Master Thesis

Ray Tracing Animations Using 4D Kd-Trees

Jens Olsson, π03
Lund University, Faculty of Engineering

April 21, 2007

Abstract

Static acceleration structures are for efficiency reasons the de-facto standard used in high-quality renderers today. Time discretization is vital for performance for such a system, forcing otherwise natural time-dependent phenomena like motion-blur to be accumulated together by multiple rendered images. In this thesis, new methods are presented for storing animation sequences in a time-dependent kd-tree, which includes a pre-processing step, and a modified surface area heuristic (SAH) cost function. A time-dependent distributed ray tracer was implemented as a plug-in for Softimage|XSI to test the methods in practice. Tests showed that the 4D kd-tree had better adaption to dynamic data compared to the 3D kd-tree even when both had the advantage of pre-processing. Applications of the 4D kd-tree were demonstrated to temporally displace instances of primitives to create the illusion of large groups of independently moving objects.

Contents

1 Introduction

Rendered animations have traditionally been generated frame-by-frame, constantly re-reading data for each concurrent frame. This makes for a simpler implementation, and most renderers only store the bare minimum of data that is necessary to generate the frame. Only a few years ago, this was still the only way to fit scene description data into the available memory on the computer systems. We can expect system bounds to expand considerably over the next years with 64-bit processing on common desktop computers. This will open up lots of new paths for research in speeding up rendering, since the complete data for a complex animation can fit into system memory, allowing processing to be carried out and reused over many frames. For simple scenes, even 2 gigabytes of memory could fit many frames of data, potentially utilizing the system resources better.

From a theoretical point of view, a time-continuous model fits better with many ray tracing techniques compared to a pre-discretized temporal model. Cook et al. [3] introduced distributed ray tracing in 1984, where rays are distributed in time as well as around the focal point, to properly capture effects like depth of field and motion blur. Kajiya [9] concretized the theoretical foundation further, describing what is commonly referred to as Monte Carlo ray tracing. Time-dependent Monte Carlo ray tracing integrates the final pixel value over the camera shutter time span, which requires sampling at arbitrary time instants, something which would not be possible with a discrete time model.

Currently, ad-hoc techniques are most often used to generate motion-blur and reuse global illumination calculations, such as post-processing motion blur and on-disk photon maps. Such techniques fail to generate physically correct effects, or simply require too large intervention from the end user. The complexities of a time-based renderer can also be outwon by the inherent savings of reusing global illumination calculations. Havran et al. presented a time-based bidirectional ray tracer [6], where they showed up to 10× speed-up over frame-by-frame rendering by elegant reuse of samples.

The question arises of how to store time-based animation in memory. Glassner [5] used Bounding Volume Hierarchies (BVHs) to good effect, where he showed considerable performance improvements for rendering the complete animation in one go. Havran [7] gives convincing empirical proof that the kd-tree data structure performs even better than BVH data structures in general. The kd-tree is a very simple data structure, where a volume is divided into subvolumes by axis-aligned splitting planes. When such a structure is used for sorting a set of primitives in space, the performance gains can be huge in a ray tracing system, because straight-forward traversal of the tree discards a majority of unnecessary intersection tests, a vital part of ray tracing.

We present an approach to extending the kd-tree to include time, proposing a simple heuristic to select a suitable splitting plane for the 4 dimensions. We also present a method to subdivide primitives in time into segments with small bounds in space. An interesting application of the time-dependent kd-tree is also demonstrated, where instances of objects are temporally displaced to give the illusion of individiual motion in a set of instanced objects.

2 Ray Tracing

Generating images by ray tracing, is essentially simulating the way images form in a camera in the real world. Kajiya defined *the rendering equation* in [9]

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int_\Omega f_r(x,\omega_o,\omega_i)L_i(x,\omega_i)(\omega_i \cdot n)d\omega_i \tag{1}$$

which describes how outgoing light L_o from a point p in direction ω_o is given by the sum of the emitted light L_e from the surface and the integral over all reflected light. The reflected light is described as the integral of light coming from all possible incoming directions ω_i over the hemisphere Ω at x, scaled by a function f_r, describing how the surface scatters the photons and a factor $\omega_i \cdot n$ which models how the energy is spread out over the surface when the incident angle decreases. The rendering equation is the most fundamental formulation of the rendering problem, and widely differing rendering techniques can all be reformulated as approximations of the rendering equation.

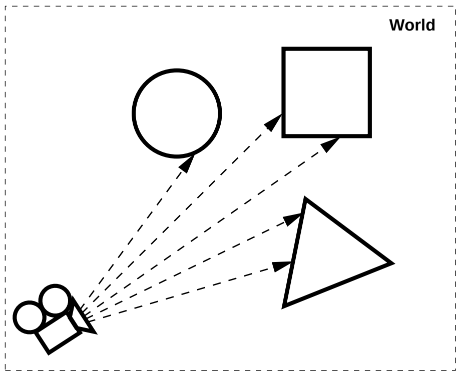

Figure 1: For each ray, the closest primitive along that ray has to be found.

In the simplest form of ray tracing, given a model of the world $\mathcal{W}$ and an imaginary camera $\mathbf{C}$, an image is simulated by intersecting rays originating from the camera with the world model, as shown in figure 1. Given a ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$, where $\mathbf{o}$ is the origin and $\mathbf{d}$ is the ray direction, there must exist a function $f(\mathcal{W},\mathbf{r}) = t_0$, where t_0 is the distance to the closest intersection point with the world along the given ray $\mathbf{r}$. There must also exist a function $f_\lambda(\mathcal{W},\mathbf{r}) = \lambda$, where λ is an estimate of the incoming light spectrum from the ray direction. Thus ray tracing is in nature very extendable, since the algorithm works unmodified as long as any primitives comprising the world model allows for the evaluation of the two functions. The function f_λ can be arbitrarily complex, it might itself do recursive ray tracing to compute its value. It may for instance model a reflective or refractive surface, or model an advanced global illumination solution, or calculate some simplified lighting from light sources in the world.

A suite of books has been written on the subject of ray tracing images. For the interested reader, an extensive treatment on the state of the art in ray tracing is given in [14]. A slightly easier read, although a bit dated, is given in [4]. In this thesis, we will mainly discuss data structures in ray tracing for accelerating ray intersection testing.

Given no prior knowledge of the world representation except a set of the primitives comprising the world, we have no other option but to intersect each ray with every primitive, since we cannot know in advance which of the primitives is closest to the ray origin. Thus for N primitives, the time complexity for finding the closest intersection point will be $\mathcal{O}(N)$. Given an image with size $m \times\ n$, we will thus have at a minumum $m \cdot n \cdot N$ intersection tests if we sample with one ray per pixel. If we assume quite reasonable numbers $m, n, N = 1024$, we have 1024^3 intersection tests to perform, over one billion possibly expensive intersection tests, which given only one ray per pixel will not even be sufficiently sampled to produce a visually pleasing image. We can clearly see that the computational costs will be unmanageable when the resolution and the number of primitives are large.

In the next section we will discuss ways to make ray tracing faster by using spatial data structures to reduce the number of ray-primitive intersections.

3 Previous Work

In general, every part of the ray tracing algorithm has to be carefully optimized. The naive algorithm has many computations showing complexity $\mathcal{O}(N)$, those computations will have complexity $\mathcal{O}(N^n)$ for n levels of recursive ray tracing. Thus any part of the algorithm naively implemented will overshadow any optimized part of the algorithm for a complex enough scene.

Common optimization techniques can be broadly categorized into three groups.

- *Fewer rays.* Reducing the amount of rays sent. Some techniques avoid initial rays from the camera, others limit the depth of recursive raytracing by some condition.
- *Generalized rays.* Ray tracing many rays simultaneously, by considering them as bundles or a larger cone structure.
- *Fewer intersection tests.* Reducing the amount of intersection tests by sorting primitives spatially or hierarchically.

For further reading, Arvo and Kirk gives a thorough but a bit dated survey of all three categories in [4]. In this thesis, we restrict ourselves to the third category of reducing intersection tests. The third category is by far the most important speed-up technique for ray tracers today, and a good approach for minimizing the intersection tests is vital for the performance of any given implementation.

There are a multitude of ways to try to reduce the amount of intersection tests that are performed. They all have one thing in common though, they can all be considered as data structures that sorts and organizes objects. We refer to all of these as “acceleration structures”. We divide existing research into three categories; static structures, dynamic structures and time-dependent structures.

