Universiteit Utrecht

Assignment 3

# Path Tracing & Extensions

November 11, 2018

*Students:*
Mathijs Molenaar
5958970

Claudio Freda
6030769

*Teacher:*
Jacco Bikker

*Course:*
Advanced Graphics

*Course code:*
INFOMAGR

> We do these things not because they are easy, but because we thought they were going to be easy.
>
> *Programmer's Proverb*

**Abstract**

We upgrade our BVH-powered raytracer by turning it into a path tracer, implementing various variance reduction techniques, and adding support for physically based materials such as metals, dielectrics, and glass, and for a physically-based camera. Finally, we update our GPU pipeline with streaming path tracing and add some final comments on the technology we used.

# 1 Introduction

In our first report we described how we implemented a basic ray tracer in OpenCL. It supported spheres, planes and arbitrary triangle models. To improve performance we added BVH support in the second assignment. This includes support for animations with a top-level BVH, binned BVH construction and refitting. Gamma correction was also present in the previous work since it is vital element in creating a realistic image.

In this report we will discuss how we converted the previously described ray tracer into an OpenCL path tracer. This path tracer supports multiple forms of variance reduction to increase the speed of convergence. To take full advantage of the path tracing capabilities, we added two microfacet shading models. Finally, we changed the camera model to more closely match a real digital camera.

# 2 Path Tracing

The starting point was the Whitted style ray tracer that we ended up with after last assignment. Such a Whitted style ray tracer traces rays from the camera into the scene. For each intersection, the rendering equation is evaluated by summing the contribution of each light. This results in direct lighting from either point or directional lights, both of which do not exist in the real world.

Although a Whitted style tracer does deliver pixel perfect shadows, it does not generate very realistic images. The main cause of this is the lack of indirect lighting or global illumination. To alleviate this problem, the ray tracer had converted into a path tracer.

A path tracer tries to more fully model light transport. Not all light rays emitted by a light source will reach the camera in one bounce. This does not mean that they should not contribute to the resulting image though. In the real world, light rays may bounce multiple times before reaching the sensor of a camera or the retina of the eye. A path tracer models this behavior by supporting multiple bounces and evaluating the full integral over the rendering equation.

Applying a complex numerical integration scheme to evaluate the rendering equation is not a very practical solution. Sampling the function requires a new ray to be sent, which may also hit a surface for which we need to integrate once more. This method would quickly explode the ray count and require a lot of computation time to generate a single image. This is why path tracers use Monte Carlo integration. At each intersection, we sample the rendering equation once and assume that the sampled value represents the whole integral.

The Whitted Style ray tracer supported point and directional lights. Both of these types of lights have an infinitely small area. This means that it is impossible for a “camera“ ray to hit a light source. We therefore replaced them with emissive triangles. These lights may be read from a 3D model file such as the famous Cornell box. With these changes in place we are able to generate our first images

with indirect lighting.

# 3 Variance Reduction

In a path tracer, using a naive sampling strategy will yield an image that converges to an accurate result. But the result will converge very slowly if no relevant techniques for variance reduction are applied. Successfully implementing variance reduction techniques helps to obtain accurate images in a reasonable time, and reduces the strain on the random number generator to produce completely uncorrelated results.

The objective of a variance reduction techniques is to increase the number of rays that are useful, while weighting the result of those rays so the final result remains the same. In this section we will discuss the variance reduction techniques that we implemented.

## 3.1 Next Event Estimation

Let us consider the hemisphere surrounding the contact point of a ray. The naive method samples a uniformly random point on this hemisphere and sends a ray through that point.

When random sampling the hemisphere, most rays end up returning a small amount of energy. We can clearly see that direct illumination causes the largest amount of the energy gathered by rays. Next Event Estimation works by directly sampling direct illumination for each intersection.

The basic principle is to divide incoming light between direct and indirect light. Whenever a ray intersects with a surface, we sample both a *random walk ray* in the hemisphere and a *shadow ray* towards a point sampled uniformly on the area of a random light source.

The shadow ray is checked for visibility between the intersection point and the random sample on the selected light source; if there are no obstacles, the light's energy is returned, weighted by the solid angle that the light casts on the hemisphere.

By using this method as presented, direct light would be actually sampled twice, once with NEE and once when a ray hits a light. To correct this, if a random walk ray hits a light source, it will return the color black instead of the color of the light source. By doing this we ensure that the total amount of energy stays constant.

NEE yields the most improvement out of all the sampling methods, which is why we decided to implement it first.

## 3.2 Importance Sampling

Instead of sampling the hemisphere uniformly, a good way to reduce variance is to sample according to a non-uniform probability distribution function (PDF). The goal of the PDF is to correctly approximate the rays that will return more energy, if this happens more rays will be sampled towards high-importance areas and the variance will be reduced. This is called importance sampling.

The final result of each ray needs to be divided by the value of the PDF for that ray, to offset the bias caused by the probability of that ray being generated. In practice, if the PDF matches some part of the BRDF or the cosine term, this can be achieved through simplification.

So if for example the Monte Carlo approximation of the light collected by hemisphere with uniform sampling would be:

When it is sampled according to a PDF it becomes:

For a simple diffuse material, the best way to do importance sampling is to sample according to the cosine of the angle between the incidence vector and the surface normal. The PDF in this case would be $o \cdot n \div \pi$, where $o$ is the output vector and $n$ is the normal.

Listing 1: Generating Cosine Weighted Directions

```
float r0 = randRandomU01(randomStream);
float r1 = randRandomU01(randomStream);
float r = sqrt(r0);
float theta = 2 * PI * r1;
float x = r * cos(theta);
float y = r * sin(theta);
float3 sample = (float3)(x, y, sqrt(1 - r0));
float3 tangent = normalize(cross(normal, edge1));
float3 bitangent = cross(normal, tangent);
float3 orientedSample = sample.x * tangent + sample.y * bitangent + sample.z *
    normal;
return normalize(orientedSample);
```

We have implemented more advanced shading models, described later, and for each we will provide the proper sampling strategy when mentioned.

### 3.2.1 Importance Sampling for NEE lights

In the NEE description we mentioned that during the NEE process, a random light was selected for sampling. As the lights' energy transfer is dependent on the solid angle cast on the hemisphere, one can see how sampling the lights proportionally to such solid angle might yield better results.

We decided to try this approach by precalculating the triangle areas, and selecting a light randomly, but using its solid angle as a weight. The energy transfer would then be multiplied by the *average* solid angle instead of the solid angle of the specific light.

This method ultimately resulted in being quite slow, due to the necessity of accessing every light to evaluate the solid angle that it casts on the hemisphere. This prevents the algorithm from scaling well with a high number of lights. We

still think it might improve variance in some scenes, so we included it, but it is disabled by default.

## 3.3 Multiple Importance Sampling

Multiple Importance Sampling is a way to blend multiple sampling strategies in a way that exploits both strategies' strengths and reduces their weaknesses. In our case, considering two sampling strategies is enough.

If we have two PDFs, $p_1(i, n, o)$ and $p_2(i, n, o)$, with $n$ being the surface normal, $i$ the incidence vector and $o$ the direction of the output ray, we can weight each of the two sampling strategies based on the relative value of their PDF. Thus we end up with two weight functions. This is called the balance heuristic.

$$w_1(i, n, o) = \frac{p_1(i, n, o)}{p_1(i, n, o) + p_2(i, n, o)} \tag{1}$$

$$w_2(i, n, o) = \frac{p_2(i, n, o)}{p_1(i, n, o) + p_2(i, n, o)} \tag{2}$$

Using both sampling strategies at the same time, but multiplying by these weights will ensure that, when one's PDF is higher, it dominates over the other, while maintaining our simulation unbiased. The weight functions can be often simplified because when doing IS we are often dividing by the PDF already. This in turn results in just dividing by $p_1(i, n, o) + p_2(i, n, o)$ instead.

