Ray Tracing Spline Surfaces With Motion Blur

John W. Peterson[*]

September 10, 1998

Abstract

An algorithm for ray tracing B-spline surfaces is described. It is based on preprocessing the surface with adaptive subdivision to build a polygonal representation of the surface, and hierarchical bounding volumes around the polygons. This allows for very efficient ray surface intersections, and avoids complex numerical methods. The algorithm is extended to allow for motion blur without sacrificing efficiency. An adaptive anti-aliasing method is also presented.

Categories and Subject Descriptors: **I**.3.7 **Computing Methodologies**: Computer Graphics— picture/image generation, three-dimensional graphics and realism, computational geometry and object modeling

General Terms: Algorithms

Key Words and Phrases: Image synthesis, ray tracing, raster graphics, motion blur, anti-aliasing

1 Introduction

Over the past several years ray tracing has become the predominant method for generating high quality images of three dimensional scenes. Since the introduction of an elegant recursive

[*]Department of Computer Science, University of Utah, Salt Lake City

shading model by Whitted [24], a large amount of subsequent research has made it a very powerful technique.

The fundamental operation of ray tracing is computing the intersection of a ray with an object. This operation usually consumes most of the time during a ray tracing operation. While this operation is relatively simple for surfaces defined by polygons or simple geometric primatives [19] the calculations for free-form surfaces has proven much more complex.

This paper presents a new method for ray tracing free-form objects. The surfaces rendered by this method are non-uniform, rational B-spline surfaces. By using *non-uniform* surfaces, it is not necessary to break down surfaces into individual polynomial patches for rendering, and the continuity and curvature changes may be exactly specified. *Rational* surfaces (i.e., using a fourth homogeneous coordinate) allow solids derived from conic surfaces (spheres, cones, cylinders, etc) to be specified exactly. There is also no restriction on the order of the polynomials defining the surface. Alternate surface representations such as Bezier or Hermite patches are easily converted to B-spline form [2].

The paper gives a brief review of other techniques for ray tracing splines, and describes a method based on subdividing the surface into a polygonal mesh approximating the surface. Section 4 shows how this technique is extended to allow for motion blur in a direct manner. Finally, an adaptive anti-aliasing method is presented that significantly reduces the time required to produce anti-aliased images.

2 Ray Tracing B-splines

Most previously published approaches to ray tracing free-form surfaces have been based on numerical methods. Kajiya [9] uses ideas from algebraic geometry to obtain a numerical procedure for finding the intersection of a ray and a patch. Toth [23] uses techniques from interval analysis to find a starting guess for Newton's method. A bounding box (the "interval extent") is formed about the surface. If it can be determined that Newton iteration for a ray-patch intersection converges within an interval, the algorithm tries to solve for the intersection. Otherwise, the interval is subdivided and the procedure repeats.

Joy and Bhetanabholta [7] use a combination of heuristics and numerical methods. The numerical techniques are based on using Quasi-Newton methods to find a local minimum for the distance between a ray and the patch (the ray intersects when it reaches zero). The heuristic portion of the algorithm uses ray coherence — the starting guess of a given ray is based on the intersection point of the previous ray to intersect a given surface. Joy uses octree cells to restrict the search space of ray-patch intersections. The cells are also used to help identify cases where the heuristics may fail.

Sweeney and Bartels [21] also implemented a ray tracing method based on Newton iteration. In their scheme, a hierarchical tree of bounding boxes is constructed around a surface. The bounding boxes are based on the surface's control mesh after it has been refined using the Oslo algorithm. The leaf nodes of this tree of bounding boxes contain a starting guess for using Newton iteration to find an intersection with that region of the patch. To intersect a ray with a patch, the ray traverses the tree of bounding volumes using a method presented in [8]. If a leaf node is reached, the intersection procedure uses Newton iteration to try and find the actual intersection point. Another implementation of Sweeney's algorithm that handles a more general form of B-Splines is presented in [14].

There are several problems with numerical techniques. The numerical algorithms are complex, and often require a substantial amount of computation for each ray-surface intersection test. They are prone to stability problems and erratic behavior with some special cases. They are often based on a number of "tolerance" values that must be occasionally adjusted for different models or rendering environments. The complexity of the algorithms also makes implementation in hardware very difficult. The method presented here avoids these problems. It is straightforward to implement and numerically stable.

3 Improved method

3.1 Overview

The new technique presented here is based on subdividing the surface using the Oslo algorithm. Subdivision is performed until the control points of the surface approximate the surface within a resolution determined by the surface's appearance on the screen. As the surface is subdivided a tree is built with polygons at the leaves. This tree is used to form a set of hierarchical bounding volumes around the surface. Ray-surface intersection is performed by intersecting the ray with the bounding volumes (as in [10]) and then testing the ray against the triangle at the leaves. The method is similar to the method used by Lane and Carpenter for scanline rendering [12], except the surface is fully subdivided as a pre-processing step before ray tracing begins.

3.2 Rational, non-uniform B-splines

A non-uniform rational B-spline surface is defined in polynomial form as:

$$\mathbf{F}(u,v) = \frac{\sum_{i=0}^{m}\sum_{j=0}^{n} \mathbf{P}_{i,j} B_{i,k}(u) B_{j,l}(v)}{\sum_{i=0}^{m}\sum_{j=0}^{n} w_{i,j} B_{i,k}(u) B_{j,l}(v)}$$

where $\mathbf{P}_{i,j}$ are an $n \times m$ array of control points for the surface and $w_{i,j}$ are the weighting factors (homogeneous coordinates) for defining rational surfaces. The $B_{i,k}(u)$ and $B_{j,l}(v)$ are the B-spline basis functions, completely defined by the orders k and l (respectively) and knot vectors $\{u_p\}_{p=1}^{n+k}$ and $\{v_q\}_{q=1}^{m+l}$ (respectively). A more complete treatment of these surfaces may be found in [22, 2, 1].

