國立臺灣大學電機資訊學院資訊網路與多媒體研究所

碩士論文

Graduate Institute of Networking and Multimedia
College of Electrical Engineering and Computer Science
National Taiwan University

Master Thesis

使用平行化計算之互動混合式光跡追蹤法

Interactive Hybrid Raytracing using Parallel Computing

周伯相
Po-Hsiang Chou

指導教授：莊永裕 博士
Advisor: Chuang Yung-Yu, Ph.D.

中華民國 99 年 7 月

July, 2010

國立臺灣大學
資訊網路與多媒體研究所

碩士論文

使用平行化計算之互動混合式光跡追蹤法

周伯相 撰

99
7

中文摘要

本論文提出一個可用於動態場景、視角、光源的混合式光跡追蹤成像法，混合使用區域著色法(Local Shading)和光跡追蹤法。過去雖然有許多相關的方法被提出，然而大多只呈現部分成像效果而已，而且硬體要求也比較高。我們選擇使用區域著色法和光跡追蹤法各自的優點並利用圖形處理器(GPU)和多核心處理器(CPU)以平行計算方式，期望在目前一般的硬體規格下能計算出柔和陰影、反射、折射，以及間接光照等效果，並且加速至互動速率下呈現。

Abstract

This thesis presents a hybrid ray tracing renderer for fully dynamic scene, camera, and light sources. Many Ray tracing algorithms have been proposed to product most visual effects but these methods are still hard to render at interactive rate on common hardwares. We combined raytracing with local shading methods and used parallel computing on GPUs and CPUs to product most visual effects like soft shadow, reflection, refraction, indirect lighting effects at interactive rate on most common hardware today.

Contents

口試委員會審定書 **i**

中文摘要 **iii**

Abstract **v**

1 Introduction **1**

2 Related Work **3**

3 Rendering **7**
3.1 Direct lighting 7
3.2 Soft shadow 8
3.3 Indirect Lighting 10

4 Raytracing **13**
4.1 Bounding Volume Hierarchies 14
4.2 KD-Tree 15
4.3 Shading 17
4.4 GPU Implementation 18

5 Experiments and Results **23**
5.1 Environment 23
5.2 Results 24

6 Conclusion and Future Work **29**
6.1 Conclusion 29
6.2 Future Work 30

Bibliography **31**

List of Figures

2.1 (a) Direct light only. (b) Pure Monte Carlo raytracing, 512 samples per ray, image size 1024x1024, rendered with 15Min19Sec. 4
2.2 Whitted-style raytracing. 4

3.1 Our rendering pipeline. 8
3.2 Illuminate the scene using virtual point lights (VPLs). 10

4.1 Acceleration structures. Mix bounding volume hierarchies with KD-tree. . 13
4.2 Direct3D rendering pipeline. 19

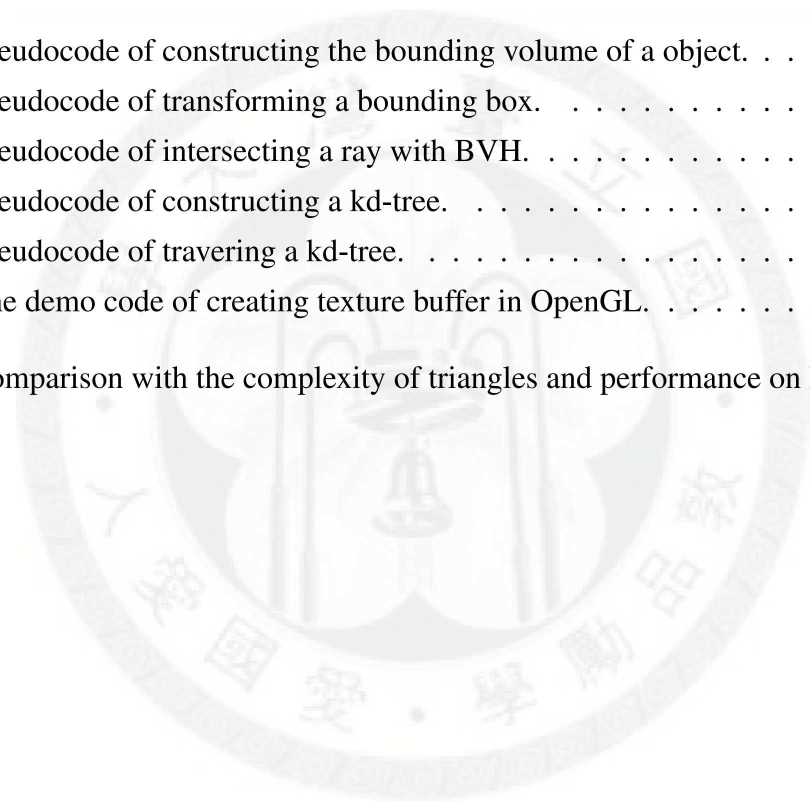
5.1 The test scenes. (a) Rolling box. (b) Rolling box donut. (c) 2 spheres. (d) Rolling box Venus. (e) Rolling box Sponza. 24
5.2 Rolling box, the frame rate is about 20 fps. 25
5.3 Rolling box donut, the frame rate is about 7 fps. 25
5.4 2 spheres, the frame rate is about 10 fps. 26
5.5 Rolling box Venus, the frame rate is about 5 fps. 26
5.6 Rolling box Sponza, the frame rate is about 4 fps. 27
5.7 Multi-resolution geometric,Venus, for comparing with the complexity of triangles and performance. 27

List of Tables

3.1 Sample code of Blinn-Phong shading model. 9
3.2 Pseudocode of accumulate the photon illumination. 11

4.1 Pseudocode of constructing the bounding volume of a object. 14
4.2 Pseudocode of transforming a bounding box. 14
4.3 Pseudocode of intersecting a ray with BVH. 15
4.4 Pseudocode of constructing a kd-tree. 16
4.5 Pseudocode of travering a kd-tree. 20
4.6 The demo code of creating texture buffer in OpenGL. 21

5.1 Comparison with the complexity of triangles and performance on PCs. . . 24



Chapter 1

Introduction

There are still limitations of visual effects in virtual reality applications on modern hardwares. Rasterization-based rendering method is used in most virtual reality applications because that the computational power was still not enough to solve the global illumination problem in real-time. However, the objects of the scene are animated in real-time application so that we don't need to compute global illumination accurately. Therefore, we can approximate the global illumination to reduce computing cost and make the visual quality believable.

Rasterization-based rendering method can produces high quality direct lighting effects such as per-pixel lighting, soft shadow and it is used in almost all the real-time applications like video/PC games and training simulators. The performance is good on modern graphics hardware but there are still limitations of many effects like global illumination, indirect lighting, reflection, refraction which we can not approximate well with a rasterization-based renderer.

The ray tracing method traces the path through pixels to virtual objects to render the realistic images. It can simulate realistic lighting effects such as reflection and refraction which are difficult to be simulated by other algorithms. It is the most popular renderer in animations, movies that need not to offer interactive experience. The high computational costs make this method hard to be used on real-time applications.

The multicore processors are more and more popular today but there still have just few of core number on PC. The other rapidly increasing processing power is GPUs. The

programming on GPUs is quite different to single processing programming. The parallel computer programs are more difficult to write because there are more potential bugs, unique programming styles, the limitation of instructions. However, the parallel computing is a trend in discovering computering power and have become popular today. With the parallel programming including CPUs and GPUs, the ray tracing algorithm can be real-time or interactive rate.

