Implementation of Kd-Trees on the GPU to Achieve Real Time Graphics Processing

Will W. Martin
mart2122@morris.umn.edu

ABSTRACT

This paper examines the parallelization of ray tracing algorithms with the goal of running the whole process on the graphics processing unit (GPU) rather than the central processing unit (CPU). The motivation behind this endeavour is to utilize the massively parallel nature of the GPU. This parallelism allows the construction of 3-dimensional images to take place in real time. To achieve this we focus on how to create and process multi-dimensional tree structures (kd-trees) to model image data. Kd-trees organize multi-dimensional data in a searchable data structure, lending itself to the efficient creation of lighting effects.

Categories and Subject Descriptors

I.3.7 [**Three-Dimensional Graphics and Realism**]: Ray-tracing; E.1 [**DATA STRUCTURES**]: Trees; B.2.4 [**High-Speed Arithmetic**]: Cost/performance

General Terms

Algorithms, Design, Performance

Keywords

Kd-trees, Surface Area Heuristic, Axis Aligned Bounding Box, Ray tracing, GPU, graphics

1. INTRODUCTION

Ray tracing is a technique used to generate high quality graphics. The ray tracing technique excels at accurately rendering realistic shadows, reflections, and refractions by attempting to model physical light rays in reverse. See Figure 1.

Rays of light are shot from the vantage point of the observer through a 2-dimensional *frame buffer* into the scene, reflecting and refracting upon contact with objects in the scene until the rays reach a light source. Each pixel in the frame buffer is then shaded according to which objects the rays interact with. The frame buffer is the 2-dimensional picture that is actually rendered. The ray in Figure 1 intersects with a tree at point p. Examples of images created through the use of ray tracing can be seen in Figures 2 and 11.

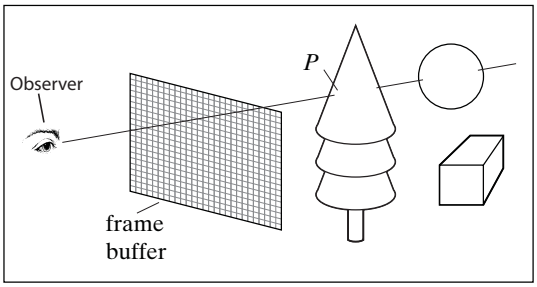


Figure 1: Viewing a single point in a 3-dimensional scene through a pixel by shooting a ray out into the 3-dimensional space. Figure based on [2]

Figure 2: Left (from [1]): Nvidia rendering example using a ray tracing algorithm demonstrating shadows and reflections. Right (from [4]): a 3D scene using ray tracing demonstrating refraction through semi transparent surfaces as well as reflection off of solid shiny surfaces.

The main drawback to ray tracing is the time required to carry out the necessary computation to propagate light rays through the entire image. This process often takes minutes to hours to render a single frame [13]. This has made ray tracing infeasible as a dynamic image rendering technique for generating real time graphics; as a result most ray tracing is done off-line. To achieve real time ray tracing, the image needs to be organized in a data structure that a ray tracing algorithm can traverse quickly. The current data structure of choice is the kd-tree [7, 8, 9, 12, 13, 14]. The basics of the kd-tree as well as their utility is covered in Section 2.

UMM CSci Senior Seminar Conference Morris, MN.

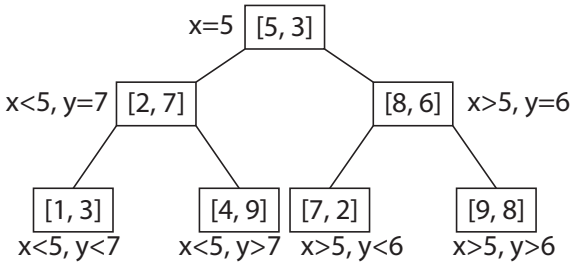


Figure 3: A simplified kd-tree. The nodes have only 2 dimensional data (2 search keys). The root's children are sorted based on their x values, while their children are sorted on y and so on. A graphical representation of this tree can be seen in Figure 4.

