Fast Ray Tracing Techniques

by

John A. Tsakok

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2008

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In the past, ray tracing has been used widely in offline rendering applications since it provided the ability to better capture high quality secondary effects such as reflection, refraction and shadows. Such effects are difficult to produce in a robust, high quality fashion with traditional, real-time rasterization algorithms. Motivated to bring the advantages to ray tracing to real-time applications, researchers have developed better and more efficient algorithms that leverage the current generation of fast, parallel CPU hardware within the past few years. This thesis provides the implementation and design details of a high performance ray tracing solution called "RTTest" for standard, desktop CPUs. Background information on various algorithms and acceleration structures are first discussed followed by an introduction to novel techniques used to better accelerate current, core ray tracing techniques. Techniques such as Omni-Directional Packets, Cone Proxy Traversal and Multiple Frustum Traversal are proposed and benchmarked using standard ray tracing scenes. Also, a novel soft shadowing algorithm called Edge Width Soft Shadows is proposed which achieves performance comparable to a single sampled hard shadow approach targeted at real time applications such as games. Finally, additional information on the memory layout, rendering pipeline, shader system and code level optimizations of RTTest are also discussed.

Acknowledgement

I'd like to thank my supervisors: Bill Bishop and Andrew Kennings for their help and support that made this research possible. Also, I'd like to thank the folks at ompf.org/forum for sharing their thoughts, ideas and resources.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recently, there has been a surge of new interest in ray tracing as a core algorithm for rendering for real-time applications such as video games since it is now feasible to perform such tasks at an interactive rate on current generation CPU hardware. In theory, ray tracing provides many advantages to the traditional rasterization used in today's rendering applications and graphics processing units (GPUs) such as its ability to be easily parallelized and capture many secondary rendering effects such as shadows, reflections and refraction.

At a high level, rasterization iterates through scene geometry and projects individual polygons into screen space for rendering. A Z-buffer is traditionally used for correct front to back occlusion which can lead to redundant raster and shading calculations which is commonly called overdraw. There are many effective culling algorithms which are aimed at reducing overdraw but none solve the problem completely. One of the advantages to a rasterization renderer is that it can be interfaced to in a simple, immediate-mode manner in which the application only specifies a set of geometry to the renderer per frame without any need for the renderer to know about any acceleration structure (AS) of the scene. This advantage leads to a simple hardware implementation which can be fast and able to easily handle dynamic scenes. Another advantage to a rasterization renderer is that polygons are processed one at a time which yields to high memory coherence during shading. This could also be a disadvantage when there is high overdraw as most of these efficient shading computations would be wasted if occluded. Additionally, a major disadvantage to rasterization is its inability to properly render secondary effects such as shadows, reflections and refraction. Such effects require multiple pass rendering or precomputed maps which are inefficient and produce visible artifacts.

At a high level, ray tracing shoots rays called primary rays from the camera into the scene and finds intersections with the scene geometry. Secondary rays can be cast from the intersection points into the scene to query the scene for secondary effects. Using these intersections, final colours are calculated through shading to produce the final image. Since the rays find the first intersection with the scene, there is theorectically zero overdraw which gives this technique a higher drawing

efficiency over rasterization. The disadvantage to this technique is that there is less memory coherency since each ray is traced independently even if many hit the same polygon.

One of the greatest advantages and motivators of ray tracing is that each ray can be traced independent of each other which leads to an easily parallelizable algorithm. With the industry trend moving from complex, single, out-of-order CPUs with large caches and speculative hardware to many in-order CPU cores, parallelization is a very important factor of a high performance, scalable rendering algorithm. Another significant property of ray tracing is its ability to trace secondary rays for capturing secondary effects correctly which is a key to improving the fidelity of current rendering techniques.

Unlike rasterization, ray tracing depends on an AS which subdivides the scene for efficient searching of intersection points. The disadvantage to depending on an AS is that it leads to a more complex hardware implementation since the rendering interface is no longer immediate and the AS must be rebuilt or updated for dynamic scenes which is not easily parallelizable.

Ray tracing alone is still an unproven algorithm for a robust solution to the future of rendering. There have been many proposals for a hybrid solution involving both rasterization and ray tracing in which rasterization provides fast rendering of primary effects such as geometry rendering while ray tracing provides an efficient and correct rendering of secondary effects.

This thesis goes in depth into the design and implementation of a high performance ray tracing solution for current generation x86 CPUs. New ideas and algorithms are presented for the acceleration and simplification of current kd-tree traversal techniques and secondary rendering effects such as soft shadows. A new omni-directional packet traversal method is proposed which simplifies packet tracing by allowing any bundle of rays from a common origin to be traced without any direction restrictions (Omni-Directional Packets). Next, a fast technique of traversing multiple frustums simultaneously is proposed which allows for the masking of individual frustums resulting in higher performance over a single frustum approach (Multiple Frustum Traversal). An alternative to a traditional pyramidal frustum is proposed using cones which can be used as an interval traversal algorithm for tracing groups of rays that have a conical shape such as multiple shadow samples to a spherical light source (Cone Proxy Traversal). Finally, a fast, approximated technique for rendering soft shadows is proposed (Edge Width Soft Shadows). This approach requires the use of one shadow ray to produce very fast, fuzzy shadows which are superior in visual quality over hard shadows with comparable performance.

Chapter 2

Background

2.1 Ray Tracing Overview

Ray tracing is a recursive algorithm that first traces primary rays from the eye into a scene to search for intersections with geometry and in turn traces secondary rays from these intersection points (IPs) into the scene again as shown in Figure 2.1. In the figure, a primary ray, which is searching for the first intersection along the ray, is traced from the eye into the scene. If the red polygon that intersected the primary ray was neither reflective nor refractive, both the reflect and refract rays can be ignored. However, a shadow ray is required to search for any occluders between the IP on the polygon and the light source. If an occluder is found, then no light is contributed to the IP and is thus shadowed. On the other hand, if the polygon were reflective and refractive, secondary reflect and refract rays are traced to sample the scene for colour contributions to the final rendering of the pixel corresponding to the primary ray sent from the eye. The logic and math used to calculate the final pixel colour are known as the shading method or shader which uses information such as the polygon colour, shadow, reflect and refract ray results as inputs. Any secondary rays can produce more secondary rays recursively to better sample the scene at the cost of performance.

For a ray tracing application, this process of shooting a primary ray from the eye, tracing secondary rays and calculating shading for a final colour is performed once for each pixel on the screen. To allow for anti-aliasing, multiple ray samples can be traced through a single pixel which are jittered at sub pixel distances to produce a final screen pixel colour. A key advantage and disadvantage of ray tracing is that rays are all traced independently of one another. The advantage is that it allows for an easy parallel implementation which can achieve high performance as more computational units are available. The disadvantage to ray independence is that there is no data coherence between pixels that are adjacent on the screen even though they hit the same object and perform the same shading operations. Figure 2.2 shows two polygons being rendered with the pixels shown in the grid. For this scene, rasterization excels since data coherence can be used during scanline

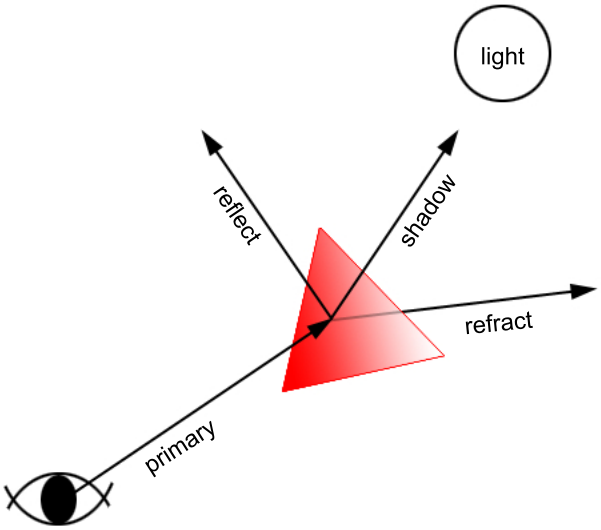

Figure 2.1: Ray tracing diagram where primary ray is traced from the eye to the scene casting secondary rays.

rendering where there is no overdraw and wasted calculations. On the other hand, ray tracing wastes computation by tracing the scene to the polygons and shading for each pixel without using any data coherence between adjacent pixels. Figure 2.3 shows another scene with 3 polygons overlapping each other. In this case, the advantages to ray tracing start to show as data coherence becomes less important as many adjacent pixels belonging to a polygon should not even be drawn since they are occluded by another polygon. As scene complexity increases, more occlusion occurs leading to more overdraw for a rasterization approach. It begins to turn out that the fast, data coherent rendering becomes wasted in the end for highly complex scenes. In the end, both rasterization and ray tracing are good at solving a subset of graphics problems but no algorithm is good for everything. This thesis will go indepth into how to leverage the benefits of ray tracing while introducing methods to make up for its shortcomings.

2.2 Acceleration Structures

As scene complexity grows, interactive ray tracing becomes infeasible if each ray must be checked against each polygon in the scene to determine the closest intersection. Much like how binary search can be used to do fast searching into a sorted list, the scene's polygons must be sorted to allow for fast searching for intersection points. This approach yields to the concept of an acceleration structure (AS) which can be built offline prior to rendering for static scenes and updated or rebuilt every frame for dynamic scenes. This section introduces three popular acceleration structures: kd-trees, bounding volume hierarchies and grids for high performance

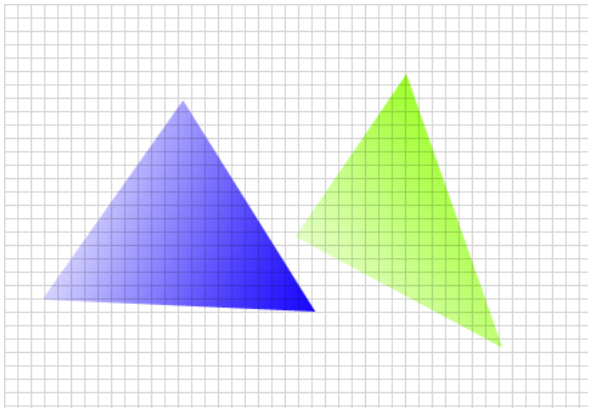

Figure 2.2: Two polygons with no overdraw.

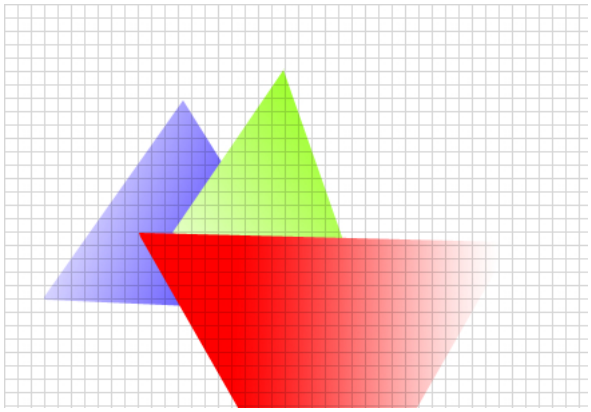

Figure 2.3: Three polygons with overdraw.

ray tracing.

2.2.1 kd-Tree

A kd-tree is basically a binary tree that defines a spatial hierarchy. Each internal node represents a 3D slab or voxel and an axis-aligned split plane which subdivides the voxel into 2 sub voxels which represent the child nodes which are referred to as the *left* and *right* child. Figure 2.4 shows a 2D space using kd-tree partitioning where each space is recursively subdivided into 2 sub spaces. For example, split plane 1 subdivides the whole space into a left and right spaces relative to the direction of the plane. In the left space, split plane 4 subdivides the space into sub spaces labeled A and E. Figure 2.5 shows how the diagram in Figure 2.4 relates to an actual kd-tree. The root node, labeled 1, represents the whole scene and split plane 1. As each node is subdivided to produce child nodes lower in the tree, the nodes become smaller due to splitting of space at each level. Once a space has stopped subdividing, it is allocated in a leaf node labeled as a square in the tree.

kd-trees are useful because they provide a structure of sorting objects/polygons within a scene to allow for fast searching. Once a tree has been built, the 3D space has been partitioned into leaf nodes where any single point within the 3D space is contained in exactly one leaf node. Figuring out which leaf node a point belongs to is on the order of $O(\log(n))$ since at each level in the tree, half of the leaf nodes are discarded on average.

In ray tracing, kd-tree leaf nodes contain the actual geometry where the internal nodes are required for sorting purposes only. Each internal node partitions the polygons into two sets: the polygons that lie in front of the split plane and those that lie behind. Since there are no restrictions to split plane positions, polygons can belong to more than one leaf node if it straddles a split plane as shown in Figure 2.4 where a polygon belongs to leaves D and H since it's split by node 5.

To achieve fast performance, the goal is to locate the polygon that a ray first hits as fast as possible. To achieve this goal, kd-trees are not used to find which leaf node a point is located in but which leaf nodes, in front to back order, a ray visits when traced through a scene. In a kd-tree, it is guaranteed that if a leaf is traversed before a second leaf during ray traversal, polygons contained within the first leaf will be in front of the polygons contained within the second. However, there is no assumed order to the polygons within the same leaf.

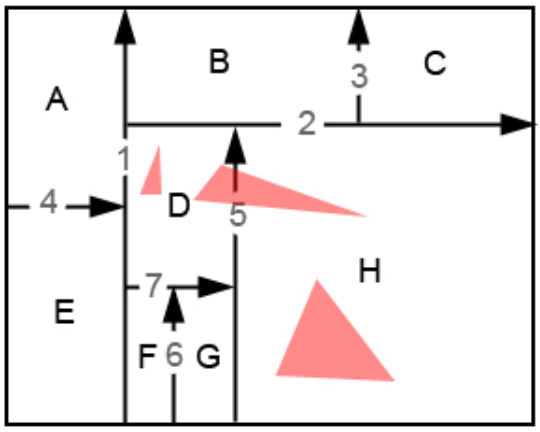

Figure 2.4: A 2D space subdivided by a kd-tree. Polygons contained within the scene are shown in red.

kd-Tree Traversal

Figure 2.6 shows the same 2D space as in Figure 2.4 but with a ray tracing through the scene. This example is used to help describe how a kd-tree is used for ray traversal. As the ray is traversed through the scene, the leaf nodes that are visited in front to back order are E, F, G and H. The job of the traversal algorithm is to be able to figure out what leaves are visited by a ray as fast as possible. At each

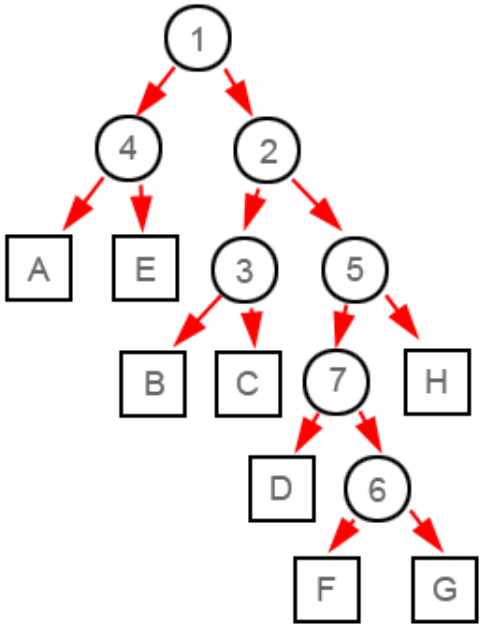

Figure 2.5: kd-tree representation of a 2D scene in Figure 2.4.

internal node in the tree, two cases can occur as shown in Figure 2.7. In the figure, an internal node is shown with a split plane where a ray is pointing in the positive direction and negative direction relative to the split plane. By convention, the child nodes are labeled "near" and "far" based on if they are closer or farther from the origin of the ray, respectively. kd-tree traversal keeps track of intervals along the ray which represents the active part of the ray in the current node. Before tracing, the interval of the ray should be from 0 to the length of the ray. However, as rays visit internal nodes, intervals can be split at splitting planes if the ray intersects the split plane within the interval of interest. Figure 2.8 shows the three cases that can occur when a ray visits an internal node where the intervals are labeled from *tn* to *tf* which are distances along the ray from the origin. In the left picture, the ray intersects the split plane within the current interval. In the centre picture, the ray intersects the split plane after the current interval. In the final picture, the ray intersects before the interval. Using these cases and keeping track of intervals of interest along the ray, a simple, recursive traversal algorithm can be derived as shown in Listing 2.1.

In the listing, the function *Traverse* takes input vectors *dir* and *origin* which are the direction and origin of the ray respectively. The values *tNear* and *tFar* are the starting intervals and are usually 0 and the length of the ray respectively. The next part of the function checks to see if the node is a leaf node or an internal node. For the case of an internal node, the *near* and *far* nodes are set relative to the ray direction as shown in Figure 2.7. The distance along the ray to the split plane is then calculated and stored in the value *dist*. Next is the logic deciding whether the

near, far or both children should be visited based on the current interval and the split plane distance. The first case occurs when the interval is in front of the split plane where the near child is visited only. The second case is when the interval is behind the split plane resulting in only the far node being visited. The third case occurs when the split plane is within the interval and thus the near child must be visited depth first followed by the far node. In this case, the interval is split at the split plane as the near child has an interval of *tNear* to *dist* and the far child has an interval of *dist* to *tFar*. Also, the far node is only visited if an intersection was not found from traversing the near node since no further traversing is necessary in that case. When the traversal function reaches a leaf node the ray is intersected with each polygon and the closest distance that lies within the active interval is returned. The function completes when either all leaf nodes that are intersected by the ray are visited or an intersection is found. For a high performance ray tracer, the function shown should be rewritten so that it is no longer recursive and dependent on the processor stack which is slow and can easily overflow for deep trees. Listing 2.2 shows an iterative version of the traversal method which maintains its own stack. The logic is very similar to the recursive version except that an explicit stack is maintained using *stackelement* types which hold tasks that need to be completed such as the node to traverse and the active interval. The nice part about kd-traversal is that the actual floating point math required is minimal per traversal step since the split planes are axis aligned. The division by *dir* can also be turned into a multiply by precomputing the inverse of the direction vector before traversal. The next section describes how to build an efficient kd-tree structure.

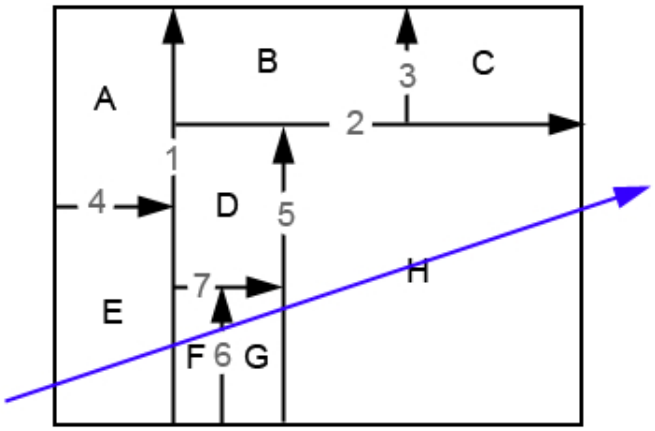

Figure 2.6: A 2D space subdivided by a kd-tree.

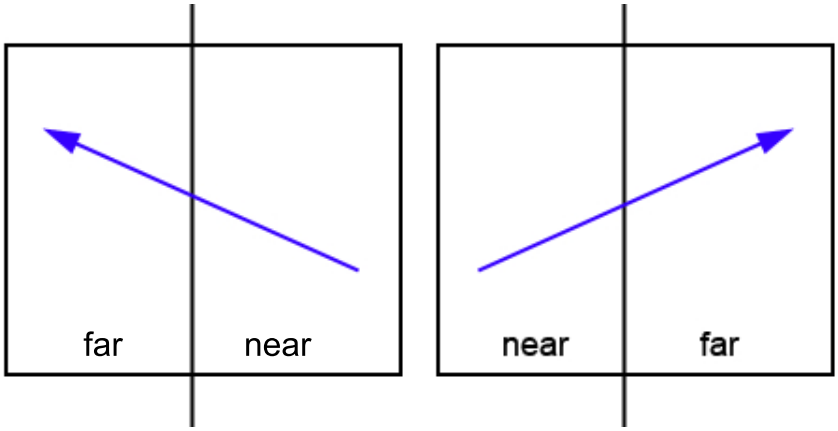

Figure 2.7: A ray within a node showing near and far voxels.

