DTOADQ

aodq

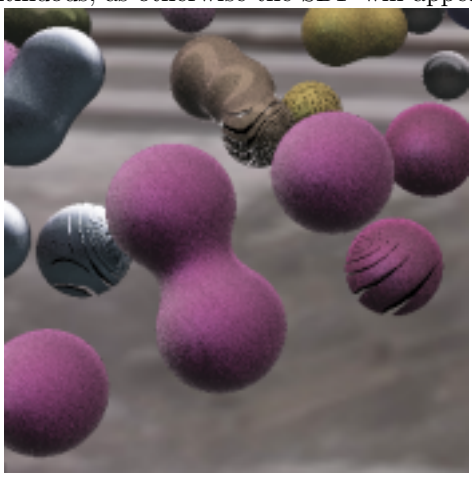
November 29, 2017

1 Signed Distance Fields

Signed Distance Fields (SDFs) are a mathematical tool to represent a 3D primitive by describing its surface via a distance field. Given an arbitrary point in space, the SDF of a primitive is the distance to its nearest point/surface. For example, the SDF of a sphere is $\parallel O \parallel - R$. With SDFs, it's possible to march a ray along the scene until it hits the surface of an SDF within a given bound, using the returned SDF distance as the distance to march. Some example SDFs are shown below:

$$S(P, R) = \parallel P \parallel - R \qquad \text{Sphere with Radius}$$
$$S(P, N) = P \cdot N \qquad \text{Plane with Normal}$$
$$S(P, H, R) =< \parallel P_{x,z} \parallel - R,\ P_y > -H \qquad \text{Torus with Radius and Hole}$$

The biggest issue with SDFs is Lipschitz continuity. The SDFs themselves have a distance gradient, that is, their bound. The bounds of a SDF is expected to be at most 1 in order to remain Lipschitz continuous, as otherwise the SDF will appear deformed as shown below.



This is a larger problem in light transport than it is in raytracing because small discontinuities can impact the lighting of an entire scene. And to add to that, most interesting operations (such as the meta balls shown in the image above) will violate Lipschitz. The easiest way to fix this is to multiply the distance field by a constant to stay below 1, but this is less of a fix and more of a hack.

In order to compute a normal of an SDF, you must perform a gradient approximation: the central distance onf the SDF at P. For example, the difference quotient:

$$\frac{(f(x+h) - f(x))}{h}$$

The most obvious way to calculate this is the six-point gradient; where for each axis you compute $f(x+h) - f(x-h)$, and normalize the result. This returns a vector pointing in the direction where

the map's SDF changes the most: the normal. A four point gradient is possible; as long as each each point of the gradient is multiplied by ϵ before normalizing. The speedup from six to four is worthwhile, as computing each gradient is expensive as it involves a sdf map call. There are more utilities with gradients not yet currently provided in DTOADQ but are worth mentioning, a few are: an additional normal call using an alternative (higher-quality) SDF map can be used to compute a 'normal' bumpmap, and analyzing the gradient by hand allows a cheap normal computation of primitives such as a spheres.

Overall, it seems that while SDFs are fun to play around with, for the case of writing a generalized light transport algorithm to apply to SDFs causes a lot of problems while being generally limiting. The biggest problem is dealing with transparent materials, and dealing with visibility checks. For transparent materials, it's required that the ray march 'through' the SDF primitive, and find the exit. The solution is to apply a 'shell' to the SDF, or in other words, force the SDF to be unsigned:

```
float Shell ( float dist, float r ) {
  return fabs(dist) - r*0.5f;
}
```

2 OpenCL

OpenCL, and GPU programming in general, is where I am least knowledgeable or interested. Well, OpenCL is easy enough to work with compared to OGL or CUDA, however I'm not good at dealing with the intrinsics of the GPU: optimizing code, knowing how memory and thread/workers should be handled, etc. For example, probably the biggest problem with the DTQ kernel right now is it is a single file megakernel. There just was never any interest to split it into multiple files nor multiple kernels. However, working directly with the OpenCL api is easy enough, and I did write my own OpenCL module to handle OCL in a rather convenient method, for example, the DTOADQ kernel call looks as such:

```
shared_info.image.Lock(); // Acquire OCL Mem from OGL image object
ocl.Run(
  ocl.CLStoreMem(shared_info.rw_image),
  shared_info.image.RWrite(),
  ocl.CLStoreMem(shared_info.image_metadata),
  shared_info.camera,
  shared_info.timer,
  ocl.CLPredefinedMem(imgs),
  ocl.CLStoreMem(shared_info.ocl_material),
  debug_values,
  ocl.CLStoreMem(shared_info.rng_states),
  // kernel size
  shared_info.image.x, shared_info.image.y
);
shared_info.image.Unlock();
```

however the problem comes with dealing with the intrinsics of GPU; optimizing code for the GPU and knowing how memory and threads/workers are handled.

