Interaction Graphics and Design

Advanced 3D Computer Graphics Project: Gradient Estimation for Real-Time Adaptive Temporal Filtering

---

Student: Basci Onur

Contents

1 Introduction

Path tracing is a physically-based rendering technique that simulates the way light interacts with surfaces to produce highly realistic images. By tracing the paths of many light rays as they bounce around a scene, path tracing accurately models global illumination, soft shadows, caustics, and other complex lighting effects. This method is widely used in offline rendering for film and visual effects due to its ability to generate photorealistic results.

However, implementing path tracing in real-time rendering presents significant challenges. The primary difficulty arises from the immense computational power required to simulate the vast number of light interactions needed for high-quality images. Unlike traditional rasterization or even ray tracing techniques, which approximate lighting for efficiency, path tracing requires a large number of samples per pixel to reduce noise and achieve convergence. Real-time applications, such as video games, have strict performance constraints, often targeting 30 to 60 frames per second, making the brute-force approach of path tracing impractical.

To overcome these limitations, advancements in hardware (such as dedicated ray-tracing cores in modern GPUs), denoising techniques, and clever sampling methods have been developed. Despite these improvements, real-time path tracing remains an ongoing challenge, balancing visual fidelity and performance to make physically accurate rendering feasible for interactive applications. In this project, I focused on one of the denoising technique called adaptive spatiotemporal variance-guided filtering (A-SVGF) to implement a denoised real time path tracer.

2 Methods

Due to the high frame rate requirements (usually 30 or 60 FPS) in interactive applications, path tracing is often limited to one sample per pixel (SPP). This low sampling rate results in significant noise due to the stochastic nature of the sampling with a limited number of rays. Adjacent pixels can exhibit large variations in luminance depending on the randomly sampled ray directions. Most denoising algorithms rely on temporal and spatial filtering. In real-time applications, coherence between frames is high, allowing for noise reduction by leveraging information from previous frames. The previous frame is reprojected and blended with the current frame, and this process is repeated for subsequent frames using the filtered image. This method is known as exponential moving averaging (EMA).

$$\hat{c}_i(x) = \alpha c_i(x) + (1-\alpha)\hat{c}_{i-1}(\overleftarrow{x}) \tag{1}$$

In this formula, c represents the current frame, $\hat{c}$ the filtered image, x the pixel, $\overleftarrow{x}$ the back-projected pixel, and α the blending factor.

However, EMA alone does not eliminate all noise. In fact, by definition, the filtered image will still contain some noise since it is blended with $c_i(x)$, the path-traced image with only one sample per pixel. Depending on the value of α, the noise can be reduced by relying less on the current frame's color. However, this comes at the cost of increased ghosting.

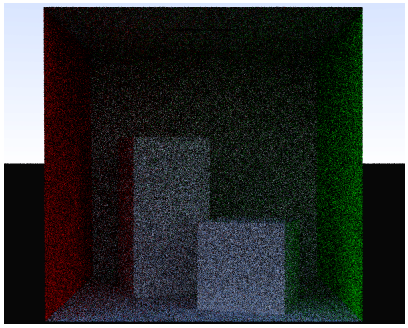

Figure 1: Path traced image with 1 spp

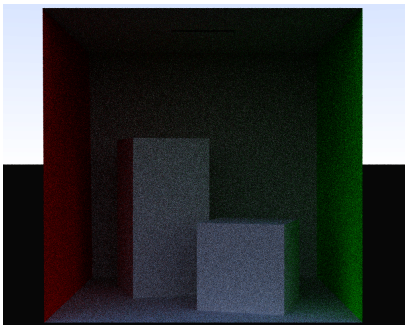

Figure 2: EMA $\alpha = 0.2$

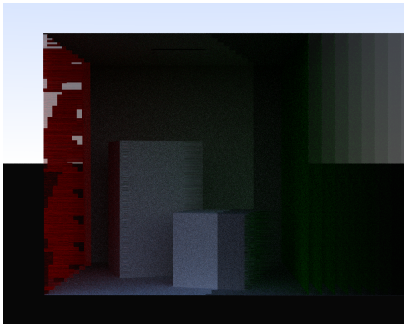

Figure 3: EMA $\alpha = 0.05$ introduces ghosting

To reduce the noise from the 1 SPP path-traced image, spatial filtering is also required. However, while filtering, it is crucial to preserve the structure of the image by maintaining edges. A simple convolution does not work in this case, as it would blur important details. The article proposes applying an edge-preserving À-Trous Wavelet Transform.

