Вісник Харківського національного університету
Серія «Математичне моделювання. Інформаційні технології. Автоматизовані системи управління»
№ 703, 2005, с.232-244

УДК 681.3

Visualization of Algebraic Surfaces

Gennadiy Chuyeshov
Kharkiv Karazin National University, Ukraine

This paper deals with the visualization of algebraic surfaces in three-dimensional space using ray tracing algorithm. This algorithm makes it possible to reduce the problem to solving a series of algebraic equations of a single variable for each pixel and the main task is to choose the most appropriate and the fastest method. It is shown that the Dekker-Brent method allows to save computation time for certain types of surfaces in comparison with commonly used Chord (Regula Falsi) method. The software, which visualizes algebraic surfaces with the control of intermediate computations, was developed

Introduction

The problem of visualization of data (i.e. plotting of a three-dimensional image of an object on computer monitor) of scientific investigations has always attracted substantial attention of researchers (see, e.g., [1, 5] and the references therein). In the course of a study a researcher may face with a problem of visualization of different types of data. Sometimes these data can be represented as mathematical functions. In this case visualization techniques depends crucially on a way a function is defined – explicit, implicit or parametric – and on the number of independent variables.

Visualization of an implicitly defined surface (given by an equation of the form $f(x, y, z) = 0$) is quite a commonly arose task in computational mathematics. Nowadays several algorithms are available.

One of these algorithms was written by *Henderson* [7]. Given a point of an implicitly defined surface with full rank Jacobian, the implicit function theorem guarantees that there exists a bijection from some neighborhood in the tangent space onto the surface. Starting from this point on the surface, Henderson's algorithm computes this bijective mapping from a small elliptic region L of the tangent space using multiple applications of Newton's method. Then the algorithm picks a point on the boundary of the image of L and computes the mapping onto a small neighborhood of this point. Since the point lies on the boundary of the previously computed region, this mapping always gives new information about the desired surface. A limitation of this algorithm is that it cannot process singular points (i.e. points with degenerated Jacobian).

Another algorithm is called *"Marching Cubes"* and described in the book by Angel [1]. The ground principle of this algorithm is a spatial subdivision into a series of small cubes. The algorithm tests the corner points of each cube and assigns its status ("+" or "–") depending on the sign of the function f at this point. Then it replaces the cube with an appropriate set of polygons which approximate the surface under construction. Every cube has $2^8 = 256$ possible combinations of corner status. By [1] the original 256 combinations of corner status can be resolved down to a total of 15

combinations; with this number it is easy to create pattern polygon sets. The union of all the polygons generated will be an approximation of the desired surface. This method is convenient and mainly applies in the case when one recovers a surface from a discrete data array.

The third algorithm is known as the *ray tracing* (see, e.g., Hill [5] or Angel [1]). This is a powerful general tool for rendering surfaces. The key point of this algorithm is as follows. Rays of light are traced from an eye back through a screen into a scene. This means that we start from the eye or camera and trace the ray through a pixel of the screen into the scene and determine whether it intersects any object in the scene or not. In other words, the major task is to find out what does the eye sees though every pixel of the screen.

From the computational point of view the main problem arising in an application of the ray tracing is to find ray-object intersection. The solution of this problem highly depends on the class of the surfaces one deals with. Nowadays ray tracing algorithm is used to visualize prisms and surfaces of revolution [6], algebraic [4] and fractal [6] surfaces. On the basis of this algorithm works are underway towards the visualization of other types of objects (see [5]).

This paper deals with visualization of a class consisting of implicit algebraic surfaces in three-dimensional space, i.e. the surfaces given by an equation of the form $P(x, y, z) = 0$, where P is polynomial with real coefficients. Due to the idea of ray tracing algorithm instead of this equation we can consider the equation of intersection between the ray and the surface which can be transformed to the algebraic equation $P^*(t) = 0$, where $P^*(t)$ is the polynomial of a single variable constructed from P. Highly developed and numerous methods for solving of algebraic equations are available (see, e.g., [3, 10, 11, 13]). And the main task is to choose the most appropriate and the fastest method. In this paper we use Dekker-Brent method (see, e.g., [3, 10, 11]). As our computations show, it makes possible to save computation time for certain types of surfaces in comparison with Chord (Regula Falsi) method (see, e.g., [10, 11, 13]) used by Hanrahan in [4].

1. Ray Tracing

This section deals with the ray tracing algorithm and its application to visualization of implicit surfaces. The particularies of this algorithm are explained in details.

