Beam casting implicit surfaces on the GPU with interval arithmetic

Francisco Ganacim, Luiz Henrique de Figueiredo, Diego Nehab
IMPA – Instituto Nacional de Matemática Pura e Aplicada, Rio de Janeiro, Brazil

***Abstract*—We present a GPU-based beam-casting method for rendering implicit surfaces in real time with anti-aliasing. We use interval arithmetic to model the beams and to detect their intersections with the surface. We show how beams can be used to quickly discard large empty regions in the image, thus leading to a fast adaptive subdivision method.**

***Keywords*-implicit surfaces; parallel algorithms; rendering**

I. Introduction

Rendering surfaces with ray casting is perhaps the clearest example of a potentially embarrassingly parallel problem in computers graphics: in principle, all pixels can be computed in parallel. In this paper, we present a case study of rendering implicit surfaces in parallel in the GPU using CUDA.

In the simplest case, rendering implicit surfaces with ray casting by point sampling requires the solution of one non-linear univariate equation for each pixel. (Using supersampling to reduce alias requires solving more equations per pixel.) Since current GPUs do not have a million processors, we cannot render large images with one processor per pixel and some acceleration scheme is still needed. We employ *adaptive beam tracing* based on interval arithmetic to guarantee that all parts of the surface are seen and to quickly discard pixels that do not see the surface. To achieve maximum performance, current GPUs require that similar work is done by each processor. We discuss how to organize the computation under this restriction.

II. Related Work

Specialized rendering methods exist for some classes of surfaces, especially algebraic surfaces, i.e., implicit surfaces defined by polynomial equations. Hanrahan [1] was one of the first to describe a ray tracing method that exploits Descartes's rule of signs to isolate roots of polynomials. He also suggested the possibility of implementing ray casting in hardware to exploit parallelism.

Interval methods were used in ray casting for parametric surfaces by Toth [2] and by Barth et al. [3] and for implicit surfaces by Mitchell [4], by Duff [5], and by Stolte [6]. Affine arithmetic [7] was used to render implicit surfaces by Figueiredo and Stolfi [8] and by Cusatis Jr. et al. [9]. Beams were introduced in rendering by Heckbert and Hanrahan [10].

With the advent of the GPU, interest in fast rendering of implicit surfaces has been renewed. Loop and Blinn [11] described specialized methods for rendering low-degree piecewise algebraic surfaces in Bernstein form. Seland and Dokken [12] rendered low-degree algebraic surfaces by using blossoms of trivariate Bernstein–Bézier functions over tetrahedra. The Bernstein–Bézier form allows a form of interval analysis. Knoll et al. [13] described CPU and GPU methods using both interval and affine arithmetic. Singh and Narayanan [14] ray cast implicit surfaces by sampling rays adaptively based on the distance to the surface and the closeness to a silhouette. They handle multiple roots by using interval analysis based on first-order Taylor approximations.

Closer to our approach is the work of Flórez et al. [15], [16] who described an interval beam-casting method for rendering implicit surfaces with adaptive anti-aliasing. Their methods ran on the CPU and took several minutes to produce high-quality images. Flórez's thesis [17] described how to improve performance using a GPU or a cluster of workstations, achieving a couple of seconds per frame at the time.

III. Beam Casting

We shall present two rendering algorithms that cast beams toward the surface. A *beam* is a right prism having a rectangular base in image space and contains all rays that can be cast starting in its base. The first beam-casting algorithm performs *uniform* subdivision of the image and uses one beam per pixel. The second beam-casting algorithm performs *adaptive* subdivision of the image and uses large beams to eliminate large empty regions in the image. Both methods use interval bisection along the beam to locate the surface.

A. Interval bisection

To render a surface given implicitly by $f(x,y,z)=0$ with beam casting, we need to find the intersection of a beam with the surface. We do this by performing *interval bisection* on the beam, which recursively divides the beam into blocks, searching for a small block nearest the viewer that contains points of the surface. The recursion is guided by the evaluation of an interval extension F of the implicit function f. The simplest interval extension is the *natural* interval extension, obtained by replacing all primitive operations and functions in the expression of f with their interval counterparts [18], [19]. Interval extensions are not limited to algebraic functions and are easy to implement using operator overloading, which CUDA supports.

