CSE 167 Final Project

Ritoban Roy-Chowdhury, Jeremy Tow

Some reflections on Tasks 1-6

Tasks 1-6 in implementing the raytracer went largely straightforwardly for us. We encountered a few significant issues

One problem we ran into was that the upper portion of our sphere was getting shaded incorrectly – a thin blue band could be scene, but it was otherwise solid black. At first we thought that we had defined the normals of the sphere incorrectly. As there is a "max(dot(normal, light_direction), 0)" term in the direct light equation, if the normals are defined incorrectly it could drag the values in the eqution down to zero. Eventually, we realized that we had forgotten to check that $t > 0$ in the sphere intersection routine. Annoyingly, we made this exact same mistake later in task 7.

One issue we had both in task 7, and in the the easier tasks were fireflights Fireflies are caused by a small amount of rays randomly hitting a very bright source. When light sources are small, the chance that a ray hits it becomes lower. The smaller the source, the greater the chance that instead of a well formed reflection, fireflies will form. When highly specular objects are brough near the light source, small light sources are created. These light sources create fireflies. There are a few ways to mitigate fireflies. One such method is intensity clamping. The intensity of fireflies are very high, so clamping the intensity values can remove fireflies. A drawback of this method is that it may reduce the overall brightness of the image. Perhaps a better method is to apply a filter after ray tracing is complete to remove fireflies. For example, if there are no intended bright single pixels, then all such pixels can be removed to remove fireflies.

Task 7: Raymarching

For our task 7, we chose to implement the raymarching algorithm. We were motivated largely by some cool videos we'd seen of Inigo Quilez on Youtube creating incredible scenes, merely writing shader code. We decided that we could probably implement the behavior in our raytracer – fortunately, Inigo Quilez has a blog where he describes how everything works in great detail.

Raymarching renders a new kind of primitive, different from the spheres or triangle meshes that we've been rendering thus far. Raymarching renders a "signed distance function", which is a function $\varphi : \mathbb{R}^3 \to \mathbb{R}$, that implicitly defines the surface by the set of points in $\mathbb{R}^3$ where $\{\varphi = 0\}$, and satisfies the additional property that $\varphi(\boldsymbol{x})$ is the minimum distance between $\boldsymbol{x}$ and the implicitly defined surface $S = \varphi^{-1}(0)$ (negative, if $\boldsymbol{x}$ is inside of the object).

Mathematically, if a signed distance function is differentiable, it satisfies the Eikonal property, that $|\nabla\varphi| = 1$.

To render such an object in a ray-tracing framework, we need a way to compute the intersection of S with a ray $t \mapsto \mathbf{p} + t\vec{\boldsymbol{d}}$. The idea is to start at $\mathbf{p}_0 = \mathbf{p}$, and repeatedly take steps $\mathbf{p}_{i+1} = \mathbf{p}_i + \varphi(\mathbf{p}_i)\vec{\boldsymbol{d}}$. These steps will get us closer to the surface S, and because φ is a signed distance function, $\varphi(\mathbf{p}_i)$ is a distance that we are guaranteed to be allowed to step along the ray without "missing" an intersection. We repeat this process until the sequence $\{\mathbf{p}_i\}$ converges (in practice, we just check that $\varphi(\mathbf{p}_i) < 1e - 3$, and that $t > 0$).

Our renderer also needs the normal vector to the surface S. Mathematically, this is $\nabla\varphi(\boldsymbol{x})$. Computationally, we estimate this numerically, either by using forward or central differences in each component. That is, we used

$$\vec{\boldsymbol{n}}(x, y, z) \approx \frac{1}{h}\begin{pmatrix}\varphi(x+h, y, z) - \varphi(x, y, z)\\ \varphi(x, y+h, z) - \varphi(x, y, z)\\ \varphi(x, y, z+h) - \varphi(x, y, z)\end{pmatrix}$$

or

$$\vec{\boldsymbol{n}}(x,y,z) \approx \frac{1}{2h}\begin{pmatrix} \varphi(x+h,y,z) - \varphi(x-h,y,z) \\ \varphi(x,y+h,z) - \varphi(x,y-h,z) \\ \varphi(x,y,z+h) - \varphi(x,y,z-h) \end{pmatrix}.$$

One does not need to explicitly divide by h or $2h$ – it is handled automatically when the normal vector is normalized.

This algorithm is quite straightforward to implement, and easily finds the location of the intersections of a ray with S given a function φ. In our code, we implement this as an additional primitive, `GeomRayMarch`, which stores φ as a `std::function(float(vec3))`. This is wrapped class implementing `ModelBase` called `Sdf` – the ray tracer can then just handle such objects. Then, in the scene definition, we simply pass a particular function φ.