One benefit of OpenCL is its ease to debug in comparison to OpenGL. One immediate benefit is printf, which surprisingly even works on the GPU. To make printf easily readable, I only use print on the center pixel of the image. Even so, due to the complexity of light transport, the vast majority of time spent on this project went directly towards debugging the DTOADQ kernel.

OpenCL also makes it relatively easy to send and receive data between the host and GPU. In comparison to OpenGL, this is a big win, however, with OpenGL4.3's Shader Storage Buffer Object, this is now no longer a selling point.

OpenCL also has huge drawbacks. The biggest problem is getting the results to be rendered in OpenGL. TODO. Also, there are minor annoyances, like the lack of a matrix data type, no function overloads, long compile times, or variadic functions. There are also annoyances with GPU programming in general. No dedicated random function. Though, possibly, long compile times is my fault for using a mega-kernel instead of splitting it into many small kernels.

It seems that the GPU is not for me or DTOADQ and something I do not enjoy to use nor am good at. However, for this project, I had to deal with it. To overcome the lack of a random function, of which high quality uniform random generation is necessary for Monte Carlo simulations, I used the MWC64X algorithm. While there are better algorithms, they fail on branches which makes it useless for DTOADQ. It seemed to work well enough. The downside is that each worker/pixel needs its own random state, preserved across multiple kernel calls. There are indefinitely better solutions.

3 Light Transport Introduction

4 Bidirectional Path Tracing

The main foundation of the mathematics and theory have come from many sources, such as Veach's thesis to PBR 3rd ed. and various open source path tracing kernels such as NanoRT, Embree, etc. However, the implementation differs from these in that the algorithm is optimized for the GPU. This section is about transforming the algorithms and mathematical models into something easily and efficiently implementable the GPU, however I will work from the ground up in the theory and mathematics behind light transport.

Calculating probability distribution functions is a very vital component in Monte Carlo Sampling. The PDF states the probability of a random variable to take the form of a given input. Its usefulness comes when a Monte Carlo sample has a smaller or higher total contribution compared to other samples. That is, one sample may have a higher probability distribution, and thus a higher contribution to the path. That's the strategy behind Monte Carlo;

$$\int f(x) \approx \frac{1}{N} \Sigma_0^N \frac{f(x)}{p(x)} \tag{1}$$

The PDF applies to everything from path contributions, solid angles of a BRDF, uniformly sampling a sphere, etc.

The biggest conceptual barrier is that, for example, given a solid angle in a hemisphere Ω, the probability for a random uniform X to equal ω is 0. That's not what the PDF calculates, instead, it's the probability that a random uniform X could potentially equal ω when compared to other ω s.

The good thing is that calculating these are relatively easy. The first step is to find the cumulative distribution function $CDF = Pr(X \leq x)$. Where Pr is the probability, X is a uniformly random variable, and x is a known constant. This gives the probability that, for a formula P, where $X \in \Omega$, and Ω is the set of all possible values for CDF, that $X \leq x$. Then the PDF p is defined by the derivative of CDF. To verify that the PDF is correct, integrate it over Ω, this final value should equal 1. This makes sense intuitively – it is a probability after all, so the sum of the probabilities of every possible input must be 1. For an example, how I calculated the PDF of an area light source $L_e^0(v) = L_o(P, \omega_o)$ is quite simple. Firstly,the area light is defined as a sphere, and thus our set of all possible values is correlated to surface area: $\Omega = [0, 2.0f * \tau * R^2]$. Thus,

$$CDF(x) = Pr(X \leq x) = \frac{x}{2.0 * \tau * R^2} \tag{2}$$

$$p(x) = \frac{dCDF(x)}{dx} = \frac{1.0}{2.0 * \tau * R^2} \tag{3}$$

$$\int_0^\Omega p(x)dx = 1.0 \tag{4}$$

In code

```
float Light_PDF ( Emitter* m ) {
  return 1.0f/(2.0f*TAU*SQR(m->radius));
}
```

Now, the importance of probability distribution functions comes when performing Monte Carlo approximation on an integral. Take, for instance, the most well-known form of the rendering equation:

$$L_o(P, \omega_o) = L_e(P, \omega_o) + \int_S f_s(P, \omega_i, \omega_o) L_i(P, \omega_i) |cos\theta_i| d\omega_i \tag{5}$$

Where:

$Lo(P, \omega_o)$ is the radiance at point P in space towards direction ω_o.

