Shading

Color Interpolation

- **Flat shading**

```
glBegin(GL_TRIANGLE);
  glColor3f(...);
  glVertex3f(...);
  glVertex3f(...);
  glVertex3f(...);
glEnd();
```

- **Gouraud shading**

```
glBegin(GL_TRIANGLE);
  glColor3f(...); glVertex3f(...);
  glColor3f(...); glVertex3f(...);
  glColor3f(...); glVertex3f(...);
glEnd();
```


Crow's Algorithm for Convex Polygons

```
scany( vert v[], int n, int bot)
  li = ri = bot;
  ly = ry = y = ceil(v[bot].y);
  for( rem = n; rem > 0 ; ) {
    while( ly <= y && rem-- > 0 ) {
      i = li; li = mod(li-1,n); ly = ceil(v[li].y);
      if( ly > y )
        differencey(&v[i],&v[li],&l,&dl,y);
    }
    while( ry <= y && rem-- > 0 ) {
      i = ri; ri = mod(ri+1,n); ry = ceil(v[ri].y);
      if( ry > y )
        differencey(&v[i],&v[ri],&r,&dr,y);
    }
    for( ; y < ly && y < ry; y++ ) {
      scanx(&l,&r,y);
      increment(&l,&dl);
      increment(&r,&dr);
    }
```

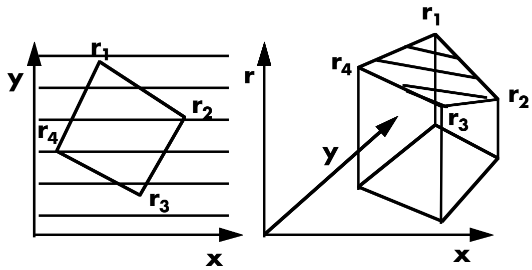
Crow’s Algorithm for Convex Polygons

```
scanx( vert *l, vert *r, int y)
  lx = ceil(l->x);
  rx = ceil(r->x);
  if( lx < rx ) {
    differencex( l, r, &s, &ds, lx );
    for( x=lx; x<rx; x++ ) {
      point(x,y);
      increment(&s,&ds);
    }
  }
```


Interpolation

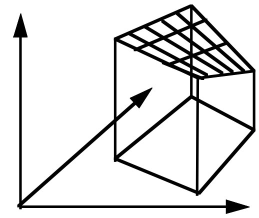
```
differencey(vert *b,vert *t, vert *e,vert *de, int y)
{
  de->x = (t->x - b->x)/ (t->y - b->y);
   e->x = b->x + (y - b->y) * de->x;
  de->v = (t->v - b->v)/ (t->y - b->y);
   e->v = b->v + (y - b->y) * de->v;
}
incrementy(vert *e, vert *de)
{
   e->x += de->x;
   e->v += de->v;
}

differencex(vert *l,vert *r, vert *s,vert *ds, int x)
{
  ds->v = (r->v - l->v)/ (r->x - l->x);
   s->v = l->v + (x - l->x) * de->v;
}
incrementx(vert *e, vert *de)
{
   e->v += de->v;
}
```

Gouraud Shading



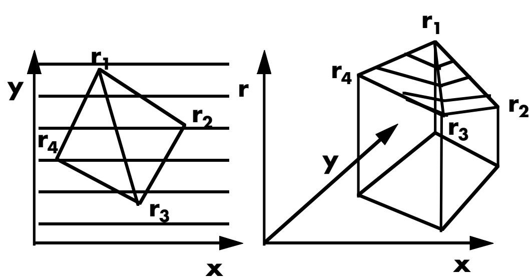
Gouraud Interpolation

Note: Not rotationally invariant

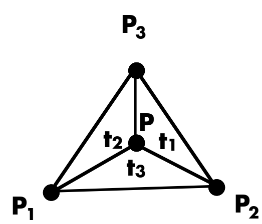


Ruled Surface

Triangle Interpolation



Gouraud Interpolation



$P=t_1P_2+t_2P_2+t_3P_3$

$C=t_1C_2+t_2C_2+t_3C_3$

Barycentric Interpolation

Polygon Interpolation

Interpolate many quantities

This lecture:

- **Texture mapping**
  - Interpolate texture coordinates
  - Lookup color at that position in a texture map/image

Future lectures:

- **Z-buffering**
  - Interpolate z
  - Resolve hidden surfaces at each pixel
- **Phong shading**
  - Interpolate normal
  - Compute new color at each pixel

Texture Maps

Map an image to a surface

Q. How is texture mapped to the surface?

- **Dimensionality**
- **2D Texture coordinates (s,t)**
  - Surface parameters (u,v)
  - Direction: reflection vectors
  - Projection: cylinder

Q. What does texture control?

- Surface color and opacity
- Reflection functions
- Illumination functions: shadow maps
- Perturb geometry: displacement and bump maps

