COS 426 Final Project Writeup
Raymarching with Signed Distance Fields

Benjamin Huang, Jake Waksbaum and Anvay Grover

May 13, 2020

Abstract

We explore an alternative technique to raytracing for rendering 3D scenes called *signed distance field raymarching* that represents the entire scene using a signe ddistance field (SDF) that maps each point in space to the distance to the nearest object. We are able to implement comparable features to the raytracer in assignment 3 at near real-time frame rates (10-30 FPS). We also render a scene that would be infeasible to produce with raytracing due to its recursive nature. We highlight the composable nature of the SDF representation by building an editor that allows users to build complex shapes by composing primitives with transformations.

1 Introduction

Raytracing is a method for rendering realistic scenes. It operates by conceptually tracing a ray backwards from the viewer's eye until it hits an object. To determine if a ray hits an object, we must be able to analytically solve the intersection equation $\vec{v}_{\text{eye}} + \vec{r}t = S$, where S is a shape in the scene. Then the solution with the smallest t represents the nearest intersection.

Raymarching is an alternative techinque to raytracing that avoids analytically solving equations to determine intersection points. Instead, it "marches" along the ray until we find an intersection with the scene by, which corresponds to checking the intersection equation for increasing values of t. Checking if we intersect with a shape at a given point still involves querying every shape in the scene, but that operation can be much faster than the computations required to solve the equations analytically. For example, we can check if $\vec{p} = \vec{v}_{\text{eye}} + \vec{r}t$ interesects a sphere with radius

r centered at $\vec{c}$ just by checking $|\vec{p} - \vec{c}| \leq r$. The analytic solution involves more computation, including square roots.

One concern with raymarching is step-size: the larger your step-size, the greater the risk of missing an intersection, but the samller the step-size, the more steps you have to take to rener the scene. One solution to this issue is to use a *signed distance field* to determine how large a step to take along the ray. The signed distance field $f : \mathbb{R}^3 \to \mathbb{R}$ is a function that assigns to each point in space the signed distance to the nearest object. This means that the distance is negative for points inside of objects. When marching, we can always take a step equal to the value of the signed distance at the current point and we can be sure we won't miss a possible intersection.

Another advantage of signed distance fields is in computing normals: to calculate lighting and shading, one needs the normal to a surface at the point. In raytracing, that involves again analytically computing the normal to the surface. With signed distance fields, we can approximate the normal by approximating the gradient of the distance field:

$$\begin{aligned}\vec{n} &\approx \left\langle \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right\rangle \\ &\approx \langle f(p + \epsilon\hat{u}_x) - f(p - \epsilon\hat{u}_x), f(p + \epsilon\hat{u}_y) - f(p - \epsilon\hat{u}_y), f(p + \epsilon\hat{u}_z) - f(p - \epsilon\hat{u}_z) \rangle\end{aligned}$$

This can also be a more efficient way of evaluating the normal.

In addition to performance benefits, the signed distance fields offers a representation of objects is more flexible than the analytic representation used in raytracing. In raytracing, the intersection and normal equations for each type of object such as sphers, boxes, cylinders, and cones, are solved by hand and those solutions are encoded directly into the raytracer. This makes it difficult to compose or modify the primitives. With signed distance fields, it is easy to compose and transform primitives. For example, given the signed distance field f of an object, we can translate it by $\vec{t}$ using $f'(\vec{p}) = f(\vec{p} - \vec{t})$, scale it by s using $f'(\vec{p}) = sf(\vec{p}/s)$ and intersect it with another object with signed distance field g using $f'(\vec{p}) = \max(f(\vec{p}), g(\vec{p}))$. We can also develop more exiotic transformations, like infinitely tiling a shape using $f'(\vec{p}) = f(\vec{p} \bmod 1)$. Some of these transformations can be used to create rich, complex scenes that use fewer primitives than one would expect, which improves rendering speed.

Our goal was to explore signed distance field raymarching, and to showcase a few of its benefits related to performance, flexibility, and compositionality. We wanted to replicate the features of the raytracer in assignment 3 at closer to real-time rendering speeds. We wanted to showcase the flexibility of signed distance fields by creating a scene that used their recursive nature to create a scene that would be infeasible

to render with a raytracer. Finally, we wanted to build an interactive editor that allows you to compose primitives and transformations into more complex shapes in an intuitive and exciting way.

In terms of previous work on signed distance field raymarching, Inigo Quilez's website [4, 5, 6] and the hg_sdf library [1] were the two biggest resources we referred to. Inigo is one of the creators of Shadertoy, an online community for sharing shaders. hg_sdf is a GLSL library for signed distance functions.

