Point-based rendering of implicit surfaces in $\mathbb{R}^4$

Alex Laier Bordignon*, Luana Sá*, Hélio Lopes*, Sinésio Pesco*, Luiz Henrique de Figueiredo*

*Universidade Federal Fluminense (UFF), Brazil
*Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), Brazil
*Instituto Nacional de Matemática Pura e Aplicada (IMPA), Brazil

Abstract

We present a point-based algorithm for rendering implicit surfaces in $\mathbb{R}^4$. Our algorithm combines a new method for approximating an implicit surface with points that uses interval arithmetic for topological robustness with a new 4D illumination model that together with a color transfer function enhance the visualization of a 2-dimensional surface in 4-dimensional space. We also discuss a GPU implementation of our algorithm.

Keywords: High-dimensional rendering, point-based approximation, implicit objects, interval arithmetic.

1. Introduction

An implicit object is the set of all solutions of an equation $F(p) = 0$, where $F: \Omega \subseteq \mathbb{R}^n \rightarrow \mathbb{R}^m$. Of special interest to computer graphics are implicit curves in $\mathbb{R}^2 (n = 2, m = 1)$ [1, 2] and implicit surfaces in $\mathbb{R}^3 (n = 3, m = 1)$ [3, 4, 5]. Moreover, several problems in visual computing can be formulated as high-dimensional implicit problems [6, 7, 8]. In this paper, we study rendering schemes for the visualization of implicit surfaces (2-manifolds) in $\mathbb{R}^4 (n = 4, m = 2)$.

Previous work. Interest in studying the visual aspect of mathematical objects in 4-dimensional space dates from the beginning of the 20th century [9, 10]. The advent of computers made it possible to try to visualize higher-dimensional objects. In 1986, Banchoff [11] introduced a method for analyzing point clouds around 2-dimensional surfaces by using interactive computer graphics and also gave applications to the graphing of complex functions. Hoffmann and Zhou [8] in 1991 presented a pipeline for visualizing implicit surfaces in $\mathbb{R}^4$ that used a “polygonalization before projection” strategy to achieve better interaction rates. They also pointed out some applications of their 4-dimensional visualization method in offset curve geometry and in collision detection. Another interesting application of visualization in 4-dimensional space is in the study of complex-valued contours [12], whose purpose is to analyze the solution set of $G^{-1}(0)$, where $G: \mathbb{C}^2 \rightarrow \mathbb{C}$ is a complex function of two complex variables ($\mathbb{C}$ is the complex plane). Note that solving $G^{-1}(0)$ is equivalent to finding the inverse image of 0 of a real function $F: \mathbb{R}^4 \rightarrow \mathbb{R}^2$. Weigle and Banks [12] in 1996 presented a meshing algorithm to approximate $G^{-1}(0)$. Nieser, Poelke, and Polthier [13] in 2010 proposed an algorithm for meshing Riemann surfaces in $\mathbb{R}^2$ from explicitly given branch points with corresponding branch indices. Their meshing approach and their proposed coloring scheme for the complex plane provide a straightforward visualization of the topological structure of Riemann surfaces, which are important mathematical objects.

It is well known that in general computing a polygonal approximation of an implicit object is a challenging problem [14] for two main reasons: it is difficult to find points on the object [15] and it is difficult to connect isolated points into a mesh [16]. Moreover, these two difficulties grow exponentially with the dimension of the space where the implicit object is embedded [17].

Contributions. To avoid the computational complexity of obtaining a polygonal approximation in high-dimension space [18] just for visualization, we propose a point-based approximation algorithm for implicit surfaces in $\mathbb{R}^4$. As has been done for implicit curves in $\mathbb{R}^2$ [19], for implicit surfaces in $\mathbb{R}^3$ [20], and for non-manifold objects in $\mathbb{R}^2$ [21] the method we present here uses interval arithmetic (IA) to improve the topological robustness of the approximation. We also introduce a new illumination model for 2-dimensional objects in 4-dimensional space that combined with a color transfer function enhances the visualization. A GPU implementation for our algorithm is also proposed and discussed.

Paper outline. Section 2 presents an overview of our new point-based rendering algorithm. Section 3 introduces the point generation of surfaces in $\mathbb{R}^4$. Section 4 describes the rendering pipeline and its GPU implementation. Section 5 shows the results and Section 6 contains our conclusions and gives suggestions for future work.

Email addresses: alex@im.uff.br (Alex Laier Bordignon), luana@mat.puc-rio.br (Luana Sá), lopes@inf.puc-rio.br (Hélio Lopes), sinesio@mat.puc-rio.br (Sinésio Pesco), lhf@impa.br (Luiz Henrique de Figueiredo).

Preprint submitted to Elsevier March 26, 2013
2. Overview of our rendering algorithm

An implicit surface \( S \) in \( \mathbb{R}^4 \) is the inverse image of \((0, 0)\) of a given function \( F: \Omega \subset \mathbb{R}^4 \rightarrow \mathbb{R}^2 \). More precisely,

\[
S = F^{-1}(0, 0) = \{(x, y, z, w) \in \Omega \subset \mathbb{R}^4 : F(x, y, z, w) = (0, 0)\}
\]

We assume that \( F \) is smooth (continuously differentiable) and that \((0, 0)\) is a regular value of \( F \), which means that the derivative of \( F \) does not vanish at any point of \( S \). By the Implicit Function Theorem, this implies that \( S \) is an embedded 2-dimensional manifold in \( \mathbb{R}^4 \).

Our algorithm first computes a topologically robust point-based approximation for the implicit surface \( S \). This means that the point sample captures the topology of \( S \) and approximates the geometry of \( S \). The algorithm then processes that point set according to a graphical pipeline explained below to produce an image depicting the surface.

