Technical Report TR-NCCA-2009-04

FAST RELIABLE RAY-TRACING OF PROCEDURALLY DEFINED IMPLICIT SURFACES USING REVISED AFFINE ARITHMETIC

Oleg Fryazinov, Alexander Pasko and Peter Comninos

The National Centre for Computer Animation
Bournemouth Media School
Bournemouth University
Talbot Campus,
Poole, Dorset BH12 5BB
United Kingdom
2009

Technical Report TR-NCCA-2009-04

ISBN 1-85899-123-4

Title Fast Reliable Ray-tracing of Procedurally Defined Implicit Surfaces Using Revised Affine Arithmetic

Author(s) Oleg Fryazinov, Alexander Pasko and Peter Comninos

Keywords Ray Tracing, Implicit Surfaces, Function Representation, Revised Affine Arithmetic

Abstract

Fast and reliable rendering of implicit surfaces is an important area in the field of implicit modelling. Direct rendering, namely ray-tracing, is shown to be a suitable technique for obtaining good-quality visualisations of implicit surfaces. We present a technique for reliable ray-tracing of arbitrary procedurally defined implicit surfaces by using a modification of Affine Arithmetic called Revised Affine Arithmetic. A wide range of procedurally defined implicit objects can be rendered using this technique including polynomial surfaces, constructive solids, pseudo-random objects, procedurally defined microstructures, and others. We compare our technique with other reliable techniques based on Interval and Affine Arithmetic to show that our technique provides the fastest, while still reliable, ray-surface intersections and ray-tracing. We also suggest possible modifications for the GPU implementation of this technique for real-time rendering of relatively simple implicit models and for near real-time for complex implicit models.

Report date 5 October 2009

Web site to download from http://eprints.bournemouth.ac.uk/

The authors' e-mail addresses

{ofryazinov,apasko,peterc}@bournemouth.ac.uk

Supplementary notes

The National Centre for Computer Animation
Bournemouth Media School
Bournemouth University
Talbot Campus,
Poole, Dorset BH12 5BB
United Kingdom

1. Introduction

In recent years, implicit surfaces (isosurfaces of trivariate real functions) have proved to be a powerful and simple solution to some complex problems in the area of modelling and animation. For example, implicit surfaces provide solutions for surface reconstruction from scattered points and for fluid simulation. Several operations, such as sweeping, metamorphosis and offsetting can be implemented quite easily with implicit models unlike traditional boundary-representation models. However, modelling with the whole range of implicit surfaces is still a complicated task because interactive rendering of arbitrary implicit surfaces is still an open problem. Because of that, most of implicit modelling tools to date are limited to a narrow range of implicit surfaces or do not have an interactive mode. Currently, there are two ways to render an implicit model: generation of a polygonal mesh and direct rendering using ray-tracing. Polygonization is a well-known and widely used technique for rendering of implicit surfaces. However, in many cases, when the model has sharp or thin features, large numbers of small-sized disjoint elements or internal microstructures, the generation of an appropriate polygonal mesh takes a long time and requires a large amount of memory. In many cases additional techniques to refine the polygonal mesh are needed. Moreover, in the case of an animated implicit surface, the polygonal mesh has to be created for each frame and it is thus inappropriate for interactive applications. A more promising technique is that of interactive direct rendering of implicit surfaces using ray-casting and ray-tracing. Traditionally the main disadvantage of direct rendering of implicit surfaces was their slow speed due to the large number of ray-surface intersection calculations. With recent developments of hardware this problem becomes less critical, but not insignificant all together.

Many techniques of ray-casting and ray-tracing implicit surfaces have been developed. However, the majority of these techniques have disadvantages, because they either work with a small range of implicit surfaces (for instance those defined only by polynomials), or not reliable. For example, classic approximate techniques, such as ray marching, are fast, but can easily miss sharp features and small components. Classical numerical techniques, such as the Newton search require different signs of the defining function at the ends of the ray interval, which is inappropriate for arbitrary rays. Sphere-tracing based techniques require a distance property of the defining function, which can not be provided for general models. Techniques based on interval analysis and other reliable numerical computations have also been applied to the ray-tracing of implicit surfaces. However, classic Interval Arithmetic is slow because of the interval overestimation.

The problem considered in this paper is that of finding a technique for ray-tracing of general implicit surfaces, that has the following properties:

- Its ray-surface intersection procedure should be reliable, i.e. no roots should be missed.
- A wide range of implicit models should be supported – meaning that the algorithm should be able to work with procedurally defined models as well as with algebraic ones.
- The procedure should be fast and suitable for a GPU implementation of interactive rendering.

In this paper we propose to use Revised Affine Arithmetic as a fast and reliable technique for calculating the range of a function for a given interval and hence the core for the ray-surface intersection procedure. The main contributions in this paper are: 1) a new algorithm for reliable ray-tracing of general procedurally defined implicit surfaces using Revised Affine Arithmetic in contrast to the formerly reported applications of Reduced Affine Arithmetic exclusively to algebraic surfaces defined by polynomials; 2) a technique for optimising the proposed ray-tracing procedure by using argument pruning and cell culling; 3) a possible implementation of this algorithm on both the CPU and GPU.

2. Related work

Ray-tracing of implicit surfaces is a well-researched area. The classic techniques for ray-tracing of implicit surfaces were presented in [Har93]. Most of the described techniques are approximate and can miss small surface features, but on the other hand they are suitable for all types of implicit surfaces. Later, several techniques were presented for particular types of implicit surfaces that provide not only speed but also reliability. Thus, in [Har94] a distance property is needed for the ray-tracing procedure, in [She99] blobs, metaballs and convolution surfaces are the type of implicit surface that can be rendered fast.

Another way to increase speed of ray-tracing is by reducing the number of processed rays intersecting the implicit surface. In [Has03], image-space subdivision is used, while [GM07a] uses progressive refinement.

Using specialised hardware also increases the speed of ray-tracing. Many papers use the GPU for real-time rendering, however most of these papers have focused on polygonal meshes [PBMH02], parametric surfaces [LB06] and volumetric data [KW03]. GPU-based ray-tracing of implicit surfaces was introduced only for particular types of the objects, such as radial-basis functions [CD05], low-degree implicit surfaces [KOKK06] and discrete isosurfaces [HSS*05]. Ray-tracing of general implicit surfaces on the GPU was performed in [FP08] [SNar] by using approximate methods.

