

Intersection Distance Field Collision for GPU

B. Krayer  and R. Görges and S. Müller

University Koblenz-Landau, Germany

Abstract

We present a framework for finding collision points between objects represented by signed distance fields. Particles are used to sample the region where intersections can occur. The distance field representation is used to project the particles onto the surface of the intersection of both objects. From there information, such as collision normals and intersection depth can be extracted. This allows for handling various types of objects in a unified way. Due to the particle approach, the algorithm is well suited to the GPU.

CCS Concepts

• **Computing methodologies** → **Collision detection**; **Mesh geometry models**;

1. Introduction

Collision detection is a widely needed and discussed field of research. Various methods exist for different kinds of objects and scenarios, ranging from simulations to gaming. A natural companion to the question of collision detection is distance. A distance field is a function $f_O : \mathbb{R}^n \rightarrow \mathbb{R}$ that returns the closest distance to a given object O :

$$f_O(\mathbf{x}) = \min_{\mathbf{y} \in O} \|\mathbf{x} - \mathbf{y}\| \quad (1)$$

For a closed object, it is possible to define an inside and outside. Let ∂O be the surface enclosing the object O , then the signed distance field (SDF) s can be defined as

$$s_O(\mathbf{x}) = \begin{cases} -d_{\partial V}(\mathbf{x}), & \text{if } \mathbf{x} \in O \\ d_{\partial V}(\mathbf{x}), & \text{else} \end{cases} \quad (2)$$

In the following, we will leave out the subscript, if it is clear or irrelevant what object the distance function refers to. Due to the definition 1 the gradient is always normalized.

$$\|\nabla s\| = 1 \quad (3)$$

Using this property as the definition of a distance field yields the well-known Eikonal equation. Being able to test for any given point how far away from an object it is and whether it is inside, provides a powerful tool to find collisions between objects.

2. Related Work

Distance fields have a wide variety of representations and applications. Hart [Har96] introduced them in the context of computer graphics with constructive solid geometry operations and the sphere tracing algorithm for efficient rendering.

A common way to store distance fields is a grid of sampled data. While this has advantages, such as ease of use and hardware-accelerated lookup in GPU applications, it requires a large amount of storage to represent high-resolution features.

Frisken et al. [FPRJ00] store the distance field in an octree data structure. Subdivisions of a node are made based on different criteria, such as the reconstruction error at certain positions. Smooth regions can then be stored with less resolution thus saving storage. Bastos and Celes [BC08] extended this data structure to efficiently work on the GPU.

Aside from storing samples, the distance function itself can be approximated with different basis functions. Carr et al. [CBC*01] project an object's SDF onto radial basis functions, which also closes holes in the data. Koschier et al. [KDB16] use a hierarchical structure, but instead fit polynomials per cell.

Distance fields have been used in many collision detection algorithms. Fuhrmann et al. [FSG03] use distance fields to test particles modeling deformable objects for intersections with another body. Xu and Barbič [XB14] used signed distance fields and sampled points from a mesh to calculate a continuous collision by traversing the path a point takes and checking for intersections. This method was also used in a haptic rendering application [XB17]. Oleynikova et al. [OMT*16] use distance fields as a mapping representation in robotics. This can both be used for the mapping itself and collision-free navigation.

A problem with many pure point sampling approaches is that a simple check if a point lies outside does not give enough information to conclude that the object itself does not intersect with the field. Macklin et al. [MEM*20] solve this problem by solving a constraint optimization problem on the primitives of a shape, such as lines or triangles, to find possible intersections.

3. Distance Fields

The following section will give a brief overview of distance fields in general and the version we use in our algorithm.

3.1. Basics

The SDF has many important properties, as shown in [Har96]. Transformations consisting of a translation \mathbf{t} , rotation \mathbf{R} and a uniform scaling a can be exactly represented as

$$s'(\mathbf{x}) = as\left(\frac{1}{a}\mathbf{R}^T(\mathbf{x}-\mathbf{t})\right) \quad (4)$$

Using that form, the movement of objects can be incorporated without recomputing the field. From this, it follows, that the gradient transforms simply by inversely transforming the input and rotating the result:

$$\nabla s'(\mathbf{x}) = \mathbf{R}\nabla s\left(\frac{1}{a}\mathbf{R}^T(\mathbf{x}-\mathbf{t})\right) \quad (5)$$

