Parallel and Lazy Construction of Grids for Ray Tracing on Graphics Hardware

Javor Kalojanov

Student at
Saarland University
66123 Saarbrücken, Germany

A thesis submitted in partial satisfaction of the requirements for the degree Master of Science at the Department of Computer Science at Saarland University.

Supervisor:
Prof. Dr.-Ing. Philipp Slusallek
Saarland University, Saarbrücken, Germany

Reviewers:
Prof. Dr.-Ing. Philipp Slusallek
Saarland University, Saarbrücken, Germany

Dr. Michael Wand
Max Planck Insitut Informatik, Saarbrücken, Germany

Thesis submitted on:
30. September 2009

Javor Kalojanov
Bleichstr. 28
66111 Saarbrücken, Germany
javor@graphics.uni-sb.de

September 30, 2009

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, 30. September 2009

(Javor Kalojanov)

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbrücken, 30. September 2009

(Javor Kalojanov)

Abstract

In this thesis we investigate the use of uniform grids as acceleration structures for ray tracing on data-parallel machines such as modern graphics processors. The main focus of this work is the trade-off between construction time and rendering performance provided by the acceleration structures, which is important for rendering dynamic scenes. We propose several parallel construction algorithms for uniform and two-level grids as well as a ray triangle intersection algorithm, which improves SIMD utilization for incoherent rays. The result of this work is a GPU ray tracer with performance for dynamic scenes that is comparable and in some cases better than the best known implementations today.

Acknowledgements

I would like to thank Prof. Philipp Slusallek for his help and support.

Secondly I want to thank Stefan Popov both for helping with ideas and for doubting various aspects of the presented approaches, which pushed the work on the thesis further. I also thank Michael Wand for agreeing to review the thesis. Iliyan Georgiev, Lukas Marsalek, Tomas Davidovic, Mike Phillips and other members of the Computer Graphics chair helped with insightful discussions and advices.

Finally I thank my mother, my father and my sister for their support and patience.

Contents

Chapter 1

Background

In this chapter we introduce general notions such as Ray Tracing, Global Illumination, and Acceleration Structures. If the reader feels confident in his knowledge in the respective area, he may safely skip the corresponding sections.

1.1 Rendering and Global Illumination

In this thesis we discuss algorithms for generating a two-dimensional image from a scene consisting of three-dimensional geometric primitives. Those are referred to as *rendering algorithms*. Very informally, the process of rendering can be described as taking a photograph of some 3D objects. We refer to these objects collectively as *scene*. The geometry (location and form) of each of the scene objects is described by a set of geometric primitives, e.g. points, triangles, spheres, polynomial patches. In this thesis we only consider triangles, but the described algorithms can be extended to work with other primitive types.

Rendering and Image Synthesis are very general notions. The purpose of the algorithms described here concerns a family of rendering algorithms called - *global illumination algorithms*.

We define global illumination simulation as the process of computing how light interacts with the scene. The light is emitted from *light sources* and then transformed by the objects in the scene. To calculate these transformations, we attach additional information to the geometric primitives in our scene. These properties define the appearance or color of the scene-objects. We consider two types of information for appearance - the *shaders* are programs that describe how the surfaces interact with light (or energy) and the *materials* are data containers for properties relevant to the shaders. Example interactions of objects and their surfaces with light include absorption, reflection, refraction, scattering etc.

Computing a global illumination solution amounts to determining the

amount of light or energy travelling from some point p in some outgoing direction $\vec{\omega}_o$, which is the same as solving the *rendering equation* [Kaj86]:

$$L(p, \vec{\omega}_o) = L_e(p, \vec{\omega}_o) + \int_{\Omega^+} f_r(\vec{\omega}_i, p, \vec{\omega}_o) L_i(p, \vec{\omega}_i) \cos\theta_i \, d\vec{\omega}_i. \tag{1.1}$$

It states that the outgoing radiance $L(p, \vec{\omega}_o)$ is the sum of the radiance emitted from the point p and the radiance reflected from p, both along the direction vector $\vec{\omega}_o$. The reflected radiance is computed by integrating the the incident energy $L_i(p, \vec{\omega}_i)$ weighted by the BRDF function $f_r(\vec{\omega}_i, p, \vec{\omega}_o)$ over all possible incoming directions $\vec{\omega}_i$. The cosine of the angle θ_i between the incident direction $\vec{\omega}_i$ and the surface normal at p is a weighting factor that accounts for the solid angle at $\vec{\omega}_i$.

Typically an algorithm that approximates the Equation 1.1 works as follows. First we want to render an image consisting of finite amount of pixels. To do this we have to compute the color of each pixel. We determine sample directions $\vec{\omega}_{prim}$ that go from the detector (eye-point, camera lens etc.) through the pixels in the scene. We find all points in the scene that are intersected by these *primary rays*. Then for each of these points p we solve the rendering equation for p and $\vec{\omega}_o$, which is the inverse of $\vec{\omega}_{prim}$.

The Rendering Equation (Equation 1.1) is Fredholm integral equation of the second kind and is not solvable analytically in the general case. Instead various Monte Carlo algorithms have been developed that compute a solution by sampling (see [Kaj86, Laf96, Vea97, Jen96]). In-depth discussion of Monte Carlo Global Illumination algorithms is beyond the scope of this thesis. Still, this class of algorithms is relevant to the results described here since all of the mentioned approaches rely heavily on ray tracing for sampling. Improving the efficiency of ray tracing improves the performance of the whole algorithm, which will hopefully become fast enough to allow real-time global illumination simulations in complex scenes in the future.

1.2 Ray Tracing

We define the operation of tracing a ray as follows. Given a ray with an origin o and direction $\vec{d}$, and a scene consisting of geometric primitives (e.g. triangles), we determine if there is a primitive in the scene that the ray intersects, and optionally which is the primitive closest to the origin. With other words we want to know if there is scalar value t, such that $o + t\vec{d}$ is a point in space that belongs to a given primitive P in the scene.

$$\left\{ t \mid \exists P \in \textit{scene}\, s.t.\, (o + t\vec{d}) \in P \right\} \neq \emptyset \tag{1.2}$$

We may optionally look for

$$\min \left\{ t \mid \exists P \in \textit{scene}\, s.t.\, (o + t\vec{d}) \in P \right\}. \tag{1.3}$$

To be able to answer such queries, one only needs to be able to perform an *intersection test* between a ray and a single geometric primitive. Given a ray one can test every input primitive for intersection and determine if there is one. Even though this can be done in linear time it is very inefficient even for very small scenes. This is why high performance ray tracing implementations rely on *acceleration structures*.

1.3 Acceleration Structures

The way ray tracing is related to performing a search, acceleration structures for ray tracing are related to general search structures like dictionaries. For ray tracing one uses spatial structures that subdivide the space and store the geometric primitives in their cells. A Binary Space Partition tree (BSP tree) for example is the analog of a binary search tree. It partitions space hierarchically via planes and stores the primitives in its leaves. Instead of testing each ray against each primitive for intersection, one first traverses the tree to find all leaves intersected by the ray and tests only the primitives stored in those leaves. In this way some cheap traversal calculations help eliminate large amount of intersection candidates - those that are not contained in leaves intersected by the ray.

The use of various acceleration structure for ray tracing has been thoroughly studied for a wide variety of applications. Early research focuses on the quality of the acceleration structure, regardless of its build time [Hav01]. Such structures are to be used in static scenes and the construction is made in a preprocessing step, making the build time irrelevant. Later on, various algorithms for fast construction and different acceleration structures have been proposed. Some of them aim to reduce the time for construction in order to allow ray tracing of dynamic scenes [IWRP06, PGSS06, WBS06, WH06, WK06, WMS06]. All of the above algorithms have been developed on and optimized for (multi-core-) CPUs. An exception are the B-kd trees of Woop [WMS06], which are also implemented in a hardware prototype specialized for ray tracing.

Recently GPUs, which have been previously specialized fixed function hardware for raster-based rendering, started to develop in the direction of a general purpose parallel processors. This, together with the introduction of the CUDA programing model [NBGS08], made them an attractive platform to implement ray tracing on [PGSS07, GPSS07]. Although GPUs are very powerful processors, their hardware architecture imposes some limitations. In order to exploit their potential the programs have to be *massively parallel*, i.e. they must be able to distribute work between hundreds of thousands of threads. For this reason a direct mapping of a CPU algorithm to the GPU rarely provides any significant speedup. While the process of tracing the individual rays is considered naturally parallel and maps well to GPUs

[AL09], this was not the case with the algorithms for construction of the spatial structures. This led to the development of parallel algorithms for construction of acceleration structure such as kd-trees [ZHWG08] and BVHs [LGS+09]. In this thesis we present parallel algorithms for fast construction of uniform grids and a hierarchical variation of them. We show that these structures and their construction map well to the hardware architecture and the parallel model exposed trough CUDA.

1.4 CUDA

This work is about parallel algorithms for GPU ray tracing. To be able to asses performance of the proposed methods we implement a full ray tracing framework in CUDA [NVI09].

CUDA is a programing model that exposes the processing capabilities of modern NVIDIA GPUs. In terms of programing language CUDA is an extension of C++, which offers the possibility to write normal C++ programs *(host code)* and *kernels (device code)*. The CUDA kernels are functions that are executed on the graphics card. Because of hardware limitations not all C++ features are supported in the code executed on the device. Those include dynamic memory allocation and virtual function calls. Memory allocation on the device is performed prior to kernel invocation trough the CUDA API.

The kernel code is executed in multiple threads in parallel. Those are grouped in thread blocks. Threads in the same block share fast on-chip memory - the *shared memory*. It can be used as a cache or to communicate between threads. To communicate between threads from different blocks one has to use the much slower *global memory*.

We spare us a more detailed introduction to CUDA and point the reader to the CUDA Programing Guide [NVI09] by NVIDIA and the paper by Nickolls et al. [NBGS08] for further reading.

Chapter 2

Introduction

The main motivation behind this work is the development of a fast, ray tracing based, rendering algorithm for dynamic scenes. Additionally we aim at approaches that work well for global illumination computations based on ray tracing. These often produce *incoherent* rays, i.e. rays that demand different computations or access different data during traversal and are therefore difficult to process efficiently on SIMD/SIMT architectures. While our grid based ray tracing algorithms benefit from ray coherence, we are able to maintain reasonable SIMD utilization also in cases where between-ray coherence is not present.

2.1 Overview

The remainder of this thesis is structured as follows. After a short introduction to uniform grids and their use as acceleration structures for ray tracing we introduce our mapping of the ray traversal implementation to a GPU. This is followed by the description of three fast and parallel grid construction algorithms. First we show how we can reduce the problem of uniform grid construction to sorting, which makes it massively parallel and maps very well to modern GPUs. Then we extend the approach to make a builder that can process multiple uniform grids in parallel and use the new algorithm to construct two-level hierarchical grids. The third algorithm constructs the two-level structure lazily. We try to separate out the set of visible primitives which allows us to construct the structure only where needed. This last approach is our attempt of finding an algorithm that maps well to the parallel GPU architecture and scales well for large and complex dynamic scenes.

2.2 Grids for Ray Tracing

In this thesis we consider two types of acceleration structures for ray tracing - *uniform grids* and *two-level grids* (see Figure 2.1).

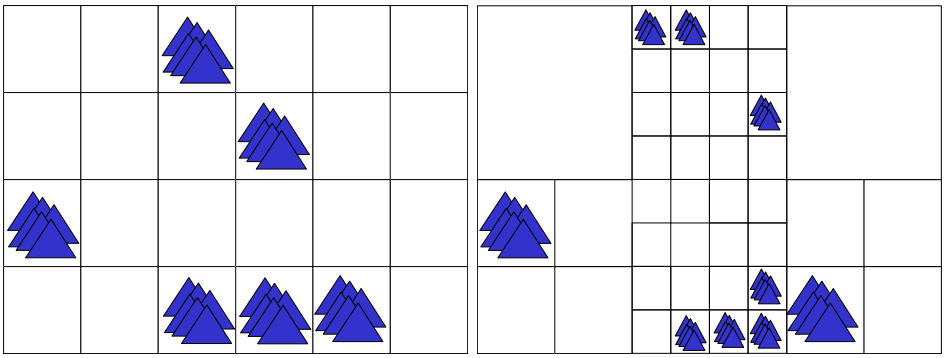

Figure 2.1: *Schematic Representation of two types of grids - uniform (left) and two-level (right) for the same scene. Only empty and non-empty cells are denoted. Each of the top level cells of the two-level grid is a uniform grid itself.*

Uniform Grids

The uniform grid structure is a regular spatial subdivision. The bounding box of the grid is the same as the bounding box of the scene. The grid is subdivided uniformly along each dimension (hence the name "uniform grid"), but the number of cells in each dimension is arbitrary. We discuss strategies of choosing optimal grid resolution for a given scene later on.