Reliable computational techniques based of Interval Arithmetic have been known for a long time. However, most of the literature relates to fields such as global optimisation rather than computer graphics. The works of [Mit91] and [Sny92] discussed applications of Interval Arithmetic

for computer graphics purposes, and Affine Arithmetic was used for ray-tracing of implicit surfaces in [dCFG99] . A good comparison of different interval techniques can be found in [MSVW01], however the list of the implicit models used in this paper is limited to those given in the polynomial form. Interval Arithmetic and Reduced Affine Arithmetic are applied for fast rendering of implicit surfaces by using the GPU in [KHK*09]. A more detailed comparison of these techniques with the one proposed here can be found in the "Results" section below. In this paper, we use Revised Affine Arithmetic [VSHFar], which was introduced recently for the purposes of constraint propagation and has not yet been used in computer graphics.

3. Background

3.1. Procedurally defined implicit surfaces

A zero level set or an isosurface of a trivariate real function f of a point with coordinates (x,y,z) is traditionally called an implicit surface and is defined as $f(x,y,z)=0$. It can also be considered as the boundary of a solid (three-dimensional manifold) defined by the inequality $f(x,y,z)\geq 0$. There are many different ways to specify the function $f(x,y,z)$. The simplest form is that of an algebraic implicit surface defined by a polynomial function. Most of the extant work on reliable ray-tracing concentrates solely on algebraic surfaces. More complex forms involve exponential, square root, trigonometric and other non-linear functions. We deal with the most general form of procedurally defined implicit surfaces, where the function f is evaluated by some procedure involving all kinds of non-linear functions as well as loops and conditional operations. This allows us to cover skeleton-based implicit surfaces [BW97], Constructive Solid Geometry (CSG) objects defined by nested R-functions [Sha07], solid noise [PH89] [Gar84], fractals and other complex objects.

3.2. Affine Arithmetic

Affine Arithmetic is a technique for performing computations on uncertain numerical values. The main idea of Affine Arithmetic is the calculation of an uncertain value (function) based on other uncertain values (arguments). Initially this model was introduced for self-validated numerical computations as an alternative to Interval Analysis – in some literature Affine Arithmetic is still considered as the modification of general Interval Arithmetic – and currently it is used in many different areas of computer science [dS97]. By keeping track of the errors for each computed quantity, Affine Arithmetic provides much tighter bounds for computed quantities compared to classical Interval Arithmetic. Uncertain values in Affine Arithmetic are represented by affine forms, i.e. polynomials of the form:

$$\hat{x} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \ldots + x_n\varepsilon_n$$

where x_n are known real coefficients and ε_n are noise symbols, i.e. symbolic variables whose values are assumed to lie in the interval $\varepsilon_n \in [-1,1]$.

In Affine Arithmetic, formula evaluation is performed by replacing operations on real quantities by their affine forms. Similar to Interval Arithmetic, the inclusion property is applied in Affine Arithmetic, i.e. for any operation $\otimes$,
$A \otimes B \subset \{a \otimes b, a \in A, b \in B\}$,
where a and b are real values and A and B are uncertain values in affine form.

All operations on affine forms can be divided into affine (exact) and non-affine (approximate) operations. An affine operation is a function that can be represented by the linear combination of the noise symbols of its arguments. For example, a multiplication by a constant is an affine operation:

$$\alpha\hat{x} = \alpha x_0 + \alpha x_1\varepsilon_1 + \alpha x_2\varepsilon_2 + \ldots + \alpha x_n\varepsilon_n$$

Non-affine operations can not be performed over the linear combination of the noise symbols. In this case an approximate affine function is used and a new noise symbol is added to the affine form to represent the difference between the non-affine function and its approximation. For example, the multiplication of two affine forms is a non-affine operation and introduces a new noise symbol, ε_{n+1}:

$$\hat{x} * \hat{y} = x_0y_0 + (x_0y_1 + x_1y_0)\varepsilon_1 + \ldots + (x_0y_n + x_ny_0)\varepsilon_n + (\sum_{i=1}^{n}|x_i|\sum_{i=1}^{n}|y_i|)\varepsilon_{n+1}$$

In the general case any operation can be represented in an affine form:

$$\hat{x} \otimes \hat{y} = \alpha\hat{x} + \beta\hat{y} + \zeta \pm \delta$$

where the value of the new noise symbol is represented by δ. In [dS97], different approximation techniques are discussed for the affine form of several functions.

3.3. Reduced Affine Arithmetic

Pure Affine Arithmetic is computationally- and memory expensive and can not be used in algorithms where the reduction of computational complexity is equally important as the quality of the computational result. In [Mes02], several reduced affine forms were introduced to reduce the number of computations in Affine Arithmetic by accumulating errors. The Affine Form 1 (AF1) is the simplest one and represents the uncertain quantity as:

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i\varepsilon_i + x_{n+1}\varepsilon_{n+1}$$

The noise symbols $\varepsilon_1,\ldots,\varepsilon_i$ represent the errors of the initial arguments. The last noise symbol represents all the errors after the non-affine operations. Reduced Affine Arithmetic [GM07b] was introduced for the AF1 and it was shown that the best results from the computational point

of view can be obtained by using only two noise symbols, i.e n=1. Thus, in Reduced Affine Arithmetic the first noise symbol represents the error on the argument interval and the second noise symbol represents the accumulation noise symbol after all the non-affine operations have taken place. Despite the fact that the general affine operation was not clearly discussed in that paper, it can be defined similarly to the ordinary affine operation with condensation as follows:

$$\hat{x} \otimes \hat{y} = (\alpha x_0 + \beta y_0 + \zeta) + \sum_{i=1}^{n} (\alpha x_i + \beta y_i) + (|\delta + \alpha x_{n+1} + \beta y_{n+1}|)\varepsilon_{n+1}$$

The length of the reduced affine form remains the same after the affine computations, however the accumulation of all the errors in one symbol leads to a wider error and thus to a widening of the bounds of the computed value.

3.4. Revised Affine Arithmetic

Revised Affine Arithmetic was introduced by Vu et al. [VSHFar] for the purposes of numerical constraint propagation and reduces the problem of growth of the error in the last (accumulating) noise symbol. The revised affine form is similar to the reduced affine form:

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i \varepsilon_i + e_x[-1,1], e_x \geq 0$$

Despite their similar forms and the fact that they have the same geometric sense, the reduced and revised affine forms have different mathematical backgrounds. While in Reduced Affine Arithmetic we accumulate errors after non-affine operations in the last noise symbol, in Revised Affine Arithmetic we accumulate the error in the symmetrical interval.

