University of British Columbia
CPSC 314 Computer Graphics
Jan-Apr 2005

Tamara Munzner

Prog 2, Kangaroo Hall of Fame, Midterm Review, (Interpolation if time) Week 6, Wed Feb 9

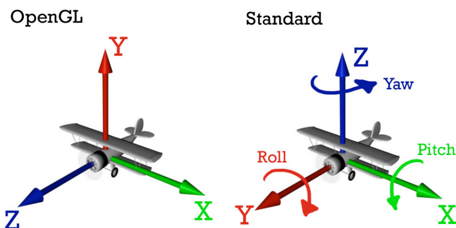
http://www.ugrad.cs.ubc.ca/~cs314/Vjan2005

Program 2 Corrections/Clarifications

- handin 314 proj2 (not 414)
- 'f' not 's' to toggle flat/smooth shading
  - 's' already in use for camera
- add: 't' to toggle between randomly colored and grey terrain
  - makes it easier to check if lighting correct
- add: 'u' to replace terrain with new randomly generated geometry
- consider adding for a bit of extra credit:
  - '+/-' toggle to increment/decrement abs cam speed

Program 2 Corrections/Clarifications

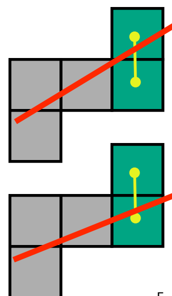
- roll/yaw confusion
  - first para. correct, second para. wrong
    - left horiz drag = yaw, right horiz drag = roll
    - image + flying expertise courtesy of Matt Baumann



Program 2 Quick Demo

Review: Midpoint Algorithm

- moving incrementally along x direction
  - draw at current y value, or move up to y+1?
    - check if midpoint between two possible pixel centers above or below line
- candidates
  - top pixel: (x+1,y+1),
  - bottom pixel: (x+1, y)
- midpoint: (x+1, y+.5)
- check if midpoint above or below line
  - below: top pixel
  - above: bottom pixel
- assume $x_1 < x_2$, slope $0 < \frac{dy}{dx} < 1$



Review: Bresenham Algorithm

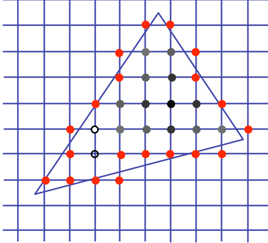
```
y=y0; e=0;
for (x=x0; x <= x1; x++) {
  draw(x,y);
  if (2(e+dy) < dx) {
     e = e+dy;
  } else {
     y=y+1;
     e=e+dy-dx;
}}
```

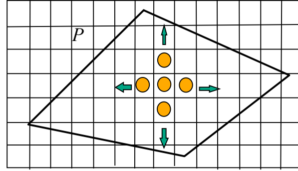
- all integer arithmetic
- cumulative error function

```
y=y0; eps=0
for ( int x = x0; x <= x1; x++ ){
  draw(x,y);
  eps += dy;
  if ( (eps << 1) >= dx ){
     y++;  eps -= dx;
  }
}
```

Review: Flood Fill

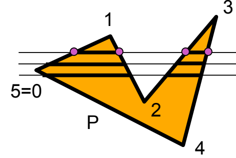
- draw polygon edges, seed point, recursively set all neighbors until boundary is hit to fill interior
- drawbacks: visit pixels up to 4x, per-pixel memory storage needed

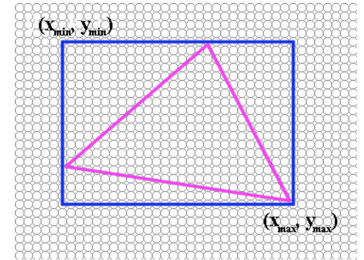




Review: Scanline Algorithms

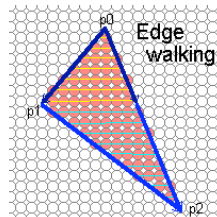
- set pixels inside polygon boundary along horizontal lines one pixel apart
- use bounding box to speed up