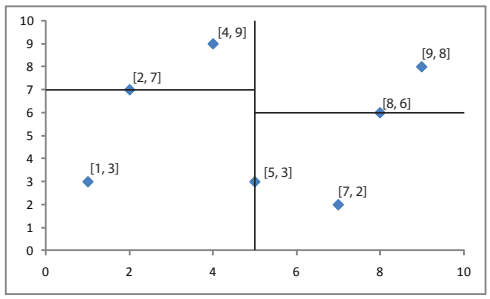


Figure 4: A graphical representation of the 2 dimensional tree shown in Figure 3. The split planes corresponding to each parent node divide up the search space.

Section 3 covers the practicality of running kd-tree generation techniques on the graphics processing unit (GPU) to generate real time graphics as well as provides a comparison between GPU and central processing unit (CPU) algorithms.

2. THE KD-TREE DATA STRUCTURE

The kd-tree is commonly used in graphics processing in the applications of ray tracing and other graphics techniques such as photon mapping [5] and point cloud modeling [14].

2.1 What is a Kd-Tree?

A kd-tree is a binary search tree that represents k-dimensional spatial data. Each coordinate value is a search key much like a single search key in a binary tree. Kd-trees that contain 3-dimensional data such as a 3-dimensional image will have 3 search keys, x, y, and z. Each leaf node in a kd-tree constructed for ray tracing also contains a list of graphics primitives, usually triangles. These triangles are the building blocks of the scene. Every object in the scene is made up of a multitude of small triangles. The kd-tree cycles through each available dimension, or search key, in the data being modeled, layer by layer in a regular pattern. All nodes on the same level are sorted by the same search key as illustrated in Figures 3 and 8. In Figure 3 a 2-dimensional tree has been constructed, instead of using polygons, points are used. The root node is chosen and all other nodes are sorted on x and brought down to the next level of the tree. Root nodes for both left and right sub trees are chosen and all nodes belonging to that sub tree are sorted upon y and brought down to the next level. This level would sort on x again if there was more data.

A graphical representation of Figure 3 can be seen in Figure 4. Each non-leaf node corresponds to a *split plane*. A split plane divides the area of the dimensional space along a line running parallel with the axis to which the level of the tree belongs, as illustrated in Figure 4. Our ability to split the data in sections graphically will be useful during tree traversal while shooting a ray though a scene. This idea will be covered more in Section 2.2.1.

2.2 Kd-Tree Tools

In order to understand how to parallelize the construction process we must first describe the tools used in the construction of a basic kd-tree. This section discusses the tools used

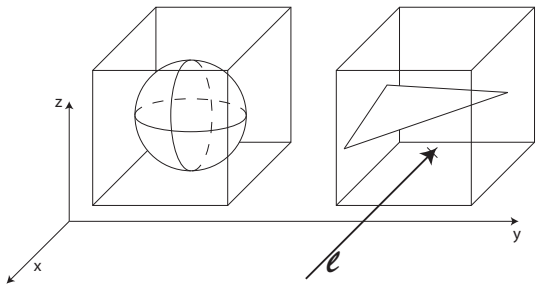


Figure 5: Two 3-dimensional objects surrounded by axis aligned bounding boxes. Based on [3]

by Shiue et. al [10] to achieve real time kd-tree construction on the GPU. Axis aligned bounding boxes and construction heuristics will be covered, as they allow us to generate an efficient kd-tree optimized for ray tracing.

2.2.1 *Axis-aligned bounding box*

An axis-aligned bounding box (AABB) is a k-dimensional box enclosing graphical elements within the kd-tree. The bounding box can be used to estimate the physical space taken up by an object or collection of objects. They are axis aligned, meaning that their bounding surfaces are parallel to the axes, making many computations performed on these bounding boxes simpler. AABBs can be used to simplify determining if a ray or line will intersect an object. An example of this is given in Figure 5 using two 3-dimensional bounding boxes. The main utility of bounding boxes can be demonstrated by looking at line l. We wish to know which objects, if any, this line intersects. This line will intersect the front and rear faces of the right hand AABB, therefore we know we need to check to see what the line intersects inside the right hand AABB. However, because we know that this line will never intersect the AABB on the left, we never need to check the contents of that AABB, saving computation time. AABBs can also be used hierarchically, where AABBs contain other AABBs as well as geometric primitives as demonstrated in Figure 6.

