Computer Graphics

Sources

- Fundamentals of Computer Graphics Fourth Edition. Tiger Book
- An Integrated Introduction to Computer Graphics and Geometric Modeling

Mass-Points

- mass-point: a point P in affine space together with a nonzero mass m.
- denote a mass-point by the pair (mP, m). Here m is the mass and $P = mP = m$ is the point; mP by itself has no meaning.
- Vectors reside in this space as objects with zero mass, we write $(v, 0)$.
- Sum of two points: sum of mass located at center of mass(!not 3 vals).

$$(m_1P_1, m_1) + (m_2P_2, m_2) = (m_1P_1 + m_2P_2, m_1 + m_2)$$

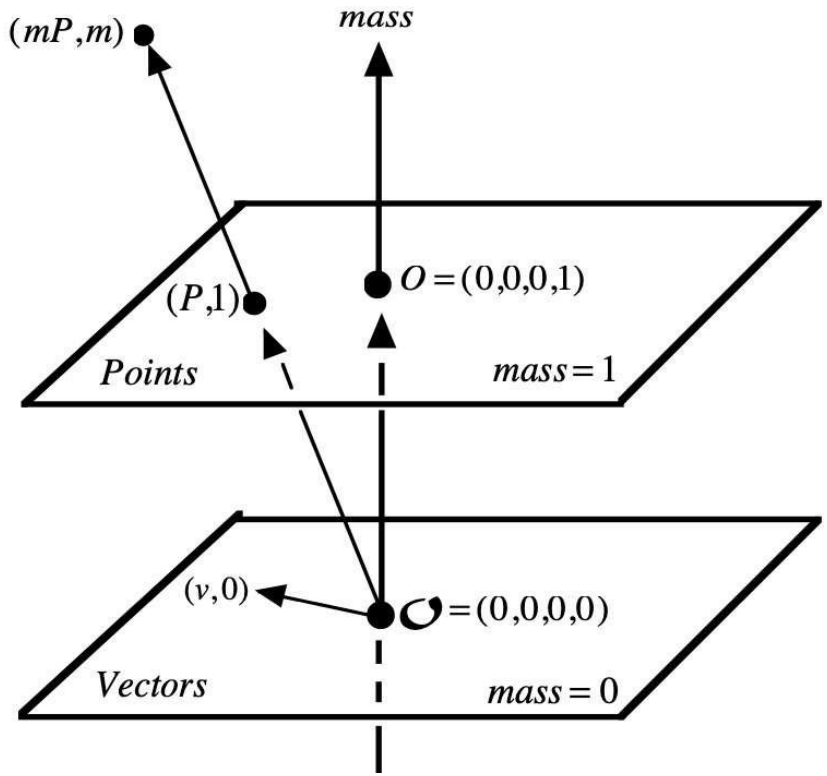
Quaternions

- Division algebras: 1, 2, 4. (8)

  -> line(real), plane(complex), space of mass-points(quaternions)

- denote this special mass-point (identity for quaternion multiplication) by *O,* identify this mass-point with the point located at the origin, in the 3-dimensional affine space of affine points. Vectors lie in the quaternion subspace orthogonal to *O*.

$$(mP,m) = mO + v\,.$$



Interpreted in terms of coordinates, Equation (3.1) means that every quaternion q can be written as $q = (q_1, q_2, q_3, q_4)$ or equivalently

$$q = q_4 O + q_1 i + q_2 j + q_3 k .$$

In many texts, the letter O is dropped, and quaternions are written as

$$q = q_4 + q_1 i + q_2 j + q_3 k .$$

Quaternion multiplication is an extension of complex multiplication. To extend complex multiplication to the quaternions, we need only extend complex multiplication to the 4-dimensional basis O, i, j, k. The point O is the identity for quaternion multiplication, so

$$O^2 = O. \qquad Oi = i = iO \qquad Oj = j = jO \qquad Ok = k = kO \tag{3.4}$$

Multiplication on the basis vectors i, j, k is defined by the following rules:

$$i^2 = j^2 = k^2 = -O \quad \text{and} \quad ij = k = -ji, \; jk = i = -kj, \; ki = j = -ik . \tag{3.5}$$

$$uv = (u_1 i + u_2 j + u_3 k)(v_1 i + v_2 j + v_3 k)$$
$$= u_1 i(v_1 i + v_2 j + v_3 k) + u_2 j(v_1 i + v_2 j + v_3 k) + u_3 k(v_1 i + v_2 j + v_3 k)$$
$$= -(u_1 v_2 + u_2 v_2 + u_3 v_3)O + (u_1 v_2 - u_2 v_1)k + (u_3 v_1 - u_1 v_3)j + (u_2 v_3 - u_3 v_3)i$$

