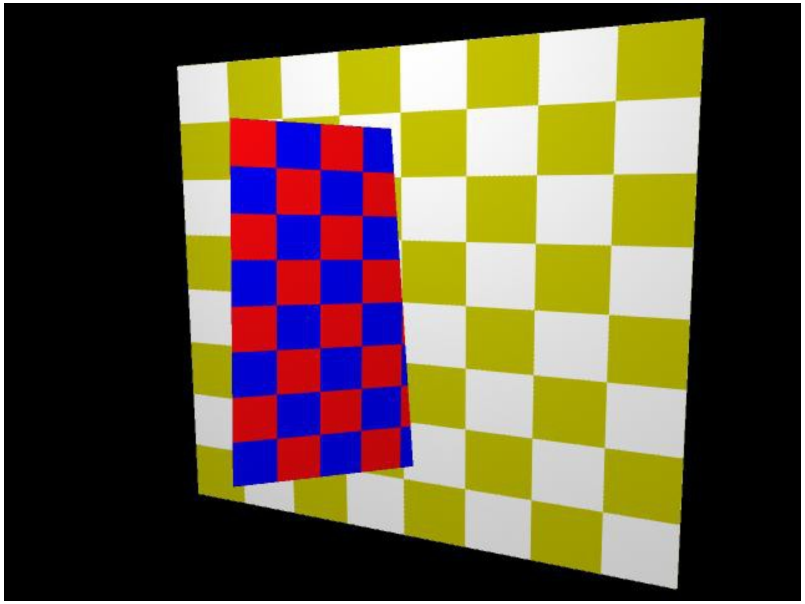
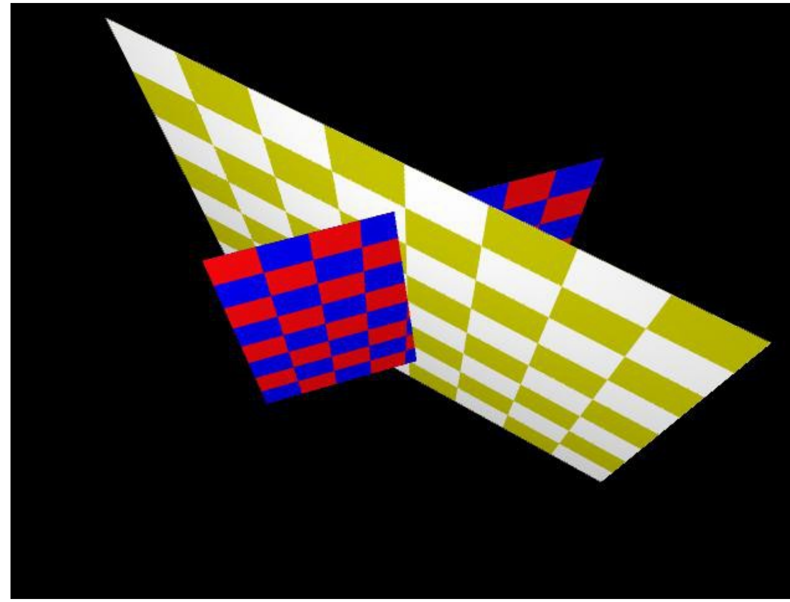
Pipeline Operations

CS 4620 Lecture 10

Hidden surface elimination

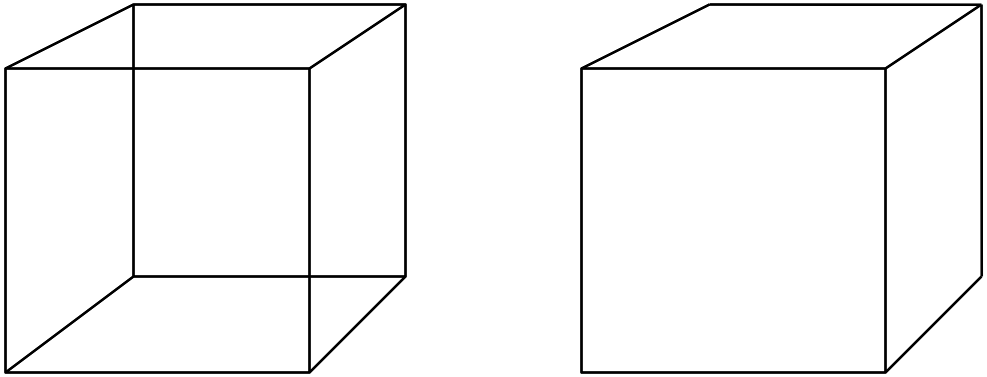




Goal is to figure out which color to make the pixels based on what's in front of what.

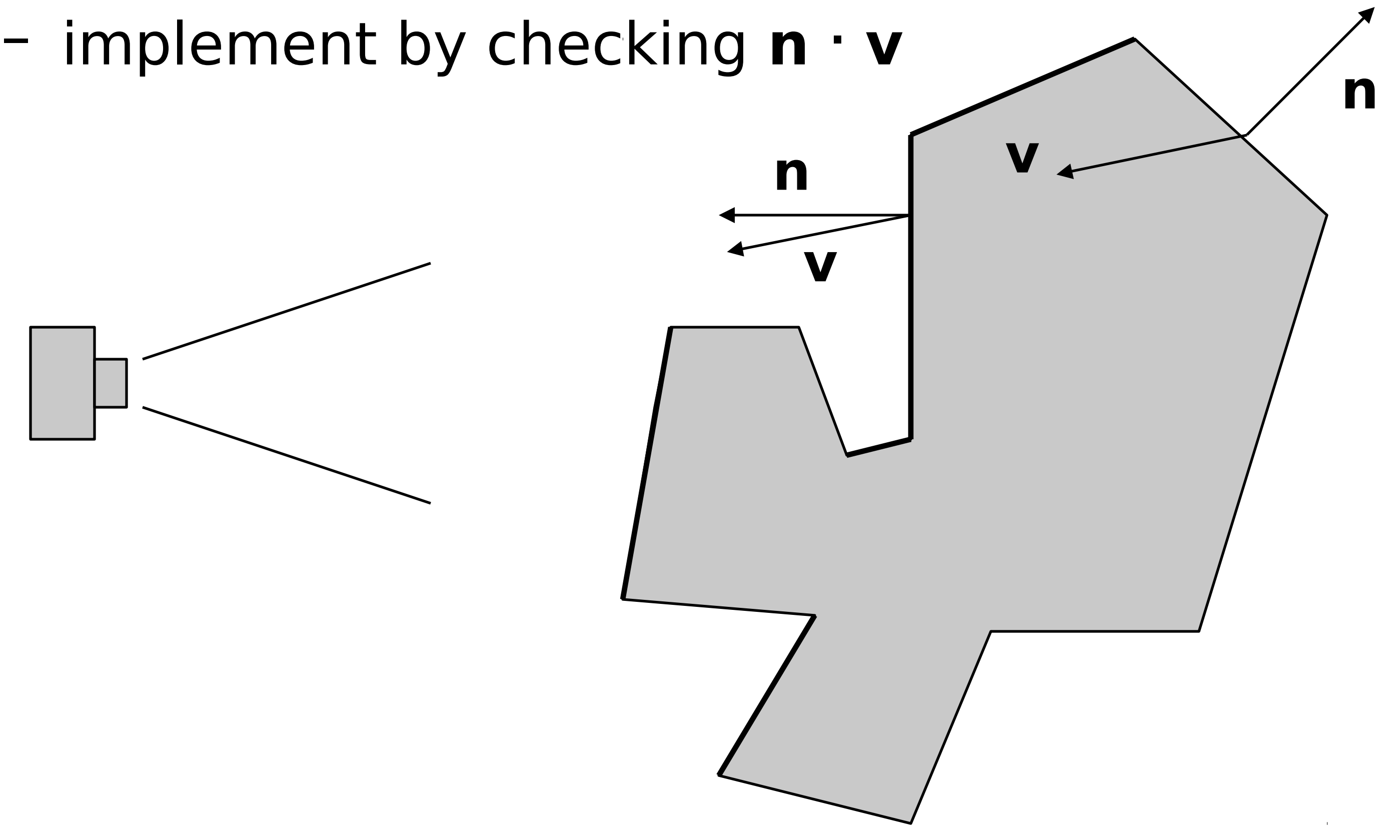
Hidden surface elimination

- We have discussed how to map primitives to image space
  - projection and perspective are depth cues
  - occlusion is another very important cue

