Abstract

This work investigates mathematical methods for modeling brightness distribution in three-dimensional scenes, with a particular focus on the "local estimation method of Monte Carlo." This modeling method is currently little known and not used in practice but has great potential, as confirmed by this study.

The aim of the work is to improve the accuracy and efficiency of synthetic image visualization by eliminating noise issues associated with the Monte Carlo method and accelerating the rendering process.

This study provides a detailed analytical review of current scientific publications on light modeling and visualization, identifying the most effective data preprocessing methods and algorithms for solving the rendering task. The "Local Estimation" method was tested using an analytical task in Matlab, as well as the only currently available domestic program that implements modeling using "Local Estimations." In the practical part, a program was developed for testing visualization methods, performing light modeling of a three-dimensional scene using the Monte Carlo ray tracing method. The generated images were subjected to noise reduction at specified intervals using a neural network denoiser. The applicability and effectiveness of the implemented methods were verified and confirmed.

As a result, software was developed for lighting modeling using the Monte Carlo method, demonstrating the ability to eliminate image distortions and accelerate visualization. The obtained results show significant improvement in visualization quality and reduction in rendering time.

The thesis is presented on 88 pages, contains 18 figures, a list of references with 22 titles, and 2 appendices.

Contents

Introduction 9
1. Mathematical modeling and visualization of an image 12
1.1 Creating a realistic image 12
1.2 Methods for mathematical modeling of illumination 16
1.3 Solving the global illumination equation by ray tracing 22
1.4 Monte Carlo method 28
1.5 Analytical solutions of the global illumination equation 32
1.6 Local estimates 34
1.7 Conclusions of the section 39
2 Light modeling and visualization tools 40
2.1 Embree library 40
2.2 Options for optimizing the modeling process 43
2.3 Using neural networks for denoising 49
2.4 Intel Open Image Denoise Library 57
3. Practical part 64
Conclusion 73
List of used literature 75
Appendix 1 78
Appendix 2 87

Introduction

In today's world, computer graphics plays a key role in various fields, from game and movie development to scientific research and design. One of the important tasks in this field is to create a realistic image, i.e. the desire to recreate on the screen a picture indistinguishable from reality, correctly demonstrating all physical phenomena such as shadows from objects, reflections, refractions, etc. This is primarily achieved by light. The distribution of brightness across the scene is the process in computer graphics that determines how light affects objects in the virtual world. Physically accurate lighting in a scene provides a number of benefits, making images more realistic and appealing to the viewer. It is easily perceived and understood by the human brain. By using physically-based light patterns, images match natural phenomena, enhancing the viewer's visual experience. Physically true lighting also allows for a variety of visual effects such as realistic highlights, shadows and reflections. From a commercial perspective, such lighting allows for more realistic and appealing visual effects. This is important in various industries such as entertainment, advertising and design. With more realistic images, companies can draw more attention to their products or services, improving their presentation and appeal to potential customers. This can help increase sales and brand competitiveness in the market. From a scientific point of view, such lighting plays an important role in research and development in computer graphics and computer vision. The use of realistic lighting models helps scientists and engineers to better understand the physical processes occurring in virtual and real scenes and to develop more efficient methods for modeling and displaying images. This can lead to the creation of new technologies and innovations that can be applied in various fields such as medicine, engineering, materials science and others.

There are many methods of illumination modeling, each of which has its own advantages and disadvantages. The paper considers such methods as ray tracing, as

well as local Monte Carlo estimates, which represent an effective approach to solving the visualization problem.

The latter in turn are a rare but promising method on which there is very little research and unsolved problems at the moment. To compete with commercially popular and frequently used methods, local estimators need to be refined and results need to be accelerated. The relevance of the current work is to investigate a promising method, and to find a method that can solve the problem of local estimation. The paper reviews the mathematical foundations, and explores options for accelerating the visualization of synthetic images, including various options for optimizing the simulation process, including the use of GPUs and neural networks. In particular, such acceleration method as denoising, and Intel Open Image Denoise library. The obtained results have practical value and relevance, as the library is currently under development and is not widely used, but it has competitive capabilities.

In practice, a program for lighting rendering was written, simulating visualization distortions arising from the method of local estimations, and a denoiser was applied, eliminating these distortions and speeding up the results. The obtained data were objectively evaluated and the results were reported at the end of the paper.

To summarize, the object of research of this paper is mathematical modeling of brightness distribution over a scene.

The aim of the work is to investigate the possibilities of accelerating the existing methods of visualization of synthetic images, in particular local estimates of the Monte Carlo method.

Research Objectives:

1) To conduct an analytical review of scientific publications to examine the the process of mathematical modeling of luminance and visualization.

2) To review the theoretical basis of visualization tools as well as its optimization. In particular, the Intel Embree and Intel Open Image Denoise libraries.

3) To use the obtained knowledge for practical implementation of software capable of demonstrating the work of the found features to improve visualization efficiency.

4) Evaluate the results obtained and formulate a final conclusion.

Based on the research objective, the first section reviews the possibilities of image visualization and identifies the main problems encountered in practice. The second section is devoted to the ways of solving these problems. The third section demonstrates the practical application of the solutions found. The conclusion contains the main conclusions of the work.

1. Mathematical modeling and image visualization

1.1 Creating a realistic image

In the process of creating an image of three-dimensional space on a computer, it is important to remember that we are working with a mathematical model that is stored in the computer's memory. In fact, the screen displays a bitmap image, which is a two-dimensional representation of the three-dimensional model. Thus, the goal of computer graphics algorithms is to establish a correspondence between the three-dimensional and two-dimensional worlds in order to achieve the most realistic visual perception.

With the development of computer graphics in the 60s and 70s of the last century, methods for drawing lines and surfaces, as well as removing hidden image elements, became the main trends. The Sketchpad project, introduced in 1963, played a key role by allowing users to interact with the computer interactively through a light pen and create vector drawings on the screen. At that time, the USSR began developing the Graphor graphics program library in the Fortran programming language. This library allowed to output graphical primitives on graph builder and display, which opened new opportunities in engineering. In the following years, computer graphics continued to evolve, including methods for modeling smooth curves, reproducing color graded surfaces, and creating virtual reality based on object design. In the 1980s, standard computer graphics libraries were established at ISO, which helped to standardize and disseminate computer graphics techniques and tools.

Thus, the development of computer graphics led to the creation of a wide range of tools and techniques, including image processing, virtual reality creation, shading algorithms, and colorization. Physically faithful lighting in a scene in computer graphics is achieved using a variety of tools and techniques that model the behavior of light according to natural physical laws.

Some of the key tools for modeling light are local and global illumination. These two crucial concepts in computer graphics and computer vision play a crucial role

in visualizing three-dimensional scenes. The figure (Figure 1) demonstrates a Cornell Box test scene in which the differences between local and global illumination from a visualization perspective can be clearly seen.

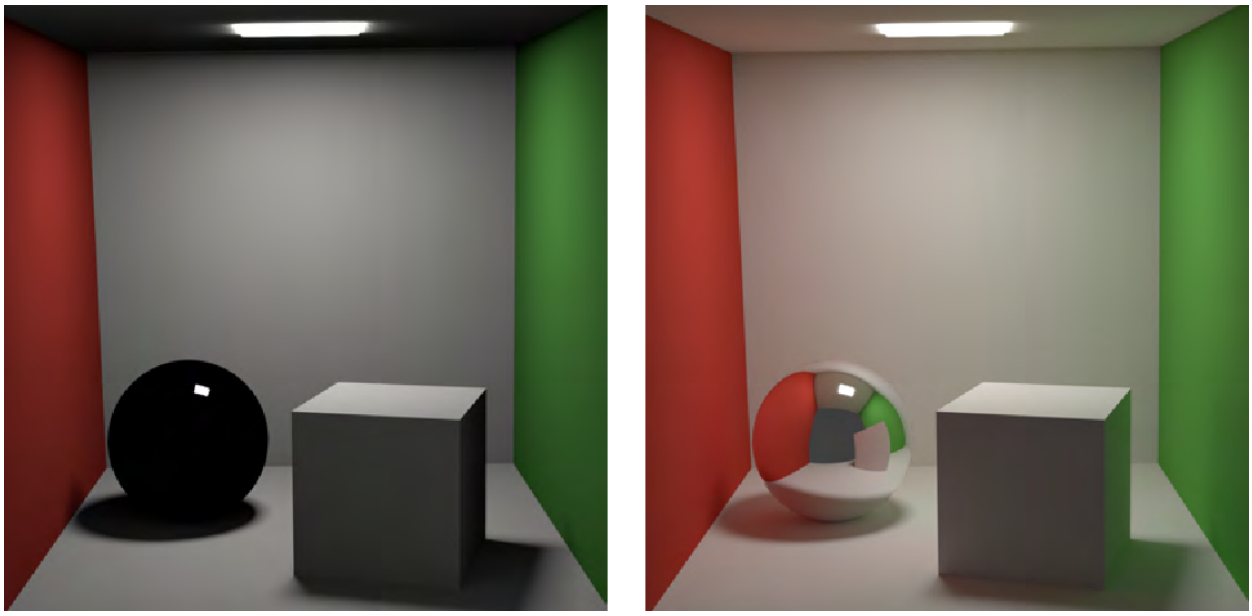

Figure 1 - Local lighting on the left and global lighting on the right.

Local illumination focuses on calculating the interaction of light in a limited context, typically considering only direct light sources and surface properties at a particular point in the scene. It uses simplified models such as the Fong reflection model, which includes ambient, diffuse, and specular reflection components. Ambient reflection represents uniformly diffuse light contributing to the overall brightness of the scene. Diffuse reflection describes the reflection of light from surfaces in various directions based on surface normals, while specular reflection defines reflection in specific directions, usually associated with glossy surfaces.

Local illumination, although a computationally efficient method of rendering scenes, has several limitations compared to global illumination. First, it lacks realism as it cannot capture the complexity of real lighting scenarios. This is because local lighting models do not take into account indirect lighting effects such as light reflection from objects and refraction, which are very important for achieving photorealistic rendering. Consequently, scenes rendered using only local illumination may look flat and lack visual saturation. Moreover, local illumination

algorithms only consider the direct interaction of light at a specific point in the scene, which limits their ability to model the interaction of light with surfaces beyond the immediate vicinity. This can lead to inaccurate depiction of lighting effects, especially in scenes with complex geometry and materials. In addition, shadows produced by local lighting models are often unrealistic, lacking softness and have sharp edges due to the lack of modeling of indirect lighting. Local illumination methods cannot accurately model global illumination phenomena such as color bleeding, where the color of one surface is affected by reflected light from neighboring surfaces. This limitation results in scenes that lack the subtle interplay of light and color observed in the real world. Materials with complex optical properties, such as those that are translucent or scatter subsurface light, also pose challenges for local illumination models because they require accounting for light scattering within the volume of the material.

Finally, achieving visually appealing results with localized lighting often requires extensive manual adjustment of lighting and material parameters, which can be time-consuming and does not always lead to satisfactory results, especially in complex scenes. As a result, although local lighting provides computational efficiency, it fails to capture the realism and complexity of real-world lighting compared to global lighting techniques.

Global illumination, on the other hand, aims to model the interaction of light in a scene in a more realistic way by considering the indirect effects of lighting. Unlike local lighting algorithms, global lighting algorithms trace light rays through the scene, taking into account multiple interactions, including reflection and refraction from surfaces, as well as scattering in different environments such as fog or water.

Such a realistic reproduction of a 3D scene, taking into account all the effects of multiple beam collisions on object surfaces, is a complex mathematical problem. This requires taking into account different types of reflections, such as diffuse, specular and directional-diffuse reflections, as well as considering the interaction of light with different materials and textures. One of the key tasks of

global illumination is modeling the shadows and reflections of a scene. This allows for more realistic images where shadows and reflections affect the overall visual perception of the scene. When global illumination is taken into account, the understanding of photometric quantities also changes.

For example, the luminosity of a surface now depends not only on its own radiation, but also on the light reflected from other objects in the scene. Important for physically accurate lighting is the use of realistic material models that take into account different types of surfaces and their interaction with light.

Global illumination includes taking into account all possible effects in the ray approximation, but in fact this task is impossible because of the infinite number of interacting objects in the Universe. And since most objects in a scene have an insignificant effect on lighting, they can be excluded and limited to meaningful collisions to construct lighting in a scene.

1.2 Methods of mathematical modeling of lighting

Global illumination in computer graphics is often implemented using ray tracing. This is a technique that is used to model the path of light in a three-dimensional scene. It works as follows: rays of light are released from each observation point (usually a camera), which pass through each pixel of the image and propagate through the scene. When the rays encounter objects in the scene, they may be reflected, refracted, or absorbed, depending on the properties of the objects' surfaces. Since global illumination describes the contribution of light that comes into the scene not only from direct light sources but also from reflected and scattered rays, it is possible to account for such complex effects of light interaction with the environment using ray tracing. To create global illumination using ray tracing, in addition to the rays released from the camera, additional rays are generated that represent reflected or scattered rays of light. These additional rays follow the direction reflected from the objects surface or a random direction to account for the scattered light. Various optimizations and approximations are often used to reduce the computational load and speed up the rendering process.

Rendering is a fundamental component of computer graphics, the process of converting the description of three-dimensional objects into an image. Algorithms for working with graphics must be passed through a certain stage of rendering so that their result can become visible in the image.

Rendering can be divided into two types: the first is real-time rendering, in which case the image is rendered at 30 frames per second or higher, and the second is deferred rendering, where there is no need to maintain a high frames per second rate and the frames are assembled into a single stream during the build stage. The first option is used to create interactive 3D graphics (games, simulators, augmented reality), the second - for 3D graphics in the film industry, visualization, advertising products. The main quality criterion for rendering is the photorealistic quality of the created image, and this directly depends on the selected rendering technology.

In rendering, a special equation appropriately named "Rendering Equation" is solved.

In 1986, James Kajiya presented a paper entitled "The Rendering Equation and its use in computer graphics" [1]. In seven pages, Kajiya described a way to mathematically calculate the physical properties of light, and it was this discovery that changed 3D rendering.

The rendering equation is an integral equation that defines the amount of light radiation in a certain direction as the sum of the own and reflected radiation [2]. The physical basis of the equation is the law of conservation of energy. Let L be the brightness along a given direction at a given point in space. Then the amount of outgoing radiation (Lo) is the sum of emitted light (Le) and reflected light. The reflected light can be represented as the sum of the incoming brightness (Li) in all directions multiplied by the reflection coefficient from a given angle.