$L_e(P, \omega_o)$ is the emittance at point P towards ω_o (such as a lamp).

S is the unit sphere in which the angle ω_i is integrated over in respects to the normal at P. Also, this means, implicitly, that the equation is in respects to solid angle; the angle at which a primitive A subtends B.

f_s is the Bidirectional Scattering Distribution Function. This will be detailed in the BSDF chapter, suffice it to say, it describes how photons are scattered (or absorbed) from point P in space from direction ω_i towards ω_o – either through reflection (BRDF), or transmittance (BTDF). In volumetric path tracing, there is also subsurface scattering, but this phenomena is only approximated in DTQ.

$L_i(P, \omega_i)$ is the rendering equation at P from ω_i

And $cos\theta_i$ is the normal attenuation.

Conceptually, this is the behavior of photons bouncing off several surfaces of a scene. While it's a very simple model, it's impossible to solve analytically for anything but the most simple of scenes, mostly due to, for example, the radiance at $P0$ relies on $P1$, and $P1$ relies on $P0$. Another thing to remember is that while it is shown to be integrated over a sphere, it is usually shown to be integrated over hemisphere Ω using either f_r or f_s.

To compute this unbiasedly, it is required to use PDFs as described before to approximate the integral using Monte Carlo simulation. The technique is to generate N uniformly random samples, weight them by their probability distribution, and then average the results:

$$\int f(x) \approx \frac{1}{N} \Sigma_0^N \frac{f(x)}{p(x)} \tag{6}$$

With standard "backward" path tracing, a photon would be emitted from a light source and scatter in the scene until it hit an 'eye' or 'camera'. While this is what happens in reality, in computer graphics, most of the samples never end up hitting the camera/eye, so it is much more efficient to instead start from the eye, and end at an emitter/light source. In this case, since we are integrating over solid angle, the probably distribution function would be related to sampling the BRDF. So for example, on a specular surface, the PDF is 1.0 where $\omega_o = reflect(\omega_i, N)$ and 0.0 otherwise. An obvious optimization is to only sample over the valid BSDF angles, which is known as importance sampling. In DTQ, there are three BSDF functions:

BSDF Spectrum: The albedo at (P, ω_i, ω_o)

BSDF Sample: Samples a ω_o given (P, ω_i)

BSDF PDF: The probability distribution of (P, ω_i, ω_o)

There are many more optimizations to be made as well. For example, at each scattering event, an additional photon can be sent towards each emitter. This is what is known as next-event estimation. While this improves performance drastically, naively averaging each contribution is incorrect, because each path has a different probability to their paths/contributions. The technique to sample them properly, multiple importance sampling, is related to BDPT, however it will be detailed . The important thing to understand is that, in this sampling strategy, there is a new PDF introduced which relates to the probability of a path being generated, and this is critical to understand the concept of BDPT.

Now, the optimization that is of focus in DTQ is Bidirectional Path Tracing. Instead of starting from an eye or light, a 'photon' (path) is generated from both, allowed to scatter in the scene for a few paths (the length is determined by Russian Roulette), and connected. It's not exactly the end-points of the paths that are being connected, but each vertex generated is connected to each other. [SHOW EXAMPLE]. This actually has an interesting correlation to next-event estimation, as it could be seen as 'extending' the amount of emitters that can be connected to.

For BDPT, like a lot of different light transport equations, things go much more smoothly when the equation uses a different measure. In this case, the measure is over surface area rather than solid angles. This looks like $v_0 \to v_1$. The transformation looks like so:

$$f_s(v \to \dot{v} \to \ddot{v}) = f_s(P, \omega_i, \omega_o) \tag{7}$$

$$L_e(v \to \dot{v}) = L_e(P, \omega_o) \tag{8}$$

$$L_e(v \to \dot{v}) = L_e^0(v) L_e^1(v \to \dot{v}) \tag{9}$$

That is, instead of looking at a point P with an incoming and outgoing angle (ω_i, ω_o), look at the path from P_{-1} (v) to P_{+1} ($\ddot{v}$). More accurately, it's the transport between three different surfaces of the scene. There's a small change to be made in the case of L_e, as DTOADQ [at least, for now], only has support for area-light. Thus consider:

$$L_e(P, \omega_o) \equiv L_e(P) \qquad \text{[for area lights]} \tag{10}$$

$$L_e(v \to \dot{v}) = L_e^0(v) \tag{11}$$

The $cos\theta_i$ is no longer applicable, instead there is now a geometry term, G, that converts a PDF with respect to solid angle to surface area, which involves the inverse squared distance, and the cosine angle between the geometric normals at v and $\dot{v}$:

$$cos\theta_n = v \cdot N(v) \tag{12}$$

$$G(v \leftrightarrow \dot{v}) = V(v \to \dot{v}) \frac{(cos\theta_n cos\dot{\theta_n})}{||v - \dot{v}||^2} \tag{13}$$

