Yūbinkyoku
A photorealistic path tracer

CS488 A5 Project Report

Name: Tristan Hume

July 24th, 2018

http://thume.ca/ray-tracer-site

Contents

1 Introduction

My goal for the project was to make a ray tracer capable of rendering photorealistic images at the level of modern renderers like Blender's Cycles, at least for a limited domain of object types. I wanted to then use it to render a final scene that looked like a photo except for some whimsical elements of the scene that clearly couldn't be real.

I attacked my goal by researching the most important factors in the realism of renders from modern renderers and came up with three main factors: Path traced global illumination, high quality physically based materials, and good High Dynamic Range (HDR) tone mapping. So I remade my A4 ray tracer around those 3 things as well as a number of other features necessary to support them in rendering a good scene.

I put about a full time month's worth of work effort into this project, since I was having a lot of fun and had a Final Scene in mind I wanted to reach. I tracked much of my time with a time tracker as I usually do for homework, and included the results in an appendix in case that's interesting.

My project is named after the Japanese word for "post office" because it made so little sense as the name for a ray tracer that it made me burst out laughing, and the fact that there's an emoji for it sealed the deal. Why the idea of naming it after the Japanese word for "post office" occurred to me in the first place is a long story.

You can look at my brochure site at any time at http://thume.ca/ray-tracer-site.

2 Primary Objective Details

2.1 Texture mapping

Texture mapping [1] allows varying material properties, most notably the colour, over the surface of an object. Standard 2D texture mapping of images involves computing a "UV coordinate" for each surface intersection, which corresponds to a position in the image where (0,0) is the bottom left of the image and (1,1) is the top right. When calculating shading, material attributes like the colour are looked up in an image texture associated with the material based on the coordinate.

But how do you map a floating point coordinate to a value in a discretized image? The simplest way to do it is called "nearest neighbour" or simply the absence of filtering, which simply takes the value of the nearest pixel in the image, rounding down. A better approach is bilinear filtering, which is a method of smoothing values out by averaging the 4 nearest pixels with weight proportional to how close they are to each point.

I implemented texture mapping by making my material take all of its attributes from a texture object that returns a 3D vector given a UV coordinate, and then implented the ability to create constant and image texture objects from Lua. Textures can be filtered either with bilinear filtering of the 4 nearest texels or nearest neighbour.

Coming up with the UV coordinates for different primitives is done in the ray intersection routine and for simple primitives is just done with math and some judgement about how to unwrap the surface. For spheres I use the polar and azumithal angle of the intersection (scaled for a maximum of 1) as the u and v coordinates of the intersection, which is the standard used for environment map image textures. For meshes, I use the CS488 framework to load OBJ files with coordinates for each vertex, then linearly interpolate the values for each vertex in barycentric coordinates to find the value for an intersection point on a triangle.

The image texture Lua function has options for applying inverse gamma correction, scaling coordinates and scaling values.

Along with texture mapping, I added support for emission textures and rendering inside spheres to support environment maps for realistic lighting and reflections on shiny surfaces.

Here's an example of using the new Lua `gr.material` command and texture creation functions:

```
uv_scale = 0.5
metalplate = gr.material({
  -- The final 1.0 parameter below is value scale, the true enables inverse gamma
  albedo = gr.image("tex/Metal_Plate_007_COLOR.png", true, uv_scale, 1.0),
  normal = gr.image("tex/Metal_Plate_007_NORM.png", false, uv_scale),
  -- Often internet roughness textures are actually glossiness, I fixed some
  roughness = gr.image("tex/Metal_Plate_007_ROUGH_fix.png", false, uv_scale),
  -- Passing one parameter to gr.const replicates that number for RGB
  metallic = gr.const(1.0),
})
```

2.2 Normal mapping

A normal map [2] is a type of texture that influences the surface normal direction used for lighting calculations. It can yield a convincing effect of small surface variations without adding extra geometry.

The normal map is read for a given shading point just like any other texture map, this gives it a colour. The convention for normal maps is that the colour encodes the perturbed normal in an orthonormal basis of the geometry tangent vector, bitangent vector and normal vector for xyz. The tangent vector is the vector tangent to the surface in the direction of increasing U coordinates, the bitangent is increasing V and can be computed as the cross product of the other two vectors.

The first step to implementing normal mapping was to add tangent vectors to my primitives, I did so for meshes, cubes and spheres. Spheres and cubes were math like the UV coordinates, but for meshes I had to do a preprocessing pass where I compute tangents for each face using the numerical derivative of the UV coordinates over the triangle in barycentric coordinates. At intersection time I then simply look up the tangent for the current face.

