3D Game Engine Design

A Practical Approach to Real-Time Computer Graphics

DAVID H. EBERLY

Magic Software, Inc.

MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ACADEMIC PRESS

A Harcourt Science and Technology Company

SAN FRANCISCO SAN DIEGO NEW YORK BOSTON

LONDON SYDNEY TOKYO

Acquisitions Editor Tim Cox
Production Editors Elisabeth Beller and Sarah Burgundy
Editorial Assistants Brenda Modliszewski and Stacie Pierce
Cover Design Ross Carron Design
Text Design Rebecca Evans
Color Insert Preparation Side By Side Studios/Mark Ong
Composition/Illustration Windfall Software, using ZzTEX
Copyeditor Ken DellaPenta
Proofreader Jennifer McClain
Indexer Steve Rath
Printer Courier Corporation

ACADEMIC PRESS
A Harcourt Science and Technology Company
525 B Street, Suite 1900, San Diego, CA 92101-4495, USA
http://www.academicpress.com

Academic Press
Harcourt Place, 32 Jamestown Road, London, NW1 7BY, United Kingdom
http://www.academicpress.com

Morgan Kaufmann Publishers
340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA
http://www.mkp.com

Library of Congress Cataloging-in-Publication Data

Eberly, David H.
3D game engine design : a practical approach to real-time computer graphics / David H. Eberly.
p. cm.
Includes bibliographical references and index.
ISBN 1-55860-593-2
1. Computer graphics. 2. Three-dimensional display systems. 3. Real-time programming. I. Title.
T385 .E373 2001
006.6'93—dc21 00-055019

This book is printed on acid-free paper.

thereby showing that the transform of a linear combination of vectors is the linear combination of the transforms.

The previous three properties show that $R(\hat{v})$ is an orthonormal transformation, a class that includes rotations *and* reflections. We need to show that reflections cannot occur. For unit-length vector $\vec{v}$, define the function M by $\hat{v} = M(\vec{v})$, a function from the unit sphere in $\mathbb{R}^3$ to the unit quaternions with zero real part. Its inverse is $\vec{v} = M^{-1}(\hat{v})$. If $\hat{w} = M(\vec{w})$ and $\hat{w} = R(\hat{v}) = q\hat{v}q^*$, then the composition

$$\vec{w} = M^{-1}(\hat{w}) = M^{-1}(R(\hat{v})) = M^{-1}(R(M(\vec{v})))$$

defines a matrix transformation $\vec{w} = P\vec{v}$, where P is an orthonormal matrix since $R(\hat{v})$ is an orthonormal transformation. Thus, $|\det(P)| = 1$, which implies that the determinant can be only $+1$ or -1. P is determined by the choice of unit quaternion q, so it is a function of q, written as $P(q)$ to show the functional dependence. Moreover, $P(q)$ is a continuous function, which in turn implies that $\delta(q) = \det(P(q))$ is a continuous function of q. By the definition of continuity, $\lim_{q\to 1} P(q) = P(1) = I$, the identity matrix, and $\lim_{q\to 1} \delta(q) = \delta(1) = 1$. Since $\delta(q)$ can only be $+1$ or -1 and since the limiting value is $+1$, $\delta(q) = 1$ is true for all unit quaternions. Consequently, P cannot contain reflections.

We now prove that the unit rotation axis is the 3D vector $\hat{u}$ and the rotation angle is 2θ. To see that $\hat{u}$ is a unit rotation axis, we need only show that $\hat{u}$ is unchanged by the rotation. Recall that $\hat{u}^2 = \hat{u}\hat{u} = -1$. This implies that $\hat{u}^3 = -\hat{u}$. Now

$$\begin{aligned}
R(\hat{u}) &= q\hat{u}q^* \\
&= (\cos\theta + \hat{u}\sin\theta)\hat{u}(\cos\theta - \hat{u}\sin\theta) \\
&= (\cos\theta)^2\hat{u} - (\sin\theta)^2\hat{u}^3 \\
&= (\cos\theta)^2\hat{u} - (\sin\theta)^2(-\hat{u}) \\
&= \hat{u}.
\end{aligned}$$

To see that the rotation angle is 2θ, let $\hat{u}$, $\hat{v}$, and $\hat{w}$ be a right-handed set of orthonormal vectors. That is, the vectors are all unit length; $\hat{u}\cdot\hat{v} = \hat{u}\cdot\hat{w} = \hat{v}\cdot\hat{w} = 0$, and $\hat{u}\times\hat{v} = \hat{w}$, $\hat{v}\times\hat{w} = \hat{u}$, and $\hat{w}\times\hat{u} = \hat{v}$. The vector $\hat{v}$ is rotated by an angle ϕ to the vector $q\hat{v}q^*$, so $\hat{v}\cdot(q\hat{v}q^*) = \cos(\phi)$. Using Equation (2.8) and $\hat{v}^* = -\hat{v}$, and $\hat{p}^2 = -1$ for unit quaternions with zero real part,

$$\begin{aligned}
\cos(\phi) &= \hat{v}\cdot(q\hat{v}q^*) \\
&= W(\hat{v}^*q\hat{v}q^*) \\
&= W[-\hat{v}(\cos\theta + \hat{u}\sin\theta)\hat{v}(\cos\theta - \hat{u}\sin\theta)]
\end{aligned}$$

the axis $\vec{U} = (u_0, u_1, u_2)$. Given the angle and axis, the components of the quaternion are $w = \cos(\theta/2)$, $x = u_0 \sin(\theta/2)$, $y = u_1 \sin(\theta/2)$, and $z = u_2 \sin(\theta/2)$.

Quaternion to Angle-Axis

The inverse problem is also straightforward. If $|w| = 1$, then the angle is $\theta = 0$ and any axis will do. If $|w| < 1$, the angle is obtained as $\theta = 2\cos^{-1}(w)$ and the axis is computed as $\vec{U} = (x, y, z)/\sqrt{1 - w^2}$.

2.3.5 Conversion between Quaternion and Rotation Matrix

To complete the set of conversions between representations of rotations, this section describes the conversions between quaternions and rotation matrices.

