Memory Aware Realtime Ray Tracing: The Bounding Plane Hierarchy

Ralf Karrenberg

Computer Graphics Group
Saarland University
Saarbrücken, Germany

Bachelor Thesis
Universität des Saarlandes
Naturwissenschaftlich-Technische Fakultät I
Fachrichtung Informatik
Bachelor-Studiengang Informatik

Betreuender Hochschullehrer / Supervisors:
Prof. Dr.-Ing. Philipp Slusallek, Universität des Saarlandes, Saarbrücken, Germany

Gutachter / Reviewers:
Prof. Dr.-Ing. Philipp Slusallek, Universität des Saarlandes, Saarbrücken, Germany
Dr. Karol Myszkowski, Max-Planck-Institut für Informatik, Saarbrücken, Germany

Dekan / Dean:
Prof. Dr.-Ing. Thorsten Herfet, Universität des Saarlandes, Saarbrücken, Germany

Eingereicht am / Thesis submitted:
August 31, 2007

Universität des Saarlandes
Fachrichtung 6.2 - Informatik
Im Stadtwald - Building E 1 1
66123 Saarbrücken

Erklärung / Declaration:
Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und alle verwendeten Quellen angegeben habe.
I hereby declare that the work presented in this bachelor thesis is entirely my own and that I did not use any sources or auxiliary means other than those referenced.

Saarbrücken, August 31, 2007

Einverständniserklärung / Agreement:
Hiermit erkläre ich mich damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek der Fachrichtung Informatik aufgenommen wird.
I hereby agree on including my work into the inventory of the computer science library.

Saarbrücken, August 31, 2007

Abstract

Realtime Ray Tracing has become available on single desktop environments over the last few years, recently moving on to supporting dynamic scenes. Approaches either use general-purpose CPUs, high-end programmable graphics cards or specialised custom hardware. One of the most important factors influencing the performance of all implementations is the algorithm's dependence on a spatial index structure. Its random memory access is very slow compared to the computational power of current computer hardware, often being the bottleneck of the system.

The goal of this thesis was the design and implementation of a spatial index structure that focuses entirely on improving memory efficiency in trade for computational complexity. Two main starting points were followed: The memory requirement of the whole structure and of each of its parts on one hand and the caching efficiency on the other. The resulting contributions are the following:

- The **Bounding Plane Hierarchy** (BPH) is a complete acceleration structure equivalent to a Bounding Volume Hierarchy (BVH) with axis-aligned bounding boxes (AABBs) that needs less than half the size of current BVH-implementations.

- **Treelets** are groups of interconnected nodes that are traversed independently from the rest of the acceleration structure by using a new two-stage algorithm. They are employed to enhance cache efficiency - especially on parallel and/or multi-threaded systems like the CELL - and can be applied to any spatial index structure.

A first implementation of the BPH with Treelets using Packet Tracing and SIMD-instructions renders animations in real-time while still offering many possibilities for optimization.

Kurzfassung

Echtzeit Ray Tracing ist in den vergangenen Jahren auf handelsüblichen Desktop-PCs möglich geworden - in letzter Zeit auch mit Unterstützung animierter Szenen. Unterschiedliche Ansätze nutzen dafür die Rechenkraft von CPUs, high-end Grafikkarten oder speziell entwickelter Hardware. Einer der wichtigsten Faktoren, der die Performance aller dieser Systeme beeinflusst, ist die Abhängigkeit des Algorithmus von einer räumlichen Indexstruktur. Deren zufälliger Speicherzugriff ist sehr langsam im Vergleich zur Rechenleistung aktueller Prozessoren und stellt daher häufig den Flaschenhals einer Implementierung dar.

Das Ziel dieser Arbeit war die Entwicklung und Implementierung einer Beschleunigungsstruktur, die vollständig auf die Optimierung der Speichereffizienz auf Kosten erhöhter Rechenlast zugeschnitten ist. Dabei wurden zwei Ansatzpunkte verfolgt: Der Speicherbedarf der gesamten Struktur sowie jedes ihrer Teile auf der einen, die Cache-Effizienz auf der anderen Seite. Folgende Beiträge wurden erarbeitet:

- Die **Bounding Plane Hierarchy** (BPH) ist eine zu einer Bounding Volume Hierarchie (BVH) mit "Achsen-Ausgerichteten Begrenzungs-Boxen" (AABBs) äquivalente Indexstruktur, die jedoch weniger als halb so viel Speicherbedarf wie aktuelle BVH-Implementierungen hat.
- **Treelets** sind Gruppen von verbundenen Knoten, die mit einem neuartigen zweistufigen Algorithmus unabhängig vom Rest der Beschleunigungsstruktur traversiert werden. Sie werden zur Verbesserung der Cache-Effizienz benutzt - speziell auf Systemem wie dem CELL, die parallel und/oder mit vielen Threads arbeiten - und können mit jeder Indexstruktur kombiniert werden.

Eine erste Implementierung der BPH einschließlich neuartiger Treelet-Traversierung unter Verwendung von Packet Tracing und SIMD-Instruktionen rendert animierte Szenen bereits in Echtzeit, während noch viele Möglichkeiten zur Optimierung bestehen.

Contents

Chapter 1

Introduction

Realtime Ray Tracing has become available on single desktop environments over the last few years, recently moving on to supporting dynamic scenes. Approaches either use general-purpose CPUs, high-end programmable graphics cards or specialised custom hardware. One of the most important factors influencing the performance of all implementations is the algorithm's dependence on a spatial index structure. Its random memory access is very slow compared to the computational power of current computer hardware, often being the bottleneck of the system.

It is a well-known issue that, in the last decades, development of CPUs has shown constantly fast growing clock frequencies while bandwidth to memory increases much slower due to the high manufacturing cost of memory with low latency. Hardware manufacturers therefore deal with this widening "memory gap" by using hierarchical cache-structures that buffer data in subsequent levels of memory. These levels decrease in latency and size the nearer to the CPU they are located. Thus, it is crucial for efficient software that relies on RAM- or even disk-access to accommodate to the underlying hardware in order to minimize worst-case access times.

The goal of this thesis was the design and implementation of a spatial index structure that focuses entirely on improving memory efficiency in trade for computational complexity. Two main starting points were followed: The memory requirement of the whole structure and of each of its parts on one hand and the caching efficiency on the other. The resulting contributions are the following:

The Bounding Plane Hierarchy (BPH)

The BPH is a complete acceleration structure equivalent to a Bounding Volume Hierarchy (BVH) with axis-aligned bounding boxes (AABBs) that needs less than half the size of current BVH-implementations.

The basic idea is that not all of the planes that define the AABBs of each node's children have to be stored explicitly - at least 6 out of 12 planes can be discarded and reconstructed during traversal. Intersection can efficiently be performed for both children at once, theoretically increasing intersection-speed by a factor of two.

In order to further reduce memory demand of the BPH and increase caching efficiency, inner nodes are compressed to 32 bytes and so-called "intermediate nodes" that optimize a common case of tree-topology are introduced.

Treelets

Treelets are groups of interconnected nodes that are traversed independently from the rest of the acceleration structure. They are employed to enhance cache efficiency by using the knowledge that a certain ray traversing a particular node will probably traverse neighbouring paths as well.

The new traversal scheme is split in two different stages: a global stage works between Treelets and a local one inside. If a ray enters a Treelet, it will only leave it again after it has completely traversed it. Thus, all internal nodes can be preloaded in one burst and made available in cache.

This technique is very likely to considerably increase efficiency of memory access by guaranteeing cache-hits inside the Treelets, especially for parallel and/or multi-threaded systems like the CELL. Another advantage is that Treelets can be applied to any spatial index structure.

In order to let the index structure be as flexible as possible, no assumption about the topology of the scene is made or used to simplify algorithms (e.g. forcing balanced trees). The BPH can be applied to any binary Bounding Volume Hierarchy using AABBs without big effort, Treelets can also be used with any other index structure.

A first implementation of the BPH with Treelets using Packet Tracing and SIMD-instructions renders animations in real-time while still offering many possibilities for optimization.

Outline

The thesis starts with a basic introduction of the Ray Tracing algorithm, its advantages and challenges and an overview of the related previous work on Interactive Ray Tracing, spatial index structures and memory efficiency. The main part introduces the BPH and Treelets and is accompanied by a detailed analysis of memory requirement, traversal statistics and rendering performance. A discussion of drawbacks of the current implementation leads to the conclusion. The thesis is finished by some ideas for future work, including approaches to solve most of the present problems.

Chapter 2

Previous Work

This chapter gives an introduction of the Ray Tracing algorithm with its advantages and challenges and an overview of related work on spatial index structures, dynamic scenes and memory efficiency.

2.1 Ray Tracing

Current graphics processing is usually done by so-called Graphics Processing Units (GPUs) that use Rasterization to display graphics. In this approach, the algorithm tests for each object where parts of it are visible on the screen. The triangles of a scene are fed to the GPU sequentially, passing several steps of a rendering pipeline where all factors are determined that influence the appearance of each individual triangle, including e.g. transformation, texturing, clipping and transparency. These GPUs nowadays are highly optimized, being able to work on many triangles in parallel and supporting a large variety of effects directly in hardware. But the individual handling of each triangle is also Rasterization's biggest drawback: Global effects like reflection, refraction or indirect illumination need information about the environment which is not available directly. Thus, no accurate simulation of physics is possible - in order to obtain the desired visual properties, the effects usually have to be "faked".

Ray Tracing on the other hand allows for a physically correct simulation by using rays shot into the scene to determine all objects that are visible at each pixel of the display. By definition it has knowledge about the complete scene, naturally supporting global effects. A simplified illustration of the algorithm is given in Figure 2.1.

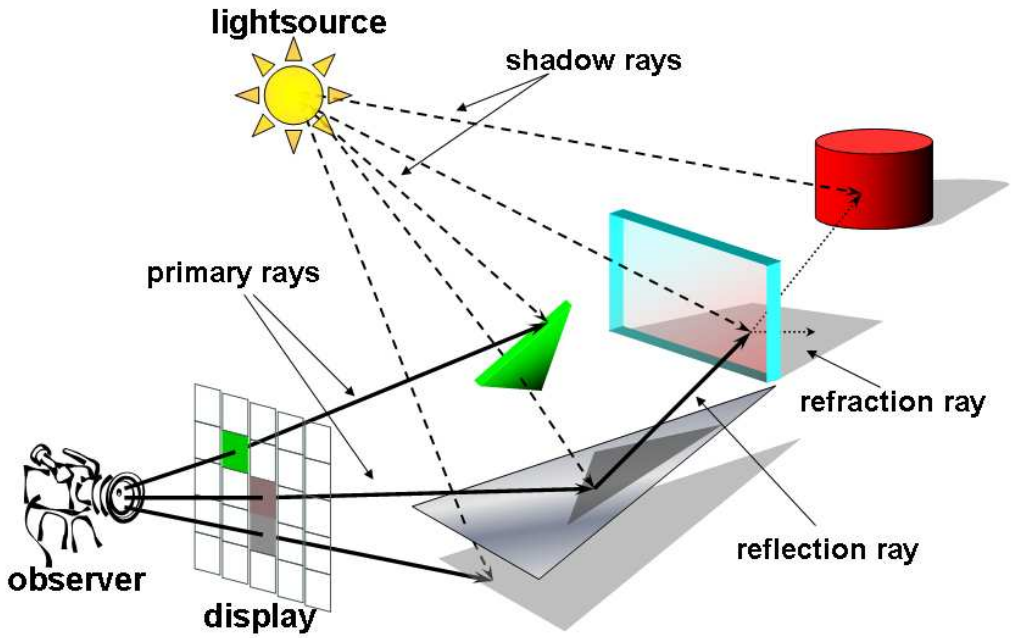

Figure 2.1: Simplified illustration of the recursive Ray Tracing algorithm.

2.1.1 Basics

In order to simulate physically correct lighting, rays would be sent from each light source into the scene, bouncing off of walls, being reflected or refracted, indirectly illuminating other objects etc. Finally, those that fall into the camera this way would be captured and displayed. The very low percentage of those rays would result in great computational overhead, so the basic Ray Tracing algorithm makes use of the fact that physical optics is symmetric: rays of light are "back traced" from the viewpoint into the scene.

During rendering, the algorithm has to test each of these rays for its nearest intersection with any of the primitives the scene consists of. For large scenes, possibly consisting of millions of primitives, the complexity of this computation quickly explodes. In order to avoid intersecting each ray with all primitives, a spatial index structure is employed. It is built in a preprocessing step before rendering starts, recursively subdividing either the scene's space or its objects. Although this greatly improves performance, this acceleration structure is still one of the main bottlenecks of the algorithm which is explained in more detail in the following Sections 2.1.2 and 2.2.

Without any oversampling or other additional techniques, the basic "Ray Casting"-algorithm shoots one ray for each pixel of the currently generated frame. The ray is traversed through the acceleration structure until it reaches a leaf node, where it is tested for intersection points with the contained primitives. If it hits any of these, the corresponding shader (a program that is responsible for determining the appearance of the object, e.g. depending on its material properties) is executed and delivers the appropriate colour, which is then displayed at the pixel the ray was shot from. For this purpose, so-called "Ray Tracing" allows the shader to recursively shoot additional rays, e.g. for reflection and refraction when a transparent object is hit or shadow rays to the light sources to determine if an object lies in shadow.

