Transformations (Lectures 4-5)

2D rotation

$$\sin(\alpha + \theta) = sin(\alpha)\cos(\theta) + cos(\alpha)\sin(\theta)$$
$$\cos(\alpha + \theta) = cos(\alpha)\cos(\theta) - sin(\alpha)\sin(\theta)$$

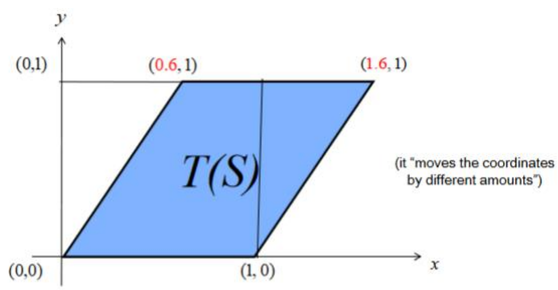
$$x = a\cos(\theta) - b\sin(\theta)$$
$$y = a\sin(\theta) + b\cos(\theta)$$

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}$$

$$R_\theta = \begin{pmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{pmatrix}$$ (2D Rotation matrix encoding)

$$S_{r,s} = \begin{pmatrix} r & 0 \\ 0 & s \end{pmatrix}$$ (2D Scaling)

$$Sh_{x,y} = \begin{pmatrix} 1 & x \\ y & 1 \end{pmatrix}$$ (Shearing)



Affine Maps

Cartesian to homogeneous coordinates:

$$v = \begin{pmatrix} x \\ y \end{pmatrix} => v = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Homogeneous to cartesian coordinates:

$$v = \begin{pmatrix} x \\ y \\ w \end{pmatrix} => v = \begin{pmatrix} x/w \\ y/w \end{pmatrix}$$

Homogenous Transformations

Shearing in 3D results in translation in 2D plane w=1

$$T_{r,s} = \begin{pmatrix} 1 & 0 & r \\ 0 & 1 & s \\ 0 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & r \\ 0 & 1 & s \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + r \\ y + s \\ 1 \end{pmatrix}$$

Conversion back to original dimension

$$\begin{pmatrix} x+r \\ y+s \\ 1 \end{pmatrix} => \begin{pmatrix} \frac{x+r}{1} \\ \frac{y+s}{1} \end{pmatrix} = \begin{pmatrix} x+r \\ y+s \end{pmatrix}$$

<u>Composition of Transformations</u>

1. Translate to origin
2. Rotate around origin
3. Translate back

$$\underbrace{\begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix}}_{\text{translation back}} \underbrace{\begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{\text{rotation around origin}} \underbrace{\begin{pmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{pmatrix}}_{\text{translation to origin}}$$

Applying transformation:

$$\begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix}$$

3D Transformation matrices

2D – 3x3 matrix, 3D – 4x4

$$\begin{pmatrix} e_{11} & e_{12} & e_{13} & e_{14} \\ e_{21} & e_{22} & e_{23} & e_{24} \\ e_{31} & e_{32} & e_{33} & e_{34} \\ e_{41} & e_{42} & e_{43} & e_{44} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix}$$

<u>Homogenous coordinates</u>

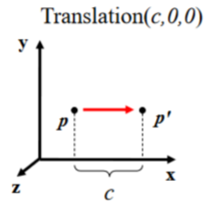
w unchanged when homogenous coordinate is multiplied by an affine matrix

$$\begin{pmatrix} e_{11} & e_{12} & e_{13} & e_{14} \\ e_{21} & e_{22} & e_{23} & e_{24} \\ e_{31} & e_{32} & e_{33} & e_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ w \end{pmatrix}$$

Affine transformations

Translation

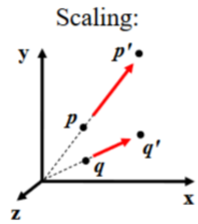
$$\begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x+a \\ y+b \\ z+c \\ 1 \end{pmatrix}$$



$$T(a,b,c) = \begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Scaling

$$\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} ax \\ by \\ cz \\ 1 \end{pmatrix}$$



$$S(r,s,t) = \begin{pmatrix} r & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & t & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad R_y(\theta) = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad R_z(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Change of Orthonormal Basis

- Change of basis is transformations
- Affine transformation often needed

Transformation

- Lines in matrix are the coordinates of the new frame written with respect to the old frame

$$\begin{pmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ n_x & n_y & n_z \end{pmatrix}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} u \\ v \\ n \end{pmatrix} \Rightarrow \mathbf{M}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} u \\ v \\ n \end{pmatrix}$$

Transforming back

$$\mathbf{M}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} u \\ v \\ n \end{pmatrix} \Rightarrow \mathbf{M}^{-1}\begin{pmatrix} u \\ v \\ n \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Inverse: if the transformation is only a rotation

$$M^{-1} = M^T$$

Camera Transformations

Equivalent to change of basis

- Coordinates of the objects in the scene are transformed with respect to a camera frame of reference