or equivalently

$$uv = -(u \cdot v)O + u \times v \tag{3.6}$$

More generally for two arbitrary quaternions $aO + u$ and $bO + v$

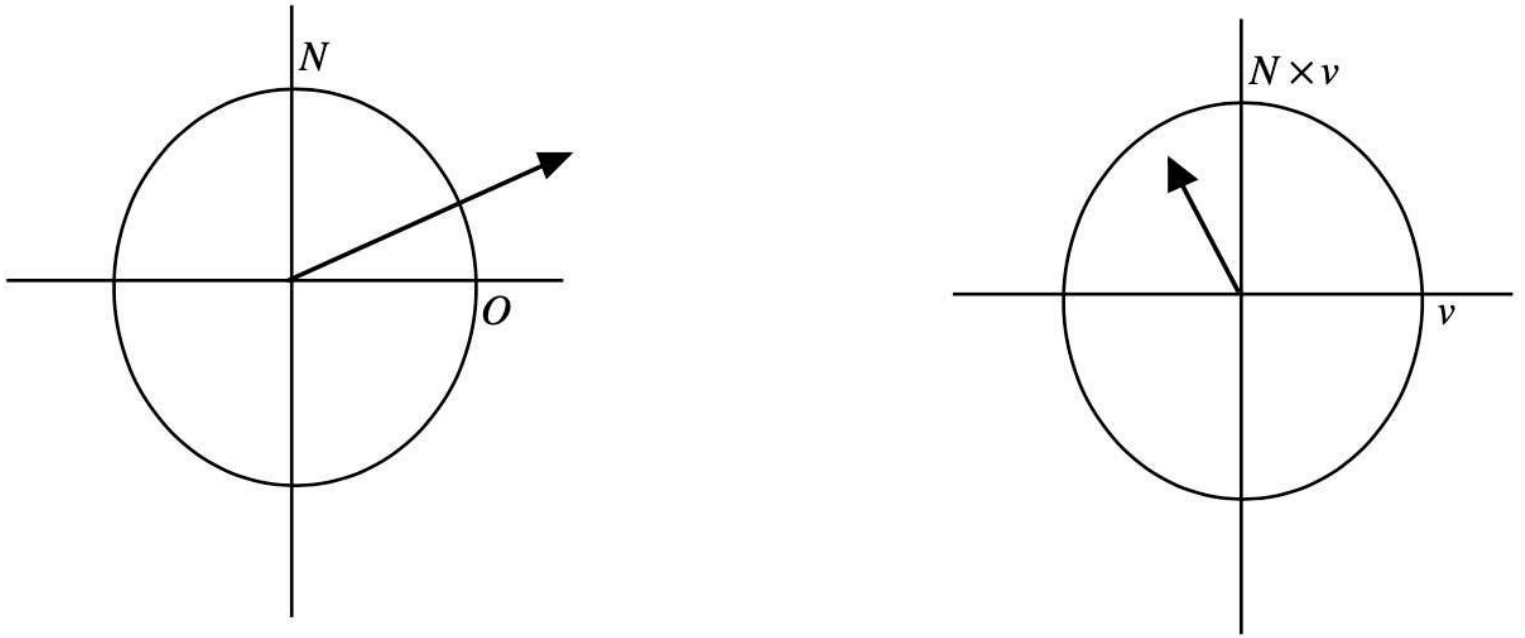
: $(aO + u)(bO + v) = aO(bO + v) + u(bO + v) = abO + av + bu + uv$

