'Brief' implementation overview

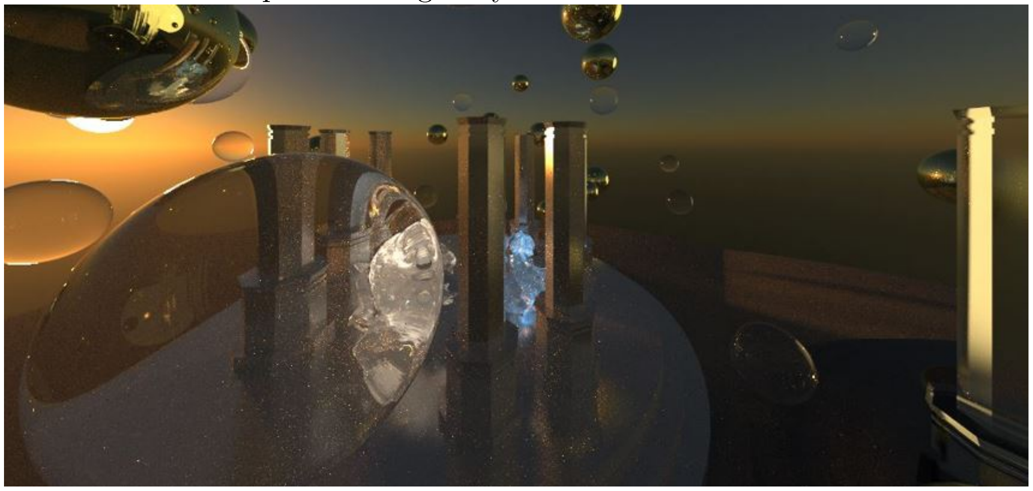
For my raytracer, I had long decided to do GPU stuff. I wanted to do it in WGSL/WebGPU because they are objectively the coziest. This will be a very meandering (sorry) discussion of the implementation details.

We build a BVH on the CPU with the following format: each node - situated in a contiguous array - is 64 bytes, split into 16 4-byte words. The first word is used to indicate whether the node is a leaf and, if it is, which of the primitives are circles. In leaf nodes, the next 12 words are used to store index data for up to four primitives (the three corners of a triangle or three 'vertices' representing the position, scale and rotation of a sphere). If it's not a leaf, we instead store the AABB's of the two child nodes and their indices. Note that, while traversing a ray in the bvh, the only time we need to 'branch' is when we might hit both children. In these cases, we need some way of remembering the branch we still need to explore after traversing the first. In a bvh with no more than two children per node, in the worst case, we will need to remember no more than N of these nodes where N is the depth of the BVH. Thus, rather than using recursion on the GPU (very scary) we can instead just keep a fixed-length (I chose 32 out of paranoia - even in scenes with hundreds of thousands of primatives, small numbers like 12 worked shockingly reliably and the max depth of my scenes was only 22 anyways) array of indices that we might need to explore later. Our implementation works by having a result buffer we constantly accumulate into which zero out whenever moving the camera or sun. We take 5-10 samples per pixel per frame to ensure the GPU is not too underutilized.

Next, to handle materials, we store a material index per vertex and assign each primitive to that of its first vertex. The material system is pretty generalized and can support basic diffusion and specular components as well as reflection, emission and transmission plus volumetric absorption/emission for translucent materials. The IOR and transmission roughness is also configurable. Each material can have any mix of these properties and up to 64 materials can be in use at once (this is arbitrary; you can increase this to however many 96 byte elements you can fit into a 64kb uniform buffer). I made a choice not to use any direct light sampling in this project - even the sun is just added on to atmosphere queries in its sundisk - to make caustics completely unbiased. With this scheme, glass and mirrors converge quickly but diffuse/specular surfaces do not. To assist convergence speed, using importance sampling, we sample towards the sundisk with some probability, otherwise, we use reservoir resampling on the diffuse/specular brdf. The reservoir resampling enormously improves the rate of convergence for materials with a specular component (like I was actually blown away when I first watched it). We return and accumulate $brdf(\omega, \omega_n)/pdf$ with our current transmittance in the outer loop. If this accumulated transmittance is low, we terminate the ray with probability $\frac{1}{2}$, otherwise double the transmittance. In this outer loop, we only need to keep track of position, direction, transmittance and (for later) absorption and emission, allowing us to once again avoid recursion (scary).