This approach makes it possible to render complex global effects like the above mentioned reflection, accurate shadows or even global illumination. On the computational part, Ray Casting is very interesting for massively complex models due to its logarithmic complexity in the number of scene objects. Finally, the coherence of adjacent rays can be exploited by highly optimized parallel algorithms (e.g. using SIMD instructions that work on up to four rays in parallel).

2.1.2 Problems

The Ray Tracing algorithm poses three main challenges:

- the expense of tracing millions of rays per second,
- the expense of finding the nearest intersection of a ray with all primitives of the scene,
- the dependency on precomputation of a spatial index structure.

The first point can be illustrated by a very simple calculation: A scene is to be rendered with a standard screen resolution of 1024*768 pixels, an average of 10 rpp (rays per pixel, including secondary rays for shadow-computation, reflection, refraction etc.) and 60 frames per second in order to deliver a smoothly running interactive scene. The resulting

$$1024 * 768 * 10 * 60 \approx 471M\, rays/sec,$$

are by far exceeding computational capacities of usual desktop processors or even larger grids of workstations for non-trivial scenes. This problem has

been addressed by a variety of algorithmic approaches that are able to cope with the issue, e.g. by exploiting the coherence of adjacent rays using so-called "Packet Tracing" where rays are grouped and computation is done for the complete packet in parallel, using the coherence between adjacent rays (see Section 3.5.1).

The second fundamental problem is introduced by testing each ray against millions of primitives to find out its nearest intersection point. This requires some organization of the scene's objects to reduce the computational complexity - an index structure is required that recursively subdivides the scene and allows to narrow down the number of primitives that have to be intersected with each ray to only a few. Section 2.2 introduces the most commonly used index structures, including the Bounding Volume Hierarchy (BVH) that serves as a fundament for this thesis.

Intersection becomes significantly faster using such subdivision schemes, but this is bought by typically 50-100 traversal steps through the acceleration structure (depending on the size of the scene) that each ray has to perform until an intersection is found, which is still very costly.

The last major issue is the building-time of index structures which has a lower bound of $O(n\ log\ n)$ [WH06, HHS06]. At least for larger scenes this is too costly to perform for each frame - a pre-processing step is required. This imposes problems to all kinds of dynamic scenes where objects move, sooner or later enforcing a recomputation of the index structure to adapt the subdivision to the changed topology.

2.1.3 Interactive Ray Tracing

Interactive Ray Tracing has become possible over the last few years with the first system running on clusters of PCs [WBWS01] and an implementation for single desktop PCs [Wal04] that was used for the OpenRT interface.

There have also been various implementations on different hardware platforms like GPUs [Pur04, PGSS07], the Cell [MFT05, BWSF] or custom hardware [SWS02, WSS05, Woo06].

2.2 Spatial Index Structures

Over the last years, so-called "kd-trees" [Ben75] were the most widely used spatial index for static scenes as they provided the best frame rates. A kd-

tree subdivides space (using axis-aligned planes) into a binary tree-structure where each leaf node holds only a few primitives. Another very popular data structure is the "Bounding Volume Hierarchy" (BVH) [Cla76, RW80] that recursively subdivides objects into groups that are contained in a Bounding Volume - usually an Axis-Aligned Bounding Box (AABB) due to its simplicity, but there are also approaches using e.g. oriented bounding boxes or spheres.

Subdivision can be done in various ways, e.g. by simply recursively splitting the nodes in the middle in one dimension or with more sophisticated heuristics like the "Surface Area Heuristic" (SAH) that tries to optimize the splitting plane by using the probability if particular resulting subtrees will be hit by a random ray (also see Section 6.4).

Although kd-trees were the acceleration structure of choice for a long time, they are limited to static scenes, not supporting dynamic behaviour due to the high computational cost of rebuilding the complete kd-tree whenever primitives of the scene change position. Previous research tried to approach this issue by isolating few dynamic objects and traversing them separately [PMS+99] or by using the coherence between successive frames [AH95]. Only recently a lot of successful improvements were developed that solved the problem at least partly:

The custom hardware of [WSS05] used a top-level kd-tree that referenced object-kd-trees with different transformations and could be rebuilt quickly in the driver, allowing a limited number of objects with rigid-body transformations. Much more convenient approaches use Bounding Volume Hierarchies [LAM03, WBS07, LYTM06] or hybrids like the B-KD-tree [WMS06] to employ fast updating instead of rebuilds as a basis for efficient handling of coherently deforming and moving objects like humans or plants.

An efficient implementation of a kd-tree only uses 8 bytes per node, a factor of 4 less than a common BVH using 32 bytes. On the other hand, due to the subdivision of space, a kd-tree usually needs about 10-12 times the number of inner nodes than the object-dividing algorithms if building all trees down to one primitive per leaf without further optimizations. This leads to an overall memory requirement of approximately 96 bytes per primitive for a kd-tree whereas a standard BVH needs 64 bytes per primitive. The BKD-tree even only needs 16 bytes per node and a total of 32 bytes per primitive, the Bounding Interval Hierarchy (BIH), another kd-tree/BVH-hybrid, has a memory requirement of 12 bytes per node and approximately 16 bytes per primitive [WK06]. This thesis's contribution - the Bounding Plane Hierarchy

(BPH) - requires 32 bytes/node and approximately 16 bytes per primitive, if using two primitives in each leaf node it even comes down to 12 bytes per primitive (see Section 5).

Although the kd-tree is by far not the smallest overall structure, its low memory requirement for each node can result in much higher cache-efficiency because more nodes fit in one cache-line. However, a kd-tree's working set of nodes that are traversed during rendering can be much larger because primitives are allowed to reside in several branches of the tree.

The starting point of this thesis was chosen to be a usual Bounding Volume Hierarchy which has proven to be a good choice for dynamic scenes as it allows for quick updates or even complete rebuilds during rendering [LYTM06]. However, a BVH needs much more information (and thus far more memory) for each node than a kd-tree, as it needs to maintain a complete box in 3D consisting of 6 planes instead of only one splitting-plane for each child.

2.3 Memory Efficiency

In the last decades, development of CPUs has shown constantly fast growing peak performance while bandwidth to memory increases much slower due to the high manufacturing cost of memory with low latency. Growth estimations predict CPU speed to increase by approximately 80% per year whereas the speed of memory devices only grows at less than 10% per year [HP03, Bas91]. Hardware manufacturers therefore deal with this widening "memory gap" by using hierarchical cache-structures that buffer data in subsequent levels of memory. These levels decrease in latency and size the nearer to the CPU they are located. Thus, it is crucial for an efficient software that relies on RAM- or even disk-access to accommodate to the underlying hardware in order to minimize worst-case access times.

Research in this field has produced a variety of algorithms that work either "cache-aware" or "cache-oblivious" [FLPR99]. An implementation that is cache-aware uses the knowledge on what hardware it will run to adjust memory access to the exact size of the cache-blocks whereas cache-oblivious algorithms try to obtain maximal efficiency on any underlying hardware they may run on without using any assumptions, minimizing cache-misses for all cases.

The Ray Tracing algorithm's possibly biggest issue is directly related to this. For an efficient solution of the question what object a given ray

intersects with, it heavily depends on the performance of its spatial index structure, introducing two problems:

- The access to this structure is mostly random which results in expensive memory access when not using any sophisticated memory-layout.
- The structure somehow has to be stored and maintained during the rendering process and can easily reach a size of several gigabytes for large models.

Previous research addressing the first issue of organizing the index structure according to cache-access-patterns has been done by [YLPM05, YM06] and already led to significant increases in efficiency. The heuristics they developed could be a way to optimize construction of the Treelets presented in this thesis (see Section 8.2).

Lots of work has been done in a field concerned with memory requirements that applies to the second issue: dealing with extremely massive and detailed scenes like a power plant or a Boeing 777 consuming gigabytes of space pose specific challenges on distributing memory [BSGM02, DWS04, DWS05].

Chapter 3

The Bounding Plane Hierarchy (BPH)

The basis of the Bounding Plane Hierarchy (BPH) is formed by a standard Bounding Volume Hierarchy (BVH) that recursively subdivides the primitives of the scene and stores a complete axis-aligned bounding box (AABB) that encapsulates all underlying geometry of each node.

The following sections introduce the main ideas behind the BPH:

- Each node of the BPH only stores those parts of its children's AABBs that are required for a reconstruction during traversal. Only those six planes have to be stored that are different from the parent's AABB, effectively reducing the size of each node by a factor of 2.

- Each node of the BPH stores information about its two children. Due to the fact that the last inner nodes above leaf level already store the boxes of the leafs, these nodes directly point to the primitives and no additional leaf nodes have to be constructed.

- Both children of an inner node can be intersected at once using a modified version of the classic slabs algorithm [KK86], which can ideally speed up intersection by a factor of two.

- Implicit leaf nodes are split in two groups: additional to the standard pointer that references one primitive, a so-called "intermediate node" is introduced that points to two subsequent primitives. These intermediate nodes replace an inner node whenever it would have two leaf nodes as children, saving large amounts of memory.

- Inner nodes are compressed, "hiding" all information needed for traversal inside the child-pointers. This way they fit exactly into 32 bytes for better cache-alignment while the range of the pointers is slightly reduced to 27 bit.

- The algorithm can efficiently be implemented using Packet Tracing with SIMD-instructions.

3.1 Planes

A standard BVH implementation has to store two complete AABBs (consisting of 12 floating point values) for each inner node with two children to provide complete information for a correct traversal. In order to reduce the size of each inner node, the BPH only stores the 6 planes that are necessary to guarantee a correct reconstruction of these boxes.

The definition of a Bounding Volume Hierarchy implies that boxes fit as tight to their underlying geometry as their shape allows. Thus, the bounds of an inner node are defined by the union of their children's bounds. Reversely formulated this means that each child node shares some of the planes of its parent - redundant information that can be spared.

The maximum amount of planes is determined by the dimensionality: For each dimension, 4 planes are needed to enclose the minimum and maximum extent of both children. Two of these - the "outer" bounds - are by definition shared with the parent node and can be reconstructed, the other two - the "inner" bounds - are necessary and have to be stored. Hence, for each dimension, two planes are mandatory that define the children's extents inside the parent. For one dimension, both of these values can belong to the same child, defining its minimum and maximum extent. If both planes belong to different children, they have to be properly stored in a way that the min-plane defines the minimal extent of the child whose maximum extent is given by the parent and vice-versa. For correct mapping of the planes during traversal, only one bit of information for each plane is needed that indicates if a plane belongs to the left or right child of the current box.

AABBs in 2D therefore only need 4 planes in addition to the parent's box, in 3D two additional planes for the third dimension (6 in total) are required. Graphical illustrations are given in Figures 3.1 (2D) and 3.2 (3D).

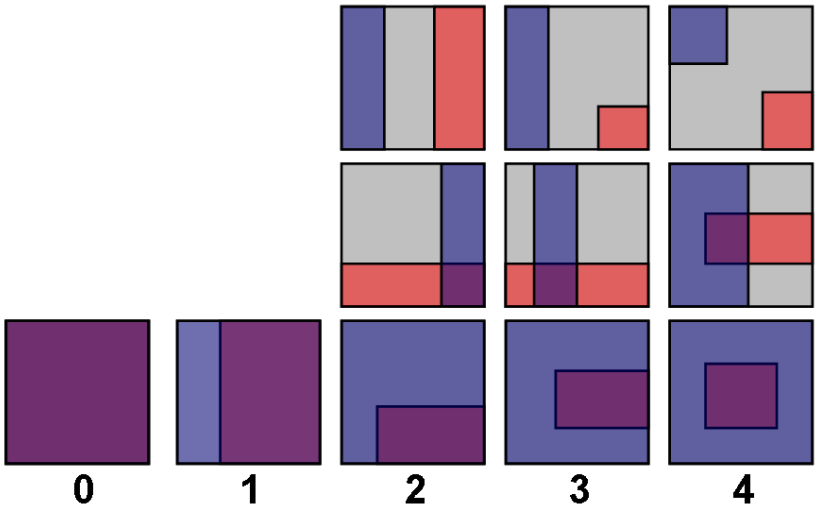

Figure 3.1: Common cases for 2D: the parent's bounds plus at most 4 additional planes define both boxes of the children. From left to right, the number of planes that have to be stored increases as denoted. The parent-node is coloured grey, the child-nodes are blue and red. Note that these are not all possible cases.

3.2 Leaf & Intermediate Nodes

On the bottom level of the hierarchy, inner nodes directly point into the array of primitives - no leaf nodes are constructed explicitly. It is possible to store more than one primitive in each leaf by allocating them sequentially in memory. They are accessed by adding offsets to the memory location of the first primitive.