The term "À-Trous" (French for "with holes") refers to the way the filter operates by progressively increasing the spacing between sampled pixels at each iteration. This approach allows for an efficient, non-destructive decomposition of the image into multiple scales. It is particularly well-suited for global illumination filtering, as it adapts to the image structure, ensuring that important features such as edges and fine details are preserved while effectively reducing noise.

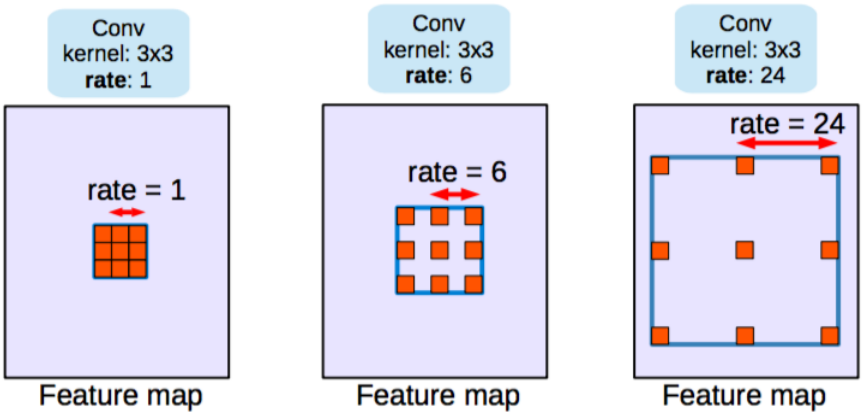

Figure 4: Illustration of three levels of the 2D à-trous wavelet transform.

One iteration of the À-Trous algorithm is defined as follows [2]:

$$\hat{l}^{(k+1)}(p) := \frac{\sum_{q \in \Omega} h^{(k)}(p,q) \cdot w^{(k)}(p,q) \cdot \hat{l}^{(k)}(q)}{\sum_{q \in \Omega} h^{(k)}(p,q) \cdot w^{(k)}(p,q)}$$

We essentially have a single kernel, h, with predefined coefficients, a weight function, w, that varies per pixel, and a function, l, representing color information. To ensure edge preservation during convolution, w also depends on depth and normal information. It is composed of three functions:

$$w^{(k)}(p,q) := w_z(p,q) \cdot w_n(p,q) \cdot w_l^{(k)}(p,q)$$

The weights for color and depth are derived from a Gaussian function based on their differences, while the weight for normals is computed using the dot product of surface normals. This ensures that pixels with significant normal variation are assigned lower weights, effectively preserving edges.

$$w_n(p,q) := \max(0, \langle n(p), n(q) \rangle)^{\sigma_n} \tag{2}$$

$$w_z(p,q) := \exp\left(-\frac{|z(p) - z(q)|}{\sigma_z \cdot |\langle \nabla z(p), p - q \rangle|}\right)$$

$$w_l^{(k)}(p,q) := \exp\left(-\frac{|\hat{i}^{(k)}(p) - \hat{i}^{(k)}(q)|}{\sigma_l \cdot \sqrt{g_{3\times3}(\mathrm{Var}(I^{(k)}(p))}}\right)$$

By applying (EMA) to the spatially filtered image, we obtain a denoised path-traced result.