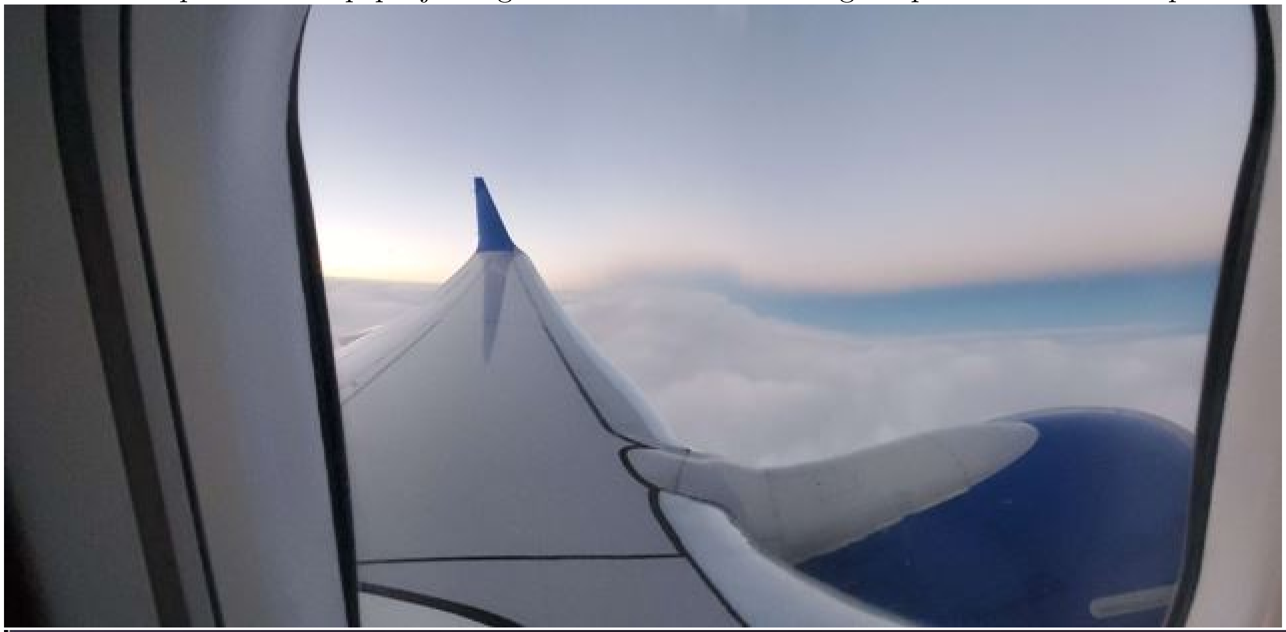
For the atmosphere, I took many ideas from "A Scalable and Production Ready Sky and Atmosphere Rendering Technique" by Sebastian Hillaire. I used two main ideas. First, by storing optical depth/transmittance parameterized by atmosphere height and sun direction, one-bounce light can be calculated very quickly to a very high accuracy. Then, the key idea is that as light bounces in the atmosphere, the PDF over ray directions approaches the isotropic sphere. If light direction is isotropic, the N^{th} bounce light radiance at any point will be c times the $N-1^{th}$ bounce radiance for a $c < 1$ that varies by atmosphere height. Therefore, at a given point, we can approximate $\sum_{i \geq 2} L_i(x) = L_2(x)\frac{1}{1-c(x)}$ where L_2 is the radiance of 2-bounce light. We can precompute L_2 and c as a function of height and sun angle like we did for transmittance. To calculate the color of the atmosphere, we simply walk down the ray, calculating L_1 via the transmittance table and $L_{>1}$ by the L_2 table. Something I realized is that, rather than sampling light contribution at discrete points, we can solve the differential equation $x' = \sigma_e(x)x + \sigma_s(x)L(x,\omega)$ where $\sigma_e(x)$ is the extinction strength at x and $\sigma_s(x)$ is the in-scattering strength. $L(x,\omega) = L_1(x)p(x,\omega) + L_2(x)\frac{1}{1-c(x)}$ (please excuse the absolute butchering of notation that just occurred). Note that we use the phase function p only for one-bounce light, assuming isotropic behavior for the rest. Doing this created astonishingly good results - even using just 10 samples yields a result indistinguishable (to me) from the 'ground truth', at a cost of just a single extra f32 multiplication per sample. (this is surely widely used but I have never heard of it so was happy to have come up with it). As a note, this is also how I compute volumetric emission/absorption. Entering a surface will add the absorption/emission of that material and leaving will subtract it, and at each bounce we do the same calculation. Since our materials assume constant absorption/emission, this is an exact modeling, not just an approximation! Anyways, jittering the samples, exponentially growing step, lalala. The paper provided a helpful list of values and functional approximations for all the $\sigma_{a,s,e}$ components on earth and plugging them in results in something earth-like to a very very pleasant degree.