Quaternion to Rotation Matrix

The problem is to compute θ and $\vec{U}$ given w, x, y, and z. Using the identities $2\sin^2(\theta/2) = 1 - \cos(\theta)$ and $\sin(\theta) = 2\sin(\theta/2)\cos(\theta/2)$, it is easily shown that $2wx = (\sin\theta)u_0$, $2wy = (\sin\theta)u_1$, $2wz = (\sin\theta)u_2$, $2x^2 = (1 - \cos\theta)u_0^2$, $2xy = (1 - \cos\theta)u_0u_1$, $2xz = (1 - \cos\theta)u_0u_2$, $2y^2 = (1 - \cos\theta)u_1^2$, $2yz = (1 - \cos\theta)u_1u_2$, and $2z^2 = (1 - \cos\theta)u_2^2$. The right-hand sides of all these equations are terms in the expression $R = I + (\sin\theta)S + (1 - \cos\theta)S^2$. Replacing them yields

$$R = \begin{bmatrix} 1 - 2y^2 - 2z^2 & 2xy - 2wz & 2xz + 2wy \\ 2xy + 2wz & 1 - 2x^2 - 2z^2 & 2yz - 2wx \\ 2xz - 2wy & 2yz + 2wx & 1 - 2x^2 - 2y^2 \end{bmatrix}. \tag{2.13}$$

Rotation Matrix to Quaternion

Earlier it was mentioned that $\cos\theta = (\text{trace}(R) - 1)/2$. Using the identity $2\cos^2(\theta/2) = 1 + \cos\theta$ yields $w^2 = \cos^2(\theta/2) = (\text{trace}(R) + 1)/4$ or $|w| = \sqrt{\text{trace}(R) + 1}/2$. If $\text{trace}(R) > 0$, then $|w| > 1/2$, so without loss of generality choose w to be the positive square root, $w = \sqrt{\text{trace}(R) + 1}/2$. The identity $R - R^{\mathrm{T}} = (2\sin\theta)S$ also yielded $(r_{12} - r_{21}, r_{20} - r_{02}, r_{01} - r_{10}) = 2\sin\theta(u_0, u_1, u_2)$. Finally, identities derived earlier were $2xw = u_0\sin\theta$, $2yw = u_1\sin\theta$, and $2zw = u_2\sin\theta$. Combining these leads to $x = (r_{12} - r_{21})/(4w)$, $y = (r_{20} - r_{02})/(4w)$, and $z = (r_{01} - r_{10})/(4w)$.

If $\text{trace}(R) \le 0$, then $|w| \le 1/2$. The idea is to first extract the largest one of x, y, or z from the diagonal terms of the rotation R in Equation (2.13). If r_{00} is the maximum diagonal term, then x is larger in magnitude than y or z. Some algebra shows

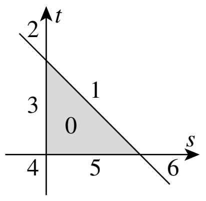

Figure 2.3 Partitioning of the st-plane by triangle domain D.

(implicitly defined by $Q = V_0$) just touches the triangle domain edge $s + t = 1$ at a value $s = s_0 \in [0, 1]$, $t_0 = 1 - s_0$. For level values $V < V_0$, the corresponding ellipses do not intersect D. For level values $V > V_0$, portions of D lie inside the corresponding ellipses. In particular, any points of intersection of such an ellipse with the edge must have a level value $V > V_0$. Therefore, $Q(s, 1 - s) > Q(s_0, t_0)$ for $s \in [0, 1]$ and $s \neq s_0$. The point (s_0, t_0) provides the minimum squared distance between $\vec{P}$ and the triangle. The triangle point is an edge point. Figure 2.4 illustrates the idea by showing various level curves.

An alternate way of visualizing where the minimum distance point occurs on the boundary is to intersect the graph of Q with the plane $s = 1$. The curve of intersection is a parabola and is the graph of $F(s) = Q(s, 1 - s)$ for $s \in [0, 1]$. Now the problem has been reduced by one dimension to minimizing a function $F(s)$ for $s \in [0, 1]$. The minimum of $F(s)$ occurs either at an interior point of $[0, 1]$, in which case $F'(s) = 0$ at that point, or at an end point $s = 0$ or $s = 1$. Figure 2.4 shows the case when the minimum occurs at an interior point of the edge. At that point the ellipse is tangent to the line $s + t = 1$. In the end point cases, the ellipse may just touch one of the vertices of D, but not necessarily tangentially.

To distinguish between the interior point and end point cases, the same partitioning idea applies in the one-dimensional case. The interval $[0, 1]$ partitions the real line into three intervals, $s < 0$, $s \in [0, 1]$, and $s > 1$. Let $F'(\hat{s}) = 0$. If $\hat{s} < 0$, then $F(s)$ is an increasing function for $s \in [0, 1]$. The minimum restricted to $[0, 1]$ must occur at $s = 0$, in which case Q attains its minimum at $(s, t) = (0, 1)$. If $\hat{s} > 1$, then $F(s)$ is a decreasing function for $s \in [0, 1]$. The minimum for F occurs at $s = 1$ and the minimum for Q occurs at $(s, t) = (1, 0)$. Otherwise, $\hat{s} \in [0, 1]$, F attains its minimum at $\hat{s}$, and Q attains its minimum at $(s, t) = (\hat{s}, 1 - \hat{s})$.

The occurrence of $(\bar{s}, \bar{t})$ in region 3 or 5 is handled in the same way as when the global minimum is in region 0. If $(\bar{s}, \bar{t})$ is in region 3, then the minimum occurs at $(0, t_0)$ for some $t_0 \in [0, 1]$. If $(\bar{s}, \bar{t})$ is in region 5, then the minimum occurs at $(s_0, 0)$ for some $s_0 \in [0, 1]$. Determining if the first contact point is at an interior or end point of the appropriate interval is handled the same as discussed earlier.

If $(\bar{s}, \bar{t})$ is in region 2, it is possible the level curve of Q that provides first contact with the unit square touches either edge $s + t = 1$ or edge $s = 0$. Because the global minimum occurs in region 2, the negative of the gradient at the corner $(0, 1)$ cannot

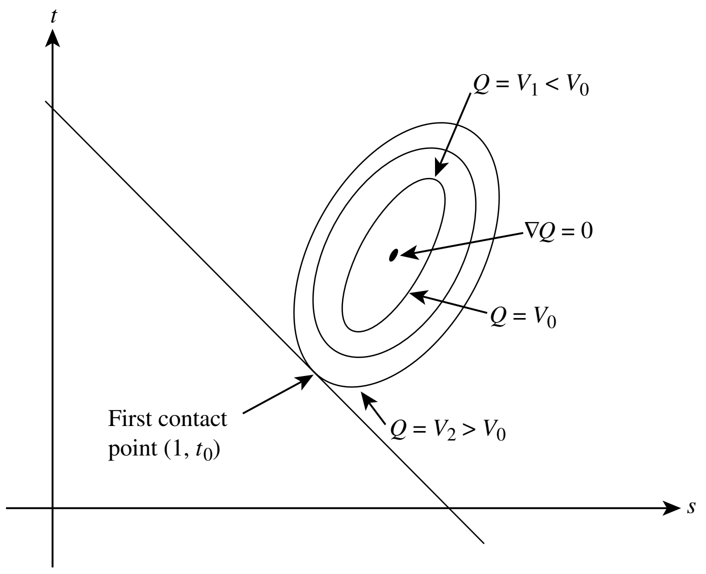

Figure 2.4 Various level curves $Q(s, t) = V$.

point inside D. If $\vec{\nabla} Q = (Q_s, Q_t)$, where Q_s and Q_t are the partial derivatives of Q, it must be that $(0, -1) \cdot \vec{\nabla} Q(0, 1)$ and $(1, -1) \cdot \vec{\nabla} Q(0, 1)$ cannot both be negative. The two vectors $(0, -1)$ and $(1, -1)$ are directions for the edges $s = 0$ and $s + t = 1$, respectively. The choice of edge $s + t = 1$ or $s = 0$ can be made based on the signs of $(0, -1) \cdot \vec{\nabla} Q(0, 1)$ and $(1, -1) \cdot \vec{\nabla} Q(0, 1)$. The same type of argument applies in region 6. In region 4, the two quantities whose signs determine which edge contains the minimum are $(1, 0) \cdot \vec{\nabla} Q(0, 0)$ and $(0, 1) \cdot \vec{\nabla}(0, 0)$.

