Ray Bilinear Patch Intersections

Shaun D. Ramsey ramsey@cs.utah.edu
Kristin Potter kpotter@cs.utah.edu
Charles Hansen hansen@cs.utah.edu
School of Computing, University of Utah

Abstract

Ray tracing and other techniques employ algorithms which require the intersection between a 3D parametric ray and an object to be computed. The object to intersect is typically a sphere, triangle, or polygon but many surface types are possible. In this work we consider intersections between rays and the simplest parametric surface, the bilinear patch. Unlike other surfaces, solving the ray-bilinear patch intersection with simple algebraic manipulations fails. We present a complete, efficient, robust, and graceful formulation to solve ray-bilinear patch intersections quickly. Source code is available online.

1 Introduction

In computer graphics there are many techniques, such as ray-tracing, which require an intersection between a 3D parametric ray and an object. Often spheres, triangles, and polygons (see [1]) are used as the object although other surface representations are possible. In this work, we consider intersections between a ray and a bilinear patch. At first glance the problem appears rather simple, however a naive approach reveals singularities. Bilinear patches are a useful primitive that may be used in ray tracing, such as in the work completed by Smits et al. [2]. We show a more complete and graceful formulation than previously presented ([3]). Also, we describe and provide complete implementation details. Source code is available online at the website listed at the end of this paper.

In ray tracing, the ray can be mathematically represented as a parametric line as shown in the following equation:

$$\vec{p}(t) = \vec{r} + t\vec{q}, \;\; \forall \;\; t \geq 0 \tag{1}$$

Any point $\vec{p}$ along the ray can be expressed by a specific t value corresponding to the parametric distance along the direction vector $\vec{q}$ from the ray origin $\vec{r}$. When $t = 0$ the returned point is the ray origin. Likewise, $t < 0$ corresponds to points that lie on the *line* that passes through $\vec{r}$, but are behind the ray origin $\vec{r}$. Thus, points along the ray are valid when $t \geq 0$.

Bilinear patches are formed as a combination of four possibly non-coplanar points ($\vec{p_{00}}$, $\vec{p_{01}}$, $\vec{p_{10}}$, $\vec{p_{11}}$). The contribution of each point is described as a weighting of the two parameters (u,v), such that any point p can be represented by the following equations with domain $(u, v) \in [0, 1]^2$:

$$\begin{aligned} \vec{p}(u,v) &= (1-u)(1-v)\vec{p_{00}} + (1-u)v\vec{p_{01}} \\ &\quad + u(1-v)\vec{p_{10}} + uv\vec{p_{11}} \end{aligned} \tag{2}$$

$$\begin{aligned} &= uv(\vec{p_{11}} - \vec{p_{10}} - \vec{p_{01}} + \vec{p_{00}}) \\ &\quad + u(\vec{p_{10}} - \vec{p_{00}}) + v(\vec{p_{01}} - \vec{p_{00}}) + \vec{p_{00}} \end{aligned} \tag{3}$$

The following variables can be used for substitution:

$$\begin{aligned} \vec{a} &= \vec{p_{11}} - \vec{p_{10}} - \vec{p_{01}} + \vec{p_{00}} \\ \vec{b} &= \vec{p_{10}} - \vec{p_{00}} \\ \vec{c} &= \vec{p_{01}} - \vec{p_{00}} \\ \vec{d} &= \vec{p_{00}} \end{aligned}$$

and, the bilinear patch equation becomes:

$$\vec{p}(u,v) = uv\vec{a} + u\vec{b} + v\vec{c} + \vec{d}, \qquad (u,v) \in [0,1]^2 \tag{4}$$

2 Derivations

To find the intersection of the ray and bilinear patch, the ray is set equal to the bilinear patch:

$$\vec{r} + t\vec{q} = uv\vec{a} + u\vec{b} + v\vec{c} + \vec{d} \tag{5}$$

where, t, u and v are the unknowns.