The split planes in a 3-dimensional kd-tree's graphical representation create AABBs. Each splitting plane can be

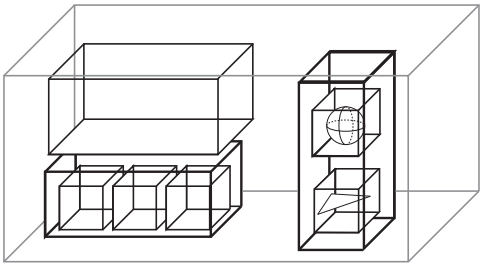


Figure 6: AABB hierarchy based on [3], showing how AABBs can enclose other AABBs in a hierarchical manor.

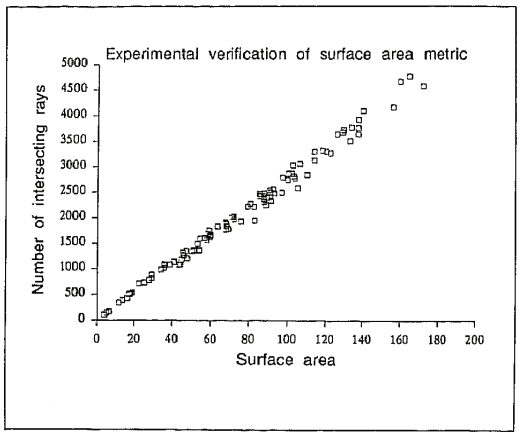


Figure 7: This graph (taken from [6]) demonstrates the relationship between the number of intersecting rays with bounding boxs and the surface area of those bounding boxs using a ray tracing algorithm. This clearly shows a linear relationship between the two.

thought of as a side of an AABB. When we search through the data in a kd-tree, if the tree is balanced, the search space is roughly cut in half with every comparison. Graphically the image is cut into sections, allowing us to rule out sections, in much the same way bounding boxes do. How much of the image is cut out by any one comparison is dictated by how the nodes are distributed throughout the tree.

2.2.2 *Surface Area Heuristic*

The surface area heuristic (SAH) can be used to better organize graphical data in a tree structure. The SAH helps evenly distribute the split planes across the image to make each comparison in the search tree divide search space as effectively as possible from a graphical perspective as well as a data structure perspective. This is based on the idea that the number of rays likely to intersect a convex object is roughly proportional to its surface area, a claim supported by a number of tests done by MacDonald and Booth [6]. They used a ray tracing algorithm to traverse the tree and collected data on which AABBs were hit by a ray. The results can be seen in Figure 7, which shows a strong linear relationship between the number of rays that intersected

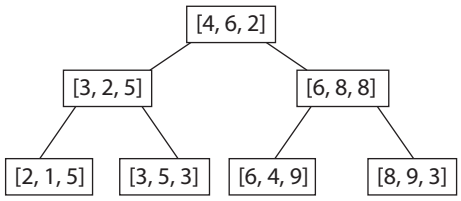


Figure 8: A representation of a three dimensional kd-tree. The graphical representation can be seen in Figure 9

.

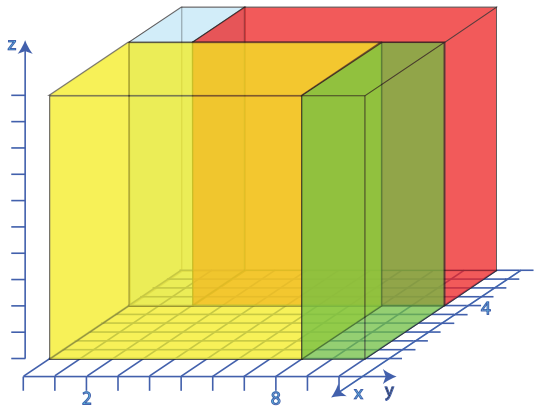


Figure 9: A graphical representation of the kd-tree shown in Figure 8. The yellow and green (front) are the right hand sub trees, the blue and red (rear) are the left hand sub trees.

with a bounding box vs. the bounding box's surface area.

The SAH gives every kd-tree a *cost*. The calculation of a tree's cost is based upon the ideas behind bounding boxes. If we start out with the kd-tree shown in Figure 8 we will get the graphical representation shown in Figure 9. Assume the scene in Figure 9 is isolated in a $10 \times 10 \times 10$ bounding box. We can see that the root node's children give us bounding boxes of sizes $4 \times 10 \times 10$ (red and blue) and $6 \times 10 \times 10$ (yellow and green). Their children in turn provide more split planes, each generating smaller bounding boxes. We will use the surface area of these bounding boxes to calculate the cost of the node that defines these bounding boxes.