The rendering equation is shown in the figure (Figure 2).

$$L_o(\mathbf{x}, \omega, \lambda, t) = L_e(\mathbf{x}, \omega, \lambda, t) + \int_\Omega f_r(\mathbf{x}, \omega', \omega, \lambda, t) L_i(\mathbf{x}, \omega', \lambda, t)(-\omega' \cdot \mathbf{n}) d\omega'$$

Figure 2 - Rendering equation.

Where:

λ - Wavelength of light.

t - Time.

Lo - Light with a given wavelength λ emitted in the direction ω at time t, from a given point x.

f - Bidirectional reflectance distribution function, the brightness reflected from ω' to ω at point x, at time t, at wavelength λ.

Le - Emitted light.

Li - Wavelength λ along the incoming direction to point x from direction ω' at time t.

(- ω' * n) - Absorption of brightness along a given angle.

∫ - Integral over the hemisphere of the incoming directions.

Prior to this, another important contribution was Parry Moon's (Parry Hiram Moon, 14.02.1898, Beaver Dam - 4.03.1988, Boston) derivation of the equation in simplified form: (Figure 3).

$$M(\mathbf{r}) = M_0(\mathbf{r}) + \frac{\sigma(\mathbf{r})}{\pi} \int_{\Sigma} M(\mathbf{r}')F(\mathbf{r},\mathbf{r}')\Theta(\mathbf{r},\mathbf{r}')d^2r'$$

Figure 3 - Emissivity equation.

Where:

M(r) - Luminosity of the point of surface r.

M0(r) - Luminosity of point r obtained directly from direct light sources.

Θ(r, r′) - Visibility function of the element d2r′ from point r.

F(r, r′) - Elementary form factor.

Its advantage was the dependence of the function M(r) only on the surface coordinates. It is determined not only by the surface's own luminescence, but also by the reflection of light incident from other surfaces, and therefore characterizes not only the surface itself, but also its position and the properties of other surfaces. In the literature it is called the radiative equation or radiosity.

The Global Illumination Equation (GEE) is a mathematical expression used in computer graphics to model illumination in three-dimensional scenes. It describes the distribution of brightness and color of surfaces in a scene, taking into account different light sources, reflections from surfaces, and light refraction. The CSO is an integral equation in which the desired brightness function of surface points depends on many factors such as the location of light sources, the properties of surfaces, and their interaction with each other.

The UGO can be represented as a Fredholm integral equation of genus II to determine the brightness field of points to the surface Σ taking into account multiple re-reflections (Figure 4), which accounts for all multiples of diffuse-mirror reflection, transmittance, and intrinsic emission, which is the subject of global illumination theory. In this (Figure 4) form, the UGO is written for brightness and describes the process of radiation transfer in a volume with multiple reflections in the five-dimensional space of coordinates and directions, the process of obtaining which is described in detail in the source [3].

$$L(\mathbf{r},\hat{\mathbf{l}}) = L(\mathbf{r},\hat{\mathbf{l}}) + \frac{1}{\pi}\int_{\Sigma} L(\mathbf{r}',\hat{\mathbf{l}}')\sigma(\mathbf{r};\hat{\mathbf{l}},\hat{\mathbf{l}}')F(\mathbf{r},\mathbf{r}')\Theta(\mathbf{r},\mathbf{r}')d^2r'$$

Figure 4 - Global illumination equation.

Where:

L(r, Î) is the luminance at point r in the Î direction.

σ (r;Î,Î′) - bidirectional scattering (reflection or refraction) function.

Θ(r, r′) - The visibility function of an element d2r′ from a point r.

F(r, r′) - Elementary form factor.

No analytical solutions of the UGO have been found, and numerical solution became possible only with the advent of fast computers. Among numerical solution methods, direct modeling, forward and backward ray tracing, finite element method (radiosity), photon map method, and others are used. At the same time, only the finite element method based on the diffuse approximation has a mathematically rigorous justification, and the other methods are mainly phenomenological in nature. Different algorithms and solution methods have both their advantages and limitations.

The successive approximation method is an iterative process in which the solution to the equation is refined at each step. This method is the most stable for solving complex equations such as the UGO and achieves high accuracy of the result.

The radiosity method is a numerical approach to modeling illuminance in three-dimensional scenes. It is based on representing the scene as a set of finite beams of rays that emanate from light sources and propagate through the scene, interacting with the surfaces of objects. The method often incorporates iterative solution methods such as Jacobi and Gauss-Seidel methods [4]. These methods allow the radiative equations to be approximately solved with high accuracy. The convergence of such methods depends on the physical properties of the system, in particular the reflection and absorption coefficients, and can be ensured if certain conditions are met, such as the condition kij < 1, where kij is the radiative transfer

coefficient from element i to element j. It is also important to determine the sequence of elements to be computed at each iteration to improve the convergence rate of the method. The general scheme of the emissivity method involves representing the scene as a grid of faces, calculating the form factors between them, solving the emissivity equation using iterative methods, and finally projecting the results onto the screen to obtain an image of the scene. To improve the image quality and eliminate the "facetedness" of the scene, it may be necessary to apply algorithms to approximate the emissivity values and distribute the unallocated luminous flux between the facets of the scene. In the context of this method, the illuminance of a scene is described in terms of radiative energy that propagates through space. The method is particularly effective for diffuse surfaces where the luminance is not too dependent on the viewing angle. As a result, it can model the illuminance of surfaces quite accurately and produce 3D images of scenes with acceptable quality in a reasonable amount of time. The disadvantage is the inability to account for specular reflections of light. The method is often based on the finite element method.

The finite element method (FEM) is based on the idea of approximating a continuous function by a discrete model consisting of piecewise continuous functions defined on a finite number of subareas (finite elements). The elements of the geometric domain are partitioned in such a way that on each of them the unknown function is approximated by some function. These functions must satisfy the boundary conditions of the problem. FEM allows to obtain an approximate solution of the problem with high accuracy by solving [5] a system of equations describing physical phenomena in each element.

1.3 Solution of global illumination equation by ray tracing method

The ray tracing method is often used for numerical solution of the UGO, which allows modeling the path of light rays from illumination sources to the camera or observation point. In [6], the authors present a solution. Its essence is that the global illumination equation can be represented in an operator form, which facilitates its numerical solution. In this form, an integral operator is used, which expresses the relationship between illuminance in a given point and its values in neighboring points. For the numerical solution of the equation, an iterative method (Figure 5) is used, based on the decomposition of the solution into a Neumann series. Each iteration of the method corresponds to the next approximation of the solution, which allows to refine the result with each step. The iterative ray tracing method uses quadrature formulas to compute integrals, replacing integrals by sums. This allows numerical estimation of illuminance at each point of the scene.

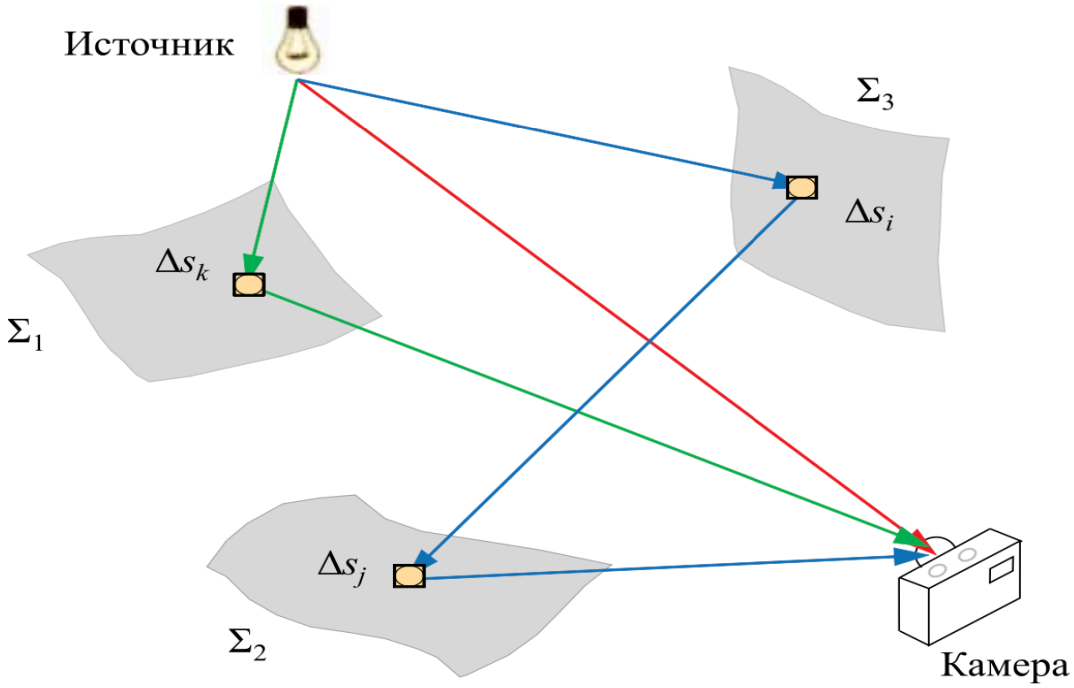

Figure 5 - Geometric interpretation of the UGO solution using the iteration method.

Two basic approaches can be used in ray tracing:

- The forward ray tracing method, which consists of launching rays from light sources and tracing their path to the camera or observer.

- The backward ray tracing method is to launch rays from the camera or observer and trace their path through the scene. This allows us to determine which objects in the scene are illuminated and how the light propagates through the scene. This method provides a more efficient use of computational resources because only rays that fall within the observation area are considered. It is used most often.

The figure (Figure 6) demonstrates the forward and backward tracing methods.

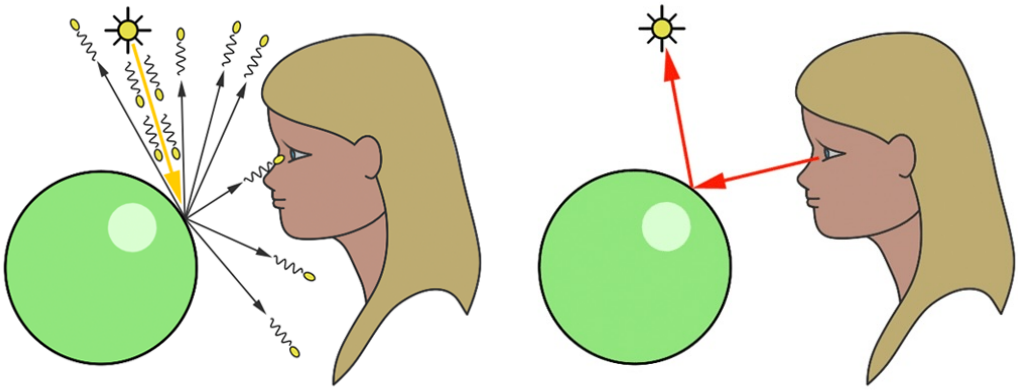

Figure 6 - Beam direction at direct tracing on the left and at reverse tracing on the right.

The process of ray trajectory construction in reverse tracing starts with determining the ray direction from each pixel on the camera CCD to the optical center of the scene. The ray then searches for the point of intersection with the object surface, which is described by mathematical equations. Depending on the shape of the surface (flat, polygonal, spherical), the intersection conditions will differ. Having detected the surface, the beam takes into account its peculiarities for correct determination of the intersection point and calculation of illumination. At this point, phenomena such as reflection and refraction of the ray occur.

When a ray of light crosses the boundary between two media, it may change its direction.

Refraction occurs when a ray of light passes from one medium to another medium that has different optical properties such as refractive indices. This change in beam direction is explained by Snellius' law [7], which establishes the relationship between the angles of incidence and refraction of the beam, as well as the refractive indices of the media. To understand where exactly the ray will be directed after refraction, the vector approach is used. This allows the new beam direction to be expressed as a combination of vectors describing the original beam and the normal to the surface through which refraction occurs. This method helps to predict how the direction of the ray will change after crossing the boundary between the media.

Reflection occurs when a ray of light reflects off a surface without penetrating the medium. The law of reflection states that the angle of incidence of a ray is equal to the angle of reflection. This phenomenon can also be explained using the vector approach, which allows us to determine the new direction of the ray after reflection from the surface. The figure (Figure 7) demonstrates the change of the ray trajectory during reflection and refraction.

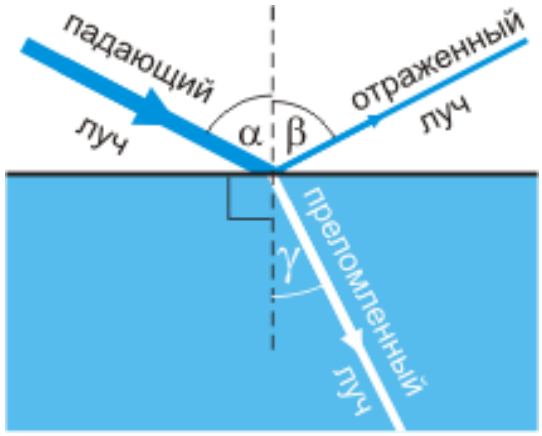

Figure 7 - Reflection and refraction of a ray at the boundary of two media.

After interaction with a surface, a whole fan of rays can be formed from a single ray. This process can be visualized as the creation of secondary light sources on each surface. In this case, as the number of scatterings increases, the number of calculations increases geometrically. Under such conditions, even a powerful computer may not be able to cope. However, many light trajectories start to

become similar to each other, allowing only the most typical ones to be calculated. This leads to the idea of stochastic ray tracing, which uses trajectory statistics to average the results. To efficiently compute such integrals, the Monte Carlo method is used in practice, which allows us to obtain an approximate value of the integral, regardless of the complexity of the integrand function.

1.4 Monte Carlo method

The Monte Carlo method is used to numerically solve various problems based on randomness or probability. It is used in a variety of fields including physics, finance, biology, engineering and many others. The principle behind the Monte Carlo method is to generate random numbers and use them to approximate the numerical solution of a problem. For example, for calculating integrals, determining probabilities, modeling physical systems, etc. It is important to note that the accuracy of the Monte Carlo method depends on the number of random numbers generated in the process, so the more numbers used, the more accurate results can be obtained. This method allows the estimation of system characteristics that may be difficult to solve analytically or require too much computation.

