Progressive path tracer based on DXR

Bachelor thesis - English abstract
Last edited: 17.11.2020

Author: Kamil Grodecki

1. Introduction

1.1. Raytracing

Ray Tracing is being called the “Holy Grail” of rendering. World surrounding us is composed of light transportation, which causes multiple bounces from different surfaces until all energy is lost. This is what RT is simulating. Therefore RT (especially Path Tracing) is a better approximation of light behaviour and surrounding reality than rasterization.

Tracing rays allows to extract values from surface hit. Then, the programmer can work with those values to present the final image. If there is no geometry intersection, then *miss shader*, which is a programmable step, is used.

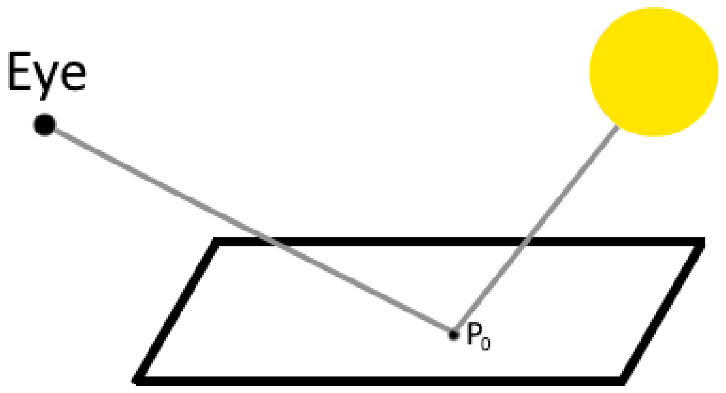

Image 1.1 - Shading with Ray tracing

Above image presents ray tracing used for shading calculation. Algorithm starts with tracing a visible camera point which data is found by calculating intersection with AS (Acceleration Structure). It returns information about traced points and lighting information which are passed to the shader through CBV. It is enough to calculate shading for a given pixel which will be described in detail below.

1.2. Path Tracing

Path Tracing is generalization of Ray Tracing, because it allows multiple reflections of light from the surfaces, while RT stops at the first intersection. In practice, PT cannot be used in real time applications due to its cost. In most cases, RT is using 1 spp in full or half-res. PT is even more costly, what could be seen in *Results* for diffuse term (2.2). Cost increases linearly with each light bounce.

Tracing rays allows to simulate light transportation in a natural way. Every reflection of ray is adding color to the path and propagates current color to the environment. Multiple bounces allow light to fill the room even if there is no direct light hit.

In theory, an infinite number of samples should present an image indistinguishable from reality. In practice, we don't have enough time for it. At some point, calculation must be stopped and we must be satisfied with achieved results at that point. There might be multiple reasons to stop the algorithm.
You can deduce, based on many metrics, that the quality of the generated image is satisfying and no further work is needed. In the case of our Path Tracer, the user is choosing input parameters that controls how long the algorithm is going to be working. Most important inputs are the number of light reflections in the path and number of frames that contributed to the final image.

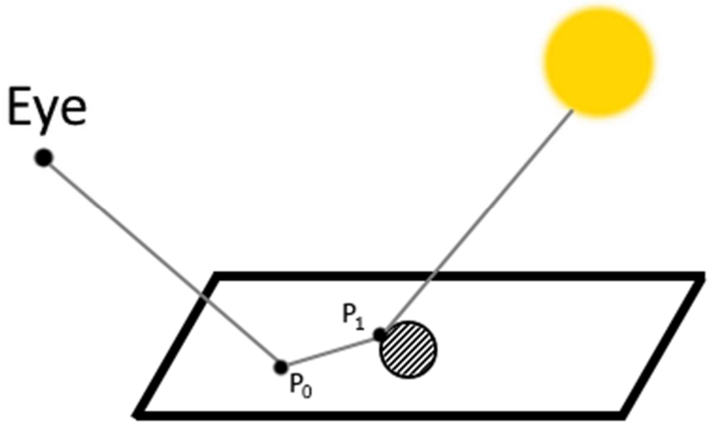

Image 1.2 - Shading with Path tracing

2. Diffuse Path Tracing GI

2.1. Implementation

Path Tracing in the most basic form is a simple algorithm. Below we present a basic version which can be used as *ground truth reference* but for commercial game engines it will most likely be too slow. However, it works good as a reference that further approximations can be compared to. In this work, I'm focused on presenting a working reference, so slow but correct results are enough.

PT algorithm consists of below steps:

1. Choose position inside pixel, transform from NDC to worldspace and find *primary ray* direction
2. Cast *primary ray*, find surface intersection and calculate shading at that point (RT ends here)
3. Find the direction of reflected light and cast new ray, then calculate surface color - This step is repeated as long as there is intersection with geometry or until path length exceeds specified maximum
4. Collect full path data - return final pixel color in current frame
5. Calculate new pixel color based on historical frames and current frame. Current frame weight is 1/N, where N is number of frames up until this point

6. Repeat 1 to 5 until maximum number of frames is reached

2.1.1. Finding primary ray direction

DispatchRaysIndex() returns top-left pixel coordinates in NDC. Depending on the sampling method, we will choose a different *pixelOffset*.
Lines 3-4 returns a sampling position on the screen.
That position is in *projection space* and we need to transform it to *world space*. There are multiple methods to accomplish that, but we are using an *inverted view projection matrix* to do so.
direction is our final *primary ray* direction that will be used in following code.

```
1      inline void GenerateCameraRay(uint2 index, uint2 dimensions,
               float4x4 projectionToWorld, inout float3 origin, out float3
direction,
               in float2 pixelOffset)
2      {
3          float2 xy = index + pixelOffset;
4          float2 screenPos = (xy / (float2) dimensions) * 2.0 - 1.0;
5
6          // Invert Y for DirectX-style coordinates.
7           screenPos.y = -screenPos.y;
8
9          // Unproject the pixel coordinate into a ray.
10         float4 world = mul(float4(screenPos, 0, 1), projectionToWorld);
11
12         world.xyz /= world.w;
13         direction = normalize(world.xyz - origin);
14     }
```

Listing 2.1 Generating primary ray direction based on NDC and inverse view projection matrix

2.1.2. Calculate radiance at intersection point

As a quick reminder - below you can see the *rendering equation* modified with Monte Carlo method, so it allows to calculate integral in finite time.

$$\frac{1}{N}\sum_{k=1}^{N}\frac{L_i(\mathbf{p},\mathbf{w_k})fr(\mathbf{w_k},w_o)\cos\theta}{p(\mathbf{w_k})}$$

In this step, we would like to calculate the right side of the equation. We need to calculate radiance of light source (**L_i**), BRDF (**fr**), cosinus between N (surface normal) and L (direction of light source) and PDF (**p**).

$$\frac{L_i(\mathbf{p},\mathbf{w_k})fr(\mathbf{w_k},w_o)\cos\theta}{p(\mathbf{w_k})}$$