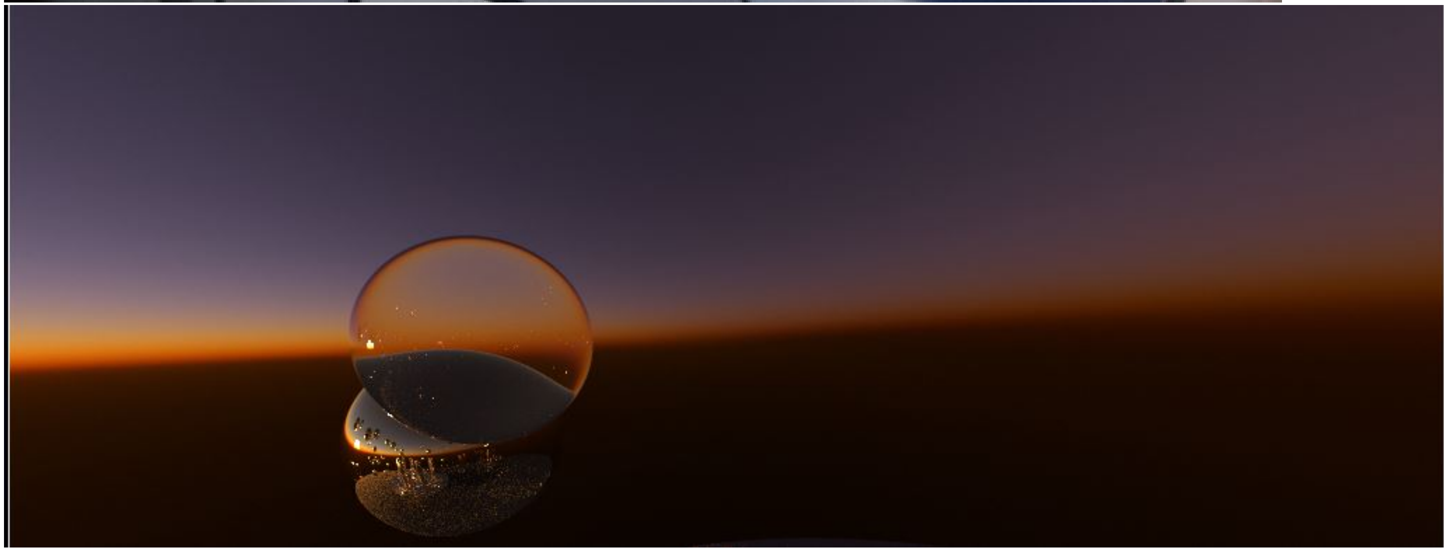
This concludes the meandering (sorry) description of implementation details. The remainder of this writeup will be a gallery.



