Ray tracer: iterative approach

Kateryna Kovalchuk*, Radomyr Husiev*, Bohdan Opyr*, Roman Zaletskyi*, Ostap Trush*
* *Faculty of Applied Sciences of Ukrainian Catholic University, L'viv, Ukraine*

***Abstract*—This report describes the implementation of a ray tracer, a rendering algorithm that simulates light propagation to create realistic 3D images. The goal is to implement it iteratively, from basic to progressively more advanced techniques, and explore the trade-off between visual fidelity and performance.**

I. Introduction

3D scene visualization is crucial across various fields, from Computer Generated Imagery (CGI) in movies to scientific problems. This process, known as rendering, often employs ray tracing.

In this project, we developed a simple ray tracer and iteratively improved it, enhancing visual fidelity and performance. The goal is to explore popular ray tracing methods and evaluate resource requirements.

The initial implementation includes fundamental ray-object intersection calculations and the Blinn-Phong lighting model [4]. Subsequent iterations introduce shadows, reflections, model and texture loading, optimizations using bounding volume hierarchies [7], and global illumination [12].

II. Background

Rasterization and ray tracing are two prevalent rendering methods. Rasterization converts each primitive (e.g., triangle) into its pixel representation, considering occlusions based on the depth (the distance to the camera). The information about which pixels correspond to what primitive is preserved. Then, a GPU-based program, a fragment shader [16], runs for each pixel of the corresponding primitive to calculate its color. Rasterization is very fast, and modern GPUs are highly optimized for it [23]. This is why it is the approach most commonly used in real-time applications, such as game development.

On the contrary, ray tracing generates at least one ray per pixel. These rays originate from the camera and are directed based on the pixel coordinates. They intersect with the scene to determine the pixel light values. Additional rays simulate complex light interactions. An example of a ray tracer is Blender's Cycles [3], a standard render engine for Computer-generated imagery (CGI).

Ray tracing is highly parallelizable, with pixel value computations largely independent. This is particularly evident in basic ray tracing techniques. Thus, ray tracers often utilize GPUs, though programming for GPUs can be complex and highly depends on the GPU chosen. Vendor-specific pipelines, like the CUDA toolchain [24][1], maximize performance for specific GPUs. Alternatively, graphics APIs offer a unified interface for CPU and GPU, streamlining parallelization.

III. The Rendering Pipeline

We chose the Open Graphics Library [18] (OpenGL) as our graphics API. There are two main ways to achieve massive parallelism with OpenGL: using compute shaders and leveraging the rendering pipeline. Compute shaders perform arbitrary parallel computations on the GPU but are far from easy to use. The rendering pipeline, designed for rasterization, transforms a 3D scene into an image through a fixed sequence of steps. It can be repurposed for ray tracing, which is our chosen approach.

One of the stages of the rendering pipeline is the fragment shader [16] – a program that runs independently for each visible pixel of a specific model. Its main purpose is to apply textures to models. Textures can be generated in this stage, too. When ray tracing, one generates each pixel of the render independently in parallel, as if it is a texture, applying this texture to a simple model, such as a triangle, and covering the entire viewport[2] with it. You can see a visualization of this in Fig. 1.

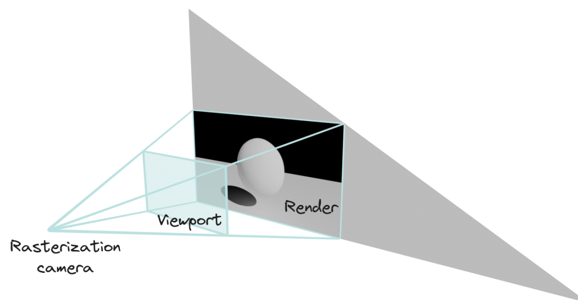

Fig. 1. Ray tracer in a fragment shader, visualized.