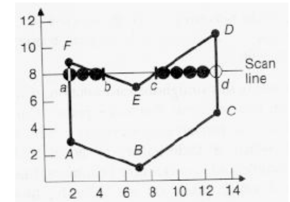
Review: Edge Walking

- basic idea:
  - draw edges vertically
    - interpolate colors down edges
  - fill in horizontal spans for each scanline
    - at each scanline, interpolate edge colors across span



Review: General Polygon Rasterization

- idea: use a parity test




```
for each scanline
    edgeCnt = 0;
    for each pixel on scanline (l to r)
        if (oldpixel->newpixel crosses edge)
            edgeCnt ++;
        // draw the pixel if edgeCnt odd
        if (edgeCnt % 2)
            setPixel(pixel);
```


Hall of Fame

But Wait, There's More!

- nice comment :)
  - “this project is fun, only one of the very few that I actually enjoyed. Too bad there's only one CG course in UBC =(
- two fourth year CG courses await you!
  - 424 Geometric Modelling
  - 426 Animation

Midterm Review

Midterm Exam

- Friday Feb 11 10am-10:50am
  - you may use one handwritten 8.5"x11" sheet
    - one side of page
  - no other notes, no books
  - nonprogrammable calculators OK
  - arrive on time!
  - sit every other seat, ID out in front of you
  - coats and bags in front of room

What's Covered

- transformations
- viewing and projections
- coordinate systems of rendering pipeline
- lighting and shading

- **not** scan conversion

Reading

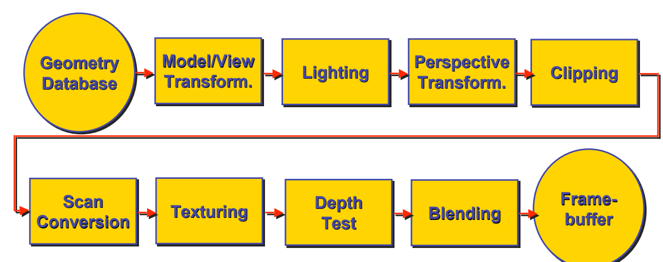
- FCS book, Red book
  - see web page for details
  - you can be tested on material in book but not covered in lecture
  - you can be tested on material covered in lecture but not covered in book

Old Exams Posted

- see course web page

The Rendering Pipeline

- pros and cons of pipeline approach



Transformations

translate(a,b,c)

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & & & a \\ & 1 & & b \\ & & 1 & c \\ & & & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

scale(a,b,c)

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a & & & \\ & b & & \\ & & c & \\ & & & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$\mathrm{Rotate}(x,\theta)$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & & & \\ & \cos\theta & -\sin\theta & \\ & \sin\theta & \cos\theta & \\ & & & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$\mathrm{Rotate}(y,\theta)$

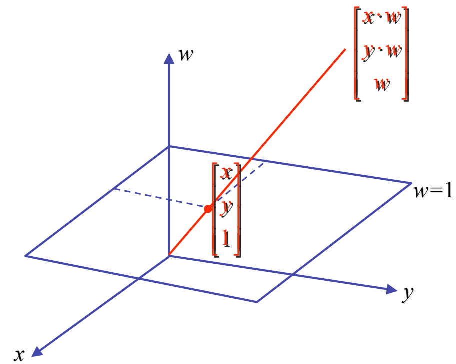
$$\begin{bmatrix} \cos\theta & & \sin\theta & \\ & 1 & & \\ -\sin\theta & & \cos\theta & \\ & & & 1 \end{bmatrix}$$

$\mathrm{Rotate}(z,\theta)$

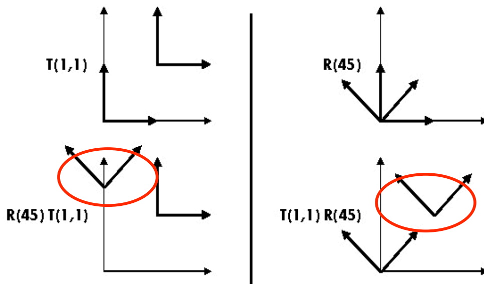
$$\begin{bmatrix} \cos\theta & -\sin\theta & & \\ \sin\theta & \cos\theta & & \\ & & 1 & \\ & & & 1 \end{bmatrix}$$