Generally it would be possible to allow an arbitrary number of primitives per leaf, but that would require additional memory for each node in order to store the exact amount. This would destroy the memory layout shown in Section 3.3, while globally fixing the number of primitives per leaf node also introduces the possibility to apply an additional optimization of a very common case:

If an inner node has two leaf nodes as children, it is replaced by a so-called "intermediate node". This means that the parent of the current inner node sets a flag indicating that this pointer references two lists of primitives that are stored sequentially instead of one inner node (see Figure 3.3). Intersection can be performed either by intersecting both lists directly or by first

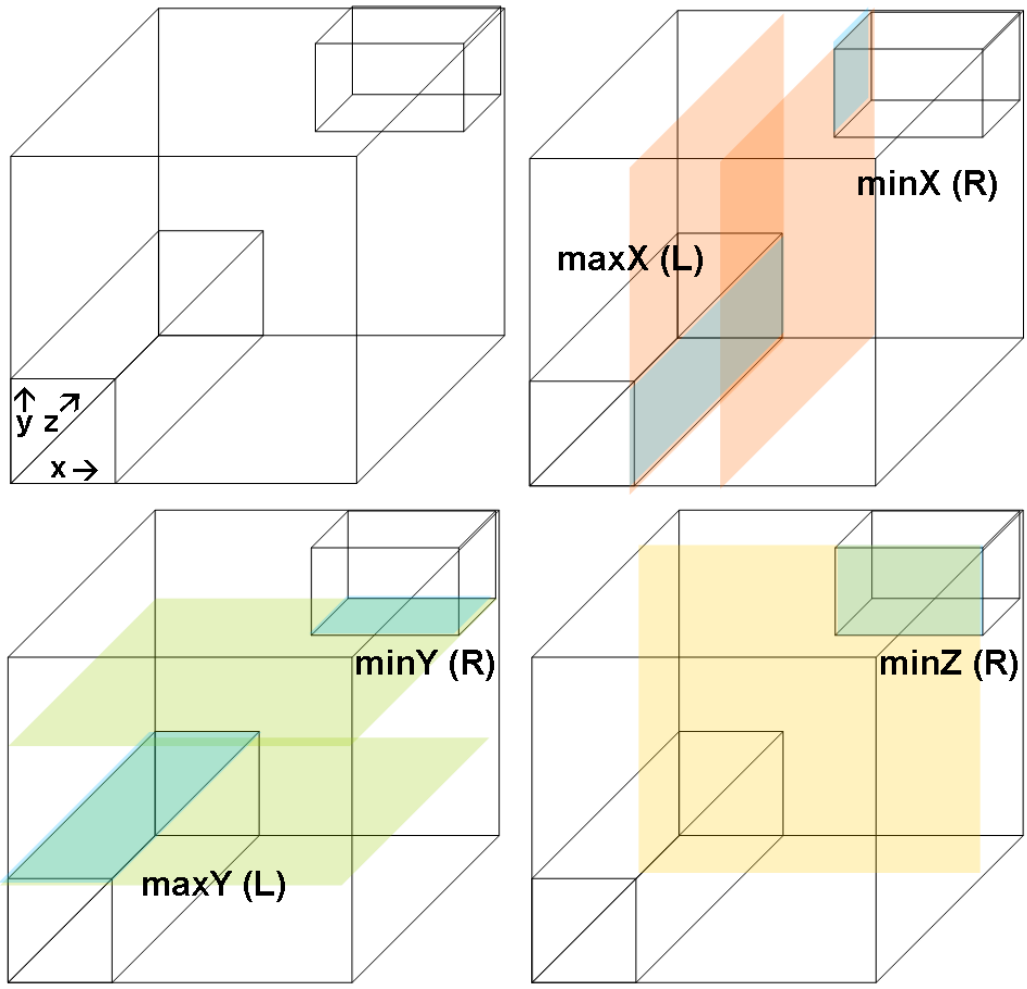

Figure 3.2: 3D Example: the parent's bounds plus 5 planes define both boxes of the children. minX, maxX etc. denote how each plane's position is identified, L and R show what bits are set to map each plane to the correct child. In this case, less than 6 planes suffice as maxZ does not have to be used.

recalculating their bounding boxes and intersecting these before continuing with the primitives.

In such a case, the memory of those two bounding boxes is saved in trade for the computational cost of either recalculating them or accepting more ray-primitive-intersections, following the main key note of this thesis.

This case is very common as can be observed in Table 5.2, but can't be applied under all circumstances: Inner nodes with two different children (one inner and one leaf node) are possible if no restrictions to the topology of the tree are made - enforcing a layout where an inner node can only have two inner nodes or two leaf nodes as children would need a specific building-algorithm that would not adapt to the scene as good as it is possible without restrictions. Figure 3.4 shows an example for the substitution applied to an arbitrary structure.

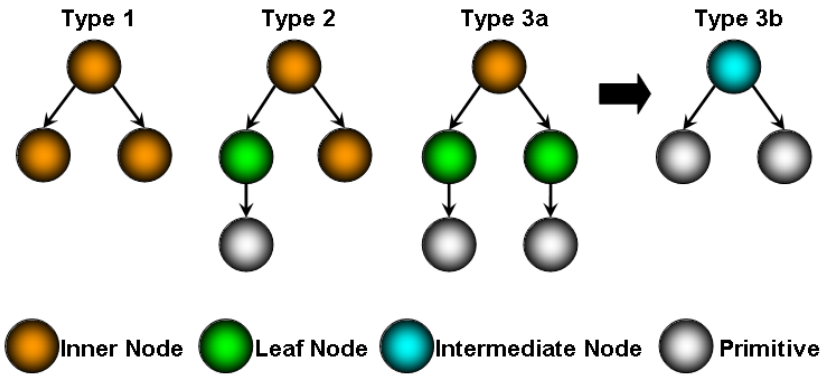

Figure 3.3: Different topologies of subtrees: type 3a is an inner node which has two leaf nodes as children and can be transformed into an intermediate node (type 3b), saving space of one inner node.

3.3 Compressed Node-Layout

Each inner node of the BPH stores the 6 crucial planes in two arrays of 3 floating point values, one for the min- and one for the max-planes. Additionally, two unsigned integers serve as the child-pointers, referencing a memory location either inside the array of inner nodes or inside the array of primitives in case of a leaf node.

If a balanced tree was assumed, one of these pointers could be spared if storing the corresponding children sequentially in memory, but as the BPH

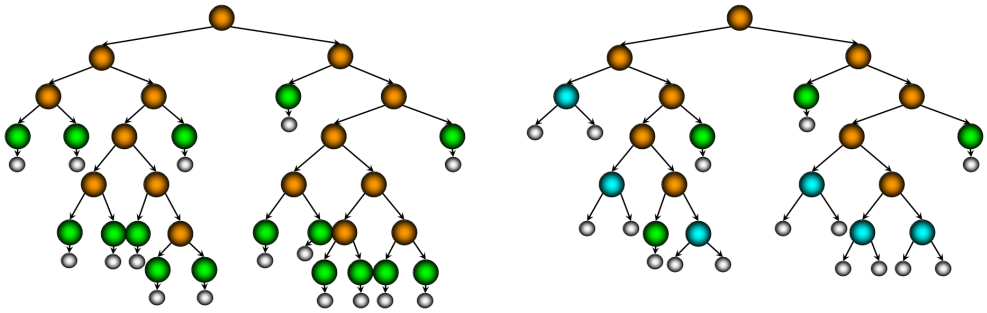

Figure 3.4: Example for replacement of inner nodes of type 3a by intermediate nodes. Cost of the structure before substitution is #inner nodes $*$ $sizeof(inner\ node) = 15 * 32$ bytes $= 480$ bytes, cost afterwards is $9 * 32$ bytes $= 288$ bytes (40% saving). Note that leaf and intermediate nodes are not explicitly stored and only shown for better comprehension. Legend: see Figure 3.3.

is designed to allow for arbitrary topologies of the tree, this optimization can not be applied: The underlying BVH is allowed to adapt to the scene as good as possible, also allowing inner nodes that have one leaf and one inner node as children (see Figure 3.3). In such a case, both pointers are obligatory and none can be discarded.

For correct traversal each inner node of the BPH needs 10 bits of additional information: 6 bits to map each plane to the left or right child, 2 bits that indicate if the left and/or right children are leaf nodes and another 2 bits that indicate intermediate nodes (they are not used if the child is an inner node). If the BPH is employed with Treelets, these 2 bits can be used to distinguish internal from external nodes of a Treelet (see Section 4.1).

In order to better fit cache-lines, all this information is stored in the lower 5 bits of each node's pointers which compresses inner nodes to 32 bytes/node (see Figure 3.5). The trade-off is a limited pointer-range of 27 bit that reduces the maximum total size of the scene that can be handled directly.

During runtime, the desired values are reconstructed by few efficient shift- and mask-operations whose influence on the overall performance is negligible.

3.4 Construction

The Bounding Plane Hierarchy is designed in such a way that an arbitrary binary BVH can easily be converted. There is only the assumption that the number of primitives per leaf node has to be fixed globally to be able

```
struct InnerNode {
    //planes of min- and max-dimensions
    float planes_min[3];
    float planes_max[3];

    //1 if plane belongs to right child, 0 otherwise
    bool rightPlane_min[3];
    bool rightPlane_max[3];

    //indices into node- or vertex-array
    unsigned int leftIndex, rightIndex;

    //flags set if children are leaf nodes
    bool leftIsLeaf, rightIsLeaf;

    //flags set if children are intermediate nodes
    bool leftIsIntermediate, rightIsIntermediate;
}

struct InnerNodeCompressed {
    float planes_min[3];
    float planes_max[3];
    unsigned int leftIndex;
    unsigned int rightIndex;
}

void encode_left(bool isLeaf, bool isInterm, bool rightPlanes[3]) {
    leftIndex = (leftIndex << 5) | (rightPlanes[0] << 4);
    leftIndex = leftIndex | (rightPlanes[1] << 3);
    leftIndex = leftIndex | (rightPlanes[2] << 2);
    leftIndex = leftIndex | (isInterm << 1) | isLeaf;
};

unsigned int decode_rightPlane_min(unsigned int i) {
    if (i == 0) return (left & 0x10) >> 4;  //10000
    if (i == 1) return (left & 0x8) >> 3;   //01000
    if (i == 2) return (left & 0x4) >> 2;   //00100
};
unsigned int decode_isInterm_left() { return (leftIndex & 0x2) >> 1; };
unsigned int decode_isLeaf_left() { return (leftIndex & 0x1); };
unsigned int decode_index_left() { return leftIndex >> 5; };
```

Figure 3.5: Compression of inner nodes and encoding/decoding-functions (only shown for left child). leftIndex/rightIndex contain all boolean values in their lower 5 bits.

to apply the intermediate-node-optimization. There are two ways to bypass this restriction: either the optimization is not performed or the restriction is relaxed as discussed in Section 8.1.4 at the cost of deteriorated memory-layout.

The transformation-algorithm recursively traverses the complete tree in depth-first order and converts each node. The conversion itself can be implemented by just a few lines of code (see Figure 3.6) that - for each dimension - match the corresponding planes of the parent node against the ones of its children. If a plane is not equal to either the minimum or the maximum plane of the parent, it is one of the crucial planes and has to be stored. Additionally, a bit is set that indicates if the plane belongs to the left or right child.

The fact that inner nodes store pointers to the primitives in the same location as pointers to usual child nodes requires one bit for each child that distinguishes inner nodes from lists of primitives (implicit leaf nodes). Another bit has to be encoded to indicate if the child is an intermediate node. After the node has knowledge of all these values, these bits are hidden inside the pointers using bit masks and shift-operations as shown in Figure 3.5.

3.5 Traversal

A naïve way of traversing the BPH is to simply reconstruct the complete AABBs of both children and intersect them using a standard ray-box intersection algorithm. This would introduce a fairly expensive computational overhead that would probably outperform any memory-related optimizations in all cases except for very large scenes.

The solution is a recursive traversal scheme that uses the hit-distances of the previously intersected parent node in combination with a slightly modified slabs algorithm [KK86]. First, the planes are mapped to their correct children by taking the ray's direction signs into account - if for one dimension the direction is negative, the corresponding min- and max-planes have to be switched. Second, intersection points of the ray with each plane and the corresponding distances are determined. In order to find out if the children are hit, the smallest maximum distance to one child's planes has to be larger than the corresponding biggest minimum. A graphical illustration is given in Figure 3.7.

This approach has approximately the same computational complexity as a standard slabs test, but it performs ray-box-intersection for both children at once instead of just one. Thus, it yields a theoretical speed up of 100% compared to a standard BVH traversal scheme, but is being slowed down by

```
InnerNode_BPH* convertToBPH(InnerNode_BVH *bvh_node, Box &parentBox) {

InnerNode_BPH *bphnode = new InnerNode_BPH();

//min-values
for (int i=0; i<3; i++) {    //loop over x-,y- and z-dimension
    if ( bvh_node->rightBox.min[i] == parentBox.min[i] ) {
        //no crucial plane of right child, store for left child in any case
        //small overhead if left box also shares this plane with parent
        bphnode->planes_min[i] = bvh_node->leftBox.min[i];
    } else {
        //crucial plane of right child -> store and set rightPlane_min[i]
        bphnode->planes_min[i] = bvh_node->rightBox.min[i];
        bphnode->rightPlane_min[i] = true;
    }
}
//max-values
for (int i=0; i<3; i++) {
    if ( bvh_node->rightBox.max[i] == parentBox.max[i] ) {
        bphnode->planes_max[i] = bvh_node->leftBox.max[i];
    } else {
        bphnode->planes_max[i] = bvh_node->rightBox.max[i];
        bphnode->rightPlane_max[i] = true;
    }
}

return bphnode;

}
```

Figure 3.6: C++-code for the conversion of a BVH into a BPH, showing how to decide which planes can be discarded.

the more complex mapping of the planes to their corresponding children.