3.1 Static Structures

For a computationally intensive simulation, it makes sense to spend some extra time calculating a more optimal acceleration structure. This has given headroom to the research of a vast amount of very different acceleration structures, since a complex structure with a high construction cost still could pay off for a complex enough simulation.

Most acceleration structures can be categorized as a variant or a hybrid of

- *Grids.* Dividing space into a grid of equally sized cells and hence fails to adapt to scene distribution. Very cheap to traverse, but must be traversed cell by cell along the ray. Variants include non-uniform grids and recursive grids, where a cell can be further subdivided if need.
- *Binary Space Partitioning Trees (BSPs).* Recursively dividing space into halves. Traversal can discard whole subtrees of the BSP directly. Variants like the kd-tree restrict the partitioning to arbitrarily positioned axis-aligned splitting planes. By careful choice of splitting planes the structure adapts well to scene distribution. More expensive to traverse compared to the grid, but fewer cells actually have to be visited due to better structure.
- *Bounding Volume Hierarchies (BVHs).* Enclosing primitives by simpler bounds that are cheap to intersect. Hierarchical trees can be built by in turn bounding local primitives together, so that large sets of primitives can be discarded very cheaply by a single intersection test. When an intersection test succeeds though, every primitive inside the bound will have to be tested. Adapts well to scene distribution.
- *Ray-Space Subdivisions.* Sort primitives as candidates for a discretized set of possible ray origins and directions. A 5D-structure that is complex to construct and has very high memory requirements. First mentioned by Arvo and Kirk in [4].

These structures all rely on primitives having a bound. We will restrict examples to axis-aligned bounding boxes, the smallest non-rotated box-shape that fully encloses a primitive. We refer to such a bound as $\mathbf{B}(P)$ for a given primitive P. Consider a model M of a car moving a certain distance over time $\mathbf{t} = [t_0, t_1]$. If we were to put a spatial bound $\mathbf{B}(M)$ on the car, the bound would be

$$\mathbf{B}(M) = \mathbf{B}\big(\cup_i \mathbf{B}(M_{t_i})\big) \ , \ t_i \in \mathbf{t} \tag{2}$$

thus spanning the entire volume the animation of the car travels through. If we were to store an entire animated scene in a static structure, traversal of the structure would in extreme cases result in having to test every primitive for intersection, because the bounds of the objects can potentially extend over the whole scene. We can conclude that static structures will have problems separating objects when primitive bounds grow because of time dynamics.

3.2 Dynamic Structures

Recent advances in real-time ray tracing has led to other needs in acceleration structures. When the ray-tracer produces as much as 30 frames per second, interactive applications become attractive. The few minutes it takes to construct a kd-tree for high-quality ray tracing becomes far too expensive, the ray-tracer needs to rebuild the scene in milliseconds.

Wald et al. has reported promising results in a series of papers. In [15], uniform grids are used with some careful techniques to efficiently ray-trace interactive scenes. In [16], dynamic BVHs are introduced, that maintains topology over time so that only bounding volumes need to be updated.

Woop et al. in parallel with Keller et al. as well as Havran et al. researched hybrids of a kd-tree and a BVH in [17], [10] and [8]. The data structures are very similar, and differ only in minor details. For each cell, instead of a single splitting plane, they use two planes, that can be positioned so that they bound primitives in the cell. For well behaved motion, the trees can be constructed so that they can be quickly updated for geometry changes by re-positioning the splitting planes.

There exists many more such structures that are being used for real-time ray tracing, but they all have in common that they either rely on rebuilding a new structure quickly, or rely on well-behaved motion to be stored.

3.3 Time-Dependent Structures

For an animation sequence of any length, it can not be assumed that the motion will be in any way well-behaved. We will in fact require the structure to be able to handle arbitrarily complex motion, simulated, pre-scripted or even completely random, as any such sequence is trivial to construct using existing 3D software. A data structure that efficiently treats a pre-scripted motion sequence is in a sense both dynamic and static, because it from frame to frame describes dynamic motion, but as a sequence it is static.

Glassner introduces the term “Spacetime Ray Tracing” in [5], where he describes the concept of ray tracing objects in 4D spacetime. In such a space, a moving 3D object could be treated as a static 4D object. Glassner uses a hybrid of space subdivision and BVHs, and reports noticeable savings in rendering time. For one sequence, a 20% speed-up is reported, and for the other test sequence a speed-up of almost 50% is noted.

The paper is followed up in [13] by Quail, where he compares Glassner’s spacetime structure with a 4D extension of ray classification. Even for simple ray classification, the memory requirements are very large, and Quail notes that his extension as expected uses much more memory than Glassner’s structure. For one example he mentions that the 4D ray classification uses almost 1000 times the amount of memory of the BVH hybrid. Quail reports that the ray classification method is up to 50% faster than the BVH hybrid for scenes with localized movement. For wide ranging movement, ray classification showed a performance improvement of 30%, but it was noted that the scheme generated very large hierarchies, hinting that ray classification treated localized motion much better.

Havran et al. [6] introduce a complete system for rendering animations, which include advanced effects such as global illumination and motion blur. They present a method to render multiple frames at once, by updating samples over the frames, compensating for camera or object motion, thus avoiding recalculation of expensive calculations. They use a global kd-tree for static objects and also nest kd-trees containing animated primitives within the global tree. The nested trees stores instances of any animated primitive, one instance for each frame. Comparing their method with frame-by-frame rendering showed a speed-up factor of around 10×.

Kato et al. reports in [11] that they used a grid structure for their Kilauea project but subdivided moving primitives temporally. There is sparse information on exact method or performance, but we expect their method to be similar to ours.

Given the popularity of the kd-tree in ray tracing, we propose a concrete extension of the structure to include time. We try to get a measure of how the increased complexity affects processing time, and show that the 4D kd-tree is a viable alternative to hybrid structures.

4 Kd-Trees

The kd-tree was introduced by Bentley [1]. The k originally denoted dimension, e.g. a 3-dimensional kd-tree was meant to be referred to as a *3d-tree*. A kd-tree is in structure a BSP-tree, with the difference that the BSP-commonly splits a volume into two by a midsection splitting plane. The kd-tree loosens this restriction and allows the splitting plane to be arbitrarily positioned, but axis-aligned. Havran [7] showed that the kd-tree performed better than the general BSP for all tested scenes. The kd-tree has a much better adaption to scene distribution, which pays off when the structure is traversed.

The kd-tree is characterized by modest memory requirements. The amount of leaf cells grows roughly linearly to the number of primitives, and a total memory limit can be used as termination criteria for the construction phase.

Positioning of the splitting planes greatly influences the performance of the structure in ray-tracing. There are though existing algorithms for generating suitable kd-trees for ray-tracing, and efficient algorithms for traversing them. We give an overview of the algorithm described by Havran [7], which currently is the de-facto standard.

4.1 Construction

Given a set of primitives $\mathcal{P} = \{P_i\}$, we construct an axis-aligned bounding box $\mathcal{B} = \mathbf{B}(\mathcal{P}) = \mathbf{B}\big(\cup_i \mathbf{B}(P_i)\big)$, which will be the bounds of the kd-tree structure.

The volume $\mathcal{B}$ will either be split into two non-overlapping subvolumes $\mathcal{B}_A$ and $\mathcal{B}_B$, or the splitting algorithm is terminated and a list of all primitives that overlap the volume is stored. For each subvolume $\mathcal{B}_A$ and $\mathcal{B}_B$, it is again decided if they are split into two subvolumes, or the splitting is terminated. Thus the kd-tree can be constructed by a recursive algorithm, each step subdividing space into finer subvolumes, see figure 2.

