Iterative Methods for Visualization of Implicit Surfaces on GPU

Rodrigo Toledo, Bruno Lévy, Jean-Claude Paul

Iterative Methods for Visualization of Implicit Surfaces on GPU

Rodrigo de Toledo[1], Bruno Levy[2], and Jean-Claude Paul[3]

[1] Tecgraf – PUC-Rio, Brazil
[2] INRIA – ALICE, France
[3] Tsinghua University, China

Abstract. The ray-casting of implicit surfaces on GPU has been explored in the last few years. However, until recently, they were restricted to second degree (quadrics). We present an iterative solution to ray cast cubics and quartics on GPU. Our solution targets efficient implementation, obtaining interactive rendering for thousands of surfaces per frame. We have given special attention to torus rendering since it is a useful shape for multiple CAD models. We have tested four different iterative methods, including a novel one, comparing them with classical tessellation solution.

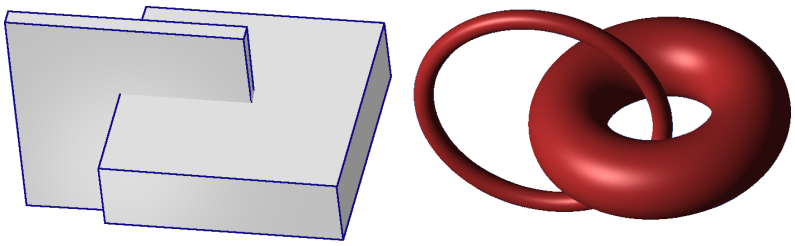

Fig. 1. The faces of two bounding boxes are used to trigger the fragment shader responsible for rendering the tori.

1 Introduction

When programmable GPU were designed, the main goal was to produce images with better quality, while using standard triangle rasterization. However, this innovation gave large flexibility to vertex and pixel processing, motivating some completely new applications. One promising research area is the creation of new GPU primitives, extending the default ones (triangle, line and point). These new primitives do not aim to substitute the typical ones, but work together. The first one to appear in the literature was the quadrilateral primitive [1], which is based on mean value coordinates [2]. Implicit surfaces have also been directly implemented on GPU based on ray casting, but, until recently, they were restricted

to second order ones. Spheres, ellipsoids and cylinders are some examples of quadrics ray-casted on GPU used in applications for molecule rendering [3, 4] and tensor-field visualization [5]. The benefits of this new primitives compared to tessellation are: image quality (precise silhouette and per-pixel depth/shading), less memory usage and rendering efficiency.

Higher-order implicit primitives are a challenge for GPU implementation. Shader languages still have important restrictions when compared to CPU programming, including no support for stacks and recursive functions. We are interested in rendering cubics and quartics with a scalable implementation. The goal is to use them in practical applications that can benefit from this speed-up. Systems for massive model visualization are an example of target use. Among possible implicit surfaces of third and fourth degree, torus is the one with the most useful shape. It is largely found in CAD models. According to Nourse et al. [6] and Requicha et al. [7], 85% of industrial objects are described by plans and quadrics and this number grows to 95% if toroidal patches are also allowed. For this reason we have devoted special attention to torus as a new GPU primitive.

Loop and Blinn [8] were the first ones that investigated GPU implicit primitives with degree up to four. In their work, the intersection equation (between ray and surface) is solved using *analytic* techniques. In contrast, we adopted *iterative* methods, which resulted in faster rendering and more precise surfaces. We have used the Sturm method for both cubics and quartics. In the specific case of tori, we have tested four different approaches, including a novel one called *double derivative bisection*. In this work, we target per-pixel and per-vertex optimizations that result in high performance. We have measured the rendering speed for a single torus and also for multiple tori (up to 16,000 tori at the same time in the screen).

2 Related Work

GPU primitives are visualized through a ray-casting algorithm implemented inside the graphics card. The main computations are done in the pixel stage of the pipeline. To trigger the per-pixel algorithm, it is still necessary to raster a set of standard primitives. To keep GPU primitives compatible with rasterized surfaces, the visibility issue between objects is solved by the z-buffer, updated by both rasterization and ray-casting methods.

