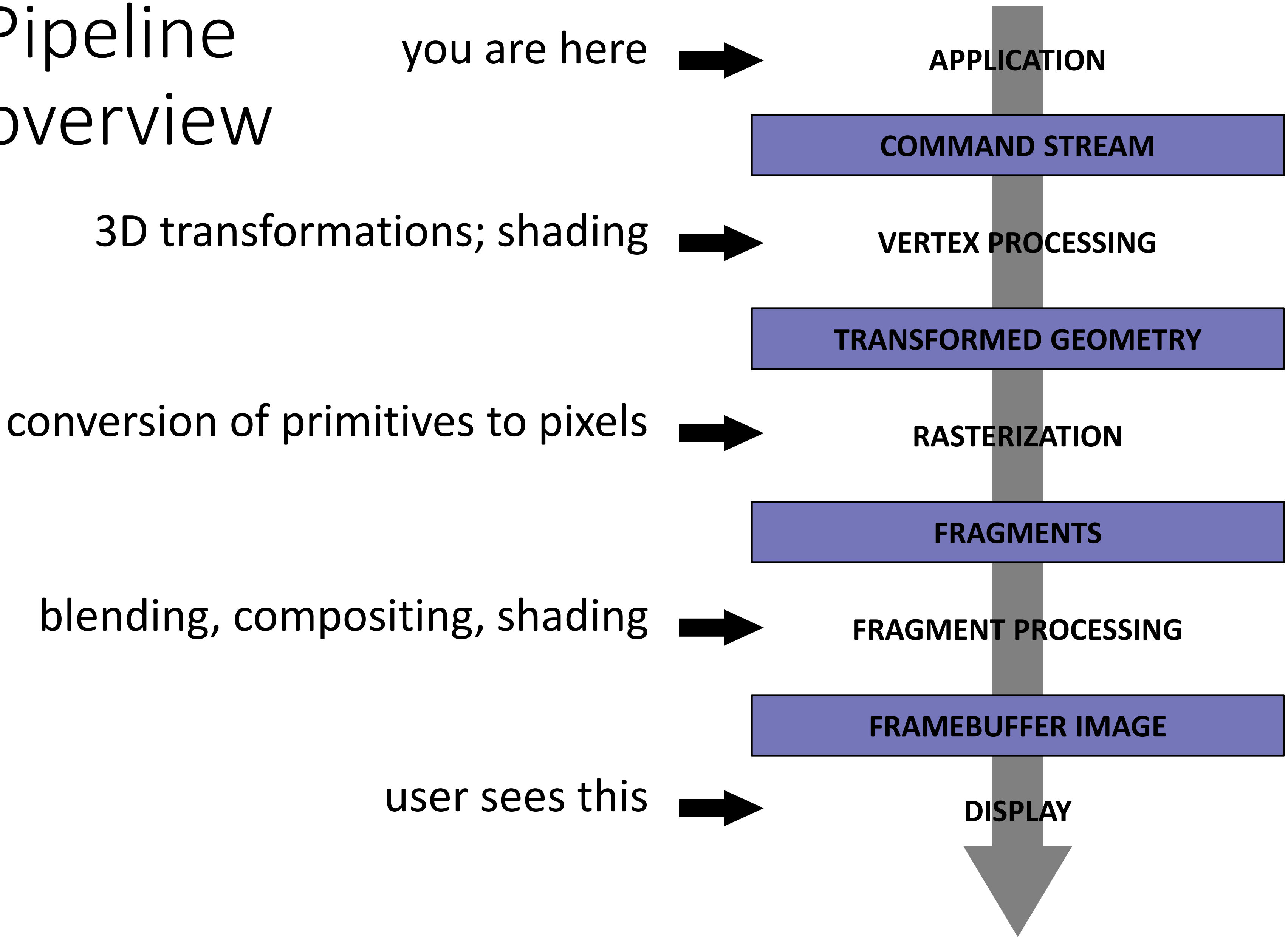
Pipeline Operations

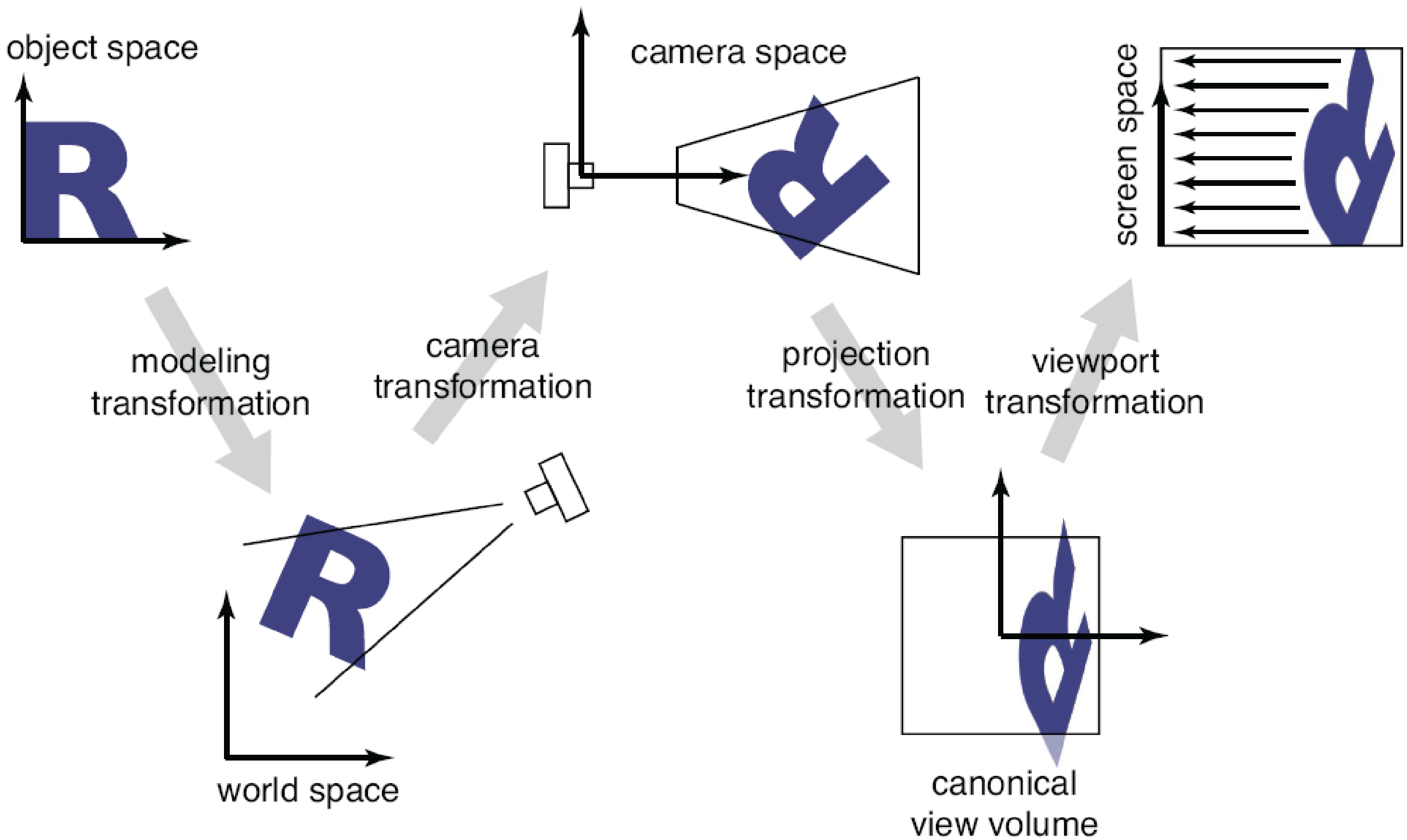
Based on slides by Steve Marschner

Pipeline overview



Pipeline of transformations

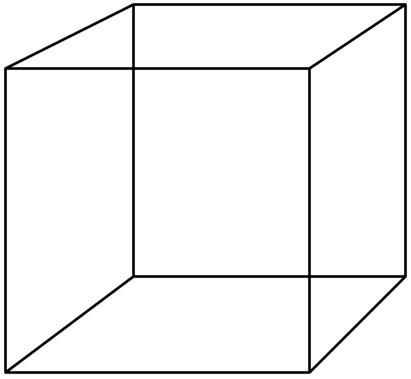
- Standard sequence of transforms

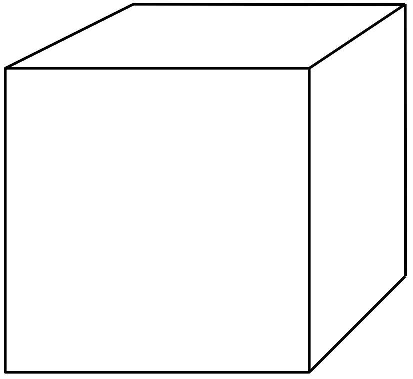


Based on slides by Steve Marschner

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

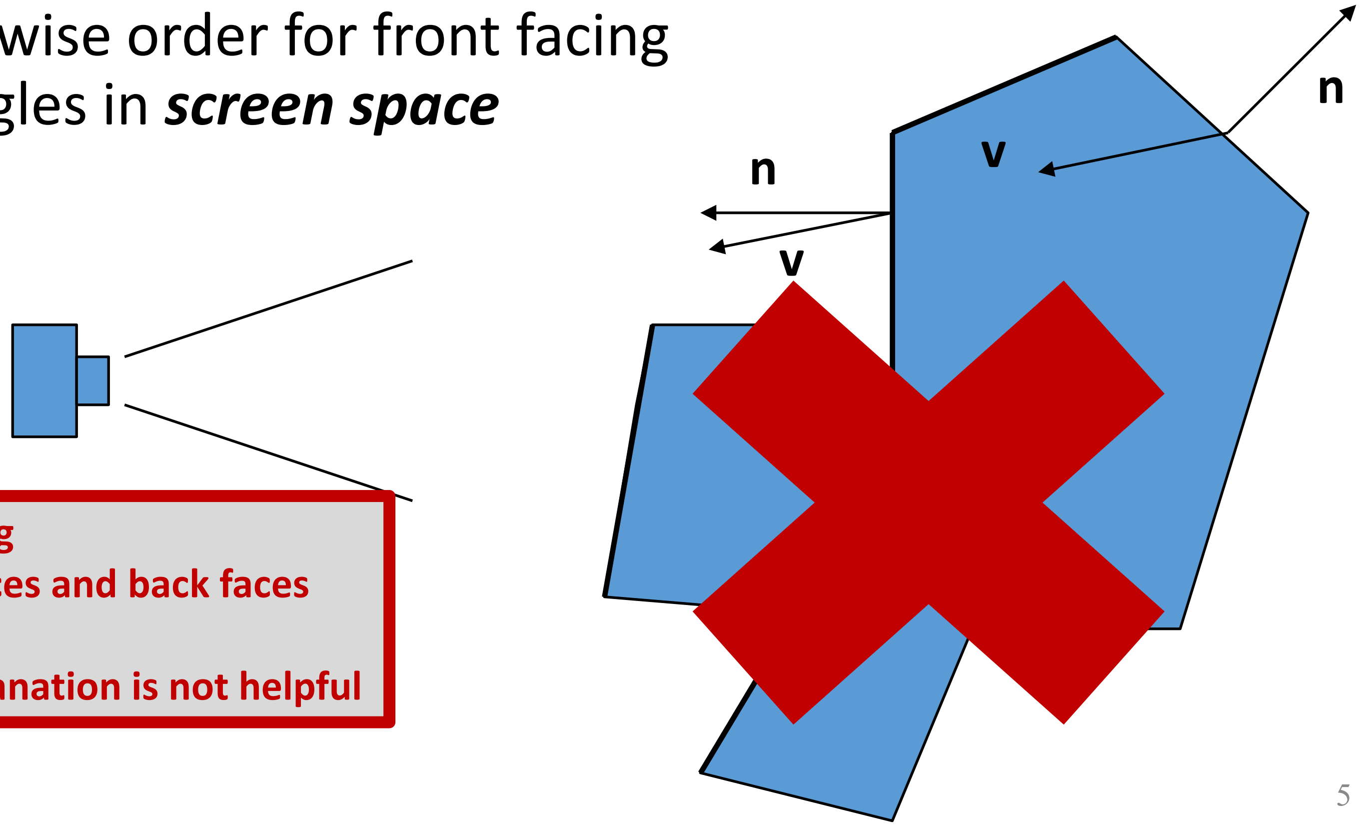