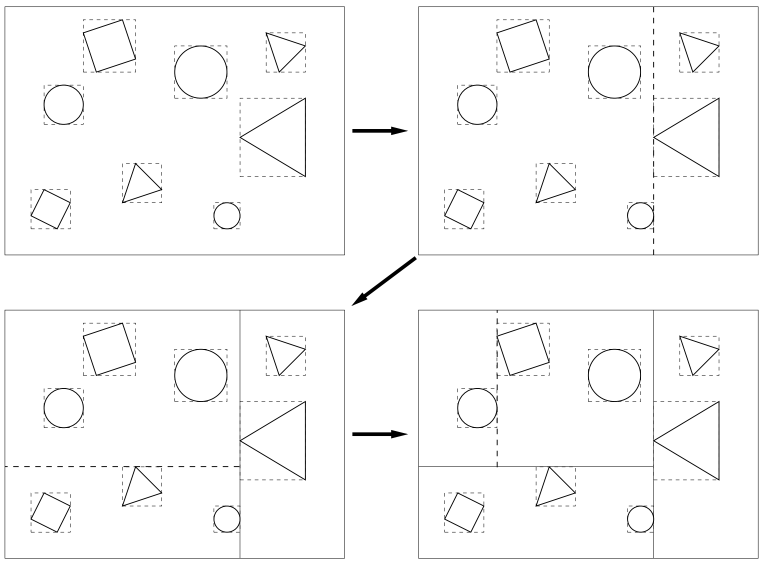
Figure 2: The enclosing volume is recursively split into subcells

4.1.1 Termination Criteria

Termination criteria are important for making the tree efficient, and are often set as user parameters for tuning the performance. A limit on how many primitives may be stored in a tree cell forces the construction algorithm to keep subdividing space until the primitives are sufficiently separated. A high leaf count in most cases leads to unnecessary intersection tests. It is also common to impose a max tree depth as a termination criterion as a large tree depth can affect the performance negatively, resulting in a large amount of processing used on traversing the tree nodes. Since the structure is built top-down, the construction can also be terminated when the structure reaches a maximum memory size. Such criterions are in conflict with each other, so that enforcing one termination criteria may lead to other criterions not being fulfilled. Restricting memory usage may force large number of primitives to be stored in some tree cells, as well as keeping a low max primitives count may force the tree depth higher than the max tree depth states. It is thus necessary to order any termination criteria based on priority.

4.1.2 Splitting

There are a multitude of ways of subdividing a volume $\mathcal{B}$ into two subvolumes. The kd-tree in its design defines the subdivision to be made by axis-aligned splitting planes. For any distribution of objects, there exists splitting surfaces that are better adapted to the scene than such a simple plane. The complexity to find the best of such surfaces is substantial though, and carries over even to arbitrarily oriented splitting planes. Havran [7] states that the number of possible positions of an arbitrarily oriented splitting plane is of $\mathcal{O}(N^3)$, increasing construction time cubically as the number of primitives N grows linearly. Thus the kd-trees restriction of axis-aligned splitting planes is reasonable, and with an arbitrarily positioned axis-aligned splitting plane, the number of possible positions is $\mathcal{O}(N)$. Axis-aligned splitting planes also pay off when traversing the structure, with only trivial arithmetic calculations for the traversal. Simpler schemes, such as splitting along the spatial median, unfortunately does not pay off, as they do not properly take into account the real cost of traversal, and generally produces badly adapted structures.

MacDonald and Booth introduced the term *surface area heuristics* (SAH) in [12], which is extensively treated by Havran [7]. The principle idea is to guide the construction of the tree based on estimated costs of actually ray-tracing the structure. The probability of a ray intersecting a subvolume $\mathcal{B}_A$ is given by the surface area ratio $p(\mathcal{B}_A|\mathcal{B}) = \frac{S_{\mathcal{B}_A}}{S_{\mathcal{B}}}$, given that the ray intersects $\mathcal{B}$. For estimated costs of intersection and tree traversal, it is possible to locally estimate costs of terminating splitting and creating a leaf, or estimate costs of a set of possible subdivisions. Pharr and Humphreys [14] uses a simplified, locally greedy algorithm, where the cost C is calculated as

- *Leaf*

  $$C = \sum_{k=1}^{N} c_i(k)$$

  All primitives will have to be intersected, where $c_i(k)$ denotes time cost of intersecting primitive k. Pharr and Humphreys also makes the assumption that intersection costs do not vary much in their implementation, and thus simplify the cost further to

  $$C = Nt_i$$

- *Split*

  $$C = c_t + p_A \sum_{k=1}^{N_A} c_i(a_k) + p_B \sum_{k=1}^{N_B} c_i(b_k)$$

  c_t denotes the cost associated with finding which of the subcells the ray intersects. p_A and p_B are the conditional probabilities $p(\mathcal{B}_A|\mathcal{B})$ and $p(\mathcal{B}_B|\mathcal{B})$ defined earlier, which defines the probability that a ray intersecting $\mathcal{B}$ intersects the respective subvolume. N_A and N_B denotes the number of primitives overlapping the two regions $\mathcal{B}_A$ and $\mathcal{B}_B$, and $\{a_k\}$ and $\{b_k\}$ are index lists such that $\{a_k; k = 1, \ldots, N_A\}$ represent the indices of all primitives overlapping $\mathcal{B}_A$, and vice-versa for $\{b_k\}$ and $\mathcal{B}_B$. Assuming that intersection costs do not vary significantly, as in the leaf case, this can be simplified as

$$C = t_t + (1 - b_e)(p_A N_A t_i + p_B N_B t_i) \tag{3}$$

where the b_e term is introduced to favor splits where one of the subcells is empty, and designed so that $b_e \in [0, 1]$ when one of the cells is empty, and zero otherwise.

For each cell, we can see that the cost function is modeled such that each of the subvolumes will be leaves, since the probabilities of the ray intersecting one of the cells is directly multiplied with leaf cost functions $C_A = N_A t_i$ and $C_B = N_B t_i$, corresponding to N_A and N_B intersection tests of primitives with no additional traversal. Thus such a scheme does not produce a globally minimized cost tree, because each of the subcells can be further split, so that the estimated cost will differ from the true cost. On the other hand, a globally minimized cost tree would require the subcell costs to be computed correctly. This requires the tree to be fully constructed, which contradicts the fact that we are searching for the best construction choices to begin with. Havran [7] states but does not prove that the problem of finding such a minimum total cost kd-tree is NP-hard.

For each volume $\mathcal{B}$, a cost C is calculated for any candidate splitting plane, as well as the cost for making the cell a leaf. The splitting planes that we are interested in are the ones which minimize the cost, and we can discard a lot of possible splitting positions directly. If we are considering splitting along a fixed axis in the vicinity of one primitive, we imagine placing the splitting plane in the center of the primitive. The primitive will now overlap both $\mathcal{B}_A$ and $\mathcal{B}_B$, adding to both N_A and N_B. Subtly moving the splitting position along the axis does not alter N_A and N_B, only changing the surface areas of the cells and thus changing the p_A and p_B terms. But because $s_A + s_B = k_1 \Rightarrow p_A + p_B = k_2$ this only linearly changes the cost, indicating that a local minimum exists at a primitive boundary along the axis. When the splitting plane is moved to one of the boundary edges, the primitive will no longer overlap one of the subcells, further reducing either N_A or N_B by one, resulting in a jump down in the cost function on the boundary. Moving the splitting position further beyond the primitive boundary will result in the inclusion of empty space in the cell volume and increased probability that the primitive will have to be tested, and thus gives a higher cost again. The same reasoning holds for multiple primitives and overlapping primitives. Only object boundary edges thus need to be considered as candidate splitting positions. For N primitives, we will then have to compute up to $3 \cdot 2 \cdot N$ costs, since for each axis, each object has two boundary edges.

The smallest cost determines leaf or split, and after splitting, the algorithm will recursively be applied to the resulting subcells.

4.2 Traversal

There are a few variations on traversing a kd-tree efficiently. Common methods are categorized into two groups

- *Sequential traversal.* The algorithm steps cell by cell, so that when one cell has been tested for intersection, a new ray traversal origin is set slightly beyond the cell border, and a new candidate cell containing the current traversal ray origin is searched for in the kd-tree. The algorithm thus walks from cell to cell until either an intersection is found, or the ray exits the kd-tree without any hit. Each look-up in the kd-tree will require multiple visits to many nodes, because the search always starts from the root node.

- *Recursive traversal.* The tree is traversed top-down, so that any subcells are tested in the order that the ray passes through them. The algorithm is applied recursively for each subcell so that the closest cells will be tested down to leaf level before more distant cells are considered. Guarantees that each node is visited only once, but traversal steps require more bookkeeping than sequential traversal, because distant cells to be tested have to be stored until closer lying cells have been processed.

The recursive algorithm generally has better performance for ray tracing applications, and is the more common method to traverse kd-trees. A number of variants are covered in [7]. We choose to present one of the simpler methods.

