Real Time Ray Tracing of Implicit Surfaces Utilizing GPGPU Computing

Raymond L. Imber

Department of Computer Science, University of Nevada Las Vegas

Dr. Donald Spickler

Department of Mathematics, Salisbury University

Dr. Enyue Lu

Department of Computer Science, Salisbury University

Summer 2012 NSF REU

Abstract:

Real time ray tracing using the GPU has recently become possible and reasonable, but one of the largest bottlenecks associated with GPU ray tracing is the movement of model data onto the graphics card for processing. Implicit Surfaces can be represented very compactly, which makes them ideal for GPU architecture. Real time ray tracing of general implicit surfaces is widely applicable, but very little work has been done regarding the rendering of general implicit surfaces on the GPU. Singh(2007)4 is one of the only published papers on the subject. Singh uses a pixel shader based approach to implement his algorithm which, while effective, is very limited. I propose two modifications to the research done by Singh. The first is a change of implementation from pixel shaders to a CUDA based implementation. The second is a modification to his algorithm to use a lesser known but much faster converging root finding method known as Ridders’ method. These modifications shuould result in a general increase in the quality and complexity that can be achieved in rendering arbitrary implicit surfaces while maintaining higher than interactive frame rates.

**Problem background introduction:**

Ray tracing is a well-known technique for producing highly realistic computer graphics visualizations, but due to the large computation time required, ray tracing has traditionally not been feasible for real time use. Ray tracing is well suited for the GPU environment because of the technique’s parallel nature. With the advent of CUDA by Nvidia, and related technologies, real time ray tracing using the GPU has recently become possible and reasonable.

One of the largest bottlenecks associated with GPU ray tracing is the movement of model data onto the graphics card for processing. Implicit Surfaces are ideal for the low memory bandwidth / high compute bandwidth limitations of the GPU because implicit surfaces can be described very compactly, requiring only the description of the surface equation to be stored.

**Related work literature survey:**

Implicit procedural geometry has been well studied by the computer graphics community, but the application of such research to GPU based ray tracing is much more limited. The main study on which my research has been based comes from Singh3. Singh proposed a ray marching algorithm matched with numerical root approximation techniques to find the roots of arbitrary implicit surfaces. His method achieved very high rendering speeds on algebraic surfaces such as polynomials up to degree 18, as well as some non-algebraic surfaces. The algorithms implemented by Singh produced extremely good results, but were implemented using shader languages, which are much more limited than CUDA. Shader languages created to main limitations. The first is that there can be no inter-thread communication, which prevented dynamic ray creation or communication between rays. The second is that shaders have very limited stack space which severly limits the level of recursion that can be achieved inside a ray. Singh was able to successfully implement shadow casting, but the limitations of Singh’s implementation didn’t allow him to implement any advanced ray tracing techniques such as reflections, refraction, caustics, or global illumination, all of which require a high degree of recursion.

**Details of the problem you’re tackling:**

Ray Tracing produces images by simulating the path of light rays through a scene. This process is accomplished by starting from the image plane and “tracing” the path of a light ray back into a scene. The process of tracing the ray involves checking if the ray has intersected any object in the scene. Such an intersection implies that the ray was reflected by that object. Color and other light information are then applied to the ray based on the reflective and refractive properties of the object. This process is repeated for a large sampling of rays to produce a final image.

The problem of ray tracing implicit surfaces can be reduced to finding the root of the surface equation with respect to the ray that has been cast onto the surface. The root represents the intersection point of the ray with the surface.

A Surface in http://latex.codecogs.com/png.latex?\mathbb%7bR%7d%5e%7b3%7d can be described implicitly by the set of all points where a function, http://latex.codecogs.com/gif.latex?f(x,y,z)=0. This set is known as a level set, or level surface, of http://latex.codecogs.com/png.latex?f(x,y,z). Checking for an intersection between a ray and a level surface is equivalent to finding a point on the ray that is also a point in the level set for http://latex.codecogs.com/png.latex?f(x,y,z). This can be reduced to finding the root of the function, http://latex.codecogs.com/png.latex?f(x,y,z) ,describing the surface, when projected onto http://latex.codecogs.com/png.latex?r(t)=%20(x,y,z), the parametric function of the ray, which can be described mathematically as, http://latex.codecogs.com/png.latex?f(r(t))%20=%200. If such a value for *t* exists in the positive domain of t, it implies the ray has intersected the level surface.

A useful colliery of this result is the gradient of http://latex.codecogs.com/png.latex?f(x,y,z) at a point is perpendicular to the level set of http://latex.codecogs.com/png.latex?f(x,y,z) at that point. This is the definition of the normal vector at a point for a surface, which is necessary for computing shading information for a surface.

