RENDERING (COMPLEX) ALGEBRAIC SURFACES[*]

J. F. Sanjuan-Estrada, L. G. Casado and I. García
Departament of Computer Archictecture and Electronic
University of Almeria
Ctra Sacramento s/n, 04120, Almeria (Spain)

Keywords: Geometric computing, Roots of complex polynomials, Interval arithmetic, Rendering complex space, Ray tracing.

Abstract: The traditional ray-tracing technique based on a ray-surface intersection is reduced to a surface-surface intersection problem. At the core of every ray-tracing program is the fundamental question of detecting the intersecting point(s) of a ray and a surface. Usually, these applications involve computation and manipulation of non-linear algebraic primitives, where these primitives are represented using real numbers and polynomial equations. But the fast algorithms used for real polynomial surfaces are not useful to render complex polynomials. In this paper, we propose to extend the traditional ray-tracing technique to detect the intersecting points of a ray and complex polynomials. Each polynomial equation with some complex coefficients are called complex polynomials. We use a root finder algorithm based on interval arithmetic which computes verified enclosures of the roots of a complex polynomial by enclosing the zeros in narrow bounds. We also propose a new procedure to render real or complex polynomials in the real and the complex space. If we want to render a surface in the complex space, the algorithm must detect all real and complex roots. The color of a pixel will be calculated with those roots with an argument inside a selected complex space and minimum magnitude of the complex roots.

1 INTRODUCTION

The ray tracing technique has captured its place as an extremely valuable tool in generating photorealistic synthetic imagery. Ray tracing has been intensively and successfully employed to emulate specular reflections and/or refractions, as well as to delineate illuminated regions from those in shadow.

In ray tracing, a ray is defined as the set of points:

$$R = \left\{ \begin{array}{c} x = a_x + t \cdot V_x; \\ y = a_y + t \cdot V_y; \\ z = a_z + t \cdot V_z; \end{array} \right\} \quad (1)$$

where (a_x, a_y, a_z) is the viewpoint of the scene, and $(V_x,\ V_y,\ V_z)$ is the unit vector defining the direction of the ray. Intersections between R and a polynomial surface defined by $f(x, y, z)$, is given by the set of points in $f(x, y, z) \cap R$, such that:

$$f(x, y, z) = f(a_x+tV_x, a_y+tV_y, a_z+tV_z) = 0 \quad (2)$$

If $f(s(t)) \neq 0;\ \forall t \geq 0$ then $f(x, y, z) \cap R = \emptyset$ and there is no intersection between the surface and the ray. In the case where there exist one or more values of t $(t_0, t_1, \ldots; t_i \geq 0)$ for which $f(s(t_i)) = 0$, then the point to visualize on the screen corresponds to the smallest value of t_i.

Polynomials form a fundamental class of mathematical objects with diverse scientific applications. According to the Fundamental Theorem of Algebra, a polynomial of degree n, with real or complex coefficients, has n zeros (roots) which may or may not be distinct. The task of computing accurate zeros of polynomials has been one of the most influential problem in the development of several important areas in mathematics.

Computer graphics can help to visualize complex polynomials. An good example is the visualization software developed by Bahman Kalantari, called polynomiography. For a given polynomial equation, the computer applies a specific root-finding method to a large number of zeros (z). For each initial value, the computer determines toward which root that value converges and assigns a colour to the point, a differ-

[*]This work has been partially supported by the Ministry of Education and Science of Spain through grants TIC2002-00228 and TIN2005-00447

ent colour for each root. Shades of colour indicate how fast that point comes close to the root (Kalantari, 2004).

In this work, we propose to extend the original ray tracing technique to complex space. Unfortunately, contemporary methods for ray-tracing polynomial surfaces work with real number. So it is necessary to design and implement a root finder algorithm that works with complex number.

This paper is organized as follows. Section 2 presents the algorithm to find roots of complex polynomial. This algorithm uses an iterative scheme that starts with an initial approximation of all the roots, refines them and updates the error bound. Section 3 shows some improvements to exploit the spatial coherence and so to speed up the root finder algorithm. Section 4 describes how to include these improvements in ray tracing technique to render polynomial surfaces in the complex space. Finally, several examples of real and complex polynomials rendered in complex space are shown in Section 5.

