Efficient SDF Collision Point Detection

Kamal Al-Ameri

August 2025

1 Abstract

Signed Distance Functions (SDFs) have been widely used in animation, but have limited use in interactive contexts. This is partially due to the computational overhead of rendering SDFs, but is also due to the complexity of collision detection between general SDFs. Most previous methods either relied on generating a discrete surface from the SDF, such as voxelization or polygonization, and other methods did not discretize the SDF, but are too expensive for real time applications, such as random sampling. We propose an algorithm that utilizes properties of SDFs to efficiently find rather accurate collision points from two arbitrary SDFs, assuming that the SDFs satisfy certain conditions that are true for most, if not all, SDFs.

2 Introduction

We will briefly introduce a key concept fundamental to this text and explain the short comings of previous methods, before explaining our proposed approach.

2.1 Signed Distance Functions

A signed distance function is a scalar-valued function in the form of $f(\vec{x})$, such that it returns the distance from any point in space to the surface of the object it represents. The distance is signed because if the point lies outside of the shape, the distance is positive. But if it lies inside of the shape, the distance is negative.

For example, the SDF of a sphere centered at C with radius R is $f(x) = ||x - C|| - R$

As the value of the SDF is 0 when the point is on the surface, this means that the SDF represents an implicit surface.

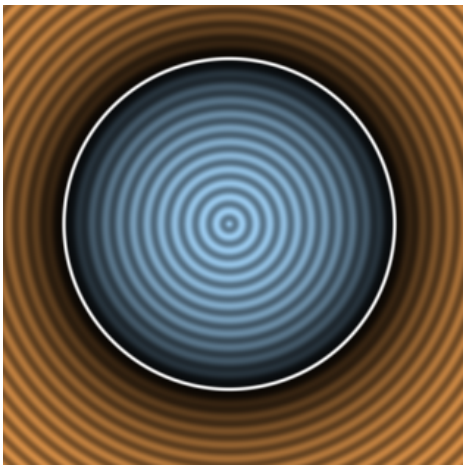

Figure 1: Visualizing the SDF of a Two-Dimensional Circle

2.2 Shortcomings of Previous Methods

2.2.1 Discretization-Based Methods

Discretization-based methods create a discrete geometry, that is easier to work with, from the implicit surface of the SDF. Note that all of these methods require regeneration of the discrete geometry whenever the implicit surface changes. Common choices are:

- Voxelization, which splits the space which the SDF occupies into evenly sized surfels, typically pixels for two-dimensional contexts or voxels for three-dimensional contexts, and prunes surfels that do not intersect the surface of the SDF. The space complexity and time complexity of this method scale exponentially with dimension, and may lose significant details for complex surfaces such as fractals.

- Polygonization, which generates a set of polygons (typically triangles) from the implicit surface. Two common algorithms for such a task are marching cubes [[1]][[2]] and marching tetrahedra. However, these algorithms too lose important detail for complex shapes, and are quite expensive to compute.

2.2.2 Random Sampling and Root Finding

The random sampling method relies on sampling the two SDF at random points, until it finds a point on the intersection of the two surfaces. However, this method may have unreliable performance because in collision-detection contexts, the intersection is often quite small. Because of this, naive random sampling may need to sample an excessive amount of samples to find an intersection.

This can be mitigated by using random sampling until a sufficiently close sample point, p, is found, and using a root finding algorithm initialized at p to find a surface point. However, Finding a sufficiently close sample point also suffers from the same problem of potential excessive sampling. Our method uses an approach derived from this method, but with efficient spacial pruning and an inside-out approach to random sampling.

3 Proposed Method

3.1 Important Property of Finite Monotonically Non-Decreasing Isotropic Shapes

We will not rigorously draw connection between the distance between two shapes A and b defined by SDFs $A(x)$ and $B(x)$ respectively, to regular definitions of distance. As we only use the term distance to draw intuitions. This, in fact, has very little to do with Euclidean distance.