From the formal point of view, the difference between the Reduced Affine Arithmetic and the Revised Affine Arithmetic is in the definition of the general non-affine operation and the tight form for multiplication. The binary affine operation is defined as:

$$f(\hat{x}, \hat{y}) = (\alpha x_0 + \beta y_0 + \zeta) + \sum_{i=1}^{n} (\alpha x_i + \beta y_i) + (\delta + |\alpha| e_x + |\beta| e_y)[-1,1]$$

where α, β and ζ can be taken from the affine approximation of the function f. The unary affine operation is defined in the same way:

$$f(\hat{x}) = (\alpha x_0 + \zeta) + \sum_{i=1}^{n} \alpha x_i + (\delta + |\alpha| e_x)[-1,1]$$

The formula for the multiplication is defined as:

$$\hat{x} * \hat{y} = (x_0 y_0 + \tfrac{1}{2} \sum_{i=1}^{n} x_i y_i) + \sum_{i=1}^{n} (x_0 y_i + x_i y_0)\varepsilon_i + e_{xy}[-1,1]$$

$$e_{xy} = e_x e_y + e_y(|x_0| + u) + e_x(|y_0| + v) + uv - \tfrac{1}{2} \sum_{i=1}^{n} |x_i y_i|$$

where $u = \sum_{i=1}^{n} |x_i|$, $v = \sum_{i=1}^{n} |y_i|$.

As for standard Affine Arithmetic, Revised Affine Arithmetic has an inclusion property. In our technique we use a shortest possible revised affine form, i.e. $i = 1$.

4. Ray-tracing with Revised AA

The main part of any ray-tracing procedure for implicit surfaces is the calculation of the zero roots of the defining function in the ray-surface intersection procedure. In this section we show how Revised Affine Arithmetic can be used for the intersection point calculation and we present several techniques for speeding up this calculation.

4.1. Ray-surface intersection

Our algorithm is based on a ray-surface intersection technique for implicit surfaces that uses interval techniques, which originally appeared in [Mit91]. We present the ray-surface intersection procedure in Algorithm 1.

Algorithm 1 Ray-surface intersection

Procedure: bool intersect(t_{min}, t_{max})
Calculate the affine form F for the function on the interval $[t_{min}, t_{max}]$
Get the range of the function from the affine form
if the range of the function does not include a 0 value **then**
  return FALSE (no roots in this interval);
end if
Calculate the argument estimation from the affine form: t'_{min}, t'_{max}
Find the pruned argument range:
$t_{min} = max(t_{min}, t'_{min})$;
$t_{max} = min(t_{max}, t'_{max})$;
if the length of the argument interval is less than some predefined accuracy **then**
  Store the midpoint of the interval as the root;
  return TRUE;
end if
Calculate the midpoint of the argument range:
$t_{mid} = (t_{min} + t_{max})/2$;
Repeat the procedure for the two subintervals:
bool b1 = intersect(t_{min}, t_{mid});
if b1 is TRUE and only the first root is needed **then**
  return TRUE;
end if
bool b2 = intersect(t_{mid}, t_{max});
if b2 is TRUE **then**
  return TRUE;
end if
return FALSE;

The basic idea of the algorithm is quite simple: we calculate the range of the function for the given argument interval

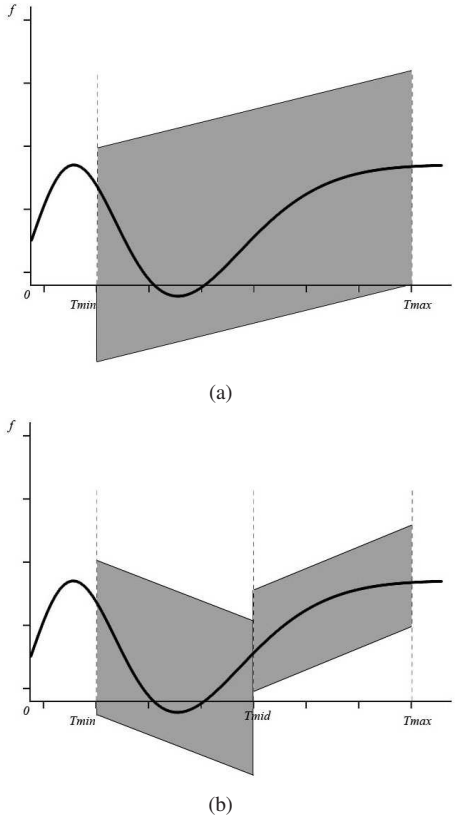


Figure 1: *a) The revised affine form for the function on the interval* $[t_{min}, t_{max}]$ *b) The revised affine form for the function on the two subintervals after the dichotomy.*

using Revised Affine Arithmetic, we reject the interval if the range does not include the zero value, otherwise we subdivide the interval into two intervals by using dichotomy and we repeat the procedure for both subintervals. An example of the affine form for the function before and after the dichotomy is shown in the figure 1. Note that in the case when only the first root is needed (for example, for primary rays), we can exit from the procedure earlier if we have found a root in the first subinterval after the recursive procedure. Below we explain several details of the algorithm.

4.1.1. Interval range for the function in Revised Affine Arithmetic

The calculation of the interval of the function over the interval of the arguments is performed in three steps. First, we obtain the revised affine form for the argument interval:

$$\hat{t} = \frac{t_{min} + t_{max}}{2} + \frac{t_{max} - t_{min}}{2}\varepsilon_1$$

We also obtain the affine forms for the coordinate variables x, y and z, as the defining function is usually defined over

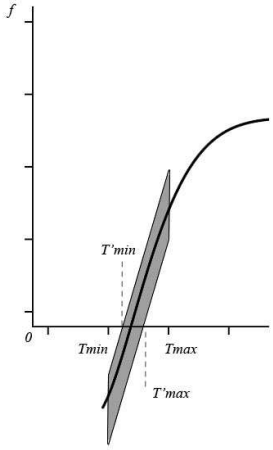


Figure 2: *Pruning of the interval* $[t_{min}, t_{max}]$ *to the interval* $[t'_{min}, t'_{max}]$ *after the evaluation of the revised affine form for the function.*

these variables:

$$\hat{x} = x_0 + \hat{t} * d_x, \hat{y} = y_0 + \hat{t} * d_y, \hat{z} = z_0 + \hat{t} * d_z$$

Here x_0, y_0, z_0 are the coordinates for the ray origin and d_x, d_y, d_z are the coordinates for the ray direction. For each ray these coordinates are constant.