The term "Monte Carlo method" appeared in 1949 in an article by Nicholas Metropolis and Stanislaw Ulam in the Journal of the American Statistical Association [8]. The name of the method comes from the capital of the principality of Monaco, famous for its casinos, where roulette is one of the popular random number generators. Stanislaw Ulam mentions that he suggested the name to Nicholas Metropolis in honor of his uncle, a gambler. The ideas behind the method were developed in the 1930s by scientists such as Enrico Fermi and John von Neumann, and then refined by Stanislaw Ulam at Los Alamos in the 1940s. They realized that the stochastic approach could be applied to approximate the multivariate integrals in the transport equations that arise when considering the motion of neutrons in an isotropic medium. Andrei Kolmogorov also contributed to the theory by proving the connection between Markov chains and integro-differential equations.

The concept of probability has its roots in gambling. The first attempts to solve combinatorial problems date back to the 10th century, when Bishop Wiebold

of Cambrai solved the problem of possible variants of throwing three dice, counting 56 variants. However, this number did not reflect the equivalence of the different outcomes. In the 13th century Richard de Fornival amended this reasoning, taking into account, for example, that the same number of points on the three dice could be obtained in different ways. This established that the total number of equally probable options is 216. The probability of an event is defined as the ratio of the area of favorable outcomes to the total area of possible outcomes.

In practice, in the context of random phenomena, we often have to deal with random variables. According to probability theory, a random variable can be continuous or discrete.

- A continuous random variable describes phenomena that can take an infinite number of values within a certain range. For example, the time taken for light to travel through a certain medium can be represented as a continuous random variable.

- In contrast, a discrete random variable takes only a finite or countable number of values. For example, the result of a die roll or the number of defective parts on an assembly line.

The Monte Carlo method is based on three basic statements of probability theory and mathematical statistics:

1. Chebyshev's inequality and the law of large numbers.

This provision allows to numerically calculate the mathematical expectation as the arithmetic mean of a sample. According to Chebyshev's inequality, the mean of a random variable from a sample will be closer to the mathematical expectation if the number of experiments is increased.

2. Moivre's central limit theorem.

It allows us to estimate the distribution of a random variable when the number of independent experiments is large. The central limit theorem of Moivre-Laplace says that a sequence of random variables converges to a normal distribution.

3- Mathematical Statistics.

All the reasoning of probability theory applies to infinite spaces, but in practice one deals with finite samples of random variables. Here, the efficiency of estimators that account for variance and unbiasedness is important.

When analyzing finite samples, the arithmetic mean is used to estimate the expectation and variance. This is confirmed even though some mathematical calculations emphasize that replacing n elements in the sample by n-1 does not significantly change the calculations, but may be important for small amounts of data.

The Monte Carlo method for calculating integrals is based on the use of random numbers and mathematical expectations. The principle of the method is to generate random points in the area of integration and calculate the values of the function at these points. Then the average value of the function is calculated as the arithmetic mean of the function values in the generated points multiplied by the area of the integration area. The Monte Carlo method requires generating random points according to a certain probability distribution in the area of integration. The function values are then evaluated at these points and their mean is calculated. The main advantage of the Monte Carlo method is that it can be used to compute integrals in multidimensional spaces and for functions that are difficult to integrate analytically.

The Monte Carlo method is used in the homogeneous Markov process method of ray wandering to model the propagation of light in a scene. It is based on random walks, where each ray of light is represented by a sequence of random steps. The probability of transition from one point to another is defined by a function, which allows us to approximate integrals describing the distribution of light brightness in the scene. Each step in the beam wandering is modeled by

choosing a starting point, determining the direction and the light intensity distribution. The light scattering coefficient can be accounted for using statistical weights, which allows to take into account various optical effects such as reflection, refraction and absorption of light on a surface.

1.5 Analytical solutions of the global illumination equation

Analytical solutions to the UGO provide a better understanding of the physical processes involved in illumination and are also used to test algorithms and programs for solving the equation. To date, only two analytical solutions to the radiosity equation have been found:

1. a point isotropic source inside a photometric sphere [9]: this solution describes the illumination produced by a point light source located inside a photometric sphere. The photometric sphere is an optical component that is a hollow spherical cavity with a diffuse white reflective coating. The solution allows us to determine the illuminance of the sphere surface as a result of multiple reflections of light from the inner surface of the sphere.

2- Point isotropic source between two parallel homogeneous diffuse surfaces: this solution allows to express analytically the illuminance on both surfaces depending on the distance to the source and characteristics of the surfaces.

The origin of the second variant is the Problem of V.V. Sobolev [10], related to the use of white camouflage for light camouflage of troops in winter, which arose during the WWII. Sobolev developed an analytical approach to the solution of this problem related to the use of photo illuminating aerial bombs (FOTAB) to illuminate the terrain during night aerial photography. A FOTAB is an aerial bomb that produces a powerful short-term light flash. It illuminates the terrain for aerial photography at altitudes up to 10,000 meters at flight speeds up to 1,000 km/h. The main charge of the FOTAB consists of a powdered aluminum-magnesium alloy that ignites and burns using oxygen in the air, producing a flash with a light intensity of more than 2 billion candela. The problem considers a point isotropic light source between two parallel diffuse surfaces, which mimics the use of FOTAB to illuminate an area of heavily cloudy winter.

In [11], the process of obtaining an analytical solution is described. The process consists of taking the equation for the illuminance of each of the planes between which the source is located to form a system of equations, which is

analyzed using the Fourier transform. The equations then take the form of functions expressed through the Hankel transform. The solution of the system of equations and the subsequent inverse Fourier transform allows us to obtain the irradiance distribution over the plane, taking into account both the direct illumination from the source and the multiple re-reflections of light between surfaces.

The final equation can be seen in the figure (Figure 8):

$$E_1(r) = \frac{h_1}{(h_1^2 + r^2)^{3/2}} + \rho_2 \int_0^\infty \frac{\mathrm{e}^{-h_2 k} + \rho_1 \mathrm{e}^{-h_1 k}\, k\, \mathrm{K}_1(k)}{1 - \rho_1 \rho_2 k^2\, \mathrm{K}_1^2(k)} \mathrm{K}_1(k)\, \mathrm{J}_0(kr) k^2 dk$$

Figure 8 is the exact analytical solution of the emissivity equation.

Where:

Ei(r) - Illuminance of the i-th plane (i=1,2).

ρi - Coefficient of the i-th plane.

hi - Distance from the i-th plane to the source.

h1+h2=1.

K1(k) - Modified Bessel function of purely imaginary argument or first order Mac-Donald function.

J0 - Bessel function of zero order.

1.6 Local estimates

The local estimation method is based on the construction of a Markov chain that describes the movement of light rays through the scene. At each iteration the brightness at the point of interest is calculated, taking into account the illumination direction and surface characteristics. This method allows us to efficiently estimate brightness at different points in the scene, including multiple reflections of light. This approach provides a mathematically rigorous solution for the CSO, which opens new possibilities for creating software tools to model the light field in realistic three-dimensional scenes.

The UGO (Figure 4) is not convenient for statistical modeling, because the required function under the integral stands at the point r′ and is defined at the point r. Thus its variables r′ and l′ are related by a relation. By integrating over space and replacing reflections with diffuse reflections, the above mentioned Radiance equation is obtained (Figure 3). As a result, the estimate of Iφ for L(r, Î) takes the form shown below (Figure 9). This expression is called the local estimate of the Monte Carlo method [12].

$$I_{\varphi} = \mathrm{M}\sum_{n=0}^{\infty} Q_n k(\mathbf{r},\mathbf{r}')$$

Figure 9 - Local estimation.

Where k(r,r′) is the Kernel of the CSO.

Qn - Weight of the Markov chain.

M - Mathematical expectation of different random ray trajectories.

The Markov chain models a sequential series of random rays wandering through the scene. The weight of the first ray Q1 is determined by the initial

brightness of the source L0 (r, Î). At the same time, k(r,r′) determines the transition probability from a point in the Markov chain to a given point r. [12]. From all nodes of the trajectory where its intersection with the scene surface occurred, the contribution to the scene illumination at point r is calculated based on the kernel of the equation k(r,r′) in the expression (Figure 9). In doing so, all probability distributions must meet the normalization conditions. The values L0(r, Î) and k(r,r′) will determine the statistical weights of the Markov chain Qn. In the case of a diffuse model, each successive statistical weight will be multiplied by the reflection coefficient.

Local estimation allows the estimation of illuminance at a given point in the scene. Thus, to calculate the illuminance at some given point r it is necessary to construct a Markov chain and at each reflection act calculate the kernel k(r,r′) for all given points. The mathematical expectation of the obtained value and will be equal to the illuminance.

Markov chain is a sequence of random events with a finite or countable number of outcomes, characterized by the fact that at a fixed present the future is independent of the past. In this case, the chain is constructed from a light source, with the initial weight put equal to the light intensity of the light source, taking into account normalization to the flux. Then the point of intersection of the beam with the scene element is found and the weight is multiplied by the reflection coefficient. Then the UGO kernel is calculated for each of the investigated points, multiplied by the weight of the beam, and summarized with the previous values. The beam is then cast according to the diffuse reflection law and the next intersection is searched for. The process is repeated until the ray either leaves the scene or its weight is below a specified threshold. By casting a large number of rays and averaging them, the illuminance values at the investigated points are obtained. Local estimation allows to calculate the illuminance value in several points at once by one ray. This is a fundamental difference from direct modeling and ray tracing [13].

The use of local estimation demonstrates a significant acceleration of calculations in comparison with direct modeling. In the same paper, a comparison of local estimation and direct modeling in solving the Sobolev problem is given, where the method produces results 80-90 times faster.

The expression in (Figure 10) is called Dual Local Estimation and is a method that extends the capabilities of local estimation by allowing direct modeling of the brightness at a given point in the scene along a given direction. This method is based on constructing a Markov chain describing the movement of light rays and representing the solution to the global illumination equation as a Neumann series.

$$I_{\varphi} = \mathsf{M}\sum_{n=0}^{\infty} Q_n k(\mathbf{r},\hat{\mathbf{l}},\mathbf{r}',\mathbf{l}') ,$$

$$k(\mathbf{r},\hat{\mathbf{l}},\mathbf{r}',\hat{\mathbf{l}}') = \frac{1}{\pi^2}\sigma(\mathbf{r}';\hat{\mathbf{l}}'',\hat{\mathbf{l}}')\sigma(\mathbf{r};\hat{\mathbf{l}}',\hat{\mathbf{l}})F(\mathbf{r}',\mathbf{r}'').$$

Figure 10 - Dual local estimation.

In this expression, the angular feature disappears as a result of integration, and the independent variables r, Î, r′ , Î′ , Î′ ′ , r′ ′ , Î′ ′ correspond to the geometry of the beam propagation [12], which leads to the possibility of modeling the UGO and calculating the brightness at a given point in a given direction from multiples higher than the first one.

To date, dual local estimation is the only method that allows us to obtain brightness values at any point in the space of a 3D scene [13].

However, the practical implementation of the local estimation method faces problems related to singularities in the CSO kernel. These singularities occur when the point where the beam collides with a surface is very close to the brightness estimation point, resulting in "spikes" of distortion in the estimation. Such situations can occur near the edges of neighboring faces when using a mesh

representation of scene objects. This means that a direct implementation of the local estimation method in the form of a program for modeling brightness transfer in complex scenes becomes less effective due to the probability of increasing "bursts" of distortions with increasing scene complexity.

In [14], the authors present a method of averaging the desired value in some neighborhood of the point under study. For example, for a point isotropic light source in the diffuse approximation, the local estimate of illuminance at a point is determined by an integral over the volume of the neighborhood of the point under consideration. Since the number of rays and collisions is finite, this integral can be rewritten and its value can be estimated by statistical testing.

The proposed method of singularity elimination opens new opportunities for the development of local Monte Carlo method estimations in the field of brightness transfer modeling and its implementation in the software for illumination modeling can significantly improve the quality of visualization.

Based on all the above, we can conclude that the method of Local Estimation becomes a new vector in the development of lighting calculations. Double local estimation for the first time in lighting practice allows to calculate directly the brightness at a given point along a given direction. This opens fundamentally new horizons of lighting design and the possibility of revising the existing regulatory framework based on existing modeling methods. Double local estimation can become a new numerical method used in realistic visualization of three-dimensional scenes.

1.7 Conclusions of the section

The development of computer graphics in the field of lighting modeling has come a long way. In this paper, historical events about simple methods of drawing lines and surfaces have been mentioned, and complex algorithms that take into account the physical properties of light have been considered, leading to significant progress in creating realistic images.

Various methods were considered to solve the global illumination equation, which is the foundation in calculating the brightness in a scene. The ray tracing method, which models the path of light rays from light sources to the camera, allowing the creation of images with a high degree of realism. The radiative method, based on representing the scene as a set of finite beams of rays, also allows modeling the illumination of surfaces, but does not take reflections into account.

Last but not least considered were local estimation methods, in particular dual local estimation, which represent effective directions in light field modeling. They open up new possibilities for lighting design and visualization by allowing direct calculation of brightness at a given point in a scene along a given direction, which has commercial applications but has not yet been used by anyone in full-fledged software.

In this part the mathematical side of visualization has been considered, but in addition, it is necessary to mention the tools and the process of practical implementation of the above-mentioned methods and algorithms, where the complexities of computation and optimization appear. The next section of this paper is devoted to it.

2 Tools for light modeling and visualization

2.1 Embree library

Ray tracing is a powerful method of creating realistic images in computer graphics, and there are many libraries and tools for its implementation. One such library is Intel Embree. Embree is a high-performance ray tracing library developed by Intel. It provides an API for implementing ray tracing on Intel processors using various optimization techniques, making it an ideal choice for a wide range of applications including computer games, virtual reality, animation and visualization [15]. One of the main features of the Embree library is its high performance. It is optimized to run on Intel multi-core processors and uses efficient algorithms and data structures for fast ray processing. This allows you to achieve high rendering speeds even when working with large and complex scenes. In addition, Embree provides various options for optimizing ray tracing and supports different types of materials and light sources, making it a versatile tool for creating a variety of visual effects. The library integrates with various computer graphics software packages and frameworks such as Blender, Maya, Unity, and Unreal Engine, ensuring its widespread use in the industry. For efficiency and speed, Embree has BVH (Bounding Volume Hierarchy) structures to efficiently manage scene complexity and speed up the process of finding ray intersections with objects.

BVH is a tree-like data structure used to speed up the search for ray intersections with objects in a scene. It partitions the scene space into a hierarchy of bounding volumes, such as spheres or parallelepipeds, which allows you to quickly cut off parts of the scene that do not intersect with rays.