IV. Implementing Simple Scenes

A. Rendering a Sphere

The simplest shading model – that is, a model of how light interacts with a scene – is flat shading, also known as no

[1]CUDA toolchain – a set of software development tools and libraries provided by NVIDIA to program and optimize applications for NVIDIA GPUs.

[2]Viewport – the virtual representation of the camera sensor mapped onto the screen.

shading, showing a uniform color or texture without additional calculations. An example of a very simple ray tracer is one that renders a black sphere with flat shading. It displays black where rays from the camera intersect a predefined sphere and white elsewhere, rendering a black circle when viewed head-on (see Fig. 2). An example of flat shading applied to a textured sphere is shown in Fig. 3.

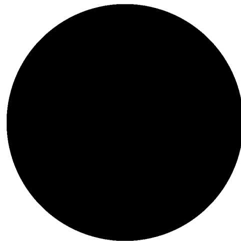

Fig. 2. A black sphere with flat shading.

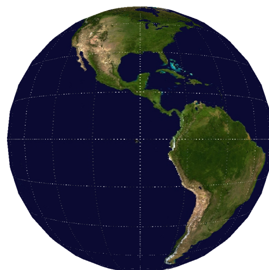

Fig. 3. A textured sphere with flat shading.

Flat shading remains relevant in game development. It is often used with baked lighting [27], a technique that suggests encoding the lighting information in the texture data. Lambertian surfaces [12], which exhibit uniform brightness regardless of viewing direction (a full explanation of these is given in section IX), can be fully represented in static scenes using this method.

B. Applying Blinn-Phong Shading

Most surfaces look different depending on the viewing angle. Therefore, more complex shading models like Blinn-Phong [4] are often preferred. Blinn-Phong estimates the light intensity using the ray direction, light source direction, and surface normal.

Direct illumination refers to light reflected from one surface. The Blinn-Phong model approximates direct lighting from a point source. Diffuse and specular components, along with roughness, are considered.

The diffuse component uses the Lambertian model, factoring in the light angle and surface albedo. Intuitively, the more shallow the incoming light angle, the more the light spreads, visually darkening the surface.

The specular (mirror-like) component is modeled for each visible surface point. The Blinn-Phong model approximates the amount of reflected light by using the directions toward the light source and the camera, avoiding the need to compute the reflected ray.

Blinn-Phong is not a Physically Based Rendering (PBR) model, as it does not accurately reflect real-world light behavior [21]. Figure 4 shows our Blinn-Phong implementation.

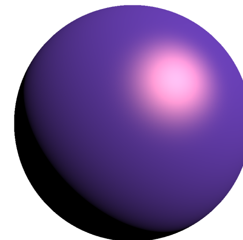

Fig. 4. A sphere with Blinn-Phong shading.

The bottom half of the sphere in Fig. 4 receives no light and appears completely dark. Ambient lighting can be added to simulate light from other directions. This is done by multiplying the flat shaded value by a small constant and adding to the previously described result, as illustrated in Fig. 6.

C. Handling Multiple Surfaces

Adding multiple objects involves only considering the intersection closest to the camera. However, computing direct illumination with multiple surfaces requires considering shadows. Shadow rays [30] originate at the initial intersection and are directed towards the light source. They determine whether a point on the surface is in shadow. This is the case if a shadow ray intersects another object on its way to the light source (see Fig. 5). A render using our implementation is shown in Fig. 6.

Adding shadow rays did not significantly impact performance. This iteration ran 3% slower than the flat-shaded variant. This is because most of the time was spent on tasks other than running the fragment shader, such as executing other stages of the rendering pipeline. Detailed measurements for this and all other improvement iterations are in the results section, mainly in Table I.

