Computers & Graphics 114 (2023) 337–346

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Special Section on CEIG 2023

Adaptive approximation of signed distance fields through piecewise continuous interpolation

Eduard Pujol *, Antonio Chica

Universitat Politècnica de Catalunya, Barcelona, Spain

ARTICLE INFO

Article history:
Received 18 May 2023
Accepted 14 June 2023
Available online 19 June 2023

Dataset link: https://github.com/UPC-ViRVIG/SdfLib.git

Keywords:
Triangle meshes
Distance fields
Ray marching

ABSTRACT

In this paper, we present an adaptive structure to represent a signed distance field through trilinear or tricubic interpolation of values, and derivatives, that allows for fast querying of the field. We also provide a method to decide when to subdivide a node to achieve a provided threshold error. Both the numerical error control, and the values needed to build the interpolants, require the evaluation of the input field. Still, both are designed to minimize the total number of evaluations. C^0 continuity is guaranteed for both the trilinear and tricubic version of the algorithm. Furthermore, we describe how to preserve C^1 continuity between nodes of different levels when using a tricubic interpolant, and provide a proof that this property is maintained. Finally, we illustrate the usage of our approach in several applications, including direct rendering using sphere marching.

1. Introduction

While the most common ways of representing objects only store the model surface to avoid dealing with volumetric data, some applications require these volumetric properties. Computing these properties from the boundary representation can be expensive. Therefore, having a method that efficiently stores these properties is an excellent way to improve the performance of these applications. In this paper, we focus on storing Signed Distance Fields (SDF) for efficient querying, encoding for any point its distance to the object's surface, as well as its inside/outside classification.

Signed distance fields (SDF) are a way of representing shapes that are widely used in computer graphics. They are useful because of their efficiency when performing inside/outside and proximity tests, determining the closest point on a surface, and finding free areas. This has made them popular for many applications [1], such as surface reconstruction [2], rendering [3,4], modeling [5], geometry processing [6,7], collision detection [8–10], and many others. Recently, they have also found use as input and output representations for deep neural networks [11–14].

We present a data structure that can accelerate signed distance field queries. The proposed data structure discretizes space into cubical nodes and stores a polynomial that represents the field behavior inside each node. As each polynomial approximates the input field inside its corresponding node, querying the resulting structure results in approximate values of the SDF. To minimize the error as much as possible and adjust it to the one expected by the user, we propose an adaptive grid structure that uses thinner discretization only in critical parts of the field. We use quadratures to estimate the error of the polynomial approximations, and thus guide the adaptive subdivision of nodes.

Computing both the polynomials, and the error they incur with respect to the input, may require a large amount of evaluations of the input distance field. In order to reduce the total cost, we use trilinear and tricubic interpolants, that allow us to reuse queries of the input field among nodes. Furthermore, these interpolants make it easier to achieve continuity between nodes, even when they are of different sizes.

In summary, the contributions presented in this paper are:

- An adaptive data structure based on an octree that represents the field using polynomial approximations, and aims to provide fast query operations.
- Approximation of the root mean square error (RMSE) between the polynomials and the input field through quadratures to guide the subdivision of nodes.
- Minimization of the input distance field queries during the structure construction by reusing them between nodes.
- Guaranteed C^0 continuity, as well as C^1 when using tricubic interpolants, even between nodes of different levels.

The rest of the paper is structured as follows. Section 2 presents the previous work related to the computation, storage, and

* Corresponding author.
E-mail addresses: eduard.pujol.puig@upc.edu (E. Pujol), achica@cs.upc.edu (A. Chica).

https://doi.org/10.1016/j.cag.2023.06.020

interpolation of distances in SDF. Section 3 introduces the data structure, as well as an outline of the approach. Sections 4 and 5 describe the polynomial interpolants that are used, as well as how the RMSE is approximated. In Section 6, we deal with the problem of achieving C^1 continuity between nodes of different subdivision levels when using tricubic interpolants. Section 7 examines the performance and results of the proposed approach. Finally, in Section 8 we report our conclusions, and we consider some avenues for future work.

2. Previous work

Exact distances There are several ways of computing signed distances, given that they may be produced from boundary representations or be the result of a closed-form expression. For triangle meshes, exact computation is accelerated using hierarchical structures that speed up the nearest primitive search. Maier et al. [15] designed a sphere hierarchy to track the minimum upper bound distance found, thus avoiding distant nodes during computation. The approach in CGAL [16] applies a similar upper bound strategy on a hierarchy of oriented bounding boxes.

Sign computation Still, signed distance computation can be problematic. Bærentzen et al. [17] introduced the angle weighted pseudo-normal as a way to compute the sign of distances to closed manifold triangle meshes correctly. When the triangle mesh has holes, is non-manifold, or has other defects, we need to resort to other methods. Xu and Barbic [18] developed a technique that obtains signed distance fields from non-manifold meshes by exploiting the properties of an offset manifold surface. Their approach may be combined with exact unsigned distance methods, as well as propagation methods. Jacobson et al. [19] go further, requiring only that the mesh has a reasonably consistent orientation. By using a generalization of the winding number to arbitrary triangle meshes, they divide a constrained Delaunay tessellation into inside and outside tetrahedra. The result is a robust segmentation of space. Then, in [20], Barill et al. proposed a tree-based algorithm that approximates this generalization of winding numbers closely, while reducing the asymptotic complexity of the method. They also extended the idea to support point clouds. Krayer and Muller [21] used a fast parallel distance transform to compute distances on the GPU. To deal with holes or other mesh defects, they extracted the sign from winding numbers computed in several directions.

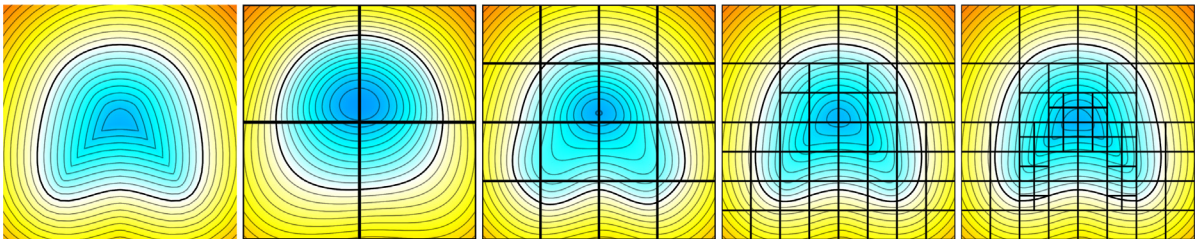
Distance transforms For most applications, being able to query an approximated version of the field is sufficient. These approximations may be built using distance transforms or sampling the exact field. The resulting values may then be interpolated between the samples. Distance transforms compute distances over a grid, either by propagation or via scan conversion. Propagation methods result in approximate values. As distance fields can be expressed as the solution of a special form of the Eikonal equation, they can be computed using methods such as the Fast Marching Method by Sethian [22]. This method orders grid nodes so that information is propagated from the boundary in the direction of increasing distance. An alternative is the Fast Sweeping Method presented by Zhao [23]. The grid points closest to the surface are initialized using the exact distance, then propagated using directional sweeps. The proven convergence of this process guarantees a linear cost. Still, Yatziv et al. [24] proposed a linear version of the Fast Marching algorithm. Cuntz and Kolb [25] presented a GPU-based approach based on propagation, using a hierarchy to reduce the total number of propagation steps. Another option is to use vector propagation that provides error bounds if the distance is propagated to a narrow band around the input surface. Schneider et al. [26] proposed a GPU version.