For each ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ that is intersected with a kd-tree root node $\mathcal{N}$, we find a parameter range $t_{\mathcal{N}} = [t_0, t_1], t_i > 0$ corresponding to the intersection points with the node bounding volume, see figure 3. If the node is a leaf, all primitives are simply intersection tested, returning the closest hit in the parameter range if any intersection test succeeds. If the node is split, the parameter t_{split} is calculated, corresponding to the intersection of the ray with the splitting plane. If $t_{split} > t_1$ only the near child is intersected by the ray, because the splitting plane intersection with the ray lies beyond the ray exit point from the cell. If $t_{split} < t_0$, the ray only passes through the far cell, because the splitting plane intersection with the ray is closer to the origin than the ray entry point in the cell. Otherwise both cells have to be tested, and the algorithm will be recursively applied with new parameter ranges $t_{near} = [t_0, t_{split}]$ and $t_{far} = [t_{split}, t_1]$ for the respective near and far cell. The far cell intersection testing is postponed until the near cell subtree has been completely traversed, and it is stored in a stack structure for later processing.

In such a way the algorithm proceeds, identifying which of the two cells are intersected, updating the parameter range appropriately and recursively traversing the tree from closest to furthest object. Given that intersection testing constitute a great proportion of ray tracing processing time, it does pay off to optimize both the kd-tree data structures and the traversal algorithm. Extra information is often stored to speed up look-ups and traversal in the tree. The method above is relevant though, because it forms the basis of the more optimized algorithms.

5 Extending the Kd-Tree

Given an animation sequence on the time interval $\mathbf{t} = [t_0, t_1]$, an arbitrary number of frames $\{f_i\}$ spread out over the time interval are sampled using ray-tracing. We would like to consider one such frame f_i as a discretized time instant, but one frame f_i really corresponds to the exposure on the image plane of the virtual camera. If we correctly model a camera, the camera shutter remains open for a time interval Δt_s, which for large exposure times gives the

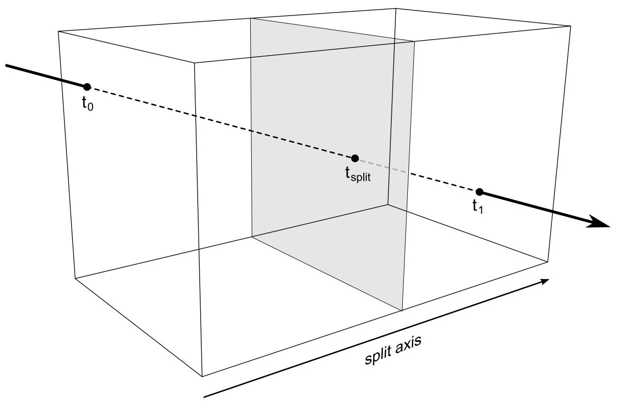

Figure 3: The order of the intersection points along the ray gives information about which of the subnodes that need to be tested for possible intersection.

motion blur effect. A set of frames $\{f_i\}$ thus corresponds to a set of continuous time slices. The traditional way of rendering motion blur further discretizes each such time slices into n subframes, but such methods are not popular in production rendering, since they easily produce artifacts for wide-ranging motion. In addition, the rendering time increases by a factor n.

More correct methods do treat each time segment correctly as continuous time, but render each time segment independently. Storing the complete animation sequence does show both advantages and disadvantages. If the shutter time is small, treating each time segment independently will read the minimum amount of data necessary, while storing the entire animation sequence will read data that is inbetween time slices and thus never used. On the other hand, for long shutter times, time slices may well start to overlap, making the independent treatment read the same data over and over. For a single camera it does not make sense that the shutter intervals can overlap, but in special effects photography, individual cameras are often used for each frame. Extreme motion blur effects in general can hold great value for certain special effects, which the continuous time model handles automatically.

We believe there is a need to clarify the implications of using a kd-tree for storing animation data, as was also suggested by Havran in [7]. We believe that the inherent simplicity of the structure can have many advantages over hybrid structures used for storing dynamic data.

In ray-tracing, the term kd-tree has become synonymous with its 3-dimensional specialization, much because a 3D structure fits naturally into the problem domain, since most processing in ray-tracing are 3D visibility queries of some sort. It does not make as much sense to fit a 4D structure to such a rendering problem, because primitive time bounds extend over the entire time span of the sequence in most cases, in principle offering no candidate splitting positions along the time axis.

We construct a simple method to preprocess the animation data so that it can be stored in a 4D kd-tree trivially. We also motivate using cost functions in line with SAHs for splitting the tree.

5.1 Preprocessing Data

Given the task of rendering N frames $\{f_i\}$, we have to store animation data for the entire time interval $\mathbf{t} = [t_{f_1}, t_{f_N} + t_s]$ where t_{f_i} is frame i shutter open time and Δt_s is shutter time. As mentioned in Section 2, spatial bounds for an animated object for the entire time interval $\mathbf{t}$ are in most cases extremely inflated compared to the spatial bound at a time instant t_i. Large spatial bounds as well as temporal bounds both indicate a need to describe the primitive in more detail over time. As we are trying to fit the data into a 4D kd-tree, it is essential to introduce new candidate splitting positions along the time axis. This only seems possible by subdividing the primitive in time into subprimitives, defined over separate non-overlapping time segments. This do solve both bound issues, since the animated spatial bounds for each such subprimitive will be reduced due to the smaller time interval, in turn giving us candidate splitting positions in time for the kd-tree.

We propose a simple iterative method for subdividing primitives in time which is flexible enough to be used on both continuous and discretized data. We denote the entire time interval for the animation, including shutter time, as $\mathbf{t}_\Omega = [t_0, t_1]$. We set time $t_a = t_0$, and for a primitive P we find the largest time $t_b \le t_1$ such that

$$\mathbf{S}(\mathbf{B}(P_{t_a})) \ge \frac{1}{\kappa}\ \mathbf{S}\big(\mathbf{B}\big(\cup_i \mathbf{B}(P_{t_i})\big)\big)\ ,\ \ t_i \in [t_a, t_b] \tag{4}$$

where $\mathbf{S}(\mathbf{B}(P))$ denotes the surface area of the axis-aligned bounding box of primitive P. This essentially restricts how much we allow the primitives animated spatial bound to inflate from the initial static bound at time t_a. We are interested in the static 4D bounding box of each subprimitive, so from a traditional point of view we enclose moving objects in 3D bounding boxes for short spans of time. Thus the parameter κ describes how much extra surface area we can allow for a cell. A higher κ will lead to more unnecessary intersection tests, while a lower κ will lead to more subprimitives being generated.

For the interval $[t_a, t_b]$, we create a new subprimitive $\widetilde{P}$ that defines the same motion as the primitive P, but only exists on the interval. We then iterate the method by setting $t_a = t_b$ and repeating the process until $t_a \ge t_1$. The resulting N subprimitives $\widetilde{P}_i$, as depicted in figure 4, all have controlled spatial bounds, and a temporal bound smaller than t_Ω for $N > 1$.

The motivation for using the surface area to model equation 4 is directly related to the surface area heuristics used in construction of a kd-tree with good performance. The probability of a ray entering a cell is not related to volume, only surface area. This can be easily visualized if one considers an axis-aligned triangle or another flat primitive, thus having a spatial bound with zero volume. If the triangle is increased in size, the probability of a ray intersecting the object is increased, which is reflected in the fact that the spatial bound surface area increases, while the volume will remain zero.

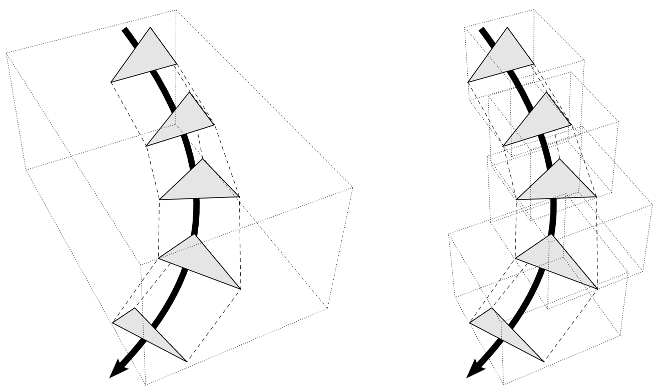

Figure 4: Splitting a primitive with large spatial bounds into smaller primitives with smaller spatial bound.

Choosing a suitable parameter κ is not directly obvious. $\kappa \leq 1$ would mean that we do not allow the surface area to increase, which in theory will force infinitely many subprimitives. Arvo and Kirk state in [4] that the projected area of a volume is on average 1/4th of the surface area of the volume, giving a hint that if for example $\kappa = 2$, we allow the surface area to double, the average projected area facing a given ray will also double, in effect increasing the probability of intersecting the primitive by a factor κ compared to intersecting the static spatial bound at t_a.

The parameter κ could vary between primitives, and could additionally be set by the user, but we believe an extra parameter for controlling the kd-tree construction is unwanted, proposing that κ be set as a fixed value, optimized for the implementation for a variety of scenes.

