Bachelor Thesis

August 2007

Volume data generation from triangle meshes using the signed distance function

Darmstadt University of Technology, Germany
Department of Computer Science
The Interactive Graphics System Group

Supervisors: PD Dr. Frank Zeilfelder, Thomas Kalbe

Simon Fuhrmann

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, 14.08.2007	
Ort, Datum	Simon Fuhrmann

Contents

1 Introduction

Volumetric data plays an important role in numerous applications such as in medical representation of CT and MRI scans and high quality rendering of surfaces. Also, signed distance fields in three dimensions, which are a preprocessing result of volumetric data computation, are used for animation of complex shapes (such as hair [1], clothing [2] or fire [3]), collision detection [4, 5], (re)construction of models [6, 7], creation of image/volume skeletons [8, 9, 10, 11, 12], morphing [13, 14] and for various other techniques, e.g. the popular set level method is initialized by a signed distance field.

Figure 1: Volumetric hair [1] and fire simulation [3]

Since triangle meshes are a common, wide spread representation for geometric objects, the need for converting these meshes to volumetric data arises. This process is called *voxelization*, and this document deals with the calculation of signed distance fields and volumetric data from meshes. Further a free and open source application is presented which allows the production of volumetric data from triangle meshes in user-defined resolution and precision.

The document is organized as follows: In section 2 the representation of meshes along with some quality characteristics is described. Meshes need to obey some requirements for the generation of volumetric data, which are discussed.

Section 3 specifies the properties of a volume data set and names a few visualization techniques for volumetric data.

In order to generate volumetric data, a *signed distance field*, or *SDF* for short, needs to be created. A SDF contains shortest signed distances at discrete points in space to a solid, evaluated with the signed distance function, which is described in section 4. Implementation details of how the signed distance function is evaluated in practice are presented.

Obviously, naive calculation of SDFs takes considerably much time, which makes the method infeasible in practice for high quality resolutions. Apart from the naive Brute Force method other techniques with much better runtime have been developed and are introduced in section 5. These techniques include hierarchical approaches, exploits of mesh characteristics and distance transforms.

2 Triangle meshes

This section gives a description of triangle meshes, the formats in which they exist, how they can be read from and written to file, and how they can be stored inside a computer program (section 2.1). This section also describes the idea behind high-quality triangle meshes with consideration to precision and mesh size.

For the sake of volume data generation the mesh needs to fulfill some requirements. If a model does not meet these requirements, the voxelization may lead to unexpected results, mainly because voxels are wrongly classified. This is discussed in section 2.3.

2.1 Representation of meshes

Triangle meshes exist in various formats both inside a file for permanent storage and inside a computer program. Typical file formats are the easy OFF model format or the PLY format from the Stanford Graphics Laboratory, which is quite similar to the OFF model format. These simple types of model formats only capture the most essential information of a mesh: Vertices (points in space) and the connectivity of these vertices in terms of triangles. Even though other primitives like quadrangles, or generally n-sided polygons are allowed, triangles are the most pleasant ones because at most three points are guaranteed to lie exactly on one plane (triangles are always planar, quadrangles or higher order polygons may be not). For efficiency and uniformity all polygons are typically converted to triangles by inserting new edges, which is always possible as long as the polygon is concave.

There are also various other, much more sophisticated model formats that can capture more information than just the model surface, for example normals, texture coordinates, material properties and animations. Examples of such model formats are ID Software's MDL (Quake), MD2 (Quake 2), MD3 (Quake 3) and MD5 (Doom 3, Quake 4) that mainly aims towards games. MA and MB are Maya formats, Maya ASCII and Maya Binary respectively. Other formats of commercial modeling and rendering software exist like LWO (Lightwave) and ASE (3D Studio Max).

Since OFF models are simple and cover all required features, only OFF models are explained in more detail. Other model formats are out of scope of this document.

Every OFF model file starts with a simple file format descriptor, a line with the string `OFF`. The next line contains whitespace separated integers (in ASCII) with the number of vertices V, the number of faces F and the number of edges E (which is rarely used and often just set to zero). The list of vertices follows, one vertex (three floating point values) per line, and the list of faces affiliates. Every face contains the number of vertices n and n vertex index values. Figure 2 contains a simple OFF model, a three dimensional triangle, which contains both triangles and quadrangles. The quadrangles are converted to triangles in the picture.

One common representation of meshes inside a computer program does not significantly differ from the representation inside a file: A list of vertices is maintained, each element

```
OFF
6 5 0
0.0 0.0  0.1
1.0 1.0  0.1
0.0 1.0  0.1
0.0 0.0 -0.1
1.0 1.0 -0.1
0.0 1.0 -0.1
3 0 1 2
3 5 4 3
4 0 3 4 1
4 0 2 5 3
4 2 1 4 5
```

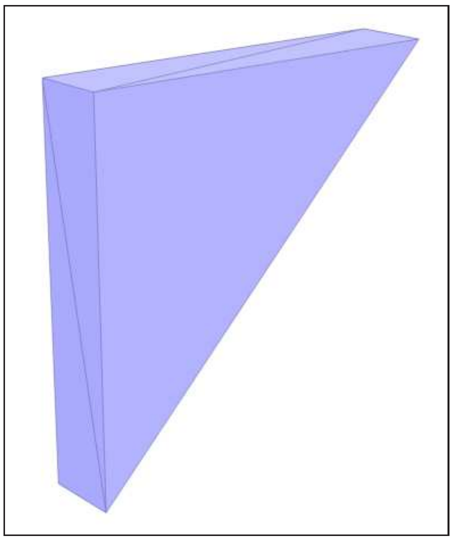


Figure 2: Example of an OFF model file: A 3D-triangle

consists of three floating point values. For each vertex a normal is calculated and stored in another list. This is required for smooth shading of the mesh. A third list contains the connectivity of the vertices, forming triangles by specifying index values to the vertices. For each face a normal is also calculated, these normals are stored in a forth list.

This structure can be built easily while parsing the OFF file. At first the list of vertices is created and the list of vertex normals is initialized with zero vectors. Then the list of face indices is build. For every face the normal is calculated and assigned to the list of face normals. Additionally the face normal is added up to the corresponding three vertex normals using the index values. After reading all faces, the vertex normals need to be normalized in a separate pass.

This slim and easy organization of vertices, faces and their normals is good for rendering as well as for reading and writing OFF files, but there are other aspects that cannot easily be accomplished with such an organization. For example there is no easy iteration over edges in the model. Apart from this simple organization there are a lot of other structures for storing model information with different advantages and disadvantages.

Most alternative representations are based on the description of edges. This is because several algorithms are based on edges, e.g. removal of edges (edge collapse) for mesh simplification or creation of edges for subdivision techniques. These algorithms need to use different representations to perform queries on the mesh in an efficient way. Favorite representations are *winged edge*, *half edge*, *directed edge* or *quad edge* [15, 16, 17], just to name a few. The main drawbacks of these representations are the memory overhead and the inability to store arbitrary meshes. For example most of the representations cannot store non-2-manifold meshes without modification. Moreover efficient rendering with these representations might not be possible.

The creation of volumetric data mainly requires iteration over all faces, sometimes edge normals need to be queried. Thus the simple linear representation of meshes is a good choice.

2.2 Quality of meshes

Since meshes are quite difficult to create without computational aid, most of the meshes are created either using computer programs for interactive modeling or 3D scanning techniques for real world objects which survey discrete points on the object's surface. The former method has its difficulties in the modeling process while the latter is very time consuming and typically results in an enormous amount of data (point clouds), which needs to be transformed to a triangle mesh, often only with semi-automatic tools. Also, these triangle meshes suffer from various artifacts (drop ins) that have been recorded during the scanning process. Figure 3 shows a few artifacts of the Stanford Dragon.

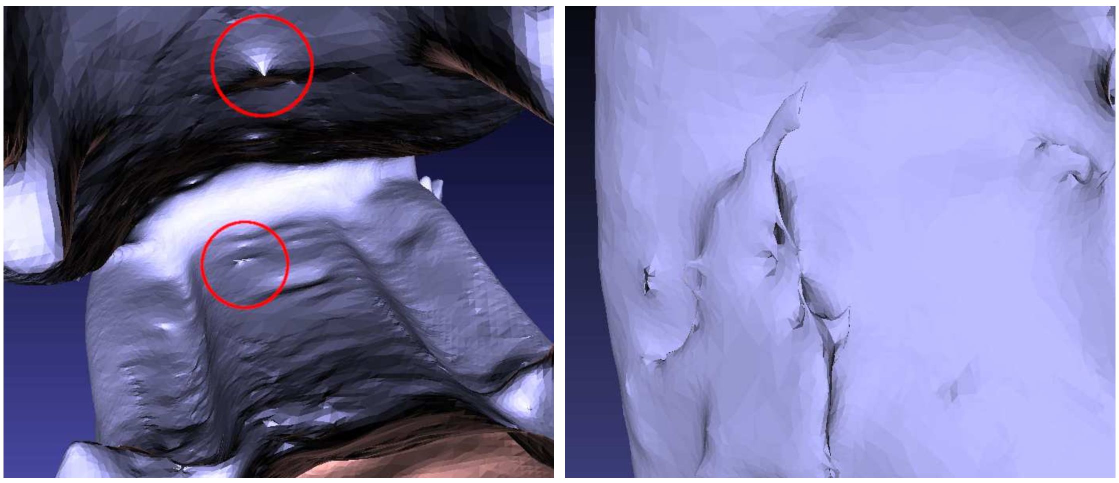


Figure 3: Artifacts of the Dragon: Mouth (left) and leg (right)

The quality of a model can hardly be captured into units. If a triangle mesh is constructed from a point cloud that has been created using a real world object, the divergence to the real world object is a quality factor. Drop ins, or noise, adulterate the quality and point clouds are typically smoothed to reduce the noise (which, on the other hand, reduces quality of sharp corners and edges). [18] discusses how to determine the errors that may occur.

The amount of triangles also gives a clue about the quality of the model but adversarial triangulation can also lead to ill-favoured meshes. A good triangulation of a model cares about local properties such as the curvature of the model. Nearly planar regions of the model can be captured using relatively few triangles, while uneven regions with high curvature need to be approximated with more triangles. Consider figure 4 which shows how triangulation should be changed in regions with high curvature.

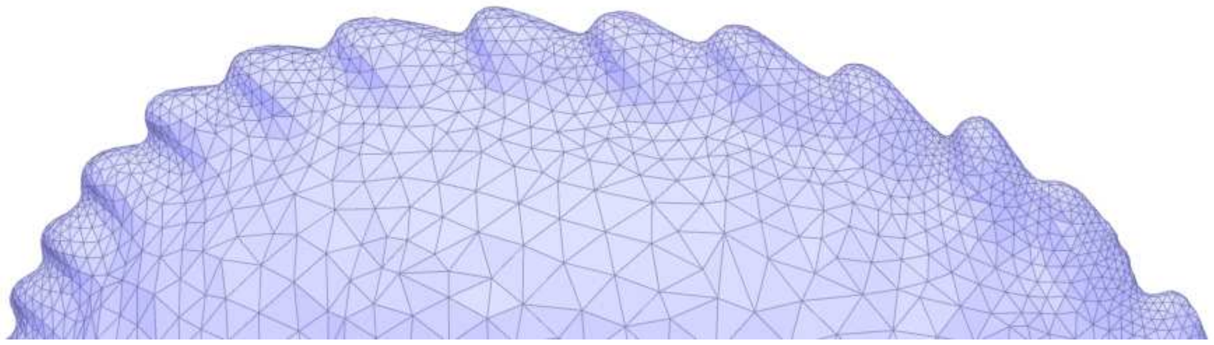


Figure 4: Dragons back with increased triangles at higher curvature

Another factor of quality is the shape of the triangles. Triangles which are nearly equilateral deliver the best visual results, while triangles with acute angles deliver worse results and can lead to numeric instabilities in algorithms. Equilateral triangles deliver best ratio between circumference and surface, thus less triangles are needed in order to cover the models surface completely. Also, a mesh of equilateral triangles implies vertices with a degree of six, which means that each vertex is shared with exactly six triangles. This produces the most attractive results as opposed to degrees smaller or greater than six.

For already existing model files or point clouds some high quality automatic remeshing techniques have been developed, which can be used to decrease the amount of triangles without drastically reducing the precision of a mesh. Figure 5 shows a meshe with its original triangulation and a remeshed version using a tool[1] that has recently been developed in Darmstadt University of Technology.

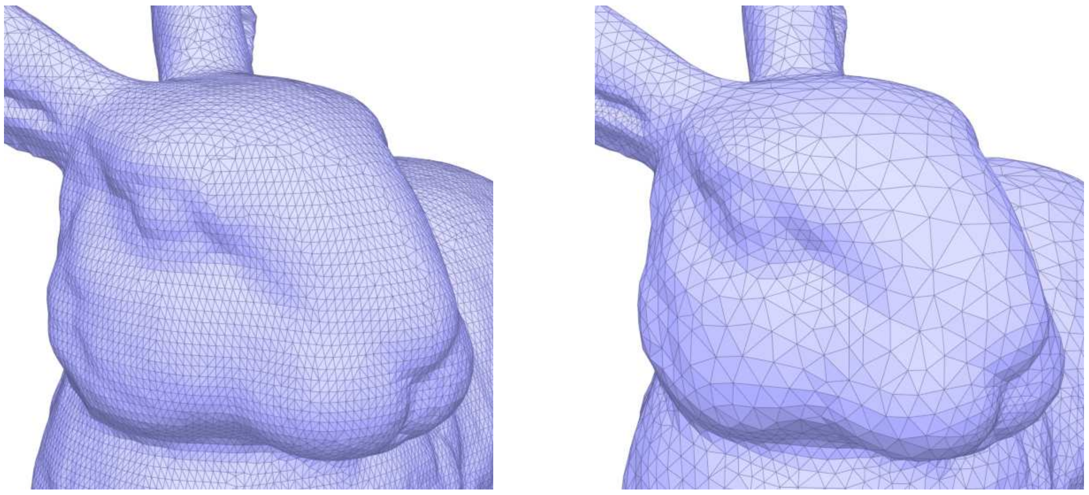


Figure 5: Original Stanford Bunny (70k faces) and remeshed version (26k faces)

[1]http://sourceforge.net/projects/cloudmesh/

2.3 Requirements on the mesh

If volume data needs to be created from triangle meshes, it is important that space is divided into an interior (everything that is inside the mesh), an exterior (everything outside the mesh) and the mesh surface itself. If the mesh partitions space in that way, we talk about *closed, orientable manifolds.* For these meshes it is required that the following conditions [32] hold:

- The mesh does not contain any self-intersections and triangles may only share edges and vertices.
- Every edge must be adjacent to exactly two triangles.
- Triangles incident on a vertex must form a single cycle around that vertex.

These conditions are important in order to correctly determine the sign of a voxel within the SDF. The sign indicates whether the voxel is inside or outside the mesh and leads to the so called *voxel classification.*

Unfortunately, the unpleasant truth is that most of the triangle meshes have invalid triangulation for voxelization and do not meet the above requirements. Lots of meshes (especially those that are created from real world objects) have small holes, self intersections or borders (which are basically bigger holes). On the other hand violations of the requirements may not necessarily introduce noticeable errors in the calculation of the signed distance field. But if errors are noticeable they typically spring from the classification of a voxel. The result of the classification also depends on the strategy that is used to classify voxels. These strategies are discussed in section 4.3.

If invalid triangulation of a mesh becomes an issue and errors are visible in the resulting volumetric data, the mesh need to be fixed. The mesh can either be inspected by some model editing software or can be remeshed, also in order to reduce the amount of triangles and speed up the voxelization process. Note that the resulting remeshed model is not necessarily valid for voxelization. If the mesh already has high quality triangulation or relatively few triangles, remeshing is rarely an option and manual editing is necessary. Luckily the editing process is mainly aimed towards filling holes (which can also be tricky and one should be careful to not create new self-intersections). Already existing self-intersections can be fixed by deleting the involved triangles (and maybe also a small region around) and fill the resulting hole. Zero surface triangles need to be deleted in every case because they may cause undefined behavior in computer programs if they are not explicitly detected.

`MeshLab`[2] is a free, small and helpful program for simple mesh inspection and manipulation.

[2]http://meshlab.sourceforge.net

3 Volume data

Volumes are the counterpart for pictures in three dimensions. Just like a picture contains *pixels* (a short from of *picture element*), a volume contains *voxels*, or *volume elements*. A single pixel in a picture typically encodes a color value, for example in RGB, RGBA, or an indexed color. Voxels however do not encode color values. It depends on the application how voxels are interpreted, but in general a voxel encodes a density value. For example a solid real-world object can be represented with two different density values. One for the background (the air around that object) and a denser one for the object itself. The technical details are discussed in section 3.1.

Pictures are made for direct visualization on screen. Volumes however cannot be displayed directly like pictures, they need to be transformed to 2D space first. This can be done using various techniques referred to as *volume visualization*. Some of these techniques are described in section 3.2.

3.1 Properties of volume data

Volumetric data consist of voxels in three dimensions. One essential information is the amount of voxels along each dimension. Secondly, the distance between two voxels along each dimension is important for unbiased visualization. This property is not present for 2D pictures, because two neighboring pixels (in a 4-neighborhood N_4) are equidistant. Thirdly, the data type of the voxels needs to be known.

- Dimension of the volume: $n_x, n_y, n_z \in \mathbb{N} \setminus \{0, 1\}$
- Distance of neighboring voxels: $d_x, d_y, d_z \in \mathbb{R}^+ \setminus \{0\}$
- Data type: $t \in \{\mathsf{UCHAR}, \mathsf{USHORT}, \mathsf{FLOAT}\}$

The uncompressed size of a volume data set is $\text{sizeof}(t) \cdot n_x \cdot n_y \cdot n_z$.

Volumes are typically stored in RAW files. The format of a RAW file is simple: All density values are stored one after another, starting with $(x, y, z) = (0, 0, 0)$, $(x, y, z) = (1, 0, 0)$, and so on. For a given dimension n_x, n_y, n_z, a loop that reads all values from a data source *ds* into a three dimensional array *data* may look like this:

```
for ($z = 0$; $z < n_z$; $++z$)
  for ($y = 0$; $y < n_y$; $++y$)
    for ($x = 0$; $x < n_x$; $++x$)
      $data[z, y, x] = ds$.next
```

Unfortunately RAW files do not contain any header information, which means that all mentioned properties above need to be provided along with the RAW file. For this purpose I propose a simple header file for RAW data that is easy to parse and to extend. Since the header information is separated from the RAW file, all existing applications based on headerless RAW files are unaffected.