The Teapot in the Stadium

Grids have a disadvantage in terms of quality for ray tracing compared to other adaptive hierarchical acceleration structures like kd-trees. The amount of intersection candidates for an arbitrary ray that the structure is able to eliminate depends on the primitive distribution in the scene. Being unable to adapt to the local density of primitives, grids suffer from the so-called "Teapot in the Stadium Problem". It can happen that many or almost all primitives fall into the same grid cell while most of the remaining cells are empty. If a ray hits the cell in question one has to perform intersection tests with all primitives inside it. In this case the structure fails to achieve its purpose - to accelerate the traversal of the ray. To deal with this disadvantage of uniform grids we investigate a slight variation of them - two-level hierarchical grids.

Two-Level Grids

The two-level grid is an uniform grid with each cell being an uniform grid itself (see Figure 2.1). The resolution of each top-level cell is independent, allowing the structure to adapt to the local primitive density. While this

adaptivity is limited, constructing an example where this limitation is apparent is not easy. We will see later on that the two-level grid is not sensitive to primitive distributions and similarly to the uniform grid is simple and fast to traverse and construct.

The two-level grid sturcture is a special case of the family of hierarchical grids described by Jevans and Wyvill in [JW89], with the exception that we fix the depth of the tree to two.

Chapter 3

Ray Tracing with Grids

In this chapter we present our ray traversal algorithms. We implement them in CUDA and test them on a CUDA capable GPU. The programing language exposes a model in which the program (here the ray traversal) is executed in multiple threads and each of them is more or less independent from the others. However the underlying hardware has SIMD architecture and we have to take this into account as well if we are to deliver an efficient implementation. This is why we discuss notions like *ray packets, coherence* and describe an optimization that helps us to improve the SIMD efficiency of the implementation.

3.1 Packets and Ray Coherence

Tracing a ray in a scene(see Algorithm 1) is usually defined as the task to find if there exists a primitive intersected by the ray and (optionally) which is the primitive closest to the ray origin. To this end an acceleration structure is traversed and the primitives that are stored in the leafs (or cells) intersected by the ray are tested against the ray for intersection.

Implementing Algorithm 1 in CUDA and executing the program results in a straight forward parallelization, but it fails to exploit the parallel hardware when the rays are not *coherent*. Incoherent rays are rays that require different instructions or access different data during traversal. Examples for coherent rays are camera rays or shadow rays from a point light source. They share the same origin and have similar directions, which means that they are likely to visit the same parts of the scene and traverse common nodes in the acceleration structure. Rays reflected from a curved surface or diffuse inter-reflection rays are usually incoherent. In practice they almost never have common origin and their directions vary a lot.

Incoherence in a set of rays becomes a problem when the rays are traversed in *packets*. Usually rays are assigned to threads. These are grouped into blocks, which consist of warps. The threads of each warp are executed in

Algorithm 1 Tracing a ray

```
    procedure TRACE(ray, acc, scene)
 2:     p ← NULL                                  ▷ Closest primitive
        d ← ∞                                     ▷ Distance to p
 4:     if OUTSIDE SCENE(acc, ray) then
            return (p, d)
 6:     end if
        while HAS MORE LEAFS(acc, ray) do
 8:         cell ← GET NEXT LEAF(acc, ray)
            for ALL q ∈ GET PRIMS(cell) do
10:             t ← INTERSECT(ray, q)
                if t < d then
12:                 (p, d) ← (q, t)
                end if
14:         end for
            if d ≤EXIT DISTANCE(cell) then
16:             return (p, d)
            end if
18:     end while
        return (p, d)
20: end procedure
```

SIMD making the warps the ray packets. Incoherence can cause performance degradation when threads inside a warp disagree on which branch in the code to take. Having a SIMD architecture means that the same instruction must be executed for all threads in the warp. If branch divergence occurs inside one warp, both code paths are executed sequentially for all threads. For each path there are active and inactive threads. The results of the computations for the inactive threads are thrown away.

3.2 Coherence in Uniform Grids

Uniform Grids are well suited for the GPU architecture since the traversal is very simple and the procedure of advancing trough the cells does not produce any control-flow divergence. We used the three-dimensional DDA traversal algorithm by Amanatides and Woo [AW87]. Since at each traversal step is loaded a cell which is neighbouring to the current one, the algorithm is cache-friendly and we also expect the (already good) performance to increase in the case of the introduction of hardware support for 3D textures for next generation GPUs.

Hierarchical structures such as kd-trees and BVH usually introduce larger traversal overhead than grids, but eliminate more intersection can-

didates. This does not hurt performance as much when tracing coherent rays in packets, since the operations for traversal can be amortized (i.e. performed once for the entire packet). On the other hand packet based algorithms are more sensitive to ray coherence. The grid traversal by Amanatides and Woo is very cheap and does not rely on ray coherence to achieve good performance. Nevertheless a GPU implementation also benefits from ray coherence. It is easy to show that following holds for a packet of rays with origins in the same grid cell.

Statement:Let R_o be a set of rays, all of which have origin (not necessarily the same) in a given grid cell o. If a cell c is intersected by any ray $r \in R_o$ and is the n-th cell along r counted from the origin, then c will be n-th cell along any ray $r_2 \in R_o$ that intersects c.

Proof: Let o be the 0-th cell along the rays in the packet. For each cell c holds that it is at distance (x, y, z) from o, where x, y and z are integers and each denotes the distance (in number of cells) along the coordinate axis with the same name. During traversal the rays advance with exactly one neighbour cell at a time and the two cells share a side. This means that c will be traversed at $x + y + z = n$ -th step of the traversal algorithm for any ray with origin in o if at all □

For the GPU implementation of the 3D DDA grid traversal this means that without having to do any additional computations, one can guarantee that all rays in the (coherent) packet will arrive at the same time at a given cell and because of the way memory operations are managed on the GPU, the primitives contained in the cell will be loaded only once for the entire packet.

3.3 Incoherence in Uniform Grids

As explained in Section 3.1, traversal of incoherent rays poses a challenge on wide SIMD machines such as modern GPUs. Incoherence can occur even in the case of primary rays when part of the rays in the packet (warp) are terminated because they hit a primitive, but is almost always the case when processing secondary rays (Figure 3.1). Mostly because of incoherence tracing secondary rays can be several times more time-consuming than tracing primary rays (see [AL09] and our results later on).

In cases where the ray coherency is low, tracing packets of rays can become disadvantageous. An important observation is that having only few active threads (rays), does not mean that the computations cannot be parallelized. At the ray-triangle intersection stage of the algorithm (Line 9 in Algorithm 1) one has to test one or more rays with one or more primitives each. In this part of the algorithm it can happen that performing intersection test for one ray and many triangles offers better utilisation.

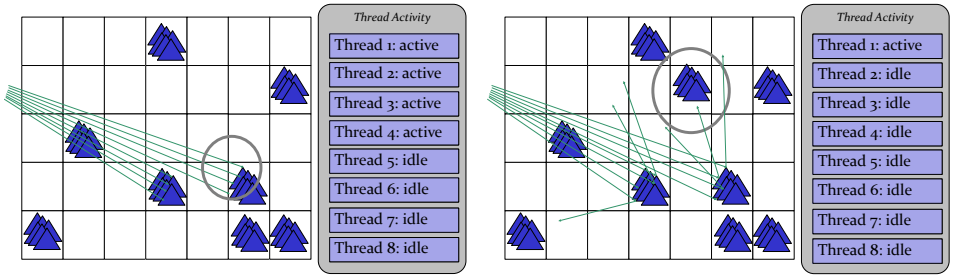

Figure 3.1: *Control flow divergence caused by early ray termination (left) and for secondary rays (right). Since one thread is mapped to exactly one ray, hardware utilization is smaller after some of the rays are intersected while others are not. In the example on the right only one thread is active and will perform intersection tests sequentially, even though these can be made in parallel.*

3.3.1 Parallel Intersection for Single Ray

In the following we describe an intersection algorithm that can test multiple triangles for intersection with one ray. In his PhD thesis [Wal04], Wald describes such approach for 4-wide SIMD, but finds it inefficient due to the overhead introduced by the need of data reordering. We use this approach as a building block for the hybrid intersection algorithm in the next section.

Performing intersection tests for the same ray in parallel introduces the need of communicating between threads. First of all the ray data must be available to all threads that will be computing the intersection. In our implementation we store the ray origins and directions in the shared memory of the GPU. We preferred this to be the best solution, instead of using global memory, which is also shared among threads but accesses have very high latencies. The disadvantage of using so much shared memory per thread is that it limits the occupancy. For our algorithm we could achieve 50% occupancy which is similar to what others (e.g. [GPSS07]) report and should be enough for hiding memory latencies. Because all ray data is in the shared memory the only thing that is left to do is to broadcast the index of the ray that has to be processed in a shared memory cell.

The second piece of information that has to be communicated between threads is the intersection distance and the indices of the primitives that are intersection candidates. We do not need to additionally store each candidate (or its index) since they are stored in a consecutive order in each cell of the accelerations structure. This allows us to store only the index of the first intersection candidate and deduce the index of the others by the id of the

Algorithm 2 Parallel Intersection for Single Ray

```
    shRay[]                                          ▷ Ray Data
 2: shDist[]                                         ▷ Distance to Intersection
    procedure SRI(shRayId, shStartId)
 4:     threadId ← inWarpId()
        ray ← shRay[shRayId]
 6:     primId ← shStartId + threadId
        q ← get Primitive(scene, primId)
 8:     shDist[threadId] ←intersect(ray, q)
        reduce(shDist, (q, d))
10:     return (q, d)
    end procedure
```

thread in the warp. To additionally simplify the implementation, we fix the amount of intersection tests per ray. For simplicity we will use the warp size (32) as this fixed number for the description. Our current implementations makes 16 tests for 2 rays in parallel to keep all the 32 threads in the warp busy. After the test the distances to the respective primitives are stored in an array in shared memory and a single reduction is used to determine the closest intersection (Line 9 in Algorithm 2).

3.3.2 Hybrid Intersection Algorithm

While the parallel intersection procedure (Algorithm 2) does not rely on ray coherence it has several drawbacks compared to the packet intersectors. It introduces significant overhead because of the between-thread communication and it can be applied only when a leaf of the acceleration structure contains some fixed amount of primitives (16 in our implementation). It makes no sense to use Algorithm 2 for leaves containing small amount of primitives because the benefit will be smaller than the introduced overhead.

We implemented a combination of both algorithms. The new hybrid approach uses packet intersection when all threads in the warp are active. In the case where there would be inactive threads during intersection we check if there are rays with *large workload*. In this case we gather all such rays and perform intersection tests for each of them in parallel. There is a third case: when not all threads are active and none of the active threads has large workload. If this happens we are only able to use the packet intersection test. To sum up our approach is to use packet intersection everywhere, except for places where we are sure that single ray parallel intersection will be faster.

To make a decision, which intersection algorithm to choose, amounts to determining how many threads will be active and if there is a thread with large workload. This can be easily done with warp vote functions which are

Algorithm 3 Hybrid Ray - Primitive Intersection

```
   shRay[]                                                    ▷ Ray Data
 2: procedure INTERSECT(cell, (p, d), scene)
      startId ← FIRST(cell)
 4:   endId ←END(cell)
      if __any(endId − startId = 0) AND __any (startId − endId ≥ 32)
   then
 6:                                                ▷ Single Ray Intersection
        shRayId[]                                          ▷ Ray-Id-Buffer
 8:     shStartId[]                                        ▷ Prim-Id-Buffer
        shDist[]
10:     threadId ← INWARPID()
        if startId − endId ≥ 32 then
12:       INSERT(threadId, shRayId[])
          INSERT(startId, shStartId[])
14:     end if
        numRays ← SIZE(shRayId)
16:     for i ∈ [0, numRays) do
          rayId ← shRayId[i]
18:       pId ← shStartId[i]
          (q, t) ← SRI(rayId, pId)
20:       if threadId = shRayId[i] then
            if t < d then
22:           (p, d) ← (q, t)
            end if
24:       end if
        end for
26:   else
                                                  ▷ Packet Intersection
28:     for ALL id ∈ [startId, endId) do
          q ← GET PRIMITIVE(scene, id)
30:       t ← INTERSECT(ray, q)
          if t < d then
32:         (p, d) ← (q, t)
          end if
34:     end for
      end if
36:   return (p, d)
   end procedure
```

available for CUDA devices with capability 1.2 or higher (Algorithm 3).

Additional hardware functionality can further reduce the overhead of reorganizing the data. Note that we perform a prefix sum computation inside a warp for the insertion in Line 12. This part of the computations would be much faster if such instruction was supported in hardware. Also the warp vote functions on Line 5 can be exchanged with a single population count operation. Knowing how many of the threads have high workload will allow us determine exactly when the single ray intersection will be more efficient. Our test for the current implementation showed that this is the case if less than 12 threads inside a warp have sufficient workload. Currently we only compute if there is at least one thread with high workload and use the single ray intersection in this case.

One should also note that all threads in the warp must be active when calling the intersection routine. To this end one must use warp voting functions to steer the control-flow in all loops and branches outer to the intersection routine. We discuss further implementation details in the next section.

