VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

PHOTOREALISTIC RENDERING USING PHOTON MAPPING METHOD

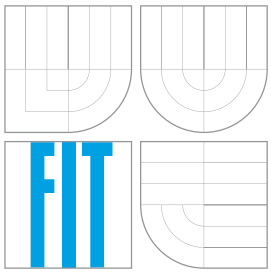
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ LYSEK

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

FOTOREALISTICKÉ ZOBRAZOVÁNÍ METODOU PHOTON MAPPING

PHOTOREALISTIC RENDERING USING PHOTON MAPPING METHOD

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ LYSEK

VEDOUCÍ PRÁCE
SUPERVISOR

Prof. Dr. Ing. PAVEL ZEMČÍK,

BRNO 2015

Abstrakt

Tato práce se zabývá metodou photon mappingu. Nejprve byla naimplementována jednoduchá metoda photon mappingu a poté jeji progresivní varianta byla naimplementována na procesoru a grafické kartě. Po implementaci progressivní varianty photon mappingu na GPU, několik akceleračních technik bylo navrženo. Na konci této práce byl představený genetický klustrovací algoritmus, který se snaží pomocí vhodnějších clusterů urychlit čas výpočtu photon mappingu na gpu.

Abstract

This master thesis focuses on photon mapping rendering technique. A simple photon mapping was implemented as a baseline and then progressive photon mapping was prepared for CPU and GPU. After implementing progressive photon mapping on GPU, further acceleration techniques were proposed. Finally, in the thesis, genetic clustering algorithm for suitable clusters on GPU was proposed.

Klíčová slova

Photon mapping, rendering, globální osvětlení, raytracing

Keywords

Photon mapping, rendering, global illumination, raytracing

Citace

Tomáš Lysek: Photorealistic Rendering Using Photon Mapping Method, diplomová práce, Brno, FIT VUT v Brně, 2015

Photorealistic Rendering Using Photon Mapping Method

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Zemčíka

.....
Tomáš Lysek
May 26, 2015

Poděkování

Chtěl bych poděkovat Pavlovi Zemčíkovi za odborné rady a vedení po celou dobu diplomové práce. Také bych chtěl poděkovat rodině a okruhu blízkých přátel, kteří mě podporovali v této práci.

© Tomáš Lysek, 2015.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

Chapter 1

Introduction

Computer graphics products are in these days everywhere, they are in movies, games, medicine, industry etc. For instance in movies, computer graphics are only way or much cheaper way to create movie about some subject. In medicine, it is nice way to explore some parts of human body without using surgery.

From the beginning of computer graphics, there was an effort to make nice photorealistic images. Physical based rendering is one way how to do it. This class of rendering methods is called global illumination methods.

About one global illumination technique - photon mapping - is this thesis. In this thesis, photon mapping renderer is implemented on CPU and GPU with focus on speed of rendering process.

I personally believe that in near future, graphics processor units have sufficient power to render global illumination methods and global illumination methods will be mainstream and because of this, I chose work on this master's thesis.

In first part of this thesis, theory about photorealistic rendering is described. First, rendering techniques are classified, then each class of rendering technique is described, from easy realtime graphics techniques to advanced physically „correct“ global illumination methods. In second part, implementations of photon mapping are described. This implementations are written with focus on speed. First, simple photon mapping implementation is described. Then on this simple renderer, few tests are performed. After this, progressive photon implementation is described, this implementation was done on CPU and on GPU. Progressive implementation is extended by few acceleration approaches. In the end of this thesis, genetic algorithm is proposed for faster speed of progressive photon mapping.

Chapter 2

Photorealistic rendering

This chapter describes classification of rendering methods and present few needed terms and formulas for classification rendering methods. Later in this chapter object order and image order methods are described. This overview is written exclusively only for this master thesis and should not be understand as encyclopedic or exhaustive overview of photorealistic problematic.

2.1 What is photorealistic rendering

Very simplified definition of rendering is: rendering is process where 2D image is created from 3D representation of scene[20]. This is obviously very vastly defined and there exists lot of approaches how to do this.

Photorealistic rendering is rendering where goal is to create image which is indistinguishable from photograph[20], this goal is typically achieved by correct simulation of light propagation in scene.

2.2 Rendering techniques

Lot of rendering techniques exists, they vary by measure of photorealism and by speed. Basic division of rendering techniques are object order, image order and global illumination.

Object Order

In this type of rendering techniques, each object is independently rendered into framebuffer using for example classic rasterization.

In image 2.1 is shown simple example of this technique, object with largest distance from image plane is rendered in each iteration (if Painter's algorithm is used) and this iterations are performed until all objects are rendered.

In basic variant it's not possible to compute shadows, reflection, refraction and another photorealistic phenomenon because all objects are rendered independently and they do not know any informations about each other[10].

This type of rendering technique is most used in realtime rendering using graphics frameworks like OpenGL and DirectX.

Complexity is linearly by number of objects in scene[10].

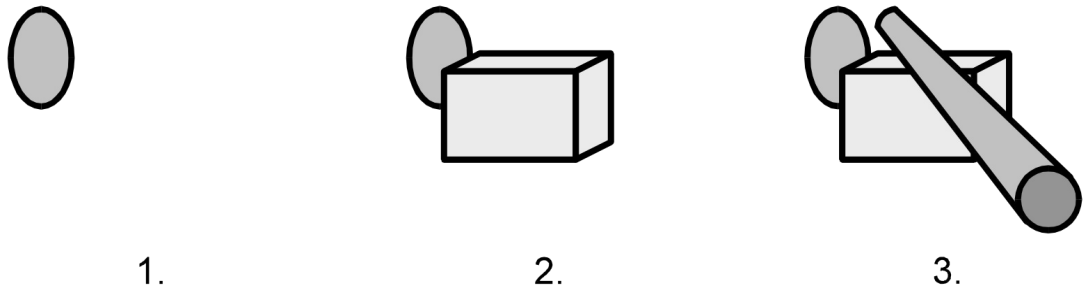


Figure 2.1: Object order[10]

Image Order

In this type of rendering techniques, image is synthesized by examining color of each pixel in framebuffer, by computing rays from viewer through image plane into the scene.

In image 2.2 is shown usage of this technique when image is computed by progressively examining pixels of framebuffer, for instance, from top to bottom, but it is not required to use this schema and it's possible to randomly select each non-evaluated pixel.

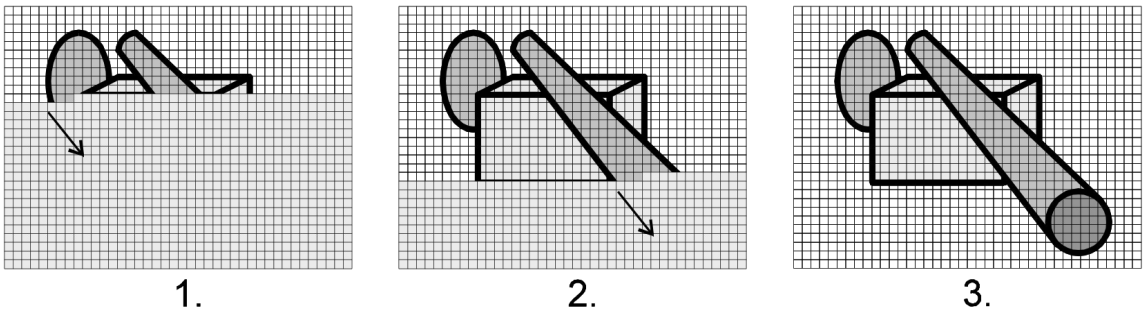


Figure 2.2: Image order[10]

In this type of rendering technique it is possible to easily compute simple shadows, reflection, refraction and another optic phenomenon. It is possible to use massive parallelism to compute image, because each pixel of scene is evaluated independently[10].

Complexity is linearly by number of pixels in image and logarithmically by number of objects in scene (if spatial subdivision is used).

Global Illumination

First, illumination of whole scene is computed and after that, image is computed by using object or image order technique.

Computing illumination of scene can be computed correctly, because for this computation whole scene is provided[10]. (In object order and image order rendering algorithm doesn't have any information about whole scene).

Global illumination provides best computed shadows, reflections, refractions and all physical phenomena, but rendering through this technique is very complex and time consuming and, typically, most of the time is spent on computing illumination of scene[10].

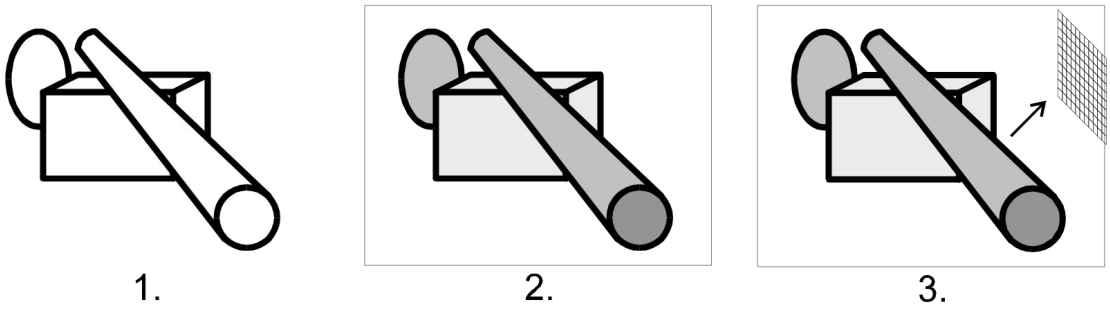


Figure 2.3: Global Illumination[10]

Conclusion

Object order methods are fastest rendering methods, but they are not photorealistic at all. Better photorealistic result provide image order methods and best photorealistic results provide global illumination methods, but this methods are not suitable for usage in realtime rendering and are used for production (offline) rendering, for rendering movies, precomputaion of lighmaps etc.

2.3 Rendering equation

Rendering equation was invented by James T. Kajiya in 1986[15]. Rendering equation solves outgoing radiance for arbitrary point x in direction $\vec{\omega}$. Rendering equation is this equation:

$$L_o(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \int_\Omega f(x,\vec{\omega_r},\vec{\omega_i}) L_o(x,-\vec{\omega_i}) \cos\Theta \, \mathrm{d}\vec{\omega_i} \tag{2.1}$$

where:

- $L_e(x,\vec{\omega})$ is outgoing radiance from point x in direction $\vec{\omega}$ (if x is not light source, this item has zero value)
- integral is integrated through hemisphere Ω with center in point x and this integral represent bounced radiance in point x from all incoming direction from this hemisphere
- $f(x,\vec{\omega},\vec{\omega_i})$ is Bidirectional reflectance distribution function - BRDF (2.4)
- $L_o(x,-\vec{\omega_i})$ is radiance which is incoming into point x from direction $-\vec{\omega_i}$
- $\cos\Theta$ is angle between direction $\vec{\omega_i}$ and surface normal $\vec{n}$ in point x

This equation is basis for all global illumination methods, because this equation compute outgoing radiance as sum of all incoming radiances and because of this, correct illumination of scene could be computed.

2.4 Bidirectional reflectance distribution function - BRDF

In real life, light, which we perceive, is most likely reflected light from some surface[14]. For physical rendering it is crucial to compute correct reflected light. Color of reflected light is defined by spectral characteristic of incoming light and mostly by properties of material.

Bidirectional reflectance distribution function, written by Fred Nicodemus in 1965[19], defines reflectance properties of material.

$$f_r(x, \vec{\omega_r}, \vec{\omega_i}) = \frac{dL_r(x, \vec{\omega_r})}{dL_i(x, \vec{\omega_i})(\vec{\omega_i}.\vec{n})d\vec{\omega_i}} \tag{2.2}$$

Image 2.4 describes bidirectional reflectance distribution function 2.2. Let assume that light is incoming from direction $\vec{\omega_i}$ and we are computing light in direction $\vec{\omega_r}$ in point x. BRDF is denoted as $f_r(x, \vec{\omega_r}, \vec{\omega_i})$ and define fraction of reflected radiance $dL_r(x, \vec{\omega_r})$ and incoming radiance $dL_i(x, \vec{\omega_i})$[14].

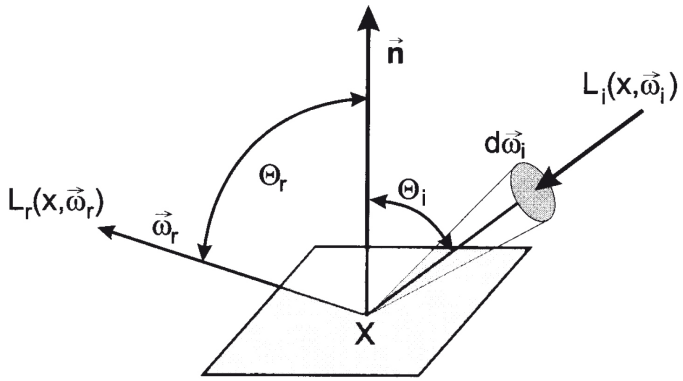


Figure 2.4: Bidirectional reflectance distribution function[14]

Some important BRDF properties[14]:

- *Positivity* - BRDF function is never negative $f_r(x, \vec{\omega_r}, \vec{\omega_i}) \geq 0$
- *Linearity* - BRDF value for some incoming angle $\vec{\omega_i}$ is not dependent on BRDF values for others incoming angles. This brdf property means, that outgoing ray is reflected regardless what comes from other directions.
- *Helmholtz reciprocity* - BRDF value in point x is same if we switch incoming and outgoing direction: $f_r(x, \vec{\omega_r}, \vec{\omega_i}) = f_r(x, \vec{\omega_i}, \vec{\omega_r})$

2.5 Biased, Unbiased and consistent methods

One way to classify rendering methods is classify to biased/unbiased and consistent methods.

Consistent rendering method means, that with increasing iteration (with increasing rendering time) quality of rendered method is getting better and image will converge to correct result.

Unbiased rendering methods often used some simplification, for instance blur some results, use interpolation etc.

Both biased and unbiased methods can be consistent. Difference between biased and unbiased methods is this[16]: „if I rendered the same image millions of times using different random numbers, would averaging the results give me the right answer?“. If answer is no, algorithm has bias.

2.6 Photorealistic elements

For good photorealistic image, it is crucial to have rendered some elements like shadows, refractions etc. If this elements will not be rendered, image does not look like from real world and this image is not realistic. Each rendering method computes this elements with some difficulty. This section will describe this elements and further in this text, in each rendering algorithm, will be described elements which are possible to compute using specific rendering algorithm.

Shadows

Shadows are areas in scene where light from light source is obstructed by some object. One type of shadows are hard shadows - this type of shadows have hard transition on edge between shadow area and non-shadow area.

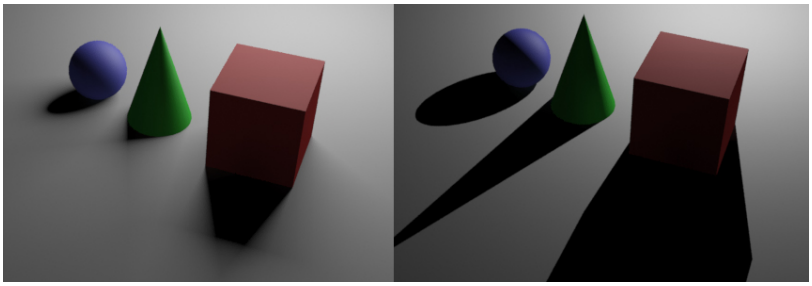


Figure 2.5: Difference between soft (left) and hard (right) shadows

Other type of shadows are soft shadow which has transition on edge between shadow area and non shadow area. Type of shadow is specified by type of light, for instance point light make only hard shadow and sphere (or area) light make soft shadow.

Reflections

The simplest example of reflections is mirror. Reflections are created on special material which reflects rays, it could be water, mirror, glass. And it could not only fully reflect light but it could only partially reflect light (polished material). So reflections is property of material.

Refraction

Refractions are made when light is going through transparent objects. Direction of refracted object is given by Snell's law and refracted ray direction is depended on refraction index of incoming material.

Caustics

Caustics are created when light rays are concentrated by refraction or reflections into little areas and there create shiny area. This physical phenomenon is shown in picture 2.6.

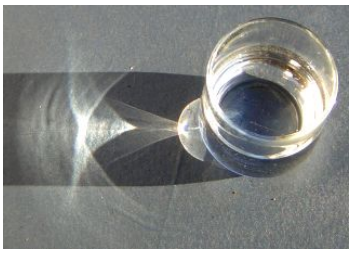


Figure 2.6: Caustics projected by glass of water[24]

Specular diffuse specular path (SDS path)

This is a special case of light bounce, where light is going through specular (transparent) object, then is reflected on diffuse material and then again going through specular object. This special case occurs for instance on the bottom of swimming pool.

Color bleeding

Color bleeding effect is created when light is reflected from material and this light takes material color, then when this colored light hit another surface, it seems as light has different color than normal.

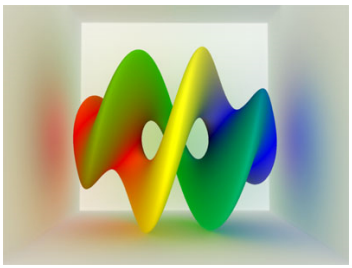


Figure 2.7: Color bleeding in simple scene[7]

2.7 Object order methods

This type of rendering methods are mostly used in realtime applications such as games, fast visualization etc. This section will describe basis of mostly used algorithm in realtime rendering.

Rendering process of object order methods is possible to divide to some block which make graphics rendering pipeline and this pipeline is possible to divide to three[1] stages: application, geometry and rasterization.

Application stage prepare scene for later use, this stage load scene from file, compute scene graph, configure graphics card etc.