3.3 Surface Subdivision

The subdivision procedure first tests to see if the surface is flat by testing the linearity of the edges and curves in the u and v directions. "Twisted" surfaces are detected by making sure the four corners lie in the plane. If a surface is not sufficiently flat, it is then subdivided. The subdivision is performed by adding a multiple knot with multiplicity k (if subdividing in the u direction) or l (if subdividing in the v direction) in the center of the knot vector. The surface is

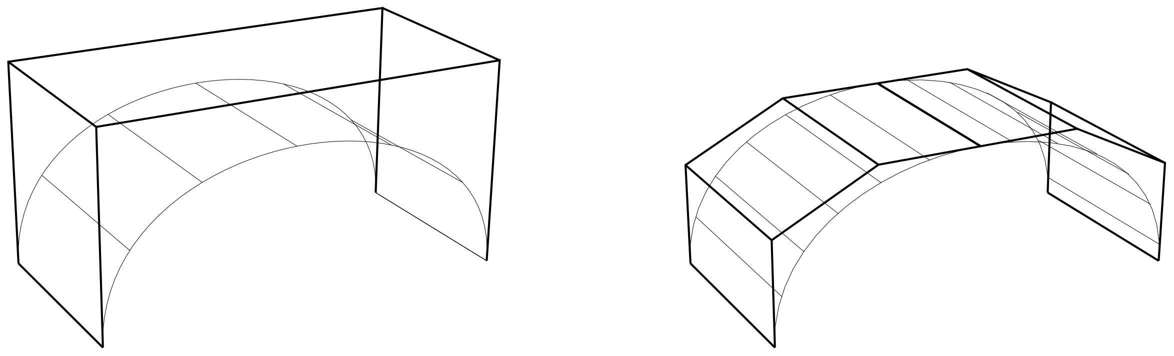

Figure 1: Subdividing a surface by inserting a multiple knot in the center

refined with the Oslo algorithm [4] to add additional control points to the surface corresponding to the extra knots. The refined surface now has two control polygons interpolating the surface along the centerline, and can be separated into two new control meshes which together represent the original surface (figure 1).

If the surface is flat in one direction but not another (e.g., a cylinder) then the surface is subdivided only in the direction that is curved. Otherwise, the parameter the surface is split along alternates directions at each level of subdivision. If a surface is flat enough in both directions, two triangles are formed from the four corner points, and they are converted to Euclidean space by dividing out the rational (homogeneous) coordinate. Figure 2 shows the polygonal representation of a dart generated by the subdivision procedure.

An important detail in surface subdivision is "crack prevention." If one surface is subdivided more than another one, cracks can form in the final polygon mesh if it is derived directly from the control points. This is prevented by checking to see if a surface is adjacent to a straight edge (one meeting the flatness criterion) before subdividing it. If it is, the midpoint of the straight edge is used instead of the control point, preventing cracks from appearing (see figure 3.)

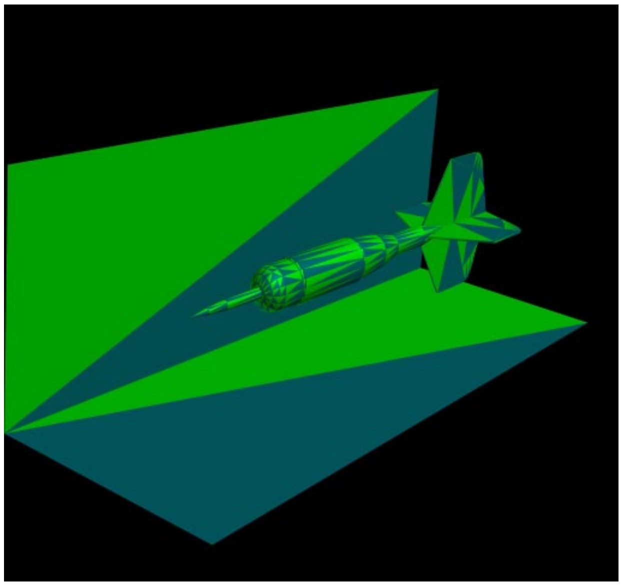
Figure 2: Polygons generated by recursive subdivision

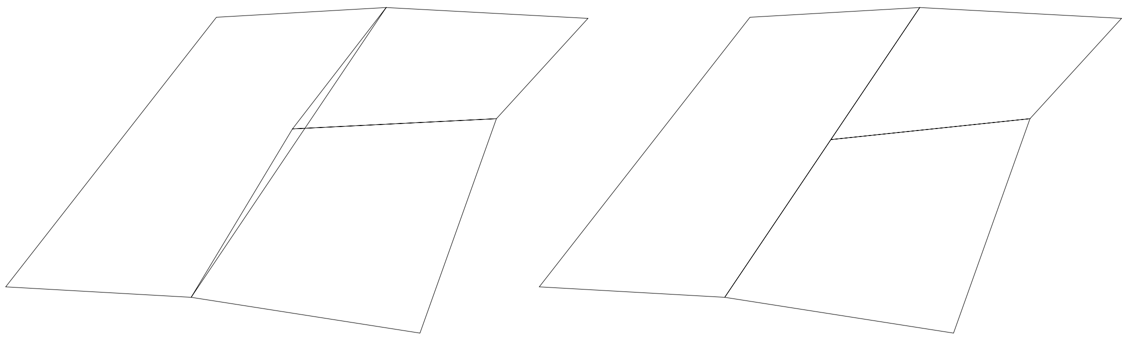
Figure 3: *a.* A crack resulting from subdivision, *b.* Crack avoided by interpolating the edge

In the Lane and Carpenter algorithm, the flatness criterion compares the flatness of the surface to the size of one raster unit in screen space. In the case of ray tracing however, the subdivision resolution must be determined in object space, since no perspective transformation is done on the objects. In order to find the flatness criterion, the projection from object space to screen space must be done explicitly.