Results

Model (# Tri)	Primary SI, UG	Path Trace SI, UG	Path Trace HI, UG	Path Trace SI, TLG
erw6 (800)	18ms	137ms	185ms	136ms
Ruins (38K)	28ms	428ms	441ms	556ms
Ogre (50K)	40ms	431ms	393ms	400ms
Sponza (60K)	23ms	470ms	450ms	712ms
Conference (284K)	92ms	1100ms	904ms	861ms
Venice (1.2M)	107ms	1520ms	1088ms	1185ms

Table 3.1: *Performance for primary rays and path tracing of a* 1024×1024 *image with one path per pixel. We compare path tracing with simple grids using a simple (SI) and our hybrid (HI) ray-triangle intersector with an uniform grid (UG) and the two-level grid (TLG), for which using different intersection algorithms made almost no difference. We suspect this is the case because there are only a few cells containing many primitives. All timings are from a GTX285.*

To have a benchmark with incoherent rays we implemented path tracing [Kaj86]. We generate secondary ray directions using cosine distribution and terminate the paths based on random decision (Russian Roulette) with probability $\frac{1}{2}$. At the end of each path we shoot a shadow ray and sample

Figure 3.2: *Four examples of diffuse path tracing with 300 paths per pixel. From left to right - Ruins, Ogre, Conference and Venice*

direct illumination. The generated rays are highly incoherent after the first bounce, because of path termination and because of the distribution for the directions of the generated secondary rays. The performance measures (Table 3.1) show performance boost due to the use of the hybrid intersector, but the overhead introduced by the need for between-thread communication is still very large. We expect the algorithm to perform a lot better on a hardware with support for population count and prefix sum on warp level.

3.4 Ray Traversal Implementation

To be able to test the performance of the algorithms described here, we implemented a ray tracing framework in CUDA. We briefly describe some important aspects of this implementation. Otherwise an interpretation of our results and fair comparison to other approaches will be difficult to make.

3.4.1 Uniform Grid Traversal

Our ray tracing implementation does not make explicit use of packets, frusta, or mailboxing like Wald's [WIK+06]. We used the traversal algorithm proposed by Amanatides and Woo [AW87] without any significant modifications, and store the grid cells in a 3D texture in order to make better use of the texture cache during traversal. Since arithmetic operations are not that expensive on the GPU especially when compared to memory operations, we tried to reduce the register usage by not storing all data required for the traversal. We recompute the cheapest terms each time we need them. In our implementation we only reserve 6 registers - 3 for the cell index and 3 for the $tMax$ variable, which tels us the value of the ray parameter t, for which the ray intersects the next slice of cells in each dimension.

3.4.2 Two-level Grid Traversal

The traversal of the two-level grid is very similar to the one of the uniform grid. We use the Amanatides and Woo [AW87] traversal both to step trough the top level cells and to traverse the leaf level cells if there are any. We

store the top-level cells in a 3D texture and all leaf level cells in a single 1D texture.

3.4.3 Triangle Intersection

Our hybrid intersection algorithm described in Section 3.3.2 describes only how a ray-triangle intersection can be mapped to a multi-threaded implementation. The actual algorithm that we used to determine whether or not the ray hits the triangle is the algorithm by Möller and Trumbore [MT97]. Besides being very fast this approach does not rely on any precomputations which is essential when rendering dynamic scenes.

We tried other methods like the algorithm by Shevtsov and et al. [SSK07], but they depend on precomputed data for all triangles for each frame. This reduces the overall performance and makes the algorithm less scalable for large scenes where most of the triangles are not rendered and the precomputations are wasted.

The Möller-Trumbore intersection algorithm worked best with our triangle representation in which a triangle is stored as three indices to a global vertex array. We also store the triangles and the vertex array in textures since this not only reduced the memory footprint of the program but also improved the rendering time.

3.4.4 Persistent Threads

The current hardware architecture of GPUs targets algorithms with homogeneous workloads in which threads require similar run times to execute their tasks. This is not the case in ray tracing since depending on the part of the scene being intersected by the ray the runtime for traversal may vary much. It turns out that this can be a major performance bottleneck when rays or tasks are assigned to threads statically because the execution of a single thread may hold the hardware resources busy, while all other threads are idle. Aila and Laine [AL09] propose a solution called “persistent threads”. Instead of assigning rays to threads prior to the kernel invocation rays are fetched by threads from a global queue. One needs to start enough threads to saturate the hardware resources and these process all rays. Although Aila and Laine used a BVH-based ray tracer in the paper we could observe the same performance increases in our system.

3.4.5 Results

The focus of this thesis is the analysis of the trade offs between build time and rendering performance provided by the acceleration structures. This is why we prefer to present the performance of our rendering algorithms for uniform and two-level girds together with the performance of the construction algorithms for the corresponding structures.

Chapter 4

Uniform Grid Construction

There has been research for ray tracing with grids and grid construction on CPUs, but we were not aware of how things look like on a graphics card. While no one expects good acceleration for ray tracing from uniform grids, they should be very fast to construct, especially on a highly parallel architecture, that has the computational power and bandwidth of modern GPUs. Unfortunately previous parallel construction algorithms either rely on atomic synchronization to resolve write conflicts, or cannot guarantee good work distribution. These problems do not influence performance as much on a multi-core CPU system, but one needs very high parallelism in order to exploit the resources of a graphics card. In the following we describe our attempt to figure out a massively parallel grid construction algorithm [KS09].

4.1 Previous Work

While fast grid construction algorithms for CPUs have been investigated [IWRP06, LD08], to our knowledge, there are no attempts to efficiently implement a *construction* algorithm for a highly parallel architecture such as the GPU. Eisemann and Décoret [ED06] propose to use GPU rasterization units for *voxelization* of polygonal scenes in a grid. Their approach is limited to computing a *boundary representation* of the scene, which is only a part of the information required for ray tracing. Patidar and Narayanan [PN08] propose a fast construction algorithm for a grid-like acceleration structure, but their algorithm is limited to fixed resolution and while it performs well for scanned models, it relies on synchronization via atomic functions which makes it sensitive to triangle distributions. Ize et al. [IWRP06] describe a number of parallel construction algorithms for multiple CPUs. Their sort-middle approach also does not rely on atomic synchronization, but the the triangle distribution in the scene influences the work distribution and the algorithm performance.

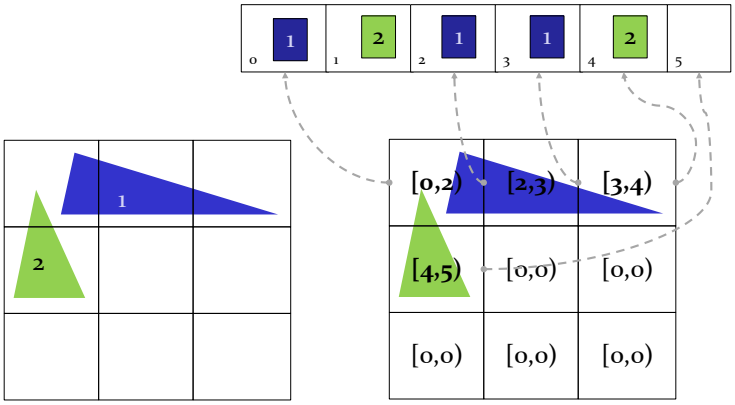

Figure 4.1: *The data structure consists of two parts. We have an array of primitive references. Each primitive is pointed to as many times as the number of cells this primitive overlaps. The grid cells are ranges in the cell array. For example the upper left cell contains triangle 1 and 2, and its range in the array contains references to these triangles.*

The idea of reducing the construction process to sorting has been used by Lauterbach et al. [LGS+09] for their LBVH structure. In the particle simulation demo in the CUDA SDK [Gre08] sorting is used for construction of an uniform grid over a set of particles. The approach described here, and concurrently proposed by Ivson et al. [IDC09], is more general because it handles geometric primitives overlapping any number of cells. This allows for construction of grids that can be used as acceleration structures for ray tracing geometric surfaces.

4.2 Data Structure

Like the compact grid representation by Lagae and Dutré in [LD08], we store the structure in two parts (Figure 4.1). An indirection array contains triangle references. The grid cells are stored separately. Each grid cell stores the beginning and the end of a range inside the array such that the triangles referenced in this interval are exactly those, contained in the cell. In Lagae's representation, a single index per cell is stored. This index is both the beginning of the interval of the current cell, and the end of the interval for the previous. We store independent cell intervals which doubles the memory consumption but simplifies the parallel building process, by

Algorithm 4 Data-Parallel Grid Construction. Kernel calls are suffixed by <<< >>>.

```
    b ← COMPUTE BOUNDS()
 2: r ← COMPUTE RESOLUTION()
    t ← UPLOAD TRIANGLES()
 4: G ← 128, B ← 256, A_refCounts ← ARRAY OF G + 1 ZEROES
    A_refCounts ← COUNT REFERENCES<<<G,B>>>(t, b, r)
 6: A_refCounts ← EXCLUSIVE SCAN<<<1,G + 1>>>(A_refCounts)
    n ← A_refCounts[G]                          ▷ Number of References
 8: A_pair ← ALLOCATE PAIRS ARRAY(n)
    A_pair ← WRITE PAIRS<<<G,B>>>(t, b, r, i)
10: A_pair ← SORT(A_pair)
    cells ← EXTRACT CELL RANGES<<< >>>(A_pair, n)
```

allowing us to initialize all cells as empty prior to the build and then only touch the non-empty cells.

4.3 Algorithm

Constructing a grid over a scene consisting of triangles (or any other type of primitive), amounts to determining the bounding box of the scene, the resolution of the grid in each dimension and, for each cell of the grid, which triangles overlap it. Our construction algorithm (Algorithm 4) consists of several steps. First we compute an unsorted array that contains all primitive-cell pairs. This array is sorted and the grid data is extracted from it in a final step. The same idea is used in the particle simulation demo in the CUDA SDK [Gre08]. The only difference is that each of the primitives handled by our algorithm can overlap arbitrary number of cells.

4.3.1 Initialization

Once the bounding box of the scene and the grid resolution is determined on the host, we upload the primitives to the GPU. Since computing the bounding box involves iteration over the scene primitives, we perform this operation while reorganizing the data for upload.

4.3.2 Counting Primitive References

Because it is not possible to dynamically allocate memory on GPUs, we have to know the size of the array that stores the primitive references in advance. To compute it, we first run a kernel that loads the scene primitives in parallel and for each determines the number of cells it overlaps (Line 5). Each thread writes the counts into an individual shared memory cell and then a reduction

is performed to count the total number of primitive-cell pairs computed by each thread block. Next we perform an exclusive scan over the resulting counts to determine the total number of primitive-cell pairs and allocate an output array on the device. The scan additionally gives us the number of pairs each block would output.

4.3.3 Writing Unsorted Pairs

Having the required memory for storage, we run a second kernel (Line 9). Each thread loads a primitive, computes again how many cells it overlaps and for each overlapped cell writes a pair consisting of the cell and primitive indices. The output of the exclusive scan is used to determine a segmentation of the array between the thread blocks. We have to avoid write conflicts inside a block since each thread has to write a different amount of pairs. We can use the shared memory to write the pair counts and perform a prefix sum to determine output locations. In our implementation each thread atomically increments a single per-block counter in shared memory to reserve the right amount of space.

4.3.4 Sorting the Pairs

After being written, the primitive-cell pairs are sorted by the cell index via radix sort. We used the radix sort implementation from the CUDA SDK examples for this step of the algorithm. However there are faster radix sort implementations, for example the one by Billeter et al. [BOA09] is more than two times faster than the one we used. Using a faster radix sort implementation will have a significant impact on the overall performance of the algorithm.

4.3.5 Extracting the Grid Cells

From the sorted array of pairs it is trivial to compute the reference array as well as the triangles referenced in each cell. We do this by invoking a kernel that loads chunks of the sorted pairs array into shared memory. We check in parallel (one thread per pair) if two neighboring pairs have different cell indexes. This indicates cell range boundary. If such exists, the corresponding thread updates the range indexes in both cells. Note that in this stage of the algorithm only non-empty cells are written to. After this operation is completed the kernel writes an array that stores only the primitive references to global memory. In this part of the implementation we read the data from shared memory and the writes to global memory are coalesced, so the introduced overhead is very small. It is also possible to directly use the sorted pairs to query primitives during rendering. However getting rid of the cell indices frees space, and accesses to the compacted data during rendering are more likely to get coalesced.

