Spline Interpolation in Curved Space

Steven A. Gabriel

Gabriel Electronics Consulting
Orlando, Florida

James T. Kajiya

California Institute of Technology
Pasadena, Calif

ABSTRACT. Cubic splines are curves of minimal acceleration in Euclidean space. In this paper, we generalize spline interpolation to curves of minimal acceleration in non-Euclidean spaces. These spaces arise when we interpolate orientations or configurations of articulated objects through time, or when a path is constrained to lie on a curved surface. Thus the generalized splines are potentially useful in keyframe animation and robotics. They are computationally more intensive than cubic splines, requiring a differential equation to be solved instead of calculating cubic polynomials. We present a method for solving that equation.

ACKNOWLEDGEMENT Thanks to Lance Williams for suggesting the problem to us and to Al Barr for mathematical discussions.

§1 Introduction

Cubic splines are a fundamental tool in computer graphics. They are widely used to make the shapes of synthetic objects and to define the paths the objects follow in animated sequences.

The cubic spline has difficulties, however, when it is used to interpolate orientations. These stem from the non-commutativity of rotation. This lack of commutativity is an expression of the *curvature* of the space of rotations. Coordinate systems in curved spaces are necessarily non-uniform and may even have singularities such as the north and south poles of the latitude and longitude system on a sphere. Cubic splines are tied to whatever coordinate system was picked to describe the curved space and are thus tugged and pulled by bendings and stretchings of the system that have nothing to do with the distance measure of the underlying space. For example, traveling 180 degrees of longitude on the Earth's equator will carry you 12,500 miles, but the same thing done at the North pole will not take you anywhere at all!

We thus seek a spline whose description depends only on the distance between points in a curved space and is independent of the particular coordinates used there. We have derived an equation whose solution is a new generalized spline that has this property. If the space is flat this spline is identical to the ordinary cubic spline.

The new spline arises from the original definition of a cubic spline: the curve interpolating a set of points that has the least total squared magnitude of acceleration of any smooth curve passing through those points. Curved spaces can be thought of as surfaces in higher dimensions. In our definition splines on a surface minimize acceleration also, but only that component that is tangent to the surface.

The surfaces these splines exist in are quite general. They are the curved spaces known mathematically as Riemannian manifolds. Manifolds are the setting for many problems in physics, especially dynamics and general relativity. We draw on some of the techniques of those fields here. A helpful introduction is found in reference [5].

whenever two charts overlap there is a smooth and invertable mapping between them. This mapping is called the chart transition function. A Riemannian manifold is one that has a metric defined upon it. The definition of a manifold is motivated by the fact that a space may be so highly curved that it cannot be completely covered with a single coordinate chart. Even the sphere is like this. There is always at least one point where a coordinate system that attempts to cover the sphere will break down, such as the north or south pole of the spherical coordinate system.

2.2 $SO(3)$ and the Quaternions

The topology of $SO(3)$ is related to that of S^3, the surface of a sphere in four dimensions, by a map to the four components of a quaternion

$$\cos\frac{\theta}{2}, \qquad n_1 \sin\frac{\theta}{2}, \qquad n_2 \sin\frac{\theta}{2}, \qquad n_3 \sin\frac{\theta}{2}$$

where θ is the angle of rotation about an axis represented by a unit vector whose components are (n_1, n_2, n_3). This map and its inverse are described in more detail by Shoemake [11]. The unit norm quaternions all lie on a sphere of radius one in four dimensions. Since S^3 has a simpler topology than $SO(3)$, it is easier to calculate interpolating curves on this sphere with our algorithm and then transform back to $SO(3)$ with the inverse of the quaternion map.

§3 Minimization

A standard procedure in calculus is finding a local minimum of a function $f(x)$. All points where $f'(x) = 0$ are candidates. If $f''(x) > 0$ at one of these points we have a local minimum, a place where the function value is less than that at any nearby x.

This procedure can be generalized to integrals of a function defined on curves. In physics these functions are sometimes called Lagrangians. We want to find a whole curve that minimizes an integral, not just a single point that minimizes a function value. The Lagrangian may depend on the velocity and acceleration of the curve as well as its position. The curve is found using the following

Theorem. If we have a Lagrangian $L(y, y', y'')$ then the curve in R^n from $y(a)$ to $y(b)$ with initial velocity $y'(a)$ and final velocity $y'(b)$ that extremizes the integral of this Lagrangian

$$\int_a^b L(y, y', y'')dt$$

satifies the differential equation

$$\frac{\partial L}{\partial y} - \frac{d}{dt}(\frac{\partial L}{y'}) + \frac{d^2}{dt^2}(\frac{\partial L}{y''}) = 0.$$

Proof. This is the Euler-Lagrange equation and proof can be found in [3] or any text on calculus of variations. Methods for determining if an extremum is a minimum are also found there. ∎

In the original derivation of the cubic spline a curve was sought that minimized the square integral of the acceleration. The Lagrangian in this case is $L(y, y', y'') = (y'')^2 = y'' \cdot y''$. Inserting this in the Euler-Lagrange equation we see that the only non zero partial derivatives are those with respect to y''

$$\frac{d^2}{dt^2}(\frac{\partial(y'' \cdot y'')}{y''}) = \frac{d^2}{dt^2}(2y'') = 2\frac{d^4 y}{dt^4}$$

and we get the vector differential equation

$$\frac{d^4y}{dt^4} = 0.$$

The only function whose fourth derivative is zero is the third degree polynomial $at^3 + bt^2 + ct + d$, where the coefficients are n-vectors. The four coefficients are chosen to match the endpoint boundary conditions. In this case the solution curve is known to be a local minimum since the L is a positive definite quadratic function.

We now want to constrain the solution curve to lie on a surface. One way is to add Lagrange multipliers to L, but this results in an in an equation with vectors whose dimension is that of the space the surface is embedded in which must be solved in conjunction with the surface equation itself. The Euler-Lagrange equation in effect only tells what the curve would look like *if* it happened to be on the surface already.

Instead, we take an intrinsic approach described below.

§4 The Fundamental Equation

This section presents our fundamental equation for spline interpolation on Riemannian manifolds. We derived it using a variational definition for the interpolating curve. In the Euclidean case above, the cubic spline interpolant was that path which extremized a variation whose Lagrangian is the square of the acceleration. We extended this process to manifolds with metric by calculating the Euler equation for a more general Lagrangian: the square of the *covariant* acceleration. When the manifold is ordinary Euclidean space with the standard cartesian coordinatization, our approach reduces back to the familiar cubic spline.

In the derivation we used modern coordinate free notation which considerably reduced the effort in obtaining the following

Theorem. A spline interpolant in a Riemannian manifold M is an integral curve of a vector field V which satisfies the following *fundamental equation*:

$$\nabla_V \nabla_V \nabla_V V - R(V, \nabla_V V)V = 0 \qquad \frac{d\gamma}{dt} = V$$

where R is the Riemann curvature tensor of M. Boundary conditions for this equation are the specification of four quantities: $\gamma(a)$, $\frac{d\gamma}{dt}(a)$, $\gamma(b)$ and $\frac{d\gamma}{dt}(b)$.

Proof. The proof has been thankfully banished to appendix 1. ▮

Note that this equation describes the spline curve as a *geometric object* within the manifold. The path determined is the same no matter what coordinate system is used.

If the manifold is embedded as a surface in a higher dimensional flat space the covariant derivative is the component of the ordinary derivative that is tangent to the surface. The covariant acceleration $\nabla_V V$ of a curve thus senses that part of acceleration that occurs within a surface and ignores the part that serves only to keep the curve on the surface.

The Riemann tensor $R(V, X)V$ measures the curvature of the manifold in the direction of V experienced as the turning of a vector X carried in that direction.

In the cubic spline equation of the last section the third derivative of velocity was zero. Here we have a similar form with the third covariant derivative of velocity being driven by the curvature of the manifold.

4.1 Special Cases of the Equation

We now show various forms of the equation in special manifolds.

4.1.1 Flat Manifolds

Flat manifolds are manifolds in which the Riemann curvature tensor is zero. In this case the equation reduces to:

$$\nabla_V \nabla_V \nabla_V V = 0.$$

A flat manifold is locally diffeomorphic to R^n. Thus, the fundamental equation gives a simple equation for spline interpolation of intrisically flat spaces which happen to be coordinatized with a curvilinear coordinate system, e.g. R^3 in spherical coordinates.

4.1.2 Product Manifolds

Both the covariant derivative and the Riemann curvature tensor split for product manifolds. This gives rise to the product theorem:

Theorem. To solve the fundamental equation in a product manifold $M = M_1 \times M_2$, it is sufficient to solve the equation in each factor M_i separately to give $\gamma_i(t)$. The full solution is then simply $\gamma(t) = (\gamma_1(t), \gamma_2(t))$.

This is the reason that spline interpolation is so easy in R^n with the standard coordinate system. In the standard coordinates covariant derivatives become ordinary derivatives and because R^n is a product the interpolant splits into n one dimensional interpolations in R^1. These are determined by

$$\frac{d^3 v_i}{dt^3} = 0 \qquad \frac{dx_i}{dt} = v \qquad 1 \le i \le n$$

These differential equations are identical to the ones in section 2 and thus have a cubic polynomial in t as the solution for each component x_i of the path.

4.2 Boundary Conditions

Both the cubic spline and the spline on a manifold satisfy fourth degree differential equations. These curves thus have four quantities that must be specified as conditions for a solution. With this relation, analogs of all the existing means of controlling interpolating cubic splines exist on manifolds.

An interpolating spline curve must pass through a given set of points called knots. Between the knots are segments of the curve. The knot positions supply two of the four conditions needed to determine a segment. Spline algorithms differ in how they provide the other two.

In the Bezier curve the tangent vectors are given at the knots by way of deltas between control points. These tangent vectors are sufficient as the two remaining conditions. C^1 continuity between segments is assured by simply having an incoming tangent knot equal the outgoing one at each interior knot.

In this formulation we can calculate all the segments independently, since they are tied to the given knot positions and velocities. No information propagates from one segment to the next. On manifolds we make Bezier splines by calculating solutions of the fundamental equation between adjacent knots with position and tangent known at each knot. Each segment is then a separate two point boundary value problem with two initial and two final conditions.

The global interpolating spline has velocity specified only at the start and the end knots. All the velocities at interior knots are left free. A condition added to lock them into a unique configuration is C^2 continuity. The price we pay for not having to give the interior velocities is non-locality, all the segments have to be solved at once. This is because the shape of one segment propagates velocity and acceleration constraints into its neighbors. In the cubic case we find all velocities at the interior knots by solving a tri-diagonal linear system. On a manifold we solve a single multi-point boundary value problem where the interior boundary equations express the knot positions and the coupling of the first and second derivatives of adjacent segments.

§5 Numerical Solution of the Equation

Numerical analysis on differentiable manifolds is a subject which has not received wide treatment. A difficulty arises because in general a manifold cannot be covered by only one coordinate chart. Any technique for integrating a differential equation on a manifold must include a means for carrying the solution across chart boundaries.

We show in this section how to reduce the coordinate free form of the equation to an ordinary nonlinear differential equation in the chart coordinates.

5.1 Local vs. Global Solutions

We will be concerned primarily with solving the local rather than the global variational problem, i.e. we will be extremizing rather than minimizing the variational problem. Besides being simpler, this has a number of advantages. In practice, a local solution is often more desirable than the global solution. Consider the case of interpolating orientations of a rigid object. Let the desired path be one of a full revolution about the z-axis with zero initial and final velocities. Because the initial and final orientations and velocities are identical, a global solution would give a path which is constant in time—no movement at all. But the desired solution is a local extremum in the neighborhood of a path which spins the object one revolution.

We show later how such a desired local solution may be specified.

5.2 Choosing State Variables

The choice of state variables in ordinary equations is usually not critical. However, if the state variables are not chosen with care when considering manifolds, it may be very difficult to cross patch boundaries. For this reason, we have found that only *intrinsic* quantities are suitable as state variables.

When solving the fundamental equation, it would be customary to express covariant differentiation and the curvature tensor in terms of the chart coordinates and their derivatives as state variables. However, when crossing from one coordinate chart to another, the second and higher derivatives are not tensors, so the chart transition equations are very complicated. However if one chooses the chart coordinates X and their higher *covariant* derivatives as state variables, crossing coordinate charts becomes tractable: all the state variables are then tensors. For this reason we choose as state variables:

$$X(t), \qquad V = \frac{dX}{dt}, \qquad U = \nabla_V V, \qquad W = \nabla_V U$$

With these state variables the fundamental equation becomes a coupled set of four first order equations.

$$\begin{aligned} \frac{dX}{dt} &= V \\ \nabla_V V &= U \\ \nabla_V U &= W \\ \nabla_V W &= R(V,U)V \end{aligned}$$

5.3 Coordinatizing the Equation

To actually compute the path numerically we must convert these equations to coordinate form for each chart the path crosses. In what follows we use Einstein summation notation.

We start with the metric tensor g_{ab} in each chart, which in general is a function of position. If the manifold is an n dimensional parametric surface embedded in R^m with Y as a coordinate vector within the surface, then the metric is

$$g_{ab} = \sum_{\nu=1}^{m} \frac{\partial y^\nu}{\partial x^a} \frac{\partial y^\nu}{\partial x^b}.$$

We next calculate the Schwartz-Christoffel symbols:

$$\Gamma^e_{ca} = \frac{1}{2} g^{eb}(\partial_c g_{ba} + \partial_a g_{cb} - \partial_b g_{ca}).$$

where $\partial_i = \frac{\partial}{\partial x_i}$. The covariant derivative in components is then

$$\nabla_V Y = \frac{dy^a}{dt} + \Gamma^a_{bc} y^b \frac{dx^c}{dt}$$

and the curvature tensor becomes

$$R(X,Y)Z = R^d_{abc} z^a x^b y^c$$

where

$$R^d_{abc} = \partial_c \Gamma^d_{ab} - \partial_b \Gamma^d_{ac} + \Gamma^e_{ab} \Gamma^d_{ec} - \Gamma^e_{ac} \Gamma^d_{eb}$$

Substituting these things into the fundamental equation and using the state variables x^i, v^i, u^i, w^i chosen above we obtain the coordinatized form of the equation

$$\begin{aligned}
\frac{dx^i}{dt} &= v^i \\
\frac{dv^i}{dt} &= u^i - \Gamma^i_{ab} v^a v^b \\
\frac{du^i}{dt} &= w^i - \Gamma^i_{ab} u^a v^b \\
\frac{dw^i}{dt} &= R^i_{abc} v^a v^b u^c - \Gamma^i_{ab} w^a v^b
\end{aligned}$$

5.4 Solving Ordinary Differential Equations on Manifolds

Solving initial value problems in manifolds is relatively simple. One evolves the equation until the state nears the boundary of the current coordinate chart. The chart transition function and its Jacobian are then used to transform the state variables to the new chart.

Solving boundary value problems is more difficult. It is usual that the chart transition functions are nonlinear. Because of this an equation which may be linear in one chart is almost certainly nonlinear in another. We are thus led to consider algorithms for solving differential equations with nonlinear boundary conditions.

We present a well behaved finite difference method modified to handle chart crossings.

5.5 Finite Differences

This treatment is taken from Keller [6, 7]. He solves a nonlinear differential equation of the form

$$N(Y) = \frac{dY(t)}{dt} - f(t, Y(t)) = 0 \qquad (a \le t \le b)$$
$$g(Y(a), Y(b)) = 0$$

where N is a nonlinear operator which measures how far a function is from being a solution to the equation and Y is an n-dimensional function of t. The boundary conditions g are also nonlinear. N can be approximated by a finite difference operator N_h on a grid of $J+1$ points with spacing $h = (b-a)/J$.

$$N_h(u_j) = \frac{1}{h}[u_j - u_{j-1}] - f(t_{j-\frac{1}{2}}, \frac{1}{2}[u_j + u_{j-1}]) = 0 \qquad 1 \le j \le J$$

Let U be the vector $(u_0, u_1, \cdots, u_J)^T$. Combining N_h with g to make a single function Φ we have

$$\Phi(U) = \begin{pmatrix} g(u_0, u_J) \\ N_h(u_1) \\ \cdots \\ N_h(u_J) \end{pmatrix}$$

The finite difference approximation to the solution of $N(Y) = 0$ is a vector U such that $\Phi(U) = 0$. This is a coupled set of algebraic equations which may be solved by Newton's method. Linearizing Φ about the current solution U^ν we have the iteration

$$A^\nu[U^{\nu+1} - U^\nu] = -\Phi(U^\nu)$$

where A^ν is the Jacobian matrix of Φ given by

$$A^\nu = \begin{pmatrix} B_a & & & & B_b \\ -L_1^\nu & R_1^\nu & & & \\ & & \cdots & & \\ & & & -L_J^\nu & R_J^\nu \end{pmatrix}$$

with

$$L_j^\nu = h^{-1}I + \frac{1}{2}\frac{\partial f}{\partial y}(t_{j-\frac{1}{2}}, \frac{1}{2}[u_j^\nu + u_{j-1}^\nu])$$
$$R_j^\nu = h^{-1}I - \frac{1}{2}\frac{\partial f}{\partial y}(t_{j-\frac{1}{2}}, \frac{1}{2}[u_j^\nu + u_{j-1}^\nu])$$
$$B_a = \frac{\partial g(u_0^\nu, u_J^\nu)}{\partial u_0} \qquad B_b = \frac{\partial g(u_0^\nu, u_J^\nu)}{\partial u_J}$$

5.5.1 Solving the Bidiagonal System A^ν

Because N_h only couples adjacent components u_j and u_{j+1} of U, matrix A^ν is block bi-diagonal, except for B_b in the upper right corner. All other submatrices are zero. The matrix A is of order $n(J+1)$, but it can be solved by an algorithm which runs up the main diagonal solving $J+1$ systems of order n, then down the diagonal performing back substitutions. For the system $AZ = V$ the procedure begins by initializing an $n \times n$ matrix M to the identity and an n-vector w to zero.

$$M = I_{n\times n} \qquad w = 0_n$$

The upward pass calculates in sequence the vectors δ_j and w and matrices P_j and M

$$\delta_j = R_j^{-1} v_j \qquad w = w + M\delta_j \qquad P_j = R_j^{-1} L_j \qquad M = MP_j \qquad J \geq j \geq 1.$$

The boundary conditions enter at $j = 0$

$$\delta_0 = v_0 - B_b w \qquad z_0 = (B_a + B_b M)^{-1}\delta_0.$$

Finally, the downward pass produces the result vector

$$z_j = \delta_j + P_j z_{j-1} \qquad 1 \leq j \leq J.$$

In the determination of δ_j and z_0, the linear systems should be solved directly instead of matrix inverting and multiplying.

More accurate methods are presented in [7, 8]. They rearrange the matrix into a tridiagonal system that has no off-diagonal B_b term. This tridiagonal system is reminiscent of the one which results when fitting the global C^2 cubic spline to a set of points. Multi-point boundary conditions are discussed. Also, Richardson extrapolation is used to obtain very accurate solutions with a coarse mesh.

5.6 Starting the Iteration

The Newton iteration begins with a guess U^0. We usually take this to be a geodesic between the endpoints, the iterations then bend the curve to fit the tangent specifications at those points. We also can control which local minimum is found by choice of the initial guess. For example, on the sphere we can converge to a curve that wraps around the sphere three times by starting with a great circle route that wraps three times.

5.6.1 Modification for Multiple Charts

If the solution curve crosses multiple chart boundaries we must modify the matrix A. If k is the index of a point where the curve crosses from a chart p to chart q and φ_{pq} is the function that gives the coordinates in chart q of a point located with coordinates of chart p we modify $N_h(u_{k+1})$ to

$$N_h(u_{k+1}) = \frac{1}{h}[u_{k+1} - \varphi_{pq}(u_k)] - f(t_{k+\frac{1}{2}}, \frac{1}{2}[u_{k+1} + \varphi_{pq}(u_k)])$$

The t parameter of f indicates that f is evaluated on chart q. When Φ is linearized with this modification and we have the condition that the state variables are all covariant quantities we get

$$L_{k+1} = \left[h^{-1}I + \frac{1}{2}\frac{\partial f}{\partial y_q}\left(t_{k+\frac{1}{2}}, \frac{1}{2}[u_{k+1} + \varphi_{pq}(u_k)]\right)\right]\frac{\partial \varphi_{pq}}{\partial y_p}$$
$$R_{k+1} = h^{-1}I + \frac{1}{2}\frac{\partial f}{\partial y_q}\left(t_{k+\frac{1}{2}}, \frac{1}{2}[u_{k+1} + \varphi_{pq}(u_k)]\right)$$

The other L and R submatrices are unchanged and the system can be solved as before.

§6 Spline Interpolation on S^n

Here we specialize the fundamental equation to one of the simplest curved spaces, the n-sphere.

6.1 Coordinate Charts of S^n

The n-sphere S^n has many different possible charts. We use a particularly simple set which covers the sphere and provides large regions of overlap. Consider S^n embedded in R^{n+1}. Slice the sphere with a plane perpendicular to the i^{th} coordinate axis of R^{n+1} and orthogonally project onto the plane images of the the two hemispheres thus formed. The chart mapping from S^n in R^{n+1} to R^n is obtained by just dropping the i^{th} coordinate. Repeat this for all the axes of R^n. This gives a set of $2(n+1)$ charts.

Any two charts that do not project onto the same plane will overlap on the sphere. If chart p was formed by projection onto the i plane and chart q onto the j plane the transition function between them is given by

$$\varphi_{pq}(x_1,\ldots,x_i,\ldots,x_n) = (x_1,\ldots,\hat{x}_i,\ldots,\sqrt{1-x\cdot x},\ldots,x_n)$$

Where the circumflex means to delete the i^{th} coordinate and the calculated quantity is inserted into the j^{th} coordinate.

For this coordinate system the metric tensor and its inverse are

$$g_{ab} = \delta_{ab} + \frac{x_a x_b}{1 - x^c x_c} \qquad g^{ab} = \delta^{ab} - x^a x^b.$$

The Christoffel coefficients and the curvature tensor are calculated accordingly.

6.2 The Equation in Coordinates

Calculating covariant derivatives and the Riemann curvature tensor in the above coordinate charts, the fundamental equation has the form

$$\frac{d}{dt}\begin{pmatrix} x \\ v \\ u \\ w \end{pmatrix} = \begin{pmatrix} v \\ u - (v\cdot v + \beta x \cdot v)x \\ w - \alpha x \\ -(v\cdot v + \beta x\cdot v)u + \alpha v - (\beta(x\cdot w)x + w\cdot v)x \end{pmatrix}$$

Recall that x, v, u, w are position, velocity, the covariant acceleration and its covariant derivative. These are n-vectors. The scalar quantities α and β are given by:

$$\alpha = u\cdot v + \beta x \cdot u$$
$$\beta = \frac{x\cdot v}{1 - x\cdot x}$$

To solve the equation with the method of section 5 we need its Jacobian matrix. This is given in appendix 2. The matrix for S^3 has 144 terms and we used a computer algebra system to aid in its calculation.

We have coded the difference method described above for the sphere equation in about 100 lines of APL2 on an IBM 4341. In all cases the method converged in under eight iterations taking a few tens of seconds.

§7 Conclusion

We have solved the problem of spline interpolation on a manifold with metric. The method we have presented here requires the solution of a nonlinear two point boundary value problem. On ordinary computers this is not a real time task but it is short compared to the time needed to render a single frame of moderate complexity.

For interactive work simpler and faster procedures are needed. One such procedure is given by Shoemake [11], in which a very efficient solution for coordinate free Bezier interpolation on the rotation group $SO(3)$ is developed through the use of algebraic operations on quaternions. However, his solution paths on this manifold differ from ours considerably. The algebraic scheme does not appear to satisfy any natural variational criterion and it is only appropriate for manifolds in which a geodesic can be easily calculated.

Further, we point out that the modified method for solving two point boundary value problems in manifolds introduced here is likely to be of value in solving optimal control problems on manifolds and in simulating the dynamics of complex articulated mechanical systems. As we have pointed out, these arise quite commonly in robotics and computer animation.

§8 References

[1] A. H. Barr (1984),*Local and global deformations of solid primitives.* SIGGRAPH '84 proceedings.

[2] R. Bishop and S. Goldberg(1968),*Tensor Analysis on Manifolds.* Dover, New York.

[3] A. Bryson and Y. Ho(1975),*Applied Optimal Control.* Hemisphere Publishing, New York.

[4] P.J. Davis(1963),*Interpolation and Approximation.* Dover, New York.

[5] J. Foster and J. Nightingale(1979),*A Short Course in General Relativity.* Longman, New York.

[6] H.B. Keller(1968),*Numerical Methods for Two Point Boundary Value Problems.* Blaisdell, Waltham, Mass.

[7] H.B. Keller(1974),*Accurate Difference Methods for Nonlinear Two Point Boundary Value Problems.* SIAM J. Numerical An. vol. 11 no. 2, April 1974.

[8] H.B. Keller(1976),*Numerical Solution of Two Point Boundary Value Problems.* SIAM, CBMS-NSF Regional conference series.

[9] C.W. Misner, K.S. Thorne, and J.A. Wheeler (1973),*Gravitation.* W.H.Freeman and Co., San Francisco.

[10] B. F. Schutz(1980), *Geometric Methods of Mathematical Physics.* Cambridge University Press, New York

[11] K. Shoemake(1985), *Animating Rotations with Quaternion Curves.* SIGGRAPH '85 Conference Proceedings.

[12] M. Spivak(1974), *A Comprehensive Introduction to Differential Geometry.* 5 vols. Publish or Perish Press. Berkeley.

[13] J. A. Thorpe(1979), *Elementary Topics in Differential Geometry.* Springer-Verlag, New York

Appendix 1: Proof of the Fundamental Equation

The notation and background to follow our proof can be found in Bishop and Goldberg. We use $D_X Y$ to denote the covariant derivative here instead of the more common $\nabla_X Y$ used in the rest of the paper and the physics references.

Theorem. In a Riemannian manifold M using the metric connection, the curve $\gamma : [a,b] \rightarrow M$ which extremizes the integral of the squared magnitude of the covariant derivative of velocity

$$J = \frac{1}{2}\int_a^b \langle D_{\gamma_*}\gamma_*(t), D_{\gamma_*}\gamma_*(t)\rangle dt$$

is an integral curve of the vector field V which satisfies

$$D_V D_V D_V V - R(V, D_V V)V = 0$$

where R is the Riemann curvature tensor of M and $\gamma_*(t)$ is the tangent vector to γ at t. Boundary conditions for this equation are the specification of four quantities: $\gamma(a), \gamma_*(a), \gamma(b)$ and $\gamma_*(b)$.

Proof.

Let $Q : [a,b] \times [-c,+c] \rightarrow M$ be a C^∞ variation on γ with fixed endpoints and fixed tangent vectors at those endpoints. This means $Q(t,0) = \gamma(t)$ and $Q(a,s) = \gamma(a)$, $Q(b,s) = \gamma(b)$, $Q_*(a,s) = \gamma_*(a)$, $Q_*(b,s) = \gamma_*(b)$ for all $s \in [a,b]$.

Let $V = Q_*\partial_1$ and $U = Q_*\partial_2$, the vector fields of the longitudinal and transverse variations respectively. These are coordinate vector fields, therefore $[U,V] = 0$. With $V(t,0) = \gamma_*(t)$ the functional J on the variation Q is

$$J(s) = \int_a^b \frac{1}{2}\langle D_V V(t,s), D_V V(t,s)\rangle dt.$$

We obtain the Euler-Lagrange equation for J' by differentiating under the integral sign with respect to s.

$$J'(s) = \int_a^b \partial_2 \frac{1}{2}\langle D_V V(t,s), D_V V(t,s)\rangle dt$$

$J'(0)$ will be set to zero to find a curve that extremizes J. The metric connection on M is compatable with the metric. This requires that

$$Z\langle X, Y\rangle = \langle D_Z X, Y\rangle + \langle X, D_Z Y\rangle$$

for all vector fields X, Y and Z on M. Using this and the definition of the pullback over Q of D

$$(Q^* D)_{\partial_i} X = D_{Q_*\partial_i} X,$$

the integrand of J' becomes

$$\langle (Q^* D)_{\partial_2} D_V V, D_V V\rangle = \langle D_U D_V V, D_V V\rangle.$$

Using the definition of the curvature tensor

$$R(X,Y)Z = D_X D_Y Z - D_Y D_X Z - D_{[X,Y]} Z$$

and the commutation of U and V we may switch the order of the covariant derivatives giving

$$\langle D_V D_U V + R(U,V)V, D_V V\rangle = \langle D_V D_U V, D_V V\rangle + \langle R(U,V)V, D_V V\rangle.$$

The metric connection has torsion zero, thus $D_X Y = D_Y X$ giving

$$\langle D_V D_V U, D_V V\rangle + \langle R(U,V)V, D_V V\rangle.$$

With compatability in the form

$$\partial_1 \langle D_V U, D_V V \rangle = \langle D_V D_V U, D_V V \rangle + \langle D_V U, D_V D_V V \rangle$$

the first term of the integrand is

$$\partial_1 \langle D_V U, D_V V \rangle - \langle D_V U, D_V D_V V \rangle.$$

Using compatability with the metric again as

$$\partial_1 \langle U, D_V D_V V \rangle = \langle D_V U, D_V D_V V \rangle + \langle U, D_V D_V D_V V \rangle$$

the first term is now

$$\partial_1 \langle D_V U, D_V V \rangle - \partial_1 \langle U, D_V D_V V \rangle + \langle U, D_V D_V D_V V \rangle.$$

The partial derivatives along the base curve of the variation can be integrated directly to give

$$\langle D_V U, D_V V \rangle \big|_{(a,0)}^{(b,0)} - \langle U, D_V D_V V \rangle \big|_{(a,0)}^{(b,0)}.$$

Since the endpoints and their tangents are fixed in the variation, U and $D_V U$ evaluate to zero and both inner products vanish. The remaining integrand is

$$\langle D_V D_V D_V V, U \rangle + \langle R(U,V)V, D_V V \rangle.$$

Choosing two of the many symmetry properties of the curvature tensor,

$$\langle R(X,Y)Z, W \rangle = \langle R(Z,W)X, Y \rangle = -\langle R(Z,W)Y, X \rangle$$

the integral becomes

$$0 = J'(0) = \int_a^b \langle D_V D_V D_V V - R(V, D_V V)V, U \rangle dt.$$

Because this holds for all variations Q we must have

$$D_V D_V D_V V - R(V, D_V V)V = 0$$

and the theorem is proved. ∎

Appendix 2: The Jacobian for the N-Sphere

We define the total extrinsic velocity $v_T = v \cdot v + \beta x \cdot v$. We first need some partials of v_T and the α and β defined in section 6.2.

$$\begin{aligned}
\frac{\partial v_T}{\partial x} &= 2\beta(\beta x + v) \\
\frac{\partial v_T}{\partial v} &= 2(\beta x + v) \\
\frac{\partial \beta}{\partial x} &= \frac{2x \cdot v}{(1 - x \cdot x)^2} x + \frac{v}{1 - x \cdot x} \\
\frac{\partial \beta}{\partial v} &= \frac{x}{1 - x \cdot x} \\
\frac{\partial \alpha}{\partial x} &= \beta u + (x \cdot u)\frac{\partial \beta}{\partial x} \\
\frac{\partial \alpha}{\partial v} &= u + (x \cdot u)\frac{\partial \beta}{\partial v} \\
\frac{\partial \alpha}{\partial u} &= \beta x + v
\end{aligned}$$

With the symbol $\otimes$ denoting outer product, $x \otimes x = x^i x^j$, the 16 $n \times n$ submatrices of the Jacobian matrix are

$$\begin{aligned}
\frac{\partial f_x}{\partial x} &= 0 \\
\frac{\partial f_x}{\partial v} &= I \\
\frac{\partial f_x}{\partial u} &= 0 \\
\frac{\partial f_x}{\partial w} &= 0 \\
\frac{\partial f_v}{\partial x} &= -Iv_T - x \otimes \frac{\partial v_T}{\partial x} \\
\frac{\partial f_v}{\partial v} &= -x \otimes \frac{\partial v_T}{\partial v} \\
\frac{\partial f_v}{\partial u} &= I \\
\frac{\partial f_v}{\partial w} &= 0 \\
\frac{\partial f_u}{\partial x} &= -I\alpha - x \otimes \frac{\partial \alpha}{\partial x} \\
\frac{\partial f_u}{\partial v} &= -x \otimes \frac{\partial \alpha}{\partial v} \\
\frac{\partial f_u}{\partial u} &= -x \otimes \frac{\partial \alpha}{\partial u} \\
\frac{\partial f_u}{\partial w} &= I \\
\frac{\partial f_w}{\partial x} &= -I(\beta x \cdot u + v \cdot u) - \beta x \cdot u - (x \cdot u)(x \otimes \frac{\partial \beta}{\partial x}) + v \otimes \frac{\partial \alpha}{\partial x} - u \otimes \frac{\partial v_T}{\partial x} \\
\frac{\partial f_w}{\partial v} &= -I\alpha - (x \cdot w)x \otimes \frac{\partial \beta}{\partial v} - x \otimes w + v \otimes \frac{\partial \alpha}{\partial v} - w \otimes \frac{\partial v_T}{\partial v} \\
\frac{\partial f_w}{\partial u} &= -Iv_T - v \otimes \frac{\partial \alpha}{\partial u} \\
\frac{\partial f_w}{\partial w} &= -\beta x \otimes x - x \otimes v
\end{aligned}$$

Object to Object Clipping

Siggraph Tutorial

Thaddeus J. Beier

May, 28 1985

The problem

There are two general ways of describing models in three dimensional computer graphics, volume descriptions or surface descriptions. We chose to use a surface description modeling enviroment, where everything is described by polygons, with normal interpolation across the polygons to give smooth shading. This has a number of advantages, just one primitive type is used to generate any model, so rendering is easier. All modeling tools need deal with only one kind of primitive, and the mathematics of polygons is simple. One problem with this description is that you sometimes need a lot of polygons to create simple shapes, i.e. spheres. Also, some things that might be easy to compute with other primitives are hard with just polygons, for example, boolean operations on three-dimensional objects.

The full three dimensional case is illustrated by the two and a half dimensional case of a solid object being clipped to the inside of a infinite prism of arbitrary cross section. As an example, fig. 1 shows a house to be clipped by a curved prism. From straight in front of the house we see that the prism is a closed curve. I'll call the curved boundary the clipper, and the house the clippee. The boundary runs counterclockwise, so that the inside is to the left of the boundary.

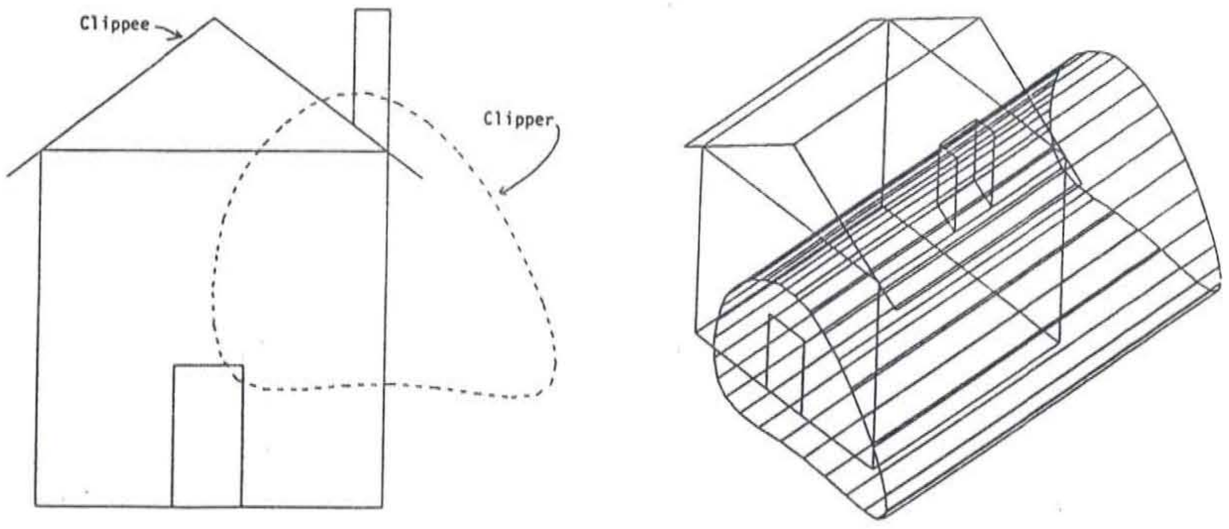


Figure 1.

The wrong, or at least slow, way.

This problem can be solved by first clipping each of the polygons in the house to each of the polygons of the prism. All of the polygons that are in fact cut by the clipper have their sidedness determined, it is easy to see which part of the polygon is inside and which is outside. (Fig 2.) For the polygons either completely inside or completely outside of the clipper, it is not so easy. Each of these must have an infinite ray fired off in any direction from part of the polygon. If the ray intersects the clipper an even number of times, then the polygon is outside the clipper, otherwise it is inside. (Fig 3.)

This approach is simple, but slow. Both the initial clipping and the determination of sidedness can take n * m time, where n is the number of polygons in the clipper and m is the number of polygons in the clippee.

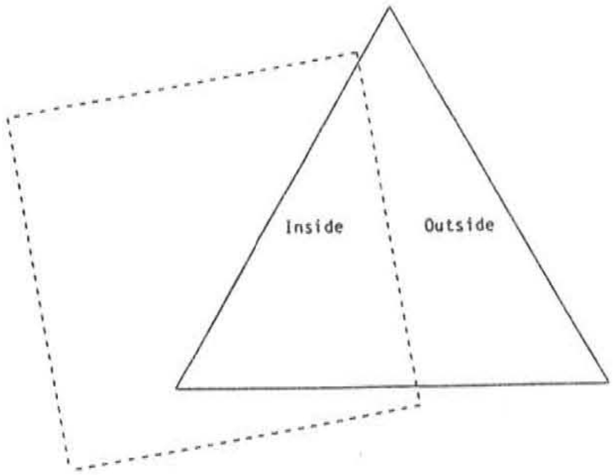


Figure 2.

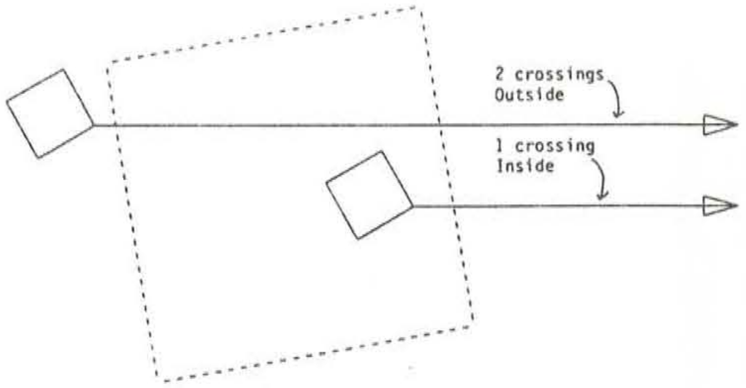
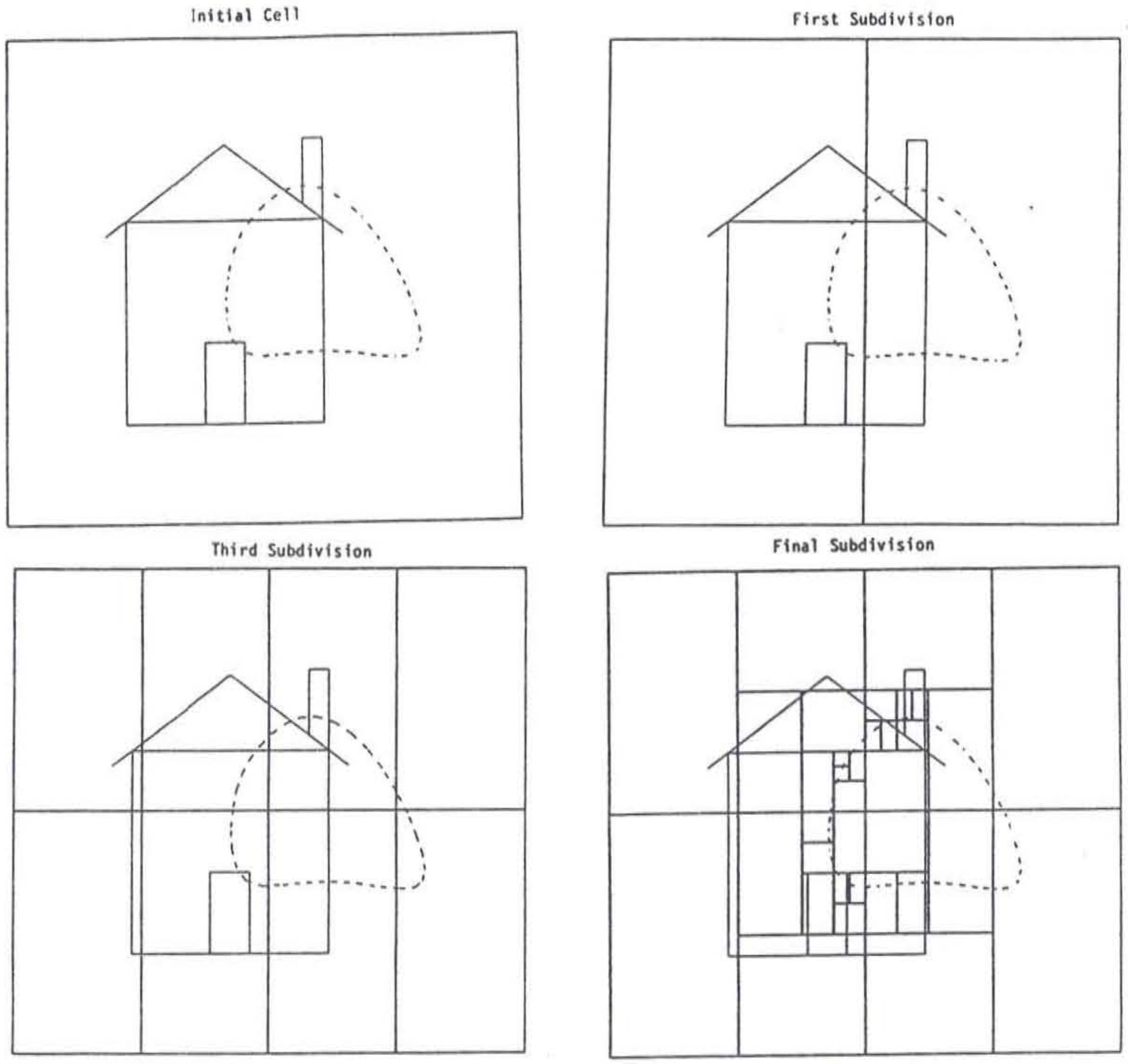


Figure 3.

The right way.

Another way to solve this problem is to try to only test polygons that are near each other, to see if they do in fact intersect. One way to do that is to partition space into cells recursively, so that finally each cell is so simple that the clipping can be done quickly. First, a cell is created that contains all of the clipper and the clippee. Then, it is divided in half along its longest dimension. Each polygon from both the clipper and the clippee that cross this boundary are clipped, each part being put into their respective cell. This process continues until the cell is simple enough that it is easy to clip the polygons in the cell.



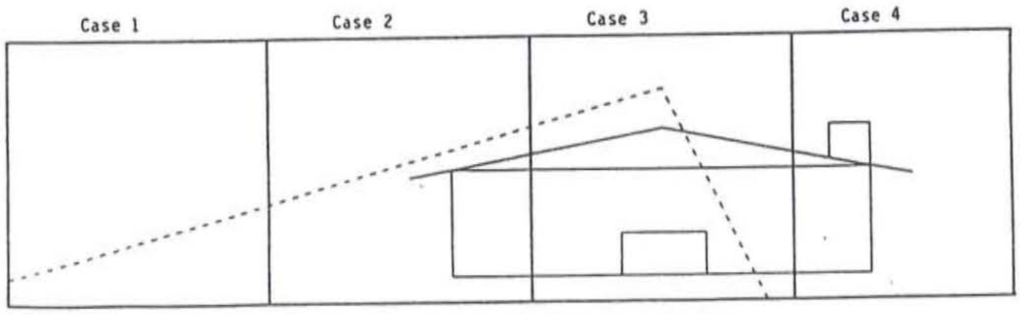
In my program, simple enough means that either:

case 1. the cell contains no clippee polygons

case 2. the cell contains only one clipper polygon

case 3. the cell contains two clipper polygons,
and they share a vertex in the cell

case 4. the cell contains no clipper polygons.



The first two cases are simple. In case one, the cell is ignored, and processing continues with the next cell. In case two, the clippee polygons are clipped to the one clipper edge, and the polygons inside the edge are written out.

In case three, the polygons are clipped first to the first edge. Every polygon that results from the clipping is tagged as to whether it was inside or outside of the first edge. Then each of these polygons is clipped to the second edge. The angle between the edges is then measured. If it is less than 180 degrees (measured couterclockwise), then the polygons that are inside of EITHER of the two edges are inside the boundary, but if the angle is more than 180 degrees, the polygons must be inside BOTH of the edges to be inside the boundary.

Finally, in case four, there are two possibilities, that all of the polygons are inside the boundary or that they are all outside. More information is needed, so we add to each cell information about the sidedness of each corner of the cell. The corners of the root cell are all outside of the region. Each time that a new cell is created, the sidedness of each of the new corners is determined. This is done by counting the number of boundary crossings along one of the edges of a cell, from an old vertex to the new one. If the number of crossings is odd, then the sidedness is the same as it was for the old corner, otherwise it is the opposite.

This approach works very well if the polygons are relatively evenly distributed and relatively small with respect to the size of the whole model. For example, on one test, with 4000 polygons in the clippee and 100 edges in the boundary, this program took 100 seconds, which is 60 times faster than the first program described.

The three dimensional case.

This technique can easily be extended to the case where the clipper is a full three dimensional closed solid. The cells would be rectangular parallelpipeds instead of rectangles, and the boundary would be made up of polygons instead of just edges. The final cells are somewhat more complicated:

case 1: no clippee polygons

case 2: one clipper polygon, a face cell

case 3: two clipper polygons that meet at an edge, an edge cell

case 4: three or more clipper polygons that meet at a point,
a vertex cell

case 5: no clipper polygons.

All of the clipping is performed analogously to the two dimensional case.

Extensions and refinements.

One problem with this technique is that polygons often divided up when they do not need to be. Therefore the final database may consist of many more polygons than is necessary, increasing its size and slowing down any further operations to be performed. One way to solve this would be to keep a tree of the ancestry of each polygon, and when the clipping is finally done, trace back up the tree as far as possible before writing out a polygon.

This approach can also be used to generate shadows. Every polygon creates a shadow volume that is used to clip every other polygon. This divides the database into two, one of shadowed and the other of unshadowed polygons.

Bibliography

1. Carlson, Wayne E., "Techniques for the Generation of Three Dimensional Data for Use in Complex Image Synthesis." Doctoral Dissertation, Ohio State University, (Sep. 1982)

2. Baumgart, B.G. "GEOMED - A Geometric Editor," AIM-232, Stanford Artificial Intelligence Laboratory, Stanford University Computer Science Department. (May 1984)

3. Carlbom, Chakravarty, and Vanderschel, "A Hierarchical Data Structure for Representing the Spatial Decomposition of 3-D Objects", IEEE Computer Graphics and Applications, April 1985.