Scan conversion methods Scan conversion methods produce exact results and are particularly efficient for distance computation close to the surface. The CSC algorithm by Mauch [27] rasterizes the Voronoi regions of every vertex, edge, and triangle of the mesh, updating distance values on the grid only when the distance is smaller than the one already stored. Sigg et al. [28] adapted the CSC algorithm to the GPU, computing the scan conversion of the Voronoi cells for each slice on a fragment program. Then, Sud et al. proposed DiFi [29], an algorithm that computes the distance field on the GPU, slice per slice, determining which primitives really contribute to each of them. This reduces the total workload. In [30] they improved their technique by decomposing the non-linear distance function of the primitives into a dot product of linear factors. This allowed them to exploit texture mapping hardware to compute the linear terms efficiently. These scan conversion methods may have leaks, where the computed sign is incorrect. Erleben and Dohlman [31] enumerated the cases where this may happen and proposed solutions.

Interpolation Once the discrete approximation has been computed, there are several options to interpolate the values. Trilinear and tricubic interpolation [32] are one option, but they cannot represent sharp features. Ju et al. [33] store Hermite data on a grid in order to recover sharp features. Instead, Qu et al. [34] proposed two alternatives, one samples distance fields on an irregular grid, while the other combines multiple distance field representations for the same model. Mitchell [9] proposed to store signed distances into a non-manifold hexahedral mesh, where the explicit connectivity between cells allows for overlapping elements, cracks, and incisions. This is especially useful for collision detection. Another issue is that the gradient of a distance field may have discontinuities at points where the closest primitive changes. This is a problem when using the computed distance field for certain applications. Sanchez et al. [35] proposed an algorithm to apply a convolution filter, such that the field is smoothed, but the initial surface is preserved.

Representations As for many applications precise querying is only required close to the surface, sparse grid representations may be used, thus improving performance and reducing memory cost. Setaluri et al. [36] used such a representation for fluid simulation on high-resolution adaptive grids. Museth presented VDB [37], an efficient hierarchical representation for sparse, time-varying volumetric data, which may be used to compute and store narrowband distance fields efficiently. Its open-source version, OpenVDB [38], has been widely adopted by the film industry.

Alternative ways of representing distance fields also exist. Wu and Kobbelt [39] presented a BSP-based structure with a linear approximation of the field on every cell. Jones [40] used a predictor based on the Vector Distance Transform to develop a lossless compression technique for distance fields. It is also possible to store a first order approximation, i.e. a plane, of the distance field for each node. Ban and Valasek [41] showed that this approximation can be interpolated inside the cells and then evaluated, to improve the representation of the field. Another option is to approximate the field using polynomials instead of interpolating sampled values. Koschier et al. [42] proposed an octree discretization of space, where each cell contains a polynomial approximation of the distance field. They use quadratures to compute the error between the polynomial and the field. This helps them decide when to subdivide a cell, and which degree to use for the polynomial approximation. Still, there is no continuity between cells, as the polynomial inside each of them is optimized separately.



(a) Ground Truth (b) Max Depth 1 (c) Max depth 2 (d) Max Depth 3 (e) Max Depth 4

Fig. 1. Final octree with different maximum depths and the same error. In this case, the input field is the distance field to a simple triangle mesh. The color shows the different values of the field, with blue tones representing negative values (points inside), red and yellow tones representing positive values (points outside), and white tones corresponding to values near zero. Isolines (lines with the same field value) are shown in gray, with one of them in black that corresponds to the zero value and, thus, the triangle mesh. Vertical and horizontal lines mark the boundaries of the octree leaves.

3. Outline

The algorithm's input is a field function and a maximum expected error (from now on *target error*) by the user. From the field function, we only need to be able to query it for any point in space. Optionally, getting for each query the field gradient can help the performance and quality of the final result. Otherwise, we use finite differences to approximate the gradient, which requires more queries and can give less accurate results. With this information, our method computes an octree representing an approximation of the field which is potentially faster to query. We use an octree structure where each tree leaf has a polynomial representing the field behavior inside it.

The octree is built in a top-down manner. The algorithm starts at a start depth decided by the user. For each node, an approximating interpolant is computed as described in Section 4. Then, each node decides if the approximation is enough or if it needs to be subdivided. The node accuracy is estimated by computing a numerical integration of the Mean Square Error of the field inside the node. Fig. 1 illustrates the constructed octree using the same maximum error and different maximum depth. As can be seen in the images, as we increase the maximum depth, the resulting field gets closer to the input field. Notice that as nodes reach the desired error, they stop subdividing. At depth 3 only the bottom part of the model is subdivided because it is more complex than the upper part. And at depth 4 only some interior parts containing the medial axis are subdivided.

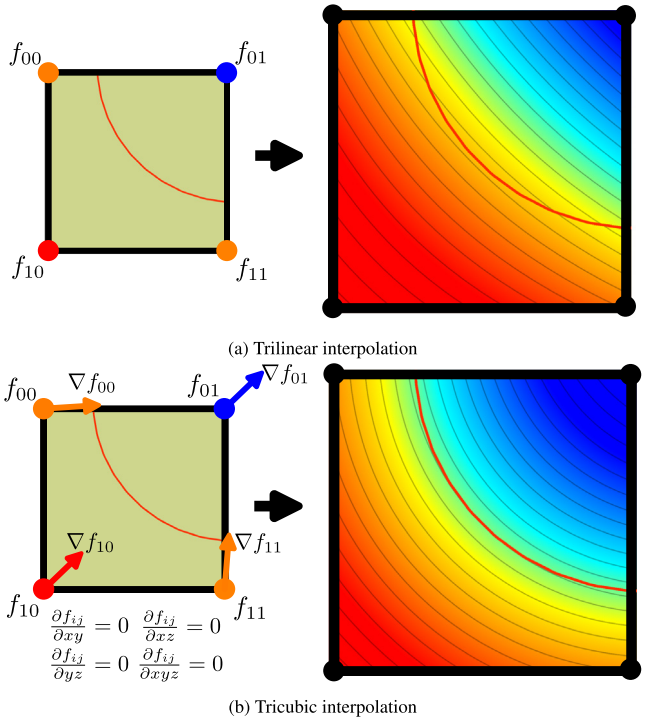
One of the main issues when using adaptative octrees for interpolating values, are the discontinuities between neighbor nodes of different size. In Section 6, we propose a building strategy that forces continuity between nodes by looking at the neighbor nodes during the octree construction to decide if a node needs further subdivision.

