Signed Distance Field in Real-time Rendering

Final Project in CS171 course

Huang Chengyu
Chen Yizhang
Cheng Ang
huangchy@shanghaitech.edu.cn
chenyzh@shanghaitech.edu.cn
chengang@shanghaitech.edu.cn

ABSTRACT

A implementation of signed distance field with ray marching rendering. The implementation can be running at real-time. The render result have soft shadow and ambient occlusion.

KEYWORDS

signed distance field, ray marching, real-time rendering

1 INTRODUCTION

The full name of SDF is Signed Distance Field. This is a way to represent the surface of objects different from the common method, which is using meshes. *Distance Field* means that we can get the distance from any sampled point in our scene to the nearest surface; *Signed* means that the distance we get is signed. Normally, if the sampled point is outside any of the objects in the scene, the distance we get is positive; if inside, then negative; if the point is exactly on the surface of one of the objects, then the distance we get would be zero.

The essence of rendering an SDF is to render the isosurfaces. We created a function $f(p)$ to evaluate the distance from point p to its nearest surface, whose domain is $\mathbb{R}^3$ and range is $\mathbb{R}$. What we want to do is to render the surface where $f(p) = 0$.

We used ray marching to render SDF [4]. Similar to ray casting, ray marching shoot rays from the camera to every pixel; the difference is that ray marching steps gradually from its origin. With every step, a point p is sampled; call $f(p)$ on p, if it returns 0, then we get the intersection point of the ray and the object.

With SDF and ray marching, it is easy for us to implement many visual effects which are hard to implement with rasterization, such as soft shadow, ambient occlusion and meatball.

We did our project based on OpenFrameworks. For the functions that are no so essential to our project such as UI and camera control, we just used the API of OpenFrameworks.[1]

2 RAY MARCHING

Since OpenGL does not support ray marching by itself, we render the screen by using a shader on a rectangle. At first, we compute the vector of the four corners of the view frustum in the view space, then assign it to the four corners of the rectangle. After that, we compute the matrix which transforms from view space to world space, so that we can compute the vector of the four corners of the view frustum in the world space. Finally, our fragment shader will interpolate the ray direction for every pixel. We also can compute the ray direction for every pixel in fragment shader directly based on the camera parameter, but since the interpolation from vertex shader to fragment shader is done by hardware, it is more efficient to do it this way.

We used sphere tracing when doing ray marching. With the function $f(p)$, we can get the distance d from any point p to the nearest surface; In other words, there is no object in the sphere centered at p with the radius d. With this guaranteed, we can let the ray directly step length d from p instead of a fixed length. [2]

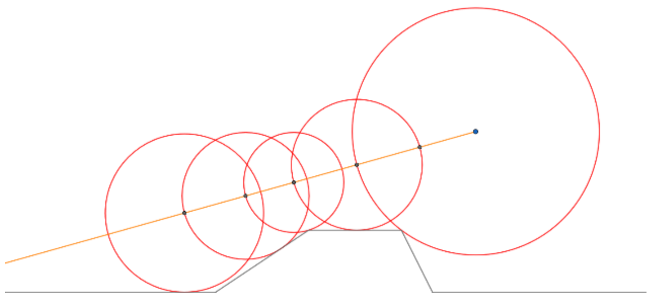


Figure 1: Sphere tracing

3 SHADING

3.1 Normal Calculation

After we get the intersection point of the ray and the surface, we have to calculate the normal of the surface before we move on. Since the surface is an isosurface, we can take the gradient of the point as its normal. We used Central Difference to calculate the gradient. One of the method to do this is to get the distance in positive and negative directions of the three dimensions respectively. The formula is shown as below.

$$\nabla f(p) = \begin{bmatrix} \frac{df(p)}{dx} \\ \frac{df(p)}{dy} \\ \frac{df(p)}{dz} \end{bmatrix}$$

$$\frac{df(p)}{dx} \approx \frac{f(p + \begin{bmatrix} h & 0 & 0 \end{bmatrix}^T) - f(p - \begin{bmatrix} h & 0 & 0 \end{bmatrix}^T)}{2h}$$

$$\frac{df(p)}{dy} \approx \frac{f(p + \begin{bmatrix} 0 & h & 0 \end{bmatrix}^T) - f(p - \begin{bmatrix} 0 & h & 0 \end{bmatrix}^T)}{2h}$$

$$\frac{df(p)}{dz} \approx \frac{f(p + \begin{bmatrix} 0 & 0 & h \end{bmatrix}^T) - f(p - \begin{bmatrix} 0 & 0 & h \end{bmatrix}^T)}{2h}$$