EFFICIENCY CONSIDERATIONS IN IMAGE SYNTHESIS

Edwin P. Berlin, Jr.
Cubicomp Corporation
3165 Adeline Street
Berkeley, CA 94703

Introduction

Our work is primarily concerned with the implementation of a production quality computer graphics system for stills and animation which runs on a microcomputer. All of our images are produced on an IBM PC or PC/AT without any additional processing hardware beyond the standard floating point coprocessor. this environment provides a unique challenge to the software engineer who must live within the constraints of severe speed and memory limitations.

This paper will discuss a few general considerations of efficient program design and will then focus on a few new algorithms which are the results of some of our work.

Language

The most obvious implementation detail which affects both running time and memory utilization is the implementation language. Assembly code can yield a factor of two to ten over the corresponding C code while simultaneously requiring less code space. This code, however, is much more expensive to produce, debug and maintain and it is difficult to transport to other hardware. For these reasons, we use C almost exclusively, with only a few low level I/O routines coded in assembly.

Arithmetic

While it is easier for the programmer to make all variables floating point, significant savings may be achieved by using fixed point or even integer arithmetic where possible. Fixed point code must pay careful attention to the position of the radix point at all times, and a detailed precision analysis is a must. For example, if you are performing an image filtering operation requiring the summation of up to 512 weighted pixels of 8 bits each, potentially 17 bits are required and you cannot always achieve accurate results with 16 bit computations. By careful arrangement of the computation or adding another variable, 16 bits can be made to suffice. Don't forget to declare positive fractions as unsigned.

We have found that 32 bit fixed point is about twice as fast as floating point, and 16 bit yields about a factor of six on the IBM PC.

Sorting

Sorting is the most ubiquitous operation in computer graphics. One hidden surface algorithm may perform more than half-a-dozen sorts on various entities. Here are some rules of thumb:

- If a list typically has fewer than 7 items (as in polygon scan conversion) use a bubble sort.

- If you are keeping items on a linked list, sorting on insertion requires linear time per insertion and therefore is O(n^2). Better to use another sort before insertion or on removal.

- Quicksort is fast, but a radix sort is O(n). A straight multi-pass radix sort requires memory for intermediate buckets and has an appreciable overhead for medium sized lists. The radix exchange sort is a better choice for 16 bit quantities. [Knuth]

Do Computations in the Proper Order

Sometimes rearranging the order of steps in a computation can make significant savings in time or space. Two 3x3 matrices may be multiplied using only 23 multiplies rather than 27 by rearranging the steps [Laderman]. Our texture mapping method (described later) saves an entire frame buffer's worth of storage.

Model Time vs. Render Time

Especially in animation applications, it is usually an acceptable tradeoff to perform view-independent computations when a model is built in order to avoid computing them each frame. The cost is the extra memory required to store this information. We store point normal vectors and polygon plane equations with a model rather than computing them each frame. We do not store edges (pointing to two vertices and one or two polygons) explicitly, since we can compute them relatively quickly and the memory saved is significant. An extreme example of this tradeoff is the BSP tree algorithm [Fuchs] which effectively solves the hidden surface problem for all views and retains this solution with the model with only a modest increase in memory.

Algorithm Design

We have found that the best way to develop efficient programs is to design algorithms at a fundamental level with specific criteria in mind. While O(n) or O(n log n) may seem better than O(n^3/2) or O(n^2), if the cross-over point is 100,000 polygons and your system can only store 10,000 polygons, those algorithms would not be the best choice.

Let's look at some new graphics algorithms:

Point Containment

Is a point P on the x,y plane contained in polygon C = $(P_0, P_1, \ldots, P_{n-1})$? This question comes up when determining overlap of polygons, and in boolean operations on polyhedra. There are two commonly described techniques for performing this test.

The first is to sum up the signed angle formed by point P and each edge of polygon C (Figure 1). If the sum is zero the point is not contained. Otherwise the sum should be $\pm$ 360 degrees assuming C does not cross itself. This method unfortunately requires the computation of an inverse trig function per vertex of C, and so is quite time consuming.

The second, and most commonly used, method is to extend a semi-infinite ray from P and count the number of intersections with the closed curve C. If the ray is aligned with the x or y axes, this computation is particularly simple. There are, however, some difficulties with this method. Points a and b in Figure 2 could count 1,2 or 3 intersections arbitrarily unless these cases are handled specially. While a ray can always be found which is not a problem case, the overhead is doing so is significant.

Our method is illustrated in Figure 3. One point on the polygon (say P_0) is chosen and P is tested for containment in the triangles formed by P_0 and every edge of C not sharing P_0. If this number is even, P is not contained in C, otherwise it is. Containment in a triangle is determined by putting P into the equation of each line bounding the triangle to see if it is on the same side of each. Note that there are no problematical cases: if P falls on an interior line it will fall into the triangle on one side or the other. If it falls on an actual edge of C, you choose what answer is meaningful to you. Note that for n vertices only n-2 tests are performed. Incidentally, the area of C may be computed at the same time, if desired, by summing the signed area of each triangle.

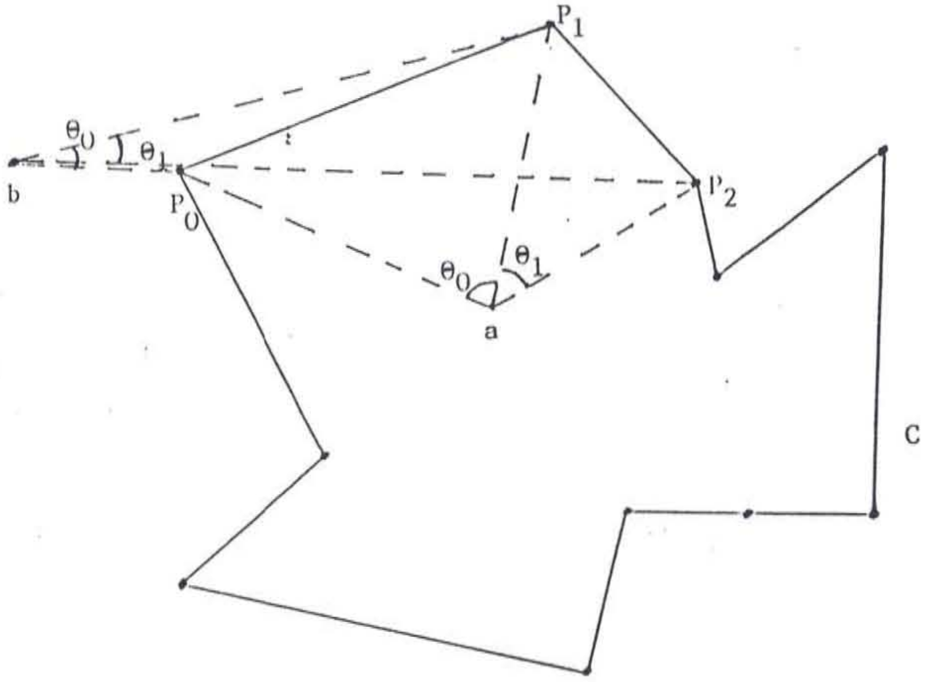


FIGURE 1.

Point containment by angle summation.
Sum at a is 360 degrees;
Sum at b is 0.

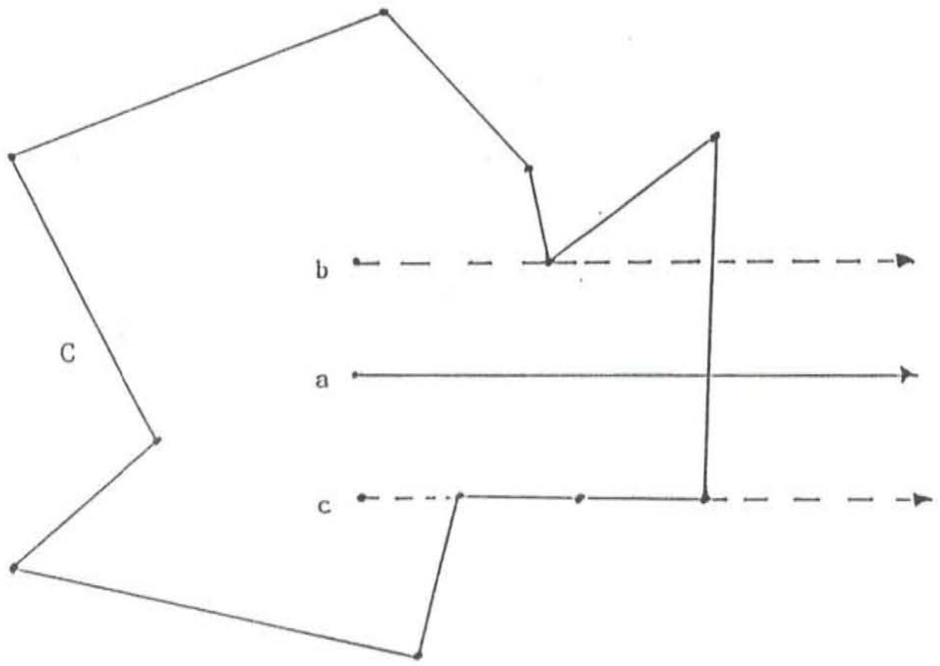


FIGURE 2.
Point containment by ray intersection.
Points b & c are problem cases.

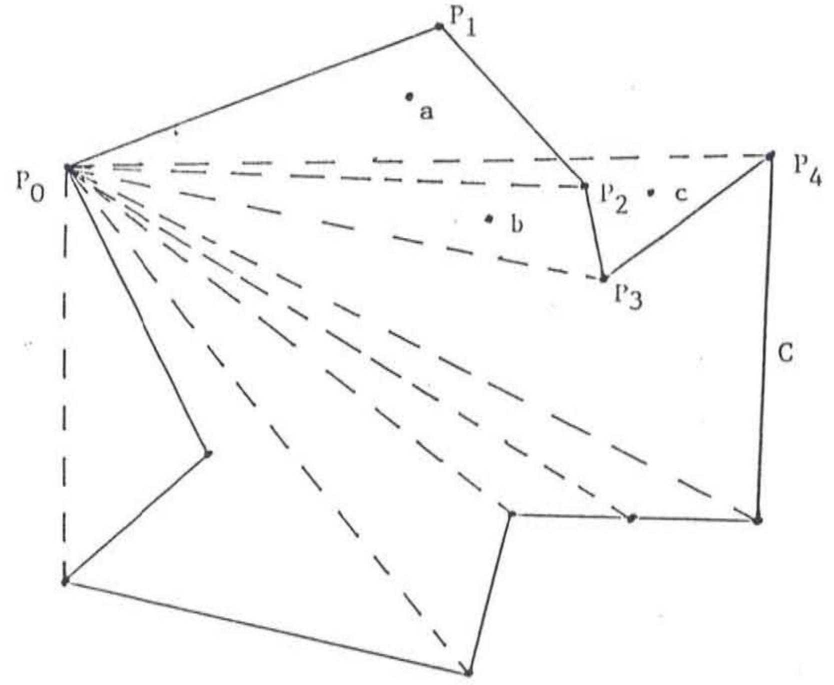


FIGURE 3.
Point containment by triangle test.
Point a is contained by one triangle;
b is contained by three;
c by two.

An extension to three dimensions is to perform the triangle test of a point in three space by testing that point against the three planes perpendicular to C through the appropriate vertices.

Texture Mapping

We implement texture mapping in a very general way as follows: Every vertex of textured polygons has a corresponding (x,y) coordinate on the source image. These coordinates may be interpolated along edges and across scanlines between edges (adjusted with a possible correction factor for accurate perspective) to indicate the region of interest on the source image. If the process of texture mapping were simply one of finding the nearest pixel on the source image and copying it to the output image implementation would be very simple. We do, in fact, provide this option for a 'quick look' at an aliased image. Proper antialiasing, however, requires a two-dimensional filtering operation. While a simple box filter can provide some measure of antialiasing, higher order filters are distinctly preferable. Rather than skimp on the filter kernal, we chose to gain efficiencies through a two-pass filter process.

The excellent treatment of two-pass filtering due to [Catmull-Smith] describes a simple process where the source image is filtered, scaled, and shifted in one direction (x or y) and the resulting image is processed similarly in the orthogonal direction. This process has one undesirable drawback - it requires one entire frame of intermediate storage or it destroys the original source image. Since wrapping an image around a cylinder, for example, using a polygonal model involves mapping many adjacent regions from the source image, it is not acceptable to destroy that image during the mapping process.

This is a case where rearranging the order of computation saves a great deal of storage. The standard two-pass method would filter along column a in Figure 4. Only after every column has been filtered can output scanlines be produced by filtering the intermediate image horizontally (Figure 5).

Our idea is to produce this intermediate image in <u>horizontal</u> scanline order rather than <u>vertical</u> scanline order by computing vertical filter components along direction b in Figure 4. Since the filter is a non-uniform finite impulse response filter, the filter computation is approximately the same for this order as the other. We can compute the current scanline of the intermediate image in a one scanline buffer and then filter horizontally as we can that buffer and write to our frame buffer. The one scanline buffer may then be re-used.

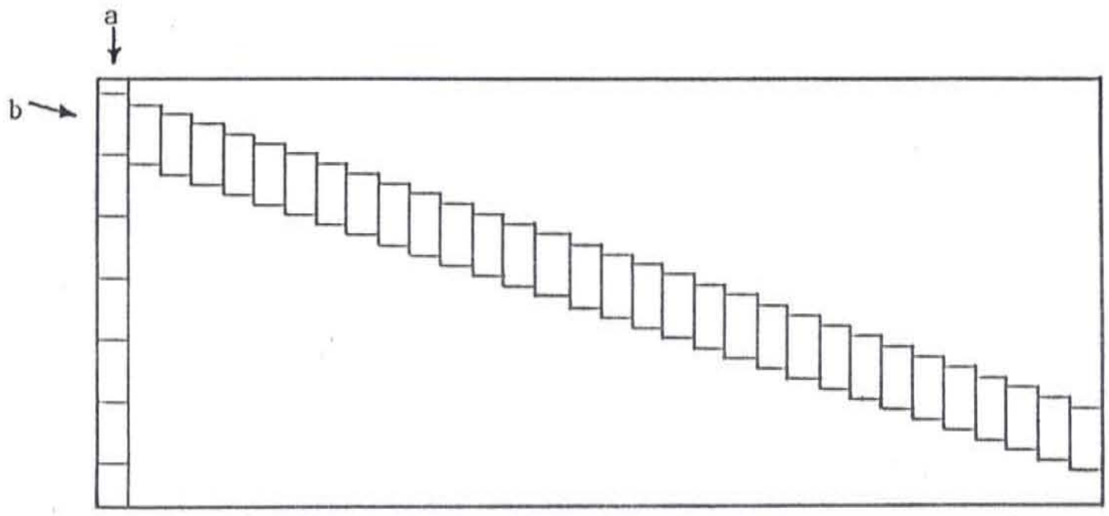


FIGURE 4.
First pass filter direction.

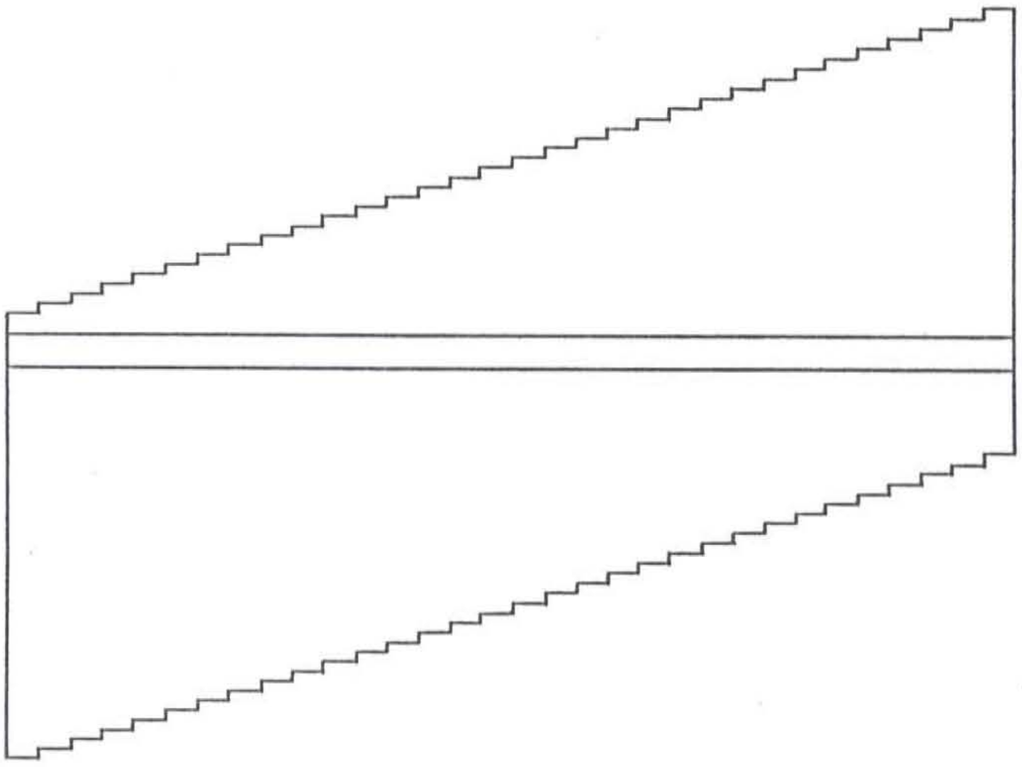


FIGURE 5.
Second pass filter direction.

Thus the procedure is as follows:

1. Scan convert textured polygons.

2. For each scanline segment, find the corresponding (x,y) coordinates on the source image at each end point of the segment. Also compute the size of the filter regions based on (dx,dy).

3. These two coordinate pairs define a line on the source image. If that line is more horizontal than vertical filter in Y first; otherwise filter in X. This will automatically perform a 90 degree rotation where necessary to avoid degenerate cases.

4. Scan in a direction perpendicular to the filter direction, filtering into a one line scanline buffer. Interpolate x or y and dx or dy to facilitate this. This buffer will contain one scanline of the intermediate image.

5. For each pixel in the final image, interpolate the index (call it i) into the scanline buffer and di to filter from the buffer into the output pixel.

This method allows us to provide good quality texture mapping in acceptable time on a micro-based system. Note also, since it will map images from any polygon to any other (provided they have the same number of vertices), it facilitates texture map specification, anamorphisms, and intentional distortions. Obvious extensions are bumpy mapping and reflectance mapping. There are more implementation details, such as how to avoid computing portions of the intermediate image which have no effect on the final image because of visibility, which we will not cover here.

Improved Shading Models Without Ray Tracing

How far can we go toward extreme realism without resorting to ray tracing [Whitted]? Clearly we can produce complex motions of polyhedra combining flat or smooth shading with various surface models [Cook], antialiased edges, translucency, texture mapping, and multiple colored light sources. Let's look at something more difficult, like reflection.

The difficulty with mirror surfaces in a scene is that the hidden surface problem must be solved from many viewpoints in each frame. Recall that the BSP tree structure effectively solves the hidden surface problem from all directions simultaneously. The basic BSP tree produces a polygon painting order (say back-to-front) as follows:

Each polygon C points to a subtree of polygons on the clockwise size of C, and a subtree of polygons on the counterclockwise side of C. Polygons are subdivided, if necessary, when the tree is created to insure no polygons cross the plane of C. To produce a complete visibility sort, test the eye point against the plane of the root polygon of the BSP tree. If the eye is on the clockwise side of the root, recursively paint the counterclockwise subtree, then the root polygon (if it is visible), then recursively paint the clockwise subtree.

Mirror polygons may be handled easily with this structure. Assume, for the moment, that the root polygon is designated reflecting. When the time comes to paint that polygon, paint it with a default color (the color of a mirror not reflecting any other surface). Then reflect the eyepoint through the plane of the root polygon and use this new eyepoint to traverse the frontmost subtree of the root. This produces a list of polygons which may be seen reflected in the root polygon. Reflect each of these through the plane of the root polygon and clip them to that polygon; many polygons will be discarded. Paint those that remain in the order generated. If the root polygon is to be other than a perfect reflector - say, tinted - paint it again as translucent or modify the shading characteristics of the reflected polygons before they are painted.

If the mirror polygon is not the root, polygons clipped to the mirror polygon must also be subdivided by the plane of that polygon, discarding any portion behind the mirror. The moral is, choose mirror polygons first when creating the tree.

For multiple reflections, the entire tree can be retained when polygons are reflected and the process applied recursively. Alternatively, the eye point may be reflected again and used on the original tree, with multiple reflections applied to each polygon for each level of reflection. This is a memory versus speed tradeoff.

The polygon/polygon clipping may be done several ways. A standard polygon 'cookie cutter' may be used [Weiler-Atherton] or, after passing bounding box tests, the polygon may be scan converted and only those segments which overlap segments of the mirror polygon are painted.

Needless to say, the resulting image is as easy to antialias as a single polygon.

What about refraction? This is a process very similar to reflection. Instead of reflecting the eyepoint in the refracting surface, the eye is moved to a new position using Snell's Law based on the angle between the eye and the surface normal. This is an approximation which is valid when the eye is not too close to the refracting surface. The far subtree is then processed as above rather than the near subtree. You need to recur for at least two refracting surfaces in order to properly represent a solid refracting object.

To implement shadows, use the tree to generate a list of polygons, this time sorted for visibility from the viewpoint of the lightsource. Use a 'cookie cutter' to remove visible portions of polygons leaving 3D polygons which when transformed to a particular view represent that portion of each polygon in shadow from a particular light source. If an entire polygon is in shadow, merely mark it so. These shadow polygons may be painted along with the original polygons or otherwise used to affect the lighting model within their interior. If the light sources do not move relative to the object, shadow polygons may be retained as part of the model. Alternatively, another polygon reconstruction algorithm may be used [Sechrest-Greenberg].

Of course, if you must ray trace, you can traverse the tree from the point of view of a ray, front-to-back, stopping at the first polygon which the ray pierces. After bounding box tests, use the 3D point containment test described earlier.

Higher Order Surfaces

There are many choices for primitives in a 3D model other than polygons. Here I will describe a fast algorithm for mixed polygons and spheres, again based on the BSP tree. Spheres will be augmented by the capability of bounding them with a number of planes.

First, using distance tests, determine all pairs of spheres which intersect. If a sphere is contained in a larger one, throw it away. If spheres partially intersect, compute the plane of intersection and add it to the list of bounding planes for both spheres. Mark these planes invisible. Now, using these planes, as well as the planes of any polygons in the scene, build a BSP tree. If a sphere needs to be subdivided, form two new spheres, each bounded by the dividing plane (one on one side, one on the other).

To paint, simply traverse the tree, painting each polygon or bounded sphere as it appears. If any subtree contains only spheres (with no cutting planes) the spheres do not intersect and may be sorted by their distance to the eye. The process of painting a quadric surface bounded by planes is well treated by [Gardner]. Some simplifications may be made if all quadrics are known to be spheres.

Reflection and refraction may still be handled as described earlier, with only reflecting spheres ray traced.

Conclusions

I have tried to give a brief overview of what would typically be the substance of several papers. This format does not allow the inclusion of implementation details or runtime statistics. I hope the main point is apparent, however, that there is still plenty of opportunity for the design of efficient algorithms which will ultimately reduce the cost of computer graphics while permitting increased quality.

References

Catmull, Ed and Alvy Ray Smith. "3-D Transformations of Images in Scanline Order." ACM SIGGRAPH '80 Conference Proceedings Vol. 14, No. 3.

Cook, Robert L. and Kenneth E. Torrance. "A Reflectance Model for Computer Graphics." ACM Transactions on Graphics, Vol. 1, No. 1, January '82.

Fuchs, Henry Et Al. "Near Real-Time Shaded Display of Rigid Objects." ACM SIGGRAPH '83 Conference Proceedings Vol. 17, No. 3.

Gardner, Geoffrey Y. "Simulation of Natural Scenes Using Textured Quadric Surfaces." ACM SIGGRAPH '84 Conference Proceedings Vol. 18, No. 3.

Hedgley, David R., Jr. "A General Solution to the Hidden-Line Problem." NASA Reference Publication 1085. 1982.

Knuth, Donald E. _The Art of Computer Programming Volume 3: Sorting and Searching._ Addison-Wesley 1973.

Laderman, Julian David. "An Algorithm for 3 X 3 Matrix by 3 X 3 Matrix Multiply in 23 Multiplies." Bulletin of American Mathematical Society, Jan. 1976.

Sechrest, Stuart and Donald P. Greenberg. "A Visible Polygon Reconstruction Algorithm." ACM Transactions on Graphics, Vol. 1, No. 1, Jan. 1982.

Weiler, K. and P. Atherton. "Hidden Surface Removal Using Polygon Area Sorting." ACM SIGGRAPH '77 Conference Proceedings Vol. 11, No. 2.

Whitted, T. "An Improved Illumination Model for Shaded Display." _Communications_ ACM Vol. 23, No. 6, June '80.

3-D TRANSFORMATIONS OF IMAGES IN SCANLINE ORDER

Ed Catmull and Alvy Ray Smith
Lucasfilm Ltd.
P.O.Box 7
San Anselmo, CA 94960

ABSTRACT - Currently texture mapping onto projections of 3-D surfaces is time consuming and subject to considerable aliasing errors. Usually the procedure is to perform some inverse mapping from the area of the pixel onto the surface texture. It is difficult to do this correctly. There is an alternate approach where the texture surface is transformed as a 2-D image until it conforms to a projection of a polygon placed arbitrarily in 3-space. The great advantage of this approach is that the 2-D transformation can be decomposed into two simple transforms, one in horizontal and the other in vertical scanline order. horizontal scanline order. Sophisticated light calculation is also time consuming and difficult to calculate correctly on projected polygons. Instead of calculating the lighting based on the position of the polygon, lights, and eye, the lights and eye can be transformed to a corresponding position for a unit square which we can consider to be a canonical polygon. After this canonical polygon is correctly textured and shaded it can be easily conformed to the projection of the 3-D surface.

KEY WORDS AND PHRASES: texture mapping, scanline algorithm, spatial transforms, 2-pass algorithm, stream processor, warping, bottleneck, foldover

CR CATEGORY: 8.2

INTRODUCTION

Texture mapping is an immensely powerful idea now being exploited in computer graphics. It was first developed by one of the authors [2] and extended by Blinn [1] who produced some startling pictures. In this paper we present a new approach to texture mapping that is potentially much faster than previous techniques and has fewer problems.

The two chief difficulties have been aliasing and the time it takes to do the transformation of a picture onto the projection of some patch. Usually the procedure is to perform some inverse mapping of a pixel onto a surface texture (Fig. 1). It is difficult to do this correctly because the inverse mapping does not happen in scanline order and also because we must integrate under the whole inverse image in order to prevent sampling errors.

We present in this paper an approach that does the mapping in scanline order both in scanning the texture map and in producing the projected image. Processing pixels in scanline order allows us to specify hardware that may work at video rates. We emphasize, however, that the approach is valuable for software as well as hardware implementations.

One of the key concepts we use is that of a "stream processor". Pixels enter the stream processor at video rate, are modified or merged in some way with another incoming stream of pixels and then sent to the output (Fig. 2). This concept has been implemented by several manufacturers for image processing. A generalization of the concept would be to allow the framebuffers to feed the streams in either horizontal or vertical scanline order.

We will show here that the class of transformations that can be applied to streams is much broader than previously believed. For example, an image in a frame buffer may be rotated by some arbitrary angle even though the data is sent through the processor in scanline order only. While this concept has been known for some time [3,4], we show here that the technique can be generalized to perspective projections. Further generalizations include quadric and bivariate curved surfaces. The ability to transform a whole raster image very quickly lets us consider doing shading calculations on a unit square where the calculations may be more amenable to stream processing and then transforming the results.

When we say "scanline order", we use a slightly broader meaning than normal. Usually this means that the order of the pixels is from left to right across a scanline and that the scanlines come in top to bottom order. We broaden the definition to include vertical scanline order. In addition, the scanlines may also occur in bottom to top or right to left order. This gives us trivially a 90 degree rotate and flopping a picture over in one pass through the picture.

EXAMPLE: SIMPLE ROTATION

For illustration, we present the simple case of rotation. We would like to rotate an entire image in the framebuffer. The rotation matrix is:

$$[x' \ y'] = \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

where x and y refer to coordinates in the original picture and x', y' are the new coordinates (c=cos, s=sin).

We want to transform every pixel in the original picture. If we hold y constant and move along x then we are transforming the data in scanline order but the results are not coming out in scanline order. Not only is this inconvenient, it is also difficult to prevent aliasing errors.

There is an alternate method for transforming all of the pixels, and that is to evaluate only the x' in a first pass and then the y' in a second pass.

So again hold y constant, but just evaluate x':

$$[x' \ y] = [cx-sy \ \ y].$$

We now have a picture that has been skewed and scaled in the x direction, but every pixel has its original y value. See Fig.4, where Fig.4a is the original picture and Fig.4b is after the horizontal scanline computation.

Next we can transform the intermediate picture by holding x' constant and calculating y. Unfortunately, the equation $y' = sx+cy$ can't be used because the x value for that vertical scanline is not the right one for the equation. So let us invert x' to get the correct x. We need x in terms of x'.

Recall $x' = cx-sy$, so

$$x = x'/c + sy/c.$$

Plug this into $y' = sx + cy$ to get:

$$y' = (sx' + y)/c.$$

©1980 ACM 0-89791-021-4/80/0700-0279 $00.75

Now we transform the y value of the pixels in the intermediate picture in vertical scanline order to get the final picture, Fig.4c.

The first pass went in horizontal scanline order on input and output. The second was vertical in both. So in two passes the entire picture was rotated.

Before we generalize, two points should be noted.

(1) A 90 degree rotate would cause the intermediate picture to collapse to a line. It would be better to read the scanlines horizontally from the source buffer and write them vertically to effect that rotate. It follows that an 80 degree rotate should be performed by first rotating 90 degrees as noted then by -10 degrees using the 2-pass algorithm.

(2) The rate at which pixels are read from the input buffer is generally different than the rate at which they are sent to the output buffer. If we're not careful we could get sampling problems. However, since all of the pixels pass through the processor, it is not difficult to filter and integrate the incoming values to get an output value.

Next we generalize to:

$$[x' \; y'] = [X(x,y) \;\; Y(x,y)].$$

This generalization will include perspective. Whatever the transformation is, we shall show that we can do the x transforms first, followed by the y transforms, but in order to do the y transforms we must be able to find the inverse of x'. This may be very difficult to do and x' may even have multiple values. So we present first a more formal way of talking about the method before addressing some of the difficulties.

THE 2-PASS TECHNIQUE

We are interested in mapping the 2-D region bounded by a unit square into a 3-D surface which is projected back into 2-D for final viewing. Since the unit square (by which we mean the enclosed points also) may be represented by point samples in a digital framebuffer, and since a framebuffer is typically arranged in rows and columns, we are interested in row-ordered or column-ordered implementations of these mappings. The technique we now present is a means of decomposing a 2-D mapping into a succession of two 1-D mappings, or scanline-ordered mappings, where a scanline may be either horizontal (a row) or vertical (a column). The technique is quite general as we shall show subsequently.

This figure illustrates the 2-pass technique:

$$\begin{tikzcd}
(u,v) \arrow[r, "f(u)"] \arrow[dr, "{l(u,v),\ r(u,v)}"'] & (x',v) \arrow[d, "g(v)"] \\
 & (x',y')
\end{tikzcd}$$

We want to map the set of points $\{(u,v): 0<u<1, 0<v<1\}$ in the unit square into the set $\{(x',y')\}$ where the desired mapping is given as an arbitrary pair of functions

$$x'=l(u,v)$$
$$y'=r(u,v).$$

We wish to replace this pair of functions with the pair

$$x'=f(u)$$
$$y'=g(v)$$

where it is understood that f(u) is applied to all points in the unit square before g(v) is applied to any of them. We call the application of f the h-pass (for horizontal) and the application of g the v-pass (for vertical).

In general, there will be a different f(u) for each value of v, so f might be thought of as a function of (u,v). We prefer however to think of v as a parameter which selects a particular f(u) to be applied to all u on scanline v. To emphasize when v is being held constant like this, we will use the notation $\mathbb{V}$. Thus $\mathbb{V}$ is an index into a table of horizontal mappings. Similarly, there will in general be a different g(v) for each vertical scanline x' (where the prime indicates that the h-pass has already occurred). We will use the notation $\mathbb{X}'$ to indicate a given vertical scanline just prior to the v-pass.

In this section, we will always have the v-pass follow the h-pass. This is just a convenience. The decomposition into the other order proceeds similarly, and we will have occasion to choose one order over the other in a later section.

An algorithm for the decomposition of l,r into f,g is the following:

(1) $f(u)=l(u,\mathbb{V})$ is the function f for scanline $\mathbb{V}$.

(2) Solve the equation $l(u,v)-\mathbb{X}'=0$ for u to obtain $u=h(v)$ for scanline $\mathbb{X}'$.

(3) $g(v)=r(h(v),v)$ is the function g for scanline $\mathbb{X}'$.

We simply take f(u) as defined in (1) and show that g(v) in (3) is consistent with it. The h-pass takes the set of points $\{(u,v)\}$ into the set $\{(x',v)\}$. We desire a function which may be applied at this time to scanline $\mathbb{X}'$. But being given $\mathbb{X}'$ is equivalent to being given the equation

$$\mathbb{X}'=l(u,v).$$

If this equation can be rearranged to have the form $u=h(v)$, then $r(h(v),v)$ is a function of v only and is the desired g.

Thus solving the equation $l(u,v)-\mathbb{X}'=0$ for u is the key to the technique. We shall show some cases where this is simple, but in general it is not. An iterative solution such as provided by Newton-Raphson iteration could be used but is expensive. We shall treat these problems in the following sections.

It should be noted that we have placed no restrictions on functions l,r. So the 2-pass technique can be applied to a large class of picture transformations and distortions, only a few examples of which will be presented here. In particular, we henceforth restrict our attention to ratios of polynomials.

We shall illustrate the 2-pass technique by applying it, in detail, to the case of a rectangle undergoing affine transformations followed by a perspective transformation and projection into 2-space. Then, in less detail, we will treat bilinear and biquadratic patches under the same type of transformation. This should serve to indicate how the method can be extended to higher degree surfaces.

THE SIMPLE RECTANGLE

Consider the (trivial) parametric representation of a rectangle given by $x(u,v)=u$, $y(u,v)=v$, $z(u,v)=0$, $w(u,v)=1$. The class of transformations we apply are exactly those which can be represented by a 4x4 matrix multiplying a 3-space vector represented in homogeneous coordinates as indicated below:

$$[x \; y \; z \; w] \begin{bmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{bmatrix} = [x'' \; y'' \; z'' \; w''].$$

Then projection into 2-space is accomplished by dividing through the homogeneous coordinate w":

$$[x' \; y' \; z'] = [x''/w'' \;\; y''/w'' \;\; z''/w''].$$

Replacing x, y, z, and w with their parametric forms and expanding the equations above gives

$$x' = (au+bv+d)/(mu+nv+p) = l(u,v)$$

$$y' = (eu+fv+h)/(mu+nv+p) = r(u,v).$$

We are interested in the 2-D projection only so we shall ignore z' from here on.

The functions l,r in this case represent an ordinary linear transformation of the unit square, followed by a perspective projection. Notice that they are both rational linear polynomials - i.e., a linear polynomial divided by a linear polynomial.

Applying the 2-pass algorithm to the functions l,r gives:

(1) The h-pass function for scanline $\bar{v}$ is

$$f(u) = (Au+B)/(Cu+D)$$

where $A=a$, $B=b\bar{v}+d$, $C=m$, $D=n\bar{v}+p$.

(2) $u=h(v)$ is obtained by solving

$$\bar{x}' = (au+bv+d)/(mu+nv+p)$$

for u:

$$u = Ev+F$$

where $E=(b-n\bar{x}')/(m\bar{x}'-a)$ and $F=(d-p\bar{x}')/(m\bar{x}'-a)$.

(3) Thus

$$g(v) = (e(Ev+F)+fv+h)/(m(Ev+F)+nv+p) = (Gv+H)/(Iv+J)$$

is the v-pass function for scanline $\bar{x}'$, where $G=f+eE$, $H=h+eF$, $I=n+mE$, $J=p+mF$.

Fig.5 shows the results of applying this f,g pair. Fig.5a is the original rectangular texture. Fig.5b is its appearance after the h-pass, and Fig.5c is the result of the v-pass.

Following are several points about this computation:

(1) The sampled image (Fig.5a) was reconstructed with a first-order filter (the so-called Bartlett window) then resampled with a zeroth-order filter (the Fourier window). This is only minimal use of sampling theory. A piece of hardware or software for production quality work would certainly employ more sophisticated filtering. Our figures look surprisingly nice despite use of the low-order filters mentioned above. (The edges are not antialiased, however.)

(2) Clipping is natural. The f function generates the final value of x'. If this value should fall outside the limits of the output buffer then it does so with no loss. The g function, which operates only on the scanlines output by f, will never need values clipped in the h-pass.

(3) The 2-pass technique does not avoid the ordinary problems of perspective projections. For example, the transformation can blow up if the denominator of either f or g goes to zero. This corresponds to the usual problem of wraparound through infinity and requires the normal solution of clipping before transformation.

(4) There is a problem introduced by the 2-pass technique not encountered before. This is what we call the "bottleneck problem". We shall discuss this in greater detail and offer a solution to it in the next section, then return to the examples.

BOTTLENECK

With the perspective transformation we have a problem analogous to that of the 90 degree rotate, that is, it is possible to have an intermediate picture collapse. In the case of rotation the solution was simple: rotate the texture 90 degrees and change the tranformation by that amount. The solution for the perspective case is the same, however it is more difficult to tell from the transformation matrix when a problem will occur.

We base our criteria on the area of the image in the intermediate picture. There are four possible ways to generate an intermediate picture:

1. transform x first
2. transform y first
3. rotate 90 degrees and transform x first
4. rotate 90 degrees and transform y first

In each case the area is easily found by integrating the area between $x'(0,y)$ and $x'(1,y)$ where

$$x' = (ax+by+c)/(dx+ey+f)$$

and y varies from 0 to 1. This gives

$$area = K*\ln(1+e/(d+f)) - k*\ln(1+e/f)$$

where $K=((ce-bf)+(ae-bd))/ee$, and $k=(cd-bf)/ee$. We use the method that gives the maximum intermediate area.

THE BILINEAR PATCH

The preceding class of transformations of the rectangle does not generate all quadrilaterals - e.g., nonplanar quadrilaterals. Since in general we cannot guarantee that all quadrilaterals are planar, we generalize to the bilinear patch.

The general bilinear patch (Fig.3a) has a parametric representation

$$x(u,v) = [u\ 1]\begin{bmatrix} a00 & a01 \\ a10 & a11 \end{bmatrix}\begin{bmatrix} v \\ 1 \end{bmatrix}$$

where $a00=(x3-x2)-(x1-x0)$, $a01=x1-x0$, $a10=x2-x0$, $a11=x0$. There are similar representations for $y(u,v)$, $z(u,v)$, and $w(u,v)$, where bij, cij, and dij correspond respectively to the aij for $x(u,v)$.

As in the preceding example we transform a bilinear patch with a 4x4 matrix multiply followed by a projection into 2-space. Hence we shall again ignore z' (but see discussion of foldover below). The transformation may be represented by the following matrix equation:

$$[x''\ y''\ z''\ w''] =$$

$$[x\ y\ z\ w]\begin{bmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{bmatrix} =$$

$$[uv\ u\ v\ 1]\begin{bmatrix} a00 & b00 & c00 & d00 \\ a01 & b01 & c01 & d01 \\ a10 & b10 & c10 & d10 \\ a11 & b11 & c11 & d11 \end{bmatrix}\begin{bmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{bmatrix} =$$

$$[uv\ u\ v\ 1]\begin{bmatrix} A & E & I & M \\ B & F & J & N \\ C & G & K & O \\ D & H & L & P \end{bmatrix}$$

After the homogeneous divide

$$x' = (Auv+Bu+Cv+D)/(Muv+Nu+Ov+P) = l(u,v)$$

$$y' = (Euv+Fu+Gv+H)/(Muv+Nu+Ov+P) = r(u,v).$$

The 2-pass algorithm gives:

(1) $f(u)=(A'u+B')/(C'u+D')$ for scanline $\bar{v}$, where $A'=A\bar{v}+B$, $B'=C\bar{v}+D$, $C'=M\bar{v}+N$, $D'=O\bar{v}+P$.

(2) For vertical scanline $\bar{x}'$, it can be shown that

$$g(v) = (A''vv+B''v+C'')/(D''vv+E''v+F'')$$

where $A''=EE'+GG'$, $B''=EF'+FE'+GH'+HG'$, $C''=FF'+HH'$, $D''=ME'+OG'$, $E''=MF'+NE'+OH'+PG'$, $F''=NF'+PH'$ with $E'=C-O\bar{x}'$, $F'=D-P\bar{x}'$, $G'=M\bar{x}'-A$, $H'=N\bar{x}'-B$.

Fig.6 shows a planar bilinear patch representing the texture in Fig.5a twisted about its center point. For this particular example, the h-pass is the identity function $f(u)=u$ and hence is not shown. Fig.7 shows the h-pass and v-pass for a nonplanar patch transformation. Note the foldover. Fig.5a is the source texture again.

All of the considerations discussed for the simple rectangle apply here also. In addition we have new problems introduced due to the higher complexity of the surface. A bilinear patch may be nonplanar, so from some views it may be double valued. That is, a line from the viewpoint through the surface may intersect the surface twice. In terms of the scanline functions, g(v) can map scanline $\bar{x}'$ back over We call this problem "foldover". It occurs at a silhouette edge of the projected surface. The solution is to compute z' for v=0 and for v=1. The endpoint of scanline x̃' which maps into the z' farthest from the is transformed first, so that later points overwrite points that would be obscured anyway. For antialiasing purposes, the location of the foldover point must be remembered and an appropriate weight computed for combining the pixel there with a background.

THE BIQUADRATIC PATCH

The highest order patch we shall discuss here is the biquadratic patch (Fig.3b). It is particularly interesting because surface patches on quadric surfaces (e.g., ellipsoids) may be represented as biquadratic patches. The parametric equation of x for a biquadratic patch has form

$$x(u,v) = [uu\ u\ 1]\begin{bmatrix} a00 & a01 & a02 \\ a10 & a11 & a12 \\ a20 & a21 & a22 \end{bmatrix}\begin{bmatrix} vv \\ v \\ 1 \end{bmatrix}$$

and similarly for y(u,v), z(u,v), and w(u,v).

$$[x''\ y''\ z''\ w''] =$$

$$[uuvv\ uuv\ uu\ uvv\ uv\ u\ vv\ v\ 1]\begin{bmatrix} A0 & B0 & C0 & D0 \\ A1 & B1 & C1 & D1 \\ A2 & B2 & C2 & D2 \\ & \cdots & & \\ A8 & B8 & C8 & D8 \end{bmatrix}$$

It can be shown, in a manner analogous to that used for the bilinear patch, that f(u) is a ratio of quadratic polynomials and g(v) is a ratio of 4th-degree polynomials. Actually there are two v-pass functions - say gi(v), i=0 or 1 - one corresponding to each of two solutions of a quadratic equation encountered in the derivation. The fact that there are two v-pass functions requires an explanation. We now turn to this and other considerations which have been added because of the introduction of higher degree surfaces.

First, we present a technique for reducing gi(v) from a ratio of 4th-degree forms to a rational quadratic polynomial like f(u). Presumably we could implement the gi(v) as they stand. However, besides being computationally nasty, they are difficult to interpret and hence hide many pitfalls. For example, the foldover problem discussed in the bilinear case could occur three times in a scanline with attendant antialiasing problems. We prefer to introduce a method which, at the cost of more memory, greatly reduces the complexity of the gi(v). It can also be applied to the simple rectangle and bilinear patch. Its utility comes however in extending the methods of this paper to higher degree - e.g., to the transformation of bicubic patches in perspective - which we reserve for a future paper.

The notion is that during the h-pass we have already computed the u's which we need in the v-pass. It is the recomputation of these u's (step (2) in the 2-pass algorithm) which makes the v-pass more difficult than the h-pass. We propose a high-precision framebuffer (e.g., 16 bits per pixel) to hold the u's as they are computed during the h-pass. Thus, if uj maps into xj' under f(u), then at location xj' in one framebuffer we store the intensity computed from the neighborhood of uj in the source picture and in another framebuffer (the one with higher precision) the value uj itself. Then during the v-pass on scanline x̃j', we merely lookup in the extra framebuffer the value of uj mapped by the h-pass into the current pixel, say (x̃j',y), on the vertical scanline. It will be the uj at (x̃j',y) in the extra framebuffer.

A difficulty which arises is that the h-pass function f(u) can cause a foldover on horizontal scanlines. This means that the intensity computed at location xj' is a function of one of two different uj's. Our solution is to have two auxiliary location framebuffers and one additional intensity framebuffer. During the h-pass a scanline is computed in an order where the deepest points are generated first, as discussed in the bilinear case. As each uj is determined, it is written into only one of the location framebuffers and the corresponding intensity is written into one of the intensity framebuffers only. This occurs until the uj corresponding to the point of foldover occurs. From this point on all uj's are stored in only the other location framebuffer and the corresponding intensities in the other intensity framebuffer. The final image is a combination of the two intensity framebuffers. In general, we believe this to be a difficult hidden surface problem and do not treat it further here.

The simplification produced by the addition of the three extra framebuffers reduces the gi(v) to

$$g(v) = (B0'vv+B1'v+B2')/(D0'vv+D1'v+D2')$$

where, for example, B0'=B0ujuj+B3uj+B6 with uj being obtained by table lookup. The problem of which gi(v) is to be used is replaced with the problem of computing in two framebuffers and solving the hidden surface problem implied.

Notice that q(v) may cause foldover in both intensity framebuffers, but in any one framebuffer there is only a single foldover per vertical scanline instead of the triple foldover implied by the original gi(v) and no auxiliary framebuffers.

We have claimed that the use of additional memory makes unnecessary the determination of the uj, the inverses of x' under f(u). To make this strictly true, we must make the following observations. A typical way to implement the function f(u) is to step along x' in equal increments (e.g., one pixel increments) and compute the inverse image u. The neighborhood of u is then used to compute the intensity at location x'. Of course, this defeats the whole purpose of avoiding inverses, assuming they can be computed at all. We propose "straightahead" mapping for implementing f(u) to avoid inverses altogether. The idea here is to step along u in equal increments, computing x'=f(u) after each increment. Let xj' be a value of x' for which we wish to know its inverse image Let ui be values of u at the equal increment points used as samples of u. Then when xi'=f(ui) is less than xj' and x(i+1)'=f(u(i+1)) is greater than xj', we either

(1) iterate on the interval [ui,u(i+1)] to obtain the desired inverse image uj, or

(2) approximate uj by uj=ui+a(u(i+1)-ui), where a=xj'-xi'.

The figures used to illustrate this paper were generated using the approximation (2) above. Filtering and sampling require integration of the intensity function of u. This integration requires computation at each ui; so the cost, if any, of straightahead implementation is a small addition to that already required.

