Continuous signed distance field representation of polygonal meshes

Mathieu Sanchez

22/08/2011

Abstract

Polygonal meshes are popular in computer graphics, but have major drawbacks. They cannot accurately represent smooth shapes and their validity can be broken with holes and self intersections. Moreover, it only defines the surface of an object rather than the volume which makes certain operations very complicated if not impossible. Implicit functions and more generally FReps can represent volumes and perform complex operations more easily, but they are less natural to work with and user-control is more complicated.

A signed distance field (SDF) function has to be generated in order to be able to combine pre-existing polygonal meshes with FRep objects. SDF for a mesh returns a distance between an arbitrary point in space and its closest point laying on the surface of the mesh. Additionally, this distance value is negative if the point is inside the surface, and positive otherwise.

The problem of SDF generation for a mesh has been partly solved in the past, however there is still room for improvement in terms of performance of the SDF evaluation and of resolution of the issues related to invalid meshes with holes and inverted normals.

Acknowledgments

I would like to thank the following people:

Prof. Alexander Pasko, who graciously supervised this project, gave me valuable advice and helped me focus on the real problems of this project.

Denis Kravtsov, for giving me this project idea, for his expertise on the subject, and the help he provided throughout the year.

Turlif Vilbrandt from Uformia AS for his feedback on geometric errors handling and for clarifying the objectives of this project.

Contents

1 Introduction **8**

2 Related work **10**

3 An efficient signed distance field function **12**
3.1 Distance to a mesh . . . 12
3.1.1 Vector calculus . . . 13
3.1.2 Barycentric coordinates . . . 14
3.1.3 Comparison . . . 14
3.2 Signed function . . . 15
3.2.1 Angle weighted pseudo-normals . . . 16
3.3 Optimisation . . . 17
3.3.1 Regular grid . . . 18
3.3.2 Octree . . . 20
3.3.3 Bounding volume hierarchy . . . 23
3.3.4 Multi-threading . . . 26
3.3.5 GPGPU of acceleration structures . . . 26
3.3.6 Comparison . . . 28

4 Mesh defects **29**
4.1 Duplicated vertices . . . 29
4.2 Degenerated triangles . . . 30
4.3 Holes . . . 31
4.4 Flipped faces . . . 34
4.5 Self-intersections . . . 35

5 Applications **37**
5.1 Modelling . . . 37
5.1.1 Boolean operations . . . 37
5.1.2 Transition models . . . 39
5.1.3 Micro-structures . . . 40
5.2 Animation . . . 42
5.2.1 Metamorphosis . . . 42
5.2.2 Space-time blending . . . 44

5.3 Rendering . . . 45
5.3.1 Polygonizer . . . 45
5.3.2 Slices . . . 48
5.3.3 Ray-marching . . . 48
5.4 Physics . . . 50

6 Application design and implementation **52**
6.1 SDF library . . . 52
6.2 Demo application . . . 57
6.3 Other applications . . . 59

7 Conclusion **60**
7.1 Limitations . . . 60
7.2 Future directions . . . 61

A Data sets **64**

B Applications **67**

List of Figures

3.1 Partitions of the domain by parameters s and t 13
3.2 Voronoi regions 14
3.3 A naive computation of the sign 15
3.4 The sign breaking shape without angle-weighted normals (left) and with (right) 16
3.5 The sign breaking shape slices without angle-weighted normals (left) and with (right) 17
3.6 The brute force method performance for 1 000 000 samples . . . 17
3.7 Space divided by a regular grid on the Stanford bunny (left) and the happy Buddha (right) 18
3.8 The grid accelerated method performance for 1 000 000 samples . 19
3.9 Space divided by an octree on the Stanford bunny (left) and the happy Buddha (right) 21
3.10 The octree accelerated method performance for 1 000 000 samples 22
3.11 Space divided by a BVH (AABB-Tree) on the Stanford bunny (left) and the happy Buddha (right) 24
3.12 The BVH accelerated method performance for 1 000 000 samples 25
3.13 Performance comparison of the different methods using a logarithmic base 10 axis for readability 28

4.1 Teapot without vertex welding (left) and with (right). Red faces are self-intersecting faces 30
4.2 Re-indexing the degenerated triangle T1 (ACB) 31
4.3 The half edge data structure. 32
4.4 The Stanford bunny with holes (left) and fixed using ear-clipping (right) in solid (top) and wireframe (bottom) 33
4.5 Flipped faces causing error in the sign and perturbing the ray-caster (left) and after being fixed (right) 34
4.6 Self-intersecting faces 36

5.1 Union between a teapot and a bunny 38
5.2 Blended union between a teapot and a bunny 38
5.3 Blending the Stanford bunny with the mirror of itself 39

5.4 A bunny with a teapot head (top row), a headless bunny, a teapot-bodied bunny and a Buddha with dragon legs (bottom row) . . . 40
5.5 Hollow solid model of the Buddha . . . 40
5.6 The happy Buddha with a regular lattice micro-structure . . . 41
5.7 Blended micro-structures, slice (left), ray-marched (right) . . . 42
5.8 A teapot morphing to a bunny . . . 43
5.9 A Buddha morphing to a dragon . . . 43
5.10 Two disks blend to one disk (top) and a slice of the Buddha blend to a Dragon (bottom) . . . 44
5.11 Space-time blending of a dragon and Buddha . . . 45
5.12 Different level of details of the Happy Buddha shaded (top row) and wire frame (bottom row) . . . 46
5.13 Different level of details of the Stanford Bunny shaded (top row) and wire frame (bottom row) . . . 46
5.14 Different level of details of the Stanford Dragon shaded (top row) and wire frame (bottom row) . . . 47
5.15 The original Stanford Dragon mesh (left) and after polygonization (right), both have a similar polygon count . . . 47
5.16 The original Stanford bunny mesh (left) and after polygonization (right) . . . 47
5.17 Slices of the bunny sign breaking shape (first row) and the bunny (second row) . . . 48
5.18 Ray-marching . . . 48
5.19 Ray-marching using distance information . . . 49
5.20 Ambient occlusion . . . 50
5.21 Cloth colliding with the Buddha (100 000 faces) and the Stanford Dragon (100 000 faces) in real-time . . . 51

6.1 The signed distance field interface . . . 53
6.2 The UML diagram of the look-up policies and multiple queries policies . . . 54
6.3 The sign policies for the signed distance field . . . 55
6.4 The discretized field classes . . . 56
6.5 Some basic operations with distance fields to be done on the CPU 57
6.6 The two error related classes for detection and correction . . . 57
6.7 The basic class diagram of the SDF-PM application . . . 58

A.1 The sign-breaking mesh. Modelled with very sharp features and close features to break the sign computations. 40 faces, no holes, no self-intersection. . . . 64
A.2 A pyramid without a bottom face. The way the hole is set up creates and infinite zero shape. 4 faces, one hole, no self-intersection. 65
A.3 The Stanford bunny, from the Stanford 3D scanning repository. 4968 faces, three holes at the bottom, no self-intersection. . . . 65

A.4 The Utah teapot by Martin Newell. 6320 faces, many holes, many self-intersections, duplicated vertices. 65
A.5 The NCCA troll, 36512 faces, many holes, no self-intersections . 66
A.6 The dragon from the Stanford 3D scanning repository. 100 000 faces, no holes, no self-intersections. 66
A.7 The dragon from the Stanford 3D scanning repository. 100 000 faces, no holes, no self-intersections, a few degenerated triangles. 66

B.1 The sdf-pm application used to test possible applications of the signed distance field. 68
B.2 Using a DirectX framework developed a few years ago. It displays the mesh in wireframe and shows the closest point on the mesh using the brute force and the acceleration structure. It also displays mesh defects with colored faces (red is self-intersecting, blue is fixed hole, green is flipped face. 69

List of Tables

3.1 Comparison of distance methods 14
3.2 Models used for comparison and number of faces 18

Chapter 1

Introduction

Computer graphics have developed over the years many ways of representing objects in 3D. The most developed representations are called Boundary representations (BReps). They only define the surface of the object. BReps are not suited for all applications. CAD for example requires solid models to be manufactured, but boundary representations can introduce errors because they are inherently not solids. Boundary representations include parametric surfaces and polygonal meshes. The latter has been the most popular in computer graphics in the past, for many reasons. They are easy to render and very efficient algorithms have been developed to display these surfaces even on limited hardware. Also, their memory requirements are relatively low. For these reasons, current graphics hardware is tailored for this representation of models. The main BRep alternative to polygonal meshes are parametric surfaces. They are resolution independent and can define smooth surfaces with very little memory. However, they are difficult to model with and often require a tessellation stage before rendering.

Volumetric representations were developed very early mostly using discrete fields. The model becomes an aggregation of voxels holding a scalar value. However voxel representation can be memory expensive and aliasing becomes quickly a problem. Representing a sphere for instance, is very difficult because the accuracy is limited by the number of voxels and the amount of memory required for good accuracy is enormous even with today's computers. An alternative to voxel representation is implicit surfaces. They are mathematical functions mapping to $\mathbb{R}$. They are resolution independent and provide more information than BReps. They define solids correctly but modelling with implicit surfaces can be tedious and somehow inaccurate. Constructive Solid Geometries (CSG) combines simple solid primitives together using simple operations such as union, intersection and subtraction to build more complex shapes.

Function representation (FReps) is a model representation which can fulfill most of the requirements of a robust, accurate, object representation. It is resolution independent. The FRep framework allows us to represent a volumetric object as a tree of basic primitives and operations such as blending, intersection,

micro-structures and many more. The FRep tree allows to represent complex solid models in a resolution independent form, with a low memory foot print. The function representation defines objects using a function mapping to $\mathbb{R}$, making this model dimension independent. The object is perfectly defined with an inside and outside. The boundary is set at the level 0. The interior of the object is defined by the positive values.

Signed distance fields representation of polygonal meshes would allow us to include pre-existant meshes into the FRep framework. It is not the only use of this technique, as signed distance fields have many applications in many fields such as Physics, CAD-CAM, rendering.

In Chapter 2, we present the reasons for using the signed distance field representation over some of the more popular representations. The discussion extends to the main applications of the signed distance field representation and how they can be computed from a polygonal mesh.

In Chapter 3, we present the main algorithms used to compute the signed distance fields and the process of finding an optimal solution for the fastest continuous signed distance field function from a triangle mesh. It concludes on a comparison of the different methods explored.

In Chapter 4, the geometric errors from triangle meshes are described and how they can affect the signed distance field function. A set of algorithms to detect the mesh defects are introduced and explained. Methods to improve or fix the defects are introduced and their limitations are discussed.

In Chapter 5, the design of the library and its extension is presented. It is followed by a short description of the software application based on the library and their design.

In Chapter 6, the work is evaluated. We outline the drawbacks as well as the advantages of this representation. Furthermore, limitations and future work is briefly discussed.

Chapter 2

Related work

Signed distance fields were first introduced for image processing. . More recently, Green (2007) introduced a method to use signed distance fields to render anti-aliased sharp glyphs using alpha-testing.

Both continuous and discrete signed distance fields of a polygonal mesh have been increasingly popular because of their numerous applications. They are commonly used in physics (Fuhrmann et al. 2003). They can detect collisions for both rigid bodies (Guendelman et al. 2003) and soft bodies (Fisher & Lin 2001). Signed distance fields are also used for animation purposes. Metamorphosis and space time blending can easily be done using distance fields (Pasko et al. 2004). They have also been used to produce melting objects (Jones 2003). Animation application can also benefit from signed distance field by generating an object's skelton as presented in Gagvani & Silver (1999). In modelling, CSG operations become trivial to do (Pasko et al. 1995). They also help in challenging modelling problems such as hair modelling (Sourin et al. 1996) or filling a model of microstructures (Pasko et al. 2011). Finally, signed distance fields are also helpful for rendering purposes as detailed in Quilez (2008). Many more applications can be found in Jones et al. (2006) and Erleben & Dohlmann (2007).

A lot of research has be done into generating a discretized field especially when accuracy is not required. Rosenfeld & Pfaltz (1966) first introduced a propagation method to approximate the euclidean distance to a boundary on a voxel grid which reduces the generation time considerably, but at the cost of accuracy, especially as the distance to the boundary increases. These distance transform methods were largely investigated and resulted in derived versions. Jones et al. (2006) classify those techniques according to how the distance is estimated, and how the distances are propagated across the volume. Even though they are fast to generate a field, they are inaccurate and cannot provide a true continuous function.

To generate an exact continuous signed distance field from a polygonal mesh, the distance to each triangle of the mesh can be computed. It would quickly become a bottleneck so Payne & Toga (1992) introduced a few optimisations such as hierarchy trees and square distances. They are presented in Section 3.

Another method was presented in Guéziec (2001) and builds upon a hierarchical representation of the triangle mesh. Large parts of the mesh can be rejected by comparing the distance to the triangles at the higher levels. Another exact continuous representation of a polygonal mesh as a scalar field was introduced by Fryazinov et al. (2011). It uses half-spaces to divide the space and uses unions and intersections to build the objects scalar function. However, the algorithm is slow and not viable for large meshes. A more direct approach consist of finding the shortest distance to the mesh and then computing the sign. Botsch & Kobbelt (2001) introduced a method to compute accurately the signed distance field to a close manifold polygonal mesh. It is an accurate method but requires some optimisations in order to be practical.

Chapter 3

An efficient signed distance field function

A distance field function provides the shortest distance to an object from an arbitrary point in space. The sign of the function indicates whether the point is inside or outside of the object. The signed distance field function can be designed for various surface representation types. This thesis only focuses on the signed distance field representation of a polygonal mesh, and more particularly of a triangle mesh. However, the polygonal mesh needs to be manifold, to truly define an object and not self-intersecting for the same reasons.

The function is formally defined:

$f(P) = sign_\Sigma(P) \cdot dist_\Sigma(P)$

where the unsigned distance of P to the mesh Σ is defined by

$dist_\Sigma(P) = \inf_{x \in \Sigma} \|x - P\|$

and the sign function is defined by

$$sign_\Sigma(P) = \begin{cases} -1 & P \in \Sigma \\ +1 & P \notin \Sigma \end{cases}$$

The algorithm developed first computes the unsigned distance to the mesh and then computes the sign. Other methods exist which compute the distance and the sign at once using a BSP-tree (Fryazinov et al. 2011).

3.1 Distance to a mesh

Computing the shortest distance of a point to a mesh can be done by comparing the distance from the point to each triangle. Because only the comparison is required during the search of the closest point, the square distances are kept until the shortest has been found and the square root is applied at the end.

It saves many cycles and most of the distance algorithms compute the square distance beforehand.

The distance to a triangle has three types of solutions referred to feature type. The feature types can be face, edge or vertex. The feature type needs to be returned by the distance function for the sign computation.

Many ways have been explored to compute the shortest distance. The most naive way can be done by computing the distance to each vertex, then each segmented line, and finally projecting the point onto the plane and testing if the projection is inside the triangle. This method is not very efficient (Ericson 2004, p. 136). Another method introduced in Jones (1995) uses an affine transformation to put the point in triangle space. The triangle is set at the origin on the XY plane. The distance becomes a 2D problem. This method has the advantage of being able to precompute most of the process in a matrix per triangle, but costs in memory. More common approaches are using vector calculus (Eberly 2008) or barycentric coordinates (Ericson 2004, p. 136). The last two methods were investigated for this thesis.

3.1.1 Vector calculus

In Eberly (2008), the triangle is defined as a sum of a base point and a weighted sum of the two leaving edges. $T(s,t) = B + sE_0 + tE_1$ where $s+t \leq 1, 0 \leq s \leq 1$ and $0 \leq t \leq 1$. It makes the distance function dependent on two variables. The solution is then to find the minimum of the function $Q(s,t) = |T(s,t) - P|^2$. The algorithm is then a case analysis provided in pseudo-code by the paper.

The implementation results in many nested if-else instructions because the plane made by the two variables s and t is partitioned to find the feature type as seen in Figure 3.1.