D. Creating Glossy Reflections

Multiple objects enable glossy reflections. Blinn-Phong assumes that only light coming directly from a source is reflected. Perfect glossy reflections model any mirror-like reflections. This is the most straightforward reflection to

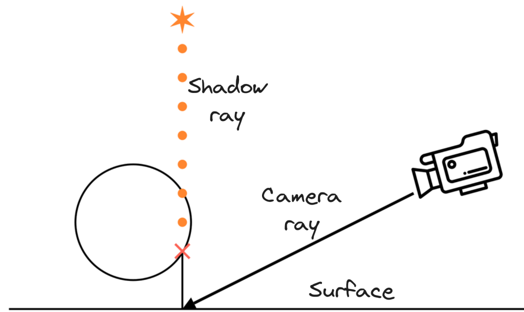

Fig. 5. Using shadow ray to find whether the surface is accessible from the light source.

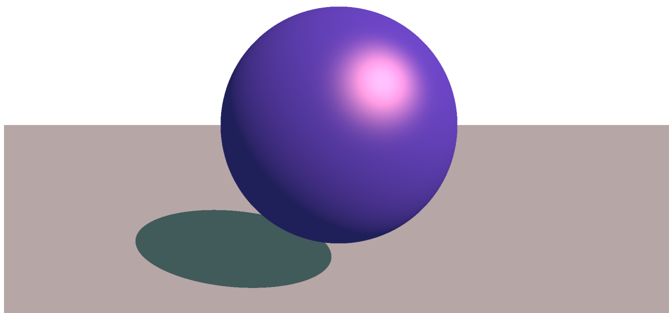

Fig. 6. A sphere and a ground plane with Blinn-Phong shading and shadow rays.

model, one where the incoming and outgoing light angles are symmetrical. Computation involves firing reflection rays from the intersection point, with their direction based on angle symmetry. The reflection ray's value is calculated similarly to a camera ray's. If necessary, new reflection rays are created recursively. The reflection ray's value is multiplied by a glossiness factor, determined by the surface material. This result is added to the direct illumination. Our render is shown in Fig. 7.

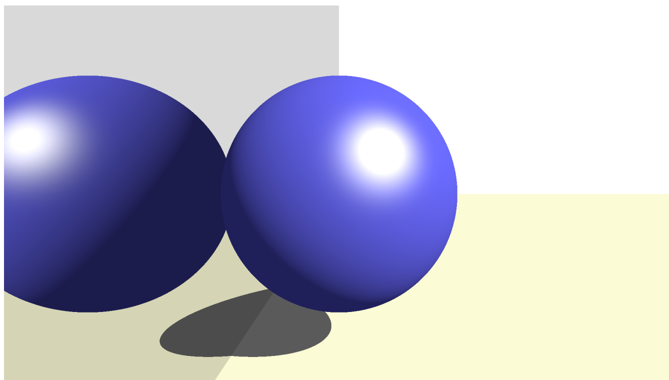

Fig. 7. A sphere, a ground plane, and a reflection plane.

V. Loading Arbitrary Scenes

Loading arbitrary scenes is crucial. Static scenes are commonly represented by triangles approximating surfaces, as triangles are simple yet powerful enough for most use cases. The Graphics Library Transmission Format [17] (GLTF) is an open and popular choice for storing this representation.

GLTF supports complex features like animations, but only materials, textures, and triangles are needed for ray tracing. Triangles are passed to the fragment shader using Shader Storage Buffer Objects [19][3], while other context data, such as the screen resolution, is passed through uniforms [20][4].

We check for intersections with all triangles in the scene. The Möller–Trumbore [22] formula checks for ray-triangle intersections and provides UV coordinates [6][5], allowing the display of arbitrary textured models.

VI. Enhancing Performance with BVH

Iterating through all triangles for every pixel becomes impractical for scenes with millions or billions of triangles. Our solution took 4.1x the time for a 4x more complex scene, demonstrating near-linear performance. To solve this, intersection search algorithms with sublinear complexity are used.

A common approach involves using a Bounding Volume Hierarchy [7] (BVH). This means creating a tree structure where the primitives, such as triangles, are the leaves. The bounding volume of each node encompasses its children, from volumes encapsulating individual primitives to the root node that contains the entire scene (see Fig. 8). Checking for ray intersections begins at the root; if intersected, checks proceed recursively through child nodes. The entire branch is pruned if no intersection exists, allowing sublinear complexity.