The traversal algorithm includes a few optimizations that are explained in the following.

3.5.1 Packet Tracing

Packet Tracing exploits the high coherence between adjacent rays by working on packets of multiple rays: those rays corresponding to neighbouring pixels usually traverse the same space (and thus the same nodes of a spatial index structure), intersect with the same primitives and query the same shader with similar parameters. This allows for efficient usage of SIMD instructions that perform all computations on groups of 4 rays in parallel without overhead (except for the lack of suiting SSE instructions as discussed in Section 6.5).

The current implementation works on packets of arbitrary multiples of those groups of 4 rays that are computed sequentially, exploiting the coherency in terms of cache-hits when nodes or primitives that were intersected by a previous group are accessed by the next as well.

Coherence degrades quickly for finer tessellated scenes and larger packets (see observations in Section 5.4 and Table 5.4).

3.5.2 Masking Rays

Masking the first and last active rays of a packet helps reducing the number of ray-box and ray-triangle intersections. Whenever a currently first active ray did not intersect with a child, it is masked out and the next ray of the packet becomes the "first active" for this branch of the structure. The same is applied to the last active ray of the packet, so only a convex part of the packet is considered when traversing lower levels of the tree.

This can also be extended to all rays of a packet by the usage of a bitmask and is very effective in reducing overhead for incoherent packets (see Section 5.4 and Table 5.4).

3.5.3 Equally Signed Packets

Separate handling of packets that contain rays that do not share the same direction-signs helps optimizing the common case (same signs) which is less complex because computation of certain parts (e.g. the matching of planes to the children they belong to) can be done once for all rays. For packet sizes up to 16x16, the percentage of those cases is above 97%.

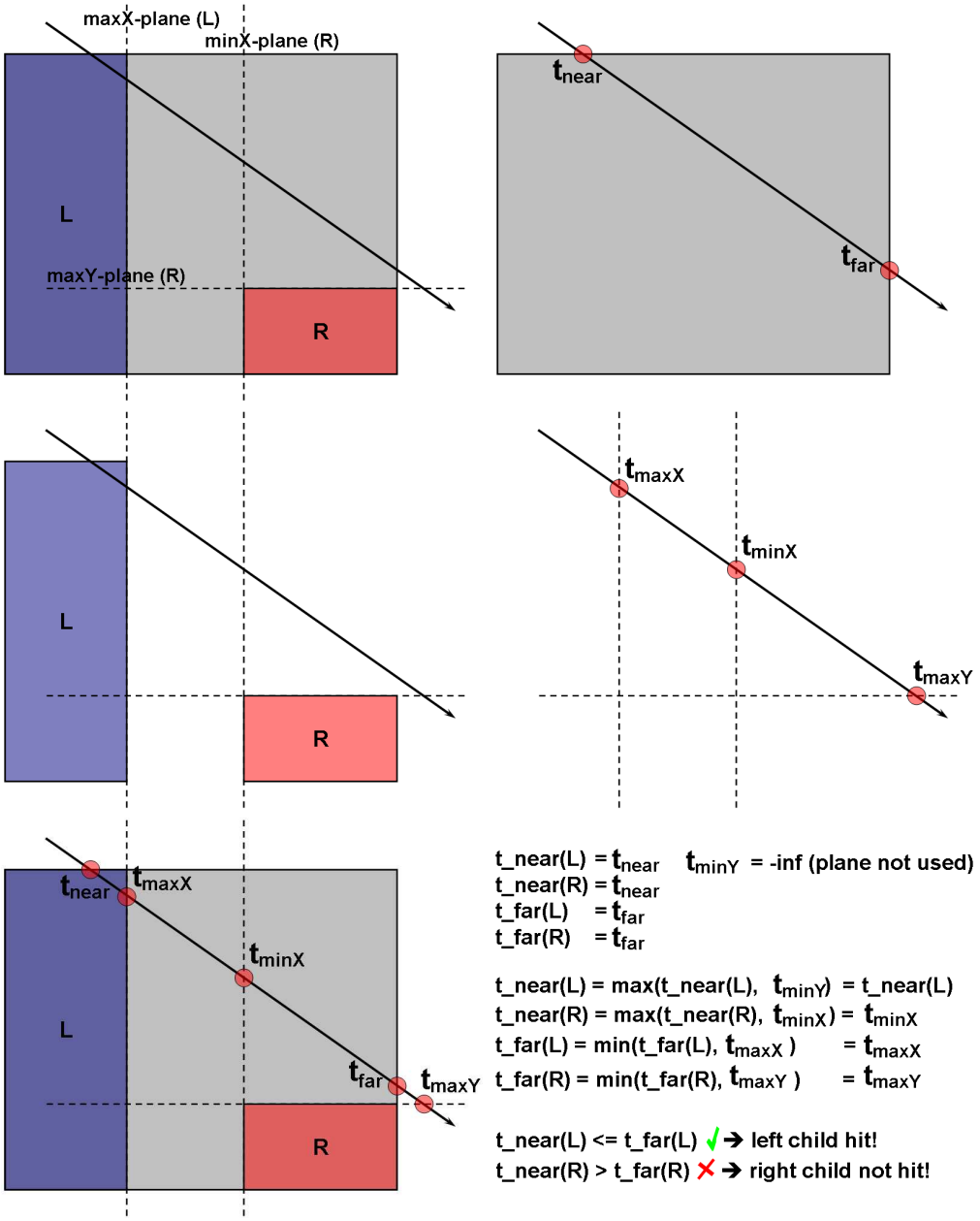

Figure 3.7: Example for the traversal of the BPH in 2D. Dashed lines represent the crucial planes stored in the node, red dots are intersection points. The parent box is coloured grey, the node's children are blue (left) and red (right).

3.5.4 Ordered Traversal

If both children of a node have a valid intersection, the nearest is traversed next and the other child is pushed onto the stack. This helps finding the nearest triangle-intersection as fast as possible.

3.6 Dynamic Scenes

In order to support dynamic scenes, simple recursive updating of the bounds is employed. No heuristically influenced complete rebuilds or other more sophisticated techniques (as e.g. in [LYTM06, WBS07]) have been used, the implementation is straightforward:

The algorithm recursively traverses down the tree in depth-first order. Each inner node recursively calls for an update of its children. At leaf level, the AABB of the enclosed primitives is recomputed. The AABB is returned, allowing the parent to save it and construct its own updated box out of the two boxes of its children. If the new AABB is equal to the old one, the current path does not have to perform any more updates and can immediately return. This procedure continues until the complete tree is updated.

As will be discussed in Section 6.2 again, this method is primarily efficient for coherently changing, deformable scenes and is very likely not to work very well for arbitrary dynamic scenes. However, it supports a very common type of animation convincingly well and is easy to apply.

Chapter 4
Treelets

Treelets are clusters of interconnected inner nodes used to enhance locality of traversal and thereby increase cache-efficiency. This basically points in the same direction as cache-oblivious/cache-aware algorithms as one goal is to increase cache-efficiency by reordering nodes in a way that those that are likely to be traversed subsequently are placed next to each other in memory. On the other hand, Treelets go one step further: they allow for completely independent computation on subsets of the acceleration structure which is perfectly suited for parallel and/or multi-threaded environments. This is especially useful for parallel processors that manage a local memory like the "Synergistic Processing Elements" (SPEs) of a CELL (see Section 4.3).

Treelets of an acceleration structure have a fixed number of nodes that can be arbitrarily chosen by the user, e.g. in dependence of the size and possible branching-factor of the scene (also see Section 8.2). Only inner nodes are grouped to form a Treelet, leaf nodes are always considered as outgoing edges that are not handled internally. Hence, triangle-intersection is "outsourced" and traversal of Treelets focuses entirely on inner nodes, allowing for specialized code-optimization.

4.1 Construction

The building procedure for Treelets recursively traverses a given binary tree alternating between breadth-first and depth-first order: A specified fixed number of nodes is selected using a breadth-first traversal scheme and is grouped together to form a Treelet. Outgoing edges on the other hand are put on a stack that produces a global depth-first ordering of the Treelets.

When constructing a new Treelet, memory for all nodes is allocated (Treelet-size multiplied with the size of an inner node) and all nodes are appended to an array in the order they are added to their Treelet. Only inner nodes are considered during this procedure, leaf nodes always produce an outgoing edge of the Treelet. If the Treelet is not completely filled yet, but no inner node can be added because the leaf level is reached, the algorithm checks if there already exists another partially filled Treelet that has enough empty slots to accommodate the new one. This is done in a greedy manner that uses the largest existing slot if no exactly fitting slot is available. If there is no such slot, a new one is created, storing the remaining number of unused nodes of the Treelet and the corresponding position in the array as an empty slot for other Treelets.

In unfortunate topologies, e.g. with high branching factor but only few nodes in each branch, this can happen several times subsequently, leading to more incompletely filled Treelets and eventually to more memory overhead. However, if a reasonable Treelet-size is chosen, the produced overhead is usually just a few nodes (see Table 5.5). This insertion-technique is essential for the memory-efficiency of the algorithm because it "recycles" most of the allocated but unused memory of incomplete Treelets. Without this optimization, the size of the structure quickly becomes a multiple of what it is without Treelets. The only other way would be to develop an algorithm that guarantees filled Treelets (see Section 8.2.2).

In order to be able to determine if a child node belongs to the current or a different Treelet during traversal, an additional bit of information has to be stored in each inner node for each child. This bit is set to the same value for all nodes of a Treelet and is alternated for all Treelets that are constructed on its outgoing edges. This guarantees that external nodes are always recognized correctly during traversal as there can never be two interconnected Treelets with the same value of that bit.

Figure 4.1 shows an example of how Treelets are applied to an arbitrary tree structure.

4.2 Traversal

The traversal scheme of Treelets consists of two stages, a global and a local one: The global scheme uses a stack and either traverses a leaf node or calls for traversal of a Treelet. This invokes the second stage, an independent traversal that - for this thesis's depth-first traversal - maintains its own small stack and operates only on the local nodes of the current Treelet,

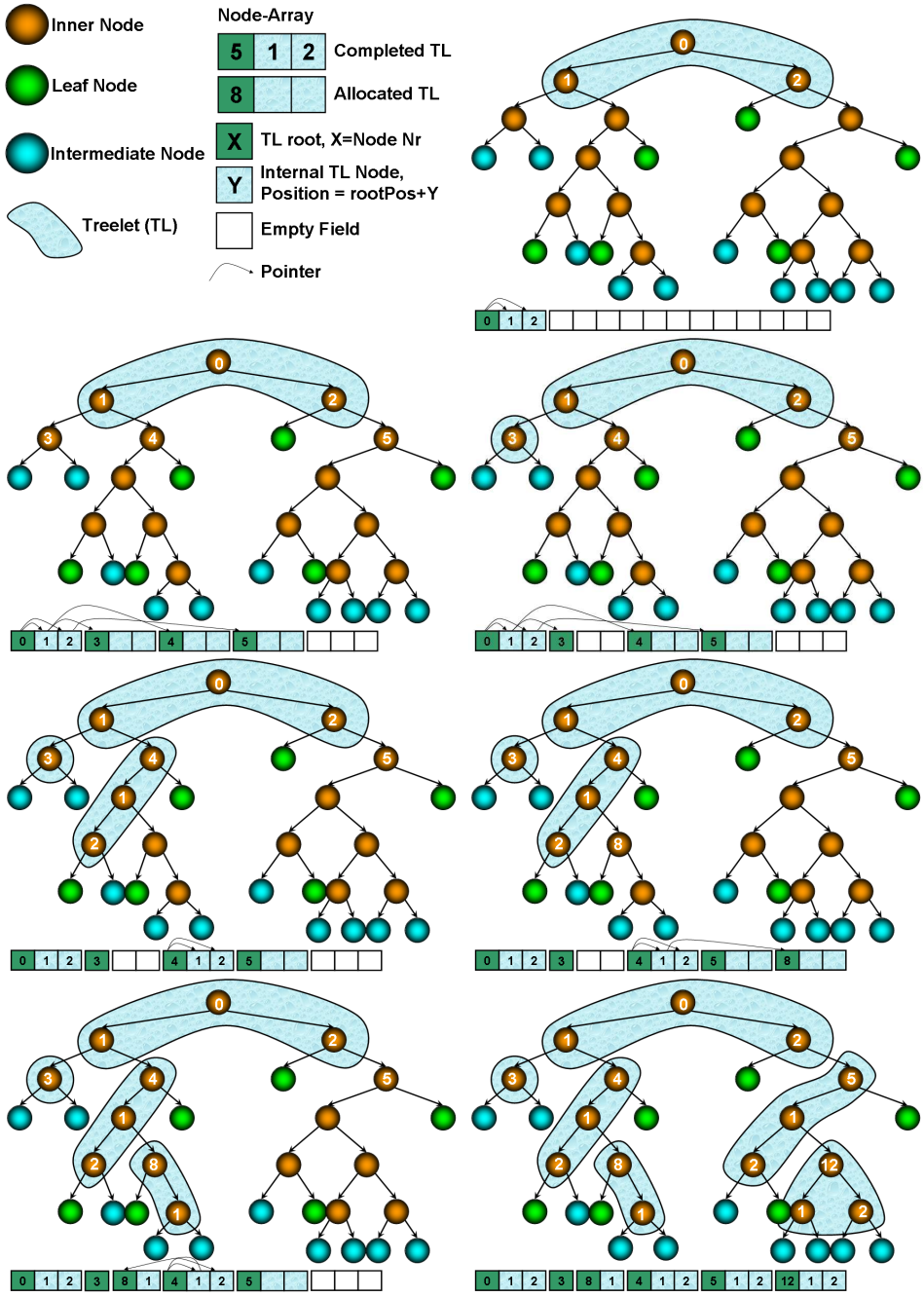

Figure 4.1: Example for construction of Treelets of size 3. For better readability, only some pointers are shown.

starting with the root. If an internal child node of the Treelet has to be traversed, it is pushed onto this local stack. Every time an outgoing edge has to be taken, the corresponding inner or leaf node is pushed onto the global stack, but the algorithm continues locally until the Treelet is traversed completely, indicated by an empty local stack. This behaviour leads to a reversed traversal order of the outgoing edges: In a standard depth-first traversal, the leftmost outgoing edge would be traversed first. Since the internal depth-first traversal of the Treelets directly pushes outgoing edges onto the global stack, the rightmost edge is pushed lastly and is thereby traversed first. As discussed in Section 8.2.4, some ordering of these edges could be an option worth exploring.