The syntax of the header file is basically in INI format, all relevant properties of the volume are specified under the raw section. Extensibility is simply given through sections with names different from raw. The #-sign is used as line comment indicator.

```
[raw]
# The filename that contains the data
data = <RAW filename>
# Data type for the voxels
type = <UCHAR | USHORT | FLOAT>
# Number of voxels along X, Y and Z
dimension = < n_x > < n_y > < n_z >
# Distance ratio between voxels along X, Y and Z
ratio = < d_x > < d_y > < d_z >
```

Imaginary extensions would be a compression section which indicates that compression is used and specifies the algorithm to decode the data, or a meta section that contains meta information about the generation of the data. Important sections that contain essential information could be marked as critical with an exclamation mark ("!") as first character of the section name, and if a client does not understand a critical extension, read should fail. An example would be the !compression section. A header file could look like that:

```
[raw]
data = aneurism.raw.bz2
type = UCHAR
dimension = 256 256 256
ratio = 1.0 1.0 1.0

[meta]
ripped_from = http://volvis.org

[!compression]
algorithm = bzip2
```

3.2 Visualization of volume data

Visualization of volume data is often interactive and in real time — OpenGL or Direct3D is used for this task. There are various methods and papers out there that aim to produce high quality, interactive images [21, 24]. The existing methods can roughly be divided into *direct volume visualization* and *ISO surfacing*.

The simplest type of direct volume visualization is to draw a point in space for each voxel with its density as alpha value for that point, but only if a specified ISO value is lower or equal to the density value. In spite of the simplicity of this approach, the results are relatively good, and performance is outstanding. Of course, visual quality of the result does not scale well, lighting and shading techniques are not possible.

One common approach is to generate an *ISO surface triangulation* of the volume using the *marching cubes* algorithm [19]. The resulting triangle mesh is then displayed. The marching cubes algorithm is easy to implement on the one hand, but on the other hand the generated triangulation typically contains a lot of small, misshaped triangles. Sharp features of the object are not captured unless the original algorithm is modified (see figure 6 for an example). Such a modification was made in the feature sensitive marching cubes algorithm [20].

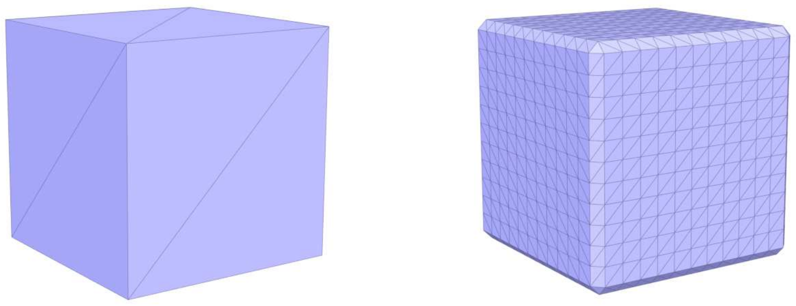


Figure 6: Original box model (left) and the MC triangulation (right)

The general problem with triangulations is that there is no good control of level of detail (or *LOD*). In other words, triangles far away from the viewer may be smaller than a single pixel of the screen, and nearby triangles are not small enough to be a good approximation of the real surface. This problem can be solved with techniques that are not based on triangulations.

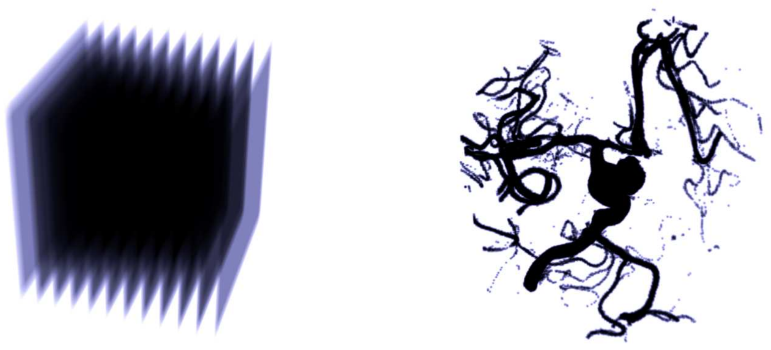


Figure 7: Texture slices concept (left) and visualization using texture slices (right)

Another common approach to visualize volumetric data is to use a *stack of 2D textures* or *texture slices*. For example for each xy-plane in the volume a texture is generated with the ISO surface contour on it. The rest of the texture remains transparent, so a viewer can see the whole ISO surface in the stack of 2D textures. This can also be implemented with OpenGL 3D textures. From experience this leads to good results with sparse data sets, e.g.

for medical data of veins. But for big, solid, volumetric objects that have been generated from triangle meshes, this approach gives poor results. Additionaly this approach is slow for big data sets. If the camera looks along the x- or y-plane, one can see the different texture slices. To avoid this, the texture slices need to be realigned if the camera moves.

A more sophisticated approach is to use *ray casting* to visualize the volume. For each pixel on the viewport, a ray is cast into the scene. If the ISO surface is intersected by the ray, a color for that pixel is calculated using the point of intersection. Typically octrees are used to speed up rendering. Ray casting enables a wide range of possibilities. For example calculation of the intersection with the ISO surface may be based on spline surfaces [22, 23]. This allows smooth approximation of the surface. Ray casting can be done interactively, even on the GPU of the graphics card, and it features automatic LOD. Similar to ray casting, *ray tracing* [24] can be implemented using recursive algorithms to trace rays on reflecting surfaces. But performance typically does not allow interactive rendering on desktop machines.

4 Signed distance fields

Signed distance fields are similar to volumetric data. The difference is that volume data sets consist of density values on discrete points in space, the voxel grid. These density values are unsigned. A SDF consists of shortest distance values from a voxel to a solid in space (the triangle mesh), and these distances can either be positive or negative, thus the values are *signed*. If the sign of the distance is negative, the voxel is located inside the solid, if it's positive, the voxel is outside.

The main difficulty is to find the sign and the distance in an efficient way. The following sections give an in-depth understanding on how the distance and the sign are calculated. Then the mapping scheme from SDFs to volumetric data is described, which is quite simple but critical if high quality data is expected.

4.1 Signed distance function

From a formal point of view, a triangle mesh M is an infinite set of points describing the surface of the mesh. The signed distance is measured with both a distance function $dist_M$ and a sign function $sign_M$. The combination of both is called the *signed distance function* sd_M. The distance function

$$dist_M(p) = \inf_{x \in M} ||x - p||$$

returns the distance from any point p to the closest point x on the mesh. Note that inf denotes the infimum, which is the greatest lower bound. The closest point x might not be unique. The sign function clearly is:

$$sign_M(p) = \begin{cases} -1 & \text{if } p \text{ inside } M \\ +1 & \text{if } p \text{ outside } M \end{cases}$$

And the signed distance function is the product of both:

$$sd_M(p) = sign_M(p) \cdot dist_M(p)$$

Typically the euclidean distance $|| \cdot ||_2$ is used as metric, which leads to *signed euclidean distance fields*. These definitions don't tell anything about generation the SDF in practice. Thus the next sections deal with the computational calculation of sign and distance that can be implemented in a program.

4.2 Distance calculation

In this section the calculation of the distance from an arbitrary point in space (a grid point) to the mesh is explained. To be more precise, this section describes how the distance from a point to a single triangle is calculated. To find the smallest distance to the whole mesh, the distances to all triangles need to be calculated. Determining the smallest distance from a point to a triangle is not as easy as it sounds at first sight, especially if this needs to be done very fast.

Let us denote the point in space with $P \in \mathbb{R}^3$, and the three vertices of a triangle with $A, B, C \in \mathbb{R}^3$. Since every triangle has an implicit orientation, the points A, B, C are specified counter clockwise. Let us further denote the three edges of a triangle as $E_1 = \overline{AB}$, $E_2 = \overline{BC}$ and $E_3 = \overline{CA}$, and the face of the triangle as F. It can be observed that there are three possible cases for the closest point P_c on the triangle:

- P_c could be a vertex A, B, C
- P_c could be on an edge E_i
- P_c could be inside the face F

We will denote every possible case as a *feature* of the triangle, thus the triangle has seven features, three vertices A, B, C, three edges E_i and the face F itself.

It would be much too expensive to calculate the closest point for every feature separately, calculate the distances to these points and select the closest one. A much better approach is to determine which case is applicable and calculate the distance only once. Two methods of calculating the distance from a point to a triangle have been implemented and compared by Mark W. Jones in [25]. The first method calculates the distance in 3D, the second one uses a transformation on the triangle and makes the problem 2D.

Firstly, inspection of the problem in 2D offers more mathematical possibilities to determine which of the above mentioned cases is applicable. Secondly, the 2D method is more efficient by a factor of 4 than the 3D method. This is basically caused by the number of square roots that need to be calculated in 3D. The 3D method needs four square root operations as opposed the one single, final square root for the 2D method. However, if the closest point on the triangle needs to be known, the 3D method seems to be more practical, because with the 2D method the resulting closest point is still in transformed space. This makes no difference for the distance, but the closest points of the two methods are different ones.

For the sake of distance calculation we are not interested in the closest point but only in the distance. Interestingly, even sign calculation with normals that directly rely on the closest point (or to be more precise, on the distance vector from P to P_c) work with the 2D method with a small trick. This is described later on.

The next steps describe how to transform a triangle to 2D space, how to determine the closest feature on the triangle, and how to calculate the final distance from P to the triangle.

4.2.1 Transformation to 2D space

A triangle is the only polygon that always lies on a plane. This makes it always possible to transform a triangle onto a plane which has a trivial mapping from 3D- to 2D-coordinates and vice versa. In this section the triangle will be transformed to the yz-plane of the coordinate system such that vertex A lies on the origin, and vertices B and C lie somewhere in the yz-plane. The normal of the transformed triangle points along the negative x-axis.

Mark W. Jones proposed a method where A lies on the origin, B lies on the z-axis and C lies in the yz-plane. Getting B on the z-axis requires an additional rotation which costs extra time per triangle. But there is no need for this rotation since it has no effect on the distance calculation. For this reason my method differs a bit from that one Jones proposed.

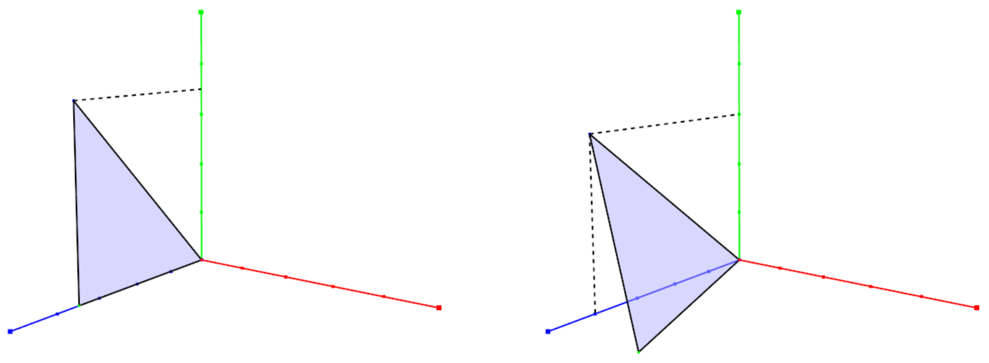


Figure 8: Triangles transformed to yz-plane with Jones' and my method

The first transformation that needs to be done is a translation of A in the origin. For this reason homogeneous coordinates are used to enable translation via matrix multiplication.

$$T_1 = \begin{pmatrix} 1 & 0 & 0 & -A_x \\ 0 & 1 & 0 & -A_y \\ 0 & 0 & 1 & -A_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The second transformation, the rotation, is a bit more tricky. The rotation is done using a rotation vector. The rotation matrix R around an arbitrary vector v with α degrees is as follows:

$$R = \begin{pmatrix} \cos\alpha + v_x^2(1-\cos\alpha) & v_xv_y(1-\cos\alpha) - v_z\sin\alpha & v_xv_z(1-\cos\alpha) + v_y\sin\alpha \\ v_yv_x(1-\cos\alpha) + v_z\sin\alpha & \cos\alpha + v_y^2(1-\cos\alpha) & v_yv_z(1-\cos\alpha) - v_x\sin\alpha \\ v_zv_x(1-\cos\alpha) - v_y\sin\alpha & v_zv_y(1-\cos\alpha) + v_x\sin\alpha & \cos\alpha + v_z^2(1-\cos\alpha) \end{pmatrix}$$

As rotation vector v we use the cross product of the normal of the triangle and the normal of the triangle in its desired position. The normal n of the triangle ABC and the normal

$\tilde{n}$ of the triangle in its desired position are:

$$n = \frac{(B-A)\times(C-A)}{||(B-A)\times(C-A)||} \qquad \tilde{n} = \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix}$$

Now we get the rotation vector v as the cross product of both normals:

$$v = n \times \tilde{n} = (0, -n_z, n_y)^T$$

The values $\cos\alpha$ and $\sin\alpha$ that are used in the rotation matrix R are calculated with the law of cosines:

$$\begin{aligned} \cos\alpha &= \left(\frac{< n \mid \tilde{n} >}{||n|| \cdot ||\tilde{n}||}\right) = -n_x \\ \sin\alpha &= \sin(\operatorname{acos}(\cos\alpha)) = \sin(\operatorname{acos}(-n_x)) \end{aligned}$$

The resulting transformation matrix T_2 arises by substituting v_x, v_y and v_z with $0, -n_z$ and n_y in the general rotation matrix R:

$$T_2 = \begin{pmatrix} \cos\alpha & -n_y\sin\alpha & -n_z\sin\alpha & 0 \\ n_y\sin\alpha & \cos\alpha + n_z^2(1-\cos\alpha) & -n_z n_y(1-\cos\alpha) & 0 \\ n_z\sin\alpha & -n_y n_z(1-\cos\alpha) & \cos\alpha + n_y^2(1-\cos\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The vertices A, B, C can now be transformed with T_1 and T_2:

$$\begin{aligned} \tilde{A} &= T_2T_1 \cdot A \\ \tilde{B} &= T_2T_1 \cdot B \\ \tilde{C} &= T_2T_1 \cdot C \end{aligned}$$

We define the z- and y-axis in 3D space as the new x- and y-axis in 2D space, and the coordinates $a, b, c \in \mathbb{R}^2$ of the triangle in 2D space can be trivially written as:

$$\begin{aligned} a &:= (\tilde{A}_z, \tilde{A}_y)^T \\ b &:= (\tilde{B}_z, \tilde{B}_y)^T \\ c &:= (\tilde{C}_z, \tilde{C}_y)^T \end{aligned}$$

4.2.2 Feature determination

The feature determination is responsible for finding the closest feature (one of the vertices A, B, C, one of the edges E_1, E_2, E_3 or the face F itself) from a point P in space. Since the 2D representation of the triangle is used to find that feature, the point P must also be transformed according to the triangle.

$$\begin{aligned} \tilde{P} &= T_2T_1 \cdot P \\ p &:= (\tilde{P}_z, \tilde{P}_y)^T \end{aligned}$$

The point P typically does not lie on the same plane as the triangle, this means that $\tilde{P}_x$ is usually not zero. By ignoring this x-value of $\tilde{P}$, $\tilde{P}$ is projected on the yz-plane, thus p is just a projection of the real point. But $\tilde{P}_x$ has not been computed for nothing. In fact this is exactly the distance from P to the triangle if and only if F is the closest feature for P. This distance can be used later if F has been exposed as the closest feature.

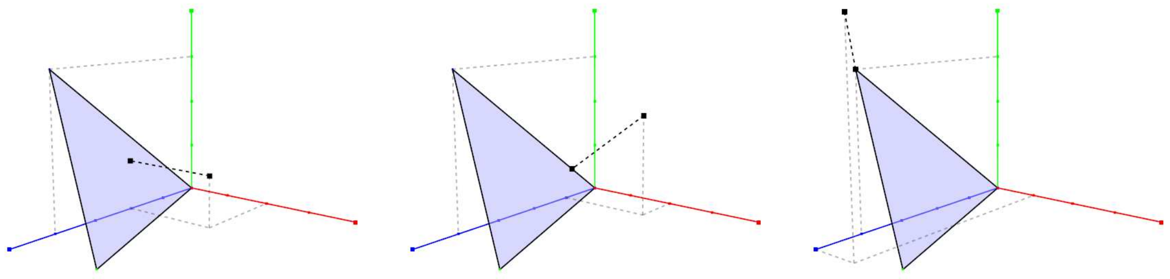


Figure 9: $\tilde{P}$ with face, edge and vertex as closest feature

The determination of the closest feature is based on a tree of decisions whether p is either right or left of the edges of the triangle. The three triangle edges are not enough for this task. For every edge two helper edges are introduced. These helper edges are right-angled on their corresponding triangle edge. See figure 10 for an illustration. All edges are expressed with a position vector (for every edge a corresponding triangle vertex can be found as position vector) and a direction vector. The direction vector of the triangle edges obviously is the difference of two vertices, e.g. $r_1 = b - a$ for e_1, the direction vector of the corresponding helper edges is e.g. $r_4 = r_5 = (r_{1y}, -r_{1x})^T$ for e_4 and e_5.

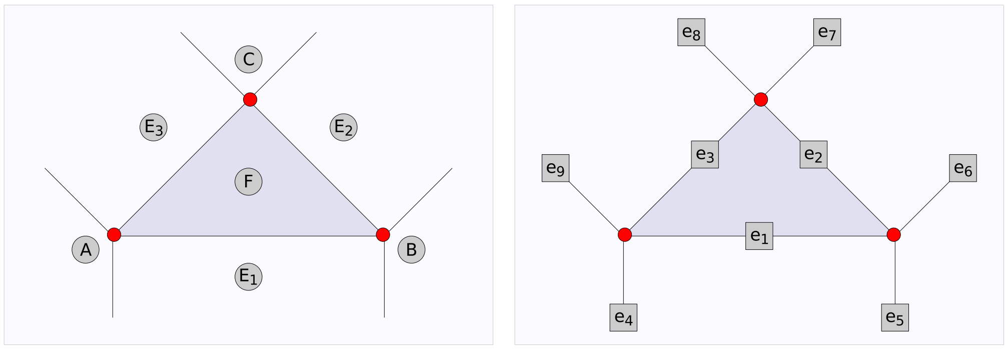


Figure 10: Triangle regions (left) and helper edges for region determination (right)

To determine whether p is on the left or right of an edge e, a special equation, the *edge equation*, can be used. The edge equation $E_e : \mathbb{R}^2 \to \mathbb{R}$ for an edge e through (X, Y) with gradient $\frac{dY}{dX}$ is defined as follows:

$$E_e(p) = (p_x - X) \cdot dY - (p_y - Y) \cdot dX$$

The meaning of a specific value of E_e is not important here, only the sign of the result is expressive, because it determines if p is left, right or directly on edge e:

$$\begin{aligned} E_e(p) &< 0 \quad \text{if } p \text{ is left of } e \\ E_e(p) &> 0 \quad \text{if } p \text{ is right of } e \\ E_e(p) &= 0 \quad \text{if } p \text{ is on } e \end{aligned}$$

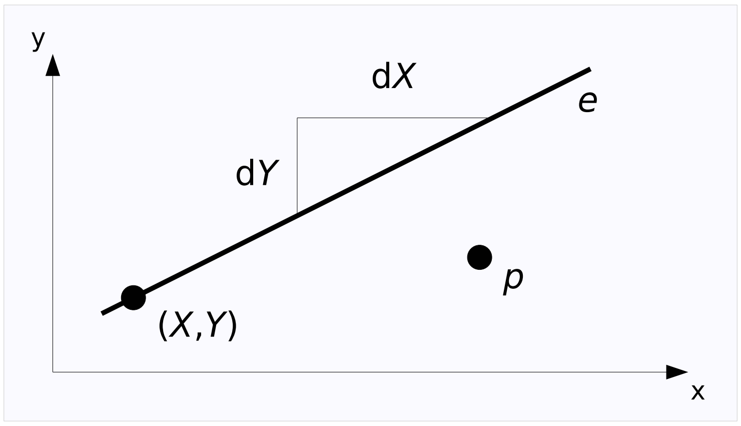


Figure 11: Edge e through (X, Y) with gradient $\frac{dY}{dX}$. p is right of e.