The crucial property of an interval extension F is that for each box B in $\mathbf{R}^3$, $F(B)$ is an interval containing $f(B)$. Thus, if for a block B we find that $0 \notin F(B)$, then a fortiori we know that $0 \notin f(B)$ and we have *proved* that the block cannot cross the surface and so can be discarded. Interval bisection is able to discard most beams that are far away from the surface. However, since $F(B)$ usually contains $f(B)$ properly, we cannot conclude from $0 \in F(B)$ that the surface does cross B. As a consequence, beams near the surface may not be discarded as early as possible.

B. Interval beam casting

We model a *beam* as $U \times V \times T$, where $U \times V$ is a *base* rectangle in image space and T is the depth interval, starting at the near plane and ending at the far plane. A beam is a *pixel beam* when its base corresponds to the area of a pixel.

The workhorse of both our methods is the *BeamCast* procedure, which performs interval bisection on the beam looking for an enclosure of the part of the surface seen from the base of the beam. *BeamCast* stops searching either when it can prove that the beam does see the surface or when the beam block is thin enough, as measured by a tolerance ε.

The interval function F requires boxes in world coordinates and *BeamCast* starts by converting the beam $U \times V \times T$ to an axis-aligned box $X \times Y \times Z$ in world coordinates. Evaluating F on that box gives an interval estimate of the entire range of f in the beam. As mentioned earlier, this estimate is very likely an overestimate, especially because the box can be much larger than the beam. Nevertheless, if the estimate does not contain 0, we can be sure that the beam does not intersect the surface. If *BeamCast* cannot discard a beam block for this reason, it bisects the block by splitting it across the depth direction and explores the two sub-blocks, taking care to explore the sub-block nearest to the viewer first, so that it can report the visible part if any.

When the bisection cannot discard a beam, it finds a small block that is likely to contain a piece of the surface.

C. Uniform versus adaptive subdivision

The *Uniform* method simply calls *BeamCast* for each pixel beam and paints the image with the background color if *BeamCast* returns the empty interval or with the proper surface color computed at the center of the returned interval otherwise. The *Adaptive* method calls *BeamCast* for beams with large bases, starting with the entire image.

To improve the uniform subdivision method, we use two simple but important observations, which are exploited in the *Subdivide* procedure. The first one is that empty regions in the image are generally larger than the area of just one pixel. We exploit this fact and eliminate large regions at once by using beams with larger bases. The second observation is that the interval returned by *BeamCast* gives us an approximation to the position of the piece of the surface seen by the beam. Thus, after we cast a large beam, we can use the lower bound of the returned interval as the start depth for its sub-beams.

Subdivide takes a beam base $U \times V$, corresponding to an region of the image, a starting depth w for the beam, and a user-supplied tolerance ε. The value d is used to track the recursion level. As described earlier, *Subdivide* paints the base rectangle with the background color if it can discard the beam using *BeamCast*. Otherwise, *Subdivide* splits the rectangle

```
procedure $BeamCast(U,V,T,\varepsilon)$
  $B \leftarrow Box(U \times V \times T)$
  if $0 \notin F(B)$ then
    return $\emptyset$
  else
    if $diam(T) < \varepsilon$ then
      return $T$
    else
      $T_1, T_2 \leftarrow split(T)$
      $J \leftarrow BeamCast(U,V,T_1)$
      if $J \neq \emptyset$ then
        return $J$
      else
        return $BeamCast(U,V,T_2)$
      end
    end
  end
end

procedure $Paint(U,V,T)$
  if $T = \emptyset$ then
    $I(U,V) \leftarrow background$
  else
    $I(U,V) \leftarrow color(center(U,V,T))$
  end
end

procedure $Uniform()$
  for all pixels $U \times V$ in the image $U_0 \times V_0$ do
    $Paint(U,V,BeamCast(U,V,[near,far],\varepsilon))$
  end
end

procedure $Subdivide(U,V,w,d,\varepsilon)$
  $T \leftarrow BeamCast(U,V,[w,far],\varepsilon)$
  if $T = \emptyset$ or $d = maxdepth$ then
    $Paint(U,V,T)$
  else
    $U_1, U_2 \leftarrow split(U)$
    $V_1, V_2 \leftarrow split(V)$
    $a \leftarrow \min(T)$
    $Subdivide(U_1,V_1,a,d+1,\varepsilon/2)$
    $Subdivide(U_1,V_2,a,d+1,\varepsilon/2)$
    $Subdivide(U_2,V_1,a,d+1,\varepsilon/2)$
    $Subdivide(U_2,V_2,a,d+1,\varepsilon/2)$
  end
end

procedure $Adaptive()$
  $Subdivide(U_0,V_0,near,0,\varepsilon)$
end
```

