An Antique Cup of Tea

Sean Liu, Toki Migimatsu, and Wilbur Yang

How do we model metallic glint and steam? Our goal for this project was to produce a scene inspired by the one below with a scratched metal kettle, brushed metal cup, and steaming tea.

Rendering Brushed Metal

Specular surfaces with textures like brushed metal or metallic lacquer have complex “glinty” appearances under sharp lighting. Rendering this effect is difficult because the energy is concentrated in tiny highlights, and capturing this detail through Monte Carlo sampling is prohibitively expensive. For this project, we extended PBRT to render specular microstructures with an efficient algorithm introduced in Yan et al.

Microfacet Model

One way to render microstructures is with the Torrance-Sparrow microfacet model. This model describes surfaces as a probability distribution of microfacets that reflect light in different directions. The microfacet BRDF accounts for Fresnel reflectance (F), self-shadowing (G), and probability distributions over surface normals (D):

$$f_r(\omega_i \rightarrow \omega_r) = \frac{F(\theta_i')G(\theta_i')D(\omega_h)}{4\cos\theta_i\cos\theta_r}$$

Typically, the microfacet normal distribution functions D only depend on ω_h, the half vector between the incoming (ω_i) and outgoing (ω_o) light directions. However, with textured surfaces like brushed or scratched metal, D also depends on the position (u, v) on the surface. We load these 4-dimensional position-normal distribution functions as normal maps that specify normal directions $(s, t, \sqrt{1 - s^2 - t^2})$ for every (u, v) coordinate.

Pixel Footprints

Surfaces like scratched or brushed metal feature sub-pixel texture highlights that can't be captured effectively with importance sampling. Yan et al. handle sub-pixel details by integrating the position-normal distribution function over a pixel's footprint $\mathcal{P}$ on the surface's (u, v) coordinate system. They model this footprint with a Gaussian G_P, which later leads to nice simplifications.

Because PBRT does not model Gaussian footprints, we compute them from the SurfaceInteraction's `dudx`, `dudy`, `dvdx`, and `dvdy` parameters. We think one natural way to do this would be to find the closest symmetric positive definite approximation to $\begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} \end{bmatrix}$, and make that the covariance matrix Σ_P for our Gaussian. The intuition is that $(\frac{\partial u}{\partial x}, \frac{\partial v}{\partial x})^T$ and $(\frac{\partial u}{\partial y}, \frac{\partial v}{\partial y})^T$ should represent the two axes of the Gaussian ellipse, except the axes need to be orthogonal by definition of the covariance matrix. We could not find a closed form solution to this convex optimization problem for 2×2 matrices, so for efficiency and simplicity, we came up with an approximation.

Let $\vec{d_1}$ be the larger vector out of $(\frac{\partial u}{\partial x}, \frac{\partial v}{\partial x})^T$ and $(\frac{\partial u}{\partial y}, \frac{\partial v}{\partial y})^T$, while $\vec{d_2}$ is the smaller one. We set the Gaussian's major axis $\vec{v_1}$ to be $\vec{d_1}/2$, scaled such that the Gaussian extends to the magnitude of $\vec{d_1}$ at two standard deviations. We then set the minor axis $\vec{v_2}$ to be $\vec{d_2}$'s orthogonal projection to $\vec{d_1}$, scaled in the same manner. Finally, we construct the covariance matrix $\Sigma_P = RDDR^T$, where $R = \begin{bmatrix} \frac{\vec{v_1}}{\|\vec{v_1}\|} & \frac{\vec{v_2}}{\|\vec{v_2}\|} \end{bmatrix}$ and $D = \begin{bmatrix} \|\vec{v_1}\| & 0 \\ 0 & \|\vec{v_2}\| \end{bmatrix}$. The mean of the Gaussian is simply the (u, v) coordinates of the SurfaceInteraction, and the Gaussian is scaled by $\frac{1}{2\pi\sqrt{|\Sigma|}}$ to integrate to 1.

Yan et al. actually construct 16 subpixel Gaussian footprints for pixels subdivided into a 4×4 grid. However, we found that using whole pixels gave us sufficient detail for our images.