Where V is the visibility test. Thus the rendering equation with respects to area is:

$$L(v \to \dot{v}) = L_e(v \to \dot{v}) \int_M L(v \to \dot{v}) G(v \leftrightarrow \dot{v}) f_s(v \to \dot{v} \to \ddot{v}) d_A \tag{14}$$

where A is the area-measurement function on M, and M is the set of all points on the surface of a scene. That is, to say in regards to SDFs, M is the set of all points P where $|map(P)| < \epsilon$. The area-measurement function goes into measure theory, which while it is beyond my grasp currently to compute, conceptually I understand a bit of it. The important thing is that A is a function on the set of all subsets of M that returns the measurement of each set, and as long as the surfaces of the scene are sampled uniformly, this never actually has to be computed. To back it up a little further, Veach presented this as a means to integrate over a measure rather than using solid angles. This

proves that you can transform the rendering equation to different measurements. For example, instead of integrating over points on surfaces, one could integrate over an entire 3D space in order to perform volumetric path tracing. Transforming the rendering equation using measurements would be interesting in the future to look into.

Anyways, with the three-point form now described, by recursively expanding, so that the three point form is integrated over every possible set of possible paths, this can be rewritten as the more usable format:

$$I_j = \int_\Omega f_j(\bar{v}) d\mu(\bar{v}) \tag{15}$$

Where $\bar{v}$ is a path $z_0 \to z_s \to y_t \to y_0(eye \to light)$, Ω is the combination of all possible paths of any length, and μ is the area-product measure [the product of all the expanded A]. Special care has to be taken with the definition of path itself, as in the source code, there never exists a path $\bar{v}$, only eye-path $\bar{z}$ and light-path . Monte Carlo approximation, as shown above, is applied to the formula:

$$I_j \approx \frac{1.0f}{N} \Sigma_0^N F \tag{16}$$

$$F = \Sigma_s \Sigma_t \frac{f_j(\bar{v}_{s,t})}{p(\bar{v}_{s,t})} \tag{17}$$

where s is the light-path length, and t is the eye-path length. In DTOADQ, they are limited to $s + t \leq DEPTH$, a kernel-defined macro. Finally, to transform $\bar{v}$ into two seperate paths, so that an eye and light path may be generated, $\frac{f_j(\bar{v}_{s,t})}{p(\bar{v}_{s,t})}$ is split into two with a connection edge c

$$\frac{f_j(\bar{v}_{s,t})}{p(\bar{v}_{s,t})} \equiv \frac{f^L(\bar{y})}{p(\bar{y})} c(\bar{y}_s \leftrightarrow \bar{z}_t) \frac{f^E(\bar{z})}{p(\bar{z})} \tag{18}$$

And now

$$F = \Sigma_s \Sigma_t \alpha_s^L c_{s,t} \alpha_t^E \tag{19}$$

$\alpha_{s|t}^{L|E}$ is the final radiance of either the emitter or eye path, and thus it is recursive in nature. That is, the value of α_i is dependent on α_{i-1}, unless $i = 0$ in which case is the radiance of either the emitter or the eye path. However, this can be simplified further, because of Helmhotz reciprocity, BRDFs return the same value if wo and wi are switched. This is also the reason why forward and backward path tracing work.

$$\alpha_i^{L|E} = \begin{cases} 1.0 & i = 0, \\ \frac{L_e^0(V_0 \to V_1)}{p_A(V_1)} | 1.0 & i = 1, \\ \frac{f_s(V_0 \to V_1 \to V_2)}{p_\sigma(V_0 \to V_1)} & i > 1 \end{cases} \tag{20}$$

For the case of $i = 0$, this would mean that the path does not exist. In other words, forward or backward path tracing, the latter of which isn't supported as camera lens are not defined in DTQ. For $i = 1$, it is either a point on the surface of a light, or the origin point on the 'delta eye'. Lastly, the BRDF is used to calculate the albedo of each primitive in the scene, and is then divided by the PDF sigma, which is the probability density of $v_i \to v_{i-1}$. α_1^L is shown below:

```
float3 N      = Sample_Cosine_Sphere(si, &pdf_pos),
       origin = light.origin + light.radius*N,
       dir    = Reorient_Angle(Sample_Cosine_Hemisphere(si, &pdf_dir), N);
float pdf_pos = 1.0f/(2.0f*TAU*SQR(light.radius)),
      pdf_amt = 1.0f/EMITTER_AMT;
pv->irradiance = light.emission/(pdf_pos*pdf_amt*pdf_fwd);
```

