UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 2885

PATH TRACING AND PHYSICALLY BASED RENDERING

Ivan Karlović

Zagreb, June 2022

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 2885

PATH TRACING AND PHYSICALLY BASED RENDERING

Ivan Karlović

Zagreb, June 2022

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Zagreb, 11 March 2022

MASTER THESIS ASSIGNMENT No. 2885

Student:	**Ivan Karlović (0108069417)**
Study:	Computing
Profile:	Computer Science
Mentor:	prof. Željka Mihajlović

Title: **Path Tracing and Physically Based Rendering**

Description:

Explore path tracing method. Investigate physically based rendering for arbitrary surfaces and scene. Develop an application for path tracing on arbitrary scene. Especially pay attention to physically based rendering approach. Discuss the influence of parameters of the models on the output. Evaluate the results and the implemented algorithms. Implement the appropriate software solution. Use C++ programming language and graphics and computing API Vulkan. Make the results of the thesis available online. Supply algorithms, source codes and results with appropriate explanations and documentation. Reference the used literature and acknowledge received help.

Submission date: 27 June 2022

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 11. ožujka 2022.

DIPLOMSKI ZADATAK br. 2885

Pristupnik:	**Ivan Karlović (0108069417)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentorica:	prof. dr. sc. Željka Mihajlović

Zadatak: **Postupak praćenja puta i fizikalno temeljen prikaz površine**

Opis zadatka:

Proučiti postupak praćenja puta. Proučiti fizikalno temeljen način izračuna osvjetljenja proizvoljne površine u sceni. Razraditi postupak prikaza scene temeljen na metodi praćenja puta. Posebno obratiti požnju na fizikalno temeljen izračun osvjetljenja površine. Programski ostvariti razrađeni postupak i na nizu scena prikazati rezultate. Načiniti testiranje na nizu primjera. Analizirati i ocijeniti ostvarene rezultate. Diskutirati upotrebljivost ostvarenih rezultata kao i moguća proširenja. Izraditi odgovarajući programski proizvod. Koristiti programski programski jezik C++ i grafičko programsko sučelje Vulkan. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 27. lipnja 2022.

CONTENTS

1. Introduction

Ever since the first graphics accelerators, the complexity and fidelity of a scene that can be rendered in real-time have been increasing. Excluding several simple rendering methods used in the early days, the most commonly used rendering method is rasterization. The geometry of the scene is divided into meshes made out of polygons (in most cases triangles) which are then projected onto the viewing plane. While this method makes it possible to visualize 3D geometry on 2D screens, it leaves much to be desired when it comes to light interaction within the scene. For example, to draw shadows, the scene needs to be rendered multiple times for each light source, increasing the cost of the rendering process while not producing true-to-life results.

Path tracing is an umbrella term for a family of rendering methods that aim to solve the problem of unrealistic lighting effects within the scene. It is considered a holy grail of photo-realistic computer graphics (4). Instead of projecting the scene onto the viewing plane, rays are dispatched from the camera backtracking the path that the light would have had to take to reach the sensor. While these methods very accurately light the scene, the brute force implementations are intractable, having exponential time complexity.

2. Tools and techologies

Programming language C++

C++ is an object-oriented, low-level, high-performance programming language. Due to its speed, it's often the first choice when developing 3D applications. The standard used in implementation is C++23.

Graphics API Vulkan

Vulkan is a graphics API defined by the Khronos consortium made out of over 150 software and hardware companies. It is a cross-platform API targeted primarily at GPU utilization. It is not a successor to OpenGL, however, it does offer performance benefits over it in exchange for verbosity and developer responsibility when developing an application. It is released as a C header file, but many wrappers for various languages exist, including C++. Along with Vulkan, a new intermediate shading language is introduced, SPIR-V. It can be compiled from both GLSL and HLSL shaders. The current Vulkan version is 1.3.217.

Vulkan Memory Allocator library

Vulkan Memory Allocator (VMA) is a small library released by AMD. It handles the intricacies of allocating memory in Vulkan for buffer and image storage. While it takes away some control from the developer, most applications do not need more complex memory management than VMA offers.

GLFW library

GLFW is a Graphics Library FrameWork, providing a compatibility layer between the application and the operating systems. It handles the creation of the application window and rendering surface, as well as inputs and events from peripherals.

nvmath library

Nvmath is one of the modules of the Nvpro core repository. It is a math library released by Nvidia.

tinygltf library

TinyGLTF is a header-only C++ library used to load glTF 2.0 models. glTF is a specification developed by the Khronos consortium aimed at efficient transmission and loading of 3D scenes and models.

MikkTSpace library

MikkTSpace is a small library used to consistently produce normal maps in tangent space.

3. Hardware raytracing in Vulkan

Until a few years ago, all raytracing done on commercial hardware was done on the CPU. However, in late 2018, Nvidia released a series of graphics cards that had hardware-accelerated ray tracing.

However, if done naively, ray tracing is still intractable. To manage complexity, a reduced number of rays are used to generate an image, but this comes at a cost of noise in the image. There are two main approaches used to reduce the generated noise. Many denoisers have been developed in recent years that take the raw output of a path tracer and clean up the image, trying to fill the blanks using neighbouring pixels or other scene data. On the other hand, many path tracing algorithms have been developed that can use fewer rays while maintaining the quality of the rendered image. One such algorithm is Reservoid-based Spatio-Temporal Importance Resampling or ReSTIR for short. The algorithm is linear in terms of the number of lights. The goal of this paper is to implement a version of this algorithm and inspect the acquired image quality.

Support for hardware raytracing in Vulkan comes in form of several extensions.

3.1. VK_KHR_ray_tracing_pipeline

This extension brings support for a new type of pipeline to Vulkan. The pipeline is much simpler than its rasterization counterpart (VkGraphicsPipeline).

Vertex and fragment shaders of the rasterization pipeline can be bound as specific meshes are being rendered. When it comes to ray tracing, a ray can hit any object in the scene which could require one of many shaders to be invoked. To resolve this issue, a shader binding table is introduced. It holds handles of every shader that could be invoked during the ray tracing process. Various handles are grouped in shader groups. Each type of shader is in its own group except for any hit and closest hit shaders which form hit groups. Reference to the shader binding table is passed as an argument to the vkCmdTraceRays(...) method.

It also introduces several new shader types:

- Ray generation shader
- Any hit shader
- Closest hit shader
- Miss shader
- Intersection shader
- Callable shader

3.1.1. Ray generation shader

Ray generation shaders are the entry point of the ray tracing pipeline. They are invoked when vkCmdTraceRays(...) is called, and their workload is submitted in three dimensions, similar to compute shaders. The first two dimensions are most often dimensions of the screen in pixels, while the last dimension can be set to one or used to determine the number of rays traced per sample. Within the shader a function traceRayEXT(...) can be invoked, tracing a ray through the bound acceleration structure and writing the result in the bound ray payload. This process is hardware accelerated, seeking intersections between the ray and the geometry within the acceleration structure.

3.1.2. Any hit shader

As the ray generated by the ray generation shader is traced through the acceleration structure, intersections are detected by hardware. On each of the detected intersections (or hits), any hit shader is invoked. It can perform arbitrary calculations and store the results in the ray payload. Since this shader is called for every intersection detected, it is advised to keep this shader as simple as possible or to not implement it at all, since it is optional.

3.1.3. Closest hit shader

This shader works similarly to any hit shader. The only difference is that this shader is invoked only after the ray was completely traced. The shader is executed over the closest hit. This shader isn't optional, but won't be executed if no geometry is hit.