Homogeneous Coordinates

-



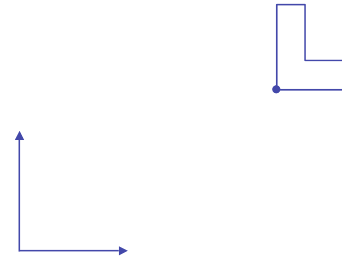
Composing Transformations



Ta Tb = Tb Ta, but Ra Rb != Rb Ra and Ta Rb != Rb Ta

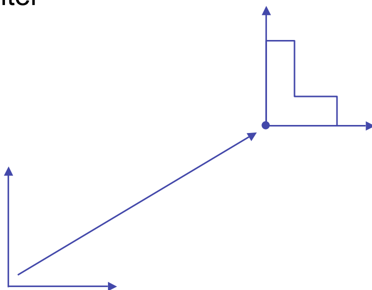
Composing Transformations

- example: rotation around arbitrary center



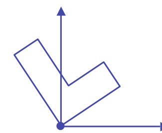
Composing Transformations

- example: rotation around arbitrary center
  - step 1: translate coordinate system to rotation center



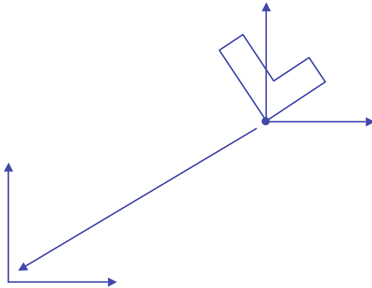
Composing Transformations

- example: rotation around arbitrary center
  - step 2: perform rotation



Composing Transformations

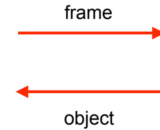
- example: rotation around arbitrary center
  - step 3: back to original coordinate system



Composing Transformations

- rotation about a fixed point
  p' = $TRT^{-1}p$
- rotation around an arbitrary axis

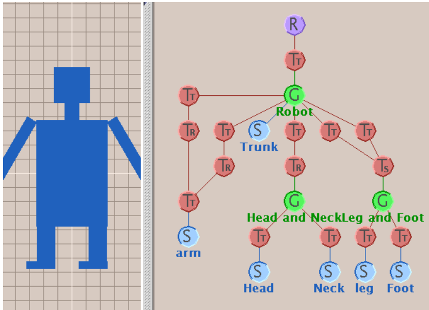
- considering frame vs. object
  - p' = DCBAp



OpenGL:
D
C
B
A
draw p

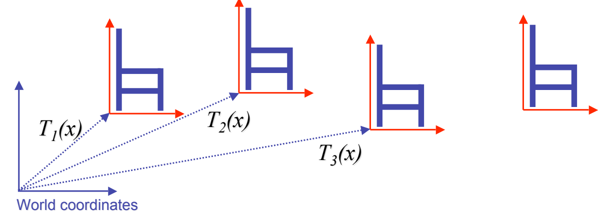
Transformation Hierarchies

- hierarchies don't fall apart when changed
- transforms apply to graph nodes beneath



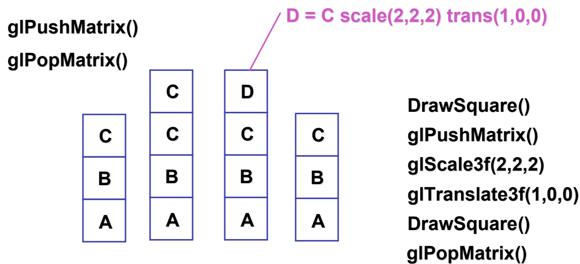
Matrix Stacks

- push and pop matrix stack
  - avoid computing inverses or incremental xforms
  - avoid numerical error



Matrix Stacks

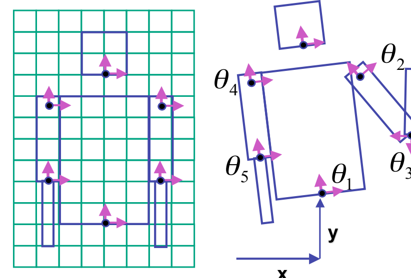
glPushMatrix()
glPopMatrix()




```
DrawSquare()
glPushMatrix()
glScale3f(2,2,2)
glTranslate3f(1,0,0)
DrawSquare()
glPopMatrix()
```