We would like to do a series of transformations in order to calculate *diffuse term*. *Lambertian surface* BRDF is albedo divided by PI number. PDF depends on the sampling method, but throughout this whole thesis, we will be using mostly *cosine hemisphere* PDF, which is defined as cos/PI. It results in:

$$\frac{L_i(\mathbf{p}, \mathbf{w_k}) * albedo * \cos\theta}{\pi} \frac{\pi}{\cos\theta}$$

$$L_i(\mathbf{p}, \mathbf{w_k}) * albedo$$

Lambertian shading does not depend on the observer's position. It is based only on dot product of **N**, **L** and light intensity. In this work, we use coefficient equal 1 for all light sources so this variable will be omitted in further discussions.
[Driscoll2009] [28] explains that conserving energy for *diffuse surfaces* requires multiplication by 1/PI. This way, we can present final equation that will be used for ray tracing *diffuse term*:

$$\frac{albedo * NoL}{\pi}$$

We also cannot forget to check if a light source is visible. It is either 0 or 1, depending on the existence of a geometry intersection between calculated point and light.

```
// Load hit data
float3 hitPos = WorldRayOrigin() + WorldRayDirection() * RayTCurrent();
float3 triangleNormal, triangleTangent, triangleBitangent;
loadHitData(triangleNormal, triangleTangent, triangleBitangent, attribs);

float4 albedo = albedoTexture.Load(int3(DispatchRaysIndex().xy, 0));

// Calculate light data
float3 V = g_sceneCB.cameraPosition.xyz;
float3 N = triangleNormal.xyz;

float3 L = normalize(g_giCB.sunDirection);
float3 toLight = L;
float distToLight = 1e+38f;

// Check visibility
float NoL = saturate(dot(N, L));
float visibility = shadowRayVisibility(hitPos, L, 1e-3f, distToLight);

// Calculate light contribution to point in world (diffuse lambertian term)
payload.color += visibility * NoL * albedo.rgb * INV_PI;
```

Listing 2.2 Calculating diffuse term for worldspace point using ray tracing

2.1.3. Choosing next ray direction

Direction of the reflected ray might be calculated in many ways depending on sampling method. In this work, we are mostly using *Correlated Multi-Jittered Sampling* [29], but other methods are also explained in the following chapters.

After choosing next ray direction, we need to create a payload and fill it with data. We also need to increment path length which is necessary to terminate recursion, when path length limit is reached.

```
if (g_giCB.samplingType == SAMPLE_UNIFORM)
{
  if (payload.pathLength == 0)
  {
     brdfSample = HammersleyDistribution(g_giCB.accFrames,
                   g_giCB.maxFrames, uint2(payload.rndSeed, payload.rndSeed2));
  }
  else
  {
      uint seed = initRand(LaunchIndex.x + LaunchIndex.y *
                          LaunchDimensions.x, g_sceneCB.frameCount, 16);
      uint seed2 = initRand(LaunchIndex.x + LaunchIndex.y *
                          LaunchDimensions.x, g_sceneCB.frameCount, 17);
      brdfSample = HammersleyDistribution(payload.pathLength,
                          g_giCB.bounceCount, uint2(seed, seed2));
  }
}
else if (g_giCB.samplingType == SAMPLE_MJ)
{
   brdfSample = SamplePoint(payload.rndSeed, payload.rndSeed2);
}
else if (g_giCB.samplingType == SAMPLE_RANDOM)
{
    brdfSample.x = nextRand(payload.rndSeed);
    brdfSample.y = nextRand(payload.rndSeed2);
}

// Continue spawning rays if path left has not reached maximum
if (payload.pathLength < g_giCB.bounceCount)
{
    float3 rayDirTS = SampleDirectionCosineHemisphere(brdfSample.x,
                                                          brdfSample.y);
    rayDirWS = normalize(mul(rayDirTS, tangentToWorld));

    // Prepare payload
    PayloadIndirect newPayload;
    newPayload.pathLength = payload.pathLength + 1;
    newPayload.rndSeed = payload.rndSeed;
    newPayload.rndSeed2 = payload.rndSeed2;
    newPayload.color = float3(0, 0, 0);
    newPayload.throughput = throughput;
37
38       // Calculate next ray bounce color contribution
39       float3 bounceColor = shootIndirectRay(hitPos, rayDirWS, 1e-3f,
newPayload);
40       payload.color += bounceColor * throughput;
41    }
```

Listing 2.3 Choosing direction of next ray reflection

2.1.4. Extract path color

Path color is calculated by terminating Path Tracing recursion (listing 2.3, lines 28-29). After *primary ray* emission, every hit is collecting colors. Reaching path length limit (listing 2.3, line 3) returns color from all paths, which results in the final pixel color in that frame.

2.1.5. Temporal accumulation

Progressive path tracing is based on *temporal accumulation* - output color is calculated using historical data. As mentioned in previous chapters, we are using Monte Carlo method. Therefore, each frame’s weight is equal to 1/N, where N is current frame count.

```
// Calculate pixel color in current pass and merge with previous frames
float4 finalColor = float4(shootIndirectRay(primaryRayOrigin,
       primaryRayDirection, 1e-3f, indirectPayload), 1.0f);

if (g_giCB.accFrames > 0) {
   float4 prevScene = RTOutput[LaunchIndex];
   finalColor = ((float) g_giCB.accFrames * prevScene + finalColor) /
                            ((float) g_giCB.accFrames + 1.0f);
}
RTOutput[LaunchIndex] = finalColor;
```

Listing 2.4 Frame accumulation over time

2.2. Results

GTX 1660 Super 6GB was used to generate images in 1920x1080 resolution.

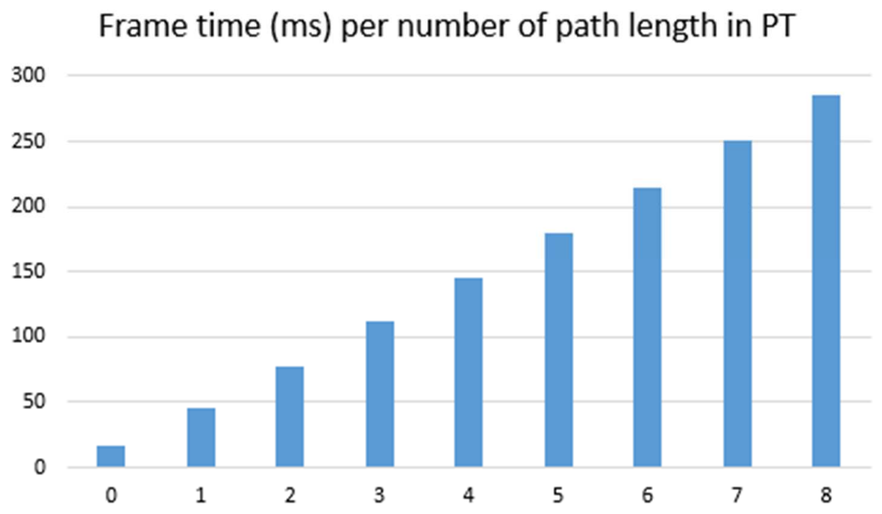

Chart 2.1. Frame time depending on number of reflections in Path Tracing algorithm using diffuse term

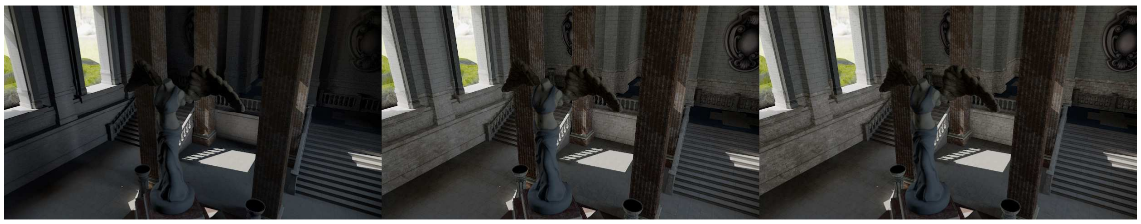

Image 2.1 - Diffuse path tracing 1000 spp - from the left: 1 bounce, 4 bounces, 8 bounces

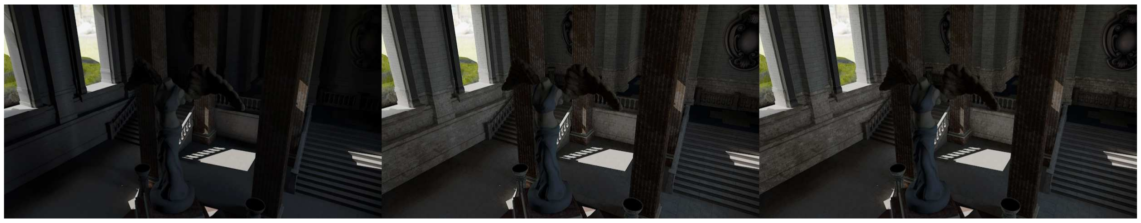

Image 2.2 - Diffuse path tracing 5000 spp - from the left: 1 bounce, 4 bounces, 8 bounces

The only source of light in the test scene was *directional lighting*. It was meant to show the worst case scenario where almost all of the light is the scene reflected light.
We also wanted to show advantages of using Ray/Path Tracing. It can fill a scene with light, even if there is a little direct light. Without Path Tracing, the only visible thing would be the window and a little of floor that is directly lit. Everything else in this scene would be completely black.