3.1.4. Miss shader

This shader is invoked after the ray has been fully traced if it intersects no gravity. It's useful for visibility testing and can be used to sample the skybox.

3.1.5. Intersection shader

While the geometry in acceleration structure is usually made out of triangles, axis-aligned bounding boxes can be used instead. With the triangles, the intersection is resolved automatically, but axis-aligned bounding boxes need an additional intersection shader to resolve them.

3.1.6. Callable shaders

Callable shaders are used to optimize code paths with varying execution times. While a single shader with a single if-else will need to execute both paths for every shader, even if only a small percentage of shaders take the longer path, two callable shaders can be used, one for each branch, that will execute separately.

3.2. VK_KHR_acceleration_structure

This extension introduces the concept of acceleration structures to Vulkan. Instead of testing the intersection of generated rays with every triangle in the scene, an acceleration structure is built that greatly accelerates this process, reducing its complexity from O(n) to O(log(n)) in terms of the number of triangles. This enables faster tracing of larger scenes.

The acceleration structures are built from two components, bottom-level acceleration structures and top-level acceleration structures. Bottom-level acceleration structures contain scene geometry. Its implementation is hardware dependent, but it is reasonable to assume that it is some kind of bounding volume hierarchy. Multiple bottom-level structures are bound together into a top-level acceleration structure against which rays are traced. The same bottom-level acceleration structure can be bound to the same top-level acceleration structure multiple times with a different offset effectively enabling geometry instancing. Each instance is given a unique identifier which is later available in the shader.

Time to build an acceleration structure is not insignificant and the extension allows for several methods of building it. It is possible to build the structure using the device

(in this case the GPU) or the host (the CPU). The acceleration structures can also be built so they are faster to trace against or that they are faster to build. Acceleration structures depicting static geometry need to be built only once, so it is advisable to use the ones that can be traced faster. On the other hand, the acceleration structures that need to be rebuilt during execution might better be served by the version that enables faster build times. As a middle ground, there are acceleration structures that can be updated, which is significantly faster than the full rebuild. This method is preferred for geometry that deforms to some extent since the process is less effective the greater the difference from the original geometry.

4. ReSTIR algorithm

ReSTIR (1) algorithm can be divided into several stages. The scene is first rendered to several buffers storing world positions and any other data needed to calculate the lighting. In the second stage, for each point, several lights are sampled from the pool of all the lights in the scene and they are added to the reservoir. The third stage introduces temporal reuse combining the current reservoir with the reservoir of the pixel where the current world position was in the previous frame. The following stage implements spatial reuse, combining the reservoirs of the neighbouring pixels with the current reservoir. In the final stage, based on the lighting data stored in the reservoir, the final colour of the pixel is calculated and drawn on the screen.

4.1. Sampling the light

The reflected radiance of a point y in the direction $\overrightarrow{\omega}$ due to direct lighting can be modelled with the integral over all light-emitting surfaces A:

$$L(y, w) = \int_A \rho(x, y, w) L_e(x, y) G(x, y) V(x, y) dA_x \tag{4.1}$$

ρ is the BSDF, L_e is the emitted radiance, G is the geometry term and V denotes the visibility between points x and y. This integral can simplified to:

$$L(x) = \int_A \rho(x) L_e(x) G(x) V(x) dx \tag{4.2}$$

This integral does not have a solution in closed form.

To compute the value of this integral, Monte Carlo integration (figure 4.1) is used. In its most basic from N random samples are uniformly picked from the integral domain and their average is computed as the result. If the expectance of the function being sampled from is

$$E[f(x)] = \int_A f(x) dx \tag{4.3}$$

the Monte Carlo estimator is

$$Q_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x)}{p(x)} \tag{4.4}$$

where $p(x)$ is the probability distribution function used to sample function $f(x)$.

Due to the law of large numbers, the Monte Carlo estimator is unbiased. As the number of samples approaches infinity, the result approaches the real value:

$$\lim_{N \to \infty} Q_N = E[f(x)] \tag{4.5}$$

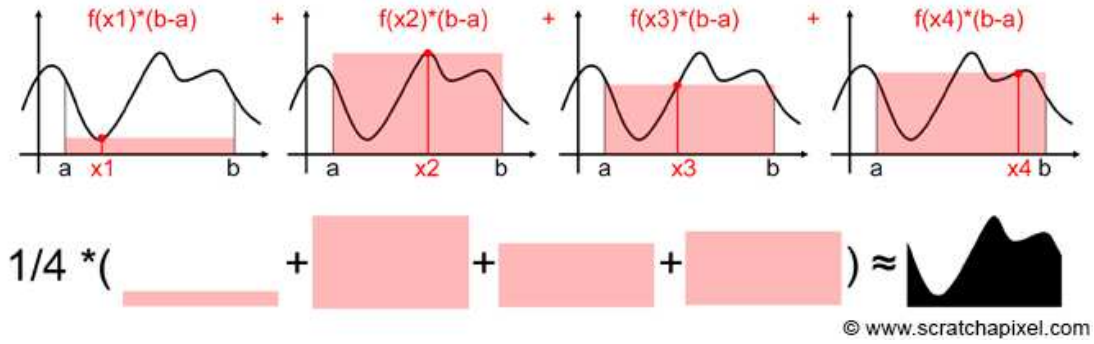

Figure 4.1: Monte Carlo integration.

This naive approach often introduces a large variance, however, many algorithms were developed over the years that aim to improve the accuracy of the process and reduce its variance.

4.1.1. Importance sampling

One of the simplest methods of reducing the variance of the Monte Carlo estimator is to use importance sampling (3) (figure 4.2). Instead of using $p(x)$ to sample from $f(x)$, the expression can be transformed to sample from some other, usually simpler, distribution $q(x)$. First, a probabilistic one is introduced to the equation 4.3:

$$\begin{aligned} E[f(x)] &= \int_{\Omega} \frac{q(x)}{q(x)} \frac{f(x)}{p(x)} dx \\ &\text{where } q(x) = 0 \implies p(x) = 0 \end{aligned} \tag{4.6}$$

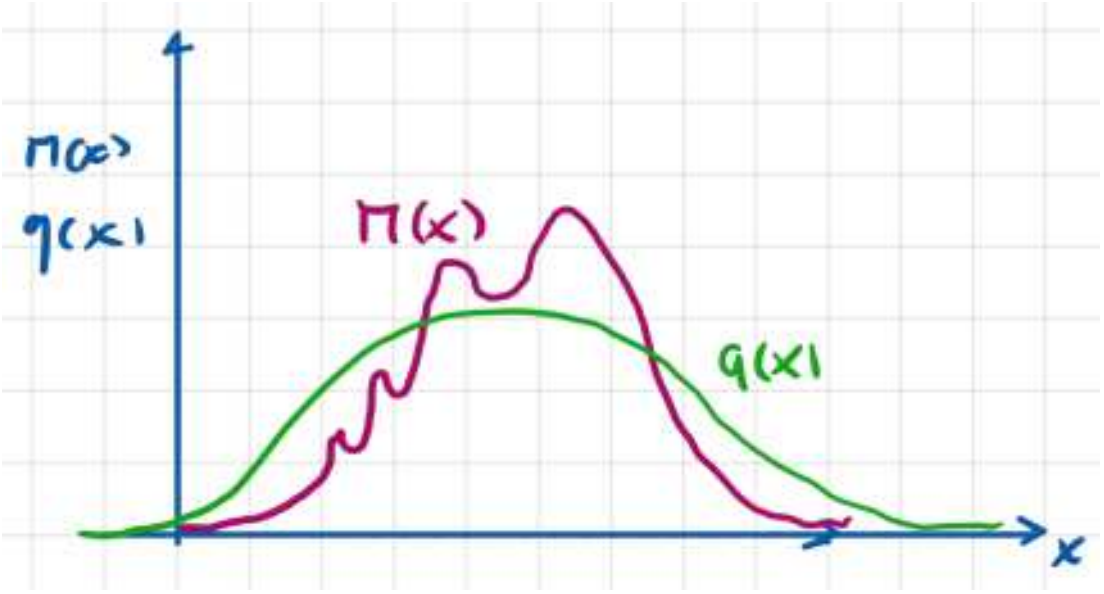

Figure 4.2: Visualization of Importance sampling. Function $p(x)$ is marked as $M(x)$ in the image.