For very specific cases of implicit surfaces, polynomials of degree less than four, the root can be easily solved analytically using well known formulae. For polynomials of degree greater than four, and non-algebraic surfaces, no such formula exists. The existence of such cases begs the need for a method to solve the root for the general case of any surface. Such algorithms are generally classified as numerical approximation methods. There are various algorithms for numerically approximating the root of an arbitrary surface, but most of them require a starting interval that is on either side of the root and relatively close to the root. Singh proposed a two stage algorithm for solving this problem.

The first stage, known as ‘ray marching’, involves evaluating intervals along the ray domain defined by an arbitrarily set ‘marching interval’. Each interval is tested for the existence of a root in the range of the surface equation that lies in the domain interval. Such a test is known as an interval extension test.

The results of this research are based on two interval extension tests. The first is known as the *sign test*. This test checks for a difference in sign between the values of the function at each end point of the interval. A difference in sign implies the existence of a root by the intermediate value theorem. The algorithm is computationally simple and fast, but, it may return a false negative if there are an even number of roots in the interval.

The second interval extension test is known as the Taylor test. The first order Taylor series expansion is derived, centered on each end point of the interval, using the first derivative of the surface equation. The first order Taylor series expansion always produces a line tangent to the centered point. This line is then extend to the mid-point of the interval for each end point. This produces two new points, t1 and t2. The minimum between the first end point and t1, and the maximum between t2 and the second end point are tested for a change in sign. The Taylor test is much more robust than the sign test because it takes into account information about the inside of the interval, but is much more computationally complex as it requires the use of a first derivative calculation.

The second stage of the ray marching algorithm, known as ‘root refinement’, uses the found interval as the initial input points into a traditional numerical root approximation algorithm. There were two numerical approximation methods tested in this research.

The first is the Bisection method. This method successively bisects the interval, using the midpoint as the approximation of the root at each iteration, as well as using the midpoint as one of the end points of the next iteration. This method is very robust as the interval always remains bracketed around the root. This implies that the approximation will always converge to the root, never diverging to infinity, even if the function is not well behaved near the root. It is also fairly computationally simple, but has a relatively slow rate of convergence to the interval, with a convergence number of only 1.

The second method used in this experiment is known as Ridders' method. Ridders' method is relatively new, created in 1979 by C. Ridders2. It is used primarily in the field of numerical analysis, but to the best of my knowledge, as never been used in the application of ray tracing. Ridders' method is as robust as the Bisection method, with the interval always bracketing the root, but has a much faster rate of convergence, with a convergence number of http://latex.codecogs.com/gif.latex?\sqrt%7b2%7d, and in practice is claimed to achieve convergence close to quadratic convergence [Press et al. (2007)3].

Ridders' method method uses the two end points and the midpoint of the interval to transform the function at the three points to a line. This is done by finding the unique exponential function of the form eax which, when multiplied by f, transforms the function at the three points into a straight line. The root of this line is then used as the approximation of the root at each iteration, as well as one of the bracketing values for the interval of the next iteration.

