Ray Tracing Implicit Surfaces on the GPU

G. Liktor

Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
Budapest / Hungary
gabor.liktor@gmail.com

Contents

Abstract:

In this paper we propose a GPU solution to render implicit surfaces at high frame rates. Ray tracing the implicit model has several benefits as opposed to processing tessellated meshes, but also invokes new kinds of problems. The main challenge is efficiently finding the first intersection point with a surface that is only implicitly defined. Our implementation uses the sphere tracing algorithm to attack this problem. We also discuss secondary issues like shading and texturing implicit models.

Keywords: Sphere Tracing, Implicit Surface, Distance Transform, Lipschitz Function, Texturing
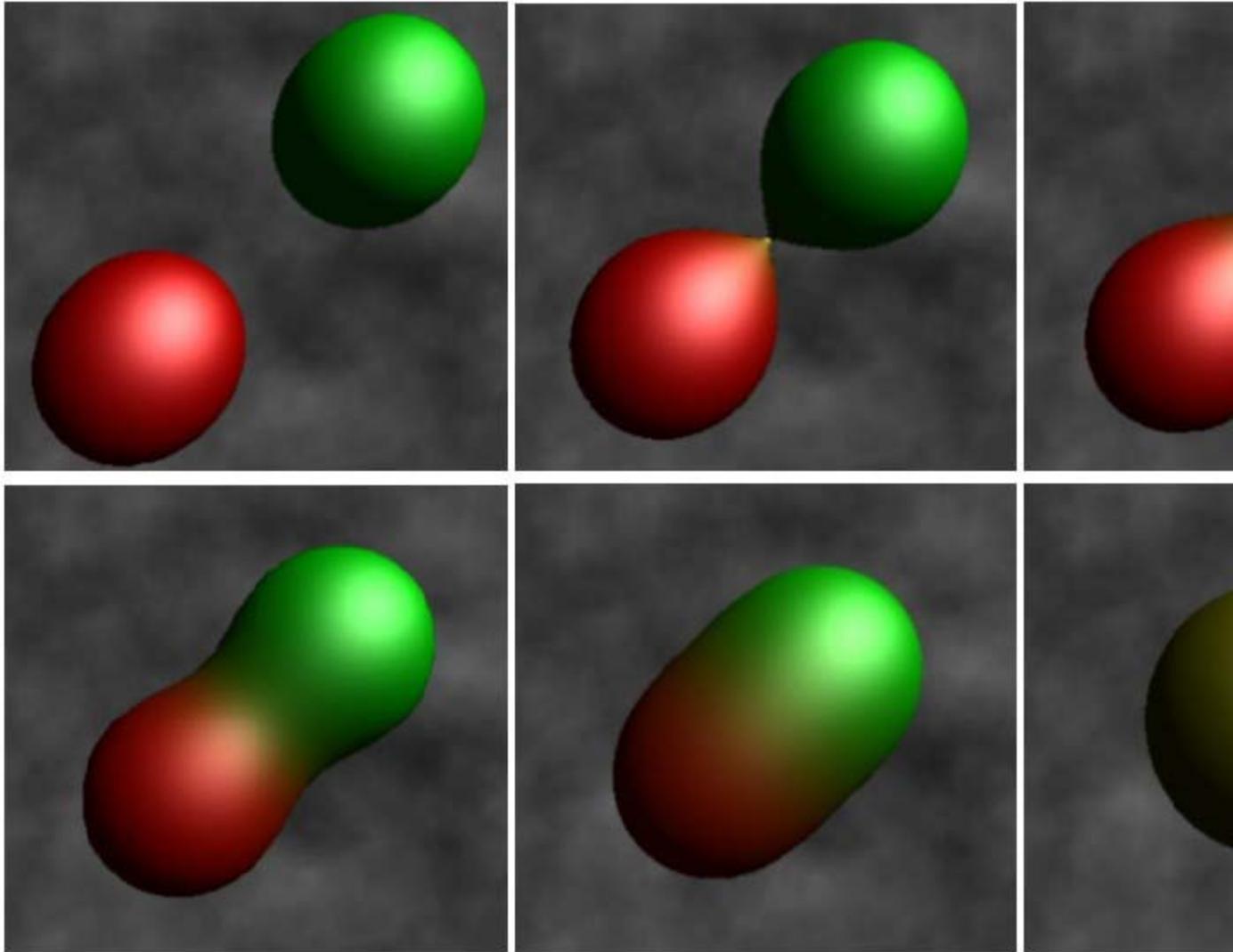
Figure 1: *Snapshots from the fusion of two blob objects. The images were rendered with our application in real time, without tesselation. The sphere tracing algorithm exploits the analytical properties of the implicit surfaces to decrease the necessary ray march steps.*

1. Introduction

The ability of handling volumetric models makes implicit surfaces very suitable for modeling natural phenomena and scientific visualization. Unlike most common methods of geometric modeling based on surfaces - polygonal surfaces or parametric surfaces - implicit modeling is a pure volumetric approach.

This section gives a short summary on the popular methods, and describes why we have chosen the ray tracing for further research. Section 2 presents the sphere tracing algorithm used to safely find the first ray-surface intersection. Section 3 improves performance by introducing a preprocessing stage, Section 4 covers shading and texturing, because

texturing an implicit surface is not trivial. Finally Section 5 describes the implementation of our solution.

An implicit surface is defined by a three-dimensional scalar function f(x,y,z), which can be evaluated at each point in space. The surface is the set of points where *f(x,y,z)=0*. To define the inside of the body, usually the *f(x,y,z) < 0* region is selected.

The main advantage of implicit surfaces is their power to describe organic objects. The geometric details of such models can be represented using less implicit primitives than required by other methods. Unfortunately, visualizing an implicit surface is a non-trivial task. During rendering we look for the point which can be seen through a pixel. While one can quickly determine if a given point is on the surface, the ray intersection point generally cannot be obtained in a simple form.

For this reason it can be useful to approximate the implicit surface with a triangle mesh before image synthesis (tessellation). We can mention the widely used marching cubes algorithm as an example, which evaluates the function at sample grid points, then produces a polygonal surface as a result. By using adaptive tessellation we can improve the accuracy of the resulting mesh where the curvature of the surface is too high, as described by Bloomenthal in [1].