We propose a rendering pipeline that can render most visual effects including direct lighting, indirect lighting, soft shadow, reflection, refraction with high visual quality in interactive frame rate. The final result was consisted of several visual effects, which we can deal with effectively. Some visual effects like direct lighting, soft shadow, indirect lighting are performed well on rasterization-based renderer. We will explain the rendering pipeline and how we produce those effects in the third section. The other visual effects like reflection and refraction are the natural output of the ray tracing renderer. We will show our ray tracer in section four. The fifth section shows the results and comparsion. The final section draws the conclusion and our future research.

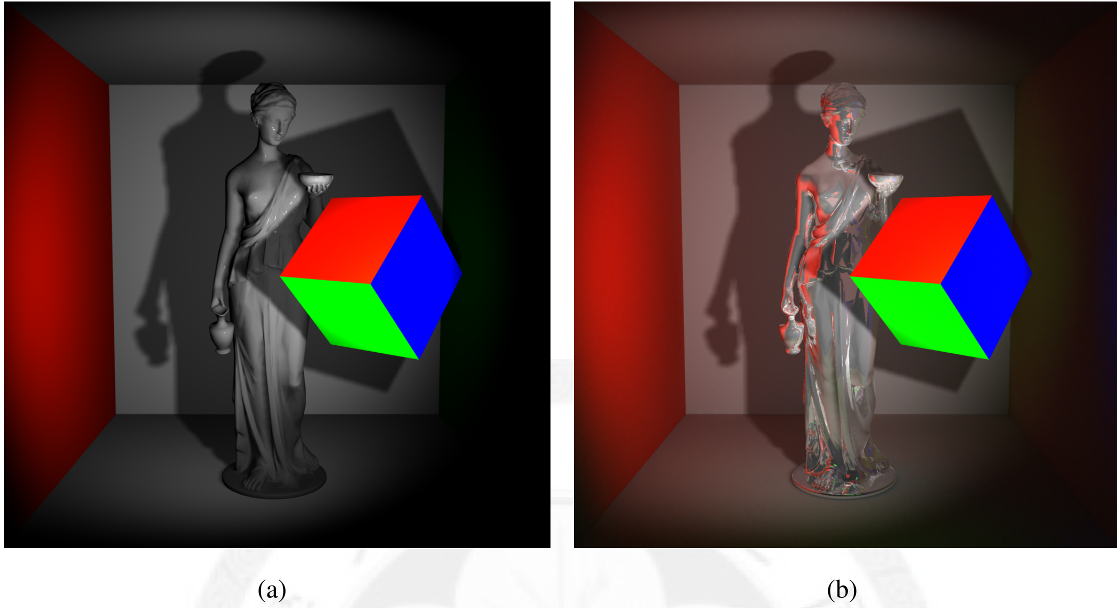
Chapter 2

Related Work

In this thesis, we put emphases on visual quality and performance at real-time or interactive frame rate on modern hardware. Global illumination is the final visual effect we pursue but it is difficult to run in real time because of the heavy computation cost. There are several algorithms have been proposed to approximate global illumination. The comparison between direct light only and global illumination is shown as Figure 2.1.

Raytracing . There is an important raytracing research came from Turner Whitted[19] in 1980 as Figure 2.2. When a ray hits a nearest surface, it will generate two new rays that are reflection ray and refraction ray and continue to hit the objects in the scene. At the intersection point, the ray tracer computes the shading that come from local illumination, reflection illumination and refraction illumination. If testing in shadow is required, it will also generate shadow rays, the point to each light sources, to check any opaque object is found. The process continues recursively until reach the maximun trace depth. In nature, the lighting of the surfaces comes from not only the exact path but also the enviroment illumination. Physically based raytracing[11] has been introduced to render photorealistic images. By increasing sampling rays of each pixel to hundreds even thousands, raytracing can render a very high quality of visual realism. Of course, computing cost also increased hundreds even thousands time.

Illumination from many lights. Global Illumination can be approximated by lit from many point lights. There is several work that have good improvement to accelerate the render speed. Lightcuts proposed by Walter et al. [18] is a scalable algorithm for handling



(a) (b)

Figure 2.1: (a) Direct light only. (b) Pure Monte Carlo raytracing, 512 samples per ray, image size 1024x1024, rendered with 15Min19Sec.

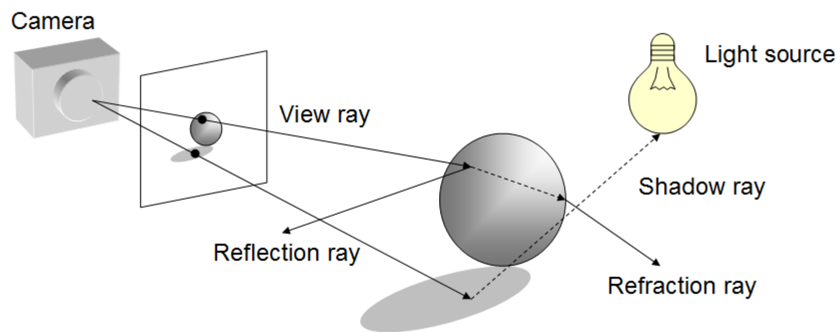


Figure 2.2: Whitted-style raytracing.

many lights by a hierarchical light tree. Matrix Row-Column Sampling for the Many-Light Problem by Hašan et al. [5] samples and clusters the lights on GPUs and accelerates the rendering speed to seconds per frame.

Instant radiosity by Keller [8] is limited to indirect illumination. The method creates a group of particles and shoots them from light sources into the scene. Every insection point creates a virual point light(VPL) and then uses the VPLs as light sources to illuminate the scene. There is extended work of Instant radiosity. Bidirectional Instant Radiosity [14] creates the VPLs starting at the camera instead of starting at the light sources. Metropolis Instant Radiosity by Segovia et al.[15] combines Instant Radiosity with Metropolis sampler [17]. Incremental Instant Radiosity by Laine et al. [9] reuses VPLs and maintains the distribution. Virtual Spherical Lights [4] integrates over a non-zero solid angle of VPLs.

Precomputed Radiance Transfer (PRT) has been shown to real-time rendering for some effects of global illumination like soft shadow, diffuse and glossy interreflections [16] [10]. For most PRT algorithms, the incident irradiance must be precomputed, a large amount of data must be stored, the scene must be static and several minutes to hours are required for precomputation.

Photon map is a global illumination algorithm developed by Henrik Wann Jensen[7] and was first implemented on the GPUs by Purcell et al.[13] change the data structure to a uniform grid and search a sample point in the grid by kNN algorithm. Some visual effects like caustics, diffuse interreflection, subsurface scattering can be simulated by this algorithm. In order to realistically simulate between lights and different objects, there must have huge numbers of photons to render high quality images.

Chapter 3

Rendering

There are two popular computer graphics APIs: Direct3D and OpenGL. Direct3D was designed by Microsoft Corporation on Microsoft's family of operating systems, including the Xbox family of video game console. OpenGL is an open standard API and is available on most modern operating systems like Windows, Mac OS X and Linux. It is also available on mobile devices such as iPhone, Android and Symbian OS in the OpenGL ES form. We use OpenGL as our main graphics API, expected that our implementation can work fine on most platform.

Figure 3.1 shows the rendering pipeline of this work. First, we build bounding volume for each object and if the object contains more then hundred triangles then we also build a kd-tree for the object. Next, we render shadow maps for each light source and we use per-pixel lighting for direct lighting and store material attributes on screen space. We then use raytracing for the multi-bounce materials. We create global photons from the light sources and approximate the indirect lighting effect. Finally we compose the final image from the images: direct lighting map, reflection and refraction map, indirect lighting map.