Typical parameters: eye position, center, up vectors

Gaze direction $\mathbf{z}$ = ||eye-center||

$\mathbf{x}$ = up × $\mathbf{z}$

$\mathbf{y}$ = $\mathbf{z}$ × $\mathbf{x}$

$$\mathbf{M}_{cam} = \begin{pmatrix} \mathbf{x}.x & \mathbf{x}.y & \mathbf{x}.z & 0 \\ \mathbf{y}.x & \mathbf{y}.y & \mathbf{y}.z & 0 \\ \mathbf{z}.x & \mathbf{z}.y & \mathbf{z}.z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\mathbf{e}.x \\ 0 & 1 & 0 & -\mathbf{e}.y \\ 0 & 0 & 1 & -\mathbf{e}.z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Perspective Camera Transformation

Camera perspective projection in OpenGL:

- Position along z of the "near plane"
- Position along z of the "far plane"
- Eye position at (0,0,0)

Projection along z inside viewing frustrum:

$$f = cotangent\left(\frac{fovy}{2}\right)$$

The generated matrix is

$$\begin{pmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{zFar+zNear}{zNear-zFar} & \frac{2 \times zFar \times zNear}{zNear-zFar} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

zNear – near plane, zfar – far plane

Transformation Properties

- Preserves…

	Rigid	Linear	Affine	Projective
lengths	✓			
angles	✓			
ratios of distances	✓	✓	✓	
parallel lines	✓	✓	✓	
straight lines	✓	✓	✓	✓

Projections

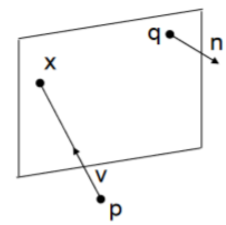
Parallel Projection

Same direction, different origins

Point **p** is projected by direction **v** in a plane **(q,n)**

Parallel projection by direction **v** into a plane(**q**,**n**):

$$\left(\begin{array}{c|c} \begin{pmatrix} \mathbf{v}\cdot\mathbf{n} & 0 & 0\\ 0 & \mathbf{v}\cdot\mathbf{n} & 0\\ 0 & 0 & \mathbf{v}\cdot\mathbf{n}\end{pmatrix} - (\mathbf{v}\mathbf{n}^T) & (\mathbf{q}\cdot\mathbf{n})\mathbf{v} \\ \hline 0\quad 0\quad 0 & \mathbf{v}\cdot\mathbf{n}\end{array}\right)$$



Perspective Projection

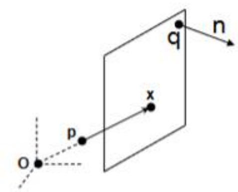
Same origin, different directions

Assumes camera “eye’ is at the origin

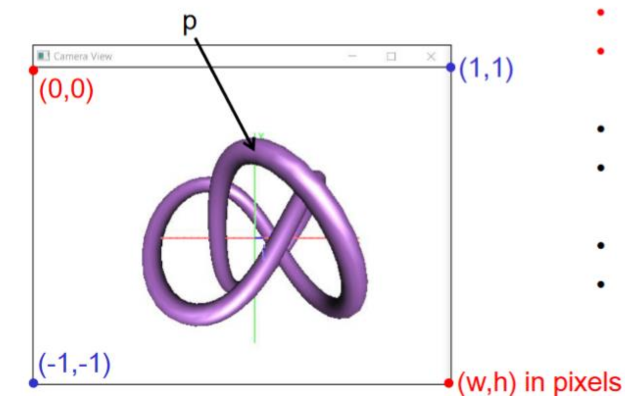
For each point **p**, the direction of projection is **p** (line from the origin to **p**)

- Perspective projection into a plane(**q**,**n**):

$$\left(\begin{array}{c|c} \mathbf{I}(\mathbf{q}\cdot\mathbf{n}) & 0\\ \hline \mathbf{n}^T & 0\end{array}\right)$$



Picking



- 500/1000 = 0.5
- 250/1000 = 0.25

- 0.5 – 0.5 = 0.0
- (1 - 0.25) – 0.5 = 0.25

- 0.0 * 2 = 0.0
- 0.25 * 2 = 0.5

Corresponding ray going into scene

P = (500, 250) – pixel coordinates you would receive in callback function

Assuming window of 1000x1000

Normalized p = (0.0, 0.5)

1. normalize p (as above)
2. compute ray endpoints
   Recall viewing transformation: $M = P\, M_{cam}$
   Given p in normalized 2D window coordinates:
      2.1 Ray $p1 = M^{-1}$ (p.x, p.y, near plane z)
      2.2 Ray $p2 = p1 - eye$
3. intersect ray with all objects (triangles) in the scene, return the object with the closest intersection to the eye position of the camera

Triangulation

Ear Triangulation

Simple implementation: $O(n^3)$ time

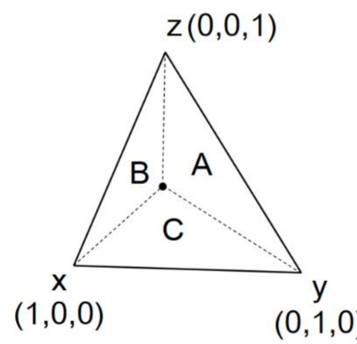
Track convex and concave list: $O(n^2)$ time

Polygon stored as a list of array or vertices

Perform CCW and intersection test

- CCW test done by cross product
- Continue finding candidates until list is empty