For simplicity, we take the domain \( \Omega \) to be a 4-dimensional box (4-box) \([h] = [x_{min}, x_{max}] \times [y_{min}, y_{max}] \times [z_{min}, z_{max}] \times [w_{min}, w_{max}] \subset \mathbb{R}^4\). We perform a subdivision of \( \Omega \) guided by a KD-tree named \( T \), discarding empty cells and locating regions with topological ambiguities, using interval arithmetic to evaluate \( F \) and its derivatives.

Once the subdivision tree \( T \) has been built, the centers of all its leaf cells are projected onto the implicit surface to yield the set \( P_0 \) of seed points for a refinement process. New points are generated on the tangent plane of the surface at each point in \( P_0 \) and again projected onto the surface, yielding a new collection of points \( P_1 \). We repeat the process with the points in \( P_1 \) to get a new set of points \( P_2 \), and so on. After \( k \) steps we have a large sample of points on surface: \( P = P_k \supset P_{k-1} \supset \cdots \supset P_1 \supset P_0 \) (see section 3). Figure 1 illustrates this point-based approximation process.

The points in \( P \) approximate the implicit surface \( S \) embedded in \( \mathbb{R}^4 \). These points are then oriented, illuminated, colored, and projected onto the display plane (see section 4). The GPU implementation of this pipeline is straightforward (see section 4.4). Figure 2 illustrates the sequence of steps in the rendering pipeline.

3. Point-set generation algorithm

Unlike previous approaches [8, 12], which build a polygonal approximation for the implicit surface in \( \mathbb{R}^4 \) or in \( \mathbb{C}^2 \), our algorithm generates a point-based approximation for the implicit object that is quite suitable for rendering purposes. This greatly simplifies the implementation since it avoids having to represent and manage the connectivity of the polygonal cells for an implicit object of codimension 2 (e.g., a 2-manifold in 4-dimensional space), especially because this connectivity becomes more complex when an adaptive subdivision scheme is adopted to explore the domain space [17].

In this section we describe the three steps of our point-set generation algorithm for implicit surfaces in \( \mathbb{R}^4 \): domain exploration using spatial adaptive subdivision (section 3.1), generation seed points (section 3.2), and the refinement step (section 3.3).

![Image](https://example.com/image.png)

Figure 1: Point-based approximation process for the implicit surface \( S \) defined by the function \( F(x, y, z, w) = ((y - 0.2w)^2 + z^2 - 1, x^2 + y^2 + (z + w)^2 - 0.49) \): (a) 2^4-tree domain subdivision; (b) center points of the leaves; (c) seed point set (\( P_0 \)); (d) final point set after 4 refinement iterations (\( P_4 \)).

![Image](https://example.com/image2.png)

Figure 2: Our rendering pipeline.
work of Balsys et al. [21] that proposed a point-based technique

...3.1.1. Interval arithmetic

R-box in [12], who use exhaustive cell enumeration to inspect the domain

...are given below.

...oracle that robustly guides the domain exploration. The details

...criteria from [19, 20], which use interval arithmetic to yield an

...some stopping criteria are met. We have chosen and adapted the

...of the node in T by assigning to the root of the KD-tree

...the initial domain, \( \Omega \), we can use the interval extension function

...as a consequence, we can use the interval extension function

...in the 4-box \([h]\) because interval estimates are in general overestimates. However, interval estimates get better as the boxes are refined.

3.1.2. Stopping criteria

A 4-box node in the KD-tree \( T \) will be subdivided until it meets one of the three stopping criteria explained below: connected component, topology, and maximum level.

Connected component. This is the main test for the subdivision of KD-tree. This test assesses whether the cell can contain points of the surface \( S \): if \( 0 \notin F([h]) \), then we can assure that there are no surface points in \([h]\). On the other hand, if \( 0 \in F([h]) \) then the surface may intercept \([h]\), and so we should subdivide it. This test ensures the robustness of the surface topology, in the sense that no connected component of the surface will be lost during the subdivision process.

Topology. The connected component criterion is not sufficient to ensure that inside the 4-box \([h]\) the surface is locally a graph of a function. If \([h]\) contains an entire component, several sheets, or a tunnel in its interior, then the interval estimate \( (dF_{1n}, dF_{2n}) \) of the derivative of \( F = (F_1, F_2) \) in \([h]\) will contain the zero vector.

Maximum depth. The previous criteria can lead to an excessive number of subdivisions. Furthermore, as mentioned before, the interval extension evaluation of \( F([h]) \) typically overestimates the image of \( F \), resulting in unnecessary subdivisions of empty regions. To avoid these drawbacks we also set a maximum depth for the KD-tree.

3.1.3. The domain exploration algorithm

Algorithm 1 summarizes the domain exploration for an initial 4-box \([h] \subseteq \Omega \) for a given function \( F : \Omega \subseteq \mathbb{R}^4 \to \mathbb{R}^2 \). The procedure divide splits the 4-box \([h]\) perpendicularly to the \( X-, Y-, Z-\) and \( W\)-coordinate axes in alternating order according to the level of the node in \( T \). Figure 3 illustrates four levels of a KD-tree subdivision starting from the root. The recursion ends when some stopping criteria are met. We have chosen and adapted the criteria from [19, 20], which use interval arithmetic to yield an oracle that robustly guides the domain exploration. The details are given below.

Note that our approach for domain exploration is different from that of Hoffmann and Zhou [8] and of Weigle and Banks [12], who use exhaustive cell enumeration to inspect the domain box in \( \mathbb{R}^4 \) or in \( \mathbb{C}^2 \).