The first step to solving the intersection is to solve for t:

$$\begin{aligned} t &= (uva_x + ub_x + vc_x + d_x - r_x)/q_x \\ t &= (uva_y + ub_y + vc_y + d_y - r_y)/q_y \\ t &= (uva_z + ub_z + vc_z + d_z - r_z)/q_z \end{aligned} \tag{6}$$

The problem now has 3 equations and 3 unknowns. The unknown t can be eliminated by setting the x and y equations equal to the z equation and factoring out u and v.

$$\begin{array}{lll} uv & (a_x q_z - a_z q_x) + & u \quad (b_x q_z - b_z q_x) + \\ v & (c_x q_z - c_z q_x) + (d_x - r_x) q_z - (d_z - r_z) q_x = 0 \end{array} \tag{7}$$

$$\begin{array}{lll} uv & (a_y q_z - a_z q_y) + & u \quad (b_y q_z - b_z q_y) + \\ v & (c_y q_z - c_z q_y) + (d_y - r_y) q_z - (d_z - r_z) q_y = 0 \end{array} \tag{8}$$

The following variables are used for substitution:

$$\begin{aligned} A_1 &= a_x q_z - a_z q_x \\ B_1 &= b_x q_z - b_z q_x \\ C_1 &= c_x q_z - c_z q_x \\ D_1 &= (d_x - r_x) q_z - (d_z - r_z) q_x \\ A_2 &= a_y q_z - a_z q_y \\ B_2 &= b_y q_z - b_z q_y \\ C_2 &= c_y q_z - c_z q_y \\ D_2 &= (d_y - r_y) q_z - (d_z - r_z) q_y \end{aligned}$$

The equations then simplify to two equations and two unknowns:

$$uvA_1 + uB_1 + vC_1 + D_1 = 0 \tag{9}$$

$$uvA_2 + uB_2 + vC_2 + D_2 = 0 \tag{10}$$

Equation 10 is used to solve for u:

$$u = \frac{(-vC_2 - D_2)}{(vA_2 + B_2)} \tag{11}$$

Equation 9 can then be used to eliminate u:

$$(\tfrac{-vC_2 - D_2}{vA_2 + B_2}) vA_1 + (\tfrac{-vC_2 - D_2}{vA_2 + B_2}) B_1 + vC_1 + D_1 = 0$$

Obtaining a common denominator and collecting like terms results in:

$$\begin{array}{lll} & v^2 & (A_2 C_1 - A_1 C_2) \\ + & v & (A_2 D_1 - A_1 D_2 + B_2 C_1 - B_1 C_2) \\ + & & (B_2 D_1 - B_1 D_2) = 0 \end{array} \tag{12}$$

Equation 12 is a quadratic equation in canonical form that we solve as described in [4]. Once v is found, we solve using equation 11 to find the corresponding u value of the intersection point. Depending on the patch and the ray however, the denominator of Equation 11 may become zero for a valid intersection. We solve this problem, by setting Equations 7 and 8 equal before solving for u. This gives a second equation for u:

$$u = \frac{v(C_1 - C_2) + (D_1 - D_2)}{v(A_2 - A_1) + (B_2 - B_1)} \tag{13}$$

With these derivations, we are ready to run the actual ray bilinear patch intersection algorithm.

Algorithm 3.1: INTERSECT($ray, patch$)

```
v_i ← QuadraticSolver(Equation 12)
comment: i is the # of roots from Eqn 12
if i = 0
   return ( false )
if i = 1
   return (SOLVE(ray, patch, v_1))
if i = 2
{ r_1 =SOLVE(ray, patch, v_1)
  r_2 =SOLVE(ray, patch, v_2)
  if r_1 = false
     return (r_2)
  else if r_2 = false
     return (r_1)
  else
     { if r_1.t < r_2.t
          return (r_1)
       else
          return (r_2)
```

Algorithm 3.2: SOLVE($ray, patch, v$)

```
if v ∈ [0, 1]
{ u ←COMPUTE U(v)
  if u ∈ [0, 1]
  { Point p ← patch(u, v)
    t ←COMPUTE T(ray, p)
    if t ≥ 0
       return (u, v, t)
return ( false )
```

3 Algorithm

Given a ray and a bilinear patch, the intersection point is computed using Algorithm 3.1. The algorithm first calls the quadratic equation solver which returns zero, one or two solutions for v depending on the number of intersection points as shown in Figure 1. For each solution of v, where $v \in [0, 1]$, the corresponding u and t values are computed using Algorithm 3.2. Because the ray can intersect the bilinear patch multiple times, we must accurately choose the correct inter-