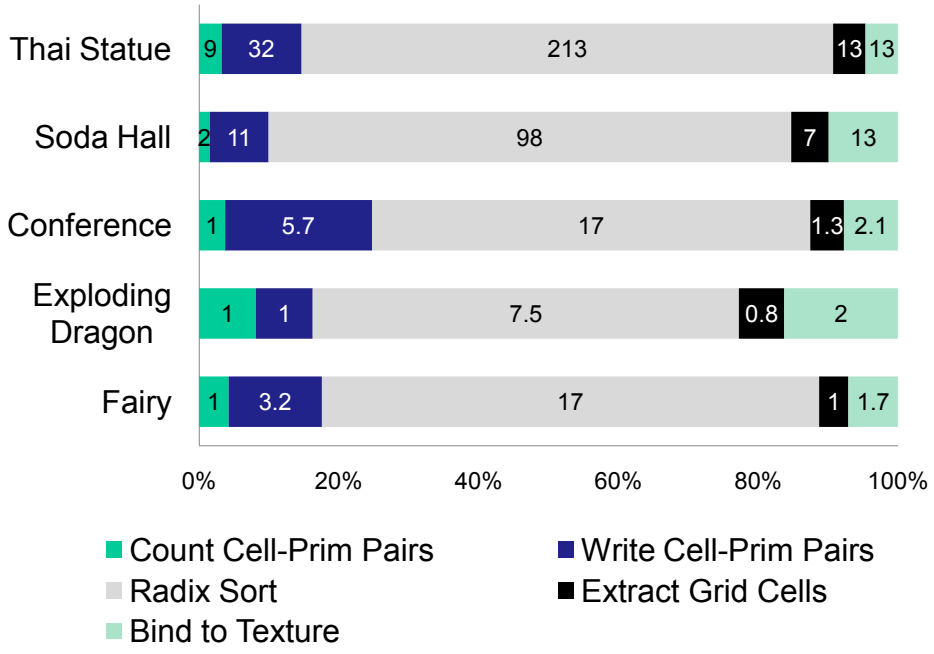

Figure 4.2: *Times for the different stages of the build algorithm in milliseconds. We also include the time needed to bind the grid cells to a 3D texture.*

4.4 Triangle Insertion

In our algorithm we have to compute which cells are overlapped by each input triangle twice. When counting the total number of references (Line 5) we conservatively count the number of cells overlapped by the bounding box of the triangle. Afterwards, when we want to write triangle-cell pairs (Line 9), we do a more precise (but still efficient) test. We check if each cell overlapped by the triangle bounding box is intersected by the plane in which the triangle lies. An exact triangle-box overlap test [AM01] did not pay off for any of the tested scenes.

We test our implementation only with scenes consisting of triangles, but the same approach can be used for various geometric primitives. The only requirement is that one can (efficiently) determine the cells of the grid that each primitive overlaps.

4.5 Analysis

Both the complexity of the algorithm and the performance of the implementation are dominated by the sorting of the primitive-cell pairs (Figure 4.2). Under the assumption that the number of cells overlapped by each triangle can be bounded by a constant, all parts of the algorithm have linear work complexity. We chose to use radix sort because it is well suited for the data we have, it also has linear work complexity, and there are fast and scalable GPU implementations [SHZO07, BOA09]. Sorting on the device alleviates the need for expensive data transfers. The only information we must communicate to the CPU during construction is the size of the references array so that we can allocate it.

An important advantage of our algorithm is that there are no write conflicts, and hence, no atomic synchronization is required throughout the build. This implies that the performance of the construction algorithm depends only on the number of primitive references that are inserted in the grid, and not on the primitive distribution in the scene. In fact, as discussed in Section 4.3.3, we use an atomic operation on shared memory when we write output pairs. This however is neither necessary (can be done efficiently via prefix sum), nor performance critical since we require a single atomic operation per primitive and not per primitive insertion in a cell.

The memory requirements for the grid and its construction can become a concern when dealing with very large models. Our method requires additional (but temporary) memory for storing primitive-cell pairs instead of only primitives. We additionally need a second array of pairs during sorting. After the sort stage we extract the primitive references from the sorted array and free the additional space. The main memory bottleneck is the space for storing the grid cells. Each of them is 8 bytes large and we also store empty cells. Despite the relatively large memory footprint, we were able to construct grids for all models that we tested, including the Thai Statue which has 10 million triangles. We discuss a memory issue that we had with this model in the results section.

Even if there is not enough memory for the grid cells, one can modify the algorithm to construct the acceleration structure incrementally. We have not implemented this since the size of the grids and the added memory transfers to the CPU and back will most likely result in build times of more than a second.

4.6 Grid Resolution

An important part of the building process is the choice of the grid resolution. This is the only factor one can vary in order to influence the quality of the structure for ray tracing. Sparser grids cannot eliminate as many intersec-

Scene	Thai Statue	Soda Hall	Conference	Dragon	Sponza	Ruins
Default	$325 \times 547 \times 280$	$262 \times 274 \times 150$	$210 \times 133 \times 50$	$104 \times 147 \times 65$	$116 \times 55 \times 52$	$53 \times 69 \times 52$
Cost-Based	$313 \times 487 \times 281$	$256 \times 268 \times 164$	$164 \times 110 \times 50$	$137 \times 105 \times 71$	$115 \times 67 \times 63$	$60 \times 69 \times 59$

Table 4.1: *Grid resolutions computed via heuristic (Default) and cost-based approach (Cost-Based). Instead of trying to make the grid cells as close to a cube as possible, one can try to find a resolution that minimizes the expected cost for tracing a random ray trough the grid.*

tion candidates but a higher resolution results in bigger cost for traversal. The resolution is typically chosen as:

$$R_x = d_x \sqrt[3]{\frac{\lambda N}{V}}, R_y = d_y \sqrt[3]{\frac{\lambda N}{V}}, R_z = d_z \sqrt[3]{\frac{\lambda N}{V}} \tag{4.1}$$

where $\vec{d}$ is the size of the diagonal and V is the volume of the scene's bounding box. N is the number of primitives, and λ is a user-defined constant called *grid density* [Dev88, JW89]. Like Wald et al. [WIK+06], we set the density to 5 in our tests. A more extensive study on good choices of grid density is done by Ize et al. [ISP07]. In the following we describe another approach to choosing resolution of uniform grids.

MacDonald and Booth [MB90] introduced the Surface Area Metric for measuring the expected cost of a spatial structure for ray tracing. Given the cost for traversing a node of the structure C_t and the cost for testing a primitive for intersection C_i, the expected cost for tracing a ray is

$$C_e = C_t \sum_{n \in Nodes} Pr(n) + C_i \sum_{l \in Leaves} Pr(l) Prim(l) \tag{4.2}$$

$Pr(n)$ and $Pr(l)$ are the probabilities with which a random ray will intersect the given node, $Prim(l)$ is the number of primitive stored in the leaf l. In the case of grids, since all cells are regarded as leaves and have the same surface area (i.e. same intersection probability), Equation 4.2 simplifies to

$$C_e(G) = C_t N_c + C_i \frac{SA(c)}{SA(G)} N_{pr} \tag{4.3}$$

where $SA(c)$ and $SA(G)$ are the surface areas of a cell and the grid's bounding box, N_{pr} is the number of primitive references that exist in the grid, and N_c is the expected number of grid cells intersected by a random ray. The surface areas of a cell and the grid can be computed in constant time, and N_c can be bounded by the sum of the number of cells in each dimension. The only non-trivial part for computing the expected cost is counting the number of primitive references in the grid. Since we were able to estimate this number relatively fast (Algorithm 4, Line 5), we tried to find the best grid resolution for several test scenes. We empirically found

Scene	Tris	References	Resolution	Time
Thai Statue	10M	19M	$325 \times 547 \times 280$	417
Thai Statue	10M	14.8M	$192 \times 324 \times 168$	280
Soda Hall	2.2M	6.7M	$262 \times 274 \times 150$	130
Conference	284K	1.1M	$210 \times 133 \times 50$	27
Dragon	180K	0.7M	$104 \times 147 \times 65$	16
Fairy Forest	174K	1.1M	$150 \times 38 \times 150$	24

Table 4.2: *Build statistics for test scenes of different sizes. Times are in milliseconds and are measured on a GTX280.*

that $C_t = 1.5$ and $C_i = 8.5$ work well for our rendering algorithm and used Equation 4.1 to have an initial estimate $\vec{r}$. We exhaustively tested all possible grid resolutions in the range $\left(\frac{3}{4}\vec{r}; \frac{5}{4}\vec{r}\right)$.

While the resulting grids (Table 4.1) improved rendering performance for our ray tracer, the differences were very small both for primary rays and for path tracing of diffuse surfaces. For example the average number of intersection tests per ray for Sponza and Ruins was reduced by up to 2 which is around 10%. Also the times for cost estimation allowed computing the cost-based resolutions only in a preprocessing stage of the algorithm. Unless noted, we used the default grid resolutions for all tests.

4.7 Results

We implemented our construction algorithm as a part of a GPU ray tracer in the CUDA programming language (see Chapter 3). All tests were performed on a machine with an NVIDIA Geforce 280 GTX with 1 GB memory and a Core 2 Quad processor running at 2.66 GHz. The only computationally demanding tasks for which we use the CPU are key frame interpolation for dynamic scenes and data transfers.

4.7.1 Construction

From the results in Table 4.2 one sees that the performance of the construction algorithm scales with the number of references in the grid. The reported times are for the runtime of the construction algorithm and include all overheads for memory allocation and deallocation, the computation of the grid resolution, and a texture bind to a 3D texture, but do not include the time for the initial upload of the scene to the GPU. When rendering dynamic scenes, we use a separate CUDA stream for uploading geometry for the next frame while rendering the current one. We were able to completely hide the data transfer by the computation for the previous frame for

Model (Triangles)	LBVH GTX280	H BVH GTX280	Grid GTX280	Model (Triangles)	LBVH GTX280	H BVH GTX280	Grid GTX280
Fairy (174K)	10.3ms 1.8 fps	124ms 11.6 fps	24ms 3.5 fps	Conference (284K)	19ms 6.7 fps	105ms 22.9 fps	27ms 7.0 fps
Exploding Dragon (252K)	17ms 7.3 fps	66ms 7.6 fps	13ms 7.7 fps	Soda Hall (2.2M)	66ms 3.0 fps	445ms 20.7 fps	130ms 6.3 fps

Table 4.3: *Build times and frame rate (excluding build time) for primary rays and simple shading for a* 1024×1024 *image. We compare performance of Günther's packet algorithm [GPSS07] with LBVH and Hybrid BVH (H BVH) as implemented by Lauterbach et al. [LGS+09] to our implementation. See Table 4.2 for grid resolutions. The grid resolution of the Bunny/Dragon varies with the scene bounding box.*

Figure 4.3: *Some of our test scenes, from left to right - Ruins, Sponza, Conference, and Soda Hall. We can render these viewpoints at 25, 43, 11 and 8 fps with simple shading in a* 1024×1024 *window and can construct grids in 10, 13, 27 and 130 milliseconds on a GTX280.*

all tested animations. Note that we also update the bounding box of the scene together with the data upload.

The time to build the full resolution grid for the Thai Statue ($325 \times 547 \times 280$) does not include 198 milliseconds for copying the grid cells to the CPU and then back to a GPU-texture. We were not able to perform the texture bind directly, because this involves duplication of the grid cells (nearly 400 MB), for which the GPU-memory was not sufficient. Note that the copy and the texture bind are only necessary if the rendering must be done on the device and the grid cells must be stored in a three dimensional texture. We include the build time for the full resolution and the sparser grid in Table 4.2 for comparison. This resolution allows us to achieve reasonable rendering performance - between 3 and 5 frames per second. Ize et al. [IWRP06] report build times of 136 and 21 milliseconds for the Thai Statue ($192 \times 324 \times 168$) and the Conference on 8 Dual Core Opterons running at 2.4 GHz with bandwidth of 6.4 GB/s each.

4.7.2 Rendering

We compare build times and rendering performance to the results by Lauterbach et al. in [LGS+09] for their LBVH in Table 4.3. To remain fair we used

a naive and non-optimized implementation for the renderer. We did not include the persistent threads implementation (see Section 3.4.4) nor the hybrid intersector (see Section 3.3.2).

The LBVH has an advantage in terms construction time, but the rendering performance suggests that it does not offer better acceleration for ray tracing than grids. The quality disadvantage is bigger for the relatively large Soda Hall model.

The Hybrid BVH [LGS+09] offers fast construction times and between three and four times better rendering performance than our implementation of grids. Nevertheless the better construction time allows us to rebuild the structure and trace a significant amount of rays before the overall performance becomes worse.

Please note that the disadvantage in terms of rendering times that our implementation has is partly due to the fact that the alternative approach make explicit use of ray-packets. On the other hand, our approach is less sensitive to ray coherency.

Despite their disadvantages grids can provide an alternative to hierarchical acceleration structures if the primitive distribution is to some extent even, or in real-time applications in which the number of rays that have to be traced is not very large. While high-quality SAH-based acceleration structures enable faster ray tracing, their construction time is a performance bottleneck in dynamic applications. The fast build times of grids are almost negligible and shift the computational demand entirely toward tracing rays, a task which is easier to parallelize.

4.8 Conclusion

We presented a robust and scalable parallel algorithm for constructing grids over a scene of geometric primitives. Because we reduce the problem to the sorting of primitive-cell pairs, the performance does not depend on the triangle distribution in the scene. When used for ray tracing dynamic scenes, the fast construction times allow us to shift the computational effort almost entirely toward the rendering process. We also showed a method for choosing the resolution of the grid in a way that minimizes the expected cost for tracing a ray. Unfortunately this could not solve the problems that grids have with triangle distributions.