Axis-Aligned Bounding Boxes [15] (AABB) are often used to define bounding volumes. In this approach, a bounding volume is a box where the edges align with the coordinate axes, simplifying intersection calculations.

BVH with AABBs significantly enhances performance in large scenes. Rendering a more complex scene, 62,975 instead of 15,743 triangles, a 4x increase, only increased the time taken to render a single frame 1.38 times.

Further performance enhancements involve two main steps. First, discard bounding boxes that intersect the ray beyond a previously found intersection. Second, traverse bounding boxes in a front-to-back order, prioritizing closer boxes. This usually allows for more effective pruning of branches. The slab method [15] is the most common method for computing the AABB-ray intersection. It already produces the distance to the intersection, making implementing these optimizations trivial. They improved the performance 2.1 times over the 62,975-face scene rendered with naive BVH and 526 times over an iterative

[3]SSBOs – a way to transmit data, especially variable-length vectors, to a shader.

[4]Uniform – a type of global shader variable, commonly used to pass parameters.

[5]UV coordinates: 2D coordinates relative to a texture or vertices, used to map textures onto 3D surfaces.

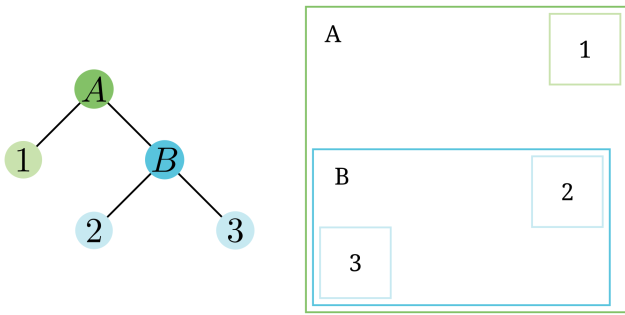

Fig. 8. BVH/AABB visualization.

approach. A comparison of these optimizations is shown in Fig. 9.

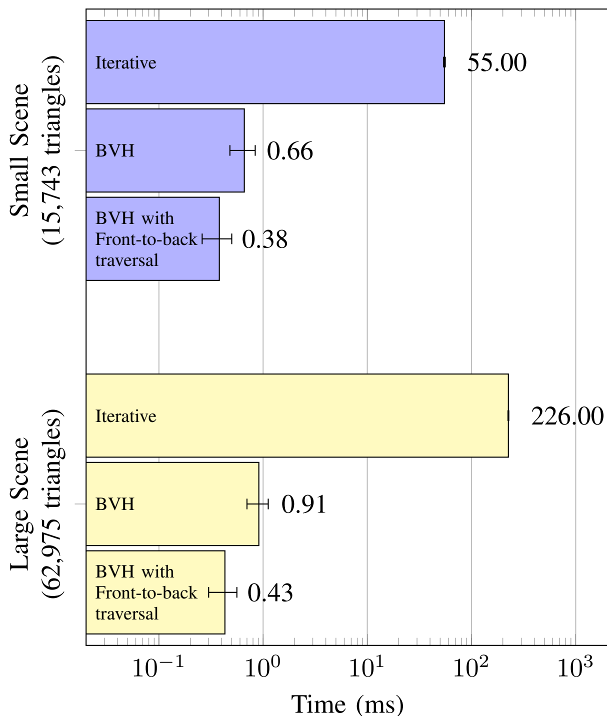

Fig. 9. Ray tracer performance comparison (log scale).

VII. Implementing Refraction

With enhanced scene-loading capabilities, we could further improve visual fidelity. A ray tracer, as described so far, can render direct illumination with shadows and perfectly glossy reflections. However, there is significantly more to simulate. Another phenomenon in which ray tracers excel is refraction – the behavior of transmissive surfaces, such as glass.