And we have normal,

$$\vec{n} = normalize(\nabla f(p))$$

$$\approx normalize(\begin{bmatrix} f(p + \begin{bmatrix} h & 0 & 0 \end{bmatrix}^T) - f(p - \begin{bmatrix} h & 0 & 0 \end{bmatrix}^T) \\ f(p + \begin{bmatrix} 0 & h & 0 \end{bmatrix}^T) - f(p - \begin{bmatrix} 0 & h & 0 \end{bmatrix}^T) \\ f(p + \begin{bmatrix} 0 & 0 & h \end{bmatrix}^T) - f(p - \begin{bmatrix} 0 & 0 & h \end{bmatrix}^T) \end{bmatrix})$$

However, calculating it needs 6 calculations of the distance. Here, we used an optimized algorithm, which only needs 4 calculations. It is shown as follows.

$$k_0 = \begin{bmatrix} 1 & -1 & -1 \end{bmatrix}^T, \; k_1 = \begin{bmatrix} -1 & -1 & 1 \end{bmatrix}^T,$$

$$k_2 = \begin{bmatrix} -1 & 1 & -1 \end{bmatrix}^T, \; k_3 = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^T$$

$$\sum_i k_i f(p + hk_i) = \sum_i k_i f(p + hk_i) - f(p)$$

$$\approx \sum_i k_i \cdot h \nabla_{k_i} f(p) \approx \sum_i k_i \cdot h(k_i \cdot \nabla f(p))$$

$$= h \cdot \nabla f(p) \sum_i k_i^2 = h \cdot \begin{bmatrix} 4 \\ 4 \\ 4 \end{bmatrix} \cdot \nabla f(p)$$

So we have

$$\vec{n} = normalize(\sum_i k_i f(p + hk_i))$$

3.2 Phong Lighting Model

After we get the normal, we applied Phong Light Model to the point. To simplify our model and focus on the important part, we now only support one parallel light and multiple point lights in a scene, where every light can only contain 1 color, which means its ambient, diffuse and specular should have the same color.

4 SHADOW

4.1 hard Shadow

We shoot a shadow ray from the intersection point of the pixel ray and object to the light source. If the shadow ray intersects with anything, then the point is in the shadow.

4.2 Soft Shadow

Hard shadow is rough and false. To get better and more realistic result, we implemented soft shadow by approximation based on the characteristic of SDF. If a shadow ray does not intersect, but is close to an object, we say the point is in the penumbra. In SDF, it is convenient to get the distance from a point on the ray to the nearest object as well as the ray length from origin to that point; we evaluate how dark the surface is. Let h be the distance from a point on the ray to the nearest object and t be the ray length from origin to that point. Set a parameter k to influence how big the penumbra is, the larger k is, the faster $k \cdot \frac{h}{t}$ changes, the smaller the penumbra is; the larger $\frac{h}{t}$ is, the darker the penumbra is. Save the minimum value of $k \cdot \frac{h}{t}$ as the darkness of the penumbra. Note that if we use sphere tracing for the stepping of shadow ray, the penumbra might not be smooth enough. The reason is, when a line from a point on the ray to the surface is perpendicular to the ray, $\frac{h}{t}$ gets to its minimum, and $k \cdot \frac{h}{t}$, in this case, should be recorded as the darkness; however, if sphere tracing is used, this situation might not be get.

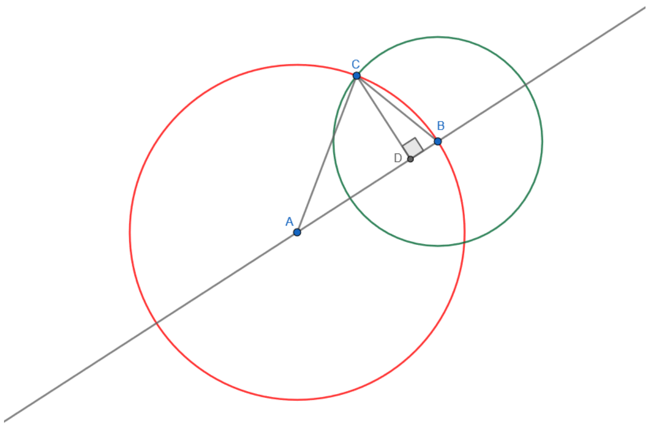


Figure 2: Soft Shadow Improvement

For example, in this picture, Point A is a sampled point, and according to sphere tracing, the next sampled point would be B. However, instead of B, point D is where $k \cdot \frac{h}{t}$ gets its minimum. Let d be the distance from A to its nearest object, d' be the distance from B to its nearest object, then:

$$p = \frac{|AB| + |AC| + |BC|}{2} = \frac{d + d + d'}{2} = d + \frac{d'}{2}$$

$$S_{\triangle ABC} = \sqrt{p(p - |AB|)(p - |AC|)(p - |BC|)}$$

$$= \sqrt{(d + \frac{d'}{2})(d + \frac{d}{2} - d)^2(d + \frac{d'}{2} - d')} = \frac{d'}{2}\sqrt{(d^2 - \frac{d'^2}{4})}$$