This equation is evaluated for the edges in figure 10 according to the tree given in figure 12. This tree can be hardcoded, but this is a time-consuming task. It is much more comfortable to check for the specific features. For example if p is left of e_1, e_2 and e_3, the closest feature is F. Or if p is right of e_4 and left of e_9, the closest feature is A. This strategy may evaluate more equations until the feature is found, but evaluation is very fast and the code is much cleaner. The maximum number of edge equations that need to be evaluated is nine (the number of edges) instead of six evaluations when using the tree for determination.

If the closest feature has been determined, the final distance calculation can be done. This task is relatively simple because most of the work has already been done.

4.2.3 Final distance calculation

Depending on which feature is closest, the distance calculation is more or less an easy task. But there is one thing that needs to be mentioned when talking about distances. Most of the time a distance is calculated using the euclidean norm of a vector $v \in \mathbb{R}^3$, $||v||_2 = \sqrt{v_x^2 + v_y^2 + v_z^2}$. As mentioned before taking the square root of a value is an expensive operation. Typically one would calculate the quadratic distance in cases where it is possible to work with it, and take the square root after the work is done. In this case the quadratic distance norm $||v||_q := ||v||_2^2 = v_x^2 + v_y^2 + v_z^2$ is used. This optimization can always be applied if only comparison, multiplication and division are used with quadratic distances. All other arithmetic operations do not work!

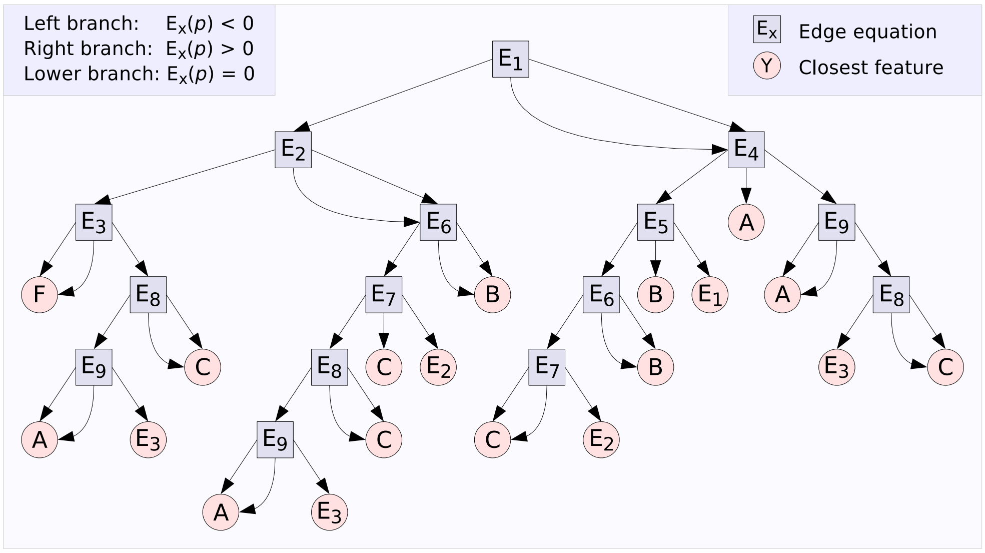


Figure 12: Decision tree for feature determination

Face feature F: The distance of this feature has already been calculated with the transformation of point P to $\tilde{P}$. With the projection of $\tilde{P}$ to p in 2D coordinates the x-value has just been ignored. This value is exactly the distance from P to the triangle.

Vertex feature A, B, C: If one of the vertices has been determined as the closest feature, call it $X \in \{A, B, C\}$, the distance can be trivially calculated with $||P - X||_2$. Note that this is done in 3D-space.

Edge feature E_i: If an edge has been determined as the closest feature, the closest point on this edge still needs to be found. All calculations can be done in three dimensions with the original coordinates of A, B, C and P to get the most accurate result. The problem of finding the closest point X on an edge E for a point P can be solved as follows:

Let us describe edge E with a position vector $\vec{v}$ (one of the triangles vertices) and a direction vector $\vec{r}$ (the difference of the two corresponding vertices of the edge), $E = \vec{v} + \lambda \cdot \vec{r}$. A Hessian plane is constructed in point P with $\vec{r}$ as perpendicular. This plane has equation $< \vec{r} \mid \vec{x} > = < \vec{r} \mid P >$. We substitute $\vec{x}$ with E and get

$$< \vec{r} \mid \vec{v} + \lambda \cdot \vec{r} > = < \vec{r} \mid P > \quad \Leftrightarrow \quad \lambda = \frac{< \vec{r} \mid P - \vec{v} >}{< \vec{r} \mid \vec{r} >}$$

The point of intersection X is calculated with substituting λ in E:

$$X = \vec{v} + \frac{< \vec{r} \mid P - \vec{v} >}{< \vec{r} \mid \vec{r} >} \cdot \vec{r}$$

Lastly we can calculate the distance with $||P - X||_2$. Since $\vec{r}$ is typically as long as the whole edge of the triangle (because it has been calculated with subtraction of two triangle vertices), $\lambda \in [0, 1]$ should always hold. But in practice it does not. This may be caused by numeric instabilities during transformation of the triangle.

4.3 Sign calculation

The sign of a voxel indicates if a voxel is located inside or outside the solid. The calculation of the sign can be done in basically two different ways. Historically, scan conversion is used to determine the sign. How scan conversion works and what the benefits and drawbacks are, is described in section 4.3.1. Then another modern technique for this task is to use the normals of the model. In orientable, valid triangle meshes, these normals can always be generated. This is comprehensively presented in section 4.3.2. Another hybrid approach can sometimes be applied. This approach uses both, sign calculation with normals and sign propagation with scan conversion. Section 4.3.3 explains the details of this method.

4.3.1 Sign calculation with scan conversion

Scan conversion is an old and prevalent technique that can be used for many tasks. In this case it is used for voxel classification, whether a voxel is inside or outside the solid. Scan conversion is proposed by many authors to generate the sign of distance fields. The main idea behind this technique is to cast rays along each row of voxels along the x-axis. For each voxel where the rays have crossed the surface of the solid an uneven number of times, the voxel is classified as inside.

Of course, the choice of the axis along which rays are cast does not change the result. But it is preferable to use an axis where sequential voxels are tightly located into memory to reduce the caching time, or, if the grid is big, to avoid heavy swapping of data. For huge grids the order of iteration is very time critical.

Scan conversion can be implemented using a state boolean *inside* that is initially set to $false$ for each row. For every row in the grid the algorithm marks the current voxel with the sign of the state boolean. Then the algorithm is advancing to the next voxel and determines how many intersections with the model happened. The state boolean is then swapped these many times and the new voxel is marked. This continues until all rows have been processed.

This sounds easy but the main questions are how to calculate the number of intersections between two voxels, and how to interpret situations when the ray crosses a vertex. The latter is illustrated in figure 13.

4.3.2 Sign calculation by using mesh normals

Another more obvious technique is to use the normals of the surface. If the closest point P_c on the surface for a point P in space has been found, the dot product of the normal and $P - P_c$ can be calculated. If both vectors point into the same direction, P is clearly located

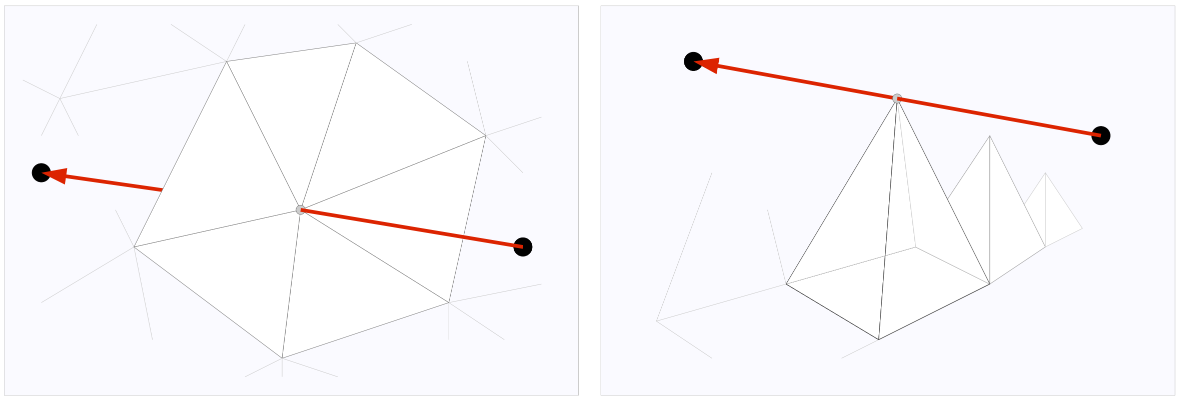


Figure 13: Scan conversion: Toggle sign (left) and don't toggle sign (right)

outside the solid, and the dot product has a positive sign. If P is located inside the solid, the dot product produces a negative sign and the vectors point in opposite directions.

The problem is that normals are not defined at vertices and edges, but these features may be to be closest to P. An easy solution would be to take the normals of the incident faces, but unfortunately this does not work because it often appears that the distance to some triangles is the same (for example if a shared vertex is the closest feature), but the normals of these triangles will not produce the same sign.

What can be done is to approximate a pseudo normal for every feature of the triangles. For faces, the normals can easily be defined by calculating $n = \frac{(b-a)\times(c-a)}{||(b-a)\times(c-a)||}$. The normal of an edge is calculated in a straightforward fashion by taking the average of both incident face normals, $n = \frac{1}{2} \cdot (n_a + n_b)$, and in fact this works for sign calculation. But what if a vertex is the closest feature?

Several vertex normals have been defined and proposed in the literature, most of them are used to accomplish smooth vertex shading of models [27]. For the sake of shading the precise direction of a vertex normal is not that much important as for sign calculation, because a sightly different normal produces only minimal variation in the lighting of the model. For voxel classification however, a sightly different normal may produce a series of erroneous classified voxels.

Two common vertex normals are presented now. Let f_i be the incident faces, and let a_i, b_i and c_i be the vertices of f_i. Let n_i be the incident face normal, calculated by $n_i = \frac{(b_i-a_i)\times(c_i-a_i)}{||(b_i-a_i)\times(c_i-a_i)||}$.

Sum of incident face normals: The normalized sum of normalized, incident face normals n_i mentioned by Gouraud [26] is the most obvious choice of a vertex normal. The normal n_s is simply calculated as follows:

$$n_s = \frac{\sum_i n_i}{||\sum_i n_i||}$$

Sum of area-weighted face normals: The normalized sum of area weighted face normal is an improvement but still very similar to the previous one. The difference is that the length of incident face normals corresponds to the area of their faces. This leads to slightly better results, especially if incident faces are subdivided. Since the cross product of two edge vectors of a triangle produces the area of the face, the vertex normal n_w is calculated with

$$n_w = \frac{\sum_i (b_i - a_i) \times (c_i - a_i)}{|| \sum_i (b_i - a_i) \times (c_i - a_i)||}$$

If normals are calculated while model data is read from file, this type of normal is as efficient to calculate as the previous one by adding the unnormalized face normal to the vertex normal instead of the normalized one.

The unpleasant truth is that tessellation variant normals, like the normals just presented, can never be used for voxel classification, because with changing tessellation these normals can come arbitrarily close to any incident face normal. See figure 14 how to change tessellation to break the first two normals n_s and n_w, thus they are unusable for voxel classification. This mainly applies to all tessellation variant normals.

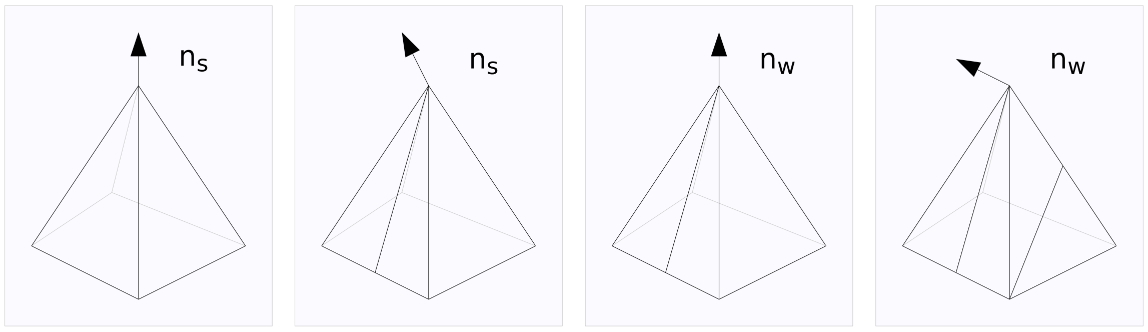


Figure 14: Tesselation changes break simple pseudo-normals

The first picture in figure 14 shows the vertex normal which has been calculated using the sum of incident face normals. The normal easily breaks if one side of the pyramid is subdivided (second picture), because this side has now two contributions to the vertex normal. If the sum of area-weighted face normals is used (third picture) the problem can be solved, because the two contributions are halved. Again, with delicate tessellation changes, it is also possible to break this normal (forth picture). *Tessellation invariant* normals are those which do not break with changing tessellation.

Largest inner product normal: For this normal we define $r := P - P_c$ as the vector from the closest point on the mesh to the voxel. Huang et al. proposes to choose the incident face normal with the largest inner product [29]:

$$n_m, \ m = \text{argmax}_i \ ||r \cdot n_i||$$

Obviously the normal is only valid for volume data generation and cannot be used for shading. Unfortunately there is a counter example that Huang's normal does not work and produces erroneous classified voxels in some cases [31].

Aanæs and Bærentzen propose the *angle weighted pseudo-normal* [27, 28] as their choice of pseudo-normal [30, 31]. In [31, 32] they prove that their choice is correct and the dot product $< r \mid n >$ is always positive if and only if the voxel is located outside the solid. (Note that r is the vector from the closest point on the mesh to P, $r = P - P_c$.) Additionally, the angle weighted pseudo-normal (*AWPN* for short) can be used for vertex shading of models as opposed to Huang's choice of normal. Also, the AWPN is easy and efficient to calculate and has simple notation.

Angle weighted pseudo-normal: For a vertex v with incident face normals n_i and angle α_i between the vertex-incident arms of the triangle, the AWPN n_α for v is calculated with:

$$n_\alpha = \frac{\sum_i \alpha_i n_i}{||\sum_i \alpha_i n_i||}$$

For the sake of voxel classification the length of n_α is not important, and we can get rid of the normalization and simply write $n_\alpha = \sum_i \alpha_i n_i$. The concept of the AWPN generalizes to face and edge normals, but they do not differ from the usual calculation. The face normal can be noted with

$$n_\alpha = \frac{2\pi n_1}{||2\pi n_1||} = \frac{n_1}{||n_1||}$$

and the edge normal with two incident faces f_1, f_2 with normals n_1, n_2 is noted in the following way:

$$n_\alpha = \frac{\pi n_1 + \pi n_2}{||\pi n_1 + \pi n_2||} = \frac{n_1 + n_2}{||n_1 + n_2||}$$

The values π and 2π stand for 180° and 360° respectively. The issue is illustrated in figure 15.

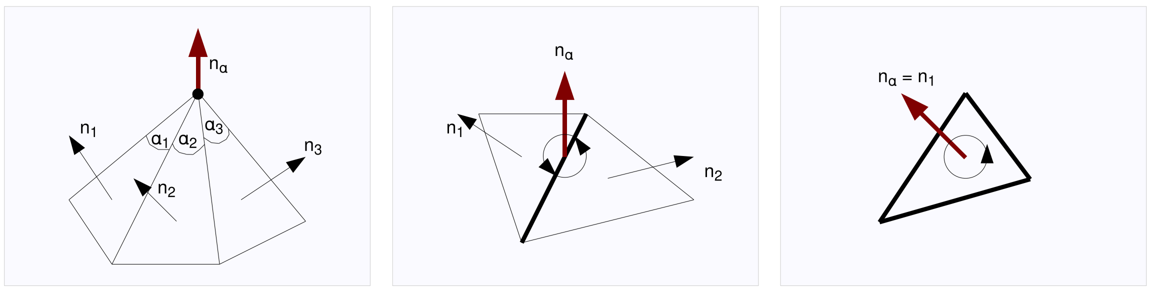


Figure 15: Calculation of the angle weighted pseudo-normal for vertex, edge and face

The advantage of using the model normals for sign calculation is that the sign can be calculated immediately after finding the closest point P_c on the mesh whereas scan conversion is typically done in a separate pass over the grid, which is quite time-consuming and labor-intensive to implement.

If one of the vertices is determined as closest feature, say it is $X \in \{A, B, C\}$, the sign of $< r \mid n > = < P - X \mid n >$ tells whether P is located inside or outside. The situation is the same if one of the edges E_i is determined as closest feature, and X is the closest point on E_i. But if the face F is the closest feature no explicit closest point P_c has been calculated so far, because the distance is already given by $\tilde{P}_x$, the x-component of the transformed grid point. Interestingly P_c does not need to be calculated. In fact it does not matter what the closest point P_c is as long as it is located somewhere on the plane that is spanned with the triangle. For example one can use $P_c = C + B - A$ or $P_c = \frac{1}{3}(A + B + C)$, but for the sake of simplicity $P_c = A$ is always used.

Since the sign calculation depends on normal information, the classification of a voxel is very sensitive about faulty normals. The most noticeable errors in the calculation of the signed distance field are those which spring from wrongly classified voxels, thus from faulty normal calculations. For example the normal between a non-2-manifold triangle constellation may be undefined (zero vector normal) or even point into a direction with no semantic meaning. Consider figure 16 that illustrates the problem.