P-NDFs as a Sum of Gaussians

According to Yan et al., the position-normal distribution function of pixel footprints can be defined:

$$D(\vec{s}) = \int_{\mathbb{R}^2} G_p(\vec{u}) \delta(n(\vec{u}) - \vec{s}) d\vec{u}$$

Where $\vec{s} = (s, t)$ is the projection of the microfacet half vector on the unit disk, $\vec{u} = (u, v)$ is the location of the reflection on the object's surface, and $n(\vec{u})$ is the texture normal specified by the normal map. Intuitively, this equation finds how many (u, v) locations over the pixel footprint $\mathcal{P}$ contain the desired normal $\vec{s}$, and returns that as a probability. In practice, we need to replace the delta function with a Gaussian instead to avoid issues with sampling and singularities:

$$D(\vec{s}) = \int_{\mathbb{R}^2} G_p(\vec{u}) G_r(n(\vec{u}) - \vec{s}) d\vec{u}$$

The Gaussian G_r over the unit disk in (s, t) normal space introduces "roughness" to our material. By our intuition, this equation finds all the normals close to the desired normal $\vec{s}$ within the pixel footprint $\mathcal{P}$. G_r can be thought of as a 4D function, Gaussian over (s, t) and highly varying over (u, v). Yan et al. show that integrating this highly varying, high dimensional function by straightforward Monte Carlo sampling fails. To get around this issue, they approximate $G_r(n(\vec{u}) - \vec{s})$ with a sum of 4D Gaussians:

$$D(\vec{s}) = \int_{\mathbb{R}^2} G_p(\vec{u}) \sum_{i=1}^{m} G_i(\vec{u}, \vec{s}) d\vec{u}$$

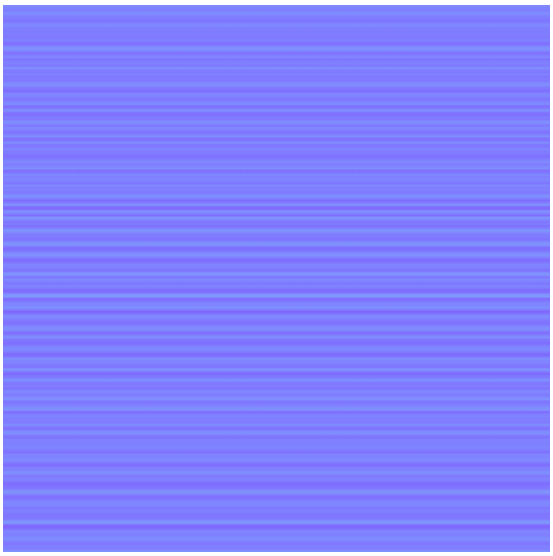
Now that all the terms are nicely Gaussian, it is possible to find a closed-form solution for this integral as a simple sum. Yan et al. do not write out this solution in its full form, but after some work (actually a lot of work), we find:

$$D(\vec{s}) = 2\pi h^2 \sum_{i=1}^{m} c_i |\Sigma_i|^{-\frac{1}{2}}$$

Where h is the interval between Gaussian elements in (u, v) space (which we specify as a parameter in `.pbrt` files), c_i is a scaling factor, and Σ_i is a covariance matrix. Defining each term explicitly takes a full page of equations, so we are also leaving it out in this writeup. Equation (21) in Yan et al. contains a typo that is fixed above.

Normal Map Generation

To create the normal map for brushed metal, we generated a 1×2048 pixel height map with uniform grayscale noise. We chose uniform noise over Gaussian because the abrasive pad used to brush metal likely creates grooves at a certain regular depth. Gaussian noise would imply that most of the surface lies at a mean height and grooves/mountains tend to stay shallow. We then stretched the height map to a 2048×2048 image and used Photoshop's 3D normal map generation function to produce a normal map like the following (cropped):