**Experiments and performance comparison/analysis:**

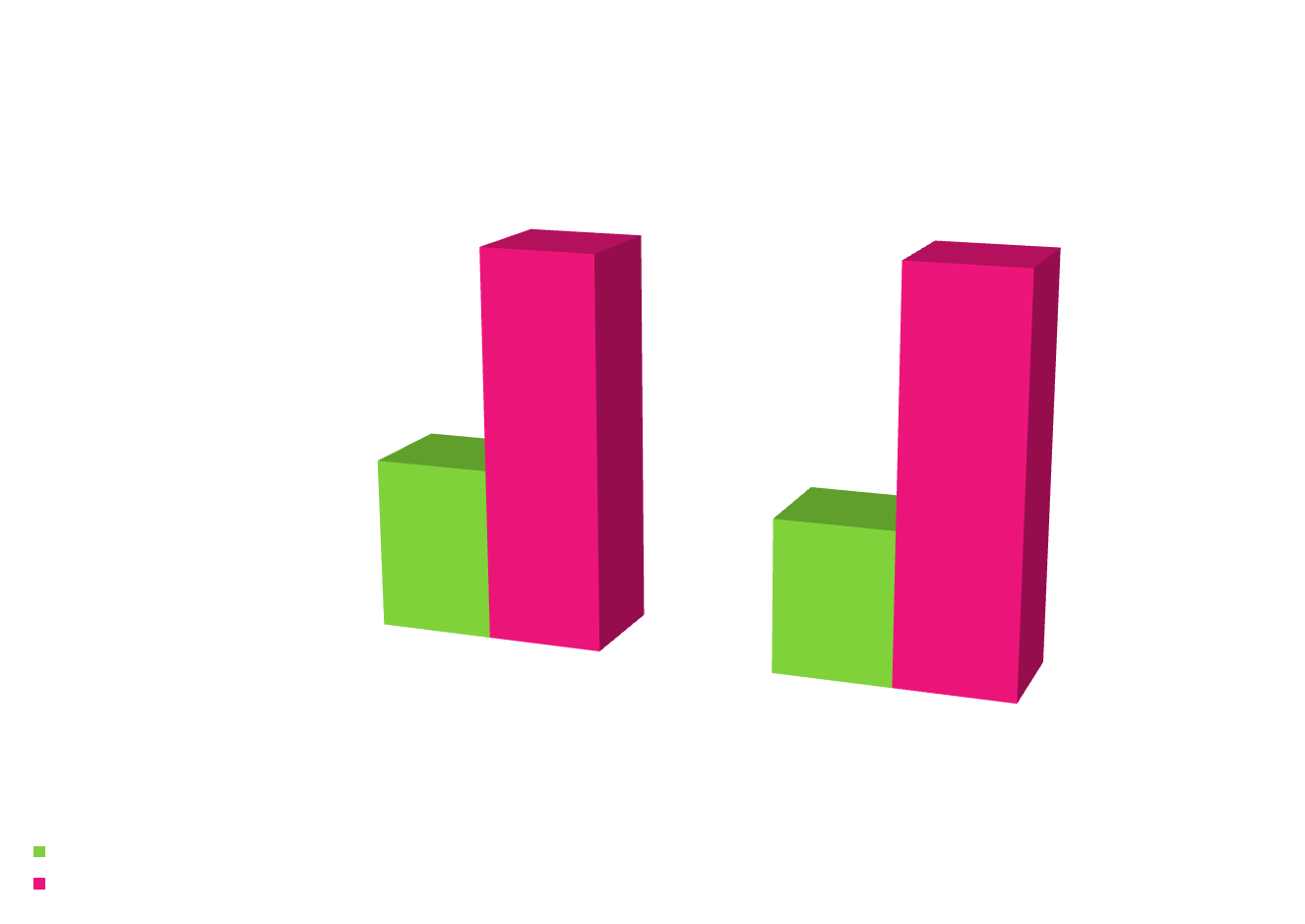
My initial goals were to use the Nvidia Optix ray tracing engine [Parker (2010)1] with Nvidia CUDA to build a ray tracer capable of rendering arbitrary implicitly defined surfaces for measurement and benchmarking. Minimally, this ray tracer must be capable of rendering implicit surfaces with simple Lambert and Phong lighting, as well as shadows and reflection.

The initial experiments using the Optix library proved to be severely disappointing, producing frame rates no greater than 4 Frames Per Second, using the ray marching algorithm as described. This is compared to Singh4, who achieved frame rates on the order of 1000 FPS, with hardware that was five years old at the time of this research. Upon examination of the implementation, I determined that the Optix library was not well suited for this specific form of ray tracing. Optix provides many features optimized for efficiently rendering polygonal surfaces and scenes, but has too much overhead for implicit surface rendering. The Optix library was simply too cumbersome for my application. This result prompted me to re-start the project from scratch, building a custom ray tracer that is optimized for implicit surface rendering.

This second attempt achieved much better results. The initial tests showed frame rates of up to 7000 FPS using analytic methods, and 2000 FPS to 3000 FPS for ray marching, which is on the same order of magnitude as Singh. It should be noted that the implementation was unstable, and would often crash the GPU from too much computational load. The frame rates measured below measure the raw performance of filling a buffer on the GPU, and do not take into account the display of that buffer to an output monitor.