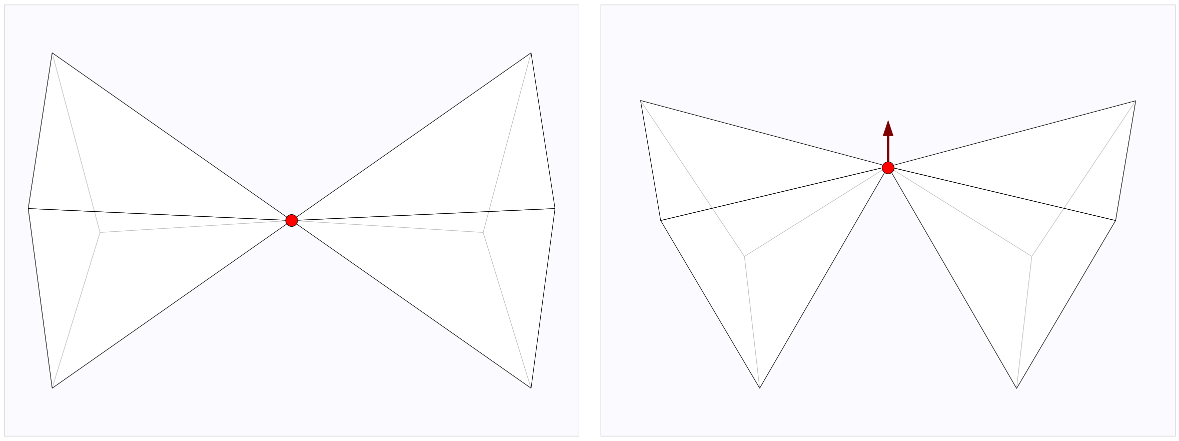


Figure 16: Undefined normal (left) and semanticless normal (right)

Surprisingly, non-2-manifold meshes are not a problem for voxel classification with normals because the involved vertex or edge between the manifolds is never used for sign calculation if the vertex normal is undefined. Another problem are zero-area faces, which either produce zero-length normals, or, even worse, NaN values which propagate in memory and may destroy the whole calculation of the distance field. This is because all operations on a NaN value lead to another NaN value. For this reason zero-area faces need to be detected and eliminated from the calculation. NaN values can be detected if the comparison of a variable x with itself does not match, e.g. if $x \neq x$ is true.

4.3.3 Hybrid technique

A hybrid technique for voxel classification uses both, classification with surface normals and scan conversion to propagate the sign. The hybrid approach inherits advantages of both methods, the simplicity and efficiency of sign determination with normals, and the easy and efficient propagation scheme of scan conversion. The difficulties of the pure scan conversion (counting intersections and interpreting vertex intersections) are omitted.

The hybrid approach can only be used if a specific part of the distance field is computed. For example this happens with distance transforms, which are described in section 5.4. To transform distances throughout the SDF, the SDF needs to be initialized with distance values in the vicinity of the surface. For these distance values the sign is also calculated. Since distance transforms only calculate new distances for uninitialized voxels, the sign is not propagated, but this can easily be done in advance.

Like with scan conversion, the algorithm works in row-order. For every row along the x-axis, a state boolean *inside* is initialized to $false$. This state boolean is assigned to the current voxel of the row if it has not been initialized yet. If the current voxel is already initialized, the sign of that voxel is assigned to the state boolean, and the algorithm advances to the next voxel. Since all voxels around the model's surface are initialized, the sign is changed in these regions, and then propagated to the subsequent, uninitialized voxels. Figure 17 illustrates this in two dimensions.

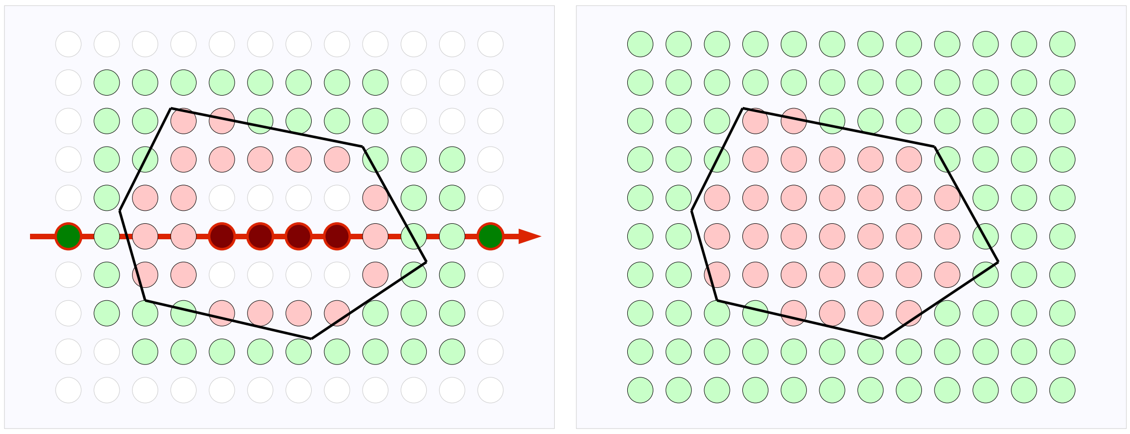


Figure 17: Sign propagation with a scan conversion variant

To calculate the distance in the shell of the surface, grid-aligned bounding boxes around each triangle are typically used, because these bounding boxes are easy to find. Then the distance and sign for each voxel inside this bounding box to the corresponding triangle is calculated.

Correct snapping of the bounding box is important as figured in 18: For the case that the minimum or maxiumum vertex of the triangle bounding box already lies excactly on a grid point, the snapping strategy needs to select the next grid point in question to prevent erroneous classified voxels (because for a zero-distance the sign is undefined). This can be done with a tricky rounding strategy.

Imagine an example: The min vertex of the triangle bounding box is $min = (2.5, 3.0, 1.3)^T$. The correct snapping would be $MIN = (2, 2, 1)^T$. Note that 3.0 does not snap to 3 in this example, and simply taking the floor value of each component does not work. Instead taking the ceil value of the component minus one does the job.

$$MIN = \begin{pmatrix} \lceil min_x \rceil - 1 \\ \lceil min_y \rceil - 1 \\ \lceil min_z \rceil - 1 \end{pmatrix} \qquad MAX = \begin{pmatrix} \lfloor max_x \rfloor + 1 \\ \lfloor max_y \rfloor + 1 \\ \lfloor max_z \rfloor + 1 \end{pmatrix}$$

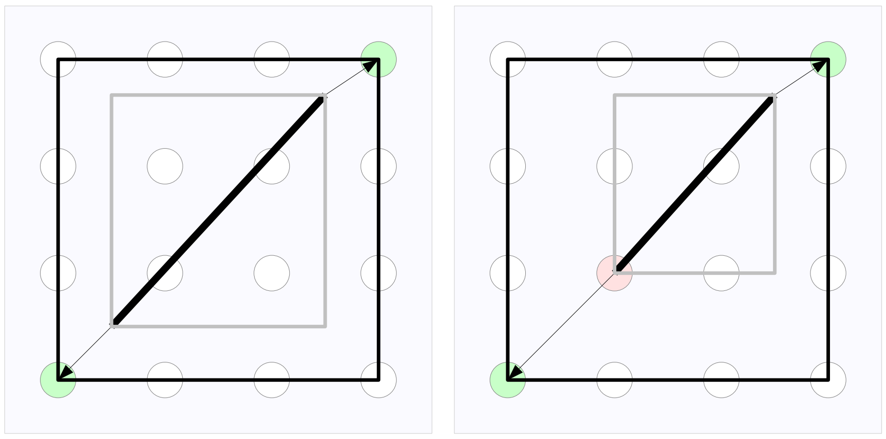


Figure 18: Grid snapping of triangle bounding boxes (in 2D)

4.4 Signed distance mapping

All techniques have been discussed for signed distance fields so far but volumetric data is expected in the end. The SDF contains distances which may be either positive or negative, whereas the volume has density values for each voxel, and these density values are always positive. The mapping from the SDF to a resulting volume is described here. Even though the mapping scheme is simple, a well considered parametrization is important to produce high quality data.

The mapping is defined per voxel, SDF and the volume have the same dimensions. Obviously, a voxel that is located inside the solid should become a high density voxel and voxels outside the solid should become lower density voxels. Voxels that have zero-distance to the surface become voxels with surface density. This is illustrated in figure 19.

Parameters to the mapping scheme are the *surface density* and the *distance interval*. The surface density is the density value that is assigned to voxels that have zero-distance in the SDF. Ideally this value is half of the maximum density value. The distance interval is the range of values (typically around zero) that is mapped to the range of available density values. Distance values outside the distance interval must be tailored since available density values are limited.

Algorithms that visualize volumes typically interpolate values between the voxels to find the desired ISO surface. This requires that enough values are available between neighbored

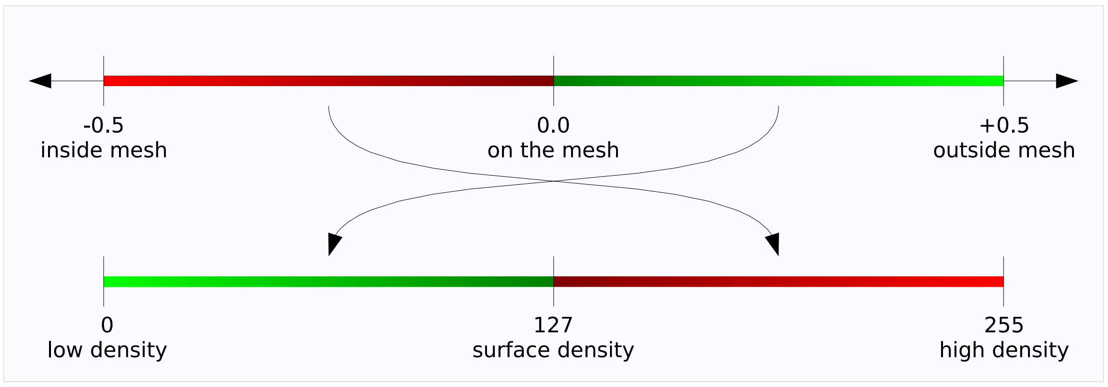


Figure 19: Mapping from signed distance values to discrete density values

voxels to produce accurate and smooth surfaces. If the distance interval was chosen too large, the density values are spaciously distributed over the volume and between neighboring voxels only few density values are located. This leads to blocky ISO surfaces especially for big, linear faces with adversarial slope. If the distance value is chosen too small, only ISO surfacing can be done because all density information is pooled around the ISO surface of the model, and ISO values different from the surface density may be very close to the surface density.

If a minimum of n density values between voxels is desired, and $t_{\max}$ is the maximum value of the data type (e.g. 256 for UCHAR), the distance interval should not be greater than $\frac{t_{\max}}{2n} \cdot \min(d_x, d_y, d_z)$ units. The distance interval should never be smaller than twice the distance between diagonal voxels, $2 \cdot \sqrt{d_x^2 + d_y^2 + d_z^2}$. Note that if d_x, d_y and d_z specify the distance *ratio* between voxels, the values do not correspond to distances. In addition if the surface density is not centered, the minimal distance interval should even be greater.

Figure 20 shows a pyramid with different mapping parameters. The first image uses a big distance range, the second one uses a much smaller one and the third one additionally uses a different rounding strategy.

The mapping needs to be defined for each data type that is used for the density values. Common data types are UCHAR and USHORT. Also FLOAT can be used, but certainly this is not a good choice for volumetric data. FLOAT should rather be used for representing SDFs directly. The mapping calculation is exemplary presented for UCHAR, but other data types are analogous.

Parameters for the calculation are di, the distance interval, and sd, the surface density. The specification converts the distance value val to a density value.

```
/* Calculate density */
int density = (int)(roundf(val * 256.0f / di) + sd);
/* Tailor to data type capabilities */
density = min(255, max(0, density));
```

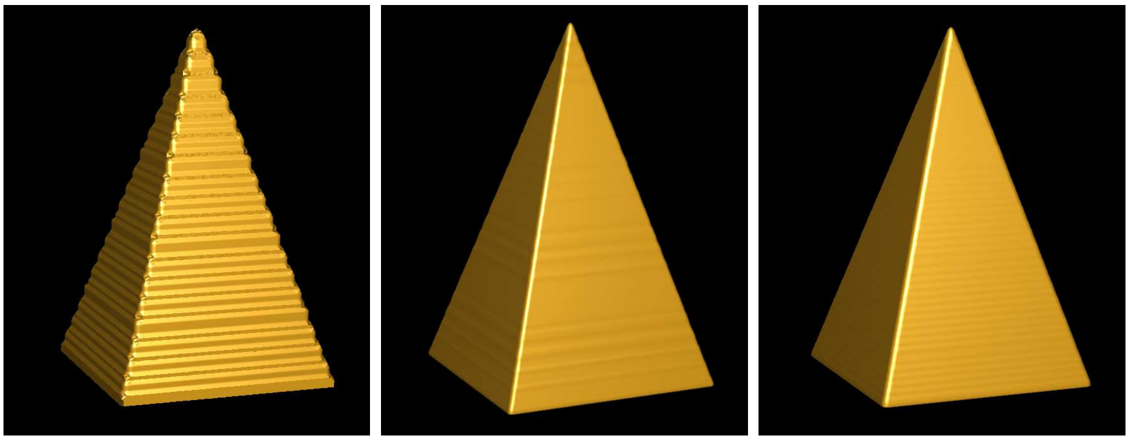


Figure 20: Distance interval [-1.0, 1.0] (left) and [-0.1, 0.1] (center, right)

5 Computation schemes for SDFs

In this section various schemes for computing complete euclidean distance fields are presented. The most obvious scheme is discussed in section 5.1, the Brute Force method. Section 5.2 tries to speed up the Brute Force with hierarchical organizations of the triangles. Another very interesting approach is to use the mesh characteristics, which is sketchily presented in section 5.3.

The term *euclidean distance fields* correlates to SDFs where every voxel has accurate euclidean distance to the solid. This is true if the Brute Force, the hierarchical methods and the mesh characteristics are used for distance calculations; these methods do not introduce errors. The distance transforms, that are discussed in section 5.4, only approximately calculate euclidean distances to the mesh, but the advantage is very short computation time.

5.1 Brute Force

The disadvantage of the Brute Force method is obvious. The computation time is very high because for every voxel it calculates the distance to all triangles. Imagine a mesh with one million triangles (like the Stanford Dragon) and a grid size of $128 \times 128 \times 128$. With a Brute Force the computer calculates the distance from every voxel (128^3 voxels) to every triangle, which makes a total of $2\,097\,152\,000\,000 \approx 2.1 \cdot 10^{12}$ distance calculations, this will approximately take 30 hours on a desktop machine. A higher resolution grid with $256 \times 256 \times 256$ voxels will take about ten days for the same model. The timings are estimations for an AMD Athlon™ 64 Processor 3700+ with *all optimizations enabled*.

The complexity of this approach clearly is $O(M \cdot N)$, with M the number of voxels and N the number of triangles of the mesh.

On the other hand the Brute Force method has also a few advantages. First, the Brute Force is a very simple method. Implementation is not that time-consuming and straight forward. Second, the Brute Force calculates an accurate euclidean distance field without errors. If high precision is required throughout the whole SDF, the Brute Force is a good method. The runtime of a naive Brute Force implementation can be drastically reduced if some optimizations are introduced.

5.1.1 Optimizations

The problem is that the computational complexity of the Brute Force is so high, and the need to accelerate the computation arises. Three areas of acceleration can be identified [34]:

- reducing the number of voxels for which the distance has to be computed,
- making the distance calculation as efficient as possible,
- reducing the number of triangles which the distance has to be computed with.

Since a complete SDF is required, it is not possible to reduce the number of voxels for which the distance has to be computed. But in order to display the ISO surface only, a complete distance field is not required, and it is enough to calculate distances in a shell around the model. This idea is further examined in section 5.4 about distance transforms.

Two optimizations are presented in order to make the distance calculation as efficient as possible. A third optimization reduces the number of triangles whose distance has to be computed. This is done by identifying and skipping triangles that cannot be closest to a given voxel.

① Choice of the major loop: A naive approach would be to iterate over all voxels, calculate the distance to each triangle and choose the smallest one. But since the distance to a triangle is computed in 2D, each triangle needs to be transformed with matrix multiplication. Even though the creation of the transformation matrix is fast, it is much more efficient to iterate over all triangles, calculate the matrix once for each triangle and then transform every voxel and calculate the distances. This prevents the transformation matrix from being calculated over and over again or rather eliminates the need for a matrix cache.

② Calculate with quadratic distances: As mentioned before in this document, it is always possible to use quadratic distances and omit the square root for each distance. Since taking the square root is a very expensive operation, this optimization speeds up the calculation for about 20%. Using squared distances works for this approach, because only distance comparisons are performed in order to find the smallest distance. If the quadratic-euclidean distance field has been calculated, square-rooting the values can be part of the mapping to volumetric data as final step.

③ **Skip triangles out of question:** Most of the triangles are far away from the current voxel whereas others are much closer. These triangles can be identified, but they cannot be skipped without further examination. The objective is to carry out a minimum of calculation upon these triangles and immediately reject it if it is out of question. The method how to do this is simple: Just after transforming the voxel to 2D space, $\tilde{P}_x$ is the minimal possible distance to the triangle. If this distance is greater or equal than the distance that has already been calculated for that voxel, the triangle can immediately be rejected. The region check and the final distance and sign computation can be omitted. This optimization gives a huge performance boost of about 30-40% and typically about $\frac{2}{3}$ of the distance calculations are early aborted. This optimization is an important one, because the main problem with the Brute Force is its global nature. This optimization makes the Brute Force a bit more local, and the higher the number of voxels and triangles, the bigger the gain.

The optimizations have been compared to each other. Instead of disabling all optimizations and comparing the timings with each of the optimizations enabled, I think it is more interesting to see how the optimizations speed up the computation with the other optimizations enabled. For this reason each of the optimizations is enabled one after another. The first optimization is not disabled, because it's more like a concept than an optimization that can easily be turned off, and recalculating the transformation matrices over and over again does not make sense. The total speed-up with all optimizations enabled is about 40-50% compared to a naive implementation.

The first series of tests was done with a remeshed version of the bunny model with 26 394 triangles, 13 199 vertices and a $64 \times 64 \times 64$ voxel grid.

Optimizations	Distance calculations	Early aborted DCs	Time
①, ②, ③	6 919 028 731	4 173 846 397	363 secs
①, ②	6 919 028 731	0	540 secs
①	6 919 028 731	0	683 secs

The next series of tests was done with a remeshed version of the dragon model with 78 219 triangles, 39 168 vertices and a $64 \times 64 \times 64$ voxel grid.

Optimizations	Distance calculations	Early aborted DCs	Time
①, ②, ③	20 504 641 536	12 043 306 166	1103 secs
①, ②	20 504 641 536	0	1607 secs
①	20 504 641 536	0	2036 secs

5.2 Hierarchical approaches

Hierarchical approaches can speed up the computation of the SDF because the hierarchy allows logarithmic access to the triangles. Thus the complexity for calculating a SDF reduces from $O(N \cdot M)$ to $O(N \cdot \log M)$ with N the number of voxels and M the number of triangles. But building these hierarchies is far from being obvious. Instead of organizing points in 3D space with regular octrees, *triangles* need to be organized and it typically happens that many of the triangles need to be part of more than one cell if an octree is used for organization.