so substituting the right hand side of Equation (3.6) for uv , we find that

$$(aO + u)(bO + v) = (ab - u \cdot v)O + (av + bu + u \times v) \tag{3.7}$$

(3.7) general form, most useful!
Neither commutative nor anticommutative

Mutually Orthogonal Planes in 4-D

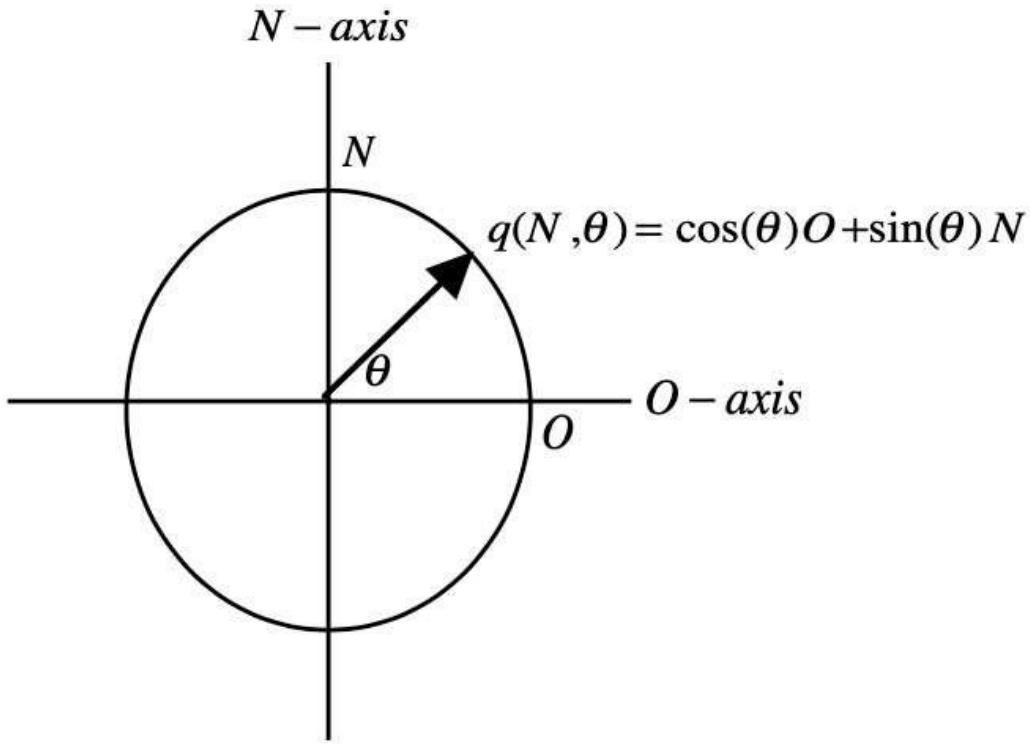


(a) the plane of O, N *(b) the plane perpendicular to O, N*

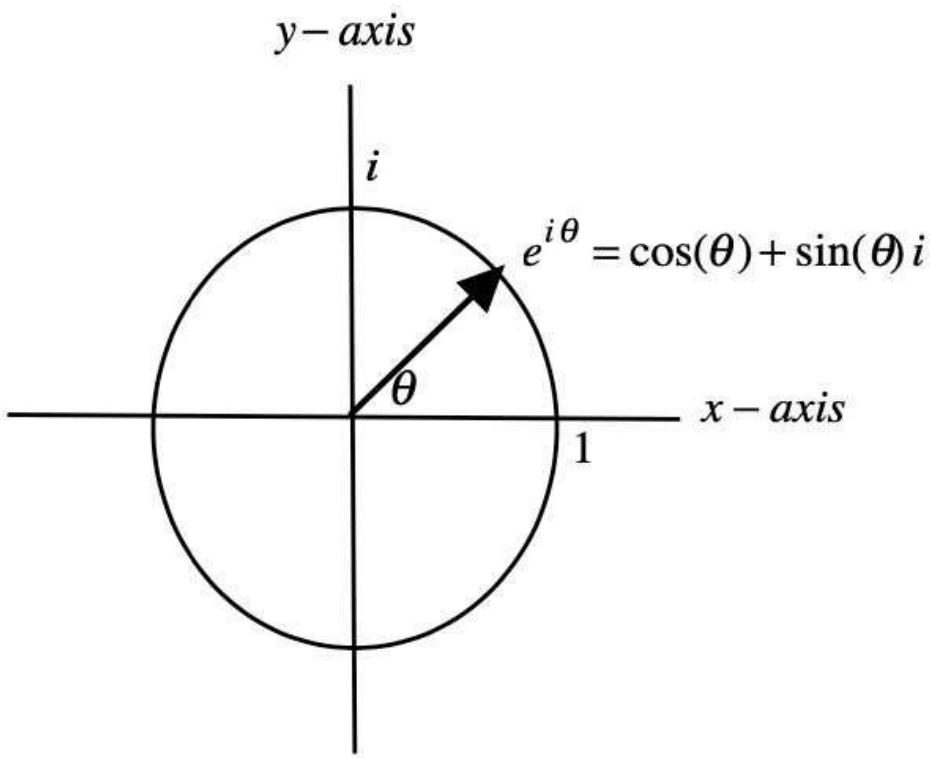
Figure 3: A mass-point represented by its projections (arrows) into two mutually orthogonal planes in 4-dimensions.