5.2 Construction

Given the preprocessed data, we have a set of unsplit primitives $\{P_i\}$, and a set of generated subprimitives $\{\widetilde{P}_j\}$, where the temporal bound edges of $\widetilde{P}_j$ offer potential candidate splitting positions for the 4D kd-tree. We would like to use the concept of surface area heuristics also for the extended kd-tree, since we again want to minimize an estimated cost of intersecting a candidate tree node. Surface areas of 4D bounding boxes which have mixed spatial and temporal axes might give an indication of trouble, but on the temporal and spatial scale of photo realistic ray tracing, it is not unreasonable to model photons and thus rays as arriving instantaneously, which will simplify the SAH concept.

As we treat the time axis as just another spatial axis in the tree, we have to relate in some way the cost of a spatial split to a temporal split. Thus we cannot blindly design some ad-hoc rule, unless it relates spatial and temporal axes in a consistent way. Returning to the cost function

$$C = t_t + (1 - b_e)(p_A N_A t_i + p_B N_B t_i)$$

explained in detail in equation 3 on page 11, we see that the term surface area heuristics in some way is misleading, because the probability terms and all the other components very easily relates to the temporal split. Given a 4D volume $\mathcal{B}$ with temporal bounds $\mathbf{t} = [t_0, t_1]$, denoting the time interval length $|\mathbf{t}| = |t_1 - t_0|$. The primitive $\mathcal{B}$ is split into two subvolumes along the time axis such that $\mathcal{B}_A$ is defined on $\mathbf{t}_A = [t_0, t_{split}]$ and $\mathcal{B}_B$ is defined on $\mathbf{t}_B = [t_{split}, t_1]$. For a given ray $\mathbf{r}$ intersecting the volume $\mathcal{B}$, assuming light travels infinitely fast, the ray will never intersect both subvolumes, as the ray only exists in a specific time instant. Thus the ray either intersects $\mathcal{B}_A$ or $\mathcal{B}_B$, giving the probabilities

$$p_A = \frac{|\mathbf{t}_A|}{|\mathbf{t}|} \qquad p_B = \frac{|\mathbf{t}_B|}{|\mathbf{t}|} \tag{5}$$

and thus $p_A + p_B = 1$. We have thus reinterpreted the given cost function so that costs for a spatial axis can be compared to a cost for the temporal axis, without explicitly relating spatial and temporal axes.

Using this cost function makes construction of the 4D kd-tree trivial. We can reuse the same construction algorithm as for the 3D kd-tree, modified to 4 axes instead of 3. The only difference in treating the temporal axis is in replacing the spatial surface area based probabilities by the temporal probability equations given in eq 5.

5.3 Traversal

As for construction, traversal of the kd-tree is a simple modification of the original recursive traversal algorithm. 4 axes have to be iterated instead of 3, and for a temporal split, simply looking at the time of the ray $\mathbf{r}$ and the splitting position in time determines which one of the subcells to be further traversed.

6 Implementation

While implementing a test framework for the 4D kd-tree, the option was to either extend an existing ray tracer, or the daunting task of developing a new renderer. The interface to animation data was also crucial, and it was early on decided that the framework would be tightly integrated with existing 3D software, which would allow us to read animation data at arbitrary resolution. Since a lot of the ray tracer internals would be time dependent, which none of the candidate ray tracing implementations were, we chose to develop a new ray tracer.

The test framework is developed in C++, and is due to time constraints quite limited in scope. It renders triangles with textures, reflections, refractions, shadows, and most importantly, physically correct motion blur. It does not model global illumination, as this has been well covered in other works. The framework core is not tied to a specific software, but the current client implementation was developed as a plug-in for Softimage|XSI, a high-end 3D animation software. It should be noted that the results are not dependent on this choice, as any given 3D software could have been used as animation source instead.

6.1 Reading Data

Scene primitives were divided into static and animated. Static objects were simply read as static triangles set to exist over the entire animation time span. Animated objects most often are hand animated, but they could equally well be objects from a rigid-body simulation, which differ a bit in source, as the hand animated object is controlled by well defined keyframes and the resulting interpolating curves, while simulated objects are effectively discretized motion. Mimicking exactly the interpolation of animation curves inside the 3D software would allow us to read only the keyframe data, but we would soon have a replication of the entire animation engine of the 3D software, which intuitively sounds unlikely to be successful. We thus choose to treat any animated object by discretizing the motion.

Our initial model samples n subframes for each animation frame of the sequence. We do not want to sample more sparsely than once each frame, because the eye is extremely sensitive to motion path artifacts. In our test framework, data is linearly interpolated for simplicity, and the parameter n is here mainly a way for the user to control motion blur smoothness. If a smoother interpolation method was chosen, like Catmull-Rom splines [2], one motion sample per frame could possibly be enough to approximate the motion.

For N frames, each sampled by n subframes, we store the vertex data for each triangle in a separate array. Any triangle sharing a vertex can thus use the same vertex data array. As we use the preprocessing method described in section 5.2 to segment the triangle in time, all subprimitives generated reference the same vertex arrays. In this way, no data is duplicated due to the splitting, and the only cost introduced in the segmentation is the cost of the subprimitive structures. The major overhead in memory will thus be the massive amounts of animation data stored as vertex arrays, but the amount of this data is known, so that it is easy to calculate how many frames of data the memory can hold, making it possible to divide a very complex animation into a set of frame blocks that all fit into memory.

Animated lights and cameras are also sampled as coordinate arrays describing their motion paths over time. Animated attributes are not supported, but it would be simple to also sample these.

The abstraction between 3D software and ray tracing implementation produces alot of memory overhead as memory is duplicated inbetween them. It would be interesting to couple them tighter so that the ray tracing implementation does not store any data. Then one would build a 4D kd-tree in the same way as before, but never store triangles explicitly in the tree. The advantages would be obvious, letting the 3D software interpolate data, so that to the ray tracing implementation, the data can still be viewed as continuous in time.

6.2 Construction

We were able to squeeze a 4D kd-tree node into 8 bytes, by using yet another bit of the splitting position, as is traditionally done for the 3D kd-tree nodes (see [14] for details), without noticing any artifacts. This results in no memory penalty for using the 4D kd-tree due to node memory size, as they are exactly the same size as the 3D counterpart.

We explicitly avoided relating temporal and spatial axes, as we did not want to introduce yet another arbitrary variable. This leads to a performance penalty in our implementation because we chose to calculate temporal costs initially, and then comparing to the cost for the dominant spatial axis.

7 Results

We tested the data structure on a variety of scenes, and present a subset of these here. We compare memory overhead and performance with the traditional accumulation motion blur as well as comparing to stochastic motion blur limited by traditional 3D kd-trees.

Comparing accumulation motion blur to stochastic motion blur is very difficult for a number of reasons. We cannot compare them based on total number of rays, because rays in the stochastic case will be far more expensive due to the interpolation of triangles for every intersection test. The accumulation motion blur quality is also balanced between the subframe sampling versus the number of subframes sampled, making it impossible to declare a definitive render time for a given test scene. Accumulation motion blur artifacts usually occur as strobing effects, while the stochastic motion blur artifacts result in large amounts of noise, again making it difficult to compare quality between them. For these reasons, our comparison will be highly subjective, and should be regarded as an example of the rendering performance for a given set of options.

We also show some interesting applications of the 4D kd-tree, where instanced primitives are displaced in time to create an illusion of more complex data.

7.1 Propeller

The first test scene is a simple propeller consisting of about 1000 triangles that features wide-ranging motion. The reference image in figure 5 was rendered with stochastic motion blur using 128 samples per pixel and 8 motionsteps linearly interpolated.

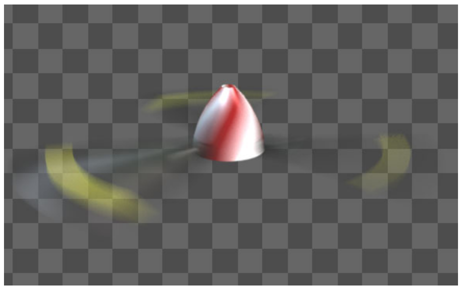

Figure 5: Reference image for the propeller test scene.

We chose to render the accumulation motion blur test with adaptive sampling for each subframe with a maximum of 4 samples, due to that the final frame will be accumulated. We rendered using 8, 16, 32 and 64 subframes, each subframe rendered using a 3D kd-tree. For the stochastic motion blur tests, we used the 4D kd-tree with 8 motionsteps, and an adaptive sampling of max 8, 16, 32 and 64 samples per pixel.