Next I needed to use that to perturb the geometry normal, which I did by first converting the colour mapping numbers to a normalized vector in the normal space by remapping from the range [0,1] to the range [-1,1] and then normalizing. I then multiply by a change of basis matrix to get the surface normal $\vec{n}_s$ from the map normal $\vec{n}_m$:

$$\vec{n}_s = \begin{bmatrix}\vec{t} & (\vec{n} \times \vec{t}) & \vec{n}\end{bmatrix} \vec{n}_m$$

2.3 Path tracing

I implemented global illumination using path tracing [3]. Path tracing is a method of producing an unbiased estimate of the result of the rendering equation [3] using Monte Carlo integration. In its simplest formulation, it does so by scattering a new ray in a random direction at every surface interaction and when it eventually hits a light multiplying the light emitted along the ray with their "throughput": the product of the BRDF and projected area cosine terms of all the interactions up to that point.

My implementation is based on the path integral formulation of the light transport equation introduced in Eric Veach's PHD thesis [4]. In the formula for throughput below, $f(p_{i+1} \to p_i \to p_{i-1})$ is the BRDF evaluated for a ray coming from p_{i+1}, scattering at p_i and going out to p_{i-1} and $G(p_{i+1} \leftrightarrow p_i)$ is the amount of light that is transmitted between those points.

$$\begin{aligned}
T(\bar{p}) &= \prod_{i=1}^{n-1} f(p_{i+1} \to p_i \to p_{i-1}) G(p_{i+1} \leftrightarrow p_i) \\
P(\bar{p}) &= \int_A \cdots \int_A L_e(p_n \to p_{n-1}) T(\bar{p}_n) dA(p_2) \cdots dA(p_n)
\end{aligned}$$

In practice the simplest formulation takes forever to converge so there are a number of optimizations that are effectively mandatory. The most important is sometimes called next event estimation but is often considered just an essential part of path tracing, which is terminating paths by sampling a ray pointing directly at a light source from a surface interaction, instead of waiting until one is hit by chance. I implement this as well as a further optimization where I do this at every interaction along the path and add that contribution to the current pixel, as well as continuing along with further scattering, this effectively allows re-using traced rays for computing lighting at multiple levels of indirection.

In order to scatter by direction on intermediate bounces but terminate by sampling a point on the light p_i, we can reformulate the throughput formula into the following one [5] with a direct lighting factor $D(\bar{p}_n)$ and a throughput $T(\bar{p}_n)$:

$$\begin{aligned}
D(\bar{p}) &= \frac{L_e(p_i \to p_{i-1}) f(p_i \to p_{i-1} \to p_{i-2}) G(p_i \leftrightarrow p_{i-1})}{p_A(p_i)} \\
T(\bar{p}) &= \prod_{j=1}^{i-2} \frac{f(p_{j+1} \to p_j \to p_{j-1}) |\cos\theta_j|}{p_\omega(p_{j+1} - p_j)} \\
P(\bar{p}) &= D(\bar{p}) T(\bar{p})
\end{aligned}$$

Another important optimization I implement is russian roulette termination, which randomly stops tracing rays with probability inversely proportional to their throughput, so less time is wasted bouncing rays that can contribute only very little light. In order to maintain lack of bias, the samples that do continue then need to be re-weighted by dividing by the PDF of the sampling method used for termination, so that they have the same probabilistic expected contribution as a uniform sample.

The final important optimization is importance sampling. Choosing a random scattering direction in a hemisphere works okay for diffuse materials but it is vanishingly unlikely to sample a direction inside a specular lobe of the BRDF at a surface, so highlights and reflections will be lost or very noisy. Importance sampling solves this using a similar trick to russian roulette, by sampling a scattering direction using a distribution with higher probability where the resulting throughput will be higher, then re-weighting using the PDF that the sample was taken from.

The simplest type of importance sampling is getting more efficient diffuse samples by sampling from a cosine-weighted hemisphere instead of uniformly, to account for the varying throughput because of the cosine term. I do this using a trick from PBRT [5] by sampling a unit disk and then projecting up onto a hemisphere, which yields a cosine distribution. I'll discuss another type of importance sampling I do in the glossy reflection section.

Together these allow reasonably efficient convergence of global illumination in many scenarios, although lots of samples are still required for a noise-free result.

2.4 Soft shadows

I implemented soft shadows by overhauling my lighting model so that lights are just spheres with an emission texture on them, just like environment maps. I then changed the lights array normally passed into the render to just a list of GeometryNodes to sample explicitly. The path tracing integrator has special code that adds the contribution from the emission texture if it hits one, while checking that either it is the first interaction or a non-sampled light, so that it doesn't double count the light it samples using next event estimation.