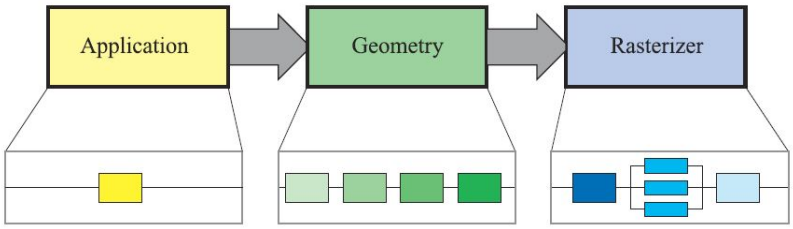


Figure 2.8: Object order rendering pipeline[1]

Figure 2.9: Blocks of geometric stage[1]

Geometric stage is possible to divide into five blocks, input to this stage is 3D model of scene (typical scene is made from triangles, so it is set of three points in 3D scene), output of this stage is set of triangles projected into 2D plane. Details of each stage:

- Model and View Transform - this block transform objects of scene into world coordinates. Typically objects are stored in object coordinates, model transform move objects to proper position and view transform has the task to position camera.
- Vertex shading - In this block, illumination of triangle is computed. This illumination is computed only for vertices of triangle and this operation is known as shading.
- Projection - This block transform scene from 3D space into 2D space. This goal is achieved by using projection. Projection is operation which transform view frustrum into unit cube[1]. There are two types of projection: orthogonal and perspective. Simply put: after orthogonal projection, parallel lines are parallel, distances are same. Perspective projection is little bit difficult, it mimics how people perceive world through eyes, far objects are smaller than object near before camera, parallel line may converge etc.
- Clipping - Only primitives which lie wholly or partially inside view volume are kept, others are discarded.
- Screen Mapping - This block transform 2D triangles from unit cube into screen coordinates.

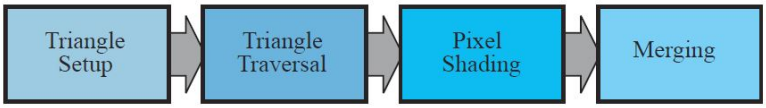


Figure 2.10: Blocks of rasterizer stage[1]

Rasterizer stage is possible to divide into four blocks, input to this stage is set of triangles projected on projection plane. Output of this stage is rendered image.

- Triangle setup - In this block, important data for triangle traversal (used in next block) is computed.
- Triangle Traversal - In this block, triangle is divided into tiny little fragments - using traversal algorithm[1]. This fragments correspond to pixel in framebuffer. Not only center of fragment is computed, among with this information, color and other data from geometry stage are interpolated too.
- Pixel shading - In this block, fragment is colored by informations given from previous blocks, it is possible to compute accurate illumination. Output of this block is color and (most often) depth.
- Merging - This block make from set of fragments output image. Problem of this block is: few fragments can correspond to one pixel on screen, so which one take? This task is called visibility problem and for this problem is possible to use painter's algorithm, stencil buffer od z-buffer[1]. Probably most used algorithm is z-buffer which store among fragment his depth and when fragment is putted into framebuffer, into z-buffer is saved depth of current fragment. Update of pixel is performed only if depth of fragment is lesser than depth of saved fragment.

Shadows

In object order method, each triangle is processed independently. This approach make difficult to determine if object has obstruction between surface and light or not.

There exist lot of approaches to do this, but probably two most used algorithms are shadow mapping and shadow volumes.

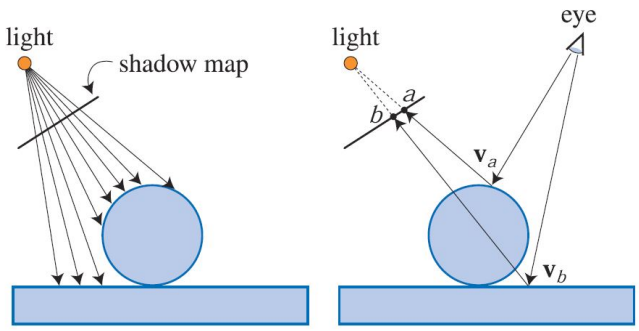


Figure 2.11: Shadow mapping[1]

In shadow mapping technique, scene is rendered from light source. This render pass is for creation special map called shadow map. Depths of rendered objects are saved in this shadow map. When scene is rendered from camera view, distance from rendered surface from position x to light is computed and from shadow map is get value for direction from light to position x and if value in shadow map is lesser than computed distance, position x is in shadow. Using this approach it is possible to create fast shadows, but this approach has lot of problem, shadows aren't precise and it isn't robust. Common problems of this technique are resolution of shadow map, self shadowing[1] etc.

Shadow volumes (in some literature called stencil shadow[18]) is accurate shadow technique. First, for all shadow-casting objects are created shadow volumes - it is area where

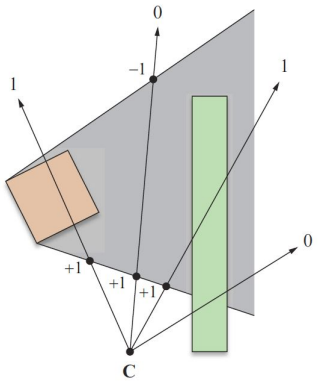


Figure 2.12: Shadow volumes[18]. Brown object is shadow-casting object, gray area is shadow volume, C is camera

object cast shadow. Then, in rendering pass from camera view, this invisible shadow volumes are rendered into stencil buffer to create special mask. When scene is rendered, light is computed only for those pixels, where mask - in stencil buffer - is zero. Using this approach it is possible to render pixel-precise shadows, this technique is very time consuming for scenes with many light.

Reflections

Again, as with shadow effects in object order methods, this rendering method can not compute reflection with ease, because information about scene is not given when illumination is computed. One way how to create reflections in object order methods is render scene from another view, save this image into texture and use this texture on material with reflection.

Refraction

Rendering transparent objects in this techniques is little hard. Easiest way to do this, is render objects from back to front (in view from camera) and use technique called alpha blending, when color in framebuffer is summed and not overwritten.

Using this approach for physically based rendering for transparent object is not possible, because of refraction on transparent objects. Using alpha blending is accurate only if refraction index of both (incoming and outcoming) materials are same. It is not possible to compute curve angle of refraction, so rendering refraction on, for instance, glass ball it is needed to render scene in special pass[1], with special curved mapping and then result of this render use as texture on „transparent“ object.

Caustics

Lot of techniques for creating caustics exist, for example Interpolated Warped Volumes[6]. In this technique, each triangle which is able to create or receive caustics are tagged if this triangle generate or receive caustic. For each generating triangle, caustic volume from edges of this triangle is computed (direction of volume is computed from light direction, surface normal and refraction index - if object is transparent or from direction of reflective ray if material is reflective). This volume represent if caustic light diverges or converges

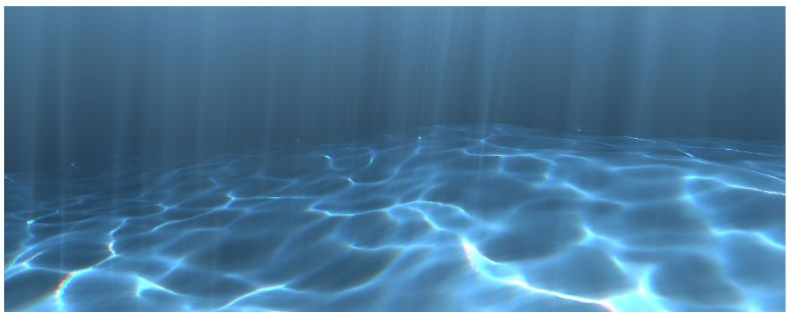


Figure 2.13: Caustics rendered using Interpolated Warped Volumes[6]

to some point and special function is computed indicating caustic intensity by distance from generating triangle. Then in rendering pass, for each fragment, lied in caustic volume, accumulated light contribution from caustics is computed.

2.8 Raycasting

Raycasting could be called very old technique because in 1525 Albrech Durer created raycasting machine through he was attempting to paint images. First computer graphics raycasting was implemented by Artur Appel in 1968[2] in beginning of computer graphics. Raycasting is object order method.

Figure 2.14: Albrecht Durers raycasting machine

In this method, for each pixel in framebuffer ray is casted from camera (from this raycasting) into scene and for this ray intersection with scene is computed. In intersection point, illumination is computed and this illumination is saved as color of pixel into result image. Basic variant doesn't compute shadows. Compared to object order techniques it does not require usage of visibility algorithm, because to framebuffer are saved only values from nearest objects in scene.

Altought this method is old, with modification this technique is still used in medicine. This modification is called volume ray casting and is used for displaying data from tomography or CT where volumetric representation of the patient's tissue is get from CT.

2.9 Raytracing

Raytracing is extension of raycasting where not only first intersection through scene is computed but also others intersections are computed recursively from this intersection.

Raytracing was first introduced by Turner Whited in 1980[23] and is basis for nearly all advanced illumination techniques.

Scheme of raytracing is shown at image 2.15 where few rays are involved, this rays are:

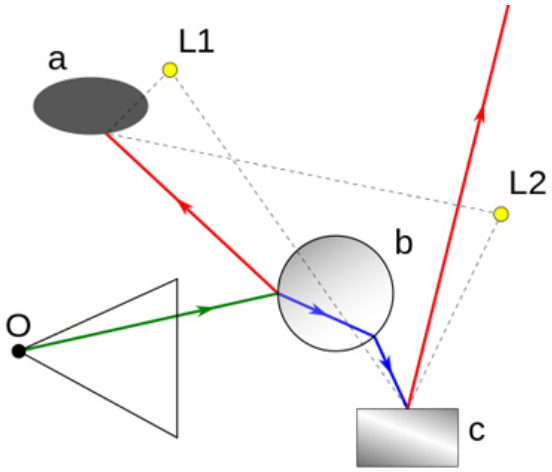


Figure 2.15: Raytracing schema[25]

- *Primary rays* - This rays are send from camera O to scene (same as in ray casting) and illumination is computed in intersection of ray and scene. Illumination is computed from recrusively sended secondary rays and from shadow rays. Computed color of this ray is color of saved pixel in framebuffer. In demonstration image this ray has green color.
- *Secondary rays* - This rays are created from reflection or/and refraction (depending on material properties) primary rays. This rays became primary rays and are recursively traced through scene like primary rays (this secondary primary rays generate secondary rays again). In demonstration image, reflective secondary ray is red, refracted secondary ray is blue.
- *Shadow rays* - Shadow rays are computed between intersection point and all light sources, when illumination is computed and thanks to this rays, correct shadows are computed. In demonstration image, shadow ray is dashed line.

Raytracing algorithm[14]: let have function *RayTrace*(ray R, depth of recursion D):

1. Find intersection point P of ray R with closest object in scene.
2. If intersection P doesn't exist (ray leave scene space), give ray R color of background and return.
3. For each light source send shadow ray from point P and if this ray doesn't have any obstruction mark this light source as not obstructed
4. Compute illumination in point P from all not obstructed light sources
5. If depth of recursion D isn't lower or equal zero send:
   - reflected ray R_R calling RayTrace(R_R, $H - 1$)

- refracted ray R_T calling RayTrace(R_T, $H - 1$)

6. Give ray R final collor as sum of illuminatio and color of reflected/refracted ray

Shadows, reflections and refractions

In raytracing, all shadows, reflections and refractions are computed precisely because in computing illumination through raytracing geometry of scene is included and therefore correct computation of this phenomena is possible.

Caustics

It is not possible to render physically correct caustics in default raytracing, because caustics are created from light rays bounced through scene. In raytracing, light is computed only from intersection points and it is not possible to deterministically send „caustic“ ray through scene and expect that ray will hit light source.

SDS path, color bleed and indirect illumination

This effects can't be rendered through default raytracing, because advanced techniques is needed and usage of this technique means that used method is not raytracing but (for instance) path tracing or photon mapping. All other techniques (except radiosity) are based on simple raytracing so it is possible to say that raytracing with upgrade to higher technique is capable to compute this photorealistic effects.

2.10 Distributed raytracing

Normal raytracing is very good in computing precise hard photorealistic elements like hard shadows or accurate mirror-like reflection. Problem occurs when area lights are in scene and soft shadows or blurred reflection are required. Reason why normal raytracing isn't capable render this advanced elements is that raytracing determines every ray direction precisely and this limit raytracing capabilities.

Robert L. Cook proposed in 1984 extension called distributed raytracing[5], where rays are not computed precisely from point to point but are sampled over some distribution function.

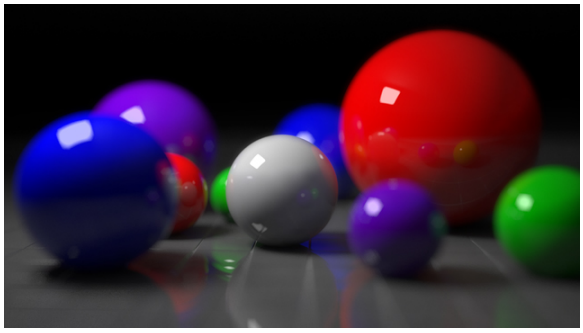


Figure 2.16: Blurred reflections and depth of field effect rendered by distributed raytracing[27]

Secondary and shadow rays are sampled over distribution function. Distribution function gives direction of secondary rays (reflected, refracted, shadow). Sampling this function is performed by selecting random direction from this distribution function, compute illumination for each sampled ray and result is averaged over all sent rays.

Photorealistic elements

Using this distribution approach it is possible to compute lot of advanced photorealistic elements. Rendering soft shadows are possible by distributing shadow rays over direction of area light. Rendering glossy reflection is possible distributing secondary rays alongside precisely computed reflected ray. Rendering depth of field is possible distributing „maximum length of ray“. Motion blur is achieved by using distribution not in space but in time.

Chapter 3

Global Illumination methods

This chapter describes global illumination techniques and describes radiosity, path tracing, bidirectionaly path tracing, photon mapping and progressive photon mapping.

3.1 Radiosity

Radiosity is one of first global illumination techniques trying to compute indirect illumination of scene. This technique is based on heat transfer radiosity methods founded in about 1950. First usage in computer graphics was in 1984 by researches at Cornell university[26].

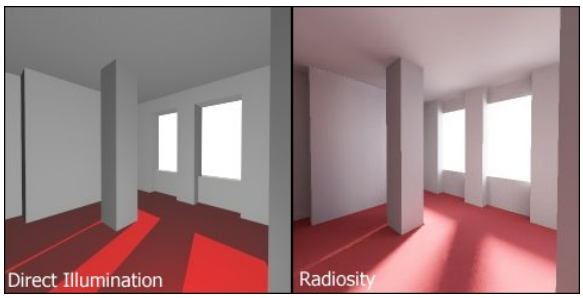


Figure 3.1: Difference between direct illumination and radiosity[26]

Radiosity is based on law of conservation of energy and this technique assumses light propagation in closed scene with only diffuse materials. First continuous equation of radiosity computation was proposed.

$$B(x) = E(x) + \rho(x) \int_S B(x')G(x, x')\,\mathrm{d}x' \tag{3.1}$$

Correct illumination is computed solving this equation, but it is nearly impossible solve this equation for complex scene. Therefore, discretization of this equation was proposed. Discretization meaning dividing scene in lot of little surfaces.

$$B_i = E_i + \rho_i \sum_{j=1}^{n} B_j F_{ij} \tag{3.2}$$

where:

- B_i is constant radiosity of surface i
- E_i is radiosity emmiting from surface i
- ρ_i is diffuse reflectance of surface i. This item has value between 0 (complete absorption of incoming light) or 1 (doesn't absorb any light - everything is reflected to scene)
- Sum represent all light bounced from others surfaces to surface i
- F_{ij} is form factor between surface i and surface j and this factor indicates how much light is taken by surface i from light emitted from surface j

For rendering scene using this method, first discretization of scene is needed. This means that all surface have to be divided into little surfaces. Number of surfaces affects speed and correctness of this algorithm. Then for each surface, form factor has to be computed. Value of this factor is proportional to visibility of each surface, angle between normals and size of surfaces. After this precomputation stage, algorithm has everything needed and computation of illumination could begin.

Computation images through radiosity means solving system of linear equations therefore one data matrix is created and this matrix is solved using classical mathematics algorithm like Gauss Seidel or Jacobi method.

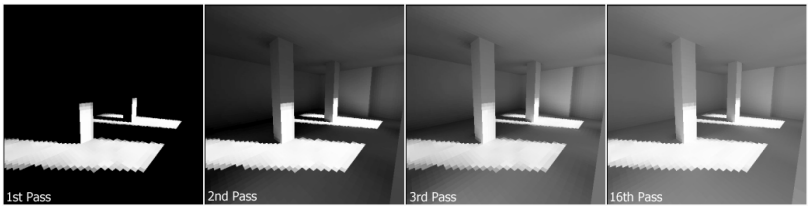


Figure 3.2: Progressive radiosity[26]

Usage of Gauss Seidel or Jacobi method isn't optimal[14]. This methods tend to be slow because illumination of light sources can be added to computation in any time and then all computation of non light sources have to be done again. In 1988 Cohen[4] invented special algorithm for radiosity which is trying to prioritize emiting light from light source first and then compute reflections from surfaces affected by this light, this approach is called progressive radiosity.

Photorealistic elements

Radiosity can compute only diffuse illumination of scene and therefore this method can compute only indirect illumination and color bleed. Using radiosity method it is not possible to compute reflected or refracted light beams. This global illumination method is suitable only for scenes with diffuse materials.

3.2 Path tracing

Path tracing was introduced by James T. Kajiya in 1986[15] with Rendering equation as a solution for this equation. Path tracing is an extension of raytracing and is unbiased consistent global illumination method.

