A Dataset and Explorer for 3D Signed Distance Functions

Towaki Takikawa[1,2] Andrew Glassner[3] Morgan McGuire[2,4]
[1]NVIDIA [2]University of Waterloo [3]Unity / Weta Digital [4]ROBLOX

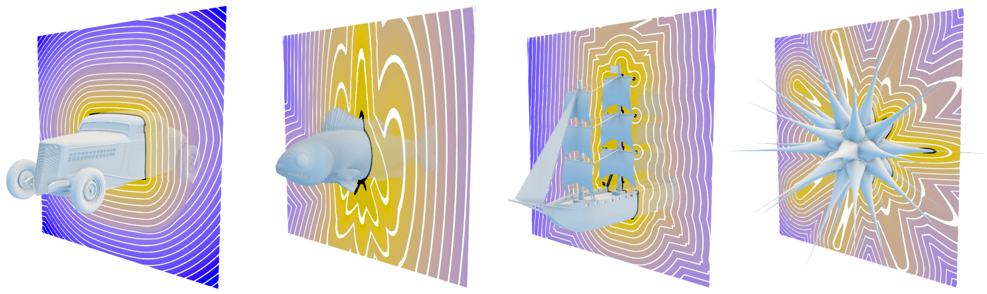


Figure 1. Four analytic signed distance functions (SDFs) from our dataset, whose zero level sets are detailed 3D shapes. These are visualized in our real-time SDF explorer, with isosurfaces in space shown on a cutting plane.

Abstract

Reference datasets are a key tool in the creation of new algorithms. They allow us to compare different existing solutions and identify problems and weaknesses during the development of new algorithms. The signed distance function (SDF) is enjoying a renewed focus of research activity in computer graphics, but until now there has been no standard reference dataset of such functions. We present a database of 63 curated, optimized, and regularized functions of varying complexity. Our functions are provided as analytic expressions that can be efficiently evaluated on a GPU at any point in space. We also present a viewing and inspection tool and software for producing SDF samples appropriate for both traditional graphics and training neural networks.

1. Introduction

The quest for efficient, accurate, flexible, and compact representations of 3D shapes is an enduring research goal in computer graphics. In this article we focus on *signed* *distance fields*, or SDFs, as a representation for surfaces that describe the boundary of volumes. To facilitate the adoption of SDFs in production and to ease the research and development of new algorithms related to this representation, we have developed a rich reference SDF dataset, a visual explorer, and a collection of point evaluators for these surfaces.

The remainder of this introduction gives a general overview of SDFs, compares them to other representations, and provides references to relevant literature. Readers already familiar with SDFs may wish to jump to Section 3 describing the dataset and Section 4 describing the SDF explorer tool and SDF batch-sampling code.

1.1. Explicit Representations

An *explicit representation* of a shape can be used to generate points on its surface. For example, consider the infinite right circular cylinder in Figure 2.

An explicit function $\mathbb{R}^2 \to \mathbb{R}^3$ that maps two real numbers (u, v) to three real numbers (x, y, z) identifying a point on this surface is given in Equation (1),

$$\begin{aligned} x &= \cos u, \\ y &= \sin u, \\ z &= v, \end{aligned} \tag{1}$$

where $u \in [0, 2\pi]$ and $v \in \mathbb{R}$.

The explicit formulation in Equation (1) has a lot of appeal. Evaluating pairs of u and v produces surface points in a predictable way (for example, u parameterizes points along a circular cross-section of the cylinder). The (u, v)-parameterization is helpful for mapping textures of any kind and can be inverted to identify (u, v)-values corresponding to a point on the surface. We could render the cylinder directly from Equation (1) by evaluating many (u, v)-pairs and drawing the resulting point cloud. More traditionally, we could render a mesh created from those points.

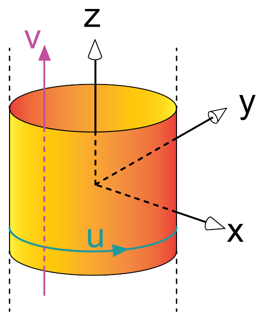


Figure 2. An infinite right circular cylinder along the Z-axis.

Explicit formulas become unwieldy when we want to represent complex and organic shapes, so we usually break up such shapes into collections of simpler shapes. Perhaps the most ubiquitous of such approach today is a polygonal mesh and, in particular, a triangular mesh. Triangular meshes are simple to understand and manipulate, compact to store, and efficient to render. Unfortunately, due to their discrete nature, polygonal meshes can only approximate smooth curves and organic shapes. This approximation can result in undesirable image properties, such as piecewise-linear silhouettes. Curved patches, and meshes of patches, can accommodate complex, curved geometry more closely than a polygonal mesh. Modern graphics systems often support a wide array of explicit representations, such as subdivision surfaces, finite element meshes, and voxels.

Explicit representations are good descriptors for many different kinds of shapes. They offer a clear parameterization of their surfaces, may be stored compactly, and are efficient to render. Explicit meshes are the dominant form of shape representation for most 3D graphics today.

1.2. Implicit Representations

In contrast, *implicit representations* describe surfaces as a set of points which have a desired value in a scalar field defined in space. For example, consider the bumpy height field in Figure 3. The function we are plotting is $\mathbb{R}^2 \rightarrow \mathbb{R}$; that is, each pair of x- and y-input values returns a single z-value.

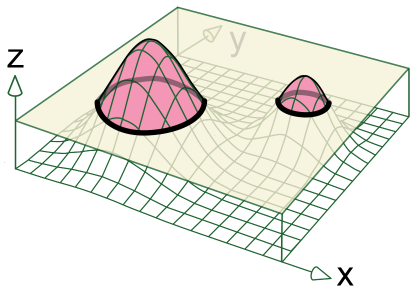


Figure 3. A bumpy landscape on a 2D domain. The points with the z-value of 0 are marked with a thick black line. We call each set of points with the same height an *isocontour*.

Each point on the surface of Figure 3 has a height z, or distance from the (x, y)-reference plane. The collection of points with a single given z-value form a shape, called an *isocontour*. Note that this contour can have multiple pieces. In the figure, we have marked the two isocountour pieces formed by points with the value $z = 0$.

We can identify such points using any reference height, and in any number of dimensions, producing an *isosurface*. In particular, we can evaluate points in 3D space using a function that maps triples of real numbers to a single real number. All points that evaluate to the same value collectively form an isosurface or *level set*, as shown in Figure 1. The set of points with a value of 0, called the kernel of the function, is also referred to as the function's *level-0 isosurface*, or more simply its *zero set* [Osher et al. 2004; Bloomenthal et al. 1997]. Following convention, in this paper and in our dataset, we consider the zero set to be the shape corresponding to each function. Note that the zero set may be thick if many connected points all evaluate to zero.

The cylinder of Figure 2 may be represented implicitly as the set of points that evaluate to 0 in Equation (2).

$$s_{\text{cylinder}}(x, y, z) = x^2 + y^2 - 1. \tag{2}$$

Using zero sets of functions to represent shapes is a highly versatile approach. The functions don't need to be differentiable, invertible, or even continuous. Point clouds are an important example of a discontinuous surface that is easily represented by an implicit function.

Implicit functions can naturally represent organic and curved shapes, are easily composed using constructive solid geometry [Zhou et al. 2016], and have no predetermined precision or level of detail. Implicit functions are widely used in physics simulation [Zhu and Bridson 2005; Adams et al. 2007; Hu et al. 2018], computational fabrication [Dunning et al. 2015; Fernandez et al. 2020], robotics path planning [Adams et al. 2007; Han et al. 2019], and other applications. Implicit functions are also good fits with optimization methods like the interior point method, which is used in computational fabrication [Fernandez et al. 2020].

2. Signed Distance Functions

We now turn our attention to a particular class of implicit functions that make up our dataset and may be viewed with our interactive explorer. Our discussion will be facilitated by some common terms and ideas, so we review these first.

2.1. Sidedness

A useful concept associated with implicit functions is *sidedness*. Every orientable surface partitions the space into three subspaces, which we call *positive*, *zero*, and *negative*, or *outside*, *on*, and *inside*, respectively. An important example is a plane, which we can represent implicitly as

$$s_{\text{plane}}(x, y, z) = ax + by + cz + d. \tag{3}$$

The positive subspace contains those points that evaluate to a positive value in Equation (3). We also say these points are those that are pointed to by the plane's direction vector (a, b, c), as in Figure 4(a). Those points that lie on the plane evaluate to zero, so by convention, those points on the plane are the shape defined by the function.

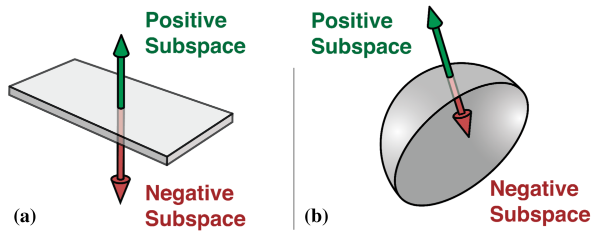


Figure 4. (a) A segment of a plane (thickened for clarity). The surface normal points to the positive subspace (in green). Points in the plane itself form the zero subspace, and the remaining points form the negative subspace (in red). (b) A sphere cut in two. The points outside form the positive subspace (in green), points on the surface form the zero subspace, and points inside form the negative subspace (in red).

