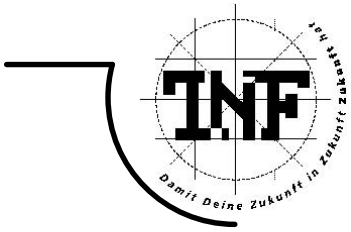
JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis



Real–time Ray Tracing of Dynamic Scenes

Diplomarbeit

zur Erlangung des akademischen Grades

Diplomingenieur

in der Studienrichtung

Informatik

Angefertigt am *Institut für Parallele und Graphische Datenverarbeitung*

Betreuung:

o. Univ.–Prof. Dr. Jens Volkert

Eingereicht von:

Stephan Reiter, Bakk. techn.

Mitbetreuung:

Dipl.–Ing. Paul Heinzlreiter

Linz, Juni 2008

Johannes Kepler Universität
A-4040 Linz · Altenbergerstraße 69 · Internet: http://www.uni-linz.ac.at · DVR 0093696

Abstract

In this thesis ray tracing of dynamic scenes in real–time is explored based on a separation of static from animated primitives in acceleration structures suited for each type of geometry.

For dynamic geometry a two–level bounding volume hierarchy (BVH) is introduced that efficiently supports rigidly animated geometry, deformable geometry and fully dynamic geometry with incoherent motion and topology changes. With selective rebuilding an updating technique for BVHs is described that limits costly rebuilding operations to degenerated parts of the hierarchy and allows for balancing updating and rendering times. Furthermore a new ordered traversal scheme for BVHs is introduced that is based on a probabilistic model.

Kd–trees are the acceleration structure of choice for static geometry and are commonly built by employing the surface area heuristic to determine optimal splitting planes. In this thesis two approaches for reducing the memory footprint of kd–trees are presented. Index list compaction compresses the list of triangle indices used by leaves to reference triangles. The cost–scaling termination criterion for kd–tree construction, on the other hand, limits the creation of deep trees by weighing the costs of splitting a node higher with an increasing depth.

Kurzfassung

Im Rahmen dieser Diplomarbeit wird die Umsetzbarkeit von Ray–Tracing dynamischer Szenen in Echtzeit untersucht, basierend auf einer Trennung statischer von animierten Objekten in jeweils geeigneten Beschleunigungsstrukturen.

Für dynamische Geometrie wird eine zweistufige Hüllkörperhierarchie (BVH) vorgestellt, die Unterstützung für starr animierte Geometrie, deformierbare Geometrie und komplett dynamische Geometrie mit inkohärenter Bewegung und Topologieänderungen aufweist. Mit der Selektiven Neukonstruktion wird eine Aktualisierungsmethode beschrieben, die aufwendige Konstruktionsschritte auf degenerierte Teile einer Hierarchie beschränkt und eine Balanzierung von Aktualisierungs– und Darstellungszeiten erlaubt. Außerdem wird ein neuer Ansatz für die räumlich geordnete Traversierung von BVHs vorgestellt, der auf einem Modell der Wahrscheinlichkeitsrechnung basiert.

Kd–Bäume sind die die Beschleunigungsstruktur der Wahl für statische Geometrie und werden im Allgemeinen unter Verwendung der sogenannten „Surface Area Heuristic“ für die Bestimmung optimaler Teilungsebenen konstruiert. In dieser Diplomarbeit werden zwei neue Ansätze für die Reduzierung des Speicherbedarfs von Kd–Bäumen präsentiert. Die Index–Listen Verdichtung komprimiert die Liste der Dreiecksindizes, welche in Blättern für Verweise auf Dreiecke verwendet wird. Das kostenskalierende Abbruchkriterium für die Konstruktion von Kd–Bäumen hingegen beschränkt die Erstellung tiefer Bäumen durch Erhöhung der Teilungskosten von Knoten mit zunehmender Tiefe.

Danksagung

Vor allen anderen möchte ich mich an dieser Stelle ganz herzlich bei meinen Eltern bedanken, die mir ein sorgenfreies Studium ermöglicht haben und in all dieser Zeit immer für mich da gewesen sind und mich unterstützt haben. Auch meinem Bruder bin ich zu Dank verpflichtet, der mit mir Zeit bei der empirischen Forschung im Bereich der Computergrafik verbracht hat und auch Ideen zur Umsetzung von Projekten in der Vergangenheit beigesteuert hat.

Mein besonderer Dank gilt auch Prof. Volkert für die Betreuung meiner Diplomarbeit und für die stets konstruktive und gut gemeinte Kritik, die zum Fortschritt des Projekts in die richtige Richtung beigetragen und über viele Monate hinweg motivierend gewirkt hat.

Bei Paul Heinzlreiter möchte ich mich herzlich für die Unterstützung bei der Bearbeitung des gewählten Themas bedanken. Er hat mir die Freiheit gelassen, eigene Wege in der Forschung zu gehen, stand aber gleichzeitig immer mit Rat zur weiteren Vorgangsweise bereit. Seine kurzfristige Erreichbarkeit hat die Erstellung dieser Arbeit immens beschleunigt, was mir ein großes Anliegen war.

Auch möchte ich mich bei Gerhard Kurka bedanken, der meine Begeisterung für Computergrafik im Laufe meines Studiums nur noch weiter angefacht hat. Mit der Betreuung meiner Bakkalaureatsarbeit, die ebenfalls das Thema Ray–Tracing behandelt hat, hat er beim Legen eines wichtigen Grundsteins für diese Diplomarbeit mitgewirkt.

Weiterer Dank gilt der Community von *ompf.org*, die eine wichtige Quelle von praktischen Hinweisen zur Umsetzung von Echtzeit–Ray-Tracing war. Außerdem möchte ich mich bei den Mitgliedern des *ioquake3*–Projekts bedanken, allen voran Thilo Schulz und Zachary Slater, die die Entwicklung eines wichtigen Projekts im Rahmen dieser Arbeit ermöglicht und unterstützt haben.

Danke!

Linz, Juni 2008 *Stephan Reiter*

Contents

Chapter 1

Introduction

Ray tracing has been pursued by academia over the last two decades as a technique for image synthesis considered by many to be superior to other approaches, such as rasterization. Tracing the paths of light in virtual scenes is both an elegant and flexible solution to the problem of rendering images and can typically be implemented on computers with little effort, which contributed to the popularity of ray tracing. However, until recently the practical use of the technique remained restricted to non–interactive scenarios due to its high computational requirements, which typically resulted in per–frame rendering times of a few minutes to many hours on the hardware of that time.

It was not until the beginning of the 21^{st} century that hardware became fast enough to support ray tracing at interactive frame rates, which was demonstrated with great success by Ingo Wald in his PhD thesis [54]. Since then a strong interest in employing ray tracing in interactive and real-time applications developed with the ultimate goal of replacing or augmenting rasterization–based solutions, which is aided by current trends in hardware design that deliver processors suited to exploiting the parallel nature of the ray tracing algorithm.

In this thesis ray tracing of dynamic scenes in real–time is explored, which is particularly challenging in that moving geometry has to be managed in a so–called acceleration structure to achieve the necessary low rendering times. This problem is approached by separating static from dynamic geometry and by using data structures suited for each type of geometry. In particular a flexible acceleration structure for dynamic primitives is presented, which efficiently supports different types of animation.

Run–time performance of an implementation of the described concepts and algorithms is evaluated by employing the well–known "Benchmark for Animated Ray Tracing" by Lext *et al.* [32]. Furthermore the applicability of ray tracing to games is demonstrated in that an existing game is refitted to ray tracing based rendering, which allows the introduction of new graphical effects that were not possible with rasterization–based rendering used in the original version.

Chapter 2

Background

Ray tracing is a conceptually simple rendering technique that has been used with great success in non–realtime rendering applications. However, in order to be able to render reasonably complex scenes at interactive frame rates, it is necessary to reduce the run–time costs of tracing rays. This chapter presents a survey of different approaches to this problem, which serve as a base for the real–time ray tracing system developed as part of this thesis.

2.1 Fast Intersection Tests

Testing rays for intersection with the geometry of a scene is at the heart of each implementation of the ray tracing technique. It is therefore desirable to optimize this operation in order to reduce the overall rendering time. Different algorithms have been proposed to find potential intersections of a given ray with geometric primitives of various types such as spheres [18], implicit surfaces [26] or free–form surfaces [4].

Although ray tracing is in particular known for its large support of different primitive types, it is possible to achieve better run–time performance by limiting the feature set of an implementation to only a single primitive type. This allows for an elimination of branches and yields a tighter inner loop that can be executed faster by current processors.

Triangles are often used as the lowest common denominator in representation of geometry. They are supported directly by most modeling applications but can also be generated through the process of tessellation to serve as a good approximation of higher–order surfaces. Being ubiquitous in implementations of rasterization, triangles can also be considered a good choice for real–time applications based on ray tracing.

A variety of algorithms has been devised to reduce the computational effort of testing a ray for intersection with a triangle. In addition to information about whether an intersection point exists or not, the results delivered by these algorithms usually also contain the barycentric coordinates of the hit point. Commonly named α, β and γ, these coordinates define a point on the surface of a triangle as the weighted sum of its three vertices. Figure 2.1 illustrates their relationship. Barycentric coordinates are used for interpolation of per–vertex attributes such as normal vectors and texture coordinates and therefore play an important role in shading. It is only natural that many algorithms addressing the problem

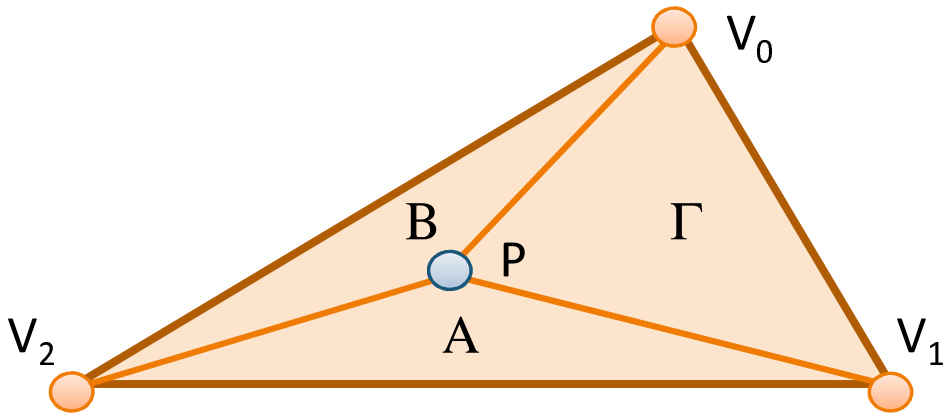


Figure 2.1: *The point P on the surface of the triangle is defined as the weighted sum of its barycentric coordinates α, β and γ and the triangle's vertices. A, B and Γ are the areas corresponding to α, β and γ.*

of intersecting rays with triangles compute α, β and γ in the process in order to save on additional operations that would have to be carried out afterwards to calculate these factors.

Wächter [60] gives an in–depth analysis of ray–triangle intersection tests with regard to speed and numerical precision, covering algorithms like Arenberg [1], Badouel [3], Plücker [25] and Möller–Trumbore [37]. He concludes that a variant of the barycentric coordinate test can be considered the most attractive in a general context because it executes fast and exhibits good numerical precision. Barycentric coordinate tests first compute the intersection of a ray with the plane going through a triangle's three vertices. Subsequent tests determine if the calculated hit point lies on the triangle's surface and can therefore be considered valid. This is performed by checking if the barycentric coordinates describing the point sum to one.

A fast implementation is given in [28]. What is interesting about it in particular is the fact that it was devised by means of automated search. Kensler and Shirely proposed the use of a genetic algorithm to find the most efficient ray–triangle test with regard to run–time. After establishing a set of rules that would guarantee the correct order of calculations, they spawned a large number of variations of the general procedure to find a hit point. A fitness function based on the measured run–time would then be used to select the best tests and use them in the subsequent "generation". By introducing new variations into each generation by use of techniques known from genetic or evolutionary programming and by repeating the selection step the result was refined iteratively.

2.2 Acceleration Structures

Rendering images with ray tracing requires processing a large number of rays even for moderate image resolutions. Testing each ray for intersection with all primitives in order to find the nearest point of intersection with the geometry is clearly too slow for non–trivial scenes. A common approach to reducing the run–time costs associated with tracing rays is to use some kind of acceleration structure. Similar to sorted data arrays that enable the use of searching algorithms with sub–linear asymptotical behavior, acceleration structures can reduce the asymptotical run–time of tracing a ray from $O(n)$ to $O(\log n)$.

With the use of acceleration structures a trade–off has to be made between construction times and the possible gains in ray tracing performance. The goal is a minimization of the overall frame rendering time. When rendering static scenes, construction is typically performed once prior to image synthesis. Per–frame times are therefore only affected by how well the acceleration structure reduces the number of ray–triangle intersection tests. When dealing with dynamic geometry, the situation becomes more difficult. Primitives that move or change shape invalidate acceleration structures, which therefore have to be updated to reflect the new scene configuration. In most applications this has to be performed every frame. Therefore care needs to be taken to ensure that the reduction of ray tracing time is greater than the additional time spent in maintaining an acceleration structure.

2.2.1 Types of Partitioning Schemes

There are two basic types of partitioning: Spatial partitioning subdivides space into disjoint volumes. References are stored for primitives that overlap at least partially with a volume. Although each point in space is represented uniquely, primitives may overlap multiple volumes and can therefore be referenced more than just once in the acceleration structure. Object list partitioning schemes distribute all primitives of a scene into disjoint sets. Primitives are therefore referenced exactly once. A point in space on the other hand may be covered by more than a single entry structure or is potentially not represented at all in the acceleration if no primitives overlap with it. A number of advantages and disadvantages go along with these inherent properties of partitioning schemes.

Acceleration structures based on object list partitioning have the benefit of enabling à priori allocation of memory based on the number of primitives because these primitives are referenced exactly once. Spatial subdivision requires the use of dynamic memory management due to the potential duplication of references. This can introduce a serious run–time overhead and result in fragmentation of memory if changing scene configurations require regular rebuilds or updates. Duplication of references may also increase the raw memory requirements. Additional memory usage stems from the fact that empty space has to be represented explicitly. This is not the case with object list partitioning schemes that skip empty space implicitly by dealing with primitives directly.

When it comes to traversal, spatial acceleration structures are usually more efficient than their counterparts. Their nature of representing space as disjoint volumes enables traversal in a strict front–to–back order. This allows algorithms, which search for the nearest intersection point of a ray with the scene's geometry, to terminate upon detection of the first hit because all subsequently detected intersections would be farther away. Object list partitioning schemes do not exhibit this property because they allow overlaps of space. When a hit point is found traversal of the acceleration structure has to be continued in the typical case because it is not possible to guarantee that no closer intersections exist.

So–called mailboxes are a concept that applies to spatial acceleration structures only. They are used to record whether a primitive has already been intersected with a given ray or not to avoid having to perform the same test again. This may occur when traversing the acceleration structure because it possibly contains more than one reference to a single primitive. Acceleration structures based on object partitioning are not subject to this behavior.

2.2.2 Adaptive vs. Uniform Structures

In addition to the classification based on the employed partitioning scheme, acceleration structures can also be described as being either adaptive or uniform.

Adaptive acceleration structures are typically defined as trees. Inner nodes partition primitives into two or more sets selected by a construction heuristic. Leaf nodes store a list containing references to primitives. Numerous different heuristics have been proposed, which all aim to improve ray tracing performance by achieving good sorting of the primitives of a scene. Kd–trees [9] are an example for adaptive spatial acceleration structures. Bounding volume hierarchies [45], bounding interval hierarchies [52] and bounded kd–trees [62] are instances of adaptive object list partitioning schemes.

In contrast to adaptive structures, uniform acceleration structures have a rigid structure. Although heuristics may be used to decide when to stop subdivision, e.g. by limiting the minimal number of primitives per node, the selection of partitions is dictated by the acceleration structure itself. Spatial acceleration structures like the grid [7] or the octree [16] fall into this category.

Adaptive acceleration structures are typically the type of choice in most applications. They are more flexible and have the potential for better partitioning of geometry, which in turn reduces the time spent tracing rays. It is, however, important to note that grids have been used with great success in interactive applications [56]. They are easy to update and can be built fast [23], which may make up for the generally lower ray tracing performance in certain scenarios.

2.2.3 Construction of Adaptive Acceleration Structures

Construction algorithms can be classified as either top–down or bottom–up approaches. Top–down construction is performed by recursively partitioning a set of primitives until a termination criterion is met, which results in a tree–like structure. Bottom–up construction groups primitives together and arranges them hierarchically. It has the disadvantage of operating on a local view, whereas top–down algorithms operate on the complete set of primitives and may therefore use additional information to build better acceleration structures.

Wächter [60] presents the steps of a general recursive procedure that constructs a spatial acceleration structure in top–down manner. These steps are further generalized to acceleration structures based on both spatial and object list partitioning schemes in the following.

Termination: A termination criterion is a combination of clauses that limit the depth of the constructed tree. Although deeper trees allow for finer partitioning of primitives, they require more memory and lead to higher traversal costs, which may offset the gains of fewer ray–triangle intersection tests. A termination criterion is also used to enforce pre–conditions of the construction algorithm, i.e. that a minimal number of primitives is provided.

Partitioning: In each step of the construction process the current set of primitives is partitioned into two or more sets based on a heuristic. Whether this results in disjoint sets or not depends on the type of the acceleration structure. The heuristic employed is the key to achieving both minimal construction times and optimal ray tracing performance. Numerous different heuristics have been proposed, which take into account different aspects of

partitioning primitives. Simple heuristics may evaluate information only about the current volume. More complex solutions like the surface area heuristic [33] also take into account the distribution of the primitives and are typically based on a cost model.

Node creation: After partitioning primitives into two or more sets an equal number of nodes are allocated. They are stored as children of the current node and initialized by recursively invoking the construction algorithm.

Post processing: Optimizations such as reordering of nodes for better caching efficiency can be applied at the end of the construction process [63].

2.2.4 The Surface Area Heuristic

The surface area heuristic (SAH) is based on a probabilistic model and is commonly used to guide construction of acceleration structures. It was introduced in [33] for use with kd–trees based upon the work of Goldsmith and Salmon, which had previously used a similar model for construction of bounding volume hierarchies [10].

The SAH tries to find the optimal placement of a splitting plane by minimizing a cost function. This function attempts to describe the costs associated with the traversal of an acceleration structure in ray tracing. Its definition is recursive and reflects the hierarchy of nodes in acceleration structures it is applied to.

When visiting a leaf all contained geometric primitives have to be tested for intersection with the ray. The costs of the visit are therefore proportional to the number of primitives. Equation 2.1 expresses these costs under the assumption that intersection tests have an average cost of $C_{intersect}$.

$$C(N) = C_{intersect} * |N| \tag{2.1}$$

For inner nodes the cost function corresponds to the sum of the costs of both sub–trees weighted by the respective probability of traversal. An additional constant factor $C_{traversal}$ takes into account the time required by the intersection test between a ray and the node itself. The cost function for inner nodes is given in Equation 2.2 with N_l and N_r corresponding to the number of primitives in the left and the right sub–tree.

$$C(N) = C_{traversal} + p_l * C(N_l) + p_r * C(N_r) \tag{2.2}$$

The weighting factors p_l and p_r are based on observations from geometric probability: Given a uniform distribution of rays the probability that a node has to be traversed is equal to the ratio of the surface area (SA) of its bounding box to the SA of the root node. By substituting the parent node for the root node the probability of traversal is expressed relatively to the parent and can be used in the recursive cost function. The complete defintion of the cost function is given in Equation 2.3.

$$C(N) = \begin{cases} C_{traversal} + p_l * C(N_l) + p_r * C(N_r) & \text{for inner nodes} \\ C_{intersect} * |N| & \text{for leaf nodes} \end{cases} \tag{2.3}$$

Due to its recursive nature the cost function can only be applied to complete (sub–) trees. For partitioning schemes where the SAH is used to guide construction in top–down order, it is therefore necessary to evaluate the cost function greedily because the sub–tree of the processed node has not been constructed yet. This is approached by assuming that the children of the node will not be subdivided further but will remain leaves in the tree, yielding Equation 2.4.

$$C_{greedy}(N) = C_{traversal} + p_l * C_{intersect} * |N_l| + p_r * C_{intersect} * |N_r| \tag{2.4}$$

Although the SAH and this type of evaluation in particular are crude approximations not taking into account occlusion of geometry and early–out scenarios in traversal algorithms, Havran [15] demonstrated that the SAH yields better results than other partitioning heuristics for kd–trees.

The SAH can also be used to implement an automatic termination criterion (ATC) [17] to stop subdivision of a node when no gain can be expected from a deeper tree. This decision is usually based on a comparison of the costs of the best split and the costs of turning the node into a leaf (Equation 2.5). Additional factors such as the maximal tree depth or the minimal number of primitives per leaf are commonly used in addition to the SAH–ATC.

$$Terminate(N) = \begin{cases} true & \min C(N) > |N| * C_{intersect} \\ false & \text{otherwise} \end{cases} \tag{2.5}$$

2.3 Ray Tracing Static Geometry

Static geometry encompasses primitives that are fixed in position, orientation and size. They make up large parts of architectural scenes both in offline rendering and real–time applications. Given their static nature it is possible to organize these primitives in highly optimized acceleration structures in a preprocessing step prior to rendering.

Kd–trees are considered state of the art by most researchers today. They adapt well to complex scenes containing primitives of differing size and large empty spaces. Furthermore the surface area heuristic allows cost–driven construction and requires no per–scene tweaking unlike other proposed heuristics [15].

2.3.1 Construction of Kd–Trees

A kd–tree is a binary spatial acceleration structure that partitions space into disjunctive volumes by recursively splitting it with axis–aligned planes. These planes can be positioned arbitrarily inside a given volume, which is the base of the great flexibility of this acceleration structure. Algorithm 1 provides a general outline of recursive kd–tree construction.

Leaves in a kd–tree contain a list of references to primitives that overlap with the volume represented by the node. Due to the fact that kd–trees are a spatial partitioning data structure, primitives are not necessarily assigned to a single leaf but may be referenced by multiple nodes if they intersect with splitting planes, as can be seen in Figure 2.2.

A top–down construction algorithm with a runtime complexity of $O(n \log n)$ was proposed in [55]. It uses a spatially sorted per–axis list of events that mark the opening and ending of a primitive with regard to an axis. Primitives that are perpendicular to a given axis are

Algorithm 1 Recursive kd–tree construction

```
function PARTITION(triangles T, voxel V) returns node
    if Terminate(T,V) then
        return new leaf node(T)
    p = FindPlane(T,V) /* find a plane p to split V */
    (V_L, V_R) = Split V with p
    T_L = {t ∈ T|(t ∩ V_L) ≠ ∅}
    T_R = {t ∈ T|(t ∩ V_R) ≠ ∅}
    return new node(p, Partition(T_L,V_L), Partition(T_R,V_R))
function BUILDKDTREE(triangles[] T) returns root node
    V = AABB(T) /* start with the full scene */
    return Build(T,V)
```

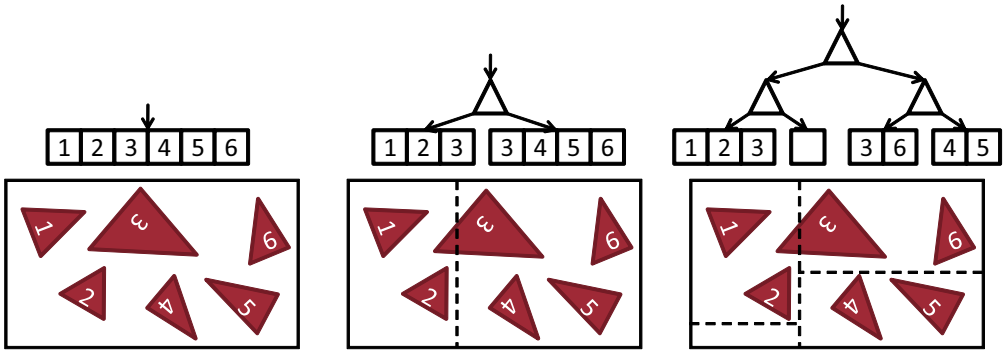


Figure 2.2: *Kd–trees partition space into disjunctive volumes with axis–aligned planes.*

represented by a single event because they have no extent in that axis and need to be handled differently.

When splitting a node the SAH–based costs for all potential splitting planes are evaluated by "sweeping" over the positions denoted by the events and keeping track of the number of primitives on both sides. Algorithm 2 illustrates this with pseudo code. It suffices to evaluate these specific positions because the cost function will never have lower values in between, due to the increase of costs of both sub–trees when primitives straddle the splitting plane. Finally the plane with minimum costs is selected to split the node into two volumes. The event lists are also partitioned into two sub–lists keeping the events' sorting order intact. Events associated with primitives straddling the splitting plane are recreated for both volumes based on clipped representations of the primitives. This guarantees correct positions within the volumes of the new nodes and enables splitting based on the minimal bounds of the primitives, which yields trees of higher quality. These new events are sorted and then inserted into the existing event lists using a single merge–sort iteration resulting in sorted event lists for the construction of the nodes at the next level of the hierarchy.

A termination criterion based on the SAH is used to decide when to stop subdivision of a given node. Allowing additional construction steps with potential backtracking has proven to improve the quality of kd–trees because the employed greedy heuristic is subject to terminating too early by not accounting for further splits, which may results in a deeper tree with total smaller costs.

Algorithm 2 SAH plane sweep

```
function FINDPLANE(triangles T, voxel V, events E) returns plane
    Ĉ = ∞, p̂ = ∅ /* initialize search */
    /* consider all K dimensions in turn */
    for k = (x,y,z) do
        /* sweep plane over all split candidates */
        N_L = 0, N_p = 0, N_R = |T| /* initialize counters */
        for i = 0; i < |E| do
            p = (E_{i,p}, k), p⁺ = p⁻ = p| = 0
            while i < |E| ∧ E_{i,pos} = p_pos ∧ E_{i,type} = − do
                inc p⁻, inc i
            while i < |E| ∧ E_{i,pos} = p_pos ∧ E_{i,type} = | do
                inc p|, inc i
            while i < |E| ∧ E_{i,pos} = p_pos ∧ E_{i,type} = + do
                inc p⁺, inc i
            /* found next plane p with p⁺, p⁻, p| */
            N_p = p|, N_R− = p| + p⁻ /* move plane onto p */
            C = SAH(V, p, N_L, N_p, N_R) /* evaluate SAH */
            if C < Ĉ then
                Ĉ = C, p̂ = p /* found better plane */
            N_p = 0, N_L+ = p| + p⁺ /* move plane over p */
    return p̂
```

2.3.2 Construction Based on SAH–Sampling

The algorithm laid out in [55] depends on geometric primitives to be spatially sorted along each axis. This enables the use of plane sweeping to incrementally compute costs as defined by the SAH for each splitting plane candidate. Although sorting of primitives can be done in $O(n \log n)$, the runtime costs of this operation make the proposed algorithm increasingly less appealing for the construction of kd–trees for scenes with larger primitive counts.

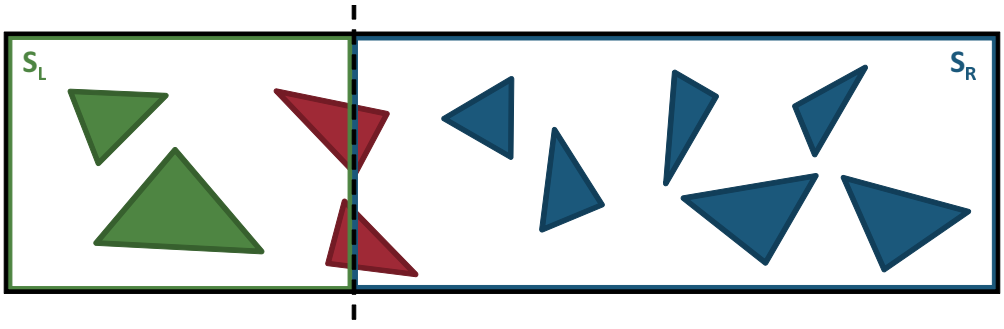


Figure 2.3: *The counters N_L and N_R record the number of primitives on each side of the plane (e.g., $N_L = 4$ and $N_R = 8$ in this example). S_L and S_R are the surface areas of the left and the right bounding box.*

Hunt *et al.* [19] introduce the concept of sampling the SAH cost function. Along each axis the values necessary to calculate the costs are evaluated at evenly spread out positions. The numbers of primitives on both sides of a given plane (N_L and N_R) are determined by classifying each primitive with regard to the splitting plane candidate. Primitives that straddle the plane will be counted for both sides. This operation does not depend on sorted geometry.

Additionally the surface areas of the left and right bounding boxes are evaluated in constant time. Figure 2.3 gives an example of sampling the required values for one splitting plane.

Given these four values (N_L, N_R, S_L, S_R) for a fixed number of candidate splitting planes, a quadratic approximation of the cost function can be generated by interpolating primitive counts and surface areas linearly. In intervals with strong variation a fixed number of additional samples may be taken to improve the quality of the approximation.

Based on the collected data the algorithm determines the optimal position of the splitting plane and continues construction of the sub–trees. When the number of primitives drops below the configured number of samples construction falls back to the sweeping algorithm, which is more efficient under these circumstances.

The sampling approach has a number of advantages over the construction algorithm that relies solely on sweeping:

- Although both algorithms exhibit the asymptotical behavior of $O(n \log n)$, sampling can be implemented more efficiently on current hardware by exploiting vector instruction. It can therefore have smaller run–time constants.
- The presented sampling algorithm defers more work to the construction of the leaves by performing a sort only for relatively small numbers of primitives compared to the upfront sort of all geometry when sweeping only. This is especially important for implementations of lazy construction, which constructs an acceleration structure on demand and may be able to skip entire sub–trees if they are not needed.
- Sampling is more efficient when the data set doesn't fit into the cache completely due to its linear read–only access pattern, which can be predicted well by hardware. Sorting of geometry is expected to be slower under these circumstances.

Published results show that the quality of the generated kd–tree is within about 4% of a tree built using sweeping. Hunt *et al.* [19] also report building times that make per–frame rebuilds of kd–trees for changing geometry practicable.

2.3.3 Ray Traversal of Kd–Trees

Traversal of a kd–tree is based on maintaining a parameter interval that corresponds to the segment along the ray intersecting the volume of the current node. This interval is initially set to the values of the near and far intersection points of the ray with the scene's bounding box. Negative near values are clamped to zero in order to restrict the search for an intersection point to geometry in the viewing direction of the ray. If this operation results in an invalid interval with $t_{near} > t_{far}$ or if no intersection with the scene's bounding box could be determined in the first place, traversal terminates with no hit.