3.1.1. Interval arithmetic

Consider a function \( F : \Omega \subseteq \mathbb{R}^4 \to \mathbb{R}^2 \) and a 4-box \([h] = ([x], [y], [z], [w]) \subseteq \Omega \). Let \( \mathbb{I}^n \) be the set of all real intervals of the form \([a, b]\) with \( a < b \) and \( a, b \in \mathbb{R} \). In our algorithm we use the interval extension \( F_1 : \mathbb{I}^m \to \mathbb{I}^n \) of the function \( F \) according to [24]:

\[
F_1([h]) = (F_1([x], [y], [z], [w]), F_2([x], [y], [z], [w])) \subseteq \mathbb{I}^2
\]

The Fundamental Theorem of Interval Arithmetic [25] says that \( F_1([h]) \) gives an interval estimate that always contains the complete range of values of \( F \) in the 4-box \([h]\):

\[
F([h]) \supseteq F_1([h]) = \{F(x, y, z, w) \in \mathbb{R}^2 : (x, y, z, w) \in [h]\}
\]

As a consequence, we can use the interval extension function \( F_1([h]) \) to reliably check that the 4-box \([h]\) does not intersect the surface, the converse is not always true, since \( (0, 0) \notin F_1([h]) \) do not guarantee that \( (0, 0) \notin F([h]) \) because interval estimates are in general overestimates. However, interval estimates get better as the boxes are refined.

Our algorithm extends to \( \mathbb{R}^4 \) the shape and tone depiction algorithm proposed by Brazil et al. [22] for non-photorealistic rendering of implicit surfaces in \( \mathbb{R}^3 \). It is also an extension of the work of Balsys et al. [21] that proposed a point-based technique to rendering non-manifold implicit surfaces in \( \mathbb{R}^3 \).

3.1. Domain exploration

A KD-tree is a kind of binary tree that it is very useful in adaptive solutions for problems involving multidimensional spaces [23]. In general, a KD-tree associates to each node a hyperbox on a metric space. Every interior node can be thought of as an implicit representation of a hyperplane partition that divides the hyperbox into two parts, known as half-spaces. The hyperbox to the left of this hyperplane is represented by the left subtree of that node and the hyperbox to the right of the hyperplane is represented by the right subtree.

In our application, the domain exploration algorithm starts by assigning to the root of the KD-tree \( T \) the initial domain, a 4-box \([h] \subseteq \Omega \subseteq \mathbb{R}^4 \). It then recursively divides the 4-box associated to a node of \( T \) perpendicularly to the \( X-, Y-, Z-, \) and \( W\)-coordinate axes in alternating order according to the level of the node in \( T \). Figure 3 illustrates four levels of a KD-tree subdivision starting from the root. The recursion ends when some stopping criteria are met. We have chosen and adapted the criteria from [19, 20], which use interval arithmetic to yield an oracle that robustly guides the domain exploration. The details are given below.

...Note that our approach for domain exploration is different from that of Hoffmann and Zhou [8] and of Weigle and Banks [12], who use exhaustive cell enumeration to inspect the domain box in \( \mathbb{R}^4 \) or in \( \mathbb{C}^2 \).
Algorithm 1: Explore([h], depth)

1: if 0 ≠ F_i([h]) then [connected component test]
2:   discard [h]
3: else
4:   if 0 ≠ dF_i([h, i]) or depth = depth_{max} then [topology and maximum depth tests]
5:     output [h]
6:   else
7:     ([h_1], [h_2]) ← divide([h], depth)
8:     Explore([h_1], depth + 1)
9:     Explore([h_2], depth + 1)
10: end if
11: end if

3.2. Seed point generation

As mentioned in [22], the quality of the final image is affected not only by the number of the seed points but also by their position, since from these seeds more points on the surface are created. Unlike [22], which uses a semi-automatic approach to place seed points on the implicit surface in \( \mathbb{R}^3 \), our algorithm obtains the seed points directly from the KD-tree leaves. The main reason for this choice is: interval arithmetic provides certain topological guarantees and we can estimate the distance of the center of a leaf to the surface itself.

The first step of our seed point generator is to select the center points of the 4-boxes associated to the leaves of a KD-tree that explores the domain of \( F(x, y, z) = (x^2 + y^2 + z^2 - 4, x^2 + y^2 + z^2 + (w - 1)^2 - 4) \). The different levels of the KD-tree: 10, 15, 20, and 25.

The second step of our seed point generator is to project the center points onto the implicit surface. For that, we have adapted the projection operator of [22], which uses a steepest descent with successive step size reduction [26] to solve the problem minimization of \( f^2(x) \), where the inverse image of \( f: \mathbb{R}^4 \to \mathbb{R} \) at 0 defines the same implicit surface.

In our case the image of the function \( F \) is a 2-dimensional space and so we propose to solve the problem of minimizing \( f: \mathbb{R}^4 \to \mathbb{R} \), where \( f(x) = ||F(x)|| \), so that \( f^2 = f_1^2 + f_2^2 \). Thus, our projection operator constructs a sequence of points from a given initial condition \( x_0 \in \mathbb{R}^4 \) as follows:

\[
x_{k+1} = x_k - \frac{f(x_k)}{||f(x_k)||} \nabla f(x_k),
\]

where \( \delta \in (0, 1) \) and \( \sigma \in (0, \frac{1}{2}) \) are parameters of the method and \( i_k \) is the minimum non-negative integer satisfying

\[
f^2(x_{k+1}) \leq f^2(x_k)\left(1 - 2\sigma \delta^k \right)
\]

The two stopping criteria for the method are: the algebraic distance \( f(x_k) \) (using a given tolerance \( \epsilon \)) and a maximum number of steps. If the sequence does not converge within the maximum number of steps, we discard the point.