When constructing a BVH, a root bounding volume is first created that encompasses the entire scene. This volume is then recursively split into two subvolumes, which in turn are also split until a certain depth of separation is reached or a stopping criterion is met. As a result, each leaf of the BVH tree contains references to objects in the scene or to their bounding volumes. One of the

advantages of BVH is its ability to quickly and efficiently cut off parts of the scene that do not intersect with rays, which significantly reduces the number of intersection checks and speeds up the ray tracing process. In addition, BVH has good scalability and can be effectively used for scenes with different object types and geometries.

In the figure (Figure 11), the recursive volume partitioning described above can be clearly observed.

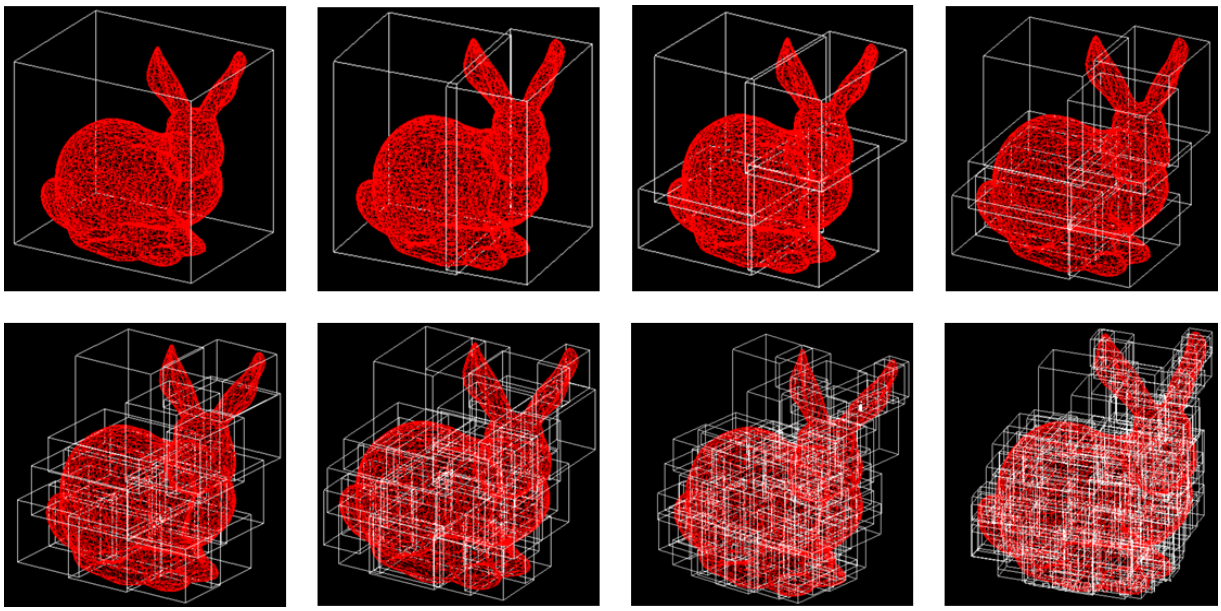

Figure 11 - Demonstration of recursive division in BVH.

In the study [16], the author experiments with different methods of BVH construction and evaluates the performance of several splitting methods and describes ways to visualize the result obtained. One of the considered partitioning methods, called the median partitioning method, resulted in trees with worse quality results than the other methods when using highly polygonal meshes. On the other hand, the use of Surface Area Heuristic (SAH) resulted in trees with better quality of results, but took more time to build the BVH.

Based on the above, we can conclude that the Embree library developed by Intel is a high-performance ray tracing library that includes optimized implementations of ray tracing algorithms and BVH support. It provides an API

for fast and efficient rendering of scenes using ray tracing. One of the key features of Embree is its optimization for the architecture of modern multi-core processors, which allows for high performance when rendering complex scenes. This makes Embree an ideal choice for a wide range of applications including computer games, virtual reality, animation and visualization. The library integrates with a variety of computer graphics software packages and frameworks, ensuring its widespread use in the industry.

2.2 Options for optimizing the visualization process

The rendering process includes several stages, each of which plays a different role in forming the final image:

1. Scene Preparation. In this stage, the objects, lights, and camera of the scene are defined. Objects can be manually created or imported from other software or files. Light comes from various sources in the scene such as light sources, reflected light and global illumination. The camera is positioned to determine the viewpoint of the scene.

2. ray tracing. This is a key step in the rendering process where an image is created by tracing rays from the camera through each pixel of the image into the scene. Each ray may intersect with objects in the scene, be reflected, refracted, or absorbed, depending on the properties of the objects' surfaces. The result of this step is the illumination for each image pixel.

3- Shadow processing. This step determines which parts of the scene are in shadow and how this affects the illumination of objects. This involves checking the visibility between light sources and objects in the scene.

4. Material Computation. Each object in the scene usually has different materials that determine how they reflect and transmit light. This step involves calculating the contribution of each material to the lighting of the scene and its interaction with the environment.

5. Post-processing. Finally, the image undergoes post-processing, which may include applying filters, color correction, effect overlay, etc. to improve its visual quality and achieve the desired visual effect.

There are inherent problems in the process of rendering an image in computer graphics. The main ones are low rendering speed and presence of noise in the image.

The rendering speed can suffer due to the complexity of the scene itself. When the scene contains many objects with complex geometry or a large number of light sources, ray tracing becomes more time consuming. Each ray must traverse

the scene, interacting with each object, which requires significant computational resources. The long rendering process slows down the development and testing of graphics projects, limits the developer's (artist, designer) ability to visualize their ideas, and slows down the workflow.

Also, when using ray tracing methods, there is an effect of image graininess inherent in the statistical nature of the Monte Carlo method, which is also called noise. Image noise can occur due to insufficient number of samples used to estimate the illumination at each point in the image. Poor quality sampling results in a grainy or blurred image due to improper estimation of complex illumination effects. These problems can be exacerbated by the use of complex materials or inefficient rendering algorithms.

There are many methods and techniques to mitigate these problems.

To address speed and speed up the process, the following are used:

1. Parallel computing on multiprocessor systems. The use of multiprocessor systems allows parallelization of computation and faster data processing.
2. Optimization of ray tracing algorithms. By optimizing ray tracing algorithms, rendering time can be significantly reduced. This includes, for example, methods for accelerating the intersection of rays with objects and efficient memory management.
3. using approximate rendering methods. Approximate rendering methods, such as deep learning algorithms or precomputed illumination methods, can significantly reduce rendering time without significant loss of quality.
4. The use of hardware acceleration technologies such as graphics processing units (GPUs). Using them can significantly speed up the process, especially if they are specialized hardware such as, for example, NVIDIA RTX technology.

The following are used to eliminate noise:

1. Method with increasing number of ray samples. It is one of the common methods and involves generating more rays for each pixel of the image,

which allows more accurate estimation of illumination and reduces the statistical noise characteristic of Monte Carlo methods.

2- Using approximate rendering methods. This is also an effective way to reduce the time required to generate an image by reducing the computational complexity of the algorithm. These methods are the incorporation of various optimizations such as adaptive sampling or approximation of complex illumination phenomena.

3. Denoizers. Various denoising algorithms are used to remove noise in images to improve the quality of images without having to increase the number of samples. Denoizers work by analyzing and filtering noise while preserving image details.

There are several types of denoising algorithms. In the paper [17], the author discusses the different variations. The following types can be distinguished:

1) SVGF. Spatio-Temporal Variance Guided Filter (SVGF) is a denoizer that uses spatio-temporal reprojection along with feature buffers such as normals, depth and dispersion calculation to drive a bilateral filter to blur high variance regions. A variation of SVGF technology is used in the computer game Minecraft RTX with the addition of irradiance caching, use of ray lengths for better reflection, and split rendering for transmissive surfaces such as water. SVGF, while very efficient, has a time lag that is noticeable in-game.

2) A-SVGF. Adaptive variance controlled spatiotemporal filtering (A-SVGF) improves SVGF by adaptively reusing previous samples that have been spatially reprojected according to temporal heuristics such as variance variation, viewing angle, etc. encoded in the moment buffer and filtering with a fast two-way filter. Thus, instead of accumulating samples based on history duration, the moment buffer acts as an alternative heuristic that uses variance variation to determine the proportion between old and new samples, resulting in reduced ghosting. Whereas in SVGF the moment buffer was only used to control blurring, in A-SVGF it is used in both the filtering and accumulation stages.

3) ReSTIR. Spatiotemporal Importance Resampling for Many-Light Ray Tracing (ReSTIR) attempts to move the spatiotemporal reprojection step in real-time denoizers to earlier rendering steps by using the probability statistics of neighboring samples.

4) DLSS. Deep Learning Super Sampling (DLSS) is a resolution enhancement technique that uses a small color buffer and direction map to increase the resolution of the output image by a factor of 2-4. It is an exclusive technology for pre-approved developers by NVIDIA, so there is currently no way to use it openly, but there is an alternative called DirectML's SuperResolution Sample.

5) Intel OIDN. Intel Open Image Denoise (OIDN) is a machine-learning auto-encoder that takes the albedo, first jump normals, and input noisy image and outputs a filtered image.

6) NVIDIA Optix 7 Denoising Autoencoder takes the same raw data as OIDN, additional albedo, normals and input noisy image, and outputs the filtered image much faster than Intel's solution, but with a loss of quality.

From the information above, we can conclude that using neural network based denoizers in image rendering has a number of significant advantages. They are able to effectively reduce noise in images, allowing for cleaner and higher quality results without the need to increase the number of samples, which reduces the time required to generate an image. This is especially important in the case of Monte Carlo method, where increasing the number of samples can significantly increase the computational load. Also, the use of denoising on neural networks can be more efficient and flexible than traditional denoising methods, as neural networks are able to automatically adapt to different types of noise and images, allowing for quality results on different scenes.

2.3 Using neural networks for denoising

There are several different types of neural networks that can be classified according to certain features. One of the most common ways of classification is by the type of connections between neurons. According to this approach, the following types of neural networks can be distinguished:

1. Multilayer Perceptrons (MLP) - this is the simplest type of neural network, which consists of one or more layers of neurons. Each neuron processes the incoming data and passes the output data to the next layer of neurons. MLP can be used for a wide range of tasks from time series prediction to pattern recognition in images.

2. Convolutional Neural Networks (CNNs) are a type of neural networks that are often used in image and video processing tasks. They are based on the principle of convolution, which allows the detection of visual features of an image. Each neuron in a CNN processes only a small region of the image, which allows the local properties of each fragment to be taken into account.

3. Recurrent Neural Networks (RNN) is a type of neural network that is well suited for dealing with sequential data such as text or sound. Unlike MLPs and CNNs, RNNs retain information about previous characters in a sequence, which allows them to take into account the context and sequence of the input data.

4. Deep Neural Networks (DNN) Deep Neural Networks (DNN) is a type of neural network that consists of multiple layers, each layer containing many neurons. Deep Neural Networks are widely used for working with large amounts of data, for example, in speech recognition or natural language processing tasks.

From the above, we can conclude that CNNs and MLPs are the most suitable choices for working with images.

In the paper [18], the author discusses a comparison between traditional multilayer perceptrons (MLPs) and convolutional neural networks (CNNs), identifies two major problems with MLPs and clearly demonstrates why CNNs are a better choice for image processing.

1. The problem with loss of spatial orientation of the image (Figure 12): MLP uses a one-dimensional image representation, which makes it difficult to recognize images and detect anomalies in spatial orientation. For example, in images represented in one dimension, changes in the relative position of key elements (e.g., nose, eyes, ears) make it difficult to identify an object. In contrast, CNN uses a two-dimensional image representation, which preserves spatial orientation and enables better object recognition.

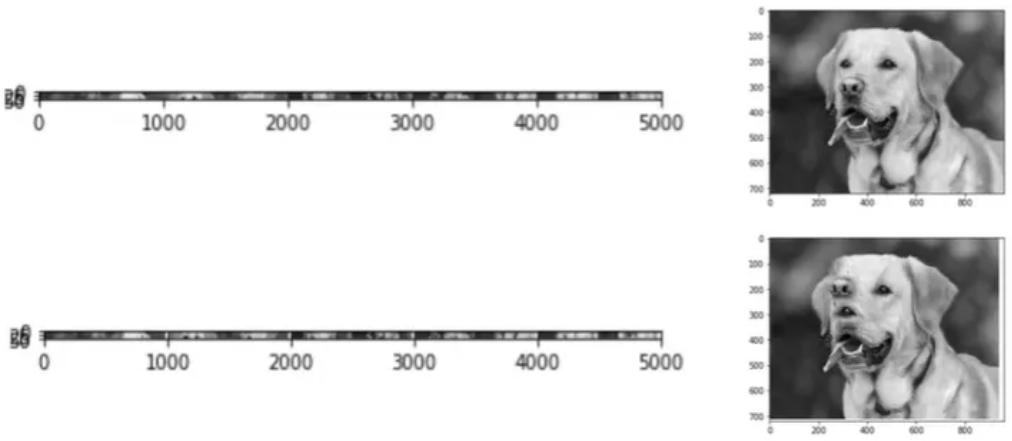

Figure 12 - Loss of spatial orientation of the image in MLP at the bottom, and no problem in CNN at the top.

1. Problem with parameter learning in neural network: MLP requires a huge number of parameters to efficiently handle large size images, which leads to high computational complexity and undesirable resource utilization. CNN addresses this problem by using the concept of convolution, which reduces the number of trained parameters while maintaining high performance in image recognition and classification.

In this regard, CNN is presented as a better solution for image processing because it preserves spatial orientation and efficiently manages the number of trained parameters, making it more suitable for a wide range of computer vision and image processing tasks.

The working principle of the neural network directly in the denoising task will consist of the following steps:

1. The first step requires deciding on a suitable neural network architecture, which should be capable of extracting image features and processing them for efficient noise removal. Based on the above study this will be CNN. It is well suited for image processing due to its ability to automatically learn the spatial structure of the data.
2. Next, the neural network is trained on a dataset containing pairs of noisy and clean images. During training, the network seeks to minimize the difference between the input noisy images and their corresponding clean versions. It does this by adjusting the weights and parameters of its layers using optimization techniques such as stochastic gradient descent.
3. A dataset containing pairs of noisy and clean images is used for training. These images can be collected from different sources or created programmatically. It is important that the clean images match the noisy images so that the neural network can learn to reconstruct clean versions of the images from the noisy input data.
4. A loss function is defined to estimate the error between the predicted and true clean images. Typically, the mean square error (MSE) is used for the denoising task, which measures the standard deviation between the pixel values of the predicted and true images.
5. In the training process, the neural network takes noisy images as input and gradually adjusts its weights and parameters by minimizing the loss function. This is accomplished using an error back propagation algorithm that propagates an error gradient through the network and adjusts the weights of the neurons according to this gradient.

