Interactive 3D Renderer of Implicit Surfaces

Student: Tristano Munini
Supervisor: Ivan Scagnetto

ACADEMIC YEAR 2020-2021

Contents

1 Introduction **1**

2 Signed Distance Functions **2**
2.1 Theory 2
2.2 Examples 4
2.3 Constructive Solid Geometry 5

3 Ray Tracing **7**
3.1 Rays and Cameras 7
3.2 Sphere Tracing 9
3.3 Lightning and Shadows 12
3.4 Anti-Aliasing 14

4 Implementation **16**
4.1 Compiling 16
4.2 Classes and Structure 16
4.3 Limitations and Future Improvements 17
4.4 Generated Images 18

1 Introduction

Starting idea and objectives.
What I've achieved.
Why ray tracing is interesting (and where can be used)(?)
1 page

2 Signed Distance Functions

2.1 Theory

As Hart says in the introduction of *"Sphere tracing: a geometric method for antialiased ray tracing of implicit surfaces"* [4], an implicit surface is defined by a function that, given a point in space, indicates whether the point is inside, on or outside the surface. The implicit surface can be defined either with an algebraic distance or with a geometric distance. As an example, the unit sphere can be defined with the algebraic implicit equation

$$x^2 + y^2 + z^2 - 1 = 0$$

or geometrically with

$$||x|| - 1 = 0$$

where $x \in \mathbb{R}^3$ and $||\cdot||$ is the Euclidean magnitude $\sqrt{x^2 + y^2 + z^2}$. In this document we will not analyze the differences between the two representations because we will use simple surfaces, for which the equations can be translated easily from one form to the other. Note that the two equations have the same value, i.e. Zero, when the provided point belong to the surface, and both became negative when the point is inside the surface.

Definition 2.1 (Distance Surface)**.** *A distance surface is implicitly defined by a function $f : \mathbb{R}^3 \to \mathbb{R}$ that characterizes $A \subset \mathbb{R}^3$, set of points that are on or inside the implicit surface:*

$$A = \{x : f(x) \leq 0\}$$

The surface can be also defined with $f^{-1}(0)$, which gives exactly the points on the surface. Even if the continuity of f and its negativity on the interior of the surface are not strictly necessary properties, they come useful in practice so, when possible, they are preferable. We can define the surface from the outside using a *point-to-set* distance:

$$d(x, A) = min_{y \in A}||x - y||$$

Thus $d(x, A)$, given a point $x \in \mathbb{R}^3$, returns the shortest distance to the surface defined by A. Note that here we interchange $d(x, A)$ with $d(x, f^{-1}(0))$ even if they are slightly different. We can do that because usually we'll handle points that aren't in surfaces interiors[1].

Definition 2.2 (Signed Distance Bound)**.** *We say that $f : \mathbb{R}^3 \to \mathbb{R}$ is a signed distance bound of its implicit surface $f^{-1}(0)$ if and only if*

$$|f(x)| \leq d(x, f^{-1}(0)) \tag{1}$$

[1] In the implementation one must consider this cases e.g. the camera (wrongly) is inside an object

In other words, the definition says that f is always at least as cautious as the true distance function $d(x, f^{-1}(0))$. This means that we can move x by $d(x, f^{-1}(0))$ in every direction and in the worst case (or best) we'll just hit the surface. Moving in this way we'll never put the point at the interior of the surface.

Definition 2.3 (Signed Distance Function)**.** *We say that f is a signed distance function (SDF) when holds*

$$|f(x)| = d(x, f^{-1}(0)) \tag{2}$$

An SDF returns the exact distance to the surface, so moving x by this distance (in the right direction) means putting it exactly on the implicit surface $f^{-1}(0)$. In the introductory article *"Rendering Implicit Surfaces and Distance Fields: Sphere Tracing"*[5] by `scratchapixel.com`, these concepts are emphasized calling them respectively: *distance underestimate (implicit) functions* (DUFs) and *distance implicit functions* (DIFs). Remembering that $DUF(x) \leq DIF(x)$ for every DUFs and DIFs respecting the definitions above.

For simple shapes we can derive SDFs by hand, but when it's not feasible we can use the Lipschitz constant λ, as Hart[4] suggested.

Definition 2.4. *Lipschitz Function We say that a function $f : \mathbb{R}^3 \to \mathbb{R}$ is Lipschitz over a domain D if and only if there is a positive, finite, constant λ such that*

$$|f(x) - f(y)| \leq \lambda ||x - y|| \tag{3}$$

Such λ is called the Lipschitz constant.

Of course there is no upper limit to λ, but there is a lower bound. Let $Lip(f)$ be the function returning such minimum value. Manipulating Equation 3 we can observe that

$$\begin{aligned}
\lambda &\geq \frac{|f(x) - f(y)|}{||x - y||} \\
\lambda &\geq \frac{f(x) - f(y)}{||x - y||} \\
\lambda &\geq \lim_{||x-y|| \to 0} \frac{f(x) - f(y)}{x - y} = f'(x)
\end{aligned}$$

the last step, given the continuity of f, permits us to estimate a safe lower bound for the Lipschitz constant. To be precise what we have to do is to compute the derivative maximum value, therefore we need to find the solutions to the function second order derivative $f''(x)$.

The following theorem (from [4]) explains why having such a λ (or a suitable approximation) is so important: because it gives us a general method for computing a DUF for any Lipschitz function.