2 ZEROS OF COMPLEX POLYNOMIALS

In this paper, we apply the algorithm proposed in (Hammer et al., 1995) to find a single root of the intersection between a complex polynomical surface and a ray. This algorithm is based on the fact that the roots of the complex polynomial of degree n match the eigenvalues of the companion matrix:

$$A = \begin{pmatrix} 0 & \cdots & 0 & \frac{-p_0}{p_n} \\ 1 & & & \frac{-p_1}{p_n} \\ & \ddots & & \vdots \\ & & 1 & \frac{-p_{n-1}}{p_n} \end{pmatrix} \quad (3)$$

Hammer showed a formulation of the Newton iteration to solve the eigenvalue problem with

$$g\begin{pmatrix} \Delta_q \\ \Delta_z \end{pmatrix} = -R \cdot d + R \cdot \Delta_z \cdot \begin{pmatrix} \Delta_q \\ 0 \end{pmatrix} + (I - R \cdot J_f) \cdot \begin{pmatrix} \Delta_q \\ \Delta_z \end{pmatrix} \quad (4)$$

where matrix R is the inverse of the Jacobian matrix (J_f), Δ_q is the residual value of coefficients of complex polynomial, Δ_z is the residual value of the eigenvalues, and

$$d = (A - \tilde{z} \cdot I) \cdot \begin{pmatrix} \tilde{q} \\ p_n \end{pmatrix} \quad (5)$$

where I is the identity matrix of dimension n, $\tilde{q}$ is an approximation of the coefficients of the deflated polynomial and $\tilde{z}$ is an approximation of the exact root of the complex polynomial $p(z)$.

$$p(z) = \sum_{i=0}^{n} p_i \cdot z^i, \quad p_i \in C \quad (6)$$

This algorithm is structured in two parts: the *Approximation* and the *IntervalIteration*.

2.1 Approximation Algorithm

First, the *Approximation* algorithm improves an initial approximation $\tilde{z}$ of a root of a complex polynomial $p(z) = \sum_{i=0}^{n} p_i \cdot z^i$. *Approximation* works by transforming the original problem to the equivalent problem of finding an eigenvalue and its corresponding eigenvector for the companion matrix A of Equation (3).

It improves the approximations of a root and the coefficients of the corresponding deflacted polynomial to avoid overestimation during the floating-point interval calculations. *Approximation* uses a residual iteration method to improve the initial approximation of a root until the accuracy requirement of *IntervalIteration* is achieved. No guarantee for the correctness of the approximation computed by *Approximation* is claimed.

In the following algorithm, the iteration $\Delta^{(k+1)} = g(\Delta^{(k)})$ is done directly using some loops that are equivalent to Equation (4).

```
Approximation(p, z̃)
1. Termination criteria: $\epsilon = 10^{-10}; k_{max} = 50$
2. Determination of an approximate eigenvector $\tilde{q}$ for
   the initial polynomial zero $\tilde{z}$
   $\tilde{q}_{n-1} = p_n$
   $\tilde{q}_i = \tilde{q}_{i+1} \cdot z + p_{i+1}$; for $i = n-2$ to 0
3. Floating-point iteration of an eigenvalue $\tilde{z}$ and an
   eigenvector $\tilde{q}$
   $k = 0$
   (a) repeat
       i. Compute the deflect $d = (A - \tilde{z} \cdot I) \cdot \tilde{q}$
          $d_0 = (-\tilde{z} \cdot \tilde{q}_0 - p_0)$
          $d_i = (\tilde{q}_{i-1} - \tilde{z} \cdot \tilde{q}_i - p_i)$; for $i = 1$ to $n-1$
       ii. Compute the components of R, the inverse of
           the Jacobian $J_f$
           $w_{n-1} = \tilde{q}_{n-1}$
           $w_i = (\tilde{q}_i + \tilde{z} \cdot w_{i+1})$; for $i = n-2$ to 0
           if $w_0 = 0$ then Err="Inversion failed"
       iii. Compute $\Delta^{(k+1)} = g(\Delta^{(k)})$
            $\Delta^{k+1}_{n-1} = d_{n-1}$
            $\Delta^{(k+1)}_i = d_i + \tilde{z} \cdot \Delta^{(k+1)}_{i+1}$; for $i = n-2$ to 0
            $t = \Delta^{(k+1)}_0 / w_0$
            $\Delta^{(k+1)}_i = -\Delta^{(k+1)}_{i+1} + t \cdot w_{i+1}$; for $i = 0$ to
            $n-2$
            $\Delta_{n-1}^{(k+1)} = t$
      iv. Update $\tilde{q}$ and $\tilde{z}$
            $\tilde{q}_i = \tilde{q}_i + \Delta_i^{(k+1)}$; for $i = 0$ to $n-2$
            $\tilde{z} = \tilde{z} + \Delta_{n-1}^{(k+1)}$
      v. $k = k+1$
   (b) until $\left(\frac{||\Delta||_\infty}{\max(||((\tilde{q})_{i=0}^{n-2}||_\infty, |\tilde{z}|)}\right) \leq \epsilon$ or $(k = k_{max})$
4. Return $\tilde{q}$, $\tilde{z}$, Err
```

We use $k_{max} = 50$ as the maximum number of iterations, and $\epsilon = 10^{-10}$ as the value for the relative error. If the condition number of the inverse R is extremely large, then the convergence of the residual iteration is slow. To avoid the possibility of an unbounded number of iterations at step 3, we halt after k_{max} iterations. It turned out that k_{max} and ϵ are good values for minimizing the effort to get sufficiently accurate approximations of $\tilde{z}$ and $\tilde{q}$.