Informally, we will define isotropic shapes as shapes that do not differ by directional information. In other words, a shape defined by a signed distance function $f(x)$, where x is a position in space, is isotropic if and only if $f(x)$ can be written as a function of $||x-c||$, where c is an offset (i.e. centroid of object), which we will call $f_t(j),\ j = ||x-c||$. An isotropic shape is finite if it has a finite volume, and is monotonically non-decreasing if $f_t(j)$ is monotonically non-decreasing. By inspection, you can see that all such functions are radially symmetric. We

Theorem 1 *Assuming that all SDFs are differentiable almost everywhere. The exact distance d between any finite monotonic isotropic shape A defined by the SDF $A(x)$, and any other shape B defined by the SDF $B(x)$, is trivial to calculate, because of the following:*

- *By definition of isotropic shapes, $A(x)$ can be written as a function of $||x-c||$, which we will label $A_t(j)$, which is a scalar function.*
- *The surface of A is the set of points x such that*

  $$A_t(||x-c||) = 0$$

  , which can be described as the set of points such that $||x-c|| = r_A$, where

  $$A_t(r_A) = 0$$

  .
- *Let S_A be the set of points x such that $A(x) = 0$.*
- *The distance between A and B is the value of $\min_{\forall x \in S_A} B(x)$. Because $A_t(j)$ is monotonically non-decreasing, the minimum such value will lie in the direction from the centroid C to the surface of B, which is $-\nabla B(C)$. Therefore, the minimum such value is $A_t(B(C))$.*

We will focus on two isotropic shapes: spheres, as this text will focus on 3-dimensional space, and points, which are degenerate spheres.

4 Core Algorithm

4.1 Collision Manifold

Suppose that we have two shapes, A and B, each defined by their signed distance function, $A(x)$ and $B(x)$ respectively. For our algorithm to work, we will need to define the collision manifold to finite space. For our purposes we will use Axis Aligned Bounding Boxes (AABBs), due to their wide use and efficiency of intersection. As such, we will define $AABB_A$ and $AABB_B$ to be the AABBs of A and B respectively.

For clarity, we will define a finite AABB as an AABB such that all elements of the *min* and *max* points defining the AABB are finite.

We will define $AABB_{int}$ to be the intersection of $AABB_A$ and $AABB_B$. If such intersection does not exist, then there is no collision between the two shapes. And if $AABB_{int}$ is non-finite, we can not continue with the algorithm, although we cannot guarantee that the two shapes are not colliding.

4.2 Sphere Packing

We will define the following parameters for the algorithm:

- K, such that $0 \leq k \leq 1$, is a parameter for the coverage of $AABB_{int}$. Ideally, it defines the exact percentage of $AABB_{int}$ that is to be covered by spheres, however, practical implementations may use it as a general coverage guide,
- ϵ_c, which defines the minimum unsigned distance between a point p to the intersection of A and B such that p is considered a point of collision,
- α, the learning rate, and
- n_{max}, the maximum number of iterations for the root-finding algorithm.

The next step of the algorithm is to pack $AABB_{int}$ with spheres. Note that it is possible to use known, non-overlapping sphere packing algorithms, like Hexagonal Close Packing, even though they can only cover up to a certain percentage of $AABB_{int}$. This is because of the following lemma:

Lemma 1 *Because SDFs are generally either lipschitz continuous or approximately lipschitz continuous, gaps between spheres may be ignored in practical implementations, as long as the spheres are sufficiently dense.*

We will define G as the set of spheres generated by the sphere packing algorithm chosen by the implementation, such that each sphere G_i is defined by it's centroid C_i and radius R_i.

A natural question is why not use the SDF to guide the sphere packing? One way to do this is to sample a random point p to get a distance d, and create a sphere at p with radius d, and to exclude sampling points located in an existing sphere. However, such a method is not concurrency-friendly, which for more complicated SDFs could decrease performance significantly.