Just think of *O, i, j, k*, (a) complex plane. line in 3d (b) orthogonal plane. vec in 3d (maybe actually just multiply by v.

Geometry of Quaternion Multiplication



(a) the quaternion plane of O, N



(b) the complex plane

Rotation in O, N

Let

- N = *a unit vector*
- p = *a quaternion in the plane of O, N*

Then

i. $L_{q(N,\theta)}(p) = q(N,\theta)\, p$ *rotates p by the angle* θ *in the plane of O, N*

ii. $R_{q(N,\theta)}(p) = p\, q(N,\theta)$ *rotates p by the angle* θ *in the plane of O, N*

iii. $R_{q^*(N,\theta)}p = p\, q^*(N,\theta)$ *rotates p by the angle* $-\theta$ *in the plane of O, N*

iv. $S_{q(N,\theta)}(p) = q(N,\theta)\, p\, q^*(N,\theta)$ *is the identity on p.*

Rotation in O, N perpendicular

Let

- N = *a unit vector*
- v = *a vector in the plane* $\perp$ O, N

Then

i. $L_{q(N,\theta)}(v) = q(N,\theta)\, v$ *rotates* v *by the angle* θ *in the plane* $\perp$ O, N

ii. $R_{q(N,\theta)}(v) = v\, q(N,\theta)$ *rotates* v *by the angle* $-\theta$ *in the plane* $\perp$ O, N

iii. $R_{q^*(N,\theta)}(v) = v\, q^*(N,\theta)$ *rotates* v *by the angle* θ *in the plane* $\perp$ O, N

iv. $S_{q(N,\theta)}(v) = q(N,\theta)\, v\, q^*(N,\theta)$ *rotates* v *by the angle* 2θ *in the plane* $\perp$ O, N.

Rotation with Sandwich

Proposition 3.4: *Let* $q(N,\phi) = \cos(\phi/2)O + \sin(\phi/2)N$. *Then a vector v can be rotated around the axis N through the angle* ϕ *by sandwiching v between* $q(N,\phi)$ *and* $q(N,\phi)^*$.

Proof: Let $v = v_{||} + v_{\perp}$, where

- $v_{||} =$ component of v parallel to N
- $v_{\perp} =$ component of v perpendicular to N.

Then

- $S_{q(N,\theta/2)}(v_{||}) = v_{||}$ (Proposition 3.2)
- $S_{q(N\theta/2)}(v_{\perp})$ *rotates* $v_{\perp}$ *by the angle* θ *in the plane* $\perp$ *to* O, N. (Proposition 3.3)

Therefore $S_{q(N,\theta/2)}(v)$ has the same effect on the components of v as rotating v in 3-dimensions by the angle θ around the axis vector N (see Chapter 12, Section 12.3.2).

Mirror with Sandwich

Thus to rotate v by 180^0 around N, we can sandwich v between N and N^*. Since for vectors $N^* = -N$, it follows that mirror image of v in the plane perpendicular to N is given by

$$-S_N(v) = NvN.$$

Thus we are led to the following result.

Proposition 3.5: *A vector v can be mirrored in the plane perpendicular to the unit normal N by sandwiching v between two copies of N.*

Proof: Let $v = v_{||} + v_{\perp}$, where

- $v_{||} =$ component of v parallel to N
- $v_{\perp} =$ component of v perpendicular to N.

Since $N = \cos(\pi/2)O + \sin(\pi/2)N = q(N, \pi/2)$:

- $-S_N(v_{||}) = -S_{q(N,\pi/2)}(v_{||}) = -v_{||}$ (Proposition 3.2)
- $-S_N(v_{\perp}) = -S_{q(N,\pi/2)}(v_{\perp}) = v_{\perp}$ (Proposition 3.3)

Other conformal transformations

Proposition 3.6: *A vector v can be scaled by the factor c by sandwiching v between two copies of* $\sqrt{c}\,O$.

Proof: $S_{\sqrt{c}\,O}(v) = \sqrt{c}\,O\,v\,\sqrt{c}\,O = c\,v$.

