Eurographics Symposium on Rendering (2025)
B. Wang and A. Wilkie (Editors)

Spatio-Temporal Dithering for Order-Independent Transparency on Ray Tracing Hardware

Felix Brüll René Kern Thorsten Grosch

TU Clausthal, Germany

(a) *Russian Roulette (2.42 ms)* **(b)** *Ours (2.51 ms)* **(c)** *Reference (3.35 ms)*

Figure 1: *Animated boss fight scene rendered using russian roulette for transparency vs our dithered approach vs a ray-traced reference. The output is temporally accumulated with DLSS.*

Abstract

Efficient rendering of many transparent surfaces is a challenging problem in real-time ray tracing. We introduce an alternative approach to conventional order-independent transparency (OIT) techniques: our method interprets the alpha channel as coverage and uses state-of-the-art temporal anti-aliasing techniques to accumulate transparency over multiple frames. By efficiently utilizing ray tracing hardware and its early ray termination capabilities, our method reduces computational costs compared to conventional OIT methods. Furthermore, our approach shades only one fragment per pixel, significantly lowering the shading workload and improving frame rate stability. Despite relying on temporal accumulation, our technique performs well in dynamic scenes.

CCS Concepts

• ***Computing methodologies*** → ***Ray tracing;*** *Rasterization; Visibility; Antialiasing;*

1. Introduction

Real-time rendering of transparency is a well-researched topic in computer graphics, and various proposals for rasterization hardware have been developed over the last decades. In most real-time applications, transparent materials occur only in small quantities. However, in certain scenarios, the occurrence of transparent fragments can drastically increase: for example, during explosions or massive battles in games (see Fig. 1). In such cases, the sudden increase in transparent particles can cause significant performance drops. This is primarily due to two factors: First, rendering exact transparency requires sorting transparent fragments before compositing, which introduces additional overhead. Second, shading multiple transparent fragments per pixel drastically increases the computational workload, especially when some effects, such as smoke or fog, require tracing shadow rays for each fragment.

Before the availability of programmable shaders, transparent geometry had to be pre-sorted on the CPU to ensure it was drawn in back-to-front order. Programmable hardware enabled the implementation of **order-independent transparency (OIT)** techniques, eliminating the need for pre-sorting transparent geometry.

Exact methods may utilize linked lists [YHGT10] or arrays [MCTB12] to record transparent fragments on the fly and

© 2025 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.
This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

DOI: 10.2312/sr.20251177

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

then sort them before composing the final image. Several faster, but approximate methods have been introduced, including various k-buffer and other bounded memory approaches [BCL*07; VVPM17; SML11; SV14; MKKP18], weighted blended OIT [MB13], and stochastic transparency [ESSL10]. The latter is based on the idea of interpreting the alpha channel as a coverage mask and leveraging the hardware's **multi-sample anti-aliasing (MSAA)** capabilities to approximate transparency.

With the rise of deferred rendering and the introduction of real-time ray tracing, hardware-accelerated MSAA has become less relevant. It is largely being replaced by **temporal anti-aliasing (TAA)** techniques [YLS20] or its machine learning-based counterparts [NVI25; AMD24; Int25], as they are computationally less expensive and allow for anti-aliasing over time, which reduces flickering between frames.

In this paper, we propose a novel approach to transparent rendering that combines the principles of stochastic transparency with temporal anti-aliasing techniques. Our method not only provides OIT but also addresses the shading workload by limiting the shading budget to one fragment per pixel, even in the presence of multiple transparent layers. This results in a unified GBuffer/VBuffer that stores both transparent and opaque fragments in a single buffer, which is beneficial for post-processing effects that are limited by the lack of transparency information (e.g., ambient occlusion or depth of field). Furthermore, our approach enables limiting ray-traced shadow rays to one ray per pixel.

The idea of combining TAA with stochastic transparency was initially proposed by Salvi [Sal16], but the results at the time were not satisfactory. We demonstrate that, with a more optimized dithering pattern and the use of machine learning-based anti-aliasing techniques, we can achieve a more stable and visually appealing result, even in dynamic scenes. Furthermore, by utilizing the early ray termination capabilities of ray tracing hardware, we can reduce the computational cost of our method compared to conventional rasterization-based OIT techniques.

Our contributions are:

- A reformulation of stochastic transparency as a temporal anti-aliasing problem, that only shades a single fragment per pixel, specifically optimized for ray tracing (Sec. 3.2).
- Alpha-to-coverage adjustments for DLSS (Sec. 3.2.1).
- A novel spatio-temporal dithering pattern (Sec. 3.3).
- A stochastic motion vector rectification technique required for screen-space dither patterns (Sec. 3.3.1).

2. Related Work

We define a transparent fragment by its color c and opacity α. Two consecutive fragments can be blended using the OVER operator as follows [PD84]:

$$(c_0, \alpha_0)\ \text{OVER}\ (c_1, \alpha_1) := \alpha_0 c_0 + (1-\alpha_0)\alpha_1 c_1 \tag{1}$$

where (c_0, α_0) must be in front of (c_1, α_1). The transmittance of the merged fragment is given by $(1-\alpha_0)(1-\alpha_1)$. This formula dictates the need for sorted composition in transparency rendering.

OIT techniques eliminate the need for sorting geometry on the CPU and instead use the GPU to composite transparent fragments. For a comprehensive overview of OIT techniques, we refer to the surveys by Wyman [Wym16] and Vasilakis et al. [VVP20].

A few techniques that combine ray tracing with OIT have been proposed, primarily implementing k-buffer approaches and leveraging the new flexibility of tracing a continuation ray when necessary [BG20; MMP*24].

2.1. Temporal Anti-Aliasing

We follow the TAA implementation by Salvi [Sal16]. The core idea is to combine the aliased color from the current frame C_n with the anti-aliased color from the previous frame F_{n-1}:

$$F_n = w\, C_n + (1-w)\, F_{n-1} \quad \text{with} \quad w = 0.1 \tag{2}$$

This exponentially weighted average ensures responsiveness to changes while maintaining temporal stability.

Salvi employs motion vectors to properly retrieve the color from the previous frame. However, to handle disocclusions and drastic illumination changes, the previous frame's color must be rectified to prevent major ghosting artifacts. Salvi's method achieves color rectification by inspecting a 3×3 neighborhood around the current pixel, computing its average color and variance. The previous frame's color is then clipped to the resulting ellipse before being used in the exponential averaging process.

DLSS [NVI25] avoids handcrafted color rectification methods by leveraging a machine learning model to predict the correct color. This typically results in sharper images while mitigating ghosting artifacts. Like TAA, it takes the aliased color and motion vectors as input, but unlike TAA, it also requires the depth buffer. Unfortunately, NVIDIA has not disclosed the detailed workings of DLSS. FSR and XeSS are similar techniques developed by AMD and Intel, respectively [AMD24; Int25].

2.2. Dithering

Ordered dithering [Bay73] is a technique that can be used to create the illusion of grayscale images by only using black and white pixels. An ordered dithering matrix serves as a screen-space threshold map to determine whether a pixel should be rendered as black or white. Below is an example of a 2×2 and a 4×4 dithering matrix:

$$D_2 = \frac{1}{4}\begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix}, \quad D_4 = \frac{1}{16}\begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix} \tag{3}$$

To emulate a grayscale image with 30% intensity, all pixels corresponding to a dither matrix value below 0.3 are rendered as white, while the rest are rendered as black.

