Real Time Algebraic Surface Visualization

Johan S. Seland
Centre of Mathematics for Applications, University of Oslo, Norway.
johans@cma.uio.no

Tor Dokken
Sintef Applied Mathematics, Oslo, Norway.
tor.dokken@sintef.no

Abstract

We demonstrate a ray casting type technique for rendering algebraic surfaces using programmable graphics hardware (GPUs). Our approach allows for real-time exploration and manipulation of arbitrary real algebraic surfaces, with no pre-processing step, except that of a change of polynomial basis.

The algorithm is based on the blossoming principle of trivariate Bernstein-Bézier functions over a tetrahedron. By computing the blossom of the function describing the surface with respect to each ray, we obtain the coefficients of a univariate Bernstein polynomial, describing the surface's value along each ray. We then use Bézier subdivision to find the first root of the curve along each ray to display the surface. These computations are performed in parallel for all rays and executed on a GPU.

Overview of our method

For a function $f:\mathbb{R}^3\to\mathbb{R}$, an implicit surface can be defined by the level set of the equation $f(x,y,z)=c$, where $x,y,z,c\in\mathbb{R}$. By reordering the terms, we can, without loss of generality, just consider the zero-set of the function, e.g. $f(x,y,z)-c=0$. For this work, all functions and surfaces are considered to be real-valued. If the function f is an algebraic polynomial, the resulting surface is called an *algebraic surface*. Such surfaces can easily describe intricate, smooth shapes with varying topology, and also allow for the interpolation and blending of such surfaces. Their polynomial nature also makes it easy to find analytic directional derivatives, normals and curvature.

Our algorithm is based on the classical ray casting approach, where rays pass from the eye, and we wish to find the location of each ray's first intersection with the surface. For a given algebraic surface $f(x,y,z)=0$ of total degree d, we are only interested in intersections inside a view volume V. Hence, it is natural to only consider the segment of each ray inside V. Let $\mathbf{f}$ and $\mathbf{b}$ represent the front and back coordinate of the ray as it enters and leaves V. The segment is then given by the parametrization $\mathbf{p}(t)=(1-t)\mathbf{f}+t\mathbf{b}$, $t\in[0,1]$. We combine this parametrization with the equation of the algebraic surface, yielding a univariate polynomial $g(t)=f(\mathbf{p}(t))$ of degree d. Now the problem of finding the closest intersection to the eye point, can be reformulated to finding the smallest $t\in[0,1]$ such that $g(t)=0$.

By using the ray casting formulation, the process of extracting the zero set of one multivariate polynomial, has been replaced by a multitude of independent root finding problems. The GPU is well suited for such parallel computations, and we summarizes our algorithm as follows:

1. Issue *spanning geometry* to the GPU. The spanning geometry defines our view volume V, and initiates rendering.
2. Let the GPU rasterize the spanning geometry into a set of fragments, yielding the coordinates $\mathbf{f}$ and $\mathbf{b}$ for each ray.
3. Find the univariate polynomial, $g(t)$, representing the value of the algebraic surface along each ray.
4. Find the smallest t such that $g(t)=0$. If no zeros are found, we discard the fragment.
5. For the surviving fragments, calculate the final depth and color by using information from the algebraic surface.

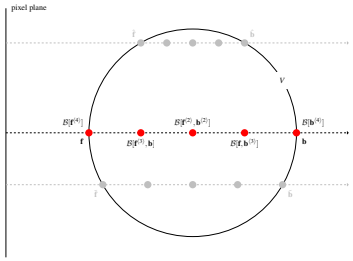

FIGURE 1: For each ray segment inside the view volume V, we find the front (**f**) and back (**b**) vectors in barycentric coordinates. Then we use blossoming to find the coefficients of the univariate Bernstein-Bézier function along each ray.

Related work

Several other works discuss ray casting like algorithms on the GPU, and current interest is high. However, direct hardware support still seems to be a long way of, and most approaches involve a considerable amount of tweaking in order to execute on a GPU. Donnelly [1] uses sphere tracing to effectively implement bounded displacement mapping using fragment shaders. Loop and Blinn [2] has demonstrated rendering of algebraic surfaces, which is very similar to our approach. Their focus has however been on rendering piecewise surfaces spanned over a large number of tetrahedrons. However, they use analytic root finding algorithms, thereby limiting the maximum degree of surfaces they can visualize to general quartic surfaces, as it is not possible to analytically solve general equations of degree 5 and higher.