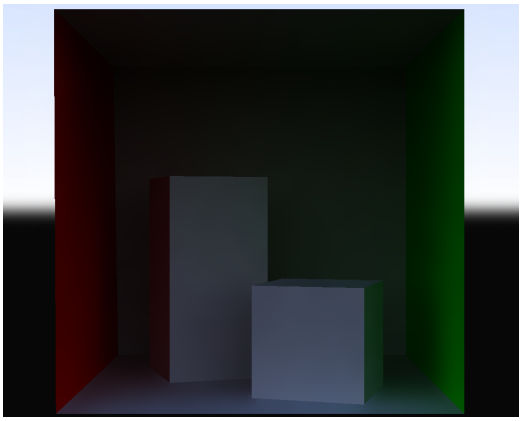

Figure 5: denoised result with alpha blending and spatial filtering

EMA applied to filtered images produces satisfactory real-time results in scenes without high variation. However, the reuse of previous frames leads to ghosting and lag in scenes with dramatic changes. To address this issue, we use A-SVGF [3]. This method proposes not to set α as a constant but as a variable that depends on the pixel's color variation. To compute α, the article suggests using the temporal gradient, defined as:

$$\delta_{i,\vec{j}} = f_i(\vec{G}_{i-1,j}) - f_{i-1}(G_{i-1,j}) \tag{3}$$

where f_i is the shading function of frame i, $G_{i,j}$ represents the sampling information at frame i for pixel j, and $\vec{G}_{i-1,j}$ is the forward-projected sample at frame $i-1$ for pixel j. The idea is to compute the difference in shading at the same point between frames. Using the temporal gradient, we compute α for each pixel as follows:

$$\alpha_i(p) = (1 - \lambda(p)) \cdot \alpha + \lambda(p)$$

where

$$\lambda(p) = \min\left(1, \frac{|\hat{\delta}_i(p)|}{\hat{\Delta}_i(p)}\right).$$

. Thus, the adaptive α varies between the constant α and 1, depending on the variation of the pixel.

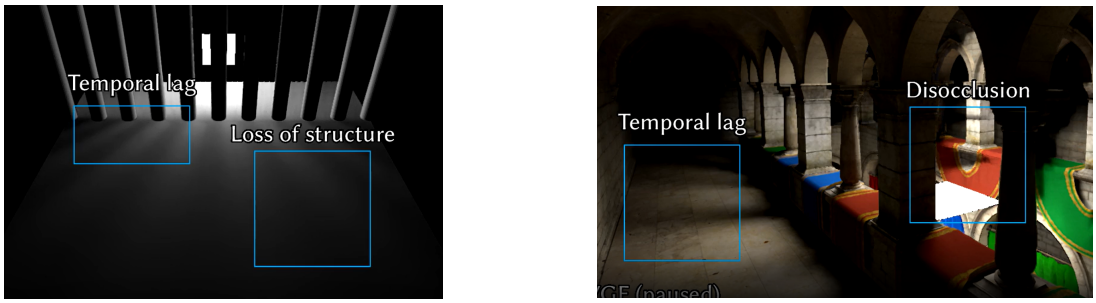

Figure 6: A constant α introduces temporal lag and ghosting in scenes with dramatic changes.

3 Implementation

3.1 Tools

3.1.1 Vulkan

Vulkan is a modern, low-overhead graphics and compute API designed to provide high-performance access to GPUs across a wide range of platforms. Unlike traditional graphics APIs, Vulkan offers greater control over GPU resources, explicit multi-threading, and lower driver overhead, making it an excellent choice for high-performance rendering tasks.

One of its key strengths is its support for extensions such as ray tracing and acceleration structures, which enable efficient real-time and offline path tracing. With features like Vulkan Ray Tracing (VK_KHR_ray_tracing_pipeline) and acceleration structures (VK_KHR_acceleration_structure) developers can leverage hardware-accelerated ray traversal and bounding volume hierarchies (BVH).

Due to the complexity of the path tracing algorithm, I chose to work with Vulkan to achieve better performance without having to manually implement acceleration structures like BVH.