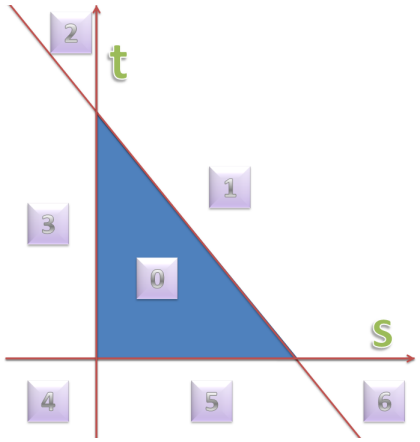

Figure 3.1: Partitions of the domain by parameters s and t

3.1.2 Barycentric coordinates

In (Ericson 2004, p. 136), the author explores a purely barycentric method to find the shortest distance. The first step is to compute the barycentric coordinates by projecting the point P onto the triangle's plane. The barycentric coordinates of a point P in the triangle A, B, C are (α, β, γ) where $P = \alpha A + \beta B + \gamma C$.

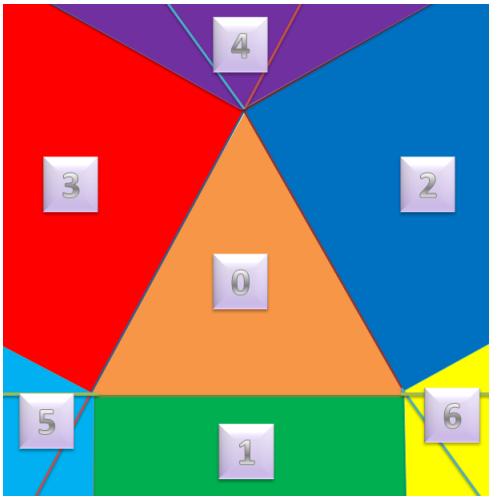

Figure 3.2: Voronoi regions

The barycentric space is then divided by the edge half-spaces, and tests are done for each Voronoi region, vertex region and edge region. See Figure 3.2. If the projected point is not in on of the feature regions, then it is directly in the triangle. Once the feature is found, the point on the triangle is computed using projections.

3.1.3 Comparison

Both methods claimed to be numerically stable, and no noticeable difference could be made between the two algorithms so only the efficiency was considered. The Table 3.1 shows the execution time of both algorithms on four different data sets. Data sets polygonal representation are shown in Appendix A.

	Calculus	Barycentric
Sign breaker, 125 000 samples	0.100s	0.093s
Bunny, 125 000 samples	9.658s	6.816s
Teapot, 125 000 samples	10.978s	7.736s
Buddha, 8000 samples	13.912s	9.410s

Table 3.1: Comparison of distance methods

While the calculus method has far less operations to do, it contains many nested if-else instructions which may damage the cache management of the CPU. The barycentric method also uses vector operations such as dot products which can easily be optimised by the compiler with SSE instructions. Both could explain the difference between the two solutions. The barycentric method is also more suitable for the GPU, as most of the operations used can be improved by the built-in intrinsic functions. However, the triangle distance function does not have a strong impact on the performance of the overall function because it relies on acceleration structures.

3.2 Signed function

In Payne & Toga (1992), the sign of the function is computed by a scanning method. A grid is scanned starting from a corner that is definitely outside the object and if a cell is crossed by the surface, the sign is changed. However, this solution works for a discrete function. For a continuous function, the sign can be computed using the sign of the dot product between the surface normal at the closest point and the vector from P to the closest point. However it does not work for polygonal meshes because of the discontinuities. It is particularly noticeable on sharp features.

Figure 3.3 shows why this method fails for most sharp features. The blue cross represents the point and the red edge of the triangle the closest face found. Because the closest feature is a vertex, the algorithm can choose both edges connected to this vertex.

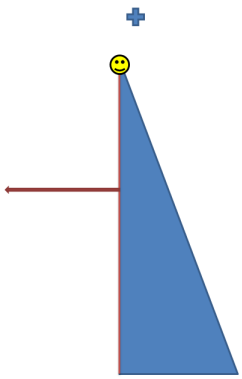

Figure 3.3: A naive computation of the sign

3.2.1 Angle weighted pseudo-normals

The solution is to use angle weighted pseudo-normals (Baerentzen & Aanaes 2005). Pseudo-normals are computed for three feature types of the mesh. Angle weighted pseudo normals are the average of the connected face normals weighted by their incident angle.

1. Faces use the face normal. The face normal can be obtained using the cross-product between two edges of the triangle.

2. Edges use the average of the two adjacent face normals. The weight here is considered to be $\pi/2$ for each normal. This implies that one edge has two faces on each side.

3. Vertices use the angle-weighted normal. The face normals of all incident faces to the vertex are summed using the incident angle as a weight. The incident angle to A of the triangle ABC is $\alpha_i = \arccos\left(\overrightarrow{AB} \cdot \overrightarrow{AC}\right)$.

Vertex normals are often computed using the sum of the normals of the incident faces, weighted by the triangle area, which is fast to compute because the result of the cross-product between the two edges is proportional to the area of the triangle. However, it is not suited for the sign computation because a long stretched face would influence the vertex normal too much and the sign would break on the other side.

The results can be seen in Figure 3.4 and slices are shown in Figure 3.5.

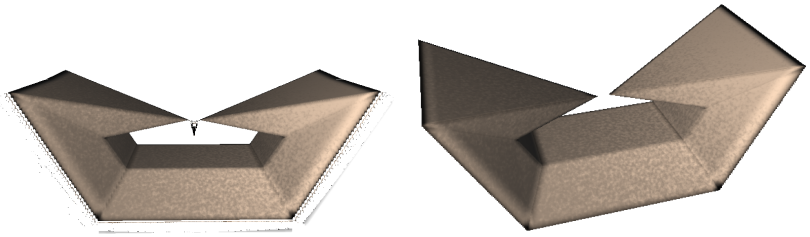

Figure 3.4: The sign breaking shape without angle-weighted normals (left) and with (right)

However, if the mesh is non-manifold, not only there is a discontinuity in the field, but the sign can not be computed accurately because it uses the hole border features. If the hole is reasonably small and convex, a polygonizer would be able to find the zero-level, but ray-casters relying on an accurate distance value to do bigger steps could easily skip the hole.

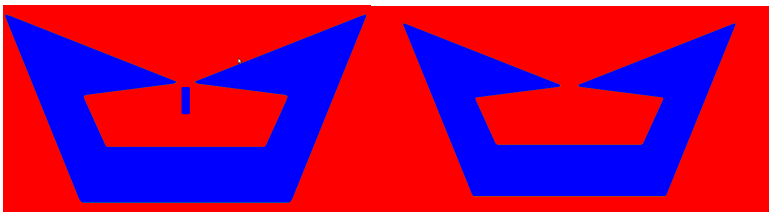

Figure 3.5: The sign breaking shape slices without angle-weighted normals (left) and with (right)

3.3 Optimisation

Computing the signed distance field on larger meshes can become time consuming. Computing the distance to every single polygon of the mesh is largely inefficient and does not take advantage of the spatial coherence. The algorithm complexity is $O(n)$ making it unsuitable for large meshes. The graph in Figure 3.6 shows the brute force performance on different datasets.

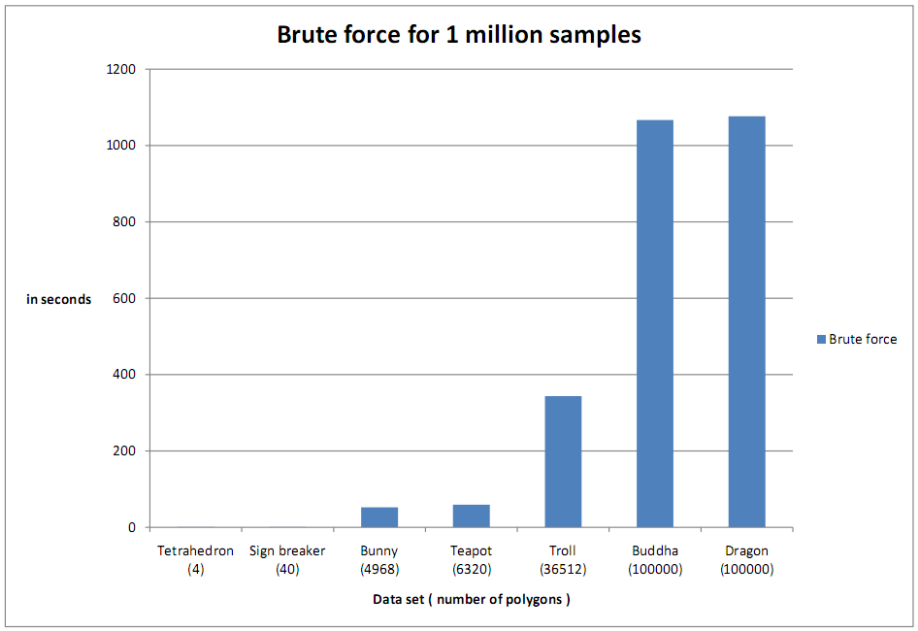

Figure 3.6: The brute force method performance for 1 000 000 samples

The table 3.2 shows the number of faces per model. As expected, the time is linear to the number of polygons. Acceleration structures are often used in ray-tracing and physics. Signed distance fields can benefit from the same structures by slightly changing the traversal methods.

	Number of faces
Tetrahedron	4
Sign breaking	40
Stanford Bunny	4968
Utah Teapot	6320
NCCA Troll	36512
Happy Buddha	100000
Stanford Dragon	100000

Table 3.2: Models used for comparison and number of faces

3.3.1 Regular grid

A regular grid splits the space into cells of equal size. Each cell then holds all the elements that intersect with the cell. This means that some elements can be duplicated. Figure 3.7 shows the space divided by the regular grid.

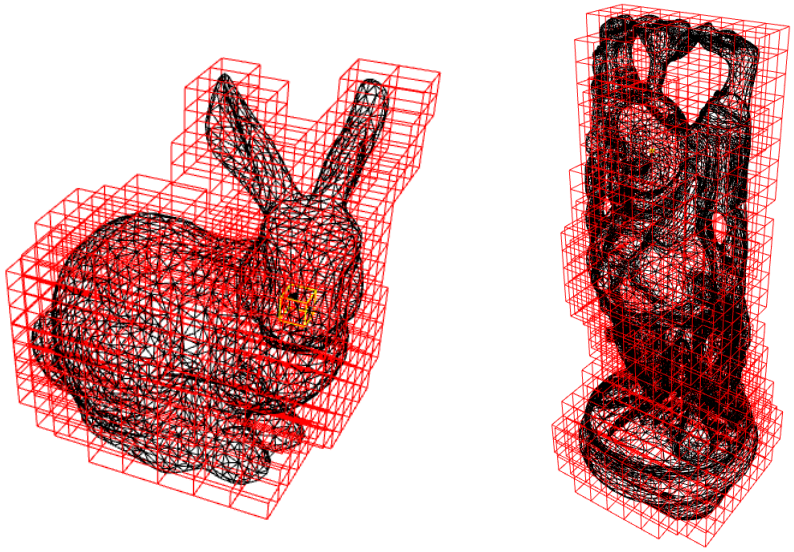

Figure 3.7: Space divided by a regular grid on the Stanford bunny (left) and the happy Buddha (right)

Construction

Finding the size of the cells is the main issue with this method. If the cell size is too small, then a cell will not contain enough elements or many elements will be duplicated. On the other hand, if the size of a cell is too large, too many elements will be contained in it. This will hardly speed up the lookup procedure.

The most common solution is to adapt the grid so that one cell can contain any rotation of the biggest element (Ericson 2004, p. 286). This method suffers from meshes with polygons varying a lot in size. The cell grids then contain too many polygons. An alternative can be to use the average size. But in the case of a good quality mesh with a low standard deviation, grid cells would be too small. The solution selected was a weighted average between the average size and the maximum size. The Figure 3.8 shows how the weighting affects the performance. There is no perfect solution and depends on the data set used.

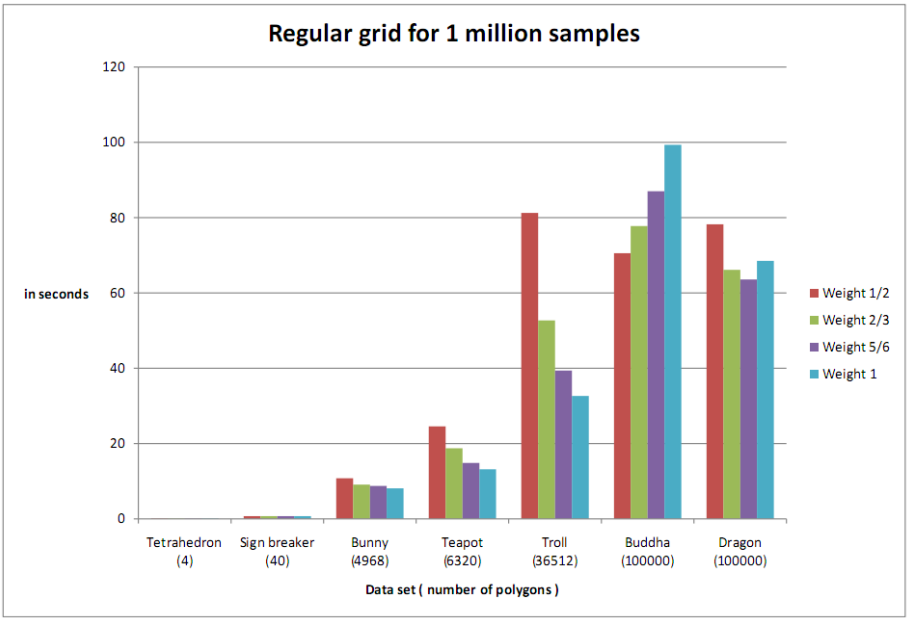

Figure 3.8: The grid accelerated method performance for 1 000 000 samples

Here is a step-by-step description of the grid construction:

1. Compute best cell size
2. Prepare an array of polygon buckets
3. For each polygon:
   (a) Compute axis-aligned bounding box of the primitive
   (b) For each overlapped cell
      i. Use a box-triangle overlap test to check the triangle is in this cell. This is optional, the box-box overlap is slightly faster but will duplicate more elements.
      ii. If true put in corresponding bucket
      iii. Else skip

4. For each cell

   (a) Register current offset and size in cell

   (b) Append cell's polygon list to the master list

The complexity of this algorithm is $O(n+m)$ where n is the number of polygons of the mesh, and m is the number cells. The construction is linear and keeps all the data in one place, the cells only store two indices and the data is kept in one memory chunk.

Traversal

To find the closest point on the mesh, the easiest solution is to check each cell individually. By keeping track of the shortest distance encountered, many cells can be rejected by computing the shortest possible distance to the cells. To improve on this technique, checking the cell the point P is in can help reject many cells very quickly when the point is in a cell that contains polygons.

Here is the algorithm of a brute force traversal:

1. Find the closest cell to the point and compute the distance to all the polygons inside it

2. For each cell in the grid

   (a) If the distance between the cell and the point is smaller than the current distance, find the closest distance to all the points inside it and compare with the current distance

In order to improve the search, a propagation method would help rejecting very early many cells, and eventually discard the rest of the grid when most of the close cells have been checked. As soon as a distance has been found, the list of cells in the radius of this distance is checked and the rest of the grid can be discarded. However, the propagation method turned out to be slower than a brute force search.

Here is the algorithm of a propagation traversal:

1. Find the closest cell. If the cell is not in the grid, project the point to its closest cell.

2. Explore the current node

3. Explore all the nodes connected to this node

4. If a distance has been found, compute the maximum radius and explore the cells in the radius

3.3.2 Octree

The octree structure splits an axis-aligned bounding box into eight axis-aligned boxes. Figure 3.9 shows the space sub-divided by an octree structure.

