Praca dyplomowa

Temat pracy:
Parallel implementation of ray-tracing based global illumination algorithms

Autor: Michał Radziszewski
Kierunek: Informatyka

Promotor: dr inż. Witold Alda

Kraków, 22 czerwca 2006
ACKNOWLEDGEMENTS

I would like to thank my Advisor Dr. Eng. Witold Alda for his support during my work on this thesis, his help and his useful advices which contribute significantly to creation of this thesis.
Abstract

Global illumination algorithms are designed to produce convincing and realistic images of an artificial world. To achieve this goal they must be physically based and general enough to allow as broad as possible class of input models. We believe that currently only non-deterministic ray tracing based algorithms can fully satisfy these requirements. However, applications that use them suffer from their complexity and long rendering periods. Modern hardware does not allow using global illumination with complex scenes in real time. But ray tracing has a great advantage. It is comparatively easy to parallelize. Using network of many machines above problems can be mitigated.

We start by short description of strict mathematical equations that are base of all light transport algorithms based on geometric optics and some numerical integration methods that may be used to solve them. We also derive a method to change location of errors due to numerical integration of a product of two functions, which is then used in our new full spectral color representation.

Next we present a parallel framework for easy using and/or implementing such algorithms. Our system is based on abstract models, which can be accessed by interface allowing only sampling. We believe that this new approach is general enough to implement many of today’s algorithms and to hide arbitrarily complex model description behind these layers of abstraction. We also exploit symmetry of light transport and give a camera almost the same interface as for model, which greatly simplify the implementation of rendering algorithms. Our framework is also independent of image pixels, and we use this feature to allow efficient store images in sequential disk files.

Then we show versatility of our system by implementing classic algorithms – Path Tracing, and Bidirectional Path Tracing with a new optimization that is consistent with our framework and is addressed to replace previous approach. We also present simple camera models and show how they fit into our framework.

Finally we briefly describe the implementation of classic 3D concepts, such as triangles, materials, acceleration structures and so on. We show how to customize them to work with physically correct algorithms and how they fit into our framework.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>4</td>
</tr>
<tr>
<td>CONTENTS</td>
<td>5</td>
</tr>
<tr>
<td>LIST OF SYMBOLS</td>
<td>9</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>10</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>10</td>
</tr>
<tr>
<td>1.2 Applications</td>
<td>10</td>
</tr>
<tr>
<td>1.3 Thesis organization</td>
<td>11</td>
</tr>
<tr>
<td>2 SHORT OVERVIEW OF RAY TRACING METHODS</td>
<td>12</td>
</tr>
<tr>
<td>2.1 Classic Ray Tracing</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Distributed Ray Tracing</td>
<td>13</td>
</tr>
<tr>
<td>2.3 Path Tracing</td>
<td>13</td>
</tr>
<tr>
<td>2.4 Robust Algorithms</td>
<td>13</td>
</tr>
<tr>
<td>3 BASIC DEFINITIONS AND THEORETICAL CONCEPTS</td>
<td>14</td>
</tr>
<tr>
<td>3.1 Light Transport</td>
<td>14</td>
</tr>
<tr>
<td>3.1.1 Geometric Optics</td>
<td>14</td>
</tr>
<tr>
<td>3.1.2 Radiometry</td>
<td>15</td>
</tr>
<tr>
<td>3.1.3 Rendering Equation</td>
<td>16</td>
</tr>
<tr>
<td>3.1.4 Volumetric Extensions</td>
<td>17</td>
</tr>
<tr>
<td>3.1.5 Integral over Paths</td>
<td>18</td>
</tr>
<tr>
<td>3.2 Numerical Integration of Rendering Equation</td>
<td>18</td>
</tr>
<tr>
<td>3.2.1 Convergence Problems</td>
<td>19</td>
</tr>
<tr>
<td>3.2.2 Monte Carlo Quadratures</td>
<td>19</td>
</tr>
<tr>
<td>3.3 Probabilistic Techniques</td>
<td>20</td>
</tr>
<tr>
<td>3.3.1 Stratified Sampling</td>
<td>20</td>
</tr>
<tr>
<td>3.3.2 Russian Roulette and Splitting</td>
<td>20</td>
</tr>
<tr>
<td>3.3.3 Importance Sampling</td>
<td>21</td>
</tr>
<tr>
<td>3.3.4 Multiple Importance Sampling</td>
<td>21</td>
</tr>
<tr>
<td>3.3.5 Biased Methods</td>
<td>21</td>
</tr>
<tr>
<td>3.3.6 Shift of Error Location</td>
<td>21</td>
</tr>
<tr>
<td>3.4 Quasi Monte Carlo</td>
<td>22</td>
</tr>
<tr>
<td>3.4.1 Discrepancy</td>
<td>22</td>
</tr>
<tr>
<td>3.4.2 Convergence</td>
<td>23</td>
</tr>
<tr>
<td>3.4.3 Techniques</td>
<td>23</td>
</tr>
<tr>
<td>3.4.4 Limitations</td>
<td>24</td>
</tr>
</tbody>
</table>
4 SOLUTION OF THE RENDERING EQUATION ................................................. 25

4.1 View dependent vs. view independent ................................................. 25

4.2 Ray Tracing – Expansion Approach .................................................. 25
  4.2.1 Mathematical Formulation ......................................................... 25
  4.2.2 Advantages and Limitations ..................................................... 25

4.3 Radiosity – Finite Elements Approach .............................................. 26
  4.3.1 Mathematical Formulation ......................................................... 26
  4.3.2 Advantages and Limitations ..................................................... 26

4.4 Other Methods .................................................................................. 26
  4.4.1 Two-pass Techniques ............................................................... 26
  4.4.2 Irradiance Caching and Photon Mapping Algorithms ............... 26

4.5 Computability of Ray Tracing ............................................................ 27
  4.5.1 Kinds of Light Paths ................................................................. 27
  4.5.2 Local Sampling Scheme ........................................................... 27
  4.5.3 Approximations ....................................................................... 28

4.6 Ray Tracing Algorithms ................................................................. 28
  4.6.1 Path Tracing ............................................................................ 29
  4.6.2 Bidirectional Path Tracing ......................................................... 29
  4.6.3 BDPT Optimization ................................................................. 30

5 SPECTRAL REPRESENTATION .......................................................... 31

5.1 Physics, Physiology and Colors ....................................................... 31

5.2 Why RGB is wrong ........................................................................... 31

5.3 Full Spectrum .................................................................................. 32
  5.3.1 Point Sampling ......................................................................... 32
  5.3.2 Conversion to XYZ ................................................................. 33
  5.3.3 XYZ to RGB transformation ................................................... 33
  5.3.4 Conversion from RGB ............................................................. 36
  5.3.5 Basic Operations .................................................................... 37
  5.3.6 SIMD Parallelization ............................................................... 38

6 FRAMEWORK ..................................................................................... 39

6.1 Overview ......................................................................................... 39
  6.1.1 Abstractness and Efficiency .................................................... 39
  6.1.2 Extensions Support ................................................................. 39

6.2 Geometry ....................................................................................... 39

6.3 Spectral Representation ................................................................. 40

6.4 Sampling Interface ......................................................................... 40
  6.4.1 Objects .................................................................................. 41
  6.4.2 Cameras ................................................................................ 41

6.5 Frame Buffers ................................................................................ 41

6.6 Solvers ............................................................................................ 41
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.3.1</td>
<td>Tree Building</td>
<td>56</td>
</tr>
<tr>
<td>9.3.2</td>
<td>Tree Traversal</td>
<td>56</td>
</tr>
<tr>
<td>9.3.3</td>
<td>Optimizations</td>
<td>57</td>
</tr>
<tr>
<td>9.3.4</td>
<td>Parallel Search</td>
<td>58</td>
</tr>
<tr>
<td>9.3.5</td>
<td>Future Extensions</td>
<td>58</td>
</tr>
<tr>
<td>9.4</td>
<td>Color and Scalar Sources</td>
<td>58</td>
</tr>
<tr>
<td>9.5</td>
<td>Textures</td>
<td>59</td>
</tr>
<tr>
<td>9.5.1</td>
<td>Filtering</td>
<td>59</td>
</tr>
<tr>
<td>9.5.2</td>
<td>Reconstruction</td>
<td>59</td>
</tr>
<tr>
<td>9.5.3</td>
<td>Image Formats</td>
<td>60</td>
</tr>
<tr>
<td>9.6</td>
<td>Materials, Reflection Models</td>
<td>60</td>
</tr>
<tr>
<td>9.6.1</td>
<td>Transparency Optimization</td>
<td>61</td>
</tr>
<tr>
<td>9.7</td>
<td>Light Sources</td>
<td>61</td>
</tr>
<tr>
<td>9.8</td>
<td>Bump Mapping</td>
<td>61</td>
</tr>
<tr>
<td>9.9</td>
<td>Volumetric Effects</td>
<td>62</td>
</tr>
<tr>
<td>9.10</td>
<td>Procedural Modeling</td>
<td>63</td>
</tr>
<tr>
<td>10</td>
<td>RESULTS</td>
<td>65</td>
</tr>
<tr>
<td>10.1</td>
<td>Path Tracing vs. Bidirectional Path Tracing</td>
<td>65</td>
</tr>
<tr>
<td>10.2</td>
<td>Reduction of Visibility Tests</td>
<td>66</td>
</tr>
<tr>
<td>10.3</td>
<td>Transparency Optimization</td>
<td>67</td>
</tr>
<tr>
<td>10.4</td>
<td>Parallel Rendering of Large Scenes</td>
<td>68</td>
</tr>
<tr>
<td>10.4.1</td>
<td>Convergence Tests</td>
<td>71</td>
</tr>
<tr>
<td>11</td>
<td>CONCLUSIONS AND FUTURE WORK</td>
<td>75</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td></td>
<td>76</td>
</tr>
<tr>
<td>APPENDIX A: HARDWARE REQUIREMENTS</td>
<td></td>
<td>79</td>
</tr>
<tr>
<td>Processor</td>
<td></td>
<td>79</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td>79</td>
</tr>
<tr>
<td>Network</td>
<td></td>
<td>79</td>
</tr>
</tbody>
</table>
List of Symbols

$L$  radiance
$E$  irradiance
$\Phi$  power
$pdf$  probability distribution function
$cdf$  cumulative distribution function
$\xi$  canonical uniform random variable
$\mu$  arbitrary measure
$\omega$  unit vector
$\lambda$  wavelength
$S^2$  space of all unit vectors
$R^3$  space of all points in scene
$\delta$  Dirac delta distribution
$N$  normal direction
$\sigma(\omega)$  solid angle measure
$f_r$  bidirectional scattering distribution function (BSDF)
$f_p$  phase function
$\sigma_a$  absorption coefficient
$\sigma_s$  scattering coefficient
$\sigma_e$  extinction coefficient
1 Introduction

1.1 Background

The idea of this thesis is to design and implement a rendering system which supports global illumination algorithms. Two main goals have been taken under consideration: find the best algorithms for this purpose and build modular and flexible software. The first one is to use only these algorithms that can be proven to be correct both mathematically and physically. This approach allows realizing the extremely important intention of making system in which user says what should be rendered instead of giving detailed description how to render a given scene. In other words, it is enough to place the lights on the scene, and the physics embedded in algorithms should do the rest. Our intention is to make scene description fully declarative, without going into details that should be computed automatically by the system. In particular, contrary to most rendering programs, presented platform is capable of lighting the scene realistically with only real light sources. It means that no artificial sources which usually light shaded areas are introduced.

Beside this main feature, the rendering system automatically solves the texture mapping. It is enough to point the file with specific material, and all the mapping steps such as filter selection and other, are performed by the platform behind user knowledge. This way the actual rendering is only a single call of API function, with possibly requested output quality as a single parameter.

In order to achieve the second goal, the platform has to allow a clear decomposition on several almost independent modules. This gives the required ease of changing only fragments of code by using abstract interfaces. Our intention is to allow to experiment with different algorithms with only minimal required code changes. Due to its architecture, the system is not optimized for maximum speed, but is easy to use and gives trustworthy results.

We strongly believe that this approach is the future of rendering. Currently available real time rendering libraries with all their quirks and tricks can easily make the programmer insane (which is a personal Author’s experience). As a result of these artificial assumptions put into the libraries, we have many common problems with compatibility between applications, graphics hardware and drivers. Very often if the application works at all, works very unstable. Today convincing images and animations with unparallel quality can be rendered using global illumination, and we believe that when the computational power achieves sufficiently high level to allow using these algorithms in real time, nobody will use the rasterization any more. This moment may come sooner not only thank to hardware progress, but also due to founding new, better and more reliable algorithms and by using parallel computations. These perspectives definitively motivated the Author to undertake the project and constantly encourage him to continue the development of this work.

1.2 Applications

Global illumination algorithms have variety of different applications, ranging from science and engineering tools to contemporary entertainment. Its usefulness is continuously growing with rapid advancement in available computational power of computers. Our main goal is to design and implement algorithms that produce realistic and physically accurate output from given description of scene. This strict physical correctness is very important in some CAD applications. For example, illumination in building designed in architectural program should be as close as possible the real result, which allows making better projects. Predictability of output from these algorithms is also invaluable help when making animations or images of artificial world. Global illumination may save huge amount of time spent to design the realistic lighting. In our system it is enough to place lights only where they should be, and scene renders excellent, without any special tricks with additional lights, special treatment of mirrors and other awful requirements of commonly available rendering applications.

Currently available computational power is not sufficient to use correct global illumination algorithms in real time. But we believe that exponential increase of computers’ speed allow this in near future. With high level of parallelization we are much nearer this goal than with classic sequential approach. When this becomes available, computer games will not have to use rasterization, and it will be much easier to design them and their artificial worlds will be look much more realistic and convincing.
1.3 Thesis organization

In the second chapter there is a brief summarization of the most significant attempts to solve global illumination problem. In the next two chapters a short theoretical introduction to global illumination is presented. The first one contains all the necessary mathematics and physics needed to precisely define the rendering problem. The second one shows some of possible solutions, describing their advantages and drawbacks. This theoretical introduction is followed by a new approach to full spectral rendering in chapter five. In the chapters six and seven, there is described in detail the framework of implemented platform, in sequential form (chapter six) and with some theory and aspects of writing parallel applications (chapter seven). The latter chapter also describes and explains the design and implementation of parallel program with all the necessary changes from sequential one. The main goal of the framework description is to explain the main ideas of the platform, rather than being detailed code reference. The latter due to its large size is available on CD attached to the thesis. The code itself consists of about 20k lines, and therefore the reference is also huge. In chapter eight it is shown how to enhance the functionality of implemented platform by making custom extensions (illustrated with many already implemented examples). The implementation of the scene consisting of geometrical objects modeled with common graphics elements is described in chapter nine. This object model may be seen as custom extension, but due to its large complexity it has been placed in a separate chapter. Finally we summarize our work and display some of the results. We also try to predict the future extensions and changes that can be done to make this platform better.

The results obtained within this thesis, regarded by the Author as most important and original are as follows:

- the entire framework which is based completely on realistic, physical illumination (chapter 5),
- optimization of Bidirectional Path Tracing algorithm (chapter 4.6.3),
- usage of full spectral representation instead of most common RGB model, presented with the analysis of drawbacks of the latter method (chapter 5),
- parallel implementation with the help of libRPC library (chapter 7 and appendix B),
- original solution for transparent objects rendered with Path Tracing and Bidirectional Path Tracing (chapter 9.6.1),
- support of rendering with usage of compressed textures (appendix C).
2 Short Overview of Ray Tracing Methods

In this chapter a short overview of basic ray tracing algorithms is presented. The main reason of putting it here is because the framework developed in the thesis is based on ray tracing ideas mainly and that we want to give a general and somehow intuitive image of the method before principal mathematical definitions are introduced in the next chapter. First it is explained a classic Whitted approach, which is a first well known working raytracer. Next there is shown how researchers overcome its artifacts, which leads to mathematical formulation of light transport problem. Finally there is a discussion of algorithms that try to accelerate the basic path tracing algorithm in more difficult lighting conditions.

2.1 Classic Ray Tracing

In 1980 Turner Whitted presented his algorithm [WHITTED1980]. His publication is widely known as a first recursive raytracer. To our knowledge, there are some earlier approaches that use the idea of ray tracing, but this one is more general and formulated in elegant way.

The algorithm is very simple, but nevertheless may produce very realistic images of multiple reflections and/or refractions. In fact, it is generalization of standard ray casting algorithm formulated in recursive way. When a ray hits surface which has a perfectly specular component (reflection or refraction or both) there are generated up to two specular rays in exactly defined directions. If a surface has a diffuse (Lambertian) component, direct lighting computations are made by connecting point on surface to each point light source and add light contribution only if it is visible. Surface then may be linear combination of diffuse and specular reflections or refractions. Contribution of specular rays (if they are present) are computed with the same algorithm (here is the recursive call) and added to direct lighting result. Typically the recursion is terminated when ray strength becomes sufficiently small due to many reflections and contributes very little to image or at the prespecified depth to avoid infinite computations. In Figure 2-1 there is an example of tracing two rays, one hit matte surface and algorithm computes only direct lighting with ray casting, and second is recursively evaluated because it hits glass.

While this algorithm realistically computes the reflections and refractions, it has serious drawbacks. First of all, it accepts only optically perfect objects, such as ideal mirrors or point light sources which do not exist in real world. This results in unnaturally sharp edges of shadows and reflections. To make things worse, when there are many surfaces that simultaneously reflect and refract light (such as glass) complexity of the algorithm with respect to light path length is exponential and therefore it needs extremely long rendering times. It is also worth noticing that in this approach light sources can lit directly only diffuse surfaces. The algorithm cannot handle reflections in which light is shining on a diffuse surface indirectly reflected from a mirror. All the light that shines from light sources directly on specular surfaces will not contribute to the final image. The subtle error, which results in specific asymmetry (mirrors are perfectly all right if they are placed near the camera but not when they are near the light source; in fact a chain of mirror reflections may lie only next to camera and may not be separated by imperfect reflections), appeared to be difficult and have not been solved until the appearance of bidirectional algorithms several years later. This class of algorithms is described later in this chapter.
All the simplifications make the final rendered image much too dark, so there must be introduced an ‘ambient light’ term similar as in rasterization, to lighten all image by constant amount and minimize these errors.

### 2.2 Distributed Ray Tracing

The algorithm, presented by Robert Cook, Thomas Porter and Loren Carpenter [Cook+1984] is designed to mitigate the problem of sharp edges of reflections and shadows in previous approach. In their article there is introduced the concept of stochastic sampling of the reflection function. Surfaces may have arbitrary angular dependent reflection, not only the linear combination of purely diffuse and purely specular reflection. When a ray hits such surface, its appearance is computed by the Monte Carlo integration of lighting multiplied by reflection function. The new algorithm has capability of handling area light sources, fuzzy reflections, depth of field and motion blur, which is significant improvement.

Unfortunately, this approach is still not correct, because sample rays have number of reflections limited to one (to compute the soft shadows) or two (to compute fuzzy reflections) which still violates conservation of energy. The ‘ambient light’ still must be used to avoid too dark parts of image.

### 2.3 Path Tracing

The first mathematically correct solution to the light transport problem in geometric optics was given by James Kajiya [Kajiya1986]. In this article he presented integral equation that describes light transport, and its direct solution, which leads to stochastic path tracing algorithm. The algorithm almost correctly calculates energy balance by tracing paths of rays. Paths are not terminated on non-specular surfaces, as they were in previous approaches, but the next scattering direction is chosen randomly. In each intersection point there is ray traced between this point and randomly chosen point on light source. Area light sources may be accidentally intersected by a ray. The violation of energy balance is slight and appears as terminating path generation after predefined number of bounces. Nevertheless, on highly reflective environment, this may lead to visibly incorrect solution.

It is worth to notice that algorithm has the same asymmetry problem as classic Whitted ray tracer. In 1990 James Arvo and David Kirk [Arvo+1990] presented an algorithm based on Boltzmann transport equation which represents the light as a stream of particles, photons, which are traced from light sources. After photons hit the surface, they scatter in random direction, while every hit (if visible) is recorded by camera. This technique removes completely problems with mirrors near light sources (thus allowing effective rendering of caustics), but it fails when mirrors appear next to camera, which was easy for previous techniques. In fact it does not solve the asymmetry problem, but inverts it. Also particle transport suffers from very poor performance especially when scene contains many light sources and majority of them are invisible to camera.

This article introduces to computer graphics one more technique, which is still in use in probabilistic rendering – a Russian roulette. It allows to randomly terminate light path without violating energy balance. Russian roulette may be also used to improve path tracing and make it completely correct (at least on physically plausible scenes, as ideal specular surfaces do not exist in real world).

### 2.4 Robust Algorithms

All previous probabilistic ray tracing techniques have been suited for some class of input scenes and fail when scene does not satisfy these requirements. For example path tracing works well only if there are no specular surfaces next to light sources and indirect illumination of the scene is comparatively small with respect to the direct one. The goal of robust algorithms is not to achieve physical correctness (as it was possible in path tracing) but to make rendering fast and reliable on as broad accepted input scene types as possible. The first robust technique was bidirectional approach. It was developed independently by Eric LaFortune [LaFortune+1993] and Eric Veach [Veach1997]. Both these techniques build path simultaneously starting from the camera and from the light source, connecting both parts somewhere in middle. The implementations of the two mentioned techniques are similar, but they are completely different in underlying theoretical framework. The LaFortune’s solution is based on solving GRDF (Global Reflectance Distribution Function), while Veach’s one transforms integral light transport equation to quadrature, and solves the quadrature by multiple importance sampling (explained further).

There are also some more algorithms worth mentioning, however our platform is not designed to support them easily. They are tricky to implement in parallel environment and in fact often worse than highly optimized bidirectional approach. We found particularly original and interesting Photon Mapping [Jensen2001] and Markov chain based Metropolis Light Transport [Veach1997]. Notice that the Photon Mapping is possible to implement in our platform, but not in parallel mode. With current kernel design Metropolis approach cannot be implemented at all.
3 Basic Definitions and Theoretical Concepts

3.1 Light Transport

In this chapter we present mathematical concepts which lie behind correct path tracing algorithm. With slight additional work it may be used also as a basis of bidirectional and Markov chain methods.

We start from the description of geometric optics, as all subsequent theory and our system can deal only with it. First we describe all effects that can be predicted by geometric optics. Next we show its assumptions and limitations that result from the model's simplification. Next we discuss some radiometric quantities, which will be computed by our system. At the end of this chapter we present derivation of the equation that governs light transport in geometric optics.

Because all the mathematics presented here is well known and appears in many other works, discussion here is brief. This thesis puts all important things in one place without getting in too much detail. One interested in light transport theory may consult [PHARR+2004] – excellent introduction to this topic, easy to understand, or [VEACH1997] – which is mathematically more advanced, using operator formulation, as well as [PAULY1999] which treats the light transport in 3D volumes, the topic not well covered in two previous papers. There are much more books on light transport theory, but the content of the above three is enough to fully understand our framework and implementation. The good references for mathematical analysis and statistics suitable for our project are [NIEDOBA2001] and [PLUCINSCY2000].

3.1.1 Geometric Optics

The basic system assumption is to allow simulation of all geometric optics phenomena – for example soft shadows, indirect lighting, dispersion, etc. using statistical methods. Other effects (such as e.g. diffraction, which requires wave optics) are ignored. The theoretical model is based on photons with following simplifying assumptions:

- the number of photons is huge while the photon energies are extremely small – any distribution of photons may be treated as continuous value;
- photons do not interact with each other, thus effects such as interference cannot be simulated;
- polarization, diffraction and all other interactions which could affect photons' movement are neglected, so photons travel along straight lines (we do not expect mirage effects seen due to refractions in unequally heated air);
- photon collisions with surfaces and particles (e.g. fog) of non transparent volumes are elastic, which means that photons cannot change wavelength during scattering (scattering function domain is one dimensional \( \lambda \) (wavelength), not \( \lambda \times \lambda \));
- speed of photons is infinitely large, the scene is assumed to be always in equilibrium state.