Based on slides by Steve Marschner

Back face culling

- For closed shapes you will never see the inside
  - therefore only draw surfaces that face the camera
  - ~~*Could* implement by checking **n · v** but v varies across the surface (might want n to vary too!)~~
  - ***Actually*** implemented by checking counter clockwise order for front facing triangles in ***screen space***



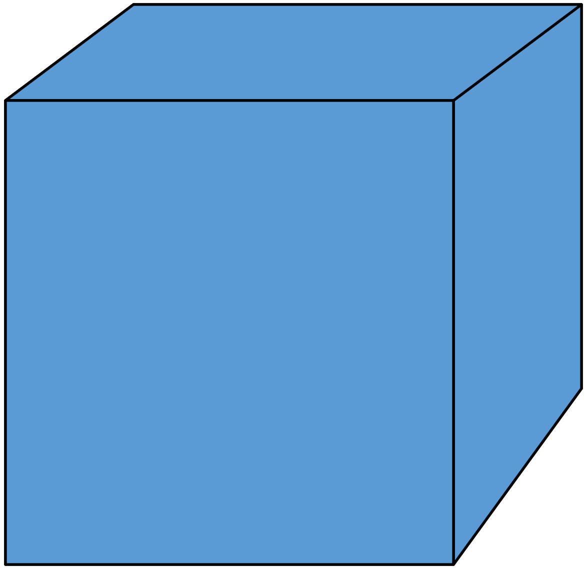
USE A 3D drawing
To show front faces and back faces