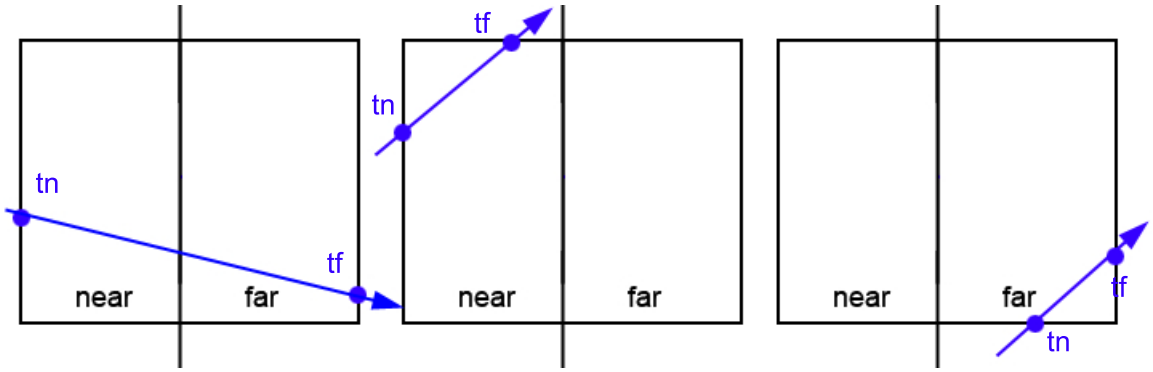

Figure 2.8: Different cases that occur with intervals and a split plane.

Listing 2.1: Recursive, kd-tree traversal algorithm

```
float Traverse(Vector dir, Vector origin, float tNear, float tFar, kdnode *n)
{
    if(!n->isLeaf){

        kdnode *near, *far;

        if(dir[node->axis] > 0.0f){
            near = node->left;
            far = node->right;
        } else {
            far = node->left;
            near = node->right;
        }

        float dist = (node->split - origin[node->axis])/dir[node->axis];

        if(tFar < dist)
            return Traverse(dir, origin, tNear, tFar, near);

        if(tNear > dist)
            return Traverse(dir, origin, tNear, tFar, far);

        float rval = Traverse(dir, origin, tNear, dist, near);

        if(rval == FLT_MAX)
            Traverse(dir, origin, dist, tFar, far);
        else
            return rval;
    }
    else
    {
        float shortestDist = FLT_MAX;
        float dist;
        for(int i=0; i<node->polyCount; i++){

            if(Intersect(node->polys[i], dir, origin, &dist)){
                if(dist <= tFar && dist >= tNear)
                    shortestDist = min(dist, shortestDist);
            }
        }

        return shorestDist;
    }
}
```

Listing 2.2: Iterative, kd-tree traversal algorithm

```
float Traverse(Vector dir, Vector origin, kdnode *n)
{
    float tNear, float tFar;
    nodestack stack;
    stack.push((n, 0, MAX_DIST));

    stackelement element;

    while(!stack.isEmpty()){

        element = stack.pop();

        n = element.node;
        tNear = element.tNear;
        tFar = element.tFar;

        while(!n->isLeaf){

            kdnode *near, *far;

            if(dir[node->axis] > 0.0f){
                near = node->left;
                far = node->right;
            } else {
                far = node->left;
                near = node->right;
            }

            float dist = (node->split - origin[node->axis])/dir[node->axis];

            n = far;

            if(tNear > dist)
                continue;

            n = near;

            if(tFar < dist)
                continue;

            tFar = dist;

            stack.push((far, dist, tFar));
        }
```

```
        if(n->isLeaf){

            float shortestDist = FLT_MAX;
            float dist;
            for(int i=0; i<node->polyCount; i++){
                if(Intersect(node->polys[i], dir, origin, &dist)) {
                    if(dist <= tFar && dist >= tNear)
                        shortestDist = min(dist, shortestDist);
                    }
            }

            if(shortestDist != FLT_MAX)
                return shorestDist;
        }
    }
}
```

kd-Tree Building

Building of a kd-tree is effectively picking a split plane for every node and having a terminating criteria for subdivision. The first question is how to pick proper split planes? A naive approach would be to pick split planes that divide the current node in half as shown in Figure 2.9 where the arrowed lines are the split planes and the grey object is the scene geometry. In the diagram, each node is split in half at each level in the tree. This approach leads to a balanced tree but is only helpful if the goal was to figure out what leaf node a certain point is in quickly as the traversal distance to each leaf is equal. However, the goal for kd-tree traversal is to quickly visit polygons that lie on the ray's path and find the closest intersection. By picking a middle split plane, the geometry is not taken into account for the splitting decision which results in large leaf nodes with lots of geometry. The disavantage to having this is that whenever a ray traverses through a leaf node, it must check for intersection with all the polygons contained within the leaf even though it does not intersect any of polygons. An example of this is shown in Figure 2.9 where the ray intersects several leaf nodes with geometry. The ray must perform intersection tests even though it is far away from the actual geometry. To minimize these redundant intersection tests, the kd-tree builder should maximize the size of leaves containing little to no geometry and minimize the size of leaves containing more geometry. By doing this, there is a lower probability of a ray hitting a leaf with many polygons and thus reducing intersection tests. Figure 2.10 shows a high quality kd-tree built for the same scene as shown in Figure 2.9. The diagram shows how the splitting planes adapt to the scene geometry and maximize the size of empty leaves. As the blue ray travels through the scene, it no longer must perform any intersection tests since all the leaves visited are empty and thus saving precious computational time.

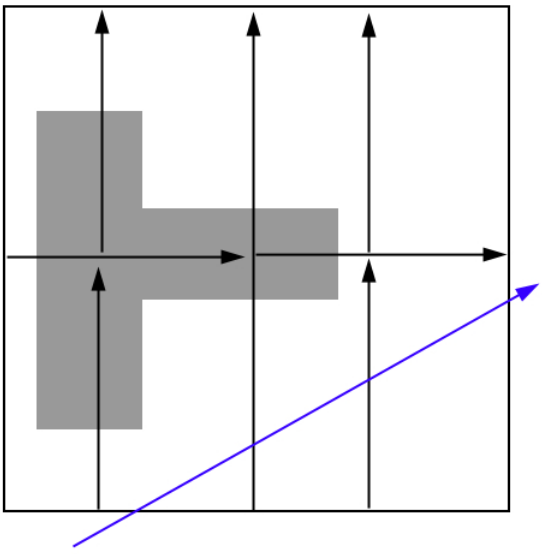

Figure 2.9: Scene with kd-tree built by using middle split planes.

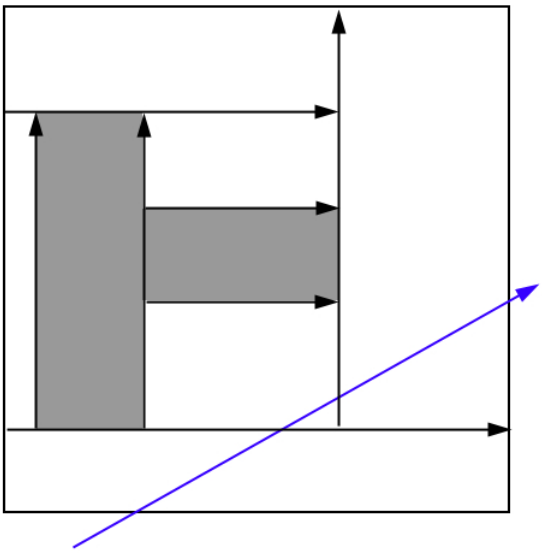

Figure 2.10: Scene with kd-tree built by using high quality split planes.

A more precise definition of picking a proper split plane is given by minimizing what is called a Surface Area Heuristic (SAH) [1] function given in Equation 2.1. A SAH is a cost function that must be minimized to achieve a more optimal split plane decision. C_{split} represents the cost of the split which is trying to be minimized. $C_{traversal}$ is the fixed cost for traversal which represents the traversal cost of splitting the current node since more splits results in a deeper tree and thus more traversal runtime costs. $C_{intersect}$ is the fixed cost for intersecting a polygon. The actual values of $C_{traversal}$ and $C_{intersect}$ do not matter but the ratio between the two are important. In this thesis, $C_{intersect}$ was picked to be $5\times$ that of $C_{traversal}$ which yielded the best results in most cases. Num_{left} is the number of polygons in the left subnode. SA_{left} represents the surface area of the left subnode and SA is the surface area of the whole node. The whole term $\frac{SA_{left}}{SA}$ represents the probability that a ray will hit the left subnode and $\frac{SA_{right}}{SA}$ is the probability for the right. Qualitatively, this cost function represents the computational cost that results for a split as the added traversal cost for adding another level to the tree plus the intersection costs of the subnodes based on the probability that the subnode is hit times the cost of the node which is really the number of polygons it contains. This cost function motivates larger leaf nodes containing little to no polygons and smaller leaf nodes containing many polygons. Using a SAH does not guarantee a global minimum cost kd-tree since it is a greedy algorithm that makes the local decision of picking the most optimal split plane for each node with no information about the tree as a whole. However, finding the global optimal solution is an intractable problem since it is impossible to know if a split plane decision is the most optimal without knowing the future.

$$C_{split} = C_{traversal} + C_{intersect} \times (Num_{left} \times \frac{SA_{left}}{SA} + Num_{right} \times \frac{SA_{right}}{SA}) \quad (2.1)$$

Using this SAH function, split planes are picked which yield the minimum cost at each node where subdivision terminates when any of the following are true:

1. there exists no split plane that yields a cost less than the current node cost $C_{intersect} \times N$,
2. the current node has reached a pre-defined maximum depth, or
3. the current node contains no polygons.

In the first case, the current node does not benefit from splitting as the cost of the actual node is the minimum. In the second case, if a pre-defined depth is set, subdivision should stop. A maximum depth is useful to bound memory consumption, particularly for large scenes. Maximum depths are also useful to prevent leaves from becoming too far from the root node which can become a performance problem as too many traversal computations are required to get to the leaves. $C_{traversal}$ of the SAH function should provide the same constraint but is not a hard, explicit constraint like a maximum depth. The last termination case is obvious as subdivision should stop when there are no more polygons to sort in the current node.

For the special case when split planes divide a polygon, the polygon belongs to both the left and right subnodes. This leads to more memory usage as these polygons are referenced by more than one leaf node. Because of this special case, it becomes difficult to predict beforehand how much memory is required to build a kd-tree with a reasonable quality as opposed to bounding volume hierarchies discussed in the following section.

The SAH cost function and terminating criteria have been discussed but which possible split planes to choose from have not been. The SAH cost function is a continuous, piecewise, linear function with respect to the split plane position along an axis, where the only points of interest are local minima/maxima where the slope changes. These points occur when the number of polygons in the left or right sub voxels changes, Num_{left} and Num_{right} in the SAH, as the surface area of each sub voxel is linear with respect to split plane position along an axis. Figure 2.11 shows a node containing polygons with candidate split positions labeled where Num_{left} and Num_{right} change value. For example, split position e has a value of $Num_{left} = 3$ and $Num_{right} = 2$ where the polygon straddling the split plane is counted for both sub voxels. Because of this observation, split plane positions can be taken as the edge of the bounding boxes for each polygon along the axis of interest. In this case, to find the most optimal SAH cost for a certain node, the SAH must be calculated for each bounding box edge (2 per polygon along an axis) for each axis (3) which

works out to being $N \times 2 \times 3 = 6N$ candidate split planes. Although using bounding box edges provides adequate results in most cases, they can be far from optimal. In Figure 2.12, candidate split planes are shown for the same node but along a vertical axis where the bounding box edges are used as candidate split planes. The grey polygon to the far right which is not fully contained in the current node of interest has a bounding box shown in blue which is used to generate a candidate split at h. Since the range of the bounding box is from the top of the node to split h, the polygon must be part of both sub voxels for all split planes above h, which in this case is all of them. The disadvantage to this is that if a split is picked at g, the polygon will be part of the lower subvoxel even though it is not even contained within it. Because of this inefficiency, this leads to the idea of a "perfect" candidate split planes which are not taken as edges of bounding boxes of polygons but from bounding boxes of clipped polygons. By clipping the grey polygon to the current node's box, the resulting bounding box shown in green is tighter and more optimal. Now, h is no longer a candidate and g is one. Also, any splits below g will not contain the grey polygon within the lower subvoxel thus reducing the number of polygons contained within leaf nodes. By using a SAH cost function with terminating criteria discussed and polygon clipping for generating "perfect" split candidates, a high quality kd-tree can be built for high performance traversal.

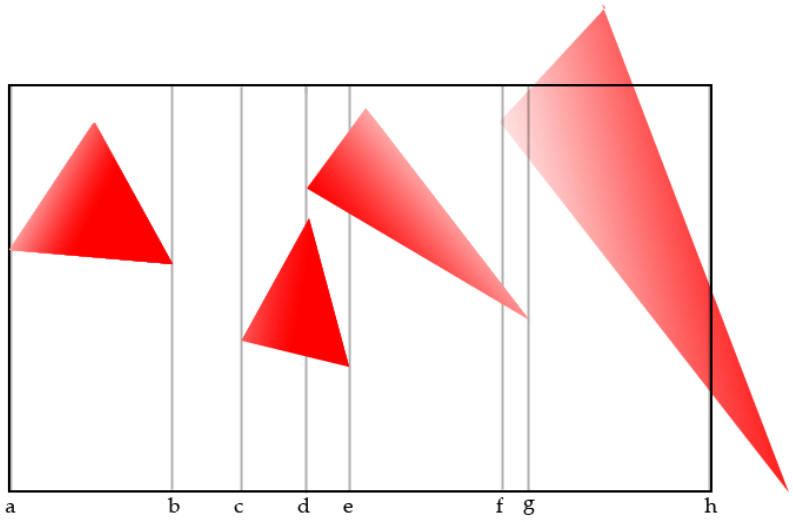

Figure 2.11: Node containing polygons and candidate splits along the horizontal axis.

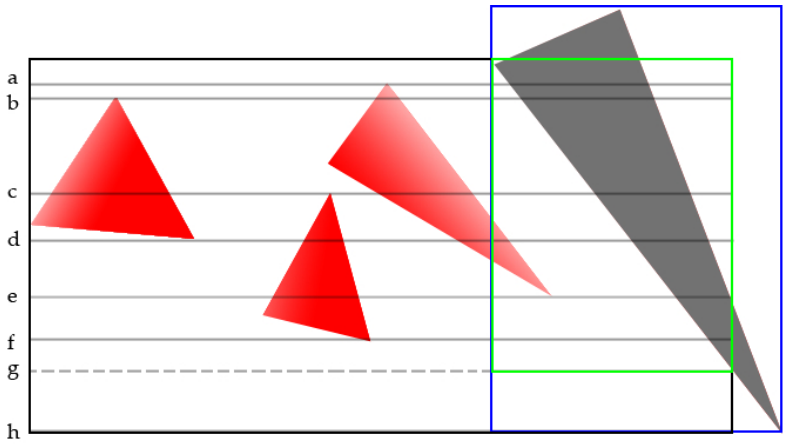

Figure 2.12: Node containing polygons and candidate splits along the vertical axis.

2.2.2 Bounding Volume Hierarchy

The bounding volume hierarchy (BVH) is an object hierarchy as opposed to the kd-tree which is a spatial hierarchy. This means that the tree sorts objects or polygons rather than strict spatial sorting. A BVH is a binary tree which partitions a group of polygons into two sets at each level in the tree where the leaf nodes contain the actual polygons like kd-trees as shown in Figure 2.13. In the diagram, the polygons are partitioned into two sets where the green and blue boxes are the resulting bounding boxes of the children nodes. Rather than a single split plane which divides the space, the polygons are just put into two sets with no spatial sorting. In general, the BVH is a simpler concept than a kd-tree in terms of traversing, building and updating dynamic scenes. BVHs have not been thoroughly implemented in this thesis' research test system due to time restrictions so they are only discussed in the Background Section but many ideas discussed in this thesis can be applied to both a kd-tree and a BVH. Though BVHs tend to produce a less optimal AS for static scene rendering, they provide a very good structure for dynamic scenes and this is essential to the future of real time ray tracing.

BVH Traversal

Traversing a BVH is very simple since no distance intervals have to be maintained and no "near" and "far" voxels must be defined. The only operation that must be done per traversal is a fast ray/slab intersection test [2] which checks to see if there is an intersection between the ray and the node's bounding box. If no intersection occurs, then no further traversal is done into the node whereas if an intersection does occur, the children nodes must be checked since they can potentially be intersected with. Listing 2.3 shows an iterative BVH traversal pseudocode. At each node a *SlabTest* is performed and if an intersection occurs, the stack pushes one of the children onto the stack and starts to traverse the other child. A disadvantage of

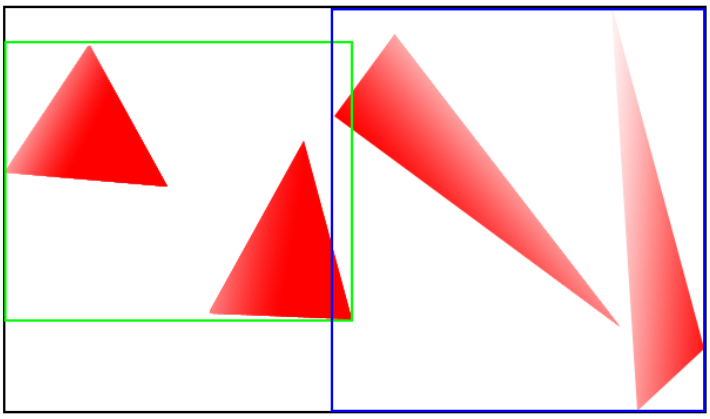

Figure 2.13: BVH node containing polygons which have been partitioned into two sets.

BVH traversal is that there is no early exit case where traversal stops when the ray intersects a polygon as done in the kd-tree traversal. The reason for this is that there is no guarantee that a closer intersection does not exist in another node on the stack since there is no spatial sorting amongst nodes. Because of this, the stack must be empty before the traversal finishes even in the case when the ray has already found the closest intersection. Several heuristics can be used to provide an earlier exit by using some spatial information such as choosing which child to traverse first based on the direction of the ray and the relative positions of both nodes. By using spatial heuristics to pick the node closest to the origin of the ray to traverse first, nodes can be culled if an intersection is found as any nodes further from the intersection distance will not provide a closer intersection.

Listing 2.3: Iterative, BVH traversal algorithm

```
float Traverse(Vector dir, Vector origin, bvhnode *n)
{
    nodestack stack;
    stack.push(n);

    stackelement element;

    float t = MAX_DIST;

    while(!stack.isEmpty()){
        n = stack.pop();

        while(!n->isLeaf){
            if(SlabTest(n, dir, origin)){
                stack.push(n->child1);
                n = n->child2;
            }
        }

        if(n->isLeaf){
            float shortestDist = FLT_MAX;
            float dist;
            for(int i=0; i<node->polyCount; i++){
                if(Intersect(node->polys[i], dir, origin, &dist)) {
                    if(dist <= tFar && dist >= tNear)
                        shortestDist = min(dist, shortestDist);
                }
            }

            t = min(t, shortestDist);
        }
    }

    return t;
}
```

BVH Building

Since at each node, BVH partitions polygons into two sets, there are $2^N - 1$ possible partitions for N polygons. By looking at Figure 2.13 it's obvious that many of these partitions are suboptimal as they don't partition the space well. For example, picking a polygon from the far right and from the far left to be in the same set will yield a partition which is comparable in size to the parent node. This observation leads to using a spatial heuristic to pick object partitioning such as an SAH cost function used in kd-tree building. Since BVH is an object paritioning, there is no

fine grain control over the SAH as spatial division is implicit from the bounding boxes of the partitioned set. This leads to suboptimal SAH when compared to a kd-tree implementation which has no restriction on spatial division. Once the objects are paritioned into two sets to produce a minimum SAH, the children are subdivided just like building kd-trees with the same termination criteria.

2.2.3 Grid

A grid is a 3D, uniform partitioning of voxels of the scene space where each voxel usually has a unit length (1.0). To figure out which voxel a point is located in, the floating point value of its coordinates are truncated to integer coordinates. Because of this simple mapping, grids are very useful for dynamic scenes where a moving object can be easily updated by using either its geometry or bounding box to set which voxels it is currently located in. Typically, 3D digital differential analyzers (3DDA) are used to trace the rays from voxel to voxel in front to back order though the grid. Other faster algorithms have also been proposed that trace groups of rays together in packets or frustums using a slice-based approach [3].

A disadvantage of grids are that they do not adapt well to varying geometric sizes and densities where some parts the scene can have many large polygons, some many very small polygons and some with none. Despite the geometric attributes, a grid will have the same uniform voxel size for each case. This problem can be mitigated with a hierarchical grid approach where there is a top level grid with larger voxels containing a finer grain grid than a lower level. However, as more levels are introduced into the hierarchy the slower it is to update for dynamic scenes which is one of the key the selling points for such a structure.