Given a valid interval the ray is intersected with the splitting plane of the root node. The resulting parameter value t_{split} is then used to determine which sub–trees need to be visited next as depicted in Figure 2.4. If the intersection point with the splitting plane is not in the active segment (i.e., $t_{split} < t_{min}$ or $t_{split} > t_{max}$) only one child node is visited. If $t_{min} < t_{split} < t_{max}$ both sub–trees need to be traversed with updated parameter intervals to reflect the ray's intersection with the volume of the next node (i.e. $[t_{min}; t_{split}]$ for the near and $[t_{split}; t_{max}]$ for the far node). In order to guide traversal into finding the nearest hit point in the minimum number of steps the near child is visited first, whereas the far child is pushed onto a stack for later traversal.

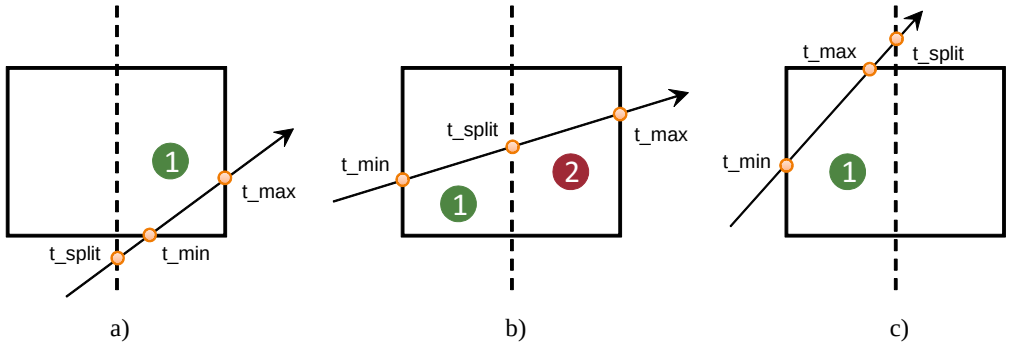


Figure 2.4: *kd–tree traversal is exercised based on three cases: b) Both sub–trees are traversed in near–to–far order if $t_{min} < t_{split} < t_{max}$. a) and c) If the intersection point with the splitting plane is not in the active segment only one child node is visited.*

Upon entering a leaf of the kd–tree the ray is intersected with all referenced primitives. If an intersection point is found that lies in the current interval, traversal can be terminated. This is due to the order in which sub–trees are visited: All further traversal steps would encompass intervals farther down along the ray. Therefore no intersection points closer to the ray's origin can be found. The check for the intersection point being in the active segment can be implemented by comparing its distance to the ray origin with the far–value of the active interval with the less–equal operator. It is a necessity introduced by a specific property of kd–trees and spatial acceleration structures in general: Objects may overlap with the volumes of more than a single node. By detecting an intersection point outside the volume of the current node no guarantee can be made that no closer intersections with other primitives of the scene exist. Figure 2.5 illustrates such a scenario.

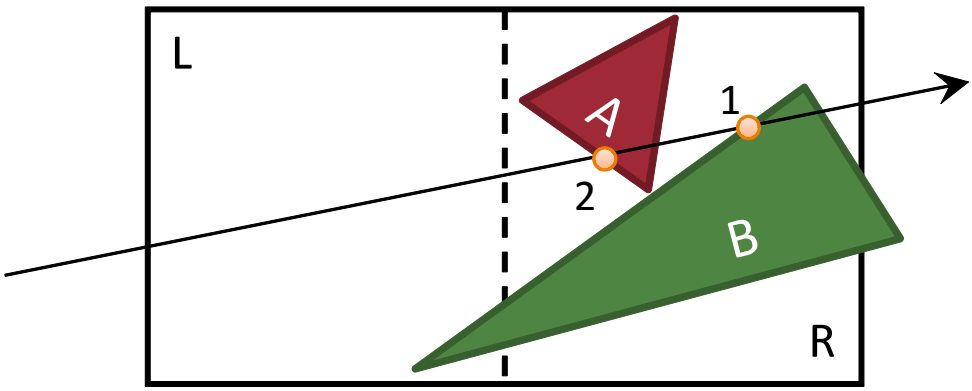


Figure 2.5: *The left node is visited first and an intersection of the ray with triangle B is found (*1*). Traversal must not stop, however, and also visit the right node because* 1 *is not contained in the left node and therefore no valid intersection point.* 2 *is subsequently determined as the nearest hit point.*

This traversal scheme can be implemented directly in a recursive manner. However, it is preferable to choose an iterative implementation, which usually delivers better performance because of the eliminated overhead of recursive function calls. Algorithm 3 gives a pseudo–code sample implementation of iterative kd–tree traversal. Note the presence of an explicit stack used to store information for resuming traversal of the far sub–tree in situations when both children have to be visited.

Algorithm 3 Iterative kd–tree traversal

```
function TRAVERSE(ray r)
    near = 0, far = Infinity
    intersect r with scene AABB and update near, far
    if near > far then
        return
    stack S, curNode = rootNode
    while true do
        while curNodeisnoleaf do
            split = intersect r with curNode's plane
            (nearChild, farChild) = determine order of curNode's children
            if split ≥ near then
                if split ≤ far then
                    S ← (farChild, split, far) /* push far segment onto the stack */
                    far = split /* continue traversal in near segment */
                curNode = nearChild
            else
                curNode = farChild
        curNode is a leaf, intersect r with triangles
        if r.hit.distance ≤ far then
            return /* early out */
        if S not empty then
            (curNode, near, far) ← S /* pop next segment from the stack */
        else
            return
```

2.3.4 Exploiting Ray Coherence for Traversal

The previously discussed algorithm traverses kd–trees for single rays. By aggregating multiple rays into so–called ray packets, which is a concept that was introduced in [54], traversal performance can be improved. When rays exhibits a common set of properties certain calculations may be performed only once and shared among all rays. This is usually exploited when testing a primitive for intersection with rays sharing a common origin, which is particularly the case for primary rays.

Ray packets can allow for further performance gains through the use of vector instructions as provided by the Streaming SIMD Extensions (SSE) on the x86–platform or by AltiVec on PowerPCs. These instructions perform mathematical operations and comparisons on a fixed number of values at a time, which is commonly referred to as the native vector size. By grouping a number of rays, which is a multiple of the native vector size, the per–instruction throughput can be increased and memory accesses may be reduced by sharing scalar data across the fields of operands for vector operations.

For kd–tree traversal with packets as described in [54] the components of the rays' direction vectors need to have the same sign. This is to ensure that the order of traversal with regard to near and far sub–trees is consistent for all rays. Another approach to guaranteeing the same order of traversal among rays is to allow arbitrary directions but force a common origin. Although beneficial for primary rays there is an associated drawback that harms ray tracing performance: For packets of rays with the same principal direction the near and far nodes can be derived from the sign–bits of the direction vectors. A simple XOR–operation suffices to select the correct nodes during traversal. When taking the common–origin route

the order of traversal has to be evaluated in each step based on the spatial arrangement of nodes and origin.

The general outline for the algorithm does not change compared to traversal with a single ray. An active parameter interval is maintained for each ray as traversal is carried out. With each step a decision is made whether to cull a certain sub–tree or to visit both in a near–to–far order. To enable support for multiple rays these decisions are not based on flags but on masks, which capture the outcome of comparisons when working with vector instructions. The traversal algorithm is changed to visit a child if at least a single ray intersects its volume.

Additionally a mask of active rays is maintained and updated with every step. It is used to quickly discard entire ray packets that do not intersect the volume of the current node. Updates to individual rays in a packet, which should only apply to the active elements, are also masked.

When processing nodes on higher levels of the kd–tree, coherent rays will most likely agree on a path through the tree. This is due to the coarse partitioning granularity of the scene close to the root node. With deeper descent into the data structure rays become more divergent and the performance gains of ray packets diminish. Note that the possible reduction of ray tracing time is proportional to the coherence of rays in a packet. Secondary rays, e.g. traced to visualize reflections, are generally less coherent due to scattering over the hemisphere of the surface of an object. Packet tracing is less efficient for these rays.

An additional modification of the original single–ray traversal algorithm affects the early–out test that is performed after the intersection tests with primitives in a leaf. The algorithm keeps track of finished rays that had intersections computed falling into the respective current segment. Only when all rays have been finished the traversal may be terminated early.

2.3.5 Multi Level Ray Tracing

The multi level ray tracing algorithm (MLRTA) as described in [44] is an attempt to further lower the time spent traversing a kd–tree by exploiting additional common properties of multiple rays. It encompasses two new techniques named entry point search and interval traversal algorithm, which will be described in the following.

Entry Point Search

Entry point (EP) search tries to locate a deep starting point for a group of rays in a kd–tree in order to minimize the number of ray traversal steps. First, the kd–tree is traversed in depth–first order until a leaf node is encountered, which becomes the first EP candidate. Along the way a quick rejection test based on a conservative representation of the rays is used to determine if all rays miss a given sub–tree, which can then be skipped. Whenever both children of the current node would have to be visited, it is pushed onto the so–called “bifurcation stack” and only the left sub–tree is traversed further.

In the subsequent phase of the entry point search the right sub–trees of the nodes on the bifurcation stack are traversed. Again, quick rejection tests are used to skip nodes. The traversal of a sub–tree can be aborted when a non–empty leaf node is found, in which case the current bifurcation node becomes the new EP candidate. If only empty leaves are encountered the current bifurcation node is discarded because ray–triangle intersections would only be found in the left sub–tree where a deeper EP is available.

The final entry point is then used to traverse the kd–tree with the rays to find their nearest points of intersection with the geometry. For coherent rays the EP is typically deep and allows for skipping ray traversal of all nodes above it. Figure 2.6 illustrates the entry point search for a sample kd–tree.

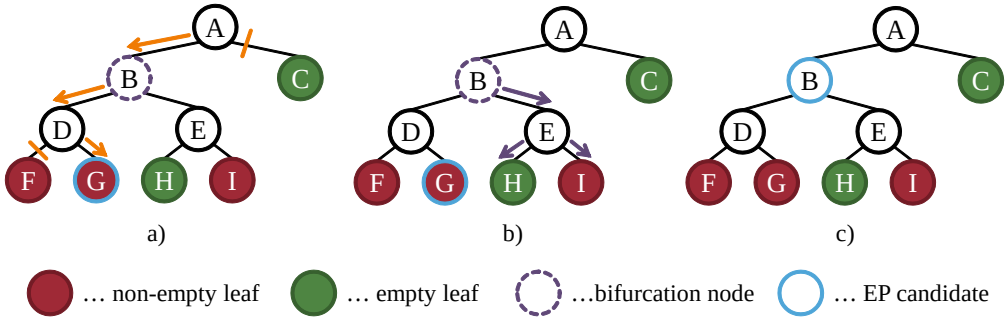


Figure 2.6: *Entry point search is performed in two phases: a) The kd–tree is traversed to collect bifurcation nodes. b) The right sub–tree of each bifurcation node is traversed to determine the top–most EP candidate. c) Normal ray traversal starts at the entry point.*

Reshetov *et al.* propose to use the frustum culling technique [2] for quick rejection tests. In this context the technique operates on a node's axis aligned bounding box (AABB) and a frustum for the rays represented by a set of planes [5]). It is a conservative test and may return false negatives implying that the frustum may intersect an AABB when in fact it does not. The probability of false negatives depends on the relation between the size of the frustum and the AABB, with small AABBs and large frusta yielding better results.

By reversing the roles of the frustum and the node's AABB in the test ("inverse frustum culling") more accurate results and subsequently better performance can be achieved by increasing the number of skipped nodes. This approach is based on the observation that the frustum for a group of rays is usually a lot narrower compared to an AABB of nodes, especially when they are close to the kd–tree's root node where proper culling of sub–trees is most effective.

Interval Traversal Algorithm

The interval traversal algorithm is derived from the previously described packet–based traversal algorithm for kd–trees. However, it differs from it in that it does not maintain an active parameter interval for each ray. Instead a single parameter interval $[t_{min};t_{max}]$ for the entire group of rays is used to decide which nodes need to be visited.

In each traversal step all rays are intersected with the splitting plane of the current node and the minimal and maximal intersection distances ($split_{min}$, $split_{max}$) are computed. Based on the current interval and the intersection distances one of three cases, as depicted in Figure 2.7, is exercised. If $split_{max} < t_{min}$ only the far child needs to be visited. In case $split_{min} > t_{max}$ it is the near sub–tree that needs to be traversed. If neither condition is met then both child nodes will be visited in a near–to–far order with an updated parameter interval, which is $[t_{min};min(t_{max},split_{max})]$ for the near and $[max(t_{min},split_{min});t_{max}]$ for the far node.

Employing a single parameter interval is beneficial to ray tracing performance in that less data needs to be maintained and that traversal can be carried out based on two comparisons of scalar values. However, the use of a single interval is associated with a loss of information

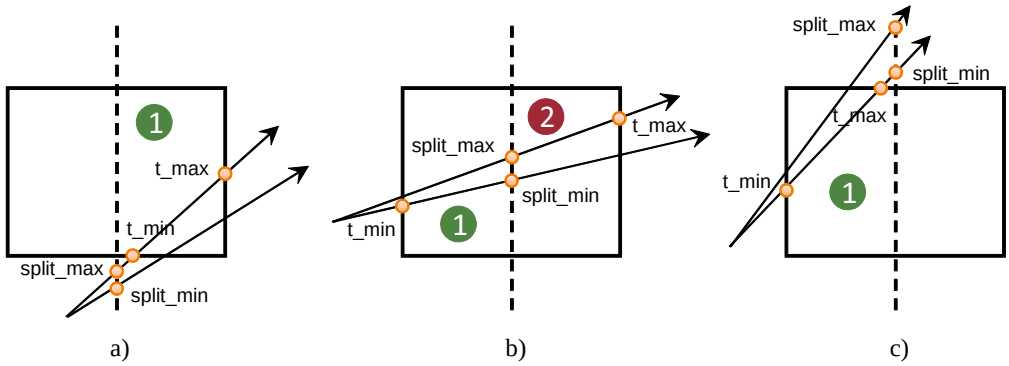


Figure 2.7: *The interval traversal algorithm is based on three cases: a) Only the far child is visited if $split_{max} < t_{min}$. c) If $split_{min} > t_{max}$ then only the near child is visited. b) Otherwise both sub–trees are traversed in near–to–far order.*

that can lead to additional traversal steps. In particular more leaves may be visited, which increases the number of ray–triangle intersection tests. This can be alleviated by clipping the rays to the AABB spawned by the triangles, thereby reducing the number of active rays in the leaf, as depicted in Figure 2.8.

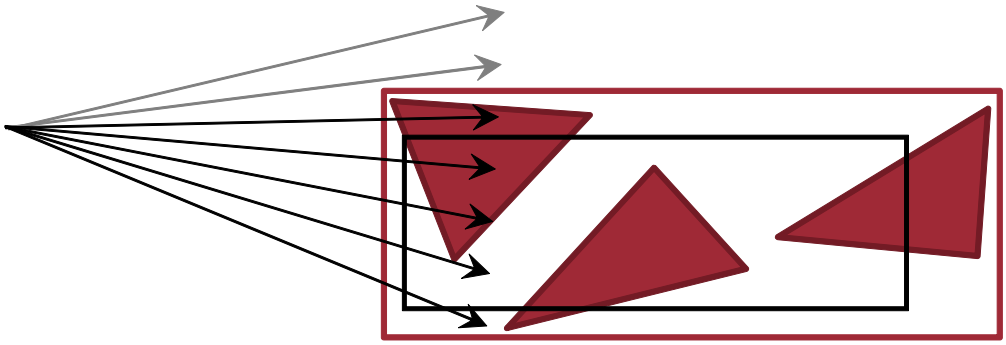


Figure 2.8: *By clipping rays to the AABB of the triangles in a leaf, the number of active rays can be reduced and fewer ray–triangle intersection tests need to be performed.*

Computing a Frustum for Rays

Boulos *et al.* [5] devised an algorithm for computing a frustum for a group of rays, which do not necessarily need to have a common origin. The algorithm first seeks the direction into which the group of rays extends forward. This is approached by determining an axis for which the direction vectors of all rays exhibit the same signs. For very incoherent rays (e.g., rays that extend spherically from a common origin) no such axis may exists, thereby forcing the algorithm to return with no valid frustum. If such a "major axis" is found, however, the algorithm proceeds with calculating the normal vectors for the frustum's bounding planes.

The direction vectors of the rays are projected onto a plane perpendicular to the major axis w by dividing them by their w–value. In this 2–dimensional space $[u \times v]$ the extents of the direction vectors correspond to the slopes of the rays in relation to the major axis. Per–axis bounds for the slopes can therefore be derived from the minimal and maximal extents of

the direction vectors along each axis. These bounds are depicted as axis–aligned lines in Figure 2.9b.

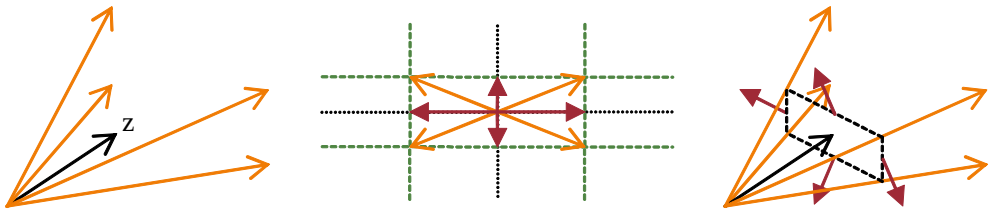


Figure 2.9: *a) The forward axis of a group of rays is determined. b) Ray directions are projected into 2D to determine the bounds of their slopes in relation to the major axis. c) Normal vectors for the frustum's bounding planes are derived from the bounds.*

The normal vectors for the frustum's four bounding planes can then derived from the lower and upper bounds for the slopes, as exercised in Equation 2.6 for the choice of z as the major axis. In case the group of rays extends in the negative direction of the major axis the normal vectors need to be flipped in order to have them point outwards consistently.

$$\vec{n_{left}} = \begin{pmatrix} -1 \\ 0 \\ lower_u \end{pmatrix} \quad \vec{n_{right}} = \begin{pmatrix} 1 \\ 0 \\ -upper_u \end{pmatrix} \quad \vec{n_{bottom}} = \begin{pmatrix} 0 \\ -1 \\ lower_v \end{pmatrix} \quad \vec{n_{top}} = \begin{pmatrix} 0 \\ 1 \\ -upper_v \end{pmatrix} \tag{2.6}$$

The definition of each plane is completed by inserting the origins of all rays into its equation and keeping the maximum of the resulting distance terms.

2.4 Ray Tracing Dynamic Geometry

Although the notion of a dynamic scene is not well–defined it is usually used to refer to scenes that contain geometric primitives that may move or change shape between any two subsequent frames. Dynamic scenes should be seen in contrast to static scenes where all primitives are bound to remain at their initial positions and may not undergo changes affecting size or orientation.

As changes applied to primitives typically invalidate acceleration structures, methods to restore them to correct representations of the contained geometry are an important aspect of handling dynamic scenes. The per–frame time limits imposed in the domain of real–time ray tracing further complicate this and often lead to compromises between the time needed for updates or reconstruction of a given acceleration structure and its quality as determined by ray tracing performance.

2.4.1 Types of Animations

According to [58] the actual motion of primitives may be categorized as either hierarchical or incoherent, with blends being referred to as semi–hierarchical.

Primitives are animated hierarchically if they can be partitioned into groups of primitives that are subject to the same transformations. A typical example for this kind of animation can be found in a car that moves along a path and has its wheels spin around their respective axis. Incoherent motion marks the opposite case with primitives moving or changing shape independently of other primitives. Explosions that are simulated using particles may exhibit this sort of behavior by having the individual primitives, which are used to render the phenomenon, moving away from its center in a chaotic manner.

Semi–hierarchical motion has aspects of both types of motion, visible, for example, when looking at a flock of hierarchically animated characters that move independently of each other.

In their publication Wald *et al.* also refer to two distinct types of animation with regard to their affecting the scene's topology. When animating characters usually only the positions of the vertices are modified and the mesh topology is left untouched. The character's geometry is therefore referred to as deformable, which stands in contrast to geometry that may undergo arbitrary changes. These changes may include modifications of mesh connectivity and addition or removal of primitives.

2.4.2 Kd–Tree Based Solutions

Kd–tree based approaches to handling dynamic scenes are mostly based on rapid reconstruction of a tree. Construction by sampling of the SAH [19] or binning of primitives [40] is performed to reduce build times compared to the standard sweeping approach. Although these techniques construct kd–trees exhibiting lower ray tracing performance, the overall frame rendering time is reduced making interactive or real–time frame rates possible. Highly parallel implementations as presented in [47] reduce build times further while maintaining kd–tree quality.

Another approach was introduced in [13]. Günther *et al.* propose to extend kd–trees to allow for limited movement of primitives, which they call "fuzzy kd–trees". Construction takes place in a preprocessing step and encompasses an analysis of the motion of primitives. This is performed by motion decomposition, which separates motion into an affine transformation and residual motion. By subtracting the former from an animation primitives are moved into a local coordinate system where frame–to–frame movement is typically much smaller. Per–primitive bounding boxes are computed reflecting the residual motion and used as a basis for construction of the fuzzy kd–tree. The extracted affine transformation is handled by applying its inverse to rays prior to traversal, which transforms them into the coordinate system of the fuzzy kd–tree.

A reduction of residual motion is desirable because larger per–primitive bounding boxes have a negative impact on the efficiency of a fuzzy kd–tree. This is due to an increase probability of intersection with rays. Better performance can therefore be achieved by building multiple trees, each for groups of primitives that exhibit similar motion. These groups can be composed automatically by minimization of a cost function based on the residual motion. The individual trees are merged into a single acceleration structure by being referenced in the leaves of a separate top–level kd–tree along with the associated transformation matrices. When traversal enters a leaf, all rays are transformed by the matrix and traversal continues in the bottom–level kd–tree. A comparable approach with a two–level standard kd–tree had been introduced previously in [54], which supported hierarchically animated geometry only.

Fast reconstruction of kd–trees and fuzzy kd–trees both have their respective drawbacks. Although fast implementations of kd–tree reconstruction have been shown, they cannot be considered to be the optimal solution to the problem. Most primitives of typical scenes either don't move at all or only marginally on a frame–to–frame basis. This fact is ignored by kd–tree reconstruction because all information from the last frame concerning the structure of the scene is discarded. By exploiting frame–to–frame coherence the time for maintaining an acceleration structure can be reduced.

Fuzzy kd–trees on the other hand are difficult to build and limited to a specific type of animated geometry, whose motion has to be known at the time of construction. Animations that modify the scene topology cannot be handled just like unpredictable motion in response to user actions in interactive applications.

Several other acceleration structures have therefore been proposed to support fully dynamic geometry with good run–time behavior. Especially bounding volume hierarchies have received a lot of attention for ray tracing of dynamic scenes because they can be updated efficiently to reflect changes to a scene.

2.4.3 Bounding Volume Hierarchies

Bounding volumes are a conservative representation of geometry. They are used in fast rejection tests to determine whether a group of primitives can potentially intersect with a ray or not. In the latter case intersection tests between the ray and the individual primitives can be skipped, which can yield a substantial improvement of run–time performance if the bounding volume encompassed a large number of primitives.

Requirements for bounding volumes are four–fold:

- A bounding volume should fit its geometry as tightly as possible in order to reduce the probability of false positives, where a ray intersects the bounding body but not the actual geometry.
- Tests to detect intersections between rays and bounding volumes should be run–time efficient to limit the introduced computational overhead introduced. It makes no sense to use bounding volumes that are more expensive to intersect with than the original geometry.
- Bounding volumes should require minimal storage.
- Construction of bounding volumes should be run–time efficient in scenarios where frequent updates due to changing geometry are necessary.

Different bodies have been proposed to serve as an approximation for geometry, e.g. spheres, axis–aligned bounding boxes (AABB), oriented bounding boxes (OBB) [11] and discrete orientation polytopes (k–DOPs) [29].

Spheres are light–weight objects that have a low memory footprint and are fast to intersect with. Their use as approximations of triangular geometry is limited because tight fits are typically not possible and the introduced empty space increases the probability of false positives. K–DOPs are convex polytopes defined by a set of k planes. They can serve as good approximations of geometry, especially with an increasing number of planes, but have high associated run–time costs regarding both construction and intersection with rays. Memory requirements depend on the number of planes but are typically higher than those of spheres, AABBs and OBBs.

AABBs have been shown to be a good compromise between all four requirements. An AABB's memory footprint is low as it can be stored in two three–dimensional vectors. These vectors correspond to the minimal and maximal extents of the approximated geometry in each dimension and can be determined in linear time for a set of primitives. Also several algorithms are available to detected intersections with rays [14, 27, 34, 48, 61].

Kay and Kajiya [27] propose to intersect a given ray with each of the six axis–aligned planes that define an AABB. The resulting distances to the intersection points are then used to form intervals for each axis. The ray is only reported to hit the AABB if all three intervals overlap, as depicted in Figure 2.10 in two dimensions. Smits proposed an efficient implementation of the algorithm, which exploits the properties of floating–point arithmetic to reduce the number of branches [48]. His version is straightforward to implement using vector instructions to test multiple rays for intersection with an AABB at once. Listing 2.1 gives an implementation in C++ for single rays.

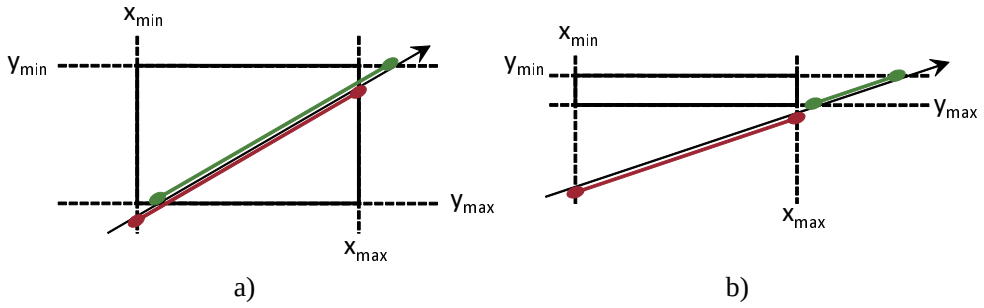


Figure 2.10: *a) The intervals along each axis overlap signalling that the ray hits the AABB. b) The intervals do not overlap because the ray misses the box.*

```
bool IntersectRayWithAABB(ray r, aabb box)
{
  float nearDist = 0.0f, farDist = Infinity;
  for(int axis = 0; axis < 3; ++axis)
  {
    float org = r.origin[axis];
    float dir = 1.0f / r.direction[axis];
    float t0 = (box.lower[axis] - org) * dir;
    float t1 = (box.upper[axis] - org) * dir;
    nearDist = max(nearDist, min(t0, t1));
    farDist = min(farDist, max(t0, t1));
  }
  return (nearDist <= farDist);
}
```

Listing 2.1: *Testing a ray for intersection with an axis–aligned bounding box*

Bounding volume hierarchies (BVH) are based on an arrangement of bounding volumes in a tree–like structure, where higher–level volumes completely encompass the bounding volumes of their descendents [45]. Although this definition does not impose any restrictions on the type of the bounding bodies and on the branching factor of the tree, AABB–based binary BVHs are commonly used in ray tracing.

BVHs are an acceleration structure based on partitioning of an object list. Primitives are therefore referenced only once, with references being stored in per–leaf lists. À priori al-

location of the required memory is therefore possible, which allows for construction and updates to be performed in–place.

2.4.4 Construction of Bounding Volume Hierarchies

Kay and Kajiya [27] introduced a top–down construction algorithm that builds BVHs by recursively splitting primitives at the spatial median. Goldsmith and Salmon [10] used a cost model, which would be the basis for the SAH for kd–trees, to guide a bottom–up construction algorithm. Primitives are inserted one after the other into an existing tree thereby modifying its structure by introducing new splits. A drawback of this approach is a dependency on the order of the primitives. Randomization of the order and building of multiple trees with subsequent selection of the best tree are attempts to remedy this problem.

Müller and Fellner [39] merged both approaches by performing top–down construction based on the cost function devised by Goldsmith and Salmon, which was further refined by Wald *et al.* [57]. The resulting algorithm is similar to the SAH–based kd–tree construction algorithm [55] albeit with differences related to the fact that a BVH partitions an object list instead of space.

In particular there are only $O(n)$ reasonable candidate planes, which are located at the primitives' bounds in each dimension, for splitting a node in a kd–tree. For BVHs and object list partitioning schemes in general, however, there are $O(2^n)$ ways of partitioning a set of primitives into two disjoint sub–sets. Evaluation of all possible configurations is therefore intractable. By taking only partitions into account that are derived from a per–axis sorted list of primitives the number of configurations is limited to $O(n)$, which can be evaluated by the construction algorithm sufficiently fast.

Similar to kd–tree construction in [55] the costs for a set of potential splitting planes are evaluated and the plane with the minimal associated costs is selected to carry out the split. Wald *et al.* note that using a set of planes situated at the primitive's bounds in each dimension leads to no discernable improvement of ray tracing performance over using the axis–aligned planes going through the centroid of each primitive's bounding box. The BVH construction algorithm therefore operates on the latter set, which encompasses only half as many candidates. Classification of primitives regarding their position relative to a plane is performed based on centroids, too. This simplifies the classification to a 1D–comparison based on the signed distance of a point to the plane, which is desirable because it is not obvious how plane–straddling primitives should be handled.

It is important to note that the calculation of surface areas is based on AABBs computed to tightly fit the primitives in both partitions. This computation can be implemented efficiently in a two–pass approach, by computing the per–candidate left bounding box iteratively though sweeping from the left and reusing it in the second pass, which sweeps from the right and also evaluates the cost function. Kd–tree construction derives the AABBs for the two partitions in $O(1)$ from the current node's AABB by splitting it with the candidate plane.

Algorithm 4 presents the SAH–based construction algorithm for BVHs in pseudo code. Its asymptotical run–time is $O(n \log^2 n)$. The comparable algorithm for kd–trees exhibits a run–time behavior of $O(n \log n)$ because it performs sorting only once unlike the presented algorithm for BVHs, which sorts primitives repeatedly with every step.

Algorithm 4 Centroid–based SAH partitioning

```
function PARTITION(triangles T) returns node
    Ĉ = C_tri * |T|, âxis = −1, mid = −1 /* initialize search */
    for axis = (x, y, z) do
        sort T using bounding box centroids in axis
        /* sweep from the left */
        triangles T_1 = ∅, T_2 = T
        for i = 1 to |T| do
            T[i].leftArea = Area(T_1)
            move triangle i from T_2 to T_1
        /* sweep from the right */
        T_1 = T, T_2 = ∅
        for i = |T| to 1 do
            T[i].rightArea = Area(T_2)
            /* evaluate the SAH–costs for the current configuration */
            C = SAH(|T_1|, T[i].leftArea, |T_2|, T[i].rightArea)
            move triangle i from T_1 to T_2
            if C < Ĉ then
                Ĉ = C, âxis = axis, mid = i
    if âxis ≠ −1 then
        sort T using bounding box centroid in bestAxis.
        T_1 = T[1..mid], T_2 = T]mid..|T|]
        return new node(âxis, Partition(T_1), Partition(T_2))
    else
        return new leaf node(T) /* found no partition better than leaf */
```

Construction of BVHs in $O(n \log n)$ can be performed by employing techniques like SAH–sampling and binning of primitives known from kd–trees. A binning approach was presented in [53].