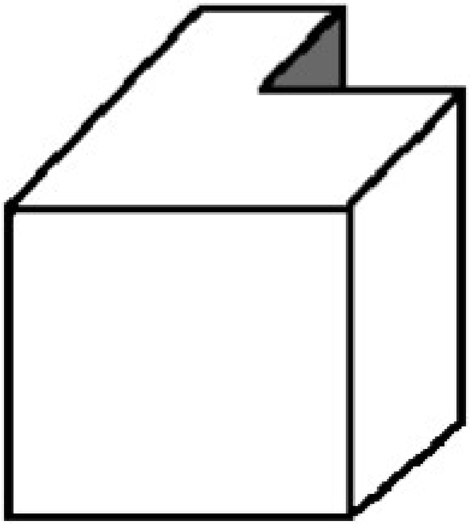


Back face culling

- For closed shapes you will never see the inside
  - therefore only draw surfaces that face the camera
  - implement by checking **n · v**



Back-face Culling



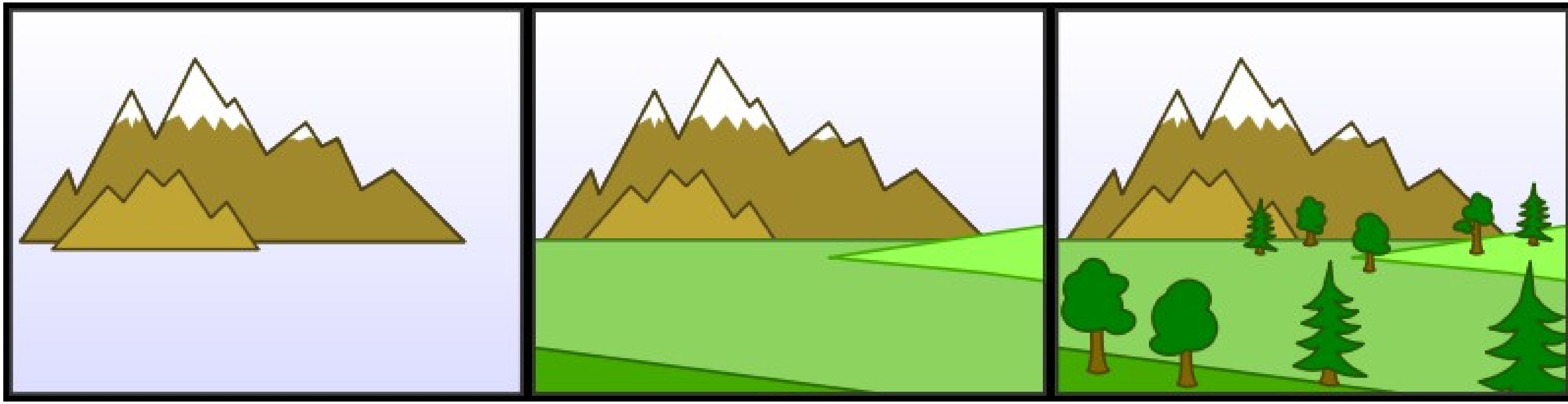
- Problems ?　　언제 문제가 생기나?

- Conservative algorithms

- Real job of visibility never solved

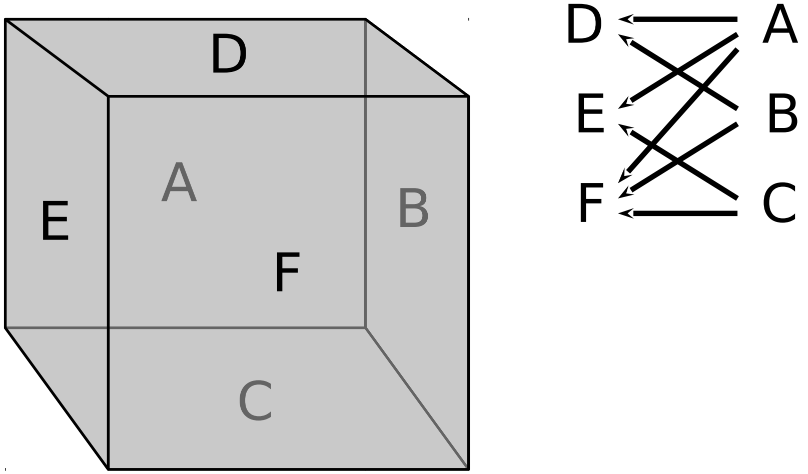
Painter's algorithm

- Simplest way to do hidden surfaces
- Draw from back to front, use overwriting in framebuffer



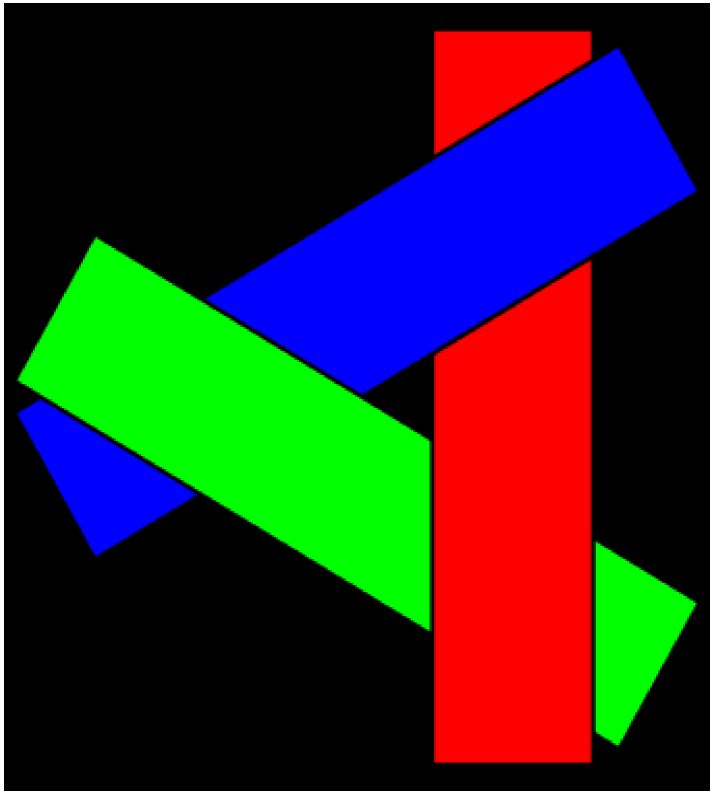
Painter's algorithm

- Amounts to a topological sort of the graph of occlusions
  - that is, an edge from A to B means A sometimes occludes B
  - any sort is valid
    - ABCDEF
    - BADCFE
  - if there are cycles there is no sort



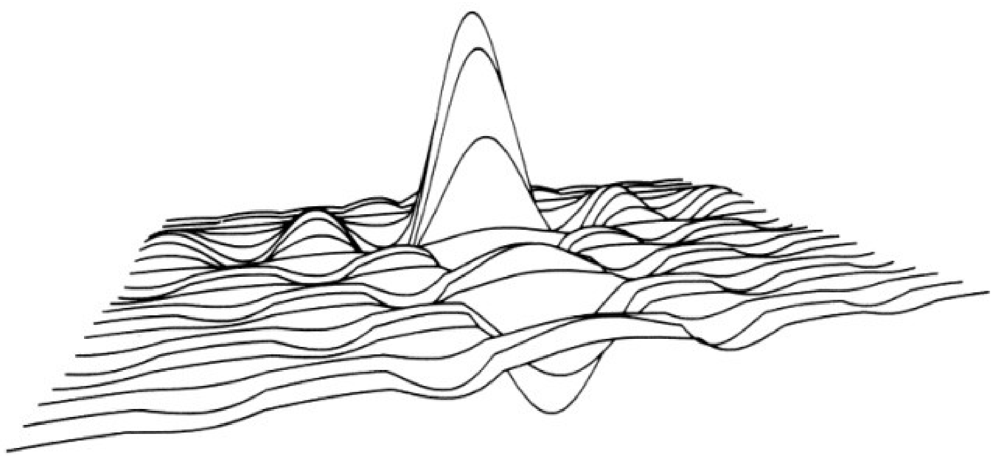
Painter's algorithm

- examples of cycles
- No sort unless polygons are divided into small pieces



Painter's algorithm

- Useful when a valid order is easy to come by
- Compatible with alpha blending (later)



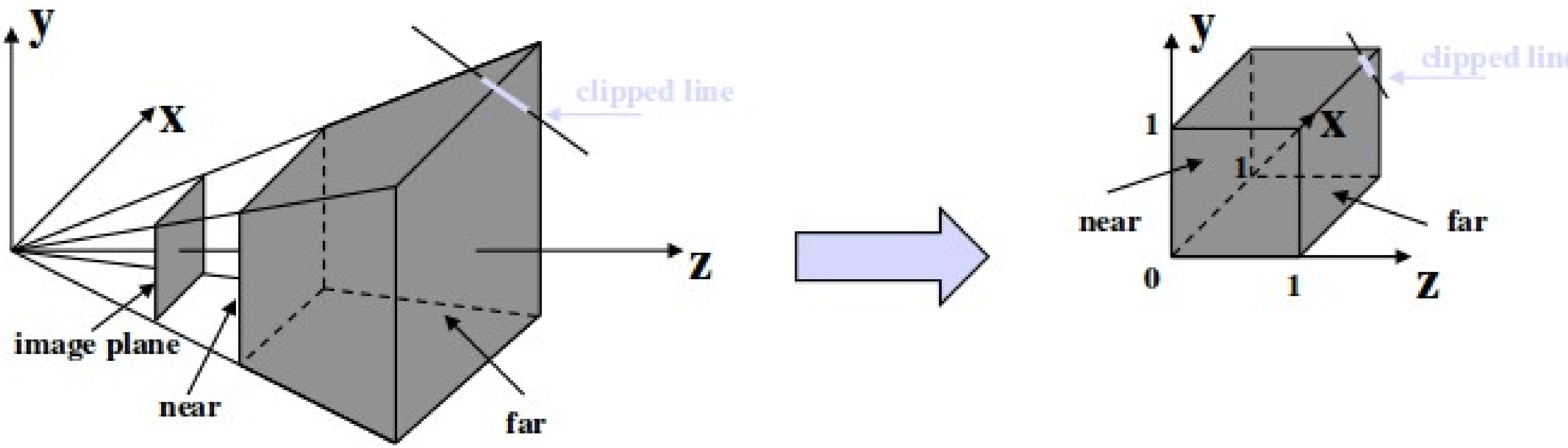
[Foley et al.]