3.1 Direct lighting

Most typical graphic acceleration hardware uses rasterization algorithms today. Rasterization-based rendering was performed well in direct lighting effects. We used Blinn-Phong shading model, developed by Jim Blinn[1], as our direct lighting model. We implemented per-pixel lighting algorithms by using fragment shaders and computing illumination at each

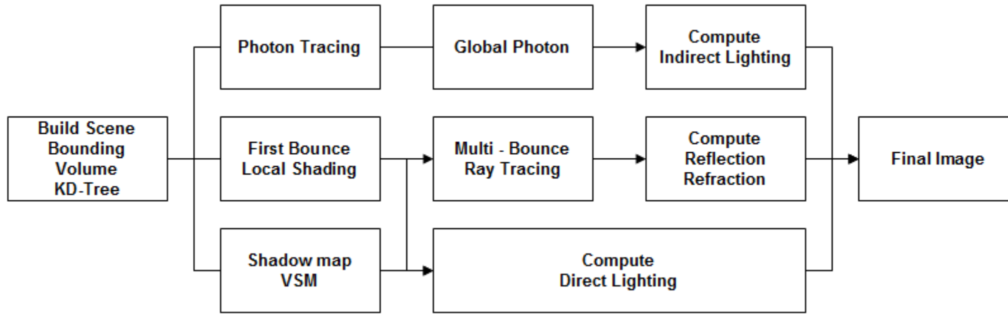


Figure 3.1: Our rendering pipeline.

pixel to produce realistic images. We compared the rasterization method with raytracing method and realized that the visual quality was exactly the same and rasterization method was much faster then raytracing method.

Blinn-Phong is the default shading model on each vertex in OpenGL or Direct3D rendering pipeline, interpolated by Gouraud shading [3] for pixel value between vertices. We can not use Blinn-Phong shading model in fixed pipeline rendering, but we can compute the Blinn-Phong equation on each pixel using fragment shaders. The illumination of each point I_p is

$$I_p = k_a i_a + \sum_{m \in lights} (k_d (L_m \cdot N) i_d + k_s (N \cdot H)^{\alpha} i_s)$$

,where k_a, k_d, k_s denotes the ambient reflection constant, the diffuse reflection constant, the specular reflection constant respectively, i_a is ambient lighting, i_d is diffuse lighting, i_s is specular lighting, L as the vector from the point to the each light source, N as the surface normal vector at the point, α is a shininess constant of the material, H as the half-angle vector between the view vector, L, and light source vector, V.

$$H = \frac{L + V}{|L + V|}$$

The sample code of Blinn-Phong shading model can be written as Table 3.1.

3.2 Soft shadow

Effect such as hard edge shadow was simple to implement using raytracing algorithms, but there was still time consuming problem with soft shadow. So there are popular shadowing

```
vec3 BlinnPhongShadingModel()
    vec3 H=normalize(L+V);
    vec3 Ip=Ka*Ia+
            Kd*max(dot(Lm,N),0)*Id+
            Ks*pow(max(dot(H,N),0),shininess)*Is;
    return Ip;
```

Table 3.1: Sample code of Blinn-Phong shading model.

algorithms were developed to exploit modern graphic hardware.

Shadow map[20] is the most common shadowing algorithm and can be implemented on current graphics hardwares and their computing cost is low even the scene is complexity. Unfortunately, shadow map algorithm has aliasing problem if not filtered well. We implemented variance shadow map, by Donnelly et al.[2], for more efficiently soft shadow. First, we render the scene to a fp16 framebuffer object from the view of light source and the normalized depth and squared depth were stored in the framebuffer. Then, filter the shadow map using a 7x7 gaussian blur. The result of filtering shadow map will discover the moments M_1 and M_2 over the filter region:

$$M_1 = E(x) = \int_{-\infty}^{\infty} x p(x) dx$$

$$M_2 = E(x^2) = \int_{-\infty}^{\infty} x^2 p(x) dx$$

The mean μ and variance σ^2:

$$\mu = E(x) = M_1$$

$$\sigma^2 = E(x^2) - E(x)^2 = M_2 - M_1^2$$

By Chebychev's inequality:

$$P(x \geq t) \leq p_{max}(t) \equiv \frac{\sigma^2}{\sigma^2 + (t-\mu)^2}$$

Next, render the scene from the view of camera and read the shadow map to get the moments M_1 and M_2. If the depth $< \mu$, then the point is not shadowed, else compute the variance σ^2 and μ:

$$\mu = pd_2 + (1-p)d_1$$

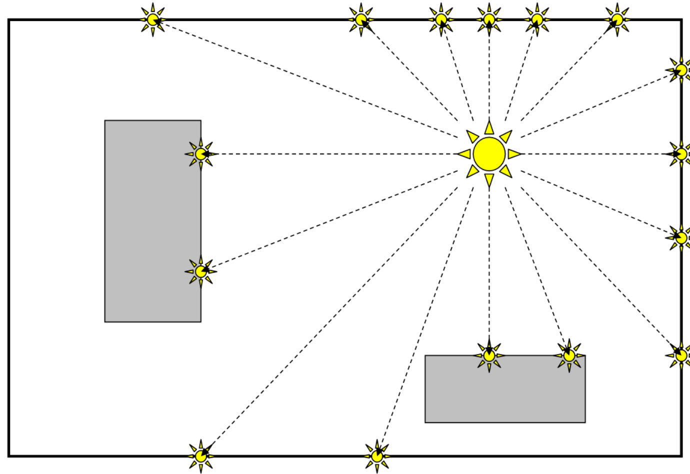


Figure 3.2: Illuminate the scene using virtual point lights (VPLs).

$$\sigma^2 = (p - p^2)(d_2 - d_1)^2$$

Where d_1 is the depth at occluder, d_2 is the depth at casting shadow planar, p is the percentage of the filter that is not occluded. Finally, scale the light intensity by p_{max}:

$$p_{max}(d_2) = \frac{\sigma^2}{\sigma^2 + (\mu - d_2)^2}$$

3.3 Indirect Lighting

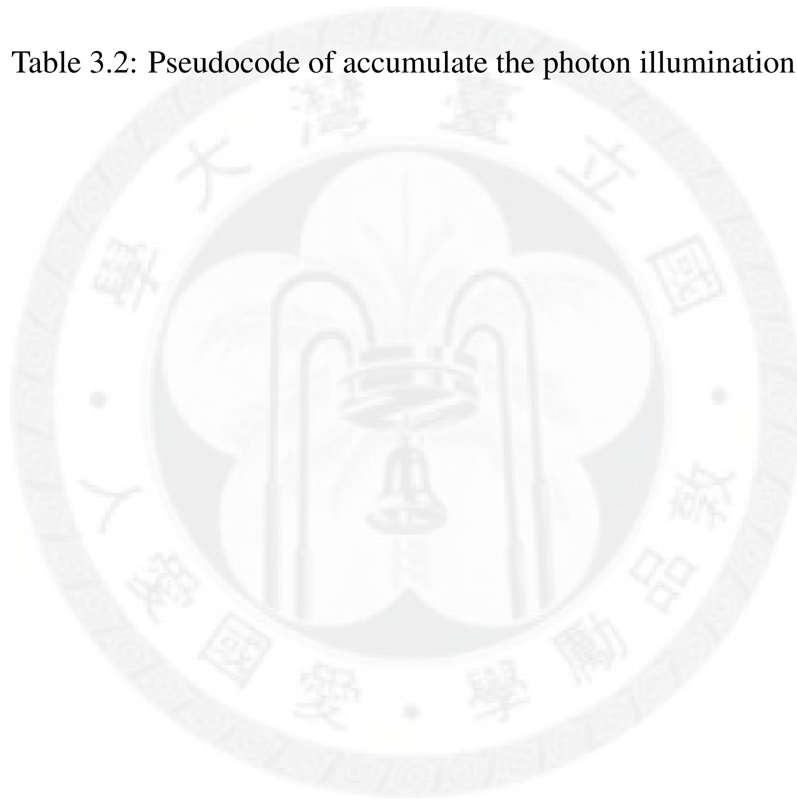
Instant Radiosity by Keller [8] is used to calculate the indirect illumination of the scene by shooting particles from the light sources and use those particles as point light sources to illuminate the scene show as Figure 3.2. Those particles were called virtual point lights (VPLs). The benefits of this method are there don't need any preprocessing and it is fitted for modern hardwares. There are two different issues. If we need a high quality indirect illumination, we can increase the number of VPLs and illuminating from each VPL with a high quality shadow map. Otherwise if the rendering speed is required, we can adjust the numbers of VPLs and ignore the shadow caused from each VPL.

