Lighting and Shading

Learn to shade objects so their images appear three-dimensional

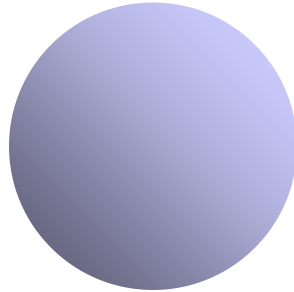
Introduce the types of light-material interactions

Build a simple reflection model---the Phong model--- that can be used with real time graphics hardware

Work on fragment shaders for different types of lighting

Shading

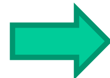
Why does the image of a real sphere look like



Light-material interactions cause each point to have a different color or shade

Need to consider

- Light sources
- Material properties
- Location of viewer
- Surface orientation



4 Key elements of image formation

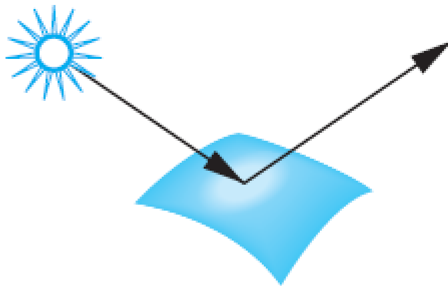
E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

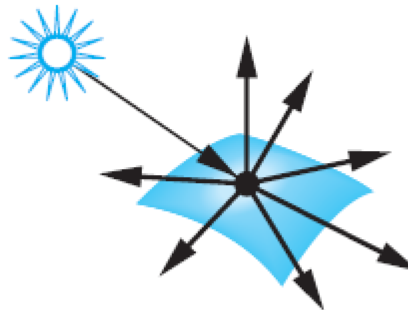
Light-Material Interaction

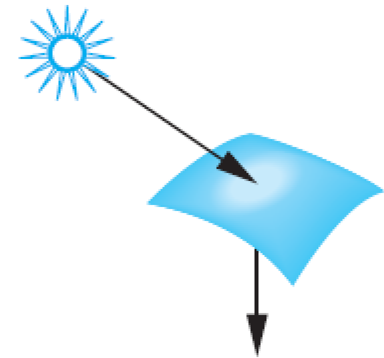
Specular surface

Diffuse surface

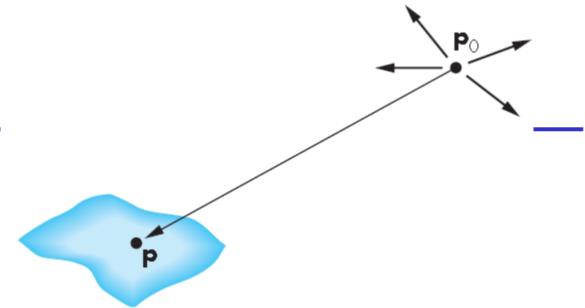
Translucent surface







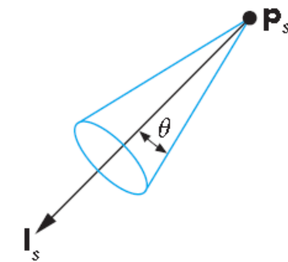
Recall Simple Light Sources



Point source

- Emits light equally in all direction
- Model with position and color – proportional to the inverse square of the distance
- Distant source = infinite distance away (parallel)

Spotlight



- Restrict light from ideal point source

Ambient light

- Uniform illumination everywhere in scene -- An intensity identical at every point

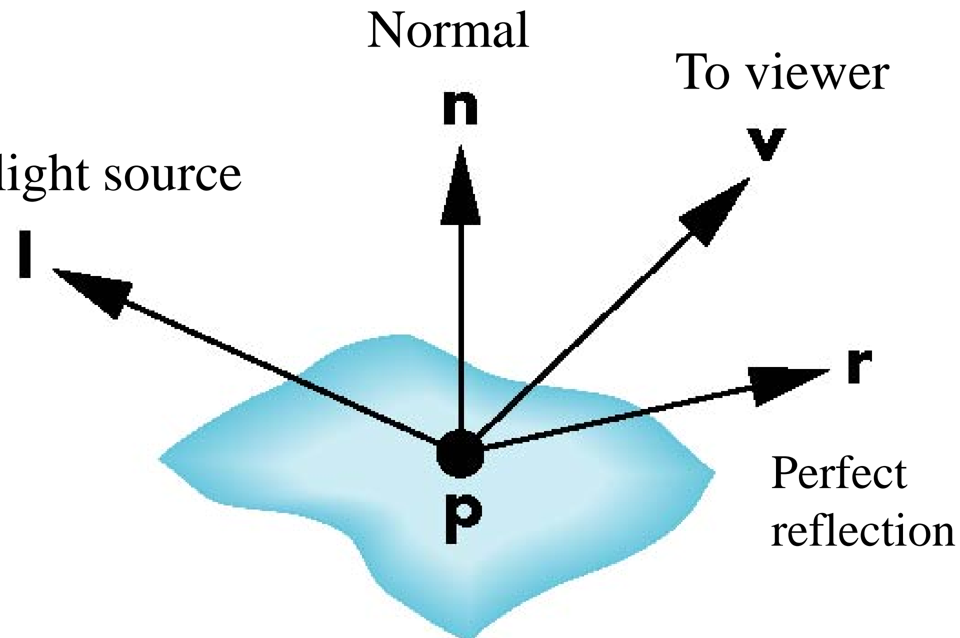
Phong Model

Uses four unit vectors to calculate a color on a surface

- Surface normal **n**
- To viewer **v**
- To light source **l**
- Perfect reflector **r**

Each light source has three components

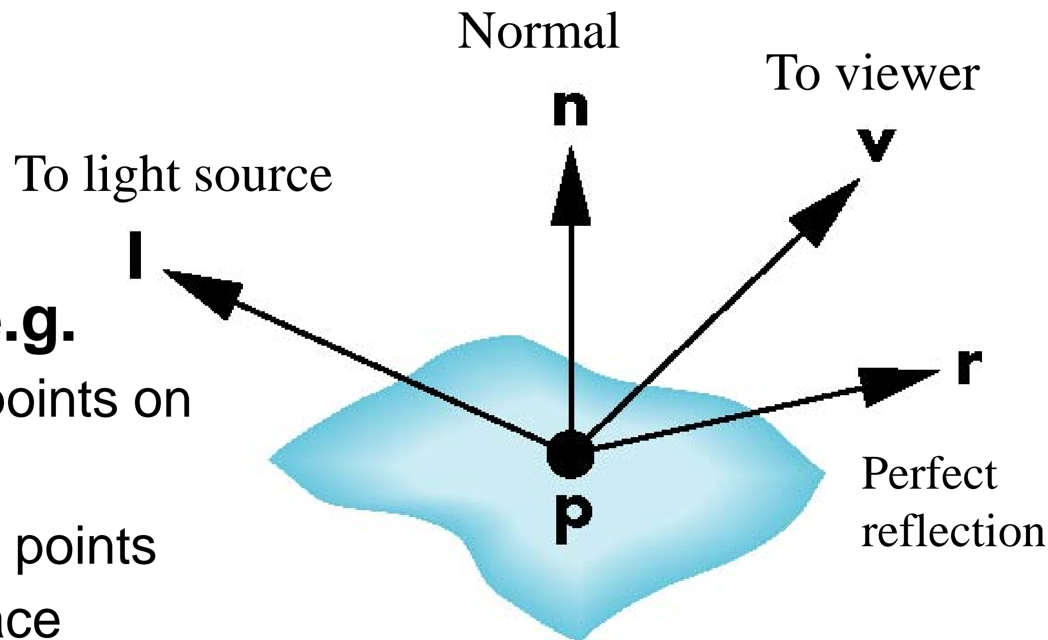
- Ambient
- Diffuse
- Specular



Computation of Vectors

Need to compute the four vectors

- Surface normal **n**
- To viewer **v**
- To light source **l**
- Perfect reflector **r**



E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

Simplifications can apply, e.g.