In traditional pipeline using rasterization, it would be solved by using of existing GI methods which are fast but are only an approximation in comparison to Ray Tracing.

3. Specular Path Tracing GI

3.1. Theory

Reflected specular light direction is enclosed in shape determined by BRDF. *Specular BRDF* has a dominant direction which is mostly correlated with NDF.

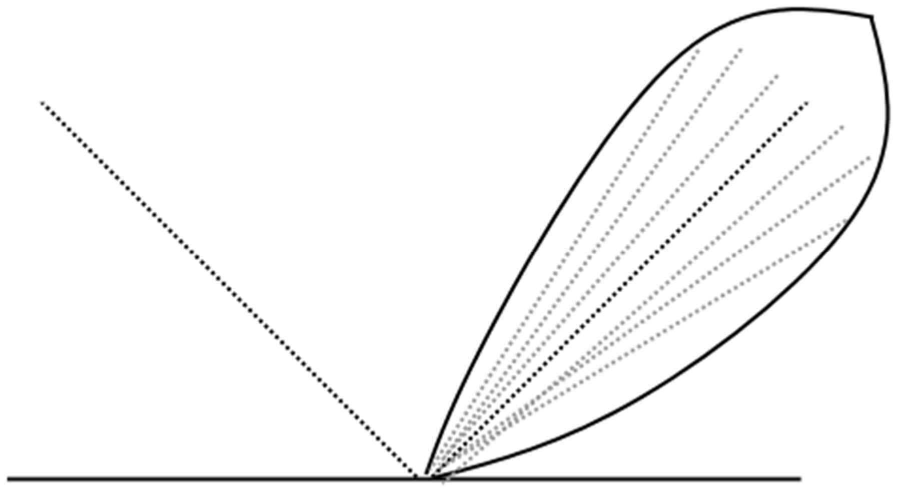

Image 3.1 - Example of specular BRDF

Final look of Cook-Torrance BRDF [30] is based on *Normal Distribution (D), Geometry shading (G), Fresnel term (F).*

$$f_r(w_i, w_o) = \frac{D(w_m)\ G(w_i, w_o, w_m)\ F(w_i, w_m)}{4\ |w_i \cdot w_g|\ |w_o \cdot w_g|}$$

Our engine uses isotropic GGX for NDF [31], Smith GGX for geometry shading and Schlick Fresnel [32]. Our choice was based on good results of GGX presented in previous years, which caused it to become industry standard. It has a long tail and presents results close to reality.

3.2. GGX PDF

Normal Distribution Function is the most important variable in determining specular BRDF shape. Because of that, it is an obvious choice to use it as PDF. Joe Schutte [34] published a blog post explaining in detail how to calculate GGX PDF for normal distribution function.

3.2.1. Derivation of a formula

$$f_r = \frac{F(w_i, w_m)\ G_2(w_i, w_o, w_m)\ D(w_m)}{4\ |w_i \cdot w_g|\ |w_o \cdot w_g|} \quad (10)$$

Usually, G_2 means G_1(N, V) * G_1(N, L). It is ***Smith*** *Geometry Shadowing*. It doesn't relate directly to the equation. It is just a way of splitting **G** into two parts. G_1(N, V) checks correlation between surface normal and *microfacet*. G_1(N, L) - correlation between *microfacet* and light source. It is described in detail by Blinn [36].

Isotropic GGX NDF:

$$D(w_m) = \frac{\alpha^2}{\pi((w_g \cdot w_m)^2\ (\alpha^2 - 1) + 1)^2} \quad (11)$$

PDF using spherical coordinates:

$$p(\theta, \phi) = \frac{\alpha^2\ cos(\theta)\ sin(\theta)}{\pi(cos^2(\theta)\ (\alpha^2 - 1) + 1)^2} \quad (12)$$

Our goal is to find a new ray direction (w_i) and normal vector for microfacet (w_m) and transform the equation using *importance sampling*. Based on random values in range [0, 1] in 2D space, we get spherical coordinates - **theta, phi**. Cao Jaiyin [37] explains complete process of creating coordinates using GGX CDF.
Moving from spherical coordinates to cartesian system, we are getting w_m as a result. Based on that variable, we will calculate reflected direction w_i:

$$w_m = (r\ sin\theta\ cos\phi, r\ cos\theta, r\ sin\theta\ sin\phi) \quad (13)$$

$$w_i = 2|w_o \cdot w_m| w_m - w_o \quad (14)$$

Moving from spherical coordinates (**theta, phi**) to cartesian (x, y, z), it is required to use Jacobian r^2sin (theta). Another transformation for w_i requires Jacobian 4|dot(w_o, w_m)|. Second Jacobian origin is explained in details by Walter [38]. Derivation of below equations are provided by Joe Schutte [34] in the blogpost that we are describing here. Our derivation below is shorter, so interested readers should read original post.

$$p(\theta, \phi) = \frac{\alpha^2\ cos(\theta)\ sin(\theta)}{\pi(cos^2(\theta)\ (\alpha^2 - 1) + 1)^2}$$

$$p(\theta, \phi) = \frac{\alpha^2\ cos(\theta)\ sin(\theta)}{\pi(cos^2(\theta)\ (\alpha^2 - 1) + 1)^2} * \frac{1}{sin(\theta)}$$

$$p(\theta, \phi) = \frac{\alpha^2\ cos(\theta)}{\pi(cos^2(\theta)\ (\alpha^2 - 1) + 1)^2}$$

$$p(w_m) = D(w_m)\ cos(\theta)$$

$$p(w_m) = D(w_m)(w_m \cdot w_g)$$

$$p(w_m, w_o) = \frac{D(w_m)(w_m \cdot w_g)}{4|w_o \cdot w_m|}$$

Using PDF to *importance sample* BRDF, we present final equation for throughput:

$$\frac{f(w_i, w_o)|w_i \cdot w_g|}{p(w_m, w_o)} = \frac{F(w_i, w_m)\ G_2(w_i, w_o, w_m)\ |w_o \cdot w_m|}{|w_o \cdot w_g|\ |w_m \cdot w_g|}$$