Secondly, we calculate the range for the function by evaluating its revised affine form. The revised affine form is obtained from the procedural definition of the function by replacing all the operations on real numbers by operations on the revised affine forms.

Finally, we obtain the range of the function from the affine form $\hat{f} = f_0 + f_1\varepsilon_1 \pm e_f$:

$$f_{min} = f_0 - |f_1| - e_f$$

$$f_{max} = f_0 + |f_1| + e_f$$

4.1.2. Argument pruning

One of the useful properties of the reduced and hence the revised affine forms is that of argument pruning (a term taken from literature of interval slope methods), that means that we can not only calculate the function range for the interval, but also can narrow the argument range in case the root exists in this interval. In [GM07b], the argument pruning formulation (the term interval optimisation is used in the paper) was suggested for Reduced Affine Arithmetic. As the geometric meaning of the revised affine form is similar to that of the reduced affine form, an analogous formulation can be used as follows. Given the revised affine form for the function $\hat{f} = f_0 + f_1\varepsilon_1 \pm e_f$ for the interval $\hat{t} = t_0 + t_1\varepsilon_1$, providing

that $t_1 \neq 0$, we have:

$$\varepsilon_1 = \frac{\hat{t} - t_0}{t_1}$$

$$\hat{f} = f_0 + f_1 \frac{\hat{t} - t_0}{t_1} \pm e_f = (f_0 - f_1 \frac{t_0}{t_1}) + \hat{t} \frac{f_1}{t_1} \pm e_f$$

The geometric meaning of this form in 2D space {t, f} is the parallelogram bounded by the lines $f = (f_0 - \frac{t_0}{t_1}) + \hat{t} \frac{f_1}{t_1} - e_f$, $f = (f_0 - \frac{t_0}{t_1}) + \hat{t} \frac{f_1}{t_1} + e_f$ and the coordinate lines $t = t_{min}$ and $t = t_{max}$. This parallelogram intersects the axis t at two points provided that $f_1 \neq 0$ and $e_f \neq 0$:

$$t' = t_0 - \frac{t_1 f_0}{f_1} \pm e_f \frac{t_1}{f_1}$$

If these two points lie inside the interval $[t_{min}, t_{max}]$, the interval can be pruned. (see Fig. 2).

4.2. Cell Culling

If we subdivide the scene to be rendered into rectangular cells, the ray-tracing procedure can be accelerated by rejecting the cells where zero roots of the function do not exist. This can be done because of the interval nature of Revised Affine Arithmetic. Given a cell intersected by a bundle of initial rays on the 3D interval $[(x_{min}, y_{min}, z_{min}), (x_{max}, y_{max}, z_{max})]$ the cell can be entirely rejected if the function range for the affine form on the cell interval does not include a zero value. The calculation of the function range is slightly different in this case as we have to convert into the revised affine form the three intervals for x, y, z independently:

$$\hat{x} = \frac{x_{min} + x_{max}}{2} + \frac{x_{max} - x_{min}}{2} \varepsilon_1$$

$$\hat{y} = \frac{y_{min} + y_{max}}{2} + \frac{y_{max} - y_{min}}{2} \varepsilon_1$$

$$\hat{z} = \frac{z_{min} + z_{max}}{2} + \frac{z_{max} - z_{min}}{2} \varepsilon_1$$

If the function range includes a zero value, the cell can be subdivided in octree- or quadtree-like manner until we reach the pixel level and apply the ray-surface intersection. The algorithm for ray-tracing in this case is presented in Algorithm 2.

The same technique can also be used for secondary rays (for example, for a shadow test). In this case we take a bundle of rays from the local area of the implicit surface, calculate the argument range for this bundle of rays and then perform the intersection test with another implicit object.

5. Implementation

In this section we present details of the implementation of ray-tracing of procedurally defined implicit surfaces on the CPU and the GPU. The implementation can be divided into three parts: the Revised Affine Arithmetic representation, the function representation in the revised affine form and the ray-tracing procedure.

Algorithm 2 Ray-tracing of implicit surfaces with cell culling

We start with a bundle of rays from pixels on the interval $[(xs_{min}, ys_{min}), (xs_{max}, ys_{max})]$ in screen space
if $xs_{max} - xs_{min} \leq 1$ AND $ys_{max} - ys_{min} \leq 1$ **then**
  We are at the pixel level, apply ray-tracing procedure for this particular ray
end if
Calculate the interval for the coordinates in object space
Calculate the affine forms for x, y and z
Calculate the affine form F for the function for these affine forms
Get the range of the function from the affine form
if the range of the function does not include a 0 value **then**
  return FALSE (object intersection with this bundle);
end if
Subdivide the interval $[(xs_{min}, ys_{min}), (xs_{max}, ys_{max})]$ into four subintervals by using a quadtree
Repeat the procedure recursively for these subintervals

5.1. Affine form representation

As we stated above, the revised affine form is a polynomial with three terms. Thus, the affine form in the software implementation can be represented as a three-component vector, where the first component represents x_0, the second represents the noise symbol for the error along the ray and the third represents the half-length of the accumulating interval. The calculations in Affine Arithmetic can be performed on these vectors. Almost all of the arithmetic operations have to be overridden as only summation in the Revised Affine Arithmetic matches the standard vector summation. For example, the subtraction and multiplication can be implemented as follows:

```
vec3 ra_subtraction(vec3 x, vec3 y){
vec3 ret;
ret[0] = x[0] - y[0];
ret[1] = x[1] - y[1];
ret[2] = x[2] + y[2];
return ret;
}

vec3 ra_multiplication(vec3 x, vec3 y){
vec3 ret;
ret[0] = x[0]*y[0]+0.5*x[1]*y[1];
ret[1] = x[0]*y[1]+y[0]*x[1];
ret[2] = x[2]*y[2]+
   y[2]*(fabs(x[0])+fabs(x[1]))+
   x[2]*(fabs(y[0])+fabs(y[1])) +
   0.5*fabs(x[1]*y[1]);
return ret;
}
```

Similarly, non-affine operations can be implemented as operations on the three-component vectors. Note that for non-affine operations we are most likely to use the affine

constructor described above. For example, the square root operation needed for CSG models with R-functions can be implemented in this way by using Chebyshev approximation described in [dS97]:

```
vec3 ra_sqrt(vec3 x){
vec2 i = ra_getinterval(x);
if (i[1] < 0) return 0;
if (i[0] < 0) i[0] = 0;
double sq1 = sqrt(i[0]), sq2 = sqrt(i[1]);
//calculate arguments for the revised affine form
double alpha = 1/(sq1+sq2);
double dzeta = (sq1+sq2)/8.0+0.5*sq1*sq2/(sq1+sq2);
double delta = (sq2-sq1)*(sq2-sq1)/(8.0*(sq1+sq2));
//create the revised affine form
vec3 ret;
ret[0] = alpha*x[0]+dzeta;
ret[1] = alpha*x[1];
ret[2] = alpha*x[2]+delta;
return ret;
}
```

In fact, any non-affine operation derived for pure Affine Arithmetic with known *alpha*, *dzeta* and *delta* can be adapted for Revised Affine Arithmetic in the same way.

