Fundamental Ray Tracing Project Based on OpenGL with acceleration

Zhenghao You

Computer science, Wuhan University, Wuhan, 430072, China

2020302111319@whu.edu.cn

ABSTRACT: In 3D computer graphics, ray tracing is an essential technique for modeling light transport for use in a wide variety of rendering algorithms for generating digital images. And it is also an important subject for computer science students who want to discover more in Computer Graphics and image rendering. In this project the author implemented a fundamental light tracing system based on C++ in visual studio. It is able to render a delicate image with an input of a nnf model file, and output a ppm file, as a final assignment of a student computer graphics research program. This paper is divided into two chapters. The first chapter is about some fundamental theories that are critical to light tracing. The second chapter is about detailed design and implementation process of the program. And the final chapter is about some actual problems and effective solutions.

Keywords: light tracing, rendering, C++

1. Introduction

Ray tracing is a way to manifest objects realistically, which was proposed by Appel in 1968. It is often used in rendering mirror effects, and is an essential algorithm in Computer Graphics. According to the algorithm, light rays are ejected from a source and deflected as they hit or pass over a surface in accordance with the laws of physical optics. Eventually, the rays enter the virtual camera and the picture is produced. However, realistic as it is, the biggest bottleneck of the algorithm is that the extremely huge amount of calculation requires much longer time to render a picture, and demands the highest level of computer hardware. That's why light tracing is now used in movie production and image production, but in game production, it is always hard to achieve ultimate image quality.

In this work, apart from fundamental functions of ray tracing, an effective acceleration process is also implemented, in order to try a way to overcome the shortcoming. The essay gives a detailed introduction of the system design, and the accelerate effect.

2. Fundamental Theories

2.1 Phong BRDF Model

The project chose Phong BRDF model as shading method, as it requires relatively less calculation, which is beneficial for reducing coding complexity and overall running time, yet the render effect is vivid enough for this elementary program [1].

Phong BRDF model divided shading color into three components: ambient component, diffuse component and specular component. The ambient component stands for the basic color of an object in

the environment (however, in the program, all object models are abstract, so this component is generally overlooked).The diffuse component calculates diffuse reflect from the light source. And the specular component calculates specular reflect light from the light source.

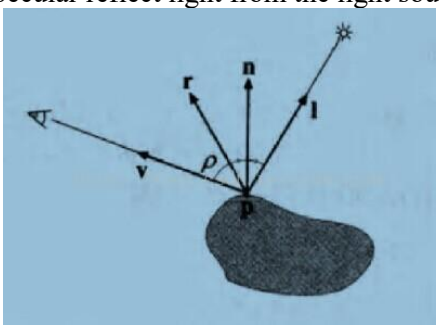

Figure 1 Brief demonstration of Phong BRDF

As the picture above demonstrates, in Phong BRDF there are four vectors: l for the direction of light, n for normal of the surface, r for a vector symmetrical about the normal to l, and v for the direction of the viewer. Calculation of diffuse and specular part is as equation (1) and equation (2) show.

$Diffuse = kd \times light \times \max(\vec{l} \cdot \vec{n}, 0)$ (1), $Specular = ks \times light \times \max(\vec{r} \cdot \vec{v}, 0)^p$ (2)

So the overall color should be diffuse color plus specular color.

2.2 Color Reflection

In the natural world, lights consist of infinite light waves with different wave length, in other word, a beam of light consists of infinite different colors. And an object appears to have a certain color simply because it reflects some certain colors more than others. So when a light hit something, just multiply the amount of each corresponding color in the light and in the object's color, and mix them together, to be the color of the reflection light. In OpenGL, every color consists of a certain amount of red, green, and blue light, each range from 0 to 1 [2-4]. In this project each color is represented by a 3-dimensional vector. Therefore, the reflection color is the dot product of the light's color and the object's color.

