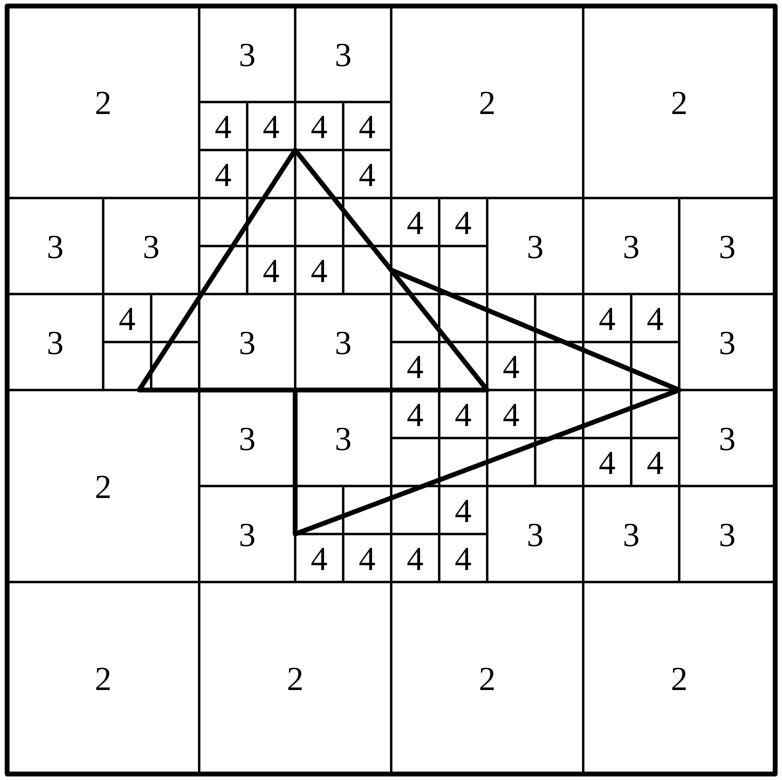
General Methods for 3D Rendering

These methods can handle arbitrary 3D polyhedral scenes—even when the polyhedra penetrate one another.

- **Warnock's Screen Subdivision Method**
- **Depth Buffering (Z-buffering)**

Warnock's Algorithm

Recursively subdivide the screen until only one triangle is visible inside each rectangle.



Warnock PseudoCode

```
void warnock(rectangle r)
{
  if r is a pixel
      if r covers no triangles, color with background color
      else color according to nearest triangle
  if r is a non-pixel rectangle
      if r contains only background, color with background color
      if r is completely contained in a triangle
          and that triangle is nearer than any other overlapping triangles
          then color according to the nearest triangle.
  else
     subdivide into four rectangles: r1, r2, r3, r4
     warnock(r1); warnock(r2); warnock(r3); warnock(r4);
}
```

Two Main Problems

1. **Testing for overlap and/or containment between rectangle and triangles**

   (a) Intersection of bounding box of triangle with rectangle.

   (b) Intersection of triangle edges with rectangle.

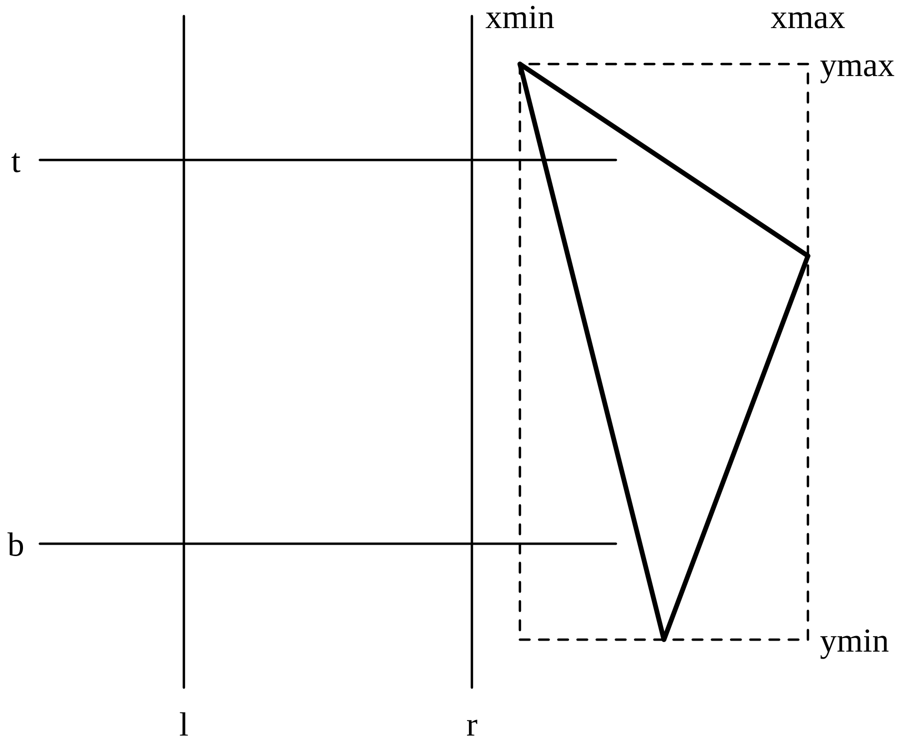
   (c) Containment of rectangle inside triangle or vice versa.

2. **Determining the nearest enclosing triangle**

   (a) Vertex test.

It is ok to miss some termination cases—if they can be caught during subsequent subdivisions.

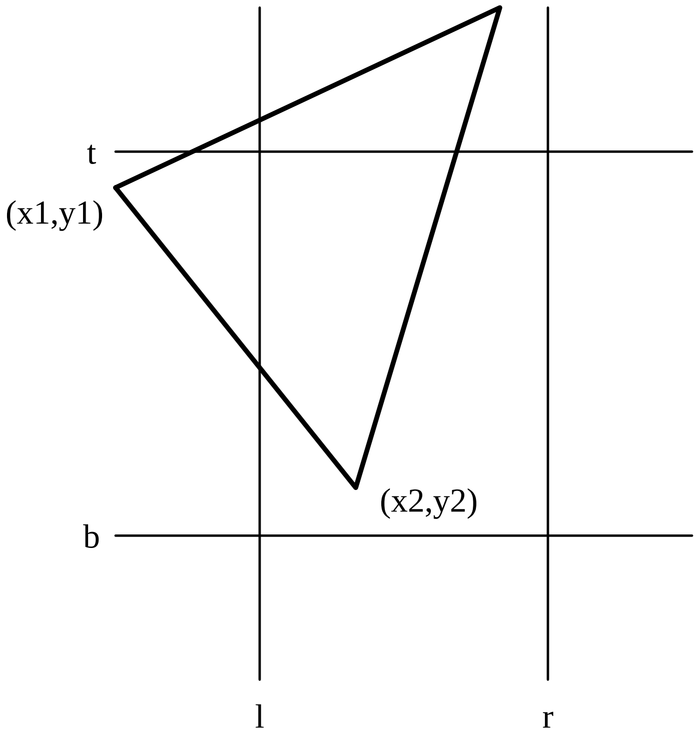
Overlap and Containment (1): Bounding Box Test