This equation can be rewritten as follows:

$$\begin{aligned} E[f(x)] &= \int_{\Omega} \frac{q(x)}{p(x)} \frac{f(x)}{q(x)} dx \\ E[f(x)] &= \int_{\Omega} w(x) \frac{f(x)}{q(x)} dx \\ E[f(x)] &= \int_{\Omega} \frac{f'(x)}{q(x)} dx \\ &\text{where } q(x) = 0 \implies p(x) = 0 \end{aligned} \tag{4.7}$$

Expression $w(x)$ is often called the importance weight, giving the name to the method, while $f'(x)$ is the weight-adjusted function $f(x)$ which is being sampled using the new probability density function $q(x)$. Function $q(x)$ can be any function as long as it satisfies the condition under the equation 4.6. The more proportional $q(x)$ is to $p(x)$, the smaller the variance. The estimator can be rewritten as s sum when taking samples:

$$E[f(x)] = \frac{1}{N} \sum_{i=1}^{N} \frac{f'(x)}{q(x)} \tag{4.8}$$

This estimator, like Monte Carlo one, is unbiased.

4.1.2. Multiple importance sampling (MIS)

While importance sampling improves the results of the naive Monte Carlo integration, the single probability density function might not give great results in all situations. Multiple importance sampling (figure 4.3) is the extension of importance sampling that enables mixing and matching several different importance sampling methods:

$$E[F] = \frac{1}{N} \sum_{i=0}^{N} E[F_i] \tag{4.9}$$

where F_i is an importance sampling methods. Since this is a linear combination of importance sampling methods, this estimator is also unbiased. The equation presented above calculates the average of the methods, but it is possible to get better results when using the weighted average:

$$E[F] = \sum_{i=0}^{N} \omega_i E[F_i] \tag{4.10}$$

This allows to prefer certain importance sampling methods for the specific parts of the domain. As long as the following equation is satisfied, the estimator remains unbiased:

$$\sum \omega_i = 1 \tag{4.11}$$

Even better results can be achieved when the weight is dependent upon the sample:

$$E[F] = \sum_{i=1}^{N} \frac{1}{N_i} \sum_{j=1}^{N_i} \omega_i(x_{i,j}) \frac{f(x_{i,j})}{p_i(x_{i,j})} \tag{4.12}$$

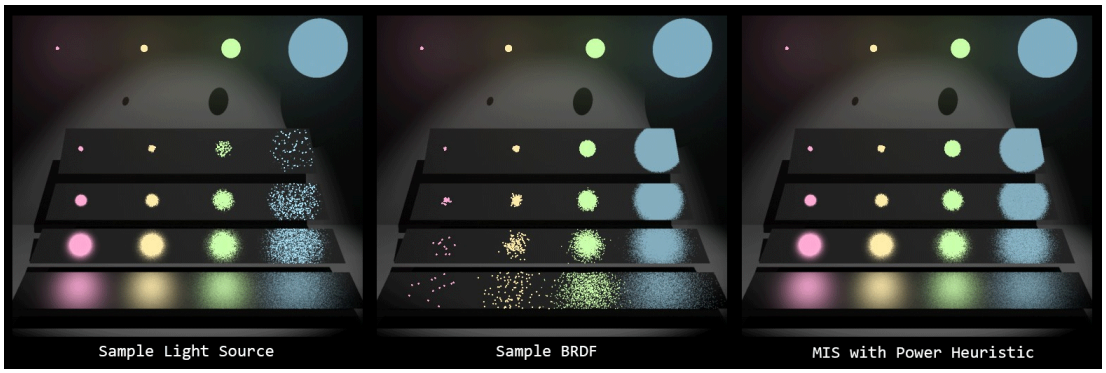

Figure 4.3: Combination of two sampling methods, each with their own strengths and weaknesses, which produces an overall better image, using MIS

To keep the estimator unbiased, the following needs to hold:

$$\begin{aligned} f(x) \neq 0 \implies \sum_{i=0}^{N} \omega_i(x) = 1 \\ h_i(x) = 0 \implies \omega_i(x) = 0 \end{aligned} \tag{4.13}$$

4.1.3. Resampled importance sampling (RIS)

While importance sampling improves the estimation, it still requires several samples to give a good result. Looking at the importance sampling estimator:

$$\begin{aligned} E[f(x)] = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x)}{p(x)} \\ E[f(x)] = \frac{1}{N} \sum_{i=1}^{N} C \end{aligned} \tag{4.14}$$

it would be possible to take a single sample (thus having perfect importance sampling) if the value of C was the same as the value of the integral being evaluated:

$$C = \int f(x)dx \tag{4.15}$$

Since approximations are already being used, C can be approximated:

$$C = \frac{1}{M} \sum_{j=1}^{M} \frac{f(x_j)}{p(x_j)} \tag{4.16}$$

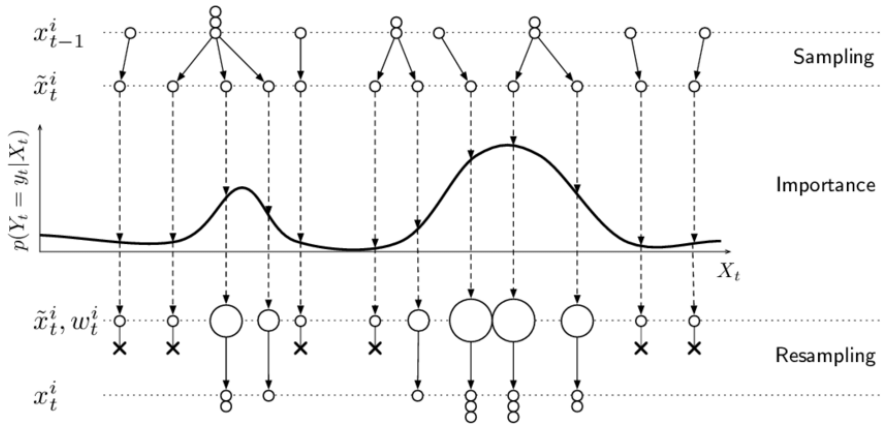

Figure 4.4: Visualization of resampled importance sampling.

Plugging equation 4.16 into 4.14 results in the resampled importance sampling (2) (figure 4.4):

$$C = \frac{1}{N} \sum_{i=1}^{N} \left[\frac{f(x_i)}{\hat{p}(x_i)} \frac{1}{M} \sum_{j=1}^{M} \frac{\hat{p}(x_j)}{p(x_j)} \right] \tag{4.17}$$

This is equivalent of taking M samples from $p(x)$ and turning them into N better ones from $\hat{p}(x)$. $\hat{p}(x)$ is an arbitrary function that can be selected for optimal behavior, while $p(x)$ should be cheap. RIS can also be combined with MIS to further improve the quality of the result.