Let $y = \frac{d'^2}{2d}$, then when point D is sampled, $k \cdot \frac{h}{t} = k \cdot \frac{|CD|}{t - |BD|} = k \cdot \frac{\sqrt{d'^2 - \frac{d'^4}{4d^2}}}{t - \frac{d'^2}{2d}} = k \cdot \frac{\sqrt{d'^2 - y^2}}{t - y}$

5 AMBIENT OCCLUSION

We also implemented an approximated effect of ambient occlusion. From every intersection point, we shoot a ray along its normal and sample along the ray. We evaluate the strength of occlusion by both the distance d from the sampled point to its nearest surface and the ray length l from the ray origin to the point. Note that we used fixed step length instead of sphere tracing here. The bigger the ratio of d and l is, the lighter the ambient occlusion is. At last, we get the arithmetic mean of all the sampled points.

6 PRIMITIVE

To present each geometry object, we have a distance function to evaluate the distance from a point to the surface of the object. All primitive have a center to locate it, and different primitive may have different formulas to evaluate. Though the concept of OOP is quite popular today, since we have to using GPU to accelerate evaluation, we have to write the distance functions in OpenGL. The limitation of GLSL makes it hard to keep the code tidy and readable.

Here is an example of sphere in SDF format to show how we implement geometry objects. Suppose the center of the sphere with radius of r is S, then the distance from any point P in the world space to the sphere is

$$f(P) = |P - S| - r$$

Some primitive with curved surface may have high cost to evaluate the distance exactly, like cone. To make the rendering run in real-time, we choose some approximate formulas to avoid too many instructions. The result of those objects may be a little worse than exact ones, but we think it is worth to using approximate ways.

7 TRANSFORMATION

Since all objects are presented by mathematical formulas, translation and rotation can be done as a projection from a local space to world space. Transformations in traditional mesh rendering are to using a transformation matrix. A series of transformations can be combined with matrix multiplication.

$$T = \prod T_i$$

Basically, the transformations in SDF are the same as ones in traditional mesh rendering. The only different part is, we have

$$P' = T \times P$$

in traditional mesh rendering, but to do the same transformation, we use

$$P' = T^{-1} \times P$$

The reason to multiply by T^{-1} instead of T is that we have to do the inverse transformation of sampled point in the space.[3]

Notice any point rendered and showed on the screen is a point sampled in space. So any transformation which will compress or dilates the space and change the density of sampled point must recover the density. So transformation like scaling should not just using the transformation matrix.

It is very easy to make repetitions with SDF and const memory cost. Any primitive can be evaluated periodically.

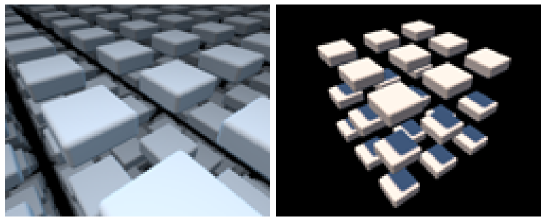


Figure 3: Infinite repetitions and finite repetitions. Image by Inigo Quilez.(https://iquilezles.org/www/articles/distfunctions/distfunctions.htm)

8 DEFORMATION

Deformation can make it easier to build some complex objects. This operation will cost a lot. In fact, though it is possible to using deformation to create some amazing geometry objects, there must be an easier and faster way to create geometry objects without deformation. The value of deformation is more in animation and SDF formatted mesh.

Using deformation may also change the density of the SDF, which has been mentioned. Exaggerated deformation can have some artifacts in rendering. The following picture of *Bending* deformation shows how the uneven SDF affects highlight and shadow area.