There is no overlap if

$$xmax < l \quad \text{or} \quad xmin > r \quad \text{or} \quad ymax < b \quad \text{or} \quad ymin > t$$

Overlap and Containment (2): Edge Intersection Tests

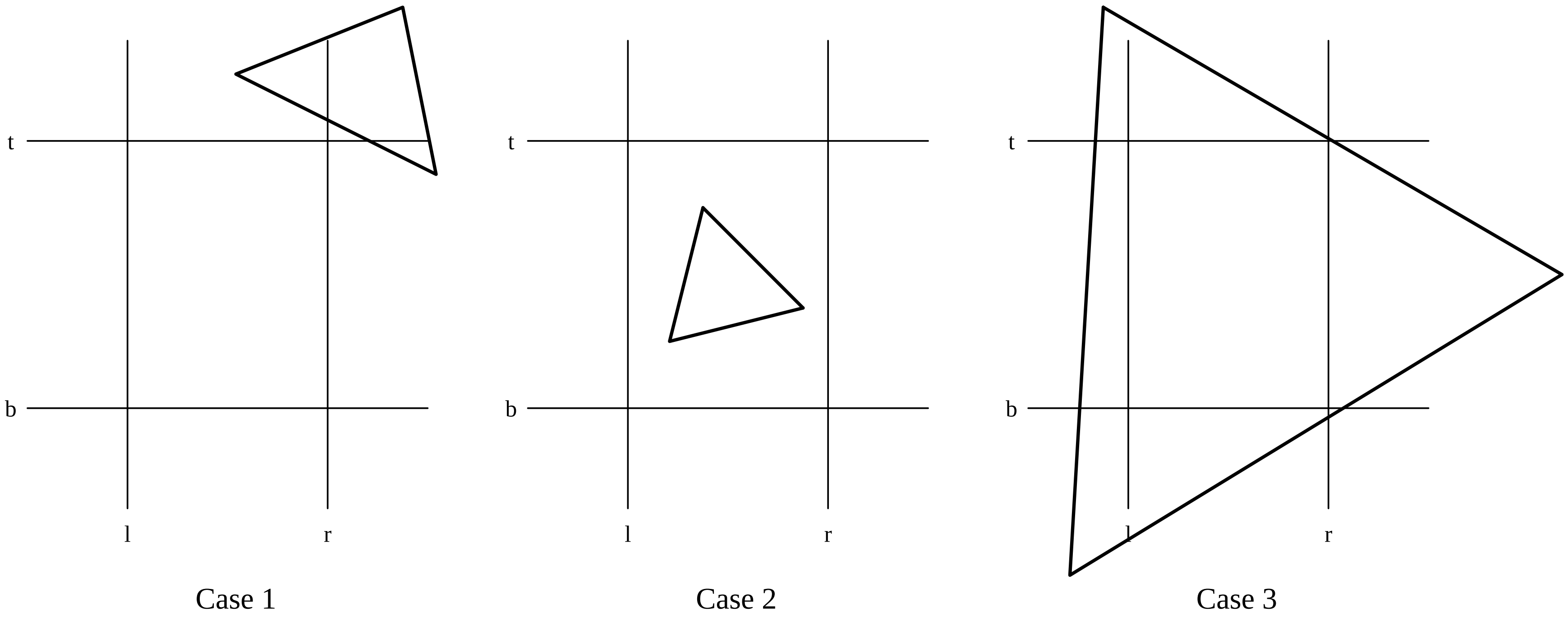


Compute

$$y = y_1 + \frac{l - x_1}{x_2 - x_1}(y_2 - y_1)$$

If $b < y < t$, then the edge from (x_1, y_1) to (x_2, y_2) intersects the left side.

Overlap and Containment (3): Containment Tests



If one vertex of triangle is inside rectangle: **Case 2**.

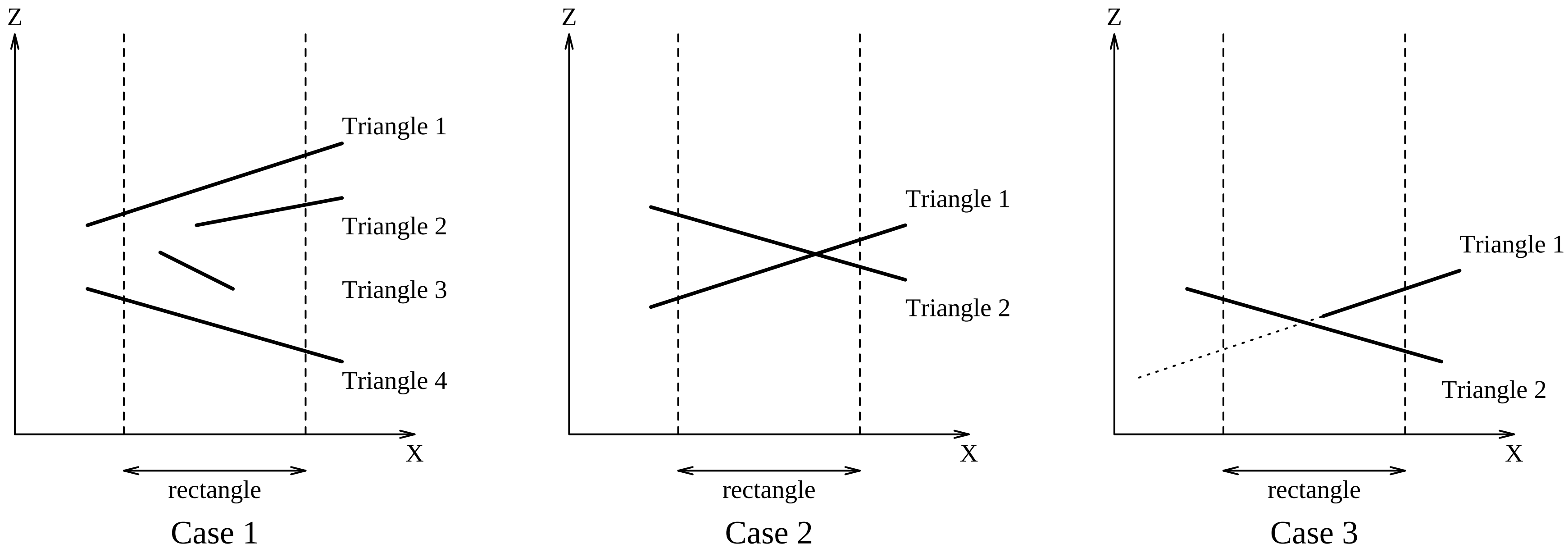
If one vertex of rectangle is inside triangle: **Case 3**.