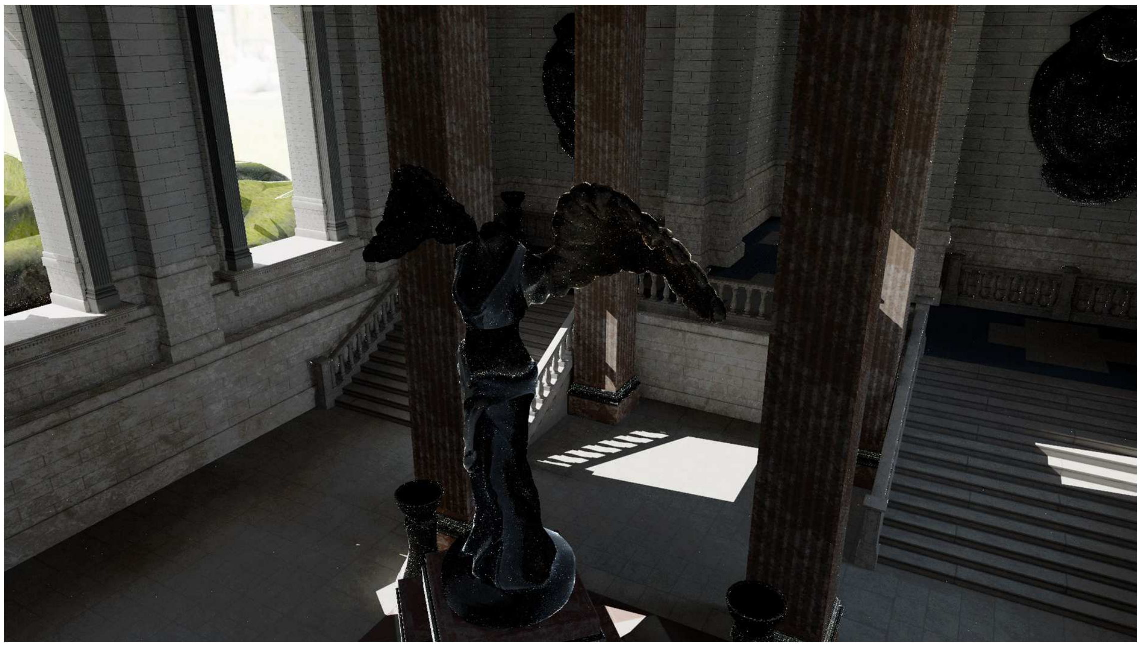
Image 3.2 - GGX importance sampling NDF (5000 spp, 4 bounces, exposure 8.1)

3.3. GGX Visible Normals

In 2014, Heitz presented a better way to decrease variance and avoid *fireflies* - bright, single pixels which decrease overall image quality.

As the main problem, he identified too big difference in PDF for different viewing angles and wasting samples for low roughness objects below horizon.
In his work [40], Heitz presents his method based on *Distribution of Visible Normals*, which he presented in his previous work, explaining NDF and Geometry Shadowing [39].

$$D_i(w_m) = \frac{G_1(w_i, w_m)\ |w_i \cdot w_m|\ D(w_m)}{|w_i \cdot w_g|}$$

However it was too complicated and too costly. In the following years, Heitz was working on BSDF and in 2017, he created a simple and fast algorithm, giving better results than one presented in 2014. *A Simpler and Exact Sampling Routine for the GGX Distribution of Visible Normals* [41] presents a simple and effective way of finding w_m with full implementation included. However it is recommended to read his previous works [39][40] with deep understanding of GGX PDF [34] and Smith Geometry Shadowing [35].

Apart from changing the way of calculating w_m [41], the only change is using *Distribution of Visible Normals* presented in the equation above instead of *NDF*. For interested readers - full derivation is presented by Joe Schutte [59].
Final throughput is:

$$\frac{F(w_i, w_m)\ G_2(w_i, w_o, w_m)}{G_1(w_o, w_m)}$$

Image 3.3 - GGX sampling of visible normals (5000 spp, 4 bounces, exposure 8.1)

4. Postprocesses

4.1. Exposure settings

In Path Tracing, image is a result of multiple light reflections. Different input parameters will present different brightness of output image. To make comparison of output images easier, we are using *exposure control:*

$$color = \frac{color * 2^{(x)}}{scale}$$

In our case, scale is 2^(-10), ***x*** is the exposure setting chosen by user through GUI.

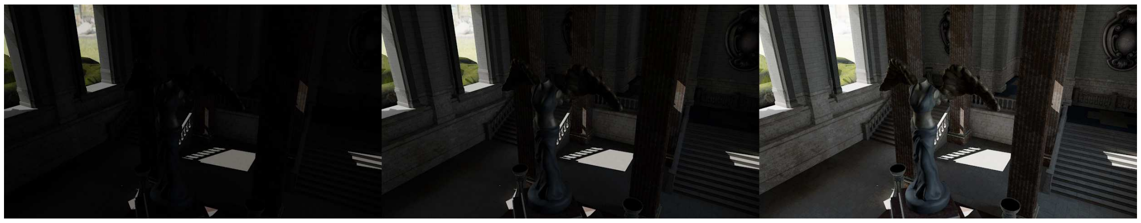
Image. 4.1 - Presenting different exposure settings. From the left: x = -12.0, x = -10.0 (basic image without changing exposure), x = -8.0

4.2. Tone mapping

In 2015, Unreal Engine 4 changed its default tone mapping to **ACES Filmic Tone Mapping Curve**. Their research shows that new algorithm presents better results than using Uncharted 2 tone mapper [42]. They used curves researched by AMPAS [44]. Results of their work are explained by Epic Games employee in detail [43].

```
float3 ACESFilm(float3 x)
{
   float a = 2.51f;
   float b = 0.03f;
   float c = 2.43f;
   float d = 0.59f;
   float e = 0.14f;
   return saturate((x*(a*x+b))/(x*(c*x+d)+e));
}
```

Listing 4.1 Implementation of ACES Filmic Tone Mapping

4.3. Linear to sRGB

Uncharted 2 [42] is converting between linear and gamma space using $x^{2.2}$. However this approximation is not truly correct and in recent years, there is a more push towards correct

transformation between linear and gamma space.
Because of that, our path tracer is using full equation to convert from linear to sRGB space [45]:

$$\gamma(u) = \begin{cases} 12.92u & u \leq 0.0031308 \\ 1.055u^{1/2.4} - 0.055 & otherwise \end{cases}$$

4.4. Implementation

```
float4 main(PixelInputType input) : SV_TARGET
{
  // Get backbuffer color
  float3 color = g_texture.Sample(g_sampler, input.uv).rgb;

  // Define scale
  const float FP16Scale = 0.0009765625f;

  // Apply exposure scale settings
  color *= exp2(g_postprocessCB.exposure) / FP16Scale;

  // Apply filmic curve and move from linear to sRGB space
  color = ACESFilm(color);
  color.r = linearToSRGB(color.r);
  color.g = linearToSRGB(color.g);
  color.b = linearToSRGB(color.b);

  return float4(color, 1.0f);
}
```

Listing 4.2 Implementation of postprocesses

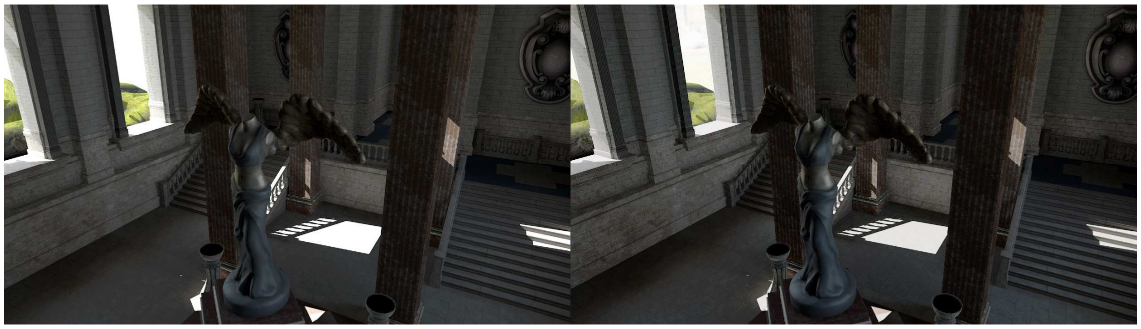

Image 4.2 - Comparison of image between and after applying postprocesses. The one postprocess on the left is exposure settings, used in order to get image of similar brightness (on the left: -6.0, ot the right: -8.0)

5. Energy compensation

5.1. White Furnace Test

In 2014, Heitz [39] explained that for a fully white environment, the correct lighting model should always return value equal 1, independently of input parameters. In this way, *White Furnace Test* allows to test if energy was added or lost in a process. Losing energy is especially visible for rough surfaces. Problems also occur in GGX that we use in this work. Because of that, it is necessary to research and find a solution for that problem.

Energy compensation is crucial in Path Tracing. Every light reflection in the path might result in adding or losing energy, which for long paths might give extremely different results.