Theorem 3.1: *Every conformal transformation of vectors in 3-dimensions can be modeled by sandwiching with a single quaternion.*

Proof: This result follows from Propositions 3.4-3.6, since by Equation (3.12) composition of sandwiching is equivalent to multiplication of quaternions.

transform with some appropriate quaternion (see Exercises 11 and 15). Recall, however, that rotation, mirror image, and uniform scaling, all have a fixed point Q of the transformation. Since $P = Q + (P - Q)$, we can compute conformal transformations on affine points P by using quaternions to transform the vectors $P - Q$ and then adding the resulting vectors to Q. In this way, quaternions can be applied to compute conformal transformations on points as well as on vectors.

Quaternion v.s. Matrix

	memory	*transformation*	*composition*
quaternions	4 scalars	28 multiplies	16 multiplies
3×3 matrices	9 scalars	9 multiplies	27 multiplies

Table 1: Tradeoffs between speed and memory for quaternions and 3×3 matrices.

Avoiding Distortion

To compose conformal transformations, we multiply the corresponding quaternions. Since every quaternion represents a conformal transformation, computing transformations by sandwiching with quaternions will never distort angles.

we can normalize the resulting quaternion q simply by dividing by its length $| q |$.

Key Frame Animation

- Can be used to interpolate in between frames.

$$q(t) = slerp(q_0, q_1, t) = \frac{\sin((1-t)\phi)}{\sin(\phi)} q_0 + \frac{\sin(t\phi)}{\sin(\phi)} q_1 ,$$

where ϕ is the angle between q_0 and q_1. Recall that SLERP guarantees that if q_0 and q_1 are unit quaternions, then $q(t)$ is also a unit quaternion, and if ϕ is the angle between q_0 and q_1, then $t\phi$ is the angle between q_0 and $q(t)$. Thus spherical linear interpolation applied to quaternions generates the appropriate intermediate rotations for key frame animation.

Conversion Formulas

$$Rot(N,\phi) = (\cos\phi)\, I + (1-\cos\phi)(N^T * N) + (\sin\phi)(N \times \_)$$

$$q(N,\phi/2) = \cos(\phi/2)O + \sin(\phi/2)N\ .$$

=> Skip derivation using trigonometry

$$q(N,\phi/2) = \frac{\sqrt{R_{1,1}+R_{2,2}+R_{3,3}+1}}{2} O + \frac{(R_{2,3}-R_{3,2})i + (R_{3,1}-R_{1,3})j + (R_{1,2}-R_{2,1})k}{2\sqrt{R_{1,1}+R_{2,2}+R_{3,3}+1}}.$$

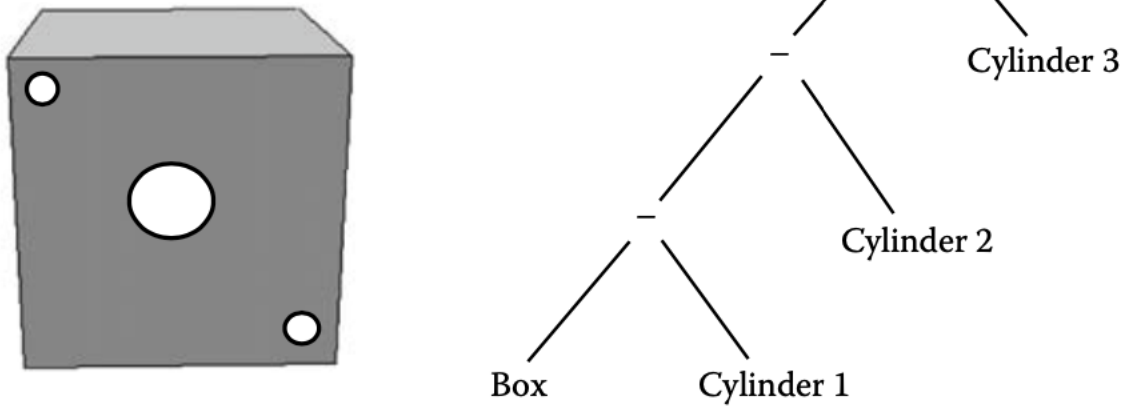
$$Rot(N,\phi) = \begin{pmatrix} q_4^2+q_1^2-q_2^2-q_3^2 & 2q_1q_2+2q_3q_4 & 2q_1q_3-2q_2q_4 \\ 2q_1q_2-2q_3q_4 & q_4^2-q_1^2+q_2^2-q_3^2 & 2q_2q_3+2q_1q_4 \\ 2q_1q_3+2q_2q_4 & 2q_2q_3-2q_1q_4 & q_4^2-q_1^2-q_2^2+q_3^2 \end{pmatrix}.$$

Solid Modeling

- Solid: three-dimensional shape with two well-defined sides.
  - Inside and outside
  - Three types of techniques
    - Construtive solid geometry (CSG)
      - Rely on ray tracing
    - Boundary representations (B-Reps)
      - Parametric patches
    - Octrees
      - Efficient spatial enumeration technique

CSG

Solids are stored in binary trees called CSG trees. The leaves of a CSG tree store primitive solids; the interior nodes store either boolean operations or nonsingular transformations.