Else: **Case 1**.

Based on these tests, we can classify every triangle as either **disjoint**, **intersecting** (including fully inside), or **enclosing**. If all triangles are disjoint, we fill with background color and return.

Nearest Enclosing Triangle

If there are any **enclosing** triangles, then we must check to see if one of them hides all of the other **intersecting** and **enclosing** triangles. At each vertex of the rectangle, compute the z depth of the plane of each overlapping triangle.



If one triangle is nearest at all four vertexes (**Case 1**), we fill the rectangle with its color and return.

Otherwise, we subdivide and process the four subrectangles recursively. Note that we miss an opportunity in **Case 3**.

Efficient Classification of Triangles

Two key facts to exploit:

- **If a triangle is disjoint from a rectangle, then it is disjoint from all subdivisions of that rectangle.**
- **If a triangle encloses a rectangle, then it encloses all subdivisions of that rectangle.**

Maintain three linked lists:

`dlist` All triangles known to be disjoint with the current rectangle.

`slist` All triangles known to enclose the current rectangle ("surround" list).

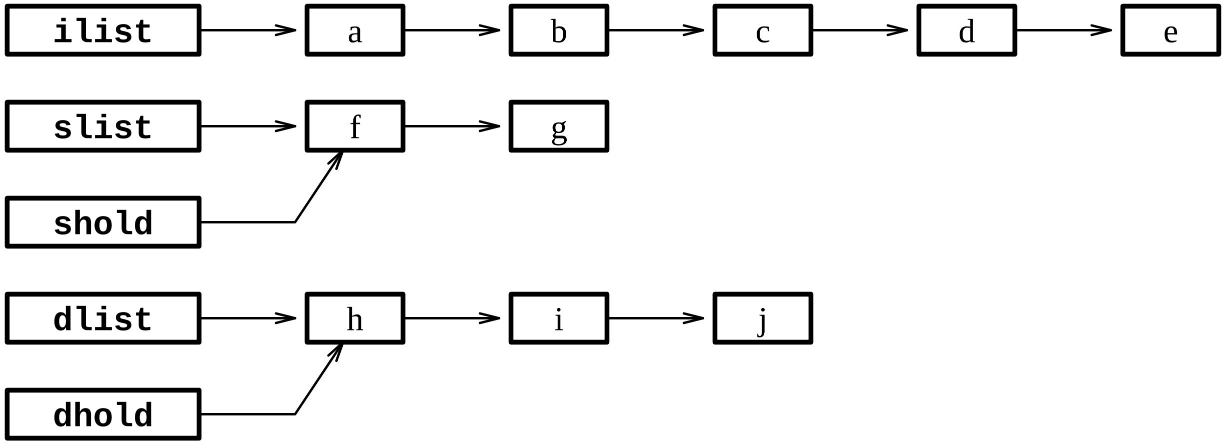
`ilist` All other triangles (they are all potentially intersecting).

Algorithm Outline

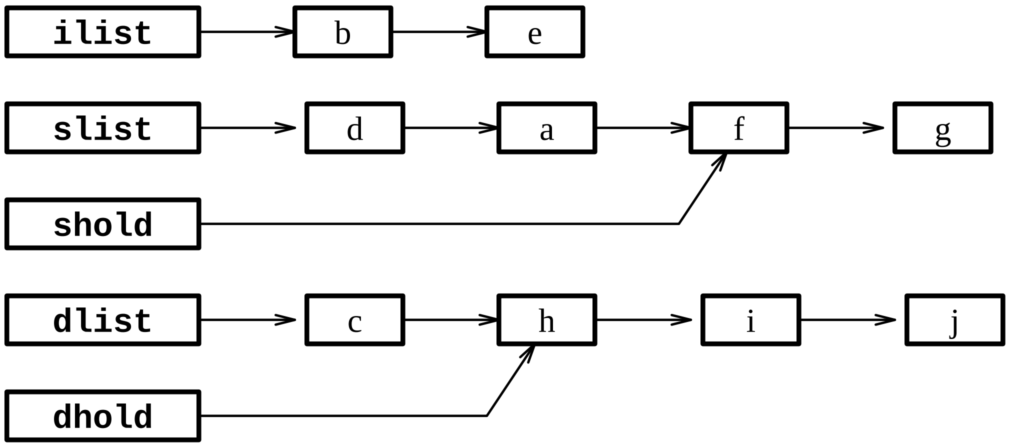
```
void warnock(rectangle r, triangle_t* dlist, triangle_t* slist, triangle_t* ilist)
{
   For each triangle on ilist
     perform overlap tests and move to dlist or slist if appropriate
   If slist is empty
      If ilist is empty
          Paint background color
          Restore ilist and return
      Else
          If there is a member of slist closer than all other elements
          of slist and ilist
              Paint with its color.
              Restore ilist and return
   Subdivide r into r1, r2, r3, r4
   warnock(r1,dlist,slist,ilist); warnock(r2,dlist,slist,ilist);
   warnock(r3,dlist,slist,ilist); warnock(r4,dlist,slist,ilist);
   Restore ilist and return.
}
```

A Fast Way to Restore `ilist`

Introduce pointers `shold` and `dhold`.