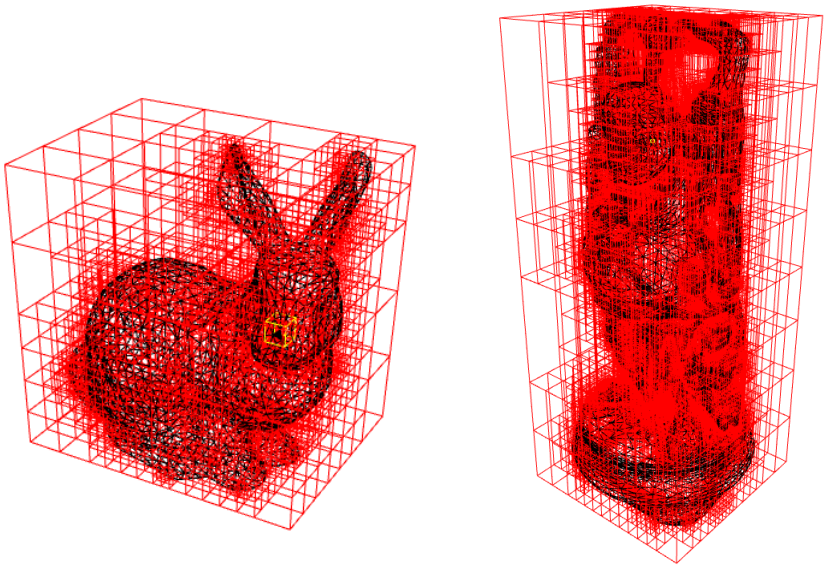

Figure 3.9: Space divided by an octree on the Stanford bunny (left) and the happy Buddha (right)

Construction

The splitting position is usually set at the centre of the box. The process is recursive and the cells are split until the box contains only a certain number of polygons, or a maximum depth has been reached. Like most tree structures, finding the correct settings is not straight forward. An application was written to test many possibilities and find out a guess based on the mesh complexity. The graph in Figure 3.10 shows how the maximum depth affects the performance of the tree. The maximum depth only affects the performance if it is too low. Because the octree divides the space in eight new spaces, depth does not need to be large.

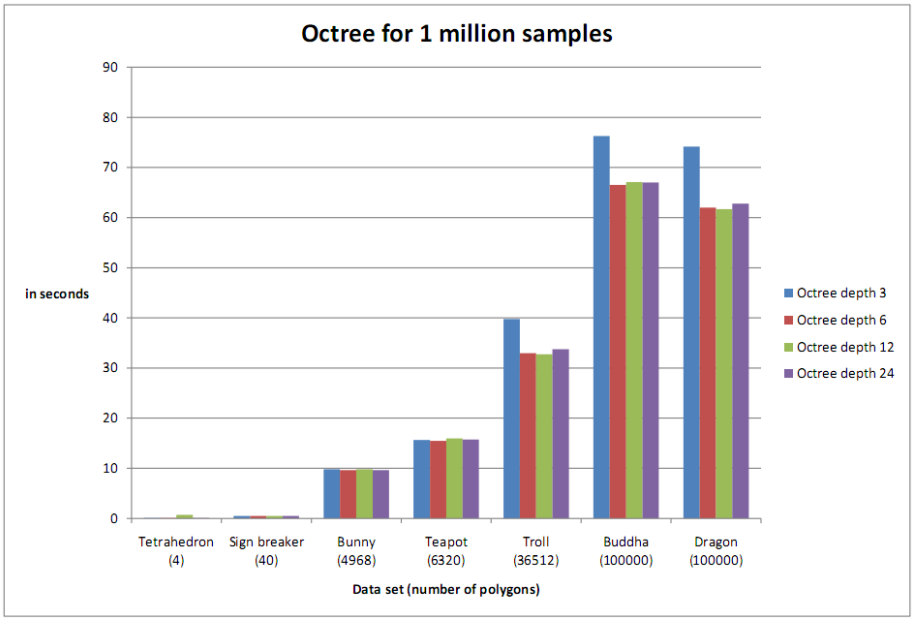

Figure 3.10: The octree accelerated method performance for 1 000 000 samples

The following describes the construction algorithm for the octree.

1. If the leaf conditions have been reached, register offset and size in the node, append the polygons in the master array
2. Otherwise
   (a) Compute the eight children boxes
   (b) Distribute polygons in buckets
   (c) Build eight children nodes recursively

When building the octree, some triangles could overlap two or more boxes. There are three common ways of handling this issue:

1. Split the polygon so that each new polygon fits in one box.
2. Duplicate the polygon so that all the children overlapping have a copy of the polygon.
3. Store the polygon at the node level.

The first solution usually results in the introduction of new issues. Numerical inaccuracies usually add minor errors, and the build process is considerably more time consuming. Splitting polygons also requires to store additional polygons.

The second solution is the most common solution. The duplicated polygons can be queried more than once during the traversal which can slow down the process a lot if there are too many duplicates. However, it allows to have a clear separation between nodes and leaves which makes the traversal faster and cleaner. The box to box overlapping is usually used to decide whether a triangle needs to go in a child box or not, however triangle-box overlapping methods as described in Ericson (2007) can be used to limit the duplications.

The third solution avoids duplicates, but the traversal method suffers from it because each node has to check for possible polygons. The third solution turned out to be the fastest, both for the build time and traversal time.

Traversal

The most naive traversal algorithm would check each node in the given order. Nodes can be rejected if the distance from the point to the node's bounding box is greater than the current shortest distance. Many boxes are rejected this way along with all the polygons it includes. However, the traversal can be improved by sorting the nodes based on their distance to the point. Using this method, the chances of finding a closer hit early on is higher, and helps to reject more nodes quickly. Although this traversal seems more effective, in practice, the brute force seems to be faster. The sort based on the distance is too costly, but checking first the closest node, and then the others in any order can improve the traversal greatly.

The traversal algorithm is presented below:

1. Check distance of all the polygons in this node
2. If this node has children
   (a) Sort the children from closest to furthest
   (b) Recursively traverse each child if the distance from the point to the child's box is smaller than the current shortest distance

3.3.3 Bounding volume hierarchy

Bounding volume hierarchies (BVH) are a general type of tree to divide a space. The BVH used in this thesis is an axis-aligned bounding box tree (AABB-tree). Each level splits the box into two smaller boxes. Figure 3.11 shows the bounding boxes of the nodes of the BVH.

Construction

Building a AABB-tree is often represented as a recursive operation. The space of the bounding box is split in two spaces by a plane. Polygons are put in one of them and one only. Choosing which side a polygon goes into can be done using the middle of the polygon. Each child's node computes its bounding box

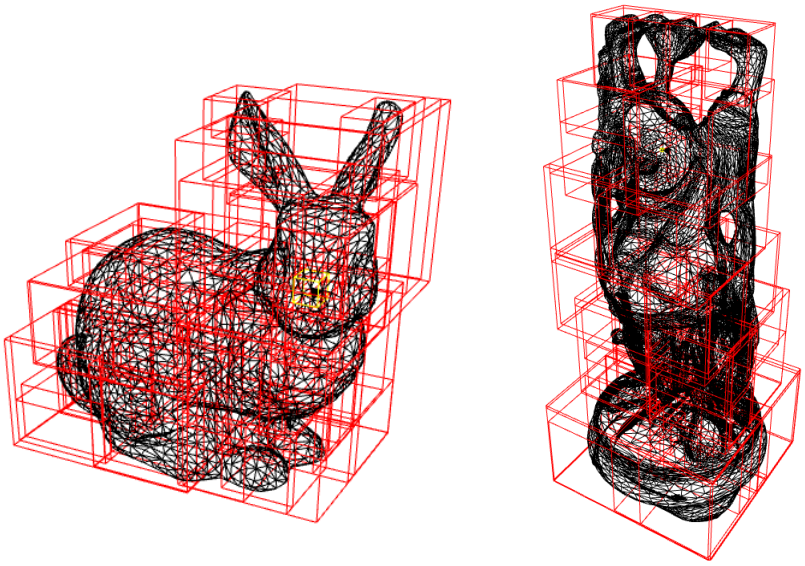

Figure 3.11: Space divided by a BVH (AABB-Tree) on the Stanford bunny (left) and the happy Buddha (right)

based on its polygon list avoiding duplicates to be created because bounding boxes can overlap slightly.

Finding the split position and the split axis is a complex problem and literature is very abundant on this subject (Ericson 2004, p. 242)(Pharr & Humphreys 2004, pp. 200-214). The easiest is to alternate each axis and take the middle point. It can largely be improved by always splitting along the longest axis, and by using the average rather than the middle for the split position. Like the octree, the BVH needs a maximum depth and a target number of polygons per node. These values can be found by a program testing different settings. The graph in Figure 3.12 shows how depth and the target number of triangles per node affects the performance. Overall they are all performing much better than all the previous techniques. A target triangle count per node of roughly six with enough depth seems to be the best options for high polygon count models.

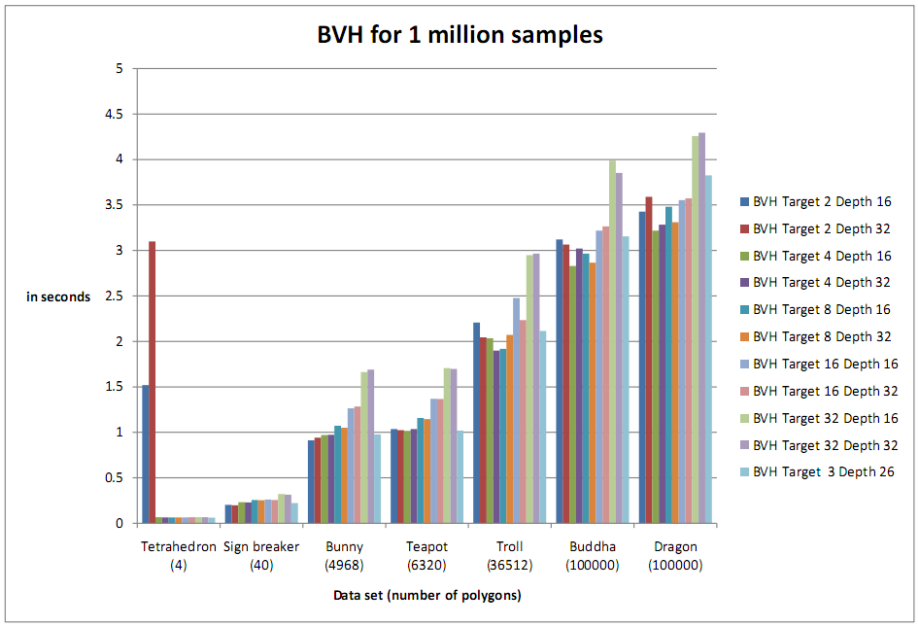

Figure 3.12: The BVH accelerated method performance for 1 000 000 samples

A better solution would be to analyze the cost of the distance function and the cost of a node traversal. The most common technique uses the surface area heuristic (SAH). It tries to split the polygons so that the total surface is roughly equal on each side of the splitting plane.

Here is the algorithm to build the tree:

1. Compute node's bounding box
2. If it has reached the leaf conditions then append polygons to the master array and keep the offset and size inside the node
3. Otherwise, find the axis
4. Find the split position
5. Put polygons in their respective buckets
6. Build the two children recursively using the two buckets

Traversal

Traversing the tree can be done by first checking the closest child and following on to the furthest node. Early rejections are the key of the traversal method. By computing the point to box distances of the child nodes, they can be traversed from closest to furthest, hence allowing an early rejection of the furthest box.

Traversal algorithm is as follow:

1. If it has children
   (a) Get the distance to both children's boxes
   (b) Traverse the closest if the box is closer than the closest hit so far
   (c) Traverse the furthest if the box is closer than the closest hit so far
2. Otherwise
   (a) Compute the distance to all the node's polygons
   (b) Update the closest hit if any

3.3.4 Multi-threading

Improving the algorithm complexity is the priority when optimising, but once it has been done raw processing power can further decrease the time to compute the distance field. Multi-threading is an easy way of improving the overall performance. Modern computers provide an ever improving multi-threading capabilities as multi-core architectures become more commonplace. However, multi-threading of a single query would not improve the performance because of the initial cost of creating the threads. The solution approached is to parallelise the calls so that many queries can be handled faster.

When the list of queries is random, the work is split equally between the available threads. This solution is very easy to implement and avoids locking operations which are slow and could affect performance. The drawback of this solution is that one thread could finish its work earlier than another thread. It means that at the end of the process, only one thread will be working.

The second solution used is to make smaller groups of tasks and allow threads to grab a task, do it, and get the next one available until no task group is left. The size of the groups should be large enough to compensate with the mutex lock happening when a thread gets the next available group. This method works well with a discretized field because each slice can be a group. Each thread processes a full slice and then grabs the next one available. It limits the number of locks by one per slice.

3.3.5 GPGPU of acceleration structures

GPU implementation of the signed distance field can be done by simply using a brute force method. However, the performance boost will be quickly neglected for meshes with high polygon count. So the acceleration structures have to be implemented on the GPU. The GPU has some restrictions which can prevent such structures to be ported so easily.

- Tree data structures usually allocate data on the fly and per-node causing memory to be spread and segmented. Instead of using arrays per node, a master array is used at the root, and all nodes refer to it through an offset and a size. This way all the tree data can be streamed to the

GPU efficiently, and nodes can access it very easily. This solution is also interesting for CPU implementations because it allows to save the trees to binary files. It also avoids memory fragmentation and hence, boosts the overall performance of the traversal.

- GPUs cannot do recursive functions, which are commonly used for the traversal and construction of the tree structure. While the build process can remain on the CPU, the traversal needs to remove the recursive calls. It can be done by implementing a stack directly in the traversal method. Stackless implementation have the theoretical benefit of avoiding call-stack overflow, and improve performance by removing the function-call overheads.

All the data structures presented in this section are offset-based which allow the structures to be sent to the GPU directly. The build functions are recursive but are only called once at initialization, so it can be done on the CPU. The traversal methods are usually recursive, but they can all be re-written in a stackless manner.

The BVH structure was put on the GPU because it gave the most promising results. The stackless traversal was implemented and tested on the CPU and then was ported on the GPU. Stackless algorithms for kd-tree traversal have been developed for ray-tracing (Popov et al. 2007). Based on those algorithms, a new stackless algorithm was tailored for shortest distance look ups. Although Popov et al. (2007) claims the CPU performance can be improved by stackless traversal, it did not improve the CPU version of the BVH traversal.

Here is the stackless algorithm for BVH traversal:

1. Push root in the nodes to process
2. While the stack is not empty
   (a) Retrieve the node at the top of the stack, and pop it
   (b) If the node has children
      i. Get both children and check if their box distance is closer than the current closest record
      ii. Push the furthest, then closest to the stack
   (c) Otherwise
      i. Compute the distance to all the node's polygons
      ii. Update the closest hit if any

Notice that the order in which we push nodes to explore helps to reject them when they are popped off the stack.

One of the issues on the GPU is the size of the local memory, limited to 16KB. The stack takes up most of that space to handle high depth trees, which is why at equal performance, a lower depth tree should be preferred to save memory both on the static stack and on the node array memory.

3.3.6 Comparison

The comparison was made by a program which normalizes the timings to get more accurate results. Datasets are in Appendix A. For the final comparison the very small models were removed because the timings were similar and non-representative of real case scenarios. The results are shown in 3.13.

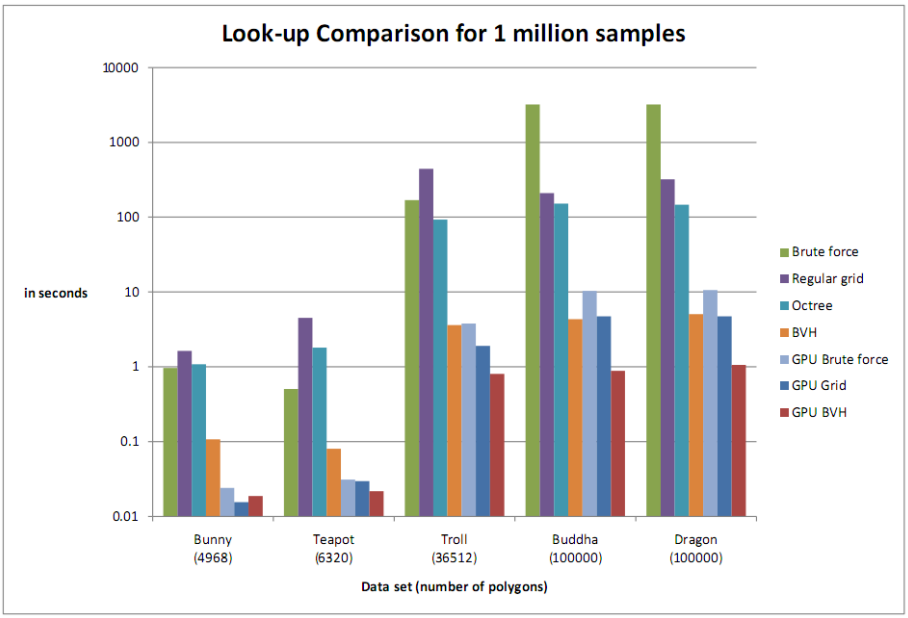

Figure 3.13: Performance comparison of the different methods using a logarithmic base 10 axis for readability

The brute force method is only useful for very low polygon count models. The octree and the grid perform significantly worse on the teapot than the bunny most probably because the teapot is not a good mesh. The skinny triangles do not fit correctly into the cells or nodes.