B-Rep

A boundary representation for a solid stores the **topology** as well as the geometry of the solid. Geometry models position, size, and orientation; topology models *adjacency* and *connectivity*.

Topology and geometry are tied together by pointers: *v*ertices point to points, *e*dges to curves, *f*aces to surfaces. The data structure that encompasses the geometry, the topology, and the pointers from the topology to the geometry is called a boundary file representation (B-Rep) of a solid.

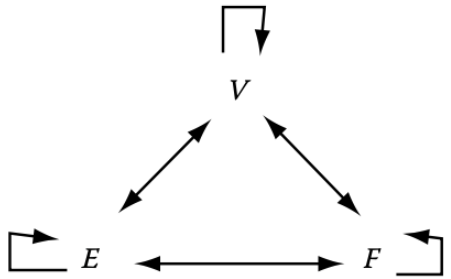
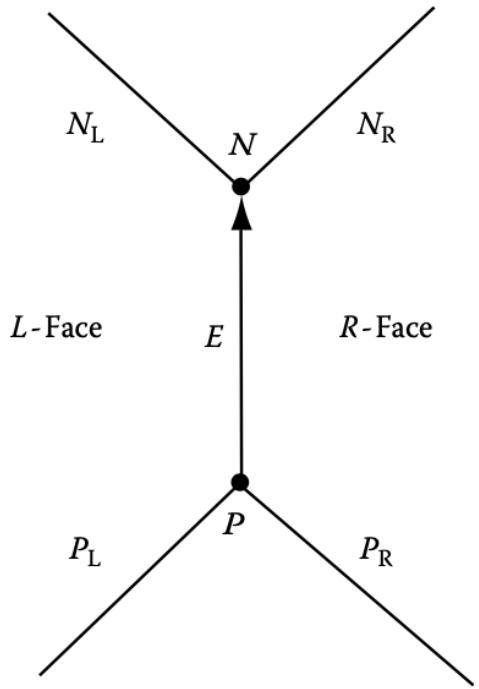


FIGURE 20.4: The nine potential sets of pointers for a topological data structure with vertices V, edges E, and faces F.



Time-space

FIGURE 20.5: The winged-edge data structure. Every edge E points to two vertices (P and N), four edges (P_L, P_R, N_L, N_R), and two faces (L-Face and R-Face).

Euler's formula

Euler's formula for solids bounded by two-dimensional manifolds asserts that

$$V - E + F - H = 2(C - G), \tag{20.1}$$

where

V = the number of vertices
E = the number of edges
F = the number of faces
H = the number of holes in faces
C = the number of connected components
G = the number of holes in the solid (genus)

For example, for a cube, $V = 8$, $E = 12$, $F = 6$, $H = 0$, $C = 1$, and $G = 0$. Therefore,

$$V - E + F - H = 2 = 2(C - G).$$

Understand G: leak water from many holes.

Shading

Purpose: smooth the appearance of polygonal models by reducing the impact of sharp edges in order to give the eye the impression of a smooth curved surface.

- Uniform shading: fast but no attempt to smooth
- Gouraud shading: smooth by linearly interpolating intensities along scanline
- Phong shading: simulate by linearly interpolating normal vectors along scanline

- Ambient reflection
- Diffuse reflection
- Specular reflection

Newell's formula

For the normal to a polygon.