1.1. Algorithm. Consider a screen (see Fig. 1) as a two-dimensional array of pixels in the space. A pixel is a rectangle-shaped unit which may be assigned only one color at a time and its color is the color of the light ray that passes from the object, though that pixel, into the eye. Ray is a theoretically infinite semi-line used for modeling a thin beam of light which starts in one point and extends in one direction. The algorithm starts by shooting rays from the eye through each pixel of the screen, determining all the objects that intersect the ray, and finding the nearest of those intersections for each ray. To find ray-object intersections we must put the parametric representation for the ray into the equation for the object and determine whether there exists a real solution of the equation obtained. If this solution exists then there is an intersection and we must choose the closest point of intersection (e.g. point H on Fig. 1). The algorithm shoots several rays from this point of intersection in order to

see (i) what objects are reflected at that point; (ii) what objects may be seen through this point; and (iii) which light sources are directly visible from that point. These additional rays are called *secondary rays* as distinct from the original, *primary ray*. The secondary rays which are sent towards the light sources to determine if any object occludes the intersection point are called *shadow rays*. On Fig. 1 one can see a primary ray sent from point E through the screen into the scene where it intersects *Object 1* at point H. There are also two shadow rays sent from point H to the light sources L_1 and L_2.

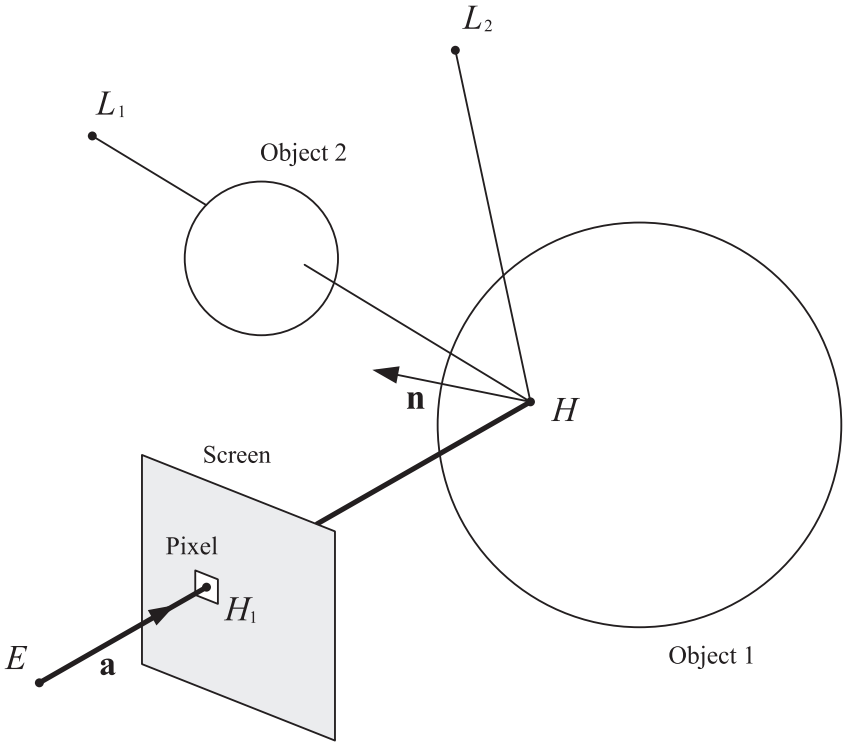

Figure 1. Ray Tracing.

It should be noted that it is more correct to refer to ray tracing algorithm as *backward* ray tracing algorithm. The point is that light rays, in fact, travel from light sources to an eye, but not from the eye of a viewer to the light source as it is assumed in the ray tracing algorithm. However we will neglect this difference and will use term "ray tracing" in the description of the method of following light from an eye to a light source.

1.2. Rays. As it was stated above a ray is an infinite semi-line used for modeling a thin beam of light which starts in one point and extends in one direction. Thus to describe a ray we can use the following parametric representation

$$\begin{cases} x(t) = x_s + d_x t, \\ y(t) = y_s + d_y t, \\ z(t) = z_s + d_z t, \end{cases} \tag{1}$$

where (x_s, y_s, z_s) are coordinates of the point H_1 which is the starting point of the ray, (d_x, d_y, d_z) are coordinates of the unit vector a which is the direction of the ray and t is a positive parameter (see Fig. 1).

1.3. Implicit Surfaces. Let f be a continuous scalar function defined on a domain $D \subseteq \mathbb{R}^3$. The *implicit surface* S generated by the function f is the locus of points at which the function takes on the value zero, i.e.

$$S = \{(x, y, z) \in D : f(x, y, z) = 0\}. \tag{2}$$

As mentioned in the previous subsection to find a ray-surface intersection we must put the parametric representation for the ray, namely (1), into the equation for the surface, namely $f(x, y, z) = 0$. This yields the following equation

$$f(x_s + d_x t, y_s + d_y t, z_s + d_z t) = 0 \tag{3}$$

with respect to a single variable t. Thus on the semi-axis $\mathbb{R}_+$ we have the equation

$$g(t) = 0, \quad t \in \mathbb{R}_+ \tag{4}$$

where $g(t) = f(x_s + d_x t, y_s + d_y t, z_s + d_z t)$.