The concept of extended GPU primitives was first introduced by Toledo and Levy [9]. They have created a framework to render quadrics on GPU without tessellation. It is possible to visualize these surfaces with textures and self-shadows. The potential applications mentioned in their work were molecule rendering (using spheres) and tensor of curvature visualization (using ellipsoids). Later, Romeiro et al. [10] extended the original idea to reconstruct CSG models.

Bajaj et al. [3] have developed special GPU primitives for molecule visualization: spheres, cylinders and *helices*. The later are a derivation of cylinders used to represent secondary structures on molecules. They have reported interactive rendering for molecules with up to 100,000 atoms.

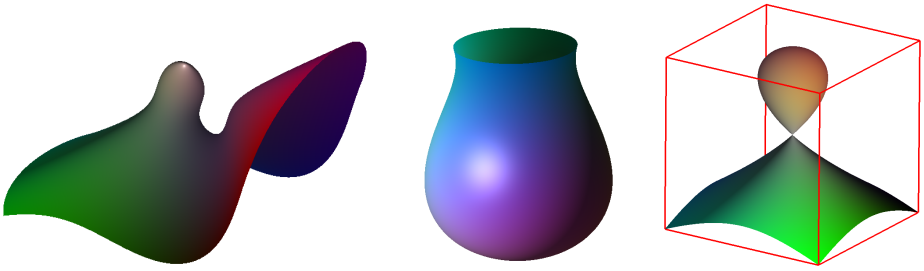

Fig. 2. Cubic surface examples. They are all clipped outside the domain $[-1, 1]^3$ (represented by a cube in the last image).

Kondratieva et al. [5] and Sigg et al. [4] exclusively use points (`GL_POINTS`) as the rasterized primitive that triggers the fragment shader. Kondratieva et al. make use of ellipsoid-GPU primitives for diffusion tensor field visualization. Sigg et al. proposed a more general quadratic surfaces approach but their examples were restricted to molecule visualization with spheres and cylinders.

Loop and Blinn [8] broke through the second-order barrier and introduced the first GPU primitives up to fourth order. Based on Bézier tetrahedron, they succeeded in rendering cubics and quartics. However, the approach adopted by Loop and Blinn have some drawbacks in speed-up and quality. The performance limitation are a result of several conditions: (i) exhaustive per-fragment computation to solve all roots; (ii) excessive waste of fragments that are rasterized but not rendered (for example, the tetrahedron does not tidily fit the torus); (iii) complex per-vertex computation to transform Bézier vertices; and (iv) view-dependent CPU computation per tetrahedron. From the quality point of view, Loop and Blinn solution has a lack of precision near silhouette edges and self intersections. This is a result of numerical noise that can be aggravated depending on the choice of near and far clipping planes, because of computations that are not done on local coordinate space. A positive aspect of their implementation is the possibility of rendering piecewise algebraic surfaces, decomposed in several tetrahedrons.

3 Cubics

We extended the idea of GPU primitives for cubic implicit surfaces. It is possible to form interesting surfaces (see Figure 2) by using cubic equations, but these shapes are not as popular as the classical quadrics (e.g., sphere, cylinders, cones). Comparing to quadrics, the computation is much more intense.

The difficult in implementing the cubic primitive is the intersection searching process. It includes a root find problem of third degree. There are many polynomial root finders known in the literature [11] that could work in the cubic case. We choose to use a binary search based on the *Sturm theorem*. The binary

search is a good solution in our case, since we have a restricted domain (we use a bounding box with local coordinates in $[-1,1]^3$).