This n dot v explanation is not helpful

Based on slides by Steve Marschner

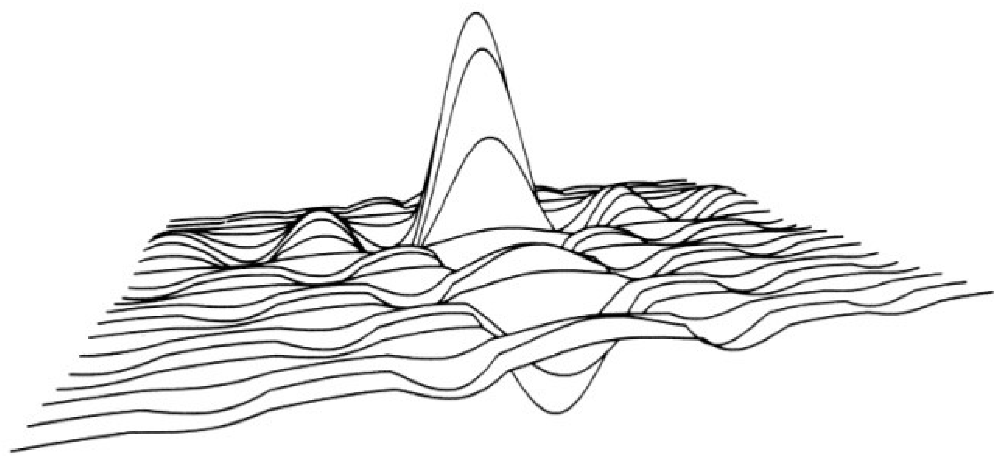
Painter's algorithm

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



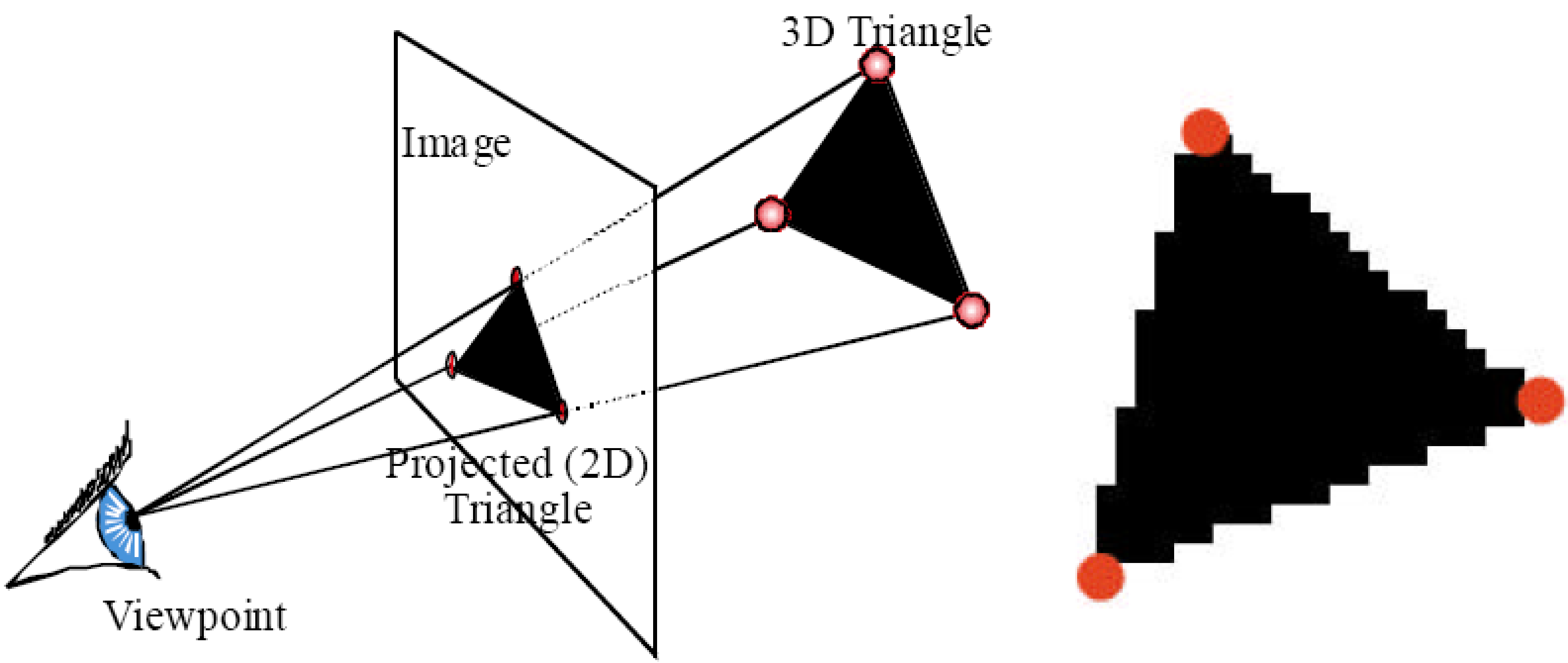
Painter's algorithm

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



[Foley et al.]

Drawing

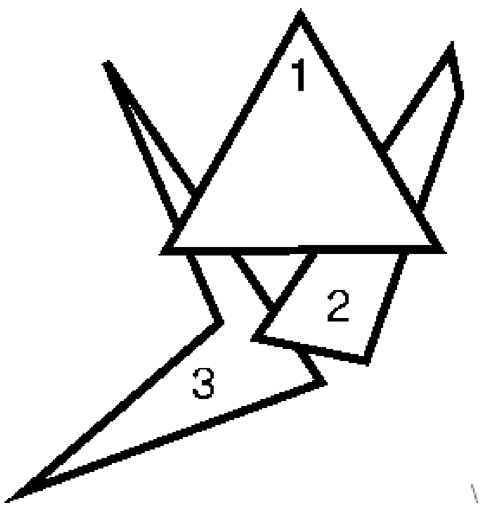


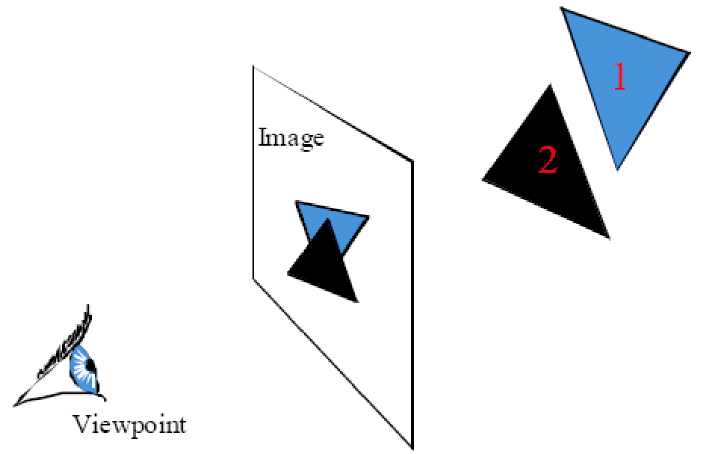
Projection (left) and rasterization (right) of a triangle.

[Durand]

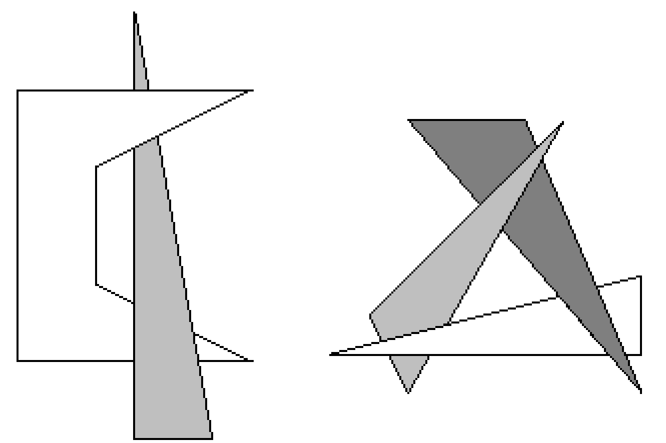
Visibility

- Painter's algorithm
  - Sort back to front
  - Draw!