4.1.4. Weighted reservoir sampling (WRS)

Weighted reservoir sampling (figure 4.5) is a family of algorithms that sample N random elements from a stream of M elements in a single pass. Each element has weight $w(x_i)$ and the probability to be selected as such:

$$P_i = \frac{w(x_i)}{\sum_{j=1}^{M} w(x_j)} \tag{4.18}$$

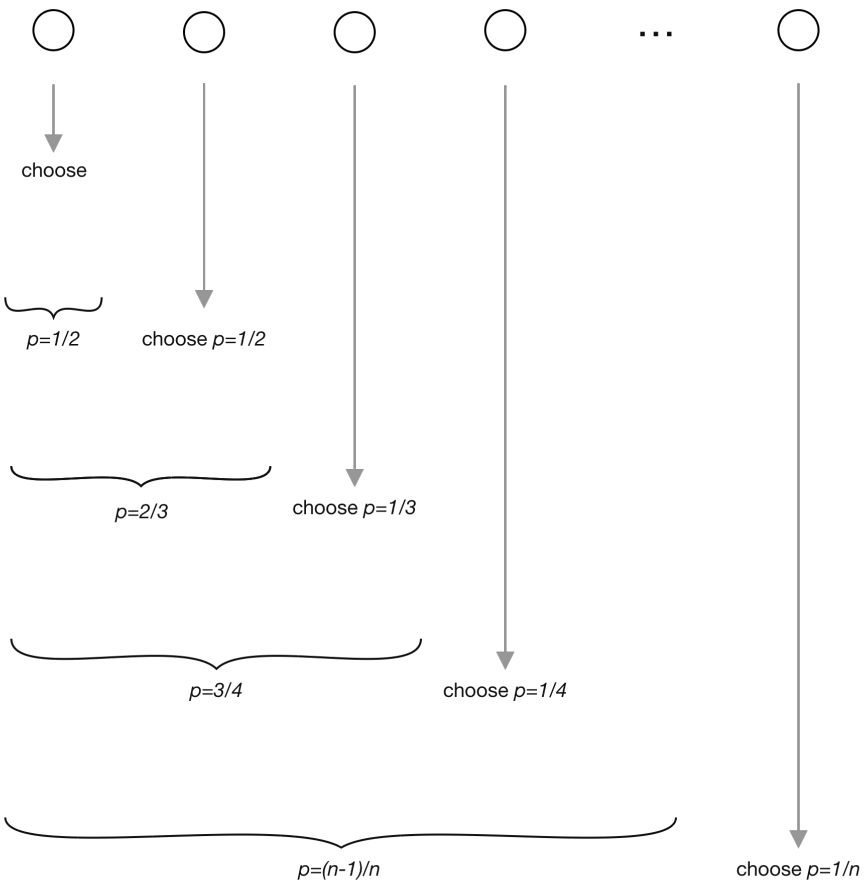

Figure 4.5: Visualization of reservoir sampling. In this case, all weights are equal.

The reservoir only needs to keep N elements in the memory and the stream size M does not need to be known in advance. In this case, one element in the reservoir is sufficient, although the principle can be expanded to multiple elements trivially. The reservoir keeps track of the element in the memory and the cumulative weight of all previously seen elements. After m samples have been processed, sample x_m occurs with the following probability:

$$P_m = \frac{w(x_m)}{\sum_{j=1}^{m} w(x_j)} \tag{4.19}$$

The current element x_m is then replaced with the following element x_{m+1} with probability:

$$\frac{w(x_{m+1})}{\sum_{j=1}^{m+1} w(x_j)} \tag{4.20}$$

Any previous sample x_i in the stream is in the reservoir with the probability:

$$\frac{w(x_m)}{\sum_{j=1}^{m} w(x_j)} \left(1 - \frac{w(x_{m+1})}{\sum_{j=1}^{m+1} w(x_j)}\right) = \frac{w(x_m)}{\sum_{j=1}^{m+1} w(x_j)} \tag{4.21}$$

The sum over probabilities of all m samples that have been evaluated so far shows:

$$\sum_{i=1}^{m} \frac{w(x_i)}{\sum_{j=1}^{m} w(x_j)} = \frac{\sum_{i=1}^{m} w(x_i)}{\sum_{j=1}^{m} w(x_j)} = 1 \tag{4.22}$$

This makes it a very efficient algorithm that integrates a potentially large number of samples while having constant memory requirements.

4.1.5. Streaming RIS using reservoir sampling

Resampled importance sampling and reservoir sampling can be combined by feeding high-quality samples generated by RIS to the reservoir. Samples for RIS are uniformly generated over the area of emitters and use $\hat{p}(x) = \rho(x)L_e(x)G(x)$, the unshadowed path contribution, as the target distribution. Shadow rays are traced only for N samples that survive the selection process.

4.1.6. Alias method

The alias method is used to enable picking a sample from distribution in O(1) time when working with a discrete number of samples. The probability of all samples is accumulated and divided by N, giving the average probability. All of the samples are divided into two queues, those with probability higher than the average ($bigs$) and those

with probability lower than the average ($smalls$). A new array is created, containing N buckets. Each bucket contains $small$ probability, big (alias) original probability and the big (alias) id. The first element is taken from each queue. The probability of the $small$ is stored in the bucket at an index equal to its id. The probability of the big is reduced in a way that the $small$ probability and the reduced amount add up to the average. The id of the big is also stored in the bucket. If the remaining probability of the big is still higher than the average, it is returned to a higher probability queue.

This process is repeated until there are no more pairs. The remaining samples will be transformed into buckets with no alias and probability equal to the average probability.

The whole alias table can be normalized so that the total probability of each bucket is equal to 1, which simplifies sampling.

To sample from this table, a random number n is picked between 1 and N. This selects the bucket. Assuming the table was normalized, another random number between 0 and 1 is picked, determining whether to pick the original sample in the bucket or its alias.

4.1.7. Spatiotemporal reuse

Neighbouring pixels often have a significant correlation between their probability density functions. Given the above mentioned distribution $\hat{p}(x) = \rho(x)L_e(x)G(x)$, both BSDF and geometry factors tend to be similar between neighbouring pixels. While it would be possible to store all the samples and their weight and then have a second pass that takes into account the samples of the neighbouring pixels, in this case, this is not necessary.

Each reservoir stores the chosen sample and the cumulative weight of all the samples encountered. Two reservoirs can be treated as weighted samples and, using the principle of weighted reservoir sampling, be combined into a new reservoir which has seen all of the samples in the streams of both initial reservoirs. Unlike the approach of having multiple passes, this method can be executed in constant time - combining k different reservoirs can be done in $O(k)$ time. However, since two reservoirs have been computed using different target distributions, $\hat{p}$ and $\hat{p}'$, the resulting sample needs to be reweighed with the term $\hat{p}/\hat{p}'$ to avoid bias.

Spatial reuse

To achieve spatial reuse M samples are created for each pixel using RIS in the first pass. Samples are stored in reservoirs, one for each pixel. In the subsequent pass, reservoirs of k neighbours of each pixel are combined with their original reservoir. While the work done here is $O(M + k)$, each pixel effectively went through $M * k$ samples. This can be further improved by repeating the second pass n times. This increases the work done to $O(M + nK)$, but the pixels each get $M * k^n$ samples

Temporal reuse

Both in offline and interactive rendering, images are often rendered sequentially. This means that information from the previous frame can be used to improve the results of the current one. Similar to spatial reuse, an additional step is added where the reservoir of the pixel is combined with the equivalent pixel of the previous frame (computed using motion vectors). Note that this has transitive properties since the reservoir of the previous frame already has information from previous frames built into its reservoir. So, the reservoir of the current pixel isn't being combined only with the data from the previous frame, but the data of all the preceding frames. However, this data can be out of date, so the weight of the reservoir of the previous frame should be reduced.