SIMPLIFICATIONS

Much of the heavy machinery in the examples above becomes unnecessary in the following two special cases:

No perspective: It is easy to see that the division at each output pixel is unnecessary in this case - i.e., the scanline mappings are polynomials instead of ratios of polynomials.

Planar patch: If the patch is known to be 2-D then, regardless of the order of its bounding curves, there can be no foldover problem with the rigid-body transformations considered here. (Lines can completely reverse direction however (Fig.6).) Hence, no extra framebuffers are needed. The problem simplifies substantially, becoming a 2-D "warp" of a rectangular texture.

For example, a planar biquadratic patch under affine projection only (no perspective) has scanline functions of form f(u)=auu+bu+c and g(v)=dvv+ev+f and can be accomplished in only one framebuffer.

SHADING

People who have implemented hidden surface programs with sophisticated lighting models have discovered that the time spent for the lighting calculation is much greater than the time spent solving the hidden surface problem. We propose here that it may be faster to perform the light calculations on a square canonical polygon and then to transform the results.

Typically, normals for a polygon are determined and then interpolated across segments. The normal at each pixel is dotted with vectors to the lights and eye in some function to find the shading.

While framebuffers have been used to store intensities and depth values, they can also be used to store normal values. The normal values can be kept in a buffer at arbitrary resolution. The stream processor can then interpolate or approximate those normals to get normals at a higher resolution, normalize them, dot them with other streams of normals, and use the dot products in intensity calculatons. The approach is:

1. Transform eye and lights relative to canonical polygon.

2. The canonical polygon normal framebuffer is filled with normals at some resolution (say 4 by 4).

3. Generate a high resolution array of normals using cubic splines (we used b-splines) first in the vertical direction, then the horizontal.

4. Normalize the normals.

5. The stream of normals is dotted with a stream of light vectors and/or eye vectors to implement the lighting function.

6. The results are transformed into position into the final frame buffer yielding the shaded polygon.

7. If we are also doing texture mapping, then the intensity of each pixel in the texture is used as the color in the lighting function and the results

The approximation of normals with cubic curves can be done in a stream processor by using difference equations. Each overlapping set of four values can be used to generate a difference equation with a matrix multiply. Then the difference equation is used to generate all of the values. Until normalization, x, y, and z may be treated alike and independently.

CONCLUSIONS

We have presented what we believe to be a powerful new way of looking at 3-D surface rendering in computer graphics. It is based on the old notion of transforming to a canonical form, where the difficult work may be performed with relative ease, then transforming back. The success of this notion in 3-D surface graphics depends on the ease of realization of the transformations to and from canonical form. We have shown that a stream processor and the 2-pass decomposition technique give a technologically feasible realization of the notion for modern computer graphics.

There is much work to be done to fully explore this approach. This paper begins the exploration of this territory and points out several of the difficulties peculiar to it.

ACKNOWLEDGEMENTS

Although we do not know the details of their work, we are aware that Larry Evans and Steve Gabriel have been pursuing an apparently similar line of research and wish to acknowledge them here. Mike Shantz and his colleagues at De Anza Systems Inc. have also done independent work on separable transformations. They have implemented second-order polynomial coordinate transformations in hardware.

The photographs for this paper were prepared by David DiFrancesco at the Jet Propulsion Lab (JPL). The source picture in all cases was generated by Turner Whitted of Bell Labs for SIGGRAPH '79 and is used with his permission and that of the CACM.

Computing facilities at JPL were generously provided by Jim Blinn and Bob Holzman. Facilities were also provided by Tom Ferrin and Bob Langridge of the University of California at San Francisco.

REFERENCES

[1] James F. Blinn, "Simulation of Wrinkled Surfaces", SIGGRAPH Proceedings, August 1978, 286-292.

[2] Edwin Catmull, "Computer Display of Curved Surfaces", Proc. IEEE Conference on Computer Graphics, Pattern Recognition, and Data Structures, Los Angeles, May 1975.

[3] Steven A. Coons, "Transformations and Matrices", Course Notes No. 6, University of Michigan, Nov. 26, 1969.

[4] A. Robin Forrest, "Coordinates, Transformations, and Visualization Techniques", University of Cambridge, Computer Laboratory CAD Document 45, June 1969.

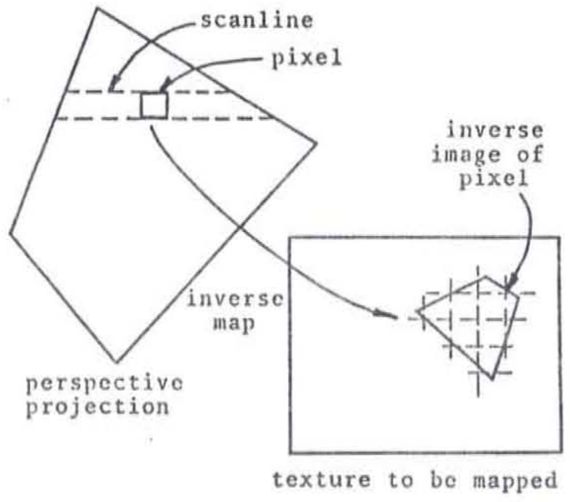


Fig.1. Texture mapping.

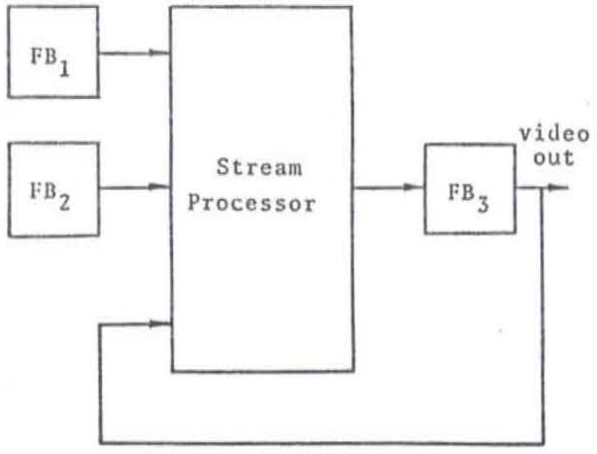


Fig.2. Stream processor.

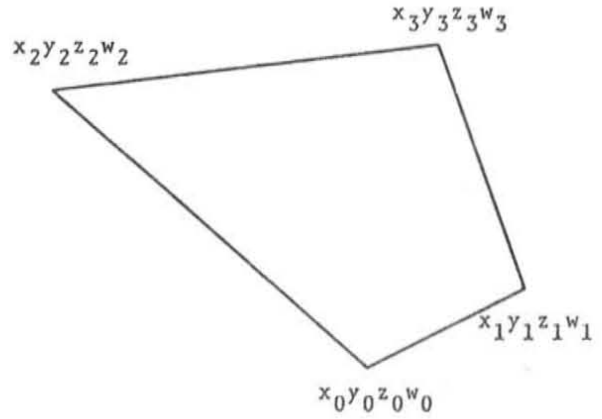


Fig.3a. Bilinear patch.

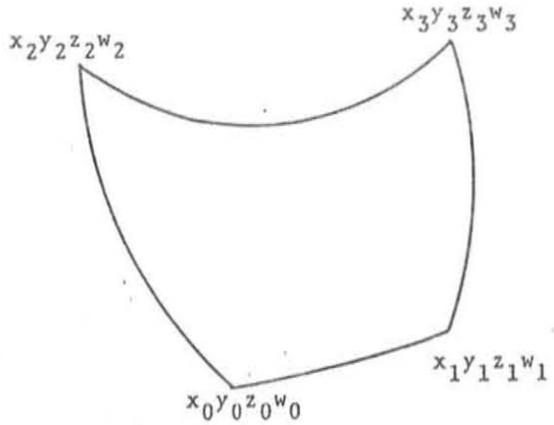


Fig.3b. Biquadratic patch.

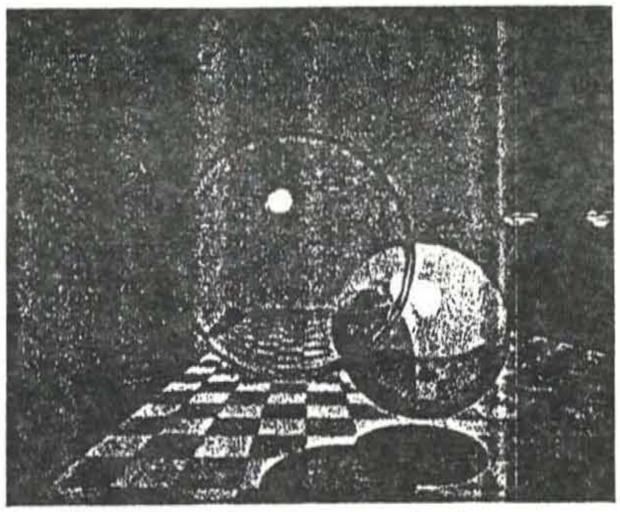


Fig.4a. Source texture by Turner Whitted. (By permission of CACM).

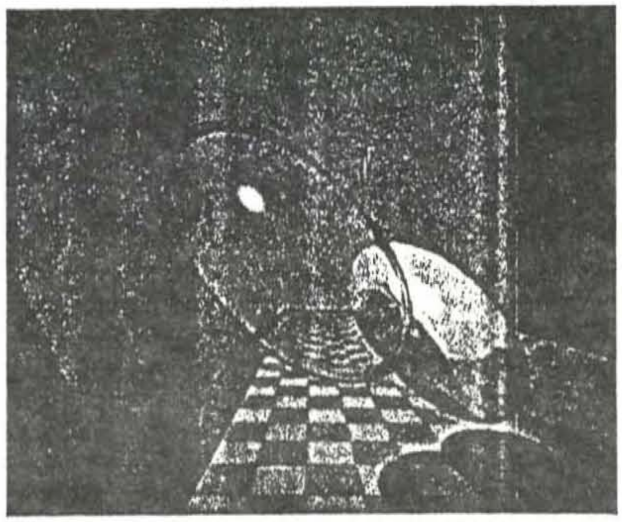


Fig.4b. Simple rotate h-pass.

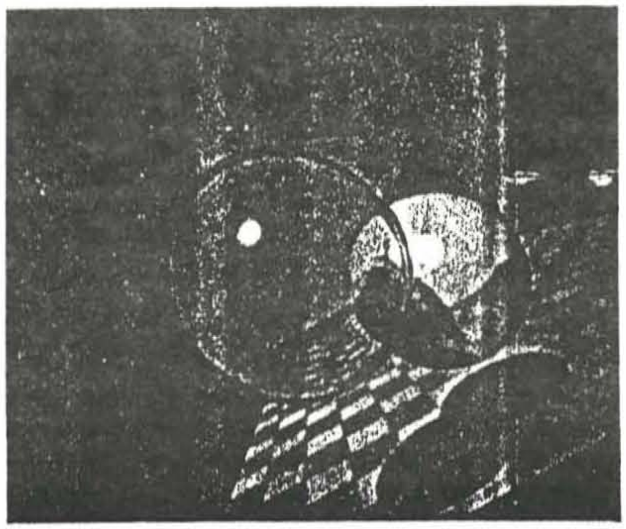


Fig.4c. Simple rotate v-pass.

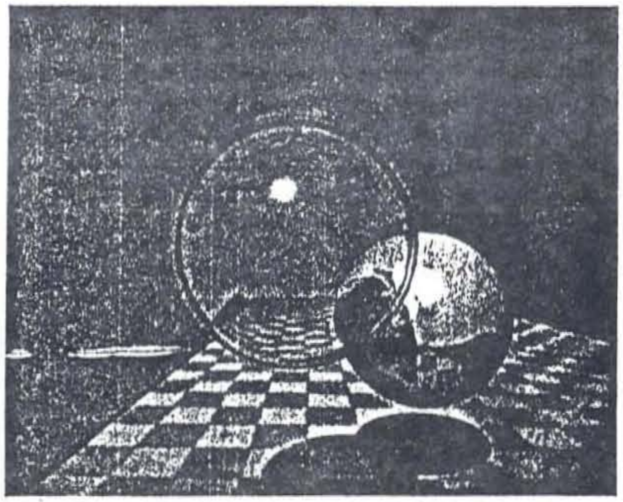


Fig.5a. Source texture by Turner Whitted. (By permission of CACM).

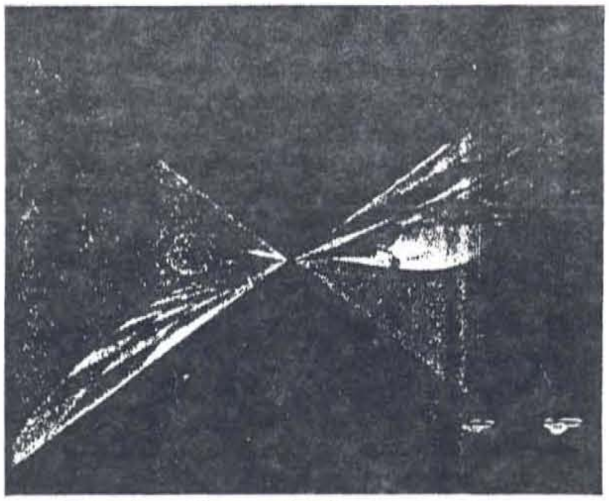


Fig.6. Planar bilinear patch twisted about midpoint.

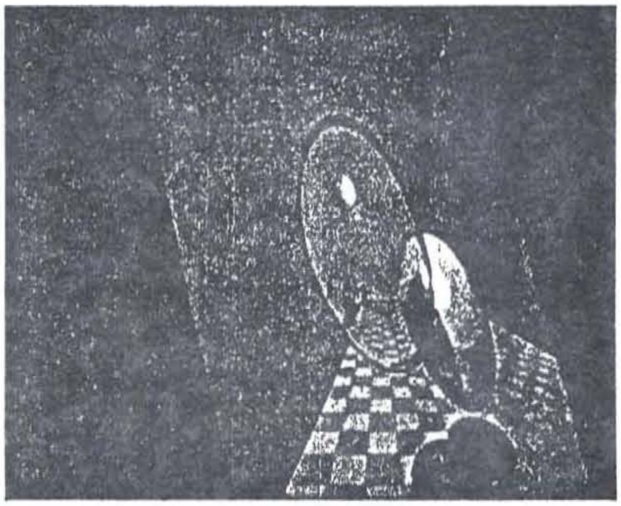


Fig.5b. Simple rectangle in perspective, h-pass.

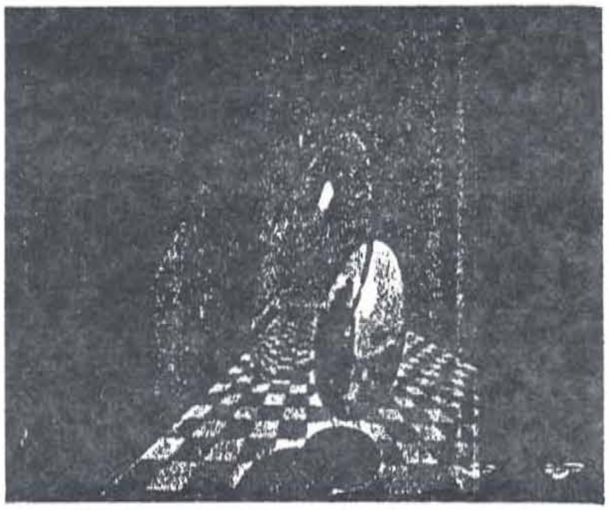


Fig.7a. Nonplanar bilinear patch h-pass.

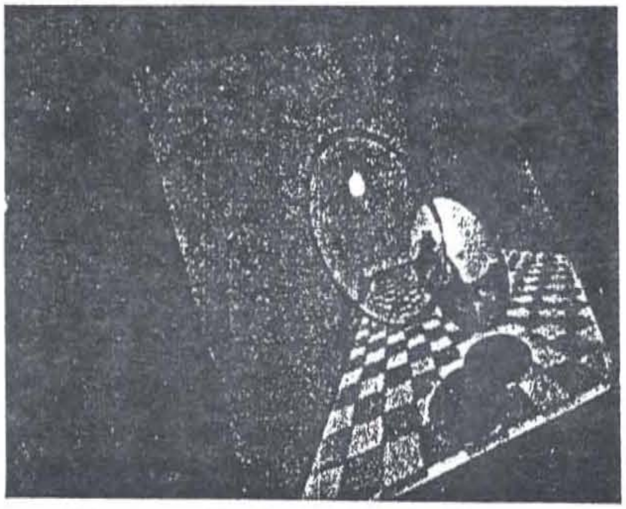


Fig.5c. Simple rectangle in perspective, v-pass.

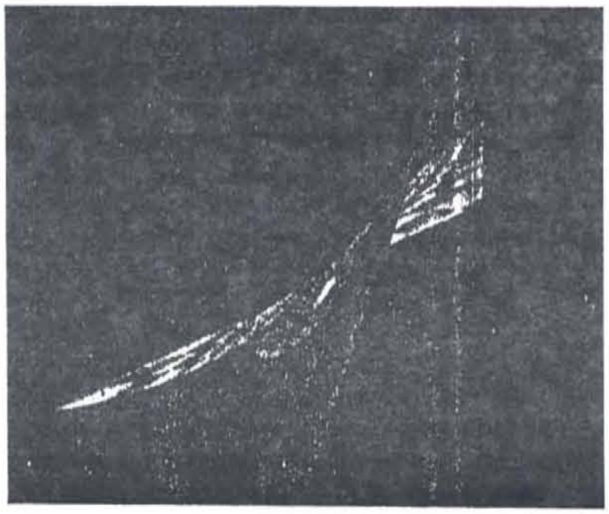


Fig.7b. Nonplanar bilinear patch v-pass.

SIGGRAPH85 TUTORIAL ON
THE USES OF SPATIAL COHERENCE IN RAY-TRACING

Space-Tracing, a Constant Time Ray-Tracer

Michael R. Kaplan
Kobra Graphics, Inc.

§1 The Evolution of Ray-tracing

It is generally acknowledged that ray-tracing is one of the most elegant, easily extensible, and powerful algorithms for realistic image computation. In terms of flexibility and the capability to model sophisticated global shading effects, it is without equal. Indeed, since ray-tracing comes closest to a direct simulation of geometric optics, most of the other algorithms for image synthesis may be viewed as approximations to it.

There are, however, a number of barriers to the general application of ray-tracing in its conventional forms. The most important are the enormous computational complexity of traditional ray-tracing algorithms, and their inherent point sampling nature. The former has made the technique impractical except for the computation of single images on relatively powerful processors, and the latter has tended to restrict the effective generation of properly sampled and filtered (anti-aliased) images. The near future, however, appears to hold out the promise for practical and high-quality ray-tracing across a wide variety of scenes, applications, and computational engines. Some of the traditional difficulties have already been overcome, while some breakthroughs are yet to be made, but they should come rapidly now that existence proofs for the more difficult problems have been publicized.

To see where ray-tracing has been, and where it is going, it is useful to briefly review some of the salient features in the evolution of non-global image synthesis algorithms. Early algorithms in this class included simple depth buffering (z-buffer), painter's, area subdivision, and scan–line hidden-surface algorithms. Later improvements, some based on earlier work, included clustering, binary space partitioning, and anti-aliased z-buffer and other sub-pixel grid based techniques. It is of course impossible in this tutorial to mention all of the important algorithms and the people who developed and improved them. It is interesting, however, to trace the general evolution of these techniques.

Early work concentrated on finding the image space areas which contained shaded pixels. The use of spatial coherence (mainly in image space) was introduced in order to speed up this part of the computation. Later work led to efficient methods of dealing with higher level primitives, such as bicubic patches. Hardware implementations of the early depth buffering techniques brought the rendering of scenes of high complexity into the realm of the single-user computation node. More recently, improvements to these same algorithms have allowed them to produce anti-aliased images.

Thus, we see that non-global image synthesis algorithms have come full-circle, with some of the former being enhanced with features of the latter in order to allow the efficient production of high-quality images. The important steps in this evolution may be summarized as; (1) Development of the basic method, (2) Application of the method to complex geometric primitives, (3) Use of coherence and knowledge about the environment being rendered to reduce computational requirements, (4) Improved shading techniques, (5) Development of low-cost, special-purpose hardware implementations, and (6) Application of anti-aliasing techniques to simple, but elegant algorithms. The next step will probably involve special hardware implementations of the new, anti-aliased depth-buffer techniques.

Let us now take a brief look at the evolution of ray-tracing. The basic methods for ray-tracing have been known for some time. Early developments naturally focused more on taking advantage of the global nature of the algorithm to produce shading effects of high-realism than on speeding up the basic computations. Recently, however, the evolution of ray-tracing has begun to bear striking parallels to the evolution of the non-global algorithms, reviewed above. Again we see developments in the areas of (1) Application of the algorithm to advanced geometric primitives (patches, prisms, fractals, clouds, etc.), (2) Use of coherence to speed processing (hierarchical object models, object clustering, scan-grid methods), (3) Improved shading techniques (distributed ray-tracing for soft-shadows, motion blur, etc.), (4) Development of hardware implementations (multiprocessor and special-purpose processor implementations), and finally (5) The development of an effective and relatively efficient anti-aliasing method (which will be presented in this seminar).

Unfortunately, the early evolution of ray-tracing did not include a technique corresponding to the image space z-buffer algorithm. The z-buffer has certain characteristics which make it extremely attractive for high-speed implementation in hardware, and, when extended through sub-pixel anti-aliasing techniques, for the production of high-quality images of high scene complexity. These characteristics include:

1. Simplicity of the basic algorithm, lending itself to hardware implementation.

2. Capability of dealing with a wide variety of primitives, when adaptive subdivision methods are used to decompose them into a common geometric type (usually polygons).

3. Ability of the algorithm to deal with scenes of extremely high complexity, especially in terms of the number of individual geometric primitive objects in the environment. This feature, probably the most important in the long run, is mainly due to the fact that the *visible complexity* of a scene is relatively independent of its *geometric complexity*. As the number of objects in a scene increases, the average visible portion of each object decreases, and the total number of visible pixels can never be more than the sampling resolution of the image space. The time to compute an image thus tends to be constant, regardless of scene complexity.

The missing element in the evolution of ray-tracing has thus been an algorithm which, like the z-buffer, makes the complexity of image computation relatively independent of the complexity of the scene, without placing undue demands on the application which produces the scene description. This paper describes a ray-tracing algorithm which has many of the desirable characteristics of an image space z-buffer, and which, when combined with developments in anti-aliasing and shading, may provide a firm base for the widespread utilization of ray-tracing in the production of high-quality images of complex scenes.

§2 Purpose of the Algorithm

Speedups to ray-tracing have generally taken three forms; (1) The speedup of an individual ray-object intersection, generally through an object-oriented scheme in which each object provides a method for efficiently intersecting a ray with its primitive type, (2) Use of coherence, particularly in object composition, to reduce the number of ray-primitive object intersection calculations, and (3) The use of special purpose hardware, taking into account the fact that in traditional ray-tracing, one ray is independent of another.

These techniques, however, have not effectively dealt with the problem of ray-tracing arbitrary scenes of high complexity, containing hundreds to hundreds of thousands of objects. In order to make ray-tracing viable for general use, scenes of this level of complexity must be computable within a "rational", predictable time, on processors of reasonable power. No matter how much the individual ray-object intersection times are reduced, the vast number of ray-object intersections which must be calculated in a conventional ray-tracer preclude the attainment of this level of performance.

Techniques have been developed, however, which do reduce the total number of ray-object intersections. Most of these methods are based on object coherence, using application generated hierarchical object descriptions, or hierarchical clustering of objects, provided by the application or automatically generated during the rendering process. The former method meshes well with applications in the mechanical engineering field, since many mechanical CAD systems deal with parts inherently described as hierarchies of primitives combined with boolean operators. The latter is less frequently used, possibly because of the difficulty of automatically generating object clustering hierarchies from arbitrary data.

The algorithm described in this paper, however, was developed for general purpose use. It has the capability of rendering a wide variety of scenes, from low to high complexity, consisting of arbitrary collections of primitive geometric objects, and on a wide range of compute engines. The performance characteristics of such an algorithm must be somewhat different from those provided by conventional ray-tracing methods. The desired characteristics are:

1. Computation time should be relatively independent of scene complexity (number of objects in the scene), so that scenes having realistic complexity can be rendered.

2. Computation time should be relatively constant for an image, depending mainly on the number of points sampled. Predictability is thus achieved.

3. The image computation time should be "rational" on commonly available processors. The definition of "rational" is, of course, dependent on the application, but a general feeling was that a few hours on a .2 megaflop (DEC VAX class) processor was within an acceptable range.

4. The algorithm should not require the application to supply hierarchical object descriptions or object clustering information. The user should be able to combine data generated at different times, and produced by different means, into a single scene.

5. The algorithm should deal with a wide variety of primitive geometric types, and should be easily extensible to new types.

6. The compute time for ray-environment intersections should be low for all rays (shadow rays and reflection rays, for example, which we refer to here as *shading* rays), rather than just first-level (hidden-surface) rays.

7. The algorithm's use of coherence should not reduce its applicability to parallel processing architectures. Instead, it is should be amenable to implementation on such architectures.

§3 The Fundamental Concept of the Algorithm

The hidden-surface problem has been described as essentially one of *sorting*, and this description goes a long way towards categorizing non-global algorithms, especially those which operate in image space. In the development of the algorithm described here, however, the global hidden-surface and shading operations of ray-tracing were categorized as essentially problems in *searching*. Since ray-tracing attempts to simulate geometric optics, its fundamental operation is that of searching space for the intersections of a ray with objects in the environment. A ray of light (or a viewer line of sight) in the "real world" does not know about the objects in its path until it strikes one. When asked to draw a picture of a chair in the corner of a room, an artist will rarely scan every object in the environment before proceeding to draw each line of his picture. The fundamental concept of *space-tracing* is thus simply, that *ray-tracing should be performed against known space, rather than the objects in it.* Only when a ray enters an area of space which is known to contain objects, and furthermore, where a ray-object intersection is highly likely, are any intersection tests performed.

§4 The Algorithm and its Implementation

The algorithm for space-tracing may be broken up into two distinct steps; pre-processing and ray-casting. It is really only a method for quickly determining ray-environment intersections, and thus does not implicitly contain any concept of shading. Each of these steps is described more fully below.

4.1 Pre-processing

In space-tracing, one minute of preprocessing can be worth an hour (or indeed many hours) of computation during the rendering step. Since ray-traced rendering involves casting at least one ray per computed image pixel, and many more if shadows, reflections, distributed ray-tracing effects, or anti-aliasing computations are to be performed, it is legitimate to perform substantial pre-processing in order to reduce ray-environment intersection costs, provided that the information developed during pre-processing can be applied to a majority of these intersection calculations. Some spatial coherence algorithms for reducing intersection costs, such as the scan-grid method, only operate on first-level rays, and thus do not speedup shading ray intersections.

The first step in the algorithm, then, is to build a data base which will allow arbitrary ray-environment intersections to be computed as quickly as possible. This data base divides all of *known space* into a hierarchical structure of cubic *boxes* aligned with the cartesian axes of the world coordinate system. It contains the information necessary to speed up the following operations:

a. Given a point in *known space*, obtain a reference to the *box* and its data which contains the point. Since space is divided adaptively and unevenly, this cannot be performed simply by indexing into a three-dimensional table of *box* references.

b. Given a ray, with origin point within a given *box*, determine the next *box* which the ray will pierce.

c. Given a *box*, obtain a list of all of the objects in the environment whose surfaces intersect the subspace described by the *box*, and which must therefore be tested for intersection with a ray which pierces it.

This data base is organized as a binary tree (a tree where each node has exactly two child nodes directly attached to it), whose non-leaf nodes are called *slicing nodes*, and whose leaf nodes are called *box nodes* and *termination nodes*. The slicing nodes contain the identification of a *slicing plane*, which divides all of space into two infinite subspaces. The slicing planes are always aligned with two of the cartesian coordinate axes of the primary space being subdivided. The child nodes of a slicing node can be either other slicing nodes, box nodes, or termination nodes. A *box node*, which is always a leaf node, describes the cubic area of space which is reached by passing through all of the binary decision points of the slicing nodes above it. The entire data structure will be henceforth referred to as the *BSP tree*.

The slicing nodes of the BSP tree are used to implement operation (a), above. The box nodes are used to implement operations (b) and (c). In order to accomplish this, each box node contains a list of the objects whose surfaces intersect it, in addition to a spatial description of the cubic subspace which it defines. The objects referenced in the box nodes are called *intersection objects*, because they contain all of the information necessary for the ray-object intersector to function. This information includes:

1. A reference to the *intersection method*, which is a procedure that the object uses to intersect its surface with a ray. In this way, the ray-object intersections can be performed in an object-oriented fashion, and each object can test for ray intersections using an algorithm which is particularly suited to its geometric type.

2. A reference to the *instance data*, which describes the particular instance of the geometric data type whose surface pierces the box.

Building the BSP tree is not a difficult task, and proceeds in the following manner:

1. Generate a list of intersection objects for the environment. This list may be created directly by the user of the software, or may be procedurally generated from higher-level object descriptions as the following steps are performed.

2. Ask each intersection object to return its spatial limits in the world coordinate system. This information is used to determine the limits of known space. Note that the synthetic camera which is "photographing" the environment must also be included in the limits of known space.

3. Starting with a box which encompasses all of known space, and which is aligned to the world coordinate axes, ask each intersection object whether its surface intersects the box. Obtain in this manner a list of all of the intersection objects whose surfaces intersect the box. If this list contains more than a previously specified number of elements (usually only one), and the box is larger than a previously specified minimum size, then subdivide the box into 8 subboxes, create the slicing nodes necessary to describe them, and recursively call the box subdivision algorithm with each of these subboxes. In this way the entire BSP tree is built. When a box is subdivided, only those objects which intersected it are passed to the subdivision calls for its subboxes and all of the other objects in the environment are culled from consideration by its subboxes. This gives the BSP tree growing algorithm approximately log complexity.

4.2 Ray-Casting - Calculating Ray-Environment Intersections

Once pre-processing has been completed, the data base it generates can be used to quickly determine the intersections which a ray makes with the objects in its environment. The intersection points found in this manner will be exactly the same as if they had been computed using the naive ray-tracing method of attempting to intersect every ray with every object in the environment. Ray-environment intersection in space-tracing is performed as follows:

1. Traverse the BSP tree, comparing the (x,y,z) values of the ray's world coordinate system origin with each of the slicing nodes encountered in the tree. Decide at each slicing node whether to branch to the left or to the right, depending on the relationship between one of the origin's coordinates and the location of the slicing plane. Eventually a leaf node will be reached. This may be a box node which describes the box which contains the ray's origin. If the ray's origin is outside of known space, a terminating node will be reached, and further tracing of the ray may be abandoned, or it may be traced for reflection mapping outside of the world environment.

2. Ask each of the intersection objects to find the intersection point between the ray and its

surface. Note that the intersection routine also knows the spatial coordinates of the box being processed, and can limit its intersection processing using this information, if such information is useful to the particular intersection method. If an intersection point is found, make sure the point is within the box under consideration. This may not be true.

3. Sort the list of intersections from step 2, and find the closest intersection of the ray against the objects in the box. *This point is the closest intersection point of the ray with any surface in the environment.*

4. If no intersection point was found in step 2, then the ray did not intersect any objects within the current box. Find the point where the ray leaves the box by intersecting it with the boundaries of the box. This is a simple intersection to compute, since the box is aligned with the axes of the world coordinate system. Now, *push* the ray just past the boundary of the box to obtain a new origin point for the ray, and proceed with step 1 above.

§5 Application of the Ray-Casting Algorithm to Shaded Picture Generation

Although many papers have been written which describe the application of ray-casting to realistic image generation, and all of these methods can be applied using the ray-casting methods provided by space-tracing, there are a number of unique features of the algorithm with respect to picture generation which should be mentioned.

One of the primary applications of ray-casting in picture generation is found in the production of shadows, particularly when the light sources are part of the scene. In this case it is not enough to pre-project the light sources into the environment, although this method works well for light sources at infinity. When ray-casting is used for shadow generation, a ray is generally sent from the object's surface in the direction of the light source. If the ray intersects another object *before* the light source is encountered, then the surface is in shadow. Unfortunately, since traditional ray-casting must generate a sorted list of all intersections of the ray in its environment, or at least a partial list, in order to perform the closeness test, shadow generation can involve the computation of numerous intersections. Since space-tracing always find the *closest* intersection in the path of the ray, however, only one test need be performed in order to resolve the shadow question. If the first intersection found is not the light source, then the surface is in shadow.

Another application of ray-casting is in the display of objects composed of the boolean intersections of primitive solids or half-spaces. These calculations may also be sped-up through the use of this algorithm, although the computation is slightly more complex than that described above. Briefly, the boolean composition tree which describes the objects being modeled is still retained, as in traditional ray-casting. The intersection list of a given ray with the component solids or half-spaces in the environment is built through the use of the space-tracing algorithm, as described above. As each component is intersected, the corresponding element in the boolean composition tree is marked. When all of the intersections of the ray in the environment have been determined, the boolean tree is traversed from bottom to top, with each component already knowing whether or not it was intersected by the ray, and if so, the location of that intersection point.

§6 Other Applications of Space-Tracing

Ray-casting was originally developed as a method for evaluating the physics of penetration, not for scene generation. A number of disciplines can make use of ray-casting, although the relatively bad performance of traditional ray-casting for complex scenes has discouraged its use. Since space-tracing works well with large numbers of objects, it can be used in such applications as pipeline interference checking and mechanical motion interference checking. Three-dimensional object picking in scenes with a large number of objects can also be performed quickly using space-tracing techniques. The closest-point first property of space-tracing is also very helpful in this application.

§7 Performance of the Algorithm

The *space-tracing* algorithm for ray-environment intersection computation has been incorporated as a fundamental part of a large image computation software package, designed to operate across a wide variety of systems, and in a wide variety of applications. Results have shown the algorithm to be extremely effective in achieving the goals set forth above. Specifically, given the proper level of subdivision for a particular scene, the algorithm can be made to operate with the same level of performance as a traditional ray-tracer would in an environment containing as few as five primitive objects, *regardless of the number of objects present in the original scene*. Tests with 6,000 objects in the environment run as quickly as those with 20. Thus, on a DEC VAX 11/780 class system, with floating point accelerator (approximate performance, 1 mips integer arithmetic, .2 mflops floating point arithmetic), a 512 by 512 non-antialiased image takes in the range of 45 minutes to 1 hour to compute. This computation time is independent of the number of objects in the scene, but depends slightly on the scene's *visible complexity*. Roughly stated, the visible complexity in this case is a measure of the proportion of the rendered pixels which contain shaded surface points.

Thus, the performance improvement of space-tracing over conventional ray-tracing is roughly proportional to the ratio of the number of objects in the scene being rendered to five objects. If the scene contains 100 objects, space-tracing is approximately 100/5 = 20 times more efficient. If the scene contains 1000 objects, it is 200 times more efficient, and so on. Obviously, cases involving hundreds of thousands of primitives may strain the memory capabilities of even a large virtual memory system. But, provided these difficulties can be overcome, the performance improvement of space-tracing for such scenes would be directly proportional to the number of objects in them.

§8 Whence the Performance

Empirical evidence has demonstrated the performance of the space-tracing algorithm, but it is not entirely obvious where the performance increases over conventional ray-tracing are derived, or why the algorithm works better at all. Hence, we present in this section some of the reasons for the success of the algorithm, and an analysis of how it performs.

Space-tracing would not be usable if the pre-processing step had the same linear complexity with number of objects as ray-tracing itself. Fortunately, the rapid culling of objects during the subdivision process eliminates this problem. Furthermore, bounding volume checks can rapidly eliminate objects from consideration in the object-box intersection tests. Again, space-tracing would not outperform conventional ray-tracing if the movement of a ray through the space boxes was substantially slower than the intersection of a ray with an object. Operation counts, however, show that in the time for a single ray-object intersection test (ie. with a sphere), approximately 7 ray-box pushes can be performed. This is mainly due to the fact that the boxes are aligned with the world coordinate system's axes, and that no 4x4 matrix transformation need be applied to the ray to transform it into the coordinate system of the boxes, as is often the case for intersection with geometric primitives.

The primary reason for the performance increase, however, comes from the fact that the only areas of known space which are heavily subdivided are those which contain a high density of objects. Areas containing a low density of objects remain more or less undivided, so that rays pass quickly through them. Once a ray enters a high density area, it has the potential of encountering a large number of small boxes. Fortunately, the probability that it will soon strike a surface is also much higher in such areas, and the ray travel is quickly terminated, not by leaving the area, but by intersecting an object within it. The shadow or reflection rays emanating from these areas are also more likely to intersect a surface close to the ray's origin, so that ray travel is again quickly terminated. Therefore, the *mean path of ray travel* , as measured by the amount of computation performed between one ray intersection and the next, remains relatively constant. A similar situation arises in the case of light scattering in liquids in suspension, and in many other cases found in statistical physics and thermodynamics.

Another improvement in space-tracing over conventional ray-tracing methods is found in the fact that it always finds the closest intersection point first. It is often true that many objects in a scene have purely diffuse surfaces. If no light is transmitted through a surface, there is no need to trace a ray past the closest point of intersection of the ray with that surface. Similarly, no sorting of the intersections needs to be performed, since the intersections are always found in order of distance from the ray origin. This means that the spatial coherence of the objects in the scene is implicity utilized to sort the ray-object intersections, as well as to find these intersections quickly.

As mentioned above, it is visible complexity, more than geometric complexity, which determines the performance of the space-tracing algorithm. Since it is generally true that, as the number of objects in a scene increases, the average portion of a single object which is visible from a given point of view decreases, the visible complexity of the scene will remain relatively constant. This is another reason for the predictable performance of the algorithm. Furthermore, by finding the closest intersection first, space tracing terminates quickly if one large object covers the other objects in a scene.

§9 The Historical Bases of the Algorithm

No development comes about in a vacuum, and this algorithm is no exception. Many sources from the field of computer graphics, and some from outside of it, have been influential in its development. It would be impossible to mention all of them here, but a list is given below, in order both to credit those whose ideas have substantially influenced the author and to point the reader to the literature for further reading. Some of the references have been included in this tutorial, and are marked by an asterisk.

Bsp Trees - [Fuchs]

Hierarchical Data Structures - [Samet]

Octrees for Rendering - [Doctor, Torborg]

Hierarchical Scene Description for Rendering - [Clark]

Hierarchical Object Description and Ray-Casting - [Roth]

Hierarchical Spatial Subdivision for Rendering - [Rubin and Whitted] *

Object Oriented Ray Tracing - [Kajiya], [Hedelman and Kaplan]

Global Shading Effects - [Whitted]

Distributed Ray Tracing - [Cook]

High Speed Ray-Box Intersection - [Rogers]

§10 Related Work

Since this algorithm was developed, approximately two years ago, the author has become aware of similar work in the field. In most cases, this work has sprung from investigations into the implementation of ray-tracing on loosely coupled parallel MIMD (multiple-instruction, multiple-data stream) computer architectures. Space-tracing naturally lends itself to these architectures, since it makes use of spatial coherence in three-dimensions, while still preserving ray-to-ray independence. In one case, an algorithm strikingly similar to space-tracing was developed for sequential implementation [Glassner]. All of the references which deal with similar algorithms, to the author's current knowledge, are listed below. Again, those that are included in this tutorial are marked with an asterisk.

Adaptive Subdivision for a Parallel Architecture - [Dippe] *

Multiprocessor Ray-Tracing - [Cleary, et al.] *

Multiprocessor Ray-Tracing - [Vatti]

A Ray-Tracing System for the Hypercube - [Goldsmith]

Space Subdivision for Fast Ray-Tracing - [Glassner]

§11 References

Clark, J.H. Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM* (Oct. 1976)

Cleary, J.G., Wyvill, B., Birtwistle, G.M., and Vatti, R. Multiprocessor Ray Tracing. *Research Report No. 83/128/17, The University of Calgary* (1983)

Cook, R., Porter, T. and Carpenter, L. Distributed Ray Tracing. *SIGGRAPH84 Conference Proceedings* (July 1984) 137-145

Doctor, L. and Torborg, J. Display Techniques for Octree-Encoded Objects. *IEEE Computer Graphics and Applications* (July 1981) 29-38

Dippe, M. and Swenson, J. An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis. *SIGGRAPH84 Conference Proceedings* (July 1984) 149-158

Fuchs, H. On Visible Surface Generation by A Priori Tree Structures. *SIGGRAPH80 Conference Proceedings* (July 1980) 124-133

Glassner, A.S. Space Subdivision for Fast Ray Tracing. *IEEE Computer Graphics and Applications* (Oct 1984) 15-22

Goldsmith, J. and Salmon, J. A Ray Tracing System for the Hypercube. *California Institute of Technology* (CCP)

Hedelman, H. and Kaplan, M. - Object Oriented Ray-Tracing. *Unpublished paper* (1983)

Kajiya, J.T. New Techniques for Ray Tracing Procedurally Defined Objects. SIGGRAPH83 Conference Proceedings (July 1983) 91-102

Rogers, D. *Personal Communication*

Roth, S.D. Ray Casting for Modeling Solids. *Computer Graphics and Image Processing* 18 (1982) 109-144

Rubin, S. and Whitted, T. A Three-Dimensional Representation for Fast Rendering of Complex Scenes. *Computer Graphics* 14 (1980) 110-116

Samet, H. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys* (June 1984) 187-260

Vatti, R. Multiprocessor Ray Tracing. *Master's Thesis, The University of Calgary* (1984)

Whitted, T. An Improved Illumination Model for Shaded Display. *Communications of the ACM* 23 (June 1980) 343-349

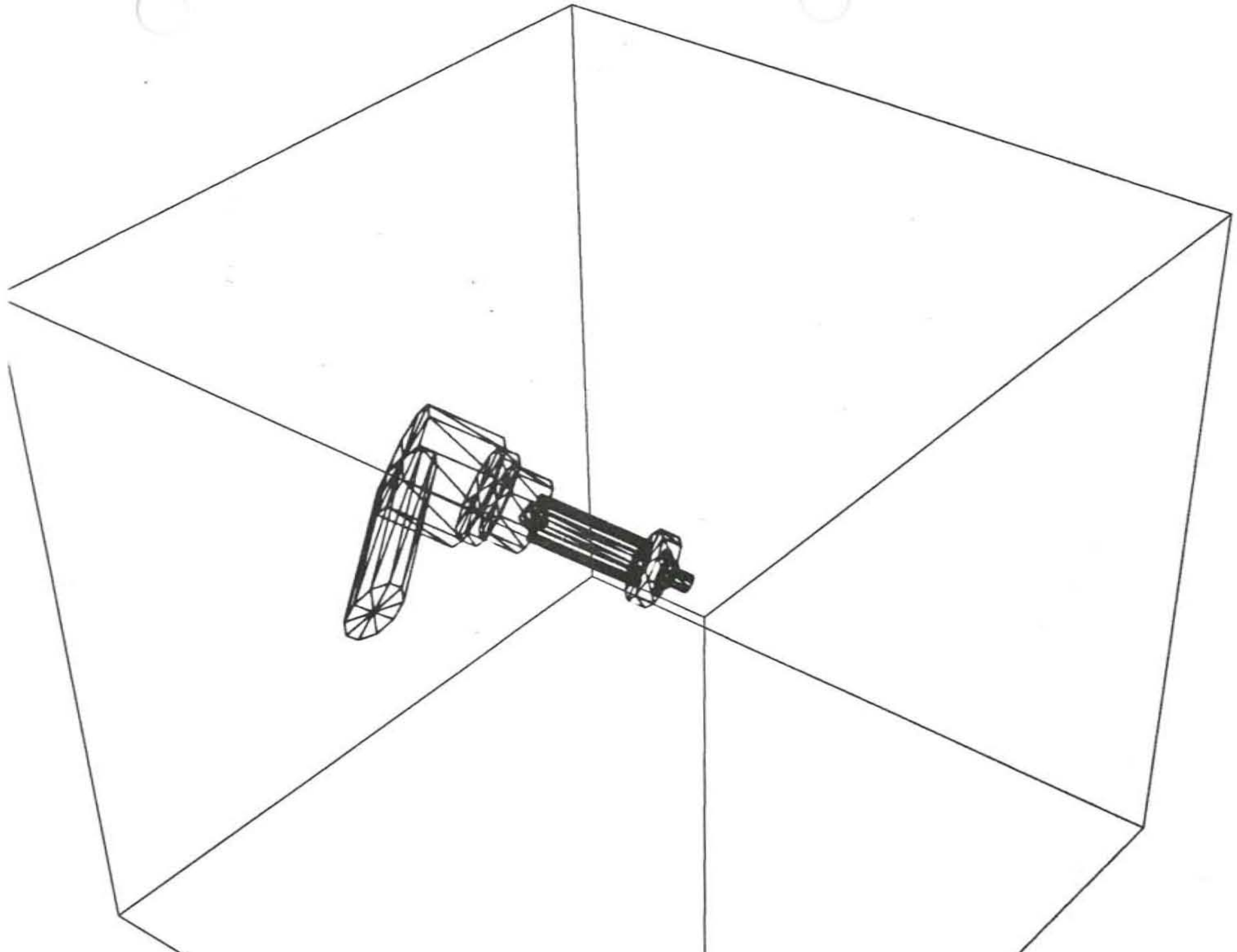


FIGURE 1

KNOWN SPACE with geometric objects. Single level of subdivision (ordinary raytracing).

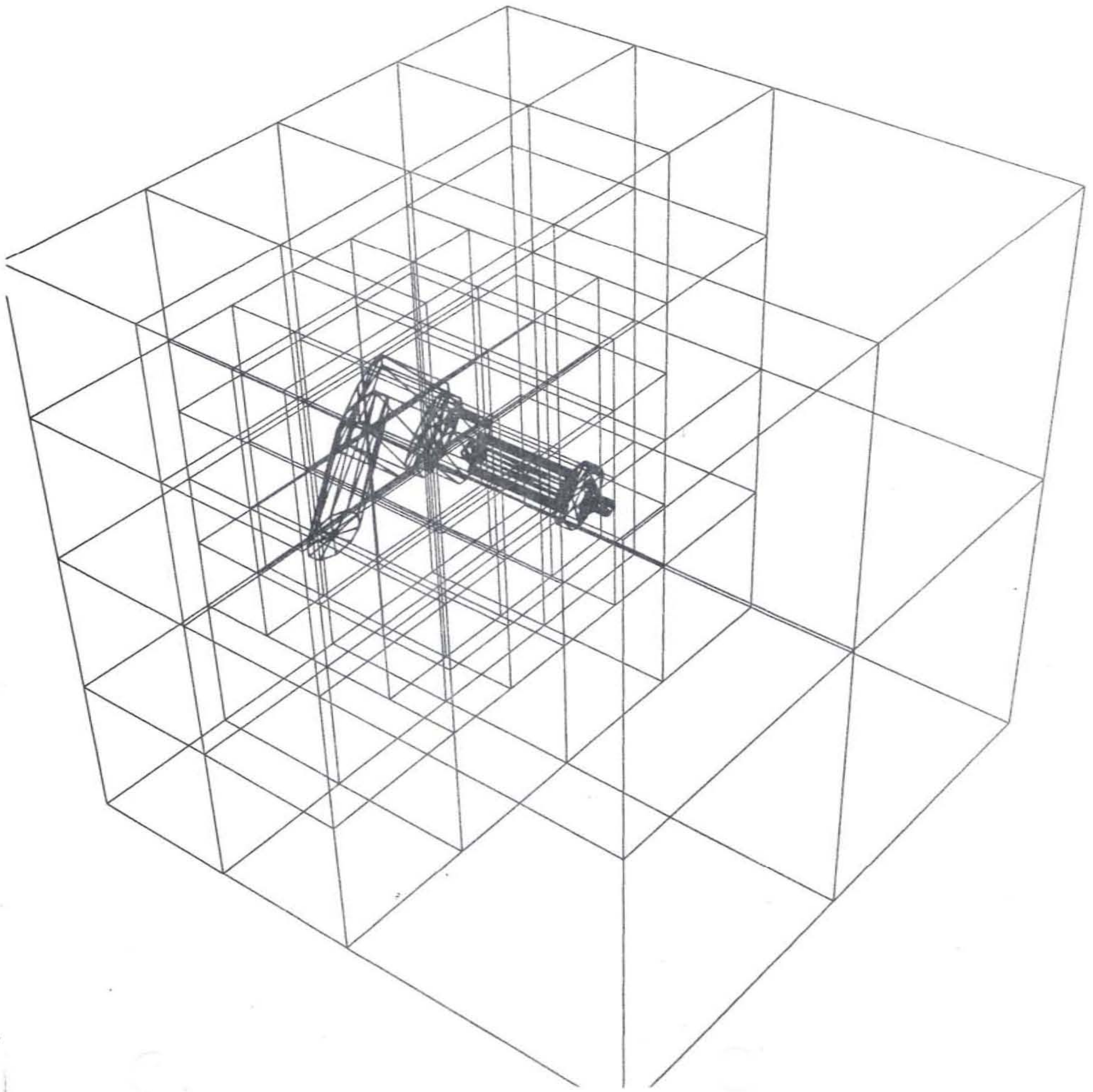


FIGURE 2

Low level of subdivision.