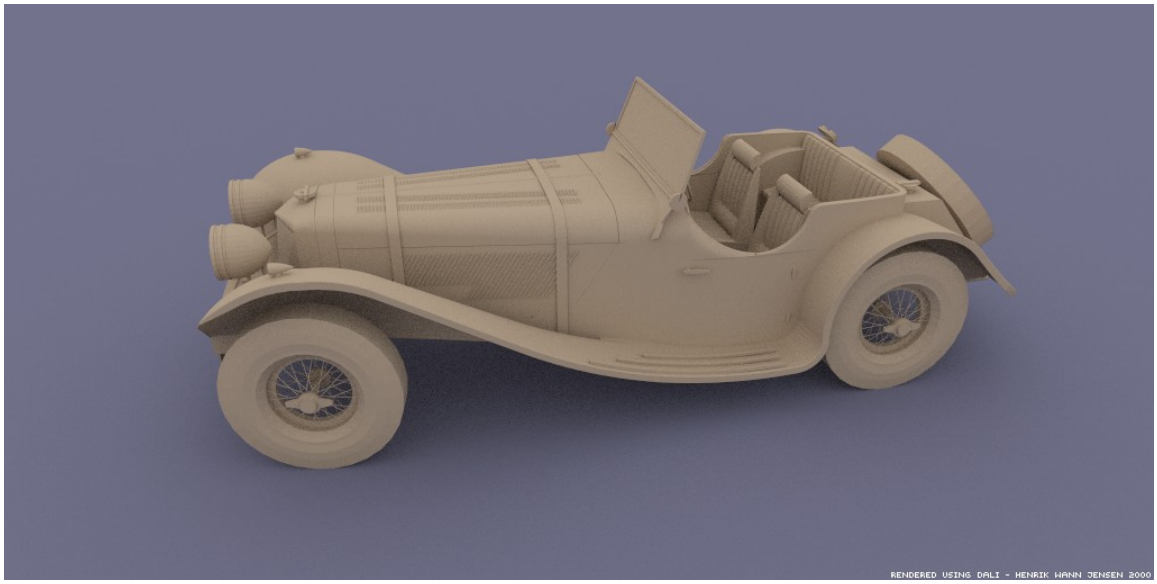


Figure 3.3: Path traced scene[12]

Path tracing is based on very similar principle as distributed raytracing with one difference. Rays are not sampled through distribution function in intersection points and their average is taken as result, but for each intersection only one ray is traversed in direction of distribution function (alongside with shadow rays computing light contribution).

For good noise-free images lot of rays per pixel have to be traversed, number of rays per pixel is depending on scene but it is normal to compute ten thousands random paths per pixel to achieve good noise free images. Good realistic illumination with hundred paths per pixel can be achieved in scenes with lot of direct illumination. Image 3.3 shows path traced image using 100 paths/pixel - notice noise areas in shadows, in this area more paths per pixel should be used.

Because this method is using for computing integral from rendering equation monte carlo way(random samples and their averaging) to compute integral, this class of global illumination methods are called monte carlo raytracing methods. Problem with classic path tracing is noise - computation error. Same as in monte carlo algorithm, path tracing has error $\frac{1}{sqrt(N)}$ where N is number of samples. This means that to halve the error, four times more samples are required[12].

Photorealistic elements

Path tracing is using raytracing as basis of algorithm, so it inherit same photorealistic elements like raytracing (even distributed raytracing). Using path tracing it is possible to compute complete global illumination - caustics and color bleed effect. Path tracing is capable to compute SDS paths but this paths have lot of noise.

3.3 Bidirectional Path tracing

Bidirectional path tracing is an extension of normal path tracing and was introduced by Lafortune[17] in 1993 and independently by Veach[22] in 1994. Bidirectional path tracing traces paths from light and from camera at once.

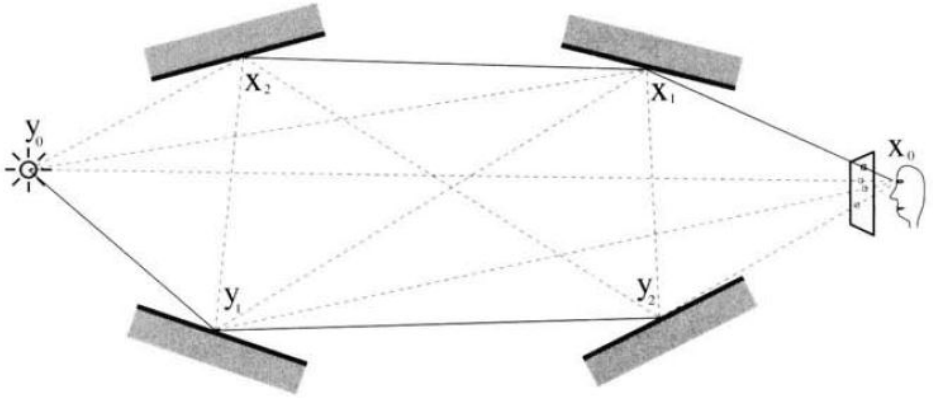


Figure 3.4: Bidirectional path tracing schema[12]

Image 3.4 shows schema of bidirectional path tracing. In this type of path tracing, two paths are investigated through scene. First is eye path, going trough pixel on image plane and has vertices denoted as x_i. Second path is starting at light source and vertices of this light path is denoted as y_i.

After traversing two paths (eye and light path) illumination between light path and eye path is computed.
Illumination of entire path starting at pixel in camera is computed as:

$$L_p = \sum_{i=0}^{N_i} \sum_{j=0}^{N_j} w_{i,j} L_{i,j} \tag{3.3}$$

where N_i is number of eye path vertices and N_j is number of light path vertices. Reflected radiance from point x_i to point x_{i-1} by contribution light vertex y_j denoted as $L_{i,j}$ is computed as[12]:

$$L_{i,j}(x_i \to x_{i-1}) = f_r(y_j \to x_i \to x_{i-1}) V(x_i, y_j) \frac{|(y_j \to x_i).\vec{n_{x_i}}|}{||x_i - y_j||^2} I(y_j \to x_i) \tag{3.4}$$

where: $f_r(y_j \to x_i \to x_{i-1})$ is BRDF in point x_i with incoming direction to this point from point y_j and outgoing direction to point x_{i-1}. $V(x_i, y_j)$ is visibility function, in image 3.4 is denoted as dashed line - shadow ray - and this function says if between points x_i and y_j isn't obstruction. Radiant intensity $I(y_j \to x_i)$ is computed as:

$$I(y_j \to x_i) = \Phi_i(y_j) |(y_j \to x_i).\vec{n}_{y_j}| f_r(y_{j-1} \to y_j \to x_i) \tag{3.5}$$

where $\Phi_i(y_j)$ is flux of incoming photon at y_j and this flux is weighted by brdf function $f_r(y_{j-1} \to y_j \to x_i)$.

For good photorealistic images lot of computation of path per pixels is needed. Using bidirectional path tracing it is possible to achieve good photorealistic results with lesser number of paths per pixel than in normal path tracing in some type of scenes[12]. Scenes with caustics and lot of indirect illumination are more suitable for bidirectional path tracing.

Photorealistic elements

Bidirectional path tracing is capable to render nearly every focused photorealistic elements. One problematic element is still SDS path where bidirectional path tracing and normal path tracing fails and lot of noise are in scenes with this paths.

3.4 Photon mapping

Photon mapping is two pass global illumination method. This method render with great accuracy all focused photorealistic elements including indirect illumination and SDS paths. Photon mapping was introduced in 1995 by Henrik Wann Jensen[13].

First pass - photon map creating

In first pass, photon map is created. Photon map is saved light contribution from all light sources, distributed in scene and this map is get by sampling light contribution to scene. For each light N photons is sent from light into scene. Photon is not same like in physics, but it is fraction of light power - it is a bigger chunk of light energy (flux) sent into the scene.

When photon is sent from light, this photon is investigate in same manner as ray in normal raytracing - photon is bouncing through scene. On every hit with non specular surface, position, photon flux $\Delta\Phi_p(x, \vec{\omega_p})$ and direction are saved in photon map and new photon is generated and sent into the scene. Direction and power of generated photon is determined from BRDF of intersected material.

As many as possible photons is generated for best photorealistic result. This lead into very huge photon map with few milions of photon saved. In second pass, lot of searches are perfomed in this map, thus it is appropriate to create acceleration structure for fast searching, for instance kd-tree.

Second pass - rendering

In second pass, raytracing is used for computing final image. For better photorealistic result, it is possible to use distributed raytracing. In photon mapping, raytracing is extended to compute indirect illumination from photon map. Indirect illumination in point x, $L_r(x, \vec{\omega})$ is approximated from rendering equation, when reflected light in point x and in direction $\vec{\omega}$ equals:

$$L_r(x, \vec{\omega}) = \int_\Omega f(x, \vec{\omega}, \vec{\omega_i}) L_i(x, \vec{\omega_i}) \cos\Theta \, \mathrm{d}\vec{\omega_i} \tag{3.6}$$

Incoming radiance $L_i(x, \vec{\omega_i})$ is replaced by:

$$L_i(x, \vec{\omega_i}) = \frac{\mathrm{d}^2\Phi_i(x, \vec{\omega_i})}{\cos\Theta \, \mathrm{d}\vec{\omega_i} \, \mathrm{d}A} \tag{3.7}$$

Which is relationship between luminance and flux (saved in photon map), after few modification this equation is get:

$$L_r(x,\vec{\omega}) = \int_\Omega f(x,\vec{\omega},\vec{\omega_i}) \frac{\mathrm{d}^2\Phi_i(x,\vec{\omega_i})}{\mathrm{d}A} \tag{3.8}$$

This integral is possible to aproximate as sum:

$$L_r(x,\vec{\omega}) = \int_\Omega f(x,\vec{\omega},\vec{\omega_i}) \frac{\mathrm{d}^2\Phi_i(x,\vec{\omega_i})}{\mathrm{d}A} \approx \sum_{p=1}^{N} f(x,\vec{\omega},\vec{\omega_p}) \frac{\Delta\Phi_p(x,\vec{\omega_p})}{\Delta A} \tag{3.9}$$

where N is number of N nearest photons from point x. This nearest photons will lie inside sphere with center at point x. Point x and his nearest neighborhood often lie on plane, so it is possible to approximate ΔA as circle and thus: $\Delta A = \pi r^2$. Final formula of indirect illumination computed from photon map is:

$$L_r(x,\vec{\omega}) \approx \frac{1}{\pi r^2} \sum_{p=1}^{N} f(x,\vec{\omega},\vec{\omega_p}) \Delta\Phi_p(x,\vec{\omega_p}) \tag{3.10}$$

For summary, in photon mapping, first photon map is created, then, in second pass, normal raytracing is used. This raytracing add into local illumination model on non specular surfaces computation of indirect illumination from photon map as weighted (by BRDF) sum of N nearest photons.

This method is consistent and biased. Photons, and their connections between photon and examining points, approximate paths from light to camera. This connections will never be done precisely and bias is created from this approximations[12].

Photorealistic elements

Photon mapping is based on (distributed) raytracing, so it inherit all raytracings photorealistic elements. However, photon mapping is capable to compute full global illumination, caustics and all SDS paths. Photon mapping is faster than radiosity and monte carlo raytraced methods[11].

3.5 Progressive photon mapping

Progressive photon mapping is reworked version of normal photon mapping, this method was described in 2008 by Hachisuka[8].

Problem in normal photon mapping is in radiance estimate (equation 3.10). This radiance estimate is source of bias in photon mapping. To reduce bias, large photon map has to be created. To completely remove bias, theoretically infinite number of photons has to be created, this means that radius of nearest search, in photon map with infinite number of photons, will converge to zero[12]. Using infinite number of photon in photon map in is only in theoretical interest because of limited size of computers memory and extremely big time complexity.

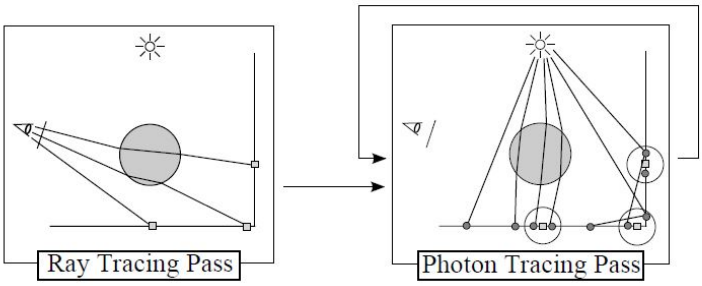


Figure 3.5: Progressive photon mapping schema[8]

Reordered photon mapping

Progressive photon mapping tries to reorganize standard photon mapping to not store one big photon map, but it tries to create lot of smaller photon maps instead and progressively computing illumination from this smaller photon maps.

As is shown in figure 3.5, raytracing is performed on scene first. This raytracing is extended in one feature: after examined ray hits diffuse surface, point of this intersection is saved, this point is called hitpoint. Hitpoint save position x, surface normal $\vec{N}$, ray direction $\vec{\omega}$, pixel location x, y, pixel weight wgt, current photon radius R, accumulated photon count N, accumulated reflected flux τ. Using only hitpoint values it is possible to assemble final image.

In other passes, photon tracing is computed in similar way as in normal photon mapping. After each photon tracing pass, illumination is computed for each hitpoint from computed photon map.

Progressive radiance estimate

In traditional photon mapping, illumination is computed from searching nearest photons and this photons is added to calculation local density $d(x)$ of photons. This local density is computed as:

$$d(x) = \frac{n}{\pi r^2} \tag{3.11}$$

where n is nearest photons within a sphere with radius r. If another photon tracing pass is used and local density is computed from same position x and even with same radius r, then value of local density function will be different - $d'(x)$ - because of different photons in radius r.

It is possible to compute average value of $d(x)$ and $d'(x)$, this will lead to smoother radiance estimate, but final result does not have more detail than each individual photon map[8] and this approach will not be consistent.

For consistent algorithm, radius has to be reduced with every iteration. Progressive photon mapping came with progressive radius reduction. Let assume that new density estimate $\hat{d}(x)$ is computed as:

$$\hat{d}(x) = \frac{N(x) + M(x)}{\pi R(x)^2} \tag{3.12}$$

where $N(x)$ is number of photons already saved in hitpoint x, $M(x)$ is number of photons from *new* photon tracing pass (in appropriate radius from x), $R(x)$ is radius in hitpoint x. Next step is reducing radius $R(x)$ by $dR(x)$, it is possible to compute new total number of photons $\hat{N}(x)$ in new radius $\hat{R}(x) = R(x) - dR(x)$ as:

$$\hat{N}(x) = \pi \hat{R}(x)^2 \hat{d}(x) = \pi (R(x) - dR(x))^2 \hat{d}(x) \tag{3.13}$$

and new photon count $\hat{N}(x)$ can be expressed as sum of already processed photons $N(x)$ and fraction (α is in range (0 - 1)) of new traced photons $M(x)$ as follows:

$$\hat{N}(x) = N(x) + \alpha M(x) \tag{3.14}$$

then, by combining this three equation it is possible to compute $dR(x)$:

$$\begin{aligned} \pi (R(x) - dR(x))^2 \hat{d}(x) &= \hat{N}(x) \\ \pi (R(x) - dR(x))^2 \frac{N(x) + M(x)}{\pi R(x)^2} &= N(x) + \alpha M(x) \\ dR(x) = R(x) - dR(x) &= R(x) \sqrt{\frac{N(x) + \alpha M(x)}{N(x) + M(x)}} \end{aligned} \tag{3.15}$$

and finally reduced radius $\hat{R}(x)$ is computed as:

$$\hat{R}(x) = R(x) - dR(x) = R(x) \sqrt{\frac{N(x) + \alpha M(x)}{N(x) + M(x)}} \tag{3.16}$$

Flux correction and radiance evaluation

After each photon trace pass, hitpoint receive another $M(x)$ photons carrying some flux. This flux has to be accumulated in proper way to already accumulated flux $\tau_N(x, \vec{\omega})$ from previous iteration. Accumulated flux of new photons $\tau_M(x, \vec{\omega})$ in position x and in direction $\vec{\omega}$ is computed as:

$$\tau_M(x, \vec{\omega}) = \sum_{p=1}^{M(x)} f_r(x, \vec{\omega}, \vec{\omega}_p) \Phi'(x_p, \vec{\omega}_p) \tag{3.17}$$

If radius does not change, simple addition between this two flux will be performed to get final accumulated flux $\tau_{\hat{N}}(x, \vec{\omega})$, but because radius is reduced after each pass, accumulated flux has to be scaled as follows:

$$\tau_{\hat{N}}(x, \vec{\omega}) = \tau_{N+M}(x, \vec{\omega}) \frac{N(x) + \alpha M(x)}{N(x) + M(x)} \tag{3.18}$$

where $\tau_{N+M}(x, \vec{\omega}) = \tau_N(x, \vec{\omega}) + \tau_M(x, \vec{\omega})$. After this correction, it is possible to compute radiance at point x as:

$$
\begin{aligned}
L(x,\vec{\omega}) &= \int_{2\pi} f_r(x,\vec{\omega},\vec{\omega}')L(x,\vec{\omega}')(\vec{\omega}'.\vec{n})d\omega' \\
&\approx \frac{1}{\Delta A}\sum_{p=1}^{n} f_r(x,\vec{\omega},\vec{\omega_p})\Delta\Phi_P(x_p,\vec{\omega_p}) \\
&= \frac{1}{\pi R(x)^2}\frac{\tau(x,\vec{\omega})}{N_{emmited}}
\end{aligned}
\tag{3.19}
$$

Photorealistic effects

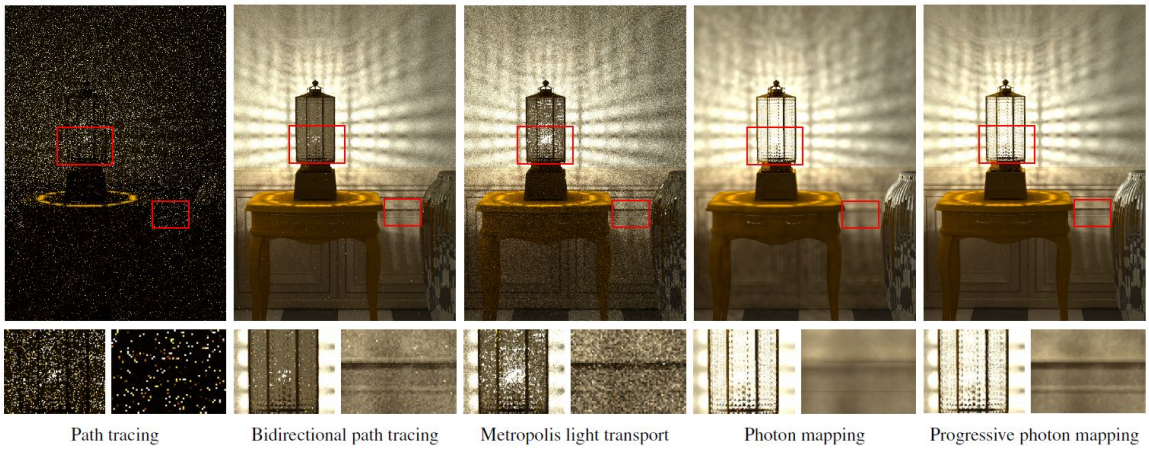


Figure 3.6: Comparision between most accurate global illumination methods[8]

By using progressive photon mapping it is possible to render all focused photorealistic effects. This method excels in SDS path like in image 3.6, where light is going through glass from lamp. All techniques have same rendering time. It can be seen that path tracing is completely out, bidirectional path tracing and MLT render more accurate image but with lot of noise. Photon mapping is in this particular case better than monte carlo ray tracing methods. Best photorealistic result shows progressive photon mapping. This method has succeeded to remove low frequency noise from image by reducing radius of nearest search.