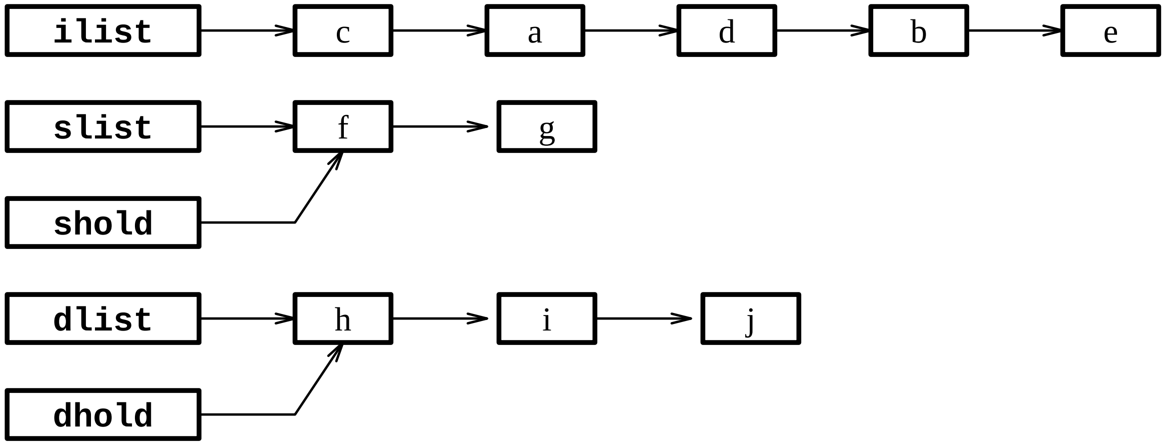


Distribute the rectangles from `ilist` onto `dlist` and `slist`:



Restoring `ilist`

To restore, pop them back onto `ilist` until we hit `shold` and `dhold`



C++ Classes

`wscene_t` . A Warnock scene. Redefines `solid()` member function.

`warnrec_t` . Represents each triangular face *after* perspective transformation. One `warnrec` is created for each front face in the scene. Subclass of `dsrec_t` which implements triangle overlap and containment tests.

`wplane_t` . Adds a method to `plane_t` for computing depth at a point.

`wrectangle_t` . Represents the subdivision rectangles in pixel coordinates. Its member functions implement the algorithm.

warnrec.h

```
class warnrec_t : public dsrec_t                    // Warnock record
{
public:
  ~warnrec_t() ;
   warnrec_t(const scene_t*,triangle_t*,warnrec_t*) ;
   warnrec_t*  givnext() const { return next; }
   warnrec_t*& setnext() { return next; }
private:
   wplane_t p ;        // plane through perspective transformed points
   warnrec_t* next ;
} ;
```

As each `triangle` is converted to a `warnrec`, it is perspective transformed.

wrectangle.h

```
class wrectangle_t
{
public:
  wrectangle_t(int,int,int,int) ;  // construct from l,r,b,t
  void fill(const warnrec_t*) const ; //
  int operator ^ (const warnrec_t&) const ; // triangle overlap test
private:
  int l , r , b , t ;                   // absolute device coordinates
  int in(const point_t&) const ;        // Is point in rectangle?
  point_t cen() const ;                 // Center of rectangle
  // minmax overlap test for triangle
  int minmax_overlap(const warnrec_t&) const ;
  // edge a—b intersects a boundary of rectangle:
  int rect_edge_sec(const point_t& a,const point_t& b) const ;
} ;
```

Drawing the Scene

```
void wscene_t::solid() const      // recursive screen subdivision
{
  int loc = 0 ;
  warnrec_t* w = 0 ;
  for (prim_t* p = prim; p; p = p→givnext())     // go through all
  {                                              // tmeshes of scene
    ((tmesh_t*)p)→setcolors(this,loc);
    triangle_t* tr = ((tmesh_t*)p)→givfacet() ;
    for (int i = 0; i < ((tmesh_t*)p)→givfacnum(); i++)
      if (tr[i].givp().frontface(point3_t(wv→givcen(),−wv→givd())))
        w = new warnrec_t(this,&tr[i],w);    // list of frontfaces
  }
  wrectangle_t(0,DWID,0,DHEI).fill(w);         // fill using list w
  delete w;                                    // free list
}
```

Depth Buffer (Z-buffer) Algorithm

```
float zbuf[DWID,DHEI];    // allocate the Z buffer
```

The Z buffer contains one floating point value for each pixel. We initialize it to contain 1.0 in every cell (this is $z = \infty$ under perspective transformation).

We fill the polygons in the usual way. However, instead of calling `setpix(x,y)`, we call `setzpix(x,y,z)`:

```
void setzpix(int x, int y, float z)
{
  if (z < zbuf[x][y]) {
    zbuf[x][y]=z;
    setpix(x,y);
    }
}
```

This only draws a pixel if it is nearer than the last known value at this location. It is a pixel-oriented painter's algorithm.

Advantages and Disadvantages

- **Very simple.** Can be implemented in hardware.
- **Works for arbitrary surface shapes.**
- **Requires huge amount of memory.**

Object-oriented Implementation

Recall our polygon filling code. It constructs an edge list and then scans the polygon one row at a time using the function `edge_t::solidscan`. We can reuse this machinery by defining a new kind of edge (a `zedge_t`) with its own `solidscan` function that computes z values in addition to x values.

To make this all work, we need to define several classes so that the correct virtual functions are called to create these `zedge_t`'s and scan them.

`zvertex_t` Subclasses `vertex3_t`.

`zedge_t` Redefines `update` and `solidscan` so that they compute z values as well as x values.

`zmesh_t` Redefines `alloc_vertex` to allocate `zvertex_t`'s.

`zscene_t` Redefines `solid` so that it allocates and frees the Z buffer.

zedge_t::solidscan

```
static void sline(int x1,int x2,int scan,double z,double dz)
{
  for (int x = x1; x < x2; x++ , z += dz)
    if (z < (*zbuf)[scan][x])              // compare to zbuffer
    {
      setpix(x,scan);
      (*zbuf)[scan][x] = z;                // set zbuffer
    }
}
void zedge_t::solidscan(double scan)
{
  double xdif = (givnext()->givx() - x) ;
  if (fabs(xdif) > 0.00001)
  {
    double dz = (((zedge_t*)givnext())->z - z)/xdif ;
    sline(int(x),int(givnext()->givx()),int(scan),z,dz) ;
  }
}
```

Interpolative Shading

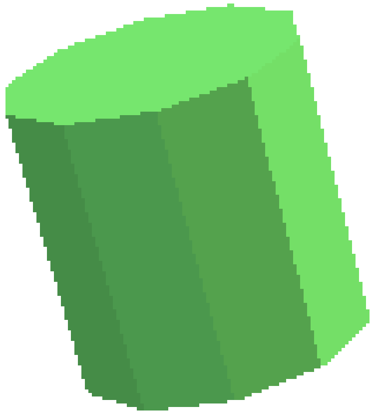
Getting beyond faceted shading.