Transformation Hierarchies

- example




```
glTranslate3f(x,y,0);
glRotatef(θ1,0,0,1);
DrawBody();
glPushMatrix();
  glTranslate3f(0,7,0);
  DrawHead();
glPopMatrix();
glPushMatrix();
  glTranslate(2.5,5.5,0);
  glRotatef(θ2,0,0,1);
  DrawUArm();
  glTranslate(0,-3.5,0);
  glRotatef(θ3,0,0,1);
  DrawLArm();
glPopMatrix();
... (draw other arm)
```

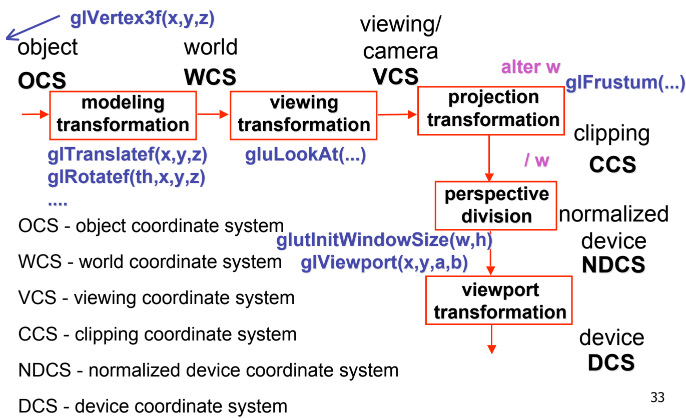
Display Lists

- reuse block of OpenGL code
- more efficient than immediate mode
  - code reuse, driver optimization
- good for static objects redrawn often
  - can't change contents
  - not just for multiple instances
    - interactive graphics: objects redrawn every frame
- nest when possible for efficiency

Double Buffering

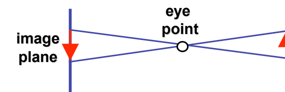
- two buffers, front and back
  - while front is on display, draw into back
  - when drawing finished, swap the two
- avoid flicker

Projective Rendering Pipeline

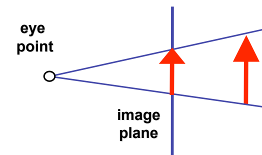


Projection

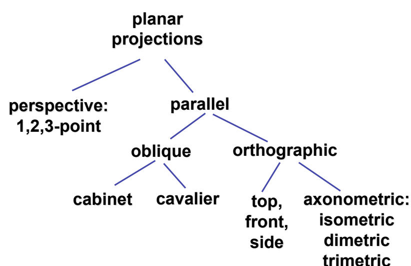
- theoretical pinhole camera



– image inverted, more convenient equivalent

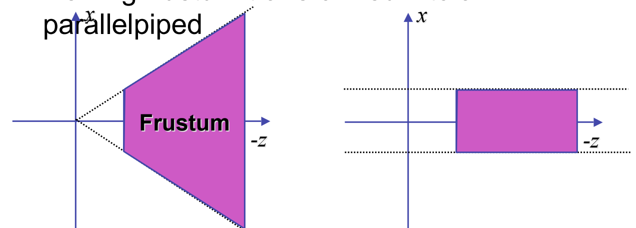


Projection Taxonomy



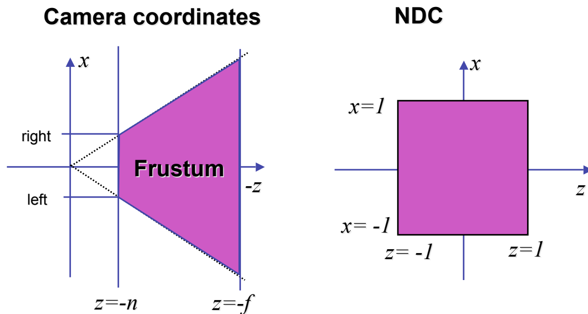
Projective Transformations

- transformation of space
  - center of projection moves to infinity
  - viewing frustum transformed into a parallelpiped