The *z* buffer

- In many (most) applications maintaining a *z* sort is too expensive
  - changes all the time when the view changes
  - many data structures exist, but complex
- Solution: draw in any order, keep track of closest
  - Z-buffer keeps track of closest depth so far
  - when drawing, compare object's depth to current closest depth and discard if greater

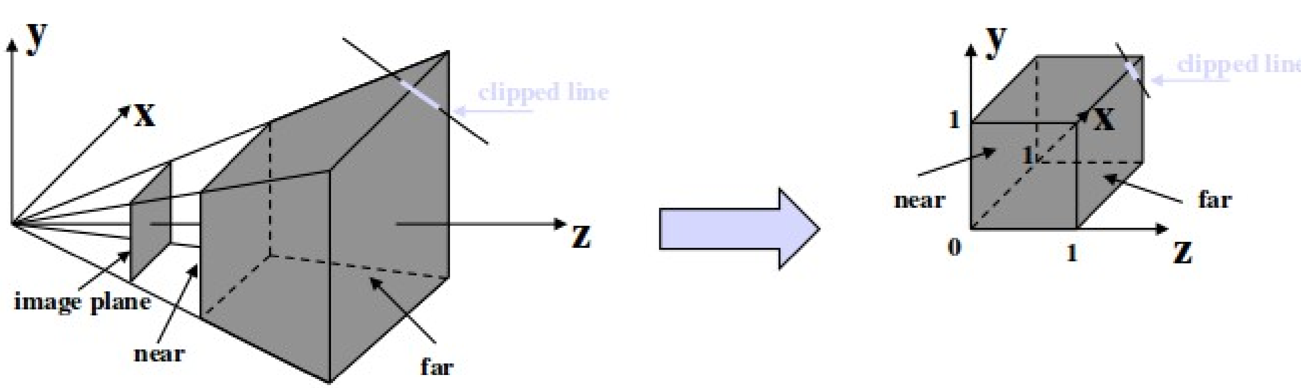
Where Are We ?

- Canonical view volume (3D image space)
- Clipping done
- division by w
- z > 0



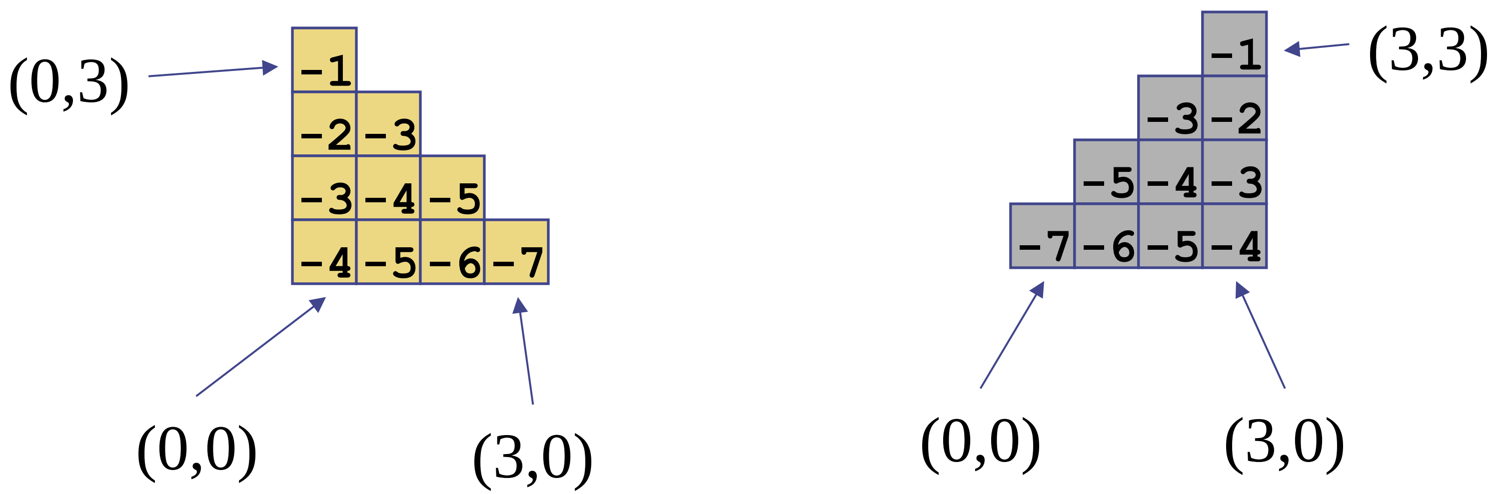
Z-Buffering: Algorithm

```
allocate z-buffer;                  // Allocate depth buffer → Same size as viewport.

for each pixel (x,y)                // For each pixel in viewport.
  writePixel(x,y,backgrnd);               // Initialize color.
  writeDepth(x,y,farPlane);               // Initialize depth (z) buffer.

for each polygon                    // Draw each polygon (in any order).
  for each pixel (x,y) in polygon         // Rasterize polygon.
    p_z = polygon's z-value at (x,y);     // Interpolate z-value at (x, y).
    if (p_z > z-buffer(x,y))              // If new depth is closer:
      writePixel(x,y,color);              //   Write new (polygon) color.
      writeDepth(x,y,p_z);                //   Write new depth.
```