Barycentric Coordinates



$p = \alpha x + \beta y + \gamma z$
$\alpha + \beta + \gamma = 1$

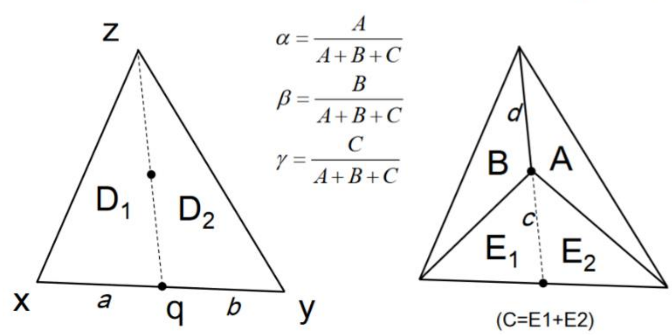
Triangle interpretation:

- barycentric coordinates describe **p** with respect to the triangle
- Triangle becomes its “frame of reference”

Sub-areas relations

$$\alpha = \frac{A}{A+B+C}, \qquad \beta = \frac{B}{A+B+C}, \qquad \gamma = \frac{C}{A+B+C}$$

$D \ = \ D1 + D2, D1 \ = aD, D2 = bD, (a + b = 1)$
$C \ = \ E1 + E2, E1 \ = \ cD1, B = dD1, (c + d = 1)$
$E2 \ = \ cD2, A = dD2$
$A = dbD, B = daD, C = c(a + b)D = cD$



Computation by Cramer’s Rule

$$\begin{pmatrix} x_x & y_x & z_x \\ x_y & y_y & z_y \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix} \qquad \alpha = \frac{\begin{vmatrix} \mathbf{p} & \mathbf{y} & \mathbf{z} \\ 1 & 1 & 1 \end{vmatrix}}{\begin{vmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \\ 1 & 1 & 1 \end{vmatrix}}, \qquad \beta = \frac{\begin{vmatrix} \mathbf{x} & \mathbf{p} & \mathbf{z} \\ 1 & 1 & 1 \end{vmatrix}}{\begin{vmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \\ 1 & 1 & 1 \end{vmatrix}},$$

$$\gamma = \alpha - \beta - 1.$$

Light

Achromatic light

- One intensity/brightness value representation (ex: 0 is black, 255 is white)
- We perceive intensities in a nonlinear way
- Intensity levels should be spaced logarithmically to achieve equal steps in brightness
- CRT monitor behavior is also not linear
    - Gamma correction
- How many intensities are enough? – A B&W image can use as many gray levels as needed
- Halftone approximation – Many gray levels perceived from fewer gray levels

Color representation

How many colors? Depends on:

- Storage requirements (memory)
- Processing time requirements (speed)
- Frame buffer manipulation algorithms
- Limitations of the monitor and computer

Working with colors

- Lookup tables: represent colors as indices to a color table
- Quantization: reduce # of colors used while minimizing perception of color changes
- Dithering: simulate many colors from few ones

Common to use 3x8=24 bit representation per r,g,b color **(true color representation)**

- 32-bit descriptions will include alpha channel (4x8=32) (most often used to define a transparency level)

RGB representation

- 3 types of cones in the retinas with different sensitivities to a different wavelengths
- So we can visually match a given color by additively mixing 3 colored lights
- Ex: C = rR + gG + bB

Chromatic color

<u>Hue</u>: color itself

<u>Saturation</u>: amount of white mixed in the color ex:

- Royal blue: highly saturated
- Sky blue: unsaturated

<u>Lightness</u>: intensity (darker or brighter colors)

<u>Brightness</u>: perceived intensity of a self-luminous object

Computer models

- Monitors designed to process RGB components
- Other computer representations exist for hardware needs and design purposes
    - Most often converted to RGB in the end

HSV (aka HSL)

User-oriented format

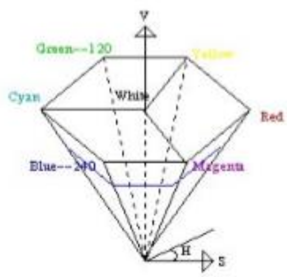
Intuitive tint, shade and tone parameters

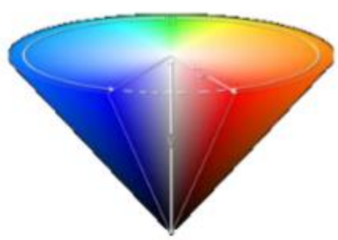
Often chosen in color input dialogs

Good for color interpolation – Hue interpolated independently from other values

Hue, saturation, value

- Hue: red, yellow, green, etc.
- Saturation: intensity of a specific hue (intense vs. dull)
- Value: lightness (light vs dark, or white vs black)





RBG to HSV

- Convert r,g,b in [0,1], to h,s,v; h in [0,360], s,v in [0,1]

```
max = Max(r,g,b)
min = Min(r,g,b)
v = max;            // max gives the dominant color
if ( max == min ) { s=0; h=0; return; }
s = (max-min)/max; // saturation: how "dominant" v is

delta = max-min;
if ( r == max )       // color is between magenta and yellow
  { h = (g-b)/delta; if(h<0)h+=6; } // h defines 60deg sector
else if ( g == max ) // color is between yellow and cyan
  { h = 2+(b-r)/delta; }
else if ( b == max ) // color is between cyan and magenta
  { h = 4+(r-g)/delta; }

h *= 60; // convert sector to degrees
```

Illumination and shading

Surfaces are shaded following illumination models

- For each point in a surface, its final color must be computed according to the illumination model before it can be painted in the image buffer during rasterization