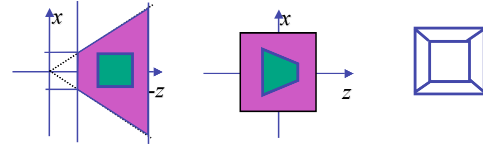
Normalized Device Coordinates

left/right x =+/- 1, top/bottom y =+/- 1, near/far z =+/- 1

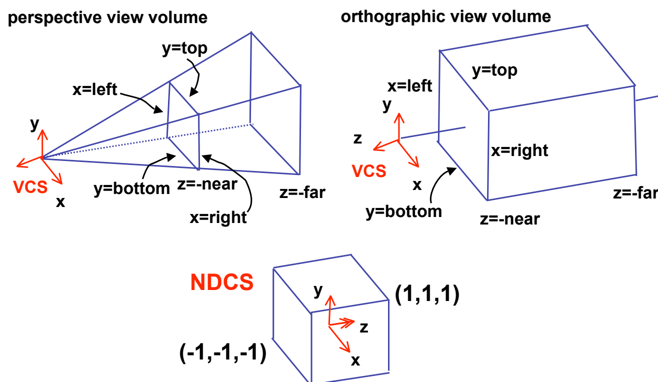


Projection Normalization

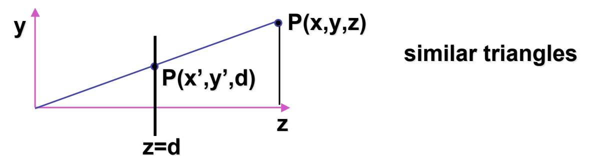
- distort such that orthographic projection of distorted objects is desired persp projection



Transforming View Volumes



Basic Perspective Projection



$$\frac{y'}{d} = \frac{y}{z} \rightarrow y' = \frac{y \cdot d}{z} \quad \text{also} \quad x' = \frac{x \cdot d}{z} \quad \text{but} \quad z' = d$$

- nonuniform foreshortening
  - not affine

Basic Perspective Projection

- can express as homogenous 4x4 matrix!

$$\begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} \xrightarrow{/w} \begin{bmatrix} x \cdot d/z \\ y \cdot d/z \\ d \end{bmatrix}$$

Projective Transformations

- determining the matrix representation
  - need to observe 5 points in general position, e.g.
    - $[left,0,0,1]^T \rightarrow [-1,0,0,1]^T$
    - $[0,top,0,1]^T \rightarrow [0,1,0,1]^T$
    - $[0,0,-f,1]^T \rightarrow [0,0,1,1]^T$
    - $[0,0,-n,1]^T \rightarrow [0,0,-1,1]^T$
    - $[left*f/n,top*f/n,-f,1]^T \rightarrow [-1,1,1,1]^T$
  - solve resulting equation system to obtain matrix

OpenGL Orthographic Matrix

- scale, translate, reflect for new coord sys
  - understand derivation from VCS!

$$P' = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bot} & 0 & -\frac{top+bot}{top-bot} \\ 0 & 0 & \frac{-2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix} P$$

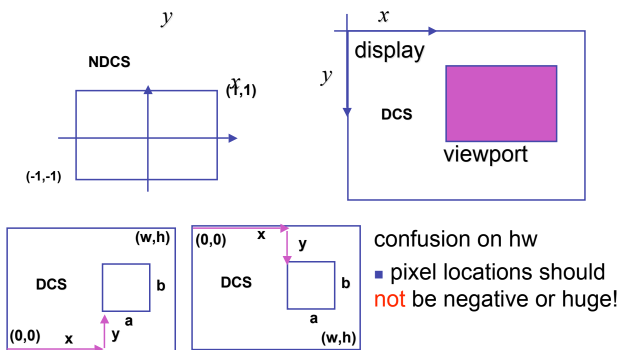
OpenGL Perspective Matrix

- shear, scale, reflect for new coord sys
  - understand derivation from VCS!