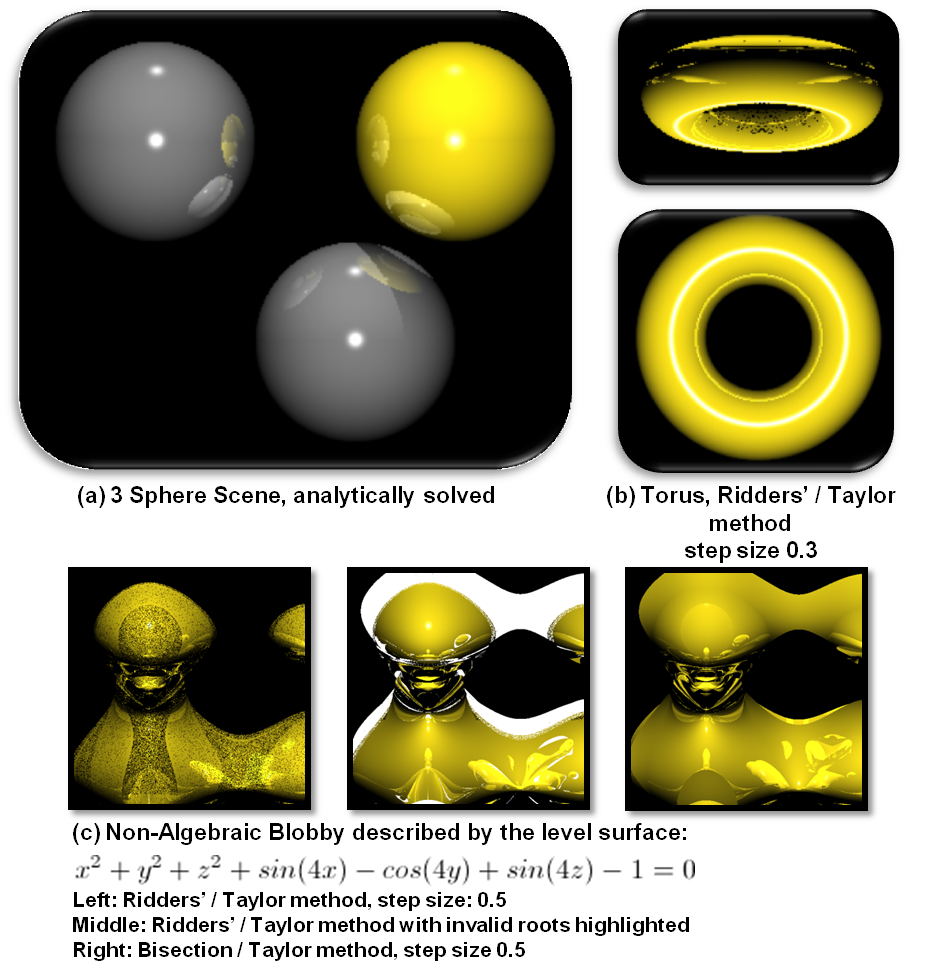
The preliminary experiment was to test how large the impact the transferring of data to and from the GPU would have on performance. Two scenes were ray traced. First a scene with a single implicitly defined sphere, and second a scene with three spheres. A Sphere was chosen because it is defined by a quadratic equation, and thus can be easily solved analytically. This allowed the measure of transfer performance independently of the ray marching algorithm.

Each scene was rendered by two implementations. The first implementation rendered to a GPU buffer and then transferred that buffer to the CPU. OpenGL was then used to render that buffer to the screen. The second implementation set up a GPU buffer that could be shared by both CUDA and openGL, allowing openGL to display the buffer directly to the screen with no intermediary transfer to the CPU. The results were as follows:



The second experiment was to measure the performance of the ray marching algorithm, and attempt to find an optimal configuration for ray marching. All four possible configurations of interval tests and numerical approximation methods were measured. Both average frames per second and standard deviation were measured. Frames per second was interpreted as the raw performance of the algorithm, while standard deviation was used to interpret the stability of the algorithm.

|  |  |
| --- | --- |
| D:\Ray\Documents\wildDoughnut\presentation\graphs\singleSphereAVG.png  D:\Ray\Documents\wildDoughnut\presentation\graphs\torusAVG.png  D:\Ray\Documents\wildDoughnut\presentation\graphs\barthAVG.png  D:\Ray\Documents\wildDoughnut\presentation\graphs\blobbyAVG.png | D:\Ray\Documents\wildDoughnut\presentation\graphs\singlesphereSTDDEV.png  D:\Ray\Documents\wildDoughnut\presentation\graphs\torusSTDDEV.png  D:\Ray\Documents\wildDoughnut\presentation\graphs\BarthicSTDDEV.png  D:\Ray\Documents\wildDoughnut\presentation\graphs\blobbySTDDEV.png |



|  |  |
| --- | --- |
| **Barth Decic as rendered from the ray tracer**D:\Ray\Documents\wildDoughnut\screenShots\BarthDecic.png | **Barth Decic Reference from Wolfram Math World4**D:\Ray\Documents\wildDoughnut\presentation\BarthDecic__example_Wolfram_MathWorld.gif |

It must be noted that one main caveats of the ray marching algorithm is that it is only as accurate as the size of the marching interval. A larger interval step size will produce fewer intervals that need to be tested, and thus be faster to compute, but roots may be missed. A smaller interval step size will be more accurate for the same reason, but will require more computation time. The Barth Decic, a tenth degree polynomial, was not able to be rendered as successfully as the other surfaces. The barth decic required an interval size that was too small for the GPU hardware used to handle without crashing. The images produced indicate the correct surface being rendered, but many roots being missed.