Lights and material must be declared first

Illumination models

- <u>Local models</u>: interaction b/n individual points in a surface and light sources
    - Very fast, but don't automatically account for refractions, reflections, and shadows
- <u>Global models</u>: interchange of light between all surfaces
    - Ex: ray tracing
    - Realistic but slower
- Set the color of a surface point according to light and surface properties

Shading

- Applies an illumination model to several pixels
- Colors and brightness vary smoothly across a surface using interpolation methods

Phong Illumination Model

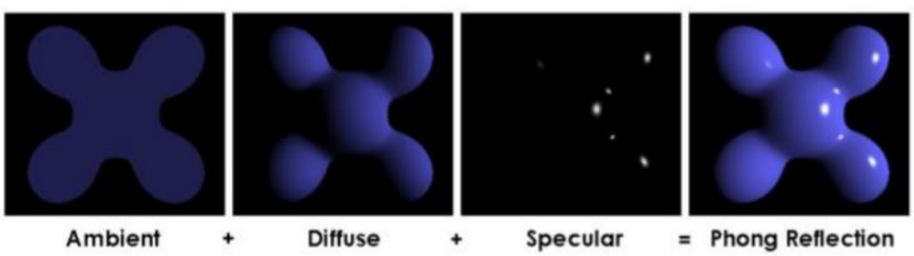
Local model

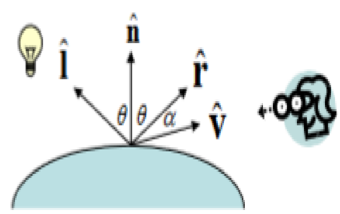
$$I = I_a k_a + I_d k_d (\widehat{l} \cdot \hat{n}) + I_s k_s (\hat{v} \cdot \hat{r})^f$$

<u>Ambient light</u>: $I = I_a k_a$
$k_a \in [0,1]$

<u>Diffuse Reflection</u>: $I = I_d k_d \cos(\theta) = I_d k_d (\hat{l} \cdot \hat{n})$
$k_d \in [0,1], \theta \in [0,90]$

<u>Specular Reflection</u>: $I = I_s k_s \cos(\alpha)^f = I_d k_d (\hat{v} \cdot \hat{r})$
$f \geq 0$ (larger f = smaller specular highlights)





Parameters

- <u>Light intensities</u>: for ambient, diffuse, and specular reflection: I_a, I_d, I_s
- <u>Material coefficients</u>: for each type of light intensity: k_a, k_d, k_s
- <u>Scene parameters</u>: involving unit vectors: light direction, surface normal, reflect ray, viewer direction: $\hat{l}, \hat{n}, \hat{r}, \hat{v}$

Extensions

Emissive light

- Emissive intensity constant k_e
- Similar to ambient color but doesn't depend on the color of light
- Models an amount of emitted light

$$I = k_e + I_a k_a + I_d k_d (\hat{l} \cdot \hat{n}) + I_s k_s (\hat{v} \cdot \hat{r})^f$$

Light-source attenuation

- w/ the eq. seen so far, two parallel surface of the same material, no matter their distance to the light source, will have the same diffuse value

$$I = I_a k_a + f_{att} I_d k_d (\hat{l} \cdot \hat{n}) + I_s k_s (\hat{v} \cdot \hat{r})^f$$
$$f_{att} = \frac{1}{d_L^2}$$

Colored lights

Illumination model is a combo of **light properties** and **material properties**:

- <u>light intensities</u> *(I)* determined per component r,g,b
- <u>material properties</u> (*k*) also per component

$$I^R = k_e{}^R + I_a{}^R k_a{}^R + I_d{}^R k_d{}^R (\hat{\mathbf{l}} \cdot \hat{\mathbf{n}}) + I_s{}^R k_s{}^R (\hat{\mathbf{v}} \cdot \hat{\mathbf{r}})^f$$

$$I^G = k_e{}^G + I_a{}^G k_a{}^G + I_d{}^G k_d{}^G (\hat{\mathbf{l}} \cdot \hat{\mathbf{n}}) + I_s{}^G k_s{}^G (\hat{\mathbf{v}} \cdot \hat{\mathbf{r}})^f$$

$$I^B = k_e{}^B + I_a{}^B k_a{}^B + I_d{}^B k_d{}^B (\hat{\mathbf{l}} \cdot \hat{\mathbf{n}}) + I_s{}^B k_s{}^B (\hat{\mathbf{v}} \cdot \hat{\mathbf{r}})^f$$

Shading

Flat Shading

- Normal of each polygon face used to illuminate the entire face
- Discontinuous shading occurs at edges between the flat faces
- Not for smooth surfaces such as a sphere

Smooth Shading

- Correct normal vectors are needed to achieve correct smooth shading results
- Then apply gouraud or phong shading

Gouraud Shading

Smooth shading without computing illumination on every point inside a triangle

- illuminate triangle vertices, and obtain 3 colors
- interpolate the 3 colors inside the triangle
- problem: specular reflection in the middle of a triangle are missed

Phong Shading

Interpolate normal from the given pre-vertex normal for each interior point

- each interior point will have a different normal
- phong illumination applied to every interior point using the interpolated normal
- fixes specular reflection (but not outer border polygonal appearance)

Normals “reconstruct the ideal surface)

Defining Normals for Phong

Normal are defined per vertex

- computed normal will define if vertices are
    - in segments supposed to be edges
    - in the middle of a smooth surface
- automatic generation of normal is possible and important
    - can manually define normal vectors
    - file formats may give lists of normal per vertex, per face, etc.
- back-face culling optimization
    - don’t render triangles with normal pointing away from viewer

Transformations and Illumination

non-rigid transformations may not preserve angles!