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Viewport Transformation

onscreen pixels: map from [-1,1] to [0, displaywidth]

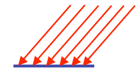


confusion on hw

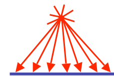
- pixel locations should not be negative or huge!

Light Sources

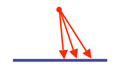
- directional/parallel lights
  - point at infinity: $(x,y,z,0)^T$



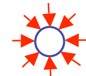
- point lights
  - finite position: $(x,y,z,1)^T$



- spotlights
  - position, direction, angle

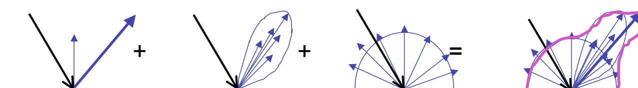


- ambient lights



Reflectance

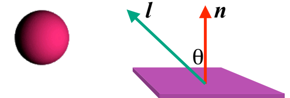
- *specular*: perfect mirror with no scattering
- *gloss*: mixed, partial specularity
- *diffuse*: all directions with equal energy



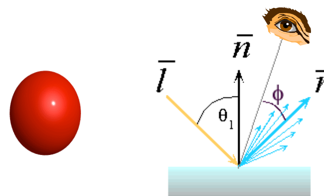
specular + glossy + diffuse =
reflectance distribution

Review: Reflection Equations

$$\mathbf{I}_{\mathbf{diffuse}} = \mathbf{k_d}\,\mathbf{I}_{\mathbf{light}}\,(\mathbf{n} \bullet \mathbf{l})$$



$$\mathbf{I}_{\mathbf{specular}} = \mathbf{k_s}\mathbf{I}_{\mathbf{light}}(\mathbf{v} \bullet \mathbf{r})^{n_{shiny}}$$



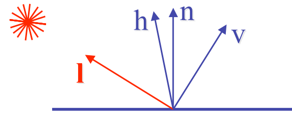
2 (**N** (**N** · **L**)) – **L** = **R**

Review: Reflection Equations 2

- Blinn improvement

$$\mathbf{I}_{\mathbf{specular}} = \mathbf{k}_{\mathbf{s}}\mathbf{I}_{\mathbf{light}}(\mathbf{h} \bullet \mathbf{n})^{n_{shiny}}$$

$$\mathbf{h} = (\mathbf{l} + \mathbf{v})/2$$



- full Phong lighting model
  - combine ambient, diffuse, specular components

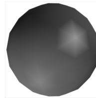
$$\mathbf{I}_{\mathbf{total}} = \mathbf{k}_{\mathbf{s}}\mathbf{I}_{\mathbf{ambient}} + \sum_{i=1}^{\#lights} \mathbf{I}_{\mathbf{i}}(\mathbf{k}_{\mathbf{d}}(\mathbf{n} \bullet \mathbf{l}_{\mathbf{i}}) + \mathbf{k}_{\mathbf{s}}(\mathbf{v} \bullet \mathbf{r}_{\mathbf{i}})^{n_{shiny}})$$

Lighting vs. Shading

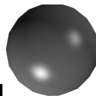
- **lighting**
  - simulating the interaction of light with surface
- **shading**
  - deciding pixel color
  - continuum of realism: when do we do lighting calculation?

Shading Models

- flat shading
  - compute Phong lighting once for entire polygon

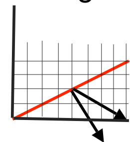


- Gouraud shading
  - compute Phong lighting at the vertices and interpolate lighting values across polygon
- Phong shading
  - compute averaged vertex normals
  - interpolate normals across polygon and perform Phong lighting across polygon



Transforming Normals

- apply nonuniform scale: stretch along x by 2
  - can't transform normal by modelling matrix



- solution:

$$P \rightarrow P' = MP$$

$$N \rightarrow N' = QN$$

$$Q = \left(M^{-1}\right)^{T}$$

normal to any surface transformed by inverse transpose of modelling transformation

Interpolation

Scan Conversion

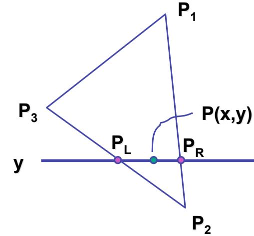
- done:
  - how to determine pixels covered by a primitive
- next:
  - how to assign pixel colors
    - interpolation of colors across triangles
    - interpolation of other properties