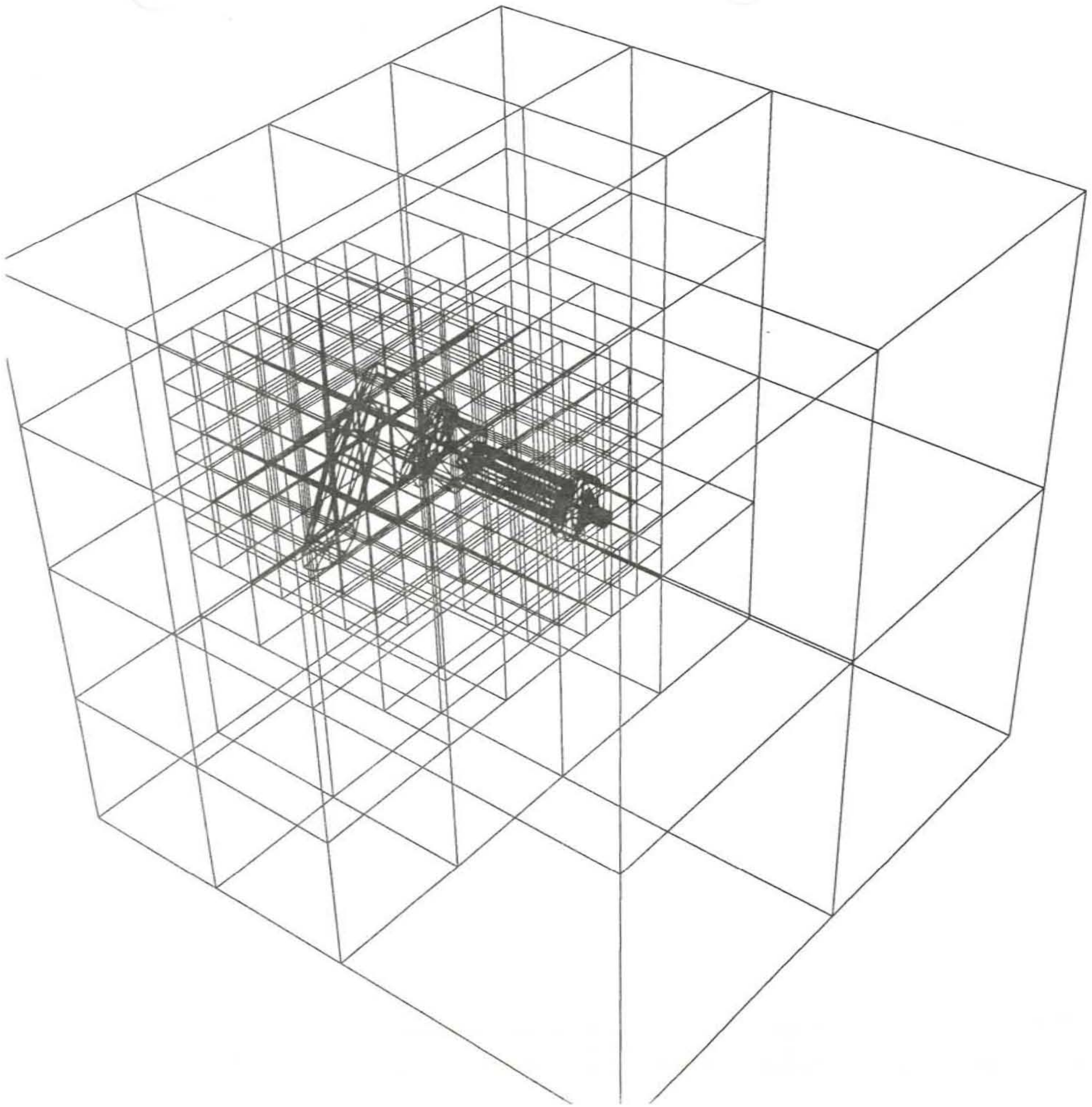


FIGURE 3

High level of subdivision.

Multiprocessor Ray Tracing

by

John G. Cleary
Brian Wyvill
Graham M. Birtwistle
Reddy Vatti

Research Report No. 83/128/17

October 1983

MULTIPROCESSOR RAY TRACING

John G. Cleary
Brian Wyvill
Graham M. Birtwistle
Reddy Vatti

Department of Computer Science
The University of Calgary

Authors address: Department of Computer Science, University of Calgary, 2500 University Drive, NW Calgary, Canada T2N 1N4.

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

ABSTRACT

A multiprocessor algorithm for ray tracing is described. The performance of the algorithm is analysed for a cubic and square array of processors with only local communication between near neighbours. Theoretical expressions for the speedup of the system as a function of the number of processors are derived. These analytic results are supported by simulations of ray tracing on a number of simple scenes with polygonal surfaces. It is found that a square network of processors generally performs better than a cubic network. Some comments are made on the construction of such a system using current (1983) micro-processor technology.

Categories and Subject Descriptors: B.3.2 [Memory Structures]: Design Styles — *Shared Memory;* C.1.2 [Processor Architectures]: Multiple Data Stream Architectures — *Multiple-instruction-stream, Multiple-data-stream processors (MIMD);* F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems — *Geometrical Problems and Computations;* I.3.5 [Computer Graphics]: Computational Geometry and Object Modelling; I.3.7 [Computer Graphics]: Three Dimensional Gaphics and Realism — *Visible Line/Surface Algorithms.*

General Terms: Algorithms, Performance, Design.

Additional Key Words and Phrases: ray tracing, parallel processing, multiple processors, animation, computer architecture, systolic array, MIMD.

1. INTRODUCTION TO RAY TRACING

The advent of cheap memory has made possible affordable raster graphics systems with the ability to display a large number of colours. These systems have permitted the generation of very realistic images representing 3D solids. Such synthetic images have many applications, including the animated film industry. The quality of these images largely depends on the algorithms used to simulate the natural visual qualities of a scene. The major features which contribute towards a realistic image are:

- hidden surface elimination;
- shading;
- shadows;
- specular reflection;
- transparency (specular transmission);
- anti-aliasing;
- perspective;
- colour.

Ray tracing is a simple way of producing very high quality realistic images on a raster display. Goldstein and Nagel [1] describe the technique as " ... basically a simulation of the physical process of photographing an object." Ray tracing algorithms have the advantage of offering a solution to all of the above problems. Some of these features, such as reflections and refractions can be included in other hidden surface algorithms, e.g. scan-line and area sub-division [7]. However, these algorithms treat such surfaces as special cases whereas ray tracing deals naturally with them.

The final picture quality provided by ray tracing has to be paid for by a large amount of computation, proportional to the number of pixels in the picture (or more to accommodate anti-aliasing) times the logarithm of the number of surfaces in the scer [6].

Ray tracing works by reversing the physical passage of a light ray. Rays are traced backwards from the eye through each pixel into the surfaces representing the scene. When the ray encounters a surface, three things may happen. If the surface is matte then the light intensity at that point on the surface will be taken as the intensity of the pixel the ray passed through. If the surface is a (partial) reflector then a new ray is started in the direction of the reflection. The final intensity of this ray will make a (partial) contribution to the intensity of the pixel. If the surface is (partially) transparent then a new refracted ray is generated passing through the surface. Again it will make a (partial) contribution to the pixel's intensity. Depending on the surface all three of these effects may occur and their respective intensities will be added to give the final pixel intensity.

If a ray emerges from the boundaries of a scene in the direction of a light source then that ray will send back a contribution towards the intensity of the pixel. Also if an 'infinitely distant' background to the scene is being used then the intensity of the background in the direction of the ray can be computed and added to the final intensity. More than one ray per pixel must be used if antialiasing is required. Whitted [8] uses four increasing the computation time proportionally.

The upshot of all this computation is that it may take anywhere from several minutes to several hours of computer time to produce a single frame. In film animation 24 frames per second have to be produced; thus a few minutes of film can take weeks to generate. This gives a strong incentive to seek ways of speeding up ray tracing. One of the properties of ray tracing is that the rays are independent of each other and a particular ray can be computed in parallel with any other ray. In this paper we propose a two or three dimensional network of processors to perform the ra tracing calculations. The execution time of the two forms are calculated theoretically and the results verified by simulation.

2. THE PROCESSOR ARRAYS

In the processor array a number of independent processors are connected by high speed links (comparable to processor speeds). The links are confined to those processors which are physical neighbours. Because the processors run independently sharing data only over the links and they execute different instructions on different data, they can be classified as MIMD (Multiple Instruction Multiple Data) systems [3, p27ff]. The great advantage of such systems is that because there is no global buss or global shared memory the system can be very easily laid out with entirely local wiring for signal paths. Provided a suitable algorithm can be found which efficiently uses the local communication links they can provide a very cheap way of constructing significant computing power. Indeed if a two dimensional network of links is used then there seem to be no limiting factors other than cost to growing the system indefinitely (although as we will see below ray tracing performance begins to decline after a certain size array is reached).

In what follows we will describe a ray tracing algorithm for cubic or square arrays of processors. In a cubic array each processor has a link to six nearest neighbours; and in a square array to four neighbours. Each processor also has a slow communication link with a host computer. If such a link is sufficiently slow there is no problem in running it over the relatively large distances needed to reach a host. This slow link will be used for a miscellany of tasks such as starting the system up, debugging and monitoring, and for communicating the final pixel intensities to a central frame buffer attached to a display.

3. SOFTWARE

Ray-tracing algorithm. There are two main tasks to be accomplished in such a system. The actual ray tracing itself and the initial loading of the scene description. We will first discuss ray tracing and then return briefly to discuss the initial scene set up.

During the ray-tracing operation the three dimensional scene is divided into rectangular volumes, which are assigned to one processor each. Each processor stores information on those parts of surfaces which pass through its own volume (that is the three dimensional scene to be represented is clipped against the volumes assigned to each processor). Each ray is represented by one packet of about twenty bytes specifying its direction and other information. Table I lists the fields needed in each ray packet together with their approximate size. The sizes vary depending on the precision of the pixel intensities, whether colour or only black and white is used, and the resolution of the display. The packets are handed from processor to neighbouring processor as the paths of the rays through the scene are simulated. When a processor receives a ray packet it checks to see if it will intercept any of the surfaces within its own volume. In the simplest case there will be no intersection. Then the ray packet is handed to the processor owning the next volume of space which the ray will pass through. (This will of course be one of the neighbouring processors). If the ray does intersect a surface then a new ray is sent in the direction of any specular reflection and of any refraction (if the surface is transparent). Finally the intensity of the diffuse reflection back along the path of the original ray is computed. This information is encoded in a *return packet* which is passed from processor to processor back to where the ray started. Each return packet contains a subset of the fields in the ray packet,

—Table I Contents of packets here —

		bytes
Ray packets:	Direction of ray	4 to 6
	Intensity	1 to 6
	Home pixel	2 to 4
	Position where ray enters volume	2 to 4
	Total	10 to 21
Return packets:		
	Intensity	1 to 6
	Home pixel	2 to 4
	Total	3 to 10

Table I Contents of packets

these are listed in Table I.

When the return packet arrives back at the front face of the processor network the intensity it carries is accumulated into the pixel through which the original ray passed. A number of such return packets may arrive for each pixel. This number depends on how often each outgoing ray is split by reflection or refraction. A ray which passes out of the edges of the three dimensional scene is checked to see if it is directed toward a light source. If it is, a return packet is sent, if it is not the ray is forgotten. Figure 1 shows the division of a scene into volumes for a 3 × 3 square array of processors. The path of a ray and its associated return packets are also shown.

The steps executed by each processor are given by the algorithm below. There are two processes involved. The first runs on all processors on the front face of a cubic array or on all the processors in a square array. It generates the initial rays through each pixel and sends the final results to the host for display. (Details of how to determine when ray tracing for a scene has finished have been omitted.) The second task runs on all the processors and receives messages from neighbours. Each incoming message can generate from zero to three outgoing messages depending on the case involved.

```
Initiation and termination process:
        receive message from host;
        case message type of:
        Start new scene:
                clear all pixel intensities to zero;
                for each pixel 'owned' by this processor
                        send ray packet;
        Finish current scene:
                send pixel intensities to host;
        end case;
```

—Figure 1. Passage of a ray through a 3 × 3 array here —

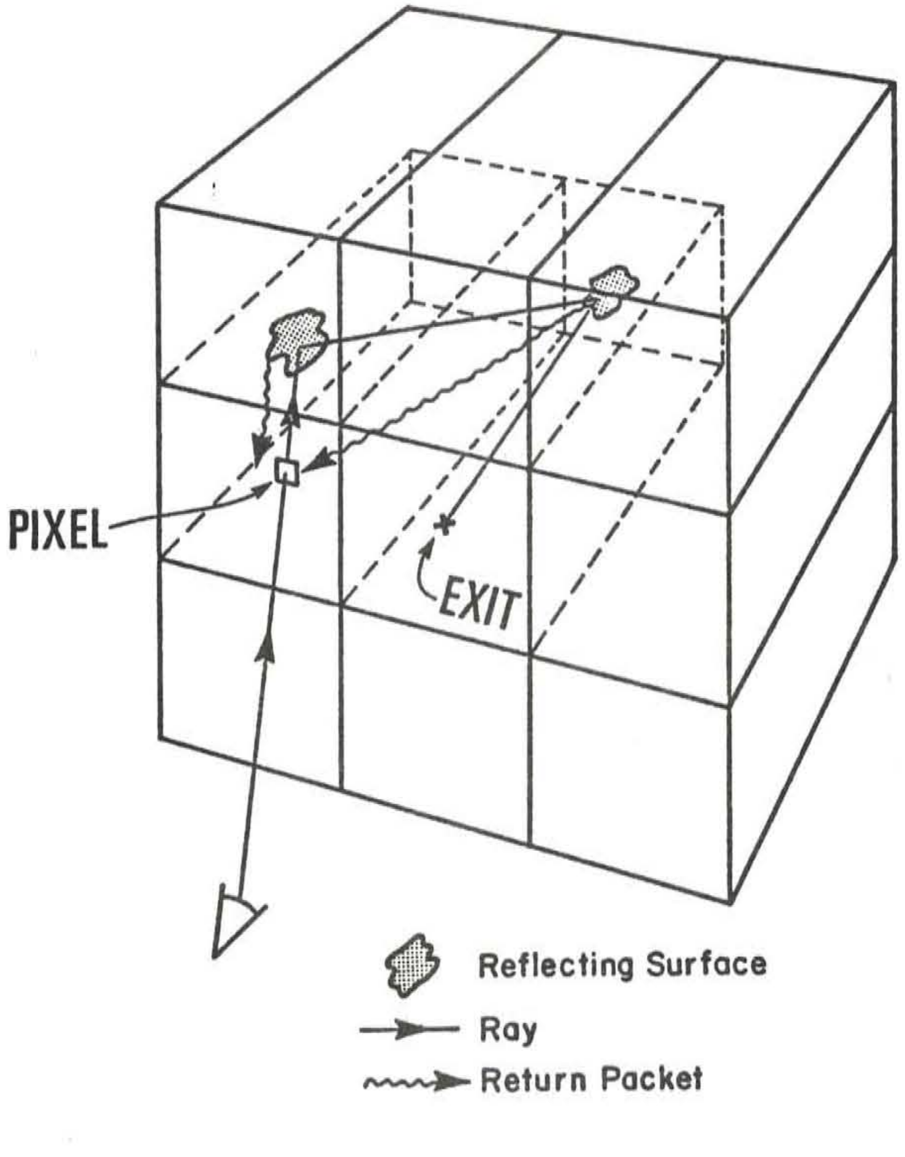


Fig. 1. Passage of a ray through 3x3 array

```
Ray processing:
        receive message from neighbours or from self;
        case message type of
        return packet:
                if packet has reached processor where original ray generated
                        add intensity into pixel intensity
                else
                        send on return packet to next processor on way home;
        ray packet:
                check ray against surfaces in current volume;
                if no intercept occurs
                        compute exit point from current volume;
                        if ray to pass out of array

                                if ray directed toward distant light source
                                        send return packet
                                else
                                        do nothing

                        else
                                send current ray packet onto appropriate
                                neighbour
                else {intercept occurs}
                        if surface is a diffuse reflector
                                compute intensity of diffuse reflection;
                                send return packet;
                        if surface is a specular reflector
                                compute intensity of reflection and
                                new direction;
                                send new ray packet in computed direction;
                        if surface is transparent
                                compute intensity and direction of
                                transmitted ray;
                                send new ray packet in computed direction;
        end case;
```

Picture set up. It is of course necessary to initialise the description of the picture in the processors. During processing the individual processors retain a record only of the parts of surfaces within their own volumes. That is the surfaces must be clipped against each of the volumes. This will be done by sending the descriptions of the individual surfaces serially from a host computer to one or more of the individual processors. Each surface need only be sent from the host to one of the processors which intersect with it from where it will be handed on from processor to processor. Those processors which intersect (part of) the surface will retain a description of that (part of) the surface and hand the description onto their neighbours. For animated

.ctures it should be possible to set up frames other than the first one by performing local increments to the positions of surfaces. So long as the number of surfaces in a scene are significantly less than the number of rays to be traced the picture set up will not be a limiting factor in overall system performance and so an analysis of its performance is not pursued further.

4. ANALYSIS OF RAY-TRACING

In the absence of actual timings on real hardware, it is impossible to predict the absolute execution times of the system. It is possible, however, to predict the relative speedup as the number of processors is increased. The total time for one picture to be traced is the time from the generation of the first ray until the last return packet is added into the frame buffer. If the number of processors is very small then the time "l be dominated by the total amount of processing to be done by the busiest processor. If however the number of processors are very large (greater than the total number of rays, for example) then the time will be dominated by the transit time for the longest ray. A lower bound is obtained by taking the maximum of the two times. In the analysis below we let T_R be the time taken by the longest ray and T_P be the time taken by the busiest processor.

These values depend on the total number of processors in the network, N, and the total number of rays to be traced, R. The results are summarised in Table II.

In a cubic network the processors form an $n \times n \times n$ network, with one face, $n \times n$, connected to the frame buffer via the host. For this configuration $N = n^3$ and each processor may be connected to up to six neighbours. The volumes associated with

— Table II Theoretical formulae for ray tracing times here —

Cubic network	Square network
T_R — **longest ray**	
$a_3 + b_3 N^{\frac{1}{3}}$	$a_2 + b_2 N^{\frac{1}{2}}$
T_P — **busiest processor**	
$R c_3 N^{-\frac{2}{3}}$	$R(c_2 N^{-1} + d_2 N^{-\frac{1}{2}})$

N — total number of processors
R — total number of rays
a, b, c — constants depending upon scene

Table II Theoretical formulae for ray tracing times

each processor are regular cubes with sides $\frac{1}{n} \times \frac{1}{n} \times \frac{1}{n}$ (the scene being traced i assumed to lie within a unit cube).

In a square network the processors form an $n \times n$ square with all the processor connected to the frame buffer. For this configuration $N = n^2$ and each processor m be connected to up to four neighbours. The volumes associated with each process are rectangular with sides $\frac{1}{n} \times \frac{1}{n} \times 1$ with the long axis of length 1 lying along the axis from the front to the back of the scene.

The significant steps in a ray tracing operation are:

- generation of a new ray;
- checking a ray for intersection with a surface within a volume;
- passage of a ray through an empty volume;
- generation of new reflected and transmitted rays if an intersection does occu
- passage of a return packet through a processor;
- addition of return packet intensity into frame buffer.

Each of these operations will take a constant (although different) time. So computi the execution time is a matter of counting how often each of these operations tak place.

Longest ray. Consider a single ray. During its passage it will pass through a numb of processors undergoing reflections and transmissions. When it finally terminates a diffuse reflector or by passing out of the scene) it will be transformed into a retu packet. The generation and termination of the ray, its ultimate addition into the fra buffer, and the number of reflections and transmissions are constant and independ of the number of processors in the system or of the total number of rays generat The number of processors passed through will vary linearly with n. The easiest way see this is to imagine that the volumes associated with each individual processor are fixed in size and that the scene is magnified as the number of processors is increased. The length of the ray then scales linearly with n and its length gives a good approximation to the number of processors it will pass through.

It is not so easy to estimate the number of surfaces that will need to be checked but which do not intersect with the ray. For the simplest algorithm this would vary linearly with the number of surfaces within a volume. However, Rubin and Whitted [6] have shown that by hierarchically decomposing the picture this can be reduced to a logarithmic dependence on the number of of surfaces in a volume. The number of surfaces will certainly be a decreasing function of the number of processors but this will be less than linear as many surfaces will intersect a number of volumes. Because of this weak dependence on the number of processors it will be assumed that the number of non-intersecting surfaces checked as a ray passes through a volume is independent of the size of the volumes and hence of the number of processors in the array. In terms of the predicted speedup this is a worst case assumption as any decrease in the number of surfaces, and so in execution time, will lead to an increased relative speedup. Further, if the effect of the decreasing number of surfaces is so great that the speedup of an array is more than linear in N then it is worthwhile for a uniprocessor to simulate the multiprocessor and so reduce the speedup to being linear in N again.

Making this assumption, all the significant events in the life of a ray take either constant time or scale linearly with n. That is $T_R = a + bn$ for constants a and b. For a square array this translates to $T_R = a_2 + b_2 N^{\frac{1}{2}}$ and for a cubic array to $T_R = a_3 + b_3 N^{\frac{1}{3}}$.

Busiest processor. Consider the busiest processor in the array. The amount of work it has to do will be proportional to the number of rays and return packets which pass through it. The number of rays will vary linearly with the total number of rays generated, R. (This assumes that increasing the number of rays will not find any radically new reflection paths through the scene. This should be a good approximation for sufficiently large R.) The number of rays passing through a volume will also be proportional to its surface area. In most cases a return packet will reverse the route of a ray, so the surface area should also be a good approximation for the number of return packets passing through the processor. So, the total computing time for a processor will be proportional to the product of the surface area of the processor and R.

For a cubic array the surface area of each volume is six times the surface area of one face with area $1/n \times 1/n$. Including the proportionality to the total number of rays to be traced this gives $T_P = c_3 R N^{-\frac{2}{3}}$ for some constant c_3.

For a square array there are two types of faces. The front and back faces have an area of $1/n \times 1/n$ and the sides an area of $1 \times 1/n$. Arguing as above this gives, $T_P = R(c_2 N^{-1} + d_2 N^{-\frac{1}{2}})$.

An effect which has not been considered in this analysis is variations in the density of rays. Consider the extreme case of a parabolic mirror in the scene facing the viewer. All the outgoing rays from the eye will be brought to a point focus. The number of rays passing through the unlucky cube which contains the focus will remain almost constant as the number of processors is increased. Little or no speedup will result then from the use of multiple processors. In less extreme cases this effect may cause the speedup as N increases to be less than predicted by the formulae above.

Speedup. The formulae in Table II contain a number of constants, which are always positive and depend on the picture being traced. Some general observations can be made from the form of the equations without knowing the exact values of the constants. In both configurations T_P is a decreasing function of N directly proportional to the number of rays to be traced. T_R however is a slowly increasing function of N. So, as N increases a point will be reached where T_R exceeds T_P and the total execution time increases as the number of processors is increased. This point can be intuitively interpreted as the point at which the costs of passing packets between processors begins to dominate the savings of having many processors. From the exact results and simulations below it seems that this turnaround point occurs when there are approximately as many processors as rays to be traced. The minimum execution time attained at this point is approximately the time for a ray to traverse the diameter of the processor network and return.

In any parallel processing system the maximum speedup attainable is equal to the number of processors. Rarely is this full speedup attained because the communication of information between processors delays results and requires processing time itself. This is also true of the current system. To see this consider the case when the number of processors is smaller than the number of rays to be traced and the execution time is determined by T_P. The speedup attained is given by the ratio of T_P for one processor to T_P for N processors. In both configurations R cancels from this expression so that the speedup is independent of R and depends only on the scene being traced and the number of processors.

For the cubic network this cancellation gives a speedup proportional to $N^{\frac{2}{3}}$ which is only slightly less than the optimum N. In the case of the square network the full speedup is attained if N is sufficiently small but approaches $N^{\frac{1}{2}}$ as N increases. Because of its initially faster speedup the square network will always out perform the cubic network for sufficiently small values of N. The results below indicate that "sufficiently small" covers most cases of practical interest.

Exact analysis of empty scene. It is possible to do an exact analysis of the ray tracing times for an empty scene. The results of this analysis for a viewpoint at distance 1 from the center of the scene are given in Table III.

The longest ray will travel diagonally through the scene intersecting approximately $n/2$ processors in a square array and $3n/2$ in a cubic array. Sharp upper and lower bounds lie within a constant of these two terms as shown in the table. These results are thus in excellent agreement with the more general analysis above.

For both a square and cubic array the largest number of rays pass through the corner processors — those furthest from the center. For the cubic array it is the corner processors on the front face which are the busiest.

For the square array the form of the results are in exact agreement with the form of the general results of the last section. The results in Table III are for a viewpoint distance of 1, however the analysis has been completed for a general viewpoint distance d. This gives the result $T_P = R\left\{(1-f)N^{-1} + \frac{1}{2}fN^{-\frac{1}{2}}\right\}$ where $f = \frac{(1+2d)}{(1+d)^2}$. As d increases the coefficient of N^{-1} approaches 1 and the coefficient of $N^{-\frac{1}{2}}$ approaches 0. This is very reasonable for with a very distant viewpoint the rays will tend to run parallel to the z-axis and there will be little message passing between processors.

For the cubic array the results for T_P are somewhat more complex. The major term is of the form $RN^{-\frac{2}{3}}$ as predicted. However there are additional terms weakly dependent on $N^{-\frac{1}{3}}$ and d. When N is about 1 or 2 the expression reduces to $RN^{-\frac{2}{3}}$ and

—Table III Exact times for empty scene here —

Cubic network	Square network
T_R — **longest ray**	
$1.5N^{\frac{1}{3}}-1 \leq T_R \leq 1.5N^{\frac{1}{3}}+0.5$	$0.5N^{\frac{1}{2}} \leq T_R \leq 0.5N^{\frac{1}{2}}+1.5$
T_P — **busiest processor**	
$RN^{-\frac{2}{3}}\left\{1 + \frac{(0.5-N^{-\frac{1}{3}})(2+N^{-\frac{1}{3}})}{\left[1+N^{-\frac{1}{3}}\right]^2}\right\}$	$R(0.25N^{-1} + 0.375N^{-\frac{1}{2}})$

N — total number of processors
R — total number of rays

Table III Exact times for empty scene

when N is large to $2RN^{-\frac{2}{3}}$. These two expressions form a lower and upper bound on T_P respectively. The values for the exact result and these bounds are plotted in Figure 2 where a uniprocessor is assigned an arbitrary time of 1. It can be seen that T_P starts at the lower bound for small N and increases to approach the upper bound as N becomes large. This departure from the exact theory is a result of variations in the density of rays. More rays pass through a given area near the front of the scene than at the rear. This effect cancels in the square array because each processor averages the density of rays from the front to the back of the scene.

An exact analysis has also been obtained for the scene with a single reflecting surface occupying all the back surface of the scene. The results for relative speedup with respect to N are essentially identical to those for the empty scene above for both the square and cubic array and so will not be further considered.

While simple scenes such as the empty scene analysed above are unrealistic they are good approximations to more realistic scenes. For example a scene which contains no (or very few) specular reflectors will be a close approximation to the empty scene so far as the relative speedup is concerned. Each ray will terminate at its first intercept and will generate a single return packet which will retrace the ray path. d can be rescaled to take account of the depth at which the intercepts occur. For example if the rays all terminate about half way into the scene ($z = \frac{1}{2}$) then d should be doubled for the square array and remain unchanged in the cubic array (this will tend to further bias the speedup results in favour of a square array over the cubic).

—Figure 2. Exact and simulated values of T_P for empty scene here —

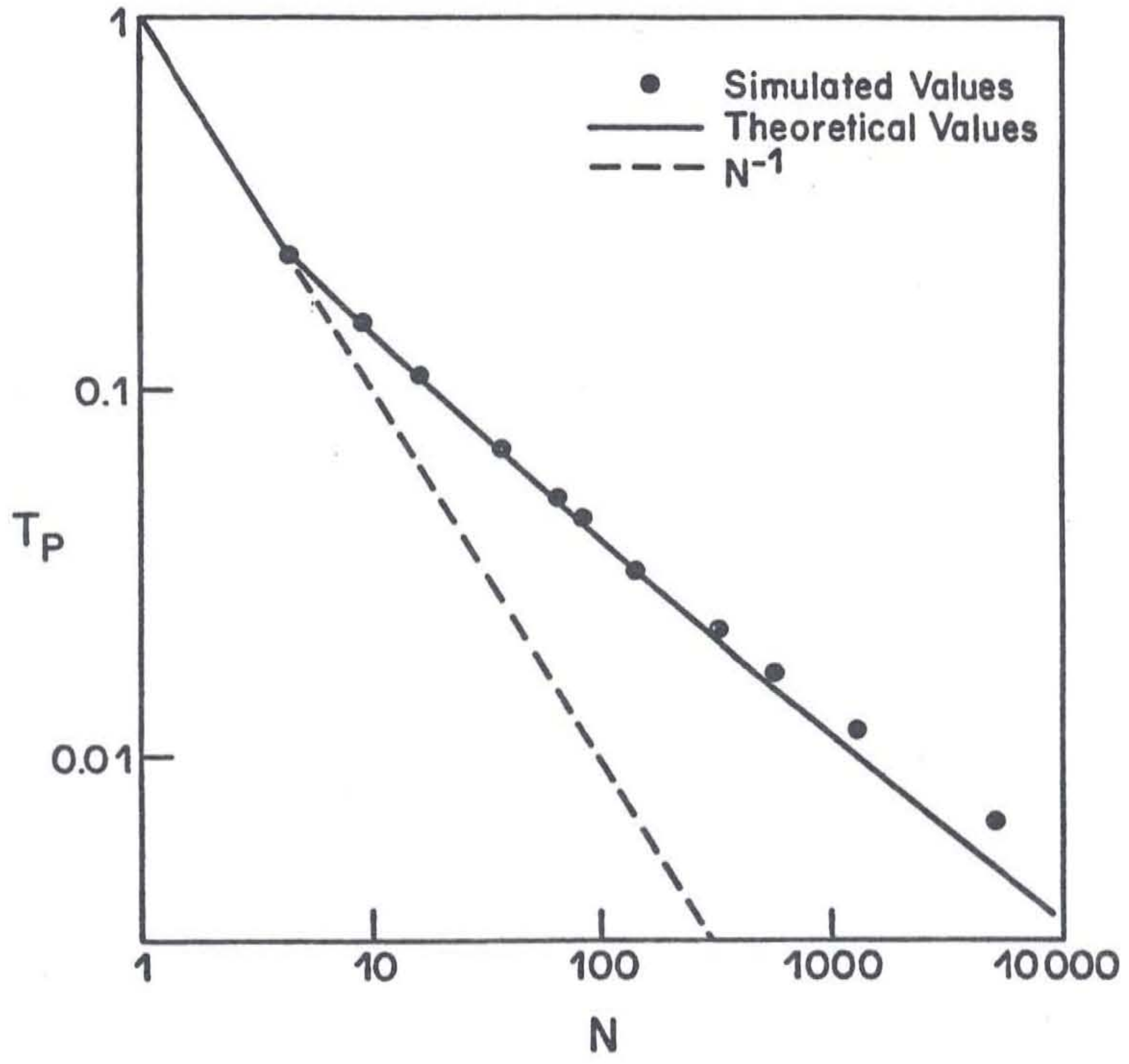


Fig. 2(a) 2D array

Fig. 2. Exact and simulated values of T_p for empty scene

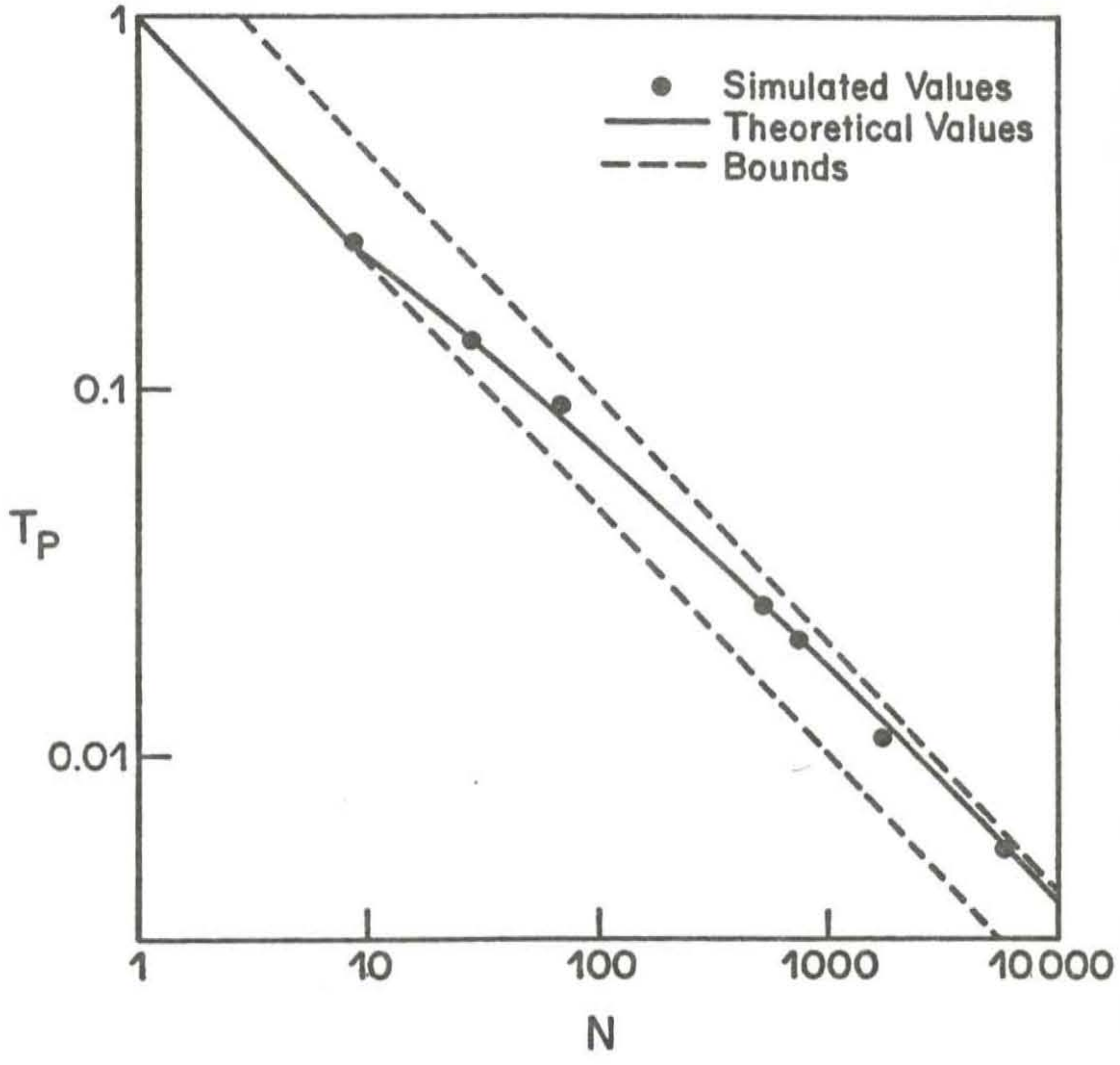


Fig. 2(b) 3D array

Fig. 2 Exact and simulated values of T_p for empty scene

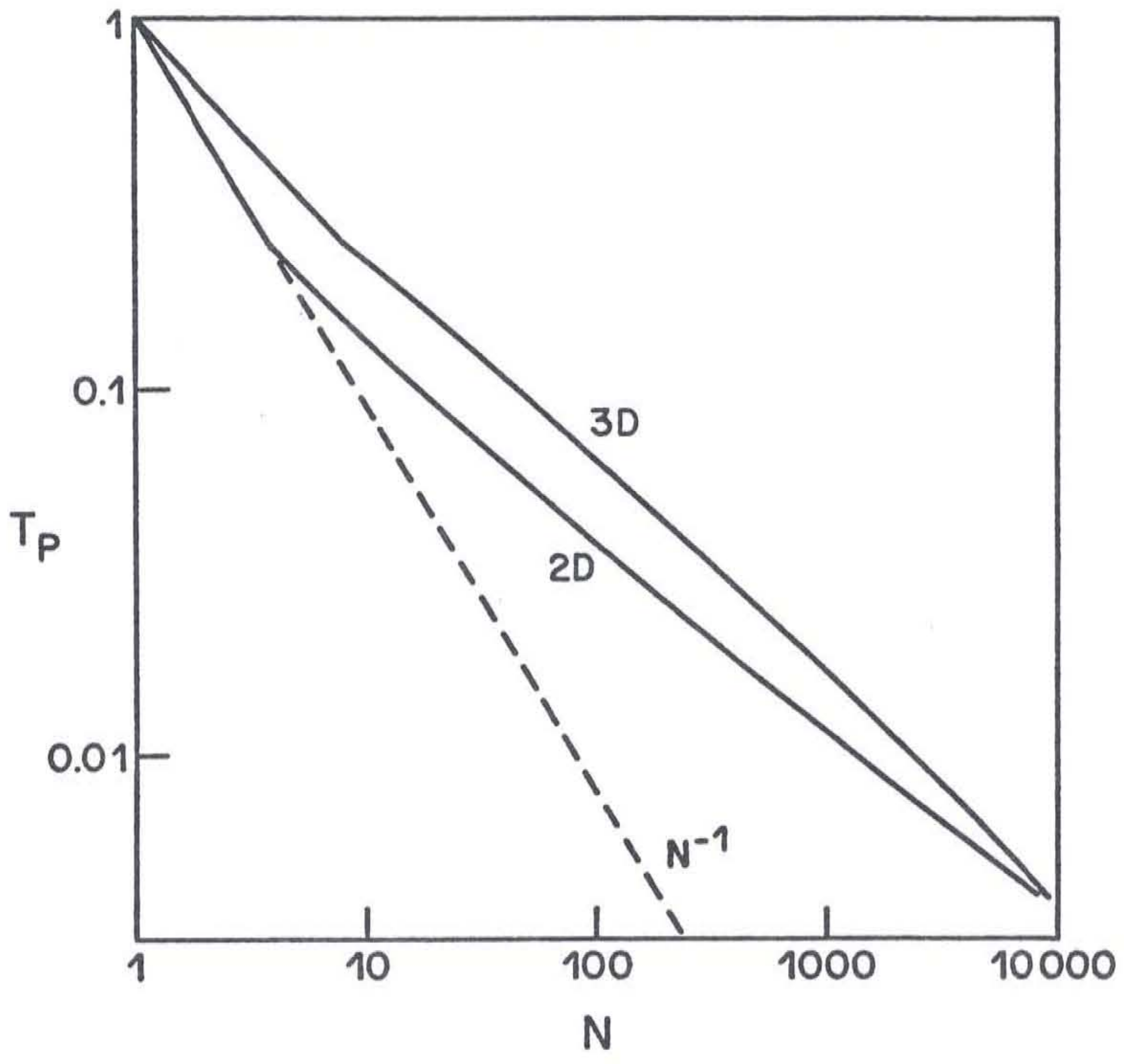


Fig. 2(c) Comparison of 2D and 3D theoretical values

Fig. 2 Exact and simulated values of T_p for empty scene

5. SIMULATION OF RAY TRACING

Techniques. **Ray tracing has been simulated for three simple scenes: an empty scene (all rays terminate by leaving the scene); a single square reflecting surface occupying all the back face of the scene; and a 'cylinder' composed of ten polygons. The first two are those for which an exact analysis was given in the last section.**

Results.

The results of the simulation were basicly to extend the conclusions of the last section to the more complex cylindrical scene. As predicted values of T_R were negligible compared with T_P up to the simulation limit of 10,000 processors (assuming a reasonable number of rays, say, 512 × 512 or more). The proportionality of T_P to R was verified by rerunning the cylindrical scene for different values of R. In no case did the values for T_P vary by more than 10%, most of this occurring when the number of processors approached the number of rays when statistical fluctuations can be expected to become significant.

As can be seen from Figure 2 the theoretical and simulated values of T_P agree well. The only significant departure is when N is above 1,000. For the simulations in question R = 72×72 of the same order as the number of processors. Absolute T_P values of about 30 were obtained then (counting 1 for each passage of a ray through a processor), so statistical fluctuations can be expected to be significant.

The simulation results for the wall and cylinder scenes were essentially the same as the empty scene and are not repeated here. The only significant difference in relative speedup was that there was a small tendency for the square array to perform relatively better than the cubic on these two more complex scenes. This resulted in a deferment of the point at which the two crossed over to about $N \approx$ 12,000 for the wall

and to $N \approx 30{,}000$ for the cylinder.

6. PHYSICAL CONSTRUCTION

In the system described the only high speed communication is between nearest neighbours. This allows the processors to be laid out in a uniform way with only local wiring connections. This property is the essence of systolic systems [4] and allows of cheap and easy construction. This systolic approach may be contrasted with that of MIMD systems with shared global memory such as the New York Ultracomputer [3]. To achieve such a global memory requires a switching network which "is likely to be the most expensive component of the completed machine" because of its complexity and the need for non-local wiring. Gottlieb has noted that such general access to global memory is very useful for a general purpose computer as it greatly eases programming. However, by tailoring the network to suit a single special purpose algorithm considerable improvement in the cost effectiveness of the hardware seems possible.

The results above indicate that a square network of processors will perform better than a cubic network. This is fortunate as it is significantly more difficult to construct the latter. One reason for this is the need to cool the system. As a cube grows it becomes more and more difficult to extract heat from the middle. In a similar way, it becomes more difficult to gain access to the middle for hardware debugging and maintenance.

Systolic algorithms and networks are often mentioned in the context of VLSI circuit technology for which they are particularly well suited [5]. Some form of special purpose VLSI ray tracing processor might be possible, although each processor will require substantial amounts of memory to store the surface descriptions (128K bytes in the system described below). It is thus likely that the system area and chip count will be dominated by the memory obviating many of the advantages of a special processor.

Actual hardware for ray tracing is currently being designed and built at Calgary as part of the JADE project to construct development and simulation software for distributed systems, [9]. Each processor will occupy one board and will have a 10MHz M68000 processor, 128K bytes of memory, and an RS-232 communication line (for slow communication with the host). The fast interprocessor links are to be implemented using 4K byte dual ported memories. Because the communication via them is simple unidirectional message passing neither test and set operations nor interrupts need be implemented. Detection of incoming messages will be done by the receiver polling its dual ported memories. Each processor will have access to four of them, three of which will have memory on board and three off on their paired board. It is hoped to build a square 10×10 array.

7. MEMORY CONSTRAINTS

As we have seen the speedup of the ray tracing algorithm is less than linear in the number of processors, especially for 10 or more processors. This implies that rather than use the algorithm we have described it would be better to duplicate the entire ray tracing calculation into each of the available processors. Each processor would then contain a description of the entire scene and process the rays for $1/N$ of the pixels to achieve the full speedup of N. The limit on this strategy is the amount of memory available. We expect that a simple polygonal surface will need about 100 bytes to describe it. The system to be built has 128K bytes of memory on each processor allowing scenes with up to 1000 surfaces. It is expected that for serious work scenes with up to 100,000 surfaces will be needed, obviously too many to fit. To increase the memory on each processor is expensive and would require more than one board per processor — further increasing cost and complexity. The strategy indicated then is tc pack each scene into as small a group of processors as possible and replicate this group across the available processors. Investigations are being undertaken into algorithms to do this and the performance tradeoffs involved.

ACKNOWLEDGEMENTS

We would like to thank Ian Witten for discussions about the hardware construction of microprocessor networks.

REFERENCES

1. Goldstein, R. A., and Nagel, R. 3-D visual simulation. *Simulation 16*, 1, (January 1971), 25-31.
2. Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L., and Snir, M. The NYU Ultracomputer — designing an MIMD shared memory parallel computer. *IEEE Trans. on Computers, C-32*, 2, (February 1983), 175-189.
3. Hockney, R. W., and Jesshope, C. R. *Parallel computers*. Adam Hilger Ltd., Bristol, England, (1981).
4. Kung, H. T. The structure of parallel algorithms. In *Advances in computers, 19*, Yovits, M. C. (Ed.). Academic Press, New York, (1980), 65-112.
5. Mead, C., and Conway, L. *Introduction to VLSI systems*. Addison-Wesley, Reading, MA, 1980.
6. Rubin, S. M., and Whitted, T. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics 14*, (1980), 110-116.
7. Sutherland, I.E., Sproull, R.F., and Schumaker, R.A. A characterization of Ten Hidden Surface Algorithms. *Computer Surveys 6*, 1, (March 1974).
8. Whitted, T. An improved illumination model in shaded display. *CACM 6*, 23,

(June 1980), 343-349.

9. Witten, I. H., and others. JADE: A distributed software prototyping environment. Research report 83/120/9. University of Calgary, Department of Computer Science, (April 1983).

A 3-Dimensional Representation for Fast Rendering of Complex Scenes

Steven M. Rubin
Turner Whitted

Bell Laboratories
Holmdel, New Jersey 07733

ABSTRACT

Hierarchical representations of 3-dimensional objects are both time and space efficient. They typically consist of trees whose branches represent bounding volumes and whose terminal nodes represent primitive object elements (usually polygons). This paper describes a method whereby the object space is represented entirely by a hierarchical data structure consisting of bounding volumes, with no other form of representation. This homogeneity allows the visible surface rendering to be performed simply and efficiently.