Theorem 2.1. *Let f be Lipschitz with Lipschitz constant λ. Then the function f/λ is a SDF of its implicit surface.*

Proof. Given a point x and a point $y \in f^{-1}(0)$ such that

$$||x - y|| = d(x, f^{-1}(0)) \tag{4}$$

Because f is Lipschitz, holds

$$|f(x) - f(y)| \leq \lambda ||x - y||$$

and because y is on the surface $f(y) = 0$, so

$$\begin{aligned} |f(x)| &\leq \lambda ||x - y|| \\ |f(x)| &\leq \lambda d(x, f^{-1}(0)) \quad \text{by Equation 4} \end{aligned}$$

Hence $\frac{|f(x)|}{\lambda} \leq d(x, f^{-1}(0))$ is a signed distance bound (DUF) for any Lipschitz function f. □

One secondary thing to note is that an SDF can return any negative number when the given point is inside the surface. A common choice is to give a constant negative number i.e. -1.

2.2 Examples

Hart's method is a general approach but in practice we can create by hand several DIFs. Different SDFs can be found in both [4, 5], but one of the most comprehensive lists is at [3] by Inigo Quilez, creator of shadertoy.com and "SDF artist".

Below are reported three simple images generated with a modified version of the scratchapixel's code[5]. Pixels' color depends on the absolute distance[2] from

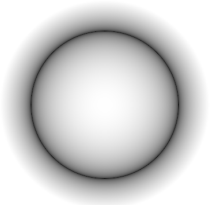


Figure 1: A circle

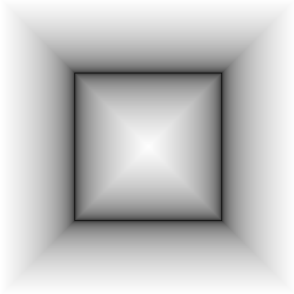


Figure 2: A square

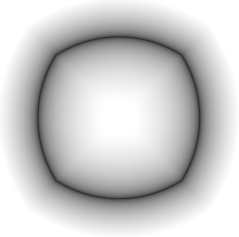


Figure 3: A squircle

the surface. Note that this images can be seen as 2D section of 3D objects done with a yz-aligned plane[3] passing through the origin. The distance equation for the circle is $x^2 + y^2 - 1$, meanwhile for the square it is $max(|x|, |y|) - 1$. To get a

[2] gamma corrected and clamped in $[0, 1]$

[3] assuming that x grows to the right, y up, and z "towards" the image

better understanding of the square's SDF we suggest this Quilez's video [2] (and the others on his channel). The squircle is the mixing of the square and circle SDFs and its implicit equation is $k * (max(|x|, |y|) - 1) + (1 - k) * (x^2 + y^2 - 1)$, where $k \in [0, 1]$. This is only one example of the operation we can perform on implicit surfaces.

2.3 Constructive Solid Geometry

One of the major benefits we get using SDFs is the easiness with which we can create complex shapes from few primitives, technique known as Constructive Solid Geometry (CSG). CSG with Boolean operations is largely used in CAD software. With DIFs we can simulate:

- union with the minimum, because we "stop" at the first surface that as been found
$$min(f_1, f_2)$$
- intersection with the maximum, because we "ignore" the first surface if there's another after
$$max(f_1, f_2)$$
- subtraction with the maximum between the first surface and the opposite of the second
$$max(f_1, -f_2)$$
- mixing, creating a shape that is in between of the given two, with interpolation of distances
$$k * f_1 + (1 - k) * f_2 \quad \text{with } k \in [0, 1]$$

Here we listed the most common operations, but in theory we can use any mathematical function: sine, cosine, modulo, exponential, etc... All these operations can be arranged in a tree structure, even if the optimal evaluation order may not be always obvious.

Note that the signed distance bound functions are closed by CSG operations so we do not have to handle new cases when we manipulate the resulting shapes.

