EFFICIENT RAY-TRACING TECHNIQUES USING GPU

Guangfu Shi

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

May 2011

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By:	**Guangfu Shi**
Entitled:	**Efficient Ray-Tracing Techniques Using GPU**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

______________________ Chair
______________________ Examiner
______________________ Examiner
______________________ Examiner
______________________ Supervisor
______________________ Co-supervisor

Approved ______________________
Chair of Department or Graduate Program Director

__________ 20 ______ ______________________
Rama Bhat, Ph.D.,ing., FEIC, FCSME, FASME, Interim Dean
Faculty of Engineering and Computer Science

Abstract

Efficient Ray-Tracing Techniques Using GPU

Guangfu Shi

Text of abstract.

Acknowledgments

Text of acknowledgments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recently, with the dramatically boost of computational power of the current generati on CPU hardware, there has been a surge of new interetest in ray tracing as a core rendering algorithm for real-time applications such as video games since it is applicable to push the rendering task to interactive level on powerful computing devices which are easily accessible for a mainstream PC platform. Recently emerged powerful graphics processing units (GPUs) and flexible programming model provides researchers and developers with the ability to have the critical part of the algorithm parallelized and capture many secondary rendering effects such as shadows, refelections and refraction.

In the context of computer graphics, the term *rendering* refers to the process of generating a two-dimension image from a three-dimensional virtual scene. Various rendering techniques have been widely used in many fields where computers are needed to generate images. Based on the algorithm used for the rendering process, rendering approach can be classified to two major categories: rasterization-based rendering and ray tracing-based rendering.

At high level, rasterazation algorithm iterats projects the polygons, which forms the geometry objects in the scene, into screen space for rendering. With the application of Z-buffer, we could have a correct occlusion result, however, it may cause redundant raster and shading calculations which is also known *overdraw*. One of the biggest advantages to a rasterization renderer is that a simple, immediate-mode graphics pipeline can be interfaced and only a set of geometry is required to be submitted to the renderer per frame without any need to introduce any acceleration structure

(AS) of the scene. This pipeline interface could lead to a hardware implementation to this pipeline interface can be fast and able to handle the dynamic scene interactively. However, the weakness of rasterization rendering, which is its inablity to properly render secondary effects such as reflection, shadows and refraction, is vital especially in the fields where high quality rendering is required. It is required to adopt some multiple pass techniques to achieve such effects which are inefficient and produce visible artifacts.

The idea behind ray tracing is straight-forward, instead of projecting geomery into screen, ray tracing shoots rays from the virtual camera into scene space and finds intersection points with the geometry object, and then secondary rays can be generated from the intersection point and cast into the scene for secondary effects. Once the intersection points are found, the final colours are going to be calculated and produce the final image. The major advantage of ray tracing over rasterization is its inherent correctness in presenting the optical phenomenon in the final image, no other special techniques need to be adopted to achieve shadows, reflection and refraction such secondary effects. Due to the nature of ray tracing, massive demand of computation power is the shortcoming and becomes the major barrier in the way to bring ray tracing to real-time applications.

The essential ray tracing task is to find the intersection points between rays and geometry, it is safe to classify ray tracing as a spatial searching problem, therefore, it will be helpful to incorporate a kind of AS which subdivides the scene to achieve efficient searching of intersection points. The AS, however, leads to a more complex hardware implementation since the rendering is no longer immediate and how to effiently build and maintain the AS bring more challenges on boosting the performance ray tracer.

In addition to accelerate the ray tracing algorithmically, the revolutionary development of computing hardware driven by *Moore's law* opens up another opportunity that cannot be ignored to make the ray tracing faster. Current generation x86 Central Processing Unit (CPU) is designed to maximize the parallelism including *Instruction Level Parallelism* and *Thread Level Parallelism*, these new features interest researchers to present new ideas and algorithms to take advantage of, such as rays packet traversal algorithm. Another common computing device in a system, which is the Graphics Processin Unit (GPU), has become much more powerful especially in float-point computing, furthermore, more flexible programming model and massive parralelism architecture make it as an alternetive device to use for ray tracing. However, none of these optimization approaches is automatically ann cheap, data structures and algorithms have to be carefully re-designed to be friendly to a specific device, and some constrains also have to be considered or they will significantly hit the overall performance.

Chapter 2

Background

2.1 Ray Tracing Algorithm Overview

The ray-tracing algorithm is adopted by most photorealistic rendering system to generate a highly realistic image of a 3D scene. The core concept of the ray-tracing algorithm is straightforward, it is based on shooting the ray from the view point to a 3D scene as it interacts with the objects and light source in the scene. It can be understood as the reversed process of how our eyes perceive the world, which is receiving the light rays emitted by the light source and bounced off to the eyes. To simulates the physical phenomenon, a rendering system will have to model the following key components :

Camera

The camera defines the position from where the scene is being observed and how the rays are generated. As an abstraction of the real camera, the camera in the rendering sytem is typically modeled with an eye position and the image or film in front of the eye, expecting the answer to what the color to display at each point in the image? Give the eye position as the origin and a point on the image, a ray can be generated by the camera and will be shot to the scene and the ray-object intersection test can be performed to determine the color value of that point on the image.

Scene Description

The "scene" to be rendered is conceptually a database of "geometric primitives", which can be from simple geometric shapes such as polygons, spheres to complex shapes such as Bezier or NURBS patches. In fact, ray tracing can handle *any* type of primitives as long as there is a proper algorithm to compute and intersection between ray and the primitive.

Choosing the types of geometric primitive the ray tracer is going to support is an important design decision to make. The ray tracer can directly support complex primitives without tessellating them into polygon, thus they can be ray traced very efficiently and accurate, for example, directly compute the intersection between the ray and a sphere is almost trivial, while a tessellated sphere requires hundreds of triangles for a reasonable accuracy. However, the ray tracer that only supports these high-level primitives suffers from huge complexity and poor maintainability. Specific routine has to be implemented to support new type of primitive. On the other hand, supporting only triangles leads to a simple and optimized ray tracer implementation, the intersection calculation routine between a ray and triangle is quite efficient. Furthermore, triangle-based mesh can be used to approximate all of the high-level primitives very well and has a widely usage in the industrial application such as video games, CAD and movies. In conclusion, a triangle-based ray tracer is a more preferable design due to its simplicity and usability.

Ray Casting

Ray casting, the key problem for a ray-tracer to solve, is to shoot rays through each pixels on the image into the scene and then find out which geometric primitive, if any, that the ray hit first and where intersection point is. All the objects in the scene will be tested against the ray the first one will be selected. Given a ray r in parametric form:

$$r(t) = o + td \tag{1}$$

where o is the ray's origin and d is the direction vector which is usually normalized so that the parameter t can be simply treated as the distance from the origin. The valid range of t is $[0, \infty]$. With surface defined by an implicit function $F(x, y, z) = 0$, it is easy to find the intersection point by substituting the ray equation into the implicit equation to obtain a new function whose only parameter is t, solving this function for t, and substitute the smallest positive root into the ray equation to find the desired point. If there are no positive roots, the ray must miss the surface. The intersection point is not sufficient, ray-tracer also needs to know the additional properties of the surface such as normal vector and materials at that point to perform the shading stage later.

As most scenes contains multiple objects, the brute-force approach that test the ray against each object can be too inefficient. To improve the performance, the acceleration structure is adopted by the ray-tracers to quickly cull whole groups of objects during the ray intersection process. The acceleration structures will be discussed in more detail in the following section.