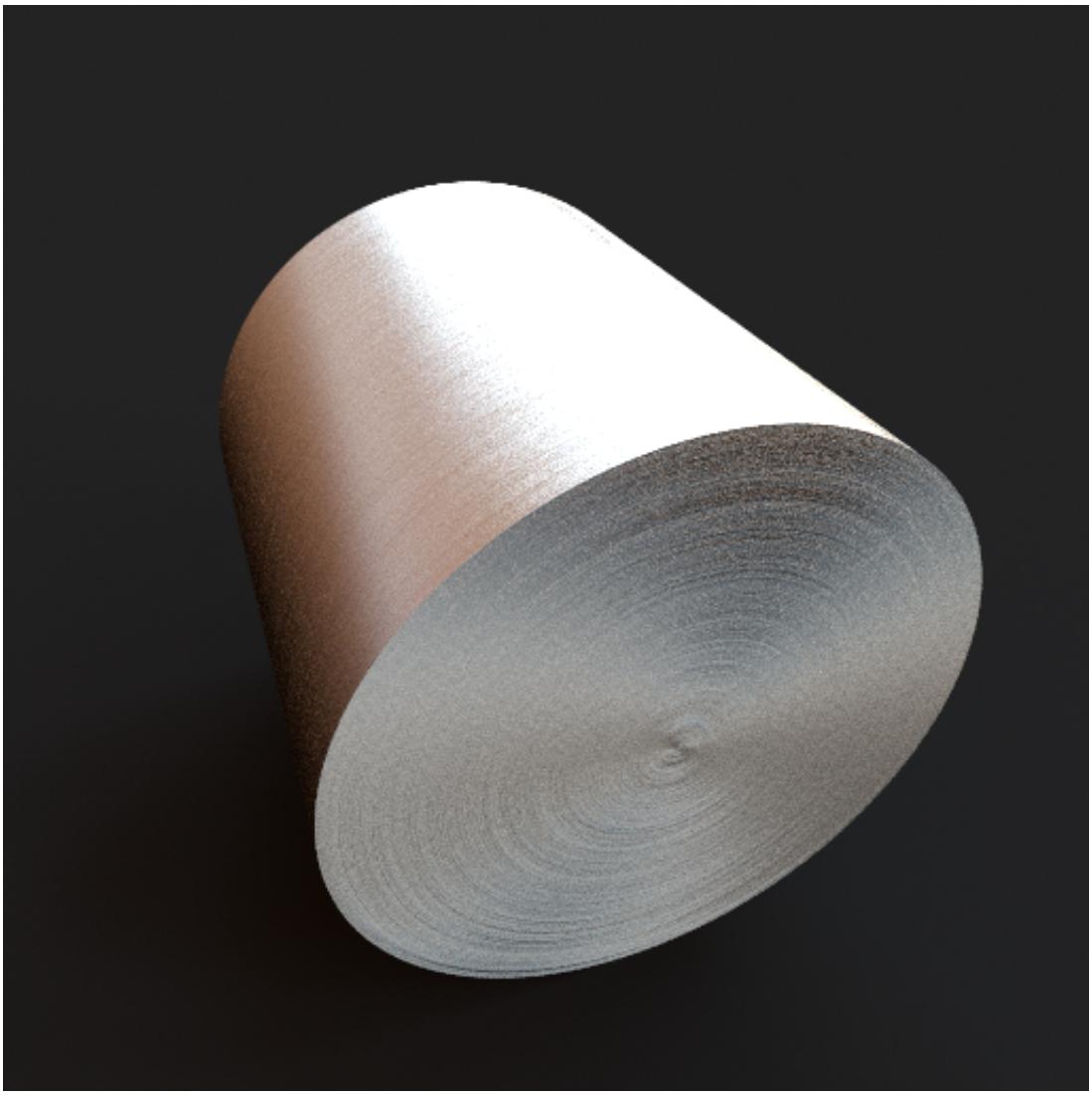
Implementation in PBRT

A 2048×2048 image with a Gaussian element interval size of $h = 0.5$ texels means there are over 16 million Gaussian elements G_i with tiny standard deviations and enormous scaling factors to integrate to 1. The pixel footprint Gaussian G_p tends to have large standard deviations and small scaling factors, since its covariance is defined by pixel differentials in texel space. This results in floating point operations with many large and small numbers (ranging from 1e-20 to 1e20), and was a conditioning nightmare.

The first step was to do as many of the floating point operations as possible in log space. Luckily, all of the Gaussians could be converted quite easily. The second step was to do some algebra with the equations to keep the magnitudes of the numbers around the same order, particularly for matrix multiplications. Third, we clip values and return early whenever numbers become too small or too large. Finally, we decided to scale the texture space defined over the range $(-1, 1)$ to the resolution of the image $(res.x, res.y)$, which helps prevent numbers from becoming too small. Figuring out how to scale the Gaussians, integrals, and Jacobians accordingly was difficult.

For importance sampling, we simply take the normal from the normal map at the given SurfaceInteraction (u, v) coordinate, and perturb it by a Gaussian with standard deviation given by G_r on the unit disk. Yan et al. implement an alternative importance sampling method on the Gaussian mixtures directly, since some of their textures don't rely on normal maps. They also use an acceleration hierarchy for speedup, but we found that simple bounding boxes on the pixel Gaussians was sufficient.

This was one image generated with conditioning issues even after converting to log space, clipping large values, and scaling the texture space. The white specks are a result of exploding numbers. This was only solved by rearranging the matrix algebra to group terms of similar orders. The texture used was 512×512 and contained Gaussian noise on top of the brushed patterns.



Brushed metal cylinder generated with a 2048×2048 normal map, Gaussian element step size of 0.5 texels, and roughness Gaussian standard deviation of 0.2.