Even though we have focused specifically on representing distance fields from meshes, any algorithm that, given a point in space, returns the field's value could be used as input. In our case, the distance to a mesh is equal to the distance to the nearest triangles. We use an SVH (Sphere Volume Hierarchies) [15] to accelerate the nearest triangle search. We also use the pseudonormals described in [17] to compute the field sign using only the nearest triangle. This sign computation only works for closed-orientable two-manifold meshes.

Finally, using this structure, we can query the approximate field faster than evaluating the input field itself. In order to query the field at a point, we only need to traverse the octree top-down until finding the leaf node containing the point. Then, we use its polynomial to compute the field value at that specific point.

4. Polynomials

For each octree leaf, we want to fit a polynomial representing the behavior of the distance field. To avoid storing a different



(a) Trilinear interpolation

(b) Tricubic interpolation

Fig. 2. Example of the two interpolation methods used. The left image shows the constraints used in each method, and the right image the resulting polynomial. The input field is the signed distance field of the surface represented in red.

number of coefficients at each leaf, we use the same degree polynomial for all nodes. We tried the trilinear and the tricubic polynomials. We choose these polynomials because both guarantee C^0 continuity between neighbor nodes, and their formulas and constraints are isotropic (uniform in all axis directions). Both equations are defined by:

$$g(x, y, z) = \sum_{i,j,k=0}^{n} a_{ijk} x^i y^j z^i$$

where n is 1 for trilinear and 3 for tricubic interpolation (see Fig. 2). The polynomial coefficients a_{ijk} dictate the polynomial's behavior. Given a node, we want to find the coefficients that better fit the input field inside the node. We calculate the coefficients by formulating a linear system in which the unknowns are the coefficients. To have a unique solution, we need the same number of equations. Each equation is a constraint that forces the final polynomial to behave in a certain way.

Given a $2D$ node described as a quad with vertices (x_1, y_1), $(x_2, y_2), (x_3, y_3), (x_4, y_4)$ and a function f that given a $2D$ point

returns the value of the input field, the 2D polynomial with $n = 1$ is:

$$g(x, y) = a_{00} + a_{10}x + a_{01}y + a_{11}xy$$

To find the polynomial coefficients, we design a linear system that forces the field's value on each node vertex:

$$\begin{pmatrix} 1 & x_1 & y_1 & x_1y_1 \\ 1 & x_2 & y_2 & x_2y_2 \\ 1 & x_3 & y_3 & x_3y_3 \\ 1 & x_4 & y_4 & x_4y_4 \end{pmatrix} \begin{pmatrix} a_{00} \\ a_{10} \\ a_{01} \\ a_{11} \end{pmatrix} = \begin{pmatrix} f(x_1, y_1) \\ f(x_2, y_2) \\ f(x_3, y_3) \\ f(x_4, y_4) \end{pmatrix}$$

We compute the polynomial coefficients by taking the inverse of this matrix and multiplying it by the vector containing the field values. Because the nodes are square, the columns of the matrix are independent, and the matrix is always invertible. Notice that, as all octree nodes have the same shape, we can calculate the approximating polynomial coefficients without building a new system for each node. To achieve this, we scale and translate the nodes to have the same vertex positions. This avoids having to perform a matrix inversion each time we need to compute an approximating polynomial. The 3D case is analogous, but we have one more axis, so instead of 4 coefficients, we have 8. We force the eight vertices of the node to have the corresponding field value.

If instead we use tricubic polynomials (with $n = 3$), we have $4^3 = 64$ coefficients. To have a system with a unique solution, we need 64 constraints. Therefore, if we want to force constraints only at the eight vertices of the node, we need to specify eight constraints per vertex. For that purpose, we use the method proposed by Lekien and Marsden [32]. On each vertex, they constrain the value field, the gradient (which has three components, one for each axis), the second derivative over xy, yz and xz and the third derivative over xyz. In our case, we set all the second and third derivatives to zero because we use triangle meshes which are planar representations. Setting all these parameters to zero reduces the number of operations required. We do not include the equations of the tricubic interpolation because it is a dense 64×64 matrix, but it may be found in [32].

As explained, we scale and translate the nodes to have the same vertex positions, so it is possible to reuse the same matrix. In our case, when computing the polynomials, all nodes are transformed into a common standard unit cube that has unit volume and its minimum point at the origin. The values of the vertices are the same in a scaled node, but the gradients are not. If the node gets smaller, the gradient should have a bigger magnitude to represent the same value change. Given a cubical node of size L and with its minimum point (x_m, y_m, z_m), we define the mapping h between the default cube and the node as $h(x, y, z) = (x_m + Lx, y_m + Ly, z_m + Lz)$. As a result, we can use coordinates (x, y, z) inside the standard unit cube $[0, 1]^3$, but evaluate the input function f, that uses the original coordinate system, using the expression $f(h(x, y, z))$. This returns the correct distance values. For the gradients, we need to apply the chain rule. For the x-axis:

$$\frac{\partial f(h(x, y, x))}{\partial x} = \frac{\partial h}{\partial x} \cdot \frac{\partial f}{\partial h}(x, y, z) = L \cdot \frac{\partial f}{\partial h}(x, y, z)$$

For the other axes and for the second derivatives, the concept is similar. We need to multiply the gradient by the node size to compute the coefficients. Notice that for the second derivative, we need to multiply it by the square of the node size.

A property of the two interpolation methods is their C^0 continuity at the node boundary. In both cases, the interpolation on a node face reduces to a bilinear and bicubic of the constraints of the vertices forming that face. Also, the interpolation of a node edge is equivalent to a linear and cubic interpolation of the two vertices forming that edge. These two properties ensure C^0 continuity when performing interpolation on a uniform grid, i.e. same sized nodes, because the shared edges and faces are formed by the same vertices, and thus have the same values and gradients used as constraints. The same is true for C^1 continuity for the tricubic interpolant (see [32]). Furthermore, with some care, C^0 and C^1 continuity can also be achieved between nodes of different sizes (see Section 6 and Appendix).

5. Error estimation

An essential part of the method is deciding when a node has to be further subdivided. Over-subdividing a node would increase the structure memory consumption, the building time, and the query time. Likewise, under-subdividing a node would produce an error bigger than the one expected. We calculate the error inside a node using the root-mean-square error (RMSE). We define the RMSE of the polynomial g inside the node c as:

$$RMSE(g) = \sqrt{\int_0^1 \int_0^1 \int_0^1 (g(x, y, z) - f(x, y, z))^2 \, dx \, dy \, dz}$$

Where f is the input field and c is a node with a minimum point at $(0, 0, 0)$ and a maximum point at $(1, 1, 1)$. We use this equation for all the nodes, so we scale and translate the nodes to have the same size and location. Computing this integral algebraically can be computationally expensive or even impossible because f does not have to be an algebraic function. So we estimate it using the trapezoidal quadrature rule, which is a method for approximating definite integrals.