This also allows me to render the area light as a physical object that exists in the world, or I can put it in the list to be sampled but not in the scene for an invisible light.

The path tracing integrator when doing next event estimation at each bounce samples the cone of directions subtended by the sphere that would fit in the bounding box of the objects, then re-weighting by the uniform cone PDF, using equations from PBRT [5]. This means at the moment only spherical area lights work correctly. This is more efficient than sampling a point on the sphere since only visible positions on the light are sampled.

This sampling method is somewhat involved, but briefly it works as follows:

1. Calculate the angle subtended by the sphere as visible from the interaction point θ_m.
2. Sample the cone of directions with that maximum angle using a uniform ϕ and a θ based on the cone CDF $(1 - \xi) + \xi \cos(\theta_m)$

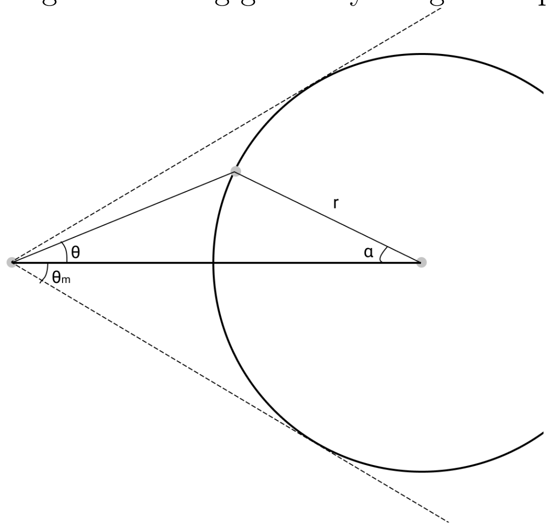
3. This gives a direction towards the light, but we want a point on its surface, so then we use the Pythangorean theorem to find the distance to the surface of the sphere.

4. Use the law of cosines to find the angle α from the interaction→center vector $\vec{w}_c$.

5. Use α, ϕ, and an orthonormal basis formed using $\vec{w}_c$ and cross products to get a normal of the sphere towards the sampled point.

6. Use vector math $r\vec{n} + c$ to get the point on the sphere surface.

Figure 1: Diagram showing geometry of light sampling



2.5 Glossy reflection

Glossy reflection falls out directly from the path tracing integrator and the specular lobe of the Disney BRDF I use (see extra objectives), which uses an empirically based GGX [6] (also known as Trowbridge-Reitz) distribution for the microfacet normals. However, without importance sampling the specular lobe of the BRDF, reflections would be hopelessly noisy.

I importance sample the specular lobe by sampling microfacet normal directions from the GGX distribution term of the specular lobe using the CDF derived in the original Disney BRDF paper [7]:

$$\phi = 2\pi\xi_1$$
$$\cos\theta = \sqrt{\frac{1-\xi_2}{1+(\alpha^2-1)\xi_2}}$$

I then reflect the outgoing light direction across this sampled normal to get the sampled incoming direction and remap the PDF used for weighting accordingly.

With path tracing the only difference between specular highlights and glossy reflection is whether the light was directly sampled!

2.6 Anti-aliasing

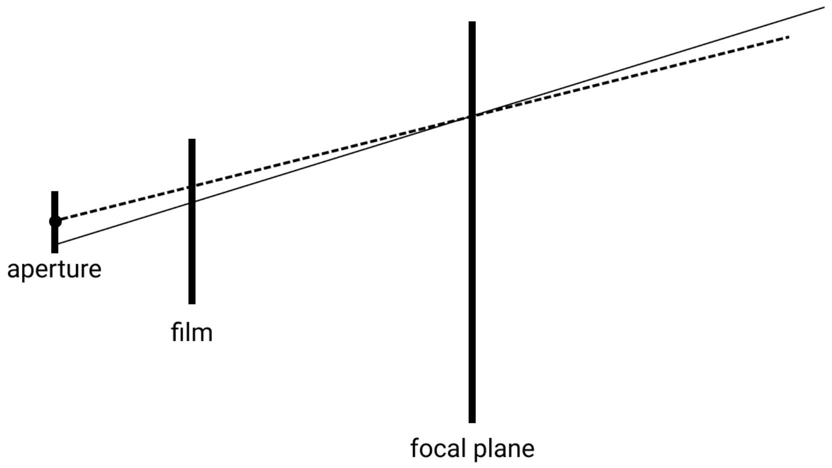
Anti-aliasing is implemented as part of the same sampling framework that the path tracing uses. On each sample a random position in a square pixel region is picked to cast the ray for.