In reality, this behavior is quite complex. The incoming light is partially reflected and refracted. The ratio of reflection to refraction depends on the properties of the media and the angle of the ray relative to the surface. The shallower the ray angle, the more light is reflected. The Fresnel equations [11] describe these interactions. However, these equations are very computationally intensive. Schlick's approximation [26] offers a faster alternative that is suitable for vacuum-medium interactions.

Fresnel can be used for more than refraction. In computer graphics, non-transmissive materials are typically classified as metals or dielectrics [8]. Metals reflect light entirely specularly, while dielectrics reflect light specularly and diffusely. The Fresnel equations determine the proportion of specular reflection.

VIII. Normal Mapping

Normal maps [5] allow for the detailing of surfaces without creating more geometry. These are typically textures used to adjust surface normals for lighting calculations. They are widely adopted as they require much less computational resources than the geometry they simulate. Normal maps can be generated procedurally, not solely specified as textures. Fig. 10 illustrates a cube with an index of refraction of 1.33, similar to water, featuring a normal map generated using smooth Perlin noise [25][6].

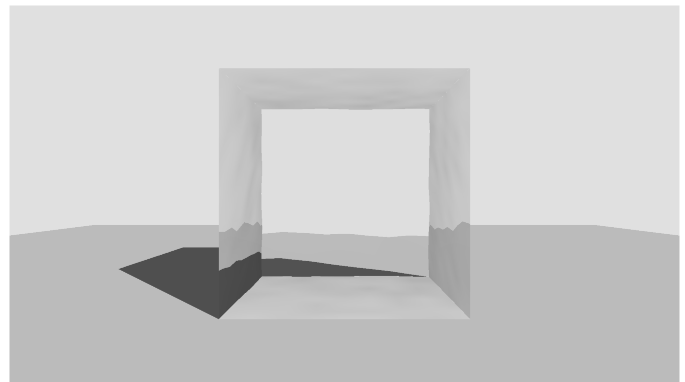

Fig. 10. A cube with reflection and refraction determined by Fresnel.

IX. Achieving Global Illumination

The currently described ray tracer renders shadows (as shown in Fig. 4) as completely dark or flat-shaded. Advanced ray tracers, as illustrated in Fig. 11, enhance shadows with sophisticated bounce lighting – in this case, light reflected from the plane onto the sphere. Indirect lighting encompasses any light that interacts with more than one surface.

Two forms of indirect lighting have been described so far: glossy and translucent. More complex scenarios requiring realistic light interactions are handled by algorithms known as global illumination techniques [12].

Global illumination typically employs bounce rays, similar to reflection rays. Unlike reflection rays, bounce rays have random directions, enabling the simulation of complex behaviors,

[6]Perlin noise: used to create natural patterns like clouds or terrain using random gradient vectors placed at grid points smoothed by a fade function.

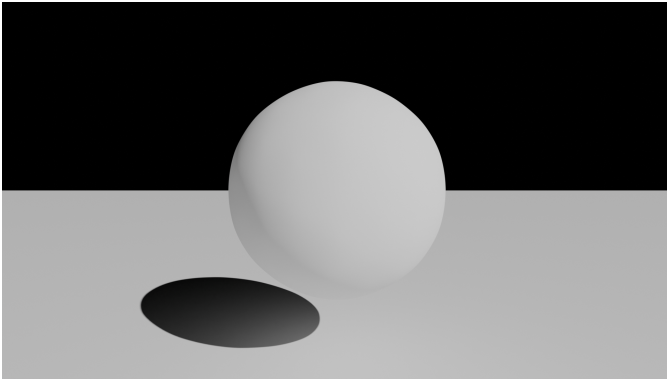

Fig. 11. A sphere and a ground plane rendered with Cycles [3].

such as diffuse surfaces. The rendering equation [13] broadly models arbitrary lighting scenarios:

$$\int_{\Omega} L(\omega_\mathrm{i}) F(\omega_\mathrm{i}, \omega_\mathrm{o}, n)(\omega_\mathrm{i} \cdot n) \,\mathrm{d}\,\omega_\mathrm{i}.$$

This equation describes the light intensity the surface emits in the direction ω_o (e.g., into the camera). Here, $L(\omega_i)$ denotes light intensity received from the direction ω_i, ω_i and ω_o represent incoming and outgoing directions, n is the surface normal vector, and $F(\omega_\mathrm{i}, \omega_\mathrm{o}, n)$ is the Bidirectional Reflectance Distribution Function [8] (BRDF) specifying the amount of light reflected from the direction ω_i in the direction ω_o. The Lambertian model is a specific instance of this model, where the BRDF is a constant function.