6. After training is completed, the results of the neural network are evaluated on a separate set of test data that did not participate in training. This allows us to evaluate the quality of the network's performance on new, real data and determine its ability to effectively remove noise.

7. During training, convolutional neural network layers extract different levels of image abstraction, ranging from low-level features such as boundaries and textures to higher-level features related to image content and structure. In convolutional neural networks (CNNs), widely used in computer vision, feature extraction is accomplished through the application of convolutional layers.

8. Convolutional layers consist of filters (or kernels) that slide over the input image, performing a convolution operation. These filters are trained to find various low-level features such as boundaries, textures, and color features. In doing so, each filter highlights certain characteristics of the image by enhancing significant pixels and suppressing irrelevant ones.

9. The next step in CNN is the subsampling (pooling) operation, which reduces the dimensionality of the image and creates invariance to minor spatial shifts of the sought features. This is done by applying maximum or average subsampling operation in certain regions of the image.

10. At higher levels of the network, deeper convolutional layers begin to extract more complex and abstract features such as shapes, structures and objects. At this stage, each neuron in the convolutional layer is a combination of lower-level features that form higher-level abstractions.

11- The feature extraction stages in CNNs provide a gradual transition from extracting simple, low-level features to more complex and abstract image features. This process allows neural networks to efficiently model and analyze complex visual data structures and apply them including denoising.

12. After training is completed, the neural network designed to filter out noise in images is applied to the new noisy input data. This process starts by passing the noisy image to the network as input data. Then, the neural network predicts a clean

version of the image based on the previously learned features obtained during training.

13. In the noise filtering process, the neural network applies various mathematical operations to each pixel of the input noisy image. These operations include convolutions, which allow the network to analyze the neighborhood of each pixel using filters of different sizes.

14. Activation functions are then applied that introduce nonlinearity into the output of each layer, providing the network with the ability to learn more complex relationships between input and output data.

15. Data aggregation is performed using techniques such as pooling or merging, which reduce the dimensionality of the data while preserving the basic characteristics of the image. This helps to reduce the computational complexity and improve the generalization ability of the network.

16. During noise filtering, the neural network applies complex algorithms that take into account the context and structure of the image. This means that the network analyzes not only individual pixels but also their surroundings to account for spatial dependencies and textural features. This approach allows the network to effectively remove noise while preserving important image details.

The quality of denoiser performance is evaluated using various metrics such as PSNR (Peak Signal-to-Noise Ratio) and SSIM (Structural Similarity Index Measure). These metrics help in determining the degree of similarity between the original clean image and the reconstructed denoiser.

PSNR expresses the ratio between the maximum possible signal value and the RMS error between the original and the reconstructed image, providing a quantitative measure of image quality. The higher the PSNR value, the smaller the differences between the original and reconstructed images.

SSIM, on the other hand, measures the structural similarity between two images by considering not only brightness but also textural features. It evaluates three aspects: brightness, contrast and texture. A high SSIM value indicates a high similarity between images, which indicates that details and textural features are preserved.

Both metrics help in determining the effectiveness of denoiser in removing noise and preserving image details. They are used to compare the performance quality of different denoiser models and also to optimize the training parameters of the networks to achieve the best results in filtering noise in images.

In the paper [19], the authors discuss the study of the effect of the nature of noise on the quality of noise removal using neural networks.

The assumption is that training a neural network on the separated noise rather than on the noisy image can lead to more efficient noise removal. This may reduce the likelihood of image detail loss or "blurring" after denoising. The use of additive Gaussian noise and a real noise model obtained from a Canon camera are compared, which can be seen in the figure (Figure 13). The experimental results show that the neural network trained on the dataset with real noise removes it quite effectively, while preserving fine details and clarity of the images. However, the network trained on Gaussian noise was not able to remove real noise as effectively, which confirms the importance of using realistic noise models when training neural networks for denoising. The separated noise approach can be more versatile and flexible because it is not tied to a specific noise type or scenario. Also, training on separated noise can improve the robustness of the neural network to different

noise levels and lighting conditions in the images. This can make the model more robust and applicable in real-world scenarios.

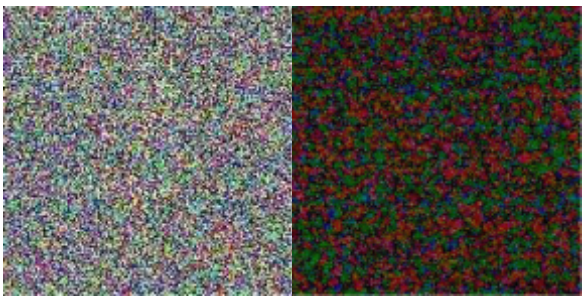

Figure 13 - On the left is the Gaussian noise and on the right is the noise produced by a real camera.

2.4 Intel Open Image Denoise Library

It is worth taking a closer look at an off-the-shelf solution that uses neural networks to remove noise - the Intel® Open Image Denoise library mentioned earlier.

This is an open source library that is part of the Intel Rendering Framework, which also includes the Embree library, and is released under the Apache 2.0 license. Its goal is to provide developers with a high-quality open source solution that significantly reduces rendering time.

The library has a simple and flexible C/C++ API that ensures easy integration into most existing or new rendering solutions.

Support for a wide range of CPUs and GPUs from different manufacturers is currently implemented:

- Intel® 64 architecture compatible processors;

- ARM64 (AArch64);

- NVIDIA graphics processors;

- AMD GPUs;

- Apple silicon.

Because Intel Open Image Denoise uses modern instruction sets such as SSE4, AVX2, AVX-512 and NEON in CPUs, Intel® Xe Matrix Extensions (Intel® XMX) in Intel GPUs, and tensor cores in NVIDIA GPUs, both GPU and CPU performance is optimized to achieve higher denoising performance. Work on the library is currently active and new versions systematically appear on the developers' GitHub [20], adding support for new hardware and optimizing current developments.

At the heart of the library is a deep learning based noise removal filter trained with a wide range of settings, from one sample per pixel to almost full convergence. Thus, it is suitable for both previewing and rendering the final frame.

To train the neural network, the Intel team used pairs of images - one with noise, the other fully converged and without noise. The neural network learned to discolor the images based on examples provided by the developers. [21]

The Intel Open Image Denoise library comes with a fully trained neural network that the team developed for use with ray traced images. For different use cases, noise levels, thematic settings, and lighting, the neural network finds a solution without painstakingly entering additional parameters.

Low-level memory manipulation is used to efficiently utilize the CPU and GPU. For example, if an application wants to support GPUs, passing pointers to memory allocated using a system allocator (e.g., malloc) will result in an error because GPUs in almost all cases cannot access such memory, whereas for CPUs it will be a completely standard operation.

Image data can be transferred to Open Image Denoise either via pointers to memory allocated and managed by the user, or by creating buffer objects. Regardless of which method is used, the data must be allocated so that it is accessible to the device (CPU or GPU). The use of buffers is usually the preferred approach because it ensures that the data allocation requirements are met regardless of the device type.

To ensure compatibility with any device, including GPUs, buffer objects are created in the library to store all transferred image data. The memory allocated using buffers will also be available to the CPU or GPU by default. The data can be copied asynchronously using appropriate methods, which can provide better performance than the conventional memcpy.

If there are several devices of the same type in the system, which is very common (e.g. a computer with a CPU and one or more GPUs), the library has functions to take this into account and flexibly parallelize work between them.

Filters are the main Open Image Denoise objects that are responsible for the actual discoloration. The library comes with a set of filters optimized for different image types and use cases.

Creating filter objects can be very expensive, so developers are strongly encouraged to use the same filter to denoise as many images as possible, as long as these images have the same size, format, and characteristics (i.e., only memory locations and pixel values may differ). Otherwise (e.g., for images with different resolutions), reusing the same filter repeatedly will do no good.

Once the filter is created, it needs to be customized by specifying the input and output images, and possibly setting values for other parameters.

Often filtering performance is more important than maximum possible image quality, so it is possible to switch between several filter quality modes.

Filters can discolor images using only a color ("cosmetic") buffer with noise. Or, to preserve as much detail as possible, it can use auxiliary buffers such as "albedo" and "normals". As an example, consider the off-the-shelf "RT" filter:

The RT (Ray Tracing) filter is a generic denoising filter that is suitable for denoising images rendered using Monte Carlo ray tracing techniques such as unidirectional and bidirectional tracing. It also supports depth of field and motion blur, but is not time-stable. It is based on a convolutional neural network (CNN) and comes with a set of pre-trained models that work well with a wide range of ray-tracing-based rendering systems and noise levels.

For denoising, a low dynamic range (LDR) or high dynamic range (HDR) image is taken as the main input image.

The RT filter has certain limitations with respect to the supported input images. In particular, it cannot discolor images that have not been rendered using ray tracing. Another important limitation is related to anti-aliasing filters. Most renderers use a high-quality pixel reconstruction filter rather than a trivial box filter (e.g., Gaussian, Blackman-Harris) to minimize anti-aliasing artifacts [22]. The RT filter supports such pixel filters, but only if they are implemented with importance sampling. Weighted pixel sampling (sometimes called splatting) introduces correlation between neighboring pixels, which leads to processing failure (noise will not be filtered out), so it is not supported.

Using off-the-shelf filters is not mandatory. To optimize the filter for a specific renderer, number of samples, content type, scene, etc., you can train the model yourself using the included training toolkit and user-provided image datasets.

The Intel Open Image Denoise source distribution includes a Python neural network training toolkit (found in the training directory) that can be used to train denoising filter models on user-provided image datasets. This is an advanced feature of the library, the use of which requires some basic knowledge of machine learning and a basic familiarity with deep learning frameworks and tools (e.g. PyTorch or TensorFlow, TensorBoard).

The training toolkit consists of the following command line scripts:

1. preprocess.py: preprocesses the training and validation datasets.
2. train.py: Trains the model using the preprocessed datasets.
3. infer.py: performs inference on the dataset using the specified training result.
4. export.py: exports the training result to a different format of the temporal model weights.
5. find_lr.py: tool to find the optimal minimum and maximum learning rates.
6. visualize.py: calls TensorBoard to visualize the statistics of the training result.
7. split_exr.py: splits a multi-channel EXR image into multiple feature images.
8. convert_image.py: converts the image to another format.
9. compare_image.py: compares two images with given quality metrics.

1. At the moment it is possible to run the learning toolkit on Linux (other operating systems are not yet supported). Other requirements include Python 3.7, PyTorch 1.8, NumPy 1.19, OpenImageIO 2.1, TensorBoard 2.4(optional) or later versions of all of the above.

2. The dataset should consist of a collection of noisy and their corresponding noise-cleaned reference images. The dataset may contain multiple noisy versions of the same image, e.g., rendered with different samples per pixel and/or using different samples.

3. The training toolkit has requirements for proper file location as well as file naming.

4. Images must be stored in OpenEXR (.exr) format. Each image characteristic (e.g., color, albedo) must be stored in a separate image file, i.e., multi-channel EXR image files are not supported. Multi-channel EXR files can be split into separate images for each characteristic using the included split_exr.py tool.

5. The training and validation datasets can only be used after they have been preprocessed using the preprocess.py script. It converts the specified training and validation datasets into a format that can be more efficiently loaded during training.

6. The same dataset can be preprocessed multiple times using different combinations of features and options.

7. Filters require separate trained models for each supported combination of input features. Thus, depending on which combinations of features the user wants to support for a particular filter, one or more models need to be trained.

8. After pre-processing the datasets, we can start training the model using the train.py script.

9. Model training involves estimation using a validation dataset and regular saving of checkpoints. Visualization of the training statistics can be done using the visualize.py script.

10. The training result produced by the train.py script cannot be immediately used by the main library. It must first be exported to the model weights format, the Tensor Archive (TZA) file.

11. Thus, having studied this library and the process of denoising in general in more detail, we can conclude that this library can help to eliminate graininess in the image when using local estimations, which will lead to faster results by reducing the rendering time.

Practical part

For any numerical method in any domain, an estimate of its accuracy is required. Obtaining such an estimate is possible in three ways by comparison with:

- An already existing generally accepted numerical method with known accuracy;
- An experiment performed with known accuracy;
- An analytical solution.

Historically, one of the first comparison methods for mathematical methods for solving the global illumination equation was an experimental setup called Cornell Box.

To prove the promise of a local estimation method, it requires comparison with other methods. The ideal for comparison of any numerical method is the availability of exact analytical solutions of particular cases. For the global illumination equation there is no analytical solution in general, but for the radiative equation (Figure 3), discussed earlier in Chapter 1.2, there are two analytical solutions. The first of these is the photometric sphere. However, it is poorly suited for accuracy comparisons because of its complete symmetry. The second special case is two infinite parallel planes and a point source between them, aka the Sobolev problem already discussed in Chapter 1.5. In its original form, the solution of the Sobolev problem is not convenient for direct calculations, so let us consider a solution of the Sobolev problem that allows us to obtain the result in a more acceptable analytical form. The final equation used in the calculations was depicted earlier in Figure 8. The matlab program was used for calculations and plotting. In parallel, the calculation was implemented by local estimation using 2000 rays, h1, h2, p1, p2 = 0.5.

The result is shown in the graph (Figure 14).

