Robust GPU Based 3D Visualization of Algebraic Surfaces

Nancy Newlin[1] , Joseph Masterjohn[2], Victor Milenkovic[2]

1: Undergraduate, University of Florida, 2: Department of Computer Science, University of Miami

Introduction

Algebraic surfaces are used to model shapes in design and manufacturing. We are interested in investigating miniscule surface components and complicated intersections of multiple surfaces. To accurately and efficiently facilitate such investigations, we need to have software that can give us high quality true depictions of each surface. It is important to have the capability to move between surfaces and zoom in on areas of interest.

The current state of the art algebraic surface visualization softwares use triangulation to approximate surface shape. Computation revolves around creating a mesh of the surface using CPU tools. All of the triangle data is transferred to a graphics card. The main issue with this method is the time needed to generate a mesh and ship that data to the graphics card. In addition to being costly, triangulation is only an approximation and therefore overlooks smaller features and incorrectly depicts complex intersections. The resultant image is not guaranteed to be accurate. To be of use for investigating minute and complicated attributes, a software needs to have a high attention to detail while remaining efficient.

The software we pose to achieve such goals takes advantage of new GPU power for a bulk of our computation. The CPU is used to apply a perspective transformation to input trivariate polynomials so the region of interest is contained within a cube. The GPU enables similar processes to run in parallel using CUDA threads. Each pixel for our display has its own thread. Each thread generates a pixel value from the segment with the x and y of the pixel and z ranging from -1 to 1. The pixel's x and y coordinates are substituted into the trivariates to make them univariate in *z*. Along the segment, we find intervals where the univariate polynomials have possible roots using Descartes' rule of sign. Each interval is shrunk to floating-point accuracy by Newton's method and subdivision. At the end of the GPU kernel process, each thread has a set of roots in order of closest to farthest. Each CUDA thread calculates a color value, a blend of each surface's assigned color weighted by the order of its roots along the segment. The thread writes the 24 bit RGB color value to a texture in GPU memory that is displayed to the user.

Two ambiguous situations may arise. One is that two roots of the same univariate cannot be separated, and the other is when roots of different univariates cannot be properly ordered. In both cases, a special ambiguity warning color is used instead of the surface color.

Descartes' rule ensures that we account for every possible root. By using intervals as our primary numerical representation, we are able to guarantee correctness of the roots, and hence image, unless we explicitly show ambiguity. Areas of interest and uncertainty can be explored with extensive zoom.

Problem Statement

Our goal was to create a software that, in an efficient manor, guarantees the visualization is mathematically correct unless uncertainty is explicitly indicated. The software is to function as a debugging tool to test the accuracy of other algorithms that seek intersections of algebraic surfaces.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1659144. Masterjohn and Milenkovic are supported by NSF grant CCF-1526335. University of Miami Computer Science department equipment was used.

UNIVERSITY OF MIAMI
DEPARTMENT of COMPUTER SCIENCE

UF UNIVERSITY of FLORIDA

NSF

Algorithms

Starting on the CPU, input trivariate polynomials are transformed so that the region of interest becomes a cube with bounds with [-1,1] in each direction. Data for each polynomial is sent to the GPU. Every CUDA thread corresponds to an *x* and *y* pixel coordinate. For each trivariate, *x* and *y* are substituted to make a univariate in *z* ranging from -1 to 1. Therefore, every thread has a unique set of univariate polynomials, and roots of these polynomials correspond to intersections of the surfaces with a line segment with the pixel *x,y* and that range of *z* [insert citation to picture]. Each thread moves along that segment in increments of $dz = 2 / n$ where n is the number of steps. It generates intervals [*z, z+dz*] for z = -1 + i *dz where *i* ranges from 0 to *n* - 1. We apply Descartes rule of signs to determine how many roots each interval contains. If there is only one possible root, that interval is reduced through Newton's method and subdivision until the roots is located with floating-point precision. If there is more than one root, the interval is subdivided until there is only one root per interval. Reduced intervals are compared to the intervals from other surfaces at the same pixel. They values are ordered from largest to smallest. If the number of roots in an interval cannot be reduced to one, or if two different root intervals overlap, it is an ambiguous case. Now, we assign a color to this pixel using the order of the roots. Ambiguous roots are assigned a special color. Each CUDA thread calculates a color value, a blend of each surface's assigned color weighted by α^i where *i* is the index of the root sorted along the segment, and α being the opacity value of the contributing color, usually 0.5 uniformly. The thread writes the 24 bit RGB color value to a texture in GPU memory that is attached to an OpenGL frame buffer object (FBO).