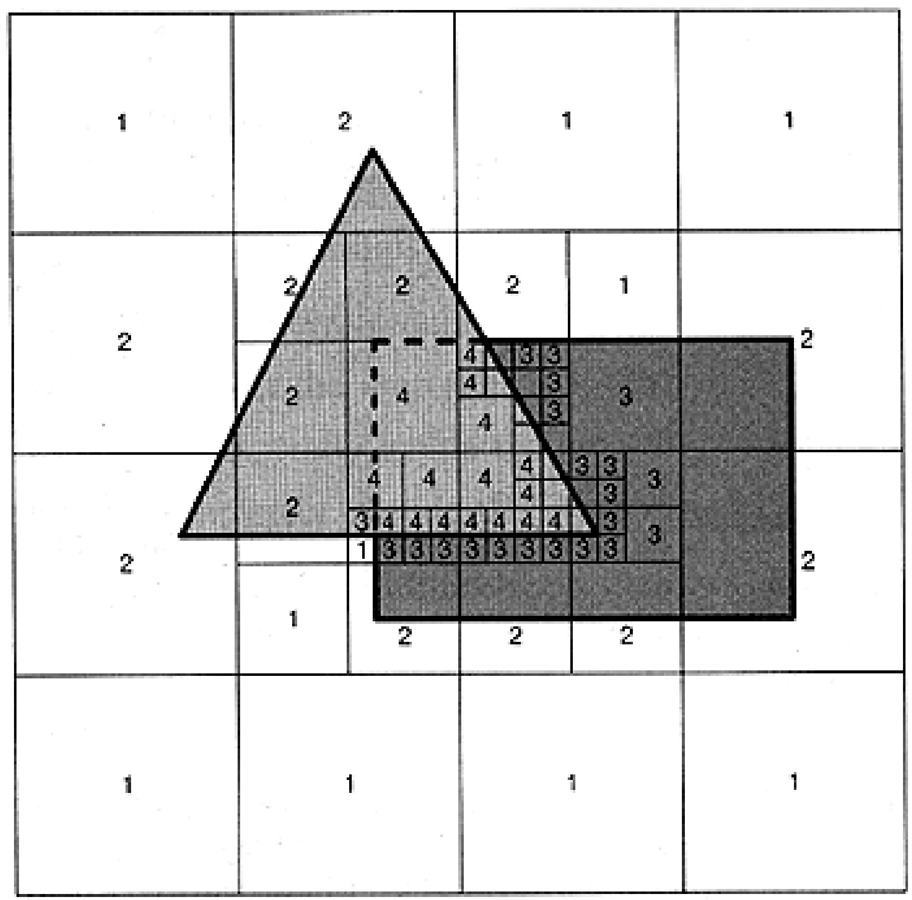
  - Doesn't always work… ☹



Based on slides by Steve Marschner

Visibility

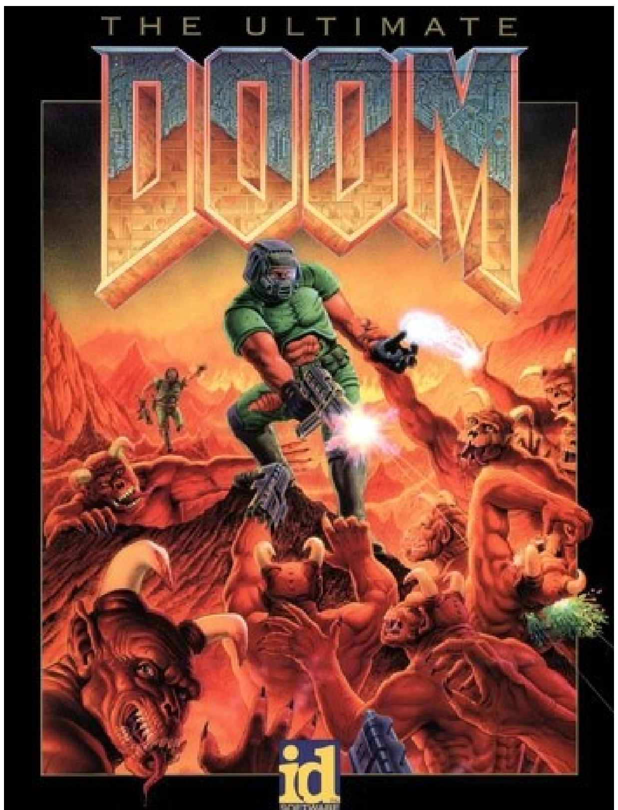
- Warnock's algorithm
  - Area subdivision
  - Apply Painter's when it will work (e.g., individual pixels)



Based on slides by Steve Marschner

Visibility

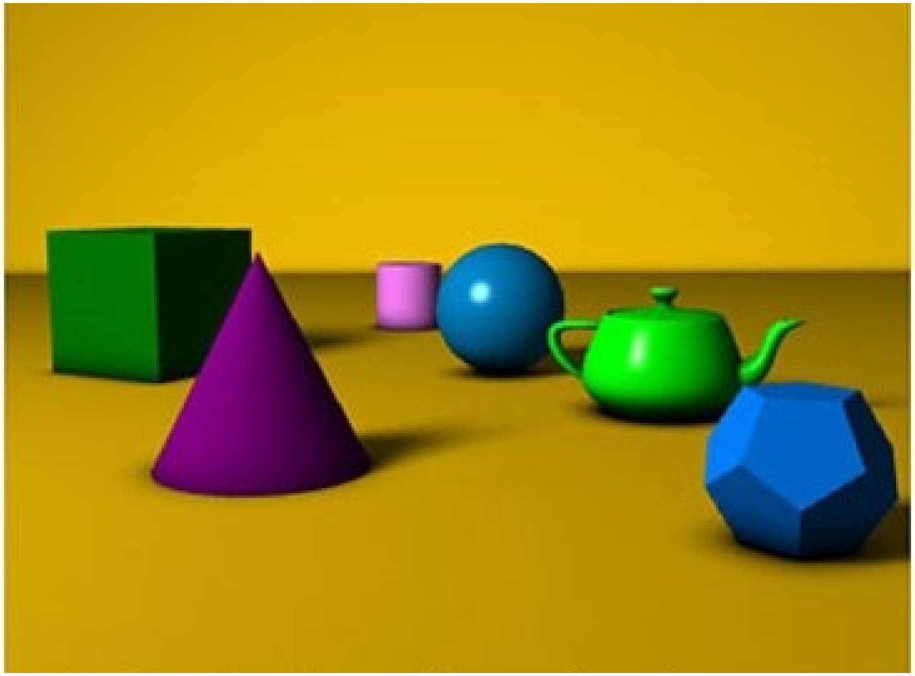
- Binary space partition
  - Linear time back to front sort
  - Key to 3D games before consumer level GPUs (Doom 1993)



Based on slides by Steve Marschner

Visibility

- Z-buffer
  - Store depth at every pixel
  - Compare when rasterizing

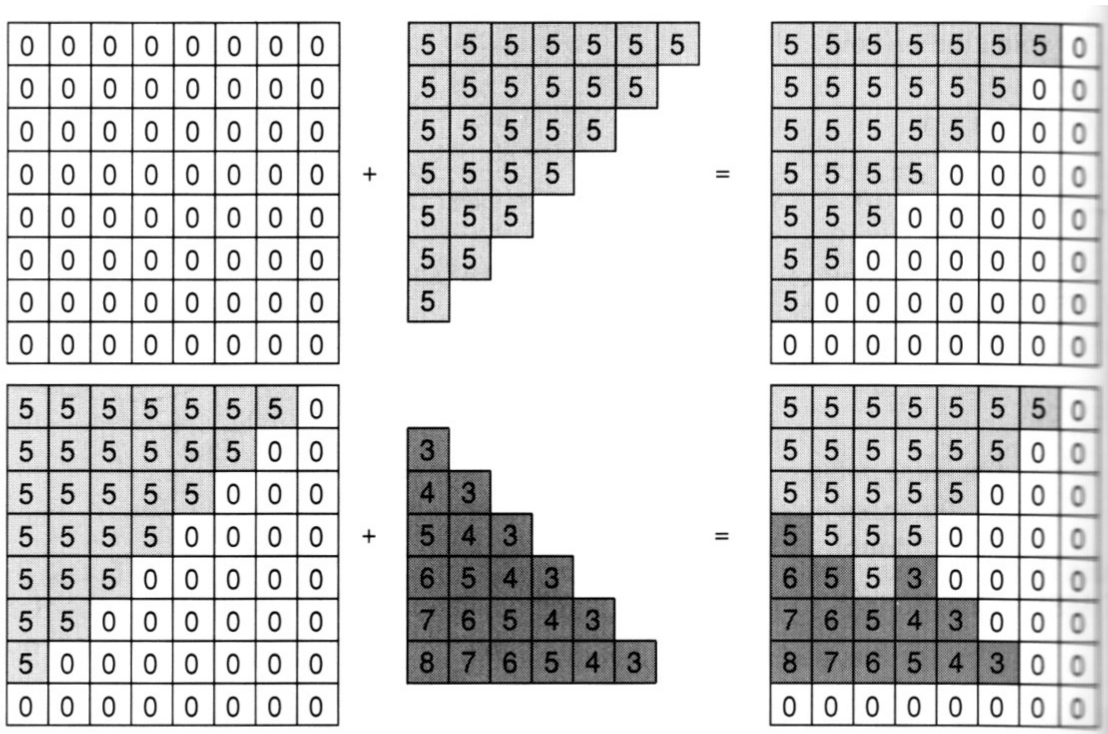


Based on slides by Steve Marschner

The *z* buffer

- In many (most) applications maintaining a *z* sort is too expensive
  - changes all the time as the view changes
  - many data structures exist, but complex
- Solution: draw in any order, keep track of closest
  - allocate extra channel per pixel to keep track of closest depth so far
  - when drawing, compare object's depth to current closest depth and discard if greater
  - this works just like any other compositing operation

The *z* buffer



[Foley et al.]