Note that the irradiance is being calculated in this case, that is, the amount of flux arriving at point pv. Though in the case of $s = 1$, this is more like the radiance. The irradiance at $t > 1$ and $s > 1$ is shown:

```
float3 sample_dir, sample_f;
float sample_pdf;
_BSDF_Sample(wi, N, vtx, &sample_dir, &sample_pdf, &sample_f, si);
*irradiance = sample_f * (*irradiance) * fabs(dot(N, sample_dir))/sample_pdf;
```

The connection strategy below describes how to connect the edges of the two paths. There is no $t = 0$ case as no physical camera lens exists on the scene.

$$c_{s,t} = \begin{cases} L_e(L_0 \to L_1), & s = 0, t > 1, \\ f_s(L_0 \to L_1 \to L2)G(E_1 \leftrightarrow L_1)f_s(L_1 \to E_1 \to E_0), & s > 0, t > 1 \end{cases} \tag{21}$$

In code:

```
Spectrum contribution = E1.irradiance * L1->irradiance;

// As there is no T=1 case, L1 -> E1 -> E0 [Le in s=1 case] must happen
contribution *= Subpath_Connection(L1, &E1, &E0);

// For s == 1, L1 = L0, a point on the surface of emitter, there is no
// L0 -> L1 -> E1 connection
if ( light_depth > 0 ) {
  // Connect L0 -> L1 -> E1
  contribution *= Subpath_Connection( L0,  L1, &E1);
}

// Geometric connection term [includes visibility check]
contribution *= Geometric_Term(&E1, L1, si, Tx);
```

Is the $s = 0, t > 1$ strategy worthwhile to compute? Those familiar with next event estimation would know that there are situations in which this is beneficial. Take, for example, a glossy surface with a small solid angle subtending to the surface of a large light source. The $s = 1$ sample strategy in this case would only contribute when the $L0$ vertex is within that solid angle. The effective strategy to sample in this case is $s = 0$. In code, $\alpha_i^E \, c_{s,t} \, \alpha_0^L$ looks like (MIS not shown):

```
float3 wi = normalize(E1.origin - E0.origin),
       wo = normalize(L1.origin - E1.origin);
float3 albedo = _BRDF_F(wi, E1.normal, wo, E1.material, E1.mat_col);
albedo /= BSDF_PDF(wo, wi, &E1);
Spectrum contribution = E1.radiance * albedo;
contribution *= Geometric_Term(&E1, &L1, si, Tx);
contribution *= tlight.emission/Light_PDF(&tlight);
```

Another question to consider is if the path should be able to reflect on the light source. While under real circumstances, this is allowed, the contribution is negligible and dismissed in DTOADQ. Also, there are no cases for $s = s, t = 0|1$ as the lens does not have a physical existence in the SDF map. The $t = 0$ case, where a photon from the light source hits the lens directly, is rather pointless anyways, as a path generated by light-tracing would have to be able to contribute to any pixel, which is not feasible on a GPU.

The amount that each path contributes differ greatly, so naively weighing each contribution uniformly by something like 1.0f, or even, by inverse path length, will producy very noisy images, and

would not be worth the additional computation time that BDPT requires. Thus, as mentioned earlier, probability distributions of the entire path are taken into account in order to generate multiple importance samples. The formula for multiple importance sampling on BDPT is as so:

$$MIS_{st}(\bar{x}) = \frac{P_s(\bar{x})}{\Sigma_i P_i(\bar{x})} \tag{22}$$

Conceptually, the numerator $P_s(\bar{x})$ is the path density that was generated, while the denominator is the path density of other connection strategies that could could have, in theory, created the path. The straight forward implementation of this has a lot of problems gone unresolved in Veach's thesis; PBR 3rd Ed, chapter 16 pg 1014-1016 addresses and describe these problems. Primarily, $P_i(\bar{x})$ will overflow CLFloat (due to the distance in the Geometric term), and the straight forward implementation is very long, hard to debug, and has a poor time complexity. But the important part is the end result of their solution:

$$r_i(\bar{x}) = \begin{cases} 1.0f, & i = s, \\ \frac{\overleftarrow{P}(\bar{x_i})}{\overrightarrow{P}(\bar{x_i})} * r_{i+1}(\bar{x}), & i < s, \\ \frac{\overrightarrow{P}(\bar{x_{i-1}})}{\overleftarrow{P}(\bar{x_{i-1}})} * r_{i-1}(\bar{x}), & i > s \end{cases} \tag{23}$$