Globally, this two-stage algorithm results in a "broadened" depth-first ordered traversal that is likely to have a larger working set of nodes than a standard depth-first traversal due to the fact that the algorithm first computes intersection of all internal nodes of a Treelet before testing outgoing edges that point to primitives. Influence of this overhead on a single-processor machine can be observed in Section 5.5 and in Table 5.6. Pseudo-code for the traversal is shown in Figure 4.2.

4.3 Implementation On The CELL Processor

On a high level of abstraction, the CELL processor consists of three main components:

- Power Processing Element (PPE) -
  the main core (64-bit Power Architecture),
- Synergistic Processing Elements (SPEs) -
  eight co-processors scheduled by the PPE,
- Element Interconnect Bus (EIB) -
  a high-bandwidth circular data bus connecting all components.

The PPE works as a controller that maintains the global stack and schedules free SPEs with tasks. Each task is to traverse a given Treelet with a given packet of rays and return outgoing edges that have to be traversed consecutively. Each SPE has its own local memory, so it can work on a complete Treelet with only one memory access when being scheduled. Every memory access during computation is then fed by the local store, resulting in minimized latency. After successful traversal, the SPE communicates its results back to the PPE, either indicating no valid intersection (finishing that

```
void traverse() {
globalStack.push_back(root);

while (globalStack not empty) {
    currentNode = globalStack.pop_back();

    if (currentNode == Leaf Node) intersect Triangle(s);
    else {
        localStack.push_back(currentNode);

        while (localStack not empty) {
            currentTLNode = localStack.pop_back();
            if (leftChild intersected) {
                if (leftChild internal && no Leaf)
                    localStack.push_back(leftChild);
                else
                    globalStack.push_back(leftChild);
            }
            if (rightChild intersected) {
                if (rightChild internal && no Leaf)
                    localStack.push_back(rightChild);
                else
                    globalStack.push_back(rightChild);
            }
        } //Treelet Node traversed

      } //Treelet or Leaf Node traversed
  }

}
```

Figure 4.2: Pseudo code for the traversal of a Treelet-based acceleration structure.

branch) or supplying a list of outgoing edges or nodes that define the lower levels outside the finished Treelet that have to be traversed. The PPE pushes these nodes onto its global stack, maybe even performing some ordering optimization, e.g. by sorting the nodes by hit distances or by scheduling leaf before inner nodes. Every time an SPE has completed a task and returned its results, it is again scheduled by the PPE with the next node from its global stack, eventually completing the traversal of this packet by the time the stack is empty. This approach is suited to use all SPEs to full capacity regardless of the structure's topology, but this is bought by a lot of communication that could result in bad overall performance.

A variant with less communication overhead would be to let each SPE traverse a complete branch consisting of several Treelets by requesting subsequent Treelets directly from main memory in order to minimize communication with the PPE. After successful traversal down to leaf level, an SPE would return those rays to the PPE that travelled down all the way to leaf level. The way intersection and shading of primitives is handled is a different question that can be done independent of the traversal-routine.

Both schemes make parallel intersection of up to as many Treelets as SPEs are on the chip possible because all SPEs work on disjoint subsets of the index structure and traversal is not influenced by different branches (except for discarding branches if a nearer intersection has been found). On the current CELL architecture 8 SPEs are available, each with a local store of 256 KB memory without cache. This allows loading one or even several complete Treelets (up to a total of over 8000 BPH-nodes) onto each SPE. Since there is no hardware caching-mechanism, a software-cache has to be used which is costly to access, but the use of Treelets amortizes this access over many nodes.

Chapter 5
Results

This chapter describes how the BPH performs with respect to memory requirement and rendering performance, additionally pointed out by some statistics for traversal. The tests confirm that the Bounding Plane Hierarchy is very well suited for saving large amounts of memory - in fact it reduces the required memory of a scene's index structure by a minimum of 35% and up to 75% compared to current acceleration structures (Section 5.2). Although achieving interactive framerates, the performance of the BPH is inferior to an optimized BVH (Section 5.3).

As for Treelets, the effectiveness of the construction-algorithm is shown in terms of its small memory overhead. Due to the fact that no suiting architecture has been tested, the provided statistics for the BPH with Treelets concentrate on the memory overhead produced. Performance of the BPH on a single-core CPU drops about 10-35% when using Treelets (Section 5.5).

5.1 Configuration

All tests were run on a Samsung X10 notebook with an Intel Centrino Processor (Pentium M) at 1.7 GHz and 512MB of RAM. The operating system was Kubuntu Linux 6.06 with kernel 2.6.

The chosen scenes consist of mostly single objects, one indoor-like ("toasters") and one outdoor-scene ("fairy") between 12 and 1,000,000 triangles, dynamic scenes range from 5,000 to 170,000 triangles with several animation-frames (see Figure 5.1 and 5.2 and Tables 5.1 and 5.3). They were rendered at a resolution of 512x512 pixels without any complex shading (basic "eye-light"-shading).

Performance comparison is drawn with a highly optimized BVH implementation ("wBVH") after [WBS07], the memory requirement is also compared to the B-KD-tree ("BKD") and the unoptimized BVH ("BVH") used for conversion to the BPH to see direct improvement. See Page 57 for references of the scenes.

Figure 5.1: The static test scenes: ground, shuttle, bunny, dragon, buddha. Additionally, the first frames of the dynamic scenes were used.

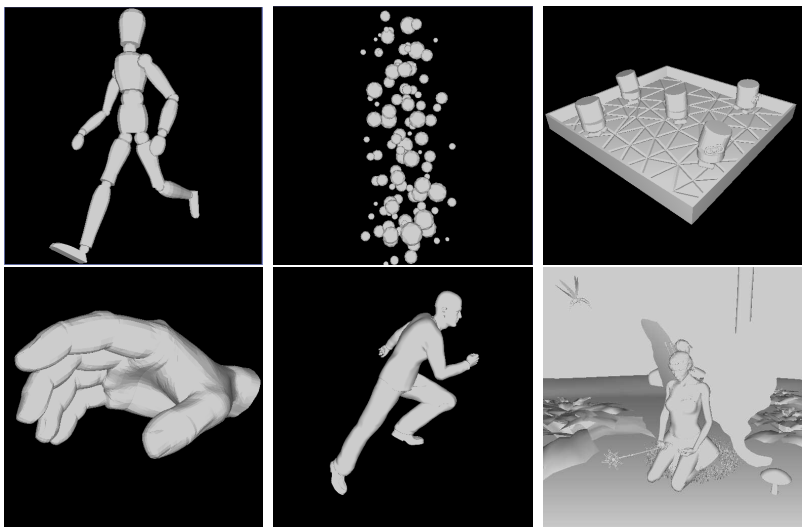

Figure 5.2: The dynamic test scenes: wood-doll, marbles, toasters, hand, ben, fairy.

Scene	Triangles	BVH	wBVH	BKD	BPH	BPH2
ground	12	0.5K	0.3K	0.3K	0.2K	0.1K
shuttle	616	31K	19K	20K	12K	0.8K
wooddoll	5,378	274K	181K	172K	105K	66K
marbles	8,800	466K	336K	282K	178K	110K
toasters	11,141	580K	356K	356K	221K	141K
hand	15,855	815K	525K	507K	310K	198K
bunny	69,451	3,373K	2,399K	2,222K	1,285K	755K
ben	78,029	3,962K	2,409K	2,497K	1,509K	955K
fairy	174,117	8,785K	5,331K	5,572K	3,347K	2,126K
dragon	871,414	45,504K	30,614K	27,885K	17,335K	10,975K
buddha	1,087,716	56,762K	37,977K	34,807K	21,624K	13,720K

Table 5.1: Overall size of the acceleration structures for the test-scenes (in bytes, triangle-data not included). The BPH only needs 38% of the size of an unoptimized BVH, and around 60% of a B-KD-tree or the highly optimized BVH ("wBVH") that uses 1-4 triangles per leaf. If the BPH uses 2 triangles per leaf node ("BPH2"), percentages even drop to 24% (BVH) and around 38% (wBVH, BKD).

5.2 Memory Requirement

Table 5.1 shows memory consumption of different index structures on the test scenes. The BPH on average saves over 60% of space compared to the unoptimized BVH, and still around 40% in comparison to the B-KD-tree and the highly optimized BVH. The latter on the other hand does not have a fixed number of primitives per leaf (whereas BVH, BKD and BPH go down to one per leaf) but heuristically stores between 1 and 4 primitives with an average of 2 on the tested scenes. This can theoretically save a complete level of the tree which corresponds to 50% of its total size. A comparison to a BPH with 2 primitives/leaf is more even and shows over 60% savings instead of 40% when compared to the single-primitive version.

Figure 5.3 gives a good overview of what memory overhead (additional to storing the primitives) will be introduced by the different structures if used for a scene whose number of triangles is known in advance: On average, the unoptimized BVH needs over 50 bytes per primitive, the optimized variant (mostly due to multiple triangles per leaf) and the B-KD-tree about 32 and the BPH below 20 - if allowing 2 triangles per leaf, the BPH even drops to 12 bytes per primitive.

The total amount of saved space can be traced back to the different

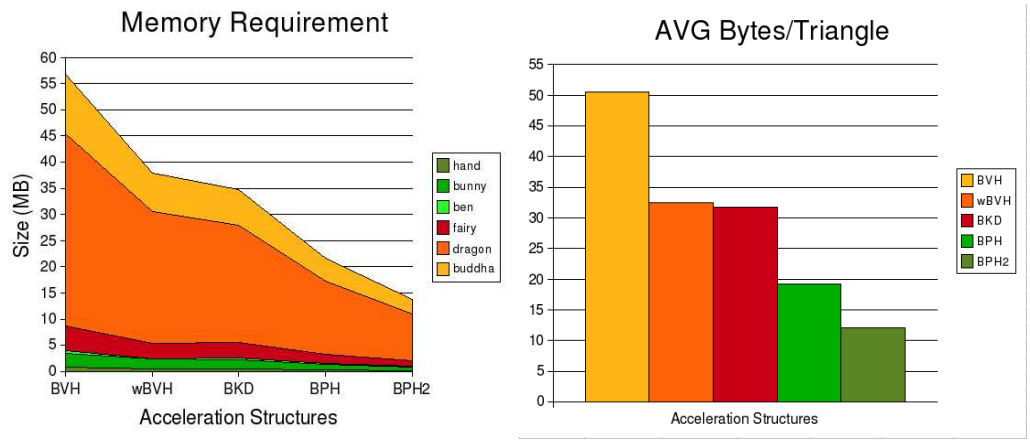

Figure 5.3: Left: memory consumption of the different acceleration structures. Right: average amount of memory used in relation to the scene's triangles.

design decisions of the BPH. The largest part of course results from the basis of the BPH: the fact that each node only stores a maximum of 6 planes instead of two full AABBs reduces the size of each individual node by 50% (as described in Section 3.1). Being able to handle the last level above the primitives implicitly because each node stores its children's bounds also helps keeping the explicitly stored tree as small as possible.

The usage of intermediate nodes (see Section 3.2) also has a large impact on the overall size. Table 5.2 shows that on average 65% of all leaf nodes are in fact intermediate nodes which saves about 35% of the total size.

