Volume Graphics (2005)
E. Gröller, I. Fujishiro (Editors)

Computing and Rendering Implicit Surfaces Composed of Radial Basis Functions on the GPU

Andrew Corrigan and H. Quynh Dinh[†]

Stevens Institute of Technology

Abstract

Analytical implicit surfaces composed of radial basis functions (RBFs) have received much recent attention. In this paper, we present a method to compute and visualize implicit surfaces represented by a summation of weighted RBFs. We use a compactly-supported RBF, and discretely sample the 3D basis function into a 3D texture that is mapped to overlapping cubes centered at surface points. We use the GPU to compute the implicit function value at every rendered pixel and directly display the shaded iso-surface without extracting a polygonal representation.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Display Algorithms

1. Introduction

We present a method to compute (or evaluate) and visualize implicit surfaces represented as a summation of weighted radial basis functions (RBFs). Implicit surfaces can describe shapes of arbitrary topology and closed surfaces without requiring a seam. Analytical implicit functions such as the one we use are more compact than discrete representations and can be evaluated at arbitrary resolution.

Implicit surfaces remain difficult to visualize and manipulate interactively because they require root-finding to locate the surface. A standard method for visualizing and interacting with implicit surfaces is to first extract an explicit representation (a mesh) using Marching Cubes [7]. Iso-surface extraction is highly non-interactive, however, and would not be appropriate for a modeling application in which the suface changes. Other standard methods for visualizing implicit surfaces are ray-tracing [6] and volume rendering [2]. These high quality rendering algorithms are ideal for creating still images, but are not generally efficient enough for an interactive visualization. Recent interactive ray-tracing algorithms rely on pre-sorting the scene and would not be effective for editing surfaces. Implicit surfaces have also been visualized using dynamic particle systems [13]. In [4], Hart *et al.* extend these ideas to handle more complex implicit shapes and provide a system that can support any implicit surface representation. Unfortunately, the number of surface points used as constraints is limited to tens, rather than hundreds.

Instead, we develop a mesh-less approach that uses the graphics hardware to perform interpolation, weighting, and summation of RBFs. We discretely sample a compactly-supported RBF into a texture (2D for implicit contours and 3D for implicit surfaces) that is mapped to cubes (or squares for 2D contours) centered at surface points. We then use the GPU to apply the weights to the RBF stored in the texture, accumulate the RBFs forming the implicit function, and render the shaded iso-surface without extracting a polygonal representation in an approach inspired by the work of Westermann et al. [12]. To improve the efficiency of the rendering, we compute only fragments that can be seen in a view-dependent manner.

In Section 2, we review work in implicit representations using RBFs and GPU-enabled methods for rendering implicit surfaces. We provide an overview of our approach in Section 3, describe the method to compute and render implicit surfaces in Section 4, and discuss results in Section 5.

2. Related Work

2.1. Implicit Functions Composed of RBFs

Analytical implicit surfaces composed of radial basis functions (RBFs) have received much recent atten-

† acorriga,quynh@cs.stevens.edu

© The Eurographics Association 2005.

tion [1, 3, 8, 9]. In these representations, the implicit function $f(\vec{x})$ is defined as follows:

$$f(\vec{x}) = \sum_{i=1}^{n} w_i \phi(\vec{x} - \vec{c}_i) + P(\vec{x}) \quad (1)$$

In the above equation, ϕ is the RBF, n is the number of RBFs; $\vec{c}_i$ is the center of RBF i; w_i is the weight of the RBF; and $P(x)$ is a polynomial spanning the null space of the basis function. We use the convention that $f(\vec{x}) = 0$ for all points on the surface; $f(\vec{x}) < 0$ inside the object; and $f(\vec{x}) > 0$ outside the object. The parameters affecting the implicit shape are the locations of the RBF, the weight, and the defintion of the RBF ϕ (shape and radius of influence). In practice, we position our RBFs at the vertices of simplified polygonal models. The weights of the RBFs are found by solving a linear system. RBFs with infinite support (e.g., thin-plate basis) generate dense systems, whereas compactly-supported RBFs result in a sparse system that can be solved using conjugate gradient. In our work, we use the RBF derived by Wendland [11] and used in [8, 9]. These RBFs are not only compactly-supported, but also provide 2nd order continuity in three dimensions and guarantee that the linear system is positive-definite. For a symmetric and positive-definite system matrix, a uniform radius is required for all RBFs.