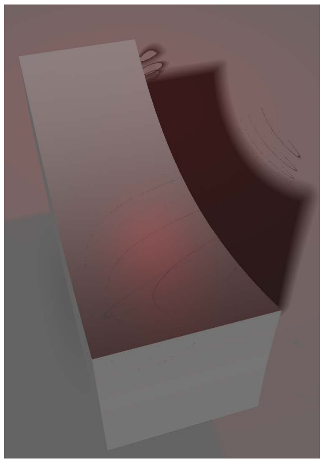


Figure 4: The highlight and shadow area have some stripes which are supposed being absent.

To avoid this kind of artifacts, a simple way is to reduce the step size while doing ray marching.

9 BOOLEAN OPERATION

We implemented some kinds of combinations between pairs of primitives corresponding to the basic Boolean operation, which are union (union two primitives together), subtraction (subtract one primitive from another), and intersection (get the intersecting part of two primitives).

All of our primitives are defined by a function, which gets a 3D vector, representing the sampled point in the space, as an input, and returns a floating point number, representing the distance from the

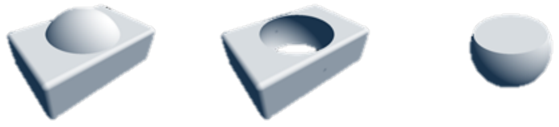


Figure 5: Union, Subtraction and Intersection. Image by Inigo Quilez.(https://iquilezles.org/www/articles/distfunctions/distfunctions.htm)

sampled point to the surface of the primitive, as an output. When we have two primitives in the space, we are actually getting two floating point numbers from them; and if we want to combine them together as a new primitive, only one floating point number should be returned.

The functions above show the relationship of the floating point number returned by the new combined primitive and the two original primitives when doing different Boolean operations.

Take the operation Union as an example: Let the input value of the two original primitives be d_1 and d_2, and the returned value of the new combined primitive be d. For any sampled points, there are five circumstances:

(1) The sampled point is outside of both of the primitives. ($d_1 > 0, d_2 > 0$)
(2) The sampled point is on the surface of one of the primitives, but outside of the other. ($d_1 = 0, d_2 > 0$)
(3) The sampled point is on the surface of both the primitives. ($d_1 = 0, d_2 = 0$)
(4) The sampled point is on the surface of one of the primitives, but inside of the other. ($d_1 = 0, d_2 < 0$)
(5) The sampled point is inside of both of the primitives. ($d_1 < 0, d_2 < 0$)

In the case of Union, we want to render the sampled point in circumstance 2 and 3. Since we only render points with $d = 0$, we can simply get d by calculating the minimum value of d_1 and d_2.

The other two Boolean operations are similar to Union. Note that Subtraction is not commutative.

10 SMOOTHING

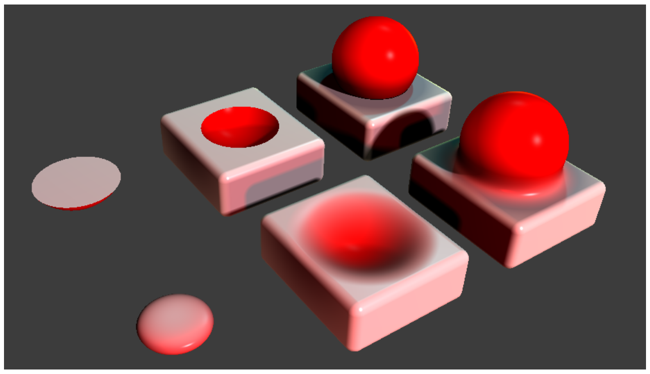


Figure 6: Example of Smoothing.

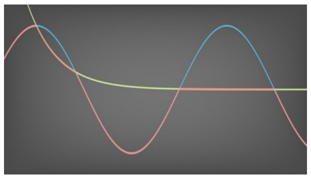
Apart from the three basic Boolean Operations, we also implemented the smoothing version of them. In the picture above, take the operation union as an example, we can see that when the generated object is smoothed, the part where the ball and the cube intersects seems to be melted.

To implement such a smoothing effect for union, instead of simply return $min(d_1, d_2)$, we designed a different function $smin$ to change the min used here [5] . The following formula shows how $smin$ is defined, where a is d_1, b is d_2, k is a smoothing parameter:

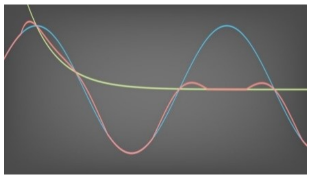
$$smin(a,b,k) = \begin{cases} a & , a-b > k \\ b & , a-b < -k \\ f(a,b,k) & , a-b \in (-k,k) \end{cases}$$

In this function, the returned value is linear interpolated from a and b with parameter h, which comes from $h = \frac{1}{2} + \frac{a-b}{2k}$, so that it is guaranteed that $h \in [0,1]$.