- Normal can be the same for all points on a flat polygon
- Light direction is the same for all points if the light is far away from the surface

How to tell if the light is far way?

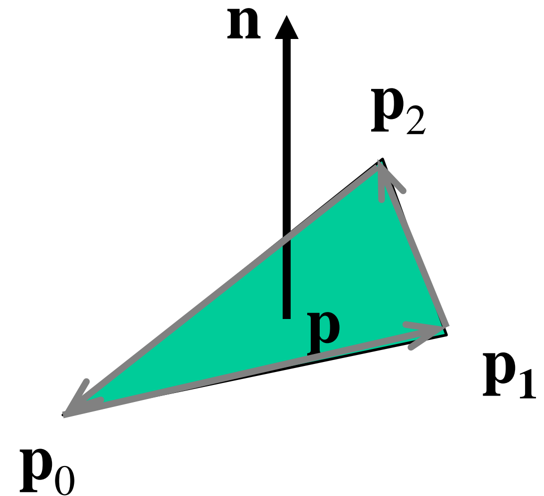
W=0

Computation of Vectors

$\mathbf{l}$ and $\mathbf{v}$ are specified by the application

$\mathbf{h}$ can be computed from $\mathbf{l}$ and $\mathbf{v}$

How to calculate $\mathbf{n}$?



Depending on surface

E.g., given three noncolinear points, e.g., the three vertices of a triangle P_1, P_2, and P_3, the outfacing normal can be obtained by

$$\mathbf{n} = \frac{(P_2 - P_1) \times (P_0 - P_2)}{|(P_2 - P_1) \times (P_0 - P_2)|}$$

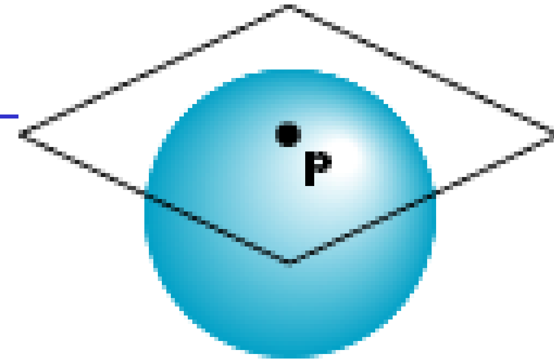
Order of vectors is important!

Normal to Sphere

How we compute normals for curved surfaces?

Depend on how we model the surface.

Normal is given by gradient



E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

$$\mathbf{n}' = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{bmatrix} \text{(Implicit form) or} \quad \mathbf{n}' = \frac{\partial \mathbf{p}}{\partial u} \times \frac{\partial \mathbf{p}}{\partial v} \text{ (Parametric form)}$$

$$\Rightarrow \mathbf{n} = \frac{\mathbf{n}'}{|\mathbf{n}'|}$$

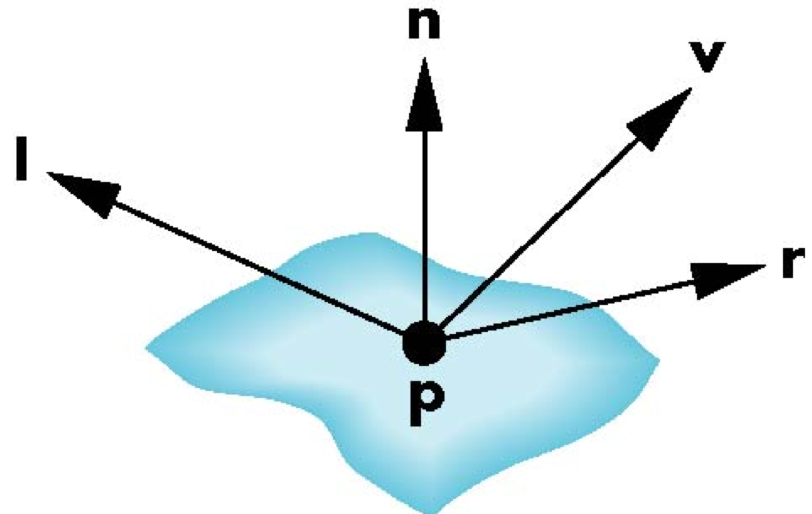
Blinn-Phong Model

For each light source and each color component, the Blinn-Phong model can be written as

$$I = \frac{k_d L_d max(\mathbf{l} \cdot \mathbf{n}, 0) + k_s L_s \max\left((\mathbf{n} \cdot \mathbf{h})^{\beta}, 0\right)}{a + bd + cd^2} + k_a I_a$$

For each color component we add

contributions from all sources



E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

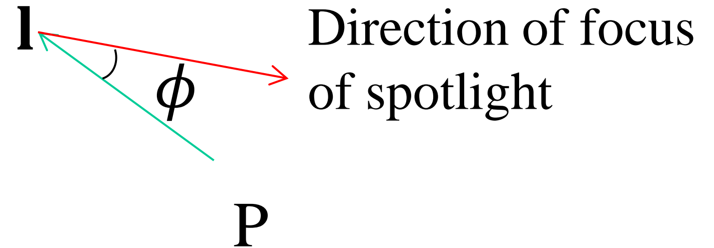
Other Issues

Point light source and a distant light source

- If w =1.0 -- a regular point light source at a finite location
- If w =0.0 – a distant light source at infinity = a parallel source with the given direction vector

Spotlights derived from point light source

- Angle between **l** (direction to the light) and the focus of spotlight
- Cutoff angle θ
- Attenuation proportional to $cos^{\alpha}\phi$



Emission term

- A *light source* allowed in the scene, e.g., the moon
- Unaffected by any other sources
- Not affect other surfaces

Per-vertex Lighting vs Per-fragment Lighting

Per-vertex lighting

- Lighting is handled in vertex shader
- Color is computed for each vertex, then interpolated for each pixel
- Efficient, but rough

Per-fragment lighting

- Lighting is handled in fragment shader
- Color is computed for each fragment (potential pixel)
- Sophisticated, but slow

Polygonal Shading

In per vertex shading, shading calculations are done for each vertex

- Vertex colors become vertex shades and can be sent to the vertex shader as a vertex attribute
- Alternately, we can send the parameters to the vertex shader and have it compute the shade

Smooth shading (default), vertex shades are interpolated across an object if passed to the fragment shader as a varying variable

Flat shading: use uniform variables to shade with a single shade

Polygonal Shading – Flat Shading

Need to calculate $\mathbf{n}, \mathbf{v}, \mathbf{l}$ for every point on a surface