Gallary

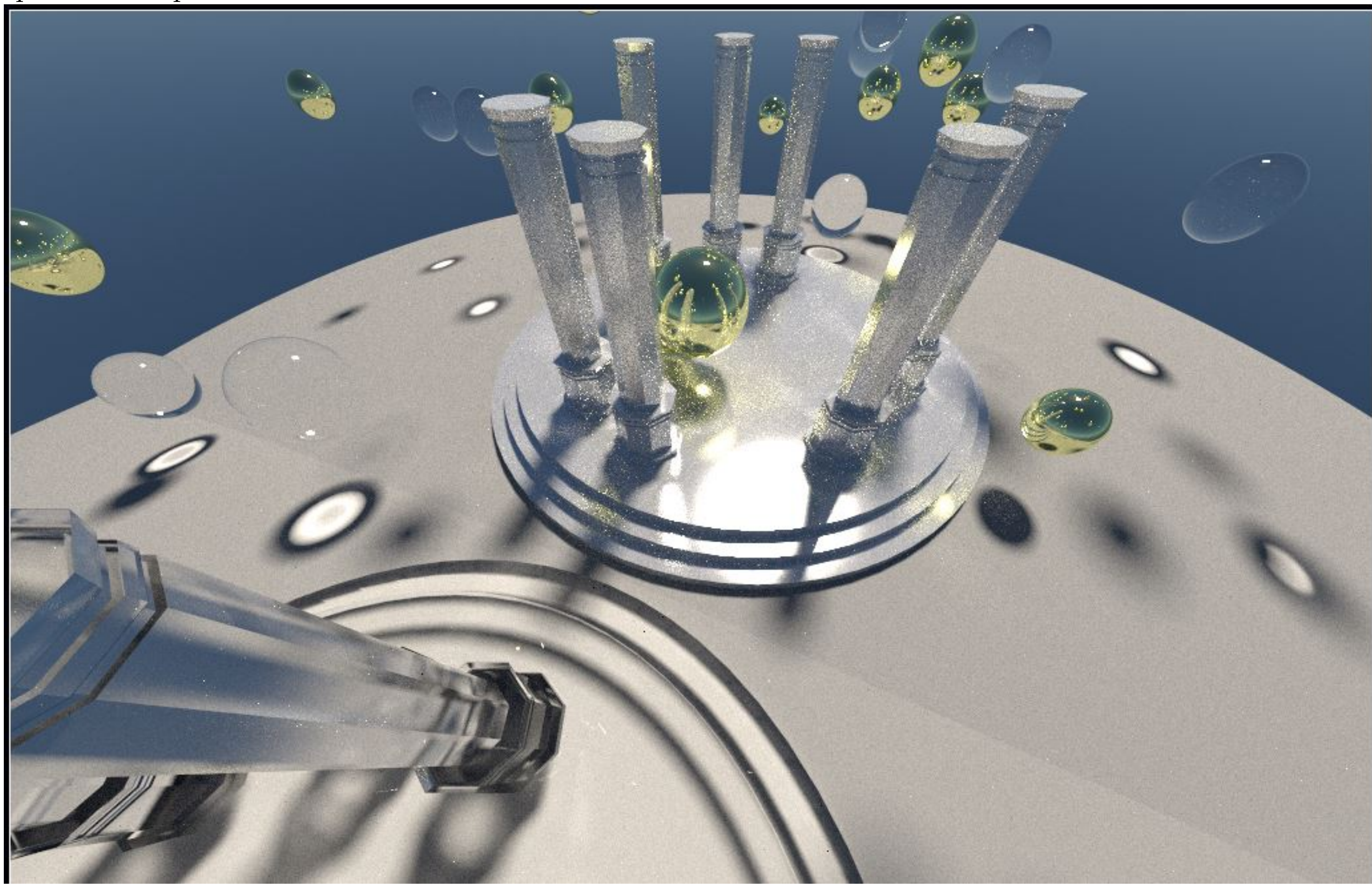
One of my goals for this project was having the atmosphere implementation be robust enough to display interesting phenomena from real-life. When I was taking a flight last year, I saw a very cool effect in which, at a high altitude, the bright part of the atmosphere 'lifts off' the surface of the earth due to its shadow. Much like how a blood moon gets its red color from light making its way through (and scattering in) earth's atmosphere, different depths of the atmosphere end up projecting a 'rainbow' into the higher parts of the atmosphere.