2.4.5 Traversal of BVHs

Traversal of a bounding volume hierarchy is conceptually simple. When an inner node is visited an intersection test is carried out between the ray and the bounding volumes of the two children. Based on the results either sub–tree may be culled or traversed recursively. Upon entering a leaf each of the referenced primitives is intersected with the ray. No early–out condition exists for standard rays because traversal is not performed in front–to–back order and potentially closer intersections with primitives may be found in other sub–trees. An iterative implementation in pseudo code is given in Algorithm 5.

A number of improvements can be applied to the described traversal algorithm to reduce its average run–time. An early–out condition can be evaluated for so–called shadow rays, which are used to determine whether a surface is illuminated by a given light source or not. This check is performed by tracing a secondary ray from the visible point on the surface to the light. In case any primitive is intersected in between the point lies in shadow with regard to this particular light source. This occlusion test does not require finding the nearest point of intersection, any intersection between the ray's origin and the light source will suffice. Implementing this early–out check for shadow–rays is straightforward but may drastically reduce the overall rendering time because shadow rays are usually spawned in large numbers.

Algorithm 5 Iterative BVH traversal

```
function TRAVERSE(ray r)
    stack S = rootNode
    while S not empty do
        curNode ← S /* pop next node from the stack */
        if curNode is leaf then
            intersect r with triangles
        else
            (leftChild, rightChild) = curNode's children
            if r intersects leftChild then
                if r intersects rightChild then
                    (nearChild, farChild) = determine order of (leftChild, rightChild)
                    S ← farChild /* push far child onto the stack */
                    S ← nearChild /* put near child on top of the stack */
                else
                    S ← leftChild
            else
                if r intersects rightChild then
                    S ← rightChild
```

Culling of sub–trees can also be optimized by visiting nodes only if they are intersected in a point that is closer to the ray's origin than the nearest recorded intersection with a primitive. Another improvement, which builds upon this behavior, affects the order of traversal. The baseline traversal algorithm makes no particular assumptions about the spatial ordering of nodes and typically traverses the sub–trees in a fixed order. By evaluating the spatial relationship of siblings during construction a weak front–to–back traversal can be implemented [35]. Depending on the direction of the ray the near node is always visited first which increases the probability that the far sub–tree can be skipped.

Another optimization has been proposed in [48] that enables stack–less traversal of a BVH by storing so–called skip pointers to the next node in the traversal sequence.

2.4.6 Traversal with Ray Packets

By grouping rays together and performing traversal of the bounding volume hierarchy with packets ray tracing performance can be increased similar to kd–tree traversal with packets.

Generalization of the single ray traversal algorithm to multiple rays is straightforward. A sub–tree is traversed if any of the rays intersects with its root node's volume. Especially large packets of rays benefit from this behavior due to the fact that detecting a ray–node overlap allows for skipping the rest of the rays. This results in typically only a single test for coherent packets that intersect with a node, which lowers the computational costs considerably. However, it is important to note that all rays need to be tested against a node in case the whole packet misses it.

Wald *et al.* propose to address this aspect by employing the frustum–based approach introduced with the multi level ray tracing algorithm by Reshetov *et al.* [44] for quick miss tests [57]. A set of planes is computed prior to traversal that acts as a conservative representation for the ray shaft. Frustum culling is then used to determine whether any of the rays of the packet can actually intersect a node or not. Large packets benefit from this approach a lot because the quick miss test is independent from the number of rays and therefore free for additional rays.

The complete algorithm by Wald *et al.* performs up to three steps to determine whether a sub–tree has to be traversed:

1. The quick hit test encompasses checking a single ray for intersection with the node in question. If a hit is detected the algorithm descends into the sub–tree directly without going through any of the subsequent phases. With regard to the fact that a ray, which misses a particular node, also misses all of its descendents, it is wise to perform the test with a ray that is not known to have missed the parent node. This is accomplished by keeping track of the first active ray in the packet over the course of traversal of a sub–tree.
2. The quick miss test uses the packet's frustum to quickly reject a node for all rays.
3. The last resort is testing the rest of the rays until a hit is detected or it is know that the entire packet misses the node.

By traversing a sub–tree with all rays in the packet when a single ray overlaps with it, the number of ray–node intersection tests can be reduced significantly for large packets. As a consequence rays may descend into lower levels of the hierarchy even if they do not overlap with the volumes represented by the nodes. Although this has no negative impact on traversal it may increase the number of ray–primitive intersection tests that are carried out when entering a leaf. Noting that these tests are typically more expensive than ray–node intersections, Wald *et al.* suggest testing all remaining active rays for overlap with leaves to determine the exact set of active rays prior to operations involving rays and primitives.

Although the complete traversal strategy can be implemented with scalar code operating on individual rays the use of vector instructions poses a performance advantage. Rays should therefore be processed in groups of the native vector size (e.g., 4 for SSE on the x86 platform) when performing ray–node overlap tests and intersection tests with the primitives referenced in the leaves of the BVH.

2.4.7 Updating Bounding Volume Hierarchies

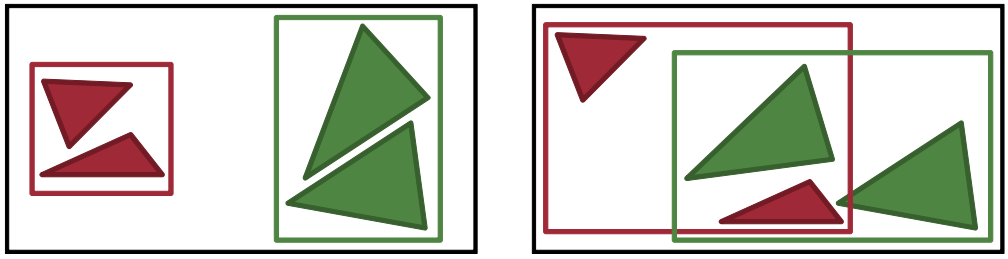
Animation of primitives invalidates acceleration structures, which therefore need to be rebuilt or updated to reflect the changes in a scene. Although bounding volume hierarchies can be built efficiently, which would make per–frame rebuilding feasible, better run–time performance can be expected from updating an existing BVH and thereby reusing information from the last frame.

Refitting is a well–known technique that updates the bounding volumes of the individual nodes in a BVH [57]. In a first step the bounding volumes are recomputed for the leaves of the tree based on the referenced primitives. Subsequently the bounding volumes of the inner nodes and the root node are updated in bottom–up order by merging the volumes of the respective children. The whole operation is linear in the number of primitives and can therefore outperform rebuilding, which has an asymptotical run–time behavior of $O(n \log n)$. Lauterbach *et al.* [31] report that refitting is about four times faster than rebuilding for their benchmark models.

It is, however, important to note that refitting does not modify the partitioning structure of a bounding volume hierarchy. The technique is merely an approach to restoring a BVH to a correct representation of the geometry. Culling efficiency may therefore degrade if primitive movement increases the overlap of nodes or empty space within the volume of a node.

In order to maintain the quality of the acceleration structure, Wald *et al.* [57] propose to perform rebuilding in the background stretched out over several frames, with the resulting hierarchy replacing the potentially degenerated existing hierarchy.

Lauterbach *et al.* present another approach in [31]. They employ a heuristic to measure degradation of a BVH and perform rebuilding only if a specified threshold is exceeded. The heuristic is based on the observation that the surface area of a node in a well–built tree is typically much larger than the combined surface areas of its children. This relationship is depicted in Figure 2.11. By assuming that the ratio between these two values is optimal when the BVH is built and recording the initial ratio for each node, they have a base for measuring the degradation of a hierarchy's quality over its lifetime.



a) b)

Figure 2.11: *a) Good partitioning of primitives results in a pair of nodes with small AABBs compared to the parent's AABB and with minimal overlap. b) Movement of primitives leads to a degeneration of the partitioning structure. Bounding boxes grow in size and the overlap between siblings increases, which negatively affects a BVH's culling efficiency.*

During refitting the ratio of a node's surface area to the combined surface areas of its children is calculated for each node. The differences to the original ratios are accumulated and divided by the number of nodes to get a measurement of the average degradation of culling efficiency. Rebuilding is then initiated if a certain threshold is exceeded. Lauterbach *et al.* report that a threshold of 40% resulted in a good balance between ray tracing time and updating time and enabled rendering at stable frame rates over the course of an animation.

2.4.8 Selective Restructuring

A more sophisticated approach to the problem of updating a BHV was introduced in [64] under the name "selective restructuring". Restructuring operates on pairs of nodes that exhibit poor culling efficiency due to an overlap of the nodes' bounding volumes. By repartitioning the primitives that are referenced in the respective sub–trees the overlap can be reduced resulting in an improvement of the BVH's quality. This operation is performed in two phases, hierarchical refinement and restructuring, that are carried out after refitting was applied to the BVH.

During hierarchical refinement the BVH is traversed in a top–down manner to prepare a set of node pairs for the subsequent restructuring phase. The algorithm is based on a queue of node pairs that is initialized with the children of the root node. It iteratively retrieves an entry from the queue until it is depleted and refines it to pairs of nodes with overlapping volumes. Potential candidates for restructuring are recorded additionally.

Refinement of a given pair of nodes (n1, n2) is performed based on the overlap of their volumes.

- In case no overlap is detected the BVH guarantees that the sub–trees of n1 and n2 do not overlap either and pairs containing nodes from both sub–trees can be ignored. (n1, n2) is no candidate for restructuring and is refined to two pairs made up by the respective children of n1 and n2. The search therefore continues strictly with pairs of nodes from a single sub–tree.
- If n1 and n2 overlap (n1, n2) is stored as a candidate pair. To further localize the overlap of the two nodes' sub–trees, (n1, n2) is refined. The children of the node with the larger volume, for example n1, are paired with the other node, yielding the pairs (n1.left, n2) and (n1.right, n2). In addition to that the pair (n1.left, n1.right) is created to also continue search in the spatially larger sub–tree.

Hierarchical refinement also encompasses the evaluation of the so–called restructuring metric for each node pair. The metric enables measuring of the benefit that can be expected from applying restructuring to the nodes based on a probabilistic model. It is used as a stopping criterion for hierarchical refinement that prevents node pairs from being restructured if the potential gain is outweighed by the costs. A detailed discussion of the metric is outside the scope of this thesis and available in the original paper [64].

The second phase of the algorithm uses the generated candidate list to perform the actual restructuring operation as depicted in Figure 2.12. The union of the primitives referenced in the sub–trees of both nodes is repartitioned yielding new nodes with less overlap and better culling efficiency.

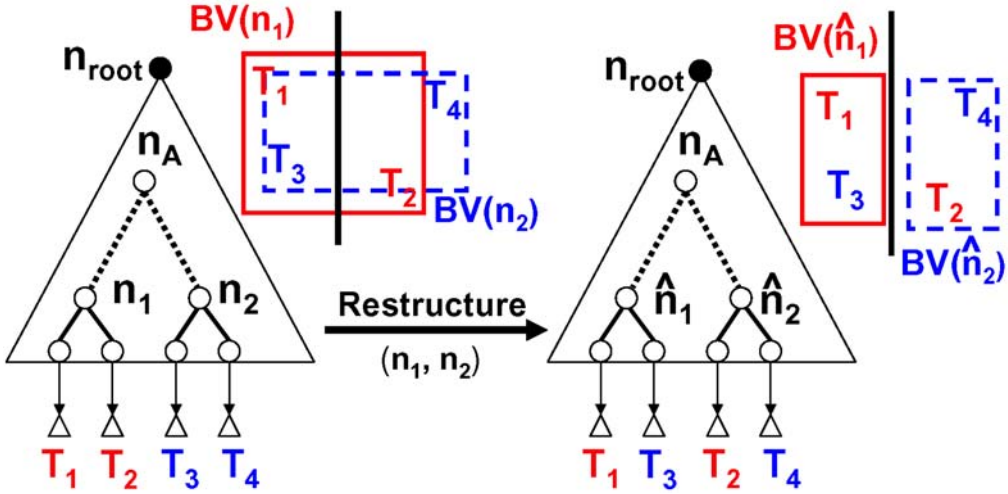


Figure 2.12: *Restructuring selects a pair of nodes with overlapping bounding volumes. The union of the primitives of both sub–trees is then repartitioned to reduce the overlap and improve the BVH's culling efficiency [64].*

Chapter 3

Design

This chapter gives an overview of the major challenges of building a real–time ray tracing system. Then, solutions based on the previously introduced approaches to accelerating ray tracing are presented and contributions of this thesis are highlighted. The chapter concludes with the design of a general purpose ray tracing library, whose implementation will be discussed in detail in the next chapter.

3.1 Challenges

Ray tracing is a conceptually simple rendering technique that is, however, difficult to implement efficiently. In particular the run–time costs associated with tracing rays are high and need to be lowered considerably for real–time applications.

An important algorithmic optimization is the use of an acceleration structure, which partitions the primitives of a scene to reduce the number of ray–primitive intersection tests. A single ray can then be traced in $O(\log n)$, which is a significant improvement over the linear asymptotical run–time of the original algorithm. However, acceleration structures need to be built and also maintained for animated geometry. This increases the overall complexity of a ray tracing implementation and may introduce an additional run–time overhead eventually prohibiting interactive frame rates.

In general the choice of an acceleration structure is not obvious due to differences in memory requirements, culling efficiency and support for fast building or updating, which is especially important in ray tracing dynamic geometry. Adaptive acceleration structures like the kd–tree and the bounding volume hierarchy are commonly used in real–time ray tracers. They can be built in the minimal asymptotical run–time for comparison–based approaches of $O(n \log n)$. The construction algorithms can also be parallelized to exploit the computational power of multi–core systems. Nevertheless per–frame rebuilds of acceleration structures are problematic. Even moderately sized scenes with primitive counts in the range of $100,000$ to 1 million result in large datasets that need to be processed during construction. The available memory bandwidth for ray tracing, which is typically already a limiting factor on general purpose architectures, is therefore further decreased, which has an adverse effect on the overall rendering performance.

Updating schemes have therefore been devised to retain the quality of an acceleration structure for animated geometry. By reusing information from the last rendered frame and modifying the structure only partially the overhead of maintaining an acceleration structure can

be reduced. This is particularly true for deformable geometry and rigid animations, which exhibit spatially localized motion of primitives. Incoherent motion, on the other hand, can result in a degradation of quality of large sub–trees. In this case complete reconstruction may be faster than attempts to perform selective restructuring because the latter incurs a certain overhead associated with determining the sub–trees that are best updated. Updating is therefore not always superior to rebuilding and needs to be applied with care. Employing an updating scheme can also negatively affect traversal due to additional data that needs to be stored to record the previous state of an acceleration structure. In adaptive acceleration structures like the BVH this typically increases the node size and therefore also the consumed memory bandwidth during traversal.

Another challenge of performing real–time ray tracing is fast tracing of incoherent rays. Run–time efficient traversal of acceleration structures is based on exploiting coherence of a set of rays by traversing them together in packets. This works reasonably well for primary rays, which can be arranged in groups with a principal direction and low divergence. The rays therefore traverse large parts of an acceleration structure together, which allows for high amortization of traversal costs. Secondary rays are spawned in the process of shading a hit point to implement effects like shadows, reflections and refractions. Depending on the distribution of light sources and the curvature of the shaded surface, secondary rays are subject to heavy scattering across the hemisphere over the surface, as can be seen in Figure 3.1. Packet tracing then degenerates quickly to traversal with only a single active ray and run–time performance drops. Attempts by Månsson *et al.* [36] to group rays into coherent packets by employing sorting techniques were unsuccessful and so efficient tracing of secondary rays remains an unsolved problem.

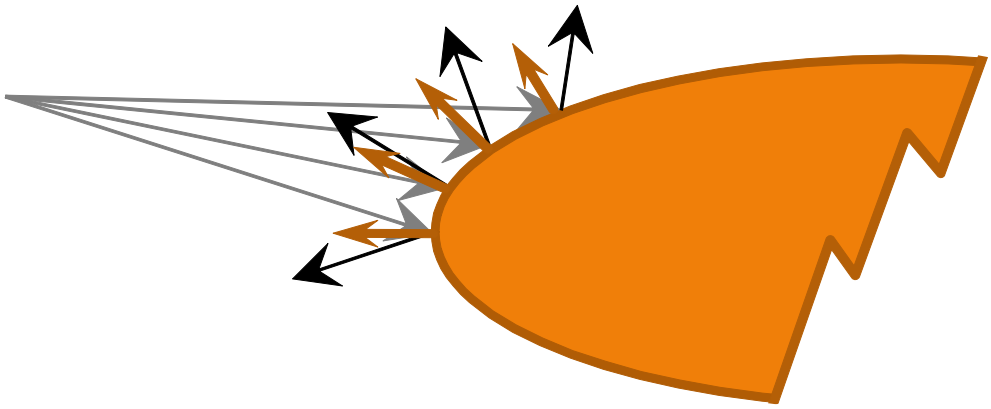


Figure 3.1: *Secondary rays are typically incoherent yielding lower run–time performance in packet–based approaches.*

Shading is also problematic because it does not map well to general purpose processors due to its floating–point intensive nature. Reducing branches to a minimum is important for achieving good performance, which can be difficult when complex material systems with a large number of effect combinations need to be supported. Additionally tracing secondary rays is prone to producing visual artifacts due to limited precision of floating point calculations. A common approach to remedy this problem is to apply an offset to a ray's starting point along its direction. However, this so–called epsilon value needs to be chosen carefully to avoid getting rays stuck in the actively shaded surface and skipping valid intersections with nearby geometry. What further complicates this is that epsilons are typically determined empirically for a given scene and generally don't map well from one scene to another.

3.2 High–Level Solutions

Chapter 2 introduced four classes of geometry based on the motion of primitives and the stability of the geometry's topology over its lifetime:

- Static geometry exhibits no motion and has a constant topology that is not subject to addition or removal of primitives.
- Primitives that are static in relation to each other but move as a group on the scene level contribute to the class of geometry with rigid animations.
- Deformable geometry encompasses meshes with static connectivity but motion of their vertices over the course of animations.
- Fully dynamic geometry may exhibit incoherent motion of individual primitives and topology changes due to the addition or the removal of primitives.

Fully dynamic geometry is the most general class and also encompasses geometry that is characterized by one of the other three classes. It is therefore possible to handle dynamic scenes with a single strategy that supports all aspects of fully dynamic geometry (e.g., by performing a complete rebuild of an acceleration structure each frame, thereby ignoring inter–frame coherency). A benefit of this approach is its simplicity with regard to its implementation because there is only one code path that needs to be created and maintained.

However, better run–time performance can be expected from approaches that address all four classes individually and exploit their characteristic properties. Especially scenes with a large proportion of static geometry benefit from a diversified approach because acceleration structures for static geometry can be created in a preprocessing step and may be optimized for high ray tracing performance.

3.2.1 Addressing Static Geometry

Kd–trees are widely regarded as the optimal data structure for handling static geometry in a ray tracer. They adapt well to geometry and can be traversed efficiently with single rays and with ray packets.

In order to be able to achieve optimal run–time performance it is necessary to perform construction based on the surface area heuristic, which delivers superior results compared to other known heuristics [15]. Algorithms that evaluate all reasonable splitting positions like the plane sweeping approach by Wald and Havran [55] should be preferred for optimal tree quality, too. Although there are faster alternatives like SAH–sampling [19], which also build good kd–trees, the reduction of build–time is typically irrelevant because the construction of acceleration structures for static geometry is performed in a preprocessing step and therefore has no effect on ray tracing performance.

An important aspect of SAH–based construction is the termination criterion. It is evaluated prior to splitting of a node to determine whether further subdivision actually increases the kd–tree's quality or not with regard to better ray tracing performance. A common approach is to evaluate the costs of the optimal splitting plane and compare them to the costs of turning the node in question into a leaf of the kd–tree. The node is split only if the associated costs are less than those of the alternative. Although this seems to be a reasonable solution for the termination problem kd–trees with better quality can be built by employing a refined criterion.

The described termination criterion performs a greedy evaluation of the SAH cost function when determining the costs of the optimal splitting plane, which is based on the assumption that a split always produces a node with two leaves as children. Further partitioning steps, which might well reduce the costs of the current sub–tree, are therefore ignored and the construction algorithm may get stuck in a local minimum. This problem can be eliminated by continuing subdivision as long as the preconditions of the construction algorithm are met (i.e., that there is at least a single primitive in the active volume) and performing backtracking if a deeper tree results in no reduction of the costs. In order to limit the increase of build time over the original termination criterion an upper bound of the number of backtracking points can be specified.

Premature termination of kd–tree construction can severely degrade ray tracing performance due to too coarse partitioning of space and an associated increase of ray–primitive intersections during traversal. However, a reduction of performance can also be observed for kd–trees that exhibit very fine partitioning. This is related to the additional traversal steps that need to be performed in a deeper tree, which do not only increase the computational load on the processor but also raise memory bandwidth requirements, which quickly become a bottleneck on general purpose processors. Although the SAH cost function incorporates a factor to represent these costs, an extension to the termination criterion can yield smaller kd–trees with equal or marginally better ray traversal behavior.

It can be observed that the absolute gains of splitting a node decline with the depth of the kd–tree, which is only natural because the effects of the splits are localized in potentially small sub–trees that have a low probability of being traversed by rays. Nevertheless the termination criterion in its current form often does not stop further subdivision because it determines that performing a split is less expensive than turning the node into a leaf, even if only marginally. In this thesis this problem is addressed by introducing a depth–based factor to limit the creation of deep sub–trees that have only little impact on the overall costs.

In order to avoid irrelevant splits the termination criterion is modified to favor the creation of leaves at increasingly deep levels. This is accomplished by scaling down the costs of leaves based on their depth in the kd–tree relatively to the costs of splitting a node. During the work on this thesis it has been discovered that an exponential scaling function gives very good results and allows for a reduction of the size of kd–trees by up to 80% without sacrifices in ray tracing performance. A detailed analysis based on a variety of test scenes is given in chapter 5.

3.2.2 A Two–Level Acceleration Structure for Dynamic Geometry

After a ray was intersected with the static geometry in the kd–tree a second dynamic acceleration structure has to be traversed to find potentially closer intersections with animated primitives. In this thesis the use of bounding volume hierarchies is investigated for this purpose and a two–level acceleration structure is proposed that supports rigidly animated geometry, deformable geometry and fully dynamic geometry with incoherent motion and topology changes.

Bounding volume hierarchies have a number of advantages over other hierarchies that make them particularly useful in the context of dynamic geometry. As an object list partitioning scheme they are fast to build and adapt well to a given distribution of primitives when the surface area heuristic is used by the construction algorithm. Furthermore it is possible to update BVHs to reuse partitioning information and exploit frame to frame coherency.

BVHs are therefore a great base for the two–level acceleration structure. Its lower level is made up from separate bounding volume hierarchies containing the raw geometry, which typically reflect the logical structure of the dynamic scene (i.e., by having such a container for each animated character). These hierarchies are not built upfront but are constructed lazily when a ray traverses them, thereby reducing the work load by evaluating only braches that are actually needed.

To facilitate this, a BVH is initially only composed of a single leaf node that encompasses the complete set of primitives. When a ray hits the leaf's volume the traversal algorithm attempts to split the node based on the surface area heuristic by evaluating a set of planes through sampling [19]. If this operation is successful the traversal algorithm proceeds by traversing the newly created children nodes, which are again subject to potential splitting. However, in case a split would yield no reduction of the overall costs the leaf is marked as final. This suppresses further splitting attempts and forces the traversal algorithm to intersect rays with all referenced primitives.

Lazy construction can be implemented efficiently for object list partitioning schemes due to the fact that the upper bound of the number of nodes can be derived from the number of primitives. This allows the construction algorithm to operate on pre–allocated memory, thereby eliminating the overhead of memory allocations, which would have a negative effect on ray tracing performance. Lazy construction is also a perfect base for a new updating scheme called "selective rebuilding" that has been developed within the context of this thesis.

Selective rebuilding can be regarded as a subset of selective restructuring. Like selective restructuring it supports partial rebuilding of a bounding volume hierarchy but performs this operation on complete sub–trees only. Similarly to refitting all bounding volumes are recalculated in bottom–up level order to adapt the hierarchy to the new configuration of primitives. In addition to that the SAH–based costs are reevaluated for each node and compared to the costs of the original configuration, which were recorded when the nodes were split in the first place. When the new costs exceed the old by a specified threshold, which is a hint that the quality of the BVH might have degraded, a rebuild of the local sub–tree is issued. In a system that uses lazy construction this is can be done by freeing all descendents of the sub–tree and turning its root node into a leaf. Lazy construction then carries out the rebuild when rays hit the node. By modifying the rebuild threshold it is possible to balance BVH quality and updating time.

The upper level of the two–level data structure is comprised of a single BVH that contains references to the hierarchies of the lower level in its leaves. These references are associated with a transformation matrix that enables rigid animation of geometry without transforming the primitives individually and updating the lower–level BVHs. This is accomplished by modifying the traversal algorithm to transform the rays for each hit reference with the associated matrix and to continue traversal in the lower level. A similar approach albeit for kd–trees was proposed previously by Wald [54].

By allowing multiple references to a single lower–level BVH, it is possible to implement geometry instancing with little extra effort. The concept of instancing was introduced by Sutherland [50] and is based on the idea that scenes typically contain objects that appear several times in different spots, possibly rotated and scaled. Instead of embedding their primitives multiple times in the acceleration structure the objects' respective geometry is defined only once and inserted via references. This reduces both the memory requirements and the time spent in building and maintaining the hierarchies because the number of primitives is reduced.

3.2.3 Tracing Rays with Multiple Acceleration Structures

Traversal of an acceleration structure with a ray typically identifies the hit primitive with a handle that can be used to access related data such as per–vertex attributes for triangles. Additional data may include the intersection distance and barycentric coordinates of the intersection point.

If more than a single acceleration structure needs to be traversed, it is possible to combine these results by keeping only the information about the nearest intersection point. However, better ray tracing performance can be achieved by reusing the intersection distance from one traversal pass in the subsequent traversal of an additional acceleration structure. This allows limiting the search range for intersections and therefore enables skipping of sub–trees that contain only primitives farther away than the already recorded intersection.

The ideal order of the traversal of acceleration structures is not obvious and might even vary from ray to ray depending on the distribution of primitives. Generally, shallow hierarchies should be traversed first because potential intersections may on average be detected in fewer steps compared to hierarchies with deep sub–trees. The efficiency of traversal algorithms is another aspect of choosing a traversal order. Kd–trees in particular can be traversed faster than other acceleration structures, which should therefore be performed early in order to be able to use detected intersection points for culling in the subsequent traversal of other acceleration structures.

3.2.4 Coherent Ray Packet Assembly

Packet–based traversal of acceleration structures allows for amortization of memory accesses and traversal steps over the rays in a given packet. However, the possible gains in run–time performance depend largely on the coherence of rays in packets. Especially secondary rays, which account for a large fraction of the overall number of traced rays per frame, are typically incoherent and yield a reduction of ray tracing performance.

Furthermore it becomes increasingly difficult to assemble packets of sufficient size in deep levels of the shading tree. Whenever the rays of a packet intersect with primitives that do not share a common material, it is necessary to invoke more than a single function to perform material specific shading operations. This reduces the number of active rays in each path, which limits the number of secondary rays, too, as illustrated in Figure 3.2.

To alleviate this problem a system based on cooperative threads was proposed in [42], where a large number of image tiles is rendered in an interleaved pattern. Whenever a request for tracing of rays is made the rays are inserted into a queue to be sorted for high coherence. Additionally the active cooperative thread is put on hold. In case enough rays were collected for assembly of coherent packets they would be processed and the respective threads that requested tracing would be marked active again if their request was fulfilled. Otherwise another tile would be processed.

Although this approach enables ray traversal of an acceleration structure with large coherent packets the overhead of sorting of rays and switching between cooperative threads is prohibitive in real–time applications. In this thesis methods for automatic grouping of rays are therefore not employed. Instead it is attempted to assemble coherent ray packets by exploiting domain specific knowledge in materials (e.g. by tracing shadow rays from one light source at a time to all active hit points, thereby creating packets of rays with a common origin).

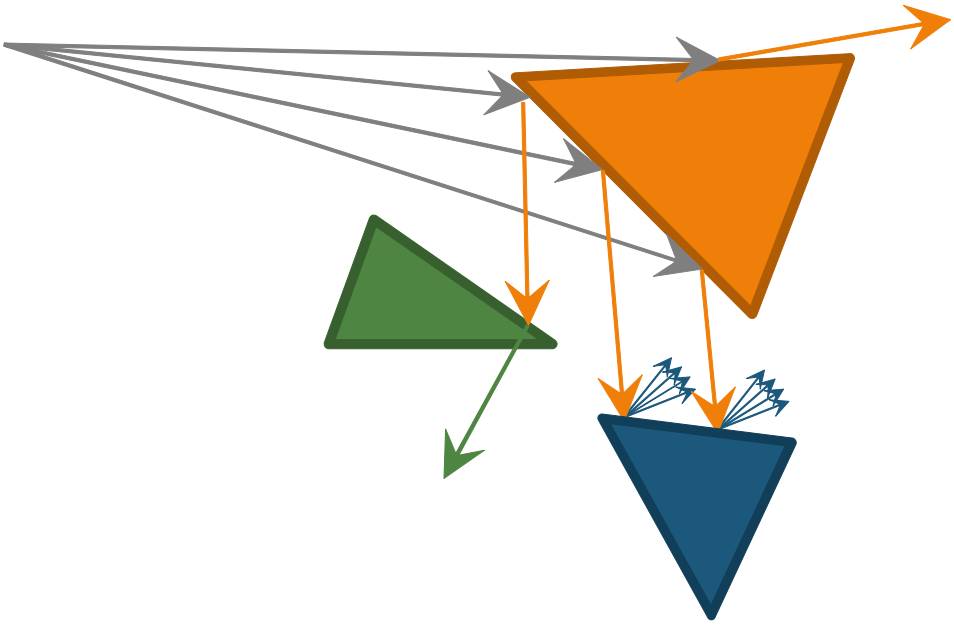


Figure 3.2: *Assembly of packets of sufficient size is diffuclt in deep levels of the shading tree where typically only few rays are active in a given path.*

3.2.5 Resolving Branches in Material Systems

Typical material systems are based on a collection of properties that are set on a per–object or per–primitive basis usually by evaluation of a description file at run–time. Shading of hit points is then implemented by interpreting these properties. In order to be able to support a large feature set, different code paths need to be created (e.g., for reflection and refraction effects). Conditions for the branches associated with these fragments of functionality need to be evaluated for each shaded ray, which may degrade performance considerably. Material specific optimization opportunities are also missed out by having a generic function for all material configurations.