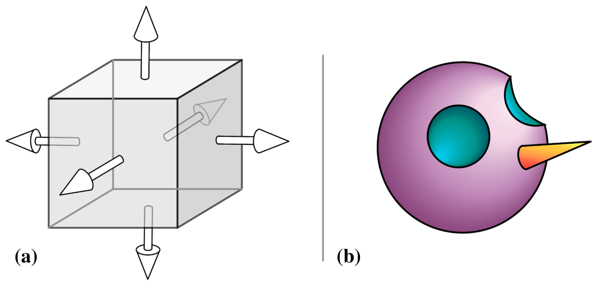


Figure 5. (a) The space on the negative side of all six planes forms the *inside*, or negative subspace, of the composite cube. (b) Using CSG to start a face. The purple sphere has two smaller spheres (blue) removed with subtraction, and a cone (orange) added with union.

The remaining points form the plane's negative subspace. In Figure 4(b) we show the three subspaces for a sphere.

In contrast, meshes are notoriously difficult to work with from a subspace perspective in practice, because meshes need to be manifold and watertight (i.e. no self intersections or gaps) in order to define partitions. Recent work on generalized winding numbers for meshes attempt to more robustly define space partitioning for noisy meshes [Jacobson et al. 2013; Barill et al. 2018]. Signed distance functions, on the other hand, robustly define sidedness by construction.

2.2. Constructive Solid Geometry

Multiple implicit functions can be combined to create more complex surfaces (and the subspaces they induce) using constructive solid geometry (CSG) [Quilez 2008]. For example, we can use six planes to create a cube by orienting the normals outward, as in Figure 5(a). We can treat this collection of planes as a single new implicit function. The set of points that are negative with respect to every plane form the inside of the cube. Such points may be identified by evaluating them with respect to each of the planes and taking the maximum of the results. Those points with a negative maximum are inside the cube, those with a maximum of zero are on the cube itself, and the rest are outside the cube. This process is equivalent to finding the intersection of the six spaces defined by the planes. Similarly, the union can be defined by taking the minimum. Figure 5(b) shows the beginning of a head model that started with a large sphere, using subtraction of smaller spheres to form the eyes, and union with a cone to form the nose.

2.3. Signed Distance Functions (SDFs)

We can make implicit functions more useful by requiring their returned value to carry not just the sign illustrated by Figure 4, but also some kind of *distance* information. That is, the magnitude of the real number returned by the function for any point describes the distance, under some measure, from that point to the surface. The notion of distance used here is flexible and may be adapted to suit different needs.

The most common variety of distance is the shortest path in space from the input point to the nearest point on the zero set. In a Euclidean metric space, this is the Euclidean distance from the point to the nearest point on the surface (which may not be unique).

We call an implicit function that combines an input point's sidedness with its distance to the surface (represented by the magnitude of the returned value) a *signed distance function*, or SDF. In this paper, we focus exclusively on SDFs in their most common environment, a 3D Euclidean metric space.

In symbols, an SDF is a function $s(P) : \mathbb{R}^k \to \mathbb{R}$ that accepts a point P in k dimensions and returns a scalar. As usual, the zero set of $s(P)$ defines the shape we want to represent.

Much more information on SDFs is available online [Quilez 2008; McGuire 2013].

2.4. Classes of SDFs

There are two important classes of SDFs: *exact* and *approximate*. Exact SDFs return the distance to the surface exactly. This means that (except at singularities) they satisfy Equation (4), known as the Eikonal equation.

$$|\nabla f| = 1 \tag{4}$$

This property is also related to the Lipschitz constant, which is the upper bound on the gradient of the function. The Eikonal equation guarantees the Lipschitz constant to be one, which is a useful property for rendering [Hart 1996].

Exact SDFs are more useful in theory and analysis than in practice. One problem with exact SDFs is that most floating-point calculations are inherently inexact on real hardware. Another problem is that procedural combinations of SDFs, such as the CSG shapes in Section 2.2, can introduce additional precision issues.

As a result, many graphics applications are designed to also support SDFs that are not exact. A popular type of inexact SDF returns a magnitude that is only guaranteed to be a lower bound on the distance to the nearest point on the zero set. These are known as *approximate* or *conservative* SDFs. Conservative SDFs do not satisfy the Eikonal equation, but still have a Lipschitz constant of one.

Approximate SDFs where the distances are clipped to some maximum magnitude are known as *truncated* SDFs [Curless and Levoy 1996]. Truncated SDFs have important use cases in robotics, where only near-surface distance values are useful, and real-world measurements introduce uncertainty [Canelhas et al. 2013; Whelan et al. 2015].

Although most SDFs used in practice for graphics applications are conservative, exact SDFs have properties that make them desirable in specific applications where one can account for the numerical issues. One example is that isosurfaces and isocontours of an exact signed distance function taken at regular intervals are equidistant from each other in Euclidean space. Such surfaces are useful for computer numerical control (CNC) tool-path planning applications in computational fabrication [Fernandez et al. 2020].

2.5. Rendering SDFs

Rendering an SDF by *ray marching* is particularly straightforward. Consider a ray $r(t) = P_0 + t\,\mathbf{d}$ that starts at a point P_0 and travels in a normalized direction $\mathbf{d}$ (that is, $\mathbf{d}$ is a vector of length 1), parameterized by a value $t \in \mathbb{R}$ which is 0 at P_0. We can evaluate an SDF $s(P)$ at any point P in space. Applying $s(P)$ to the ray function gives us a new function, $g(t) = s(r(t))$, also parameterized by t. The intersections of the ray with the zero set of the SDF are the points on the ray where the SDF evaluates to zero: $g(t) = 0$. This lets us conveniently consider finding the intersection of a ray and surface as a root-finding problem.

A simple and fast (but flawed) method for finding these roots is called *ray marching*. We step (or march) along the ray by adding a constant amount Δt to the ray's t parameter to produce each new point, where we evaluate the SDF. When the sign of two sequential values is different, we can binary search for the zero between them, as shown by the red ray in Figure 6.

The problem with this method is that the search can fail when the step size moves the query point entirely over a region of changed sign, as shown by the green ray in Figure 6.

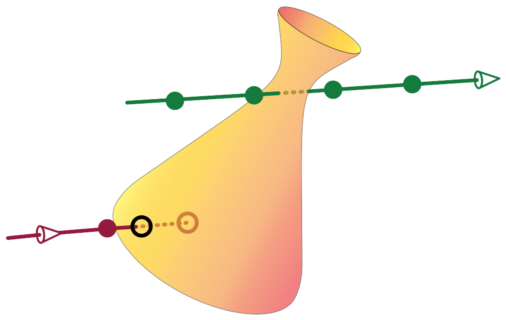


Figure 6. Intersecting a SDF by ray marching. The green ray passes through the shape but the marcher never detects the intersections because the sampling points (dots) created by equal steps of Δt all happen to lie outside the shape. The marcher finds the intersection along the red ray, because one of the sample points (the red circle) lands inside the object, initiating a step of numerical root-finding to locate the intersection point (the black circle).

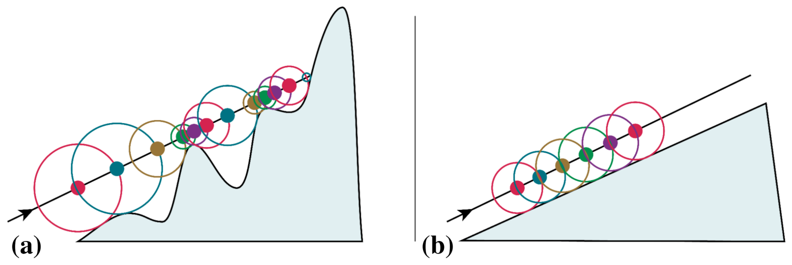


Figure 7. (a) A 2D example of sphere tracing. At each step along the ray (a dot), we find a guaranteed lower bound to the surface (the radius of circle around that dot). We can advance at least that far along the ray for the next step. (b) Here the ray is nearly parallel to a surface, so the step size will be nearly constant. The closer the ray is to the surface, the smaller the step size.

This problem is solved by the superior, but potentially slower, method of *sphere tracing*, a special case of the Newton-Raphson method [Hart 1996]. We characterize the process by numbering the steps we take along the ray. At each step i, we are at point P_i, and seek a Δt_i that tells us how far to move in the next step (the change in t corresponds to a distance because the ray vector $\mathbf{d}$ has length 1). In ray marching, Δt_i is the same for every step i. In sphere tracing, we set $\Delta t_i = |s(P_i)| - \epsilon$ (where ϵ is chosen to compensate for numerical floating-point issues). This means that we're guaranteed that the nearest point on the surface is not within a sphere of radius Δt_i around P_i, so we can step forward along the ray by that distance and be sure we haven't missed the surface. Figure 7(a) shows the idea in two dimensions, using circles rather than spheres to intersect a cross-section of a bumpy landscape.

In this algorithm we don't need to perform the binary search associated with ray marching. Instead, we step forward by each Δt_i until the SDF returns a magnitude smaller than some pre-determined threshold. This method can also handle approximate SDFs, because they provide a lower bound of the distance to the surface.

As it is usually implemented, sphere tracing can deliver poor performance in some situations. For example, in Figure 7(b) a ray is nearly tangent to a surface and diverging from it, but it still takes many steps. We can address this problem, and improve the overall performance of sphere tracing, if we know the values of certain partial derivatives [Hart 1996; Galin et al. 2020]. Unfortunately, these values can be difficult to compute in the general case.