Chapter 4

Data structures and algorithms

This chapter describes important techniques which was implemented in this thesis. First ray-triangle algorithm is described, then spatial subdivision is described. After that, section about brief introduction of evolutionary algorithm is written. In the end of this chapter, summary of important aspects of GPGPU computing are written.

4.1 Ray-triangle intersection

Lot of computer graphic algorithms depend on examination ray with scene. Nowadays, nearly every complex scene is created from thousands triangles, so for reasonable speed of final renderer, fast ray-triangle test has to be performed.

Every ray is defined by origin O and direction $\vec{D}$ of ray. Every triangle is defined by three vertices A, B, C. Ray-triangle intersection is possible to compute by combination parametric equation of line and barycentric coordinates of point x in triangle. Final relation is[9]:

$$O + t.\vec{D} = A + u.(B - A) + v.(C - A) \tag{4.1}$$

where t is parameter indicating intersections distance from origin of ray. u, v are barycentric coordinates. For determination if ray is in intersection with triangle, these conditions have to be fulfilled:

$$\begin{aligned} 0 &\leq t \leq t_{max} \\ u &\geq 0 \\ v &\geq 0 \\ (u + v) &\leq 1 \end{aligned} \tag{4.2}$$

Lot of useful methods was invented. One of this method is method invented by Jiri Havel in 2010[9] and according to informations written in this article, this method is fastest invented method. This method compute two perpendicular planes to triangle as in image 4.1.

This perpendicular planes are computed as:

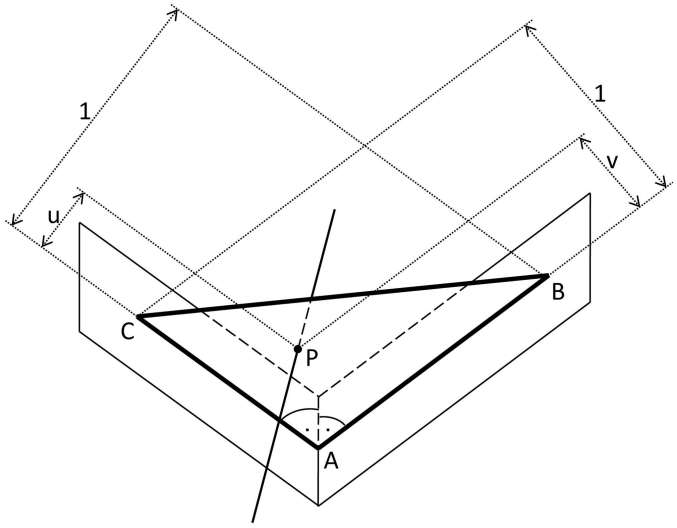


Figure 4.1: Perpendicular planes for barycentric coordinates calculation[9].

$$\begin{aligned} \vec{N_1} &= \frac{\vec{AC} \times \vec{N}}{\vec{N}^2}, \quad d_1 = -\vec{N_1}.A \\ \vec{N_2} &= \frac{\vec{N} \times \vec{AB}}{\vec{N}^2}, \quad d_2 = -\vec{N_2}.A \end{aligned} \tag{4.3}$$

After computing this perpendicular planes, this values have to be computed:

$$\begin{aligned} det &= \vec{D}.\vec{N} \\ t' &= d - (O.\vec{N}) \\ P' &= det.O + t'.\vec{D} \\ u' &= P'.\vec{N_1} + det.d1 \\ v' &= P'.\vec{N_2} + det.d2 \end{aligned} \tag{4.4}$$

and for test, if ray intersect given triangle, these condition have to be fulfilled:

$$\begin{aligned} sign(t') &= sign(det.t_{max} - t') \\ sign(u') &= sign(det - u') \\ sign(v') &= sign(det - u' - v') \end{aligned} \tag{4.5}$$

if this conditions have met, then, if need exact point of intersection is needed, barycentric coordinates u, v and parameter t are computed as:

$$\begin{aligned} t &= \frac{1}{det}.t' \\ u &= \frac{1}{det}.u' \\ v &= \frac{1}{det}.v' \end{aligned} \tag{4.6}$$

From this parameters it is possible to compute point in space by using one of these equation:

$$\begin{aligned} P &= O + t.\vec{D} \\ P &= A + u.(B - A) + v.(C - A) \end{aligned} \tag{4.7}$$

4.2 Spatial subdivision - KD TREE

If scene has N triangles, then for test, if ray intersect with scene, N intersections tests have to be computed. Fast ray-triangle intersection is not enough to perform these sort of test if N is big enough (hundred thousands, millions) in reasonable time. For scenes with lot of triangles, reducing intersection number has to be included. This could be done by using special technique called spatial subdivision (or use spatial index).

Spatial subdivision techniques divide space into subspaces. In ray-scene intersection tests, they compute intersection with these subspaces first and after that they compute intersection only with triangles which lie in subspaces intersected by current ray.

Lot of different techniques (indices) exists - regular grid, bsp, octree, kd-tree. Most used spatial index in computer graphics is kd-tree[21].

In kd-tree, space is divided into two subspaces by plane. This space and plane are often axis aligned. Position of dividing plane is key to performance of this spatial index.

It is possible to place dividing plane to make equal subspaces, randomly or to use sophisticated heuristic. One of best results (in performance) show Surface Area Heuristic [28].

Surface area heuristic computes few candidate planes, for every plane compute SAH cost function and then choose best plane with lowest cost function. SAH cost function is computed as:

$$SAH(x) = C_{ts} + \frac{C_L(x)A_L(x)}{A} + \frac{C_R(x)A_R(x)}{A} \tag{4.8}$$

where C_{ts} is cost of traversing another node. C_L respectively C_R is cost of traversing left respectively right child, it is possible to substitute this values as number of triangles in left respectively right child. A_L, A_R and A are areas of left, right or parent bounding box.

Traversing kd tree could be done in several ways. This ways differs by using (or not using) recursion or by using ropes. One way how to use traversal without recursion is to use ropes. Ropes are made in each node and bind edges of node with their neighbor subtree. Image 4.2 shows this ropes in simple kd-tree.

When ropes are used, it is possible to use simple iterative algorithm[21] for searching in this tree. This algorithm is using top-down approach. When it goes into child without intersected triangle, this algorithm find appropriate rope and using this rope find another subtree to examine. This process is repeatedly made until intersected triangle is found or algorithm find rope with NULL - end of space.

4.3 Genetic algorithm

There are several algorithms for searching solutions in solution space. This algorithms are for instance breath first search, hill climbing or random search. Another possible algorithm is searching algorithm based on evolution - genetic algorithms.

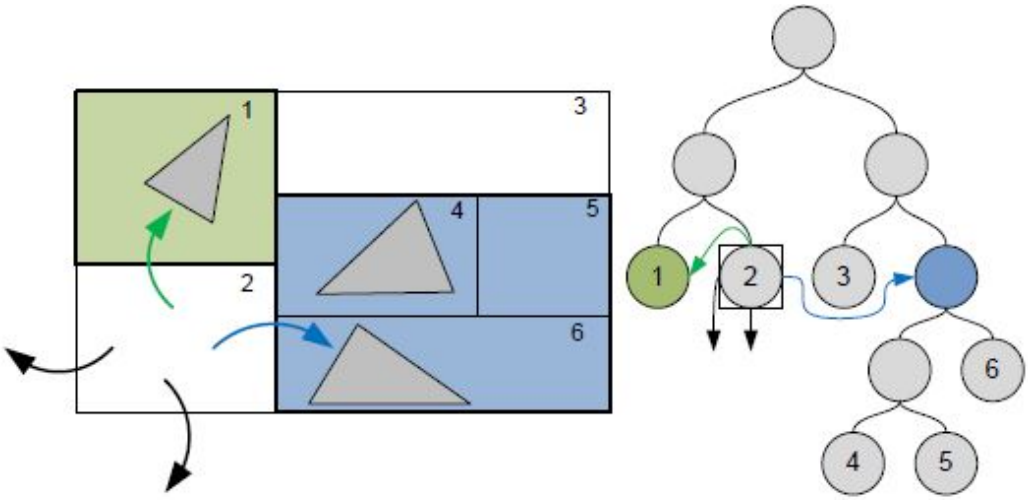


Figure 4.2: Kd-Tree with ropes[21]

Genetic algorithm try to mimic evolution in nature[3]. Simply put, it uses lot of randomly generated solutions, this solutions try to combine and mutate and try to find best solution by evaluation solution with best fitness function value.

For describing how genetic algorithm exactly works, few new terms have to be introduced. This terms are: chromosome, fitness function, selection, crossover and mutation.

Chromosome

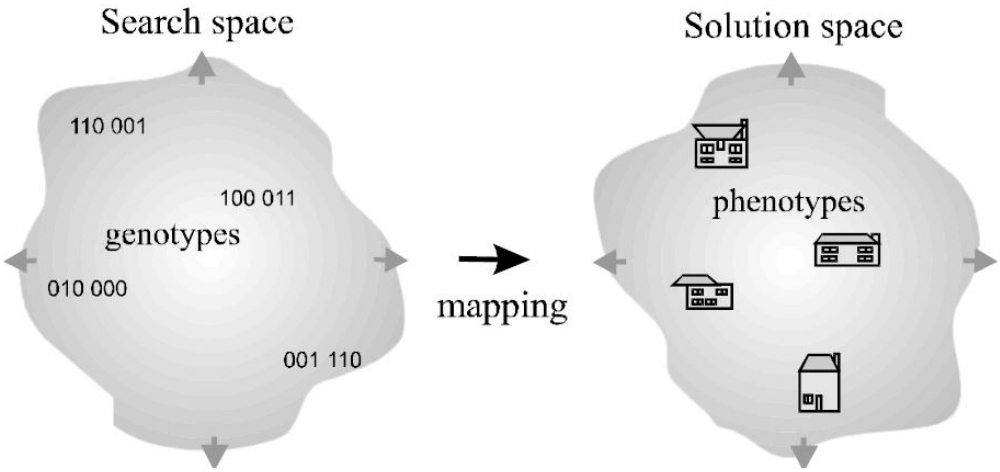


Figure 4.3: Genotypes and phenotypes in genetic algorithms[3]

Genetic algorithm works with phenotypes. Phenotype is one candidate solution in solution space. On image 4.3 is shown example where ideal house is searching. Phenotypes on this images are each possible house. Phenotypes consist of list of parameters (in house example it could be size and position of windows, door, roof etc). Genetic algorithm in computer works with genotypes. Genotypes are coded phenotypes into sequence of ones

and zeros and genetic algorithm on computer is working with this sequences. One coded parameter from phenotype is called gene. One genotype is often in computer held as string and is called chromosome[3].

Fitness function

Fitness function is used for evaluation how solution is good. Input to this function is one chromosome, output is a number. Goal of genetic algorithm vary by usage of this function. One genetic algorithm tries to maximize this function, other tries to minimize this function. Perfect fitness function is probably most crucial part of genetic algorithm.

Selection

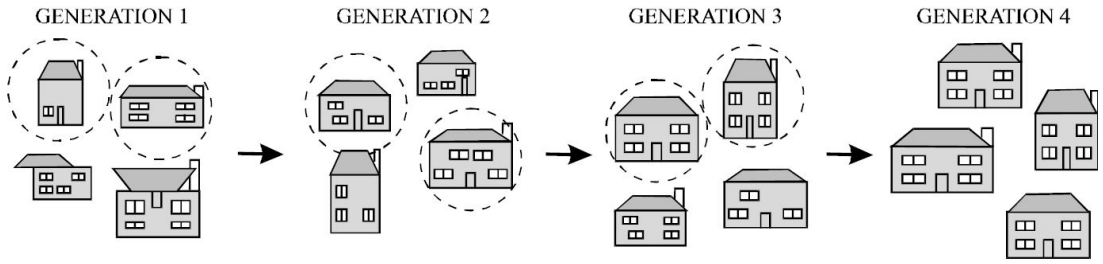


Figure 4.4: Four iterations of genetic algorithm with selection best phenotype [3]

Because genetic algorithm is iterative algorithm, new population is created by mutation and combining two chromosomes. For choosing which chromosome to pick (for mutation/crossover), selection algorithm is used. This algorithm could choose best solutions, random solution or use tournament - random choose fraction of chromosomes from population and choose best from this random fraction. In image 4.4 is shown selection of best chromosome.

Crossover

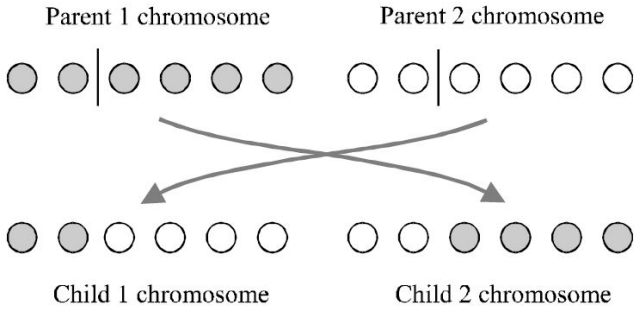


Figure 4.5: One point crossover operator [3]

When new population is created, this population is often created by using crossover operator. Input to crossover operator are two chromosomes - parents and output to crossover operator are two chromosomes - children. Crossover operator combines genes from both

parents and is generating two new children. In literature this crossover operator is often demonstrated on one point crossover. This one point crossover is shown in picture 4.5 and it choose random genome, split both parents in this genome position and two children combine by using different split genomes.

Mutation

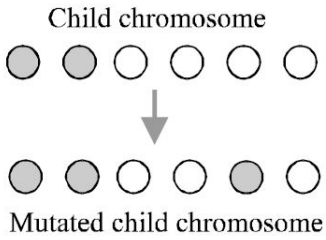


Figure 4.6: Simple mutation operator [3]

Mutation is randomly chosing one gene and this gene is mutated - change to random value. Input to this operation is one chromosome, output is one chromosome. It is often used to mutate generated children. On image 4.6 is shown simple mutation which tried change one random gene to different value.

Algorithm

Genetic algorithm is iterative algorithm. Genetic algorithm first initialize random population. Each solution then evaluate by fitness function. In each iteration, new population is generated by combining: selection, crossover operator and mutation. After each iteration each new chromosome is evaluated by fitness function. This process is repeating until satisfying solution is find or until number of iteration exceed some limit.

Evolution with mutation

If problem does not allow to use crossover operator, it is often used special evolution called $1 + \lambda$ evolution. This evolution means, that in every iterations, population is created by mutation one parent (best solution from previous iteration). Mutation is used for creating λ children and in new population are one parent and λ children.

4.4 Introduction to GPGPU computing

Graphics processor unit (GPU) can be used nowadays not only as co-processor for fast graphics computation, but it can also be used for general purpose task. It is possible to compute nearly every sort of algorithm on nowadays GPU.

GPU has specific architecture and for describing programing task it is using special languages. Most used languages (frameworks) are CUDA and OpenCL. Because in this thesis, renderer was made using OpenCL, OpenCL is here shortly described.

OpenCL - Open Computing Language - is multi-platform programing language, programs written in opencl has potential to run on CPU, GPU, DSP, FPGA. This multi-platformity is achieved by special memory model and execution model. OpenCL does not

guarantee that all programs will run fast on each platform ie. some nuances used on programming GPU will be ineffective on CPU and vice versa.

Execution model

Every OpenCL program consists of host side and device side. Device is in opencl special harwdare on which OpenCL program will run (GPU, CPU). Host is on CPU side and control how opencl will perform tasks. On host side, program allocate memory for computation, set size of workgroups and define in which order tasks will be run.

Minimal programmable (and only one) unit in OpenCL is called kernel. One kernel equal one thread in parallel computing. Kernels are grouped into groups called workgroup.

Memory model

OpenCL has more memory spaces. Each vary by size and by access time. This memory spaces are: global memory, local memory and private memory. Global memory is biggest memory in opencl memory model. This memory is shared between all kernels and workgroup. This memory has biggest access time. On nowadays GPU it has size of gigabytes.

Private memory has smallest memory, but it has fastest access time. This memory is private for each kernel and this memory is mapped on registers, so size is in kilobytes. Local memory is shared between all kernels in one workgroup and each kernel has possibility to read or write from this memory.