2.3 Faster kd-Tree Traversal

This section describes techniques for greatly accelerating the kd-traversal algorithm which take advantage of the coherency between adjacent primary rays. This coherency is a measure of how closely rays trace the scene and follow a similar path when traversing the kd-tree. Adjacent primary rays that form a square tile on the screen can be traced together, thus decreases the revisiting of kd-tree nodes which would have been visited by each ray if traced individually and reducing memory bandwidth. However, as tile sizes increase and more rays are traced together, there is less coherency between the rays since they start to diverge at the lower part of the tree when the granularity of paritioning becomes finer. The following sections use this idea of coherency, tiles and tracing rays together to achieve superior performance to that of the traditional mono ray tracing algorithms.

2.3.1 Packets

The idea of tracing multiple rays together is called packet or bundle tracing [4] where the screen is split into square tiles which determine the size of the packet such as a 2×2 packet traces a tile of width 2 containing 4 rays. However, as rays start to diverage towards the bottom of the tree, inactive rays are masked out so as to keep correctness as shown in Figure 2.14. In the figure, rays R1 to R4 are traced together in a packet visiting Nodes A, B and C. When traversing Node B, R1 and R2 are inactive and must be masked out while R3 and R4 remain active and are allowed to affect further traversal decisions within Node B and intersect any geometry within the node. Figure 2.15 shows two cases which can occur at a split plane. The intervals labeled on each ray are just an example of a possible interval case that might occur. Like mono tracing traversal, voxels are determined to be either *near* or *far* based on the direction of the rays as the near voxel is closest to the origin. This determination leads to a directional restriction on the rays within a packet as they must all have the same direction sign for each axis in order to be able to agree on which voxel is *near* and *far*. Packets that do not obey this restriction must be split into smaller packets that have same direction signs. In the Omni-Directional Ray Packet section, a solution to this problem is presented which allows for no direction restriction and leads to a simpler implementation. Listing 2.4 shows pseudocode for a standard packet traversal of 4 rays. In the pseudocode, each variable with an array index is a 4 wide variable which is used for each of the 4 rays. The first part of the traversal checks each rays' interval to determine if it is active in the current node. Rays that are inactive have an interval in which $t\_far \leq t\_near$ which is a convenient side effect of the way the intervals are updated. The next step is to determine if all the rays are inactive which, if true, stops the traversal of the current node. The next step is the same as the mono ray traversal which calculates the distance to the splitting plane. Using this distance, the split plane distance along each ray is calculated. Using the direction of the first ray, the *near* and *far* children are defined. Once the far child is determined, it is pushed on the stack with the updated interval for that node and traversal continues into the near node.

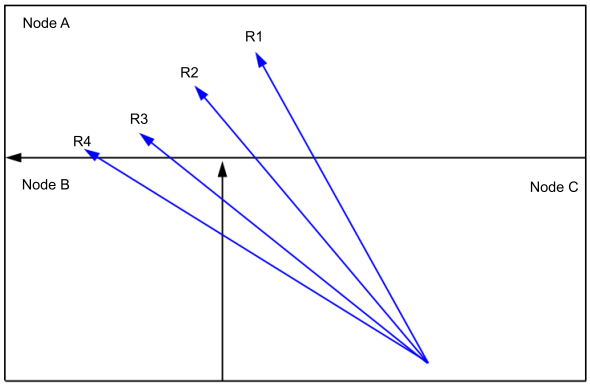

Figure 2.14: Diagram of 4 rays being traced together using packet travesal.

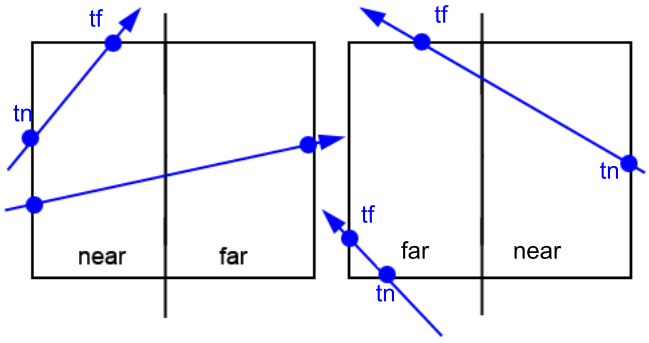

Figure 2.15: Diagram of 2 rays being traced together for 2 cases.

Listing 2.4: Iterative, kd-tree 2x2 packet traversal algorithm

```
while ( !node.isLeaf ) {
    active[i] = ( t_near[i] < t_far[i] );

    if (for all i=0..3(!active[i]))
        break;

    dist = split − origin[axis];

    d[i] = dist / dir[i][axis];

    Node ∗near, ∗far;

    if(dir[0] < 0.0f) {
        near = ( KDTreeNode ∗ ) node.left;
        far = ( KDTreeNode ∗ ) node.right;
    } else {
        near = ( KDTreeNode ∗ ) node.right;
        far = ( KDTreeNode ∗ ) node.left;
    }

    stack.push( far, max( d[i], t_near[i] ), t_far[i] );
    ( node, t_far[i] ) = ( near, min( d[i], t_far[i] ) );
}
```

Using SIMD

By using standard packet traversal with up to 4 rays per packet, memory bandwidth due to node traversal can be reduced by a factor of 4 in the most optimal circumstances where the scene is not too complex. However, computations within the traversal loop must be performed per ray which increases the compute time by a factor of 4 within the traversal loop. By using an Single Instruction Multiple Data (SIMD) instruction set such as Intel's SSE, floating point computations can be sped up by a factor of 4. To fully utilize SSE for tracing 4 rays at a time, data must be reorganized in a manner which is easy to load and unload from these special XMM vector registers.

To best take advantage of architectures with different SIMD widths, data must be reorganized from an "array of structures" format to a "structure of arrays" format. For example, in Listing 2.5, a *Vector* structure is defined and a group of four vectors in *FourVectors* which show a standard "array of structures" grouping. Since SIMD instructions do not perform fast horizontal operations on elements within a vector but element by element operations between two vectors, an "array of structures" format requires data reorganization to do any useful computations on it. Listing 2.6 shows a "structure of arrays" format which lends itself to a SIMD

architecture. The same structure of 4 vectors is now rearranged such that each axis is an array of data for each vector. Using this new format, four vectors can be added to four other vectors using SSE without any data rearrangement as shown in Listing 2.6. First, the data is loaded into the XMM registers for each axis. Next, the vector elements are added four at a time and finally stored back into memory as the output. Such a format can be used throughout the ray tracing pipeline from ray generation to shading to reduce floating point computations significantly.

Listing 2.5: "array of structures" format for a group of 4 vectors

```
typedef struct{
    float x,y,z;
} Vector;

typedef struct {
    Vector v[4];
} FourVectors;
```

Listing 2.6: "structure of arrays" format for a group of 4 vectors with an example of adding 4 vectors to another 4

```
typedef struct {
    float x[4], y[4], z[4];
} FourVectors;

void VectorAdd(FourVectors *out, FourVectors *in1, FourVectors *in2)
{
    __m128 r1[3], r2[3], ro[3];

    r1[0] = _mm_load_ps(in1->x);
    r1[1] = _mm_load_ps(in1->y);
    r1[2] = _mm_load_ps(in1->z);

    r2[0] = _mm_load_ps(in2->x);
    r2[1] = _mm_load_ps(in2->y);
    r2[2] = _mm_load_ps(in2->z);

    for(int i=0; i<3; i++)
        ro[i] = _mm_add_ps(r1[i], r2[i]);

    _mm_store_ps(out->x, ro[0]);
    _mm_store_ps(out->y, ro[1]);
    _mm_store_ps(out->z, ro[2]);
}
```

2.3.2 Frustums

With a SIMD width of 4 on almost all desktop CPUs, four rays can be traced together in a packet with very little overhead. Larger number of rays can be traced together in group by tracing multiple packets of 4 to further reduce memory bandwidth due to traversal. The disadvantage of doing this is that more SSE operations would have to be done in the traversal loop to calculate and update the intervals for each ray. This tradeoff leads to a frustum approach which allows a large group of rays with multiple packets to be traced together with a constant number of operations within the traversal loop with respect to the number of rays in the group. A frustum can be used to bound a large group of rays and act as a proxy during traversal such that the traversal logic has no knowledge of the internal rays and just traverses the frustum. As the frustum is traversed through the scene, it visits leaf nodes from front to back order just like ray traversal. The idea of frustum traversal is not to figure out what leaves the frustum visits but what leaves the frustums don't visit through various culling algorithms such as direct frustum culling using the frustum planes or inverse frustum culling [5] using the nodes' bounding box planes. The frustum traverses the tree by checking to see whether or not a node's bounding box intersects the frustum and if so traverses the children. When a leaf node is visited, the internal rays are used to intersect the contained polygons. This approach allows a large group of rays to be traversed together without any per ray operations within the traversal loop.

Another faster form of frustum traversal is the frustum interval traversal algorithm [5] which updates active intervals rather than using culling algorithms. The idea is similar to standard packet tracing except the way intervals are handled. Rather than tracing a single ray or a packet of rays, the extremal (boundary) rays of a frustum are traced. For primary rays, these extremal rays are usually the corner rays of the tile. Using the corner rays, distances to the split plane are calculated using SSE operations but only a single interval is updated rather than 4 intervals or a group of intervals. This single interval encompasses all the intervals of the internal rays since they are calculated with the extremal values. By using a single interval to describe a group of rays, information is lost and thus more nodes are traversed than a standard frustum traversal using culling. Figure 2.16 shows 3 cases of frustums at a node where the active segment/interval of the frustum is highlighted. Since only a single interval is used, the segment does not tightly bound the active region within the node causing redundant traversed nodes. However, the advantage to this approach is that the traversal loop requires less operations than frustum culling since only intervals are updated. Like ray packets, the extremal values of the frustum must have the same direction signs. Pseudocode for the interval traversal algorithm is provided in Listing 4.2 where the traditional traversal is modified to remove any direction restriction on rays. In the Cone Proxy Traversal section, an algorithm of tracing a cone shaped frustum using intervals is proposed instead of a pyramid shaped frustum.

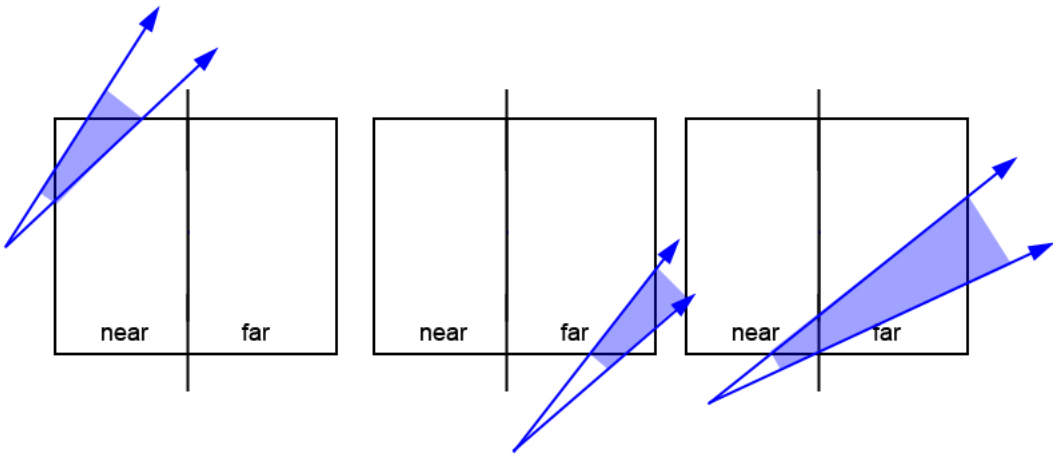

Figure 2.16: Diagram of 3 cases of frustum interval traversal.

2.3.3 Multi Level Ray Tracing Algorithm

The Multi Level Ray Tracing Algorithm (MLRT) [5] is a two stage algorithm comprised of an entry point (EP) search followed by an intersection point search (XP). The EP search stage uses a frustum traversal algorithm to search for subtrees in which all rays within the frustum must intersect a polygon. The root of this subtree is referred to as an entry point within the tree. Once an entry point is found, the XP search begins as all the internal rays within the frustum can be traced for intersections from the entry point rather than the root of the tree, thus saving traversal costs. The advantage of MLRT is that it produces very fast results for scenes with large polygons and many flat, axis aligned surfaces such as walls. The reason for this is that the EP search depends on a whole frustum of rays to completely intersect a large polygon or a flat, axis aligned surfaced referred to as a "water-tight" object. The disadvantage to MLRT is that it requires a modified SAH function to produce good results and strongly depends on the type of geometry to obtain successful EP search results.

2.4 Faster BVH Traversal

BVH traversal can reap the same benefits as a kd-tree using packets and frustum culling as the ideas can be easily extended to BVH traversal. An algorithm that can be used to further improve BVH traversal is an early hit test [6]. This test can be used to quickly determine if a frustum of rays intersects a certain box. This can be accomplished using a single ray from the frustum, called an active ray, to test if it intersects the box in question. If so, the box's children are then traversed. If the early hit test fails, the rest of the rays can be tested against the box to determine if the frustum actually intersects the box which is a slow process since it can involve

intersecting all the rays in a frustum at many points during traversal. Rather than test all rays if the early hit test fails, a faster way would be to use frustum culling to determine if the frustum does not intersect the box. Using this method, the result of the frustum culling only determines if the frustum does not intersect the box but cannot determine if the frustum actually hits the box since it is a conservative test. This means that if the frustum culling test fails, the last resort would be to test every ray against the box. If this last resort test fails, then the box is not traversed, else, the box is traversed and the ray that hit the box becomes the new active ray for the frustum. The advantage of this algorithm is that it has a quick test for intersection using the early hit test, a quick test for non intersection using conservative frustum culling and a slower, more accurate fall back test. In practice, performance using this algorithm is very comparable to the fastest kd-tree traversal algorithms.

2.5 Dynamic Scenes of kd-Trees and BVHs

The difficulty with dynamic scenes is that ray tracing depends on a high quality AS to achieve high performance. With dynamic scenes, these structures must be either updated or rebuilt every frame. For kd-trees, any small change to the geometry can invalidate the whole tree because of the strict spatial paritioning. Scanning [7] and binning [8] algorithms have been used to approximate the SAH cost function through regular sampling of the cost for different split positions. This approach has led to an order of magnitude speed up for the rebuilding of moderately complex scenes but still at a non-interactive rate. Parallel approaches to kd-tree construction have been moderately successful as the difficulty lies in minimizing communication overhead and providing scalability [9]. Fuzzy kd-trees using motion decomposition [10] has been shown to provide interactive rates for scenes using predefined animations and skinned meshes.

Since BVHs are an object hierarchy rather than a spatial hierarchy, any slight changes to the scene will not invalidate the whole tree as the bounding boxes can just be refitted to produce a correct and updated tree very quickly. If the topology of the scene is known and never changes throughout the lifetime of the scene, only box refitting is necessary to produce a high quality tree. The original topology to be updated can be precomputed before rendering, or can come from the joint structure of a skinned mesh. For more general scenes, refitting can be done per frame but results in deterioration of tree quality which can be fixed through periodic rebuilding [11] or asychronous rebuilding [12]. To speed up the actual rebuilding process, a binning algorithm can be used to achieve a relatively high quality tree at a non-interactive rate.

2.6 Secondary Effects

Secondary rays due to reflection and refraction become incoherent after each bounce thus reduces the efficiency of packet and frustum traversal methods [13] for tertiary rays. Methods such as packet reordering using hardware scatter and gather operations for large SIMD widths have been proposed to increase efficiency when tracing groups of rays [14]. Also, n-way BVH structures have been investigated to increase the performance of mono ray traversal by using SIMD instructions to test a single ray against n BVH nodes in parallel. Such n-way structures such as a 4-way QBVH [15] and a 16-way BVH are targeted at hardware architectures with a large SIMD width such as Intel's Larrabee [16] or current generation graphics processing units (GPUs).

To achieve high quality soft shadows, approaches such as cone tracing [17], soft shadow volumes [18] and multi-sampled approaches achieve high quality shadows at non-interactive rates. Parker's [19] faked soft shadows using single sample soft shadows produce believable results with a single shadow sample but require a special ray/polygon intersection test and a lower quality BVH. In this thesis, a similar idea is presented in Chapter 5 which also requires a single shadow sample but requires only a change to the traversal logic with no required changes to any other part of the ray tracing system.

Chapter 3

Implementation of Scene Builder and Runtime Renderer

3.1 Test Application

The ray tracing application, called "RTTest", was implemented fully in C and was developed on an Intel 3.2 GHz Pentium D, which has 2 cores and 2 MB L2 cache, with 1 GB of RAM. The goal of RTTest was to act as a prototyping platform for testing and benchmarking new traversal methods for moderately complex static scenes. RTTest has both offline and runtime components: The offline component is a kd-tree builder which takes in the scene geometry and builds an efficient kd-tree for fast runtime rendering; the runtime component is the actual renderer which uses the precomputed kd-tree to render the scene in real time.

The RTTest runtime has been designed to take advantage of Intel's Streaming SIMD Extensions (SSE) for calculating floating point operations of a vector width of 4 giving an effective 4× speed up for floating point operations. Throughout the ray tracing pipeline, from ray generation to traversal, to shading, SSE has been used to take full advantage of all 4 lanes in the floating point unit.

On a coarser level, the RTTest runtime is also a threaded application that spawns the same number of threads as the number of CPU cores available and is able to render on a subset of the scene independently from another with very little synchronization.

3.2 RTTest Offline

The following sections are related to the RTTest offline tool used for 3D scene importing and processing for the RTTest runtime. The offline tool supports the importing of scene data from AutoDesk's 3D Studio Max and id Software's Quake III Arena. Scene data comprise of geometry, texture coordinates and material properties used for rendering.

3.2.1 kd-Tree Building

The RTTest offline tool builds a kd-tree for the input scene geometry using a traditional Surface Area Heuristic (SAH) function as described earlier in Section 2. When implementing the kd-tree builder, run-time performance was not considered as a factor since it is used as an offline tool.

Since the kd-tree builder is a recursive algorithm and trees can reach depths of 30 or more, the code implementation of the builder is iterative while maintaining its own stack rather than relying on the CPU stack which is implicitly maintained through recursive function calls and can quickly overflow. Another implementation remark is that a maximum depth size is set to prevent the tree from becoming too deep which can lead to bad performance since traversal would take too long to get to the leaf nodes. Another reason for specifying a maximum depth is that it constrains the memory size of the tree structure since additional splitting would increase the number of internal nodes and primitives. Also, as the tree gets deeper, more duplicated primitives occur which can lead to leaf and primitive indices that overflow 32 bit values.

In the RTTest implementation, candidate split planes were taken as the edges of each polygon's clipped bounding box to the current node's bounding box. It is important to clip polygons before generating split plane candidates since this can lead to a more optimal SAH solution. By clipping the polygon to the node's bounding box before generating a bounding box for candidate split planes, a tighter bound to the polygon can be achieved and thus a more optimal SAH solution. Without clipping, the kd-tree builder failed to find any splitting node in some cases where polygons were not axis-aligned which led to large leaves with lots of polygons and bad rendering performance.

Since tree traversal is a key kernel operation, the memory layout of the tree nodes is important to maximize cache coherency. Modern CPUs tend to have 128 bytes per cacheline which should be able to fit multiple, contiguous nodes. To maximize cache usage, node data sizes should be minimized by using the minimal set of information necessary for node traversal. The following shows the memory layout for a kd-tree node used in the system which comprises of 14 bytes:

```
typedef struct {
    float split;
    BYTE axis;
    BYTE isLeaf;
    int left;
    int leaf;
} kd_node;
```

left is either the index of the left child node if the node is an internal node or the index of leaf data if it's a leaf. In RTTest, the left and right child nodes are allocated contiguously which means that if the index of the left child is x, the index

of the right child is $x+1$. The result of this is that another index is not required per node and generating the right index from the left during runtime requires only one addition and both children can be fetched together to a single cacheline. *split* is the splitting plane value for the node and *axis* is the axis of the split plane. For leaf nodes, *isLeaf* is 1 and for internal nodes it is 0. For leaf nodes, *leaf* is the number of polygons contained within the leaf and is unused for internal nodes. Since leaf nodes have no children, *left* is used instead as a pointer to extra leaf data in a *kd_leaf* structure as described in the following section. It is possible to compact the *kd_node* structure down to 8 bytes [20] requiring bitshifting to unpack the data during traversal. This was not used in RTTest.