However, DTOADQ can simplify a bit further; unlike above, there are two paths for which to evaluate. Specifically, in the above term, for where $i < s$, we have $\frac{\overleftarrow{p}(\bar{x}_i)}{\overrightarrow{p}(\bar{x}_i)}$, which is the light path (i is iterating towards light-length s). The eye contribution is inversed from this, (also iterating towards s, hence why $\bar{x}_{i-1}$ is used). . In DTOADQ's case, the weights are calculated in the same direction relative to its originator; in this case the perspective of the eye path. Not only this, but unlike any implementation I've found of BDPT, I don't recalculate each path's probability and MIS every vertex. That is, usually on calculating MIS for each connection, the MIS and probability values are recalculated for each vertex in the path. As this is not possible given that only the light-path exists in memory, an array for sum/probability is stored corresponding to each light vertex, and a single sum/probability is stored corresponding to E_1. Thus, the weight on each path looks like:

$$\mathcal{W}_i(\bar{x}) \equiv r_i(\bar{x}) \tag{24}$$

$$\mathcal{W}_i(\bar{x}) = \begin{cases} 1.0f, & i = 0, \\ \frac{\overleftarrow{p_i}(\bar{x_i})}{\overrightarrow{p_i}(\bar{x_i})} * \mathcal{W}_{i-1}(\bar{x}), & i > 0 \end{cases} \tag{25}$$

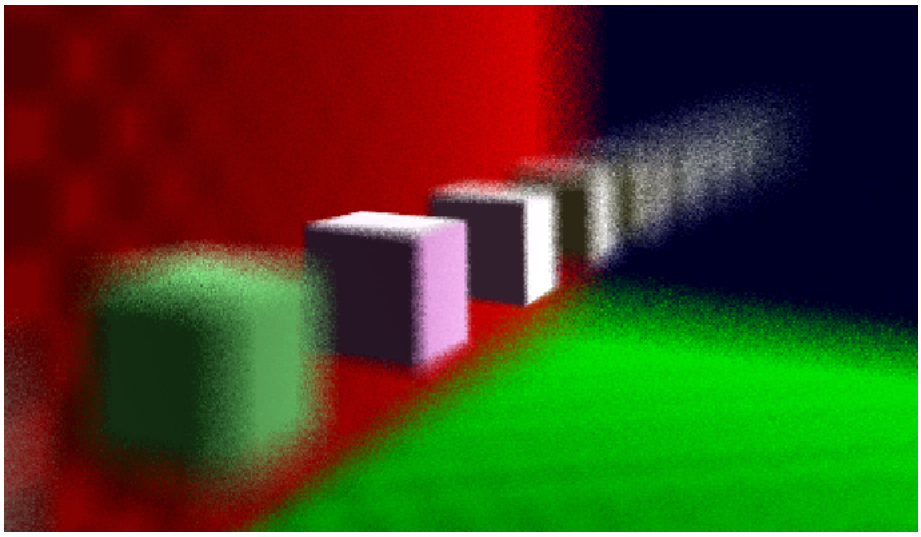
And then the expanded form of $\mathcal{W}_i(\bar{x})$ looks like

$$\mathcal{W}_i(\bar{x}) = \Pi_{n=2}^{i}\left(\frac{\overleftarrow{p_\sigma}(x_n)\overleftarrow{G}(x_n)}{\overrightarrow{p_\sigma}(x_n)\overrightarrow{G}(x_n)}\right) * \frac{\overleftarrow{p_A}(x_1)}{\overrightarrow{p_A}(x_1)} \tag{26}$$

And now the MIS looks like:

$$MIS_{s,t}(\bar{y}, \bar{z}) = \frac{1.0f}{\mathcal{W}_s(\bar{y}) + \mathcal{W}_t(\bar{z}) + 1.0f} \tag{27}$$

The camera handling function is based off glLookAt. The difference is that it supports field of view, depth of field and antialiasing. The model isn't perfect and has a few bugs, mostly because the camera used in DTQ is a pinhole camera, or as I call it delta camera. In general, very little time was spent on the camera mechanics. In the future, there will be a physical camera lens in the SDF, so as to actually simulate depth of field, aperture, etc. Shown below is an exaggerated depth-of-field in the raytracer.