This technique can also be applied to transparency rendering, where it is known as screen-door transparency. Transparent fragments are interpreted as either opaque or invisible based on the alpha test against the corresponding dither matrix value. To allow rendering of multiple transparent objects, each object requires a different screen-space dither matrix. Otherwise, objects with the same

© 2025 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

Listing 1: *Dither Temporal AA threshold computation:*

```
1 uint2 p = pixel + uint2(frameIndex);
2 uint stratify = (p.x + p.y * 2) % 5;//[0,4]
3 float blue=BlueNoise64x64Tex[pixel % 64];//[0,1]
4 float threshold = (stratify + blue) / 5.0;
```

transparency will mask each other out. The generation of suitable matrices, with visually pleasant spatial properties, is discussed by Mulder et al. [MGvW98].

Stochastic transparency implements screen-door transparency on a sub-pixel level by using MSAA hardware [ESSL10]. In their implementation, the coverage mask emulates the visibility of a transparent fragment, similar to hardware-accelerated alpha-to-coverage, but they extend it with randomized coverage masks for each fragment:

Unlike screen-door transparency, the spatial distribution of coverage bits is not important, since composition occurs at the sub-pixel level. Nevertheless, the coverage mask must exhibit good **stratification** properties. For example, with $8\times$ MSAA and $\alpha = 60\%$, the mask should contain either 4 or 5 active bits, representing $4/8 = 50\%$ or $5/8 = 62.5\%$ coverage, respectively.

Because this method inherently produces a discretized transparency appearance, the authors propose an additional **Alpha Correction**. They accumulate the transparency of all fragments within a pixel to compute the correct occlusion factor for the background. While this improves visual quality, it requires an additional pass over the transparent geometry, which is not feasible in our case, as we aim to terminate rays early.

Error diffusion [FS75], another widely used dithering technique, is preferred for image dithering as it distributes the quantization error more effectively. However, it is unsuitable for animated content, as a small change in a single pixel can affect the entire image.

One application of temporal dithering is extending the displayed color range of monitors from 6-bit to 8-bit or 8-bit to 10-bit [DF03]. This is achieved by alternating between two colors to create intermediate shades.

Dither Temporal AA (DTAA) is a material shader in Unreal Engine 4 that implements spatio-temporal dithering for transparency [Epi25]. It uses a 5×5 screen-space threshold map in combination with a 64×64 blue noise mask. The 5×5 map is shifted vertically by one pixel each frame to vary the thresholds over time, while the blue noise mask remains static and helps to distribute the quantization error spatially. The exact computation is shown in Listing 1, and the 5×5 threshold map for the first frame is illustrated below:

$$\frac{1}{5}\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 0 & 1 \\ 4 & 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 4 & 0 \\ 3 & 4 & 0 & 1 & 2 \end{bmatrix} \tag{4}$$

This implementation has a major issue: It uses the same threshold map for all objects, causing objects with similar transparency to mask each other out. This can be resolved by using a random permutation of the numbers 0 to 4 for each object.

Wolfe et al. describe a method for generating spatiotemporal blue noise [WMAR22]. They extend classical 2D blue noise algorithms by incorporating a temporal component. Their approach demonstrates improved spatial and temporal characteristics when compared to 3D blue noise.

3. Our Method

We begin by describing our baseline renderer, which renders all transparent fragments in front-to-back order using ray tracing (Sec. 3.1). This renderer is comparatively slow, as it traces multiple rays and shades all fragments along the ray's path.

In Sec. 3.2, we introduce a minimal stochastic transparency method that traces only a single ray and shades a single fragment using classical Russian roulette.

In Sec. 3.3, we present our novel spatio-temporal dithering technique, that works on top of the Russian roulette method.

In Sec. 3.4, we describe a hybrid approach that seamlessly combines the baseline renderer with the stochastic methods, enabling fine-grained control between performance and quality.

In Sec. 3.5, we explain how to implement our method on rasterization hardware.

Sec. 4 presents an extensive comparison of the different methods.

3.1. Baseline Renderer

Our baseline renderer traces one primary ray per pixel and evaluates shading in the closest-hit shader (Listing 2). The accumulated color and remaining visibility are tracked in the ray payload. If the remaining visibility is above a small threshold (0.001) after returning from the closest-hit shader, a continuation ray is traced. The process is repeated until the accumulated visibility falls below the threshold.

The closest-hit shader evaluates the material's BRDF and traces a *deep shadow ray* to connect with the light source. The deep shadow ray accounts for transparency by accumulating transmittance in the any-hit shader, resulting in scalar shadow values.

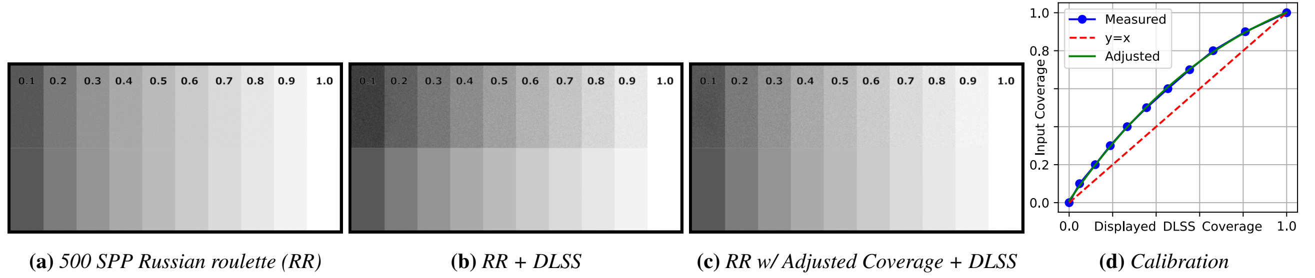
As a post-processing anti-aliasing solution, we apply TAA [Sal16], DLSS [NVI25], or FSR [AMD24]. To support these techniques, the baseline renderer outputs motion vectors and depth values from the last shaded fragments, typically corresponding to opaque surfaces.

3.2. Russian Roulette

Our efficient Russian roulette transparency implementation traces a single ray per pixel from the ray generation shader (Listing 3). The Russian roulette decision is handled in the any-hit shader, which evaluates the intersection opacity α and compares it against a random threshold t. If $\alpha < t$, the intersection is ignored.

The random threshold t is deterministically derived from a hash

© 2025 The Author(s).



(a) *500 SPP Russian roulette (RR)* **(b)** *RR + DLSS* **(c)** *RR w/ Adjusted Coverage + DLSS* **(d)** *Calibration*

Figure 2: *Russian roulette transparency on a semi-transparent plane with 10 opacity levels. The top half of the plane is a white material with varying opacity levels, while the bottom half is an opaque material with varying shades of gray for calibration purposes. The background is black.* ***(a)*** *500 samples per pixel of Russian roulette show that the method converges to the correct result.* ***(b)*** *Applying DLSS to Russian roulette transparency yields results that are darker than the reference.* ***(c)*** *Based on the measurements in* ***(b)****, we derived a calibration function that adjusts the material opacity to produce correct results with DLSS.* ***(d)*** *The calibration graph plots the measured opacity values from* ***(b)*** *and shows how our polynomial fit approximates the measurements.*

Listing 2: *Shader for baseline renderer:*

```
#define MIN_VISIBILITY 0.001
#define MIN_ALPHA 0.01
void rayGen() {
 RayPayload p;
 p.color = float4(0, 0, 0, 1);
 RayDesc ray = GetCameraRay();
 while(p.color.a > MIN_VISIBILITY) {
  TraceRay(ray, p);
  ray.TMin = p.nextT;
 }
}
void anyHit(inout RayPayload p) {
 if(getMaterialAlpha() < MIN_ALPHA)
  IgnoreHit();
}
void closestHit(inout RayPayload p) {
 float alpha = getMaterialAlpha();
 float3 c = shadeMaterial(); // +shadow ray
 p.color.rgb += p.color.a * alpha * c;
 p.color.a *= (1 - alpha);
 p.nextT = RayTCurrent();
 if(p.color.a <= MIN_VISIBILITY)
  WriteColorAndDepthAndMotionVector(p);
}
```

of the intersection's 3D position and the frame index, using the 4D hash function by McGuire [McG16] (see Appendix).