An important point to note is that the sampling strategies need not be calculated at the same time or at the same point in the algorithm. If, in the end, they all return energy to the same pixel, the Monte Carlo approach will ensure that our integral is correct.

We are going to use MIS to balance direct lighting between NEE and importance sampling of the hemisphere. NEE itself is a form of limited importance sampling for direct light, where the PDF is a constant of $1/\text{solid angle}$ at the solid angle of the light, and $0$ otherwise.

Instead of leaving the direct light calculation exclusively to NEE, we would like to use MIS to blend between the two. To do so we consider two PDFs, $p_{NEE} = 1/\text{solid angle}$ and $p_{IS}(i, n, o)$ being whatever PDF are we using for importance sampling of the hemisphere. We will apply MIS only when a ray effectively hits a light, so we don't have to consider the case where $p_{NEE} = 0$. We will thus intervene in two places in our algorithm:

1. When doing NEE, instead of simply multiplying the light energy by the solid angle, we divide by $1/\text{solid angle} + p_{IS}(i, n, o)$. The vector from the intersection point to the light is used as the $o$ vector.

2. When a random walk ray directly hits a light, instead of discarding the ray, we return the light's colour weighted by $\frac{p_{IS}(i,n,o)}{1/\text{solid angle}+p_{IS}(i,n,o)}$. At this point

in the algorithm, the $p_{IS}$ of the original distribution that generated the ray would have already been integrated into the ray's multiplier, and thus discarded. To be able to make use of it later, we just add a field into our `RayData` struct to hold the $p_{IS}$ value.

# 4 Shading Models

An important improvement to our path tracer is the support for more material models. Originally, the ray tracer only supported Lambertian diffuse and perfect reflections and refractions. We have now added support for two microfacet models that aid in the rendering of metals, dielectrics and some translucent dielectrics.

How light is reflected off a surface is defined by the Bidirectional Reflectance Distribution Function (BRDF). Additionally, other BxDF functions exists such as Bidirectional Scattering-Surface Reflectance Distribution Functions which are able to describe Sub Surface Scattering. Because we lost too much time with debugging, we had no time to implement sub surface scattering. What we do support is a Bidirectional Transmittance Distribution Function (BTDF). This function describes how light scatters at when hitting a surface. For all of these functions it is important that they obey the Physically Based Rendering (PBR) rules:

1. It should be positive: $f_r(\omega_o, \omega_i) \geq 0$
2. Helmholtz reciprocity should be obeyed: $f_r(\omega_o, \omega_i) = f_r(\omega_i, \omega_o)$
3. It should be energy conservative: $\int_\Omega f_r(\omega_o, \omega_i) cos(\theta_o) d\omega_o \leq 1$

The first rule prevents rays from carrying negative light/energy. The second rule is vital to ray and path tracing. To get a good rate of convergence, we send rays from the camera into the scene, instead of tracing rays from the lights. This is only possible because of the Helmholtz reciprocity which dictates that it does not matter in which direction we trace. Finally, energy conservation is vital for proper indirect lighting, but not very relevant for direct lighting. When the BRDF exceeds 1, the point that we are shading starts emitting light. Since the BRDF is only supposed to model reflection, this is incorrect. With path tracing this problem "explodes" because a ray might bounce many times before reaching a light source. If it increased intensity with each bounce, it will return way more light than it is supposed to.

## 4.1 Microfacet Reflection

Probably the most used BRDF of all time is Blinn-Phong. It has been used in video games for a very long time and has been part of the fixed function hardware since the early days of graphics cards. The Blinn-Phong BRDF respects the first two

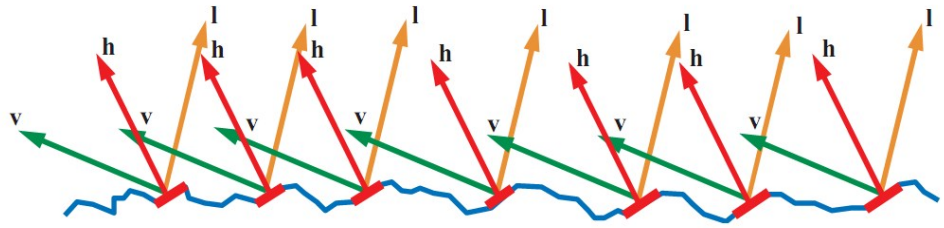


Figure 1: Microfacet model

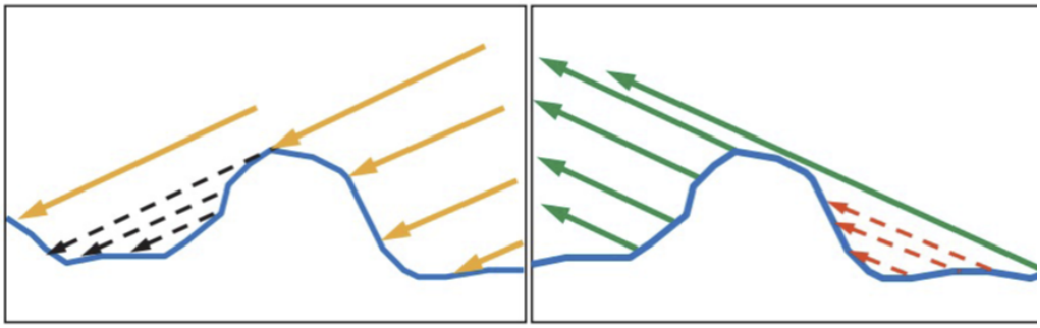

Figure 2: Microfacet shadowing

rules of a PBR BRDF, but not energy conservation. Now that games are starting to support indirect lighting, the gaming industry is making a switch to microfacet BRDFs.

The idea behind microfacets is that materials that are flat at a macroscopic (visible to the human eye) level, might not be flat at a microscopic level. The model assumes that at that microscopic level, the material has bumps, called facets (figure 1). These facets reflect light according to the Fresnel law. The amount of facets oriented in a certain direction is described by the Normal Distribution Function (NDF). Finally, facets may block incoming light or light that is reflected from other facets (2). The geometry function models the amount of light being lost to this shadowing.

All 3 functions are combined in the microfacet BRDF function (equation 3). The microfacet equation is plug-and-play, as one can freely mix and match between different Fresnel approximation functions, NDFs and geometry functions. We have chosen to use the popular Schlick approximation [12] for the Fresnel curve. We use the GGX NDF with the height corrected Smith geometry function as presented by Dice [9].

$$f_r(o,i) = \frac{D(h)F(i,h)G(o,i,h)}{4(n \cdot o)(n \cdot i)} \tag{3}$$

With this BRDF we can now generate pretty images of metallic surfaces. The reflected light gets its colour from the Fresnel curve being different for the RGB

channels.

## 4.2 Importance Sampling and PBR

We would like to select a PDF for Importance Sampling that is more appropriate than the cosine-weighted PDF. Especially for low-roughness materials, the cosine-weighted PDF really inadequately approximates the actual BRDF, which has the shape of a lobe.