Sturm Theorem The theorem is based on a set of functions, known as *Sturm functions*, which are derivate from the base function $f(x)$:

$$f_0(x) = f(x), \quad f_1(x) = f'(x)$$
$$f_n(x) = -\left\{ f_{n-2}(x) - f_{n-1}(x) \left[\frac{f_{n-2}(x)}{f_{n-1}(x)} \right] \right\}, \quad n \geq 2$$

So, for a cubic function, there are four Sturm functions we need to generate (the last one, f_3, is a constant). With them, it is possible to find the number of real roots of an algebraic equation over an interval. After evaluating the set of functions for the two points defining the interval, the difference in the number of sign changes between them gives the number of roots in the interval.

Algorithm Based on Sturm theorem, we can do a binary search to find the first positive intersection between the ray $R : (x, y, z) = o + t\boldsymbol{v}$ and the cubic surface. The initial interval is between $t_1 = 0$ and $t_2 = \lambda$, where the point $P_2 = o + \lambda\boldsymbol{v}$ is the point where the ray leaves the domain $[-1,1]^3$. At each iteration, the interval is divided into two, $t_m = \frac{t_1+t_2}{2}$. Remark that, since we search only for the first intersection, we can use the number of sign changes ($n1$) of the origin point ($t = 0$) in all iterations. So, we just recompute the function values and the number of sign changes for the searching point (t_m).

Results Sturm algorithm, although fast, is not as high performance as quadric ray casting. We achieved 300 fps in a GeForce 7900 graphics card. The quality of the results is very good (see Figure 2). However, in some special situations (close to singular points), the computation exceeds the precision of the GPU and some errors may appear, which are evident after zooming. A positive point of Sturm approach is the discard that is done before loop. If the fragment is not discarded at this moment, it means that there is at least one intersection. Another interesting consequence of using binary search is the possibility of adaptive quality. We can adjust the number of iterations according to the surface distance to the camera (Level Of Detail), improving speed when the surface is viewed from afar.

4 Quartics and Tori

We use Sturm technique, described on previous section, for generic quartics. It demands one more function evaluation than for cubics. In this section, we have given special attention to the GPU torus because, among all cubics and quartics, it is the most common primitive found in massive CAD environments (for example, a quarter of torus is typically used for pipe junctions).

For the torus GPU primitive we use a well fitted bounding box to trigger the fragments running our shader. A torus can be described by a quartic implicit function. Equation 1 defines a zero centered torus with main direction in z. This canonical position is the one used by our fragment shader to ray cast the torus, including one more definition: biggest and smallest radii sum is 1 ($R + r = 1$).

Given a ray $\boldsymbol{R} : (x, y, z) = o + t\boldsymbol{v}$, where $\boldsymbol{v} = [v_x, v_y, v_z]$ is a normalized vector, Equation 2 describes the ray-torus intersection[4].

$$(x^2 + y^2 + z^2 - (r^2 + R^2))^2 - 4R^2(r^2 - z^2) = 0 \quad (1)$$

$$T(t) : at^4 + bt^3 + ct^2 + dt + e = 0 \quad (2)$$

where $a = 1,\ b = 4(\boldsymbol{o} \cdot \boldsymbol{v})$

$$c = 2((\boldsymbol{o} \cdot \boldsymbol{o}) - (R^2 + r^2) + 2(\boldsymbol{o} \cdot \boldsymbol{v})^2 + 2R^2(v_z)^2)$$

$$d = 4((\boldsymbol{o} \cdot \boldsymbol{v})((\boldsymbol{o} \cdot \boldsymbol{o}) - (R^2 + r^2)) + 2R^2 v_z o_z)$$

$$e = ((\boldsymbol{o} \cdot \boldsymbol{o}) - (R^2 + r^2))^2 - 4R^2(r^2 - o_z^2)$$

Finding the root of a quartic equation for several pixels in interactive rates is not a simple task. We have tried four different approaches (described in the following subsections) to choose an adequate algorithm for our GPU torus.