In our case, we at least need to split the interval once, because without subdividing the approximated RMSE will always be zero. As we saw in the previous section, the polynomials used to represent the field inside the leaf nodes are forced to be equal to the field in the node vertices. So, evaluating the error only using those points is not an option. We subdivide the space into eight parts equivalent to the octree children. This way, we can recycle the computed values to compute the children's polynomials without making more queries if the node is finally subdivided.

In a 2D case, the final formula for approximating the RMSE would be:

$$\int_0^1 \int_0^1 E((x, y)) \, dx \, dy \approx \int_0^1 \frac{1}{4}(E((0, y)) + 2E((0.5, y)) + E((1, y))) \, dy =$$

$$= \frac{1}{16}(E((0, 0)) + 2E((0.5, 0))) + E((1, 0)) + 2E((0, 0.5)) + 4E((0.5, 0.5)) + 2E((1, 0.5)) + E((0, 1)) + 2E((0.5, 1)) + E((1, 1))$$

where $E(x, y) = (g(x, y) - f(x, y))^2$. In this case, we need to make 5 more queries to the function f with respect to the values we already needed to compute the polynomial g for the current node. In the 3D case, we need 19 more queries. In both cases, if the node is finally subdivided, we would be able to reuse the queried values to build the interpolants on the children nodes.

6. Forcing continuity

As we saw in the previous section, the interpolation polynomials guarantee that neighbor nodes of the same size are continuous in their shared faces. This property is not automatically fulfilled between nodes of different depths because the nodes have different sizes. In some applications, having discontinuities in the field can be a problem. In this section, we propose a method for forcing this continuity between different-sized nodes during the structure construction.

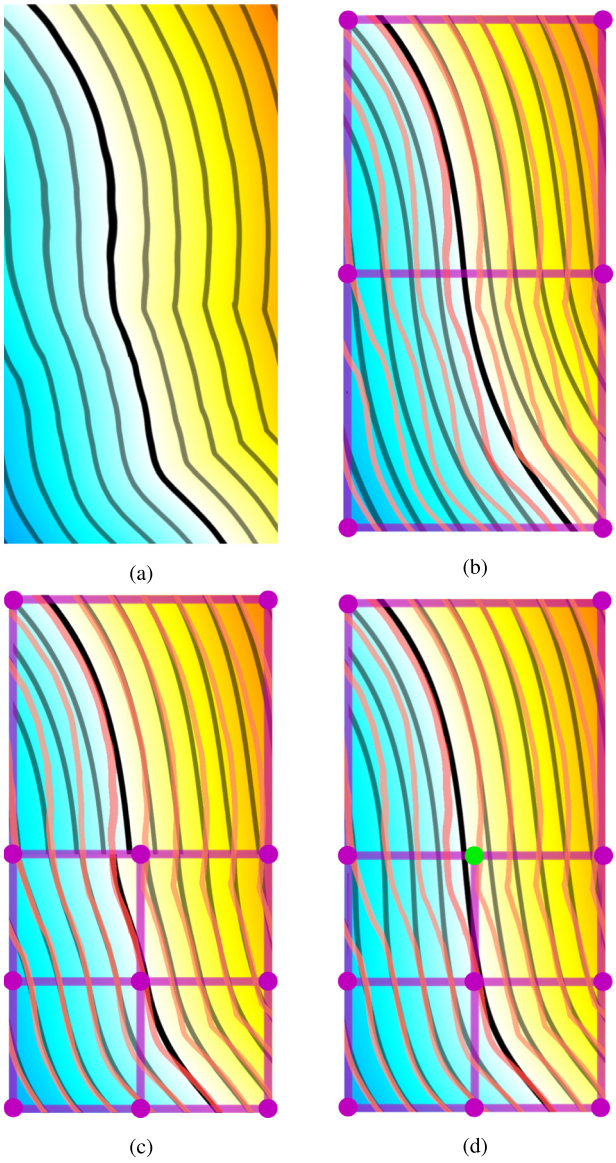


Fig. 3. Example showing how the structure discontinuities are solved for the input field in (a). For (b), (c), and (d), red lines are the isolines of the input field. When we have two nodes of the same size (b), we do not have discontinuities. After subdivision of the bottom node (c), discontinuities appear. By forcing the vertex in green (d) to match the polynomial in the top node, we preserve the continuity.

Two adjacent neighbors can share a face or an edge. The vertices of these shared faces and edges are the same between nodes of the same size. Therefore, the field in these shared parts will be the same because, as we proved earlier, the interpolation only depends on the values of the vertices forming these parts. But, between neighbor nodes of different sizes, not all the vertices are shared, and the field might not be continuous. To force it, we make the unshared vertices to be equal to the neighbor's field. Fig. 3 illustrates how these discontinuities in the field are solved. In this example, we use a large target error, so the final field does not equal the ground truth. Notice that the upper node is not subdivided further even though the input field has a bump that is not present in the final result.

In the base algorithm, the octree is built using a top-down strategy from a starting depth. The algorithm subdivides the octree until no more nodes need to be subdivided, or it reaches a maximum depth. A node is subdivided if its polynomial approximation exceeds the expected error. We want to force the node polynomial of the subdivided nodes to have continuity with its neighbors. To do this, we build the octree using a breadth-first strategy, so that each node has information on its neighbors status before subdividing it. For each depth, we process all the nodes in two steps. First, we decide which nodes should be subdivided according to the error of their polynomial approximation. Then, we iterate through all the nodes again. If a node does not need more subdivision, we write the node in the final octree as a leaf. If it has to be subdivided, we create the node's children, visit the neighbor nodes, and force the non-shared vertices of the children to be equal to the neighbor interpolated field. Notice that this final step is not needed if all the nodes neighbors have decided to perform a subdivision step.

This algorithm guarantees continuity in all the field, but it can produce artifacts in some places. The problem arises when we decide to subdivide a node and not its neighbors. In this situation, some subdivided node vertices are forced to some value to guarantee continuity. Forcing these vertices to be equal to their non-subdivided neighbor nodes is a strong restriction. This strong constraint can result in some artifacts that were not present in the input field.

One way to reduce these artifacts is to balance the octree. An octree is balanced if the depth difference between neighbors is at most one. This strategy solves the problematic artifacts, but it drastically increases the size of the final octree. To reduce memory consumption, we only use the balancing strategy in parts of the octree where the error at the border between neighbors is too large.

The idea is to avoid having these strong constraints in critical parts by further subdividing the leaf neighbor nodes. In the second step of the algorithm, instead of always forcing the non-shared vertices to the neighbor interpolated field, we only force the non-shared vertices that have an error smaller than the threshold. If the error computed at that vertex is larger, we perform an extra subdivision on the neighbor nodes sharing that vertex until they have the same depth to avoid forcing the vertex. When subdividing these leaf nodes, we do not check their node error or force their own neighbor nodes to perform additional subdivisions, because the parent node had already fulfilled the target error condition.