The closest-hit shader only shades the fragment that survives Russian roulette, treating it as fully opaque. Consequently, only one fragment per pixel is shaded, significantly improving performance over the baseline renderer. Motion vectors and depth values for post-processing are derived from this fragment.

To better accumulate the stochastic noise, we adjust TAA following Salvi [Sal16], using a 7×7 neighborhood for color rectification when transparent fragments are present.

Listing 3: *Shader for Russian roulette renderer:*

```
void rayGen() {
 RayDesc ray = GetCameraRay();
 TraceRay(ray);
}
void anyHit(inout RayPayload p) {
 float alpha = getMaterialAlpha();
 float t = hash4D(getPos3D(), frameIndex);
 if (alpha < t) IgnoreHit();
}
void closestHit(inout RayPayload p) {
 float3 c = shadeMaterial(); // +shadow ray
 WriteColorAndDepthAndMotionVector(c);
}
```

3.2.1. Alpha-to-coverage for DLSS

We verified the Russian roulette method by comparing the accumulated result against the baseline renderer. The method produces correct results for simple sample accumulation and TAA. However, when used together with DLSS or FSR, we observed more aggressive accumulation, resulting in overly transparent outcomes.

DLSS often makes fast-moving transparent effects, like raindrops, appear too faint. We believe this happens because DLSS tries to suppress flickering, which may cause it to mistakenly tone down these effects. Since our method can produce color variations that resemble flicker, DLSS reduces their intensity, resulting in unintended transparency. In the case of FSR, we attribute the overly transparent appearance to its luminance instability module and other flicker suppression mechanisms like color clamping.

To address this, we conducted an experiment with a transparent plane featuring 10 opacity levels, as shown in Fig. 2. Based on this setup, we derived a correction function that converts material opacity α into DLSS-adjusted coverage values α_{DLSS}:

$$\alpha_{DLSS} = 0.0113 + 1.656\alpha - 0.821\alpha^2 + 0.156\alpha^3 \quad (5)$$

Although the correction function works well in the experimental setup, as shown in Fig. 2c, it produces slight deviations in other

© 2025 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

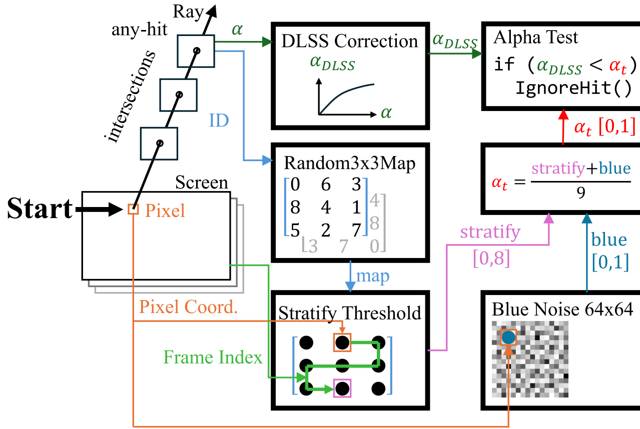


Figure 3: *Overview of our STD method for an any-hit intersection: During an intersection, the alpha value and object ID are used to evaluate a modified alpha test against a dithered threshold α_t.*

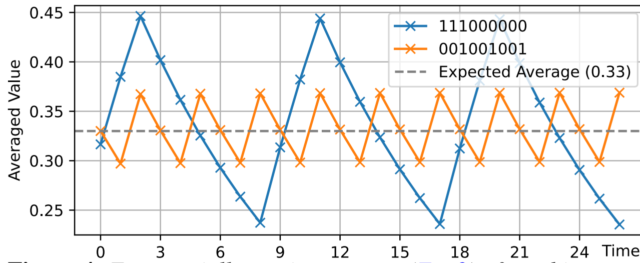


Figure 4: *Exponentially moving average (Eq. 2) of two binary sequences after a few iterations. The first binary sequence exhibits significantly higher variance in the averaged value, as all the ones are clumped together. Higher variance results in more noticeable flicker and, in the context of TAA, causes more aggressive color clamping.*

scenarios. Nevertheless, the approximation is sufficiently accurate for most cases. As FSR exhibits similar behavior to DLSS in this regard, we apply the same correction function to both methods.

We also tested a simple gamma correction, but it did not fit the measurements as well as the polynomial function.

3.3. Our Spatio-Temporal Dithering (STD)

The initial results of the Russian roulette method exhibit significant noise, as shown in Fig. 1. To mitigate this, we introduce a spatio-temporal dithering pattern that distributes thresholds more evenly across space and time, as depicted in Fig. 3.

We propose a novel spatio-temporal dithering approach, conceptually similar to DTAA [Epi25], but designed to produce a much smoother surface appearance. Our method employs a 3×3 screen-space threshold map with nine unique thresholds, which are shifted in a serpentine pattern each frame. Increasing the number of thresholds from five to nine results in a more polished appearance. Additionally, we overlay a 64×64 blue noise mask, similar to DTAA, but reduce its contribution to $1/9$.

Threshold Map Generation: To prevent objects from masking each other out, we precompute multiple 3×3 threshold maps.

Listing 4: *Threshold computation for our spatio-temporal dither:*

```
1  float threshold(uint2 pixel, uint frameIndex) {
2   uint2 p = shift(pixel, frameIndex) % 3u;
3   uint3x3 map = getRandom3x3Map();
4   uint stratify = map[p];//[0,8]
5   float blue=BlueNoise64x64Tex[pixel%64];//[0,1]
6   return (stratify + blue) / 9.0;
7  }
8  // serpentine raster shift
9  uint2 shift(uint2 pixel, uint frameIndex) {
10  uint xoff = frameIndex % 3;
11  uint yoff = (frameIndex / 3);
12  if (yoff % 2) xoff = 2 - xoff;
13  return pixel + uint2(xoff, yoff);
14 }
15 // select 1 out of 288 precomputed maps
16 uint3x3 getRandom3x3Map() {
17  float rng =
18  hash3D(InstanceID(),GeometryIndex(),HitKind());
19  return get3x3Map(uint(rng * 288)); //buffer
20 }
```

These maps require good spatial and temporal properties. Temporal properties are particularly important, as TAA accumulates color values using an exponentially moving average (Eq. 2). For example, consider the threshold sequences $[0,1,2,3,4,5,6,7,8]$ and $[0,3,6,1,4,7,2,5,8]$. After applying an alpha test with $t < 3$, we obtain $[1,1,1,0,0,0,0,0,0]$ and $[1,0,0,1,0,0,1,0,0]$, respectively. Although both sequences have the same average value of 0.33, the first sequence exhibits worse temporal properties due to the clumping of similar thresholds, as illustrated in Fig. 4. Therefore, sequences with well-distributed thresholds should be preferred.

