Procedural Phasor Noise

Tan Cetiner

March 2024

1 Introduction

Procedural noise algorithms are widely used in computer graphics to generate natural-looking textures, terrains, and other visually complex features. One such algorithm is Procedural Phasor Noise, proposed in the paper by Thibault Tricard, Semyon Efremov, Cédric Zanni, Fabrice Neyret, Jonàs Martínez, and Sylvain Lefebvre [1]. Procedural phasor noise is a technique that can efficiently generate high-quality noise patterns with a wide range of frequencies and spatial properties.

The importance of procedural noise in computer graphics cannot be overstated. It allows for the creation of detailed and realistic-looking environments without the need for storing and processing vast amounts of texture data. Procedural noise algorithms are particularly useful in applications such as game development, visual effects, and architectural visualization, where the generation of complex, yet consistent, patterns is essential.

The Procedural Phasor Noise paper presents a novel approach to generating high-quality procedural noise that addresses some of the limitations of traditional noise functions, such as Perlin noise and Worley noise. The proposed technique aims to provide a more efficient and versatile noise generation method that can be easily integrated into real-time rendering pipelines.

2 Overview of the paper

The Procedural Phasor Noise paper introduces a new algorithm for generating high-quality procedural noise. The key idea behind the phasor noise technique is to represent the noise as a sum of sinusoidal waves, or "phasors," with varying frequencies and phases.

The paper begins by explaining the limitations of traditional noise functions, such as their inability to generate noise with a wide range of frequencies and their sensitivity to aliasing artifacts. The authors then present the phasor noise algorithm as a solution to these problems.

The phasor noise algorithm works by first generating a set of random phasors, each with a unique frequency and phase. These phasors are then combined using a weighted sum to produce the final noise value. The authors describe the mathematical formulation of the phasor noise function and explain how it can be efficiently computed using a combination of trigonometric identities and lookup tables.

One of the key advantages of the phasor noise algorithm is its ability to generate noise with a wide range of frequencies and spatial properties. By adjusting the distribution of the phasor frequencies and phases, the authors demonstrate that the algorithm can produce noise patterns with different levels of detail and coherence.

Additionally, the paper discusses the implementation details of the phasor noise algorithm, including the use of optimization techniques to improve the performance of the noise generation

[1] Tricard, T., Efremov, S., Zanni, C., Neyret, F., Martínez, J., & Lefebvre, S. (2019). Procedural phasor noise. ACM Transactions on Graphics (TOG), 38(4), 1-12.

process. The authors also provide examples of the noise patterns generated by their algorithm and compare them to the results of other popular noise functions.

Overall, the Procedural Phasor Noise paper presents a novel and efficient approach to generating high-quality procedural noise, with potential applications in a wide range of computer graphics and visualization tasks.

3 Implementation Details

The implementation of the procedural phasor noise algorithm was done within the base code provided in the INF584 - Image Synthesis course. The base code, referred to as the MyRenderer project, was designed to support physically-based rendering with point lights.

To preserve the simplicity of the base code and focus on the phasor noise implementation, the parts of the base code not directly used for the phasor noise, such as the ray tracing source code, were removed. This allowed for a more streamlined and efficient implementation.

The key steps in the implementation process were:

- Utilizing the phasor noise code provided by Thibault Tricard on the ShaderToy website. This code was adapted and integrated into the fragment shader of the base code.
- Modifying the base code to support the use of the procedural phasor noise texture in addition to the physically-based rendering with point lights. This was achieved by adding the necessary uniform variables, such as the phasor noise texture, to the fragment shader and making the corresponding changes in the C++ side of the code.
- Ensuring that the procedural phasor noise texture could be applied to the meshes while still allowing for the physically-based rendering with point lights. This was accomplished by incorporating the phasor noise evaluation into the main rendering function, where the final albedo color is calculated by multiplying the material albedo with the phasor noise color.