The advantage of the tessellation is that it converts the implicit model according to the requirements of the conventional rendering pipeline. Nevertheless, it gives only an approximation of the original surface. To get correct results, the modeling function must be evaluated in huge number of sample points. This is especially a crucial problem when the object changes dynamically over time, requiring the tessellation to be repeated for each frame.

During design time, particle based visualization can be a good compromise [1]. In this case only the geometric shape is shown, but rendering speed must be at least interactive. Here the modeling system shows the surface by scattering particles on it. These particles can start, for example, on the bounding sphere of the object, and can follow the gradient field until the surface is hit.

The other main method of rendering implicit surfaces is ray tracing. In this case tessellation is needless; ray tracing locates the intersection of rays from the camera (through pixels) and the original surface. Besides reducing the overhead it has another advantage: it automatically provides smooth *level of detail* (LOD). Objects nearer to the camera occupy more pixels than farther ones, so they receive more rays. Polygonal methods can achieve smooth LOD only by changing the mesh resolution depending on the distance. This property can be ideal for terrains and many similar applications where dynamic LOD is needed. And at last but not least, ray tracing is a highly parallel procedure. It can be done for every pixel independently, so we could implement it in the pixel shader of the GPU to achieve high rendering speed.

Ray tracing of analytically defined surfaces can be more efficient if one can exploit the special properties of a given family of implicit functions. As an example, Figure 1 shows analytically ray traced blob objects. In a recent work Kanamori et. al.[8] introduced a robust method for displaying surfaces of thousands of metaballs in real time on the GPU. Their solution utilizes the finite support areas of the metaballs' density functions, and by replacing the metaballs with their outer spheres, they can determine the relevant components of the surface for each ray with rasterization and depth peeling. In [9] Reimers and Seland proposed a ray-casting solution for algebraic surfaces. For this other important class of analytical functions they speeded up the intersection finding by transforming the surface polynomial into a so called frustum form, which adapts to the view frustum. The frustum form then provides a more efficient ray parametrization for root finding.

In this paper we describe a method for rendering general implicit surfaces. We use the Lipschitz property for numerical root solving, which can be applied to large family of implicit functions. We also extend the same algorithm with a preprocessing step for the ability to handle complex and non-Lipschitz implicit models as well.

---

2. Sphere Tracing

2.1. The root finding problem

The ray tracer shoots rays from the eye position through the centre of each pixel. A ray is described by the following equation:

$$\vec{r}(t) = \vec{r}_0 + \vec{v}t, \, t \in (0, \infty).$$

Where $\vec{r}_0$ is the eye position, $\vec{v}$ is the ray direction. The intersection of the ray and the surface is the solution of the $f(\vec{r}(t)) = 0$ relation. So ray tracing depends on finding the root of this equation. While ray parameter t generally cannot be expressed from the equation above, root finding is solved by some iterative method.

During the so-called *ray-marching* we make small steps along the ray, starting from the eye position. Evaluating the function at each visited point, the intersection occurs when it changes sign. However, one can always find an "evil" function for any step size, so that this function changes sign two times between two steps. Such evil function represents a "thin" surface the ray-marching steps over (See Figure 2). Therefore, our root finding algorithm must also take into account some analytical properties of the implicit function, only evaluation is not enough.
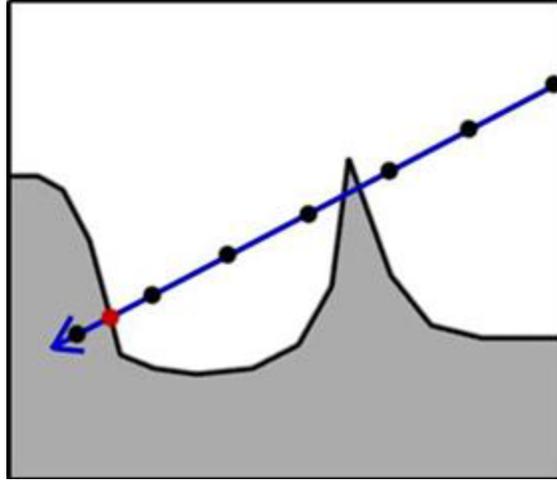
Figure 2: *A ray misses the first hit by stepping over thin geometric details*

Several intersection finding algorithms are discussed in [2] which are suitable mostly for displacement surfaces, but some of them can also be used for general implicit rendering as well. The simplest one is the linear search, which takes fix steps along the ray until the function first changes sign. After that we have a reference point in the body and another reference point outside the body, so we can refine our search by a numerical root finding algorithm. Linear search is safe if step size *dt* is small enough.

However, this method uses too small knowledge of the ray-traced function. If *dt* is too small, then the function must be evaluated many times, which makes it slow. If *dt* is greater, then finding the first ray hit is not guaranteed. Let us call *dt* a *safe step size*, if

$$\text{sgn}(f(\vec{r}(t))) = \text{sgn}(f(\vec{r}(t + dt'))) \text{, } dt' < dt$$

so the sign of the function remains the same along the step. We can get a better searching algorithm if we can give a proper estimation for such a safe step size. With other words we are looking for a $d(\vec{r})$ function which gives at every $\vec{r}$ point the largest possible *dt*.

Sphere tracing was proposed first time by John C. Hart [3] as a ray tracing algorithm for implicit surfaces. In the following subsection we define the Lipschitz property, and show how to use sphere tracing for functions with such property.

2.2. The sphere tracing algorithm

Let us introduce some definitions first. We say that the real function *f* is *Lipschitz* over domain *H* (Lipschitz continuous), when there is a non-negative real *L* so that

$$|f(x) - f(y)| \le L|x - y| \text{, } \forall x, y \in H$$

This means that the steepness of a sector of two points cannot be larger than a bound. The value of *f(x)* cannot change more than *L* in a unit distance. The smallest *L* is called the Lipschitz constant.

The $f_d : \Re^3 \rightarrow \Re$ function is distance bound by its implicit surface $f_d^{-1}(0)$ if

$$|f_d(\bar{x})| \le d(\bar{x}, f_d^{-1}(0)),$$

where $d(\bar{x}, f_d^{-1}(0))$ is the distance between point $\bar{x}$ and the surface. This is important because if we have found such a distance bound function, we can estimate the distance to the surface, i.e. the safe step size. In [3] Hart proves that every function with a Lipschitz constant can be turned into a distance bound function.