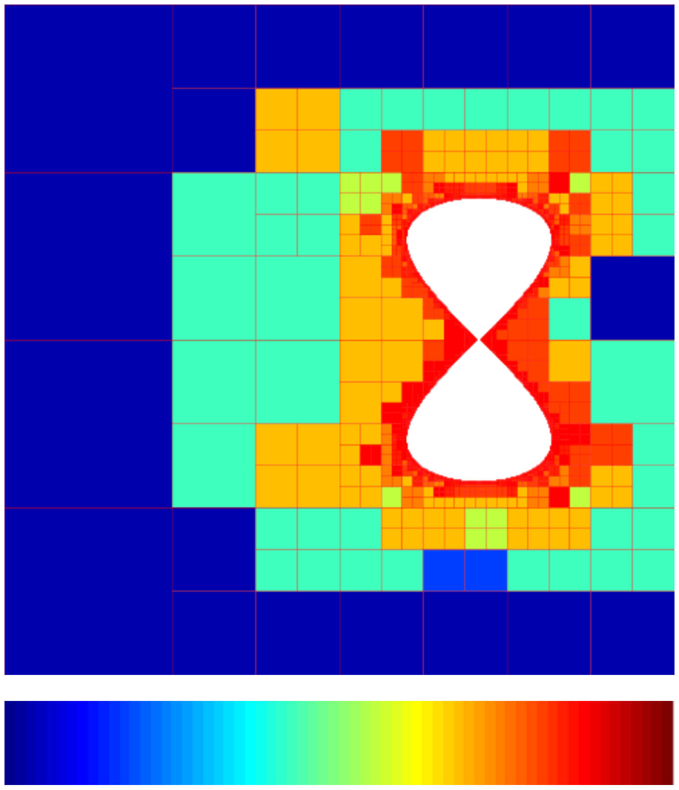


Fig. 1. Behavior of the adaptive method for the *Octdong* surface. The colors represent the level at which the boxes were eliminated. The subdivision reaches the maximum level at the white pixels, which show the surface

into four equal parts and recursively calls itself for each part. If *Subdivide* reaches the maximum allowed depth, then the surface is sampled and the base rectangle is painted with the corresponding color. Note that care is taken during recursion to adjust the tolerance to $\varepsilon/2$, since for beams with large bases, the tolerance for *BeamCast* can be large as well, but as we split the bases to gain accuracy, we also need to use a smaller tolerance for *BeamCast*.

The behavior of the adaptive method is illustrated in Figure 1. Note how regions that are far from the surface are eliminated with larger beams while regions closer to the surface require smaller beams before they are eliminated. This reflects the overestimation inherent to interval methods.

D. Anti-aliasing

One simple way to reduce the aliasing in the rendering is to take multiple samples per pixel and combine the results to find the pixel color. This is easily achieved in *Subdivide* by increasing the value of *maxdepth* beyond the resolution of the desired image and by making *color* combine the contributions of subpixels. Moreover, by increasing *maxdepth*, we also increase the precision of the algorithm along the depth direction, which gives us not only better images, but also a better approximation of the surface geometry, thus mitigating the effects of interval overestimation.

