ACCELERATING RAY TRACING FOR DYNAMIC SCENES USING KD-TREE MERGING

Eleftherios Eleftheriades

Supervising professor: Yiorgos Chrysanthou

A Dissertation Submitted in Partial Fulfillment of the requirements

for the Degree of Bachelor of Science at the University of Cyprus

June, 2010

ACKNOWLEDGEMENTS

First of all I would like to thank my supervisor, Yiorgos Chrysanthou, who had the idea for this research, and for his indispensible guidance and help.

I would like to thank Marios Papas, for his great contribution to the coding, measurements and algorithmic aspects of this research.

ABSTRACT

In ray-tracing, a method for rendering graphics by casting rays from the screen to the 3d geometry of the scene, an acceleration structure is built inorder to accelerate the ray/triangle intersection checks. Kd-Trees built using the cost estimation function of the Surface Area Heuristic are an example of such a structure. However they require a considerable amount of time to be constructed. In a dynamically changing environment, this entire structure needs to be constructed every time a part of the geometry changes.

There are two issues to consider when using such (or in fact any) acceleration structure. Build the structure carefully in order to accelerate rendering, or sacrifice the optimality of the structure in an attempt to speed up construction times.

In this dissertation we follow a different approach. We try to avoid reconstruction of the entire scene. In most cases, only a small part of the scene changes, while the bigger part such as buildings and trees remains static. One could create the tree of the static objects once, and then merge it to the tree of the dynamic objects that will be constructed each time. We propose a method for efficiently merging these trees.

Eleftherios Eleftheriades – University of Cyprus, 2010

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Computer graphics is a computer science discipline that is traditionally involved with the representation of images on a computer. It's worth distinguishing real time computer graphics from offline rendering. Real time computer graphics find applications in games, virtual reality, simulators etc. In these applications, the computer generates 30 or more images -called frames- per second. Offline rendering is usually met in movies, where more time is spent to produce more realistic images.

Rendering is the process of generating a 2d image from a 3d representation of the scene. The 3d scene is generally represented by polygons or simple geometric objects called primitives. Currently there exist two main methods for rendering, rasterisation and ray-tracing.

Rasterisation projects the 3d space on the 2d plane which is our screen, then discritizes the projections to find the pixels covered by the projected polygons. This is a very fast method for rendering, because it's supported by the GPU hardware. Rasterisation however lacks realism, because it's not physically based. Effects such as caustics, subsurface scattering and generally global illumination have to be added using separate methods adding greatly to complexity and time.

Ray-tracing, can be used for generating realistic images. The rendering process simulates the rays of light as they interact with the 3d scene. Ray-tracing suggests that we shoot rays from each pixel of the screen to our scene, and follow the path of those rays, as they intersect with objects, get reflected, refracted etc. This method produces images with a much higher degree of realism. In order to properly trace the path of a ray, we need to check where exactly each ray intersects our scene. A naive way of doing this, is to test each ray against all the scene's polygons to find the first polygon that is intersected. This has cost O(Rays x Polygons). Considering the resolution of the screen and number of polygons that may exist in a scene, these checks are expensive in terms of time. This problem led to the invention of acceleration structures that help reduce the number of polygon intersection tests.

1.1 Acceleration structures for ray-tracing

Researchers in ray-tracing have proposed several structures for organizing the scene so the number of intersection tests is greatly reduced. These structures are divided in to two categories: bounding volume hierarchies and space subdivision.

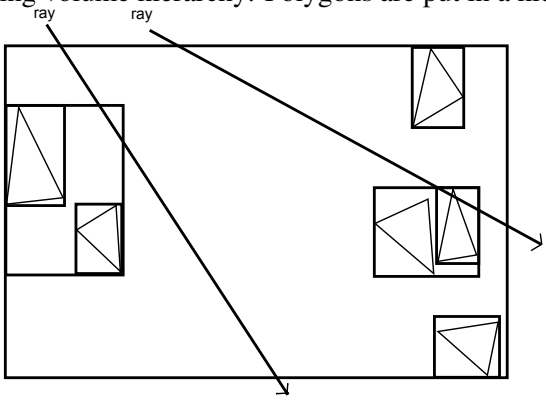
1.1.1 Object hierarchies

An obvious way to reduce the cost of intersection is to fit a bounding volume around each polygon or a group of polygons. If the ray doesn't intersect the bounding volume then no further test is necessary. However if a ray intersects the bounding volume a further test is required to see if the ray intersects the enclosed object. These volumes are then enclosed in other volumes forming a volume hierarchy (Slater et al. 2001).

Object hierarchy techniques reference each polygon exactly once (Wald et al. 2007b).

Because this structure is easy to update as an object moves, it is the most popular method for dynamic scene representation. However not the fastest in terms of traversal time.

Figure 1.1: Bounding volume hierarchy: Polygons are put in a hierarchy of boxes

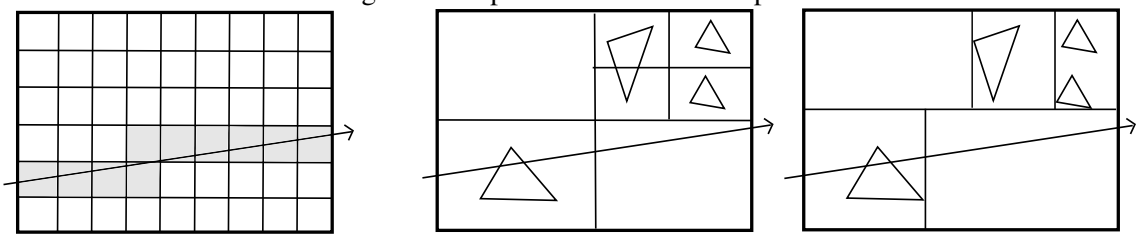


1.1.2 Space subdivision

Space subdivision techniques are a way to organize space rather than objects. These techniques divide the scene's space in cells. A polygon may lie in more than one cell. Some structures for space subdivision are uniform grids, and BSP trees. Uniform grids divide space into equal sized cells, and in each cell there is a reference to the polygons that are contained in that cell. Because some cells may contain more polygons than others, a better way is to subdivide space non-uniformly. An example of non-uniform space partitioning technique are space partitioning trees. Binary Space Partitioning trees start with the entire infinite space. Next a splitting plane is chosen to divide the space into two cells. These cells are then recursively subdivided, until all cells are left with a small number of polygons. A special case of BSP trees are kd-trees where the splitting plane is always perpendicular to the axes. This constraint greatly simplifies both the building and traversal of the tree. An other example of space partitioning are octrees who again start with the entire scene and subdivide it along the middle of each axis into 2x2x2 equal octants. Octants with a large enough number of objects are further subdivided.

In contrast to object hierarchies, space subdivision can generate cells smaller than some of the polygons inside. This allows enclosing polygons more tightly and hence fewer polygon intersections, yet more work to be performed during the building. For this reason kd-trees are the popular method for static scene representation.

Figure 1.2: Space subdivision examples



1.1.3 Construction time vs Render time

Construction time, which is the time needed to build the acceleration structure, and render time, which is the time to traverse it and do polygon intersections, are two competing processes. To build a structure that minimizes render cost requires time. To build the structure quickly, will mean sacrificing its quality. In a dynamically changing environment, both times are of the essence. For each frame the tree needs to be both built and traversed. BVHs usually have faster construction times, slower render times and can be updated, while BSP methods have slower construction times, are usually faster in rendering times, and can't be easily updated.

When building or updating a structure, some heuristic function is used to determine how the geometry will be organized into a hierarchy. The best known heuristic for kd-trees is the Surface Area Heuristic, explained in detail in 2.1

1.1.4 The kd-tree Acceleration structure

In this dissertation we will focus on the kd-tree acceleration structure. The kd-tree is a hierarchical binary subdivision of k-dimensional space into homogeneous regions, using (k-1)-dimensional axis-aligned hyperplanes. In the 3 dimensions the splitting entities are planes perpendicular to one of the coordinate axes.

Kd-trees are constructed in a top-down manner as follows:

Initially the entire scene is enclosed in a box. We then select a plane perpendicular to one of the coordinate axes and split the box in two parts. We then recursively subdivide each part. The recursion depth, and where to perform the split each time, play a vital role in the efficiency of this acceleration structure. Kd-trees are thought to be the best method for ray-tracing in static scenes. They are perform better than BSP trees because the planes are always perpendicular to the coordinate axes, which greatly simplifies traversal, and construction. The reason they are not chosen for dynamic scenes is that they can't be updated.

1.2 Motivation

As mentioned, using an acceleration structure, greatly improves rendering times but also incurs the extra cost of building such a structure. In a dynamically changing environment such as those seen in computer games, this structure needs to be rebuild for each frame, in order to take into account the changes in scene geometry.