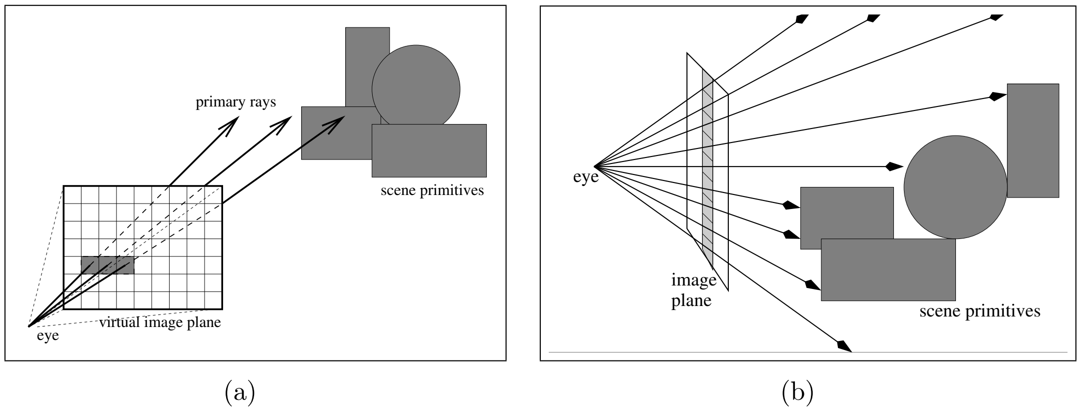

Figure 2.2: *The principle of ray casting: In order to compute an image of a scene, "primary rays" are shot from the camera, through each pixel of the virtual image plane, and cast into the scene (a). For each such ray, the closest object hit by this ray is determined by intersecting the ray with the geometric primitives that make up the scene (b).*

Light and Visibility

In order to correctly support shadows, light sources only contribute to the incident illumination if the hit point is not occluded from the position of the respective light source, which is checked by tracing a shadow ray towards the direction of the light source. Additional to direct illumination from light sources, illumination from arbitrary other directions (e.g. from the reflection and refraction directions for specular effects) can be considered by casting a secondary ray into the respective direction and recursively evaluating the light being transported along this rays. This recursive evaluation then proceeds in exactly the same way as for the primary ray. Of course, these secondary rays can in turn trigger another recursion level of new rays, etc.

Ray-Surface Interaction

The appearance of a surface is determined by the how the surface interact with the light. The properties of the surface are typically abstracted as the *material* which is actually a parameterized descriptions of the appearance at each point on the surface. These properties are modelled mathematically by the *Bidirectional Reflectance Distribution Function* (BRDF). This function takes the incoming light direction, the outgoing light direction and the point hit by the light as the inputs, returns the amount of energy reflected by the surface at that point.

Recursive Ray Tracing

The rays shot from the camera (termed as "primary rays") can be reflected about the surface normal at the intersection point and transmitted when intersect the transparent object, spawning secondary rays that ray-tracer need to recursively call the ray-tracing routine to trace, the contribution of the secondary rays will be added to the primary rays.

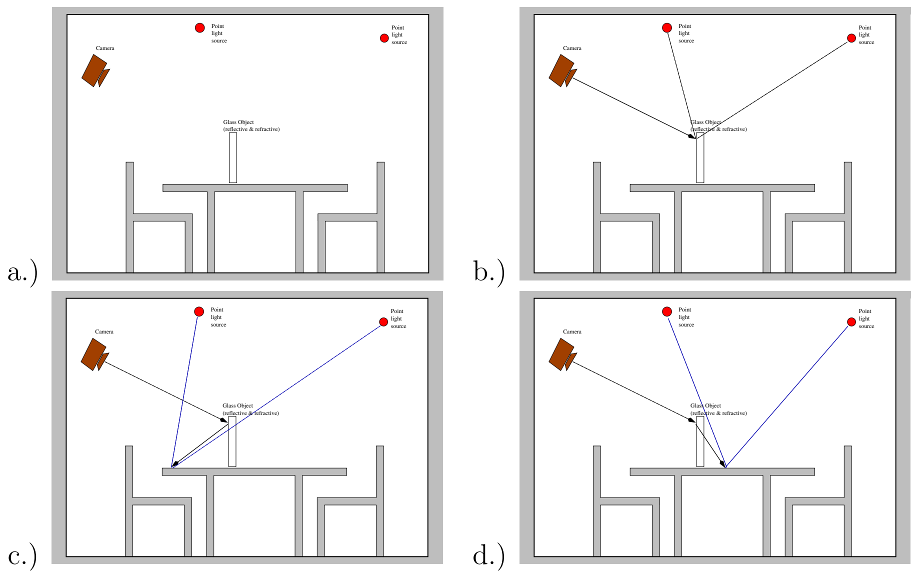

Figure 2.3

2.2 Acceleration Structures

The ray shooting algorithm which only uses a list of of all objects in the scene is also called a naive ray-shooting algorithm. It is a linear search process that tests the ray with all the objects and choose the one with the closest intersection found, if such an object exists. Thus the time complexity is $O(n)$. As the scene complexity grows, the application using naive RSA runs unacceptably slowly. The straightforward approach to accelerate ray shooting algorithm is to reduce the number of unnecessary ray-primitive intersection tests. Thus we introduce the concept of acceleration structure.

The AS essentially is a kind of data structure which stores the spatial relationships of the geometric objects in the scene, thus it is also called *spatial data structure*. The idea behind it is similar to accelerating other type of searching problem, if we can establish a kind of index structure which offers the relationship between the elements before performing query, lots of unnecessary tests can be skipped and the performance can be boosted. Over the last 20 years, many different kind of acceleration structures have been developed, however in principle these techniques mainly differ in whether they hierarchically organize the scene primitives (as done by Bounding Volume Hierarchy), or whether they subdivide the space into a set of non-overlapped voxels (as done by kd-trees or grids).

Generally speaking, acceleration structures can be classified into two types by the approaches they adopt: spatial subdivision and object subdivision. Spatial subdivision approach subdivides the space into regions, hierarchically organizes the geometric objects which fall in the same spatial region into an object called *cell*, and maintains the spatial relationships between the cells. When the ray shooting algorithm is performing, we test the ray against the cells instead of actual geometric objects, if the intersection is found, we go deep in this cell for further test, otherwise this cell will be skipped cause there is no chance that the ray can hit the any geometric objects in this cell, a significant number of intersection tests will be reduced. The most widely-used spatial subdivision structures are uniform grid and kd-tree.

Object subdivision approach, on the other hand, logically breaks the object in the scene down into a set of objects groups, similar with building a scene graph. For example, a desk can be broke down into four legs and a surface, this forms a hierarchy structure to represent a desk object. If a ray does not hit the desk's bounding volume, there is no chance that the ray hits any parts of the desk and they will be culled, otherwise the ray will be recursively tested against each part of desk. The most widely-used structure of this approach is bounding volume hierarchy (BVH).

2.2.1 Grid

2.2.2 Bounding Volume Hierarchy

Bounding volume hierarchy (BVH) are an acceleration structure based on primitive subdivision. The primitives are partitioned into a hierarchy of non-overlapped sets. In the hierarchy, the leaf node keeps the bounding volume of the attached primitive as the nodes' bounding volume, it also stores the actual primitive reference, while the interior node stores the bounding volume of its children node. When a ray is traversed through the tree, any time it misses a node's bounding volume, the entire sub-tree rooted by that node will be skipped. There are two important properties of BVH structure, one is that each primitive referenced by the hierarchy only once, thus it will not be tested against the ray multiple times; the other one is that the memory consumption to represent the BVH is bounded. Figure 2.4 shows a simple scene and its corresponding BVH tree.