E. Implementation details

In our implementation we use OpenGL, GLSL, and CUDA. We start by using OpenGL to draw a quad covering the whole screen area. This quad is then textured by a shader written in GLSL. The actual surface rendering happens when we generate the texture. We allocate a texture buffer in the GPU main memory and then map it to the CUDA address space. In CUDA, we fill the buffer with the pixel colors.

The implementation of *Uniform* is straightforward. First, we create a 2D CUDA grid with the same dimensions of the final texture. Each thread in the grid executes the CUDA kernel that implements *BeamCast*. The U and V parameters of *BeamCast* are easily derived from the thread indexes, the T parameter is the interval $[-1,1]$, and ε is set to a constant. The result of the computation is saved in a memory buffer and later used in the *Color* CUDA kernel, which implements the shading using the normal computed from the gradient ∇f.

The implementation of *BeamCast* in a CUDA kernel has a few subtleties due to the CUDA architecture and to hardware limitations. In particular, we cannot use recursive functions. To circumvent this limitation we used a stack-based approach, because the bisection algorithm can be seen as a depth-first search on a binary tree. The naive implementation of a stack proved to have very low performance, due to the time needed for a CUDA thread to perform a global memory transaction. Achieving high performance in a GPU requires careful management of accesses to global memory and manual caching of data in shared memory for repeated use. Thus, we implemented the stack using one integer to hold the status of the depth search. This integer stores in its bits the path taken by the search; the number of bits used shows the depth. This integer is stored in a thread register, allowing extremely fast updates.

We can see *Subdivide* as a breadth-first search on a quadtree in image space. At a given level, each cell is the base of a beam, for which we call *BeamCast*. The result of *BeamCast* is an interval that gives a lower bound to the position of the surface on the cell. If this interval is not empty, then the cell is subdivided and its children are queued to be processed in the next level of the quadtree. If the interval is empty, the cell is discarded. The cells of the active level of the quadtree are processed concurrently by threads displaced in a 2D CUDA grid. The output of one level is stored in a memory buffer and is used as input to the next call. To ensure that we are using the maximum processing power of the GPU at each CUDA call, we start the the adaptive process by finding the shallower subdivision level that fills the GPU. After all levels are processed, the result is stored in a buffer and then used by the shading CUDA kernel.

We have tried to reduce warp divergence by grouping threads according to the spatial position of the beams.

IV. Results

We tested the uniform and the adaptive beam-casting methods on the surfaces shown in Figures 2, 3, and 4. Each pair of images in these figures shows the same surface rendered with one sample per pixel (left) and with four samples per pixel as anti-aliasing (right). We have included a Blinn blob [20] to show that the method is not restricted to algebraic surfaces.

The performance of the methods for rendering these surfaces is shown in Tables I and II for two image resolutions, 512×512 and 1024×1024 pixels. The results are given in frames per second. We also give the relative performance of the two methods. All tests were made with an NVidia GTX 470.

In almost all cases, we achieve real-time frame rates with both methods. In all cases the adaptive method significantly outperforms the uniform method. Moreover, for complex functions the adaptive method gives near real-time frame rates even for large high-quality images. As expected, the performance depends on the complexity of the function defining the surface: the performance decreases for complex expressions due to interval overestimation caused by correlations and probably also due to increased register pressure.

TABLE I
PERFORMANCE IN FPS FOR 512×512 IMAGES

	1 sample per pixel			4 samples per pixel		
	Unif	Adap	A/U	Unif	Adap	A/U
Sphere	860	1120	1.30	280	464	1.65
Dingdong	815	947	1.16	284	425	1.49
Klein	274	362	1.32	76	133	1.75
Mitchell	189	240	1.26	45	72	1.60
Octdong	544	660	1.21	173	267	1.54
Steiner	532	840	1.57	168	389	2.31
Tangle	185	244	1.31	49	86	1.75
Teardrop	842	1287	1.52	283	580	2.04
Barth	235	287	1.22	67	105	1.56
Heart	574	806	1.40	180	339	1.88
Chmutov	212	333	1.57	50	95	1.90
Blob	781	1322	1.69	257	570	2.21