This dissertation will examine the kd-tree acceleration structure. This structure is somewhat expensive to build, yet has superior rendering performance. Currently there exist two schools of though, one supporting that the tree should be built as good as possible to minimize render cost, while the other such as in the papers (Hunt et al. 2006), (Shevtsov et al.) supports building the

structure quickly even though the render time will be impacted by a small percentage.%
Neither of the above however considers reusing information from previews frames. In a scene exist both static and dynamic objects. Buildings and terrain are typically examples of static objects, where as people and cars of dynamic. One could build one tree for the static scene and one for each dynamic object, and then merge them to produce the overall scene tree. The static tree would not need to be rebuild again since it doesn't change from frame to frame. This means that a considerable amount of time will be spared. Currently however no algorithm exists that can merge kd-trees efficiently.

1.3 Goal

The goal of this research is to develop a quick and efficient merging algorithm for kd-trees. The algorithm will focus on merge time and build quality. This research doesn't focus on making tree traversal any faster, neither does it introduce any low level hardware optimizations. Any of the existing methods however should be compatible with our algorithm.

Chapter 2

Related Work

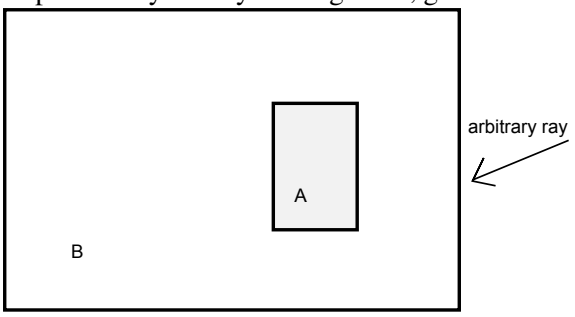
This chapter's purpose is to do a review of current ray-tracing acceleration structures used for dynamic and static scenes. Firstly we will present the surface area heuristic, considered to be the best known method for deciding on how to organize the acceleration structures for efficient ray tracing. Next we will go deeper on how the Bounding volume hierarchy - the most commonly used structure for use in dynamic scenes - is built, traversed and updated. Later on, we briefly refer to how grids, octrees and BSP trees divide space without giving much detail. Finally we will explain all about the kd-tree acceleration structure, and the methods proposed for using it in dynamic scenes.

2.1 The Surface Area Heuristic

The surface area heuristic is a function that evaluates how good a subdivision of space is. The development of this function comes from the geometric probability theory which states that if a spatial region A is entirely contained in an other spatial region B, then the probability an arbitrary ray hits region A, given that it hits region B ($P(A|B)$) is proportional to the surface area of A divided by the surface area of B.

$$P(A|B) = \frac{SA(A)}{SA(B)} \tag{1}$$

Figure 2.1: The probability the ray hits region A, given that it hits region B.



This probability helped in the development of two equations: The first evaluates the quality of the entire subdivision of space, by giving a cost which will be the cost of traversing the tree constructed for that subdivision. The second is a locally greedy approximation that can be used to determine next best possible subdivision.

The SAH assumes that rays are uniformly distributed, infinite lines. This assumption does not hold in ray-tracing because after a ray intersects an object, it stops (so it's not infinite), additionally if the ray hits a reflective object it will get reflected to some direction which means that not all rays are uniformly distributed. Even if these assumptions don't hold the SAH still gives a good estimation on what an optimal split is.

The Cost of traversing tree T is:

$$C(T) = \sum_{n \in nodes} \frac{SA(V_n)}{SA(V_S)} K_T + \sum_{l \in leaves} \frac{SA(V_l)}{SA(V_S)} K_I \tag{2}$$

- SA(V) the surface area of V
- K_T The cost of traversing one node
- K_I The cost of intersecting one polygon
- V_n, V_l, V_S The axis aligned bounding box(AABB) of internal node n, leaf l or the entire scene.

The best kd-tree T to subdivide a scene S would be the one for which equation 2 is minimal. The number of possible trees however makes it infeasible to explore all of them. Therefore a locally greedy approximation is made: The cost of subdividing cell V with plane p is computed as if both resulting children would be made leaves. This of course, is an over estimate since the children may be further subdivided, and hence their costs will be lower.

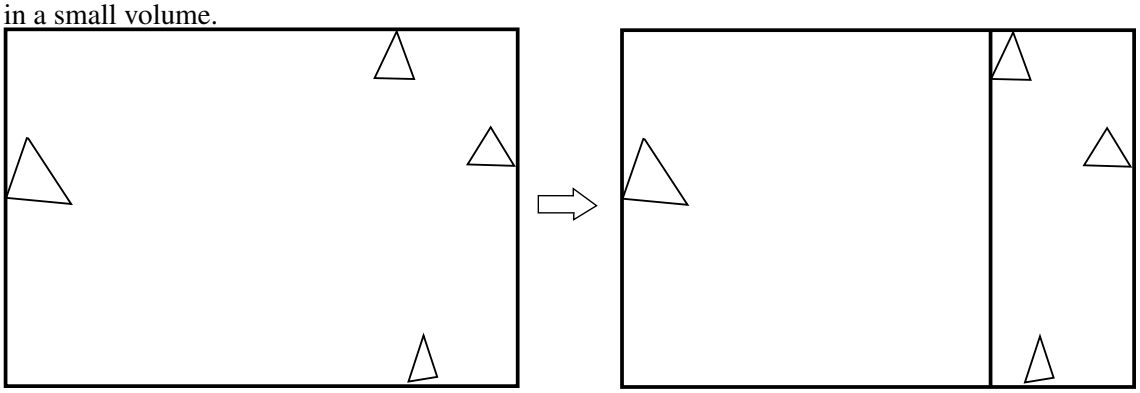
The locally greedy cost for one subdivision of V with p is:

$$C_V(p) \approx K_T + K_I\left(\frac{SA(V_L)}{SA(V)}|T_L| + \frac{SA(V_R)}{SA(V)}|T_R|\right) \tag{3}$$

- V_L, V_R The AABB of the left / right polygons.
- V The AABB of all polygons
- $|T_L|$, $|T_R|$ The number of polygons in sub-volume V_L or V_R

One evaluates this function for all possible split plane arrangements to find the one with the smallest cost $C_V(p)$. As we can deduce from the equation arrangements that have a large volume and a large number of objects on either the left or the right side have a greater cost, in comparison to arrangements where there is a small number of objects in a big volume, and a large number of objects in a small volume.

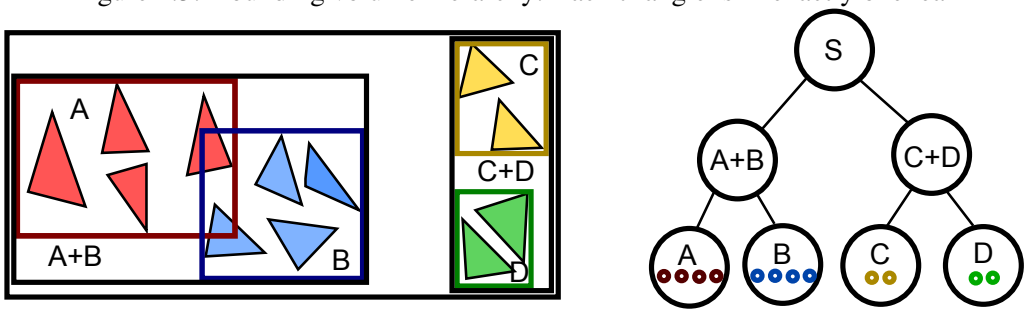
Figure 2.2: The SAH will have the lowest cost when the arrangement is a large number of objects in a small volume.



2.2 Bounding Volume Hierarchies

Perhaps the best acceleration structure for dynamic scenes is the bounding volume hierarchy. BVHs are trees that store a closed bounding volume at each node. In addition each internal node has references to an arbitrary number of child nodes, and each leaf node also stores a list of primitives (such as triangles). Each node's bounding volume is guaranteed to enclose the bounding volumes of its descendants. Each primitive is in exactly one leaf, while each spatial location may be in an arbitrary number of leaves (fig. 2.3). Several shapes can be used as bounding volumes; such examples include spheres, axis aligned boxes and oriented boxes. The selection of which bounding volume to use depends in which volume the objects would be best fitted and how easy it is to check for an intersection. Usually a tree contains only one shape for bounding volumes, to make construction and traversal simpler.

Figure 2.3: Bounding volume hierarchy: Each triangle is in exactly one leaf



2.2.1 Construction

BVHs can be build either top-down, or bottom-up. Top down construction involves dividing all the objects in groups, then each group in other groups and so on. The bottom-up approach, starts combining primitives in groups and puts them on a bounding volume, then combines groups of bounding volumes, and continues to do so until all are contained in one volume.

Following, an example of a top-down approach that uses the SAH to build a binary BVH of axis aligned bounding boxes with complexity $O(nlog^2n)$ as taken from (Wald et al. 2007a).