4.1 Sturm

We have extended the algorithm used in our cubic GPU primitive (see Section 3). Compared to ray-cubic intersection, ray-torus intersection has an extra computation since there is one more function to be evaluated in each iteration, totaling five functions. As a result, Sturm is not so fast for solving the torus-ray intersection (see Table 1). Another problem with the Sturm approach is the numerical precision. The complexity of terms on each Sturm function may overflow the floating-point capacity. This problem is viewing-angle dependent and in some cases may produce incorrect images (see Figure 4).

4.2 Double Derivative Bisection

This technique is also a binary search, as in Sturm technique. The idea is an extension of *Bisection method*. In this method, given two points t_0 and t_1, where $f(t_0)$ and $f(t_1)$ have different signs, we can ensure that there is at least one root (where $f(x)$ is a continuous function). Using the interval's midpoint $t_m = 0.5(t_0 + t_1)$ we evaluate the function, $f(t_m)$. Based on its sign we reduce the interval to be between t_0 and t_m or to be between t_m and t_1. This is a simple method that is always successful.

Derivative Bisection We extend the bisection algorithm for working on two other special situations: both $f(t_0)$ and $f(t_1)$ have positive signs but with one and only one local minimum in the interval; and both $f(t_0)$ and $f(t_1)$ have negative signs but with one and only one local maximum in the interval. In these circumstances we can guarantee that, if there is an intersection (actually, up to two intersections), the *Derivative Bisection* algorithm can find it (them).

The algorithm does a binary search for the local minimum/maximum similarly to simple bisection but verifying the sign of derivatives. If the searching

[4] We use a correction of Hanrahan equation [12] suggested by Eric Haines (see: http://steve.hollasch.net/cgindex/render/raytorus.html).

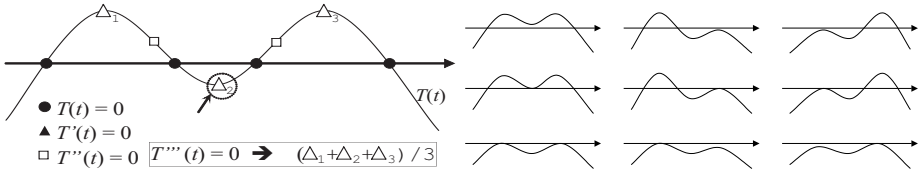

Fig. 3. *Left:* The round points mark the roots, the triangles mark the first derivative roots and the squares mark the second derivative root. The third derivative root is actually the average of the first derivative roots (see Equation 4). Its location must be somewhere around the second first-derivative root ($\triangle_2$). *Right:* All possible plots for the function $T(t)$ where there is at least one root (or one ray-torus intersection), except the four roots case.

point (t_m) crosses the abscissa (in other words, $f(t_m)$ changes the sign), the algorithm switches for a simple bisection search. If local minimum/maximum is found and $f(t_m)$ did not change the sign, then there were no roots.

Double Derivative Bisection for Torus The ray-torus intersection involves the solution of an equation of fourth degree (Equation 2), which yields a maximum of four possible intersections. One can easily imagine a ray traversing a torus and crossing its boundary four times. If we plot the evaluation of the torus intersection function $T(t)$ for this four-times crossing ray, we obtain something close to the form presented in Figure 3 (*left*). T has at most two local maximum and at most one local minimum (see all possible cases for T in Figure 3).

The basic idea in the *Double Derivative Bisection* is to initially divide the problem into two, running the single derivative bisection twice, first on the portion before the local minimum ($\triangle_2$ in Figure 3) and, if no intersection was found, on the portion right after. However, finding exactly the local minimum includes the solution of the derivative equation (Equation 3), which is a third-degree equation. Instead, we approximate the local minimum location by a much simpler computation. We compute the third-derivative root, which is the average of the three first-derivative roots (based on Vieta's Formulas):

$$
\begin{aligned}
& T'(t) \ : \ 4at^3 + 3bt^2 + 2ct + d = 0 \qquad (3)\\
& T'(t) = (t - \triangle_1)(t - \triangle_2)(t - \triangle_3) \quad \Rightarrow \quad \frac{3b}{4} = -(\triangle_1 + \triangle_2 + \triangle_3)\\
& T'''(t) = 4t + b\,, \quad T'''(t_M) = 0 \quad \Rightarrow \quad t_M = -\frac{b}{4} = \frac{\triangle_1 + \triangle_2 + \triangle_3}{3}
\end{aligned}
$$