The surface area of an AABB, like any rectangular prism with side lengths l, m, n, can be calculated:

$$\boxed{\begin{aligned} SA &= 2ln + 2lm + 2mn \\ &= 2\left((l+m)\,n + lm\right) \end{aligned}} \tag{1}$$

We can drop the factor of two in this equation because this equation is always divided by itself to form a ratio as in Equation 2.

$$\boxed{\bar{C} = k\left(\frac{SA_{\text{self}}}{SA_{\text{root}}}\right)} \tag{2}$$

The *average cost* of each node can be calculated using

$$SA_{(4,6,2)} = ((10+10)10 + 10 \times 10) = 300$$
$$SA_{(3,2,5)} = ((4+10)10 + 4 \times 10) = 180$$
$$SA_{(6,8,8)} = ((6+10)10 + 4 \times 10) = 220$$
$$SA_{(2,1,5)} = ((4+2)10 + 4 \times 2) = 88$$
$$SA_{(3,5,3)} = ((4+8)10 + 4 \times 8) = 152$$
$$SA_{(6,4,9)} = ((6+8)10 + 6 \times 8) = 188$$
$$SA_{(8,9,3)} = ((6+2)10 + 6 \times 2) = 92$$

Table 1: Calculations of surface area for each node in Figure 8

$$\bar{C}_{(4,6,2)} = 2\left(\frac{300}{300}\right)$$
$$\bar{C}_{(3,2,5)} = 2\left(\frac{180}{300}\right) \bar{C}_{(6,8,8)} = 2\left(\frac{220}{300}\right)$$
$$\bar{C}_{(2,1,5)} = 0\left(\frac{88}{300}\right) \bar{C}_{(3,5,3)} = 0\left(\frac{152}{300}\right)$$
$$\bar{C}_{(6,4,9)} = 0\left(\frac{188}{300}\right) \bar{C}_{(8,9,3)} = 0\left(\frac{92}{300}\right)$$

Table 2: The above table shows the average cost of each node in Figure 8 based upon the surface areas calculated in Table 1.

Equation 2 where SA_{root} is the surface area of the root node, SA_{self} is the surface area of the AABB containing the current node, and k is the number of children that belong to the current node.

Note that each leaf node has a cost of zero because k will be zero for these nodes. The root node will have a cost of k because we make it its own parent node, making the ratio of the surface areas equal to one. To calculate the cost of the entire tree we sum up the costs of each node in N, where N is the set of all nodes:

$$\boxed{\sum_{i \epsilon N} \bar{C}(i)} \tag{3}$$

The surface area calculations of each node in Figure 8 are shown in Table 1. The costs are then calculated in Table 2 using those surface areas. Summing up the costs for each node calculated in Table 2 we find the cost for the entire tree:

$$\begin{aligned} \sum_{i \epsilon N} \bar{C}(i) &= 2 \times \left(1 + \frac{18}{30} + \frac{22}{30}\right) \\ &\quad + 0 \times \left(\frac{88}{300} + \frac{152}{300} + \frac{188}{300} + \frac{92}{300}\right) \\ &= 4.67 \end{aligned} \tag{4}$$

The SAH strives to produce a kd tree with the smallest cost. This ensures that the probability of a ray traversing the tree will have close to an equal chance of traversing either child of any sub tree, thus utilizing the whole tree effectively. The cost calculation of a tree is theoretically run in $O(n)$ where n is the number of nodes in the tree. The calculation must be carried out for each *splitting plane candidate*. Each node has a variety of places it can be split. Each one of these split possibilities are called splitting plane candidates. Running the SAH on each of them allows us to pick the splitting plane candidate with the lowest cost. With this in mind, we can set up the following construction algorithm for kd-trees:

1. Calculate the SAH costs for all splitting plane candidates using Equation 3.
2. Split the tree on the lowest cost candidate.
3. Distribute the remaining graphical elements to the proper sides of the new sub tree.
4. Recursively traverse both sub-trees.