Because of the convex hull property of B-splines [1], a reasonable approximation is to first find a bounding box for the surface by finding the minimum and maximum values of its control points. The center point of this bounding box is found, and the distance from it to the eyepoint is calculated. The object space resolution then becomes:

$$subdiv\_res = (2 * viewsize * (-dist + viewdist)/(viewdist * number\_of\_pixels)$$

where *dist* is the distance from the center of the surface to the eyepoint, *viewsize* is the "radius" of the image plane, *viewdist* is the distance from the origin of the initial rays to the viewing plane, and *number_of_pixels* is the size of the final image raster.

While subdivision with the Oslo algorithm requires more computation than the methods presented in [3, 18, 12], it is only done once as a preprocessing step, which is minor compared to the time spent on ray tracing the scene. It also allows for much more general surface definitions.

In order to generate the bounding volumes for the ray tracing process, a tree is constructed as the surface is subdivided. The root node is created with the original surface, and two children are created every time the surface is split. This continues until the subdivision generates two triangles (i.e., the surface was flat enough) which become leaf nodes of the tree.

Once the tree is built, a set of hierarchical bounding boxes is built around the subdivision tree. A bounding box is formed by finding the maximum and minimum extents of each pair of triangles at the leaf nodes. The bounding boxes are grouped hierarchically by finding the union of each successive pair of nodes up the tree (figure 4). These bounding boxes are used for finding the ray surface intersection by using a traversal algorithm such as [10].

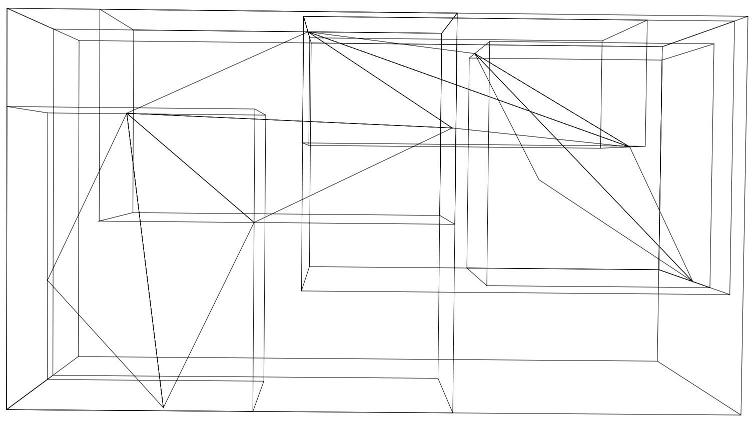

Figure 4: Hierarchical bounding volumes constructed around a refined surface

3.4 Ray-triangle intersection

During the ray tracing process, a ray traverses the hierarchy generated by the subdivision processes. If it reaches a leaf node containing triangles, the following procedure is used to see if a ray intersects them.

In order to perform ray-triangle intersections, barycentric coordinates are used on each of the triangles [2]. This introduces two coordinates r and s along the two sides of the triangle that map its sides onto half of a unit square (see figure 5). For symmetry a third coordinate is introduced, $t = 1 - (r + s)$. These coordinates have the property that:

$$1 = r + s + t$$

Barycentric coordinates are also used for interpolating normals and texture mapping parameters.

In order to easily convert from a point in three space in the same plane as the triangle to the barycentric coordinates of the triangle, a system of equations is computed for each triangle:

$$\begin{bmatrix} V_{0x} & V_{1x} & V_{2x} \\ V_{0y} & V_{1y} & V_{2y} \\ V_{0z} & V_{1z} & V_{2z} \end{bmatrix} \begin{bmatrix} r \\ s \\ t \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

where $\mathbf{V}$ is the vertices of the triangle, and $\mathbf{p}$ is the point where the ray intersects the plane