5 DTOADQ BRDF

DTOADQ supports a "universal" physically-based sampling and spectrum BRDF. The spectrum BRDF is a combination of many different modified BRDF functions into one. There is a microfacet, subsurface and diffusive component in the DTOADQ spectrum BRDF, which are all combined together, with many variables to tune these components. There is diffusive, glossy, specular and transmittive sampling in the BRDF importance sample, with variables to allow specific control over which strategies to choose and how they are combined together. Regardless of how the BRDF is sampled, they use the same spectrum BRDF.

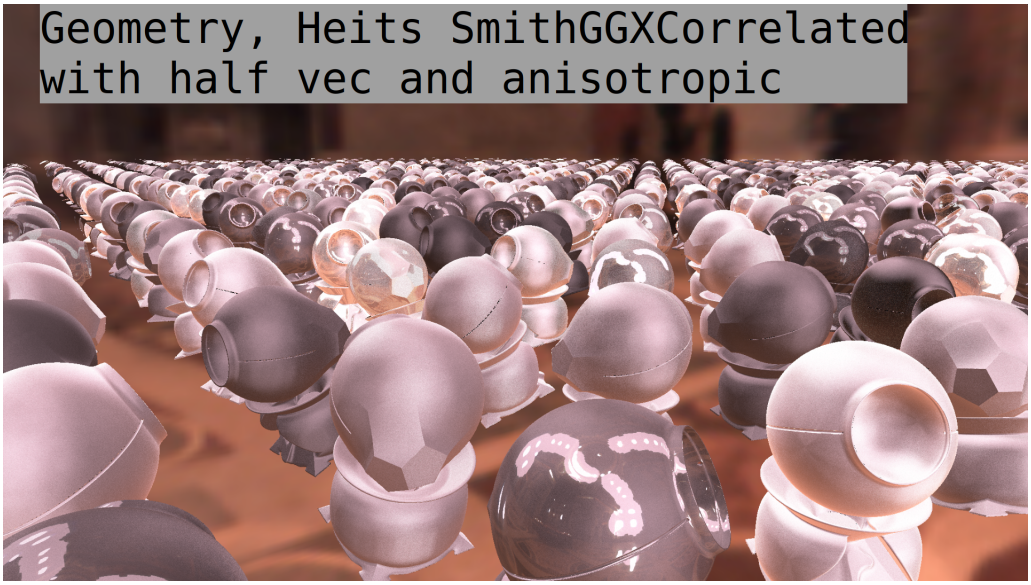
In order to develop these BRDFs, I wrote an SDF model based off the mitsuba model and wrote an IBL path-traced shader for quick prototyping; shadertoy.com/view/XtSyDm . The main reason to not use DTOADQ was the lack of cubemap support, which would have taken too long to implement.

The Microfacet model consists of a fresnel, distribution and geometric term. It's important to note that these have all been modified to use the half-vector $H = \|L+V\|$. This gives better visuals and allows for a specific energy conservation model. Anyways, the fresnel term approximates the fresnel factor, it's based off both Schlick's approximation and Disney's diffusive schlick fresnel, with modifications to use a *metallic* term as well as an albedo;

```
const float3 Fresnel_L = Schlick_Fresnel(cos_NL),
             Fresnel_V = Schlick_Fresnel(cos_NV);
const float F0 = m->fresnel * m->metallic,
Fresnel_diffuse_90 = F0 * SQR(cos_HL);
microfacet *= (1.0f - F0) * diffusive_albedo +
    mix(1.0f, Fresnel_diffuse_90, Fresnel_L) *
    mix(1.0f, Fresnel_diffuse_90, Fresnel_V);
```

The Geometric term is used to control how microfacets 'shadow' each other. The DTQ BRDF uses Heits' [2014] SmithGGXCorrelated modified to use a half vector, combined with an anisotropic term based off the GGX Anisotropic model:

