Pipeline Operations

CS 4620 Lecture 13

Pipeline

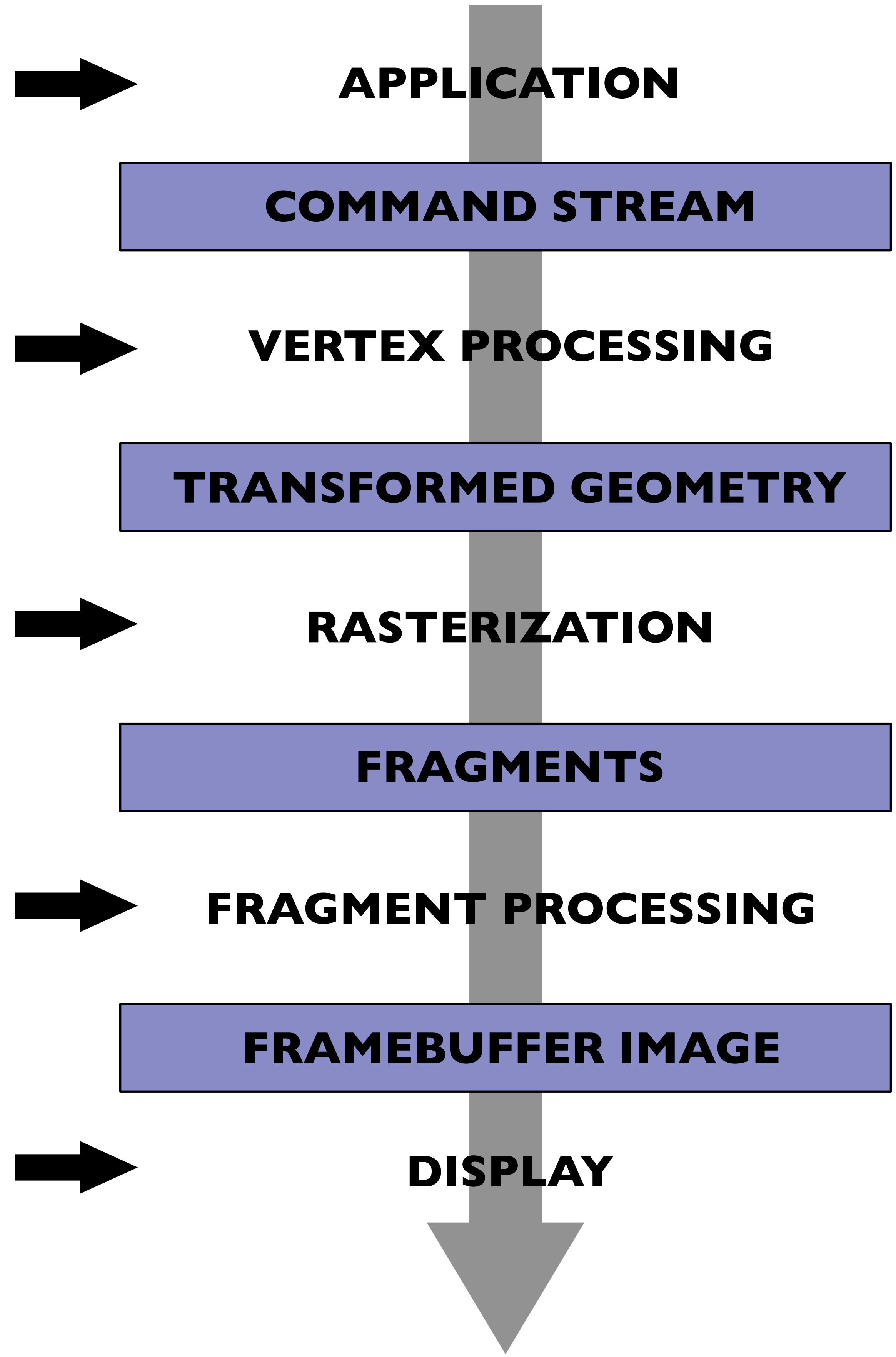
you are here

3D transformations; shading

conversion of primitives to pixels