In the next chapter we discuss an acceleration structure type which is fast to build, but maintains high quality even for scenes with non-uniform triangle distributions.

Chapter 5

Two-Level Grid Construction

In the previous chapter we saw that uniform grids cannot offer good acceleration for ray tracing in scenes with non-uniform triangle distributions. Two extreme examples in this respect are the fairy and the conference scene in which the sensitivity to the "Teapot in the Stadium Problem" (see Section 2.2) is obvious. Still using an SAH hierarchy like the Hybrid BVH by Lauterbach et al. [LGS+09] will turn the construction time in the main performance bottleneck.

In this chapter we describe a modification of the uniform grid that handles non-uniform primitive distributions in a scene much better. The two-level grid is a fixed depth hierarchy consisting of a relatively sparse top level grid. Each of the cells of the grid is an uniform grid itself. Despite being restricted to a depth of two, the structure handled well all practical examples that we tried. More important we were able to reduce its construction to sorting as well, which resulted in several times faster build times and comparable rendering performance to the Hybrid BVH.

5.1 Data Structure

The representation of the two level grid (see Figure 5.1) is very similar to the one of the uniform grid. We store references to the primitives in the leaf cells in a single reference array. The second level cells (or leaf cells) are ranges in this array. We store all second level cells in a single *leaf array*. Each top level cell corresponds to a range in that array. We need to know the resolution of the grid in this cell as well as the place in the array where the leaf ranges are stored. We use two 32bit values for that. In the first we store a the array index of the first leaf cell, in the second we compress the cell resolution in 12 bits and leave out 4 bits for some flags that denote status like "is empty" or "has children". We use some of the flags during traversal and others are relevant for the construction algorithm described in the next chapter.

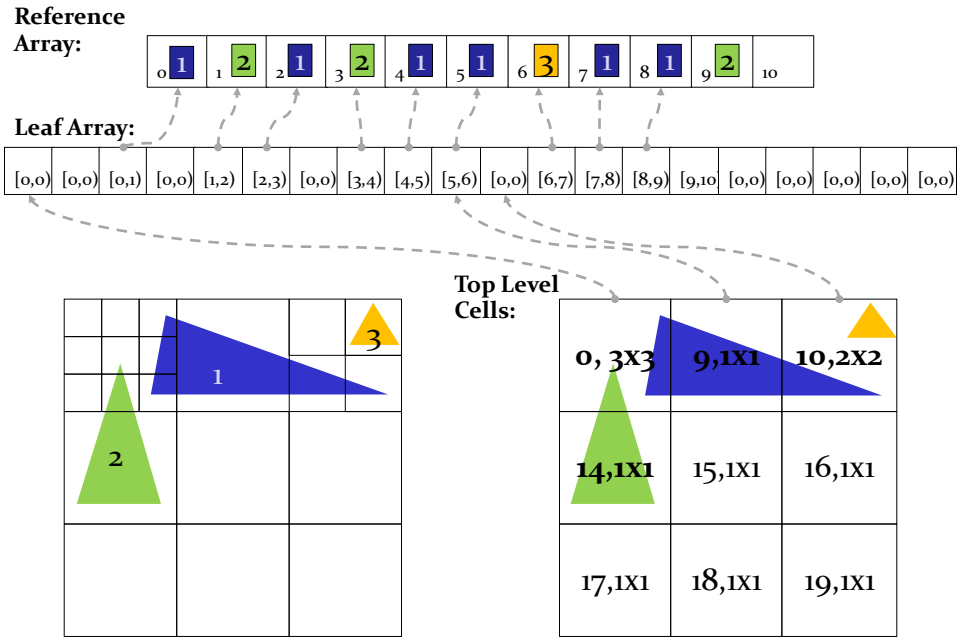

Figure 5.1: *The two level grid is represented by the top level cells (bottom right), the leaf array (second line) and the primitive reference array(top).*

5.2 Algorithm

We build the two-level grid top-down and like in the previous chapter we reduce the problem to sorting cell-primitive pairs (Algorithm 5). We avoid the need for any synchronization again and the work is easily distributed to an arbitrary number of processing units. What makes this algorithm interesting and very fast is that we are able to build the entire second level with a single sort. We do this by assigning false cell indices to the data that has to be sorted. In this way we directly construct the reference array for the whole second level.

5.2.1 Building the Top Level

We construct the top level of the grid exactly as we would build an uniform grid (see Chapter 4). The only difference is that we keep the sorted array of pairs (Line 5). Having determined the first level of the grid we build the second level. The basic idea behind the construction of the second level is to use an algorithm for parallel construction of multiple uniform grids.

Algorithm 5 Data-Parallel Two-Level Grid Construction. Kernel calls are suffixed by <<< >>>. Scan and sort may consist of several kernel calls.

$b \leftarrow$ COMPUTE BOUNDS()
$r \leftarrow$ COMPUTE TOP LEVEL RESOLUTION()
$t \leftarrow$ UPLOAD TRIANGLES()
$data \leftarrow$ BUILD UNIFORM GRID(t, b, r) ▷ **See Previous Chapter**
$A_{topPairs} \leftarrow$ GET SORTED TOP LEVEL PAIRS$(data)$
$n \leftarrow$ GET NUMBER OF TOP LEVEL REFERENCES$(data)$
$topCells \leftarrow$ GET TOP LEVEL CELL RANGES$(data)$
$\mathrm{G} \leftarrow (r_y, r_z), \mathrm{B} \leftarrow r_x$
$A_{cellCounts} \leftarrow$ ARRAY OF $r_x * r_y * r_z + 1$ ZEROES
$A_{cellCounts} \leftarrow$ COUNT LEAF CELLS <<<G,B>>>$(b, r, topCells)$
$A_{cellCounts} \leftarrow$ EXCLUSIVE SCAN$(A_{cellCounts}, r_x * r_y * r_z + 1)$
▷ **Set leaf cell pointers and cell resolution:**
$topCells \leftarrow$ INITIALIZE <<<G,B>>> $(A_{cellCounts}, topCells)$
$\mathrm{G} \leftarrow 128, \mathrm{B} \leftarrow 256$
$A_{refCounts} \leftarrow$ ARRAY OF $\mathrm{G} + 1$ ZEROES
$A_{refCounts} \leftarrow$ COUNT LEAF REFS<<<G,B>>>$(t, b, r, n, A_{topPairs}, topCells)$
$A_{refCounts} \leftarrow$ EXCLUSIVE SCAN$(A_{refCounts}, \mathrm{G} + 1)$
$m \leftarrow A_{refCounts}[\mathrm{G}]$
$A_{pair} \leftarrow$ ALLOCATE LEAF PAIRS ARRAY(m)
$A_{pair} \leftarrow$ WRITE PAIRS<<<G,B>>>$(t, b, r, n, A_{topPairs}, topCells, A_{refCounts})$
$A_{pair} \leftarrow$ SORT(A_{pair})
$leafCells \leftarrow$ EXTRACT CELL RANGES<<< >>>(A_{pair}, m)

5.2.2 Counting Leaf Cells

The first thing we do is to determine and write how many leaf cells contains each top level cell (Lines 8 to 10). We run a kernel with as many threads as there are top level cells. Each thread computes the grid resolution inside its cell and writes the number of cells in the output array. As in Section 4.6 the resolution of the cell is given by the size of its bounding box and the number of overlapped primitives.

The scan performed in Line 11 gives a segmentation of the leaf array between top level cells. This means that the n-th value is the position in the leaf array where the first leaf of the n-th top level cell is located. Now we have all information required for the top level cells - we know where to find the sub-cells and we already computed the resolution of the cells. We write this information in the cells in Line 13.

5.2.3 Counting Primitive References

Having initialized the data for the top level cells allows us to start writing the array of pairs that will later be converted to the final array with primitive references. We work on the array of pairs that we built while constructing the top level grid. The primitives are loaded together with the indices of the top level cell in which they need to be inserted. We lookup the resolution of the top level cell and count the number of sub-cells overlapped by the primitive as we did for the uniform grid (Section 4.3.2).

Using the same strategy as before we store the counts in shared memory and perform reduction when all primitives are processed. This and the scan on Line 17 gives us the size of the pairs array together with a segmentation for the next step of the algorithm.

5.2.4 Writing Unsorted Pairs

We allocate the memory required to store the cell-primitive pairs and start populating it. The result of the last scan segments the output between thread blocks and we resolve write conflicts within the same block with atomic incrementation of a shared memory counter. So far the only difference to the corresponding part of the uniform grid construction is that we use the top-level grid to load the primitives.

The other difference is in the information that we write in the output array. We do not write the real cell index, since it is not unique. In Figure 5.1 there are 9 leaf cells with index 0, 2 with index 1 and so on. We want a single sort to yield the final reference array. To this end we pair the location of the cell in the leaf array with the primitive. This location is given by the sum of the cell index and the start of the leaf array segment for the corresponding top level cell. The last we can directly look up.

5.2.5 Sorting the Pairs

As with the uniform grid the order of the primitive references in the sorted array is the same as in the final reference array. We used the same radix sort implementation but a faster one will have even greater impact on performance compared to the uniform grid construction. Still since the sorting stage is abstracted away both in the algorithm and its implementation it will be easy to replace the sort implementation with an optimal one for a given hardware architecture.

5.2.6 Extracting the Leaf Cells

We compute the leaf cells as we did this for the uniform grid. There we used the cell indices to search for the starts and ends of a range for a given cell. Having used pseudo-indices for pair generation is not a problem since these indices give us the locations of the cells in memory and are unique. So the cell ranges in the sorted array are exactly what we need. After we output the leaf array we extract the reference array out of the pairs array to compact the data and free space. This concludes the construction.

5.3 Analysis

Apart from being fast, the most important feature of the construction algorithm is that its work complexity is linear. This is the main reason to choose to use radix sort. The complexity is linear also due to the fact that we can determine the cell resolution in constant time from the number of overlapped primitives and the extent of the bounding box of the cell.

The runtime is dominated by the sorting again (see Figure 5.2). We perform several additional steps compared to the uniform grid construction to build the second level of the grid. Nevertheless the introduced overhead is very small. As with the uniform grid construction we do not require any synchronization inside the kernels. Atomic synchronization is not necessary, but we use atomic increment on shared memory for convenience when writing pairs in the unsorted array.

The algorithm is able to exploit the high parallelism of modern GPUs because the work is evenly distributed among as many threads as the hardware is able to manage. This is possible because in each step threads are mapped to tasks independently. In addition to the high parallelism the runtime is independent of the primitive distribution in the scene exactly like the uniform grid construction.

In terms of memory footprint the two-level grids contain more primitive references than the uniform grids but have less cells since they adapt to the local density of primitives. The main memory bottleneck here is not the space required to store the cells but the memory occupied by the references.

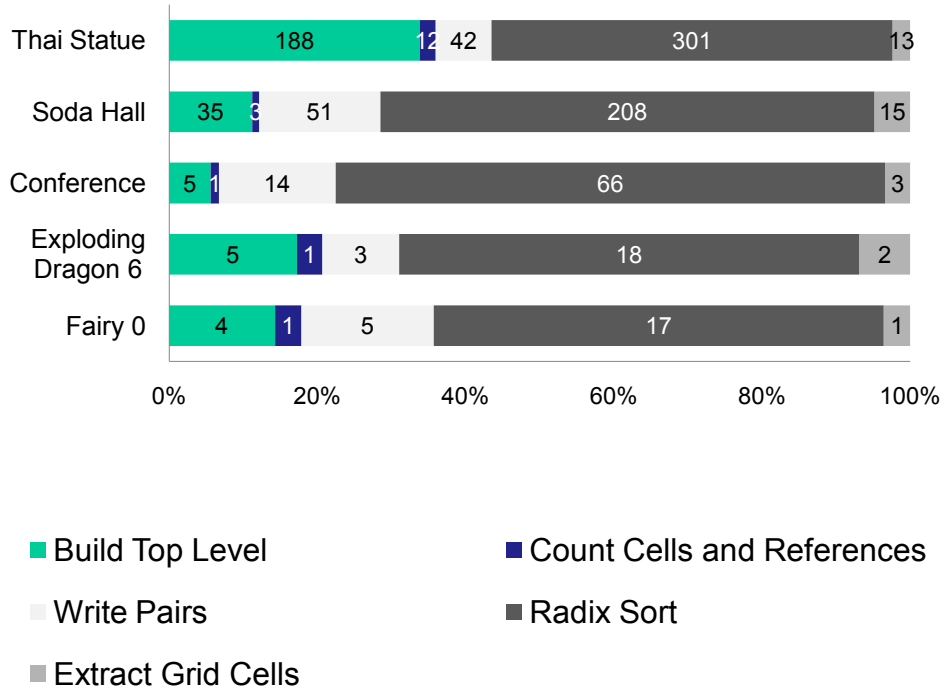

Figure 5.2: *Times for the different stages of the build algorithm in milliseconds. The complete runtime is the sum of the measured values. Because of memory constraints the resolutions of the top level cells for the Thai Statue were divided by 2 in each dimension. Fairy 0 and Exploding Dragon 6 means that the first and seventh frame from the animations were measured. We do not include the time to bind the top level cells to a texture since it was always less than 1 ms and thus accounted for by the rounding of the other values. Timings are from a GTX285.*

Still the two-level grids are more memory friendly than the uniform grids. Due to memory constraints we had to make the grid for the Thai Statue sparser (see Figure 5.2).