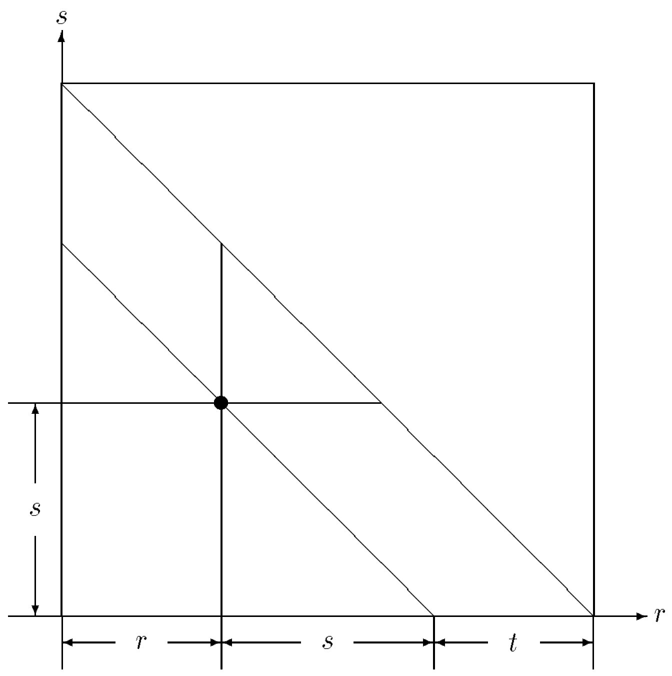

Figure 5: Barycentric coordinates

of the triangle. If we simply use the inverse of $\mathbf{V}$ for calculating r, s, and t from a given point $\mathbf{p}$, problems arise because the inverse of the matrix becomes unstable (degenerate) when the triangle lies on or near one of the coordinate planes. In order to avoid this, a row of one's is added to the bottom row of the matrix and the point in three space (representing $1r + 1s + 1t = 1$). Now the computation becomes:

$$\begin{bmatrix} V_{0x} & V_{1x} & V_{2x} \\ V_{0y} & V_{1y} & V_{2y} \\ V_{0z} & V_{1z} & V_{2z} \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} r \\ s \\ t \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

With this method, we find a matrix $\mathbf{Q}$ with a least squares solution[20]:

$$\mathbf{Q} = (\mathbf{V}^{\mathrm{T}}\mathbf{V})^{-1}\mathbf{V}^{\mathrm{T}}$$

This method is stable independent of the triangle's orientation. Since the t coordinate is easily determined from r and s, only the first two rows of the $\mathbf{Q}$ matrix need to be computed and stored for each triangle.

To test if a ray intersects a triangle, we first find the point, $\mathbf{p}$, where the ray crosses the plane the triangle is in. r and s are found with:

$$\mathbf{pQ} = \begin{bmatrix} r \\ s \end{bmatrix}$$

and

$$t = 1 - (r + s)$$

If any of r, s or t are outside the range $0 \geq x > 1$, then the ray misses the triangle. If an intersection does occur, then r, s and t may be used for linear interpolation of the normals and surface parameters (u, v) for texture mapping:

$$\mathbf{N}_{interp} = r\mathbf{N}_0 + s\mathbf{N}_1 + t\mathbf{N}_2 = \begin{bmatrix} N_{0x} & N_{1x} & N_{2x} \\ N_{0y} & N_{1y} & N_{2y} \\ N_{0z} & N_{1z} & N_{2z} \end{bmatrix} \begin{bmatrix} r \\ s \\ t \end{bmatrix}$$

where $\mathbf{N}_i$ is the normal at each vertex.

In the next section we show how this algorithm is modified to accommodate for motion blur.

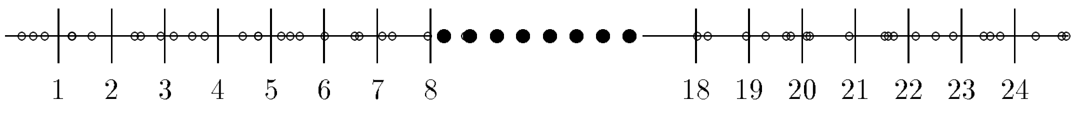

Figure 6: Pre–sampled times (three samples per time segment)

4 Motion blur

Motion blur is a means of removing temporal aliasing from animated sequences [11]. It also provides a unique way of showing the dynamics of a moving system in a single image.

The algorithm presented in [6, 5] is a conceptually simple technique for anti-aliasing called distributed ray tracing. Anti-aliasing is performed for each pixel by taking a number of samples that are distributed spatially (for spatial anti-aliasing) and at different points in time (for motion blur). Distributed ray tracing also provides for penumbra, diffuse reflections and depth of field blur, although these issues are not addressed here.

4.1 Rendering moving surfaces

Motion blur is complicated because the objects need to be sampled as they move to different positions in space, yet we don't wish to lose the bounding box information developed above — moving a large tree of polygons for each ray would be prohibitively expensive. The solution is to "move" the rays to the objects instead of moving objects to the rays.

In order to accomplish this the time for an image is pre-sampled. For example, if we are to use 25 samples for a given pixel (i.e., a 5×5 sampling grid), we would divide the frame time into 25 even segments. To provide for stochastic sampling, a small number of random times are selected from each segment (e.g., 25 segments $\times$ 3 samples per segment $\Rightarrow$ 75 samples) (figure 6).

For each moving surface, a *blur matrix* and its inverse are pre-computed for each of the sampled times. The surface itself (and the hierarchy of bounding volumes around it) is defined in a canonical space (e.g., about the origin). The top level node in the tree describing a moving surface has an array of blur matrices transforming it into the proper position for each pre-sampled time. To intersect a ray with a moving object, we first transform the ray by the inverse blur matrix for the time the ray was fired. Once transformed, the ray proceeds down the tree looking for an intersection as discussed above.

If the ray intersects the object, the intersection point and surface normal are transformed back into the original scene space, using the original blur matrix. If this intersection point is the closest to the ray's origin, it is used for subsequent illumination calculations.

If a single moving object is modeled with a group of surfaces, then these surfaces can be enclosed in a single bounding box, and only one set of blur matrices is needed for the whole object. If a moving object contains moving components, the situation is more complex. For example, consider a moving airplane with a spinning propeller. In order to compute a bounding box for the entire plane, we need to first find the union of the propeller's bounding boxes at each of the time samples for the image (these bounding boxes would be computed in the coordinate system for the airplane). Once the union for these boxes is found, a bounding volume for both the airplane and the propeller can be computed. A ray intersecting the propeller would be bent twice — first into the airplane's coordinate system, and again into the propeller's coordinate system.