Chapter 5

Analysis and Plan of work

This chapter focuses on analysis of global illumination methods and after that analysis, one method for implementation is chosen. After that, plan of work is described.

5.1 Selection of technique

Lot of techniques was described in first part of this thesis. This thesis was in beginning, when I was thinking about some master thesis, about photorealistic rendering so reasons why I chose photon mapping is described in this section.

Object order methods are fast and even using this methods it is possible to achieve some kind of photorealism. I don't think that it is good idea to create photorealistic renderer with object order methods, because raytracing methods are much more elegant in achieving photorealistic elements and every photorealistic phenomena is approximation from real world.

By using raytracing it is possible to achieve better photorealistic results than with object order methods, but in classic raytracing photorealism is still poor. Distributed raytracing is in my opinion too costly compared with photon mapping or monte carlo raytracing techniques.

Radiosity is nice technique for computation physically correct indirect diffuse illumination, but radiosity has limitations - closed scene, scene only with diffuse materials and because of this, it is unusable in real world scenes.

Monte carlo raytracing methods are probably most precise rendering methods, because they are consistent and unbiased. Error of this methods has high frequency - it is very disturbing for human eye and for removing this error, lot of iteration is needed and with huge number of iteration come huge render time.

Photon mapping methods are consistent, but they are biased. With small photon map, lot of low frequency noise is in the rendered scene. This methods are much faster than monte carlo methods, they can render better images in smaller computation time. In my opinion this class of rendering methods has best potential to be real time global illumination methods in near future and because of this my thesis will focus on photon mapping methods.

5.2 Work plan

Because my work started nearly two years before finishing this thesis, I didn't have any complex-two-years plan. I chose to implement photon mapping technique and decide what

next step will be done after it. Normal photon mapping was extended by progressive variant and progressive variant of photon mapping was done on GPU. All this implementations was implemented with focus on speed not photorealism.

Chapter 6

Simple photon mapping implementation

This chapter describes implementation of simple photon mapping renderer. First decomposition is described and then for each block short info will be written. Next, information about tests performed on normal photon mapping is written.

6.1 Decomposition

Photon mapping renderer is possible to divide to three section: scene preprocessing, first pass - creating photon map and second pass - rendering. It is possible to make this block diagram:

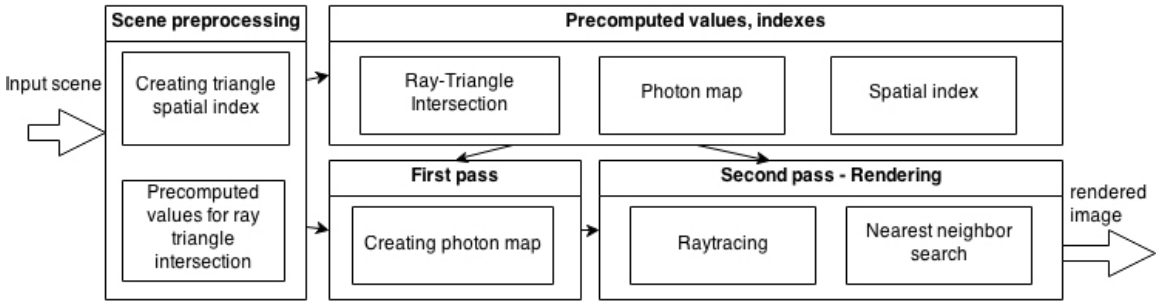


Figure 6.1: Block diagram of simple photon mapping renderer

Scene is loaded to rendering engine. On this scene, spatial index is created and for each triangle importnant data are computed. In first pass, photon map is created by using ray-triangle intersection and fast spatial index. In second pass raytracing is performed on scene. When local illumination is computed in raytracing, nearest photons are searched in photon map and illumination is determined from this nearest photons.

6.2 Scene preprocessing

In this part of renderer, scene is loaded from scene file. From scene graph, one array of triangles is computed. On this array of triangles spatial index is created and for each triangle from this array important data are computed.

Scene loading and scene preprocessing

Scene is loaded from COLLADA (.dae) file using ASSIMP library. ASSSIMP uses own internal scene interpretation, thus scene is loaded to own scene structure and classes.

Scene graph is loaded from scene, transformation matrix is computed for each scene graph node and from this values one array of triangles in space is computed.

Spatial index

Then on array of triangles, spatial index is created. Spatial index is kd-tree with ropes and dividing plane is computed from Surface Area Heuristic.

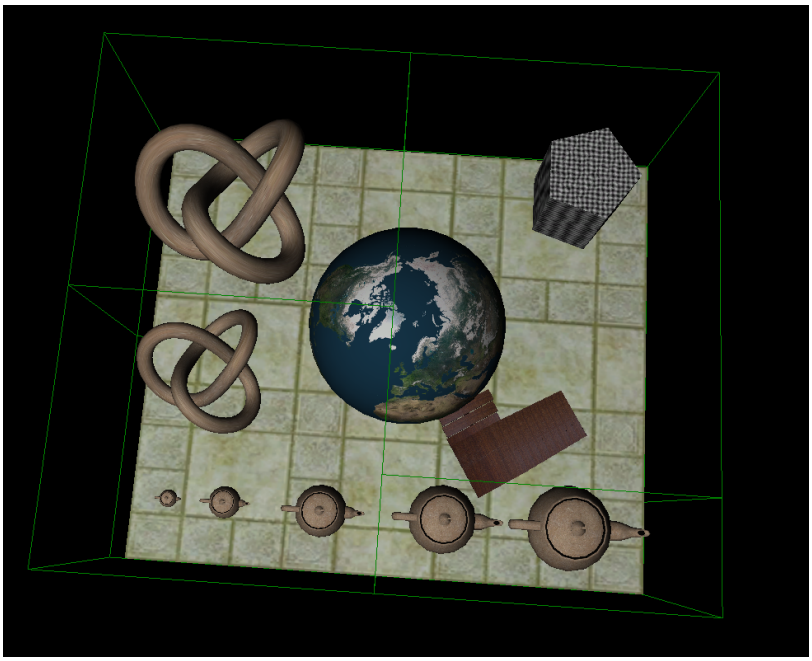


Figure 6.2: Spatial index with SAH

On image 6.2 is shown two levels of KD-tree with SAH. On this image is possible to se how surface area heuristic divide space with regard to number of triangles and surface of subspace.

Ropes is created from top to down with kd-tree creation. First, all ropes are set to NULL, then when subspace is created, child nodes inherit parents nodes and left child has proper rope to right child and vice versa. Using this approach ropes are distributed through kd-tree and each ropes point to subspace of scene. Test how kd-tree with ropes accelerated speed of raytracing is written in experiment section.

Precomputed values

For determining if ray intersect triangle, Havels method was chosen and implemented. This method has possibility to precompute some data (both perpendicular planes) and because of this, this data are precomputed in preprocessing.

6.3 First pass - Photon map creation

In this pass, photon map is created. For each light, photons are sent into scene with proper fraction of light energy.

Direction of sent photon is determined by so called rejection sampling. If sent photon hit surface, photon is saved into photon map (for now only in array of photons) and another photon is sent in proper direction. Direction of reflected photon vary by type of material. For completely diffuse material, direction of reflected photon is randomly in hemisphere around normal, for mirror like surface, direction is in normal reflected direction.

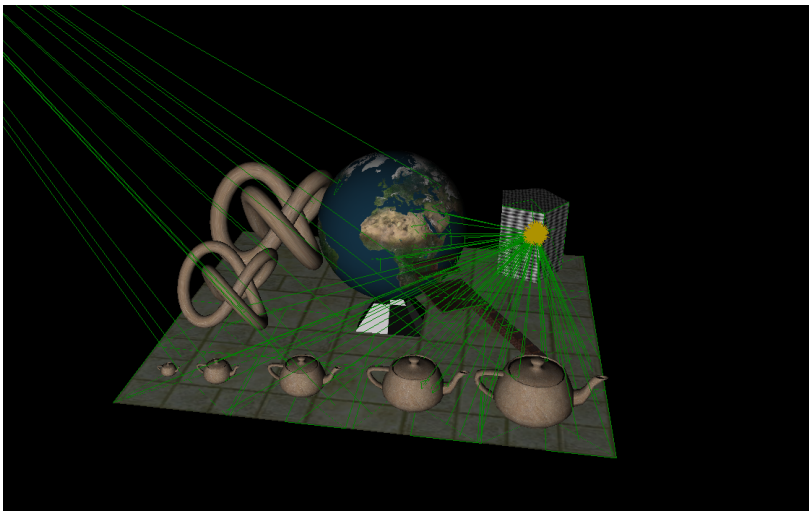


Figure 6.3: Photon map creation with low number of photons

If transparent specular surfaces are in scene, two maps are created. One for indirect illumination (normal photon map) and second only for caustics (caustic photon map). In caustic photon map, only those photons wich go through specular materials are saved.

Using this approach, millions of photons are created. Searching through this huge number of photons is very slow. Because of this, I used kd-tree over photons for fast nearest search.

6.4 Second Pass - Rendering

Final image is gathered in this pass. Rays are generated from camera information. For each pixel in final image, direction of ray is determined. Then this ray is sent into scene, when ray hit surface, it normally compute direct illumination through shadow rays and reflected/refracted illumination and then it add to local illumination model, illumination computed from photon maps.

N nearest photons are searched through normal photon map in each ray/scene intersection. For caustic illumination radius search is performed in fixed radius.

After implementing described blocks simple photon mapper was created. It is capable to render indirect illumination and caustics. Image 6.4 shows output of this process. Rendering this images consume lot of time and because of this, in next section slow block will be measured and for slowest blocks improvements will be proposed.

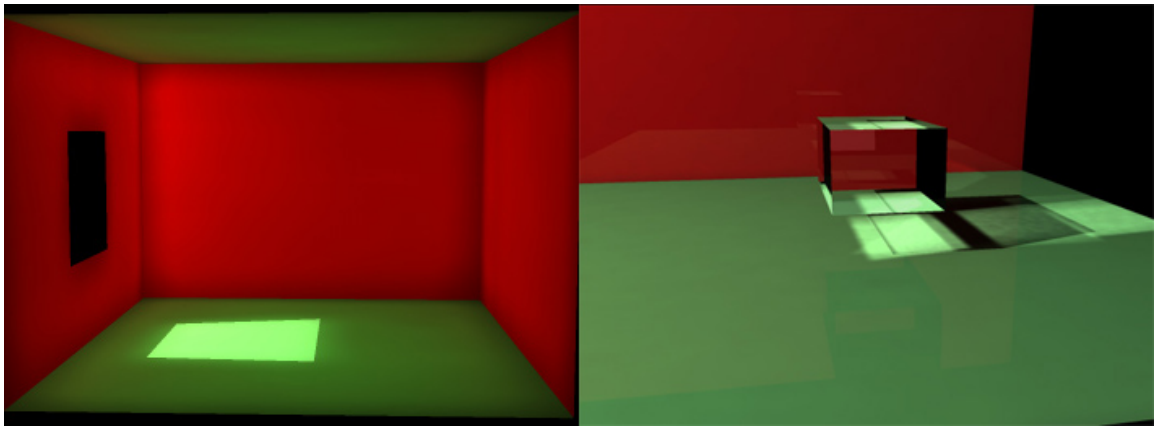


Figure 6.4: Scene with indirect illumination is rendered on left image, on right image caustics are rendered

6.5 Experiments - the best spatial index

Previous section introduce simple implementation of photon mapping renderer. Lot of blocks have possibility to implement them in different ways, this blocks are ray-triangle intersection, nearest neighbor search, spatial subdivision.

Nearest neighbor block has biggest influence on speed of photon mapping, so several tests was performed on this block. Speed of fast raytracing depend on good spatial index and because of this, performance evaluation between few possibilities was made. Ray-triangle intersection block was ommited from this tests, because author write in corresponding article that his algorithm is fastest and even then nearest neighbor and spatial index have bigger influence on resulting speed.

One possible way how to perform nearest neighbor is implement own function for this search, other is to use special library. After implementing own function this function was evaluated along with FLANN library and flann library was nearly four times faster. Because of this, few tests on two nearest neighbor libraries was made - for choosing fastest library and fastest index.

First library is FLANN - Fast Library for Approximate Nearest Neighbors. Second library is ANN Approximate Nearest Neighbor Library. ANN is older library, this library use for searching kd-trees and bd-trees. FLANN library use for indexing randomized KD-tree. Both libraries are very often used in computer vision for nearest neighbor searching in image feater and they are very well optimized for multidimensional datasets. Both libraries provide approximate searching - search with a small acceptable error.

Performed experiments were these:

- **Spatial Subdivision test** - comparing spatial indices, octree and KD-tree for acceleration ray-triangle intersection
- **Creating photon map** - compare speed of creating searching indices on KD-tree and BD-tree in FLANN and ANN libraries
- **Nearest searching - neighbor number** - Comparing nearest neighbor search time with increasing number of neighbors to find

- **Nearest searching - size of map** - Comparing nearest neighbor searching time with increasing size of photon map
- **Approximate nearest search** - Identification of maximmum acceptable error and comparison of the approximate searching times

All the experiments were performed on laptop with Intel core i7 M620 processor @ 2.67GHz with 2x 2GB DDR3 RAM 1066MHz, 7-7-7-20. Form compiling, the MSVC11 (Visual studio 2012) compiler was used. All the measurements were performed on real photon mapping data.

Spatial subdivision test

This test compares speed of spatial indexing methods. In this test, three methods are compared. First, the naive method with no indexing method - simply bruteforce method - was measured. Then this method is compared to octree and KD-tree with ropes.

To compare these methods, simple scene with 12140 triangles was created (image 6.2). On this scene 100 000 ray-triangle intersections were performed. All intersection tests was pointing on same place in scene.

Name	Speed	Precomputing
Naive	47.567s	-
Octree	1.159s	**0.04s**
KD-tree with ropes	**0.453s**	0.16s

Table 6.1: Comparing spatial indices

Table 6.5 shows that KD-tree with ropes is approximately two times faster than octree. This test also shows that using spatial indices is indeed efficient and any of the methods outperforms the naive approach. The KD-tree was 88 times and octree was 44 times faster then naive method.

These test also shows speed of creation of spatial index. Octree is approximately four times faster than KD-tree. As the spatial index is created only once in this method, while the ray-triangle intersections are performed many times - in photon mapping and raytracing. The KD-tree with ropes is generally the best of the tested ones.

One note to this tests: comparison of speed between this indices may vary on other scenes, some scene could have better speed with octree. Probably it is best to try which index is best on each scene.

Creating photon map

This test compares times of creating spatial index in photon mapping depending on the total number of photons in photon map. As it was written, for this test ANN library an FLANN library were used. From ANN library KD-tree and BD-tree was used and from FLANN library KD-tree and single index KD-tree was used. Single index KD-tree is optimized for lower dimensional data.

For this test scene on image 6.4 (left side) was used and photon maps with 50k, 100k, 200k, and 500k photons was generated. To generate this amount of photons, block from photon mapping renderer was used, so this test was performed on the real photon mapping data.

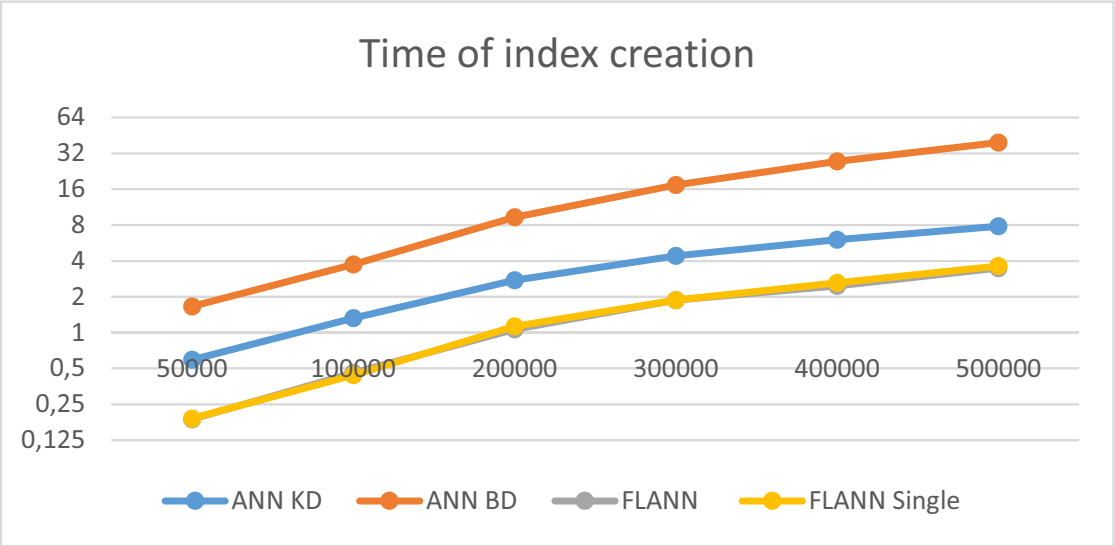


Figure 6.5: Dependece of map creation time on photon count.

Size of Map	50 000	100 000	200 000	300 000	400 000	500 000
ANN KD	0,591234	1,327776	2,751757	4,408452	6,009744	7,812247
ANN BD	1,660428	3,72888	9,296865	17,34999	27,49657	39,44626
FLANN	**0,187711**	0,458326	**1,066961**	1,878107	**2,476542**	**3,484599**
FLANN Single	0,190411	**0,439525**	1,131965	**1,873107**	2,61855	3,626007

Table 6.2: Average times of creating photon map with increasing photons count.

Results show image 6.5 and table 6.2. This results show that BD-trees have the worst time of index creation. Both of the FLANN indices KD-tree and single index have approximately the same time of index creation in fact, KD-tree is little faster than single KD-tree. The ANN KD-tree is approximately five times faster than BD-tree, but two times slower than both of FLANN indices.