Since \mathbf{R} is a rotation matrix, the transformed gradient is still normalized, as stated in equation 3. Constructive solid geometry operations can be very efficiently computed. The operations needed for our algorithm are the intersection $A \cap B$ and union $A \cup B$ of two objects A and B .

$$f_{A \cap B}(\mathbf{x}) = \max(f_A(\mathbf{x}), f_B(\mathbf{x})) \quad (6)$$

$$f_{A \cup B}(\mathbf{x}) = \min(f_A(\mathbf{x}), f_B(\mathbf{x})) \quad (7)$$

Since the greatest increase in value occurs orthogonal to the surface, the gradient of the SDF, as with other implicit surfaces, is the normal of the isosurface at the given point. Furthermore, combining the distance and normal information yields a projection operator π that projects a point onto the closest surface. If no unique closest point exists, the gradient and π are not defined.

$$\pi(\mathbf{x}) = \mathbf{x} - f(\mathbf{x})\nabla f(\mathbf{x}) \quad (8)$$

3.2. Higher Order Fields

While multiple data structures exist to represent distance fields, one of the most well-suited ones for GPU computation is uniform sampling, since the GPU provides hardware-accelerated lookup of interpolated values. If an object feature is smaller than the voxel size of the sampling, all values around it will be positive and thus a zero-crossing will be missed in the interpolation, which would prevent collisions. Increasing resolution may be expensive due to the memory requirements. We chose a similar representation to Ban and Gabor [BV20], but use the Dutch Taylor expansion described in [Kra03, Chapter 4] to correctly increase the order of approximation to second degree. The Dutch Taylor expansion for a distance field is given by:

$$f(\mathbf{x}) = \mathbf{x}_0 + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)\nabla f(\mathbf{x}_0) \quad (9)$$

The formula only differs by a factor of $\frac{1}{2}$ from the usual first-order approximation, but when used in a linear interpolation yields a second-order accurate result. As in [BV20] we rearrange the terms to store in a four component texture for 3D points. A voxel element

$\mathbf{v}_{i,j,k}$ with coordinates $\mathbf{x}_{i,j,k}$ is given as:

$$\begin{aligned} \mathbf{v}_{i,j,k} &= \begin{pmatrix} \frac{1}{2}\nabla f(\mathbf{x}_{i,j,k}) \\ f(\mathbf{x}_{i,j,k}) - \frac{1}{2}\nabla \mathbf{x}_{i,j,k} \cdot \mathbf{x}_{i,j,k} \end{pmatrix} \\ &= \begin{pmatrix} \mathbf{a} \\ b \end{pmatrix} \end{aligned} \quad (10)$$

Given an interpolated value \bar{v} for a point \mathbf{x} , we can then compute the approximated field value as

$$f(\mathbf{x}) \approx \mathbf{x} \cdot \bar{\mathbf{a}} + \bar{b} \quad (11)$$

$$= \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix} \cdot \bar{\mathbf{v}} \quad (12)$$

Section 4.3 will describe the generation and storage of the first-order data and the memory requirements compared to just storing the sampled values. Figure 1 shows a comparison of the zeroth-order and first-order fields.

4. Collision detection

This section will describe the collision detection algorithm and its implementation on the GPU. We don't require any specific broad-phase detection and thus any algorithm can be used. The only requirement is that a set of potentially colliding pairs of objects is generated.

4.1. Algorithm

For each pair of objects, we determine a region of possible intersection. We choose the intersection of the axis-aligned bounding boxes (AABB) of the two objects, which itself will be an AABB. A possible intersection shape will then be contained in that box. We then place initial collision particles inside of that box.