The implementation of the algorithm is designed so that at most one floating-point division is used when computing the minimum distance and corresponding closest points. Moreover, the division is deferred until it is needed, and in some cases no division is needed.

Quantities that are used throughout the code are computed first. In particular, the values computed are $\vec{D} = \vec{B} - \vec{P}$, $a = \vec{E}_0 \cdot \vec{E}_0$, $b = \vec{E}_0 \cdot \vec{E}_1$, $c = \vec{E}_1 \cdot \vec{E}_1$, $d = \vec{E}_0 \cdot \vec{D}$, $e = \vec{E}_1 \cdot \vec{D}$, and $f = \vec{D} \cdot \vec{D}$. The code actually computes $\delta = |ac - b^2|$ since it is possible for small edge lengths that some floating-point round-off errors lead to a small negative quantity.

In the theoretical development, $\bar{s} = (be - cd)/\delta$ and $(bd - ae)/\delta$ were computed so that $\vec{\nabla} Q(\bar{s}, \bar{t}) = (0, 0)$. The location of the global minimum is then tested to see if it is in the triangle domain D. If so, then the information to compute the minimum distance is known. If not, then the boundary of D must be tested. To defer the division by δ, the code instead computes $\bar{s} = be - cd$ and $\bar{t} = bd - ae$ and tests for physics systems: any collision detection back end can be fit with any physics system front end.

Hierarchical organization of data allows the application to tag each node with a set of flags indicating how the collision test should propagate. The simplest choice is whether or not to recurse on the call or to terminate immediately. Other choices involve specifying what types of calculations should occur (test only, first time only, first point of contact, do only bounding volume comparisons but not triangle-triangle tests, go all the way to triangle-triangle tests, etc.).

Remembering information about a previous intersection may help in localizing the search for the next call of the collision system. The usual space-time trade-off applies: more memory is used to retain state information in exchange for a faster execution. Whether space or time is important depends on the application and its data. For example, retaining state information is a key feature in the GJK and extended GJK algorithms (Gilbert, Johnson, and Keerthi 1988; Cameron 1996; van den Bergen 1999), but bounding volume trees typically do not retain state information and are designed to localize the search by fast intersection tests between the bounding volumes (Gottschalk, Lin, and Manocha 1991; Gregory et al. 1998). Both approaches are viable, but in this chapter we will discuss only the bounding volume tree ideas.

6.2 Intersection of Dynamic Objects and Lines

In the following sections, the line is stationary and defined by $\vec{P} + s\vec{D}$ for $s \in \mathbb{R}$. The other objects are moving with constant linear velocity $\vec{W}$ over a time interval $t \in [0, t_{\max}]$. If $\vec{D} \times \vec{W} = \vec{0}$, then the object is moving parallel to the line. The static test for intersection is sufficient for this case.

The algorithms presented here determine only if the line and object will intersect on the time interval. Computation of the first time of contact is typically more expensive. For the sphere, capsule, and lozenge, finding the first time of contact involves solving a quadratic equation, which requires taking a square root.

6.2.1 Spheres

Source Code

Library
Intersection

Filename
IntrLin3Sphr

The moving sphere has center $\vec{C} + t\vec{W}$ for $t \in [0, t_{\max}]$ and radius $r > 0$. The distance between a point and a line is given by Equation 2.14. Replacing the time-varying center in this equation leads to a quadratic function in t that represents the squared distance,

$$Q(t) = \left| \left(\vec{W} - \frac{\vec{D} \cdot \vec{W}}{\vec{D} \cdot \vec{D}} \vec{D} \right) t + \left((\vec{C} - \vec{P}) - \frac{\vec{D} \cdot (\vec{C} - \vec{P})}{\vec{D} \cdot \vec{D}} \vec{D} \right) \right|^2 =: at^2 + 2bt + c.$$

The coefficient a is positive because of the assumption that the direction of motion is not parallel to the line. If $Q(t) \le r^2$ for some $t \in [0, t_{\max}]$, then the line intersects the sphere during the specified time interval. The problem is now one of determining the

Table 6.10 Coefficients for unique points of triangle-OBB intersection for $\vec{A}_0 \times \vec{E}_j$.

$\vec{L}$	*Coefficients*
$\vec{A}_0 \times \vec{E}_0$	$x_1 = -\sigma \ \text{sign}(\vec{A}_2 \cdot \vec{E}_0)a_1, \ \ x_2 = +\sigma \ \text{sign}(\vec{A}_1 \cdot \vec{E}_0)a_2$
	$y_1 = \begin{cases} 0, & \sigma p_0 = \min_k(\sigma p_k) \\ 1, & \sigma(p_0 + \vec{N} \cdot \vec{A}_0) = \min_k(\sigma p_k) \end{cases}$
	$x_0 = \frac{\vec{A}_0 \times \vec{E}_0 \cdot (\vec{D} \times \vec{E}_0 - y_1 \vec{N} - x_1 \vec{A}_1 \times \vec{E}_0 - x_2 \vec{A}_2 \times \vec{E}_0)}{\lvert \vec{A}_0 \times \vec{E}_0 \rvert^2}$
$\vec{A}_0 \times \vec{E}_1$	$x_1 = -\sigma \ \text{sign}(\vec{A}_2 \cdot \vec{E}_1)a_1, \ \ x_2 = +\sigma \ \text{sign}(\vec{A}_1 \cdot \vec{E}_1)a_2$
	$y_0 = \begin{cases} 0, & \sigma p_0 = \min_k(\sigma p_k) \\ 1, & \sigma(p_0 - \vec{N} \cdot \vec{A}_0) = \min_k(\sigma p_k) \end{cases}$
	$x_0 = \frac{\vec{A}_0 \times \vec{E}_1 \cdot (\vec{D} \times \vec{E}_1 + y_0 \vec{N} - x_1 \vec{A}_1 \times \vec{E}_1 - x_2 \vec{A}_2 \times \vec{E}_1)}{\lvert \vec{A}_0 \times \vec{E}_1 \rvert^2}$
$\vec{A}_0 \times \vec{E}_2$	$x_1 = -\sigma \ \text{sign}(\vec{A}_2 \cdot \vec{E}_2)a_1, \ \ x_2 = +\sigma \ \text{sign}(\vec{A}_1 \cdot \vec{E}_2)a_2$
	$y_0 + y_1 = \begin{cases} 0, & \sigma p_0 = \min_k(\sigma p_k) \\ 1, & \sigma(p_0 - \vec{N} \cdot \vec{A}_0) = \min_k(\sigma p_k) \end{cases}$
	$x_0 = \frac{\vec{A}_0 \times \vec{E}_2 \cdot (\vec{D} \times \vec{E}_2 + (y_0 + y_1) \vec{N} - x_1 \vec{A}_1 \times \vec{E}_2 - x_2 \vec{A}_2 \times \vec{E}_2)}{\lvert \vec{A}_0 \times \vec{E}_2 \rvert^2}$