The same idea was used in our implementation. First we create uniform sampling rays from the light sources, tracing each ray to intersect the scene by ray tracer and create a photon, each photon contains position, direction, color, and the power of light. Next, the data of all photons were stored to a texture buffer, which is updated every frame. Final,

we calculate the illumination for each photon and accumulate them on fragment shader as Table 3.2.

```
//world_pos, the position of the pixel in world space
vec4 IndirectIllumination(PHOTON photons[], vec3 world_pos)
    vec4 illumination=0;
    for each(PHOTON photon in photons)
        if(IsInFront(photon.direction,world_pos))
            illumination+=CalcuateIllumination(photon,world_pos);
    return illumination;
```

Table 3.2: Pseudocode of accumulate the photon illumination.



Chapter 4

Raytracing

The Whitted-style raytracing was used for our raytracing renderer and the shadow rays were ignored because we render soft shadow using shadow map algorithm. Raytracer was used on dynamic scene so fast reconstruction of the acceleration trees were needed. The real time kd-tree construction algorithm was presented by Kun Zhou et al. [21]. The computing cost for fully animated scene is still too high. In order to reduce the computing cost of reconstructing acceleration tree, we try to mix bounding volume hierarchies with kd-trees as Figure 4.1. If the triangles of the objects were greater then hundreds the system will build kd-tree for the object else the system will simply build the bounding volume for the object. While the object is moving, we update the transformation matrix and the bounding volume and the reconstruction of kd-tree is not required.

In order to downscale the computing cost of the raytracer, we use a trivial method in our framework that we first use fragment shader to render an image of the scene that each

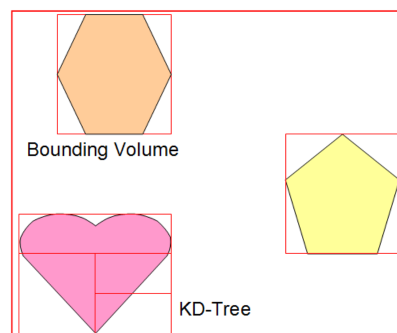


Figure 4.1: Acceleration structures. Mix bounding volume hierarchies with KD-tree.

color channel contain material factors, include reflection factor and refraction factor. Next we group pixels and put in a thread pool, use the material map on screen space as a look up table, then trace each ray tracing that contain reflection factor or refraction factor. The framework works on parallel architectures with multi cores processors.

4.1 Bounding Volume Hierarchies

Bounding volume hierarchies(BVH) is a tree structure to accelerate the intersection of ray and objects. We use the axis-aligned bounding box(AABB) for our bounding volume. There is a bounding volume for each object, and the volume is contained in the bounding volume of parent's node. A leaf node of bounding volume may consists of triangles or a kd-tree node. The computing cost of a ray-AABB intersection is cheaper then a group of ray-triangle intersection and a kd-tree node intersection. So we use bounding volume test first to reject most geometric objects that will not intersect the ray.

```
BOUNDBOX BuildBoundVolume(vertices[])
    BOUNDBOX box;
    for(i=0;i<vertices.size();i++)
        Union(box,vertices[i]);
    return box;
```

Table 4.1: Pseudocode of constructing the bounding volume of a object.

```
BOUNDBOX Transform(BOUNDBOX boundbox,mat4 matrix)
    BOUNDBOX box;
    for(i=0;i<8;i++)
        vec3 corner=Transform(boundbox.corner[i],matrix);
        Union(box,corner);
    return box;
```

Table 4.2: Pseudocode of transforming a bounding box.

We construct the BVH using bottom-up methods. First we create bounding volume for each object, Table 4.1 shows the pseudocode. If there are overlap between the bounding

```
bool IntersectBVH(Ray ray, NODE node)
    if(Is_Intersected(ray,node.boundbox))
        if(node.isLeaf)
            return IntersectTriangles(ray,node.triangles);
        else if(node.isKDTree)
            return IntersectKDTree(ray,node.kdtree);
        else
            bool isIntersected=false;
            for each(NODE child_node in node)
                isIntersected|=IntersectBVH(ray,child_node);
            return isIntersected;
    else
        return false;
```

Table 4.3: Pseudocode of intersecting a ray with BVH.

volumes, these nodes are considered that they have the same parent node. Because our objects in the scene are movable, updateing bounding volume tree must be a lightweight processing. When the object moved, we don't need to rebuild the bounding volume of the object. We use the transformation matrix of the object to transform the bounding volume show as Table 4.2. The traversal of BVH on CPUs is trivial, when a ray intersects with bounding volume of the root, if there is no intersection then the ray was removed else all child nodes of the root need to do intersection test. A leaf node of the tree may contains a group of triangles or a kd-tree structure. We continue to do intersection test of all the child node recursively until the nearest intersection point is found show as Table 4.3.

4.2 KD-Tree

KD-tree is a binary space partitioning structure and the space is partitioned by axis-aligned planes. A property of kd-tree is that the nearer child node will be tested first, so the nearest intersection point is guaranteed. We use kd-tree for triange intensive objects and the root of the kd-tree is the bounding box of the object. Assume that all the object are rigid bodies then we can try to construct an optimized kd-tree and do not need

to reconstruct every frame even if the object is moving. The pseudocode of constructing a kd-tree of a object show as Table 4.4.

```
NODE BuildKDTree(GEOMETRY geometry,int depth)
    if(geometry.triangle.size()<=MIN_TRIANGLES
        || depth>=MAX_TREEDEPTH)
        return CreateLeafNode(geometry);
    EDGE bestEdge=FindBestEdge(geometry.edges);
    NODE node=CreateTreeNode();
    GEOMETRY left_geometry=SplitGeometry(geometry,bestEdge,LEFT);
    GEOMETRY right_geometry=SplitGeometry(geometry,bestEdge,RIGHT);
    node.left=BuildKDTree(left_geometry,depth+1);
    node.right=BuildKDTree(right_geometry,depth+1);
    return node;

EDGE FindBestEdge(EDGE edges[])
        Sort(edges);
        EDGE bestEdge=NULL;
        float bestCost=0;
        for each(EDGE edge in edges)
            cost=CostFunction(edge);
            if(cost<bestCost)
                bestCost=cost;
                bestEdge=edge;
        return bestEdge;
```

Table 4.4: Pseudocode of constructing a kd-tree.