5.3.1 Triangle Insertion

When constructing two-level grids a lot more primitive references are generated for the second level cells. To insert a triangle in a cell we use two different tests. When counting the number of references we just compute the number of cells overlapped by the bounding box of the triangle. When writing the pairs we also test if the overlapped cell is intersected by the plane in which the triangle lies. If this more precise test fails we write a dummy pair and discard it in the sorting stage. This is identical in the uniform grid construction (see Section 4.4).

The number of dummy pairs varies much in the different test scenes. A worst case example is the conference where nearly half of the pairs to sort are dummies. Since this was not the case in most of the other test scenes we did not consider using the exact triangle insertion both when counting and when writing the pairs. Using sorting to discard triangle insertions makes the algorithm simpler and more elegant. Additionally the existence of very fast radix sort implementations makes it unclear whether or not different triangle insertion strategies will pay off. Another approach to reducing the workload for the sorting part of the algorithm is to use sparser grid resolution.

5.3.2 Grid Resolution

For our test we determined the grid resolution based on the heuristic 4.1 used for uniform grids. We divide the resolution by 6 in each dimension for the top level since this seemed to yield the best rendering times for most of the scenes we tried. Unless otherwise noted the resolution of each cell was again determined by the formula and left unchanged.

Further analysis of the grid resolution may turn out to be important for several reasons. First we are currently unable to use the hybrid intersector since there are hardly any cells with enough primitives (16 in our current implementation). We did not have enough time to make in-depth analysis of different strategies for choosing the resolution of the top level grid and each top level cell. The simple heuristic we used yielded very good results in most cases. However our tests indicate that having sparser subdivision of the top level cells also makes sense because it reduces both the construction time and the memory footprint without necessarily hurting rendering performance.

5.4 Results

We extended GPU ray tracer with an implementation of the two-level grid builder in CUDA. We tested performance on a machine with an NVIDIA Geforce 285 GTX[1] with 1 GB memory and a Core 2 Quad processor running at 2.66 GHz. As in the previous chapter, the only computationally demanding tasks for which we use the CPU are key frame interpolation for dynamic scenes and data transfers.

5.4.1 Construction

We give the performance of our implementation in Table 5.1. As with the uniform grids we include all overheads except the initial upload of the scene primitives to the GPU. The runtime is dominated by the sorting of the primitive-cell pairs on the second level of the grid. The number of pairs

[1] In the previous Chapter we used a 280 GTX.

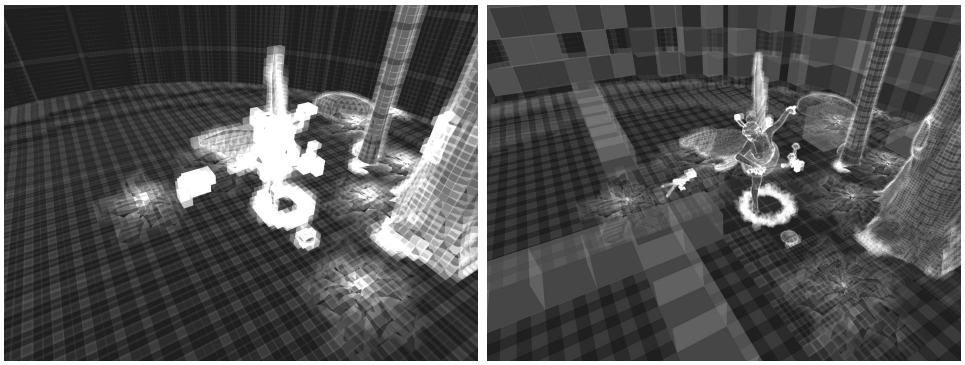

Figure 5.3: *The Fairy scene. Visualized are the amount of intersection tests per ray that are required with the uniform grid (left) and the two-level grid (right). The two-level grid adapts to the higher concentration of primitives and is able to eliminate more intersection candidates.*

Scene	Tris	References	Cells (top)	Cells (leaf)	Time
Thai Statue	10M	22.7M	226K	6M	556
Soda Hall	2.2M	15.6M	48K	11.7M	312
Venice	1.2M	6.5M	27K	6.5M	129
Conference	284K	4.9M	6K	1.5M	89
Exploding Dragon	252K	1.3M	5K	1.3M	29
Fairy Forest	174K	1.2M	3K	889K	28

Table 5.1: *Build statistics. Times are in milliseconds and are measured on a GTX285. The top level cells for the Thai Statue are with halved resolutions in each dimension. We used the seventh frame of the Exploding Dragon animation and the first frame of the Fairy Forest animation.*

(same as the number of primitive references) varies a lot in the different scenes. In the Conference the number of references is 5 million, which is very large compared to the uniform grid case where the references are 1 million. On the other hand the Fairy scene has almost the same amount (1.1 compared to 1.2 million) of references for both grids. With the exception of the Conference and perhaps the Soda Hall the amount of primitive references is reasonable compared to the uniform grid. The measured 6th frame from the Exploding Dragon animation is worst case example for the performance of the construction.

Note that the number of primitive references required for rendering is smaller than the total number of references. This is the case because we use the sorting stage to discard cell-triangle pairs that were generated using a more conservative triangle insertion test.

Model (Tris)	LBVH GTX280	H BVH GTX280	Grid GTX280	2-lvl Grid GTX285
Fairy (174K)	10.3ms 1.8 fps	124ms 11.6 fps	24ms 3.5 fps	28ms 9.9 fps
Conference (284K)	19ms 6.7 fps	105ms 22.9 fps	27ms 7.0 fps	89ms 11.8 fps
Exploding Dragon (252K)	17ms 7.3 fps	66ms 7.6 fps	13ms 7.7 fps	19ms 8.3 fps
Soda Hall (2.2M)	66ms 3.0 fps	445ms 20.7 fps	130ms 6.3 fps	143ms 12.6 fps

Table 5.2: *Build times and frame rate (excluding build time) for primary rays and simple shading for a* 1024×1024 *image. We compare performance of Günther's packet algorithm [GPSS07] with LBVH and Hybrid BVH (H BVH) as implemented by Lauterbach et al. [LGS+09] to our implementation. The grid for Soda Hall is not built with the default parameters like in Figure 5.2. Here we halved the resolution of each top level cell in each dimension.*

Model (Tris)	Ogre (50K)	Ben (78K)	Fairy (174K)	Expl. Dragon (252K)
Grid	20.5 fps	18.5 fps	7.5 fps	6.8 fps
Two-Level Grid	31 fps	31 fps	15 fps	16 fps

Table 5.3: *Frame rate (including build time) for primary rays and simple shading for a* 1024×1024 *image on a GTX285. This implementation includes persistent threads but not the hybrid intersector since the later did not influence performance. We halved the resolution of the each top level cell of the two-level grid.*

5.4.2 Rendering

We extend Table 4.3 from the previous section with the build and render times for the two-level grid in Table 5.2. Here as well as before, we did not include the persistent threads implementation (see Section 3.4.4) nor the hybrid intersector (see Section 3.3.2). In this comparison both the construction and the rendering performance is sub-optimal. Construction times can be reduced with about one third with a faster radix sort implementation (Billeter et al. [BOA09]).

To give the reader an impression of how much the performance is influenced by optimizations such as persistent threads we give the frame rates of our current implementation in Table 5.3. Note that these performance figures will change as soon as we improve the sorting implementation and find a better strategy to choose the resolution of the top level cells of the

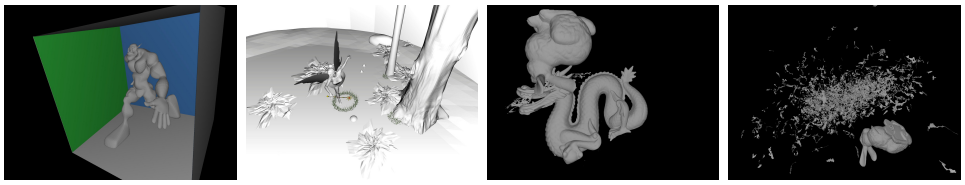

Figure 5.4: *Some of our test scenes, from left to right - Ogre, Fairy Forest and two frames from the Exploding Dragon animation. Using two-level grids we achieve performance of 31, 15, and 16 fps with simple shading in a* 1024×1024 *window on a GTX285. Frame rates include rebuild of the grid and all overheads except the time to display the rendered frame-buffer.*

grid. Additionally hardware support for population count and prefix sum on warp level will allow us to use the hybrid ray-triangle intersector with very little overhead. This will improve SIMD utilization and overall rendering performance. Here, as well as in all other measurements, we did not make any preprocessing of the scenes.

5.5 Conclusion

The acceleration structure discussed in this chapter is a simple extension of the uniform grid that copes better with non-uniform triangle distributions. Our goal was to both improve the quality of the acceleration structure and maintain the fast build times. The results show that the rendering performance increased and did not vary much when rendering parts of the scene with more complex geometry. Unfortunately in some scenes the build times of the two-level grids are significantly larger than those of uniform grids. This is caused by the large number of triangle references that have to be sorted when building the second level of the structure. The build times of the two-level grid can turn into a performance bottleneck in some scenes but this problem can be solved by incorporating a faster radix sort implementation (e.g. the one proposed by Billeter et al. [BOA09]).

We saw that two-level grids are suited for GPU ray tracing because because of their fast construction and very competitive rendering performance. They offer sufficient robustness toward primitive distribution and hardware-friendly traversal and construction. However our interest in this acceleration structure was also motivated by the fact that the two-level grid is a spatial subdivision unlike BVHs and unlike kd-trees it is a very shallow hierarchy. In the next chapter we exploit these properties in a scalable, lazy construction algorithm.

Chapter 6

Lazy Two-Level Grid Construction

Until now we discussed acceleration structures that can be built in linear time form scratch. This strategy is very general and can handle arbitrary motion regardless of its type. For example it does not matter if one wants to render a moving character or an explosion - since the structure is built from scratch for each frame the coherence or incoherence of the motion does not influence the performance. However there is a downside as well - it is easy to see that rebuilding the complete structure from scratch does not scale well with the numbers of primitives in the scene. In this chapter we describe a two-level grid construction algorithm that solves this problem by building the structure lazily.

6.1 Sampling Lazy Construction

The idea of lazy construction in terms of ray tracing is to build the acceleration structure on the fly. It exploits the fact that not all of the input primitives are relevant to the frame currently being rendered. So instead of constructing an expensive acceleration structure over the complete scene prior to rendering one builds the hierarchy while rendering. Only parts of the acceleration structure that are visited by rays are actually computed. This allows to avoid wasting build time for irrelevant parts of the scene. As recent work suggests [HMF07] the complexity of a lazy construction algorithm depends on the set of *visible* primitives instead of all primitives. Please note that the term visible is slightly misleading since an object can contribute to the image without being visible for example by casting shadow on a directly visible object. In the following we will refer as visible to primitives or objects that contribute to the final rendering.

While algorithmically (or theoretically) the lazy construction of hierarchical acceleration structures has no downsides, in practice this is not

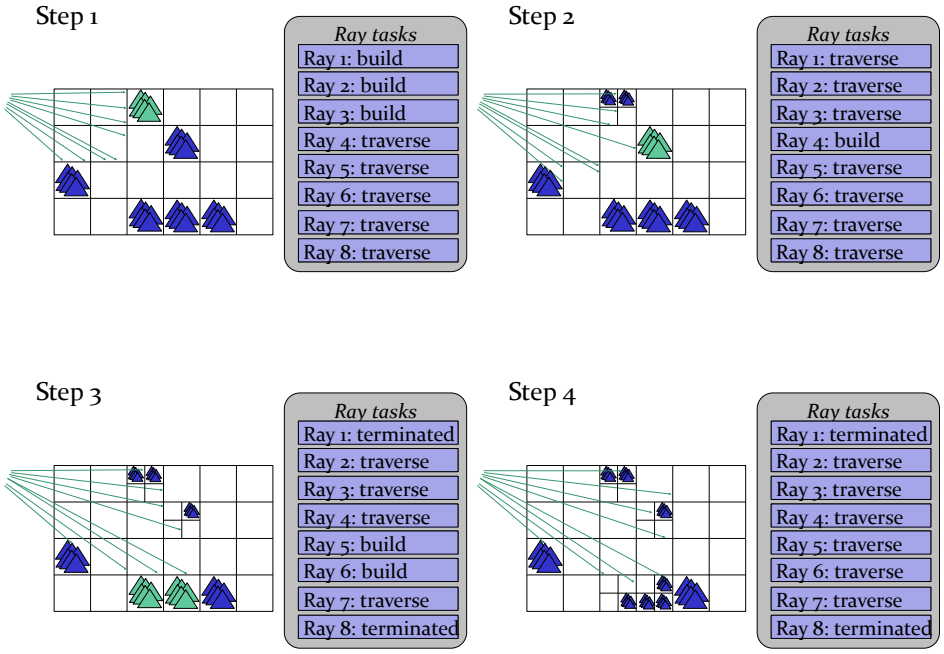

Figure 6.1: *Traditional lazy build algorithms traverse and construct simultaneously. This results in incoherent computations.*

the case. Approaches for lazy construction have been hard to implement efficiently on todays hardware. Simultaneous traversal and construction results in more complex code that is difficult to optimize and does not run efficiently on SIMD architectures such as todays CPUs and GPUs (Figure 6.1). However the scalability offered by the lazy build strategies is a big advantage, especially if one considers the tendency of the scenes used in various rendering applications to constantly grow larger. In the following we describe our attempt to map a lazy build algorithm for two-level grids to GPUs using CUDA.