To do so we use the NDF, also called the D function, as our PDF. To generate important directions according to the GGX NDF we use the following algorithm, created from the information found in the blog article *Notes on Importance Sampling* [5];

Listing 2: Generating GGX Weighted Halfway Directions

```
float r0 = randRandomU01(randomStream);
float phi = 2.0f * PI * r0;
float r1 = randRandomU01(randomStream);
float theta = acos(sqrt((1.0f - r1) / ((alpha*alpha - 1.0f) * r1 + 1.0f)));
float x = cos(phi) * cos(PI/2 - theta);
float y = sin(phi) * cos(PI/2 - theta);
float z = sin(PI/2 - theta);
float3 sample = (float3)(x,y,z);
float3 tangent = normalize(cross(normal, (float3)(1.0f, 0.0f, 0.0f)));
float3 bitangent = cross(normal, tangent);
// Transform hemisphere to normal of the surface (of the static model)
// [tangent, bitangent, normal]
float3 orientedSample = sample.x * tangent + sample.y * bitangent + sample.z *
    normal;
return orientedSample;
```

This returns a halfway vector, the reflection vector is retrieved by simply doing a reflection using the halfway vector as the surface normal.

This also allows us do to one simplification: we will remove the D term from our BRDF, as it is already accounted for by the probability distribution of the rays.

### 4.2.1 GGX simplification

The Frostbite paper [9] includes one useful magic trick: instead of the G distribution, they use what they call a *Vis* distribution. This function is identical to G except that it already includes the division by $4(n \cdot o)(n \cdot i)$.

$$\text{Vis}(n, o, i, h) = \frac{G(o, i, h)}{4(n \cdot o)(n \cdot i)} \tag{4}$$

The *Vis* function includes a lot of simplifications and most importantly: it removes useless divisions.

Listing 3: GGX Correlated Vis function

```
float V_SmithGGXCorrelated(float NdotL, float NdotV, float alphaG)
```

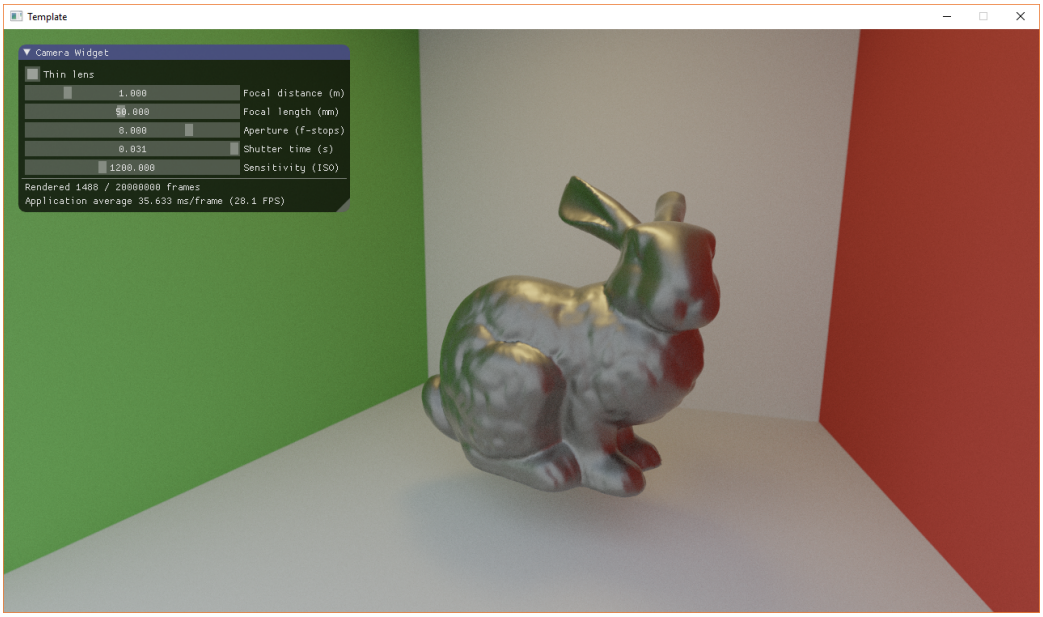


Figure 3: Rendering of a rough metal Stanford bunny.

```
{
// V_SmithGGXCorrelated = G_SmithGGXCorrelated / (4.0f * NdotL * NdotV);
float aG2 = alphaG * alphaG;
float Lambda_GGXV = NdotL * sqrt ((- NdotV * aG2 + NdotV ) * NdotV + aG2 );
float Lambda_GGXL = NdotV * sqrt ((- NdotL * aG2 + NdotL ) * NdotL + aG2 );
return 0.5f / ( Lambda_GGXV + Lambda_GGXL );
}
```

The final BRDF for metals, having removed the D term because of IS weighting, now looks like this:

$$f_r(o,i) = F(i,h)\text{Vis}(n,o,i,h) \tag{5}$$

We are placing such an importance on the simplification of divisions because we noticed through numerous tests that divisions, especially by cosine terms at grazing angles where such terms are very small, tend to disproportionally blow up floating point arithmetic errors. This sometimes leads to unwanted white or black dots that take a lot of samples to converge back to an acceptable value.

From a bias-consistency (see [2]) standpoint, we can say the simplification of expressions increases the *consistency* of the algorithm while preserving its *lack of bias*.

## 4.3 Dielectrics

We would like to also be able to model dielectric surfaces. The Fresnel term in the BRDF equation determines how much light is reflected, and the remaining amount of light is refracted into the material.

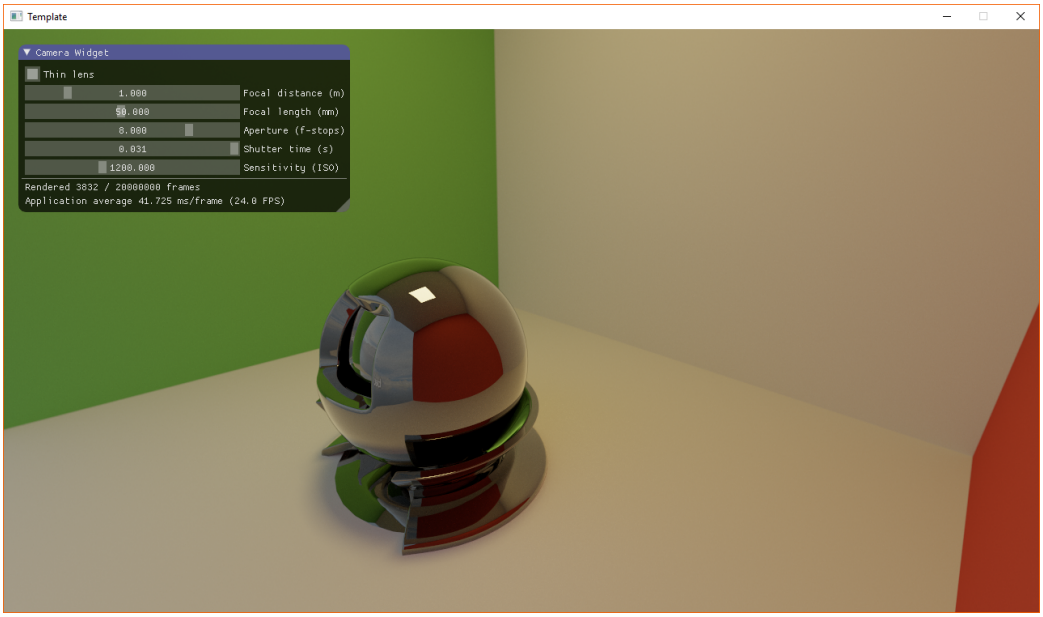


Figure 4: Rendering of a maximum smoothness (mirror) metal sphere.

Metals tend to absorb all that light almost immediately which is why using only the BRDF suffices. However, at non-grazing angles the dielectric only reflects a small part of the light. The refracted light is scattered within the material, which is called Subsurface Scattering (SSS). For some materials this happens just beneath the surface and so the light will exit at almost the same point. If this distance is much smaller than the size of a pixel, we can assume that it exists at the point of entrance. With this assumption we can model the Subsurface Scattering locally using a diffuse shading model. We use the normalized Disney diffuse as suggested by Dice in "Moving Frostbite to PBR" [9]. With this approach we are able to model some dielectric materials such as plastics.