For this, we follow the principles of ordered dithering, ensuring that neighboring thresholds differ as much as possible. Specifically, we generate all $9! = 362880$ permutations of the sequence $\{0,1,\dots,8\}$ and filter out configurations C where any direct matrix neighbor differs by only one, considering wraparound edges. This filtering process reduces the set to 1512 configurations, which are then ranked based on the sum of squared differences between neighboring thresholds:

$$\text{score} = \sum_{(i,j)} (C(i,j) - C(i+1,j))^2 + (C(i,j) - C(i,j+1))^2 \quad (6)$$

We select 288 of the highest scoring configurations, as they provide visually pleasing threshold distributions. Below are a few examples of the generated matrices:

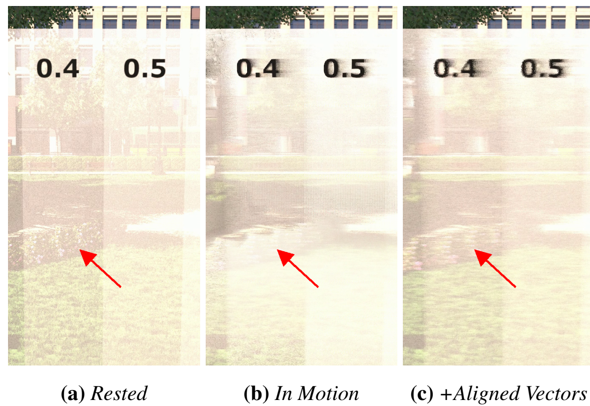
$$\begin{bmatrix} 0 & 6 & 3 \\ 8 & 4 & 1 \\ 5 & 2 & 7 \end{bmatrix} \quad \begin{bmatrix} 6 & 2 & 4 \\ 1 & 5 & 8 \\ 3 & 7 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 4 & 7 \\ 8 & 2 & 5 \\ 3 & 6 & 0 \end{bmatrix} \quad (7)$$

Each object selects a different threshold map by choosing a randomly determined configuration based on a hash of the object's ID and front-facing flag. The full threshold computation is provided in Listing 4.

To summarize the changes compared to DTAA:

1. We use a 3×3 threshold map with 9 different thresholds instead

© 2025 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.



(a) *Rested* **(b)** *In Motion* **(c)** *+Aligned Vectors*

Figure 5: *Our* 3×3 *dithering pattern rendered with TAA. Moving the camera in **(b)** causes visible artifacts: the color of the bush in the middle changes significantly. By aligning the motion vectors to multiples of three pixels, the artifacts are mitigated, as shown in **(c)**. The effect is best visible in the supplemental video.*

Listing 5: *Stochastic Bilinear Interpolation: mvec represents the 2D motion vector in uv-coordinates, and frameDim denotes the screen resolution in pixels.*

```
1  float align = 3.0; // set to 5 for 5x5 grid
2  float2 pixelMvec = mvec * frameDim;
3  float2 pixelMin = floor(pixelMvec/align)*align;
4  float2 pixelMax = ceil(pixelMvec/align)*align;
5  float2 f = frac(pixelMvec / align);
6  float rng = hash3D(float3(pixel, frameIndex));
7
8  if (rng > f.x) {
9   if((rng-f.x)/(1-f.x)>f.y) pixelMvec = pixelMin;
10  else pixelMvec= float2(pixelMin.x, pixelMax.y);
11 } else { // rng <= f.x
12  if(rng/f.x <= f.y) pixelMvec = pixelMax;
13  else pixelMvec= float2(pixelMax.x, pixelMin.y);
14 }
15 mvec = pixelMvec / frameDim;
```

of a 5×5 threshold map with 5 thresholds. A smaller region with more thresholds provides better stratification and reduces surface noise.

2. We shift the map in a serpentine pattern instead of a vertical pattern. A vertical shift would only cycle through 3 thresholds, whereas a serpentine shift ensures that all 9 thresholds are used. Furthermore, the serpentine shift ensures that consecutive thresholds differ by more than one between frames.
3. We use unique threshold maps for each object instead of a common map for all objects.
4. Our blue noise mask contributes only $1/9$ to the final threshold, reducing the visibility of the static blue noise mask.

3.3.1. Motion Vectors for Screen-Space Dithering

Screen-space aligned dithering patterns introduce visible artifacts during subpixel motion when combined with TAA (see Fig. 5). This occurs because the expected temporal sequence of the dithering thresholds will be disrupted by camera motion:

Consider DTAA, where each pixel cycles through a fixed threshold sequence ($0.1 \rightarrow 0.5 \rightarrow 0.9 \rightarrow 0.3 \rightarrow 0.7 \rightarrow 0.1 \rightarrow \ldots$). If the camera does not move, TAA will use the same pixel from last frame for reprojection. In that case the temporal threshold sequence will be sampled as intended. However, with camera motion, reprojection may sample a pixel that used a different threshold. This disrupts the expected temporal sequence and results in some pixels moving faster or slower through the sequence, causing visible flickering, as seen in our video. In the worst case, a pixel may repeatedly sample the same threshold.

To prevent this issue, we align the motion vectors with the dithering pattern size. For DTAA, motion vectors are quantized to multiples of 5 pixels, while for STD, they are quantized to multiples of 3 pixels.

Rounding motion vectors to the nearest quantized value may cause visible lag in the background. Therefore, we use stochastic bilinear interpolation instead [FP23], which is shown in Listing 5.

Motion vector adjustment is necessary when using our dithering pattern with TAA. We also recommend applying it with FSR, as it causes similar but less severe artifacts. Using the motion vector adjustment with DLSS slightly reduces temporal lag during spontaneous motion but also slightly increases noise for low-transparency borders. Aside from that, DLSS does not exhibit the same artifacts as TAA and FSR. Therefore, we do not use the motion vector adjustment with DLSS.

3.4. Hybrid Approach

While our stochastic methods improve performance, they introduce noise that can degrade visual quality. To balance performance and quality, we propose a hybrid approach that combines the baseline renderer with stochastic methods (Russian roulette or dithering).

The implementation is straightforward: We trace rays and accumulate visibility in the closest-hit shader until the accumulated visibility falls below a threshold (e.g. 50%). Beyond this threshold, we switch to the stochastic method and trace only one final ray. If the accumulated visibility reaches zero before the stochastic method is applied, the stochastic method is skipped entirely.

The threshold is dynamically adjustable based on the rendering budget. A threshold of 0% corresponds to the baseline renderer, while 100% corresponds to the pure stochastic method.

As a threshold can be problematic for scenes containing many low-transparency objects, we suggest falling back to a k-buffer in such cases.

3.5. Raster Implementation

The implementation of our STD (Sec. 3.3) for rasterization works as follows: First, we perform a single pass over all geometry by rendering a VBuffer [BH13] with default depth testing into a texture. The VBuffer stores 16 bytes of geometric information, including the instance ID, primitive index, and barycentric coordinates.

© 2025 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

For transparent materials, we apply the STD threshold test (Listing 4) and discard the fragment if the test fails. This ensures that we obtain identical hit points as in the ray tracing implementation.

Next, we use the VBuffer to shade each pixel. In our implementation, we use ray-traced deep shadows for simplicity. However, in a practical rasterization-based implementation, this should be replaced with (deep) shadow maps [LV00]. The shading pass also generates motion vectors, which, along with the depth buffer, serve as inputs for the temporal anti-aliasing techniques.

Notably, implementing a hybrid version of the rasterized method proves difficult, as we do not have in-order traversal of the transparent fragments. One possible implementation could utilize a k-buffer to obtain the first fragments in sorted order, but this does not ensure a fixed visibility threshold. Another possibility would be to use a linked list [YHGT10] to record and sort all fragments. After accumulating the foremost fragments, until the visibility threshold is reached, we could switch our STD method and select one final fragment from the remainder of the list.

4. Results

In this section, we compare the following methods:

- **Baseline (Base)**: A full ray tracing renderer that shades all fragments (Sec. 3.1).
- **Russian Roulette (RR)**: A minimal stochastic transparency method that shades only one fragment per pixel (Sec. 3.2).
- **Our Spatio-Temporal Dithering (STD)**: Our novel spatio-temporal dithering method (Sec. 3.3).
- **Hybrid X%** Our hybrid with a threshold of X% (Sec. 3.4).
- **STD Raster**: Our STD implemented for rasterization (Sec. 3.5).
- **Dither Temporal AA (DTAA)**: The Unreal Engine 4 dithering method (Sec. 2.2), implemented within our framework.
- **DTAA***: A modified version of DTAA that employs unique threshold maps for each object, as described in Sec. 2.2.