4.2 Temporal aliasing issues

Aliasing can be caused by correlations between an object's direction of travel and the spatial position of the time samples within the pixel. Because of the limited number of time samples used for a given image, care must be taken to avoid artifacts from making the time samples at the same positions in the pixel. Cook recommends using a precomputed square of sample indices, with the time segments chosen according to a pre-computed grid (figure 7). The time

7	11	3	14
4	15	13	9
16	1	8	12
6	10	5	2

Figure 7: Sample time distribution square (from [5]).

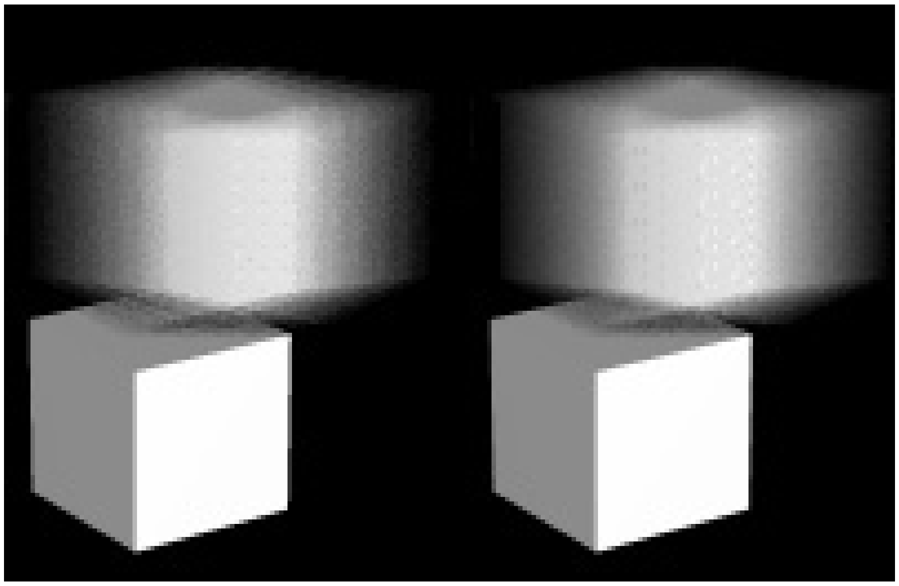

Figure 8: *a.* Motion blur using the same time sample square origin. *b.* Motion blur using random time sample square origin.

segment chosen for each sample is based on the number in the square corresponding to the sample's spatial position within the pixel.

Even with such a distribution method patterns may still be visible (figure 8a). In order to avoid this, the time segments are by selected by choosing a random origin for the sample index square, and letting the indices "wrap around." Figures computed this way are free from the artifacts (figure 8b).

5 Adaptive super–sampling

Super-sampling an entire image is usually too expensive, so various techniques are used to perform super-sampling only where necessary. The first, presented in Whitted's original paper [24] looks at the values at the corners of the pixels. If they differ by a specified tolerance, then the pixel is subdivided, and the corners of each of these sub-pixels is sampled and compared again. The process continues until the difference between adjacent samples falls below a given tolerance or the total number of samples for a given pixel exceeds a pre-set limit. This algorithm does not work well with motion blur, since the initial rays can easily miss moving objects.

Lee, Redner and Uselton [13] present a scheme based on statistical theory for determining the number of samples needed for a given pixel. After an initial set of samples is cast, it determines if additional ones are necessary by analyzing the statistical variance of these extra samples. When the statistical "confidence estimate" of the returned samples falls below a preset tolerance, the method quits sampling.

Experience with this this algorithm, however, indicates two problems with this approach. First, for the statistical measurements to be valid, a non-trivial number of initial samples must be made before the statistical analysis can produce useful results. For example, Lee et. al. used eight initial samples per pixel. However, in areas of the image with low spatial frequences, only one sample per pixel is usually sufficient. Second, the statistical analysis reaches only two conclusions for a large majority of the pixels: Either the minimum number of samples was sufficient, or the maximum number of samples allowed was required.

5.1 The pixel thresher

Since Lee's method tends to require more samples than necessary to obtain an acceptable image, a simpler scheme was developed for performing adaptive anti-aliasing. The method is an extension of Whitted's scheme of comparing adjacent pixel intensities. If a pixel has a value significantly different from its neighbors, it is super-sampled (if it hasn't been already). Since the act of super-sampling the pixel may change its value, the values of surrounding pixels are also checked, and super-sampled if they are different. These changes are propagated across previously sampled scanlines as well as adjacent pixels.

In order to keep track of the comparisons, a data structure called the *thresh buffer* is kept for several of the most recently rendered scanlines. The thresh buffer contains the following fields for each pixel:

color The color of the pixel (e.g., the red, green and blue values).

coverage The coverage of the pixel, determined by how many rays fired from this pixel actually intersected an object in the scene. This is used for compositing ray traced images on other backgrounds (see [16, 15]).

ss_flag A boolean flag, set to true if the pixel has been super-sampled.

changed A boolean flag, set to true if the pixel has changed value (either an initial sample, or because of super sampling).

The algorithm is shown in figure 9.

The *pixel_diff* routine compares the current pixel to its neighbors. It only compares the color and coverage values if the pixel has not been super sampled (*ss_flag* false) and at least one of the two has changed (*changed* is set), otherwise the test is redundant. This heuristic works because pixels in need of super-sampling are most often adjacent to each other (e.g., edges of objects).