If we have found a distance bound function for the ray traced surface, then the safe *dt* step size along the ray is at least $f_d(\bar{r}(t))$. This is the essence of the sphere tracing. The distance function defines "unbounding spheres" in the space, i.e. spheres with radius small enough not to intersect the surface (Figure 3).

With the distance function the ray-marching can be performed as follows:

```
t = 0
f = f(origin + direction*t)
while (t < D){
        dt = f(origin + direction*t)
        if(dt < epsilon)
                return t //there is intersection
        t = t + dt
}
return -1 //no intersection
```

Listing 1: *The pseudo code of sphere tracing*

The method converges only if the ray has an intersection with the surface so we must give a maximum *D* bound for *t*. Root finding is the least efficient at the contours, because there the rays fall almost parallel to the surface, so the distances will be small.

Of course, sphere tracing can be useful only if the practically used analytic functions can be turned into distance bound functions. In his paper Hart proved it for many frequently used modelling functions, such as standard geometric primitives, soft bodies, etc. For details see [3].
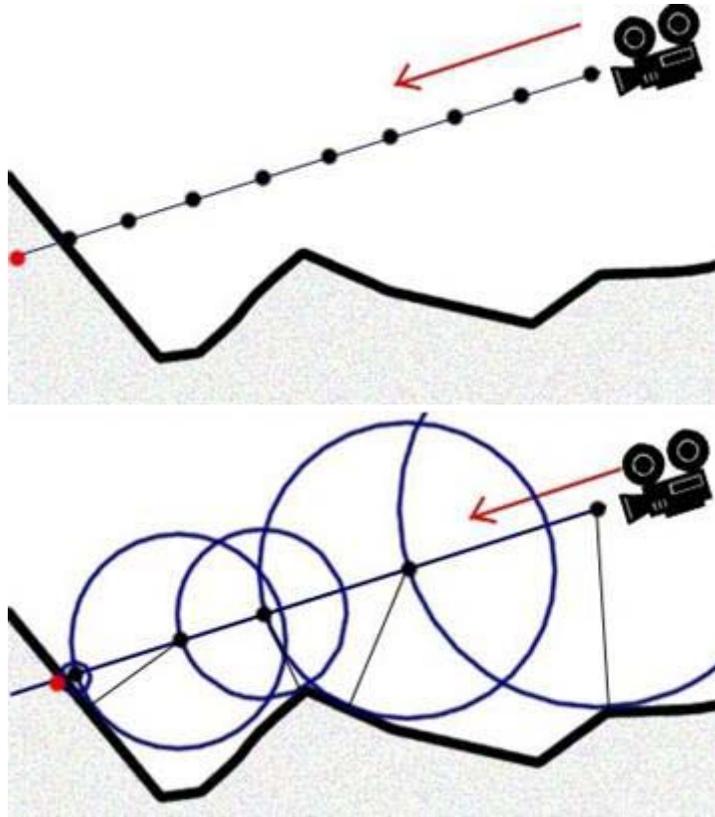
Figure 3*: Comparison of the principles of linear search and sphere tracing.*
Sphere tracing reduces step numbers by estimating distances from the surface.

We can apply sphere tracing to CSG (Constructive Solid Geometry) models if we can interpret the CSG operators on distance functions as well (Figure 4). For example:

.   Union:

$$d(\bar{x}, A \cup B) = \min\{ f_A(\bar{x}), f_B(\bar{x}) \}$$

This is logical, for the distance to the union of two bodies can be at most the distance to the nearest one.

.   Complement:

$$d(\bar{x}, \mathfrak{R}^3 \setminus A) = -f_A(\bar{x})$$

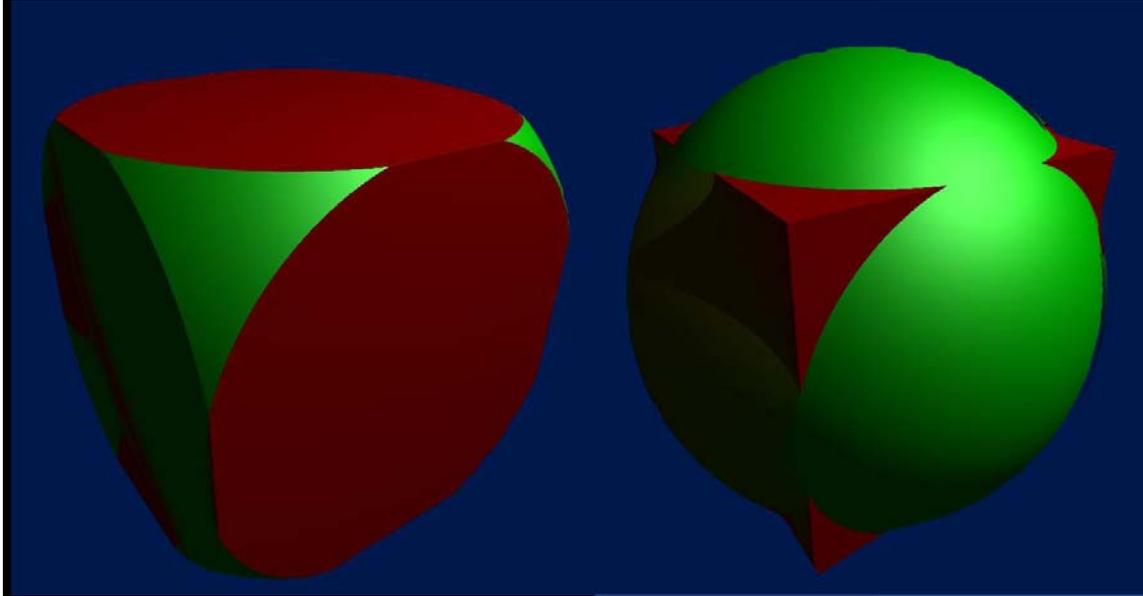.   Intersection: $d(\bar{x}, A \cap B) \geq \max\{ f_A(\bar{x}), f_B(\bar{x}) \}$

Figure 4*: an example of CSG modeling with simple implicit objects*

Transformations can also be handled. If the transformation is isometric, only the inverse transformation must be applied to the parameters of the *f* distance function. In general, if the transformation is the *T* operator, then the distance function of the transformed surface is $f_s(T^{-1}(\vec{x}))$. According to the chain rule, the Lipschitz constant of the composition of two functions cannot be greater than the product of the two Lipschitz constants. So we get the Lipschitz constant of the transformed function by calculating the Lipschitz constant of the inverse transformation, and multiply it with that of the original distance function's.