The bounding volumes selected for this algorithm are parallelepipeds oriented to minimize their size. With this representation, any surface can be rendered since in the limit the bounding volumes make up a point representation of the object. The advantage is that the visibility calculations consist only of a search through the data structure to determine the correspondence between terminal level bounding volumes and the current pixel. For ray tracing algorithms, this means that a simplified operation will produce the point of intersection of each ray with the bounding volumes.

Memory requirements are minimized by expanding or fetching the lower levels of the hierarchy only when required. Because the viewing process has a single operation and primitive type, the software or hardware chosen to implement the search can be highly optimized for very fast execution.

KEY WORDS AND PHRASES: computer graphics, visible surface algorithms, hierarchical data structures, object descriptions

CR CATEGORIES: 8.2, 6.22

Introduction

With increases in display resolution and processing power, computer generated images have become more complex. The state of the art has progressed far beyond "stick figure" representations of scenes to the point where details of realism are sought. In addition, the objects to be displayed are often complex, making the display process both time and space intensive.

The most popular approach to complex image generation has been to approximate surfaces with collections of polygons. Generally, a good approximation requires an enormous number of polygons and a corresponding amount of display time. To help address the problem of complex object descriptions, alternative representations have been used. Complex surface representations such as quadric and cubic patches [8,9] are well suited to curved objects but are not general enough to model an arbitrary scene. Another alternative, procedural representation [10], is completely general and relatively compact but still requires a mechanism for generating and displaying non-procedural primitive surface elements. Three dimensional "point" representations [4,6] are also completely general, relatively easy to display, but incredibly wasteful of memory. Hierarchical representations which decompose the object space into repeatedly simpler subspaces [1,2,3], are a promising form which allow arbitrary scene descriptions in an easily usable data structure. This paper will present a new twist on hierarchical representations of scenes that has many computational advantages.

Clark [1] proposed the use of hierarchical geometric representations to speed both clipping and visibility calculations. Each level of the hierarchy consists of bounding volumes which enclose the lower levels. At the bottom of the tree, object representations are encoded in some conventional form such as polygons. He introduced the notions of "resolution clipping" and "graphical working set". According to these notions, levels of the data structure which describe detail at a greater resolution than can be resolved in the image are clipped from the current object description along with those portions of the scene which lie outside the viewing area.

In its "multi-stage combinatorial geometry model", MAGI [3] utilizes a tree structured object description. The branches of the hierarchy are arbitrarily oriented rectangular parallelepipeds that enclose subvolumes of the object (see Figure 1). It is again necessary to switch to an alternative form of representation at the bottom level in order to describe the object. The visible surface algorithm is a ray tracing technique in which the inverse of the coordinate transformation of each bounding box is applied to the ray at each node in the tree. The hierarchical structure reduces the number of candidate objects against which the ray must be tested for intersection, and the successive transformations insure that the intersection calculations will be simple ones. Appropriately, the objects being displayed are trees (e.g. deciduous or coniferous instead of binary). Because of the enormous complexity of these objects, image generation times are several hours.

Reddy and Rubin [2] present three forms of hierarchical decomposition which, individually or in a combination of two, are sufficient to completely describe an object. At the initial levels of the hierarchy, rectangular parallelepipeds (like the MAGI system) partition the object space. At lower levels of the hierarchy, one of two other representations can be used. The first low-level representation divides this sub-volume of the object space into eight equal sized subspaces by placing a partition in the middle of each axis. These eight subspaces are either empty, full, or further subdivided in the same binary manner (see Figure 2). The second low-level representation also divides the object subspace with partitions perpendicular to the axes. In this scheme, however, there can be multiple partitions along each axis at arbitrary locations (see Figure 3). Although the point accessing algorithm is more expensive in the second representation, less space is needed to represent an object due to the flexibility of the partitioning.

The representation proposed in this paper constructs the object space completely out of hierarchically structured subspaces. The terminal nodes do not contain another type of primitive, but are themselves displayable. Some of the advantages of this uniformity are an ability to examine the object at arbitrary magnification and the flexibility of combining common subspaces that share micro-descriptions.

The subspaces that we have been using are rectangular parallelepipeds. This simple unit can be described and traversed with a single transformation matrix, thus lending the process well to easy and efficient hardware implementation. The next section discusses this representation in detail.

Representation

Conventional visibility calculations suffer a combinatorial explosion when confronted with complex object descriptions [7]. Typical algorithms for hidden surface elimination require that each surface in the scene be sorted into a computationally effective order and then compared with a neighborhood of other surfaces. As the scene becomes complex, the neighbor-

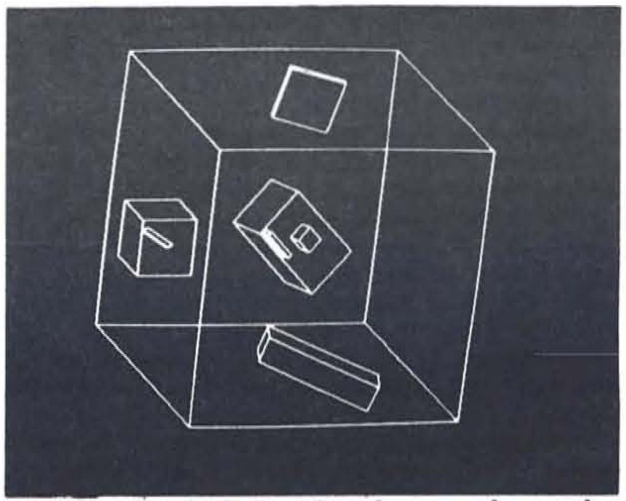


Figure 1: Arbitrarily oriented rectangular parallelepipeds modeling a hierarchically described object space.

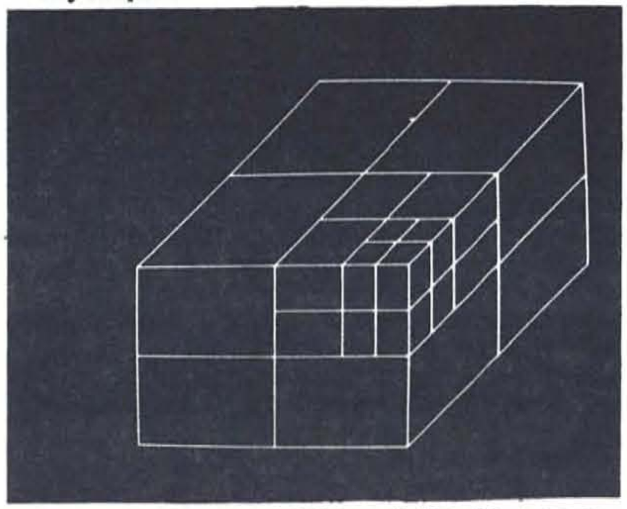


Figure 2: Binary subdivision of the object space along the X, Y, and Z axes.

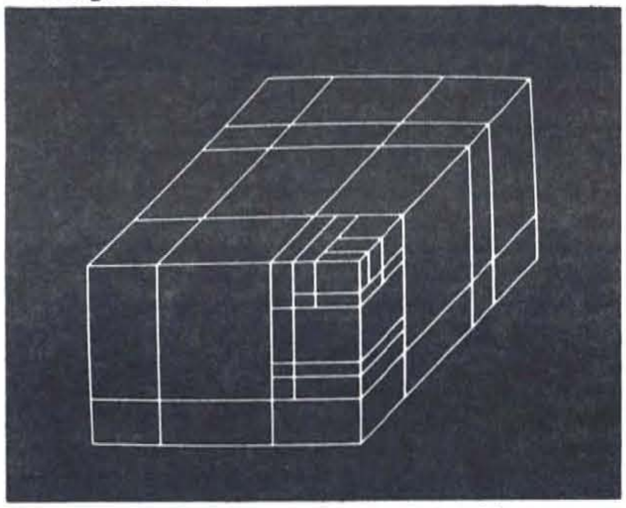


Figure 3: Unequal subdivision of the object space with arbitrarily placed planes perpendicular to the X, Y, and Z axes.

hood size becomes combinatorially unmanageable. A ray tracing algorithm, on the other hand, executes in a time that increases linearly with the number of objects in the scene. This method determines visibility by extending lines (rays) from the image plane into the object space. Whenever a ray intersects one or more objects, the nearest point of intersection is the visible one. In addition, it needs no explicit clipping operation and is the only algorithm suited to the global illumination models of Kay [6] and Whitted [5].

A structured object space makes hidden surface elimination more efficient. By dividing the object into a small number of subspaces (perhaps ten) the ray traced from the screen can easily be checked for object intersection at a macro level by comparing it with the simple subspaces. Each of these subspaces is an arbitrarily rotated, translated, and scaled rectangular parallelepiped (see Figure 4a). It is described with a four-by-four transformation matrix which transforms the ray in the object space into a ray within this subspace. All that is required to determine intersection is a vector transformation and a comparison against the limits of the subspace boundary. If the ray misses all of the parallelepipeds at the top level, then the corresponding screen pixel is "empty" (as is often the case in simple scenes). If it successfully penetrates a subspace, then the search proceeds at the lower level where all of the sub-subspaces of that subspace are examined in the same manner (see Figure 4b). When the ray enters a parallelepiped that is not further subdivided, then it has reached a solid surface whose characteristics can be displayed.

Coupled with the standard benefits of ray tracing are a new set of features that this hierarchy provides. Logarithmic access time (instead of polynomial time) stands out as the best feature. It typically takes only five or six levels of subdivision to represent complex scenes because each level contains about order of magnitude more detail. Thus, each linear increase in access time caused by an additional level of subdivision yields an exponential increase in resolution at that level. Object spaces are often shallow trees with a high branching factor.

Even with a shallow tree such as this it is possible to conserve storage at lower levels when there are common object space features. Any subspaces in the entire tree that have the same detail, regardless of the environment, can share their descriptions since all environmental information (location, orientation, and scale) is contained in the higher levels of the hierarchy. Most objects can take advantage of this feature because they have some common properties (usually surface details such as tree bark, windows in buildings, etc.) Thus, the hierarchy of subspaces is actually constructed as a graph instead of a tree. No efficiency is lost and much space is saved.

Another advantage of this scheme is its total generality. Any object can be represented since, in the limit, the parallelepipeds can form a point representation. In the following sections we will show how planar polygons can be represented in terms of their bounding volumes, and how bi-parametric curved surfaces can be reduced to a point representation. Thus rectangular parallelepipeds are very simple objects with which to work. Since they provide uniformity of the object description they allow arbitrary magnification of the view to be done with no adjustment to the description or viewing algorithm.

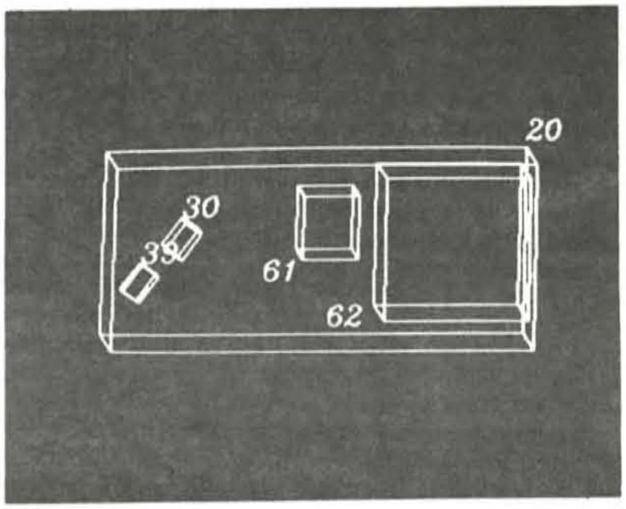


Figure 4a: The top level (box 20) of a hierarchically structured object with 4 subspaces.

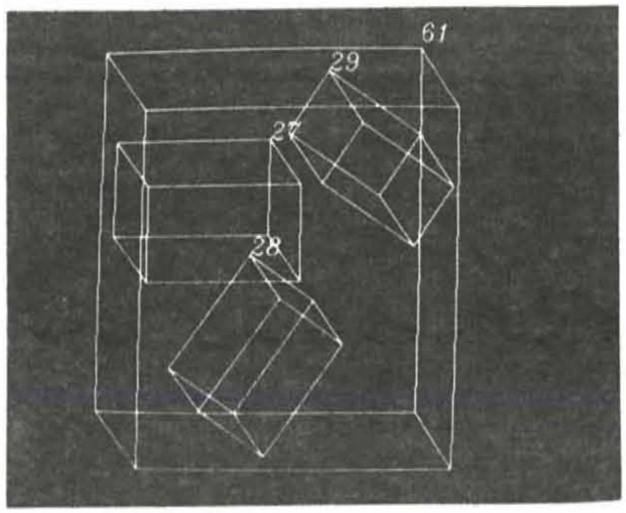


Figure 4b: The second level (box 61) of the hierarchically structured object in Figure 4a.

Input

Creation of hierarchical databases is a non-trivial operation. The problem is that the partitioning at each level requires an understanding of the entire object space. When working with dynamically moving objects, an understanding of the motion is also needed to partition the object correctly. Even static objects must be intelligently divided to make the viewing algorithm effective. This section will discuss a semi-automatic scheme for hierarchical decomposition and will propose some fully-automatic possibilities.

All data for an object starts as a point or surface representation. This information comes from digitizers or algorithms and is rarely aggregated in any hierarchical form. Therefore, the first form of data is a two-level hierarchy where the top level represents the entire object space and the lower level is the complete scene description with thousands of parallelepipeds corresponding to the data points. In order to build a proper hierarchy, we have a structure editor which is able to introduce intermediate level spaces in the object. This editor allows random traversal of the object hierarchy and arbitrary creation, deletion, and transformation of the object components. In addition, the editor can overlay a non-hierarchical description of an object (the typical initial form) with the hierarchy that is being edited to visually assist the human operator. Early experience shows that the editor is easy to use: a hierarchical description of the city of Pittsburgh, with hundreds of terminal nodes in the hierarchy, took only a day or two to enter. The human who assists the editor in the hierarchy creation is looking for object coherence, tightness of fit within the domain of rectangular parallelepipeds, and dynamic motion consistency. In addition, the human must decide when to instruct the editor to combine similar low-level spaces that are to share descriptions.

Automating of the hierarchy creation process, although not currently implemented, could be done by a number of methods. The simplest technique would look for clusters within the object space. It would view the object space as a three-dimensional histogram and select peaks which represent object coherence. There are a number of ways to extract multi-dimensional histogram peaks which could be used [11]. One possibility is to reduce the resolution of the object space and find clusters in that. Lower resolution spaces are easier to deal with and accurately represent the original data [12,13].

For dynamic objects, an initial intermediate level of hierarchy could be built around the known degrees of freedom. The initial object space would then contain a top level which is "the world", a middle level for the moving units, and a low level for the scene detail. The automatic hierarchy routines would then work from there in the same manner.

Automatic combining of common subspaces would be a pre-processing step that finds common detail in the initial data and merges the descriptions. This sort of pattern matching is a difficult problem which can suffer a combinatorial explosion. Reasonable solutions must use some pruning of the alternatives to arrive at an answer.

It can be seen that the creation of a correct hierarchy requires careful consideration. Sloppy hierarchies will cost dearly in the scene rendering stage but good hierarchies are hard to create automatically. In fact, it is possible to understand the benefits of rendering hierarchical scenes in terms of the expense of creating them. Simpler object representations need less work to create an object space but need more work to render a scene. Thus, this scheme derives many of its benefits from being able to off-load all of the combinatorial problems to the model creation stage. For many problems of computer graphics this is a desirable tradeoff because the display must be as fast as possible.

Display

Display consists of two operations: visibility determination and shading. The shader used with this algorithm is described elsewhere [5]. For ray tracing algorithms, visibility determination is essentially a process of intersecting each ray with a set of objects and chosing the nearest point of intersection (Figure 5).

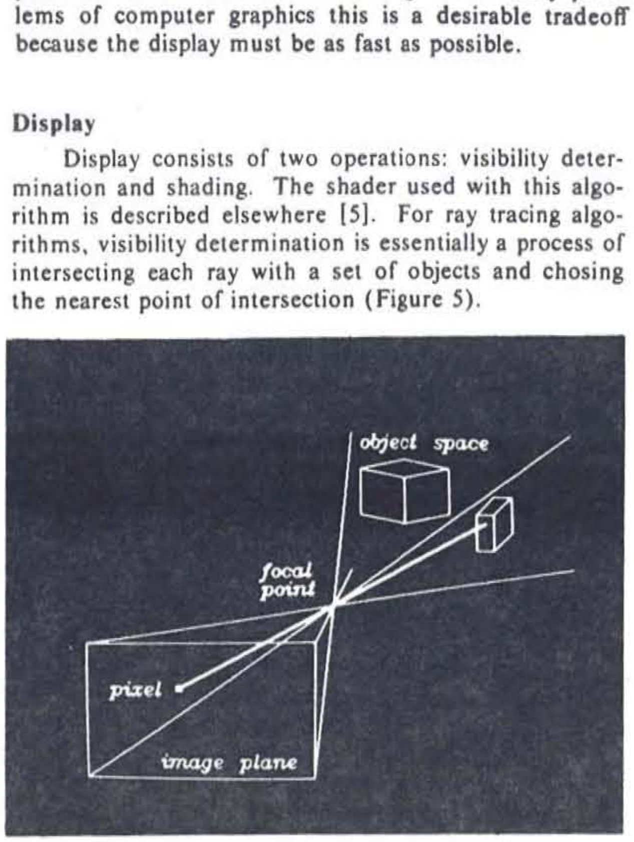


Figure 5: Determining correspondence between a pixel and visible surface by ray tracing.

The structured database improves the efficiency of the operation by minimizing the object set to be tested against each ray. Equally important to the performance of the algorithm is the speed of the intersection mechanism. It is for this reason that rectangular parallelepipeds (RP's) are chosen as the **only** data elements in the object description. Each RP is centered in its own coordinate system with faces defined by

$$x = \pm x_b$$
$$y = \pm y_b$$
$$z = \pm z_b$$

as illustrated in Figure 6. A ray is defined parametrically by

$$x = a_x t + b_x$$
$$y = a_y t + b_y$$
$$z = a_z t + b_z.$$

After a ray is transformed into the coordinate system of the RP, to test a ray against the faces $y = \pm y_b$ one solves the equations

$$t_1 = (y_b - b_y)/a_y$$
$$t_2 = (-y_b - b_y)/a_y$$

The smaller of the two t values defines the nearer point of intersection. Then if

$$-x_b \le a_x t + b_x \le x_b$$

and

$$-z_b \le a_z t + b_z \le z_b$$

the face is pierced by the ray. The tests to determine if the ray pierces the $x = \pm x_b$ and $z = \pm z_b$ faces are identical.

As the ray progresses deeper into the hierarchy, the dimensions of the RP's become progressively smaller until at the terminal level each box is essentially a point in three space. Associated with each terminal level box is a normal vector corresponding to the surface normal of the object which generated the box. The surface normal is unrelated to the orientation of the box and is computed by the data generation procedure instead of the display procedure.

It would be wasteful to represent planar polygons by collections of infinitesimally small bounding boxes. For the special case of rectangles, the terminal level bounding box can represent the polygon exactly if one of its three dimensions is equal to zero and the other two coincide with the dimensions of the rectangle. Arbitrary planar polygons can then be represented by the intersection of rectangles. Figure 7 shows a triangle represented by the intersection of three rectangles. The important feature of this representation is that the display procedure treats all cases the same whether they be described by collections of points or collections of polygons.

Data Management

Simplicity of the display procedure, which is the key to the speed of this algorithm, is gained by transferring much of the processing load to the data generation stage. In some cases the data generation can be performed off line either automatically or through the use of the structure editor. In many cases, however, data generation and display must occur concurrently. One example is the automatic expansion of curved surfaces into a hierarchical point representation. Subdivision algorithms for bi-parametric surfaces [14,15] are natural candidates for the task. We have augmented straightforward subdivision with routines for generating bounding boxes for each subpatch and for maintaining the hierarchy at each step. Unnecessary processing is minimized by subdividing a patch or subpatch only if its bounding box is pierced. When the hierarchy is extended via subdivision, the new branches are retained from one pixel to the next to gain the benefits of object space coherence. In a similar fashion procedurally defined objects can be expanded into a point representation, although we have not implemented such an expansion procedure.

Naturally, unlimited expansion of the object description may fill the memory available to the display process. Bounding boxes that are generated as a result of subdivision or some other procedure are labeled "temporary" when they are created. When the memory

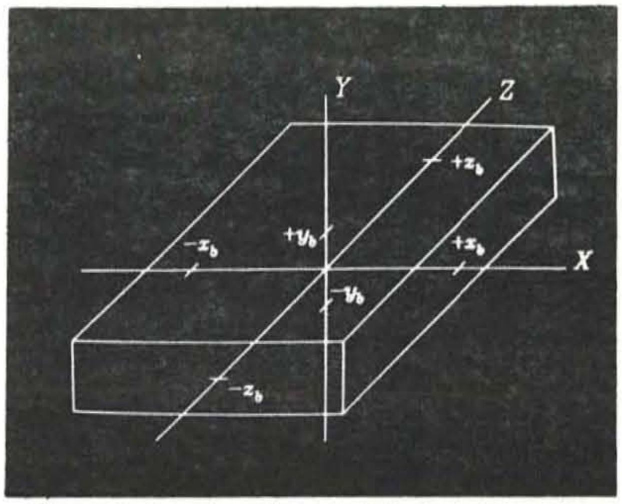


Figure 6: A rectangular parallelepiped defined by $X = \pm X_b$, $Y = \pm Y_b$, and $Z = \pm Z_b$.

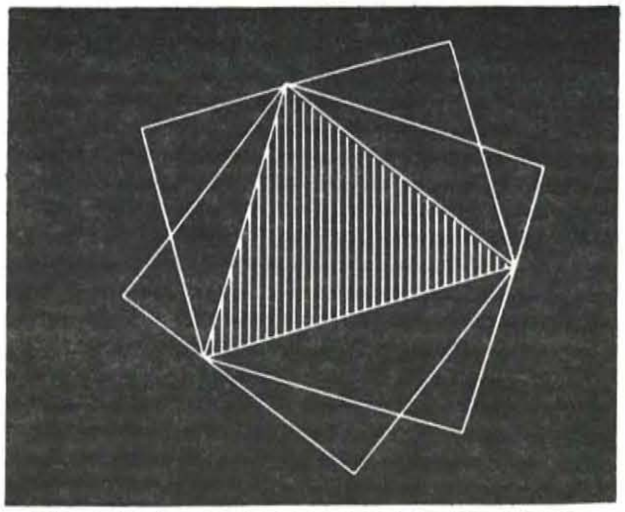


Figure 7: A triangle represented by the intersection of three bounding boxes.

becomes full, all "temporary" elements that were not visited during the previous pixel are deleted.

Implementation

The choice of representation presented here and the attempts to simplify the display process are motivated by our desire to produce a technique that can be easily realized in hardware. However, two initial versions of the display algorithm have been implemented in software.

The original program, running on a PDP-11/40[1], incorporates the structure editor as an adjunct to the display routines. Using the editor, a detailed polygonal description of the city of Pittsburgh containing over 38,000 terminal nodes was created. By using instances of such items as windows on buildings, the number of terminal nodes actually stored is less than 600. This database was used to generate the images in Figures 8

[1] PDP and VAX are trademarks of the Digital Equipment Corporation

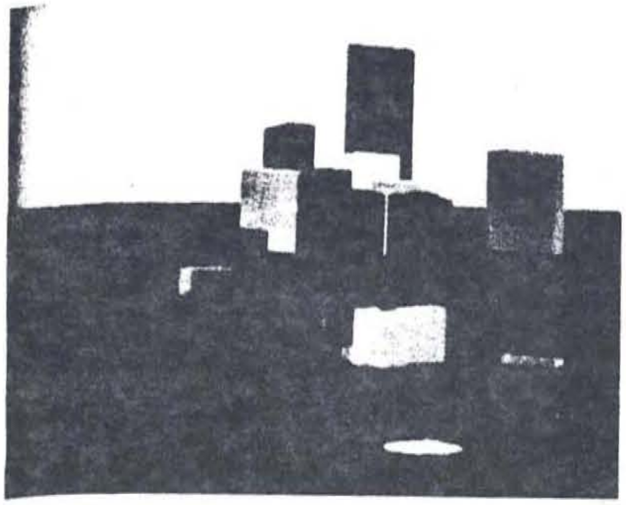


Figure 8: City of Pittsburgh from a distance.

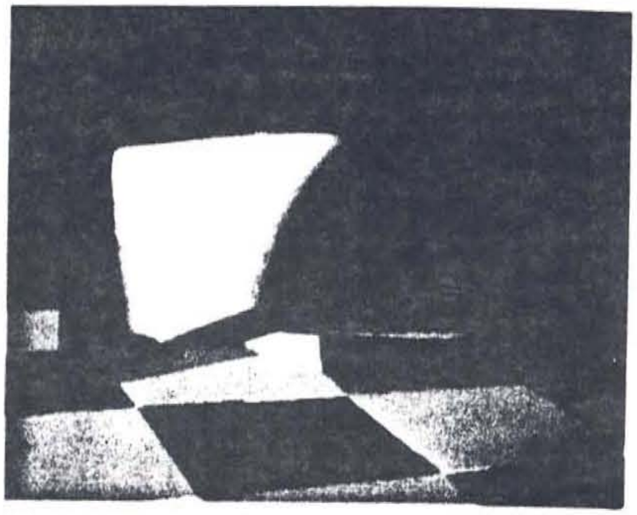


Figure 10: A single Bezier patch.

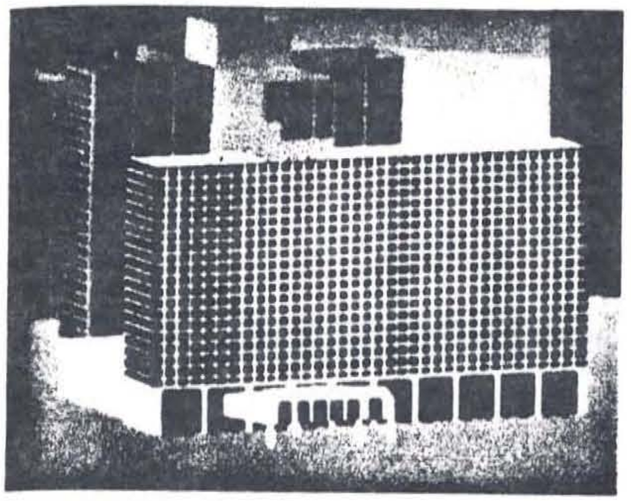


Figure 9: City of Pittsburgh close up.

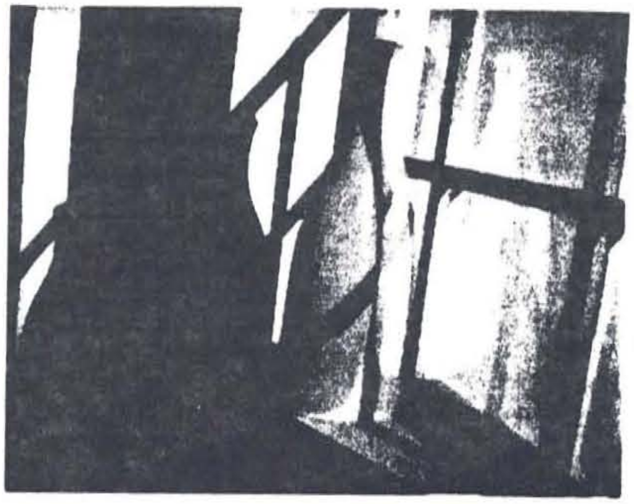


Figure 11: A scene composed of curved and polygonal objects.

and 9 on a VAX-11/780. Figure 8 required less than an hour processor time to generate, but Figure 9 took nearly six hours due to increased detail.

A second version of the display algorithm has been incorporated into a set of VAX programs previously described [5]. Although the programs utilize floating point arithmetic and execute very slowly, they allow us to use a very good shader. In addition, the VAX's one megabyte memory makes it possible to experiment with automatic expansion of a hierarchical object description via patch subdivision. Figures 10 and 11 were produced with this program. The bi-cubic patch shown in Figure 10 would require more than 300 million sample points to represent the surface with the same amount of resolution produced here. Figure 10 required seven hours of computing time to display, and Figure 11, which contains 36 bicubic patches and a collection of assorted polygons, required three hours.

For ray tracing, we have found that the object space coherence feature of the hierarchical representation yields an improvement for bi-cubic surface display of about one hundred to one. As expected, the hierarchical structure produces an execution time proportional to the logarithm of the number of terminal nodes in each scene. For the case of bicubic patches, about 40 percent of the time is used searching the object description tree and another 40 percent is spent subdividing patches to update the object description. Since the display process is oriented to a hardware implementation, we have made no effort to optimize the software. We are just beginning to study ways of extending this representation to display methods other than ray tracing.

Summary

Hierarchical object descriptions provide a substantial performance advantage for the display of complex scenes because the initial stages of the display process consist of a logarithmic search through the object description. We have described a method by which the entire visibility calculation is reduced to this logarithmic search, both simplifying and improving the performance of the display processor. This simplicity is gained through the use of a single form of representation that is sufficiently general to accommodate a wide variety of object types.

References

[1] Clark, J.H. Hierarchical geometric models for visible surface algorithms. **CACM** ,19-10, October 1976, pp. 547-554.

[2] Reddy, D.R. and Rubin, S., Representation of three-dimensional objects, CMU-CS-78-113, Dept of Computer Science, Carnegie-Mellon University, April 1978.

[3] Brooks, J. et al, An extension of the combinatorial geometry technique for modeling vegetation and terrain features, Mathematical Applications Group Inc., NTIS AD-782-883, June 1974.

[4] Csuri, C. et al, Towards an interactive high visual complexity animation system, **SIGGRAPH '79** proceedings, August 1979, pp. 289-299.

[5] Whitted, T., An improved illumination model for shaded display, to appear **CACM**

[6] Kay, Douglas S., Transparency, refraction and ray tracing for computer synthesized images, M.S. Thesis, Cornell University, January 1979.

[7] Sutherland, I., Sproull, R., and Schumacher, R., A characterization of ten hidden surface elimination algorithms. **ACM Computing Surveys** , Vol. 6, No. 1, March 1974, pp. 1-55.

[8] Forrest, A.R., On Coons and other methods for the representation of curved surfaces. **Computer Graphics and Image Processing** , vol 1, 1972.

[9] Riesenfeld, R.F., Applications of b-spline approximation to geometric problem of computer aided design, PhD thesis, Syracuse University, 1972.

[10] Newell, M., The utilization of procedure models in digital image synthesis, PhD Thesis, Computer Science, University of Utah, Salt Lake City, Utah, 1975.

[11] Rosenfeld, A. and Kak, A.C., **Digital Picture Processing** ,Academic Press, New York, 1976.

[12] Kelly, M.D., Visual identification of people by computer, AIM-130, PhD Thesis, Computer Science, Stanford University, Stanford, Ca., July 1970.

[13] Price, K. and Reddy, D.R., Matching segments of images, **IEEE Trans. on Pattern Analysis and Machine Intelligence** ,1,1, 1979, pp 110-116.

[14] Catmull, E., A subdivision algorithm for computer display of curved surfaces, UTEC-CSc-74-133, PhD thesis, Computer Science Dept., Univ. of Utah, Dec. 1974.

[15] Lane, J.M., Carpenter, L.C., Blinn, J.F., and Whitted, T., Scan Line Methods for Displaying Parametrically Defined Surfaces. **CACM** ,23-1, January 1980, pp. 23-34.

An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis

Mark Dippé
Berkeley Computer Graphics Laboratory

John Swensen
Computer Science Division

Department of Electrical Engineering
and Computer Sciences
University of California
Berkeley, California 94720
U.S.A.

Abstract

An algorithm for computing ray traced pictures is presented, which adaptively subdivides scenes into S subregions, each with roughly uniform load. It can yield speedups of $O(S^{2/3})$ over the standard algorithm.

This algorithm can be mapped onto a parallel architecture consisting of a three dimensional array of computers which operate autonomously. The algorithm and architecture are well matched, so that communication overhead is small with respect to the computation, for sufficiently complex scenes. This allows close to linear improvements in performance, even with thousands of computers, in addition to the improvement due to subdivision.

The algorithm and architecture provide mechanisms to gracefully degrade in response to excessive load. The architecture also tolerates failures of computers without errors in the computation.

CR Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors) - *Multiple-instruction-stream, multiple-data-stream processors (MIMD)* I.3.3 [**Computer Graphics**]: Picture/Image Generation - *display algorithms*; I.3.7 [**Computer Graphics**]: Three-dimensional Graphics and Realism - *animation; color, shading, shadowing, and texture; visible line/surface algorithm;*

General Terms: Algorithms

Additional Key Words and Phrases: adaptive, parallel, ray tracing, subdivision

1. Introduction

Realistic three dimensional image synthesis is computationally very expensive. Rather than becoming less expensive, the use of more realistic techniques with highly complex scenes has increased the cost per image.[12] We are interested in efficient realistic rendering of scenes that change over time, using algorithmic and architectural strategies.

The most viable rendering algorithm to date for creating realistic images is ray tracing, because it models the complex effects of light in an environment more effectively than other existing synthesis techniques.

In the ray tracing model, rays are sent from the eye through each pixel of the picture plane and traced as they are reflected and transmitted by objects in space. When a ray hits an object, new rays may be generated, due to reflection, transmission, and/or relevant light sources. These new rays are in turn traced. The ray tracing process thus forms a tree with the eye at the root and rays as the branches. The initial branch is the ray piercing the picture plane. Internal nodes represent objects intersecting the ray, and leaves represent light sources or rays leaving the picture space. The reader is referred to Turner Whitted's excellent introduction[22] for a more detailed description of ray tracing.

Our approach is to adaptively subdivide the ray tracing process, and to implement this subdivision on parallel hardware.

The three dimensional space of a scene to be rendered is divided into several subregions. Initially the space is divided to assign volume more or less uniformly, and object descriptions are loaded into the appropriate subregions. As computational loads are determined, the space is redistributed among the subregions to maintain uniformity of load.

The rendering process begins when the subregion containing the eye or camera casts rays at the desired image resolution. Associated with each ray is its home pixel, so that the pixel can be appropriately colored after the ray tracing operations are complete. When a ray enters a subregion, it is intersected with the object descriptions contained within the subregion. Rays that exit a subregion are passed to the appropriate neighbor.

Each ray resulting from a ray-object interaction contains the fraction of the ray's contribution to its pixel. This fraction is the product of the fractional value of the impinging ray and the value resulting from the object intersection. Color information associated with the spectral properties of the ray/object interaction is also included.

When a ray terminates, becoming a leaf of the ray tracing tree, the rendered value is added to a frame buffer.

Subregion loads are monitored to determine the need for redistributions of space. When a subregion's load becomes too large relative to its neighbors' loads, a change in subregion definition is initiated.

A parallel architecture implementing this algorithm uses a three dimensional array of computers, each with its own independent memory. Each of the computers is assigned one or more subregions. Neighboring computers contain adjacent subregions, and communicate via a variety of messages. Messages not directed toward an immediate neighbor are passed on in the appropriate direction.

Image quality can be traded off with performance, and to this end, the algorithm and architecture provide various means of degrading to achieve a desired rate of image generation.

2. Adaptive Subdivision Algorithm

The synthesis problem is primarily concerned with the visibility of objects with respect to a viewpoint and with the interaction of light in the environment with these visible objects. Visibility is determined by a two dimensional projection of three dimensional space. Lighting interaction is much more complex in that its effect spans three dimensional space in a non-projective manner.

Previous synthesis techniques can be categorized by their generality of lighting model, and by their use of projective qualities of images.[18]

1) projective: z-buffer, painter, Watkins, priority, Warnock, Franklin[8]

These algorithms render and determine surface visibility primarily in image space, using projective transformations. They effectively model those aspects of the scene that are naturally projective with respect to the viewpoint. However, phenomena that are not directly projective with respect to viewpoint, such as shadows or inter-object reflections, introduce many complications.

2) quasi-projective: shadow polygons,[5] cluster planes, three dimensional cookie cutter[1]

These algorithms operate to a greater degree in three dimensional space. They do this by adding information that is non-projective, such as shadow polygons, and/or by attempting to sort three dimensional space, either by separating planes or by the faces of polygonal objects. However, complexity is increased when shadow polygons are incorporated in the rendering process. In addition, objects are often split, because three dimensional space cannot be easily sorted on the basis of visibility. These algorithms generally prepare the information for an efficient projective solution of the visibility problem.

3) non-projective: ray tracing, hierarchical bounding volumes,[17] wave based algorithms[14]

Algorithms in this group perform image synthesis in three dimensions. Modeling of a general class of lighting effects is facilitated. Hierarchical bounding volumes can be thought of as a modeling operation rather than a rendering one, but it is intimately related to rendering. It is a type of three dimensional subdivision which does not sort but uses containment information to aid in visibility determination. Ray tracing is the primary example of algorithms that inherently operate in three dimensional space, i.e. no projection with respect to viewpoint is necessary. The main disadvantage is that, in general, all of three dimensional space must be considered to arrive at a solution.

The complexity of ray tracing is associated with the testing of rays for intersection with the objects of the scene. The distribution of complexity in space is determined by the distribution of objects, and by the distribution or flow of rays among the objects. A region of space with many objects but with no rays has low complexity, as does a region with many rays but no objects. On the other hand, a region in which many rays are interacting with many objects has very high complexity.

Up to now, most algorithms have subdivided the two dimensional projection of three dimensional space when rendering. Our algorithm is completely non-projective in nature, and subdivides three dimensional space itself. The essential characteristics of the algorithm are:

1) Three dimensional space is divided into several subregions. Object and light source descriptions are distributed among the subregions according to their position. Each of the subregions is processed independently.

2) Rays are cast into three dimensional space and processed in the subregions along their paths. The rays within a particular subregion are tested for intersection with only those objects within that subregion. Rays that exit the subregion are passed to neighboring subregions. The rays are processed until they terminate and become leaves of the ray tracing tree.

3) The shapes of the subregions are adaptively controlled to maintain a roughly uniform distribution of load.

For any given ray, we only consider subregions along the path of the ray, and ignore all others. Thus, the problem is reduced from considering all objects, to considering only those objects along the one dimensional ray.

The three dimensional method for subdividing the ray tracing problem can be applied to the general image synthesis problem. The parameters can be thought of as:

1) objects, and

2) distribution of light through space.

The problem is to find the visual stimulus from such a world.

3.1. Subregions

There are several issues concerning the shape of the subregions that subdivide space:

1) the complexity of subdividing the problem, e.g. intersecting objects or rays with the boundaries,

2) the ability to subdivide space without splitting objects, and

3) the uniformity of the distributed loads attainable with the shape.

The complexity of scenes is certainly not uniform, but varies according to the characteristics of the components of the scene. Our subregions do not divide space uniformly, but allow arbitrary subdivisions of space within the topological and geometrical constraints of the subregions. This provides us with a very powerful technique with which to subdivide the space to accommodate non-uniform complexity. The ability to dedicate processing power where complexity is concentrated and not waste it where it is unneeded is one of the fundamental aspects of the system.

Among the polyhedral shapes which could bound the subregions, the three most promising candidates are orthogonal parallelepipeds, "general cubes", and tetrahedra. More general shapes such as quadric surfaces are under investigation and may be useful in the future, but are not considered in this paper.

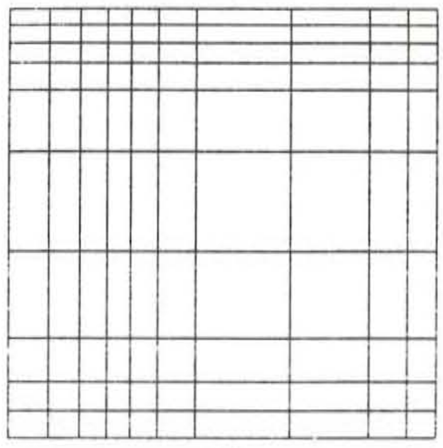
The most intuitively simple polyhedra are orthogonal parallelepipeds, which are constrained to have all boundaries parallel or perpendicular to the major axes. Figure 1a shows a two dimensional analog of orthogonal parallelepipeds. As subregions grow and shrink to redistribute the complexity of the scene, the boundaries remain orthogonal, and the subregions remain convex.

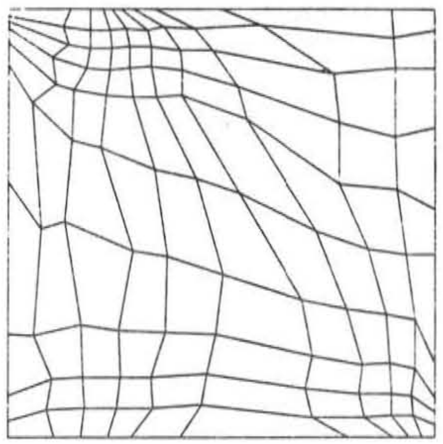
Boundary-intersection testing for orthogonal parallelepipeds is not a significant overhead. However, the orthogonality constraint does not allow local adjustments to a subregion to be made without affecting many other subregions, and in general, a scene's computational complexity will be less uniformly distributed than with more general boundaries. Unless either very few basic objects are contained in each subregion, or the scene has a uniform distribution of complexity over space, the low overhead of orthogonal parallelepipeds is unlikely to offset their greater non-uniformity of load.

General cubes resemble the familiar cube, except they have relaxed constraints on planarity of faces and on convexity. 2-D analogs of general cubes are shown in figure 1b.

With these relaxed constraints, the complexity of boundary testing is increased over that for the orthogonal polyhedra, and hence the overhead for each subregion is increased. However, general cubes allow much more local control of subregion shape, with the consequence that more uniform distributions of load can be achieved than with orthogonal subregions. Furthermore, the redistributions can be performed locally.

Tetrahedra are the simplest shapes, and are inherently convex. A space-filling collection of tetrahedra can be constructed with groups of six tetrahedra forming a cube, which are then arranged to fill space. The boundary of each subregion is defined by its four corners, and the interface with two neighboring subregions is defined by three of these corners. Figure 1c shows an analog of


(a)


(b)

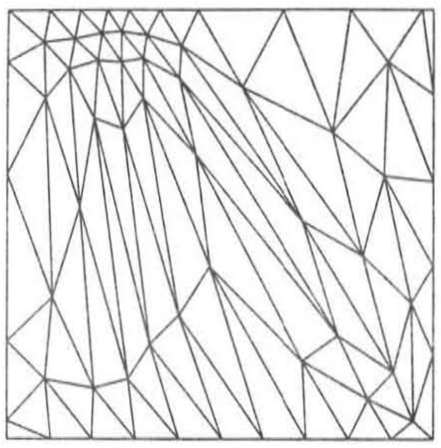

(c)

Figure 1. 2-D analogs of orthogonal parallelepipeds, general cubes, and tetrahedra, simulated with the same load.

tetrahedral subregions.

Tetrahedra have fewer boundaries than cubes, so the overhead of boundary testing is lower for them than for general cubes. They allow local control of subregion shape, but because vertices are shared among more subregions with tetrahedra than with general cubes, the control cannot be as local. Because tetrahedra have fewer vertices than cubes, they may not contain subregion-sized objects as well.

Simple, fixed connectivities have been assumed throughout the discussion. We have also considered arbitrary topologies for the subregions. However, fixed connectivities allow simpler calculations for the determination of how rays move among subregions, and the tradeoffs between more general topologies and shapes have yet to be fully determined. For these reasons, we only discuss arrays of general cubes for the remainder of the paper.

Given the basic scenario of how the subdivision algorithm operates and how space is subdivided, how do we carry out an actual subdivision to yield a uniform distribution of the problem? A direct optimal solution requires global knowledge, and is quite difficult. We would like to avoid these problems if possible.

Our solution is to allow neighboring subregions to share each others' load information, and to allow relatively more loaded subregions to adjust their boundaries to reduce load. This mechanism is a feedback scheme. It also provides a mechanism for adapting to the changing complexity of a scene in a distributed manner.

2.2. Adaptive Redistribution

It is difficult to calculate the distribution of load without actually simulating the ray tracing process. The algorithm redistributes the load among the subregions to adapt to changing conditions induced by the movement of objects, lights, the eye or camera, and other time varying behavior.

Load is redistributed by moving the points defining subregions and passing the object information as needed. The load of each subregion is compared to that of its neighbors. When a subregion's load is higher than its neighbors, some load should be transferred to them. More general relative load measures can also be used.

The load metric is determined primarily by the product of

1) number of objects and their complexity, and

2) number of rays.

Load is transferred by moving corners of a subregion. To simplify matters we only move one corner at a time. The corner's new position is chosen such that enough volume is transferred to equalize loads.

Once the new position for a corner of a subregion has been determined, object descriptions and other information are redistributed to reflect the new subdivision.

If the scene being rendered is too complex, then redistribution cannot entirely alleviate the problem. In such a case, degradation techniques must be applied. These techniques are discussed later.

This is a simplified scenario of the redistribution process, some of the more difficult points of which we discuss in the following section.

2.2.1. More Sophisticated Approaches

Subregions with elongated shapes, or those which are very concave, may cause rays to pass through more subregions than would be necessary with fatter, convex subregions. The load metric reflects the undesirability of elongated or concave subregions. When a subregion becomes too undesirable in shape, its load can be increased by a factor indicating its desire to become shapely. This more general framework will allow subregions to become unshapely when it is advantageous, while maintaining shapely subregions in general.

When selecting a corner of a subregion to move, we must take into account the difficulty involved in shifting the load; it may be easier to transfer load from one corner than another. We also wish to transfer the load differently to each of the affected neighbors, with more of the load going to those that are least loaded. Another important constraint on redistribution is to minimize the splitting of objects among neighboring subregions. This can be done relatively easily, because the overloaded subregion which is exporting objects can choose the new position of a corner to avoid splitting. In addition, by subdividing the subregions into smaller regions, within which statistics are kept about object distribution and ray flow, more precise decisions about object splitting and load movement can be made.

Our mechanisms for redistribution will not necessarily produce exactly uniform load distributions; this is tolerable, as long as the differences in load are small percentages of the average.

We would like to avoid oscillations in the redistribution process. Small oscillations can be damped by adding hysteresis so that load disparities of a certain size are required before a redistribution is allowed. Instability due to the shuffling of large objects across boundaries can also be detected and limited.

Global information about the distribution of loads is maintained, and is used to direct effective redistribution. When loads are highly disparate, large transfers of load are used. As the disparity decreases, smaller loads are transferred.

There are many subtle issues involved with the redistribution process, and further analyses and experiments must be performed to determine the best choices for this application. We hope to complete these studies in the near future.

3. Parallel Architecture

Our parallel implementation of the algorithm uses independent computers, each communicating with a few neighbors. Computers are responsible for one or more subregions and communicate with neighboring computers using messages. To simplify the discussion, we assume one subregion per computer.

Computers handle all rays and redistributions affecting their subregions. They have several other tasks as well. Messages may be sent to non-neighboring computers by passing them through intervening computers. Thus, computers must route messages.

Computers at the boundary of the array handle infinite extents of space. They also have fewer neighbors than those in the center, and so they are logical candidates to manage auxiliary storage devices and network interfaces. Computers in the center would access these devices via requests through messages. Because we do not want a direct connection to a frame buffer for each computer, the frame buffer will be connected to boundary computers and accessed via messages.