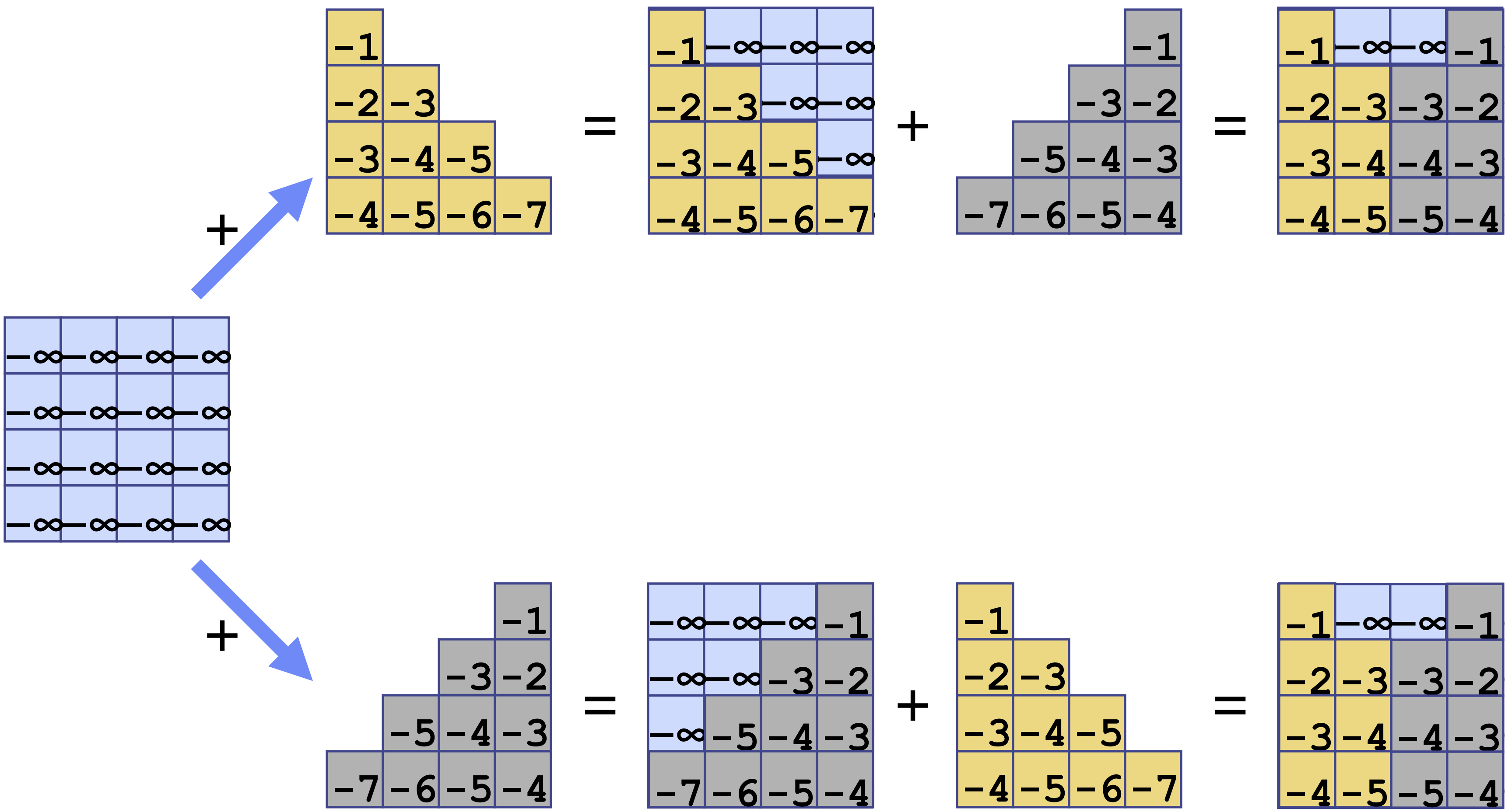
Z-Buffering: Example

Scan convert the following two polygons.
The number in the pixel represents the z- value for that pixel

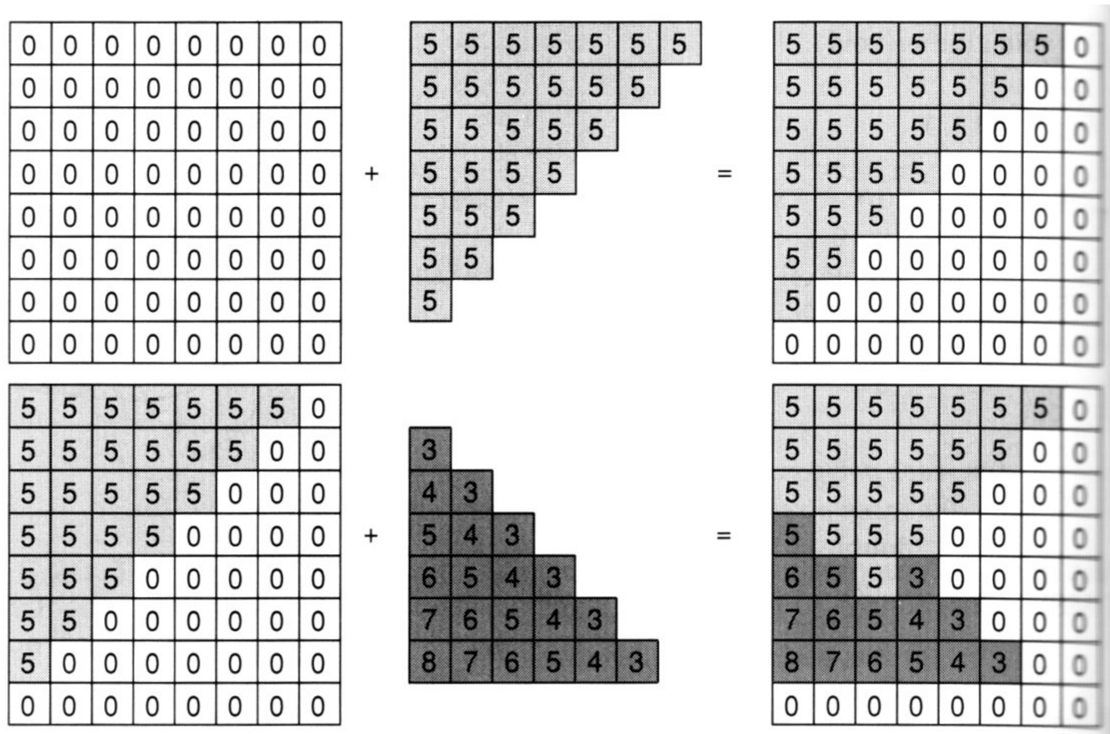


Does order matter?

Z-Buffering: Example



The *z* buffer



[Foley et al.]

– another example of a memory-intensive brute force approach that works and has become the standard

Z-Buffering : Summary

- Advantages:
  - Easy to implement
  - Fast with hardware support → Fast depth buffer memory
  - On most hardware
  - No sorting of objects
  - Shadows are easy (later)
- Disadvantages:
  - Extra memory required for z-buffer:
    - Integer depth values
    - Scan-line algorithm
  - Prone to aliasing
    - Super-sampling

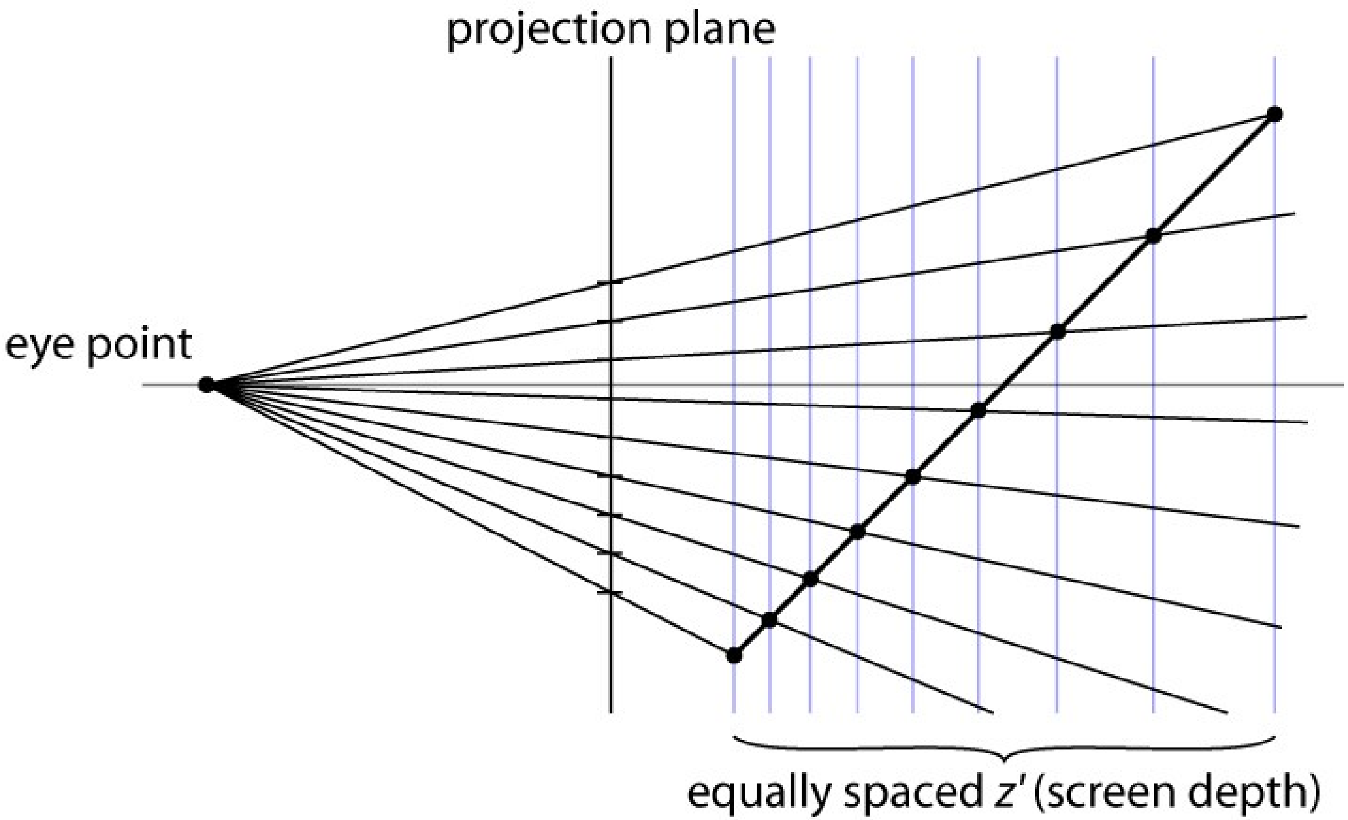
Pipeline for basic *z* buffer

- Vertex stage (input: position / vtx; color / tri)
  - transform position (object to screen space)
  - pass through color
- Rasterizer
  - interpolated parameter: z' (screen z)
  - pass through color
- Fragment stage (output: color, z')
  - write to color planes only if interpolated z' < current z'

Precision in *z* buffer

- The precision is distributed between the near and far clipping planes
  - this is why these planes have to exist
  - also why you can't always just set them to very small and very large distances
- Generally use z' (not world z) in z buffer