SDFs can be converted into other representations, which can then be rendered using any convenient algorithm. For example, we can discretize the space in which the SDF is embedded into a regular grid, and evaluate the SDF at each vertex (or center) of each cell. By fitting a planar section to each voxel that contains part of the surface, we can construct a surface mesh [Lorensen and Cline 1987]. We can also render the voxels directly as little cubes, drawing each voxel that contains some of the surface.

Applying texture to the surface of SDFs is challenging, because by their nature these level sets do not have a native UV surface parameterization. Such parameterizations, however, can be constructed [de Groot et al. 1981; Schmidt et al. 2006].

The demoscene [Barrallo 2008] shows a parallel evolution of these techniques [Burger et al. 2002; Quilez 2009] to the academic literature, which occasionally overlaps with authors of academic literature also contributing to the demoscene under aliases.

2.6. Deep Learning

SDFs have been found to be a good fit with deep-learning algorithms. Such systems learn by optimizing their internal parameters using an algorithm called backpropagation, which requires each element in the process to be differentiable. Many of the SDFs in our dataset are differentiable, and approximate derivatives can be computed for the others from finite differences. Compared to meshes, such implicit functions are easier for neural networks to learn, because their smooth functional representation can be learned directly without restrictions on topology [Park et al. 2019; Mescheder et al. 2019; Davies et al. 2020].

Learned implicit surfaces have been actively used for a variety of applications, including digital humans [Saito et al. 2019], view synthesis [Mildenhall et al. 2020], shape compression [Davies et al. 2020], level of detail [Takikawa et al. 2021], inverse rendering [Zhang et al. 2021], and robotics perception [Zhu et al. 2021].

3. SDF Reference Dataset

3.1. Motivation

Progress in many fields depends on high-quality reference datasets. Algorithms can be evaluated and compared based on their performance on these datasets, and their strengths and weaknesses can be demonstrated. Standard databases also help during the design and development stages of new research, helping us learn about our algorithms as we develop them. Standard databases help industry practitioners compare the performances of different systems to help them choose the techniques that are best suited to their needs.

In computer graphics, 3D models such as the Utah teapot [Torrence 2006] and Stanford bunny [Bun 1994], and complete scenes such as the Cornell box [Goral et al. 1984], Moana Island [Pritchett and Tamstorf 2018], and many others [McGuire 2017], are frequently used in the literature. In machine learning, references like the Iris [Fisher 1936] and MNIST [LeCun 1998] datasets propelled early work, and large scale datasets such as ImageNet [Deng et al. 2009] and CIFAR-100 [Krizhevsky et al.

2009] were instrumental in the development of modern deep learning. Reference datasets have great value in the evaluation and development of algorithms.

Datasets of 3D meshes have recently become common for use in deep learning for 3D shapes [Chang et al. 2015; Zhou and Jacobson 2016; Koch et al. 2019]. Unfortunately, these datasets share all the familiar problems associated with mesh data: the surfaces can be non-manifold, non-watertight, and have inconsistently oriented normals. These problems can require extensive data cleaning, which can be an impediment to using the models in large-scale data processing applications such as deep learning.

To avoid these problems, we provide a dataset of SDFs, which are inherently free of the issues that plague mesh datasets. We note that it is possible to compute a SDF from a mesh, but the problems we just considered can make that difficult and computationally expensive. A native SDF also can represent continuous curves and efficiently express large amounts of detail (such as fractals).

Despite all of their appealing qualities, SDFs can still be a challenging representation to work with in practice because, until now, they have lacked such a standard set of reference functions. Projects that involve SDFs must begin with a sub-project to collect suitable and well-formed SDFs. Simple functions, such as isolated quadrics and polygons, are not hard to craft by hand, but complex functions meant to capture organic or complex man-made objects can be difficult to create and confirm that they are free of errors. This is not a trivial task, as there are no standard tools available for artists to intuitively build SDFs. The need for reliable reference functions is usually addressed by assembling an ad hoc set of SDFs or by converting alternate representations (e.g., surface meshes) into SDFs.

Unfortunately, trying to derive SDFs directly from existing surface meshes introduces problems. For example, voxelization methods usually introduce quantization errors into the shape they're representing, creating models with inaccuracies in some places, and reduced precision in others. The conversion process can also create distorted and disconnected models, making them slower to evaluate and harder to understand than a simpler, cleaner model created by hand for just that surface.

Worse, if a new algorithm demonstrates problems, one must spend time investigating and confirming the quality of the SDF functions being used, checking for errors such as coding bugs or numerical issues. This is time that would be better spent developing the new algorithm under study. A reliable reference database of SDFs solves these problems.

The main contribution of this work is such a dataset. We present a set of 63 curated, validated, diverse SDFs of varying complexity, designed to provide a wide range of shapes and features. Figure 8 shows the SDFs provided by our system. They fall into the general categories of *animal*, *natural*, *geometry*, *manufactured*, *vehicle*, and *fractal*. The list of shapes is also given in Table 1.

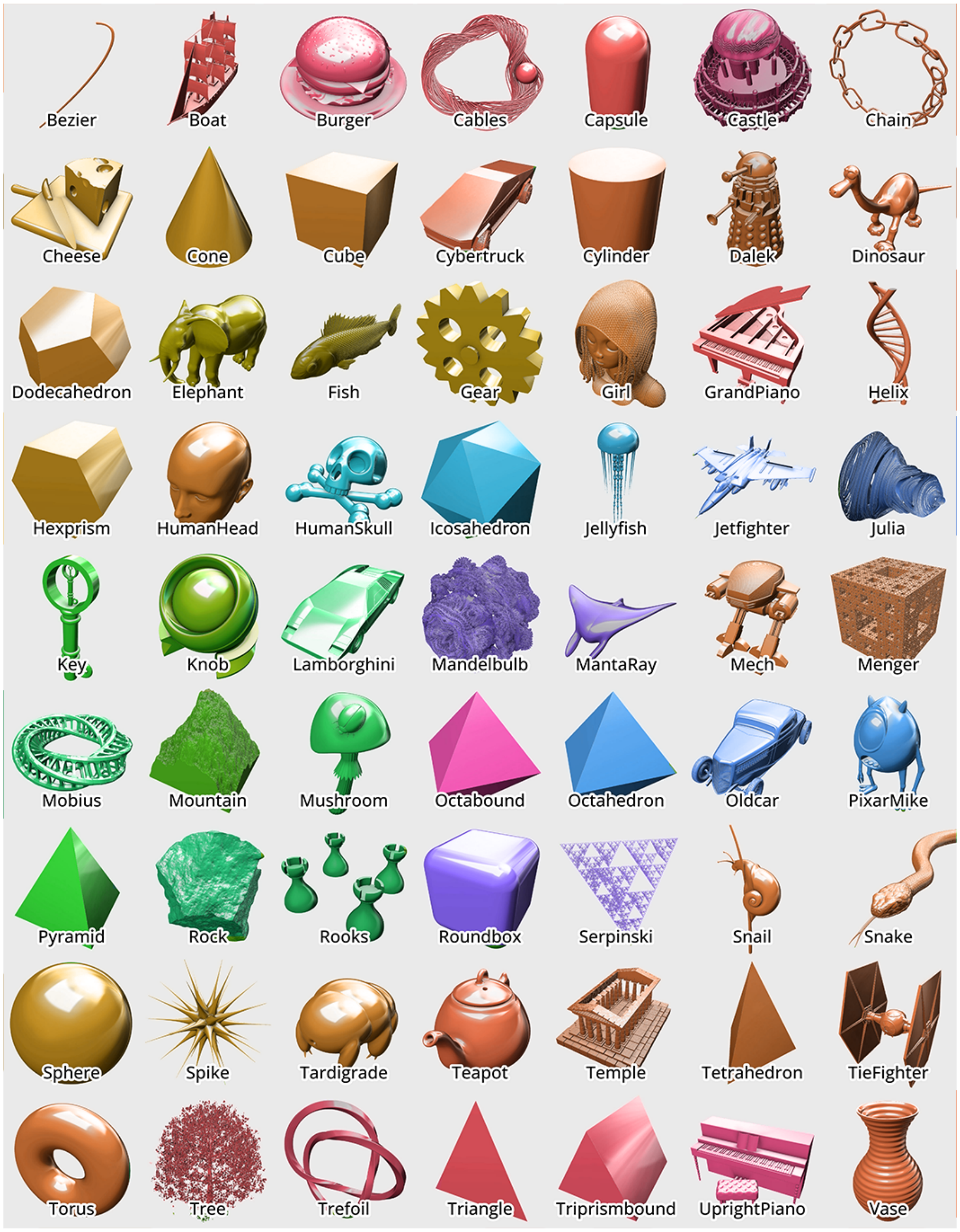


Figure 8. A gallery of our SDF dataset using arbitrary colors.