Algorithms that create kd-trees for the purposes of ray tracing only put graphical data in the leaf nodes. This changes our simplified model of the SAH slightly. In Equation 2 we simply used the number of children k as the cost of the node, but now we must differentiate between leaf and non-leaf nodes. Instead of multiplying the surface area ratio by k we will multiply it by K_T or K_I depending on whether the node will add the cost of traversing to a child node, or the cost of checking the a leaf node for intersections, respectively. This changes Equation 3 to Equation 5 with two summations, one over I which is a list of all non-leaf or internal nodes and the other over L which is a list of all leaf nodes. i_{parent} is the parent of node i and likewise j_{parent} is the parent of node j. This equation was also used by Wald [12], and MacDonald and Booth [6].

$$\sum_{i \epsilon I} \frac{SA(i)}{SA(i_{\text{parent}})} K_T + \sum_{j \epsilon L} \frac{SA(j)}{SA(j_{\text{parent}})} K_I \tag{5}$$

An approximation of this equation is provided by Wald [12] and used by Zhou et al. [14], given in Equation 6.

$$\boxed{SAH[x] = C_{ts} + \frac{C_L[x] SA_L[x]}{SA_{\text{parent}}} + \frac{C_R[x] SA_R[x]}{SA_{\text{parent}}}} \tag{6}$$

Here C_{ts} is a constant that represents the cost of traversing a node, C_L and C_R are the costs of the left child and right child respectively, for a split plane at position x, SA_L and SA_R are the surface areas of the left child and right child respectively given a split plane at position x, and SA_{parent} is the surface area of the node being split.

To avoid having to build the entire left and right sub-trees to find the total cost of split planes we have not begun to consider yet, we can approximate C_L and C_R by the number of graphical elements that will reside within the left and right nodes being created. This assumes that the left and right child of the node being split are both leaf nodes and makes Equation 6 a greedy approximation, overestimating the cost of any given split [14].

2.2.3 Empty space minimizing and split median

Another method of determining splitting planes employed by Zhou et al. [14] used a combination of median splitting and empty space minimizing. Empty space minimizing is a technique that looks at all graphical data within a node to be split. If a node has no graphical data from an edge to a predetermined threshold C_e as seen in Figure 10(a), the

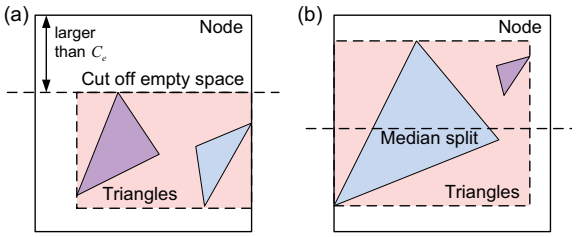


Figure 10: (Taken from [14]) (a) cut off empty space; (b) spatial median split.

node will be split to separate the empty space into its own node.

Figure 10(b) illustrates a median split. This splitting technique takes the longest axis in the node and splits it in half. The graphical data (triangle in this case) has to be *clipped* or cut into multiple triangles to be filtered down through the tree without overlapping the median split. Both these techniques are useful because they require no explicit cost calculations. Each node is split in an effort to minimize white space within bounding boxes, or to split nodes in half in a way that keeps them as square or cubic as possible.

3. GPU VS. CPU

Heavily computational tasks such as construction of a kd-tree used in both static and dynamic off-line ray tracing algorithms have typically been run on the central processing unit (CPU) where the number of tasks that can be run in parallel is restricted to how many cores the CPU has. Static and dynamic on-line ray tracing algorithms have the potential to be constructed on the graphics processing unit (GPU), which is a highly parallel processing unit and can support many more asynchronous tasks in parallel.

3.1 Benefits Provided by The Threading Abilities of the GPU

The CPU is a robust blunt tool for doing computationally expensive tasks. Today, instead of making a single core CPU faster, more cores are placed on a single CPU, allowing the CPU to run multiple processes at once. Most retail computers at the time of this paper have 4 cores on them. The graphics processors on computers, however, have been designed to do computationally inexpensive tasks in an extremely parallel manner, driven by the gaming industry. The modern GPU has the ability to spin up 10^3 to 10^4 threads to achieve optimal performance [14].