For textures I only do bilinear filtering with 4 texels so for high resolution textures anti-aliasing is also necessary for textured surfaces to look correct. In my draft proposal I had adaptive anti-aliasing as an objective but I replaced it in my final proposal because I realized the fact that my planned final scene would use a lot of textures would mean practically every pixel would need significant antialiasing.

2.7 Depth of field

I implement the thin lens model of depth of field [8] by finding where the original ray intersects the focal plane, then sampling a random point on a disk representing the aperture of the camera and casting a ray from that point to the point on the focal plane.

Figure 2: Diagram showing geometry of new ray calculation



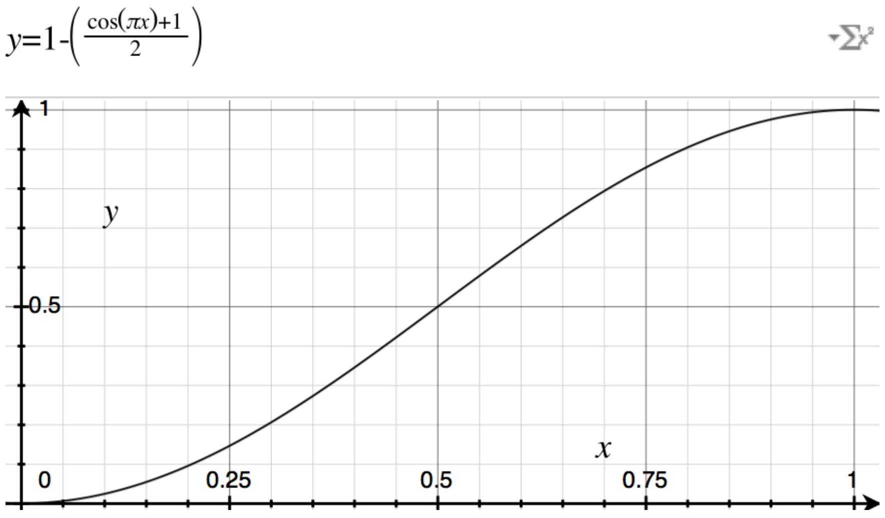
The aperture is communicated to the renderer using an extra optional parameter to the `gr.render` lua command. The focal length is taken from the length of the view direction vector that was already a parameter. This way if the view direction is computed as the difference between the eye point and a look at point, the focus will be on the look at point.

2.8 Animation

I implemented animation in Lua by constructing multiple scenes and rendering them to separate files, then using FFMPEG to stitch them together. The animation is done with math on a time parameter ranging from 0 to 1, logic to figure out which phase of the animation the parameter corresponds to, and a cosine-based curve for smoothly easing in and out as suggested by the seventh principles of traditional animation [9].

Do note that I can't take credit for the metal and plastic hex bars aesthetic, which was taken from another animation I found, see the Credits section for more details.

Figure 3: Plot of easing function used

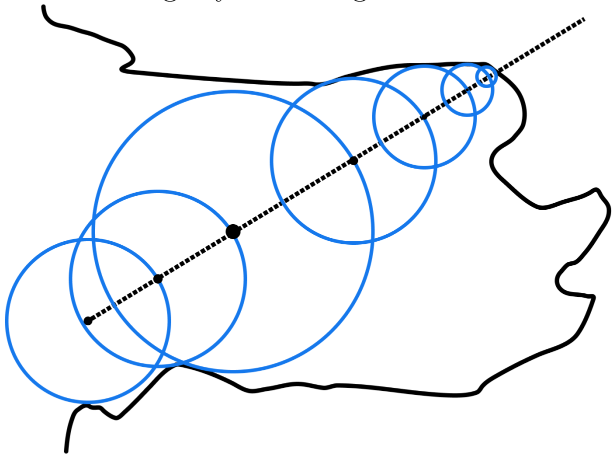


2.9 Constructive Solid Geometry

I implemented Constructive Solid Geometry (CSG) in a different way than is standard for ray tracers. I implemented a primitive for ray marching a distance field [10]. This is a method of rendering a 3D surface given a function from a point to the distance to a surface by taking steps along a ray based on the function.

Basically, given a function that gives the distance to a surface, every step you evaluate the function at a point on the ray, then step that distance along the ray since you're guaranteed that there's no surface closer than that distance by the function, you iterate this until you get within ϵ of a surface or you reach a maximum distance from the scene beyond which you know there is nothing. The normal can then be found using the finite difference method on the distance field.

Figure 4: Diagram illustrating ray marching a distance field indicated by circles



Once that is implemented, CSG is simple to implement by taking the minimum (union) or maximum (intersection) of two distance functions, subtraction is taking the maximum with the negative of a signed distance function.