Interpolating in projection



linear interp. in screen space ≠ linear interp. in world (eye) space

Solution: linearly interpolate 1/z'

Read, if you are interested, http://www.lysator.liu.se/~mikaelk/doc/perspectivetexture/

Flat shading

- Shade using the real normal of the triangle
  - same result as ray tracing a bunch of triangles
- Leads to constant shading and faceted appearance
  - truest view of the mesh geometry

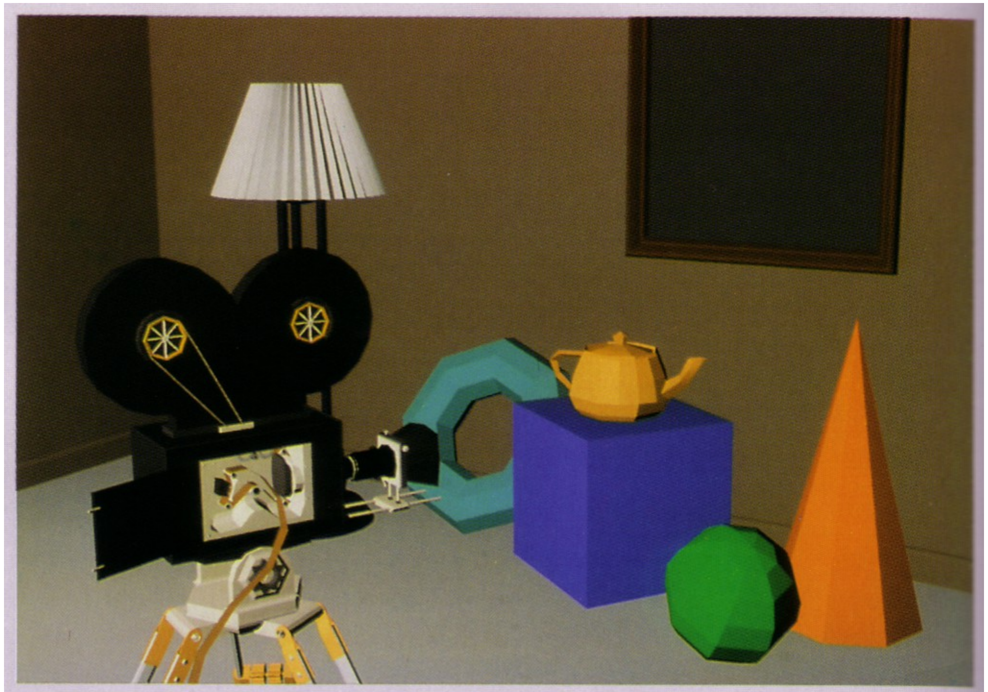


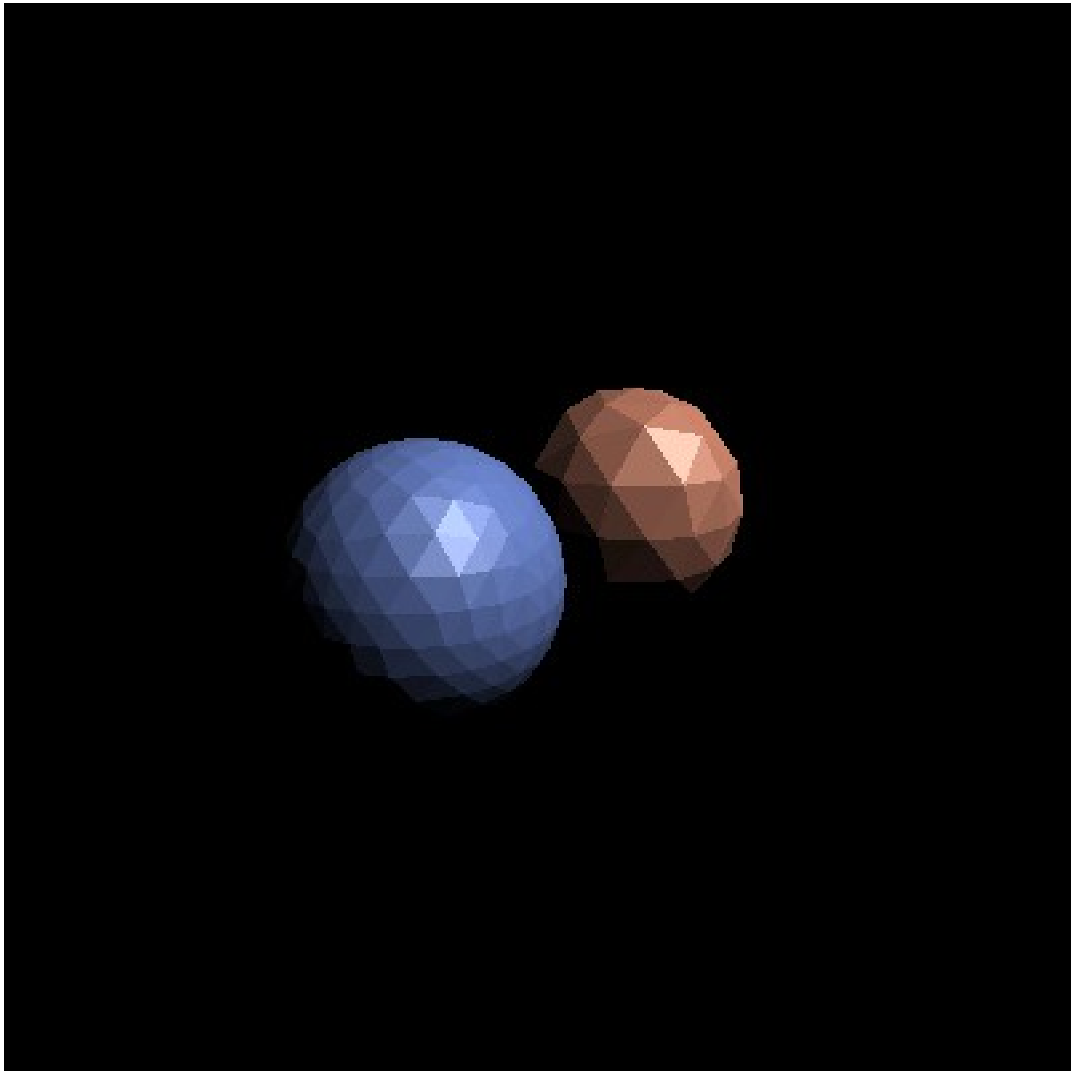
Plate II.29 *Shutterbug.* Individually shaded polygons with diffuse reflection (Sections 14.4.2 and 16.2.3). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

[Foley et al.]

Pipeline for flat shading

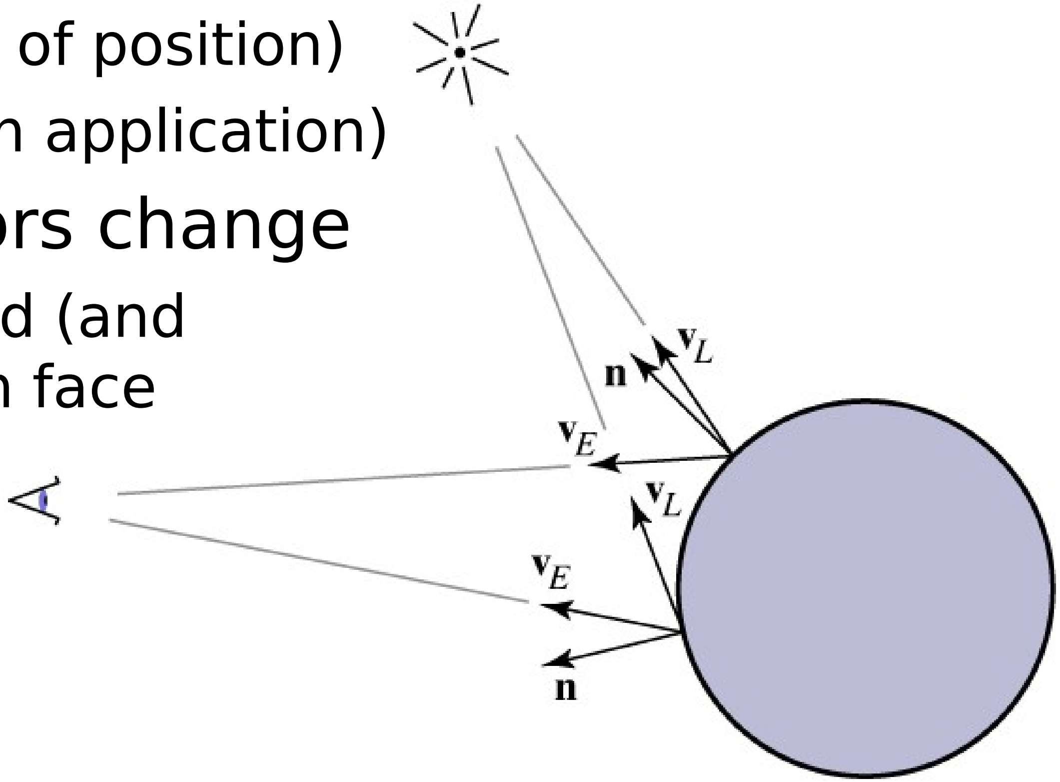
- Vertex stage (input: position / vtx; color and normal / tri)
  - transform position and normal (object to eye space)
  - compute shaded color per triangle using normal
  - transform position (eye to screen space)
- Rasterizer
  - interpolated parameters: z' (screen z)
  - pass through color
- Fragment stage (output: color, z')
  - write to color planes only if interpolated $z' <$ current z'