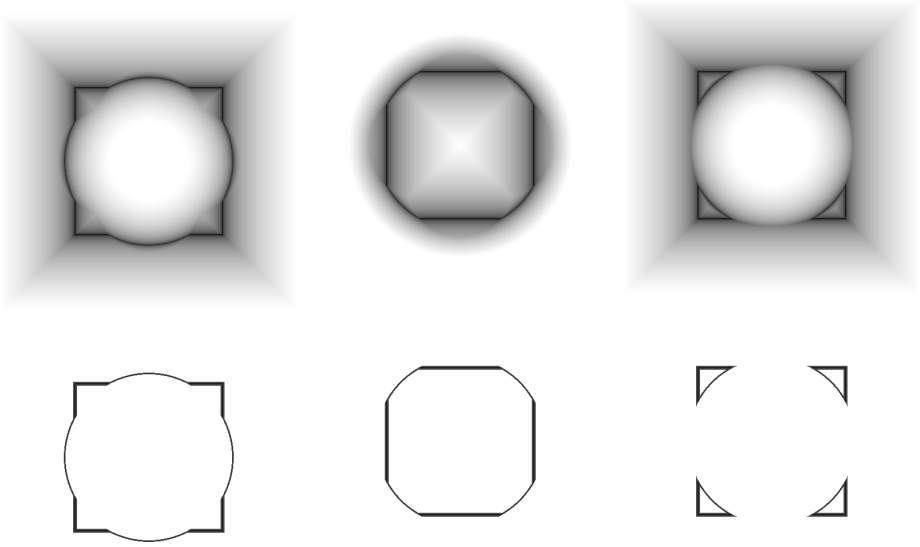


Figure 4: Square-Circle union and its contour

Figure 5: Square-Circle intersection and its contour

Figure 6: Square-Circle subtraction and its contour

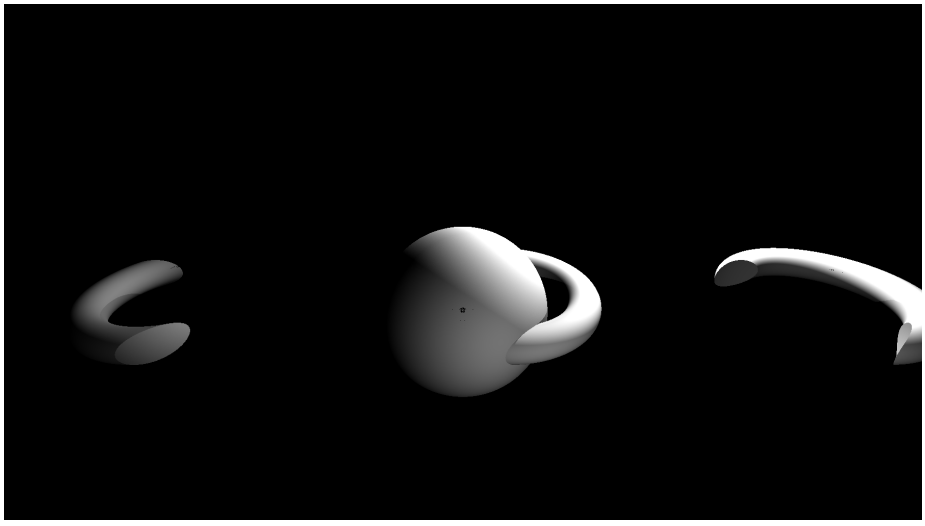


Figure 7: Left to right: intersection, union and subtraction of a sphere and a torus. TODO produce better image

3 Ray Tracing

In the introduction of the book "Ray Tracing from the Ground Up" [7] the author loosely defines two classes of renderers: projective algorithms and image-space algorithms. The former nowadays is the most popular because it's based on the graphics pipeline model and can easily take advantage of modern graphics cards. The basic idea is to project the objects on the scene into a 2D plane and then compute the appearance of the objects that can be seen. On the other hand, the image-space algorithms start from the pixel and try to figure out where the light came form for that pixel. Following the light's path we can compute hard shadows, soft shadows, reflection, refraction, depth of field, motion blur, caustics, ambient occlusion, indirect lighting and other effects. But following every light ray's bounce it's extremely costly, especially if there are several light sources in the scene, and that's why this approach historically wasn't the one to be used for interactive application. Instead, given the high quality and fidelity this method can achieve, it's been widely used to create movies, clips and images, In these application, after an initial low-res prototyping, the computer can elaborate the final product even for days.

3.1 Rays and Cameras

As we said before, the idea behind a ray tracer is to generate a ray for each pixel and check whether the ray hits a surface. If so it has also to find out if a light source lighten the point or if it is in shadow.

The easiest way to generate such rays is the following: imagine to put a 2D rectangle into a 3D scene; draw a square grid on the rectangle and for each square project rays perpendicular to the rectangle's faces; if a ray hits an object color the corresponding square with the surface color; if a ray hits no objects then color the pixel with black (or a default color). Because the rays are parallel to one another this method gives us a really simple orthogonal camera.

Orthogonal cameras are useful but in general prospective cameras are preferred because they resemble more the human eye. To model a perspective camera all we have to do is to put a point behind the rectangle with the grid and generate rays that start from the point and pass through the grid's squares. The common choice is to make the rays pass through the squares' center. The model just described is called a pinhole camera, shown in Figure 8, because mimics the real life cameras. One trick here is that we put che plane before the point so we don't have to flip the image before showing it to the user.

A ray is defined as a point and a direction, as shown in Figure 9, in formulas:

$$r(t) = O + tR$$

where $O, R \in \mathbb{R}^3$ and R is a unit vector. At each time t we can compute a point position on the ray, in this way the ray tracing algorithm can check for each t whether the ray intersected a surface or not.

When generating rays we have to take in consideration several factors: camera position and orientation are the more obvious, but also the field of view and

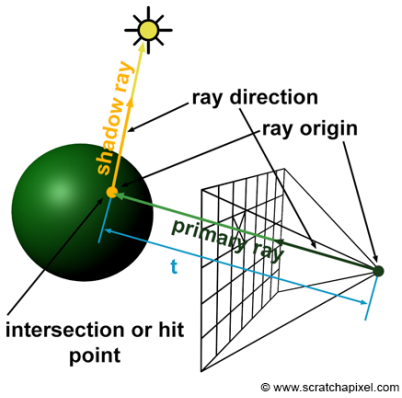


Figure 8: A pinhole camera inside a scene with a sphere and a light source

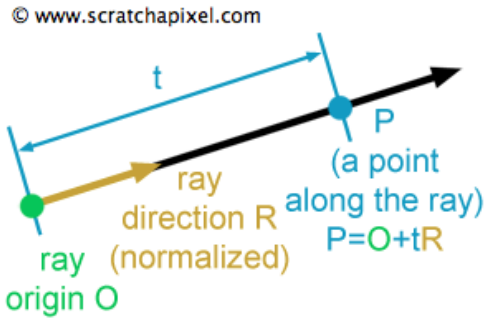


Figure 9: Ray model

the image aspect ratio. First of all we need to derive the rays' direction with the pixel's centers on the image plane (the rectangle with a grid) and the camera origin (also called *eye*). But these two values refers to different coordinate systems because the pixel position refers to the top left corner of the image plane and the camera origin is defined in world coordinates. Here with world coordinates we mean the coordinate system with which the object (and the camera) are placed in the scene. One way to solve this problem is to produce the rays as if the camera was placed in the origin looking towards $-z$ and then translate and rotate the rays according to the true camera position and orientation.