2.3 Basic Concept of Ray Tracing [5]

Different from how light spreads in the real word, light tracing requires to send a ray from where the camera is, through each pixel in the screen set in the scene, into the scene to see if it hits something and calculate its color.

The process is repeated one by on until all pixels get a RGB value [2], and output these pixels to form the rendered image.

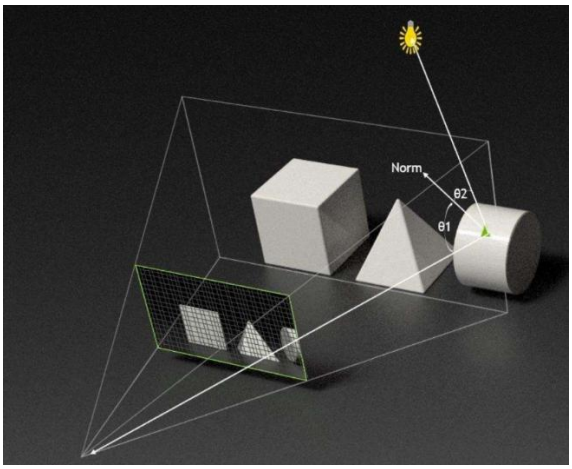

Figure 2 Brief concept of light tracing

3.Detailed Design

3.1 Classes Design

3.1.1 Ray

A ray has a starting point, and a direction, two three dimensional vectors.

3.1.2 Fill

Fill class stands for a way to fill a point on a surface. It includes the color of the point, Kd and Ks attribute for Phong BRDF, and shine attribute as the specular exponent, which is also known as roughness.

3.1.3 Surface

Surface class is a virtual class symbolizing different surfaces of different objects in the scene. It contains several virtual functions: intersect function judging if a surface intersects with a ray, boxIntersect function checking if a surface intersects with a box, checkPosition function is used in building the main box, which will be discussed later.

Surface class has two subclasses, sphere class and triangle class. Sphere class has a center point and a radius value, triangle class has the location of its three points. Both subclasses have the functions of their parent class.

3.1.4 HitRecord

HitRecord class is a class used to record the object, specifically the point hit on an object. It records the position, fill information, normal vector, the direction of the ray, and its distance to the viewpoint. These data are further used in shading process.

3.1.5 Light

Light class simply store the position and color of the light source.

3.1.6 Box

Box class is used in bounding-box acceleration process. These virtual boxes encompass all the scene and divide it into several areas. As the boxes are placed horizontally, recording six coordinates of each face is enough. Besides, a vector of integers of the index of objects inside it.

The class has a intersect function to check if a ray hits a box, and a separate function to divide a bigger box into eight smaller boxes, in order to build a tree structure, which will be discussed later.

3.2 Basic Functions

3.2.1 Triangle-Ray intersection [6]

In this function is used to check if a ray intersects a triangle. As the ray is represented by the original coordinate plus t multiples the direction, so checking intersection is checking if there is a valid t.

Firstly, check if the direction of the ray is parallel with the triangle (perpendicular with the normal), if so, return false.

Secondly, if not parallel, that means the ray is definitely hitting somewhere on the plane where the triangle is located (point p). In this case, vector ap/bp/cp should be perpendicular to the normal, then get the t value.

Then, examine if p is inside the triangle. If it is, vector ap and ac are at the same side of ab, so cross(ap,ab)*cross(ac,ab)>0. For example, in the graph below, vector AE and AB are at the same side of AC, however, AB and AD are at different sides of AC. Similarly, vector ap and ab is at the same side of ac, vector bp and bc is at the same side of ba. After 3 checks, if all true, then set the hitrecord to be p and return true.