In all examples of this work, we have set \( \delta = 0.1, \sigma = 0.01, \epsilon = 10^{-8} \), and the maximum number of steps to be 16.

3.3. Refinement step

We again have adapted the refinement step proposed by Brazil et al. [22] to deal with surfaces in \( \mathbb{R}^4 \).

Starting from the seed points \( P_0 \), we compute, after \( k \) refinement steps, a sample of points \( P \) on the target implicit surface: \( P = P_k \cup P_{k-1} \cup \cdots \cup P_1 \cup P_0 \). For this, we perform the following procedure.

For each sample point \( p \in P_k \), we randomly select four new points, one in each quadrant of the tangent plane of the surface at \( p \), and then project them onto the surface. To obtain these four random points on the tangent plane at \( p \), we first have to find an orthonormal basis for that plane. We use the following approach. Consider the 4 × 2 matrix \( Q \) whose the columns are the normals \( n_1 = \nabla F_1(P) \) and \( n_2 = \nabla F_2(P) \), where \( F_1 \) and \( F_2 \) are the two components of \( F \). Using the QR-decomposition we write \( N = QR \). The last two columns \( u \) and \( v \) of \( Q \) span the orthonormal complement of the subspace spanned by \( n_1 \) and \( n_2 \). Thus, the four new random points on the tangent plane of \( p \) are obtained as follows:

\[
\begin{align*}
p_1 &= p + l(u_1 u + l u_2 v) \\
p_2 &= p - l(u_1 u + l u_2 v) \\
p_3 &= p - l(u_1 u - l u_2 v) \\
p_4 &= p + l(u_1 u - l u_2 v)
\end{align*}
\]

where \( l \) is the size of the main diagonal of the 4-box and \( u_{ij} \) are independent random variables uniformly distributed in the interval [0, 1]. Figure 5 illustrates the process.

Next, we use Equation 1 to project \( p_1, p_2, p_3, p_4 \) onto the surface, obtaining \( q_1, q_2, q_3, q_4 \) (Figure 6). The point \( p \) and the new points \( q_1, q_2, q_3, q_4 \) are included in the set \( P_{k+1} \).

Figure 7 shows an example of a sequence of four refinement steps obtained from the seed point set \( P_0 \).

Since the space dimension is high and as a consequence the spatial subdivisions grows exponentially, we adopt a strategy different from the one used by Balsys et al. [21]. We subdivide the space until a given maximum depth as they do but after that we can include more points by using refinement, thus allowing one to control two parameters: the maximum depth and the number of refinement steps. In Section 5 we discuss their balance.
We now propose a rendering pipeline for surfaces in \( \mathbb{R}^4 \). We start by describing how to orient 4-dimensional space using Euler angles and the projection scheme (section 4.1). We then describe the illumination model (section 4.2) and how to enhance the visualization with transfer functions (section 4.3). Finally, we describe how to implement this pipeline on the GPU (section 4.4).

### 4. Rendering pipeline

We now propose a rendering pipeline for surfaces in \( \mathbb{R}^4 \). We start by describing how to orient 4-dimensional space using Euler angles and the projection scheme (section 4.1). We then describe the illumination model (section 4.2) and how to enhance the visualization with transfer functions (section 4.3). Finally, we describe how to implement this pipeline on the GPU (section 4.4).

#### 4.1. Space orientation and projection

The relative orientation between two orthogonal 4-dimensional cartesian coordinate systems \( xyzw \) and \( XYZW \) is described by a real orthogonal \( 4 \times 4 \) rotation matrix \( R \), which is commonly parameterized by six so-called Euler angles \( \theta_1, \ldots, \theta_6 \) or by two quaternions [18]. Rotations can be classified as active or passive. In active rotations, the object is rotated and the coordinate system is left unchanged and the coordinate axes system is rotated. We use passive rotations parameterized by the six Euler angles.

To obtain the relative orientation between the original coordinate system \( xyzw \) in which the surface lies and the viewer’s coordinate system \( XYZW \), we perform a sequence of successive rotations of \( xyzw \) to align it with \( XYZW \). This sequence has three phases:

1. The W-axis is oriented by three basic rotations \( R_{yz}(\theta_1), R_{zw}(\theta_2), R_{zw}(\theta_3) \).
2. The Z-axis is oriented by two basic rotations \( R_{xy}(\theta_4) \) and \( R_{y}(\theta_5) \), orthogonal to the oriented w-axis.
3. The XY-plane is oriented by one single basic rotation \( R_{x}(\theta_6) \).

Here, \( R_{ij} \) is a rotation matrix in the \( ij \)-plane. So, the final rotation matrix \( R \) is defined as follows:

\[
R = R_{yz}(\theta_0)R_{yz}(\theta_2)R_{xy}(\theta_4)R_{zw}(\theta_3)R_{yz}(\theta_2)R_{yz}(\theta_1).
\]

We have opted for an orthogonal projection over the viewer’s XY-plane. In other words, we define the mapping \( \Phi : \mathbb{R}^4 \rightarrow \mathbb{R}^2 \) by \( \Phi(p) = \Pi(\Pi(p - c)) \), where \( \Pi(x, y, z, w) = (x, y) \) is the orthogonal projection on the XY-plane, \( \Pi \) is the rotation matrix defined above, \( \gamma \) is the scaling factor, and \( c \) the visualization center position. For simplicity, we will assume \( c = 0 \) and \( \gamma = 1 \).