3.2.2 Scene Data

Figure 3.1 shows a high level overview of the data structures and memory layout used by the renderer. The *kd_node* which was explained in the previous section represents the internal nodes and leaves of the kd-tree. In memory, all the kd-trees are allocated in an array of contiguous memory and reference each other via the *left* data member. However, for leaf nodes, *left* refers to an index into another block of contiguous memory which holds leaf data. Since each leaf node can contain a variable number of polygons, the size of each *kd_leaf* structure is variable where the *leaf* data member of the *kd_node* structure that's pointing to the leaf is used to determine the size at runtime. To maximize cache coherency, all *kd_leaf* structures are allocated in a block of contiguous memory. The *kd_leaf* structures contain the actual polygon data that is required for intersection and may be duplicated among different leaves since kd-tree leaves can share polygons. The reasoning for duplicating the polygon data among leaves is that there is a clear benefit in keeping polygon data close together in memory since each polygon is accessed sequentially during intersection and as the first polygon is fetched from memory so are others into the same cacheline. Storing unique polygon data in another memory area and then refering to them through another form of indirection would cause incoherent memory accesses and inferior performance. However, for larger scene sizes, duplicating the polygon data among the leaves would not be practical due to memory constraints. The remaining data member of the *kd_leaf* is *ext* which represents the bounding box of the leaf node and is required for packet culling and intersection tests.

The *poly* structure holds all the information that is required for an intersection test. The first bunch of parameters are used for the projected Barycentric coordinate intersection test. The remaining parameters are shading parameters for that specific polygon. *normals* are the vertex normals which are used for normal interpolation during shading. *material_id* is a reference to a *sh_info* structure that describes the material properties that are used to shade the polygon. Finally, *uvcoords* are texture coordinates for each vertex used during shading. For multitexturing, multiple UV data members can be used for each texture.

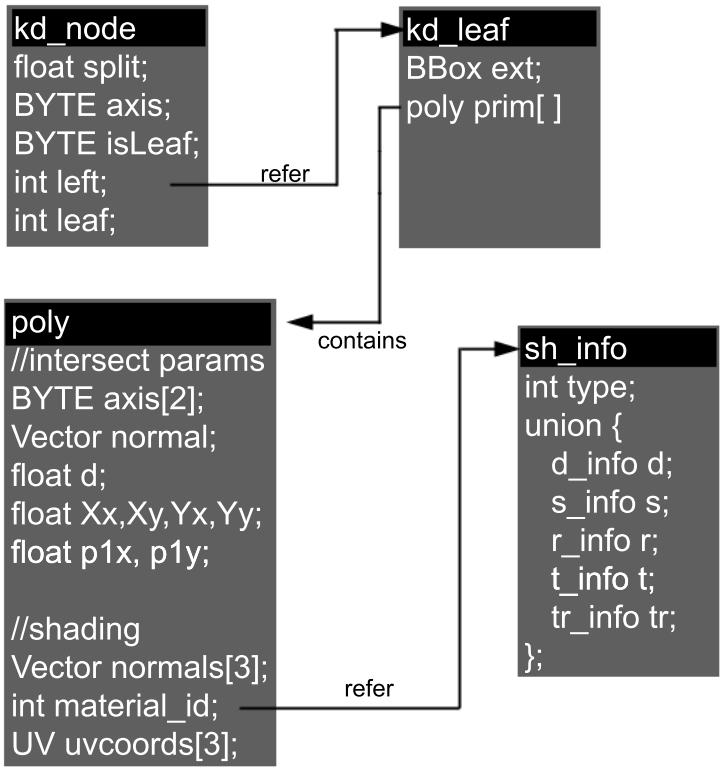

Figure 3.1: Memory map of the RTTest runtime renderer.

In retrospect, it would be possible to keep all the shading data in the *poly* structure in another memory location since it has nothing to do with intersection but would require another incoherent memory access during shading. However, as more shading data members are required for each polygon, it might be a viable solution but for now, the current design serves its purpose.

The *sh_info* data structures are kept in another contiguous memory location that contain data for different types of materials. These materials can be defined in a 3D editor such as 3D Studio Max and can be referenced by each polygon through its *material_id*. Further information on the shading system is discussed in the Shading section.

3.3 RTTest Runtime

Figure 3.2 shows a high level view of the rendering pipeline used in RTTest. For a multi-core machine, RTTest will spawn the same number of threads as number of cores and each thread will process a rendering task via the same pipeline. Each rendering task specifies a tile on the screen which must be rendered and processed in parallel on a multicore processor. At the beginning of the pipeline, the task is popped from a stack and the tile parameters are used to generate primary ray packets and bounding frustums. The frustums are then passed to the traversal system which traverses the internal nodes of the kd-tree in search for leaf nodes. Once a leaf node is hit, the internal ray packets within the frustum are culled against the leaf's bounding box. Any packets that are not culled are then forwarded to the intersector which performs polygon intersection tests against each packet. If there are any remaining packets left to be intersected, the frustum is further traversed in search of the next leaf node for intersection. When each packet in the frustum has been intersected, the intersection results of each packet are then passed to a proxy shader. The proxy shader is a general shader which operates on one ray packet at a time and is responsible for spawning secondary ray packets for reflection or refraction if required and simple fixed function shading operations such as parameterized diffuse lighting and specular highlights. These secondary traces are sent to the kd-tree for traversal, intersection and shading which do not further spawn additional rays. Once the resulting colours of the secondary traces have been calculated, the proxy shader computes the final colours which are then sent to the final stage for writing into the backbuffer.

3.4 RTTest Runtime Profiling

Table 3.1 shows profile statistics of the more processing intensive stages of the render pipeline for both primary and secondary rays when rendering the scene shown on the thesis cover page which features reflection and refraction secondary effects. The secondary ray traversal stage takes up the most cycles due to the fact that each

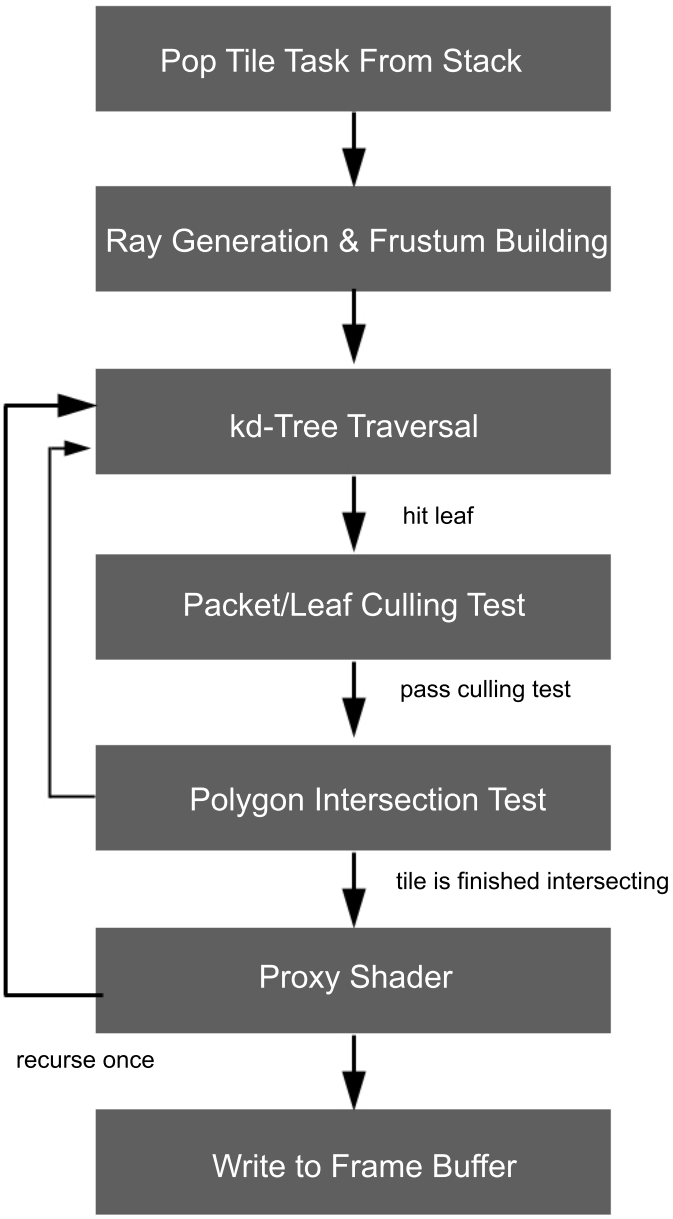

Figure 3.2: High level view of the rendering pipeline

ray packet (4 rays) is traced individually rather than together within a frustum (256 rays) since many of the secondary rays are incoherent. The primary intersection calculations take up slightly more cycles than the secondary ray intersections since there are less secondary rays traced overall. The primary ray shading is very computationally expensive since it is responsible for building the secondary ray packets based on different material types and handling the diffuse, specular, reflection and refraction colours to produce the final screen colour that has to be written to the back buffer.

Table 3.1: RTTest runtime profiling for the rendering pipeline

Stage	Processing %
Primary Ray Traversal	21.00%
Primary Ray Intersection	13.00%
Primary Ray Shading	12.90%
Secondary Ray Traversal	34.31%
Secondary Ray Intersection	12.00%
Secondary Ray Shading	5.65%

Chapter 4

Algorithms and Techniques for Ray Tracing

This chapter documents the algorithms, techniques and ideas used for the implementation of a high performance ray tracer. Several new algorithms are introduced that simplify and accelerate current techniques. Figure 4.1 shows the scenes used for benchmarking these ray tracing core algorithms. The Quake III scene is a custom map made for id Software's Quake III Arena first person shooter game which has a moderate polygon count and larger polygons with even distribution. Transformed kitchen has a high polygon count and non-axis aligned geometry. Sponza has a moderate polygon count with complex geometry and good variation of large and small polygons. Finally, Fairy is a high polygon scene with many little details.

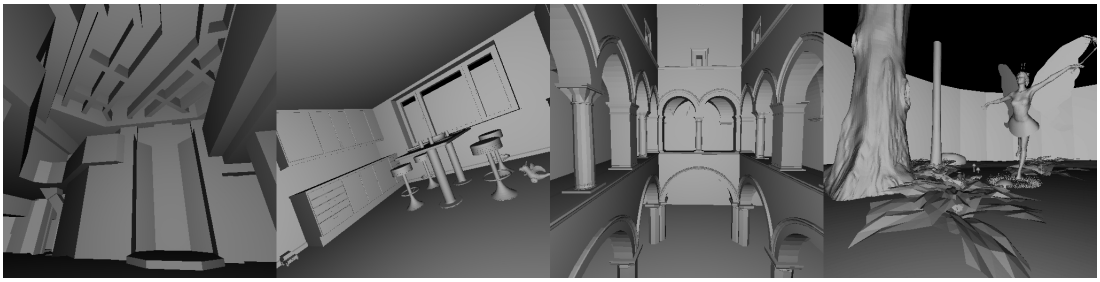

Figure 4.1: Benchmark scenes from left to right: Quake III (12.8k polys), Transformed Kitchen (110k polys), Sponza Attrium (67k polys), Fairy (174k polys)

All tests were done on an Intel Pentium D processor (2 Cores) at a 512 $\times$ 512 resolution from the views shown in Figure 4.1 with a single point light with attentuation and simple shading.

4.1 Traversal

The following sections describe and analyze some traversal techniques for high performance ray tracing. Traversal is the most important optimization target since it's responsible for the actual tracing of rays and the minimizing of redundant, expensive intersection tests.

4.1.1 Goals of Fast Traversal

The following are the goals for fast traversal:

1. minimize internal node visiting,
2. maximize cache coherency,
3. minimize memory usage,
4. minimize redundant intersection tests, and
5. minimize traversal operations in the traversal loop.

From the renderer's perspective, internal nodes are all overhead since they just help get at the leaf nodes where the actual geometry lies. Internal node visits can be minimized via tracing coherent rays simultaneously and finding a good entry point into the tree. The tracing of coherent rays simultaneously by using packets or frustums can decrease the number of internal nodes visited since common paths along the tree are not revisited.

To maximize cache coherency, the working set of data must be kept to a minimum. By doing proper data layout so data is accessed in sequence and by minimizing the size of data structures, cache coherency can be increased. Also, operating on a subset of the problem such as rendering tiles can increase cache coherency by picking tile sizes in which the working set of data fit in the cache. Care must be done when accessing large 2D arrays since they are implemented as row major 1D arrays in C. This means that accessing elements adjacent in the same column are not adjacent in memory which can lead to an incoherent memory access for large row sizes.

Traversal is a memory intensive operation as new nodes are accessed in the traversal's inner loop which are usually incoherent in memory since the children nodes are not guaranteed to be adjacent in memory to the parent node. To reduce these accesses, packet and frustum traversal help by allowing rays to be traced simultaneously, thus reducing the number of accesses to the same nodes along a common traversal branch. As rays become less coherent, however, simultaneous ray tracing becomes ineffective since there are no longer common branches that are visited amongst a group of rays.

The most effective optimization on a traversal algorithm is to reduce the number of traversal steps. A secondary optimization would be to reduce the actual number of traversal operations actually performed in the inner traversal loop. By using SSE instructions and good old fashion code optimization, the number of traversal operations can be reduced.

4.1.2 Omni-Directional Packets

The traditional packet traversal algorithm requires that all rays must have the same direction signs per axis as described in the Background chapter. Because of this restriction, any bundle which does not abide will have to be split into smaller packets and traced individually leading to more traversal steps and more complex code due to fall back code paths. A modification to the traditional kd-tree traversal algorithm is proposed to remove this restriction to allow for a simpler code implementation that requires no direction sign maintenance or packet splitting fall back code paths.

By using the following two modifications to the traditional kd-tree, ray packets can be traversed with no direction sign limitations:

1. Independent of ray directions, always trace from $front$ voxel to the *back* voxel which are defined as the voxel on the same and opposite side of the split plane as the camera, respectively

2. Treat negative intersection distances as large (infinite) positive distances

The following is an example of a 2×2 bundle traversal code which has been modified to handle rays with no direction restriction:

Listing 4.1: Omni-Directional Packet Traversal

```
while ( !node.isLeaf ) {
    active[i] = ( t_near[i] < t_far[i] );

    if (for all i=0..3(!active[i]))
        break;

    dist = split − origin[axis];

    d[i] = dist / dir[i][axis];

    //modification 1
    for all i=0..3
        d[i] = (d[i] < 0 ? FLT_MAX : d[i]);

    //modification 2
    int node_index = ( dist < 0.0f ) ? 1 : 0;
    Node *front = ( Node * ) ( node.left + ( node_index ^ 0x1 ) );
    Node *back = ( Node * ) ( node.left + node_index );

    stack.push( back, max( d[i], t_near[i] ), t_far[i] );
    ( node, t_far[i] ) = ( front, min( d[i], t_far[i] ) );
}
```

The first modification compensates for negative distances, d, by setting negative distances to FLT_MAX. This requires an extra conditional move per ray. The second modification is shown by selecting the *front* and *back* variables based on the *dist* variable independent of ray directions.

The same two modifications can be made to the frustum, interval traversal algorithm by applying the negative distance compensation to the extremal (corner) rays of the frustum and selecting *front* and *back* voxels based on ray origin only. Listing 4.2 shows a modified version of the interval traversal algorithm by [21] which allows frustums to be traced with no direction sign restrictions.

Listing 4.2: Frustum Interval Traversal Algorithm with Omni-Directional Modification

```
while ( !ISLEAF( node ) )
{
    const float split = node->split;
    const unsigned int k = DIMENSION( node );
    adr += OFFSET( node );
    const float dist = split - ray4.origin;

    // Modification #1: Pick front and back based on origin
    int node_index = ( dist < 0.0f ) ? 1 : 0;
    KDTreeNode *front = ( KDTreeNode * ) ( adr + ( node_index ^ 0x1 ) );
    KDTreeNode *back = ( KDTreeNode * ) ( adr + node_index );

    const sse_t d = _mm_mul_ps( _mm_set_ps1( dist ), oneOverDir4.t[k] );

    const sse_t cmp = _mm_cmplt_ps(d, _mm_setzero_ps());

    // Modification #2: Compensate for infinite case
    d = _mm_or_ps(_mm_and_ps(cmp, _mm_set_ps1(MAX_TRACE_LENGTH)),
        _mm_andnot_ps(cmp, d));

    const float dMin = _mm_cvtss_f32( sseHorizontalMin( d ) );
    const float dMax = _mm_cvtss_f32( sseHorizontalMax( d ) );

    node = back;

    if ( dMax < near )
        continue;

    node = front;

    if ( dMin > far )
        continue;

    stack[stackIndex] = back;
    stack[stackIndex].near = MAX( dMin, near );
    stack[stackIndex].far = far;
    stackIndex++;

    far = MIN( dMax, far );
}
```

For primary rays, there's a 2× performance increase for omni-directional ray packets over standard ray packets when the packets have differing direction signs. This is the case because standard ray packets must be split into two on average and traced individually which requires two times more processing than omni-directional

ray packets. However, the overall performance benefit is minimal since these types of packets represent only a small fraction of the primary ray packets for any given view: 4/1024 for a 1024×1024 resolution [13].

The practical advantages to using this modification is that direction signs no longer have to be checked per bundle which can be helpful for a simpler software implementation or a fixed function hardware implementation. In RTTest, omni-directional ray packets are used for the packet traversal of secondary rays since it's simpler to implement in the Shader Proxy when spawning secondary traces.

4.1.3 Frustums

For primary rays, a frustum interval traversal is used to provide a speed up over standard packet traversal as shown in Table 4.1 where *FPS* is the frames per second, N is the average number of traversed node per packet and I is the average number of intersections per packet. As seen in the performance data, frustums incur a higher number of intersection tests. The reason for this is that all packets in a frustum are traversed and intersected together, resulting in leaf nodes that are visited which only intersect a fraction of the packets within a frustum. This means that some packets within the frustum visit leaf nodes that they wouldn't have intersected if traversed alone. Despite more redundant intersection tests, frustum approaches yield a better overall performance due to the fact that they traverse much less nodes. The reason behind this is that frustums support larger groups of rays than 2×2 packets which allows highly coherent primary rays to be traced together and thus visiting nodes along the common path within the tree only once. In Table 4.1, peak frame rates are shown in bold where most scenes peak at a 8×8 tile size with the exception of the Quake scene which peaks at 16×16 due to the simplicity of the scene which allows for a larger group of rays to follow a common path through the tree during traversal.

As described in the following sections, the use of a cone shaped frustum was also investigated and resulted in superior performance over packets for primary rays but were less optimal than standard pyramid shaped frustums used for the benchmarks in this section. Also, the idea of multiple frustums is also discussed in the following sections. This idea can be used to deliver improved performance over standard frustum traversal for primary rays.

For secondary rays, frustums were not used since as rays become more incoherent, frustums will incur too many redundant intersection tests and there are less common paths in the tree to exploit. For frustums bounding incoherent rays, many nodes can be intersected and traversed while only a small fraction of internal rays are actually active since a frustum approach does not use any information about the internal rays during traversal. For primary rays, frustums incur redundant traversal and intersection to a lesser extent due to coherency but packets should still be culled against leaf bounding boxes prior to intersection tests as shown in the next section to prevent redundant intersection computation.

Table 4.1: Frame rate (FPS), # traversed Nodes (N) and # intersection tests (I) for 2×2 packet tracing and increasing tile sizes for primary rays.

Packet 2×2	FPS	N	I
Quake	15.00 fps	38.68	3.46
Kitchen	9.00 fps	46.50	11.12
Sponza	13.60 fps	33.81	3.83
Fairy	7.80 fps	54.62	11.78
Frustum 4×4	FPS	N	I
Quake	22.00 fps	10.38	4.39
Kitchen	13.00 fps	13.94	30.04
Sponza	17.50 fps	11.05	9.86
Fairy	11.30 fps	15.74	39.21
Frustum 8×8	FPS	N	I
Quake	37.00 fps	2.90	5.83
Kitchen	**16.90 fps**	4.92	77.74
Sponza	**24.00 fps**	4.17	22.20
Fairy	**14.80 fps**	5.78	100.16
Frustum 16×16	FPS	N	I
Quake	**44.00 fps**	0.911	9.90
Kitchen	14.00 fps	2.38	269.60
Sponza	20.60 fps	2.14	63.71
Fairy	12.40 fps	3.06	347.40
Frustum 32×32	FPS	N	I
Quake	37.00 fps	0.35	22.77
Kitchen	6.80 fps	1.60	1081.16
Sponza	9.90 fps	1.52	226.81
Fairy	6.80 fps	2.10	1156.30

4.1.4 Packet/Leaf Culling

A packet/leaf culling algorithm similar to a packet BVH traversal was used to filter out any redundant intersection tests due to frustums. Table 4.2 shows the frame rate improvement due to packet/leaf culling and the percentage of packets that are culled for a tile size of 8×8. The numbers shown in Table 4.1 in the previous section are performed with packet/leaf culling turned on and are used to generate the percentage of frame rate change and percentage of culled packets. From the table, the importance of a fast culling algorithm is evident. Especially for scenes with a high polygon count such as Kitchen and Fairy, the performance benefit is around $2\times$ solely due to an efficient culling test. As tile sizes and scene complexity increase, it becomes more important to perform a simple packet/leaf culling algorithm for high performance rendering.