The use of run–time code generation for materials was proposed in [43]. This retains the flexibility of a run–time parameterizable material system and allows for performance improvement by elimination of branches and folding of constants. Code generation is non–trivial and time consuming to implement, particularly when dealing with multiple platforms. It is therefore advisable to rely on external libraries, such as the low–level virtual machine (LLVM) [30], for this purpose.

The LLVM is an abstraction from hardware and is based on a virtual instruction set. It supports emission of code through an object–oriented interface and run–time generation of machine code for a large number of platforms. Interoperation with C/C++ code is possible through dynamic linking, which is handled transparently by the LLVM. A large number of optimization passes are available that can be applied selectively to balance compilation times and the quality of the generated code.

3.3 Designing a Flexible Ray Tracing Library

The presented high–level solutions form the base for a ray tracing library that was developed over the course of this thesis. The library enables ray tracing of reasonably complex triangle–based dynamic scenes at real–time frame rates, which is employed in a variety of rendering applications that will be discussed in chapter 5.

However, an additional design goal for such a ray tracing library was to make sure that it is useful not only in the context of image synthesis but also in other areas where ray tracing is employed (e.g., collision detection). This was approached by restricting the library to deal with raw geometry only, thereby leaving management of additional per–triangle domain specific attributes (e.g., normal vectors, materials) to the applications. Therefore the library mainly provides containers for triangle–based geometry with support for fast ray tracing queries.

Kd–trees are supplied for static geometry, whereas the intended use for the previously introduced two–level acceleration structure lies in the handling of dynamic geometry. Results from traversing these data structures encompass a unique identifier for the hit triangle, the distance of the hit point from the ray's origin and the barycentric coordinates of the hit point. This set of data describing a hit point should be general enough to be useful in a broad range of applications and can be extended based on the identifier by looking up additional triangle–specific attributes.

Chapter 4

Implementation

Aspects specific to the implementation of the ray tracing library developed within the context of this thesis are discussed in this chapter. After an introduction of the library's architecture the implementation of its subsystems is described in detail.

4.1 Architecture

The architecture of the ray tracing library is based on the idea of supplying a collection of geometry containers that can be combined to create a fast ray tracing system with support for static and dynamic geometry. C++ is the language of choice for the object–oriented implementation because it enables the use of SIMD instructions through intrinsics in up–to–date compliers and allows for low–level management of data structures (i.e., specification of alignment and reinterpretation of pointers as integer values), both of which are crucial to achieving good run–time performance.

Following the previously discussed design of the library kd–trees are used for static geometry and are created by a builder that employs a SAH sweeping–based construction approach to create high–quality kd–trees. Objects of the class `StaticKdTree` are immutable and can be serialized to disk for later reuse. It is therefore possible to use the builder supplied by the library in preprocessing tools separate from the actual rendering application.

Containers for dynamic geometry (`DynamicGeometry`) can be instantiated directly and expose access to resizable buffers for triangle vertices and indices. When an application has finished updating a container's buffer with new geometry it has to invoke one of the container's update methods to refresh its internal bounding volume hierarchy. Refitting and selective rebuilding are usually the updating scheme of choice to keep updating times low. However, if triangles are added to the container or removed from it a full rebuild is always required in order to allow for resizing of internal data structures.

Dynamic geometry containers correspond to the lower level of the previously introduced two–level acceleration structure. The class `DynamicScene` implements its upper level and arranges dynamic geometry containers in a BVH for fast ray traversal. It exposes methods for adding and removing containers and associating them with a transformation matrix, which can be used for rigid animations. Similar to geometry containers it is necessary to rebuild a `DynamicScene` after its list of containers was modified. Rebuilds are also required after changes to transformation matrices. Refitting or selective rebuilds, however, do

not need to be performed manually because these operations are carried out automatically after they were applied to a container at the lower–level.

All three data structures share a common interface for ray traversal, which is given in Listing 4.1. As most applications will typically use a `StaticKdTree` and a `DynamicScene` the class `SceneGroup` was introduced. It implements the ray traversal interface by forwarding method calls to two wrapped geometry containers, thereby supplying a single invocation point for ray tracing requests.

```
interface ITraceable
{
  void Trace(ray r, int flags);
  void Trace(ray rays[], int count, int flags);
  void Trace(packet p, int flags);
  void Trace(packet packets[], int count, int flags);
  void Trace(packet packets[], int count, int flags, frustum f);
}
```

Listing 4.1: *The general interface for ray traversal.*

The class `Tracer` provides a simplified interface as given in Listing 4.2 and acts as another wrapper for either an acceleration structure or a *SceneGroup*. When multiple rays are passed to a `Tracer` it automatically performs packet assembly with respect to the requirements and features of the wrapped object and invokes its fast packet–based traversal methods. The tracing results are subsequently extracted from the packets and written to the original rays.

```
class Tracer
{
  void Trace(ray r, int flags);
  void Trace(ray rays[], int count, int flags);
}
```

Listing 4.2: *Simplified ray traversal interface.*

The overall architecture of the ray tracing library is illustrated in a UML class diagram in Figure 4.1.

4.2 Basic Data Structures

4.2.1 Rays

A ray is defined by a point, which is commonly called its origin, and a direction vector. Ray objects in the ray tracing library contain fields for these two values, which should be set in the constructor. However, for practical reasons the definition of the ray structure also includes fields to capture the results of tracing operations (i.e. an identifier for the hit object, the distance of the hit point and its barycentric coordinates).

Additionally there are fields to record traversal specific data, such as the number of intersected triangles or the number of visited nodes, which is particularly useful for analyzing the complexity of a scene or the quality of an acceleration structure. Given the fact that collection of this data introduces a certain computational overhead that reduces ray tracing performance, it is possible to disable this functionality and mask the associated fields through a preprocessor directive.

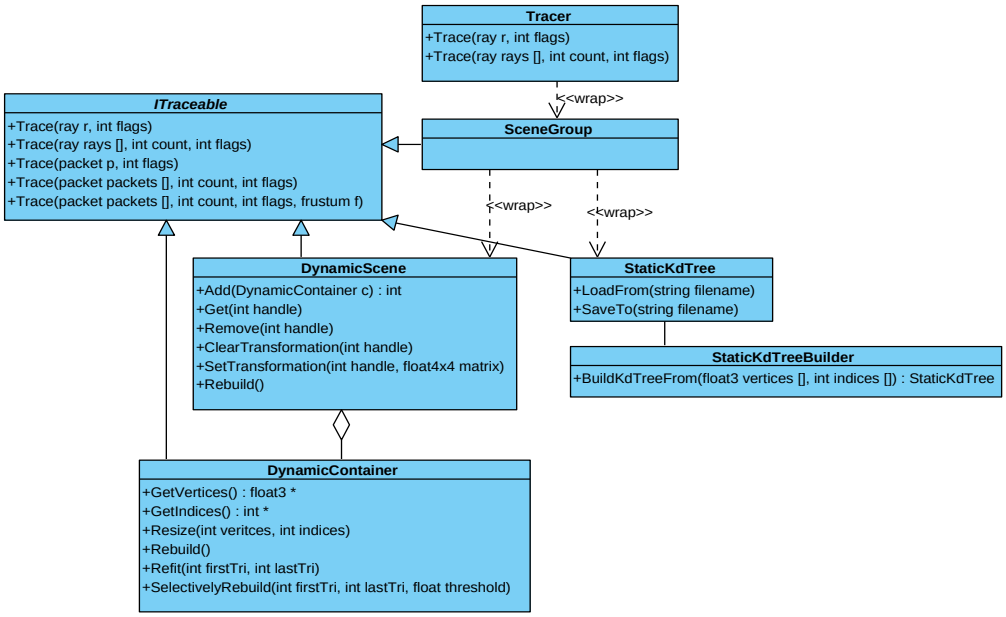


Figure 4.1: *A UML class diagram for the ray tracing library.*

```
struct ray
{
  union
  {
    struct
    {
      int triId;
      float distance, beta, gamma;
    };
    __m128 results;
  } hit;

  float3 origin, direction;

#ifdef RECORDRAYSTATS
  struct stats
  {
    int visitedInnerNodes, visitedLeaves;
    int intersectedTriangles;
    int splitNodes;
  } staticStats, dynamicStats;
#endif

  struct
  {
    void *data;
  } user;
}
```

Listing 4.3: *Rays have not only an origin and a direction but also fields for storage of traversal results and statistics.*

An additional field named `user.data` is included to enable applications to associate custom data with each ray, yielding the final definition of the structure as given in Listing 4.3.

The fields for the results of intersection tests are laid out to fit into a 128–bit SSE register and can therefore be manipulated with vector instructions. It is important to note that SSE requires data to be aligned on 16–byte memory boundaries, which is performed automatically by the compiler only for ray objects that are allocated on the stack. If rays need to be created on the heap the programmer is responsible for guaranteeing proper alignment. This can be accomplished by replacing the traditional `malloc()` function with a function that allocates aligned memory (e.g. `_mm_malloc()` that is supplied by most compilers).

4.2.2 Packets

Packets serve as temporary containers for exactly 4 rays, which corresponds to the native vector size of the employed SSE instruction set. They are not created by the programmer directly. Instead the class `Tracer` is responsible for assembling packets from individual rays depending on the requirements of the configured ray tracing system. It serves as a front–end for the acceleration structures that are supplied by the library and relieves the programmer from ensuring that only groups of rays are created that can be processed together by the implemented traversal algorithms (e.g. by sorting ray according to their direction vectors for kd–tree traversal).

The definition of the packet structure is very similar to the previously given ray structure. In particular a packet also contains fields for the origins and the directions of its 4 rays. Additionally the per–component multiplicative inverses of the direction vectors are stored, which are useful in the context of ray–plane intersections that are performed frequently during the traversal of acceleration structures. By pre–calculating $(1/dir.x, 1/dir.y, 1/dir.z)$ for each ray it is possible to replace divisions with faster multiplications, which may increase run–time performance considerably.

Fields for hit data are also part of the definition of a packet albeit again in vector form for all 4 rays. Fields for traversal statistics, however, are not replicated for each ray and capture data on the granularity level of entire packets. Nevertheless this results in no loss of information because all rays in a packet are subject to the same operations when a packet–based traversal algorithm is employed.

Additionally packets contain references to the 4 rays that they wrap. These references are used to update the rays with traversal results and statistics after acceleration structures were traversed with the packet. Listing 4.4 gives the final definition of the packet structure.

4.2.3 Frustum of Rays

The multi level ray tracing algorithm by Reshetov *et al.* [44] uses the frustum of a group of rays to quickly discard entire sub–trees during kd–tree traversal. The traversal algorithm for bounding volume hierarchies by Wald *et al.* [57] is based on the same idea to exploit ray coherence. Packet assembly by the `Tracer` class therefore encompasses the computation of a frustum for rays to support the use of these traversal schemes, which is performed based on the algorithm devised by Boulos *et al.* [5]. The algorithm operates on rays that do not necessarily need to share a common origin and is therefore also useful in the context of tracing secondary rays. After the parameters for the frustum's 4 bounding planes have been

```
struct packet
{
  vector3 origin;
  vector3 direction;
  vector3 invdir;

  struct
  {
    __m128 triId;
    __m128 distance, beta, gamma;
  } hit;

  ray *rays[4];

#ifdef RECORDRAYSTATS
  ray::stats staticStats, dynamicStats;
#endif
}
```

Listing 4.4: *Packets are temporary wrappers for 4 rays and have similar fields albeit in vector form.*

computed, they are stored in a SIMD friendly form devised in the following to allow for an efficient culling of axis–aligned bounding boxes.

Culling of an AABB involves proving that all of its 8 vertices are on the outside of at least one of the frustum's bounding planes. This can be approached by making sure that the signed distances of all vertices to a given plane with an outward facing normal are positive. However, Greene [12] proposed that testing a single vertex suffices to determine whether the entire AABB is located on the outside of a plane or not. The so–called *n*–vertex is the vertex of an AABB that lies farthest in the negative direction of a given plane's normal vector. If this vertex is on the outside of a plane then the other vertices are outside, too, as illustrated in Figure 4.2.

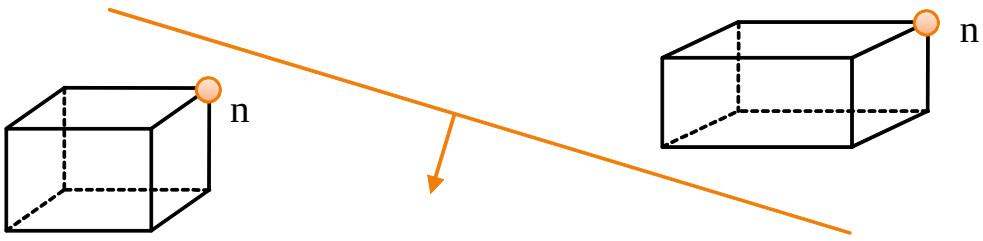


Figure 4.2: *The n–vertex of an AABB is the vertex laying farthest in the negative direction of a plane's normal vector.*

Based on the representation of an AABB with two extremal points the position of the *n*–vertex can be composed component–wise based on the signs of the plane's normal vector. For positive values of the normal vector the lower extremal value in that axis is used for the *n*–vertex, otherwise it is the upper extremal value. By separating the positive from the negative values of a plane's normal, as proposed by Reshetov *et al.* [44], the signed distance of the *n*–vertex can be calculated directly as demonstrated in Equation 4.1.

$$\begin{aligned} dist = \quad & max(0,n_x) * lower_x + min(0,n_x) * upper_x + \\ & max(0,n_y) * lower_y + min(0,n_y) * upper_y + \\ & max(0,n_z) * lower_z + min(0,n_z) * upper_z - d \end{aligned} \tag{4.1}$$

By pre-computing $max(0,n)$ and $min(0,n)$ for each plane and storing them in vector form along with the planes' distance terms d, culling of AABBs can be implemented efficiently with SIMD instructions to evaluate all of the frustum's 4 bounding planes in parallel.

4.3 Kd–Trees for Static Geometry

Kd–trees for static geometry are implemented in the class `StaticKdTree`. It serves as an immutable container for both the triangles and the acceleration structure and supports methods for traversal with rays and packets. `StaticKdTree` objects are typically created by the `StaticKdTreeBuilder` based on a set of triangles. Serialization to files for later reuse is also supported, which allows the creation of static kd–trees in preprocessing tools.

4.3.1 Construction of Kd–Trees

The `StaticKdTreeBuilder` is responsible for the construction of static kd–trees in the ray tracing library and provides a simple to use interface for this task, which accepts a list of triangles and returns an instance of the `StaticKdTree` class. The builder employs the surface area heuristic to create high–quality kd–trees and is based on the plane sweeping approach by Wald and Havran [55].

The first step performed by the builder is the initialization of sorted per–axis lists with opening, closing and planar events for the triangles. The implementation uses an OpenMP parallel for–loop to perform event creation and list sorting for each axis in parallel on a maximum of three processing units. At the same time it determines the axis–aligned bounding box for all triangles.

Construction of the kd–tree is then carried out in top–down manner by recursively invoking a partitioning function, which returns a new sub–tree based on a set of inputs:

- An AABB describing the active volume,
- A list of references to the triangles in the volume,
- Event lists for each axis,
- The current depth and
- The remaining number of backtracking points.

The function initially determines if all pre–conditions for further subdivision are met by checking that the number of triangles is greater than zero and that the current depth does not exceed the maximal allowed depth. If this is the case then the splitting plane with the minimal associated costs is identified using the plane sweeping algorithm, otherwise a leaf node is returned. Given a valid plane the termination criterion is then evaluated to determine if splitting the active volume is beneficial to the quality of the kd–tree. In case an abort is suggested, a backtracking point may be used to continue the subdivision albeit with

a subsequent evaluation and potential backtracking to undo it if it does not exhibit lower costs than a leaf.

Splitting a node involves a series of operations:

- The given bounding volume is split into two new AABBs using the splitting plane.
- Triangles are partitioned into two subsets, which are not necessarily disjoint, based on a left/right–classification of their positions relative to the splitting plane.
- Two new sets of per–axis events lists are created in a more involved procedure. First of all the events for triangles that intersect with the splitting plane are deleted and the lists are spliced into a left and a right event list for each axis retaining the sorting order. Events for plane–straddling triangles are then recreated based on clipped representations of the triangles (e.g., created using the Sutherland–Hodgeman clipping algorithm [51]). These events are sorted and inserted into the left and right lists using a single mergesort iteration, yielding again sorted event lists that can be used in subsequent subdivision steps.

After these steps an inner node is allocated with the sub–trees resulting from invoking the partitioning function for the left and right data sets assigned as children.

The construction algorithm can be parallelized by spawning a second thread to handle subdivision of either the left or the right data set. However, care must be taken not to deplete the system's resources by spawning an exponential number of threads through kd–tree construction. In fact multi–threading is most efficient if there is a single active thread running on each processing unit. Such behavior can be implemented by introducing a shared counter to control the number of threads. It is initialized to reflect the available processing units. A thread can only be spawned if the counter is greater than zero, in which case it is decremented using an atomic operation so not to require locking. Upon termination of the thread the counter is incremented again.

Although it might seem that this threading scheme involves the creation and destruction of a large number of threads, it is actually very efficient. This is due to the top–down approach to kd–tree construction where all threads are spawned on high levels close to the root node and live throughout the entire construction of their respective sub–trees.

4.3.2 Memory Layout

Kd–tree Nodes

Kd–tree nodes can be packed into 8 bytes by employing the data structure as given in Listing 4.5 and storing the nodes contiguously in memory (i.e. in an array).

The first 4 byte sized field of a leaf node is a signed integer that stores the number of triangles contained in the leaf. The second field is an offset into an array of indices where the actual references to the triangles are stored sequentially.

Inner nodes contain an axis–aligned splitting plane and references to their children. Despite the fact that the kd–tree is a binary hierarchy it is possible to describe the whole tree with only a single reference per inner node by exploiting the arrangement of nodes in memory. If nodes are stored in depth–first order in a contiguous block of memory it suffices to remember the address of the right child because the left child always follows its parent node. Another approach is to store siblings in pairs and to remember the address of the first node. Although both approaches are equally powerful with regard to their ability to describe

```
union node
{
  struct
  {
    int numberOfIndices; // number of triangle indices
    int offsetToIndices; // offset to array of indices
  } leaf;

  struct
  {
    int offsetAndAxis;   // 0..2 axis, 2..30 offset
    float splitPos;      // position of the splitting plane
  } inner;
}
```

Listing 4.5: *Kd–tree nodes can be packed tightly into a structure of 8 bytes.*

kd–trees, storing nodes in pairs has the added advantage that children can be accessed in C–style array notation, therefore yielding easier to read code. The `StaticKdTreeBuilder` therefore allocates an array for all nodes and arranges siblings in pairs, as depicted in Figure 4.3.

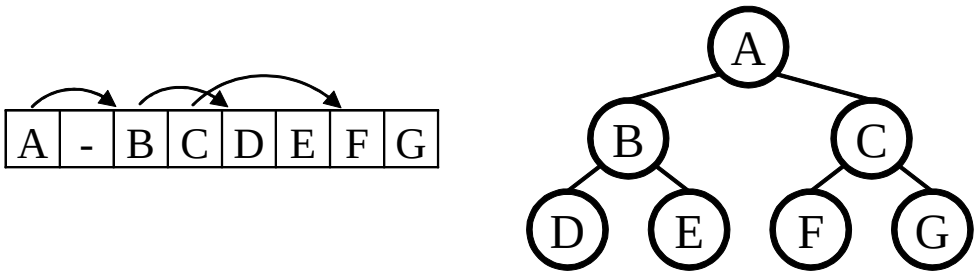


Figure 4.3: *By storing siblings as a pair in contiguous memory it is possible to access both nodes based on the address of the first node.*

By storing offsets to child nodes in multiples of bytes, the lower two bits of the integer representation of an offset are always zero due to nodes having a size of 8 bytes. These two bits suffice to store the axis of the splitting plane. The position of the plane is a single precision floating point number and is stored in the second 4 byte sized field of the inner node.

In order to be able to discern between leaves and inner nodes based on their representation in memory an additional marker has to be introduced. A viable solution is the use of the most significant bit (MSB) of the first field. The MSB corresponds to the sign bit of signed integers and is never set in leaf nodes because they cannot contain a negative number of triangles. It can therefore be freely used to signal that a node has to be interpreted as an inner node. A potential drawback of this approach is the reduction of the range of offsets to child nodes. However, in 32–bit systems the operating system typically reserves the upper half of the address space and therefore limits the available memory for nodes anyway.

Triangles

The ray–triangle intersection test devised by Kensler and Shirley [28] can be implemented efficiently by precomputing data. In particular the algorithm expects triangles to be defined not based on three vertices but on a single point and two edge vectors. This transformation is carried out during the construction of a `StaticKdTree` object, as depicted in Figure 4.4. Additionally the face normal vector is calculated to save on an additional cross product operation at run–time, which increases memory requirements from 32 bytes to 48 bytes per triangle.

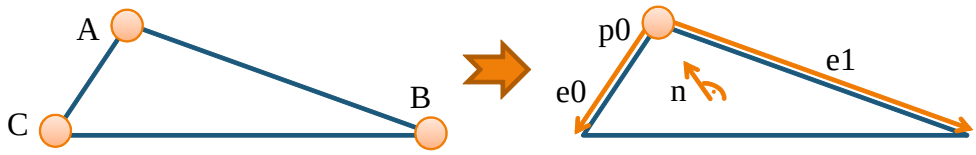


Figure 4.4: *By pre–computing data the ray–triangle intersection test can be optimized.*

Indices

Packing kd–tree nodes into 8 bytes has the side effect that it is not possible to store references to more than a few triangles directly in the leaves. Using references of 2 bytes in size limits the maximum of triangles in a given scene to 2^{16}, which is not sufficient even for moderately complex scenes. Additionally it requires a modification of the construction algorithm to continue subdivision of nodes until they contain no more than 4 triangles regardless of the termination criterion, which may reduce ray tracing performance.

In order to be able to reference a potentially large number of triangles in a leaf of a kd–tree an extra data structure is required. This data structure has the form of an array that stores references to triangles (e.g. with 4 bytes per reference). As described previously, leaves then contain an offset into this array and a counter that specifies the length of the range beginning at the offset.

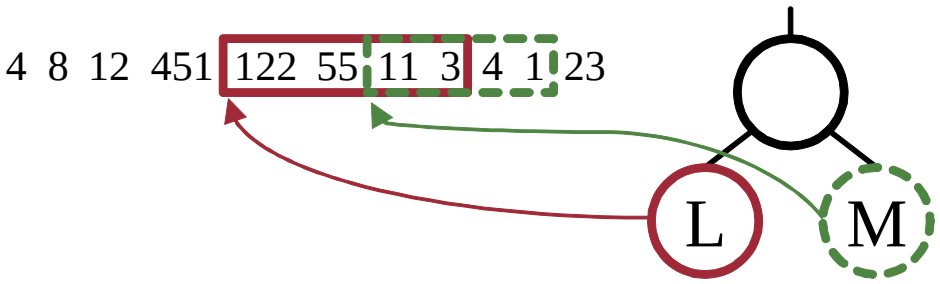


Figure 4.5: *The overall number of indices can be reduced by merging the per–leaf lists and allowing them to overlap.*

By allowing ranges to overlap, as can be seen in Figure 4.5, the size of the data structure can be reduced. Despite the fact that combining lists into a single list of minimal size is an np–complete problem, good results can be attained in polynomial run–time by employing Algorithm 6.

The algorithm creates an empty global index list and performs traversal of the kd–tree in depth–first order. When a leaf is visited its local index list is sorted (e.g., ascendingly) and it

is determined whether its contents are already present in the global list in exactly the same order or not. If no occurence is found the contents of the leaf's list are simply appended to the global list. Otherwise the existing range of indices is reused. In both cases the local list is dropped and the leaf is updated to reference a section of the global list.

Algorithm 6 Index list compaction

```
function TRAVERSE(node n, list indices)
    if n is leaf then
        sort n.indices /* e.g., ascendingly */
        if indices contains n.indices then
            update n to reference existing indices
        else
            append n.indices to indices
            update n accordingly
    else
        Traverse(n_left, indices)
        Traverse(n_right, indices)
function COMPACTINDICES(tree T) returns list
    indices = ∅ /* initialize the global index list */
    Traverse(T_root, indices)
    return indices
```

The main computational effort of this algorithm is the search for an existing set of indices in the global list. To optimize this aspect the use of string searching algorithms like the Boyer–Moore algorithm [6] is tempting, which finds strings with m characters in a text of length n in $\Omega(n/m)$. However, in the context of this thesis it has been determined that a brute–force approach is almost always faster. This is due to the fact that leaves typically reference only a few triangles when a SAH–based construction approach is employed. Therefore the m is small and the theoretical run–time benefits of the Boyer–Moore algorithm are outweighed by its organizational overhead. For a selection of scenes, which will be described in chapter 5, the distribution of indices can be seen in Figure 4.6.

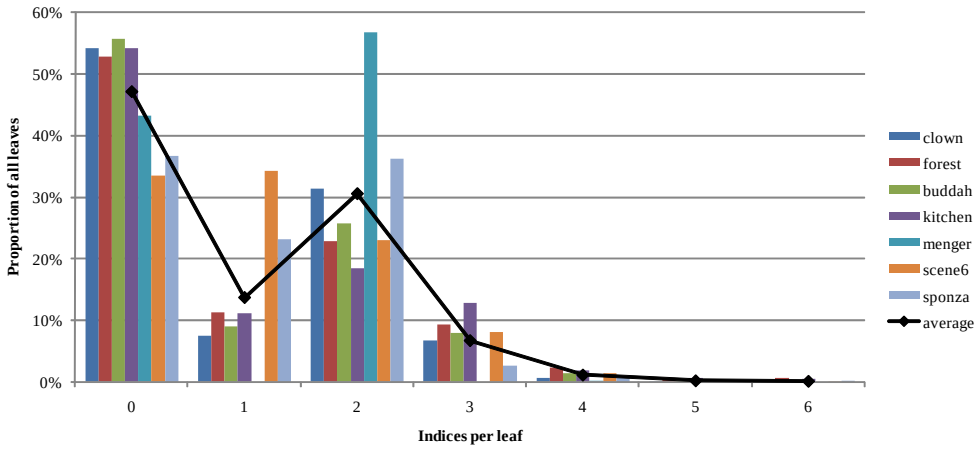


Figure 4.6: *Leaves typically contain only few references to triangles when a SAH–based construction algorithm is used.*

Nevertheless the run–time efficiency of the algorithm can be improved based on properties of the traversal order. By traversing a kd–tree in depth–first order leaves in close spatial proximity, which are likely to contain shared triangles, are visited in sequence. Appending

to the global index list increases the probability of finding an existing set of indices towards the end of the list. The algorithm can therefore be optimized by restricting the range of the brute force search to encompass only the last n entries without sacrificing good packing of indices.

As can be seen in Figure 4.7 only mild growths of the global index list result from restricting search ranges to widths of $n \in [100; 1000]$. The overhead of index list compaction, however, is reduced considerably and increases kd–tree construction times by less than 8% on average for the given selection of scenes.

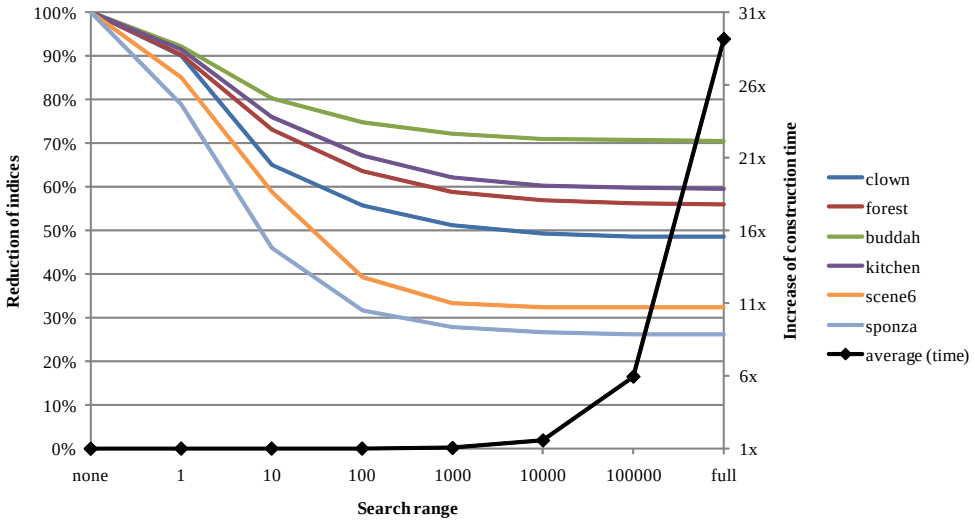


Figure 4.7: *The run–time overhead of index list compaction can be reduced significantly with little influence on the quality of the generated list by limiting the search range*[1].

4.3.3 Ray Traversal

Traversal of the kd–tree with both packets and single rays is implemented iteratively to avoid the overhead of recursive function calls. For this purpose an explicit stack storing nodes along with the near and far parameter intervals for the rays is required. The necessary amount of memory is allocated on the stack based on the maximal allowed kd–tree depth, which has advantage of being inherently thread–safe compared to the use of heap–allocated memory that needs to be replicated for each active thread in the ray tracing system.

The selection of the near and the far node in each traversal step is based on a bit–field containing the index of the near node for each axis. By storing siblings in pairs with the node corresponding to the lower volume first, the near node has the index 0 for rays pointing in the positive direction of an axis and the index 1 otherwise. This bit–field can therefore be composed directly from the sign–bits of the ray's direction vector's components, as given in Listing 4.7. For packets it is guaranteed that all rays have direction vectors with equal signs, which allows the composition of the bit–field based on an arbitrarily selected (i.e., the first) ray.