The GPU implementations perform better overall but the CPU implementation of the BVH seems to perform better on high polygon counts. This is due to the algorithm complexity. The GPU grid implementation and the GPU brute force suffer from their complexity against the BVH.

The GPU BVH outperforms all the other methods on almost all datasets. Just like the CPU implementation, the BVH structure benefits from its simple traversal method.

Chapter 4

Mesh defects

Signed distance field function will always succeed to produce a result but geometric errors tend to give unexpected results or break continuity. Degenerated triangles have unexpected behaviour for triangle distance algorithms because no geometric normal exists. Non-manifold meshes produce discontinuities in the field because of the sign computations. Flipped faces will also break the sign computation, because it relies on the geometric normal, which is based on the vertices order of the triangle. Self-intersection will also break sign computations because the mesh is not a boundary mesh anymore.

4.1 Duplicated vertices

Vertex welding is usually applied to gather all the neighbor vertices that are very close to each other, and should have been joined. In this project, we are only interested in perfectly matching vertices, the ones that are not perfectly matching will result in holes which will be detected later on. Duplicated vertices are an issue not only because they are memory consuming but also because they generate more errors during the next checks. For instance, the teapot model was originally made of patches, which are tessellated and then put in a single model. The patches boundary generate the same vertices. It causes the self-intersection algorithm to mistakenly detect self-intersections and the hole detection to report too many holes as seen in Figure 4.1.

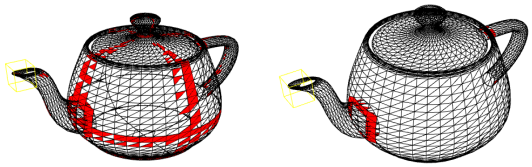

Figure 4.1: Teapot without vertex welding (left) and with (right). Red faces are self-intersecting faces

The objective is to find pairs of vertices with the same value. The naive approach would be to test all the vertices against all the others. The complexity of this algorithm is $O\left(n^2\right)$ which makes it very slow for models with high polygon counts.

The solution explored is using a hash map to keep track of the vertex index. Hash maps use functions to find the correct position in memory. A hash function should give different values for all possible keys, but if it cannot, the different values are stored in a bucket. So the function has to correctly distribute the keys in different buckets. The hash function to to sort points in 3D space is not straight forward. The hash function used is defined in Ratcliff (2006). The coordinates are interpreted as integers, shift the bits so that the same bits do not overlap. The function provides a good distribution so that the buckets are evenly distributed. The expected complexity is constant making the vertex welding of the mesh linear.

4.2 Degenerated triangles

Detecting degenerated triangles is a fairly straight forward task. If the length of the cross-product between two of the triangle edges is null, then the triangle is degenerated. An alternative using angles could be used but is slower and less numerically stable.

Fixing the degenerated triangles however, cannot be done by simply removing the triangle. The degenerated triangles need to be put into two groups: the needles and the caps as defined in Botsch & Kobbelt (2001). The needles are triangles which have two vertices at the same position. The caps are three distinct vertices aligned.

The first group can effectively be removed by simply removing the face. The second group needs more attention because it can generate cracks in the mesh due to T-junctions. Unlike most algorithms, no vertices should be moved and the geometry should be exactly the same. The two triangles creating the T-junction can be merged into one. It does not affect the geometry because the degenerated triangle insures those two triangles are on the same plane. The drawback of this solution is that it removes a valid triangle. Another solution is to split the large triangle connected to the degenerated triangle into two triangles. The splitting process actually only involves re-indexing the large triangle and the

degenerated triangle. This solution has the advantage of keeping the same face count and if the three vertices were not exactly aligned it will preserve this slight misalignment in the shape. Another added bonus is that if any of the adjacent triangles are missing due to a gap in the mesh, the algorithm will still work and not change the shape.

The figure shows the re-index process. The triangle T1 is defined by ACB, T2 by ACD, T3 by AEB and finally T4 by BEC. After the re-indexing, T1 connects BCD and T2 connects ABD, and the others are left unchanged.

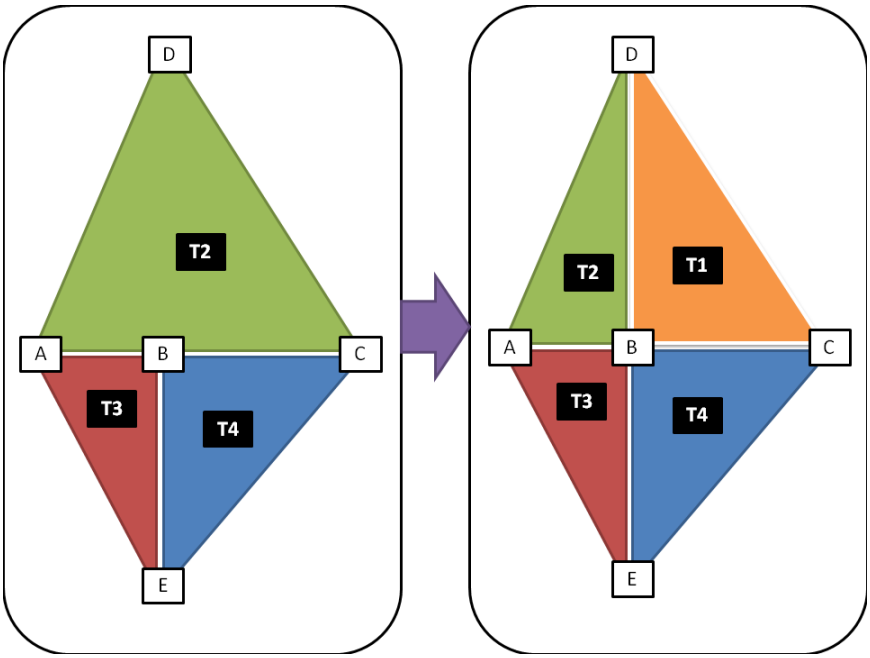

Figure 4.2: Re-indexing the degenerated triangle T1 (ACB)

4.3 Holes

Checking

Checking for holes in a mesh requires a data structure capable of neighborhood queries. The simplest method would be to have an edge map which is cheap in memory, but the half-edge data structure is more interesting because it allows to walk around a hole in order to generate a polygon allowing us to fill the hole.

The half-edge data structures stores faces, half-edges and vertices. Each element holds neighborhood information.

1. The face stores the indices and one half-edge of the face.

2. The half-edge is a directional edge. That is an edge which goes from A to B, the corresponding half being B to A. The half-edge keeps track of
    - Its base (or source) vertex, A.
    - The next half-edge in the face, going from B to the next point.
    - The face it belongs to.
    - Its twin, the other half-edge going in the other direction.
3. Vertices keep track of one of the edges leaving the point.

Figure 4.3 shows the different elements an half-edge can access directly.

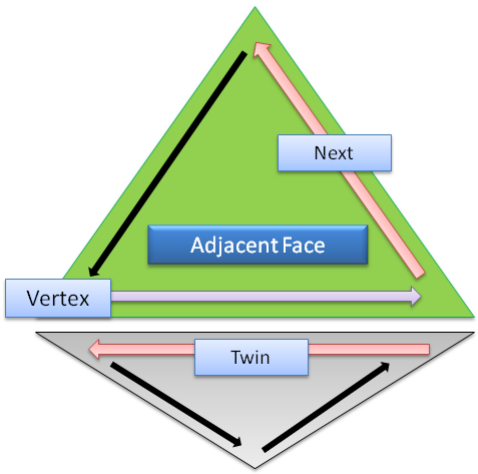

Figure 4.3: The half edge data structure.

If an half-edge does not have a twin, then the edge is a boundary edge. To get the next boundary edge, we have to loop around the destination vertex until we find a new boundary edge. The next edge around a vertex given an incident half-edge can be found by taking the twin's next edge. If it does not have a twin then it is a boundary edge. All the holes are found using this technique and are gathered a single polygon hole. To fix the hole now it is only a matter of triangulating a concave n-sided polygon. Along with the edges, we also keep the normals of the faces adjacent to the hole. They are used to determine where the hole faces.

Triangulating

Triangulating convex polygons is straight forward. From any vertex of the polygon that can be used as a seed, we triangulate using a triangle fan. This

procedure tends to generate many sharp triangles which is not optimal for the acceleration structures and for the mesh quality in general. Another drawback is that it cannot handle concave polygons. Concave polygons are non-convex polygons. Convex polygons are polygons in which any segment which starts and ends in the polygon is entirely in the polygon. The method used in this thesis to handle concave polygons is the ear cutting algorithm.

An n-sided polygon can always be triangulated in $n - 2$ triangles. The principle of the ear cutting algorithm is to remove triangles from the polygon one by one. The triangles are made from three consecutive points P_{i-1}, P_i and P_{i+1}. The triangle can be removed on two conditions.

- First, P_i needs to be a convex vertex. A convex vertex has an interior angle less than π. Getting the interior angle is not required. The solution is to compute the normal of the triangle, and comparing it with a vector which points out of the polygon. Checking the sign of the dot product is enough to know if a vertex is convex.

- Then, no point in the polygon must be inside the triangle. Testing if a vertex P is inside a triangle ABC can be achieved by building three triangles, ABP, BCP and CAP. If the normals of the three triangles point towards the same direction, then P is inside ABC. To test the normals against each other, the sign of the dot product can be used.

If both conditions are met, then the triangle P_{i-1}, P_i, $P_{i+1}$is added and the point P_i is removed from the polygon. The operations is repeated until only three vertices are left in the polygon.

Figure shows the bunny with the holes in the original model and fixed using ear clipping.

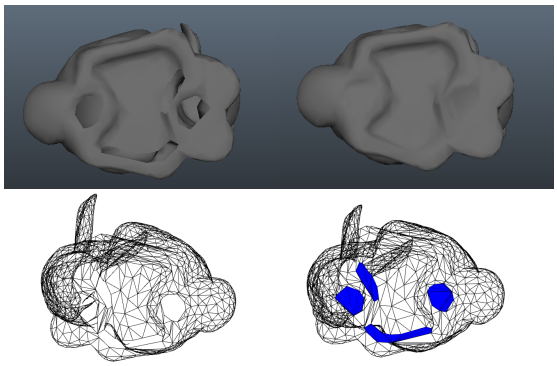

Figure 4.4: The Stanford bunny with holes (left) and fixed using ear-clipping (right) in solid (top) and wireframe (bottom)

4.4 Flipped faces

Flipped faces are often a problem for continuous signed distance fields. If the sign is computed through ray-casting or scanning, flipped faces do not generate errors because the rays will collide on both the front and the back face. However, the angle-weighted pseudo-normals rely on the winding of the triangle to generate normals. Detecting that at least one face is flipped can be achieved by comparing all the edges together, and if there is one pair which has the same start and end, then a face is flipped. This is good enough to discard a mesh, however, a better feedback to the user would be to highlight those faces or to fix those faces directly. Fixing the flipped faces is called unifying normals. The reason why it cannot be absolutely fixed is that the user might prefer one side to the other. In the case of a sphere, normals could point inside and outside. Both solutions are valid, it depends what the user really wants. The Figure 4.5 shows the Stanford bunny with flipped faces and the same bunny after the flipped faces are fixed.

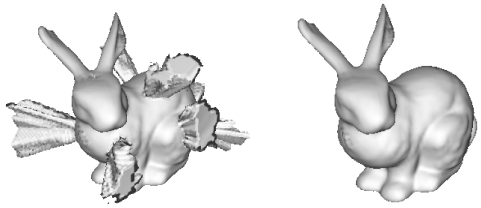

Figure 4.5: Flipped faces causing error in the sign and perturbing the ray-caster (left) and after being fixed (right)

The first step is to bucket faces into two opposite groups. Faces cannot be evaluated individually, because a group of adjacent triangles would not be considered flipped because the direct neighbors are not flipped compared to each other. The solution explored is to start from a seed, and propagate across the mesh. At first, all the faces are set to have an unknown state. The seed is set to be correct. Then, for each neighbor face which state is unknown, the state is set to correct or flipped depending on the shared edge. If the twin half-edges of the two faces have the same source and target, then the two faces have an opposite state. Then, the neighbor faces do the same process with their neighbors. Although this procedure feels recursive, a recursive function would risk a stack overflow with big meshes. Instead, a queue is used to emulate a custom stack.

Using only one seed is not enough. A shell sphere has two surfaces which do not connect. The previous procedure can be easily fixed by iterating through all the faces of the mesh, and if a face is still set to unknown, the same process is applied. It means that the seed will be the first face of each disconnected

surface.

Finally, polygonal meshes can represent non-orientable surfaces. In this case, there are no solution. The most famous non-orientable surfaces or the Mobius stripe or the Klein bottle. The Klein bottle is shown in Figure 4.6. In both cases, all the faces are both flipped and correct. Those cases are easily handled by checking expected states with actual states. Instead of only setting the state of the neighbors with an unknown state, neighbors with an unknown state must match the expected state depending on the shared edge. If there is a mismatch, then the surface is non-orientable. The policy is then to set all the faces to flipped and warn the user that this mesh is not a valid mesh.

In the case that a minority of faces are flipped, normals are unified following the majority. This allows minor mistakes in modeling to be handled quickly. If close to half of the faces are flipped, a reasonable choice cannot be made. The main reason of having half of the faces flipped is when mirroring a geometry.

4.5 Self-intersections

Self-intersection is another major issue with polygon meshes because it cannot be detected by the topology. To detect self-intersection the vertex information (2D or 3D) needs to be available.

A self-intersection can be detected if a face collides with any other face of the mesh. The collision between two triangles is explained in Möller (1997). The test is made of successive simple tests. First, the triangle vertices are tested against the other triangle's plane. If the vertices are all on the same side of the plane, then no intersection happened. Then both triangles are projected onto the principal axis of the intersection line between the two planes. The collision intervals of each projection is compared and if they overlap then a collision occurred between the two triangles.

The issue with comparing all the triangles between each other is its complexity in $O\left(n^2\right)$. Even if we can reduce it to only test against the following faces rather than all the faces, the complexity is still similar ($O\left(\frac{n^2}{2}\right)$). The BVH developed in Sub-section 3.3.3 can be reused to detect self-intersection. A new traversal method is implemented to detect possible collision between two triangles. The axis-aligned box which surrounds the triangle being tested can be tested against the bounding volumes.

1. If the node has children
   (a) For each child node
   (b) If the triangle bounding box intersects with the child's box, recursively explore that child
2. Otherwise
   (a) Test each node's triangles against the tested triangle.

(b) Ignore any connected face

The performance can be further improved by setting both faces as intersecting. While testing each face, those already set as intersecting can be ignored. Figure 4.6 shows the Klein bottle with the self-intersecting faces highlighted in red.

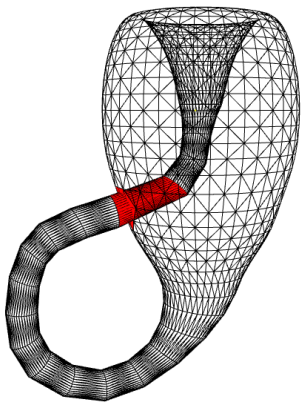

Figure 4.6: Self-intersecting faces

Fixing self-intersections is a large subject and out of the scope of this thesis. Rather than automatically trying to fix the mesh and guessing what the user tried to achieve, the algorithm detects it and asks the user to fix it.

Chapter 5

Applications

Signed distance fields can be applied to numerous areas. They can be used for accurate collision detection, for operations combining different meshes, for morphing between meshes of arbitrary topology and for corrections of the topologies of the meshes. The signed distance field representation can also be used for the correct definition of solids for CAD/CAM application. Finally, they can be efficiently rendered and used for the creation of advanced effects such as ambient occlusion or subsurface scattering in an easy way.