Name	Category	Type	Animated?	Render (ms)	Author	Copyright	Notes
Bezier	Geometry	E		0.2	Quilez	MIT	
Boat	Vehicle	C		1.8	‘dr2’	CC BY-NC-SA 3.0	
Burger	Misc	C		5.1	‘XorDev’	CC BY-NC-SA 3.0	
Cables	Manufactured	C	✓	6.3	‘yuntaRobo’	CC BY-NC-SA 3.0	
Capsule	Geometry	E		0.3	Quilez	MIT	
Castle	Manufactured	C		13.1	‘sukupaper’	CC BY-NC-SA 3.0	Low numerical precision on the main dome.
Chain	Manufactured	C		0.7	‘eiffie’	CC BY-NC-SA 3.0	
Cheese	Misc	C		0.5	Alfonso	CC BY-NC-SA 3.0	
Cone	Geometry	E		0.3	Quilez	MIT	
Cube	Geometry	E		0.2	Quilez	MIT	
Cybertruck	Vehicle	C		1.2	‘BigWings’	CC BY-NC-SA 3.0	
Cylinder	Geometry	E		0.5	Quilez	MIT	
Dalek	Misc	C		1.8	‘Antonalog’	CC BY-NC-SA 3.0	
Dinosaur	Animal	C		6.3	Quilez	CC BY-NC-SA 3.0	
Dodecahedron	Geometry	E		0.3	Hooper	CC BY-NC-SA 3.0	
Elephant	Animal	C		4.6	Quilez	CC BY-NC-SA 3.0	
Fish	Animal	C	✓	4.5	‘BigWings’	CC BY-NC-SA 3.0	Very conservative.
Gear	Manufactured	C		0.7	Malin	CC BY-NC-SA 3.0	
GrandPiano	Manufactured	C		3.6	‘jedi_cy’	CC BY-NC-SA 3.0	Low numerical precision on keys and strings.
Girl	Animal	C		14.5	Quilez	CC BY-NC-SA 3.0	
Helix	Geometry	E		0.3	‘XorDev’	CC BY-NC-SA 3.0	
Hexprism	Geometry	E		0.2	Quilez	MIT	
HumanHead	Animal	C		10.9	Hooper	CC BY-NC-SA 3.0	
HumanSkull	Animal	C		1.4	‘monsterkodi’	CC BY-NC-SA 3.0	
Icosahedron	Geometry	E		0.2	Hooper	CC BY-NC-SA 3.0	
Jellyfish	Animal	E	✓	0.8	‘BigWings’	CC BY-NC-SA 3.0	
Jetfighter	Vehicle	C		12.5	Berkeby	CC BY-NC-SA 3.0	
Julia	Fractal	E		3.9	Quilez	CC BY-NC-SA 3.0	
Key	Manufactured	C		0.3	‘Flopine’	CC BY-NC-SA 3.0	
Knob	Manufactured	E		0.8	Takikawa	MIT	
Lamborghini	Vehicle	C		2.1	Berger	CC BY-NC-SA 3.0	
Mandelbulb	Fractal	E		10.7	McGuire	BSD-2	
MantaRay	Animal	C	✓	0.7	‘dakrunch’	CC BY-NC-SA 3.0	
Mech	Manufactured	C	✓	5.4	Edis	CC BY-NC-SA 3.0	
Menger	Fractal	E		1.3	Quilez	MIT	
Mobius	Manufactured	E		0.5	Warne	CC BY-NC-SA 3.0	
Mountain	Nature	C		2.0	McGuire	MIT	
Mushroom	Nature	C		1.4	Quilez	CC BY-NC-SA 3.0	
Octabound	Geometry	C		0.1	Quilez	MIT	
Octahedron	Geometry	E		0.2	Quilez	MIT	
Oldcar	Vehicle	C		5.7	Berger	CC BY-NC-SA 3.0	
PixarMike	Animal	C		1.0	Quilez	CC BY-NC-SA 3.0	
Pyramid	Geometry	E		0.2	Quilez	MIT	
Rock	Natural	E		2.4	Alekseev	CC BY-NC-SA 3.0	
Rooks	Manufactured	C		4.0	‘eiffie’	CC BY-NC-SA 3.0	
Roundbox	Geometry	E		0.3	Quilez	MIT	
Serpenski	Fractal	E		5.2	‘al13n’	CC BY-NC-SA 3.0	
Snail	Animal	E		2.3	Quilez	CC BY-NC-SA 3.0	
Snake	Animal	C	✓	3.2	‘BigWings’	CC BY-NC-SA 3.0	Truncated conservative.
Sphere	Geometry	E		0.2	Quilez	MIT	
Spike	Manufactured	C		1.1	‘Patapom’	CC BY-NC-SA 3.0	
Tardigrade	Animal	C		2.4	‘ZGuerrero’	CC BY-NC-SA 3.0	
Teapot	Manufactured	E		0.8	‘klk’	CC BY-NC-SA 3.0	
Temple	Manufactured	C		9.4	Quilez	CC BY-NC-SA 3.0	
Tetrahedron	Geometry	E		0.1	McGuire	MIT	
TieFighter	Vehicle	E		1.3	‘nimitz’	CC BY-NC-SA 3.0	
Torus	Geometry	E		0.2	Quilez	MIT	
Tree	Nature	C	✓	7.4	‘Maurogik’	CC BY-NC-SA 3.0	
Triangle	Geometry	E		4.0	Quilez	MIT	
Trefoil	Geometry	C		0.8	‘dr2’	CC BY-NC-SA 3.0	
Triprismbound	Geometry	C		0.1	Quilez	MIT	
UprightPiano	Manufactured	C		0.7	Quilez	CC BY-NC-SA 3.0	Low numerical precision on keys.
Vase	Manufactured	E		0.3	Bakane	CC BY-NC-SA 3.0	

Table 1. Table of SDFs from our dataset. Under Type, E means exact, and C means conservative. Render is the time to sphere trace that SDF at 1280×720 resolution on an NVIDIA RTX Titan. We credit original authors by surname or `shadertoy.com` username. The GLSL files contain detailed copyright and source information.

Another contribution is a set of shaders, which we call *samplers*, that evaluate large numbers of points generated from a pattern built into the sampler. This makes it easy to generate vast amounts of training and validation data for deep-learning applications.

3.2. Design Principles

We designed our dataset with several key features in mind.

Efficiency. Because most algorithms evaluate SDFs at millions of input points, we have designed our models with efficiency as a top priority. Since each point is computed independently from the others, SDF evaluation is embarrassingly parallel and can therefore take advantage of GPUs. Our SDFs are implemented as standalone pieces of GLSL shader code designed to run on GPUs.

Extensibility. The only requirement for our SDF shaders is that the shader implements a function called `sdf()` that accepts an input of a 3D point, and optionally a time, and returns a float describing the function's signed distance for that point. This design makes it easy to extend the dataset. To add a new SDF, write a GLSL shader that implements the `sdf()` function as above, and then place it in the folder tree of SDF shaders. Because different algorithms require different SDF sampling schemes, our samplers are also extensible in the same fashion by implementing the `sdf_sampler()` function. Listing 1 shows an example of an example shader that implements a spherical SDF. Note that to properly render this SDF, the SDF needs to be conservative.

Usability. To provide a smooth user experience for different work loads, we provide command-line tools to sample from the shader and create rendered images even on headless setups, locally or on the cloud. This means that programmers can treat the dataset as a standard API, and we also make available PyBind11 bindings to directly stream points into a NumPy tensor.

```
// Adding a new SDF is really as simple as adding a single function.
float sdf(vec3 p)
{
    return length(p)-0.5;
}
```

Listing 1. Adding a new SDF.

3.3. Dataset Collection and Preprocessing

Many of our functions were derived from SDFs originally found "in the wild" from various internet sources, where they often contained a variety of undesirable artifacts and special cases. For example, many of our functions came from the ShaderToy archives [Quilez and Jeremias 2017], where SDFs are often designed only for specific camera angles, and the distance functions are not sufficiently conservative from other angles.

We closely examined and often revised or rewrote each candidate SDF. We first extracted the SDF from the context of the full program in which it was found (we credit the source and author at the top of each such shader and also show that information within our visualization tool). We then removed any shading or material computation from the source and wrapped the SDF with our single-function interface.

We transformed the domain of each SDF so that the level 0 isosurface fit within an axis-aligned cube from -1 to 1 along each axis, centered at the origin. We rotated the domain so that the SDFs all had similar orientation with their "fronts" facing along the positive X-axis and "tops" facing along the positive Y-axis.

We then investigated the SDF throughout space using the explorer tool described in the next section. In cases where the SDF was an approximation that was not conservative, we scaled the entire range until it was conservative. We then optimized some of the more complicated SDFs by simplifying logic or removing selected parts that were disproportionately expensive.