Nearest search

This test was performed in order to compare speed of indexing methods depending on the number of neighbors to search for. The indexing methods for this test were the same as in the previous test - ANN KD-tree + BD-tree as well as FLANN randomized KD-tree + single index KD-tree.

The same scene as in previous test was used. The photon map with 500 000 photons was created and on this same map, the search indices were created. During the testing, the progressively increasing number of nearest neighbor to find were used.

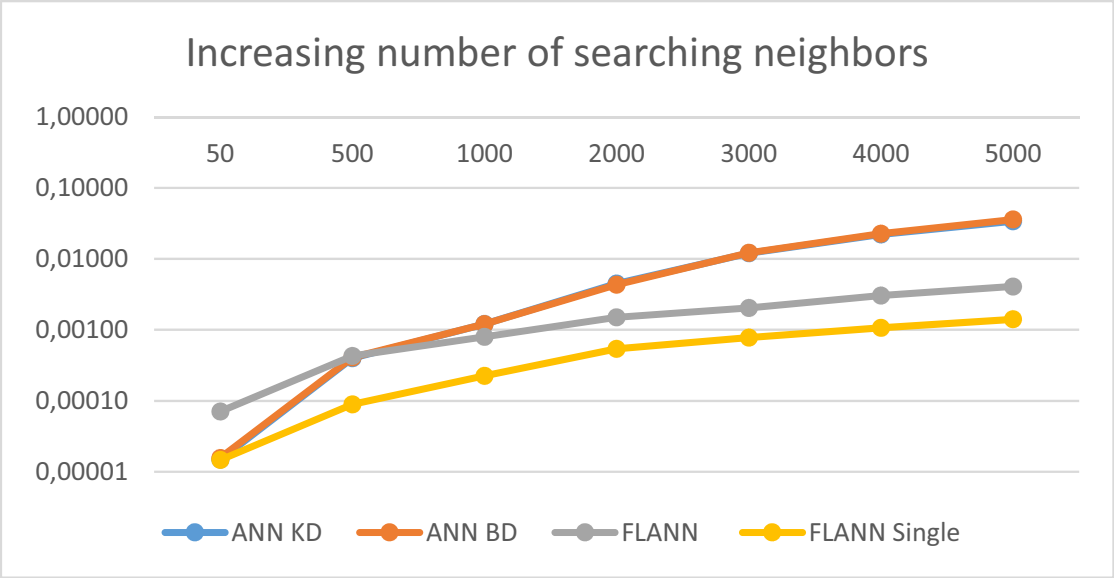


Figure 6.6: Dependece of searching time on photon count.

N	50	500	1000	2000	3000	4000	5000
ANN KD	**0,00001**	0,00040	0,00121	0,00452	0,01205	0,02204	0,03394
ANN BD	0,00002	0,00041	0,00121	0,00432	0,01218	0,02281	0,03590
FLANN	0,00007	0,00043	0,00080	0,00151	0,00203	0,00304	0,00410
FLANN Single	**0,00001**	**0,00009**	**0,00022**	**0,00054**	**0,00078**	**0,00107**	**0,00141**

Table 6.3: Average times for searching N photons with photon map of size 500 000 photons.

Table 6.3 and figure 6.6 show results. The results show that both ANNs indices and FLANN single index have the best performance for cases in which little number of photons is reaquiered to be searched for. However, with the increasing number of photons to search for, ANN is increasingly worse and FLANN KD-tree becomes better than the other two indices. FLANN single index tree has best results in this test.

Size of photon map

This test is similar to the previous one but in this case, the number of neighbors is fixed and the size of the photon map is changing. In this experiment, the number of neighbors was set to 5 000, same as in photon mapping. The size of the photon maps ranges from 50 000 to 500 000.

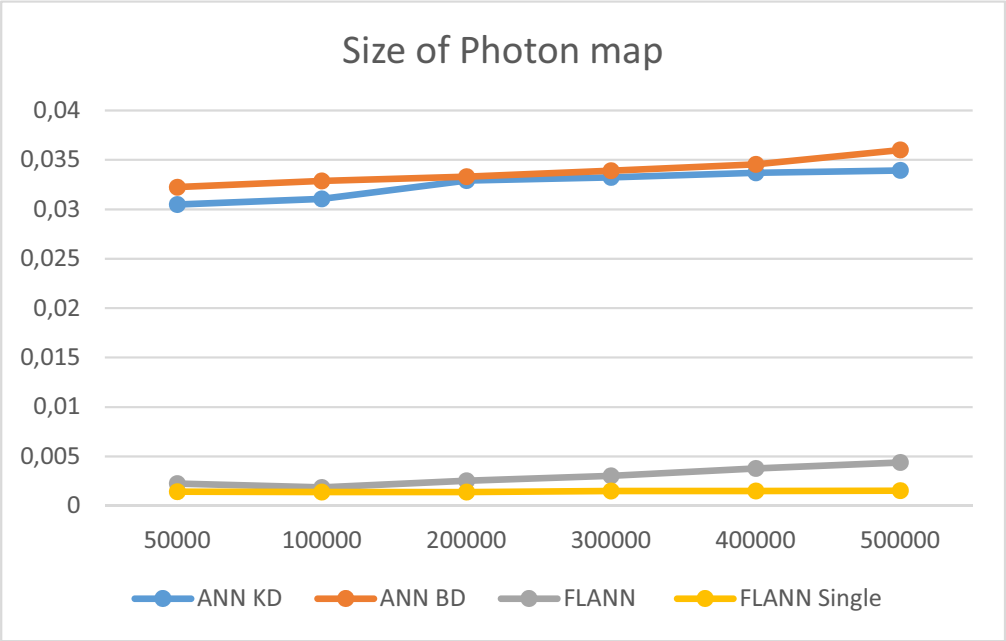


Figure 6.7: Dependece of searching time on map size.

Size of map	50000	100000	200000	300000	400000	500000
ANN KD	0,030481	0,03107	0,032918	0,033227	0,033702	0,03393
ANN BD	0,032248	0,032861	0,033306	0,033909	0,034564	0,035993
FLANN	0,002246	0,001861	0,00254	0,003049	0,003783	0,004382
FLANN Single	**0,001426**	**0,001393**	**0,001396**	**0,001489**	**0,001509**	**0,001528**

Table 6.4: Average times for searching 5 000 photons depending on size of photon map.

The results (figure 6.7 and table 6.4) shows that increasing size of photon map does not have too big influence on the achieved speed and that the size of the number of photons to search for has biggest impact on speed.

Approximate nearest search

The FLANN and ANN libraries provide an approximate searching method - search in which some error is allowed in the result and which is somewhat faster than the exact case. It should be interesting to find out how much influence the approximate searching has on the quality of rendered images and how much acceleration can be achieved. As the approximate searching leads into worse results in terms of quality, it should be found out how much error is acceptable and then how much it influences the speed.

For this test same scene as in previous test was used and only indirect illumination was rendered. This indirect illumination was achieved by search for the nearest photons in the photon map. Photon map size was 500 000 photons and 5 000 photons were searched.

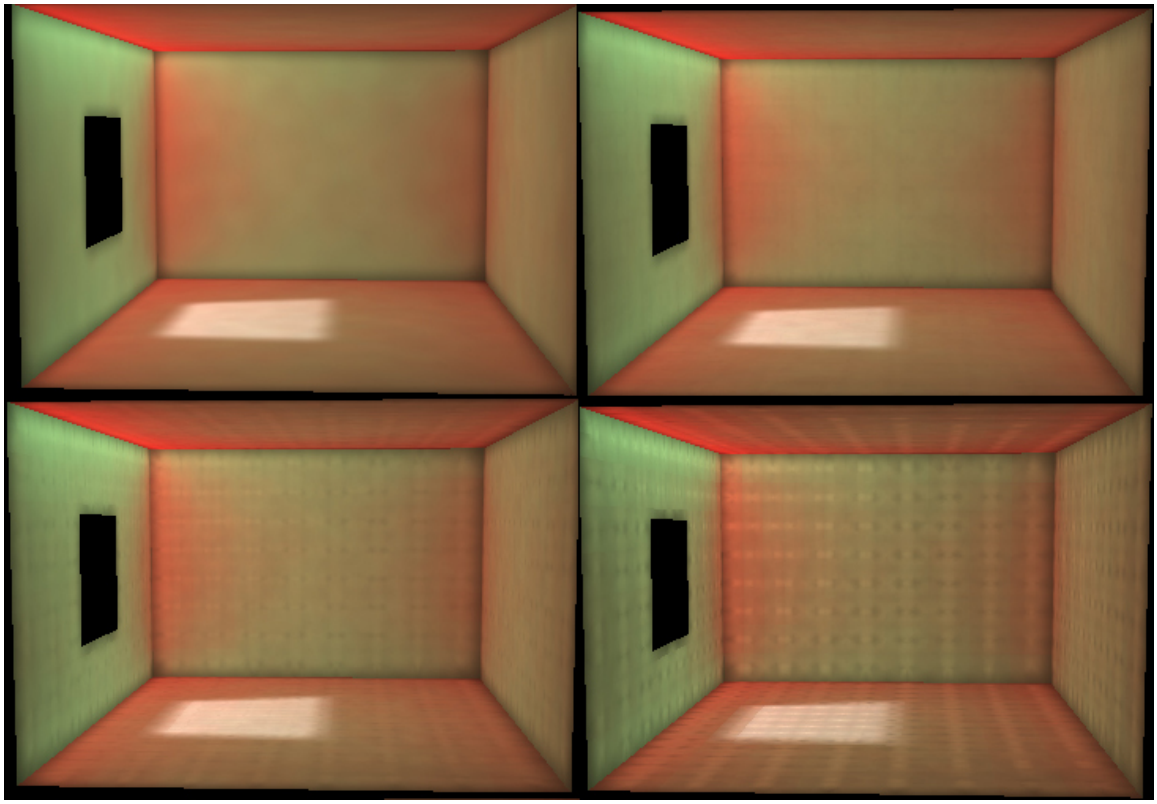


Figure 6.8: On top left image is epsilon equal zero. Top right epsilon equal to 1. Bottom left epsilon equal 5. Bottom right epsilon equal 10.

In first part of this experiment, reasonable size of error - epsilon - has to be find out. Result of increasing epsilon shows image 6.8. In this image it is possible to see that with increasing epsilon, quality of indirect illumination decrease and regular pattern are visible with bigger epsilon. Reasonable epsilon is in range between zero and one. Everything bigger than this result in worse quality.

Image 6.9 and table 6.5 shows results of this test as and it can be seen results aren't good. With epsilon equal 1 - maximum acceptable error for quality of rendered image - speed of fastest index is accelerated only by 12%.

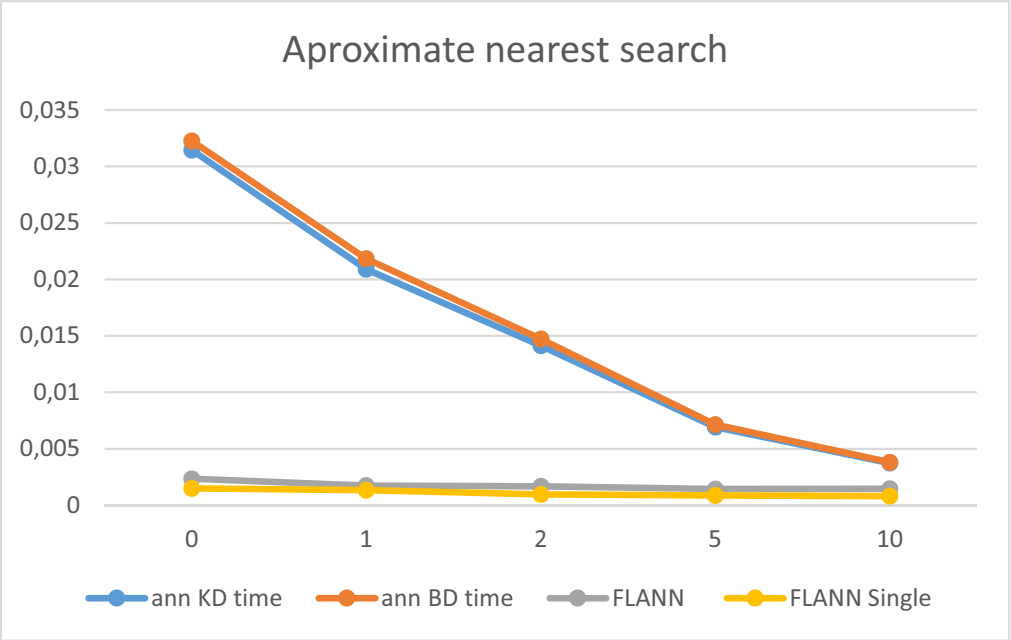


Figure 6.9: Dependece of searching time on acceptable error - epsilon.

Epsilon	0	1	2	5	10
ann KD time	0,031451	0,020913	0,01414	0,006924	0,003739
ann BD time	0,032258	0,021839	0,014725	0,007142	0,003817
FLANN	0,002356	0,00175	0,001694	0,00145	0,001465
FLANN Single	**0,001499**	**0,001328**	**0,000981**	**0,000881**	**0,000817**

Table 6.5: Average times for searching 5 000 photons depending on acceptable error - epsilon

6.6 Summary

This chapter described implementation of simple photon mapper. In implementation, decomposition was introduced and for each decomposed block implementation was proposed. After naive implementation few experiments on implementation of photon mapping was performed. This experiments show that probably best index for accelerating nearest search is Single index KD-tree from flann library. One test focused on computation with acceptable error - epsilon. This test showed that acceleration lead to worse result with only small speed-up.

One note to the end of photon mapping section: speed of searching 5 000 nearest points in photon map (which consists only from array of points in 3D space) is 1 millisecond, this time is only for searching indices of nearest photons in photon map and summing up all searched photons takes 5 milliseconds, so finding even more faster nearest neighbor search is useless and size of photon map has to decrease. This drawback solves progressive photon mapping and about this technique next chapter will be.

Chapter 7

Progressive photon mapping implementation

This chapter will describe implementation of progressive photon mapping. This implementation is first done on CPU and than on GPU. GPU implementation is unoptimalized and for slowest block, few acceleration techniques is proposed.

7.1 CPU implementation

Progressive photon mapping is extension of normal photon mapping. Normal photon mapping was created in previous chapter and thanks to that, base for progressive photon mapping should be used from normal photon mapping implementation.

The biggest changes are that raytracing is performed before photon tracing and in other passes photon tracing with hitpoint illumination and image synthesize are performed periodically.

Raytracing

Raytracing from normal photon mapping is extended. Extension is about saving special points where ray hits diffuse surfaces. This special points are called hitpoints. In each hitpoint few values are saved, this values are: position, normal, pixel location on final image, current photon radius, accumulated photon number, accumulated color, weight of hitpoint. Raytracing block returns one array of hitpoints.

Photon tracing

One change in photon tracing block is pre allocated array for photons. Beside that, there are no changes in this block. Photon tracing is evaluated in each photon tracing iteration.

Hitpoint illumination

Illumination from photon tracing is saved in this block. For each hitpoint x photons in photon map are searched and those photons which lies in radius of $R(x)$ are added to computation. Few another computation are performed after summing up illumination from photon map. This computation are described in chapter about photon mapping 3.5.

Finding photons in radius $R(x)$ are slowest operation from this block. This searching are not performed naively, but for this purpose FLANN single index kd-tree is used so before each hitpoint illumination computation, kd-tree is build upon current photon map.

Image synthesize

In this block, image from hitpoints are created. For each hitpoint, color is putted in framebuffer in proper position and with proper weight. This weight is changing depending if corresponding ray was primary or if this ray was reflected/refracted and thus his weight is corresponding to material weight of traversed materials.

Result and conclusion

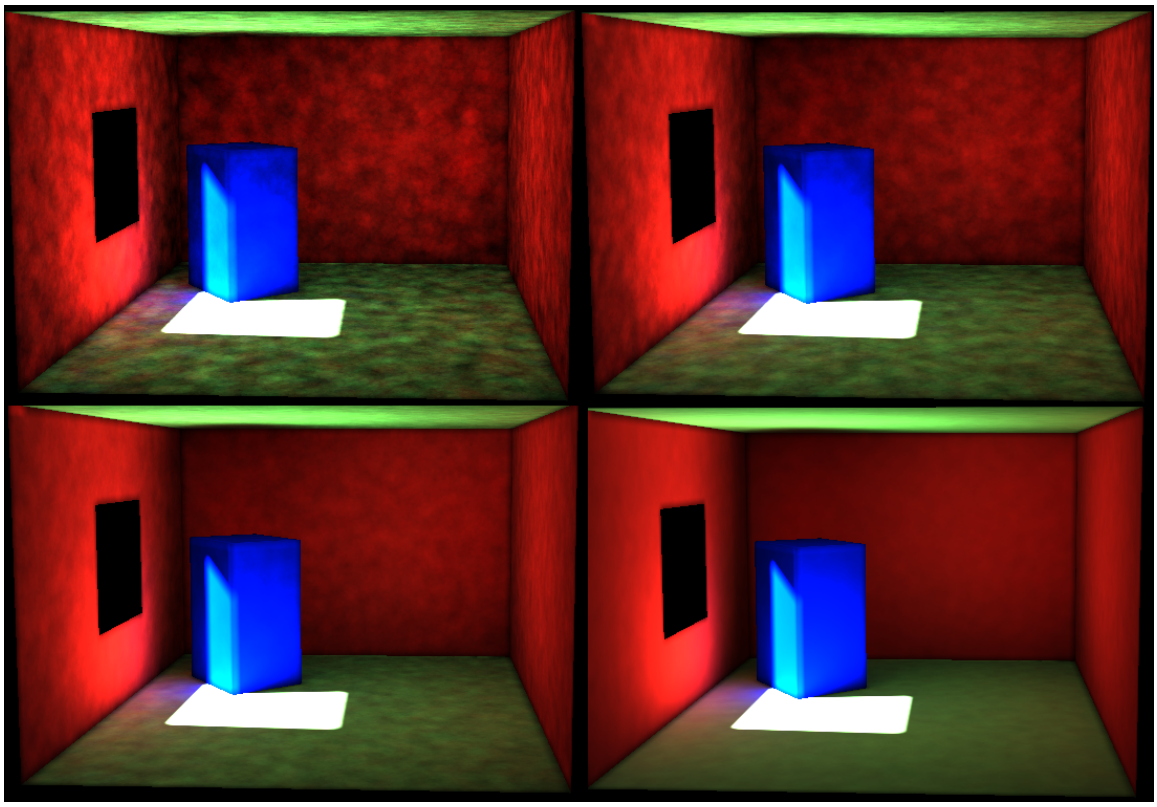


Figure 7.1: Progressive photon mapping result. Top left image with 1 iteration, top right image with 2 iterations, bottom left image with 10 iterations, bottom right image with 100 iterations.