After consideration all of these assumptions each photon can be described by its state defined as a triple \((x, \omega, \lambda)\), where \(x\) is current location, \(\omega\) is direction of movement and \(\lambda\) is photon wavelength (however no wave effects can be simulated, \(\lambda\) is used only in dispersion and color computations). This creates 6-dimensional phase space \(\Psi\) for photons:

\[
\Psi = R^3 \times S^2 \times R^+
\]

with an associated phase space density function \(f\):

\[
 f: \Psi \times R^+ \rightarrow R^+_0,
\]

where value in point \((x, \omega, \lambda, t)\) is photon density in state \((x, \omega, \lambda)\) at time \(t\). Despite the assumption that system is always in equilibrium state with respect to illumination, the time is important at least for one reason. It is necessary to extend the solution by integrating over time to account for motion blur, which is often necessary for animation effects. Although implemented platform does not support motion blur directly, the current design allows incorporating it in future without any changes to the platform core code. As noted in [VEACH1997 pp. 79-80], radiometric quantities can be defined by measuring the distribution of photons by a suitable geometric measure. The total radiant energy \(Q\) is obtained by counting energy of all photons in defined part of a phase space. Using the assumption with huge number of photons with small energy, we can assign to radiometric quantities continuous values instead of discrete ones, so \(Q\) may take any non-negative real value.
3.1.2 Radiometry

Here we define the radiometric quantities that are required by the light transport equation. Each of them is defined by measuring the distribution of energy with respect to some parameters.

(1) Power

Power of radiation is defined as energy per unit of time:

$$\Phi = \frac{dQ}{dt}.$$  

In some works ([PAULY1999], [SZIRMAY1999]) this quantity is called radiant flux (as energy radiated through a given boundary per unit of time over a given range of spectrum). In implemented system this quantity among others is used to describe total emission from object.

(2) Irradiance

Irradiance is defined as power per unit area, and is measured in watts per square meter:

$$E(x) = \frac{d\Phi(x)}{dA(x)}.$$  

Irradiance is always defined with respect to some point on surface \(x\), with defined normal \(N(x)\). Computations of radiance in the system must be done with caution, because surfaces may be opaque (i.e. blocking light from opposite side). Irradiance is used to describe how strongly is given surface illuminated.

(3) Radiance

Radiance is considered to be the most basic quantity in radiometry, and is defined as a power per area per solid angle (see Figure 3-1):

$$L(x, \omega) = \frac{d^2\Phi(x, \omega)}{dA^\perp(x)d\sigma(\omega)},$$  

where \(dA^\perp\) is projected area measure. Radiance is measured in watts per square meter per steradian. It may be rewritten to the more convenient expression

$$L(x, \omega) = \frac{d^2\Phi(x, \omega)}{\int_{\omega \circ N(x)}dA(x)d\sigma(\omega)} = \frac{d^2\Phi(x, \omega)}{dA(x)d\sigma^\perp(\omega)},$$

which uses standard area measure on surfaces. Light transport equations are based on radiance, which has the useful property – it is constant when light travels along straight lines in vacuum. To render an image it is enough to know the radiance on camera lens, however some techniques try to compute radiance everywhere. During scattering of photons it is important to distinguish between incident and exiting radiances. These quantities (defined on the same \(x\)) are often marked as \(L_i\) and \(L_o\). For more detailed discussion see [VEACH1997 pp. 83].

![Figure 3-1. The image explains radiance definition as the power emitted from surface \(dA\) in the cone with angular divergence \(d\omega\).](image)

(4) Intensity

Radiant intensity is defined as power per unit angle, and is measured in watts per steradian:

$$I(x, \omega) = \frac{d\Phi(x)}{d\sigma(\omega)}.$$
It is not used in strict physically based systems, but is crucial to describe emission from point-based light sources. As we want to be able to render scenes created in typical modeling programs, we have to allow placing point lights in the scene, despite the fact that they are not physically plausible. It is worth noticing that in physically correct systems emission from any particular point, and thus also this quantity, is always equal to zero.

(5) Power Density

Power density is similar to radiance, but is defined with respect to volume:

$$L_v(x, \omega) = \frac{d^3\Phi(x, \omega)}{dV(x)d\sigma(\omega)},$$

where $V(x)$ means volume measure. This quantity appears only in papers describing volumetric transport, and is used to describe the emission from volume (for example correctly modeled fire).

3.1.3 Rendering Equation

All the rendering algorithms solve the light transport problem. In 1986 James Kajiya [KAJIYA1986] first noticed that his theoretically correct algorithm solves equation described in his paper, and all currently available algorithms make simplifications of some kind, trading the accuracy of solution for speed. In this chapter we show how to derive this equation and then we point out simplifications made by historical algorithms.

In subsequent equations for simplicity we drop the wavelength parameter. From theoretical point of view it is not necessary to consider the wavelength, because different wavelengths may be computed in separate runs. However, in practice it is much too expensive, so our system does additional integration over wavelength while solving the original rendering equation. Whenever it is possible (when there is no dispersion) the generated path is used to transport several spectral samples at once. If there is a single coefficient describing the power of sample required and system has the whole spectrum instead, the perceived luminance of all spectral samples together is used, trading color quality (but not luminance) for speed of computations. The system still converges to the true solution, but convergence for luminance is somehow faster than for chrominance. More detailed discussion on this topic is presented in chapter 5.

First it is necessary to correctly describe the local scattering properties of surface. This may be seen as formalizing the reflectance functions such as Lambertian, which defines matte material or Phong, which can add glossy highlights. When light hits surface it creates irradiance equal to

$$dE = L_i(\omega_i)N(x)d\sigma(\omega_i),$$

where $dE$ is created irradiance, $L_i$ is incident radiance from direction $\omega_i$, $N$ is surface normal in point $x$ and $d\sigma$ is differential solid angle. It may be seen that change of observed radiance is proportional to change of irradiance. This fact allows defining reflectance function as a following constant of proportionality [PHARR+2004 pp. 250]:

$$f_r(x, \omega_i, \omega_o) = \frac{dL_o(\omega_o)}{dE} = \frac{dL_o(\omega_o)}{L_o(\omega_o)N(x)d\sigma(\omega)},$$

where $f_r$ is called Bidirectional Reflectance Distribution Function (BRDF), $L_o$ means observed surface brightness, and $E$ is surface irradiance. Such functions have two properties, namely energy conservation – integral over all possible output directions must be less than one:

$$\forall x \forall \omega_o \int f_r(x, \omega_i, \omega_o)\omega_o N(x)d\sigma(\omega_o) < 1,$$

and reciprocity (input and output direction could be swapped without changing the value of function). Notice that this function works only for reflection. In transmission reciprocity is generally not satisfied. To make all this work, often there is defined a Bidirectional Transmittance Distribution Function (BTDF), which is analogous to BRDF, but input and output directions lies on opposite sides of surface. Thus, to fully describe scattering properties of surface there are needed four functions (BRDF plus BTDF for each of two sides of surface). This is inconvenient and in many works is replaced by a single Bidirectional Scattering Distribution Function (BSDF), also used in the implemented system. This system requires only that function $f_r$ is energy preserving. It does not require the reciprocity. BRDF also has a probabilistic interpretation, for details see [SZIRMAY1999 pp. 11]. Thus, the Lambertian reflectance function has a constant value from range $[0, 1/\pi]$, and corrected Phong model [SZIRMAY1999 pp. 21-24] may be written as

$$f_r(x, \omega_i, \omega_o) = k_r \frac{n + 2}{2} \max(0, \cos^n \theta),$$
where \( k_s \) is constant from range \([0, 1/\pi]\), \( n \) is arbitrary non-negative real value which sets the strength of highlights, and \( \theta \) is the angle between mirror reflection direction and specified \( \omega_o \). Correction is necessary, since the original formulation [PHONG1975] is asymmetric and has energy balance difficulties, which is no problem when used as local illumination technique, but here is not physically plausible.

Now, we show how to derive the light transport equation. First we use the relationship obtained from BRDF definition:

\[
dL_o(x, \omega_o) = f_r(x, \omega_i, \omega_o) L_i(x, \omega_i) \omega \circ N(x) d\sigma(\omega).
\]

To compute output radiance at point \( x \) we integrate above expression over all possible directions:

\[
L_o(x, \omega_o) = \int f_r(x, \omega_i, \omega_o) L_i(x, \omega_i) \omega \circ N(x) d\sigma(\omega),
\]

which is so called local scattering equation. Radiance leaving any surface may be expressed as a sum of emitted radiance \( L_e \) and out-scattered radiance \( L_o \):

\[
L = L_e + L_o.
\]

Using ray casting operator \( T \) we may express an incident radiance \( L_i \) in terms of radiance leaving surfaces \( L \):

\[
L_i(x, \omega_i) = L(T(x', \omega_i), \omega_i),
\]

where \( T \) returns first ray intersection point or computation is terminated if ray escapes to infinity. Using this altogether we get

\[
L(x, \omega_o) = L_e(x, \omega_o) + \int \sum_{\text{lightsources}} L_i(T(x', \omega_i), \omega_i) f_r(x, \omega, \omega_o) \omega \circ N(x) d\sigma(\omega),
\]

which is a general equation solved by all physically based rendering systems.

All other rendering algorithms may be explained as simplifications made to the above equation. First, one-pass rasterization using commonly available libraries, such as OpenGL or DirectX computes only one scattering and handles only point light sources, so whole right hand expression takes form:

\[
\sum_{\text{lightsources}} L_i(T(x', \omega_i), \omega_i) f_r(x, \omega, \omega_o) \omega \circ N(x),
\]

which gives scattering functions very seriously limited – usually to Lambertian (constant) reflection or sometimes to Phong specular reflection. Such dramatic simplifications result in possibility of real time rendering, but at the cost of very poor lightning quality. Classic Whitted ray tracer handles multiple reflections, but only from ideal specular surfaces and for point light sources. This may be seen as replacing integral with a sum, without immediate substitution of emitted radiance. Cook’s Distributed Ray Tracing [COOK+1984] computes right hand integral, but only once for area light sources or twice for soft reflections. First algorithm that can correctly compute rendering equation without simplification is path tracing (see chapter 4.6.1).

### 3.1.4 Volumetric Extensions

Classic rendering equation cannot handle light transport in media which could participate in light behavior itself. It means the assumption of constant radiance for rays traveling between surfaces. This assumption is strictly correct only in vacuum, and is good approximation when rays travel on short distances in clean air. Classic light transport equation cannot simulate phenomena such as dusty or foggy atmosphere, emission form fire or large open environments when light paths may be hundreds of kilometers long.

In this chapter we show how to extend the light transport equation to handle these phenomena correctly. We do not give the full derivation here because it is mathematically challenging, and it may be found in [PAULY1999 pp.8-11] or [RUSHMEIER+1998]. We only show its form and comment how it is used in our system.

Participating media may affect ray by increasing or decreasing its radiance while it travels. Increase is due to in-scattering or emission, and decrease due to out-scattering or absorption. The whole participating medium may be described by defining three coefficients – absorption, emission and scattering in every point of 3D space. To make the definition complete in every point at which scattering coefficient is greater than zero there must be also given a phase function. It takes the same arguments and performs similar role as BRDF in classic equation. In current version of our platform we cannot handle emission from volume, so the extensions given here omit it.

Absorption coefficient \( \sigma_a \) is defined in the following way:

\[
\frac{dL(s)}{ds} = -\sigma_a L(s),
\]

where \( ds \) means infinitesimally short distance around some point, and \( L(s) \) is ray radiance near that point. Absorption coefficient, measured in \([1/m]\), is the probability density of absorption of photon per unit distance, and may be any non-negative real value. In vacuum, where there is no absorption, \( \sigma_a \) is equal to zero. The radiance decreasing is proportional to the current radiance and absorption coefficient.
The scattering coefficient $\sigma_s$ defines both in- and out-scattering. It is the probability density of scattering photon per unit distance. Out-scattering is similar to absorption. Using these two coefficients together there may be obtained an extinction coefficient, $\sigma_{ext} = \sigma_a + \sigma_s$, which describes the total attenuation of light while traveling along straight path:

$$\frac{dL(s)}{ds} = -\sigma_{ext}L(s).$$

The in-scattering is more complicated. To properly describe its effect it is necessary to define the phase function. Phase function has the form of $f_p(x, \omega_i, \omega_o) \rightarrow R^+$. It is the probability distribution function describing the probability density of ray scatters in point $x$ in direction $\omega_o$, provided that it comes from direction $\omega_i$. Then, the whole radiance increase due to in-scattering may be described by

$$\frac{dL(s)}{ds} = \frac{\sigma_s}{4\pi} \int f_p(x, \omega_i, \omega_o) L_i(x, \omega) d\sigma(\omega).$$

Above equation is similar to one that uses BRDF, but it is weighted by scattering coefficient, $\sigma_s$ and does not have a dot product with normal inside the integral. It may be seen as generalization of BRDF equation with surfaces described as appropriate Dirac delta distribution for scattering coefficient.

Putting all these expressions together, the total radiance change is:

$$\frac{dL(s)}{ds} = -\sigma_{ext}L(s) + \frac{\sigma_s}{4\pi} \int f_p(x, \omega_i, \omega_o) L_i(x, \omega) d\sigma(\omega).$$

To use in one equation both surfaces and volumes, ray casting operator $T$ has to be modified. It no longer finds the first intersection on nearest surface, but must properly integrate the radiance change with respect to ray lengths.

### 3.1.5 Integral over Paths

The light transport equation (in both forms, classic and volumetric) may be solved directly, or transformed into the integral over paths, and then solved as common integral. Integral over paths is theoretical basis of many algorithms, including bidirectional path tracing. The strict derivation of this integral may be found in [VEACH1997, pp. 222-225] or [PHARR+2004, pp. 738-740], and in [PAULY1999, pp. 11-14] for volumetric extensions.

Radiance reaching the camera may be expressed as an integral over space of all possible lengths (light reaching the camera directly, with one scattering, with two scatterings, etc.). In the surface-only version, the scattering events may occur only on surfaces, and the integral has the form:

$$I = \int L_e(x_0 \rightarrow x_1) G(x_0 \leftrightarrow x_1) W(x_0 \rightarrow x_1) dA(x_0) dA(x_1)$$

$$+ \int L_s(x_0 \rightarrow x_1) G(x_0 \leftrightarrow x_1) f_s(x_0 \rightarrow x_1 \rightarrow x_2) W(x_1 \rightarrow x_2) dA(x_0) dA(x_1) dA(x_2),$$

where $L_e$ is emission from particular point in particular direction, $W$ is camera sensitivity, $f$ is BRDF and $G$ is so-called geometric factor, defined as

$$G(x_i \leftrightarrow x_j) = \frac{\cos(\theta_i) \cos(\theta_j)}{||x_i - x_j||^2},$$

where cosines are measured on angle between vector $x_i$ to $x_j$ and surface normal at $x_i$ or $x_j$ respectively. The volumetric formulation is similar, but scattering can occur anywhere in space – inside volumes or on surfaces. When scattering is in volume, BRDF is replaced with phase function, and geometric factor does not have appropriate cosine term.

### 3.2 Numerical Integration of Rendering Equation

The purpose of numerical integration methods is to compute integral of the form

$$I = \int_{\Omega} f(x) d\mu(x),$$

where $f(x)$ is arbitrary function and $\mu(x)$ is measure, and the only possible operation on $f$ is evaluating it on single points. Function $f$ may be even not given in analytical form, and be quite complex to compute. The integral is usually approximated by
The numerical method has to choose quantity \( n \), all the points \( x_i \), at which the function \( f \) is evaluated and assign them appropriate weights. The rate of convergence of these method says how quick the approximate integral converges to true result if we split the original domain \( \Omega \) on \( n \) smaller domains \( \Omega' \) (using \( n \) times more sample points) and return as a result the sum of approximations for each \( \Omega' \). For more strict discussion about this topic the reader should see [PRESS1992]. To compare the rate of convergence there is used the notation known form computational complexity. For example using the quadrature with rate \( O(n^{-2}) \) means that to reduce the error twice we have to use four times more points.

The classic quadratures are based on polynomial interpolation of integrated function. The commonly known examples are Gauss-Legendre rule and Simpson rule. The Simpson rule achieves convergence rate of \( O(n^{-4}) \) provided that function is sufficiently smooth. These quadratures are very good for one dimensional smooth functions, but fail when function does not have these properties.

3.2.1 Convergence Problems

Standard one-dimensional quadratures are extended to multiple dimensions using tensor product rules, which result in integral estimation of form:

\[
\hat{I} \approx \sum_{i_1=1}^{n_1} \sum_{i_2=1}^{n_2} \ldots \sum_{i_d=1}^{n_d} w_{i_1} w_{i_2} \ldots w_{i_d} f(x_{i_1}, x_{i_2}, \ldots, x_{i_d}).
\]

This new rule has still the same convergence, but uses \( n^k \) point samples instead of \( n \), making the total convergence rate much worse. Also if the function \( f \) is discontinuous the best achievable convergence rate is only \( O(n^{-1}) \). This fact is easy to understand because sample points cannot say anything about location of discontinuity, it always fall between them. Putting this altogether, for multidimensional discontinuous function we can have at most the \( O(n^{-1/k}) \) convergence rate, where \( k \) is the dimensionality of space. Unfortunately, these functions are common in graphics.

3.2.2 Monte Carlo Quadratures

The solution of the problem of exponential complexity with respect to its dimension of tensor product quadratures is surprisingly simple. It is enough to take \( n \) points on random and assign them weights connected with probability of random particular point to achieve the convergence rate of \( O(n^{-1}) \) regardless of dimensionality. The MC quadrature has the form

\[
\hat{I} \approx \frac{1}{n} \sum_{i=1}^{n} \frac{1}{p(x_i)} f(x_i).
\]

Using a bit of statistics it may be shown that this estimation gives the correct result on average:

\[
E[\hat{I}] = E\left[ \frac{1}{N} \sum_{i=1}^{N} \frac{1}{p(x_i)} f(x_i) \right] = \frac{1}{N} \sum_{i=1}^{N} \int_{\Omega} \frac{f(x)}{p(x)} p(x) dx = \int_{\Omega} f(x) dx.
\]

The error of Monte Carlo estimator is the average difference between the actual (computed with finite number of samples) estimator value and is expected value, equal to true integral value. The probabilistic bound of error may be found using Czebyszew’s inequality

\[
Pr\left[ |F - E[F]| \geq \sqrt{V[F]} \delta \right] \leq \delta,
\]

where \( V[F] \) is variance of estimator equal to

\[
V[\hat{I}] = V\left[ \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)} \right] = \frac{1}{N^2} \sum_{i=1}^{N} V\left[ \frac{f(x_i)}{p(x_i)} \right] = \frac{1}{N^2} \sum_{i=1}^{N} V_0 = \frac{1}{N} V_0,
\]

where \( V_0 \) is variance of one sample. Above equation is correct provided that samples are independent. Inserting this result into Czebyszew’s inequality we get

\[
Pr\left[ |\hat{I} - I| \geq \sqrt{N} \frac{V_0}{\delta} \right] \leq \delta,
\]

thus for any \( \delta \) value the error decreases with rate \( O(n^{-1/3}) \).

This result may seem not to be obvious. There is achievable a better convergence rate than for carefully chosen samples of standard quadratures by putting samples in random way. It is less surprising after examination...
of tensor product rules. In these rules, when projected on one dimension, many samples are in fact in the same point, providing no additional information of function behavior in that dimension. Using carefully chosen sampling patterns the convergence rate may be slightly increased, resulting in quasi Monte Carlo methods. These methods are examined in more detail in chapter 3.4.

### 3.3 Probabilistic Techniques

Although Monte Carlo quadratures have reasonably good convergence rate, the variance of particular single sample (and then the error) may still be very high. Thorough decades there have been developed many methods addressed to decrease this variance; some of them are presented in this chapter.

#### 3.3.1 Stratified Sampling

Random samples tend to clump together leaving large portions of domain relatively empty. Clearly, this is not good and results in increased variance. The stratified sampling method is used to divide the whole integration domain $\Omega$ into smaller domains $\Omega_i$ with volume $v_i$, and draw $n_i$ samples in each $\Omega_i$. The total number of samples does not change, and samples are better distributed. Provided that no $n_i$ is equal to zero, the result is still correct, and provided that $n_i$ is proportional to relative volume $v_i$, the stratified sampling variance is never greater. To make this technique work efficiently, the true integral value in each domain $\Omega_i$ must be as different as possible from each other. When these values are equal, stratified sampling does not help at all (for derivation of these see [VEACH1997, pp. 50-51]).

Unfortunately stratified sampling has two major drawbacks. When this method has to work with high dimensional integrand, number of domains $\Omega_i$ becomes prohibitive. In such cases this technique is almost useless. Also the number of samples taken must be known in advance, which prevents from using this method in our system, where user decides when to stop computation.

#### 3.3.2 Russian Roulette and Splitting

These two techniques are designed to adaptively change sampling density without introducing bias. Using them it is possible to place more samples where integrand is large and few otherwise. The idea was first used in neutron transport algorithms in the 40’s and was adapted to computer graphics by J. Arvo and D. Kirk [ARVO+1990].

When the integrand is sum of many terms, Russian roulette allows skipping at random some of them in the following way:

$$F(x) = F_1(x) + F_2(x) + \ldots + F_n(x)$$

$$F_i = \begin{cases} 
\frac{1}{q_i} F_i', & \text{with probability } q_i \\
0, & \text{otherwise}
\end{cases}$$

It is easy to verify (for proof see [VEACH1997, pp. 67]) that replacing original estimators $F_i'$ with $F_i$ does not introduce bias. Russian roulette technique also works well when the number of terms is not finite, but the sum is well defined. In that case computations are done in recursive way:

$$F(x) = F_{rest}(x)$$

$$F_{rest}(x) = F_i(x) + F_{rest_{i+1}}(x)$$

and then the computed sum is:

$$S = \alpha_1 (F_1(x) + \alpha_2 (F_2(x) + \alpha_3 (\ldots (\alpha_n (F_n(x)) \ldots ))) = \sum_{i=1}^n \left( \prod_{j=1}^n \alpha_j \right) F_i(x),$$

where $\alpha_i$ is $1/q_i$ and $n$ is the number of last accounted term due to probabilistic termination. Strictly speaking, when there is necessary to store all the terms, the Russian roulette is not perfectly correct. While summation always ends after some time with probability one, there is always finite, greater than zero probability of obtaining arbitrarily long sequence and thus overflow any buffer. In practice it seems reasonable to set much larger buffer than average sequence size and use Russian roulette as acceleration technique only. For example when average probability $q_i$ is 0.9 and buffer size is 256, the probability of incorrect termination (too small result) is approximately $0.9^{256} \approx 2 \times 10^{-12}$. This allows to terminate otherwise infinite paths in Path Tracing. It should be noticed that while this technique does not introduce bias, it increases variance. Nevertheless when the skipping probabilities are chosen carefully Russian roulette method can still increase the computation effectiveness by reducing sample costs.
The splitting method works in opposite direction to the Russian roulette. When contribution of one of estimators is large it may be split to two estimators computed independently. This technique is rarely used in computer graphics, so we do not describe it in detail here. If one is interested how it works may consult [ARVO+1990].

### 3.3.3 Importance Sampling

This is by far the most important variance reduction technique used in computer graphics. It allows placing more samples where the integrand is large by carefully choosing sampling probability density. When this density is similar in shape to integrand, the variance is reduced. What is more, when these two functions are exactly proportional \( p(x) \propto f(x) \), the variance is zero. Unfortunately this requires the knowledge of the integral value to compute proportionality factor, so perfect importance sampling is not possible. The exact derivation of these results is presented in details in [PHARR+2004, pp. 673].