In order to do so we need to remove the dependency from the actual output image size. Given two coordinates in raster space we normalize them by dividing respectively by the image width and height. Now the coordinate are in Normalized Device Coordinates (NDC) space. Actually the precise operation is the following, in which we also offset the coordinate to be in the pixel's center:

$$PixelNDC_x = \frac{(Pixel_x + 0.5)}{ImageWidth}$$
$$PixelNDC_y = \frac{(Pixel_y + 0.5)}{ImageHeight}$$

Now we have to map this coordinates form $[0, 1]$ into $[-1, 1]$ because the eye of the camera is at the origin, so the center of the image plane (also called screen or screen plane) is on the z-axis. In practice we are aligning the sign of the four quadrants of the xy-plane in world coordinates with the image plane.

$$PixelScreen_x = (2 * PixelNDC_x - 1) * AspectRatio$$
$$PixelScreen_y = 1 - 2 * PixelNDC_y$$

Remember that we want a right handed system so we map: the top left cor-

ner, $(0, 0)$ in raster space, to $(-1, 1)$ in screen space; the bottom right corner, $(ImageWidth, ImageHeight)$ in raster space, to $(1, -1)$ in screen space; and the last two accordingly.

The $AspectRatio$ used in the formula equals $ImageWidth/ImageHeight$ and we must consider it when the output image is not squared. In fact we need to produce more pixel in one direction otherwise the generated image will look squished.

One of the last thing to consider is the field of view (fov), the view angle of the camera. Assuming that the distance between the camera eye and the image plane equals to 1, we want to know by how much enlarge or shrink the image plane horizontally and vertically. Knowing the fov angle α we just need to multiply the screen coordinates by $\tan(\alpha/2)$.

The last thing to take in consideration is that our camera may not be always positioned in the world origin and look toward $-z$. One of the possible solutions is to consider the camera origin, also called eye, as a point with an associated normal direction, i.e. the camera's view direction. With this in mind we can move the point and its normal in the homogeneous 4D space with a 4D matrix. In this particular case we use the camera transformation matrix because we want to move from camera coordinates to the world ones.

3.2 Sphere Tracing

Introduced by Hart[4] in 1996, the sphere tracing algorithm gives a better alternative to other ray tracers like ray marching. In ray marching each ray is analyzed by constant steps of t. We can detect a ray-surface contact using the sign of the distance function: when the ray enters the surface the distance became negative. Let's say that for a ray r, $d(r(t_k), S) > 0$ and $d(r(t_{k+1}), S) < 0$, for some instant t_k and the following instant t_{k+1} and an implicit surface S. Then using bisection on the segment between the two points $r(t_k)$ and $r(t_{k+1})$ we look for the point P on the segment for which $d(P, S) = 0$. To actually find this point P on the surface S can be costly, so usually a point that is as near to the surface as a given threshold is found.

We can speed up the algorithm by increasing the step size, i.e. $|t_k - t_{k+1}|$, but doing so leads two problems: performing bisection is slower because the segments are longer; if an object is small the algorithm may evaluate the ray just before and after the object, thus not seeing it. While the first problem can be managed, the second can cause completely wrong images to be produced. What we would like is an algorithm with an adaptable step size, to go fast when there are no surfaces near and to slow down when there are objects in the proximity: sphere tracing does that.

Remember that before we said that $d(x, S)$ for some $x \in \mathbb{R}^3$ means we can move x in every direction by $d(x, S)$ being sure at worst to just hit the surface. In other words we are drawing a safe sphere around the point, in which there is no surface. Form the surface perspective this sphere is called an *unbounding sphere*[4], and the union of all the unbounding spheres describes the surface because no sphere penetrate the object. The concept is approximately the

opposite of *bounding volume*, a volume that surrounds the object, because here we define an area of space not containing the object. In sphere tracing the ray-surface intersection is determined by a sequence of unbounding spheres, as shown in Figure 11.

```
t = 0
while (t < D) {
  d = f(r(t))
  if (d < ϵ) {
    // intersection
    reutrn t
  }
  t += d
}
// no intersection
return ∅
```

Figure 10: Pseudo code for the geometric implicit surface rendering algorithm

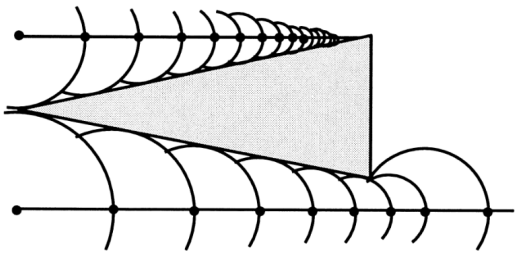


Figure 11: A hit and a miss (image from [4])

The pseudo algorithm for rendering one implicit surface is listed in Figure 10. In the pseudocode D is the maximum ray traversal distance, ϵ is the minimum distance from a surface to consider a hit, t and $r(t)$ are the usual ones and f can be a DIF or a DUF. As said before the DUFs are "more cautious" because are a lower bound to the maximum distance traversable without hitting the surface. When we can't use the exact signed distance function of the surface we can use safely its bound, knowing that the convergence will be slower because the algorithm will make more steps. The following theorem from [4] shows that the speed drop is negligible because the algorithm is still linear even if we are using a signed distance bound. We look at sphere tracing as a root finding algorithm.