Besides the special role of disk and frame buffer connections for boundary computers, there are other special roles that certain other computers have:

1) The computer containing the eye or camera must cast the initial rays for each frame.

2) Interface and monitoring tasks, assigned to some computers, are used for dealing with user controlled system parameters as well as any other global tasks, such as initially distributing the image description or watching the system load as a whole and changing certain parameters automatically in response to load changes.

The parallelization of the image synthesis problem is based on subdivision of three dimensional space into adjacent polyhedral regions. The computers responsible for the subregions operate independently, adaptively redistributing the space as loads are determined, and gracefully degrading if their load is too large.

3.1. Architectural Perspective

A number of special purpose graphics engines and systems have been proposed and/or built. We briefly describe some of this work in the context of our architecture.

3.1.1. Multicomputers

The LINKS-1[15] multicomputer has been built to generate ray traced pictures. It consists of 64 unit computers, each of which is connected to two neighbor computers, a root computer, and a result collection computer. The root computer controls the system and facilitates non-neighbor communication among computers.

This topology allows work to be distributed by the root computer so that it can be performed independently in parallel, or pipelined from neighbor to neighbor, or some combination of both. Unfortunately, if scene descriptions are too complex to be duplicated in each computer's memory, then substantial communication among the computers is required, but this is hindered by the restricted connection topology. Furthermore, expansion of the system will be limited by the use of the global root and collection computers.

Recent work by other researchers[4,20] has also addressed the application of parallelism to ray tracing. They consider geometrically uniform, orthogonal subdivisions of space. Both efforts favor two dimensional arrays of computers over three dimensional arrays.

However, they do not address the issues of achieving uniform load distribution over the subregions. The ability to adaptively redistribute over time is crucial to the success of this approach, not only because of temporal changes in the scene, but because load distributions are extremely difficult to calculate without actually simulating the ray tracing process.

3.1.2. Graphics Engines and Sub-processors

Clark uses twelve of his specialized VLSI processors, Geometry Engines,[11] to perform the floating point calculations necessary for geometric calculations prior to the rendering process. These types of operations are also useful for ray tracing operations, and similar hardware should be eventually be included in our system. However, we feel that because of the experimental nature of our system, hardware complexity should initially be applied to support general purpose processing, at the expense of special purpose operations.

Fiume and Fournier[7] describe a multiprocessor architecture which processes spans of scanlines using parallel processors, allowing some degree of anti-aliasing to be performed in each processor. The Pixel Planes system developed by Fuchs[9] uses a 1-bit processor per small number of pixels to perform scan-conversion and hidden-surface elimination on a per-polygon basis, using a z-buffer technique. In Pixel Planes, polygons are broadcast to the array of processors, one at a time, and are processed in parallel. The applicable algorithms for both of these architectures are by nature restricted to projected image space, and are not appropriate for our application.

A number of processor per polygon architectures have been proposed.[10,21] These determine surface visibility at the pixel level using a number of depth comparators. As these have all worked in projected image space, they are also not appropriate for our approach.

3.2. The Nature of the Parallelization

A ray crossing the three-dimensional array of computers can pass over many computers, so global information about the system may be old. In particular, knowledge that all computers have completed their ray tracing operations may be out of date.

Rather than wait for messages to propagate, we adjust the computation so that all computers complete at approximately the same time. This has the consequence that new frames might be started before older frames have finished. This is not a problem, because minor variations can be absorbed by allowing computations on old frames to continue after newer frames begin. Ray tracing and frame buffer updating can overlap if several image frames are stored. In this way, late updates to frames can be made before displaying them.

3.3. Parallel Redistribution

Messages are used to initiate a redistribution. Each computer has 26 neighboring computers, even though it is only directly connected to 6, and each corner is shared among 8 computers. Thus, the load and redistribution messages must be routed through neighbors to allow complete determination of redistribution parameters for each computer (see figure 2). Load and redistribution messages will contain routing information to speed up the process.

In addition to object information and ray flow within a computer, other factors are related to load in the parallel architecture:

1) the number of messages dealt with,

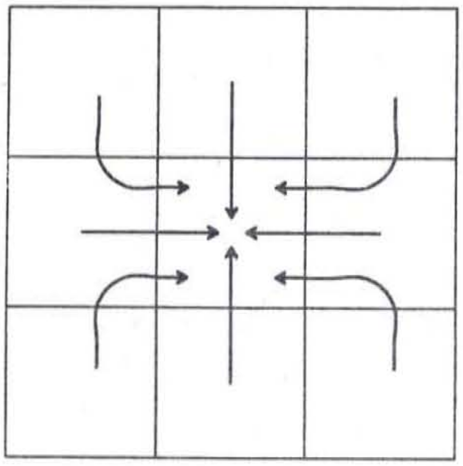


Figure 2. Load message routing.

2) the size of message queues (this measures how far behind a computer is, relative to its neighbors), and

3) idle time.

There other possible factors which are under investigation.

Once the computer has determined the new position for a corner, it sends out redistribution messages and begins passing objects and other information. To handle ties between two computers which decide to move the same corner at the same time, each computer is assigned a priority according to its position in the array.

If an overloaded computer's neighbors are also overloaded, so that no load transfers can take place, the computer must reduce its load using the degradation techniques discussed later.

To ensure that redistributions are carried out as quickly as possible, computers check their message queues periodically. If a redistribution message is found, it is acted upon immediately.

Note that the redistribution process is carried out in a local manner without a complex protocol.

3.4. Messages

We list some of the different messages and their fields:

1) ray: pixel, point of origin, direction, percentage of contribution to the pixel of this ray, color parameters of the ray

2) pixel: address, value

3) global parameter: name, value; for instance there is a message that tells the computer containing the eye or camera whether or not to send out the initial rays and perform rendering

4) object information: description; general objects such as splines,[2] fractals,[13] or blobby models[3] will be handled

5) disk i/o: request type, associated data

6) load value: computer ID, value, and possibly volume information for redistribution purposes; these messages are copied to a monitoring computer for global load statistics tracking

7) redistribution notification: corner to move, new location, routing information

By monitoring the messages being passed as well as examining load messages, the performance of the system can be easily measured. Thus, messages form a direct basis for evaluating the system.

4. Performance Degradation

If our algorithm/architecture is to be successful in an interactive environment, it must allow performance to degrade gracefully. Since any machine constructed must be finite, some problems will exceed its resources, and degradation will be unavoidable. If a computer is unable to reduce its load by moving boundaries, it can decrease its load by more drastic means.

One possible load-reducing action is to delete ray messages in a controlled manner, until a computer can keep up with the desired frame rate. This amounts to a local reduction in the height of the ray tracing tree. Messages significantly older than the youngest received message would be logical candidates for deletion. Some messages, such as disk requests, should not be deleted.

At a more global level, the image resolution or the depth of the ray tracing tree can be reduced, creating fewer rays, at the expense of less realistic simulation. In addition, the level of detail for rendering of objects can be reduced; for example, splines or fractals might be subdivided into fewer polygons.

Generally, the small errors in pixel intensity these degradation measures introduce will be visually tolerable. If they are not acceptable, the total workload must be decreased by reducing the rate of image generation.

By allowing a user to trade image quality for speed, an extremely effective tool for practical image synthesis is provided.

These adaptivity parameters may be controlled by the user, or by a computer which has been gathering statistics. They are used to tune the performance of the system.

5. Analysis

Symbol	Meaning
B	number of objects
C	number of parallel computers
D	depth of ray tracing tree
L	number of lights
N	number of rays from an object interaction
R	number of rays
S	number of subregions
β	boundary-intersection cost
κ	co-resident object overhead
ν	non-uniform distribution cost
π	ray passing overhead
σ	pixel energy summation cost
τ	subregion traversal overhead

We analyze the standard ray tracing algorithm, the adaptive subdivision algorithm, and the parallel architecture. Our coupled parallel implementation is also compared with a frame-parallel implementation.

The cycle counts for data transfers and floating point operations assume the computers are implemented using microprocessors augmented with floating point chips. Computers with faster floating point speeds will also tend to have wider data paths, so the relative speeds of arithmetic operations and data transfers will be similar.

5.1. Standard Algorithm

To perform ray tracing, rays are cast into a scene, and are tested for intersection with each object in the scene. As a result of an intersection, each ray may produce 0, 1, or more offspring, depending on the reflective and refractive properties of the objects, as well as the number of light sources. In a highly reflective environment, several levels of interaction testing - new ray generation must be performed for a realistic rendering.

We assume the ray tracing tree generated by recursive reflection/refraction rays is a complete tree of depth D. It has R_i nodes at each level, where $R_{i+1} = R_i N$ and N corresponds to the average number of rays produced by a ray-object interaction. The total number of arcs in this tree is given by

$$R_0 \sum_{i=0}^{D-1} N^i = R_0 (\frac{N^D - 1}{N-1}),$$

where R_0 is the number of pixels, typically 512^2 to 1024^2. In addition to recursive rays, rays to light sources are cast at each node, so the total number of rays is

$$R \approx R_0 (\frac{N^D - 1}{N-1})(1 + L),$$

where L is the number of light sources. We assume 10% of the intersections transmit rays, as well as reflect them. In addition, we consider two light sources and 500×500 pixels, and a ray tracing tree of depth 5 (a depth of 4 has been used to produce reasonable pictures). Thus, $N = 1.1$, $L = 2$, $R_0 = 250000$, and $D = 5$, so that the total number of rays is

$$R = 2.5 \times 10^5 \frac{(1.1^5 - 1)}{(1.1 - 1)}(1 + 2) \approx 4.6 \times 10^6.$$

The number of ray tracing operations for the standard algorithm is RB, for B objects, and if $B = 1000$, 4.6×10^9 ray tracing operations must be performed.

The cost of each ray tracing operation depends on the complexity of the objects; while a sphere is almost trivial to intersect with rays, a fractal object can be extremely costly. We assume a basic object which has roughly the same complexity as 250 triangles represented using a Rubin-Whitted[17] style model. This is a reasonable unit of object complexity, but is still simpler than many object models used today. When a ray is tested against one of our basic objects, we assume it will almost always be rejected after one bounding volume intersection test, at a cost of the equivalent of 25 floating point multiplications, which each require 100 cycles to execute. Thus a basic object requires about 2500 machine cycles to test for intersection with a ray, on the average.

5.2. Subdivision Algorithm

Testing each ray against each object can be avoided by adaptively subdividing the scene into a number of subregions, each with approximately the same load, and checking only the objects within the subregions along the ray's path. The algorithm for computing the image performs the ray tracing operations within the restricted domain of the subregions.

Subdividing space distributes load so that it is roughly uniform, but has accompanying overhead. We discuss the sources of the overhead before continuing with the analysis.

Breaking space up into subregions forces rays to pass through several subregions in order to cross the entire space; this traversal cost is τ. If the subregions are well-shaped, τ is bounded by $3S^{1/3}$. Each ray may be processed up to τ times, but this has no effect on the total number of rays.

Each ray must be tested for intersection not only with each object in the subregion, but with the boundaries as well, because a ray may intersect an object at a point outside the subregion, even though the object is partially inside the subregion. This adds a cost of β for each subregion, which is about the same as testing 4 basic objects.

Although our redistribution algorithm tries to avoid splitting objects across subregion boundaries, some splitting will be unavoidable, so some testing will be duplicated. If κ is the co-residency cost, this is as if there were κ times as many non-co-resident basic objects. For a scene consisting of a single sphere, κ could be S, but this type of scene is inappropriate for our algorithm. If instead we assume complex objects, such as spline surfaces, the amount of testing duplication may be small. Such object models are usually expanded into more primitive representations (e.g. triangles), and these expanded representations are distributed among the subregions. Determination of realistic values for κ require further investigation.

The number of ray tracing operations required to compute an image using the improved algorithm is the sum, over all subregions, of the traversal cost for each ray, times the number of rays in the subregion, times the sum of the objects in the subregions (including co-residency) and the boundary testing cost:

$$\sum_S \tau R_S (\kappa B_S + \beta).$$

If the distribution of load is uniform over all the subregions, then $R_S B_S$ is the same for all subregions. Using the arithmetic-geometric mean inequality,[16] it can be shown that

$$R_S B_S \leq RB/S^2.$$

Therefore, the number of ray tracing operations is bounded by

$$\sum_S \frac{\tau R(\kappa B + \beta S)}{S^2} = \frac{\tau R(\kappa B + \beta S)}{S}.$$

Assuming β is 4, τ is $S^{1/3}$, $S = 125$, and ignoring κ, we have

$$\frac{5 \times 4.6 \times 10^6 (10^3 + 4 \times 125)}{125} \approx 2.8 \times 10^8.$$

This is an order of magnitude improvement over the standard algorithm. As long as κ is less than 10, this algorithm is faster than the standard algorithm. Note that B must be greater than S if the speedup is to be large; otherwise the overhead of boundary testing will become significant.

Assuming complex scenes, boundary testing overhead is small relative to object intersection calculations. Under these conditions, the cost reduces to

$$\frac{\kappa\tau RB}{S} \approx \frac{\kappa S^{1/3}RB}{S} = \kappa RBS^{-2/3}.$$

Thus we obtain an $O(S^{2/3})$ speedup over the standard algorithm.

5.3. Parallel Architecture

In a parallel implementation of the algorithm, each computer is responsible for one or more subregions. We consider the limiting case of one subregion per computer.

Each ray may in the worst case be passed across a subregion boundary after each iteration. The cost, π, of passing a ray is less than 1/5 the cost of one basic object intersection, assuming a 100-byte ray message can be copied in 400 cycles or less.

When ray tracing is carried out in parallel, the pixel energies are distributed among all the computers, and must be collected before they can be sent to a frame buffer. In the worst case, one half of all rays will contribute to pixel intensities, and if the ten-byte energy messages are sent across at most $C^{1/3}$ computers, at a cost of 50 cycles per message per computer, the energy summation cost, σ, is $25RC^{1/3}$ cycles for all computers, or roughly $25RC^{-5/3}$ for each computer. The ratio of σ to the ray tracing cost per computer is less than 1%, and therefore σ can be ignored.

Unlike σ, the ray passing overhead must be included in the cost of a parallel implementation. The worst case number of ray tracing operations performed by each computer is bounded by

$$\max_C\{\tau R_C(\kappa B_C+\beta+\pi)\}.$$

If a uniform distribution of load can be achieved, then the worst case number of ray tracing operations per computer is

$$\frac{\tau R(\kappa B+\beta C+\pi C)}{C^2}.$$

When we substitute the same values as before (again ignoring κ), with $C = 125$ and $\pi = 1/5$, we get

$$\frac{5\times4.6\times10^5(10^3+4\times125+.2\times125)}{125^2} \approx 2.3\times10^6.$$

This is three orders of magnitude faster than the standard algorithm, due to both the $C^{2/3}$ factor from the subdivision, and the linear speedup from the C computers.

In general, loads will not be completely uniform. The cost, ν, of this non-uniformity is proportional to the ratio of the maximum load for any computer to the average load over all computers. This changes the number of ray tracing operations to

$$\frac{\nu\tau R(\kappa B+\beta C+\pi C)}{C^2}.$$

Accurate estimates of ν will require more extensive analysis, although preliminary studies indicate that values below 2 can be achieved for some scenes.

We have demonstrated speedups of $S^{2/3}$ due to subdivision, and C due to parallelism, with some loss due to ν, κ, β, and π. These improvements will increase with scene complexity.

5.4. Comparison of Coupled and Frame-Parallel Implementations

In justifying a specialized architecture, we must show that the performance of the coupled parallel architecture will exceed that of an equivalent number of independent computers assigned to the problem. In particular, when producing films of computer-generated images, an obvious exploitation of parallelism is to assign to each computer the task of generating a single frame, and let them compute independently, using a serial implementation of our new algorithm, or the standard algorithm. We compare this alternate parallel strategy to our parallelization.

5.4.1. Storage Requirements

The new algorithm implemented on the parallel architecture requires considerably more storage than a standard implementation. With the frame-parallel implementation, each pixel's tree of rays may be traced separately, and by traversing each tree of rays in depth-first order, very few rays need to be maintained in storage. With our parallel implementation, the entire ray tracing tree is traced concurrently, and is traversed in breadth-first order.

All objects must generally be maintained in local storage for a frame-parallel implementation, as each ray must be tested against all objects. When the new algorithm is implemented on our coupled parallel architecture, all objects must be held in some local memory, but they are distributed among all computers; each computer has on the order of B/C objects in its memory, on the average.

Each computer of the coupled parallel architecture need only be configured with on the order of $1/C$ times the data memory of a single computer, with consequent savings in addressing and decoding hardware, and memory management tables. However, if frame-parallelism is used, the C independent computers would each require separate image stores and access to a full memory complement, so that C times as much total memory could be required. Were a memory hierarchy using disk backing store used, the additional cost of many disk units, as well as the time penalty of remote access would be incurred.

Therefore, the coupled parallel architecture can use less total storage than as many frame-parallel computers running serial algorithms.

5.4.2. Storage Structures

For independent computers, it is assumed that memory is implemented in the standard hierarchy of paging disk / main memory / (possibly) cache memory. As always, performance will be severely degraded if the program's working sets do not fit in the appropriate memory.

A dense interconnection of the coupled parallel architecture does not allow convenient communication between computers and disk memory. Assuming only computers at the boundaries of the architecture are connected to backing-store devices, most disk accesses must pass through several other computers. This of course increases the cost of paging, and in order for coupled parallel computers to achieve memory performance similar to that of the independent parallel computers, a relatively higher ratio of local memory to problem size is necessary; sufficient local memory is crucial to this architecture.

6. System Considerations

An initial implementation of a parallel system would consist of eight computers, each managing one or more subregions. With so few computers, each would communicate with only three others, instead of six, as discussed earlier. One computer would communicate with a host computer, which would provide interactive control and access to disk storage. Another computer would communicate either with a frame buffer, or with a host connected to the frame buffer.

Each computer would consist of a commercial microprocessor with floating point support, 1/4 to 1/2 megabytes of RAM, and six unidirectional byte-parallel ports (possibly with DMA access to memory). Those computers communicating with a host or frame buffer would have additional ports. Each computer would run with its own clock, independent of the others.

After performing studies with the initial implementation to verify costs of communication, redistribution, etc., a larger system could be constructed. With larger systems, each computer would communicate with six neighbors or peripheral devices, and dedicated frame buffers and disks would be used.

An obvious physical topology for such a system would be an array of cubical modules, appropriately interconnected. However, this topology would not allow convenient access to central computers for debugging during operation.

An alternate topology resembles the layout of the CDC 6600,[19] with a number of panels fanning out from a central core. Each panel would house several subregion computers and their cooling. Inter-panel communication would be routed through the core, and because the topology is relatively compact, inter-computer communication times should be small. If panels were attached to the core with hinges, they could be spread apart to allow a technician to access all computers during operation.

As an alternative to many medium-speed computers, a few very fast computers could be used, such as the multiprocessor Cray X/MP. A two-processor version of the algorithm would have multiple subregions managed by each processor. Inter-processor communication would make use of the high speed inter-CPU data paths available on the Cray X/MP. The extreme speed of the CPUs would allow studies to be performed in a reasonable period of time, including efficient simulations of different topologies.

6.1. Hardware Issues

The parallel architecture for this algorithm allows high bandwidth, inexpensive communication among the computers, tolerance of computer failures, and great flexibility in the choice of CPU.

Messages are sent between neighboring computers via dedicated links, which allows each communication to be independent of all other computers. Because links are not shared, all communication can run in parallel, and very simple hardware and protocols can be used for them.

This lack of sharing also allows failed or uncooperating computers to be ignored by other computers, so failures are localized at the computer, and do not propagate into the system. Furthermore, there are simple extensions to the adaptive subdivision algorithm which allow subregions lost to failed computers to be adopted by healthy neighbors.

The algorithms executed in each computer are general purpose, and require no special-purpose hardware. Consequently, any processor with a large address space and support for floating point computations can be used. This would allow development of a prototype using commercial microprocessors, while leaving the option of later upgrading to more sophisticated microprocessors, other CPUs, or special purpose ray tracing hardware, as appropriate, with virtually no impact on software.

7. Conclusions

We have presented an adaptive subdivision algorithm and a parallel architecture for the image synthesis problem. The algorithm provides a roughly uniform distribution of load among the computers. Degradation in response to overloading is also part of the system, ensuring that cost/quality tradeoffs in image generation can easily be made. The architecture has a fixed polyhedral connectivity, with communication between computers via messages. Failures of computers are tolerated by the system without loss of accuracy, and without severe degradation of performance.

The algorithm can yield performance improvements on the order of $S^{2/3}$, and the parallelization itself will provide linear gains in the number of computers used (i.e. C computers can reduce actual compute time by a factor of C). Since messages are directly correlated with load,

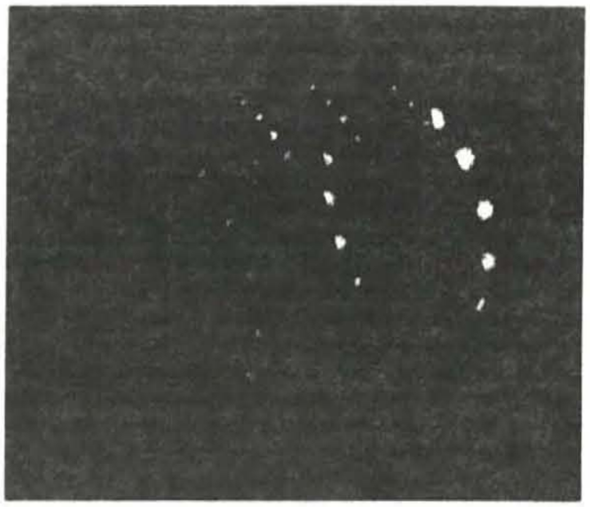


Figure 3. Image generated by a preliminary simulator.

simulation and performance analysis of the system is simplified. We are carrying out such simulations at the current time (figure 3).

The adaptive nature of the system is a very important property, and we are examining other techniques for adaptive control of the synthesis problem. In addition, the synchronic nature of the parallelization may provide additional improvements.

It is interesting to note that a two dimensional analog of our algorithm/architecture can be applied to the projective style of image synthesis. While subdivision of projective solutions has been studied to a great extent, the adaptability of our algorithm and the high degree of parallelism in the architecture will provide new performance gains.

There are many issues left to be resolved, but some, such as antialiasing of the subdivision algorithm,[6] have already been addressed. The algorithm is also applicable to the general image synthesis problem, and we are investigating new methods for increased realism within this framework.

Acknowledgments

We would like to thank the anonymous referees, as well as Brain Barsky, Rick Speer, Steve Upstill, Helena Winkler, and Princess Minnie for their careful readings and helpful criticisms and comments on this paper. A special thanks to E.B. for the *Lemma in E♭ minor*, and a host of others, for wagging the appropriate tails.

This work was supported in part by the National Science Foundation, under grant number ECS-8204381, the Semiconductor Research Corporation under grant number 82-11-008, and Lawrence Livermore National Laboratories under grant number LLL-4695505.

References

1. Peter R. Atherton, Kevin J. Weiler, and Donald P. Greenberg, "Polygon Shadow Generation," pp. 275-281 in *SIGGRAPH '78 Conference Proceedings*, ACM,(August, 1978).

2. Richard H. Bartels, John C. Beatty, and Brian A. Barsky, *An Introduction to the Use of Splines in Computer Graphics*, Technical Report No. UCB/CSD 83/136, Computer Science Division, Electrical Engineering and Computer Sciences Department, University of California, Berkeley, California, USA. (August, 1983). Also Tech. Report No. CS-83-9, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada.

3. James F. Blinn, "A Generalization of Algebraic Surface Drawing," *ACM Transactions on Graphics*, Vol. 1, No. 3, July, 1982, pp. 235-256. Also published in *SIGGRAPH '82 Conference Proceedings* (Vol. 16, No. 3),

4. John G. Cleary, Brian Wyvill, Graham M. Birtwistle, and Reddy Vatti, *Multiprocessor Ray Tracing*, Technical Report No. 83/128/17, Department of Computer Science, The University of Calgary (October, 1983).

5. Franklin C. Crow, "Shadow Algorithms for Computer Graphics," pp. 242-248 in *SIGGRAPH '77 Conference Proceedings*, ACM,(July, 1977).

6. Mark E. Dippé, *Spatiotemporal Functional Prefiltering*, Ph.D. Thesis, University of California, Berkeley, California (1984).

7. Eugene Fiume, Alain Fournier, and Larry Rudolph, "A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General-Purpose Ultracomputer: Preliminary Report," pp. 11-21 in *Proceedings Graphics Interface '83*, (May, 1983).

8. W. Randolph Franklin, "A Linear Time Exact Hidden Surface Algorithm," pp. 117-123 in *SIGGRAPH '80 Conference Proceedings*, ACM,(July, 1980).

9. Fuchs, H. and Poulton, J., "Pixel-Planes: A VLSI-Oriented Design for a Raster Graphics Engine," *VLSI Design*. No. 3, 1981, pp. 20-28.

10. Fussell, D. and Rathi, B., "A VLSI-Oriented Architecture for Real-Time Display of Shaded Polygons," pp. 373-380 in *Graphics Interface '82*, (1982).

11. Clark, James H., "The Geometry Engine: A VLSI System for Graphics," pp. 127-133 in *SIGGRAPH '82 Conference Proceedings*, (July, 1982).

12. Roy A. Hall and Donald P. Greenberg, "A Testbed for Realistic Image Synthesis," *IEEE Computer Graphics and Applications*, Vol. 3, No. 8, November, 1983, pp. 10-19.

13. James T. Kajiya, "New Techniques for Raytracing Procedurally Defined Objects," *ACM Transactions on Graphics*, Vol. 2, No. 3, July, 1983, pp. 161-181.

14. Hans P. Moravec, "3D Graphics and the Wave Theory," pp. 289-296 in *SIGGRAPH '81 Conference Proceedings*, (August, 1981).

15. H. Nishimura, H. Ohno, T. Kawata, I. Shirakawa, and K. Omura, "LINKS-1: A Parallel Pipelined Multimicrocomputer System for Image Creation," pp. 387-394 in *Proceedings of the 10th Symposium on Computer Architecture*, SIGARCH,(1983).

16. George Pólya and Gabor Szegö, *Problems and Theorems in Analysis I*, Springer-Verlag, New York (1972).

17. Steven M. Rubin and J. Turner Whitted, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," pp. 110-116 in *SIGGRAPH '80 Conference Proceedings*, ACM,(July, 1980).

18. Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker, "A Characterization of Ten Hidden Surface Algorithms," *ACM Computing Surveys*, Vol. 6, No. 1, March, 1974, pp. 1-55.

19. J. E. Thornton, *Design of a Computer: The Control Data 6600*, Scott, Foresman and Company, Glenview, Illinois (1970).

20. Michael Ullner, *Parallel Machines for Computer Graphics*, Ph.D. Thesis, California Institute of Technology, Pasadena, California (1983).

21. Weinberg, Richard, "Parallel Processing Image Synthesis and Anti-Aliasing," pp. 55-62 in *SIGGRAPH '81 Conference Proceedings*, (August, 1981).

22. J. Turner Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, Vol. 23, No. 6, June, 1980, pp. 343-349.

Antialiasing by Stochastic Sampling

Rob Cook

Computer Graphics Department

Computer Division

Lucasfilm Ltd

ABSTRACT

Ray tracing, ray casting, and other forms of point sampling are important techniques in computer graphics, but their usefulness has been undermined by aliasing artifacts that have been considered unavoidable. This paper shows that these artifacts are not inherent, but are a consequence of using a regularly spaced sampling grid. If the samples occur at appropriate nonuniformly spaced locations, frequencies above the Nyquist limit do not alias, but instead appear as uniform white noise of the correct average intensity. In addition to solving the aliasing problem for ray tracing, stochastic sampling forms the basis of distributed ray tracing, which can simulate fuzzy phenomena, such as motion blur, depth of field, penumbras, gloss, and translucency. The correct simulation of these phenomena involves integrals that are difficult to do directly. Stochastically distributing the rays, however, performs the integrations with a Monte Carlo technique.

General terms: Algorithms

An earlier version of this paper appeared as Lucasfilm Technical Memo #94, "Antialiased Point Sampling", 1983.

Categories and Subject Descriptors: I.3.3 [**Computer Graphics**] Picture/Image Generation; I.3.7 [**Computer Graphics**] Three-Dimensional Graphics and Realism.

Keywords: Antialiasing, filtering, image synthesis, Monte Carlo integration, motion blur, raster graphics, ray tracing, sampling, stochastic sampling.

1. Introduction

Because pixels are discrete, computer graphics is inherently a sampling process. Frequencies greater than half the sampling frequency (the *Nyquist limit*) can produce aliasing artifacts, such as "jaggies" on the edges of objects [4], jagged highlights [16], strobing and other forms of temporal aliasing [10], and Moire patterns in textures [4]. These artifacts are tolerated in some real time applications in which speed is more vital than beauty, but they are unacceptable in realistic image synthesis.

There are two types of rendering algorithms in computer graphics: analytic and discrete. Analytic algorithms can filter the image, eliminating the high frequencies that can cause aliasing. This filtering tends to be complicated and time consuming, but it can eliminate certain types of aliasing very effectively [4,5]. Discrete algorithms, including ray tracing, sample the image at infinitesimally small points, and so by their nature appear to preclude filtering the image. Thus they are plagued by seemingly inherent aliasing artifacts. This is unfortunate, for these algorithms are much simpler, more elegant, and more amenable to hardware than the analytic methods. They are also capable of many features that are difficult to do analytically, such as shadows, reflection, refraction [14,8], constructive solid geometry [11], motion blur, depth of field, translucency, and blurry reflections [3].

There are two existing discrete approaches to alleviating the aliasing problem: oversampling and adaptive sampling. Oversampling reduces aliasing by raising the Nyquist limit. It does not eliminate aliasing. No matter how much a picture is oversampled, there are still frequencies that will alias and alias badly. For example, even with a million by a million samples per pixel, a picket fence with vertical pickets spaced one and one ten millionth of a pixel apart will alias with large scale effects on the picture. The more oversampled the picture, the less the likelihood of noticeable aliasing, but it can still happen (and probably will when least expected).

In adaptive sampling, additional rays are traced near edges [14]. This complicates an otherwise simple algorithm. Unlike oversampling, it can antialias edges reliably, but in order to do this it may require a large number of rays if there are a large number of edges. In the picket fence example above, it would either alias or require a large number of rays per pixel.

This paper presents a new discrete approach to antialiasing called *stochastic sampling*. It is a Monte Carlo technique [7] that avoids aliasing by sampling at appropriate, nonuniformly spaced locations. Stochastic sampling is inherently different from either oversampling or adaptive sampling, but it can be combined with either of them. It eliminates aliasing as effectively as any analytic method, instead of simply making the problem more palatable. Frequencies above the Nyquist limit do not alias, but instead appear as uniform white noise of the correct average intensity. This applies to *all* forms of aliasing, including highlight aliasing.

Stochastic sampling opens up new possibilities for image synthesis. Ray tracing can be treated more fully as a sampling process, and the sampling need not be restricted to spatial sampling. Done with proper antialiasing, rays can sample motion, the camera lens, and the entire shading function. This is called *distributed* or *probabilistic ray tracing* [3].

Distributed ray tracing allows the simulation of fuzzy phenomena, such as motion blur, depth of field, penumbras, gloss, and translucency. Ray traced images are sharp because ray directions are determined precisely and exclusively from geometry. Fuzzy phenomena would seem to require large numbers of additional samples per ray. By distributing the rays stochastically instead of adding more of them, however, fuzzy phenomena can be rendered with no additional rays. In the case of motion blur, for example, rather than taking multiple time samples at every spatial location, the rays are distributed in time so that rays at different spatial locations are traced at different instants of time.

2. Uniform Point Sampling

In a point-sampled picture, frequencies greater than the Nyquist limit are inadequately sampled. Sampling theory predicts that if the samples are uniformly spaced, these frequencies can appear as aliases, i.e., they can appear falsely as low frequencies [2,9].

Consider for the moment one-dimensional sampling. Let a signal $f(x)$ be sampled at regular intervals of time, i.e., at times nT for integer n, where T is the time period between samples so that $1/T$ is the sampling frequency. This sampling is equivalent to multiplication by the *shah* function:

$$\text{III}(x/T) = \sum_{n=-\infty}^{\infty} \delta(x/T - n)$$

Let the Fourier transform of $f(x)$ be $F(x)$. The Fourier transform of $\text{III}(x/T)$ is $(1/T)\text{III}(xT)$. After sampling, information about $f(x)$ is preserved only at the sampling points.

This is illustrated in the frequency domain in Figure 1. Figure 1a shows a signal that is a single sine wave whose frequency is below the Nyquist limit. Sampling involves convolving the signal with the sampling grid of Figure 1b to produce the spectrum shown in Figure 1c. An ideal reconstruction filter, shown in Figure 1d, would extract the original signal, as in Figure 1e. In Figures 1f through 1j, the same process is repeated for a single sine wave whose frequency is above the Nyquist limit. In this case, the sampling process can fold the high frequency sine wave into low frequencies, as shown in Figure 1h. These false frequencies, or aliases, cannot be separated from frequencies that are a part of the original signal. The part of the spectrum extracted by the reconstruction filter contains these aliases, as shown in Figure 1j.

Sampling theory thus predicts the aliasing of frequencies greater than the Nyquist limit. This

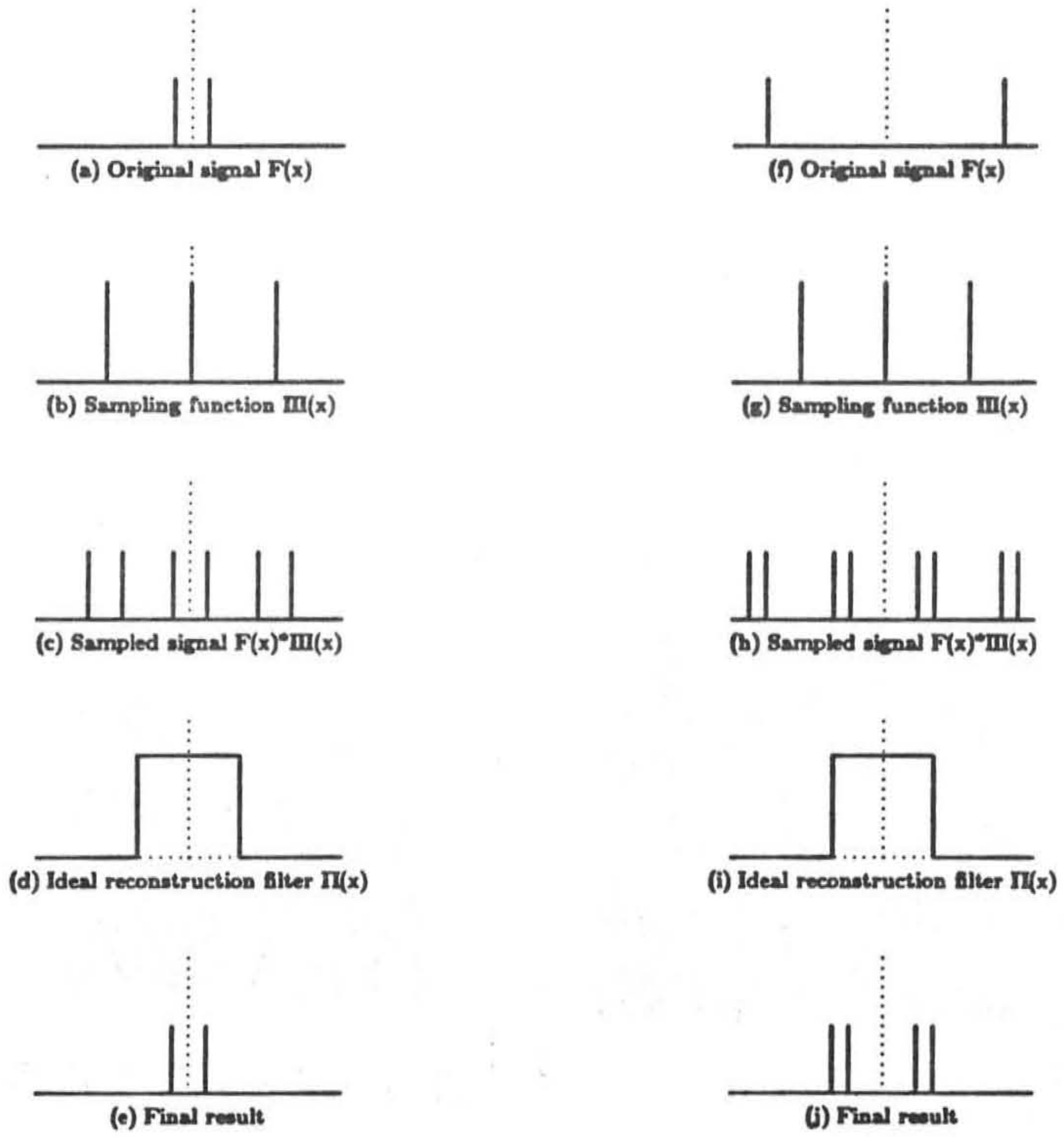

(a) Original signal F(x)
(f) Original signal F(x)
(b) Sampling function III(x)
(g) Sampling function III(x)
(c) Sampled signal F(x)*III(x)
(h) Sampled signal F(x)*III(x)
(d) Ideal reconstruction filter Π(x)
(i) Ideal reconstruction filter Π(x)
(e) Final result
(j) Final result

Figure 1. Point sampling shown in the frequency domain. The original signal F(x) is convolved with the sampling grid III(x) and the result is multiplied by an ideal reconstruction filter Π(x). The process is shown for a sine wave with a frequency below the Nyquist limit in (a) through (e) and above the Nyquist limit in (f) through (j).

aliasing has been thought to be unavoidable, but it is actually a consequence of the regularity of the sampling grid. If the sample points are not regularly spaced, the theory that predicts aliasing is not applicable. Frequencies greater than the Nyquist limit do not necessarily appear as aliases. If the positions are chosen correctly, the energy in those frequencies appears as white noise rather than as aliasing.

3. Poisson Disk Distribution

One excellent distribution of sample locations is found in the human eye. The eye has a limited number of photoreceptors, and, like any other sampling process, it has a Nyquist limit. Yet our eyes are not prone to aliasing. Recently some experiments have been able to detect some aliasing in the human eye, but only under stringently controlled conditions [15].

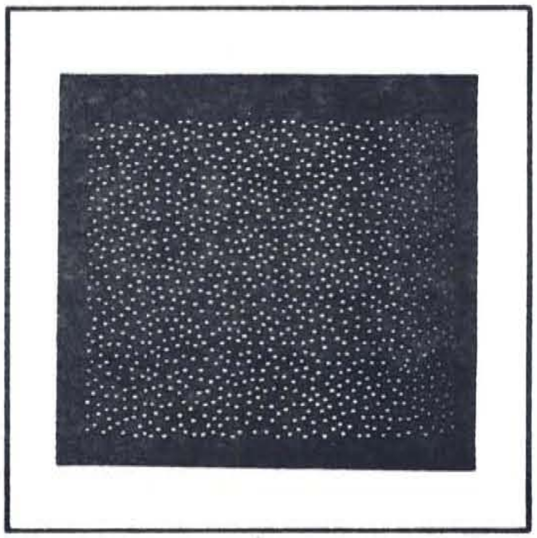


Figure 2a. Monkey eye photoreceptor distribution

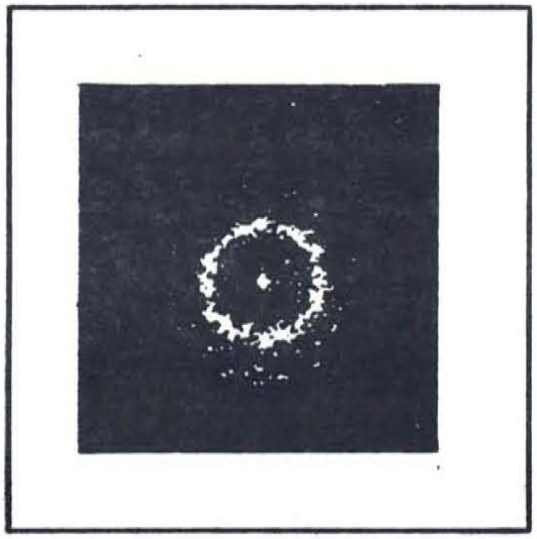


Figure 2b. Optical transform of monkey eye

Yellott [17] has studied the distribution of cones in the eye, noting that this distribution is similar in humans and in rhesus monkeys, Figure 2a is a picture of the distribution of cones in a region of the eye of a rhesus monkey. Yellott took the optical Fourier transform of this distribution, with the result shown in Figure 2b.

This distribution is called a *Poisson disk distribution*, and it is shown schematically in the frequency domain in Figure 3b. There is a spike at the origin (the DC component) and a sea of noise beyond the Nyquist limit. In effect, the samples are randomly placed with the restriction that no two samples are closer together than a certain distance, and that any circular region greater than a certain radius contains at least one sample.

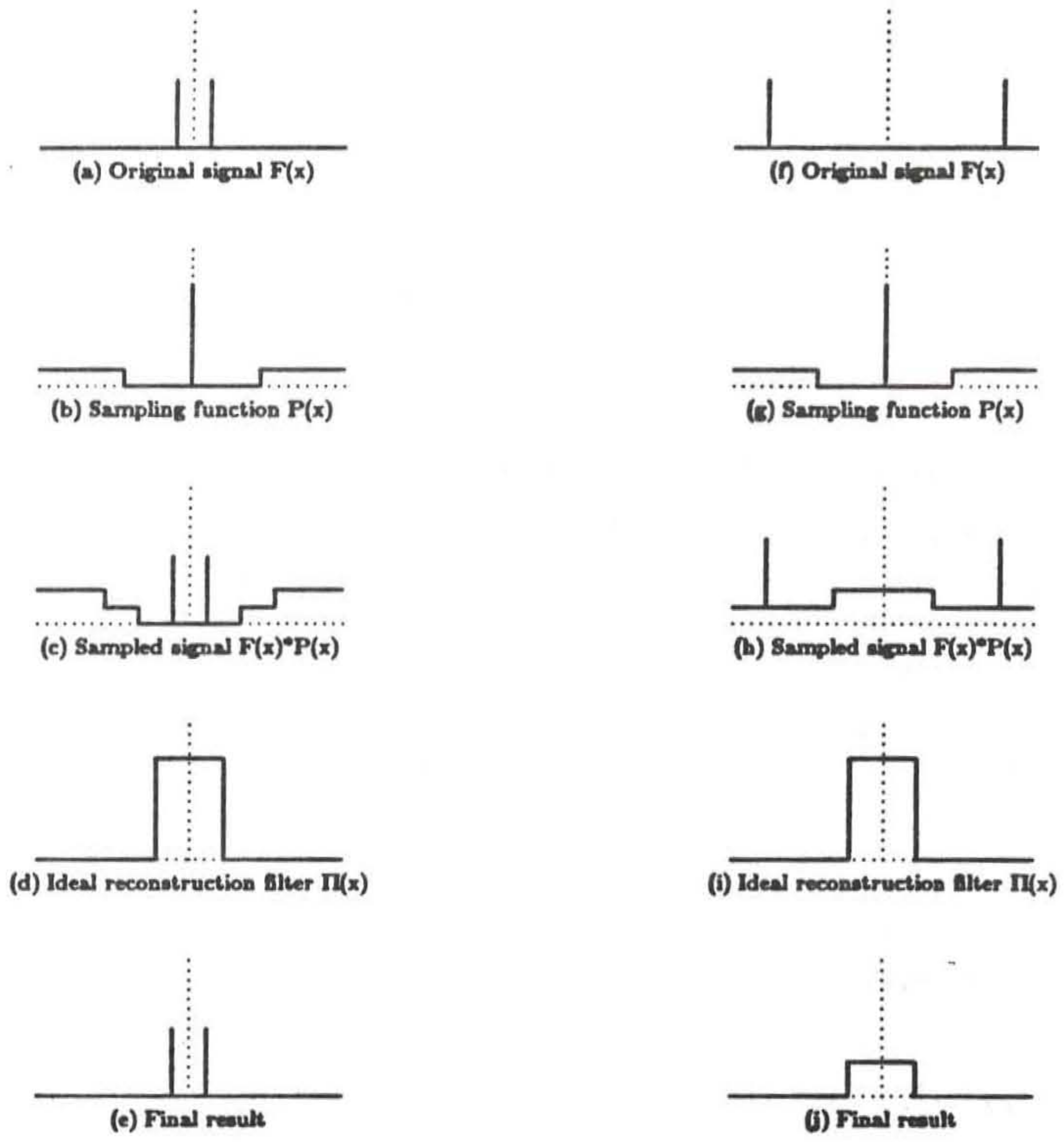

(a) Original signal F(x)
(f) Original signal F(x)
(b) Sampling function P(x)
(g) Sampling function P(x)
(c) Sampled signal F(x)*P(x)
(h) Sampled signal F(x)*P(x)
(d) Ideal reconstruction filter Π(x)
(i) Ideal reconstruction filter Π(x)
(e) Final result
(j) Final result

Figure 3. Poisson sampling shown in the frequency domain.

Now let us analyze point sampling using a Poisson disk sampling distribution instead of a regular grid. Figure 3a shows a signal that is a single sine wave whose frequency is below the Nyquist limit. Convolution with the Poisson sampling grid of Figure 3b produces the spectrum in Figure 3c. The ideal reconstruction filter of Figure 3d would extract the original signal, Figure 3e.

Figure 3f shows a sine wave whose frequency is above the Nyquist limit. Convolution with the Poisson sampling grid produces the spectrum in Figure 3h. An ideal reconstruction filter would extract white noise, as shown in Figure 3j. This white noise replaces the aliasing of Figure 1j.

The minimum distance restriction controls the magnitude of the noise. For example, film grain appears to have a random distribution [13], without the minimum distance restriction of a Poisson disk distribution. With a random distribution, the samples tend to bunch up in some places and leave large gaps in other places. Film does not alias, but it is more prone to noise than the eye.

The implementation of Poisson disk sampling to image rendering is straightforward. A lookup table is created by generating random sample locations and discarding any locations that are closer than a certain distance from any of the locations already chosen. Locations are generated until the sampling region is full. Filter values are calculated that describe how each sample affects the neighboring pixels, and these filter values must be normalized. The locations and filter values are stored in a table. This method would produce good pictures, but it would also require a large lookup table.

4. Jittering

4.1. Theory

Jittering, or adding noise to a set of sample locations, is a form of stochastic sampling that approximates a Poisson disk distribution and is well suited to image rendering. Jitter was analyzed in one dimension ("time") by Balakrishnan [1], who calculated the effect of "time jitter", in which the nth sample is jittered by an amount ζ_n so that it occurs at time $nT+\zeta_n$, where T is the sampling period. If the ζ_n are uncorrelated, jittering has the following effects:

- High frequencies are attenuated, i.e., the signal is filtered.
- The energy lost to the attenuation appears as uniform white noise. The intensity of the

noise equals the intensity of the attenuated part of the signal.

- The basic composition of the spectrum otherwise does not change.