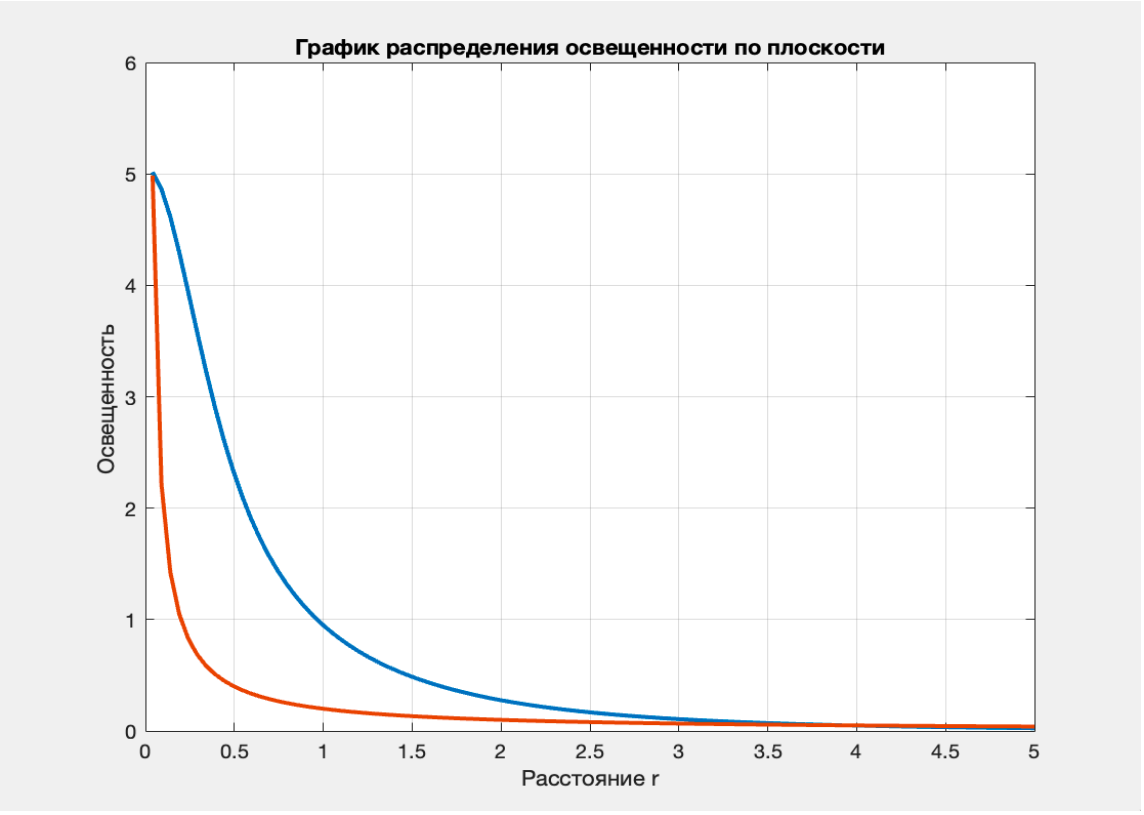

Figure 14 - Comparison of local estimation (orange) and the radiative method (blue).

The time measurement resulted in the following result:

Local estimation algorithm execution time: 0.0034 seconds

Execution time of the Sobolev method algorithm: 0.1917 seconds

The local estimation algorithm is 56.13 times faster than the Sobolev method.

Testing was performed on a Macbook Air M1 (2020) computer.

The only program to date that implements the calculation and modeling of the scene using local estimation is Light-in-Night. This is a domestic software platform for lighting system designers, which uses a module for calculating illuminance levels developed by Russian scientists of the Lighting Engineering Department of the National Research University "MPEI". The program is an important step in the field of import substitution, contributing to the development of the domestic software industry.

The program's main competitor is Dialux, which is known for its extensive functionality and wide range of features. It offers a rich selection of lighting design

and rendering tools, including customization of lighting parameters and scenarios. Dialux also has a long history of industry use and widespread adoption, making it a popular choice among professional designers.

The Light-in-Night program in turn has high performance and stable operation on resource-constrained devices, making it accessible and effective for a wide range of users. The intuitive interface simplifies the process of lighting design without overloading with unnecessary elements. The program offers a wide range of functions, and thanks to the flexibility of settings can be adapted to different user needs. Thanks to active development and the addition of new features, Light In Night is an ambitious project with high potential, and the introduction of an accurate and fast modeling method, which is local estimation, the program has a significant advantage.

It is also worth mentioning the previously announced Intel Embree, which is responsible for optimizing the computation process in this program, using efficient algorithms and data structures such as BVH (Bounding Volume Hierarchy) to ensure high rendering performance, including in complex scenes.

Modeling in Light-in-Night starts with the creation of a 3D model of the architectural object or room where lighting design is planned. The user can import a ready-made model from other CAD programs or create it in Light-in-Night itself, using tools for building walls, floors, ceilings and other elements.

Once the model is created, the user places fixtures and light sources according to the project requirements. Light-in-Night provides a wide range of luminaires of different types and models, allowing for accurate modeling of lighting systems of different characteristics and configurations.

Lighting parameters such as light intensity, beam directionality, color temperature, etc. are then adjusted. Light-in-Night allows the user to perform illuminance calculations, taking into account various factors including light reflections from surfaces, shadow areas and material transparency.

After completing the setup and calculations, the user can visualize the results of the simulation, obtaining a three-dimensional image of the scene taking into

account the lighting. Light-in-Night provides the ability to analyze the quality of lighting, identify shadow areas and evaluate the uniformity of lighting in a room.

Depending on the needs of the project, the user can make adjustments to fixture placement, lighting parameters or fixture type selection to achieve the optimal solution.

Visualization in Light-in-Night is the process of creating a three-dimensional image of an illuminated scene based on illuminance calculations. This process has a number of important functions and applications:

1. Visualization allows architects, designers and engineers to visualize how lighting will look indoors or outdoors after installing certain fixtures and adjusting lighting parameters. This helps to better understand the aesthetic and functional aspects of the project and make necessary adjustments during the design phase.
2. Visualization helps to assess the quality of lighting in different areas of a room or outdoors. Users can analyze lighting uniformity, shadow areas, brightness and contrast to optimize light distribution for optimal indoor or outdoor conditions.
3. Visualization aids in decision making in lighting design. By visually analyzing simulation results, users can compare different luminaire layouts, lighting parameters and luminaire types to select the most efficient and satisfying solution.

4. Visualization is also an important tool for interacting with customers or clients. Providing a visualization of the end result helps customers to better understand the project concept and make informed decisions.

Thus, visualization in Light-in-Night plays a key role in the lighting design process, providing a visual representation of the outcome and helping to make informed decisions to achieve optimal lighting in buildings and streets.

The result of modeling with local estimations can be seen in the figure below (Figure 15).

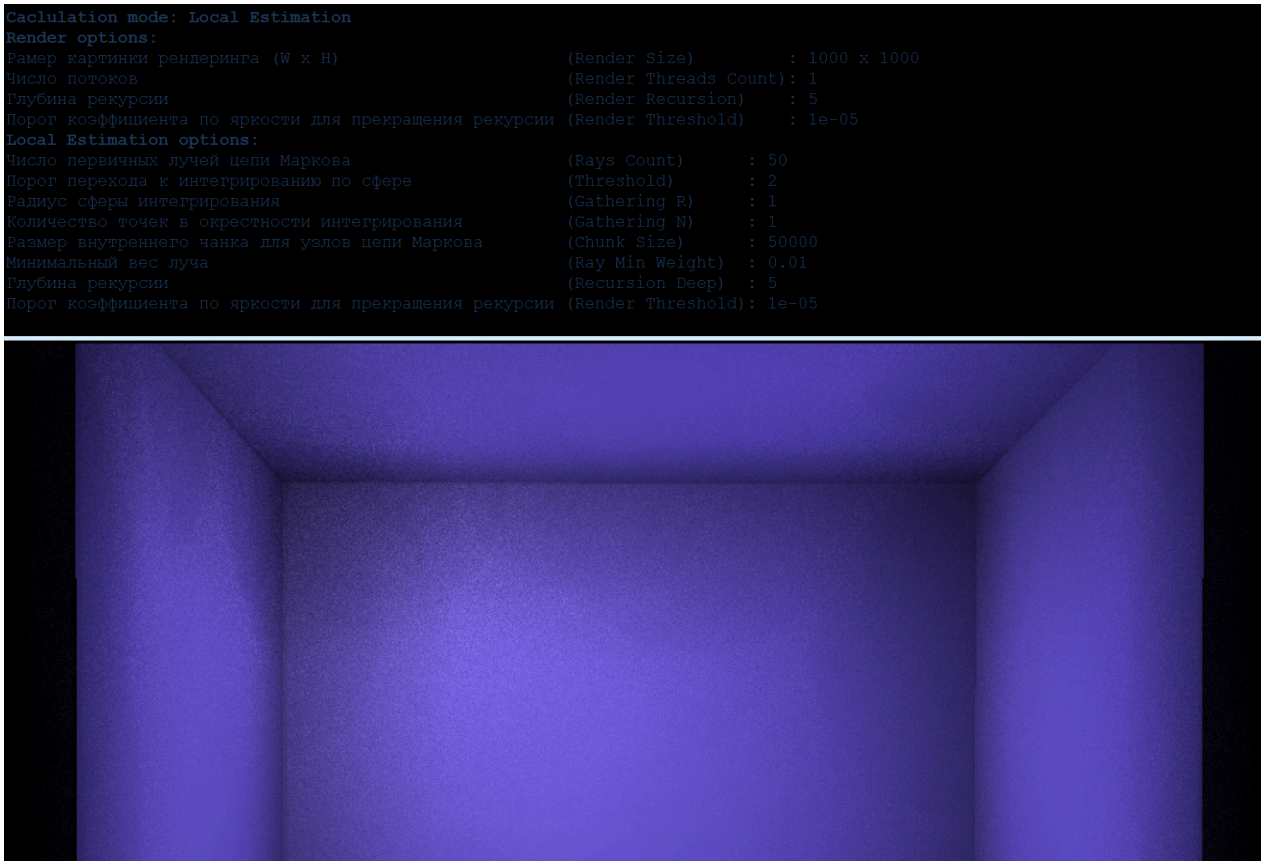

Figure 15 - The result of modeling the scene using local estimations.

As can be seen from the rendering result, closer to the edges there is an increased effect of noise, which is characteristic of the Monte Carlo method. For verification purposes, a program was written to simulate different degrees of noise. The Python programming language was used in the writing. A three-dimensional scene of Cornell Boxes was implemented. The scene consists of several objects located within an enclosed space and includes light sources, objects with different material properties and surfaces, and background elements. To visualize this scene, a ray tracing method based on the Monte Carlo method was applied. This method allows modeling the path of light from the sources to the camera, taking into account its interaction with objects and materials in the scene. By randomly generating light rays and their subsequent reflection, refraction and absorption, an image can be obtained that visually corresponds to real physical lighting processes. The figure

(Figure 16) shows several different time intervals and changes in the percentage of noise.

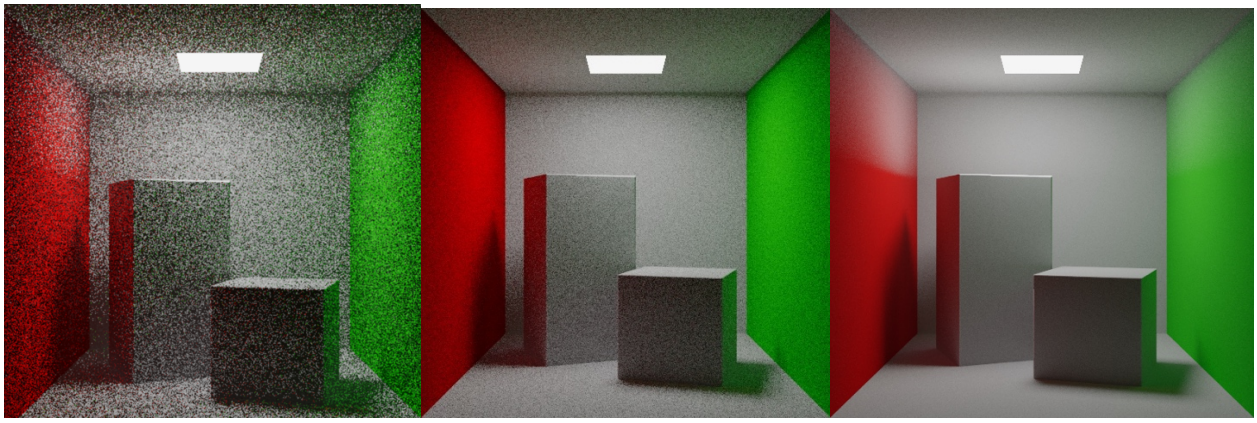

Figure 16 - Manifestation of noise of different intensity.

Further, the Intel Open Image Denoise library was introduced into the current program. After a specified number of iterations, the image was taken, and for clarity, its processed version was displayed in a separate window. The figure (Figure 17) shows the difference between the denoised image and its cleaned copy. Two windows work in parallel.

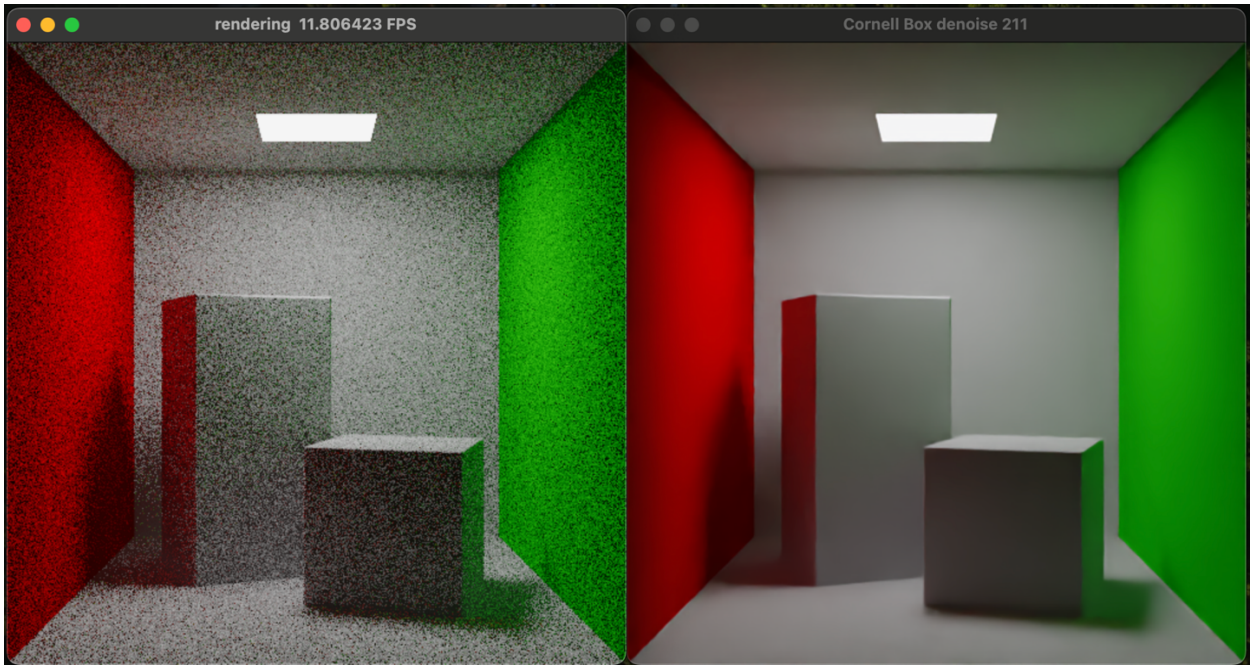

Figure 17 - On the right is an OIDN-cleaned copy of the left image.