Naively, we could compute the normal vector N to a polygon from its vertices $P_0, \ldots, P_n$ by selecting two adjacent edge vectors $P_{i+1} - P_i, P_{i+2} - P_{i+1}$ and setting

$$N = (P_{i+1} - P_i) \times (P_{i+2} - P_{i+1}),$$

Problem: too close or even collinear, thus unstable

Newell's Formula

$$N = \sum_{k=0}^{n} P_k \times P_{k+1} \quad \{P_{n+1} = P_0\}.$$

$$|N| = 2 \times Area(Polygon).$$

Unifrom shading

$$I_{uniform} = \underbrace{I_a k_a}_{\text{ambient}} + \underbrace{I_p k_d (L \cdot N)}_{\text{diffuse}} + \underbrace{I_p k_s (R \cdot V)^n}_{\text{specular}},$$

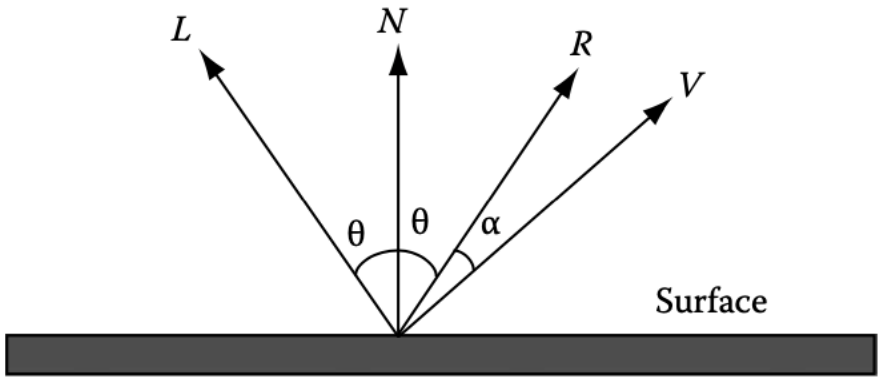
where

N is the unit vector normal to the polygon

L is the unit vector pointing from the polygon to the light source

V is the unit vector pointing from the polygon to the eye

$R = 2(L \cdot N)N - L$ is the direction of the light vector L after it bounces off the polygon



Discontinuities in intensity along polygon edges are called Mach bands.

The vectors N, L, R, V are constant along each polygonal surface.

Gouraud Shading

Linear interpolation of intensities.

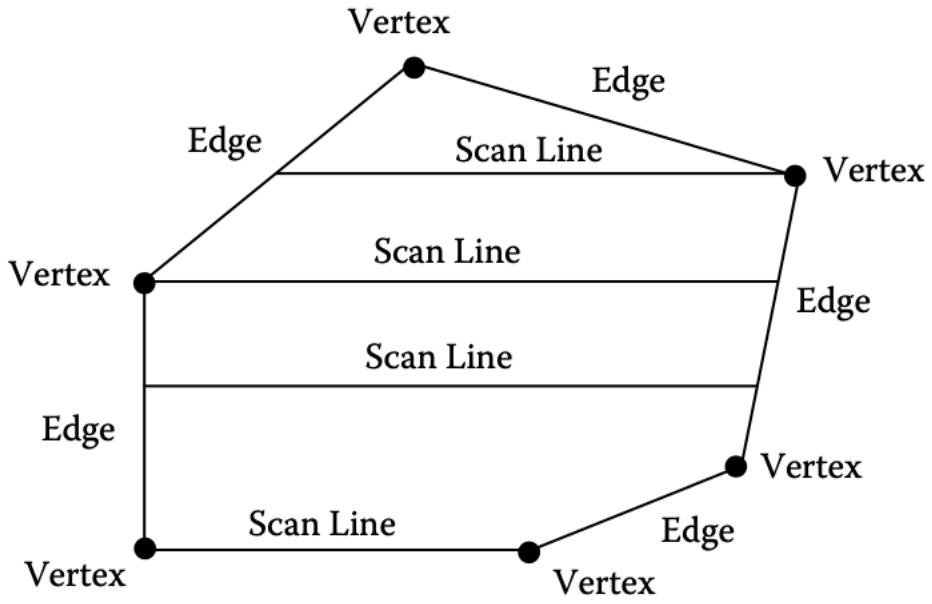


FIGURE 21.2: Gouraud shading. First compute the intensity at each vertex. Then interpolate these vertex intensities along each edge. Finally interpolate these edge intensities to the pixels in the interior of the polygon by interpolating edge intensities along scan lines.