For the best split, the cost equation was bring up by Stefan Popov et al. [12]. We are looking for v that minimizes $C(v)$ with

$$C(v) = K_T + C_l(v) + C_r(v)$$

$$C_l(v) = K_I n_l(v) \frac{2(s_1 + s_2)(v - v_{min}) + 2s_1 s_2}{SA(N)}$$

$$C_r(v) = K_I n_r(v) \frac{2(s_1 + s_2)(v_{max} - v) + 2s_1 s_2}{SA(N)}$$

Where v denote the position of a split location for the current node N, bounded by v_{min} and v_{max}. $s1$ and $s2$ be the extent of node N in the other two axis. $n_l(v)$ and $n_r(v)$ denote

the number of objects to the left and right side of v. K_T is the constant of traversal cost, K_I is the constant of intersection cost.

Due to the limitation of GPUs programming, recursive function is not allow for GPUs implementation, we use short-stack method by Horn et al. [6] for the traversal of a kd-tree on both CPUs and GPUs version. The pseudocode of travering kd-tree show as Table 4.5.

4.3 Shading

When a ray hits on the surface, the shading of the intersection point will be processed. If the material of the surface is reflective and/or refractive and not reach the maximum of tracing depth, it will split into two rays, reflection ray and refraction ray, and trace those rays recursively. The shading color will be combined from local shading color, reflective color and refractive color. Here, we ignore the intersection test from the shadow ray, because we use soft shadow map algorithm instead of raytrace shadow.

To compute the lighting of intersection point, we need the normal vector of the intersection point. We use Barycentric coordinate system (1827) by August Ferdinand Möbius to interpolate the normal vector from the tree normal vectors of the vertices. Lets define three vertices r_1, r_2 and r_3 of a triangle T. A point r located inside the triangle will be considered as a weighted sum of those three vertices:

$$r = \gamma_1 r_1 + \gamma_2 r_2 + \gamma_3 r_3$$

Where γ_1, γ_2 and γ_3 are the weights for the vertices, and the sum of the weights is constant:

$$\gamma_1 + \gamma_2 + \gamma_3 = 1$$

Next, we pick two axis that the distance of the vertices on the other axis must not zero. The vertex $p(x, y)$ in the triangle (x_1, y_1), (x_2, y_2) and (x_3, y_3), the values of each weights:

$$T = (x_1 - x_3)(y_2 - y_3) - (x_2 - x_3)(y_1 - y_3)$$

$$\gamma_1 = \frac{(y_2 - y_3)(x - x_3) + (x_3 - x_2)(y - y_3)}{det(T)}$$

$$\gamma_2 = \frac{(y_3 - y_1)(x - x_3) + (x_1 - x_3)(y - y_3)}{det(T)}$$

$$\gamma_3 = 1 - \gamma_1 - \gamma_2$$

After found the normal vector of the intersection point, we compute the illumination by Lambertian reflection model.

4.4 GPU Implementation

General purpose computing on graphics processing units(GPGPU) programming have become popular today because of the high computing power. But the architecture of modern GPUs is designed for rasterization-based rendering, so there are some tricks when we would like to dig the computing power of GPUs. Besides the parallel raytracing on multi-core CPUs, we have also implemented the raytracing by the following APIs.

Direct3D is a part of Microsoft's DirectX API and is used to render 3D graphics. Refer to the Direct3D9 rendering pipeline, as Figure 4.2, only vertex shaders and pixel shaders that can be programmed using shader language. We consider that each pixel as a viewing ray, so the pixel shaders can be used for raytrace computing. There is another problem that we can not use the data of the geometries, so we need to pack the triangle data into textures. The value of color channels we use in pixel shaders is normalized, we need to rescale the coordinates on texture and restore on pixel shaders. A triangle consists of three positions and three normal vectors, so eighteen value of floating point are needed for a triangle data. We use a four channel texel, A32R32G32B32F texture format, to store triangle list and five texels are used for an element of a triangle data. Shader model 3.0 is used for our shader codes, there are up to 224 of 4D constant float registers, we put light source parameters, camera parameters and the data of all objects including material attribution, bounding volumes, transformation matrix in the constant registers and were updated every frame.

OpenCL (Open Computing Language) is a framework for writing parallel computing programs use of CPUs, GPUs, and other processors and it also gives application access to the GPUs for general purpose computing. Its architecture is similar to NVidia's CUDA and Microsoft's DirectCompute, provide parallel computing using task-based and data-based parallelism. The good new is that we don't need to pack the data and rescale it

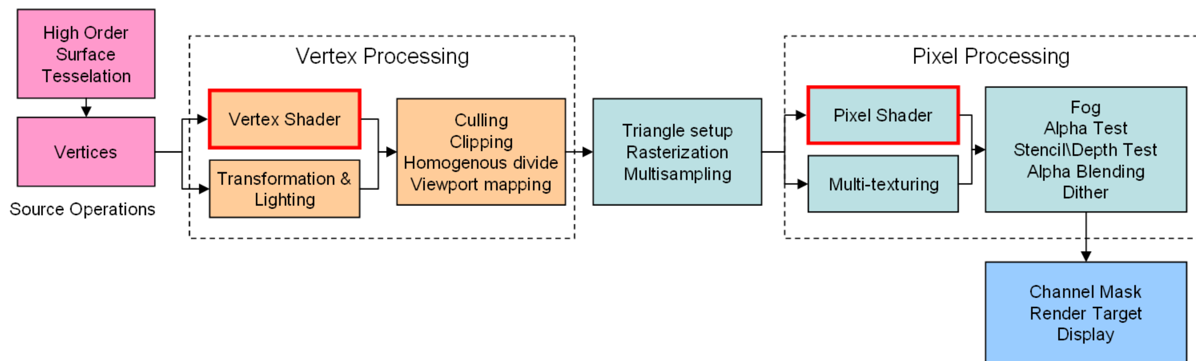


Figure 4.2: Direct3D rendering pipeline.

into a normalized texture. But the request of memory alignment is exacting. We realloc memory of primitives, light sources, triangles, kd-tree nodes, bounding volume nodes, index of triangle list, and store them into four components of vector data type. Finally, we compile the kernel and deal each pixel with a global thread.

OpenGL is a industry standard for high performance graphics APIs, and is managed by Khronos Group today. The rendering pipeline is similar to Direct3D (Figure 4.2) and also is for rasterization-based rendering. It is programmable for coloring each pixel call fragment shaders, just an alternative name of pixel shaders in Direct3D. There is an important API, Texture Buffer, for our works on OpenGL version 3.1. That means we can use raw data sources as a 1D texture in our fragment shaders, we don't need to rescale the data of triangles and normalize them that we can accurately access the data in the program. The code of creating texture buffer shows as Table 4.6.

```
bool IntersectKDTree(RAY ray,KDTREE tree)
    ray=Transform(ray,tree.matrix);
    tHit=ray.tmax;  tMin=tMax=ray.tmin;
    bool push;
    NODESTACK stack[MAX_STACKSIZE];
    NODE root=tree.root;
    while(tMax<ray.tmax)
        NODE node;
        if(IsEmpty(stack))
            node=root;
            tMin=tMax;  tMax=ray.tmax;
            push=true;
        else
            stack.pop(node,tMin,tMax);
            push=false;
        while(node.isLeaf==false)
            int a=node.axis;
            tSplit=(node.split-ray.point[a])/ray.direction[a];
            (first, second)=order(ray.direction[a],
                                                node.left, node.right);
            if(tSplit>=tMax || tSplit<0)
                node=first;
            else if(tSplit<=tMin)
                node=second;
            else
                stack.push(second, tSplit, tMax);
                node=first;
                tMax=tSplit;
                push=false;
            if(push)
                root=node;
        tHit=IntersectTriangle(ray,node.triangles);
        if(tHit<tMax)
            return true;
    return false;
```

Table 4.5: Pseudocode of travering a kd-tree.

```
glGenBuffers(1,&mBufferGL);
glBindBuffer(GL_TEXTURE_BUFFER,mBufferGL);
glBufferData(GL_TEXTURE_BUFFER,size,mem,GL_STREAM_DRAW);
glBindBuffer(GL_TEXTURE_BUFFER,0);
glGenTextures(1,&mTextureGL);
glBindTexture(GL_TEXTURE_BUFFER,mTextureGL);
glTexBuffer(GL_TEXTURE_BUFFER,mInternalFormatGL,mBufferGL);
glBindTexture(GL_TEXTURE_BUFFER,0);
```

Table 4.6: The demo code of creating texture buffer in OpenGL.