Rendering Steam

Efficient sampling of participating media is important for creating high quality images with path tracing. PBRT's current medium sampling method uses distance sampling, which focuses more samples closer to the camera than ones farther away. While this works well for scenes with lights that are removed from the volumes, it is less effective for scenes where the lights are embedded in the volumes and in view of the camera. Kulla et al. proposed an importance sampling method called "equi-angular sampling," which "focus[es] more samples where the incoming light is strong."

Homogeneous Media

For a given ray passing through a homogeneous medium and a point light, the method re-parameterizes t such that “the origin is at the orthogonal projection of the light onto the ray. The integration bounds a and b represent the boundaries of the homogeneous medium (see figure below).

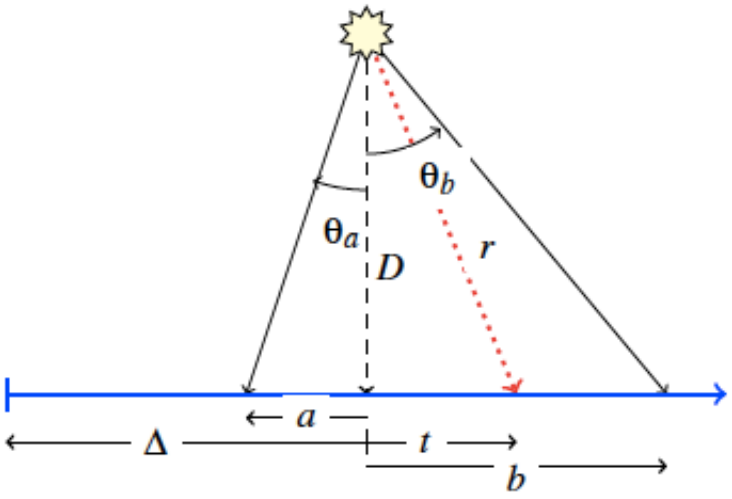


Figure 2: *Single scattering from a point light source*

Then, according to this new parameterization of the ray, we can use the following sampling function and normalized pdf over the interval [a, b]:

$$\begin{aligned} \text{pdf}(t) &= \frac{D}{(\theta_b - \theta_a)(D^2 + t^2)} \\ t(\xi) &= D\tan\Big((1-\xi)\theta_a + \xi\theta_b\Big) \\ \theta_x &= \tan^{-1} x/D \end{aligned}$$

Implementation in PBRT & Challenges

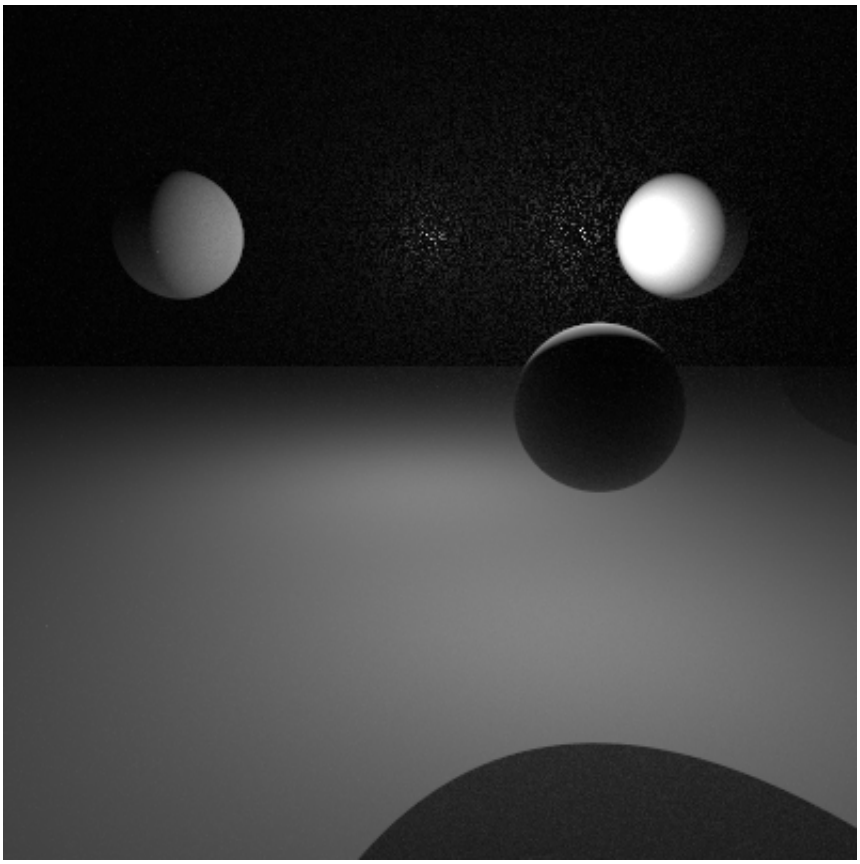
We created a new Medium class for equiangular sampling in homogeneous media. The Medium::Sample() function takes in a ray and a pointer to a MediumInteraction and returns an associated weight for the final radiance (Section 15.2 in textbook).