2.2. Rendering Optimizations for Implicit Surfaces

In [9], Reuter *et al.* use a sphere to bound the space spanned by an RBF positioned on the surface. They project each sphere to the image plane, and for each pixel of the resulting disc, they compute a 3D surface point by interpolating between the near and far tangent planes of the sphere using regula falsi. Their performance is on the order of several seconds per frame for models of several hundred RBFs (e.g., 2.1 seconds for the Stanford Bunny with 450 RBFs).

More related work appears in [5] where Jang *et al.* use truncated Gaussians to perform scattered data interpolation of volume datasets. They spatially partition the RBF-encoded volume, resulting in cells that contain a variable number of RBFs. For each fragment, the shader accesses RBF centers and weights stored in textures, and computes the implicit function value by evaluating the Gaussian RBFs incident on the fragment. This requires a computationally expensive shader that calls the exponential function in a loop traversing all the RBFs in a cell and context switching between fragment shaders that handle loops of different sizes. We also evaluate the implicit function value for each fragment. However, our approach is completely different and is less computationally intensive for the fragment shader.

3. Overview

We have developed an application to visualize and edit implicit surfaces composed of RBFs. Our system loads point constraints where RBFs are centered, solves for the weights using conjugate gradient, and directly computes and renders the implicit surface on the GPU. Once an implicit surface has been loaded, the user can interact with the surface by selecting a region on the surface and pushing or pulling the selected region (Figure 2b). As the user edits the surface, the weights are updated by re-solving the system matrix in software. Our system evaluates and displays the implicit surface on every frame, so the surface visualization is simultaneously updated as the weights are updated. In the next section, we describe the novel components in our system – namely, the computation and rendering of the implicit surface.

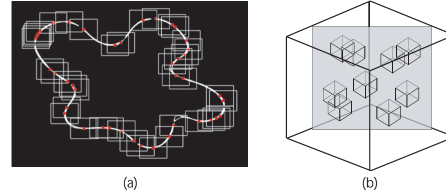

Figure 1: *(a) Squares where 2D RBFs are texture mapped are shown outlined on top of the iso-contour. (b) A slicing plane cutting the 3D domain intersects multiple RBFs.*

4. Computing and Rendering Point-Based Implicit Surfaces

In developing our implicit modeling system, we adapt ideas from the work of Westermann *et al.* [12] for interactive volume rendering and from the work of Strzodka [10] for computing distance fields. In [10], Strzodka computes the 2D distance transform by pre-computing an arbitrarily defined 2D distance function from a point and storing that distance field in a texture. The distance field for a given shape is then generated by rendering the pre-computed texture centered at every boundary point with the blend mode set to GL_MIN. In our approach, we store a discrete version of the compactly-supported RBF in a texture. For 2D contours, we render squares centered at points on the contour and texture map the square with the 2D RBF function. Instead of blending, a fragment shader applies the weight associated with each RBF and accumulates the weighted RBF values at each pixel as each square is rendered as shown in Figure 1a. The implicit function value at each pixel is accumulated in the pbuffer. Note that pbuffers can store signed floating point values. The texturing hardware performs the interpolation of the texture coordinates and texture values so that each pixel samples the RBF function at coordinates corresponding to the pixel's distance from the RBF center. After all RBFs have been rendered to the pbuffer, the iso-contour is then displayed by passing through only fragments that have a total value of zero. We again use a fragment shader to check and pass or discard fragments. Note that to add polynomial components, we render a quadrilateral covering the entire domain that is textured with the linear weights.

Unfortunately, computing and rendering the 3D implicit

© The Eurographics Association 2005.

surface is not as straight-forward because the graphics hardware renders only to 2D framebuffers not 3D volumes. ATI's *uberbuffers* provide capabilities to render directly to 3D textures, but only a single 2D slice can be rendered to the 3D texture at any one time. Hence, the sampled 3D RBF stored in a 3D texture cannot be weighted and rendered directly to a volume in one pass as was done for 2D implicit functions. Instead, we build upon the approach of [12] for interactive volume rendering to compute the 3D implicit function via rendering of 2D slices of the 3D RBF texture. In [12], Westermann *et al.* render planes parallel to the image plane that are clipped against the 3D texture domain. The hardware interpolates the 3D texture coordinates and blends successive parallel planes that have been textured with the volume data. Instead of a single 3D texture that spans the full volume, we have an RBF texture that is texture mapped to multiple locations and at each location, spans only a small cube with dimensions equal to the radius of the RBF.