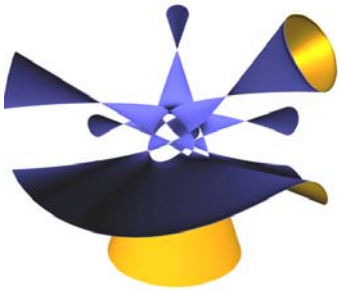

FIGURE 2: A quintic surface, having the maximum possible number of ordinary double points, 31. It is also known as a Dervish surface.

The Bernstein basis

For the space of real polynomials of total degree d, it is well known that they can be expressed in the Bernstein-Bézier (BB) basis, which is preferable for use in computations since it is numerically optimal[3].

The Bernstein basis functions $B^d_{\mathbf{i}}$ of degree d can be expressed as:

$$B^d_{\mathbf{i}}(\boldsymbol{\beta})=\frac{d!}{\mathbf{i}!}\boldsymbol{\beta}^{\mathbf{i}},\quad |\mathbf{i}|=d,\quad |\boldsymbol{\beta}|=1. \tag{1}$$

Here, $\boldsymbol{\beta}=\boldsymbol{\beta}(\mathbf{x})=(\beta_1(\mathbf{x}),\dots,\beta_{s+1}(\mathbf{x}))$ denotes the barycentric coordinate of a point $\mathbf{x}\in\mathbb{R}^s$ with respect to a set of base points, $(\mathbf{v}_1,\dots,\mathbf{v}_{s+1})\in\mathbb{R}^s$, which form a non-degenerate simplex $\Sigma_s=\mathrm{conv}(\mathbf{v}_1,\dots,\mathbf{v}_{s+1})$. For instance, the 2-simplex is a triangle and the 3-simplex a tetrahedron. A polynomial then takes the form $f(\mathbf{x})=\sum_{\mathbf{i}}b_{\mathbf{i}}B^d_{\mathbf{i}}(\boldsymbol{\beta})$.

The de Casteljau algorithm provides a fast and numerically stable way of evaluating a BB-polynomial. Let $b_{\mathbf{i}}$ denote the coefficients of an algebraic function in BB-form and compute repeated convex combinations of the base points by setting:

$$\begin{gathered}b^r_{\mathbf{i}}(\boldsymbol{\beta})=\sum_{j=1}^{s+1}\beta_j b^{r-1}_{\mathbf{i}+\mathbf{e}_j}(\boldsymbol{\beta}),\quad b^0_{\mathbf{i}}(\boldsymbol{\beta})=b_{\mathbf{i}},\\ \mathbf{e}_1=(1,0,\dots,0),\dots,\mathbf{e}_{s+1}=(0,\dots,1).\end{gathered} \tag{2}$$

The de Casteljau algorithm also admits a subdivision formula for Bézier curves, and the control polygon of such a curve converges to the curve itself. In particular, such a curve have no more roots than there are sign changes in the control polygon.

If we modify the de Casteljau algorithm by introducing a sequence of parameter points, $(\boldsymbol{\beta}_1,\dots,\boldsymbol{\beta}_d)$ and use a distinct point at each level of recursion, we arrive at the blossom of the polynomial, which we denote as $\mathcal{B}$. The blossom is given by:

$$\mathcal{B}(f)[\boldsymbol{\beta}_1,\dots,\boldsymbol{\beta}_d]=p^d_{\mathbf{0}},\quad \text{where } p^r_{\mathbf{i}}=\sum_{j=1}^{s+1}\beta_{r_j}p^{r-1}_{\mathbf{i}+\mathbf{e}_j},\quad p^0_{\mathbf{i}}=b_{\mathbf{i}}. \tag{3}$$

The key to finding a curve representing the value of f along the line segment $\mathbf{xy}$ is a surprising property of the blossom, which we summarize in the following lemma:

Lemma 1 *Given two points $\boldsymbol{\alpha}=\boldsymbol{\beta}(\mathbf{x})$ and $\boldsymbol{\gamma}=\boldsymbol{\beta}(\mathbf{y})$, $\mathbf{x},\mathbf{y}\in\Sigma_s$ and a multivariate BB-polynomial $f(\boldsymbol{\beta})$. Then the straight line segment from $\mathbf{x}$ to $\mathbf{y}$ is mapped to a univariate BB-polynomial of degree d, and its coefficients are given by repeated blossoming of the endpoints, $\mathcal{B}(f)[\boldsymbol{\alpha}^{(d)}],\mathcal{B}(f)[\boldsymbol{\alpha}^{(d-1)},\boldsymbol{\gamma}],\dots,\mathcal{B}(f)[\boldsymbol{\gamma}^{(d)}]$.*

To summarize, the blossom provides us with the algorithms needed to convert a trivariate polynomial into a set of univariate polynomials with respect to each ray.

