Eurographics'95*. Maastricht.

Fernando, R. 2004. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley Professional, March.

Jevans, D., and Wyvill, B. 1989. Adaptive voxel subdivision for ray tracing. In *Graphics Interface '89*, 164–172.

Klimaszewski, K. S., and Sederberg, T. W. 1997. Faster ray tracing using adaptive grids. *IEEE Computer Graphics and Applications 17*, 1, 42–51.

Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., and Manocha, D. 2008. Fast bvh construction on gpus. *Computer Graphics Forum 28*, 2, 375–384.

Möller, T. A. 2001. Fast 3d triangle-box overlap testing. *journal of graphics tools 6*, 1, 29–33.

Nvidia. 2008. *CUDA Programming Guide 2.0*.

Parker, S., Parker, M., Livnat, Y., Sloan, P.-P., Hansen, C., and Shirley, P. 2005. Interactive ray tracing for volume visualization. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, ACM, New York, NY, USA, 15.

Satish, N., Harris, M., and Garland, M. 2008. Designing efficient sorting algorithms for manycore gpus. NVIDIA Technical Report NVR-2008-001, NVIDIA Corporation, Sept.

Sengupta, S., Harris, M., Zhang, Y., and Owens, J. D. 2007. Scan primitives for gpu computing. In *Graphics Hardware 2007*, ACM, 97–106.

Shevtsov, M., Soupikov, A., and Kapustin, A. 2007. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Comput. Graph. Forum 26*, 3, 395–404.

Wald, I., Ize, T., Kensler, A., Knoll, A., and Parker, S. G. 2006. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics*, 485–493. (Proceedings of ACM SIGGRAPH 2006).

Wald, I., Boulos, S., and Shirley, P. 2007. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26*, 1.

Wald, I. 2004. Realtime ray tracing and interactive global illumination. *PhD thesis, Saarland University*.

WALD, I. 2007. On fast Construction of SAH based Bounding Volume Hierarchies. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing*.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time kd-tree construction on graphics hardware. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, ACM, New York, NY, USA, 1–11.