There's a number of simple distance field tricks that produce aesthetically nice results that are hard to do with other methods of rendering. One of these is Inigo Quilez's smooth minimum operator, which produces rounded corners when doing CSG instead of sharp ones by blending the functions together near the intersections. I modify his hex bar primitive to use this operator to have smooth corners for use in my animation.

Another trick is domain repetition, which uses floating point modulus to map points back to a limited box, which leads to an infinite repeating field of that object when you trace rays using it. This is only gives correct distances if the object is mirrored across all sides of the repetition, which my hex bars are not since I rotate them. Uncorrected, this leads to rendering artifacts in some places, I avoid those artifacts using a trick for distance estimators which are wrong by a bounded amount, which is stepping only a fraction (I used 0.75) of the given distance along the ray.

2.10 Final scene

My goal was to render a scene that was as photorealistic as possible while having whimsical elements that clearly couldnt be real. The scene shows off high quality textures and models (not done by me) in a scene I arranged and tuned the materials for. I also tried to compose an artistically nice scene that tells a story. It uses a portal, a Mandelbox, global illumination, and various hand-tuned Disney BRDF materials.

I chose to do a scene of a desk with a MacBook on it, surrounded by various typical desk objects as well as a Minecraft poster and Lego creation. The MacBook screen would actually be a portal (see extra objectives) to a rendered Minecraft world, with a Minecraft character and pig stepping out of the screen onto the MacBook's keyboard. I liked the concept because I thought it would make for an interesting juxtaposition between the photorealistic desk scene and the blocky pixelated characters becoming real, with a bit of an ironic message that graphics (like Minecraft's) can be captivating without going for realism at all. I also included a Mandelbox fractal desk sculpture as another whimsical element that couldn't really exist in real life. I decided to fill some space on the desk with a book and figured that PBRT [5] would be fitting since it taught me so much, and conveniently the PBRT website had a 3D model of the book! I named the scene "Realism" as a dual nod to the photorealism and the content being the characters becoming real.

While I didnt make any of the models and textures except the desk and Mandelbox base, I still spent over 15 hours working on this scene. I spent the time in Blender and other tools arranging objects, designing materials (no models came with Disney BRDF materials) and lighting, composing the scene, posing characters, trimming geometry, and importing various model formats including a Minecraft world and Lego model. I didn't time it but I also probably spent another 10 hours watching Blender tutorials on YouTube, which taught me how to use Blender at all, and also some things about composing scenes and how to achieve photorealism (e.g. it's how I learned about filmic tone mapping).

3 Extra Objective Details

3.1 Disney BRDF

All my materials use an implementation of a subset of the Disney BRDF [7] (also called the Principled BRDF, or the metallic-roughness BRDF). This is a high quality physically based microfacet BRDF including proper Fresnel and other effects for dielectrics and metals. My implementation is a subset of the full material with three parameters: base colour, metalness and roughness. The BRDF takes the form of a standard isotropic microfacet BRDF:

$$f(l,v) = \text{diffuse} + \frac{D(\theta_h)F(\theta_d)G(\theta_l,\theta_v)}{4\cos\theta_l\cos\theta_v}$$

The D, F and G functions are the microfacet normal distribution, fresnel and masking/shadowing functions respectively. This simulates a rough surface statistically as many small mirror-like microfacets, and models the proportion of light reflecting off them and being shadowed by them. The roughness controls the variance of the distribution of microfacet normals, and the metalness controls whether the material acts like a dielectric with diffuse colour and untinted specular, or a metal with tinted specular and no diffuse term. I used the custom diffuse, GGX normal distribution and Schlick Fresnel equations from the original paper, and the masking/shadowing equation G for the GGX/Trowbridge-Reitz distribution came from PBRT [5] since it wasn't in the original paper.

After implementing the Disney BRDF, I rendered a simple scene with both Blenders Cycles renderer and mine had edges that were slightly too dark on metallic surfaces. After much debugging I found that the G formulation from the Unreal engine [11] I was previously using didn't match the original formulation that Cycles used, so I switched. After that I had two Cornell box images which were a perfect match, which gave me confidence and allowed me to tune materials in Blender.

3.2 Phong Shading

To implement Phong shading, I simply interpolated the per-vertex normals using the barycentric coordinates in the same way as the UV coordinates, although I implemented Phong shading first.

3.3 Filmic Tone Mapping

An important part of the photorealism of my renders is very high quality tone mapping. I implemented a parser for the 1D and 3D OpenColorIO lookup table (LUT) formats, and a function to apply them correctly with linear and trilinear interpolation respectively.

The good tone mapping allows for scenes with realistic high dynamic range lighting to render properly without getting blown out and with photo-like colour. In renders of my final scene with no tone mapping or gamma correction the colours are bad, the sun on the keyboard is totally blown out, and the shades of black on the display rim are lost.