The next step is to determine whether there exists a root of equation (4), because if there are no real roots of this equation then the ray does not intersect the surface. It is natural to restrict the visualization of a surface to some volume which presents a part of the real space (this volume is user-defined and lies behind the screen). In further considerations we call this volume as *extent*. Here two types of the extents are involved: a parallelepiped and a sphere. As these extents are convex, a ray may intersect any of them through an interval only and it is quite easy to calculate the end points of this interval, t_{in} and t_{out} where $0 \le t_{in} \le t_{out}$. Surely, the interval $[t_{in}, t_{out}]$ depends on the parameters of the ray $(x_s, y_s, z_s, d_x, d_y, d_z)$. For every ray we first check whether it intersects the extent and if it does, we then determine t_{in} and t_{out}. The task is now to check whether this ray intersects the surface. If the ray intersects the surface, then the parameter t_0 of the intersection point lies between t_{in} and t_{out}, i.e., $t_{in} \le t_0 \le t_{out}$.

Assume that $\{t_1, ..., t_k\} \subset [t_{in}, t_{out}]$ are roots of equation (4). It is clear that to visualize the surface we should find a ray-surface intersection which possesses the properties (i) it is situated behind the screen and (ii) it is closest to the screen. The intersection which satisfies these properties corresponds to the smallest solution of (4) from the interval $[t_{in}, t_{out}]$, i.e.,

$$t_{\min} = \min_{i=1...k} t_i .$$

Having found this root we can calculate the coordinates (x_0, y_0, z_0) of the point on the surface by putting this root into equation (1) of the ray. This, in turn, allows us to find the surface normal vector at this point by the following formula

$$\mathbf{n} = \nabla f(x_0, y_0, z_0) \equiv \left(\frac{\partial f}{\partial x}(x_0, y_0, z_0), \frac{\partial f}{\partial y}(x_0, y_0, z_0), \frac{\partial f}{\partial z}(x_0, y_0, z_0) \right).$$

This normal vector is engaged in calculation of intensities of the color components at the point (x_0, y_0, z_0).

2. Algebraic Surfaces

As it was mentioned in the introduction an algebraic surface is the surface given by an equation of the form

$$P(x, y, z) = 0. \tag{5}$$

where P is a polynomial with real coefficients. It can be written as

$$P(x, y, z) = \sum_{i=0}^{n} \sum_{j=0}^{m} \sum_{k=0}^{l} a_{ijk} x^i y^j z^k, \quad a_{ijk} \in \mathbb{R}, \tag{6}$$

where $n + m + l = d$ and $d \equiv \deg(P)$ is the degree of the polynomial P. Thus equation (5) has the following form

$$\sum_{i=0}^{n} \sum_{j=0}^{m} \sum_{k=0}^{l} a_{ijk} x^i y^j z^k = 0. \tag{7}$$

After substitution of the parametric representation (1) for the ray into (6) we obtain the polynomial

$$P^*(t) = \sum_{i=0}^{n} \sum_{j=0}^{m} \sum_{k=0}^{l} a_{ijk} (x_s + d_x t)^i (y_s + d_y t)^j (z_s + d_z t)^k \equiv \sum_{i=0}^{d} p_i t^i \tag{8}$$

of a single variable t with real coefficients p_i. Thus from (7) we get the equation

$$\sum_{i=0}^{d} p_i t^i = 0, \quad t \in [t_{in}, t_{out}], \tag{9}$$

where $[t_{in}, t_{out}]$ is the ray-extent intersection interval.

According to previous section the problem is now to find t_{min}, i.e. to find the smallest root of equation (9) on the specified interval. We solve this problem in two steps:

1. *Root Isolation.* This step allows us to find an interval (if it exists), where t_{min} is isolated, i.e. an interval, where equation (9) has the only root t_{min}.

2. *Root-Finding* is calculates a prescribed approximation of t_{min} in an isolating interval (provided this interval exists and we have found it).

2.1. Root Isolation. To isolate the root we involve *Descartes' Sign Rule* [2] – a method of determining the maximum number of positive real roots of a polynomial.

The idea of this method is as follows.

Let $a = (a_0, ..., a_n)$ be a sequence of real numbers and let $a' = (a'_0, ..., a'_m)$ be the subsequence of non-zero elements of a. Then the number $\mathrm{var}(a)$ of variations in a is the number of integers i such that $0 \le i < m$ and $a'_i a'_{i+1} < 0$.

Let $A(x) = \sum_{i=0}^{n} a_i x^i$ be a real polynomial. It is uniquely determined by a string a of its coefficients $a_0, ..., a_n$. We define $\mathrm{var}(A)$ as $\mathrm{var}(a)$.

Descartes' Sign Rule asserts that the number of positive real roots (taking into account their multiplicity) of a real polynomial A is equal to $\mathrm{var}(A)-2k$, where k is a non-negative integer. In other words, a number of allowable roots can be $\mathrm{var}(A)$, $\mathrm{var}(A)-2$, $\mathrm{var}(A)-4$ and so on. We emphasize that this rule is applicable on the positive semi-axis.