Simplifications:

- $\mathbf{n}$ is a constant for a flat polygon and can be precomputed
- $\mathbf{v}$ is a constant for a distant viewer
- $\mathbf{l}$ is a constant for a distant light

If all the three vectors are constants, the shading calculation can be done once for each polygon

⟹ Every point has the same color/shade on the polygon

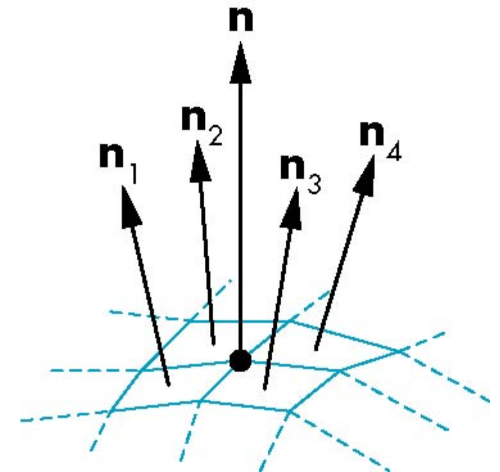
Smooth Shading - Gouraud Shading (Per-vertex)

Gouraud used the average of the normals around a mesh vertex

Gouraud Shading

- Find average normal at each vertex
- Apply modified Phong model at each vertex
- Interpolate vertex shades across each polygon

$$\mathbf{n} = (\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4) / |\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4|$$



Phong Shading (Per-fragment)

- Find average vertex normals $\mathbf{n}_A$ and $\mathbf{n}_B$
- Interpolate vertex normals $\mathbf{n}_C$ and $\mathbf{n}_D$

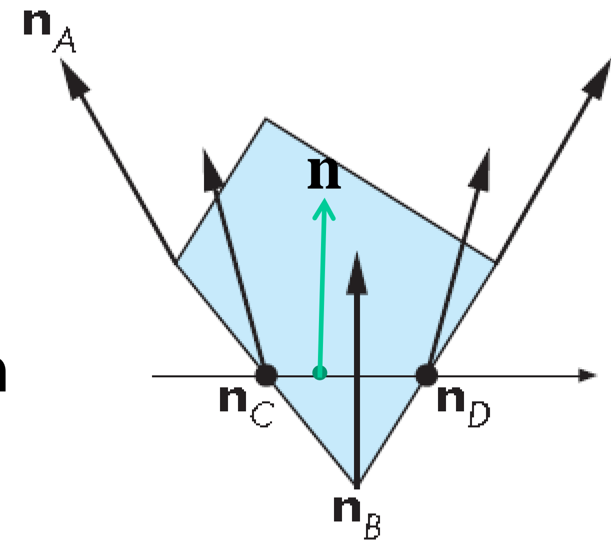
across edges

$$\mathbf{n}_C(\alpha) = (1-\alpha)\mathbf{n}_A + \alpha\mathbf{n}_B$$

- Interpolate edge normals across polygon

$$\mathbf{n}\,(\alpha,\beta) = (1-\beta)\mathbf{n}_C + \beta\mathbf{n}_D$$

- Apply modified Phong model at each fragment



From Vertices to Fragments

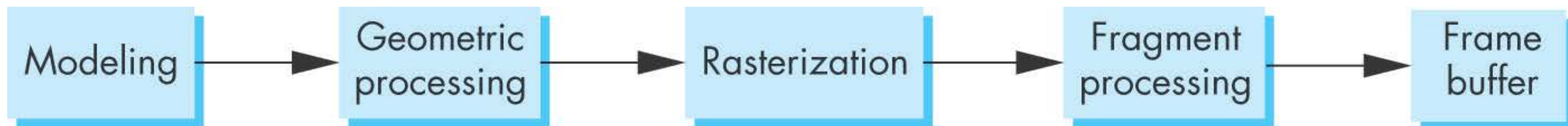
Assign a color to every pixel

Pass every object through the system

Required tasks:

- Modeling
- Geometric processing
- Rasterization
- Fragment processing

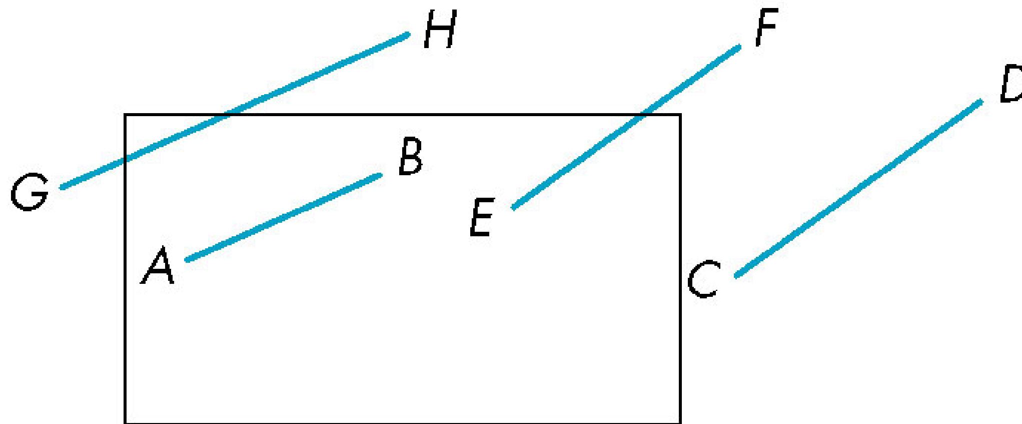
} clipping



Clipping 2D Line Segments

Brute force approach: compute intersections with all sides of clipping window

- Inefficient: one division per intersection



Cohen-Sutherland Algorithm

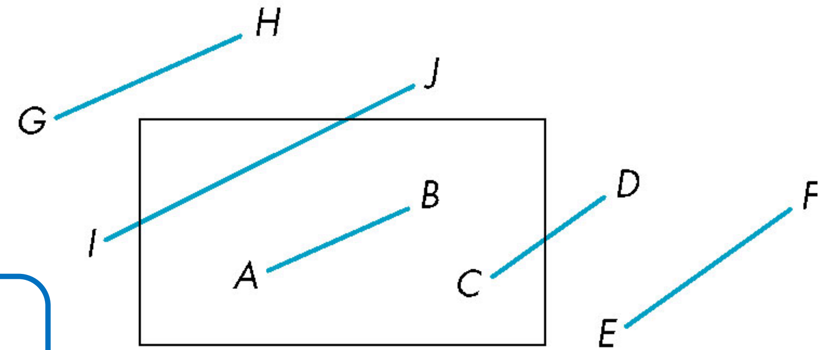
Idea: eliminate as many cases as possible without computing intersections

For each endpoint, define an outcode $b_0 b_1 b_2 b_3$

$b_0 = 1$ if $y > y_{max}$, 0 otherwise
$b_1 = 1$ if $y < y_{min}$, 0 otherwise
$b_2 = 1$ if $x > x_{max}$, 0 otherwise
$b_3 = 1$ if $x < x_{min}$, 0 otherwise

Using Outcodes

Consider the 5 cases below



AB: outcode(A) = outcode(B) = 0

- Accept line segment

CD: outcode (C) = 0, outcode(D) $\neq$ 0

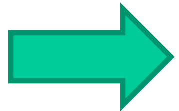
- Compute intersection
- Location of 1 in outcode(D) determines which edge to intersect with