The BRDF for dielectric materials is thus the following.

$$f_s(o,i) = \frac{D(h)F(i,h)G(o,i,h)}{4(n\cdot o)(n\cdot i)} + (1 - F(i,n))\text{Disney}(n,o,i,h) \tag{6}$$

## 4.4 Monte Carlo Fresnel Blending for IS in Dielectrics

When considering dielectrics, we would like to use two different distributions for the diffuse and specular components of the BRDF, being that especially at low roughness, the GGX NDF does not approximate the diffuse component of the BRDF very well, resulting in a lot of unwanted variance.

What we did is what we call *Monte Carlo blending*. Instead of having a mixed BRDF, we choose between the diffuse and specular BRDFs at random based on the Fresnel component. For dielectrics, the Fresnel component is equal for all color channels, so it's a single value.

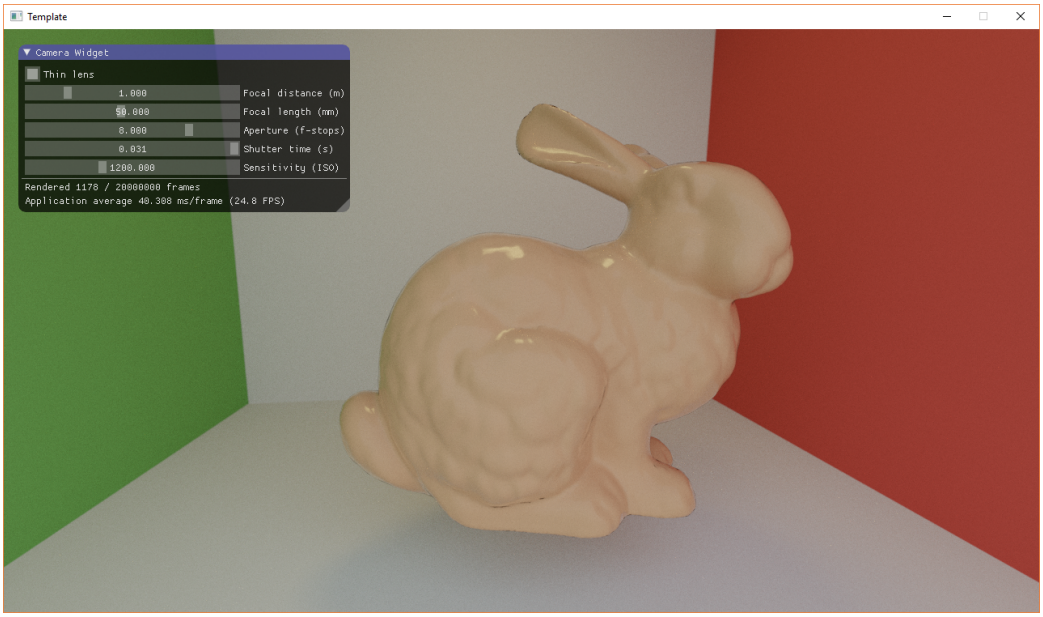


Figure 5: Rendering of a smooth ceramic Stanford bunny.

Based on the chosen BRDF, we decide on the sampling strategy to use. For diffuse, we use the standard cosine-weighted distribution, while for specular we use GGX weighted distribution.

Of course now the Fresnel terms $F$ for specular and $1 - F$ for diffuse are included in the sampling strategy, so for dielectrics we don't multiply by these factors anymore. One could say that we divide by these factors and simplify the expression.

This also means that the two BRDFs become rather simplified in their final form. As a reminder, the normal distribution and Fresnel terms are already represented in the sampling strategies we used.

$$f_{r,\text{Specular}}(o,i) = \text{Vis}(n,o,i,h) \tag{7}$$
$$f_{r,\text{Diffuse}}(o,i) = \text{Disney}(n,o,i,h) \tag{8}$$

We noticed a significant decrease in the variance of low-roughess Dielectrics after implementing this technique.

### 4.4.1 Monte Carlo Blending and MIS

Earlier in the report, we described the MIS technique for variance reduction. Changing MIS to support this blend is not trivial, and we modified MIS to be able to combine the two techniques.

1. On the NEE side, we made sure to do the same selection between the two

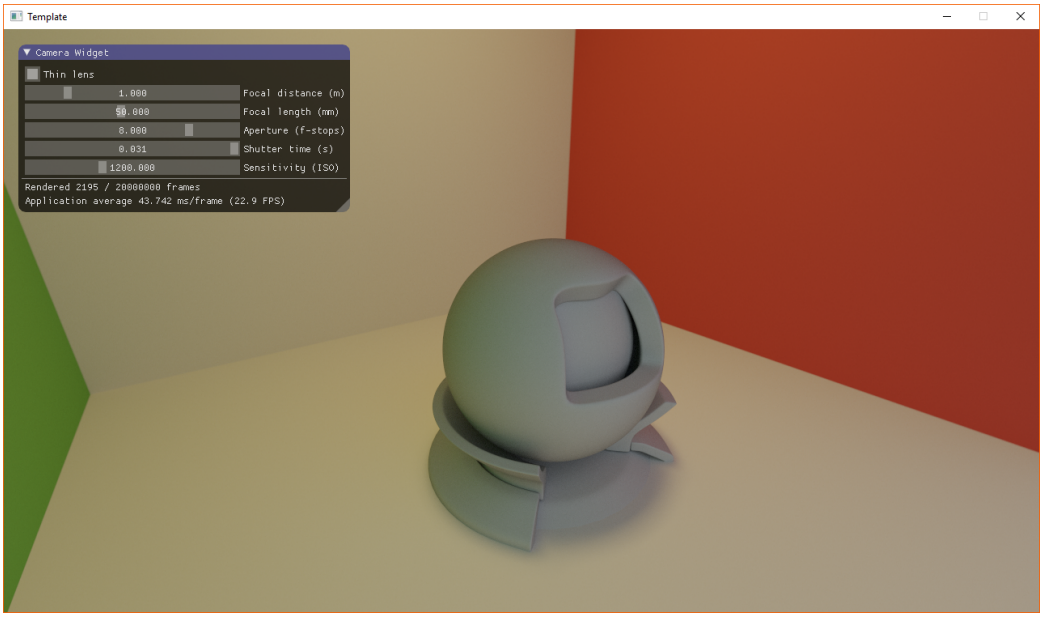


Figure 6: Rendering of a rough rubber sphere.

distributions with the same probability to select our $p_{IS}$. The Monte Carlo approach ensures that this will converge to the correct value.

2. On the random walk side, nothing has to be changed because the value of the selected PDF has already been saved and we don't need to know the actual expression.

As a sidenote, we believe a similar method might be used to combine the Resampled Importance Sampling (RIS) technique, which is affine in a way to our Monte Carlo blending, with MIS. However, we did not have the time to implement RIS and test whether this was true.

## 4.5 Microfacet Refraction

We have use a similar microfacet model to represent refractive materials in our path tracer. We implemented the techniques presented in the paper "Microfacet models for refraction through rough surfaces" [13].

This paper presents a model for refractive materials using microfacet theory, although unfortunately it does not include a modeling of subsurface scattering, so not every kind of dielectric can be represented. However, it is pretty good for glass and glass-like materials.

We don't want to get too much in detail about its contents, because it would go out of the scope of this report. We will summarize the main points.