In the implementation of Descartes' Sign Rule in our problem we should first transform the polynomial P^* to another one, which defined on $\mathbb{R}_+$, but not on $[t_{in}, t_{out}]$. Below for the sake of simplicity we denote $a = t_{in}$, $b = t_{out}$.

We first consider the polynomial

$$A^{(1)}(x) = P^*(x(b-a)+a).$$

If $\alpha_1,...,\alpha_k$ are the roots of P^* such that $\alpha_i \in [a,b]$ for $1 \le i \le k$, then

$$\alpha_1' = \frac{\alpha_1 - a}{b-a}, ..., \alpha_k' = \frac{\alpha_k - a}{b-a}$$

are the roots of $A^{(1)}$ from the interval $[0,1]$. In order to avoid the consideration of the roots which are greater than one we further transform $A^{(1)}$ into the polynomial

$$A^{(2)}(x) = (x+1)^n A^{(1)}\left(\frac{1}{x+1}\right).$$

It is easy to see that $a+\beta_1(b-a),...,a+\beta_l(b-a)$ are the roots of P^* from the semi-interval $(a,b]$ if and only if $\beta_1^{-1}-1,...,\beta_l^{-1}-1$ are the positive real roots of $A^{(2)}$.

We consider the following cases.

1. **The case when** $\mathbf{var}A^{(2)} = 0$. In this case according to Descartes' Sign Rule there is no positive roots of $A^{(2)}$. Hence there are no roots of P^* on $[t_{in}, t_{out}]$. It means that the ray does not intersect the surface.

2. **The case when** $\mathbf{var}A^{(2)} = 1$. We obviously have a single positive root of $A^{(2)}$, which implies that P^* has only one root on $[t_{in}, t_{out}]$.

3. **The case when** $\mathbf{var}A^{(2)} \ge 2$. We cannot determine the exact number of roots, but we can bisect $[t_{in}, t_{out}]$ and consider two subintervals separately. Since we need the smallest root we can apply this technique to the same polynomial, but on the left subinterval first, and only if there are no roots we can switch to the right one.

Thus after a number of steps we can determine whether there exist roots of P^* in $[t_{in}, t_{out}]$ and if they exist, find an interval containing the smallest one.

2.2. Root-Finding. To find a prescribed approximation of t_{min} we use the *Dekker-Brent method* as it was described in [3, 10, 11]. The Dekker-Brent method combines the bisection and some more advanced root-finding algorithm (either quadratic interpolation or secant). On each step this method operates with three abscissas a, b and c, where

- b is the latest and the closest approximation of the root;
- a is the previous approximation;

- c is either previous or even an older approximation (it is possible that $a = c$).

Using the values of the polynomial in a, b and c and involving either the inverse quadratic interpolation method[1] (if $a \neq c$) or the secant method (when $a = c$), we obtain an approximation b^* of the root.

The key of the Dekker-Brent method is that we take b^* as the next approximation of the root *only* if the following criteria hold:

$$|b^* - b| < \frac{3}{4}|c - b| \quad \text{and} \quad |b^* - b| < \frac{1}{2}|b - a|.$$

Otherwise instead of b^* we take $\frac{1}{2}(b + a)$ following the bisection method. Then we take new $b' = b^*$, $a' = b$ and c' we keep the same if $P^*(c)P^*(b') \leq 0$ otherwise we take $c' = a'$. We stop our process when

$$\left|P^*(b)\right| \leq \varepsilon_m \quad \text{or} \quad \left|c - b\right| \leq 4\varepsilon \max\left\{\left|b\right|, 1\right\}$$

where ε_m is the machine precision and $\varepsilon \geq \varepsilon_m$ is the prescribed tolerance. Ideally, to find the best approximation for the root we should take $\varepsilon = \varepsilon_m$, but in our case we choose the tolerance ε empirically basing on the quality of image.

As it is mentioned in [3, 10, 11] numerous computer experiments have shown that Dekker-Brent method has a faster convergence in comparison with the customary methods. The point is that this method combines the sureness and the universality of the standard bisection method with relatively fast convergence of the secant or the quadratic interpolation method. Our own comparative computations (see Fig. 2) have shown that the Dekker-Brent method in most cases works faster then the Regula Falsi[2] (false position) and Bisection method. In our computations we have measured how much time[3] did it take the program to build the preview[4] image of the surface using three different methods. Analysis of the table on Fig. 2 shows that Dekker-Brent method allows us to save up to 1-2% of computation time in comparison with Regula Falsi method and up to 2-3% in comparison with Bisection method. Moreover, it is well-known (see, e.g. [10]) that Regula Falsi can sometimes be fooled. The following computational experiment was performed. A surface which is defined by 10^{th} degree polynomial (Barth Decic [14], see Fig. 13) was built using three different methods: Dekker-Brent, Bisection and Regula Falsi. The 1000 iteration threshold was set. If the number of iterations exceeds this threshold the execution of calculations is terminated unless required accuracy (10^{-6}) is reached. As a result of this experiment the image built using Regula Falsi method had defects in some regions (cf. Fig. 3 and Fig. 4). However, these defects were detected neither in other two images nor in the image from [14]. The point is that in the course of calculation polynomials of the form