2 Methodology

2.1 Shader compilation

To build a raytracer, we need to set up a canvas, set up WebGL, and make it so our shader is called on every single pixel. Thisis all taken care of in `src/lib/shader.js`, which provides a function `initCanvas` that takes a canvas, and returns a callback function `render` and a function `newShader` that will compile and set up a shader program from a fragment shader source. The `render` function is called with the time, camera angles, camera position, and shader during each frame.

The `newShader` program is a closure around the `Shader` constructor used to remember the GL context. The `Shader` class also defined in `src/lib/shader.js`, and its main job is to lazily construct the GL shader program, the vertex array, and the uniform locations that are used in `render`. It does this by creating private properties (marked by a hash) `#program`, `#vao`, and `#uniforms`. Then, it defines getter methods for the public properties of the same name that check if the private property has been initialized, and if not runs the correct computation and caches the result in the private property. This is useful, because the steps of preparing the GL program can take some time, and this ensures that we wait until each part is needed before we compute it. We don't need to compile all of the shaders available on the main page of our application; we can wait until they are selected.

2.2 Raymarching

We wanted to share raymarching engine between all of our raymarching scenes. To that end, we define functions that can be used to define a signed distance field in `shaders/lib.frag`, and we define the rest of the raymarching engine in `shaders/engine.frag`. To build the full fragment shader, we splice the scene-specific part in between the library and the engine. That fragment has to define a

few variables and function:

- The signed distance field itself `float sdf(vec3 p, out Material mat)`. It returns the distance to the nearest object at the point `p`, and puts the material of that object in `mat`.
- `const int MAX_STEPS` determines the maximum number of steps the raymarcher will make.
- `const int MAX_DISTANCE` determines the maximum distance a point can be from an object and still be considered to have hit it.
- `const vec4 BACKGROUND_COLOR` is the color of pixels that don't hit an object.
- `const int N_LIGHTS` is the number of lights in the scene.
- `Light lights[N_LIGHTS]` is an array of light objects.
- `void init()` is a function that is called once at the beginning of the fragment shader. This allows the scene to initialize scene-specific global variables.

The core of our raymarching engine is the `march` function in `shaders/engine.frag`. For a given ray `r` defined by an origin and direction, the march function will calculate the first intersection point along `r`. It does this by iteratively stepping along the ray, starting from the origin. On each iteration, it calculates the signed distance `d` from the point `p` to the nearest object by calling `sdf`. If this distance is very near or less than 0, `march` returns a Hit object, containing the distance to the object, the point in the scene where the intersection occurred, and the material of the nearest object. This behavior is modified wit the introduction of reflection and refraction, described below. If the distance is not near 0, then `p` is moved `d` along the ray. This should guarantee that the sampled points along the ray never skip over an intersection point. This loop continues to iterate for `MAX_STEPS`.

If no intersection is found in `MAX_STEPS`, then a Hit object is constructed for the final sample point along the ray. In certain situations, this data can be used to improve performance by assuming such Hit objects represent valid scene intersections unless the distance is *very large*. For example, the number of steps required to render an infinite plane effectively can be vastly cut down by assuming even distant Hit objects would eventually hit the plane.

2.3 Hard and soft shadows

To calculate hard shadows at a particular point `p` in the scene, we follow an analogous process to what we implemented in assignment 3. For each light in the scene, the

`shadow` function in `shaders/engine.frag` is called. A marching loop very similar to the one described above is run to march from the point in the scene to the light. If an object is intersected during this march, then `shadow` returns 0. Otherwise, it returns 1.

An enormous advantage of raymarching over raytracing is the ability to calculate soft shadows using the same number of rays as hard shadows. In fact, there are only a few additional lines of code to calculate soft shadows. To approximate light sources as spheres of light instead of point light sources, we want to assume only some fraction of the light will make it to a point `p` if a ray comes too close to an object. Because we are using signed distance fields, we already have this information! So, at every time step, we calculate a ratio of the light that the point receives, whose value corresponds to the distance to the nearest intersection. Furthermore, we want this ratio to inversely correspond to the distance from the origin point `p` that the intersection occurred. We take the `min` across these ratios, initializing to the max value. Our original equation was

$$r = \texttt{min}(r, \frac{\texttt{dist\_to\_obj}}{\texttt{dist\_from\_p} \cdot w}),$$

where w is a parameter that controls the sharpness (closer to 0 is sharper). It became clear that the inverse relation to the distance from the origin should not be linear. With a linear drop-off, immense shadow artifacts would crop up when light sources were placed near objects. As such, we added a log to the denominator:

$$r = \texttt{min}(r, \frac{\texttt{dist\_to\_obj}}{\log(\texttt{dist\_from\_p} \cdot w + 1.0)}),$$

which effectively eliminated the shadow artifacts. Note that this ratio is actually in the range $[-1, 1]$ to optimize a multiplication and divide out of the marching loop. As such, before returning, we use the `smoothstep` function to change the range to $[0, 1]$.