The Multiple Frustum Traversal section introduces the use of multiple frustums to help further decrease these redundant intersection operations.

Table 4.2: Overall performance change from packet culling and the percentage of packets culled per frame.

8x8	Framerate increase %	Culled Intersections %
Quake	15.60%	40.35%
Kitchen	182.00%	85.60%
Sponza	76.50%	82.40%
Fairy	208.00%	86.60%

4.1.5 Multiple Frustum Traversal

In this section, a novel traversal technique is proposed which allows better rendering performance and uses ideas similar to beam tracing [22] for the purpose of ray acceleration. Since using an interval frustum algorithm does not require SSE for traversal, it is possible to traverse a packet of frustums, not rays, together using SSE instructions much like beams. This can be done by splitting a tile into 4 smaller subtiles as shown on the left in Figure 4.2. The advantage to doing this would be to enable quick masking of inactive or "completed" (intersected) frustums to minimize the number of traversed nodes with comparable computations to tracing a 2×2 packet. By using quick masking, the overall "tile" that is traced will shrink when frustums become inactive or completed, thus allowing more efficiency with larger tiles. This idea of using multiple frustums instead of a single frustum is very similar to the idea of using packet tracing over mono-ray tracing by taking advantage of the underlying SSE architecture.

Multiple Frustum Traversal (MFT) can be implemented by keeping track of near (t_near) and far (t_far) values for each frustum. For SSE enabled systems

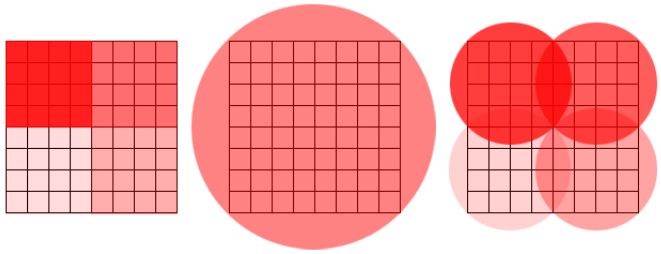

Figure 4.2: left: MFT used for 4 pyramidal frustums, centre: single cone frustum bounding a tile, right: MFT used for 4 conical frustums

with four wide vector units, four frustums can be traced in parallel while keeping traversal data within CPU registers. A frustum can be masked if it is inactive ($t\_far < t\_near$) OR if it is complete. By keeping track of a simple frustum mask, many branches can be culled during traversal since less inactive rays are "forced" down branches that they wouldn't have visited if traversed alone. Also, when a frustum is masked, it will not be involved in the decision making at each split plane. The following list describes the essential changes to the traditional, kd-tree frustum interval traversal for MFT:

1. Treat each frustum as a ray with an interval (t_near, t_far) using frustum interval traversal.
2. Maintain an active mask for each frustum: active = (t_near $\leq$ t_far) AND there exist packets inside frustum are not intersected.
3. Using SSE, traverse 4 frustums simultaneously while updating intervals (like packet tracing).
4. When pushing a node on the stack, push the active mask with it.
5. If all frustums are inactive, stop traversal into current node.
6. When a leaf is visited, perform intersection tests using packets bounded by active frustums only.
7. If all rays in the frustum has intersected, make frustum inactive.

Figure 4.3 shows how intersections and traversal can be culled by tracing 2 frustums simultaneously, instead of one larger frustum or tracing 1 and 2 individually. By tracing one large frustum, node A is visited only once as well as nodes B and C. The advantage to tracing one large frustum is that all nodes are visited only once. However, the disadvantage is that not all rays within the frustum actually intersect with nodes B and C resulting in redundant intersection tests. Another disadvantage is that if all the rays that intersect node B or C have already hit a

polygon in node A, then traversal would still continue into nodes B and C until all the rays within the frustum have been intersected.

If the large frustum is split into frustum 1 and 2, and traced indvidually, nodes A and B would be visited twice and node C once. The advantage to this method would be that frustum 1 would never intersect node C, thus decreasing the number of redundant intersection tests over the single frustum approach. However, the disadvantage to this method is that nodes A and B are traversed twice even though they lie within the common path of the larger frustum, thus leading to more traversal operations.

Using MFT, frustums 1 and 2 can be together using SSE such that node A, B and C are visited only once like the single frustum approach. Note that this example features only 2 frustums but on a SSE enabled system, 4 frustums are traced simultaneously using MFT. The advantage of MFT over the single frustum approach is that when traversing node C, frustum 1 is masked out such that no intersection tests are performed for rays within frustum 1 against any geometry within node C. Also, if any two frustums have completely intersected geometry in node A, they will be masked out during any further traversal into nodes B and C. Note that MFT never performs more traversal or intersection operations over a single frustum approach or an approach that traces frustums individually. The only overhead of using MFT is using SSE to maintain 4 intervals and a frustum mask.

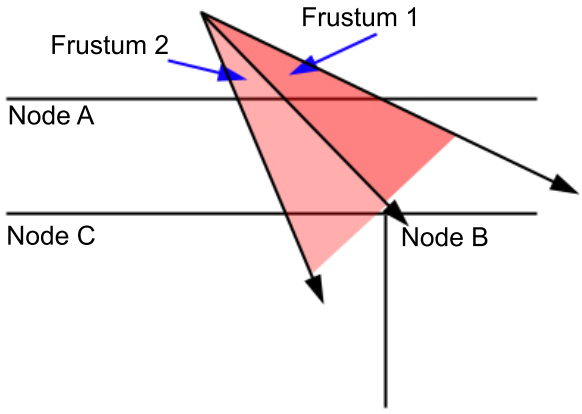

Figure 4.3: Diagram of MFT with 2 frustums being traced hitting nodes A, B and C.

Table 4.3 shows statistics of benchmark scenes using MFT. The columns labeled % *FPS*, % *N*, and % *I* show the percentage difference of MFT compared to the single frustum traversal for the same tile sizes as shown in Table 4.1. For each test case, MFT shows a improvement of a higher frame rate, less nodes traversed and less intersection tests performed. When comparing the percentages of improvement,

there is a significant difference in the number of intersection tests solely due to frustum masking. Another important metric is the peak performance comparison for MFT versus single frustum peak performance shown in Table 4.4. The statistics show that for every test case, MFT yields the highest peak performance at around 12% over the single frustum peak performance. Table 4.4 also shows that MFT achieves peak performance at a larger tile size. The reason for this is that at higher tile sizes, MFT allows all frustums to be active in areas of the scene where there is low geometric complexity and masks out inactive frustums in areas of high geometric complexity as shown in Figure 4.3 where both frustums are active in node A and frustum 1 is masked in Node C. By doing masking, fewer redundant intersection tests occur in areas of high complexity and the speed benefit of a large furstum occurs in areas of low complexity.

MFT has been successfully implemented in RTTest and it offers the highest performing code path in the renderer. Due to the ease of implementation and the increase in performance in all test scenes, MFT is a very useful technique for improving kd-tree traversal. It can also improve the XP search stage of the MLRT algorithm since it also uses an interval frustum traversal.

Table 4.3: Statistics for using MFT. Each tile is divided into four quadrants and traced as four simultaneous frustums

MFT 4×4	FPS	N	I	% FPS	% N	% I
Quake	24.00 fps	10.34	3.79	9.00%	-0.40%	-13.70%
Kitchen	14.30 fps	13.57	16.57	10.00%	-2.65%	-44.84%
Sponza	18.80 fps	10.74	5.74	7.40%	-2.80%	-41.80%
Fairy	12.30 fps	15.34	21.25	8.85%	-2.50%	-45.80%
MFT 8×8	FPS	N	I	% FPS	% N	% I
Quake	39.30 fps	2.87	4.39	6.20%	-1.00%	-24.70%
Kitchen	18.00 fps	4.60	30.71	6.50%	-6.50%	-60.50%
Sponza	26.00 fps	3.92	9.96	8.30%	-6.00%	-55.14%
Fairy	16.10 fps	5.40	39.67	8.78%	-6.57%	-60.39%
MFT 16×16	FPS	N	I	% FPS	% N	% I
Quake	49.00 fps	0.88	5.87	11.36%	-3.40%	-40.70%
Kitchen	**18.50 fps**	2.24	83.56	32.14%	-5.88%	-69.00%
Sponza	**27.50 fps**	1.93	22.86	33.50%	-9.81%	-64.11%
Fairy	**16.90 fps**	2.670	104.41	36.29%	-12.75%	-69.95%
MFT 32×32	FPS	N	I	% FPS	% N	% I
Quake	**50.00 fps**	0.33	10.10	35.14%	-5.71%	-55.64%
Kitchen	12.50 fps	1.56	328.20	83.82%	-2.50%	-69.64%
Sponza	19.50 fps	1.31	68.98	96.97%	-13.82%	-69.59%
Fairy	12.30 fps	1.93	371.24	80.88%	-8.10%	-67.90%

Table 4.4: Peak performance comparison between a single frustum or packet versus MFT

Scene	MFT framerate	MFT tile size	non-MFT framerate	non-MFT tile size
Quake	**50.00 fps**	32×32	44.00 fps	16×16
Kitchen	**18.50 fps**	16×16	16.90 fps	8×8
Sponza	**27.50 fps**	16×16	24.00 fps	8×8
Fairy	**16.90 fps**	16×16	14.80 fps	8×8

BVH Multiple Frustum Traversal

MFT could also apply to BVH frustum traversal proposed by Wald [6] by having four frustums with four different active rays from each frustum. The four active ray tests could be done in parallel using SSE as proposed by Wald [6] but the difference is that each ray should be from different frustums. The early miss frustum test should be done on four frustums using interval arithmetic instead of a single frustum using SSE. Using the results from the active ray and early miss frustum tests, a frustum mask can be maintained like in the MFT kd-tree implementation. Also, the frustum mask can also keep track of frustums that have been completely intersected. Using this frustum mask, only ray packets within active frustums need to perform and per ray operations such as the "test of last resort" [6] and polygon intersection tests. Also, during traversal, if all of the frustums are inactive, traversal into the current node should be stopped. To summarize, the following modifications to the traditional BVH frustum traversal should be done to support MFT:

1. Maintain 4 active rays from 4 different sub frustums.
2. Perform 4 active ray tests simultaneously using SSE; for tests that pass, update the frustum active mask.
3. If all tests pass: continue traversal into node.
4. If any of the 4 active ray tests fail, perform early miss frustum culling for all 4 frustums together using SSE; for tests that pass, update the frustum active mask.
5. If all early miss tests pass, stop traversing current node.
6. For all frustums that failed the early miss test, update the frustum's active ray and continue traversal.
7. For frustums that could not find a new active ray, make it inactive in the frustum mask; Otherwise, make it active.

8. If all frustums are inactive, stop traversing current node.
9. When pushing a node on the stack, push the active mask with it.
10. When a leaf is visited, perform intersection tests using packets bounded by active frustums only.
11. If all rays in the frustum have intersected, make the frustum inactive.

The BVH version of MFT has not been implemented in RTTest but predicted results should be similar to the kd-tree version as traversal and intersection operations are able to be further culled with little overhead. In the Future Work chapter, profile results are planned to be obtained for this flavor of MFT.

4.1.6 Cone Proxy Traversal

Cone proxies provide an alternative to a pyramidal proxy for ray traversal acceleration. The following sections derive a fast, interval method of a cone frustum traversal through a kd-tree hierarchy. This traversal algorithm can then be used as a proxy frustum to accelerate bundles of rays by exploiting coherency without any knowledge of the internal rays. This method can also be used to accelerate cone tracing [17] to render high-quality, secondary effects.

4.1.7 Cone Frustums

A cone is defined by a ray and an angle, α, of separation between between the ray and outer edge as shown in Figure 4.4. Unlike standard mono-ray traversal which calculates single distances to the split plane, cone traversal calculates two intersection distances to the split plane as shown in Figure 4.4 as t_1 and t_2. These two distances are defined as the distances along the ray of the cone in which the outer edges of the cone intersect the split plane at a single point. By using these two distances, an interval traversal algorithm can be derived which is done in the Cone Traversal section.

The following shows a derivation of t_1 and t_2. Figure 4.4 shows a cone with origin s, direction v and angle α. The cone is intersecting with a split plane with normal n, distance d at an angle ϕ. Using these definitions we want to derive an equation for the distances t_1 and t_2 along vector v from s in which the cone intersects the split plane at a point. The following equations represent the two distances of interest, t_1 and t_2, as t_{12}.

$$point\_along\_ray = s + t_{12}v \tag{4.1}$$

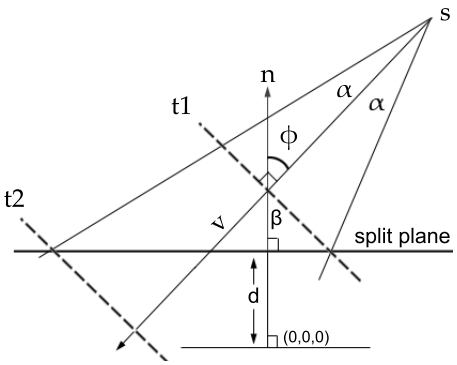

Figure 4.4: A cone with origin s, direction v and angle α intersecting a split plane with normal n, distance d at an angle ϕ.

solving for the distance from the plane gives:

$$distance\_from\_plane = (s + t_{12}v) \cdot n - d \tag{4.2}$$

using trigonometry gives another equation for the same distance:

$$distance\_from\_plane = t_{12} \tan \alpha \cos(\beta) \tag{4.3}$$

where $\beta = 90° - \phi$

equating the two distances

$$(s + t_{12}v) \cdot n - d = t_{12} \tan \alpha \cos(90° - \phi) \tag{4.4}$$

solving for ϕ by finding the angle between vectors n and v:

$$\phi = \cos^{-1}(\frac{n \cdot v}{|n||v|}) \tag{4.5}$$

solving for t_{12} gives the following:

$$t_{12} = \frac{d - n \cdot s}{n \cdot v - \tan\alpha \cos(\phi - 90^\circ)} \tag{4.6}$$

substituting for ϕ and solving for the cos term gives

$$t_{12} = \frac{d - n \cdot s}{n \cdot v \pm \tan\alpha \sqrt{1 - (\frac{n \cdot v}{|n||v|})^2}} \tag{4.7}$$

since the plane normal n can assumed to be normalized and if the ray v is also normalized, then the equation for $t12$ can be simplified to:

$$t_{12} = \frac{d - n \cdot s}{n \cdot v \pm \tan\alpha \sqrt{1 - (n \cdot v)^2}} \tag{4.8}$$

Because of the $\pm$, two values are obtained which correspond to the two points of intersection, t_1 and t_2, as shown in Figure 4.4.

4.1.8 Cone Traversal

Using Equation 4.8, a cone traversal algorithm is derived. For primary rays, $\tan\alpha$ for each cone can be precomputed offline before rendering since they are view direction independent. Everything in the denominator is cone dependent therefore they can be precomputed per frame for each cone before traversal when the frustum is built. During traversal, to obtain $t1$ and $t2$, the distance from the split plane, $d - n \cdot s$, must be calculated as usual and multiplied by the precomputed values per cone. The following code illustrates the traversal code which requires only one more multiply and two conditional moves over mono ray traversal:

Listing 4.3: cone frustum traversal code

```
//d − ns
split_minus_origin = split − origin[axis];

temp1 = split_minus_o ∗ precompute[0][axis];
temp2 = split_minus_o ∗ precompute[1][axis];

t1 = min(temp1, temp2);
t2 = max(temp1, temp2);
```

The precomputed values are calculated per frame for each cone as shown in the following where *dir* is the vector v in Figure 4.4:

Listing 4.4: cone frustum precomputed values

```
for ( int axis = 0; axis < 3; axis++ )
{
    beta = tanAlpha * sqrt ( 1.0f - dir[axis] * dir[axis] );
    precompute[0][axis] = 1 / ( beta + dir[axis] );
    precompute[1][axis] = 1 / ( -beta + dir[axis] );
}
```

Using these calculations for solving for distances $t1$ and $t2$, the traversal algorithm can be derived where $t1 <= t2$ is always true.

The following shows the traversal algorithm for cones where negative distance compensation is used to allow for omni-directional traversal as described in the Omni-directional Ray Packets section:

Listing 4.5: full cone frustum traversal code

```
while ( !node.IsLeaf ) {
    split_minus_o = split - origin[axis];
    temp1 = split_minus_o * precompute[0][axis];
    temp2 = split_minus_o * precompute[1][axis];

    //negative distance compensation
    temp1 = (temp1 < 0) ? FLT_MAX : temp1;
    temp2 = (temp2 < 0) ? FLT_MAX : temp2;

    t1 = min(temp1, temp2);
    t2 = max(temp1, temp2);

     //modification 2
    int node_index = ( dist < 0.0f ) ? 1 : 0;
    Node *front = ( KDTreeNode * ) ( node.left + ( node_index ^ 0x1 ) );
    Node *back = ( KDTreeNode * ) ( node.left + node_index );

    if ( t2 <= t_near )
        node = back;
    else if ( t1 >= t_far )
        node = front;
    else {
        stack.push( back, max( t1, t_near ), t_far );
        (node, t_far ) = ( front, min( t2, t_far ) ); }
}
```

The advantage of using cone traversal in terms of a pyramid traversal as described in Listing 4.2 is that no horizontal max/min functions are required. One can argue that it is possible to precompute the max/min directions before pyramid traversal but that still requires additional computation resulting in inferior performance in some cases. Another advantage over pyramids is that cones can be traversed omni-directionally requiring no SSE which is not the case for an omni-directional pyramid traversal since all four corner distances must be calculated first using SSE before applying a horizontal min/max operation as shown in Listing 4.2. For smaller tile sizes, cones provide better performance and no directional restrictions over pyramids as shown in Table 4.5 in the following section where a single cone beats out a single frustum for a 4×4 tile size even though it traverses more nodes.

4.1.9 Cone vs Pyramid

This section analyzes some profile results for the use of cones versus pyramids as a frustum for primary rays. Much like how MFT was used for pyramid frustums, MFT can be applied to cone frustums as well. An added benefit to MFT cones over intersection and traversal culling due to masking is that multiple cones bound the primary ray tiles tighter than a single cone as shown in Figure 4.2 where a single cone is shown in the middle and multiple cones on the right. Table 4.5 shows how single cone and MFT cone traversal compare with a single pyramid frustum traversal for the Sponza benchmark scene. The three leftmost columns specify the frame rate, average number of nodes traversed per packet and the number of intersections performed per packet, respectively. The three rightmost columns show the same statistics as a percentage of difference relative to a single frustum approach as summarized in Table 4.1.

The data shows that there is significant overhead to using a single cone in terms of the number of nodes visited and intersection tests over a pyramid frustum due to the fact that cones more loosely bound primary rays. It's worth noting that the 4×4 single cone beats out the 4×4 single frustum in framerate due to the fact that cone traversal doesn't require a min and max direction value to be calculated from the extremal rays before each traversal which leads to less computational overhead over the cone traversal which doesn't require such calculations. However, as tile sizes increase, this overhead gets amortized over more rays, thus allowing the pyramid frustum to overtake the cone frustum in terms of performance.

The use of MFT cones provides a better overall performance than the pyramid frustum due to the masking of frustums which leads to less intersection tests but slightly more traversed nodes. When compared to MFT pyramids as shown in Table 4.3, MFT cones are still inferior. Despite the fact that cones does not provide an overall performance benefit due to the fact that cones do not tightly bound primary rays, this section provides some good insight into the possibilities of using frustum acceleration of different shapes.

From the performance numbers, cones do not look like a practical frustum method for ray acceleration. However, for a beam tracer where rays are replaced with beams with actual volume, cones can provide better performance than a pyramid frustum due to the smaller number of traversal operations required with no direction limitations. The use of cones for beam tracing is beyond the scope of this project but it is an interesting approach noted for future work.

Table 4.5: Performance metrics for single cone and MFT cone traversal compared to a single pyramid frustum for Sponza.