The key source code snippets illustrating the implementation are as follows:

The fragment shader code includes the procedural phasor noise evaluation function ‘phasorNoise()‘, which is then used to modulate the final albedo color in the ‘main()‘ function:

Listing 1: PPNFragmentShader.glsl

```
// .......
// Previous Code

vec2 eval_noise(vec2 uv, float f, float b)
{
    init_noise();
    float cellsz = 2.0 *_kr;
    vec2 _ij = uv / cellsz;
    ivec2 ij = ivec2(_ij);
    vec2 fij = _ij - vec2(ij);
    vec2 noise = vec2(0.0);
    for (int j = -2; j <= 2; j++) {
        for (int i = -2; i <= 2; i++) {
            ivec2 nij = ivec2(i, j);
            noise += cell(ij + nij , fij - vec2(nij),f,b);
        }
    }
    return noise;
}
```

```
vec3 phasorNoise(vec2 uv) {
    vec2 phasorNoise = eval_noise(uv, _f, _b);
    float phi = atan(phasorNoise.y, phasorNoise.x);
    float I = length(phasorNoise);
    return vec3(sin(phi) * 0.3 + 0.5);
}

void main() {
    vec3 n = normalize(fNormal);
    vec3 wo = normalize(-fPosition);
    vec3 noiseColor = phasorNoise(fTexCoord);
    vec3 finalAlbedo = material.albedo * noiseColor;
    colorResponse = vec4(pbrShading(wo, n, finalAlbedo, material.roughness,
        material.metallicness), 1.0);
}
```

By integrating the procedural phasor noise implementation into the existing base code, the final result supports both the physically-based rendering with point lights and the application of the phasor noise texture on the meshes.