Nevertheless, if the function \( f(x) \) is equal to the product \( f_1(x)f_2(x) \cdots f_n(x) \), the variance may be reduced by using probability density that is equal or at least similar to product of some of the factors. However, the choice of pdf (probability distribution function) must be done at the algorithm design time, and if it is wrong, the importance-sampled results may be much worse than when constant pdf is used.

### 3.3.4 Multiple Importance Sampling

This technique was designed to increase reliability of common importance sampling of functions when the appropriate pdf cannot be chosen at design time. We do not describe the multiple importance sampling in details. The best reference on this method is in thesis of its inventor [VEACH1997]. The idea behind this method is to define more than one pdf (each of them is potentially good candidate for importance sampling) and let the algorithm chose the best one at runtime, when the actual shape of integrand is known. The algorithm does this by computing appropriate weights and returning the estimator as weighted sum of samples from these pdfs.

However, there is one difficulty that has large impact on our system design. The classical Monte Carlo techniques require only two methods – sampling points \( x_i \) with given probability \( p(x_i) \) and computing \( f(x_i) \). The multiple importance sampling requires additional operation – computing \( p(x) \) for arbitrary \( x \). The operation is ‘compute the hypothetical probability of returning given \( x \)’. This is usually more difficult than computing probability while sampling \( x_i \), since we have no knowledge of random choices necessary to select arbitrary value, as we have in sampling procedure. Fortunately the multiple importance sampling theory allows using approximation for this method, but imprecise results may decrease performance.

### 3.3.5 Biased Methods

The purpose of these methods is a further reduction of variance. Biased methods are often more efficient than unbiased ones. Unfortunately an error (bias) introduced by them is difficult to estimate, and thus results are not trustworthy. However, the incorrectness of these images may be much less visually distracting than high-frequency noise of correct unbiased methods. For quick reference on these methods see [JENSEN+2003].

The best known biased methods are adaptive sample placement with respect to variance and various kinds of interpolation. Also final output filtering is usually biased. The adaptive sampling is done by estimating the variance (not integrand, as in Splitting or Russian roulette) and put more samples where it is large. The bias is introduced by using the same samples to estimate variance and integral value. It manifests itself as extra patterns in resulting image, such as borders around light sources.

The best known interpolation schemes are irradiance caching and photon mapping. The bias is introduced by caching previously computed samples and accelerates computation of new ones by interpolation of previously computed values. The bias is visible as excessive blurring of final output.

### 3.3.6 Shift of Error Location

We use a special technique to shift error from numerical integration of product of two functions in a way that new error is less harmful.

Suppose we want to integrate an expression:

\[
I = \int_{a}^{b} f(x)v(x)dx
\]
We know that \( \forall x \in [a, b], f(x) \in [0, F_{\max}] \), and \( w(x) \) is a given non-negative function, so we can integrate it separately with arbitrary precision letting \( W = \int_a^b w(x) dx \). From these assumptions it is clear that independently of function \( f(x) \) \( I \) is never greater than \( W \cdot F_{\max} \). But if we compute a standard estimator of \( E \) in form

\[
I \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)w(x_i)}{p(x_i)}
\]

The above condition may not be achieved, unless a perfect importance sampling with respect to \( w(x) \) or whole product is used. Such error (when incorrectly \( I > W \cdot F_{\max} \)) may be harmful for applications which assume that it cannot happen. Note that simple clamping the result to \( W \cdot F_{\max} \) leads to inconsistent estimator – result will be too small on average (negative results do not happen, but occasionally the too large result will be clamped). However, the estimator computed by our different method:

\[
I \approx W \sum_{i=1}^{N} \frac{f(x_i)w(x_i)}{p(x_i)} / \sum_{i=1}^{N} \frac{w(x_i)}{p(x_i)}
\]

where samples in both sums are necessarily the same, satisfies above requirements.

Now we would like to show that \( I \) is always in interval \([0, W \cdot F_{\max}]\). Functions \( w(x) \) and \( f(x) \) are non-negative, so we have to check only if \( I \leq W \cdot F_{\max} \). To do this we use mathematical induction. When \( N = 1 \) the \( I = W f(x) \), so it satisfies requirements. Assuming that for arbitrary \( N \) we have

\[
I_N = W \frac{a}{b} < W \cdot F_{\max}
\]

we check the value for \( N+1 \) (with assigning \( F \) as \( F_{\max} \))

\[
I_{N+1} = W \frac{a + f(x) \cdot c}{b + c} = W \frac{a/b + F \cdot c/b}{1 + c/b} \leq W \frac{F + F \cdot c/b}{1 + c/b} = W F \frac{1 + c/b}{1 + c/b} = W F.
\]

To show that this estimator is consistent it can be written in the other form:

\[
I \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)w(x_i)}{p(x_i)} \left( W / \left( \frac{1}{N} \sum_{i=1}^{N} \frac{w(x_i)}{p(x_i)} \right) \right) \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)w(x_i)}{p(x_i)} \left( W / \hat{W} \right)
\]

Now it is clear that the right hand factor converges to one and left hand factor is standard estimator. Unfortunately correlation between these two estimators introduces bias. In fact this technique (when \( W = 1 \)) is sometimes used in filtering the raytracer output and the bias is not considered to be problematic [VEACH1997, pp. 308], and it is easier to use than perfect importance sampling with respect to \( w(x) \). Also forcing \( W \) to be equal to 1, we can automatically normalize the \( w(x) \) function, regardless of its actual value.

### 3.4 Quasi Monte Carlo

After all these results one may think if random numbers are absolutely necessary to make multidimensional integration work effectively. In fact virtually all implementations use random numbers in theory and some kind of pseudorandom number generator in practice. The answer is that samples sequence does not have to be strictly random, but it must satisfy some constraints. In this chapter we show how to use carefully designed pseudorandom sequences to boost the capabilities of Monte Carlo algorithms.

#### 3.4.1 Discrepancy

Discrepancy may be thought as the measure of how well samples are scattered thorough domain. The integration, which uses as sample locations points from sequence with low discrepancy, tends to converge faster than if it uses arbitrary random points. There is no one definition of this measure and derivations of integration errors are only estimations, but work well in practice.

The most common discrepancy measure is so called star discrepancy. It is defined for sequence of \( N \) points on unit box \( B \) as [PHARR+2004, pp. 316-318].
\[ D_N = \sup_{b \in B} \left| \frac{\# \{ x_i \in b \}}{N} - \lambda(b) \right|, \]

where the \( D \) is discrepancy of \( N \) points, \( B \) is unit box, \( b \) is arbitrary axis aligned box with one corner at the origin included in \( B \), \( \# \) is the number of sequence points that falls inside \( b \), and \( \lambda \) is measure of \( b \) (i.e. volume in 3D space) with assumption \( \lambda(B) = 1 \) (see Figure 3-2). Loosely speaking, the more proportional the \( \# \) of points to the measure of \( b \) is, the less discrepancy given sequence has.

![Figure 3-2. Discrepancy estimation.](image)

Using carefully designed sequences we can achieve discrepancies of order

\[
O\left( \frac{(\log N)^d}{N} \right)
\]

for sequences in \( d \)-dimensional spaces. Assuming that we have additional knowledge of how many samples will be generated (this must be given before first sample is computed) the dimensionality effect may be reduced by one – giving \( d-1 \) instead of \( d \). It is also worth noticing, as said in [SZIRMAY1999, pp. 46-47], that as expected regular (grid) sequences have extremely bad discrepancy for large number of dimensions – \( O(N^{-1/d}) \), but optimal for one dimensional integrals (\( O(1/N) \)). The purely random sequence has with probability one asymptotic discrepancy of

\[
O\left( \sqrt{\frac{\log \log N}{2N}} \right),
\]

which is independent of dimension and so much better than regular grid, but still worse than carefully designed deterministic sequence.

### 3.4.2 Convergence

The convergence of quadratures using low discrepancy sequences for founding sample points is justified using Koksma-Hlawka inequality. The math used in this derivation is a bit tricky, so this topic will not be covered here. One that is interested may consult [SZIRMAY1999, pp. 43-45]. There is one important point to be mentioned – all these results are invalid if function has an unknown discontinuity which is not parallel to one of axis. Unfortunately, these are common in computer graphics (i.e. silhouettes of objects). Nevertheless, despite the theoretical drawbacks, when used with care, these methods can improve performance in compare to classic Monte Carlo.

### 3.4.3 Techniques

In the implemented system quasi Monte Carlo sequence is used only to sample camera film. The choice for this purpose is two-dimensional Halton sequence because of it simplicity and effectiveness. Halton sequence is based on van der Corput radical inverse with different base in each dimension. These bases have to be relatively prime numbers in order to make sequence properly distributed over whole domain. It is advised to use \( n \) first primes for \( n \) dimensions in sequence. This gives the best results. Using larger primes makes this sequence ineffective, but still correct. In Figure 3-3 there is shown a sequence of first 1024 such points stretched to width four times greater than height. As can be seen on this figure, good low discrepancy sequences in contrast to uniform grid are effective even with distorting mappings. More detailed reference on these methods can be found in [PHARR+2004, pp. 318-322] or [SZIRMAY1999, pp. 47-48].

N-th point in radical inverse is computed by division of \( N \) by base till we have zero. All remainders are saved in form \( r_1 r_2 \ldots r_n \). These remainders are digits in system with given base. Next there is constructed the...
inversion of the form $0.r_n r_{n-1} \ldots r_1$, and the result is converted to standard real value. In this system there is used two-dimensional sequence, so most effective bases are 2 and 3. For the first one the algorithm is fast – requires only bit operations. For the 3-base when divisions are necessary there is used a technique proposed by A. Keller [KELLER1996].

![2D Halton sequence.](image)

### 3.4.4 Limitations

All quasi MC techniques can be difficult in finding errors when used without great care. First of all, the whole system is no more random and strong law of large numbers is no longer useable. The samples from different quasi random sources may fall into difficult to discover correlations, producing non-equal distribution. Even if not, there are visible artifacts resulting from these techniques, such as aliasing. Also, the samples may correlate with functions by which they are transformed, so quasi random importance sampling also must be taken with extreme caution.
4 Solution of the Rendering Equation

In this chapter we present some selected methods of how to solve rendering equation as well as describe their advantages and drawbacks, explaining why each of them can or cannot be used in general physically correct rendering system. We start by making general distinction between view dependent and view independent techniques. Next we summarize the most popular examples of both kinds – ray tracing, radiosity and mixed approaches as well. Finally we explain why these methods have some internal limitations and never can simulate certain effects.

4.1 View dependent vs. view independent

The view dependent techniques compute solution that is valid only for particular view (i.e. camera location). Output from these algorithms is usually an image that can be immediately displayed. The general advantage is that these techniques require very small amount of additional memory and are capable of rendering huge scenes without using external storage. Implemented system which runs on 32-bit machines uses algorithms that can be placed in this group and its limitations are about ~10M different figures on scene.

On the other hand, view independent methods compute the solution for all views simultaneously. The output is some kind of temporary data, which requires additional processing to be displayed. The most common representation is lighting maps for all scene primitives. Light maps are simple additional grayscale textures which mixed with normal textures give the appearance of illuminated polygons without using any lights. The advantage of this approach is that lighting maps can be very fast rendered by graphic hardware which results in real-time walkthroughs in global illuminated scenes. Unfortunately to achieve this scene must be static – any even smallest change to the scene makes entire solution invalid. What is more, the size of intermediate solution for non trivial scenes is unacceptably huge. To satisfy requirements of possible input scenes for this rendering platform, we have to make this solution six dimensional, which makes this method impossible to use in practice on such general scenes. The dimensions are three for space because scene contains not only flat figures but also participating media, two for direction – specular highlights and one for wavelength – exact simulation of color and dispersion.

4.2 Ray Tracing – Expansion Approach

4.2.1 Mathematical Formulation

The rendering equation presented in previous chapter can be solved in many different ways. This equation may be rewritten in operator form

\[ L = L_e + TL_e \]

where \( T \) is light transport operator. The operator contains both light scattering and transport. To assure that this equation can be solved it is enough to the norm \( ||T|| < 1 \). However, in scenes with refraction this may not be satisfied, but as pointed in [VEACH1997, pp. 112-114], there is less strict condition requiring only that there exists a natural \( k \) such that \( ||T^k|| < 1 \).

The classic ray tracing may be seen as recursively expanding the light transport operation, by substituting the whole left side of equation for unknown radiance \( L_e \), which leads to:

\[ L = L_e + TL_e + T^2 L_e + \ldots + T^n L_e + \ldots = \sum_{i=0}^{\infty} T^i L_e. \]

The analogical method may be used while tracing rays in natural direction of light – from sources to camera. There is also a method that allows trace rays in both directions together. For details see [VEACH1997, pp. 116-120].

4.2.2 Advantages and Limitations

The obvious advantage of that approach is low memory consumption and possibility of using exact representation of \( T \) instead of approximations of some kind. Almost all algorithms based on this technique have constant memory requirements, independent on scene size. This allows rendering huge scenes which are completely beyond the range of other techniques. The usage of original operator \( T \) allows making algorithm that is correct up to numerical errors. Such an algorithm while increasing rendering time can in theory reduce the
error to arbitrarily low level. In practice this allows reliable rendering of high contrast, sharp lighting features with fast changing indirect illumination, such as glossy highlights, caustics or perfect mirrors. What is more, the ray tracing approach with all of its read-only data structures is one of easiest techniques in graphics to parallelize, and thus resulting parallel programs have high speedup.

The advantage of constant memory complexity also leads to major ray tracing drawbacks. Since these methods store no data, they are usually slower than other ones that do. Moreover classic approaches have problems with some of possible arrangements of ideal specular surfaces on scenes. Details are discussed in chapter 4.5. It is also worth to notice that few of hybrid approaches actually can reliably deal with them.

### 4.3 Radiosity – Finite Elements Approach

#### 4.3.1 Mathematical Formulation

The finite elements methods solve rendering equation in iterative approach:

\[
L^{(1)} = L_e
\]

\[
L^{(n)} = L_e + \hat{T}L^{(n-1)}
\]

These methods cannot use the exact operator \( T \), instead they use its approximation, hence the name ‘finite elements’. The computed solution \( L \) is radiance of entire scene, not only the one that falls into camera. The approximation of transport operator is usually represented as a sparse matrix, and \( L \) is a vector, so the iteration is natural solution for finite element methods, however first approaches used Gaussian elimination here [GORAL1984].

#### 4.3.2 Advantages and Limitations

The most obvious application of these methods is when it is necessary to obtain illumination of the whole scene, not the single image. Provided that the scene is static this allows performing fast hardware accelerated walkthrough with computed information used as light maps. Also, when scene is relatively simple (comparing to ray traced ones) and with diffuse materials, radiosity generally converges faster to correct solution than other methods.

Unfortunately, the memory consumption for this algorithm is huge, especially in glossy environment. Using radiosity with non-diffuse scenes is in general very tricky and rarely gives good results. The approximation to operator \( T \) introduces some non-negligible error in each iteration step. The error is particularly large on fine geometric details and on glossy highlights in non-diffuse version of algorithm.

### 4.4 Other Methods

Methods presented here are tailored to have advantages of both previous approaches without their drawbacks. However, it is not easy to design such algorithm, and thus they are not ideal. These techniques tend to be faster than pure ray tracing and perform far better job than ordinary Radiosity with non-diffuse scenes, but usually suffer from high memory consumption, especially with highly complex scenes.

#### 4.4.1 Two-pass Techniques

These algorithms use both radiosity and ray-tracing steps interchangeably. In simplest version, the surface materials can be a linear combination of ideal diffuse and ideal specular component. Then the integral in light transport is broken into two parts, each computed with better suited method. This approach allows combining mirrors with scenes that are well suited for radiosity, but still cannot deal with surfaces that are neither diffuse nor specular, such as Phong-based metallic gloss.

#### 4.4.2 Irradiance Caching and Photon Mapping Algorithms

Both of these algorithms are essentially a ray tracing enhanced with methods of storing previously computed data. Hence they can be faster than classic ray tracing, but at the cost of large memory consumption and difficulties with parallelization.

The irradiance caching technique is based on the assumption that indirect illumination changes slowly over entire scene. It simply splits the rendering equation into two parts – direct illumination, which is computed in classic way, and indirect, which computation is boosted by cache. When the algorithm starts, the cache is empty. When algorithm computes the indirect illumination in given point, it first checks into cache if there is previously computed illumination data for nearby points. If so, the returned value is interpolation of some kind between them, otherwise the illumination is computed and its value is added to cache. The cache structure is
simple, and algorithm is well suited only for scenes with majority of diffuse or almost diffuse surfaces. The specular ones are taken out of caching and accounted with classic ray tracing.

Photon mapping [JENSEN2001] uses different technique for the similar purpose. The algorithm first emits many photons from light sources, and then traces them one by one into scene. When a photon hits non-specular surface, it is stored in a specialized structure, the so called photon map. After filling map with photons, the scene is rendered by modified version of ray casting. If a ray hits the surface, the radiance is computed by interpolating the contributions of nearby photons from the map. Major advantage of this method is ability to deal with all kinds of surfaces while storing previously computed data, what cannot be done with irradiance caching. Unfortunately, tracking the photons in advance, without any information about viewer position introduces a new problem. If a scene contains many invisible or barely visible light sources and relatively few with large contribution to image, the photon map is extremely inefficient. The work and storage space is wasted, leading to small amount of visible photons and excessively blurry image.

4.5 Computability of Ray Tracing

The mathematical formulation of ray tracing is, in general case, incomputable. The problem arises from using mathematical abstractions, such as ideal mirrors, pinhole cameras, point light sources, and so on. To make this effect happen, scene must contain these idealized objects. Image of any physically plausible scene is always computable, however, using for example materials that are almost specular in difficult configuration may cause huge errors regardless of used algorithm, hence there is impossible to design absolutely robust algorithm, which can cope with any scene.

In the rest of this chapter we explore this problem in more detail. First we show standard classification of light transport paths with extensions described in [VEACH1997, pp. 231-237]. After that we discuss how two particular, commonly used approaches can possibly deal with this difficulty.

4.5.1 Kinds of Light Paths

The paths are built by tracing rays between camera and any light source. Path vertexes are all scatterings that occur while building path. The sequence of vertexes contains additional two camera vertexes at the beginning and two light source ones at the end. Every vertex may be one of two kinds – specular or diffuse. Vertex is specular if it defines scattering event with BRDF described as Dirac delta distribution, otherwise it is diffuse. There are two important things to notice. First, not only ideal mirrors are specular, also ideal anisotropic surfaces, where possible output directions form one dimensional set, belongs to this class. Second, kind of vertex depends on actual scattering event, not the kind of material, which could be a linear combination of diffuse and specular component.

The additional vertexes that come from camera and light source have similar, intuitive meaning. The first one states if appropriate element has a finite area, i.e. point vs. area light source. The second describes the directional emission or sensitivity of component, i.e. idealized laser is specular, while light bulb is not. The example full path is shown in Figure 4-1. On this example caustics from area light source made by glass sphere on diffuse triangle are observed by pinhole camera (described in detail in 8.3.1).

![Figure 4-1. Example of full path.](image)

The light paths themselves can be classified by checking if sequences of vertexes satisfy certain conditions. This allows precisely checking the possibilities and limitations of any algorithm by checking which kinds of light paths it can account for, and which paths it skips.

4.5.2 Local Sampling Scheme

This technique is most widely used in ray tracing. The definition is simple – path can be built only by adding one vertex and edge at a time, or by concatenating two existing subpaths with edge. The example of non
local sampling is case when there are two points on different surfaces and it is checked a visibility between them using a nearby mirror. In that case this sampling can potentially add two edges at once.

As is stated in [VEACH1997, pp. 239], if the path could be sampled by local sampling algorithm, it must have two subsequent diffuse vertexes. Otherwise local algorithm cannot sample it at all, and the image is missing certain light effects. We do not show the strict proof here, which may be found in that work. Loosely speaking, two subsequent diffuse vertexes are enough to make path 'sampleable', since if they are near light source or camera, the sample ray can hit this object accidentally, otherwise, the algorithm may construct two sub-paths that each of them ends on the one of diffuse vertexes, and then make a valid connection between them. The necessity of this condition may be understood more clearly by examining the Figure 4-2. The example shows caustic on seabed made by point light source rendered indirectly through water surface by pinhole camera. In this case it is clear that neither camera nor light source can be hit accidentally since their area is zero, and the valid connection near specular vertex cannot be made. It is worth to notice that if camera was underwater, the connection could be made between camera and seabed, making the rendering with local path sampling valid.

![Figure 4-2. Difficult light path.](image)

### 4.5.3 Approximations

This class of methods does not generate full light transport paths. Instead, some kind of interpolation is used. For example, photon mapping algorithm, while shooting photons into scene, builds half completed paths. When the map is rendered, the camera sub-path is connected to many photons (and then light sub-paths) at once. However, the connection is imprecise (see Figure 4-3). This eliminates all the problems caused by computability of ray tracing and specular surfaces at the cost of introducing blur. The certain lighting features are never completely missing from image, but when scene has difficult configuration, they appear out of focus.

![Figure 4-3. Radiance interpolation.](image)

### 4.6 Ray Tracing Algorithms

In this section we describe in detail ray tracing based algorithms, which are implemented in our platform. First we explain a few variants of simple path tracing. Then we discuss the bidirectional path tracing, which is much more advanced and efficient than path tracing. Finally we present our new optimization that is addressed to accelerate this algorithm and which implementation is consistent with our framework.
4.6.1 Path Tracing

The path tracing method is based on generating light paths from camera towards light sources. The problem of infinite path lengths is solved by Russian roulette. The solution is strictly correct, since path does not tracing require the storing of full path, only the fixed number of vertices must be preserved.

The path is constructed incrementally. First, camera vertex and direction is chosen. Then, in loop, nearest intersection is found, and ray is scattered in random direction. The loop stops when either ray exits the scene and escapes to infinity or Russian roulette continuation test fails. The latter condition with probability one after some finite number of steps finish the loop. Notice that path generated in this way may contain more than one light source, and then be, in fact, multiple paths in one. This correlation may produce some unnecessary artifacts, but nevertheless sampling many lights at reduced cost is usually profitable. The path contribution to radiance is computed in the following way:

\[
L = L_e + \sum_{i=1}^{n} \left( \frac{f_{i1}}{q_{i1}p_{i1}} L_{e} + \frac{f_{i2}}{q_{i2}p_{i2}} \left( L_{e} + \frac{f_{i3}}{q_{i3}p_{i3}} \left( \ldots + \frac{f_{i}^{n-1}}{q_{i}p_{i}} L_{e} \right) \ldots \right) \right) = \sum_{i=1}^{n} \left( \prod_{j=1}^{i-1} \frac{f_{ij}}{q_{ij}p_{ij}} \right) L_{e},
\]

where \( L_e \) is emitted radiance, \( q \) is Russian roulette continuation probability, \( p \) is scattering probability measured with respect to projected solid angle, and \( f_i \) is BRDF. Notice that probability also can be measured with respect to ordinary solid angle. In the latter case \( f_i \) has to be scaled by cosine between surface normal and scattering direction.

There are a few variants of this method defining how to handle light sources. Simplest of them only check for emission whenever ray intersects something. This approach, however, is extremely inefficient when light sources in scene occupy small area and is incorrect with point light sources. The second version handles light sources in special way. Whenever ray hits something, there is randomized one extra point on one of light sources. Then, the point is connected with main path. If the visibility test does not fail, the light source contribution is added to path contribution. This method solves the small light sources problem, but creates another difficulty. When there are many invisible light sources this approach is probably worse than previous. Also when there is glossy surface near light source the resulting variance is high, and if the surface is specular algorithm fails. The third most advanced technique takes both approaches together and uses multiple importance sampling to combine these results. It is significantly more robust than both previous ones used in single at slight additional computation cost. This method is described in detail in [PHARR+2004, pp. 743-753]. The other variant (this with special handling of light sources) is described in [KAJIYA1986], but it has today only historical meaning.