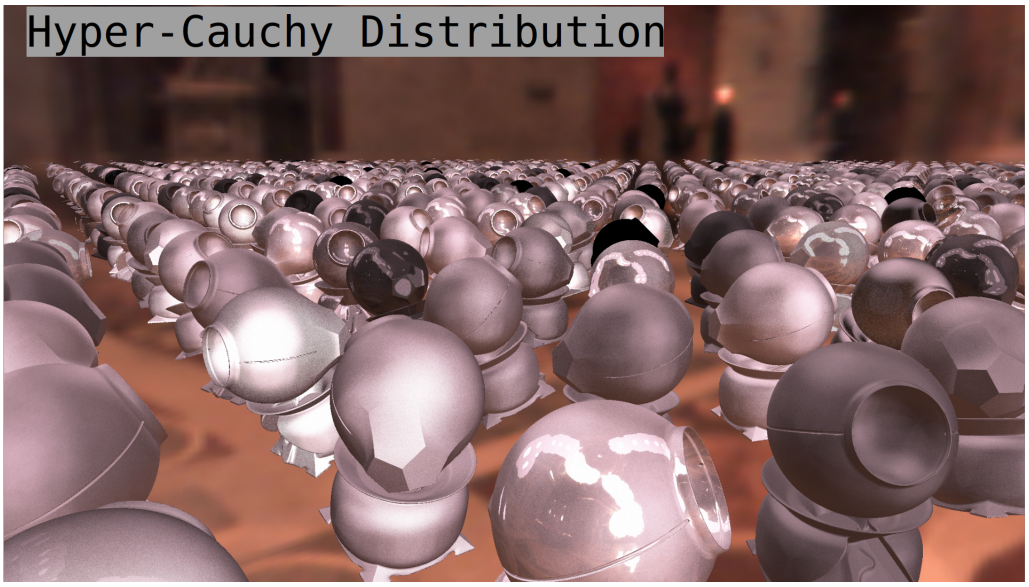
```
const float Param  = sqr(0.5f + sqr(m->roughness)),
Aspect = sqrt(1.0f - m->anisotropic*0.9f),
Ax     = Param/Aspect, Ay = Param*Aspect,
GGX_NV = Smith_G_GGX_Correlated(cos_HL, cos_NV, Ax),
GGX_HL = Smith_G_GGX_Correlated(cos_NV, cos_HL, Ay);
microfacet *= 0.5f / (GGX_NV*Ax + GGX_HL*Ay);
```



And finally, the Distribution term controls how microfacets are spread around the normal. It's based off the Hyper Cauchy with a roughness and metallic (shape) parameters [9]. The latter of which should most likely

be its own parameter in the future. The more readable mathematical model is shown:

$$\frac{(M-1)\,(\sqrt{2})^{2\,S\,-\,2}}{\pi\,R^2\,cos^4\theta\,(2+\frac{tan^2\theta}{M^2})^M} \tag{28}$$



These combine to form the microfacet under the half vector microfacet energy conservation model [Edwards et al. 2006].

$$\frac{F * G * D}{4\,(H \cdot V)\,((N \cdot L) \vee (N \cdot V))}$$

The diffusive component is a combination of both the subsurface and lambertian component. The lambertian is simply $\alpha^{0.5} * \frac{1}{\pi}$, and the subsurface component is based off the Henyey-Greenstein formula [Henyey], modified for half-vec:

$$\frac{m(1-r^2)}{4\pi}\,\frac{1}{(1+r^2-2\,r\,(H \cdot V))^{3/2}} \tag{29}$$

where m is metallic and r is roughness. This is then combined to the diffusive component of the BRDF using Fresnel factor like so:

```
const float R = (-0.3f + 1.3f*m->roughness)*cos_HV,
            M = (1.0f - m->metallic)*5.0f;
const float Rr_term = M * (1.0f - sqr(R))*IPI *
                      (1.0f/(pow(1.0f + sqr(R) - 2.0*R*cos_HL, 3.0f/2.0f)));
const float3 Retro_reflection = diffusive_albedo * Rr_term *
                  (Fresnel_L + Fresnel_V + (Fresnel_L*Fresnel_V*(Rr_term)));
diffusive_albedo = mix(diffusive_albedo, Retro_reflection, m->subsurface);
```

TODO: With 0.95 to exaggerate

The microfacet and diffusive components are then added together.

6 Video/Image Emitter

7 Future Improvements

Every emitter is chosen uniformly, however a better system would weight the probability of each emitter by

$$p_l = \frac{\phi_l r_l}{\Sigma_i \phi_i r_l}$$

Thus emitters that have a higher contribution to the scene, that is, a larger emittance and radius, will have a higher probability to be selected.

// What's defined: V1 -¿ V2 [we're evaluating V1] This is for a lens. // So, it's ultimately completely useless until an // actual lens exists in the SDF scene. Note that this isn't an actual // point on a surface of the scene (that's V2, next up for eval), this // is just an infinitesimal point based off where the camera currently // is, just like a delta light. So while the point actually exists on a // scene, the evaluation is on a "delta camera"

8 Bibliography

[9] Samuel D. Butler, Experimental and Theoretical Basis for a closed-Form Spectral BRDF model [Henyey] Diffuse Radation in the Galaxy, 1942. L, G. Henyey and J. L. Greenstein.