I can also use different LUTs for different scenes. I rendered my final scene with a more camera-like LUT with lots of dynamic range, but the rest of my scenes with a higher contrast LUT that gives punchier colours but isnt as photorealistic.

I can also use a false colour LUT to see what the dynamic range of my scene is like.

Part of the tone mapping is a 3D LUT that desaturates very bright colours, simulating bleed between film layers, which can produce a pleasing effect with very bright colours getting desaturated. This effectively allows for more dynamic range to be shown.

3.4 Mandelbox Fractal

Using distance field ray marching it is possible to render Mandelbrot-style iteration fractals by using the running derivative of the iteration as a distance estimate [12]. I implemented the distance function for one such fractal, the Mandelbox.

I then used CSG to subtract and intersect with some cubes to carve out an interesting region inside the fractal, since the outside isnt too impressive.

3.5 Portals

In order to implement my final scene concept I needed portals, which for my case meant a way to have an object act as a portal where rays passing through it would end up in a different scene, bidirectionally.

I implemented this by giving all rays and surface hits a world number field. Then I modified my code so all scattering and light sampling would observe the world field and cast follow up and shadow rays with the same world.

Then I added a portal SceneNode subclass that checks the world ID of incoming rays and first tests it against that index of its children, as well as a special portal node pointer, and if it hits the portal before it hits the current world, it spawns a new ray in the next world number modulo its number of children, and does the required changing and remapping ray t values to spawn the new ray at the portal.

3.6 Blender Export

I put my final scene together in Blender and it took over 15 hours to do, if I had to do it entirely with conversion scripts and lua files it would have been infeasible. So I wrote a script using Blenders Python API that exports all objects in the scene into OBJ files and a lua file that references them all and sets them up with the right materials.

I can then reference the objects defined in this lua file in my final scenes lua file, while adding arrangements like portals that Blender cant do. It also exports object transforms separately so I can do things like replace a cube with a Mandelbox.

I used the same set of film emulation LUTs included in Blender so that I could match Cycles almost exactly.

3.7 Low Discrepancy Sampling

I implemented sampling using the Sobol (0,2) sequence with a fast gray code technique from the PBRT [5] book. This is a sequence defined as a specialized binary optimization of the radical inverse method of sequence generation. These samples have a more even (lower discrepancy) distribution and can be drawn from progressively and efficiently, leading to faster render convergence rates.

I first implemented a `Sampler` container class which stores a 2D array of samples, where the indices are the sample number for the given pixel, and the dimension within the current sample. A dimension is an individual sample like the position in the pixel or aperature, or the scattering direction. The `Sampler` stores 8 dimensions per sample and generates further dimensions uniformly at random.

I then implemented a `SampleGenerator` class that stores a Sobol (0,2) sequence sampler state, which is two 32 bit integers for each of the 8 dimensions. These states are initialized with a random number, which represents a scrambling of the initial state so that different pixels and dimensions use different samples, then updated throughout the entire render using an efficient gray code based technique for sampling the sequence while filling a buffer.

Once the `Sampler` buffer for a given pixel's samples has been filled, each dimension is shuffled independently. This is done since while the initial scramble effectively mixes around the range of the sequence, samples from different scrambles will still be drawing from the same sequence and thus correlated, for example you might have rays that always bounce down on the second bounce when they bounced up on the first bounce, leading to biased results. The shuffle wrecks this correlation across dimensions while still maintaining the distribution within all the samples.

3.8 Progressive Rendering

I made my renderer progressive by saving a matrix of (0,2) sampler states and running my sampler in epochs of gradually increasing sample count, while keeping the samples for each pixel in each epoch in the innermost loop for cache/branch prediction efficiency. Every epoch it saves an image of its progress so far.

3.9 Multithreading

I made my renderer multithreaded using all available hyperthreads using the C++11 threading library. It distributes rows of image alternately to each thread for evenly distributed difficulty. I implemented a multi-threaded progress bar with proper locking to print nice console progress with ANSI escape codes.

It scales linearly with number of cores. I rendered most images on this page on a 64 core pre-emptible Google Cloud instance for only $0.50 an hour. For all the renders in my brochure plus some more I didn't include, often rendered with way more samples than necessary since I was creating the scenes at the same time and had nothing better for the machine to do, I used less than 20 hours of rendering time on the cloud machine, most of which was spent on the animations.

3.10 Miscellaneous Acceleration

I implemented a number of optimizations so that I could render large scenes with path tracing in a reasonable amount of time. I dont have graphs because these are mostly asymptotic or varying by scene so I could make the graphs show whatever I wanted. The first optimization on this list made it possible to render large meshes at all, and the rest gave noticeable speedups depending on the scene.