TABLE II
PERFORMANCE IN FPS FOR 1024×1024 IMAGES

	1 sample per pixel			4 samples per pixel		
	Unif	Adap	A/U	Unif	Adap	A/U
Sphere	300	447	1.49	79	140	1.77
Dingdong	300	412	1.37	83	142	1.71
Klein	88	131	1.48	21	41	1.95
Mitchell	54	70	1.29	13	23	1.76
Octdong	190	263	1.38	52	87	1.67
Steiner	180	377	2.09	48	137	2.85
Tangle	56	85	1.51	14	27	1.92
Teardrop	288	555	1.92	80	196	2.45
Barth	77	104	1.35	20	35	1.75
Heart	193	332	1.72	51	110	2.15
Chmutov	60	94	1.56	13	20	1.53
Blob	262	545	2.08	70	182	2.60

V. Conclusion

We have presented two simple methods for beam casting implicit surfaces on the GPU. The first one uniformly samples the surface using beam casting at each pixel. The second one performs an adaptive space subdivision, using large beams to eliminate empty regions and to optimize the casting of smaller beams. We have shown that the adaptive subdivision method can be used to efficiently super-sample the surface, generating anti-aliased images in real time.

Flórez et al. [15] described a similar adaptive subdivision method, but for the CPU only. Their depth bisection method is performed down to machine precision to classify image regions into three classes (outside, inside, and undefined) to detect aliasing. In our method the maximum level of depth bisection depends on the level of the cell in the quadtree: larger beams are bisected fewer times. We cannot expect to reach machine precision in the depth bisection of beams with large bases because the size of a cell limits the precision of the interval evaluation on the corresponding beam blocks.

The CPU method by Flórez et al. [16] is similar in structure to our uniform method but their anti-aliasing procedure is more sophisticated. The GPU method in Flórez's thesis [17] is similar to our uniform method but did not achieve real time.

Directions for future work

The first, and most natural, extension for this work is to replace naive interval arithmetic with affine arithmetic for interval evaluation [7]. Besides the expected improvement in the convergence speed due to quadratic convergence and exploitation of first-order correlations, we believe that affine arithmetic will help diminish the overestimation problems related to the camera transformation because affine arithmetic can represent beams directly, even if they are not axis-aligned.

We also wish to understand better the role of thread divergence inside the CUDA warps. In our implementation each beam is processed by one thread. If the threads in one warp diverge, i.e., execute different computations, a huge performance loss is inflicted. For instance, even if all but one of the beams inside a warp are eliminated in the first steps of *BeamCast*, all threads will still execute the full computation until the remaining beam is eliminated or is entirely processed. We plan to decrease thread divergence by using a thread remapping mechanism, based on the geometric position of the beam, relative to the surface.

Other extensions for this work include casting reflection and refraction beams and shadow beams.

Acknowledgments

This work is part of the first author's M.Sc. work at IMPA. The second author is partially supported by CNPq. This work was done in the Visgraf laboratory at IMPA, which is sponsored by CNPq, FAPERJ, FINEP, and IBM Brasil.

References

[1] P. Hanrahan, "Ray tracing algebraic surfaces," in *ACM SIGGRAPH '83*, pp. 83–90.
[2] D. L. Toth, "On ray tracing parametric surfaces," in *ACM SIGGRAPH '85*, pp. 171–179.
[3] W. Barth, R. Lieger, and M. Schindler, "Ray tracing general parametric surfaces using interval arithmetic," *The Visual Computer*, vol. 10, no. 7, pp. 363–371, 1994.
[4] D. P. Mitchell, "Robust ray intersection with interval arithmetic," in *Graphics Interface '90*, pp. 68–74.
[5] T. Duff, "Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry," in *ACM SIGGRAPH '92*, pp. 131–138.

[6] N. Stolte and A. Kaufman, “Novel techniques for robust voxelization and visualization of implicit surfaces,” *Graphical Models*, vol. 63, pp. 387–412, November 2001.