Forcing the continuity between nodes adds a strong dependency on the building algorithm. Still, because the interpolant of each node is evaluated independently of its neighbors, and the same is true for the error computation, it is possible to parallelize the structure construction for each depth level.

7. Results

7.1. Method analysis

All the timings in the paper were obtained on an Intel(R) Core(TM) i7-12700 with 32 GB of RAM and a GeForce RTX 4080 with 16 GB. Query times result from averaging the time needed to solve signed distance field queries at arbitrary points inside the octree bounding box. All the errors reported in this section are expressed with respect to the model bounding box diagonal length. We estimate the RMSE using Monte Carlo integration (MC to shorten the titles) in the experiments. In all the models generated in this section, when not specified, we use tricubic interpolation, forcing the continuity of the field.

Tables 1 and 2 show the behavior of the technique with a bunch of models (see Fig. 4) with different characteristics. The error in all the models is always smaller than the requested one. The resulting errors are not closer to the objective because

Fig. 4. Models used for benchmarking.

Table 1
Results obtained using the trilinear (TL) and tricubic (TC) interpolations with a maximum depth of 9 and a target error of $5e-4$. *Building time* is the time taken to build the octree, *Max error* is the maximum error found during the Monte Carlo RMSE estimation, and *Memory* is the size of the generated octree.

Model	Triangles	Building time		Query time		MC RSME		Max error	
		TL	TC	TL	TC	TL	TC	TL	TC
Armadillo	345944	10.563 s	4.216 s	0.0795 us	0.0954 us	3.30e−04	2.51e−04	0.00190	0.0035
Happy	814216	13.266 s	6.094 s	0.0745 us	0.0836 us	3.47e−04	2.43e−04	0.00235	0.00294
Bunny	70346	5.857 s	1.928 s	0.0709 us	0.0794 us	3.25e−04	2.39e−04	0.00169	0.00347
Screw	420902	8.828 s	4.011 s	0.0713 us	0.0761 us	3.56e−04	2.63e−04	0.00252	0.00421
Plate	1000000	14.476 s	5.823 s	0.064 us	0.0618 us	3.13e−04	1.99e−04	0.00146	0.00182
Temple	151328	5.306 s	3.238 s	0.069 us	0.0675 us	3.25e−04	2.39e−04	0.0722	0.074
Dragon	7218906	44.173 s	29.427 s	0.0797 us	0.0903 us	3.44e−04	2.29e−04	0.056	0.0617
Sponza	66442	8.718 s	6.999 s	0.0644 us	0.0675 us	2.86e−04	2.11e−04	0.05968	0.0615
Slender	754688	15.506 s	6.983 s	0.0646 us	0.077 us	3.43e−04	2.15e−04	0.0036	0.0044

Table 2
Size of the generated octree got using the trilinear (TL) and tricubic (TC) interpolations with a maximum depth of 9 and a target error of $5e-4$.

Model	Memory	
	TL	TC
Armadillo	66.15 MB	103.72 MB
Happy	56.25 MB	106 MB
Bunny	39.88 MB	55.62 MB
Screw	47.2 MB	82.18 MB
Plate	41.58 MB	63.34 MB
Temple	67.37 MB	142.14 MB
Dragon	69.68 MB	121.71 MB
Sponza	82.14 MB	226.05 MB
Slender	73.56 MB	122.81 MB

every subdivision implies a significant reduction. Also, we are not trying to generate an octree with a mean error similar to the one requested, but rather a method that produces an octree in which all the nodes have an error smaller than the target. The maximum error always surpasses the target error defined by the user, but only in critical parts of the model. In most cases, these large errors are generated at the medial axis of the model, where the field makes pointy peaks that are difficult to represent for either of the two interpolation methods. Notice that the building time and the final memory cost not only depend on the number of triangles, the model's shape is also essential. Furthermore, the tricubic interpolation has faster building times than the trilinear, because it needs fewer subdivisions to approximate the field. Still, the trilinear one has a smaller size because each trilinear node needs 8 times fewer coefficients than the tricubic. Even though the tricubic is more expensive in memory, it offers C_1 continuity, while the trilinear only has C_0. This is important for certain applications, and it is apparent when rendering the field using sphere marching.

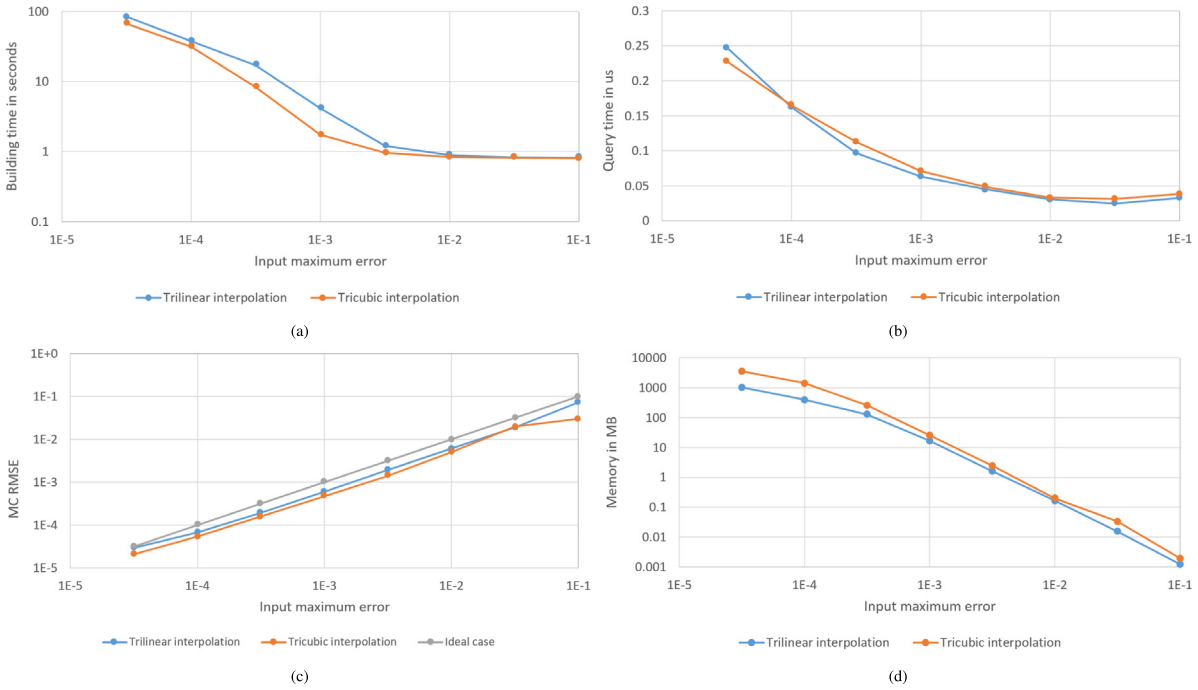


Fig. 5. Analysis of the algorithm with respect to the requested target error using the Armadillo model and a maximum depth of 9. Notice that the target input error (x-axis) in all the plots is in logarithmic scale. In plots (a), (b) and (d), their *y*-axis is also in logarithmic scale. In (c), the line called *Ideal case* represents the perfect case, where the method always gets a structure that represents exactly the field with the desired error.