BVH Construction

Several stages are needed in the BVH construction process.

Algorithm 1: Build BVH from primitives

```
input  : The collection of the primitives in the scene
output: The root node of BVH tree
begin
    Initialize the array for primitives Recursively build BVH tree Map the BVH tree
    to a compact linear representation
end
```

Firstly, we and store centroid of the bounding box, the complete bounding box and the reference to the primitive for each primitive into an array as a working buffer.

Next, a partition procedure will be performed to split the primitive into two subsets and recursively build the BVH for the subsets. This produces a binary tree where each interior node holds the pointer to its children nodes and each leaf node holds the references to one or a list of primitives. The partition step can be more complex,

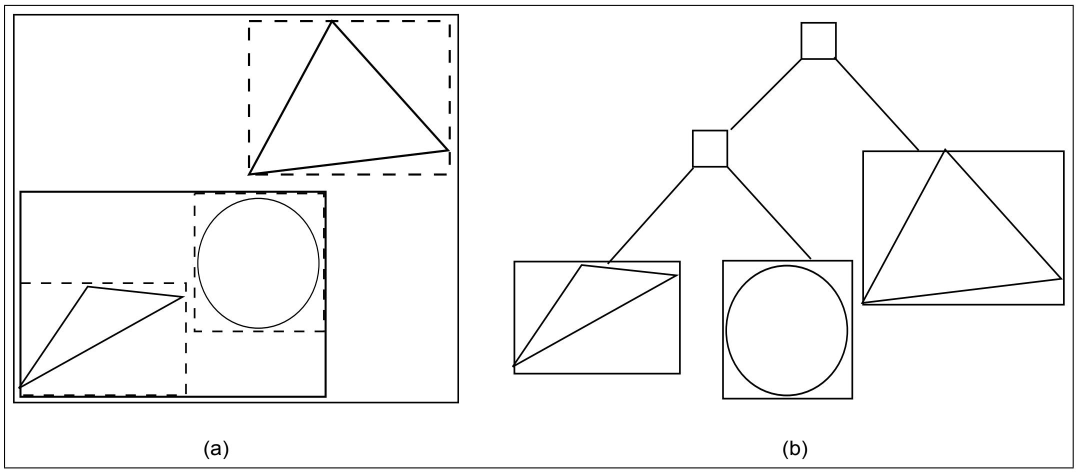

Figure 2.4: Bounding Volume Hierarchy for a Scene
(a) A simple scene containing several objects. The bounding boxes of the primitives are shown by dashed lines. The smaller triangle and the sphere are grouped together with a bounding box before being bounded by the bounding box that is corresponding the entire scene (both shown in solid box). (b) The corresponding bounding volume hierarchy, the root node stores the bounding box of the entire scene, it has two children node here, one is a leaf node with the bigger triangle attached, the other one is an interior node that encompasses the sphere and smaller triangle as leaf nodes.

given n primitives, there are $2^n - 2$ possible partitions, while many of them may lead to suboptimal BVH. Generally we choose the partition plane along a coordination axis, meaning that there are about $6n$ candidate partitions ($2n$ partitions for each axis). We use one axis of the three to place the partition. In practice, the axis with the greatest variation of bounding box centroid for the current set of primitives is a good choice. When choosing the position to place the partition plane, a general goal is to select a partition that doesn't have too much overlap of the bounding boxes of the two resulting primitive sets, this is based on the fact that overlapping of two primitive sets will cause the traversal of both children subtrees requiring more computation clipping collection of primitives.

There are several schemes to choose where to place the partition: a simple method is to partition at the midpoint of the primitives' centroids along the splitting axis. However, this method may result a substantial overlap of two primitive subsets with certain distribution of primitives. Another straightforward but more adaptive partition scheme is to partition the primitives into two subsets with equal number of primitives. These two primitive partitioning approaches above can work well for some distributions of primitives, but they often choose the partitions leading to more nodes of the tree being traversed by the ray and hence unnecessarily ray-primitive intersection computations at rendering time. A more widely-used scheme used by best current algorithms for building acceleration structures are based on the “surface area heuristic” (SAH) model which provides a well-grounded cost model used to determine which of a number of partitions of primitives leading a better BVH for ray-primitive intersection tests. The SAH model will be discussed in depth in the chapter. As the partition has been fixed, we generate two interior nodes and classify the geometries into them by testing the primitives against the bounding volume the generated nodes.

Instead of only store the root node of the tree and visiting the nodes by manipulating the pointers, it is an important optimization to store the BVH tree into a compact linear array in depth-first array. This representation makes the BVH traversal more cache-friendly thus improves the overall performance. In the memory, the first child of each interior node is right next to the interior node, for the second child node, the offset to it is stored explicitly in the data structure of BVH tree node. See Figure 2.5 for an illustration of the topology of BVH tree nodes and its representation in memory.

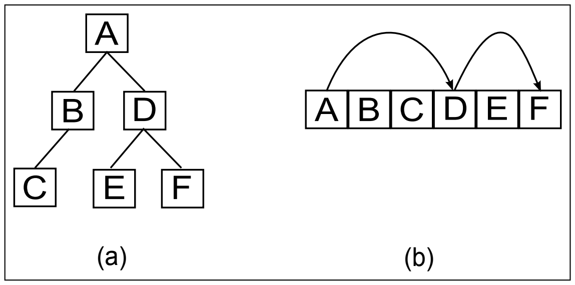

Figure 2.5: (a) The topology of the nodes in BVH tree. (b) The memory layout of the nodes of the BVH tree. The nodes are stored in depth-first order.

BVH Traversal

The traversal of BVH is a simple process, the ray is tested against the bounding volume of the node in the tree from the root node which represents the entire scene down to the leaf nodes. If an intersection occurs (including the case that the ray starts inside of the node), for an interior node, a down traversal is performed to visit the children nodes and save the far node onto a stack, for a leaf node, each primitive will be tested against the ray. If there is no intersection, no further test will be be performed in the current sub-tree and the next node to be visited is retrieved from the stack (or finish the traversal if the stack is empty). The intersection test between the ray and bounding volume is fast, take bounding box for example, ray-box is fast ray-slab test [1].

Algorithm 2: BVH Traversal

```
begin
    // Initialise the stack, push the root node
    push root onto stack  while true do
        // Retrieve the top node of the stack
        node ⟵ stack[nodeIndex]
        if the ray intersects the bounding box of the current node then
            if node is leaf node then
                Intersect ray with primitives in leaf node
            else
                Put far BVH node on stack, advance to near node
        else
            if stack is empty then
                break
            Advance nodeIndex
end
```

2.2.3 KD-Tree

Before introducing kd-tree, a short introduction to *Binary Space Partitioning* (BSP) tree has to be made. A BSP tree is a data structure developed in the purpose of solving the hidden surface removal problems in computer graphics. The binary space partitioning is a process of adaptively and recursively subdivide the voxel of the scene into irregular sized regions with a chosen split plane until a termination criteria has been satisfied. Different scheme of choosing the split plane leads to two major variants of BSP tree, polygon-aligned form and axis-aligned form. The polygon-aligned form chooses the plane aligning the polygon in the scene as the split plane, while axis-aligned form always chooses the planes perpendicular to a certain coordinate axis. The kd-tree is actually the axis-aligned form of the BSP tree which makes both traversal and construction of the tree more efficient. We will focus on kd-tree here in this thesis.