4.6.2 Bidirectional Path Tracing

This algorithm is first working approach to solve asymmetry of current light transport algorithms. For example, if there is a specular surface next to light source, the simple path tracing fails utterly, while it is perfectly fine to place mirror next to pinhole camera. The particle tracing [ARVO+1990], in fact, does not solve this problem. While it enables efficient rendering of caustics from point light sources it prohibits placing secular surfaces next to camera, thus it only inverts this asymmetry.

The bidirectional path tracing, however, allows having strong points of both these methods together, and much more. Its only limitations come from local path sampling scheme. All paths that are possible to account for in this approach, are fine as well as in bidirectional path tracing, which cannot be said about simple path tracing and particle tracing.

The algorithm works by generating one subpath starting from camera and another subpath starting from the randomly selected light source. Then these paths are connected with deterministic step. In the original algorithm these paths are connected by every pair of vertexes. This immediately makes two drawbacks. First, to make algorithm effective, at least one path has to be stored, which makes Russian roulette not perfectly correct. Second, in highly reflective environment when these subpaths are long, the number of visibility tests that comes from connecting them, becomes impractically huge. In fact, the number is proportional to square of average subpath length.

The original implementation [VEACH1997] gives an optimization addressed to reduce this problem. This optimization is known as ‘efficiency optimized Russian roulette’, but as it requires the variance estimate of computing pixel is impossible to implement in our framework. This is caused by that in this framework frame buffer is write only, and it not necessarily contains any pixels, so we design another technique which is addressed to replace it (see next chapter). Veach’s thesis describes also a few other optimizations and tricks, from which some we have implemented, but we do not describe them here.
4.6.3 BDPT Optimization

This optimization is an important achievement in the project as it enables a substantial time saving. The speedup obtained for the optimized code is of a factor 1.7 when compared to non-optimized algorithm while it still produces output without noticeable worse quality. Similarly, when optimized algorithm is set to work with an equal amount of time than non-optimized one, the resulting image is significantly better. The variance of complex bidirectional sample can be split into variance introduced by paths of length one, two, ..., camera path length + light path length + 1. The whole variance is large whenever any of its components is, so the algorithm is most effective if these variances are roughly the same. However, simple BDPT version computes much more exactly the paths of length about half of the maximum value, than for maximal and one length. This is achieved at high cost of performing many visibility tests, and the variance reduction does not justify it because variance for extreme paths is higher still. The all possible paths for example case are shown in Figure 4-4.

The optimization is done as follows. First, samples with one subpath with zero length (that randomly intersect camera or light source) are estimated always because they are cheap. Second, samples with direct lighting and direct view (those with one vertex in light or camera subpath) are also always estimated, since on many scenes they can be highly effective. For the rest of samples there is computed exactly one sample for each path length. The number of potential candidates is calculated, and there is chosen one particular sample to be estimated using Russian roulette.

The algorithm works as follows (for example from Figure 4-4):

<table>
<thead>
<tr>
<th>length(edges)</th>
<th>possibilities</th>
<th>eye indexes</th>
<th>light indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>1-1</td>
<td>1-1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1-2</td>
<td>2-1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>1-3</td>
<td>3-1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2-4</td>
<td>3-1</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>3-5</td>
<td>3-1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>4-5</td>
<td>3-2</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>5-5</td>
<td>3-3</td>
</tr>
</tbody>
</table>

There is a loop over all possible lengths. Maximum number of possibilities for generating path with given length can be computed as \( \min (\text{length}-2, \text{total-length}+1, \text{eye_length}-1, \text{light_length}-1) \). The maximal index for both eye and light subpaths is \( \min (\text{length}-2, \text{eye_length}-1) \) or \( \min (\text{length}-2, \text{eye_length}-1) \), respectively. The algorithm randomizes one of the possibilities of generating the path with requested length, creates the path using maximal indexes and finally multiplies its original contribution by number of all possibilities for requested length, since only one path is actually computed instead all of them. After brief examination of above table it is clear that if shorter path is long, the time saving can be substantial, yet the variance increase is minimal, since the most noise is produced by shortest paths, which are not affected by this technique. The results of using this optimization are presented in chapter 10.2.
5 Spectral Representation

In this chapter we discuss the different color models and we describe this one which is used in implemented platform. More detailed reference on this topic is [DEVLIN2002] and [STONE2001]. The most important property that it must have is physical accuracy and ability to simulate various optical phenomena such as dispersion. The speed of computations is far less important than this, but nevertheless, the implementation uses some parallelization to achieve the best possible performance.

We start by discussion of the physics that is behind colors and physiology of human color sensitivity which is connected with artificial color models. Next we explain why RGB color model is good for displaying results and is completely wrong when used for computations. After this we propose a point-sampled spectrum model with multiple importance sampling that satisfies all requirements mentioned in the beginning of thesis. Then we explain how to make all the necessary conversions in order to display physically correct results on standard monitors and for use RGB-based textures with full spectral rendering. Next we discuss the basic operations such as scattering and dispersion handling, which may be seen as special case of scattering. Finally we describe the parallelization of computations which greatly reduces the speed penalty for using physically correct model.

5.1 Physics, Physiology and Colors

The color phenomenon is caused by spectral mixture of light that falls into the human visual system. The color depends on both composition of illumination and scattering properties of surface. To properly account for this effect the rendering equation and all other expressions from theoretical work, which are originally designed to operate on one real value as light intensity, have to be extended to operate on full spectrum. The spectrum is described as light intensity in function of wavelength. This function may be treated as an additional dimension of integration domain, making the resulting image three dimensional ($x \times y \times \lambda$).

However, human visual system cannot distinguish between arbitrary spectral light distributions. The space of colors recognizable by human observers contains only three independent values, hence the popularity of three components color models. To provide all necessary color information there is no need to store whole light distribution. It is enough to have three components obtained by integrating the actual distribution weighted by appropriate three functions. There are infinitely many spectral distributions that produce the same triple after integration. All of these are indistinguishable to human observer. These colors are called metamers. The resulting image instead of three dimensions has three two dimensional components.

There are some popular color models in computer graphics, any of them is well adapted to particular purposes that was designed for, and fails in others. The most common is RGB model. It is used to synthesize images from three independent light colors – red, green and blue. It is additive – (0, 0, 0) means black, and luminance increases with increase of any of the components. The model is used for displaying images and for storing them in some formats, such as bitmap. Unfortunately, this model also is commonly used in rendering, which is inappropriate (see chapter 5.2). The CMYK model (cyan-magenta-yellow-black) is widely used in printing. The fourth component is used for printing effectiveness. It is subtractive since inks acts as filters not lights and then (0, 0, 0, 0) means white (at least as white as printing paper is). There is one more popular model used in color choose dialog boxes since it is intuitive to most of users. The components are hue, saturation and luminance and the model is usually transformed to RGB after color selection.

All the previous models are strongly dependent of display hardware, and since its limitations cannot represent all of the colors that can be seen by human observers. There is a standard model – XYZ color space. It is independent of any hardware and can represent all the necessary colors. It was defined by the CIE (Comission Internationale de l'Eclairage) as three weighting functions to obtain x, y and z component. These components do not have any meaning of real colors as in RGB model. Instead the y value may be perceived as luminance and x and z as two chrominance values which define the color. It is worth to notice that all the weighting functions are strictly positive, which cannot be said about RGB weights. The colors which after integration give negative values in at least one of RGB components cannot be represented in this model and are perfectly fine in XYZ.

5.2 Why RGB is wrong

The use of RGB model does not have any physical justification. The model was designed to effectively display an image on monitor screen and nothing more. Usage of this model to rendering calculations is huge abuse. The light reflection computations under the assumption of elastic photon scattering are performed as a multiplication of spectrum that represents illumination and a spectrum describing reflectivity property of surface.
This multiplication actually must be performed on whole functions, not RGB triplets in order to get proper results. Consider, for example yellow sodium light which illuminates surface that reflects light only at red wavelengths. It is obvious, that result should be black, but in RGB model there is no yellow color. Yellow is represented as a mixture of green and red, which after multiplication gives red instead of black.

What is more the RGB model fails utterly when simulating dispersion. When light from light sources with almost parallel output rays is emitted on prism, the result will be three stripes – red, green and blue on the screen and rest remain dark (see Figure 5-1). Using full spectrum representation instead gives nice, smooth rainbow colors. Notice that rainbow may be obtained by RGB representation when necessary amount of divergence is used. If the light source angular distribution is conical and divergent enough, there will be smooth spectrum as result. On similar trick there is based a simple NVIDIA shader demo. The texture which is seen by glass is magnified by smoothing filter. The magnification is performed independently on each channel, and is greater on red than on blue. However, both of these methods do not have any physical basis an obviously are incorrect, but nevertheless look convincing.

![Figure 5-1. RGB dispersion problem.](image)

### 5.3 Full Spectrum

The only correct solution is to account for a whole (at least whole visible) spectrum. There are some techniques to achieve this purpose. The most common are linear combination of proper basis functions and point sampled continuous function. Effectiveness of the first one is strongly dependent on the actual functions that are chosen as basis and their match to original spectral distribution. Due to that this approach is still not completely correct and has problems with dispersion, it will not be described here. There is good reference on this method in [PEERCY1993]. The natural solution in Monte Carlo based rendering system is, despite that it at first seems less efficient, random point sampling. While it gives at first noisy image, this technique fast converges and after a few samples per pixel computed, gives near correct results. What is more, the technique is correct, since as the sample number increases, the error goes to zero, which is unachievable in previously mentioned approach.

#### 5.3.1 Point Sampling

In implemented platform we use the point sampling with fixed number (actually 28, chosen mainly due to hardware architecture) of point samples. In further parts it is called batch of samples. Good way of randomizing samples are importance sampling with respect to human eye sensitivity (in cameras) or with the light source spectral distribution (in light sources). Using both these techniques, the actual color values are then integrated using multiple importance sampling. This can handle smooth (like tungsten bulbs) light distributions and very narrow spikes (like neon bulbs) in the same scene. The spikes may be even represented as Dirac delta coefficients. The two greatest strengths of this technique are randomly selected wavelengths and well defined wavelength value for each sample element. The first one gives the correctness, since when more samples are computed, the more different wavelengths are explored, and due to law of great numbers the result converges to true value. It may be noticed that no technique with fixed number of spectral samples and fixed basis function set both can not be correct. The second allows simulating dispersion at the cost of additional color noise. We handle it by dropping all but one spectral sample when entering dispersive environment. This is done by randomly selecting wavelength to preserve, with constant probability. The probability should not take into account perceived luminance, because its effect was considered while selecting the wavelengths for batch of samples, and it is harmful to do it twice. All the samples except the selected one are then set to zero, and probability of the selected is multiplied by batch size (28). Then the wavelength parameter becomes well defined and computations are performed with usage of its actual value. However, this cannot be done with basis functions. Even if all but one coefficient are set to zero, the remaining one basis function still has some finite extent in wavelength space, which prevents from doing computations with explicit wavelength required.
5.3.2 Conversion to XYZ

This conversion is done while calculating output image. The full physically correct spectrum data is far too large to store them on disk, especially when animation is being rendered. However, when necessary, it is easy to switch off the transform and force output of raw spectrum data.

The conversion itself is straightforward. Computed spectrum is given as a set of point samples, and x, y and z weighting functions are given as arrays with 1nm precision. Graphs of these functions are in Figure 5-1. Integration is done by sum up all spectral samples multiplied by linearly interpolated appropriate weight. In the actual implementation the output results are swapped to YXZ. This allows using the same routine to compute the luminance of sample, which is the Y value.

5.3.3 XYZ to RGB transformation

This transformation is not part of core system, but we discuss it here in order to have complete description of color calculations done by platform as a whole. The core system output is XYZ data, since it is hardware independent standard and this transformation is done by external routine.

In most of related works there is no appropriate conversion for color results. Often there are described only mappings of grayscale results. Algorithm presented here is our proposal to solve this problem. It contains two independent parts. First is luminance mapping (in literature often called tone mapping), where we can use one of previously created algorithms, and second is the actual color conversion.

The luminance mapping is necessary because common display hardware has very limited contrast in comparison with contrasts that appear in nature. Usually these algorithms map the computed luminance from range $[0, \infty)$ to range $[0, 1]$, which is then quantized linearly (typically to 256 levels) and displayed.

The simplest solution is to clamp all the too large values to one. Unfortunately this often produces huge pure white patches on resulting image. The next common algorithm is to find brightest value, scale it to one, and scale the rest linearly. This approach also introduces problems. First, the actual image brightness is dependent on the brightest pixel, which is often effect of noise. Second, the dark regions of image typically are far too dark to distinguish details on them.

There is simple solution which is addressed to cope with those problems – nonlinear mapping. In our platform we use for this purpose function $y' = 1 - 2^{-\sigma y}$, where $y$ is computed luminance, $y'$ is mapped luminance and $\sigma$ is parameter – brightness scaling, which may be seen by photographic analogy as sensitivity of film.

In Figure 5-3 there is comparison between these techniques. Function values range from 0 to 1, and function domain is from 0 to infinity. The clamping is marked red, with long portion of domain with the same value which is clearly not good. The linear mapping is blue, which is potentially problematic, since bright pixels may drastically reduce its slope and thus make the rest of image impenetrably dark. With black are drawn five nonlinear functions. For them, the sensitivity adjustment can be performed independently of brightest image pixels.

In some advanced approaches there are used local contrast preserving techniques. They use different scaling factors in different parts of image, based on local average brightness. The result is large contrast on the whole image, but at the price of non-monotonic mapping. The artifacts producing by it are among others dark frames around light sources.
After mapping the luminance, the RGB output has to be produced. Due to limited color gamut of RGB model the exact result cannot be produced. In our algorithm we assume the following: mapped luminance ($y'$) must be preserved exactly in order to keep whole mapping monotonic, the hue also must be roughly preserved (it cannot be changed i.e. from red to blue while mapping) and only one parameter that might be modified by large amount is saturation. In this technique all problems that emerge while mapping results in fade-to-white effect. Especially, when sufficiently strong lighting is used every color becomes pure white – (1, 1, 1) in RGB model. This should not be seen as especially harmful effect, since it is similar to overexposure in photography. Careful adjustment of $\sigma$ parameter gives reasonably good results in almost all lighting conditions.

The algorithm requires color profile matrix. If matrix is not provided it uses default sRGB color profile [http://www.w3.org/Graphics/Color/sRGB.html]. To properly satisfy the above requirements it is also necessary the second row of inversed color profile. The row contains information how change of any of RGB components affects luminance ($y$). It is also important that all the elements of this row are non-negative and its sum is one. Otherwise the mapping result is unpredictable. If the determinant of profile matrix is non-zero the inversion is computed by algorithm. Otherwise appropriate row must be given. The algorithm itself contains the following steps:

1) scaling of all $xyz$ components:
   \[
   \text{scale} = \frac{y'}{y} \\
   x' = x \times \text{scale} \\
   z' = z \times \text{scale}
   \]

2) multiplying results by $XYZ \rightarrow$ RGB conversion matrix
   \[
   \begin{bmatrix}
   r \\
   g \\
   b \\
   \end{bmatrix}
   \times
   \begin{bmatrix}
   \ldots & \ldots & x' \\
   \ldots & \ldots & y' \\
   \ldots & \ldots & z' \\
   \end{bmatrix}
   \]

3) clamping, since the results are in general out of displayable $[0, 1]$ range
   \[
   \begin{bmatrix}
   r_{cl} \\
   g_{cl} \\
   b_{cl} \\
   \end{bmatrix}
   = \text{clamp}
   \begin{bmatrix}
   r \\
   g \\
   b \\
   \end{bmatrix}
   \]

4) computing the luminance change after clamping
   \[
   \begin{bmatrix}
   \Delta y \\
   \end{bmatrix}
   = \begin{bmatrix}
   y' \\
   \end{bmatrix}
   - \begin{bmatrix}
   \ldots & \ldots & \ldots \\
   \end{bmatrix}
   \times
   \begin{bmatrix}
   r_{cl} \\
   g_{cl} \\
   b_{cl} \\
   \end{bmatrix}
   \]
with assumption that the three marked components are nonnegative and is sum is equal to one

5) computing the maximum achievable luminance correction, two different cases

a) if \( \Delta y < 0 \), clamped to zero negative values, outside of device gamut

\[
\begin{bmatrix}
  c_r \\
  c_g \\
  c_b \\
\end{bmatrix} = \begin{bmatrix}
  r \\
  g \\
  b \\
\end{bmatrix}
\]

b) if \( \Delta y > 0 \), clamped to one too large values, unachievable saturation for required luminance

\[
\begin{bmatrix}
  c_r \\
  c_g \\
  c_b \\
\end{bmatrix} = \begin{bmatrix}
  1 \\
  1 - g \\
  1 - b \\
\end{bmatrix} \times \Delta y / c_{max}
\]

notice that in both cases \( |\Delta y| \leq c_{max} \), where \( c_{max} \) is the luminance of computed maximum correction and if these terms are equal the result is pure white, because of previous assumption

6) actual correction of luminance, with possible loss of saturation

\[
\begin{bmatrix}
  r \\
  g \\
  b \\
\end{bmatrix} = \begin{bmatrix}
  r_{cl} \\
  g_{cl} \\
  b_{cl} \\
\end{bmatrix} + \begin{bmatrix}
  c_r \\
  c_g \\
  c_b \\
\end{bmatrix} \times \Delta y / c_{max}
\]

The resulting spectrum is in expected range \([0, 1]\) up to numerical errors. However, it is necessary to perform one more clamping before the quantization in order to prevent integer overflow, but this last operation cannot change the luminance by large amount.

In Figure 5-4 and Figure 5-5 there is a comparison between Author’s new technique with luminance correction and a simple clamping proposed by literature (http://www.w3.org/Graphics/Color/sRGB.html). The colors are defined by blackbody radiation (1500K for red and 6600K for gray) with carefully chosen mapping sensitivity. Each subsequent color patch has the sensitivity set to value 1.25 times greater than previous. In the Figure 5-4 there is comparison of using the same technique for different colors. First row is computed with luminance correction – it exposes the fade-to-white effect. On the other hand, the second row has problems with lack of monotone. The red color an the left side of image appears slightly brighter than the gray one, but as we move to the right, with increasing sensitivity of cameras, the relative luminance is eventually swapped. Clearly this effect, when which part of image is brighter depends on camera sensitivity, is huge drawback of standard color mapping.

![Figure 5-4. Comparison between colors using the same mapping technique. The upper line is for the new technique developed in this project, the lower is from literature.](image)

In the Figure 5-5 these mappings are compared directly how they deal with saturated and gray color. The effect is roughly the same for white color, while when increasing saturation more and more luminance levels become unachievable with standard mapping technique.

![Figure 5-5. Comparison between mapping techniques using the same color. The upper halves of both rows are for the new technique, lower ones are from literature.](image)
5.3.4 Conversion from RGB

This conversion is the most troublesome part of the whole system, because it is only its part which is not well-defined. In fact, there is infinitely many possibilities of such conversions leading to totally different rendered images, and the platform must choose arbitrarily one of them, so the only solution to have totally predictable output images is not use RGB data at all. However, the RGB image maps are almost solely in use in modeling programs, so platform has to allow rendering it, and then provides guessed conversion, but it must be remembered that when using it, output may be completely incorrect.

Despite the fact that there are infinitely many correct solutions to this problem, it is easy to pick up the wrong one. Extreme caution must be taken while performing this conversion, since the possible ones for reflectance materials and for light sources form separate sets. That is, there exists no single such conversion that is correct for materials and light sources both. This is caused by white color problem. When we talk about idealized white material we understand the material as it reflects all light that fall onto it and therefore absorbs nothing. So the conversion for materials maps from RGB triplet (1, 1, 1) to spectrum which is constant and have the value of one. If this conversion is used for light source, the effect is oddly reddish illumination. This is since when we talk about white light we do not have in mind the light which constant spectrum but the daily Sun light. Its spectrum is complex result of Sun light spectrum and scattering in Earth atmosphere. In Figure 5-6 there is presented its actual relative power measured by CIE, which is far from constant. Human visual system adapts to these conditions, and perceives this as the neutral colorless light. Also when talking about light sources the term ‘white’ is abuse, because we have not ‘absolute white’ light (but we have ‘absolute white’ material). The light may be at most colorless, but we can never say that this light source is white and that not. This effect causes another trouble since the given RGB is in [0, 1] range and the light source can theoretically have arbitrarily large power.

The implemented in our platform conversion is defined only for materials. The plausible conversion means that resulting spectra satisfies some requirements. First, it is absolutely necessary that pure white (1,1,1) and pure black (0,0,0) are mapped to appropriate spectrum, since there are single good possibilities for both of them. The second is not to mess up colors. If it is rendered a textured surface illuminated with white light we want the output image to match the texture color up to proportionality constant. We state this more precise in the next paragraph. And the last one is that in most cases resulting spectrum should be smooth. The basis of conversion are three functions \( r(\lambda) \), \( g(\lambda) \) and \( b(\lambda) \). The functions are mostly arbitrary choice with requirement that they are non-negative, and for simplicity are spline based with few parameters to adjust to satisfy previously mentioned requirements. The actual spectral curve for blue component is computed as \( 1.0-(r(\lambda)+g(\lambda)) \), which guarantees that functions sum to one. The converted spectrum is \( s(\lambda) = R*r(\lambda) + G*g(\lambda) + B*b(\lambda) \), where \((R, G, B)\) is given color triplet, which satisfies the requirement of pure white and pure black mapping.

The conversion for light sources may be concocted using simple trick. To make it work properly it is enough to use a product of daylight illumination spectrum (given as standard CIE illuminant D65) and \((R,G,B)\) triplet converted in the same way as for materials. Thus the RGB data acts as modulator of white light. To make
the conversion complete we introduced the extra power parameter, which explicitly describes how bright the
light is, and therefore resolves the problem of white light relativity.

Now there can be defined missing spline functions. These are set to satisfy the special cases of
conversion – for pure red, green and blue. For red component it must be satisfied that

$$\text{mapping}(\alpha, 0, 0) \times \text{illuminant D65} \rightarrow XYZ \rightarrow RGB = (\beta, 0, 0).$$

Green and blue functions are defined in similar way. Thus the conversions to and from spectrum cannot change
color; it can change only intensity since white light is only relative term. Notice that the XYZ→RGB conversion
in this case does not contain tone mapping. One of possible solutions with the rgb matching curves is presented
in Figure 5-7. It is spline based, and contains 12 changeable basis functions for both red and green. The whole
spectrum is divided on 24 ones, but violet and red endings are fixed. The precision of the algorithm cannot be
perfect, and instead of ‘0’ after conversion appears slight error. The actual ‘zero’ value is no more than 3% of
\(\beta\), and is guaranteed to be non-negative. Also notice that while the error is sufficiently small the implied condition
with ratio \(\alpha/\beta\) has to be the same for each color is satisfied automatically by the fact that spectrums for all
primary colors sum to one.

![Figure 5-7. Spectrum for RGB colors and RGB matching curves.](image)

### 5.3.5 Basic Operations

The implementation of multiplication, addition, minimum, and so on operators are obvious, since it is
enough to perform appropriate calculation per component, as in RGB model. However, trickier when using full
spectrum is linear blending. For example, when it is necessary to define a material which is linear combination
of diffuse and specular components, with proportions varying with wavelength, the more complex algorithm
must be used.