Even with all of this cleanup, some of the SDFs in the dataset reveal artifacts in some situations. Table 1 identifies the issues for those functions that have issues. As an example, Figure 9 shows a close-up of a piano's keys. We can see two types of errors. The first is that although the white keys are modeled separately, they actually overlap and appear as a solid block. The second problem is evident in the smaller keys

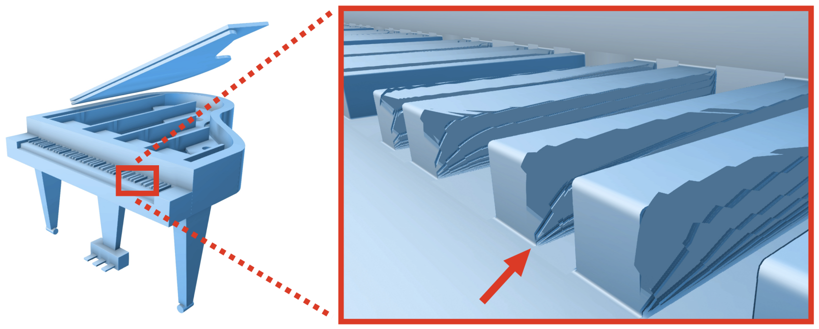
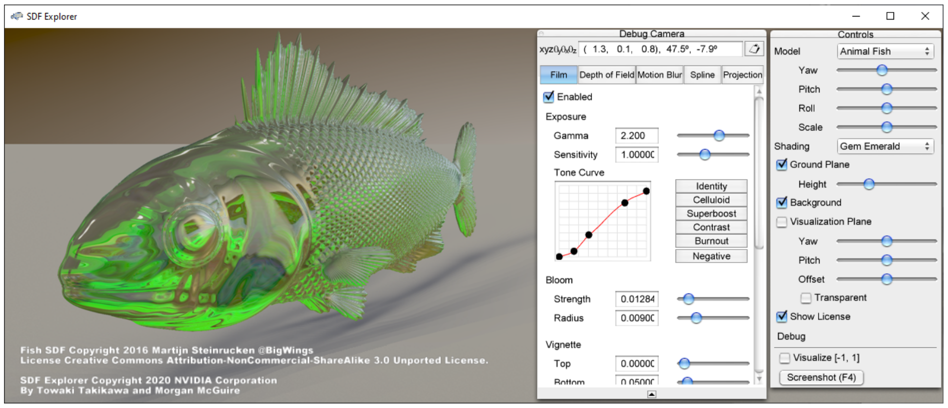


Figure 9. Some problems in an SDF. Left: A grand piano. Right: A close-up of the keys. The smaller keys (black on a real piano) appear jagged on one side due to the accumulation of numerical errors.

(colored black on an actual piano). Each black key is a copy of a single prototype. Though it appears to be a six-sided block, it is actually composed of a large number of simpler primitives. The prototype key shape was repeated with a variety of trig functions to create all the smaller keys. The result is an accumulation of numerical errors that cause a side of some keys to appear jagged and disfigured.

4. The Explorer

We built a visualization and analysis program for exploring the dataset and interactively debugging new SDFs. We provide platform-independent source that we tested on Windows, macOS, and Linux. The user interface for this program is shown in Figure 10. Additional features are exposed through a command line batch-processing interface.



xyzθyθxθz (1.3, 0.1, 0.8), 47.5º, -7.9º

☑ Enabled
Exposure
Gamma 2.200
Sensitivity 1.0000C
Tone Curve
Bloom
Strength 0.01284
Radius 0.0090C
Vignette
Top 0.0000C
Bottom 0.0500C

Model Animal Fish
Yaw
Pitch
Roll
Scale
Shading Gem Emerald
☑ Ground Plane
Height
☑ Background
☐ Visualization Plane
Yaw
Pitch
Offset
☐ Transparent
☑ Show License
Debug
☐ Visualize [-1, 1]

Fish SDF Copyright 2016 Martijn Steinrucken @BigWings
License Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

SDF Explorer Copyright 2020 NVIDIA Corporation
By Towaki Takikawa and Morgan McGuire

Figure 10. The user interface for the explorer tool, here visualizing the Fish SDF using an emerald matcap. Each of the on-screen elements can be configured, including copyright text display and background image.

The next sections describe key components of this visualization program.

4.1. Shading

SDFs are purely mathematical functions with no surface decoration. Smooth, featureless shapes can be hard to interpret by eye. One way to improve their legibility is to add surface detail with *analytic shading*. This requires finding a surface normal, which we compute from the finite differences of six samples of the SDF. We can then apply Gooch-style wrap shading [Gooch et al. 1998] by fading between warm and cool colors based on the direction of the normal with respect to the viewer.

We additionally add view-dependent shading effects through *material capture* (or *matcap*) shading [Sloan et al. 2001]. A matcap is like an environment map, but instead of retrieving a reflection color based on a normal, the diffuse color is directly retrieved from a texture. Matcap shading allows us to quickly switch between different types

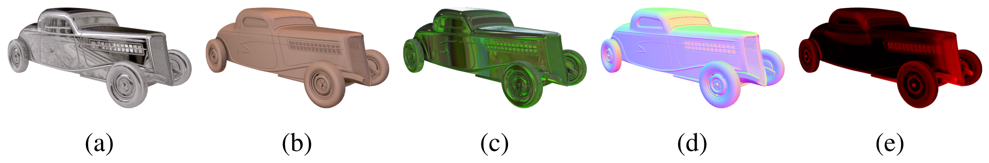


Figure 11. Five of the many shading options supported by the SDF Explorer tool, applied to the OldCar model. (a–c) Three examples of data-driven matcap shading. Thousands of matcap spheres are available online. (d) Surface normal (x, y, z)-values mapped to (r, g, b). (e) Magnitude of the mean curvature visualized as a gradient from flat (black) to maximally curved (red).

of surface materials, simulating even complex surface shaders. Matcap textures are readily available from online repositories,[1] although we do not distribute any with our dataset because many have unknown copyright, licence, and provenance. We additionally compute ambient occlusion using a real-time approximation [Evans 2006]. We also support some special shading modes, such as colorized surface normals.

We can display the mean curvature over the surface, as this quantity is useful for many geometry-processing algorithms such as surface fairing [Desbrun et al. 1999]. For SDFs, the mean curvature amounts to the second-order derivative, which we compute using finite differences. We visualize the absolute value of the mean curvature by colorizing the surface with a gradient. Figure 11 shows examples of matcap shading, surface-normal shading, and mean-curvature shading.

4.2. Interactivity

Our explorer can deliver images of SDFs at interactive rates on most modern computers, which is important for exploring the space and moving the camera and model in real time. For example, at a resolution of 1280×720, we measured 15 frames per second (fps) using a 2017 MacBook's Intel(R) Iris(TM) Plus Graphics 64 GPU under macOS, and over 150 fps for a 2019 Windows 10 desktop with an NVIDIA GeForce 2080 Ti GPU. It provides 200–400 fps on a NVIDIA GeForce RTX 3090 GPU at 1920×1080 resolution, depending on the complexity of the SDF. Table 1 show timings of these SDFs.

4.3. Visualization

3D fields such as SDFs can be challenging to understand, because their volumetric nature is difficult to draw and comprehend. A common tool for addressing this is to show cross-sections and projections of the field on an arbitrary plane that a person can manipulate in real time. We provide this plane, commonly referred to as a cutting plane, as an interactive object.

[1] https://github.com/nidorx/matcaps

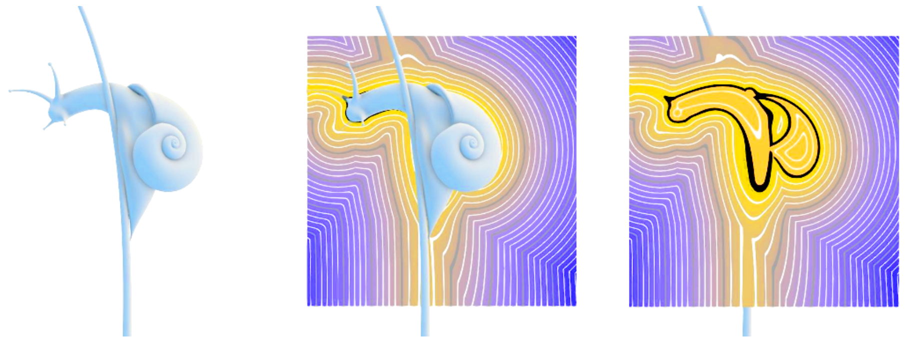


Figure 12. Rendering with a cutting plane. The black line corresponds to the zero level-set, and the lighter lines show the isocurves at equally-spaced thresholds.

Our explorer highlights the level 0 set with a thick black line and provides two color cues for the value of the SDF. A color gradient from yellow to blue runs from zero to increasingly positive values. The interior (negative values) are colored with solid orange. We also draw isocontours for different values of the SDF, spaced 1/64 units apart. Figure 12 shows an example of the cutting-plane visualization.

A significant feature of the cutting plane is that it lets us see the structure of the SDF in space. Rendering an SDF usually means showing its zero set, so we don't see the underlying 3D field. Sometimes problems in the zero set are the result of larger issues in the spatial field. Other times we may want to perform operations on the level sets (like produce offset surfaces by selecting a non-zero set), only to find that the shapes are unexpected or even degenerate.

For example, when rendering the snake object from Figure 8, we found that it consumed far more time per frame than we expected. Visualizing the SDF using the cutting plane in Figure 13, we can see why: the field is a truncated field, which never exceeds a maximum value. We can see this issue on the cutting plane because after the first few contours, the cutting plane is just a solid color, rather than displaying more contours. Thus no matter how far a point is from the snake, the SDF will always report that we're no farther than that maximum.

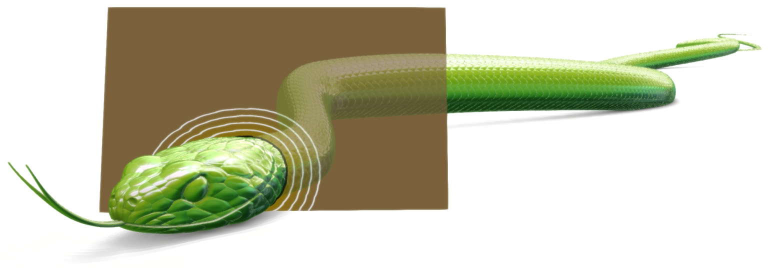


Figure 13. Diagnosing a slow render time. The contour lines on the cutting plane show that this snake was modeled with a truncated SDF, because after a certain distance the contours no longer appear. This explains the slow rendering time for this model.