5.2. Representation of the function

As we stated that our algorithm works with procedurally defined implicit objects, this means that an object can be defined by a real-valued function of real-valued arguments. In the same way this function can be rewritten by using the following rules:

- Calculate the revised affine forms for x, y, z (see example in the section 4.2)
- Each variable depending on the input arguments is replaced by a variable of the revised affine type, while each variable not depending on the input arguments and constants is left in the real form.
- If in the implementation of the Revised Affine Arithmetic operations are overridden, the rest of the code for the initial function is left unchanged, otherwise each operation has to be explicitly replaced by its revised affine version.

The returned value of the rewritten function is the range of the function in the affine form, which is used in the ray-surface intersection procedure described earlier.

Similarly to other ray-tracing techniques for implicit surfaces, we use finite differences to obtain the normal vector for the shading and the secondary rays calculation. Therefore the real-valued defining function of real-valued arguments should be presented as well as the function in the revised affine form.

5.3. Modifications for the GPU implementation

The speed of the calculation can be drastically improved by using hardware acceleration on the GPU. In this section we present possible modifications for the implementation of our algorithm to be accelerated this way.

5.3.1. Ray-surface intersection for the GPU

Both the data structure and the ray-surface intersection procedure can easily be transferred to the GPU. The ray-tracing process is performed in the well-known GPGPU manner: the function definition, the Revised Affine Arithmetic code and the intersection code are in the fragment (pixel) shader which applies to the screen-sized polygon. The GPU-oriented ray-surface intersection procedure without a stack and recursion was proposed in [KHK*09]. However, this implementation does not support argument pruning. In the following procedure we propose the modification of this implementation that supports argument pruning. Note that the usage of arrays requires a modern GPU (for example, the NVidia series 8 and above or any other OpenGL 3 compatible card):

```
bool find_point(in vec3 vecStart, in vec3 vecDir,
     out vec3 vecIntersect)
{
 int mask = 0;
 vec2 vInterval = vec2(0.0,1.0);
 vec3 func = aa_func(vecStart, vecDir, vInterval);
 vec2 interval_func = aa2ia(func);
 if (interval_func.x > 0.0 || interval_func.y < 0)
  return false;
 int d = 0;
 int dlast = int(log2(length(vecDir)/eps));
 if (dlast > 32) dlast = 32;
 vec2 stack[32];
 //
 stack[0] = vec2(0.0,0.5);
 for (;;)
 {
  vInterval = stack[d];
  if (vInterval.y-vInterval.x < eps)
  {
   vecIntersect = vecStart+vInterval.x*vecDir;
   return true;
  }
  func = aa_func(vecStart, vecDir, vInterval);
  interval_func = aa2ia(func);
  if (interval_func.x <= 0.0 && interval_func.y >= 0)
  {
   float t0 = (vInterval.x+vInterval.y)*0.5;
   float t1= (vInterval.y-vInterval.x)*0.5;
   float tmin1 = t0-model.x*t1/model.y-t1*abs(model.z/model.y);
   float tmax1 = t0-model.x*t1/model.y-t1*abs(model.z/model.y);
   if (tmin1 > vInterval.x) vInterval.x = tmin1;
   if (tmax1 < vInterval.y) vInterval.y = tmax1;
   if (d == dlast)
   {
    vecIntersect = vecStart+vInterval.x*vecDir;
    return true;
   }
   else
   {
    d++;
    mask *= 2;
    stack[d] = vec2(vInterval.x, (vInterval.x+vInterval.y)*0.5);
    continue;
   }
  }
  if (mod(mask, 2))
  {
   for (int j = 0; j <= dlast; j++)
   {
    mask /= 2;
    d--;
    if (d == -1) break;
    if (!mod(mask, 2)) break;
   }
   if (d == -1) break;
  }
  mask += 1;
  vInterval = stack[d];
 }
}
```

The description of the general stackless algorithm for ray-tracing can be found in [KHK*09]. In our implementation several changes were introduced to reflect interval pruning. Thus, the interval in each step of the depth is stored in the *stack* variable, the position of the current interval can be found with the depth variable *d* and the variable *mask* stores the current position of the interval related to the traversal procedure. We also have to limit the size of the stack because current graphics hardware does not support dynamic arrays.

5.3.2. Cell culling for the GPU

Because of its recursive nature, cell culling can not be implemented on the GPU directly. However, we can use a simplified version of cell culling in a modified version of the rendering algorithm. The modifications are as follows:

1. Instead of rendering one screen-sized polygon we render a number of non-overlapping screen-sized polygons with a size greater than 1*1 pixels. For example, we fill the screen space with a uniform grid of polygons and display them.
2. For each vertex we store polygon parameters, i.e., screen spaced coordinates.
3. We apply a vertex shader that includes the calculation of object space coordinates, the affine forms for the coordinate variables and the affine form for the function. If the function interval includes a zero value, we set the intersection flag equal to a 1 in the uniform variable, otherwise we store a 0.
4. In the fragment shader we read from the uniform variable for the intersection flag and if its value is 0, we reject this ray, otherwise we calculate the ray-surface intersection as usual.

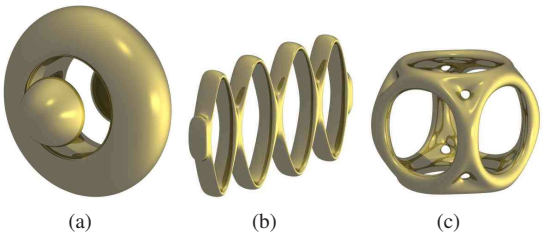
The main idea of these modifications is to store the intersection flag in the vertex shader and to pass it to the fragment shader by using a uniform variables mechanism. Because of the attribute interpolation at the rasterisation step in graphics hardware the pixels where there is no intersection will have a 0 as a value of their uniform variable and will be rejected.