3.1.2 Nvvk

Although Vulkan offers many advantages, I found it more challenging to work with compared to higher-level APIs like OpenGL. This is primarily because Vulkan provides significantly more control over GPU and CPU communication, requiring developers to be very explicit about their intentions. For example, rendering a simple triangle using rasterization can take around 1,000 lines of code! Fortunately, there are helper libraries available that reduce boilerplate code and speed up development. For this project, I used nvvk, a collection of Vulkan utility functions and helper classes developed by NVIDIA to simplify Vulkan development. While retaining all the benefits of the Vulkan API, nvvk offers helper classes for memory management, shader loading, descriptor management, and much more.

3.2 Pipeline

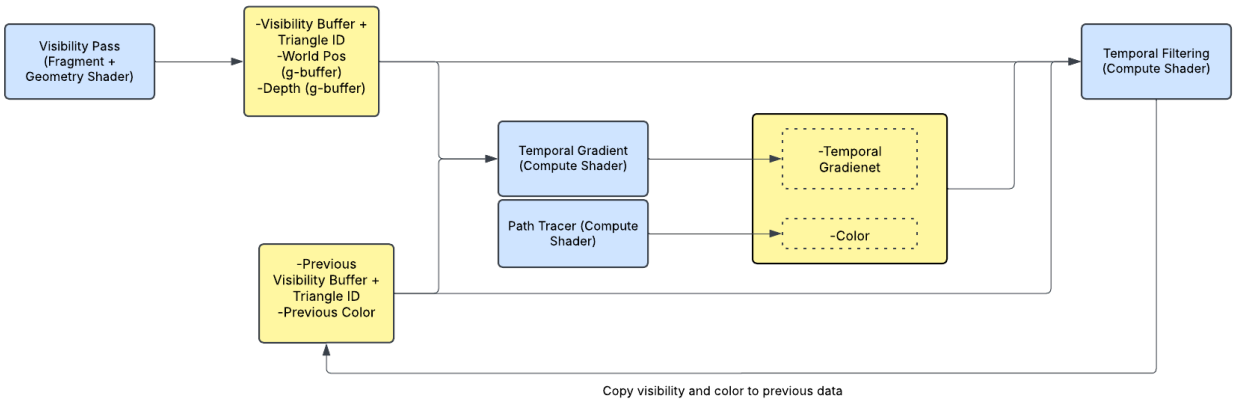

Figure 7: An overview of my implementation: The blue boxes represent shaders, while the yellow boxes represent the resources either used by or generated for the shaders, depending on the direction of the arrows

In Figure 7, you can see an overall pipeline of my implementation. It is basically composed of a visibility pass, a 1 spp path-traced scene computation, temporal gradient calculation, and finally, temporal filtering. At the end of the temporal filtering, the current scene information is copied as the previous frame to be used for the next frame's computation.

3.2.1 Visibility Pass

Instead of using Equation 4, I used a slightly modified version that I found more intuitive in the implementation, defined as:

$$\delta_{i,\vec{j}} = f_i(G_{i,j}) - f_{i-1}(\overleftarrow{G}_{i,j}) \tag{4}$$

The article states that both of these equations compute the temporal gradient, but the second one requires us to keep all the states necessary for the shading function of the previous frame. While this is true, since I don't use a complicated scene (see Figure 5), the only information I needed to keep was the point light's color and position.

As you can see, the computation of the temporal gradient requires a backprojection of the sampling. The article does this by using a visibility buffer. However, it was not clearly indicated which information they kept in the visibility buffer.[1] Therefore, I decided to store the triangle index in a G-buffer and the triangle vertices for each index in a separate buffer that I call the visibility lookup table (LUT). This approach is more memory-friendly since we don't need to store the triangle vertices information for each pixel. Instead, we can simply use the triangle index for a pixel to retrieve the corresponding vertices from the visibility LUT. I also don't store the normal information, as it can be deduced from the normalized cross product of the triangle vertices.