Trying to simultaneously traverse and construct the structure results in complex and divergent code - a type of tasks that does not map well to modern GPUs. However we observe that the separation of the set of visible primitives from all primitives does not have to occur during traversal and can be done in a preprocessing step. We base our construction algorithm on this observation and introduce an additional step with the purpose to separate relevant from irrelevant parts of the scene (Figure 6.2).

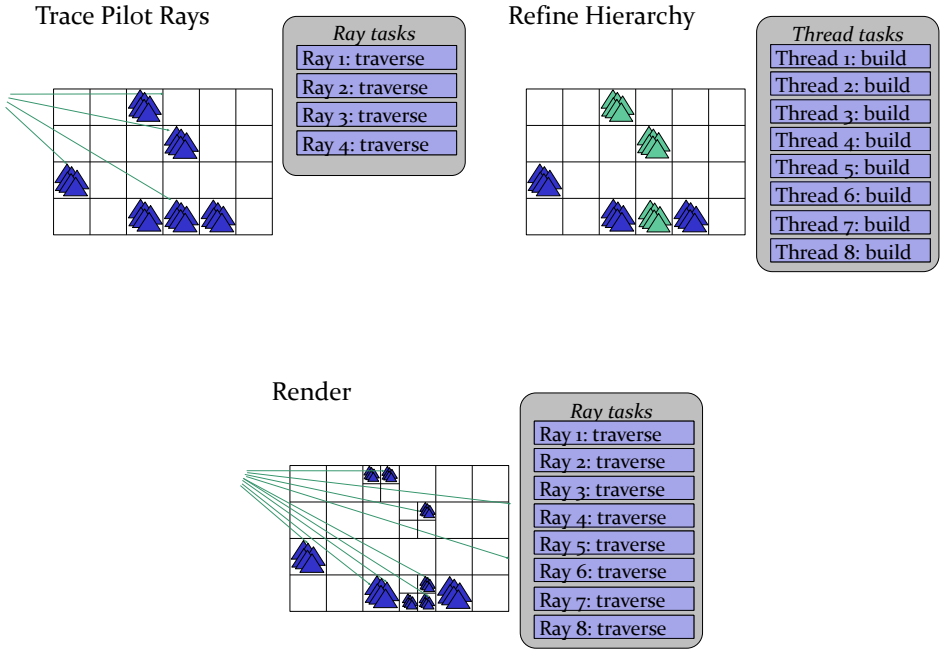

Figure 6.2: *Our approach separates out relevant parts of the scene prior to rendering. The algorithm breaks down to several simple parts that map better to GPUs. We build the coarser top level and trace a small amount of pilot rays (upper left). After we mark the visited top level cells and refine them (upper right) we trace the remaining rays.*

6.2 Algorithm

The construction algorithm (Algorithm 6) is essentially the same as the two-level grid construction (Algorithm 5, Chapter 5). The difference is that we build only part of the second level cells. To determine which of the first level cells are to be refined we trace a set of *pilot rays* - a small subset of all rays needed for rendering (Figure 6.2).

6.2.1 Building the Top Level

We assume that the top level of the grid is already present. In our test we rebuild the top level grid from scratch every frame. A more efficient solution is to update only parts where an object moved, but this information should be provided by the application invoking the renderer. Due to the lack of time we were not able to incorporate a scene graph in our application and we could not reuse parts of the acceleration structure between frames.

Algorithm 6 Lazy Two-Level Grid Construction. Kernel calls are suffixed by <<< >>>. Scan and sort may consist of several kernel calls.

$b \leftarrow$ COMPUTE BOUNDS()
$r \leftarrow$ COMPUTE TOP LEVEL RESOLUTION()
$t \leftarrow$ UPLOAD TRIANGLES()
$data \leftarrow$ BUILD UNIFORM GRID(t, b, r)
$A_{topPairs} \leftarrow$ GET SORTED TOP LEVEL PAIRS$(data)$
$n \leftarrow$ GET NUMBER OF TOP LEVEL REFERENCES$(data)$
$topCells \leftarrow$ GET TOP LEVEL CELL RANGES$(data)$
$A_{visible} \leftarrow$ TRACE PILOT RAYS <<< >>> $(data)$ ▷ **Mark visible cells**
$\mathrm{G} \leftarrow (r_y, r_z), \mathrm{B} \leftarrow r_x$
$A_{cellCounts} \leftarrow$ ARRAY OF $r_x * r_y * r_z + 1$ ZEROES
$A_{cellCounts} \leftarrow$ COUNT LEAF CELLS <<<G,B>>>$(b, r, topCells, A_{visible})$
$A_{cellCounts} \leftarrow$ EXCLUSIVE SCAN$(A_{cellCounts}, r_x * r_y * r_z + 1)$
▷ **Set leaf cell pointers and cell resolution:**
$topCells \leftarrow$ INITIALIZE <<<G,B>>> $(A_{cellCounts}, topCells, A_{visible})$
$\mathrm{G} \leftarrow 128, \mathrm{B} \leftarrow 256$
$A_{refCounts} \leftarrow$ ARRAY OF $\mathrm{G} + 1$ ZEROES
$A_{refCounts} \leftarrow$ COUNT LEAF REFS<<<G,B>>>$(t, b, r, n, A_{topPairs}, topCells)$
$A_{refCounts} \leftarrow$ EXCLUSIVE SCAN$(A_{refCounts}, \mathrm{G} + 1)$
$m \leftarrow A_{refCounts}[\mathrm{G}]$
$A_{pair} \leftarrow$ ALLOCATE LEAF PAIRS ARRAY(m)
$A_{pair} \leftarrow$ WRITE PAIRS<<<G,B>>>$(t, b, r, n, A_{topPairs}, topCells, A_{refCounts})$
$A_{pair} \leftarrow$ SORT(A_{pair})
$leafCells \leftarrow$ EXTRACT CELL RANGES<<< >>>(A_{pair}, m)

6.2.2 Tracing the Pilot Rays

The only difference to the standard two-level grid construction algorithm is that we construct the second level cells only for a subset of the top level cells. We determine which top level cells are visible by tracing pilot rays in the uniform grid consisting of the top level cells. During traversal we mark all top level cells that are visited by a pilot ray and need refinement. In the remaining part of the construction we only consider top-level cells that are marked as visible.

Tracing pilot rays in the coarse top level part of the structure is a performance critical part of the algorithm. This is the case because the temporary acceleration structure is very coarse and thus expensive to traverse. We could afford to trace around 1% of the total rays whithout introducing too much overhead in most scenes, but there were cases where tracing even 0.5% was too slow. We do not consider this to be a significant problem, since the budget of pilot rays grew bigger with the size of the scene and the performance was satisfactory with the larger Venice and Soda Hall scenes.

We generate the pilot rays by rendering uniformly distributed pixels in the image plane. This makes the rays very incoherent. Since the top level cells are large and contain many primitives we used the hybrid ray-triangle intersection algorithm described in Section 3.3.2. This alone doubled the amount of pilot rays we could trace for a given amount of time.

6.2.3 Building the Second Level

The remaining parts of the construction algorithm remain the same with the exception of some small modifications that allow to discard non-visible cells. The later are considered leaves or empty and not subdivided further. This reduces the amount of input data for the reference counting and the writing and sorting of pairs. The kernels we need to modify are the one that counts the cells and the one that initializes the resolution of the top level cells (Lines 11 and 14)[1].

6.3 Analysis

The advantage of the lazy construction algorithm over the conventional one is that the performance scales with the number of visible primitives instead of the total number of primitives. It is also important that the algorithm remains GPU friendly because we keep the computations coherent.

There are some downsides of the lazy construction as well. Some overhead is introduced because of the need to trace pilot rays on the coarser level of grid. In some cases this pays off by reducing the number of cells one has

[1] In our implementation we perform both the counting of cells and the initialization of the resolutions in the same kernel to avoid computing the resolutions twice.

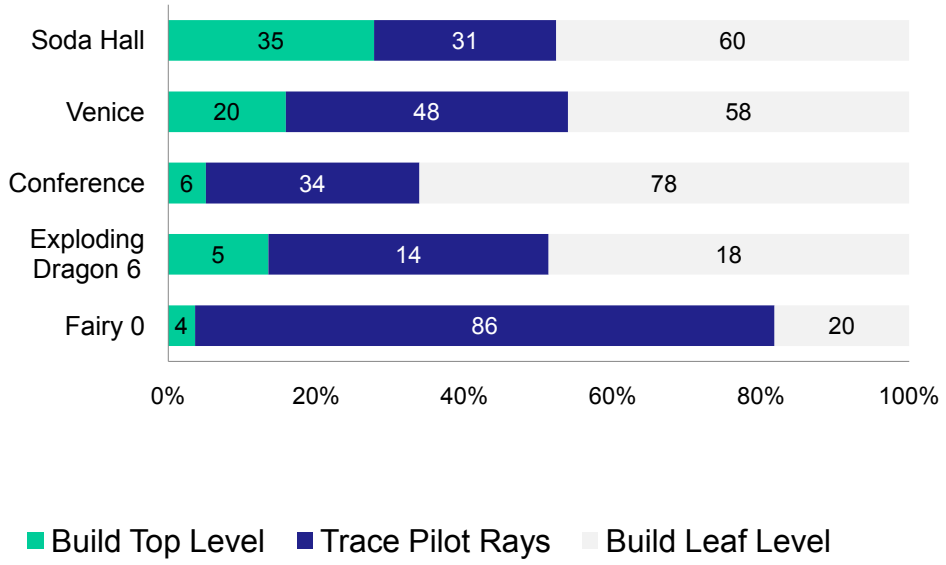

Figure 6.3: *Times for the different stages of the build algorithm in milliseconds. The pilot rays traced are 1% of the primary rays. Timings are from a GTX285.*

to consider for refinement, but if the entire scene is visible, full construction has to be performed anyway (see Figure 6.3). Another disadvantage is that we search for visible parts of the scene by sampling with a small subset of all rays. This means that it is possible to miss some of the visible top level cells. In this case the algorithm is correct but the structure will have lesser quality and the rendering performance will drop. We further discuss performance in the results section.

6.4 Results

From the results reported in Figure 6.3 and Table 6.1 one sees that the lazy construction does not make sense in scenarios where the amount of visible primitives is close to the total number of primitives. This is the case in the Conference, the Exploding Dragon and the Fairy Forest scene. There almost all top level cells are reached by pilot rays and have to be refined. As a result the lazy approach cannot reduce the workload for the sorting stage and the tracing of pilot rays is not compensated for.

Since constructing an acceleration structure for the smaller scenes like

Scene	Tris	Refs (std)	Refs (lazy)	Cells (top)	Leafs (std)	Leafs (lazy)	Time (std)	Time (lazy)
Soda Hall	2.2M	15.6M	3.5M	48K	11.7M	0.9M	312	109
Venice	1.2M	6.5M	3.5M	27K	6.5M	2.1M	129	126
Conference	284K	4.9M	4.5M	6K	1.5M	1.3M	89	118
Exploding Dragon	252K	1.3M	1M	5K	1.3M	1M	29	37
Fairy Forest	174K	1.2M	1M	3K	889K	795K	28	110

Table 6.1: *Build statistics. Times are in milliseconds and are measured on a GTX285. "Refs" denotes the number of primitive references, "Cells" are the top level cells, "Leafs" are the leaf cells. We used the seventh frame of the Exploding Dragon animation and the first frame of the Fairy Forest animation.*

Scene	View	Refs (std)	Refs (lazy)	Build (std)	Build (lazy)	Render (std)	Render (lazy)
Soda Hall	1	15.6M	3.5M	312ms	109ms	24 fps	24 fps
Soda Hall	2	15.6M	2.7M	312ms	120ms	19 fps	18 fps
Soda Hall	3	15.6M	2.7M	312ms	114ms	18 fps	18 fps
Venice	1	6.5M	3.3M	129ms	127ms	15 fps	12 fps
Venice	2	6.5M	2.0M	129ms	80ms	18 fps	14 fps

Table 6.2: *Lazy Build statistics. We compare build and render times for several view-ports (*1024×1024*) of the Soda Hall and Venice scenes. See Figure 6.4 for the views. We used 1% pilot rays for Soda Hall and 0.25% pilot rays for Venice. Tests were made on a GTX285.*

the Conference and the Fairy Forest is not a target for the lazy construction algorithm we concentrate on the larger Soda Hall and Venice. Although we could try to find setting for which the lazy construction works better by changing the grid resolution or shooting less pilot rays, constructing acceleration structure for such small scenes lazily makes little sense because conventional algorithms already perform well there. We are interested in the trade-off between build time and rendering performance in large scenes.