Visibility reuse

The selected $\hat{p}$ distribution is the unshadowed path contribution, so $\hat{p}$ does not sample f perfectly, introducing noise to the image. To combat this, the selected reservoir sample is checked for occlusions. If the sample is occluded, it is discarded, reducing the overall noise.

5. Implementation

The implementation of the ReSTIR algorithm can be broken down into several sections. First, the scene is loaded using tinyGLTF and the data is copied over to the GPU. The lights are put in the alias table to speed up picking samples. Finally, four passes are made on the GPU, constructing the resulting image.

The application takes two parameters. The first parameter is the path to the *.gltf file that is to be rendered on screen. The second parameter is the number of point lights that are to be uniformly distributed across the scene if the scene contains no other sources of light.

5.1. Loading the scene

The scene is loaded as a *.gltf file using tinyGLTF. First, all of the lights are collected from the scene. Point lights store their position, colour and luminance. Triangle lights are stored as three points, the emissive factor of their material, luminance and the area of the light. The triangle light points are stored in world space. If there are no lights in the scene, several point lights are generated at random positions in the world. The alias table is then created out of triangle lights. The probability of each light is determined by the product of the luminance of the light and its area divided by the sum of the same products of all the lights. If there are no triangle lights present in the scene, the alias table is created out of point lights. The probability of a point light is calculated as its luminance divided by the sum of the luminance of all the point lights in the scene. Vertices and indices are then loaded into their respective buffers. Each vertex contains information about its position, normal, colour, tangent and uv coordinates. Since the scene can be made out of multiple nodes, a matrix is loaded for each node, containing the offset and rotation info.

The materials are loaded and are in form of one of two shading models:

- Metallic roughness

– Specular glossiness

Both models contain emissive factor, alpha mode (opaque, mask or blend) and the scale of the normal texture. Metallic materials contain base colour and metallic and roughness factors, while specular materials contain diffuse colour and specular and glossiness factors.

Finally, textures are loaded into the scene.

The implementation has several parameters:

– Light sample count - number of light samples evaluated when generating a reservoir
– Visibility reuse - whether to use visibility reuse or not
– Temporal reuse - whether to use temporal reuse or not
– Spatial reuse - whether to use spatial reuse or not
– Spatial reuse iterations - how many times to apply spatial reuse
– Spatial reuse neighbour count - how many neighbouring pixels to use
– Depth threshold - the maximal difference in depth between the neighbours
– Normal threshold - the maximal difference between the normals of the neighbours

5.2. Base pass

Base pass utilizes a basic graphics pipeline. Vertex shader transforms the data into projection space and passes it to the fragment shader. Fragment shader samples the various textures and stores the data into four textures:

– albedo
– normal
– material
– position

This data will be consumed by the next pass.

5.3. Reservoir generation and temporal and visibility reuse pass

This step uses the raytracing pipeline. Most of the work is done in the ray generation shader. The workload can be broken up into three parts:

- Sample/reservoir generation
- Visibility reuse
- Temporal reuse

5.3.1. Sample/reservoir generation

All of the data from the previous pass is loaded for the pixel. The alias table is sampled. If point lights are present, they are used as-is. If triangle lights are used, a random point is picked on the triangle, turning the sample into a point light for the purpose of the algorithm. Note that all of the random numbers generated so far (for sampling alias table and generating the point on the triangle) are generated within the shader and differ for each pixel. The pixel data and the light sample data are then passed to the BRDF and used to calculate $\hat{p}$. Only the luminance of the pixel is calculated here since the full colour is not needed at this point. The full colour is calculated in the final, lighting pass. Finally, with $\hat{p}$ calculated, the sample is added to the reservoir. This is repeated for the light sample count parameter. The default value used is 32. The value can be doubled or halved during the execution of the program, ranging from 1 to 1024.

Visibility reuse

If visibility reuse is enabled, for each sample in the reservoir a ray is traced from the position of the pixel in world space to the position of the sampled light in world space. If the ray is obstructed (meaning the pixel is shadowed), the sample is rejected. In this case, provided that there is only a single sample in the reservoir, all data for this pixel will come from temporal and spatial reuse, assuming they are enabled.

Temporal reuse

ProjectionView matrix is stored from the previous frame, as well as an image storing the final reservoirs. Multiplying the matrix with the world space position of the pixel gives its screen coordinates in the previous frame. This in combination with the image of the reservoirs makes it possible to retrieve the final reservoir data of that world space

position in the previous frame. If the position, the albedo colour or the normal of the pixel differs too much between the current and previous frame, the sample is rejected. If it passes all checks, the reservoirs from the previous and current frame are combined.

5.4. Spatial reuse pass

Now that all initial reservoirs have been calculated for each pixel, the spatial reuse step can be performed. This is done in the compute shader. A random neighbouring pixel is selected from a predefined radius (30.0f in this case). If the difference in depth between the neighbour and the main pixel is greater than the depth threshold parameter, the neighbour is rejected. Likewise, if the difference between the normal of the main pixel and the neighbour is greater than the threshold, the neighbour is rejected. If the neighbour passes both tests, its reservoir is combined with the main pixel reservoir. This process is repeated for several neighbours determined by the spatial reuse neighbour count parameter. This whole pass is then repeated several times controlled by the spatial reuse iterations parameter.

5.5. Lighting pass

The lightning pass is the final pass of the algorithm. It is a standard graphics pipeline pass that renders the final image onto a quad. The fragment shader takes the data from the textures generated in the first pass along with the data from the reservoir and passes it to the BRDF that calculates the final pixel colour. In the case of a reservoir with multiple samples, the final result is the average of BDRF calculations for each sample.

6. Performance

The hardware used to test the implementation can be viewed in the table 6.1.

Component type	Component used
CPU	Intel Core i9-9900KS
GPU	Nvidia GeForce RTX 2080Ti
RAM	32GB DDR4 3600CL16

Table 6.1: Specifications of the test machine

The performance is tested across two models:

- Sponza
- SciFi Helmet

For each model, there are several test configurations, as per table 6.2. All tests for a model are done at the same camera angle and position. The frame time is measured 1 minute after the parameters were set to ensure all cached data is "warmed up". 10 frames are captured and their frame times are averaged. Every test is done on 1920x1080 resolution. Sponza model has 500 uniformly distributed point lights and 262267 triangles, while the SciFi Helmet has 100 uniformly distributed point lights and 23358 triangles.

	Default	Light sample count	Visibility reuse	Temporal reuse
Light sample count	32	8, 32, 128, 512	32	32
Visibility reuse	ON	ON	ON, OFF	ON
Temporal reuse	ON	ON	ON	ON, OFF
Spatial reuse	ON	ON	ON	ON
Spatial neighbours	4	4	4	4
Spatial iterations	2	2	2	2
	Default	Spatial reuse	Spatial neighbours	Spatial iterations
Light sample count	32	32	32	32
Visibility reuse	ON	ON	ON	ON
Temporal reuse	ON	ON	ON	ON
Spatial reuse	ON	ON, OFF	ON	ON
Spatial neighbours	4	4	2, 4, 8, 16	4
Spatial iterations	2	2	2	1, 2, 3, 4

Table 6.2: Various tests and their parameters

6.1. Light sample test

Light sample count determines how many light samples are evaluated when forming the initial reservoir. For this test, they range from 8 to 512.