2.2 Interval Iteration Algorithm

The algorithm *IntervalIteration* computes a verified enclosure of a root of a complex polynomial using an interval iteration. Starting with good approximations $\tilde{z}$ for a root of the complex polynomial $p(z)$ and $\tilde{q}$ for the coefficients of the deflated polynomial $q(z) = \frac{p(z)}{z-\tilde{z}}$, a verification strategy based on Schauder's fixed-point theorem is used to determine (if possible) a guaranteed enclosure of a polynomial root z^* (Granas and Dugundji, 2004) and (Jimenez-Melado and Morales., 2005). In addition, guaranteed enclosures of the coefficients of the deflated polynomial $q^*(z) = \frac{p(z)}{z-z^*}$ are returned.

IntervalIteration(p, $\tilde{z}$)

```
1. Computation of the interval evaluation
   (a) Compute an enclosure of $d$ (Eq. (5))
         $[d]_0 = (-\tilde{z} \cdot \tilde{q}_0 - p_0)$
         $[d]_i = (\tilde{q}_{i-1} - \tilde{z} \cdot \tilde{q}_i - p_i)$; for $i = 1$ to $n-1$
   (b) Compute an enclosure of $[R]$, the inverse of the
       Jacobian $J_f$
         $[w]_{n-1} = \tilde{q}_{n-1}$
         $[w]_i = (\tilde{q}_i + \tilde{z} \cdot [w]_{i+1})$; for $i = n-2$ to 0
         if $0 \in [w]_0$
         then Err="Verified enclosure failed"
   (c) Compute enclosures of $\Delta_q^{(0)}$ and $\Delta_z^{(0)}$
         $[\Delta]_{n-1}^{(0)} = d_{n-1}$
         $[\Delta]_i^{(0)} = [d]_i + \tilde{z} \cdot [\Delta]_{i+1}^{(0)}$; for $i = n-2$ to 0
         $[t] = [\Delta]_0^{(0)} / [w]_0$
         $[\Delta]_i^{(0)} = -[\Delta]_{i+1}^{(0)} + [t] \cdot [w]_{i+1}$; for $i = 0$ to
         $n-2$ $[\Delta]_{n-1}^{(0)} = [t]$
2. Interval iteration
   $k = 0$; $\epsilon = 10^{-1}$; $k_{max} = 10$
   (a) Repeat
      i. Slightly enlarge the enclosure interval
            $[\Delta]^{(k)} = [\Delta]^{(k)} \bowtie \epsilon$
      ii. Determine a new enclosure interval of (4)
            $[v]_{n-1} = 0$
            $[v]_i = ([\Delta]_{n-1}^{(k)} \cdot [\Delta]_i^{(k)} + \tilde{z} \cdot [v]_{i+1})$; for $i = n-2$
            to 0
            $[v]_0 = \frac{[v]_0}{[w]_0}$
            $[\Delta]_i^{(k+1)} = ([\Delta]_i^{(0)} + [v]_{i+1} - [v]_0 \cdot [w]_{i+1})$ for
            $i = 0$ to $n-2$
            $[\Delta]_{n-1}^{(k+1)} = [\Delta]_{n-1}^{(0)} - [v]_0$
      iii. $k = k+1$
   (b) until $([\Delta]^{(k)} \subset [\Delta]^{(k-1)})$ or $(k = k_{max})$
3. Verification of the result
   if $([\Delta]^{(k)} \subset [\Delta]^{(k-1)})$ then
      $[q]_i = (\tilde{q}_i + [\Delta]_i^{(k)})$; for $i = 0$ to $n-2$
      $[q]_{n-1} = p_n$
      $[z] = (\tilde{z} + [\Delta]_{n-1}^{(k)})$
   else Err="Inclusion failed"
4. Return $[q]$, $[z]$, Err
```

We use $k_{max} = 10$ as the maximum number of iterations, and $\epsilon = 0.1$ as the value for the epsilon inflation. It turned out that these are good values for minimizing the effort if no verification is possible.