4×4	FPS	N	I	% FPS	% N	% I
Single Cone	20.00 fps	12.10	13.32	**14.29%**	9.50%	35.09%
MFT Cone	18.80 fps	11.23	6.93	7.43%	1.63%	-29.72%
8×8	FPS	N	I	% FPS	% N	% I
Single Cone	22.20 fps	5.03	33.40	-7.50%	20.51%	50.47%
MFT Cone	25.00 fps	4.28	13.32	4.17%	2.73%	-40.00%
16×16	FPS	N	I	% FPS	% N	% I
Single Cone	15.50 fps	2.95	106.08	-24.76%	37.90%	66.50%
MFT Cone	25.10 fps	2.27	33.40	21.84%	6.00%	-47.57%
32×32	FPS	N	I	% FPS	% N	% I
Single Cone	6.40 fps	2.30	382.47	-35.35%	51.62%	68.63%
MFT Cone	15.50 fps	1.62	106.08	56.57%	6.63%	-53.23%

Despite providing poorer performance numbers for primary ray accerlation due to loose bounding, cones can provide better performance than pyramids for a group of rays which are cone-shaped. Cone-shaped ray groups are useful for doing multi-sampled soft shadows to spherical light sources as shown in Figure 4.5. Since they provide a tighter bound to internal rays, cone frustums yield a better overall performance than pyramid frustums as shown in Table 4.6.

Table 4.6: Performance metrics for a 16 sample soft shadow rendering using cone and pyramid frustums for a light radius of 1 and 2.

radius = 1	FPS	N	I
Cone	**2.72 fps**	118.28	116.10
Pyramid	2.00 fps	120.72	129.31
radius = 2	FPS	N	I
Cone	**1.97 fps**	164.30	378.91
Pyramid	1.36 fps	174.23	449.69

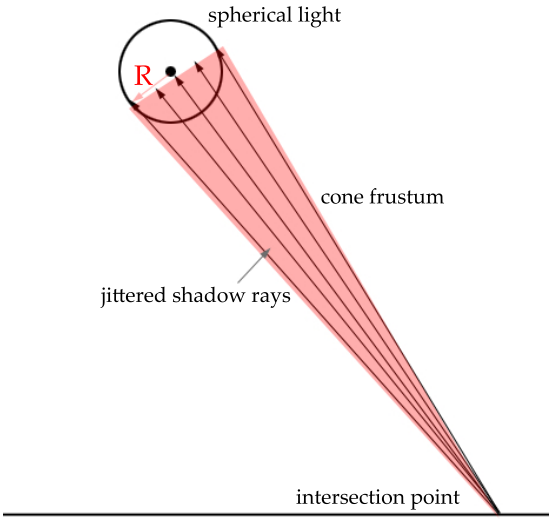

Figure 4.5: Diagram of multiple shadow samples to a spherical light using a cone frustum.

4.2 Ray/Polygon Intersection

For fast ray/polygon intersection, the projected Barycentric coordinates test [20] was implemented in RTTest. The advantage to using this test is that the coordinates must be calculated anyways for proper texture lookup during shading.

By using the projected Barycentric test, the vertices of the actual polygon do not have to be present at all which further saves on memory and achieves better cache coherency. The following listing shows the structure *render_poly* that was used for representing polygons for intersection. All the data members are precomputed parameters for the Barycentric test which amounts to 42 bytes.

```
typedef struct
{
    BYTE axis[2];
    Vector normal;
    float d, Xx, Xy, Yx, Yy, p1x, p1y;
} render_poly;
```

In RTTest, all intersection tests are performed for four (SIMD_WIDTH) rays simultaneously using SSE. The result of an intersection test is kept in a *trace_rt* structure which is shown in the following listing:

Listing 4.6: trace result structure

```
typedef struct
{
    float t[SIMD_WIDTH];
    float pid[SIMD_WIDTH];

    //barycentric coords
    float a1[SIMD_WIDTH];
    float a2[SIMD_WIDTH];
    float a3[SIMD_WIDTH];
    bool isCompleteHit; //all hits same poly
} trace_rt;
```

The data members are arrays of SIMD_WIDTH since they are results for each ray. *isCompleteHit* is true if all rays in the packet intersect the same polygon. This is helpful to know during shading since packets no longer have to be broken and can be quickly shaded in groups of SIMD_WIDTH. The t data member is the distance along each ray in which the intersection occurs. These values can be used with the ray directions to find the actual point of intersection during shading. The *pid* data member contains the id numbers of the polygon intersected for each ray. When *isCompleteHit* is true, all elements in *pid* are the same. *a1*, *a2* and *a3* are the resulting Barycentric coordinates which are used for interpolation along the polygon between the vertices during shading.

After intersection, these results are passed to the Proxy Shader which is responsible for using these results to produce the final colour output for the packet of rays as discussed in the following section.

4.2.1 Shader System

The shader system used in RTTest is fixed function which means that the system exposes a set of parameters which can be used to customize certain material attributes that can be set through a scene editor such as 3D Studio Max. To support a fully programmable shading model as in the current generation of GPUs, shaders must written in C and compiled/linked to the runtime code. For a research application, this fixed function design can be adequate enough to prototype new shaders but for a production application, requiring the developer to recompile and link the application each time a new shader is changed is not acceptable. An alternative to this would be to support a runtime compiler or interpreter such as LLVM [23] to support a specific shading language to allow on the fly changes to shaders without any rebuilding of the application or any need for an offline compiler. Such an implementation is beyond the scope of this project but remains a future feature that should be implemented to make the RTTest rendering engine usable by developers for production applications.

Exposing every possible configurable shading parameter for maximum configurability would be detrimental to performance since there would be too many conditionals and branches within the performance critical shader which is executed at least once for every pixel. To maximize the number of different effects that can be rendered, a set of shading parameters has been chosen and exposed from the shading API to the materials. The following listing documents the different shader parameters exposed to the materials:

Listing 4.7: Shader and material types and parameters

```
typedef enum {
    M_D, //DIFFUSE
    M_S, //DIFFUSE + SPECULAR
    M_R, //DIFFUSE + SPECULAR + REFLECTIVE
    M_T, //DIFFUSE + TRANSPARENT
    M_TR, //DIFFUSE + SPECULAR + TRANSPARENT + REFLECTIVE
    NUM_MATERIAL_TYPES
}SH_TYPE;

typedef struct {  //DIFFUSE
    float base[4];
    texture *diffMap;
    texture *normMap;
} d_info;

typedef struct {  //SPECULAR
    d_info diffuse;
    int exp;
    float blend;
} s_info;

typedef struct {  //DIFFUSE + SPECULAR + REFLECTIVE
    s_info specular;
    float blend;
} r_info;

typedef struct { //DIFFUSE + TRANSPARENT
    d_info diffuse;
    float blend;
    float refract_index;
} t_info;

typedef struct {  //DIFFUSE + SPECULAR + TRANSPARENT + REFLECTIVE
    r_info reflective;
    float blend;
    float refract_index;
} tr_info;
```

```
typedef struct {
    SH_TYPE type;
    union {
        d_info d;
        s_info s;
        r_info r;
        t_info t;
        tr_info tr;
    };
} material;
```

The *SH_TYPE* type defines the different shaders that can bound to a material. These are specified as the *type* data member in the *material* structure. The *M_D* type is used for diffuse materials that are not reflective or refractive and is paramerterized by the *d_info* structure. In *d_info*, a base floating point colour can be specified but it is ignored if there is a diffuse texture map specified through the *diffMap* texture pointer. A normal map can also be optionally specified through the *normMap* texture pointer for higher quality rendering. Since all the shader types support a base colour with normal mapping, they each contain diffuse parameters which appear at the top bytes of the structure. This memory layout helps during the shading stage since it can be assumed that the first 24 bytes of each material contains diffuse shading information without knowing the actual shader type. The next type, *M_S* is used for shiny diffuse materials with specular highlights. The *exp* integer specifies the Phong highlighting exponent and while the *blend* float specifies the intensity of the specular highlights. The next type is *M_R* which has parameters *r_info* for reflective surfaces. Since reflective surfaces are usually specular and can have a base colour, the *r_info* structure contains an *s_info* and a reflective intensity factor *blend*. The difference between a specular and a reflective shader is that a reflective shader actually sends out secondary rays for sampling whereas a specular does not. Another type that requires secondary refraction rays is the transparent type, *M_T* which is parameterized by *t_info*. The transparent type supports a diffuse base, a transparency factor *blend* and an index of refraction *refract_index*. Finally, the most processing intensive shader is the transparent-reflective type, *M_TR*. This type supports both secondary reflection and refraction rays which means that each shaded pixel requires more than twice the traversal costs over a diffuse shaded pixel. Since the type is reflective, *tr_info* contains a *r_info* and transparency parameters *blend* and *refract_index*. Since a material can be of only one shader type, the *material* structure contains a union of all types.

All the fixed function shading is handled in the "Shader Proxy" stage of the rendering pipeline as shown in Figure 3.2. This proxy is responsible for taking in the trace results of each ray as specified by Listing 4.6 and producing a resulting colour. Using the *trace_rt* structure, the polygons are referenced from the *pid* data member for each ray in the packet. For each polygon intersected, the vertex normals are loaded into registers and interpolated at each hitpoint using the Barycentric

coordinates from the *trace_rt* structure. Next, the *material_id* for each intersected polygon, as shown in Figure 3.1, is used to reference a *material* structure for shading. The Shader Proxy uses these *material* structures and interpolated normals to compute the following:

1. build and trace secondary reflection and refraction packets,
2. build and trace shadow rays,
3. texture lookups for diffuse and normal maps using bilinear filtering,
4. compute diffuse term: $N \cdot L$ using interpolated normals or normal map,
5. compute Phong specular term: $(R \cdot V)^{\alpha}$, and
6. using reflection, refraction, diffuse, specular, ambient and shadow results, produce the final colour.

Once the final colours have been determined, they are sent to the colour writing stage which writes the final colour to the framebuffer.

4.2.2 Code Level Optimization

Writing high performance software such as a ray tracer can be a difficult task requiring some code level optimizations to best utilize the CPU hardware. This section provides an overview of some techniques to optimize code.

Proper data layout is very important for maximizing cache coherency via locality. By keeping data structures as small as possible and data accesses in sequence, cache utilization can be maximized. One such example of a structure to optimize are tree nodes which can be optimized down to 8 bytes (14 bytes were used in RTTest). By storing all nodes in an array of contiguous memory instead of heap allocated memory for each node, memory access patterns will be coherent and local.

The use of SSE intrinsics is essential for achieving high performance rendering. SSE allows full utilization of all 4 lanes of the floating point unit which effectively gives a 4× speed up for all floating point calculations. When using SSE in performance critical areas such as the traversal loop, branching should be replaced by masking to allow better thoroughput and reduce inefficiencies due to branch misprediction. One of the most effective ways of utilizing SSE is to work on 4 rays/pixels at a time. Coding in terms of groups of 4 can be difficult for some programmers as another dimension is introduced throughout the code. To help with the transition to working with packets of rays, a simple math library can be used for common functions that work on packets of rays much like a single ray. Listing 4.8 shows some common functions that can be used to handle ray packets. The BLEND macro can act as a helper for the common masking operation of selecting values from either

SIMD values, x or y. The *DotProduct_mm* computes 4 dot products simultaneously between 4 vectors in array *v1* and 4 vectors in array *v2*. The *inv_mm* function computes a fast reciprocol by using an initial approximation given by *_mm_rcp_ps* and performing one iteration of Newton's Method for more precision. This is a common method in graphics to perform fast reciprocols without a costly division. The final function is used for subtracting 4 vectors in array b from 4 vectors in array a and returning the result as 4 vectors in array *out*. These helper functions are far from complete but show how common functions using SSE can be wrapped to allow for better code readability and easier coding.

Listing 4.8: SSE ray packet helpers

```
#define BLEND(mask, x,y) _mm_or_ps(_mm_and_ps(mask,x), _mm_andnot_ps(mask, y))

inline __m128 DotProduct_mm(__m128 v1[3], __m128 v2[3])
{
    __m128 x,y,z;

    x = _mm_mul_ps(v1[0], v2[0]);
    y = _mm_mul_ps(v1[1], v2[1]);
    z = _mm_mul_ps(v1[2], v2[2]);

    return _mm_add_ps(_mm_add_ps(x,y), z);
}

inline __m128 inv_mm(__m128 in)
{
    __m128 estimate = _mm_rcp_ps(in);
     __m128 diff = _mm_sub_ps(_mm_set_ps1(1.0f), _mm_mul_ps( estimate, in));
    return _mm_add_ps(_mm_mul_ps(diff, estimate), estimate);
}

inline void VectorSubtract_mm(__m128 out[3], __m128 a[3], __m128 b[3])
{
    for(int i=0; i<3; i++)
        out[i] = _mm_sub_ps(a[i], b[i]);
}
```

In all cases of the rendering loop, the number of divisions and square roots should be minimized since they are costly floating point operations. The most common use is during vector normalization which requires a square root to find the length of the vector and a division to normalize it. When building rays for tracing, it is not necessary to normalize the rays before traversal since all calculated split plane distances will be correctly ordered. However, vector normalization is usually required in shading operations but should be used sparingly.

A common operation in many numerical processing applications is casting from

a floating point value to an integer. In a renderer such as a ray tracer, all internal values are calculated in floating point, but colour values are usually stored in the frame buffer as RGB integer values. Since over a million pixels are processed every frame at HD resolutions, the float to int casting is a target for optimization. Using a traditional float to int cast "(int)" is a very expensive operation because the ISO standard C compiler will have to change the floating point mode from rounding to truncate before casting which effectively flushes the floating point pipeline [24] [25]. By using the function in Listing 4.9, casting can occur using rounding without changing the floating point mode. SSE supported hardware exposes a fast float to int conversion using both trunction and rounding as *_mm_cvttps_epi32* and *_mm_cvtps_epi32* which can be used instead of the code in Listing 4.9 [24] for a group of values.

Listing 4.9: Fast float to int cast using rounding [24]

```
__inline long int
lrintf (float flt)
{ int intgr;

    _asm
    { fld flt
        fistp intgr
    } ;

    return intgr ;
}
```

Chapter 5

Edge Width Soft Shadows

Soft shadows are traditionally rendered by using multiple shadow ray samples per intersection point to an area light as shown in Figure 5.1 where multiple shadow rays are traced to the area light source. The results of these shadow ray traces are then used to determine the factor of occlusion. For a high quality, smooth soft shadow, at least 32 shadow ray samples per point are required. This is effectively 32 times more processing over primary rays. In Figure 5.2, multiple sponza renderings are shown ranging from 4 to 32 shadow samples per point. From renderings, significant banding can be seen for shadows with 4 to 8 samples which is already very performance intensive and not feasible for a real time application on current CPU hardware. For larger light sources which produce a larger pneumbra, even more samples must be taken to produce visibly smooth results.

This section proposes a novel technique for rendering fake soft shadows with smooth pneumbra edges called "Edge Width Soft Shadows" using only one shadow sample only. For most views, this method can achieve slightly less performance than tracing a single, traditional shadow ray which produces hard shadows. The visible advantage by using edge width shadows can be seen in Figure 5.3. A 32 multi-sampled soft shadow, an edge width soft shadow and a hard shadow Sponza rendering are shown in the top, middle and bottom pictures, respectively. From the comparison, the edge width soft shadows look much more realistic than the hard shadows and comparable to the multi-sampled shadows.

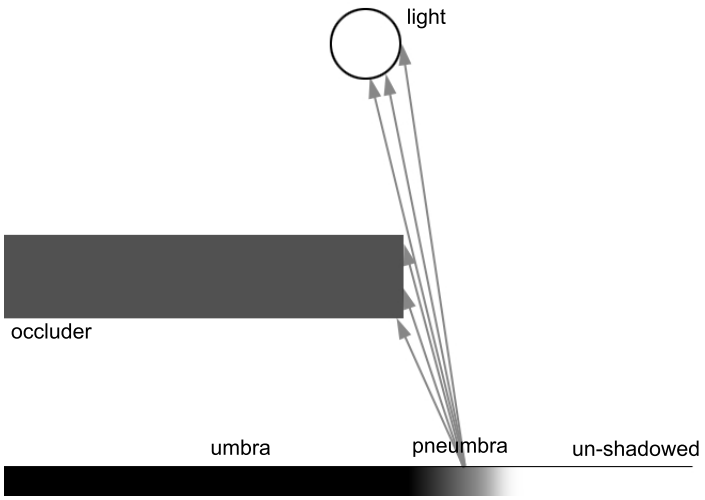

Figure 5.1: Diagram of an area light casting a soft shadow due to an occluder with umbra, pneumbra and un-shadowed regions

Algorithm

The key to soft shadowing algorithms using a single shadow sample is to use a fast method to detect if a point is either in the following regions: un-shadowed (lit), pneumbra or umbra. When the point is in the un-shadowed region, there is no occluder between the point and light source which results in the light's full intensity, referred to as a magnitude of 1.0. When in the pneumbra region, the light is partially occluded which results in an intensity factor from the light between 0.0 and 1.0. When in the umbra region, the point is fully occluded and has an intensity factor of 0.0 from the light. Figure 5.4 shows a diagram of an area light casting a soft shadow on a ground surface where the umbra, pneumbra and un-shadowed regions are labeled. Points in the umbra region are unlit while there's a smooth, gradient transition from dark to light in the pneumbra region leading to the un-shadowed region. Traditional soft shadow techniques use multiple shadow rays to sample the area light to calculate what percentage of the point is occluded as shown in Figure 5.1. For points within the umbra, all shadow rays intersect an occluder before reaching the light source whereas points in the un-shadowed region cast shadow rays which all reach the light source. For points in the pneumbra, some shadow rays hit an occluder and some reach the light source which leads to a partially shadowed region.