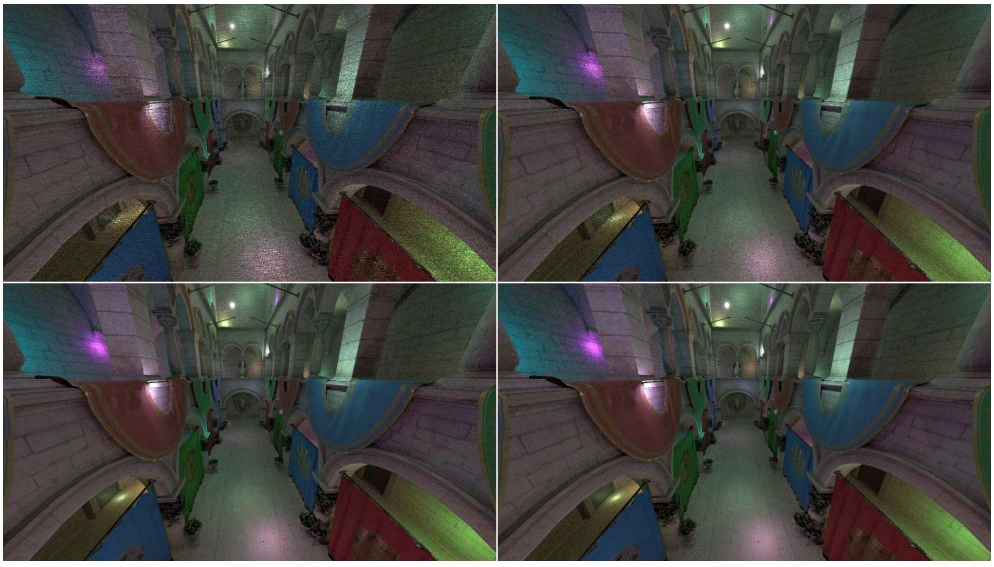

Figure 6.1: Sponza; 8, 32, 128 and 512 light samples

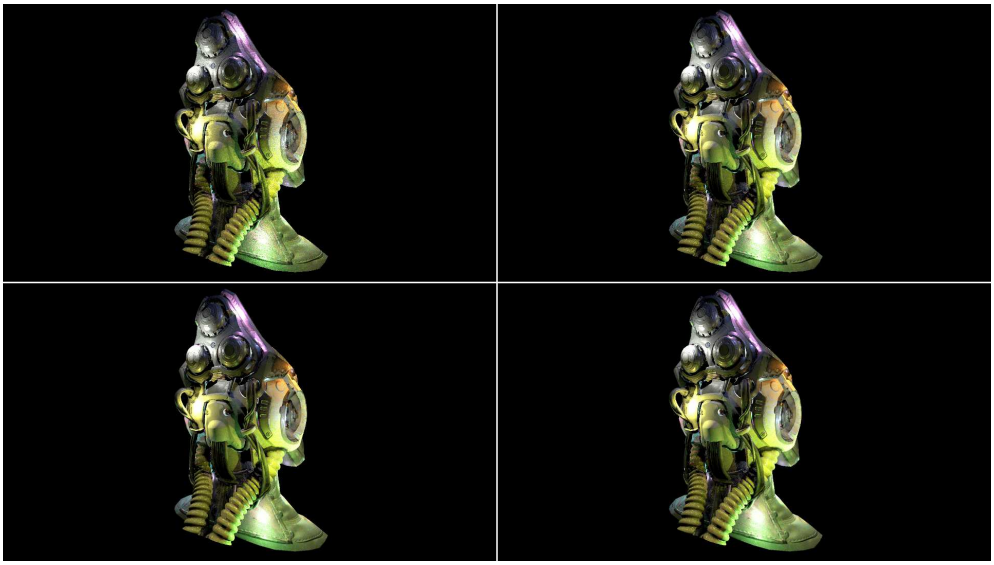

Figure 6.2: SciFi Helmet; 8, 32, 128 and 512 light samples

Test images can be seen in figures 6.1 and 6.2. The performance results can be found in the table 6.3 below.

	8 samples	32 samples	128 samples	512 samples
Sponza	4.6ms	9.8ms	20.4ms	64.4ms
SciFi Helmet	2.9ms	3.2ms	4.5ms	10.8ms

Table 6.3: Light samples test performance results

The number of light samples has a significant performance impact on the performance of the algorithm. However, going beyond 128 samples has diminishing effects on the noise and quality of the final image. With state-of-the-art denoising methods, it might be possible to get away with only using 32 light samples.

6.2. Visibility reuse test

Visibility reuse is used to discard samples that don't have a line of sight to the target pixel. The test is with visibility reuse on and off.

Figure 6.3: Sponza; visibility reuse turned on and off

Figure 6.4: SciFi Helmet; visibility reuse turned on and off

Test images can be seen in figures 6.3 and 6.4. The performance results can be found in the table 6.4 below.

	ON	OFF
Sponza	9.8ms	7.7ms
SciFi Helmet	3.2ms	2.1ms

Table 6.4: Visibility reuse test performance results

The main purpose of visibility reuse is to decrease the variance of the final image. Since visibility reuse involves tracing a shadow ray per pixel, it affects the performance

of the algorithm.

6.3. Temporal reuse test

Temporal reuse takes reservoirs of the pixel from the previous frame and combines them with the current one, adding information to the pixel being evaluated. The test is with temporal reuse on and off.