- So that I could render large meshes, I integrated the FastBVH bounding box hierarchy library into my mesh primitive. This took about 2 hours.
- I implemented my own bounding box hierachy checks at the scene node level, although that hierarchy is just the manually constructed hierarchy from Lua.
- I baked the hierarchical transform down into each GeometryNode and only apply the transform if it is not the identity, to reduce the number of matrix multiplies per ray per object.
- When implementing formulas, I follow best practices for floating point performance and avoid or precompute square roots, trig functions and divisions wherever possible using a number of standard tricks.
- I use the Moller-Trumbore ray-triangle intersection method which is just an algebraic improvement on the one presented in class, but is much faster.

Together, these make my renderer nearly as fast as Blender's Cycles renderer.

4 Bibliography

[1] J. F. Blinn and M. E. Newell, “Texture and reflection in computer generated images,” *Communications of the ACM*, vol. 19, pp. 542–547, Oct. 1976.

[2] J. F. Blinn, “Simulation of wrinkled surfaces,” *ACM SIGGRAPH Computer Graphics*, vol. 12, pp. 286–292, Aug. 1978.

[3] J. T. Kajiya, *The rendering equation*, vol. 20. ACM, Aug. 1986.

[4] E. Veach, “Robust monte carlo methods for light transport simulation,” 1997.

[5] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 3rd ed., 2016.

[6] B. Walter, S. R. Marschner, H. Li, and K. E. Torrance, “Microfacet models for refraction through rough surfaces,” in *Proceedings of the 18th Eurographics conference on Rendering Techniques*, pp. 195–206, Eurographics Association, 2007.

[7] B. Burley, W. S. A. SIGGRAPH, and 2012, “Physically-based shading at disney,” 3.

[8] R. L. Cook, T. Porter, and L. Carpenter, “Distributed ray tracing,” in *ACM SIGGRAPH computer graphics*, vol. 18, pp. 137–145, ACM, 1984.

[9] J. Lasseter, “Principles of traditional animation applied to 3d computer animation,” in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987, Anaheim, California, USA, July 27-31, 1987*, pp. 35–44, 1987.

[10] T. Reiner, G. Mückl, and C. Dachsbacher, “Interactive modeling of implicit surfaces using a direct visualization approach with signed distance functions,” *Computers & Graphics*, vol. 35, pp. 596–603, June 2011.

[11] B. Karis and 2013, “Real shading in unreal engine 4,” *gamedevs.org.*

[12] T. McGraw and D. Herring, “Shape Modeling with Fractals.,” *ISVC*, vol. 8887, no. Chapter 51, pp. 540–549, 2014.

5 Appendices

5.1 Credits

This is an overview of all the things I didn’t do myself and the sources I used for them.

- I used the FastBVH library found on Github for my mesh acceleration, but had to integrate it myself.

- I learned a lot of what I know from sitting down and reading the PBRT book [5], many parts of my ray tracer are based on reading the theory and rationale for doing something a certain way in my paper copy of the book while reading it start to finish, and then coding up my own version later. Things built following the methods from PBRT include my path tracing, importance sampling, area light sampling, depth of field and my low-discrepancy sequence generation.

- I copy-pasted the following small code snippets from the PBRT renderer because they are either mostly constants, or there was no description of the theory from which I could write my own: Inverse sRGB mapping, The Sobol (0,2) sequence matrix constant arrays and constructing an orthonormal basis from a single vector.

- The primitive signed distance function equations I use for modeling including the box, sphere, ellipsoid and hex bar all come from Inigo Quilez's SDF modeling website.

- My Mandelbox distance estimator is based on code from a blog post by Mikael Hvidtfeldt Christensen explaining the theory behind the distance estimator. It is nigh-impossible to implement a Mandelbox estimator without referencing code since the way it is designed is based on aesthetics and approximations that don't always have a perfect theoretical basis.

- All the models and textures in all my scenes except the Mandelbox pedestal and desk were downloaded from the internet, although I re-did the materials myself using the Disney BRDF. The Minecraft hill was sculpted by me in Minecraft and then exported using the jmc2obj tool.

- The Lego in my final scene was based on a model found on the Mecabricks Lego CAD site, simplified a bit by me, and then exported.

- The Film emulation LUTs I load in for tone mapping come from Troy Sobotka's Filmic Blender addon, which was later incorporated directly into Blender. I also had an email conversation with Troy where he clarified some of my understanding of how the different colour spaces fit together.

- The idea for my animation was based heavily on an animation from the /r/blender subreddit which depicted deforming copper and black plastic hex bars moving in a sine wave pattern. I can't take credit for the idea of that aesthetic.

