Ray Tracing Implicit Surfaces

David Rusk
drusk@uvic.ca
University of Victoria

Abstract

A ray tracer for rendering implicit surfaces is developed. It is written in C++ and uses no external libraries. The implementation of the implicit surface ray tracer guarantees ray intersections will be found by placing upper bounds on the rates of change of the implicit function and its derivative following the procedure of Kalra and Barr [Kalra and Barr 1989].

Keywords: ray tracing, implicit surfaces

1 Introduction

Implicit surfaces are a method for modelling objects using an equation where the surface function is set to equal 0 as in equation 1 below. There are two ways commonly used to render these surfaces: polygonization and ray tracing. Polygonization involves breaking down the surface into polygons and rendering them, while ray tracing calculates the values of the pixels in the image one-by-one. This project investigates using the second technique, ray tracing, for rendering implicit surfaces.

The challenge when ray tracing implicit surfaces is reliably finding ray intersections with the surface. The techniques of Kalra and Barr [Kalra and Barr 1989] were used to guarantee the success of this process, as long as upper bounds on the net rate of change of the implicit function and its derivative are known.

Kalra and Barr describe two steps to their method. The first is a space pruning algorithm which narrows down the search space for ray intersections. The second is the ray intersection algorithm itself. These algorithms will be explained in more detail in section 6.

2 Background

As mentioned in the introduction, this project mainly implements the techniques of Kalra and Barr [Kalra and Barr 1989]. However, there have been many other publications in this area, so a brief overview of some of them will be given here.

Mitchell [Mitchell 1990] proposed using interval arithmetic in order to find intersections reliably using two steps: root isolation and then root refinement. Many other authors worked on improving performance of the interval method, such as Capriani et al. [Capriani et al. 2000] and Florez et al. [Florez et al. 2006].

Implicit surfaces have not been heavily utilized despite their advantages (easy blending of surfaces, constructive solid geometry, collision detection), mainly because they are expensive to compute. Therefore, much research has gone into improving computation speeds. deGroot and Wyvill [de Groot and Wyvill 2005] developed an algorithm called Rayskip which exploits object space coherence and temporal coherence to reduce the number of implicit function evaluations while ray tracing, thereby improving rendering times.

Recently real time ray tracing of implicit surfaces has also become an active area of research. Singh and Narayanan [Singh and Narayanan 2010] have developed a GPU-based real time implicit surface ray tracer. Knoll et al. [Knoll et al. 2007] have developed a real time implicit surface ray tracer using SIMD optimized interval arithmetic and coherent ray traversal. They claim it is able to render any programmable implicit function based only on its definition, in real time on the CPU without special hardware. They also developed a similar algorithm for use on the GPU [Knoll et al. 2009].

3 Methodology

This section will give some background information about ray tracing, implicit surfaces, and LG surfaces, which will be useful for understanding the algorithms used in this project.

3.1 Ray Tracing

Rendering three-dimensional objects is a fundamental task in computer graphics. The goal is to take a scene of three-dimensional objects and produce a two-dimensional image showing the objects as viewed from a specific location. In particular, the input is a set of objects and the output is an array of pixels.

There are two common ways to consider how each object contributes to each pixel: object-ordered rendering and image-ordered rendering. In object-ordered rendering iteration is over the objects, and for each object all the influenced pixels are found and updated. In image-ordered rendering iteration is over the pixels, and for each pixel the objects that affect it are found and used to calculate that pixel's value.

Ray tracing is an image-ordered rendering technique. For each pixel it follows these steps:

1. Compute the ray from the camera to the current pixel.
2. Find the first object the ray intersects.
3. Use the intersection point, the object's surface normal at that point, and any lighting, to calculate the pixel value.

For more information about ray tracing, see Shirley et al. [Shirley and Marschner 2009].

3.2 Implicit Surfaces

Implicit surfaces are a method for modelling objects using an equation where the surface function is set to equal 0. For example, 1 describes a sphere.

$$f(x, y, z) = x^2 + y^2 + z^2 - r^2 = 0 \tag{1}$$

In this case the value of the function will be negative inside the sphere, positive outside, and 0 on the boundary.

In order to render the implicit surface, points in space are sampled to see whether they are inside, outside, or on the border. This process is expensive to evaluate, but it has some advantages. For more information, see Shirley et al. [Shirley and Marschner 2009].