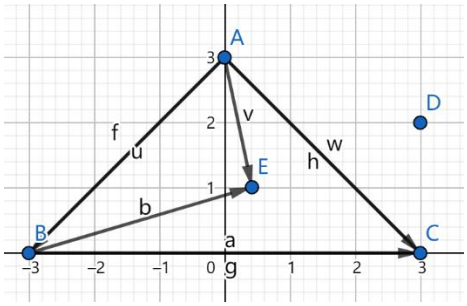

Figure 3 Check if the point is inside the triangle

3.2.2 Sphere-Ray intersection [6]

If p is on the sphere, length of cp is definitely rad. If the distance of a point on the ray and the center is radius, it's an intersect. So check if the quadratic equation has answers, or has only one answer(tangent situation is ignored). If it has 2 answers, choose the one with smaller distance to the viewpoint, as it is in the front.

After checking if it is valid, set the hitrecord and return true.

3.3 Light Tracing Process

3.3.1 Check what the ray hits

When a ray is sent through a pixel into the scene, the first step to check which object does it hit. So triangle-ray intersection function and sphere-ray intersection function mentioned above are used to check if the ray is intersect with a certain object. Traverse all surfaces, and choose the one with the smallest distance to be hr.

3.3.2 Checking Shadows and Shading

This step requires to calculate the original color of an object without being influenced by other objects (except for shadows). When the ray hits something, using Phong BRDF model mentioned above, it is easy to calculate the color of the hitting point. The color composes of diffuse reflection and specular reflection.

However, if the pint is blocked by another object, just skip shading process. In order to check if it blocked, it is necessary to send rays from the point to each light source one by one, and traverse all objects again to see if the ray intersects anything. If so, that means the point is blocked.

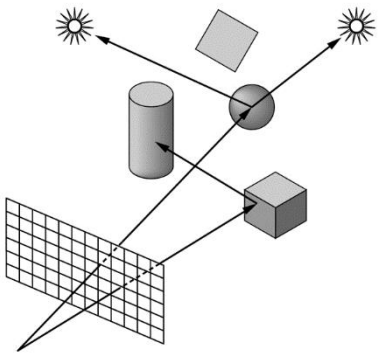

Figure 4 Basic light tracing process

3.3.3 Calculating Reflection

In this part I use recursive function. When the original shading calculation of the first point is done, let the be the new camera, and send out a new ray towards the reflect direction. Do the whole process

again, from checking what the ray hits, to doing a new shading of the point the new ray hits (if it does hit something,), and add up the new shading color to the first point, as its reflection color. By this way, the reflection calculating can be done no matter how many times are needed, as long as the ray keeps bouncing between objects. When the ray finally goes out from the scene after bouncing several times, the process ends.

However, to avoid infinite bouncing, a constant c is set, c plus by 1 when a cycle is done. When c equals constant maxraydepth, return black color and end. And when c is not up to maxraydepth, but the bouncing ray hits nothing, return black color and end as well. Especially when the first ray hit nothing, return background color.

3.4 Bounding Box Acceleration [7]

3.4.1 Data Structure

Firstly, a Node structure is designed along with the Box class. It has a pointer to a box, and eight pointers to eight son nodes.

The boxes are in three levels, the mainBox covers everything in the scene, and it's divided into 8 congruent rectangulars. The smaller boxes are put in a vector<Box> called "firstLay".

The boxes in "firstLay" are further divided into 8 congruent smaller boxes, and put them in a 2d vector<Box>, so there are overall 64 small bounding boxes that contain different objects.

In the same way, the mainNode has a pointer to the main box and eight pointers to eight son nodes (also in a vector<node> called "firstNodes"). Each son node has a pointer to a box in "firstLay". and eight pointers to eight grandson nodes (also in a 2d vector<Node>), which are all correspondent to 64 smallest bounding boxes.

On the whole, the tree structure is convenient in searching what a ray hit, it's much easier to search layer after layer, and save time from unnecessary searches.

The reason why a put all the boxes and nodes in a same level in a vector is that its more flexible in initializing and checking, without naming every one of them.

Before starting tracing, go through all objects to locate the mainBox, and divide it layer after layer, save the smaller boxes in vectors and nodes, and complete the tree structure. Then go through all objects to record them in intersectLists in every box they're in.