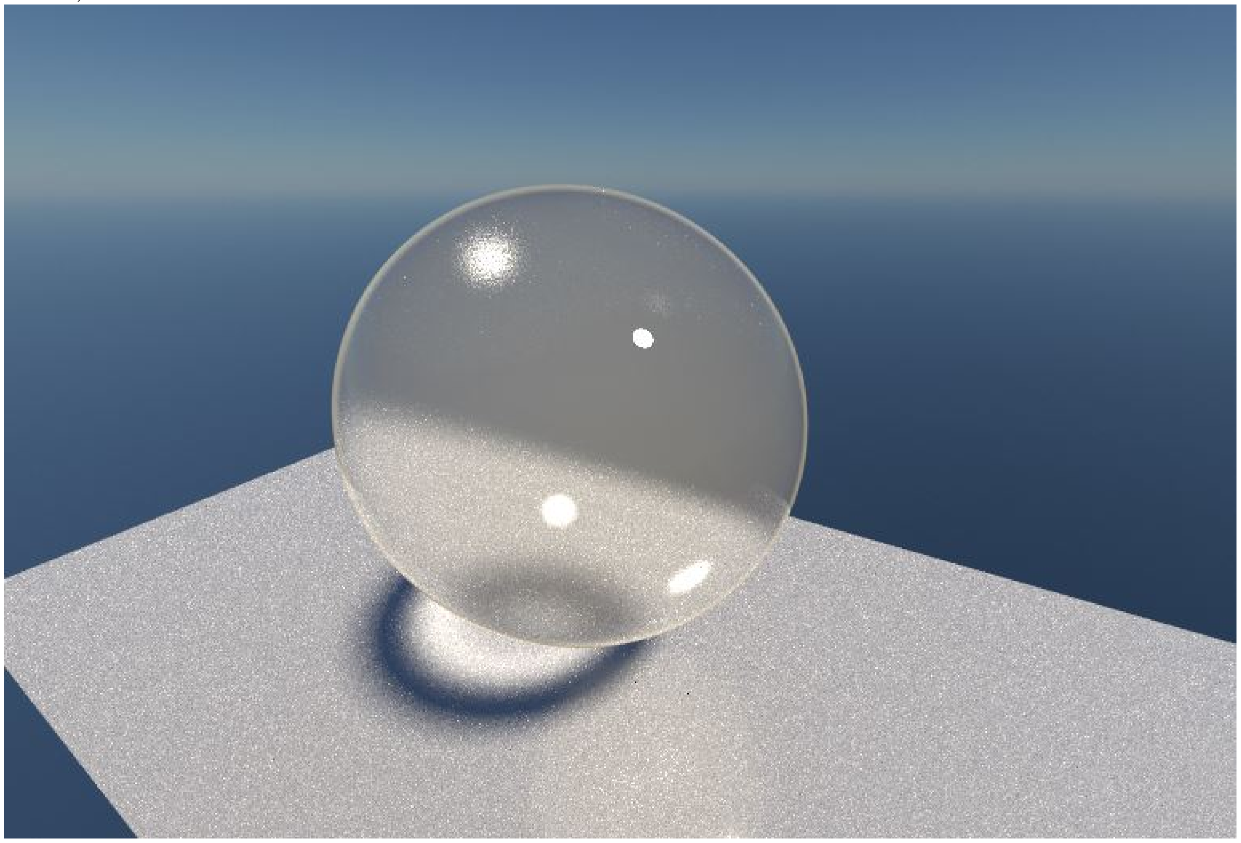
The effect, while subtle and less obvious due to the lack of anything on the ground and different color grading, is clearly visible.

Undoubtedly, the place where this render falls short is how it handles materials with a specular component in outdoor scenes.