Two BRDFs are presented, a reflective function for the specular part and a transfer function for the refractive part. The reflective BRSF is exactly the one

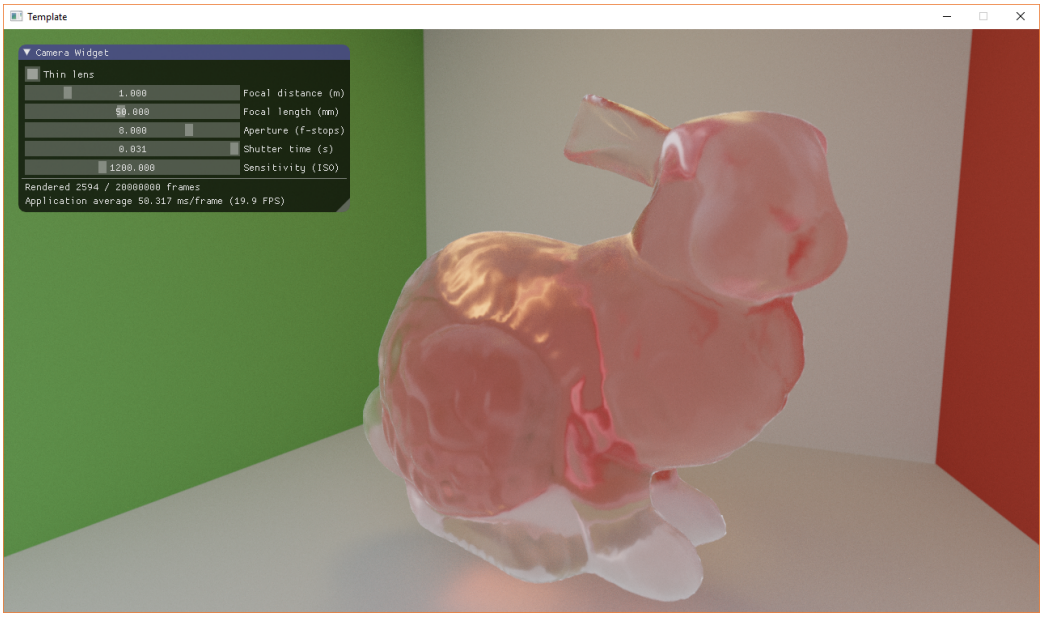


Figure 7: Rendering of a rough red glass bunny.

we presented before for metals (equation 3), while the refractive BRTF is shown below.

$$f_{r,\text{Refractive}}(i,o,n) = \frac{|i\cdot h|\,|o\cdot h|}{|i\cdot n|\,|o\cdot n|}\frac{\eta_o^2(1-F(i,h))G(i,o,h)D(h)}{(\eta_i\,|i\cdot h| + \eta_o\,|o\cdot h|)^2} \tag{9}$$

While this function looks very complicated, much of its complexity is due to it considering the $o$ vector that results from applying Snell's law of refraction to the $h$ halfway and $i$ incidence vectors.

The authors proceed to describe different choices for the F, D and G functions, and explain their importance sampling strategy when choosing each function. Importance sampling is built into the technique proposed and allows to make several assumptions that allow the technique to be especially effective.

The important thing to take away from this paper is that importance sampling allows us to consider, instead of the reflection/refraction vector $o$, the $h$ halfway vector as the parameter to the PDF $p(h)$. This allows us to skip considering Snell's law until the very end, simplify much of the complexity, and consider the sampling of both refraction and reflection at the same time.

The $p(h)$ is chosen with the same strategy we used with metals: $p(h) = D(h)$, simply. The problem of generating a direction that follows such a PDF is a simple one to solve and is tackled in [13] (we also already did it for GGX a few sections ago).

What weight should be applied, then, to keep the result unbiased? The authors present a simple expression that is equal for both refraction and reflection.

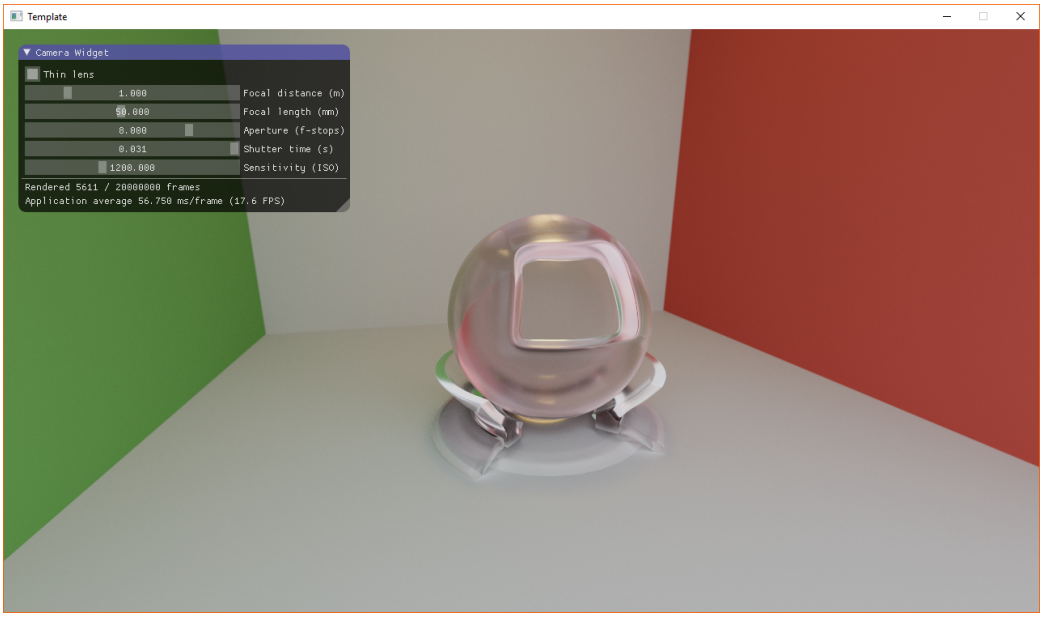


Figure 8: Rendering of a rough glass ball.

$$\text{weight}(o) = \frac{|i \cdot h| \, G(i, o, m)}{|i \cdot n| \, |h \cdot n|} \tag{10}$$

The vector $o$ will be simply the reflection of $i$ the normal $h$ for the reflective part, while for the refractive part will be the result of the application of Snell's law on $i$ and $h$.

This will be chosen at random based on the Fresnel factor with the same Monte Carlo blending approach that we used for dielectric materials. The Fresnel factor is thus also integrated into the sampling strategy.

The result is that, after applying the aforementioned weighting function, we can just return the full ray energy, because all of the other BRDF components are already included in the sampling strategy.

We would also like to note that we added Beer's law into the model, allowing for the rendering of tinted glass.

## 4.6 A Note on the choice of BRDFs

At the beginning, we decided to use GGX because we had it already implemented for specular. Unfortunately, we noted that GGX has some properties that make it very bad for glass, and decided to replace it with Beckmann in the end.

The property we are talking about is that GGX approaches zero much slower than Beckmann, and causes an inordinate amount of outlier directions. With *inordinate amount* we mean that, while such directions are very rare, on thousands of rays they tend to happen quite often.

When such directions are generated, they tend to have very high weights and show visible white dots on the image that take a long time to converge back to a correct value.

With opaque materials this was not very noticeable because reflections would be blended with either black for metal or a diffuse BRDF for dieletrics. With refractive materials, however, the direction is always sampled fully, either for a reflected or a refracted ray.

While this didn't make the method unbiased, it made it quite inconsistent. A path tracer does not behave very well when very high weights are generated.

Just switching to the Beckmann distribution, which goes to zero very fast when distant from its centre, almost completely solved this problem. We would like to recommend the Beckmann distribution for work with path tracing glass-like materials, unless one has a lot of time and resources available to wait for convergence.

We would also like to include the algorithm for generating important directions according to the Beckmann distribution, for completeness.