4.3 Pruning Spheres

To set up for this section, we will provide the following theorem:

Theorem 2 *For any two SDFs $A(x)$ and $B(x)$, by definition of SDFs, $A(x) \leq 0$ indicates whether a point x lies on the boundary or inside of the shape represented by $A(x)$. Similarly, $B(x) \leq 0$ indicates whether a point x lies on the boundary or inside of the shape represented by $B(x)$.*

The intersection of $A(x)$ and $B(x)$ is the set of points x such that $A(x) \leq 0$ and $B(x) \leq 0$, which is equivalent to $max(A(x), B(x)) \leq 0$, because of the following:

- *If $A(x) \leq 0 \ \wedge \ B(x) \leq 0$, then the larger of $A(x)$ and $B(x)$ must also be less than or equal to 0.*
- *Conversely, if $max(A(x), B(x)) \leq 0$, then both $A(x)$ and $B(x)$ are less than or equal to 0.*

By Theorem 1 and Theorem 2, the exact distance between any finite monotonically non-decreasing isotropic SDF and the intersection of any two SDFs is trivial to calculate.

Thus, we will define $F \subseteq G$ to be the set of spheres in G such that for each sphere $F_i \in F$, $|max(A(C_i), B(C_i)) - R_i| \leq \epsilon_c$. In other words, it is the subset of packed spheres that intersect both shapes, with a given tolerance. If this set is empty, then either there is no collision, or we ran into a false negative.

4.4 Finding a Collision Point

In order to conclude the algorithm and find a collision point x, we will use a root finding algorithm to find a point x such that $A(x) \leq \epsilon_c \ \wedge B(x) \leq \epsilon_c$. However, the function $f(x) = max(A(x), B(x))$ that we used for intersections is non-differentiable at certain points. To prevent this from causing issues, we consider the following function:

$$f(x) = A(x)^2 + B(x)^2$$

Because $0^2 = 0$, this function satisfies the constraint that $f(x) = 0$ when x is a point on the surface of the intersection of the two shapes. And given the gradients of $A(x)$ and $B(x)$, which can be easily approximated if not available, the gradient of $f(x)$ is:

$$\nabla f(x) = 2A(x)\nabla A(x) \ + \ 2B(x)\nabla B(x)$$

Thus, we may use gradient descent to find a point x such that $f(x) \leq \epsilon_c$:

$$x_{n+1} = -x_n \alpha \nabla f(x),\ n > 0,\ n + 1 < n_{max}$$

Terminating the algorithm once $f(x) \leq \epsilon_c$.

5 Implementation and Benchmarks

5.1 Thimni, our Implementation

We have written an n-dimensional implementation of this algorithm in rust [3], with examples and a demo provided [4].

The examples include a simple two-dimensional collision showcase, a simple three dimensional collision showcase, and a three-dimensional collision show case between a sphere, a plane, and a dynamically evolving fractal shape. The demo provided is a simple first person demo set on a large, dynamic fractal, where the player is a capsule shape destroying the fractal.

For performance, Thimni does not use an ideal sphere packing algorithm, but rather uses one that is highly suboptimal when it comes to overlap. Thimni by itself does not provide a linear algebra library, and rather allows the use of any linear algebra library with a vector struct implementing certain functions.

5.2 Benchmarks

All of the following benchmarks are done on a i7-7700HQ CPU, on a linux operating system. They use the glam linear algebra library. These benchmarks generates a number of a shape scattered around a region of space, with an R*-tree to optimize nearest neighbour queries.

shape	number of instances	avg. runtime (ms)
sphere	1000	5
menger sponge fractal	100	105

References

[1] William E. Lorensen and Harvey E. Cline. "Marching cubes: A high resolution 3D surface construction algorithm". In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '87. New York, NY, USA: Association for Computing Machinery, 1987, pp. 163–169. ISBN: 0897912276. DOI: 10.1145/37401.37422. URL: https://doi.org/10.1145/37401.37422.

[2] William E. Lorensen and Harvey E. Cline. "Marching cubes: A high resolution 3D surface construction algorithm". In: *SIGGRAPH Comput. Graph.* 21.4 (Aug. 1987), pp. 163–169. ISSN: 0097-8930. DOI: 10.1145/37402.37422. URL: https://doi.org/10.1145/37402.37422.

[3] Kamal Al-Ameri. *Thimni, A rust crate for SDF collision detection.* Codeberg repository. 2025. URL: `https://codeberg.org/0x177/thimni`.

[4] Kamal Al-Ameri. *A thimni-based collision demo on a destructable, dynamic fractal.* Devlog. 2025. URL: `https://0x177.codeberg.page/coll_demo_pub.html`.