[7] L. H. de Figueiredo and J. Stolfi, “Affine arithmetic: concepts and applications,” *Numerical Algorithms*, vol. 37, no. 1–4, pp. 147–158, 2004.

[8] ——, “Adaptive enumeration of implicit surfaces with affine arithmetic,” *Computer Graphics Forum*, vol. 15, no. 5, pp. 287–296, 1996.

[9] A. de Cusatis Jr., L. H. de Figueiredo, and M. Gattass, “Interval methods for ray casting implicit surfaces with affine arithmetic,” in *SIBGRAPI ’99*. IEEE Press, 1999, pp. 65–71.

[10] P. S. Heckbert and P. Hanrahan, “Beam tracing polygonal objects,” in *ACM SIGGRAPH ’84*, pp. 119–127, 1984.

[11] C. Loop and J. Blinn, “Real-time GPU rendering of piecewise algebraic surfaces,” in *ACM SIGGRAPH ’06*, pp. 664–670.

[12] J. Seland and T. Dokken, “Real-time algebraic surface visualization,” in *Geometric Modelling, Numerical Simulation, and Optimization*, G. Hasle, K.-A. Lie, and E. Quak, Eds. Springer, 2007, pp. 163–183.

[13] A. Knoll, Y. Hijazi, A. Kensler, M. Schott, C. Hansen, and H. Hagen, “Fast ray tracing of arbitrary implicit surfaces with interval and affine arithmetic,” *Computer Graphics Forum*, vol. 28, no. 1, pp. 26–40, 2009.

[14] J. Singh and P. Narayanan, “Real-time ray tracing of implicit surfaces on the GPU,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 2, pp. 261–272, 2010.

[15] J. Flórez, M. Sbert, M. A. Sainz, and J. Vehí, “Efficient ray tracing using interval analysis,” in *PPAM ’07, Lecture Notes in Computer Science,* vol. 4967. Springer, 2008, pp. 1351–1360.

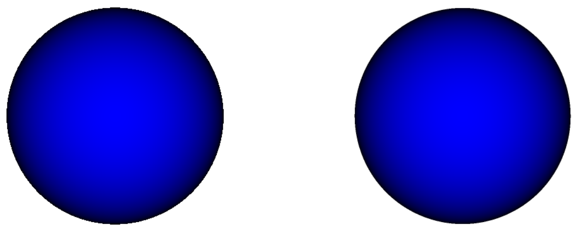
[16] J. Flórez, M. Sbert, M. Sainz, and J. Vehí, “Improved adaptive antialiasing for ray tracing implicit surfaces,” *International Journal of Computer Information Systems and Industrial Management Applications*, pp. 9–14, 2009.

[17] J. Flórez, “Improvements in the ray tracing of implicit surfaces based on interval arithmetic,” Ph.D. dissertation, Universitat de Girona, 2008.

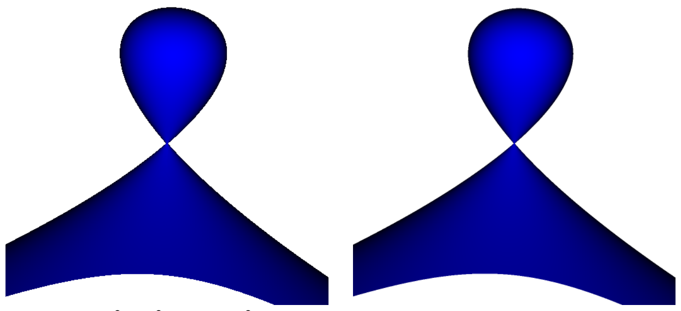
[18] R. E. Moore, *Interval Analysis*. Prentice-Hall, 1966.

[19] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*. SIAM, 2009.

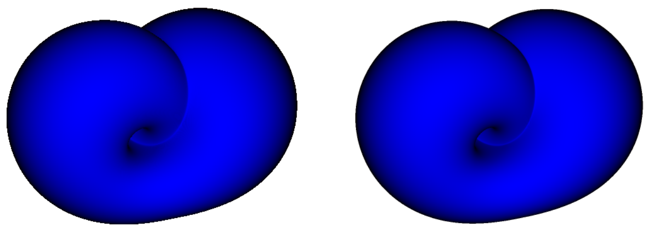
[20] J. F. Blinn, “A generalization of algebraic surface drawing,” *ACM Transactions on Graphics*, vol. 1, pp. 235–256, 1982.