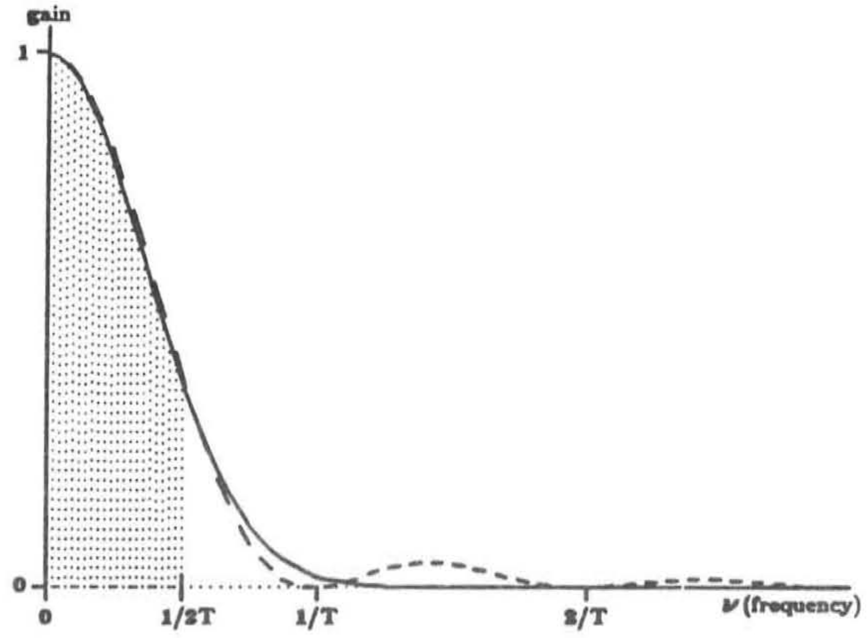


Figure 4. Attenutation due to jitter. The broken line shows the filter for white noise jitter, the solid line for Gaussian jitter. The shaded area is inside the Nyquist limit.

Balakrishnan analyzed two types of uncorrelated jitter: white noise jitter and Gaussian jitter.

For *white noise jitter*, in which the values of ς are uniformly distributed between $-\gamma T$ and γT, the gain due to jitter as a function of frequency ν is

$$\left[\frac{\sin(2\pi\gamma\nu T)}{2\pi\gamma\nu T}\right]^2 \tag{1}$$

This function for $\gamma = 1/2$ is plotted with a dashed line in Figure 4.
For *Gaussian jitter* in which the variance of the values of ς is σ^2, the gain is

$$e^{-(2\pi\nu\sigma)^2} \tag{2}$$

as shown with a solid line in Figure 4 for $\sigma = T/6.5$. The examples in this paper were made using white noise jitter.

As with any non-ideal filter, jittering does not eliminate aliasing completely, but it does reduce it substantially. The Nyquist limit of $1/(2T)$ is indicated in the figure by the shaded area. Notice that the width of the filter can be scaled by adjusting γ or σ. This gives control of the tradeoff between decreased aliasing and increased noise.

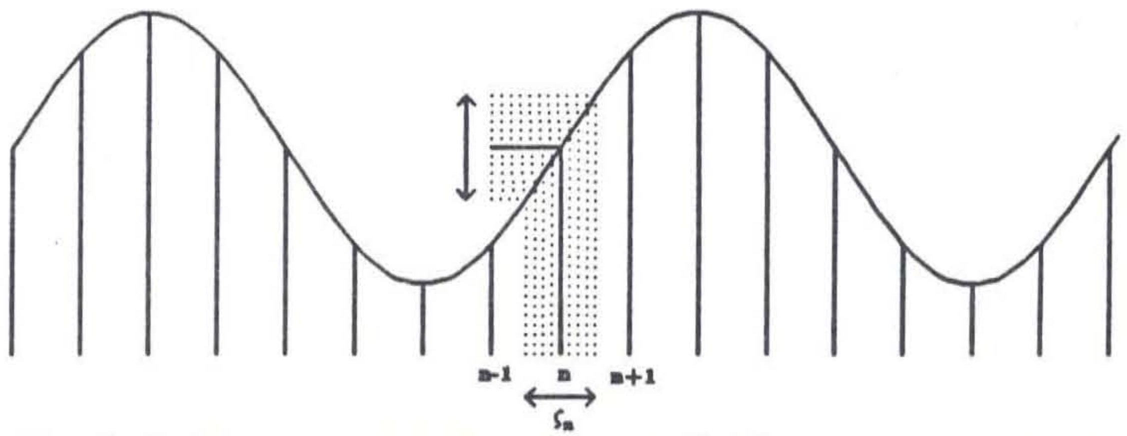


Figure 5a. The effect of white noise jittering on a sine wave with a frequency below the Nyquist limit. Sample n shown occurs at a random location in the dotted region. The jitter indicated by the horizontal arrow results in a sampled value that can vary by the amount indicated by the vertical arrow.

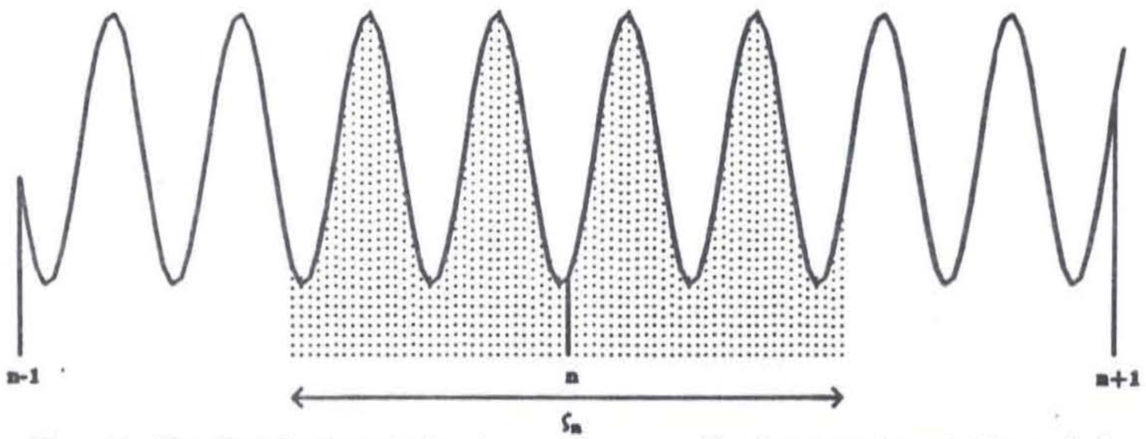


Figure 5b. The effect of white noise jittering on a sine wave with a frequency above the Nyquist limit. The jitter indicated by the horizontal arrow results in a sampled value that is almost pure noise.

For an intuitive explanation of these equations, consider the sine wave shown in Figure 5a, with samples at regularly spaced intervals λ as shown. These samples are inside the Nyquist limit and therefore sample the the sine wave properly. Jittering the location of each sample n by some ς_n in the range $-\lambda/2 < \varsigma_n < \lambda/2$ is similar to adding some noise to the amplitude; note that the basic sine wave frequency is not lost. This noise is less for sine waves with a lower frequency relative to the sampling frequency.

Now consider the sine wave shown in Figure 5b. Here the sampling rate is not sufficient for the frequency of the sine wave, so regularly spaced samples can alias. The jittered sample, however, can occur at any amplitude. If there are exactly a whole number of cycles in the range $-\lambda/2 < \varsigma_n < \lambda/2$, then the amplitude we sample is completely random, since there is an equal probability of sampling each part of the sine wave. In this case, none of the energy from the sine wave produces aliasing; it all becomes white noise. This corresponds to the zero points of the dashed line in Figure 4. If the sine wave frequency is not an exact multiple of λ, then some part of the wave will be more likely to be sampled than others. In this case there is some attenuated aliasing, and some white noise because there is some chance of hitting each part of the wave. This attenuation is greater for higher frequencies, because with more cycles of the wave, there is less preference for one part of the wave over another. Note also that the average amplitude of the noise (the DC component or grey level) is equal to the average amplitude of the sine wave. The grey level of the signal is preserved.

4.2. Implementation

The extension of jittering to two dimensions is straightforward. Consider a pixel as a regular grid of *subpixels*, each with one sample point. Each sample point is placed in the middle of a subpixel, and then noise is added to the x and y locations independently so that each sample point occurs at some random location within its subpixel. This method produces no perceptible aliasing, and has a low noise level. Because there is no minimum distance restriction, though, the pictures produced with this distribution should be somewhat noisier than those produced with a true Poisson disk distribution.

Once the visibility at the sample points is known, the sample values must be resampled and filtered to obtain pixel values. The most common analytic filter is a box filter, used primarily because it is relatively easy to compute [4]. Box filtered pictures can have aliasing artifacts, from

Moire patterns to spiral rope effects on thin polygons. Weighted filters are much better and can be done analytically, but they are computationally expensive [5].

Weighted filters are easy with point sampling. The filter values that specify the amount that each point contributes to the surrounding pixels are stored in a table, and changing filters is simply a matter of changing the lookup table. Filter values are a function of the position of a sample point relative to the surrounding pixels. If the random component of the sample location is small compared to the width of the filter, the effect of the random component can usually be ignored and the filters can be prenormalized. In our case, the deviation of the sample location from the center of the subpixel is small, so we can use the filter value for the center of each subpixel regardless of the pattern rotation or the jitter.

5. Examples

We have been unable to find a case that aliases noticeably due to stochastic point sampling, even with test cases intentionally designed to alias badly. Of course, with prior knowledge of the random numbers used in the jittering, one could construct a case that aliases. But we claim that one cannot construct an environment that aliases noticeably without such prior knowledge, or at least that the aliasing would also occur with any analytic algorithm at the same data precision.

One case designed to alias is the comb of triangular slivers illustrated in Figure 6a. Each triangle is 1.01 pixels wide at the base and 50 pixels high. The triangles are placed in a row horizontally 1.01 pixels apart. In Figures 6b and 6c, a comb containing about 50 triangles is rendered by sampling with 16 samples per pixel. In Figure 6b, the comb is rendered with a regular 4 by 4 grid of samples. In Figure 6c, the regular 4 by 4 grid is jittered by $\varsigma = \pm 1/8$ pixel in x and y. Figure 6b is grossly aliased: there are just a few large overlapping triangles spaced $100/4 \doteq 25$ pixels apart. This aliasing is replaced by noise in Figure 6c. The results are similar if the comb is sheared horizontally.

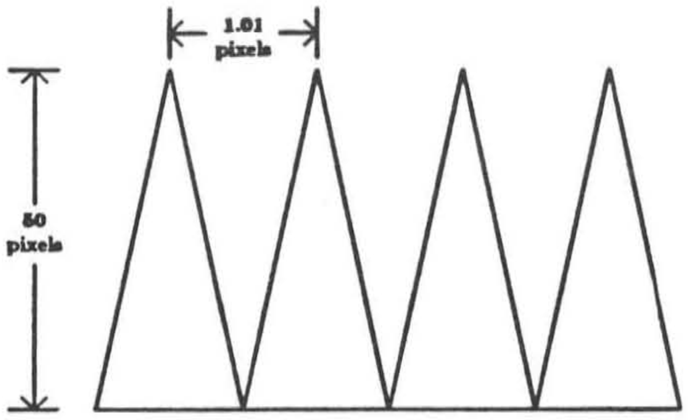


Figure 8a. Comb of triangles example.

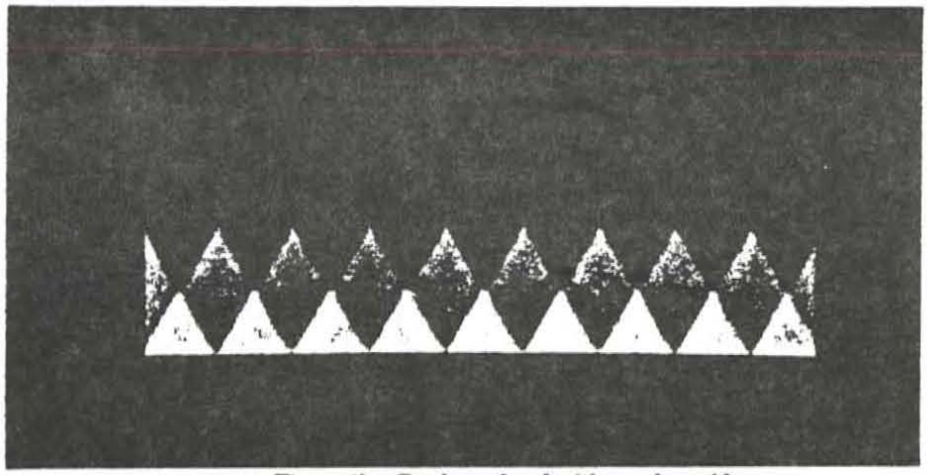


Figure 8b. Comb rendered with regular grid.

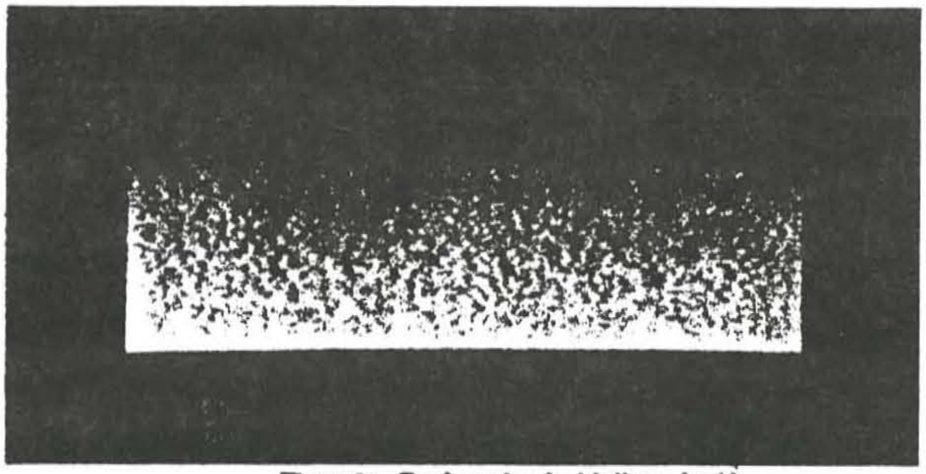


Figure 8c. Comb rendered with jittered grid.

One interesting case of aliasing was encountered, but it was not due to point sampling. A picket fence with pickets $2+\epsilon$ pixels apart is inside the Nyquist limit for $\epsilon>0$, but it will alias for small ϵ because of the nonideal reconstruction filter of the video. The pixels have correct values that start out with odd pixels whiter than the even pixels. After $1/\epsilon$ pickets, however, the even pixels will be whiter than the odd pixels. In the transition region the picture is a solid grey. The transition from distinct stripes to solid grey and back to stripes recurs with a period of $1/\epsilon$. This aliasing is distinctly noticeable, but it is caused by the imperfect reconstruction filter, not by the point sampling.

6. Distributed Ray Tracing

In the previous sections, we applied stochastic sampling to the two-dimensional distribution of the sample points used for determining visibility in a z buffer or ray casting algorithm. This is appropriate because the final signal we are sampling in image synthesis is a two-dimensional function. But this function may depend on integrals over many other dimensions. For example, the shade of a visible point is an integral of the light incident from all directions. Motion blur is an integral over time. Depth of field is an integral over the lens area.

In the past, ray tracing has made certain simplifying assumptions in order to avoid the evaluation of these integrals. But the evaluation of these integrals is essential for rendering a whole range of fuzzy phenomena, such as penumbras, fuzzy reflections, translucency, depth of field, and motion blur. Thus ray tracing has been limited to sharp shadows, sharp reflections, sharp refractions, pinhole cameras, and instantaneous shutters.

These integrals can be evaluated with stochastic sampling. Ray tracing can be regarded as sampling a *d*-dimensional function, where the additional dimensions are things like time and reflection angle. All that is necessary to sample this function is to distribute the sample points in *d* dimensions instead of in just two. This is called distributed or probabilistic ray tracing.

No extra samples are required for distributed ray tracing beyond those needed for two-dimensional stochastic sampling. The final function is two-dimensional, and that is what determines the sampling rate, no matter how many dimensions have been integrated to produce that final two-dimensional function.

These are some of the benefits of distributed ray tracing:

- Distributing reflected rays according to the specular distribution function produces gloss (blurred reflection).
- Distributing transmitted rays produces translucency (blurred transparency).
- Distributing shadow rays through the solid angle of the light sources produces penumbras.
- Distributing ray origins over the camera lens area produces depth of field.
- Distributing rays in time produces motion blur.

6.1. Stochastic Sampling Applied to Distributed Ray Tracing

One way to distribute the rays in the additional dimensions is with uncorrelated random values. For example, one could pick a random time for each ray or a random point on a light source for each shadow ray, with no correlation from ray to ray. This approach produces pictures that are exceedingly noisy, due to the bunching up of samples.

So once again it is important to impose a Poisson disk restriction, so that samples that are close spatially are not too close in other dimensions. Yet aliasing can result if the sample locations in the additional dimensions are too correlated with their spatial locations. For example, if the samples on the left side of the pixel are consistently at an earlier time than those on the right side of the pixel, an object moving from right to left might be missed by every sample while an object moving from left to right might be hit by every sample. A convenient method for generating a

52	61	4	13	20	29	36	45
14	3	62	51	46	35	30	19
53	60	5	12	21	28	37	44
11	6	59	54	43	38	27	22
55	58	7	10	23	26	39	42
9	8	57	56	41	40	25	24
50	63	2	15	18	31	34	47
16	1	64	49	48	33	32	17

Figure 7. The Franklin 8x8 magic square

good distribution is to use magic squares, which are fairly homogeneous arrangements of a sequence of numbers. For example, to assign times to an 8 by 8 grid of sample points, one could use 8 by 8 magic squares [6], such as the one shown in Figure 7. The s^{th} sample would have a prototype time

$$t_s = \frac{M_s+0.5}{64.0},$$

where M_s is the s^{th} value in the magic square. A random jitter of $\pm 1/128$ should be added to the prototype time to obtain the actual time for a sample. For example, the sample in the upper left subpixel would have a time $51.5 \leq t \leq 52.5$.

6.2. Weighted Distributions

Sometimes we need to weight the samples. For example, we may want to weight the reflected samples according to the specular reflection function, or we may want to use a weighted temporal filter. We would like to distribute the sample points so that the chance of a location being sampled is proportional to the value of the filter at that location.

One approach would be to distribute the samples evenly, and then later weight each ray according to the filter. A better approach is shown in Figure 8, in which the filter is divided in regions of equal area. Each region is sampled by one sample point, so that each sample now represents an equal area under the filter, with the samples spaced further apart for smaller filter values and closer together for larger filter values. This avoids the multiplications necessary for the weighting and also puts the samples where they will do the most good.

Position each sample point at the center of its region, and then jitter its location to a random location in the region. This approximates the filter with a set of step functions. Note that the size of the jitter varies from sample to sample.

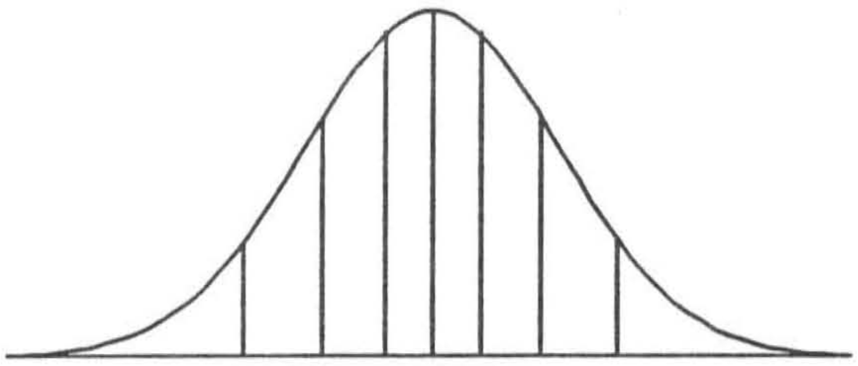


Figure 8. Sample distribution for a weighted function

6.3. Summary of Distributed Ray Tracing

Distributed ray tracing is discussed in detail in a previous paper [3]. This section summarizes the distributed ray tracing algorithm from the viewpoint of stochastic sampling.

The intensity of a pixel on the screen is an analytic function that involves several nested integrals: integrals over time, over the pixel region, and over the lens area, as well as an integral of reflectance times illumination over the reflected hemisphere and an integral of transmittance times illumination over the transmitted hemisphere. This integral can be tremendously complicated, but we can point sample the function regardless of how complicated it is. If the function depends on n parameters, the function is sampled in the n dimensions defined by those parameters. Rather than adding more rays for each dimension, the existing rays are distributed in each dimension according to the values of the corresponding parameter.

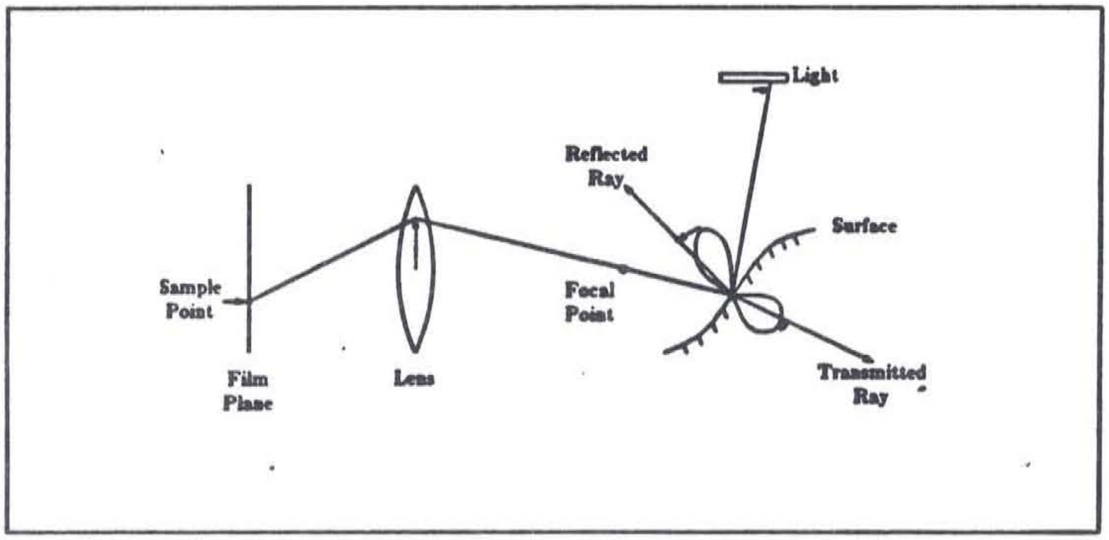


Figure 9. Distributed Ray Tracing.

The distributed ray tracing algorithm is illustrated in Figure 9. For a single ray:

- Determine the spatial location of the rays by jittering.
- Determine the time for the ray from a jittered magic square. Move the objects accordingly.
- Construct a temporary ray from the eye point (center of the lens) to a point on the screen. Determine a location on the lens from a jittered pattern of sample locations on the lens. Trace a ray from that location through the focal point of the original ray.
- Determine which object is visible using standard ray casting techniques.
- Trace the shadow rays. For each light source, determine the location on the light for the shadow ray. The number of rays traced to a location on the light should be proportional to the intensity and projected area of that location as seen from the surface.
- Trace a reflection ray. The reflection direction is determined by jittering a set of directions

that are distributed according to the specular reflection function.

- Trace a transparency ray. The transparency direction is determined by jittering a set of directions that are distributed according to the specular transmission function. Trace a ray in that direction from the visible point.

The visible surface calculation is straightforward. Since each ray occurs at a single instant of time, the first step is to update the positions of the objects for that instant of time. The next is to construct a ray from the lens to the sample point and find the closest object that the ray intersects.

Intersecting surfaces are handled trivially because we never have to calculate the line of intersection; we merely have to determine which is in front at a given location and time. At each sample point only one of the surfaces is visible. The intersections can even be motion blurred, a problem that would be terrifying with an analytic method.

The union, intersection, difference problem is easily solved with ray tracing or point sampling [11]. These calculations are also correctly motion blurred.

7. Examples of Distributed Ray Tracing

Figure 10 is closeup of the ray traced picture *1984*. This is the 4 ball; it remains stationary for most of the time the shutter is open and moves quickly to the upper right just before the shutter closes. The blur is quite extreme, and yet the image looks noisy instead of aliased. This picture was made with 16 samples per pixel.

Figures 11a and 11b are two frames from the short film *The Adventures of Andre and Wally B.* [12] These extreme examples of articulated motion blur were rendered with a scanline algorithm that uses point sampling and a z buffer to determine visibility. In these frames, an adaptive method automatically sampled regions with a lot of motion blur at 64 samples per pixel and other

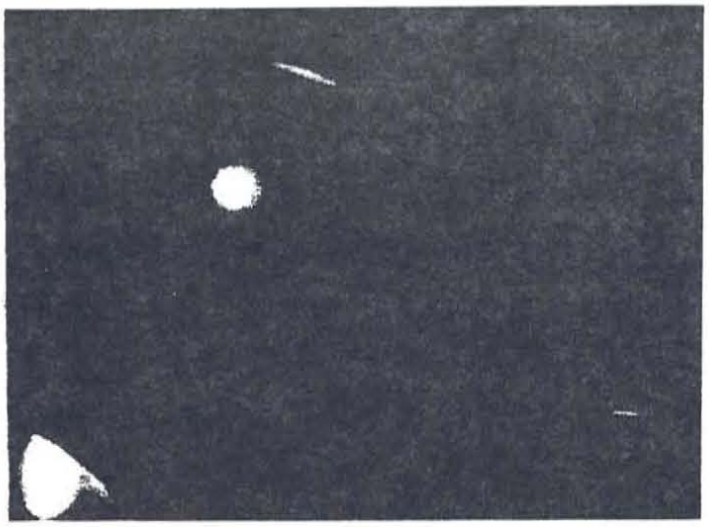


Figure 10. Closeup of *1984*.

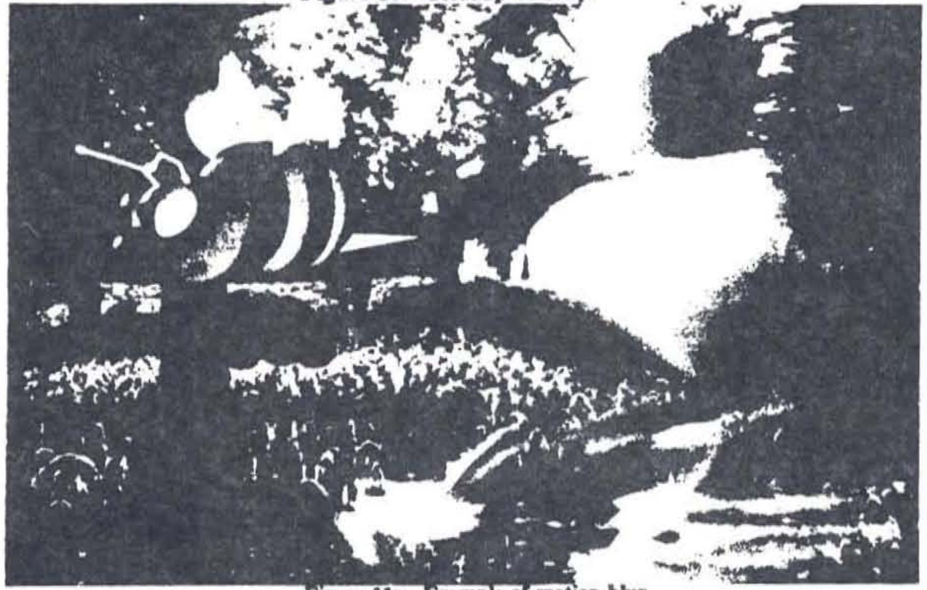


Figure 11a. Example of motion blur.

Figure 11b. Example of motion blur.

regions with 16 samples per pixel. This cuts down considerably on the noise level and helps avoid needless computation.

Other examples of distributed ray tracing, including depth of field, penumbras, blurry reflections and translucence, have appeared in a previous paper [3]. In all cases, areas of extreme blur become noisy instead of aliasing.

8. Discussion and Conclusions

Correctly chosen nonuniform sample locations can turn the aliasing of higher frequencies into white noise. The magnitude of this noise is determined by the sampling frequency. Frequencies beyond the numerical precision of the machine will alias, but this is also true of analytic antialiasing methods.

Based on this, 16 samples per pixel is usually enough. In fact, we have found that we need more than this only in extremely motion blurred or out of focus shots. The number of samples can be adjusted for each pixel based on the maximum amount of blur expected for that pixel. In fact, stochastic sampling should also work well when integrated with adaptive sampling.

A simple and effective implementation of stochastic sampling is to jitter a two-dimensional regular grid. In distributed ray tracing, the locations in the non-spatial dimensions can be chosen by jittering a magic square. Care must be taken in bounding moving and out of focus objects.

Stochastic sampling has one major disadvantage. Because the samples are not regularly spaced, forward differencing cannot be used to exploit pixel to pixel coherence.

Stochastic sampling offers the first true antialiasing for ray tracing and point sampling. The ability to antialias these techniques gives them new importance. The shading calculations, which have traditionally been point sampled, are automatically antialiased with stochastic sampling, eliminating problems such as highlight aliasing. Another potential application is texture map sampling. Combined with distributed ray tracing, stochastic sampling also provides a solution to motion blur, depth of field, penumbras, blurry reflections, and translucency.

9. Acknowledgements

The idea of dithering sample locations originally came from Rodney Stock, who provided inspiration and motivation for this work. Alvy Ray Smith found the article on the distribution of cells in the eye. Andy Moorer helped with the theory. Tom Porter made the *1984* picture, suggested the extension of the two-dimensional technique to motion blur, and helped test many of the ideas. A number of discussions with Loren Carpenter were very helpful.

1. BALAKRISHNAN, A. V., "On the Problem of Time Jitter in Sampling," *IRE Transactions on Information Theory*, pp. 226-236 (April 1962).

2. BRACEWELL, RONALD N., *The Fourier Transform and its Applications.*, McGraw-Hill, New York (1978).

3. COOK, ROBERT L., THOMAS PORTER, AND LOREN CARPENTER, "Distributed Ray Tracing," *Computer Graphics* **18**(3), pp. 137-145 (July 1984).

4. CROW, FRANKLIN, "The Use of Greyscale for Improved Raster Display of Vectors and Characters," *Computer Graphics* **12**(3), pp. 1-5 (August 1978).

5. FEIBUSH, ELIOT, MARC LEVOY, AND ROBERT L. COOK, "Synthetic Texturing Using Digital Filtering," *Computer Graphics* **14**(3), pp. 294-301 (July 1980).

6. FRANKLIN, BENJAMIN, "Letter to Peter Collinson, c. 1750," in *The Wonders of Magic Squares*, ed. Jim Moran, Random House, New York (1981).

7. HALTON, JOHN H., "A Retrospective and Prospective Survey of the Monte Carlo Method," *SIAM Review* **12**(1) (January 1970).

8. KAY, DOUGLAS S. AND DONALD P. GREENBERG, "Transparency for Computer Synthesized Images," *Computer Graphics* **13**(2), pp. 158-164 (August 1979).

9. PEARSON, D. E., *Transmission and Display of Pictorial Information,* Pentech Press, London (1975).

10. POTMESIL, MICHAEL AND INDRANIL CHAKRAVARTY, "Modeling Motion Blur in Computer-Generated Images," *Computer Graphics* **17**(3), pp. 389-399 (July 1983).

11. ROTH, S. D., "Ray Casting for Modeling Solids," *Computer Graphics and Image Processing*(18), pp. 109-144 (1982).

12. SMITH, ALVY RAY, LOREN CARPENTER, ED CATMULL, ROB COOK, TOM DUFF, CRAIG GOOD, JOHN LASSETER, SAM LEFFLER, EBEN OSTBY, TOM PORTER, WILLIAM REEVES, AND DAVID SALESIN, *The Adventures of André and Wally B.,* Created by the Lucasfilm Computer Graphics Project. July 1984.

13. SOCIETY OF PHOTOGRAPHIC SCIENTISTS AND ENGINEERS,, *SPSE Handbook of Photographic Science and Engineering,* Wiley, New York (1973).

14. WHITTED, TURNER, "An Improved Illumination Model for Shaded Display," *Communications of the ACM* **23**, pp. 343-349 (1980).

15. WILLIAMS, DAVID R. AND ROBERT COLLIER, "Consequences of Spatial Sampling by a Human Photoreceptor Mosaic," *Science* **221**, pp. 385-387 (22 July 1983).

16. WILLIAMS, LANCE, "Pyramidal Parametrics," *Computer Graphics* **17**(3), pp. 1-11 (July 1983).

17. YELLOTT, JOHN I. JR., "Spectral Consequences of Photoreceptor Sampling in the Rhesus Retina," *Science* **221**, pp. 382-385 (22 July 1983).

Distributed Ray Tracing

Robert L. Cook
Thomas Porter
Loren Carpenter

Computer Division
Lucasfilm Ltd.

Abstract

Ray tracing is one of the most elegant techniques in computer graphics. Many phenomena that are difficult or impossible with other techniques are simple with ray tracing, including shadows, reflections, and refracted light. Ray directions, however, have been determined precisely, and this has limited the capabilities of ray tracing. By distributing the directions of the rays according to the analytic function they sample, ray tracing can incorporate fuzzy phenomena. This provides correct and easy solutions to some previously unsolved or partially solved problems, including motion blur, depth of field, penumbras, translucency, and fuzzy reflections. Motion blur and depth of field calculations can be integrated with the visible surface calculations, avoiding the problems found in previous methods.

CR CATEGORIES AND SUBJECT DESCRIPTORS: I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism;

ADDITIONAL KEY WORDS AND PHRASES: camera, constructive solid geometry, depth of field, focus, gloss, motion blur, penumbras, ray tracing, shadows, translucency, transparency

1. Introduction

Ray tracing algorithms are elegant, simple, and powerful. They can render shadows, reflections, and refracted light, phenomena that are difficult or impossible with other techniques[11]. But ray tracing is currently limited to sharp shadows, sharp reflections, and sharp refraction.

Ray traced images are sharp because ray directions are determined precisely from geometry. Fuzzy phenomenon would seem to require large numbers of additional samples per ray. By distributing the rays rather than adding more of them, however, fuzzy phenomena can be rendered with no additional rays beyond those required for spatially oversampled ray tracing. This approach provides correct and easy solutions to some previously unsolved problems.

This approach has not been possible before because of aliasing. Ray tracing is a form of point sampling and, as such, has been subject to aliasing artifacts. This aliasing is not inherent, however, and ray tracing can be filtered as effectively as any analytic method[4]. The filtering does incur the expense of additional rays, but it is not merely oversampling or adaptive oversampling, which in themselves cannot solve the aliasing problem. This antialiasing is based on an approach proposed by Rodney Stock. It is the subject of a forthcoming paper.

Antialiasing opens up new possibilities for ray tracing. Ray tracing need not be restricted to spatial sampling. If done with proper antialiasing, the rays can sample motion, the camera lens, and the entire shading function. This is called *distributed ray tracing.*

Distributed ray tracing is a new approach to image synthesis. The key is that no extra rays are needed beyond those used for oversampling in space. For example, rather than taking multiple time samples at every spatial location, the rays are distributed in time so that rays at different spatial locations are traced at different instants of time. Once we accept the expense of oversampling in space, distributing the rays offers substantial benefits at little additional cost.

- Sampling the reflected ray according to the specular distribution function produces gloss (blurred reflection).
- Sampling the transmitted ray produces translucency (blurred transparency).
- Sampling the solid angle of the light sources produces penumbras.

- Sampling the camera lens area produces depth of field.
- Sampling in time produces motion blur.

2. Shading

The intensity I of the reflected light at a point on a surface is an integral over the hemisphere above the surface of an illumination function L and a reflection function R[1].

$$I(\phi_r,\theta_r) = \int_{\phi_i}\int_{\theta_i} L(\phi_i,\theta_i)R(\phi_i,\theta_i,\phi_r,\theta_r)\,d\phi_i\,d\theta_i$$

where

(ϕ_i,θ_i) is the angle of incidence, and

(ϕ_r,θ_r) is the angle of reflection.

The complexity of performing this integration has been avoided by making some simplifying assumptions. The following are some of these simplifications:

- Assume that L is a δ function, i.e., that L is zero except for light source directions and that the light sources can be treated as points. The integral is now replaced by a sum over certain discrete directions. This assumption causes sharp shadows.
- Assume that all of the directions that are not light source directions can be grouped together into an ambient light source. This ambient light is the same in all directions, so that L is independent of ϕ_i and θ_i and may be removed from the integral. The integral of R may then be replaced by an average, or ambient, reflectance.
- Assume that the reflectance function R is a δ function, i.e., that the surface is a mirror and reflects light only from the mirror direction. This assumption causes sharp reflections. A corresponding assumption for transmitted light causes sharp refraction.

The shading function may be too complex to compute analytically, but we can point sample its value by distributing the rays, thus avoiding these simplifying assumptions. Illumination rays are not traced toward a single light direction, but are distributed according to the illumination function L. Reflected rays are not traced in a single mirror direction but are distributed according to the reflectance function R.

2.1. Gloss

Reflections are mirror-like in computer graphics, but in real life reflections are often blurred or hazy. The distinctness with which a surface reflects its environment is called *gloss*[5]. Blurred reflections have been discussed by Whitted[11] and by Cook[2]. Any analytic simulation of these reflections must be based on the integral of the reflectance over some solid angle.

Mirror reflections are determined by tracing rays from the surface in the mirror direction. Gloss can be calculated by distributing these secondary rays about the mirror direction. The distribution is weighted according to the same distribution function that determines the highlights.

This method was originally suggested by Whitted[11], and it replaces the usual specular component. Rays that reflect light sources produce highlights.

2.2. Translucency

Light transmitted through an object is described by an equation similar to that for reflected light, except that the reflectance function R is replaced by a transmittance function T and the integral is performed over the hemisphere behind the surface. The transmitted light can have ambient, diffuse, and specular components[5].

Computer graphics has included transparency, in which T is assumed to be a δ function and the images seen through transparent objects are sharp. Translucency differs from transparency in that the images seen through translucent objects are not distinct. The problem of translucency is analogous to the problem of gloss. Gloss requires an integral of the reflected light, and translucency requires a corresponding integral of the transmitted light.

Translucency is calculated by distributing the secondary rays about the main direction of the transmitted light. Just as the distribution of the reflected rays is defined by the specular reflectance function, the distribution of the transmitted rays is defined by a specular transmittance function.

2.3. Penumbras

Penumbras occur where a light source is partially obscured. The reflected intensity due to such a light is proportional to the solid angle of the visible portion of the light. The solid angle has been explicitly included in a shading model[3], but no algorithms have been suggested for determining this solid angle because of the complexity of the computation involved. The only attempt at penumbras known to the authors seems to solve only a very special case[7].

Shadows can be calculated by tracing rays from the surface to the light sources, and penumbras can be calculated by distributing these secondary rays. The shadow ray can be traced to any point on the light source, not just not to a single light source location. The distribution of the shadow rays must be weighted according the projected area and brightness of different parts of the light source. The number of rays traced to each region should be proportional to the amount of the light's energy that would come from that region if the light was

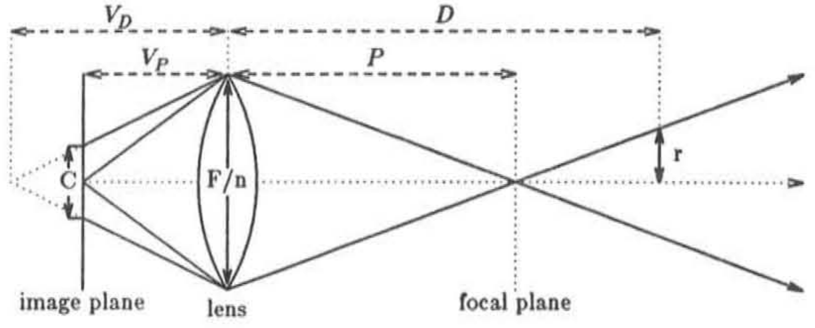


Figure 1. Circle of Confusion.

completely unobscured. The proportion of lighted sample points in a region of the surface is then equal to the proportion of that light's intensity that is visible in that region.

3. Depth of Field

Cameras and the eye have a finite lens aperture, and hence their images have a finite depth of field. Each point in the scene appears as a circle on the image plane. This circle is called the circle of confusion, and its size depends on the distance to the point and on the lens optics. Depth of field can be an unwanted artifact, but it can also be a desirable effect.

Most computer graphics has been based on a pinhole camera model with every object in sharp focus. Potmesil simulated depth of field with a postprocessing technique. Each object is first rendered in sharp focus (i.e., with a pinhole camera model), and later each sharply rendered object is convolved with a filter the size of the circle of confusion[8]. The program spends most of its time in the focus postprocessor, and this time increases dramatically as the aperture decreases.

Such a postprocessing approach can never be completely correct. This is because visibility is calculated from a single point, the center of the lens. The view of the environment is different from different parts of the lens, and the differences include changes in visibility and shading that cannot be accounted for by a postprocessing approach.

For example, consider an object that is extremely out of focus in front of an object that is in focus. Visible surface calculations done with the pinhole model determine the visibility from the center of the lens. Because the front object is not in focus, parts of the focused object that are not visible from the center of the lens will be visible from other parts of the lens. Information about those parts will not available for the postprocessor, so the postprocessor cannot possibly get the correct result.

There is another way to approach the depth of field problem. Depth of field occurs because the lens is a finite size. Each point on the lens "looks" at the same point on the focal plane. The visible surfaces and the shading may be different as seen from different parts of the lens. The depth of field calculations should account for this and be an integral part of the visible surface and shading calculations.

Depth of field can be calculated by starting with the traditional ray from the center of the lens through point p on the focal plane. A point on the surface of the lens is selected and the ray from that point to p is traced. The camera specifications required for this calculation are the focal distance and the diameter of the lens $\frac{F}{n}$, where F is the focal length of the lens and n is the aperture number.

This gives exactly the same circle of confusion as presented by Potmesil[8]. Because it integrates the depth of field calculations with the shading and visible surface calculations, this method gives a more accurate solution to the depth of field problem, with the exception that it does not account for diffraction effects.

Figure 1 shows why this method gives the correct circle of confusion. The lens has a diameter of $\frac{F}{n}$ and is focused at a distance P so that the image plane is at a distance V_P, where

$$V_P = \frac{FP}{P-F} \text{ for } P>F .$$

Points on the plane that is a distance D from the lens will focus at

$$V_D = \frac{FD}{D-F} \text{ for } D>F$$

and have a circle of confusion with diameter C of[8]

$$C = |V_D-V_P|\frac{F}{nV_D}$$

For a point I on the image plane, the rays we trace lie inside the cone whose radius at D is

$$r = \frac{1}{2}\frac{F}{n}\frac{|D-P|}{P}$$

The image plane distance from a point on this cone to a point on the axis of the cone is r multiplied by the magnification of the lens.

$$R = r\left(-\frac{V_P}{D}\right).$$

It is easily shown that

$$R = \frac{C}{2}.$$

Hence any points on the cone have a circle of confusion that just touches the image point I. Points outside the cone do not affect the image point and points inside the cone do.

4. Motion Blur

Distributing the rays or sample points in time solves the motion blur problem. Before we discuss this method and how it works, let us first look in more detail at the motion blur problem and at previous attempts to solve it.

The motion blur method described by Potmesil[9] is not only expensive, it also separates the visible surface calculation from the motion blur calculation. This is acceptable in some situations, but in most cases we cannot just calculate a still frame and blur the result. Some object entirely hidden in the still frame might be uncovered for part of the the time sampled by the blur. If we are to blur an object across a background, we have to know what the background is.

Even if we know what the background is, there are problems. For example, consider a biplane viewed from above, so that the lower wing is completely obscured by the upper wing. Because the upper wing is moving, the scenery below it would be seen through its blur, but unfortunately the lower wing would show through too. The lower wing should be hidden completely because it moves with the the upper wing and is obscured by it over the entire time interval.

This particular problem can be solved by rendering the plane and background as separate elements, but not all pictures can easily be separated into elements. This solution also does not allow for changes in visibility within a single object. This is particularly important for rotating objects.

The situation is further complicated by the change in shading within a frame time. Consider a textured top spinning on a table. If we calculate only one shade per frame, the texture would be blurred properly, but unfortunately the highlights and shadows would be blurred too. On a real top, the highlights and shadows are not blurred at all by the spinning. They are blurred, of course, by any lateral motion of the top along the table or by the motion of a light source or the camera. The highlights should be blurred by the motion of the light and the camera, by the travel of the top along the table, and by the precession of the top, but not by the rotation of the top.

Motion blurred shadows are also important and are not rendered correctly if we calculate only one shade per frame. Otherwise, for example, the blades of a fan could be motion blurred, but the shadows of those blades would strobe.

All of this is simply to emphasize the tremendous complexity of the motion blur problem. The prospects for an analytic solution are dim. Such a solution would require solving the visible surface problem as a function of time as well as space. It would also involve integrating the texture and shading function of the visible surfaces over time. Point sampling seems to be the only approach that offers any promise of solving the motion blur problem.

One point sampling solution was proposed by Korein and Badler[6]. Their method, however, point samples only in space, not in time. Changes in shading are not motion blurred. The method involves keeping a list of all objects that cross each sample point during the frame time, a list that could be quite long for a fast moving complex scene. They also impose the unfortunate restriction that both vertices of an edge must move at the same velocity. This creates holes in objects that change perspective severely during one frame, because the vertices move at drastically different rates. Polygons with edges that share these vertices cannot remain adjoining. The algorithm is also limited to linear motion. If the motion is curved or if the vertices are allowed to move independently, the linear intersection equation becomes a higher order equation. The resulting equation is expensive to solve and has multiple roots.

Distributing the sample points in time solves the motion blur problem. The path of motion can be arbitrarily complex. The only requirement is the ability to calculate the position of the object at a specific time. Changes in visibility and shading are correctly accounted for. Shadows (umbras and penumbras), depth of field, reflections and intersections are all correctly motion blurred. By using different distributions of rays, the motion can be blurred with a box filter or a weighted filter or can be strobed.

This distribution of the sample points in time does not involve adding any more sample points. Updating the object positions for each time is the only extra calculation needed for motion blur. Proper antialiasing is required or the picture will look strobed or have holes[4].

5. Other Implications of the Algorithm

Visible surface calculation is straightforward. Since each ray occurs at a single instant of time, the first step is to update the positions of the objects for that instant of time. The next is to construct a ray from the lens to the sample point and find the closest object that the ray intersects. Care must be taken in bounding moving objects. The bound should depend on time so that the number of potentially visible objects does not grow unacceptably with their speed.

Intersecting surfaces are handled trivially because we never have to calculate the line of intersection; we merely have to determine which is in front at a given location and time. At each sample point only one of the surfaces is visible. The intersections can even be motion blurred, a problem that would be terrifying with an analytic method.

The union, intersection, difference problem is easily solved with ray tracing or point sampling[10]. These calculations are also correctly motion blurred.