5.3 Rendering Performance

The rendering frame rates of the BPH show real-time performance between 1 and 29 frames per second for the static and between 1 and 10 frames per second for the dynamic scenes at a resolution of 512x512 pixels. Frame rates drop towards bigger scenes due to the computational complexity and towards larger packet sizes due to the decreasing coherency of the contained rays. As Table 5.3 shows, this can be observed especially for a packet size of 16x16 where the simple "ground"-object with only a few primitives is rendered faster than with smaller sizes due to the high coherence of rays, whereas for the massive models the frame rate drops substantially because due to the fine tessellation the coherence is worse than for smaller sizes.

Scene	% of total leafs	bytes saved	% of total size
ground	71	140	42
shuttle	63	6,636	35
wooddoll	65	59,108	36
marbles	59	90,944	34
toasters	61	118,720	35
hand	63	172,312	36
bunny	73	820,428	39
ben	66	864,276	36
fairy	67	1,946,980	37
dragon	61	9,231,712	35
buddha	61	11,535,216	35
AVG	64	-	36

Table 5.2: Savings due to the usage of intermediate nodes. Each intermediate node saves one inner node (= 32 bytes). See Section 3.2 and Figures 3.3 and 3.4.

The performance of the current implementation is clearly inferior to the chosen reference, the highly optimized BVH. This is mostly a result of the shortcoming of algorithmic optimizations (e.g. no SAH, no "early hit test" and no frustum culling were implemented, see Section 6.4) as well as a general amount of inefficiency by reason of abstract design and lack of code optimization.

Table 5.3 shows frame rates both for the BPH and the highly optimized BVH for different packet sizes, an illustration is given by Figures 5.4 and 5.5.

Scene	wBVH			BPH		
	4x4	8x8	16x16	4x4	8x8	16x16
ground	60	84	90	21	24	29
shuttle	51	72	77	15	16	17
wooddoll	36	47	44	11	11	11
marbles	26	35	32	7	7	6
toasters	25	35	33	5	5	5
hand	22	31	28	7	7	6
bunny	17	21	17	6	5	4
ben	22	26	22	7	6	4
fairy	7	11	9	2	2	2
dragon	9	8	6	3	2	1
buddha	8	7	5	3	2	1
wood-doll(29)	37	45	43	10	10	9
marbles(500)	20	31	30	5	5	5
toasters(245)	23	31	30	5	5	5
hand(44)	21	29	27	6	6	6
ben(30)	18	24	19	4	5	5
fairy(21)	6	9	8	2	2	1

Table 5.3: Performance comparison of the BPH with the highly optimized BVH for different packet-sizes, measured in frames per second. The current implementation of the BPH is on average 3-5 times slower, worsening towards bigger packet-sizes and larger scenes due to missing early hit- and frustum-tests. Brackets indicate the number of frames the dynamic scenes consist of. All numbers are average values for representative views.

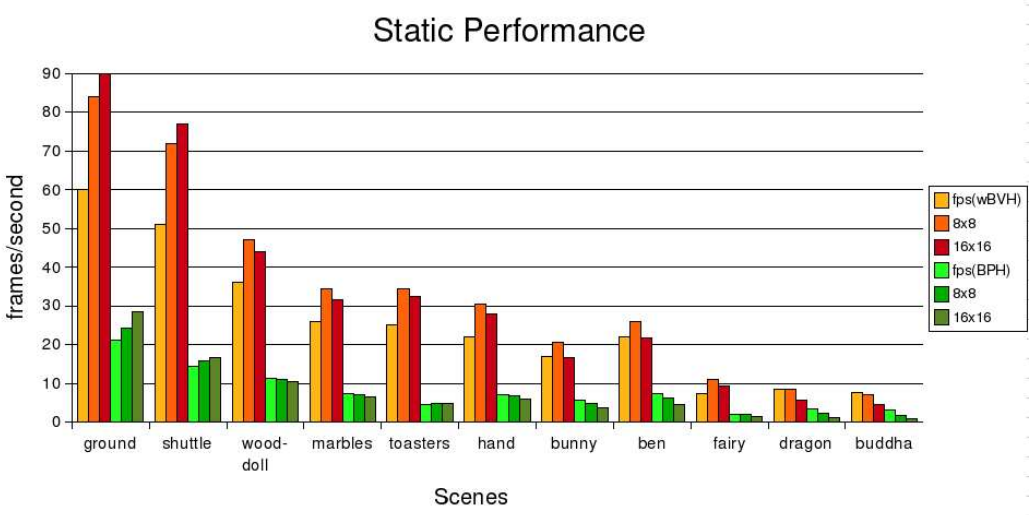

Figure 5.4: Rendering performance of the BPH compared to a highly optimized BVH for different packet sizes (static scenes).

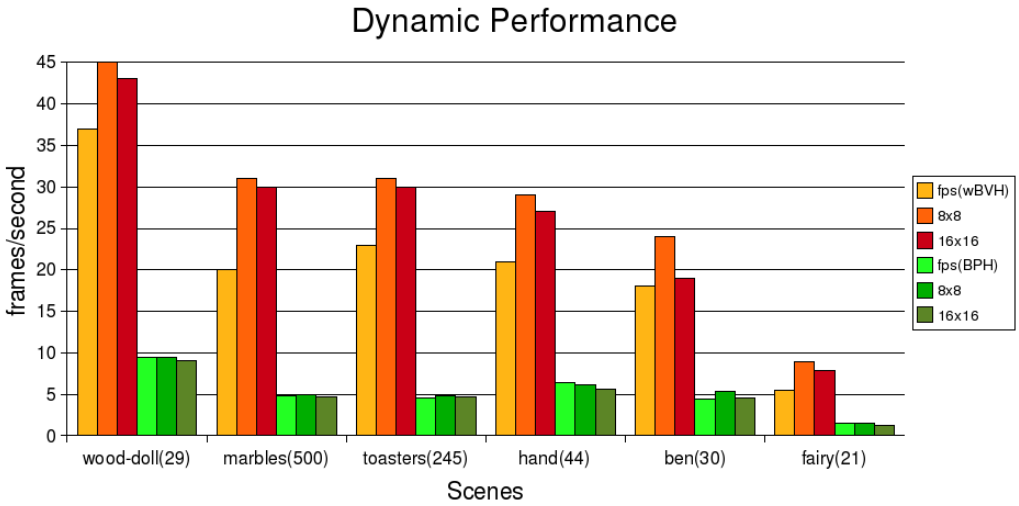

Figure 5.5: Rendering performance of the BPH compared to a highly optimized BVH for different packet sizes (dynamic scenes).

5.4 Traversal Statistics

Table 5.4 provides further statistics for traversal and masking of rays. The number of packet-node- and packet-triangle-intersections allows for a more detailed analysis of the BPH's performance than the overall outcome in fps, as it is platform-independent and only represents the algorithmic efficiency in finding the correct intersection as fast as possible. Just as observed for the frame rates (which of course result mostly from the intersections), the number of intersections increases towards larger scenes simply because the tree is larger and towards larger packet-sizes because of decreasing coherence of the rays.

Table 5.4 also shows that this performance is to a large degree more depending on the topology of the scene and the resulting tree than on the raw number of triangles as e.g. the fairy has only a fifth of the primitives as the buddha but for smaller packet sizes needs more traversal steps to find the best intersection, because the buddha is a quite regular, balanced structure which results in a more balanced tree. For the larger packet-sizes on the other hand, intersections for the statue increase rapidly which is a result of the fine tessellation: a packet intersects with a part of the structure which consists of many more triangles that have to be tested than in case of the fairy.

The last columns of Table 5.4 show high efficiencies for keeping track of the rays that did not intersect with the parent-node to save on their computation on all lower levels of the current path. The percentage of these rays of a packet that can be masked out this way increases for larger scenes and larger packet-sizes. This is closely related to the statistics for intersection, because decreasing coherence of the individual rays of a packet causes more rays to be masked out each step, resulting in only a few rays arriving on leaf level. As this implies that a packet has to go down lots of different paths, the overall number of intersections increases. By using masks, the influence of the increasing computational complexity for decreasing coherence is reduced.

5.5 Treelet Overhead

The results in Table 5.5 show that the building-algorithm works highly effective in terms of memory overhead: although there are usually lots of incompletely filled Treelets (i.e. the desired size of nodes could not be grouped together due to the topology of the acceleration tree), there is rarely more than one Treelet where the empty allocated nodes could not be filled up with other Treelets. Thus, the memory overhead produced usually lies well below

Scene	#node-isecs/packet			#tri-isecs/packet			%rays masked out		
	4x4	8x8	16x16	4x4	8x8	16x16	4x4	8x8	16x16
ground	3	3	4	3	3	4	1	3	6
shuttle	6	6	2	1	1	3	10	32	57
wooddoll	8	10	7	1	3	9	31	66	86
marbles	14	17	12	2	4	16	35	69	88
toasters	22	25	13	4	7	18	20	50	76
hand	13	17	15	3	6	12	33	67	87
bunny	16	23	35	4	11	82	48	81	94
ben	12	18	30	3	9	53	48	81	94
fairy	50	66	89	17	32	130	25	62	86
dragon	26	59	162	11	42	473	69	92	98
buddha	30	76	207	13	51	471	70	92	98

Table 5.4: Traversal statistics of the BPH: average number of packet-node-intersections, packet-triangle-intersections and average percentages of masked out rays per packet.

1% of the total size.

The building is by definition (see Section 4.1) very sensitive to the given structure of the tree, which means that for certain unfortunate topologies the algorithm may produce very different overhead for different Treelet-sizes, which can be observed e.g. at size 512 for the bunny.

Traversal of the BPH with Treelets usually performs 10-20% more packet-node intersections than the reference without Treelets for the smaller scenes and 20-35% more for the larger scenes. This is largely dependent on how the Treelets of the chosen size adapt to the given topology (see Table 5.6). For larger trees, the overhead of intersections that would not be performed without the complete traversal of each Treelet grows due to more traversal steps. However, with a suited "early miss"-test (e.g. using frustum culling) as discussed in Section 6.3 these cases should not be too expensive.

Another important aspect is the fact that on the testing machine, the computational complexity probably outperforms the memory access and thus reduces the effect of improved memory and cache efficiency. An adapted implementation on a suited architecture like the CELL or any on-chip multi-core processor was not tested but should be able to gain large amounts of performance out of the greatly enhanced cache-efficiency in trade for this overhead.

Scene	TL2	TL3	TL7	TL8	TL16	TL64	TL256	TL512	TL10K
ground	0(0)	0(0)	32(1)	64(1)	320(1)	2K(1)	8K(1)	16K(1)	320K(1)
shuttle	0(0)	0(0)	0(0)	192(2)	192(1)	192(1)	4K(1)	4K(1)	308K(1)
wooddoll	0(0)	32(1)	96(1)	192(1)	448(1)	2K(1)	2K(1)	10K(1)	215K(1)
marbles	32(1)	64(1)	0(0)	32(1)	32(1)	544(1)	3K(1)	3K(1)	142K(1)
toasters	0(0)	0(0)	64(1)	128(1)	384(1)	384(2)	384(1)	9K(1)	99K(1)
hand	0(0)	64(1)	64(1)	128(2)	384(1)	896(1)	9K(13)	1K(1)	9K(1)
bunny	32(1)	0(0)	96(1)	96(1)	352(1)	1K(1)	1K(1)	10K(32)	315K(1)
ben	32(1)	64(1)	160(2)	224(2)	224(1)	224(1)	6K(1)	15K(1)	91K(1)
fairy	32(1)	64(2)	192(2)	96(1)	352(1)	2K(1)	4K(1)	12K(1)	173K(1)
dragon	32(1)	32(1)	0(0)	96(1)	96(1)	2K(1)	8K(1)	16K(1)	265K(1)
buddha	32(1)	32(1)	64(1)	32(1)	32(1)	1K(1)	3K(1)	3K(1)	136K(1)
AVG incTL	0.55	0.73	0.91	1.27	1	1.09	2.09	3.82	1
AVG bytes	18	32	70	116	256	1K	4K	9K	191K

Table 5.5: Memory overhead introduced by usage of Treelets in Bytes. TLx indicates that each Treelet consists of x inner nodes, brackets denote the number of incomplete Treelets, "AVG incTL" the average unfilled Treelets and "AVG bytes" the corresponding average overhead in terms of additional memory requirement compared to the underlying index structure. For example the bunny has one incomplete Treelet when using 256 nodes/Treelet, resulting in approximately 1KB overhead and for a Treelet-size of 8 the scenes have an average of 1.27 incomplete Treelets resulting in an average of 116 bytes overhead.

Scene	BPH	TL2	TL3	TL7	TL8	TL16	TL64	TL256	TL512	TL10K
ground	3	3	3	3	3	3	3	3	3	3
shuttle	6	6	6	6	6	6	7	7	7	7
wood-doll	8	9	9	9	9	9	9	9	9	9
marbles	14	14	15	15	15	15	15	15	15	16
toasters	22	23	24	23	23	23	25	25	25	28
hand	13	16	16	14	14	15	16	17	18	22
bunny	16	18	19	19	19	20	20	21	22	25
ben	12	14	15	14	14	15	14	14	14	16
fairy	50	60	56	59	61	58	61	63	63	64
dragon	26	31	33	43	35	30	34	33	33	38
buddha	30	44	43	56	53	39	41	39	41	43
AVG	18	22	22	24	23	21	22	22	23	25

Table 5.6: Traversal overhead introduced by usage of Treelets, measured as the average number of packet-node-intersections. BPH denotes the reference without usage of Treelets, TLx indicates the Treelet-size (each Treelet consists of x inner nodes). Packet size was set to 4x4.