Both outcodes are nonzero for other 3 cases, perform AND

- EF: outcode(E) AND outcode(F) (bitwise) $\neq$ 0
  – reject
- GH and IJ: outcode(G) AND outcode(H) =0
  – Shorten line segment by intersecting with one of sides of window and reexecute algorithm

Efficiency

Inefficient when code has to be reexecuted for line segments that must be shortened in more than one step

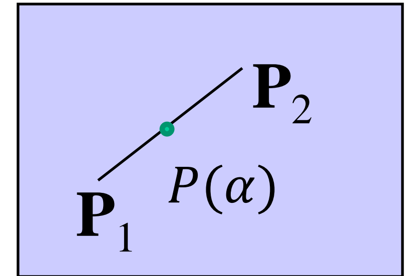


For the last case, use Liang-Barsky Clipping

Liang-Barsky Clipping

Consider the parametric form of a line segment

$$P(\alpha) = \begin{bmatrix} x(\alpha) \\ y(\alpha) \end{bmatrix} = (1-\alpha)P_1 + \alpha P_2 \quad 1 \geq \alpha \geq 0$$



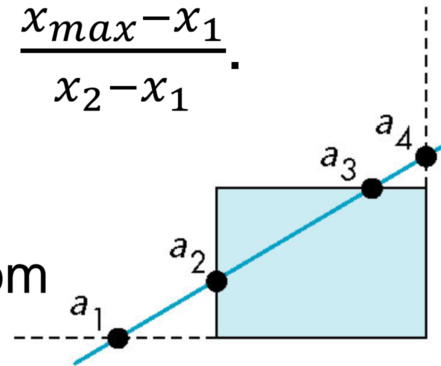
$$P(\alpha) = \begin{bmatrix} x(\alpha) \\ y(\alpha) \end{bmatrix} = \begin{bmatrix} (1-\alpha)x_1 + \alpha x_2 \\ (1-\alpha)y_1 + \alpha y_2 \end{bmatrix}$$

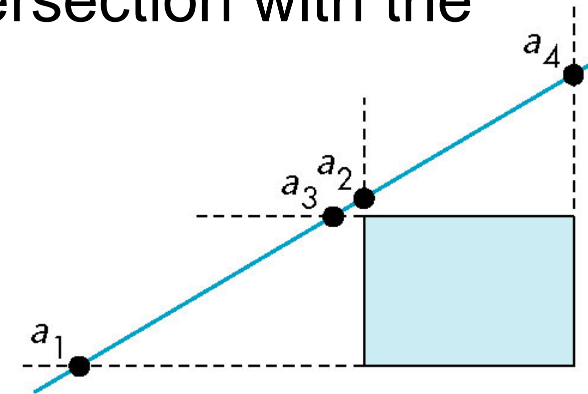
We can distinguish between the cases by looking at the ordering of the values of α where the line determined by the line segment crosses the lines that determine the window

Liang-Barsky Clipping

When the line is not parallel to a side of the window, compute intersections with the sides of window

For example, α_4 is the parameter for the intersection with the right side $x = x_{max} \Longrightarrow \alpha_4 = \frac{x_{max} - x_1}{x_2 - x_1}$.





E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

In (a): $\alpha_4 > \alpha_3 > \alpha_2 > \alpha_1$

- Intersect right, top, left, bottom
- shorten

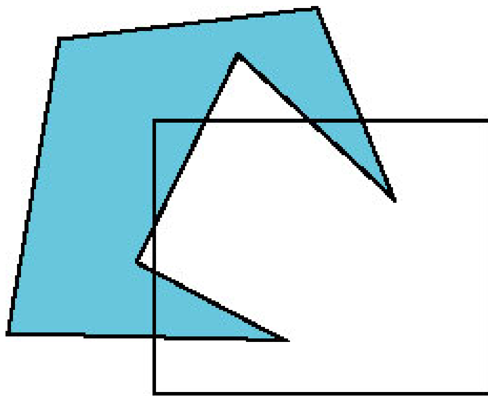
In (b): $\alpha_4 > \alpha_2 > \alpha_3 > \alpha_1$

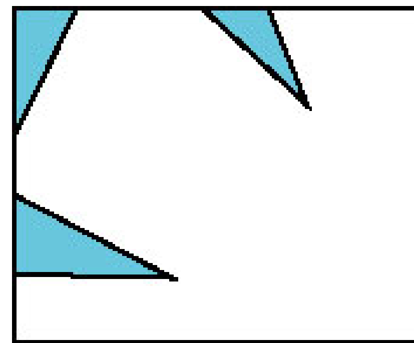
- Intersect both right and left before intersecting top and bottom
- reject

Polygon Clipping

Not as simple as line segment clipping

- Clipping a line segment yields at most one line segment
- Clipping a polygon can yield multiple polygons
  - Increase number of polygons



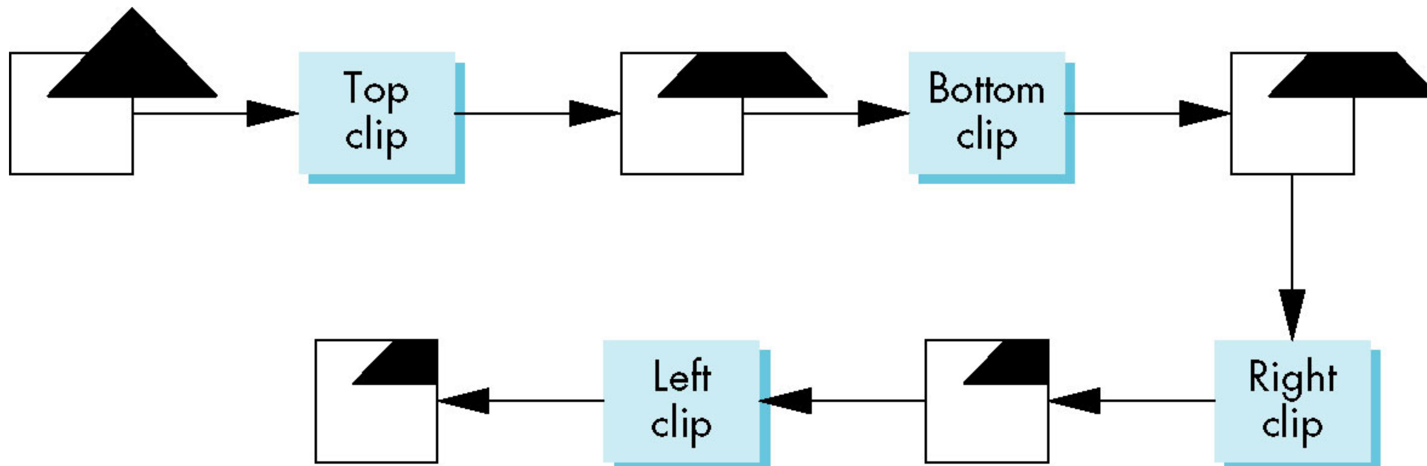


Clipping a convex polygon can yield at most one other polygon

E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

Pipeline Clipping of Polygons

For all edges of polygon, run the pipeline



Three dimensions: add front and back clippers

Not efficient for many-sided polygon

Rasterization (Scan Conversion)