In the usual case of projecting \( \mathbb{R}^3 \) onto the screen, there is only one direction orthogonal to the display plane. By contrast, in \( \mathbb{R}^4 \) we have a 2-dimensional vector space orthogonal to the display plane. Thus, we adopt an orthogonal projection onto the XY-plane after a passive rotation given by the matrix \( R \), the complement space to XY-plane is the ZW-plane. To obtain the coordinates of the Z and W canonical vectors in the \( xyzw \) coordinate system we use that the matrix \( R \) is orthogonal: its inverse is its transpose. So, the Z and W canonical vectors in the \( xyzw \) coordinate system are the two last rows of \( R \). From now on, the fourth and the third rows of \( R \) will be called the first observer vector (W-direction) and the second observer vector (Z-direction), and will be denoted by \( r_1 \) and \( r_2 \), respectively.

#### 4.2. Illumination model

The other main contribution of this work is a new illumination model to compute the light intensity of a point on a surface \( S \) (2-dimensional manifold) embedded in \( \mathbb{R}^4 \).
In our model, the illumination \( I \in [0, 1] \) at a point \( p \in S \) is given by:

\[
I = I_a + I_s \cdot f_s \cdot f_b,
\]

where \( I_a \) is the ambient light coefficient, \( I_s \) is the specular reflection coefficient, \( f_s \) is the silhouette attenuation factor, and \( f_b \) is the boundary attenuation factor. Figure 2 shows an schematic overview of our proposal.

**Ambient light coefficient.** \( I_a \) is the usual constant ambient light parameter of illumination for all objects in a scene. Figure 8 shows the effect of \( I_a \) on the final illumination.

**Specular reflection coefficient.** Given a point \( p \in S \), let \( \alpha_1 \) be the angle between the first observer vector \( r_1 \) and the tangent plane \( \pi \) at \( p \). Analogously, let \( \alpha_2 \) be the angle between the second observer vector \( r_2 \) and the tangent plane \( \pi \).

As discussed above, \( \pi \) has an orthonormal basis given by the two vectors \( v_1, v_2 \in \mathbb{R}^3 \). So, to compute \( \alpha_1 \) and \( \alpha_2 \) (Figure 9), we first project the two observer vectors \( r_1 \) and \( r_2 \) onto the plane \( \pi \) according to:

\[
\tilde{r}_1 = (r_1, v_1) v_1 + (r_1, v_2) v_2
\]

\[
\tilde{r}_2 = (r_2, v_1) v_1 + (r_2, v_2) v_2
\]

Then, we compute the cosine of the angles using:

\[
\cos(\alpha_1) = \frac{\langle \tilde{r}_1, \tilde{v}_1 \rangle}{||\tilde{r}_1||}, \quad \cos(\alpha_2) = \frac{\langle \tilde{r}_2, \tilde{v}_2 \rangle}{||\tilde{r}_2||}
\]

Finally, we define the specular coefficient as

\[
I_s = (1 - \cos^2(\alpha_1) \cos^2(\alpha_2))^{k_s}
\]

where \( k_s \) is a constant between 0 and 1. Thus, \( I_s \) will be close to zero (no illumination) for small values of \( \alpha_1 \) and \( \alpha_2 \). Figure 10 illustrates the effects of the \( k_s \) parameter.

**Silhouette intensity depends on the angles \( \alpha_1 \) and \( \alpha_2 \).**

**Silhouette attenuation factor.** Again, let \( v_1 \) and \( v_2 \) be a basis for the tangent plane \( \pi \) at point \( p \in S \) and \( V = \text{span}(\Phi(v_1), \Phi(v_2)) \), where \( \Phi(p) = \Pi(Rp) \) and \( \Pi(x, y, z, w) = (x, y) \) is the orthogonal projection on the XY-plane. According to [8], the point \( p \) is a silhouette point with respect to projection \( \Phi \) if \( \dim(V) \leq 1 \), which means that the projection of tangent space is a line or a point.

If \( r_1 \) and \( r_2 \) are the vectors in the XYZW coordinate system that span the ZW-plane, then a point \( p \) is a silhouette point if and only if the vectors \( v_1, v_2, r_1, r_2 \) are linearly dependent. Equivalently, \( p \) is a silhouette point if and only if the vectors \( n_1, n_2, m_1, m_2 \) are linearly dependent, where \( n_1, n_2 \) are the normal vectors of the surface at \( p \) and \( m_1, m_2 \) form a basis for the XY-plane (these two vectors correspond to the first two rows of \( R \)). We can test this linear dependence using the equation:

\[
\det(n_1, n_2, m_1, m_2) = 0
\]

We use this determinant in the definition of the silhouette attenuation factor:

\[
f_s = \min(1, (k_s \cdot |\det(n_1, n_2, m_1, m_2)|)),
\]

where \( k_s \in [0, 1] \) is a parameter that controls the intensity of the silhouette. Note that \( f_s \) is approximately zero near the silhouette. In Figure 2 we used \( k_s = 0.015 \). Figure 11 illustrates the effects of the variation of \( k_s \).

**Boundary attenuation factor.** The idea behind the boundary attenuation factor is that it tends to zero for points that are close to the boundary of the surface. More precisely, the boundary attenuation factor \( f_b \) at a point \( p \) in the surface \( S \) is given by

\[
f_b = \min(1, (k_b \cdot D(p))),
\]

where \( D(p) \) is the distance between \( p \) and the boundary of \( S \) and \( k_b \in [0, 1] \) is a parameter that controls the attenuation of the boundary. In Figure 2 we used \( k_b = 0.015 \). Figure 12 illustrates the effects of \( k_b \) over the global illumination model.