2.3 CAllPolyZero Algorithm

Hammer et al. designed an algorithm for finding a single complex root (*CPolyZero*) which uses algorithms *Approximation* and *IntervalIteration* (Hammer et al., 1995). Because of our interest in finding all the complex and real roots of polynomial, we have designed a new algorithm called *CAllPolyZero*. By repeating the deflation of a verified zero from the reduced polynomial $pdeflated[q]$, the approximation of a new zero in the reduced polynomial and the verification of the new zero in the original polynomial, we get all simple zeros of the polynomial. The new deflated polynomial is computed by the algorithm *IntervalIteration* based on the current value of $[q]$. For approximating a new zero, the deflated polynomial $pdeflated$ is used. The verification of the new zero is done in the original polynomial p because zeros of the approximate deflated polynomial are smeared out because $[q]$ has interval-valued coefficients, while p has point-valued coefficients.

AllCPolyZeros(p, $\tilde{z}$)

```
1. $pdeflated = p$
2. for $i = 1$ to $n-1$ do verification of a new zero
   (a) Approximate a new zero of pdeflated:
       Approximate($pdeflated, \tilde{z}$)
```

(b) Verify the new zero for p

(c) Deflate verified zero from $pdeflated$

This algorithm finds all roots of complex and real polynomials. If *AllCPolyZeros* algorithm works with a real polynomial (imaginary part of each coefficient is zero) and a real initial approximation of $\tilde{z} \in R$, it will never get enclosures of a pair of conjugate complex zeros of $p(z)$, because all complex arithmetic operations deliver a real result. However, for finding a complex conjugate zero of a real polynomial, *AllCPolyZeros* algorithm must start with a non-real initial approximation $\tilde{z}$.

3 SPATIAL COHERENCE

In this section, some improvements to speed up of *CAllPolyZero* algorithm are shown. This algorithm needs an initial approximation $\tilde{z}$ to start the process. If this parameter is near to the exact root then it is necessary a few number of iterations. This means that is necessary to supply a good initial approximation $\tilde{z}$ to reduce the algorithm execution time.

Numerous optimization methods for ray tracing have been suggested since it was first introduced (Whitted, 1980). Many have suggested the exploitation of spatial coherence (Glassner, 1989). Once a single ray has been processed, a ray emitted for a nearby pixel at a similar direction will hit, most likely, a nearby target. We propose to exploit spatial coherence, so that if a primary ray hit a surface in pixel (j, k), it is very probable that the primary rays of neighboring pixels hit the same object. These neighboring pixels are $(j-1, k-1)$, $(j, k-1)$, $(j+1, k-1)$, $(j-1, k)$, $(j+1, k)$, $(j-1, k+1)$, $(j, k+1)$ and $(j+1, k+1)$.

In order to find the initial approximation of the root associated to pixel (j, k), we propose to choose the average value of the roots in the neighboring pixels. Following, we show how to compute this average initial approximation $\tilde{z}$ for pixel (j, k):

1. *CAllPolyZero* only can use the roots associated to pixels: $(j-1, k-1)$, $(j, k-1)$, $(j+1, k-1)$ and $(j-1, k)$. Notice that the roots associated to the remaining neighboring pixels have not been computed yet.
2. If a primary ray does not hit an object in a neighboring pixel, then this pixel is not used in this average.
3. If every primary rays of neighboring pixels does not hit an object then the initial approximation $\tilde{z}$ is $0 + i$.
4. If several primary rays of neighboring pixels hit some objects:
   (a) If the primary rays hit the same object, we calculate only a single average value of the initial approximation.
   (b) If the primary rays hit several objects, several average values of the initial approximations are computed, one for each object.

4 RAY TRACING IN COMPLEX SPACE

The previous sections describes our general algorithm for computing all roots of a polynomial, and computations are done in the complex space. In this section, we will briefly describe the technique used to compute the colour of each pixel in an image rendered by ray tracing techniques. The traditional ray tracing uses the minimum positive root to assign the colour of a pixel in real space. A complex number $z = x + i \cdot y$ can be represented in complex space, like $\rho \cdot e^{i \cdot \theta}$, the magnitude represents its modulus ρ and the angle θ its complex argument (see Figure 1).

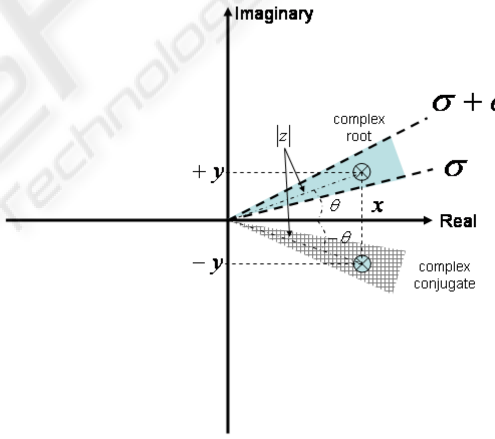

Figure 1: Sampling complex space.