Listing 4: Generating Beckmann Weighted Halfway Directions

```
float r0 = randRandomU01(randomStream);
float r1 = randRandomU01(randomStream);
float phi = 2.0f * PI * r0;
float theta = atan(-alpha * alpha * log1p(-r1));

float x = cos(phi) * cos(PI/2 - theta);
float y = sin(phi) * cos(PI/2 - theta);
float z = sin(PI/2 - theta);

float3 sample = (float3)(x,y,z);

//float3 normal = normalize(cross(edge1, edge2));
float3 tangent = normalize(cross(normal, (float3)(1.0f, 0.0f, 0.0f)));
float3 bitangent = cross(normal, tangent);

// Transform hemisphere to normal of the surface (of the static model)
// [tangent, bitangent, normal]
float3 orientedSample = sample.x * tangent + sample.y * bitangent + sample.z *
    normal;
return orientedSample;
```

Regarding opaque materials, after testing both distributions we decided to remain with GGX because of the nice simplification provided by Frostbite's $\text{Vis}_{\text{GGX}}$ distribution, as discussed earlier.

## 4.7 Some Biased Tricks

A recommendation we found in [13] is a trick to reduce weights. It introduces a small bias, so it should not be used if complete lack of bias is the objective, but it severely improves consistency by avoiding high weights.

The trick consists in using a different, slightly larger, roughness value when generating rays than the one used for calculating weights. The recommended value is presented below.

$$\alpha' = (1.2 - 0.2\sqrt{|i \cdot n|})\alpha \tag{11}$$

We would like to note that this has been calculated to work with the Beckmann distribution. It would likely not work at all for GGX.

Another trick we used is to place a pretty large higher limit on the weight values. While this introduced a certain bias, it allowed to ignore outliers, yielding a faster convergence and better-looking pictures.

We also retroactively applied this last trick to our opaque materials.

Both tricks can be easily removed to get an unbiased image, if needed.

# 5 Physically Based Camera

As part of our upgrade to the ray tracer, we improved the camera. Inspired by the frostbite engine, we approximated a real digital camera in our new path tracer. We implemented all the steps that a real digital camera would take to produce the final image, including gamma correction (figure 9). Additionally, we also implemented depth of field using lens sampling.

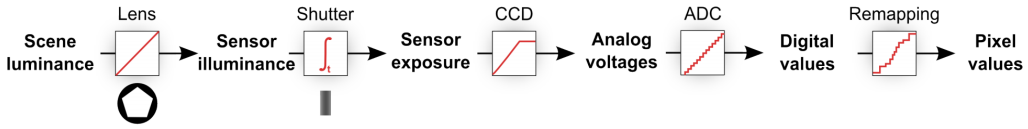


Figure 9: Digital camera pipeline. Courtesy of Dice (course notes Moving Frostbite to PBR).

## 5.1 Depth Of Field

The first extension to the camera was to add a depth of field effect. Depth of field is the blur effect of objects that are out of focus. There are two ways to achieve this in a path tracer. First of, we could add a post process blur effect such as used in practically all video games [4]. The second option, which is only available in a path tracer, is to model the lenses of the camera. We have opted for the second approach since it is more accurate and is a natural fit for path tracing.

Real photo camera's have an array of lenses that transform the rays. There has been some research into realistic camera models for computer generated graphics [8]. The lens configuration however, is something camera constructors are not willing to share. For this reason, and because of time constraints, we have decided to keep our model a bit simpler.

Our camera supports a single thin lens, which is an infinitely thin lens. The most important property of a lens is the focal distance. This is the distance from

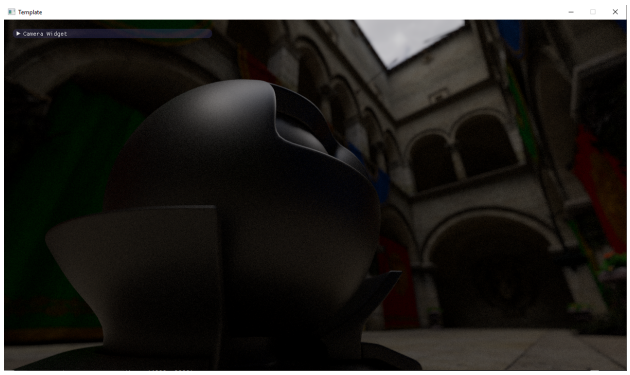

Figure 10: Depth of Field effect in Sponza

the center of the lens to the focal point. The focal point is the point where rays that perpendicularly hit the lens converge. The distance from the lens to the sensor, also called projected distance, may vary however. The object distance is the distance from which rays converge at the projected distance behind the lens. The relation between these three values is described by the thin-lens equation: $\frac{1}{D} + \frac{1}{V} = \frac{1}{F}$.

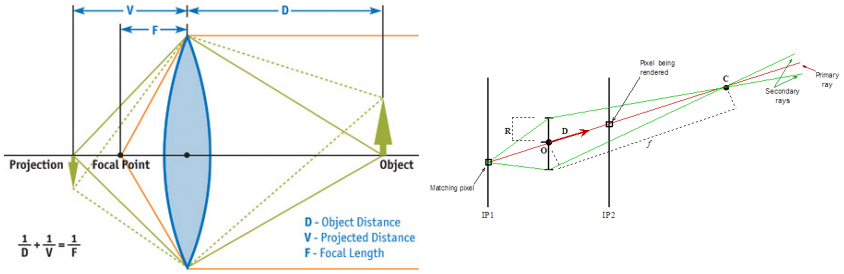


Figure 11: Thin-lens model

We implemented this in our path tracer as suggested by *Depth of Field* [3]. To generate a primary ray, we start of by picking a random point on the lens that is not obstructed by the aperture (more on aperture in section 5.2). Using the thin-lens equation we can calculate the point of convergence for that pixel. The point of convergence lies on the line through the pixel and the centre of the lens, at a distance equal to D (the object distance). The outgoing ray is than simply the ray originating from our lens sample towards the point of convergence

## 5.2 Exposure

In their paper "Moving Frostbite to PBR" [9], Dice describes how they implemented a physically based camera system in the Frostbite engine. The main idea here is that artists can use physical light units and adjust the camera as if it was a real digital camera. The first part of this is to model exposure. The exposure of

a camera determines how bright or dark an image will appear. It is controlled by just three camera settings: aperture, ISO and shutter speed [1].

The aperture is a physical mechanism at the end of a camera's lens that can widen and contract to control the effective area of the lens. The aperture is described in f-stops which describes the relation between the focal length and the diameter of the entrance pupil (effective aperture). $N = \frac{f}{D}$ where N is the f-number, f is the focal length and D is the entrance pupil diameter. Note how a lower f-number equates into a larger aperture and thus a brighter image.

The ISO of a camera describes how sensitive the sensor is to light. A higher ISO means that the result image will be darker. The ISO values start at a 100 (Base ISO) and double with each step. The reason that we cannot exclusively use the ISO to control the exposure is because that does not work in real digital camera's. When using very high ISO a real camera will start to show noise. We do not have this artifact in path tracing and we see no point in artificially adding noise.

Finally, the shutter speed or shutter time is the amount of time that the sensor is exposed to light. Increasing the shutter time allows for more photons to reach the camera's sensor and thus increasing the exposure. This is often used to make pictures in the dark. In real life, the "disadvantage" of increasing the shutter time is that it increases motion blur. Motion blur could easily be integrated in the path tracer by using the tech that we developed in the second assignment.