Algorithm 1 PARTITIONSWEEP($SetS$)

```
1:  bestCost = T_tri * |S|
2:  bestAxis = -1, bestEvent = -1
3:  for axis = 1 to 3 do
4:      Sort S using centroid of boxes in the current axis
5:      S1 = ∅,   S2 = S
6:      for i = 1 to |S| do
7:          S[i].leftArea = Area(S1)
8:          move triangle i from S2 to S1
9:      end for
10:     S1 = S,   S2 = ∅
11:     for i = 1 to |S| do
12:         S[i].rightArea = Area(S2)
13:         cost = SAH(|S1|, s[i].leftArea,|S2|, S[i].rightArea)
14:         move triangle i from S1 to S2
15:         if cost < bestCost then
16:             bestCost = cost
17:             bestEvent = i
18:             bestAxis = axis
19:         end if
20:     end for
21: end for
22: if bestAxis = -1 then
23:     return  make S leaf
24: else
25:     sort S in axis bestAxis
26:     S1 = S[0..bestEvent)
27:     S2 = S[bestEvent..|S|)
28:     return  make inner node with axis bestAxis
29: end if
```

What this algorithm does, is the following: Initially the bestCost is initialized to be the cost if S were a leaf. Next it sort's the triangles of S in ascending order based on the center of each triangle. Next there is a loop that sweeps the triangles of S, shifting them from subset S2 to subset S1 and calculating surface area of the box containing the triangles of S1. Following there is a loop that sweeps triangles of s from right to left, calculating the surface area of the box containing the triangles of S2, and evaluates the SAH using an equation similar to (Eq.3).

2.2.2 Traversal

The BVH has a very simple traversal algorithm. For leaves, the ray is tested against the bounding volume and if there is an intersection, the list of triangles is tested. For internal nodes, if the ray intersects the bounding volume, then the children are recursively tested for intersection. Because the children are not spatially ordered, all children that intersect the ray must be checked, even if we already found a triangle intersection. Finally the algorithm must return the intersected triangle, which is closest to the ray origin.

Algorithm 2 BVHTRAVERSE(Node n, Ray r)

```
1:  if n.isLeaf() then
2:      return closestIntersectionIn(n.triagnles)
3:  else
4:      for i = 1 to n.children do
5:          if r overlaps n.child[i] then
6:              t = t ∪ BVHTraverse(n.child[i], r)
7:          end if
8:      end for
9:      return closest t
10: end if
```

2.2.3 Packet Traversal

Faster methods, instead of testing one ray at a time, use the above algorithm with a ray packet. A ray packet is a group of coherent near by rays. It's worth mentioning some of the key strategies that (Wald et al. 2007a) use in order to demonstrate what makes packet traversal much quicker that testing single rays.

Early hit test. Consider a group of coherent rays. If a ray intersects a big volume then rays close to that one are also likely to intersect it. This means that one can conservatively skip some intersection checks by considering the entire packet as 'hit'. If any ray of the packet intersects a node then immediately enter the subtree without considering any of the remainder rays.

Early miss exit. The early hit test can accelerate traversal in the case where we find a hit in the first few rays. In the common case where all rays miss however it has a linear (to the number of rays) cost. To avoid this, one could check the bounding frustum of the ray packet against the bounding volume. If the frustum misses then we can return a miss without further tests.

First active ray tracking. Let's say that the first few rays missed the bounding box, and the 4th ray hit. This as mentioned in the early hit test means that the entire packet will enter the subtree. Since the first 3 missed the subtree's root node, then there is no chance that they will hit any of it's children. To avoid this, we start testing for the first active ray (we consider the rays that missed inactive).

Leaf traversal. Some rays reach leaf nodes in order to speed up traversal even if they don't intersect a parent bounding volume. As a last step each active ray is tested with the bounding volume of the leaf.

2.2.4 Update

The BVH is a hierarchy of objects which makes it easy to update, since the bounding volumes can move along with any moving objects, something not possible with a space hierarchy. The procedure of updating involves recursively traversing the hierarchy recompute the child node's bounding volumes and refit the parent's boxes. Below is an algorithm for updating a deformable scene. A deformable scene is one whose triangles move, but no triangles are split created or destroyed over time.

Algorithm 3 UPDATEBBOXES(Node n)

```
1: if n.isLeaf() then
2:     bounds(n) ← boundsOf (triangles(node))
3: else
4:     UPDATEBBOXES(n.child[0])
5:     UPDATEBBOXES(n.child[1])
6:     n.bounds = bounds(n.child[0]) ∪ bounds(n.child[1])
7: end if
```

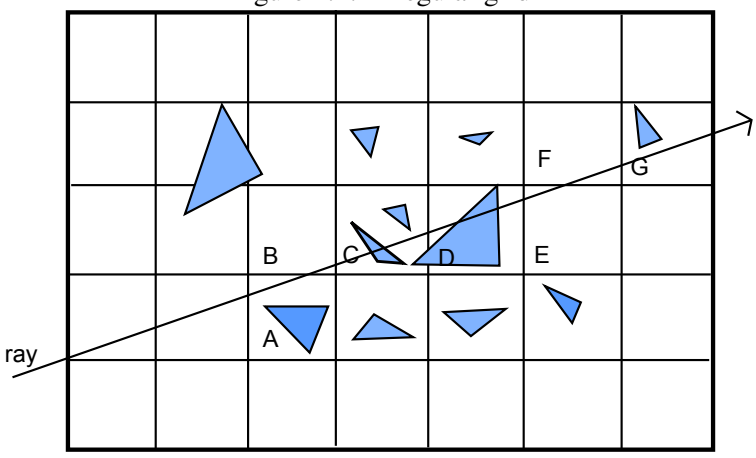
2.3 Space Subdivision

In section 2.2 we've talked about bounding volume hierarchies. Space subdivision is an other way of organizing scene geometry. In space subdivision techniques, space is organized in non-overlapping cells and each cell has a reference to the polygons that lie in it. In contrast to the bounding volume hierarchy each spatial location belongs to exactly one cell and a polygon may lie in more than one leaves. An apparent benefit of a spatial subdivision over an object subdivision, is that by traversing in the correct order, once a hit is found the recursion can return immediately without checking the rest of the intersected cells. In this section we will present grids, octrees, and BSP trees which are ways for spatial subdivision, without going in much detail. In the next section we will go in greater detail about another method for space subdivision the kd-trees which are a special case of BSP trees.

2.3.1 Grids

The idea of regular grids, is to bound the scene with an axis aligned bounding box, and divide the space into a uniform partition. Each cuboid cell stores a reference to the polygons that intersect it (fig. 2.4).

Figure 2.4: A regular grid



As with all spatial subdivision methods a polygon may lie in more than one cell, for example the polygon of cell D, is also referenced by cell C. The fact that the grid is regular makes construction and traversal a very simple and quick process. This structure can be represented as a 3d array of polygon lists. To construct it, one could interpolate between the vertices of each polygon and determine in which cells the polygon lies. Then he can insert a reference to the polygon at the proper array indices. A regular grid can be traversed by interpolating between the origin and direction of the ray with a method called 3DDA (digital differential analyzer).
Something worth noting is that in the example of (fig. 2.4) there are two intersections between the ray and the triangles of cell C. However one of the intersections happens outside of cell C so it is discarded. If there are more than one intersections in the same cell, then the one closest to the ray origin is returned. Grids are easy to be constructed and updated but perform badly in scenes with varied geometric density.

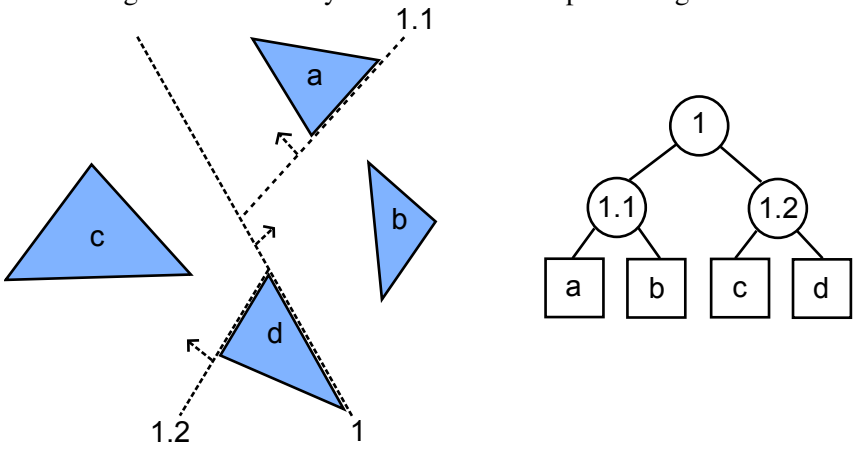
2.3.2 Octrees

Octrees are an adaptive method for space subdivision. Instead of uniformly dividing space like grids, octrees subdivide space at the places where there is a high concentration of primitives. This not only reduces the number of empty cells, but also reduces the number ray-triangle intersections, since the cells of a densely populated area are smaller and more likely to be missed. To construct such a structure we start by having the entire scene in an axis aligned bounding box. If the number of objects is big enough, we subdivide the box in the middle of each axis to produce 8 cells called octants. Next we recursively check each octant to see if it requires further subdivision. In order to traverse an octree we clip the ray in to the scene's bounding box and traverse the tree in order to find the first cell (leaf) it intersects. Then by checking the side of the cell the ray exits, we can discover the next cell.

2.3.3 BSP Trees

BSP trees binary subdivide space using arbitrary planes. Each internal node represents a region of space and holds a plane that divides that space into a positive and negative half space. If the equation of the plane is $f_n = ax + by + cz + d$ then the set of points, within the node's region of space, that have $f_n > 0$ are in the positive half space. A leaf node doesn't hold a plane but corresponds to an unpartitioned region of space, which we call a cell.