Payne and Toga propose a basic approach for organizing and calculating distances to triangles in a hierarchy [35]. They utilize the data coherency of the mesh by storing the triangles in a tree of bounding boxes. With these bounding boxes it is possible to calculate, from a fixed point in space, the smallest and the greatest possible distance for all triangles inside a bounding box. If a box has been found with its smallest distance being greater than the greatest distance of another box, this box can be pruned with all its children.

For the sake of redistancing for set level methods, Strain uses a quadtree (in two dimensions) with distance information in the nodes [36]. This special quadtree, namely the *distance tree*, uses *concentric triples* as splitting criterion for the leafs. The correctness of his approach relies on the distances in the nodes being calculated with the max-norm distance function $|D_\infty|$, because the square tree cells are spheres in the max-norm.

Another recent approach is the Meshsweeper algorithm by Guéziec [37]. The algorithm uses a multi-resolution hierarchy of bounding volumes generated by geometric simplification of the polygonal mesh. As the mesh refines, the accuracy of the closest point increases until the true closest point is found. The refinement process can be interrupted at any time, providing an approximation to the distance with both an upper and lower bound. Also, the algorithm is dynamic and exploits coherence between subsequent queries.

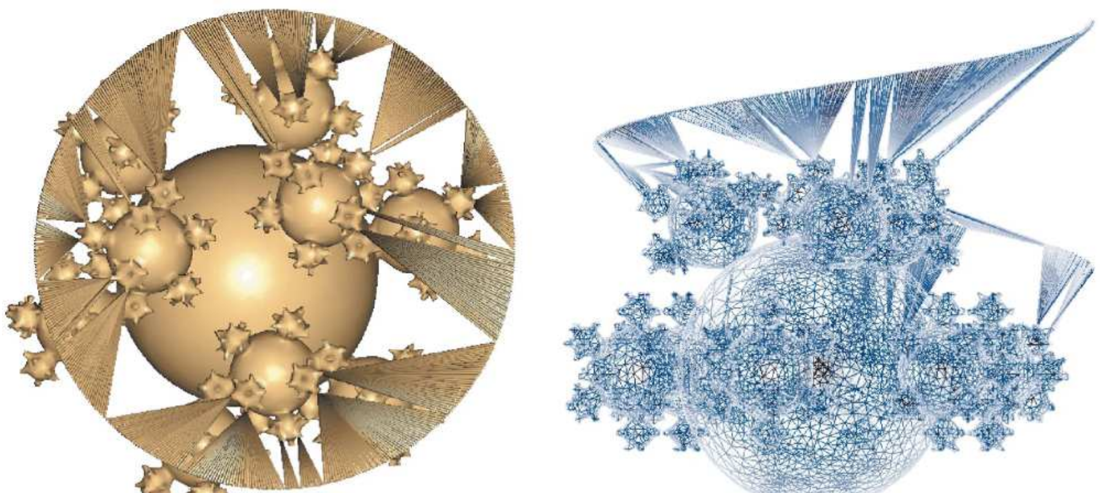


Figure 21: Meshsweeper: Distances from a curve to the mesh, taken from [37]

Generally speaking the use of hierarchies can accelerate the calculation of a complete, accurate SDF. But the implementation of such an organization is not easy and takes up a lot of memory. Additionally, initialization of the hierarchy takes some time and queries are also computationally expensive. It turns out that a hierarchy is profitable only if the mesh contains a lot of triangles. The higher the number of triangles in the hierarchy, the bigger is the gain — and also the memory needed for the hierarchy.

5.3 Exploiting mesh characteristics

Mesh characteristics are a very interesting method for calculating an accurate SDF up to a certain distance. The influence of a triangle becomes local if distances only up to a certain value d are required. Mauch et al. presented their Characteristics/Scan-Conversion (CSC) algorithm [38] that computes the signed distance field for triangle meshes up to a given distance d. Relying on the connectivity of the triangle mesh, a special kind of polyhedra serves as bounding volumes for *Voronoi cells*. These polyhedra are much easier to compute than, and are known to contain the Voronoi cells.

Faces become 3-sided prisms (”towers”) that are built up orthogonally to the faces, these towers contain the faces and extend to both sides of the mesh. Edges also become 3-sided prisms (”wedges”) filling the space between the towers. Wedges contain the edges and extend to only one side of the mesh. Vertices become n-sided cones, filling the space left by towers and wedges and also extend only to one side of the mesh. These three types are illustrated in figure 22.

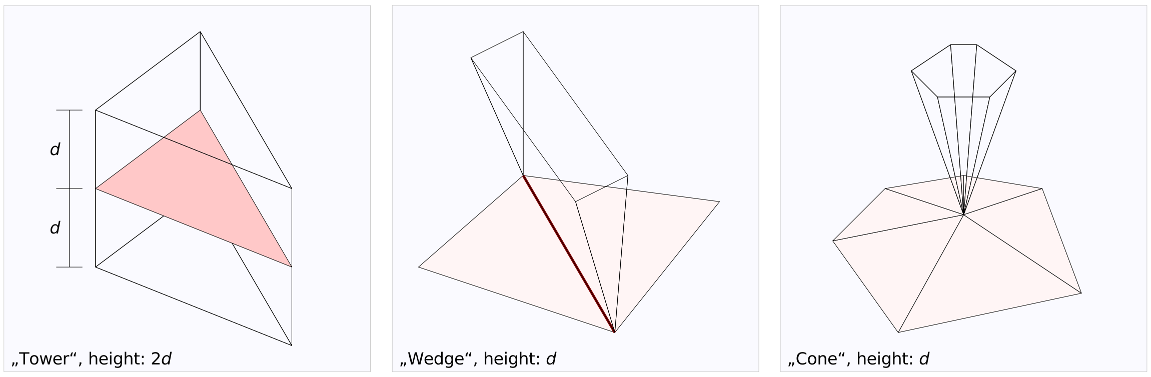


Figure 22: Tower of height $2d$ (left), wedge (center) and cone (right) of height d

Edges of the mesh can be either classified as convex, concave or planar. Accordingly, vertices can be classified as convex, concave, planar or saddle. For saddle vertices, the polyhedron is no longer a cone but has a more complex shape.

These characteristics contain all points that are closest to their respective feature up to a certain distance, and the characteristics are similar to truncated Voronoi regions. The distance from all voxels within a characteristic to the generating feature can easily be calculated then. As a bonus, the sign does not need to be calculated for every voxel, because the sign is homogeneous for each characteristic (if towers are split into an inner and outer part). Unfortunately the characteristics need to be made to overlap slightly to avoid missing grid points. In some situations overlapping characteristics do not have the same sign. The sign of the characteristic with the closer feature is used then. To find the voxels inside a characteristic, scan conversion is typically used.

Recently a characteristics scan conversion algorithm was proposed by Sigg et al. in [33]. They use the graphics hardware to do the scan conversion but slicing of the characteristics is still done on the CPU. Another approach from Sud et al. also involves hardware acceleration. They claim that their method [39] is about two orders of magnitude faster than a software implementation.

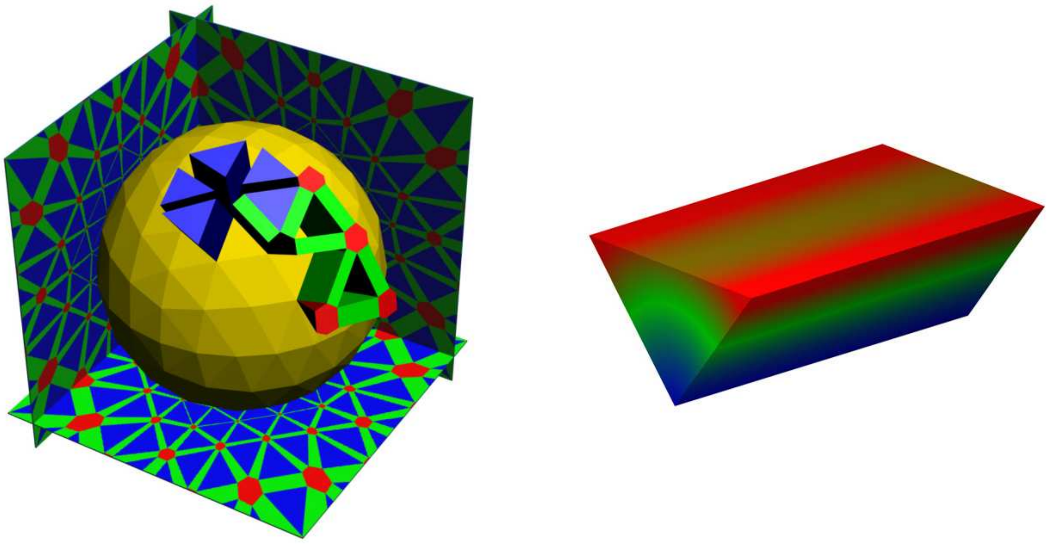


Figure 23: Generalized Voronoi cells (left) and DF within a wedge (right), taken from [33]

5.4 Distance Transforms

Distance Transforms aim to produce a complete euclidean distance field in very short time. Since distance transforms do not compute an accurate distance field, it is sometimes called a *quasi-euclidean distance field*. Typically some parts of the SDF are initialized, either with sign classification using scan conversion, or with true distances in the vicinity of the surface.

The former method using a sign classification as initialization is called *binary segmentation*. This is done in order to locate the zero-distance surface. Since only a binary decision is made at each voxel, the result is only a crude approximation of the real surface and typically looks blocky. In the latter method, distances are calculated in a shell around the surface with any of the direct methods that has been discussed. These distances are then used as initialization. Since the zero-distance surface is created with accurate distance measurement, this approach has *subvoxel accuracy*. This leads to much smoother surface reconstructions.

If p is a voxel, initialization I with the binary classification is accomplished using:

$$I(p) = \begin{cases} -\infty & p \text{ is interior} \\ +\infty & p \text{ is exterior} \end{cases}$$

If p is a voxel, subvoxel accuracy is initialized with:

$$I(p) = \begin{cases} sd_M(p) & p \text{ is in the shell} \\ -\infty & p \text{ is interior, not in shell} \\ +\infty & p \text{ is exterior, not in shell} \end{cases}$$

Typically the biggest/smallest possible value that can be stored without inconvenience is used as $+\infty/-\infty$. It is even better to separate the sign and the distance, because distance transforms are naturally working with unsigned distances. Starting from this initialization, the distance values are then propagated throughout the whole SDF with a distance transform until all voxels have reasonable distances set. Since distances far away from the surface are not calculated with a direct method, errors are introduced. These errors need to be evaluated and compared in order to make statements about the quality of the method.

Distance transforms, or *DT* for short, can be classified [46] according to how the distance of uninitialized voxels is being *estimated*, and how the distance is being *propagated* over the SDF. Techniques for distance estimation are:

- **Chamfer DTs** or *CDTs*. The new distance of a voxel is calculated from distances of neighbored voxels in a defined neighborhood plus a mask constant.
- **Vector DTs** or *VDTs*. Every voxel that has been processed contains a vector to the nearest surface point. The vectors of unprocessed voxels are calculated from voxels in a defined neighborhood plus a constant vector from a mask (*vector template*). The distance is then calculated from the stored vectors.
- **Eikonal solvers**. The distance of a voxel is computed by a first or second order estimator from the distances of its neighbors. This type is not further discussed.

CDTs are fast but suffer from poor accuracy. VDTs take much more memory but errors are marginal. Schemes for propagating the distance are:

- **Sweeping scheme**. The propagation starts in one corner of the SDF and ends in the opposite corner, typically as simple as a loop over the data. Several passes are required, typically one or more forward and backward passes.
- **Wavefront scheme**. The propagation starts at the zero-distance surface of the SDF and proceeds towards all directions with increasing distance until all voxels are processed.

Implementation effort is much bigger in wavefront schemes, but wavefront schemes can easily be interrupted if the desired maximum distance has been reached.

5.4.1 Chamfer DTs

Chamfer distance transforms were first introduced by Borgefors in [40]. Advantages of the CDT are that the computation of the quasi-euclidean DF is very fast and takes no additional memory because the CDT directly works on the SDF data. The algorithm works as follows:

For every new voxel that has been selected using one of the propagation schemes, a mask is centered over that voxel. Then the smallest distance of all voxels covered by the mask plus their corresponding mask constant is used as the new distance value. If no neighboring voxel plus its mask constant is smaller than the current voxel, no change is made because the mask constant for the current voxel is zero. Computation with quadratic distances is not possible because addition is carried out upon the distances.

For example very simple masks are the $3 \times 3 \times 3$ city-block and chessboard masks, more accurate masks are the quasi-euclidean and the complete $3 \times 3 \times 3$ masks, illustrated in figure 24. The green mask components are used in forward pass, and the yellow ones are used in the backward pass.

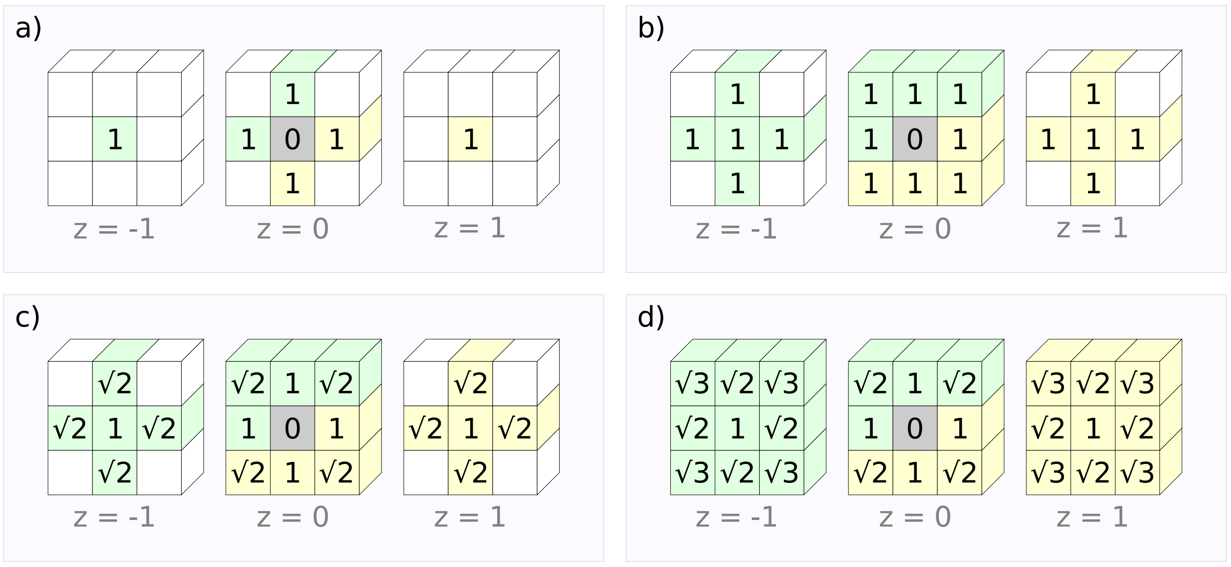


Figure 24: CDT Masks: (a) City-block; (b) Chessboard; (c) Quasi-euclidean; (d) Complete

In [40] Borgefors originally proposed a distance transform in two dimensions with integer values only and distances 3 and 4 are used instead of 1 and $\sqrt{2}$ as approximation for mask c and $z = 0$ in figure 24. Later in [41] Borgefors introduced a 5×5 matrix for distance estimation. A complete $5 \times 5 \times 5$ matrix in three dimensions is illustrated in figure 24. Some elements are left blank in order to prevent recalculations for already calculated distances.

The main problem with CDTs is that they suffer from poor accuracy. E.g. with the city-block mask, a far away voxel has (quasi) the same distance as accurately calculated

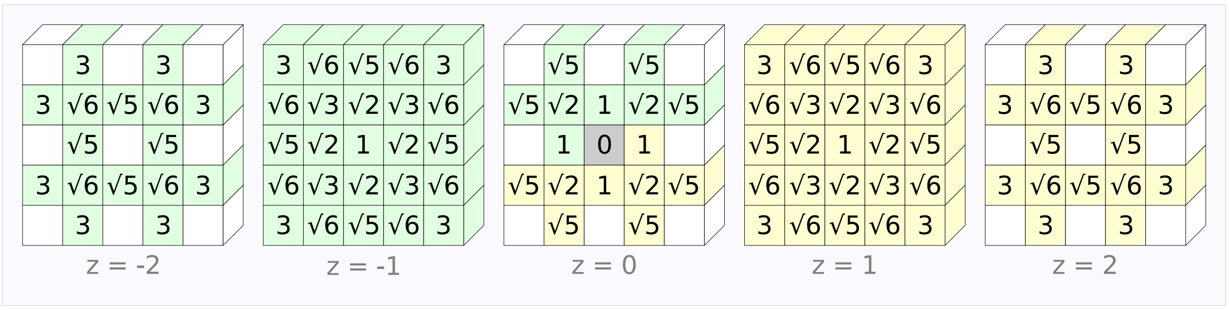


Figure 25: Complete $5 \times 5 \times 5$ CDT mask

with the $|| \cdot ||_\infty$-norm. This is often not acceptable and the complete $3 \times 3 \times 3$ masks delivers much better results. The distance of a far away voxel calculated with a complete mask is the length of the shortest path that is constructed out of horizontal, vertical and diagonal elements between the voxels. This path is always longer than the real euclidean distance, and CDTs mainly produce positive errors. This is called *overestimation*. With the complete $5 \times 5 \times 5$ mask, also diagonal elements over two voxels are accepted. This leads to better approximations, and in fact: The bigger the mask the better the approximation. But masks bigger than $5 \times 5 \times 5$ lead to very long computation times, the time constant correlate to the number of elements in the mask ($3^3 = 27$; $5^3 = 125$; $7^3 = 343$; ...).

In [42] Svensson and Borgefors introduced a weighted CDT in order to reduce overestimation of the distances. This is done by assigning different weights to the steps, depending on the neighborhood relation between the voxels in the path. Figure 35 shows the errors that happen under rotation with a $3 \times 3 \times 3$ and a $5 \times 5 \times 5$ mask.

5.4.2 Vector DTs

The Vector DTs are similar to the Chamfer DTs. But in contrast to CDTs for each distance that is calculated in the shell of the surface, a vector pointing to the closest point on the surface is also stored. Unprocessed voxels are then calculated from neighboring voxels plus a constant vector from the template, instead of using distances directly. The distance of a voxel is calculated from the vector. During the transform it is possible and much more efficient to use quadratic distances, as only comparisons are carried out.

Vector distance transforms were first proposed by Danielsson in 1980, two years before the CDT was introduced by Borgefors. In his paper [43] he proposed the four-point sequential Euclidean distance mapping algorithm, 4SED, and the more accurate but slower eight-point sequential Euclidean distance mapping, 8SED. The numeral denotes the number of neighbors used in a 3×3 matrix. Further he pointed out the worst case errors for 4SED and 8SED. Many papers and algorithms based on Danielsson's work have been published with the aim to reduce the errors.