To put it all together we need to calculate the exposure based on these three parameters. The first step is to calculate the exposure value at ISO 100, also called $EV_1 00$. We can define $EV_s$ with s being the ISO value as: $EV_s = log_2(\frac{N^2}{t})$. Here N is the relative aperture (f-number) and t is the exposure time (or shutter speed) in seconds. The relation between any $EV_s$ and $EV_1 00$ is defined as $EV_1 00 = EV_s - log_2(\frac{S}{100})$. We combine these two formula's to calculate the $EV_1 00$ for our camera:

$$EV_s = log_2(\frac{N^2}{t}) \; and \; EV_{100} = EV_s - log_2(\frac{S}{100})$$

This gives:

$$\begin{aligned} EV_{100} + log_2(\frac{S}{100}) &= log_2(\frac{N^2}{t}) \\ EV_{100} &= log_2(\frac{N^2}{t}) - log_2(\frac{S}{100}) \\ EV_{100} &= log_2(\frac{N^2}{t} / \frac{S}{100}) \\ EV_{100} &= log_2(\frac{100N^2}{t\,S}) \end{aligned} \tag{12}$$

The EV quantity is not to be confused with the photometric exposure H (in

lux.s) which describes the scene luminance reaching the sensor. H is defined as:

$$H = \frac{qt}{N^2}L = tE$$

Here L is the incident luminance (candela/m2), t is the shutter time, N is the aperture f-number and According to *Film speed — Wikipedia, The Free Encyclopedia* [14], a typical value of q is 0.65. The ISO standard defines three different ways to relate photometric exposure and sensitivity: Standard Output Sensitivity, Saturation Based Sensitivity and Noise Based Sensitivity. We are using Saturation Based Sensitivity (SBS) since it is easy to use and because Frostbite did it. SBS is defined as the maximum possible exposure that does not lead to a clipped or bloomed camera output. The conversion from SBS to photometric exposure is defined as: $H_sbs = \frac{78}{S_sbs}/$. Using equation 12 we can now calculate the maximum luminance, which is the luminance that maps to an output of 1.0.

$$\begin{aligned} H_{sbs} &= \frac{78}{S} \\ \frac{q\,t}{N^2}L_{max} &= \frac{78}{S} \\ L_{max} &= \frac{78}{S}\frac{N^2}{q\,t} \end{aligned} \tag{13}$$

## 5.3 Tone Mapping

The second to last step of the whole camera process is to add tone mapping. When a scene contains both very dark and very bright area's, it has a high dynamic range. One option would be to set the maximum luminance to the brightest part of the screen. The problem with this approach is that it darkens the image and makes badly lit area's almost completely black. Adjusting the camera such that the maximum luminance is lower resolves this problem, but results in bright parts being clamped at 1.0. This is why digital photo camera's use tone mapping to try to remap pixel values to within the range of 0.0 to 1.0.

For our path tracer we have implemented two types of tone mapping. First of is a very simple tone mapping formula, which was suggested by Reinhard in his paper "Photographic tone reproduction for digital images" [11]. $L_d(x,y) = \frac{L(x,y)}{1+L(x,y)}$ is a simple yet reasonably effective, bringing all luminance's to within the displayable scale. In this paper Reinhard also presents a more advanced tone mapping scheme, that uses Guassian blur to examine the luminance of neighbouring pixels. We did not implement this because of time and performance considerations. The second tone mapping algorithm that we support is Uncharted 2 tone mapping. It was presented at the Game Developers Conference [7] although no formula was provided. Fortunately, *Filmic Tonemapping Operators* [6] provides a implementation of the Uncharted 2 tone mapping function. Since it

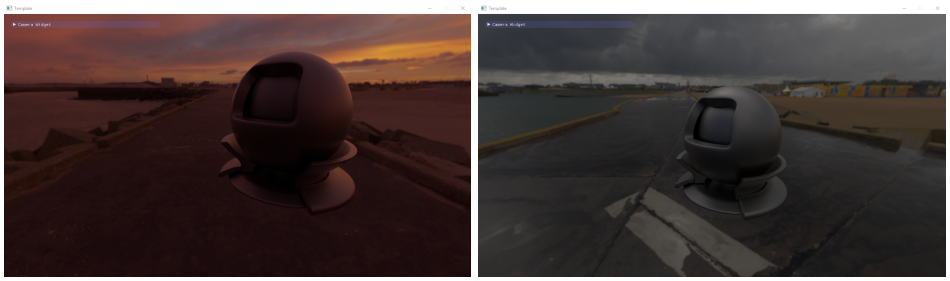

Figure 12: Image Based Lighting (Skydome)

is a black box function, we will not show the formula here but instead refer to *Filmic Tonemapping Operators* [6].

The final part of the camera pipeline is to apply gamma correction. We already implemented gamma correction in our ray tracer so for details we would like to refer to our first assignment's paper.

# 6 Skydome

A useful feature in any PBR shading environment is Image Based Lighting (IBL). IBL is a rendering technique which involves capturing an omnidirectional representation of real-world light information as an image[15]. A more easy explanation of this is that one takes a real world (HDR) 360 degree image and "wraps" it around the scene. In games this technique has been used for decades to model the sky, which is where it got the name skydome.

Our first implementation of IBL used a cubemap, but we quickly found out that there are little (free) cubemap images available on the internet. We therefore switched to sphere mapping, which accepts 360 panorama images which are much more widely available. We sample the skydome when a random ray misses the scene and goes into the void. The idea of a skydome is that it represents objects at an infine distance. We may thus set the origin of those rays to the origin of the scene. We then simply sample the skydome by converting the rays direction to polar coordinates. This technique is an easy way to render individual objects in a convincing way without having to create and path trace a large scene. Figure 12 provides some images of our skydome implementation.

# 7 Performance

## 7.1 Russian Roulette

To avoid needlessly allocating resources to rays that return little energy, we apply the standard Russian Roulette method.

Every ray has a certain probability of dying and being discarded. If the ray survives, its value is boosted by the inverse of the probability of death. This simple method heavily reduces the number of rays that have to be checked, especially after a couple of bounces, and does not introduce bias.

Being our path-tracer implemented on the GPU, we also introduced a hard limit on the maximum number of bounces, as we don't want to do a shading pass on a the very small set of rays that are so lucky as to survive for many bounces. This introduces bias, but so little that it's almost unnoticeable.

How do we choose the probability? At the end of each shading pass, we get the value of the integral of the BRDF. The maximum value among the three color channels (clamped to 1) will be our probability. This method has been suggested by our teacher J. Bikker and we had good results with it.

## 7.2 Wavefront Path Tracing

All modern (>2012) AMD GPUs (GCN) have the same core architecture. Each Compute Unit (CU) consists of 4 16-wide SIMD processors resulting in wavefronts of 64 threads. This is unlike previous architectures where SIMD processors where used to process vector data types within threads.

With the new architecture, registers to store thread local data are called Vector General Purpose Registers (VGPRs). Each compute unit has access to 64KB of vector registers which equals 1KB per SIMD lane. Each register contains 4 bytes, which means the total number of registers per lane equals 256. Larger datatypes such as vectors and double precision floats are stored in consecutive registers.

Our original implementation of the path tracer used one mega kernel. Even without microfacet BRDFs or refraction, the kernel used 124 registers and 80 scratch registers in 32-bit mode according to AMD CodeXL. Scratch registers are virtual registers in (slow) global memory because there are not enough real registers available. Suprisingly, when compiling in 64 bit mode, the AMD compiler manages to store all data into 215 registers, without making use of scratch registers.

The problem with using so many registers is that it does not allow the GPU to hide latency with compute. GPUs, unlike CPUs, do not have an advanced caching system. Instead they rely on quickly switching between wavefronts within a CU to hide the latency. When a kernel uses so many registers, it limits the occupancy of the CU. For example, when using 215 registers there is not enough room (256 - 215 = 41 registers) for another wavefront executing that same kernel.

It is clear that limiting our kernels to 1 or 2 instances forms a real performance issue. "Megakernels considered harmful: wavefront path tracing on GPUs" [10] suggests splitting the kernel into multiple smaller ones. We have implemented this in our path tracer as a way to reduce the registry pressure. We split our program into 4 main steps: primary ray generation, ray intersection, shading and shadow ray intersection.