The computation to find the root of the third derivative ($T'''(t_M) = 0$) is very simple since it uses only b, whose value is $4(\boldsymbol{o} \cdot \boldsymbol{v})$: $t_M = -(\boldsymbol{o} \cdot \boldsymbol{v})$. Although the third derivative is only an approximation (see Figure 3), it is good enough to divide the root finding algorithm into two for applying the derivative bisection (that is why we call our technique: *double derivative bisection*).

Double derivative bisection is slower than the Sturm technique (see Table 1). However, we have got results without the numerical problems found with Sturm, guaranteeing an error $\epsilon(r) \leq 0.0014$ relative to the minor radius r.

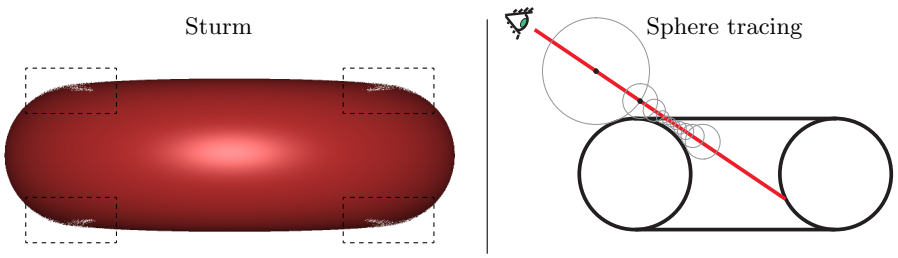

Fig. 4. *Sturm* Due to floating-point imprecision, Sturm method results in visual defects (which are more evident in the viewing-angle shown in this picture). *Sphere tracing* Example of critical situation for ray-torus intersection. The ray almost touches the torus reducing the step size. Therefore, more iterations are necessary.

4.3 Sphere Tracing

The sphere tracing was proposed by Hart in 1996 [13]. The idea is to find the ray-intersection by stepping closer and closer through the ray. Given the Euclidean distance function $d(x)$ to a surface, we can march along the ray from point x the distance $d(x)$ without penetrating the surface. For our canonical torus described in Equation 1, the distance function is $d(x) = ||(||(x, y)|| - R, z)|| - r$.

Compared to other methods, sphere tracing for ray-torus intersection needs less computation in each iteration. However, there are some critical situations whose approximation is very slow, increasing the iterations (see Figure 4).

To overcome these critical points, we have tested two sphere tracing for each ray with different starting points. The first one starts on the entering point in the torus bounding box, the second one starts on t_M (see Section 4.2). If after some iterations with the first sphere tracing $d_i(x)$ becomes greater than $d_{i-1}(x)$ then we proceed with the second sphere tracing.

With our two-rays implementation, we achieved better performance than the single-ray (see Table 1). However, the convergence of this algorithm is still slow. In the next subsection we present the implementation that resulted in the best performance among the four ones we have tried.

4.4 Newton-Raphson

The Newton-Raphson method (a.k.a. *Newton's method*) also uses the derivative evaluation in each iteration (as in Sturm and in double derivative bisection). The Newton's formula derives from the Taylor series, which is:

$$f(x + \delta) \approx f(x) + f'(x)\delta + \frac{f''(x)}{2}\delta^2 + \cdots \tag{4}$$

If δ is small enough, we can ignore the high-order terms so, for each iteration, we can move a δ step with:

$$\delta = -\frac{f(x)}{f'(x)}. \tag{5}$$

Newton-Raphson algorithm converges quadratically. This means that near a root, the algorithm doubles the significant digits after each step [11]. However, far

from the root, it may have a bad behavior. For example, if the current position is too close to a local extreme, the derivative is almost zero and δ vanishes to infinite. For this reason we took some extra cares in our GPU ray-torus intersection implementation.