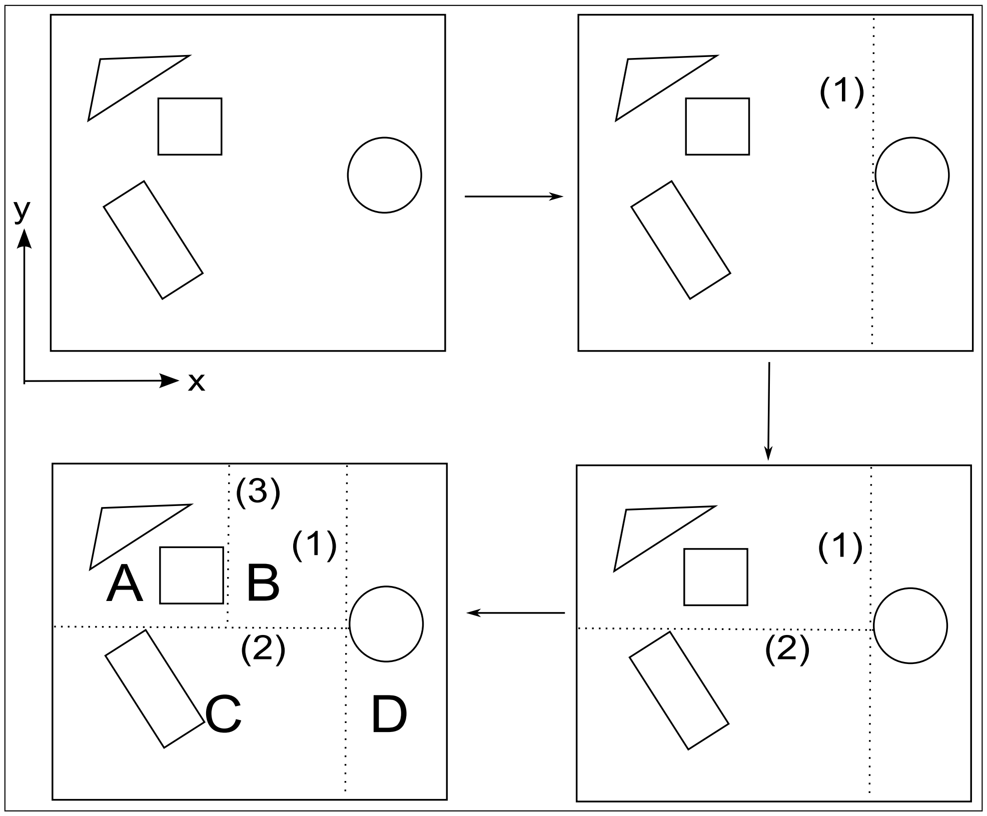

Figure 2.6: The top view of a scene which is recursively subdivided by the split planes (shown in dashed lines and labeled by numbers) into several small regions (labeled by the letters).

Figure 2.6 provides an overview of how the scene is recursively subdivided along one of the coordinate axes. In the figure, a scene is recursively subdivided by 3 split planes. Initially, the entire scene as the root node of the tree, the first split is along the x axis and it subdivides the scene into two regions. Then the left region is refined a few more times with the number 2 splitting plane. In the subdivision process, some criteria such as which axis is used to place the splitting planes, at which position along the axis the plane is placed and at what point subdivision terminates can all substantially affect the performance of the kd-tree in practice.

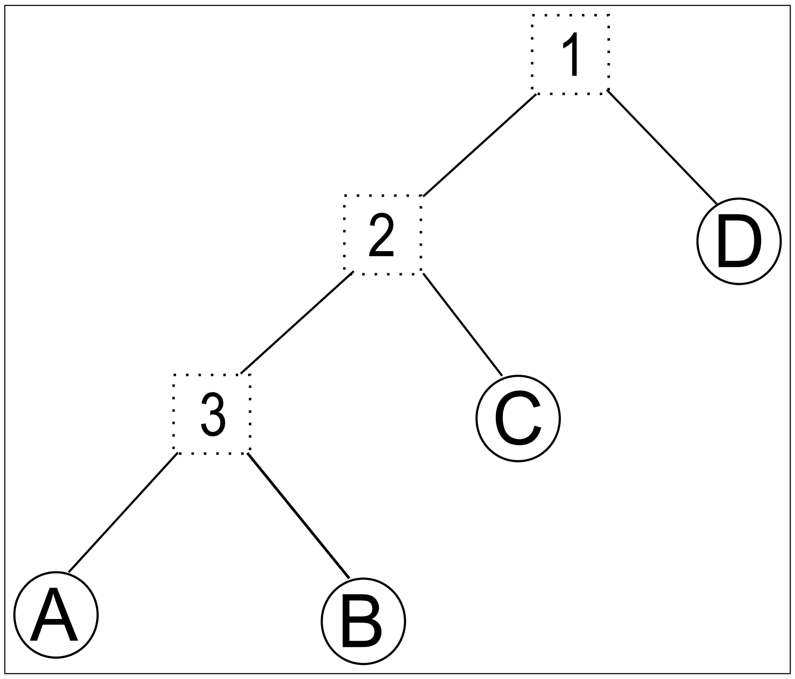

Figure 2.7: The kd-tree built from the scene in Figure 2.6

KD-Tree Data Representation

As a special case of binary tree, the node of kd-tree has the similar representation. Each interior node has to contain the following three data fields:

- Split axis: which axis is used to to place the splitting plane for this node
- Split position: the position of the splitting plane along the splitting axis
- Reference to children nodes: information associates to the children of this nodes

While each leaf nodes only contains the reference to the collection of primitives attached to it. Both leaf nodes and interior nodes share a common data filed which is a flag indicating the this node is a leaf or interior.

In practice all these data fields can be put in a compact data structure which consume 8 bytes (assuming the system uses 4-byte floating point values and pointers) regardless if the node is an interior or a leaf node using the **union** in C/C++ programming language. They share the same memory location using one bit of data to indicate it is leaf node or interior and the node-specific data is encoded into a 4-byte unsigned integer value. For interior node,

Listing 2.1: Data layout of kd-tree node

```
struct KDTreeLeaf{
  unsigned int flag;
  // bits 0..30   : offset bits
  // bit 31(sign) : flag whether node is a leaf
}
struct KDTreeInterior{
  float split;
  unsigned int flag;
  // bits 0..1  : splitting dimension
  // bits 2..30   : offset bits
  // bit 31(sign) : flag whether node is a leaf
}
union KDTreeNode{
  struct KDTreeLeaf;
  struct KDTreeInterior;
}
```

KD-Tree Construction

A kd-tree construction essentially is a process of recursively subdivide the current spatial region into two subregions with a certain axis-aligned split plane that is selected using a particular strategy. The construction process starts with the extend of the entire scene. Initially, the root node is leaf node, and all newly generated nodes are also leaf nodes, when a node is split by split plane, it will become an interior node, and the objects associated with it will be sorted to into its two new descendants.

The recursion will be terminated when a certain termination criteria is reached. Commonly we use the maximum depth of the recursion or a pre-defined threshold of the number of the primitives attached to a leaf node as the termination criteria. In practice, the value $8 + 1.3\log(N)$ is found a reasonable maximum depth of the tree for a variety of scenes [2].The construction of kd-tree is described in pseudo-code in Algorithm 3.