The folowing graph demonstrates the datastructure.

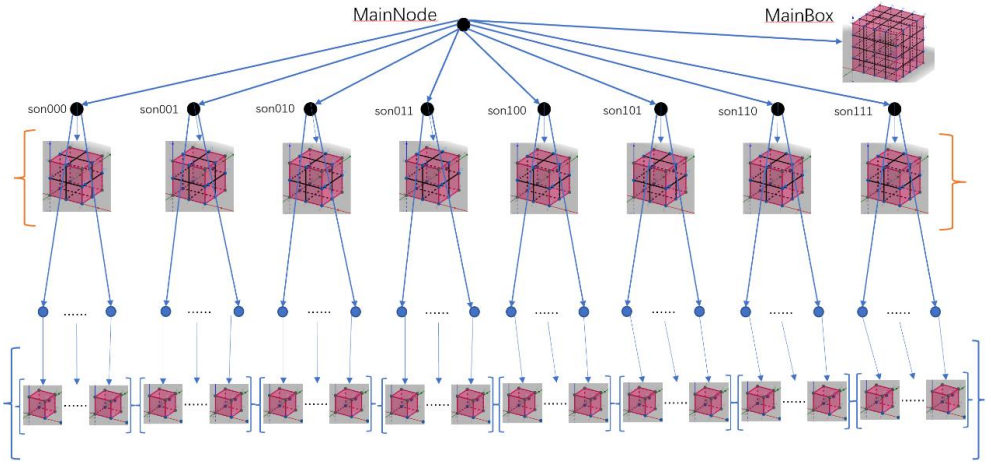

Figure 5 Data structure

3.4.2 More functions

3.4.2.1 checkPosition()

This is the function for positioning the mainBox. Go through all surfaces to check the biggest and

smallest xyz value, and let them be positions of the six surfaces of the mainBox.

3.4.2.2 separate()

This is the fuction to divide a box in a node and initial them into eight son boxes in eight son nodes, and initial eight son pointers in father node.

3.4.2.3 boxIntersect()

This is a function checking if an object is intersect with a box.

Spere::boxIntersect() is designed to check if the center of the sphere is inside the range of the box a radius wider in all six directions.

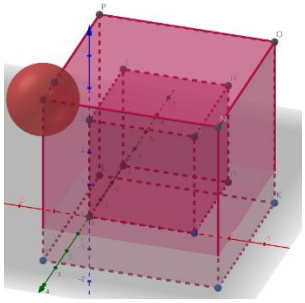

Figure 6 Basic demonstration of Spere::boxIntersect()

Triangle::boxIntersect() is designed to biuld the smallest rectangular around the triangle with its biggest and smallest xyz values. And check if the two rectangular intersect.

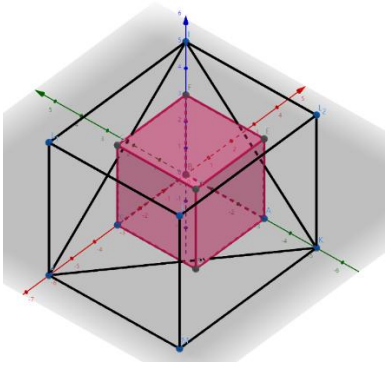

Figure 7 Triangle::boxIntersect()

Note that all boxIntersect functions are not absolutely precise. Some objects that do not intersect with a certain box but close to it may also be judged to be intersected. There is nothing wrong with it, as to precisely check if a triangle or sphere is intersect with a box is relatively complicated and unnecessary. Having possibly multiple checks in one object saves more time than to check whether it intersect with a box or not precisely.