X. Importance Sampling

The indirect lighting approach described suffers from slow sampling processes. The most impactful lighting comes from a few specific directions, such as the nearest bright source for diffuse surfaces. Ray tracers employ importance sampling [29] to mitigate this, where rays are sampled from a specially crafted non-uniform probability distribution. This approach adjusts the sample values by distribution-dependent factors to compensate for the probability distributions being non-uniform.

For example, the cosine-weighted distribution [28] is commonly used with the Lambertian model. This distribution ensures that the average of all samples aligns with the expected value of the rendering equation. The later a bounce occurs, the less it generally contributes to the resulting image. This is because its value is multiplied by multiple BRDFs and $\omega_i \cdot n$ terms, most of which are less than 1. Therefore, to optimize performance, typically, only one new ray per bounce is created, with multiple rays originating from the camera and their values averaged.

Fig. 12 shows the Cornell box 3D test model rendered using our cosine-weighted sampling implementation at 128 samples per pixel. Increasing the number of samples reduces image noise as the average converges toward the rendering equation's actual value.

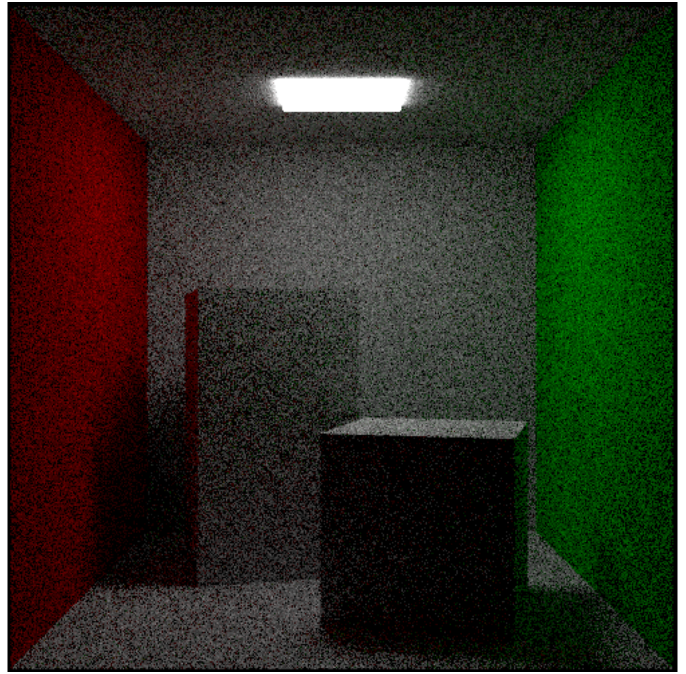

Fig. 12. Cornell box.

XI. Results

We explored different ray-tracing algorithms and implemented them ourselves, presenting the results in the corresponding sections of this report. We measured the average time to render a single frame at each improvement iteration. The results, along with the standard deviation of the measurements, can be seen in Table I. Performance measurements were conducted using mangohud logging [9][7] on an Nvidia GeForce RTX 3080 GPU, rendering at a resolution of 500 by 500 pixels. Note that the scene complexity presented in the table is just an indication. While the performance of a naive ray tracer grows linearly as the scene complexity grows, even implementing a single AABB check for an intersection with the entire scene can affect the performance drastically. The code and scenes used for the rendering are available on our GitHub organization page [14].