We implemented all methods in the Falcor framework [KCK*22] and conducted our experiments on an NVIDIA RTX 4090 GPU at a resolution of 1920×1080 pixels. We used DLSS version 310.1.0 and FSR version 3.1.3, evaluating both methods at native resolution. The convolutional DLSS preset was chosen over the newer transformer-based preset, as the latter introduces significant ghosting with our methods.

Rendering times reflect the full ray tracing pass, including acceleration structure rebuilds, but exclude post-processing anti-aliasing techniques (TAA, DLSS, FSR).

For a fair comparison, alpha testing is applied to cutout materials such as grass and leaves, rather than treating them as transparent.

We report SSIM [WBSS04] values for each method, which are computed by comparing the rendered image against the baseline. An SSIM value of 1.0 indicates a perfect match, while lower values indicate less similarity.

We begin with a comparison of the visual quality of the different methods (Sec. 4.1), evaluating them in a static scene, under camera movement, and with object movement. Next, we present a performance comparison between the baseline renderer, our STD and the hybrid variant (Sec. 4.2). Finally, we analyze the quality and performance trade-offs of the hybrid renderer across various threshold settings (Sec. 4.3).

Additionally, we show a small comparison of our STD with spatiotemporal blue noise masks [WMAR22] in the appendix (Sec. 7.2).

4.1. Quality

We begin by comparing Base, RR, STD, and DTAA in a synthetic scenario to analyze their temporal behavior with DLSS in a static scene and under camera motion. For this purpose, we inserted a semi-transparent plane into the Emerald Square [NB17].

As shown in Fig. 6, our STD produces a clearer surface than the other stochastic techniques. The blue noise pattern of DTAA remains distinctly visible, whereas RR results in a highly noisy surface. During camera motion, as illustrated in Fig. 7, our technique becomes noisier but still delivers the best results.

Fig. 8 shows the results under object movement. DTAA skips many transparent objects due to its use of a single threshold map for all objects. This issue is significantly improved in DTAA*. While our STD generally shows slight improvements over DTAA*, these differences are less noticeable in motion. For example, our cloud appears less noisy on the left side, and the area indicated by the blue arrow is closer to the reference.

We also tested our methods with smoke particles, as shown in Fig. 9. RR remains quite noisy, while STD and DTAA* produce similar results, as most of the noise is masked by the inherent noisiness of the smoke particles. Near the edges of the smoke cloud, our STD is slightly noisier than DTAA, but the opaque background appears sharper.

Fig. 10 shows our STD in the fight scene with different post-processing anti-aliasing techniques. DLSS produces the best results, followed by FSR and then TAA. TAA results in significant blurring during object motion, while FSR is sharper but still appears blurry compared to DLSS.

4.2. Performance

Since all stochastic methods exhibit similar performance, we compare only the baseline renderer with our STD and the hybrid renderer variations. Hybrid 95% refers to the variant that uses the baseline renderer until visibility falls below 95%, at which point it switches to our STD.

Fig. 11 presents the performance breakdown in the fight scene, while Fig. 12 shows the results in Emerald Square with smoke particles.

In the fight scene, we observed that our STD is consistently faster than the baseline renderer, with the performance gain depending on the number of transparent particles on screen. As expected, the hybrid versions fall between our STD and the baseline renderer, but thresholds below 75% do not provide speedups due to the relatively low number of transparent objects in the scene.