As in [12], we compute planes parallel to the image plane which we call *slicing* planes. Each slicing plane is intersected with the cubes bounding RBFs as shown in Figure 1b. For each cube that is incident on the slicing plane, the resulting intersection is a quadrilateral or triangle that is texture mapped using the 3D texture coordinates derived from the intersection. As with computing 2D implicit functions, the texture-mapped quadrilateral or triangle is multiplied with the corresponding RBF's weight and accumulated in a pbuffer for each slicing plane. Note that the pbuffer stores only one slice (potentially, an off-axis slice) of the 3D implicit function at any one time. To display the iso-surface, we pass only fragments in the pbuffer that have a value of zero (+/- a small delta) and accumulate the slices in a back-to-front manner in the pbuffer's back buffer. Only two fragment shaders are required for processing a slice – one to sample the RBF and accumulate the weighted RBF value in a pbuffer, and the second to add linear components to the slice and extract the iso-contour. Hence, only one shader context switch is required per slice in the following pseudocode:

For each slice
  For each incident RBF
    Render intersection polygon (calls 1st *shader)*
  Render slice polygon (calls 2nd *shader)*

For each rendered pixel, the implicit function is computed by weighting and summing all RBFs incident on the pixel. Hence, when zooming-in or displaying larger views, more surface pixels are displayed, but few aliasing artifacts appear because the implicit function is computed exactly for each pixel (see Figure 2). Sampling artifacts due to the discretization of the RBF function are minimal because the texture is mapped to a small area (RBF cube intersection versus the entire slicing plane as in [12]) and the graphics hardware interpolates texture values. Since we use only one pre-computed RBF function to texture-map to all surface constraints, the RBF texture can be high in resolution. We have found a resolution of 128^3 to be sufficient for avoiding sampling artifacts. Displaying more pixels as a result of zooming-in or drawing larger views does incur a rendering penalty since each pixel has to be passed through the fragment shader for computing the implicit function.

4.1. Software and Hardware Optimizations

The main bottleneck in our approach are the operations in the fragment shader. Thus far, we have described the slicing plane as traversing the volume from back-to-front. This traversal direction is necessary in conventional volume rendering when successive slices are blended, but we want to display only the iso-surface of the implicit function. Hence, we can render the slicing plane in front-to-back order and display only pixels that are visible (and closest) to the eye, implemented via depth testing. With depth-culling, pixels that are occluded are never passed to the fragment shader where the most expensive computations take place.

4.2. Surface Shading and Texturing

We render iso-surfaces with diffuse shading by pre-computing the RBF gradient and storing the gradient in the color channels of the texture storing the RBF values. As the weights are applied to RBF values, they are also applied to the gradient and accumulated simultaneously in the pbuffer. The gradient is used as the Normal vector in diffuse shading. We also texture the implicit surface with a 3D solid texture. For each slice, 3D texture coordinates are generated for all pixels that are part of the iso-surface by interpolation from the points of intersection with the slicing plane.

5. Results

The parameters that affect rendering speed are the number of slicing planes, the RBF radius, and the size of the viewport. A large RBF radius will result in more RBF intersections with slicing planes, requiring more computation. The number of slicing planes affects rendering quality. With few slices, the difference in shading from one slice to the next is more apparent because the spacing between slices is large, whereas with many slices, the slices are close together and the difference in shading is gradual and smooth (as shown in Figure 2). Table 1 are timing results on various models.

Viewport			1024^2	512^2	512^2	512^2
Model	Points	Radius	Slices: 50	50	100	200
Whiffle	92	0.2	0.07	0.05	0.08	0.16
Cylinder	220	0.2	0.30	0.15	0.29	0.58
Bunny	500	0.2	0.20	0.14	0.27	0.52
Zebra	800	0.2	0.47	0.28	0.57	1.12
Octopus	1000	0.2	0.47	0.31	0.61	1.23

Table 1: *Timing results (in secs.) without depth-culling*

© The Eurographics Association 2005.