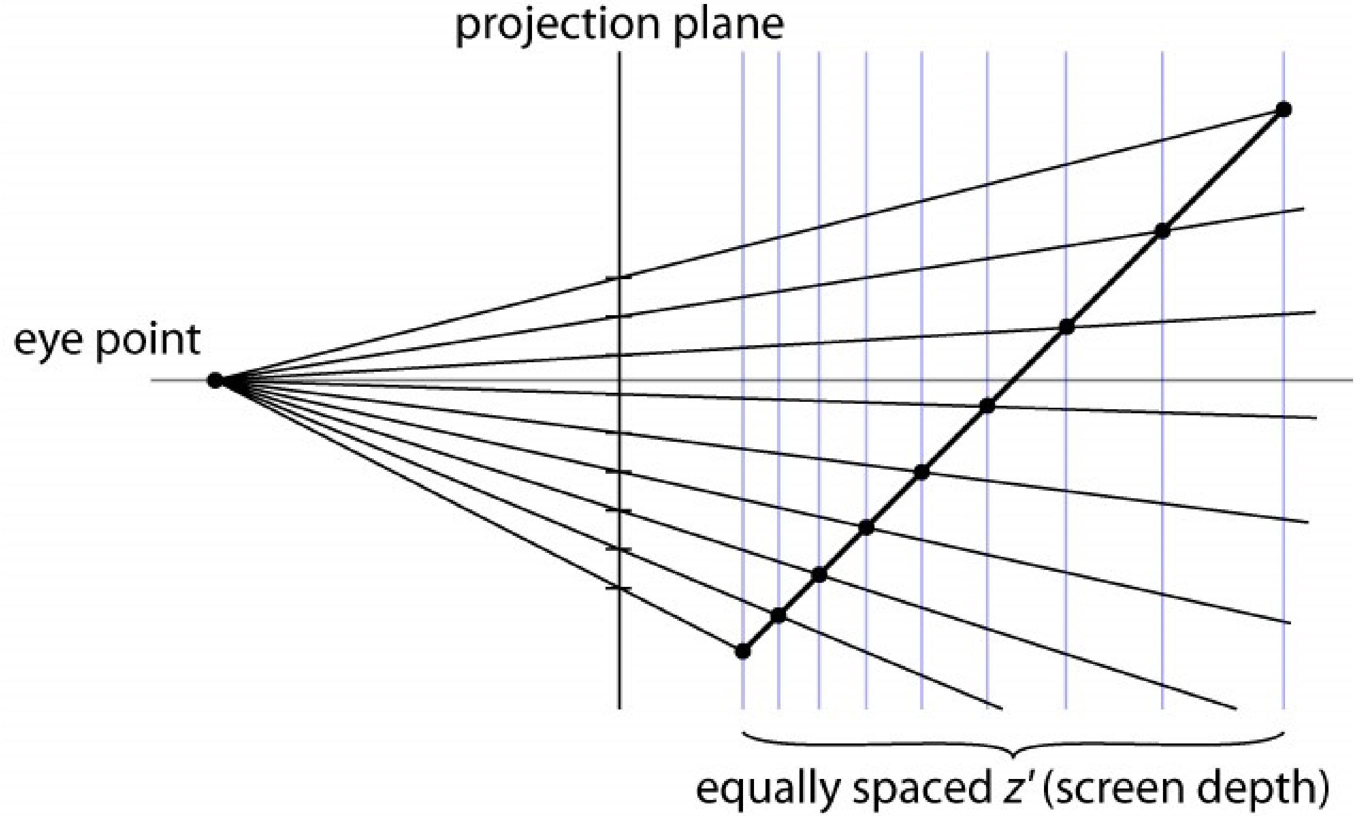
- another example of a memory-intensive brute force approach that works and has become the standard

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

Based on slides by Steve Marschner

Interpolating in projection



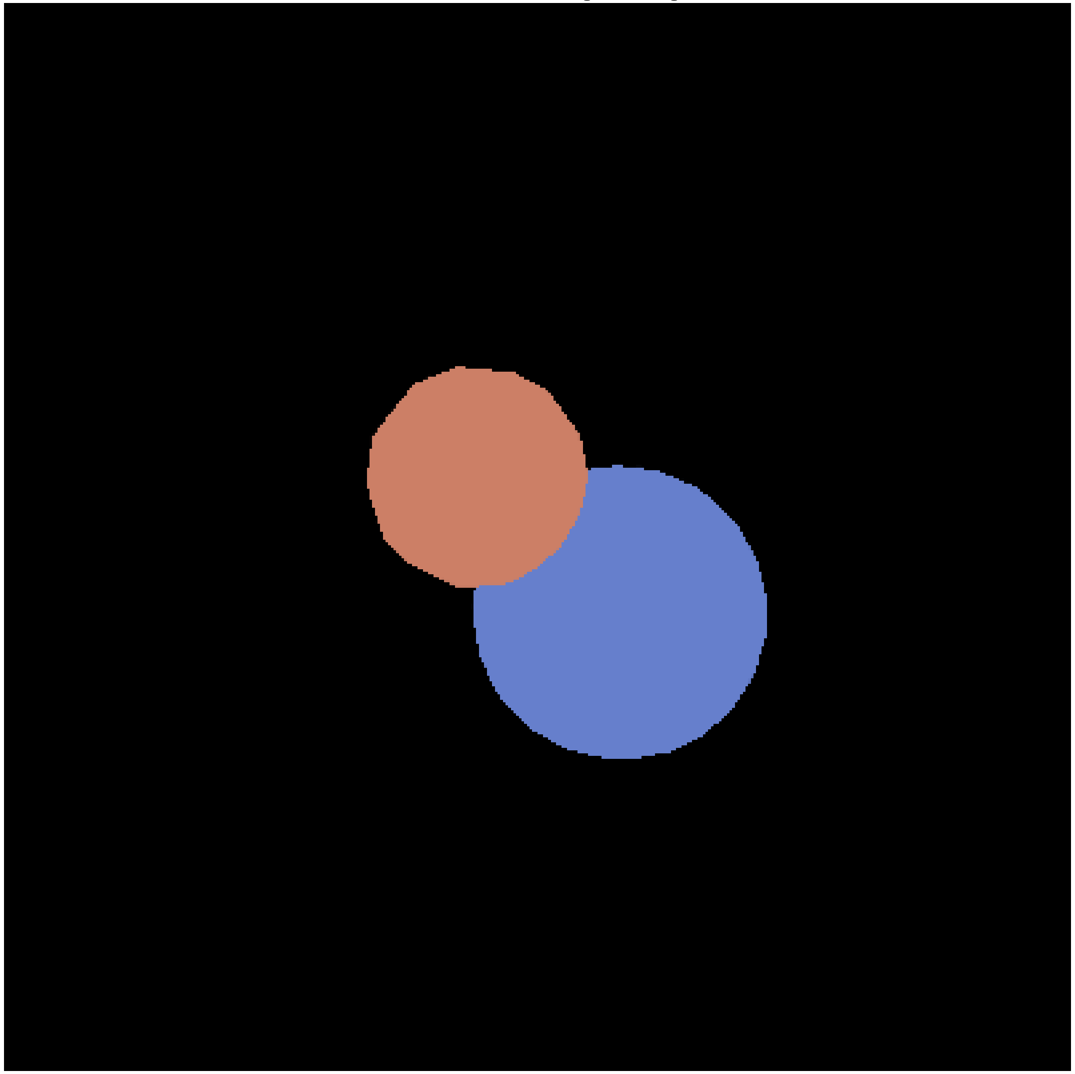
linear interp. in screen space ≠ linear interp. in world (eye) space

More precision close to near plane (can revisit projection demo)

Pipeline for minimal operation

- Vertex stage (input: position / vtx; color / tri)
  - transform position (object to screen space)
  - pass through color
- Rasterizer
  - pass through color
- Fragment stage (output: color)
  - write to color planes

