INF584 – Image Synthesis, Final Project Report

Khalig Aghakarimov
khalig.aghakarimov@polytechnique.edu

April 1, 2021

Abstract

Rendering or image synthesis is the process of generating a photorealistic or non-photorealistic image from a 2D or 3D model by means of a computer program. Rendering has widely spread applications in different fields like architecture, video games, simulators, movie and TV visual effects. Besides, there are many existing algorithms that are researched to achieve rendering, such as rasterization, ray tracing or ray casting. Ray Tracing is a technique for generating an image by tracing the path of light as pixels in an image plane and simulating the effects of its encounters with virtual objects. This project report discusses a paper about a specific rendering method called "Efficient Divide-And-Conquer Ray Tracing using Ray Sampling" and gives the details about its implementation.

Keywords: Image Synthesis, Rendering, Ray Tracing, Divide-and-conquer algorithms, K-binning

1 Introduction

Ray Tracing is a very powerful technique that is capable of generating a high degree of a visual realism, although it usually becomes costly in terms of the time complexity. However, to avoid this issue of a big computational time, a usage of acceleration data structures is recently proposed. To name a few, Grid, k-d tree, Bounding Volume Hierarchy (BVH) can be used depending on the problem. As the name suggests, the purpose of using such structures is to speed up the process of Ray Tracing. But they usually require an extensive memory and this memory is not determined before construction. Thus, there are also different algorithms that are specifically designed to resolve such problems. Divide-And-Conquer Ray Tracing or DACRT is one of them and it mainly tries to solve intersection problems between a large number of rays and primitives by recursively subdividing the problem size until it can be easily solved. The peculiarity about DACRT is that it traces rays and constructs acceleration data structures at the same time. Besides, it requires no storage cost for acceleration data structures and the required memory is minimal and deterministic. The working mechanism of Divide-And-Conquer Ray Tracing is described in the following figure:

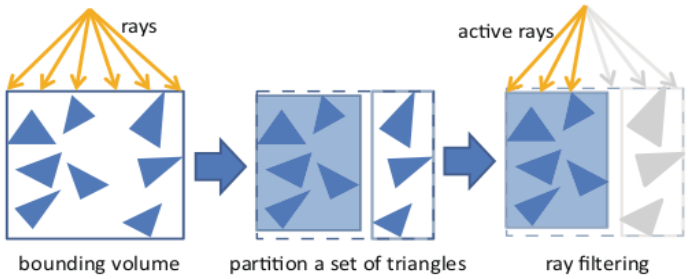


Figure 1: DACRT mechanism, figure from the paper [1]

As described also in *Figure 1*, a set of triangles in the bounding volume are partitioned into two subsets of triangles. Similarly, a set of all rays are partitioned based on the newly constructed bounding volumes. In other words, the rays are selected based on the bounding volume they are involved. This is the last

stage called ray filtering in the figure above. DACRT continues this way until it reaches a sufficiently small number of rays or triangles in the recursive process. In the following section, you will get acquainted with the paper that covers one of the specific DACRT algorithms which is called "Efficient Divide-And-Conquer Ray Tracing using Ray Sampling" by Kosuke Nabata, Kei Iwasaki and Yoshinori Dobashi. [1]

2 Paper overview

There have been a lot of propositions and related work about the Divide-And-Conquer Ray Tracing. The problem with the previous DACRT methods is that the partitioning of the triangles occurs as a uniform distribution of rays which can become inefficient for a concentrated distribution. In addition to this, ray filtering may not always reduce the number of active rays efficiently. For instance, *Figure 2* illustrates such a case where only one ray is reduced. Thirdly, ray filtering is usually a computationally expensive process.

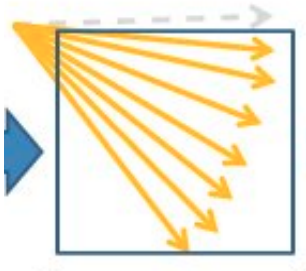


Figure 2: Inefficient scenario in ray filtering

However, the current extensive approach of DACRT described in paper [1] involves two new aspects which makes it different from the others:

1. Ray Sampling: it serves to accelerate ray tracing and increase the performance of tracing different sort of rays.

2. A new cost estimator: it avoids the inefficient subdivision of intersection scenarios for which the number of rays is not necessarily reduced (the cases similar to *Figure 2*). The new equation is simple as well as efficient.