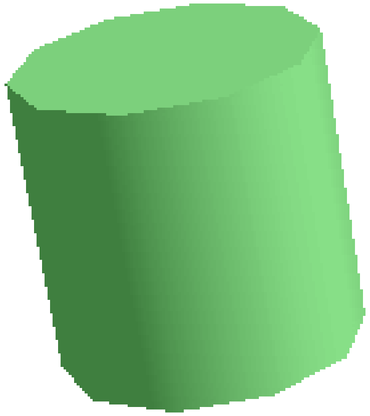
- **Smooth 3D object representations**. Quadrics, superquadrics, splines.

  Computing surface normals can be very expensive.

- **Interpolative shading**. Approximate curved objects by polygonal meshes. Compute a surface normal that varies smoothly from one face to the next.

  Computation can be very cheap. Many objects are well-approximated by polygonal meshes. Silhouettes are still polygonal.





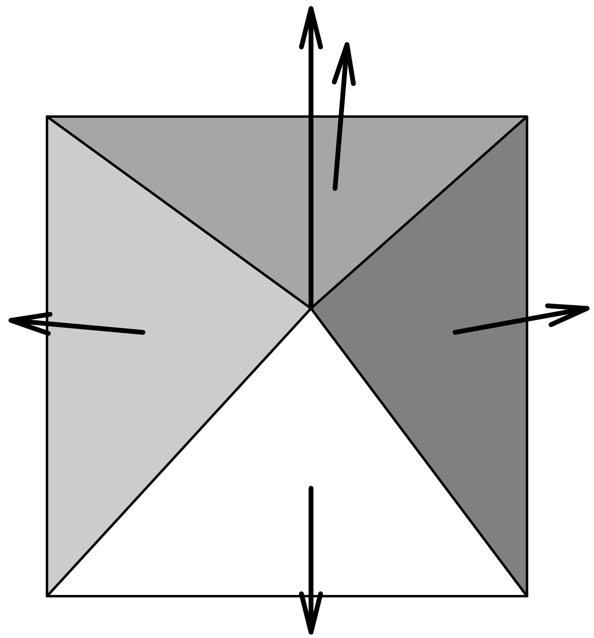
Two kinds of interpolative shading

- **Gouraud Shading.** Cheap but gives poor highlights.
- **Phong Shading.** Slightly more expensive, but gives excellent highlights.

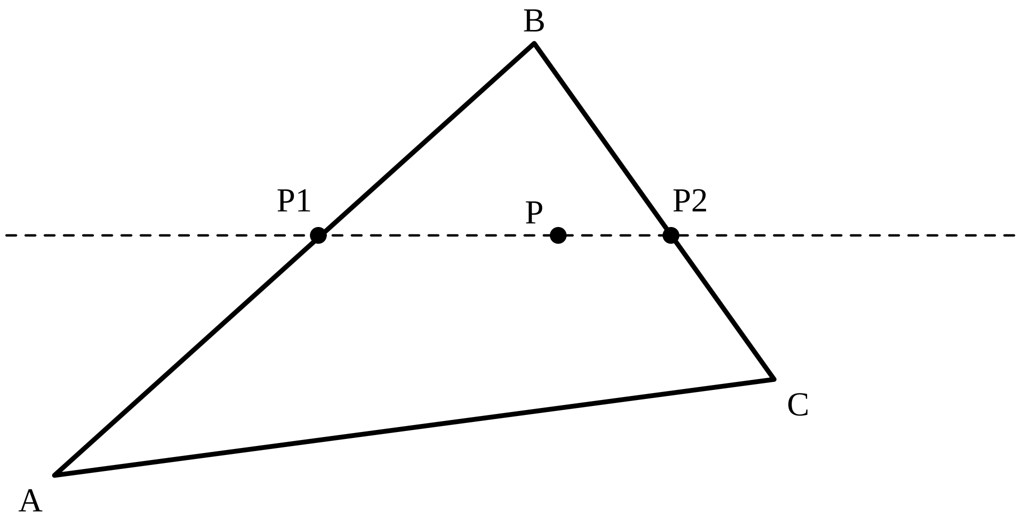
Vertex Normals

All interpolative shading methods rely on **vertex normals**.

A vertex normal is the average of the normals of all of the faces sharing that vertex.



Gouraud Shading



- **Compute RGB Color at Each Vertex.** Use the Phong illumination model (or any other).

- **Compute RGB Colors at P_1 and P_2.** By linear interpolation:

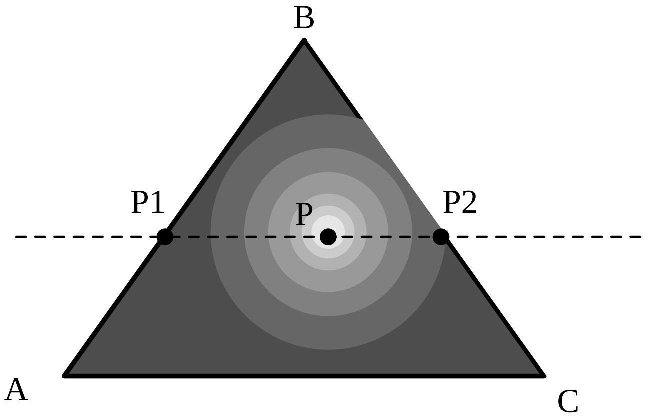
$$s = \frac{\|P_1 - B\|}{\|A - B\|} \qquad (R_{P_1}, G_{P_1}, B_{P_1}) = s(R_A, G_A, B_A) + (1 - s)(R_B, G_B, B_B)$$

- **Compute RGB Colors at P.** By linear interpolation:

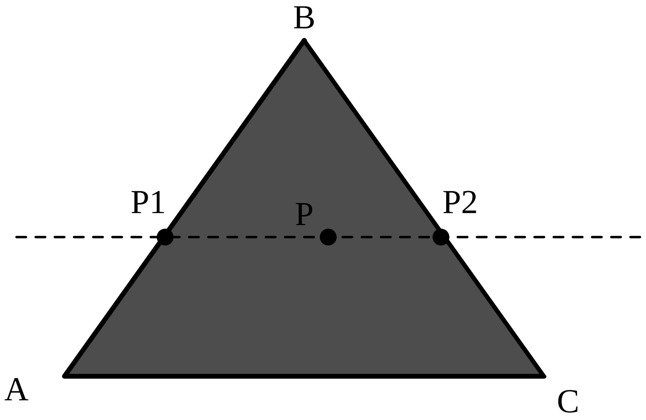
$$s = \frac{\|P - P_2\|}{\|P_1 - P_2\|} \qquad (R_P, G_P, B_P) = s(R_{P_1}, G_{P_1}, B_{P_1}) + (1 - s)(R_{P_2}, G_{P_2}, B_{P_2})$$

Poor Highlights from Gouraud Shading

- Suppose we are approximating a sphere and that the true highlight should appear in the *center* of a face (e.g., at P).
- The computed RGBs at A, B, and C will not have highlights (because they are far away from P).
- Any point in the interior will therefore not have a highlight.