The Emerald Square demonstrates that the baseline renderer and

© 2025 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

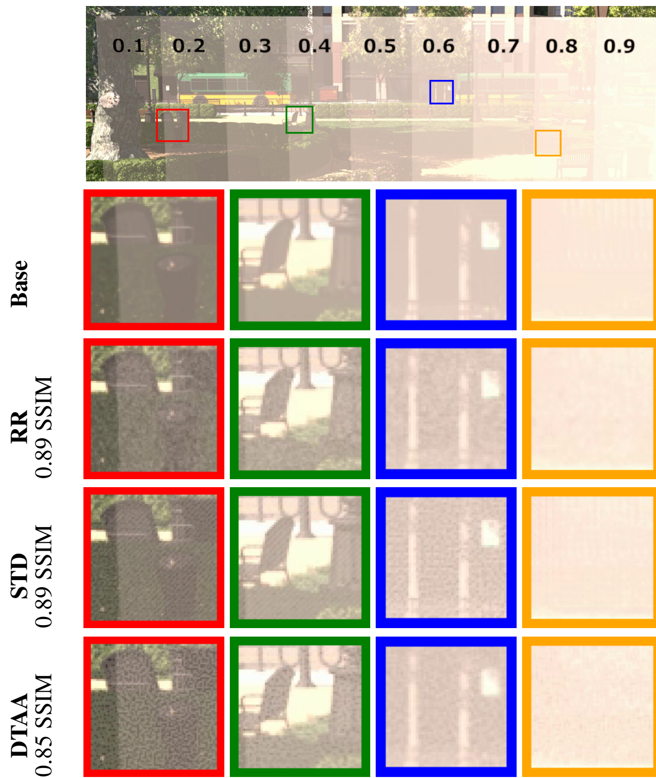


Figure 6: *Algorithms in a static scene with DLSS.*

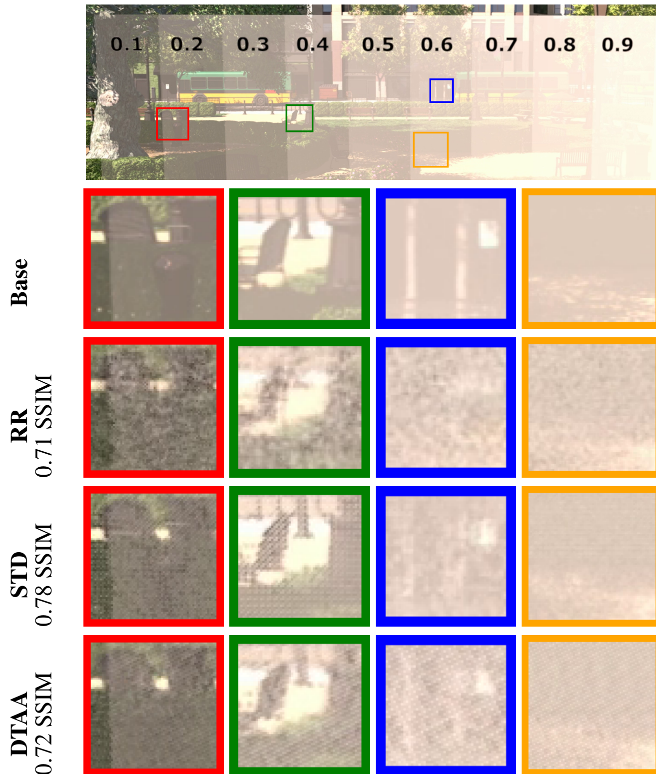


Figure 7: *Algorithms under left to right camera movement.*

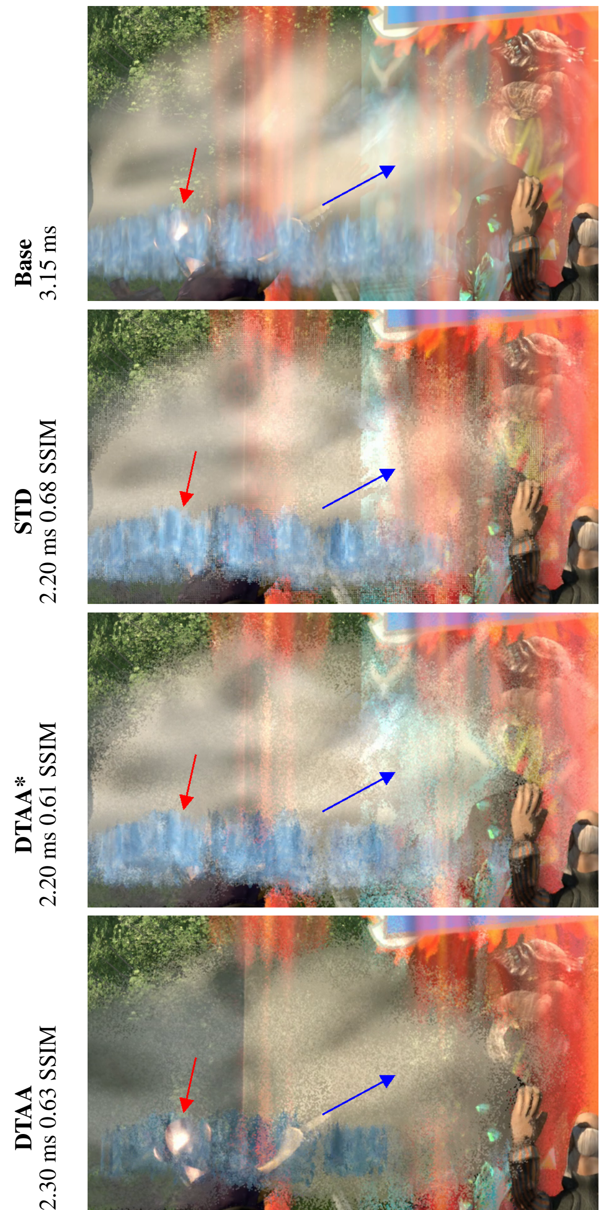


Figure 8: *Algorithms in fully animated fight scene. The red arrow shows that DTAA masks out too many transparent objects, compared to the other methods.*

our STD have similar rendering times when no transparent particles are present. However, when smoke particles appear, the baseline renderer becomes significantly slower. Here, our STD is more than three times faster due to the large number of transparent particles in the smoke cloud. One contributing factor is that shading smoke particles becomes increasingly expensive as they are deeper in the cloud. This occurs because the shadow rays must traverse a larger portion of the cloud when connecting to the light source.

Notably, the rendering time of our STD remains almost constant when passing through the smoke cloud.

© 2025 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

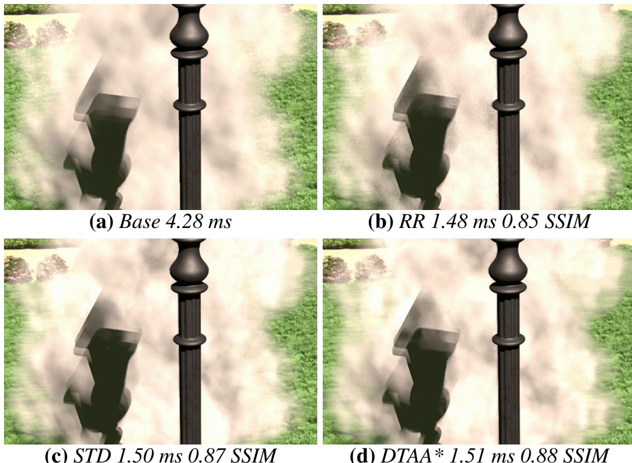

(a) *Base 4.28 ms* (b) *RR 1.48 ms 0.85 SSIM*
(c) *STD 1.50 ms 0.87 SSIM* (d) *DTAA* 1.51 ms 0.88 SSIM*

Figure 9: *Algorithms with smoke particles in the Emerald Square. The shadow of the lamp post can be seen on the smoke particle layers. The camera is moving from right to left.*

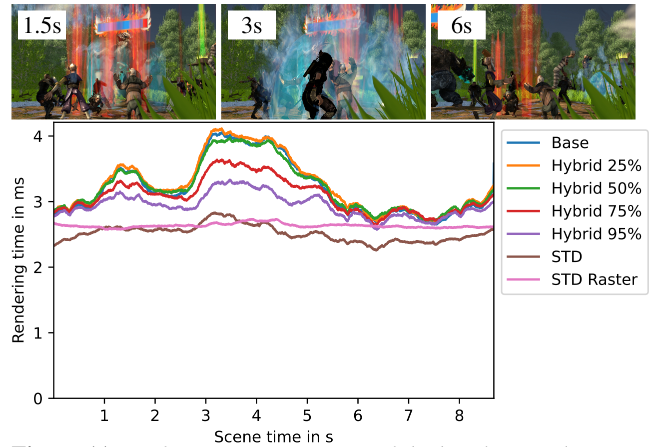


Figure 11: *Performance comparison of the baseline renderer, our STD and the hybrids in the fight scene.*

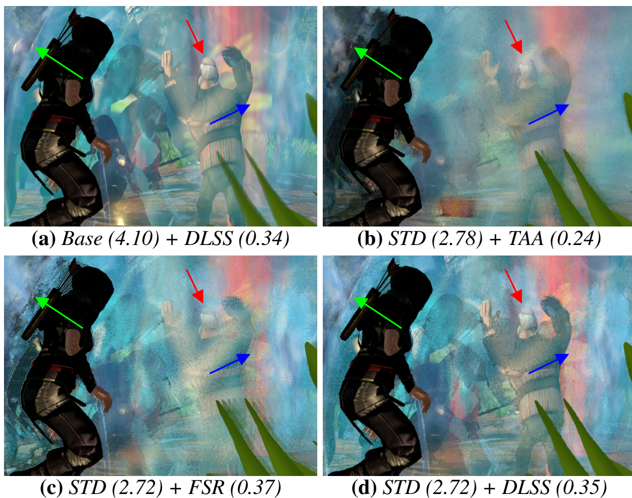

(a) *Base (4.10) + DLSS (0.34)* (b) *STD (2.78) + TAA (0.24)*
(c) *STD (2.72) + FSR (0.37)* (d) *STD (2.72) + DLSS (0.35)*

Figure 10: *Our STD in the fight scene with TAA, DLSS and FSR. The green arrow highlights severe ghosting in TAA and FSR. The red and blue arrows indicate notable regions of transparency accumulation. The numbers in parentheses represent rendering times in milliseconds.*

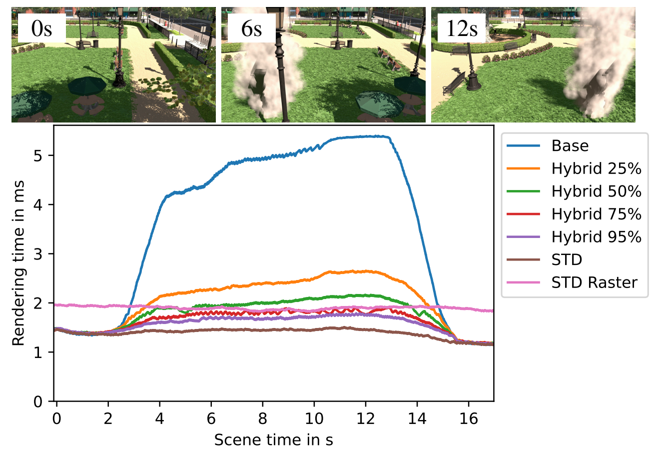


Figure 12: *Performance comparison of the baseline renderer, our STD and the hybrids in the Emerald Square with smoke particles.*

Our ray-traced STD and rasterized STD exhibit similar performance in the fight scene. However, in the Emerald Square, our ray-traced STD is approximately 30% faster than the rasterized STD, despite utilizing frustum culling for the rasterizer. For larger scenes, ray tracing is generally more efficient due to its logarithmic time complexity with respect to the number of objects.