---

3. Preprocessing

3.1. Distance transform

Until now we estimated distances from the surface only analytically. This narrowed the field of application of sphere tracing to a special class: Lipschitz functions. It is proved that most of the primitive functions used in CSG modeling can be turned into a Lipschitz function, but there are other applications as well. We would also like to use sphere tracing for general implicit functions. On the other hand, calculating the distance function can be quite costly, especially when our model consists of several primitive functions (like a fluid with hundreds of blobs). Sometimes we cannot afford to calculate the distance function in every step. In [4] Donnelly presents an implementation of sphere tracing displacement maps using preprocessed distance maps. We followed a way similar to that algorithm.

The first stage of the preprocessing is sampling. According to a 3D grid, we divide the bounding volume (AABB - Axis Aligned Bounding Box) of the surface to small volume

elements called *voxels*. In each voxel we calculate the function in a representative point. Storing these values in a 3D array we have got a discrete volumetric model of the original implicit object. Choosing the proper sampling frequency is important, because we must find the golden mean between undersampling and the computational and storage overhead. In real time, when we want to store data in the memory of the GPU, we cannot afford large data size. In the case of a single channel floating point texture, even a 256^3 sized texture fills 64MB from the texture memory.

Now we encode the object in a 3D array: if the sample in the voxel was outside the body, then we store 1, otherwise we store 0. If we can find for every voxel the nearest one with value 1 and store the distance between them, then we have got the radius of the greatest sphere which does not intersect the surface. The resulting distance array is called distance map, and the procedure described in the previous paragraphs is a distance transform. Figure 5 illustrates the derivation of a basic distance field from a sampled geometry.

Having completed the distance transform, we can visualize the original surface with sphere tracing, since we know the distance to the nearest point of the surface at each point of the discretized space. Passing along the ray, we do not even need to compute the distance function, *dt* can be read directly from memory. This way the complexity of ray tracing gets independent of the complexity of the original function.
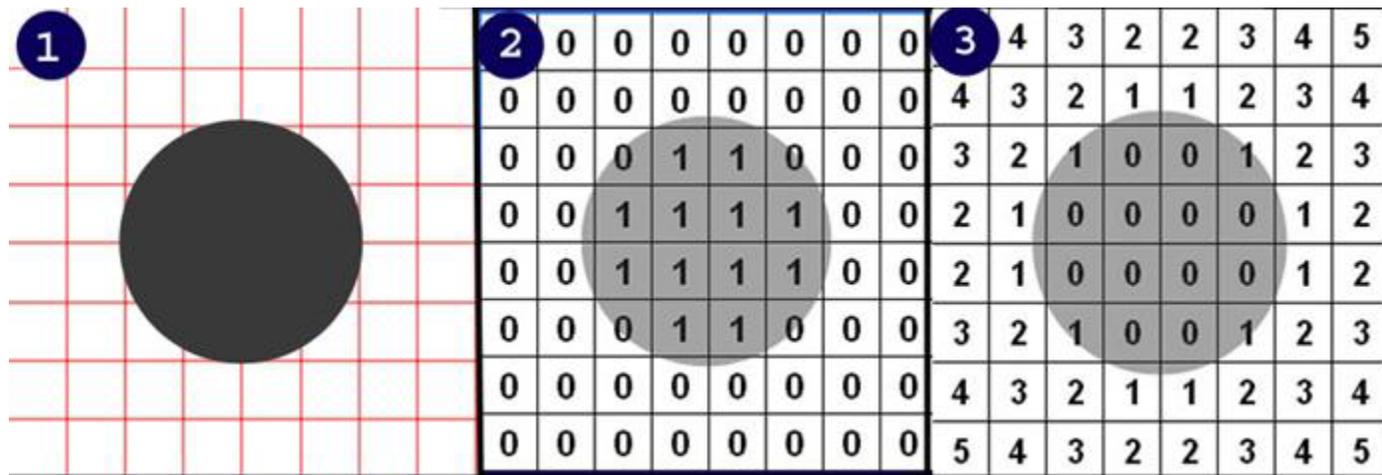

Figure 5: *The idea of the distance transform demonstrated in two dimensions. In this example we applied the Manhattan metric, in which the result is the sum of the horizontal and vertical distances of the points.*

3.2. Danielsson's algorithm

For the computation of the distance map we used the Danielsson's algorithm ([4], [5]). Making use of the dynamic programming, it gives approximately right results for the Euclidean distances. It can be ranked among the fastest algorithms for having linear complexity (*O(n)* for *n* samples). In three dimensions the size of the samples is proportional to the cubic of the sampling frequency.

Here, for illustration we describe the two dimensional variant. The input is array T with 0/1 items encoding the object which we can take as a black and white image. The output is array T^{dist}, which has a same size, but consists of vectors pointing to the nearest point of the body. Initially we set:

$$T^{dist}(i,j) = \begin{cases} [0,0], & \text{if } T(i,j) = 1 \\ [\infty,\infty], & \text{if } T(i,j) = 0 \end{cases}$$

The algorithm is based on the recognition that if we know the nearest surface points to the neighbors of a given point p, then we can infer the nearest surface point to p from that. In the first pass we slide a "window" from the upper left corner to the lower right corner by passing along the image row by row. The vector belonging to the current point is produced of the vectors of the left and top neighbours:

$$T^{dist}(i,j) := \min \begin{cases} T^{dist}(i,j); \; T^{dist}(i-1,j) + [-1,0]; \\ T^{dist}(i,j-1) + [0,1] \end{cases}$$

By the end of this pass, the vector in the lower right corner holds the final value. But for other points we did not consider the right and bottom neighbours, so the second pass of the algorithm runs backward, reversing the "window":

$$T^{dist}(i,j) := \min \begin{cases} T^{dist}(i,j); \; T^{dist}(i+1,j) + [1,0]; \\ T^{dist}(i,j+1) + [0,-1] \end{cases}$$

Each item was visited two times, and the procedure finds approximately the right distances. See Figure 6 below illustrating the passes. When selecting the minimum, we compare the length of the vectors. Euclidean distance needs a square root, but squaring is monotonous, so the square root can be left out during comparation.