XII. Conclusion

Throughout this project, we explored various ray-tracing techniques. Initially, we developed a real-time ray tracer capable of handling glossy reflections, refraction, and direct lighting. Subsequent iterations extended its capabilities to encompass complex lighting scenarios with global illumination. We also investigated various methods to optimize rendering performance. Our implementation maintained real-time performance for every level of visual fidelity except for global illumination. However, as expected, performance decreased as visual fidelity increased, even in earlier stages. The better

[7]Mangohud – an overlay for OpenGL and Vulkan applications to monitor different metrics, including frames per second.

TABLE I
RAY TRACER PERFORMANCE.

Last feature	Scene	Avg frame time	Std. dev
Without arbitrary scene loading			
Flat shading (Fig. 2)	1 primitive	0.29ms	0.04ms
Blinn-Phong (Fig. 4)	1 primitive	0.29ms	0.04ms
Shadow rays (Fig. 6)	2 primitives	0.30ms	0.04ms
Reflections (Fig. 7)	3 primitives	0.30ms	0.05ms
With arbitrary scene loading			
Iterative	15743 faces	55ms	1ms
Iterative	62975 faces	226ms	2ms
BVH	15743 faces	0.66ms	0.18ms
BVH	62975 faces	0.91ms	0.21ms
Front-to-back	15743 faces	0.38ms	0.12ms
Front-to-back	62975 faces	0.43ms	0.13ms
GI (Fig. 12)	45 faces	125ms	2ms

the visual fidelity of the image, the more optimizations were required for it to work within reasonable timeframes.

XIII. FUTURE WORK

A. Denoising

There are ways to improve both the visual fidelity and the performance further. Ray-tracing noise patterns are predictable, allowing post-processing techniques to improve the final image quality. Denoising [1] enhances visual quality, removing most of the noise without substantial increases in render time.

B. Visual fidelity

We implemented light transport according to bidirectional reflectance distribution functions. This is not a complete lighting model. The assumption is that light exits the surface at the entry point. In reality, this is not always the case. Phenomena such as volumetric scattering [10], necessary for accurate fog simulation, cannot be fully modeled this way.

C. Performance

A common technique for further improving performance is adaptive sampling, allocating more rays to those areas of the image that are more noisy. In this way, a similar total sample count converges on a clean image significantly faster. Yet another approach is ReSTIR [2]. ReSTIR suggests resampling the global illumination at each step, sampling more from the directions that produce the most light, and reusing the data from neighboring pixels and previous frames.

REFERENCES

[1] A. T. Áfra, "Intel® Open Image Denoise," 2024, https://www.openimagedenoise.org.

[2] B. Bitterli, C. Wyman, M. Pharr, P. Shirley, A. Lefohn, and W. Jarosz, "Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting," *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, vol. 39, no. 4, July 2020.

[3] Blender Foundation, "Cycles Renderer," Computer software, 2024. [Online]. Available: https://www.blender.org/features/rendering

[4] J. F. Blinn, "Models of light reflection for computer synthesized pictures," *SIGGRAPH Comput. Graph.*, vol. 11, no. 2, p. 192–198, jul 1977. [Online]. Available: https://doi.org/10.1145/965141.563893

[5] ——, "Simulation of wrinkled surfaces," *SIGGRAPH Comput. Graph.*, vol. 12, no. 3, p. 286–292, aug 1978. [Online]. Available: https://doi.org/10.1145/965139.507101

[6] J. F. Blinn and M. E. Newell, "Texture and reflection in computer generated images," *Proceedings of the 3rd annual conference on Computer graphics and interactive techniques*, 1976. [Online]. Available: https://api.semanticscholar.org/CorpusID:408793

[7] J. H. Clark, "Hierarchical geometric models for visible surface algorithms," *Seminal graphics: pioneering efforts that shaped the field*, 1976. [Online]. Available: https://api.semanticscholar.org/CorpusID:8231831