Chapter 6

Drawbacks

This chapter gives an overview of the most important factors limiting this thesis's implementation. For one thing the BPH inherits some general drawbacks of Bounding Volume Hierarchies that are a topic of current research and for another thing there are some algorithmic techniques that were not used (Surface Area Heuristic, "early hit test" and frustum culling). Finally, the SSE instruction set does not supply certain operations that could have been useful.

6.1 Construction Using First Frame

For a static scene it is easily possible to find a good acceleration structure that matches the scene's topology. If rendering an animation without a priori knowledge of its deformations however, the only way to construct the hierarchy is to use the first frame. This frame of course not necessarily describes a state of the scene that is in any case representative, resulting in a possibly inefficient topology for the complete set of frames.

An illustration for such a case can be given by a person with folded arms: In the first frame, his/her left hand is on his right side and vice-versa, so the structure will be built with primitives of the left hand inside a subtree where mostly parts of the right half of the body reside. However, usually the person's left arm will be on his left side and vice-versa, so the bounds of both arms will not represent a common state of the animation. This will result in a scenario like the one described in the next section that can only be avoided if the structure is somehow (at least partially) rebuilt during rendering.

6.2 Updating Without Rebuilds

If primitives of the same subtrees move to opposing corners of the scene, their corresponding bounding volumes enlarge and enclose large parts of the scene. Thus, they have to be intersected with many more incoming rays than just the few that actually hit the primitives. This has no critical influence if the primitives reside in different subtrees that merge relatively high up in the tree, as this will only result in a few more cases where all rays miss the next subtree (one "early hit test" and one "early miss" as explained in Section 6.3), but if primitives of a subtree close to leaf level are affected, this complete part of the tree has to be traversed by all rays, eventually resulting in major performance decrease.

A detailed model of a large city serves as a good example: Suppose in the beginning of the animation a group of people is standing in an elevator that descends from the top of a skyscraper down to the ground floor. They would - depending on the depth of the BVH - be grouped together at a relatively low level not far above the leaf nodes. Now let all people be moving to different corners of the scene, maybe even one on top of another skyscraper and one down into the subway.

In such a case, the bounding volume of the group will be enlarged over the whole city and with it all parent nodes up to the root node. For a BVH-depth of N and the group being one level above the leaf nodes this would lead to N-1 additional early hit tests plus one frustum exit. The breakpoint is that this overhead has to be computed for every ray shot into the scene, while maybe only 1% or even none of the rays will really have an intersection with one of the people from the group. Figure 6.1 shows a simplified example of such a scenario.

Nevertheless there is a lot of research focusing on partial [WBS07] or complete [LYTM06] rebuilds that are only used if the topology of the acceleration tree (or of a subtree) is being considered too inefficient by some metric. These rebuilds make efficient rendering of many more kinds of animation possible than just coherent deformations and are a topic for future enhancements of the BPH.

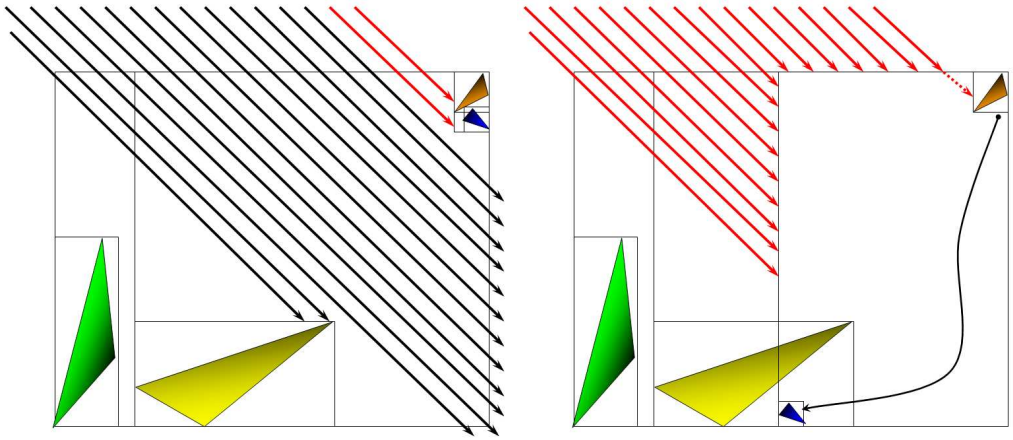

Figure 6.1: Simplified example of a situation where updating of a BVH without complete rebuilds results in inefficient topology. Left: only two rays intersect with the AABB in the upper right. Right: one of the triangles of that subtree has moved to the lower left. Now all rays intersect with the box and thus have to test for intersection with both children although only one ray really hits.

6.3 Early Hit Test & Frustum Culling

Basically all algorithmic techniques used for BVHs or other derivations should be applicable to the BPH as well. Yet, the algorithmic complexity is higher and especially the need for a stack-based traversal scheme can impose problems to certain optimizations like an "early hit test" in the style of [WBS07]: This optimization checks for an intersection of the first ray and immediately enters the corresponding child, sequential testing of all rays is only done if this test fails. This possibly introduces overhead in terms of rays traversing nodes they don't hit (rays that would usually be masked out) but on the other hand saves all intersection computation of the remaining rays of the packet.

This test has shown to be especially useful in combination with a frustum test that, if the early hit test fails, conservatively checks if the complete packet misses the box. This way, lots of individual ray-box-tests can be spared. See Section 8.1.1 for an explanation of how this could be applied to the BPH.

6.4 Surface Area Heuristic (SAH)

The current implementation does not pay attention on how to build a good acceleration structure in the first place. This means a simple "spatial median" split is used in each step of the building phase, subdividing the scene in two equally sized parts and then fitting the AABBs to the corresponding underlying geometry. It has been shown that using heuristics to choose the possibly best splitting-plane to adapt to scene topology usually results in significant performance increase. Although there is still no proven "best" way to do this, the so-called "Surface Area Heuristic" (SAH) [GS87] has shown to be very effective, resulting in up to two times speedup over spatial median and up to six times over object median (i.e. choosing the splitting plane where it divides the objects equally) [WBS07].

When building a BVH/BPH using a SAH-based algorithm, the plane that subdivides the current node is chosen heuristically as the one that minimizes the expected time of a random ray intersecting the bounding volume. This expected time is estimated by the SAH:

$$T = 2T_{AABB} + \frac{A(S_L)}{A(S)}N(S_L)T_{tri} + \frac{A(S_R)}{A(S)}N(S_R)T_{tri},$$

where T is the execution time for an average ray intersecting with the root node, T_{AABB} and T_{tri} are the times for intersection with an axis-aligned bounding box and a triangle, S is the set of triangles in the root node, S_L and S_R are the subsets of the two child nodes, $A(S)$ is the area of the bounds of the triangles in set S and $N(S)$ is the number of triangles in set S.

Given a root node and a splitting plane, this heuristic computes for both child-nodes the ratio of the AABB-surface area to the root-surface area and multiplies each with the corresponding intersection time for all triangles inside the child. These times are added up together with the time for the two AABB-ray-intersections of the two child-nodes.

6.5 SSE Instruction Set

The "Streaming SIMD Extensions" for current CPUs do not provide certain desirable functions that could have been helpful for the implementation of the intersection-algorithm of the BPH and could possibly make the code much faster.

As an example, a "conditional select"-function would be very useful. Such a function would construct a SIMD-vector `A` out of two other SIMD-vectors `X` and `Y` depending on boolean evaluation of a third SIMD-vector (or a 4bit-immediate constant) `B` for each entry of the vectors:

```
_mm_select_cond_ps(__m128 A, __m128 X, __m128 Y, __m128 B) :=
{
A[0] = B[0] ? X[0] : Y[0];
A[1] = B[1] ? X[1] : Y[1];
A[2] = B[2] ? X[2] : Y[2];
A[3] = B[3] ? X[3] : Y[3];
}
```

An even more suitable variant would be to swap values depending on a boolean evaluation:

```
_mm_swap_cond_ps(__m128 X, __m128 Y, __m128 B) :=
{
if B[0] then swap(X[0], Y[0]);
if B[1] then swap(X[1], Y[1]);
if B[2] then swap(X[2], Y[2]);
if B[3] then swap(X[3], Y[3]);
}
```

The intersection-algorithm needs to accommodate to cases where rays come from directions with negative signs on some or all axes. In such a case, at first the min- and max-planes of those dimensions with negative signs have to be switched. Afterwards these planes have to be mapped to the node's left and right child which is done by using the encoded information of the node. Both steps perform operations that could nicely be handled with such conditional-load- or conditional-switch-operations. The sign correction is shown in the example code of Figure 6.2.

```
//bounding planes of children, saved as SSE-vectors
__m128 planes_min = node.planes_min;
__m128 planes_min = node.planes_max;

//direction signs of packet, negative sign if (dir < 0)
__m128 packet_dir_sign = _mm_cmplt_ps(packet.dir, _mm_zero);
```

```
WITHOUT SELECT/SWAP-INSTRUCTION:
float planes_min_x, planes_min_y, planes_min_z;
float planes_max_x, planes_max_y, planes_max_z;

//check signs & correct planes
if (((float*)&packet_dir_sign)[0]) {
    planes_min_x = ((float*)&planes_max)[0];
    planes_max_x = ((float*)&planes_min)[0];
} else {
    planes_min_x = ((float*)&planes_min)[0];
    planes_max_x = ((float*)&planes_max)[0];
}
if (((float*)&packet_dir_sign)[1]) {
    planes_min_y = ((float*)&planes_max)[1];
    planes_max_y = ((float*)&planes_min)[1];
} else {
    planes_min_y = ((float*)&planes_min)[1];
    planes_max_y = ((float*)&planes_max)[1];
}
if (((float*)&packet_dir_sign)[2]) {
    planes_min_z = ((float*)&planes_max)[2];
    planes_max_z = ((float*)&planes_min)[2];
} else {
    planes_min_z = ((float*)&planes_min)[2];
    planes_ma_xz = ((float*)&planes_max)[2];
}
//write back sign-corrected planes
planes_min = _mm_load_ps(planes_min_x, planes_min_y, planes_min_z, 0);
planes_max = _mm_load_ps(planes_max_x, planes_max_y, planes_max_z, 0);
```

```
WITH SELECT-INSTRUCTION:
_mm_select_cond_ps(planes_min, planes_max, planes_min, packet_dir_sign);
_mm_select_cond_ps(planes_max, planes_min, planes_max, packet_dir_sign);
```

```
WITH SWAP-INSTRUCTION:
_mm_swap_cond_ps(planes_min, planes_max, packet_dir_sign);
```

Figure 6.2: Example-code for the sign-correction of BPH-planes without and with conditional-select and conditional-swap instructions

Chapter 7
Conclusion

This thesis introduces two new techniques to improve memory efficiency of spatial indices for Ray Tracing.

The Bounding Plane Hierarchy reduces the size of previous index structures by up to 75% while still running at interactive frame rates and supporting animated scenes. Theoretically, the BPH even increases the basic traversal operations by a factor of two compared to a standard BVH that clips against each child's box.

Treelets introduce a technique of clustering nodes that allows for a new way of traversal which operates completely independent on subsets of the index structure. This is perfectly suited for multi-core architectures like the CELL where several subsets can be traversed in parallel. Additionally, Treelets are not restricted to the BPH and can easily be applied to most acceleration structures.

Both algorithms were shown to allow for efficient implementation using packet tracing with SIMD-instructions, although there are several things left for optimization (see Section 8) and it still has to be shown how Treelets perform in an appropriate environment.

The BPH achieves a major improvement of the memory requirement of spatial index structures that works towards bridging the "memory gap" for Realtime Ray Tracing, Treelets are capable of enhancing memory efficiency even more on multi-core architectures. This leads to the conclusion that this thesis's contributions are very likely to increase performance in the future as the additional computational complexity will become negligible while the enhanced memory efficiency will have a strong, increasing impact.

Chapter 8
Future Work

As discussed in Section 6, There are several things that are worth of implementation or further research, as will be summed up in the following.

8.1 BPH Optimization

The Bounding Plane Hierarchy is currently lacking some of the possible optimizations that have proven to result in major performance increases for BVH implementations, most importantly the "early hit test" and conservative frustum culling for an "early miss test" as described in Section 6.3. While frustum culling is directly convertible to the BPH, the early hit test is more complicated.