∞	∞	∞	∞
∞	0;0	0;0	∞
∞	∞	0;0	∞
∞	∞	∞	∞

1.

∞	∞	∞	∞
∞	0;0	0;0	-1;0
∞	0;-1	0;0	-1;0
∞	0;-2	0;-1	-1;-1

2.

1;1	0;1	0;1	-1;1
1;0	0;0	0;0	-1;0
1;-1	0;-1	0;0	-1;0
2;-1	1;-1	0;-1	-1;-1

3.

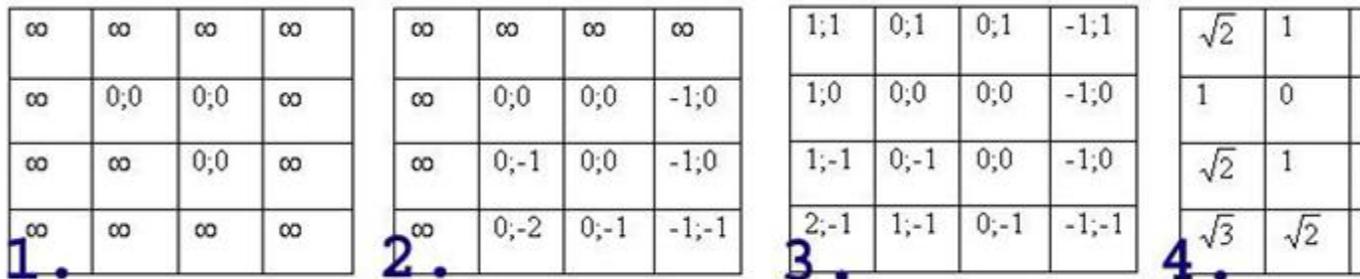| $\sqrt{2}$	1
1	0
$\sqrt{2}$	1
$\sqrt{3}$	$\sqrt{2}$

4.

Figure 6: The steps of the Danielsson's algorithm illustrated in two dimensions

This algorithm can be easily extended to three dimensions. The two pass system is the same, but this time we iterate between the two opposite corners of a box. We consider the neighbours along the z axis as well.

Finally we get the distance map by calculating the length of each distance vector. The accuracy of this algorithm could be improved by introducing further passes (for example iterating between the other two corners, too), but in practice this method proved to be enough.

---

4. Shading and texturing

After finding the intersection with the ray, the final step of rendering is shading. For shading we must know the direction of the surface normal, the material parameters of the surface point, and the incoming illumination.

4.1. Illumination

We get the surface normal of an implicit model by taking the gradient at the specified point:

$$\vec{n}(\vec{p}) = grad_f(\vec{p}) = \left[\frac{df(\vec{p})}{dx}, \frac{df(\vec{p})}{dy}, \frac{df(\vec{p})}{dz} \right]$$

When computing incoming illumination we can render self-shadows easily with a small improvement. From the surface point we start rays toward each light source using the same algorithm as applied from the camera. If the ray can step out from the bounding box without hitting the surface, then the point is visible from that light source. If the ray gets into the body, then the point is in shadow.

4.2. Texturing

A texture is commonly parameterized over the two dimensional unit square $(u, v) \in [0, 1]^2$. Its points are mapped to the surface by the texture projection. In the case of implicit surfaces the traditional texturing methods usually cannot be used. There are no fix points on the surface like vertices, so we cannot assign (u, v) coordinates to the surface points. Other kind of texturing must be used.

4.2.1   3D textures

These kinds of textures respect the principles of implicit modeling. A 3D texture is also a function defined over the unit cube in (u, v, w) coordinates. Procedural textures describe the texture space with mathematical functions, so that their level of detail can be increased infinitely. However, the possible texture patterns are bounded to the mathematically definable ones. Figure 7 shows examples for frequently used procedural textures. A texture can be a discrete 3D array as well, similar to two dimensional bitmaps. The main

disadvantage is the needed storage space. A texturing artist cannot conveniently handle 3D textures, so their usage is rare.
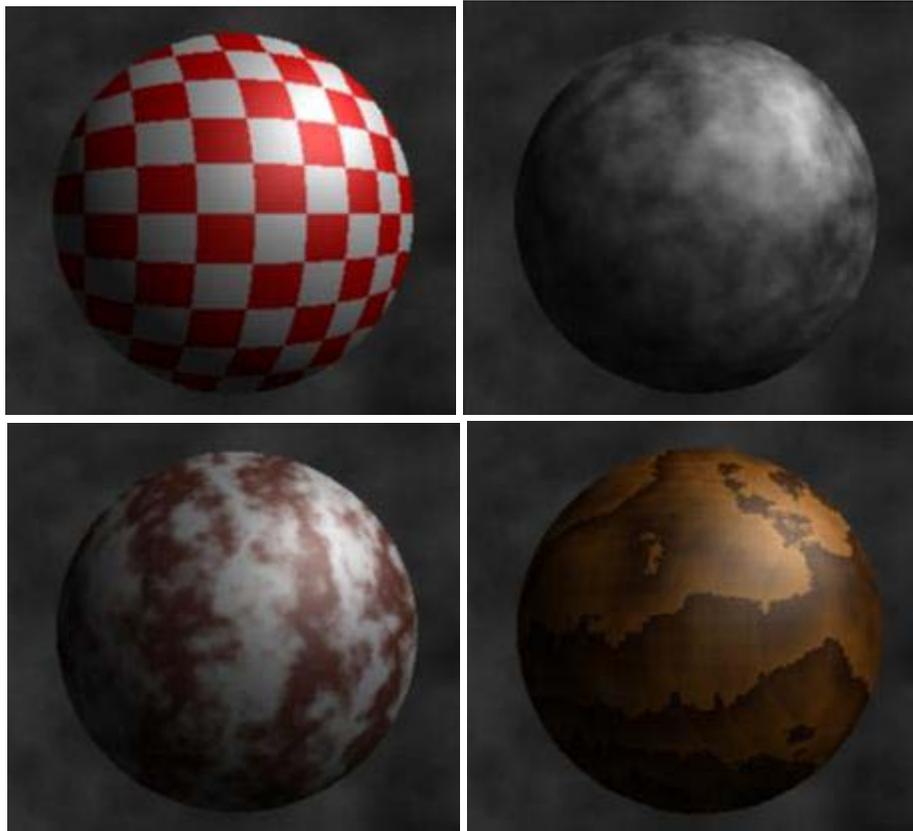

Figure 7: Examples for procedural 3D textures: checkerboard, turbulence, marble, and wood.