5.2. Solution

Multiple ways of dealing with that problem were created in the past. However, Turquin's work in 2019 [46] turned out to be simple, giving satisfying results and provided full code implementation. Li [48] presented research on real life objects using *multiple scattering*. Comparison of achieved results with Turquin's work proved that his model is close to reality. It was provided by Krzysztof Narkowicz [47].

5.3. Deriving equation

[Turquin2019] [46] split his research in two parts - first regarding energy, second - approximation of *Fresnel term*. Basic BRDF equation is split into single scattering BRDF and multi scattering BRDF.

$$\rho(w_o, w_i) = \rho_{ss}(w_o, w_i) + \rho_{ms}(w_o, w_i)$$

$$\rho(w_o, w_i) = \rho_{ss}(w_o, w_i) + F_{ms} k_{ms}(w_o)\ \rho_{ss}(w_o, w_i)$$

5.3.1. Energy term

Let assume that F = 1. We can prove that:

$$\rho(w_o, w_i) = (1 + k_{ms}(w_o))\ \rho_{ss}(w_o, w_i) = \frac{\rho_{ss}(w_o, w_i)}{E_{ss}(w_o)}$$

$$k_{ms}(w_o) = \frac{1 - E_{ss}(w_o)}{E_{ss}(w_o)}$$

E_{ss} is albedo for *single scattering*. Above derivation is explained in detail by Turquin in his work [46]. In this work, it is enough to assume that the above statements are correct and that BRDF is equal to **rho$_{ss}$** normalized by single scattering albedo **E_{ss}**. E_{ss} can be precalculated and stored in texture, in our case name DFG.

5.3.2. Fresnel term

Turquin concluded that for his model, approximation $F_{ms} = F_0$ is enough, presenting good results. More details can be found in his work [46].

Final equation is given as:

$$\rho(w_o, w_i) = (1 + F_0 \frac{1 - E_{ss}(w_o)}{E_{ss}(w_o)})\ \rho_{ss}(w_o, w_i)$$

```
DFG = dfgTexture.SampleLevel(g_sampler, float2(saturate(
 dot(triangleNormal, -WorldRayDirection())), roughness), 0.0f).xy;
Ess = DFG.x;
throughput *= float3(1, 1, 1) + specularAlbedo * (1.0f / Ess - 1.0f);
```

Listing 5.1 Implementation of energy compensation algorithm [Turquin2019]

5.4. Results

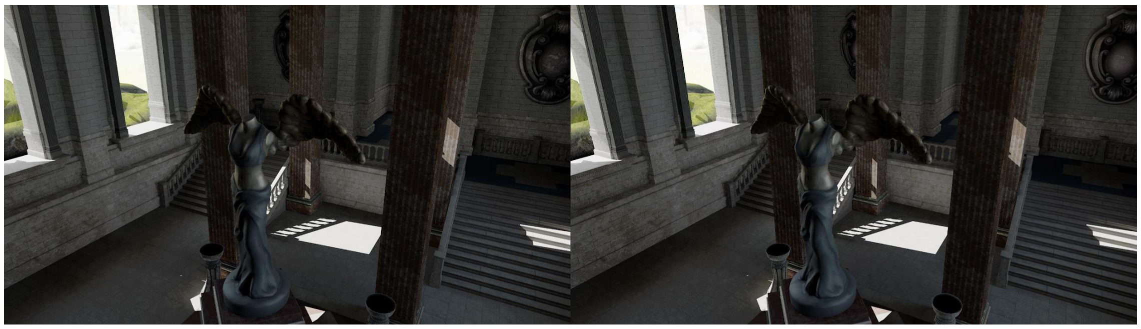

Image 5.1 On the left side - no energy compensation, on the right - using [Turquin2019]

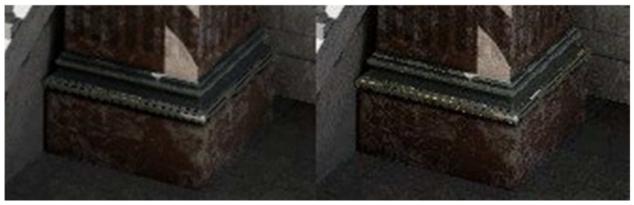

Image 5.2 On the left side - no energy compensation, on the right - using [Turquin2019] - zoomed-in image

Using energy compensation results in a more natural look. Without that technique, metal elements of the environment are too dark and matte. Above image presents how [Turquin2019] work improves image output. Both images were generated using 5000 spp with path length equal 4.

6. Sampling

6.1. Random sampling

Trivial method to choose the next ray direction is a random approach. Convergence is slower than other methods and output directions might be similar due to the random nature of that algorithm.
Therefore, instead of sampling all of the surrounding, samples might be clumping near a single point. In fully random sampling, we cannot really predict positions of samples. Usually quality will be low.
Based on numerical methods theory, with infinite number of samples we will achieve the same result as with other sampling methods. However, if it is possible, we would like to present an acceptable image as soon as possible. It is a naive method, which is not suitable for commercial usage.

6.2. Uniform sampling

Uniform sampling chooses samples with uniform distribution. It improves upon *random sampling* because there is no possibility of focusing all samples in a single point.

To choose a sample, we are using a low-discrepancy *sequence*. In the computer graphics field, most popular algorithms are ones developed by Halton [51] and Hammersley [52]. Dammertz [53] presented an algorithm for fast calculation of numbers in Hammersley sequence.

```
float radicalInverse_VdC(uint bits)
{
    bits = (bits << 16u) | (bits >> 16u);
     bits = ((bits & 0x55555555u) << 1u) | ((bits & 0xAAAAAAAAu) >> 1u);
    bits = ((bits & 0x33333333u) << 2u) | ((bits & 0xCCCCCCCCu) >> 2u);
    bits = ((bits & 0x0F0F0F0Fu) << 4u) | ((bits & 0xF0F0F0F0u) >> 4u);
    bits = ((bits & 0x00FF00FFu) << 8u) | ((bits & 0xFF00FF00u) >> 8u);
    return float(bits) * 2.3283064365386963e-10; // / 0x100000000
}

vec2 hammersley2d(uint i, uint N)
{
    return vec2(float(i)/float(N), radicalInverse_VdC(i));
}
```

Listing 6.1 Hammersley sequence created with [Dammertz2012]

However, this code requires improvement. Reversing bits, created by Warren [54], can be replaced with **reversebits()** introduced in HLSL SM 5.0.
Line 4 in listing 5.9 introduce pseudorandomness which improves quality of sampling. Variable **random** is created by using **DispatchRaysIndex()** oraz **DispatchRaysDimensions()**.

```
float2 HammersleyDistribution(uint i, uint N, uint2 random)
{
    float2 u = float2(float(i) / float(N), reversebits(i) * 2.3283064365386963e-
10);
    u = frac(u + float2(random & 0xffff) / (1 << 16));
    return u;
}
```

Listing 6.2 Improved Hammersley sequence used in this thesis

6.3. Correlated Multi-Jittered sampling

Correlated Multi-Jittered sampling [29] is continuation of works published in 1994 from the creators of original *Multi-Jittered sampling* [55]. Original work divided samples to ensure that there is only one sample in each row and column. To improve that, Kensler added condition ensuring that there is more distance between each sample. Thanks to that improvement, [Kensler2013] decreased samples clumping and increased overall quality.