### 4.3. Transfer functions

The rendering can be much improved by using a transfer function to highlight important properties of the surface. In our model we use a color attribute for each point \( p \in S \) and define a transfer function \( T : S \rightarrow C_{rgb} = [0, 1] \times [0, 1] \times [0, 1] \), such that \( T = g \circ t \), where \( t : S \subset \mathbb{R}^2 \rightarrow \mathbb{R} \) assigns a scalar value for one particular property and \( g : \mathbb{R} \rightarrow C_{rgb} \) defines the colormap.

For example, let us consider the surface \( S = G^{-1}(0) \) where \( G(U, V) = V - \cos(U) \) and \( U, V \in \mathbb{C} \). This is the graph of the complex cosine function. We would like to visualize the surface \( S \) using a colormap that indicates the phase of \( U \). First, we transform this complex problem into real problem by writing \( U = x + yi \) and \( V = z + wi \) and solving \( F^{-1}(0) \) where \( F(x, y, z, w) = (z - \cos(x) \cosh(y), w + \sin(x) \sinh(y)) \). To visualize the phase of \( U \), we use the transfer function \( t(x, y, z, w) = \text{atan}(y/x) \) (Figure 13a). To visualize the length of \( U \), we use \( t(x, y, z, w) = \sqrt{x^2 + y^2} \) (Figure 13b). Other examples of simple transfer functions are shown in Figures 13c–f.

For the particular case of complex valued functions visualization in the plane, an impressive color scheme, called domain coloring, was proposed by Farris [27]: it associates to each complex number a color on a diagram, typically intensity portrays absolute value and hue portrays angle. Other methods for the visualization of complex valued functions (e.g., phase portraits and chessboard-like schemes) are described in the recent book by Wegert [28]. These color schemes can be also implemented as a color transfer function in our rendering pipeline.
Figure 8: Effect of the ambient light coefficient.

Figure 10: Effect of the specular reflection coefficient.

Figure 11: Effect of the silhouette attenuation factor.

Figure 12: Effect of the boundary attenuation factor.
We now report experiments with our method using the functions listed in Table 1 with the domain $\Omega$ taken as the 4-box $[h] = [-2, 2]^4$. We run all the experiments on an Intel Core i7 2.6 GHz with 4Gbytes of memory and Intel HD Graphics 3000 on board.

and Polthier [29] extended the work of Farris and others on domain coloring by proposing lifted domain coloring, an elegant topology-aware coloring scheme for visualizing Riemann surfaces. We differ from their work because we do not need or use any explicit knowledge of the topology of the surface being studied.

4.4. GPU implementation

Our rendering scheme can be easily implemented in the GPU. First, we send each point $p = (x, y, z, w) \in \mathcal{S}$ to the GPU with its two associated tangent vectors $v_1(x, y, z, w)$ and $v_2(x, y, z, w)$, a total of 12 floats. Next, the vertex shader applies the rotation and projection matrix, enabling the modification of the rotation and projection matrix in real time. After this, the fragment shader computes the illumination and applies the transfer function, whose parameters can be modified in real time. Finally, the final color is computed using the transfer function multiplied to the illumination factor. A typical shader is shown in the Appendix.

5. Results

We now report experiments with our method using the functions listed in Table 1 with the domain $\Omega$ taken as the 4-box $[h] = [-2, 2]^4$. We run all the experiments on an Intel Core i7 2.6 GHz with 4Gbytes of memory and Intel HD Graphics 3000 on board.

Figure 13: Visualization of $F^{-1}(0)$ where $F(x, y, z, w) = (z - \cos(x) \cosh(y), w + \sin(x) \sinh(y))$ using different transfer functions: (a) $t(x, y, z, w) = \arctan(y/x)$; (b) $t(x, y, z, w) = \sqrt{x^2 + y^2}$; (c) $t(x, y, z, w) = \arctan(w/z)$; (d) $t(x, y, z, w) = \sqrt{z^2 + w^2}$; (e) $t(x, y, z, w) = x$; (f) $t(x, y, z, w) = y$; (g) $t(x, y, z, w) = z$ and (h) $t(x, y, z, w) = w$.

4.4. GPU implementation

Our rendering scheme can be easily implemented in the GPU. First, we send each point $p = (x, y, z, w) \in \mathcal{S}$ to the GPU with its two associated tangent vectors $v_1(x, y, z, w)$ and $v_2(x, y, z, w)$, a total of 12 floats. Next, the vertex shader applies the rotation and projection matrix, enabling the modification of the rotation and projection matrix in real time. After this, the fragment shader computes the illumination and applies the transfer function, whose parameters can be modified in real time. Finally, the final color is computed using the transfer function multiplied to the illumination factor. A typical shader is shown in the Appendix.

5. Results

We now report experiments with our method using the functions listed in Table 1 with the domain $\Omega$ taken as the 4-box $[h] = [-2, 2]^4$. We run all the experiments on an Intel Core i7 2.6 GHz with 4Gbytes of memory and Intel HD Graphics 3000 on board.

![](image1.png)

![Image 2](image2.png)

![Image 3](image3.png)

Figure 13: Visualization of $F^{-1}(0)$ where $F(x, y, z, w) = (z - \cos(x) \cosh(y), w + \sin(x) \sinh(y))$ using different transfer functions: (a) $t(x, y, z, w) = \arctan(y/x)$; (b) $t(x, y, z, w) = \sqrt{x^2 + y^2}$; (c) $t(x, y, z, w) = \arctan(w/z)$; (d) $t(x, y, z, w) = \sqrt{z^2 + w^2}$; (e) $t(x, y, z, w) = x$; (f) $t(x, y, z, w) = y$; (g) $t(x, y, z, w) = z$ and (h) $t(x, y, z, w) = w$.

and Polthier [29] extended the work of Farris and others on domain coloring by proposing lifted domain coloring, an elegant topology-aware coloring scheme for visualizing Riemann surfaces. We differ from their work because we do not need or use any explicit knowledge of the topology of the surface being studied.