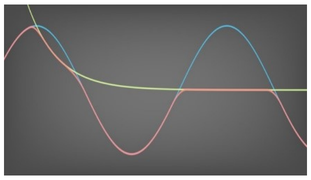
However, there are still problems in this approach. Before we talk about the problem, let's look at three figures.



(a) Using *min*.



(b) Using $f(a,b,k) = a + h(b-a)$.



(c) Using $f(a,b,k) = a(1-h) + hb - kh(1-h)$.

Figure 7: Example of smoothing. Green and blue curves are $a = sin(x)$ and $b = e^{-x}$. Red one is the value of *smin*.

Curves in the figure7b, which is not smooth and have abrupt changes. We can research its derivative to figure out what is going on:

$$\frac{df}{dx} = \frac{da}{dx} + \frac{dh}{dx}(b-a) + h(\frac{db}{dx} - \frac{da}{dx})$$

We can see when $h = 0$, $a - b = -k, \frac{df}{dx} = \frac{da}{dx} + k\frac{dh}{dx}$, but when $a - b < -k$, it is simply $\frac{df}{dx} = \frac{da}{dx}$. Clearly, there is discontinuity at the point $a\check{}b = -k$.

Note that for the case where $h = 1$ and $a - b = k$, $\frac{df}{dx} = \frac{da}{dx} - k\frac{dh}{dx}$ and similar discontinuity happens again.

What we want is to avoid those discontinuity, so we changed the first derivative to

$$\frac{df}{dx} = \frac{da}{dx} + \frac{dh}{dx}(b-a) + h(\frac{db}{dx} - \frac{da}{dx}) - k\frac{dh}{dx} + 2kh\frac{dh}{dx}.$$

And after integrating this function, we finally get $f(a,b,k) = a(1-h) + hb - kh(1-h)$, whose result is the figure7c.

We also implemented the smoothing of material, which can be seen in the above figure, where the red ball seems to be melted into the cube. Implementing smooth material is relatively simple, since we only had to calculate the interpolation of the two materials and does not have to care too much about the discontinuity.

11 REFECTION AND REFRACTION

The last thing is the implementation of reflection and refraction, including Fresnel Effect. The implementation here is similar to which is done when doing ray tracing, which both shoot a ray after deciding the exit direction at the intersection point of the ray and the object, and mix the shading effect of all the rays based on their reflectivity and refractivity. For the Fresnel Coefficient, we used Schlick Approximation.

$$R(\theta) = R_0 + (1-R_0)(1-cos(\theta))$$
$$R_0 = (\frac{n_1 - n_2}{n_1 + n_2})^2$$

As we are implementing Fresnel Effect, each ray will shoot a ray alone the reflection direction and refraction direction respectively after it is shot to the object's surface; let bounce time be n, and the nth bounce will generate at most 2^n rays. This is easy to implement with recursion, but since GLSL does not support recursion, we choose to implement this effect by using iteration. Since the number of bounces is bounded, we can maintain a 2^n array with rays inside. With each iteration, we go through the array and check if the ray intersects with an object; if so, we generate two new rays and put them into the array.

To implement refraction, we need to do two modifications with sphere tracing. On the one hand, we have to get the absolute value of the step length of sphere tracing, because when doing refraction, a refracted ray will do its step inside an object; on the other hand, the step length should also be timed by a coefficient $stepcale = 1 - \epsilon, \epsilon > 0$, since when determining whether a sampled point is inside or outside an object, $stepScale = 1$ will cause artifacts described above.

REFERENCES

[1] [n.d.]. *openFramework Documentation.* Retrieved July 6, 2020 from https://openframeworks.cc/documentation/

[2] Peer Play. [n.d.]. *Raymarching Shader - Unity CG/C Tutorial Chapter[1] = Shader Theory.* Retrieved July 6, 2020 from https://youtu.be/oPnft4z9iJs

[3] Inigo Quilez. [n.d.]. *Distance Function.* Retrieved July 6, 2020 from https://iquilezles.org/www/articles/distfunctions/distfunctions.htm

[4] Inigo Quilez. [n.d.]. *Raymarching Distance Fields.* Retrieved July 6, 2020 from https://www.iquilezles.org/www/articles/raymarchingdf/raymarchingdf.htm

[5] Vinicius Santos. [n.d.]. *Smooth Min Explained.* Retrieved July 6, 2020 from http://www.viniciusgraciano.com/blog/smin/