8.1.1 Early Hit Test

It is not possible to implement the early hit test the same way as done in a standard BVH because the hit-distances of each ray with the parent box are needed when intersecting a child using a slabs-like algorithm. If only the first ray's intersection was computed in a previous traversal step, the necessary information for a ray-by-ray intersection is no more available in subsequent steps. Nevertheless, the test can be done differently, although with a little more overhead:

For adaptation to the BPH, it is necessary to keep track of the complete current box that is traversed with the help of the stack. This means that a box is kept on the stack that is updated in every traversal step with the planes of the corresponding child. If an early hit test was successful, a flag is set, indicating that no complete information of the exact hit-distances

of all rays is available and all further individual tests are skipped. If in a subsequent step all rays have to be tested sequentially, this flag lets the algorithm first perform a full packet-box intersection with the box of each child (reconstructed using the box from the stack) in order to gain complete information again. If the flag is not set, the algorithm performs the standard optimized intersection of both children at once.

The worst-case scenario for this algorithm are alternating early-hit- and reconstruct-cases resulting in two full box-intersections every two traversal steps - which amortizes to the same cost as when not using the early hit test. In any other case the intersection of many rays of the packet is outperforming the computational cost of one reconstruction at a lower level by far. See Figure 8.1 for a graphical illustration.

8.1.2 Rebuilds During Rendering

As discussed in Sections 6.1 and 6.2, no rebuilding of the structure is currently implemented, which limits the variety of dynamic scenes that can be handled efficiently. Adapting e.g. the technique of [LYTM06] should be fairly straightforward.

8.1.3 Surface Area Heuristic (SAH)

The building phase of the current implementation only uses a simple median split strategy that is very likely to produce inferior topologies compared to using a "Surface Area Heuristic"-based partitioning scheme as discussed in Section 6.4. Implementation of the SAH is not too complicated and can increase performance for non-symmetric scenes significantly.

8.1.4 Arbitrary Primitive-Lists

In the current implementation, the number of primitives per leaf is globally fixed (see Section 3.2). This restriction can be relaxed if using additional integer fields for each node that indicate the length of the list of primitives that is referenced by a leaf pointer. If applying the intermediate-node-optimization, numbers of both lists of the children just have to be added as the primitives lie subsequently in memory. This possibly allows for better optimized index structures that e.g. heuristically assign several primitives to certain leaf nodes. On the other hand, the desired integer would destroy the careful memory-layout and thus probably worsen overall performance.

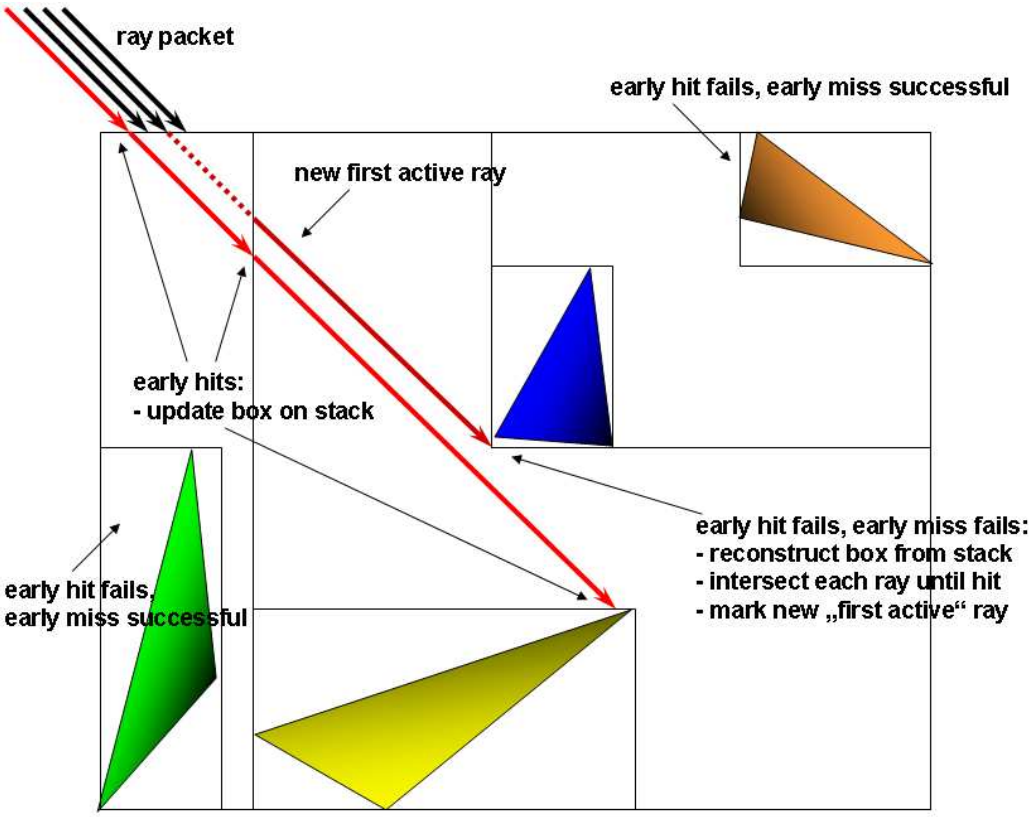

Figure 8.1: Example of the traversal-algorithm using both an "early hit test" and frustum culling ("early miss"). The current box has to be kept on the stack and has to be updated in each step to be able to reconstruct it at lower levels. In this case, 7 early hit tests are performed (one for each AABB), 4 are successful. For the remaining cases, two frustum tests succeed (no ray hits) and one fails. Only for this one case, a reconstruction and individual testing of all rays has to be performed. The cost of 4 cheap early hit tests is very low compared to the spared individual ray-box-intersections.

8.2 Treelet Optimization

There are also various ways worth trying to improve Treelets:

8.2.1 Cache-Oblivious Treelets

Most obviously using some heuristic in the building phase instead of the simple "breadth-first" to determine what child to add next to the current Treelet could again increase cache-efficiency during runtime. A specific metric that e.g. always adds the child with the largest volume or that computes a cache-oblivious layout as presented in [YLPM05] could be worth applying.

8.2.2 Enforcing Filled Treelets

The algorithm described in Section 4.1 simply constructs Treelets top-down beginning with the root of the tree, often resulting in many Treelets that can not be filled completely for branches that do not fit exactly. Although this does not affect memory requirement due to the clever reordering strategy, it might be desirable to have most or even all Treelets consist of the same amount of inner nodes.

There are two possible solutions for this problem that are obvious. The first one is a straightforward extension to the current algorithm that counts the remaining inner nodes of all branches before deciding what node to add to the current Treelet. This could also be implemented by maintaining the remaining inner nodes of each branch that has not yet been visited and by enforcing these values to be of multiples of the chosen Treelet-size when completing a Treelet. Possibly this could also be a part of a metric as in Section 8.2.1 that is used to decide what node to add next.

The other possibility is a bottom-up construction algorithm that does not result in many unfilled Treelets on the lowest level, but only in a few unfilled Treelets high up in the structure. However, this task is quite challenging because neighbouring paths are not known when ascending in the tree, resulting in situations where it is unclear if the algorithm should go down some neighbouring path or ascend further in order to add the missing nodes to complete the Treelet. If ascending further, the neighbouring path is likely not to consist of a multiple of the Treelet-size and thus will not result in filled Treelets. Adding nodes from that path on the other hand might affect a subtree that would have fit exactly otherwise.

8.2.3 Finding Optimal Treelet-Size

Currently, the user has to decide for a Treelet-size. This decision could be handled automatically by the algorithm in various ways, possibly resulting in less overhead and an optimized Treelet-structure, also considering the above mentioned desire for completely filled Treelets. Random or incremental construction of several sizes (chosen e.g. in dependence of the size of the scene) and using a metric that decides the best one by measuring the number of incomplete Treelets, the total number of empty node-slots and their ratio against the number of complete Treelets would be a straightforward solution. One could also develop a heuristic that analyses the tree and based on that selects a size. The average path-length of the tree would be an option for obtaining such a metric, in comparison with the number of leafs this could give a useful estimation of the branching factor.

8.2.4 Improved Traversal Order

Traversal is unordered right now, which means nodes that lie outside the current Treelet are pushed onto the stack in the order they are encountered and thus traversed in the same order. It is very well possible that some ordering after the hit-distances of the ray-packet could result in faster traversal despite the computational overhead. A different data structure, e.g. a heap that always has the nearest child in its root could also be used instead of the stack. As mentioned in Section 4.3, this job could be very well suited for the PPE that is only responsible for scheduling most of its time and possibly has resources left while the SPEs work on Treelets.

Acknowledgement

The massive models (bunny, dragon, buddha) are part of the Stanford 3D Scanning Repository.
http://graphics.stanford.edu/data/3Dscanrep/

All animated models were taken from the Utah 3D Animation Repository.
http://www.sci.utah.edu/w̃ald/animrep/

The shuttle is courtesy of Viewpoint Datalabs International, Inc., Copyright 1996.
http://people.scs.fsu.edu/b̃urkardt/data/obj/obj.html

The simple "ground.obj" is part of an assignment of the Computer Graphics lecture at Saarland University.
http://graphics.cs.uni-sb.de/Courses/ws0506/cg1/index.html

Bibliography

[AH95] Stephen J. Adelson and Larry F. Hodges. Generating Exact Ray-Traced Animation Frames by Reprojection. *IEEE Comput. Graph. Appl.*, 15(3), 1995.

[Bas91] F. Baskett. Keynote address. *International Symposium on Shared Memory Multiprocessing*, April 1991.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[BSGM02] William V. Baxter, Avneesh Sud, Naga K. Govindaraju, and Dinesh Manocha. Gigawalk: interactive walkthrough of complex environments. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, pages 203–214, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[BWSF] Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedrich. Ray Tracing on the CELL Processor. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 15–23.

[Cla76] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554, 1976.

[DWS04] Andreas Dietrich, Ingo Wald, and Philipp Slusallek. Interactive Visualization of Exceptionally Complex Industrial Datasets. In *ACM SIGGRAPH 2004, Sketches and Applications*, August 2004.

[DWS05] Andreas Dietrich, Ingo Wald, and Philipp Slusallek. Large-Scale CAD Model Visualization on a Scalable Shared-Memory Architecture. In Günther Greiner, Joachim Hornegger, Heinrich Niemann, and Marc Stamminger, editors, *Proceedings of 10th International Fall Workshop - Vision, Modeling, and Visualiza-*

tion (VMV) 2005, pages 303–310, Erlangen, Germany, November 2005. Akademische Verlagsgesellschaft Aka.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285, Washington, DC, USA, 1999. IEEE Computer Society.

[GS87] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20, 1987.

[HHS06] Vlastimil Havran, Robert Herzog, and Hans-Peter Seidel. On the fast construction of spatial data structures for ray tracing. In Ingo Wald and Steven G. Parker, editors, *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, pages 71–80, Sep 2006.

[HP03] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[KK86] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 269–278, New York, NY, USA, 1986. ACM Press.

[LAM03] Thomas Larsson and Tomas Akenine-Möller. Strategies for bounding volume hierarchy updates for ray tracing of deformable models. Technical report, Feb 2003.

[LYTM06] Christian Lauterbach, Sung-Eui Yoon, David Tuft, and Dinesh Manocha. Rt-deform: Interactive ray tracing of dynamic scenes using bvhs. In *IEEE Symp. on Interactive Ray Tracing*, 2006.

[MFT05] B. Minor, G. Fossum, and V. To. TRE: Cell Broadband Optimized Real-Time Ray-Caster. In *Proceedings of GPSx*, 2005.

[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. (Proceedings of Eurographics), to appear.

[PMS+99] Steven Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen. Interactive Ray Tracing. In *Symposium on Interactive 3D Graphics*, pages 119–126, 1999.

[Pur04] Timothy J. Purcell. Ray Tracing on a Stream Processor. *PhD dissertation, Stanford University*, March 2004.

[RW80] Steven M. Rubin and Turner Whitted. A 3-dimensional representation for fast rendering of complex scenes. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 110–116, New York, NY, USA, 1980. ACM Press.

[SWS02] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. Saarcor – a hardware architecture for ray tracing. In *Proceedings of the conference on Graphics Hardware 2002*, pages 27–36. Saarland University, Eurographics Association, 2002. available at http://www.openrt.de.

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

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1), 2007.

[WBWS01] Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive Rendering with Coherent Ray Tracing. In Alan Chalmers and Theresa-Marie Rhyne, editors, *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, volume 20. Blackwell Publishers, Oxford, 2001. available at http://graphics.cs.uni-sb.de/w̃ald/Publications.

[WH06] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in O(n log n). In Ingo Wald and Steven G. Parker, editors, *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, pages 61–69, Sep 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. In *Proceedings of Graphics Hardware*, pages 67–77, 2006.

[Woo06] Sven Woop. *DRPU: A Programmable Hardware Architecture for Real-time Ray Tracing ofCoherent Dynamic Scenes*. PhD thesis, 2006.

[WSS05] Sven Woop, Jörg Schmittler, and Philipp Slusallek. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. In *Proceedings of ACM SIGGRAPH 2005*, July 2005.

[YLPM05] Sung-Eui Yoon, Peter Lindstrom, Valerio Pascucci, and Dinesh Manocha. Cache-oblivious mesh layouts. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 886–893, New York, NY, USA, 2005. ACM Press.

[YM06] Sung-Eui Yoon and Dinesh Manocha. Cache-efficient layouts of bounding volume hierarchies. In *Computer Graphics Forum (Eurographics), volume 25, issue 3*, 2006.