After all changes was performed, functional progressive photon mapping renderer was created. Image 7.1 shows result of this renderer with progressive iterations. In this scene 100 thousands photons per iterations was used, initial radius of all hitpoints was set to 5. CPU implementation was created only for experimental purposes and for evaluation GPU vs CPU speed on this task.

7.2 Naive GPU implementation

This section describes naive implementation of progressive photon mapping on Graphics Processing Unit.

Raytracing

Each ray from final image is traversed in one thread on GPU. On GPU it is not possible to make recursion, so raytracing has to be rewritten to iterative algorithm. Because simple scene is used (from image 7.1) and this scene has few triangles, this triangles could be saved in constant memory and no spatial index is used. For ray-triangle intersection Havels algorithm is used. After finding intersections with scene, hitpoints is saved into global memory.

Photon tracing

Photon tracing uses very similar routines for scene traversal like raytracing. Each initial photon from light is traversed in separate kernel. Same as in raytracing, it is not possible to make this block recursively and this block is made iteratively with fixed number of iteration - in this simple scene this is not a problem.

One problem occurs when photon has to determine random direction - in photon generating or in photon reflection on diffuse surface. OpenCL and GPGPU does not have any sort of random generator and therefore random generator is needed. It is possible to use classical congruential generator with saving seeds between iterations.

Hitpoint illumination

In this block, illumination is computed in parallel for each hitpoint. First, photons in radius $R(x)$ has to be find out. Most naively solution is going through all photons in photon map, for each photon check distance and sum up those who has distance lower than $R(x)$. This is very naive solution and will be accelerated further in this text. After finding all photons in proper radius, few computation has to be made. This computation is in chapter about progressive photon mapping.

Image synthesize

All needed data for image synthesize are saved along with each hitpoint and therefore computation of luminance is very easy. For each hitpoint one kernel is executed and this kernel compute color for his hitpoint. This color is scaled by hitpoints weight and is atomically added to framebuffer to proper position given by hitpoints framebuffer position. This block is possible to perform after each photon tracing and hitpoint illumination pass or at the end of all iterations.

Evaluation of naive implementation

Implementation of CPU and GPU renderer was evaluated. CPU implementation was evaluated on laptop with Intel i7-4702MQ processor and it was written with c++ and it was compiled with Intel C++ 15.0 compiler. GPGPU implementation was written in OpenCL

and was evaluated on nVidia GeForce GT 750M. Speed of CPU implementation was evaluated using std::chrono library and GPU implementation was evaluated using nvidia nsight timeline profiler. Evaluation was performed on same scene as in image 7.1 .

	GPU	CPU
Raytracing	0.706 ms	1172 ms
Photon tracing	7.897 ms	317 ms
Hitpoint Illumination	2041 ms	1593 ms
Synthesize	0.247 ms	3 ms

Table 7.1: Performance evaluation between CPU and GPU naive implementation with 320*280 resolution and 100 thousands photons in photon tracing pass

	GPU	CPU
Raytracing	12.107 ms	6 197 ms
Photon tracing	7.897 ms	317 ms
Hitpoint Illumination	38 603 ms	23 957 ms
Synthesize	4.924 ms	91 ms

Table 7.2: Performance evaluation between CPU and GPU naive implementation with 1920*1080 resolution and 100 thousands photons in photon tracing pass

Table 7.1 and table 7.2 show performance between GPU and CPU implementation for 320*280 and 1920*1080 resolution. In each photon tracing iteration 100 thousands photons was traced. As it can be seen, GPU is far more faster than CPU implementation in all blocks except hitpoint illumination. CPU block of hitpoint illumination is using kd-tree and if this block will use naive search similar to GPU it will be much more slower. For resolution 320*280 naive solution on CPU was performed in 68 seconds.

Speed of raytracing and photon tracing is influenced by number of pixels / photons and by complexity of scene. Even when testing scene was simple, photon tracing and raytracing made good basis for hitpoint illumination block which will be same in all types of scenes - there will be big array of hitpoints with big array of photons.

Because hitpoint illumination is slowest block in this process, and because for raytracing is lot of research plus nvidia optix library, further in this tesis will be focus on accelerating hitpoint illumination on GPU.

7.3 Acceleration techniques

Bruteforce implementation of hitpoint illumination is very naive and very slow. In this simple implementation, each thread in workgroup load one photon from memory, compute distance, check if this distance is lower than proper radius and optionally accumulate photon flux.

Memory loads are very costly on GPU, even if streaming multiprocessor is multiplexing work between few workgroups, it will wait long time in sleep mode to load data from global memory. One way how to mitigate memory waiting is to use local memory.

This section will persent accelerating approaches to accelerate hitpoint illumination block. This approaches will try to use lot of local memory and will reduce number of

searched photons. After proposing all acceleration approaches, evaluation will be made for all approaches.

Local memory

Idea is to load block of photons into local memory and go through this photons from local memory. Access to local memory is faster than access to global memory, so it should lead to acceleration. Each thread in workgroup will load one photon into local memory. Kernel driver will recognize this as block loading and this block loading should be done faster then sequential load of each photon.

Photon sorting

When illumination is computed for hitpoint x, only those photons which is lying on same mesh as hitpoint x have to be included in computation. When illumination is computed by bruteforce, photons from all meshes are loaded from global memory even it is unnecessary.

One way how to solve this problem with unnecessary loaded photons is to sort photons into unique photon map for each mesh. This task should be done in separate kernel. Each thread will load one photon, check its mesh, by special operation called atomic increment will get position in separated photon map an this photon will save into proper position in memory. Speed of this kernel is 1 ms so it is negligible in comparison with acceleration achieved by this sorting.

When hitpoint illumination is computed and sorted photon maps is filled, it is possible to load only those photons which is lying on correct mesh. It is not possible to use local memory in this approach because it is not guaranteed that all hitpoints in one workgroup will lie on same mesh.

Hitpoint sorting

It is needed to ensure that all hitpoints in workgroup lie on same mesh to use coherent approach (all threads are reading from same memory location) for fast loading from memory.

Hitpoint sorting could be done on CPU side, because this work will done only once per whole render process. Hitpoints are loaded to CPU side after raytracing pass. This hitpoints are sorted to separate arrays and then this arrays are merged together. In the end of each hitpoint array, special filler hitpoints have to be saved. This is because ending workgroup of one hitpoint array is not filled fully with hitpoints from one mesh, for instance if size of workgroup is 256 and in last workgroup of hitpoint array is only 150 hitpoints, 106 filler hitpoints have to be saved for achieving same mesh in workgroup. If filler hitpoint wasn't used, in last workgroup hitpoints from two mesh will be.

If it is ensured that hitpoints in all workgroup are same, performance should be accelerate because all threads in workgroup are loading sequentially from same memory and this should achieve faster time.

Hitpoint sorting and local memory

If all photons are separated and similar hitpoints are saved in one workgroup it is possible to use local memory to save photons from separated photon map. Loading from global memory to local memory is similar like in previous approach.

Manual calculation of distance

This approach is here only to demonstrate possible bug from opencl when built in distance function is slower than manual calculation of distance.

Hitpoint clustering and spatial grid

This approach is most complex approach from all previous approaches. It tries to reduce numbers of searched photons from separated photon map by dividing this photon map by uniform grid and make close clusters. Hitpoints in one workgroup will be as close as possible - to reduce number of traversed bins through one workgroup.

Photons are not saved into one separate photon map for each mesh, but into special bins for each mesh. Numbers, size and positions of this bins is determined by creating grid structure. This structure is created for each mesh on CPU side.

Algorithm for creating regular grid divide mesh by longest axis by fixed numbers of bins. Pseudocode for creating regular grid is this:

Compute maximal and minimal position of mesh
Compute size of bin as
$d = max((longest\_size/max\_numbers\_of\_bins), default\_radius)$
Compute numbers of bins nx, ny, nz in each axis
Save minimal position $minPosition$

First minimal and maximal position is computed, this is done for navigating in mesh. Size of one bin should be equal to default radius of one hitpoint, but if mesh is too large it will generate to much bins in grid and therefore limitation from $max\_numbers\_of\_bins$ is used. Size of bin is equal to default radius of hitpoint because if this is true, navigation in grid is simple, it only compute photons in neighbor bins.

From this algorithm, four numbers and one vector is get. Vector $minPosition$ and number d is for computing position in regular grid. This position is in bins on axis and for computing offset in linear memory numbers nx, ny, nz are needed. Computing position x, y, z and $offset$ in grid from $position$ is computed by this formulas:

$$\begin{aligned} \vec{l} &= position - minPosition \\ x &= abs(l.x)/d \\ y &= abs(l.y)/d \\ z &= abs(l.z)/d \\ offset &= (x + y * nx) + (z * nx * ny) \end{aligned} \tag{7.1}$$

Because threads in workgroup have to load same things from memory, $position$ and radius R has to be same for all threads in workgroup. Computation $position$ of workgroup center and workgroup radius R could be done in CPU side and saved in every hitpoint.

Each thread then compute subspace where photons will be searched for. This subspace consist of numbers of bins computed from radius R and d. Then photons are loaded from this bins.

Saving photons into grid bins is made in same kernel as in photon sorting, this kernel has to only compute proper offset for pointer, increment that pointer and then save photon

to proper memory position. Speed of this kernel is still fast - 2 ms - and is still negligible in comparison with achieved acceleration.

Hitpoints in workgroup should be grouped together to close clusters, this will achieve small radius of this whole workgroup and thus number of traversed bins will be smaller. Size and position of clusters are made from k-means. Hitpoints are clustered on CPU, because this process will be done in whole rendering process only once.

For clustering, k-means algorithm was used first. K-means divide points into K clusters, so K has to be computed as $K = hitpoint\_size/workgroup\_size$. Image 7.2 shows how k-means divide hitpoints into clusters.

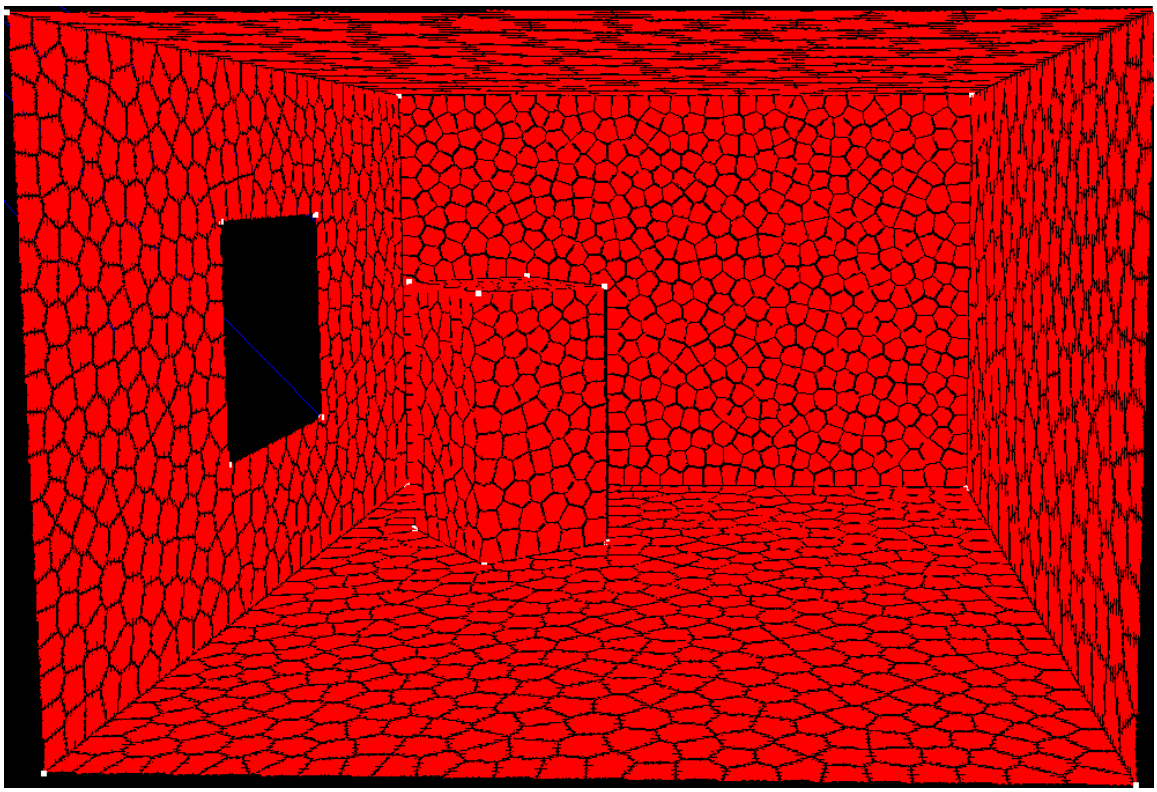


Figure 7.2: Hitpoint clustering made by k-means.

K-means clusters could be positioned in two bins. This could lead to more traversed bins in hitpoint illumination block. Another possible solution how to reduce memory loads are sort hitpoints in bins of cluster. Hitpoints sorted in grid could achieve that all workgroup will load only neighbors bins and this could lead to some acceleration. Image 7.3 shows how sorted hitpoints are positioned in scene.

7.4 Summary

All acceleration approaches was implemented on same scene and with same GPU as previous tests. Table 7.3 shows results of this approaches. As it can be seen, all proposed approaches lead to some sort of speedup. Local memory accelerate computation nearly twice. Photon sorting accelerate naive solution nearly four times.

Hitpoint sorting uses photon sorting with sorted hitpoints on CPU side, so some CPU overhead is needed. Because this task is performed only once - after raytracing pass - this

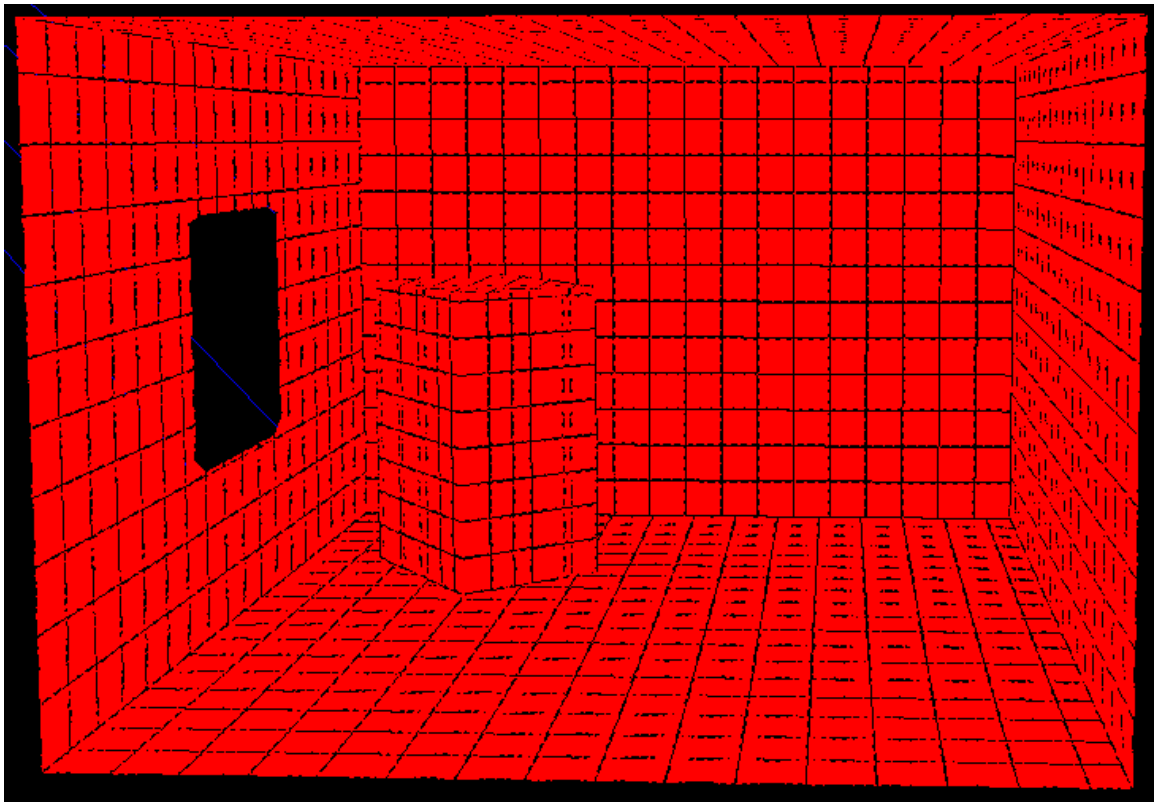


Figure 7.3: Hitpoints sorted in grid bins.

	320*280	1920*1080
Naive solution	2041 ms	38 603 ms
Local memory	1109 ms	17 750 ms
Photon sorting	611 ms	10 539 ms
Hitpoint sorting	484 ms	8 384 ms
Hitpoint sorting, local memory	327 ms	5 422 ms
Manual distance	125 ms	1 941 ms
K-means clustered hitpoints	172 ms	2 852 ms
Grid structure + k-means	34 ms	371 ms
Grid structure + grid clusters	69 ms	378 ms

Table 7.3: Performance evaluation between all proposed acceleration techniques.

overhead does not matter. Another accelerating approach use local memory with hitpoint sorting. This approach is much faster than all previous approaches.