Chapter 5

Experiments and Results

5.1 Environment

We have implemented the rendering pipeline in a program and tested on a desktop PC with Intel CPU Q6600 2.4GHz, 8GB ram, ATI Radeon HD 4850 GPU at 625MHz with 512MB video memory at 993MHz. The resolution of images were 1024*1024. The OpenGL version is 3.2. The second hardware was a laptop with Intel Core i5-450M, 4GB ram, ATI Mobility Radeon HD 5650.

There are several scenes we used for experiments show as Figure 5.1. The camera, light sources, objects of the scenes are fully dynamic.

- *Rolling box* 22 triangles. A rolling box has different color of each side in a normal room with a moving spot light on it. For emphasizing the effect of indirect lighting.
- *Rolling box donut* 598 triangles. A rolling reflectional donut within a rolling refractional box.
- *2 Spheres* 1066 triangles. A moving sphere with a static sphere with different reflection factor.
- *Rolling box Venus* 31420 triangles. A rolling box with a reflectional venus.
- *Rolling box Sponza* 66466 triangles. A triangle intensive scene. A rolling reflectional box in Sponza.

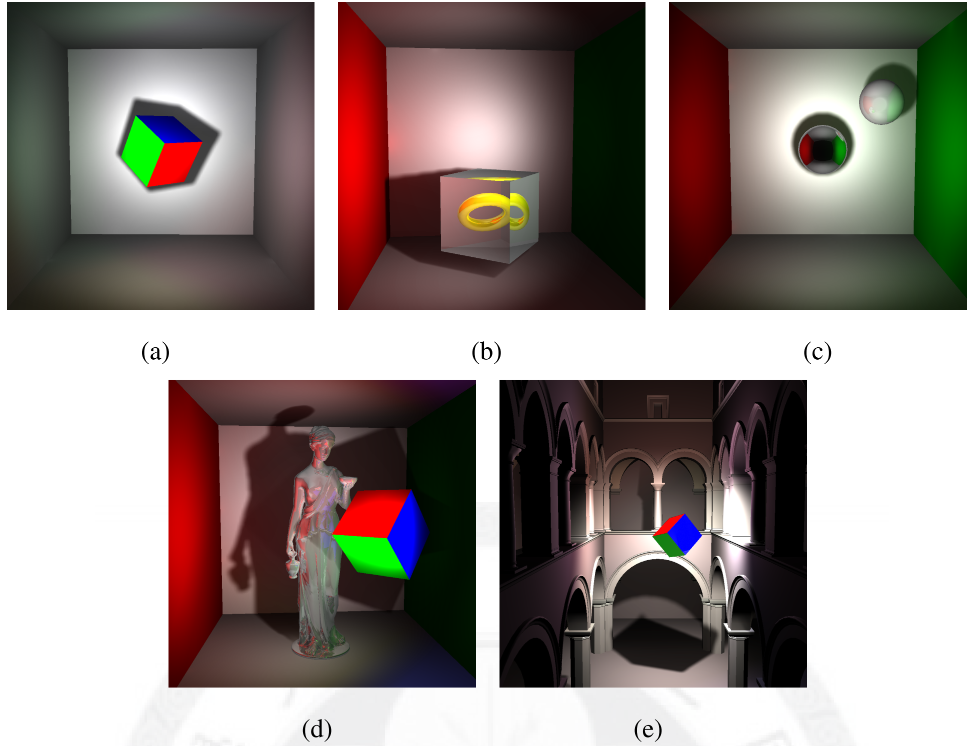


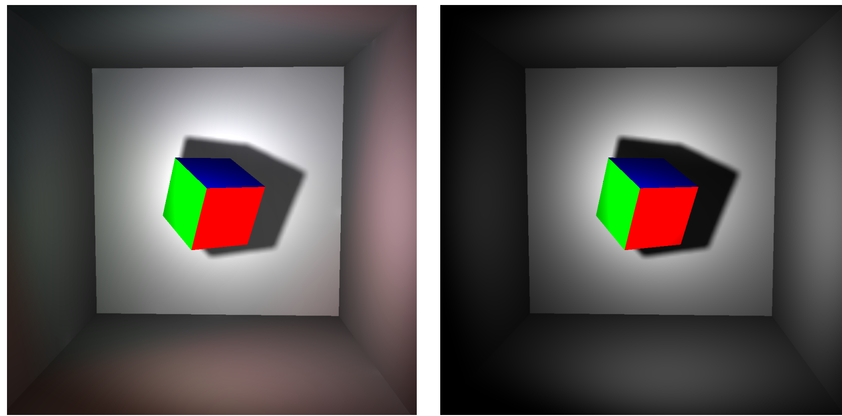
Figure 5.1: The test scenes. (a) Rolling box. (b) Rolling box donut. (c) 2 spheres. (d) Rolling box Venus. (e) Rolling box Sponza.

Triangles	641	1583	3153	6293	12571	18847	25125	31403
Desktop (FPS)	7	6.3	6	5.5	5	4.8	4.7	4.7
Laptop (FPS)	4.2	3.8	3.5	3.4	3.3	3.3	3.3	3.2

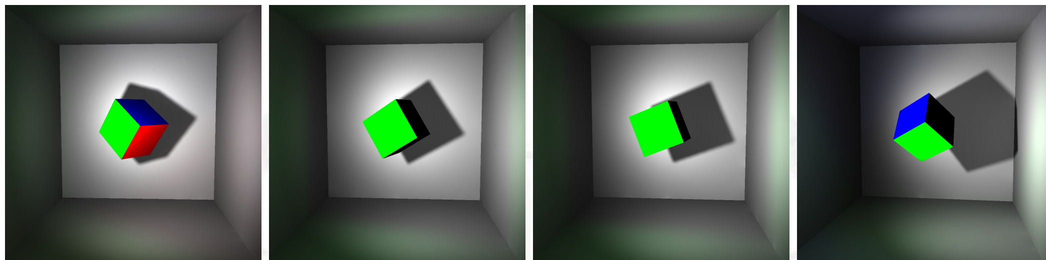
Table 5.1: Comparison with the complexity of triangles and performance on PCs.

5.2 Results

We have rendered a sequence of animation images of each scene, the camera, light sources, objects are all movable. There are also comparison between indirect lighting and direct lighting only as Figure 5.2(a)(b), 5.3(a)(b), 5.4(a)(b), 5.5(a)(b) and 5.6(a)(b). We also setup a scene with a multi-resolution geometric,Venus , as Figure 5.7 to test the relation between the complexity of triangles and performance as Table 5.1.