4.2.2 Texturing surfaces

We can use a helper surface to project 2D textures onto the implicit surface. The helper surface can be textured in the usual way. The only question is how to set up the projection which assigns every $f^{-1}(0)$ surface point to the points of the helper surface. Many techniques have been developed addressing this problem with different levels of accuracy and complexity.

The simplest texturing method is *planar projection.* Essentially we assign a texture to one side of the bounding box, and then determine a surface point (*u, v*) according to its position in modeling space. This leads to poor texturing results because of the distortion of the texture along the projection axis (Figure 8a).

Tri-planar projection (Figure 8b) uses all the three main texturing planes (XY, YZ, XZ), to avoid distortion. It is based on the fact that texturing is accurate when the surface element is parallel to the texturing plane, and the distortion is maximal if it is perpendicular.

According to the surface normal we can assign each surface element a dominant plane, and we texture this surface according to this texturing plane. To get smooth interpolation of the three textures, we use all the three texturing planes, but weight them with the components of the surface normal. For procedural textures, nice results can be achieved this way.

In [6] Wyvill et al. show an interesting, particle based approach for accurate surface texturing. This method shoots particles from the textured surface points, which hit the helper surface following the gradient field of the implicit function. Then the texture coordinates of the intersected point are assigned to the original surface point where the particle was coming from. This algorithm, as Figure 8c-d demonstrates, results in surprisingly nice and correct projection even at curved regions. However, its real time application is limited because the algorithm must evaluate the gradient for every visible surface point many times. In the implementation section we present a possible approximating solution by preprocessing the texture coordinates.
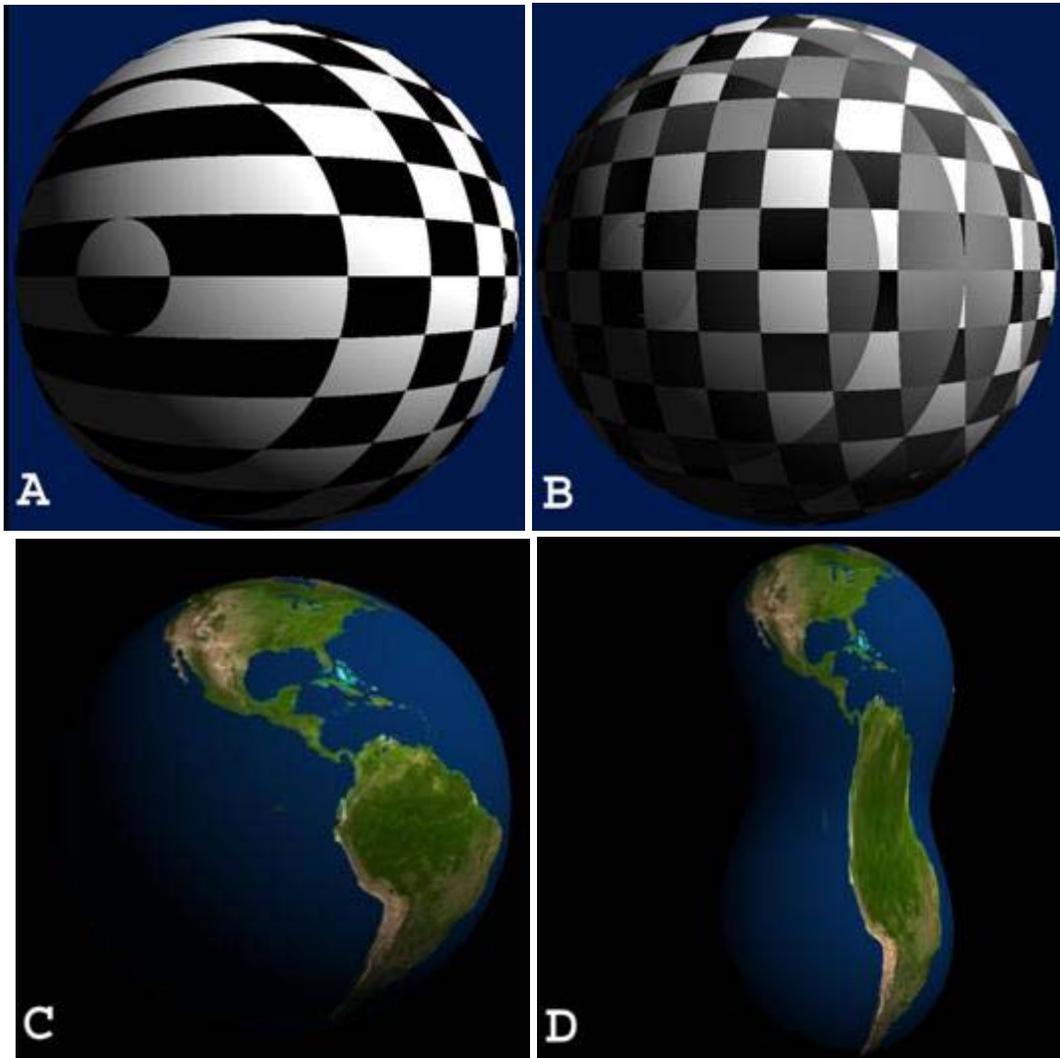
5. Implementation and results

The demonstrative application was implemented in C++, using DirectX 9. Our goal was to make the pixel shader perform ray tracing, and to leave only the preprocessing step to the CPU.

The frame rates of the test renders were measured in the following environment:

. Intel Core2 Duo E4500 (2,2 GHz) CPU

. Nvidia 8600GTS, 256MB GPU

. 2GB DDR2 RAM

All images were rendered at full screen (1280×1024) but in the illustrations they were cropped to fit into a square.

5.1. Preprocessing

In the preprocessing step we generate a three dimensional distance map of the implicit function using the Danielsson's algorithm that is described in section 3. Naturally, this step can be omitted if we render the surface directly, using distance bound functions. We store the resulting distance map on the GPU as a 3D scalar texture. The ray tracing step will determine dt with a texture read. Here we emphasize the importance of texture filtering. If texture filtering is enabled then the hardware uses tri-linear interpolation between the neighboring texels. Table 1 summarizes how the rendering speed scales with the resolution of this 3D texture, and the number of ray casting iterations.