4.4. GPU implementation

Our rendering scheme can be easily implemented in the GPU. First, we send each point $p = (x, y, z, w) \in \mathcal{S}$ to the GPU with its two associated tangent vectors $v_1(x, y, z, w)$ and $v_2(x, y, z, w)$, a total of 12 floats. Next, the vertex shader applies the rotation and projection matrix, enabling the modification of the rotation and projection matrix in real time. After this, the fragment shader computes the illumination and applies the transfer function, whose parameters can be modified in real time. Finally, the final color is computed using the transfer function multiplied to the illumination factor. A typical shader is shown in the Appendix.

5. Results

We now report experiments with our method using the functions listed in Table 1 with the domain $\Omega$ taken as the 4-box $[h] = [-2, 2]^4$. We run all the experiments on an Intel Core i7 2.6 GHz with 4Gbytes of memory and Intel HD Graphics 3000 on board.

<table>
<thead>
<tr>
<th>function</th>
<th>$F(x, y, z, w) =$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$(y - 2zw, \ x - x^2 + w^2)$</td>
</tr>
<tr>
<td>$b$</td>
<td>$(y^2 + z^2 - 1, \ x^2 + y^2 + (z + w)^2 - 1.44)$</td>
</tr>
<tr>
<td>$c$</td>
<td>$(x^2 - y^2 - z^2 + w^2 - 0.5, \ 2xy - 2zw)$</td>
</tr>
<tr>
<td>$d$</td>
<td>$(x^2 + y^2 - 1.0, \ z^2 + w^2 - 1.0)$</td>
</tr>
<tr>
<td>$e$</td>
<td>$(x - \exp(z) \cos(w), \ y - \exp(z) \sin(w))$</td>
</tr>
<tr>
<td>$f$</td>
<td>$(x^2 + 3xz + \cos(xw) - 3w + 1, \ 2zw + y^3 + xy)$</td>
</tr>
<tr>
<td>$g$</td>
<td>$(x^3y^2 + 2w, \ z^2w^2 + xy)$</td>
</tr>
<tr>
<td>$h$</td>
<td>$(x^3 - 3xy^2 - z, \ 3x^2y - y^3 - w)$</td>
</tr>
<tr>
<td>$i$</td>
<td>$(x^2 - y^2 - z^2 + w^2 - 1, \ 2xy - 2zw - 1)$</td>
</tr>
<tr>
<td>$j$</td>
<td>$(x^2 + z^2 + w^2 - 0.64, \ y^2 + z^2 + w^2 - 0.64)$</td>
</tr>
<tr>
<td>$k$</td>
<td>$(y^2 + z^2 - 1, \ x^2 + y^2 + (z + w)^2 - 1)$</td>
</tr>
</tbody>
</table>

Table 1: Functions used in our experiments.

5.1. Point generation process

Table 2 shows the processing times (in milliseconds) to build the KD-tree using different maximum depths to generate the seed points and to run two refinement steps for the function $a$ in Table 1. Here we can see that the total time is proportional to the maximum depth parameter. It is important to note that the user can balance the maximum depth with the number of refinement steps in order to get the desired image quality.

Table 3 shows processing times for the point generation algorithm using the functions listed in Table 1. We took the maximum depth of KD-tree as 40. This table shows that the processing time depends significantly on the evaluation complexity.
of the function and its derivatives (transcendent functions are more expensive), caused mainly by their interval evaluation. We can observe this fact also in Figure 14 which shows a graph of the total preprocessing time for different maximum levels of the KD-tree, using two refinement steps.

5.2. Frame rates

Table 4 shows the frame rates and the total number of points rendered considering the functions listed in Table 1. We took the maximum depth of the KD-tree as 40. We can observe that, as expected, the frame rate is inversely proportional to the number of points. The same fact can also be observed in Table 5, which shows the evolution of frame rates for different maximum depth of KD-tree using function $b$ of Table 1.

5.3. Function gallery

Figure 15 shows projections of the implicit surfaces listed in Table 1. These examples include surfaces with genus (Figures 15c, d and f) and with several connected components (Figure 15g). For all of them, we took the maximum depth as 40 and use two refinement steps.

5.4. Kaleidoscope of surfaces

Another way to visualize the surfaces in $\mathbb{R}^4$ is to generate an animation by varying the six Euler angles along a trajectory. The user defines this trajectory in this 6-dimensional space using a parametrized curve that is discretized into a number of steps. At each step, the surface is projected and illuminated using our pipeline. The visual result of this strategy feels like a kaleidoscope where a fixed surface assumes different forms at different projections.

A simple curve like the line segment that joins the initial and final Euler angles positions already generates an interesting result, as one can see in Figures 16 and 17. Figure 16 starts visualising the surface perpendicularly to the XY-plane and finishes perpendicularly to ZW-plane. This surface corresponds to a $(x + iy) - (z + iw)^2 = 0$, where $i^2 = -1$. The resulting images show how the surface’s boundary curve projection on the XY-plane that has winding number two around the XY-plane origin is transformed to a curve in the ZW-plane that has a winding number one around the origin on that plane. This result is expected since the surface is a graph of a complex function that takes the square root of $(x + iy)$. Figure 17 shows an example of a genus-one surface that is not the graph of a function, in fact it corresponds to an algebraic complex curve: $U^2 - V^2 = 0$, with $U, V \in \mathbb{C}$.