3.4.2.4 Box::Intersect()

This is the function to see if a ray hits a box, using Liang-Barsky clipping algorithm. [8]

A ray would definitely hit on the plane of six surfaces of the box (three pairs of parallel planes), and get three pairs of corresponding t values, as the graph demonstrates below. The situation when the ray is parallel to a set of planes is ignored as it is almost impossible. The biggest t value of three smallest xyz surfaces is supposed to be tmin, and the smallest t value of three biggest xyz surfaces is supposed to be tmax. These two amounts are the two t values of the two ray-box intersected point. If tmax does exceeds tmin, the ray hits the box. But considering the starting point may be inside the box, just check if tmax is valid, if true, return true.

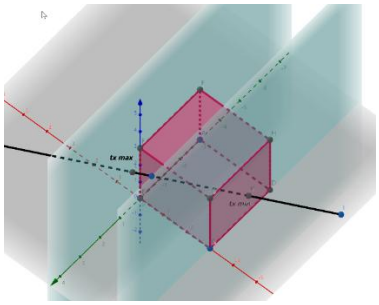

Figure 8 Demonstration of getting txmax and txmin

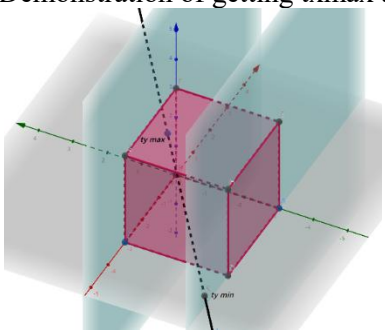

Figure 9 Demonstration of getting tymax and tymin

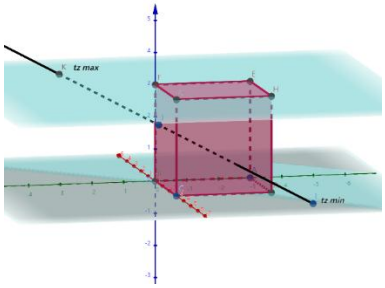

Figure 10 Demonstration of getting tzmax and tzmin

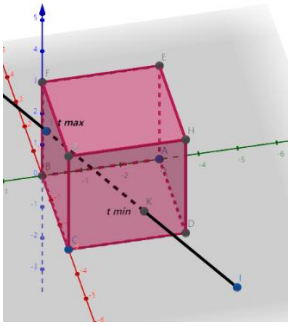

Figure 11 Geometric meaning of tmin and tmax

3.4.3 Bounding Box Accelerate Strategy

After the data structure is built. Whenever going through everything to find out what a ray hits, check if it hits the mainBox first. If it does, check the eight son boxes one by one, if the ray hits some of them, continue to check their sons. If the ray hits some boxes, it the last layer, collect their "intersectList" in a new vector as potential intersect objects. Then go through the new vector, instead going through every object.

Through debugging, the process is visualized. The program output “An intersect” whenever the ray hits a small box, and output the index of all potential objects, and their amount. In **Figure 12** and **Figure 13**, when calculating pixel (250,200), the ray hit six small boxes, and stored 3251 objects to its potential intersect list, rather than the original over 8000 objects.