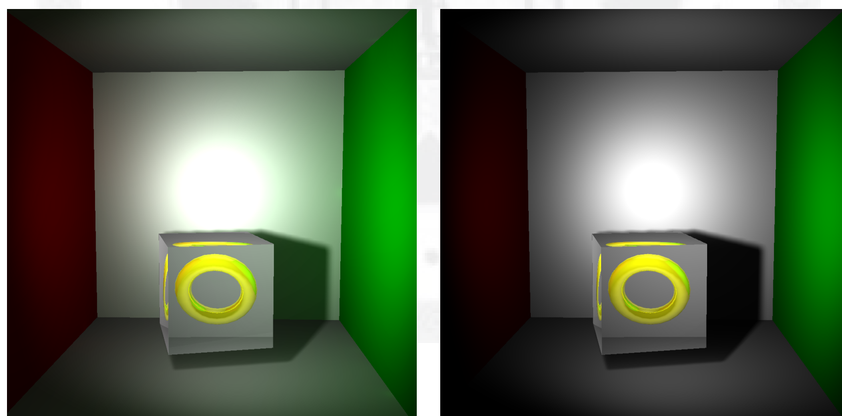


(a) Final image (b) Direct lighting only

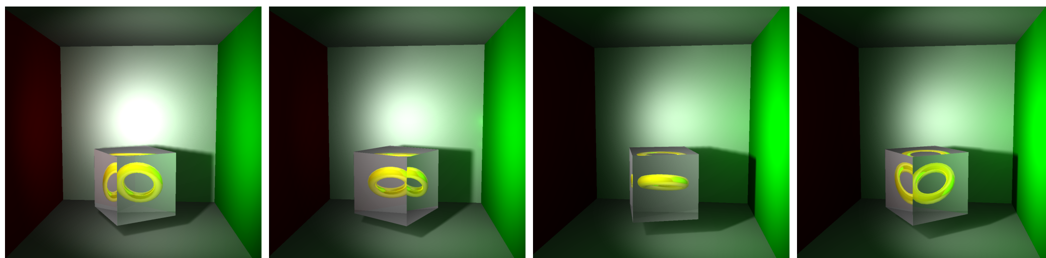


(c) A sequence of animation

Figure 5.2: Rolling box, the frame rate is about 20 fps.

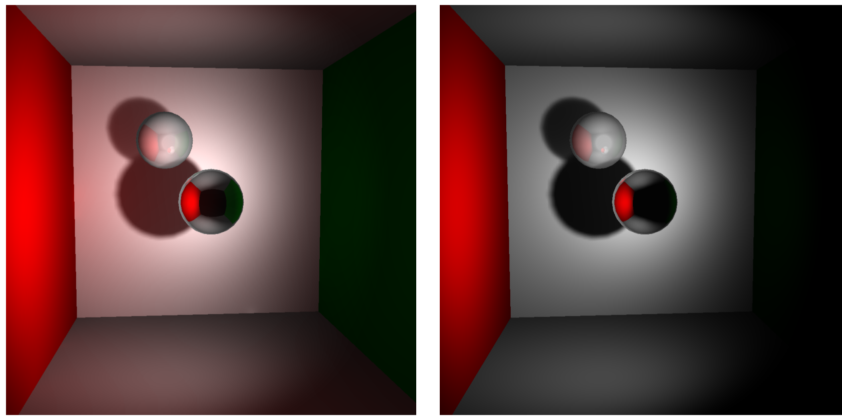


(a) Final image (b) Direct lighting only

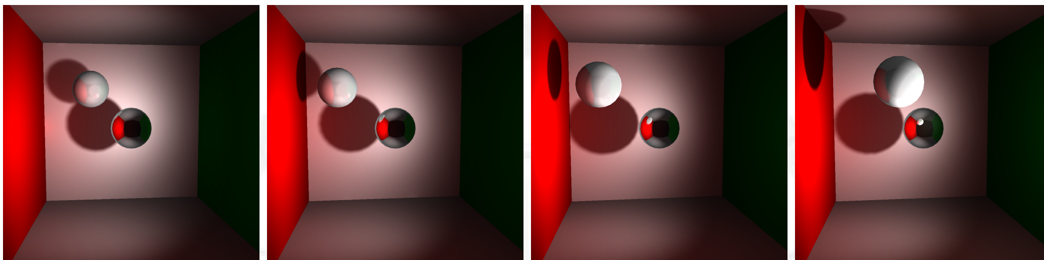


(c) A sequence of animation

Figure 5.3: Rolling box donut, the frame rate is about 7 fps.

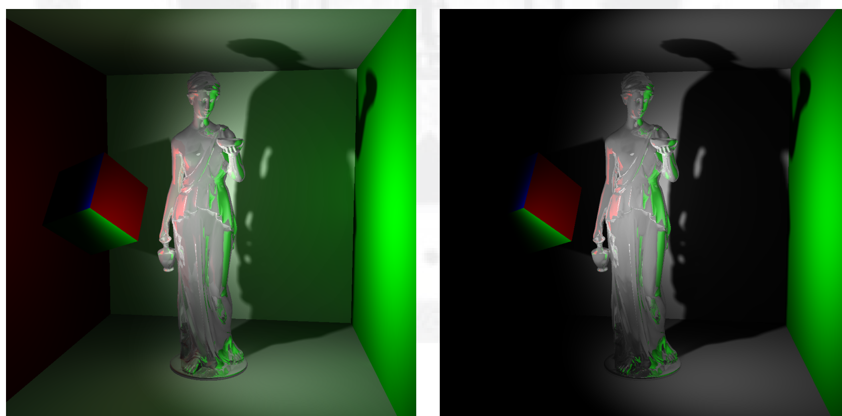


(a) Final image (b) Direct lighting only

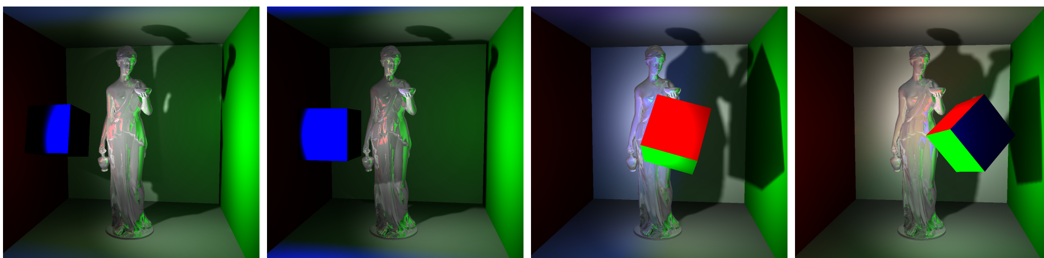


(c) A sequence of animation

Figure 5.4: 2 spheres, the frame rate is about 10 fps.