4 Resulting Images

Figure 1: ram.off

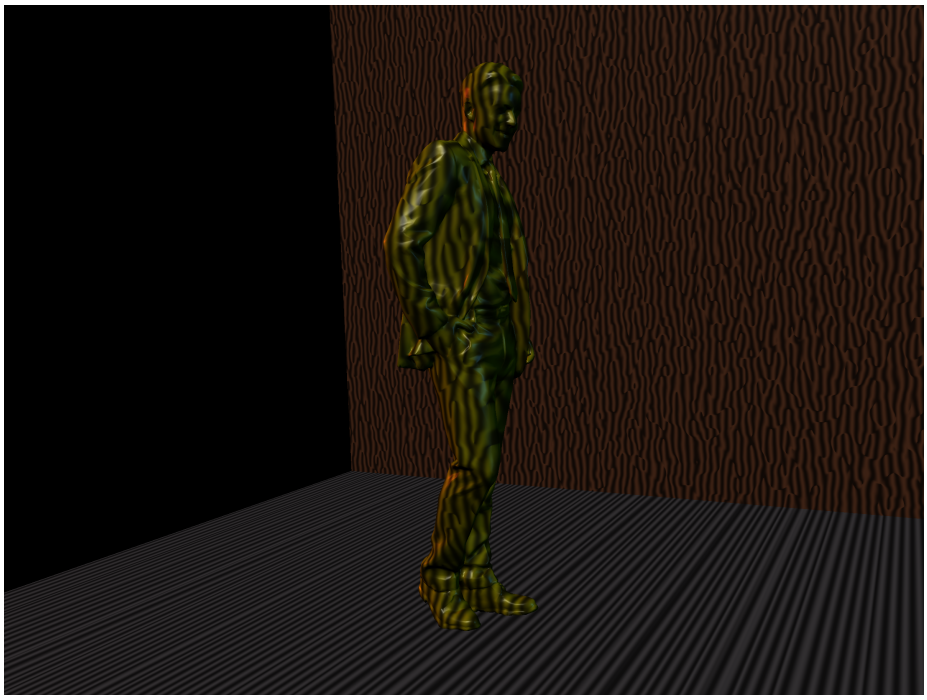

Figure 2: denis.off

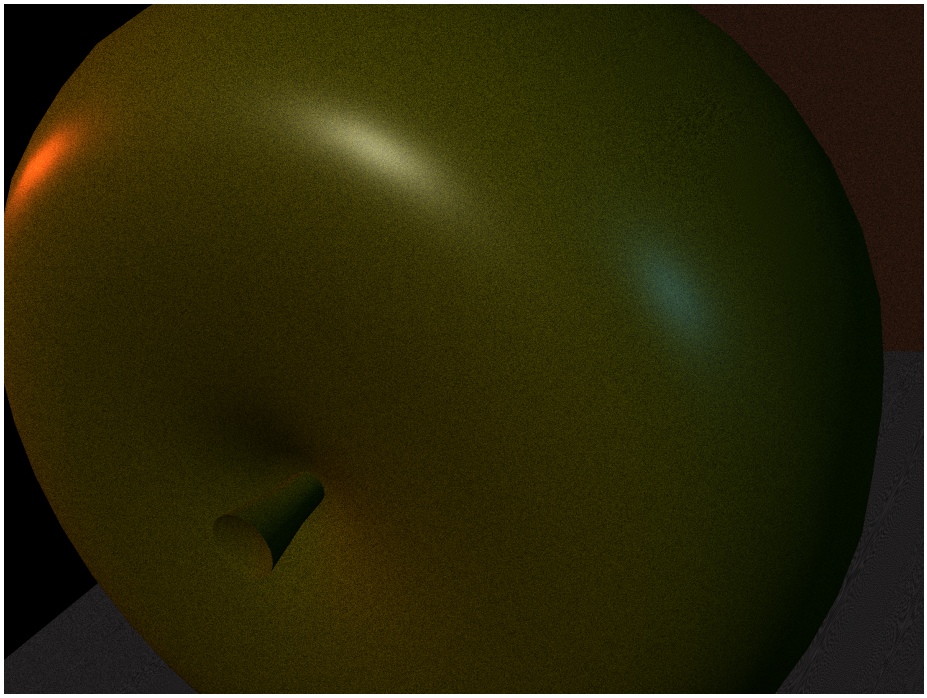

Figure 3: apple.off

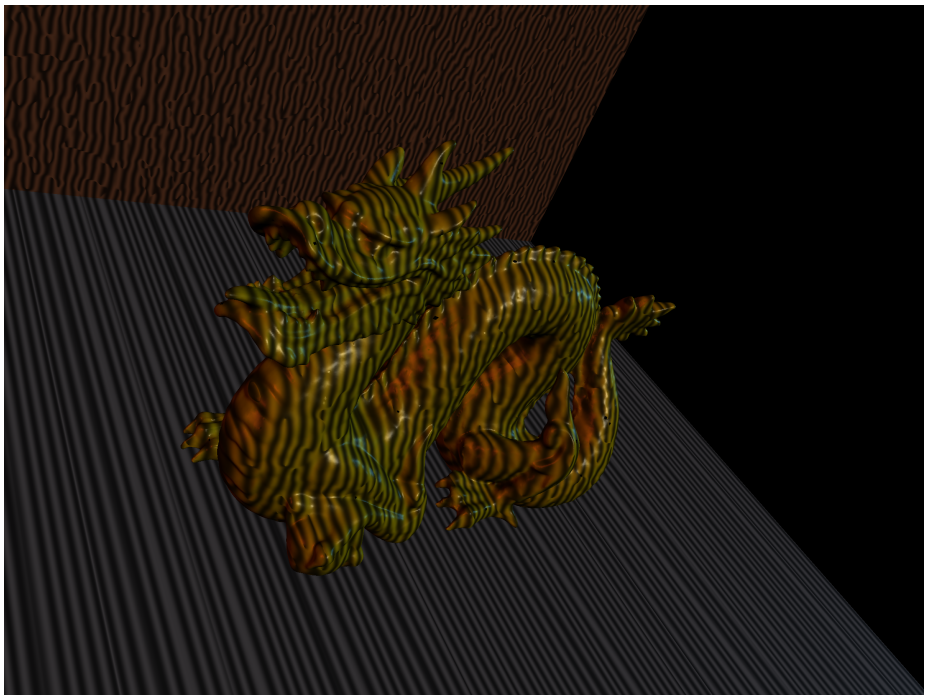

Figure 4: dragon.off

5 Conclusion

In this project, an experimental approach was taken to integrate the procedural phasor noise algorithm into the existing physically-based rendering pipeline provided by the MyRenderer base code. The goal was to combine the phasor noise as a texture on the meshes while preserving the physically-based shading with point lights.

The implementation involved adapting the phasor noise code from the ShaderToy website and integrating it into the fragment shader of the base code. The necessary changes were made on the C++ side to enable the use of the phasor noise just like a texture alongside the physically-based rendering.

The resulting images showcase the application of the procedural phasor noise on the meshes, providing an interesting and visually appealing texture. However, one key limitation of the current implementation is the performance impact. As observed, the addition of the phasor noise calculation has resulted in a significant drop in the frame rate (FPS) of the rendered scene.

Future work on this project could focus on optimizing the performance of the phasor noise implementation, potentially by exploring techniques such as caching, level-of-detail rendering, or leveraging hardware-accelerated noise generation. Additionally, further experimentation with the phasor noise parameters could lead to the discovery of more visually compelling texture variations.

Overall, this project demonstrates the feasibility of integrating a procedural noise algorithm like phasor noise into an existing physically-based rendering pipeline, although the performance aspect remains a challenge that requires further investigation and optimization.