In many cases, such as two objects just touching at their sides, the intersection box will be very short in one or two directions. To avoid putting too many samples in directions that do not need it, we employ a scheme to distribute the resolutions per dimension according to the length ratios of the intersection box. We start with an initial resolution N for an n -dimensional cube, giving N^n particles. Let $b_i, i = 1 \dots n$ be the i -th largest box side length and $r_i, i = 1 \dots n$ the corresponding resolutions in those dimensions. We compute the ratios $a_i = \frac{b_i}{b_1}$ for $i = 2 \dots n$. Using the same ratios for the resolutions lets us solve for r_1 in the continuous case.

$$\begin{aligned} \prod_{i=1}^n r_i &= N^n \\ r_1^n \prod_{i=2}^n a_i &= N^n \\ r_1 &= \frac{N}{\sqrt[n]{\prod_{i=2}^n a_i}} \end{aligned} \quad (13)$$

The other r_i for $i > 1$ can then be computed by using the ratios a_i as $r_i = r_1 * a_i$. For very short dimensions, some of the r_i will result in values less than 1. To counter that, we start from the smallest r_j , set it to at least 1 and convert it to an integer by rounding. Afterward, we calculate the resulting number of particles as the product of all r_i . We compute the ratio of the actual and desired number $q = \frac{\prod_{i=1}^n r_i}{N^n}$. All $r_i, i < j$ will then be multiplied with $\frac{1}{j-\sqrt[q]{q}}$, which divides the excess resolution equally between the remaining dimensions. This process is repeated for all $j = n \dots 1$. Algorithm 1 shows this process in 3D.



Figure 1: Comparison of the same model (1a) visualized from a zeroth-order (1b) and a first-order field (1c) with lower resolution. The zeroth-order field uses a resolution, such that its memory usage is similar to the first-order field. Still, the first-order field provides more details and manages to capture some thin object parts better.

Instead of testing, whether one of the collision points is inside both

Algorithm 1: The algorithm computing the sample resolution for a given intersection box in three dimensions.

Input: Sorted (descending) bound extends b_1, b_2, b_3 , base resolution N

Output: Sorted (descending) resolutions

```

1  $a_2 \leftarrow \frac{b_2}{b_1}$ 
2  $a_3 \leftarrow \frac{b_3}{b_1}$ 
3  $r_1 \leftarrow \frac{N}{\sqrt[3]{a_2 a_3}}$ 
4  $r_2 \leftarrow r_1 a_2$ 
5  $r_3 \leftarrow \max(\text{round}(r_1 a_3), 1)$ 
6  $q \leftarrow \sqrt{\frac{r_1 r_2 r_3}{N^3}}$ 
7  $r_1 \leftarrow \frac{r_1}{q}$ 
8  $r_2 \leftarrow \max(\text{round}(\frac{r_2}{q}), 1)$ 
9  $q \leftarrow \frac{r_1 r_2 r_3}{N^3}$ 
10  $r_1 \leftarrow \max(\text{round}(\frac{r_1}{q}), 1)$ 
11 return  $(r_1, r_2, r_3)$ 

```

of the objects, we instead project them onto the intersection surface. The general idea of the algorithm is shown in figure 2. According to equations 6 and 8 we define an update step for each collision point \mathbf{x}_i as:

$$\mathbf{x}'_i = \begin{cases} \pi_A(\mathbf{x}_i) & \text{if } f_A(\mathbf{x}_i) > f_B(\mathbf{x}_i) \\ \pi_B(\mathbf{x}_i) & \text{else} \end{cases} \quad (14)$$

Due to sampling inaccuracies and discontinuities, we repeat the update step multiple times to arrive at a final position \mathbf{x}_i^f . If an intersection was found \mathbf{x}_i^f will be on the surface of at least one of the two objects with $f_{A \cap B}(\mathbf{x}_i^f) < \epsilon$, where ϵ is an iso value to allow for a small offset surface for the collision. \mathbf{x}_i^f is also possibly inside one of the two objects if they are not just touching. The intersection depth d_i is found by considering the distance to the union of both objects given by equation 7

$$d_i = f_{A \cup B}(\mathbf{x}_i^f) \quad (15)$$

The collision normal \mathbf{n}_i from object B to object A is then obtained

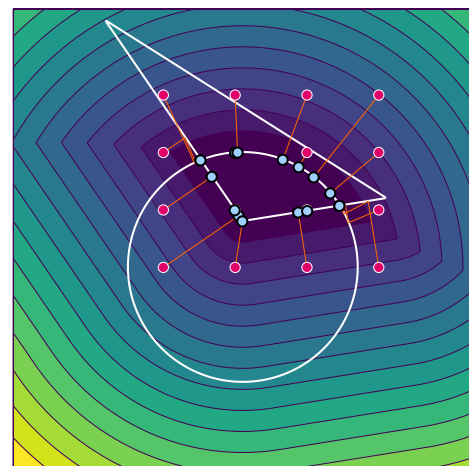


Figure 2: Visualization of the collision detection between a circle and a triangle shown in white. The background lines show the isolines of the distance function of their intersection. The red points with white outlines are the initial sample points, that are projected onto the intersection object.

by the gradient:

$$\mathbf{n}_i = \nabla f_B(\mathbf{x}_i^f) \quad (16)$$

By the properties of SDFs, \mathbf{n}_i is equal to the normal of the closest point on B and pointing to the outside of the object. Algorithm 2 shows the projection algorithm.