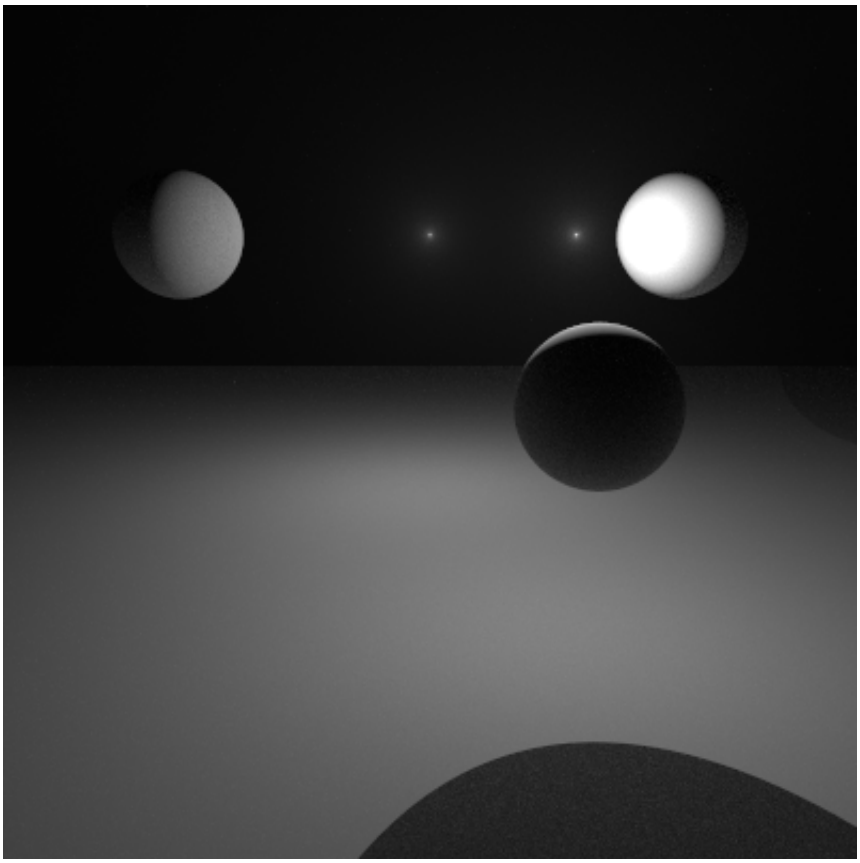
Upon receiving a ray, we first sample an exponential distribution defined over [0, inf), similar to 15.2.1 in the textbook, to see if we have a medium interaction. If the sampled position along the ray is beyond the medium boundary, then we register a surface interaction and return the corresponding weight (transmittance / pdf). If the sampled position is within the medium boundary, then we sample another position using the equiangular sampling method. The t value of this final position is then stored in the MediumInteraction for the integrator to handle. One challenging part of implementing this section is figuring out the correct weight to return. The transmittance could be computed easily for the homogeneous media. As for the pdf of the final sampled position, since we sampled an exponential distribution before sampling the equi-angular distribution function,

we had to normalize the equi-angular pdf with the exponential distribution pdf.

Another challenge in implementing this section was getting the positions of the point lights. Medium classes in PBRT do not have access to lighting information by default, so we created another integrator that copied most components of the VolumeIntegrator, but also passed in light information to Medium::Sample().