[1] The Menger–Sponge scene was omitted from the graph because it does not benefit from index list compaction. Its regular structure allows for perfect partitioning of triangles, which are therefore referenced exactly once. Skipping the compaction operation for faster kd–tree construction is advisable if the number of indices is only marginally higher than a scene's primitive count.

```
void Trace(ray &r, int flags)
{
  float3 invDir = 1.0f / r.direction;
  float nearDist = 0.0f, farDist = (flags & ShadowRays) ?
    min(1.0f, r.hit.distance) : r.hit.distance;
  if(!ClipToSceneBox(nearDist, farDist, r, invDir))
    return; // the ray missed the scene's AABB

  Mailbox mailbox;
  int stackIdx = 0, signMask = GetSignMask(r.direction);
  float nears[MaxDepth], fars[MaxDepth];
  node *nodes[MaxDepth], *cur = _nodes;
  while(true)
  {
    while(!cur->IsLeaf()) // traverse down to a leaf
    {
      int a = cur->inner.GetAxis();
      float d = (cur->inner.splitPos - r.origin[a]) * invDir[a];
      node *children = cur->inner.GetChildren();
      int nearIndex = (signMask >> a) & 0x1;
      if(d >= nearDist)
      {
        if(d <= farDist)
        {
          nodes[stackIdx] = children + (nearIndex ^ 1);
          nears[stackIdx] = d; fars[stackIdx++] = farDist;
          farDist = d; // visit the near child first
        }
        cur = children + nearIndex;
      }
      else
        cur = children + (nearIndex ^ 1);
    }

    int *indices = cur->leaf.GetIndices();
    for(int i = 0; i < cur->leaf.numberOfIndices; ++i)
    {
      if(!mailbox.SkipIntersectionTestWith(indices[i]))
        _triangles[indices[i]].IntersectWith(r, indices[i]);
    }

    if(r.hit.distance <= ((flags & ShadowRays) ? 1.0f : farDist))
      break; // early out

    if(stackIdx > 0) // pop the next node from the stack
    {
      cur = nodes[--stackIdx];
      nearDist = nears[stackIdx]; farDist = fars[stackIdx];
    }
    else
      break;
  }
}
```

Listing 4.6: *Single ray traversal of kd–trees.*

```
int GetSignMask(float3 v)
{
  union { float f; unsigned int i; } tmpx, tmpy, tmpz;
  tmpx.f = v.x; tmpy.f = v.y; tmpz.f = v.z;
  return ((tmpx.i >> 31) & 0x1) |
         ((tmpy.i >> 30) & 0x2) |
         ((tmpz.i >> 29) & 0x4);
}
```

Listing 4.7: *The index of the near node in each axis is extracted from the sign–bits of the ray's direction vector.*

When ray traversal reaches a non–empty leaf, the referenced triangles are intersected with the ray using the algorithm devised by Kensler and Shirley [28]. The ray is then updated if an intersection point closer to its origin is found, which encompasses writing the zero–based index of the triangle, the hit distance, and the barycentric coordinates of the intersection point to the corresponding fields in the ray. In order to be able to skip triangles that were tested for intersection with a given ray or packet in another leaf previously an inverse mailbox is used.

Mailboxes keep track of intersection tests that have been carried out during traversal. In their original form they are implemented by assigning each ray or packet a unique identifier and writing it to a per–primitive field after an intersection test was performed. The field therefore always contains the identifier of the ray/packet that it was intersected with last, which allows for skipping of primitives with a simple comparison of ids. Within a multi–threaded ray tracer this has the disadvantage of requiring such a field for each active thread to avoid having concurrent traversal operations overwrite each other's records, which would reduce the efficiency of the technique.

Inverse mailboxes are based on storing a list of intersected primitives for each ray or packet, which therefore do not need to be assigned unique identifiers. By allocating this list on the program stack the technique can be implemented in a thread–safe way and is useful also in multi–threaded environments. Although the number of intersection tests is typically low on average for kd–trees, which allows to keep a perfect record of all intersections with little memory overhead on average as well, it is preferable to keep track of intersections based on hashing to guarantee $O(1)$ behavior both in memory use and run–time. An efficient implementation of this approach encompasses the allocation of a fixed size array with 2^n entries and the insertion of primitive identifiers based on their lower n bits. Arrays with 32 or 64 entries typically suffice to reduce hash collisions to acceptable levels, with larger arrays having no measurable positive effects on ray traversal performance for the selection of scenes introduced previously.

The implementation of single ray kd–tree traversal is given in Listing 4.6. In addition to standard ray traversal, which is used to find the closest point of intersection of a ray with geometry, the function has special support for shadow rays.

Shadow rays are traced as secondary rays from a point on the surface of an object to a light source. They are used to determine whether the point is illuminated or in shadow due to the existance of an object blocking the incoming light. To determine that a point is in shadow it suffices to find any intersection between the point and a light source. Furthermore all potential intersections behind the light can be ignored as they have no effect on the illumination of the point. Based on these two observations the early–out test and the initial

```
void Trace(packet &p, int flags)
{
  __m128 nearDist4 = Zero4, farDist4 = (flags & ShadowRays) ?
    _mm_min_ps(One4, p.hit.distance.f4) : p.hit.distance.f4;
  int active = ClipToSceneBox(nearDist4, farDist4, p);
  if(active == 0) return; // all rays missed the scene's AABB

  int terminated = active ^ 0xf; Mailbox mailbox;
  int signMask = GetSignMask(p.rays[0]->direction);
  int stackIdx = 0; __m128 nears[MaxDepth], fars[MaxDepth];
  node *nodes[MaxDepth], cur = _nodes;
  while(true)
  {
tr: while(!cur->IsLeaf()) // traverse down to a leaf
    {
      int axis = cur->inner.GetAxis();
      __m128 d = _mm_mul_ps(_mm_sub_ps(_mm_set1_ps(cur->inner.splitPos),
        p.origin[axis].f4), p.invdir[axis].f4);
      node *children = cur->inner.GetChildren();
      int nearIndex = (signMask >> axis) & 0x1;
      int near = _mm_movemask_ps(_mm_cmpge_ps(d, nearDist4)) & active;
      if(near != 0)
      {
        int far = _mm_movemask_ps(_mm_cmple_ps(d, farDist4)) & active;
        if(far != 0)
        {
          nodes[stackIdx] = children + (nearIndex ^ 1);
          nears[stackIdx] = d; fars[stackIdx++] = farDist4;
          farDist4 = d; active = near; // visit the near child first
        }
        cur = children + nearIndex;
      }
      else
        cur = children + (nearIndex ^ 1);
    }

    int *indices = cur->leaf.GetIndices();
    for(int i = 0; i < cur->leaf.numberOfIndices; ++i)
    {
      if(!mailbox.SkipIntersectionTestWith(indices[i]))
        _triangles[indices[i]].IntersectWith(p, indices[i]);
    }

    terminated |= _mm_movemask_ps(_mm_cmple_ps(p.hit.distance.f4,
      (flags & ShadowRays) ? One4 : farDist4)) & active;
    if(terminated == 0xf) break; // early out test

    while(stackIdx > 0) // pop the next node from the stack
    {
      nearDist4 = nears[--stackIdx];
      farDist4 = _mm_min_ps(p.hit.distance.f4, fars[stackIdx]);
      active = _mm_movemask_ps(_mm_cmple_ps(nearDist4, farDist4)) &
        ~terminated;
      if(active != 0) { cur = nodes[stackIdx]; goto tr; }
    }
    break;
  }
}
```

Listing 4.8: *Traversal of kd–trees with a single packet.*

far distance are modified to enable faster tracing of shadow rays, which are assumed to have non–normalized direction vectors extending from the point in question to the light source.

Listing 4.8 presents the implementation of kd–tree traversal with a single packet. It is a vectorization of the single ray traversal routine and therefore exhibits similar control flow and operations albeit in SIMD instruction form. In particular masks are used instead of flags to capture and evaluate the results of comparisons.

Traversal begins by intersecting all rays in the given packet with the scene's axis–aligned bounding box to determine the initial parameter intervals. An additional result of this operation is a mask that records which rays hit the scene's AABB and should therefore be subject to traversal. If no rays can be considered "active" with regard to this then traversal is skipped and the function returns with no ray–triangle intersections.

During traversal of the kd–tree this mask is updated with each step to reflect the rays that are active in the current sub–tree, which is necessary to be able to accurately determine the nodes that need to be visited. In addition to the active mask a "termination" mask is maintained that records the rays for which their respective closest intersection with geometry has already been found. It is updated subsequently to the intersection tests in a leaf and evaluated to assess whether the early–out condition of traversal has been met (i.e., all rays have been terminated) or not.

Popping nodes from the stack is a slightly more involved operation in packet traversal compared to single ray traversal. When a node and a parameter interval is taken from the top of the stack the interval's far value is updated to factor in the distances to the closest hit points. The active mask is then recomputed and combined with the termination mask to restrict the set of active rays precisely to those rays that benefit from visiting the retrieved node in that closer intersection points might be found. If this yields a mask with zero active rays another interval–node pair is taken from the stack until traversal can either be aborted due to an empty stack or resumed.

Traversal for multiple packets is implemented by invoking single packet traversal for each packet individually. This delievers better results on average than the use of the multi level ray tracing algorithm (MLRTA), which was also implemented but deactivated, due to two aspects:

- MLRTA requires coherent ray packets to be efficient and is therefore unsuitable to secondary rays, for which its overhead results in reduced run–time performance compared to standard packet traversal.
- MLRTA performs best when traversing kd–trees that are built based on a modified cost model favoring nodes with large volumes. Large volumes reduce the probability that packets take different paths through the kd–tree and enable entry point search to find starting points deep in the tree. However, modifying the cost model in this way reduces the run–time performance of standard packet traversal and therefore the efficiency of tracing secondary rays in our system.

4.4 Two–Level Dynamic Acceleration Structure

The previously introduced two–level acceleration structure is based on the concept of managing raw geometry in the lower level and arranging geometry chunks on the scene level by means of instancing and transformation in the upper level.

The class `DynamicGeometry` implements the lower level of the acceleration structure and acts as a container for raw geometry. Geometry is represented as an array of vertex positions and a list of triangle indices connecting the points to triangles. For this purpose the class provides functions for resizing the internal buffers for vertex positions and triangle indices and also for acquiring pointers to the buffers to allow for writing and updating data.

The upper level of the two–level data structure is implemented in `DynamicScene`, a class that acts as a collection of geometry containers. It provides a method for adding a reference to a `DynamicGeometry` object, which returns a handle to the thereby created instance of the geometry on the upper level. Based on this handle it is then possible to associate the instance with a transformation matrix to position its geometry or to remove the instance again.

To enable fast ray tracing both classes maintain an internal acceleration structure for their contents, which was chosen to be bounding volume hierarchies in the context of this thesis. Leaves of BVHs in the lower level therefore contain references to triangles, whereas geometry containers are referenced in the leaves of the upper level BVH. This difference in contents affects both the traversal and the construction algorithm of the respective bounding volume hierarchies, which makes it difficult to create a single BVH implementation directly accommodating the two levels.

In order to facilitate the creation of a single BVH implementation it is necessary to encapsulate the differences in functionality and data between the upper and the lower level:

- Although ray traversal of the BVH is identical the actual operations carried out when a leaf is visited differ between the lower level, where ray–triangle intersection tests are performed, and the upper level, where the ray is subject to transformations and further traversal of the referenced geometry containers. By encapsulating this leaf–specific behavior in a functor a single traversal algorithm (for each type, rays and packets) can be implemented and customized based on the level of the acceleration structure it is used at.
- BVHs of the lower level partition triangles, which stands in contrast to the partitioning of geometry container instances on the upper level. However, both types of entities have in common that an axis–aligned bounding box can be computed for them, which is all that is needed for the implementation of most construction algorithms. By having both classes, `DynamicGeometry` and `DynamicScene`, manage an array of AABBs corresponding to their contents a single construction algorithm can be supported. This also applies to updating schemes like refitting and selective restructuring, which operate on bounding boxes only, too.

In the following the implementation of the bounding volume hierarchy is described, with subsequent discussion of the integration with the classes of the two–level acceleration structure.

4.4.1 Bounding Volume Hierarchy

Memory Layout

Bounding volume hierarchies are an object partitioning scheme and reference each primitive exactly once. Construction can therefore be performed in–place by reordering the primitives in the array they are stored in and building a node hierarchy with leaves referencing runs of primitives. To reduce the memory overhead of moving primitives during construction an

additional indirection in the form of an index list can be introduced, as already known from the discussion of the kd–tree implementation.

Each node in a BVH primarily represents an axis–aligned bounding box, which can be described with its two extremal points that are called `lower` and `upper` in the definition of the node structure as given in Listing 4.9. The hierarchy of nodes is established by having inner nodes contain references to their children. Following the allocation scheme of the kd–tree, siblings are stored consecutively in memory, which allows access to both nodes through a single pointer to the first node. The associated offset from the current inner node to its children is stored in the field `offsetToChildren`. In addition to this each node also stores the offset to its parent, which is required for bottom–up traversal of the BHV as performed in updating algorithms. Although the root node has no parent the field `offsetToParent` is still useful in that it allows marking of the BVH root (by assigning the field an invalid offset, i.e. zero) to signal the end of a bottom–up traversal sequence.

```
struct node
{
  float3 lower, upper;
  int firstObj, lastObj;
  int offsetToParent;

  // INNER: 7 .. 0 near index
  // LEAF: 31 .. leaf node, 30 .. final leaf, 29 .. 0 depth
  int nearMask;

  int offsetToChildren; // INNER: offset to the children
  float splitCosts;     // INNER: original costs for splitting
}
```

Listing 4.9: *Structure definition of a BVH node that is 48 bytes in size.*

The fields `firstObj` and `lastObj` are used as offsets into the index array, which is a necessity in leaves in order to be able to carry out ray–triangle intersection tests during traversal. However, keeping offsets to the indices in all nodes is convenient in a dynamic environment because it allows updating algorithms to quickly determine the set of primitives in a given sub–tree. Selective restructuring uses this information to reset an inner node to a leaf for subsequent rebuilding if its partitioning quality drops below a specified threshold. To facilitate this an additional field (`splitCosts`) records the SAH based costs of the initial split.

Ordered traversal of the BVH is implemented based on an evaluation of the spatial relationship of nodes at construction time, which will be discussed later in detail. 8 cases depending on the signs of the components of a ray's direction vector are handled, as illustrated in Figure 4.8 for two dimensions (4 cases). For this purpose the lowest 8 bits of the field `nearMask` are used in each inner node to store the index of the near node.

The same field is used to store the depth of leaves, which is necessary in order to stop incremental construction from exceeding the maximal allowed depth of the BVH. `nearMask` is also used to distinguish between inner node and leaves based on the top–most bit, which is set only for leaves. Leaf nodes are classified further with a second bit as either final or non–final with regard to potential splitting as part of lazy construction.

```
void Traverse(ray &r, float3 invDir, int flags, int signMask, LeafOpType
    leafOp)
{
  node *cur = _nodes;
  if(cur->IntersectWith(r, invDir))
  {
    int stackIdx = 0;
    node *nodes[MaxDepth];
    while(true)
    {
      if(cur->IsLeaf())
      {
        if(cur->CanSplit())
        {
          switch(SplitNode(cur))
          {
          case Finalized: break; // done, this is leaf
          case Split: goto inner; // this has become an inner node
          case InProgress:
            if(stackIdx > 0) // we have other nodes to visit
              swap(cur, nodes[stackIdx - 1]);
            continue; // try again in next iteration
          }
        }
        leafOp(r, invDir, cur->firstObj, cur->lastObj, flags);
      }
      else
      {
inner:  node *children = cur->GetChildren();
        bool left = children[0].IntersectWith(r, invDir);
        bool right = children[1].IntersectWith(r, invDir);
        if(left)
        {
          if(right)
          {
            // push far child onto the stack and traverse near sub-tree
            int nearIndex = cur->GetNearIndex(signMask);
            nodes[stackIdx++] = children + (nearIndex ^ 1);
            cur = children + nearIndex;
          }
          else
            cur = children;
          continue;
        }
        else if(right)
          { cur = children + 1; continue; }
      }

      // pop a node from the stack or exit
      if(stackIdx > 0) cur = nodes[--stackIdx]; else break;
    }
  }
}
```

Listing 4.10: *Traversal of a BVH with a single ray.*

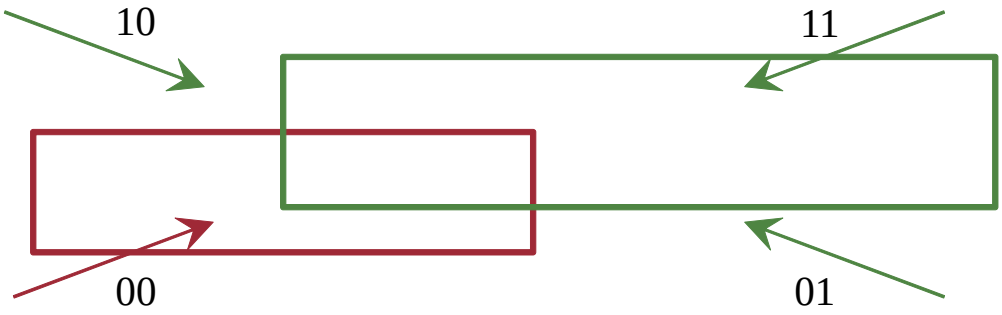


Figure 4.8: *Ordered traversal of a BVH is based on a pre–evaluation of the near node for rays based on the signs of their direction vectors' components.*

Ray Traversal with Lazy Construction

Lazy construction is a technique that builds an acceleration structure on demand during ray traversal. Whenever a leaf node is visited, which has not been marked as final, the construction algorithm is invoked to either split the node or turn it into a final leaf if further subdivision is not beneficial. Ray traversal and construction are therefore not separated but integrated into a single process.

Listing 4.10 presents an iterative implementation of single ray traversal. The core operation is the detection of intersections between the ray and the bounding boxes of the nodes, which can be implemented efficiently using Smit's algorithm [48]. The results of these intersection tests determine which sub–trees of a given inner node need to be traversed. In case both children need to be visited it is the near sub–tree that is traversed first in order to be able to find close intersection points and skip traversal of parts of the far sub–tree that are farther away.

When a leaf is visited it is checked whether it has been marked final or not. In the first case the leaf operation supplied for traversal is carried out for the referenced entities in the leaf. In the latter case the construction algorithm is invoked, which returns a status code describing the modifications made to the BVH:

- If the leaf was marked as final the traversal algorithm may continue treating the node as a leaf, which involves execution of the specified leaf operation.
- If the node was split traversal has to continue treating the current node as an inner node, which involves determining which of its two children need to be visited and in what order.
- If no operation was performed the type of the node has to be reevaluated by the traversal algorithm.

The third case can occur in multi–threaded environments only in the situation where more than one thread try to modify the given leaf node, which is an operation that involves writing to shared memory and therefore has to be protected against race conditions. An efficient way of handling this scenario is to resume traversal of another node on the traversal stack if possible and continue with the current leaf node later on when the construction algorithm active in another thread has most likely been finished.

Traversal of a BVH with a single packet exhibits identical control flow and differs only in the use of vector instructions for processing of multiple rays in parallel compared to

```
void Traverse(packet packets[], int count, int flags, frustum f,
  int signMask, LeafOpType leafOp)
{
  node *cur = _nodes;
  int first = cur->IntersectWith(packets, f, 0, count);
  if(first < count)
  {
    int stackIdx = 0; node *nodes[MaxDepth]; int firsts[MaxDepth];
    while(true)
    {
      if(cur->IsLeaf())
      {
        if(cur->CanSplit())
        {
          switch(SplitNode(cur))
          {
          case Finalized: break; // done, this is leaf
          case Split: goto inner; // this has become an inner node
          case InProgress:
            if(stackIdx > 0) // we have other nodes to visit
            {
              swap(cur, nodes[stackIdx - 1]);
              swap(first, firsts[stackIdx - 1]);
            }
            continue; // try again in next iteration
          }
        }
        leafOp(&packets[first], f, count - first,
          cur->firstObj, cur->lastObj, flags, MaskOp(cur));
      }
      else
      {
inner:  node *children = cur->GetChildren();
        int cfirsts[] = {
          children[0].IntersectWith(packets, f, first, count),
          children[1].IntersectWith(packets, f, first, count) };
        if(cfirsts[0] < count)
        {
          if(cfirsts[1] < count)
          {
            int nearIndex = cur->GetNearIndex(signMask);
            firsts[stackIdx] = cfirsts[nearIndex ^ 1];
            nodes[stackIdx++] = &children[nearIndex ^ 1];
            cur = &children[nearIndex]; first = cfirsts[nearIndex];
          }
          else
            { cur = children; first = cfirsts[0]; }
          continue;
          }
        else if(cfirsts[1] < count)
          { cur = children + 1; first = cfirsts[1]; continue; }
      }
      if(stackIdx > 0) // pop a node from the stack or exit
        { cur = nodes[--stackIdx]; first = firsts[stackIdx]; }
      else break;
    }
  }
}
```

Listing 4.11: *Traversal of a BVH with multiple packets based on a frustum.*

scalar instructions in the implementation of single ray traversal. Traversal with multiple packets, however, allows for an optimization of the handling of inner nodes and branching, as described by Wald *et al.* [57].

Listing 4.11 presents the implementation of multi–packet traversal of BVHs as part of the ray tracing library. Throughout the traversal of a specific path the index of the first active packet is updated, which requires an additional field to be part of each entry on the traversal stack. When an inner node is encountered it is then determined which of its sub–trees need to be traversed by performing up to three operations per child node, as given in Listing 4.12:

- Initially the first active packet is intersected with the AABB of the node. If it is determined that a ray in the packet hits the AABB the function returns signalling that the node has to be visited. This operation is called the "quick hit test" by Wald *et al.*.
- If no intersection could be determined and there are still potentially active packets left the so–called "quick out test" is performed by determining whether the frustum, which was computed for the given set of packets, allows for intersections to be found or not.
- As a last resort the remaining packets are tested individually until the first hit is found or all rays are certain to miss the node.

```
int node::IntersectWith(packet packets[], frustum f,
  int first, int count)
{
  if(IntersectWith(packets[first]))
    return first; // active packet hits the node

  if(++first < count)
  {
    if(f.MissesBox(lower, upper))
      return count; // frustum misses the node

    // check the remaining rays and return on the first hit
    do
    {
      if(IntersectWith(packets[first]))
        return first; // found a hit
    } while(++first < count);
  }
  return count; // no hit could be found
}
```

Listing 4.12: *Testing in three stages whether a node has to be visited or not [57].*

When a leaf is visited the same check for further subdivision as with single ray traversal is carried out. If this results in an execution of the leaf operation (e.g., intersection of the referenced triangles with the rays), the operation functor typically intersects all of the still active packets with the current node in order to filter packets missing the node.

Splitting a Node

Lazy construction in a multi–threaded environment has to be implemented carefully to avoid race conditions that may result in the creation of invalid bounding volume hierarchies. In

order to ensure thread safety an attempt to lock the leaf, which is to be subdivided, is performed first. If this fails due to the fact that the node is already locked by another thread, the process is aborted forcing ray traversal to try again later. Otherwise it is ascertained that the node in question is still a leaf because it may have been modified during the time the lock was acquired. If this check returns that the node has already been transformed into an inner node traversal may be resumed immediately after unlocking.

The first step of node subdivision encompasses the search for a splitting plane. For this purpose two strategies are followed by the implementation:

- For nodes containing more than n primitives (e.g., 16 primitives) SAH sampling is used to find a good splitting plane quickly [19] by computing the costs for a fixed number of 16 evenly distributed planes along each axis. Using interpolation the per–axis splitting plane with the lowest costs is determined and the best of the resulting three axes is selected.
- Otherwise plane sweeping in $O(n \log^2 n)$ is employed with sorting of the remaining primitives and evaluation of all reasonable splitting planes based on the centroids of the primitives [57]. For small triangle counts this has the benefit of finding the optimal plane in run–times comparable to SAH sampling.

In case the costs of the selected splitting plane exceed the costs of keeping the node as a leaf the node is marked as final and ray traversal is resumed after unlocking the node. However, if it is beneficial to split the leaf node the construction algorithm continues by allocating a consecutively stored pair of nodes, which is used for the new child nodes. This can be performed in a lockless manner by employing an atomic operation to increment the BVH's free memory pointer by the size of two nodes (i.e., by using the compare–and–swap instruction).

Subsequently the referenced primitives are partitioned into two disjoint sets by reordering the indices visible in the leaf's window of the global index list. In addition to the corresponding offsets in the index list the computed bounding boxes for both sets are used to initialize the new child nodes. Each of these two leaves is then marked final if it contains less than two primitives or has reached the maximal depth of the BVH.

The final steps of the construction algorithm involve updating the former leaf node to turn it into an inner node. To enable support for selective rebuilding the costs of the new configuration are stored in the field `splitCosts` of the node. Then the spatial relationship of the children as described in the following are evaluated to compute the mask, which serves as the base for ordered traversal of the BVH. Finally the type of the node is changed to an inner node, which is important to be performed last because it will affect traversal in other threads immediately.

After unlocking the node ray traversal of the BVH can be continued treating the former leaf as an inner node.

Precomputed Traversal Order of Siblings

Traversing a bounding volume hierarchy in a rough near–to–far order allows for skipping of nodes if they're farther away than a previously found intersection of a ray with geometry. A typical approach to implementing ordered traversal uses the distances of a ray to the bounding boxes of an inner node's children to determine the closer node. Although this solution yields the most accurate traversal order for a single ray, it has certain drawbacks when applied to packet–based traversal algorithms.

As depicted in Figure 4.9, the near node may differ for rays in a packet especially when they are incoherent. Basing the traversal order on an arbitrary ray in the set of rays intersecting both nodes is possible albeit associated with a reduction of accuracy, particularly when a large number of rays is active. To increase the accuracy the near node could be selected based on the accumulated data of all rays. However, this approach conflicts with multi–packet ray traversal, which makes the decision to visit a node based on the first intersection of a single packet with its bounding box. Intersection distances are therefore only available for a limited number of rays.

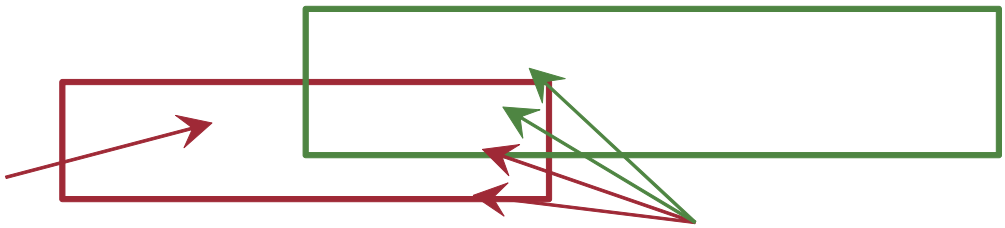


Figure 4.9: *The order of traversal may differ for rays in a packet.*

In general identifying the closer node based on a comparison of distances negatively affects the run–time efficiency of the traversal loop due to the increased register pressure and an additional comparison.

Wald *et al.* [57] propose an alternative that uses pre–computed data to find the closer node for a given ray or packet during traversal. For each inner node the axis of greatest separation (n_{axis}) of its children is stored along with the index of the child that should be visited first by rays extending in the positive direction of the axis (n_{first}). During traversal the index of the near node is then calculated by combining the sign–bit of the first ray's direction vector in n_{axis} and the node's order bit n_{first} using the XOR–operation, as given in Equation 4.2.

$$firstChild = sign(ray[0].direction[n_{axis}]) \oplus n_{first} \tag{4.2}$$

An advantage of this solution is its minimal run–time overhead both in ray traversal and BVH construction. Furthermore only 3 bits of additional storage is required for each inner node, which can typically be allocated in existing fields, thereby not increasing the size of the node data structure. Despite the fact that Wald *et al.* report good performance in practice a new approach based on a probabilistic model was developed in the context of this thesis, which increases the accuracy of the traversal order without introducing a higher run–time overhead in ray traversal.

Given a set of rays extending uniformly from a point p in space and intersecting both children of an inner node n, the probability of rays hitting a specific child first can be derived from the surface areas of the children's bounding volumes. As depicted in Figure 4.10 a "corridor" exists for each configuration of siblings that encompasses all rays intersecting both nodes. It can be observed that the proportion of rays intersecting a node m first equals the ratio of the unoccluded surface area of m in the corridor to that of the total surface area in the corridor, both visible from p. By visiting the node with the larger visible surface area first the correct order of traversal is more likely to be carried out for rays originating in p.

This approach can be generalized for rays originating at arbitrary points by pre–computing the index of the near child for each ray direction octant. The results can then be packed into an 8 bit wide mask in inner nodes and accessed based on the signs of a ray during traversal.

```
int EvaluateNearMask(node *l, node *r)
{
  int nearMask = 0;
  for(int signMask = 0; signMask < 8; ++signMask)
  {
    // compute bounding box describing the volume of interest
    int front = 0; float3 intL, intU;
    for(int axis = 0; axis < 3; ++axis)
    {
      if(signMask & (1 << axis))
      {
        intL[axis] = max(l->lower[axis], r->lower[axis]);
        intU[axis] = max(l->upper[axis], r->upper[axis]);
        front |= (l->upper[axis] >= r->upper[axis]) ? 0 : (1 << axis);
      }
      else
      {
        intL[axis] = min(l->lower[axis], r->lower[axis]);
        intU[axis] = min(l->upper[axis], r->upper[axis]);
        front |= (l->lower[axis] <= r->lower[axis]) ? 0 : (1 << axis);
      }
    }

    // intersect the bounding boxes with the volume of interest
    float3 leftL = max(l->lower, intL), leftU = min(l->upper, intU);
    float3 rightL = max(r->lower, intL), rightU = min(r->upper, intU);
    bool lvalid = (leftL.x <= leftU.x && leftL.y <= leftU.y &&
      leftL.z <= leftU.z);
    bool rvalid = (rightL.x <= rightU.x && rightL.y <= rightU.y &&
      rightL.z <= rightU.z);

    // calculate the surface areas of the visible sides
    float3 dl = leftU - leftL, dr = rightU - rightL;
    float leftArea = lvalid ? 0 : (dl.x * (dl.y + dl.z) + dl.y * dl.z);
    float rightArea = rvalid ? 0 : (dr.x * (dr.y + dr.z) + dr.y * dr.z);
    if(lvalid && rvalid)
    {
      // subtract the occluded surface areas
      float3 di = min(leftU, rightU) - max(leftL, rightL);
      ((front & 0x1) ? leftArea : rightArea) -= max(0.0f, di.y * di.z);
      ((front & 0x2) ? leftArea : rightArea) -= max(0.0f, di.x * di.z);
      ((front & 0x4) ? leftArea : rightArea) -= max(0.0f, di.x * di.y);
    }

    // pick the node with the larger visible area as the near node
    nearMask |= (leftArea >= rightArea) ? 0 : (1 << signMask);
  }
  return nearMask;
}
```

Listing 4.13: *Evaluating the index of the near node for each octant of the direction cube.*

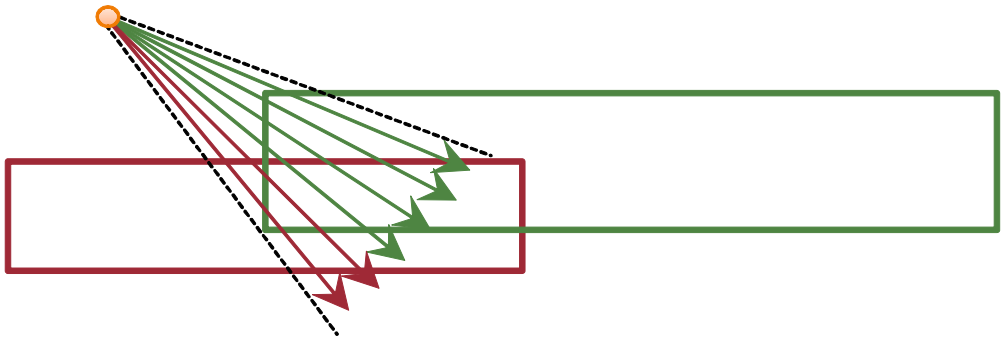


Figure 4.10: *A "corridor" exists for each configuration of siblings that encompasses all rays intersecting both children of a given inner node.*

For a given octant the volume where rays may intersect both siblings can be derived from the nodes' bounding boxes, as shown in Figure 4.11a. Along each axis the interval of interest spans from the nearest of the lower extremal values to the nearest of the upper extremal values, which can be calculated using the *min*(*max*) function for positive(negative) directions. Clipping the bounding boxes to this volume allows for calculating the surface areas of the node's visible sides in the corridor. Occlusion can be taken into account be subtracting the area of the occluding surface from the hidden surface on a per–axis basis, yielding the final algorithm in Listing 4.13.