Thus, by combining the Monte Carlo ray tracing method and the Intel Open Image Denoise library, it was possible to create a visually appealing and realistic image of a 3D scene while minimizing visual artifacts in the form of noise.

Having the task of measuring the effectiveness of denoising, a series of experiments were conducted, comparing the images before and after denoising at every 1000 rendering iterations. As well as taking the first cleaned image as a sample, and comparing it to subsequent cleaned images to understand how much the cleaning would change over time while reducing the effects of noise.

At first, the idea of a pixel-by-pixel comparison of the images came up. However, due to the randomness of the Monte Carlo method and neural network processing that could slightly change the hue of each pixel each time, this made pixel-by-pixel comparison impractical, as even minor changes not perceived by the human eye could result in large differences in the comparison and visually identical images had a similarity of no more than 40%.

The Hamming distance measurement was then considered because it allows us to estimate the difference between two binary sequences. However, this distance does not account for structural differences between images and is not sensitive enough to small changes that may occur during denoising. From which the results also did not reflect the objective reality.

In the end, the SSIM (Structural Similarity Index) method was chosen as the most appropriate method for evaluating denoising performance. This method takes into account structural and textural changes in images and is also based on psychophysiological features of human perception. SSIM allows us to evaluate the similarity between two images not only at the pixel level, but also at the level of structure and texture, which makes it a more sensitive and realistic method of comparison. This makes it more accurate and informative compared to the pixel-by-pixel comparison and Hamming distance in our task. The screenshot below (Figure 18) shows the results of the comparisons with the percentage of similarity, time and rendering iteration number.

```
Образец для сравнения создан. Время 28.967365026474. Итерация 1000
SSIM между зашумленным и очищенным на каждой итерации: 74.04%. Время 29.017259120941162. Итерация 1000
SSIM между первым очищенным и очищенным на каждой итерации: 100.00%
SSIM между зашумленным и очищенным на каждой итерации: 81.48%. Время 57.64058494567871. Итерация 2000
SSIM между первым очищенным и очищенным на каждой итерации: 99.80%
SSIM между зашумленным и очищенным на каждой итерации: 85.48%. Время 86.0340428352356. Итерация 3000
SSIM между первым очищенным и очищенным на каждой итерации: 99.77%
SSIM между зашумленным и очищенным на каждой итерации: 88.01%. Время 114.50102686882019. Итерация 4000
SSIM между первым очищенным и очищенным на каждой итерации: 99.76%
SSIM между зашумленным и очищенным на каждой итерации: 89.77%. Время 142.82362985610962. Итерация 5000
SSIM между первым очищенным и очищенным на каждой итерации: 99.75%
SSIM между зашумленным и очищенным на каждой итерации: 91.07%. Время 171.19955110549927. Итерация 6000
SSIM между первым очищенным и очищенным на каждой итерации: 99.74%
SSIM между зашумленным и очищенным на каждой итерации: 92.06%. Время 199.48858880996704. Итерация 7000
SSIM между первым очищенным и очищенным на каждой итерации: 99.73%
SSIM между зашумленным и очищенным на каждой итерации: 92.85%. Время 227.8775761127472. Итерация 8000
SSIM между первым очищенным и очищенным на каждой итерации: 99.73%
SSIM между зашумленным и очищенным на каждой итерации: 93.49%. Время 256.2768929004669. Итерация 9000
SSIM между первым очищенным и очищенным на каждой итерации: 99.73%
SSIM между зашумленным и очищенным на каждой итерации: 94.03%. Время 284.6457030773163. Итерация 10000
SSIM между первым очищенным и очищенным на каждой итерации: 99.72%
SSIM между зашумленным и очищенным на каждой итерации: 94.49%. Время 312.956041097641. Итерация 11000
SSIM между первым очищенным и очищенным на каждой итерации: 99.72%
SSIM между зашумленным и очищенным на каждой итерации: 94.88%. Время 341.2959289550781. Итерация 12000
SSIM между первым очищенным и очищенным на каждой итерации: 99.72%
SSIM между зашумленным и очищенным на каждой итерации: 95.22%. Время 369.77904176712036. Итерация 13000
SSIM между первым очищенным и очищенным на каждой итерации: 99.71%
SSIM между зашумленным и очищенным на каждой итерации: 95.51%. Время 398.32773900032043. Итерация 14000
SSIM между первым очищенным и очищенным на каждой итерации: 99.71%
```

Figure 18 - Results of comparisons using the SSIM method.

From the results we can conclude that the cleaned image obtained at 28 seconds after 6 minutes of rendering will differ by only 0.3% from the cleaned image obtained at 398 seconds. Visually, the images also look identical, indicating the effectiveness of denoising with different noise levels and the ability to significantly reduce processing time.

Conclusion

In conclusion, lighting plays a fundamental role in creating realistic visual effects in computer graphics. Lighting modeling is based on solving the Global Illumination Equation (GEE), which describes the complex interactions of light with objects and the environment. One method of modeling this phenomenon is local Monte Carlo estimation, which allows to take into account the complex interactions of light with surfaces in the scene. The principle of their work was analyzed in detail, the unique program Light-in-Night, the only one for today, implementing in practice the processing by this method, as well as its closest competitor in the field of lighting modeling, was considered. In the course of the study of local estimations, a problem confirmed in practice was identified related to the Monte Carlo method, which introduces distortions in the form of noise into the final image. In the course of searching for possible solutions, options in the field of denoising were considered, the most effective of which turned out to be convolutional neural networks. Intel Open Image Denoise library, which uses this technology, has shown itself from the best side and proved to be effective in image cleaning. In the practical part, a program was written that creates a famous scene Cornell Boxes and renders it using Monte Carlo method. The Intel Open Image Denoise library was also used, successfully removing noise in parallel with the ongoing rendering. Finally, the obtained results were objectively measured and the effectiveness of denoising was confirmed in practice.

The described technologies, in addition to the lighting industry, play an important role in the development of state-of-the-art visualization technologies applied to other fields including film, gaming, virtual reality, and scientific visualization. Research in lighting simulation continues, with a focus on improving algorithms, increasing computational efficiency, and using artificial intelligence to optimize the process. This allows for new perspectives in creating even more realistic and impressive visual effects. The results obtained in this work show the prospects of introducing denoising devices into the process of lighting modeling,

with which the desired visualization result is achieved many times faster. This includes the method of local estimations and the only Light-in-Night program that implements it. Denoising is able to show efficiency and present a unique method, and the program that implements it, to make a competitive advantage.

List of references used

1. Kajiya J.T. The Rendering Equation // Proc. Conf. Computer Graphics. 1986. V. 20. No. 4. Pp. 143—150.
2. Wikipedia.org [Офиц. сайт] https://ru.wikipedia.org/wiki/Уравнение_рендеринга (дата обращения 01.03.2024).
3. Будак В.П., Макаров Д.Н. Компьютерная графика с приложением в светодизайн: Учебник для высших учебных заведений. М.: Редакция журнала «Светотехника», 2022. 75 с.
4. Фам Ван Ат, О сходимости методов Якоби и Гаусса– Зайделя, Ж. вычисл. матем. и матем. физ., 1975, том 15, номер 4, 1031–1035 https://www.mathnet.ru/links/fd521a80696c62aa2d9b17a4ae3cafd9/zvmmf6255.pdf
5. Будак В.П., Макаров Д.Н. Компьютерная графика с приложением в светодизайн: Учебник для высших учебных заведений. М.: Редакция журнала «Светотехника», 2022. 83 с.
6. Будак В.П., Макаров Д.Н. Компьютерная графика с приложением в светодизайн: Учебник для высших учебных заведений. М.: Редакция журнала «Светотехника», 2022. 88 с.
7. Wikipedia.org [Офиц. сайт] https://ru.wiki-pedia.org/wiki/Закон_Снеллиуса (дата обращения 12.03.2024).
8. The Monte Carlo Method Nicholas Metropolis and S. Ulam Journal of the American Statistical Association Vol. 44, No. 247 (Sep., 1949), pp 335-341 (7 pages) Published By: Taylor & Francis, Ltd.
9. Будак В.П., Макаров Д.Н. Компьютерная графика с приложением в светодизайн: Учебник для высших учебных заведений. М.: Редакция журнала «Светотехника», 2022. 158 с.
10. Соболев В.В. Точечный источник света между параллельными плоскостями // Доклады Академии Наук СССР. 1944. Т. XLII, No 4. С. 175–176.

11. Будак В.П., Макаров Д.Н. Компьютерная графика с приложением в светодизайн: Учебник для высших учебных заведений. М.: Редакция журнала «Светотехника», 2022. 161 с.

12. Ермаков С.М., Михайлов Г.А. Курс статистического моделирования. М.: Главная редакция физико-математической литературы изд-ва «Наука», 1976. 320 с.

13. Будак В.П., Желтов В.С., Калатуцкий Т.К. Локальные оценки метода Монте-Карло в решении уравнения глобального освещения с учетом спектрального представления объектов // Компьютерные исследования и моделирование. 2012. Т. 4, No 1. С. 79.

14. В. П. Будак, А. В. Гримайло, В. С. Желтов, С. А. Якушев ГрафиКон 2023: 33-я Международная конференция по компьютерной графике и машинному зрению, 19-21 сентября 2023 г., стр 112

15. Github.com [Офиц. репозиторий] https://github.com/embree/embree (дата обращения 10.03.2024).

16. Nikolas Sanden Fileadmin.cs [Офиц. Сайт] https://fileadmin.cs.lth.se/cs/Education/EDAN35/projects/2022/Sanden-BVH.pdf (дата обращения 13.03.2024).

17. Alain Galvan alan.xyz [Офиц. Сайт] https://alain.xyz/blog/ray-tracing-denoising (дата обращения 17.03.2024).

18. Gaurav Rajpal Medium.com [Офиц. Сайт] https://medium.com/swlh/a-comprehensive-guide-to-convolution-neural-network-86f931e55679 (дата обращения 10.03.2024).

19. Н. А. Сизов, В. П. Раевский, Д. П. Дурандин [и др.]. — Текст: непосредственный // Молодой ученый. — 2019. — № 27 (265). — С. 34-36. — URL: https://moluch.ru/archive/265/61326/ (дата обращения: 13.03.2024).

20. Github [Офиц. Репозиторий] https://github.com/OpenImageDenoise/oidn-/tree/master (дата обращения 22.03.2024).

21. Intel [Офиц. Сайт] https://www.intel.com/content/dam/develop/external/us/en/documents/intelopenimagedenoiselibrarysavestimeboostsquality.pdf (дата обращения 16.03.2024).
22. Демяненко Я. М. edu.mmcs.sfedu.ru [Офиц. Сайт] https://edu.mmcs-.sfedu.ru/pluginfile.php/49502/mod_resource/content/4/Лекция%205%20Пространственная%20обработка%20изображений.pdf (дата обращения 13.03.2024).