Examples of Signed Distance Functions.

Fortunately for us, Inigo Quilez' blog[1] derives several handy signed distance functions for us, and we implement several of them in our `sdf_scene.inl` file. We'll describe a few of them here.

The signed distance function for a sphere, centered at the origin, of radius r is

$$\varphi(\boldsymbol{x}) = \|\boldsymbol{x}\| - r.$$

The signed distance function for a plane through the origin with normal vector $\vec{\boldsymbol{v}}$ is

$$\varphi(\boldsymbol{x}) = \boldsymbol{x} \cdot \vec{\boldsymbol{v}}.$$

We also implemented the SDFs for an axis-aligned box, but the formula for it is a little more ugly. One can bevel the edges of such a box simply by subtracting off a constant r.

A nice feature of signed distance functions is they compose nicely. For example, the signed distance function of a union is simply

$$\varphi_{S_1 \cup S_2} = \min\left(\varphi_{S_1}, \varphi_{S_2}\right).$$

It is worth noting that this will only remain an exact SDF in the exterior of $S_1 \cup S_2$ – inside, this only provides a bound on the distance, but will not be the exact distance to the nearest point. But in general, because our rays will be outside of our objects, this is fine. The situation is less ideal for intersections where the natural formula

$$\varphi_{S_1 \cap S_2} = \max\left(\varphi_{S_1}, \varphi_{S_2}\right).$$

is exact *inside* of $S_1 \cap S_2$, and will be only approximate outside of it[2]. Nonetheless, in practice this poses little problem for raymarching – and if it does, instead of taking a full step of length $\varphi(\mathbf{p}_i)$, we can just reduce it by a constant factor. We can also use intersection to implement a "difference" operation,

$$\varphi_{S_1 \setminus S_2} = \max\left(\varphi_{S_1}, -\varphi_{S_2}\right).$$

Taking the union of a box and a sphere, we get the figure

[1]https://iquilezles.org/articles/distfunctions/

[2]This is apparently poses many difficulties in some applications; I recently read a SIGGRAPH Asia paper trying to use neural networks make this more accurate

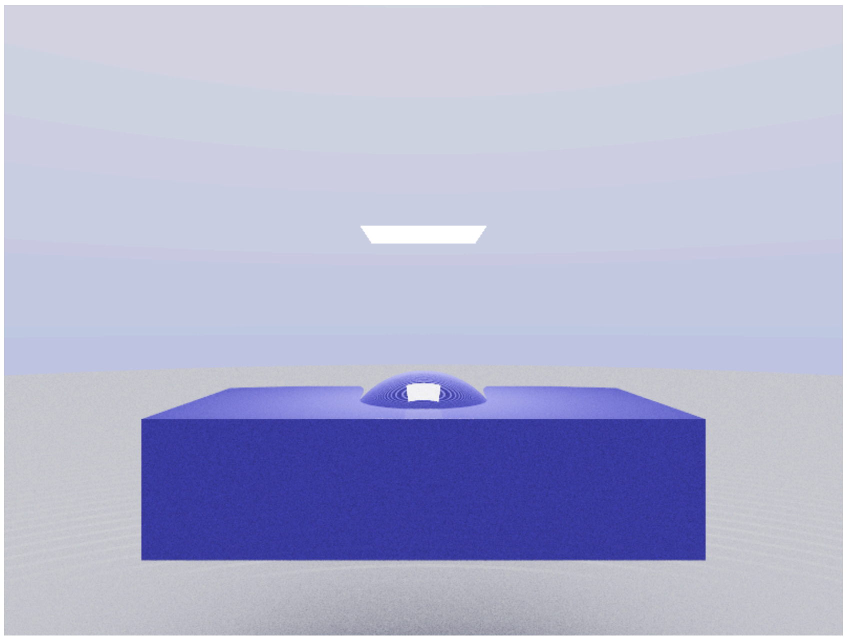

Figure 1. The union of a raymarched box and sphere

.

A cool feature of raymarching is that one can render an *infinite* amount of geometry for nearly free. As a simple example, imagine φ was the sdf of some geometry in $[0,1]^3$. Then, the SDF $(x,y,z) \mapsto \varphi(\mathrm{fract}(x), y, z)$ would be the SDF of the geometry φ, replicated infinitely many times along the x-axis. fract here returns the fractional part of an integer. In general, we defined function `repeat`, with the following behavior: Given a vector $\vec{\boldsymbol{v}}$, we split the position vector $\boldsymbol{x}$ into a component in the direction of $\vec{\boldsymbol{v}}$ and a component in the plane perpendical to $\vec{\boldsymbol{v}}$. The component in the direction of $\vec{\boldsymbol{v}}$, we reduce modulo $\|\vec{\boldsymbol{v}}\|$, and then add it back to the other component of $\mathbf{p}$ – this essentially replicates the geometry infinitely many times along the vector $\vec{\boldsymbol{v}}$. And we can nest multiple `repeat` calls to get infinite grids!