Ye [47] and Leymarie and Levine [48] extended the algorithm to use signed vectors, the SSED algorithms. 1992 Mullikin extended the SSED algorithm to three dimensions developing the efficient vector distance transform (EVDT) [44], one of the most accurate VDTs in three dimensions with six passes reported in literature. It is shown that it is not possible to use less than four inspections for 3D images with a sweeping scheme [49]. Breen et al. developed a wavefront version of a VDT [50, 51].

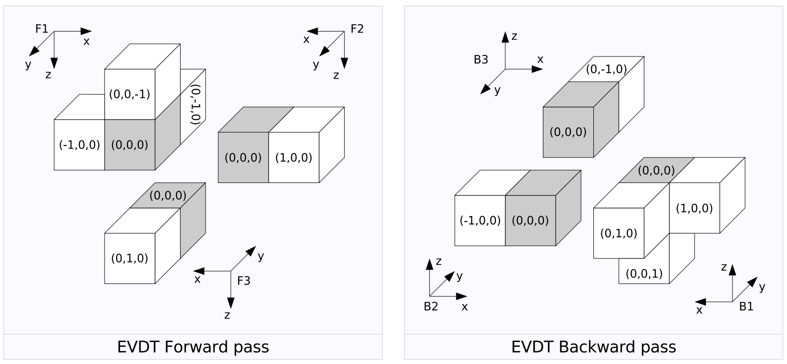


Figure 26: Forward and backward pass of Mullikin's EVDT

Initially Mullikin developed EVDT to maintain only two slices, the current and the previous slice of the vector field, to reduce the memory requirements. But an alternative implementation with a full vector grid has shown that using a limited number of vector slices introduces a large number of errors compared to those implementations that use a full vector grid. This is because during the backward pass the wrong feature voxels are selected as closest. The average error of a voxel is 0.48% in contrast to 0.08% with a full vector grid implementation [45]. (This comparison is based on the UNC CThead data set.)

Satherley and Jones developed the vector-city vector distance transform (VCVDT) in 2001. Their approach [45] exists in a four-pass and an eight-pass variant. They claim that their four-pass variant outperforms Mullikin's EVDT with full vector grid in both, execution time (about 50% faster) and accuracy (about 10% smaller average error per voxel). The eight-pass variant takes about twice the time of the four-pass variant to execute and the gain in accuracy leads to 8% smaller average error per voxel.

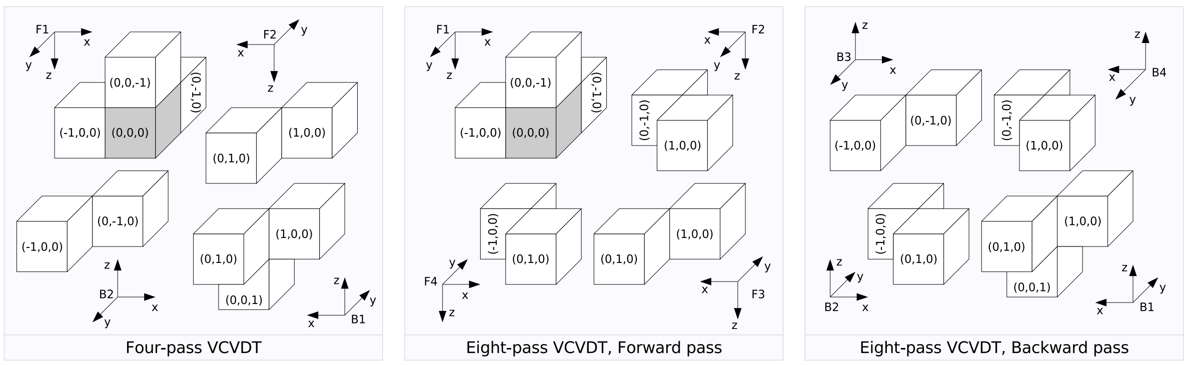

Four-pass VCVDT
Eight-pass VCVDT, Forward pass
Eight-pass VCVDT, Backward pass
F1
F2
F3
F4
B1
B2
B3
B4

Figure 27: Four-pass and eight-pass templates of the VCVDT

Since the sweeping scheme of vector based DTs is not that obvious, it is briefly explained here: First, the coordinate system is a bit different from the system that is used in OpenGL. The origin of the system is located in the upper left back corner. The x-axis runs to the right, the y-axis runs to the front, the z-axis runs to the bottom. This is illustrated in figure 28. Then for each pass, all masks are first applied to every z-level-plane. For example the mask $F1$ is applied to the z-level-plane by iterating over the row along x, then proceeding to the next row by incrementing the y-coordinate. If $F1$ has been applied to the z-plane, $F2$ is applied then. $F2$ is applied from the opposite corner of the plane as indicated by the three arrows near the mask in figure 27. Then the algorithm proceeds with the next z-plane. The backward pass is applied similarly, beginning with the last z-level-plane proceeding to the first.

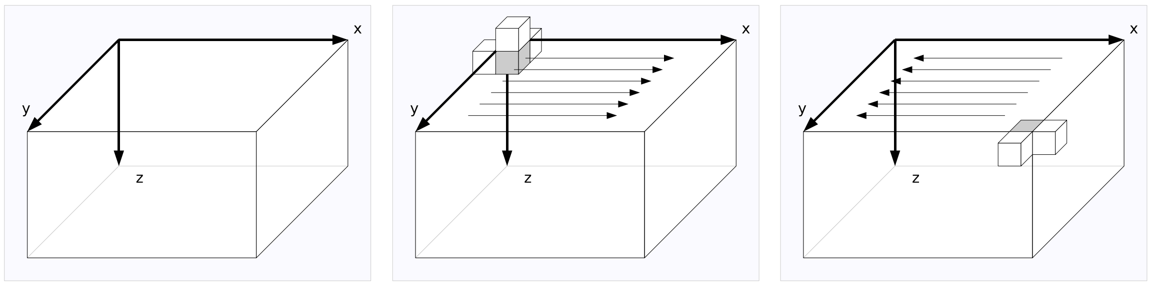


Figure 28: VDT sweeping scheme

5.4.3 Comparisons

In this section the different distance transforms are compared to each other and to an accurate data set generated with a Brute Force implementation. The methods that are compared are the (four-pass) Vector-City Vector DT (VCVDT), the Efficient Vector DT (EVDT) with full vector grid and the Chamfer DT with complete $3 \times 3 \times 3$, $5 \times 5 \times 5$ and $7 \times 7 \times 7$ masks. Three different data sets have been used for comparison, created from a sphere model, the Stanford Bunny and the Stanford Dragon. The models are pictured in figure 29.

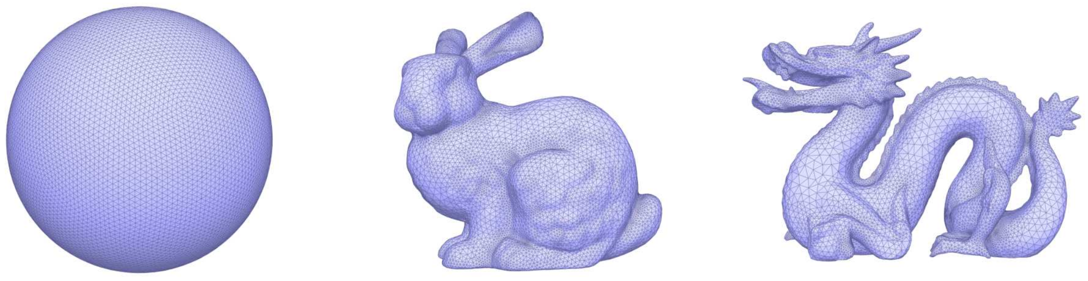


Figure 29: Meshes used for data set generation: Sphere, Bunny and Dragon

Table 1 presents the comparison for the sphere data set. The sphere model contains 20 480 faces and the generation of this data set took 4017 seconds with the Brute Force method. The Vector DTs took less than a second, the Chamfer DTs took a few seconds, especially those with bigger masks.

	Distance range		Error range		Avg. error
Method	min	max	min	max	per voxel
Brute Force	-0.141537	0.712578	-0.00000000	0.00000000	0.0000000000
CDT $3 \times 3 \times 3$	-0.141567	0.712578	-0.00871976	0.03681710	0.0024885700
CDT $5 \times 5 \times 5$	-0.141567	0.712578	-0.00405547	0.01103570	0.0007510680
CDT $7 \times 7 \times 7$	-0.141567	0.712578	-0.00227544	0.00512920	0.0003037370
EVDT	-0.141565	0.712675	-0.00242539	0.00257551	0.0000450559
VCVDT	-0.141565	0.712675	-0.00242539	0.00257551	0.0000443075

Table 1: Comparison of DTs. Dataset: Sphere SDF, $128 \times 128 \times 128$

Table 2 presents the comparison for the bunny data set. The bunny model contains 26 394 faces and the generation of this data set took 5012 seconds with the Brute Force method.

	Distance range		Error range		Avg. error
Method	min	max	min	max	per voxel
Brute Force	-0.0235327	0.813534	-0.000000000	0.00000000	0.0000000000
CDT $3 \times 3 \times 3$	-0.0244095	0.889875	-0.000988752	0.08612260	0.0072726100
CDT $5 \times 5 \times 5$	-0.0244095	0.831956	-0.000988752	0.03178610	0.0020146300
CDT $7 \times 7 \times 7$	-0.0244095	0.817409	-0.000988752	0.01491010	0.0008404240
EVDT	-0.0239957	0.813534	-0.001412430	0.00240487	0.0000493570
VCVDT	-0.0239957	0.813534	-0.001412430	0.00240487	0.0000495798

Table 2: Comparison of DTs. Dataset: Bunny SDF, $128 \times 128 \times 128$

Table 3 presents the comparison for the dragon data set. The dragon model contains 67 980 faces and the generation of this data set took 12012 seconds with the Brute Force method.

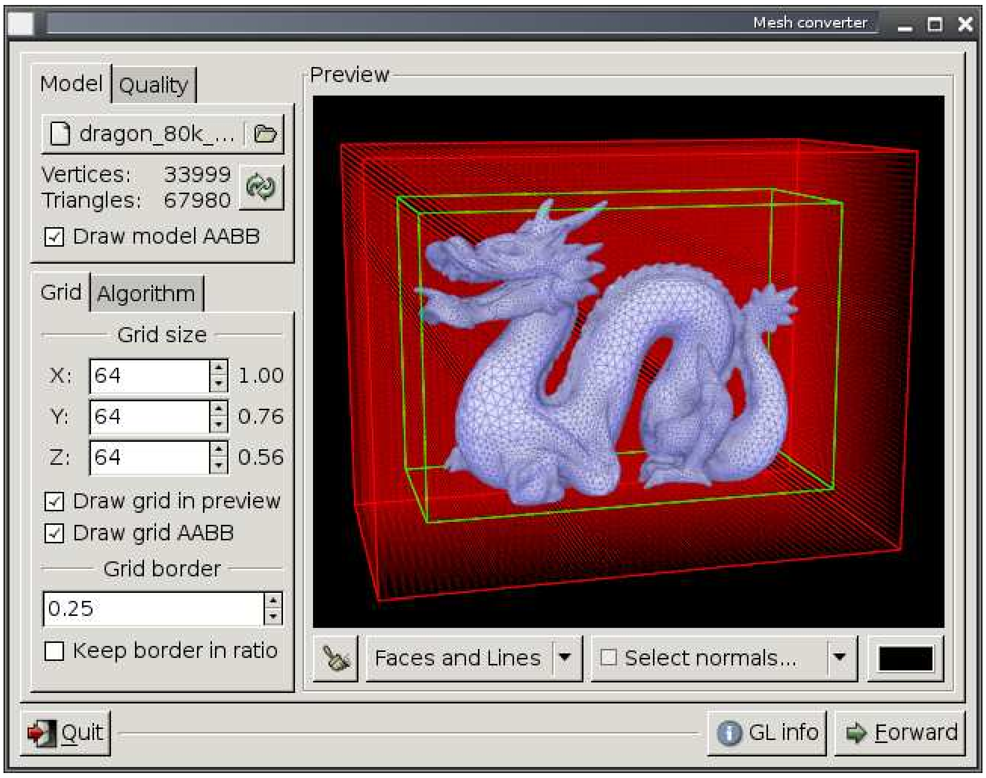
	Distance range		Error range		Avg. error
Method	min	max	min	max	per voxel
Brute Force	-0.00549662	0.635325	-0.00000000	0.00000000	0.0000000000
CDT $3 \times 3 \times 3$	-0.00549662	0.709336	-0.00000000	0.09250550	0.0071269400
CDT $5 \times 5 \times 5$	-0.00549662	0.656531	-0.00000000	0.03559130	0.0022700200
CDT $7 \times 7 \times 7$	-0.00549662	0.640696	-0.00000000	0.01744320	0.0009754010
EVDT	-0.00549662	0.635674	-0.00000012	0.00190179	0.0000403534
VCVDT	-0.00549662	0.635674	-0.00000012	0.00190179	0.0000403429

Table 3: Comparison of DTs. Dataset: Dragon SDF, $128 \times 128 \times 128$

6 A tool for volume generation

This section gives a very short overview of a tool for volume data generation. This tool was created as a result of surveying calculation techniques for SDFs and volumetric data for this bachelor thesis. It can be downloaded from my homepage[3] at Darmstadt University of Technology.

An arbitrary mesh in OFF model format can be loaded from file, models in other formats can be converted, e.g. with MeshLab. Also the triangle quality, the average valence of vertices and the number of unreferenced vertices in the chosen mesh can be calculated. The model viewer can display the model interactively with various visual settings.



dragon_80k_...

Vertices: 33999
Triangles: 67980

☑ Draw model AABB

Grid size

X: 64 1.00
Y: 64 0.76
Z: 64 0.56

☑ Draw grid in preview
☑ Draw grid AABB

Grid border

0.25

☐ Keep border in ratio

Figure 30: The tool for volume data generation: Volume customization

The resulting volumes generated with the tool are customizable. The tool offers possibilities to choose the grid size (dimension of the grid) and the border around the mesh. This border should probably be chosen according to the distance range which is described below.

The tool can generate the SDF with different algorithms. One of the algorithms is a Brute Force approach. This is a relatively simple algorithm that requires the following amount of memory to execute, where *len* is the size of the grid, *tsize* is the size of the chosen data

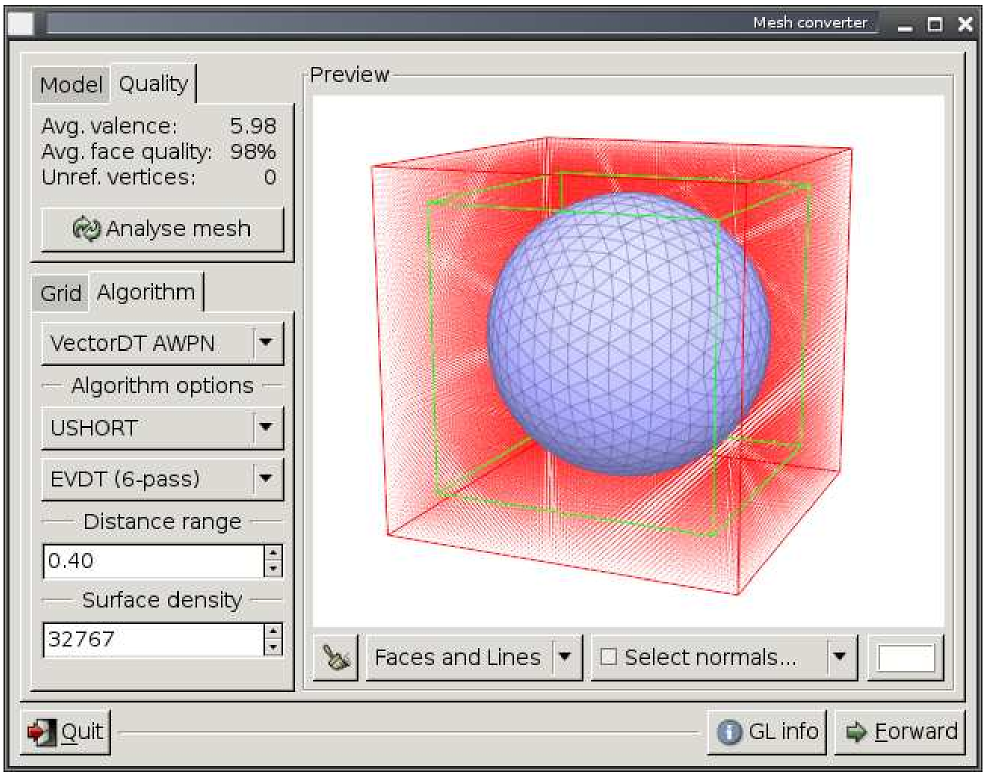
[3]http://www.student.informatik.tu-darmstadt.de/~sf1017/

type and $msize$ is the size of the model:

$$mem_{BF} = msize + len \cdot (4 + tsize) \text{ bytes}$$

The implemented Brute Force method has all optimizations enabled that are discussed in section 5.1.1.

The tool also supports Distance Transforms, namely CDTs and VDTs. The Champfer DTs exist in three mask sizes, $3 \times 3 \times 3$, $5 \times 5 \times 5$ and $7 \times 7 \times 7$ masks. The memory requirements are the same as for the Brute Force because the CDTs directly work on the SDF data.



Avg. valence: 5.98
Avg. face quality: 98%
Unref. vertices: 0
VectorDT AWPN
USHORT
EVDT (6-pass)
Distance range: 0.40
Surface density: 32767
Faces and Lines

Figure 31: The tool for volume data generation: Algorithm selection

Two Vector DTs with high accuracy has been implemented. First, there is the Efficient VDT (EVDT) from Mullikin and the Vector-City VDT (VCVDT) from Satherley and Jones. The memory requirements are relatively high because float vectors are propagated.Anyway, data sets up to $256 \times 256 \times 256$ should even work on cheap, up to date desktop PCs.

$$mem_{VDT} = msize + len \cdot (4 + 12 + tsize) \text{ bytes}$$

The tool is capable of generating data sets in different formats. These formats are UCHAR, USHORT and FLOAT. The FLOAT data type is a bit different from volume data, it represents the SDF directly. Thus the resulting data set also contains negative values for voxels inside the solid. The tool can easily be extended with additional data types if necessary.

Another very important setting is the distance range that heavily affects the quality of the resulting data set, especially for small data types like UCHAR. The distance range defines the range of distances that are mapped to the destination data type, as illustrated in figure 19 and 32. The smaller the range, the better the quality of ISO surfaces, but the smaller the range that a model can shrink or grow in size. More detailed information can be found in section 4.4. The border around the mesh should be adapted in order to prevent the ISO surface from being capped by the margins of the grid if the volume is rendered with a very low ISO value. If this is an issue, the grid border should have at least the same size as the distance range.

Yet another setting is the surface density value. Changing this value should be done with care. Typically the surface density is located in the middle of the data type interval, e.g. 127 for UCHAR, but this setting can be adjusted for special purposes such as morphing. If the surface density is located in the middle of the interval, the distance range dr extends to both sides of the model surface with $\frac{dr}{2}$. If the surface density is changed, the distance range does not extend equally to both sides.