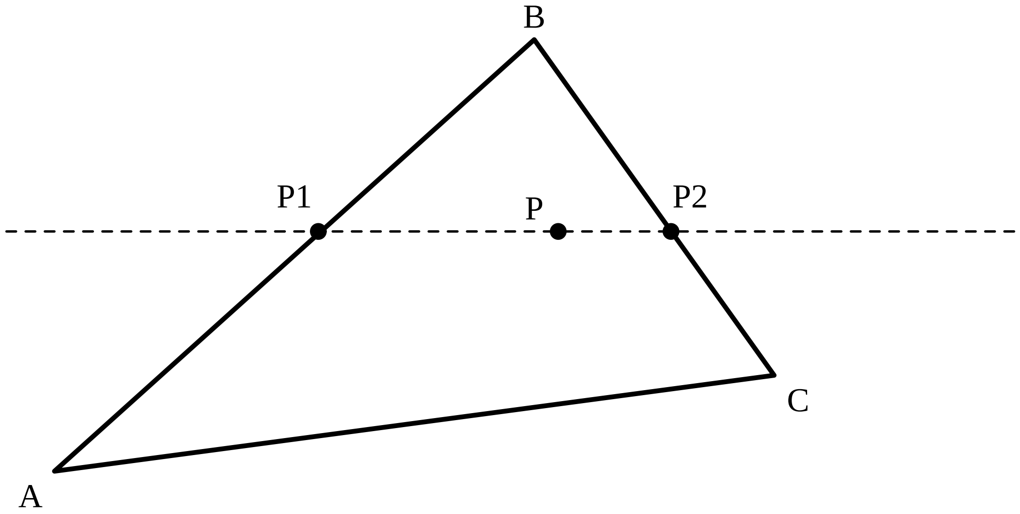
Desired



Gouraud Result

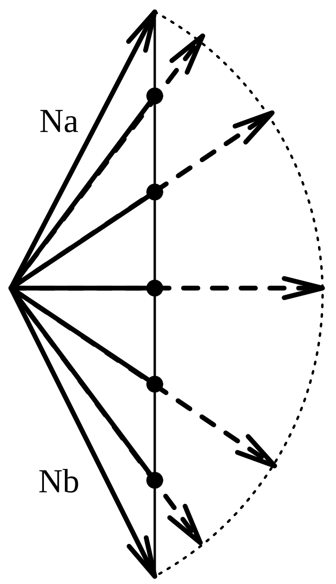
Phong Shading

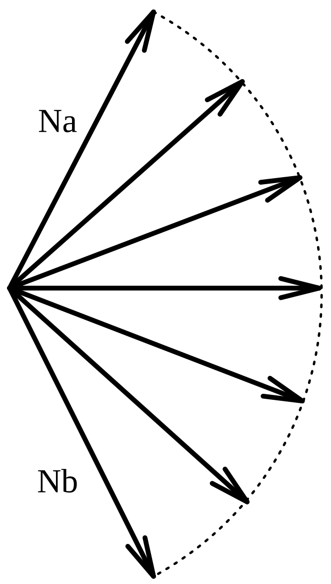
Interpolate the **normals** instead of the **RGB values**.



- **Compute normals at each vertex (A, B, and C).**
- **Compute the normals at P_1 and P_2.** By interpolation using the normals from A and B (and C and B).
- **Compute the normal at P.** By interpolating the normals from P_1 and P_2.
- **Compute RGB values at P.** Using Phong's rule.

Interpolating the Normals





- **Equally-spaced points along line N_A to N_B.** Does not produce equal angles. Requires renormalizing each interpolated vector.

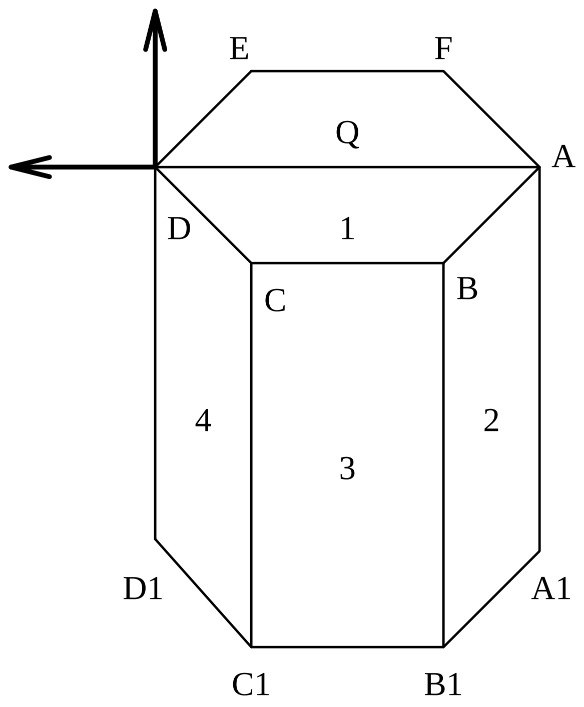
- **Equally-spaced angles**. Compute

$$\phi = \frac{cos^{-1}(N_A \cdot N_B)}{n}$$

  Construct a rotation matrix that will rotate N_A by angle ϕ so that it coincides with N_B after n rotations.

Describing Objects for Phong Shading

- **Use quandrangles instead of triangles.** Avoids shading artifacts.
- **Use "cloned" vertexes to retain sharp edges where needed.**



Vertex D needs two normals. One for smooth shading of the cylinder sides and another for shading the top. We create this by duplicating all vertexes that lie on a sharp edge.