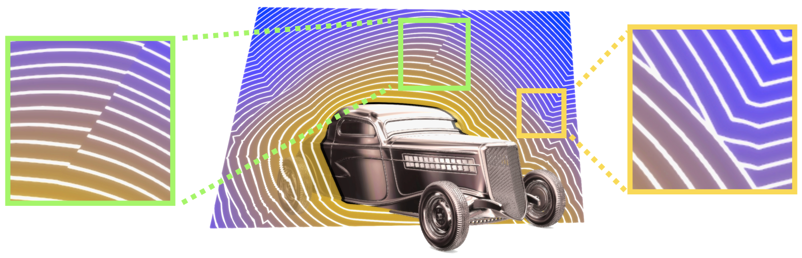


Figure 14. Detecting SDF discontinuities. Middle: A car with its SDF on a cutting plane. Left: A close-up from the center-top of the cutting plane, showing disconnected contour lines. Right: A close-up from the right side of the cutting plane, showing contours lines touching one another.

Some SDFs can be discontinuous, where the field jumps from one value to another. Though the zero set can look fine, when we move outwards from that set (to create an offset surface, for example, or use the SDF to control a simulation), we can get unexpected results. These discontinuities can be revealed by our explorer in two ways, both shown in Figure 14. In the center of Figure 14 we show a car and its SDF on a cutting plane through the middle of the vehicle. At the left of Figure 14 we show a close-up from the center-top of the cutting plane, where discontinuities are shown by disconnected contours. The right image of Figure 14 shows a close-up from the right side of the cutting plane, where discontinuities are revealed by contour lines that touch.

The ability to examine cross sections interactively lets us discover discontinuities, holes, and other imperfections. The SDF and the cutting plane can both be controlled in real time using on-screen sliders.

4.4. Batch Sampling

SDFs are a particularly attractive shape representation for deep learning and are the focus of active research.

We support this work with compute shaders for sampling the SDF at many points following a pattern or distribution. The shaders can be invoked directly from GLSL, or using our tool as a command-line utility invoked from a shell within any framework. In each case, it performs GPU sampling at millions of points, returning large amounts of data suitable for deep learning.

The command-line interface for batch sampling is shown in Listing 2.

```
sdf-explorer --sample <Pattern> <N> <SDF> <outfile>
```

Listing 2. Calling a sampler from the command line.

We look at these arguments in order:

`<Pattern>` is the base name (i.e., no path or filename extension) of a GLSL file in the `sampler` directory. This is extensible by users. The provided patterns are:

grid A regular, axis-aligned 3D lattice within a $2 \times 2 \times 2$ cube about the origin. When the argument `<N>` is not a perfect cube, this takes the first `<N>` points from the rounded-up sequence of $\lceil \sqrt[3]{N} \rceil^3$ samples.

image A regular, axis-aligned 2D lattice within the 2×2 square in the $z = 0$ plane centered at the origin. When the argument `<N>` is not a perfect square, this takes the first `<N>` points from the rounded-up sequence of $\lceil \sqrt{N} \rceil^2$ samples.

jitter Jittered points within the lattice from the grid pattern. These may fall up to half a lattice cell width outside of the $2 \times 2 \times 2$ sampling cube.

metropolis Importance sampled towards the zero level set using the Metropolis-Hastings algorithm.

metropolis_curvature Importance sampled towards areas of high absolute curvature on the zero level set.

near Points displaced slightly along the gradient direction from the surface set. The displacement magnitudes have a Gaussian distribution with mean 0 and standard deviation 0.005, and are clamped to the range $[-0.3, 0.3]$.

rand Points selected uniformly at random within the $2 \times 2 \times 2$ cube about the origin.

surface Points on the surface discovered by tracing random rays.

Figure 15 shows point clouds sampled using several of these samplers.

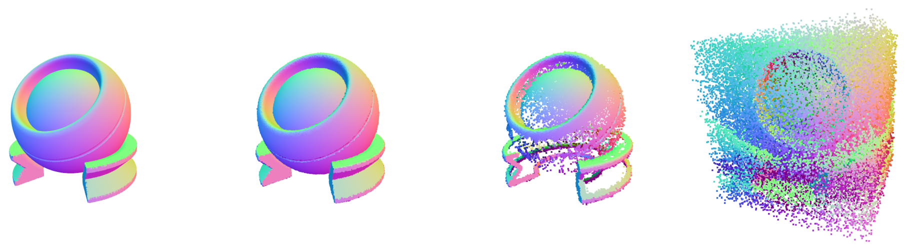

(a) Surface (b) Near Surface (c) Curvature (d) Random

Figure 15. Visualizations of four different patterns from our samplers. Each point cloud contains 1 million samples. The coloring comes from assigning the (x, y, z)-components of the SDF's surface normal at each point to red, green, and blue, respectively.

Value	Count	Format
N	1	uint32
Position x, y, z	N	float32 $\times$ 3
$s(x, y, z)$	N	float32
Gradient $\nabla s(x, y, z)$	N	float32 $\times$ 3

Table 2. Structure of an output file

`<N>` is a positive integer, specifying the number of samples to compute.

`<SDF>` is the SDF to sample. The parameter is the base name of a file in the `sdf` directory hierarchy. This set is extensible. The provided files for our dataset are shown in Figure 8.

`<outfile>` is the output file to write the result to, relative to the current directory. This is a binary file using platform-native endian encoding, following the format in Table 2.

The **surface** and **near** sampling strategies each make 100 attempts per sample to find the surface using a random ray. In order to give a guaranteed running time, they will then fail and return a result that is all `NaN` (floating point not-a-number) for the position, SDF value, and gradient in the file.

Our samplers can be directly used from Python-based deep-learning training pipelines through the provided PyBind11 bindings. Listing 3 shows the code snippet used to load the byte array in C++ directly into a Python tensor.

Listing 4 additionally shows an example of a Python script that uses PyBind11 bindings to execute the sampler, read the binary, and then visualize the sampled point-clouds using Open3D [Zhou et al. 2018].

4.5. Building and Running

Our code has been tested for Windows, macOS, and Linux. The source depends on the G3D Innovation Engine [McGuire et al. 2017]. For efficiency, the program uses sphere tracing for visualization. The program can be run interactively, or as a command line utility for producing sampling datasets and images in batch scripts. This can be useful during deep-learning training or other optimization processes.

We provide a precompiled 64-bit Windows executable of the explorer. To rebuild that or to build it on other platforms, install G3D (SVN revision 7096) from the instructions at `https://casual-effects.com/g3d`. Then, run the provided Visual Studio 2019 solution file on Windows or the compilation script `icompile --opt` from G3D in the root directory of the explorer on Linux or macOS.

For each OS, the executable can be run directly from the command line to output to a file (see Section 4.4 for the arguments). If the program is invoked without any arguments, it opens a window and displays the interactive user interface. To

```cpp
// This C++ function is called from Python via PyBind11,
// and the inputs are tensors to populate.
bool deserialize(py::array_t<float> sTensor,
                 py::array_t<float> dTensor,
                 py::array_t<float> nTensor,
                 std::string filename) {

    // Request buffers and pointers
    py::buffer_info sBuf = sTensor.request();
    py::buffer_info dBuf = dTensor.request();
    py::buffer_info nBuf = nTensor.request();
    auto *pPtr = (float *) pBuf.ptr;
    auto *dPtr = (float *) dBuf.ptr;
    auto *nPtr = (float *) nBuf.ptr;
    auto idx  = [&](int i, int j) -> size_t { return i*3 + j; };

    // Open stream to read input
    fs::path binaryPath(filename);
    std::ifstream f(filename, std::ios::binary);

    // First 4 bytes is the size, then read arrays
    int n;
    f.read((char*)&n, sizeof(int));
    for (uint32_t i = 0; i < n; i++) {
        // Read XYZ position
        float sample[3];
        f.read((char*)&sample[0], 3 * sizeof(float));
        sPtr[idx(i, 0)] = sample[0];
        sPtr[idx(i, 1)] = sample[1];
        sPtr[idx(i, 2)] = sample[2];
    }
    for (uint32_t i = 0; i < n; i++) {
        // Read signed distance
        float dist;
        f.read((char*)&dist, 1 * sizeof(float));
        dPtr[i] = dist;
    }
    for (uint32_t i = 0; i < n; i++) {
        // Read normals
        float sample[3];
        f.read((char*)&sample[0], 3 * sizeof(float));
        nPtr[idx(i, 0)] = sample[0];
        nPtr[idx(i, 1)] = sample[1];
        nPtr[idx(i, 2)] = sample[2];
    }

    f.close();
    return true;
}
```

Listing 3. Reading the Byte Array in C++ into a Python tensor.

```python
import os
import torch
import subprocess as sp
import sdf_deserializer
import open3d as o3d

# Paths
# Set these for your installation
viewer_path = "path/to/sdf-explorer/build"
sample_path = "path/to/samples"
pattern = "near"
N = 1000000
sdf = "Fish"

bin_path = os.path.join(sample_path, f"{pattern}/{sdf}.bin")
cmd = f"sdf-explorer␣--sample␣{pattern}␣{N}␣{sdf}␣{sample_path}"

# Sample points using the G3D SDF explorer,
# by executing the CLI program
proc = sp.check_output(cmd.split(), shell=False, cwd=viewer_path)

# Deserialize directly from bin->torch.Tensor
# Takes as input the position tensor, the distance tensor,
# the normal tensor, and the filepath of the binary format.
pos = torch.zeros((N, 3), dtype=torch.float32)
dist = torch.zeros((N, 1), dtype=torch.float32)
normal = torch.zeros((N, 3), dtype=torch.float32)
sdf_deserializer.deserialize(pos, dist, normal, bin_path)

# Remove invalid points, marked by a NaN value
valid_idx = pos.sum(1).isnan()

pos = pos[valid_idx]
normal = normal[valid_idx]

# Prepare point visualization in Open3D
normal = (normal + 1.0) / 2.0

vis = o3d.visualization.Visualizer()
vis.create_window(height=720, width=1280)
pcd = o3d.geometry.PointCloud()
pcd.points = o3d.utility.Vector3dVector(pos)
pcd.colors = o3d.utility.Vector3dVector(normal)

vis.add_geometry(pcd)
vis.run()
vis.destroy_window()
```

Listing 4. Python script to sample, load, and visualize our dataset.

build the explorer for running on a Linux cluster that does not have a display, set `#define USE_EGL 1` at the top of `App.cpp` to create an EGL graphics context and run with the `--headless` argument.