All of this test was written in OpenCL. OpenCL has built-in function distance() and even when this functioun should be very fast, it isn't. Manual distance in table 7.3 means previous acceleration with manual distance computation. Using this hack, this block gets nearly three times speedup.

K-means clustered hitpoints experiment in table 7.3 is previous approach (Manual distance) with clustered hitpoints from k-means. Reason why this approach is slower is overhead from filler hitpoints, more later.

When grid structure was used on this clustered hitpoints from k-means, speed of hitpoint

Resolution	K-means time[ms]	Grid time[ms]	K-means creation[s]	Grid creation[s]
320*280	34	69	1.06	0.15
854*480	107	117	3.68	0.28
1280*720	234	194	12.65	0.29
1920*1080	371	378	57.11	0.47

Table 7.4: Evaluation of hitpoint illumination block performed by k-means or grid clusters. Evolution is done on several resolutions.

illumination was accelerated most. Clustering hitpoints into grid made similar speed for bigger resolution, table 7.4 shows evaluation between k-means and grid clustering. In this table is shown speed of creating this two algorithms. Clustering into grid bins is much more time efficient than using k-means clusters.

Resolution	No-clustering	K-means hitpoints	Grid hitpoints
320*280	22101	31424	70720
854*480	86366	126208	129856
1280*720	193062	284736	232704
1920*1080	432515	639488	473008

Table 7.5: Number of hitpoints based on clustering algorithm

K-means does not respect size of workgroups. If size of cluster is $size\_of\_workgroup+1$ then in second workgroup of this cluster are $1 + (size\_of\_workgroup - 1)$ filler hitpoints. What is really interesting is performance of k-means with lot of filler hitpoints. Table 7.4 shows number of filler hitpoint depending on resolution. K-means, in resolution 1920*1080, uses much more filler hitpoints than clustering into grid and even then, k-means is marginally faster.

The best acceleration of hitpoint illumination block is made by combining clustered hitpoints and regular grid. Speed of naive solution was accelerated 60x, for resolution 320*280, and 104x, for resolution 1920*1080. In comparison of CPU implementation and GPU implementation speed of hitpoint illumination block was accelerated 46x, for resolution 320*280, and 64x for resolution 1920*1080.

Chapter 8

Evolutionary clustering for GPU progressive photon mapping

This chapter describes evolution based algorithm for hitpoints clustering for progressive photon mapping. The idea for this clustering algorithm is to find out better clusters than k-means does, in order to achieve faster speed of hitpoint illumination block.

Problem of k-means clustering is that it does not respect size of workgroups and thus lot of useless hitpoints are involved in computation and power of GPU is wasted. Benefit of evolutionary algorithm could be better clusters not only in numbers of hitpoints in clusters but also in better positioning of clusters, because conditions of final clusters are directed by proper fitness function. And in this fitness function various data could be included.

8.1 Creating evolutionary algorithm

Chromosome representation

Because input of this clustering algorithm is array of hitpoints and output is array of centers of clusters, in chromosome centers of clusters have to be saved and some connection between hitpoints and clusters has to be saved also.

After considering various options, this chromosome structure was implemented:

- **centroids** - array of vectors, centers of clusters
- **hitToCentroid** - array of arrays of vectors, array of hitpoints are saved here for each centroid[1]
- **fitness** - unsigned int, fitness value of current chromosome
- **fitnessPercent** - float, scaled fitness value

Generation of initial population

Each candidate solutions in initial population are created by mutation k-means result. K-means bring evolution algorithm good candidate solution. Another way how to generate initial population is pick clusters centers randomly. This was originally used, but k-means lead to faster evolution and because of that, generation from k-means was selected.

[1] centroid = center of cluster

Mutation

First mutation randomly choose one cluster A, then pick nearest cluster B of this randomly chosen cluster A. The following operation depends on the number of hitpoints in this clusters, this operations are:

- **if** $size(A) + size(B) < wg\_size$ - merge centroids
- **if** $size(A) + size(B) > 2 * wg\_size$ - do nothing
- **else** - merge with probability depending on sum of sizes of this clusters, if size is approaching to $2 * wg\_size$ probability is lower than if sum is near wg_size.

If merge is performed, position of final centroid is computed as average point of all hitpoints in both centroids.

Second mutation choose random centroid and change his position randomly in specific bounding box. This mutation is performed with low probability and this mutation is in this process only for breaking solutions from their local minimum.

Third and fourth mutation are removing or adding new centroids into solution. If removing mutation is performed, clusters with lower number of hitpoints have higher probability of removing. If adding mutation is performed, position of new cluster is selected randomly. This mutations has 50% probability of performing and count of added/removed centroids is chosen randomly in range between one and five.

Evolution configuration

Because only mutations was proposed, evolution algorithm is using $1 + \lambda$ evolution. Size of population is 50. Number of iterations is set to 3000. After 3000 iterations, best candidate solution is set as result of evolution.

8.2 CPU computed fitness functions

Three fitness functions was proposed and measured, first reduce number of filler hitpoints, second reduce number of traversed bins in grid and third tries to reduce number of block memory accesses to global memory.

Reducing filler hitpoints fitness

Lot of filler hitpoints was generated in k-means clusters and this filler hitpoints waste GPUs power. First fitness tries to minimize number of filler hitpoints and fitness function is computed as:

$$Fitness = \sum_{c \in clusters} filler\_hitpoints(c) \tag{8.1}$$

where

- **clusters** - all clusters in candidate solution
- **c** - one cluster

- **filler_hitpoints(c)** - number of filler hitpoints in cluster c computed from size of cluster and workgroup size

This fitness function assumes sorted hitpoints in nearest clusters, so this procedure must be done before every fitness evaluation (this is the slowest procedure in whole genetic algorithm).

Value of this fitness function for k-means is 2438.

Grid bin counting fitness

Previous fitness only minimize numbers of filler hitpoints. It has no respect of cluster size (number of workgroups), geometric size (size of shared radius), memory load of this cluster (in respect of grid structure).

Extension of this naive fitness function is involvement of grid structure. In evaluation, partial fitness of one cluster is computed as number of traversed bins in grid. For this computation precise average of cluster hitpoints has to be computed and shared radius of all hitpoints in cluster has to be computed too. Grid bin counting fitness is defined like this function:

$$Fitness = \sum_{c \in clusters} bins(c) * workgroups(c) \tag{8.2}$$

where:

- bins(c) - number of traversed bins
- workgroups(c) - number of workgroups with size of cluster c

Value of this fitness function for k-means is 3165.

Memory access counting fitness

Grid bin counting fitness assume that processing all bins has same time, but this could be true only if all bins has same number of saved photons. Thousands of photons are in bins where light has direct connections with bin, dozens of photons are in bins near edge of mesh, with only „indirect“ photons saved. This imbalance make previous fitness wrong and some sort of memory access computation should be included.

For memory accesses prediction, photon distribution map was created. This map will not change drastically if topology of scene does not change. Because of this, precomputation of this distribution map is possible. This map is computed as average of thousand iterations and this map is saved as array of integers, where size of array is equal to number of bins in scene.

Then, memory access counting fitness is defined as:

$$\begin{aligned} Fitness &= \sum_{c \in clusters} memory\_access(c) * workgroups(c) \\ memory\_access(c) &= \sum_{x \in bins(c)} \frac{photons\_distribution(x)}{workgroup\_size} \end{aligned} \tag{8.3}$$

where:

- $bins(c)$ - set of traversed bins from cluster c
- $photons\_distribution(x)$ - return number of photons in bin x

Value of this fitness function for k-means is 9877.

Speed evaluation

Speed evaluation was made on same scene as all previous tests. Resolution of this evaluation was 320*280 and only bottom surface was used. This surface was chosen because this surface has most photons saved in all scene - light directly hits this surface. Evaluated speed of this surface, rendered with grid structure and clusters made by k-means, was 23ms.

Test	Fitness value	Number of clusters	Speed [ms]	Creating time [s]
1	582	77	19,977	547
2	390	57	22,256	520
3	646	78	24,192	551
4	582	76	21,128	537
5	646	69	23,917	503
6	645	83	23,789	663
7	710	76	22,307	533
8	709	88	20,955	605
9	582	68	27,134	502
10	454	61	22,815	452
average	594	73,3	22,845	541
minimal	390	57	19,977	452

Table 8.1: Evaluation of evolution of reducing filler hitpoints fitness with 3000 iterations

Test	Fitness value	Number of clusters	Speed [ms]	Creating time [s]
1	2408	103	19,713	694
2	2400	104	18,691	735
3	2432	104	19,334	702
4	2363	101	19,187	698
5	2387	106	18,998	734
6	2364	100	18,795	673
7	2411	102	19,767	694
8	2384	101	18,454	704
9	2394	104	19,503	701
10	2448	105	20,155	716
average	2399	103	19,259	705
minimal	2363	100	18,454	673

Table 8.2: Evaluation of evolution of grid bin counting fitness with 3000 iterations

Tables 8.1, 8.2 and 8.3 shows evaluations of their respective fitness function. Best fitness function is fitness function called memory access counting fitness. Using this fitness

function in genetic algorithm lead to average speed of hitpoint illumination block 18,24 ms and fastest speed in this experiment was 16,26 ms.

Speed of creating solution by k-means is 3 ms. Speed of creating solution by evolutionary algorithm is 705 seconds in average.

All fitness function has one big drawback, low value of this fitness function does not every time mean better final speed. Even most complex fitness function for counting memory accesses has this drawback.

Test	Fitness value	Number of clusters	Speed [ms]	Creating time [s]
1	7252	104	16,435	715
2	7499	99	16,823	678
3	7528	105	19,173	721
4	7346	104	19,415	712
5	7317	100	16,262	704
6	7546	105	18,652	710
7	7205	96	19,868	671
8	7302	101	18,689	690
9	7256	99	19,503	688
10	7380	106	17,608	755
average	7363	101	18,241	704
minimal	7205	96	16,262	671

Table 8.3: Evaluation of evolution with memory access counting fitness with 3000 iterations

8.3 GPU computed fitness function

Because it is hard to describe suitability of candidate solution and previous fitness functions does not work well, another try to make good fitness function is done directly on GPU.

When fitness function is evaluated for candidate solution, this solution is sent to GPU where only hitpoints with this solution are evaluated in hitpoint illumination block. Value of fitness function is then speed of performing this block directly on GPU. This fitness value is not estimated and is without any error. Problem with computing speed of performing on GPU is changing times between evaluation, times has little variance in 0.2 - 0.4 ms. Because this variance, five runs are performed on GPU and fastest run is taken as result of evaluation.

Evolution algorithm with this GPU fitness function created fastest clusters, as is shown in table 8.4. Average speed is 14.68 ms and fastest speed is 13.84 ms. This clusters was find out with 10 times lower number of iterations, but it takes enormous time of evaluation on GPU and thus final time of creation is very huge.

8.4 Summary

This evolutionary based algorithm generate faster clusters for hitpoint illumination block than k-means does and including this algorithm in rendering process should lead to faster execution times on GPU. Big drawback of this algorithm is time consumption. This algorithm is only usable as precomputation for static scenes.

Test	Fitness value / speed [microseconds]	Number of clusters	Creating time [s]
1	15123	118	1667
2	14424	115	1698
3	14348	103	1617
4	15017	108	1642
5	13846	111	1660
6	15002	121	1799
7	15326	121	1730
8	14881	109	1756
9	14425	110	1604
10	14470	111	1642
average	14686	112	1681
minimal	13846	103	1604

Table 8.4: Evaluation of evolution with GPU fitness, 300 iterations

Slowest operation in evolution with CPU fitness functions is sorting hitpoints into clusters. For accelerating, better structure of chromosome should be find out, or this sorting should be done GPU. If GPU fitness is used, slowest function is computing fitness of candidate solution directly on GPU and for acceleration better GPU should be used.

One way how to reduce number of this slow operations and thus reduce time of evolution is to optimize evolutionary algorithm. But this task is very hard, lot of experiments has to be performed for finding better probabilities of mutation, size of population, better mutation, suitability of crossover operator. Problem in each experiment is randomness, evolution based algorithms are based on random values and for deciding if (for instance) this specific probability of mutation is better, dozens of evolution has to be performed and it takes lot of time.

Chapter 9

Conclusion

Goal of this master thesis was implement various number of photon mapping implementations and perform few test on them, this goal was achieved. This implementation was done on normal photon mapping on CPU, progressive photon mapping on CPU and GPU. In last part of this thesis genetic algorithm for acceleration of photon mapping was proposed.

In normal photon mapping implementation, whole renderer was made. This renderer include fast ray-triangle intersection, spatial subdivision, loading scene from assimp library. Performed experiments on this renderer was on achieved acceleration from spatial index and mostly on indices from FLANN and ANN libraries. Best spatial index from this two libraries was kd-tree from FLANN library. In the end of photon mapping section, test of approximate nearest search on photon mapping was performed. This test shown that this type of searching is inappropriate for photon mapping and this approximate search lead to worse final result.

In progressive photon mapping implementation, progressive variant of photon mapping was implemented on CPU first, and then it was implemented on GPU. This two implementation was compared between each other. Speed of GPU implementation's block was measured and slowest block was block for computing illumination from photon map. This block was implemented very naively first, but for this block various acceleration technique was proposed. Best technique is combining grid structure upon photon map and clustered hitpoints from k-means. Speed of this slowest block was accelerated 64 times compared to CPU variant on GPU.

In last chapter, slowest block of progressive photon mapping on GPU was accelerated through evolutionary based algorithm. This genetic algorithm tries to find more suitable clusters than k-means does. Three fitness functions on CPU and one fitness function on GPU was proposed for this evolution based clustering algorithm. Result speed of illumination block was nearly half accelerated, but it takes enormous time to compute suitable clusters on bigger resolution (30 minutes on lowest resolution for one mesh) and because of that it could be used only as precomputation for static scenes, when it is precomputed once, saved and used many times later.

This master's thesis show me that evolution based algorithms could lead to faster rendering time and I want to try implement them in various occasions - in constructing kd-tree, to faster ray-triangle intersections. But first, I have to made really photorealistic render and I plan to made it on OptiX framework.

While I was working on master thesis I learned a lot. I learned how advanced global illumination methods work, I improved skills in math, geometry, programing, analytic thinking and many other little skills. It was long hard journey, but it was worth it.

Bibliography

[1] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[2] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[3] P. Bentley. *Evolutionary Design by Computers*. Number pt. 1 in Evolutionary Design by Computers. Morgan Kaufman Publishers, 1999.

[4] Michael F. Cohen, Shenchang Eric Chen, John R. Wallace, and Donald P. Greenberg. A progressive refinement approach to fast radiosity image generation. *SIGGRAPH Comput. Graph.*, 22(4):75–84, June 1988.

[5] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18(3):137–145, January 1984.

[6] Manfred Ernst, Tomas Akenine-Möller, and Henrik Wann Jensen. Interactive rendering of caustics using interpolated warped volumes. In Kori Inkpen and Michiel van de Panne, editors, *Graphics Interface*, pages 87–96. Canadian Human-Computer Communications Society, 2005.

[7] James Gurney. Color bleeding. http://gurneyjourney.blogspot.cz/2010/05/color-bleeding.html, 2015. [Online; accessed 06-January-2015].

[8] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. *ACM Trans. Graph.*, 27(5):130:1–130:8, December 2008.

[9] Jiří Havel and Adam Herout. Yet faster ray-triangle intersection (using sse4). *IEEE Transactions on Visualization and Computer Graphics*, 16(3):434–438, 2010.

[10] Adam Herout. *PGR Opora*. 2008.

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

[12] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Ltd., Natick, MA, USA, 2001.

[13] Henrik Wann Jensen and Niels Jorgen Christensen. Efficiently rendering shadows using the photon map. In *Edugraphics + Compugraphics Proceedings*, pages 285–291. Publications, 1995.

[14] Jiří Sochor a Petr Felkel Jiří Žára, Bedřich Beneš. *Moderní počítačová grafika*. Computer Press, 2004. ISBN 80-251-0454-0.

[15] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986.

[16] Keenan Krane. Bias in rendering, 2006.

[17] Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. In *PROCEEDINGS OF THIRD INTERNATIONAL CONFERENCE ON COMPUTATIONAL GRAPHICS AND VISUALIZATION TECHNIQUES (COMPUGRAPHICS ?93*, pages 145–153, 1993.

[18] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics, Third Edition*. Course Technology Press, Boston, MA, United States, 3rd edition, 2011.

[19] Fred E. Nicodemus. Directional reflectance and emissivity of an opaque surface. *Applied Optics*, 4(7):767–775, 1965.

[20] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010.

[21] Stefan Popov, Johannes Guenther, Hans-Peter Seidel, and Philipp Slusallek. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum*, 2007.

[22] Eric Veach and Leonidas Guibas. Bidirectional Estimators for Light Transport. 1994.

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

[24] Wikipedia. Caustic (optics) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Caustic%20(optics)&oldid=637425857, 2015. [Online; accessed 06-January-2015].

[25] Wikipedia. Śledzenie promieni. http://pl.wikipedia.org/wiki/%C5%9Aledzenie_promieni, 2015. [Online; accessed 08-January-2015].

[26] Wikipedia. Radiosity (computer graphics) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Radiosity%20(computer%20graphics)&oldid=636016049, 2015. [Online; accessed 09-January-2015].

[27] Wikipedia. Ray tracing (graphics) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Ray%20tracing%20(graphics)&oldid=637522305, 2015. [Online; accessed 09-January-2015].

[28] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 Papers*, SIGGRAPH Asia '08, pages 126:1–126:11, New York, NY, USA, 2008. ACM.