A transformed normal vector may not be normal to its corresponding transformed surface anymore (use the transposed inverse)

Do not mix...

- Phong illumination model (an equation), with
- Phong shading model (when all interior points

Computing smooth normals

Different methods can be used to determine the normal of a vertex from the normal of the triangles sharing the vertex:

- Weight uniformly: take the average
- Weight by area
- Weight by inverse area
- Plane fitting to shared vertices
- Weight by angle

Textures and Other Mappings

- Texture mapping – improves realism
- Reflection mappings – way to display reflections in real-time
- Light maps – lighting effects from texture mapping
- Bump maps and displacement maps – to improve geometry detail appearance

Mapping techniques usually involves mapping a 2D data array onto the surface of a 3D object

Texture Mapping

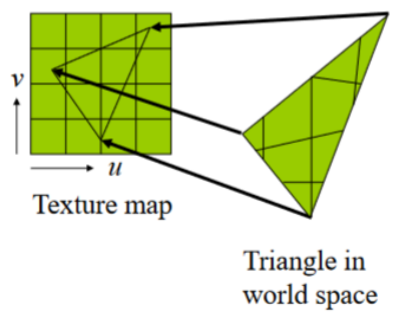
Extension to Phong's model

- rewrite the diffuse component I_d as a function of the texture map
- aka "color mapping"

Problem – for every triangle to be rendered, need to define how to map the triangle to the texture image

in OpenGL – each vertex associated with (u, v) texture coordinates

- texture coordinates are always in [0,1]

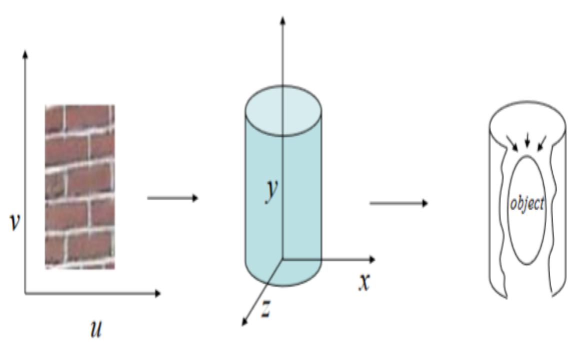


Mapping using intermediate surface

- first map to an "easy" surface
- then map to the final object

ex: cylinder mapping

$(\theta, h) \to (u, v): (u, v) = (\frac{\theta}{\theta_{max}}, h/\ h\_max)$



Spherical Mappings

- texture mapping can be used to alter some or all of the constraints in the illumination equation: diffuse color, alter the normal, etc.
- GLSL uses keyword **sampler** to access the colors in the texture uniform sampler2D texId;

Difficulties

- Textures don't define geometry
  - Think of as color mappings
  - Light may appear incorrectly
  - Aliasing effects
- Aliasing is the under-sampling of a signal, and it's especially noticeable during animation
  - Details may pop in and out of view
  - Solution: mipmapping

Mipmapping

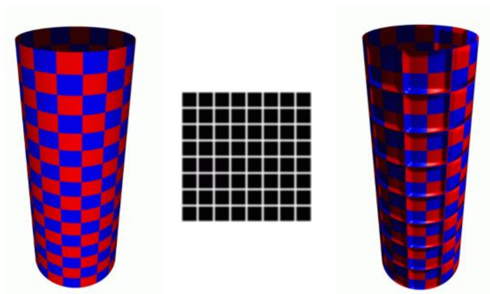
- Original high-resolution texture map is scaled and filtered into multiple resolutions before being applied to a surface
- Texture can appear in full detail if it's seen in a close-up, or can be rendered quickly and smoothly from a lower MIP level when the object appears smaller of further away
- Usually, powers of 2 are used for the MIP map levels (1024x1024 -> 512x5125 -> 256x256 -> etc.)

Billboards

Texture map is treated as a 3D object in a scene

- It's not mapped to a solid object surface
- It's just a plane w/ the image
- The plan rotates to always face the viewer
- Parts of the texture can be transparent

Bump mapping



Heightmap buffer w/ perturbations to be applied to the surface normals, creating the effect of a different surface

New normal are used for illumination (contrary to simple textures)

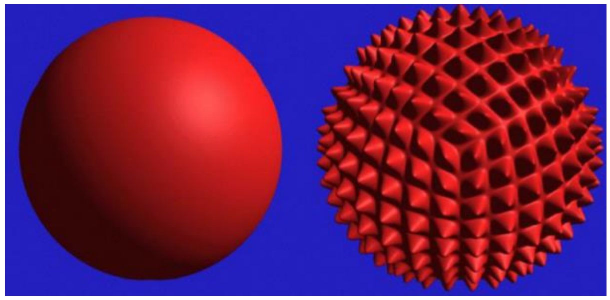
The geometry doesn't change

Limitations

- It's impossible to create bumps on the silhouette of a bump-mapped object
- Bump maps don't resolve self-occlusion