In our algorithm the selected root is that with the minimum magnitude and with its complex argument θ in a selected range given by $\sigma \leq \theta \leq \sigma + \delta$. The selected root will determine the final colour of a pixel. This means that the rendering process is guided not only by the magnitude of the roots but also it can play with their complex arguments. This algorithm will allow to sample the complex space, so that different images can be obtained by choosing the interval angle $[\sigma, \sigma + \delta]$ (see Figure 1). For example, for rendering a scene in the real space, $\sigma = 0$ and $\delta = 10^{-10}$ are appropriate values. However, for $\sigma = 0.1$ and $\delta = \frac{\pi}{4}$, the selected roots belong to the complex space with angles between $0.1 \leq \theta \leq 0.1 + \frac{\pi}{4}$ and their complex conjugate $-0.1 - \frac{\pi}{4} \leq \theta \leq -0.1$. In this case, the real roots are not included in the search space.

This procedure allows us to render three-dimensional complex algebraic surfaces in the complex space with an angle bounded. For rendering all complex space using ray tracing, we can sample all space with different values of δ and σ. Due to the symmetry of the conjugate complex roots, it is only necessary to sample the complex space determined by $\sigma \geq 0$ and $\sigma + \delta \leq \pi$.

When we render a scene defined by complex algebraic surfaces, we can use a maximum δ value equal to π (and $\sigma = 0$). The result is that we obtain a large amount of roots associated to the same pixel because we are dealing with the full complex space. Our proposal consists of sampling the complex space with a narrow aperture angle; i.e. small values of δ. This method allows us to generate an animated sequence of images, each corresponding to a different value of δ. The animated sequence of images gives an interesting information about the distribution of roots in the complex space.

5 EXPERIMENTATION

We use the C-XSC library, a C++ class library for eXtended Scientific Computing, to implement the proposed algorithm. Its wide range of numerical data types, operators and functions for scientific computation makes C-XSC especially well suited as a specification language for programming with automatic result verification.

The automatic verification of the numerical results is based on interval arithmetic. The easiest technique for computing verified numerical results is to replace any real or complex operation by its interval equivalent and then to perform the computations using interval arithmetic. This procedure leads to reliable and verified results. However, the diameter of the computed enclosures can be as wide as to be practically useless. We have applied the principle of iterative refinement (*IntervalIteration*) to our algorithm, which will allow us to compute an interval error less than a desired accuracy. The verified enclosure of the solution is given by the approximation of the root and the enclosure of its error.

In order to observe the performance of our algorithm, we have used a set of polynomials with more than twenty different polynomials. However, in this paper, we only show six interesting polynomials due to space limitations (see Table 1).

The *CAllPolyZero* algorithm needs the coefficients of polynomial obtained by Equation (2) to transform $f(x,y,z)$ in an univariate polynomial $f(t)$. So that, the sphere polynomial is $p_2 \cdot t^2 + p_1 \cdot t + p_0$, while the Whiney polynomial is $p_3 \cdot t^3 + p_2 \cdot t^2 + p_1 \cdot t + p_0$ and $p_4 \cdot t^4 + p_3 \cdot t^3 + p_2 \cdot t^2 + p_1 \cdot t + p_0$ for Bicube, Steiner, Chair and Tangle polynomials. Finally, the degree of Boy polynomial is six, like $p_6 \cdot t^6 + p_5 \cdot t^5 + p_4 \cdot t^4 + p_3 \cdot t^3 + p_2 \cdot t^2 + p_1 \cdot t + p_0$.

Table 1: Polinomial surfaces.

Surface	Polynomial equation $f(x,y,z)$
Sphere	$x^2 + y^2 + z^2 - 1$
Whitney	$x^2 \cdot z + y^2$
Steiner	$x^2 \cdot y^2 + x^2 \cdot z^2 + x \cdot y \cdot z + y^2 \cdot z^2$
Chair	$x^4 - 1.2 \cdot x^2 \cdot y^2 + 3.6 \cdot x^2 \cdot z^2 +$ $16 \cdot x^2 \cdot z - 7.5 \cdot x^2 + y^4 +$ $3.6 \cdot y^2 \cdot z^2 - 16 \cdot y^2 \cdot z -$ $7.5 \cdot y^2 - 0.2 \cdot z^4 - 7.5 \cdot z^2 + 64.0625$
Tangle	$x^4 - 5 \cdot x^2 + y^4 - 5 \cdot y^2 + z^4 - 5 \cdot z^2 + 11.8$
Boy	$-729 \cdot x^6 + 1374.6156 \cdot x^5 \cdot z -$ $2187 \cdot x^4 \cdot y^2 - 810 \cdot x^4 \cdot z^2 + 324 \cdot x^4 \cdot z -$ $2749.23 \cdot x^3 \cdot y^2 \cdot z - 305.47 \cdot x^3 \cdot z^3 -$ $2187 \cdot x^2 \cdot y^4 - 1620 \cdot x^2 \cdot y^2 \cdot z^2 +$ $648 \cdot x^2 \cdot y^2 \cdot z + 1832.82 \cdot x^2 \cdot y \cdot z^3 -$ $1832.82 \cdot x^2 \cdot y \cdot z^2 + 324 \cdot x^2 \cdot z^4 -$ $144 \cdot x^2 \cdot z^2 - 4123.85 \cdot x \cdot y^4 \cdot z +$ $916.41 \cdot x \cdot y^2 \cdot z^3 - 729 \cdot y^6 -$ $810 \cdot y^4 \cdot z^2 + 324 \cdot y^4 \cdot z - 610.94 \cdot y^3 \cdot z^3 +$ $610.94 \cdot y^3 \cdot z^2 + 324 \cdot y^2 \cdot z^4 -$ $144 \cdot y^2 \cdot z^2 - 216 \cdot y + 432 \cdot z^6 - 288 \cdot z^5 +$ $64 \cdot z^4$