Grid res. Iterations	5	10	15	20
64×64×64	146 fps	91 fps	62 fps	42 fps
128×128×128	147 fps	**83 fps**	**42 fps**	**27 fps**
256×256×256	40 fps	**6. 5 fps**	**2. 5 fps**	-

Table 1: *Performance tests of the algorithm. Only the bold cells gave visually adequate results at full screen. The model we used was the blob object of Figure 8d (without texturing). As the table shows, large distance fields should be avoided because of the cubic*

growth of the texture. At the resolution of 256×256×256 the size of the texture prevented efficient texture caching on the hardware; this explains the FPS breakdown at the last row.

5.2. Vertex shader

The sphere tracing algorithm runs entirely on the pixel shader. To invoke the pixel shaders, we render the bounding box of the surface as a standard DirectX triangle list. The vertex shader has the only task to transform the bounding box vertices into the projective space. The pixel shader must know the original world coordinates of the vertices, so we pass them through the TEXCOORD0 register.

Figure 9: *Smooth metamorphosis of two objects with different geometry and texture (no preprocessing here!). This kind of topology change is hard to carry out in the conventional vertex based modeling methods. The implicit modeling solves it by making the weighted sum of the interpolated modeling functions. 24 FPS, 1280×1024 (70 FPS without textures).*

5.3. Pixel shader

First we enter into the bounding box with the ray. The coordinates of the ray hit are interpolated by the hardware in world space. Here we apply the inverse modeling transformation to the hit and the eye position to get the ray in modeling space. While the ray is in the bounding volume we can determine the distance from the surface by reading from the preprocessed 3D texture. Finally we store in the *inside* boolean if the ray has left the bounding volume.

We locate the intersection using sphere tracing: we find a pos1 point inside the surface and a pos2 point outside the surface. To simplify texture reading, the bounding volume is scaled to the $[-0,5; 0,5]^3$ cube. Sphere tracing roughly locates the intersection between pos1 and pos2, so we can use a numeric root finding algorithm to refine the result. In this case our algorithm implements the regula falsi method.

```
void ps_SphereTracing(float4 xpos:TEXCOORD0, float4 texcoord:TEXCOORD1,
out float4 color: COLOR0, out float depth: DEPTH0)
{
        //get ray position and direction in model space
 [.]
        for(int i=0; i < SPHERE_STEPS, i++)
        {
                t=tex3D(SphereTex, float4(pos1.x + 0.5f, pos1.y - 0.5f, pos1.z+0.5f,
0.0f)).x;
                pos1 += look * t;
        }

        if( pos1.x > 0.5f || pos1.x < -0.5f || pos1.y > 0.5f || pos1.y <   -0.5f || pos1.z >
0.5f || pos1.z < -0.5f )
                inside = false;
        //"overshoot" the surface to ensure pos1 is inside the body
        f1=F(pos1);
        if(f1>=0)
                pos1 = pos1+ look*delta;
        inside=inside && (f1<0);

        if(inside)
        {
                //pos1 is inside, pos2 is outside. perform regula falsi
                pos2 = pos1 - look * 4 * delta;
                f2 = F(pos2);
                spos = (pos1 * f2 - pos2 * f1) / (f2 - f1);
                //step2
                for(int j = 0;  j < FALSI_STEPS;  j++)
                {
                        t=F(spos);
```

```
                    if(t>0) {
                            f2=t;
                            pos2=spos;
                    }
                    else{
                            f1=t;
                            pos1=spos;
                    }
                    spos=(pos1*f2-pos2*f1)/(f2-f1);
            }
[.] //shading
            //modify the depth value of the fragment
            depth = spos.z / spos.w;
            return;
        }
        else{
            color = backcolor;
            depth =1.0f; //background. This fragment should be discarded
            return;
        }
}
```

Listing 2: *The intersection finding in the pixel shader*

5.4. Notes on efficiency and execution coherence

In connection with our implementation, we should discuss some minor drawbacks and questions about the sphere tracing algorithm. The first question is execution coherence. The classical sphere tracing, as shown in *listing 1* in an earlier chapter, is not really suitable for SIMD platfroms like the GPU, because the different rays take different number of steps along their path, depending on the distance function. For example rays roughly perpendicular to the surface would converge quite fast, while others parallel to the contours require much more steps. This situation is illustrated on Figure 10. That would mean an inefficient, divergent execution on the GPU, where the SIMD processors would have to iterate along each ray until the "slowest ones" reach the surface. Therefore we made two modifications in the algorithm.
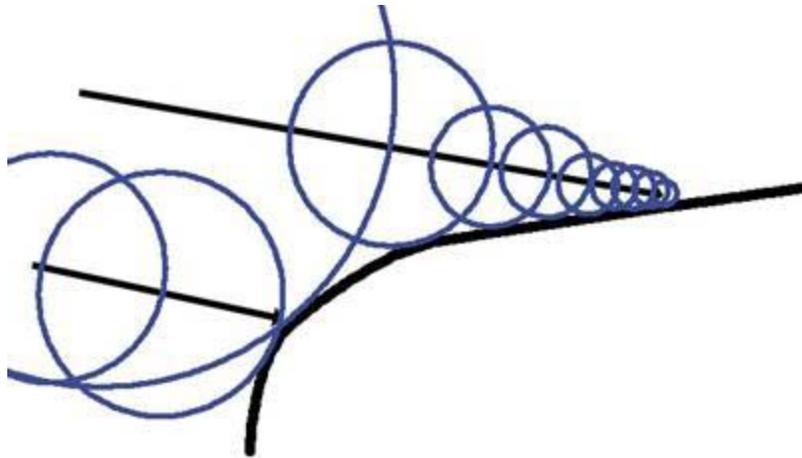
Figure 10*: Sphere tracing suffers from performance drawbacks on the contours.*

First of all, if we examine our pixel shader in *listing 2*, we can see that each ray take the same number of iterations (SPHERE_STEPS) during ray casting, instead of terminatig the rays after the distance is less than a bound. The disadvantage of this solution is that we need to adaptively choose a global step number for all rays, but this serves coherent execution well. The regula falsi root finding is also a pre-defined iteration, and because the constant FALSI_STEPS is generally small, it can be unrolled by the compiler. With these modifications we could improve execution coherence as far as possible.