Figure 2.5: A binary subdivision of 2d space using lines

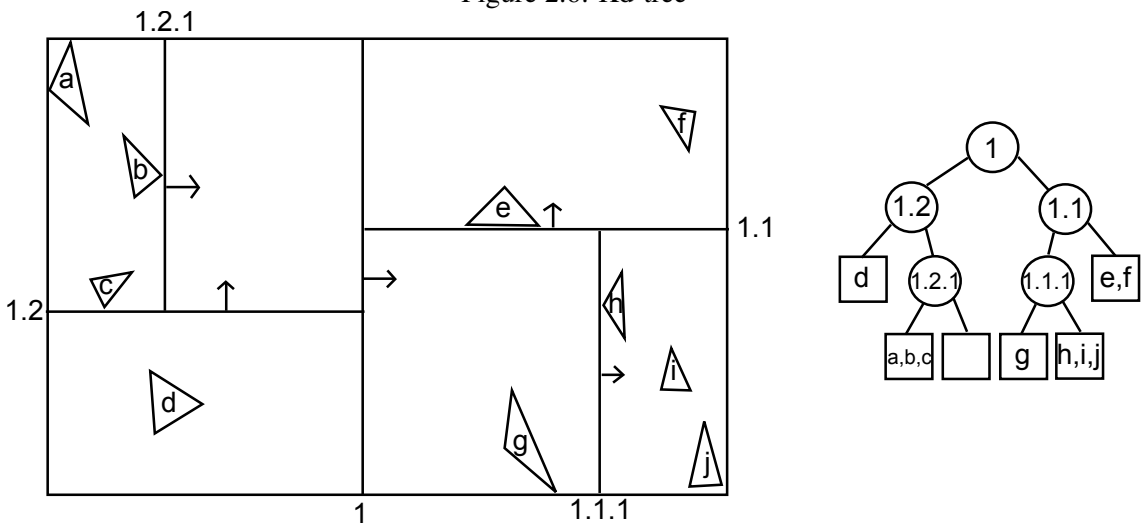


In order to build the BSP tree we select a polygon of the scene and use the plane defined by that polygon as the root of the tree and use it to partition space into the positive and negative subspaces. Each subspace is recursively partitioned using a polygon in that subspace. Any polygon lying partly in both subspaces is split along the intersection with the plane, and the two fragments are placed in the corresponding sets. Polygons coplanar with the partitioning plane of a node are stored at that node. Because of the complexity of this structure, and the difficulty to be constructed and traversed efficiently, a simplified version of it called kd-trees are used in ray tracing instead.

2.4 Kd-Trees

Considered to be the best acceleration structure for ray-tracing static scenes, the kd-trees which are a special case of BSP trees subdivide space only using planes perpendicular to the axes. Similarly to BSP trees, each node in the kd-tree represents a region of space, and contains a plane that divides that space in to two half spaces. Leaves instead of having a plane, have a list of polygons that intersect their space.

Figure 2.6: Kd-tree



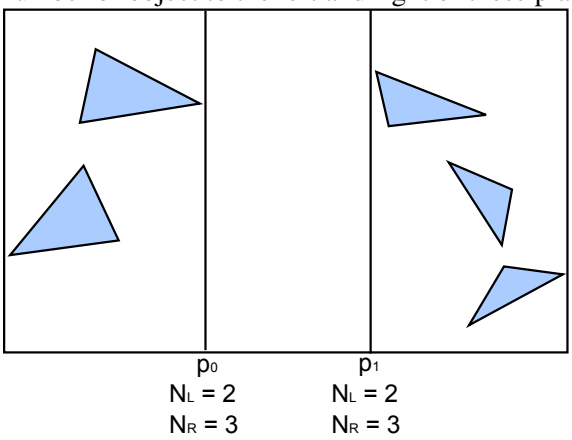
2.4.1 Optimal, KD-Tree construction

Every data structure can be build with a lot of different possible ways. Commonly there is a trade-of between build time and build quality. The algorithm for building an optimal kd-tree acceleration structure in the lowest theoretical bound of O(nlogn) is explained in the next section. Sub-optimal methods that improve build time at the cost of build quality (which affects render time) also exist and are described in 2.4.2

2.4.1.1 Using SAH to find the best splitting plane in O(nlogn)

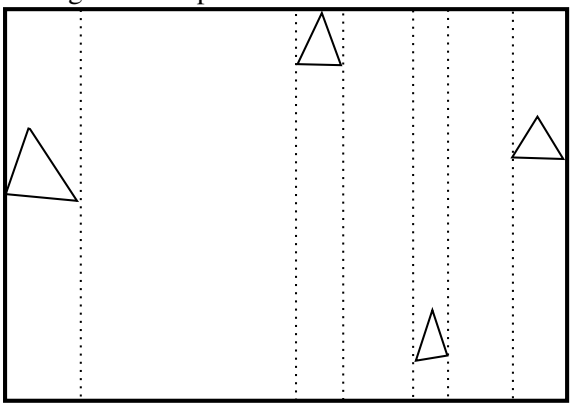
In order to build the kd-tree top-down we must decide on where to position the splitting plane of each subspace. As mentioned in section 2.1, there is a a function to estimate the cost of a candidate split plane p. It is time to present the algorithm that given a list of polygons and their axis aligned bounding box, will in O(nlogn) time, return the split plane with the lowest cost.
One may initially wonder how this is possible since the number of possible positions to place the plane are infinite. However, if we take a closer look at the function, we can see that if the numbers N_L, N_R don't change for two planes p_0, p_1, then the cost function depends solely on the surface area of V_L and V_R, which is a linear function in respect to the position of the splitting plane p.

Figure 2.7: The number of object to the left and right of these planes are the same



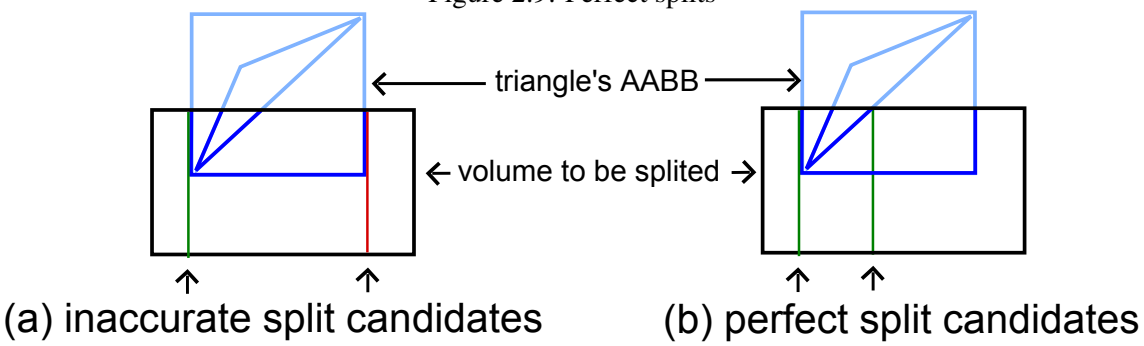
This linearity means that the minimum C(p) value will be found at either the plane p_0 or p_1, depending if the linear function is increasing or decreasing. This observation means that C(p) can have its minima only at those finitely many planes where N_L and N_R change. These planes will be hence forth referred to as "split candidates".

Figure 2.8: Split candidates for the x-axis



An intuitive choice for split candidates is the 6 planes defining the polygon's AABB, after all it seems to be there the number objects changes. It's worth noting that may not be the case after the first split, where instead of considering the original bounding box of the object, we should consider the bounding box of the part of the triangle inside the volume we are trying to split. These are called perfect split candidates.

Figure 2.9: Perfect splits



Following a description of how we can determine the best plane to use, in order to partition a cell V. The algorithm described refers only to finding the best split plane on the x-axis. This process should be repeated for the y and z axes. To efficiently evaluate the costs on each candidate we can use a plane sweep algorithm. The algorithm can be visualized as a plane sweeping our cell from left to right, stopping at the split candidates, updating the number of triangles to the left and right side of the plane (N_L and N_R) and evaluating the cost. In the x-axis example these split

candidates are the left and right bounding planes of each polygon. In order to do this sweep, for each polygon, we generate two events: A ”start event” refers to the left bounding plane, and an ”end event” refers to the right bounding plane. If the polygon is planar (meaning that the left and right bounding planes are at the same position) we generate only one event called ”planar event”. Each event has a pointer to the polygon that the event belongs to, and a position which is the x-coordinate of the bounding plane. E=(position, type, polygon).

Once all events are generated, we sort the event list by ascending position. Events with equal plane positions are sorted by their types in the order of End $<$ Planar $<$ Start.

We symbolize N_l as the polygons purely to the left of plane p, N_r as the polygons purely to the right, and N_{on} as the polygons intersected by the volume. We symbolize c_s, c_e, c_p the number of start, end, and planar events at the current sweep plane position.

Initially N_l = 0, N_r = number of polygons, N_{on} = 0. and as the sweep plane moves, we update these numbers (lines 17, 28). Planar polygons may be positioned in either side of the volume, so we are free to place them on the side which makes the cost smaller. Polygons on the splitting plane should be placed in both the left and right sub-volume.