Result of flat-shading pipeline



Local vs. infinite viewer, light

- Phong illumination requires geometric information:
  - light vector (function of position)
  - eye vector (function of position)
  - surface normal (from application)
- Light and eye vectors change
  - need to be computed (and normalized) for each face

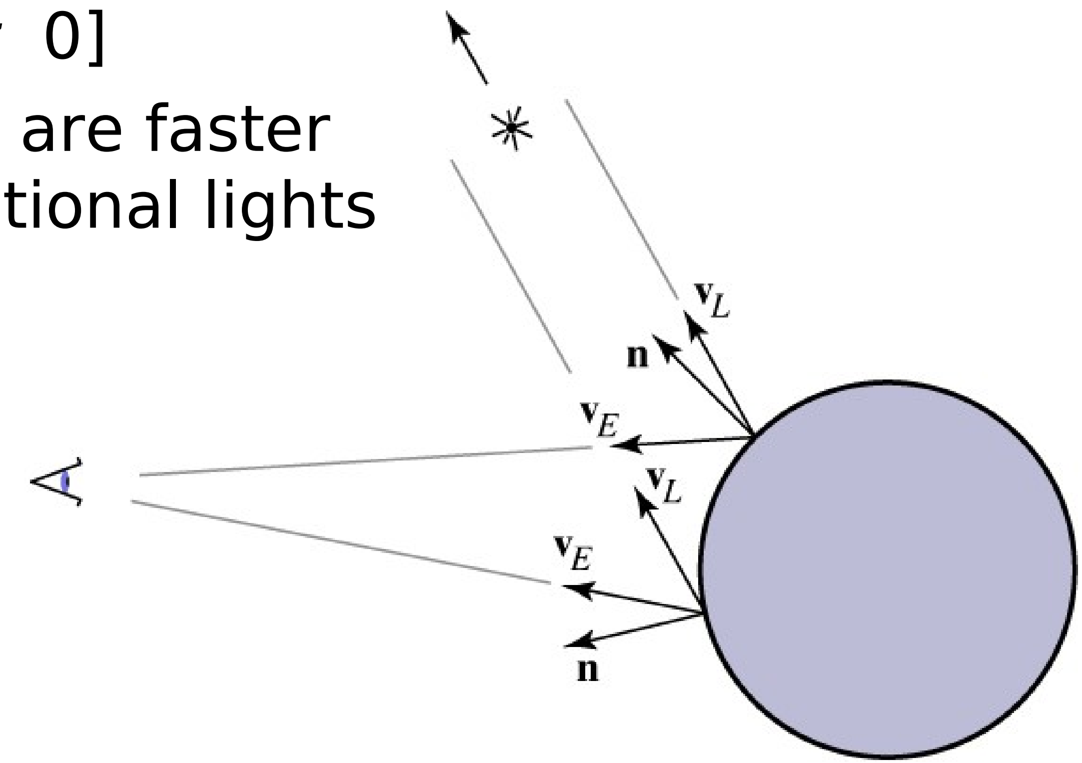


Local vs. infinite viewer, light

- Look at case when eye or light is far away:
  - distant light source: nearly parallel illumination
  - distant eye point: nearly orthographic projection
  - in both cases, eye or light vector changes very little
- Optimization: approximate eye and/or light as infinitely far away

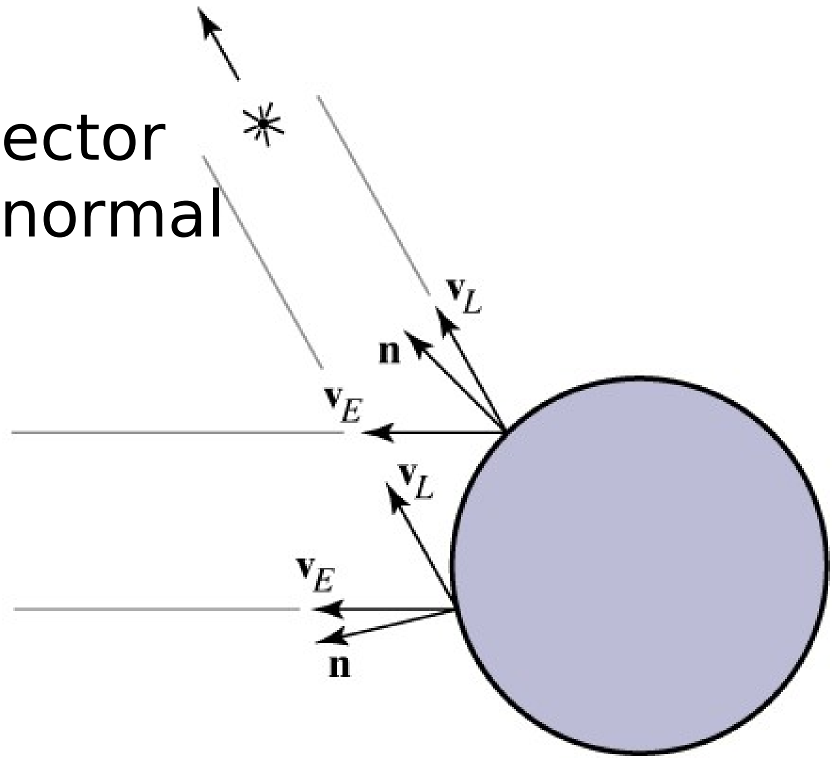
Directional light

- Directional (infinitely distant) light source
  - light vector always points in the same direction
  - often specified by position [x y z 0]
  - many pipelines are faster if you use directional lights

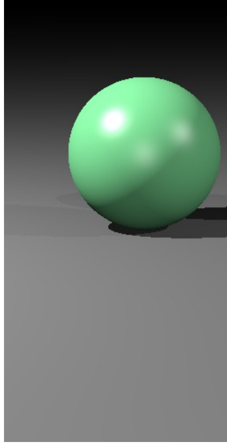


Infinite viewer

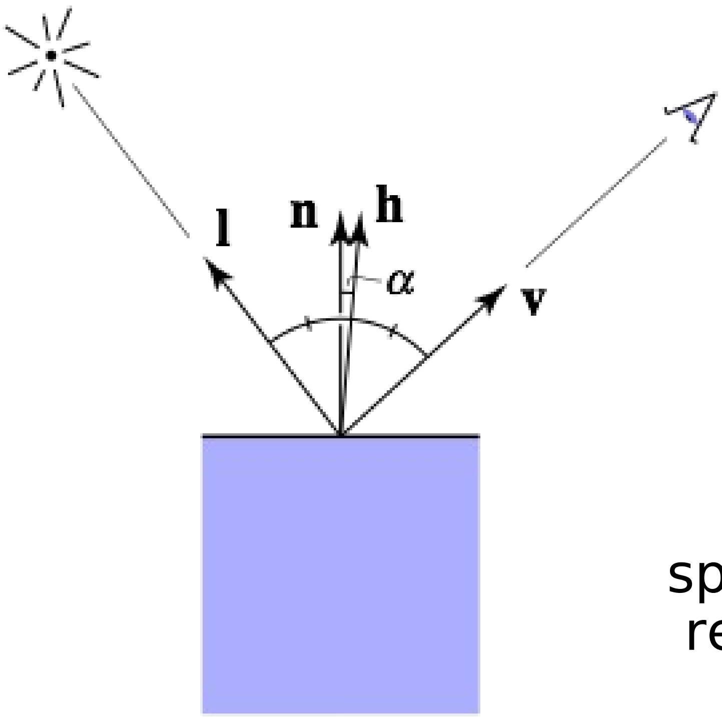
- Orthographic camera
  - projection direction is constant
- "Infinite viewer"
  - even with perspective, can approximate eye vector using the image plane normal
  - can produce weirdness for wide-angle views
  - Blinn-Phong: light, eye, half vectors all constant!