2.4 Phong shading

Phong shading is very similar to what we implemented in assignment 3. In `calcColor` in `shaders/engine.frag`, we loop over all of the lights in the scene-defined `lights` array and accumulate the contributions of each. Those contributions are calculated in `phongContrib`. The Phong reflection model [3] says that a light with RGB intensity $\vec{i}$ shining on a material with specular reflection $\vec{k}_s$, diffuse reflection $\vec{k}_d$, and shininess α will contribute a term

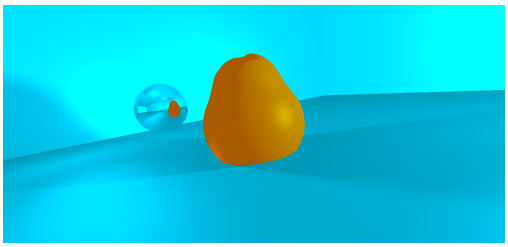
$$(\hat{L} \cdot \hat{N})(\vec{k}_s \odot \vec{i}) + (\hat{R} \cdot \hat{V})^{\alpha}(\vec{k}_d \odot \vec{i})$$

where $\hat{L}$ is the vector from the point to the light-source, $\hat{N}$ is the normal to the surface, $\hat{R}$ is the direction a ray of light would go if reflected perfectly off the surface, and $\hat{V}$ is the vector from the surface to the viewer. $\odot$ denotes element-wise multiplication. It's also important that if either of the dot products is negative, that term be ignored (there is no negative light here) so we clip them below to zero.

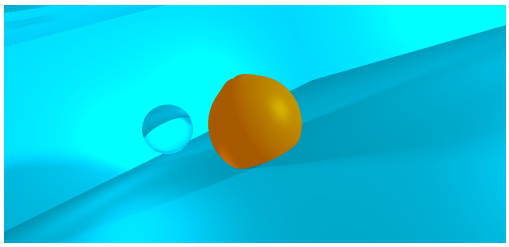
2.5 Reflection and refraction

We have 3 kinds of reflection types for the materials of our objects. NONREFLECT models a simple opaque surface that does not reflect light. This is our default reflection type. MIRRORREFLECT models a simple mirror-like surface that reflects all the light incident on it. When we have a ray intersection with such a surface, we use the GLSL *reflect* function to compute the new direction of the ray. We then continue marching in this new direction. As a result, the color that we compute for this ray corresponds to the intersection point of this reflected ray.

Lastly, we have the GLASSREFLECT reflection type, which models a glass-like surface and is the most tricky to get right. We go for a relatively simple glass-like effect where we refract the direction of the ray when the ray first enters the glass, continue marching along the ray, and then refract again when we exit the glass. A more sophisticated approach, that we attempted to take but were unsuccessful in, would involve modeling glass as a material that both reflects and refracts light. We use the GLSL function *refract* : $(\vec{i},\ \vec{n},\ \eta) \rightarrow \vec{r}$ to compute the directions of our refracted ray. Here, $\vec{i}$ is in the incident ray direction, $\vec{n}$ is the normal vector at the position, η is the ratio of refractive indices of the first material to the second, and $\vec{r}$ is our new direction. Once we enter the glass, we subtract distances to march along the refracted ray since our distances are now negative. We continue marching inside the glass object using a nested for-loop within our outer raymarching for-loop. When our signed distance becomes positive again, i.e., when we have exited the object, we refract our ray again. However, this time we use the negated normal $-\vec{n}$, since the normal at our current position points away from the surface of the object. We also use the reciprocal of η here, since refractive indices are now reversed. Despite only using refraction to model glass-like surfaces, we are still able to achieve good results. The images below contrast the mirror and glass materials. In the mirror object, we see the reflection of our surrounding scene, whereas this is not the case for the glass object.