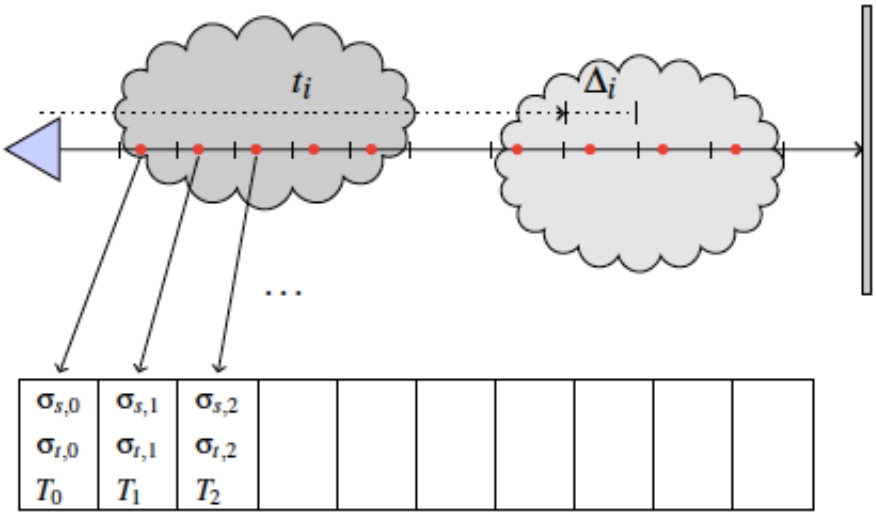
The following images were run with 256 samples/pixel, with PBRT's original homogeneous medium class (first) and our new homogeneous medium class using equiangular sampling (second).





Heterogeneous Media

The above method could be extended to heterogeneous media as well. According to Kulla et al., their method could be used with decoupled ray marching:



Where we can divide the ray into small segments that are assumed to be homogeneous. Kulla et al. also

suggest building a per ray data-structure that stored the $\sigma_{s,i}$, $\sigma_{t,i}$, and $T_i(t)$ of each segment i throughout the ray; that way the transmittance values could be looked up with a O(log(n)) binary search for each shadow ray that's traced.

Implementation in PBRT & Challenges

PBRT's heterogeneous medium class uses delta tracking over a 3D grid of volume density values. We created another heterogeneous medium class based on this class, but used ray marching instead. However, PBRT's volume integrator does not trace multiple shadow rays for a ray that goes through a medium. Instead, it takes one sample per ray via Medium::Sample() and continues to trace more rays from that sampled position. If the sampled position was a medium interaction, the volume integrator spawns a ray in a direction sampled by the phase function. If the sampled position was not a medium interaction, then the ray registers a surface interaction and spawns a ray from the surface. Since at most one shadow ray is traced per ray, it did not make sense to store an entire array of transmittance values as suggested in the paper in PBRT. Hence, we directly computed the transmittance of the sampled position via ray marching for calculating its weight.

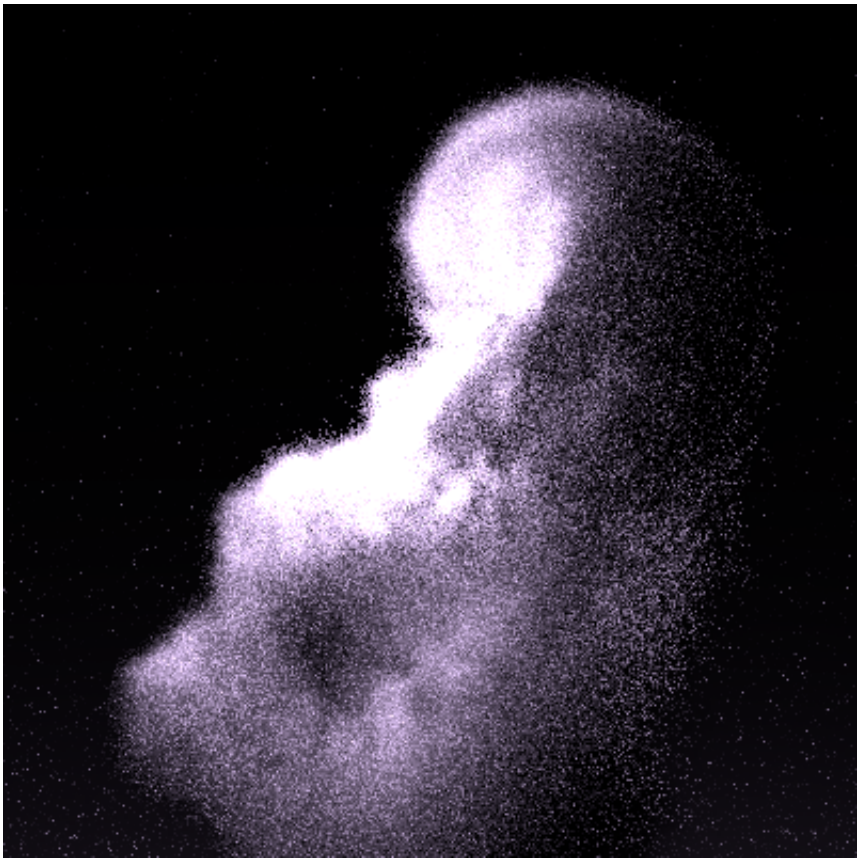
Our Medium class also takes in a 3D grid of volume density values. In ray marching, we'd compute sigma*s*i and sigma*t*i for each segment by multiplying the overall sigma*t and sigma*s by the density value for that segment. The density value of a location in the grid is computed by trilinear interpolation.