Implementation. We use a well fit bounding box around the torus from where the ray-casting starts. Therefore, the starting point is not so far from the root, but we can still push it forward. Before applying the Newton iterations, we start by executing a simple ray-sphere intersection (the sphere radius is the sum of the torus radii). If there is no intersection, we can discard the fragment; if the intersection is negative (before the starting point), we ignore it; and if the intersection is positive, then we move the starting point to this position. However, it is still possible to have a local extreme between the starting and the intersection points. To avoid a vanishing situation, we use some bounds to guarantee that the step is not bigger than λ. Empirically, we found that $\lambda = 0.15$ (relative to our canonical torus) efficiently avoids the vanishing cases without loosing performance. The Newton-Raphson algorithm gave the best performance for our GPU-Torus. We use a threshold to stop the iterations that assures an error smaller than 0.1% of the minor radius r ($\epsilon(r) \leq 0.001$). We have also tested with $\epsilon(r) \leq 0.00003$. The results are shown in Table 1.

4.5 Normal Computation

One possible way to compute the normal vector for a point lying in the torus surface is by taking the three partial derivatives of the torus function at this point, which is quite expensive. Actually, we have implemented a geometric solution. Considering the canonical torus (Equation 1), the normalized normal $\boldsymbol{n}$ at the point P lying on torus surface is:

$$\boldsymbol{n}_{torus} = \frac{P - C}{r} \text{ , where } \quad C = \begin{cases} C_{xy} = \parallel P_{xy} \parallel \\ C_z = 0 \end{cases} \tag{6}$$

5 GPU Torus Results

5.1 Rendering One Torus

We have done several tests measuring the performance of each one of the four GPU Torus methods: Sturm, Bisection, SphereTracing and Newton. We have considered two different implementations for SphereTracing (one-ray and two-rays) and two implementations for Newton (varying the threshold). The results are presented in Table 1, which also contains the performance of traditional polygonal rasterization method. We have measured the frame rate from different angles for each different method, averaging them on the last column of Table 1. Among all GPU Primitive methods, *Newton 0.001* have presented the best performance. As presented in next subsection, the performance of our GPU torus becomes interesting for multiple tori. However, we can see that even for individual torus, we can obtain competitive numbers. For an error $\epsilon(R + r) \approx 0.000350$, the polygonal version is slightly better. On the other hand, for an error $\epsilon(R + r) \approx 0.000015$, the GPU Torus is much faster.

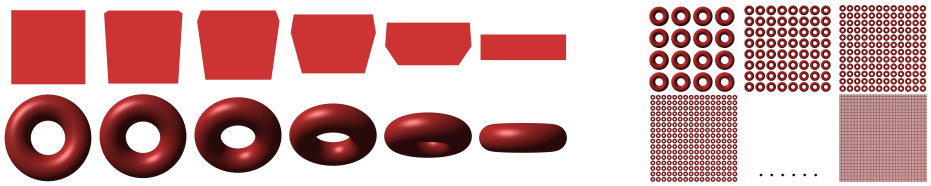

Fig. 5. *Left:* Several viewing-angles used for testing torus rendering performance (see Table 1). In the first row the bounding-box used for our GPU primitive, in the second row the torus itself. *Right:* Rendering multiple tori for the results in Figure 6.

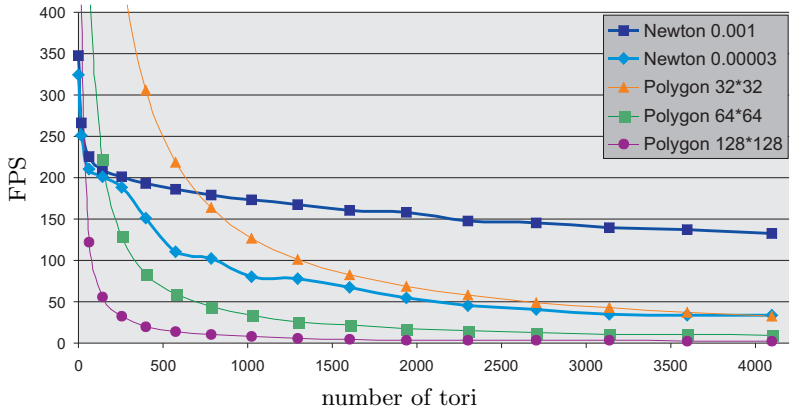

Fig. 6. Performance of different techniques for multiple tori rendered on the screen.