Finding the First Time of Intersection

Given that the two triangles do not intersect at time $t = 0$, but do intersect at some later time, a simple modification of the algorithm for testing for an intersection provides the first time of intersection. The first time is computed as the *maximum* time $T > 0$ for which there is at least one separating axis for any $t \in [0, T)$, but for which no separating axis exists at time T. The idea is to test each potential separating axis and keep track of the time at which the intervals of projection intersect for the first time. The largest such time is the first time at which the triangles intersect. Also, it is important to keep track of which side each of the intervals is relative to the other interval. Finally, knowing the separating axis associated with the maximum time T allows us to reconstruct a point of intersection.

The code for stationary triangles needs to be modified to handle the case of constant velocities. The velocity of the first triangle is subtracted from the velocity of the second triangle so that all calculations are done relative to a stationary first triangle. If the triangle velocities are $\vec{V}_0$ and $\vec{V}_1$, define the relative velocity to be $\vec{W} = \vec{V}_1 - \vec{V}_0$. Let the time interval be $[0, t_{\max}]$.

Table 6.11 Coefficients for unique points of triangle-OBB intersection for $\vec{A}_1 \times \vec{E}_j$.

$\vec{L}$	*Coefficients*
$\vec{A}_1 \times \vec{E}_0$	$x_0 = +\sigma \operatorname{sign}(\vec{A}_2 \cdot \vec{E}_0)a_0,\ \ x_2 = -\sigma \operatorname{sign}(\vec{A}_0 \cdot \vec{E}_0)a_2$
	$y_1 = \begin{cases} 0, & \sigma p_0 = \min_k(p_k) \\ 1, & \sigma(p_0 + \vec{N} \cdot \vec{A}_1) = \min_k(\sigma p_k) \end{cases}$
	$x_1 = \frac{\vec{A}_1 \times \vec{E}_0 \cdot (\vec{D} \times \vec{E}_0 - y_1 \vec{N} - x_0 \vec{A}_0 \times \vec{E}_0 - x_2 \vec{A}_2 \times \vec{E}_0)}{\lvert \vec{A}_1 \times \vec{E}_0 \rvert^2}$
$\vec{A}_1 \times \vec{E}_1$	$x_0 = +\sigma \operatorname{sign}(\vec{A}_2 \cdot \vec{E}_1)a_0,\ \ x_2 = -\sigma \operatorname{sign}(\vec{A}_0 \cdot \vec{E}_1)a_2$
	$y_0 = \begin{cases} 0, & \sigma(p_0 = \min_k(p_k)) \\ 1, & \sigma(p_0 - \vec{N} \cdot \vec{A}_1) = \min_k(\sigma p_k) \end{cases}$
	$x_1 = \frac{\vec{A}_1 \times \vec{E}_1 \cdot (\vec{D} \times \vec{E}_1 + y_0 \vec{N} - x_0 \vec{A}_0 \times \vec{E}_1 - x_2 \vec{A}_2 \times \vec{E}_1)}{\lvert \vec{A}_1 \times \vec{E}_1 \rvert^2}$
$\vec{A}_1 \times \vec{E}_2$	$x_0 = +\sigma \operatorname{sign}(\vec{A}_2 \cdot \vec{E}_2)a_0,\ \ x_2 = -\sigma \operatorname{sign}(\vec{A}_0 \cdot \vec{E}_2)a_2$
	$y_0 + y_1 = \begin{cases} 0, & \sigma p_0 = \min_k(p_k) \\ 1, & \sigma(p_0 - \vec{N} \cdot \vec{A}_1) = \min_k(\sigma p_k) \end{cases}$
	$x_1 = \frac{\vec{A}_1 \times \vec{E}_2 \cdot (\vec{D} \times \vec{E}_2 + (y_0 + y_1)\vec{N} - x_0 \vec{A}_0 \times \vec{E}_2 - x_2 \vec{A}_2 \times \vec{E}_2)}{\lvert \vec{A}_1 \times \vec{E}_2 \rvert^2}$

Axes $\vec{N}$ or $\vec{M}$

The problem is to make sure the minimum interval containing $\max(v_0 + t_{\max}w, v_1 + t_{\max}w, v_2 + t_{\max}w)$ does not intersect $\{u\}$. The pseudocode is

```
if ( v0 > u )
{
    if ( v1 >= v0 )
    {
        if ( v2 >= v0 )
        {
            min = v0;
            if ( min + tmax*w > u )
                return no_intersection;
        }
        else
        {
            min = v2;
            if ( min > u and min + tmax*w > u )
                return no_intersection;
```

Table 6.12 Coefficients for unique points of triangle-OBB intersection for $\vec{A}_2 \times \vec{E}_j$.

$\vec{L}$	*Coefficients*
$\vec{A}_2 \times \vec{E}_0$	$x_0 = -\sigma\ \text{sign}(\vec{A}_1 \cdot \vec{E}_0)a_0,\ \ x_1 = +\sigma\ \text{sign}(\vec{A}_0 \cdot \vec{E}_0)a_1$
	$y_1 = \begin{cases} 0, & \sigma p_0 = \min_k(\sigma p_k) \\ 1, & \sigma(p_0 + \vec{N} \cdot \vec{A}_2) = \min_k(\sigma p_k) \end{cases}$
	$x_2 = \frac{\vec{A}_2 \times \vec{E}_0 \cdot (\vec{D} \times \vec{E}_0 - y_1 \vec{N} - x_0 \vec{A}_0 \times \vec{E}_0 - x_1 \vec{A}_1 \times \vec{E}_0)}{\lvert \vec{A}_2 \times \vec{E}_0 \rvert^2}$
$\vec{A}_2 \times \vec{E}_1$	$x_0 = -\sigma\ \text{sign}(\vec{A}_1 \cdot \vec{E}_1)a_0,\ \ x_1 = +\sigma\ \text{sign}(\vec{A}_0 \cdot \vec{E}_1)a_1$
	$y_0 = \begin{cases} 0, & \sigma p_0 = \min_k(\sigma p_k) \\ 1, & \sigma(p_0 - \vec{N} \cdot \vec{A}_2) = \min_k(\sigma p_k) \end{cases}$
	$x_2 = \frac{\vec{A}_2 \times \vec{E}_1 \cdot (\vec{D} \times \vec{E}_1 + y_0 \vec{N} - x_0 \vec{A}_0 \times \vec{E}_1 - x_1 \vec{A}_1 \times \vec{E}_1)}{\lvert \vec{A}_2 \times \vec{E}_1 \rvert^2}$
$\vec{A}_2 \times \vec{E}_2$	$x_0 = -\sigma\ \text{sign}(\vec{A}_1 \cdot \vec{E}_2)a_0,\ \ x_1 = +\sigma\ \text{sign}(\vec{A}_0 \cdot \vec{E}_2)a_1$
	$y_0 + y_1 = \begin{cases} 0, & \sigma p_0 = \min_k(\sigma p_k) \\ 1, & \sigma(p_0 - \vec{N} \cdot \vec{A}_2) = \min_k(\sigma p_k) \end{cases}$
	$x_2 = \frac{\vec{A}_2 \times \vec{E}_2 \cdot (\vec{D} \times \vec{E}_2 + (y_0 + y_1)\vec{N} - x_0 \vec{A}_0 \times \vec{E}_2 - x_1 \vec{A}_1 \times \vec{E}_2)}{\lvert \vec{A}_2 \times \vec{E}_2 \rvert^2}$

```
            }
        }
        else if ( v1 >= v2 )
        {
            min = v2;
            if ( min > u and min + tmax*w > u )
                return no_intersection;
        }
        else
        {
            min = v1;
            if ( min > u and min + tmax*w > u )
                return no_intersection;
        }
    }
    else if ( v0 < u )
    {
        if ( v1 <= v0 )
        {
```