Texture Mapping <=> Image Warp

2D Texture Space

↓ **Parameterization**

3D Object Space

↓ **Model Transformation**

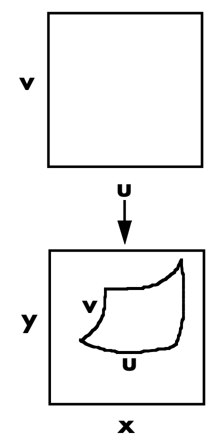
3D World Space

↓ **Viewing Transformation**

3D Camera Space

↓ **Projection**

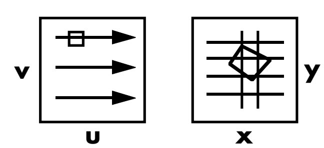
2D Image Space



Mapping Algorithms

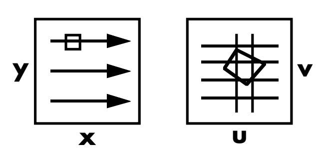
Forward mapping (video effects)

```
for all (u,v)
  (x,y) = f(u,v)
  image[x][y] = texture[u][v]
```



Backward mapping (workstations)

```
for all (x,y)
  (u,v) = f^-1(x,y)
  image[x][y] = texture[u][v]
```



Generalized Interpolation

Vertex: $(x,y,z,1,r,g,b,a,u,v,...,1)$

Parameters affinely related ↓ **Transform and Clip Vertices**

$(xw,yw,zw,w,r,g,b,a,u,v,...,1)$

↓ **Project Vertices**

$(xw/w,yw/w,zw/w,\underline{w/w},r/w,g/w,b,a/w,u/w,v/w,....,1/w)$

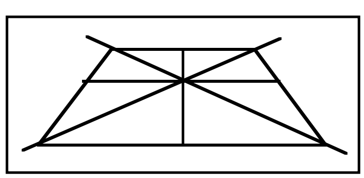
↓ **Interpolate between vertices (difference and increment)**

"Hyperbolic Interpolation," J. Blinn ...

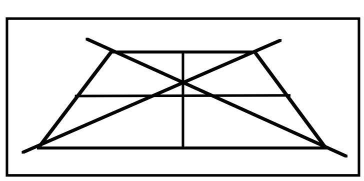
↓ **Project at Pixels (divide by [1/w])**

...

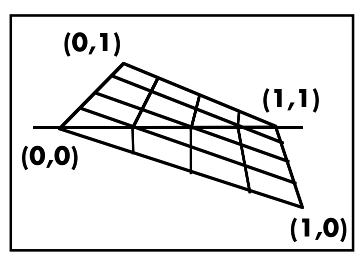
Linearly Interpolating Texture Coords



Correct Linear Perspective



Incorrect Perspective



Linear Interpolation, *Bad*
Perspective Interpolation, *Good*

Polygonal Mapping

Parameterization may be written as a linear map

$$P = P_0 + uP_1 + vP_2$$

The transformation from texture space to image space be performed with a single 2d projective matrix

$$(x, y, w) = P((V(M(T(u, v, 1))))) = (PVMT)(u, v, 1) \quad \begin{pmatrix} x \\ y \\ w \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix}$$

The inverse transformation from image space to texture space is also a 2d projective matrix

$$\begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix} \begin{pmatrix} x \\ y \\ w \end{pmatrix}$$

Incremental Texture Mapping Loop

```
x = xmin;
unum = A*x+B*y+C;
vnum = D*x+E*y+F;
den  = G*x+H*y+I;
for( ; x<xmax; x++ ) {
  image[x][y]=texture(unum/den,vnum/den);
  unum += A;
  vnum += D;
  den  += G;
}
```

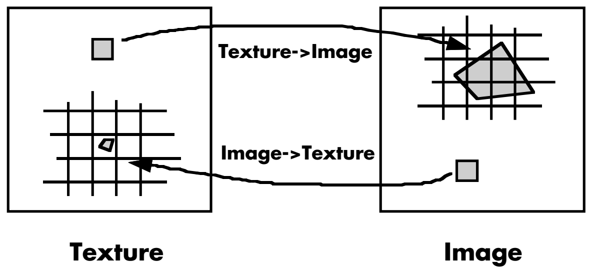
$$u = \frac{unum}{den} = \frac{Ax + By + C}{Gx + Hy + I}$$

$$v = \frac{vnum}{den} = \frac{Dx + Ey + F}{Gx + Hy + I}$$

$$x' = \frac{x}{w} = \frac{au + bv + c}{gu + hv + i}$$

$$y' = \frac{y}{w} = \frac{du + ev + f}{gu + hv + i}$$

Filtering

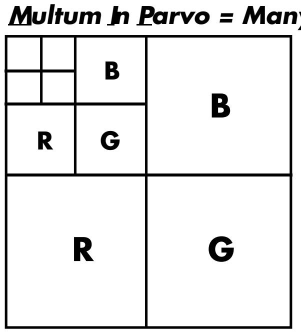


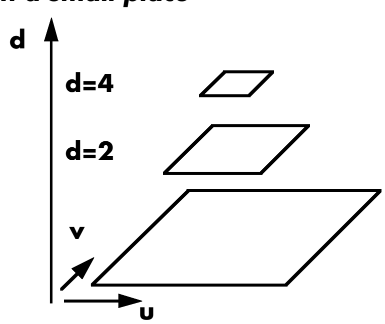
Two cases:

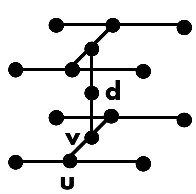
1. Magnification / Interpolation
2. Minification / Decimation / Averaging

MipMaps - L. Williams

Multum In Parvo = Many things in a small place







Trilinear mip-map interpolation