4.2. Accuracy

In contrast to other point-based algorithms, we do not only check for basic containment but search out the collision shape. The projection operator is unique for all points except where no unique closest point exists. The set of such points is a null set and thus nearly all points have a defined projection. An intersection can be found if a sample point lies in the Voronoi region of that intersection. Therefore it is possible to miss features when those regions are not sampled.

Algorithm 2: Pseudocode for the iterated projection.

```

1 Function iterateProj:
  Input: Initial point  $\mathbf{p}$ , SDFs  $s_0, s_1$ , intersection epsilon  $\epsilon$ ,
    maximum steps maxSteps
  Output: Projection data
2 for  $i$  to maxSteps do
3    $d_a \leftarrow s_0(\mathbf{p})$ 
4    $d_b \leftarrow s_1(\mathbf{p})$ 
5   if  $d_a \geq d_b$  then
6      $\mathbf{g} \leftarrow \nabla s_0(\mathbf{p})$ 
7   else
8      $\mathbf{g} \leftarrow \nabla s_1(\mathbf{p})$ 
9   if  $\max(d_a, d_b) < \epsilon$  then
10     $\text{result} \leftarrow \{ \mathbf{p}, \nabla s_1(\mathbf{p}), \max(d_a, d_b), \min(d_a, d_b) \}$ 
11    return true, result
12   $\mathbf{p} \leftarrow \mathbf{p} - \max(d_a, d_b)\mathbf{g}$ 
13 return false, null

```

4.3. GPU

As with other particle-based algorithms, the described procedure is independent for each particle and thus can be efficiently implemented on the GPU. An additional advantage is the hardware-accelerated filtering for textures, which improves the lookup speed of distances and gradients of sampled distance fields. We extend the SDF generation algorithm for meshes from [KM19] to store $\mathbf{v}_{i,j,k}$ according to equation 10. We obtain the gradient as follows. If the voxel center lies on the closest triangle, we return the triangle normal. Otherwise the normal is the normalized vector from the voxel center to the closest point, weighted by the sign of the distance value at that point. This ensures that the normal always points to the outside of the mesh. Due to the second-order approximation, we do not use the filtered grid approach found in [BV20]. Other types of objects can be implemented either by sampling them directly from their functional representation or by implementing the function directly in code. As the first-order field needs four values instead of one, storing them at full precision would also increase the memory requirement four times. We instead use a 16 bit floating point type for our textures, thus only requiring twice the memory of a zeroth-order field stored with 32 bit precision at the same resolution. This is roughly equivalent to a first-order field with $\sqrt[3]{2}$ times the zeroth-order field's resolution.

Algorithm 3 shows the basic process. In the first step, the number of actual collisions is counted. To avoid the overhead of having one compute operation spawned per object pair, we handle them all at once. We calculate the appropriate sample resolutions per pair according to algorithm 1. The particle counts for each pair are accumulated into an array. We then spawn parallel compute invocations for the total number of particles. Since the particle count array is monotonically increasing, we can use binary search on it. Each compute invocation searches for the first entry that is greater than its invocation index. The found entry is the collision pair to which the invocation belongs. This can be used to find all the necessary information regarding the two collision objects. Additionally, the

preceding particle count or 0 for the first entry is used to find the number of particles of the pair and the relative index of the invocation. This one-dimensional relative index is transformed into the corresponding n -dimensional index for the pair's sample resolution. That index is then used to find the actual position inside the intersection region. Using the position, the projection described in section 4.1 is computed. If the final position is inside the intersection, we atomically increase a per pair counter to get the total number of actual collisions. Offsets and storage requirements for all pairs are computed with a parallel prefix sum performed on the actual collision counts. Afterward, enough memory is reserved to be able to store the data for all collisions. The process is performed again, but this time the offsets of the previous step are used to store the actual collision information. These can then be processed further, either on the CPU or GPU.