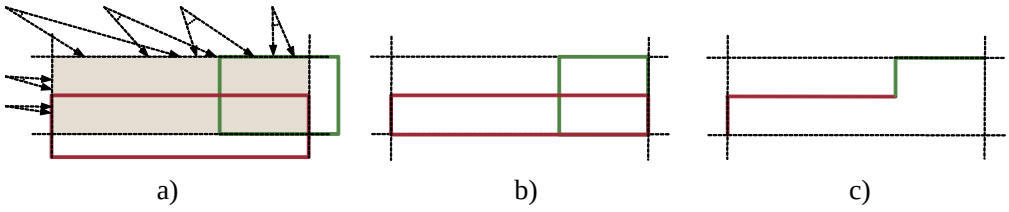


Figure 4.11: *a) Rays in a specific direction octant can intersect both nodes only in the volume of interest. b) The surface areas in the corridor can be calculated based on clipped bounding boxes. c) Taking occlusion into account to get the final values.*

An analysis of the effects of the approach on run–time performance is given in chapter 5.

Refitting of BVHs

Refitting is an operation that traverses the BVH in a bottom–up manner to update the bounding boxes of the nodes to reflect changes to geometry. The algorithm is implemented in 4 stages in the ray tracing library developed in the context of this thesis:

1. The bounding boxes of the specified triangles are updated and the set of leaves containing these triangles is assembled based on a per–triangle pointer to its hosting leaf.
2. Bounding boxes are recalculated for each leaf node in the previously assembled set. If a difference regarding the AABB is detected the leaf's parent node is inserted into a data structure that maps from the depth in the tree to a set of nodes.
3. The data structure is processed beginning with the set at the deepest level. For each node in the set its bounding box is recalculated based on the AABBs of its children. Additionally the traversal order of its children is reevaluated. If the new bounding box

differs from the old the node's parent is inserted into the data structure at $depth - 1$, thereby ensuring that bottom–up traversal continues only if necessary. This processing schemes is an implementation of parallel bottom–up traversal of multiple sub–trees in level order and ends when only a single node is waiting to be updated at a depth closer to the root than all of the updated leaves.

4. Finally the rest of the path from the single active node to the root of the BVH is traversed, updating nodes on the way and stopping early when the new AABB for a node equals the old AABB. Although the functionality of this stage is already present in the previous stage, processing a single sub–tree only allows for an efficient implementation of bottom–up traversal, as given in Listing 4.14.

Figure 4.12 illustrates the behavior of the algorithm for a sample BVH.

```
node *n = ...
while(true)
{
  node *children = n->GetChildren();
  n->nearmask = EvaluateNearMask(&children[0], &children[1]);
  float3 lower = min(children[0].lower, children[1].lower);
  float3 upper = max(children[0].upper, children[1].upper);
  if(n->lower != lower || n->upper != upper)
  {
    n->lower = lower; n->upper = upper;
    if(n->IsRootNode())
      break;
  }
  else
    break;
  n = n->GetParent();
}
```

Listing 4.14: *Bottom–up traversal used during refitting when only a single sub–tree is processed.*

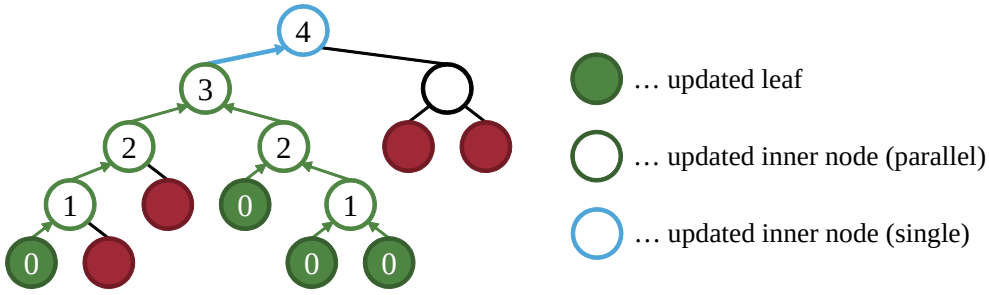


Figure 4.12: *BVH refitting is performed in a bottom–up manner. Numbers in the figure correspond to the time of processing of a node.*

Selective Rebuilding

Selective rebuilding is an extension of the refitting algorithm with conditional rebuilding of sub–trees. For each inner node the SAH cost function is reevaluated in a greedy manner with a subsequent comparison of the resulting value and the node's original costs, as expressed in Listing 4.15. If the difference exceeds a specified threshold the node is inserted into a list

to facilitate deletion of its sub–tree and transformation into a leaf, thereby making the node again subject to splitting during ray traversal.

```
bool RebuildFavorable(node *n, float threshold)
{
  node *children = n->GetChildren();
  float sa = EvaluateSurfaceArea(n);
  float lsa = EvaluateSurfaceArea(&children[0]);
  float rsa = EvaluateSurfaceArea(&children[1]);
  int leftCount = children[0].lastObj - children[0].firstObj + 1;
  int rightCount = children[1].lastObj - children[1].firstObj + 1;
  float splitCosts = (lsa * leftCount + rsa * rightCount) / sa;
  return (splitCosts >= n->splitCosts * threshold);
}
```

Listing 4.15: *Selective rebuilding reevaluates the SAH cost function for each inner node and uses the result to decided whether the node should be rebuilt or not.*

After the list `rebuildInners` of nodes, which are subject to rebuilding, has been collected their sub–trees are deleted iteratively using the following algorithm, which also compacts the memory reserved for nodes. This is necessary to enable allocation of nodes merely by incrementing a pointer to the next free node, which can be performed with an atomic instruction and therefore does not require locking.

1. The sub–tree of the current node is traversed in top–down order to collect all nodes that need to be freed in a list `pairs`. Due to the fact that nodes are allocated in pairs it suffices to store the offset of the first child of each inner node. In case a node is found that is already marked for deletion in the list `rebuildInners`, the node is removed from `rebuildInners` as it will be deleted in the current iteration.

2. The current node is turned into a leaf with the contained triangles being updated to reference it as their hosting node.

3. Deletion of nodes and compaction of memory is carried out using the functionality given in Listing 4.16 and illustrated in Figure 4.13: For each pair of nodes that is marked for deletion in the list `pairs` the BVH's pointer to the next free node is decremented by 2. If this results in the current pair being moved to the free memory area no further work is due. However, if nodes that are still in use end up in free memory they are moved to the area of used memory by overwriting the current pair. By repeatedly deleting the node in the list `pairs` with the highest address (e.g., by sorting the list first) the number of copy operations can be reduced since the deletion from the tail of used memory is more likely to happen.

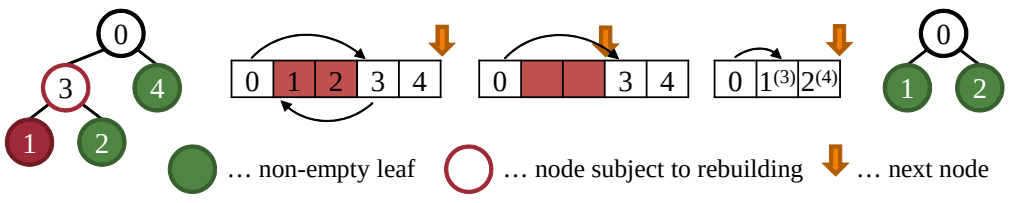


Figure 4.13: *Deletion of the sub–tree of node 3 with subsequent compaction of the node array.*

```
...

for(vector<int>::iterator it = pairs.begin(); it != pairs.end(); ++it)
{
  // decrement the pointer to free memory by the size of two nodes
  _nextNode -= 2;
  if(*it >= _nextNode)
    continue;

  // the pair of nodes that should be freed is not at the tail
  // overwrite them with the nodes that are at the tail
  node *dest = _nodes + *it, *src = _nodes + _nextNode;

  // update the parent of src to with the new position of its children
  node *parent = src->GetParent();
  parent->offsetToChildren = dest - parent;

  // copy the nodes individually
  for(int i = 0; i < 2; ++i)
  {
    // if a nodes is moved that is subject to later rebuiling
    // its address needs to be updated in the list
    list<int>::iterator idx = find(rebuildInners.begin(),
      rebuildInners.end(), &src[i] - _nodes);
    if(idx != rebuildInners.end())
      *idx = &dest[i] - _nodes;

    dest[i] = src[i];
    dest[i].offsetToParent = parent - &dest[i];

    if(src[i].IsLeaf())
    {
      // update offsets of triangles to their hosting node
      int leaf = &dest[i] - _nodes;
      for(int j = src[i].firstObj; j <= src[i].lastObj; ++j)
        _objGetter.Get(indices[j]).leaf = leaf;
    }
    else
    {
      // update hierarchy connectivity information
      node *children = src[i].GetChildren();
      children[0].offsetToParent = &dest[i] - &children[0];
      children[1].offsetToParent = &dest[i] - &children[1];
      dest[i].offsetToChildren = children - &dest[i];
    }
  }
}
```

Listing 4.16: *Deletion of nodes and compaction of memory is performed as the last operation of selective rebuilding.*

4.4.2 Integration of the BVH

In the following the integration of the BVH with the classes of the two-level hierarchy is described with regard to construction, updating and traversal of the data structure.

As previously stated the construction and updating functionality of the BVH operates on bounding boxes only and is not tied to the specific contents of the classes `DynamicScene` and `DynamicGeometry`. It is therefore necessary for these two classes to maintain an internal array of AABBs accessible to the BVH and keep it in sync to the actual contents (i.e. triangles or instances), which is an operation that is best carried out prior to invocations of the BVH's update methods, as demonstrated for refitting of raw geometry in Listing 4.17. As can be seen it is also the responsibility of `DynamicGeometry` to forward updates to its parents, which makes it necessary for geometry–objects to track the scenes they have been instantiated in.

```
void Refit(int firstTriangle, int lastTriangle)
{
  // update bounding boxes of the triangles in the specified range
  UpdateTriangles(firstTriangle, lastTriangle);

  // list all leaves containing parts of the specified triangles
  std::set<int> leaves;
  for(int i = firstTriangle; i <= lastTriangle; ++i)
    leaves.insert(_triangles[i].leaf);

  // reevaluate the bounding boxes for the leaves and
  // propagate the updated bounding boxes up to the root node
  if(_hierarchy.Refit(leaves))
  {
    // continue propagation in the scene for all leaves containing
    // instances of this geometry
    for(typename std::map<DynamicScene *, std::vector<int>>::
      iterator it = _parents.begin(); it != _parents.end(); ++it)
    {
      it->first->Refit(this);
    }
  }
}
```

Listing 4.17: *Prior to refitting the bounding boxes of the triangles are recalculated. In case the geometry's root node was modified the refitting algorithm is continued on the upper level.*

Although traversal of the BVH is identical in both levels of the hierarchy, the operations carried out in leaves differ depending on the contents. It is therefore necessary to customize this aspect of the BVH, which is accomplished through the use of a functor object. Ray–triangle intersections are performed in the class `DynamicGeometry`, which update rays with information about the hit triangles encompassing their zero–based index in the container and the handle of the container instance. This is in contrast to the class `DynamicScene` where rays are transformed and traversal of the instances contained in a leaf is carried out. The implementation of the functor used by objects of the class `DynamicScene` is given in Listing 4.18.

Figure 4.14 illustrates the control flow of ray traversal of the two–level dynamic acceleration structure.

```
class TraversalLeafOp
{
  const std::map<int, Instance> &_instanceMap;
  const std::vector<int> &_indices;
  const unsigned int _signMask;

public:
  TraversalLeafOp(const std::map<int, Instance> &instanceMap,
    const std::vector<int> &indices, unsigned int signMask) :
    _instanceMap(instanceMap), _indices(indices),
    _signMask(signMask)
  { }

  void operator()(ray &r, const float3 &invDir, int firstObj,
    int lastObj, int flags) const
  {
#ifdef RECORDRAYSTATS
    ++r.dynamicStats.visitedInnerNodes;
#endif
    for(int i = firstObj; i <= lastObj; ++i)
    {
      const Instance &inst =
        _instanceMap.find(_indices[i])->second;
      if(inst.transformed)
      {
        // transform the ray and traverse the hierarchy
        float3 org = r.origin, dir = r.direction;
        r.origin = inst.invtransform * float4(org, 1);
        r.direction = inst.invtransform * float4(dir, 0);
        inst.geometry->Trace(r, 1.0f / r.direction, flags,
          _signMask, inst.handle);
        r.origin = org; r.direction = dir; // restore
      }
      else
      {
        // traverse the hierarchy without ray transformation
        inst.geometry->Trace(r, invDir, flags,
          _signMask, inst.handle);
      }
    }
  }
}
```

Listing 4.18: *Functor used by* `DynamicScene` *to traverse lower level hierarchies in its leaves.*

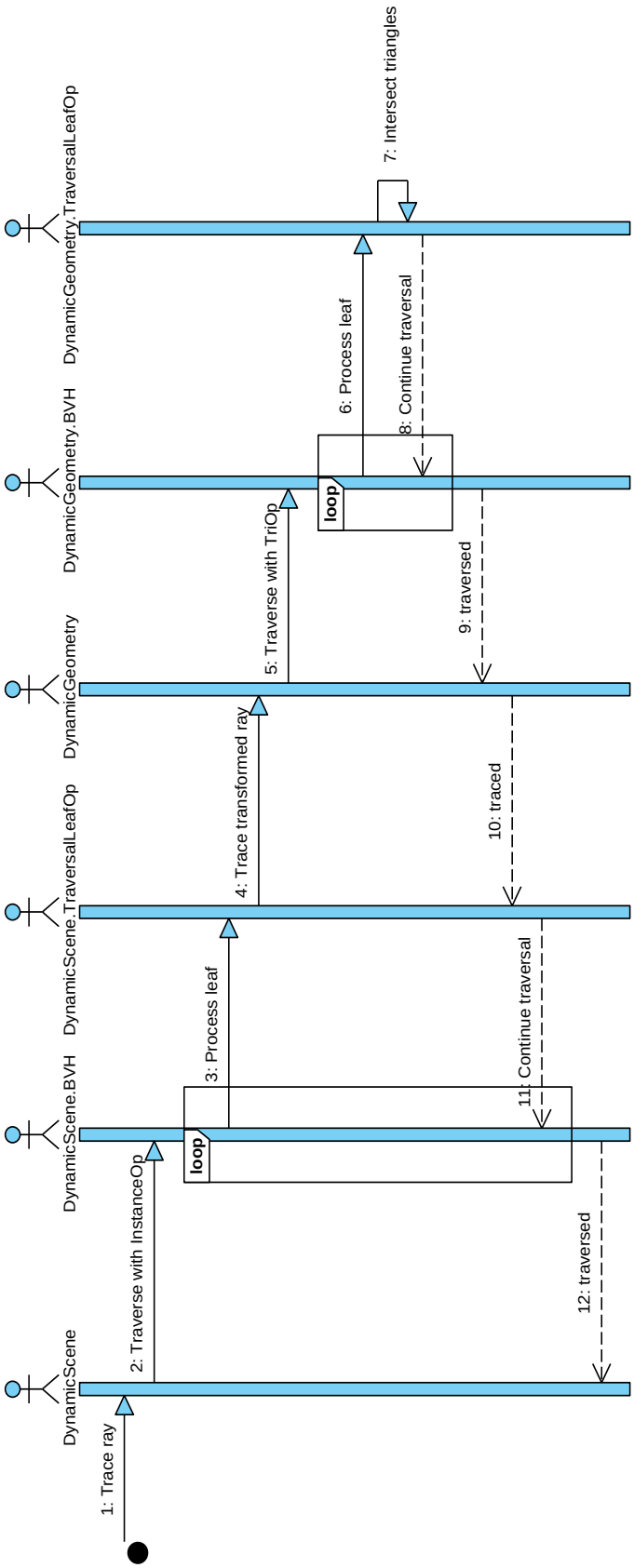


Figure 4.14: *Control flow of ray–traversal of the two–level acceleration structure.*

Chapter 5

Applications & Results

In this chapter an evaluation of the ray tracing library developed as part of this thesis is given based on separate benchmarks for scenes with static geometry only and for scenes also comprised of animated primitives. Additionally the applications used for the benchmarks are presented. Finally the use of real–time ray tracing in the context of games is demonstrated.

Benchmark data presented in this chapter was collected running the 64–bit version of Ubuntu Linux 7.10 on a system with a dual–core AMD Athlon X2 3800+ processor and 3 GB RAM. Rendering applications were configured to process tiles comprised of 8×8 pixels, which were dynamically distributed to two threads. GCC 4.2 was used for compiling the library and the applications.

5.1 Tracing Static Scenes

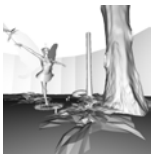
An evaluation of run–time performance in rendering static scenes was performed with a benchmarking application developed specifically for this purpose. The application provides an intuitive user interface for loading scenes and viewing statistical data both in graphical and numerical form, as shown in Figure 5.2.

Several different rendering modes are available, such as simple diffuse lighting, color coding of the visible triangle or mapping of traversal steps to color, which is particularly useful for analyzing the complexity of the acceleration structure. Furthermore the average number of traversal steps and ray–triangle intersection tests is also displayed in addition to a set of values derived from the per–frame rendering time (current frame rate, average number of frames per second, etc.).

The application supports loading of a variety of scenes at run–time, which are mainly taken from a collection created by Wächter [59]. For each scene a pre–defined camera position and orientation is available, which were specified by Wächter in order to allow for comparability of published results. The following scenes are included in the collection:

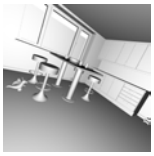
- CLOWN modeled by C. Wächter (44,154 triangles)
- FAIRY FOREST from the Utah 3D Animation Repository [46] (174,117 triangles)
- HAPPY BUDDAH from the Stanford 3D Scanning Repository [49] (1,087,716 triangles)

Clown (44,154 triangles)	kd–tree	BVH
a) Average frames per second	20.3	23.6
b) Average number of visited nodes per ray	36.7	31.0
c) Average number of intersected triangles per ray	1.9	6.2

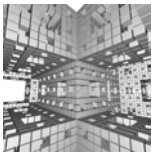


Fairy Forest (174,117 triangles)	kd–tree	BVH
a) Average frames per second	12.7	13.5
b) Average number of visited nodes per ray	63.3	60.2
c) Average number of intersected triangles per ray	4.7	14.8

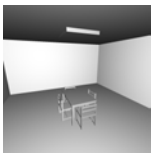
Happy Buddah (1,087,716 triangles)	kd–tree	BVH
a) Average frames per second	15.6	10.7
b) Average number of visited nodes per ray	36.1	64.0
c) Average number of intersected triangles per ray	3.2	18.3



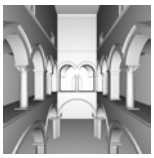
BART Kitchen (110,561 triangles)	kd–tree	BVH
a) Average frames per second	13.7	9.5
b) Average number of visited nodes per ray	53.6	80.9
c) Average number of intersected triangles per ray	5.9	27.5



Menger Sponge (1,920,000 triangles)	kd–tree	BVH
a) Average frames per second	15.6	15.3
b) Average number of visited nodes per ray	53.1	56.3
c) Average number of intersected triangles per ray	3.2	9.5



scene6 (804 triangles)	kd–tree	BVH
a) Average frames per second	27.8	35.9
b) Average number of visited nodes per ray	20.5	10.7
c) Average number of intersected triangles per ray	1.4	2.8



Sponza Atrium (67,462 triangles)	kd–tree	BVH
a) Average frames per second	17.6	12.8
b) Average number of visited nodes per ray	42.6	68.4
c) Average number of intersected triangles per ray	2.8	13.6

Figure 5.1: *Benchmark data for a collection of static scenes rendered with simple diffuse shading at a resolution of* 512^2 *pixels.*

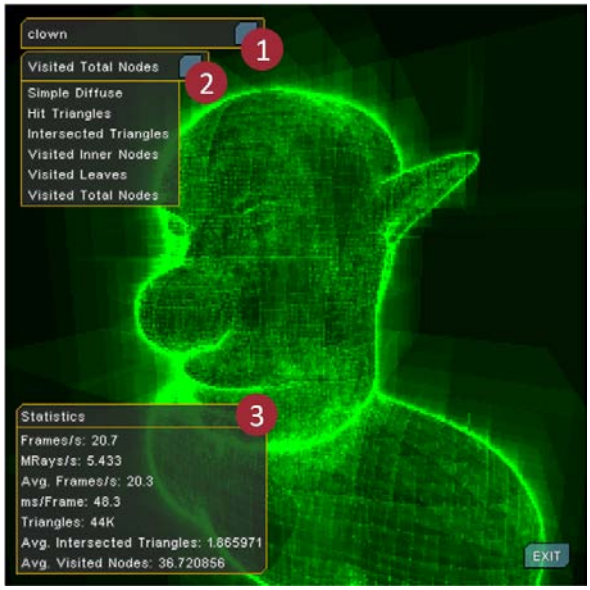


Figure 5.2: *1) A variety of scenes can be loaded through a drop–down menu. 2) 6 rendering modes are supported, with the current selection yielding a display of the number of visited nodes. 3) Statistics about scene complexity and run–time performance are supplied.*

- KITCHEN from the Benchmark for Animated Ray Tracing [32] (110,561 triangles), with a transformation applied by C. Wächter to challenge building of efficient acceleration structures with axis–aligned partitioning schemes
- MENGER SPONGE supplied by C. Wächter, fractal geometry (1,920,000 triangles)
- SCENE6 from P. Shirley's global illumination test scenes (804 triangles)

As an additional scene a model of the Sponza Atrium [8] (67,462 triangles) is used in the benchmarks due to its popularity in the ray tracing research community. The camera was configured to point at the scene's origin in the positive direction along the x–axis with the y–axis acting as the "up"–vector.

5.1.1 Rendering Performance

Benchmark data obtained by rendering the scenes with simple diffuse shading are given in Figure 5.1. No secondary rays were spawned in the process and data was collected for both kd–trees and BVHs serving as the acceleration structure for the geometry. The data shows that neither the kd–tree nor the BVH is the obvious choice for ray tracing of static scenes when only primary rays are processed due to the fact that the Clown, the Fairy Forest, and scene6 seem to be better suited for rendering with BVHs, while higher frame rates for the other scenes can be achieved with a kd–tree.

Kd–trees have the benefit of enabling flexible partitioning of primitives with planes that can adapt well to geometry especially in architectural scenes. Furthermore the ray traversal algorithm traverses the data structure in near–to–far order, which allows for terminating traversal after the first valid intersection has been found. Both aspects contribute to the typically high culling efficiency of kd–trees, yielding a low number of ray–triangle intersection tests per ray.

Traversal of BVHs on the other hand scales very well to large coherent packets, which is the case with rays originating from the camera. This can give a performance advantage in some of the scenes despite lower triangle culling efficiency. Furthermore BVHs enable fast skipping of empty space because unlike in kd–trees empty space is not represented explicitly in empty nodes and therefore does not have to be traversed.

5.1.2 Cost–scaling Termination Criterion

In chapter 3 a new termination criterion for SAH–based construction of kd–trees was introduced, which is based on the idea of limiting tree depth by splitting deep nodes only if sufficiently large gains can be expected from the operation. For this purpose the traditional SAH–based termination criterion is modified to scale down the costs of a leaf relatively to the costs of an inner node based on the depth in the kd–tree. In the context of this thesis an exponential function was used, as shown in Figure 5.3, which was parameterized by specifying the scale factor at the maximal kd–tree depth.

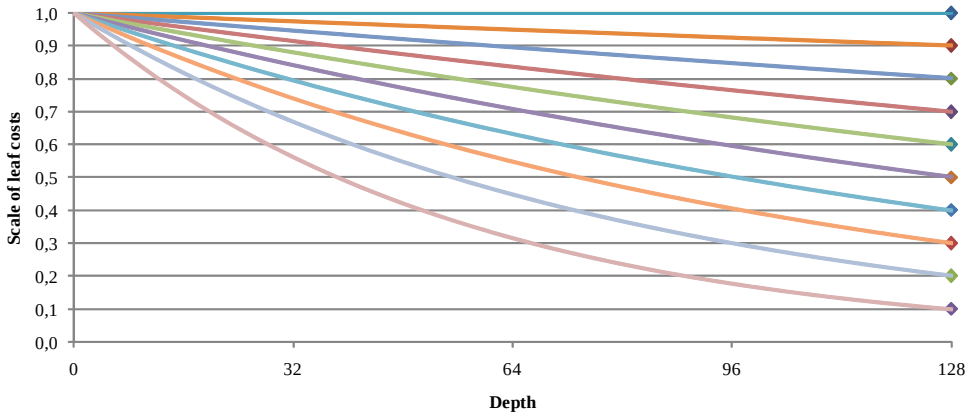


Figure 5.3: *Exponential scaling of leaf costs, parameterized by specifying the scale factor at the maximal kd–tree depth (here: 128, which was set through a constant in the code).*

Figure 5.4 presents the results of this new termination criterion with regard to rendering time, building time and the size of the built kd–trees for each of the previously discussed scenes. As can be seen rendering time is affected only marginally when leaf costs are scaled down to no less than 10% of their original value at the maximal kd–tree depth. For some scenes like the Menger Sponge or the BART Kitchen this results in a minor drop of rendering performance of about 5.3%, whereas a speed–up of 6.3% can be measured for the Happy Buddah scene. However, at the same time the building time is shortened to 38.0% on average compared to the building time with the non–scaling termination criterion. The size of the generated kd–trees experiences a similar drop to about 46.0% on average for the given set of scenes.

Therefore the new termination criterion has very positive effects on ray tracing of static scenes with kd–trees, especially when considering the reduced storage requirements of kd–trees, which is an important aspect of the distribution of applications with pre–compiled scene data. By tweaking the scaling factor on a per–scene basis the size of the kd–tree can be reduced without negatively affecting ray traversal performance.

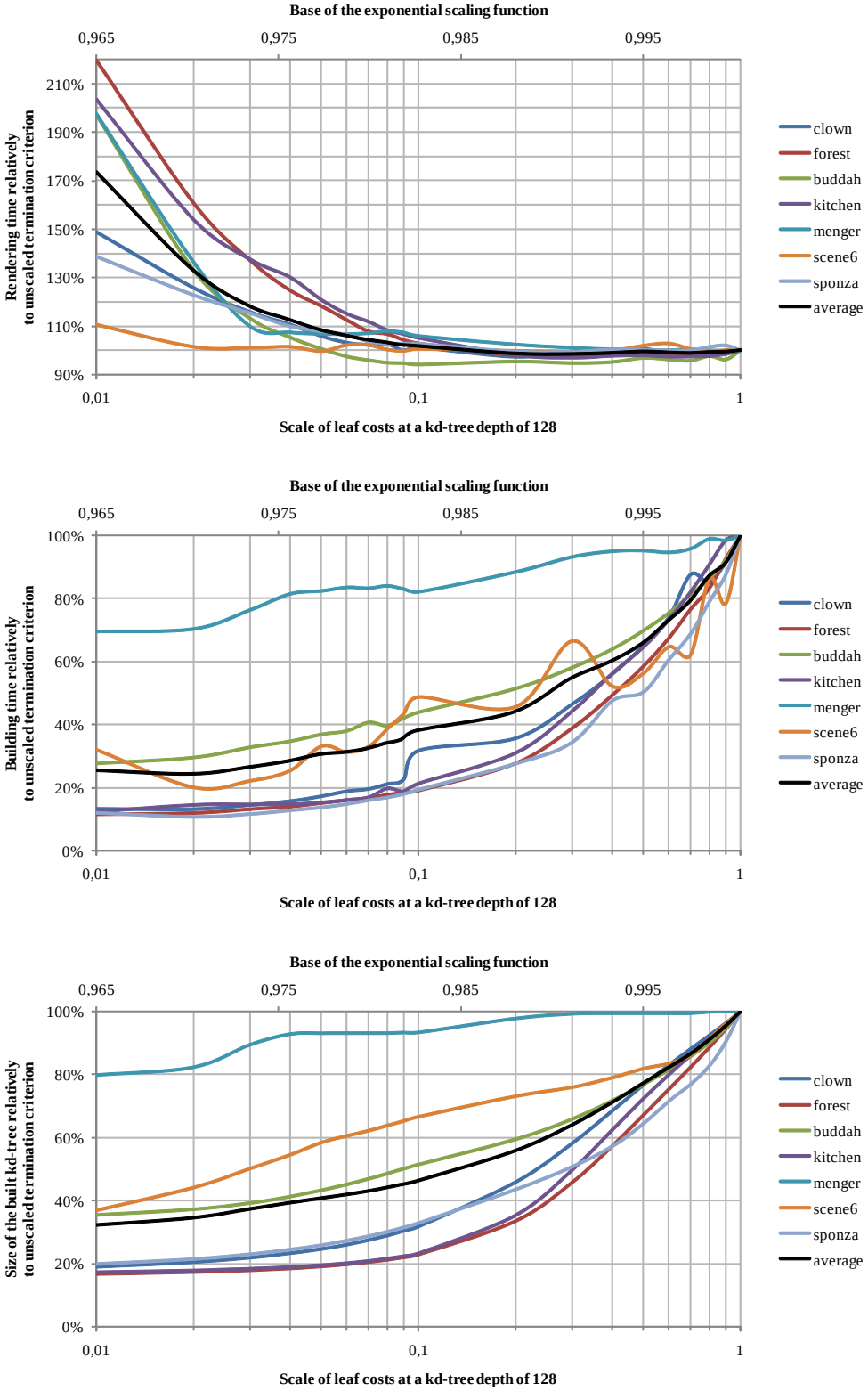


Figure 5.4: *The graphs present data regarding rendering time, building time and the size of the built kd–trees for each scene relatively to kd–tree construction with the traditional SAH–based termination criterion.*

5.2 Benchmark for Animated Ray Tracing

The Benchmark for Animated Ray Tracing (BART) was created by Lext *et al.* [32] and encompasses a set of scenes with scripted animations of geometry and a pre–determined camera path. The intention behind creating the benchmark was to provide a common data set for researchers, which could then evaluate the run–time performance of their ray tracing applications and publish the results in a comparable manner, as done in the appendix of this thesis.