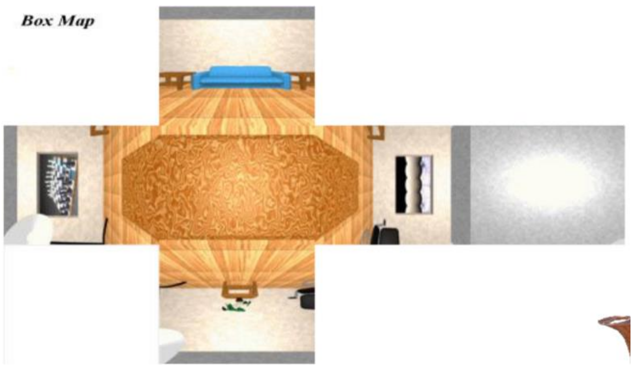
Displacement maps

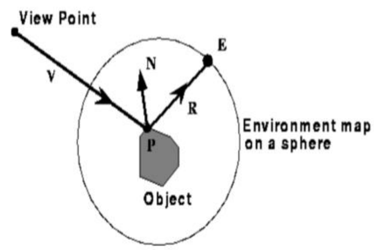
- Use the texture map to actually move the surface point
- The geometry must be displaced before visibility is determined
- Does change geometry



Environment mapping

Can simulate reflections by using the direction of the reflected ray to index a spherical texture map at "infinity"





Light maps

First use a global rendered to realistically render the scene with lights, shadows, etc.

Then use the image as a color map in the illumination of real-time environments – Popular in games

VISBILE-SURFACE DETERMINATION

Algorithms may try to exploit "coherences" (face, frame, etc.)

Spatial organization/ subdivision may also be used

Only compare objects projected on a same cell, etc.

Generic solutions should work for "triangle soups"

Algorithms to determine which polygons go in front and which go behind during rasterization

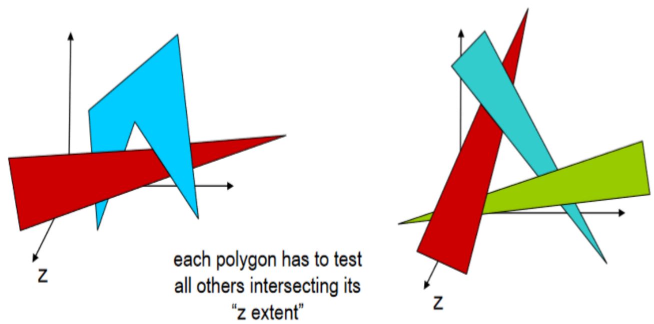
Back-face culling

- Allows to only draw a polygon if its normal is facing the camera
- Significant optimization

Depth sorting/Painter's Algorithm

1. Sort polygons according to z coordinate

2. Resolve ambiguities when polygon's z extends overlap, split polygons if needed (some scenes don't need this step)
3. "paint" polygons in order, starting from the furthest ones from the camera



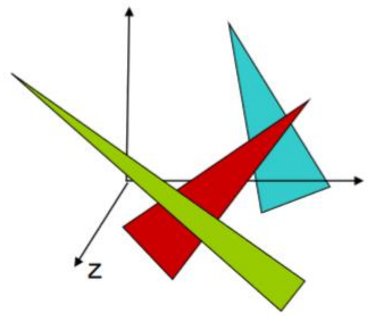


Image above needs 2nd step image above can skip 2nd step

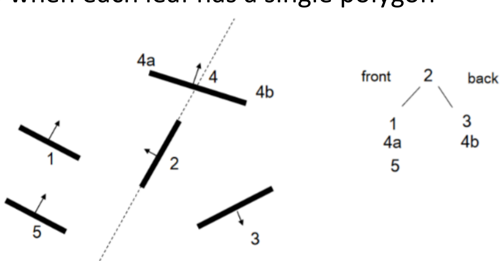
Binary Space Partition (BSP) Trees

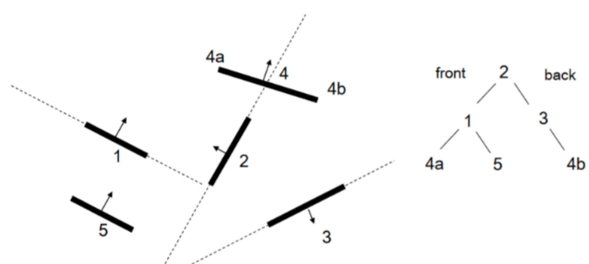
1. Build a tree "ordering" the polygons according to "half spaces separability"
2. Traverse the tree according to viewpoint

Works in 2D and 3D

BSP Tree construction

1. Choose a separation plane passing by a polygon and classify other polygons in front or back spaces; subdivide if needed
   note: normal vector of a polygon is used to define its "front"
2. Repeat for each unprocessed "front and back space"
3. Stop when each leaf has a single polygon

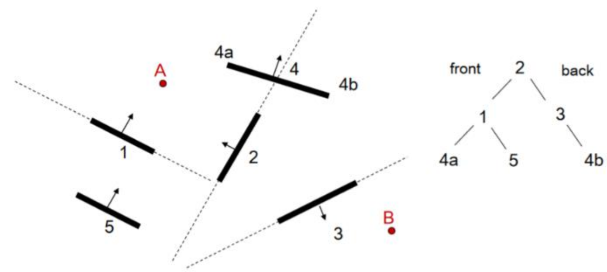




(steps 1-2)