(a) Mirror spherical object



(b) Glass spherical object

2.6 Recursion

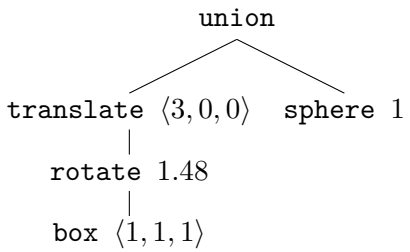
A fantastically convenient side-effect of using signed distance fields is the ease with which recursive and fractal scenes can be composed. By adding a loop to our signed distance function and applying something similar to a tiling operator on each iteration, we can easily form recursive structures.

However, recursive signed distance fields, even with low levels of recursion, are costly operations that significantly impact performance. As such, it is often important to wrap recursive construction in bounding boxes. This is often easy to accomplish as the base of the recursive object is a primitive distance field. The bounding box can then simply be formed by adding a slightly larger version of the primitive.

2.7 Editor

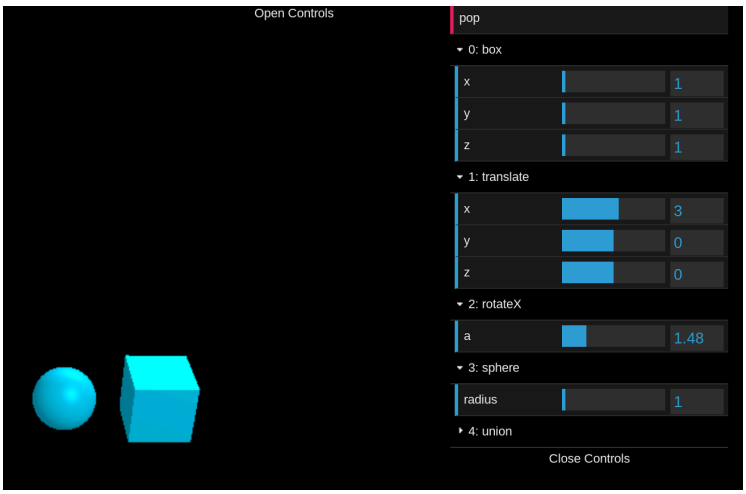
To show how you can build up an SDF from primitives and operations, we built an interactive editor. Because we didn't have so much time, we developed a stack-based interface where you push operations onto a stack of operations. This is similar to RPN notation for math, in that we are implicitly representing a tree with a post-order traversal.

For example, to draw a box with its upper right-hand corner at $\langle 1, 1, 1 \rangle$ translated 3 units to the right, and also a sphere with radius 1 centered at the origin, we conceptually want the following tree of operations.



To input that into the editor, we just follow a post-traversal:

- `box` $\langle 1, 1, 1 \rangle$
- `translate` $\langle 3, 0, 0 \rangle$
- `rotate` 1.48
- `sphere` 1
- `union`



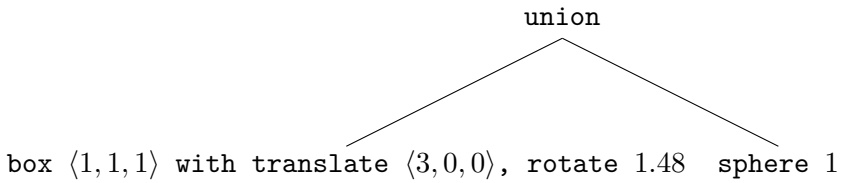
It's important to note that until you push the `union` operation onto the stack, you will not be able to see the sphere. That's because by default the editor only considers the first "item" on the stack. It won't notice the second item until you join it into one item with `union` or some other operation.

The editor operations are split into three types: shapes, operations, and transformations. Shapes push a primitive shape onto the stack. All shapes are centered at the origin initially and can be translated using `translate`. Operations combine two shapes, for example by taking their union or intersection. Transformations modify a single shape, for example by translating or rotating it.

The editor is mostly implemented in `src/pages/editor/gui.js` and `src/lib/generator.js`. Most operations are implemented using buttons on the main `dat.gui` instance. These buttons correspond to instance methods on the `Config` class. In each of these methods, we push an operation onto the stack of operations, add a folder to the `dat.gui` which represents the stack of operations, and add an input for each paramter of the operation (`radius`, `offset`, etc.).

In the `Config` class, again, we make use of lazy evaluation and caching to avoid compiling the shader program were unnecessary. For this reason, we hide all public fields that we expose to the main `dat.gui` behind getter and setter methods that internall call the `#invalidate` method. Similarly, we add a hook to each input of the stack `dat.gui` to call `#invalidate`. `#invalidate` clears the cached shader program, so that the next attempt to access the shader will cause it to be assembled as a string and compiled. Without this, we would be reassembling the shader and compiling on every frame, which would be very inefficient.