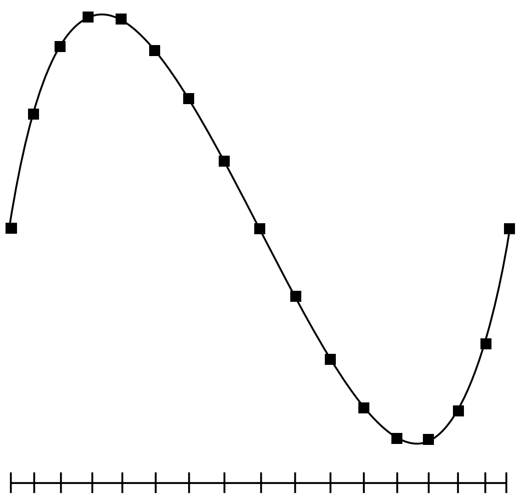

Figure 7.10 Subdivision of a curve by midpoint distance.

bisect. One criterion is to measure on a subinterval the variation between the curve and the line segment connecting the end points of the subinterval. If the subinterval is $[t_i, t_{i+1}]$, then the variation is

$$V = \int_{t_i}^{t_{i+1}} \left| \vec{x}(t) - \vec{\ell}_i(t) \right|^2 \, dt,$$

where $\vec{\ell(t)}_i$ is the line segment connecting the points $\vec{x}(t_i)$ and $\vec{x}(t_{i+1})$,

$$\vec{\ell}_i(t) = \vec{x}(t_i) + \frac{t - t_i}{t_{i+1} - t_i} \left(\vec{x}(t_{i+1}) - \vec{x}(t_i) \right). \tag{7.23}$$

The integration is shown for $i = 0$ to illustrate some optimizations that can be made in computing variation. Define $\vec{x}_j = \vec{x}(t_j)$ for $j = 0, 1$. The integral can be decomposed as

$$V = \int_{t_0}^{t_1} \vec{\ell} \cdot \vec{\ell} \, dt - 2 \int_{t_0}^{t_1} \vec{x} \cdot \vec{x}_0 \, dt - 2 \int_{t_0}^{t_1} \vec{x} \cdot (\vec{x}_1 - \vec{x}_0) \frac{t - t_0}{t_1 - t_0} \, dt$$
$$+ \int_{t_0}^{t_1} \vec{x} \cdot \vec{x} \, dt = V_1 - 2V_2 - 2V_3 + V_4.$$

than explicitly writing summation signs, if an expression contains a repeated index, the assumption is that the index is summed over the appropriate range of values. For example, if $A = [A_{ij}]$ is an $n \times n$ matrix and $\vec{x} = [x_j]$ is an $n \times 1$ vector, then the expression $A\vec{x}$ is written as $\sum_{j=0}^{n-1} A_{ij}x_j$ in the standard notation, but as $A_{ij}x_j$ using the summation convention. The index j is repeated, so an implied summation occurs over j. The second part of tensor notation specifies derivatives using indices. If $\vec{x}(\vec{p})$ is an $n \times 1$ vector-valued function of the $m \times 1$ vector $\vec{p}$, then the derivative of the ith component of $\vec{x}$ with respect to the jth component of $\vec{p}$ is denoted $x_{i,j}$. In tensor notation, indices before the subscripted comma refer to components and indices after the comma refer to derivatives. Second derivatives have two indices after the comma, third derivatives have three, and so on.

For a polynomial curve of degree at most three, the identities equivalent to Equation (7.24) for surfaces are

$$\begin{aligned} \vec{x}(s,t) &= \frac{1}{2}\left(\vec{x}(s+\delta,t) + \vec{x}(s-\delta,t) - \delta^2\vec{x}_{ss}(s,t)\right) \\ \vec{x}(s,t) &= \frac{1}{2}\left(\vec{x}(s,t+\delta) + \vec{x}(s,t-\delta) - \delta^2\vec{x}_{tt}(s,t)\right) \end{aligned} \tag{8.3}$$

Similarly, the identities equivalent to Equation (7.25) for surfaces are

$$\begin{aligned} \vec{x}_{ss}(s,t) &= \frac{1}{2}\left(\vec{x}_{ss}(s+\delta,t) + \vec{x}_{ss}(s-\delta,t)\right) \\ \vec{x}_{tt}(s,t) &= \frac{1}{2}\left(\vec{x}_{tt}(s,t+\delta) + \vec{x}_{tt}(s,t-\delta)\right) \end{aligned} \tag{8.4}$$

Now we will describe the algorithm for the block with parameter values $s \in [s_0, s_1]$ and $t \in [t_0, t_1]$. Define $s_m = (s_0 + s_1)/2$, $t_m = (t_0 + t_1)/2$, and $d = s_m - s_0 = t_m - t_0$. At each of the four corner points it is assumed that the following quantities are precomputed: $\vec{x}$, $\vec{x}_{ss}$, $\vec{x}_{tt}$, and $\vec{x}_{sstt}$. The subscripts indicate partial derivatives with respect to the listed variables. The formulas shown below are valid because of Equations (8.3) and (8.4).