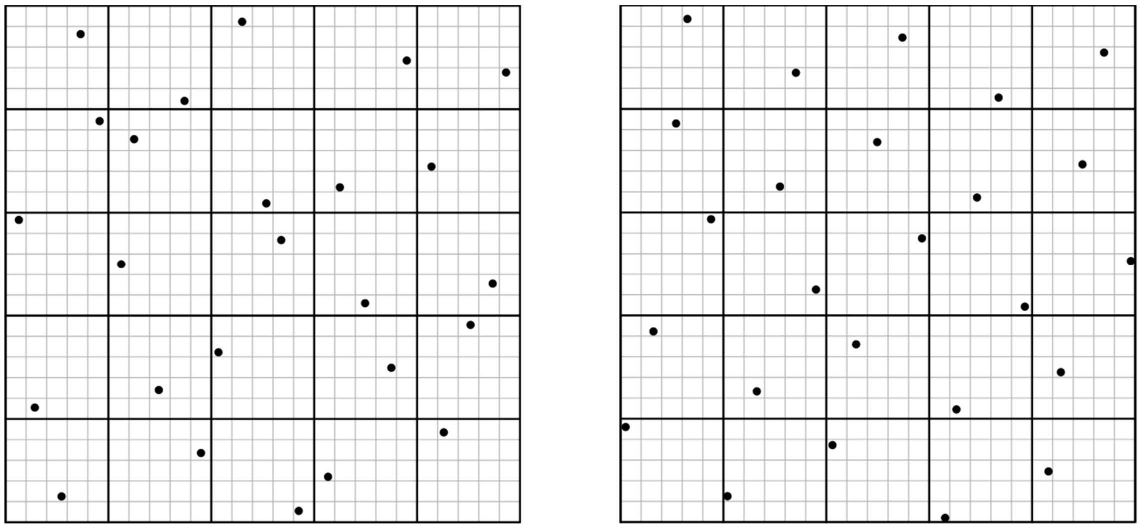

Image 6.1 - On the left: multi-jittered sampling, on the right: correlated shuffling. (Above image is part of Kensler's work and is not intellectual property of authors of this thesis [29])

In 2018, Pixar studios presented *progressive multi-jittered sampling*. One of the authors is Kensler [56]. Presented results are promising, however due to complexity of an algorithm, it was not tested in this thesis, but it should be tested in the future due to potential high quality of that sampling method.

6.4. Choosing point on hemisphere

Above sampling methods need to transformed from 2D space in range [0, 1] to hemisphere. It can be achieved in multiple ways, most popular are *uniform mapping* and *cosinus mapping*. In this work, we decided to use only *cosinus mapping* due to better image quality.

6.5. Results

As expected, worse results are achieved by using *random sampling* due to potential focusing and the same areas. Output images contain the most noise in comparison to other presented methods.
It is not obvious for many cases, if *uniform sampling* or *correlated multi-jittered sampling* provides better results. In our test scene (Sun Temple provided by UE4 and NVidia) there is visible superiority in using *correlated multi-jittered sampling* in a few places.

To discuss differences in sampling methods, we will use image 5.12. On the right we can see that *uniform sampling* causes visible aliasing. Let’s ignore that and assume that in the future we will add an antialiasing algorithm to the engine. Even with that assumption, *correlated multi-jittered sampling* provides better results. Take a look at the right side of the grey statue - convergence is better than *uniform sampling*.

In the left bottom corner, the difference between middle and bottom images are too small to deduce which one is better. Probably because very little rays reached that region. Overall *correlated multi-jittered sampling* proved to be the best method among those presented in our work. Therefore, we are using that method throughout this thesis.

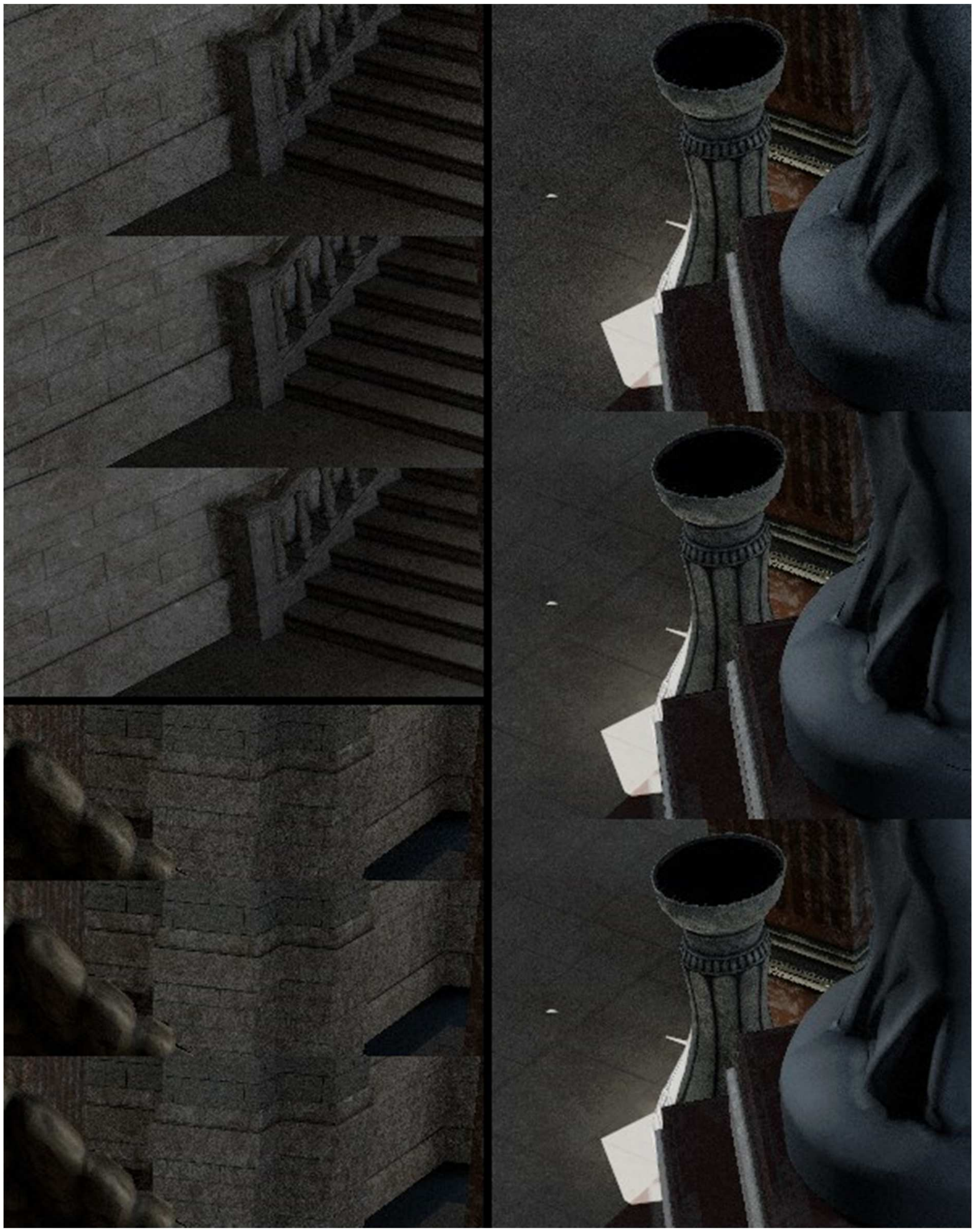

Image 6.2 - For each of the images: - top uses: random sampling, middle: uniform sampling, bottom: correlated multi-jittered sampling

7. Summary

7.1. Generated images

Image 7.1 - Without miss shader using skybox color, scene becomes darker and overall contrast increases