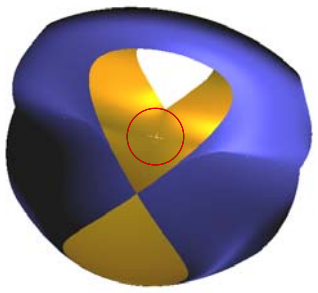

FIGURE 3: The quartic surface $x^4-2x^2y^2+y^4-2x^2z^2-2y^2z^2-16z^2+z^4=0$. The encircled white dots illustrate numerical instabilities.

GPU implementation

To initiate our algorithm, we must send geometry encompassing the barycentric domain to the GPU. Theoretically any convex shape will suffice, but in practice shapes for which it is trivial to calculate the barycentric front and back coordinates for each ray are best suited.

Blossoming The most computationally demanding subtask of the visualization is the calculation of the coefficients of the curve representing the value of f along each ray. We find these coefficients by blossoming and form an algorithm based on (3).

If recursion had been supported by the GPU, we could have implemented (3) directly. However, current generation GPUs do not support recursion, so for a given degree d, we use the CPU to unroll the recursion and generate a long expression of shader source code, which represents the blossom. We use uniform variables to represent the coefficients of the surface, and Since the blossom is recalculated for every frame, this allows us to easily modify the surface by altering the coefficients, allowing for real-time shape manipulation.

Root finding Because of its simple data structure, we decided to use a heuristic root finding method based on recursive subdivision of the coefficients, as described by Schneider [5]. The idea is to recursively subdivide the polynomial in half by using the de Casteljau algorithm (2). At each level of recursion we count the number of sign changes in the coefficients, and if both sets of coefficients contain sign changes we choose to continue to subdivide the leftmost of them,while storing the other one as we might need to unwind the recursion if the first one turns out to be a false zero. If only one set of coefficients has sign changes, we of course choose this one for further subdivision. This effectively uses the Bézier convex hull property to ensure that a zero is within an interval given by the coefficient endpoints.

Since it could happen that there is a false zero we must be able to unwind the recursion and choose another interval to subdivide. On a regular CPU this would be trivial, but as the GPU lacks a stack we store the other potential coefficients and the recursion level, and when we detect a false zero we unroll to the last set of potential coefficients. The number of possible false zeros is bounded by the degree of the surface, so we know in advance how many sets of coefficients we might need to store. We terminate the recursion when we can decide that there is no root (no sign changes of the coefficients), or when we reach a fixed level of subdivisions.

Conclusion and future work

Our work presents a real-time rendering strategy for algebraic surfaces up to total degree five. However, as GPU technologies advance we believe it can be improved further. The two most obvious issues, and thus prime candidates for further work, are to increase rendering speed and the degree of surfaces we can visualize.

Using our current root finding algorithm, we are able to stably isolate roots for polynomials up to degree 10, the limiting factor being that we run out of temporary registers on the GPU. DX10 GPUs will greatly increase the number of such registers.

Our current approach is computationally intensive, and we are currently experimenting with other methods, such as B-spline knot insertion[4] or hybrid methods like Brent-Dekker. However, these methods are very dynamic, and effective GPU implementations are not trivial, and will probably be based on multipass methods.

Our current blossoming algorithm is very poor at utilizing the vectorization possible within each fragment pipeline. The approach of [2] uses tensor contraction, which has a higher computational complexity than our method, but is very effective on a GPU. Efficient blocking of discrete B-spline matrices could be a way to get the best of both worlds, and be the key for effectively rendering high order surfaces.

Acknowledgements

This work was supported by contract number 158911/I30 of The Research Council of Norway.

References

[1] W. Donnelly. *GPU Gems 2*, chapter Per-Pixel Displacement Mapping with Distance Functions, pages 123–136. Addison-Wesley, 2005.

[2] C. Loop and J. Blinn. Real-time GPU rendering of piecewise algebraic surfaces. *ACM Trans. Graph.*, 25(3):664–670, 2006.

[3] T. Lyche and J. M. Pĕna. Optimally stable multivariate bases. *Advances in Computational Mathematics*, 20:149–159, 2004.

[4] K. Mørken and M. Reimers. An unconditionally convergent method for computing zeros of splines and polynomials. *Mathematics of Computation*, To appear.

[5] P. J. Schneider. *Graphics gems*, chapter A Bézier curve-based root-finder, pages 408–415. Academic Press Professional, Inc., San Diego, CA, USA, 1990.