For midpoints $(s_m, \bullet)$, where $\bullet$ is either t_0 or t_1:

$$\begin{aligned} \vec{x}_{ss}(s_m,\bullet) &= 0.5\left(\vec{x}_{ss}(s_0,\bullet) + \vec{x}_{ss}(s_1,\bullet)\right) \\ \vec{x}_{sstt}(s_m,\bullet) &= 0.5\left(\vec{x}_{sstt}(s_0,\bullet) + \vec{x}_{sstt}(s_1,\bullet)\right) \end{aligned}$$

$$\vec{x}_{tt}(s_m, \bullet) = 0.5\left(\vec{x}_{tt}(s_0, \bullet) + \vec{x}_{tt}(s_1, \bullet) - d^2\vec{x}_{sstt}(s_m, \bullet)\right)$$

$$\vec{x}(s_m, \bullet) = 0.5\left(\vec{x}(s_0, \bullet) + \vec{x}(s_1, \bullet) - d^2\vec{x}_{ss}(s_m, \bullet)\right).$$

For midpoints $(\bullet, t_m)$, where $\bullet$ is either s_0 or s_1:

$$\vec{x}_{tt}(\bullet, t_m) = 0.5\left(\vec{x}_{tt}(\bullet, t_0) + \vec{x}_{tt}(\bullet, t_1)\right)$$

$$\vec{x}_{sstt}(\bullet, t_m) = 0.5\left(\vec{x}_{sstt}(\bullet, t_0) + \vec{x}_{sstt}(\bullet, t_1)\right)$$

$$\vec{x}_{ss}(\bullet, t_m) = 0.5\left(\vec{x}_{ss}(\bullet, t_0) + \vec{x}_{ss}(\bullet, t_1) - d^2\vec{x}_{sstt}(\bullet, t_m)\right)$$

$$\vec{x}(\bullet, t_m) = 0.5\left(\vec{x}(\bullet, t_0) + \vec{x}(\bullet, t_1) - d^2\vec{x}_{tt}(\bullet, t_m)\right).$$

At the center point (s_m, t_m):

$$\vec{x}_{ss}(s_m, t_m) = 0.5\left(\vec{x}_{ss}(s_0, t_m) + \vec{x}_{ss}(s_1, t_m)\right)$$

$$\vec{x}_{tt}(s_m, t_m) = 0.5\left(\vec{x}_{tt}(s_m, t_0) + \vec{x}_{tt}(s_m, t_1)\right)$$

$$\vec{x}_{sstt}(s_m, t_m) = 0.5\left(\vec{x}_{sstt}(s_0, t_m) + \vec{x}_{sstt}(s_1, t_m)\right)$$

$$\vec{x}(s_m, t_m) = 0.5\left(\vec{x}(s_0, t_m) + \vec{x}(s_1, t_m) - d^2\vec{x}_{ss}(s_m, t_m)\right).$$

If L full subdivisions are performed, then $M_\ell = 2^\ell(2^{\ell-1} + 1)$ new midpoints and $C_\ell = 4^{\ell-1}$ new centers are generated at subdivision ℓ. The total number of midpoints is

$$M = \sum_{\ell=1}^{L} 2^\ell(2^{\ell-1} + 1) = \frac{2}{3}(4^L - 1) + 2(2^L - 1),$$

and the total number of center points is

$$C = \sum_{\ell=1}^{L} 4^{\ell-1} = \frac{1}{3}(4^L - 1).$$

for some phase angle ψ_1. The boundary conditions for c_0 are used to obtain $1 = \sin(\psi_0)/\sin\theta$ and $0 = \sin(\omega + \psi_0)/\sin\theta$, which are satisfied when $\psi_0 = \theta$ and $\omega = -\theta$. Thus,

$$c_0(t, \theta) = \frac{\sin((1-t)\theta)}{\sin\theta}.$$

The boundary conditions for c_1 are used to obtain $0 = \sin(\psi_1)/\sin\theta$, and $1 = \sin(\omega + \psi_1)/\sin\theta$, which are satisfied when $\psi_1 = 0$ and $\omega = \theta$. Thus,

$$c_1(t, \theta) = \frac{\sin(t\theta)}{\sin\theta}.$$

The spherical linear interpolation, abbreviated as *slerp*, is defined by

$$\operatorname{slerp}(t; q_0, q_1) = \frac{q_0 \sin((1-t)\theta) + q_1 \sin(t\theta)}{\sin\theta} \tag{9.3}$$

for $0 \le t \le 1$.

Although q_1 and $-q_1$ represent the same rotation, the values of $\operatorname{slerp}(t; q_0, q_1)$ and $\operatorname{slerp}(t; q_0, -q_1)$ are not the same. It is customary to choose the sign σ on q_1 so that $q_0 \cdot (\sigma q_1) \ge 0$ (the angle between q_0 and σq_1 is acute). This choice avoids extra spinning caused by the interpolated rotations.

For unit quaternions, slerp can be written as

$$\operatorname{slerp}(t; q_0, q_1) = q_0 \left(q_0^{-1} q_1\right)^t, \tag{9.4}$$

in which case $\operatorname{slerp}(0; q_0, q_1) = q_0$ and $\operatorname{slerp}(1; q_0, q_1) = q_0(q_0^{-1}q_1) = q_1$. The term $q_0^{-1}q_1 = \cos\theta + \hat{u}\sin\theta$, where θ is the angle between q_0 and q_1. The time parameter can be introduced into the angle so that the adjustment of q_0 varies uniformly with time over the great arc between q_0 and q_1. That is, $q(t) = q_0[\cos(t\theta) + \hat{u}\sin(t\theta)] = q_0[\cos\theta + \hat{u}\sin\theta]^t = q_0(q_0^{-1}q_1)^t$.

The derivative of slerp in the form of Equation (9.4) is a simple application of Equation (9.1):

$$\operatorname{slerp}'(t; q_0, q_1) = q_0(q_0^{-1}q_1)^t \log(q_0^{-1}q_1). \tag{9.5}$$

It is possible to add *extra spins* to the interpolation. Rather than interpolating the shortest great arc between the two quaternions, it is possible to wrap around the great circle n times before stopping at the destination quaternion. The formula in Equation (9.3) requires addition of a phase angle to θ,

$$\operatorname{SlerpExtra}(t; q_0, q_1) = \frac{q_0 \sin((1-t)(\theta + 2\pi n) + q_1 \sin(t(\theta + 2\pi n))}{\sin\theta}.$$

9.1.3 Spherical Cubic Interpolation

Source Code

Library

Core

Filename

Quaternion

The cubic interpolation of quaternions can be achieved using a method described by Boehm (1982), which has the flavor of bilinear interpolation on a quadrilateral. The evaluation uses an iteration of three slerps and is similar to the de Casteljau algorithm (Farin 1990). Imagine four quaternions p, a, b, and q as the ordered vertices of a quadrilateral lying on the unit hypersphere. Interpolate c along the great circle arc from p to q using slerp. Interpolate d along the great circle arc from a to b. Now interpolate the interpolations c and d to get the final result e. The end result is denoted *squad* and is given by

$$\text{squad}(t; p, a, b, q) = \text{slerp}(2t(1-t); \text{slerp}(t; p, q), \text{slerp}(t; a, b)). \tag{9.6}$$