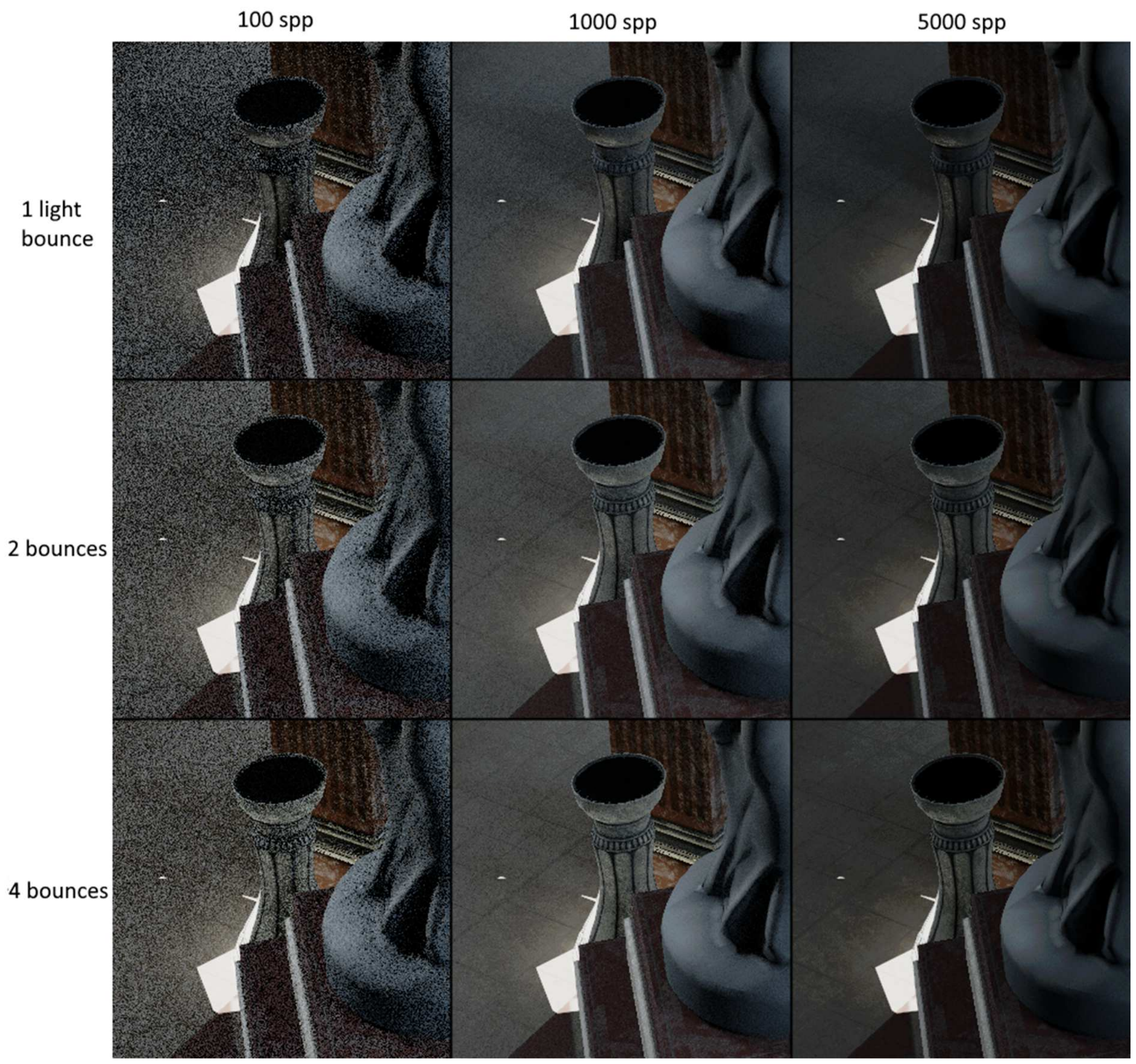

Image 7.2 Comparison of image for different number of frames (spp) and path length

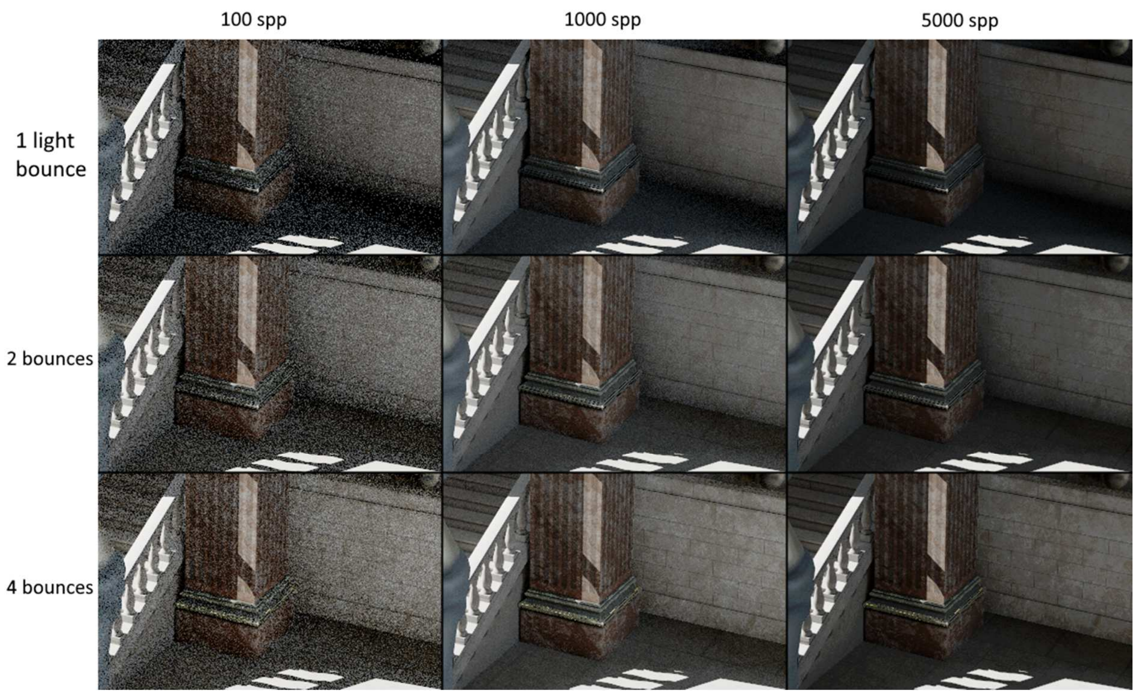

Image 7.3 Comparison of image for different number of frames (spp) and path length

7.2. Problems

The only existing problem is too bright textures for increasing number of accumulated frames. It is especially visible, when number of frames exceeds 2000 and path length is greater than 3.

7.3. Future improvements

Originally this paper was my CS Bachelor thesis (that I translated to English). Even though I finished most of it a few months before the deadline, I decided not to over engineer it and focus on other stuff. Therefore this engine and whole thesis was accomplished in 3 months, including learning DX 12 and DXR from scratch and doing research on ray tracing methods and theory.
There is a lot of room for improvement, especially in DX12 base code. I would recommend https://github.com/TheRealMJP/DXRPathTracer - Matt Pettineo's path tracer offers slightly different options but code is much cleaner.

7.3.1. Denoiser

Denoiser might increase iteration speed of artists working with this engine. It does not improve convergence but it would improve temporal results. Therefore, artists might decide faster if generated lighting is satisfying and bake it in full project. Because *progressive path tracer* does not allow camera movement, only temporal components are needed in denoiser.

7.3.2. Antialiasing

Using even FXAA will improve image quality. However because there is a lot of time available per frame, using methods of higher quality like MSAA is a better choice. Interesting MSAA implementation seems to be one presented by Matt Pettineo in 2015 who worked on **The Order: 1886** [57]. There is GitHub repository [58] with full implementation of that algorithm, created by its author.

7.3.3. Soft shadows

Only hard shadows are implemented in our engine. Soft shadows provide a more realistic look to the scene.

7.3.4. More efficient sampling methods

Correlated multi-jittered sampling [29] proved to be a good enough algorithm. However, using more effective methods might improve convergence rate. We are mainly interested in *progressive multi-jittered sampling* [56] mentioned in this thesis.

7.3.5. Translucent materials

All of commercial engines can use translucency and transparency. It is industry standard and it might improve attractiveness of our path tracer.