To translate from a stack of operations to a GLSL string, we first recover the tree structure. In `buildSDF` we iterate over the stack, popping off previous operations according to the arity of the current operation, and build a tree. Notably, this tree is not the same as the conceptual tree of operations because some operations apply to the domain of a SDF, and some to the range. For example, to translate an object we offset the input to the SDF, whereas to union two SDFs we take the min of their outputs. So, in terms of the GLSL AST, we want a tree of function calls acting on the range where the leaves are primitive shapes, with each leaf is annotated with a list of transformations to apply to the input to that function. So the intermediate tree for the above example would be



Once we have this tree, we can recursively traverse the tree, transforming nodes to the appropriate calls. When we reach a leaf, we build up the input to the primitive shape function by composing the functions for each domain-modifying operation. Notably, because domain-modifying operations are in a sense the inverses, we compose them in the opposite order in which they were applied. This tree would become the following GLSL fragment, spliced into a template that is pretty straight-forward.

```
min(sdSphere(p, 1.00), sdBox(opTx(opRotX(p, 1.48), vec3(3, 0, 0)), vec3(1,
1, 1)))
```

Once you get the hang of the input model, the interface is very intuitive! Fig. 2 shows a few shapes and scenes we composed using the editor.

As we mentioned, the editor operations can be divided into 3 main types: shapes, operations and transformations. We describe each of these in greater detail here.

Shapes refer to the geometric primitives we can render. These are written as functions that take certain parameters, always including a position vector representing

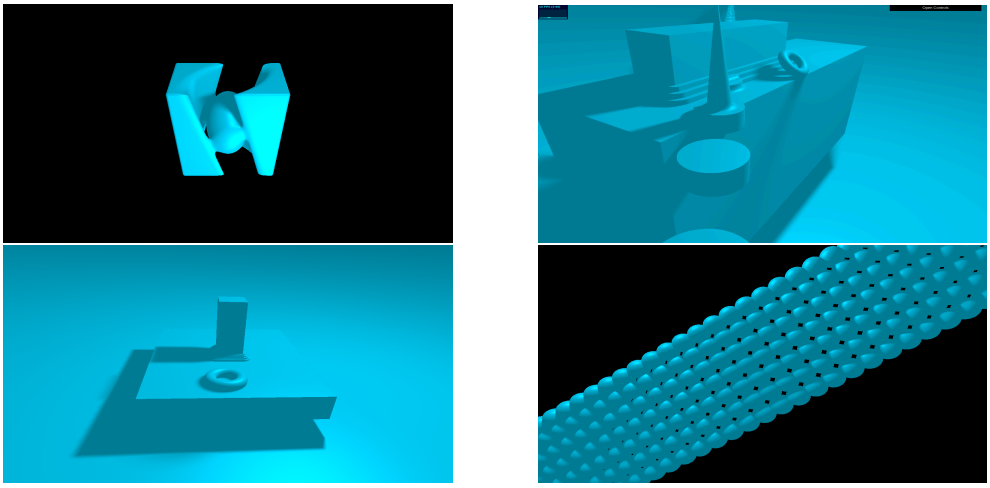


Figure 2: Assorted products of the editor

our current point, and return a float that represents the signed distances to the shape from our current point [4]. We can construct the following shapes in our editor:

- **Sphere**: Parametrized by the position $\vec{p}$ and the radius of the sphere r.
- **Box**: Parametrized by the position $\vec{p}$ and the the coordinate of the corner of the box $\vec{b}$ in the positive quadrant.
- **Round Box**: Similar to Box, but has an additional parameter r which is a float that is subtracted off the edges of the box. The value of r determines the roundedness of the box.
- **Cylinder**: Parametrized by the position $\vec{p}$ and the height h and radius r of the cylinder.
- **Round Cylinder**: Similar to how the Round Box works. We take a float as an additional parameter which is 'shaved' off the edges of the cylinder.
- **Plane**: Parametrized by the position $\vec{p}$ and the normal vector $\vec{n}$ to the plane.
- **Torus**: Parametrized by the position $\vec{p}$ and the major and minor radii of the torus.
- **Elongated Torus**: A variant of Torus that can be extended in the x, y and z directions. We have parameters for elongating the torus in each of these directions.