Describing Objects for Phong Shading (2)

Vertex Array

vertex	coordinates	vertex	coordinates
v[0]	A	v[12]	A
v[1]	B	v[13]	B
v[2]	C	v[14]	C
v[3]	D	v[15]	D
v[4]	E	v[16]	E
v[5]	F	v[17]	F
v[6]	A1	v[18]	A1
v[7]	B1	v[19]	B1
v[8]	C1	v[20]	C1
v[9]	D1	v[21]	D1
v[10]	E1	v[22]	E1
v[11]	F1	v[22]	F1

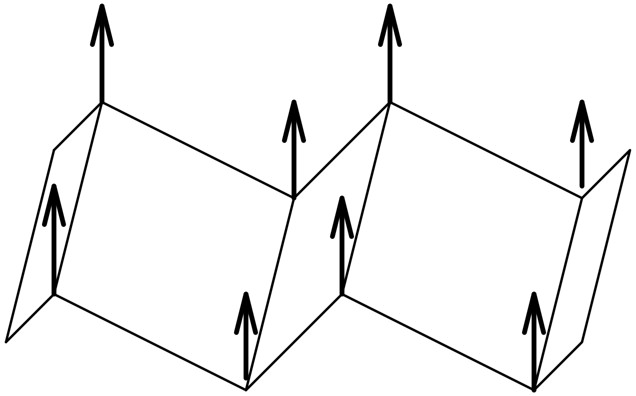
Face Array

face	vertexes			
q[0]	12	15	16	17
q[1]	12	13	14	15
q[2]	0	6	7	1
q[3]	1	7	8	2
q[4]	2	8	9	3

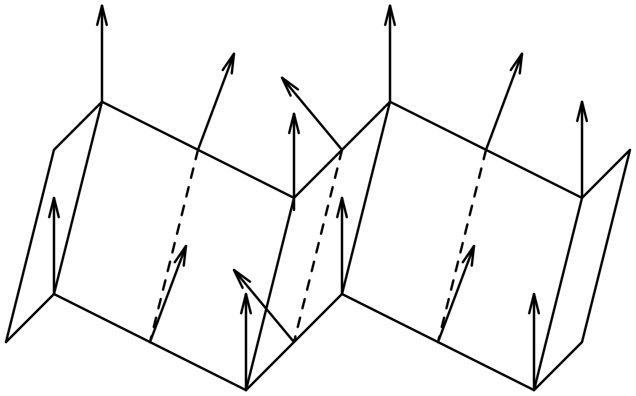
etc.

Describing Objects for Phong Shading (2)

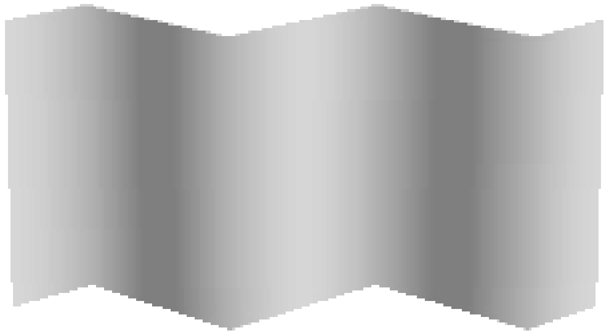
Creating a wave object. All vertex normals point up, so the interpolated vertexes also point upwards.

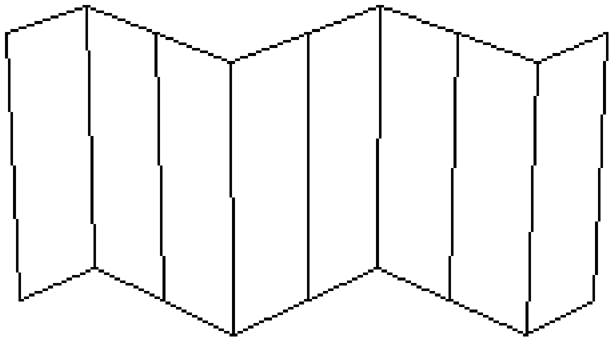


Solution: Subdivide each quadrangle. The intermediate vertexes will have non-vertical normals.

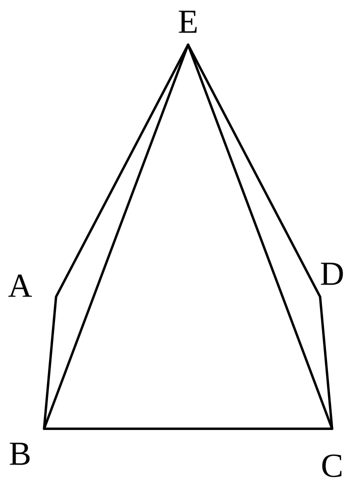


Describing Objects for Phong Shading (3)

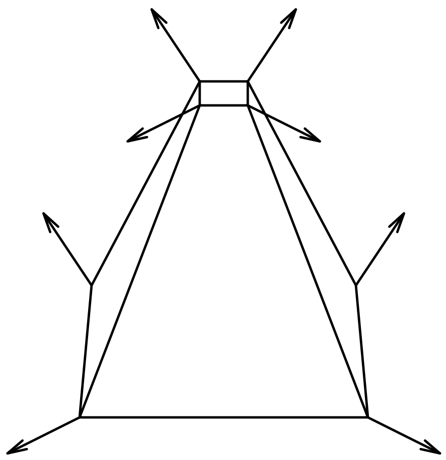




Describing Objects for Phong Shading (4)