Specular shading (Blinn-Phong)



- Close to mirror ⇔ half vector near normal
  - Measure “near” by dot product of unit vectors



$$\mathbf{h} = \mathrm{bisector}(\mathbf{v}, \mathbf{l}) = \frac{\mathbf{v} + \mathbf{l}}{\|\mathbf{v} + \mathbf{l}\|}$$

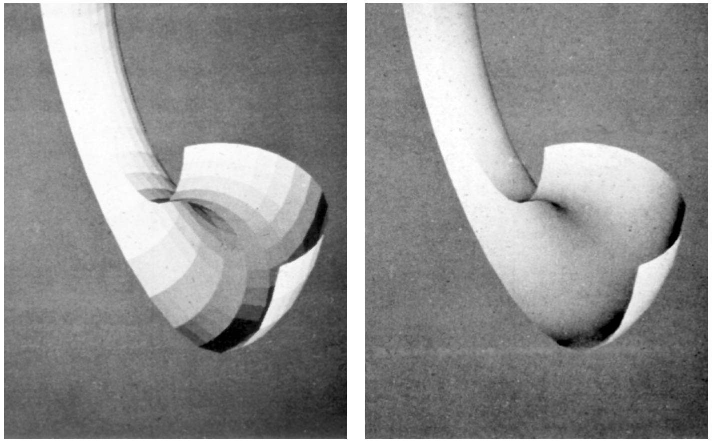
$$L_s = k_s \, I \max(0, \cos\alpha)^p = k_s \, I \max(0, \mathbf{n} \cdot \mathbf{h})^p$$

specularly reflected light

specular coefficient

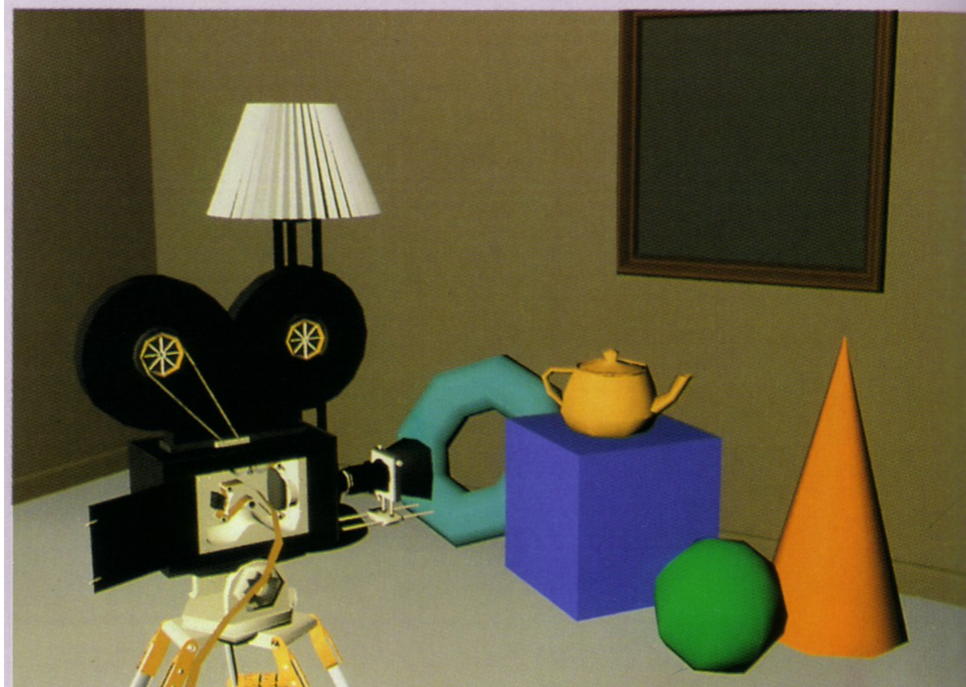
Gouraud shading

- Often we’re trying to draw smooth surfaces, so facets are an artifact
  - compute colors at vertices using vertex normals
  - interpolate colors across triangles
  - “Gouraud shading”
  - “Smooth shading”



[Gouraud thesis]

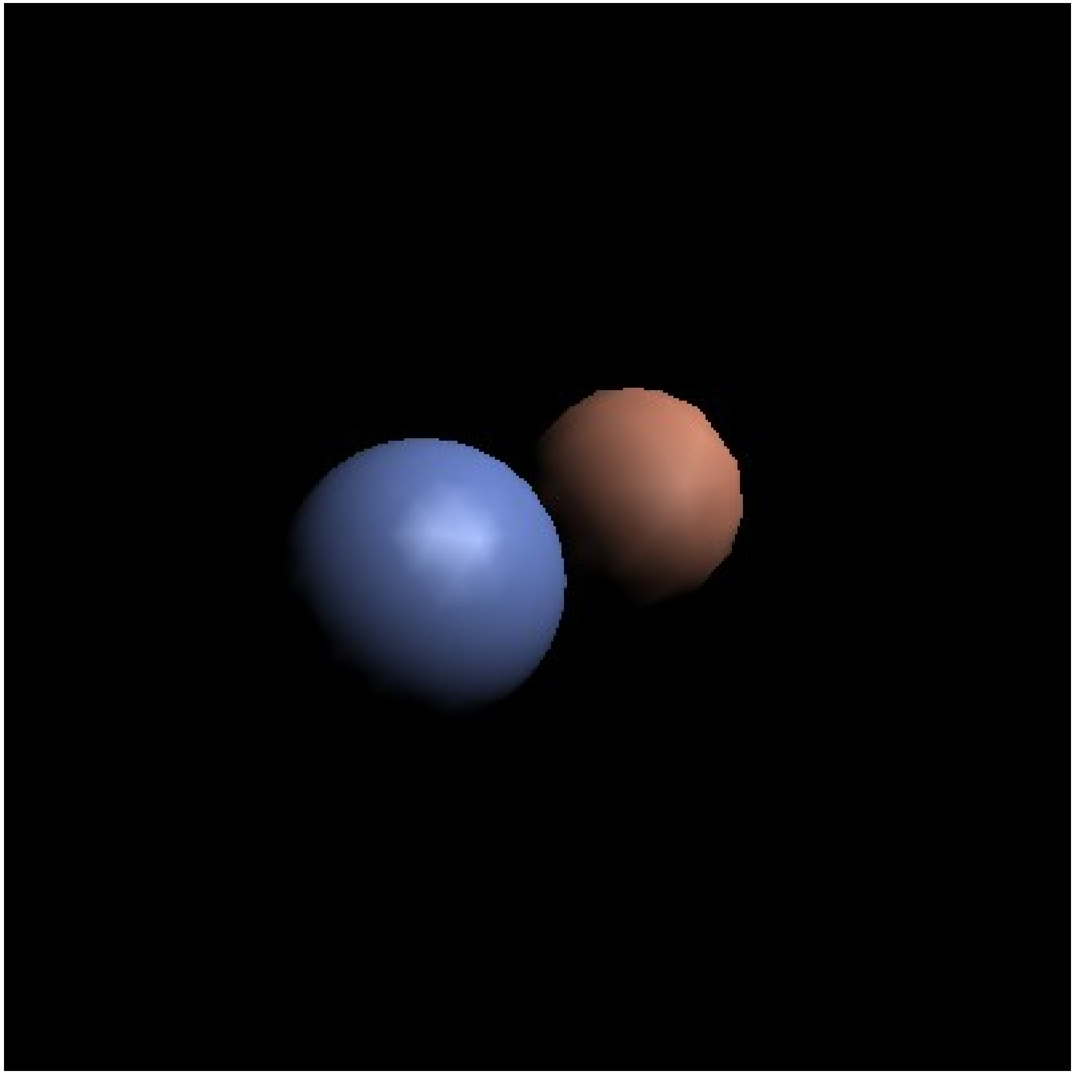
Plate II.30 *Shutterbug.* Gouraud shaded polygons with diffuse reflection (Sections 14.4.3 and 16.2.4). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar’s PhotoRealistic RenderMan™ software.)