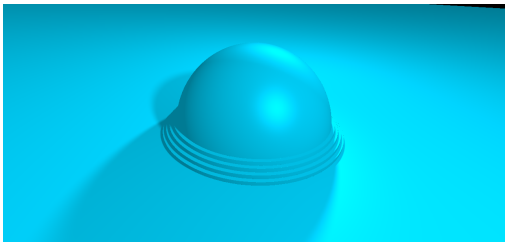
- **Link**: Produces a shape which produces one half of a chain-link geometric structure. It is parametrized by the position $\vec{p}$, the major and minor radii of the link and the length of the link.
- **Ellipsoid**: Parametrized by the position $\vec{p}$ and one of the radii r of the ellipsoid.
- **Cone**: Parametrized by the position $\vec{p}$, the vector $\vec{c}$ which is normal to the surface of the cone and the height of the cone h.
- **Pyramid**: Parametrized by the position $\vec{p}$ and the height of the pyramid h.

Operations act on the ranges of our primitives, i.e., they take as input the signed distance values given by our primitives and compute some new signed distance value. The stairs operations are drawn from [1]. We provide support for the following operations in our editor:

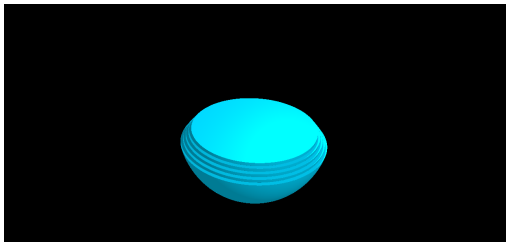
- **Union**: A binary operation that combines two shapes into a larger composite shape. We take as input 2 floats f_1 and f_2 for the signed distances to our two shapes respectively, and return the minimum of f_1 and f_2.
- **Smooth Union**: An alternative way of performing Union where we have a smooth transition between the edges of our two shapes where they are connected. We take an additional float k as an input that determines the degree of the smoothness of the union between the given shapes. We restrict k to be between 0 and 2.
- **Subtract**: A binary operation that combines two shapes by 'subtracting' the second from the first. Intuitively, Subtract removes the common part of the two shapes from the first shape. We take as input 2 floats f_1 and f_2 for the signed distances to our two shapes respectively, and return the maximum of $-f_1$ and f_2.
- **Smooth Subtract**: Similar to how Smooth Union operates.
- **Intersection**: A binary operation that combines two shapes by taking the 'intersection' of the shapes. Intuitively, Intersection only retains the common part of our two given shapes. We take as input 2 floats f_1 and f_2 for the signed distances to our two shapes respectively, and return the maximum of f_1 and f_2.
- **Smooth Intersection**: Again similar to how Smooth Union operates.
- **Union Stairs**: A binary operation that takes the union of two shapes and creates a set of steps between the edges of our two shapes at the places where

they connect. We take two additional floats r and n as parameters, where r is the width of each step and n is the number of steps.

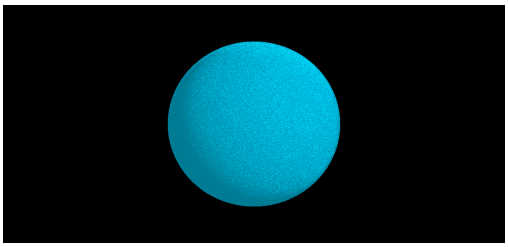
- **Difference Stairs**: Behaves the same way as Union Stairs, but subtracts the second shape from the first instead of taking the union.
- **Intersection Stairs**: Behaves the same way as Union Stairs, but takes the intersection of the two shapes instead. Fig. 3 shows the Union and Intersection Stairs.
- **Noise**: A unary operation that adds Perlin noise to the signed distances of a shape. We adapt a Perlin noise implementation [2], which is parametrized by a position vector. We use this noise function in our operation. We then use the GLSL *mix* function to 'add' some of the noise to the signed distance value of our shape. Our function also takes in a float 'noiseProp' as input, which determines how much noise is added to the signed distance. $(1 - \text{noiseProp})$ is the third argument we pass to *mix*. We restrict noiseProp to between 0 and 0.5.
- **Onion**: A unary operations that creates a hole inside of a shape, thus creating a hollow object. We take a float 'thickness' as input, which determines the size of the hole; a smaller values for thickness corresponds to a larger hole.