Algorithm 4 PLANESWEEP(T, V)

1: $sort(E, x_{axis})$
2: **for** (e = E_{head}; e;) **do**
3: $c_e = c_p = c_s = 0$
4: sweepPos = $e_{position}$
5: **while** e **and** $e_{position}$=sweepPos **and** e_{type}='end' **do**
6: inc c_e
7: e = e→next
8: **end while**
9: **while** e **and** $e_{position}$=sweepPos **and** e_{type}='planar' **do**
10: inc c_p
11: e = e→next
12: **end while**
13: **while** e **and** $e_{position}$=sweepPos **and** e_{type}='start' **do**
14: inc c_s
15: e = e→next
16: **end while**
17: N_l += c_e, N_o -= c_e, N_r -= c_p
18: $CostL = (N_l + N_{on} + N_p) * (sweepPos - V_{min}) + (N_r + N_{on}) * (V_{max} - sweepPos)$
19: $CostR = (N_l + N_{on}) * (sweepPos - V_{min}) + (N_r + N_{on} + N_p) * (V_{max} - sweepPos)$
20: **if** $CostL \leq CostR$ **then**
21: $Cost = CostL, pside =' left'$
22: **else**
23: $Cost = CostR, pside =' right'$
24: **end if**
25: **if** $Cost < min_{Cost}$ **then**
26: $min = (cost, sweepPos, pside)$
27: **end if**
28: N_l += c_p, N_o += c_s, N_r -= c_s
29: **end for**
30: **return** $min_{sweepPos}, min_{pside}$

2.4.1.2 Using SAH to find the best splitting plane in O(n)

In the previews algorithm the complexity O(nlogn) was due to the sorting in the first line. If the events E were presorted, the algorithm's complexity would drop to O(n). In order to do this, we must maintain 3 sorted event lists - one for each axis-. These lists will need to be spliced, after the volume is partitioned into the left and right sub-volumes. Two lists per axis will be produced from the splicing, one for the polygons to the left of the splitting plane and one for the polygons to the right.

The splicing of the lists could be done by tagging the polygons as ”left”, ”right” or ”on” the splitting plane, then pass over each event list once splicing it to the 2 lists. Objects tagged as ”on” should go in a temporary list. The reason for the temporary list, is that their $event_{possition}$ needs to be updated so we have perfect split candidates (fig. 2.9). After updating with the new event positions, this list needs to be sorted and merged with both the left and right event lists. Assuming that only ($\sqrt{n}$ triangles overlap p, the complexity remains O(n)).

2.4.1.3 Building the tree in O(nlogn)

Now that we can find the best plane to subdivide each space, it’s time to recursively subdivide the scene. Each recursion step involves finding the best plane to divide the scene, updating the event lists, and repeating the process for the left and right half-spaces.

Algorithm 5 BUILDKDTREE($Polygons\ p, Events\ e, Volume\ v$)

```
1:  (cost, plane, planarside) = PARTITIONSWEEP(e,v)
2:  if cost < costifleaf(p,v) then
3:      TagPolygons(p, plane)
4:      (e_L, e_R) = SpliceEventsBasedOnPolygonTag(e)
5:      (p_L, p_R) = SplicePolygons(p, plane)
6:      (v_L, v_R) = Split(v, plane)
7:      newNode.plane = plane
8:      newNode.left = BUILDKDTREE(p_L, e_L, v_L)
9:      newNode.right = BUILDKDTREE(p_R, e_R, v_R)
10:     return newNode
11: else
12:     return makeLeaf(p,v)
13: end if
```

Complexity: The algorithm each recursion approximately halves the number of triangles on each side. $T(N) = N + 2T(\frac{N}{2}) = NlogN$

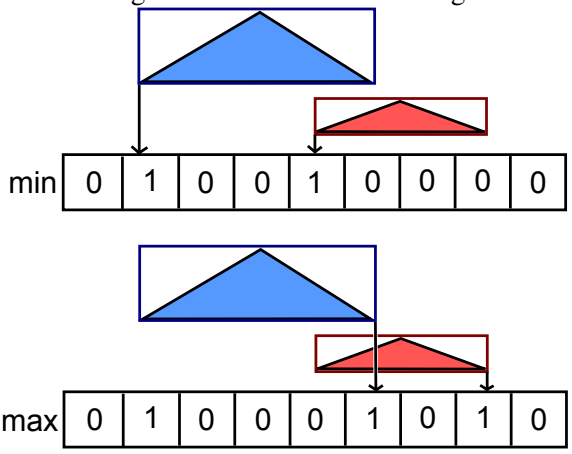
2.4.2 Suboptimal fast KD-Tree construction

Although the algorithm described in 2.4.1.3 is the theoritical lower bound for constructing a kd-tree using the surface area heuristic, some researchers proposed lowering the constants of the asymptotic analysis. The suggestions made were that the events could be grouped in bins, or sub-sampled. We will briefly describe two of the proposals.

2.4.2.1 Min-Max Binning

In the paper (Shevtsov et al.) the authors propose a binning algorithm to evaluate SAH cost quickly. We describe the method they use to sweep the x axis. The algorithm uses two arrays of bins Min[0..31] and Max[0..31] as counters. For each polygon's AABB they update exactly one bin in the min array (where the AABB begins), and one bin in the max array(where the AABB ends). They also adaptively skip some primitives at the higher kd-tree levels, processing only each l-th, primitive in the current node, where $l = log_{10}(N)$.

Figure 2.10: Min-Max binning



A simple pass over the bins evaluates SAH cost. Another pass over the polygons splices them in the two lists, determines the tight boundaries of geometry in the two children and adjust the

final splitting plane position to one of those boundaries. At the lower tree levels, the algorithm switches to exact evaluation.

2.4.2.2 Adaptive sampling

An adaptive sampling method for determining a split plane given a set of polygons and their bounding volume is given in (Hunt et al. 2006). The algorithm proposed in the paper first samples 8 planes per axis distributed uniformly. Next 8 adaptive samples per axis are taken based on the information obtained by the first samples. The sampling produces a piecewise quadratic function. For each of the 15 pieces they calculate the local minima, and place the split plane at the overall minimum. At the lower tree levels, this algorithm also switches to exact evaluation. For the sample scenes tested by this algorithm the render time increased by a mere 3.6

2.4.3 Kd-tree Traversal

Every kd-tree regardless of the method we use to construct it has one method for traversing it. Initially we clip the ray to the scene's bounding volume. Next test if the ray intersects any of the bounding volumes of the child nodes. If it does, then we clip the ray to the child's bounding volume and continue recursively to the first child intersected. The algorithm on the next page is the implementation of this algorithm with a stack to avoid recursion.

2.4.4 Kd-trees For dynamic scenes

In dynamic scenes the trees are rebuild each frame, using a method for suboptimal construction as those mentioned in (sec 2.4.2). The trees are traversed using ray packets similarly to the way BVHs do. Optionally a fuzzy kd-tree can be build that takes into account changes in the scene over a period of time.

Algorithm 6 TRAVERSEKDTREE($Ray\ ray$)

```
1:  (tMin, tMax) = Clip ray to the scene's BB
2:  node = root
3:  while node ≠ NULL do
4:      if node.isLeaf() then
5:          (success, polygon) = checkForIntersection(node.polygons, ray, tMin, tMax)
6:          if success then
7:              return polygon
8:          end if
9:          (node, tMin, tMax) = popNode()
10:         if node = NULL then
11:             break
12:         end if
13:     else
14:         leftFirst = (ray.origin[node.plane.axis] < node.plane.position) or
            (ray.origin[node.axis] = node.plane.position and ray.direction[node.plane.axis]<0)
15:         if leftFirst then
16:             firstChild = node.leftChild
17:             secondChild = node.rightChild
18:         else
19:             firstChild = node.rightChild
20:             secondChild = node.leftChild
21:         end if
22:         tPlane = (node.plane.position - ray.origin[node.plane.axis])/ray.direction[node.plane.axis]
23:         if tPlane > tMax or tplane ≤ 0 then
24:             node = firstChild (Check only 1^st child since either tMax doesn't go further
                than the splitting plane or it's facing away from it)
25:         else if tPlane < tMin then
26:             node = secondChild (Check only 2^nd child. The ray doesn't overlap the 1^st)
27:         else
28:             push(secondchild, tPlane, tMax)
29:             tMax = tPlane
30:             node = firstChild
31:         end if
32:     end if
33: end while
34: return NULL
```

Chapter 3

Algorithm

3.1 Overview

In this chapter we will discuss the methodology to merge the trees for the static and dynamic scenes. Using the method of (Wald and Havran 2006) to build trees. We will build one tree for the static scene, and one for each dynamic object. In order to do the merging we will need three auxiliary algorithms. The first algorithm given a tree and a plane that intersects the tree's volume, returns an estimate of the number of polygons to the left, the right and on that plane. The second algorithm is a modified version of BSP tree merging adapted for kd-trees. The third algorithm will be a modified version of (Wald and Havran 2006) that will find the best splitting plane from a list of trees (instead of a list of polygons).