Algorithm 3: Recursive KD-tree construction

```
function BuildRec($T$, $V$)
begin
    if Terminate($T$, $V$) then
        return CreateLeafNode($T$);
    $p$ = FindSplitPlane($T$, $V$);
    ($V_L$, $V_R$) = Split $V$ with $p$;
    ($T_L$, $T_R$) = ClassifyTriangles($T$, $V_L$, $V_R$, $p$);
    return CreateNode($p$, BuildRec($T_L$, $V_L$), BuildRec($T_R$, $V_R$));
end

function KDTreeBuildRec($T$)
begin
    $V$ = CalcBounds($T$);
    return BuildRec($T$, $V$);
end
```

As shown in the above algorithm, in each partition of the current node, a split plane has to be selected, this is done by the function *FindSplitPlane*. For a kd-tree, a split plane can be positioned arbitrarily, as long as they are perpendicular to one of the x, y, z axis. However, different strategy of choosing the split plane may lead to a considerable performance gap up to a factor of two of more when traversing the built kd-tree [**?**]. More detail will be discussed in chapter 3.

KD-Tree Traversal

Given a ray R and a kd-tree, the kd-tree traversal is a procedure that identify the sequence of the kd-tree leaves intersected by the ray. There are several types of traversal algorithms developed, such as sequential, recursive and those with neighbor-links. Due to the simplicity, here we only describe the recursive traversal algorithm. The traversal is in a top-down fashion, starting from the root node, recursively descends the branches of the kd-tree along the ray path. Before going into any details of the ray traversal algorithm, a few terminologies have to be introduced, For each interior node the ray visits, the mutual position of the origin of a ray and the axis-aligned box $AB(s)$ of this node has to be considered. The first configuration is when the origin of a ray is located outside, also known as "the ray with external origin", and the other one is when the origin of the ray is inside the $AB(s)$, we call it "the ray with internal origin". When a bounding box is intersected by a ray R, there are two important points along the ray path which can be expressed by two signed distance. The point where R enters the bounding box is called *entry point*, corresponding to the *entry signed distance*. The point where R leaves the bounding box is called *exit point*, corresponding to the *exit signed distance*.

When the ray enters a kd-tree node (one traversal step), the intersection between the ray and the bounding box associating to the current node creates a parameter interval t_{near}, t_{far} which are the signed distance of the entry and exit point. Intersecting the ray with bounds of the entire scene (the tree's root node) gives the initial t_{near}, t_{far}, then the parameter interval is updated incrementally during the traversal. If the ray misses the overall bounding box ($t_{near} > t_{far}$), then the traversal can exit immediately.

If the current node is a leaf node, each primitive attached is tested against the ray and update a parameter $t_{closest}$ to find the closest intersection point.

If the current node is an interior node, it has to determine which of the two children the ray enters first. We calculate the parametric distance d to the splitting plane of the node in the same manner as was done in computing the intersection of a ray and axis-aligned plane for the ray-bounding box test, $d = (t_{split} - r_{origin}[dim])/r_{dir}[dim]$, and then compare d to the current ray segment t_{near}, t_{far}. If the ray segment lies completely on one side of the splitting plane ($d >= t_{far}\ or\ d <= t_{near}$), the subtree on the other side can be skipped immediately and the traversal will continue to proceed to the corresponding child voxel. Figure 2.8 shows some configurations of this case.

If none of the children nodes can be culled, the position of the ray's origin with respect to the splitting plane is enough to determine which of the child node is closer in turn should be visited first, figure 2.9 illustrate this case.

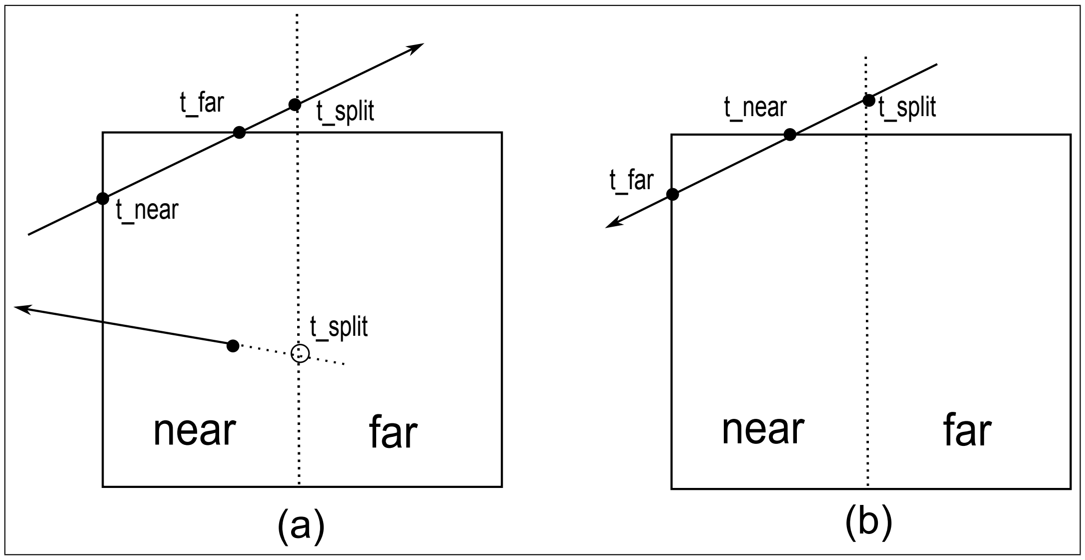

Figure 2.8: Two cases where the traversal will only proceed one child node.

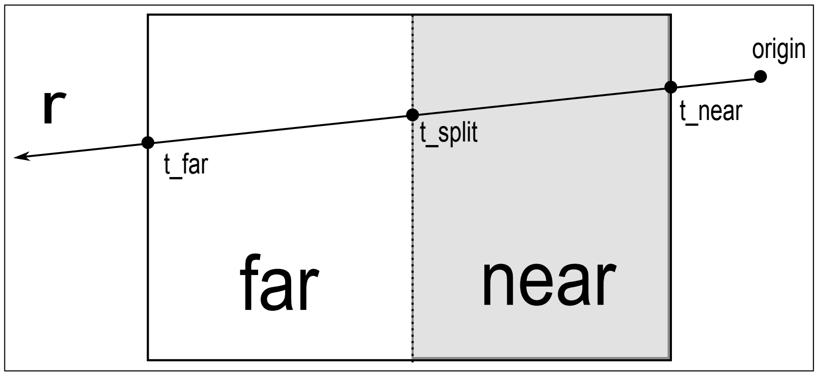

Figure 2.9: The children nodes are labeled as "near" and "far" by checking $t_{split} - r_{origin}[dim]$. If it is positive, the ray enters the left node first; otherwise the right node is the "near" node. The parameter interval for traversal the "near" and "far" node are t_{near}, t_{split}, t_{split}, t_{far} repectfully.

Figure 2.10 shows the basic process of ray traversal through the tree.