Figures 3 and 4 show a sphere where only p_0 is a complex coefficient. Both images are different because the first one was rendered for $\sigma = 0$ and $\delta = \frac{\pi}{18}$, while $\delta = \frac{7 \cdot \pi}{18}$ in the second one. It is important to see how the complex roots transform a real sphere (see Figure 2) in a sphere with special effects. These effects can modify the texture, size and illumination of the surface.

The Tangle polynomial was rendered in real (see

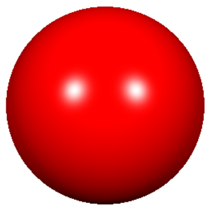

Figure 2: Sphere polynomial where all coefficients are real numbers. This image was rendered in real space for $\sigma = 0$ and $\delta = 0$.

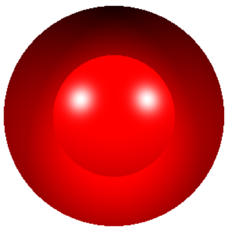

Figure 3: Sphere polynomial where p_0 is a complex coefficient. This image was rendered in complex space for $\sigma = 0$ and $\delta = \frac{\pi}{18}$.

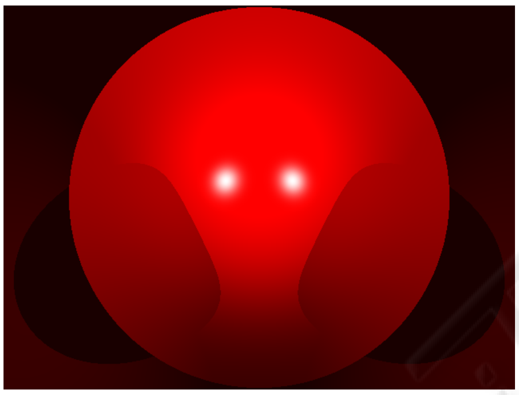

Figure 4: Sphere polynomial where p_0 is a complex coefficient. This image was rendered in complex space for $\sigma = 0$ and $\delta = \frac{7 \cdot \pi}{18}$.

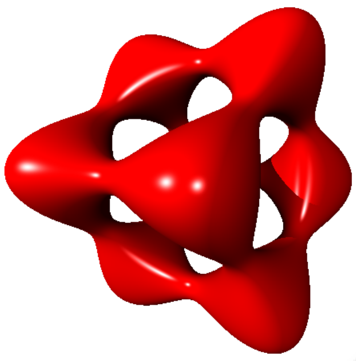

Figure 5: Tangle polynomial where all coefficients are real numbers. This image was rendered in real space for $\sigma = 0$ and $\delta = 0$.

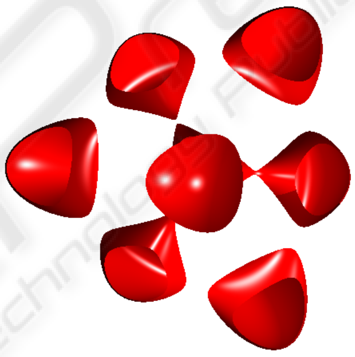

Figure 6: Tangle polynomial where p_0 is a complex coefficient. This image was rendered in complex space for $\sigma = 0$ and $\delta = \frac{\pi}{180}$.

Figure 5) and complex space. The images rendered in complex space were represented by p_0 (see Figure 6) and p_2 (see Figure 7) as complex coefficient. All these images show big differences. On one hand, same pieces of real surface disappear in the complex space, as shown in Figure 6. On the other hand, it is very interesting to highlight the shadows appearing on the Tangle surface (see Figure 7).

Figure 9 shows the Whitney polynomial rendered in the complex space for $\sigma = 0$ and $\delta = \frac{\pi}{18}$. When p_0 is a complex coefficient then some shadows appear on the image. These shadows are very different from Whitney polynomial rendered in real space (see Figure 8). However, a new piece of surface around imaginary axis of Whitney surface appears when we sample the complex space between 0 and $\frac{\pi}{18}$.