For performing DACRT, we need a set of active rays, a set of triangles and a bounding volume. As already mentioned, the main idea behind DACRT is the subdivision of the bounding volume until we reach a leaf node. We stop whenever the number of triangles or the number of active rays are sufficiently small to subdivide further.
As a starting point, we introduce the notion of K-binning which is described in [2]. For each recursive step, the bounding volume is divided into K partitions or so-called bins. The algorithmic details will be provided in the next section. After K-binning, we can summarize the approach itself in 4 following stages:

1. *Ray Sampling*: If the number of active rays is large enough, then 100 rays are sampled from the active rays, which will themselves be called sampled rays. The sampled rays are selected randomly when the number of active rays is more than 1000.

2. *Partitioning using Cost function*: Since we have all possible subdivision information after applying K-binning, we are trying to find the best subdivision among all (K-1) cases using the cost function below:

$$C = C_T + C_I(\alpha_{L,j}N_{L,j} + \alpha_{R,j}N_{R,j}) \tag{1}$$

   where C_T and C_I stand for constant costs of ray-bounding volume and ray-triangle intersections, $\alpha_{L,j}$ and $\alpha_{R,j}$ are the intersection ratios for the bins derived during the K-binning process and $N_{L,j}$ and $N_{R,j}$ are the numbers of the triangles in the left and right sub bounding volumes. The partitioning with the minimal cost is selected for the current recursive step.

3. *Traversal order*: This is a very simple step where the determination of the traversal order is performed based on the bounding volume with larger number of closer sample rays first.

4. *Skip Ray Filtering*: In the case when the inefficient scenario happens in the ray filtering similar to the *Figure 2*, we do not apply the filtering and take all the active rays. The skipping criterion for a non-leaf node of binary bounding volume is:

$$\alpha > 0.5 \tag{2}$$

where α stands for the intersection ratio.

3 Algorithmic implementation

The implementation of the method described in the paper is realized via C++ language.

3.1 K-binning

Binning is performed for each recursive step. The first thing is about determining the axis based on which we should partition and construct K bins. This usually becomes the longest axis for the given bounding volume. After determining the axis, we are filling the following necessary information for each bin j that will be used later:

- V_{left}, V_{right}: left and right bounding volumes. These play the role of child nodes, which are constructed based on the upper right and lower left triangle points to ensure that every triangle is inside the given bounding volume.

- T_{left}, T_{right}: triangles in the left and the right bounding volumes. The detection to which bounding volume the current triangle belongs to is just about the comparison of the binning axis position with the barycenter coordinate of the triangle for the same axis:

$$P_{bounding\_axis} > barycenter_{bounding\_axis} \tag{3}$$

where P stands for the 3D point where the current binning is occuring and barycenter is the 3D barycenter of the current triangle. In this case, the triangle corresponds to the right bounding volume and otherwise to the left one.

3.2 Sampling

Ray Sampling is achieved via the built-in function of the C++ library, more precisely *std::sample*. It selects 100 elements from the given set of all rays (without replacement) such that each possible ray has an equal probability of appearance [3]. Randomness is achieved using the random number generator *std::mt19937*.

3.3 Axis-aligned Bounding Box (AABB)

For the sake of an easy implementation, I decided to use the Axis-aligned Bounding Box for representing a bounding volume. It is nothing else than a simple cube that covers a certain region in a 3D space. The two parameters: the left lower and the right upper vertices are enough to represent an AABB. The triangles happen to be inside this bounding box. In our implementation, there is one more method which is related to AABB and it basically finds the entry and exit points of a given ray hitting the bounding box.

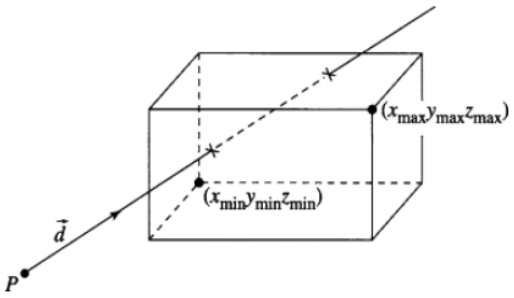
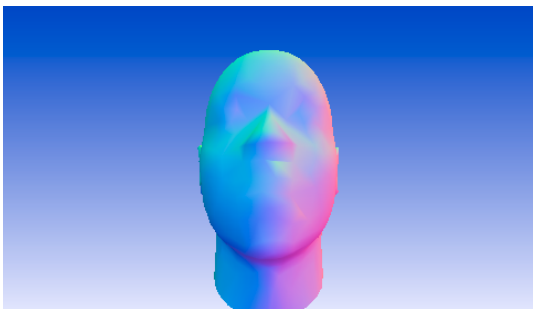