Produces a set of fragments

Fragments have a location (pixel location) in the buffer and other attributes such color and texture coordinates that are determined by interpolating values at vertices

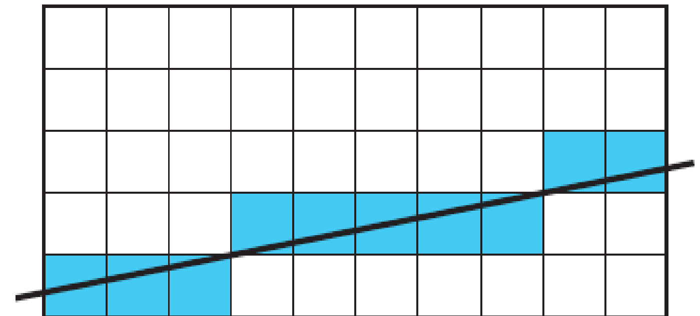
Scan Conversion of Line Segments -- DDA Algorithm

Digital Differential Analyzer

- DDA was a mechanical device for numerical solution of differential equations
- Line $y = mx + h$ satisfies differential equation

$$\frac{dy}{dx} = m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$ → two endpoints

Along scan line Δx = 1



```
For(x=x1; x<=x2,ix++) {
    y+=m;
  write_pixel(x, round(y), line_color)
}
```

E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

Bresenham's Algorithm

m is a floating point

Bresenham's algorithm eliminates all fp calculations

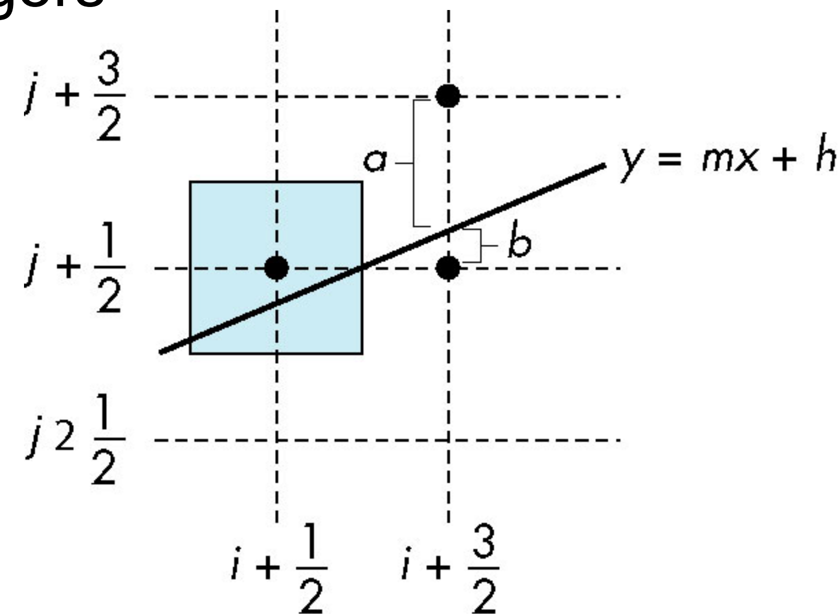
Consider only $0 \le m \le 1$, other cases by symmetry

Assume pixel centers are at half integers

Decision variable:

$$d = (x_2 - x_1)(a - b)$$
$$= \Delta x(a - b)$$

d < 0 use upper pixel
d > 0 use lower pixel



Polygon Rasterization

How to tell inside from outside – inside-outside testing

- Convex easy
- Nonsimple difficult
- Odd even test: count edge crossings with scanlines
  - Inside: odd crossings
  - Outside: even crossings
- Winding test: number of times of a point encircled by the edges
  - Inside if winding number $\neq 0$

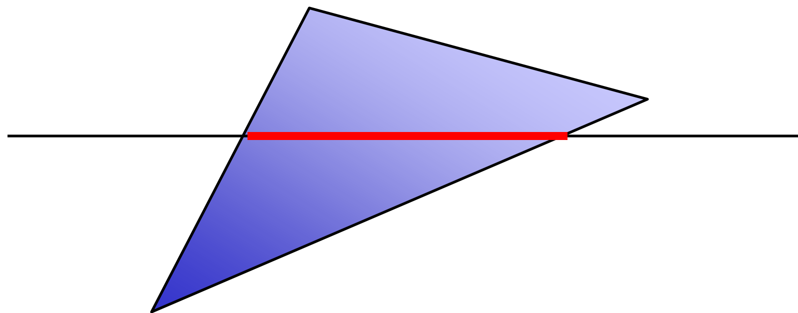
Filling in the Frame Buffer

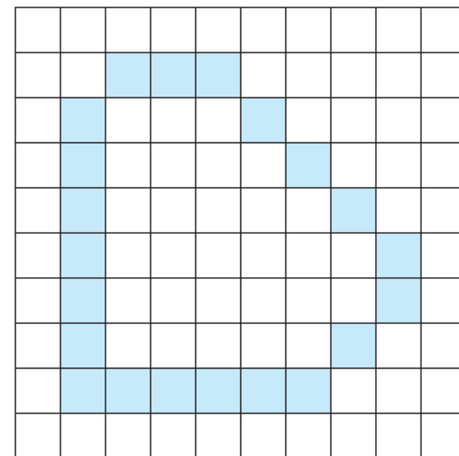
Fill at end of pipeline: coloring a point with the inside color if it is inside the polygon

- Convex Polygons only
- Nonconvex polygons assumed to have been tessellated
- Shades (colors) have been computed for vertices (Gouraud shading)