Figure 6.5: Sponza; temporal reuse turned on and off

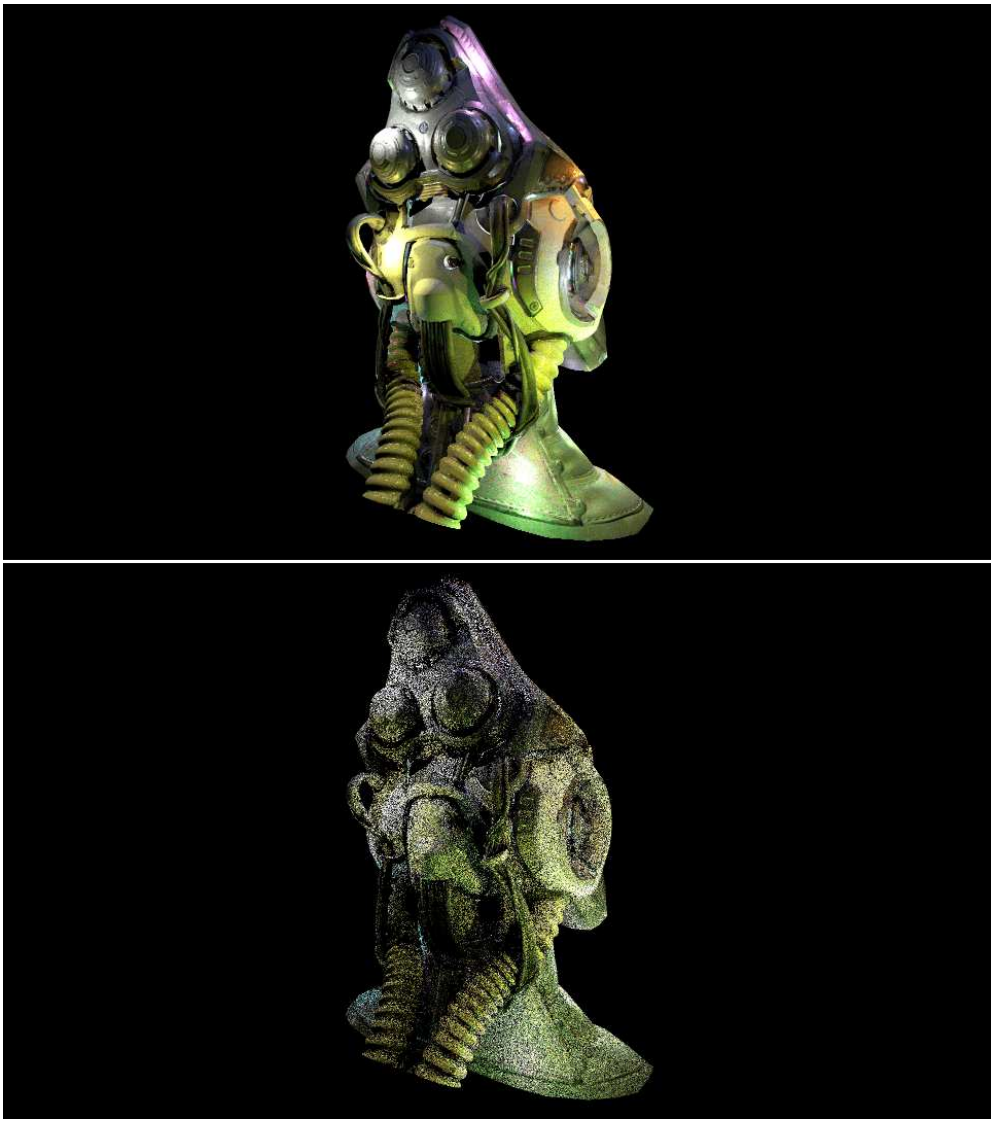

Figure 6.6: SciFi Helmet; temporal reuse turned on and off

Test images can be seen in figures 6.5 and 6.6. The performance results can be found in the table 6.5 below.

	ON	OFF
Sponza	9.8ms	6.7ms
SciFi Helmet	3.2ms	2.4ms

Table 6.5: Temporal reuse test performance results

While temporal reuse affects the performance of the algorithm, the quality of the image with the temporal reuse turned on is far superior to the one with it turned off.

6.4. Spatial reuse test

Spatial reuse takes reservoirs of the neighbouring pixels from the previous frame and combines them with the current one, adding information to the pixel being evaluated. The test is with spatial reuse on and off.

Figure 6.7: Sponza; spatial reuse turned on and off

Figure 6.8: SciFi Helmet; spatial reuse turned on and off

Test images can be seen in figures 6.7 and 6.8. The performance results can be found in the table 6.6 below.

	ON	OFF
Sponza	9.8ms	7.9ms
SciFi Helmet	3.2ms	2.8ms

Table 6.6: Spatial reuse test performance results

Turning spatial reuse on has a performance impact on the algorithm, however, the improvements to the image, while present, are not as significant as the case is with

temporal reuse.

6.5. Spatial neighbours test

Spatial neighbour count determines how many neighbours are evaluated on each iteration of spatial reuse. For this test, they range from 2 to 16.

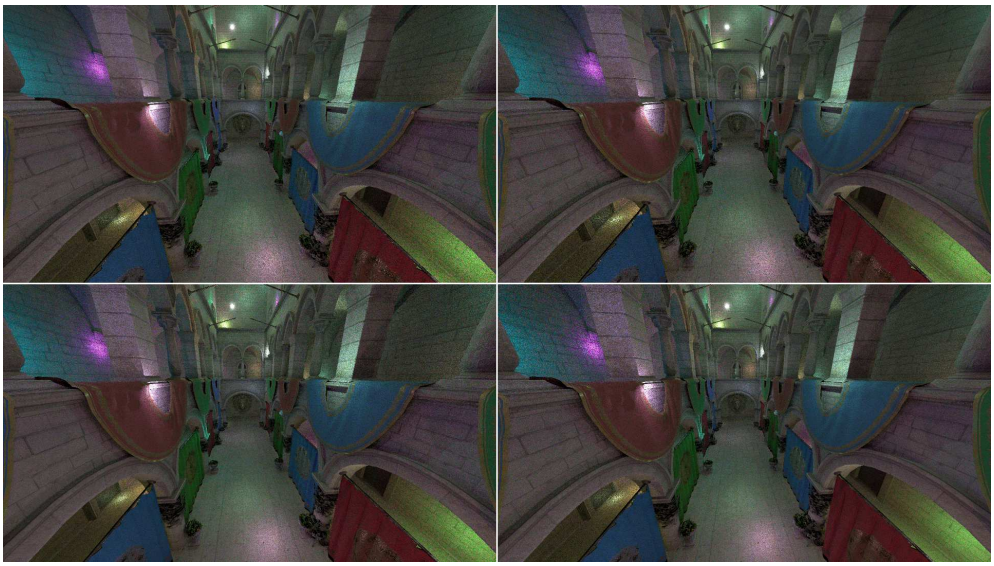

Figure 6.9: Sponza; 2, 4, 8 and 16 spatial neighbours

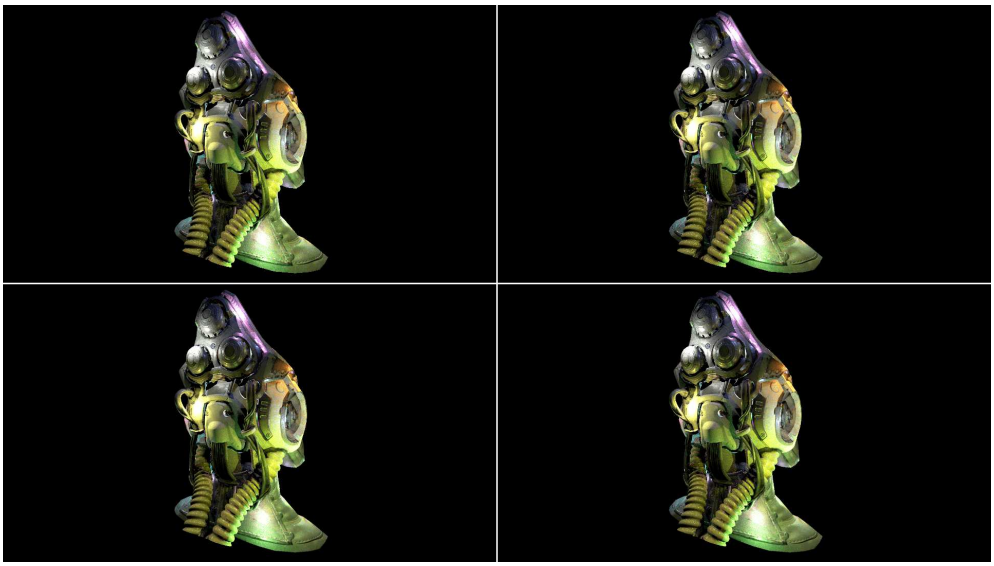

Figure 6.10: SciFi Helmet; 2, 4, 8 and 16 spatial neighbours

Test images can be seen in figures 6.9 and 6.10. The performance results can be found in the table 6.7 below.

	2 neighbour	4 neighbours	8 neighbours	16 neighbours
Sponza	9.8ms	9.8ms	9.9ms	9.9ms
SciFi Helmet	3.2ms	3.2ms	3.2ms	3.2ms

Table 6.7: Spatial neighbours test performance results

The number of spatial neighbours has hardly any performance impact. Using more neighbours improves the result, however, there are only so many neighbouring pixels, so going beyond a threshold might not yield any improvements in the image quality.

6.6. Spatial iterations test

Spatial iterations count determines how many iterations of spatial reuse are done for each frame. For this test, they range from 1 to 4.