3.3 LG Surfaces

Kalra and Barr [Kalra and Barr 1989] point out that it is not possible to guarantee correct intersection of a ray with an arbitrary implicit surface just using evaluation of the implicit function at sampled points in space. This is because if the sampling is not fine grained enough, features that are smaller than the distance between samples will be missed.

Therefore, extra information is needed to guarantee the intersections. Kalra and Barr use Lipschitz constants to put bounds on the net rate of change of the implicit function and its derivative along the direction of the ray.

A Lipschitz constant is a constant that satisfies the inequality in equation 2.

$$|f(x_1) - f(x_2)| < LipschitzConstant * |x_1 - x_2| \quad (2)$$

Let $f(x)$ be the implicit function, and let a ray be defined by its direction α and origin β as in equation 3.

$$r(t) = \alpha * t + \beta \quad (3)$$

Then the implicit function's value along the ray can be defined as in equation 4.

$$F(t) = f(\alpha * t + \beta) \quad (4)$$

The derivative of the original implicit function along the ray direction is then as shown in equation 5.

$$g(t) = dF/dt \quad (5)$$

Then let L be the Lipschitz constant for $f(x)$, and let G be the Lipschitz constant for $g(t)$.

$$|f(x_1) - f(x_2)| < L * |x_1 - x_2| \quad (6)$$

$$|g(t_1) - g(t_2)| < G * |t_1 - t_2| \quad (7)$$

Using this background information, an LG surface can then be defined as an implicit function where L and G exist and can be computed. Kalra and Barr's algorithm guarantees correct ray intersections for LG surfaces.

4 User Interface

The current implementation does not have an interactive graphical user interface because ray tracers generally are not used in real time, though that is an active area of research (see references in the Background section). Instead the application is run from the command line with constant parameters such as image dimensions defined in a special header file. The scene to be rendered is also currently specified directly in the code. This is less than ideal because it requires recompiling the project to change the scene. Some domain specific language should be developed for specifying the scene, which the application can parse and use to build the scene. It should also be possible to pass arguments on the command line to change the different parameters like image dimensions.

5 Data Structures

This section describes some of the data structures and interfaces used to implement the implicit surface ray tracer. Only two of the more fundamental and interesting ones are presented here.

5.1 Implicit Surface Interface

The system has an interface class for implicit surfaces. To add a new type of implicit surface to the system, simply create a subclass which implements the interface's methods. The C++ code for the interface is shown below:

```
class ImplicitSurface
{
public:
    /**
     * Returns a bounding box around this
     * surface.  It does not have to be the
     * minimal bounding box.
     */
    virtual Box BoundingBox() = 0;

    /**
     * Returns the value of the implicit
     * function that defines this surface
     * at the specified point in space.
     */
    virtual double ImplicitFunction(
            Vector3D point) = 0;
    /**
     * Returns the gradient of the
     * implicit function that defines this
     * surface at the specified point in
     * space.
     */
    virtual Vector3D Gradient(
            Vector3D point) = 0;

    /**
     * Returns the gradient of the
     * implicit function in the direction
     * of the ray, at the point t along
     * the ray.
     */
    virtual double DirectionalGradient(
            Ray ray, double t) = 0;

    /**
     * Returns the Lipschitz constant for
     * this surface in the region defined
     * by the minPoint and maxPoint.
     */
    virtual double LipschitzConstant(
            Vector3D minPoint,
            Vector3D maxPoint) = 0;

    /**
     * Returns the Gradient Lipschitz
     * constant for this surface along the
     * specified ray, between ray
     * distances t1 and t2.
     */
    virtual double GradLipschitzConstant(
            Ray ray,
            double t1, double t2) = 0;
};
```

5.2 Octree

The octree is a tree data structure where each node has up to eight children. It is used in the space pruning algorithm described in section 6.1. Each node in the tree represents a volume in space, and child nodes are subsections of this volume. To create child nodes, the current node's volume is split into eight equal pieces. A similar data structure which can be visualized in two dimensions is the quadtree which has four children per node. Figure 1 visualizes the recursive nature of the space subdivision, and how that is translated into a tree structure. The octree works the same way, but in three-dimensions by splitting in the z-axis as well.

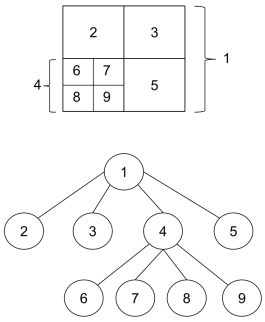

Figure 1: *Example of a quad tree's spacial representation and tree representation.*

The octree structure aids in finding the first box a ray intersects (needed for the ray intersection algorithm given in section 6.2. This is because if it is discovered that the box associated with a node does not intersect with the ray, then all of its children can be ignored as they definitely will not intersect either.