6. Results

In our tests we used a modified version of POV-Ray renderer for the CPU and a stand-alone renderer based on the GLSL language for the GPU. The results were generated on a PC with an Intel Pentium 4 3.20GHz processor and an NVidia 9600 graphics card. Because of the nature of the POV-Ray renderer we do not use cell culling in the CPU renderer, but cell culling is used for GPU ray-tracing.

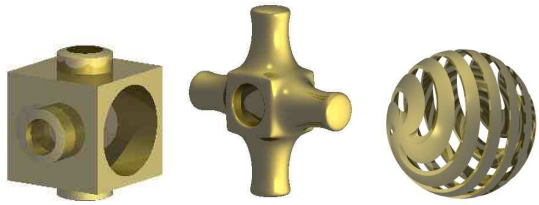
6.1. Offline ray-tracing of procedurally defined implicit surfaces

We tested our ray-tracing algorithm on a wide range of procedurally defined implicit models (see Figs. 3, 4, 5). First,



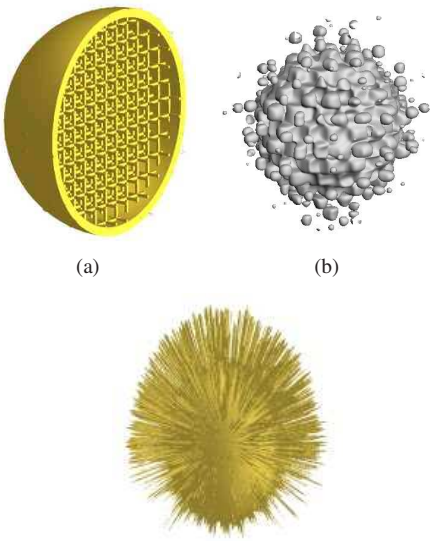
(a) (b) (c)

Figure 3: *Ray-tracing of algebraic surfaces: a) Mitchell b) Bretzel c) Decocube*



(a) (b) (c)

Figure 4: *Ray-tracing of non-algebraic procedural implicit surfaces: a) CSG with R-functions b) CSG with using blending union and blending intersection c) Sphere with trimming*



(a) (b) (c)

Figure 5: *Ray-tracing of procedural implicit surfaces with thin elements or small disjoint components: a) Sphere with microstructure b) Sphere with procedural noise c) Procedural hair*

we compare our procedure with other reliable techniques based on uncertain computations, the technique based on Interval Arithmetic described in [KHK*09], the technique based on pure Affine Arithmetic described in [dCFG99] and the technique based on Reduced Affine Arithmetic described in [GM07b]. The results can be found in the table 1. Note that in works [GM07b] and [KHK*09] Reduced Affine Arithmetic was applied only to arithmetic operations. However, the models starting from CSG and below in this table have non-affine operations other than multiplications. To fairly compare our approach with Reduced Affine Arithmetic we extended it with the affine forms derived for non-affine operations in our work.

The results show that other rendering algorithms based on standard Interval and Affine Arithmetic are significantly slower than our algorithm which is based on Revised Affine Arithmetic. The reason for this is that with Interval Arithmetic algorithms there is an overestimation and with Affine Arithmetic algorithms there is an overestimation as well as a large number of terms in the polynomial form and thus there is a large number of arithmetic operations in the affine operation calculations. Reduced Affine Arithmetic proves to be faster than Interval and standard Affine Arithmetic for algebraic models, however the overestimation of the function in Reduced Affine Arithmetic is wider than the overestimation range for the Revised Affine Arithmetic. Therefore the range of a function based on Revised Affine Arithmetic is tighter than the range in all other techniques and hence the speed of the calculation of the ray-surface intersections is significantly better, especially for models with a large number of non-affine operations.

The reliability of the Revised Affine Arithmetic allows us to test our technique on several procedurally defined implicit models with small features or thin surfaces. For example, by using the proposed ray-surface intersection calculation we can reliably render such models as models with internal structure (see Fig. 5a), stochastic procedural models with disjointed components (see Fig. 5b) and even procedurally-defined hair (see Fig. 5c).

Our ray-tracing technique can be used with complex scenes with a number of procedurally defined implicit objects. For example, we show how a functionally defined scene "Virtual Shikki" [VPP*04] can be rendered using our technique (see Fig. 6). Note that because of the thin elements in the models approximate techniques and polygonization do not work well for this scene unless we use very small steps for the approximate techniques of ray-tracing and a large size for the polygonization grid and hence slow down the rendering process significantly.

6.2. Real-time rendering

We tested the GPU implementation with several procedurally defined models (see Fig. 7). We compared our technique with the technique presented in [KHK*09] for ray-tracing with Interval Arithmetic and Reduced Affine Arithmetic. The results are presented in the table 2. In this table we also present a comparison of our technique using the cell culling procedure in the vertex shader and without the cell culling procedure. As can be seen from the table, Revised Affine Arithmetic gives faster ray-surface intersection – hence the higher speed. The use of cell culling (see Fig. 7b) depends on the nature of the model and can increase the rendering speed for several models.

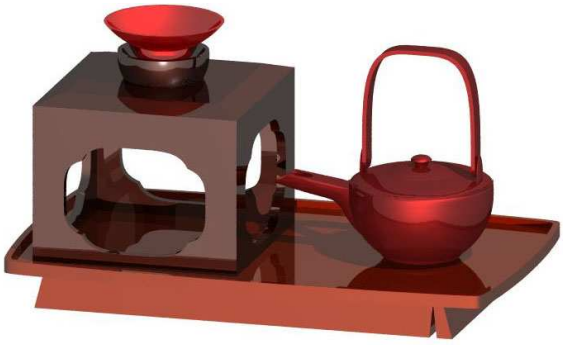
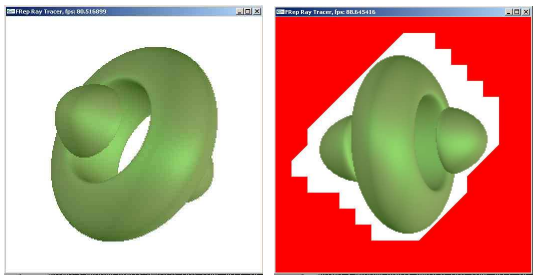


Figure 6: *Ray tracing of procedural scenes: Virtual Shikki*



(a) (b)

Figure 7: *Example of real-time rendering using the GPU: a) The Mitchell surface b) The Mitchell surface with culling (red denotes rays where ray-surface intersection is not calculated because of early culling).*

7. Conclusion

In this paper we presented a technique for ray-tracing of general procedurally defined implicit models based on Revised Affine Arithmetic. By using the inclusion property of Revised Affine Arithmetic we were able to obtain reliable ray-tracing of models and at the same time Revised Affine Arithmetic proved to be the fastest compared to other interval techniques. We also used argument pruning and cell culling to further accelerate the ray-tracing procedure.