Lext *et al.* [32] provide a code framework for loading their scenes, which served as the base for a second benchmarking application developed in the context of this thesis for animated scenes. Figure 5.5 presents the user interface of the program, which provides similar functionality to the previously presented application for benchmarks of ray tracing static scenes.

Additional features include support for a larger variety of rendering modes tailored to the available scene data (e.g. rendering with diffuse illumination only or rendering with all effects enabled including shadows, reflections, and refractions). Furthermore it is possible to change the updating scheme of the BVH at run–time and record a video clip of the rendered animation, as given in Figures 5.12, 5.13 and 5.14.

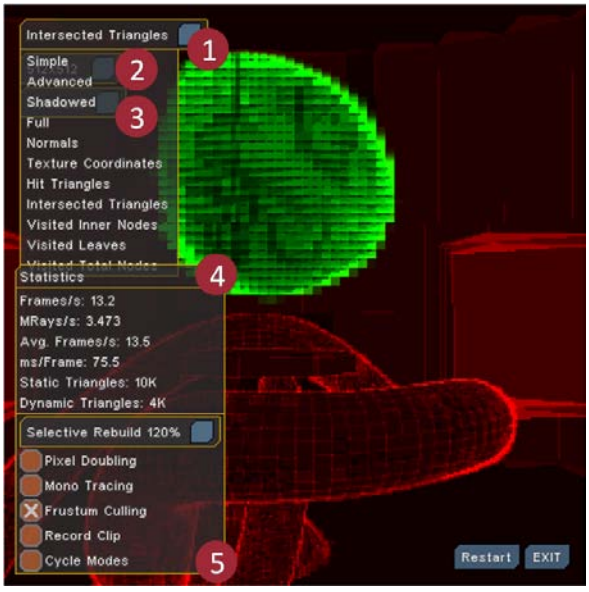


Figure 5.5: *1) 11 rendering modes are supported, with the current selection yielding a display of the number of intersected triangles of static (red) and dynamic (green) geometry. 2) The size of the application window can be changed to a pre–defined set of window sizes. 3) The BART scenes can be selected from a drop–down menu. 4) Statistics about run–time performance and the complexity of the scene are displayed. 5) The behavior of the application can be modified at run–time.*

Benchmark data was collected by measuring the per–frame rendering time for a large number of different configurations, including different rendering modes, updating schemes and the separation of static from dynamic geometry in two acceleration structures. From this data several key figures were derived, which will be discussed in the following.

Figure 5.6 provides data for an analysis of the effects of handling static geometry in a kd–tree separately from dynamic geometry in the two–level bounding volume hierarchy, which stands in contrast to handling all primitives solely in the BVH.

Kitchen	BVH only	BVH + kd–tree	Speed–up
a) Diffuse	118.09s	140.27s	-15.8%
b) Diffuse with shadows	782.81s	319.22s	+145.2%
c) Diffuse with reflections, refractions	945.24s	485.84s	+94.6%
d) All effects	1623.22s	668.05s	+143.0%
Museum (4096)	BVH only	BVH + kd–tree	Speed–up
a) Diffuse	30.78s	41.43s	-25.7%
b) Diffuse with shadows	250.70s	159.28s	+57.4%
c) Diffuse with reflections, refractions	672.61s	421.74s	+59.5%
d) All effects	892.16s	542.29s	+64.5%
Robots	BVH only	BVH + kd–tree	Speed–up
a) Diffuse	118.18s	132.28s	-10.7%
b) Diffuse with shadows	745.44s	210.08s	+254.8%
c) Diffuse with reflections, refractions	9299.19s	1016.11s	+815.2%
d) All effects	9763.40s	1095.44s	+791.3%

Figure 5.6: *Benchmark data for the BART scenes rendered at a resolution of* 800×600 *pixels with a maximal recursion depth of 3 for reflections and refractions.*

As can be seen the BVH–only approach performs better than the combined approach when no secondary rays are traced, which is the case with the diffuse rendering mode. However, once secondary rays are spawned the use of an additional kd–tree for static geometry allows for a reduction of rendering times. This is due to the fact that traversal of BVHs exhibits poor run–time performance when packets comprised of only a few rays are traced, which is typical for reflection and refraction rays but also for shadow rays in case only a limited number of lights are present in a scene.

By moving static geometry to a kd–tree the BVH contains less primitives and can be traversed faster. As traversal of kd–trees with small numbers of rays incurs only a minimal performance hit the overall rendering time is reduced. Especially the highly reflective Robots scene benefits from this approach with measured speed-ups of up to 800%.

Figures 5.7 through 5.9 present a detailed view of the per–frame distribution of time spent on the various supported effects, such as diffuse lighting, shadowing, reflections, and refractions. The data for the graphs was derived from the data collected for Figure 5.6 as given in Equation 5.1.

$$\begin{aligned} t_{shadows} &= t_{diffuse+shadows} - t_{diffuse} \\ t_{reflections+refractions} &= t_{diffuse+reflections+refractions} - t_{diffuse} \\ t_{total} &= t_{diffuse} + t_{shadows} + t_{reflections+refractions} \end{aligned} \tag{5.1}$$

The resulting time t_{total} is within $\pm 2\%$ of the measured $t_{all-effects}$. The quality of the approximation of the individual factors can therefore be considered good enough for display in graphical form.

5.2.1 Ordered Traversal of BVHs

In chapter 4 an implementation of ordered traversal of bounding volume hierarchies based on a probabilistic model was introduced. Figure 5.10 presents the rendering times of the

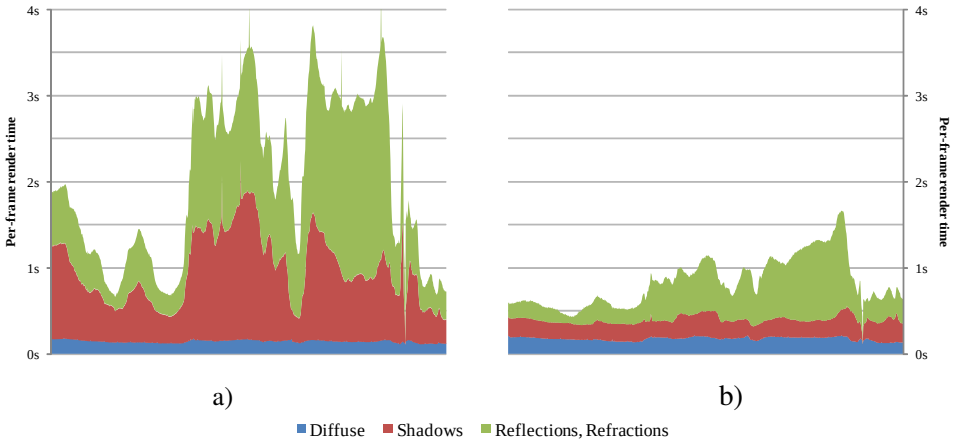


Figure 5.7: *Per–frame times spent rendering the animation sequence of the BART Kitchen with a) BVH only and b) BVH + kd–tree at a resolution of* 800×600 *pixels with a maximal recursion depth of 3 for reflections and refractions.*

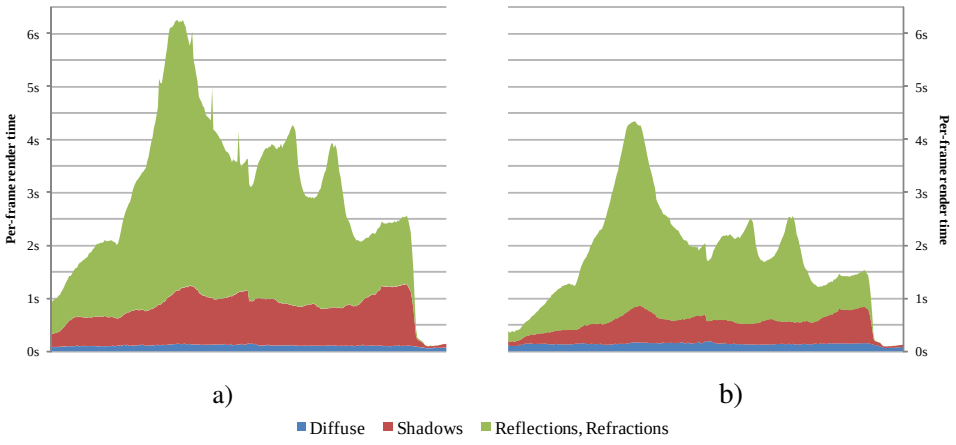


Figure 5.8: *Per–frame times spent rendering the animation sequence of the BART Museum (4096) with a) BVH only and b) BVH + kd–tree at a resolution of* 800×600 *pixels with a maximal recursion depth of 3 for reflections and refractions.*

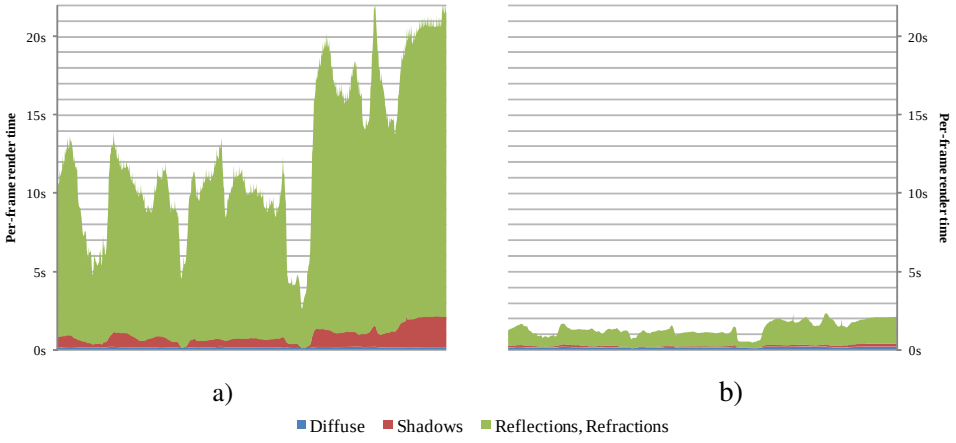


Figure 5.9: *Per–frame times spent rendering the animation sequence of the BART Robots with a) BVH only and b) BVH + kd–tree at a resolution of* 800×600 *pixels with a maximal recursion depth of 3 for reflections and refractions.*

BART scenes with simple color coding of the visible triangles for both the new approach and the approach proposed by Wald *et al.* [57]. All geometry was stored solely in a BVH in order to avoid skewed results due to the use of a kd–tree as a second acceleration structure.

Animation sequence	BVH_{Wald}	BVH_{Reiter}	Speed–up
Kitchen	65.42s	62.72s	+4.30%
Museum (64)	17.55s	17.41s	+0.81%
Museum (256)	18.06s	17.60s	+2.61%
Museum (1024)	19.33s	18.77s	+3.01%
Museum (4096)	22.39s	21.97s	+1.95%
Museum (16384)	32.79s	32.02s	+2.41%
Museum (65536)	78.27s	78.26s	+0.02%
Robots	76.74s	72.78s	+5.44%

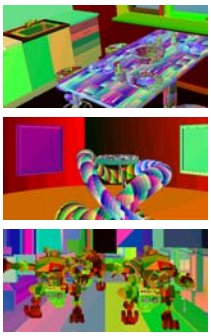


Figure 5.10: *Benchmark data for the BART scenes rendered with BVHs and different ordered traversal schemes with simple triangle coloring at a resolution of* 512^2 *pixels.*

The probabilistic model developed in the context of this thesis yields consistently better run–time performance than the approach presented by Wald *et al.* [57] with a speed–up of 4.3% for the Kitchen scene, 5.4% for the Robots scene and up to 3% for variations of the Museum scene.

5.2.2 Updating Schemes

Selective rebuilding was introduced as an updating scheme for bounding volume hierarchies in chapter 3. In addition to this technique the ray tracing library also supports full rebuilding and refitting, which is conceptually equivalent to selective rebuilding with an infinitely high rebuilding threshold. An evaluation of these updating schemes was performed by rendering the variations of the BART museum scenes with color coding of visible triangles and separation of static from dynamic geometry.

As visible in Figure 5.11, selective rebuilding with a threshold of +30% delivers the best run–time performance for the Museum scenes with more than 256 triangles dynamic triangles with speed–ups up to 450% over refitting and 250% over rebuilding. For variations with fewer dynamic triangles the gains from updating the BVH are negligible and the conceptually simple approach of fully rebuilding the BVH for each frame is preferable.

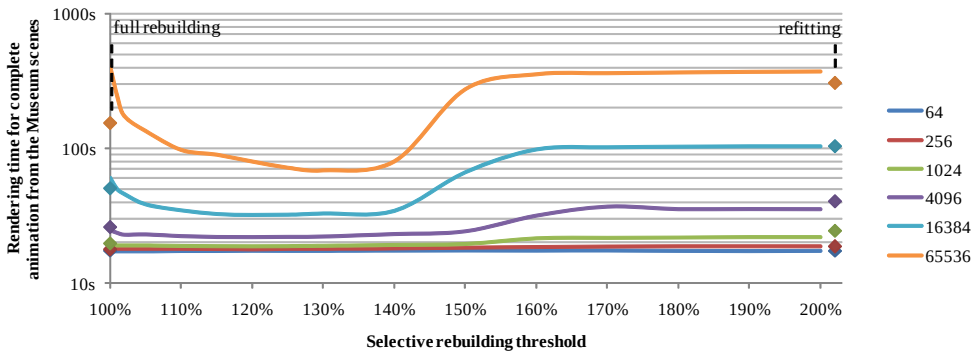


Figure 5.11: *Benchmark data for the Museum scenes with varying counts of animated triangles rendered at a resolution of* 512^2 *pixels with simple color coding and different updating schemes.*

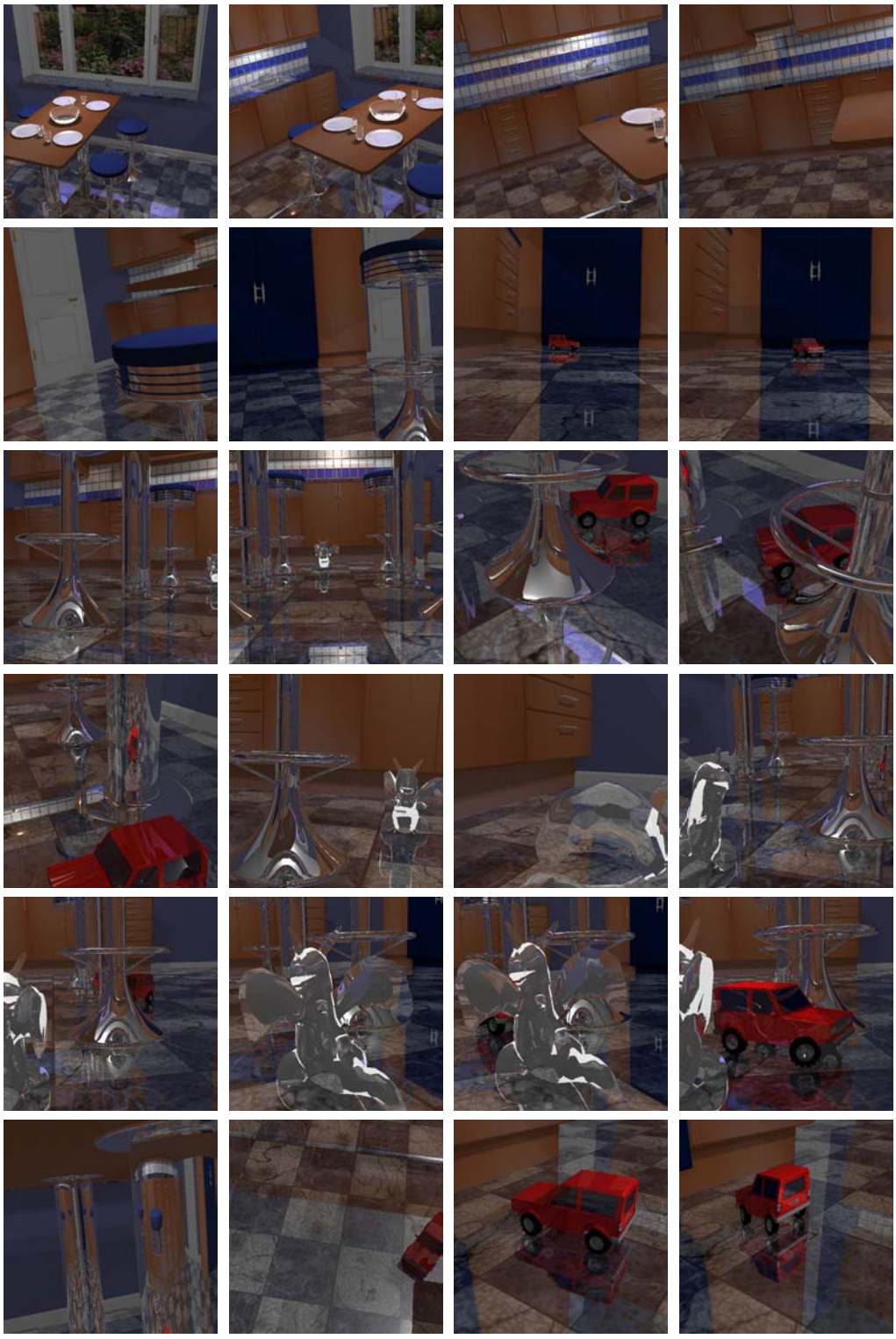


Figure 5.12: *Animation sequence comprised of 800 frames from the BART Kitchen scene rendered at a resolution of* 512^2 *pixels with all effects enabled (recursion depth 3) in 668.3s (1.20fps on average).*

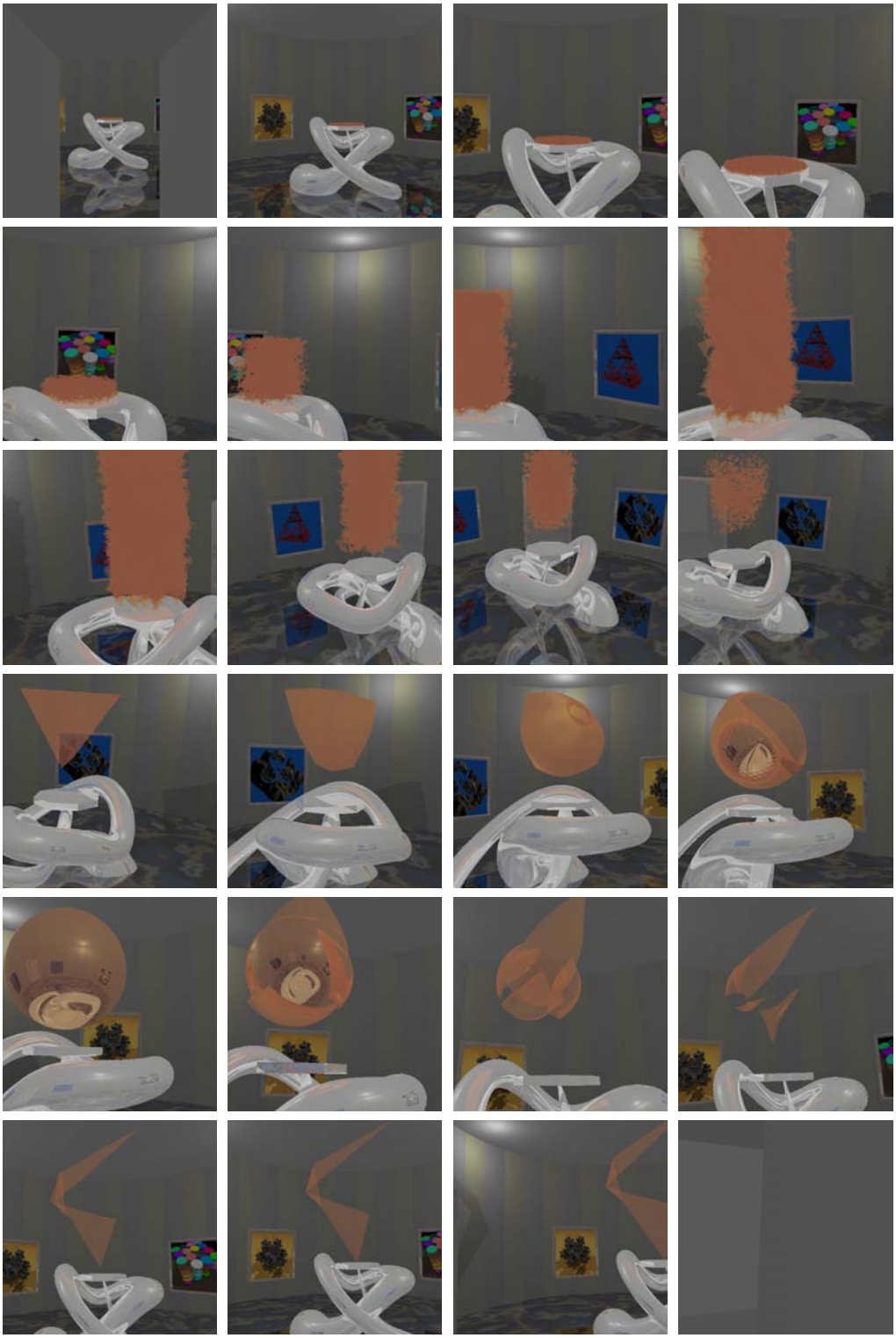


Figure 5.13: *Animation sequence comprised of 300 frames from the BART Museum scene with 4096 dynamic triangles rendered at a resolution of* 512^2 *pixels with all effects enabled (recursion depth 3) in 542.3s (0.55fps on average).*

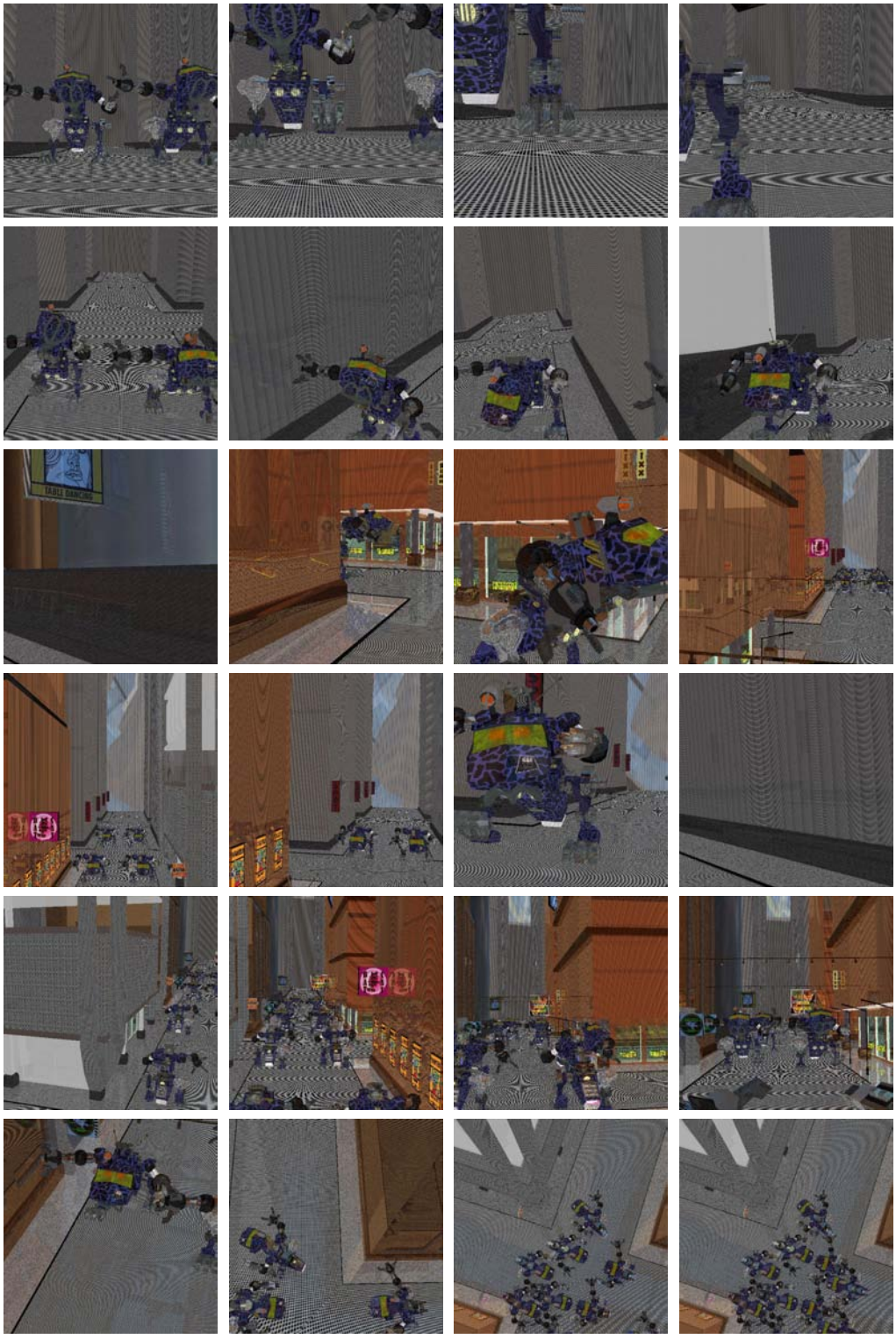


Figure 5.14: *Animation sequence comprised of 800 frames from the BART Robots scene rendered at a resolution of* 512^2 *pixels with all effects enabled (recursion depth 3) in 1095.4s (0.73fps on average). Heavy aliasing related to under–sampling of textures is visible in the images due to the fact that only bilinear filtering is performed.*

5.3 Real–time Ray Tracing in Games

As part of this thesis the use of real-time ray tracing in games was also explored. Employing the assets from Raven Software's game *Star Trek Voyager: Elite Force* [41] a map viewer was developed with the goal of rendering the original scenes augmented with new effects specific to ray tracing, such as pixel-perfect reflections.

5.3.1 Tools

Creating the map viewer involved retrieving data from the original game and preparing it for rendering in a ray tracer, which resulted in the creation of an extensive tool chain that will be discussed briefly in the following.

Shaders

Elite Force and other games based on id Software's *Quake 3 Arena* (Q3A) engine [20] support the use of a shader language for describing the look and behavior of surfaces and objects in the game, as discussed in detail in the Q3A shader manual [24]. In order to be able to achieve a similar look and behavior in the map viewer a parser for this language was created with Coco/R [38] based on an attributed grammar. Using the parser surface shading and animation code is generated and compiled to a dynamic link library with the GNU C compiler (GCC) for use in rendering and scene updating in the map viewer.

Geometry data

Geometry of *Elite Force* maps is stored in a proprietary binary data format, which was however made publicly available with the release of the source code to the mapping tools of the Q3A engine under an open source license. It was therefore possible to create a tool that would load the original map files for further processing. In particular static geometry is separated from dynamic geometry by checking each primitive whether the assigned shader specifies an animation for it or not. Additionally all geometry is tesselated to triangles and written along with information about scene entities, such as lights, weapons, and character spawn points, to an XML file for simplified loading of map data in the viewer.

Post Processing of Static Geometry

Static geometry is then modified by adding an epsilon offset to so–called "decals", which are planar surfaces typically rendered last in a rasterizer to display markers on top of regular world geometry. Adding an offset is necessary in order to avoid precision problems in a ray tracer that would manifest themselves as graphical artifacts like noise in the rendered image. For this stage the shader parser is again used to determine whether a given triangle is part of a decal or not. Finally a kd–tree is built for static geometry and stored in binary form to disk to accelerate loading of maps in the viewing application.

5.3.2 Map Viewer

The pre–processed data is then loaded and used by the map viewer in the following manner:

Figure 5.15: *Garden from Raven Software's Star Trek Voyager: Elite Force rendered with full lighting effects at a resolution of* 1275×642 *pixels in the map viewer developed within the context of this thesis.*

- The dynamic link library containing the shaders is loaded and a shader manager is instantiated that manages creation and destruction of shader objects.
- Static world geometry is loaded from the pre–built kd–tree. A dynamic container is created for animated world geometry, such as water surfaces, and initialized with data from the XML file. Additional per–triangle data, such as normal vectors or texture coordinates, is retrieved from the XML file. All animated geometry is then updated by shaders each frame.
- Entities are parsed and an acceleration structure for lights is created to enable fast range queries for determining the set of lights affecting a given surface. Furthermore object entities, such as weapons and player characters, are loaded from external `.md3` and `.mdr` files, which contain models in another proprietary format by id Software and Raven Software. Each model file is loaded only once and stored in a dynamic container. Instancing is then used to render models at multiple positions in the scene.

The map viewer exhibits a graphical user interface similar to the previously described two benchmarking applications for static and for animated geometry, as depicted in Figure 5.16. It provides access to multiple rendering modes, like textured rendering, display of normal vectors and texture coordinates, and traversal statistics. Furthermore the behavior of the application can be modified, for example by forcing mono ray tracing, by enabling lighting, shadows, and advanced effects like reflections.