6. Conclusions and future works

In this paper, we presented a new point-based rendering scheme for surfaces in 4D space. We introduced new ideas including a simple and robust point-based approximation for implicit surfaces and a suitable illumination model for a 2-dimensional surface in $\mathbb{R}^3$. We also proposed an animation scheme, called the kaleidoscope of surfaces, that interpolates two different observer positions to facilitate the surface visualization.

We plan to extend this technique to render 2- or 3-dimensional implicit objects embedded in even higher dimensional spaces.
Table 3: Preprocessing times in milliseconds using the functions of Table 1 (maximum depth of KD-tree is 40).

<table>
<thead>
<tr>
<th>function</th>
<th>KD-tree subdivision</th>
<th>seed point generation</th>
<th>refinement step 1</th>
<th>refinement step 2</th>
<th>total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2657</td>
<td>293</td>
<td>368</td>
<td>1417</td>
<td>4735</td>
</tr>
<tr>
<td>b</td>
<td>2799</td>
<td>395</td>
<td>821</td>
<td>2897</td>
<td>6912</td>
</tr>
<tr>
<td>c</td>
<td>5147</td>
<td>560</td>
<td>727</td>
<td>2911</td>
<td>9345</td>
</tr>
<tr>
<td>d</td>
<td>1436</td>
<td>208</td>
<td>665</td>
<td>2796</td>
<td>5105</td>
</tr>
<tr>
<td>e</td>
<td>25733</td>
<td>5418</td>
<td>2614</td>
<td>4109</td>
<td>37874</td>
</tr>
<tr>
<td>f</td>
<td>35023</td>
<td>6360</td>
<td>2514</td>
<td>9129</td>
<td>53026</td>
</tr>
<tr>
<td>g</td>
<td>4462</td>
<td>612</td>
<td>1351</td>
<td>4095</td>
<td>10520</td>
</tr>
<tr>
<td>h</td>
<td>3773</td>
<td>492</td>
<td>693</td>
<td>2713</td>
<td>7671</td>
</tr>
<tr>
<td>i</td>
<td>4944</td>
<td>541</td>
<td>688</td>
<td>2760</td>
<td>8933</td>
</tr>
<tr>
<td>j</td>
<td>2419</td>
<td>329</td>
<td>833</td>
<td>3433</td>
<td>7014</td>
</tr>
<tr>
<td>k</td>
<td>2288</td>
<td>321</td>
<td>732</td>
<td>2859</td>
<td>6200</td>
</tr>
</tbody>
</table>

Figure 14: Comparing the total preprocessing time in milliseconds for different maximum levels of the KD-tree.
Figure 15: Projections of different implicit surfaces in $\mathbb{R}^4$. 

(a) function $a$
(b) function $b$
(c) function $c$
(d) function $d$
(e) function $e$
(f) function $f$
(g) function $g$
(h) function $h$
(i) function $i$
(j) function $j$
Figure 16: Kaleidoscope of the implicit surface defined by function $a$.

Figure 17: Kaleidoscope of the implicit surface defined by function $c$. 
Appendix: Fragment shader example

uniform sampler1D g_colormap_texture;
uniform mat4x4 g_reduction_mat;
uniform float g_colormap_max;
uniform float g_colormap_min;
uniform float g_Kambient;
uniform float g_Kspecular;
uniform float g_Ksilhouette;
uniform float g_Kborder;
uniform int g_is_kdtree;
varying vec4 g_t1;
varying vec4 g_t2;
varying vec4 g_coords;

float compute_det(mat4 m)
{
    // omitted to save space
}

void main(void)
{
    vec4 obs1 = vec4(g_reduction_mat[0][2], g_reduction_mat[1][2], g_reduction_mat[2][2], g_reduction_mat[3][2] ) ;
    vec4 obs2 = vec4(g_reduction_mat[0][3], g_reduction_mat[1][3], g_reduction_mat[2][3], g_reduction_mat[3][3] ) ;
    vec4 p1 = dot(obs1,g_t1)*g_t1 + dot(obs1,g_t2)*g_t2;
    vec4 p2 = dot(obs2,g_t1)*g_t1 + dot(obs2,g_t2)*g_t2;
    float ct1 = dot(obs1,p1) / sqrt (dot(p1,p1)); //cos theta 1
    float ct2 = dot(obs2,p2) / sqrt (dot(p2,p2)); //cos theta 2
    float det = abs(compute_det(mat4(g_t1,g_t2,obs1,obs2)));
    vec4 t = abs(g_coords));
    float t1 = max (abs(t.x),abs(t.y));
    float t2 = max (abs(t.z),abs(t.w));
    float t3 = max (t1,t2);
    float dist_border = 2.1-t3;
    float Fambient = g_Kambient;
    float Fspecular = pow(1.0-(ct1*ct1*ct2*ct2),g_Kspecular);
    float Fsilhuette = g_Ksilhouette *det*det;
    float Fborder = dist_border*g_Kborder;
    Fambient = min(Fambient,1.0);
    Fspecular = min(Fspecular,1.0);
    Fsilhuette = min(Fsilhuette,1.0);
    Fborder = min(Fborder,1.0);
    vec4 the_illumination = Fborder*Fsilhuette*vec4(Fambient+Fspecular);
    //transfer function
    float x = g_coords.x;
    float y = g_coords.y;
    float z = g_coords.z;
    float v = g_coords.w;
    float theta = atan(y/x);
    if (x==0) theta = 0.0;
    float v = theta;
    token_transfer_function
    v = clamp(v,g_colormap_min,g_colormap_max);
    v = (v - g_colormap_min)/(g_colormap_max - g_colormap_min);
    //compute illumination
    vec4 the_color = texture1D(g_colormap_texture, v); //transfer function
    gl_FragColor = the_color*the_illumination;
}
References