Theorem 3.1. *Given a function $f : \mathbb{R} \to \mathbb{R}$ with the Lipschitz bound $\lambda \geq Lip(f)$ and an initial point t_0, sphere tracing converges linearly to the smallest root grater than t_0.*

Proof. We can write the sphere sequence as

$$t_{i+1} = g(t_i) = t_i + \frac{|f(t_i)|}{\lambda} \tag{5}$$

to emphasize the similarities with the Newton's root finding method. Let r be the smallest be the smallest root greater than the initial point t_0. Since $f(r) = 0$, then $g(r) = r$, and at any non root $|f(t_i)|/\lambda$ is positive. Hence Equation 5 converges to the first root. Without loss of generality we assume f to be non-negative in the region of interest, eliminating the need for the absolute

value. The Taylor expansion of $f(t_i)$ about the root r is:

$$g(t_i) = g(r) + (t_i - r)g'(r) + \frac{(t_i - r)^2}{2}g''(\tau) \tag{6}$$

for some $\tau \in [t_i, r]$ and $g'(r) = 1 + f'(y)/\lambda$. The error term becomes:

$$\begin{aligned} e_{i+1} &= t_{i+1} - r = g(t_i) - g(r) = \\ &= g'(r)e_i + \text{higher-order terms} \end{aligned}$$

Since $g'(r)$ is constant in the iteration Equation 5 converges linearly to y. □

Corollary 3.1.1. *Sphere tracing converges quadratically if and only if the function is steepest at its first root.*

Proof. In the event that $f'(r) = -\lambda$, the linear term of the error of Equation 6 drops out, leaving the quadratic and higher-order terms. □

One thing to highlight is about the rays that pass near a surface without hitting it. Such rays will slow down because their distance to the surface becomes smaller and smaller until they pass over it, as shown in Figure 11. This effect can be seen in practice coloring the pixels with respect to the number of steps the ray had to do before hitting something or traversing the maximum distance, take a look at Figure 12.

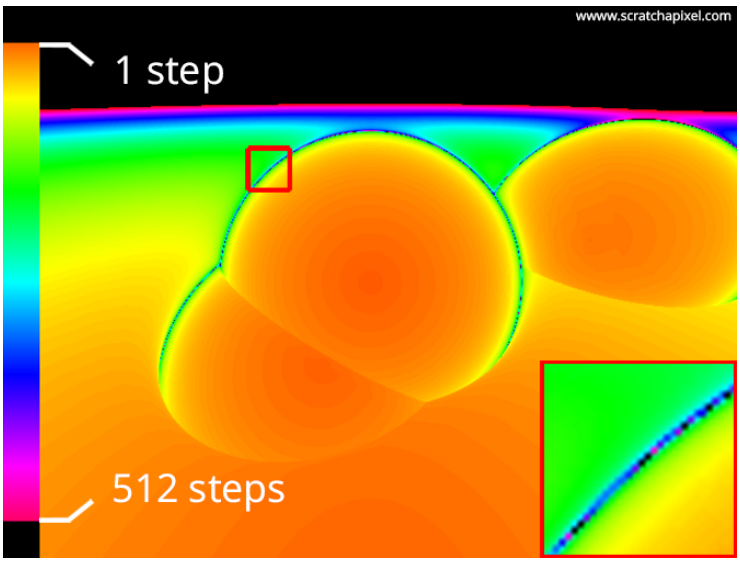


Figure 12: This is a scratchpixel img, TODO generate img with my version

3.3 Lightning and Shadows

Shading is a complex field and here we address only the problem of computing simple lighting and hard shadows, leaving for future improvements topics as non punctual lights, reflection, refraction, and semi-transparent objects.

Even in this simplified setting we need to compute the surface normal at a point, that's because we need to know the quantity of light that the point receives, i.e. if the surface is facing or not the light source. Because we are handling implicit functions we can't rely on already existing information, as happens with graphics pipeline rendering and *.obj* files for example. We need to compute the surface's normal at a point with only its implicit definition. To do so we can take advantage of the gradient of the function. Remembering that the gradient is the rate-of-change of the function over a direction, given f we want to compute the gradients with respect to $(1,0,0)$, $(0,1,0)$ and $(0,0,1)$. These results combined will form a 3D vector, which after being normalized becomes the normal we are looking for.