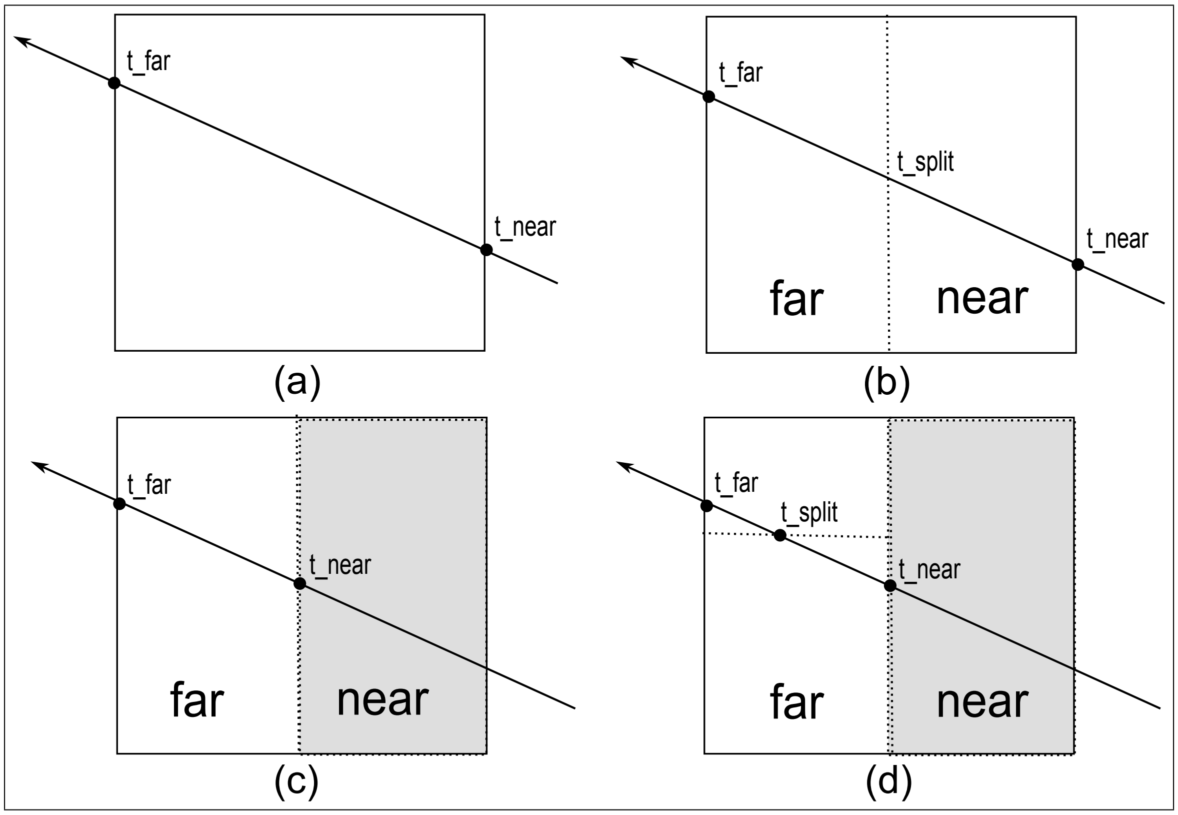

Figure 2.10: (a) The ray intersects against the bounding box of the kd-tree. The two intersection points can be simply represented by a parametric range $[t_{near}, t_{far}]$. (b) Assume this node is an interior node, it is necessary to consider the two children nodes. The node that the ray enters first is labeled as "near", corresponding the range $[t_{near}, tsplit]$. If the near node is leaf node which contains geometric primitives, ray-primitives intersection tests will be performed; otherwise, its children nodes will be processed. (c) If no intersection found in the near node, then the far node (on the left) is processed. (d) The traversal process will continue in a depth-first, front-to-back order, until the closest intersection is found or the ray exits the tree.

Algorithm 4: Recursive KD-Tree Traversal

```
function KDTreeTraversalRec($r$, $rootNode$)
begin
    $t_{near}, t_{far} \leftarrow$ RayBoxIntersection($r$, $rootNode$);
    if $t_{near} > t_{far}$ then
        // The ray misses the bounding box
        return;
    TraversalRec($rootNode$, $t_{near}$, $t_{far}$);
end

function TraversalRec($node$, $t_{near}$, $t_{far}$)
begin
    Initiailise $t_{closest}$ to the maximum distance;
    if $node$ is leaf node then
        Intersect all the primitives in the node against the ray;
        return $t_{closest}$;
    $d \leftarrow$ ($node.split$- $r.origin[node.dim]$) / $r.dir[node.dim]$;
    if $d <= t_{near}$ then
        // Cull the "near" node
        return TraversalRec( RetrieveFarNode($node$), $t_{near}$, $t_{far}$);
    else if $>= t_{far}$ then
        // Cull the "far node
        return TraversalRec( RetrieveNearNode($node$), $t_{near}$, $t_{far}$);
    else
        // Need to proceed both children nodes
        $t_{hit} \leftarrow$ TraversalRec(RetrieveNearNode($node$), $t_{near}$, $d$);
        if $t_{hit} <= d$ then
            return $t_{hit}$;
        return TraversalRec(RetrieveFarNode($node$), $d$, $t_{far}$);
end
```

Chapter 3

Efficient KD-Tree For Ray-Tracing

3.1 KD-Tree Construction Algorithm

The general kd-tree construction algorithm has been presented in the previous text, in this section, more advanced techniques adopted on the construction of kd-tree will be discussed in depth.

3.1.1 Surface Area Heuristic

In the algorithm 3, a recursive construction of kd-tree in top-down fashion has been presented, essentially all kd-tree construction algorithm follow the same scheme. The key operation in the construction of kd-tree is to effectively picking a split plane in every subdivision. There are several best known methods for positioning the splitting plane in the kd-tree:

In [3], Vlastimil Havran introduced "surface area heuristic" (SAH), which provides a well-grounded cost model that estimates the average cost of traversing an arbitrary ray through the kd-tree. The estimated cost is used to determine which split plane among a number of split candidates should be chosen in order to lead to a more efficient acceleration structure for ray-primitives testing. Most of the best current acceleration structures' building algorithms are based on SAH cost model.

The SAH cost model provides an estimation of the computational expense of performing ray intersection tests, including the time spent traversing nodes of the tree and the time spent on ray-primitive intersection test for a subdivision of primitives. The goal for the kd-tree construction algorithm to follow is to minimize the total cost, particularly for a greedy algorithm like algorithm 3, the goal is to minimize the cost for each subdivision step.

For each subdivision step, we have to decide whether to make the current region a leaf node with all of the overlapping primitives attached or refine the region to create two subregions. In the first case, any ray that passes through this region will be tested against all of the primitives leading to a cost of $\sum_{i=1}^{N} C_{isect}(i)$

Where N is the number of primitives and $C_{isect}(i)$ is the cost of computing the intersection between the ray and the ith primitive.

For the other case which is to split the region, the cost can be modeled with the following equation:

$$C(A, B) = C_{trav} + p_A \cdot \sum_{i=1}^{N_A} C_{isect}(a_i) + p_B \cdot \sum_{i=1}^{N_B} C_{isect}(b_i) \tag{2}$$

Where the C_{trav} is the cost of traversing the interior node, p_A and p_B are the probabilities of the ray passing through each of the child nodes. a_i and b_i are the primitives attached on the two children nodes, and N_A and N_B are the number of the primitives overlapping with of the subregions respectively.

The probabilities p_A and p_B can be computed using the ideas from geometric probability. A convex volume A contained in another convex volume B, the conditional probability that a random ray passing through B will also pass through A is the ratio of their surface areas, s_a and s_b:

$$p(A|B) = \frac{s_A}{s_B} \tag{3}$$

3.1.2 KD-Tree Construction With SAH

As described in the chapter 2, the kd-tree is built with a recursive top-down algorithm. At each step, we have an axis-aligned region of space and a set of primitives that overlap the region. Either the region is split into two subregions and turned into an interior node, or a leaf node is created with the overlapping primitives, terminating the recursion. The strategy of choosing the split plane in the kd-tree construction process may have considerable impact on the performance of ray-tracer. An naïveapproach is to place the split plane at the middle of region along certain axis splitting each node in half at each level and generating a balanced binary tree. Similar to the binary search tree, this approach is beneficial when the goal is to quickly figure out in which leaf node a certain point locates. However, in addition to quickly query the geometric primitive hit by the ray, the goal of kd-tree traversal is to find the closest intersection point. With the strategy of picking the spatial middle point as the split plane, it is likely to create the leaf nodes which contain lots of geometr as the distribution of geometry is not taken into account. The disadvantage of this approach is that whenever a ray traverses through a leaf node, it has to test for intersection with all of the primitives even there is no chance they could intersect. To minimize the redundant intersection tests, the distribution of geometry has to be somehow encoded into the kd-tree built from a certain scene to provide heuristic information for the ray tracer in kd-tree traversal stage. It is optimal to maximize the space of the leaf node which contains little geometry and minimize the space that contains more geometry. Figure 3.2 shows the different effect when traversing the tree built by two different approaches.