This detection of a pneumbra region is what differentiates a soft shadow multi-sampled approach from a hard shadow single sampled approach. The goal of the edge width shadowing method is to be able to approximate the location of the pneumbra with a single sample shadow ray. The difficulty in doing this is that the information returned from a single shadow sample is binary: either it's occluded or lit. From this binary result, there is no way of detecting if a point is actually fully

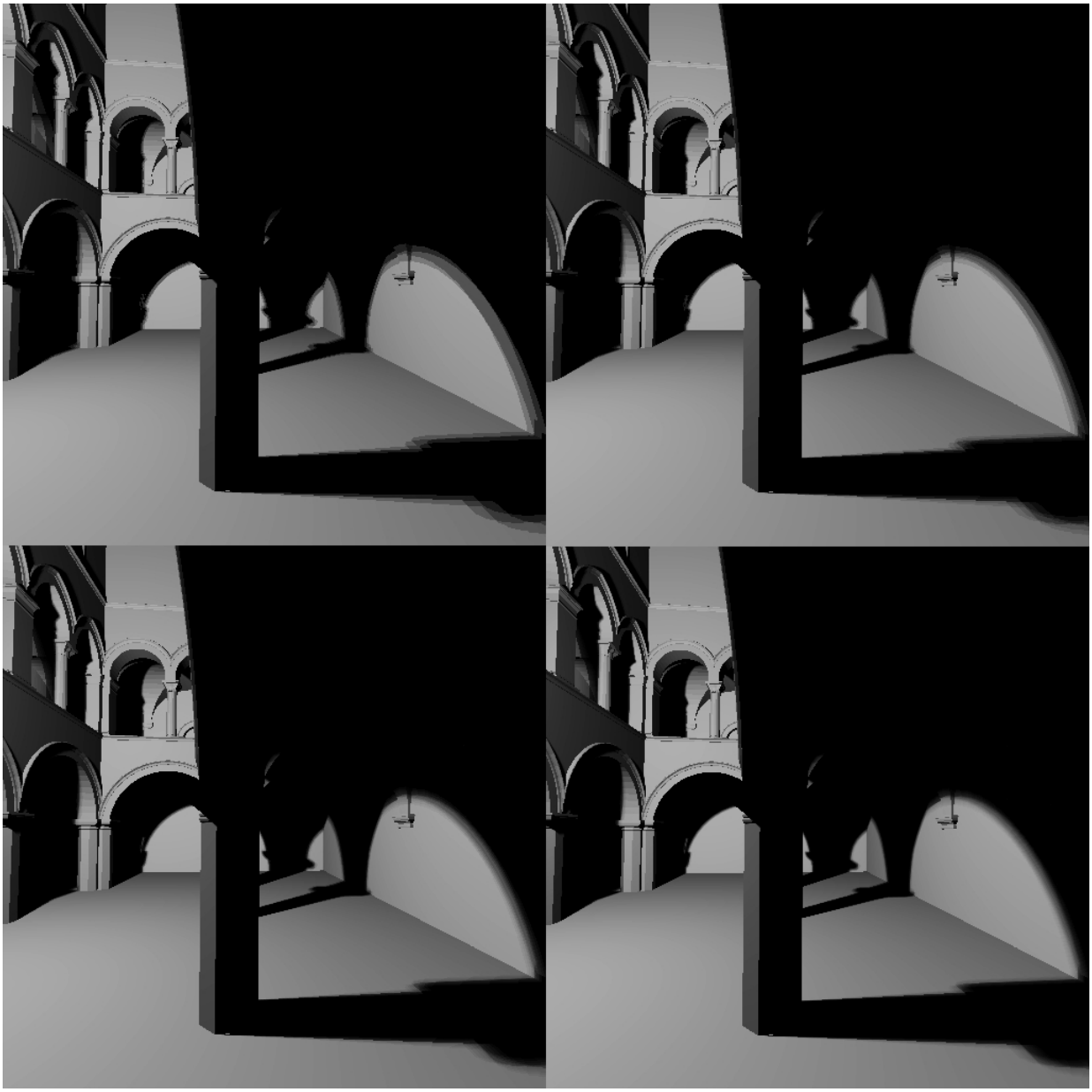

Figure 5.2: Sponza scene rendered with multisampled soft shadows; topleft: 4 samples, topright: 8, bottomleft: 16, bottomright: 32

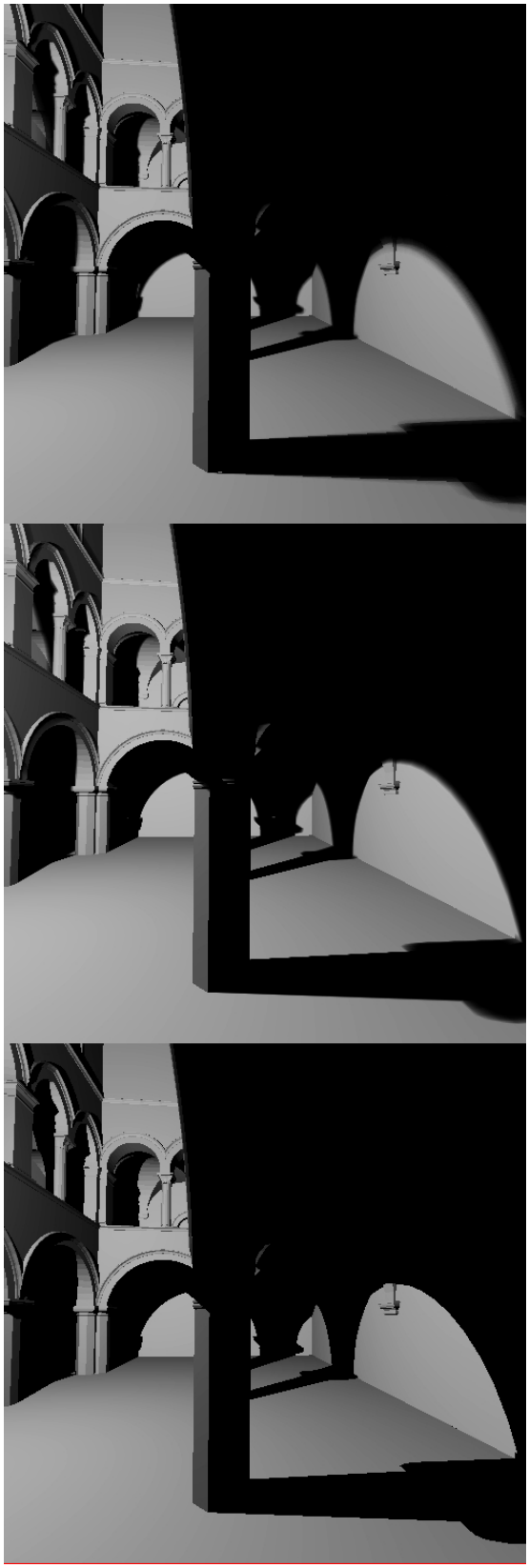

Figure 5.3: Sponza scene rendered with soft shadows; top: multisampled soft shadows (32); middle: edge width soft shadows, bottom: single sample hard shadows

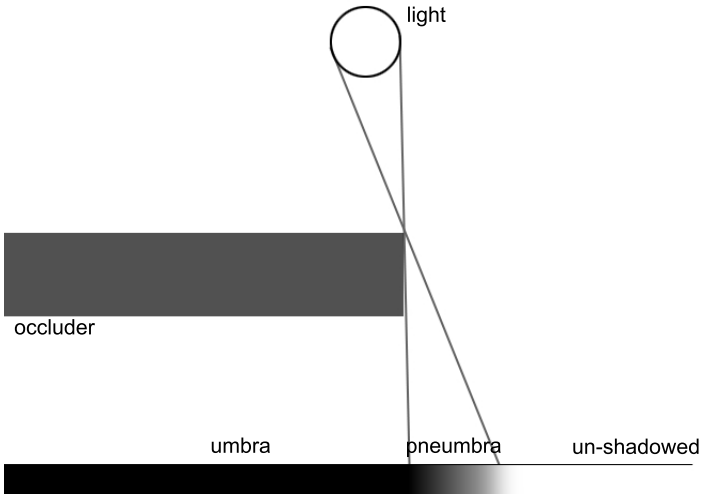

Figure 5.4: Diagram of an area light casting a soft shadow due to an occluder with umbra, pneumbra and un-shadowed regions

or partially lit. This leads to the definition of an "edge width" which is defined as the distance between the entry and exit point of a ray which intersects a closed polygonal object as shown in Figure 5.5. Rather than calculating a binary result using a single sample, edge widths can be calculated and used to better approximate soft shadows using a single sample.

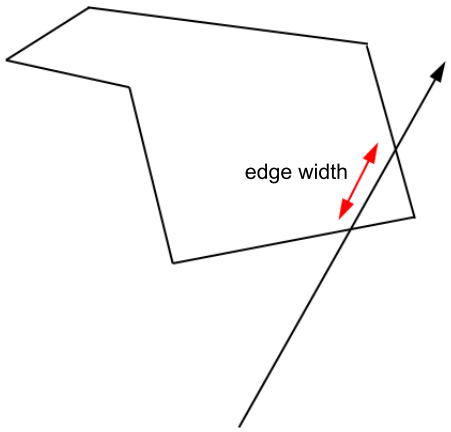

Figure 5.5: Diagram of an edge width

In Figure 5.5, as the ray moves to the right and closer to the edge of the object, the edge width becomes smaller and eventually 0 at the tangent. By using edge width value, occluder edges can be detected which lead to an approximate pneumbra region as shown in Figure 5.6. In the figure, edge width values are used to compare to a fixed value R which represents a number proportional to the radius of the

light source. Any distances between 0 and R yield an interpolated value between 0 and 1 resulting in a partially shadowed pneumbra region where a value of 0 is un-shadowed and a value of 1 is full occluded. Simple linear interpolation was used as shown in Equation 5.1 but to calculate the occlusion factor, the following is used:

$$occlusion\_factor = \begin{cases} 0 & \text{for } edge\_width \leq 0 \\ \frac{edge\_width}{R} & \text{for } 0 \leq edge\_width \leq R \\ 1 & \text{for } edge\_width \geq R \end{cases} \tag{5.1}$$

By comparing Figure 5.4 using edge widths to Figure 5.6 using multi-sampling, it is clear that the pneumbra regions are different in width and location.

The larger the value of R, the larger the pneumbra region is which can be correlated to the radius of the light source.

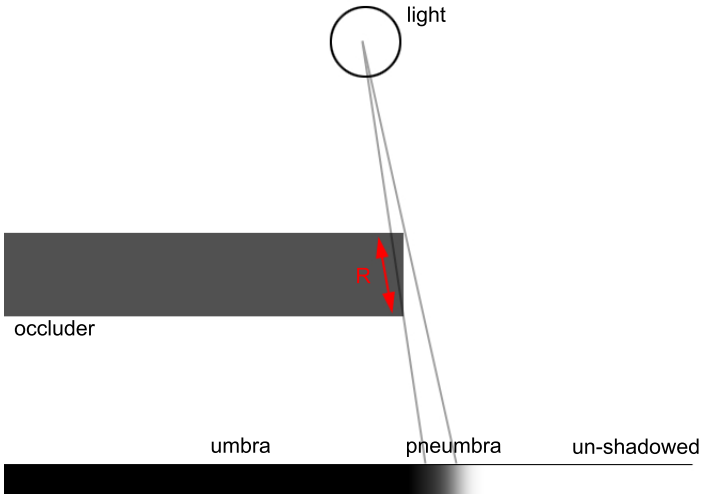

Figure 5.6: Diagram of fake soft shadows using a maximum edge width value of R.

At a glance, the overall idea using edge widths is simple: trace a single shadow ray to the light and calculate the edge width to determine the factor of occlusion. However, in practice, the implementation is a bit more involved as polygons are not visited in correct front to back order within leaf nodes, and special cases such as shadows cast ontop of shadows have to be handled explicitly.

Detailed Algorithm + Pseudocode

Standard shadow rays are either traced from the light to the point or vice versa until an occluder is hit or the length of the ray has been traced returning a binary result of either occluded or not occluded. Edge width shadow rays are traversed in the same way but handle leaf nodes differently and have different terminating criteria. When an edge width ray is traversing through the scene, the polygons must

be visited in strict front the back ordering. This allows the algorithm to determine if the ray is entering or leaving an object as shown in Figure 5.5. This gives a ray two states: inside or outside an object. Every edge width ray is assumed to start outside an object and toggles state each time it intersects a polygon along its path. Using this assumption, only correct soft shadows can be rendered if all geometry are enclosed polygons with no "holes" or "leaks" such as a polygon floating in the air with no actual volume. If part of the scene contains malformed geometry that does not abide by these restrictions, the shadows occluded by geometry will simply not be soft shadowed but hard shadowed.

Using the two ray states, edge widths are calculated based on the distance between the entry polygon and exit polygon with the maximum of all edge widths along the path being the final edge width result returned. The ray traversal is terminated when either the ray has finished tracing from the light to the point or that an edge width was calculated that is larger than the fixed value proportional to the light radius, R. The special case when a ray finishes traversal to the end and has an "inside object" state returns an infinite edge width which means the point is fully occluded. This special case makes sure that malformed geometry will still be hard shadowed.

Traversal through a kd-tree can assume to visit leaves in strict front to back order. Within leaves, polygons must be sorted in front to back order to allow for proper edge width calculation. Rather than sort the polygons, the intersection distances are calculated for each intersected polygon and put in a list. This list of intersected distances is then sorted using any sorting algorithm from smallest to largest. Since leaves usually contain very few polygons for a moderately complex scene, a simple bubble sort works quite well.

Figure 5.7 shows a diagram of a ray traversing through leaves A, B and C while intersecting with labeled polygons p1 to p6. For each polygon, the normal vector is drawn to better show which side the polygon is facing where the front facing side is in the positive normal direction. From the figure, tracing an edge width shadow ray while maintaining proper inside/outside state can be tricky since there is no assumed order within leaves and entry and exit polygons might be in different leaves. This problem can be solved by keeping track of unpaired entry polygons with no exit polygons. At any state of the ray traversal, only two possibilities can occur: either there's an outstanding entry polygon with no matching exit polygon or all pairs are matched. For example, as the ray in Figure 5.7 finishes visiting leaf A, p1 is an outstanding polygon with no exit polygon to pair with. When the ray reaches leaf B, p1 pairs with p2 and an edge width is calculated. After leaf B is finished being visited, p3 is an outstanding entry polygon. Once in leaf C, p3 is paired with p4 and p5 is paired with p6. At the end of traversal, the maximum edge width from all pairs is returned.

Listing 5.1 shows pseudocode for the edge width shadow traversal algorithm. The function returns a floating point value between 0 and 1 which is the fractional distance along the shadow ray where the shadow ray is the unnormalized delta

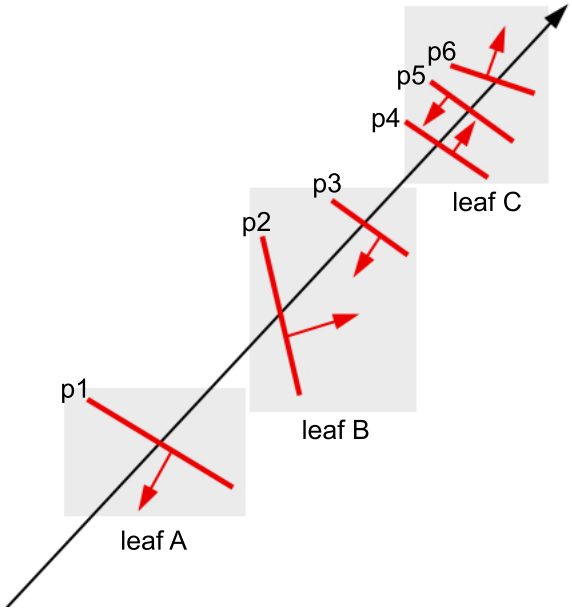

Figure 5.7: Diagram of ray traversal through a scene hitting various leaves and polygons.

vector between the light and point of interest. The *radius* is not the actual radius of the light but a value that represents the maximum fractional distance along the shadow ray that an edge width can be before the point is determined to be fully occluded. The *radius* value should be proportional to the light radius and can be used to control the width of the pneumbra. The pseudocode in Listing 5.1 is shown for a single shadow ray rather than a shadow ray packet as implemented in RTTest. Through proof of concept, the algorithm can be efficiently implemented using SSE for shadowing packets at a time.

The *GetEdgeWidth* function is very similar to any normal monotracing traversal routine except for the way leaves are handled. *maxEdgeWidth* is the maximum edge width calculated so far and is initialized to 0. *distances* is scratch space for the intersection distances calculated within a leaf and used for sorting. The first element of the array is reserved for any outstanding distances to an entry polygon that has not yet been paired. Since there are no distances stored before the traversal starts, the first element of *distances* is initialized to 1.0f meaning the maximum distance or infinite.

Once a non-empty leaf is hit, all the polygons within the leaf are intersected with no backface or front face culling since both entry and exit polygons are of interest. For each intersected distance within the interval range of $tNear \leq t \leq tFar$ it is added to the *distances* list. The next step is to sort the intersection distances from smallest to largest and determine what state the ray is in. The state can be determined by the first element of *distances*. If the first element is not 1, then there exists an entry polygon that was unpaired and the ray is put in an "inside"

state. The next step is to iterate through the remaining distances and calculate edge widths only on pairs where the first distance is from an entry polygon and the next is from an exit polygon while keeping track of the maximum edge width. After all the distances have been handled, the state is checked for an outstanding entry polygon and *distances[0]* is set for the next leaf. If the maximum edge width is greater than the input value *radius*, the traversal exits early since the point is determined to be fully occluded. If there is no early exit and the ray finishes traversing to the point, either the maximum edge width is returned or 1.0f if there is an outstanding entry polygon and the ray is in an "inside" state.

Listing 5.1: Edge width shadow tracing pseudocode

```
float GetEdgeWidth(ray shadowRay, float radius)
{
    float maxEdgeWidth = 0.0f;
    float distances[MAX_POLYS_PER_LEAF];

    distances[0] = 1.0f;
    int distCount = 0;

    while(!stack.isEmpty){
        ... pop from stack, setup for traversal

        while ( !node.IsLeaf ) {
        ... any standard traversal from the light to the point
        }

        //enters if a leaf node with actual polygons
        if(node.IsLeaf && node.PolyCount){
            bool hit = false;
            distCount = 0;

            for(int i=0; i<node.PolyCount; i++){
                float t = intersect_poly(node.polys[i], shadowRay);

                if(t >= tNear && t <= tFar){
                    distances[distCount + 1] = t;
                    hit = true;
                    distCount++;
                }
            }

            if(hit) {
                Sort(&distances[1], distCount);

                bool isInside = (distances[0] == 1.0f) ? false : true;

                for(int i=1; i< (distCount + 1); i++){
                    isInside = !isInside;

                    if(isInside)
                        continue;

                    float width = distances[i] - distances[i-1];
                    maxEdgeWidth = max(difference, maxEdgeWidth);
                }

                distances[0] = (isInside ? distances[distCount - 1] : 1.0f);
```

```
                //early exit
                if(maxEdgeWidth > radius)
                        return 1.0f;
            }
        }
    }
    //return 1.0 if there's an outstanding entry poly
    return ((distances[0] == 1.0f) ? maxEdgeWidth : 1.0f);
}
```

The advantage of the implementation of the algorithm is that it requires no change to the AS, geometry or intersection routines. The full implementation can reside in the shadow ray leaf handling routine. Also, it is not necessary to reference any other polygon data except for the essential data used for intersections. No vertices, normals or edges have to be referenced or calculated. This results in a simple implementation. However, there are several disadvantages to this method that lead to incorrect shadowing artifacts that are discussed in the following sections.

Light Leaks

The most obvious artifact to edge width shadows are light leaks as shown in Figure 5.8 where the R value is larger than the thickness of the occluder which leads to no umbra. This artifact can be seen in Figure 5.9 where a thin plate leaks light and thus has no umbra within the shadow. There are several ways that light leaking can be minimized but no solution that solves every case. What is common amongst all methods is that extra information is required to weed out cases where light can possibly leak through thin occluders. One simple way is to allow the artist or scene designer to specify that thin occluders be treated differently during edge width shadow ray traversal. This can be done through adding extra data to the polygons such that edge width values calculated from these polygons are larger than what is actually calculated via a scaling factor.

Another way of handling light leaks without any artist intervention can be to only allow edge width values to be calculated from entry and exit polygons that share at least one vertex, otherwise the edge width value is infinite. By doing this, it forces edge width values to be calculated from polygons that have a high probability of sharing an actual edge which weeds out cases of light leaking through thin objects with entry and exit polygons that share no common vertices. For large polygons, this method works very well but can lead to slight artifacts within the pneumbra for scenes with very small polygons since restricting edge width calculations to only polygons sharing a vertex will not allow rendering of larger pneumbra widths. Figure 5.10 shows a diagram of a red ray hitting an entry and exit polygon. The two polygons can be considered for edge width calculations since they share a common vertex. On the other hand, the blue ray yields an infinite edge width since the entry and exit polygons do not abide by the common vertex restriction.

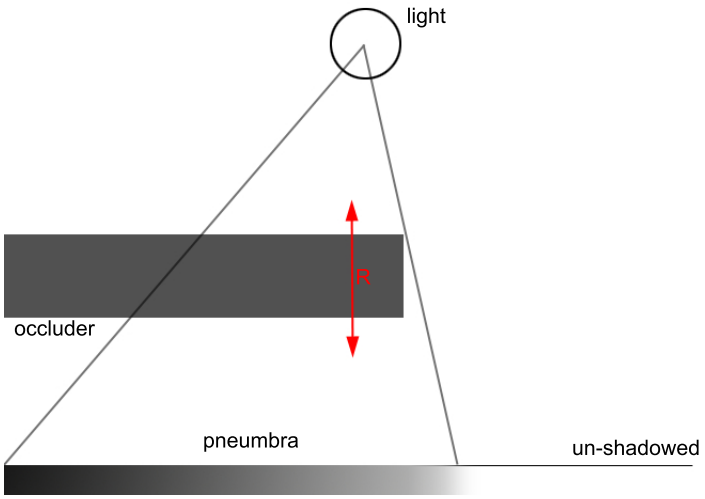

Figure 5.8: Diagram of light leaking through an occluder where there is no umbra

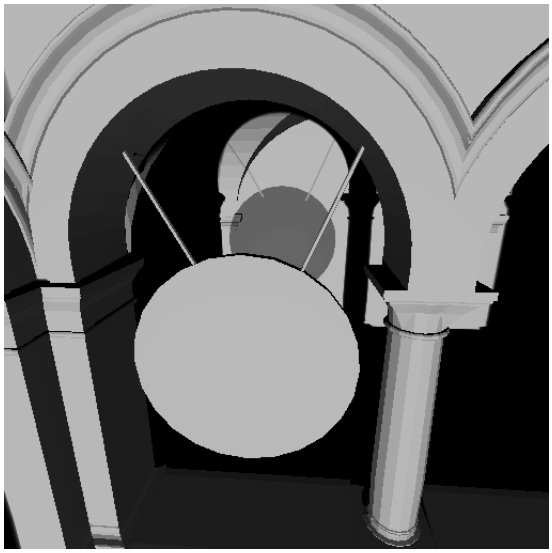

Figure 5.9: Rendering of light leaking through a thin plate.