Appendix 1

Software code to demonstrate the operation of denoising at rendering time in Monte Carlo simulation of light in Python:

```
import taichi as tai
from taichi.math import *
from PIL import Image
import pygame
import time
import oidn
import numpy as np
from skimage.metrics import structural_similarity as
compare_ssim
import cv2

tai.init(arch=tai.gpu, default_ip=tai.i32, default_fp=tai.f32)

pix_resolution = (512, 512)
pix_buffer = tai.Vector.field(4, float, pix_resolution)
pixels = tai.Vector.field(3, float, pix_resolution)

ray = tai.types.struct(origin=vec3, direction=vec3, color=vec3)
material = tai.types.struct(albedo=vec3, emission=vec3)
moving = tai.types.struct(position=vec3, rotation=vec3,
scale=vec3, matrix=mat3)
obj = tai.types.struct(distance=float, transform=moving,
material=material)

walls = obj.field(shape=8)
walls[0] = obj(
    transform=moving(vec3(0, 0.809, 0), vec3(90, 0, 0), vec3(0.2,
0.2, 0.01)),
    material=material(vec3(1, 1, 1) * 1, vec3(100)),
)
walls[1] = obj(
    transform=moving(vec3(0, 0, -1), vec3(0, 0, 0), vec3(1, 1,
0.2)),
    material=material(vec3(1, 1, 1) * 0.4, vec3(1)),
)
walls[2] = obj(
    transform=moving(vec3(0.275, -0.55, 0.2), vec3(0, -197, 0),
vec3(0.25, 0.25, 0.25)),
    material=material(vec3(1, 1, 1) * 0.4, vec3(1)),
)
walls[3] = obj(
    transform=moving(vec3(-0.275, -0.3, -0.2), vec3(0, 112, 0),
vec3(0.25, 0.5, 0.25)),
    material=material(vec3(1, 1, 1) * 0.4, vec3(1)),
)
walls[4] = obj(
```

```
    transform=moving(vec3(1, 0, 0), vec3(0, 90, 0), vec3(1, 1,
0.2)),
    material=material(vec3(0, 1, 0) * 0.5, vec3(1)),
)
walls[5] = obj(
    transform=moving(vec3(-1, 0, 0), vec3(0, 90, 0), vec3(1, 1,
0.2)),
    material=material(vec3(1, 0, 0) * 0.5, vec3(1)),
)
walls[6] = obj(
    transform=moving(vec3(0, 1, 0), vec3(90, 0, 0), vec3(1, 1,
0.2)),
    material=material(vec3(1, 1, 1) * 0.4, vec3(1)),
)
walls[7] = obj(
    transform=moving(vec3(0, -1, 0), vec3(90, 0, 0), vec3(1, 1,
0.2)),
    material=material(vec3(1, 1, 1) * 0.4, vec3(1)),
)



@tai.func
def turn(a: vec3) -> mat3:
    sin_a, cos_a = sin(a), cos(a)
    return (
        mat3(cos_a.z, sin_a.z, 0, -sin_a.z, cos_a.z, 0, 0, 0, 1)
        @ mat3(1, 0, 0, 0, cos_a.x, sin_a.x, 0, -sin_a.x,
cos_a.x)
        @ mat3(cos_a.y, 0, -sin_a.y, 0, 1, 0, sin_a.y, 0,
cos_a.y)

    )


@tai.func
def dist(obj: obj, pos: vec3) -> float:
    position = obj.transform.matrix @ (pos -
obj.transform.position)
    q = abs(position) - obj.transform.scale
    return length(max(q, 0)) + min(max(q.x, max(q.y, q.z)), 0)


@tai.func
def near_obj(p: vec3):
    index, min_dis = 0, 1e32
    for i in tai.static(range(8)):
        dis = dist(walls[i], p)
        if dis < min_dis:
            min_dis, index = dis, i
    return index, min_dis
```

```
@tai.func
def normals(obj: obj, p: vec3) -> vec3:
    e = vec2(1, -1) * 0.5773 * 0.005
    return normalize(
        e.xyy * dist(obj, p + e.xyy)
        + e.yyx * dist(obj, p + e.yyx)
        + e.yxy * dist(obj, p + e.yxy)
        + e.xxx * dist(obj, p + e.xxx)
    )


@tai.func
def ray_cast(ray: ray):
    w, s, d, cerr = 1.6, 0.0, 0.0, 1e32
    index, t, position, hit = 0, 0.005, vec3(0), False
    for _ in range(64):
        position = ray.origin + ray.direction * t
        index, distance = near_obj(position)

        ld, d = d, distance
        if ld + d < s:
            s -= w * s
            t += s
            w *= 0.5
            w += 0.5
            continue
        err = d / t
        if err < cerr:
            cerr = err

        s = w * d
        t += s
        hit = err < 0.001
        if t > 5.0 or hit:
            break
    return walls[index], position, hit


@tai.func
def sampling(normal: vec3) -> vec3:
    z = 2.0 * tai.random() - 1.0
    a = tai.random() * 2.0 * pi
    xy = sqrt(1.0 - z * z) * vec2(sin(a), cos(a))
    return normalize(normal + vec3(xy, z))


# local estimate

# оценка освещённости в заданной точке с учетом освещения от
источника света, модель Блинна-Фонга
@tai.func
def estimate_radiance(source_position: vec3, direction: vec3,
normal: vec3) -> vec3:
```

```
    light_position = vec3(0.0, 1.0, 0.0)
    light_color = vec3(1.0, 1.0, 1.0)
    light_intensity = 1.0

    light_direction = normalize(light_position -
source_position)
    diffuse_term = max(dot(light_direction, normal), 0.0) *
light_intensity * light_color

    view_direction = normalize(-direction)
    half_vector = normalize(light_direction + view_direction)
    specular_term = pow(max(dot(half_vector, normal), 0.0),
32.0) * light_intensity * light_color

    return diffuse_term + specular_term


@tai.func
def double_local_estimate(source_position: vec3,
target_position: vec3, normal: vec3, num_samples: int,
threshold: float) -> vec3:
    total_radiance = vec3(0.0)
    total_weight = 0.0

    # for _ in range(num_samples):
        # Генерация случайного направления луча
        # direction = hemispheric_sampling(normal) # вместо
direction - target_position

    # Вычисление излучения от источника в этом направлении
    radiance = estimate_radiance(source_position,
target_position, normal)

    # Вычисление веса на основе косинусного фактора
    weight = dot(target_position, normal)

    # Накопление излучения с весом
    total_radiance += radiance * weight
    total_weight += weight

    # Проверка, падает ли вес ниже порога
    # if total_weight < threshold:
    #     break

    # Нормализация излучения по общему весу
    normalized_radiance = vec3(0.0)
    if total_weight > 0:
        normalized_radiance = total_radiance / total_weight

    return normalized_radiance


@tai.func
```

```
def raytrace(ray: ray) -> ray:
    for _ in range(3):
        object, position, hit = ray_cast(ray)
        if not hit:
            ray.color = vec3(0)
            break

        normal = normals(object, position)
        ray.direction = sampling(normal)
        ray.color *= object.material.albedo
        ray.origin = position

        # вызов локальной оценки
        ray.color += double_local_estimate(position, ray.origin,
normal, num_samples=1000, threshold=0.01)
        ray.direction = sampling(normal)
        ray.color *= object.material.albedo
        ray.origin = position

        intensity = dot(ray.color, vec3(0.3, 0.6, 0.1))
        ray.color *= object.material.emission
        visible = dot(ray.color, vec3(0.3, 0.6, 0.1))
        if intensity < visible or visible < 0.00001:
            break
    return ray


@tai.kernel
def level_create():
    for i in walls:
        r = radians(walls[i].transform.rotation)
        walls[i].transform.matrix = turn(r)


@tai.kernel
def render(camera_position: vec3, camera_lookat: vec3,
camera_up: vec3):
    for i, j in pixels:
        z = normalize(camera_position - camera_lookat)
        x = normalize(cross(camera_up, z))
        y = cross(z, x)

        half_width = half_height = tan(radians(35) * 0.5)
        lower_left_corner = camera_position - half_width * x -
half_height * y - z
        horizontal = 2.0 * half_width * x
        vertical = 2.0 * half_height * y

        uv = (vec2(i, j) + vec2(tai.random(), tai.random())) /
vec2(pix_resolution)
        po = lower_left_corner + uv.x * horizontal + uv.y *
vertical
        rd = normalize(po - camera_position)
```

```
            ray = raytrace(ray(camera_position, rd, vec3(1)))
            buffer = pix_buffer[i, j]
            buffer += vec4(ray.color, 1.0)
            pix_buffer[i, j] = buffer

            color = buffer.rgb / buffer.a
            color = pow(color, vec3(1.0 / 2.2))
            color = (
                mat3(0.6, 0.4, 0.05, 0.07, 1, 0.01, 0.03, 0.1, 0.8)
                @ color
            )
            color = (color * (color + 0.02) - 0.0001) / (color * (1
* color + 0.4) + 0.2)
            color = (
                mat3(1.6,  -0.5, -0.07, -0.1,  1.1, -0.006, -0.003,
-0.07, 1)
                @ color
            )
            pixels[i, j] = clamp(color, 0, 1)


def save_image(image, filename):
    Image.fromarray(image).save(filename)


def denoise_image(img_noised):
    result = np.zeros_like(img_noised, dtype=np.float32)

    device = oidn.NewDevice()
    oidn.CommitDevice(device)

    denoise_filter = oidn.NewFilter(device, "RT")
    oidn.SetSharedFilterImage(
        denoise_filter, "color", img_noised, oidn.FORMAT_FLOAT3,
img_noised.shape[1], img_noised.shape[0]
    )
    oidn.SetSharedFilterImage(
        denoise_filter, "output", result, oidn.FORMAT_FLOAT3,
img_noised.shape[1], img_noised.shape[0]
    )
    oidn.CommitFilter(denoise_filter)
    oidn.ExecuteFilter(denoise_filter)
    result = np.array(np.clip(result * 255, 0, 255),
dtype=np.uint8)

    oidn.ReleaseFilter(denoise_filter)
    oidn.ReleaseDevice(device)

    return result


# Попиксельное сравнение
```

```
def compare_images(image1, image2):
    if image1.shape != image2.shape:
        raise ValueError("Need same dimensions.")
    num_similar_pixels = np.sum(image1 == image2)
    total_pixels = image1.shape[0] * image1.shape[1] *
image1.shape[2]

    similarity_percentage = (num_similar_pixels / total_pixels)
* 100

    return similarity_percentage


# Сравнение ssim
def calculate_ssim(image1, image2):
    img1 = cv2.imread(image1)
    img2 = cv2.imread(image2)
    gray_img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    gray_img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
    ssim_score, _ = compare_ssim(gray_img1, gray_img2,
full=True)
    ssim_percentage = (ssim_score + 1) / 2 * 100
    return ssim_percentage


# Сравнение расстоянием Хеллингера
def hellinger_distance(hist1, hist2):
    sqrt_diff = np.sqrt(hist1) - np.sqrt(hist2)
    distance = np.sqrt(np.sum(sqrt_diff ** 2)) / np.sqrt(2)
    return distance


def main():
    start_time = time.time()

    rendering_iterations = 1
    window = tai.ui.Window("rendering", pix_resolution)
    canvas = window.get_canvas()
    level_create()
    pygame.init()
    flag = 0
    flag2 = 0

    while window.running:
        render(vec3(0, 0, 3.5), vec3(0, 0, -1), vec3(0, 1, 0))
        canvas.set_image(pixels)
        window.show()
        pygame.display.set_caption(f"Cornell Box denoise
{rendering_iterations}")

        if rendering_iterations % 100 == 0:
            filename = f'CornellBoxNoised.png'
            tai.tools.imwrite(pixels.to_numpy(), filename)
```

```
            img_noised = np.array(Image.open(filename),
dtype=np.float32) / 255.0
            result = denoise_image(img_noised)
            save_image(result, "CornellBoxDenoised.png")
            img = Image.open("CornellBoxDenoised.png")

            screen = pygame.display.set_mode(pix_resolution)
            pygame_img = pygame.image.fromstring(img.tobytes(),
img.size, img.mode)
            screen.blit(pygame_img, (0, 0))
            pygame.display.update()

            # flag += 1 # Если нужно протестировать сравнением
то rendering_iterations выставить кол-во итераций для создания
образца
            if (flag == 1):
                save_image(result, "CornellBoxSample.png")
                flag2 = 1
                elapsed_time = time.time() - start_time
                print(f"Образец для сравнения создан. Время
{elapsed_time}. Итерация {rendering_iterations}")


            if rendering_iterations % 1 == 0 and flag2 == 1:
                image_path1 =
"/Users/stashev/PycharmProjects/misisHomeworks/taichiRend/Cornel
lBoxDenoised.png"
                image_path2 =
"/Users/stashev/PycharmProjects/misisHomeworks/taichiRend/Cornel
lBoxNoised.png"
                image_path3 =
"/Users/stashev/PycharmProjects/misisHomeworks/taichiRend/Cornel
lBoxSample.png"

                ssim = calculate_ssim(image_path1, image_path2)
                ssim2 = calculate_ssim(image_path1, image_path3)
                ssim3 = calculate_ssim(image_path2, image_path3)
                elapsed_time = time.time() - start_time
                print(f"SSIM между зашумленным и очищенным на
каждой итерации: {ssim:.2f}%. Время {elapsed_time}. Итерация
{rendering_iterations}")
                print(f"SSIM между первым очищенным и очищенным
на каждой итерации: {ssim2:.2f}%")
                print(f"SSIM между зашумленным и очищенным на
1000 итерации: {ssim3:.2f}%. Время {elapsed_time}. Итерация
{rendering_iterations}")


                # original_image =
Image.open("CornellBoxSample.png")
                # denoised_image =
Image.open("CornellBoxNoised.png")
                # original_pixels = np.array(original_image)
```

```
                # denoised_pixels = np.array(denoised_image)
                # original_hist = np.histogram(original_pixels,
bins=256, range=(0, 1))[0]
                # denoised_hist = np.histogram(denoised_pixels,
bins=256, range=(0, 1))[0]
                # original_hist = original_hist /
np.sum(original_hist)
                # denoised_hist = denoised_hist /
np.sum(denoised_hist)
                # similarity = hellinger_distance(original_hist,
denoised_hist)
                # elapsed_time = time.time() - start_time
                # print(f"Hellinger Distance: {similarity},
Elapsed Time: {elapsed_time} seconds, {rendering_iterations}
iteration")
                # original_image =
Image.open("CornellBoxSample.png")
                # denoised_image =
Image.open("CornellBoxNoised.png")
                # original_pixels = np.array(original_image)
                # denoised_pixels = np.array(denoised_image)
                # # similarity = ssim(original_pixels,
denoised_pixels, multichannel=True)
                # # print(f"Similarity: {similarity}, Elapsed
Time: {elapsed_time} seconds")
                # similarity_percentage =
compare_images(original_pixels, denoised_pixels)
                # elapsed_time = time.time() - start_time
                # print(f"Similarity Percentage:
{similarity_percentage}%, Elapsed Time: {elapsed_time} seconds,
{rendering_iterations} iteration")

        rendering_iterations += 1


if __name__ == "__main__":
    main()
```

Appendix 2

Realization of the solution of the Sobolev problem by means of Local Estimates of the Monte Carlo method in Matlab:

```
Phi = 1; % Световой поток источника
N = 2000; % Количество лучей
T = 5; % Количество соударений
r = linspace(0.04, 5, 100); % Диапазон значений r
rho = ones(1, N + 1) * 0.5; % Коэффициенты отражения для всех
лучей 0.5

% Засекаем время для локальной оценки
tic;
illumination_local = zeros(size(r));
for i = 1:length(r)
    illumination_local(i) = localEstimate(Phi, N, T, rho, r(i));
end
local_time = toc;

h1 = 0.5;
h2 = 0.5;
rho1 = 0.5;
rho2 = 0.5;

% Засекаем время для метода Соболева
tic;
illuminationSobolev = zeros(size(r));
for i = 1:length(r)
    result = h1 / ((h1^2 + r(i)^2)^1.5);
    for k = 0.01:0.01:10
        term1 = exp(-h1 * k) * rho1 * k * besselk(1,k)+exp(-h2 *
k);
        term2 = 1 - rho1 * rho2 * k^2 * besselk(1,k)^2;
        term3 = besselk(1,k) * besselj(0, k * r(i)) * k^2;
        result = result + term1 / term2 * term3 * 0.01; % ? dk
    end
    illuminationSobolev(i) = result;
end
sobolev_time = toc;

% Выводим результаты
fprintf('Время выполнения алгоритма локальной оценки: %.4f
секунд\n', local_time);
fprintf('Время выполнения алгоритма метода Соболева: %.4f
секунд\n', sobolev_time);
fprintf('Алгоритм локальной оценки выполняется в %.2f раз
быстрее, чем метод Соболева.\n', sobolev_time/local_time);


figure;
plot(r, illuminationSobolev, 'LineWidth', 2);
```

```
hold on;
plot(r, illumination_local, 'LineWidth', 2);
hold off;

xlabel('Расстояние r');
ylabel('Освещенность ');
title('График распределения освещенности по плоскости');
grid on;


function result = localEstimate(Phi, N, T, rho, r)
    result = 0;
    pi_val = pi;

    % Вычисление веса каждого луча Q
    Q = ones(1, N + 1); % Изначально у всех 1
    for i = 2:N+1
        Q(i) = Q(i - 1) * rho(i-1); % Умножаем на коэффицент
отражения (0.5), т.е. уменьшаем каждый следующий в 2 раза
    end

    % Вычисление освещенности для каждого луча и каждого
соударения
    for n = 1:N
        for i = 1:T+1
            F = 1 / (pi_val * r); % ? Функция F обратно
пропорциональная расстоянию
            result = result + Phi / (pi_val * N) * Q(i) * F;
        end
    end
end
```