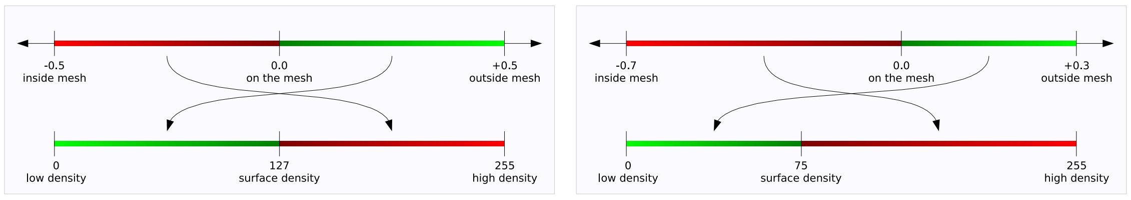
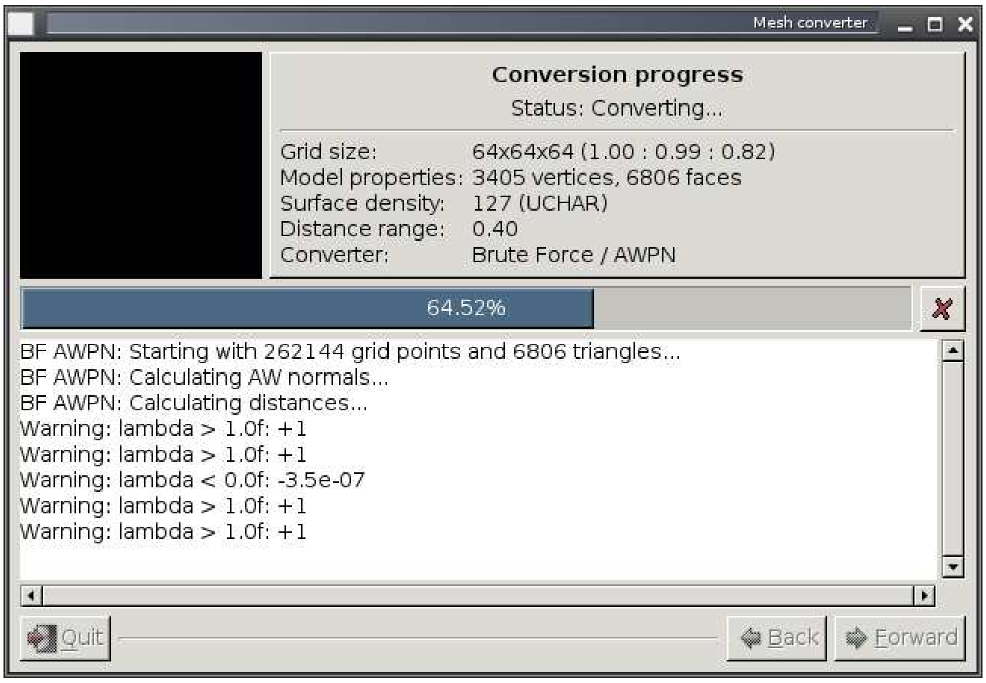


Figure 32: Two mappings, $dr = 1$ (left and right), surface density changed (right)

After customizing the desired volume, the conversion progress can be started. Depending on the model, the grid size and the algorithm, this may take from a second up to several hours. Typically VDTs are preferable and the conversion should be very fast. Note that CDTs are not that fast because complete masks are used, especially if big neighborhoods are considered. The Brute Force should be avoided as long as the mesh and the grid are small. The conversion progress can be canceled at any time. The progress is pictured in figure 33.

After the conversion is completed, the volume is visualized with a very simple method which draws a point for each voxel greater than a desired ISO value. More important, the volume can be saved to file. The format is simply RAW, all values are linearly stored in binary. Because the program stores binary values, problems with big endian and little endian encodings of the platform system may occur. Additionaly, a header file is created. The specification of this header file is proposed in section 3.1.



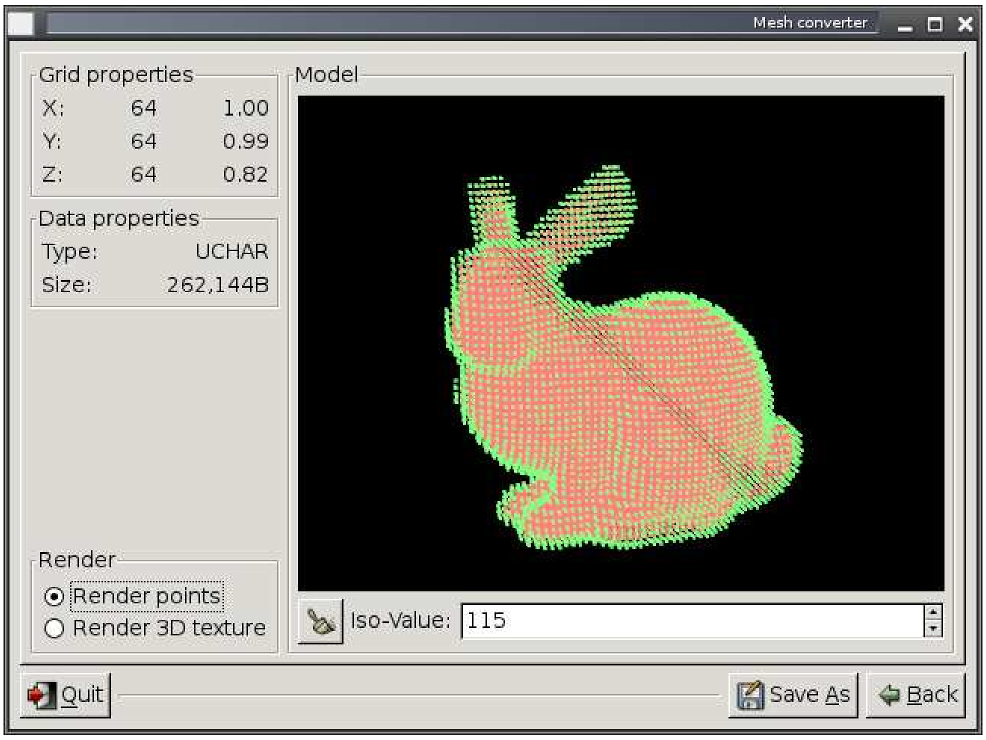
Conversion progress
Status: Converting...

Grid size: 64x64x64 (1.00 : 0.99 : 0.82)
Model properties: 3405 vertices, 6806 faces
Surface density: 127 (UCHAR)
Distance range: 0.40
Converter: Brute Force / AWPN

64.52%

```
BF AWPN: Starting with 262144 grid points and 6806 triangles...
BF AWPN: Calculating AW normals...
BF AWPN: Calculating distances...
Warning: lambda > 1.0f: +1
Warning: lambda > 1.0f: +1
Warning: lambda < 0.0f: -3.5e-07
Warning: lambda > 1.0f: +1
Warning: lambda > 1.0f: +1
```

Figure 33: The tool for volume data generation: Conversion progress



Grid properties

X:	64	1.00
Y:	64	0.99
Z:	64	0.82

Data properties

Type:	UCHAR
Size:	262,144B

Render
[✓ Render points]
Render 3D texture

Iso-Value: 115

Figure 34: The tool for volume data generation: Result ready to save

7 Results

The images in this sections are visualized with Thomas Kalbe's tool for volume visualization. The volumes are ISO-surfaced with techniques described in [22, 23] to create a smooth approximation of the surface.

The following volumes was generated with the Brute Force method and with complete Chamfer masks of size $3 \times 3 \times 3$, $5 \times 5 \times 5$ and $7 \times 7 \times 7$. The volume was ISO-surfaced at a very low density level to see the errors that accure with the Chamfer masks. The grid contains $128 \times 128 \times 128$ voxels with USHORT values.

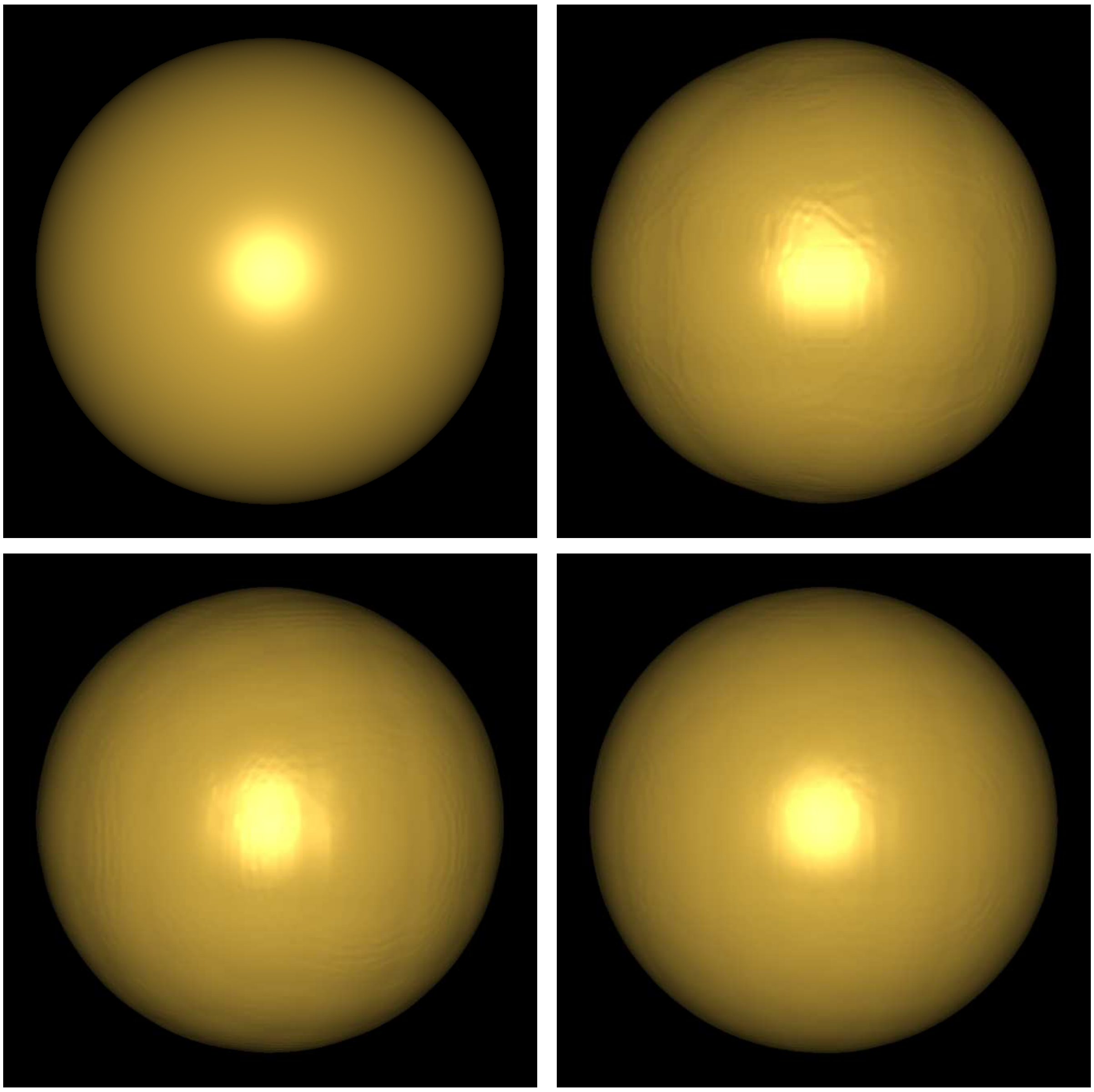


Figure 35: Distant ISO surfaces: Brute Force, CDT $3 \times 3 \times 3$, $5 \times 5 \times 5$, $7 \times 7 \times 7$

The next sets of images show the Stanford Bunny and the Stanford Dragon. The volumes are created with the Vector-city Vector Distance Transform on a grid with $128 \times 128 \times 128$ voxels and USHORT values.