Such blending is done with an extra spectrum, which must lie in interval \([0, 1]\) for each component. While sampling such material the luminance \((y)\) of the spectrum is used as selector, trading the luminance
variance to color one. It is necessary that computed \(y\) value lies in such interval, because selector has
probabilistic interpretation – with probability \(y\) is selected the first material, and \((1-y)\) the second one. Here it is
crucial that despite the spectrum sampling method error the probability is less than or equal to one, otherwise
system may crash with invalid floating point operation.
5.3.6 SIMD Parallelization

The SIMD CPU instruction set gives a simple and effective way of parallelizing spectral operations, and therefore making its speed penalty less painful. Almost every operator is parallelized that four components are computed together. However, this approach is ineffective when using function which is linearly interpolated data stored as an array in memory. Unfortunately currently available processors (year 2006) cannot made indirect addressing in parallel – it cannot be performed four memory dereferences with one instruction. To compute linear interpolation in such way we then need 2*4 memory accesses in loop, which effectively kills any speedup. However in spectrum implementation in such cases it is often computed three functions at once (conversions to XYZ and from RGB). In these methods the parallelization can be done in different manner – take only one spectral sample, but compute three functions with one run. After some tests we found that wasting one field in register is much more efficient than sequential memory access.
6 Framework

6.1 Overview

In this chapter we present the framework for rendering platform. The general requirement is to make it maximally elastic, and allow implementing algorithms mentioned in previous chapters within it. We start this chapter by discussing abstractness and efficiency considerations, and we explain the method used to make this platform easily extensible. After this there are shown auxiliary methods that are used within platform itself and also in plug-ins. Next we describe all the interfaces that are necessary to perform rendering. First there is sampling interface, which allows hiding the complexity of cameras and objects behind functions with clear mathematical meaning. Then there is discussed a new way to represent frame buffer independently of image storing details such as pixels. Finally we describe the interface which is implemented by rendering algorithm. The interface is extremely simple because all of tricky rendering details are handled by other classes, and allows focusing attention on mathematics and efficiency of algorithm.

The framework is not a reference to the code. We try here to explain general ideas, and we rather try to answer the question why something is implemented in given way instead of getting into details how it was implemented. To read the implementation description check the platform documentation in html format attached to this thesis.

6.1.1 Abstractness and Efficiency

The main issue of designing application is to well project the interfaces between components. When doing this, there often appears a question whether to hide the implementation behind a layer of abstraction or not. In the first case the whole application is more modular and easy to modify, but in latter one faster, since abstraction hurts performance. In our system we prefer clear, elegant design than speed, so almost whole platform is based on such interfaces. For example we do not assume that object is built from triangles, in fact we do not even assume that it contains a 2D figures. We also use frame buffer as storage addressed by two real values from interval [0, 1] without bothering if it has pixels inside or not. All of these give great flexibility in gaining almost any behavior at a little cost of changing existing code.

To make things clear we are not fanatics of strict object oriented methodology and we do not make nonsense decisions such as hiding behind abstract interfaces everything, especially when the implementation is simple. For example we always assume that quaternion contains four real values (hence it’s name) and vector three ones.

6.1.2 Extensions Support

The extension support is implemented using operating system Dynamic Link Library concept [RICHTER1999]. The plug-in has to implement array of methods, which are then loaded to platform address space using system mechanisms. To achieve language independence and portability we have not used object oriented code in interfaces. Instead, the implementation is exported as arrays of common functions, which are then mapped to methods using C++ inline functions mechanism. In some cases (described in few next chapters) these functions do something more than simple calling imported code, making implementation of plug-ins much easier.

6.2 Geometry

All the geometric issues can be solved solely by implementations of cameras and objects. However, it is much more convenient to have a predefined set of methods that satisfy most common cases. The functions included in platform allow easy positioning objects and camera in the scene. The operations that can be performed by them are translation, rotation and uniform scaling. This is in most cases enough freedom for camera and for objects which are rigid (i.e. parts of objects cannot move relative to each other).

Translation is performed in obvious way using vector addition. For the reasons of numerical stability, the rotations are stored in quaternions [DELOURA2000]. System deliberately does not allow using non-uniform scaling, since in most cases such operation is unnecessary and always cause troubles. For example, non-uniform scaling can change orientation of normals even if object is not rotated.

All the geometric transformation is handled transparently to plug-ins. The rendering algorithm works in global coordinate system, while camera and all objects perform their operations in local ones. This is done by
decorating every method call by appropriate transformation of output results and reverse transformation of input parameters by mentioned in previous chapter inline functions.

6.3 Spectral Representation

Implemented platform uses carefully designed full spectral representation of colors instead of RGB triplets. In chapter 5 we discuss why it is crucial to do this to correctly account for certain lighting effect. Among others these effects are dispersion in water or glass and correct simulation of colors that cannot be represented directly in RGB model, such as yellow which is turn to mixture of red and green. The platform has a dedicated class to perform spectral operations, and whenever a function requires a color as input or output, the reference to its instance is passed.

6.4 Sampling Interface

This is the main interface of the implemented platform. The rendering can be performed only by means of sampling. A camera and objects must both have methods that enable them to being sampled. Any other operation on them is not allowed, so sampling must be designed to be sufficient to enable use of ray tracing algorithms.

Sampling interface allows very elegant and concise way of specifying what could be done with cameras and objects. These sampling methods are simple to understand for one familiar with bidirectional path tracing theory, and yet general enough to express the full algorithm and much more in terms of them. Sampling of function may be understand as randomizing arguments of function \( y = f(x) \). The randomizing is most often done with unequal probability. In fact in effective routines probability strictly depends on shape of \( f(x) \). In sampling there can be distinguish three basic operations:

- randomize the argument \( x \) (for example direction of scattering) with specific probability density and return \( x, f(x) \) and probability; care must be taken because interface expects the probability to be expressed with respect to particular measure; as opposed to next method, this must be implemented in a way such that all three values (argument, value and probability) are exact;
- what probability would return previous method if it returns given argument \( x \), for example what is the probability of scattering ray in given direction \( \omega \); the result sometimes is difficult to compute exactly and can be approximated, but crude approximation hurts performance;
- simple evaluation of \( f(x) \), answering what is the value \( y \) for given \( x \); again this value has to be computed exactly.

Notice that \( x \) is a variable in some space \( \Omega \), typically more complex than simple real values. Again, \( y = f(x) \) is not necessarily a real number, and then \( f \) transfers \( x \in \Omega \rightarrow y \in \Omega \).

All the sampling routines can be grouped in three different categories:

- queries about point on surface – in objects these queries are connected with selecting an emitting point at random, and in cameras they select a point on lens; the argument \( x \) is here a point that lie on surface, the value \( y \) is spectrum, which describes spatial emission/sensitivity (irradiance) or with point light or pinhole cameras is delta distribution coefficient; probability is measured with respect to area; these functions are used by ray tracer to start a path;
- queries about transmission – such as ‘find nearest intersection of ray with object/lens’ or ‘randomize a point of interaction’ if object contains volumetric effects; the argument \( x \) is point in volume or point on surface (surface is from mathematical point of view a volume described with delta distribution), and the value \( y \) is spectrum that represents attenuation on ray path or zero if ray crosses wall; probability is measured with respect to distance; ray tracer uses them to incrementally construct a path and to check a visibility between points;
- queries about scattering – about direction in which ray scatters after collision with object, the same group of queries is also used to sample the directional properties of emission (objects) and sensitivity (camera); the argument \( x \) is direction vector and the value \( y \) is BSDF represented as spectrum, \( y \) may be zero if scattering actually absorb everything; probability is measured with solid angle for volumetric scattering, projected solid angle for surface scattering or is delta distribution for perfect specular surfaces; these functions are used only with respect to points acquired by one of two previous methods – either surface sampling or transmission; ray tracer uses them to incremental construct of paths.

Thus the basic interface contains nine functions altogether. However, to make the implementation more efficient, the query about transmission attenuation is divided into two separate functions. One tests ray with respect to surfaces only, thus returning a binary value if ray hits a surface or not, and another calculates volumetric attenuation completely ignoring surfaces. The interface also has to contain one more function that is not directly related to sampling – routine that enables object or camera to free temporary data, since most implementations require storing something between calls which generates points and queries about scattering.
The sampling interface also decorates every function call with transformation to suitable coordinate system, allowing transparently the rendering algorithm to work in global coordinate system and objects and camera to work in local ones. The sampling interface is purely abstract and it cannot be implemented directly. Instead there are separate camera and object interfaces that are lower in hierarchy and these can be implemented.

6.4.1 Objects

The object interface extends the sampling one with an additional function, which returns the total emission of it. This is simple emission power, which is measured in Watts. This value is used to guide random selection of object to emission if there is more than one emissive object.

6.4.2 Cameras

The camera interface has a simplified sampling one, since in this framework camera is transparently placed into environment. That is, camera only record light and if cameraman is to be seen on scene, it must be modeled explicitly as an object. Thus, camera never uses queries about \( f(x) \) and probability value for transmission. However, camera can be intersected ad random, but the ray is only recorded, and then is released, it never gets caught by camera.

The camera version of query about directional sensitivity as a side effect records a image sample that was done by tracing light into scene which light can eventually fall into camera (called later light image). The other methods have no side effects, and the sample that is generated by tracing ray from camera to scene is recorded by a separate call (called later camera image). Also the camera is responsible to make the light and camera images consistent, since irradiance computed by these two methods tends to differ on most camera models.

The camera does not store image directly, instead it cooperates with frame buffer. All the samples forming a light image camera converts to unit square (with centre typically in \((0.5, 0.5)\)) and with the associated spectrum send to store in frame buffer. To the data to store camera attach appropriate correction factor accounting for inconsistency between camera and light images. The samples created in opposite direction, when camera shoots rays into scene, are somehow easier to compute. The frame buffer coordinates are usually available as position on camera film, and no correction factor is necessary.

6.5 Frame Buffers

The task of frame buffer is to store partially rendered image. The method in which image is stored is not defined. Its interface is totally abstract and does not even assume using pixels. This way of thinking may be difficult for one used to graphic cards rasterization scheme, but in ray tracing the best idea is to break off with rasterization point of view completely. This gives many new possibilities, such as store all the rendering data in physically accurate format or use other pattern of samples to store data instead of rectangular grid. All these changes can be done transparently to the rest of system.

The frame buffer interface shows a unit rectangle addressed by two real values. The image in fact must be compound of two images which behaves in different way. One is a classic image generated by tracing rays from camera, and the second is particle-tracing style light image. In first one, the brightness is independent of sample density. The sample density affects only variance, and then image quality. On the other hand, in second one, the image brightness is dependent on both ray power and ray density. Thus the both images must be treated differently. To obtain luminance from camera image, frame buffer divides the local ray energy by local ray density, while in light image the weight is common. The weight cannot be deduced by frame buffer and thus is directly driven by rendering algorithm. To obtain the final local color value the frame buffer sums the camera image ratio and light image ratio. Not all rendering algorithms use both of these images, so for efficiency reasons these should be allocated only when necessary.

To provide convenient access to its data, frame buffer provides a specialized output interface that allows doing it in two ways. First, the data can be read in XYZ format value by value using standard unit square addressing. Second, the interface provides the functions to write the content of whole buffer into memory stream in implementation specific format and to merge the memory stream with buffer content. These functions are used to transmit rendered data through network.

The frame buffer functions can be accessed from multiple threads altogether. To avoid data inconsistency platform decorates each such call to specific method with appropriate critical section associated transparently with frame buffer, so its implementation does not have to cope with multiple threads.

6.6 Solvers

The solver is the engine of platform. It is responsible for rendering scene, and thus manages the work done by other components. Despite the playing role, it has a simple interface. The interface in fact has only one
major function – rendering given object with given camera. All the nuisances connected with scene representation, such as geometry, textures and so on are performed by object implementations and all the distributed application concerns are done by the platform itself.

However, the design and implementation of effective solver is challenging task. While almost all implementation details are solved in advance, there is still advanced mathematics to cope with. The platform is provided with implementation of path tracer and bidirectional path tracer. In most cases these algorithms are enough and there is no need to implement different ones, but if there is such need, the platform makes this task as easy as possible.
7 Parallelization and Network Protocol

7.1 Overview

In this chapter we discuss some of possible approaches to designing and implementing parallel applications. We start by summarizing the most common techniques to actually make the computations parallel. We describe here the Single Instruction Multiple Data CPU architectural feature, shared memory model for creating multithreaded applications and explicit message passing and Remote Procedure Call that enables distributing application on different machines. Next we discuss different aspects of dividing application into many as independent as possible computation processes. We describe here basics of PCAM methodology, discussing a ray tracer design with its support, and some partitioning schemes. We finish with description of some models that help to organize communication between created tasks. The mentioned models include among others an example ways of organizing communication when every thread can communicate with each other, and a master-slave model, in which one machine is selected and acts as a central part of communication system.

7.2 Techniques

7.2.1 SIMD

Single Instruction Multiple Data (SIMD) is by far the simplest parallelization method. It is done on CPU instruction level. At the execution time of one instruction, operation on whole pack of data is performed. One may consult [INTEL2002] for the detailed reference on Intel SIMD implementation. Usage of SIMD requires only careful data arrangement in memory to allow fast load, store and in-memory operations on whole packets. All the communication stuff required by other described further methods is unnecessary.

In implemented platform this technology is used as boost for full spectral representation of color. This representation operates on long arrays of spectral samples, and thus is ideal candidate to parallelizing in such way. In literature there is common a similar parallelization addressed to matrix and vector operation. Unfortunately Intel implementation is ineffective when using SIMD with double precision data – the pack size is only two numbers, and typical vector has three components, so speedup is very little. After quick analysis of required precision we found that faster single precision representation which is good for color calculations is completely unsuitable to geometric ones. Its precision is only 23 bits. For example if object is in 1024 units distance to scene origin, and output image has 1024 pixel width and the round-off error is more than three bits the figures may ‘float’ on screen by distance more than one pixel.

7.2.2 Shared Memory

Shared memory is parallelization technique for one machine with multiple CPUs. One process is subdivided into separate tasks, which are executed in different threads. All the threads still belongs to the process, but can be executed by different CPUs. This gives simple communication mechanism – all data may be transferred through memory, since every thread have the same address space. There is no need for specialized communication library, but care must be taken to synchronize memory accesses. When one thread starts to write into given data structure, the structure has to be locked to both read and write until the write is finished, otherwise the interactions between threads may be unpredictable and structure becomes inconsistent. The only disadvantage of this model is limited scalability. Every processor competes to memory access, and if the machine has too many of them, the memory bandwidth becomes bottleneck.

In implemented platform this method is used extensively. The number of computing threads is equal to the number of CPUs on machine. Synchronization is made by making scene description read-only, which do not need synchronization. All private temporary data is placed on threads’ private stacks. This can be done since ray tracing requires very little temporary storage. The only one data structure that requires synchronization is frame buffer, where the output of all rendering threads is written.

7.2.3 Message Passing

When the processes have to be executed on different machines, some more complex communication strategy must be used. One of them is explicit message passing. It may also be used to communication for processes on the same machine, but in this case is ineffective. Shared memory, when applicable, is clearly better.
When a process wants to communicate with another one, it cannot simply write required data into memory. Instead, it has to call a specialized library routine, which performs data transfer. The receiver of this message has to listen to communicates, which might be done with separate thread or blocking the main flow of execution. These routines may be blocking or non-blocking. Sending data is usually non-blocking, and receiving waits for the message to appear, but this is not a rule. Only the blocking communication routines may be used for synchronization.

In implemented platform this technique is not used directly. Instead using a library such as MPI or raw sockets, we have implemented a custom variation of RPC protocol using the latter technique. This allow to hide all the tricky socket details behind the layer of abstraction, and keep the efficiency and portability of socket based code, without any troubles caused by standard libraries.

### 7.2.4 Remote Procedure Call

The RPC technique is designed to make the communication in distributed application easier, and hide the tricky networking details behind the layer of abstraction. Instead of directly sending data through the network, the application calls appropriate procedures, which is more similar to multithreaded case. However, there is one difficult issue to solve – one process cannot directly access the data of another, so all the structures necessary to execute the procedure and output result have to be specified. This is done with interface definition language. Thus, such file contains all the procedures with parameters marked as input, output, dynamic array, and so on. Then, using data specified in interface definition, special compiler generates stubs which are used to transfer data and call from client to server.

In our platform we use solely above model. However, to achieve the best possible performance and portability we have to design and implement custom RPC library. We believe it is more general and can be used without the platform, so we distribute it as a separate tool. The documentation is also in separate document (Appendix B), but it is integral part of this thesis, as the whole library was primarily constructed to satisfy the requirements of this rendering platform.

### 7.3 Computations Partitioning

#### 7.3.1 PCAM Methodology

Partitioning Communication Agglomeration Mapping (PCAM) is the commonly used methodology for design of parallel applications. The name stands for sequence of activities that should be done to ensure good performance and scalability of such application. Typically good parallel algorithm is not a simple parallelized good sequential one, so the whole algorithm should be design as a parallel from scratch, and good methodology makes this much easier.

In the partitioning step, the whole computation should be partitioned onto as many as possible tasks, which can be performed in parallel. While doing this, one should not bother with the communication issues or processor availability. These are accounted for in future steps. Ray tracing algorithms, or at least these without storing precomputed data, are extremely good candidates to partition. In fact, after fast scene initialization step, every single rendering sample can be computed as a separate task. This gives the astonishing number of independent computations ranging from hundreds of thousands for low resolution previews to even few billions for final renderings. This gives great flexibility in subsequent steps. To make the application complete, there is necessary to make the two additional tasks, which in fact only store data. One keeps the whole scene description which should be read only, and another is the frame buffer, into which all computational tasks write. The frame buffer is write only.

In the communication step, there is checked how every task has or has not to communicate with each other while performing computations. On this step there is popular a task-channel model. In it a task represents a computational process, and a channel represents data transfer between two tasks. Channels not always have to be programmed explicitly, instead, for example, they may be done automatically by compiler for multiprocessor machines, but it is convenient to mark as a channel every data transfer between tasks. In ray tracing, the tasks read scene description, and write the output computed light intensity. So there are two channels connecting each computational task with both data tasks, and computational tasks do not communicate with each other.

In the agglomeration step, the tasks which create fine grain decomposition as a result of partitioning step are grouped together in a way which minimizes the communication overhead. In ray tracing we have huge freedom here, since each computational task has the statistically similar amount of work, and these tasks do not communicate at all. After some experiments we found that it is near to optimum make the number of agglomerated tasks proportional to its size. Thus the final number of agglomerated task is a square root of its initial number, multiplied by a constant which is agglomeration parameter. This gives a chance of good load balancing without problems with excessive number of tasks.
In the mapping step all tasks are assigned to processors. We prefer omit this step while designing application and instead perform it at runtime. This allows with a very slight computational overhead balance the computations over machines with different computational power. We also made a slight change to the tasks structure. We add a one extra task which manages all work – a task farm. The scene description task is duplicated on every machine because of frequent accesses to its data. Only tasks on the same machine can use a common scene description. Also to improve efficiency each machine has independent frame buffer task. The main managing task finally collect data from all frame buffers after the rendering was finished.

7.3.2 Domain Decomposition

The decomposition of ray tracing application onto little tasks is driven by data. Each task has then exactly the same code, but is executed on different input. One must be careful here to not misunderstand domain decomposition. Even if at first sight domain seems to be a whole image and thus each task copes with part of it this is not true. The actual domain is the set of samples, which are eventually mapped to image through complex hashing functions. In effect, every task contributes to whole image, not part of it. This gives good load balancing and possibility of interactive computations, which are described in subsequent chapters.

The different technique to decompose an application is functional decomposition. If there are found different tasks that can be execute in parallel, application can be decomposed by cutting code into parts and executing different code fragment on different machines. We not found this approach to be particularly useful in ray tracing, so we do not describe it here in detail.

7.3.3 Task Farm Load Balancing

The load balancing in our platform is done at runtime. We have implemented a task farm model. It is a central storage with all available tasks waiting for being processed, and every computational process take from here first unoccupied task when it finish previous work. We found that lack of scalability in ray tracing application is far less painful than it is commonly though about central task storage. The task is defined by only two 64-bit numbers – first sample to compute and sample number in task, so even if computational processes ask very frequently for new tasks, they cannot clog the master.

The actual number of tasks can be controlled by setting the constant mentioned in agglomeration step. Load balancing done in such way allows performing computations effectively on very inhomogeneous networks. This constant should be set empirically to match the particular computation environment once, and the square root computation method of final number of tasks copes with different rendering sizes.

We do not perform runtime load balancing on single machine with multiple CPUs. The task which such machine takes is further split onto equal parts, one for every thread. We assume here that every processor on such machine has roughly the same computational power. This assumption on one machine is usually correct, while it cannot be assumed that different machines have equal quality. The multiprocessor machine is then treated as exceptionally fast single worker. The central task dispatcher never investigates internal structure of computational processes.

7.3.4 Interactivity vs. Efficiency

The most effective way of rendering image in parallel is not to gather the final results from all machines until the last moment when all tasks have been done. Every send of results takes a huge amount of network bandwidth, and if worker does this more than once, it is wasteful form efficiency point of view.

However, doing computations in that way disables the option of preview the scene until it was finally rendered, which might take large amount of time. To solve this problem, we have added a switch that when is turned on forces the workers to sent results to central storage as soon as they are available. To avoid total clog of network, the minimal sendable amount of computed data is doubled after every send. This is good from practical point of view, since to increase the image quality by fairly visibly amount, the number of samples computed has to be effectively doubled.

7.4 Operation mode

The platform operates in server mode. It is started without any data and then waits for commands from user. These may be given directly form command line or by remote driver. We found that this is much more convenient than having application working in batch mode, as most MPI based ones does. The whole network of machines may be start up and configured once, then arbitrary number of rendering tasks may be performed. What is more, the working parameters can be monitored while computations are performed and after them. This greatly facilitates testing different algorithms and debugging, which is very difficult in batch mode.
7.5 Communication scheme

7.5.1 Master-Slave Model

The platform network protocol is implemented as master-slave model. The single application can be run in mode of master or slave. In the first case it contains the task farm and then controls all computations. The application is designed to contain two components – master and worker. There are compiled into one executable file. In master mode both of these are created since task farm uses little of computer power and is convenient to have possibility of performing computations on just only machine. In slave model only worker component is started, and such applications to be operational need to connect to remote master.

The user communicates with master application only. All instructions that set scene, computation parameters and so on, are then redirected to local worker. Thus when user starts computations, this local one worker is in correct state and all remote others need synchronization. The protocol before starts computation performs it transparently to user. After all workers have the appropriate state, the work is begun. Results are then sent back to the user indirectly through master when certain progress has been reached or only when the work is finished, depending on interactivity switch.

The application uses RPC protocol. Both master and slave create an RPC server, which listens to commands. Typically the worker is active one and starts communication with master. The master only can do it with connection to starting and stopping computations. It sends a signal to awake worker and abort signal, if user wants to abnormal terminate unfinished computations. This approach is very effective, since application sends the message when it knows it is necessary and no one has to repeatedly query for specific event. If, for example, master queries worker if it has finished computations, it may clog the network with empty messages or waste many time waiting too long, depending on how often queries are sent.

The connections are not kept alive. In fact every of them is broken when the procedure is finished. Instead, master keeps an array with all workers’ addresses, and a worker has the master’s address. When an application wants to call a remote procedure, it first establishes connection, then executes a remote procedure, and after that closes connection. We found that reaction time typically has little significance in compartment with time required to transmit long data, so this approach is effective since it less loads the network.

7.5.2 Master-Slave Interface Description

The interface is defined by set of remote procedures. Some of them are sent in synchronous mode and rest not. The asynchronicity means that procedure returns after all data is sent, without waiting for server response. This allows reusing the buffers without risk of deadlock. The procedures can be divided into two groups. In the larger one the master acts as server and worker as a client, and in the remaining one the roles are swapped. The procedures with which master can signal to worker are:

- **awake** – it awakes the worker; master sends this signal to every worker in its table at the beginning of computations;
- **abort** – it is send only when user chooses to abort the computations; when worker receives the signal it stops computations as fast as possible without making its state inconsistent or damaging data in frame buffer.