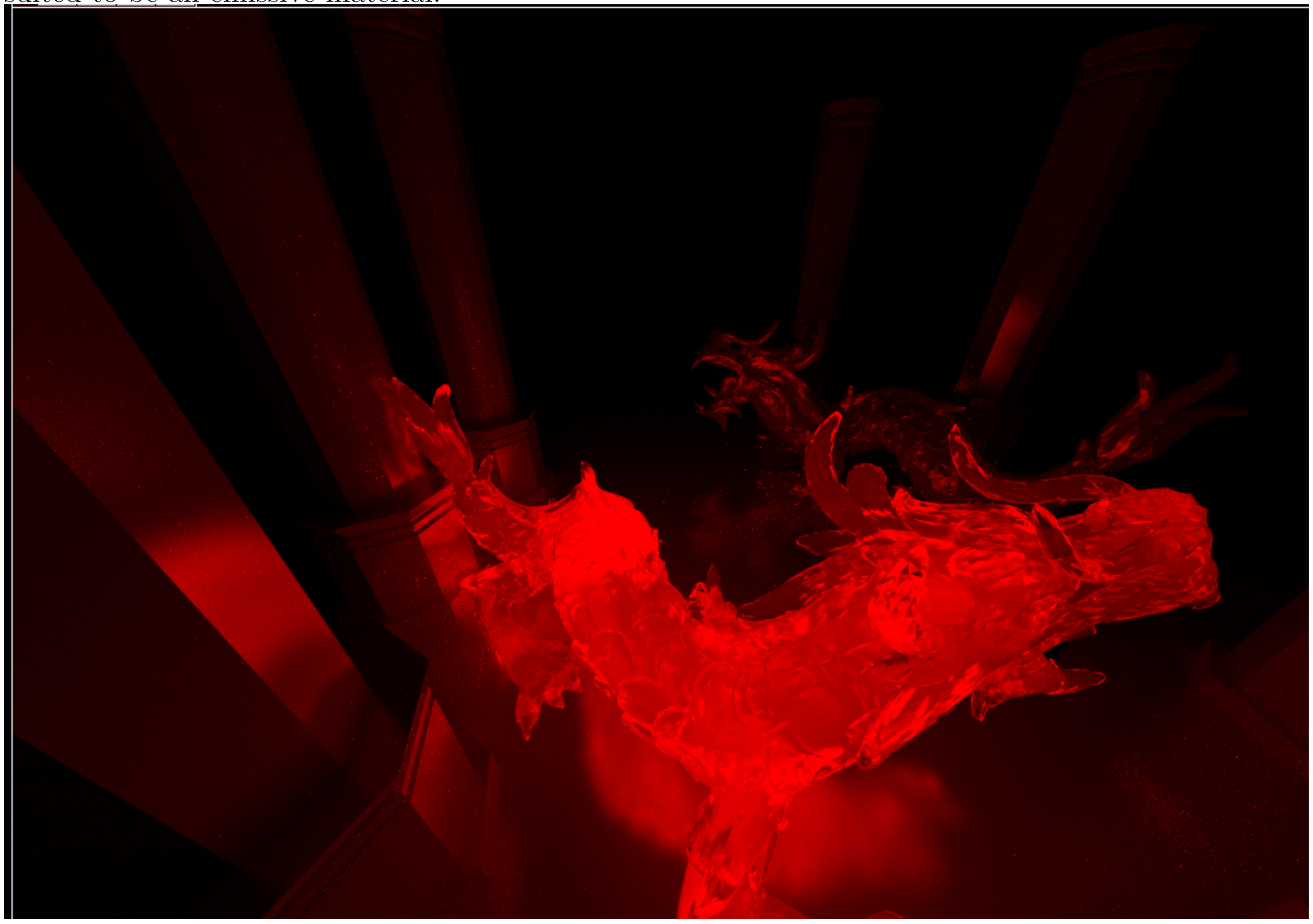
In this image, the frontmost set of pillars is made out of a material that specularly reflects half of the light hitting it with a hardness of 400. The ground is a gray, completely diffuse, material. Any light bounce going in the specular direction will get a very high throughput. Since the sun is so bright, these two very large magnitude events end up returning a sample that is exceptionally bright. Even after two minutes of converging, there are still very visible specs of light from this effect. Due to the low probability of sampling the specular region, the amplitude of these 'extreme bright events' are decreased nearly linearly by the number of resamples we take. However, they're so high in the first place that even with this measure, they are still very prominent. As a side note, since we don't special-case any rendering the glass caustics are very pretty.