The derivative of squad in Equation (9.6) is computed as follows. Let $u(t) = c_0(t, \theta)p + c_1(t, \theta)q$, $v(t) = c_0(t, \phi)a + c_1(t, \theta)b$, and $S(t) = \text{squad}(t; p, a, b, q) = c_0(2t(1-t), \psi(t))u(t) + c_1(2t(1-t), \psi(t))$, where $\cos(\theta) = p \cdot q$, $\cos(\psi) = a \cdot b$, and $\cos(\psi(t)) = u(t) \cdot v(t)$. The derivative is

$$S'(t) = c_0(2t(1-t), \psi(t))u'(t) + \frac{dc_0(2t(1-t), \psi(t))}{dt}u(t) + \\ c_1(2t(1-t), \psi(t))v'(t) + \frac{dc_1(2t(1-t), \psi(t))}{dt}v(t).$$

Using the chain rule and Equation (9.5), it can be shown that

$$\left.\frac{dc_0(2t(1-t), \psi(t))}{dt}\right|_{t=0} u(0) + \left.\frac{dc_1(2t(1-t), \psi(t))}{dt}\right|_{t=0} v(0) = \text{slerp}'(0; p, a)$$

and

$$\left.\frac{dc_0(2t(1-t), \psi(t))}{dt}\right|_{t=1} u(1) + \left.\frac{dc_1(2t(1-t), \psi(t))}{dt}\right|_{t=1} v(1) = \text{slerp}'(0; q, b).$$

Consequently, $S'(0) = p \log(p^{-1}q) + 2p \log(p^{-1}a)$ and $S'(1) = q \log(p^{-1}q) - 2q \log(q^{-1}b)$. The derivative of squad at the end points are

$$\begin{aligned} \text{squad}'(0; p, a, b, q) &= p[\log(p^{-1}q) + 2\log(p^{-1}a)] \\ \text{squad}'(1; p, a, b, q) &= q[\log(p^{-1}q) - 2\log(q^{-1}b)]. \end{aligned} \tag{9.7}$$

are to be viewable through a common geometric portal, then both regions must have a portal associated with them, and the two portals coexist in space in identical locations.

The regions and portals together can form an arbitrarily complex scene. For example, it is possible to stand in one region, look through a portal into an adjacent region, and see another portal from that region into yet another region. The rendering algorithm must draw the regions in a back-to-front order to guarantee the correct visual results. This is accomplished by constructing an abstract directed graph for which the regions are the graph nodes and the portals are directed graph edges. This graph is not the parent-child scene graph, but represents relationships about adjacency of the regions. Each region is represented as a scene graph node that contains enough state information to support traversal of the adjacency graph. The portals are represented by scene graph nodes but are not drawable objects. Moreover, the portal nodes are attached as children to the region nodes to allow culling of portals. If a region is currently being visited by the adjacency graph traversal, it is possible that not all portals of that region are in the view frustum (or part of the current set defined by the intersection of the frustum and additional portal planes). The continued traversal of the adjacency graph can ignore such portals, effectively producing yet another type of culling. Finally, the region nodes can have additional child nodes that represent the bounding planes of the regions (the walls, so to speak, if the region is a room) and the objects that are in the regions and that need to be drawn if visible. The pseudocode for rendering a convex region in the portal system is given below. The object `planeSet` is the current set of planes that the renderer uses for culling and (possibly) clipping. The planes maintained by the portal are those formed by the edges of the convex polygon of the portal and the current camera location.

```
void Render (Region region)
{
    if ( not region.beingVisited )
    {
        region.beingVisited = true;
        for ( each portal in region )
        {
            if ( portal.IsVisibleWithRespectTo(planeSet) )
            {
                planeSet.Add(portal.planes);
                Render(portal.adjacentRegion);
                planeSet.Remove(portalPlanes);
            }
        }
        Render(region.boundingPlanes);
        Render(region.containedObjects);
        region.beingVisited = false;
    }
}
```

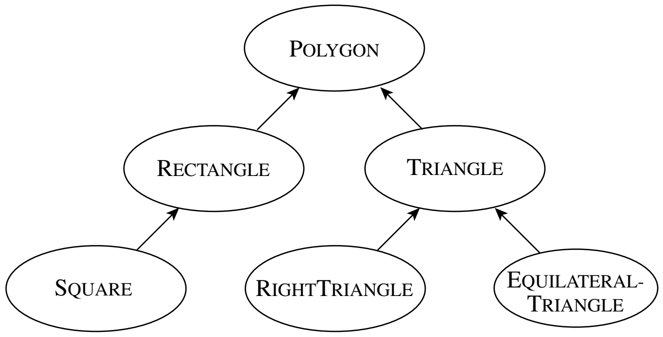

Figure A.1 Single-inheritance hierarchy.

stores a link to the base class (if any) to allow an application to determine if a class is inherited from another class. The simplest representation stores no class information and only the link to the base class. However, it is useful to store a string encoding the name of the class. In particular, the string will be used in the streaming system that is described later. The string may also be useful for debugging purposes in quickly identifying the class type.

```
class MgcRTTI
{
public:
    MgcRTTI (const char* acName, const MgcRTTI* pkBaseRTTI) :
        m_kName(acName)
    {
        m_pkBaseRTTI = pkBaseRTTI;
    }

    const MgcRTTI* GetBaseRTTI () const
    {
        return m_pkBaseRTTI;
    }

    const MgcString& GetName () const
    {
        return m_kName;
    }
```

Table B.1 Signs of the Sturm polynomials for $t^3 + 3t^2 - 1$ at various t values.

t	*Sign* $f_0(t)$	*Sign* $f_1(t)$	*Sign* $f_2(t)$	*Sign* $f_3(t)$	*Sign changes*
$-\infty$	$-$	$+$	$-$	$+$	3
-3	$-$	$+$	$-$	$+$	3
-2	$+$	0	$-$	$+$	2
-1	$+$	$-$	$-$	$+$	2
0	$-$	0	$+$	$+$	1
$+1$	$+$	$+$	$+$	$+$	0
$+\infty$	$+$	$+$	$+$	$+$	0

Table B.2 Signs of the Sturm polynomials for $(t-1)^3$ at various t values.

t	*Sign* $f_0(t)$	*Sign* $f_1(t)$	*Sign* $f_2(t)$	*Sign changes*
$-\infty$	$-$	$+$	0	1
0	$-$	$+$	0	1
$+\infty$	$+$	$+$	0	0

The Sturm sequence is $f_0(t) = t^3 + 3t^2 - 1$, $f_1(t) = 3t^2 + 6t$, $f_2(t) = 2t + 1$, and $f_3 = 9/4$. Table B.1 lists the signs of the Sturm polynomials for various t values. Letting $N(a, b)$ denote the number of real-valued roots on the interval (a, b), the table shows that $N(-\infty, -3) = 0$, $N(-3, -2) = 1$, $N(-2, -1) = 0$, $N(-1, 0) = 1$, $N(0, 1) = 1$, and $N(1, \infty) = 0$. Moreover, the number of negative real roots is $N(-\infty, 0) = 2$, the number of positive real roots is $N(0, \infty) = 1$, and the total number of real roots is $N(-\infty, \infty) = 3$.