The above two functions are both asynchronous. In the subsequent procedures worker always acts as a client and in all but **confirm** master is server. This one can be executed on both master and worker. These procedures are:

- **whatToDo** – query to master about new activity to do, it is called when worker enters the network, is awakened or finishes previous task; the call is **synchronous** since the worker cannot done anything until receives the answer; the mechanism of it is somehow complicated and is described in next chapter;
- **recvSamples** – procedure is used to send computed tasks’ results to other application; currently send may be done only to master, but in future due to efficiency considerations the protocol is likely to be extended to support sends to other workers also; the call is **asynchronous**, and for efficiency reasons is executed on separate thread to make data transfer asynchronous also, which allows to perform computations while sending or receiving data;
- **confirm** – it is called when worker receives computed tasks and finishes merging it with its own work; the call is **asynchronous**;
- **farewell** – signal sent to master whenever worker is about to shutdown; this gives the possibility of gracefully remove the worker from network, without any harm, so any worker should be quitted by appropriate command, never by killing process; call is **asynchronous**;
- **error** – signal sent to master whenever worker encounters a catastrophic error; whenever this occurs, worker send error number, and since it cannot continue working, it goes to sleeping state; the call is **asynchronous**.
7.5.3 Activity Assignment Mechanism

Whenever worker does not have an assigned task and is not sleeping, it asks a whatToDo query. Depending on if there are unfinished computations, master gives it a new task or orders it to wait by putting it into sleeping state. The possible activities are:

- **sleep** – nothing to do, worker enters the idle state;
- **set** – order of load scene description, the command contains all necessary data to perform this task; it is executed only on remote workers and is ordered when computations are about to start or new worker enters the network with computations in progress; after setting scene description worker clears output buffer;
- **compute** – order of compute next task, task is defined by two 64-bit numbers – number of first sample to compute and number of samples in tasks; worker computes task without clearing output buffer;
- **send** – order of send results of finished tasks to specified worker or master, currently only send to master is available; after sends out the data worker is obliged to clear its buffer.

The worker behaves as finite state automaton. Its work is directed by whatToDo questions driven to master. Dependent on response worker can take an appropriate action and/or change state. The diagram depicting all possible state and transitions between them is in Figure 7-1. The transition may be also triggered by awake and abort signals.

![Workers' states and transitions diagram.](image)

Worker after starting its process is in starting state. It asks master whatToDo, and if there are no computations performed is ordered to sleep. Otherwise it is ordered to load appropriate scene description. The path that allows worker to immediately go to computing state is taken only for local worker that has just been awakened. The remote worker goes to computing state through setting data. When this activity lasts so long that all work has been done by others till it finishes, it is ordered to sleep.

When worker computes the given task (in computing state) it may be assigned next task or if it is remote it may be ordered to send already computed data to master or in future to other worker. If there is nothing to do the local worker may be forced to sleep. However, this transition is not possible for remote worker, since it must returns the results before it goes to sleep. After remote worker sends data it may get another task if there is still something to do or be forced to sleep otherwise.
The worker may be gracefully stopped only in sleeping state. There is also possible to abort unfinished computations without loosing so far rendered data, however, in this case the continuation of rendering is impossible. Killing worker, which can be done in any state, can lead to unpredictable behavior and crash of whole network.

7.6 Master-Slave Architecture

Both master and slave have a multithreaded architecture. The master server is executed on separate thread, still allowing driving application from command line. Every request is also processed on separate thread, which allows processing many workers’ requests simultaneously. If the application works in master mode the server can process both master and worker procedures, and in slave mode, only the worker procedures can be executed. In first case, both master and worker procedures are listened on common thread.

The worker has separate thread to execute scene data setting and computation of tasks. This thread repeatedly calls `whatToDo` in loop after every completion of task, until is eventually ordered to sleep. In this case it sleeps until is explicitly awaken by `awake` signal. When worker has to perform computations on multiprocessor machine, it creates additional threads for every CPU except the first, and splits task on equal parts. Then using all possible threads (including the thread used to create additional ones) executes the subdivided task. When main task finish its computations it waits for other threads, and when all of them finish work the task is done.

Worker can be stopped gracefully in sleeping state. The computations may also be aborted in computing state, but at the cost of loss all previously computed data by stopped worker, so exit command works only in this two states. Killing worker in any other state, especially in sending, can potentially lead to crash of whole network.

7.7 Remote driver

The computations are typically ordered by remote driver. This is a special application, which can set description of scene, begin rendering and finally receive an image through the API of platform available as RPC calls. The API and actual communication between all the applications and threads is presented in Figure 7-2. The synchronous calls are marked with solid lines, its responses with dashed lines. The asynchronous calls are marked with dotted lines. The case is for clarity for only two tasks and one remote worker. The master and local worker are in the same application, but on separate threads and therefore act independently. All sequence of actions on the same thread are deterministic and always have the order as on diagram, but sequences of actions on different threads can be changed in different runs. The sending data (by `recvSamples` procedure) from remote worker is performed on other thread, so its call actually might be swapped with `whatToDo` question.
The API that is presented on diagram is missing two extra functions. First, driver can execute getData to read scene description which was set before, and master can call an error signal with the error number if something went wrong and computations cannot be continued.

**7.8 Generating Animations**

The animation typically contains many frames, and each individual frame has much less strict quality requirements than still images. This makes parallelization of individual frames inappropriate. Better solution is to render many frames in parallel, each of them with sequential approach. The automatic generation of animation from script with above type of parallelism was developed and described by Michal Zmarz [ZMARZ2006].
8 Custom Extensions

8.1 Overview

The strength of this platform is in its extensibility. Almost any of algorithms used here can be replaced independently of others. What is more, algorithms can be changed at run time and platform may consist of any combination of elements while rendering. In this chapter we present predefined extensions that are available mostly in separate libraries attached to main platform. We describe here standard objects, cameras and frame buffers. We omit solvers, since they are direct implementation of ideas mentioned in theoretical framework.

8.2 Custom Objects

These objects are implementations of object framework addressed to make the usage of platform easier. The main concern in designing the necessary set of them was easy rendering of scenes exported from standard modeling programs such as 3DS. We have not implemented file converter for such particular program, but instead we provided a set of objects that realizes the common ideas used in them in much better, more physically accurate way.

8.2.1 Container

The container is empty object which allows putting other objects inside them. It is convenient to have a container, because solver implementation does not have to bother with multiple objects in scene. Instead, it always renders one object with one camera. This substantially simplifies its implementation. Thus platform always starts with empty container, which is unchangeable and irremovable. This container then is passed as an argument when computation starts.

It is possible to construct hierarchical structures by placing one container into another. Transformations of such container automatically transform its contents, making scene preparation much easier. Careful usage of containers and clones described in next chapter can lead to substantial memory savings.

8.2.2 Clone

The clone is an object that clones another one. It has no own implementation but redirects all calls to cloned one. Clone contains only own geometrical transformation, and thus can be situated in different place in different position and can be rescaled. Rest remains the same as in original. Careful usage of clones and possibly together with containers can lead to substantial memory savings on scene representation. For example an complex environmental scene may contains dozens of trees of several kinds, and each of tree can contain thousands of leaves of several kinds. Avoiding excessive replication of similar geometry allows to squeeze such scene in 32-bit system memory, which otherwise would be far beyond the range of any available system. Notice that here appears one of most major advantages of ray tracing. Such trick is impossible in all kinds of radiosity methods, since all objects are differently lit and therefore cannot be represented as one.

8.2.3 Light Sources

The light source set is addressed directly to replace the drastically simplified and therefore not physically plausible classic lighting concepts. This set contains three lights, based on different mechanisms. All of these lights can have blackbody or daylight basic emission. This basic emission optionally can next be modulated by RGB map.

(1) Spherical Lights

These lights are spheres with finite radius, which can be placed anywhere in scene. The basic emission from them is constant in every direction – the effect is similar to scattering from sphere with diffuse material. The spherical lights replaces point light ones. They give far better and more realistic results. The main advantage of them is the soft, naturally looking luminance and fuzzy shadows. One wanting to experiment with graphic cards artifacts may deliberately set the sphere radius to zero, and therefore force point light source. This gives awfully looking sharp shadows, and therefore is strongly discouraged.
(2) Spotlights

The spotlights emit from a rectangle or ellipsis. The emission is from one side, along with normal direction. The other side emits nothing. The divergence of emission can be controlled by Phong-like exponent, and therefore this kind of light is similar to scattering from glossy material. Area spotlights are addressed to replace point based spotlights. The advantages of using finite area are the same as in spherical lights. Phong-like control of emission gives one more advantage. It removes the artificial boundary between the two emission defining angles known from DirectX or OpenGL, and gives the smooth falloff of radiance when angle becomes more perpendicular to surface normal. The limitation that light cannot emit backward (similar to placing limit on divergence angle on $\pi$ radians) is not a problem, since such light has very little use in practice. Similarly, forcing emission area to be zero, creates a point spotlight, but making this is highly discouraged.

(3) Surrounding Light

This light is modeled as an infinitely large sphere which surrounds the entire scene. There is strongly recommended to use at most only one surrounding light in the scene, otherwise rendering result is unpredictable. Because of model assumptions, the exact point of emission of light source is impossible to represent directly (since it is in infinity) and the only important thing is direction in which the point is visible from scene. Therefore this kind of light is far better alternative to simple directional light. When surrounding light is defined as daylight emitter modulated with sky map, it can be used easily to model sunlight on open scenes. This model removes the common artifact of sharp shadow. For example, if scene contains a plane high in the sky and the ground level, when naive shadow casting algorithm is used, the plane still cast sharp shadow on ground. On the other hand, with the better surrounding light, the shadow automatically becomes more and fuzzier when plane takes height, without any additional quirks and tricks.

The implementation of this kind of light has to solve one difficulty. Since light’s infinite size and finite irradiance, the power emitted from it is also infinite. Thus the simple algorithm for generating light samples with probability of choosing light proportional to its relative power fails. There is approach to solve this problem which estimates effective quantity by computing area of bounding volume for entire scene [PHARR+2004, pp. 615], but in our opinion this is likely to fail also. Consider, for example, large environmental set lit mainly by evening sky and the cameraman next to bonfire. Despite the fact that on visible illumination largest influence has the bonfire, there are minority of samples generated on it due to total size of map. We use different assumptions here. First, the infinite light source is only one. Second, if scene contains local light sources, they are probably important to rendering result. Thus infinite light is treated as special case and always is given 50% samples. The remaining half is then distributed among local ones by standard algorithm. This approach cannot give the ultra fast convergence, but is fairly robust and cannot fail completely.

8.2.4 Standard Model

This object is designed to allow easy usage of triangle based models with global illumination. It can parse the Wavefront obj file, and use the read information to construct kd-tree based geometry and materials set compatible with rest of platform, which allows rendering of this model. Due to complexity of algorithms used here, this object is described in detail in next chapter.

8.3 Custom Cameras

In this chapter we describe implemented in our platform camera models. We implemented the simplest models which allow simulating important real-lens effect such as depth of field or chromatic aberration. These implementations have nothing in common with graphic cards’ ones. For example there are no clipping planes and projection matrices. While these models are not capable of simulating more advanced phenomena of real lens systems it is easy to make such a camera within our framework. The good reference on this topic is [KOLB+1995]. All the cameras in fact are capable of generating two images. One when rays are traced from camera into scene and second when light falls into camera. Unfortunately these techniques are inconsistent. The first one gives constant luminance over entire film, and the second, as in real camera, does not, so one of them has to be scaled. In our implementations we prefer the idealized cameras, so we scale the second one to give constant luminance over entire film in both techniques.

8.3.1 Pinhole Camera

This is the simplest model camera with lens with zero area. Such camera can be parameterized by only width and height of the film and a distance from film to the point lens. It may be noticed that scaling all of the three values by arbitrary positive amount do not change camera behavior, so the actual implementation always use 1.0 as this distance. View angle in horizontal direction is then directly bounded with film width, and a default value of 2.0 makes the angle to be $\pi/4$ radians. The geometry of this camera is shown in Figure 8-1.
The camera is placed in global coordinate system having lens in origin, up aligned with Y axis and looks straight in Z direction. It generates rays that start in origin. Ray direction is selected by randomizing a point on film, and then founding direction from selected point to the lens. The camera cannot be hit accidentally, since it has a zero area lens. When there is a ray traced in opposite direction (from scene to camera) camera checks if it falls onto image film. To assure that these two methods makes images with consistently computed irradiance, the incoming ray, if it hits film, is scaled by

$$\frac{\left( x^2 + y^2 + 1 \right)^{\frac{1}{2}}}{w \cdot h}$$

where \( x \) and \( y \) are points coordinates on film and \( w \) and \( h \) are film width and height respectively (see the appendix for derivation of this expression).

The camera model has also an extension, which enables to effectively rendering stereovision images. It allows shifting lens by some distance in x direction, and therefore makes the asymmetric image. If two asymmetric images are concoct into a stereovision one, there are some data on left side of one image and on right side of another which do not have corresponding elements of the second picture, and therefore has to be thrown out, which leads to wasteful rendering. The exact meaning of the offset parameter is explained in Figure 8-1.

![Figure 8-1. Pinhole camera geometry.](image)

The sample position is not implemented as a random value. Experiments show that the image could be much better quality if the low discrepancy sequence is used here. We use in our implementation Halton sequence with base 2 in horizontal direction and 3 in vertical one. The sample location is generated using sample number, which assures that every task in parallel mode do not compute the same data.

### 8.3.2 Thin Lens Camera

This is the extension to pinhole camera model which allows rendering depth of field effect. This model uses, known from physics lessons, infinitely thin ideal concentrating lens. This lens is simplified model of real lenses. It cannot simulate the effects of spherical and chromatic aberration. The objects that are seen in focus form ideal plane in some distance to camera. The distance is given by the equation:

$$\frac{1}{f} = \frac{1}{o} + \frac{1}{o'}$$

where \( f \) is the distance from lens focus to its centre, and \( o \) and \( o' \) are the distances of point and its image, respectively. Direct using of these parameters is however inconvenient, so the implementation uses this equation in form allowing to enter directly the \( o \) parameter. The \( o' \) value (distance to film) is forced to be 1.0, since thin lens camera extends pinhole one. The implementation requires a one more parameter – the lens diameter. Loosely speaking, it controls the strength of depth of field effect.

The implementation comparing to pinhole model has some additional functionality. The ray starting point is randomized on circular camera lens with given diameter. The outgoing ray direction is computed with the following expression:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} f_x \\ f_y \\ 1 \end{bmatrix} \cdot o + \begin{bmatrix} l_x \\ l_y \\ 0 \end{bmatrix},$$

where the \( f_x \) and \( f_y \) (and 1) means the randomized location on a film plane, \( l_x, l_y \) (and 0) is the randomized location on lens and \( o \) is the distance to focal plane. The output direction is of course normalized. One may check correctness of this expression by using diagrams for concentrating lens and some simple trigonometric manipulations. The incoming ray is mapped to film plane by:
\[ \begin{bmatrix} u \\ v \end{bmatrix} = \left[ \begin{bmatrix} \frac{r_y - l_y}{r_z} \cdot o/r_z - \frac{l_z}{l_z} \\ \frac{r_y - l_y}{r_z} \cdot o/r_z - \frac{l_z}{l_z} \end{bmatrix} \begin{bmatrix} f_w \cdot o \\ f_h \cdot o \end{bmatrix} + \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \right], \]

where \( r_x, r_y \) and \( r_z \) are incoming ray direction, \( f_w \) and \( f_h \) are film width and height and rest of symbols have the same meaning as in previous equation. To check correction of this expression one may evaluate \( f_x \) and \( f_y \) and check if they are equal to \( u \) and \( v \), respectively.

There are two interesting limit cases in thin lens camera. First, if lens diameter goes to zero the model becomes more and more similar to pinhole camera (to check this substitute zero for \( l_x \) and \( l_y \) in the equations). When the focal plane is moved further from camera, the camera becomes a finite aperture without lens camera.

### 8.3.3 Chromatic Aberration on TLC

This is extension to Thin Lens Camera model, which allows independently setting the focal plane for before-violet (360nm) and after-red (830nm) wavelengths. The plane for actual wavelength is computed by linear interpolation. The mechanism is simple. Camera rays are generated for the \( \lambda = (360nm + 830nm)/2 \) wavelength (green light). Spectral sample just before recording in frame buffer is split into different wavelength components. The location \((u, v)\) on frame buffer for each wavelength is computed by using inverted equation from Thin Lens Camera model (there is known a ray direction – the same as computed, and focal plane – from wavelength, and the location on plane is searched for).

### 8.3.4 Appendix. The derivation of irradiance correction

This derivation is based on assumption that camera image is reference one, and light image is to be corrected to be consisted with it. The derivation is performed using standard light transport framework and treating camera as part of image with introduction of two extra vertexes (see Figure 8-2).

![Figure 8-2. Light image correction.](image)

The rays traced from \((x, y, 1)\) vertex to \((0, 0, 0)\) are generated with weight one, but as can be derived from light transport theory, effective irradiance on camera film generated by rays that fall into camera is ray power weighted by:

\[
\cos \varphi \left[ 1 \begin{bmatrix} x \end{bmatrix} + 1 \begin{bmatrix} y \end{bmatrix} \right] \cdot \begin{bmatrix} 0 \end{bmatrix} = \begin{bmatrix} x \end{bmatrix} + 1 \begin{bmatrix} y \end{bmatrix} \cdot \begin{bmatrix} 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \end{bmatrix} = \left( x^2 + y^2 + 1 \right)^{3/2}.
\]

To make both images consistent on first step of mapping, light rays have to be divided by above expression. Then the camera film is normalized to unit square, and all light samples are squeezed within it. To account for this effect the light rays must be divided by film area, which altogether leads to expected final scaling factor:

\[
\left( x^2 + y^2 + 1 \right)^{3/2} \div W \cdot H.
\]

If all rays that form light image are scaled by this factor, the light image becomes consistent with camera image, and then bidirectional techniques can be used without any troubles.

### 8.4 Custom Frame Buffers

Due to abstractness of frame buffer interface there is great flexibility of designing the implementation. We have provided two different frame buffers suited to solve particular tasks. First implementation is the classic storage of data in XYZ floating point format. The second implementation works in completely different way. It stores raw data that comes from rendering on disk. In this chapter we describe these two approaches in detail.

#### 8.4.1 Sample Grid Buffer

The samples that come to store in buffer are converted to XYZ values. Storing samples is performed not identical for light and camera images. The incoming sample \((u, v)\) coordinates are treated as a centre of
resampling filter. Then all grid samples in buffer which are affected by filter range, are updated. For camera image, the x, y and z components are increased by multiplication of filter weight in given grid sample coordinates and sample value. The grid sample weight is updated by only filter weight. The operation of storing samples for one point in grid can be written formally by expression:

\[
S_j = \frac{\sum_{i=1}^{N} w(u_i, v_i) s_i}{\sum_{i=1}^{N} w(u_i, v_i)},
\]

where \(S_j\) is j-th grid sample, \(s_i\) is incoming sample with coordinates \(u_i, v_i\), and \(w\) is filter weight for given coordinates. This operation introduces slight bias into computations, but error seems to be smaller than in unbiased one.

The update for light image is performed in two steps. First, it is created a small auxiliary array with size equal to range of filter. In this array there are stored points each associated with grid sample from main array. The auxiliary points are set to sample value multiplied by filter weight in given point. Then auxiliary array is normalized that all filter weights sum to one, and finally its content is added to main array.

### 8.4.2 Raw Storage Buffer

The advantage of this approach is absolutely lossless storage and usage of fixed and small size memory buffer. However, this is done at the cost of generating huge files on disk and necessity of usage external tools to perform image processing necessary to display it. Also, the buffer cannot sent data through network because of the huge file size (potentially gigabytes), and then its merge and overwrite methods necessary to fit it in the interface do nothing. Thus in parallel mode the data after computations is scattered on different machines and has to be collected manually and then may be concatenated to one file. The representation of data in these files is built from records, one per sample. The record contain sample number, \((u, v)\) coordinates of sample as two fixed point numbers from range \([0, 1]\), the flag whether the sample belongs to light or camera image and a spectrum data as a series of pairs \((P, \lambda)\), where \(P\) is power at wavelength \(\lambda\). The camera samples weights are accounted for on the fly, and the light image common weight is placed in the header. The two files are concatenated by adding light weights and copying all sample records.

This buffer allows also storage of standard XYZ data in sequential file. This functionality is used for rendering huge images, so data size is important. Thus the sample record has only \((u, v)\) coordinates and XYZ components. Storing data in sequential format despite the larger size has an obvious advantage. The file to which data is appended can be gigabytes large, so rewriting random portions of it is slow. The approach used here allows using small, fixed size write memory buffer, and when buffer is full, send the whole content of it in one disk write operation.

These functionality is useful to generate images other than classic 8-bit RGB ones with extremely large resolution and quality. With XYZ only flag it enables rendering of huge images that otherwise cannot fit into memory. It is also convenient to debugging application, since in raw storage format each sample data is explicitly available and can easily be checked if is correct.
9 Standard Object Model

9.1 Overview

The standard object model is designed to provide easy rendering of models created by standard modeling programs such as 3DS. The core of model design is constructed from three layers. Each layer has its own particular interface. The most basic are figures. Figures define the model geometry. Materials can be placed on figures. Materials describe how the light interacts with surfaces. The definition of material is not complete – it shows only if the surface is matte, glossy, transparent, and so on. There is still space to define the color of particular material. The colors and also something more is defined by color and scalar sources. Color sources differs in return values – color ones return full spectrum and scalar ones single real value.

To the figures additionally may be attached bump maps, with meaning similar as in classic libraries. Each figure can also have an emission. The emission defined in this way is in opposite to OpenGL or DirectX fully functional and treated as an ordinary light source.

The standard model beside the flat figures can also define volume regions filled with fog, smoke, dust or so one. This gives much versatility to model and therefore rendered images are more convincing than ones made only from triangles.

9.2 Figures

Figures are geometric primitives which describe model shape. The classic figures used in graphics are 2D surfaces suspended in 3D environment, but the interface allows easy to write extension to support for example CSG. The figure interface deals with ray-figure intersections, bounding boxes and differential geometry. It contains a few functions, each of them described in detail in one of subsequent chapters. Intersection and bounding is discussed in here and in kd-tree chapter, and the rest when describing embedded light sources.

9.2.1 Vertexes, Normals and Texture Coordinates

Classic 2D figures uses concept of vertexes. This allows sharing data between them and therefore greatly reduces memory consumption. For example, a triangle and quadrilateral has three or four vertexes respectively, while statistics say that in large closed mesh there is about twice as many figures as vertexes.

A vertex store more data than simple its position in 3D space. It stores in addition normal direction, which can be used in smoothing mesh, either by faster Phong-like interpolation or better looking subdivision surfaces, when also object silhouette is correct. A vertex also may have texture coordinates. They are two real numbers which describe how to stretch a 2D map onto object. If model does not contain textures the coordinates can be omitted. In the vertex there is optionally anisotropy direction. This direction is not used in current version, but in future will be used in anisotropic material, such as brushed metal.

9.2.2 Intersections

When a figure test if ray hits it returns true, the figure has to fill in some of the intersection data. This filling due to efficiency is split into two procedures. First intersect fills only coordinates of intersection and rest optionally. There can be found a better intersection with nearer figure, so all previous work is wasted in that case and figure should compute only data necessary to second procedure. This is called when it is sure that given intersection is final and nothing better exists. Figure has to complete the normal, anisotropy direction and texture coordinates. For example in triangle this is performed by interpolating vertex data using barycentric intersection coordinates.