Result of minimal pipeline

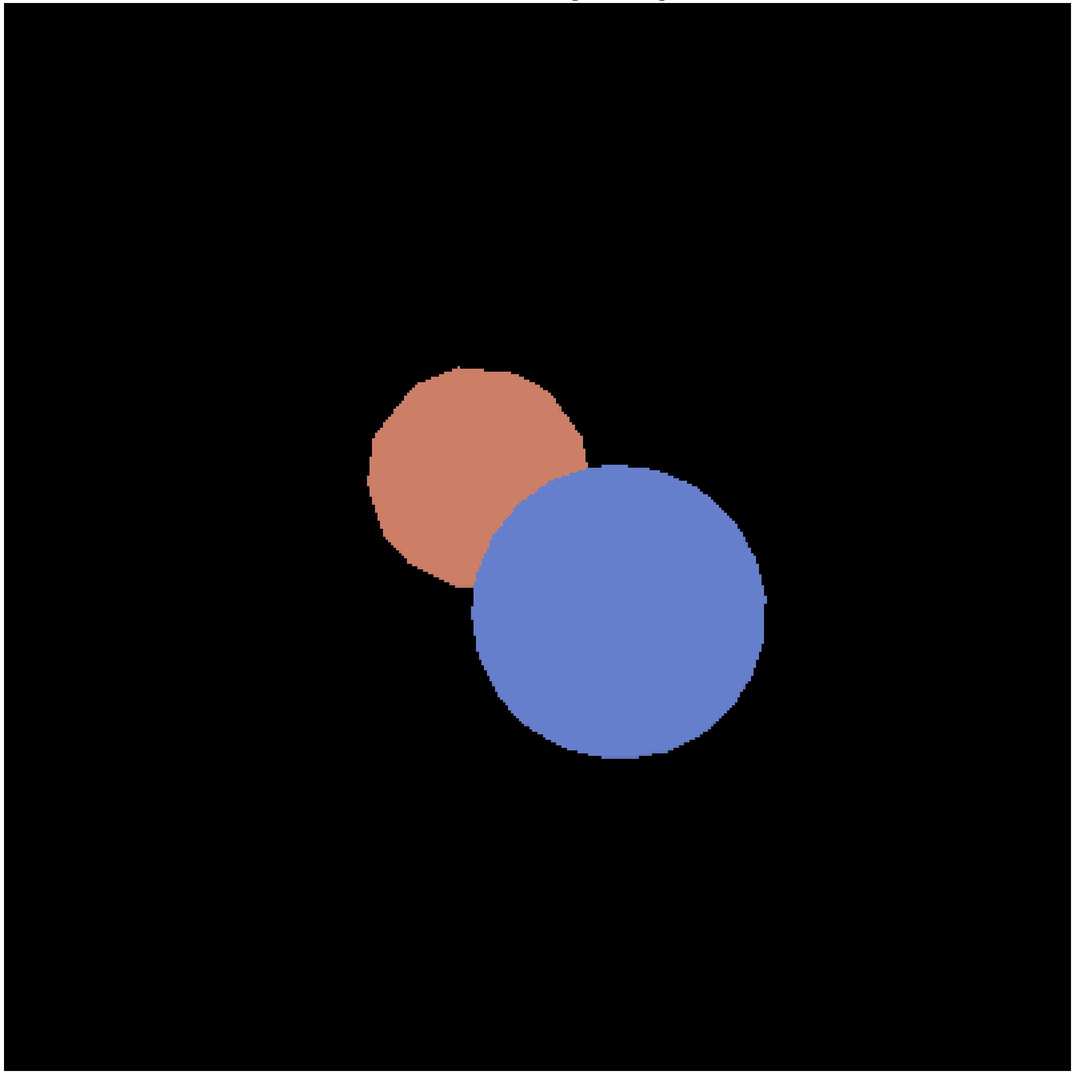


Pipeline for basic *z* buffer

- 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'

Based on slides by Steve Marschner

Result of *z*-buffer pipeline



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

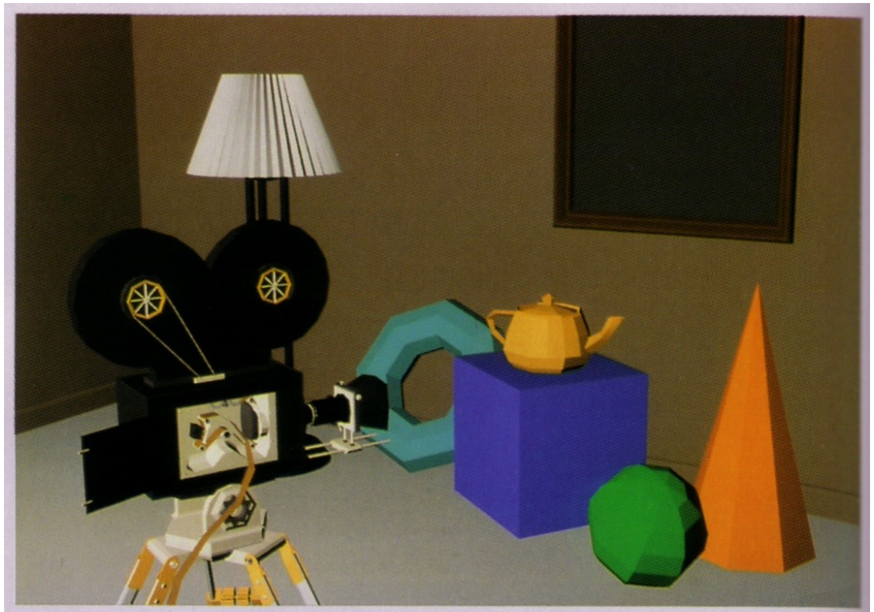


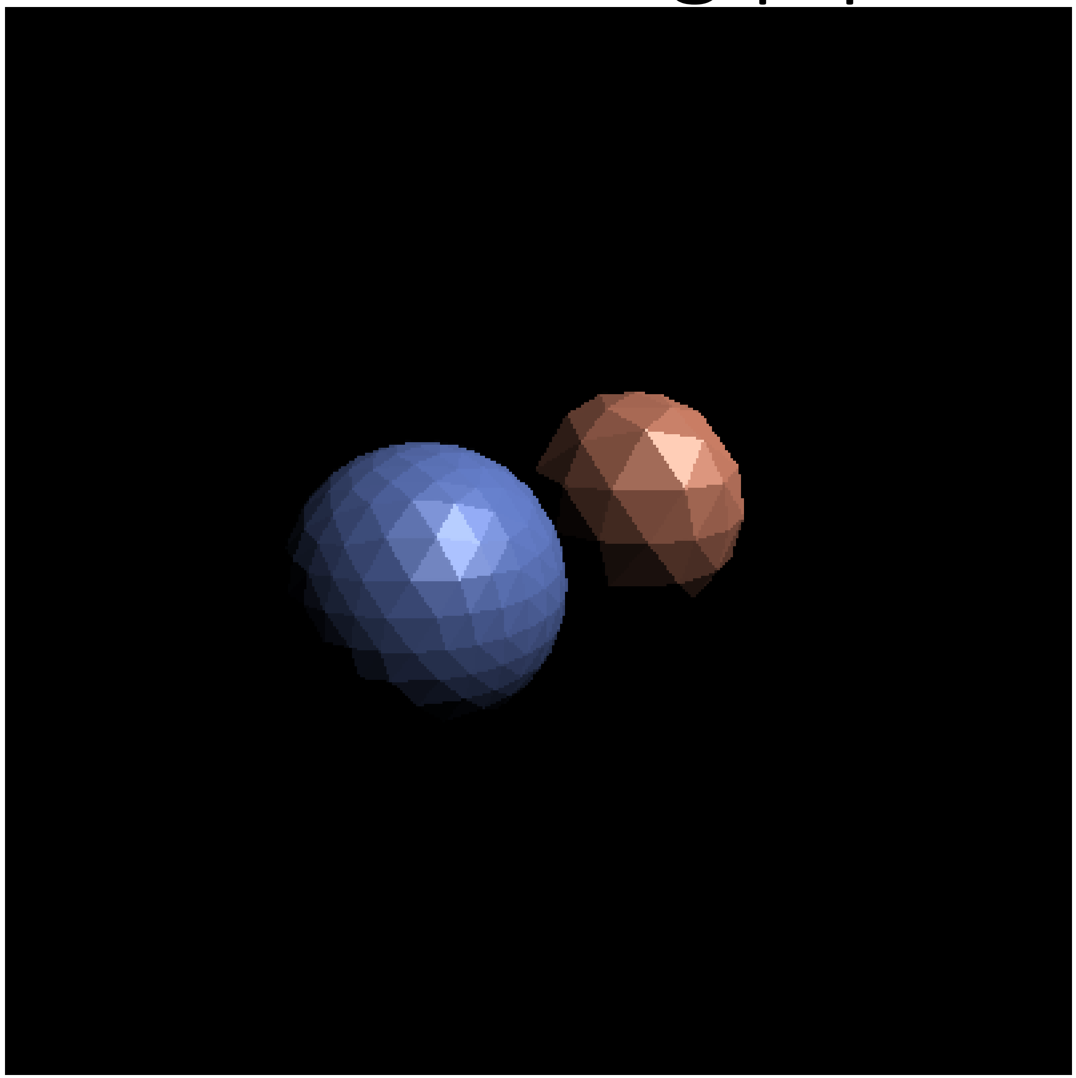
Plate II.29 *Shutterbug.* Individually shaded polygons with diffuse reflection (Sections 14.4.2 and 16.2.3). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

[Foley et al.]