---

[1] *Inverse interpolation is a convenient tool in root-finding procedures because it leads to the explicit formula for the root (to obtain the root all we need is to calculate the value of interpolation polynomial at zero).*

[2] *In Russian mathematical literature this method is known as the Chord method (see, e.g., [13]).*

[3] *One should bear in mind that the time strictly depends on the power of the CPU which has been used for the computations. In our case it was Intel Celeron CPU 1133 MHz.*

[4] *Preview image is an image visualized without illumination (cf. Fig. 6 and Fig. 7).*

represented on Fig. 5 arises. It is known [10] that for this kind of polynomials Regula Falsi requires too many iterations and it looks that in our case 1000 iterations is not enough for Regula Falsi.

Thus Dekker-Brent method is more appropriate for visualization of algebraic surfaces in comparison with Bisection and Regula Falsi methods.

Surface	Bisection Method	Chord Method (Regula Falsi)	Dekker-Brent Method
Barth Sextic (Fig. 9)	7m 44s (464s)	7m 37s (457s)	7m 37s (457s)
Cusp Catastrophe (Fig. 10)	1m 55s (115s)	1m 54s (114s)	1m 54s (114s)
Torus (Fig. 11)	4m 18s (258s)	4m 17s (260s)	4m 15s (255s)
Pillow/Tooth Object (Fig. 12)	1m 56s (116s)	1m 55s (115s)	1m 54s (114s)
Bernoulli Lemniscate[5] (see, e.g., Fig. 6 or 7)	4m 08s (248s)	4m 07s (247s)	4m 05s (245s)
Tanglecube (Fig. 14)	4m 03s (243s)	4m 01s (241s)	3m 59s (239s)

Figure 2. Comparative Computations. (Conditions: resolution is 1024×768 and antialiasing is on)

Figure 3. Dekker-Brent Method.

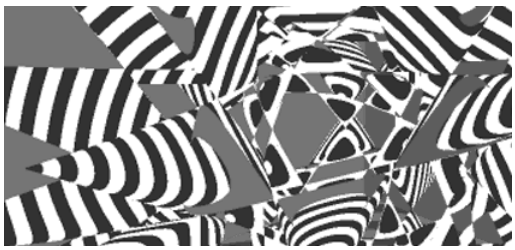

Figure 4. Chord (Regula-Falsi) Method.

3. Software

To test and put to practical use described above algorithms the following software was developed. The programming language used for the development of these programs was *C++* (see, e.g., [12]). These programs were built using *Microsoft Visual C++ .NET 2003*.

3.1. RAY. This program implements the algorithms described in Sections 1 and 2. To input the surface into to the program we need to represent it in form (6). In other words, input data looks like a sequence of vectors

$$\left\{\left(i, j, k, a_{ijk}\right)\right\}_{i=0,\, j=0, k=0}^{i+j+k\le d}, \tag{10}$$

[5] *The surface defined by the equation* $(x^2+y^2+z^2)^2 - 2a^2(x^2-y^2-z^2) = 0,\ a = 2.7.$

where each vector corresponds to the term $a_{ijk}x^i y^j z^k$ in the expression (6). It must be mentioned that additionally to the equation of the surface we input such data as extent parameters, material of the surface (i.e. its texture and optical properties), positions, constraints and intensities of the light sources, position of the viewer, position and constraints of the screen and so on.