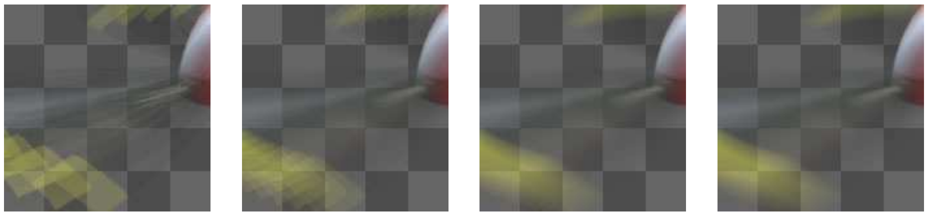

Figure 6: Accumulation motion blur test results.

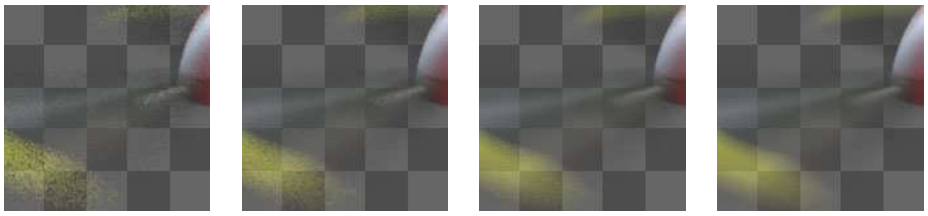

Figure 7: Stochastic motion blur test results.

Propeller test common data

method	accelerator	tree memory size	primitives memory size
accumulation	3D kd-tree	15.3 kB	83.3 kB
stochastic	4D kd-tree	384.3 kB	664.4 kB

Propeller test results

accumulation	8 subframes	16 subframes	32 subframes	64 subframes
time (minutes)	0.75	1.48	3.30	6.51
total rays	796925	1594750	3190770	6381735

stochastic	8 samples	16 samples	32 samples	64 samples
time (minutes)	0.44	0.84	1.67	3.33
total rays	653510	1423635	2980755	6083995

Propeller test rendering time breakdown (seconds)

accumulation	8 subframes	16 subframes	32 subframes	64 subframes
update data	2.77	5.47	12.20	24.41
build tree	0.14	0.26	1.17	1.74
raytrace	41.62	82.75	184.63	363.71

stochastic	8 samples	16 samples	32 samples	64 samples
update data	0.58	0.47	0.47	0.45
build tree	0.45	0.45	0.47	0.47
raytrace	25.13	49.30	99.20	198.96

For these particular settings the two methods seem to match each other quite well in quality and amount of rays traced. The stochastic motion blur tests all render a bit faster, probably because rays more efficiently can be used on interesting areas, whereas the accumulation buffer method wastes more rays tracing unimportant areas for every subframe. The human eye is more forgiving to noise than strobing artifacts, and the quite noisy stochastic motion blur images do look better when played in a sequence. The accumulation buffer motion blur tests do converge to a very good quality image, and for many scenes the supersampling could possibly be lowered.

One attractive aspect of the stochastic motion blur method is the fact that we practically have motion vectors for each sample for free, since these can often be extracted directly from the interpolation step of the triangles. Motion vectors are playing a large role in production rendering today for post-processing motion blur. Although such post-processing techniques do fail to correctly model motion blur, carefully combining rough stochastic motion blur with post-processing blur is a good approximation to fully sampled motion blur.

We also compared the performance of stochastic motion blur while using the tradidional 3D kd-tree instead of our 4D kd-tree. We made one set of tests for the basic primitives, and one test where the primitives were subdivided in time using our segmentation method.

Stochastic rendering time comparison (minutes)

kd-tree	segmented	8 samples	16 samples	32 samples	64 samples
3D	no	1.2	2.4203	4.93723	9.98932
3D	yes	0.49167	0.95130	1.89583	3.78645
4D	yes	0.43802	0.83828	1.66927	3.33178

Stochastic rendering time breakdown (seconds)

kd-tree	segmented	update data	build tree
3D	no	0.484	0.031
3D	yes	0.484	0.625
4D	yes	0.469	0.468

Notably, the construction time for the non-segmented 3d kd-tree is significantly lower, primarily because the number of primitives is a tenth of the number of primitives generated by the segmentation method. More interesting is that the build time for the 3D kd-tree is higher than the 4D kd-tree for the same number of primitives, which indicates that the 3D kd-tree algorithm has problem sorting the primitives efficiently.

There seems to be a slight improvement in rendering performance in using the 4D kd-tree. The traversal cost is comparable between the two, so the rendering times indicates that the 4D kd-tree has better adaption to the generated data.

7.2 Head

The second test scene is a slightly heavier mesh, consisting of around 10k triangles, deformed by a twist deformer applied on the mesh and rendered with extreme motion blur effects. The reference image was again rendered with 128 samples per pixel, using stochastic motion blur.

The shutter time for the test spanned over 10 frames, which forces the accumulation blur method to use quite many subframes to converge to a continuous

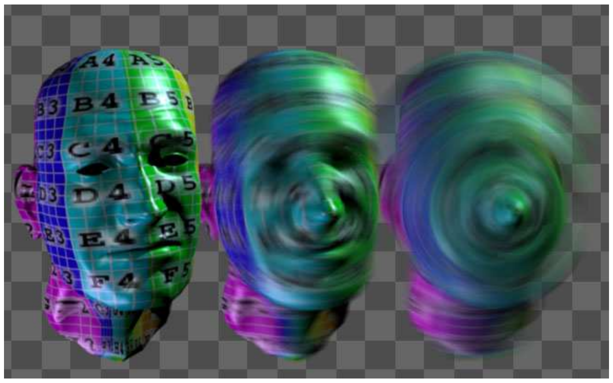

Figure 8: The head test scene, the right-most image shows the motion blur settings used in the test.

motion blur effect. We used 8, 16, 32 and 64 subframes for the accumulation motion blur effect, while 4, 8, 16 and 32 samples per pixel of stochastic sampling gave a comparable quality due to better sample distribution in time. Tree depth and leaf size were restricted to 32 and 8.

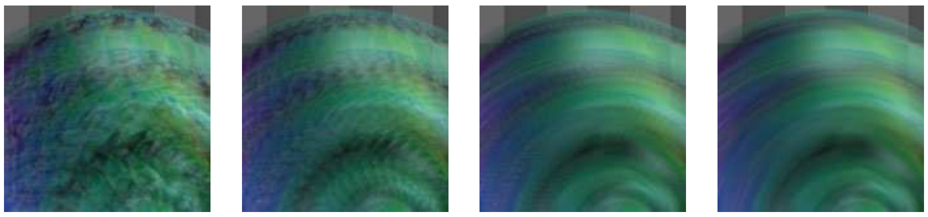

Figure 9: Accumulation motion blur test results.

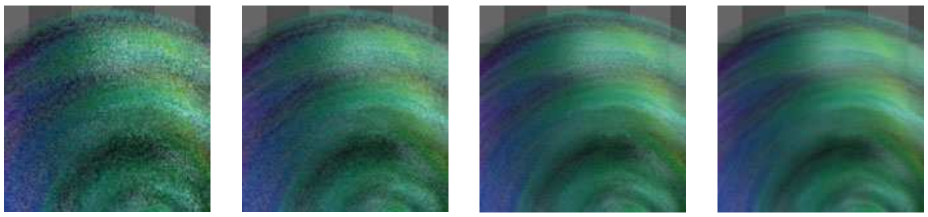

Figure 10: Stochastic motion blur test results.

Head test common data

method	accelerator	tree memory size	primitives memory size
accumulation	3D kd-tree	0.2 MB	0.9 MB
stochastic	4D kd-tree	18.4 MB	12.2 MB

Head test results

accumulation	8 subframes	16 subframes	32 subframes	64 subframes
time (minutes)	1.3	2.6	5.0	8.9
total rays	1961956	3909952	7806048	15599764

stochastic	4 samples	8 samples	16 samples	32 samples
time (minutes)	1.2	1.9	3.4	6.2
total rays	933204	1886628	3822216	7687188

Head test rendering time breakdown (seconds)

accumulation	8 subframes	16 subframes	32 subframes	64 subframes
update data	4.3	8.3	16.3	26.1
build tree	2.6	5.5	10.3	17.6
raytrace	70.0	143.5	271.6	487.6

stochastic	4 samples	8 samples	16 samples	32 samples
update data	1.5	1.5	1.5	1.5
build tree	26.6	26.4	26.0	26.0
raytrace	45.8	87.9	173.7	344.8

We notice right away that for the compared render settings, the accumulation motion blur method can trace twice as many rays as the stochastic motion blur method in the same amount of time. We notice that the construction time for the 4D kd-tree is far larger compared to the 3D kd-tree, even more than we saw in the propeller test case. The extreme motion blur subdivides the 10k triangles to over 220k subprimitives, which explains the increase in build time.