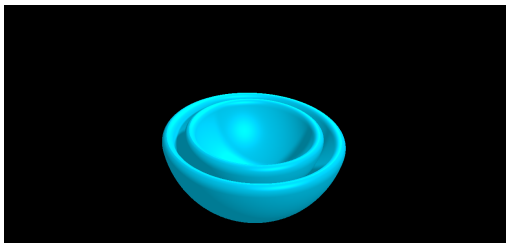
(a) Union Stairs



(b) Intersection Stairs



(c) Noise



(d) Double Onion

Figure 3: Different Operations on Spheres

Transformations act on the domain of our primitives, i.e., they take as input some position vector $\vec{p}$ and compute a new position. Since transformations act on the

domain, we need to perform the 'opposite' transformation on our given inputs to achieve the correct effect on the range. We provide support for the following transformations in our editor:

- **Translate**: Shifts the given shape in a desired direction. We take a vector $\vec{d}$ as input and subtract it from the position $\vec{p}$.
- **Scale**: Acts on the size of a given shape. We take a float s and divide the position $\vec{p}$ by s.
- **Infinite Tile**: Reproduces a shape an infinite number of times. We take a float c that is the repetition period. It is important to note here that we require that our position be in the positive x-quadrant for Infinite Tile to work. Otherwise our shape will disappear from the scene.
- **Finite Tile**: A variant of Infinite Tile which produces a finite number of repetitions of a shape. Here we take an additional vector $\vec{l}$ as input where $\vec{l}$ is the corner of the box which we use to limit the number of repetitions of our shape.
- **RotateX**: Rotates a shape around the x-axis, i.e., in a plane perpendicular to the x-axis. We take an additional float as input, which is the angle of rotation.
- **RotateY**: Similar to RotateX, but rotates around the y-axis instead.
- **RotateZ**: Similar to RotateX, but rotates around the z-axis instead.
- **Bend**: Bends the given shape about the x-axis. We take a float as input which determines the degree of the bend.
- **Twist**: Creates a twisting-like effect about the y-axis. We take a float as input which determines the degree of the twisting effect.

3 Results

We have three main deliverables to show as part of our project. First, we have the *seasquid.frag* shader program, which renders an animated seasquid-like object with soft shadows inside of a blue box. The scene is shown in Fig. 4. While the scene geometry is not too complicated, the scene illustrates many notable results. The seasquid object is the composition of primitives and operations on those primitives. In particular, we use the Sphere and Box primitives, Smooth Union, Smooth Subtraction and Bend operations, and the Rotation and Translation transformations. In addition, we have a mirror/glass sphere in the scene. We are able to render Soft

Shadows with two lights at up to 30 FPS. This is very hard to achieve with the raytracing technique we used in Assignment 3. Our interface also provides parameters for changing the x, y and z coordinates of the camera position and the camera angle. We also provide support for changing the x, y and z speeds of the camera. These speeds start at 0 by default, but can be increased to obtain a rotating camera.

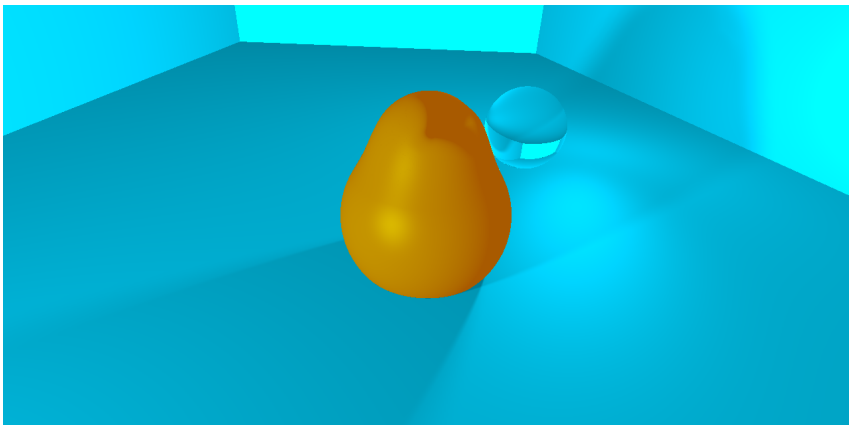


Figure 4: Scene rendered by the *seasquid.frag* program

Second, we have the *recursive1.frag* shader program. Fig. 5 shows the scene rendered by this shader at two different stages of its animation. The scene renders at 15 frames per second, which would not naively be possible in a raytracing engine.