I fill the visibility LUT in the geometry pass using gl_primitiveID, and I fill the visibility buffer in the fragment shader. Additionally, I keep the world position and depth information in a G-buffer, which is required for the temporal filtering. (see equation 2).

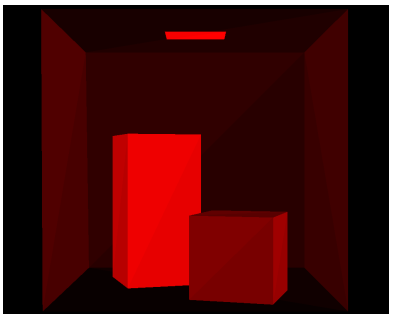

Figure 8: Visibility buffer; each tone of red represent a triangle ID

3.2.2 Path Tracer

I render the scene with 1 spp using a compute shader with a path tracing algorithm. In this part, I benefited from Vulkan's ray tracing extensions. The code is also inspired by NVIDIA's Vulkan mini path tracer tutorial. The compute shader generates a very noisy path-traced image. (See figure 1)

3.2.3 Temporal Gradient

To back-project samples, I use barycentric coordinates. In the compute shader, we have access to the current world space coordinates and the visibility LUT from both the current and previous frames. We can compute the barycentric coordinates from the triangle vertices and the area of the triangles (see Figure 9). Since the sample points' barycentric coordinates do not change, we can calculate the world position of the sample in the previous frame using the equation

$$p = \alpha v_1 + \beta v_2 + \gamma v_3$$

where α, β, and γ are the barycentric coordinates, and v_1, v_2, and v_3 are the triangle vertices' coordinates from the previous frame.

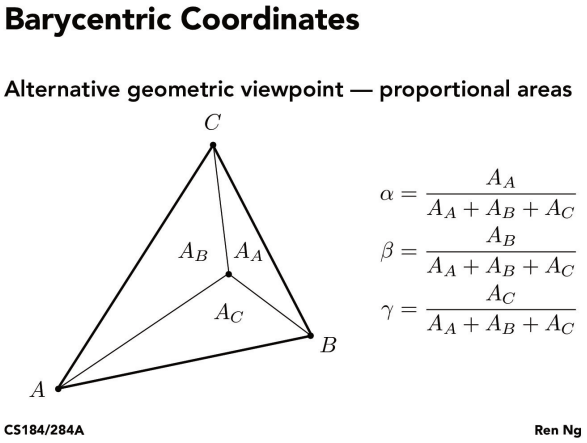

Figure 9: Illustration of barycenteric coordinate calculation from triangle areas

With the current world position and the previous frame's world position, the only thing left is to calculate the shading function defined by f in Equation 4. The shading function is not specified in the article, so I decided to implement a simple Blinn-Phong shading model.

Figure 10: Temporal Gradient obtained from a scene with dramatic light changes

3.2.4 Temporal Filtering

In the last pass, I apply the à trous wavelet transform to the path-traced image. I use a 3×3 box filter for the h kernel (see Equation 2). I also use the temporal gradient information to apply adaptive temporal filtering with EMA. Finally, we render the output to the window and copy the current visibility buffer and image to the previous frame.

4 Results

In this project, I worked with a scene containing two boxes in a semi-closed room. To make the scene more challenging, I also added a movable sphere light source. This light source produces very noisy results, as many rays do not intersect it and thus yield dark results, while those that do hit it result in very bright outputs (see Figure 11). I also created a reference image, which is computed offline with 8192 rays per pixel (see Figure 12). The final result is obtained with $\alpha = 0.2$ and 9 iterations of the à trous wavelet transform (see Figure 13).