Transparency is easy even if the transparency is textured or varies with time. Let τ be the transparency of a surface at the time and location it is pierced by the ray, and let R be the reflectance. R and τ are wavelength dependent, and the color of the transparency is not necessarily the same as the color of the reflected light; for example, a red transparent plastic object may have a white highlight. If there are n-1 transparent surfaces in front of the opaque surface, the light reaching the viewer is

$$R_n\prod_{i=1}^{n-1}\tau_i + R_{n-1}\prod_{i=1}^{n-2}\tau_1 + \cdots + R_2\tau_1 + R_1 = \sum_{i=1}^{n} R_i \prod_{j=1}^{i-1} \tau_j.$$

If the surfaces form solid volumes, then each object has a τ, and that τ is scaled by the distance that the transmitted ray travels through that object. The motion blur and depth of field calculations work correctly for these transparency calculations.

The distributed approach can be adapted to a scanline algorithm as well as to ray tracing. The general motion blur and depth of field calculations have been incorporated into a scanline algorithm using distributed sampling for the visible surface calculations. Special cases of penumbras, fuzzy reflections, and translucency have been successfully incorporated for flat surfaces.

6. Summary of the Algorithm

The intensity of a pixel on the screen is an analytic function that involves several nested integrals: integrals over time, over the pixel region, and over the lens area, as well as an integral of reflectance times illumination over the reflected hemisphere and an integral of transmittance times illumination over the transmitted hemisphere. This integral can be tremendously complicated, but we can point sample the function regardless of how complicated it is. If the function depends on n parameters, the function is sampled in the n dimensions defined by those parameters. Rather than adding more rays for each dimension, the existing rays are distributed in each dimension according to the values of the corresponding parameter.

This summary of the distributed ray tracing algorithm is illustrated in Figure 2 for a single ray.

- Choose a time for the ray and move the objects accordingly. The number of rays at a certain time is proportional to the value of the desired temporal filter at that time.

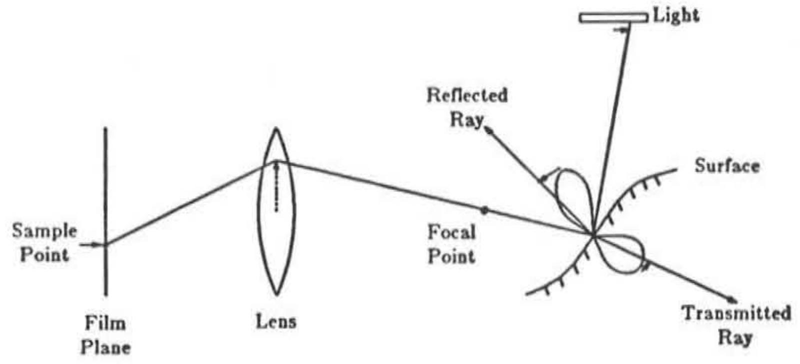


Figure 2. Typical Distributed Ray Path

- Construct a ray from the eye point (center of the lens) to a point on the screen. Choose a location on the lens, and trace a ray from that location to the focal point of the original ray. Determine which object is visible.
- Calculate the shadows. For each light source, choose a location on the light and trace a ray from the visible point to that location. The number of rays traced to a location on the light should be proportional to the intensity and projected area of that location as seen from the surface.
- For reflections, choose a direction around the mirror direction and trace a ray in that direction from the visible point. The number of rays traced in a specific direction should be proportional to the amount of light from that direction that is reflected toward the viewer. This can replace the specular component.
- For transmitted light, choose a direction around the direction of the transmitted light and trace a ray in that direction from the visible point. The number of rays traced in a specific direction should be proportional to the amount of light from that direction that is transmitted toward the viewer.

7. Examples

Figure 3 illustrates motion blurred intersections. The blue beveled cube is stationary, and the green beveled cube is moving in a straight line, perpendicular to one of its faces. Notice that the intersection of the faces is blurred except in in the plane of motion, where it is sharp.

Figures 4 and 5 illustrate depth of field. In figure 4, the camera has a 35 mm lens at f2.8. Notice that the rear sphere, which is out of focus, does not blur over the spheres in front. In figure 5, the camera is focused on the center of the three wooden spheres.

Figure 6 shows a number of moving spheres, with motion blurred shadows and reflections.

Figure 7 illustrates fuzzy shadows and reflections. The paper clip is illuminated by two local light sources which cast shadows with penumbras on the table. Each light is an extended light source (i.e., not a point light source) with a finite solid angle, and the intensity of its shadow at any point on the table is proportional to the amount of light obscured by the paper clip. The table reflects the paper clip, and the reflection blurs according to the specular distribution function of the table top. Note that both the shadows and the reflection blur with distance and are sharper close to the paper clip.

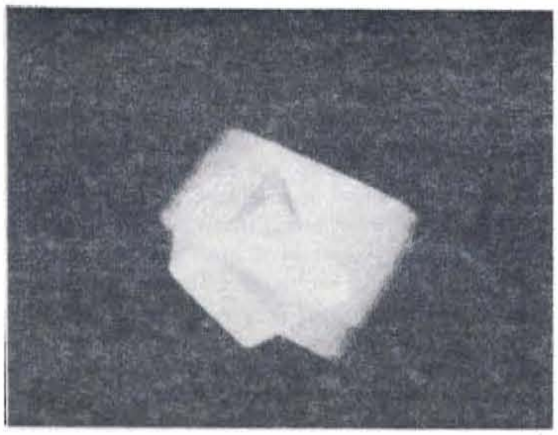


Figure 3. Motion Blurred Intersection.

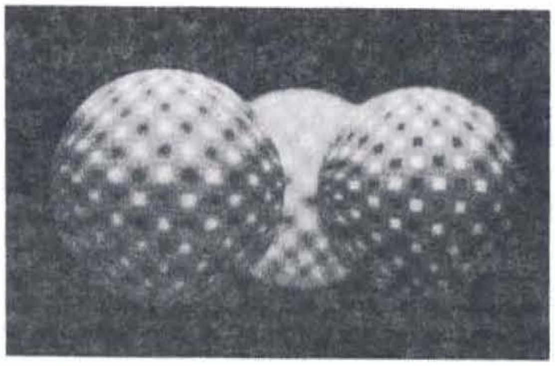


Figure 4. Depth of Field.

Figure 5. Depth of Field.

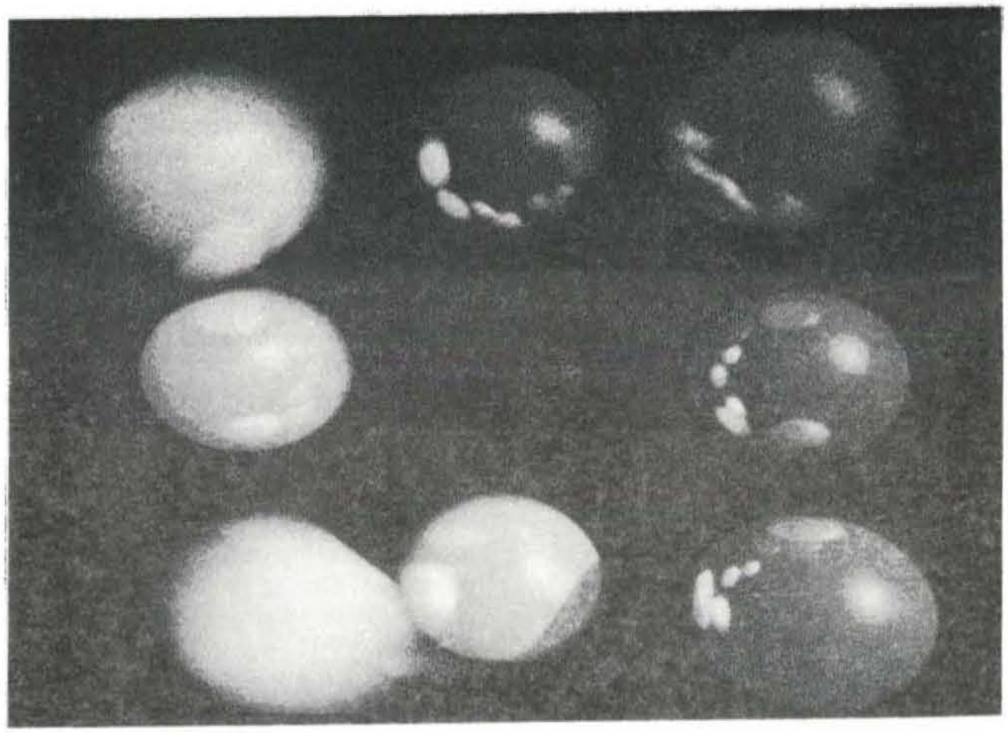


Figure 6. Balls in Motion.

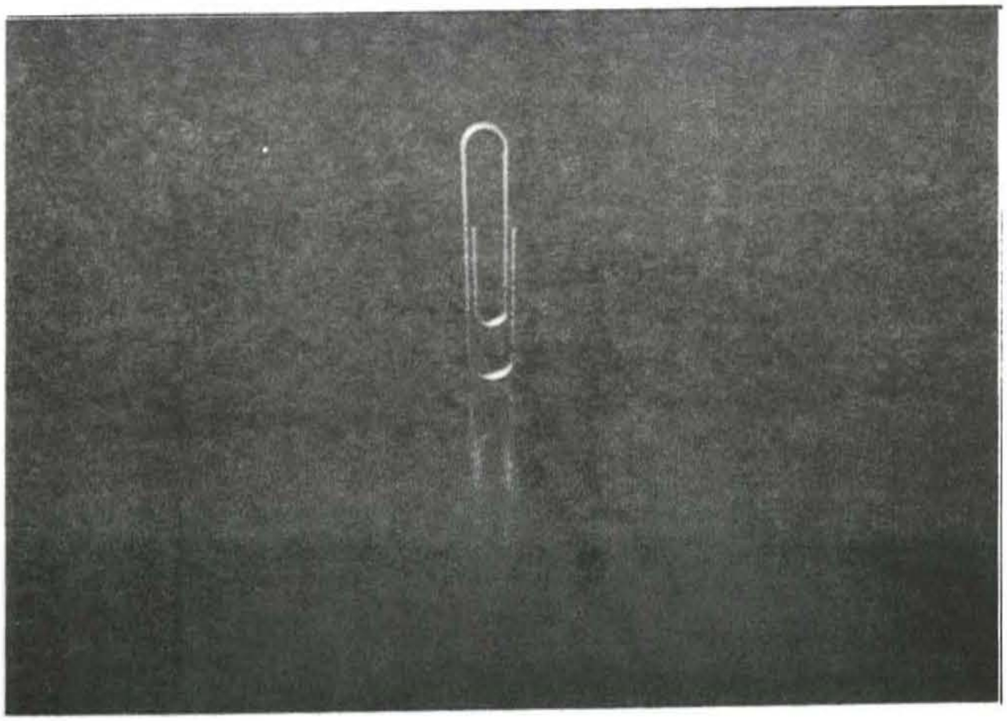


Figure 7. Paper Clip.

Figure 8 shows 5 billiard balls with motion blur and penumbras. Notice that the motion is not linear: the 9 ball changes direction abruptly in the middle of the frame, the 8 ball moves only during the middle of the frame, and the 4 ball only starts to move near the end of the frame. The shadows on the table are sharper where the balls are closer to the table; this most apparent in the stationary 1 ball. The reflections of the billiard balls and the room are motion blurred, as are the penumbras.

Figures 3, 5, and 7 were rendered with a scanline adaptation of this algorithm. Figures 4, 6, and 8 were rendered with ray tracing.

8. Conclusions

Distributed ray tracing a new paradigm for computer graphics which solves a number of hitherto unsolved or partially solved problems. The approach has also been successfully adapted to a scanline algorithm. It incorporates depth of field calculations into the visible surface calculations, eliminating problems in previous methods. It makes possible blurred phenomena such as penumbras, gloss, and translucency. All of the above can be motion blurred by distributing the rays in time.

These are not isolated solutions to isolated problems. This approach to image synthesis is practically no more expensive than standard ray tracing and solves all of these problems at once. The problems could not really be solved separately because they are all interrelated. Differences in shading, in penumbras, and in visibility are accounted for in the depth of field calculations. Changes in the depth of field and in visibility are motion blurred. The penumbra and shading calculations are motion blurred. All of these phenomena are related, and the new approach solves them all together by sampling the multidimensional space they define. The key to this is the ability to antialias point sampling.

9. Acknowledgements

Rodney Stock proposed the approach to antialiased point sampling that formed the basis of the paradigm explored in this paper. John Lasseter drew the environment map of the pool hall for "1984". Ed Catmull worked with us in the image synthesis working group and helped develop and refine these ideas. He and Alvy Ray Smith provided invaluable suggestions along the way. Tom Duff wrote the ray tracing program that we adapted to distributed ray tracing.

References

1. COOK, ROBERT L., TURNER WHITTED, AND DONALD P. GREENBERG, *A Comprehensive Model for Image Synthesis*. unpublished report
2. COOK, ROBERT L., "A Reflection Model for Realistic Image Synthesis," Master's thesis, Cornell University, Ithaca, NY, December 1981.
3. COOK, ROBERT L. AND KENNETH E. TORRANCE, "A Reflection Model for Computer Graphics," *ACM Transactions on Graphics*, vol. 1, no. 1, pp. 7-24, January 1982.
4. COOK, ROBERT L., "Antialiased Point Sampling," Technical Memo #94, Lucasfilm Ltd, San Rafael, CA, October 3, 1983.
5. HUNTER, RICHARD S., *The Measurement of Appearance*, John Wiley & Sons, New York, 1975.
6. KOREIN, JONATHAN AND NORMAN BADLER, "Temporal Anti-Aliasing in Computer Generated Animation," *Computer Graphics*, vol. 17, no. 3, pp. 377-388, July 1983.
7. NISHITA, TOMOYUKI, ISAO OKAMURA, AND EIHACHIRO NAKAMAE, *Siggraph Art Show*, 1982.
8. POTMESIL, MICHAEL AND INDRANIL CHAKRAVARTY, "Synthetic Image Generation with a Lens and Aperture Camera Model," *ACM Transactions on Graphics*, vol. 1, no. 2, pp. 85-108, April 1982.
9. POTMESIL, MICHAEL AND INDRANIL CHAKRAVARTY, "Modeling Motion Blur in Computer-Generated Images," *Computer Graphics*, vol. 17, no. 3, pp. 389-399, July 1983.
10. ROTH, S. D., "Ray Casting for Modeling Solids," *Computer Graphics and Image Processing*, no. 18, pp. 109-144, 1982.
11. WHITTED, TURNER, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, vol. 23, pp. 343-349, 1980.

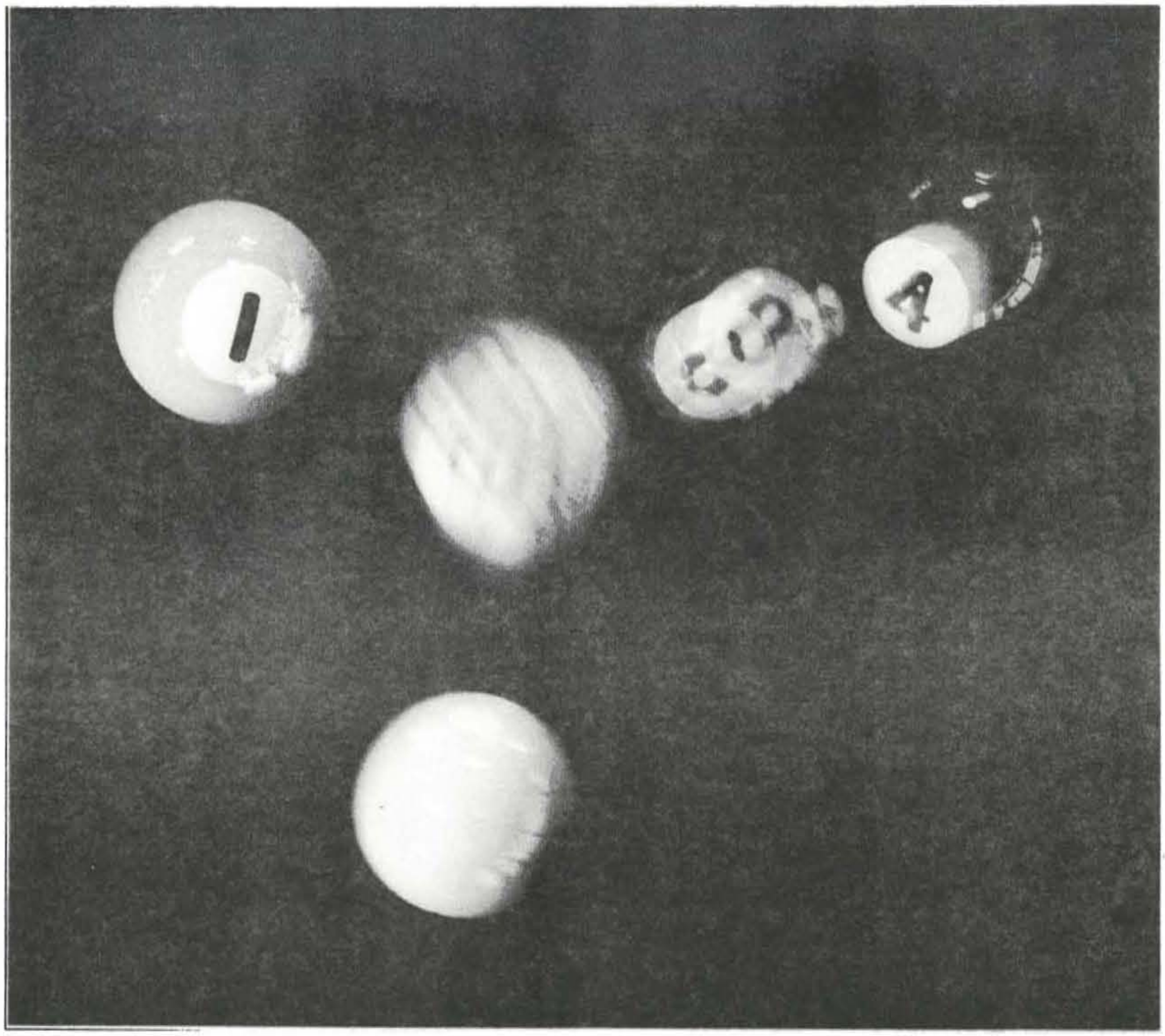


Figure 8. 1984.

State of the Art In Image Synthesis - '85 Course Notes

Ken Perlin

Courant Institute of Mathematical Sciences

New York University

I) Introduction

Computer Graphics is often regarded as a big bag of algorithmic tricks. In order to systematically create useful algorithms we must take a broader view. Many so called "tricks" are actually based on a small and unified set of very simple and elegant algorithmic concepts. We must learn to work with these before we can generalize to new algorithms from existing ones.

"Pre-integration" is such a concept. We will look at several algorithms based on this concept. From these we will create several new algorithms. In each case we will rely on our understanding of the underlying concept. These algorithms should not be taken as definitive in any sense. They were selected on the basis of pedagogical value. We will assume that the student is familiar with elementary calculus and with the basic techniques and terminology of CGI.

Each section ends with a set of exercises. These are not meant to be easy. Much of the "meat" of the course is buried in these exercises. Every exercise has a known solution. Read them carefully; play with them; make up your own.

II) Atmospheric density

We will now introduce the first of our applications.

In order to enhance the sense of realism in a computer generated image, or just to create a mood, it is desirable to simulate atmospheric attenuation.

For any pixel this involves finding the total attenuation along the ray path from the observer to the visible surface. The amount of light reaching us along the ray is then approximated by

$$light_{total} = t\ light_{surface} + (1-t)\ light_{ambient} \qquad 2.1$$

where t is an atmospheric attenuation factor given by

$$t = e^{-I} \qquad 2.2$$

where I is the integral of atmospheric density along the ray path.

If we think of atmospheric density as a function which maps points in space to a density at each point, then to find atmospheric attenuation we must take the line integral of this function along the ray from the observer to the surface.

For general functions over space it is very difficult to compute line integrals. The obvious solution is to integrate numerically: marching along the ray incrementally, evaluating the density at each increment, and summing the resulting densities. This approach is generally very expensive [1].

Gardner [2] gets around the problem by an ingenious trick: he uses infinitely thin semi-transparent films of mist on ellipsoidal forms instead of actual volumes. Unfortunately, the sense of reality tends to disappear when the observer passes through or too near to one of these films.

By restricting the problem domain, we can do better than this. Let us approach the problem in reverse. For what types of atmosphere can we directly compute the line integral I analytically?

The simplest case is an atmosphere of constant density d at all points in space. In this case

$$I = t\,d \tag{2.3}$$

where t is the total path length from observer to surface.

This technique was first used by Turner Whitted [3] and in several scenes produced by MAGI for the motion picture TRON [4].

Now suppose we wish to simulate layers of mist. This effect is observed in nature on foggy mornings when only the air close to the ground condenses.

Mathematically, layered mist is described by a density function over space varying in y, but not in x or z. Thus we say

$$d(x,y,z) = f(y) \tag{2.4}$$

for some function $f(y)$. For convenience, we will assume that the observer is located at the origin (ie. $(x,y,z)_{observer} = (0,0,0)$). Taking arbitrary line integrals of $d(x,y,z)$ is a little tricky. We will approach it in two steps.

First we consider a vertical ray, where the surface point is $(0,y,0)$ for some y. In this case we can simply integrate f:

$$I = F(y) = \int_{t=0}^{t=y} f(t)\,dt \tag{2.5}$$

to find the density integral I along the ray.

Next we consider a general ray where the surface point is at some arbitrary (x,y,z). We observe that the values of f encountered along this ray path are identical with those along the path to (0,y,0). The only difference is that the ray is "stretched" by a factor of $\frac{\sqrt{x^2+y^2+z^2}}{y}$. Thus I will be given by

$$I = \frac{\sqrt{x^2+y^2+z^2}}{y} F(y) \qquad 2.6$$

For completeness, we mention that a perfectly horizontal ray ($y = 0$) will have a constant density along its path, so here we must use the simple constant density model:

$$I = \sqrt{x^2+z^2}\, f(0) \qquad 2.7$$

Notice that we have reduced the problem of finding our density integral to that of finding the integral of a function over one dimension. Given a function f over one dimension, we can generally construct its integral F directly. For example, if we wish to define f as the piece-wise linear function in figure 2.1, then we construct F as the piece-wise parabolic function in figure 2.2.

The above technique for rendering layered mist can be seen in the short film *First Flight* [5]. We note in particular that the non-homogeneity of the atmosphere in this film imparts a strong impression of three dimensionality as the observer flies through the scene.

We must point out that this entire approach represents a vastly simplified model of atmospheric behaviour. For a good survey, discussion and bibligraphy of more expensive but more accurate atmospheric simulation techniques, read [1].

Exercises:

1. Instead of planar layers, create concentric spherical layers with density a function of radius. Read Jim Blinn's "Blobby Molecules" paper [6]. How could you combine his ideas together with the idea of clusters of your variable density spheres to make reasonable cloud formations? Work out the details; make pictures if possible.

2. Blinn uses the Gaussian functions $a^2e^{-r^2/\sigma^2}$, where r is radius. Polynomial functions will also work and are faster. Use these instead. Generalize this in any way that you can think of.

3. Create cylindrical or conical density columns. These are good for simulating spot light cones.

4. Generalize the above problem to allow "cylinders" or "cones" of *arbitrary* cross-section shape. These are good for simulating shafts of light or shadow volumes cast into dusty environments.

5. Create algorithms for very irregular atmospheric shapes. Be imaginative. Try to make directly integrable fractal-like clouds.

In each case the key idea is to construct the density function itself so that integrating its value along any straight line path can be reduced to a one dimensional *solvable* integral. Problem 5 is by far the hardest, but also the most interesting.

Important points to remember:

1. When a completely general solution cannot be found, restrict the problem domain.

2. Whenever possible, reduce the dimensionality of the problem domain.

3. When the integral of a function is needed, have the integral available (pre-computed or else trivially computable) before entering the inner loop of the computation.

III) Discrete Integrals

Now that we have seen one use of integration, let us examine the process of integration more closely.

Consider an array of numbers. We can view this array as a function f over a set of integers. Suppose we want to find the average value of this function over some interval $[a,b]$ of the domain. This is just the discrete integral

$$\frac{1}{b-a}\sum_{i=a}^{i=b} f(i) \tag{3.1}$$

Taking a tip from the atmosphere example, we will precompute the integral function F as follows:

$$F(x) = \sum_{i=0}^{i=x} f(i) \tag{3.2}$$

Then the average value of f within $[a,b]$ is given by

$$\frac{F(b) - F(a)}{b-a} \tag{3.3}$$

We can use this to solve the related problem: "For any x, what is the average value of $f(x)$ from $x-\frac{\delta}{2}$ to $x+\frac{\delta}{2}$?" Adopting the above equation, we obtain:

$$\tilde{f}_\delta(x) = \frac{F(x+\frac{\delta}{2}) - F(x-\frac{\delta}{2})}{\delta} \tag{3.4}$$

If F is precomputed, we may obtain $\tilde{f}_\delta$ for *any* δ at a cost of a few arithmetic operations.

The above idea generalizes nicely to images, where we may want an efficient way to find the average intensity value within some rectangle of the image. For example, suppose we want to map a texture image onto a surface. Sometimes many pixels of the texture image will fall within a single surface pixel. In this case, simply choosing one texture pixel would lead to severe undersampling artifacts. To avoid such artifacts we can use the average value of all texture image pixels that fall within the surface pixel.

Actually summing up all of those texture pixels would be prohibitively slow. Fortunately we can use the $\int$ operation as follows. Think of the image as a function g over a two dimensional domain. We precompute the integral G of this function:

$$G(x,y) = \sum_{i=0}^{i=x}\sum_{j=0}^{j=y} g(x,y) \qquad 3.5$$

where [0,0] is taken as the lower left corner of the image.

Imagine a rectangular area of the image, centered at $[x,y]$, of width α and of height β. Generalizing equation 3.4 to two dimensions, we can define the average intensity in this rectangle by:

$$g_{[\alpha,\beta]}(x,y) = \frac{G(x-\frac{\alpha}{2},y-\frac{\beta}{2}) - G(x-\frac{\alpha}{2},y+\frac{\beta}{2}) - G(x+\frac{\alpha}{2},y-\frac{\beta}{2}) + G(x+\frac{\alpha}{2},y+\frac{\beta}{2})}{\alpha\beta} \qquad 3.6$$

This is essentially the basis of the method of antialiased texture mapping introduced by Frank Crow in [7]. His paper also contains a fascinating discussion of many of the practical issues involved in storing and using integrated images.

Unfortunately, the above approach does not solve all undersampling problems. For example, suppose the texture image consists of a star field. Consider what happens to one moving surface pixel over the course of an animation sequence. At any one frame, some set of stars will lie within the rectangle of the texture image which is mapped onto this surface pixel. Every time a star crosses into or out of this rectangle there will be a visible "pop" up or down in intensity at the surface pixel.

To see why this is so, we again go back to a single dimension. The one dimensional analogue of an image of stars is shown in figure 3.1, where the stars are represented by a function f which is zero almost everywhere, with an occasional value of one. $\ddot{f}$ is shown in figure 3.2. We will assume from now on that δ is fixed at some reasonable value. Note that $\ddot{f}(a)$ is zero, whereas the neighboring $\ddot{f}(a+\epsilon)$ is one.

This is essentially the situation when a star "pops" between neighboring animation frames; a surface pixel moving very slightly can shift either just into or just out of a non-zero region of $\ddot{g}$.

To solve this problem, consider again the one dimensional case. If we apply the $\ddot{f}$ operation to $\ddot{f}$ itself, we obtain $\ddot{\ddot{f}}$ (figure 3.3). Note that the difference between $\ddot{\ddot{f}}(b)$ and $\ddot{\ddot{f}}(b+\epsilon)$ is very small.

$\ddot{\ddot{f}}$ is obtained from $\ddot{f}$ in exactly the same way that $\ddot{f}$ was obtained from f. First we integrate $\mathbf{F}$ to obtain

$$\mathbf{F}(x) = \sum_{i=0}^{i=x} F(i) \qquad 3.7$$

A little algebra then tells us that

$$\ddot{\ddot{f}}_{\delta}(x) = \frac{\mathbf{F}(x-\delta) - 2\,\mathbf{F}(x) + \mathbf{F}(x+\delta)}{\delta^2} \qquad 3.8$$

The analogous equations in two dimensions are

$$\mathbf{G}(x,y) = \sum_{i=0}^{i=x}\sum_{j=0}^{j=y} G(x,y) \qquad 3.9$$

and

$$\ddot{\ddot{g}}_{[\alpha,\beta]}(x,y) = \frac{1}{\alpha^2\beta^2}\begin{pmatrix} \mathbf{G}(x-\alpha,y+\beta) & -\,2\,\mathbf{G}(x,y+\beta) & +\,\mathbf{G}(x+\alpha,y+\beta) \\ -\,2\,\mathbf{G}(x-\alpha,y) & +\,4\,\mathbf{G}(x,y) & -\,2\,\mathbf{G}(x+\alpha,y) \\ +\,\mathbf{G}(x-\alpha,y-\beta) & -\,2\,\mathbf{G}(x,y-\beta) & +\,\mathbf{G}(x-\alpha,y-\beta) \end{pmatrix} \qquad 3.10$$

Exercises:

1. The "$\int$" operation blurs every point out to a rectangle. This process is called "convolution." We say that the function is "convolved" with a rectangular filter. Similarly, "$\iint$" is f convolved with a triangular filter.

   Both the rectangle and the triangle are piece-wise polynomial functions. A function is piece-wise polynomial if we can split the domain up into pieces, and within each piece describe the function as some polynomial. Try to describe both the rectangle filter and the triangle filter in this way.

2. Mathematically we represent the operation "f convolved with g" by $f \Theta g$. It can be shown that $f \Theta g = f' \Theta \int g$. In other words, if we use the integral of one function and the derivative of the other, we will still get the same convolution.

   What does the first derivative of the rectangle filter look like? What does the second derivative of the triangle filter look like? In equation 3.4 we saw that we can blur with a rectangle filter simply by subracting one shifted integral of f from another. In equation 3.8 we blurred with a triangle filter using only three shifted copies of the double integral of f. Why?

3. Try to implement convolution filters having other piece-wise polynomial shapes in both the one dimensional and two dimensional cases. Hint: to convolve f with a piece-wise *nth* degree polynomial filter, you will need to pre-integrate f $n+1$ times in the one dimensional case, and $2n+2$ times in the two dimensional case.

4. Read the section in Frank Crow's paper on numerical accuracy. What are the numerical problems created by the repeated integrations required in problem 3? Work out how many extra bits are demanded by the integration process as a function of image resolution of the filter's polynomial degree.

5. Take Frank Crow's lead and find ways of reducing this number of bits. If you are ambitious, try to work out the optimal encoding scheme for any given filter and resolution. You should be able to do this using only simple calculus. What does such an encoding scheme do to running time efficiency?

6. Read the paper by Feibush et al. on polygon antialiasing [8]. Why is this an example of a pre-integration technique (and what is the domain of integration)? How would you generalize this technique to polygons over which there is a linear intensity variation? Now generalize to polygons over which the intensity is described by *any* polynomial. Analyze the cost of your solution in terms of both running time and memory storage.

IV) Blur

Now that we have developed the underlying concept of pre-integration, we can apply it to problems that might seem quite different from texturing.

For example, we could use the results of the previous section and construct a "blur" program. This program will take as input an image g in scan line order and output a blurred image $\overset{\#}{g}_{[\delta,\delta]}$, also in scan line order.

Notice that δ is constant over the entire image. This lets us implement a blur program without having to store the entire image to be blurred. In order to produce $\overset{\#}{g}_{[\delta,\delta]}(x,y)$ we need only access a small rectangle of pixels. This means that we only need to store a buffer of 2δ scan lines at any one time.

Now we have a blur operation that works fine over most of an image. But there will be problems near the edge of the image (ie. when a pixel is less than δ from some edge of the image). Frank Crow's solution is to replicate the image ad infinitum, so that pixels near the right edge get a little influence from the leftmost pixels, etc.

This is not what we want for an image blurring program. Imagine an image which is very bright near its left edge and very dark near its right edge. To adopt the above "wraparound" procedure would in this case be disastrous.

To demonstrate an alternate solution we once again return to the one dimensional domain. Figure 4.1 shows a function f over a one dimensional "image". Think of the two vertical lines as left and right image boundaries. We will set f to black (intensity = zero) outside of these boundaries.

Figure 4.2 shows $\overset{\#}{f}$. Notice that there is a noticeable dropoff in intensity near the image boundaries. This is called "vignetting". It occurs whenever an image shown against a black background is blurred. This is because some of the black area surrounding the image mixes into the blurred image.

We can completely remove vignetting as follows. We create a completely white (intensity = one) image I (figure 4.3). $\overset{\#}{I}$ (figure 4.4) will then be an accurate measure of how much background is mixed into *any* image blurred by the same δ. Specifically, the intensity at any point will measure the fractional contribution to that point which is *not* background.

Finally, we derive a completely antivignetted, blurred f by dividing $\overset{\#}{f}$ by this image, pixel by pixel (figure 4.5).

$$f_{blur(\delta)} = \frac{\overset{\#}{f}_{[\delta,\delta]}}{\overset{\#}{I}_{[\delta,\delta]}} \qquad 4.1$$

As a practical matter, if we are going to blur many images by the same δ then we only need to compute $\overset{\#}{I}_{[\delta,\delta]}$ once.

Interestingly, the above technique allows us to do an antivignetted blur within a region of any shape. Given a matte image M which has intensity = one in the region to be blurred, and intensity = zero elsewhere, we can blur only within M by

$$f_{blur(M,delta)}(x,y) = \text{if } (M(,y) = 0) \textit{ then } f(x,y) \textit{ else } \frac{\overset{\#}{f}_{[\delta,\delta]}}{\overset{\#}{M}_{[\delta,\delta]}} \qquad 4.2$$

Several years ago we applied this technique in a collaboration between MAGI and Walt Disney Productions in order to create computer assisted "airbrushed" animation [9].

We wanted to combine conventionally cel animated characters with a true three dimensional CGI animated background. It was imperative to give the characters an airbrushed quality, smoothly blending their dark and light shaded areas. This would give them the rounded appearance needed to aesthetically fit into a CGI background. Unfortunately, airbrushing is an incredibly tedious task when applied to each of the twenty four frames required for every second of an animation. We asked the animators at Disney what *wouldn't* be tedious. They said "drawing pencil lines."

So we used the above f_{blur} algorithm. The animator would draw the outlines of animated characters, and draw other lines to delineate internally lighter and darker regions. The line drawings were then digitally scanned and the different regions filled in by a conventional "paint" program, which also produced the needed matte M for each animated character. The algorithm was applied within M for each animated character to blend its various regions smoothly together. Finally we used each character's M to digitally merge it into the background.

Each character was assigned an actual moving location in the CGI environment. This allowed us to vary δ correctly with the character's changing distance from the CGI camera, and also to place the character appropriately in front of or behind the various objects in its environment.

Exercises:

1. It can be shown that for any f and g, $(\int f) \Theta g = \int (f \Theta g)$. This means that for the blur program we can integrate our blurred image *after* doing all our shifts and adds and subracts. Why is this good in terms of numerical accuracy? Try to implement the blur program in this way.

2. We now know that except for differences in numerical accuracy we can freely move the integration step to *anywhere* in the convolution computation. If the image to be blurred is a silhouette image then its derivative is nonzero only at a small number of pixels. In this case we might want to convolve with the derivative of the image, and then throw in an extra integration step at the end. Why would this be faster? Try to implement a specialized silhouette blurring program which works in this way.

3 Come up with other such special cases. What about motion blur?

4 Suppose the matte image M has intermediate values between zero and one (for example, $\frac{1}{2}$). What does this mean? How could you use this extra information to advantage?

5 We have used the concept of "dividing" one image by another, pixel by pixel. Think of other applications in which parallel arithmetic on images is useful. Suppose we have a graphics language in which every image is a variable, with all operations done in parallel on image pixels. What other things could you do with such a language? Read [10] and [11].

6 Several years ago, the Computer Graphics Lab at NYIT developed a technique of casting shadows by doing a visible surface calculation of the scene as viewed from light source l, then projecting this image of l-visible surfaces back onto the scene [12]. Any surface not projected back onto itself was known to be in shadow.

Suppose we adopt this technique in the following manner: We compute the visible surface calculation from l. We call this image L. Then we choose a surface S we wish to illuminate. Next we create a matte image M from L as follows:

$$M(x,y) = \text{if } (L(x,y) = S) \textit{ then } 1 \textit{ else } 0$$

We could now project M back onto surface S to create the cast shadow on S. But we can do better.

Suppose we want a visibly spherical light source (like the Sun). This should produce soft shadows. We can calculate how soft the shadow image should be along the silhouette of M by computing the ray distance from the occluding edge to S at each edge pixel.

How would you use this information, along with what you know about blurring, to produce soft cast shadows? Hint: you will need to do two blur operations.

7 Come up with a very different application for the $\int^{\rightarrow}$ and $\int^{\rightrightarrows}$ operators. For example, combine the techniques of Section II with those of Sections III or IV.

V) Conclusion

You can get a lot of mileage out of a basic concept if you really understand what it means. The better your understanding, the more you can do. The most important thing is to separate out the elegant concept itself from all of the applications using it, with their attendant details.

We have dealt here only with the concept of pre-integration. There are a few other such concepts underlying much of the published work in this field. Some examples are polygons, ray tracing, normal perturbation, and procedurally defined shapes (or "shape-tals" [sic]). Usually once a concept has been itself clearly identified, a great deal of good work in the field has followed.

As a final exercise, pick the algorithmic concept of your choice. Try to reduce it to a simple mathematical formulation. Use this formulation to create as many different algorithms as possible.

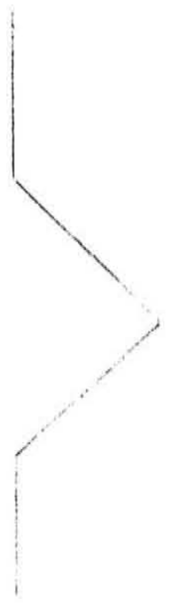


figure 2.1

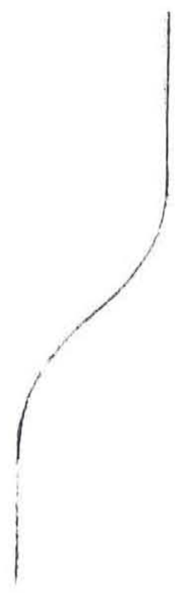


figure 2.2

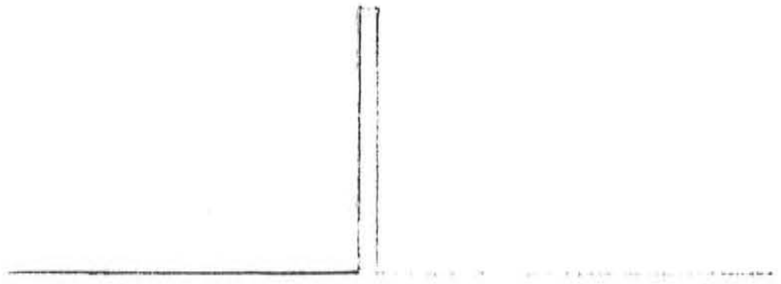


figure 3.1

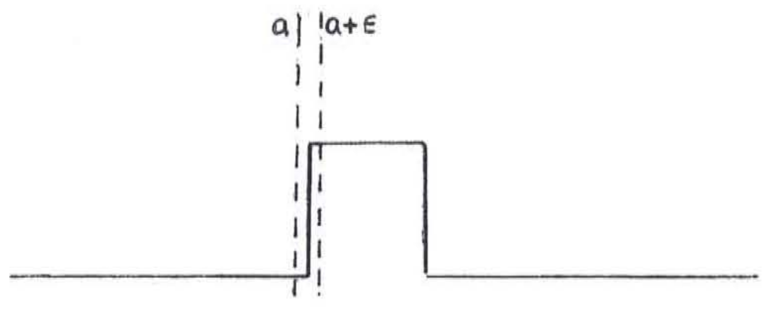


figure 3.2

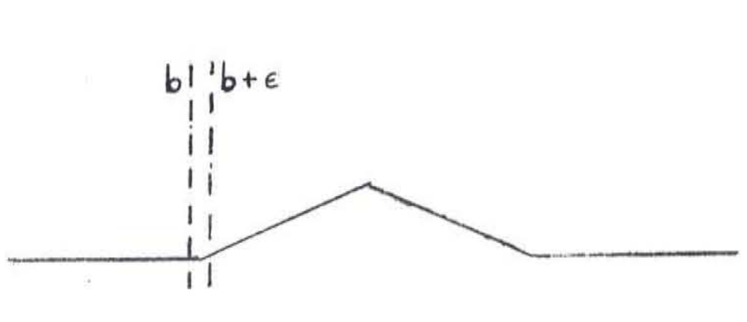


figure 3.3

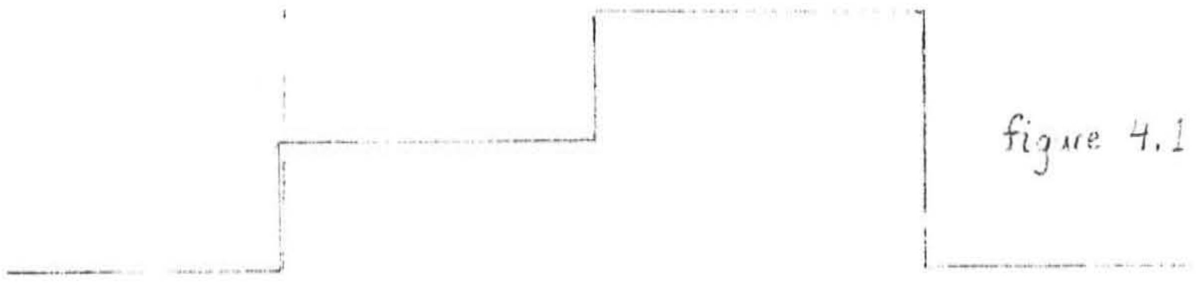


figure 4.1

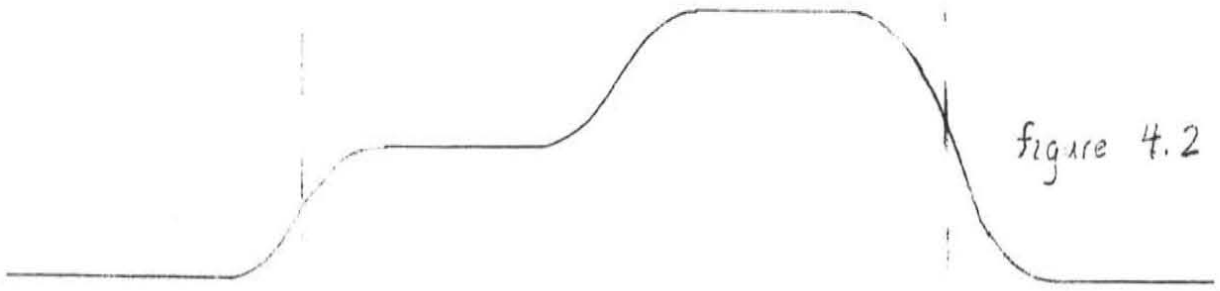


figure 4.2

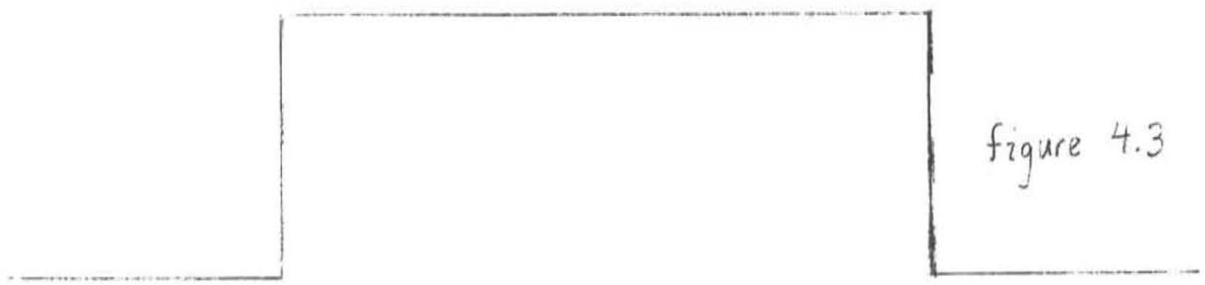


figure 4.3

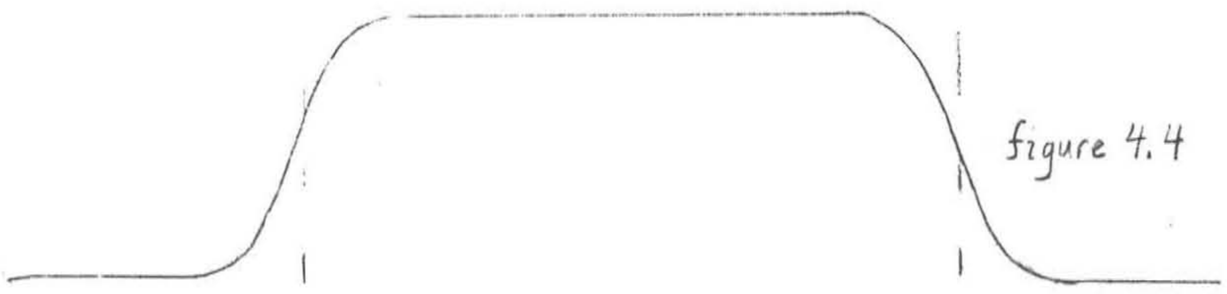


figure 4.4

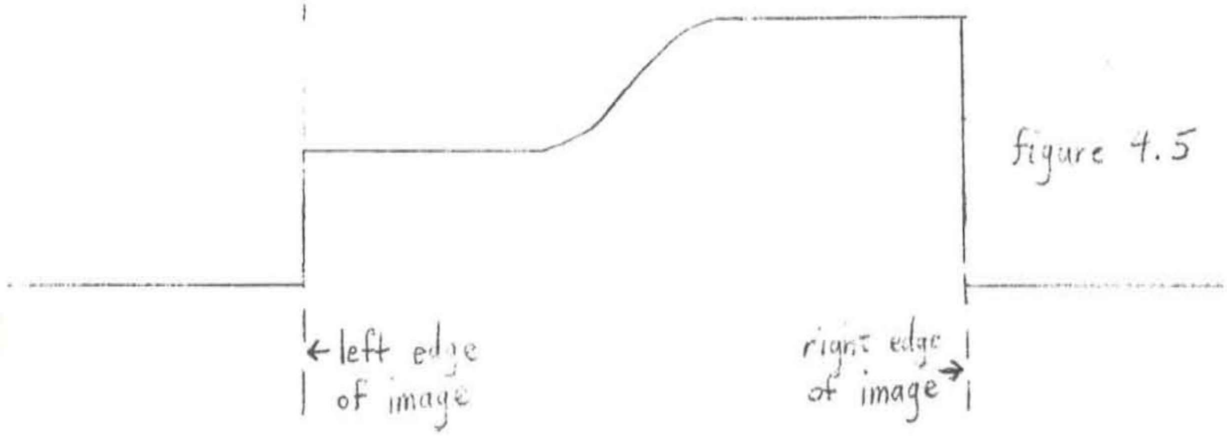


figure 4.5

ACM Siggraph 1985 Volume 11
Conference Proceedings

DATE	ISSUED TO
11/19	M Peraci
GAYLORD 40	