More formally, we define ∇f as the gradient of the implicit function $f(x,y,z)$ and let $P = \langle x_0, y_0, z_0 \rangle$ be a point on the surface, thus $f(x_0, y_0, z_0) = 0$. All we need to do is to observe the rate-of-change in the proximity of P, this can be done numerically with

$$\begin{aligned}
\nabla f_{x_0} &= f(x_0 + \delta, y_0, z_0) - f(x_0 - \delta, y_0, z_0) \\
\nabla f_{y_0} &= f(x_0, y_0 + \delta, z_0) - f(x_0, y_0 - \delta, z_0) \\
\nabla f_{z_0} &= f(x_0, y_0, z_0 + \delta) - f(x_0, y_0, z_0 - \delta) \\
\nabla f &= \langle \nabla f_{x_0}, \nabla f_{y_0}, \nabla f_{z_0} \rangle
\end{aligned}$$

The following steps assure us that ∇f has actually the normal's direction. First we define

$$h(t) = \langle x(t), y(t), z(t) \rangle$$

to be a parametric definition of a curve that lies on the surface and that passes through P. We also define

$$g(t) = f(h(t)) = 0$$

Note that exists a particular t_0 such that

$$g(t_0) = f(h(t_0)) = f(x(t_0), y(t_0), z(t_0)) = f(x_0, y_0, z_0) = f(P)$$

Now we can calculate the gradient at

$$\begin{aligned}
\frac{dg}{dt} &= \left.\frac{\partial f}{\partial x}\right|_P \left.\frac{dx}{dt}\right|_{t_0} + \left.\frac{\partial f}{\partial y}\right|_P \left.\frac{dy}{dt}\right|_{t_0} + \left.\frac{\partial f}{\partial z}\right|_P \left.\frac{dz}{dt}\right|_{t_0} = \\
&= \left\langle \left.\frac{\partial f}{\partial x}\right|_P, \left.\frac{\partial f}{\partial y}\right|_P, \left.\frac{\partial f}{\partial z}\right|_P \right\rangle \cdot \left\langle \left.\frac{dx}{dt}\right|_{t_0} \left.\frac{dy}{dt}\right|_{t_0} \left.\frac{dz}{dt}\right|_{t_0} \right\rangle = 0
\end{aligned}$$

The equation comes from the higher dimensions version of the chain rule $g'(t) = h'(t) \cdot f'(g(t))$. The vector form comes from $a \cdot b = a_x b_x + a_y b_y + a_z b_z$, and highlight the perpendicularity between the two vectors.

Now that we have the surface normal N at P we can use it inside a rendering equation that, in simple words, gives the amount of light reflected by the surface. Putting in formulas

$$\frac{(n \cdot l) c_{light}}{4\pi}$$

where n is the surface normal, l a unit vector pointing the light and c_{light} its color. Remember that the dot product $n \cdot l$ gives the cosine of the angle of incidence of the light rays. The denominator comes from the solution of an integral over a hemisphere centered in P, more on shading in [1]. If there are more light sources we can just iterate over them and sum the contribution of each. We can also consider adding an ambient light to give a minimum lighting to all the surfaces, thus removing the completely black areas from the image.

For what concerns shadow we can just cast a second ray from P towards a light. We follow the ray until we hit either a surface or the light itself: in the first case we just stop computing this light's contribution to the shading of P (its contribution is equal to zero because no light ray can reach the point); in the second we procede as described before. The ray can be computed with a modified version of the sphere tracing algorithm that stops at the first surface hit.

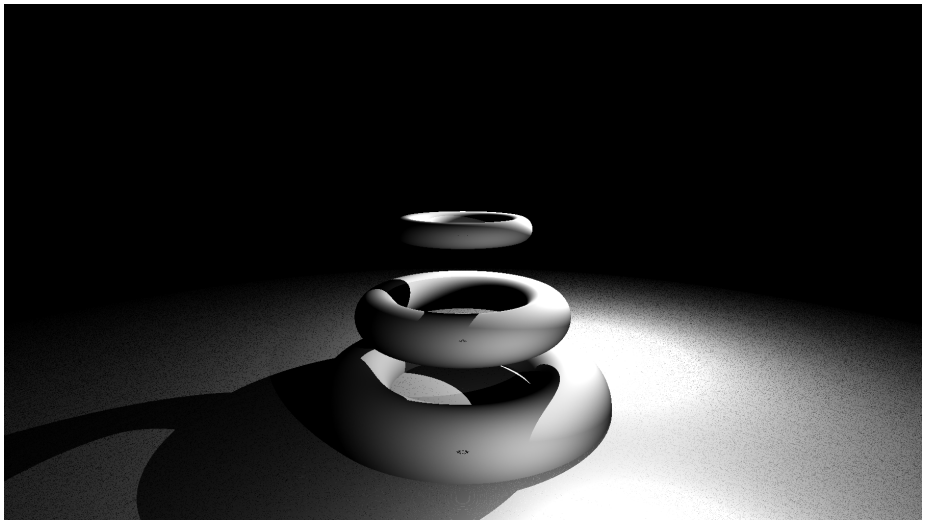


Figure 13: A scene with 3 torus, a big sphere (the terrain) and two point lights rendered with hard shadows and no anti-aliasing.

3.4 Anti-Aliasing

Since in ray tracing a scene is discretized in a finite number of pixel using sampling, ray tracing is subject to aliasing. This topic is discussed extensively in [7] and with a particular attention towards sphere tracing and Hart's method in [4, 5], It's important to note that we can not resolve in a definitive way the aliasing problem, we can only mitigate its effects with anti-aliasing techniques.

One of the easiest approach is super sampling in which we trace several rays for each pixel and average the results, as described and implemented in [6]. The sampling can be done in different ways and with different weights for each ray contribution. A common choice is to decide the number of rays per pixel and give a random perturbation to each rays' direction. Obviously this approach weighs on the hardware, because computationally is similar to computing a bigger image.