By varying the times of the initial rays, this approach works particularly well with moving objects. If all of the initial rays were cast at the same time, the object would only be sampled at that point along its path. In order to avoid this, a sample time grid (much like the one used within a single pixel in section 4.2) is used for determining the time of the initial rays. When one initial ray intersects the object as it moves along its path, the pixel thresher finds this initial sample different from the surrounding ones, and eventually super-samples the entire path of the object.

```
{ Add an initial row of samples to the thresh buffer }
for each scanline in the image
    for each pixel on the current scanline do
        get an initial sample
        ss_flag ← false
        changed ← true

    do
        for each row in the thresh buffer do
        pixels_changed ← false
            for each pixel in current row do
                if not ss_flag and
                   ( pixel_diff( above ) or
                     pixel_diff( below ) or
                     pixel_diff( left ) or
                     pixel_diff( right ) )
                then
                   super sample current pixel
                   ss_flag ← true
                   changed ← true
                   pixels_changed ← true
    while pixels_changed

    Output oldest row in thresh buffer

    { Clear changed flags before adding next row }
    for each row in the thresh buffer do
        for each pixel in current row do
            changed ← false
```

Figure 9: The pixel thresher algorithm

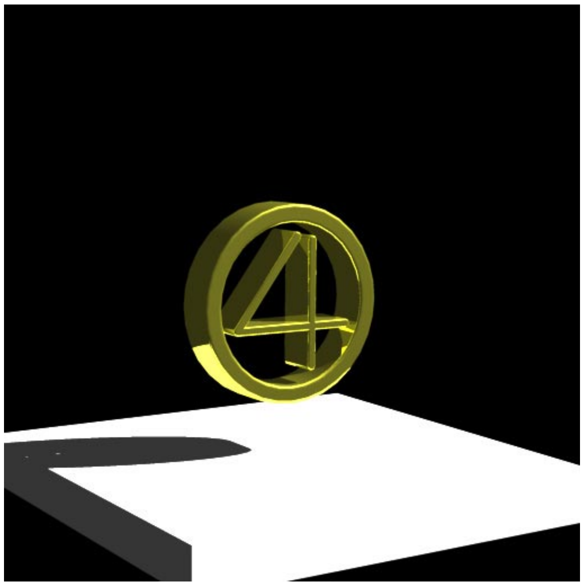

Figure 10: Stationary version of the logo

The heuristic used by the pixel thresher fails if isolated subpixel details occur in the image. This could be mitigated by using more than one sample in the initial sample set. In practice, this hasn't been a problem. Figures 11b and 12b show (in red) which pixels were super-sampled for some sample images. Most of the super-sampling is concentrated around the high frequency or blurred portions of the image.

6 Results and Summary

The method for ray tracing splines presented here was compared to a numerical method previously implemented by the same author [14]. For typical images (such as figure 12) the method presented here is more than twice the speed of the numerical methods. It has also proven much more stable, since the algorithms used are free from stability and convergence problems inherent with numerical methods.

The ray tracing program is part of the Alpha_1 geometric modeling system developed at the University of Utah [17]. Alpha_1 provides an extensive set of tools for modeling objects with B-spline surfaces. The ray tracing package provides a way to get much higher quality images