4.3. Hybrid Quality

Fig. 13 shows the visual results of varying the thresholds. At 50%, the hybrid renderer produces results that are nearly indistinguishable from the baseline renderer, but the runtime improvements are less significant. Switching from pure STD to the hybrid renderer at 95% results in a noticeable improvement in visual quality for the first intersection but also leads to a significant drop in performance.

Ultimately, the threshold can be dynamically adjusted to fit the current frame budget. The decline in visual quality should be less noticeable to users when many transparent objects are present during large-scale object movement, where stable frame rates are likely more important.

5. Conclusion

We presented a novel spatio-temporal dithering technique (STD) that produces stable and visually pleasing results. Our method is significantly faster than naive transparency rendering, as it shades only a single fragment per pixel and does not require evaluating all possible fragments per pixel. This gives our technique a clear performance advantage over existing OIT techniques.

We demonstrated how our technique can be integrated with existing anti-aliasing methods. By thoroughly investigating its interaction with DLSS, we discovered that transparency accumulation is too aggressive and proposed a correction function to mitigate

© 2025 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

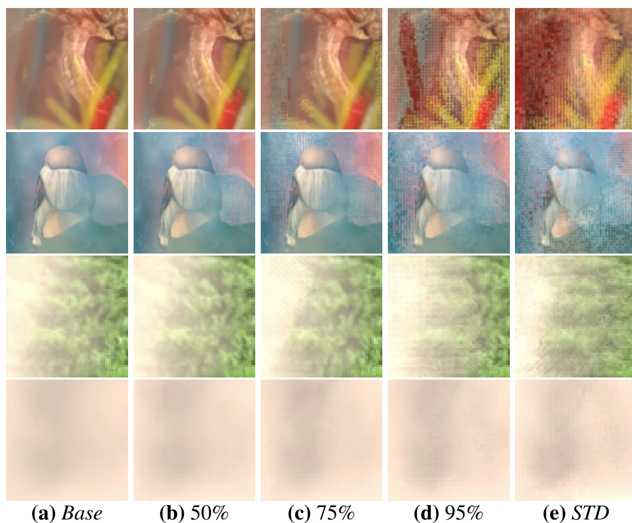


Figure 13: *Quality of the hybrid renderer in the fight scene (top) and Emerald Square (bottom).*

this issue. Additionally, we identified potential motion vector issues that cause major artifacts in TAA and FSR and introduced a motion vector alignment technique to address this problem.

In scenes with only a few transparent surfaces, the baseline renderer is usually sufficient, as our method offers minimal performance gains while introducing visible artifacts. However, when transparency becomes more prevalent, our STD helps maintain stable frame rates. The hybrid renderer serves as a flexible framework, seamlessly adjusting the rendering method to balance performance and visual fidelity in real time.

We believe our technique will become even more relevant in the future, as higher pixel densities and refresh rates will make its noise less perceptible. Furthermore, our method integrates well with popular screen-space effects, as it shades only one fragment per pixel per frame. This simplifies the application of effects such as depth-of-field and makes it more feasible to fully rely on ray-traced shadows.

6. Future Work

Since our technique relies on dither masks, there is a risk of objects masking each other out. Developing a technique with similar quality that avoids systematically masking out objects would be highly beneficial. Currently, we select object masks based on the object ID and front-facing flag, which prevents proper rendering of objects with more than two intersections.

One possible solution is to pin the noise pattern to the UV coordinates of objects, similar to the approach of Wyman and McGuire [WM17], rather than using screen-space coordinates. We conducted initial experiments in this direction, but the results were generally noisier than our current technique.

The visual quality of our method is still noticeably different from that of the baseline renderer. Further improvements to overall quality would be desirable. It would also be interesting to explore the potential of using outputs from our stochastic method to train machine learning-based anti-aliasing techniques.

Currently, our technique is applied only to primary rays. In theory, similar techniques could be developed for shadow rays, reflections, and refractions.

We also tested our STD with NVIDIA's RayReconstruction, a combined denoising and anti-aliasing technique. While the results for smoke particles were acceptable, flat surfaces were challenging to render. Further investigation is needed to adapt our technique for compatibility with RayReconstruction.

Our technique can also be adapted for interactive rendering of Gaussian splats.

Supplementary Material

The source code can be found on our Github project page: https://github.com/TU-Clausthal-Rendering/SpatioTemporalDithering.

Acknowledgements

This work was partially supported by the German Research Foundation (DFG) grant GR 3833/4-1, Project Nr. 524961573.

References

[AMD24] AMD. *FidelityFX Super Resolution 3.1.3 (FSR)*. 2024. URL: https://gpuopen.com/fidelityfx-superresolution-2/ 2, 3.

[Bay73] BAYER, B. "An optimum method for two level rendition of continuous tone pictures". *Proc. of IEEE Int. Communication Conf.* Vol. 1. 1973, 2611–2615. URL: https://web.archive.org/web/20130512190753/http://white.stanford.edu/~brian/psy221/reader/Bayer.1973.pdf 2.

[BCL*07] BAVOIL, LOUIS, CALLAHAN, STEVEN P., LEFOHN, AARON, et al. "Multi-fragment Effects on the GPU Using the K-buffer". I3D '07. New York, NY, USA: ACM, 2007, 97–104. URL: http://doi.acm.org/10.1145/1230100.1230117 2.

[BG20] BRÜLL, FELIX and GROSCH, THORSTEN. "Multi-Layer Alpha Tracing". *Vision, Modeling, and Visualization*. The Eurographics Association, 2020. DOI: 10.2312/vmv.20201183 2.

[BH13] BURNS, CHRISTOPHER A. and HUNT, WARREN A. "The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading". *Journal of Computer Graphics Techniques (JCGT)* 2.2 (2013), 55–69. URL: http://jcgt.org/published/0002/02/04/ 6.

[DF03] DALY, SCOTT and FENG, XIAOFAN. "Bit-depth extension using spatiotemporal microdither based on models of the equivalent input noise of the visual system". *Proceedings of SPIE - The International Society for Optical Engineering* 5008 (2003). DOI: 10.1117/12.472016 3.

[Epi25] EPIC GAMES. *Unreal Engine 4: Dither Temporal AA*. 2025. URL: https://www.unrealengine.com/ 3, 5.

[ESSL10] ENDERTON, ERIC, SINTORN, ERIK, SHIRLEY, PETER, and LUEBKE, DAVID. "Stochastic Transparency". I3D '10. Washington, D.C.: ACM, 2010, 157–164. URL: http://doi.acm.org/10.1145/1730804.1730830 2, 3.

[FP23] FAJARDO, MARCOS and PHARR, MATT. "Fast Procedural Noise By Monte Carlo Sampling". *Eurographics Symposium on Rendering*. The Eurographics Association, 2023. DOI: 10.2312/sr.20231141 6.

© 2025 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

[FS75] FLOYD, R.W. and STEINBERG, L. "An adaptive algorithm for spatial gray scale". *Proc Soc Inform Display* 17 (1975) 3.

[Int25] INTEL CORPORATION. *Intel® Xe Super Sampling (X^eSS) API Developer Guide*. 2025. URL: https://www.intel.com/content/www/us/en/developer/articles/guide/xe-super-sampling-developer-guide.html 2.

[KCK*22] KALLWEIT, SIMON, CLARBERG, PETRIK, KOLB, CRAIG, et al. *The Falcor Rendering Framework*. 2022. URL: https://github.com/NVIDIAGameWorks/Falcor 7.

[LV00] LOKOVIC, TOM and VEACH, ERIC. "Deep shadow maps". *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. USA: ACM Press/Addison-Wesley Publishing Co., 2000, 385–392. DOI: 10.1145/344779.344958 7.