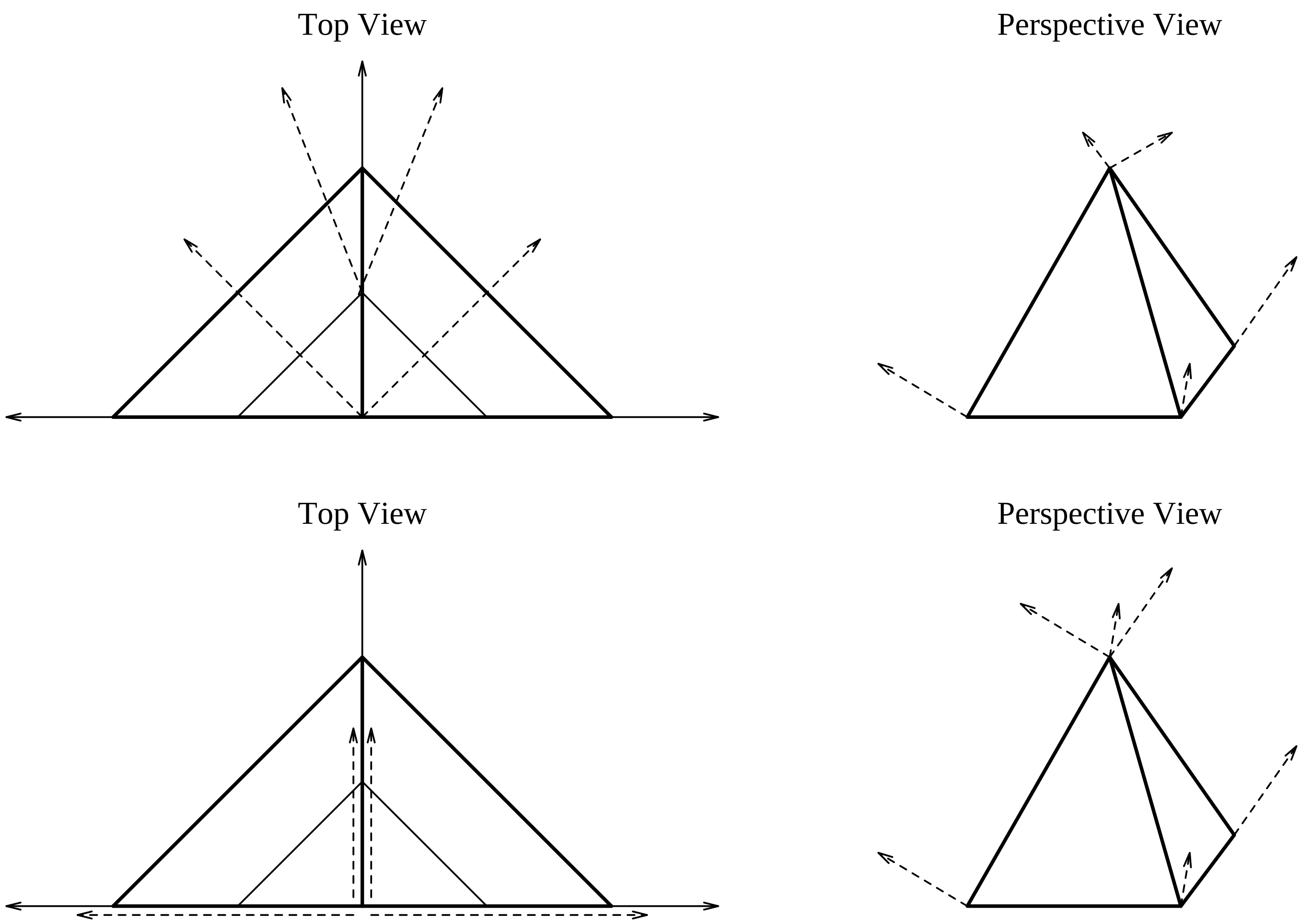


To model a cone, we use a pyramid, but we must clone the top vertex many times to avoid smoothing it away. We also must clone the bottom vertexes.



Why Use Quadrangles? Why Duplicate E Twice?

We could have represented the sides of the cone as triangles and used one copy of E for each triangle. However, this would create sudden discontinuities in the interpolated normals, with the result that the cone will not be smoothly shaded.



Implementing Phong Shading

- **Subclass `tmesh` with `pmesh`.** A `pmesh` is a mesh of quadrangles.
- **Computing Vertex Normals.** After reading in a `pmesh`, we loop over the vertexes and compute their vertex normals.
- **Interpolating Normals.** When the polygon filling algorithm constructs a list of edges, we subclass `edge_t` to create `pedge_t` to add two fields:

`norm`	Normal at the edge-scanline intersection
`rotmat`	Rotation matrix to apply for each new scan line.

  For each new scan line, the fill algorithm rotates the `norm`s of each of the edges. The `solidscan` member function then interpolates between these two norms.
- **Other Subclasses.**

  `pvertex_t` subclasses `vertex3_t` to add the vertex norm.

  `quadrang_t` subclasses `dstriang_t` to support fourth vertex.

  `pquad_t` subclasses `convex_t` to intercept `solid()`.

  `dsrec4_t` subclasses `dsrec_t` to implement quadrangle overlap tests.

Computing Vertex Normals

```
prim_t& pmesh_t::read(char* filename)
{
   tmesh_t::read(filename);
   for (int i = 0; i < vernum; i++)          // compute vertex normals
   {
      point3_t sum ;
      for (int j = 0; j < facnum; j++)
         if (facet[j].givv()[0] == i          // does quadrangle j
          || facet[j].givv()[1] == i          // share vertex i ?
          || facet[j].givv()[2] == i
          || facet[j].givv()[3] == i)
         sum += facet[j].givp().givnorm();
      ((pvertex_t*)vertex[i])->setnorm(sum.normalize());
   }
   return *this;
}
```

Interpolating Normals

```
pedge_t::pedge_t(const pvertex_t& a,const pvertex_t& b) : edge_t(a,b) , norm(a.givnorm())
{
  float ydif = ymax − ymin ;
  float corr = ydif/(1−ymax+int(ymax)) ;    // correction for normal
  rotmat = ::rotmat(a.givnorm(),b.givnorm(),ydif);
  norm *= ::rotmat(b.givnorm(),a.givnorm(),corr);    // start value
}

void pedge_t::update()  // move to next scan line
{
  x += dx;
  norm *= rotmat;
}
```

Interpolating Normals Across a Line

```
void pedge_t::solidscan(double scan)            // horizontal fill line
{
  int xpix = int(x+0.5) ,                       // start location of scanline
      ypix = int(scan+0.5) ,
      n    = int(next→givx()+0.5) − xpix ;      // no. of pixels
  point3_t pixnorm(norm) ;                      // pixel normal
  mat33_t m = ::rotmat(pixnorm,((pedge_t*)next)→norm,n) ; // rotation per pixel
  for (int i = 0; i ≤ n; i++)                   // set n+1 pixels
  {
    col(illum(pixnorm));                        // set colormap index of this pixel
    setpix(xpix++,ypix);
    pixnorm *= m;                               // compute norm of next pixel
  }
}
```

rotmat

```
mat33_t rotmat(const point3_t& a,const point3_t& b,float n)
{
   double d = a*b ;
   mat33_t m ;
   double cph = cos(acos(d)/n),
          cx = a.givy()*b.givz() - a.givz()*b.givy(),   xx = cx*cx,
          cy = a.givz()*b.givx() - a.givx()*b.givz(),   yy = cy*cy,
          cz = a.givx()*b.givy() - a.givy()*b.givx(),   zz = cz*cz,
          l  = xx + yy + zz,
          u = (1 - cph)/l,          v = sqrt((1-cph*cph)/l),
          uxy = u*cx*cy,            vx  = v*cx,
          uxz = u*cx*cz,            vy  = v*cy,
          uyz = u*cy*cz,            vz  = v*cz;
   m[0] = cph + u*xx;   m[1] = uxy + vz;      m[2] = uxz - vy;
   m[3] = uxy - vz;     m[4] = cph + u*yy;    m[5] = uyz + vx;
   m[6] = uxz + vy;     m[7] = uyz - vx;      m[8] = cph + u*zz;
   return m;
}
```