Two approaches

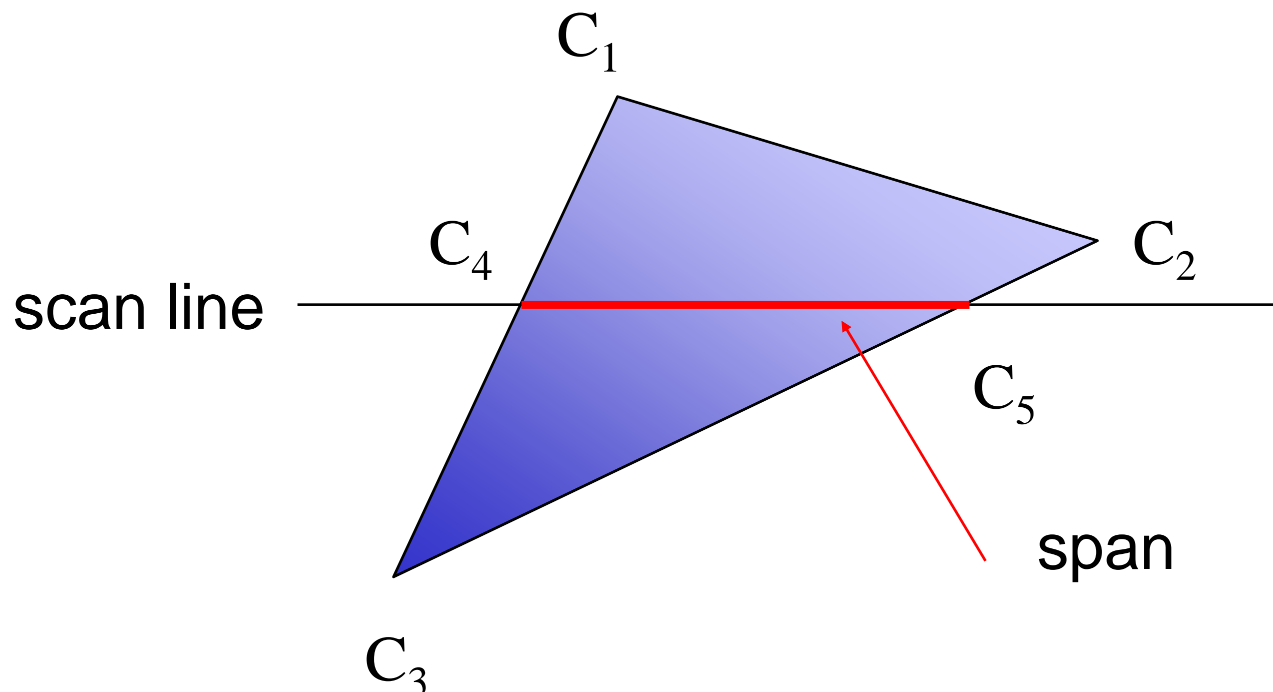
- Scanline fill
- Flood fill





Scanline Fill: Using Interpolation

C_1 C_2 C_3 specified by `glColor` or by vertex shading
C_4 determined by interpolating between C_1 and C_2
C_5 determined by interpolating between C_2 and C_3
Interpolate points between C_4 and C_5 along span

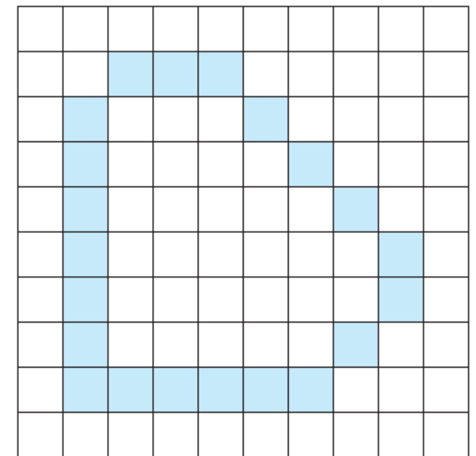


Flood Fill

Starting with an unfilled polygon, whose edges are rasterized into the buffer, fill the polygon with inside color (BLACK)

Fill can be done recursively if we know a seed point located inside. Color the neighbors to (BLACK) if they are not edges.

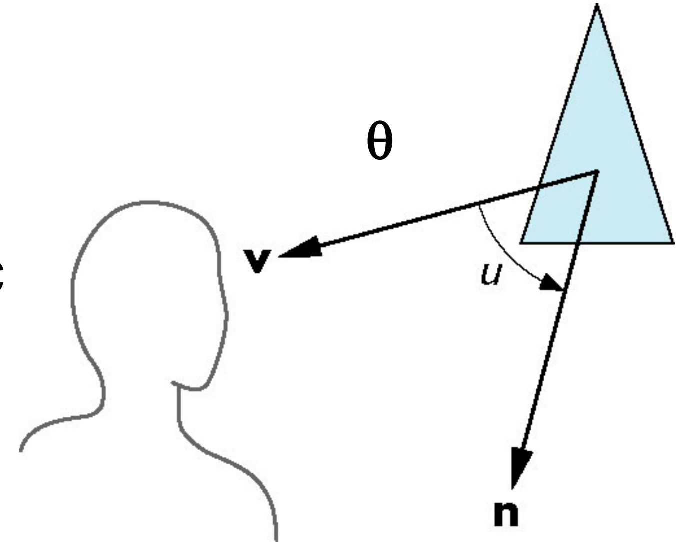
```
flood_fill(int x, int y) {
    if(read_pixel(x,y)= = WHITE) {
        write_pixel(x,y,BLACK);
        flood_fill(x-1, y);
        flood_fill(x+1, y);
        flood_fill(x, y+1);
        flood_fill(x, y-1);
} }
```



Back-Face Removal (Culling)

Only render front-facing polygons

- After transformation (projection normalization), the view is orthographic
$$\mathbf{v} = (\,0\;0\;1\;0)^T$$
- The coordinates are normalized device coordinates
- If the plane of face has form
$$ax + by + cz + d = 0$$



E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

Need only test the sign of c

Hidden Surface Removal

Object-space algorithms:

- Consider the relationships between objects
  - Pairwise testing
  - Painter's algorithm with depth sorting
- Reduce number of polygons
- Works better for a smaller number of objects

Image-space algorithms:

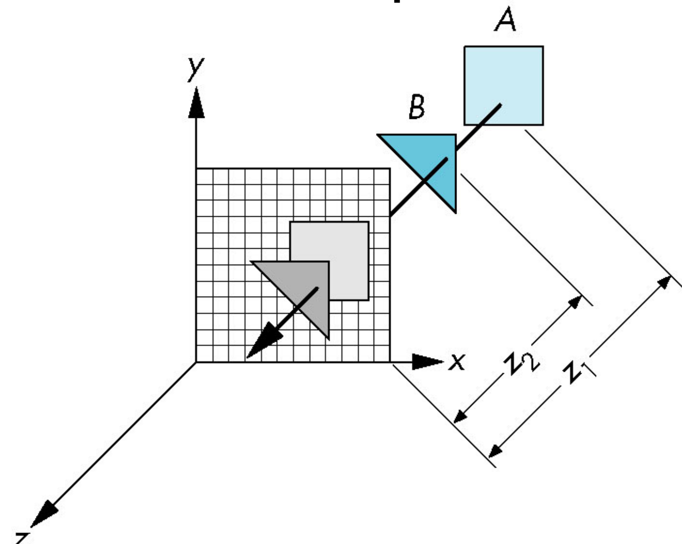
- Works at fragment/pixel level
- Z-buffer algorithm

z-Buffer Algorithm

Use a buffer called the z or depth buffer to store the depth of the closest object at each pixel found so far

As we render each polygon, compare the depth of each pixel to depth in z buffer

If less, place shade of pixel in color buffer and update z buffer



Buffer

Define a buffer by its spatial resolution ($n \times m$) and its depth (or precision) k, the number of bits/pixel

Writing in buffers

- Bit block transfer (bitblt) operations

Writing model

- Read destination pixel before writing source

Writing Model

Source and destination bits are combined bitwise

s: source bit

d: destination bit

16 possible functions (one per column in table)

- 0 and 15: clear mode
- 3 : replace mode
- 7: OR mode
- 6: XOR mode

How to choose the mode?

s	d	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

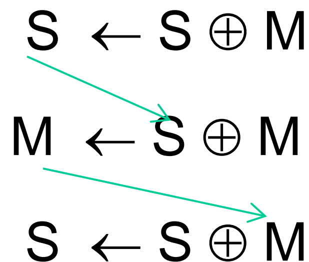
XOR (Exclusive OR) Mode

Property of XOR: return the original value if apply XOR twice

$$d = (d \oplus s) \oplus s$$

XOR is especially useful for swapping blocks of memory such as menus that are stored off screen (***backing store***)