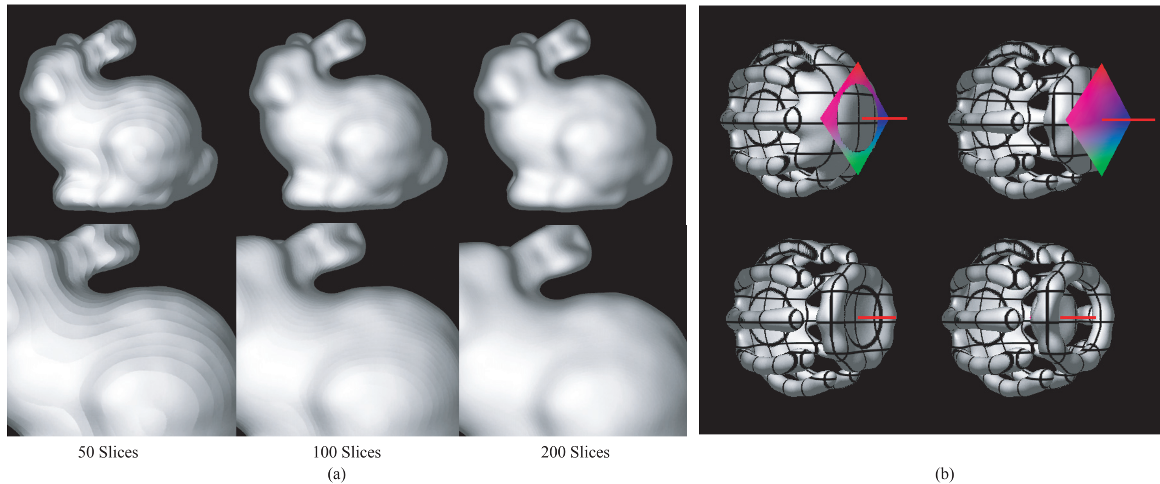

Figure 2: *(a) Top: Stanford Bunny rendered with varying number of slices. Bottom: Close-ups show that the low resolution of the RBF texture is not apparent. (b) Editting the whiffle ball by pulling and pushing the surface.*

6. Conclusions and Future Work

We have presented a method to compute and visualize implicit surfaces on the GPU. Our implicit surfaces are analytically defined as a summation of compactly-supported radial basis functions. Although we demonstrate our approach with a particular type of RBF, we note that it is not restricted to this RBF. The only limitation is that the RBF must be compactly-supported because we pre-compute and store the sampled RBF in a 3D texture.

References

[1] Carr, J., R.K. Beatson, J.B. Cherrie, T.J. Mitchell, W.R. Fright, and B.C. McCallum. "Reconstruction and Representation of 3D Objects With Radial Basis Functions", *Computer Graphics Proceedings* (SIGGRAPH 2001), August 2001, pp.67–76.

[2] Drebin, R.A., L. Carpenter, P. Hanrahan. "Volume Rendering", *Computer Graphics Proceedings* (SIGGRAPH 88), pp.65–74.

[3] Dinh, H.Q., G. Turk, and G. Slabaugh. "Reconstructing Surfaces By Volumetric Regularization Using Radial Basis Functions", *IEEE PAMI*, October 2002, pp.1358–1371.

[4] Hart, J.C., E. Bachta, W. Jarosz., and T. Fleury. "Using Particles to Sample and Control More Complex Implicit Surfaces" *Proceedings of Shape Modeling International 2002*, May 2002.

[5] Jang, Y., M. Weiler, M. Hopf, J. Huang, D.S. Ebert, K.P. Gaither, and T. Ertl. "Interactively Visualizing Procedurally Encoded Scalar Fields", *Proceedings of Eurographics Symposium on Visualization*, 2004.

[6] Levoy, M. "Efficient Ray-Tracing of Volume Data", *ACM Transactions on Graphics*, July 1990, Vol.9(3), pp.245–261.

[7] Lorensen, W. and H.E. Cline, "Marching Cubes: A High Resolution 3-D Surface Construction Algorithm," *Computer Graphics* (SIGGRAPH 87), Vol.21(4), July 1987, pp. 163–169.

[8] Morse, B.S., T.S. Yoo, P. Rheingans, D.T. Chen, and K.R. Subramanian. "Interpolating Implicit Surfaces from Scattered Data Using Compactly Supported Radial Basis Functions" *Proceedings of Shape Modeling International*, May, 2001, pp.89–98.

[9] Reuter, P., I. Tobor, C. Schlick, and S. Dedieu. "Point-based Modelling and Rendering Using Radial Basis Functions", *Proceedings of 1st Intl Conf. on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, Feb. 2003, pp.111–118.

[10] Strzodka, T. "Generalized Distance Transforms and Skeletons in Graphics Hardware", *Proceedings of Symposium on Visualization*, 2004.

[11] Wendland, H. "Piecewise Polynomial, Positive Definite and Compactly Supported Radial Functions of Minimal Degree", *Advances in Computational Mathematics*, 1995, Vol.4, pp.389–396.

[12] Westermann, R. and T. Ertl. "Efficiently Using Graphics Hardware in Volume Rendering Applications", *Computer Graphics Proceedings* (SIGGRAPH 98) July 1998, pp.169–177.

[13] Witkin, A. and P. Heckbert. "Using Particles to Sample and Control Implicit Surfaces", *Computer Graphics Proceedings* (SIGGRAPH 94), pp.269–277.

© The Eurographics Association 2005.