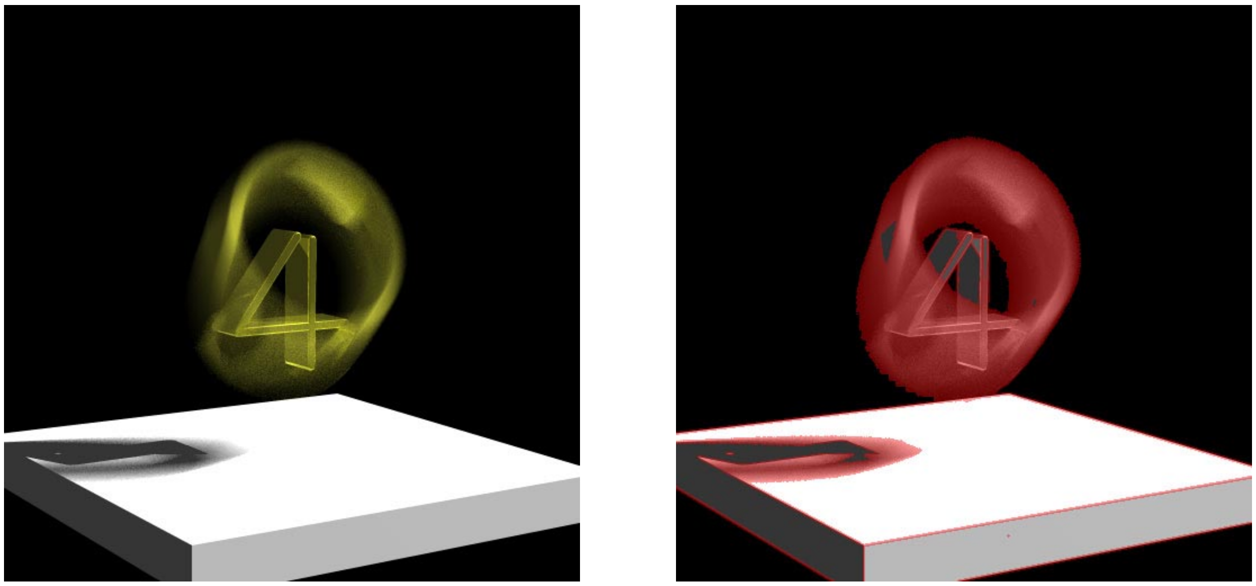

Figure 11: *a.* Motion blurred object. *b.* Pixels super-sampled in *a.*

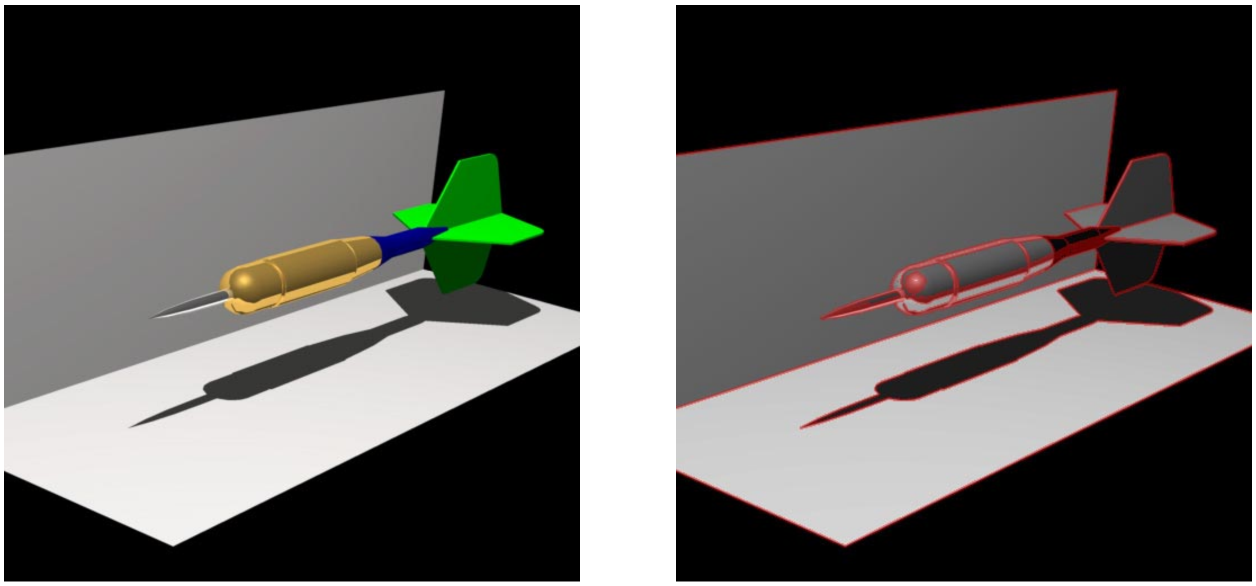

Figure 12: *a.* Dart. *b.* Pixels super-sampled in *a.*

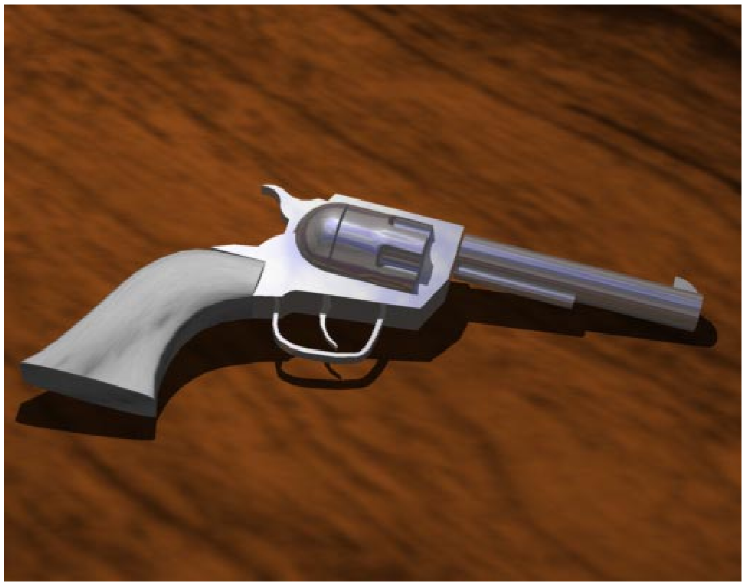

Figure 13: Gun model

than is possible using interactive techniques. The motion blur facilities allow the creation of animation without temporal aliasing effects.

Figures 10–13 demonstrate some objects modeled with Alpha_1 and rendered with the ray tracing package. Figure 10 shows a three dimensional logo modeled with quadratic B-splines. In figure 11a, the ring surrounding the four is rotating and moving during the frame. Figure 13 shows a gun model developed with Alpha_1. The model uses a wide variety of surface types, e.g., flat surfaces in the body, rational quadratic surfaces (the barrel) and free-form cubic surfaces (the handle).

7 Conclusions and future work

The development of the algorithm for ray tracing splines presented here is similar to the evolution of scanline algorithms several years ago. In the case of scanline algorithms, approximating the surface by subdivision techniques is generally prefered now over using complex numerical methods, because it is more stable, more efficient, and less complex. The results of this work indicate the same is true for ray tracing.

The polygonal approximation method for ray tracing splines does represent a tradeoff between CPU time and memory usage since it requires the polygonal representation of the spline and the tree of hierarchical bounding volumes to be stored during the ray tracing process. The benefit, however, is that the intersection calculation (usually the most time consuming portion of the ray tracing process) executes much faster.

The use of a binary tree for the bounding volume hierarchy follows directly from the subdivision process. One possible way to reduce the memory needed for the bounding volumes would be to use a tree with a higher order than a binary tree. For example, if a new node in the tree was created for every other subdivision performed, than each node would have four children instead of two. This would decrease the number of nodes traversed before reaching a leaf, but increase the number of intersection tests required at each node. Discovering the performance tradeoffs (in terms of both memory usage and CPU time) has yet to be addressed.