Algorithm 3: Pseudocode for the collision detection algorithm.

```

Input: Array of intersecting pair ids P, Array of accumulated
  sample particle counts C, Resolutions per particle R,
  total number of collisions  $n$ 
Output: Array of collision points
1 counts  $\leftarrow$  ZeroArray( $n$ )
2 parallel for  $i$  to  $n$  do
3    $\text{idx} \leftarrow \text{lowerBound}(C, i)$ 
4    $a_i, b_i \leftarrow P[\text{idx}]$ 
5   bounds  $\leftarrow \text{getIntersectBounds}(\text{idx})$ 
6    $\mathbf{p} \leftarrow \text{getPointInBound}(\text{bounds}, \text{idx})$ 
7    $\mathbf{p}, \text{hit} \leftarrow \text{iterateProj}(\mathbf{p}, s_{a_i}, s_{b_i})$ 
8   if hit then
9      $\text{atomicInc}(\text{counts}[\text{idx}])$ 
10 memoryBarrier()
11 totalCount  $\leftarrow \text{prefixSum}(\text{counts})$ 
12 result  $\leftarrow$  Array(totalCount)
13 parallel for  $i$  to  $n$  do
14    $\text{idx} \leftarrow \text{lowerBound}(C, i)$ 
15    $a_i, b_i \leftarrow P[\text{idx}]$ 
16   bounds  $\leftarrow \text{getIntersectBounds}(\text{idx})$ 
17    $\mathbf{p} \leftarrow \text{getPointInBound}(\text{bounds}, \text{idx})$ 
18    $\mathbf{p}, \text{hit} \leftarrow \text{iterateProj}(\mathbf{p}, s_{a_i}, s_{b_i})$ 
19   if hit then
20      $\text{prevEntry} \leftarrow \text{atomicInc}(\text{counts}[\text{idx}])$ 
21      $\text{result}[\text{prevEntry}] \leftarrow \text{hitData}$ 
22 return result

```

5. Results

We implemented our algorithm in C++ using OpenGL compute shaders for the parallel computations of the intersection finding. Distance fields are accessed via the bindless texture extension. This allows us to directly access textures via an id. A common alternative is using an atlas texture. Times are measured with OpenGL timer queries to accurately get the compute shader execution time. The actual collision handling was delegated to the Bullet Library. For the simulation, the SDF is used to approximate the inertial tensor of

an object. That is accomplished by integrating the formula over all voxels, where a voxel is considered part of the object if its center distance is not positive. Figure 3 shows example scenes computed with our algorithm. The tests were run in Windows 10 with 8GB of RAM, an Intel Core i7-6700HQ CPU, and an NVIDIA GeForce GTX 1070 GPU. We evaluated the algorithm for different scenes. One scene contains a terrain with two models falling on top of each other. The second scene consists of the same terrain, with many objects being dropped on it. The third scene has several balls rolling down a pipe and falling into a bowl. Table 1 shows the results. As all collisions are computed in parallel, we see the average time per colliding object pair going down as the number of those pairs goes up. As the base resolution N corresponds to roughly N^3 sample particles, higher values have a large impact on execution time, though the time per pair still stayed in the sub-millisecond range.