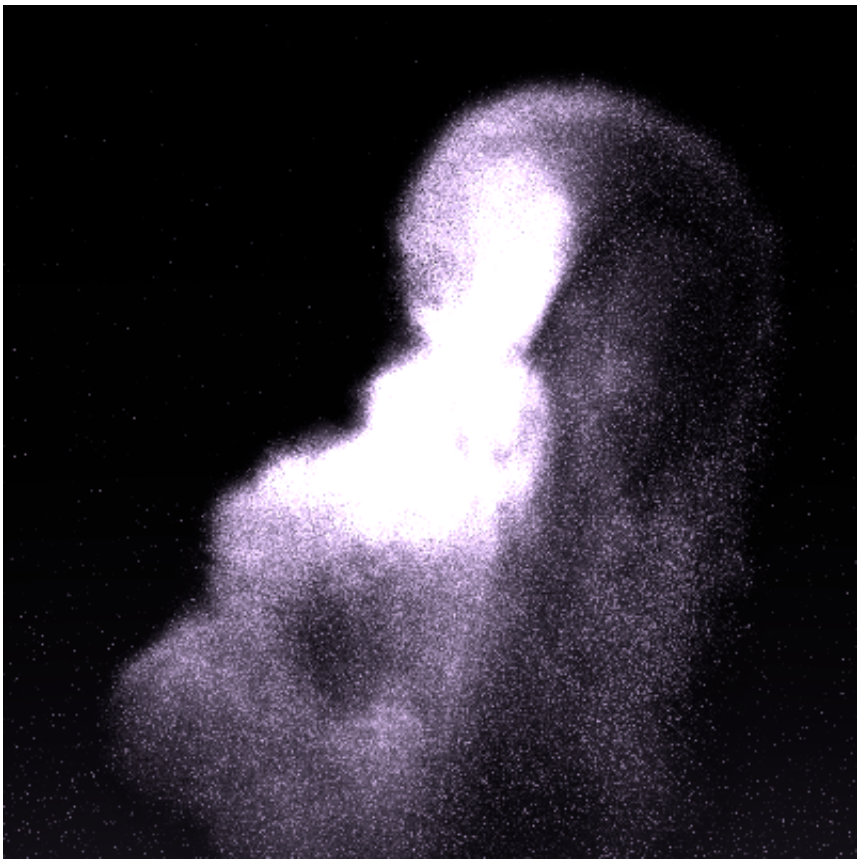
We extended our custom integrator to uniformly sample a light in the scene, and then uniformly sample a position on the surface of the light before passing it into the Medium::Sample() function. We implemented equi-angular sampling with respect to the location of the sampled light surface. Point lights and triangular area lights are supported. As equi-angular sampling involves computing angles with light sources, it does not work with infinite lights. So our integrator ignores infinite lights when sampling light positions to pass into Medium::Sample().

While equi-angular sampling works well for heterogeneous media with embedded light sources, it does not work as well for heterogeneous media when the light sources are outside of the volume. In section 4.2 of the paper, Kulla et al. suggested using another discrete density PDF to distribute samples within the medium. Figure 8 of Kulla et al's paper shows examples of when equi-angular sampling works better than discrete density, and vice versa. Due to time constraints, we were not able to implement this part of the paper.

One bug we ran into when implementing the equi-angular method was seeing a dark border around the steam. This occurred because some of the sampled positions in the volume grid had zero density, meaning that the scattering coefficient for the sampled position was zero. Hence, this sample yielded a weight of zero (transmittance * scattering / pdf), and subsequent bounces from this sample was classified as having zero contributions to the final radiance. While the zero contribution is technically correct, samples such as these are not medium hits and should not have a MediumInteraction created. This was fixed by checking the density of the sampled position before returning. If the density was zero, then we do not register a medium hit.

The following images were rendered with 64 samples/pixel using this scene (updated); PBRT's method (first) and equi-angular sampling (second). The equi-angular sampling method captures the embedded light source better than PBRT's delta tracking method. The contrast is better shown in .exr format (see here & here).





Simulation and Modeling

Modeling

We made all of the models in our scene using Blender and exported the scene to PBRT using Cinema4D.