The following sequence of images for Steiner surface is very interesting. The complex roots of this surface do not cover the object, but they are located around three ”imaginary” axes of surfaces. For example, the Steiner surface shows the three axes which appear in the complex space (see Figure 10). It is important to show that this sequence is obtained sampling the complex space, so that this allow us to analyse the evolution of the complex roots for the Steiner surface.

Finally, we have rendered a sequence of images for the Boy surface (see Figure 11). This sequence begins with a representation of the Boy surface with all coefficients in real space and rendered in the real space. Later the Boy surface with p_0, p_1, p_2, p_3 and p_4 as complex coefficients were rendered in complex space. This example shows how the surface is disappearing around the real axis of complex space when the coefficient higher is a complex number. In this case, the Boy surface completely disappear around real axis when p_5 or p_6 are complex coefficients. That means that complex roots for these coefficients are far from real axis.

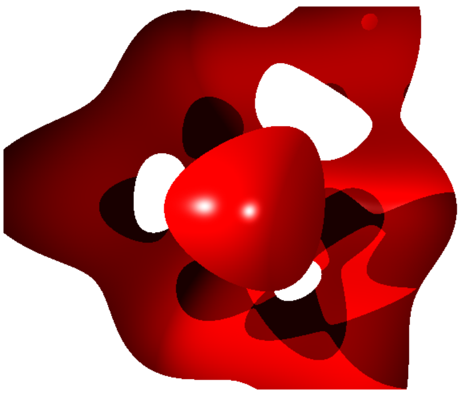

Figure 7: Tangle polynomial where p_2 is a complex coefficient. This image was rendered in complex space for $\sigma = 0$ and $\delta = \frac{\pi}{18}$.

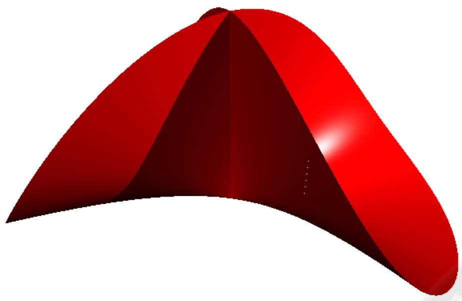

Figure 8: Whitney polynomial where all coefficients are real numbers. This image was rendered in real space for $\sigma = 0$ and $\delta = 0$.

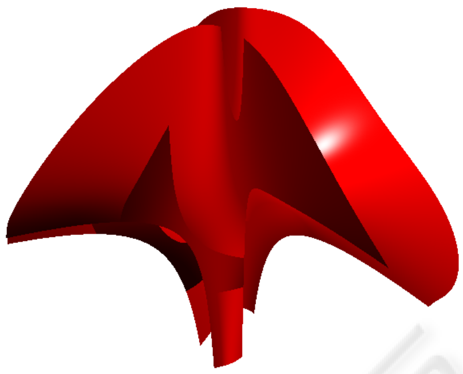

Figure 9: Whitney polynomial where p_0 is a complex coefficient. This image was rendered in complex space for $\sigma = 0$ and $\delta = \frac{\pi}{18}$.

6 CONCLUSION

We have designed and implemented a complex root finder algorithm to render complex polynomial surfaces in complex space. For this problem, it is not possible to use the Sturm sequences of a complex polynomial as a root finder algorithm. This is due to some of the polynomial coefficients are complex and additionally we are interested in the complex roots of the intersection. We solve this problem as an eigenvalue problem, where we have used the polynomial root finding algorithm proposed by Hammer with some additional extras. On the one hand, we have solved the problem of distinguishing zeros which are very close one to each other with high accuracy. On the other hand, we have extended this algorithm to find all the complex roots associated to each pixel of the rendered image.

Finally, we have proposed a new procedure to render images with a ray tracing technique in the complex space. This technique allows us to build a sequences of images from which we can analyse the evolution of the complex roots of several complex and real polynomial surfaces in a three-dimensional space. The typical effects of rendering real surfaces such as reflection, refraction or translucent can also be applied to rendering complex algebraic surfaces.

REFERENCES

Glassner, A. (1989). *An Introduction to Ray Tracing*. Academic Press, Boston.

Granas, A. and Dugundji, J. (2004). Fixed point theory. *Bulletin of the American Mathematical Society*, 41(2):267–271.

Hammer, R., Hocks, M., Kulisch, U., and Ratz, D. (1995). *C++ Toolbox for Verified Computing I: Basic Numerical Problems: Theory, Algorithms, and Programs*. Springer-Verlag, Berlin.

Jimenez-Melado, A. and Morales., C. (2005). Fixed point theorems under the interior condition. *Procceding of the American Mathematical Society*, 134(2):501–507.