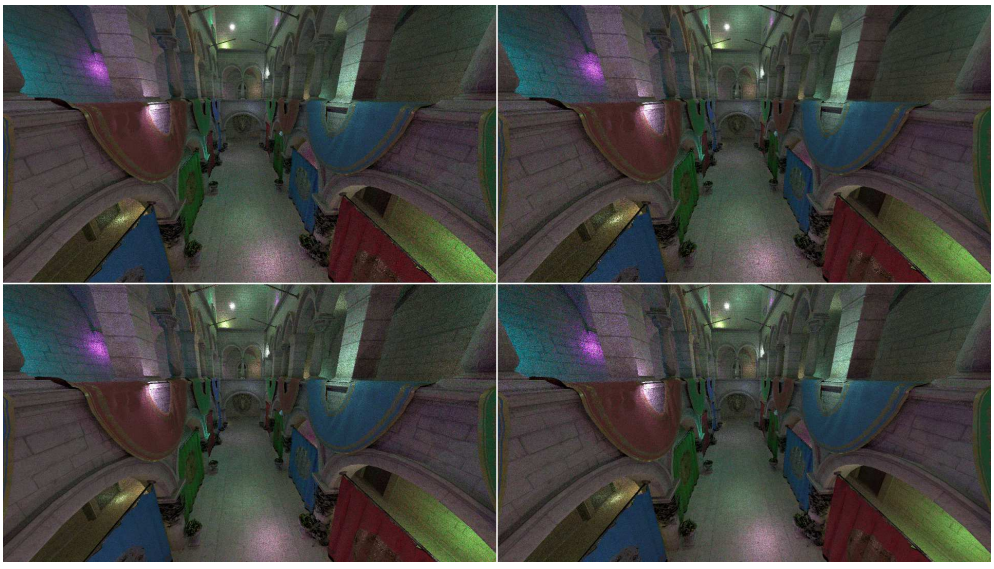

Figure 6.11: Sponza; 1, 2, 3 and 4 spatial iterations

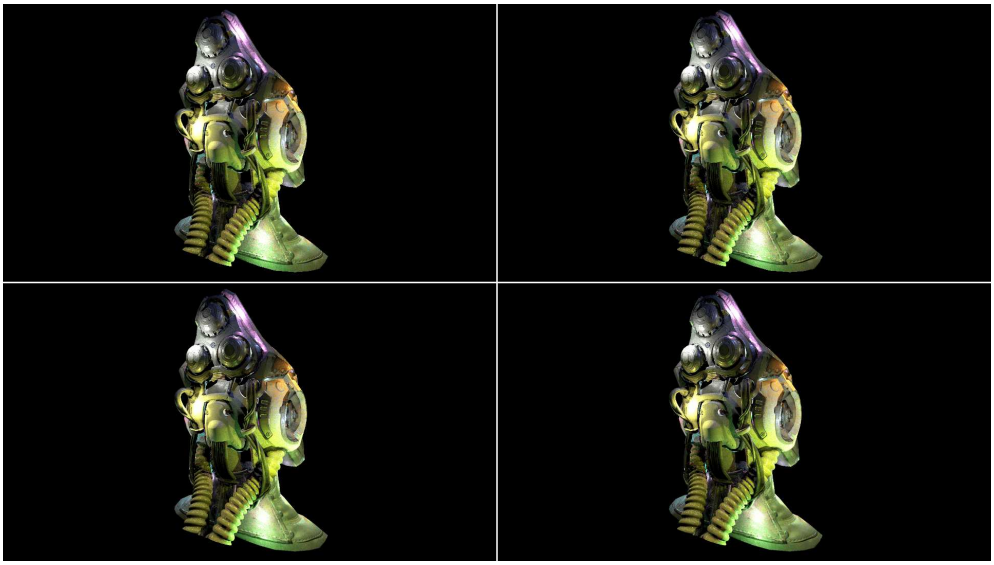

Figure 6.12: SciFi Helmet; 1, 2, 3 and 4 spatial iterations

Test images can be seen in figures 6.11 and 6.12. The performance results can be found in the table 6.8 below.

	1 iteration	2 iterations	3 iterations	4 iterations
Sponza	9.8ms	11.9ms	13.9ms	15.7ms
SciFi Helmet	3.2ms	4.1ms	5.1ms	6.1ms

Table 6.8: Spatial iterations test performance results

Increasing the number of spatial iterations improves the quality of the image. Similarly to the number of spatial neighbours, there are diminishing returns once all neighbouring pixels have been reused. The number of iterations should be chosen depending on the denoising method used. Higher quality denoiser could use the output of a single iteration, while some others might require 2 or even 3 iterations.

6.7. Increasing the number of lights

The main benefit of this algorithm is that it is not dependent on the number of lights in the scene. While that is theoretically correct, when increasing the number of lights in the Sponza model to 200000, the performance was significantly reduced. This can be tracked down to the alias table. Alias table is a simple buffer uploaded to the GPU whose size is linearly dependent on the number of light sources. Since light samples

are picked at random from the alias table, the increase in the size of the table led to GPU cache trashing, explaining the reduced performance.

7. Conclusion and further work

ReSTIR is an algorithm that gives significantly better results than the previous state-of-the-art method. It leverages data from the previous and current frames to better reconstruct the final image. Temporal reuse has shown to be more impactful on the final image than spatial reuse. It should be noted that the final image produces visible noise, however, this is to be expected when using stochastic algorithms, such as this one. The amount of noise is the main improvement from the previous state-of-the-art method. Modern denoisers should have a few issues with removing the noise from the final output of the algorithm.

This algorithm works only with direct lights, however, new works were published extending it to global illumination. The used implementation also has a critical flaw in regards to light sample selection, the used alias method causing GPU trashing. Since only several light samples are evaluated by pixel, a solution might lie in splitting the light sources into groups that are then presented to neighbouring pixels, reducing the issue of cache trashing.

This method, especially with the dawn of hardware-accelerated ray tracing, is a large step towards photo realism at interactive framerates. At this pace, it might even be achieved within the next decade.

BIBLIOGRAPHY

[1] Benedikt Bitterli, Chris Wyman, Matt Pharr, Peter Shirley, Aaron Lefohn, i Wojciech Jaros. Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. https://cs.dartmouth.edu/wjarosz/publications/bitterli20spatiotemporal.html, July 2020.

[2] Justin F. Talbot. Importance resampling for global illumination. http://diglib.eg.org/handle/10.2312/EGWR.EGSR05.139-146, 2005.

[3] Eric Veach i Leonidas J. Guibas. Optimally combining sampling techniques for monte carlo rendering. http://www.cs.jhu.edu/~misha/ReadingSeminar/Papers/Veach95.pdf, August 1995.

[4] Turner Whitted. An improved illumination model for shaded display. https://www.cs.drexel.edu/~david/Classes/Papers/p343-whitted.pdf, 1980.

Postupak praćenja puta i fizikalno temeljen prikaz površine

Sažetak

ReSTIR je algoritam koji omogućava iscrtavanje praćenjem zrake u realnom vremenu. Cilj ovog rada jest istražiti taj algoritma, napisati implementaciju te je testirati u odnosu na nekoliko parametara.

Ključne riječi: ReSTIR, praćenje zrake, iscrtavanje, računalna grafika

Path Tracing and Physically Based Rendering

Abstract

ReSTIR is an algorithm that enables path-traced rendering in real-time. The goal of this thesis is to explore this algorithm, write an implementation and test it in consideration of multiple parameters.

Keywords: ReSTIR, path tracing, rendering, computer graphics