Resources

[1] https://docs.microsoft.com/en-us/windows/win32/direct3d12/what-is-directx-12-
[2] https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/
[3] https://devblogs.microsoft.com/directx/directx-raytracing-and-the-windows-10-october-2018-update/
[4] https://devblogs.microsoft.com/directx/variable-rate-shading-a-scalpel-in-a-world-of-sledgehammers/
[5] https://devblogs.microsoft.com/directx/announcing-new-directx-12-features/
[6] https://docs.microsoft.com/en-us/windows/win32/direct3d12/direct3d-11-on-12
[7] https://www.khronos.org/blog/ray-tracing-in-vulkan
[8] https://www.khronos.org/
[9] https://developer.nvidia.com/vulkan-turing
[10] Martin Mittring 2007 - *Finding next gen: CryEngine 2* -
https://www.semanticscholar.org/paper/Finding-next-gen%3A-CryEngine-2-Mittring/fe2cf22b09709ef2a27768dc2b7693c07c02c69d
[11] Louis Bavoil et al 2008 - *Image-space horizon-based ambient occlusion* -
https://www.researchgate.net/publication/215506032_Image-space_horizon-based_ambient_occlusion
[12] Pooria Ghavamian 2019 - *Real-time Raytracing and Screen-space Ambient Occlusion*
http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1337203&dswid=3596
[13] https://developer.nvidia.com/vxao-voxel-ambient-occlusion
[14] https://www.gdcvault.com/play/1026159/Exploring-the-Ray-Traced-Future
[15] https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9985-exploring-ray-traced-future-in-metro-exodus.pdf
[16]
https://github.com/NVIDIAGameWorks/GettingStartedWithRTXRayTracing/blob/master/11-OneShadowRayPerPixel/Data/Tutorial11/diffusePlus1ShadowUtils.hlsli
[17] Diede Apers, Petter Edblom, Charles de Rousiers, and Sébastien Hillaire (Electronic Arts) - *Interactive Light Map and Irradiance Volume Preview in Frostbite*
[18] Schied et al. 2017 - *Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination*
https://research.nvidia.com/publication/2017-07_Spatiotemporal-Variance-Guided-Filtering%3A
[19] Zwicker et. al 2015 - *Recent Advances in Adaptive Sampling and Reconstructionfor Monte Carlo Rendering*
[20] https://developer.nvidia.com/rtx/raytracing/dxr/DX12-Raytracing-tutorial-Part-1
[21] https://developer.nvidia.com/blog/rtx-best-practices/
[22] Christiaan Gribble - *Multi-Hit Ray Tracing in DXR*
[23] Eric Haines, Tomas Akenine-Möller - *Ray Tracing Gems*
[24] Johan Köhler (Treyarch) - *Practical Order Independent Transparency*
[25] [Whitted1980], Whitted et al - *An Improved Illumination Model for Shaded Display*
[26] [Green2003] Robin Green - *Spherical Harmonic Lighting: The Gritty Details*
[27] [Kajiya1986] James T. Kajira - *The Rendering Equation*
[28] [Driscoll2009] Rory Driscoll - *Energy conservation in games*
http://www.rorydriscoll.com/2009/01/25/energy-conservation-in-games/
[29] [Kensler2013] Andrew Kensler - *Correlated Multi-Jittered Sampling*

[30] [CT1982] Robert L. Cook, Kenneth E. Torrance - *A Reflectance Model for Computer Graphics* https://dl.acm.org/doi/10.1145/357290.357293
[31] [Walter2007] Bruce Walter et al - *Microfacet Models for Refraction through Rough Surfaces* https://www.cs.cornell.edu/~srm/publications/EGSR07-btdf.pdf
[32] [Schlick94] Christophe Schlick - *An Inexpensive BRDF Model for Physically-Based Rendering*
https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.50.2297&rep=rep1&type=pdf
[33] Chris Wyman - http://cwyman.org/code/dxrTutors/tutors/Tutor14/tutorial14.md.html
[34] Joe Schutte - *Importance Sampling techniques for GGX with Smith Masking-Shadowing: Part 1* https://schuttejoe.github.io/post/ggximportancesamplingpart1/
[35] [Smith1967] B. Smith - *Geometrical shadowing of a random rough surface*
https://ieeexplore.ieee.org/document/1138991/
[36] [Blinn1977] James. F. Blinn - *Models of light reflection for computer synthesised pictures* https://www.microsoft.com/en-us/research/wp-content/uploads/1977/01/p192-blinn.pdf
[37] Cao Jaiyin - *Sampling Microfacet BRDF*
https://agraphicsguy.wordpress.com/2015/11/01/sampling-microfacet-brdf/
[38] Bruce Walter - *Notes on the Ward BRDF*
https://www.graphics.cornell.edu/~bjw/wardnotes.pdf
[39] Eric Heitz - *Understanding the Masking-Shadowing Functionin Microfacet-Based BRDFs*
http://jcgt.org/published/0003/02/03/paper.pdf
[40] Eric Heitz, Eugene d'Eon - *Importance Sampling Microfacet-Based BSDFs using the Distribution of Visible Normals* https://hal.inria.fr/hal-00996995v1/document
[41] Eric Heitz - *A Simpler and Exact Sampling Routine for the GGXDistribution of Visible Normals* https://hal.archives-ouvertes.fr/hal-01509746/document
[42] John Hable (Naughty Dog) - *Uncharted 2: HDR Lighting*
https://www.slideshare.net/ozlael/hable-john-uncharted2-hdr-lighting
[43] Krzysztof Narkowicz - *ACES Filmic Tone Mapping Curve*
https://knarkowicz.wordpress.com/2016/01/06/aces-filmic-tone-mapping-curve/
[44] Academy Color Encoding System Developer Resources - *aces-dev*
https://github.com/ampas/aces-dev
[45] Michael Stokes et al. - *A Standard Default Color Space for the Internet - sRGB*
https://www.w3.org/Graphics/Color/sRGB
[46] [Turquin2019] Emmanuel Turquin - *Practical multiple scattering compensation for microfacet models* - https://blog.selfshadow.com/publications/turquin/ms_comp_final.pdf
[47] Krzysztof Narkowicz - *Comparing Microfacet Multiple Scattering with Real-Life Measurements* https://knarkowicz.wordpress.com/2019/08/11/comparing-microfacet-multiple-scattering-with-real-life-measurements/
[48] [Li2018] - *Al6061 surface roughness and optical reflectance when machined by single point diamond turning at a low feed rate -*
https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0195083
[49] Brian Karis [Karis2013] - *Real Shading in Unreal Engine 4*
https://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_slides.pdf
[50] Sebastien Lagarde et al - Moving Frostbite to Physically Based Rendering 3.0
https://seblagarde.files.wordpress.com/2015/07/course_notes_moving_frostbite_to_pbr_v32.pdf

[51] J. H. Halton - *Algorithm 247: Radical-inverse quasi-random point sequence* https://dl.acm.org/doi/10.1145/355588.365104
[52] J. M. Hammersley - *Monte Carlo Methods* https://link.springer.com/book/10.1007%2F978-94-009-5819-7
[53] [Dammertz2012] Holger Dammertz - *Hammersley Points on the Hemisphere* http://holger.dammertz.org/stuff/notes_HammersleyOnHemisphere.html
[54] Henry S. Warren - *Hacker's Delight*
[55] K. Chiu et al - *Multi-jittered sampling. (Graphics Gems IV)*
[56] Per Christensen et al - *Progressive Multi-Jittered Sample Sequences* https://graphics.pixar.com/library/ProgressiveMultiJitteredSampling/paper.pdf
[57] Matt Pettineo (Ready at Dawn) - *Rendering The Alternate History of The Order: 1886* from SIGGRAPH 2015
[58] Matt Pettineo (Ready at Dawn) - implementation of MSSA from presentation *Rendering The Alternate History of The Order: 1886* https://github.com/TheRealMJP/MSAAFilter
[59] Joe Schutte - *Importance Sampling techniques for GGX with Smith Masking-Shadowing: Part 2* https://schuttejoe.github.io/post/ggximportancesamplingpart2/