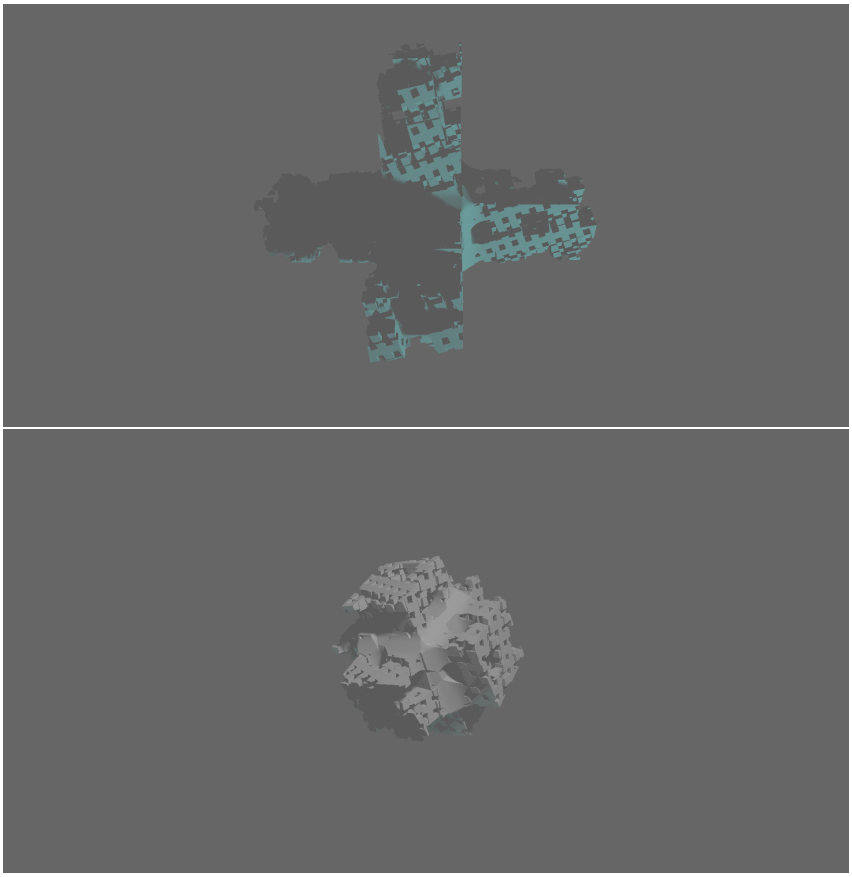


Figure 5: Scene rendered by the *recursive1.frag* shader at different stages

Our third deliverable is the editor which we described in a previous section. The editor automates the assembling of shader programs, using which we can quickly build our own customizable scenes. We show some such scenes in Fig. 2, as well as in Fig. 3. The editor again illustrates the benefits that the composition of primitives and operations in raymarching can provide. Our editor is also built in such a way that it easy to add new features to it.

4 Future Work

There are a number of features that we can possibly improve on in our editor. As we described in the Editor section, we compile our stack of operations to a GLSL string, which is then compiled. So, when we change a certain parameter in our GUI, we reassemble a shader program and compile it. Our display then shows the result of

compiling this new shader program. This means that there is a 'jump' in our display whenever a shader program is compiled. It might make for a better display if, for instance, we could animate the transition of the radius of a sphere being increased. A tradeoff here, however, would be that it would quite expensive to do this kind of a computation repeatedly.

We also think that while the way our GUI works right now, using a stack of operations, is quite intuitive once the user understands its functioning, it might be more accessible and interactive to have a UI where the user can click and drag shapes onto the screen. This way, the user can see the operations being carried out on the screen itself. Again, this might require some support for animation. It would also be great if we could have a menu from which to select the features we want in our editor at any given time. At the moment, our screen can get a bit cluttered because of the number of features we have.

References

[1] *hg_sdf library*. URL: http://mercury.sexy/hg_sdf.

[2] *Perlin morph*. URL: https://www.shadertoy.com/view/Ms2yWz.

[3] *Phong reflection model*. URL: https://en.wikipedia.org/wiki/Phong_reflection_model.

[4] Inigo Quilez. *Distance functions*. URL: https://iquilezles.org/www/articles/distfunctions/distfunctions.htm.

[5] Inigo Quilez. *Normals for an SDF*. URL: https://iquilezles.org/www/articles/normalsSDF/normalsSDF.htm.

[6] Inigo Quilez. *Penumbra shadows in raymarched SDFS*. URL: https://iquilezles.org/www/articles/rmshadows/rmshadows.htm.

[7] *WebGL Tutorial*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial.

[8] *WebGL2 Shadertoy*. URL: https://webgl2fundamentals.org/webgl/lessons/webgl-shadertoy.html.