Currently the set of procedurally defined implicit models does not include models with conditional operators. Some

	Resolution (pixels)	Number of operations All / Non-affine / Multiplications	IA	AA	RAA*	RevAA
Mitchell	1280*1024	19 / 6 / 6	38	33	7	6
Bretzel	1280*1024	16 / 9 / 9	25	86	22	18
Decocube	1280*1024	30 / 17 / 17	17	226	19	13
CSG	640*480	96 / 40 / 32	126	129	20	18
Sphere with trimming	1024*768	142 / 54 / 37	837	2566	365	285
Sphere with noise	800*600	36 / 11 / 5	17	51	18	9
CSG with blending	640*480	105 / 42 / 32	266	72	34	31
Hair	640*480	88 / 34 / 22	4004	1935	708	658
Sphere with microstructure	640*480	65 / 33 / 22	1006	1079	328	293
Virtual Shikki	320*240	822 / 306 / 213	29244	>50000	422	390

Table 1: *Comparison of the ray-tracing procedures for different computational models. IA stands for Interval Arithmetic, AA for Affine Arithmetic, RAA* for Reduced Affine Arithmetic extended by non-affine operations and RevAA for Revised Affine Arithmetic. The timings for ray-tracing all the rays are shown in seconds.*

	IA	RAA	RevAA (without culling)	RevAA (with culling)
Mitchell	28.1	90.1	92.2	97.4
Bretzel	83.8	90.2	90.2	90.5
Cup	0.56	5.12	5.95	6.1
CSG	4.3	14.6	15.6	17.2

Table 2: *Comparison of ray-tracing procedures on the GPU. IA stands for Interval Arithmetic, AA for Affine Arithmetic, RAA for Reduced Affine Arithmetic and RevAA for Revised Affine Arithmetic. All models were rendered with using only primary rays at a resolution of 512*512 pixels. Timings are shown in FPS (frames per second) and all models were rendered using the same camera parameters.*

research was done using Interval Arithmetic [Dia08], however further research using Affine Arithmetic and especially Revised Affine Arithmetic has to be done in this area. Also, during our tests we found that an affine form can be found not only for standard arithmetic and mathematical operators, but also for any set of operations. We suggest that the calculation of the function can be speeded up by replacing these parts of the code by some special affine functions. This is also an area that merits further research.

References

[BW97] BLOOMENTHAL J., WYVILL B. (Eds.): *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[CD05] CORRIGAN A., DINH H. Q.: Computing and rendering implicit surfaces composed of radial basis functions on the GPU. In *International Workshop on Volume Graphics* (June 2005).

[dCFG99] DE CUSATIS JR. A., FIGUEIREDO L. H., GATTASS M.: Interval methods for ray casting surfaces with affine arithmetic. In *Proceedings of SIBGRAPI'99 - the 12th Brazilian Symposium on Computer Graphics and Image Processing* (1999), pp. 65–71.

[Dia08] DIAZ J. F.: *Improvements in the Ray Tracing of Implicit Surfaces based on Interval Arithmetic*. PhD thesis, Departament d'Electronica, Informatica i Automatica, Universitat de Girona, Girona, Spain, Nov. 2008.

[dS97] DE FIGUEIREDO L. H., STOLFI J.: *Self-Validated Numerical Methods and Applications*. Brazilian Mathematics Colloquium monographs. IMPA/CNPq, Rio de Janeiro, Brazil, 1997.

[FP08] FRYAZINOV O., PASKO A.: Interactive ray shading of FRep objects. In *WSCG' 2008, Communications Papers proceedings* (2008), pp. 145–152.

[Gar84] GARDNER G. Y.: Simulation of natural scenes using textured quadric surfaces. *SIGGRAPH Comput. Graph. 18*, 3 (1984), 11–20.

[GM07a] GAMITO M. N., MADDOCK S. C.: Progressive refinement rendering of implicit surfaces. *Computers & Graphics 31*, 5 (2007), 698–709.

[GM07b] GAMITO M. N., MADDOCK S. C.: Ray casting implicit fractal surfaces with reduced affine arithmetic. *The Visual Computer 23*, 3 (2007), 155–165.

[Har93] HART J. C.: Ray tracing implicit surfaces. In *Siggraph 93 Course Notes: Design, Visualization and Animation of Implicit Surfaces* (1993), pp. 1–16.

[Har94] HART J. C.: Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12* (1994), 527–545.

[Has03] HASAN M.: *An Efficient F-rep Visualization Framework*. Master's thesis, Faculty of Mathemetics, Physics and Informatics, Comenius University, Bratislava, Slovakia, Aug. 2003.

[HSS*05] HADWIGER M., SIGG C., SCHARSACH H., BUHLER K., GROSS M.: Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum 24* (September 2005), 303–312(10).

[KHK*09] KNOLL A., HIJAZI Y., KENSLER A., SCHOTT M., HANSEN C. D., HAGEN H.: Fast ray tracing of arbitrary implicit surfaces with interval and affine arithmetic. *Computer Graphics Forum 28*, 1 (2009), 26–40.

[KOKK06] KANAI T., OHTAKE Y., KAWATA H., KASE K.: GPU-based rendering of sparse low-degree implicit surfaces. In *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia* (2006), pp. 165–171.

[KW03] KRUGER J., WESTERMANN R.: Acceleration techniques for GPU-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (2003), pp. 287–292.

[LB06] LOOP C., BLINN J.: Real-time GPU rendering of piecewise algebraic surfaces. *ACM Transactions on Graphics 25*, 3 (2006), 664–670.

[Mes02] MESSINE F.: Extensions of affine arithmetic: Application to unconstrained global optimization. *Journal of Universal Computer Science 8*, 11 (2002), 992–1015.

[Mit91] MITCHELL D. P.: Three applications of interval analysis in computer graphics. In *Frontiers in Rendering course notes* (1991), pp. 1–13.