All these algorithms will be the tools used by the kd-tree algorithm proposed.

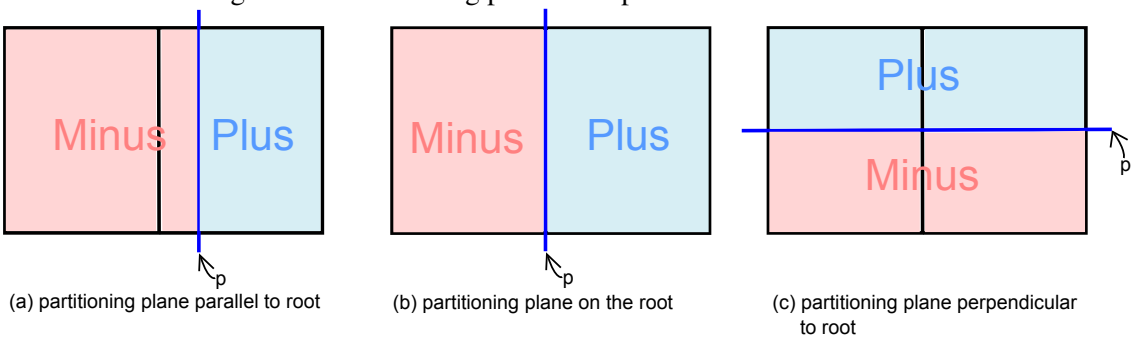
3.2 Partitioning a Kd-tree with a plane

In this section we will discuss how we can partition a tree T with a plane, to produce two trees T^+ and T^-. We name T^- the tree to the left of the partitioning plain, and T^+ the tree to the right. Here we must differentiate between leaf nodes and internal nodes. If the tree is infact a leaf node then the algorithm splits the polygons of the leaf and returns two leaves: Polygons entirely left of the splitting plane are inserted to the T^- leaf, those entirely to the right are inserted to the T^+ leaf. Finally, polygons that are intersected by the splitting plane are inserted to both leaves. If the tree is not a leaf then three steps are performed:

1. The partitioning plane p is compared against the plane of the root node
2. The subtrees, T_L or T_R, in which p lies are partitioned reccursively
3. The resulting subtrees from the above partition are combined to form T^+ and T^-

The partitioning plane p can be in three possible possitions with respect to the plane of the root node as shown in Figure 3.1.

Figure 3.1: Partitioning plane in respect to the root of the tree



In the case where the partitioning plane and the root's plane are parallel$^{(a)}$ then only the subtree where p lies needs to be partitioned. In the case that the p is exactly on the root's plane$^{(b)}$ no subtree is partitioned. Finally in the case where the p is perpendicular to the root's plane$^{(c)}$ both

subtrees need to be partitioned.

The last step is to put the pieces from the partitoning of T_L and T_R together to form T^+ and T^-.

(a) The partitioning plane lies on the right subtree.

$\{T_R^-, T_R^+\}$ = partitionTreeWithPlane(T_R, p)

T^-.left = T_L

T^-.right = T_R^-

T^+ = T_R^+

the other case is analogous.

(b) The partitioning plane lies on the root's plane.

T^- = T_L

T^+ = T_R

(c) The partitioning plane is perpendicular to the root's plane

$\{T_L^-, T_L^+\}$ = partitionTreeWithPlane(T_L, p)

$\{T_R^-, T_R^+\}$ = partitionTreeWithPlane(T_R, p)

T^-.left = T_L^-

T^-.right = T_R^-

T^+.left = T_L^+

T^+.right = T_R^+

As said in the beginning this algorithm will partition a tree with a given plane. This will be a useful tool of the algorithm discussed in the following chapter.

Algorithm 7 PARTITIONTREEWITHPLANE($Tree\ t, Plane\ p$)

```
1:  if t.isLeaf() then
2:      (minusPolys, plusPolys) = partitionObjects(t.polygonList, p)
3:      minus = makeLeafNode(minusPolys)
4:      plus = makeLeafNode(plusPolys)
5:      return (minus, plus)
6:  else
7:      if p.axis ≠ t.plane.axis then
8:          (tlm, tlp) = PARTITIONTREEWITHPLANE(t.leftChild, plane)
9:          (trm, trp) = PARTITIONTREEWITHPLANE(t.rightChild, plane)
10:         minus.leftChild = tlm
11:         minus.rightChild = trm
12:         plus.leftChild = tlp
13:         plus.rightChild = trp
14:     else
15:         if plane.position < tree.plane.position then
16:             (tlm, tlp) = PARTITIONTREEWITHPLANE(t.leftChild, plane)
17:             minus = tlm
18:             plus.leftChild = tlp
19:             plus.rightChild = t.rightChild
20:         else if plane.position > tree.plane.position then
21:             (trm, trp) = PARTITIONTREEWITHPLANE(t.rightChild, plane)
22:             plus = trp
23:             minus.leftChild =tree.leftChild
24:             minus.rightChild = trm
25:         else
26:             minus = tree.leftChild
27:             plus = tree.rightChild
28:         end if
29:     end if
30: end if
31: return (minus, plus)
```

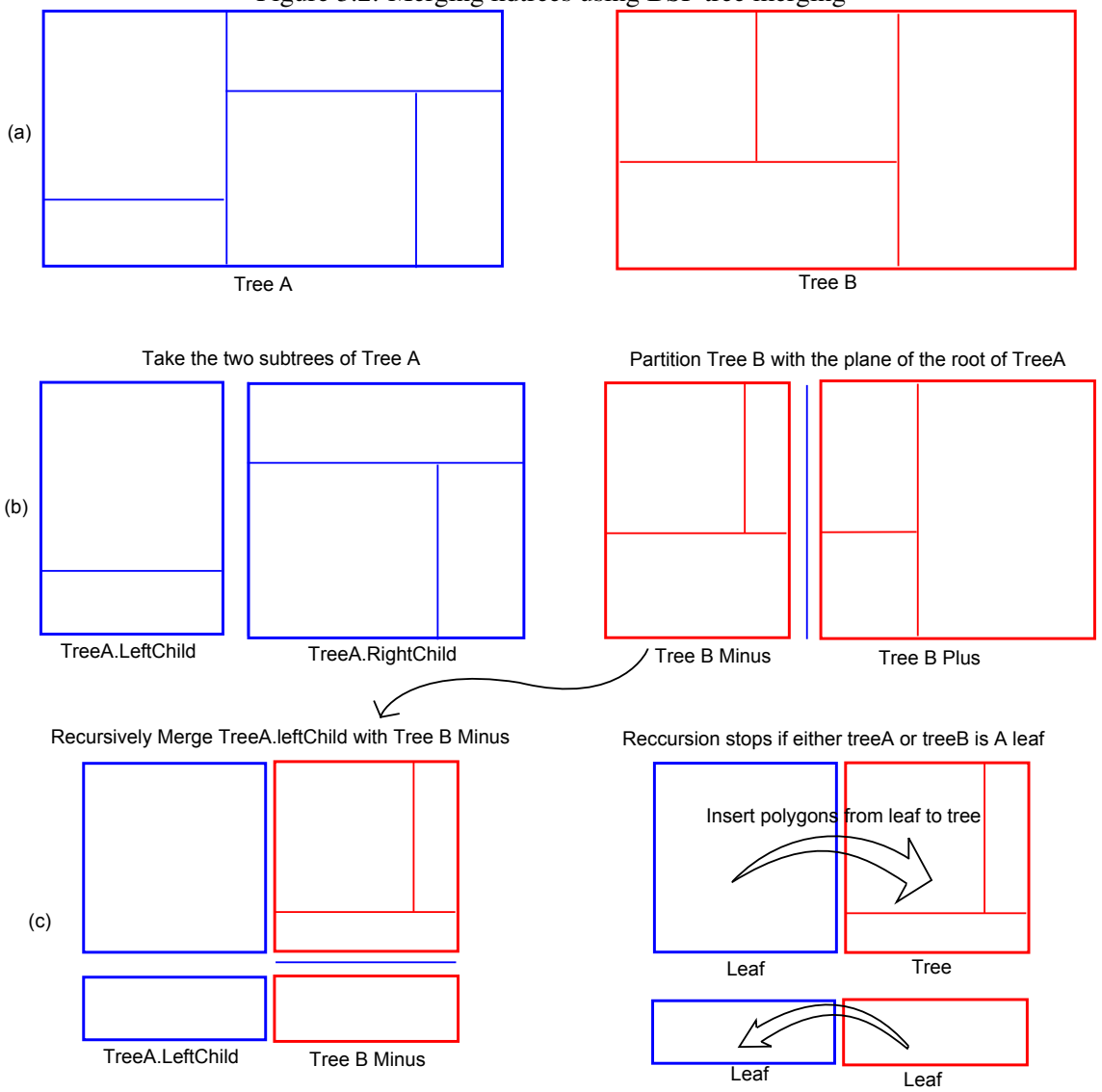
3.3 Kd-Tree Merging

A method we explored for merging kd-Trees is the already existent BSP merging algorithm. The only difference is that it uses the function partitionTreeWithPlane for kd-trees instead of the one for BSP trees. This algorithm given two trees treeA and treeB will merge them into one. During the merge the original trees are destroyed.