4.6. Additional Features

Our explorer offers a variety of helpful features for understanding SDF fields and communicating that insight with others. Most of these features are inherited from the underlying G3D Innovation Engine code on which we build.

We can reload all of our shaders, both for shapes and samplers, on demand (by pressing F5), even while the explorer is running. This is convenient for debugging your own SDFs or carrying out extended exploration of the dataset and shaders by modifying them interactively. If your modifications create a shader syntax error, a dialog will report the problem. You can correct the error and press "Reload" on that dialog without restarting the program.

You can save screenshots (press F4) and animations (press F6) of your interactive session. Screenshots will save to the directory from which the executable was launched. By default screenshots are in JPG format and animations are in MP4 format at half resolution. To change these settings and optionally enable capturing the user interface dialogs, press F11 to bring up the advanced developer controls and then select the movie slate icon.

To change advanced camera settings, press F11 to bring up the advanced developer controls and select the camera icon. This allows control over the tone mapping, the camera's field of view and clipping planes, post-process antialiasing, and vignetting. You may also set the camera to an explicit numerical position and orientation from this window.

To move the camera interactively, right click on the 3D part of the view and hold down the right mouse button. Use the mouse to rotate, and use the arrow keys or the W, A, S, and D keys to translate the camera in the style of a first-person video game. The Z key will translate up and the C key will translate down. Hold Shift to move more quickly and Alt to move more slowly. Game controllers are also supported.

All of the features above may be specified via the command-line interface or through the programming API. Thus it is possible to program systems that generate a wide variety of images and animations without direct human control.

We also provide various diagnostic information, such as shader profiling, and a `log.txt` file that reports CPU and GPU use, as well as any program errors if it should crash.

5. Conclusion

We have presented two contributions in this paper. First, we have released a free and open-source dataset of 63 curated, hand-edited, and validated signed distance fields offering a wide variety of shapes and complexity. Our SDFs are encoded in standard GLSL shaders, so they can be conveniently incorporated in existing code. The dataset can be easily extended by writing new shaders that support a single function call.

Our second contribution is an explorer that supports two important tasks. First, our explorer provides real-time interactive visualization of SDFs in 3D, along with a cutting plane that reveals contour lines of the function in space. Both the shape and the cutting plane may be moved and oriented interactively. The system can record images or animations. We provide all of our options through a command-line interface and an API, so users can generate images and animations programmatically.

Second, our explorer offers a convenient process for creating sample data appropriate for use in other tasks, especially training and testing deep-learning systems. A second class of GLSL shaders define different sets of patterns of points in space. The system evaluates the SDF at those points, and returns the results. This process may also be invoked from the command line or via an API.

Our entire dataset is available under the specified open source and Creative Commons 3.0 licenses and may be downloaded from our public git repository and GitHub-hosted website. Our C++ and Python source code in the repository is available under the MIT license.

We encourage others who develop new SDFs and general-purpose shading samplers to submit a pull request in our public repository so we may consider their inclusion in future versions of the dataset.

It is our hope that these reliable and diverse SDFs, along with our explorer which offers visualization and evaluation, will help make it easier to work with SDFs, compare the output of different algorithms, and encourage the creation of new and powerful applications of this flexible form of shape representation.

Acknowledgements

We thank Joey Litalien for help with porting some of the SDFs and PyBind11 support, and Alexander Majercik and Michael Stengel for their technical help with G3D.

References

ADAMS, B., PAULY, M., KEISER, R., AND GUIBAS, L. J. 2007. Adaptively sampled particle fluids. *ACM Trans. Graph. 26*, 3 (July), 48–es. URL: https://dl.acm.org/doi/10.1145/1276377.1276437. 4

BARILL, G., DICKSON, N., SCHMIDT, R., LEVIN, D. I., AND JACOBSON, A. 2018. Fast winding numbers for soups and clouds. *ACM Transactions on Graphics 37*, 4. URL: https://www.dgp.toronto.edu/projects/fast-winding-numbers. 5

BARRALLO, J. 2008. The speed of mathematics. In *Bridges Leeuwarden: Mathematics, Music, Art, Architecture, Culture*. Tarquin Publications, St Albans Hertfordshire, England, 479–480. URL: http://archive.bridgesmathart.org/2008/bridges2008-479.pdf. 9

BLOOMENTHAL, J., BAJAJ, C., BLINN, J., WYVILL, B., CANI, M.-P., ROCKWOOD, A., AND WYVILL, G. 1997. *Introduction to implicit surfaces*. Morgan Kaufmann, San Francisco, CA, USA. 3

1994. Stanford bunny. Stanford University Computer Graphics Laboratory. URL: http://graphics.stanford.edu/data/3Dscanrep/. 9

BURGER, B., PAULOVIC, O., AND HASAN, M., 2002. Realtime visualization methods in the demoscene, March. Department of Computer Graphics and Image Processing, Comenius University. URL: https://old.cescg.org/CESCG-2002/BBurger/index.html. 9

CANELHAS, D. R., STOYANOV, T., AND LILIENTHAL, A. J. 2013. SDF Tracker: A parallel algorithm for on-line pose estimation and scene reconstruction from depth images. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, New York, NY, USA. URL: https://doi.org/10.1109/IROS.2013.6696880. 6

CHANG, A. X., FUNKHOUSER, T., GUIBAS, L., HANRAHAN, P., HUANG, Q., LI, Z., SAVARESE, S., SAVVA, M., SONG, S., SU, H., ET AL. 2015. ShapeNet: An information-rich 3D model repository. *arXiv preprint*. URL: https://arxiv.org/abs/1512.03012. 10

CURLESS, B., AND LEVOY, M. 1996. A volumetric method for building complex models from range images. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, Association for Computing Machinery, New York, NY, USA, SIGGRAPH '96, 303–312. URL: https://dl.acm.org/doi/abs/10.1145/237170.237269. 6

DAVIES, T., NOWROUZEZAHRAI, D., AND JACOBSON, A. 2020. Overfit neural networks as a compact shape representation. *arXiv preprint arXiv:2009.09808*. URL: https://arxiv.org/abs/2009.09808v2. 9

DE GROOT, E., BARTHE, L., AND WYVILL, B. 1981. Implicit decals: Interactive editing of repetitive patterns on surfaces. *Computer Graphics Forum 33*, 1, 141–151. URL: https://hal.archives-ouvertes.fr/hal-00876004. 9

DENG, J., DONG, W., SOCHER, R., LI, L.-J., LI, K., AND FEI-FEI, L. 2009. ImageNet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition*, IEEE, New York, NY, USA, 248–255. URL: https://ieeexplore.ieee.org/abstract/document/5206848. 9

DESBRUN, M., MEYER, M., SCHRÖDER, P., AND BARR, A. H. 1999. Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '99, 317–324. URL: https://dl.acm.org/doi/abs/10.1145/311535.311576. 16

DUNNING, P., BRAMPTON, C., AND KIM, H. 2015. Simultaneous optimisation of structural topology and material grading using level set method. *Materials Science and Technology 31*, 8, 88–894. URL: https://www.tandfonline.com/doi/full/10.1179/1743284715Y.0000000022. 4

EVANS, A. 2006. Fast approximations for global illumination on dynamic scenes. In *ACM SIGGRAPH 2006 Courses*, Association for Computing Machinery, New York, NY, USA, SIGGRAPH '06, 153–171. URL: https://dl.acm.org/doi/abs/10.1145/1185657.1185834. 16

FERNANDEZ, F., BARKER, A. T., KUDO, J., LEWICKI, J. P., SWARTZ, K., TORTORELLI, D. A., WATTS, S., WHITE, D. A., AND WONG, J. 2020. Simultaneous material, shape and topology optimization. *Computer Methods in Applied Mechanics and Engineering 371*, 1. URL: https://doi.org/10.1016/j.cma.2020.113321. 4, 7

FISHER, R. A. 1936. The use of multiple measurements in taxonomic problems. *Annals of Eugenics 7*, 2, 179–188. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1469-1809.1936.tb02137.x. 9

GALIN, E., GUÉRIN, E., PARIS, A., AND PEYTAVIE, A. 2020. Segment tracing using local Lipschitz bounds. *Computer Graphics Forum 39*, 2, 545–554. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13951. 8