6 Algorithms

This section describes the ray intersection method developed by Kalra and Barr [Kalra and Barr 1989]. They break their method down into two algorithms: space pruning and ray intersection.

6.1 Space Pruning

The space pruning algorithm is run first, and its primary purpose is to improve efficiency. The basic idea is that the algorithm will prune away parts of space that definitely do not contain any piece of the LG surface, and thereby reduce the space that needs to be searched for intersections. This algorithm is not actually needed for correctness, the ray intersection is still guaranteed to be calculated correctly even if no pruning is done.

The algorithm proceeds as follows:

1. Start with a bounding box around the surface. It does not have to be a tight bounding box, it just needs to contain the entire surface.
2. Recursively subdivide the box to some level n.
3. Keep sub-boxes that contain part of the surface, as shown in Figure 2.

This overview of the algorithm leaves out the details of how it is determined whether a sub-box contains part of the surface. This will be explained next.

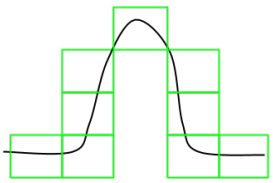

Figure 2: *Sub-boxes containing part of the implicit surface.*

A box is said to straddle the surface if it has at least one corner inside the surface and one outside. Figure 3 shows a few examples of boxes where green ones straddle the surface and red ones do not.

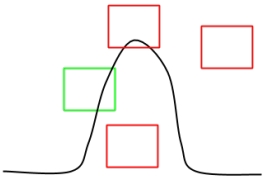

Figure 3: *Straddling boxes are shown in green, non-straddling are shown in red.*

If a box straddles the surface, it definitely contains a part of the surface and is kept. However, even if it does not straddle the surface, it still might contain a portion of the surface, such as the red box at the function peak in Figure 3.

If a box does not straddle the surface, some more calculations have to be performed in order to decide what to do with it. First let us define x_0 to be the center of the box, and d to be the half-length of the principal diagonal of the box, as shown in Figure 4.

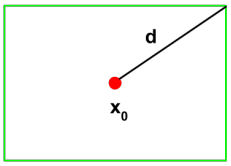

Figure 4: *A box showing the definition of x_0 and d.*

Let L be defined as in equation 6. Essentially it is the maximum rate of change of the implicit function $f(x)$.

$$|f(x_0)| > L * d \tag{8}$$

If equation 8 is satisfied, then $f(x)$ must stay the same sign and never assume a value of 0. This means there is no zero-crossing (the implicit function evaluates to 0 on its surface). Therefore the box definitely does not contain part of the surface and can be pruned.

If equation 8 is not satisfied, then the box is subdivided into eight, and each sub-box is checked.

The end result of the space pruning is an octree data structure (see section 5.2) with the boxes completely enclosing the implicit surface.

6.2 Ray Intersection

The second algorithm from Kalra and Barr is the ray intersection algorithm itself. From the space pruning algorithm, a set of boxes that completely encloses the surface have already been found. Therefore all intersections of rays and the surface will occur in the volume of this set of boxes.

The basic idea is to find the closest box to the ray origin (along the direction of the ray), then compute the intersection in this box. If there is no intersection, the next nearest box is checked. Finding the closest box is aided by keeping the boxes in an octree as discussed in section 5.2.

Let t_1 and t_2 be two points along the ray (see definition of ray in equation 3). Then let $t_m = (t_1 + t_2)/2$ and $d = (t_2 - t_1)/2$. F is as defined in equation 4 and $g(t)$ is defined as in equation 5. Recall that G is the Lipschitz constant for $g(t)$, as defined in equation 7.

$$|g(t_m)| > G * d \tag{9}$$

If $F(t_1)$ and $F(t_2)$ have opposite signs and the constraint in equation 9 is met, then there is exactly one intersection between t_1 and t_2 and it can be calculated with a numerical method such as Newton's method.

If $F(t_1)$ and $F(t_2)$ have the same sign and the constraint in equation 9 is met, then there is no intersection between t_1 and t_2. Therefore the next box is considered, and if there are no more boxes, it can be concluded that the ray does not hit a surface.