3.2 Running Kd-Tree construction on the GPU

The main method used in [14] makes a root node for the start of their kd-tree, and puts all triangles into the root node. The AABB for each triangle is calculated in parallel on the GPU, then the list of triangles within their AABB's are passed into the *large node stage*. We know that if we were to employ the SAH we would want to use the approximation to speed up our node splitting. This approximation, however, assumes that the children of the node being split are both leaves. This assumption is usually always wrong for nodes with large numbers of triangles such as the upper level nodes. The large node stage uses the empty space minimizing heuristic when applicable. When a node does not meet the criteria of the empty space minimizing heuristic, the split median is used. Once a node is split the triangles are distributed to the child nodes. These nodes are then split. Any triangles that overlap are clipped and these new clipped triangles are distributed to the children as well. The new set of child nodes is then processed the same way in parallel on the GPU until the number of triangles in a leaf node drops below 64. Once the number of triangles in a node drops below 64 they are considered small nodes. These nodes are then processed on the GPU using the approximated SAH in Equation 6. The splitting plane candidates used in the approximation are restricted to the faces of the AABBs initially created in the main method for each triangle. The triangles can then be sorted into the two new child nodes. The triangles that overlap both nodes are brought down into both children to cut down on memory reallocation instead of creating two new smaller triangles. The method used in [14] determines the optimal split plane in $O(n)$ time and the sorting of triangles into child nodes can be done in $O(1)$ time. [14]

Scene	Off-line CPU builder			GPU builder		
	T_{tree}	T_{trace}	SAH	T_{tree}	T_{trace}	SAH
Fig.11(a)	0.085s	0.022s	79.0	0.012s	0.018s	67.9
Fig.11(b)	0.108s	0.109s	76.6	0.017s	0.108s	38.3
Fig.11(c)	0.487s	0.165s	68.6	0.039s	0.157s	59.7
Fig.11(d)	0.559s	0.226s	49.6	0.053s	0.207s	77.8
Fig.11(e)	1.226s	0.087s	74.4	0.077s	0.078s	94.6
Fig.11(f)	1.354s	0.027s	124.2	0.093s	0.025s	193.9

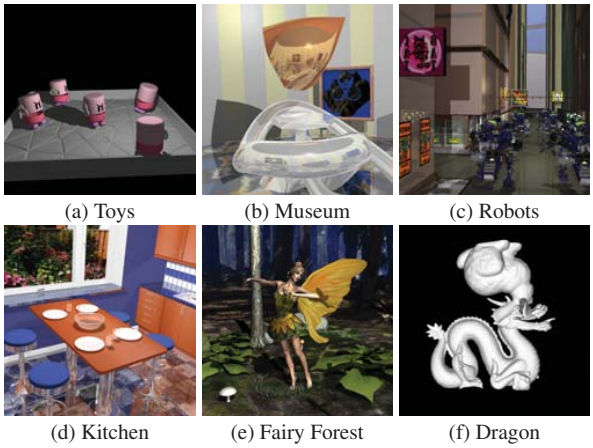
Table 3: Comparing the kd-tree construction time, ray tracing time, and real SAH cost as calculated with Equation 5 after tree construction of a off-line CPU builder and the GPU builder. All images created were 1024×1024.

The SAH and split median both strive to create cubic nodes. The SAH strives to create a node with the smallest surface area. A rectangular prism of area A has the smallest surface area when its sides are of equal length. The split median always cuts the node in half on the longest side ensuring nodes don't get too rectangular. The SAH, however, is much more computationally expensive even with the approximation, taking into account all primitives in a given node. This minimizes the chance of splitting a primitive in half, however it is very time consuming for nodes containing large amounts of graphical primitives.

3.3 Results

Figure 11 contains 6 images used in [14] to test the kd-tree building technique described above. The results of this test can be seen in Table 3. The GPU tree builder takes considerably less time to create a kd-tree. The times it takes to traverse the tree with the CPU built kd-trees closely resemble those of the GPU built kd-trees. The SAH cost of both trees were calculated using Equation 5 after the trees were built. Using a combination of free space minimizing, split median and our greedy approximation of an SAH results in trees of acceptable cost.