The use of hierarchical bounding volumes allows the method to be extended for motion blur while still maintaining an efficient intersection algorithm. A heuristic adaptive anti-aliasing method was developed allowing a small number of initial samples in the image. The method does an effective job of performing super-sampling on only those pixels that are likely to require it, even if there are moving objects in the scene.

8 Acknowledgements

The subdivision algorithm given in section 3.3 was originally implemented by Russ Fish and Beth Cobb as part of a scanline oriented rendering program. The model in figure 13 was created by Glenn Mcminn, the dart in figure 12 was created by the author and Tom McCollough, and the logo in figure 10 was created by Rich Riesenfeld. I would like to thank Rod Bogart and Tom Malley for helpful discussion, and Beth Cobb, Spencer Thomas and Tom McCollough for providing comments on drafts of this paper.

This work was supported in part by DARPA (DAAK1184K0017) and the National Science

Foundation (MCS-8121750). All opinions, findings, conclusions or recommendations expressed in this document are those of the author and do not necessarily reflect the views of the sponsoring agencies.

References

[1] Richard H. Bartels, John C. Beatty, and Brian A. Barsky. *An Introduction to the Use of Splines in Computer Graphics.* Technical Report TR CS-83-09, University of Waterloo, May 1985. Published as SIGGRAPH-85 Tutorial notes; also available from U.C. Berkeley as TR UCB/CSD 83/136.

[2] Böhm, Wolfgang, Gerald Farin, and Kahmann, Jürgen. A Survey of Curve and Surface Methods in CAGD. *Computer Aided Geometric Design*, 1(1):3–60, July 1984.

[3] Ed Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces.* PhD thesis, University of Utah, December 1974.

[4] Elaine Cohen, Tom Lyche, and Richard Riesenfeld. Discrete B-splines and Subdivision Techniques in Computer-Aided Geometric Design and Computer Graphics. *Computer Graphics and Image Processing*, 14(2):87–111, October 1980.

[5] Robert L. Cook. Stochastic Sampling in Computer Graphics. *ACM Transactions on Graphics*, 5(1):51–72, January 1986.

[6] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed Ray Tracing. *Computer Graphics*, 18(3):137–145, July 1984. (Proc. SIGGRAPH 85).

[7] Kenneth I. Joy and Murthy N. Bhetanabholta. Ray Tracing Parametric Surfaces Patches Utilizing Numerical Techniques and Ray Coherence. *Computer Graphics*, 20(4):279–284, August 1986.

[8] James T. Kajiya. New Techniques for Ray Tracing Procedurally Defined Objects. *Transactions on Graphics*, 2(3):161–181, July 1983.

[9] James T. Kajiya. Ray Tracing Parametric Patches. *Computer Graphics*, 16(3):245–254, July 1982. Proceedings of SIGGRAPH82.

[10] Timothy L. Kay and James T. Kajiya. Ray Tracing Complex Scenes. *Computer Graphics*, 20(4):269–277, August 1986.

[11] J. Korein and Norman Badler. Temporal Anti-aliasing in Computer Generated Animation. *Computer Graphics*, 17(3):377, July 1983.

[12] Jeffrey M. Lane, Loren C. Carpenter, Turner Whitted, and James F. Blinn. Scan Line Methods for Displaying Parametrically Defined Surfaces. *Communications of the ACM*, 23(1):23–34, January 1980.

[13] Mark Lee, Richard Redner, and Samuel Uselton. Statistically Optimized Sampling for Distributed Ray Tracing. *Computer Graphics*, 19(3):61–65, July 1985. Proceedings of Siggraph '85.

[14] John W. Peterson. Ray Tracing General B-Splines. In *ACM Mountain Regional Conferance Proceedings*, page 87, ACM (Mountain Region), Santa Fe, NM, April 1986.

[15] John W. Peterson, Rod G. Bogart, and Spencer W. Thomas. The Utah Raster Toolkit. In Lou Katz, editor, *Proceedings of the Third Workshop on Computer Graphics*, Usenix, November 1986.

[16] Thomas Porter and Tom Duff. Compositing Digital Images. *Computer Graphics*, 18(3):253, July 1984. Proceedings of SIGGRAPH 84.

[17] Utah Alpha_1 Project. *Alpha_1 User's Manual.* University of Utah, Dept. of Computer Science, Salt Lake City, Utah, 1986.

[18] Ron Pulleyblank and John Kapenga. The Feasibility of a VLSI Chip for Ray Tracing Bicubic Patches. *IEEE Computer Graphics and Applications*, 7(3):33–44, March 1987.

[19] Scott Roth. Ray Casting for Modeling Solids. *Computer Graphics and Image Processing*, 4(18):109, 1982.

[20] Gilbert Strang. *Linear Algebra and its Applications.* Academic Press, New York, 1976.

[21] Michael Sweeney and Richard H. Bartels. Ray Tracing Free-Form B-Spline Surfaces. *IEEE Computer Graphics and Applications*, 6(2):41, February 1986.

[22] Wayne Tiller. Rational B-Splines for Curve and Surface Representation. *IEEE Computer Graphics and Applications*, 3(6):61–69, September 1983.

[23] Daniel L. Toth. On Ray Tracing Parametric Surfaces. *Computer Graphics*, 19(3):171–179, July 1985.

[24] Turner Whitted. An Improved Model for Shaded Display. *Communcations of the ACM*, 23(6):96, June 1980.