I wanted to implement rough glass (to the great detriment of my video submission haha, I spent like 2 hours debugging this and only had 40 minutes to model, render and edit the video)

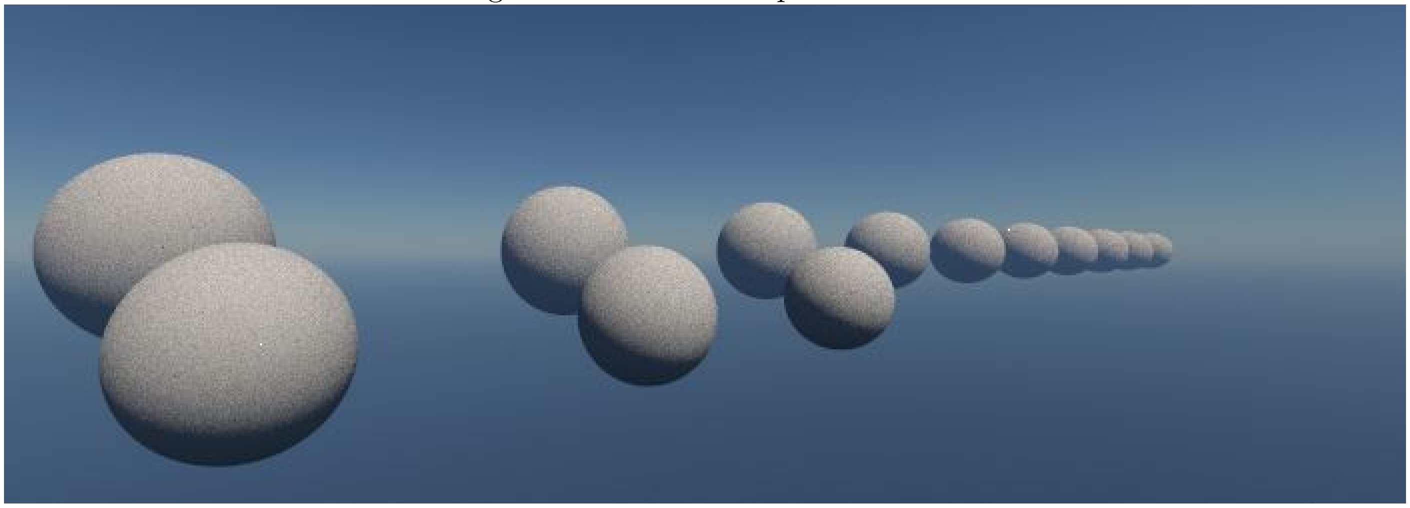


I really like the effect, but I'm not completely certain of its mathematical accuracy. We just modify the refraction direction into a direction randomly chosen in a small cone around it. This might cause bad stuff to happen at the edges, so we ensure that the random ray has the same dot product with the surface's normal as the old refraction direction (enforcing by flipping over the refraction direction if not.) This results in a little highlight around the edge of the shape, which seems like a flaw with this methodology.