Pipeline for flat shading

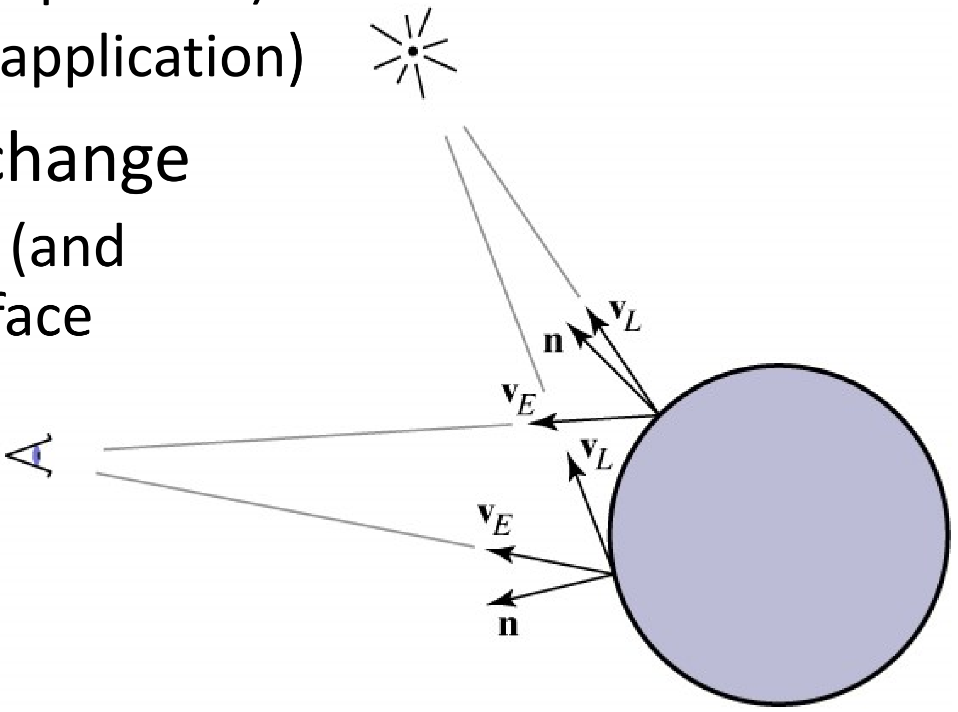
- Vertex stage (input: position / vtx; color and normal / tri)
  - transform position and normal (object to eye space)
  - compute shaded color per triangle using normal
  - transform position (eye to screen space)
- Rasterizer
  - interpolated parameters: z' (screen z)
  - pass through color
- Fragment 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



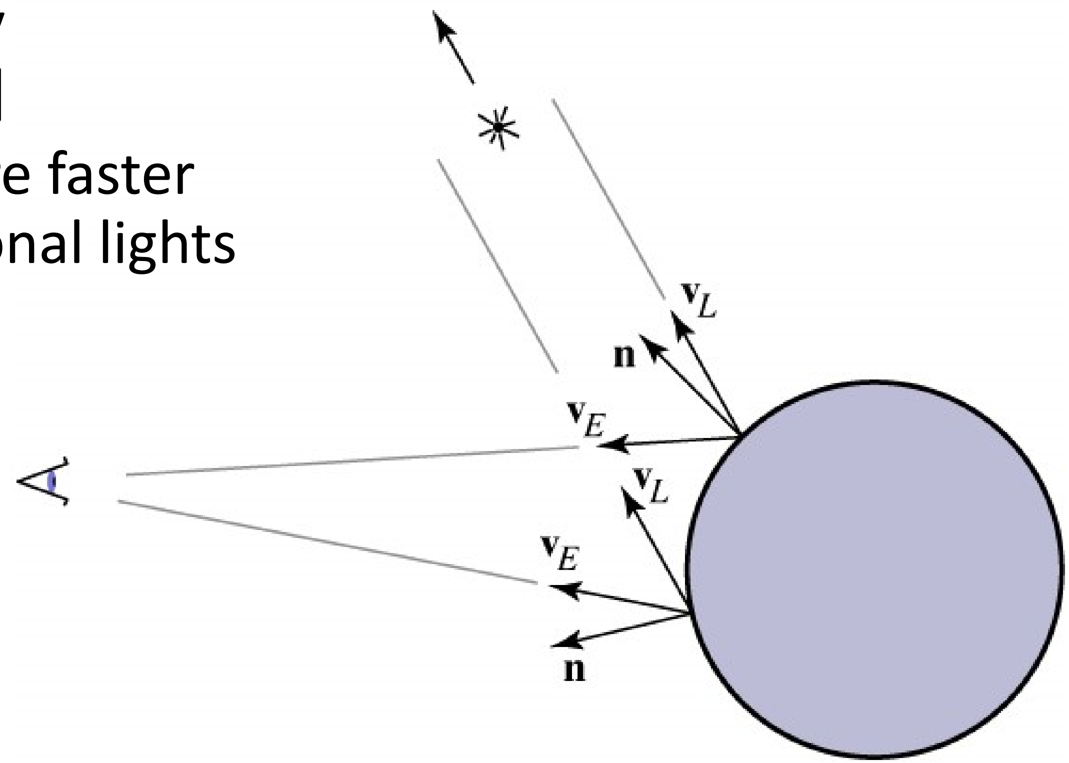
Based on slides by Steve Marschner

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

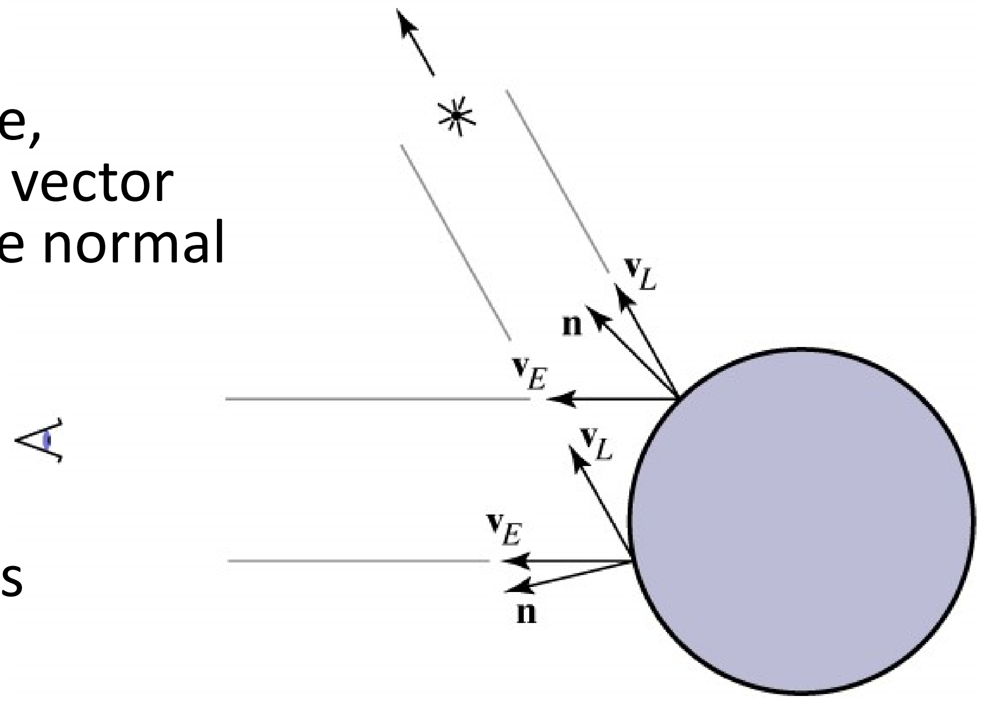
- Directional (infinitely distant) light source
  - light vector always points in the same direction
  - often specified by position [x y z 0]
  - many pipelines are faster if you use directional lights



Based on slides by Steve Marschner

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!