5.1 Modelling

Function representations allow operations which are difficult or impossible with a polygonal representation. For example, defining a micro-structure inside the volume would be tedious and heavy using only a polygonal representation. In a similar way, if only a surface is defined, and one wants to make a thick surface out of it, FReps can help us easily achieve this result.

5.1.1 Boolean operations

Boolean operations with implicit function can be done using minimum and maximum values of each function.

- $f_1 \vee f_2 = \max(f_1, f_2)$ defines the union operation
- $f_1 \wedge f_2 = \min(f_1, f_2)$ defines the intersection operation
- $f_1 \setminus f_2 = \min(f_1, -f_2)$ defines the subtract operation

But these functions are C1-discontinuous where $f_1 = f_2$ Pasko et al. (2011). It means that the field is not continuous anymore preventing further operations to be applied. Better R-functions were introduced in Pasko et al. (1995) to preserve the field continuity except at the joints at the zero level. The operations are defined as follow:

- $f_1 \vee_a f_2 = f_1 + f_2 + \sqrt{f_1^2 + f_2^2}$
- $f_1 \wedge_a f_2 = f_1 + f_2 - \sqrt{f_1^2 + f_2^2}$
- $f_1 \setminus_a f_2 = f_1 - f_2 - \sqrt{f_1^2 + f_2^2}$

An example of a union is shown in Figure 5.1.

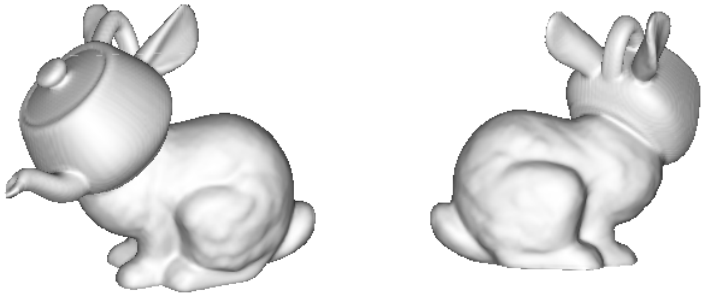

Figure 5.1: Union between a teapot and a bunny

Pasko et al. (1995) introduced blended boolean operations which builds upon the functions previously defined.

- $f_1 \vee_b f_2 = f_1 \vee_a f_2 + \frac{a0}{1+\frac{f_1}{a1}^2+\frac{f_2}{a2}^2}$
- $f_1 \wedge_b f_2 = f_1 \wedge_a f_2 + \frac{a0}{1+\frac{f_1}{a1}^2+\frac{f_2}{a2}^2}$
- $f_1 \setminus_b f_2 = f_1 \setminus_a f_2 + \frac{a0}{1+\frac{f_1}{a1}^2+\frac{f_2}{a2}^2}$

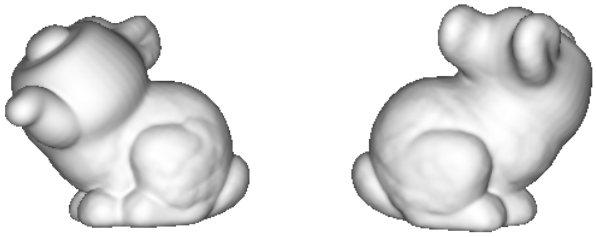

Figure 5.2: Blended union between a teapot and a bunny

The parameters $a0$, $a1$ and $a2$ define the blending behaviour. $a0$ defines the blend intensity, $a1$ and $a2$ define the relative influence of their respective functions. A higher value of $a1$ would make $f1$ blend more than $f2$.

However, those operations break the distance properties which means the new field does not hold the actual distance to the mesh. Keeping the distance properties are far from trivial (Fayolle et al. 2008).

5.1.2 Transition models

To blend two models together simple boolean operations are not suitable because the discontinuity would show exactly where it changes. A simpler method is to define a blending zone. To realize the models in Figure 5.3 and Figure 5.4, we use a linear blend between the two functions using a plane and a threshold distance as the parameter. The parameter is clamped in the range $[0, 1]$.

The blending plane is defined by a normal $\overrightarrow{N}$ and an offset d. If the normal is normalized then the distance of P to the plane is $dist(P) = N_x P_x + N_y P_y + N_z P_z + d$. To convert this value to a transition value in the range $[0, 1]$, the distance is divided by the half-length, then remapped from $[-1, 1]$ to $[0, 1]$

$$m = \min\left(\max\left(\frac{\left(\frac{2 \times dist(P)}{l} + 1\right)}{2}, 0\right), 1\right)$$

Then a linear interpolation can be used between the two models f and g

$$t(x) = m \cdot f(x) + (1 - m) \cdot g(x)$$

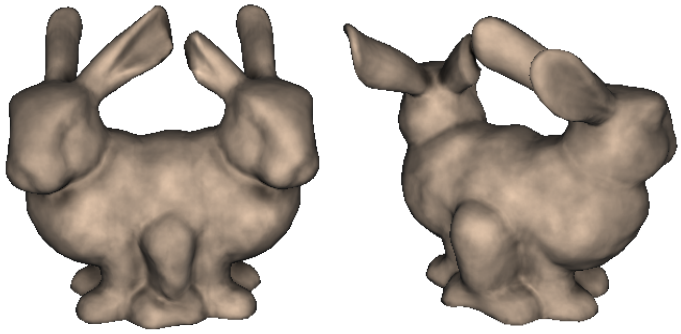

Figure 5.3: Blending the Stanford bunny with the mirror of itself

This method is not convenient to blend because we need to define a blending zone, and it does not produce a real blended union but rather a transform from one model to another along a parameter.

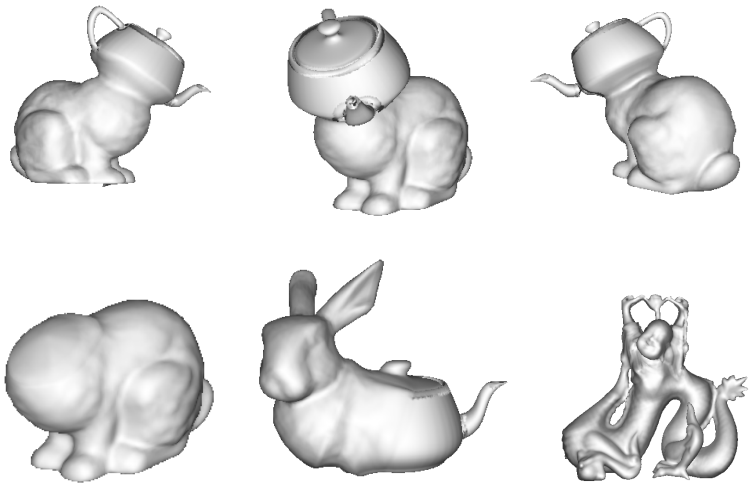

Figure 5.4: A bunny with a teapot head (top row), a headless bunny, a teapot-bodied bunny and a Buddha with dragon legs (bottom row)

5.1.3 Micro-structures

Based on Pasko et al. (2011), a micro-structure suitable for 3D printing was implemented using the signed distance function.

Let $f(x)$ be the function defining the happy Buddha model as a solid. Then, $g(x) = f(x) - thickness$ defines a new solid pushed slightly inside the model. Subtracting $g(x)$ to $f(x)$ will give a hollow solid model as seen in Figure 5.5.

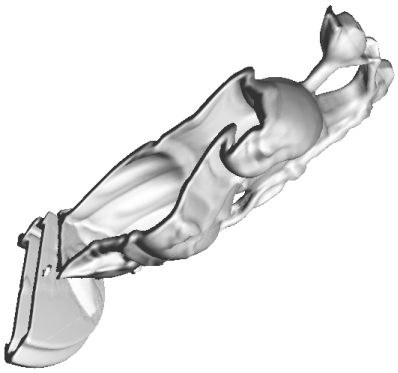

Figure 5.5: Hollow solid model of the Buddha

The micro-structure can be defined with a regular lattice.

First the repeating pattern is done using a sinus wave. For each axis, we define a wave pattern as follow:

$s_i = \sin(f \times P_i + o) - l$

where

- f is the frequency. The higher the value the more often the pattern will be repeated.
- o is the offset. Values of the offset should be in $[-\pi, +\pi]$.
- l is the thickness of the repeating solids. The value l should be in $[-1, 1]$ where -1 means everything is solid, and 1 means nothing is solid.

To get the regular lattice each axis columns are made of the intersection between the other two other axis waves. The final lattice is made of the union between all the axis columns.

Finally, the solid object can be defined as the union between the hollow solid and the intersection between $g(x)$ and the lattice. The results can be seen in Figure 5.6.

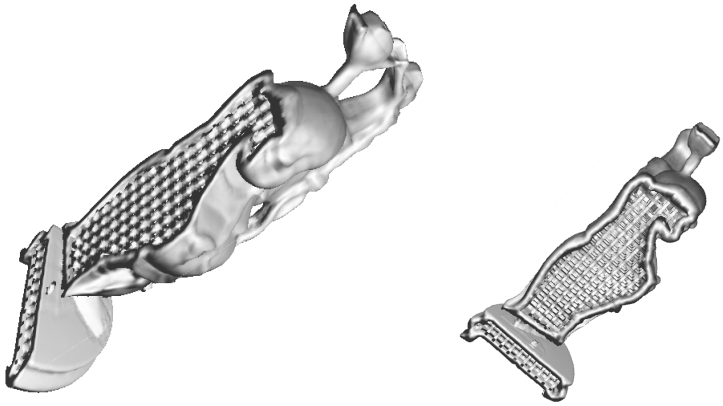

Figure 5.6: The happy Buddha with a regular lattice micro-structure

To further improve the model, blended unions shown in subsection 5.1.1 can be used to get smoother joints. The lattice union is replaced by the blended union and the intersection between the lattice and the inner solid uses the blended intersection. Results can be seen in 5.7.

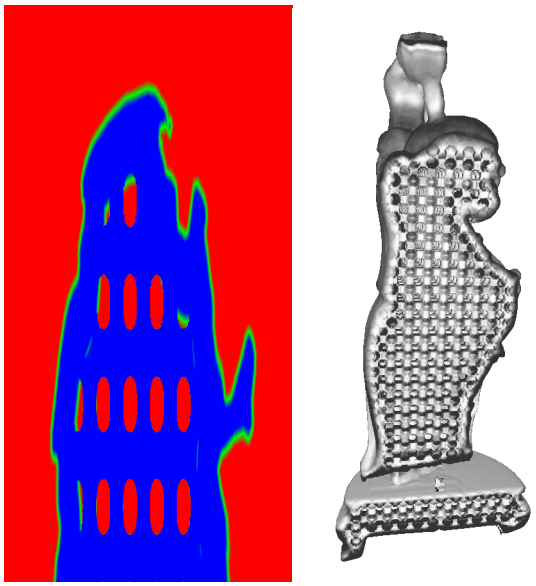

Figure 5.7: Blended micro-structures, slice (left), ray-marched (right)

5.2 Animation

Animation can benefit from distance field. Morphing and metamorphosis can be done relatively easily compared to polygonal techniques.

5.2.1 Metamorphosis

Since the mesh is represented as a function, the morphing can be seen a normal linear interpolation. Let $f(x)$ be the first model and $g(x)$ be the second model, then $m(x) = \alpha \cdot f(x) + (1 - \alpha) \cdot g(x)$ defines the morphing model between f and g. α is a parameter between 0 and 1. When α takes the value 1, the morphing object is the first model, when $\alpha = 0$ then the morphing object is the second model. An example of a linear morph from a teapot to a bunny can be seen in Figure 5.8. Figure 5.9 shows the Buddha model morph to the Dragon model.

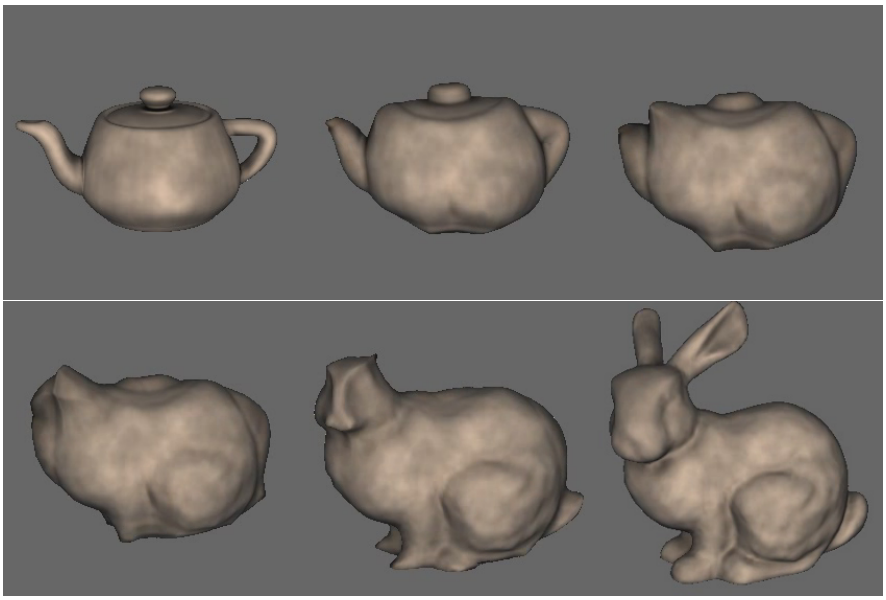

Figure 5.8: A teapot morphing to a bunny

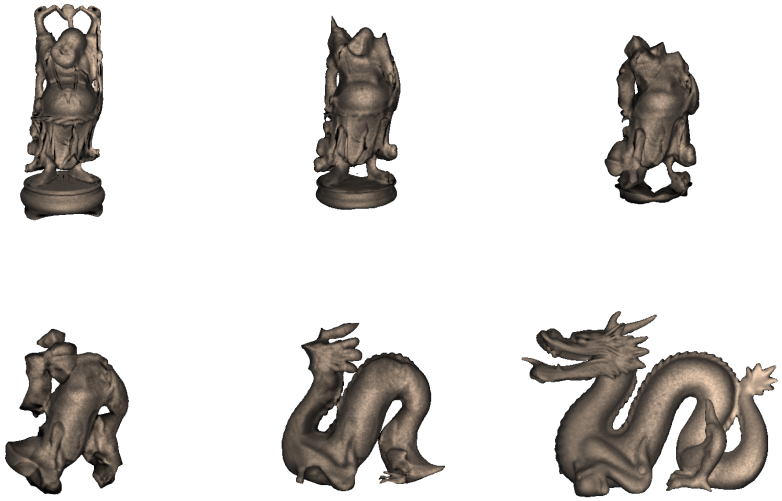

Figure 5.9: A Buddha morphing to a dragon

5.2.2 Space-time blending

The main drawback of linear morphing is that elements can be disconnected from the rest of the shape and blobs can appear during the morph. Linear morphing also lacks parameters to control the morphing. It has only one way of morphing.

Space-time blending was introduced in Pasko et al. (2004). It is a dimension independent method to transform one shape into another. The algorithm is easier to visualize in two dimensions. Both shapes are extended in Z to infinity. Both shapes are cut along two planes in Z. The resulting two shapes are blended together using a bounded blending union. Figure 5.10 shows space-time blending in two dimensions. Finally, slices in the Z axis can be used to get the intermediate shapes.

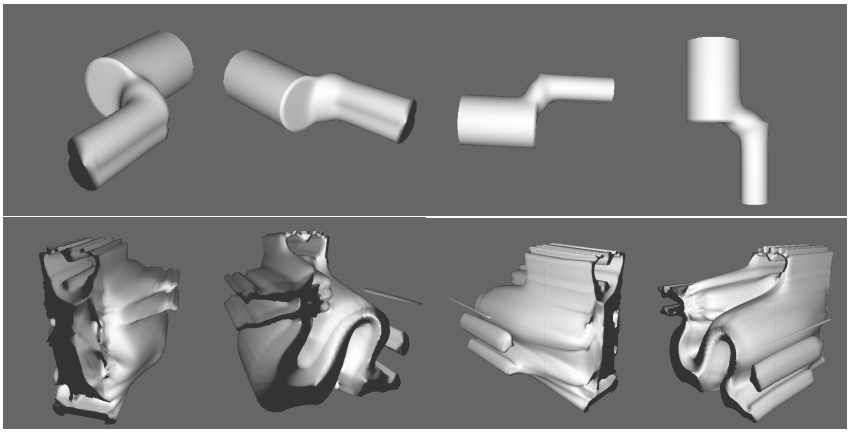

Figure 5.10: Two disks blend to one disk (top) and a slice of the Buddha blend to a Dragon (bottom)

In 3D, the extra dimension can be time. The process is similar but done in 4D and then slices along time will give the intermediate models. The space-time blending produces better results than linear morphing and avoid blobs to stand disconnected from the rest of the shape. Figure 5.11 shows the results obtained using space-time blending. The parameters used are $a0 = 12$, $a1 = 5$, $a2 = 5$, $a3 = 0.1$, clipping plane distance at 0.05, and bounding plane distance at 11.