The program starts by creating buffers which will contain the random walk rays, shadow rays and shading data. We allocate enough room to support up to a configurable number of rays per pass. We start of by running the ray generation kernel which fills up the random walk ray buffer. Then, we run the intersect kernel which traces the random walk rays and stores the intersection data needed for shading in a separate buffer.

Afterwards, the shading kernel is run which performs the shading for the intersections, adding luminance to the pixels whose ray have hit a light. Additionally, the shading pass generates new random walk rays and shadow rays if necessary. When rays get too long (up to a configurable amount) they are terminated.

For efficiency reasons, the kernel compacts the output to the left of the ray arrays using an atomic counter. We have experimented with using more advanced techniques like using compaction within a (64 thread) OpenCL work group. This did not result in a considerable performance difference. We also tested using separate counters for shadow and random walk rays, but decreased performance dramatically.

At this point we can read the number of surviving rays; which helps us to adjust the work group size and to determine if to stop. Finally, we execute the kernel that traces the shadow rays and adds the luminance to the resulting pixels if necessary. We now go back to the start of the loop and append new rays to the random walk ray buffer. This process is repeated until we traced a path for each pixel.

## 7.3 OpenCL Notes

First of all, we were scared that the CPU synchronization we introduced with wavefront path tracing would form a huge bottleneck. We tested the performance impact by always stopping after a fixed amount of iterations and using a fixed work group size. For these tests each pass processed as many rays as there are pixels on the screen (1280x720) and we stop after 10 passes. Without CPU synchronization, those passes took 23.4ms. With CPU synchronization (but still running the full work groups) we managed 25.4ms. So on average, we lost 0.2ms per pass (2ms / 10 passes).

We have also experimented with OpenCL 2, which is currently only available on Intel and AMD. Since NVidia clearly does not support OpenCL 2 from a business standpoint, we felt like ditching NVidia support could be explained as a business decision too. The main two advantages of using OpenCL 2 is device-side enqueue and pipe objects. We did not get to work on the pipes because of some problems with device-side enqueue and the AMD driver.

Device side enqueue is arguably the most important new feature introduces in OpenCL 2.0. It gives a kernel the ability to enqueue other kernels with no CPU interference. This is a feature that NVidia also provides in their CUDA API. In our

scenario it would have removed the need for a CPU read back / synchronization.

Unfortunately, there is an undocumented limitation that images cannot be passed to kernels using device-side enqueue. The documentation states that images are an opaque data type that can only be used as kernel or function arguments. A possible work around would be to have a main kernel which performs shading (which requires textures) and executes the other kernels in a while loop. This does not seem to be possible however, since there is no way to make a kernel wait for the execution of another one when using device-side enqueue.

After implementing the textures as buffers with manual linear sampling we were finally able to get the program up and running. On Intel that is, the AMD driver seems to be bugged in terms of device-side enqueue where it will freeze the computer and require a hard reboot. Furthermore, running our old OpenCL 1.2 code in OpenCL 2.0 mode resulted in a 3x performance decrease which is unacceptable. When not using any new features, the resulting ISA should be the same. It is clear that the AMD OpenCL 2.0 compiler is still in the early days even though the OpenCL 2.2 standard has already been released. It seems like Khronos is pushing out all kind of fancy new APIs which only exist on paper.

Furthermore, the debugging capabilities of OpenCL are just disappointing. GPU crashes causing a whole system to freeze make debugging almost impossible. On the CPU side, debugging is not much better. Both Intel and AMD provide tools for debugging OpenCL on the CPU. AMD's tool (CodeXL) simply crashes (some alignment error within the AMD driver) when compiling our kernels in debug mode. The Intel tool does work but debugging is very bare. Breakpoints are supported but only on one predetermined thread. This means that debugging crashes are almost impossible. When you hit a breakpoint, other work items executed in the background. When you "step over" the breakpoint, one of those other threads will have crashed in the meantime. The error messages provided are not very helpful either. Even in debug mode, Visual Studio will break somewhere deep in a Intel Threaded Building Blocks (TBB) thread. The only useful information provided is whether it was a read or write error.

Our last point is the way AMD's compiler eats registers for breakfast. This is a known issue about which many people have complained on their forum. AMD does not seem to have any interest in improving this. NVidia's CUDA compiler does give the programmer the ability to limit the register usage. With all the trouble we went through with OpenCL (spending almost 75% of our time on blind debugging) we get why CUDA is so dominant. Although we do not agree with NVidia's stance on OpenCL, we have to admit that CUDA is far superior.

# 8 Conclusion

We feel we have learned a lot throughout the development of this project, by learning how path tracing works and trying various physically based rendering techniques. We experimented with various BRDFs and learned the qualities and

flaws of each. We improved our camera system and our GPU pipeline. We are very satisfied with the results obtained, given that we are able to render very pretty images.

Something that we would have liked to do is looking at more advanced shading models including subsurface scattering. Unfortunately there was not time for that. Another thing we would've liked to do is looking into RIS and figure out how to make it work with our other variance reduction techniques.

As a final note, we would like to thank our teacher Jacco Bikker for administering this course, we feel that his insight and feedback were extremely helpful in learning these advanced graphics techniques.

# References

[1] *Camera Exposure*. Camebridge in Colour. URL: http://www.cambridgeincolour.com/tutorials/camera-exposure.htm.

[2] Keenan Crane. "Bias in Rendering". In: (2006). URL: https://www.cs.cmu.edu/~kmcrane/Projects/Other/BiasInRendering.pdf.

[3] *Depth of Field*. University of Washington, 1999. URL: https://courses.cs.washington.edu/courses/cse457/99sp/projects/trace/depthoffield.doc.

[4] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004. ISBN: 0321228324.

[5] Tobias Franke. *Notes on Importance Sampling*. Mar. 2014. URL: http://blog.tobias-franke.eu/2014/03/30/notes_on_importance_sampling.html.

[6] John Hable - Filmic Games. *Filmic Tonemapping Operators*. http://filmicgames.com/archives/75. May 2010.

[7] Naughty Dog - GDC. *Uncharted 2: HDR Lighting*. http://www.gdcvault.com/play/1012351/Uncharted-2-HDR. 2010.

[8] Craig Kolb, Don Mitchell, and Pat Hanrahan. "A realistic camera model for computer graphics". In: *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. ACM. 1995, pp. 317–324.

[9] S Lagarde and C De Rousiers. "Moving Frostbite to PBR". In: *Proc. Physically Based Shading Theory Practice*. 2014.

[10] Samuli Laine, Tero Karras, and Timo Aila. "Megakernels considered harmful: wavefront path tracing on GPUs". In: *Proceedings of the 5th High-Performance Graphics Conference*. ACM. 2013, pp. 137–143.

[11] Erik Reinhard et al. "Photographic tone reproduction for digital images". In: *ACM transactions on graphics (TOG)* 21.3 (2002), pp. 267–276.

[12] Christophe Schlick. "An Inexpensive BRDF Model for Physically-based Rendering". In: *Computer graphics forum*. Vol. 13. 3. Wiley Online Library. 1994, pp. 233–246.

[13] Bruce Walter et al. “Microfacet models for refraction through rough surfaces”. In: *Proceedings of the 18th Eurographics conference on Rendering Techniques*. Eurographics Association. 2007, pp. 195–206.

[14] Wikipedia. *Film speed — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Film%20speed&oldid=761416299. [Online; accessed 31-January-2017]. 2017.

[15] Wikipedia. *Image-based lighting — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Image-based%20lighting&oldid=727870218. [Online; accessed 31-January-2017]. 2017.