Newton's Method

This algorithm starts with an interval that needs to be shrunk down. Descartes has already verified that this interval has only one root. The result is an interval with ε width encompassing the root. Here, ε has a value of 1.0^{-6}. Newton requires an initial guess of the root value, which in our case is the midpoint of the interval, x_m. The derivative, *f'(x)* and function value *f(x)*, are calculated at x_m. We initialize a temporary interval, *tn*, to hold the value resulting from this calculation: tn =*xm* - (f(xm)f'(xm)). We compare tn and the original interval to see if there was any progress made in shrinking. The interval value tn had now becomes the beginning interval. Its midpoint is stored in x_m, and this process continues for every resulting interval until the width of *tn* is less than ε. This interval is the root. When the new interval cannot be shrunk and has not yet reached the desired width, the algorithm returns an empty interval. This signals to the program to use subdivision.

Subdivision

An interval is subdivided for the purpose of isolating a single root. The algorithm deals with three values evaluated on the function: midpoint $f(x_m)$, lower endpoint $f(x_l)$, and higher endpoint $f(x_h)$. The sign of $f(x_l)$, and $f(x_h)$ are compared to $f(x_n)$. If there is a sign change between either endpoint and the midpoint, the algorithm returns a new interval of that endpoint to the midpoint.

Descartes' Rule of Signs

Descartes' rule of signs finds how many roots to expect, but not where they lie. By looking at the coefficients of each term, we note where the signs change. The number of sign changes is the maximum number of possible positive roots. If there are less, it the total can only decrement in pairs of two roots. For univariate f(z) on interval [a,b], we determine the coefficients of the numerator of f(1/(w+(1/b-a)) + a) and count the number of sign changes. To calculate the coefficients, we use repeated synthetic division. This process is done for every interval along the z direction to see if there is a root contained. It is useful in detecting double roots which will have two or more sign changes.

Conclusions and Future Work

After this project, we conclude that modern graphics cards can display multiple algebraic surfaces in real time using this technique.

In the future, we want to enhance the user interface. The software would have an option for automatic guided tour of the ambiguous places. It would iterate through each marked location and zoom in on the issue area. Though regions of uncertainty are labeled, it's possible for a user to accidently overlook them.

The ultimate goal is to have this software be used to find the intersections of algebraic surfaces itself, instead of as a debugging tool for those that do.

Results

To showcase multiple surfaces intersecting at a small region, we modeled 4 spheres with the following equations, the intersection of which takes the shape of a tetrahedron with altitude 0.001:

Yellow $x^2 - 2x + y^2 + 2y + z^2 = 1$
Red $x^2 + 2x + y^2 + 2y + z^2 = 1$
Green $x^2 + y^2 + 2y + z^2 = 1$
Blue $x^2 + y^2 + z^2 - 2z = 3.999$

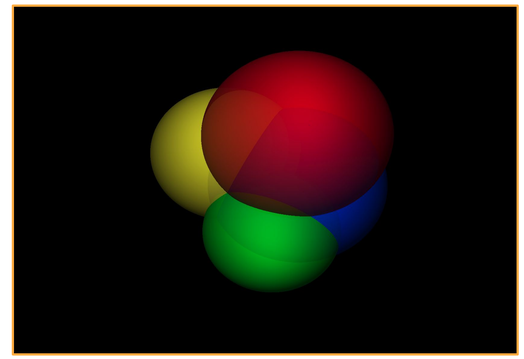

Image 1. The visible region is 5 units in depth, giving a full view of all four surfaces. Through the red surface, the blue and yellow are visible due to blending.

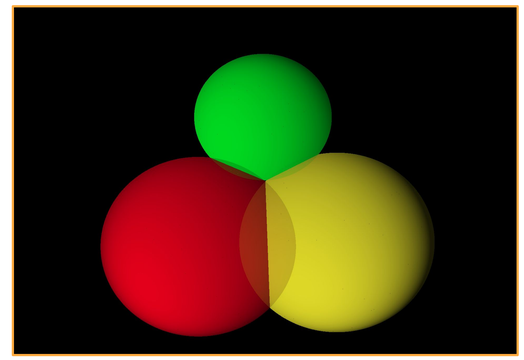

Image 2. A different angle showing the green, red, and yellow spheres. The intersections are shown in the middle where the colors are blended together.

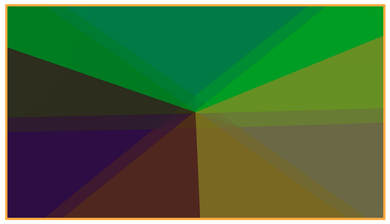

Image 3. The region of intersection of all four spheres. In the center, a tetrahedron is contained within all surfaces. The visible region has depth 0.01 for zoom.

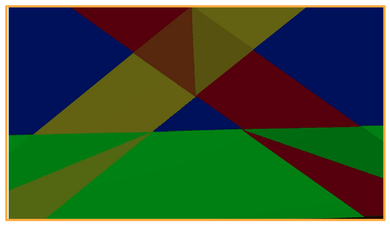

Image 4. Translated inside of the tetrahedron, at the same zoom as image 3.