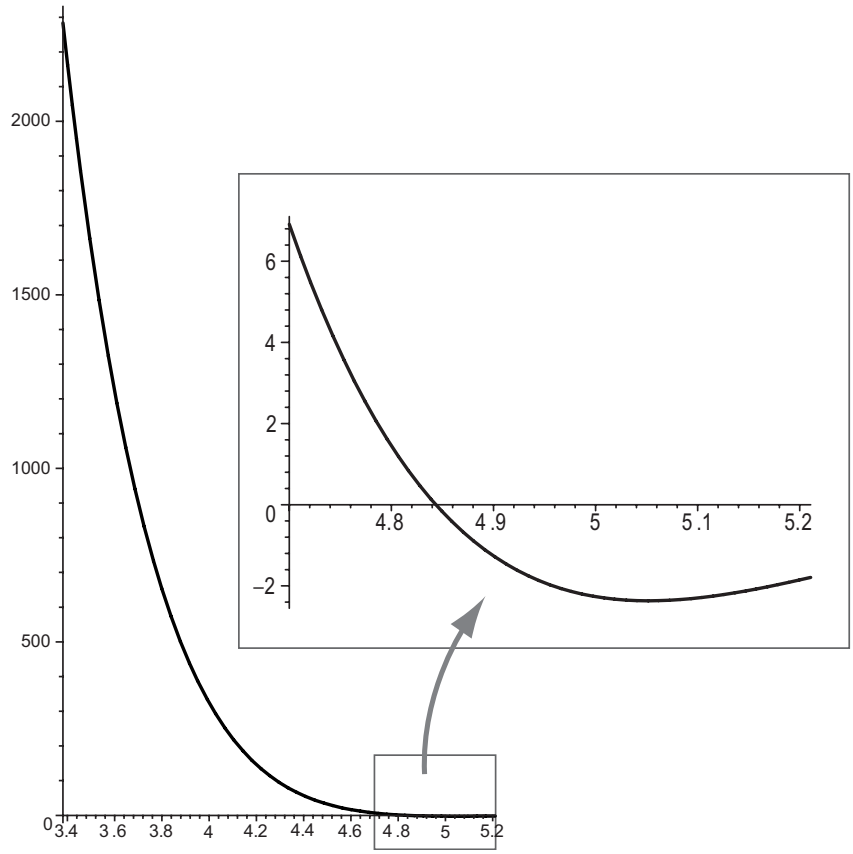

Figure 5. Polynomials arising in the course of the ray tracing of a Barth Decic [14].

According to Section 2 using (10) the algorithm constructs a polynomial $P^*(t)$ for each ray. Either to construct this polynomial or to carry out the transformations of the polynomial to isolate the root as it was described in Subsection 2.1 this program involves appropriate symbolic methods. All these symbolic methods are constructed from the following operations implemented in this program: the sum and the product of two univariate polynomials and the calculation of the coefficients of binomials of the form $B_n(t) = (at + b)^n$. Then using the ray tracing algorithm and the numerical methods described in Subsections 2.1 and 2.2 the program builds an image of the surface on the screen.

This image can be saved as *Windows Bitmap*, the standard Microsoft Windows format for raster images. User also can choose either to perform the computation of illumination or visualize the surface without performing these computations. To understand this feature compare Fig. 6, where the computation of illumination were

not performed (*preview mode*) with Fig. 7, where performed the full set of computations. It should be noted that computation of illumination takes a lot of time: approximately 100 times more for the image on Fig. 7, than for the image on Fig. 6. We have such difference because we need to compute the shadows and it is the most time-consuming part of the process of computation. The point is that to compute the shadows we need to shoot 50 secondary rays towards the light sources from each visible reference point on the scene. In our model we have two area light sources which are represented by uniformly distributed point light sources (25 points in each area light source). Moreover we need to compute the intensities of the color components of the reflected light for each visible reference point on the scene.

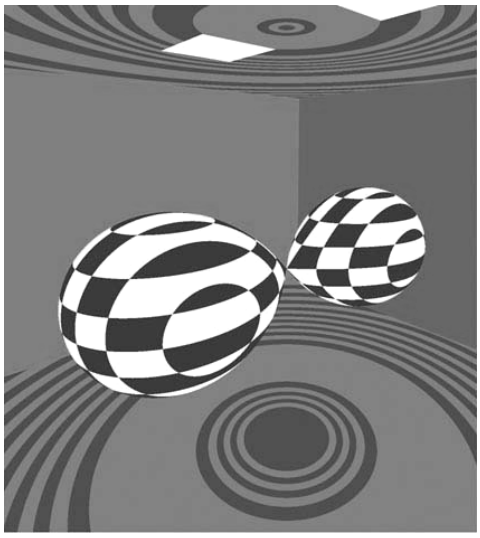

Figure 6. Preview Mode On.

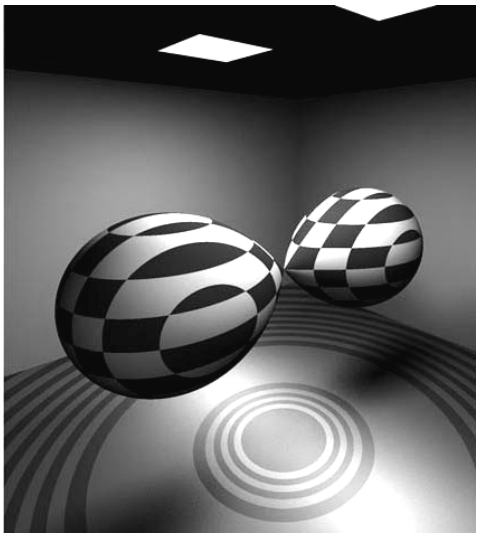

Figure 7. Preview Mode Off.

Thus the preview mode is very useful feature because if we are carrying out an intermediate visualization to adjust the parameters of the surface or an extent the preview image is informative enough to make a conclusion so there is no need to perform the computation of the illumination model.