In [4] Hart proposes an interesting method that takes advantage of sphere tracing algorithm's nature: we can replace rays with cones. Visually the idea is represented in Figure 14, here the cone is approximated with the pixel's volumetric extension. Intersecting cones with objects is difficult but here we

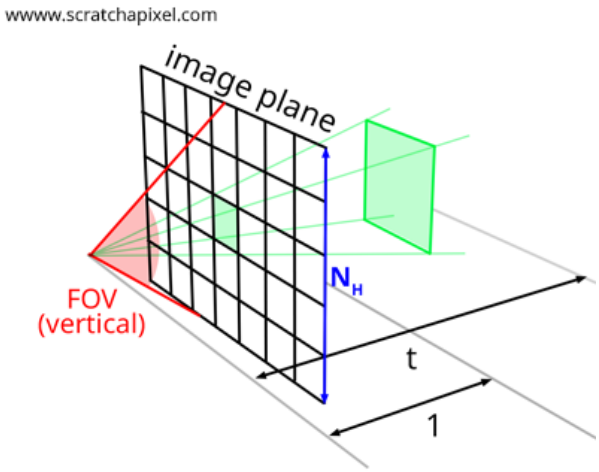


Figure 14: Pixel volumetric extension

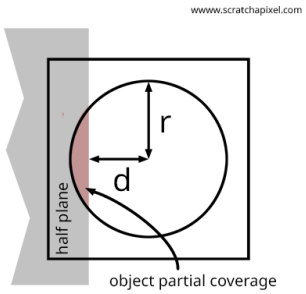


Figure 15: A section of cone, pixel extension and unbounding sphere

can use the concept of unbounding spheres. Assuming image plane 1 unit (in world coordinates) away form camera's eye, with

$$r = 2\frac{\tan(\alpha/2)}{N_H}t$$

where α is the fov of the camera, we can compute the pixel size at distance t. Pixel size at t can be used to estimate the cone radius at t. The formula is similar to the one used to handle the camera fov, but here we divide the value by N_H because we want the size relative to a single pixel.

As sphere tracing progresses along the ray, if the distance d returned by a DUF produces an unbounding sphere that projects to an area smaller than a pixel on the screen, then this is considered a cone intersection. That's because the sphere at least touches the surface and the cone contains the sphere, so it must be that also the cone intersects the surface. In this case the point along the ray is treated as if it were on the surface, and the corresponding pixel is shaded as in a surface-hit situation. A section at t of the cone, a sphere, a pixel and a surface is shown in Figure 15. Following the description in [5], the anti-aliasing algorithms has the following steps:

1. for each pixel
2. init pixel's color with black;
3. evaluate the ray at t;
4. if the object is at a distance $d \leq \epsilon$ then the ray intersects the object. Shade the object, save the result in the pixel and return;
5. if the object is at a distance $d \leq r$ then add the shading result blending it with the pixel's color;
6. else move along the ray by d and repeat the loop;
7. when $t > D$, return the pixel's color.

Every time $d \leq r$ we compute a partial coverage of the intersected object over the area of the pixel, look at Figure 15. We accumulate the result with alpha blending, using the following formulas:

$$\alpha = \frac{1}{2} - \frac{d\sqrt{r^2 - d^2}}{\pi r^2} - \frac{1}{\pi} \arcsin \frac{d}{r}$$
$$c = c_1 + c_2(1 - \alpha_1)$$

Even if overall the method seems easy in the implementations we need to be careful, because we risk to add several contributions of the same object. One solution could be storing each possible contribution and use only the one with the smallest d to actually modify the pixel's color. This must be done whenever the ray passes sufficiently near to a surface.

4 Implementation

In this section I provide a high level description of the C++ implementation of an SDF Renderer utilizing sphere tracing. The described code can be found at Nyriu/RayTracer.

4.1 Compiling

The source code can be compiled with the common CMake workflow:

```
mkdir build
cd build
cmake ..
make
cd ..
./main
```

The project was written and tested on a Manjaro operating system, but it should compile on other machines. There are two major dependencies: SDL2 and OpenGL Mathematics (GLM). The latter can be easily removed by re-implementing some of the matrix and vector operations. SDL2 is used to open a window that shows the rendered images. This dependence can be also removed by commenting out the relative lines from the *Renderer* class and not compiling the *Window* class.

4.2 Classes and Structure

The main header files and classes are:

- *ImplicitShape.h* in which is defined the homonymous abstract class and its children *Sphere*, emphTorus, etc... Each class represents an implicit surface described via an SDF function, and provides methods to compute point-to-set distances and normal at a given point. The shapes have other properties such as the diffuse and specular colors. In this file are also defined the CSG operation, here considered as a particular case of implicit shape. Union, subtraction and intersection can be used to form a structure that resemble a binary tree;
- the *Camera* is a perspective camera and can be placed in a scene, oriented with a target point or with a given view direction, and has a modifiable field of view;
- *Ray* consists in a point and a direction and gives the possibility to calculate the ray position at a given t;
- *Light.h* describes an abstract *Light* class that is implemented with the *PointLight* class. A light exposes publicly its color, position and intensity;

- a *Scene* consists of a series of shapes, lights and a camera. The implementation offers methods to add and retrieve objects in the scene;
- the *Tracer* implements the sphere tracing algorithm and is responsible of pixel shading. Here the ray's path is followed until its first bounce, then shadowing is verified and shading is computed. If there's no bounce then a default color is returned.
- the *Renderer* asks the camera rays and passes them to the tracer. When the tracer has returned the pixel's color, the renderer saves it to an *Image* that is then displayed in a *Window* or saved to file;
- *Color* abstraction of an RGB triplet;
- *Image* squared 2D grid of pixels, each accessible and modifiable with xy coordinates. As usual the top left corner is $(0, 0)$, x grows towards the right and y towards the bottom;
- the *Window* handles SDL calls to open, close and update an SDL window.
- *geometry* contains geometric abstractions such as 3D points, vectors and matrices and operations on them;
- *utilities.h* contains miscellaneous functions both used in the implementation and during implementation.