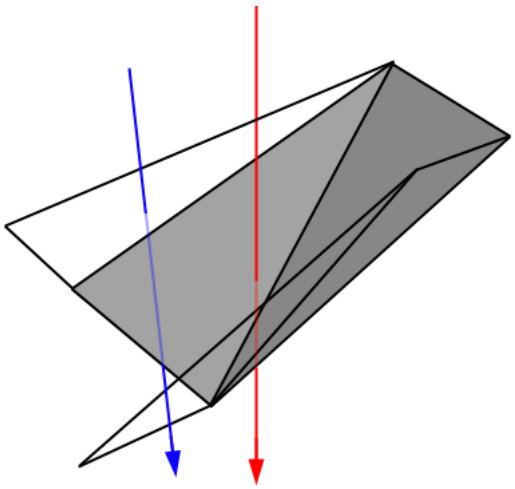

Figure 5.10: Polygons that share a vertex can be considered as entry and exit polygons.

Other Artifacts

Besides light leaks, edge width shadows can lead to other artifacts such as no pneumbra when lights cast shadows directly inline with the edge as shown in Figure 5.6 when the light moves directly above the edge of the occluder. One can argue that such a case is rare and barely noticible in a dynamic scene such as a game application. As mentioned earlier, malformed geometry consist of objects that are not fully enclosed by polygons such as a curtain with no thickness. These types of geometry will end up being rendered with hard shadows instead of soft shadows, thus resulting in a noticable artifact that does not deter much from the overall scene.

Performance

Figure 5.11 shows some soft shadow rendering using edge widths and no light leak fixing. The results are pretty believable and are a definite improvement over hard shadows. Table 5.1 compares overall performance results from the Sponza rendering shown in Figure 5.3 using single sample, edge width, 4 sample and 32 sample shadowing. The 4 sample approach results in significant banding and a very low performance with a third of the performance of the edge width method. Using 32 samples creates smooth believable shadows at an interactive rate of just over 1 frame per second. It should be noted that the 32 sample rendering was using secondary cone frustums for acceleration resulting in better performance than multiple shadow ray packets. The edge width shadow method performs at a very fast speed compared to the multi sampled approach and comparable to the single sample hard shadow performance. Despite significant artifacts, an edge width approach provides a good

improvement in terms of visual fidelity over hard shadows with moderate overhead which is practical for a high performance application such as a game.

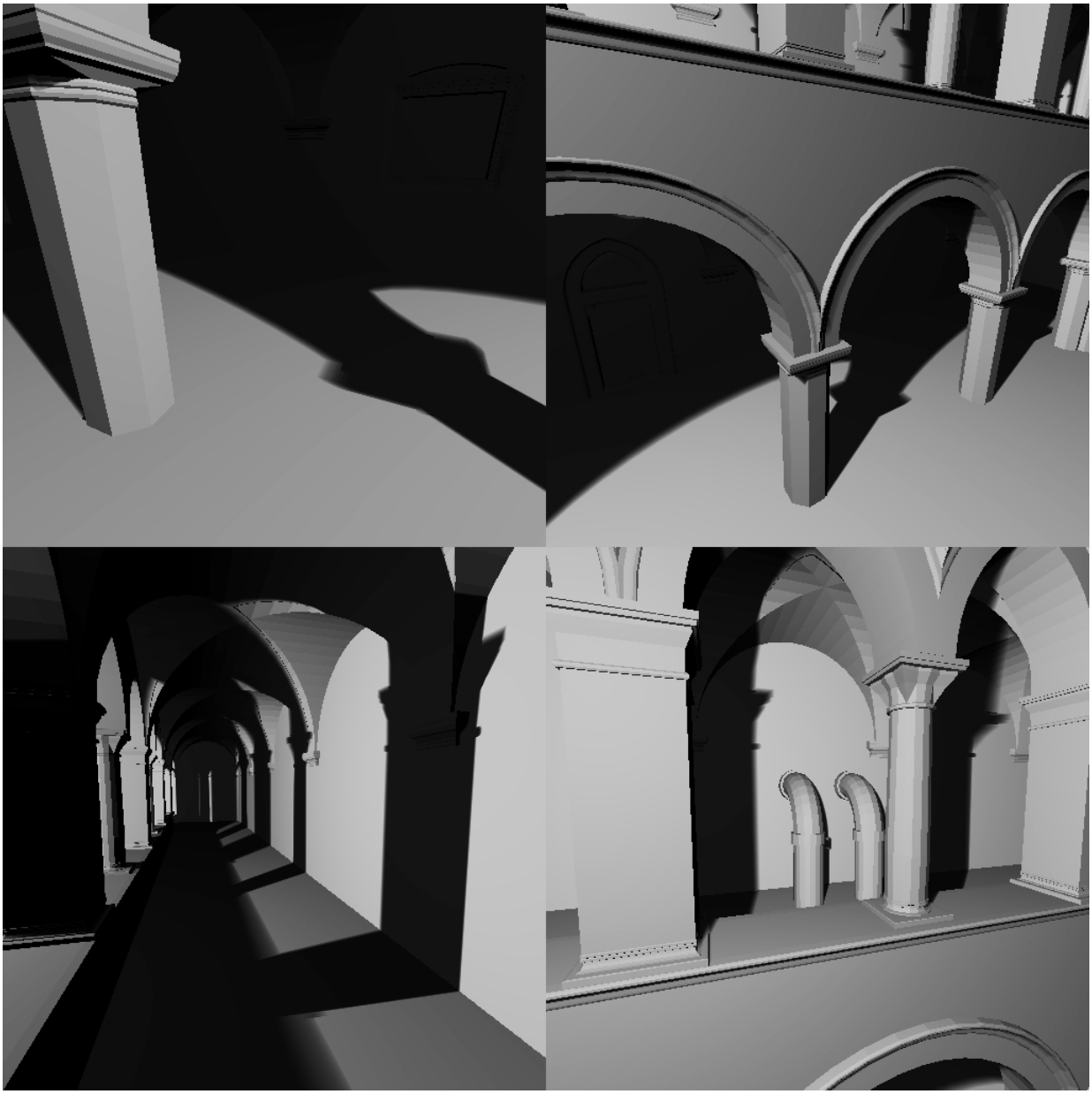

Figure 5.11: Soft shadows using edge widths.

Future Shadow Work

The edge width algorithm proposed has been implemented sucessfully for a kd-tree. Future work includes applying this algorithm for a BVH to support dynamic objects. Such a task proves to be more challenging since leaf nodes are not guar-

Table 5.1: Performance metrics for edge width, single sample and multi sampled shadows in the Sponza scene.

Shadow Type	FPS
Single Sample	15.0 fps
Edge Width	12.0 fps
4 Sample	4.26 fps
32 Sample	1.42 fps

anteed to be visited in front to back order which could lead to large sorting lists of distances, thus hindering performance.

Chapter 6

Effect of Processor Hardware and Threading

One of the main selling points for a ray tracing algorithm is its ability to be highly parallelized since each ray is traced independently. The following tests show how overall frame rate performance can change with different processor architectures and the number of physical hardware cores. RTTest is threaded by dividing the screen into fixed sized tiles and worked on independently by available threads. In the application, the number of threads spawned is always equal to the number of hardware cores available.

Figure 6.1 shows the scene used for benchmarking. The scene is comprised of two Ferrari F40s within a garage setting which is 89 754 polygons in total. All primary rays are using multi frustum traversal of a 16×16 tile size with perfect reflections traced using mono tracing for the metallic garage and the cars. Glass transparency through the windows are computed using coherent packets and the shadows are calculated in real time using edge widths. The benchmark scene was rendered at a 512×512 resolution.

The two processors that are used for benchmarking are the Intel Pentium D and the Intel Core 2 Quad with specifications shown in Table 6.1. The Pentium D is the predecessor to the Core 2 architecture which was clocked lower but was rearchitected for higher performance using a larger L2 cache.

Table 6.1: Processor specifications used for benchmarking.

Processor	# Cores	Clockrate	L2 Cache per Core
Intel Pentium D	2	3.2 GHz	1 MB
Intel Core 2 Quad	4	2.4 GHz	4 MB

Table 6.2 shows the resulting frame rate for each processor while utilizing 1,2 and 4 cores. For the Pentium D, there is almost a 2× speedup when moving to

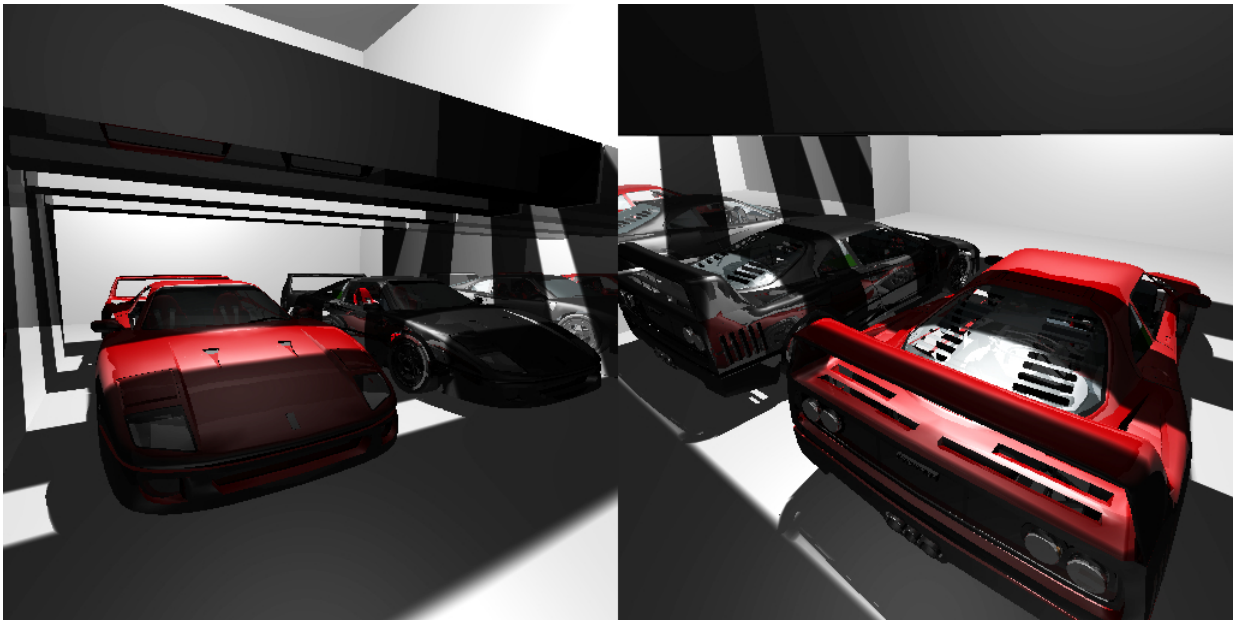

Figure 6.1: Scene used for benchmarking.

2 cores. On the Core2, the performance is almost double that of the Pentium D for 1 and 2 cores despite the fact that it is clocked at a lower rate. This speedup can be attributed to the revamped architecture with more efficient decoding stages, execution units, caches and buses. The Core 2 scales perfectly between 1 and 2 cores and slightly less than a factor of 2 between 2 and 4 cores. The reason that perfect speedup cannot be achieved in all cases is that each core shares the same memory bus which is a major point of contention if data is not already present in the cache. This is supported from the fact that a larger cache size provides perfect speedup from 1 to 2 cores on the Core 2 architecture but less than perfect speed up on the Pentium D due to the smaller cache. From the performance numbers, future performance can be predicted as going to 8 cores would achieve true "real-time" performance above 30 fps. Doubling the cores again to 16 cores would allow the same scene to be rendered at HD resolutions at a reasonably interactive rate. Due to the highly parallel nature of ray tracing, the future seems promising as processor architectures become more and more parallel in nature.

Table 6.2: Benchmarks for differenct processors

Processor	1 Core	2 Cores	4 Cores
Intel Pentium D	2.7 fps	5.0 fps	–
Intel Core 2 Quad	4.5 fps	9.0 fps	17.4 fps

Chapter 7

Conclusions

An introduction and overview of relevancy of ray tracing was discussed and how it compares to rasterization. Acceleration structures such as kd-trees, BVH and grids were introduced for both the building and traversal of such structures. The use of a SAH cost function was shown and the terminating criteria for tree building was established. Next, an indepth look into packet and frustum tracing was done showing the importance of exploiting the coherency between rays. By tracing multiple rays together, an SSE implementation can be used to take full advantage of the full vector width of the floating point unit. Using frustum traversal, the two stage MLRT algorithm was discussed followed by the early miss frustum algorithm for BVH traversal. Following the traversal discussion, background information on dynamic scenes using kd-tree and BVH rebuilding was given. Next, new ideas using packet reordering and n-ary BVH trees were introduced for increasing the efficiency of ray packets for secondary rays. Various shadow algorithms were then discussed as a setup for the edge width soft shadow algorithm introduced in Chapter 5.

Chapter 3 described the inner workings of the RTTest offline and runtime applications. A kd-tree was chosen for the acceleration structure to build using RTTest offline using "perfect splits" through polygon clipping. The memory layout of important structures were given along with the RTTest runtime rendering pipeline. These pipeline stages were profiled and the statistics of percentage of frame time usage were shown and discussed.

Chapter 4 introduced a new omni-directional packet traversal algorithm which has no direction limitations and results in a simpler implementation that avoids many special cases involving packet splitting. Next, benchmarks were shown for the frustum interval traversal algorithm as implemented in RTTest which yields a 2× speed up over standard packet traversal in most cases. The importance of a packet/leaf culling test was shown through performance numbers which can be detrimental to performance if not done for complex scenes. Next, the Multiple Frustum Traversal algorithm was introduced which traverses multiple frustums simultaneously using SSE to provide for quick masking of frustums and thus less nodes traversed and fewer intersection tests. Following the MFT algorithm, the Cone

Proxy Traversal algorithm was introduced which performs better than a pyramid shaped frustum for small tile sizes for primary ray acceleration. Cones were also shown to beat out pyramids when bounding multiple shadow samples to a spherical light source.

After the traversal algorithms, the ray/polygon intersection tests using the projected Barycentric coordinates test was described. Next a description of the fixed function shader system was detailed which supports diffuse, specular, reflective and transparent materials. Following the shader system description, various code level optimizations were introduced which were learned during the development of RTTest.

Chapter 5 introduces a novel shadowing algorithm using edge widths to render fast soft shadows with less accuracy. The ideas behind the algorithm were discribed in detail along with pseudocode. Visually, the shadows create a believable fuzzy shadow which are superior in quality to hard shadows. Artifacts such as light leaking were described along with methods of reducing them. Performance numbers show that the edge width method is comparable to performance to that of a single sampled hard shadow approach.

Chapter 6 shows the results of simple tests done on two different Intel CPUs with different number of cores and cache sizes. The Core 2 architecture provided a significant improvement over the older Pentium D architecture. As the number of utilized cores is doubled from 1 to 2, there is perfect linear speed up on the Core 2 architecture. Moving from 2 to 4 cores provides less than perfect speedup due to the memory bus contention which is a shared resource and contention point. Larger cache sizes can help alleviate the memory bus contention and even the integrated memory controllers in the new Intel Nehalem architecture.

In conclusion, this thesis introduces novel techniques and algorithms for a practical, real time ray tracing engine. A modification to the traditional kd-tree packet traversal algorithm is proposed which allows any packet of rays from a common origin to be traced together with no direction restrictions. This leads to a simpler implementation for both hardware and software renderers. Next, the Multiple Frustum Traversal algorithm is detailed which takes full advantage of the SIMD width for traversal via the updating of a frustum mask which allows the culling of many redundant traversal and intersection operations. The use of MFT results in a 12% increase in overall performance over a single frustum approach. Next, the Cone Proxy Traversal algorithm is derived which provides superior performance to a pyramid frustum for primary ray acceleration only for small tile sizes. Cones have also been shown to be a useful frustum for multi-sampled soft shadows to spherical light sources due to a tighter bound to the internal rays. Finally, a novel soft shadow algorithm is proposed called Edge Width Soft Shadows which requires one single sample edge width traversal. Edge width soft shadows have been shown to be high performance and provide smooth pneumbra transitions. Artifacts such as light leaking have also been discussed with several partial solutions.

References

[1] D. J. MacDonald and K. S. Booth, "Heuristics for ray tracing using space subdivision," *Vis. Comput.*, vol. 6, no. 3, pp. 153–166, 1990. 14

[2] T. L. Kay and J. T. Kajiya, "Ray tracing complex scenes," *SIGGRAPH Comput. Graph.*, vol. 20, no. 4, pp. 269–278, 1986. 17

[3] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker, "Ray tracing animated scenes using coherent grid traversal," *ACM Trans. Graph.*, vol. 25, no. 3, pp. 485–493, 2006. 20

[4] I. Wald, C. Benthin, M. Wagner, and P. Slusallek, "Interactive rendering with coherent ray tracing," in *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001* (A. Chalmers and T.-M. Rhyne, eds.), vol. 20, pp. 153–164, Blackwell Publishers, Oxford, 2001. available at http://graphics.cs.uni-sb.de/ wald/Publications. 21

[5] A. Reshetov, A. Soupikov, and J. Hurley, "Multi-level ray tracing algorithm," in *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, (New York, NY, USA), pp. 1176–1185, ACM, 2005. 25, 26

[6] I. Wald, S. Boulos, and P. Shirley, "Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies," *ACM Transactions on Graphics*, vol. 26, no. 1, 2007. 26, 47

[7] W. Hunt, W. R. Mark, and G. Stoll, "Fast kd-tree construction with an adaptive error-bounded heuristic," in *2006 IEEE Symposium on Interactive Ray Tracing*, IEEE, Sept. 2006. 27

[8] S. Popov, J. Gunther, H.-P. Seidel, and P. Slusallek, "Experiences with streaming construction of SAH KD-trees," in *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pp. 89–94, Sept. 2006. 27

[9] I. Wald, W. R. Mark, J. Gunther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley, "State of the art in ray tracing animated scenes," in *STAR Proceedings of Eurographics 2007*, pp. 89–116, The Eurographics Association, Sept. 2007. 27

[10] J. Gunther, H. Friedrich, I. Wald, H.-P. Seidel, and P. Slusallek, "Ray tracing animated scenes using motion decomposition," *Computer Graphics Forum*, vol. 25, pp. 517–525, Sept. 2006. (Proceedings of Eurographics). 27

[11] C. Lauterbach, S.-E. Yoon, D. Tuft, and D. Manocha, "Rt-deform: Interactive ray tracing of dynamic scenes using bvhs," *rt*, vol. 0, pp. 39–46, 2006. 27

[12] T. Ize, I. Wald, and S. G. Parker, "Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures," in *Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualization*. 27

[13] A. Reshetov, "Omnidirectional ray tracing traversal algorithm for kd-trees," *rt06*, vol. 0, pp. 57–60, 2006. 28, 41

[14] I. Wald, C. P. Gribble, S. Boulos, and A. Kensler, "SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering," Tech. Rep. UUSCI-2007-012, 2007. 28

[15] H. Dammertz, J. Hanika, and A. Keller, "Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays," in *Computer Graphics Forum (Proc. 19th Eurographics Symposium on Rendering), 2008*, pp. 1225–1234. 28

[16] I. Wald, C. Benthin, and S. Boulos, "Getting rid of packets: Efficient simd single-ray traversal using multi-branching bvhs," in *2008 IEEE Symposium on Interactive Ray Tracing*, IEEE, Aug. 2008. 28

[17] J. Amanatides, "Ray tracing with cones," *SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 129–135, 1984. 28, 48

[18] S. Laine, T. Aila, U. Assarsson, J. Lehtinen, and T. Akenine-Möller, "Soft shadow volumes for ray tracing," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 1156–1165, 2005. 28

[19] S. Parker, P. Shirley, and B. Smits, "Single Sample Soft Shadows," 1998. 28

[20] I. Wald, *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. 31, 54

[21] C. Benthin, *Realtime Ray Tracing on Current CPU CPU* . PhD thesis, Computer Graphics Group, Saarland University, 2006. 39

[22] P. S. Heckbert and P. Hanrahan, "Beam tracing polygonal objects," in *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 119–127, ACM, 1984. 43

[23] S. Reiter, "Realtime Ray Tracing of Dynamic Scenes," Master's thesis, Johannes Kepler University of Linz, 2008. 55

[24] E. de Castro Lopo, "Faster floating point to integer conversions.," 2001. http://mega-nerd.com/FPcast/. 60

[25] Intel Software Network, "Fast floating point to integer conversions," 2005. http://softwarecommunity.intel.com/articles/eng/2076.htm. 60