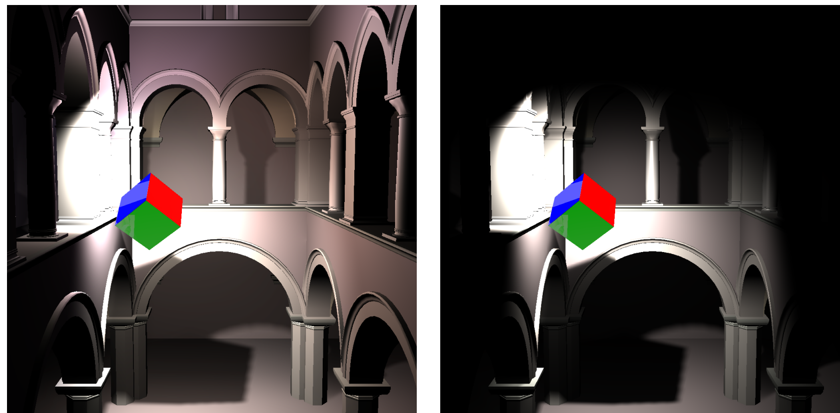


(a) Final image (b) Direct lighting only

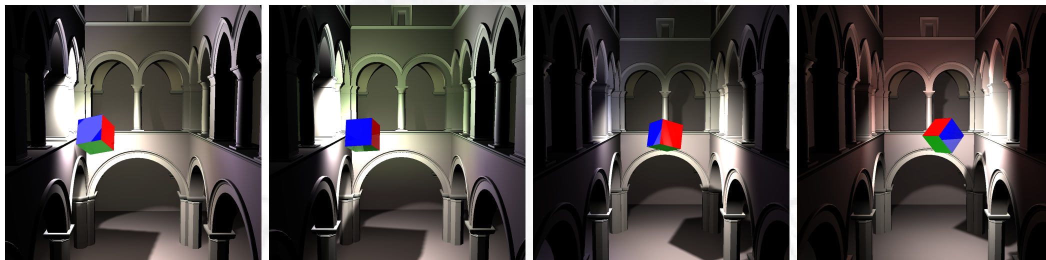


(c) A sequence of animation

Figure 5.5: Rolling box Venus, the frame rate is about 5 fps.



(a) Final image (b) Direct lighting only



(c) A sequence of animation

Figure 5.6: Rolling box Sponza, the frame rate is about 4 fps.

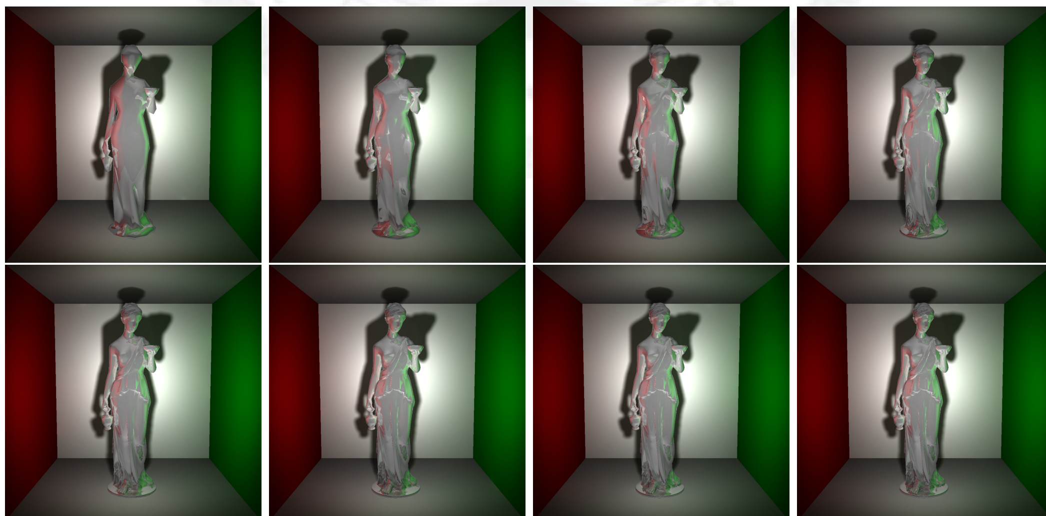


Figure 5.7: Multi-resolution geometric,Venus, for comparing with the complexity of triangles and performance.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we have implemented a rendering pipeline to render most effects of global illumination with fully dynamic scenes, camera, and light sources without precomputation in interactive frame rate. The visual quality is satisfied for an interactive application. Our rendering pipeline was designed for parallel computing thus every processor including all cores of CPUs, GPUs was fully used. Our framework can work sufficiently on the great majority of the mainstream hardware, that we have tested on a mainstream desktop PC and a mainstream laptop PC.

We suffered many difficulties on our implementation. One of the major difficulty is debugging on GPU programming. There are already many tools for shader programming on both Direct3D and OpenGL, because they have developed for a long time. But OpenCL is a new standard, there are few samples and tools to help. There is a situation we usually meet. There is not error when we compiled the kernel code, but when we execute our program, the program crashed and we got no message. We also can't trace the kernel code step by step. The other difficulty is that we can't exactly control the compiler and how our shader code to be compiled when our shader code become larger and more complex. Sometimes we have confidence in our high level shader code would work correctly, but in fact the result is obviously not we want. We almost get it to work by try and run.

6.2 Future Work

Our raytracer can deal with moving objects of rigid body but not soft body. The limitation comes from the acceleration tree, that we don't rebuild kd-tree every frame. We will try to solve this issue by using another efficient acceleration tree. There is an important visual effect missing to our rendering pipeline, caustic. It is time consuming to represent this effect. Next step, we may try to use thousands of photons to trace transparent objects on the raytracer and render those photons efficiently. There are still rooms to improve the performance, many properties can be adjusted. We will work on our framework to make the performance satisfied.

Bibliography

[1] J. F. Blinn. Models of light reflection for computer synthesized pictures. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198, New York, NY, USA, 1977. ACM.

[2] W. Donnelly and A. Lauritzen. Variance shadow maps. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 161–165, New York, NY, USA, 2006. ACM.

[3] H. Gouraud. Continuous shading of curved surfaces. *IEEE Trans. Comput.*, 20(6):623–629, 1971.

[4] M. Hašan, J. Křivánek, B. Walter, and K. Bala. Virtual spherical lights for many-light rendering of glossy scenes. In *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, pages 1–6, New York, NY, USA, 2009. ACM.

[5] M. Hašan, F. Pellacini, and K. Bala. Matrix row-column sampling for the many-light problem. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 26, New York, NY, USA, 2007. ACM.

[6] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-d tree gpu raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174, New York, NY, USA, 2007. ACM.

[7] H. W. Jensen. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 21–30, London, UK, 1996. Springer-Verlag.

[8] A. Keller. Instant radiosity. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[9] S. Laine, H. Saransaari, J. Kontkanen, J. Lehtinen, and T. Aila. Incremental instant radiosity for real-time indirect illumination. In *Proceedings of Eurographics Symposium on Rendering 2007*, pages 277–286. Eurographics Association, 2007.

[10] R. Ng, R. Ramamoorthi, and P. Hanrahan. All-frequency shadows using non-linear wavelet lighting approximation. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 376–381, New York, NY, USA, 2003. ACM.

[11] M. Pharr and G. Humphreys. *Physically Based Rendering*. Morgan Kauffman, 2004.

[12] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Experiences with streaming construction of SAHKD-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 89–94, sep 2006.

[13] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.

[14] B. Segovia, J. C. Iehl, R. Mitanchey, and B. Péroche. Bidirectional instant radiosity. In *In Proceedings of the 17th Eurographics Workshop on Rendering*, pages 389–398, 2006.

[15] B. Segovia, J. C. Iehl, and B. Peroche. Metropolis instant radiosity. *Comput. Graph. Forum*, pages 425–434, 2007.

[16] P.-P. Sloan, J. Kautz, and J. Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 527–536, New York, NY, USA, 2002. ACM.

[17] E. Veach and L. J. Guibas. Metropolis light transport. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 65–76, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[18] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D. P. Greenberg. Lightcuts: a scalable approach to illumination. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1098–1107, New York, NY, USA, 2005. ACM.

[19] T. Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980.

[20] L. Williams. Casting curved shadows on curved surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 270–274, New York, NY, USA, 1978. ACM.

[21] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):1–11, 2008.