Firstly we use the plane of the root node of tree A to partition tree B. Next we recursively call this

function two times. First to merge the left child of tree A and the minus part of tree B and one more to merge the right child of tree A with the plus part of tree B. Recursion ends when either tree A or tree B is a leaf. In such case we insert the polygons of the leaf to the tree. In the case where both are leaves we can select either leaf and insert it's polygons to the other leaf and return that leaf.

Figure 3.2: Merging kdtrees using BSP tree merging

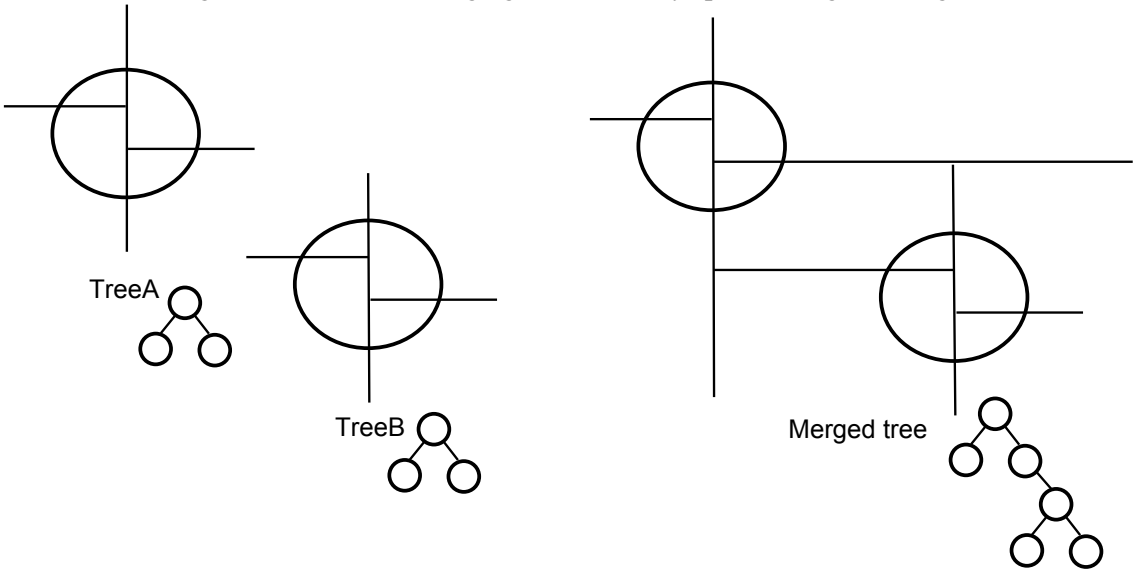


Algorithm 8 MERGEBSPTREES($Tree\ treeA, Tree\ treeB$)

```
1:  if treeB.isLeaf() then
2:      insertObjectsInTree(treeA, treeB.polygons)
3:  else if treeA.isLeaf() then
4:      insertObjectsInTree(treeB, treeA.polygons)
5:      return treeB
6:  else
7:      (minus, plus) = PARTITIONTREEWITHPLANE(treeB, treeA.plane)
8:      treeA.left = MERGEBSPTREES(treeA.left, minus)
9:      treeA.right = MERGEBSPTREES(treeA.right, plus)
10: end if
11: return treeA
```

Although this algorithm produces a correctly merged tree, it doesn't produce a good quality merge. Consider two entirely separate trees. After being merged tree B may land under a leaf of tree A (fig. 3.3).

Figure 3.3: BSP tree merging doesn't always produce a good merge



3.4 Partition Trees

In subsection 2.4.1.3, we have shown how one could construct a tree from a list of polygons. This was done by sweeping over the polygons, evaluating the surface area heuristic at the bounding planes of each polygon, partitioning space and splitting the polygons to the positive and negative half spaces at the plane position with the lowest cost. The algorithm we propose for doing the merging works in the same manner. In our case, instead of having a list of polygons we have a list of trees. We sweep over the list of trees to find the splitting plane with the lowest cost. Next we insert the splitting plane and separate the trees entirely to the left or entirely to the right of the splitting plane. Trees that lie on the splitting plane get partitioned using the algorithm ”partition tree with plane” as described in 3.2. In contrast to the original algorithm, recursion will stop when there is exactly one tree in each leaf node, and the leaf’s volume is tight on the volume of the tree. The algorithm described above will work well with trees that are separate. In the case where two trees intersect, the split candidates alone not be the best choice for the splitting plane. To increase the resolution of the find best plane method, we use the dissolve function which effectively removes the root node of each tree making doubling the amount of trees, and reducing their size to approximately half of the original, thus doubling the amount of candidate split planes.

Algorithm 9 PARTITIONTREES(treeList,events[3],v,comingFromDissolve)

Require: A list of trees in treeList.
Require: The sorted EPS events of each tree in each of the 3 axes.
Require: A Volume v that encloses all trees in treeList.
Require: Information about last step in comingFromDissolve.

```
1: [cutInfo,cleanSplit] = FINDBESTPLANE(treeList,events,v)
2: if ||treeList|| > 1 or not Tight(treeList,v) then
3:     if not cleanSplit and not comingFromDissolve then
4:         [treeList,events] = DISSOLVE(treeList,events,v)
5:         if allAreLeafs(treeList) then
6:             return MAKETREEFROMLEAFS(treeList,v)
7:         else
8:             return PARTITIONTREES(treeList,events,v,true)
9:         end if
10:    end if
11:    for each tree in treeList do
12:        if tree is left from cutInfo.plane then
13:            leftTrees.add(tree)
14:            leftEvents.add(tree.events)
15:        else if tree is right from cutInfo.plane then
16:            rightTrees.add(tree)
17:            rightEvents.add(tree.events)
18:        else
19:            [leftTree,rightTree] = PartitionTreeWithPlane(tree,cutInfo.plane)
20:            leftTrees.add(leftTree)
21:            leftEvents.add(leftTree.events)
22:            rightTrees.add(rightTree)
23:            rightEvents.add(rightTree.events)
24:        end if
25:    end for
26:    node = new TreeNode
27:    node→left = PartitionTrees(leftTrees,leftEvents,leftV,false)
28:    node→right = PartitionTrees(rightTrees,rightEvents,rightV,false)
29:    node→plane = cutInfo.plane
30:    return node
31: else
32:    return MAKETREEFROMLEAFS(treeList, v)
33: end if
```

3.5 Implementation Details

In order to test our algorithms we built upon the ray-tracer Miro created by the University of San Diego. Miro supports direct illumination, specular reflections and hard shadows. Miro was an ideal ray-tracer to use for our experiments, since it had as little features as possible making the code clean an easy to be extended. Unfortunately Miro lacked some features such as animation and textures.

The first step was to implement the kd-tree acceleration structure. We implemented an algorithm similar to (Wald and Havran 2006) for the build, as described in (alg. 5). After that we implemented a single ray traversal algorithm, the same one used by the popular PBRT ray-tracer (alg. 6).

Following that we started experimenting with various methods for kd tree merging.

Initially we implemented the BSP merge algorithm, trying to improve merge speed with optimizations. After various experiments we found that the algorithm didn't produce a good quality tree.

An other method we explored was inserting the polygons of the dynamic tree in the static tree. When all the polygons were inserted they would land in the static tree's leafs. Some of the leafs would have an increased number of polygons and so we checked if further subdivision was necessary. Eventually this method proved to give big merge times so it was abandoned.

Finally we devised the algorithm PartitionTrees that we described in (sect. 3.4). This algorithm provided us with very fast merge times, and excellent results in the case where the trees were separate. We made numerous experiments to improve the quality of the tree in the case where the two trees overlap, either entirely or partially and finally came up with the dissolve method.

To produce more realistic renderings we added textures to the ray tracer, beautifying the produced images.

In order to test our algorithms we initially used scenes referenced by various papers such as the bunny and the dragon of Stanford University's repository. Although those scenes were good for the initial merge experiments, they were static scenes. In order to really see how the merge process worked in dynamic scenes, we used the BART scene parser. BART stands for benchmark for animated ray-tracing, and it was especially designed to stress some of the most common problems of updating acceleration structures. The BART parser required some modifications in order to be integrated to Miro. After making those modifications we started measuring times and producing results.

Chapter 4

Results

The results of our algorithm are shown below. As we can see, the merge time is very small, the render speed of is only reduced by 2%, and we gain on average 640ms, which a is 6% speedup on the total build and render time.