Pipeline for Gouraud shading

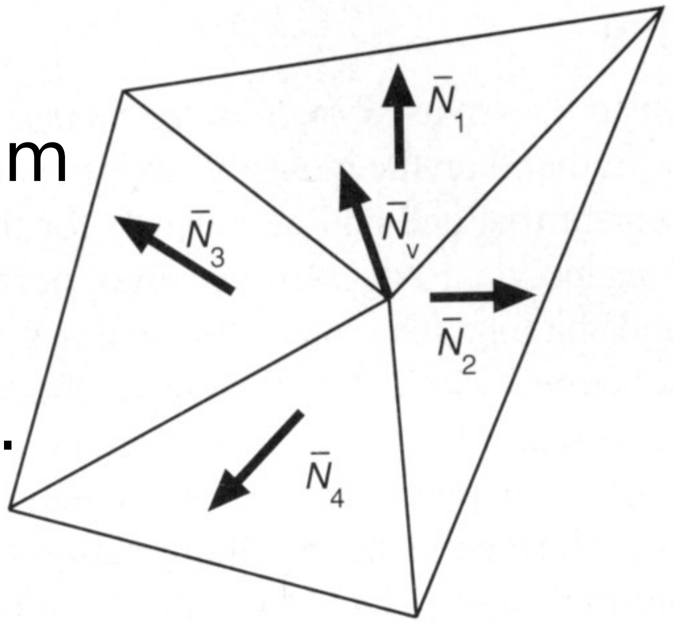
- Vertex stage (input: position, color, and normal / vtx)
  - transform position and normal (object to eye space)
  - compute shaded color per vertex
  - transform position (eye to screen space)
- Rasterizer
  - interpolated parameters: z' (screen z); r, g, b color
- Fragment stage (output: color, z')
  - write to color planes only if interpolated $z' <$ current z'

Result of Gouraud shading pipeline



Vertex normals

- Need normals at vertices to compute Gouraud shading
- Best to get vtx. normals from the underlying geometry
  - e. g. spheres example
- Otherwise have to infer vtx. normals from triangles
  - simple scheme: average surrounding face normals



[Foley et al.]

$$N_v = \frac{\sum_i N_i}{\|\sum_i N_i\|}$$

Non-diffuse Gouraud shading

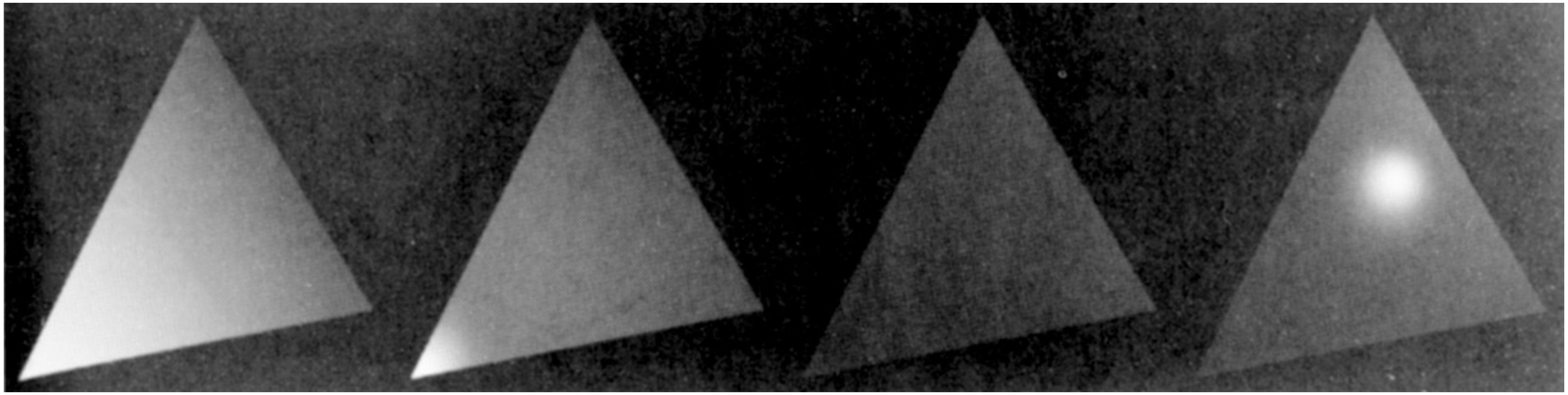
- Can apply Gouraud shading to any illumination model
  - it's just an interpolation method
- Results are not so
  models like specul
  - problems with any
    highlights smaller
    than a triangle

Plate II.31 *Shutterbug.* Gouraud shaded polygons with specular reflection (Sections 14.4.4 and 16.2.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

[Foley et al.]

Phong shading

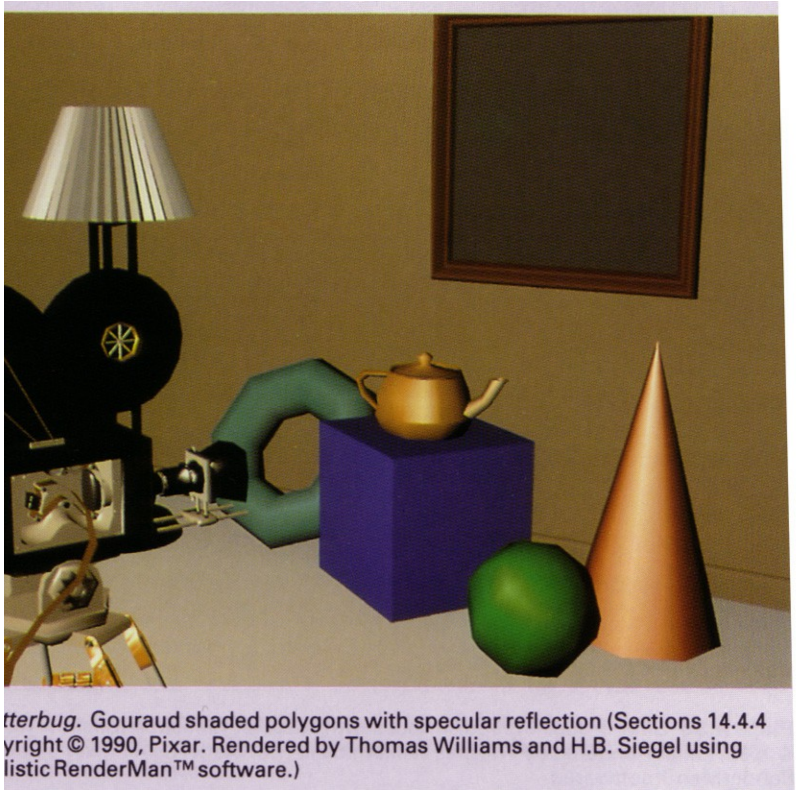
- Get higher quality by interpolating the normal
  - just as easy as interpolating the color
  - but now we are evaluating the illumination model per pixel rather than per vertex (and normalizing the normal first)
  - in pipeline, this means we are moving illumination from the vertex processing stage to the fragment processing stage

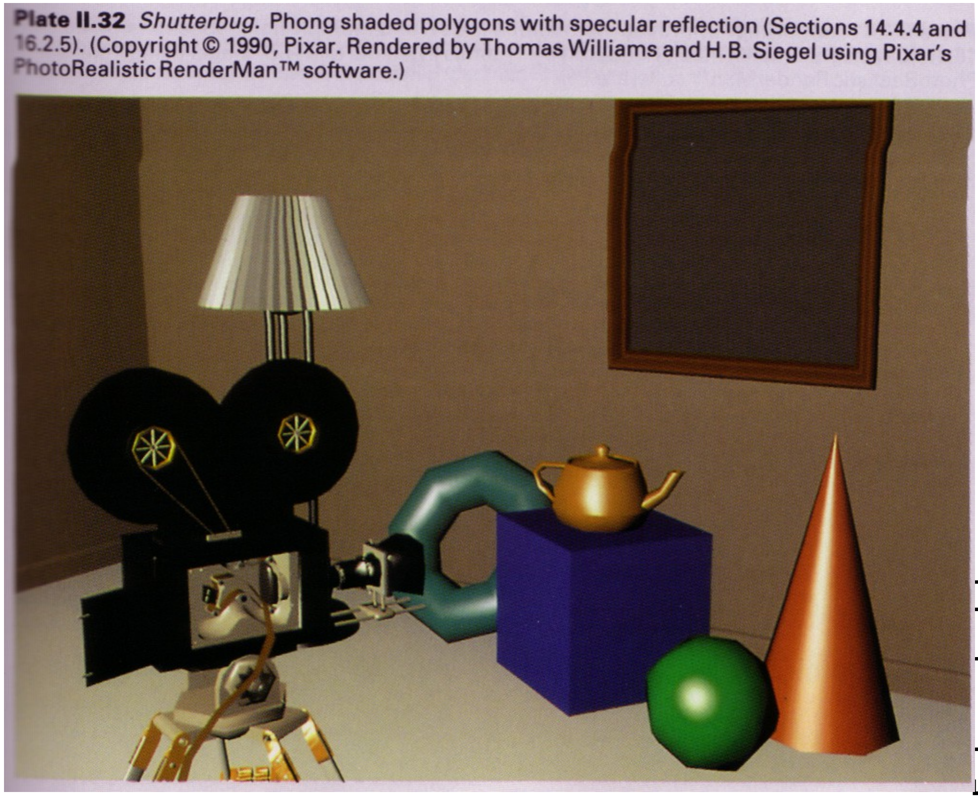


[Foley et al.]

Phong shading

- Bottom line: produces much better highlights





[Foley et al.]

Pipeline for Phong shading

- Vertex stage (input: position, color, and normal / vtx)
  - transform position and normal (object to eye space)
  - transform position (eye to screen space)
  - pass through color
- Rasterizer
  - interpolated parameters: z' (screen z); r, g, b color; x, y, z normal
- Fragment stage (output: color, z')
  - compute shading using interpolated color and normal
  - write to color planes only if interpolated $z' <$ current z'

Result of Phong shading pipeline