However, the main drawback of space-time blending is that it is highly non-linear and is difficult to control correctly. There are dramatic changes in the intermediate shapes which can be undesirable. Some methods have been used to overcome these issues, by using a non linear sampling of the time. Also, parts of the shape may not blend with the target shape at all causing the intermediate slices to change drastically from one step to another because the slice stops right at the clipping plane.

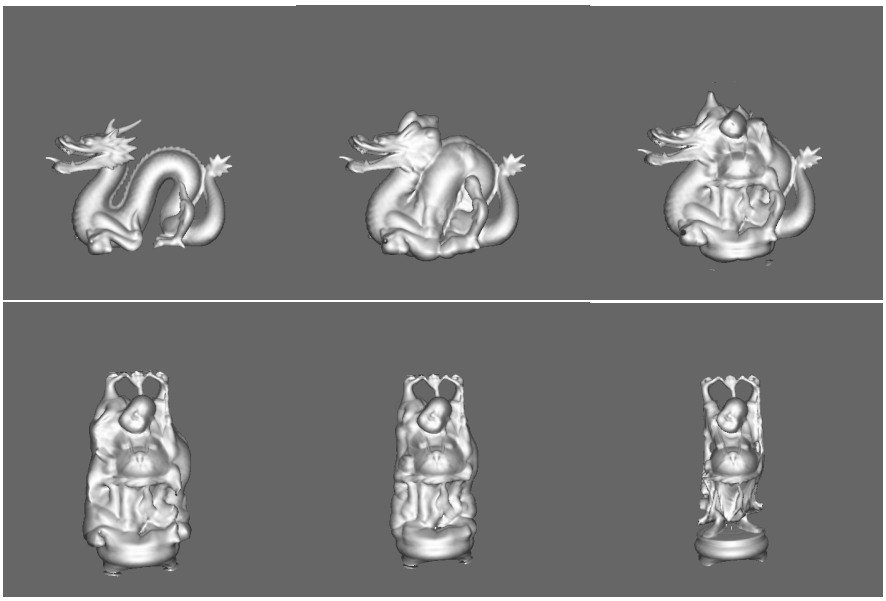

Figure 5.11: Space-time blending of a dragon and Buddha

5.3 Rendering

Rendering volume data is a large subject in computer graphics. FReps are often polygonized using marching cubes or an improved version of it. But it is not the only way of rendering scalar fields.

5.3.1 Polygonizer

The most common algorithm for polygonizing a scalar field is the marching cube algorithm. Many versions of it based on it exist such marching tetrahedra, dual marching cubes, marching cube with edge transform, adaptive marching cubes. All polygonizers allow to change the accuracy of the polygonization. The number of polygons can be increased or decreased. Thanks to this feature, the polygonizer can be used to remesh models at different accuracy levels. Using signed distance fields from a mesh and the polygonizer, a level of detail engine can be achieved. Figures 5.12, 5.13 and 5.14 show different remeshing of a few meshes. The polygonizer used was developed during my Animation Software Development module at Bournemouth University. The base algorithm uses the marching cubes and uses vertex relaxation to improve the mesh quality and accuracy to the field. To produce those images, a legacy DirectX application reads in meshes generated by a stand-alone application which uses the iso-surface meshing library.

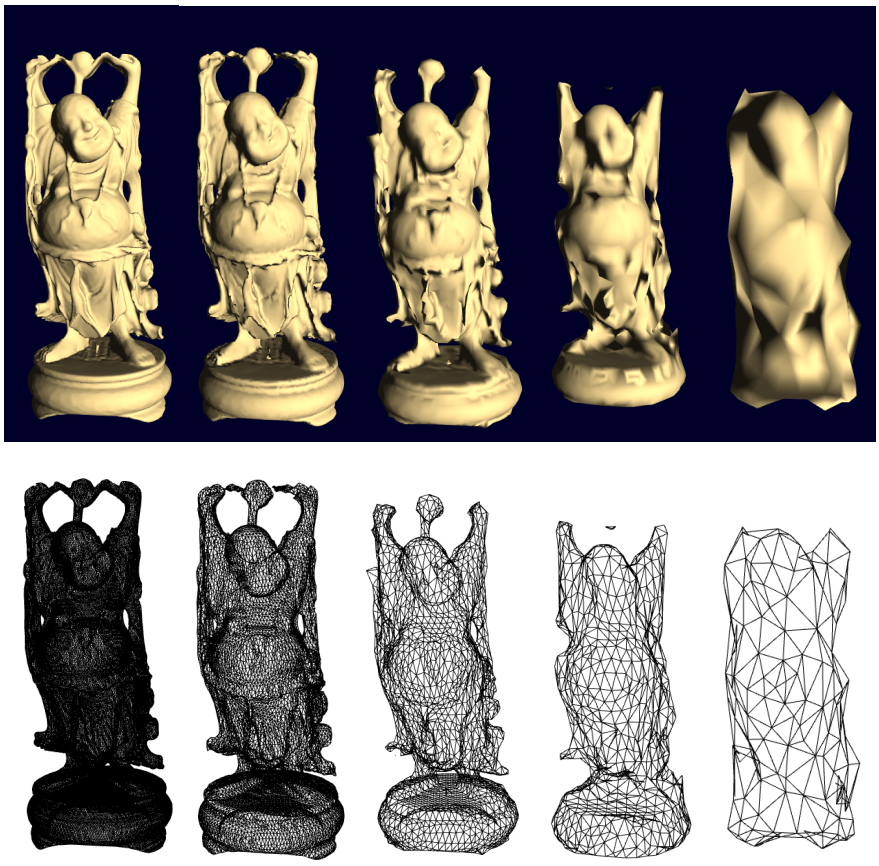

Figure 5.12: Different level of details of the Happy Buddha shaded (top row) and wire frame (bottom row)

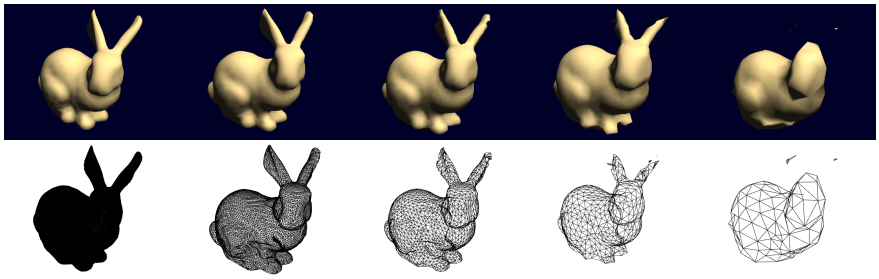

Figure 5.13: Different level of details of the Stanford Bunny shaded (top row) and wire frame (bottom row)

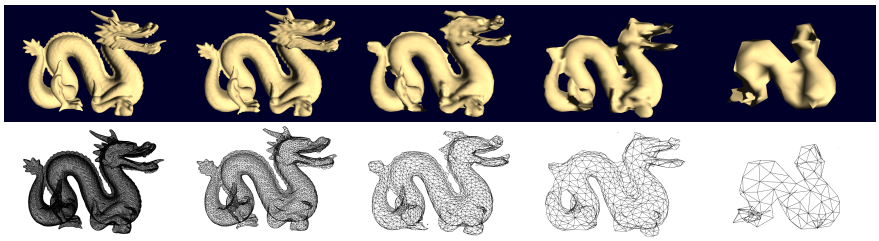

Figure 5.14: Different level of details of the Stanford Dragon shaded (top row) and wire frame (bottom row)

An advantage of this method is that it can also improve the quality of the mesh. The Stanford Dragon for instance comes in with degenerated triangles and relatively bad triangle quality. The mesh generated by the polygonizer and the relaxation produces good quality meshes which are close to the original shape (Figure 5.15).

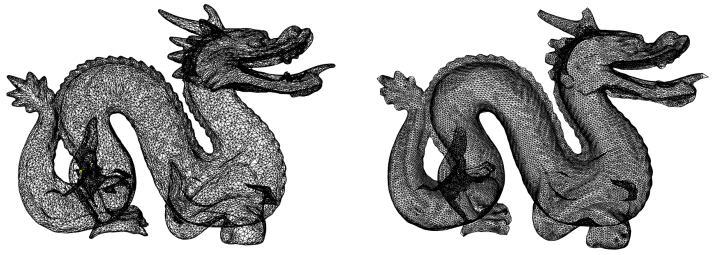

Figure 5.15: The original Stanford Dragon mesh (left) and after polygonization (right), both have a similar polygon count

The polygonizer also indirectly fills holes. The Stanford Bunny has four holes at the base. The polygonizer successfully fills three of the four holes (Figure 5.16).

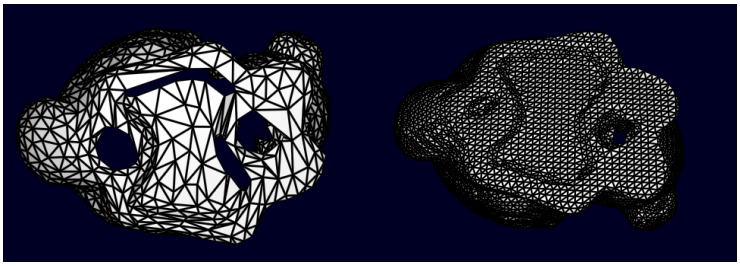

Figure 5.16: The original Stanford bunny mesh (left) and after polygonization (right)

5.3.2 Slices

The easiest way to visualize a field is to slice the volume along a plane, and apply colours to the values. In Figure 5.17, red is used to symbolize greater than zero and blue for less than zero. In between, a gradient where pure green is exactly zero.

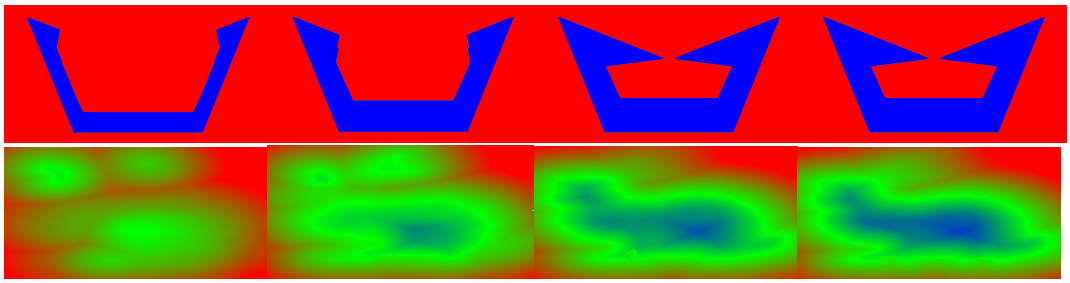

Figure 5.17: Slices of the bunny sign breaking shape (first row) and the bunny (second row)

Slices are good for debug purposes or understanding the field behaviour clearly. More advanced transfer functions which map the values to a colour can improve the understanding. But to visualize the zero-level and provide more realistic visualization another method is required.

5.3.3 Ray-marching

Ray-casting is a common way of rendering scalar fields. Rays are casted from the eye towards the volume. The ray is marched through the volume. Samples are taken along the ray to test whether it is inside or outside the volume. When a sample is found to be inside, the ray has hit the surface and it needs to be shaded. See Figure 5.18.

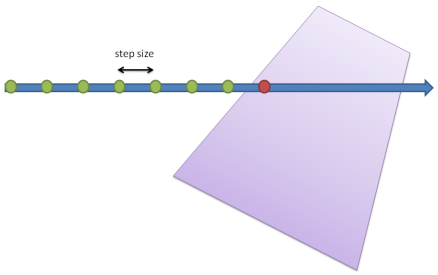

Figure 5.18: Ray-marching

Using the two values sampled above and below the surface, a refinement process can be done to get a more accurate position. The refinement will converge to the actual intersection between the surface and the ray. Many numerical methods for faster location of the intersection can be found. The most common is a simple bisection search but successive linear interpolations may be used to converge faster.

To shade the point correctly, a normal is required. The normalized gradient of the field can be used as a normal. To numerically approximate the gradient, the function is evaluated around the point.

$$G_x = f\left(\overrightarrow{P} - \overrightarrow{\triangle_x}\right) - f\left(\overrightarrow{P} + \overrightarrow{\triangle_x}\right)$$
$$G_y = f\left(\overrightarrow{P} - \overrightarrow{\triangle_y}\right) - f\left(\overrightarrow{P} + \overrightarrow{\triangle_y}\right)$$
$$G_z = f\left(\overrightarrow{P} - \overrightarrow{\triangle_z}\right) - f\left(\overrightarrow{P} + \overrightarrow{\triangle_z}\right)$$

where $\overrightarrow{\triangle_x} = (h, 0, 0)$, $\overrightarrow{\triangle_y} = (0, h, 0)$ and $\overrightarrow{\triangle_z} = (0, 0, h)$, h are small values. A small value in this context is a value small enough to not miss the small features, and large enough to avoid numerical instabilities.

The classic ray marching algorithm only relies on the sign of the field to go through the volume. A major improvement can be done by using the sampled value as a step-size. The field value is the shortest distance to the field so the surface is at least that far from the sampled position. See Figure 5.19.To be able to reach the actual surface, the step size is the maximum between the distance field value and a minimum step-size. Once the sample is negative, the bi-section search can be used to get an accurate value.

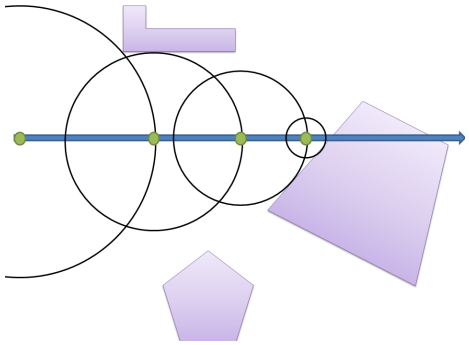

Figure 5.19: Ray-marching using distance information

Still relying on the properties of the field, some effects can be achieved very efficiently (Quilez 2008). Ambient occlusion can be done by sampling only a few positions along the surface normal. Each sampled is compared against the expected value. The expected value is the distance between the position and the sample position along the normal. If no occlusion happens then both values will be identical. Figure 5.20 shows the results obtained with the bunny.

Using the same process soft-shadows can be done by sampling along the

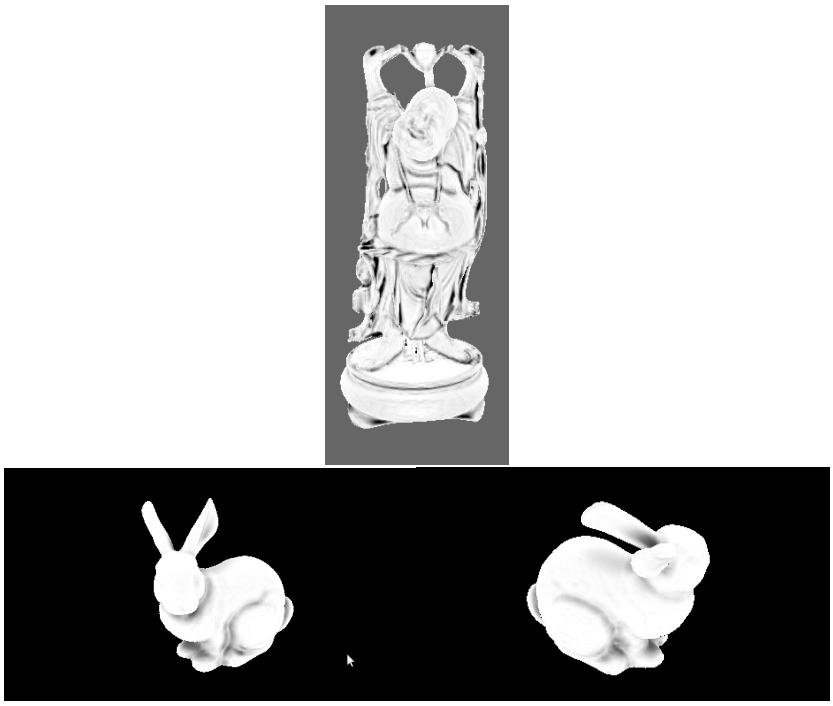

Figure 5.20: Ambient occlusion

segment between the point and the light.