6. Limitations

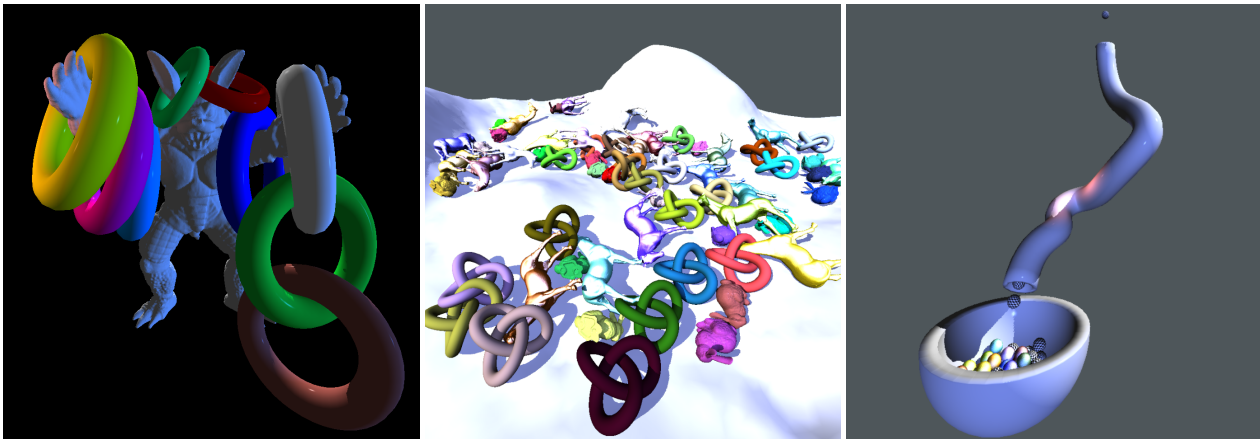
As discussed in section 4.2, intersections can be found, if a point lies in the intersection's Voronoi region. Thus, depending on the number of sample points and the complexity of the model, small intersections might be missed. A related scenario can be seen in figure 4, where a badly spaced sample does not reach the actual intersection, as the projection has to jump between two nearly parallel lines. This problem is similar to the one encountered in sphere tracing. This might be compensated by using an overrelaxation scheme, such as in [KSK*14].

7. Conclusion and future work

We presented an algorithm to find the intersection between two arbitrary objects represented by signed distance fields. Because of that, it is not restricted to triangle meshes and does not need any special handling for different types of objects, such as a distinction between convex and concave. Collisions are detected simultaneously for both objects. This is in contrast to previous techniques. These either used a specific geometric representation, such as triangles, or had to check sampling points for each of the two colliding objects separately. Due to its parallel nature, it can be efficiently implemented on the GPU. Many areas could be explored in the future. Currently, collisions are computed independently in each call, but found collisions could be cached for subsequent checks. Instead of uniformly sampling the intersection region, a random or quasi-random sampling might overcome possible aliasing artifacts. To reduce the number of generated collisions, a basic binary grid structure could be used to only allow for one collision in a voxel region. Additionally, a final step could find the k collisions with the deepest penetration and discard all others per object pair.