BSP Tree Traversal

- Tree constructed at pre-processing
- Viewpoint can move but not polygons
    - Tree must be reconstructed/updated every time a polygon moves
- Need to know “front” and “back” of polygons
    - Easy determination based on normal



A: 3, 4b, 2, 5, 1, 4a

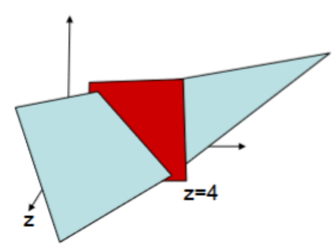
B: 5, 1, 4a, 2, 4b, 3

Pseudocode:

```
traverse ( node )
if ( node is null ) return;
if viewer in node's front half-space

    1. traverse ( node->back );
    2. display node;
    3. traverse ( node->front );

else

    1. traverse ( node->front );
    2.  display node;
    3. Traverse ( node->back );
```

(Recursively process children of node)

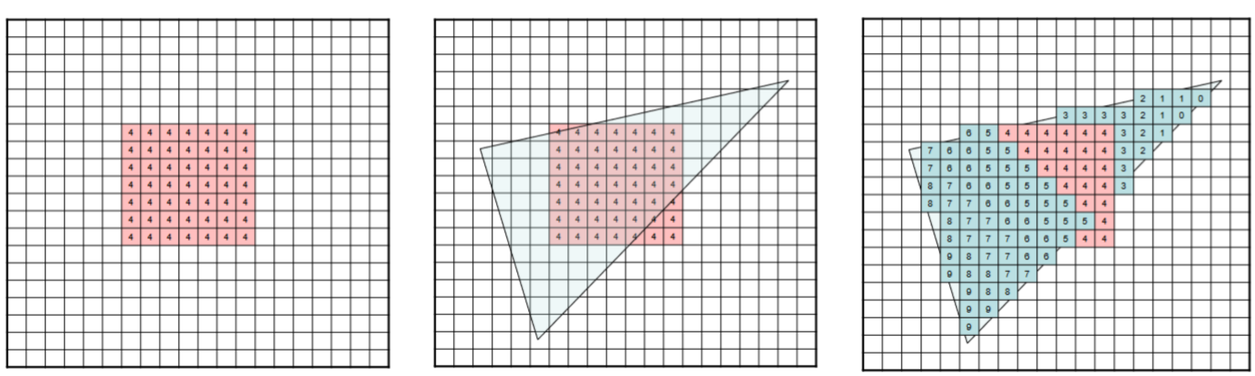
Z-Buffer



No sorting, rasterize polygons in any order

But maintain a buffer

- Each pixel in the buffer stores the z-value of the respective pixel in the image
- New pixels are only displayed if their z-values are greater (closer to viewer) than previous values



Very fast, but uses some memory (but memory is cheap)

OpenGL uses Z-buffer

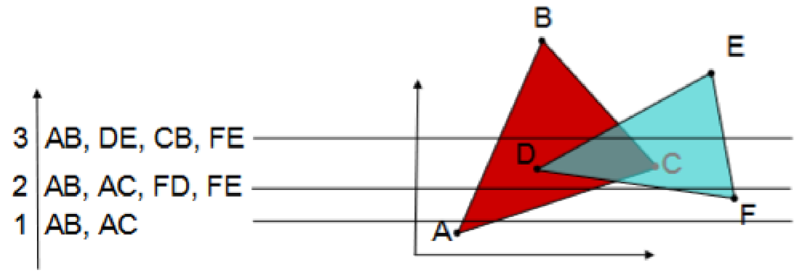
Scan-line

“scan-line” algorithms represent an efficient and generic approach to process polygons

The scan line will change “its properties” when the next event happens

- Events are new vertices encountered on the Y direction
- All events are sorted vertically as pre-computation

Tables are used to describe events are the current scan line



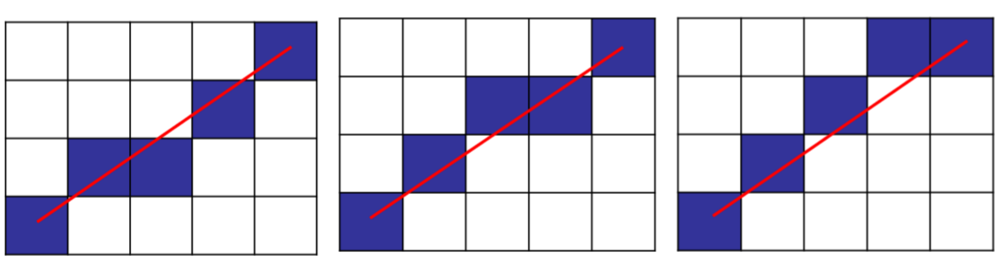
Ex:

```
for (each scan line)
{       update active surface table;

        for (each pixel on scan line)
        {       determine surfaces in active surface table

                        projecting to current pixel;
                find closest surface among them;
                paint pixel;

        }

}
```

RASTERIZATION

Scan converting lines

- Determine the sequence of pixels that lie as close to the ideal lien as possible
- No gaps, best approximation, consistency, etc.
- Same principle applies to other primitives such as circles, polylines, and rectangles