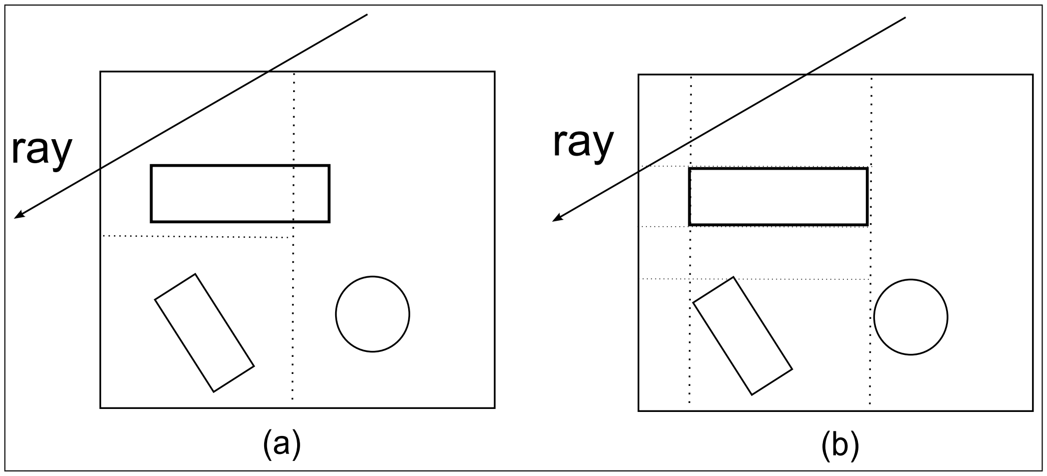

Figure 3.2: (a) The scene is subdivided using the spatial middle point and the created leaf nodes will cause redundant intersection tests. (b) The scene is subdivided based on the SAH cost model resulting a higher quality kd-tree. When a ray traverse through the kd-tree, it can quickly skip those leaf nodes without performing intersection tests since the empty region is isolated from the region that contains more geometry.

With the definition of the SAH cost model, we can easily adopt the SAH in the kd-tree construction algorithm. As shown in the algorithm 5.

The termination criteria of the SAH-based algorithm is more stable when to stop subdivisions. As the cost of a leaf node can be modeled as $C_{leaf} = K_{isect}|T|$, the subdivision will be terminated when the further split of this leaf node has higher cost than not splitting at all.

The "FindSplitPlaneSAH" function takes the list of triangles T, the voxel that to be split V and an axis k as the input parameters, outputs the split with the minimum cost. However, this process is not trivial compares to the spatial median splitting which only costs $O(1)$. To calculate the cost for each split candidates, we have to determine the number of primitives of both left and right child N_L and N_R. This can be done by classifying the triangle list which cost $O(N)$. Therefore the overall complexity for finding the optimal split algorithm is $O(N^2)$.

Algorithm 5: Find the optimal split plane based on SAH

```
function ClassifyTriangles($T$, $V_L$, $V_R$, $p$) return ($T_L$, $T_R$)
begin
    $T_L = T_R = \phi$;
    forall $t \in T$ do
        if $t$ overlaps with $V_L$ then
            append $t$ to $T_L$;
        if $t$ overlaps with $V_R$ then
            append $t$ to $T_R$;
end

function FindSplitPlaneSAH($T$, $V$, $k$) return $p_{best}$
begin
    forall $t \in T$ do
        $p$ = CalcBoundEdge(CalcBounds($t$), $k$);
        Append $p$ to split plane list $P$;
    forall $p \in P$ do
        ($V_L$, $V_R$) = split $V$ with $p$;
        ($T_L$, $T_R$) = ClassifyTriangles($T$, $V_L$, $V_R$, $p$);
        $C$ = SAHCost($V$, $p$, $|T_L|$, $|T_R|$);
        if $C < C_{min}$ then
            $(C_{min}, p_{best}) = (C, p)$;
end
```

Fast KD-tree Construction in $O(N \log N)$

It is easy to observe that the algorithm 5 is inefficient mainly due to the computation of the SAH cost for each split candidate requires the N_L and N_R which again triggers an iteration over the triangle list. In [4], Wald and Havran introduced a fast SAH-based kd-tree construction algorithm whose time complexity is $O(N \log N)$. The improved algorithm takes the advantage of the increment of the split plane position, as the split plane "sweeping" over the split candidates, the $N_L$and $N_R$can be updated using a incremental scheme. Consider a split plane $p$in an axis kand its position is denoted as ξ, there is a certain number of triangles lying on the left, right side of it, which is also can be called as the number of end, starting planes respectively. Let us denote the number of starting triangles for the split plane $p$as p^+, the number of end triangles p^-.

Let us consider n split planes candidates $\{ p_0 \}$, along one fixed axis k, and assume that all the planes are sorted in ascending order. For the first split plane candidate p_0 which has the minimum coordinates value, there will be no triangle on the left, all the triangles are lying on the right side of it.

$$N_L^{(0)} = 0 \qquad N_R(0) = N$$

As the split plane "sweeping" from split plane $p_i - 1$ to p_i, the $N_L$and $N_R$will change as follows:

1. The triangles started at $p_i - 1$ will intersect with left voxel V_L.
2. The triangles ended at p_i will no long intersect with right voxel V_R

Therefore the $N_L$and $N_R$can be updated incrementally using the following equations:

$$N_L^i = N_L^{i-1} + p^+ N_R^i = N_R^{i-1} - p^- \tag{4}$$

Follow this updating rule, all the $N_L$and $N_R$of all the split planes candidates can be computed incrementally by iterating over the all the possible split position p_i. Firstly, we need to fix a dimension k, for this k, we go through all the triangles $t$and generate the split candidates associated with t, for each candidate, an "event" will be generated and stored. The "event", e= (e_t, e_ξ, e_k, e_{type}), is actually defined as a data structure which contains four data fields: a reference to the triangle whose bounding box's face defines the split candidate, the coordinate value of split position and the event type. The reference to $t$can be simply an index of it, denoted as e_t, the plane position is a float-point value, denoted as e_ξ, the dimension k, and event type, e_{type}, is an enumeration of several flags to indicate the relation between the triangle referenced in this event and the split plane this event corresponds to, start event and end event. According to the definition, a start event always corresponds to a split position at minimum face of the triangle's bounding box, while end event will be generated by a split plane at the position of the maximum face. Three lists of events against all axises will be generated by iterating over all the triangles, and will be eventually merged into one list E in an interleaved fashion respect to the dimension, this obviously requires the event structure to have an tag to indicate which axis the event corresponds to. Secondly, each of the list of consecutive events for the same axis needs to be sorted, and the comparison of two events e_a and e_b is shown as equation 5. For those events with different positions, they will be sorted by ascending the coordinate values, for those events with same positions, they will be sorted by comparing the event type, that is, the end event will precede the start event. Suppose there are two adjacent triangles which share one vertex, only one split plane candidate will be generated at the vertex, while there will be two events to be stored, one end event references the triangle that is "before" the plane, and one start event references the triangle that is "beyond" the plane. So by counting the start and end events, the $N_L$and $N_R$can be easily determined regarding a certain plane.