Extending on the ambient occlusion idea, sub-surface scattering can also be achieved by sampling along the ray inside the volume. The further away from the surface, the lesser the weight. Finally the furthest away from the surface the more it is obstructed.

5.4 Physics

Distance fields can be used to detect accurately a collision and provide all the necessary information to correctly react to the collision. For this application the project done for CGI Techniques, a cloth simulation, was used for a quick proof of concept. Fuhrmann et al. (2003) used signed distance fields to collide cloth with meshes in a more efficient way.

For the simplest solution, the signed distance field is discretized over a grid. During the simulation, each particle is tested against the field. If a position returns a negative value, then the particle is inside the object. To improve the collision results, any value less than a certain positive threshold returns a collision event. The value is dependent on the model and avoids the cloth material in between the particle to go through the object because it is not tested.

To react to the collision, the collision position and the normal are required. Successive linear interpolations between the last particle position and the current can be used to get a value closer to the surface. The normal of the collision point is simply its gradient which has already been introduced in Subsection 5.3.3.

To sample the discretize field, the simplest solution is to find the closest cell in the grid. A better solution often used on the GPU to sample textures is tri-linear sampling.

Figure 5.21 shows the results obtained using this method.

Figure 5.21: Cloth colliding with the Buddha (100 000 faces) and the Stanford Dragon (100 000 faces) in real-time

Because this method relies on particles, it is flexible and could be used for many other particle based systems such as fluid simulation. Distributed behavioural model could probably benefit from the signed distance fields too especially because the function returns the euclidean distance.

Chapter 6

Application design and implementation

To support this thesis, three applications have been developed in C++ around the core library. An OpenCL extension was also implemented. The library relies on C++ and boost, and the polygon error module relies in the half edge library which was written for the Personal Inquiry module.

6.1 SDF library

The library is built to be easily integrated in an application with a single facade through the template class signed_distance_field_from_mesh shown in Figure 6.1. The policy templates allow to select different behaviour for the sign computation, the look up method and the multiple queries.

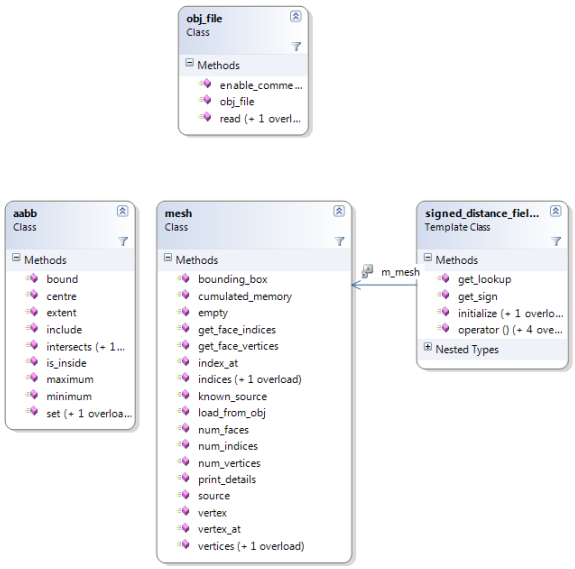

Figure 6.1: The signed distance field interface

The look up policy can be a brute force method, a regular grid, an octree and a BVH accelerated.

The sign computation could be none if only the distance is required, direct computation, or angle weighted pseudo normals.

The multiple queries allows to do multiple queries more efficiently than just successively doing many single queries. Policies are iterative, threaded and GPU accelerated through the OpenCL extension.

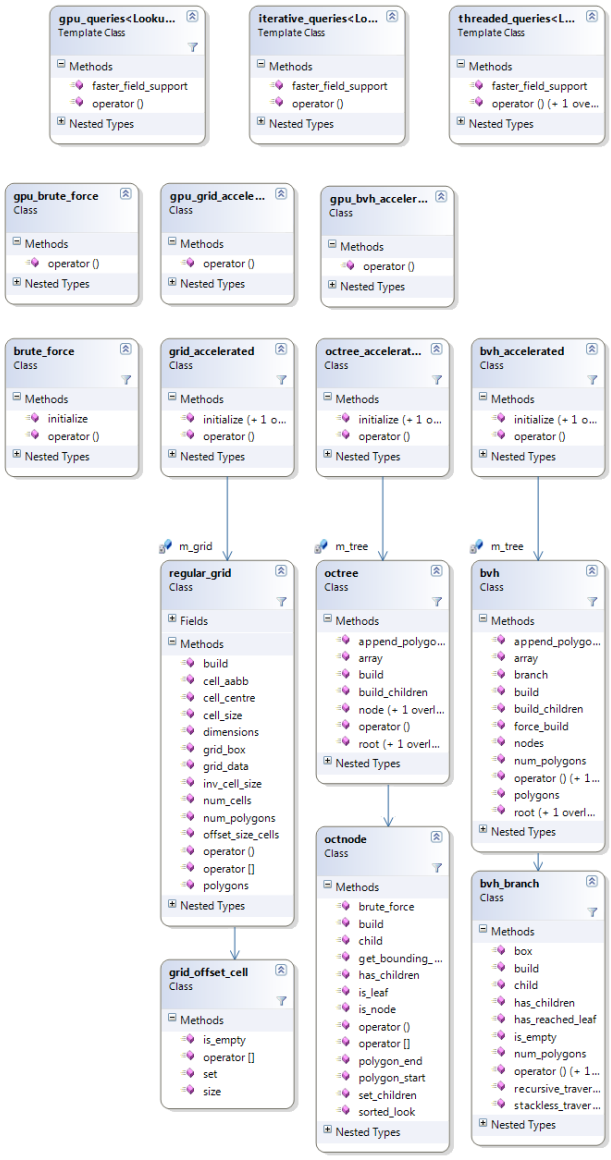

Figure 6.2: The UML diagram of the look-up policies and multiple queries policies

The diagram in Figure 6.2 shows the look up policies and multiple queries. The multiple queries policies at the top allow to sample a discrete field or arbitrary points in space. The iterative queries will simply do then one after the other while the threaded queries will split the work over a few threads. Finally, the GPU queries select the right GPU implementation according to the look-up

method. By default it will select a GPU brute force, but if it is a grid or a BVH it will then get the appropriate GPU implementation and use those acceleration structures on the GPU. BVH trees use a caching system to avoid building the trees when the mesh did not change. If the mesh is from a known source, it will generate the cache file with the "BVH" extension where the source file is. If no source can be identified, then no cache reading or writing is performed.

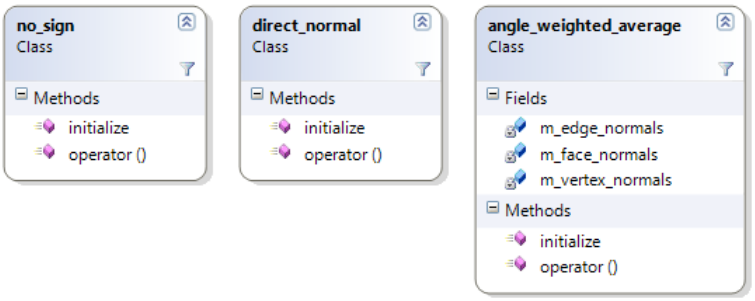

Figure 6.3: The sign policies for the signed distance field

The diagram in Figure 6.3 shows the sign policies. The two first ones were just used during development, but the angle-weighted average's only drawback is memory. The angle weighted pseudo normals will use cache files if the source of the mesh is identified. When a mesh is loaded from an obj, it sets the mesh to a known source. When the sign object is initialised, if the cache for the angle weighted pseudo normals exists then it will load it, otherwise it will generate the normals and cache them on disk where the source is with the "awn" extension. If the source is not known, no file reading or writing will happen.

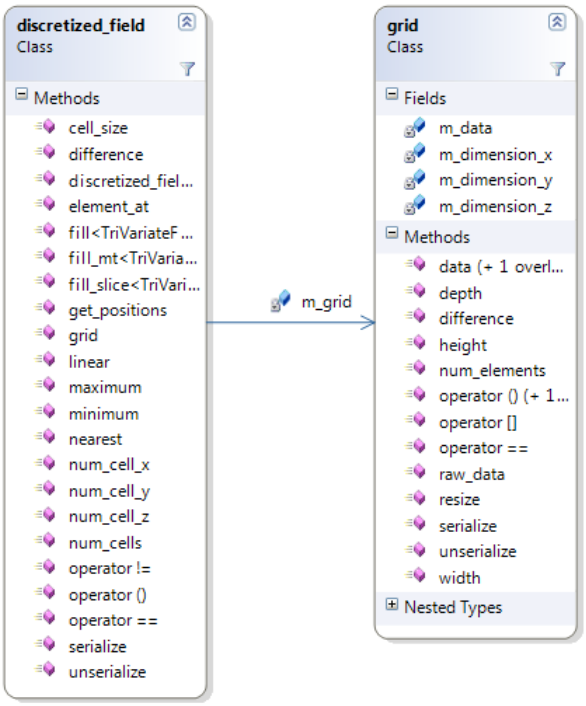

Figure 6.4: The discretized field classes

A discretized field is a regular grid which holds values for each cell. This class shown in Figure 6.4 provides a simple interface that allows to fill it by any tri-variate function. This allows to fill a field with the signed distance field from a mesh, or more advanced functions like blended unions or micro-structures. The fill operation is multi-threaded if possible.

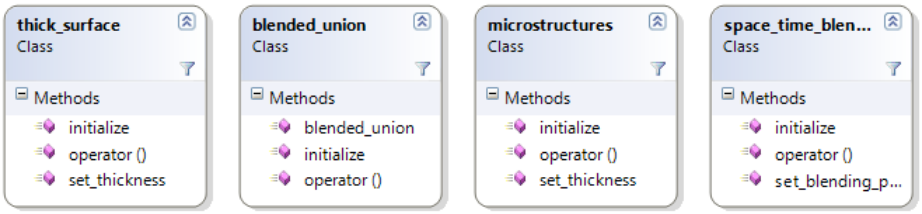

Figure 6.5: Some basic operations with distance fields to be done on the CPU

Some basic operations were implemented on the CPU (Figure 6.5). They overload the operator() so that it can act as a functor and be used by template functions when filling a discrete field.

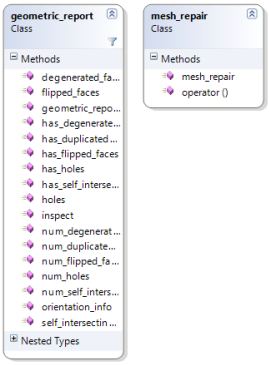

Figure 6.6: The two error related classes for detection and correction

Finally geometry errors are handled by two classes. The geometric report inspects the mesh to report errors, and the mesh_repair object fixes the mesh. In the future the mesh_repair class should have more features to enable options and set-up correctly how errors should be handled.

6.2 Demo application

The demo application uses OpenGL to display models in different ways. It loads meshes and discretized volumes and allow to blend them, morph them and fill them of micro-structures on the fly in the fragment shaders. This application is mainly a test bed for the core library and visualize volumes. It also shows ambient occlusion and other effects achieved thanks to distance fields. It can also display the polygonal mesh and the field slices. A screenshot of this application is shown in appendix B.1.

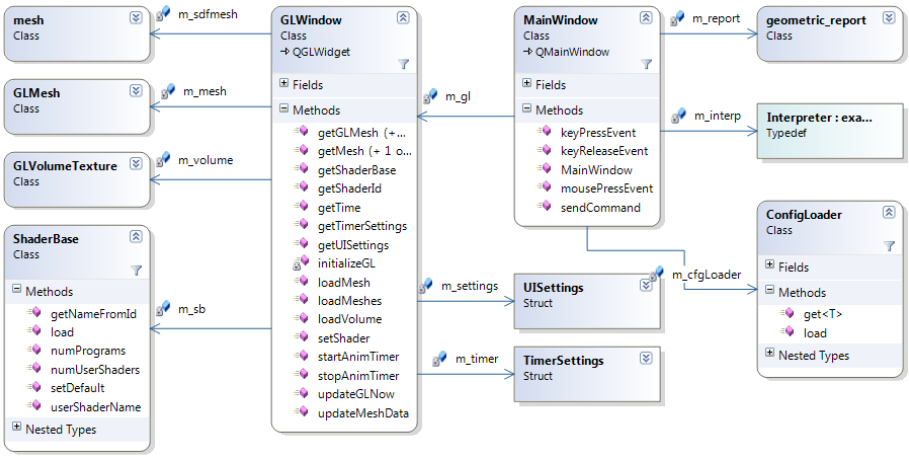

Figure 6.7: The basic class diagram of the SDF-PM application

The Figure 6.7 shows the class diagram of the application. The MainWindow contains all the user interface elements through the Qt interface. The application was originally built around a console where the logger is. The interpreter was calling the appropriate functions given a command to parse. The ConfigLoader loads the initial settings for the application, and the geometric_report is part of the SDF library and handles geometric errors while loading meshes. The GLWindow is responsible for the rendering window. The UISettings and TimerSettings are updated by the MainWindow according to the user interactions. The ShaderBase loads the shader library from a file. GLWindow renders accordingly to the UI settings.

The application features:

- Loading mesh a discretize them on the fly using the best available method
- Loading cached discretized fields
- Transition between models using a cutting plane as shown in
- Linear morphing as shown in
- Space time blending as shown in
- Microstructures as shown in
- Visualize the mesh as a polygon mesh, using the accelerated ray-caster and the volume data, slices.
- Ambient occlusion using signed distance fields
- Inspect meshes for geometrical errors and fix them except self-intersections

6.3 Other applications

A few more applications were built for debugging purposes or comparison for this thesis.

- An application was built to compare different techniques with different data sets. It also tests different settings and writes them into a log file.
- An application generates discrete fields using command line arguments and configuration files.
- A legacy application written for DirectX to visualize the wireframe model and get the distance to the field as well as a the closest point to the mesh. It also displays the tree structures, flipped faces, self-intersecting faces and holes. A screen shot can be seen in Appendix B.2.
- The level of details application (asd-ism) is a program that uses the ism library developed for ASD, and the sdf library developed for this thesis. It generates an object file.
- The cloth simulation developed for CGI Techniques originally was edited to support collision with meshes using signed distance fields. To handle collisions it requires both the .obj file with the .bdf file.

Chapter 7

Conclusion

Signed distance field representations are powerful techniques to represent objects which allow complex structures, animation and produce good visual results. Distance fields proved to be very efficient for several applications that are usually difficult. This thesis only showed a few possible applications of the signed distance fields. Signed distance fields received an increasing interests over the last years covering a wide range of applications and fields. Further integration of FReps into existing applications seem promising and in the future there will be more application areas that will benefit from this representation. One of the remaining issue with volumetric representation and FReps is the ability to easily build models. The current domination of BReps and polygonal meshes increase this issue because the industries are not familiar with anything else than BReps. A signed distance field representation of a polygonal mesh allows to build a bridge between the two representations.

This project provides an exact signed distance field function which is continuous and handles minor errors. The library is extended by a GPGPU implementation using QtOpenCL which allows to sample the function of distance function efficiently at any point in the field, which is suitable for advanced polygonizers. The library can easily be integrated in the FRep framework allowing polygonal meshes to benefit from the FRep features. The library is flexible and portable and allowed to implement new features on various old projects easily by just wrapping the main interface in a class.

While signed distance fields are not the best representation for all purposes, a hybrid solution would allow to efficiently perform operations in the best representation.