GOOCH, A., GOOCH, B., SHIRLEY, P., AND COHEN, E. 1998. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, Association for Computing Machinery, New York, NY, USA, 447–452. URL: https://dl.acm.org/doi/abs/10.1145/280814.280950. 15

GORAL, C. M., TORRANCE, K. E., GREENBERG, D. P., AND BATTAILE, B. 1984. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph. 18*, 3, 213–222. URL: https://dl.acm.org/doi/abs/10.1145/964965.808601. 9

HAN, L., GAO, F., ZHOU, B., AND SHEN, S. 2019. FIESTA: fast incremental euclidean distance fields for online motion planning of aerial robots. *IEEE/RSJ International Conference on Intelligent Robots and Systems abs/1903.02144*. URL: https://arxiv.org/abs/1903.02144. 4

HART, J. C. 1996. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12*, 527–545. URL: https://link.springer.com/article/10.1007/s003710050084. 6, 8

HU, Y., FANG, Y., GE, Z., QU, Z., ZHU, Y., PRADHANA, A., AND JIANG, C. 2018. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Transactions on Graphics 37*, 4. URL: https://dl.acm.org/doi/abs/10.1145/3197517.3201293. 4

JACOBSON, A., KAVAN, L., AND SORKINE-HORNUNG, O. 2013. Robust inside-outside segmentation using generalized winding numbers. *ACM Trans. Graph. 32*, 4 (July). URL: https://dl.acm.org/doi/abs/10.1145/2461912.2461916. 5

KOCH, S., MATVEEV, A., JIANG, Z., WILLIAMS, F., ARTEMOV, A., BURNAEV, E., ALEXA, M., ZORIN, D., AND PANOZZO, D. 2019. ABC: A big CAD model dataset for geometric deep learning. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, New York, NY, USA, 9593–9603. URL: https://ieeexplore.ieee.org/document/8954378. 10

KRIZHEVSKY, A., NAIR, V., AND HINTON, G. 2009. CIFAR-10 and CIFAR-100 datasets. URL: https://www.cs.toronto.edu/~kriz/cifar.html. 10

LECUN, Y. 1998. The MNIST database of handwritten digits. URL: http://yann.lecun.com/exdb/mnist/. 9

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Computer Graphics 21*, 4, 163–169. URL: https://dl.acm.org/doi/abs/10.1145/37402.37422. 8

MCGUIRE, M., MARA, M., AND MAJERCIK, Z. 2017. The G3D innovation engine. URL: https://casual-effects.com/g3d. 20

MCGUIRE, M., 2013. The Graphics Codex. URL: http://graphicscodex.com. 6

MCGUIRE, M., 2017. Computer graphics archive. URL: https://casual-effects.com/data. 9

MESCHEDER, L., OECHSLE, M., NIEMEYER, M., NOWOZIN, S., AND GEIGER, A. 2019. Occupancy networks: Learning 3D reconstruction in function space. In *IEEE Conference on Computer Vision and Pattern Recognition*, IEEE, New York, NY, USA, 4455–4465. URL: https://ieeexplore.ieee.org/document/8953655. 9

MILDENHALL, B., SRINIVASAN, P. P., TANCIK, M., BARRON, J. T., RAMAMOORTHI, R., AND NG, R. 2020. NeRF: Representing scenes as neural radiance fields for view synthesis. In *European Conference on Computer Vision*, Springer, Berlin-Heidelberg, Germany, Lecture Notes in Computer Science, vol.12346, 405–421. URL: https://link.springer.com/chapter/10.1007/978-3-030-58452-8_24. 9

OSHER, S., FEDKIW, R., AND PIECHOR, K. 2004. *Level set methods and dynamic implicit surfaces*. Springer, New York, NY, USA. 3

PARK, J. J., FLORENCE, P., STRAUB, J., NEWCOMBE, R., AND LOVEGROVE, S. 2019. DeepSDF: Learning continuous signed distance functions for shape representation. In *IEEE Conference on Computer Vision and Pattern Recognition*, IEEE, New York, NY, USA, 165–174. URL: https://ieeexplore.ieee.org/document/8954065. 9

PRITCHETT, H., AND TAMSTORF, R. 2018. Moana island scene. Tech. rep., Disney Enterprises, Inc. URL: https://www.disneyanimation.com/resources/moana-island-scene/. 9

QUILEZ, I., AND JEREMIAS, P. 2017. Shadertoy. URL: https://www.shadertoy.com/. 14

QUILEZ, I. 2008. Modeling with distance functions. URL: http://iquilezles.org/www/articles/distfunctions/distfunctions.htm. 5, 6

QUILEZ, I., 2009. The world with two triangles. Talk at Graphical Visionday. URL: http://www.visionday.dk/VD09/graphical/slides/Inigo.pdf. 9

SAITO, S., HUANG, Z., NATSUME, R., MORISHIMA, S., KANAZAWA, A., AND LI, H. 2019. PIFu: pixel-aligned implicit function for high-resolution clothed human digitization. In *IEEE/CVF International Conference on Computer Vision*, IEEE, New York, NY, USA, 2304–2314. URL: https://ieeexplore.ieee.org/document/9010814. 9

SCHMIDT, R., GRIMM, C., AND WYVILL, B. 2006. Interactive decal compositing with discrete exponential maps. In *ACM SIGGRAPH 2006 Papers*, ACM, New York, NY, USA, 605–613. URL: https://dl.acm.org/doi/abs/10.1145/1179352.1141930. 9

SLOAN, P.-P. J., MARTIN, W., GOOCH, A., AND GOOCH, B. 2001. The lit sphere: A model for capturing NPR shading from art. In *Proceedings of Graphics Interface 2001*, Canadian Information Processing Society, Ottawa, Ontario, Canada, 143–150. URL: http://www.cs.utah.edu/npr/papers/LitSphere_HTML.). 15

TAKIKAWA, T., LITALIEN, J., YIN, K., KREIS, K., LOOP, C., NOWROUZEZAHRAI, D., JACOBSON, A., MCGUIRE, M., AND FIDLER, S. 2021. Neural geometric level of detail: Real-time rendering with implicit 3D shapes. *CoRR abs/2101.10994*. URL: https://arxiv.org/abs/2101.10994. 9

TORRENCE, A. 2006. Martin Newell's original teapot. In *ACM SIGGRAPH*, Association for Computing Machinery, New York, NY, USA, 29–es. URL: https://doi.org/10.1145/1180098.1180128. 9

WHELAN, T., KAESS, M., JOHANNSSON, H., FALLON, M., LEONARD, J. J., AND MCDONALD, J. 2015. Real-time large-scale dense RGB-D SLAM with volumetric fusion. *The International Journal of Robotics Research 34*, 4–5, 598–626. URL: https://journals.sagepub.com/doi/abs/10.1177/0278364914551008. 6

ZHANG, K., LUAN, F., WANG, Q., BALA, K., AND SNAVELY, N. 2021. PhySG: inverse rendering with spherical gaussians for physics-based material editing and relighting. *IEEE Conference on Computer Vision and Pattern Recognition*. URL: https://kai-46.github.io/PhySG-website/. 9

ZHOU, Q., AND JACOBSON, A. 2016. Thingi10K: A dataset of 10,000 3D-printing models. *arXiv preprint*. URL: https://arxiv.org/abs/1605.04797. 10

ZHOU, Q., GRINSPUN, E., ZORIN, D., AND JACOBSON, A. 2016. Mesh arrangements for solid geometry. *ACM Trans. Graph.* (July). URL: https://dl.acm.org/doi/abs/10.1145/2897824.2925901. 4

ZHOU, Q.-Y., PARK, J., AND KOLTUN, V. 2018. Open3D: A modern library for 3D data processing. *CoRR abs/1801.09847*. URL: https://arxiv.org/abs/1801.09847. 20

ZHU, Y., AND BRIDSON, R. 2005. Animating sand as a fluid. *ACM Transactions on Graphics 24*, 3, 965–972. URL: https://dl.acm.org/doi/abs/10.1145/1073204.1073298. 4

ZHU, L., MOUSAVIAN, A., XIANG, Y., MAZHAR, H., VAN EENBERGEN, J., DEBNATH, S., AND FOX, D. 2021. RGB-D local implicit function for depth completion of transparent objects. *IEEE Conference on Computer Vision and Pattern Recognition*. URL: `https://ui.adsabs.harvard.edu/abs/2021arXiv210400622Z/abstract`. 9

Author Contact Information

Towaki Takikawa
NVIDIA and
University of Waterloo
200 University Ave W
Waterloo ON, N2L3G1
tovacinni@gmail.com
https://tovacinni.github.io

Andrew Glassner
Unity / Weta Digital
P.O Box 15208
9-11 Manuka Street
Miramar, Wellington
New Zealand 6022
aquamusic@gmail.com
https://www.wetafx.co.nz/
https://glassner.com

Morgan McGuire
ROBLOX
and University of Waterloo
200 University Ave W
Waterloo ON, N2L3G1
morgan@casual-effects.com
https://casual-effects.com

Towaki Takikawa, Andrew Glassner, Morgan McGuire, A 3D Signed Distance Function Dataset, *Journal of Computer Graphics Techniques (JCGT)*, vol. 11, no. 2, 1–29, 2022
https://jcgt.org/published/0011/02/01/

Received:	2020-12-22		
Recommended:	2021-04-06	Corresponding Editor:	Wojciech Jarosz
Published:	2022-04-26	Editor-in-Chief:	Marc Olano