K. Zhou et al. [14] also restricted the number of processors the GPU could utilize to see how the algorithm scales. A graph of this data can be seen in Figure 12; The number of triangles in the image directly correlates with the gracefulness the GPU algorithm's ability to scale. The dragon picture, Figure 2(f), scales by a factor of 6 while the toys picture, Figure 2(a), only scales by a factor of 3. Note the



(a) Toys (b) Museum (c) Robots

(d) Kitchen (e) Fairy Forest (f) Dragon

Figure 11: Test scenes (Taken from [14]) for kd-tree construction and ray-tracing. (a) 11K triangles, 1 light; (b) 27K triangles, 2 lights; (c) 71K triangles, 3 lights; (d) 111K triangles, 6 lights; (e) 178K triangles, 2 lights; (f) 252K triangles, 1 light.

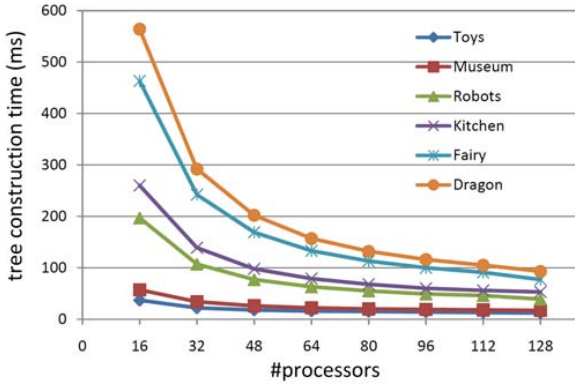


Figure 12: Tree construction times(taken from [14]) (in ms) incrementing the number of processors available on the GPU

dragon was constructed with about 23 times as many triangles as the toys picture.[14]

The capabilities of this algorithm can be seen in Table 4. The two algorithms that the GPU builder was tested against come from [11] and [9]; [11] used an AMD Opteron 2.6GHz CPU and [9] used a Dual Intel Core2 Duo 3.0GHz CPU (4 cores). We can see the GPU builder outperforms both online CPU kd-tree builder/ray tracers on both scenes. The GPU builder shows promise in creating realistic looking graphics on dynamic scenes.[14]

4. CONCLUSIONS

The whole process of ray tracing from generating the scene to rendering an image can be run on the GPU. This allows high quality 3-dimensional images to be created in real time as seen in [14]. These techniques can be used in graphics, not only in offline tasks like pixar animations, but online tasks like video games and real time 3-dimensional image rendering.

Scene	Wald	Shevtsvo	GPU builder
Fig. 11(a)	10.5fps	23.5fps	32.0fps
Fig. 11(b)	n/a	n/a	8.00fps
Fig. 11(c)	n/a	n/a	4.96fps
Fig. 11(d)	n/a	n/a	4.84fps
Fig. 11(e)	2.30fps	5.84fps	6.40fps
Fig. 11(f)	n/a	n/a	8.85fps

Table 4: (Taken from [14]) The GPU builder compared with other CPU on-line kd-tree builders and ray-tracers.

5. REFERENCES

[1] R. Austinat. GPU ray tracing: New pictures reveal current possibilities. `http://www.pcgameshardware.com/aid,661682/GPU-ray-tracing-New-pictures-reveal-current-possibilities/News/`, Sept. 2008.

[2] F. S. Hill, Jr. *Computer Graphics Using Open GL*. Macmillan Publishing Company, second edition.

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

[4] B. Grass. X3D class examples of ray tracing. `https://www.movesinstitute.org/pipermail/x3d-courses/attachments/20110428/0943b920/attachment-0004.jpg`, Apr. 2011.

[5] W. Jarosz, H. W. Jensen, and C. Donner. Advanced global illumination using photon mapping. In *ACM SIGGRAPH 2008 classes*, SIGGRAPH '08, pages 2:1–2:112, New York, NY, USA, 2008. ACM.

[6] J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6:153–166, 1990. 10.1007/BF01911006.

[7] S. Popov, J. Gunther, H.-P. Seidel, and P. Slusallek. Experiences with streaming construction of SAH kd-trees. In *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 89 –94, Sept. 2006.

[8] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. *ACM Trans. Graph.*, 24:1176–1185, July 2005.

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

[10] L.-J. Shiue, I. Jones, and J. Peters. A realtime GPU subdivision kernel. *ACM Trans. Graph.*, 24:1010–1015, July 2005.

[11] I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26, January 2007.

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

[13] I. Wald, J. T. Purcell, J. Schmittler, C. Benthin, and P. Slusallek. Realtime ray tracing and its use for interactive global illumination. *Eurographics State of the Art Reports*, 2003.

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