If S represents screen and M represents a menu, the sequence



For example, S=1010, M=1100

$S = S \oplus M = 0110$

$M = S \oplus M = 1010$

$S = S \oplus M = 1100$

swaps S and M

E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

Mapping

Modify color in fragment processing after rasterization

Three Major Mapping Methods

- **Texture Mapping**
  - Uses images to fill inside of polygons
- **Environment (reflection mapping)**
  - Uses a picture of the environment for texture maps of reflection surface
  - Allows simulation of highly specular surfaces
- **Bump mapping**
  - Emulates altering normal vectors during the rendering process

Texture Mapping

Map an image to a surface or map every texel to a point on a geometric object – **Backward mapping in practice**

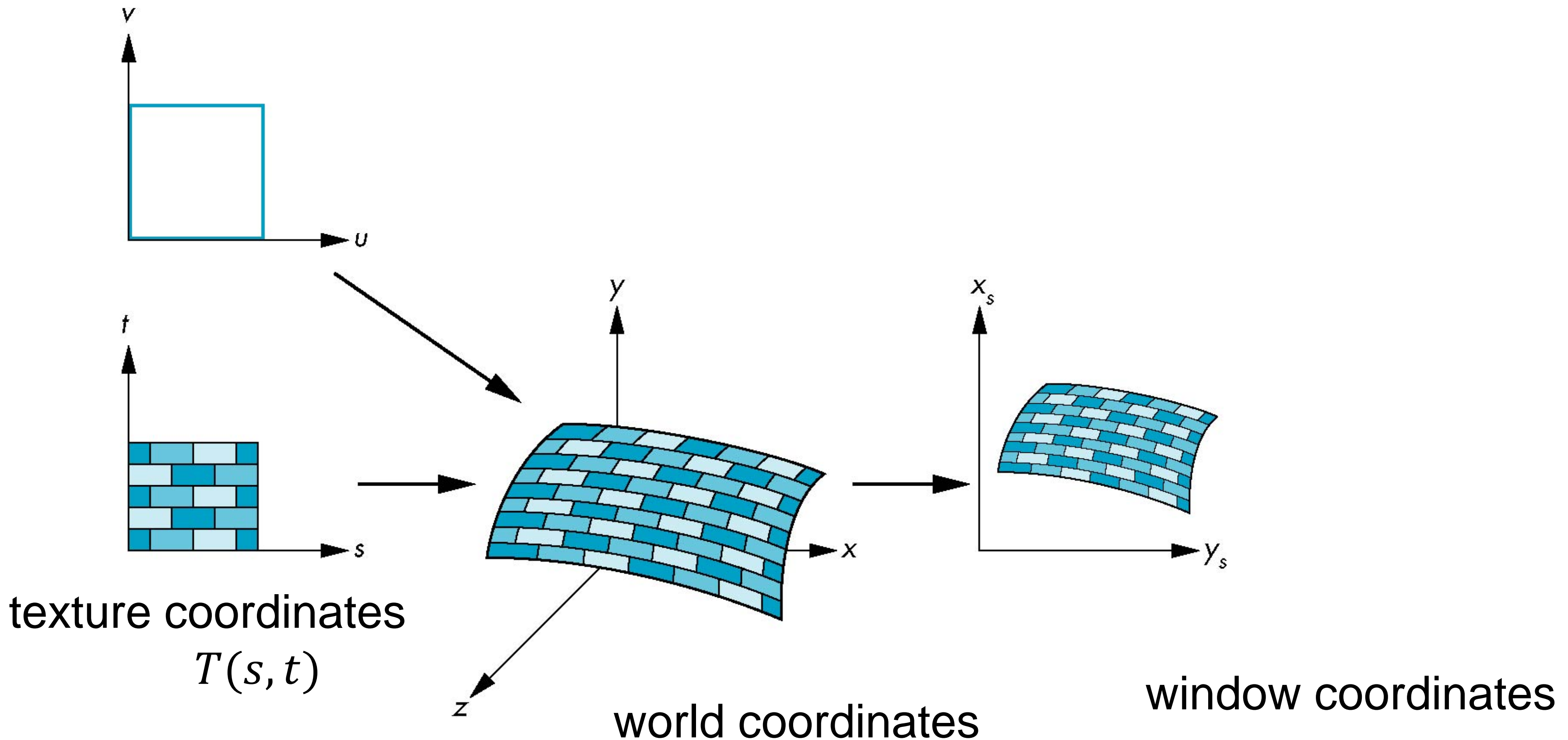
Textures are stored in images - 2D arrays.

Each element is called a ***texel***

Coordinate systems

- Parametric coordinates
  - May be used to model curves and curved surfaces
- Texture coordinates
  - Used to identify points in the image to be mapped
- Object or World Coordinates
  - Conceptually, where the mapping takes place
- Window/screen Coordinates
  - Where the final image is really produced

Texture Mapping



Two-part mapping

First mapping: map the texture to a simple intermediate surface, e.g.,

- Cylindrical mapping
- Spherical mapping
- Box mapping

Second mapping: map from intermediate object to the actual object

OpenGL Texture Mapping

Texture mapping is part of fragment processing

Three steps to applying a texture

1. Generate the texture map
   - read or generate image
   - assign to texture
   - enable texturing
2. assign texture coordinates to vertices
   - Texture coordinates can be interpolated
   - Proper mapping function is left to application
3. specify texture parameters
   - wrapping, filtering

Associate Texture Coordinates with Vertices of Object

Treat texture coordinates as a vertex attribute, similar to vertices and vertex colors.

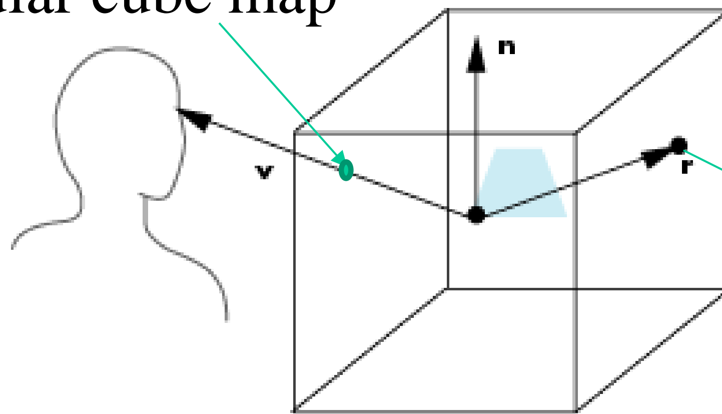
Pass the vertex texture coordinates to the vertex shader.

The rasterizer interpolates the vertex texture coordinates to fragment texture coordinates.