We also get a hint that a lot of data is being generated when we look at the size of the kd-tree in memory, so even though the method clearly handles extreme motion blur, memory gets consumed quite rapidly, even though the kd-tree nodes occupy the same amount of space in both the 3D and the 4D implementation.

Stochastic rendering time comparison (minutes)

kd-tree	segmented	4 samples	8 samples	16 samples	32 samples
3D	no	16.7	36.3	65.4	130.7
3D	yes	2.2	3.5	6.1	11.1
4D	yes	1.2	1.9	3.4	6.2

Stochastic rendering time breakdown (seconds)

kd-tree	segmented	update data	build tree
3D	no	1.2	0.5
3D	yes	1.5	51.9
4D	yes	1.5	26.0

It does seem naive to expect a 3D kd-tree to hold animation data for 10 frames and still render it efficiently, as the rendering times for the non-segmented case are extremely long. Segmenting the primitives improves rendering performance a lot, but as in the propeller case, the 4D kd-tree build time is actually lower, indicating that the 3D kd-tree builder must again have trouble constructing the tree. The speed-up of rendering using the 4D kd-tree is significant.

The difference in time when updating the data is actually the time to subdivide primitives into subprimitives, so for this scene, the segmentation time is approximately 0.266 seconds.

7.3 Armadillo

The third test scene is a dense mesh: the armadillo from the Stanford 3D Scanning Repository, comprised of around 350k triangles, and deformed by a bone structure with animation. We used 2, 4, 8 and 16 subframes of accumulation motion blur, compared to 2, 4, 8 and 16 samples per pixel for stochastic motion blur. The shutter time spanned 4 frames, and the tree depth and leaf size were limited to 24 and 8. Raising the κ segmentation to 4 for this scene gave a significant speed increase.

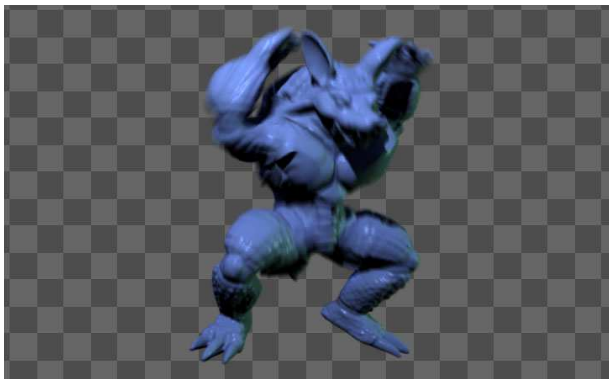

Figure 11: The armadillo test scene.

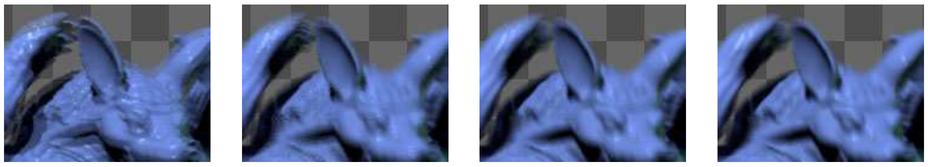

Figure 12: Accumulation motion blur test results.

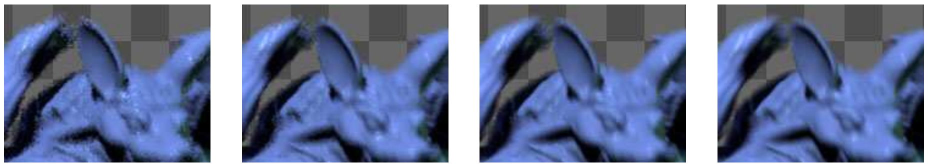

Figure 13: Stochastic motion blur test results.

Armadillo test common data

method	accelerator	tree memory size	primitives memory size
accumulation	3D kd-tree	4.7 MB	38.7 MB
stochastic	4D kd-tree	20.1 MB	144.7 MB

Armadillo test results

accumulation	2 subframes	4 subframes	8 subframes	16 subframes
time (minutes)	1.0248	1.9813	4.2963	9.15208
total rays	583460	1166384	2332464	4665212

stochastic	2 samples	4 samples	8 samples	16 samples
time (minutes)	3.1	3.8	5.0	7.3
total rays	569680	1110336	2169484	4159652

Armadillo test rendering time breakdown (seconds)

accumulation	2 subframes	4 subframes	8 subframes	16 subframes
update data	8.5	13.6	27.8	54.3
build tree	28.9	58.3	134.4	296.6
raytrace	22.8	45.3	92.1	191.5

stochastic	2 samples	4 samples	8 samples	16 samples
update data	22.4	22.5	22.5	22.5
build tree	116.6	116.5	116.6	116.5
raytrace	46.1	83.7	156.9	294.4

For such a dense mesh, the kd-tree build time starts to become significant in the total rendering time. Our kd-tree implementation could possibly be further optimized, but the problem lies in the method of segmenting the primitives, thus multiplying the number of primitives by a significant factor. For this scene for example, the 3D kd-tree for accumulation motion blur has to sort 345926 primitives, i.e. the number of triangles in the scene. This results in about one million nodes in the kd-tree for the test kd-tree settings. When we consider animated triangles, and segment these temporally for this scene, we get more than four million primitives to sort for $\kappa = 2$. The total kd-tree node count for such large scenes can quickly rise to tens of millions.

This is one of the effects of the design choices of our method; we restrict the growth of the bounding box of any animated primitive, but with the penalty that the primitive count instead will rise. Such a dense mesh as this is comprised of extremely tiny triangles in screen space, so even though the motion blur effect is not very extreme, the tiny triangles motion paths can be extremely elongated relative to their size, meaning that our method will segment the triangle into possibly hundreds or thousands of new subprimitives.

Stochastic rendering time comparison (minutes)

kd-tree	segmented	2 samples	4 samples	8 samples	16 samples
3D	no	3.2	5.4	10.1	18.2
3D	yes	3.3	4.0	5.4	8.1
4D	yes	3.1	3.8	5.0	7.3

Stochastic rendering time breakdown (seconds)

kd-tree	segmented	update data	build tree
3D	no	20.8	26.6
3D	yes	22.5	118.8
4D	yes	22.5	116.5

For very low sampling, the 3D and 4D kd-trees render in almost the same time, but looking at the build times, it seems that the 4D kd-tree is still faster in rendering, the build time mainly equalling them out. For higher sampling rates, the 4D kd-tree clearly is better adapted for the generated data, as was indicated earlier in the propeller and head test cases.

7.4 Temporally Instanced Primitives

A common technique in rendering is to create a set of lightweight primitives that all reference a template primitive. Even though the scene may be comprised of thousands or millions of primitives, each object is only a carbon copy, referencing data stored only once in memory. Storing the template primitive in a static 3D acceleration structure will naturally restrict all copies to exactly the same time instant, which is the reason that instancing static objects is the most common.

Being able to sort the template primitive once into a time-dependent 4D acceleration structure has some attractive features for instancing techniques. We can store the complete animation for a template primitive, and with simple means create copies that exist displaced in time relative to the template primitive, since the 4D acceleration structure allows us to ray-trace at any given time instant without having to rebuild the data.

We implemented a simple prototype for instancing animated primitives. We derived a simple subtype of the primitive class, which contains a reference to the template primitive, a spatial transform as well as a temporal transform. Care has to be taken to update the bounding box appropriately, since the bounding box of the template primitive will most likely no longer be axis-aligned after transformation. In our implementation, we modeled the template primitive as defining a cyclic animation, so that any time sample outside the source animation time span was displaced back into the time interval.

We created a simple animation cycle of a bunny that comprised 20 frames. No static instancing technique can create the same effect as temporal instancing, the only option is to duplicate data for each copy. We made some simple tests of 1, 10, 100, 1000, 10000, 100000 and 1000000 bunnies, measuring memory overhead when duplicating the data versus instancing temporally. Duplicating the data, memory allocation grew at a linear rate proportional to the number of primitives, as is natural to expect. The memory overhead for the larger primitive counts were thus extrapolated. Figure 16 depicts the difference in memory overhead for the two cases.