Table 3
OpenVDB [38] comparison with our tricubic version for the Armadillo model and a resolution of 64 × 64 × 64 for OpenVDB. *Same input error* uses the maximum error of OpenVDB as target error for our method, while in *Same final error* we pursued having the same resulting average error.

Method	Build	Query	RMSE	Max error	Memory
Open VDB	0.456 s	0.084 us	6.1e−4	0.0127	7 MB
Same input error	3.28 s	0.089 us	3.16e−4	0.003	70 MB
Same final error	1.4 s	0.067 us	5.8e−4	0.0052	16.05 MB

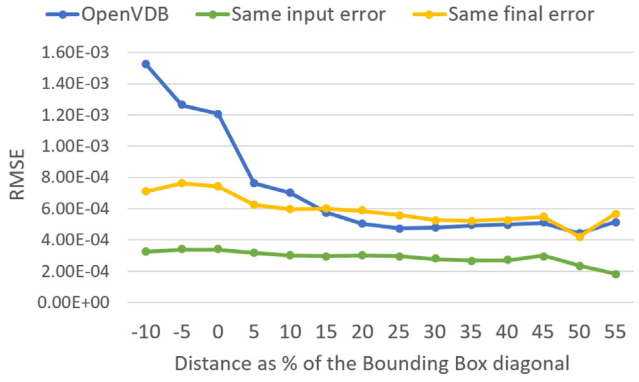


Fig. 6. RMSE error measured at points at different distances from the surface, using the structures defined in Table 3.

Fig. 5 shows the evolution of several properties with respect to the input error. As we can see, even though the trilinear and tricubic give different results, both give similar errors to the target error. The building time is linearly correlated to the target error, except for large target errors. This is caused because the building time also includes other tasks, like processing the mesh, that do not depend on the input error. Also, the octree size has an inverse quadratic behavior with respect to the error.

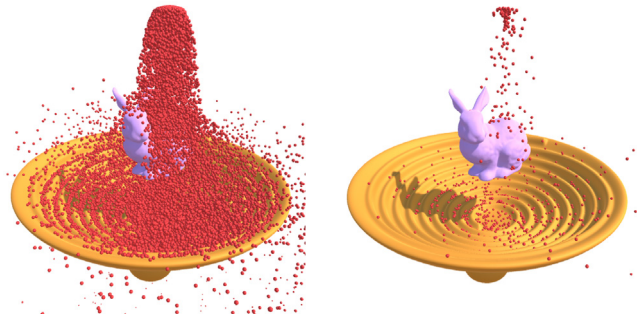
We compare our method with the OpenVDB function for generating signed distance fields. In order to have a dense field, we have forced OpenVDB to subdivide the whole bounding box and store distance values everywhere, thus avoiding its sparse capabilities. Their method first computes distances to the nearest triangles only in nodes that contain them, and then expands them via flooding. Because of this, the final OpenVDB structure has both an approximation error coming from their interpolation strategy, and from the flooding strategy. Fig. 6 contains a plot showing the different RMSE errors at different distances to the surface. As we can see, our adaptive method does a good job of ensuring the same error level in the whole field representation. Conversely, OpenVDB does not ensure a uniform error and has larger errors when close to the surface. Even though our method is more expensive than the OpenVDB in building time and structure size, we can guarantee a uniform error.

7.2. Applications

As we mentioned before, signed distance fields can be used in many applications. We implemented three different applications using our structure: a particle simulation with collision with rigid-bodies, real-time lighting effects like ambient occlusion and soft-shadows, and a ray casting technique to render objects represented with a distance field without requiring any mesh.

Solving collision between a particle and a mesh can be expensive because you need to search for every frame if the particle is colliding with some triangle. Particles are usually represented as spheres with some radius. Therefore, to detect if a particle is colliding with an object, we only need to check if the center of the particle is at a distance smaller than its radius.

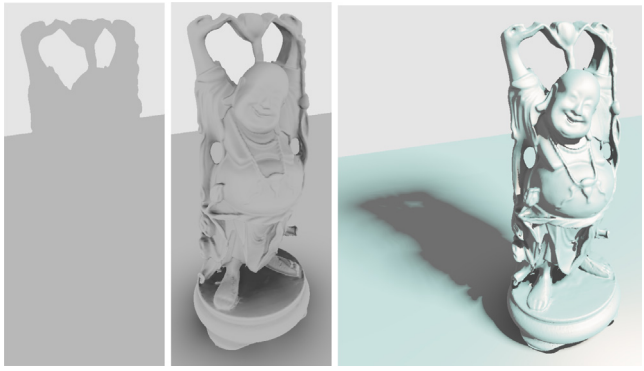
We implement a method for detecting and solving particle collision only using the distance field of the object. When a particle collides with the object, we calculate its response by making a binary search between its current point and its previous position to find the position where the particle is at a distance to



(a) 100000 particles with our method at 5.5ms per frame.

(b) 1000 particles with the SVH at 10.9ms per frame.

Fig. 7. Particle simulation with object collision.

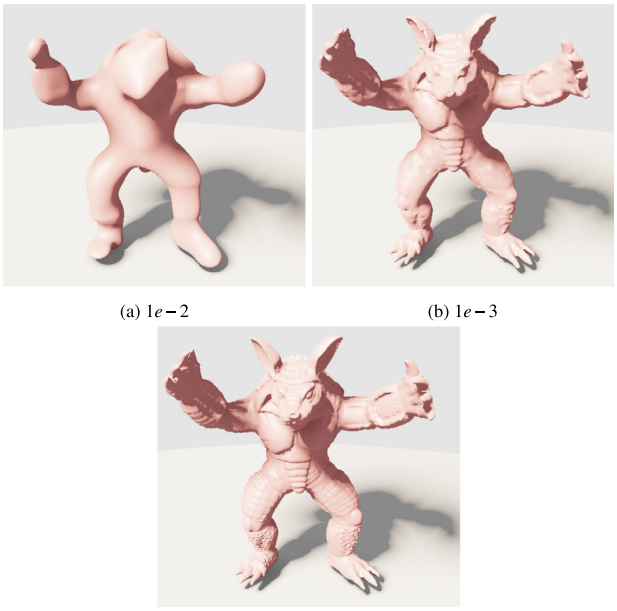


(a) (b) (c)

Fig. 8. Render of the same model (Happy) with different illumination techniques. (a) With a plain color (at 6779 FPS). (b) uses SDF based ambient occlusion on the same mesh (at 895 FPS). Finally, (c) is the combination of using the ambient occlusion technique plus a directional light with soft shadows (at 392 FPS).