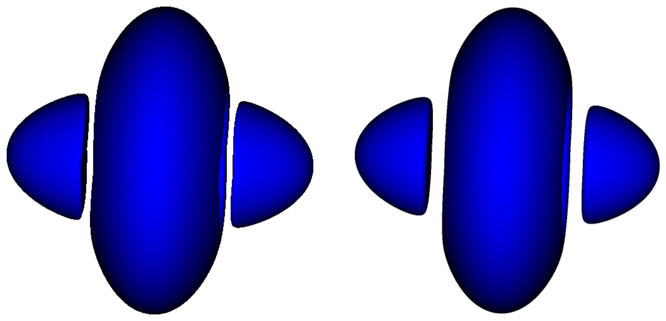
Sphere: $x^2+y^2+z^2-r^2=0$.



Dingdong: $x^2+y^2-z(1-z^2)=0$.

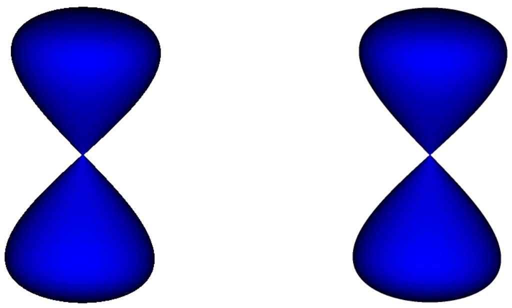


Klein: $(x^2+y^2+z^2+2y-1)((x^2+y^2+z^2-2y-1)^2-8z^2)+16xz(x^2+y^2+z^2-2y-1)=0$.

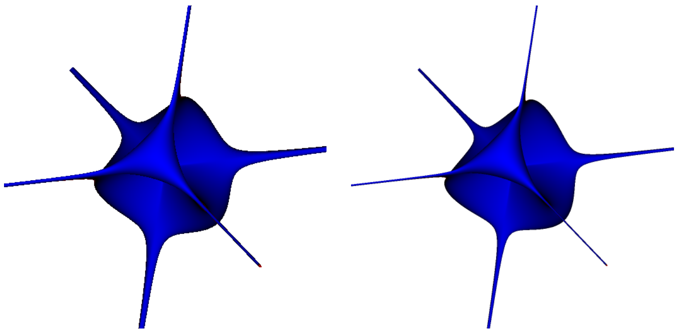


Mitchell: $4(x^4+(y^2+z^2)^2+17x^2(y^2+z^2))-20(x^2+y^2+z^2)+17=0$.

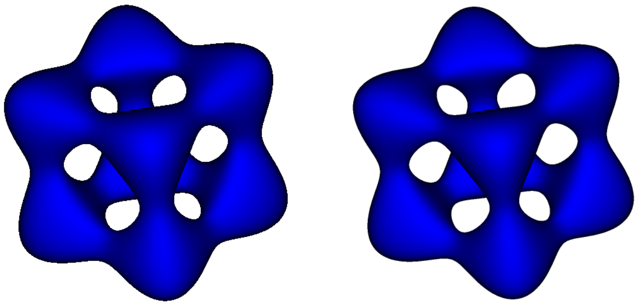
Fig. 2. Results: 1 sample per pixel (left) and 4 samples per pixel (right).



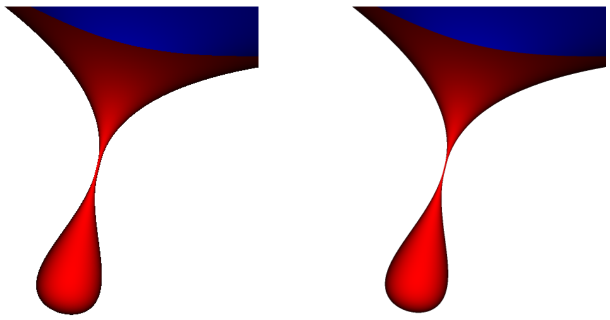
Octdong: $x^2+y^2+z^4-z^2=0$.