Figure 3: AABB visualization

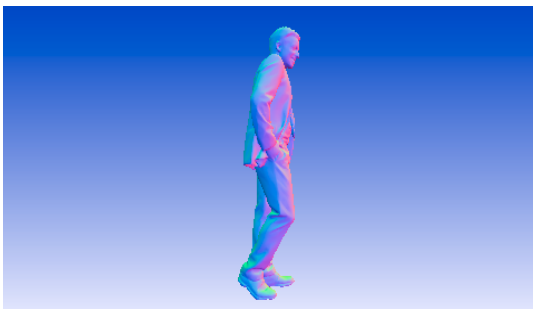
4 Computational Results

The implemented program is tested on different meshes and triangles are used in those meshes as primitives. The code was run on Intel® Core™ i5-10210U Processor 1.6 GHz (6M Cache, up to 4.2 GHz, 4 cores) / Intel® UHD Graphics 620, NVIDIA® GeForce® MX250, 2GB GDDR5 and it runs smoothly with no lag. Below you can see some rendering results of different mesh trials. They are obtained using:

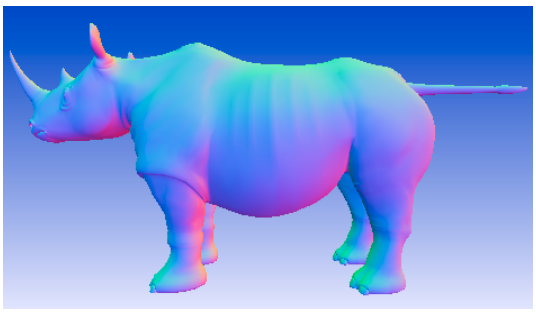
- K-binning of K=5,
- stopping criterion of 10 rays,
- stopping criterion of 20 triangles.



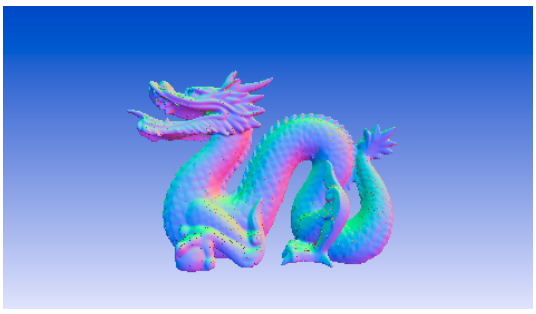
(a) Face mesh



(b) Denis mesh



(a) Rhino mesh



(b) Dragon mesh

The following bar chart gives an intuition about the efficient performance of our method (Note that the running time is measured in minutes):

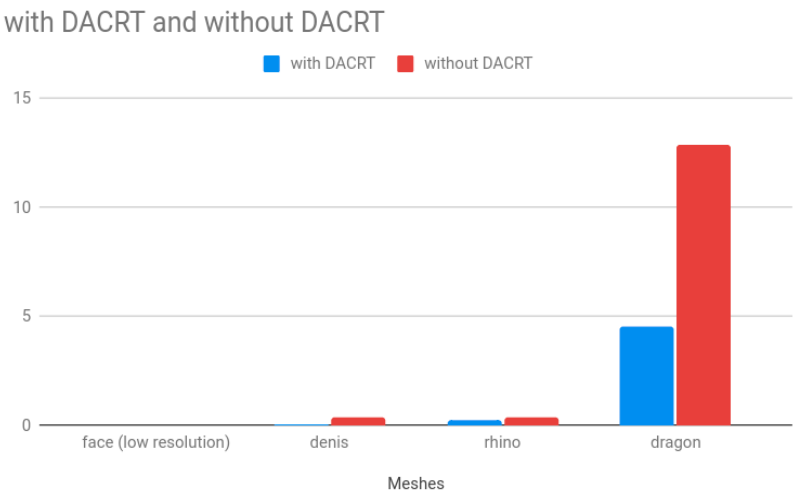


Figure 6: Time performance comparison on meshes

After looking at the bar chart, it becomes obvious that the approach of the paper is more efficient in terms of the performance since it takes less time to run the program. The bar chart does not clearly give information about the face mesh, because it takes only a few milliseconds to run. Thus, for more accurate numerical results, one can have a look at the next table:

Meshes	**Running time (with DACRT)**	**Running time (without DACRT)**
face (low resolution)	0.0048	0.01
denis	0.0342	0.403
rhino	0.257	0.355
dragon	4.56	12.847

Table 1: Numerical results

5 Code

The code is available publicly in the following GitHub link.

References

[1] *Efficient DACRT using Ray Sampling*. URL: http://web.wakayama-u.ac.jp/~iwasaki/project/dacrt/hpg2013final.pdf.

[2] *On fast Construction of SAH-based Bounding Volume Hierarchies*. URL: https://dl.acm.org/doi/10.1109/RT.2007.4342588.

[3] *Sampling in C++*. URL: https://en.cppreference.com/w/cpp/algorithm/sample.