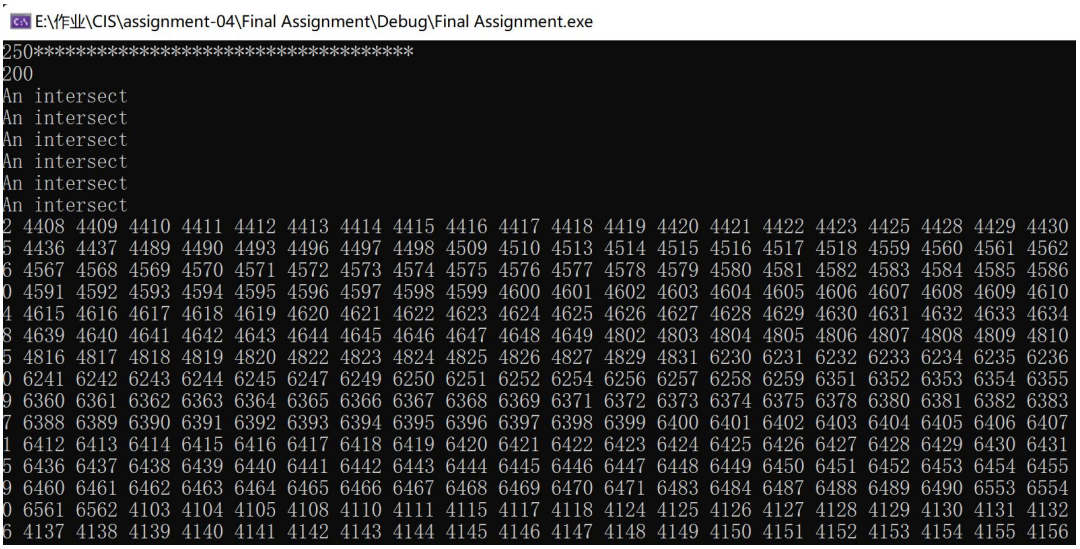

Figure 12 debug outcome 1

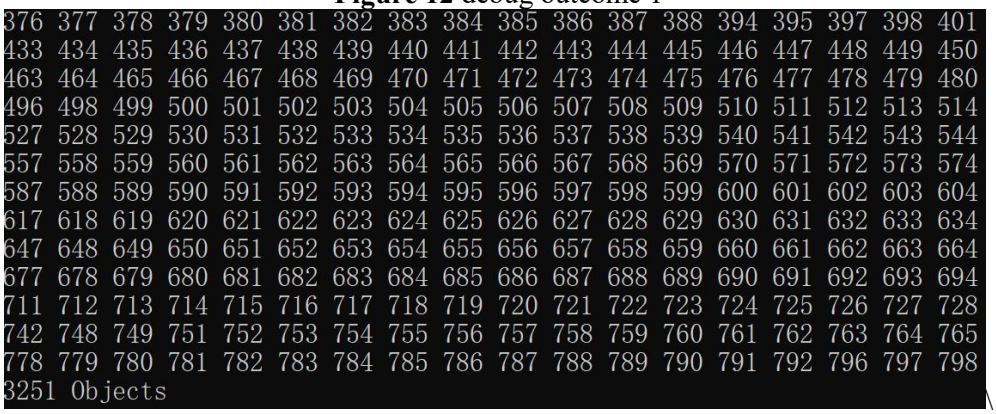

Figure 13 Debug outcome 2

In average, objects been checked intersecting became approximately 10% to 40% of the total. And running time shrank to about half an hour from the original 70 minutes.

4.Problems and solutions

4.1 Overexposure

The range of color rgb value in openGL is from 0 to 1. However, the recursive process may result to a rgb value which is too high, approximately 5 to 6 at most, 1 to 2 at least. This may render nearly all object completely white. As Figure 14 shows.

Figure 14 Overexposed image

$$Color = \frac{color}{color+1} (3)$$

To solve this problem, this project adopts a common rescale method, as equation (3) demonstrates [9].

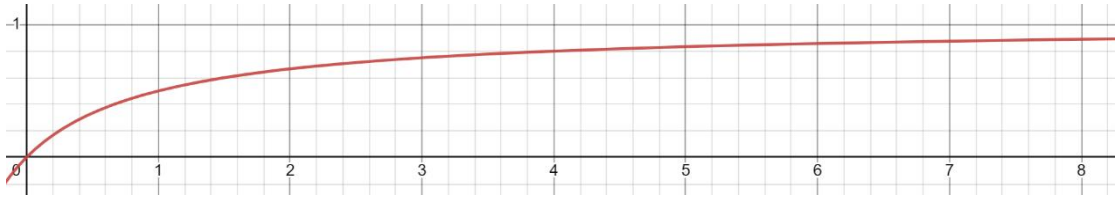

Figure 15 Graph of the function

As **Figure 15** shows, this rescale equation make small value similar to itself, while keep big value under 1 and keep different color perceptible.