Kalantari, B. (2004). Polynomiography and applications in art, education, and science. *Computers & Graphics*, 28(3):417–430.

Whitted, T. (1980). An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349.

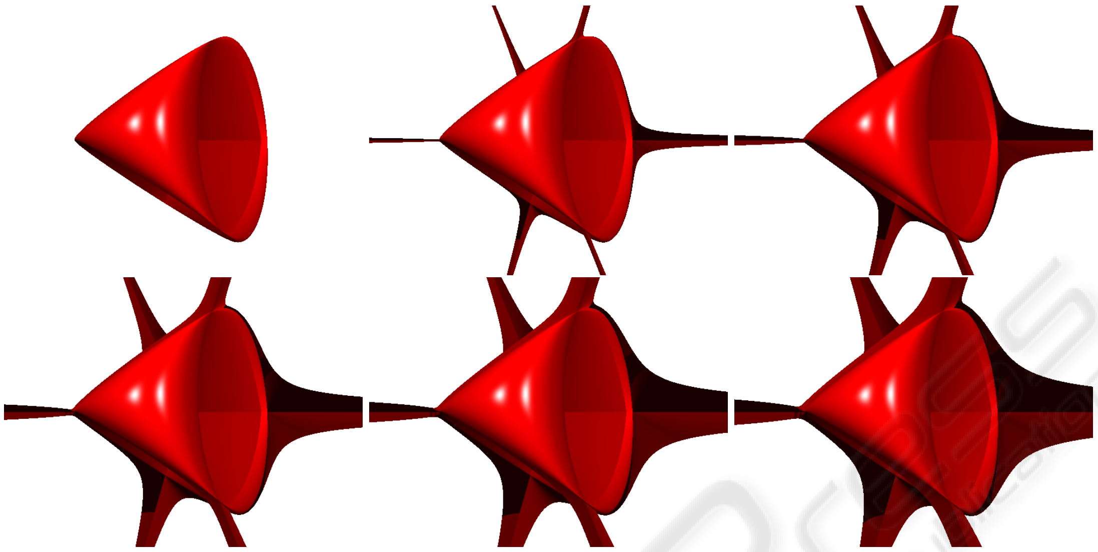

Figure 10: From up-left to down-right, Steiner surface with all real coefficients are shown, obtained using the following δ values: 0, $\frac{\pi}{180}$, $\frac{\pi}{90}$, $\frac{\pi}{60}$, $\frac{\pi}{45}$ and $\frac{\pi}{36}$. All Steiner surfaces are rendering for $\sigma = 0$.

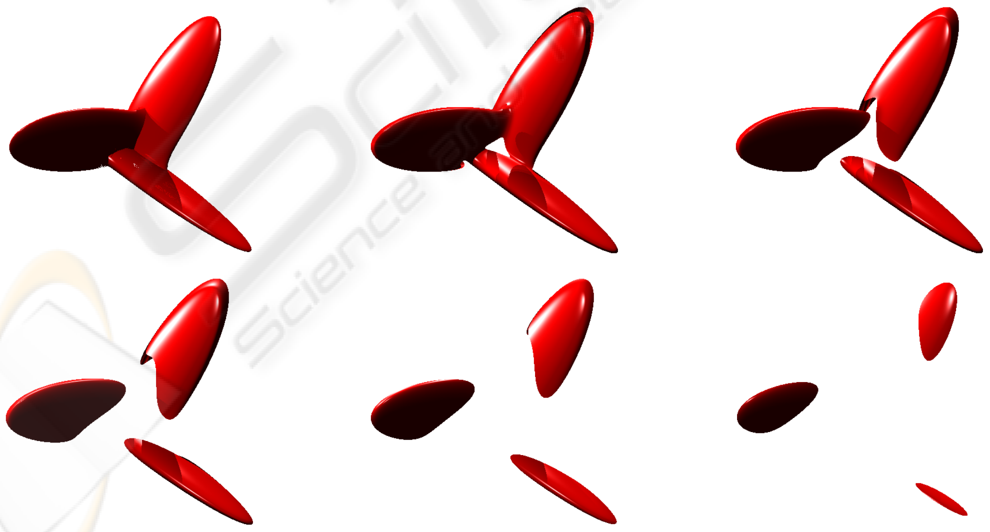

Figure 11: From up-left to down-right, Boy surface are shown: A Boy surface with all real coefficients was rendered in real space for $\sigma = 0$ and $\delta = 0$ in first image. The others images show a boy surface with a complex coefficient in complex space for $\sigma = 0$ and $\delta = \frac{\pi}{18}$. The evolution of these complex coefficients was p_0 (second image), p_1 (third image), p_2 (fourth image), p_3 (fifth image) and p_4 (sixth image).