7.1 Limitations

The input meshes have to be valid, but unfortunately it is rarely a concern of the artist. Operations on polygonal meshes can often break the boundary representation without being detected directly. The polygonal error checking

and fixing provided is too limited. Many more errors can arise from bad meshes that are not detected yet. Mesh fixing is also a large topic especially in object reconstruction from range scans. This is a large topic but the possibility of offering a few simple solutions to the user to fix holes and self-intersections is a valuable feature. Ultimately, those errors should be handled by a user because algorithms cannot understand what the user really wanted, and errors should be solved by the user to insure it is what they wanted in the first place.

The signed distance field on its own also lacks texturing capabilities. Basic procedural texturing is easily achieved but 2D texturing is a very difficult task which does not have a solution yet.

7.2 Future directions

This project could benefit from more work into optimisation. The regular grid and the octree in particular were quickly left aside because the initial results were not as good as the BVH tree. However, the traversal methods for the octree and the grid can be largely improved. The BVH tree can also be improved by optimising the cache efficiency. A node could be fit into 32 bits easily. Another possible improvement that is often used in ray-tracing is to traverse the tree with a block of coherent rays. It would be possible to query the tree for a few positions at once if those positions are close enough.

The Meshsweeper algorithm seems to be a good alternative to the usual hierarchy tree. It seems to be more difficult to implement but might generate faster results. Looking into the scanning methods and methods which work better for grids would also be a good addition to this project. Most of the time, the field is sampled regularly over a space. Even the polygonizers start from a grid, and then sample more from arbitrary positions.

In terms of applications, shattering seems to be a natural extension of the signed distance fields because it contains volumetric information. The division of the space can be done efficiently and intersected with the object.

Finally, a natural extension to the library could include the signed distance field to a parametric patch.

Bibliography

J. A. Baerentzen & H. Aanaes (2005). 'Signed Distance Computation Using the Angle Weighted Pseudonormal'. *IEEE Transactions on Visualization and Computer Graphics* **11**:243–253.

M. Botsch & L. P. Kobbelt (2001). 'A Robust Procedure to Eliminate Degenerate Faces from Triangle Meshes'. In *VISION, MODELING AND VISUALIZATION (VMV01*, pp. 2–3.

D. Eberly (2008). 'Distance Between Point and Triangle in 3D'. Available at `http://www.geometrictools.com/Documentation/DistancePoint3Triangle3.pdf` [Accessed 26 June 2011].

C. Ericson (2004). *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

C. Ericson (2007). 'Physics for games programmers: Collisions using separating-axis tests. GDC2007.'. Available at `http://realtimecollisiondetection.net/pubs/GDC07_Ericson_Physics_Tutorial_SAT.ppt` [Accessed 19 July 2011].

K. Erleben & H. Dohlmann (2007). 'Signed Distance Fields Using Single-Pass GPU Scan Conversion of Tetrahedra'.

P.-A. Fayolle, et al. (2008). 'Heterogeneous objects modelling and applications'. chap. SARDF: signed approximate real distance functions in heterogeneous objects modeling, pp. 118–141. Springer-Verlag, Berlin, Heidelberg.

S. Fisher & M. C. Lin (2001). 'Deformed distance fields for simulation of non-penetrating flexible bodies'. In *Proceedings of the Eurographic workshop on Computer animation and simulation*, pp. 99–111, New York, NY, USA. Springer-Verlag New York, Inc.

O. Fryazinov, et al. (2011). 'BSP-fields: An exact representation of polygonal objects by differentiable scalar fields based on binary space partitioning'. *Computer-Aided Design* **43**(3):265 – 277.

A. Fuhrmann, et al. (2003). 'Abstract Distance Fields for Rapid Collision Detection in Physically Based Modeling'.

N. Gagvani & D. Silver (1999). 'Parameter-controlled volume thinning'. *CVGIP: Graph. Models Image Process.* **61**:149–164.

C. Green (2007). 'Improved alpha-tested magnification for vector textures and special effects'. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, pp. 9–18, New York, NY, USA. ACM.

E. Guendelman, et al. (2003). 'Nonconvex rigid bodies with stacking'. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pp. 871–878, New York, NY, USA. ACM.

A. Guéziec (2001). ''Meshsweeper': Dynamic Point-to-Polygonal-Mesh Distance and Applications'. *IEEE Transactions on Visualization and Computer Graphics* **7**:47–61.

M. W. Jones (1995). '3D distance from a point to a triangle'. Tech. rep., Department of Computer Science, University of Wales.

M. W. Jones (2003). 'Melting Objects'. In *The Smalltalk Report*, pp. 155–158.

M. W. Jones, et al. (2006). '3D distance fields: A survey of techniques and applications'. *IEEE Transactions On Visualization and Computer Graphics* **12**:581–599.

T. Möller (1997). 'A Fast Triangle-Triangle Intersection Test'. *Journal of Graphics Tools* **2**:25–30.

A. Pasko, et al. (1995). 'Function Representation in Geometric Modeling: Concepts, Implementation and Applications'.

A. Pasko, et al. (2011). 'Procedural function-based modelling of volumetric microstructures'. *Graphical Models* **73**(5):165 – 181.

G. Pasko, et al. (2004). 'Space-time blending'. *Journal of Computer Animation and Virtual Worlds* **15**:109–121.

B. A. Payne & A. W. Toga (1992). 'Distance Field Manipulation of Surface Models'. *IEEE Comput. Graph. Appl.* **12**:65–71.

M. Pharr & G. Humphreys (2004). *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

S. Popov, et al. (2007). 'Stackless KD-Tree Traversal for High Performance GPU Ray Tracing'. *Computer Graphics Forum* **26**(3):415–424. (Proceedings of Eurographics).

I. Quilez (2008). 'Rendering worlds with two triangles'. Available at `http://www.iquilezles.org/www/material/nvscene2008/nvscene2008.htm` [Accessed 17 July 2011].

J. Ratcliff (2006). 'Code suppositry'. Available at `http://codesuppository.blogspot.com/2006/04/welding.html` [Accessed 06 August 2011].

A. Rosenfeld & J. L. Pfaltz (1966). 'Sequential Operations in Digital Picture Processing'. *J. ACM* **13**:471–494.

A. Sourin, et al. (1996). 'Using Real Functions with Application to Hair Modelling'.

Appendix A

Data sets

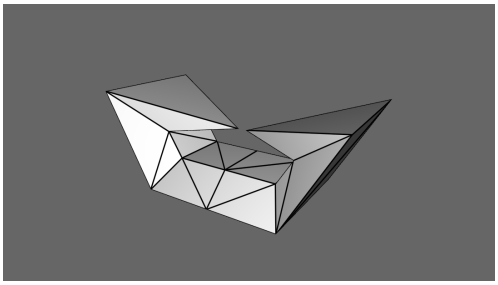

Figure A.1: The sign-breaking mesh. Modelled with very sharp features and close features to break the sign computations. 40 faces, no holes, no self-intersection.

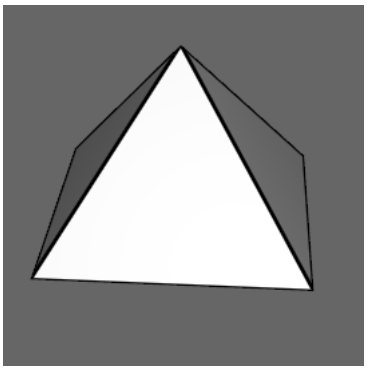

Figure A.2: A pyramid without a bottom face. The way the hole is set up creates and infinite zero shape. 4 faces, one hole, no self-intersection.

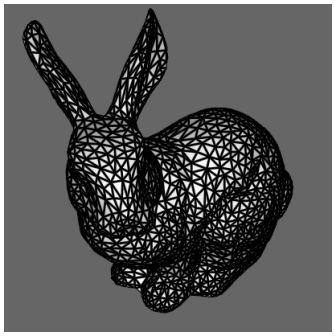

Figure A.3: The Stanford bunny, from the Stanford 3D scanning repository. 4968 faces, three holes at the bottom, no self-intersection.

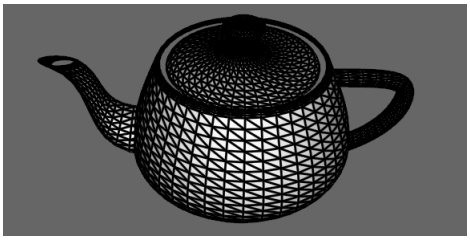

Figure A.4: The Utah teapot by Martin Newell. 6320 faces, many holes, many self-intersections, duplicated vertices.

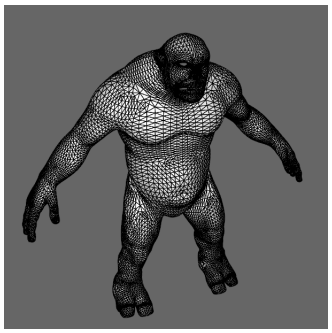

Figure A.5: The NCCA troll, 36512 faces, many holes, no self-intersections

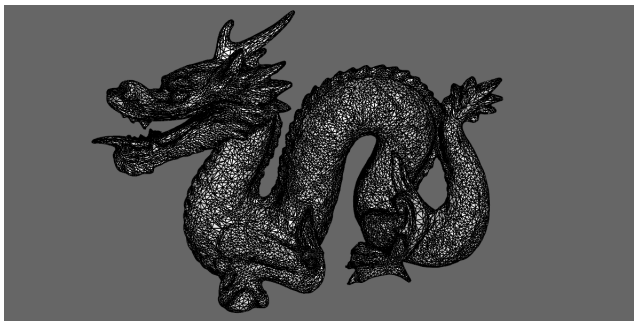

Figure A.6: The dragon from the Stanford 3D scanning repository. 100 000 faces, no holes, no self-intersections.

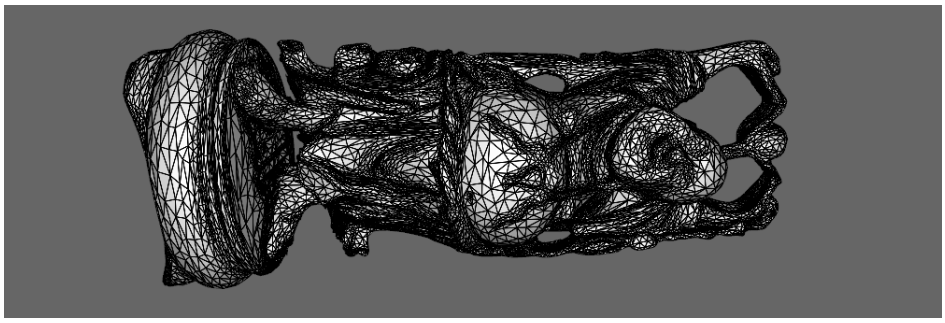

Figure A.7: The dragon from the Stanford 3D scanning repository. 100 000 faces, no holes, no self-intersections, a few degenerated triangles.

Appendix B

Applications

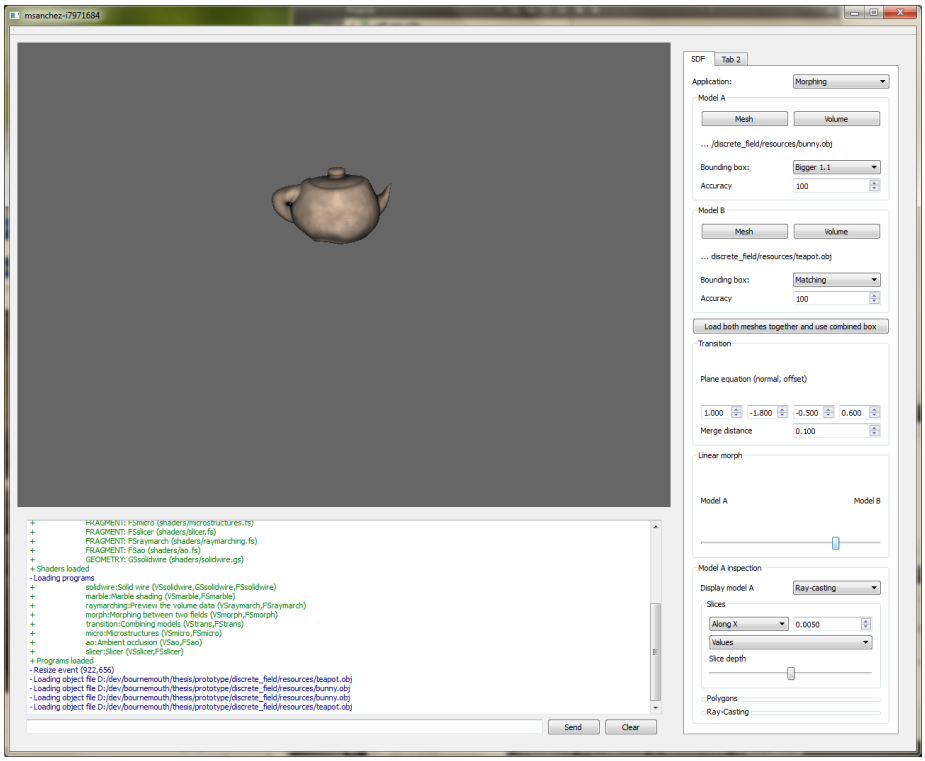

Figure B.1: The sdf-pm application used to test possible applications of the signed distance field.

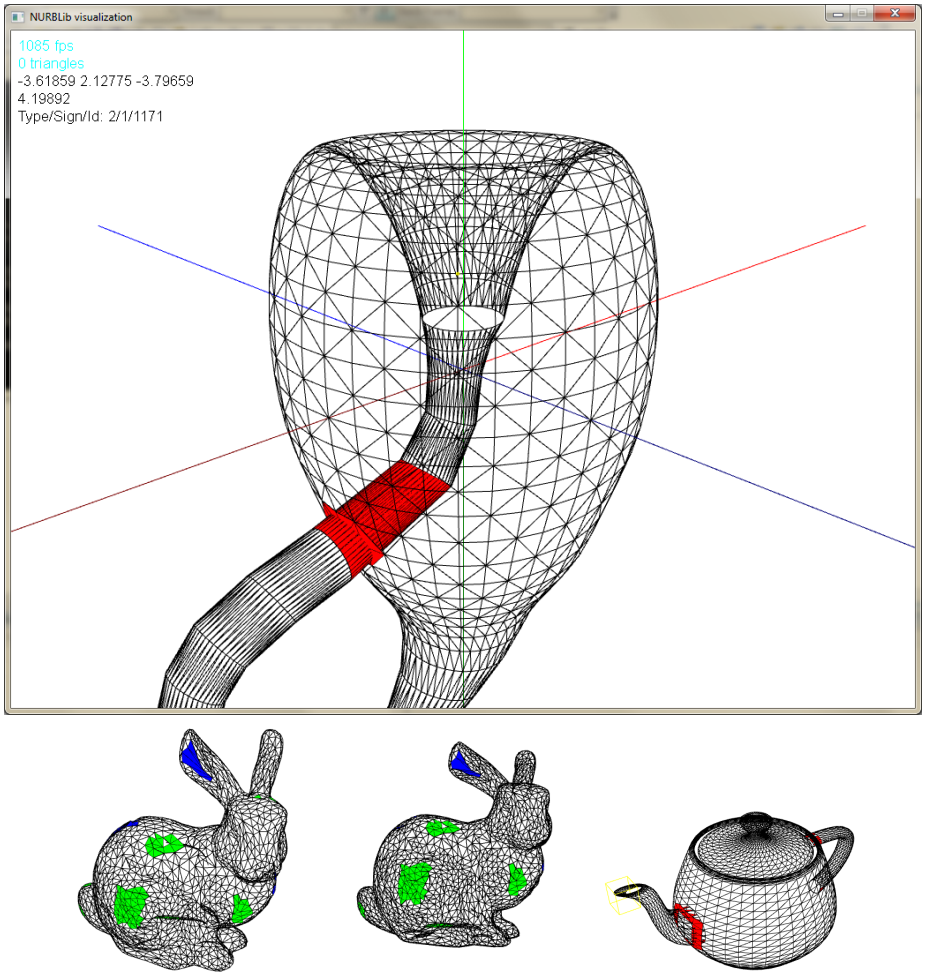

Figure B.2: Using a DirectX framework developed a few years ago. It displays the mesh in wireframe and shows the closest point on the mesh using the brute force and the acceleration structure. It also displays mesh defects with colored faces (red is self-intersecting, blue is fixed hole, green is flipped face.