Image after rescaling is as **Figure 16**.

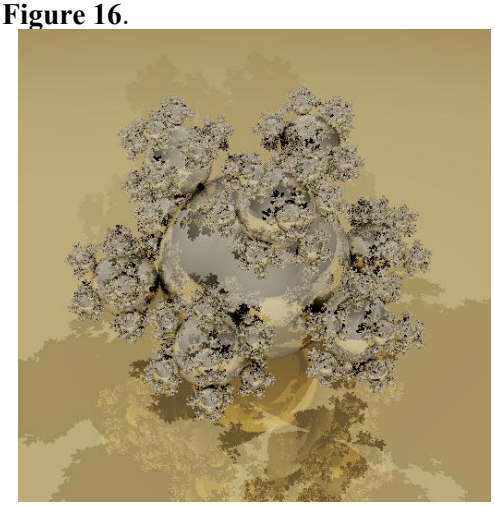

Figure 16 Rescaled image

4.2 Noise

Almost every image rendered at the beginning has huge amount of discrete black dots on each objects' surface, which makes the whole image fussy.

This is because when the ray interacts an object, the hitting point is not precisely on the surface of the object, as float type numbers are discrete. As a result, the hitting points are inside or outside of the surface, with extremely close distances. Consequently, inside points are blocked by the object itself when checking shadows, therefore the system automatically set the color of the pixel black.

To solve this problem, the easiest method is to increase the threshold value of valid block from 0 to 0.00001, to make sure a point on a surface is not blocked by itself.

5. Conclusion

Overall, an effective ray tracing system is implemented successfully, which can input a nnf file of model information and output an elegant image with correct reflection and shadows. Noise problem is also avoided in the algorithm. And most importantly, a completely self-designed accelerate structure is built according to basic bounding-box theories. It can reduce approximately 50% to 70% of the total rendering time.

However, refraction is always an indispensable part of ray tracing. Due to its high optical complexity and huge amount of computation, refraction is not implemented in this project. It is still an interesting area, and worth trying in the future. Then, transparent objects like glasses or picture under water can also be rendered.

References

[1] Blinn, J.F. Models of light reflection for computer synthesized pictures. Proc. SIGGRAPH, San Jose, Calif., pp. 192-198, 1977.
[2] Broadbent, A. D. A critical review of the development of the cie1931 rgb color-matching functions. Color Research and Application 29, 4, 2004.
[3] M. Woo, J. Neider, and T. Davis. OpenGL Programming Guide, Second Edition. Addison-Wesley Longman Publishing Co., 1997.
[4] R. Wright, B. Lipchak, and N. Haemel, OpenGL SuperBible: Comprehensive Tutorial and Reference. Addison-Wesley Professional, fourth ed., 2007.
[5] Spencer, G.H., and Murty, M.V.R.K., 1962, "General Ray Tracing Procedure," Journal of the Optical Society of America, Vol, 52, June, pp.672-678, 2009.
[6] Liu, Y.K., Wang, X.Q., Bao, S.Z., Gombo˘i, M., Z˘alik, B.. An algorithm for polygon clipping, and for determining polygon intersections and unions. Computers & Geosciences 33, 589–598, 2007.
[7] REINHARD, E., SMITS, B., AND HANSEN, C. Dynamic acceleration structures for interactive ray tracing. In Rendering Techniques 2000: 11th Eurographics Workshop on Rendering, 299–306. 2000.
[8] Y.-D. Liang and B.A. Barsky. A new concept and method for line clipping. ACM Transactions on Graphics (TOG), vol. 3, no. 1, pp. 1-22, 1984.
[9] D. Guo, Y. Cheng, S. Zhuo and T. Sim, "Correcting over-exposure in photographs", Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR), pp. 515-521, Jun. 2010.