The next example shows that the number of polynomials in the Sturm sequence is not necessarily the degree$(f) + 1$. The function $f(t) = (t-1)^3$ has a Sturm sequence $f_0(t) = (t-1)^3$, $f_1(t) = 3(t-1)^2$, and $f_2(t) \equiv 0$ since f_1 exactly divides f_0 with no remainder. Table B.2 lists sign changes for f at various t values. The total number of real roots is $N(-\infty, \infty) = 1$.

B.5.2 Methods in Many Dimensions

Root finding in many dimensions is a more difficult problem than it is in one dimension. Two simple algorithms are summarized here: bisection and Newton's method.

About the Author

David Eberly is the President of Magic Software, Inc. *(www.magic-software.com),* a company known for its Web site that offers free source code and documentation for computer graphics, image analysis, and numerical methods. Previously he was the Director of Engineering at Numerical Design Limited, the company responsible for the real-time 3D game engine, NetImmerse. His background includes a B.A. degree in mathematics from Bloomsburg University, M.S. and Ph.D. degrees in mathematics from the University of Colorado at Boulder, and M.S. and Ph.D. degrees in computer science from the University of North Carolina at Chapel Hill. He is co-author with Philip Schneider of the forthcoming *Geometry Tools for Computer Graphics,* to be published by Morgan Kaufmann.

As a mathematician, Dave did research in the mathematics of combustion, signal and image processing, and length-biased distributions in statistics. He was a research associate professor at the University of Texas at San Antonio with an adjunct appointment in radiology at the U.T. Health Science Center at San Antonio. In 1991 he gave up his tenured position to retrain in computer science at the University of North Carolina. During his stay at U.N.C., MAGIC (My Alternate Graphics and Image Code) was born as an attempt to provide an easy-to-use set of libraries for image analysis. Since its beginnings in 1991, MAGIC has continually evolved into the "net library" that it currently is, now managed by the company Magic Software, Inc. After graduating in 1994, he remained for one year as a research associate professor in computer science with a joint appointment in the Department of Neurosurgery working in medical image analysis. His next stop was the SAS Institute working for a year on SAS/Insight, a statistical graphics package. Finally, deciding that computer graphics and geometry were his real calling, Dave went to work for Numerical Design Limited, then later to Magic Software, Inc. Dave's participation in the newsgroup *comp.graphics.algorithms* and his desire to make 3D graphics technology available to all are what has led to the creation of this book. The evolution of Magic will continue and the technology transfer is not yet over.

About the CD-ROM

Contents of the CD-ROM

The accompanying CD-ROM contains source code that illustrates the ideas in the book. A partial listing of the directory structure is

```
/Wild Magic 0.4
    LinuxReadMe.txt
    WindowsReadMe.txt
    /Linux
        /WildMagic
            /Applications
            /Include
            /Library
            /Licenses
            /Object
            /SourceFree
            /SourceGameEngine
    /Windows
        /WildMagic
            /Applications
            /Include
            /Library
            /Licenses
            /SourceFree
            /SourceGameEngine
            /Tools
```

The read-me files contain the installation instructions and other notes. The path `Windows/WildMagic` contains the distribution for use on a computer whose operating system is one of Windows 95, Windows 98, Windows NT, or Windows 2000. The path `Linux/WildMagic` contains the distribution for use on a computer whose operating system is Linux. Compiled source code is already on the CD-ROM. The application directories, located in `Applications`, contain compiled executables that are ready to run.

The distributions are nearly identical. The Windows text files have lines that are terminated by carriage return and line feed pairs whereas the Linux text files are terminated by line feeds. The Windows distribution contains an OpenGL renderer and a Win32 application layer, both dependent on the operating system. The Windows distribution also has a rudimentary software renderer and it has a tool for converting bitmap (*.bmp) files to Magic image files (*.mif). Both the Windows and Linux distributions contain an OpenGL renderer and an application layer that is dependent on GLUT. The Windows code is supplied with Microsoft Developer Studio Projects

(*.dsp) and Microsoft Developer Studio Workspaces (*.dsw). The Linux code is supplied with make files. The other portions of the distributions are the same.

License Agreements

Each source file has a preamble stating which of two license agreements governs the use of that file. The license agreements are located in the directory `Licenses`. The source code in the path `SourceGameEngine` is governed by the license agreement `Licenses/3DGameEngine.pdf`. The remaining source code is governed by the license agreement `Licenses/free.pdf`. All source code may be used for commercial or noncommercial purposes subject to the constraints given in the license agreements.

Installation on a Windows Sytem

These directions assume that the CD-ROM drive is drive `D` and the disk drive to which the contents are to be copied is drive `C`. Of course you will need to substitute the drive letters that your system is using. Copy the CD-ROM subtree `D:\Wild Magic 0.4\Windows\WildMagic` to `C:\SomePath\WildMagic`. Since the files are copied as read-only, execute the following two commands, in order, from a command window: `cd C:\SomePath\WildMagic` and `attrib -r *.* /s`. The distribution comes precompiled, but if you want to rebuild it, open the workspace `C:\SomePath\WildMagic\BuildAll.dsw` and select the `BuildAll` project (the default one that shows up in the project list box is `BezierSurface`). Build both the `Debug` and `Release` configurations. This builds the `SourceFree`, `SourceGameEngine`, and `Application` source trees, in that order. Each of the directories `SourceFree`, `SourceGameEngine`, and `Applications` has a top level workspace to build only those pieces.

Installation on a Linux System

Mount the CD-ROM drive by: `mount -t iso9660 /dev/cdrom /mnt`. If your desired top level directory is `/HomeDirectory/SomePath` (substitute the actual path to your home directory), and if your current working directory is `/HomeDirectory/SomePath` then use `cp "/mnt/Wild Magic 0.4/Linux/WildMagic" -r .` to generate the source tree `/HomeDirectory/SomePath/WildMagic`. Note that "`.`" is the last argument of "`cp`". Since the files are copied as read-only, execute the following two commands, in order, (assumes your current working directory is still `/HomeDirectory/SomePath`): `cd WildMagic` and `chmod a+rw -R *`. The distribution comes precompiled, but if you want to rebuild it, run `make` on the makefile in the `WildMagic` subdirectory. Build the `Debug` configuration by `make CONFIG=Debug` and the Release configuration by `make CONFIG=Release`. Each of the directories `SourceFree`, `SourceGameEngine`, and `Applications` has a top-level makefile to build only those pieces.

You need some form of OpenGL and GLUT on your machine. I downloaded Mesa packages from the Red Hat site, `Mesa-3.2-2.i686.rpm`, `Mesa-devel-3.202.i686.rpm`, and `Mesa-glut-3.1-1.i686.rpm`, and used the Gnome RPM tool to install them. I told the tool to ignore the fact that GLUT is 3.1 and Mesa is 3.2. The installation puts the libraries in `/usr/X11R6/lib` and the headers in `/usr/X11R6/include`. The makefiles for applications use the libraries `libGL.la`, `libGLU.la`, and `libglut.la`.