The measured results in Table 6.2 suggest that the lazy construction pays off for various view-ports. This is due to the fact that in both test scenes there is large amount of occluded geometry and most of the scene is not contributing to the final image. It is interesting to see that the performance does not deteriorate much even if the view-port covers a large portion of the scene (first views for both Soda Hall and Venice in Table 6.2). This shows that the algorithm can handle scenes in which the visible primitives are relatively large part of the total primitives (e.g. 50% for the fist view of the Venice scene).

In Figure 6.4 we analyze the quality of the acceleration structure. We

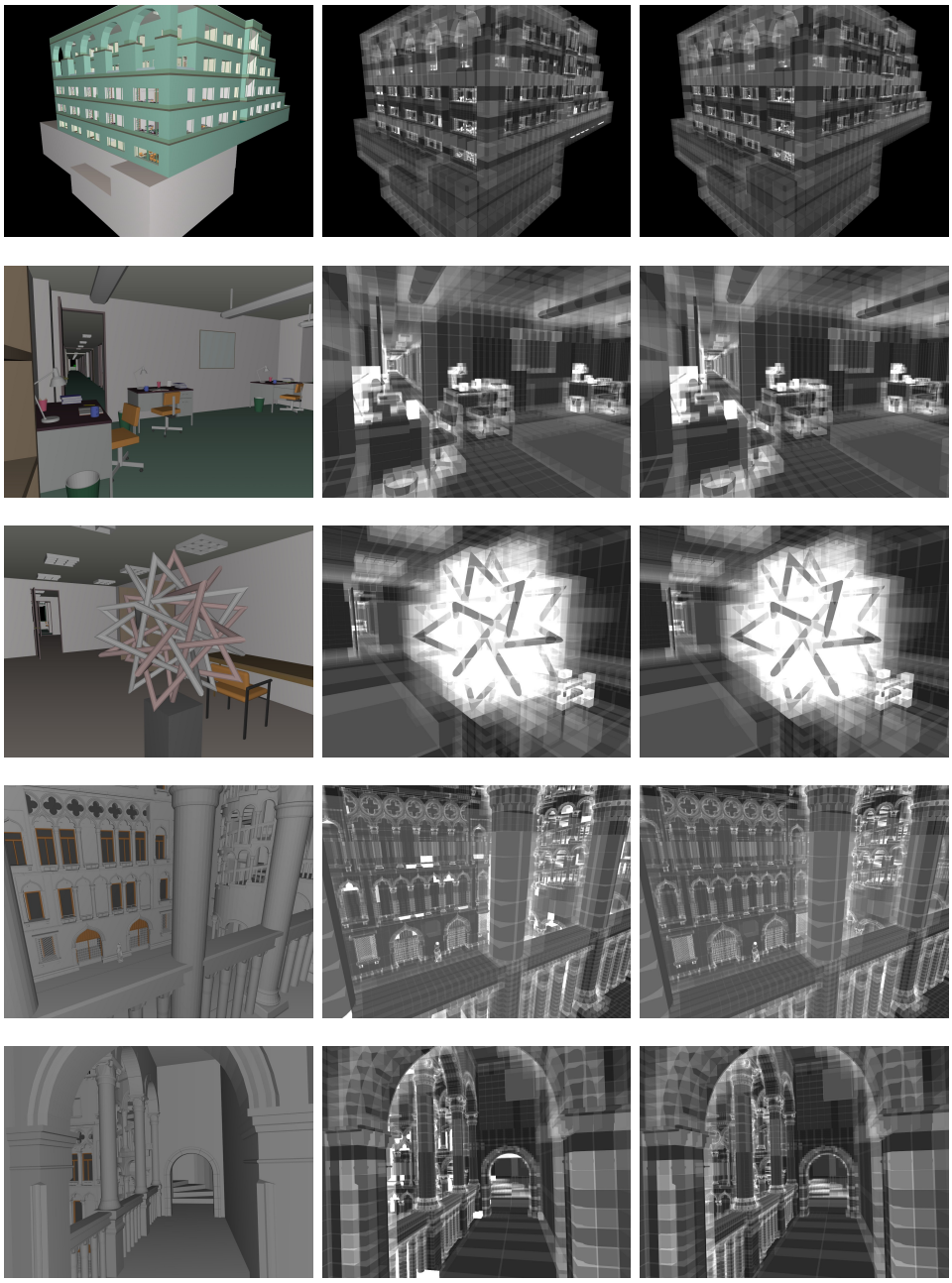

Figure 6.4: *The view points for Soda Hall (first three rows) and Venice (fourth and fifth row). Left column - image with simple shading, middle column - amount of intersection tests with the lazy construction, right column amount of intersection tests with the conventional construction.*

observe that even 1% of the rays are enough to "discover" the visible parts of the scene and there is no big difference in the amount of intersection tests that are required to render the image. This, together with the small differences in rendering performance (Table 6.2), shows that the acceleration structure and the lazy build algorithm can be used successfully in applications that render very large scenes.

6.5 Conclusion

In this chapter we showed how using sampling to steer the construction of the two-level grid structure can be done efficiently and improves the time for construction when rendering large scenes. We are able to efficiently discover most of the visible parts of the scene and concentrate the construction process to them, instead of building the complete acceleration structure at once.

The use of the lazy strategy also makes sense in a scenario in which the structure is constructed incrementally. This can be done for both static and dynamic scenes, but the later require an application with scene-graph support which we do not have currently. However there is no evidence that the lazy construction will fail in those cases.

We hope that as the size of the scenes grows the need and use of lazy construction algorithm will become larger. They offer better scalability than traditional construction algorithms, even if this is at the cost of some overhead. Our results show that there are practical examples where the scalability can compensate for this overhead.

Chapter 7

Conclusion

In this work we investigated various approaches for ray tracing static and dynamic scenes on GPUs. We focus on the use of not very popular acceleration structures such as the uniform grid and show that their construction and traversal map well to the parallel model exposed by CUDA.

Our contributions include a hybrid ray-triangle intersection algorithm that improves SIMD efficiency during traversal of incoherent rays. We also found out that a very efficient way to construct uniform grids on graphics cards is to reduce the build process to sorting primitive-cell pairs.

Since the uniform grids did not offer good acceleration in scenes with non-uniform triangle distributions we extended the data structure to the two-level grids. They are a hierarchical extension that adapts much better to the local primitive density in a scene. We propose a fast sort-based construction algorithm for the new structure. The two-level grids can be built almost as fast as the uniform grids and provide better ray traversal acceleration.

We also propose a lazy construction algorithm for two-level grids that maps well to GPUs. Our results show that using ray tracing for sampling can be used to efficiently extract the visible part of the geometric primitives in large scenes.

Altogether the proposed algorithms are interesting because they are not direct mapping of existing approaches that are known to work good on CPUs. The discussed algorithms are parallel and intended for the massively parallel architecture of modern GPUs. We could show that acceleration structures and algorithms that are believed to perform bad on sequential hardware can exploit the GPU architecture and in many cases provide performance better than state-of-the-art implementations of concurrent approaches.

Bibliography

[AL09] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 145–149, New York, NY, USA, 2009. ACM.

[AM01] Tomas Akenine-Möller. Fast 3d triangle-box overlap testing. *Journal of Graphics Tools*, 6:29–33, 2001.

[AW87] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*, pages 3–10. Elsevier Science Publishers, Amsterdam, North-Holland, 1987.

[BOA09] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient stream compaction on wide SIMD many-core architectures. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 159–166, New York, NY, USA, 2009. ACM.

[Dev88] Olivier Devillers. *Methodes dóptimisation du tracé de rayons.* PhD thesis, Université de Paris-Sud, June 1988.

[ED06] Elmar Eisemann and Xavier Décoret. Fast scene voxelization and applications. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 71–78, New York, NY, USA, 2006. ACM.

[GPSS07] Johannes Günther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Realtime ray tracing on GPU with BVH-based packet traversal. *Symposium on Interactive Ray Tracing*, pages 113–118, 2007.

[Gre08] Simon Green. Particles demo, 2008. NVIDIA CUDA SDK v2.2 http://www.nvidia.com/object/cuda_sdks.html.

[Hav01] Vlastimil Havran. *Heuristic Ray Shooting Algorithms.* PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[HMF07] Warren Hunt, William R. Mark, and Don Fussell. Fast and lazy build of acceleration structures from scene hierarchies. In *IEEE/EG Symposium on Interactive Ray Tracing 2007*, pages 47–54. IEEE/EG, Sep 2007.

[IDC09] Paulo Ivson, Leonardo Duarte, and Waldemar Celes. Gpu-accelerated uniform grid construction for ray tracing dynamic scenes. Master's Thesis Results 14/09, PUC-Rio, June 2009.

[ISP07] T. Ize, P. Shirley, and S. Parker. Grid creation strategies for efficient ray tracing. *Symposium on Interactive Ray Tracing*, pages 27–32, Sept. 2007.

[IWRP06] T. Ize, I. Wald, C. Robertson, and S.G. Parker. An evaluation of parallel grid construction for ray tracing dynamic scenes. *Symposium on Interactive Ray Tracing*, pages 47–55, 2006.

[Jen96] Henrik Wann Jensen. Global illumination using photon maps. *Rendering Techniques*, pages 21–30, 1996. (Proceedings of the 7th Eurographics Workshop on Rendering).

[JW89] David Jevans and Brian Wyvill. Adaptive voxel subdivision for ray tracing. In *Proceedings of Graphics Interface '89*, pages 164–72, Toronto, Ontario, June 1989. Canadian Information Processing Society.

[Kaj86] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, 1986.

[KS09] Javor Kalojanov and Philipp Slusallek. A parallel algorithm for construction of uniform grids. In *HPG '09: Proceedings of the 1st ACM conference on High Performance Graphics*, pages 23–28, New York, NY, USA, 2009. ACM.

[Laf96] Eric Lafortune. *Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering.* PhD thesis, Katholieke Universitiet Leuven, Belgium, 1996.

[LD08] Ares Lagae and Philip Dutré. Compact, fast and robust grids for ray tracing. *Computer Graphics Forum (Proceedings of the 19th Eurographics Symposium on Rendering)*, 27(4):1235–1244, June 2008.

[LGS+09] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast BVH construction on GPUs. In *Proceedings of Eurographics*, 2009.

[MB90] J. David MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Visual Computer*, 6(6):153–65, 1990.

[MT97] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skardon. Scalable parallel programming with CUDA. In *Queue 6*, pages 40–53. ACM Press, 2 2008.

[NVI09] NVIDIA. *NVIDIA CUDA Programming Guide 2.2.* NVIDIA, 2009.

[PGSS06] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Experiences with streaming construction of SAH KD -Trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 89–94, sep 2006.

[PGSS07] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3), September 2007.

[PN08] Suryakant Patidar and P.J. Narayanan. Ray casting deformable models on the GPU. *Sixth Indian Conference on Computer Vision, Graphics & Image Processing*, pages 481–488, 2008.

[SHZO07] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[SSK07] M. Shevtsov, A. Soupikov, and A. Kapustin. Ray-Triangle Intersection Algorithm for Modern CPU Architectures. In *Proceedings of GraphiCon 2007*, pages 33–39, 2007.

[Vea97] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation.* PhD thesis, Stanford University, 1997.

[Wal04] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination.* PhD thesis, Computer Graphics Group, Saarland University, 2004.

[WBS06] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. SCI Institute Technical Report UUSCI-2006-015, University of Utah, 2006. (conditionally accepted at ACM Transactions on Graphics, preprint available at http://www.sci.utah.edu/~wald/Publications/webgen/2006/BVH/download/togbvh.pdf).

[WH06] Ingo Wald and Vlastimil Havran. 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*, pages 61–69, sep 2006.

[WIK+06] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics*, 25(3):485–493, 2006.

[WK06] Carsten Wächter and Alexander Keller. Instant ray tracing: The bounding interval hierarchy. In *Rendering Techniques 2006, Proceedings of the Eurographics Symposium on Rendering*, 2006.

[WMS06] Sven Woop, Gerd Marmitt, and Philipp Slusallek. B-kd trees for hardware accelerated ray tracing of dynamic scenes. Technical report, Computer Graphics Group, Saarland University, 2006. (submitted for publication).

[ZHWG08] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, pages 1–11, New York, NY, USA, 2008. ACM.