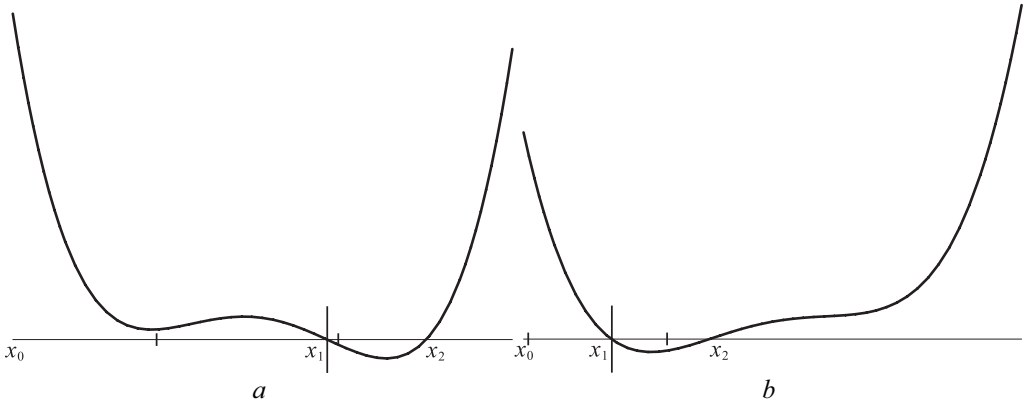

Figure 8. Polynomials arising in the course of the ray tracing of a torus.

3.2. ZEROFIND. This program is an additional tool developed for analyzing and testing the numerical methods we have used. Given an equation of the surface (sequence of vectors), a ray and an interval $[t_{in}, t_{out}]$, this program plots the graph of the $P^*(t)$ on $[t_{in}, t_{out}]$, then calculates an isolating interval for $t_{\min}$ using described above algorithm, marks the end points of this interval on the axis and, finally, marks an approximation of $t_{\min}$ obtained via Dekker-Brent root-finding algorithm.

This program is very useful because for each ray we can either to see how the plot of the $P^*(t)$ looks like or to trace the behavior of the numerical methods (root isolating and root-finding). We use this tool to check the correctness of our computations on intermediate steps. This checking is necessary because a "jump" of the smallest root to another is possible for some directions of rays. For instance, in the course of the ray tracing of a torus may appear the picture shown on Fig. 8.

Our trial computations show that if we apply Newton's method to the polynomial shown on Fig. 8(a), from the point x_0, then we can "catch" the root x_2 instead of the smallest root x_1. This is not the case for the polynomial on Fig. 8(b).

4. Images

Equations for visualization of the surfaces in Figures 9, 12, 13 and 14 are taken from [8], [9], [14] and [15] respectively. All computations were performed on Intel Celeron CPU 1133 MHz.

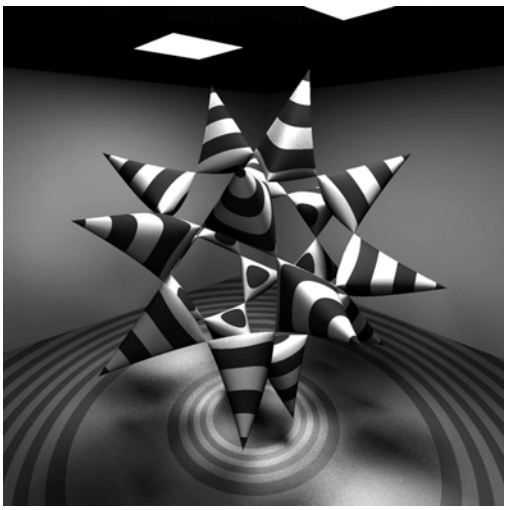

Figure 9. Barth Sextic [8]

$$4(G^2x^2 - y^2)(G^2y^2 - z^2)(G^2z^2 - x^2) - (1 + 2G)$$
$$(x^2 + y^2 + z^2 - 1)^2 = 0,\ G = \frac{\sqrt{5} - 1}{2} = 0.618\ldots$$

Computation time: 10h 42m 06s.

Figure 10. Cusp Catastrophe

$$z^3 + xz + y = 0.$$

Computation time: 3h 05m 40s.

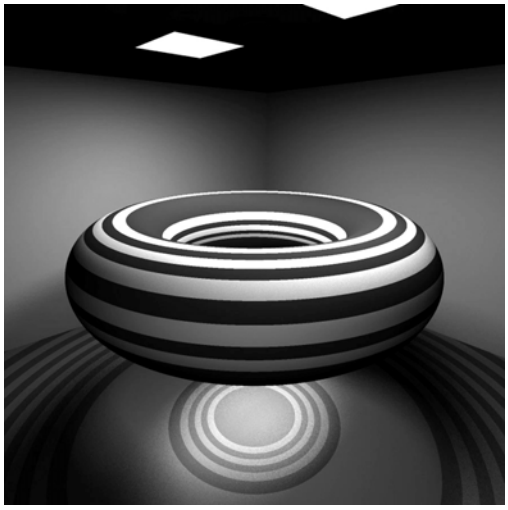

Figure 11. Torus

$$(x^2 + y^2 + z^2 - r^2 - a^2)^2 - 4a^2(r^2 - z^2) = 0,\ a = 2.5,\ r = 0.9.$$

Computation time: 5h 31m 13s.