To compute the intensity at a vertex, we need to know the unit normal vector at the vertex. Since each vertex may belong to many polygons, we first use Newell's formula (Equation 21.1) to calculate the unit normal for each polygon containing the vertex. We then calculate the unit normal vector at the vertex by averaging the unit normals of all the polygons containing the vertex:

$$N_{vertex} = \frac{\Sigma_{vertex \in polygon} N_{polygon}}{\left|\Sigma_{vertex \in polygon} N_{polygon}\right|}. \tag{21.3}$$

$$I_{uniform} = \underbrace{I_a k_a}_{\text{ambient}} + \underbrace{I_p k_d (L \cdot N)}_{\text{diffuse}} + \underbrace{I_p k_s (R \cdot V)^n}_{\text{specular}},$$

Phong Shading

Simulates curved surfaces by interpolation n instead of i

1. First, calculate the normals at the individual vertices of the polygon.
2. Next, interpolate these normals along the edges of the polygon.
3. Finally, interpolate these edge normals along scan lines to the pixels in the interior of the polygon.

Hidden Surface Algo.

Image Space Algorithm

For each pixel:
Find the closest polygon.
Render the pixel with the color.
and intensity of this polygon.

Object Space Algorithm

For each polygon:
Find the unobstructed part of the polygon.
Render this part of the polygon.

The speed of image space algorithms is $O(nN)$, where N is the number of pixels and n is number of polygons, because for each pixel we must examine every polygon. The speed of object space algorithms is $O(n^2)$, where n is the number of polygons, because to determine which part of each polygon is visible, we need to compare every polygon with every other polygon.

z-buffer, scan line

depth sort, binary space partitioning tree (bsp-tree)

Ray casting can work both in image space and in object space

z-Buffer (Depth Buffer)

The z-buffer is a large memory array which is the same size as the frame buffer—that is, the z-buffer stores an entry for each pixel. This entry consists of the current depth as well as the current color or intensity of the corresponding pixel.

To compute the depth of each pixel incrementally, we proceed almost exactly as we did in Chapter 21 for Gouraud and Phong shading, where we computed the intensities and normal vectors at the pixels of a polygon incrementally.

1. First, compute the depth at the vertices of the polygon. After pseudoperspective, this depth is typically just the value of the z-coordinate at each vertex.
2. Next, compute the depth along the edges of the polygon.
3. Finally, compute the depth along scan lines for the pixels in the interior of the polygon.

Scan Line

To decide which polygon to paint for each pixel, the scan line algorithm for hidden surfaces maintains two special data structures: an *active edge list* and an *edge table*.

Ray Casting

Ray Casting Algorithm

Through each pixel:

- Fire a ray from the eye.
- Intersect this ray with the plane of each polygon.
- Reject intersections that lie outside the polygon.
- Accept the closest remaining intersection—that is, the intersection with the smallest value of the parameter along the line.

Radiosity

$$Radiosity = Emitted\ Energy + Reflected\ Energy.$$

23.2.1 The Rendering Equation

Energy conservation for light is equivalent to

$$Total\ Illumination = Emitted\ Energy + Reflected\ Energy.$$

We can rewrite these innocent looking words as an integral equation called the *Rendering Equation*.

Rendering Equation

$$I(x, x') = E(x, x') + \int_S \rho(x, x', x'') I(x', x'') dx'', \tag{23.1}$$

where

$I(x', x'')$ is the total energy passing from x'' to x'.
$E(x, x')$ is the energy emitted directly from x' to x.
$\rho(x, x', x'')$ is the reflection coefficient—the percentage of the energy transferred from x'' to x' that is passed on to x.

Continuous form of the Radiosity Equation is just the Rendering Equation restricted to diffuse reflections. -> be more specific about the form of the reflected energy.

Radiosity Equation—Continuous Form

$$B(x) = E(x) + \rho_d(x) \int_S B(y) \frac{\cos\theta \cos\theta'}{\pi r^2(x,y)} V(x,y) \mathrm{d}y, \tag{23.2}$$

where

$B(x)$ is the radiosity at the point x, which we identify with the intensity or energy reflecting off a surface in any direction—that is, the total power leaving a surface/unit area/solid angle. This energy is uniform in all directions, since we are assuming that the scene has only diffuse reflectors.

$E(x)$ is the energy emitted directly from a point x in any direction. This energy is uniform in all directions, since we are assuming that the scene has only diffuse emitters.

$\rho_d(x)$ is the diffuse reflection coefficient—the percentage of energy reflected in all directions from the surface at a point x. By definition, $0 \leq \rho_d(x) \geq 1$.

$V(x, y)$ is the visibility term:

$V(x, y) = 0$ if x is not visible from y

$V(x, y) = 1$ if x is visible from y.

θ is the angle between the surface normal (N) at x and the light ray (L) from x to y.

θ' is the angle between surface normal (N') at y and the light ray (L) from y to x.

$r(x, y)$ is the distance from x to y.