4.1 Evaluation Criteria

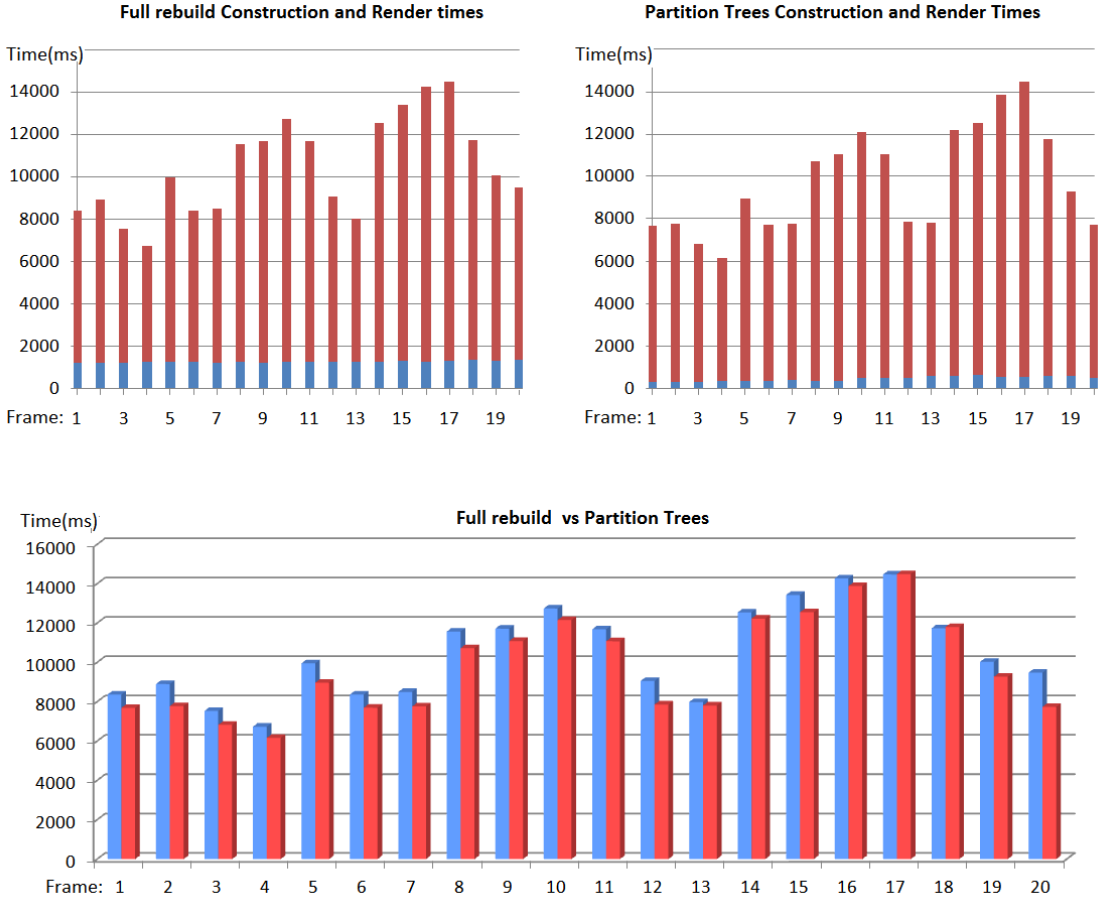
Measurements were taken from BART kitchen scene at 800x600 resolution with 1 light source, 2 reflection bounces and shadows. The reason we chose this criteria is that interactive scenes usually don't have many light sources, or reflection bounces to maintain low render times.

4.2 Measurements

Table 4.1: Build Times and Render times for the selected scenes (in milliseconds)

	Partition Trees				Full Rebuild	
Scene Name	Copy	Dynamic	Merge	Render	Build	Render
BART Kitchen Frame 2	32	80	178	7394	1198	7165
BART Kitchen Frame 4	32	78	185	6529	1174	6357
BART Kitchen Frame 6	32	110	201	8624	1234	8716
BART Kitchen Frame 8	33	110	219	7400	1214	7285
BART Kitchen Frame 10	36	104	200	10747	1213	10503
BART Kitchen Frame 12	37	114	333	10591	1216	10467
BART Kitchen Frame 14	42	109	436	7220	1219	6767
BART Kitchen Frame 16	48	111	473	11918	1288	12143
BART Kitchen Frame 18	47	108	378	13958	1286	13184
BART Kitchen Frame 20	52	112	420	8687	1287	8743

Figure 4.1: Comparison of Full rebuild vs Partition Trees

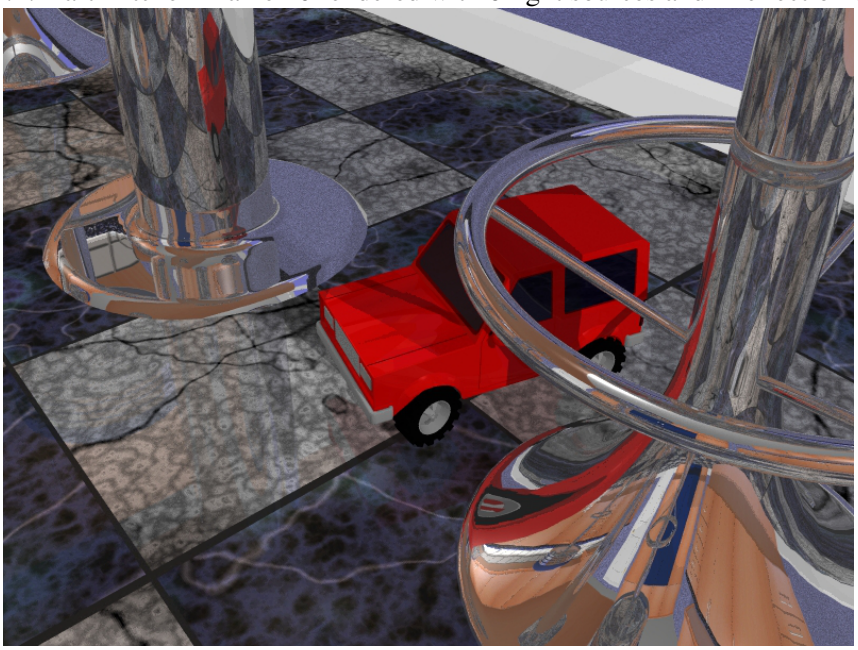


In the first two charts we show the times for structure construction (blue) and traversal (red) for the two methods. We can observe that render times are 10 times slower than construction times. The reason for this is that we didn't focus on traversing the tree quickly using packet tracing or GPU hardware. If that were the case, construction and render times would be approximately the same. This would mean that instead of affecting only the 10% of the total time, our solution would affect the 50% which is in actuality a 30% overall speedup.

The second chart is a side by side comparison of the total (build + render) times for full rebuild (blue) and partition trees (red). In 99% of cases our algorithm produces faster overall times.

The times were measured taking frames from BART Kitchen scene at one second intervals at resolution 800x600 pixels, with hard shadows (1 light source) and reflections (2 bounces).

Figure 4.2: Bart Kitchen Frame 10 rendered with 6 light sources and 2 reflection bounces



Chapter 5

Conclusion And Future Work

5.1 Conclusion

After weighting the results of merge+render times of the merge method versus the build+render times of the optimal method, we have come to the conclusion that merging trees is a viable option for accelerating ray-tracing, but only under some constraints. As we have seen the algorithm seems to work best for trees that are separable, and worse for trees that overlap. This means that the algorithm will benefit less from scenes with highly overlapping meshes. An other important observation we need to stress is that whatever the benefit we get from the fast construction of the tree, always comes at the cost of higher traversal times. The more light sources and reflection bounces in a scene, the more important it is for the traversal time to be fast. This is because checking shadows and reflections requires extra tree traversals. Luckily in interactive ray-tracing we usually have a small number of light sources, and not much specular reflections. Finally we would like to point out that because scenes get more and more complex over time, speeding up construction becomes all the more important. Traversal time is fixed to O($rays.log_2 n$) and construction time is O($n.log_2 n$). This means that construction time will be the bottleneck in the future, and this is the problem we address.

5.2 Future Work

There is a lot of promising work to be done to improve this algorithm. We are not satisfied with the quality of the merge in trees that overlap, and we believe that there is a lot of room for improvement in that area. From our experiences since the algorithm we proposed works better with non-overlapping trees, it would probably be a good idea to cluster the scene's polygons in non overlapping clusters, construct a tree for each cluster and then merge those trees. An other idea for improvement is building the dynamic tree using a sub-optimal construction method thus further decreasing merge times. That would probably have better render times over building the entire scene sub-optimally, because the tree for the static scene would be build with the optimal algorithm. Our algorithm could also be parallelized, and run on GPUs to produce interactive ray-tracing results.

References

HAVRAN, V. 2000. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

HUNT, W., MARK, W. R., AND STOLL, G. 2006. Fast kd-tree construction with an adaptive error-bounded heuristic. In *2006 IEEE Symposium on Interactive Ray Tracing*, IEEE.

SHEVTSOV, M., SOUPIKOV, A., AND KAPUSTIN, E. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes.

SLATER, M., STEED, A., AND CHRYSANTHOU, Y. 2001. *Computer Graphics and Virtual Environments: From Realism to Real - Time*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

WALD, I., AND HAVRAN, V. 2006. On building fast kd-trees for ray tracing, and on doing that in o(n log n). 61–69.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph. 26*, 1, 6.

WALD, I., MARK, W. R., GUNTHER, J., BOULOS, S., IZE, T., HUNT, W., PARKER, S. G., AND SHIRLEY, P. 2007. State of the Art in Ray Tracing Animated Scenes. In *Eurographics 2007 State of the Art Reports*.