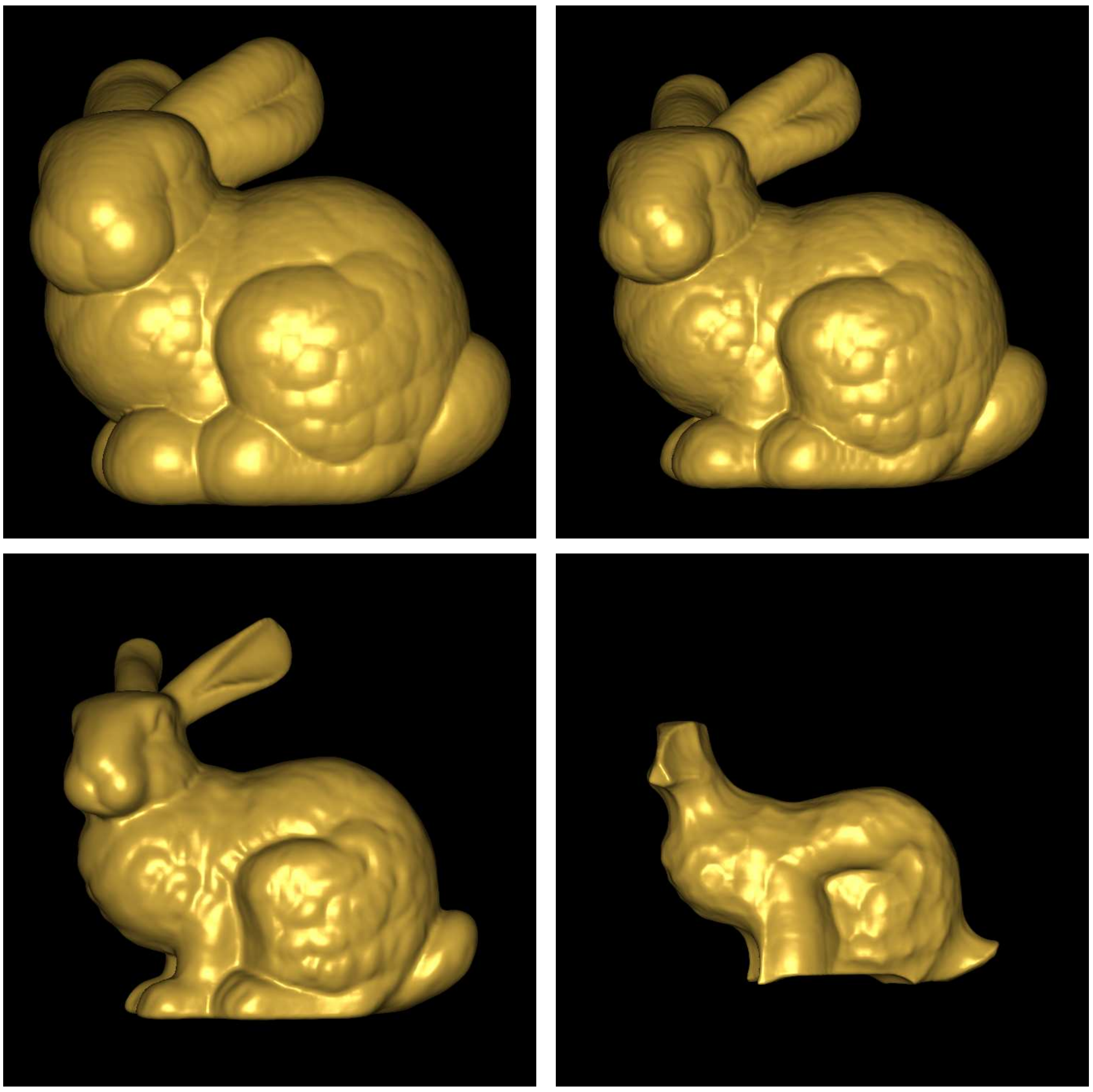


Figure 36: The Stanford Bunny at ISO values 6400, 19200, 32768 and 56320

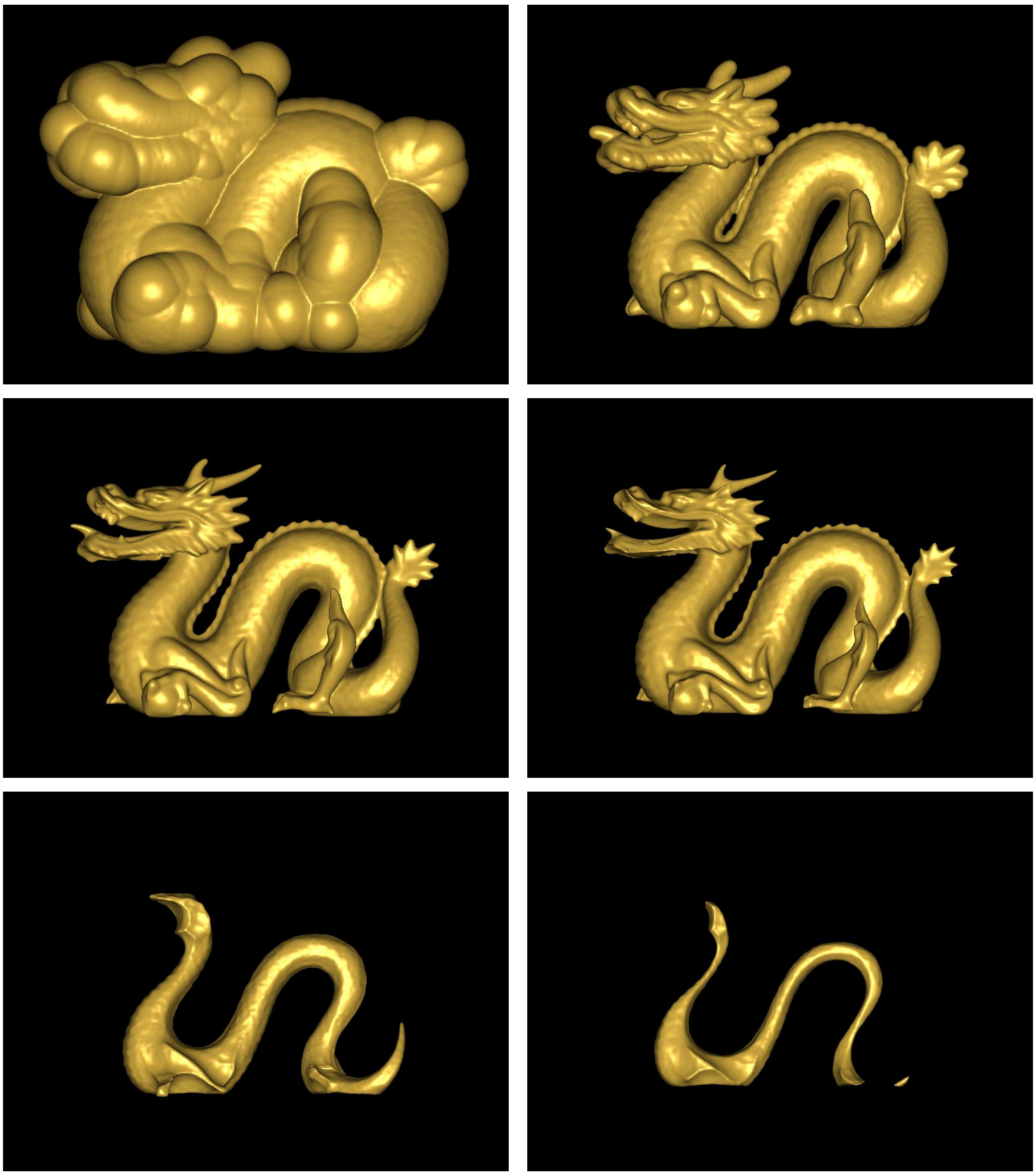


Figure 37: The Stanford Dragon at ISO 6400, 28160, 32768, 35560, 44800 and 51200

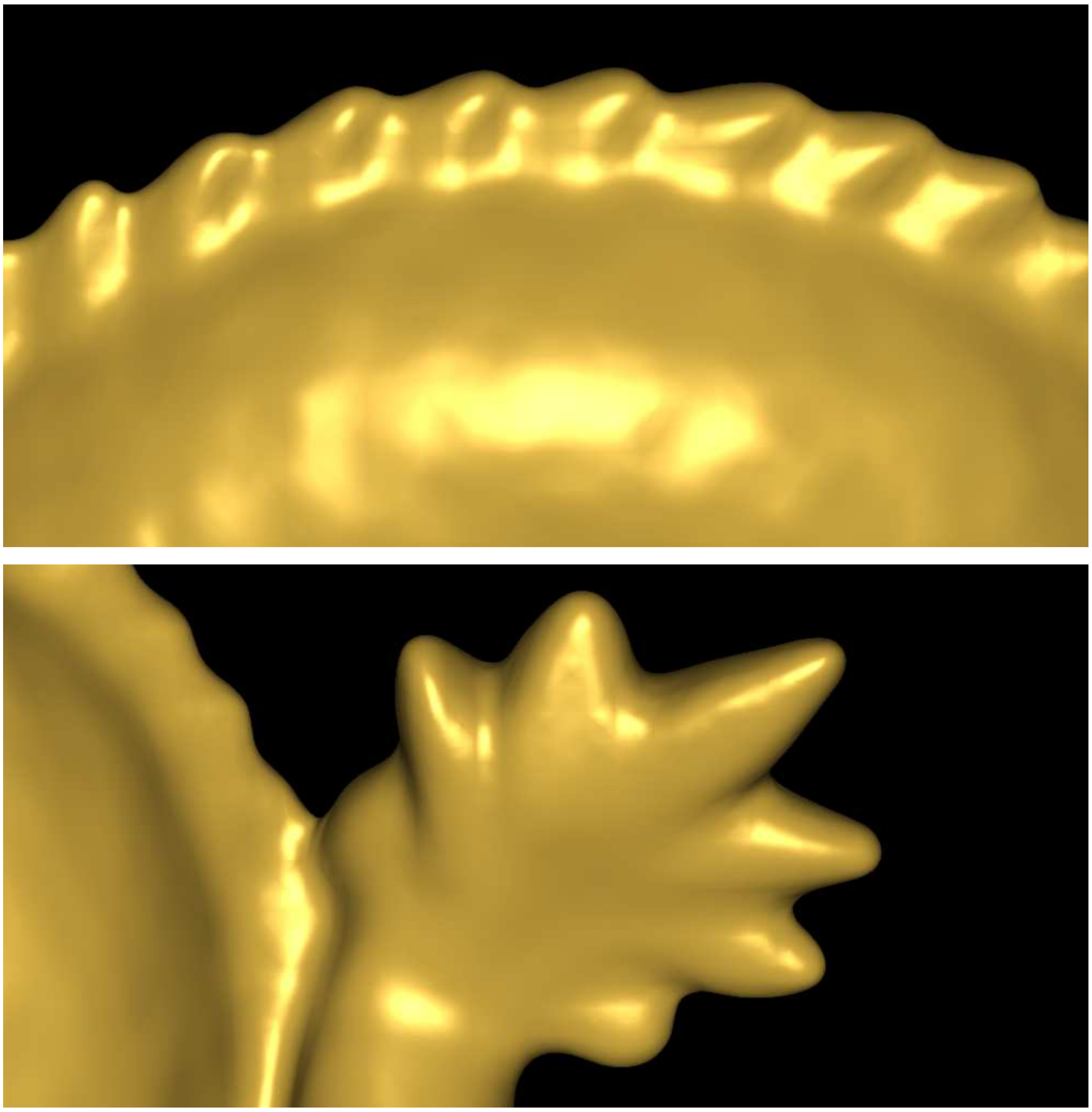


Figure 38: The Stanford Dragon's back and tail with very smooth curvature

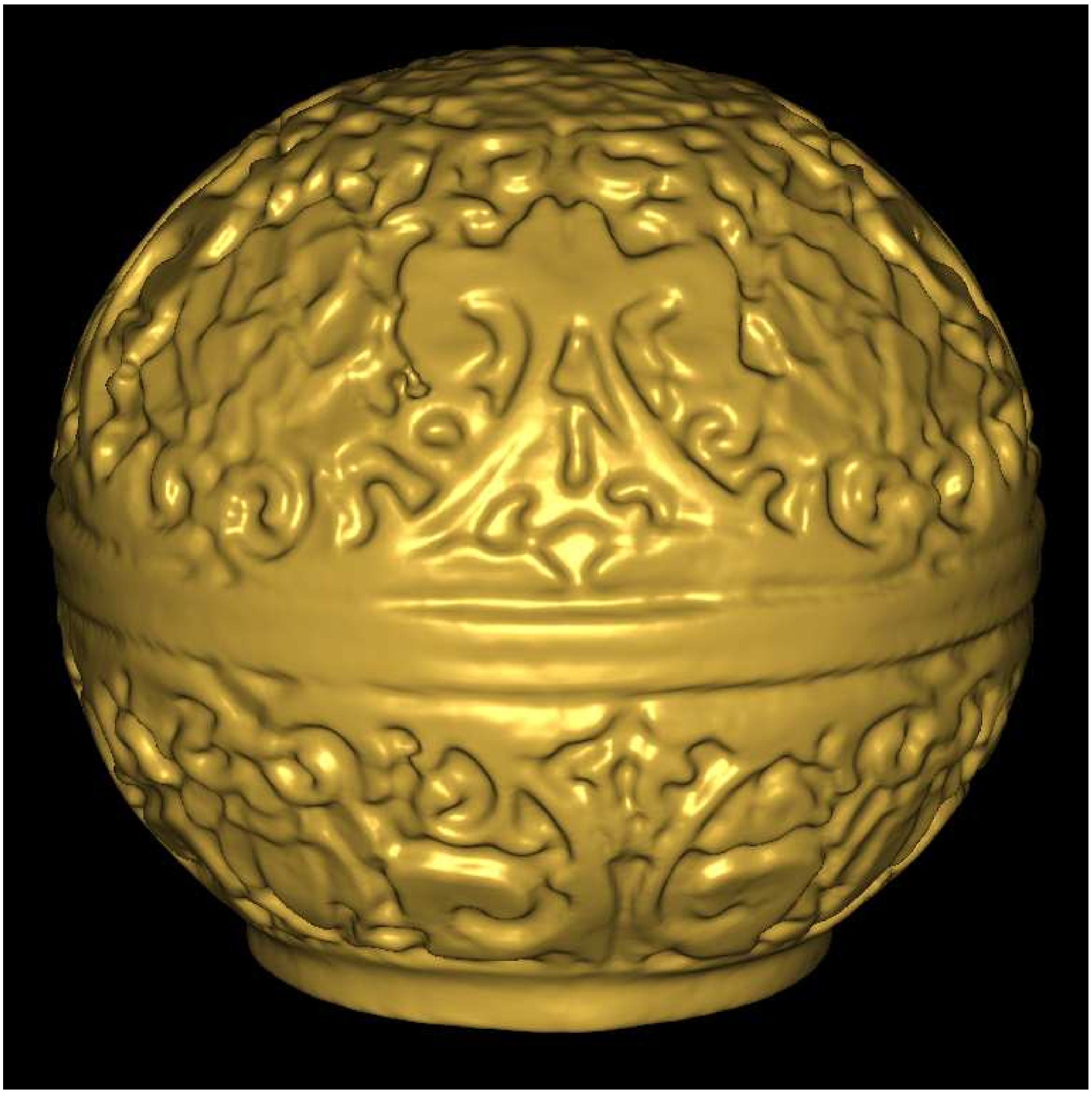


Figure 39: A model called "Red Spherical Box" from http://shapes.aimatshape.net

List of Figures

References

[1] Lena Petrovic, Mark Henne and John Anderson. ”Volumetric Methods for Simulation and Rendering of Hair”. In *Pixar Technical Memo #06-08*, Pixar Animation Studios.

[2] R. Bridson, S. Marino and R. Fedkiw. ”Simulation of Clothing with Folds and Wrinkles”. In *Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation (SCA)*, pages 28-36, 2003.

[3] Ye Zhao, Xiaoming Wei, Zhe Fan, Arie Kaufman and Hong Qin. ”Voxels on Fire”. In *Proceedings of IEEE Visualization 2003*, October 19-24, Seattle, Washington, USA, 2003.

[4] G. Hirota, S. Fisher and A. State. ”An Improved Finite Element Contact Model for Anatomical Simulations”. In *The Visual Computer*, volume 19/5, pages 291-309, 2003.

[5] E. Guendelman, R. Bridson and R.P. Fedkiw. ”Nonconvex Rigid Bodies with Stacking”. In *ACM Trans. Graphics*, volume 22/3, pages 871-878, 2003.

[6] Alexander Hornung and Leif Kobbelt. ”Robust Reconstruction of Watertight 3D Models from Non-uniformly Sampled Point Clouds Without Normal Information”. In *Eurographics Symposium on Geometry Processing*, 2006.

[7] Brian Curless and Marc Levoy. ”A Volumetric Method for Building Complex Models from Range Images”. In *Proc. ACM SIGGRAPH*, 1996.

[8] I. Bitter, A.E. Kaufman and M. Sato. ”Penalized-Distance Volumetric Skeleton Algorithm”. In *IEEE Trans. Visualization and Computer Graphics*, volume 7/3, pages 195-206, 2001.

[9] O. Cuisenaire and B. Macq. ”Applications of the Region Growing Euclidean Distance Transform: Anisotropy and Skeletons”. In *Proc. ICIP ’97 VOLI*, pages 200-203, 2000.

[10] G. Malandain and S. Fernandez-Vidal. ”Euclidean Skeletons”. In *Image and Vision Computing*, volume 16/5, pages 317-327, 1998.

[11] U. Montanari. ”A Method for Obtaining Skeletons Using a Quasi-Euclidean Distance”. In *J. ACM*, volume 15/4, pages 600-624, 1968.

[12] Y. Zhou and A.W. Toga. ”Efficient Skeletonization of Volumetric Objects”. In *IEEE Trans. Visualization and Computer Graphics*, volume 5/3, pages 196-209, July/Sept. 1999.

[13] D. Cohen-Or, A. Solomovic and D. Levin. "Three-Dimensional Distance Field Metamorphosis". In *ACM Trans. Graphics*, volume 17/2, pages 116-141, 1998.

[14] D.E. Breen and R.T. Whitaker. "A Level-Set Approach for the Metamorphosis of Solid Models". In *IEEE Trans. Visualization and Computer Graphics*, volume 7/2, pages 173-192, 2001.

[15] B. Baumgart. "Winged-edge polyhedron representation". *Technical Report CS–320*, Stanford University, Stanford, CA, 1972.

[16] Swen Campagna, Leif Kobbelt and Hans-Peter Seidel. "Directed Edges — A Scalable Representation for Triangle Meshes". In *Journal of Graphics Tools: JGT*, volume 3(4), pages 1-12, 1998.

[17] David P. Dobkin. "Computational Geometry and Computer Graphics". *TR-383-92*, page 22, 1999.

[18] P. Cignoni, C. Rocchini, and R. Scopigno. "Metro: Measuring error on simplified surfaces." In *Computer Graphics Forum*, volume 17(2), pages 167-174, 1998.

[19] W. E. Lorensen and H. E. Cline. "Marching Cubes: A high resolution 3D surface construction algorithm". In *Proc. SIGGRAPH '87 (Anaheim, Calif., July 27-31, 1987)*, volume 21/4, pages 163-169. ACM SIGGRAPH, New York, July 1987.

[20] Leif P. Kobbelt, Mario Botsch, Ulrich Schwanecke and Hans-Peter Seidel. "Feature Sensitive Surface Extraction from Volume Data". SIGGRAPH 2000.

[21] L. Carpenter, R. A. Drebin and P. Hanrahan. "Volume rendering". In *Proc. SIGGRAPH '88 (Atlanta, Georgia, August 1-5, 1988)*, volume 22/4, pages 65-74. ACM SIGGRAPH, New York, August 1988.

[22] Christian Rössl, Frank Zeilfelder, Günther Nürnberger and Hans-Peter Seidel. "Visualization of Volume Data with Quadratic Super Splines". In *VIS '03: Proceedings of the 14th IEEE Visualization 2003*, IEEE Computer Society, 2003.

[23] Christian Rössl, Frank Zeilfelder, Günther Nürnberger and Hans-Peter Seidel. "Reconstruction of Volume Data with Quadratic Super Splines". In *Transactions on Visualization and Computer Graphics*, volume 10(4), pages 397-409, July-August 2004.

[24] M. Levoy. "Efficient ray tracing of volume data". In *ACM Transactions on Graphics*, volume 9/3, pages 245-261, July 1990.

[25] M. W. Jones. "3D distance from a point to a triangle". *Technical Report CSR-5-95*, Department of Computer Science, University of Wales, Swansea, February 1995.

[26] H. Gouraud. "Continuous shading of curved surfaces". In *IEEE Transactions on Computers*, volume C-20/6, pages 623-629, 1971.

[27] G. Thürmer and C.A. Wüthrich. "Computing Vertex Normals from Polygonal Facets". In *J. Graphics Tools*, volume 3/1, pages 43-46, 1998.

[28] C. H. Séquin. "Procedural spline interpolation in unicubix". In *Proceedings of the 3rd USENIX Computer Graphics Workshop*, pages 63-83, 1986.

[29] J. Huang, Y. Li, R. Crawfis, S.-C. Lu and S.-Y. Liou. "A complete distance field representation". In *Visualization, 2001. VIS '01. Proceedings*, pages 247-254, 2001.

[30] Henrik Aanæs and J. Andreas Bærentzen. "Pseudo-Normals for Signed Distance Computation". In *Proc. Conf. Vision, Modeling, and Visualization*, pages 407-413, 2003.

[31] J. Andreas Bærentzen and Henrik Aanæs. "Signed Distance Computation Using the Angle Weighted Pseudo-Normal". In *IEEE Trans. Visualization and Computer Graphics*, volume 11/3, pages 243-253, May/June 2005.

[32] J. Andreas Bærentzen and Henrik Aanæs. "Generating Signed Distance Fields From Triangle Meshes". *IMM-TECHNICAL REPORT-2002-21*, 2002.

[33] Christian Sigg, Ronald Peikert and Markus Gross. "Signed Distance Transform Using Graphics Hardware". In *Proc. IEEE Conf. Visualization '03*, pages 83-90, 2003.

[34] M.W. Jones. "The Production of Volume Data from Triangular Meshes Using Voxelisation". In *Computer Graphics Forum*, volume 15/5, pages 311-318, 1996.

[35] B.A. Payne and A.W. Toga. "Distance Field Manipulation of Surface Models". In *Computer Graphics and Applications*, volume 12/1, 1992.

[36] John Strain. "Fast Tree-Based Redistancing for Level Set Computations". In *J. Computational Physics*, volume 152/2, pages 664-686, 1999.

[37] André Guéziec. "Meshsweeper: Dynamic Point-to-Polygonal Mesh Distance and Applications". In *IEEE Trans. Visualization and Computer Graphics*, volume 7/1, pages 47-60, January-March 2001.

[38] S. Mauch. "A Fast Algorithm for Computing the Closest Point and Distance Transform". *Technical Report caltechASCI/2000.077*, Applied and Computational Math., Calif. Inst. of Technology, 2000.

[39] A. Sud, M.A. Otaduy and D. Manocha. "Difi: Fast 3D Distance Field Computation Using Graphics Hardware". In *Computer Graphics Forum*, volume 23/3, 2004.

[40] G. Borgefors. "Chamfering: A fast method for obtaining approximations of the Euclidean distance in N dimensions". In *Proceedings, 3rd Scandinavian Conference on Image Analysis, Copenhagen, Denmark*, pages 250-255, 1983.

[41] G. Borgefors. "Distance Transformations in Arbitrary Dimensions". In *CVGIP: Computer Vision Graphics Image Processings*, volume 27, pages 321-345, 1984.

[42] S. Svensson and G. Borgefors. "Digital Distance Transforms in 3D Images Using Information from Neighbourhoods up to $5 \times 5 \times 5$". In *Computer Vision and Image Understanding*, volume 88, pages 24-53, 2002.

[43] P.-E. Danielsson. "Euclidean Distance Mapping". In *Computer Graphics and Image Processing*, volume 14, pages 227-248, 1980.

[44] J.C. Mullikin. "The Vector Distance Transform in Two and Three Dimensions". In *CVGIP: Graphical Models and Image Processing*, volume 54/6, pages 526-535, 1992.

[45] R. Satherley and M.W. Jones. "Vector-City Vector Distance Transform". In *Computer Vision and Image Understanding*, volume 82/3, pages 238-254, 2001.

[46] Mark W. Jones, J. Andreas Bærentzen and Milos Sramek. "3D Distance Fields: A Survey of Techniques and Applications". In *IEEE Transactions on Visualization and computer graphics*, volume 12/4, July/August 2006.

[47] Q. Z. Ye. "The signed Euclidean distance transform and its applications". In *Proceedings, 9th International Conference on Pattern Recognition*, pages 495-499, 1988.

[48] F. Leymarie and M. D. Levine. "Fast rater scan distance propagation on the discrete rectangular lattice". In *CVGIP: Image Understand*, volume 55(1), pages 84-94, January 1992.

[49] I. Ragnemalm. "The Euclidean distance transform in arbitrary dimension". In *Pattern Recognition*, Lett. 14, pages 883-888, 1993.

[50] D.E. Breen, S. Mauch and R.T. Whitaker. "3D Scan Conversion of CSG Models into Distance Volumes". In *Proc. IEEE Symp. VolumeVisualization*, pages 7-14, Oct. 1998.

[51] D. Breen, S. Mauch and R. Whitaker. "3D Scan Conversion of CSG Models into Distance, Closest-Point and Colour Volumes". In *Volume Graphics, M. Chen, A.E. Kaufman, and R. Yagel, eds.*, pages 135-158, London: Springer, 2000.