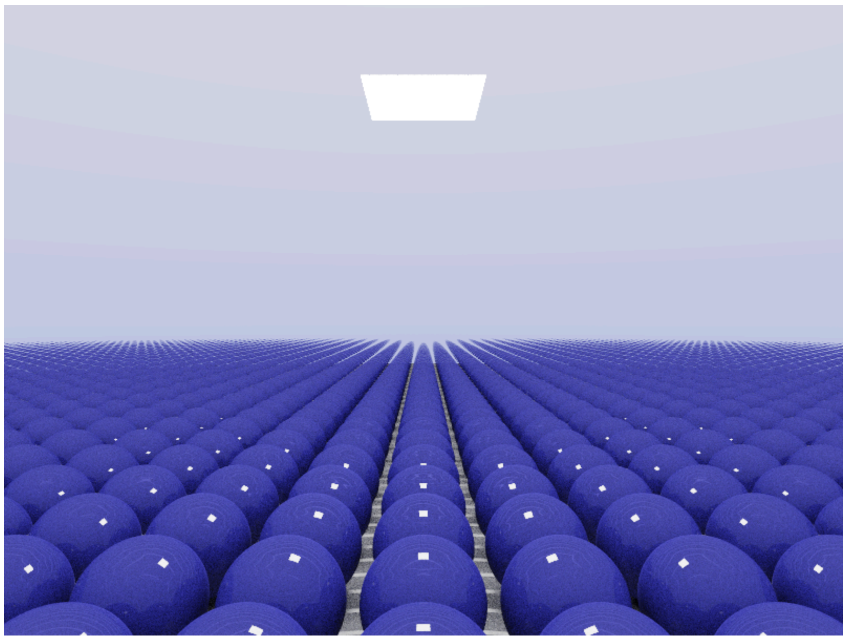

Figure 2. An infinite grid of spheres

.

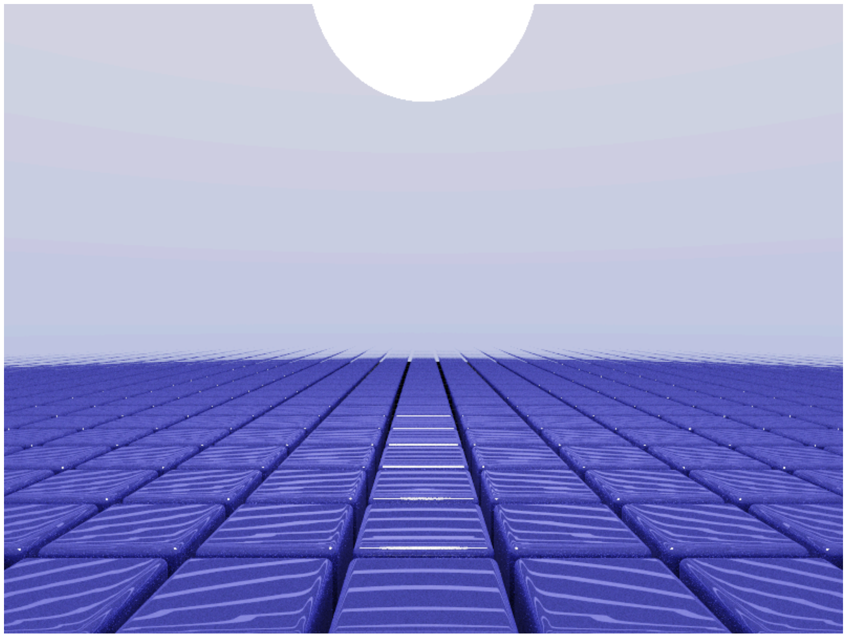

Figure 3. An infinite grid of stepping stones

.