[MSVW01] MARTIN R., SHOU H., VOICULESCU I., WANG G.: A comparison of Bernstein hull and affine arithmetic methods for algebraic curve drawing. In *Proc. Uncertainty in Geometric Computations* (July 2001), Kluwer Academic Publishers, pp. 143–154.

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. *ACM TOG 21*, 3 (2002), 703–712.

[PH89] PERLIN K., HOFFERT E. M.: Hypertexture. *SIGGRAPH Comput. Graph. 23*, 3 (1989), 253–262.

[PP04] PASKO G., PASKO A.: Trimming implicit surfaces. *Vis. Comput. 20*, 7 (2004), 437–447.

[Sha07] SHAPIRO V.: Semi-analytic geometry with R-functions. *Acta Numerica 16* (2007), 239–303.

[She99] SHERSTYUK A.: Fast ray tracing of implicit surfaces. *Computer Graphics Forum 18*, 2 (1999), 139–147.

[SNar] SINGH J. M., NARAYANAN P. J.: Real-time ray-tracing of implicit surfaces on the GPU. *IEEE Transactions on Visualization and Computer Graphics* (2009, to appear).

[Sny92] SNYDER J. M.: Interval analysis for computer graphics. In *Computer Graphics* (1992), pp. 121–130.

[VPP*04] VILBRANDT C., PASKO G., PASKO A. A., FAYOLLE P.-A., VILBRANDT T., GOODWIN J. R., GOODWIN J. M., KUNII T. L.: Cultural heritage preservation using constructive shape modeling. *Computer Graphics Forum 23*, 1 (2004), 25–42.

[VSHFar] VU X.-H., SAM-HAROUD D., FALTINGS B.: Enhancing numerical constraint propagation using multiple inclusion representations. *Annals of Mathematics and Artificial Intelligence* (2009, to appear).

Appendix A: Formulas of surfaces used in the paper

Mitchell

$f = 20*(x^2+y^2+z^2) - 4*(x^4+(y^2+z^2)^2) - 17x^2*(y^2+z^2) - 17$

Bretzel

$f = 2 - 60*z^2 - (x^2*(1.21-x^2)^2*(3.8-x^2)^3 - 10*y^2)^2$

Decocube

$f = 0.02 - ((x^2+y^2-0.82)^2+(z^2-1)^2)*((y^2+z^2-0.82)^2+(x^2-1)^2)*((x^2+z^2-0.82)^2+(y^2-1)^2)$

CSG

$f = b|(s\&((c_1 \setminus c_2)|(c_3 \setminus c_4)) \setminus c_5$, where $b = (0.36-x^2)\&(0.36-y^2)\&(0.36-z^2)$, $s = 0.7056-x^2-y^2-z^2$, $c_1 = 0.09-y^2-z^2$, $c_2 = 0.04-y^2-z^2$, $c_3 = 0.09-x^2-z^2$, $c_4 = 0.04-x^2-z^2$, $c_5 = 0.25-x^2-y^2$

Sphere with noise

$f = 81-x^2-y^2-z^2+(3.8*sin(1.5*x)+sin(1.111*x+1.1*sin(1.5*x))*1.624)*(3.8*sin(1.5*y)+sin(1.111*x+1.1*sin(1.5*x))*1.299)*(3.8*sin(1.5*y)+sin(1.111*x+1.1*sin(1.5*x))*2.598)$

Sphere with microstructure

$f = (((1-x^2-y^2-z^2)\&((sin(20*y)-0.9)\&(sin(20*z)-0.9))|((sin(20*x)-0.9)\&(sin(20*z)-0.9))|((sin(20*x)-0.9)\&(sin(20*y)-0.9)))|((1-x^2-y^2-z^2) \setminus (0.75-x^2-y^2-z^2)))\&(-z)$

CSG with blending

$f = (((c_1 \vee_b c_2) \wedge_b s) \vee_b b) \wedge_b (-c_3)$, where $b = (0.36-x^2)\&(0.36-y^2)\&(0.36-z^2)$, $s = 0.7056-x^2-y^2-z^2$, $c_1 = 0.09-y^2-z^2$, $c_2 = 0.09-x^2-z^2$, $c_3 = 0.25-x^2-y^2$, $\vee_b$ denotes blending intersection: $f_1 \vee_b f_2 = f_1+f_2-\sqrt{f_1^2+f_2^2}+\frac{0.5}{1+f_1^2+f_2^2}$, $\wedge_b$ denotes blending union: $f_1 \wedge_b f_2 = f_1+f_2+\sqrt{f_1^2+f_2^2}+\frac{0.5}{1+f_1^2+f_2^2}$

Hair

$f = o|(((1.8*sin(1.8*x*\frac{9}{\sqrt{x^2+y^2+z^2}})+s_x)*(1.8*sin(1.8*y*\frac{9}{\sqrt{x^2+y^2+z^2}})+s_y)*(1.8*sin(1.8*z*\frac{9}{\sqrt{x^2+y^2+z^2}})+s_z)-10)\&(o+2)\&y)$, where $o = (1-\frac{x^2}{16}-\frac{y^2}{36}-\frac{z^2}{16})|(1-\frac{x^2}{1.6129}-\frac{(y+2.5)^2}{2.25}-\frac{(z-3)^2}{1.6129})$, $s_x = 1.538*sin(1.33*x*\frac{9}{\sqrt{x^2+y^2+z^2}}+1.4*sin(1.8*x*\frac{9}{\sqrt{x^2+y^2+z^2}}))$, $s_y = 1.538*sin(1.33*y*\frac{9}{\sqrt{x^2+y^2+z^2}}+1.4*sin(1.8*y*\frac{9}{\sqrt{x^2+y^2+z^2}}))$, $s_z = 1.538*sin(1.33*z*\frac{9}{\sqrt{x^2+y^2+z^2}}+1.4*sin(1.8*z*\frac{9}{\sqrt{x^2+y^2+z^2}}))$

Sphere with trimming: described in [PP04]

Virtual Shikki: files in HyperFun format can be found here: http://www.hyperfun.org/App/shi/Shikki.html

In formulas & denotes set-theoretic intersection with R-functions: $f_1\&f_2 = f_1+f_2-\sqrt{f_1^2+f_2^2}$, | denotes set-theoretic union with R-functions: $f_1\&f_2 = f_1+f_2+\sqrt{f_1^2+f_2^2}$ and $\setminus$ denotes set-theoretic subtraction with R-functions: $f_1 \setminus f_2 = f_1-f_2-\sqrt{f_1^2+f_2^2}$