5.2 Rendering Multiple Tori

One advantage of our GPU torus is that the bottleneck is no longer on the vertex stage, but on pixel stage. It means that the performance will not be reduced as much as using the polygonal torus version when increasing the number of tori. In Figure 6 we compare the GPU Newton technique in two different thresholds with three different resolutions of polygonal torus. We can verify that *Newton 0.001* is the fastest for more than 700 tori. Comparing *Newton 0.001* and *polygonal 128*128* (which have equivalent error) the Newton method is always faster. For more than 16000 tori, *Newton 0.001* is the only one that keeps an interactive frame rate (50 fps).

6 Conclusion and Future Work

The iterative methods presented in this paper are faster than analytical solutions for ray-casting, mainly because they compute only one root (the one responsible for the first ray-surface intersection). The tests presented in Table 1 were recorded in a 1024×1024 viewport, if we reduce to 640×480 the GPU torus

with *Newton 0.001* method renders at 1300 fps in comparison to 500 fps obtained by analytical approach [8].

In future work, it is possible to extend the GPU iterative methods to higher order surfaces. However, some numerical precision problems may appear. A possible future application for our GPU tori could be their use for CAD models visualization. For instance, in a industrial environment, tori (and slices of torus) are easily found in tubular structures, chains and CAD patterns. In this kind of application, the tori must be used in combination with triangle meshes and with other extended GPU primitives, such as cylinders and cones. The frame rate obtained for multiple tori corroborates for the use of GPU primitives for massive CAD models visualization.

References

1. Hormann, K., Tarini, M.: A quadrilateral rendering primitive. In: HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, New York, NY, USA, ACM Press (2004) 7–14
2. Floater, M.S.: Mean value coordinates. Comput. Aided Geom. Des. **20** (2003) 19–27
3. Bajaj, C., Djeu, P., Siddavanahalli, V., Thane, A.: Texmol: Interactive visual exploration of large flexible multi-component molecular complexes. In: VIS '04: Proceedings of the conference on Visualization '04, Washington, DC, USA, IEEE Computer Society (2004) 243–250
4. Christian Sigg, Tim Weyrich, M.B., Gross, M.: Gpu-based ray-casting of quadratic surfaces. In: Symposium on Point-Based Graphics, ACM Siggraph (2006) 59–65
5. Kondratieva, P., Krüger, J., Westermann, R.: The application of gpu particle tracing to diffusion tensor field visualization. In: Proceedings IEEE Visualization 2005. (2005)
6. Nourse, B., Hakala, D.G., Hillyard, R., Malraison, P.: Natural quadrics in mechanical design. Autofact West **1** (1980) 363–378
7. Requicha, A.A.G., Voelcker, H.B.: Solid modeling: a historical summary and contemporary assessment. IEEE Computer Graphics and Applications **2** (1982) 9–22
8. Loop, C., Blinn, J.: Real-time gpu rendering of piecewise algebraic surfaces. In: SIGGRAPH '06: ACM SIGGRAPH 2006 Papers, New York, NY, USA, ACM Press (2006) 664–670
9. Toledo, R., Levy, B.: Extending the graphic pipeline with new gpu-accelerated primitives. In: International gOcad Meeting, Nancy, France. (2004) also presented in Visgraf Seminar 2004, IMPA, Rio de Janeiro, Brazil.
10. Romeiro, F., de Figueiredo, L.H., Velho, L.: Hardware-assisted rendering of csg models. In: Proceedings of SIBGRAPI 2006 - XIX Brazilian Symposium on Computer Graphics and Image Processing, Manaus, SBC - Sociedade Brasileira de Computacao, IEEE Press (2006) –
11. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in C: The Art of Scientific Computing (2nd ed.). Cambridge University Press, Cambridge (1992) ISBN 0-521-43108-5.
12. Hanrahan, P.: A Survey of Ray - Surface Intersection Algorithms. In: An introduction to ray tracing. Academic Press Ltd. (1989) 33–77
13. Hart, J.C.: Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. The Visual Computer **12** (1996) 527–545