Fig. 9. Rendering of the Sponza mesh with real time ambient occlusion and soft shadows added using the distance field. The scene has a positional light behind the center column. The scene was rendered at 344 FPS.

the object equal to its radius. We take that as the position where the particle started to collide with the object. Then, we only need to get the gradient of the field in that position and compute the response forces, the bounce and the friction. We compare our distance field approximation method with the same strategy but using SVH to compute the distance to the object. Using a scene with the relief plate and a bunny (see Fig. 7), we were able to simulate 100000 particles spending 5.5 ms per frame. Using the SVH structure, we were only able to simulate 1000 particles at 10.9 ms per frame. We use the distance field computed in Table 1.



(a) $1e-2$ (b) $1e-3$

(c) $1e-4$

Fig. 10. Renders of the distance field of the Armadillo model with different target errors.

We implement two techniques to improve the rendering of triangle meshes. First, we implement a method proposed by [4], to estimate the ambient occlusion using our SDF approximation. We also simulate soft shadows [43] by estimating the amount of light coming from the light source. Figs. 8 and 9 are images created using the ambient occlusion and soft shadow techniques accelerated with our algorithm. As we can see, even though the Sponza scene is more difficult to traverse for a sphere marching based technique than the Happy model, both of them achieve similar frame rates.

Also, we can use sphere marching to directly render the models instead of rasterizing them. In Fig. 10, we can see three renders of the same model using the sphere marching technique and different target errors. Notice that between (b) and (c) the difference is in the details of the surface. So, for computing shadows and ambient occlusion, the field of (b) could be enough.

8. Conclusions

We have presented an octree-based data structure that adaptively represents a signed distance field, using a polynomial for each node. This allows for very efficient queries. Choosing these polynomials to be trilinear or tricubic interpolants also makes it possible to guarantee continuity between different nodes. We provide a proof that this is the case. The subdivision of a node on the octree is guided by the root mean square error between the current interpolant assigned to it and the input field. This error cannot generally be computed exactly. Instead, we use an approximation using a simple quadrature. Both the determination of the interpolants and the computation of the error need to evaluate the input field. Nevertheless, because of the way the interpolants are computed, we reuse many of the evaluations, reducing the total cost. An implementation of the algorithm and the visualization tools used to generate the images are available at https://github.com/UPC-ViRVIG/SdfLib.

We have focused this work on representing distance fields, but it could also be useful for any scalar or vector field. However, depending on the intended use, the fidelity of different properties could be relevant. For example, having a smooth gradient

is important for blending applications [35]. For others, sharp features is a requirement, and representing them precisely using our approach might require too much subdivision.

Furthermore, with the current implementation, all nodes use either the trilinear interpolant or the tricubic. The tricubic needs eight times more coefficients, and in some places, especially far away from the surface, the trilinear might be enough. This, combined with the fact that it could be useful to require an error that depends on the distance to the surface, suggests the possibility of mixing different degree interpolants. We would need to guarantee that continuity is achieved between nodes of different depth and degree.

Right now, we only save a representation of the field in the node leaves. By storing this information in inner nodes, we could have different resolutions of the same field. This could be useful in some applications that require different precisions in the queries, for example, in ray marching. Samples that do not require much detail could stop before reaching the leaf node.

Finally, even though the structure is adaptive, it still consumes a considerable amount of memory. For very large scenes or very detailed models, this could be an important limitation. Compressing this representation should be helpful in this regard.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Algorithm implementation and visualization tools available at: https://github.com/UPC-ViRVIG/SdfLib.git.

Acknowledgments

This work has been partially funded by Ministeri de Ciència i Innovació (MICIN), Agencia Estatal de Investigación (AEI) and the Fons Europeu de Desenvolupament Regional (FEDER) (project PID2021-122136OB-C21 funded by MCIN/AEI/10.13039/501100011033/FEDER, UE). The first author gratefully acknowledges the Universitat Politècnica de Catalunya and Banco Santander for the financial support of his predoctoral grant FPI-UPC grant.

Appendix. Continuity

In order to prove the C^1 continuity of our tricubic approach, we will use three facts. The first two come from Lekien and Marsden [32]. First, when two cells of the same size share a common face, and the constraints used on the vertices of that face on each of the cells are the same, the tricubic interpolants are C^1 continuous on that face. We also know that the relationship between the coefficients of the interpolant and the constraints we are using is linear and invertible. This means that given a set of constraints, there is one and only one tricubic interpolant that satisfies them. Finally, even though we compute the interpolant relative to the cell, changing coordinates from one cell to another only requires a uniform scaling and a translation. As a result, the computed polynomials are tricubic interpolants regardless of the cell we use as reference.

Now, let us consider the case where we have two cells of different sizes connected through a face. Let us call the largest cell C_L and the smaller one C_S (see Fig. A.11). Remember that, in order to force continuity between these two cells, our approach evaluates the interpolant I_L of C_L and its derivatives, and forces

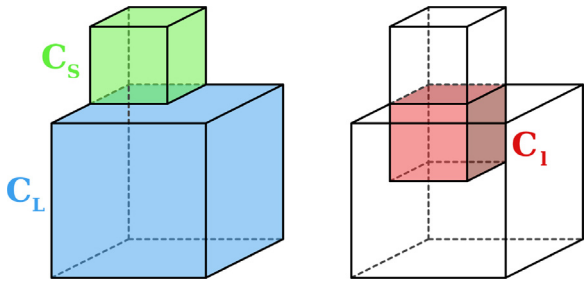


Fig. A.11. Left: Two cells C_L and C_S of different sizes connected through a face. Right: Subcell C_l inside C_L that shares a common face with C_S.

the vertices of C_S on the common face to use them as constraints for the computation of the interpolant I_S of C_S. We need to prove that this guarantees C^1 continuity between C_L and C_S.

If the large node C_L were to be subdivided, one of its subnodes would have the same size as the smaller one C_S and share a face. We will call this subnode C_l, and consider the interpolant I_l that would be obtained from using as constraints the evaluation of I_L at the vertices of C_l. But given that there is only one interpolant satisfying these constraints, and that we can transform I_l to the same coordinate system as I_L, I_L and I_l need to be the same polynomial. Furthermore, as I_l and I_S used the same constraints on their common face, they are C^1 continuous on that face. Thus, I_L and I_S are also C^1 continuous.

The trilinear version is C^0 continuous because of similar properties. The relationship between the coefficients of the interpolant and the used constraints is linear and invertible, so changing from the base of one cell to another still produces a trilinear interpolant, and these polynomials simplify to bilinear interpolants at the cell's faces.