[8] R. L. Cook and K. E. Torrance, "A reflectance model for computer graphics," *ACM Trans. Graph.*, vol. 1, no. 1, p. 7–24, jan 1982. [Online]. Available: https://doi.org/10.1145/357290.357293

[9] flightlessmango, "MangoHud," 2024. [Online]. Available: https://github.com/flightlessmango/MangoHud

[10] J. Fong, M. Wrenninge, C. Kulla, and R. Habel, "Production volume rendering: Siggraph 2017 course," in *ACM SIGGRAPH 2017 Courses*, ser. SIGGRAPH '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3084873.3084907

[11] A. Fresnel, *Oeuvres Complètes d'Augustin Fresnel*. Paris: Imprimerie Imperiale, 1866.

[12] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile, "Modeling the interaction of light between diffuse surfaces," *SIGGRAPH Comput. Graph.*, vol. 18, no. 3, p. 213–222, jan 1984. [Online]. Available: https://doi.org/10.1145/964965.808601

[13] J. T. Kajiya, "The rendering equation," *SIGGRAPH Comput. Graph.*, vol. 20, no. 4, p. 143–150, aug 1986. [Online]. Available: https://doi.org/10.1145/15886.15902

[14] B. O. Kateryna Kovalchuk, Radomyr Husiev and R. Zaletskyi, "Github organization." [Online]. Available: https://github.com/triffois

[15] T. L. Kay and J. T. Kajiya, "Ray tracing complex scenes," *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, 1986. [Online]. Available: https://api.semanticscholar.org/CorpusID:624126

[16] Khronos Group, "Fragment Shader - OpenGL Wiki," 2024. [Online]. Available: https://www.khronos.org/opengl/wiki/Fragment_Shader

[17] ——, "glTF – GL Transmission Format," 2024. [Online]. Available: https://www.khronos.org/gltf/

[18] ——, "OpenGL - Open Graphics Library," 2024. [Online]. Available: https://www.opengl.org/

[19] ——, "Shader Storage Buffer Object," 2024. [Online]. Available: https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object

[20] ——, "Uniform (GLSL) - OpenGL Wiki," 2024. [Online]. Available: https://www.khronos.org/opengl/wiki/Uniform_(GLSL)

[21] LearnOpenGL. Pbr: Theory. Accessed: 2024-07-16. [Online]. Available: https://learnopengl.com/PBR/Theory

[22] T. Möller and B. Trumbore, "Fast, minimum storage ray-triangle intersection," *Journal of Graphics Tools*, vol. 2, no. 1, pp. 21–28, 1997. [Online]. Available: https://doi.org/10.1080/10867651.1997.10487468

[23] NVIDIA Corporation. Life of a Triangle - NVIDIA's Logical Pipeline. Accessed: 2024-07-16. [Online]. Available: https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline

[24] ——, "CUDA Toolkit," 2024. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[25] K. Perlin, "An image synthesizer," *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, p. 287–296, jul 1985. [Online]. Available: https://doi.org/10.1145/325165.325247

[26] C. Schlick, "An inexpensive brdf model for physically-based rendering," *Computer Graphics Forum*, vol. 13, no. 3, pp. 233–246, 1994. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.1330233

[27] D. Seyb, P.-P. Sloan, A. Silvennoinen, M. Iwanicki, and W. Jarosz, "The design and evolution of the uberbake light baking system," *ACM Trans. Graph.*, vol. 39, no. 4, aug 2020. [Online]. Available: https://doi.org/10.1145/3386569.3392394

[28] P. Shirley, T. D. Black, and S. Hollasch. (2024, April) Ray tracing: The rest of your life. [Online]. Available: https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html

[29] E. Veach and L. Guibas, "Optimally combining sampling techniques for monte carlo rendering," in *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, vol. 95, 01 1995, pp. 419–428.

[30] T. Whitted, "An improved illumination model for shaded display," *Commun. ACM*, vol. 23, no. 6, p. 343–349, jun 1980. [Online]. Available: https://doi.org/10.1145/358876.358882