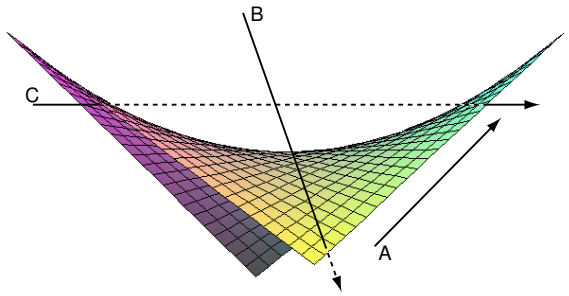

Figure 1. *A bilinear patch being intersected by a variety of rays. A ray may completely miss the surface (A), intersect the surface only once (B), or intersect the surface twice (C).*

Algorithm 3.3: COMPUTE U(v)

$a = vA_2 + B_2$
$b = v(A_2 - A_1) + B_2 - B_1$
if ($|b| \geq |a|$)
    return ($\frac{v(C_1-C_2)+D_1-D_2}{b}$)
else
    return ($\frac{-vC_2-D_2}{a}$)

section point. Our algorithm chooses the intersection point with the smallest t which is greater than or equal to zero. This t value corresponds to the first intersection between the ray and the bilinear patch.

Once the roots of the quadratic equation are determined, they are passed to Algorithm 3.2, which computes the u and t values.

Due to the numerical instability that may occur when calculating a u value using Equation 11, a second derivation of u is formed by Equation 13. Algorithm 3.3 computes the u value by choosing the largest absolute value of the denominators of Equations 11 and 13. This approach ensures that the computation of u will be stable. If $u \notin [0, 1]$, then this is not an intersection and we do not calculate t for this (u, v) pair.

To calculate the t value, we calculate $\vec{p}$ using Equation 2. Given $\vec{p}$, we then use Equation 1 to solve for t. Note that this approach is the same as using the equations in Equation 6 to compute t. Because the formulation of the ray equation forces a division when solving for t, Algorithm 3.4 uses the direction vector component with the largest absolute value. Divisions by zero are again impossible since a direction vector of all zeroes is invalid.

Algorithm 3.4: COMPUTE T($\vec{r}, \vec{q}, \vec{p_1}$)

if ($|q_x| \geq |q_y|$ && $|q_x| \geq |q_z|$)
    $t = (p_{1x} - r_x)/q_x$
else if ($|q_y| \geq |q_z|$)
    $t = (p_{1y} - r_y)/q_y$
else
    $t = (p_{1z} - r_z)/q_z$
return (t)

Ray Formulation	Intersections	Seconds
Ray Equation	790,000,000	754
Two Plane	790,000,000	963

Table 1. *Performance comparison between ray formulation using the ray equation and two-plane approaches.*

4 Discussion

When implemented on a Xeon 2.66 GHz with 1.00 GB of RAM, on average over 1,000,000 intersection tests were performed per second. These intersections were also tested visually in the Utah Real Time Ray Tracer on a variety of bilinear patches.

Another common formulation of ray bilinear patch intersections represents the ray as the intersection of two planes (see [3]). This is done in an effort to remove the singularities present in our approach. Such an implementation changes the ray equation and solves the system in much the same way as presented above. However, the two-plane approach only moves the singularities to a different location in the solution and the performance decreases due to the set-up time of the two planes. Table 1 compares the performance of the two approaches.

5 Web Information

Source code and a simple driver program are available at:
http://www.acm.org/jgt/papers/RamseyPotterHansen04

References

[1] Shirley, Peter and R. Keith Morley, *Realistic Ray Tracing*, A K Peters, 2003, pp 29-38.

[2] B. Smits, P. Shirley, M.M. Stark, *Direct Ray Tracing of Displacement Mapped Triangles*, Rendering Techniques 2000, pp 307-318.

[3] A.S. Glassner, editor, *An Introduction to Ray Tracing*, Academic Press, 1997, pp 94-101.

[4] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in C*, Cambridge University Press, 1988, p 156.