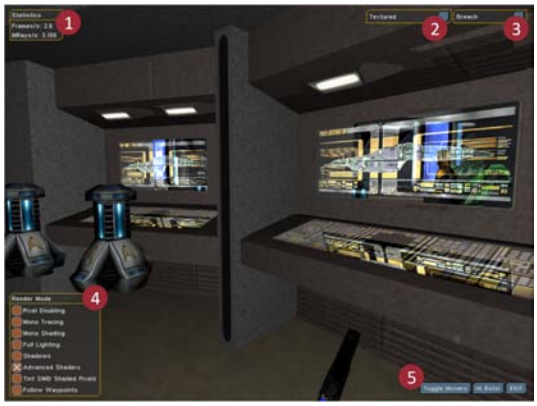
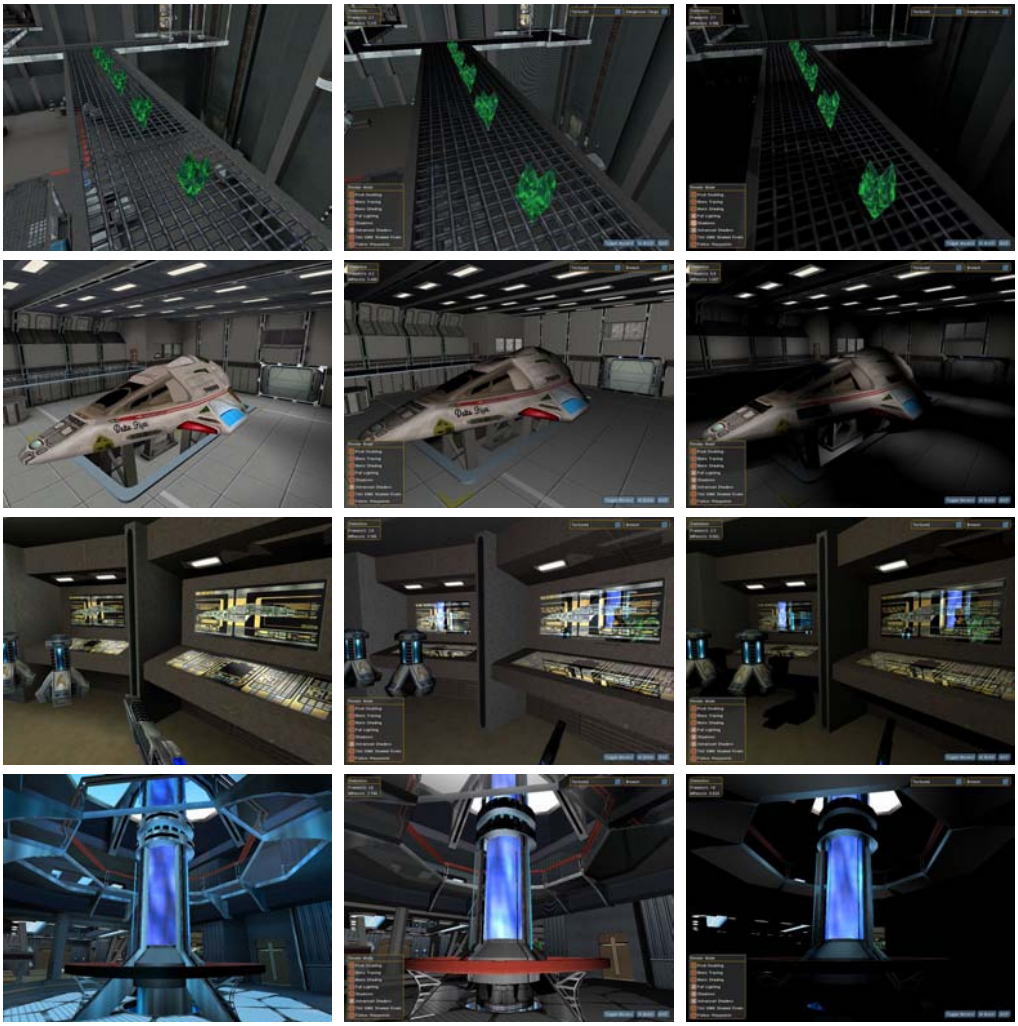


Figure 5.16: *1) Statistics encompassing the frame rate and the ray processing rate are updated continuously. 2) A series of rendering modes are supported, such as textured rendering, display of normal vectors and texture coordinates, and traversal statistics. 3) Maps can be loaded at run–time through a drop–down menu. 4) The behavior of the application can be modified, for example by enabling lighting, shadows, and advanced effects like reflections. 5) Moveable objects like doors can be activated. Player characters present in the scene can also be animated.*

The user interface also enables interaction with the scene by allowing the user to toggle moveable objects like doors and elevators between their states and to issue the playback of animations of player characters present in the scene. Another feature of the map viewer is its support for fly–throughs that can be created by the user by placing waypoints in the scene. Playback then uses spline interpolation to move the camera along the recorded path, which is particularly useful for recording of video clips.

5.3.3 Results

Figure 5.15 shows images rendered with the map viewer with full lighting effects.



a) b) c)

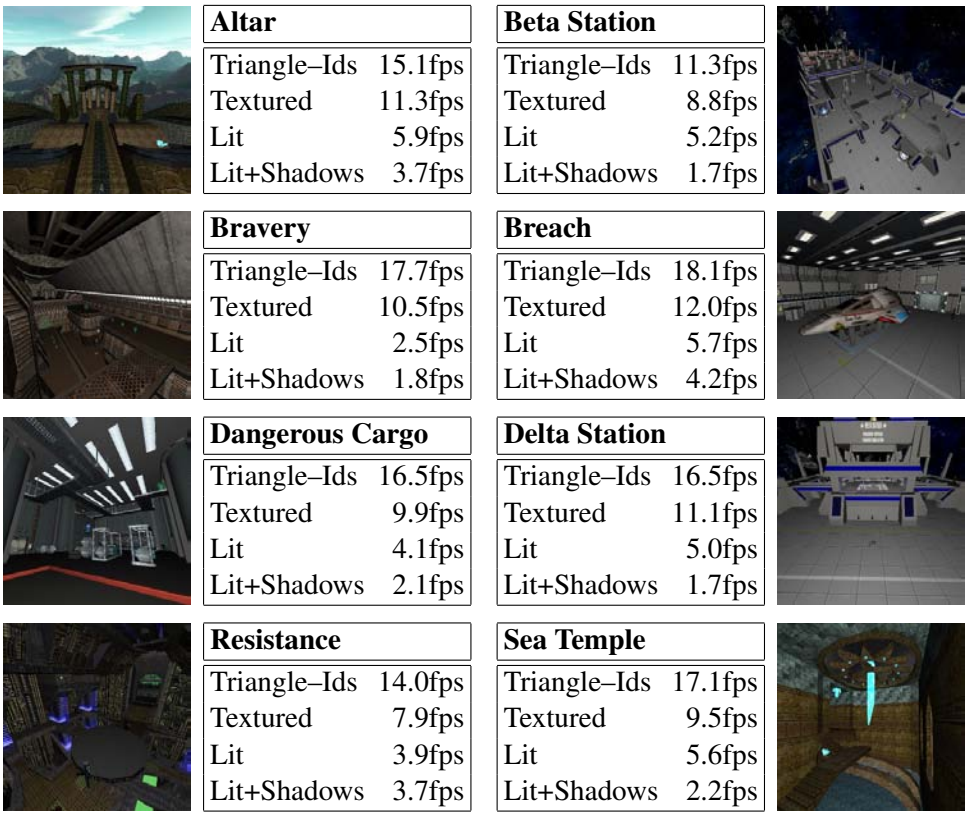
Figure 5.17: *a) Screenshots from Raven Software's original game Star Trek Voyager: Elite Force. b) Screenshots from the map viewer rendered at a resolution of* 1024×768 *pixels with the camera serving as a diffuse light source. c) Screenshots from the map viewer with shadows and full direct lighting from the light entities in the maps. The images appear darker because light emitted from surfaces is not taken into account and the camera no longer serves as a light source.*

Another set of images is presented in Figure 5.17 with images shown in the first column originating directly from Raven Software's game. Images in the second column show results from the map viewer developed in the context of this thesis with the camera serving as a point light. These images were rendered at a resolution of 1024×768 pixels at 2–4 frames per second.

The reflection effects implemented with recursive ray tracing are clearly visible in consoles and metallic structures. Although reflections are not directly supported by the shader language, they are applied selectively to the scene by recognizing patterns in the original shaders involving environment maps and replacing them with reflections.

Finally, the images in the third column present the scenes with additional lighting and shadows and were rendered at 1–2 frames per second at a resolution of 1024×768 pixels. It is important to note that these images appear darker than the other screenshots due to the fact that light emitted from surfaces (e.g. Voyager's warp core) is not taken into account and the camera no longer serves as a light source. By employing lighting data stored in the per–map lightmaps it would be possible to obtain results with similar lighting to the original game.

At a lower resolution of 512^2 pixels frame rates of up to 20fps were achieved for scenes from *Elite Force* when full lighting is disabled. Figure 5.18 presents performance data for a few selected views of different maps.



Altar	
Triangle–Ids	15.1fps
Textured	11.3fps
Lit	5.9fps
Lit+Shadows	3.7fps

Beta Station	
Triangle–Ids	11.3fps
Textured	8.8fps
Lit	5.2fps
Lit+Shadows	1.7fps

Bravery	
Triangle–Ids	17.7fps
Textured	10.5fps
Lit	2.5fps
Lit+Shadows	1.8fps

Breach	
Triangle–Ids	18.1fps
Textured	12.0fps
Lit	5.7fps
Lit+Shadows	4.2fps

Dangerous Cargo	
Triangle–Ids	16.5fps
Textured	9.9fps
Lit	4.1fps
Lit+Shadows	2.1fps

Delta Station	
Triangle–Ids	16.5fps
Textured	11.1fps
Lit	5.0fps
Lit+Shadows	1.7fps

Resistance	
Triangle–Ids	14.0fps
Textured	7.9fps
Lit	3.9fps
Lit+Shadows	3.7fps

Sea Temple	
Triangle–Ids	17.1fps
Textured	9.5fps
Lit	5.6fps
Lit+Shadows	2.2fps

Figure 5.18: *Performance data for selected maps from Raven Software's Star Trek Voyager: Elite Force rendered at a resolution of 512^2 pixels with different rendering modes. Lighting incurs a rather high performance hit due to the presence of a large number of lights in the maps, which were never meant to be used for direct lighting at run–time but only for the generation of lightmaps in a pre–processing step.*

5.3.4 Ray Tracing in the Original Game

In 2005 the source code for the *Quake 3 Arena* engine was released by id Software under an open source license and has since been maintained and developed further by the community [21]. This code base provided an excellent basis for implementing a new ray tracing based rendering backend for games that use the Q3A engine, such as *Elite Force* and *Quake 3 Arena*, which gave the engine its name.

Large fractions of the code from the map viewer could be reused for this purpose albeit with modifications related to interfacing with the Q3A engine. In particular loading of models and map geometry, parsing of shaders, and creation of textures no longer had to be performed by the ray tracer but were handled by the engine, which made the use of pre–processing tools obsolete in this context. However, the code generation for shaders posed a problem because in its form used by the map viewer it required the presence of a configured C++ compilation environment, which could not be expected to be available on non–development machines and would therefore have hindered the intended distribution of the program. The Low–Level Virtual Machine (LLVM) [30] was used to solve this problem by generating machine code for surface shading and animation at run–time [43].

Figures 5.19 and 5.20 present images from the fully playable games *Quake 3 Arena* and *Elite Force* rendered with the developed ray tracing backend for the Q3A engine.

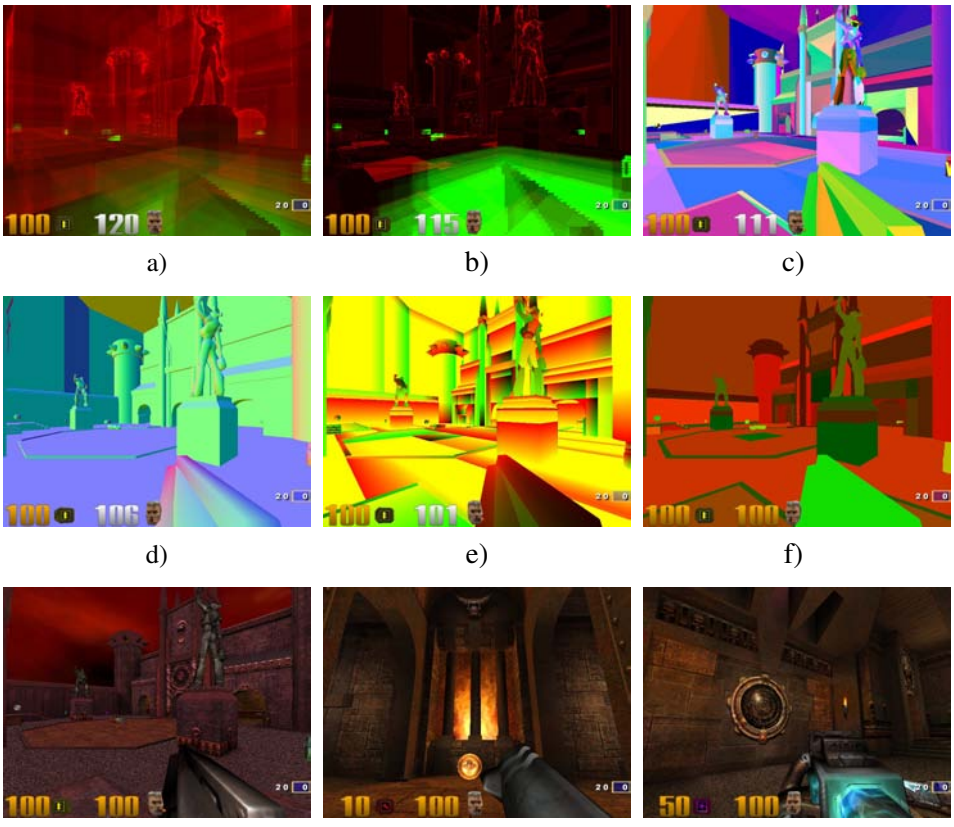


Figure 5.19: *All images were rendered at a resolution of* 800×600 *pixels using a new rendering backend for the Q3A engine [21] and the original assets from id Software's Quake 3 Arena [20]. a) Visited static (red) and dynamic (green) nodes (max. 128). b) Intersected static (red) and dynamic (green) triangles (max. 16). c) Color coding of triangles. d) Normal vectors. e) Texture coordinates. f) Color coding of shaders. g) View rendered with full effects at 4fps. h+i) Views rendered at 5fps.*

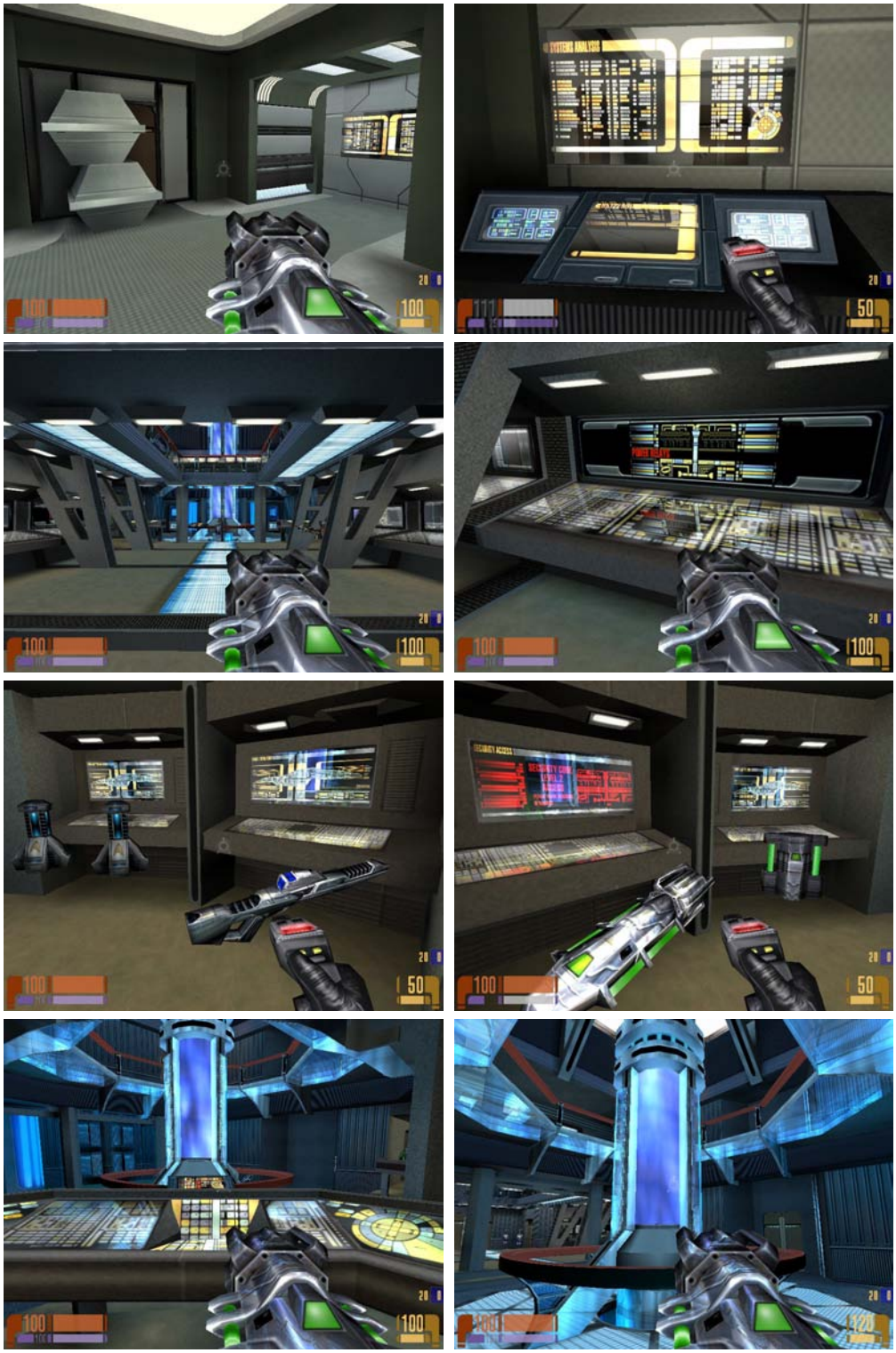


Figure 5.20: *Walk through the engineering room from Raven Software's Star Trek Voyager: Elite Force [41] rendered at a resolution of* 800×600 *pixels with the ray tracing based rendering backend developed in the context of this thesis.*

Chapter 6

Conclusions

In this thesis real–time ray tracing of dynamic scenes was explored based on a separation of static primitives from animated primitives in acceleration structures suited for each type of geometry.

For dynamic geometry a two–level bounding volume hierarchy was introduced that efficiently supports rigidly animated geometry, deformable geometry and fully dynamic geometry with incoherent motion and topology changes. Selective rebuilding was shown to be an efficient approach to updating a BVH after movement of primitives. It restricts costly rebuilding operations to degenerated parts of the hierarchy, which benefit from repartitioning of the contained primitives, and allows for balancing updating and rendering times. Run–time performance of ray traversal of BVHs could additionally be improved by using a new ordered traversal scheme based on a probabilistic model, which relies on pre–computed data to determine the near node in each traversal step.

Kd–trees were shown to exhibit better run–time behavior than BVHs when it comes to tracing of secondary rays, which typically account for a large proportion of all traced rays in a frame. They also yielded lower per–frame rendering times for some static scenes when tracing primary rays only and therefore remain the acceleration structure of choice for static geometry. By using a construction algorithm, which employs the surface area heuristic for finding optimal splitting planes, the best results in terms of run–time performance can be achieved. However, this typically results in a fine partitioning of primitives, which increases the overall size of the data structure.

In order to reduce the memory footprint of kd–trees two approaches were introduced: Index list compaction compresses the list of triangle indices used by leaves to reference triangles. The cost–scaling termination criterion for kd–tree construction, on the other hand, limits the creation of deep trees by weighing the costs of splitting a node higher with an increasing depth. It was shown that this allows for reducing the size of kd–trees by up to 80% without negatively affecting ray traversal performance. Furthermore, a speed–up could be measured for some scenes, which is an effect related to fewer traversal steps that need to be carried out on average in shallow kd–trees.

With the creation of a map viewer for animated scenes from Raven Software's game *Star Trek Voyager: Elite Force* the ability of the developed ray tracing library to deliver interactive frame rates for visually complex scenes was demonstrated on already relatively old hardware at the time of writing this thesis. Subsequently parts of the map viewer were integrated into a new rendering backend for id Software's engine from *Quake 3 Arena*,

which allows playing games based on the engine with graphics augmented with ray tracing specific effects, such as pixel–perfect reflections. The use of the Low–Level Virtual Machine (LLVM) for shader code generation in this context proved to be a very good choice with regard to both ease of integration and run–time performance.

However, the quality of the rendered images was compromised by the presence of aliasing effects related to under–sampling of textures. By employing ray differentials, as proposed by Igehy [22], and using mip–mapping or summed area tables for texture sampling this problem can be solved albeit with an increased number of rays that need to be traced, which may negatively affect the achieved frame rate.

Efficient tracing of secondary rays remains another problem that needs to be solved, which is mostly related to the fact that the assembly of large coherent packets is often not possible. Especially traversal of BVHs with small packets exhibits poor run–time behavior, as shown in chapter 5. By processing multiple tiles in an interleaved manner, as proposed in [42], this problem could be solved in that an increased number of active secondary rays would raise the potential for assembly of coherent packets. A problem of this approach, however, is the overhead of switching between tiles that used to be prohibitive when the technique was applied to primary rays previously.

In general it seems the time for real–time ray tracing has arrived in that processors with a higher number of cores in each generation will allow for rendering increasingly complex scenes due to the excellent scalability of the ray tracing technique.

Appendix

BART Measurement Reports

Machine:	AMD Athlon X2 3800+, 2×2 GHz	**Memory:**	3 GB
Model:	kitchen	**Frames:**	800
Primitives:	110,561	**Complexity level:**	-
Resolution:	800×600	**Mode:**	predetermined[1]
Average frame time:	0.84s	**Worst frame time:**	2.13s
Deviation:	0.35	**Continuity:**	0.71
Total time:	668.05s	**Preprocessing time:**	9.36s
Model:	museum	**Frames:**	300
Primitives:	14,239	**Complexity level:**	6
Resolution:	800×600	**Mode:**	predetermined[1]
Average frame time:	1.81s	**Worst frame time:**	4.87s
Deviation:	0.56	**Continuity:**	0.33
Total time:	542.29s	**Preprocessing time:**	1.19s
Model:	robots	**Frames:**	800
Primitives:	71,708	**Complexity level:**	-
Resolution:	800×600	**Mode:**	predetermined[1]
Average frame time:	1.37s	**Worst frame time:**	2.32s
Deviation:	0.31	**Continuity:**	0.36
Total time:	1,095.44s	**Preprocessing time:**	0.77s

[1] Static geometry was extracted.

Bibliography

[1] J. Arenberg. Ray/triangle intersection with barycentric coordinates. *Ray Tracing News*, 1(11), 1988.

[2] U. Assarsson and T. Möller. Optimized view frustum culling algorithms for bounding boxes. *Journal of Graphics Tools*, 5(1):9–22, 2000.

[3] D. Badouel. An efficient ray-polygon intersection. In *Graphics gems*, pages 390–393. Academic Press Professional, Inc., 1990.

[4] C. Benthin, I. Wald, and P. Slusallek. Interactive ray tracing of free-form surfaces. In *AFRIGRAPH '04: Proceedings of the 3rd international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 99–106. ACM, 2004.

[5] S. Boulos, I. Wald, and P. Shirley. Geometric and arithmetic culling methods for entire ray packets. Technical report, School of Computing, University of Utah, 2006.

[6] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

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

[8] M. Dabrovic. Model of the Atrium Sponza Palace, Dubrovnik, Croatia, 2001. URL `http://hdri.cgtechniques.com/~sponza/`.

[9] D. S. Fussell and K. R. Subramanian. Fast ray tracing using k-d trees. Technical report, University of Texas at Austin, 1988.

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

[11] S. A. Gottschalk. *Collision queries using oriented bounding boxes*. PhD thesis, The University of North Carolina at Chapel Hill, 2000.

[12] N. Greene. Detecting intersection of a rectangular solid and a convex polyhedron. In *Graphics Gems IV*, pages 74–82. Academic Press Professional, Inc., 1994.

[13] J. Günther, H. Friedrich, I. Wald, H.-P. Seidel, and P. Slusallek. Ray tracing animated scenes using motion decomposition. *Computer Graphics Forum*, 25(3):517–525, 2006. (Proceedings of Eurographics).

[14] E. Haines. Bounding box intersection via origin location. *Ray Tracing News*, 14(1), 2001.

[15] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2000.

[16] V. Havran. A summary of octree ray traversal algorithms. *Ray Tracing News*, 12(2), 1999.

[17] V. Havran and J. Bittner. On improving kd-trees for ray shooting. *Journal of WSCG*, 10(1):209–216, 2002.

[18] J. Hultquist. Intersection of a ray with a sphere. In *Graphics gems*, pages 388–389. Academic Press Professional, Inc., 1990.

[19] W. Hunt, W. R. Mark, and G. Stoll. Fast kd-tree construction with an adaptive error-bounded heuristic. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*. IEEE, 2006.

[20] id Software. Quake 3 Arena, 1999. URL `http://www.idsoftware.com/`.

[21] id Software, Z. Slater. ioquake3, 2005. URL `http://ioquake3.org/`.

[22] H. Igehy. Tracing ray differentials. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 179–186, 1999.

[23] T. Ize, C. Robertson, I. Wald, and S. G. Parker. An evaluation of parallel grid construction for ray tracing dynamic scenes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*. IEEE, 2006.

[24] P. Jaquays, B. Hook, J. Carmack, C. Antkow, K. Cloud, and A. Carmack. Quake III Arena Shader Manual, 1999. URL `http://www.heppler.com/shader/`.

[25] R. Jones. Intersecting a ray and a triangle with Plücker coordinates. *Ray Tracing News*, 13(1), 2000.

[26] D. Kalra and A. H. Barr. Guaranteed ray intersections with implicit surfaces. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 297–306. ACM, 1989.

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

[28] A. Kensler and P. Shirley. Optimizing ray-triangle intersection via automated search. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*. IEEE, 2006.

[29] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[30] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.

[31] C. Lauterbach, S.-E. Yoon, D. Tuft, and D. Manocha. RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 39–46. IEEE, 2006.

[32] J. Lext, U. Assarsson, and T. Möller. A benchmark for animated ray tracing. *IEEE Computer Graphics Applications*, 21(2):22–31, 2001.

[33] D. J. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *Visual Computer*, 6(3):153–166, 1990.

[34] J. Mahovsky and B. Wyvill. Fast ray-axis aligned bounding box overlap tests with plücker coordinates. *Journal of Graphics Tools*, 9(1):35–46, 2004.

[35] J. A. Mahovsky. *Ray tracing with reduced-precision bounding volume hierarchies*. PhD thesis, University of Calgary, 2005.

[36] E. Månsson, J. Munkberg, and T. Akenine-Möller. Deep coherent ray tracing. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing*, pages 79–85, 2007.

[37] T. Möller and B. Trumbore. Fast, minimum storage ray/triangle intersection. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 7. ACM, 2005.

[38] H. Mössenböck, A. Wöß, and M. Löberbauer. Der Compilergenerator Coco/R. In *Peter Rechenberg - Festschrift zum 70. Geburtstag*. Universitätsverlag Rudolf Trauner, Linz, Austria, 2003. URL `http://ssw.jku.at/coco/`.

[39] G. Müller and D. W. Fellner. Hybrid scene structuring with application to ray tracing. Technical report, Braunschweig University of Technology, 1999.

[40] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Experiences with streaming construction of SAH KD-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 89–94. IEEE, 2006.

[41] Raven Software. Star Trek Voyager: Elite Force, 2000. URL `http://www.ravensoft.com/`.

[42] S. Reiter. Offloading ray processing onto the GPU using cooperative worker threads. In *Poster Compendium of the 2006 IEEE Symposium on Interactive Ray Tracing*, page 8, 2006. Poster abstract.

[43] S. Reiter. Run-time code generation for materials. In *Poster Compendium of the 2008 IEEE Symposium on Interactive Ray Tracing*, 2008. Poster abstract (accepted for publication).

[44] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1176–1185. ACM, 2005.

[45] S. M. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. *SIGGRAPH Computer Graphics*, 14(3):110–116, 1980.

[46] SCI Institute at the University of Utah. The Utah 3D Animation Repository. URL `http://www.sci.utah.edu/~wald/animrep/`.

[47] M. Shevtsov, A. Soupikov, and A. Kapustin. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum*, 26(3):395–404, 2007. (Proceedings of Eurographics).

[48] B. Smits. Efficiency issues for ray tracing. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 6. ACM, 2005.

[49] Stanford University Computer Graphics Laboratory. The Stanford 3D Scanning Repository. URL `http://graphics.stanford.edu/data/3Dscanrep/`.

[50] I. E. Sutherland. Sketch pad a man-machine graphical communication system. In *DAC '64: Proceedings of the SHARE design automation workshop*, pages 6.329–6.346. ACM, 1964.

[51] I. E. Sutherland and G. W. Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974.

[52] C. Wächter and A. Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. In T. Akenine-Möller and W. Heidrich, editors, *Rendering Techniques 2006 (Proc. of 17th Eurographics Symposium on Rendering)*, pages 139–149, 2006.

[53] I. Wald. On fast construction of SAH based bounding volume hierarchies. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing*, 2007.

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

[55] I. Wald and V. Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–69. IEEE, 2006.

[56] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics*, 25(3):485–493, 2006.

[57] I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics*, 26(1):6, 2007.

[58] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the art in ray tracing animated scenes. In *Eurographics 2007 State of the Art Reports*, 2007.

[59] C. Wächter. Compilation of models for ray tracing of static scenes, 2006. URL `http://ompf.org/forum/viewtopic.php?f=4&t=64`.

[60] C. Wächter. *Quasi-Monte Carlo Light Transport Simulation by Efficient Ray Tracing*. PhD thesis, Ulm University, 2007.

[61] A. Williams, S. Barrus, R. K. Morley, and P. Shirley. An efficient and robust ray-box intersection algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 9. ACM, 2005.

[62] S. Woop, G. Marmitt, and P. Slusallek. B-KD trees for hardware accelerated ray tracing of dynamic scenes. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/Eurographics symposium on Graphics hardware*, pages 67–77. ACM, 2006.

[63] S.-E. Yoon and D. Manocha. Cache-efficient layouts of bounding volume hierarchies. *Computer Graphics Forum*, 25(3):507–516, 2006. (Proceedings of Eurographics).

[64] S.-E. Yoon, S. Curtis, and D. Manocha. Ray tracing dynamic scenes using selective restructuring. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, page 55. ACM, 2007.

Curriculum Vitae

Personal Details

Stephan Reiter
Date of Birth: September 18, 1984
Place of Birth: Linz, Austria

Education

Primary and Secondary School, 1991 — 2003, Linz, Austria.
Matura passed with distinction.

Bakk. techn., Informatik, 2004 — 2007, Johannes Kepler University, Linz, Austria.
Thesis entitled “Laufzeit–effizientes Raytracing”.

Publications

Stephan Reiter: Run–Time Code Generation for Materials. 2008 IEEE Symposium on Interactive Ray Tracing, Los Angeles, CA, USA, August 9 — 10, 2008, (accepted for publication).

Reinhard Wolfinger, Stephan Reiter, Deepak Dhungana, Paul Grünbacher and Herbert Prähofer: Supporting Runtime System Adaptation through Product Line Engineering and Plug–in Techniques. 7th IEEE International Conference on Composition–Based Software Systems, ICCBSS 2008, Madrid, Spain, February 25 — 29, 2008. Received the best paper award.

Stephan Reiter and Reinhard Wolfinger: Erfahrungen bei der Portierung von Delphi Legacy Code nach .NET. Nachwuchs Workshop. SE 2007 — the Conference on Software Engineering, Hamburg, Germany, March 27 — 30, 2007.

Stephan Reiter: Offloading Ray Processing onto the GPU using Cooperative Worker Threads. 2006 IEEE Symposium on Interactive Ray Tracing, Salt Lake City, UT, USA, September 20 — 22, 2006.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, Juni 2008 *(Stephan Reiter)*