**Conclusion and future work:**

The success of a marching method is highly dependent on the surface being rendered, but certain patterns did begin to emerge. The interval extension test used seemed to have the greatest impact on both performance and stability of the algorithm. The Taylor test consistently produces higher frame rates and lower standard deviation, showing the Taylor test to be both fast and robust for use with the GPU architecture. This is counter to the algorithmic analysis which indicated the Taylor test to be more computationally intensive, and thus less effective. Contrary to the Taylor test, my findings were unable to show any strong correlation of performance or stability for Ridders' method. Ridders' method had very different results depending on the surface being rendered. This is contrary to the algorithmic expectations which say that Ridders' method should produce a lower standard deviation. Ridders' method uses the square root operation as a major component of the algorithm and it is known that the square root operation is not well suited for the GPGPU architecture. I hypothesize that the square root operation is at least partially responsible for the deviation from expected performance.

This result seems to imply that the simplicity of an algorithm is not necessarily the best indicator of how well suited an algorithm is for GPGPU architecture, and there are other factors that may indicate better how well suited an algorithm is for GPGPU architecture. While telling, this is simply a rough hypothesis and requires in depth research far beyond the scope of this project.

Ridders’ method did not influence the speed or stability as much as hypothesized, but visual inspection of the non-algebraic blobby(c), seems to indicate that Ridders’ method produces more accurate roots. While the Bisection method seemed to produce more acceptable roots, as seen by more of the highlighted area being shaded yellow, the roots that were returned by Ridder's method produced more accurate specular reflection, indicating more accurate roots. This should be examined further with more rigorous accuracy tests.

As a possible future accuracy test, I propose outputing a matrix of root approximations instead of a buffer of color values. This matrix could then be compared with analytical methods for surfaces that can be solved analytically, to find an absolute error statistic. For surfaces that cannot be solved analytically a relative error statistic could be found by comparing the results of the different algorithm configurations.

The implementation itself was fairly naïve and the load would often crash the GPU. Optimization of the code should produce greatly improved program stability. The code itself is long and complex, with a large stack memory footprint. The code itself should be written more compactly, to save GPU text memory. The memory footprint should also be minimized by optimizing the pert thread loops. Another possible optimization is to compute the scene in parts between separate GPU kernel calls as opposed to the current implementation of one monolithic kernel call that computes everything at once. This would prevent the GPU from becoming computationally overloaded and crashing, and would not require much extra I/O transfer overhead as the result buffer does not have to leave the GPU between kernel calls. There are two ways in which the kernel could be distributed. The first is by screen space, rendering the scene by "tiles" representing one portion of the image buffer. Since each ray in a scene is independently calculated, such a modification would not affect the resulting output at all. The second way to distribute the calculation could be to distribute the ray marching by z-space. Each kernel call would be responsible for only a small subset of ray marching intervals. The CPU would then call the kernel inside a loop, marching down all rays simultaneously, until all ray intervals have been evaluated. This also would not affect the result as the interval extension tests are independently calculated, and do not require knowledge of any other interval. Both distribution algorithms should be tested to find which one is optimal.

While this research explored different methods for evaluating the ray intervals themselves, Singh explored not just different interval evaluation methods but also schemes for marching down the ray and choosing intervals more efficiently, such as adaptive marching using function distance measurements and silhouette tests. My colleagues and I have also discussed a possible recursive marching scheme in which the intervals could be determined by successively subdividing the ray interval in half. Such marching schemes could have a large impact on the algorithm performance and deserve more in depth study and evaluation.

**References:**

[1] Parker Steven G. (2010). "OptiX: A General Purpose Ray Tracing Engine". ACM Transactions on Graphics, SIGGRAPH 2010 Proceedings, August 2010

[2] Press, WH; Teukolsky, SA; Vetterling, WT; Flannery, BP (2007). "Section 9.2.1. Ridders' Method". Numerical Recipes: The Art of Scientific Computing (3rd ed.). New York: Cambridge University Press. ISBN 978-0-521-88068-8.

[3] Ridders, C. (1979). "A new algorithm for computing a single root of a real continuous function". IEEE Transactions on Circuits and Systems 26: 979–980. doi:10.1109/TCS.1979.1084580

[4] Jag Mohan Singh, P.J. Narayanan, "Real-Time Ray Tracing of Implicit Surfaces on the GPU," IEEE Transactions on Visualization and Computer Graphics, vol. 99, no. RapidPosts, pp. 261-272, , 2009