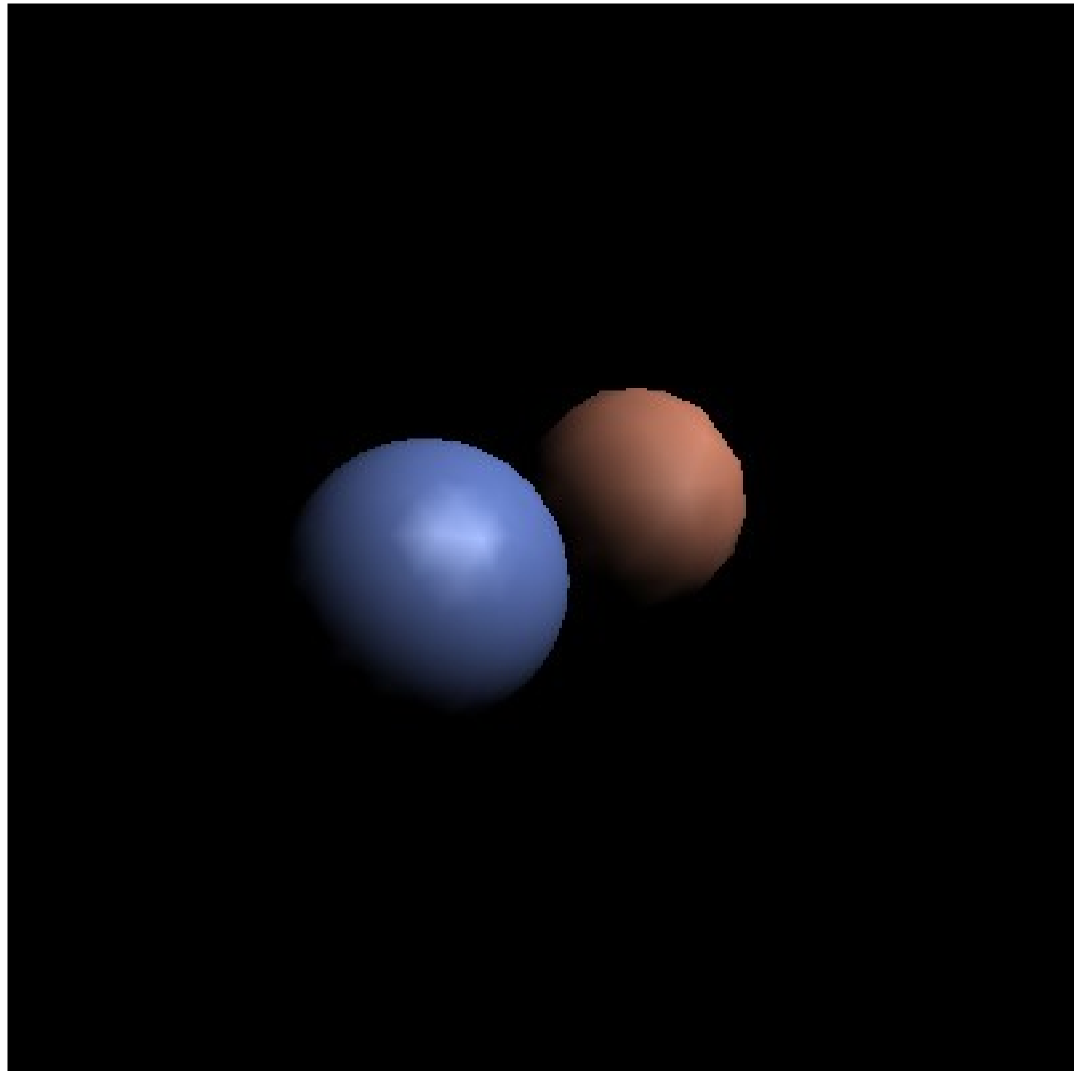


Based on slides by Steve Marschner

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'

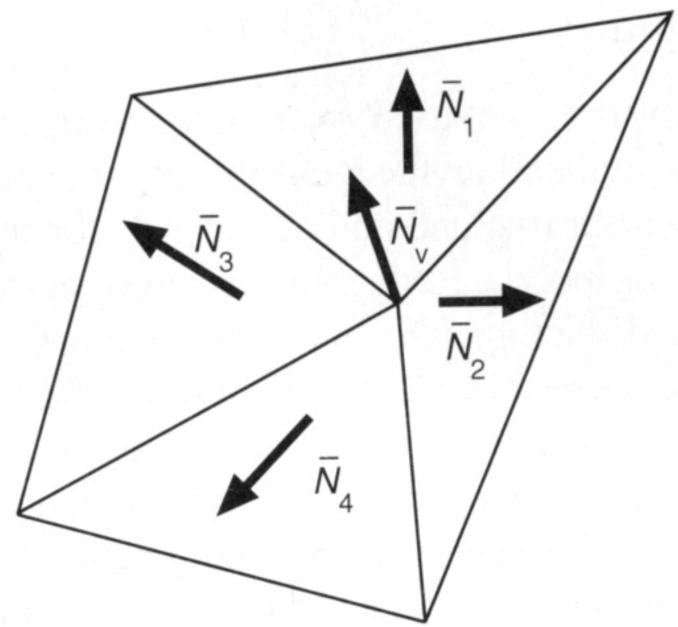
Result of Gouraud shading pipeline



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

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



[Foley et al.]

Based on slides by Steve Marschner

Non-diffuse Gouraud shading

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

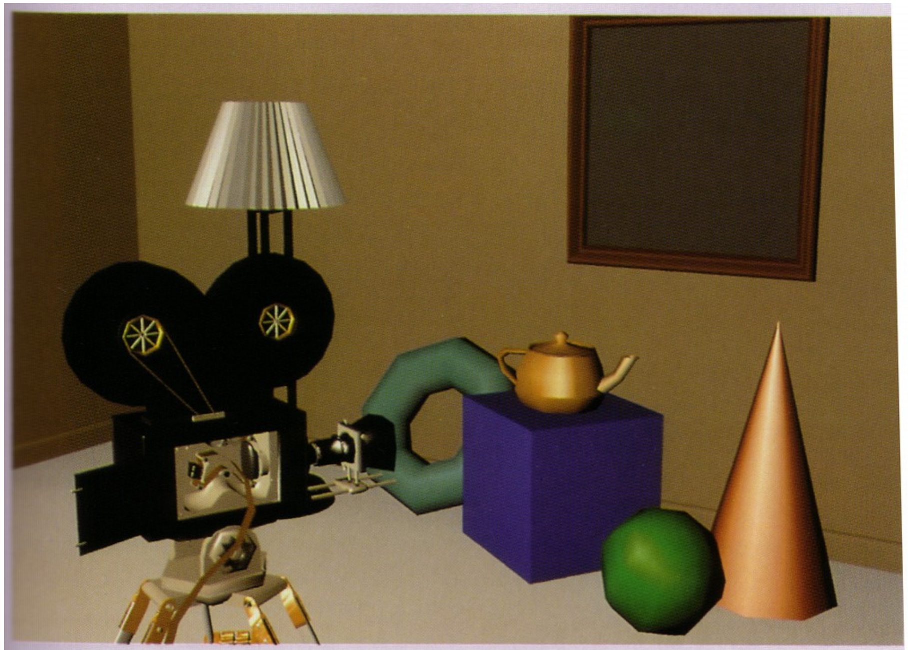
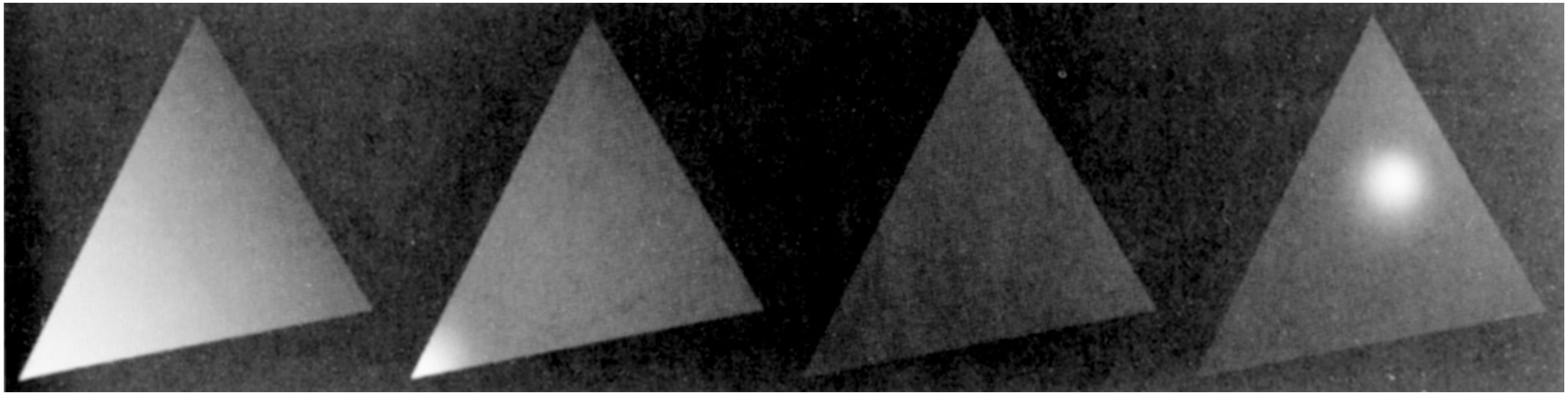


Plate II.31 *Shutterbug.* Gouraud shaded polygons with specular reflection (Sections 14.4.4 and 16.2.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

[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.]

Based on slides by Steve Marschner

Phong shading

- Bottom line: produces much better highlights

tterbug. Gouraud shaded polygons with specular reflection (Sections 14.4.4
yright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using
listic RenderMan™ software.)

Plate II.32 *Shutterbug.* Phong 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.]

Based on slides by Steve Marschner

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