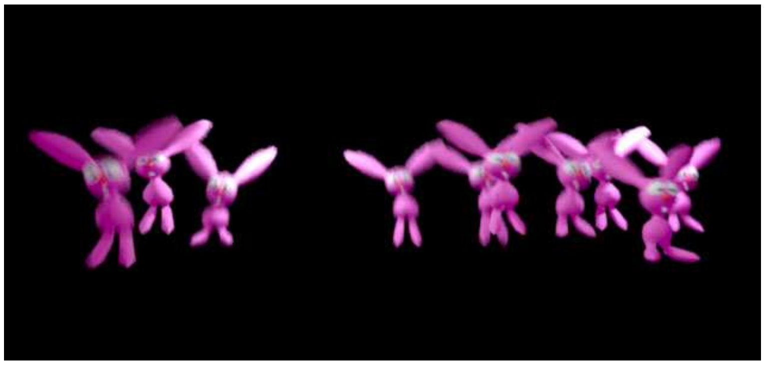

Figure 14: 10 temporally instanced primitives.

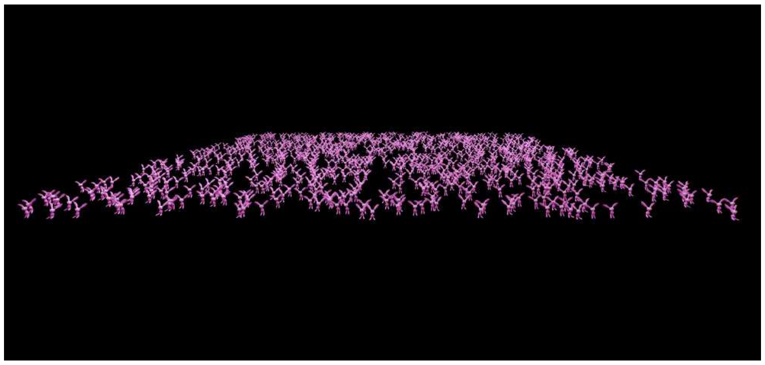

Figure 15: 1000 temporally instanced primitives.

As expected, rendering even a million primitives is very cheap using temporal instancing. There is an initial overhead of the template primitive, clearly visible in figure 16 in the initial few samples of the temporal instancing curve, and we expect that for heavier template primitives, the overhead will be even more pronounced. The template primitive overhead will of course also limit the total number of primitives possible to fit into memory. The final images, see figure 14 and figure 15, really shows the value of arbitrarily displacing the instances in time. The illusion of individual bunnies is extremely good, even though they are all cheap copies of one single animation source.

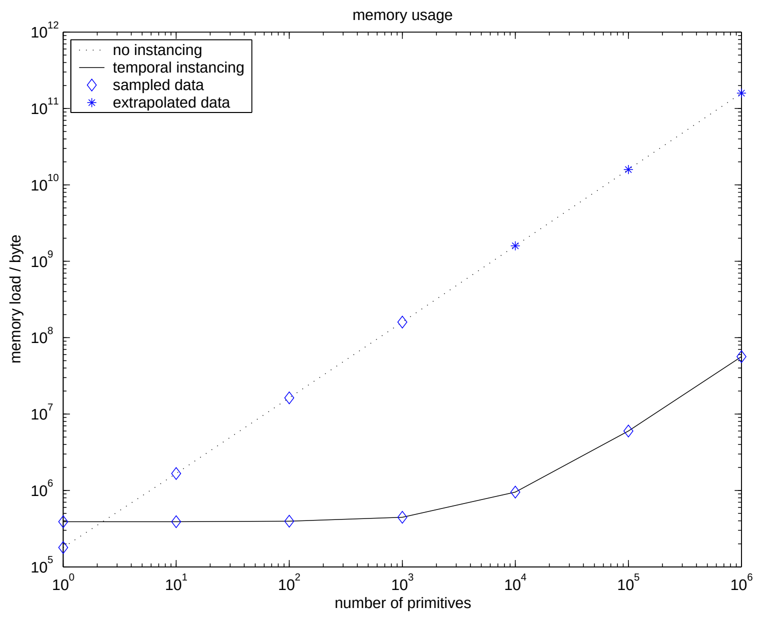

Figure 16: Memory load when rendering a given number of character primitives.

8 Conclusions and Future Work

In this thesis, we presented methods to store complete animation sequences in 4D kd-trees. A straight-forward method was also introduced to subdivide primitives along the time axis to limit bounding box inflation due to motion. A cost function was formulated that naturally extended traditional kd-tree surface area heuristics to accomodate time.

Using the method on a set of test scenes indicated that the extra costs in both construction time and memory load can pay off in rendering time due to better scene adaption of the structure.

We did not optimize the code for interpolating the triangles, and we believe further work into fast cpu-optimized triangle interpolation would be very important to validate storing animation data in memory. We currently estimate costs at least twice for every node, as we explicitly avoided relating spatial and temporal axes. It would be interesting to further study how to avoid these double cost calculations, preferably without trivially relating the axes.

We also demonstrated using the 4D kd-tree to create temporally displaced instanced primitives, which is not possible with a static acceleration structure unless data is duplicated explicitly for each copy.

Temporal instancing holds tremendous potential for practical applications. It would be very interesting to see further work in using it to render massive crowd simulations. It would simply be a matter of authoring a seamless set of animation loops as are traditionally used in games, storing these once in memory for every template character, and switching between them as the simulation progresses.

9 Acknowledgements

I would like to thank my supervisors Petrik Clarberg and Tomas Akenine-Möller for valuable discussions and comments along the way. I would also like to thank all the authors mentioned in the bibliography, as their works gave me the inspiration and energy for writing a thesis in computer graphics.

References

[1] Bentley, J. L. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM.* 18(9) pp. 509-517. 1975.

[2] Catmull, E., Rom, R. A class of local interpolating splines. *Computer Aided Geometric Design.* pp. 317-326. Academic Press, 1974.

[3] Cook, R. L., Porter, T., Carpenter, L. Distributed Ray Tracing. *Proceedings of ACM SIGGRAPH 84.* pp. 137-145. ACM Press, 1984.

[4] Glassner, A. S. (Ed.). *An Introduction to Ray Tracing.* Academic Press, 1989.

[5] Glassner, A. S. Spacetime Ray Tracing for Animation. *IEEE Computer Graphics and Applications.* 8(2) pp. 60-70. 1988.

[6] Havran, V., Damez, C., Myszkowski, K., Seidel, H. An Efficient Spatio-Temporal Architecture for Animation Rendering. *ACM SIGGRAPH 2003 Sketches & Applications.* 2003.

[7] Havran, V. *Heuristic Ray Shooting Algorithms.* PhD thesis, Czech Technical University, 2000.

[8] Havran, V., Herzog, H., Seidel, H.-P. *On the Fast Construction of Spatial Hierarchies for Ray Tracing. Proceedings of IEEE Symposium on Interactive Ray Tracing.* pp. 71-80. 2006.

[9] Kajiya, J. T. The Rendering Equation. *Siggraph '86 Proceedings.* 20(4) pp. 143-150. ACM Press, 1986.

[10] Keller, A., Wächter, C. Instant Ray Tracing: The Bounding Interval Hierarchy. *Rendering Techniques 2006: EuroGraphics Symposium on Rendering.* 2006.

[11] Kato, T. The "Kilauea" Massively Parallel Ray Tracer. *Practical Parallel Rendering.* pp. 249-328. AK Peters, 2002.

[12] MacDonald, J. D., Booth, K. S. Heuristics for ray tracing using space subdivision. *Visual Computer.* 6(6) pp. 153-165. 1990.

[13] Quail, M. *Space Time Ray Tracing using Ray Classification* Macquarie University, 1996.

[14] Pharr, M. and Humphreys, G. *Physically Based Rendering.* Morgan Kaufmann, 2004.

[15] Wald, I., Ize, T., Kensler, A., Knoll, A. and Parker, S. G. *Ray Tracing Animated Scenes using Coherent Grid Traversal. SIGGRAPH '06: ACM SIGGRAPH 2006 Papers.* 25(3) pp. 485-493. ACM Press, 2006.

[16] Wald, I., Boulos, S. and Shirley, P. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics.* 2006.

[17] Woop, S., Marmitt, G. and Slusallek, P. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. *Proceedings of Graphics Hardware.* 2006.