There is one more case to consider: when the constraint in equation 9 is not met. Then the interval is subdivided into intervals $[t_1, t_m]$ and $[t_m, t_2]$, and they are tested. This condition can occur when there is more than one intersection in the box.

Subdividing stops when the interval distance d falls below the tolerance of the numerical method used to compute the intersection.

7 Implementation

There are many open source ray tracers available on the internet. However, for pedagogical reasons it was decided to implement a simple ray tracer that did not depend on external libraries. This also made it easier to modify and adapt to support implicit surfaces because many of the open source ray tracers are part of larger packages or have a lot of additional functionality which makes them complicated to use.

The implementation was done in C++. Output images were written to PPM files because the PPM format is very simple and can be implemented trivially without external libraries.

The project source code is available online at https://github.com/drusk/ImplicitRayTracer

8 Results

The system developed can currently render spheres with adjustable locations, radii, colours, reflectivity and transparency. Figure 5 shows a simple example. The ground is made using a sphere with a very large radius. The light source (not in the field of view) is also a sphere. Except for the ground sphere, all of the spheres are reflective. Only the green sphere has some transparency.

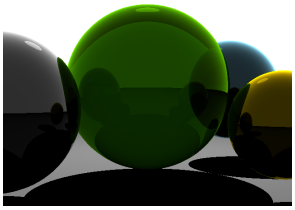

Figure 5: *Example of ray traced spheres.*

9 Conclusion

An implicit surface ray tracer was developed from scratch with no external dependencies. It uses the techniques of Kalra and Barr [Kalra and Barr 1989] to find guaranteed ray intersections with the implicit surfaces. This is done by using two values: "L" which is an upper limit on the rate of change of the implicit function, and "G" which is an upper limit on the rate of change of the gradient. This is significant because it opens up ray tracing as an alternative to polygonization when rendering implicit surfaces.

10 Future Work

A major area of future work is calculating the Lipschitz constants L and G for other useful implicit functions. Kalra and Barr [Kalra and Barr 1989] mention a forthcoming report with additional Lipschitz constants, but to our knowledge no report was ever published. This will allow new types of surfaces to be rendered, and greatly increase the utility of implicit surfaces for modelling when using ray tracing.

Future development of the application could add support for blending surfaces which are close to each other. A simple two-dimensional example of this concept is shown in Figure 6.

Figure 6: *Two-dimensional blend of two circles.*

Adding constructive solid geometry (CSG) operations would also be useful for modelling more advanced structures. CSG combines primitives like spheres and cylinders by using the boolean set operations union, intersection and difference.

References

CAPRIANI, O., HVIDEGAARD, L., MORTENSEN, M., AND SCHNEIDER, T. 2000. Robust and efficient ray intersection of implicit surfaces. *Reliable Computing*, 9–21.

DE GROOT, E., AND WYVILL, B. 2005. Rayskip: Faster ray tracing of implicit surface animations. In *Proceedings of the 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, ACM, New York, NY, USA, GRAPHITE '05, 31–36.

FLOREZ, J., SBERT, M., SAINZ, M., AND VEH, J. 2006. Improving the interval ray tracing of implicit surfaces. In *Advances in*

Computer Graphics, T. Nishita, Q. Peng, and H.-P. Seidel, Eds., vol. 4035 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 655–664.

KALRA, D., AND BARR, A. H. 1989. Guaranteed ray intersections with implicit surfaces. *SIGGRAPH Comput. Graph. 23*, 3 (July), 297–306.

KNOLL, A., HIJAZI, Y., HANSEN, C., WALD, I., AND HAGEN, H. 2007. Interactive ray tracing of arbitrary implicits with simd interval arithmetic. In *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, 11–18.

KNOLL, A., HIJAZI, Y., KENSLER, A., SCHOTT, M., HANSEN, C., AND HAGEN, H. 2009. Fast ray tracing of arbitrary implicit surfaces with interval and affine arithmetic. *Computer Graphics Forum 28*, 1, 26–40.

MITCHELL, D. 1990. Robust ray intersection with interval analysis. *Proceedings on graphics interface*.

SHIRLEY, P., AND MARSCHNER, S. 2009. *Fundamentals of Computer Graphics*, 3rd ed. A. K. Peters, Ltd., Natick, MA, USA.

SINGH, J., AND NARAYANAN, P. J. 2010. Real-time ray tracing of implicit surfaces on the gpu. *Visualization and Computer Graphics, IEEE Transactions on 16*, 2 (March), 261–272.