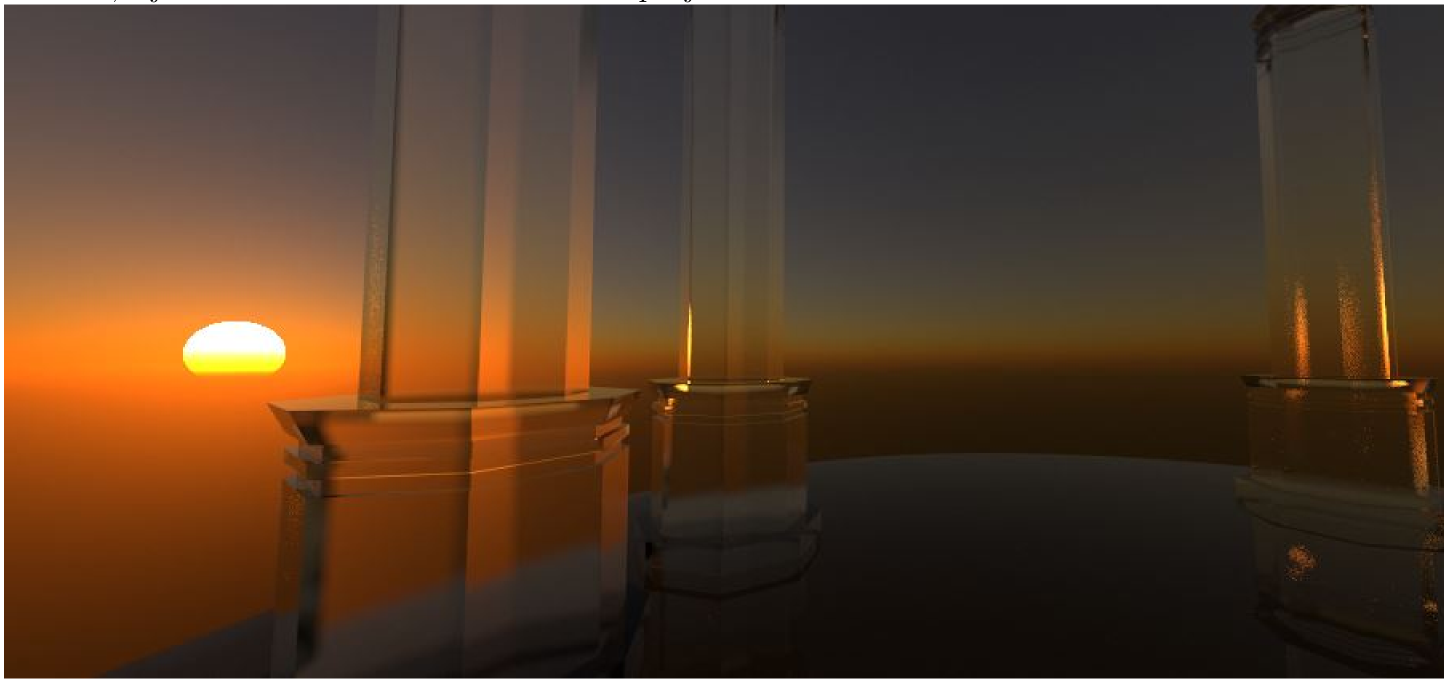
The dragon model I used is the dragon2 from cardinal3d's assets that I've decimated down to 40 thousand triangles or so. I really like this guy. In my opinion he's perfectly suited to be an emissive material.



Another nifty effect is that very far objects are obscured by the atmosphere a la real life. The back row of spheres starts 3km away from the user to the north with every next sphere another 2km to the east. The fading effect is subtle but present!



Also, I just love the look of sunsets in this project.



The unimportant stuff (like not really a focus but I spent time on it so I want to justify that by having it in the writeup)

One thing that was important to me was making the usage of the raytracer very easy. Scenes like those shown here/in the video can be made like so:

```
const registry = materials.registry
const blueGlass = registry.register(new Material({
  diffuseStr:0, transness:1, transmitCol:[1,1,1],ior:1.3,
  emissionCol:[0.01,0.03,0.05],absorbCol:[0.02,0.02,0.015],roughness:0.03
}))
if(registry.device) registry.upload();

export const Dragon = async function(bvhctx){
  const dragon = await new Mesh(assets, "pillars-dragon");
  bvhctx.addMesh(dragon, blueGlass, affineTransform(
    {xrot:-Math.PI/2, offset:[-4,0,-6]}))
  for(let i=0; i<30; i++){
    const theta = Math.random()*2*Math.PI;  const r = Math.sqrt(Math.random())*70
    const h = Math.random()*40-1;           const rad = Math.random()*2+2
    bvhctx.addCircle([Math.cos(theta)*r,h,Math.sin(theta)*r],rad,materials.mirror)
  }
}
```

I didn't want to learn file parsing, so I have a short blender script that consumes a mesh and outputs a vertex/index buffer binary, which is what the new Mesh command grabs. Short of making an actual editor, I wanted to make modifying scenes as easy as possible (to the detriment of the amount of time I could spend on the video :sad:)

I arbitrarily chose to make the canvas 1024×720. On my 4080, for simpler scenes, I could take around 1000 samples per pixel per frame, but for more complex scenes (or scenes with more bounces) the number was closer to 100.

Running the raytracer requires Node (and a good GPU). Run

```
node server.js
```

in webgpu_rt's top level directory (for handling file fetching and to allow module usage) and go to localhost:3000. I also plan to upload the site at some point but I don't know what url to which I would yet. The scene can be modified in scenes.js in src/rt