$$e_a < e_b = \begin{cases} true & (a_{p,k} < b_{p,k}) \vee ((a_{p,k} = b_{p,k}) \wedge (a_{type} < b_{type})) \\ false & otherwise \end{cases} \tag{5}$$

To compute the $N_L$and $N_R$for each split plane candidates p_i using the sorted event list to find the best one, we consider all dimensions in one loop, for each dimension, a separate $N_{L,k}$, $N_{R,k}$ needs to be stored. For each dimension, we perform the "sweeping" using the above incremental updating scheme. We firstly consider the sequence of the p_i-related events, count the end and start events to determine the number of start and end triangles p_i^+ and p_i^-. Then the $N_L$and $N_R$for the split planes can be maintained and updated by applying the update equations. Once the N_L, $N_R$are determined, the SAH cost is readily to compute and we can find the best split plane by choosing the one with minimum cost. The algorithm is shown as follows:

Algorithm 6: The $O(N)$ algorithm of finding the best SAH split plane

```
pre: E has been sorted. function FindBestPlane($E_{x,y,z}$, $V$, $N$) return $\hat{p}$
begin
    ($\hat{C}_{min}$,$p$) = ($\infty$,$\emptyset$);
    forall $k$ in {$x$, $y$, $z$} do
        {start: all triangles are right side only for each k}
        $N_{L,k} = 0$, $N_{R,k} = N$;
        for $i \leftarrow 0$ to $E_{num}$ do
            $p$= ($E_p[i]$, $E_k[i]$);
            $p^+$= $p^-$= 0;
            {iterate over all plane candidates.}
            while $i < E_{num} \wedge p_k = E_k[i] \wedge p_\xi = E_\xi[i] \wedge E_{type}[i] = -$ do
                inc $p^-$;
                inc $i$;
            while $i < E_{num} \wedge p_k = E_k[i] \wedge p_\xi = E_\xi[i] \wedge E_{type}[i] = +$ do
                inc $p^+$;
                inc $i$;
            $N_{R,k}$ -= $p^-$;
            $C_p$= SAH($p$,$V$,$N_{L,k}$,$N_{R,k}$);
            if $C_p < \hat{C}_{min}$ then
                ($\hat{C}_{min}$,$p$) = ($C_p$,$p$);
            $N_{L,k}$ += $p^+$;
    return $\hat{C}_{min}$;
end
```

After finding the best split plane $\hat{p}$, classifying the triangle list into two sub-lists is another important step in one recursion step. Since the best split in current recursion is found using a sorted events list, we have to find a way to compute two sub-lists of events $E_L$and E_R, and more importantly, to maintain the ascending order of them so that they can become inputs for the following recursion step without performing any sorting. As the input event list $E$is sorted, and the best split plane $\hat{p}$ has been found, if an event is of end type and its position is less than the $\hat{p}$, the triangle this event references must be in the left sub-list of triangle T_L, symmetrically, the triangle referenced in a start event with a greater position than the $\hat{p}$must belongs to the T_R. The remaining triangles should be counted in both $T_L$and T_R. The triangle classification algorithm is shown as following:

Algorithm 7: The triangle classification algorithm in $O(N)$.

```
function ClassifyTriangles($T$,$E$,$\hat{p}$) return $T_L$,$T_R$,$E_L$,$E_R$
begin
    forall $e \in E$ do
        if $e_{type}$=+ $\wedge$ $e_k$=$\hat{p}_k$ $\wedge$ $e_\xi$<$\hat{p}_\xi$ then
            set $flags[e_t].LeftFlagBit$;
        if $e_{type}$=− $\wedge$ $e_k$=$\hat{p}_k$ $\wedge$ $e_\xi$>$\hat{p}_\xi$ then
            set $flags[e_t].RightFlagBit$;
    forall $t \in T$ do
        if $flags[t].LeftFlagBit$ is set then
            $T_L$ appends $t$;
        if $flags[t].RightFlagBit$ is set then
            $T_R$ appends $t$;
    forall $k \in \{x, y, z\}$ do
        forall $e \in E_k$ do
            if $flags[e_t].LeftFlagBit$ then
                $E_L$[k] appends $e$;
            if $flags[e_t].RightFlagBit$ then
                $E_R$[k] appends $e$;
end
```

3.1.3 Optimizations of KD-tree Construction on CPU

Parallelization

There have been many attempts to parallelize the SAH-based KD-tree construction on CPU. Many approaches follow the same parallel pattern [5], [6]. As the builder often starts with a large amount of geometry, the initial phase of the algorithm is a breadth-first top-down hierarchy construction process to organize all the geometry primitives into a few nodes at the top of their hierarchies using a single thread. In this phase the spatial median [7] or geometry count median [5] partitioning strategy are preferred over expensive full SAH cost computation in many known approaches. When the number of nodes at a level meets or exceeds the number of cores, the algorithm switches to depth-first node-parallel construction. Each subtree is assigned to a separate thread and perform the SAH partition independently.

A "nested" parallel algorithm was introduced in [8] providing a pattern utilizing both node-level and geometry-level parallelism. At the top levels of the tree, the major functions in the sequential algorithm have been parallelized, these functions include **FindBestPlane**, **ClassifyTriangles** and **FilterGeometry**.

FindBestPlane The parallelized version of **FindBestPlane** is a process with 3-passes which are **PreScan**, **Push**, **SAHScan**. Given the array of generated events, we first decompose the event list into n contiguous chunks, the memory for each chunk is allocated in each thread. In the **PreScan** phase, each of the $n-1$ thread counts the number of start and end events in its corresponding chunk as there is no need to scan the last chunk, yielding N_L and N_R specifically for the chunk. The **Push** is a prefix sum process that sums up the total N_L and N_R of previous chunk to the total of the current chunk, yielding correct N_L and N_R at the beginning of each chunk. This **Push** phase can be implemented with both sequential and parallel algorithm, however, it is not very beneficial to parallelize it using a multi-core CPU that is only with a few cores. For the **SAHScan** phase, each thread performs an iteration over its corresponding chunk of events, calculating the N_L and N_R and finding the minimum SAH cost within the current chunk. A final sequential reduction is required to get the minimum SAH across all of the n chunks.

ClassifyTriangles

FilterGeometry

Memory Management

Chapter 4

Implementation and Results

Chapter 5

Conclusion

Bibliography

[1] T. L. Kay and J. T. Kajiya, "Ray tracing complex scenes," *SIGGRAPH Comput. Graph.*, vol. 20, pp. 269–278, August 1986.

[2] M. Pharr and G. Humphreys, *Physically Based Rendering: From Theory to Implementation.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

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

[4] I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in o(n log n)," in *IN PROCEEDINGS OF THE 2006 IEEE SYMPOSIUM ON INTERACTIVE RAY TRACING*, pp. 61–70, 2006.

[5] A. S. Maxim Shevtsov and A. Kapustin, "Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes," *EUROGRAPHICS*, vol. 26, pp. 395–404, 2007.

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

[7] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," *ACM Trans. Graph.*, vol. 27, pp. 126:1–126:11, December 2008.

[8] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart, "Parallel sah k-d tree construction," in *Proceedings of the Conference on High Performance Graphics*, HPG '10, (Aire-la-Ville, Switzerland, Switzerland), pp. 77–86, Eurographics Association, 2010.