Teacup

We started constructing our scene by modeling the teacup: it is the simplest model in our scene and the main focus of the image. To create the mesh, we traced the cross section of a teacup from a reference image and spun it 360 degrees, and we subdivided the mesh twice. The UV map was created so that the inside, bottom, and outside surface are connected components. This would ensure that our scratched metal texture runs seamlessly along the side of the cup.

Table

The table was inspired by this picture of an oriental tea tray. Rectangular cubes were created for each of the wooden beams that make up the tray. Their edges were beveled and randomly rotated slightly to add more

realism to the scene. The cylinders that run across the tray were created with Blender's array operator, which duplicates a mesh to your specifications. The cylinders and the rest of the table are treated as separate objects due to the difference in the geometry.

Teapot

The teapot started off as a normal cup during the initial stages of our scene. Adding a spout to the scene proved to be complicated; surely there is a better method that we did not attempt. We spun the spout to create a cylinder, and then we used Blender's proportional editing features to sculpt the shape of the spout. Connecting the spout to the main body of the teacup, we realized that this method would not work visually since there was a discontinuity in the connection between the spout and the body. We solved this issue by using Blender's Boolean operator to compute the union of the two meshes, decimating the joined mesh, and subdividing the new mesh. This resulted in a mesh with few visible artifacts between the spout and the body.

Simulation

Tea

The tea was simulated using Blender's fluid simulation system to achieve the most realism. Blender uses the Lattice Boltzmann Method. Simulating the fluid on a grid of width 500, we have an icosphere set to inflow near the spout and a cube set to outflow inside the cup, and we give the inflow a bit of initial velocity. After baking the fluid with 1e-6 viscosity, we obtain plausible water. Noticing that the water was not quite as realistic, we rotated it to be more vertical and placed it in the correct position manually, and then we subdivided it twice.

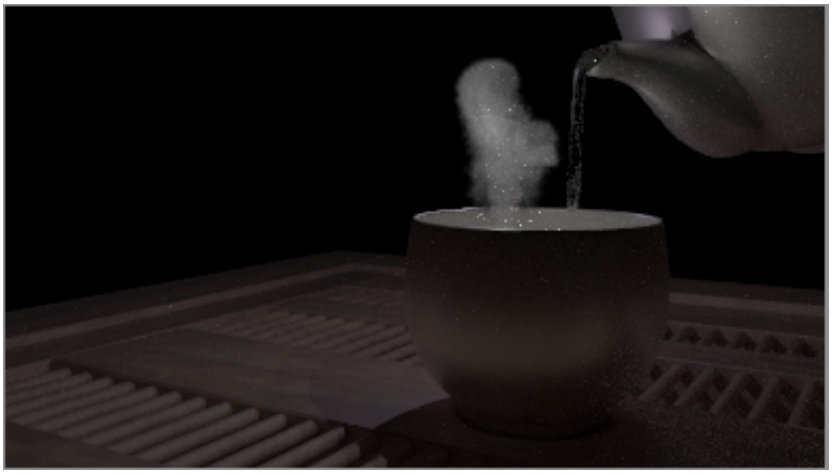
Rendering the Final Image

Final Image

Brushed metal has poor conditioning with point lights, often producing black and white images due to oversaturation. It works best with environment maps. Equiangular sampling, however, cannot handle infinite lights due to its reliance on equiangular sampling, and shines best with point lights. Thus, combining the two in the same scene poses problems. We had significant trouble balancing parameters to minimize noise in both the metal and steam. Given more time, we would have liked to create a cohesive scene that showed off both techniques.

For our final image, we took the best of both worlds. We had both an environment map and several point lights in and near the steam. The brushed metal in the scene ignored the point lights, and the steam ignored the environment map.

Example image of environment map and brushed metal taken away:



Contributions

Sean: Implement sampling technique for participating media to efficiently render steam.

Toki: Implement 4D Gaussian mixtures to capture specular microstructure.

Wilbur: Model the scene with Blender / Cinema 4D, run fluid simulations with Blender.

Works Cited

Chen, Shiyi, and Gary D. Doolen. "Lattice Boltzmann method for fluid flows." Annual review of fluid mechanics 30.1 (1998): 329-364.

Kulla, Christopher, and Marcos Fajardo. "Importance sampling techniques for path tracing in participating media." Computer Graphics Forum. Vol. 31. No. 4. Blackwell Publishing Ltd, 2012.

Yan, Ling-Qi, et al. "Position-normal distributions for efficient rendering of specular microstructure." ACM Transactions on Graphics (TOG) 35.4 (2016): 56.