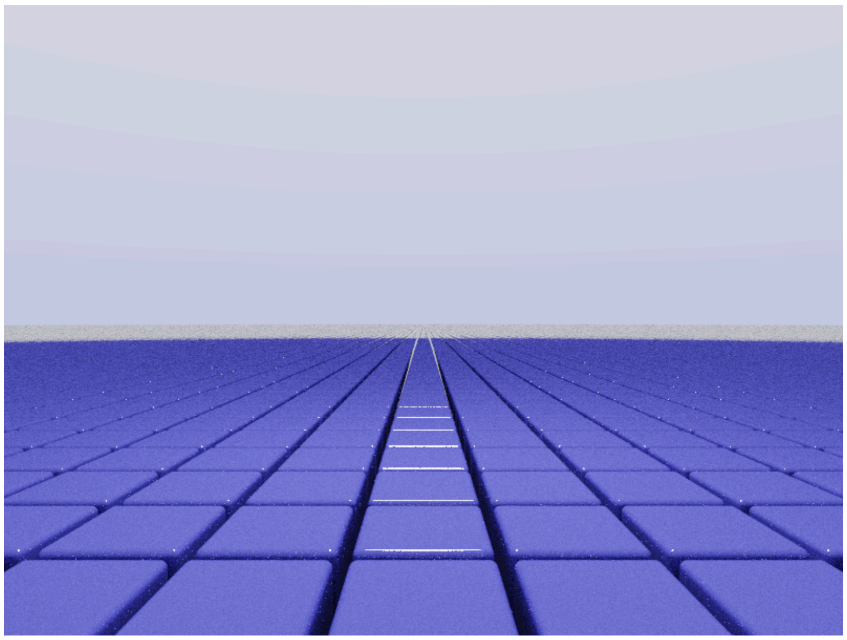

FIGURE 4. An infinite grid of stepping stones, without banding artifacts

.

One can transform an SDF just by modifying the point $\boldsymbol{x}$ passed in – $\boldsymbol{x} \mapsto \varphi(\boldsymbol{x} - \vec{\boldsymbol{a}})$ translates by the vector $\vec{\boldsymbol{a}}$, and similar for rotation. For scaling by a factor of a, one can take the SDF $\boldsymbol{x} \mapsto a\varphi\left(\frac{\boldsymbol{x}}{a}\right)$. Putting all of this together, it's not too difficult to construct a Menger sponge fractal! One starts with a box, and then creates an SDF for a 3d cross shape, as the union of 3 boxes. Then, we use the difference operator to "cut away" the cross from the outside box. Repeating this in a loop for each of the "smaller" boxes, we can easily implement an SDF for the Menger sponge fractal!

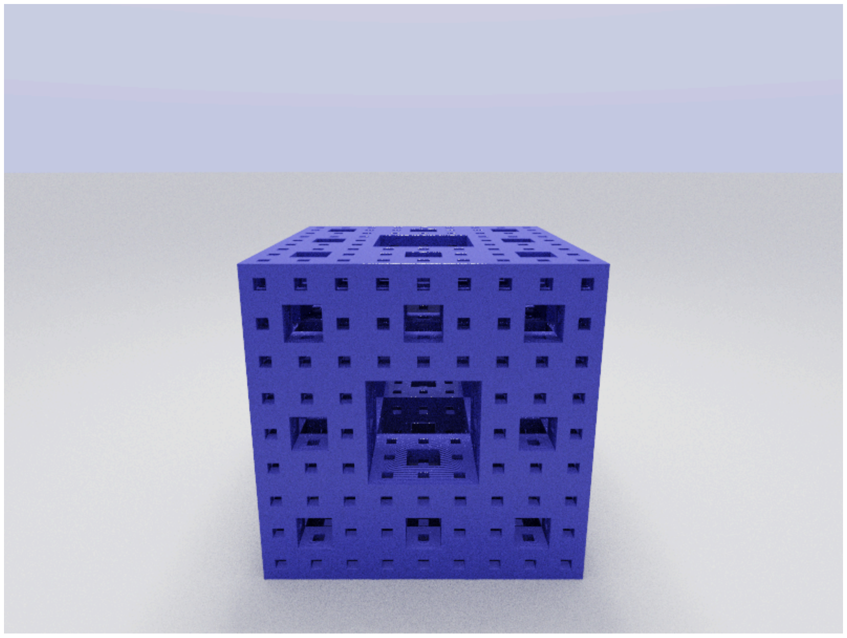

FIGURE 5. A Menger Sponge Fractal, based closely on Inigo Quilez's approach

.

We actually did experiment with some of our own own fractals – we in particular tried implementing a "pyramid" fractal, something like a 3d Koch fractal. You can still find the remnants of this in our code, in the `pyramid` function – unfortunately, our implemented proved far too slow to be practical.

One might note that some of our examples have some "banding" artifacts – we were quite confused as to the cause of these originally, but did manage to solve the bug near the end. We had simply forgotten to check that $t > 0$ in the raymarching algorithm – essentially causing "self-intersections" with the geometry we just hit after the first bounce. Why this results in banding artifacts, we don't quite understand, but we did fix it. Unfortunately, by the time we fixed it, there was not enough time to re-render all of our images, so the artifacts remain in some of them – and in some of the examples, it looks pretty cool, so lets say it wasn't a bug, it was a feature!