9.2.3 Triangles, Quadrics and Meshes

The platform provides implementation of basic 2D surfaces. Most commonly used are triangles, which are simplest 2D shapes. The properties and implementation of them is obvious – simple solving linear system. The quadrics available in platform contains sphere and based on four vertexes bilinear surface. In spheres it is important to use a stable algorithm while computing the intersections – ‘school’ one may give arbitrarily large error with the smaller intersection distance. The latter surface is forced to be bilinear, since four points almost always happen to be non coplanar, and therefore the formation of proper planar quadrilateral is impossible.
There are no good way of defining how small must be error of treating four points as they are on common plane. If the bound is too strict, almost every model would not load properly, otherwise on zooms there appear distracting gaps between shapes.

In future we plan to extend our shapes with meshes. These meshes can be built from triangles and bilinear surfaces. The advantage of meshes over simple shapes is possibility of using subdivision algorithms, which can make triangle based surface look smooth.

9.3 KD Tree

Scenes which are used in practice often have huge number of figures (commonly triangles). However, the typical figure distributions in such scenes are that any ray intersects only few of them, missing the rest by far distance. According to this it is extremely inefficient having all the figures in one array and testing all of them by loop. There exists some acceleration techniques to solve this problem, and best of them allow to achieve the statistical logarithmic complexity of intersection founding (worst case is still linear).

The most basic requirement for our platform is that is must be general. That is, the acceleration structure must work well for as many as possible real scenes figure distributions. To achieve this we cannot use some classic techniques. I.e., grid accelerator fails completely if the figure distribution is uneven, or portals and PVSs (potentially visible sets) commonly used in fpp games are strongly dependent of scene geometry. After some checks we found that only pure tree-based structures are general enough to be used in our platform. We test the Kd-Tree and Oct-Tree and found that first of them is better both in performance and memory consumption.

9.3.1 Tree Building

Tree is build from array of figures. The two operations needed to be implemented in figure in order to place it in tree are computing axis aligned bounding box of figure and ray-figure intersection test using latter in traversal. For the optimization (described in the next chapter) figure have to have third operation – test if it has any common part with axis aligned box. This function is optional and figure can always answer yes in this method – tree still is correct, but a bit larger and slower.

Tree is built by recursive approach. At each level of recursion we split current tree bounding box into two halves. Surprisingly, the performance slightly depends of location of subdivision plane, so techniques that approximate the median of figure distribution are of little use. Our implementation uses the simplest possible method here – dimension for subdivision is selected as largest box extent (or alphabetical order if box is equal in two largest dimensions), then splitting plane splits box on halves. This approach has two advantages – tree is built fast and memory consumption per node is minimal since it is not necessary to store splitting data. Array of figures then is split into four categories: figures that fall into left sub-tree, right sub-tree, both of them and those that are too large to be effectively pushed down to both sub-trees. We found that performance is very sensitive to decision if we put a figure that lies on division plane into both sons (hoping that this figure reference will not be duplicated in future subdivisions) or place it in root of these sub-trees. Next there are built two sub-trees with the same algorithm and appropriate figure arrays. Notice that to avoid messing up figures between two recursive calls we have to temporary store the ones that falls to both sub-trees.

9.3.2 Tree Traversal

The fast ray traversal is based on assumption that if we extend ray into infinity if it intersects box it must exits from it. Ray may starts in box, so actually it may exits form it without entering here. First we search for exit point from box for whole tree, and if point cannot be found this means ray misses all geometry. Searching is done by recursive algorithm with exit point as parameter. On every step algorithm checks if ray intersects with splitting plane of node. If so, potentially both sub-trees have to be searched. First, it is searched nearest with respect to ray direction sub-tree with founded intersection as exit point, and if no intersection is found, the further sub-tree is searched with the current exit point. If ray does not intersect the splitting plane it passes by the only one sub-tree – this for which the exit point belongs to. Next we search for nearest intersection all the node geometry by simple loop. Notice that if reference to figure is stored in more than one node, it may be found intersection that is outside currently searched node bounding box. This intersection must be skipped, as it may cause routine to return incorrect result – see Figure 9-1. Extreme caution must be taken when checking if point lies inside box due to precision errors. The point itself is found by intersecting figure with ray, with is potential source of such errors. In common case when figure is flat and lies exactly on node border (i.e. floor of architectural model) it may be missed when incorrect algorithm is used. There has to be performed an error estimation. In our algorithm the estimation is based on the size of node, and thus size of figure. However, the error must not be overestimated, since this cause return of the missing tiny triangles problem.
There are two routines which use tree traversal – founding the nearest intersection and visibility tests. In the latter case the traversal is optimized – it exits from recursion after finding first intersection in given interval, not necessarily the nearest.

### 9.3.3 Optimizations

All the optimizations mentioned here can be applied while building the tree. In basic version of algorithm we find bounding box of figure, then we test it against split bound of node. If figure box lies on splitting plane, the figure not necessarily must have common parts with both sub-node boxes (see Figure 9-2). In optimized version in that case we perform two additional tests – we check if one (or both) of sub-boxes has a common part with the figure (the actual figure, no its bound). Notice that all figures must have common part with at lest one of boxes, so if first test returns false we can assume second do not, and not call it at all.

![Figure 9-2. Test for right down box.](image)

The next optimization we called shearing of tree. If the original building algorithm places reference to figure in both sons of one node (this happens most often in leafs, but not always) we can remove these references and put it back into higher node as one reference (example subtree structure which allows shearing is shown in Figure 9-3). The initial structure does not accelerate the searching at all and consumes additional memory. When not using mailboxing (see next chapter) it even decreases performance, so it is always worth doing the shearing. As a result there can appear empty nodes, removing these further improve memory usage and performance.

![Figure 9-3. Shearing case.](image)

The last optimization we have used is to improve memory coherence and therefore reduction the number of cache misses. The resulting tree from building algorithm consists many small blocks of memory (one for each node) at random addresses. We correct this by save whole tree in temporary file on disk, then throw out whole structure from memory, and reload tree into one large block. We restore tree in a way that left sub-tree is placed immediately after its parent, so the entering left sub-tree rarely causes cache miss.
9.3.4 Parallel Search

In our platform it is important to allow parallel tree search. There is one common and efficient optimization that is difficult to use in parallel mode – mailboxing. It is addressed to avoid more than one intersection test for ray with the same figure during one search. This may occur if reference to figure is stored in more than one node, and ray fails the first intersection test. In that case it is obvious that ray fail this test again while searching other node, so performing it one more time is useless. The mailboxing cuts out redundant tests by placing a global counter for whole tree. The counter is incremented by one for each search call. Every figure has an extra field (so called mailbox), where is placed the ray number after performing intersection test. This allows skipping the intersection test if mailbox value is equal to ray number. Above technique, however, raises numerous problems. First, the figure must store extra data – it is no pure interface any longer, and the whole structure is no more read-only. Also, from theoretical point of view, the complexity of searching becomes linear, since once a fixed number of searches (equal to mailbox maximum value) the whole structure must be reset. Fortunately this is not meaningful in practice, since on 32-bits systems this happens only once for 4G searches. From practical point of view the most painful is synchronization of parallel searching. It is clear that when more than one thread performs searching in parallel, the ray numbers and mailboxes might be messed up, and the necessary tests accidentally skipped. This is clearly not good. To make the algorithm work are three solutions – synchronize searching routines, or create private storage for each of threads, or make the whole tree read-only after being built (resign from mailboxing). First option is extremely inefficient, because most computation time is spent searching tree, so not allowing parallel tree search is effectively eliminating any speedup. The second one is memory and time consuming, and requires the knowledge of how many threads there are. After some tests we found that after such an ‘optimization’, the whole algorithm is slower than without it.

We strongly argue for not using mailboxing in parallel mode. This may be surprising that this approach is not faster, but after some examination it is clearer. First, tree is huge structure, probably the largest in whole application. Making it read-only greatly increases cache performance. What is more, the memory consumption is reduced. It is also worth to notice that shearing makes the lack of mailboxing much less painful. The figures for which references are doubled near root are unlikely to be tested twice, because ray probably hits something in large subtree, and searching the sibling subtree is not necessary. The references to figures that are doubled just before leaves are effectively removed by shearing. Taking this all into account there is unlikely that figure would be tested more than once, so expensive in parallel mailboxing has more drawbacks than advantages.

9.3.5 Future Extensions

The presented here kd-tree has one major drawback. When the object for which tree was built is soft (some its parts change position relatively to other parts, i.e. a man moving his hand) the motion blur cannot be simulated. The tree building is expensive (O(N log N) complexity) and it is built once per frame, which prevents from simulating this effect. In fact, this kd-tree allows only using motion blur when camera or object as whole move.

In future platform version this tree might be extended to four dimensions, where the fourth one is time. The whole tree logic remains virtually the same. Only necessary changes are in figures and ray representation. Two dimensional figures then become three dimensional volumes placed in four dimensional space. Rays then have a fourth dimension which means time when ray is traced. Of course time affects only ray start point parameter. Ray direction is still three dimensional because of the assumption that light speed is much greater than any object speed in scene. The tree size in time (and the time resolution) relative to space size might be obtained by comparing the object movement speed multiplied by exposure time with is size.

9.4 Color and Scalar Sources

These sources have very simple interface. They have only one method. Arguments are coordinates and addressing mode. The coordinates used are dependent on dimensionality of source. The two dimensional sources uses mapped \((u, v)\) values, two+ dimensional uses \((u, v, time)\), three dimensional uses natural intersection coordinates \((x, y, z)\) and the three+ uses \((x, y, z, time)\) coordinates. The addressing mode has the meaning very similar to DirectX one. Return value of these methods is full spectrum for color sources and single real value for scalar sources.

There are few kinds of both color and scalar sources. The most interesting are complex sources, which allow mixing results of two other sources by performing particular operation on their results and texture sources which allow using standard image formats. The sources also can define a procedural noise patterns such as marble or wood.
9.5 Textures

The textures are raster images that can be mapped onto figures in artificial scenes. The good use of texturing can greatly increase the realism of the rendered image, but textures themselves create many difficulties to rendering algorithms. The most commonly mentioned in literature is problem of aliasing and various filtering techniques addressed to mitigate it. There is also constant problem with large textures size and therefore lacks of memory. The common solutions are the numerous techniques of progressive objects resolution. However, almost all progressive approaches fail less or more in ray tracing, so we omit it completely and focus on careful usage of compressed image formats. In this chapter we describe in detail both filtering and rendering of compressed textures.

9.5.1 Filtering

Under the term of filtering there are understood two different operations. One is reconstruction, when continuous function is recreated from point samples and second is prefiltering in order to remove too high frequencies from function which cause aliasing. Unfortunately in ray tracing prefiltering is difficult and expensive to perform. There are exists techniques to do this, for example tracing ray differentials, but they are slow and result tend to be too blurry. We do not implement them and instead we rely on randomized supersampling as the natural solution in Monte Carlo approach. This seems to be a bit slower for large textures squeezed on small area, but never introduces excessive blurring. Nevertheless, to obtain good looking results the good reconstruction filter always must be applied. Naïve approach of reading sample value that is nearest to intersection point leads to awful looking images as in early 3D PC games.

9.5.2 Reconstruction

There are many standard filtering techniques available in literature to solve this problem, so we do not describe them here. The excellent reference on this topic is [MITCHELL+1988]. However, after some tests, we found that especially when large magnification occurs, there appears problem of anisotropy. Due to this, we strongly argue to use rotational symmetric filters in high quality rendering applications. This seems pretty reasonable, because having particular axes along which filter behaves differently clearly leads to distracting artifacts – see Figure 9-4 and Figure 9-5 for comparisons, notice the square formed by \( f = 0 \) constraint in left figure and circle in right one (the negative lobes of filter were exaggerated eight times to make them visible in printing).

![Figure 9-4. Classic Mitchell filter.](image1)

![Figure 9-5. Isotropic Mitchell filter.](image2)

The classic filter design put lots of attention to make filter separable. Filter satisfies this criterion when \( f(x, y) = g(x)h(y) \), where \( f \) is filter function, and typically \( g \equiv h \). Separable filters are much more computational efficient than inseparable ones, hence their popularity. Rotational symmetric filter functions can be described in general as \( f((x^2 + y^2)^{0.5}) \), thus such functions are one dimensional curves rotated around filter centre. To make filter both rotational symmetric and separable, the function \( f \) must first remove the root and then change the addition to multiplication. There is such class function what can do it – exponential ones, for which \( f(x + y) = f(x)f(y) \). The root is removed by squaring function argument. Thus, the only filters that satisfy both criteria together are Gaussian filters. Unfortunately, they have infinitely large support, and then must be clamped. Also Gaussian filters make more blur than is necessary. The blur, however, may be an advantage in cases when image has unwanted details which should be eliminated, as compression artifacts in JPEGs. In all other cases the best choice are spline based filters or fourth degree polynomials converted to rotational symmetry. The test results which are obtained by downsampling the reference image (taken from [PHARR+2004]) eight times in horizontal and vertical directions and restore it to original size by various filters are presented below.
Nearest point sampling looks awful and should be never used in practice. The Gaussian filter produces slight blurry image, and yet not removes all anisotropy. The Mitchell one produces less blur at the cost of ringing and exposing jagged edges, the square in filter function leaks to the reconstructed image. Our rotational symmetric filter with fourth degree polynomial as a base eradicates square appearance at the cost of slight additional ringing. In design of this filter only one parameter is necessary which changes the proportion of blur and ringing. Due to rotational symmetry it never can introduce anisotropy as in Mitchell filter.

9.5.3 Image Formats

The good candidate to use as compressed texture to rendering is surprisingly standard JPEG format. With some work it can be converted to random access and therefore allows reasonably fast reading of arbitrary pixel value. For the needs of this platform we have implemented the image library that supports JPEGs and some kinds of bitmaps. We believe that this library is more general and can be used not only in this project, so we have described it in separate article attached to this thesis (Appendix C).

9.6 Materials, Reflection Models

Materials describe the way how light is scattered from surfaces. Materials actually do not define the particular color of surface, they might be matte, polished, transparent and so on. Colors are defined by sources, which are finally used by materials. Materials are responsible for implementing the scattering part of sampling interface. Therefore materials have three methods – to randomize scattering direction, evaluation of BSDF and evaluation of probability.

The provided material library gives a great flexibility of defining surface. Platform defines Lambertian matte, Phong plastic gloss, Cook metallic gloss ([COOK+1981]) and perfect mirror. Each material has a flag
deciding if scattering occurs on the same or opposite side of surface. Each material but matte has additional third possibility of reflecting light back opposite to input direction.

The implementation of basic materials is simple, and in fact is a direct translation of theoretical concepts. However, to avoid duplicating code and therefore huge library size and maintenance problems, materials architecture is carefully designed. First, using material type and flag information ideal output direction is calculated. For Lambertian ones, the direction is alongside or opposite normal. For the rest, it is ideal mirror reflection or refraction or negation of input direction. After computing ideal direction actual scattered one is computed by disturbing it by particular function or returning it as is for perfect surfaces. When disturbing happens to ‘scatter’ ray on wrong side of surface (possible with Phong or Cook model), then ray is immediately absorbed by surface. This probabilistic computation of radiance cutoff on grazing angles increases variance only slightly, but greatly simplifies otherwise very complex pdfs and other equations.

To make the materials much more useful, platform provides complex material – a powerful tool to combine two other materials to achieve more interesting and convincing effects. The complex material is a linear combination of two simpler ones. The combination can be done in two different ways. It may be wavelength dependent to make special effects or constant to allow faster rendering. For the purpose of selector therefore can be used color or scalar source. The implementation of such materials is done by randomly selecting material to scatter, and by weighting in appropriate way returned probability or BSDF from these materials.

### 9.6.1 Transparency Optimization

Due to efficiency reasons there is one case when the materials behavior is different from described above. It is common to see models containing coarse geometry, with fine details added by transparency maps instead of extra triangles. While rendering, these models cause platform to create paths that have unnecessary vertexes on invisible surfaces and therefore excessive variance. The optimization allows figure to ask material if it is fully transparent in given intersection point and abandon intersection if so. This optimization does not increase rendering time by measurable amount and results in a huge noise reduction on those scenes – see chapter 10 for results.

### 9.7 Light Sources

In addition to stand alone light sources our platform also allows to use embedded light sources. This is similar to emissive material in DirectX, but it is physically correct. Our embedded light sources are attached directly to figures, and they acts as a fully functional lights – they lighten other objects, instead of only modifying given figure appearance.

The main difficulty in designing this kind of light is cooperation between figure and emitter while each one does not know anything about other’s preferences. This results in lack of reliable method for importance sampling. If, for example, figure randomizes point for emission with uniform area probability, this potentially may be extremely inefficient if emitter’s irradiance is strongly irregular. The invert method, when the emitter randomizes sample point in \((u, v)\) terms can also be very inefficient if figure mapping is not area preserving, for example when using standard square to sphere mapping this results in majority of points near poles. In current implementation this problem is not solved. The chosen technique is to select point from figure distribution, since majority of lights are uniform, and in that case figure selection performs better. To solve this problem in future versions there can be used Multiple Importance Sampling [VEACH1997]. This technique allows using both sampling techniques and selecting one point on the fly in a way that minimizes variance. The actual implementation, where only one sample can be computed, may use one technique selected ad random with probability 0.5, and therefore with doubled weight to account for it. In the worst case the variance can be twice as optimal, but technique never fails completely as current one when for example emitter has a dim texture with one bright spot.

To implement the extension figures have to have an extra methods to convert from \((u, v)\) to \((x, y, z)\) and compute a local ratio of parameterized and world space areas, i.e. \(dA_{uv}(f(u, v))/dA_{wz}(u, v)\), where \(A_{wz}\) means area in world space, \(A_{uv}\) means area in \((u, v)\) space and \(f\) is conversion between these spaces.

### 9.8 Bump Mapping

The bump maps technique [BLINN1978] was developed to make smooth surfaces appear as they are wrinkled. The bumps are created by modifying surface normals without actual change of surface geometry. This technique is simple and can produce convincing effect. Unfortunately it has no physical basis and therefore cause troubles for correct algorithms. Artifacts produced by bump maps are: light leaks and pure black patches (see Figure 9-7) and asymmetry. The light leaks can be removed by performing an additional test if both input and output rays lie on the same side of surface for reflection and opposite sides for transmission. Unfortunately black spots cannot be removed so easily. Solution presented in [VEACH1997, pp. 156-158] results in calling material
function with incorrect input – angle between normal and ray direction larger than \( \pi/2 \) radians, which is visible in figure.

The asymmetry can be corrected by multiplying original BSDF by additional factor

\[
f'_r(x, \omega_i, \omega_o) = f_r(x, \omega_i, \omega_o) \left[ \frac{\omega_i \circ N_g}{\omega_i \circ N_s} \right],
\]

where \( N_g \) denotes true geometric normal. The scaling takes place only if BSDF is used for trace light and does not appear if BSDF is used to trace camera rays. This expression is derived in [VEACH1997, pp. 151-153].

Bumps are attached directly to figures. The platform implements bump map by the extended scalar sources. Such scalar defines the relative ‘height’ of points above true surface. The extension allows getting the gradient in \( u \) and \( v \) dimensions instead of value in point. After this there is computed a quaternion of rotation that maps unit vector to the new one computed from gradient. Then the actual surface normal is rotated by previously computed quaternion, what makes a disturbed normal. Finally the disturbed normal instead of true one is given to figure material. The bump is also responsible of correcting material output due to introduced asymmetry.

### 9.9 Volumetric Effects

The volume objects contains four components – bounding box which defines the region of space affected by participating medium, two objects – 3D or 4D color sources which define the attenuation and scattering coefficients respectively and a special volumetric material (to define a phase function). This material is similar to surface ones, but of course acts without normals and its domain is full sphere of directions. All these elements have the same possibilities as their surface analogs.

The volumes can overlap. If they happen to do so, attenuation is summed, and scattering occurs on all of the media simultaneously until one of them randomizes first a point of interaction. Then this point is nearest, and becomes a next vertex of path, with scattering done by this medium which randomizes the point. Of course, more dense the medium is, the more chance to scattering it has. The addition is not performed directly, but when the path length is founded, each medium acts independently by attenuating light on path, multiplying it by computed amount. When ray is finally attenuated by all the media, the final effect is identical to one large medium which is obtained by summing coefficients of all smaller media.

The media can be homogenous (the scattering and attenuation coefficients equal in the whole cube) or not. In the first case all the necessary information can be computed analytically, so it is worth to treat it as a special case – when for example we want to model a building with dusty atmosphere the performance gain can be substantial. Otherwise (for example fog over the lake, with varying density) numerical integration has to be performed. This integration can be excluded from the light transport equation analytically and be solved as one dimensional quadrature with probably smooth integrand. This allows to effective implementation of classical integration schemes for this special case.

All the expressions presented here are derived in [PAULY1999] and [PHARR+2004]. For both mediums, attenuation of light due to absorption coefficient can be computed directly from coefficient definition:

\[
\frac{dL(x, \omega)}{dx} = -L(x, \omega)\sigma_a(x),
\]

which analytical solution is:

\[
L(x + s\omega, \omega) = L(x, \omega) \cdot e^{-\int_0^s \sigma_a(x + t\omega)dt},
\]

where the \( s \) is distance for which ray travels through participating medium. This equation becomes trivial in homogenous medium, where \( \sigma_a \) is constant, hence this special case:

\[
L(x + s\omega, \omega) = L(x, \omega) \cdot e^{-\sigma_a \cdot s}.
\]
Despite the virtually identical definition, the scattering has to be treated in different way, since due to it radiance is not only decreased but also can increase, and to compute this sampling is necessary. Since the good importance sampling is important, the reasonable idea is to make \( cdf \) from attenuation expression:

\[
cdf(x + s\omega,\omega) = 1 - e^{-\int_{s}^{\infty} \sigma_s(x + t\omega) \, dt}.
\]

In this case the probability pdf of scattering light as it travels through medium is

\[
\text{pdf}(x + s\omega,\omega) = \sigma_s(x + s\omega) e^{-\int_{s}^{\infty} \sigma_s(x + t\omega) \, dt},
\]

which is product of probabilities that light is not scattered before \( x + s\omega \), and is scattered in this point. This allows avoiding using of any extra weight factor in addition to phase function when scattering light in volume, and therefore not increasing variance. The sampling distance \( s \) to first scattering event using canonical random variable one may use the following expression:

\[
\ln(1 - \xi) = \int_{0}^{s} \sigma_s(x + t\omega) \, dt.
\]

In our platform we solve this equation by using adaptive technique of some kind enhanced by randomizing the offset of first step to mitigate aliasing. Notice that all these expressions become very simple in homogenous media. When the nearest intersection point happens to be behind the solid wall, the procedure is slightly modified. As interaction object is selected the surface, not participating medium with the probability

\[
P = 1 - cdf(x + S\omega,\omega),
\]

where \( S \) is the distance to nearest solid surface. Thus the surfaces can be seen as delta distribution within volumes, i.e. volume that has infinite density in 2D subset of points in \( R^3 \).

To complete the definition of participating medium there is also necessary a volumetric material which defines a phase function. Good reference to this topic is \([PAULY1999]\) and \([PHARR+2004]\). The framework of our platform allows also combining basic volumetric materials to create more complex one, in the same way as for ordinary surface ones.

### 9.10 Procedural Modeling

The procedural modeling allows creation of very convincing images with very little memory usage. Our platform, as software rendering system allows a huge freedom of defining such functions, which is far beyond the range of currently available graphic hardware. The provided by our platform are based on Perlin noise \([PERLIN1985]\) multidimensional textures or volumetric densities which simulates natural phenomena – marble, wood and clouds.