References

[1] Jones MW, Baerentzen JA, Sramek M. 3D distance fields: A survey of techniques and applications. IEEE Trans Vis Comput Graphics 2006;12(4):581–99.
[2] Calakli F, Taubin G. SSD: Smooth signed distance surface reconstruction. In: Computer Graphics Forum. 30, (7):Wiley Online Library; 2011, p. 1993–2002.
[3] Jamriška O, Havran V. Interactive ray tracing of distance fields. In: Central european seminar on computer graphics, vol. 2. Citeseer; 2010, p. 1–7.
[4] Evans A. Fast approximations for global illumination on dynamic scenes. In: ACM SIGGRAPH 2006 Courses. Association for Computing Machinery; 2006, p. 153–71.
[5] Frisken SF, Perry RN. Designing with distance fields. In: ACM SIGGRAPH 2006 Courses. 2006, p. 60–6.
[6] Liu S, Wang CC. Fast intersection-free offset surface generation from freeform models with triangular meshes. IEEE Trans Autom Sci Eng 2010;8(2):347–60.
[7] Xia H, Tucker PG. Finite volume distance field and its application to medial axis transforms. Internat J Numer Methods Engrg 2010;82(1):114–34.
[8] Macklin M, Müller M. Position based fluids. ACM Trans Graph 2013;32(4):1–12.
[9] Mitchell N, Aanjaneya M, Setaluri R, Sifakis E. Non-manifold level sets: A multivalued implicit surface representation with applications to self-collision processing. ACM Trans Graph 2015;34(6):1–9.
[10] Macklin M, Erleben K, Müller M, Chentanez N, Jeschke S, Corse Z. Local optimization for robust signed distance field collision. Proc. ACM Comput. Graphics Interact. Tech. 2020;3(1):1–17.
[11] Park JJ, Florence P, Straub J, Newcombe R, Lovegrove S. Deepsdf: Learning continuous signed distance functions for shape representation. In: Proceedings of the IEEE/CVF Conference on computer vision and pattern recognition. 2019, p. 165–74.
[12] Wang P, Liu L, Liu Y, Theobalt C, Komura T, Wang W. NeuS: Learning neural implicit surfaces by volume rendering for multi-view reconstruction. 2021, arXiv preprint arXiv:2106.10689.

[13] Yariv L, Gu J, Kasten Y, Lipman Y. Volume rendering of neural implicit surfaces. Adv Neural Inf Process Syst 2021;34:4805–15.
[14] Takikawa T, Litalien J, Yin K, Kreis K, Loop C, Nowrouzezahrai D, Jacobson A, McGuire M, Fidler S. Neural geometric level of detail: Real-time rendering with implicit 3D shapes. In: Proceedings of the IEEE/CVF Conference on computer vision and pattern recognition. 2021, p. 11358–67.
[15] Maier D, Hesser J, Männer R. Fast and accurate closest point search on triangulated surfaces and its application to head motion estimation. In: 3rd WSEAS International conference on signal, speech and image processing. 2003, p. 5.
[16] CGAL 5.4.1 - 3D Fast Intersection and Distance Computation (AABB Tree): User Manual.
[17] Bærentzen JA, Aanaes H. Signed distance computation using the angle weighted pseudonormal. IEEE Trans Vis Comput Graphics 2005;11(3):243–53.
[18] Xu H, Barbič J. Signed distance fields for polygon soup meshes. In: Graphics interface 2014. AK Peters/CRC Press; 2014, p. 35–41.
[19] Jacobson A, Kavan L, Sorkine-Hornung O. Robust inside-outside segmentation using generalized winding numbers. ACM Trans Graph 2013;32(4):1–12.
[20] Barill G, Dickson NG, Schmidt R, Levin DI, Jacobson A. Fast winding numbers for soups and clouds. ACM Trans Graph 2018;37(4):1–12.
[21] Krayer B, Müller S. Generating signed distance fields on the GPU with ray maps. Vis Comput 2019;35:961–71.
[22] Sethian JA. A fast marching level set method for monotonically advancing fronts. Proc Natl Acad Sci 1996;93(4):1591–5.
[23] Zhao H. A fast sweeping method for eikonal equations. Math Comp 2005;74(250):603–27.
[24] Yatziv L, Bartesaghi A, Sapiro G. O (N) implementation of the fast marching algorithm. J Comput Phys 2006;212(2):393–9.
[25] Cuntz N, Kolb A. Fast hierarchical 3D distance transforms on the gpu.. In: Eurographics (Short Papers). 2007, p. 93–6.
[26] Schneider J, Kraus M, Westermann R. GPU-based real-time discrete euclidean distance transforms with precise error bounds.. In: VISAPP (1). 2009, p. 435–42.
[27] Mauch S. A fast algorithm for computing the closest point and distance transform. 2000, http://www.acm.caltech.edu/seanm/software/cpt/cpt.pdf.
[28] Sigg C, Peikert R, Gross M. Signed distance transform using graphics hardware. In: IEEE Visualization, 2003. VIS 2003.. IEEE; 2003, p. 83–90.
[29] Sud A, Otaduy MA, Manocha D. DiFi: Fast 3D distance field computation using graphics hardware. Comput Graphics forum 2004;23(3):557–66.
[30] Sud A, Govindaraju N, Gayle R, Manocha D. Interactive 3d distance field computation using linear factorization. In: Proceedings of the 2006 Symposium on interactive 3D graphics and games. 2006, p. 117–24.
[31] Erleben K, Dohlmann H. Signed distance fields using single-pass gpu scan conversion of tetrahedra. Gpu Gems 2008;3:741–63.
[32] Lekien F, Marsden J. Tricubic interpolation in three dimensions. Internat J Numer Methods Engrg 2005;63(3):455–71.
[33] Ju T, Losasso F, Schaefer S, Warren J. Dual contouring of hermite data. In: Proceedings of the 29th Annual conference on computer graphics and interactive techniques. 2002, p. 339–46.
[34] Qu H, Zhang N, Shao R, Kaufman A, Mueller K. Feature preserving distance fields. In: 2004 IEEE Symposium on volume visualization and graphics. IEEE; 2004, p. 39–46.
[35] Sanchez M, Fryazinov O, Fayolle P-A, Pasko A. Convolution filtering of continuous signed distance fields for polygonal meshes. Comput Graph Forum 2015;34(6):277–88.
[36] Setaluri R, Aanjaneya M, Bauer S, Sifakis E. SPGrid: A sparse paged grid structure applied to adaptive smoke simulation. ACM Trans Graph 2014;33(6):1–12.
[37] Museth K. VDB: High-resolution sparse volumes with dynamic topology. ACM Trans Graph 2013;32(3):1–22.
[38] Museth K, Lait J, Johanson J, Budsberg J, Henderson R, Alden M, Cucka P, Hill D, Pearce A. OpenVDB: an open-source data structure and toolkit for high-resolution volumes. In: Acm Siggraph 2013 Courses. 2013.
[39] Wu J, Kobbelt L. Piecewise linear approximation of signed distance fields. In: VMV. 2003, p. 513–20.
[40] Jones MW. Distance field compression. J. WSCG 2004.
[41] Bán R, Valasek G. First order signed distance fields. In: Eurographics (Short Papers). 2020, p. 33–6.
[42] Koschier D, Deul C, Bender J. Hierarchical hp-adaptive signed distance fields. In: Symposium on Computer Animation. 2016, p. 189–98.
[43] Tan YW, Chua N, Koh C, Bhojan A. RTSDF: Real-time signed distance fields for soft shadow approximation in games. 2022, arXiv preprint arXiv:2210.06160.