[MB13] MCGUIRE, MORGAN and BAVOIL, LOUIS. "Weighted Blended Order-Independent Transparency". *Journal of Computer Graphics Techniques (JCGT)* 2.2 (2013), 122–141. URL: http://jcgt.org/published/0002/02/09/ 2.

[McG16] MCGUIRE, MORGAN. *The Graphics Codex*. 2.13. Casual Effects, 2016. URL: https://graphicscodex.com 4, 11.

[MCTB12] MAULE, MARILENA, COMBA, JOAO L. D., TORCHELSEN, RAFAEL, and BASTOS, RUI. "Memory-Efficient Order-Independent Transparency with Dynamic Fragment Buffer". SIBGRAPI '12. IEEE Computer Society, 2012, 134–141. URL: http://dx.doi.org/10.1109/SIBGRAPI.2012.27 1.

[MGvW98] MULDER, J.D., GROEN, F.C.A., and van WIJK, J.J. "Pixel masks for screen-door transparency". *Proceedings Visualization '98 (Cat. No.98CB36276)*. 1998, 351–358. DOI: 10.1109/VISUAL.1998.745323 3.

[MKKP18] MÜNSTERMANN, CEDRICK, KRUMPEN, STEFAN, KLEIN, REINHARD, and PETERS, CHRISTOPH. "Moment-Based Order-Independent Transparency". *I3D '18* 1.1 (2018), 7:1–7:20. DOI: 10.1145/3203206 2.

[MMP*24] MOENNE-LOCCOZ, NICOLAS, MIRZAEI, ASHKAN, PEREL, OR, et al. "3D Gaussian Ray Tracing: Fast Tracing of Particle Scenes". *ACM Transactions on Graphics and SIGGRAPH Asia* (2024). URL: https://gaussiantracer.github.io/ 2.

[NB17] NICHOLAS HULL, KATE ANDERSON and BENTY, NIR. *NVIDIA Emerald Square, Open Research Content Archive (ORCA)*. 2017. URL: http://developer.nvidia.com/orca/nvidia-emerald-square 7.

[NVI25] NVIDIA. *Deep Learning Super Sampling*. 2025. URL: https://research.nvidia.com/labs/adlr/DLSS4/ 2, 3.

[PD84] PORTER, THOMAS and DUFF, TOM. "Compositing Digital Images". *SIGGRAPH Computer Graphics* 18.3 (1984), 253–259. URL: http://doi.acm.org/10.1145/964965.808606 2.

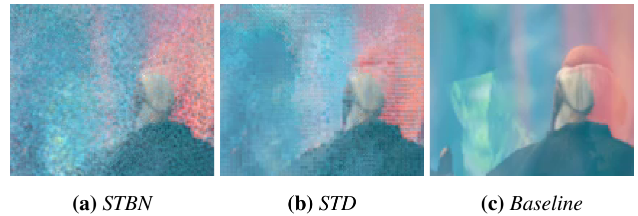
[Sal16] SALVI, MARCO. "An Excursion in Temporal Supersampling". *Game Developers Conference*. 2016. URL: https://developer.download.nvidia.com/gameworks/events/GDC2016/msalvi_temporal_supersampling.pdf 2–4.

[SML11] SALVI, MARCO, MONTGOMERY, JEFFERSON, and LEFOHN, AARON. "Adaptive Transparency". HPG '11. Vancouver, British Columbia, Canada: ACM, 2011, 119–126. URL: http://doi.acm.org/10.1145/2018323.2018342 2.

[SV14] SALVI, MARCO and VAIDYANATHAN, KARTHIK. "Multi-layer Alpha Blending". I3D '14. ACM, 2014, 151–158. URL: http://doi.acm.org/10.1145/2556700.2556705 2.

[VVP20] VASILAKIS, A. A., VARDIS, K., and PAPAIOANNOU, G. "A Survey of Multifragment Rendering". *Computer Graphics Forum* 39.2 (2020), 623–642. DOI: https://doi.org/10.1111/cgf.14019 2.

[VVPM17] VASILAKIS, ANDREAS-ALEXANDROS, VARDIS, KONSTANTINOS, PAPAIOANNOU, GEORGIOS, and MOUSTAKAS, KONSTANTINOS. "Variable k-buffer using Importance Maps". The Eurographics Association, 2017. DOI: 10.2312/egsh.20171005 2.



(a) *STBN* **(b)** *STD* **(c)** *Baseline*

Figure 14: *STBN (blue noise) and STD in the animated fight scene.*

[WBSS04] WANG, ZHOU, BOVIK, A. C., SHEIKH, H. R., and SIMONCELLI, E. P. "Image Quality Assessment: From Error Visibility to Structural Similarity". *Trans. Img. Proc.* 13.4 (2004), 600–612. DOI: 10.1109/TIP.2003.819861 7.

[WM17] WYMAN, CHRIS and MCGUIRE, MORGAN. "Hashed Alpha Testing". *I3D '17*. 2017, 11. URL: https://casual-effects.com/research/Wyman2017Hashed/index.html 10.

[WMAR22] WOLFE, ALAN, MORRICAL, NATHAN, AKENINE-MÖLLER, TOMAS, and RAMAMOORTHI, RAVI. "Spatiotemporal Blue Noise Masks". *Eurographics Symposium on Rendering*. The Eurographics Association, 2022. DOI: 10.2312/sr.20221161 3, 7, 11.

[Wym16] WYMAN, CHRIS. "Exploring and expanding the continuum of OIT algorithms". HPG '16. Dublin, Ireland: Eurographics Association, 2016, 1–11. URL: https://cwyman.org/papers/hpg16_oitContinuum.pdf 2.

[YHGT10] YANG, JASON C., HENSLEY, JUSTIN, GRÜN, HOLGER, and THIBIEROZ, NICOLAS. "Real-time Concurrent Linked List Construction on the GPU". The Eurographics Association, 2010. URL: http://dx.doi.org/10.1111/j.1467-8659.2010.01725.x 1, 7.

[YLS20] YANG, LEI, LIU, SHIQIU, and SALVI, MARCO. "A Survey of Temporal Antialiasing Techniques". *Computer Graphics Forum* 39 (2020). URL: http://behindthepixels.io/assets/files/TemporalAA.pdf 2.

7. Appendix

7.1. Hash Function

Hash function by McGuire [McG16], that is used for Russian roulette (Sec. 3.2) and threshold map selection (Sec. 3.3). We also tested a variety of other hash functions, but found no significant differences.

Listing 6: *Hash by McGuire*

```
1  float hash(float2 v) { // [McG16]
2   return frac(1.0e4 * sin(17.0 * v.x + 0.1 * v.y)↩
        * (0.1 + abs(sin(13.0 * v.y + v.x))));
3  }
4  float hash3D(float3 v) {
5   return hash(float2(hash(v.xy), v.z));
6  }
7  float hash4D(float4 v) {
8   return hash(float2(hash(v.xy), hash(v.zw)));
9  }
```

7.2. Spatio-Temporal Blue Noise

We also implemented a screen-space **spatio-temporal blue noise (STBN)** technique, based on the spatio-temporal blue noise masks by Wolfe et al. [WMAR22]. To prevent objects from masking each

© 2025 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

other out, we use a different offset into the blue noise texture, similar to our STD method.

Overall, STBN produces similar results when viewed from a distance, but the pattern appears noisier than our STD, as shown in Fig. 14. While our STD produces a more pixelated appearance during motion, STBN resembles white noise.

Another disadvantage of STBN is its incompatibility with our motion vector alignment technique (Sec. 3.3.1), as the blue noise texture is 128×128 in size, which is too large for stochastic bilinear interpolation. Although this works well with DLSS, it causes flickering when used with TAA.

© 2025 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.