  See https://gfycat.com/WideeyedAstonishingDonkey

- I learned how Moller-Trumbore triangle intersection works on http://www.scratchapixel.com/

- I learned some basics of physically based rendering and normal mapping on https://learnopengl.com/

- I wrote this report entirely from memory and referencing my own source code so that I could be sure I wasn't too directly imitating anyone else's descriptions, but I did look at PBRT for the path tracing equations. Also while I drew the diagrams myself without reference, they're all based on diagrams I've seen before.

5.2 Time Spent

Below is a task list with times I spent on different tasks during the course of the project. It's copy-pasted from my personal planning markdown file. It's not important, I did it mostly for my own reference, I just figure since I already have it it might be interesting to know what took lots of time and what didn't.

The times listed on the task list significantly underestimate the time I spent on the project. I used a time tracking app to track the time I was sitting down accomplishing things, and tracked a total of 73 hours there before starting this report. I transcribed some of those times when I was working on a specific task onto the list after I finished them, but missed some time and didn't put time spent on miscellaneous tweaking, Blender-ing and debugging in the list. An additional 18 hours of work went into A4 and the extensions I did before submitting that, including CSG, which were used as the base for the project. Finally, I spent an additional 5 hours on this report not included in other numbers because I'm adding this in right before printing.

Most notably these times don't include the many many hours I spent on research and learning, reading websites, graphics papers, and nearly the entire PBRT [5] book from start to finish, planning out what I needed to do when I next sat down, and reading lots of papers for techniques that I might want to implement, most of which I didn't. I probably spent another 70+ untracked hours on this, and it was crucial for being able to sit down and accomplish things immediately because I already understood what I needed to do.

```
Done as part of A4, not itemized by time:
+ Constructive Solid Geometry
- Mandelbox
- Multithreading
- Fancy progress bar

Done after submitting project proposal:
- Set up basic build in new directory
- New fancy mesh type (1h)
     - Phong interpolated normals
- Acceleration structure (2h)
- HDR tone mapping (2h impl + 30m testing)
- Create new map-based material (1h refactor&lua)
  + Texture Mapping (1h for ImageTexture)
  + Normal Mapping (3h 30m for normal mapped meshes & cubes)
- Disney Principled BRDF Evaluation (2h + 1h testing)
- Check Principled BRDF against Cycles
- Investigate gaps in mesh (40m)
  - Use geometry normal for error displacement
- Progressive sampling infrastructure (1h)
  + Sampling-based Anti-aliasing
+ Depth of field (1h)
- Sobol (0,2) sampling (1h)
- Refactor to prepare for path tracing (40m)
- Cornell box scene (30m)
+ Path Tracing (1h)
  + Glossy Reflection
- Profile & Optimize (1h 30m)
- Bounding boxes in scene hierarchy (1h)
- Disney Principled BRDF Importance Sampling (1h 20m)
- Model Cornell box in Blender (30m)
- Debug edge darkening (1h)
- Cloud rendering (45m)
- Fix large render bugs (1h)
- Model more of desk/macbook (2h)
- Emitted light and bounce control support (30m)
- Refactor lights to actually be part of the scene (50m)
+ Soft shadows / Area lights (3h11m)
- Rendering inside spheres, environment emission objects, uv mapping spheres (2h)
- Studio scene (3h)
  + Animation
- Export basic geometry from Blender (3h)
- Fix boxes so can render inside them (20m)
- Export materials from Blender (1h 30m)
- Debug salt and pepper artifacts (1h)
- Portals (2h)
- Non-filtered textures (20m)
- Fix non-filtered textures (20m)
+ Final Scene
  - Minecraft (1h)
  - Add characters (2h 30m)
  - Book in scene (30m)
  - Mandelbox and stand (1h)
  - Stapler (1h)
  - Poster (1h)
  - Desk wood texture scale (15m - no change)
  - Lego (1h 20m)
- Minor optimizations (10m)
- Get rendering working on server again (50m)
- Make final scene sync-able (10m)
- Render final scene
- Small scenes to demo features (2h 30m)
  - texture and normal mapping examples
  - new cornell render with no soft shadows and no gloss
  - new cornell render with soft shadows but no gloss
  - cornell box with copper sphere and glossy base
  - textured diagonal cube against sky
  - depth of field test scene with studio base and environment map
- CSG demo animation (1h 20m)
  - Implement parameterized SDFs
- Nice abstract looping animation (2h 30m)
- Extra feature demo scenes (2h)
  - Portal Cornell box
  - Mandelbox
  - Non-tone-mapped final scene
  - Tone mapping desat demo
  - Non-phong shading
  - Low discrepancy sampling
  - Progressive rendering
- Brochure site (5h 25m)
```