The file *main.cpp* can be modified to load different scenes and to change the output image properties, here we can also enable or disable the SDL window rendering only to file in PPM format. Inside *scenes.cpp* different scenes are defined with different shapes and light as an example. A scene should also contain a camera, if that's not the case the renderer's camera will be used (if provided).

4.3 Limitations and Future Improvements

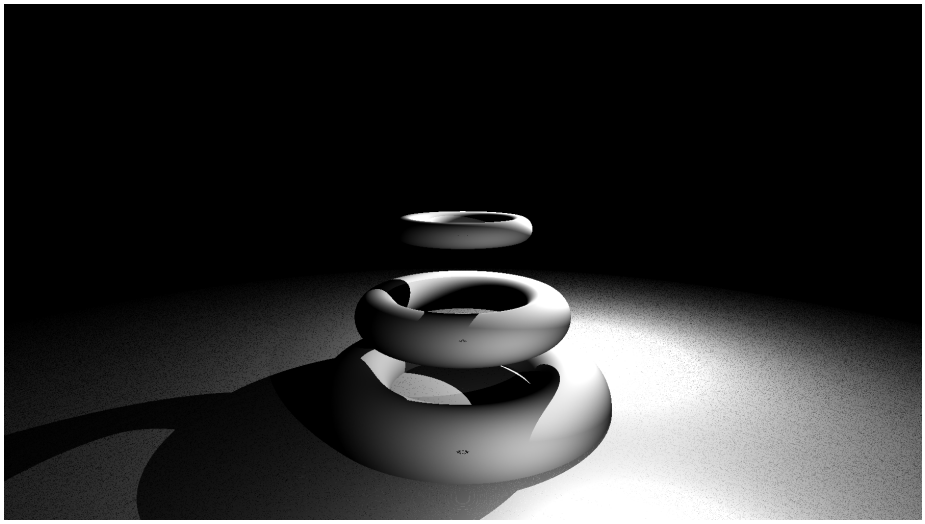
The current implementation doesn't permit real time interaction. That's because the code run exclusively on CPU and sequentially. This limitation can be overcome re-implementing the shading part in parallel e.g. with CUDA. Since each ray does not depend on the others, the ray computation can naturally benefit from parallelism.

As a consequence of the computation being sequential, the rendering has a single buffer. Thus the frame rate is limited by the generation of every single image. Also the code waits the image to be fully completed before showing it to the user.

Another limitation is that the scene must be described in C++ and there is no hot reloading if something in the scene changes. In future could be interesting to create a simple parser for plain text files that describe scenes.

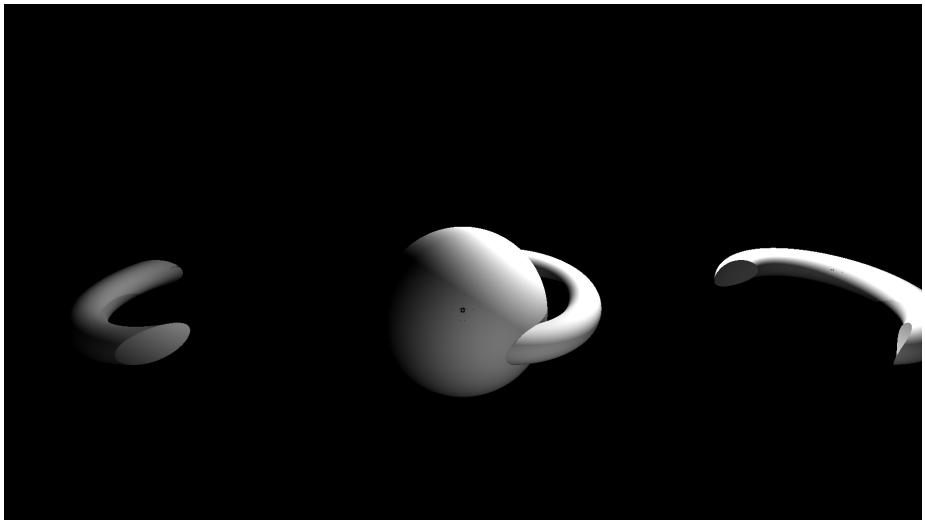
4.4 Generated Images

create toruses without self shadowing



show lambertian and different lighting

show CSG and highlight shadows on union shapes



References

[1] Tomas. Akenine et al. *Real-Time Rendering*. CRC Press.

[2] *Deriving the SDF of a Box*. URL: https://www.youtube.com/watch?v=62-pRVZuS5c.

[3] *Distance Functions*. URL: https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm.

[4] Jhon C. Hart. "Sphere tracing: a geometric method for antialiased ray tracing of implicit surfaces". In: *The Visual Computer* (1996).

[5] *Rendering Implicit Surfaces and Distance Fields: Sphere Tracing*. URL: https://www.scratchapixel.com/lessons/advanced-rendering/rendering-distance-fields/introduction.

[6] Peter Shirley. *Ray Tracing in One Weekend*.

[7] Kevin Suffern. *Ray Tracing from the Ground Up*. A K Peters, Ltd., 2016.