Incremental algorithm / DDA (digital differential analyzer)

- Given endpoints, compute slope m
- Increment x by 1, starting with leftmost point
- Calculate $y_i = mx_i + B$
- Paint pixel (x_i, round(y_i))

```
void line ( int x0, int y0, int x1, int y1 )
   int x;
   float deltay = y1 - y0;
   float deltax = x1 - x0;
   float m = deltay / deltax;
   float y = y0;

   for ( x = x0; x <= x1; x++ )
    { paint ( x, round(y) );  // round(y): int(y+0.5f), y>0
      y = y + m;
    }
```

- Inefficient: too many floating-point operations
- Ok for most (short) lines, but can accumulate error
- Needs floating-point operations

Bresenham

- Uses only integer arithmetic
- No round function, incremental calculation
- Applicable to circles, but not conics
- Best fit, minimizes error

Midpoint Algorithm

```
void midpointline ( int x0, int y0, int x1, int y1 )

int deltax = x1-x0;
int deltay = y1-y0;
int d = deltay+deltay - deltax; // initial value of d (2dy-dx)
int incE = deltay+deltay;        // increment to move to E (2dy)
int incNE = deltay+deltay-deltax-deltax; // inc to move to NE (2dy-2dx)
x = x0;
y = y0;

paint ( x, y );                  // first point

while ( x<x1 )
 { if ( d<0 )
    { d = d + incE;              // great, only integer arithmetic !!!
      x = x + 1;
    }
   else
    { d = d + incNE;
      x = x + 1;
      y = y + 1;
    }
   paint ( x, y );               // paint current point
 }
```

Problems:

Endpoint order

- Ensure that p0, p1 and p1, p0 generates same pixels
  - Change choice used when d = 0 or
  - Switch endpoints to ensure same result

So far, we considered integer endpoints

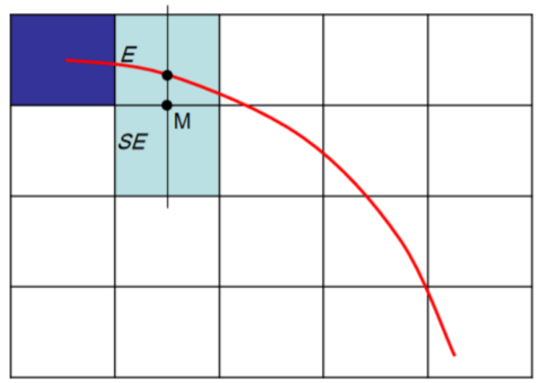
- Closest pixel from real points can be used
- Additional care needed when drawing clipped lines, to ensure the slope remains the same

Scan converting Circles

Circle has eight-way symmetry

CirclePaint(x,y)

Paint (x, y);
Paint (y, x);
Paint (y, -x);
Paint (x, -y);
Paint (-x, -y);
Paint (-y, -x);
Paint (-y, x);
Paint (-x, y);



Same logic as midpoint line algorithm

Scan converting polygons

Scan line

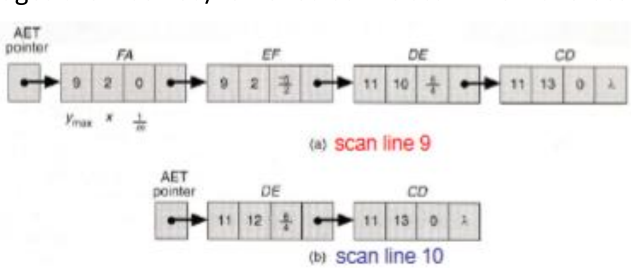
- Computes spans that lie b/n left and right edges of the polygon
- Handles convex and concave polygons
- Same midpoint technique used to calculate and update the extremes of the spans
    - No need to calculate analytically the intersections b/n the polygon edges and the scan line

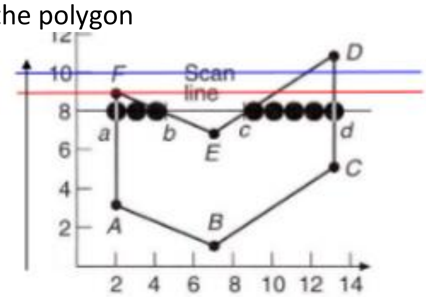
Spans are filled in 3 step process

1. Find scan line intersections with polygon edges
2. Sort intersections by increasing x
3. Fill pixels using the odd-parity rule:
   - Initially even
   - Invert on each intersection
   - Only draw when odd

Data structure – active-edge table (AET)

- Edges sorted on their x intersection values
- Edges are inserted/removed as the scan line traverses the polygon





- global Edge Table ET containing all edges sorted by their smallest y coordinate is used to add edges

    - Use Bucket sort, buckets are the number of scan lines
    - within each bucket, edges are in increasing x order of the lower endpoint
- edge table:
    - 1 bucket for each scan line, sorted by smallest y

Problems

- Horizontal edges
- Silvers
- Calculating the intersections
- Exploiting edge coherence

Conex polygons

Simpler to deal with these

- Easier management of scan lines
- Triangles even simpler

How to decompose arbitrary polygons in convex pieces

- Scan line algorithm for trapezoidal decomposition
- Polygon triangulation methods
    - Optimal method is $O(n)$
- Simplest approach: triangulation by "ear cuts"
    - $O(n^2)$