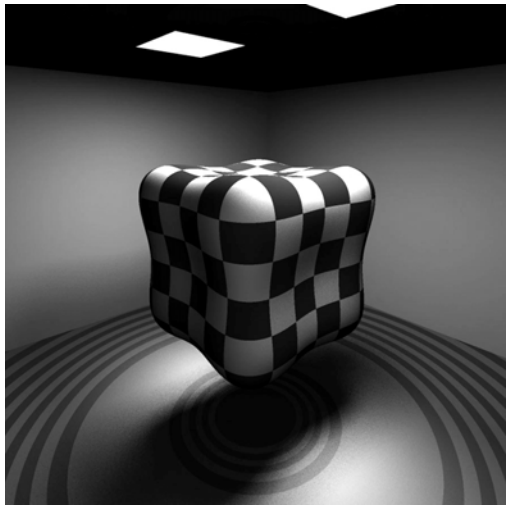

Figure 12. Pillow/Tooth Object [9]

$$x^4 + y^4 + z^4 - (x^2 + y^2 + z^2) = 0.$$

Computation time: 3h 05m 33s.

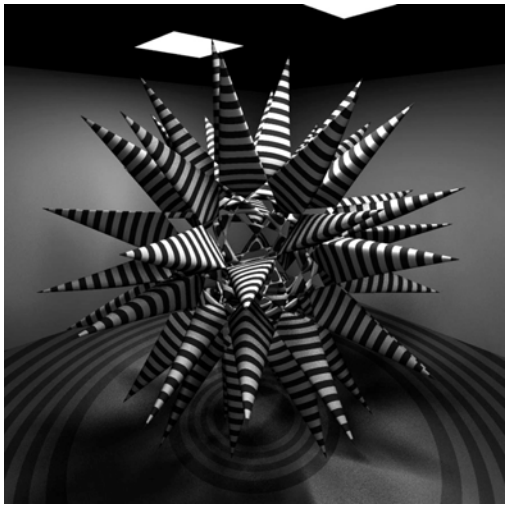

Figure 13. Barth Decic [14]

Computation time: 43h 09m 42s.

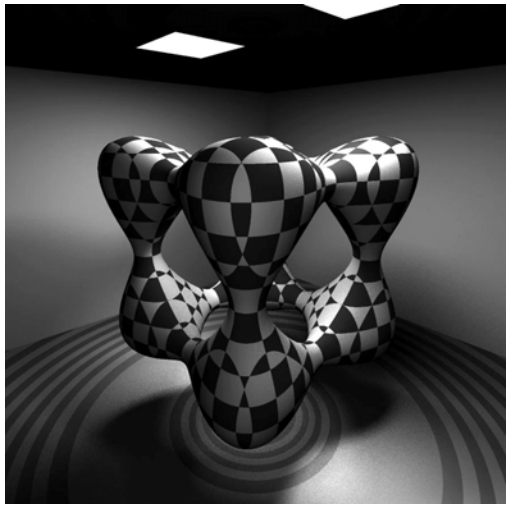

Figure 14. Tanglecube [15]

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

Computation time: 5h 39m 57s.

REFERENCES

1. Angel E., Interactive Computer Graphics. A Top-Down Approach with OpenGL, 2nd ed. Addison-Wesley, Reading, MA, 2000.
2. Collins G. E., Akritas A. G., Polynomial Real Root Isolation Using Descarte's Rule of Signs. Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computations, pp 272-275, 1976.
3. Forsythe G. E., Malcolm M. A., Moler C. B., Computer Methods for Mathematical Computations. Prentice-Hall, Englewood Cliffs, NJ, 1977.
4. Hanrahan P., Ray Tracing Algebraic Surfaces. Computer Graphics, 17(3), pp 83-90, 1983 (Proceedings of the SIGGRAPH 83).
5. Hill F. S., Computer Graphics Using OpenGL, 2nd ed. Prentice Hall PTR, Upper Saddle River, NJ, 2001.
6. Kajiya J. T., New Techniques for Ray Tracing Procedurally Defined Objects. ACM Transactions on Graphics, 2(3), pp 161-181, 1983.
7. Mike Henderson's Implicit Surface Algorithm. http://www.geom.uiuc.edu/docs/forum/henderson/
8. Nordstrand T., Barth Sextic. http://www.uib.no/People/nfytn/sexttxt.htm.
9. Nordstrand T., Surfaces. http://www.uib.no/People/nfytn/surfaces.htm.
10. 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.
11. Quarteroni A., Sacco R., Saleri F., Numerical Mathematics. Springer-Verlag, Berlin, Heidelberg, New York, 2000.
12. Stroustrup B., The C++ Programming Language, 3rd ed. Addison-Wesley, Reading, MA, 1997.
13. Turchak L. I., Osnovy chislennyh metodov (The Priciples of the Numerical Methods). Nauka, Moscow, 1987.
14. Weisstein E., "Barth Decic" from MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/BarthDecic.html.
15. Weisstein E., "Tanglecube" from MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/Tanglecube.html.