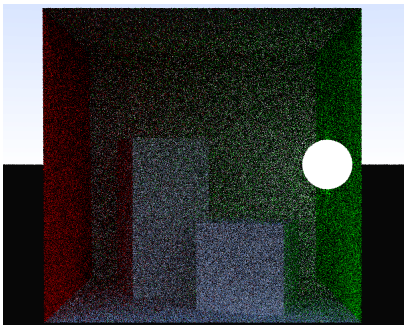

Figure 11: Path traced image with 1 spp

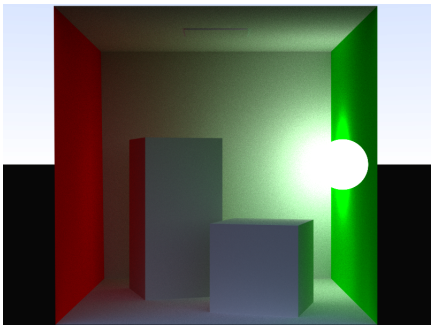

Figure 12: reference image with 8192 rays per pixel

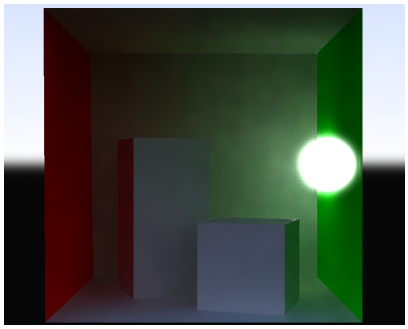

Figure 13: Scene with A-SVGF filtering

We can see that the A-SVGF algorithm clearly denoises a large amount of noise, although a small amount of noise is still present. The structure of the scene is well preserved, but we notice some artifacts, especially from the SVGF. The most noticeable one is the loss of structure in the shadows, where the edges of the shadows are smoothed. We can also observe a pattern appearing near the light source, indicating that the filter could not denoise the high-frequency noise coming from the light source.

This can be addressed by increasing the number of iterations for the wavelet transform; however, increasing it too much can break the structure of the objects.(see figure 14)

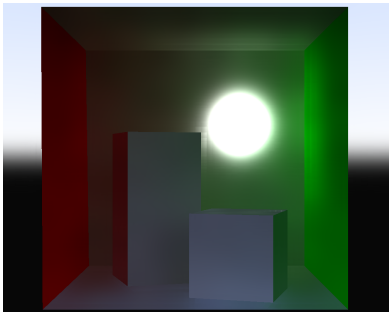

Figure 14: Wavelet transform with iteration = 15

I also compared the results using adaptive and constant alpha blending values. To do this, I created a scene where the light source quickly moves to the right. We can see that in the constant alpha case, the room is still lit even though the light source is no longer in the room. In contrast, in the adaptive alpha case, the light suddenly becomes dark when the light is no longer in the room.

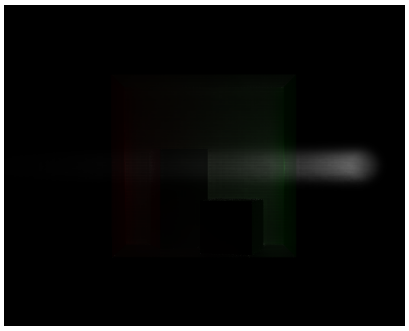

Figure 15: constant alpha per pixel

Figure 16: adaptive alpha per pixel

5 Conclusion and Future Work

We observed that A-SVGF gives satisfactory results with small artifacts in simple scenes. I would like to test these results in more complicated scenes with different materials. I am curious to see how this algorithm would perform with textured materials. My guess is that it would blur the texture.

Another thing I would like to try is machine learning-based denoising for ray tracing. We noticed that A-SVGF does not completely denoise the image. I would like to compare the results and also the performance. Since machine learning-based methods work directly on the image, I suspect they might not require additional steps like calculating the temporal gradient, and thus may perform faster.

6 Bibliography

References

[1] Christopher A Burns and Warren A Hunt. The visibility buffer: a cache-friendly approach to deferred shading. *Journal of Computer Graphics Techniques (JCGT)*, 2(2):55–69, 2013.

[2] Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*, pages 1–12. 2017.

[3] Christoph Schied, Christoph Peters, and Carsten Dachsbacher. Gradient estimation for real-time adaptive temporal filtering. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 1(2):1–16, 2018.