On the other hand, we altered our pixel shader to get around the weakness of sphere tracing on the countours. We examine the distance from the surface in every step, and if we find it less than a proper bound (epsilon), we degrade into simple linear search, because in the vicinity of the surface it takes larger steps than sphere tracing. This sphere tracing - linear search hybrid proved to be a quite efficient combination.

Another interesting performace question of a GPU implementation is cache coherence. Sampling along the rays in the distance map can easily turn into a bottleneck if we cannot utilize the texture caching abilities of the hardware. Unfortunately, when the surface has a complex form, and the per-ray samples get distant from each other because of varying step size, caching will be inefficient. Linear search has the advantage of coherent sampling, but as we mentioned before it also needs an order of magnitude higher step number for fine details.

For this paper we tested our implementation only on simpler surface forms, therefore we cannot say that sphere tracing is compulsorily faster than linear search in every case, but all of our tests have shown that even the cache friendly sampling, the inablity to adapt to the surface made linear search inferior. If the distance map is small, like $32 \times 32 \times 32$ then linear search outperforms sphere tracing, but with the increasing of the resolution, the latter will be better.

Of course if we use the Lipschitz property and analyically estimate the distance from the surface, this issue means no problem. Therefore we should use analytic methods when

possible and use preprocessing only when the surface is too complex to evaluate, or non-Lipschitz.

5.5. Texturing

5.5.1. Procedural textures

In our application, we applied 3D procedural and projective textures to the models. Procedural texturing algorithms use a simple modeling function, which is perturbed with a pseudo-random spatial noise. The right choice of this noise function is important to achieve natural impression. Completely random noise does not provide realistic look. In natural textures smaller frequency components are stronger. For example, a stone has a basic tone of color, and the high frequency components give the variety of the surface for what we do not see it solid.

We have used the *Perlin noise* [7] to modulate our procedural textures. This method can be ported to the GPU easily, because the random values are fetched iteratively from a 2D noise texture. In every iteration step we increase the noise frequency to the double, and half the amplitude. The sum of the results is an infinitely refining noise function; in practice we should stop when the "wavelength" equals to the pixel size.

5.5.2. Projective textures

We have implemented an efficient approximation for the particle based texturing method. Because tracing particles from every surface point to the helper surface is very costly, we do not want to do that for every frame on the GPU. Instead, we have moved the texture projection to the preprocessing step.

Our algorithm exploits the fact that after distance transform, we already have a three-dimensional scalar texture in the GPU memory. Choosing a float4 texture instead of a float texture, we can store a 3D vector besides the distance value at each voxel. In this vector we can store texture coordinates for instance.

In the preprocessing step we trace particles along the gradient field accurately from every sample points, and store the (u, v) coordinates of the hits in the 3D texture. When rendering in real time, the texture coordinates of any point of the surface can be read from this texture by fast hardware interpolation. Of course, this algorithm suits only for rigid models with fixed geometry, and gives only an approximate result depending on the texture resolution. The problem is that real texture projection is not linear, but the interpolation between texture samples is. However, as the tests showed, this method can be useful in many cases.

Figure 11: *The integration of our shaders into the standard rendering pipeline was demonstrated on this simple teapot scene. By maintaining the depth buffer in the pixel shader, we can achieve transparency, accurate shadow map and environment map effects.*

6. Conclusion and future work

According to our experiences, sphere tracing proved to be efficient and is able to reach real time frame rates on the GPU, especially when we can afford preprocessing. This is the case when rendering implicit surfaces with fixed geometry.

Rendering deforming objects with this technique (Figure 9), however generally excludes preprocessing. The efficiency of the Lipschitz function based approach depends on how accurately we can estimate the distance from the surface. When the model consists of several primitive functions, the calculation of the distance function can be expensive. The speed of the algorithm is determined by the complexity of the contained functions.

We successfully integrated this ray-traced visualization algorithm into the standard rendering pipeline. By rendering the bounding box, then modifying the contents of the depth buffer, the implicit objects can be used together with the polygonal ones (Figure 11). The image synthesis needs an extra rendering pass for each surface, but this can be improved in the future.

The preprocessing method can be refined as well. If we mention sampling it is worth thinking how large storage space the samples consume if we take them according to a 3D grid. In the future it would be useful to use more efficient, hierarchical structures to perform adaptive sampling.

We also note that sphere tracing is actually too cautious. We always step along the ray small enough not to hit the surface, though in reality it is not a problem if we cross the surface, this is even necessary for the following numerical root finding. So the real solution would be finding those largest spheres which radius does not intersect the surface more than once. What we must ensure is that we can always find the first intersection. This so-called *relaxed sphere tracing* would mean a significant improvement.

Acknowledgements

References

[1] Jules Bloomenthal, *Implicit Surfaces*. Article on implicit surfaces in the Encyclopedia of Computer Science and Technology (2000)

[2] László Szirmay-Kalos, Tamás Umenhoffer: *Displacement Mapping on the GPU - State of the Art*. Computer Graphics Forum (2008) 27:(6) pp. 1567-1592.

[3] John C. Hart: *Sphere Tracing: a geometric method for the antialiased ray tracing of implicit surfaces*. The Visual Computer, vol. 12., p. 527-545 (1996)

[4] William Donnelly*: Per-Pixel Displacement Mapping with Distance Functions*. Nvidia GPU Gems, p. 123-136 (2005)

[5] Donald G. Bailey: *An Efficient Euclidean Distance Transform*. Combinatorial Image Analysis, p. 394-408 (2004)

[6] Brian Wyvill, Mark Tigges: *Texture Mapping the Blobtree*. University of Calgary, (1998)

[7] Alan Watt, Mark Watt: *Advanced Animation and Rendering Techniques*, Chapter 7: procedural texture mapping and modelling. (1992, ACM Press)

[8] Yoshihiro Kanamori, Zoltan Szego and Tomoyuki Nishita: *GPU-based Fast Ray Casting for a Large Number of Metaballs*. Eurographics, Vol. 27 (2008), No. 2.

[9] Martin Reimers, Johan Seland: *Ray Casting Algebraic Surfaces using the Frustum Form*. Eurographics, Vol. 27 (2008), No. 2.