References

- [BC08] BASTOS T., CELES W.: GPU-accelerated Adaptively Sampled Distance Fields. In *2008 IEEE International Conference on Shape Modeling and Applications* (2008), IEEE, pp. 171–178. URL: <http://ieeexplore.ieee.org/document/4547967/>, doi:10.1109/SMI.2008.4547967. 1
- [BV20] BÁN R., VALASEK G.: First Order Signed Distance Fields. *Eurographics 2020 - Short Papers* (2020), 4 pages. URL: <https://diglib.eg.org/handle/10.2312/egs20201011>, doi:10.2312/EGS.20201011. 2, 4
- [CBC*01] CARR J. C., BEATSON R. K., CHERRIE J. B., MITCHELL T. J., FRIGHT W. R., MCCALLUM B. C., EVANS T. R.: Reconstruction and representation of 3D objects with radial basis functions. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '01* (2001), ACM Press, pp. 67–76. URL: <http://portal.acm.org/citation.cfm?doid=383259.383266>, doi:10.1145/383259.383266. 1
- [FPRJ00] FRISKEN S. F., PERRY R. N., ROCKWOOD A. P., JONES T. R.: Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (USA, 2000), SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., p. 249–254. URL: <https://doi.org/10.1145/344779.344899>, doi:10.1145/344779.344899. 1
- [FSG03] FUHRMANN A., SOBOTTKA G., GROSS C.: Distance Fields for Rapid Collision Detection in Physically Based Modeling. In *Proceedings of GraphiCon 2003* (2003), p. 8. 1
- [Har96] HART J. C.: Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer* 12, 10 (1996), 527–545. URL: <http://link.springer.com/10.1007/s003710050084>, doi:10.1007/s003710050084. 1, 2
- [KDB16] KOSCHIER D., DEUL C., BENDER J.: Hierarchical adaptive signed distance fields. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Goslar, DEU, 2016), SCA '16, Eurographics Association, p. 189–198. 1
- [KM19] KRAYER B., MÜLLER S.: Generating signed distance fields on the GPU with ray maps. *The Visual Computer* 35, 6-8 (2019), 961–971. URL: <http://link.springer.com/10.1007/s00371-019-01683-w>, doi:10.1007/s00371-019-01683-w. 4
- [Kra03] KRAAIJPOEL D. A.: Seismic ray fields and ray field maps: Theory and algorithms. In *Seismic Ray Fields and Ray Field Maps: Theory and Algorithms* (2003). 2
- [KSK*14] KEINERT B., SCHÄFER H., KORNDÖRFER J., GANSE U., STAMMINGER M.: Enhanced Sphere Tracing. *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference* (2014), 8 pages. URL: <http://diglib.eg.org/handle/10.2312/stag.20141233.001-008>, doi:10.2312/STAG.20141233. 5
- [MEM*20] MACKLIN M., ERLEBEN K., MÜLLER M., CHENTANEZ N., JESCHKE S., CORSE Z.: Local Optimization for Robust Signed Distance Field Collision. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3, 1 (2020), 1–17. URL: <https://dl.acm.org/doi/10.1145/3384538>, doi:10.1145/3384538. 1
- [OMT*16] OLEYNIKOVA H., MILLANE A., TAYLOR Z., GALCERAN E., NIETO J., SIEGWART R.: Signed distance fields: A natural representation for both mapping and planning. In *RSS 2016 Workshop: Geometry and beyond-Representations, Physics, and Scene Understanding for Robotics* (2016), University of Michigan. 1
- [XB14] XU H., BARBIC J.: Continuous Collision Detection Between Points and Signed Distance Fields. *Workshop on Virtual Reality Interaction and Physical Simulation* (2014), 7 pages. URL: <http://diglib.eg.org/handle/10.2312/vrphys.20141218.001-007>, doi:10.2312/VRIPHYS.20141218. 1
- [XB17] XU H., BARBIC J.: 6-DoF Haptic Rendering Using Continuous Collision Detection between Points and Signed Distance Fields. *IEEE Transactions on Haptics* 10, 2 (2017), 151–161. URL: <https://ieeexplore.ieee.org/document/7577891/>, doi:10.1109/TOH.2016.2613872. 1



(a) Armadillo model with multiple tori put on around its hands and ears. (b) Multiple different models with different complexity, with interlocking and concave shapes after a bowl. (c) Multiple balls fall through a winding tube into a bowl being dropped on a simple terrain.

Figure 3: Examples of different scenes.

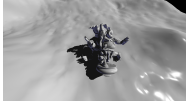
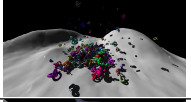
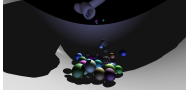
Scene	N	Avg. #collisions	Avg. #pairs	Avg. time [ms]	Avg. time/pair [ms]	Scene image
Armadillo and Buddha statue on terrain	5	4.57	3	0.31	0.1017	
	10	9.38	3	0.31	0.1048	
	15	16.18	3	0.38	0.1271	
	20	37.99	3	0.47	0.1554	
Terrain with many objects	5	14 704.20	1209.40	4.10	0.0034	
	10	118 520.59	1325.405	15.27	0.0115	
	15	354 805.26	1204.096	37.16	0.0309	
	20	808 656.76	1121.437	69.97	0.0624	
Pipe with balls and bowl	5	2015.98	408.13	1.72	0.0042	
	10	65 590.03	505.23	5.01	0.0099	
	15	288 067.48	565.70	17.15	0.0303	
	20	705 999.38	576.88	41.21	0.0714	

Table 1: Timings of the example scenes. All scenarios used a variable time step and were timed using 1000 samples each.

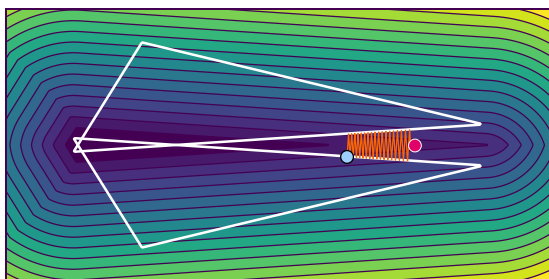


Figure 4: Problematic case for the projection operator. An under-sampled region with two objects being close to parallel to each other. Even with a high iteration count, the projection will not converge.