	Projection Angle						Avg		
	0°	18°	36°	54°	72°	90°			
Bbox pixels	524 k	516 k	495 k	430 k	324 k	209 k	416 k		
Torus pixels	312 k	305 k	285 k	256 k	203 k	146 k	251 k		
Fragment waste	40.4%	40.9%	42.2%	40.5%	37.3%	29.8%	38.5%	*Error* (10^{-6})	
	FPS							$\epsilon(r)$	$\epsilon(R+r)$
Sturm	68	68	71	82	108	163	93	25	11
D.D. Bisection	15	15	16	19	25	46	23	1400	600
Sphere Tracing	16	16	16	20	27	46	24	1000	429
2-rays S. Tracing	66	66	65	80	100	153	88	1000	429
Newton 0.00003	246	254	261	302	380	537	330	30	13
Newton 0.001	267	283	291	324	410	590	**361**	1000	429
Polygon 32*32	2653	2649	2668	2704	2730	2744	2691	11236	4815
Polygon 64*64	1238	1238	1238	1238	1238	1238	1238	2811	1205
Polygon 128*128	369	369	369	369	369	369	369	703	301
Polygon 256*256	96	96	96	96	96	96	96	176	75
Polygon 512*512	24	24	24	24	24	24	24	44	19
	Megapixels/second							StdDev	$\frac{\text{StdDev}}{\text{Average}}$
Sturm	35.64	35.16	35.15	35.32	35.06	34.07	35.07	0.485	1.38%
Bisection	7.86	7.75	7.92	8.18	8.12	9.61	8.24	0.631	7.65%
Sphere Tracing	8.39	8.27	7.92	8.61	8.76	9.61	8.60	0.527	6.14%
2-rays S. Tracing	34.60	34.12	32.18	34.46	32.46	31.98	33.30	1.111	3.34%
Newton 0,00003	128.95	131.31	129.22	130.08	123.36	112.23	125.86	6.587	5.23%
Newton 0,001	139.95	146.31	144.07	139.56	133.10	123.31	137.72	7.654	5.56%

Table 1. Comparison between several torus rendering techniques. The tests were done in a 1024×1024 viewport with a torus filling the window, with a GeForce 7900 graphics card. **Projection Angle:** The number of pixels of the ray-casting area projection is determinant for the final frame rate and it varies according to the torus angle. For this reason we have done tests with 6 different viewing-angles (including top and perpendicular viewing), see Figure 5. **Fragment waste:** Number of fragments discarded divided by total fragments. **Polygon $N * N$:** Polygonal version of torus with N rings and N sides. ***Error*:** Our GPU torus techniques use an error threshold measured relative to the smaller radius: $\epsilon(r)$. We computed the error of the polygonal tori relative to their total radius: $\epsilon(R+r) = 1 - \cos\left(\frac{\pi}{N}\right)$. We can extract from one error the other one by using the radii proportion of our testing torus ($r = 0.3$ and $R = 0.7$). **Megapixels/second:** It is the corresponding multiplication of FPS and bounding-box pixels. This number indicates how many times the ray-casting algorithm was executed (in millions) per second. **StdDev:** Some of our ray-casting techniques suffer different per-pixel performance depending on the viewing-angle. To identify this fact we computed the standard deviation of each technique. Bisection and SphereTracing techniques had the most view-dependent performance.