Environment/Reflection Map

Instead of using the view vector to determine the texture, ***environment map*** uses reflection vector to locate texture in ***cube map***

corresponding texel
for regular cube map



corresponding texel
for reflection map

E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

Bump Mapping

Perturb normal for each fragment before applying lighting

- Add noise to the normal or
- Store perturbation as textures and lookup a perturbation value in a texture map

Bump mapping must be performed in shaders

Approximating the Perturbed Normal

$$\mathbf{n'} = \mathbf{p'}_u \times \mathbf{p'}_v$$

$$\approx \mathbf{n} + (\partial d/\partial u)\mathbf{n} \times \mathbf{p}_v + (\partial d/\partial v)\mathbf{n} \times \mathbf{p}_u$$

The vectors $\mathbf{n} \times \mathbf{p}_v$ and $\mathbf{n} \times \mathbf{p}_u$ lie in the tangent plane

→ The normal is displaced in the tangent plane

→ $\mathbf{n'}$, $\mathbf{p'}_u$ and $\mathbf{p'}_v$ form a local coordinate space – ***Tangent Space***

Tangent Space and Normal Matrix

However, n’, p'_u and p'_v may be not unit vectors and not orthogonal to each other

Need to get an orthogonal basis

- Normalized normal: $\mathbf{m} = \frac{\mathbf{n}'}{|\mathbf{n}'|}$
- Tangent vector: $\mathbf{t} = \frac{\mathbf{p}'_u}{|\mathbf{p}'_u|}$
- Binormal vector: $\mathbf{b} = \mathbf{m} \times \mathbf{t}$

A transformation matrix is used to transform the view and light to tangent space

$$\mathbf{M} = [\mathbf{t} \quad \mathbf{b} \quad \mathbf{m}]^{\mathbf{t}}$$

Blending Equation

We can define source and destination blending factors for each RGBA component

$$s = [s_r, s_g, s_b, s_\alpha]$$

$$d = [d_r, d_g, d_b, d_\alpha]$$

Suppose that the source and destination colors are

$$b = [b_r, b_g, b_b, b_\alpha]$$

$$c = [c_r, c_g, c_b, c_\alpha]$$

Blend as $c' = [b_r s_r + c_r d_r, b_g s_g + c_g d_g , b_b s_b + c_b d_b , b_\alpha s_\alpha + c_\alpha d_\alpha]$

Example

Suppose that we start with the opaque background color $(R_0,G_0,B_0,1)$

- This color becomes the initial destination color

We now want to blend in a translucent polygon with color (R_1,G_1,B_1,α_1)

Select `GL_SRC_ALPHA` and `GL_ONE_MINUS_SRC_ALPHA` as the source and destination blending factors

$$R'_1 = \alpha_1 R_1 + (1- \alpha_1) R_0, \ldots$$

→ The composition method discussed earlier

Note this formula is correct if polygon is either opaque or transparent

Instance Transformation

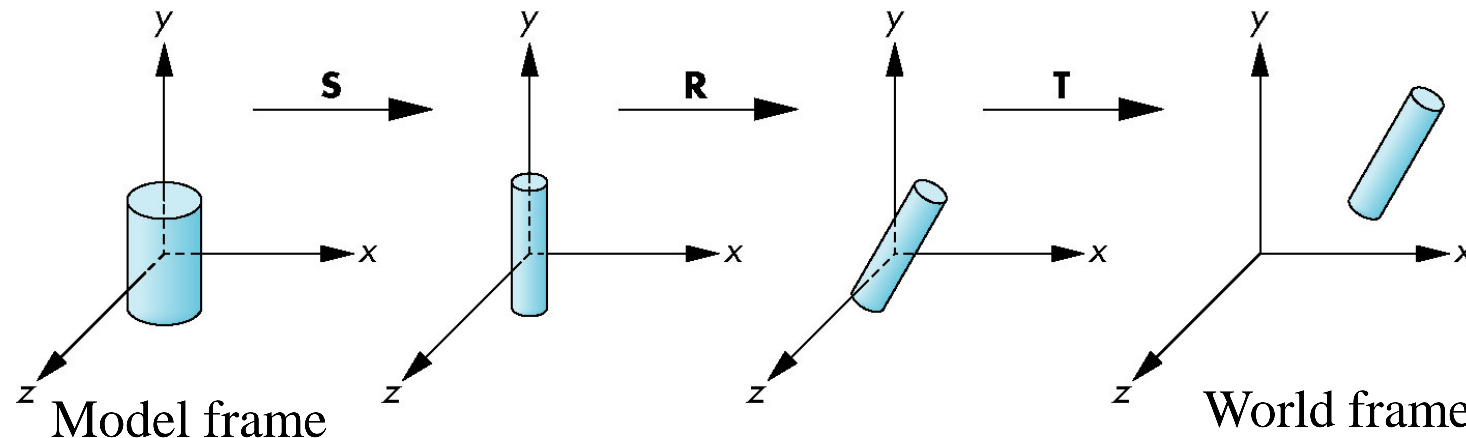
Start with a prototype object (a *symbol*), e.g.,

- Geometric objects
- Fonts

Each appearance of the object in the model is an *instance*

- A instance transformation from model frame to world frame by scaling, rotation, and translation

$$\mathbf{M} = \mathbf{TRS}$$



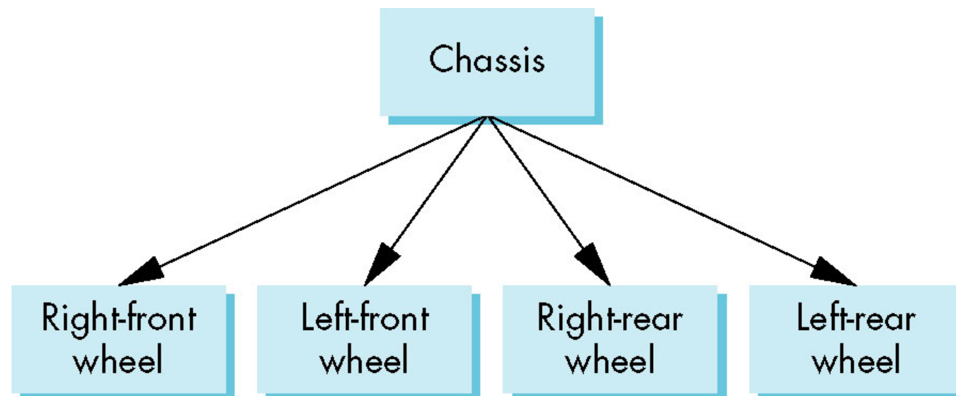
Relationships in Car Model

Symbol-instance table does not show relationships between parts of model

Consider model of car

- Chassis + 4 identical wheels
- Two symbols

Rate of forward motion determined by rotational speed of wheels



Modeling with Trees

Must decide what information to place in nodes and what to put in edges

Nodes

- What to draw
- Pointers to children
- information on incremental changes to transformation matrices (can also store in edges)

Edges

- May have information on incremental changes to transformation matrices (can also store in nodes)

Left-child right sibling binary tree

Tree travelsal

- Stack-based travelsal
- Preorder travelsal

Animation

Procedural Approaches

- **Physically-based models and particle system**
  - Describing dynamic behaviors
    - Fireworks
    - Flocking behavior of birds
    - Wave action
- **Language-based models**
  - Describing trees or terrain
  - Representing relationships
- **Fractal geometry**

Design approaches based on procedural methods

Newtonian Particle

Particle system is a set of particles

Each particle is an ideal point mass

- Gives the positions of particles
- At each location, we can show an object

Six degrees of freedom

- Position
- Velocity

Each particle obeys Newtons' law

$$\mathbf{f} = m\mathbf{a}$$

Force Vector

Depending on how particles interact with each other

- Independent Particles O(n)
  - Gravity
  - Drag
- Coupled Particles O(n)
  - Spring-Mass Systems
  - Meshes
- Coupled Particles O(n^2)
  - Attractive and repulsive forces