Interpolation During Scan Conversion

- interpolate values between vertices
  - z values
  - r,g,b colour components
    - use for Gouraud shading
  - u,v texture coordinates
  - N_x, N_y, N_z surface normals
- equivalent methods (for triangles)
  - bilinear interpolation
  - barycentric coordinates

Bilinear Interpolation

- interpolate quantity along L and R edges, as a function of y
  - then interpolate quantity as a function of x

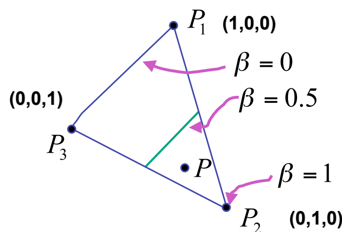


3. Barycentric Coordinates

- weighted combination of vertices

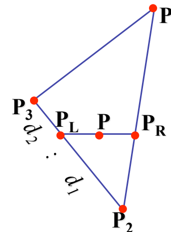
$$\begin{cases} P = \alpha \cdot P_1 + \beta \cdot P_2 + \gamma \cdot P_3 \\ \alpha + \beta + \gamma = 1 \\ 0 \le \alpha, \beta, \gamma \le 1 \end{cases}$$

"convex combination of points"



Computing Barycentric Coordinates

- for point P on scanline



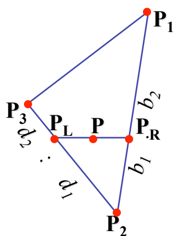
$$P_L = P_2 + \frac{d_1}{d_1 + d_2}(P_3 - P_2)$$

$$= (1 - \frac{d_1}{d_1 + d_2})P_2 + \frac{d_1}{d_1 + d_2}P_3 =$$

$$= \frac{d_2}{d_1 + d_2}P_2 + \frac{d_1}{d_1 + d_2}P_3$$

Computing Barycentric Coords

- similarly



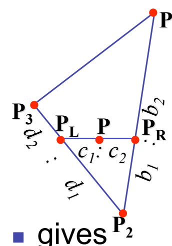
$$P_R = P_2 + \frac{b_1}{b_1 + b_2}(P_1 - P_2)$$

$$= (1 - \frac{b_1}{b_1 + b_2})P_2 + \frac{b_1}{b_1 + b_2}P_1 =$$

$$= \frac{b_2}{b_1 + b_2}P_2 + \frac{b_1}{b_1 + b_2}P_1$$

Computing Barycentric Coords

- combining



$$P = \frac{c_2}{c_1 + c_2} \cdot P_L + \frac{c_1}{c_1 + c_2} \cdot P_R$$

$$P_L = \frac{d_2}{d_1 + d_2}P_2 + \frac{d_1}{d_1 + d_2}P_3$$

$$P_R = \frac{b_2}{b_1 + b_2}P_2 + \frac{b_1}{b_1 + b_2}P_1$$

- gives

$$P = \frac{c_2}{c_1 + c_2}\left(\frac{d_2}{d_1 + d_2}P_2 + \frac{d_1}{d_1 + d_2}P_3\right) + \frac{c_1}{c_1 + c_2}\left(\frac{b_2}{b_1 + b_2}P_2 + \frac{b_1}{b_1 + b_2}P_1\right)$$

Computing Barycentric Coords

- thus

$$P = a_1 \cdot P_1 + a_2 \cdot P_2 + a_3 \cdot P_3$$

with

$$a_1 = \frac{c_1}{c_1 + c_2} \frac{b_1}{b_1 + b_2}$$

$$a_2 = \frac{c_2}{c_1 + c_2} \frac{d_2}{d_1 + d_2} + \frac{c_1}{c_1 + c_2} \frac{b_2}{b_1 + b_2}$$

$$a_3 = \frac{c_2}{c_1 + c_2} \frac{d_1}{d_1 + d_2}$$

Computing Barycentric Coords

- can verify barycentric properties

$$a_1 + a_2 + a_3 = 1$$

$$0 \le a_1, a_2, a_3 \le 1$$