Steiner: $x^2y^2+y^2z^2+x^2z^2+xyz=0$.

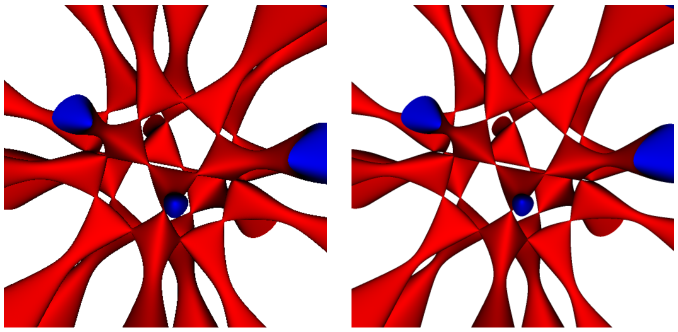


Tangle: $x^4-5x^2+y^4-5y^2+z^4-5z^2+11.8=0$.

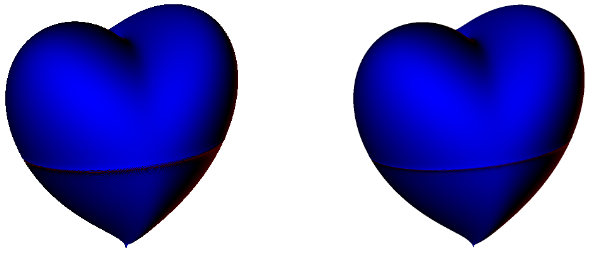


Teardrop:$0,5x^5+0,5x^4-y^2-z^2=0$.

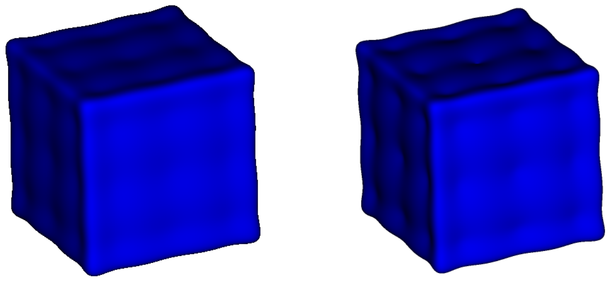
Fig. 3. Results: 1 sample per pixel (left) and 4 samples per pixel (right).



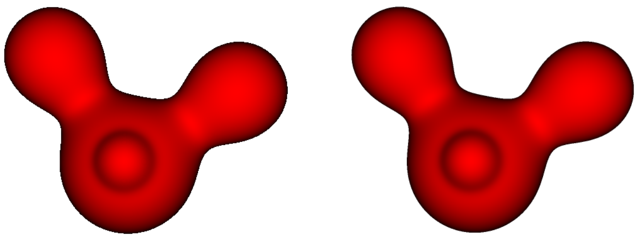
Barth:$4(c^2x^2-y^2)(c^2y^2-z^2)(c^2z^2-x^2)-(1+2c)(x^2+y^2+z^2-1)^2=0$, $c=(1+\sqrt{5})/2$.



Heart: $(x^2+9/4y^2+z^2-1)^3-x^2z^3-9/80y^2z^3=0$.



Chumtov: $f(x)+f(y)+f(z)+1=0$, $f(t)=64t^7-112t^5+56t^3-7t$.



Blinn Blob: $\sum_{j=1}^{4} D_j(p)-1=0$, $D_j(p)=b_j\exp(-a_j|p-r_j|^2)$, $r_1=(0,0,0)$, $r_2=(1,0,0)$, $r_3=(0,1,0)$, $r_4=(0,0,0.5)$, $a_1=4$, $a_2=a_3=8$, $a_4=16$, $b_1=b_2=b_3=b_4=e$.

Fig. 4. Results: 1 sample per pixel (left) and 4 samples per pixel (right).