The noise is simply an extended hash function defined on integer lattice converted to real values from range \([-1, 1]\). The domain extension from integers to all real values in our implementation is done by cubic interpolation. Details on many scales are made by summing up octaves of noise, which forms so called turbulence. The subsequent octaves are created by using the same function, but multiplying input coordinates for it by factor of two for each octave. To assure that resulting series converge, and therefore subsequent octaves can possibly add arbitrarily many details with limited attitude, each subsequent octave must be attenuated by factor less than one. Thus the final expression for turbulence is

\[
t(x) = h(x) + ah(2x) + a^2h(4x) + \ldots + a^n h(2^n x),
\]

where \( x \) is point, \( t(x) \) is turbulence in \( x \), \( h(x) \) is hash function extended to real values, \( a \) is attenuation and \( n \) is number of calculated octaves. When \( x \) is multidimensional, the multiplication by power of 2 is performed independently on each component, and results can be transformed to one value as a sum of components multiplied by large primes, different for each component. If \( a \) is less than one, the maximum turbulence value never exceeds \( h_{\text{max}}(1-a)^n \), where \( h_{\text{max}} \) is a maximum output value for hashing function.

The convincing textures can be created by using a regular function disturbed by turbulence. We have implemented marble as a disturbed sine, wood as a sawtooth function rotationally symmetric around y axis and clouds as pure turbulence condensed by exponential function. The results are presented in Figure 9-8.
Figure 9-8. Sample procedural textures generated on the base of Perlin noise.
10 Results

In this chapter there is a discussion of results obtained by rendering different scenes with implemented platform. First are shown detailed comparisons of how different algorithms render the same scene and how our optimizations to them can decrease variance or rendering time. Next there is presented a test of parallelization effectiveness by rendering a difficult, large scene with best algorithms in sequential and parallel mode, with production quality. This test also shows the complexity of algorithms based on randomness and strong law of large numbers – to reduce the error twice the running time must be increased four times. Finally there are shown some potential extensions to the platform, which can greatly increase its functionality without large changes of internal architecture.

All these images are also provided in png format on attached CD. Some of them are rendered in stereovision technology described in chapter 8.3.1. They are available in version for both left and right eyes and may be directly viewed by specialized hardware.

10.1 Path Tracing vs. Bidirectional Path Tracing

The idea of this comparison is to show how much better algorithm can improve on scene with tricky lighting conditions (Figure 10-1). The scene contains specular dispersive glass and polished procedural marble floor. Marble is modeled as linear combination of diffuse and specular materials. The remaining surfaces are diffuse. Notice the highlight on ceiling due to refraction in glass ball and scattering from marble.

Figure 10-1. Reference image, 1280x960, computed with 512 samples per pixel.
Figure 10-2. Path Tracing vs. Bidirectional Path Tracing

The image generated by Path Tracing uses two samples per pixel, and Bidirectional Path Tracing one, to make rendering times approximately exact (Figure 10-2). The ordinary path tracing is hopeless on this scene. The convergence is extremely slow on entire image, and especially on highlights and even parallelization helps little. This case shows how important is using good quality algorithms, even in parallel environment. On the Figure 10-1 there is a reference image created by Bidirectional Path Tracing with 512 samples per pixel. The reference image was computed in approximately 1 hour time on network of 9 machines (with 13 CPUs in total).

10.2 Reduction of Visibility Tests

This technique is addressed to reduce rendering time in BDPT possibly without introducing large additional variance. Our name ‘combinatory optimization’ is due to way of its work – it computes in how many possibilities the full path with given length can be concoct from light and eye subpaths and cuts out all but random chosen one.

Figure 10-3. Reference image, 1280x720 computed with 8 samples per pixel.
Version 1. No optimization, 17’22'', 307k samples.

Version 2. Optimization, 10’20'', 307k samples.

Version 3. Optimization, 17’18'', 504k samples.

Figure 10-4. Combinatory Optimization in action.

The above pictures (Figure 10-4) show the effect of optimization. All images was generated on single machine with single 2.4 GHz Intel P4 processor. Rendering is done in one sample per pixel to easy asses the visual errors. Right picture is generated 1.67 times faster (17’18’’ vs. 10’20’’) without introducing more visible artifacts. There is also provided for comparison an image generated in the same time as without optimization and somehow better quality (bottom row). The difference can be even greater when scene contains much glass causing total internal reflection. We cannot compare this technique with Efficiency Optimized Russian Roulette proposed by Veach [VEACH1997] because our abstract framework does not assume the image is formed from pixels, and these are base of Veach’s optimization. For comparison, the reference image was rendered in about 3 hours.

10.3 Transparency Optimization

These pictures (Figure 10-5) are rendered deliberately in low quality (one sample per pixel) to make errors easy comparable. Left image is rendered with classic approach and the right one uses special handling of fully transparent materials. The triangle in middle of the box has three transparency levels – full transparent, half transparent and no transparent with ideal black color. Computing times are equal and yet the right image has much less noise on fully transparent parts of triangle.
10.4 Parallel Rendering of Large Scenes

The scene rendered in this test has high geometric complexity. The castle room has over 300k triangles and quadrilaterals with huge size variation. The dragon (model from Stanford University) has over 800k triangles, with similar sizes. Together with both armors and weapons scene consists of about 1.5M figures. The only lights placed onto the scene are the two spheres. All illumination comes directly form them or through multiple reflections. Renderings were performed on network of 9 machines, 5 of them with single processor and 4 with two ones (eleven 2.8 GHz P4 and two 3.0 GHz P4 processors in total). The machines were connected by Fast Ethernet. The rendering time for reference image was about 12 hours, however after a few hours the image quality was acceptable.

We have found that as the requested image quality is increased, the measured speedup becomes linear. The sequential part – parsing scene representation and building trees takes approximately one minute on each machine regardless of requested quality. The time necessary for gathering of final result can be neglected (sending about 1280*720*3 bytes of data from eight machines to master one).
We also show what square root order of method convergence means in practice. In the Figure 10-6, Figure 10-7 and Figure 10-8 there are images with progressively increased rendering time and therefore image quality. The reference image (Figure 10-9) uses 512M samples in total (approx. 546 samples per pixel). During rendering there was a temporary save for every quadruple number of samples, meaning that each subsequent image has error reduced by half. For example, if a red component of pixel \( p_i \) in image \( j \) has true value \( 100 \) and computed value in range \( 100 \pm 10 \) with probability \( p=99.7\% \) the same pixel in image \( j+1 \) will have a \( 100 \pm 5 \) computed value with the same probability.
Figure 10-9. Reference image – the scene rendered with 512M samples in resolution 1280x720.
## 10.4.1 Convergence Tests

These tests are performed on 640x360 version of castle room image from previous chapter. Error is measured using both $L_1$ and $L_2$ norms. The $L_1$ norm is computed as a sum of absolute values of differences between each pixel of current and reference image:

$$\sum_{i=1}^{n} |p_i - r_i|,$$

where $p$ is a pixel from currently compared image, $r$ is a respective pixel from reference image and $n$ is three times number of pixels in image. The pixel components are treated as equally important and summed independently. The $L_2$ norm is square root from sum of squared differences:

$$\sqrt{\sum_{i=1}^{n} (p_i - r_i)^2}.$$

There is also a $L_\infty$ norm in common usage (maximum absolute difference value) but it makes no sense with non-deterministic algorithms. These make absolutely no guarantee on value of single pixel since number of samples on it is typically small. The error estimate on the whole image is much more reliable since for it there are used all image samples. As one may expected, results of these experiments show that $L_\infty$ norm is even not monotonous, error measured in this way could even increase with number of samples.

In the

<table>
<thead>
<tr>
<th>SPP</th>
<th>error</th>
<th>$err_i / err_{i+1}$</th>
<th>$err_i / err_4$</th>
<th>$\log_2$</th>
<th>$\log_2 \log_{2,i} - \log_2 \log_{2,i+1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.35</td>
<td>36427.9</td>
<td>1.100</td>
<td>1.379</td>
<td>0.464</td>
</tr>
<tr>
<td>2</td>
<td>0.50</td>
<td>33107.9</td>
<td>1.115</td>
<td>1.254</td>
<td>0.326</td>
</tr>
<tr>
<td>3</td>
<td>0.71</td>
<td>29704.8</td>
<td>1.125</td>
<td>1.125</td>
<td>0.170</td>
</tr>
<tr>
<td>4</td>
<td>1.00</td>
<td>26409.3</td>
<td>1.133</td>
<td>1.000</td>
<td>0.000</td>
</tr>
<tr>
<td>5</td>
<td>1.41</td>
<td>23308.7</td>
<td>1.135</td>
<td>0.883</td>
<td>-0.180</td>
</tr>
<tr>
<td>6</td>
<td>2.00</td>
<td>20529.1</td>
<td>1.133</td>
<td>0.777</td>
<td>-0.363</td>
</tr>
<tr>
<td>7</td>
<td>2.83</td>
<td>18114.6</td>
<td>1.135</td>
<td>0.686</td>
<td>-0.544</td>
</tr>
<tr>
<td>8</td>
<td>4.00</td>
<td>15953.5</td>
<td>1.125</td>
<td>0.604</td>
<td>-0.727</td>
</tr>
<tr>
<td>9</td>
<td>5.66</td>
<td>14183.9</td>
<td>1.123</td>
<td>0.537</td>
<td>-0.897</td>
</tr>
<tr>
<td>10</td>
<td>8.00</td>
<td>12631.3</td>
<td>n/a</td>
<td>0.478</td>
<td>-1.064</td>
</tr>
</tbody>
</table>

there are presented results for $L_2$ norm. The first column contains number of samples per pixel (SPP). The second column gives the error estimate computed using the given norm. In the third column there is a quotient between two subsequent errors, in the fourth column there is quotient between current error and error from image with one sample per pixel. In the next column there is base 2 logarithm of previous column, and in the last column there is a difference of two subsequent logarithms. Each row contains $\sqrt{2}$ times more samples than previous one.

The results are not exactly as expected, since the increase of sample number four times actually does not reduce the error twice. This is caused by the fact that there are compared bitmaps instead of raw output. The RGB components of them are produced by nonlinear tone mapping procedures, which distorts the error comparison. Also tone mapping makes no sense of using relative errors (in all comparisons there are used absolute pixel differences). Nevertheless, we believe that main function of platform is rendering images, so despite all the difficulties numerical comparisons should be made on displayable images instead of raw output data.
Figure 10-10. Graph of convergence results from column 3 and 4 of Table 1.

Figure 10-11. Graph of convergence results from column 5 and 6 of Table 1.
In Table 2 there is presented a comparison of errors using $L_1$ norm. The actual values are completely different, but the relationships between them are similar. The difference could possibly be larger when using raw output data, due to nonlinear tone mapping, which drastically reduces the image dynamic range.

Table 2. Convergence test results for $L_1$ norm.

| SPP | error   | $err_i$ / $err_{i+1}$ | $err_i$ / $err_{4}$ | $\log_2$ | $\log_2$ - $\log_2$  
|-----|---------|------------------------|---------------------|-----------|-----------------------
| 1   | 0.35    | 23246100               | 1.109               | 1.413     | 0.499                 | 0.149                 |
| 2   | 0.50    | 20965300               | 1.123               | 1.275     | 0.350                 | 0.168                 |
| 3   | 0.71    | 18664400               | 1.135               | 1.135     | 0.182                 | 0.182                 |
| 4   | 1.00    | 16446600               | 1.142               | 1.000     | 0.000                 | 0.192                 |
| 5   | 1.41    | 14397200               | 1.144               | 0.875     | -0.192                | 0.194                 |
| 6   | 2.00    | 12589000               | 1.143               | 0.765     | -0.386                | 0.193                 |
| 7   | 2.83    | 11014200               | 1.142               | 0.670     | -0.578                | 0.192                 |
| 8   | 4.00    | 96421400               | 1.136               | 0.586     | -0.770                | 0.184                 |
| 9   | 5.66    | 84887600               | 1.133               | 0.516     | -0.954                | 0.181                 |
| 10  | 8.00    | 74897300               | n/a                 | 0.455     | -1.135                | n/a                   |

These relationships seem to be independent also of scene used for test. To support this statement we present results for completely different scene presented on Figure 10-12, computed in resolution 320x240.

![Figure 10-12. Reference image.](image)

Table 3. Convergence results for scene from Figure 10-12 with $L_2$ norm.

| SPP | error   | $err_i$ / $err_{i+1}$ | $err_i$ / $err_{4}$ | $\log_2$ | $\log_2$ - $\log_2$  
|-----|---------|------------------------|---------------------|-----------|-----------------------
| 1   | 0.35    | 13572.1                | 1.138               | 1.525     | 0.608                 | 0.186                 |
| 2   | 0.50    | 11931.4                | 1.153               | 1.340     | 0.423                 | 0.205                 |
| 3   | 0.71    | 10350.7                | 1.163               | 1.163     | 0.218                 | 0.218                 |
| 4   | 1.00    | 8901.70                | 1.175               | 1.000     | 0.000                 | 0.232                 |
| 5   | 1.41    | 7578.34                | 1.179               | 0.851     | -0.232                | 0.238                 |
| 6   | 2.00    | 6427.53                | 1.188               | 0.722     | -0.470                | 0.248                 |
| 7   | 2.83    | 5411.03                | 1.190               | 0.608     | -0.718                | 0.252                 |
| 8   | 4.00    | 4545.18                | 1.189               | 0.511     | -0.970                | 0.249                 |
| 9   | 5.66    | 3823.52                | 1.185               | 0.430     | -1.219                | 0.245                 |
| 10  | 8.00    | 3226.63                | 1.193               | 0.362     | -1.464                | 0.255                 |
| 11  | 11.3    | 2703.69                | 1.191               | 0.304     | -1.719                | 0.252                 |
| 12  | 16.0    | 2269.69                | 1.191               | 0.255     | -1.972                | 0.253                 |
| 13  | 22.6    | 1905.20                | 1.195               | 0.214     | -2.224                | 0.257                 |
Figure 10-13. Graph of convergence results from column 3 and 4 of Table 3.

Figure 10-14. Graph of convergence results from column 5 and 6 of Table 3.
11 Conclusions and Future Work

The final effectiveness of platforms depends on both – used algorithms and parallelization. However, the gains from them are quite different. We have found that parallelization of all algorithms implemented within the platform gives almost linear speedup with shared memory and asymptotically linear (with respect to increasing the image quality requirements) with message passing. In the latter case only sequential parts are preprocessed (i.e. trees are built while reading scene description, performed independently on each machine) and they gather the final result. What is interesting, the Intel Hyper Threading Technology seems to be almost completely useless while used for rendering. The speedup for two different processors is about 1.96, while for one HTT processor is near 1.0. The speedup from parallelization is almost independent on what scene represents, while the more sophisticated algorithms are designed to handle special, trickier cases. The rendering time reduction resulting from usage of better algorithms range from nothing to theoretically unbounded value, and is strongly varying on particular scene materials and geometry. In our test cases Bidirectional Path Tracing is sometimes better than ordinary Path Tracing ranging from several to hundreds times.

The platform also shows that whenever algorithm is capable of rendering given scene, the usage of parallelization can improve image quality substantially. If not, even with using large number of machines, final image still has considerable and visually distracting noise.

Platform implemented for this thesis shows that Global illumination algorithms can give extremely good quality images without bothering the user with necessity to specify step by step how scene should be rendered. For example, glass from 10.1 is defined by:

```plaintext
material glass:mtcpx { ← complex material, mix of reflection and refraction
  mtl material: mtsrz {
    cs color :cscst {value [0.9 0.9]}
    nb 1.25 ← refraction index n = nb + na*(A-360nm)
    na 0.001 ← placing value other than 0 here causes dispersion
  }
  mtl2 material: mtsrl {
    cs color :cscst {value [0.9 0.9]}
  }
  cs color: csfre {
    ← proportions of reflection and refraction by Fresnel equations
    nb 1.25
    na 0.001
  }
}
```

All tricky details how make this effect are solved by platform internally. The code producing approximated effect on DirectX or OpenGL is rather not so concise. What is more, these libraries can only approximate similar effect with poor quality. However, despite the large functionality of existing platform, there are some issues which can enhance the platform further.

First, the transparency optimization works well only for fully transparent materials. However, it is possible to extend it to support arbitrary material which do not change ray direction while scattering, that can only decrease ray radiance. To do this, when searching for intersections platform must store a set of found transparent intersections in addition to nearest known opaque intersection. The final radiance attenuation due to transparency when nearest opaque intersection is found is product of all transparent attenuations that came from their intersections nearer than opaque one. To make this optimization work correctly, platform requires also some minor adjustments to sampling procedures of complex materials. Implementing this optimization will also allow placing participating media in the same tree with figures. Currently they are stored in separate container, which decreases performance a bit.

Second, the kd-tree cannot be used to rendering motion blur of soft objects. This effect can be accounted for only if object is moved as a whole and not if, for example, a man moves his hand. Extending 3d kd-tree to 4d one with explicit representation of time as 4th dimension should help, but for the price of excessive rendering time and storage requirements.

Finally, it is possible to implement Photon Mapping in parallel mode, but the platform has to be modified to do it. In that case, the rendering is no more a single step of taking more and more samples. It must be split into two phases. In first of them, each worker builds a part of a photon map. After that it has to be performed an all-to-all communication step to make the photon map complete on each node. Provided that the network is capable of transporting at least as much messages as number of workers at one time, there is possible to perform this step with linear complexity using logical ring network topology. When all data has been exchanged, the rendering is performed in the same way as in current platform.
Bibliography

[ARVO+1990]
James Arvo, David Kirk
*Particle Transport and Image Synthesis*, SIGGRAPH 1990

[BLINN1978]
James Blinn
*Simulation of Wrinkled Surfaces*, SIGGRAPH 1978

[COUR+1981]
Robert Cook, Kenneth Torrance

[COUR+1984]
Robert Cook, Thomas Porter, Loren Carpenter
*Distributed Ray Tracing*, SIGGRAPH 1984

[COUR1986]
Robert Cook

[DELOURA2000]
Mark DeLoura
*Perełki Programowania Gier*, Wydawnictwo Helion 2002

[DEVLIN2002]
Kate Devlin, Alan Chalmers, Alexander Wilkie, Werner Purgathofer
*Tone Reproduction and Physically Based Spectral Rendering*, Eurographics 2002
http://www.cs.bris.ac.uk/~devlin/tonemapping.html

[GORAL1984]
Cindy Goral, Kenneth Torrance, Donald Greenberg, Bennett Battaile
*Modeling the Interaction of Light between Diffuse Surfaces*, SIGGRAPH 1984, pages 213-222

[HECKB1989]
Paul Heckbert
http://www.cs.cmu.edu/~ph/

[INTEL2002]
IA-32 Intel Architecture Software Developer’s Manual, volume 1 and volume 2
http://www.intel.com/

[JENSEN2001]
Henrik Wann Jensen

[JENSEN+2003]
Henrik Wann Jensen, James Arvo, Philip Dutre, Alexander Keller, Art Owen, Matt Phar, Peter Shirley
*Monte Carlo Ray Tracing*, SIGGRAPH 2003 course 44

[KAJIY1986]
James Kajiya
*The Rendering Equation*, SIGGRAPH 1986
[KELLER1996]
Alexander Keller
The Fast Calculation of Form Factors using Low Discrepancy Sequences, Spring Conference on CG 1996
http://graphics.uni-ulm.de/FormFactorII.pdf

[KOLB+1995]
Craig Kolb, Don Mitchell, Pat Hanrahan
A Realistic Camera Model for Computer Graphics, SIGGRAPH 1995
http://graphics.stanford.edu/papers/camera/

[LAFORTUNE+1993]
Eric P. Lafortune, Yves D. Willems
Bi-Directional Path Tracing, Proceedings of Compugraphics '93, Alvor, Portugal (December 1993), pp. 145-153
http://www.graphics.stanford.edu/~eric/Portugal.html

[MITCHELL+1988]
Don Mitchell, Arun Netravali
Reconstruction Filters in Computer Graphics, SIGGRAPH 1988
http://www.mentallandscape.com/Papers_siggraph88.pdf

[NIEDOBA2001]
Janina Niedoba
Notes from lectures, University of Science and Technology (AGH) 2001/2002, available (in Polish only) at:

[PAULY1999]
Mark Pauly

[PEERCY1993]
Mark Peercy
Linear Color Representations for full Spectral Rendering, SIGGRAPH 1993 conf., volume 29, pages 191-198

[PERLIN1985]
Ken Perlin
An Image Synthesizer, SIGGRAPH 1985

[PHARR+2004]
Matt Pharr, Greg Humphreys
Physically Based Rendering - from Theory to Implementation, Morgan Kauffmann Publishers 2004
http://www.pbrt.org/

[PHONG1975]
Bui Tuong Phong
Illumination for Computer Generated Pictures, Communications of the ACM 1975

[PLUCINSKY2000]
Agnieszka Plucinska, Edmund Plucinski
Probabilistyka, Wydawnictwa Naukowo-Techniczne, Warszawa 2000

[PRESS1992]
William Press, Saul Teukolsky, William Wetterling, Brian Flannery
http://www.library.cornell.edu/nr/bookcpdf.html
[RICHTER1999]
Jeffrey Richter
_Programowanie aplikacji dla Microsoft Windows_, Wydawnictwo RM 2002

[RUSHMEIER+1998]
Holly Rushmeier, David Banks, Peter Shirley
_A Basic Guide to Global Illumination_, SIGGRAPH 1998 course 5

[SMITS1999]
Brian Smits
_An RGB-to-spectrum conversion for reflectances_, Journal of Graphics Tools 3(2), pp. 1-14
http://www.cs.utah.edu/~bes/papers/color/

[STONE2001]
Maureen Stone
_A Survey of Color for Computer Graphics_, SIGGRAPH 2001 Course 4

[SZIRMAY1999]
Laszlo Szirmay-Kalos
_Monte-Carlo Methods in Global Illumination_, Vienna University of Technology 1999

[VEACH1997]
Eric Veach

[WHITTED1980]
Turner Whitted
_An Improved Illumination Model for Shaded Display_, Communications of the ACM 1980

[ZMARZ2006]
Michal Zmarz
_Automatic parallel generation of animations_, to be published
Appendix A: Hardware Requirements

Processor

The system is designed to work on IA-32 architecture with SSE2 instruction set support. Processors that satisfy these requirements are Intel Pentium4 and some newer models of AMD Athlon. This requirement is critical to enable the SIMD parallelization. We found that these processors allow a large speedup, and is not worth to make a different version of platform to older ones. This is because rendering on these was much slower, and what is more, we plan in future to convert to 64-bit Intel or AMD platforms. All of these supports SSE2 instruction set, so making an additional application version for processors that do not is simply a waste of time.

Memory

We cannot state the memory limit to use the system. The core uses less than 1MB, and all the rest is use to store scene description and output results. Of course, the more memory system has, the better effects in shorter time can be achieved. To estimate requirements for particular scene it is convenient to know some system characteristics:

- the triangle costs 32 bytes, kd-tree node size is on average 12-16 bytes large, and the number of kd nodes is typically $0.3N \ln N$, where $N$ is the total number of triangles;
- the output data for one pixel is 28 bytes large ($x, y, z, w$ for camera image and $x, y, z$ for light image, each component is float); using parallel rendering on more than one machine doubles output memory consumption;
- the textures can be kept in jpeg format while rendering, so these above two are only major source of memory consumption.

Knowing all these facts allows to fairly precise estimation of memory consumption. We found that the entire scene should be placed in physical memory. Even slight use of swapping drastically degrades performance. We test the scene 20% larger than system physical memory and the slowdown was about three times. The absolute theoretical limit on 32-bit systems is 2GB of memory consumption, but in practice due to heap fragmentation the limit is further reduced to somehow slight above 1GB. Modern systems memory management is based on assumption that virtual address space is much larger than physical memory size. Unfortunately this assumption becomes no longer true and this is strong argument to move to 64-bit platforms in future.

Network

We have tested this platform on 10Mbit and fast Ethernet. On the first one the delays was too long to allow interactive mode. On fast Ethernet this mode works well with networks of few to several machines. When rendering images with large resolution on much larger network, at least master connection should be on gigabit Ethernet. However, the rendering without earlier preview of partial results is no trouble. All the workers send the results back once, and the delays of few minutes are unimportant in compartment with total rendering time.