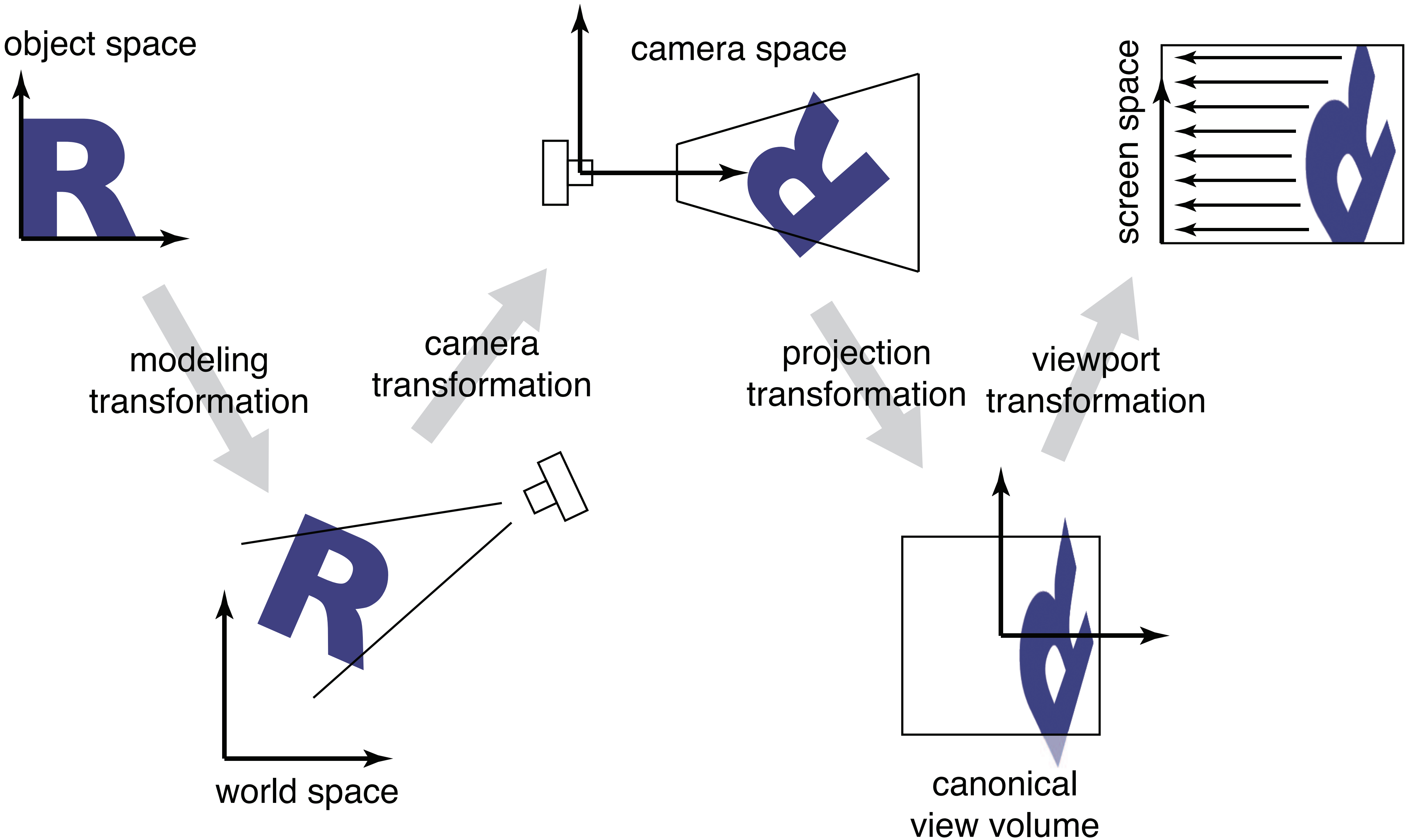
blending, compositing, shading

user sees this



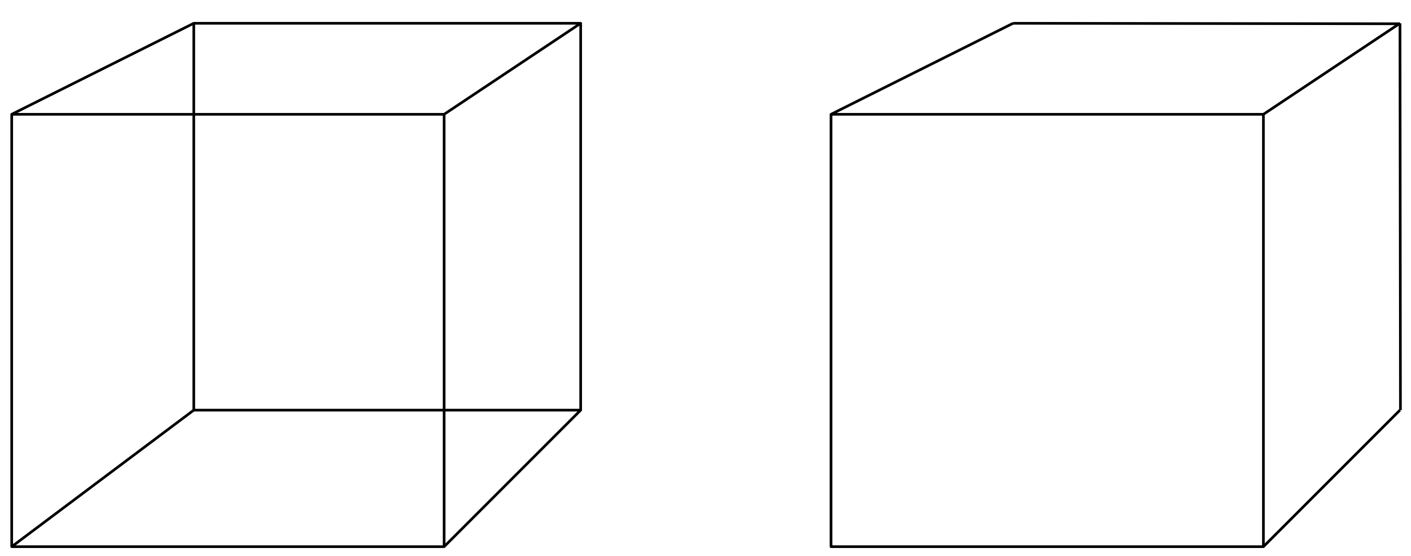
Pipeline of transformations

- **Standard sequence of transforms**



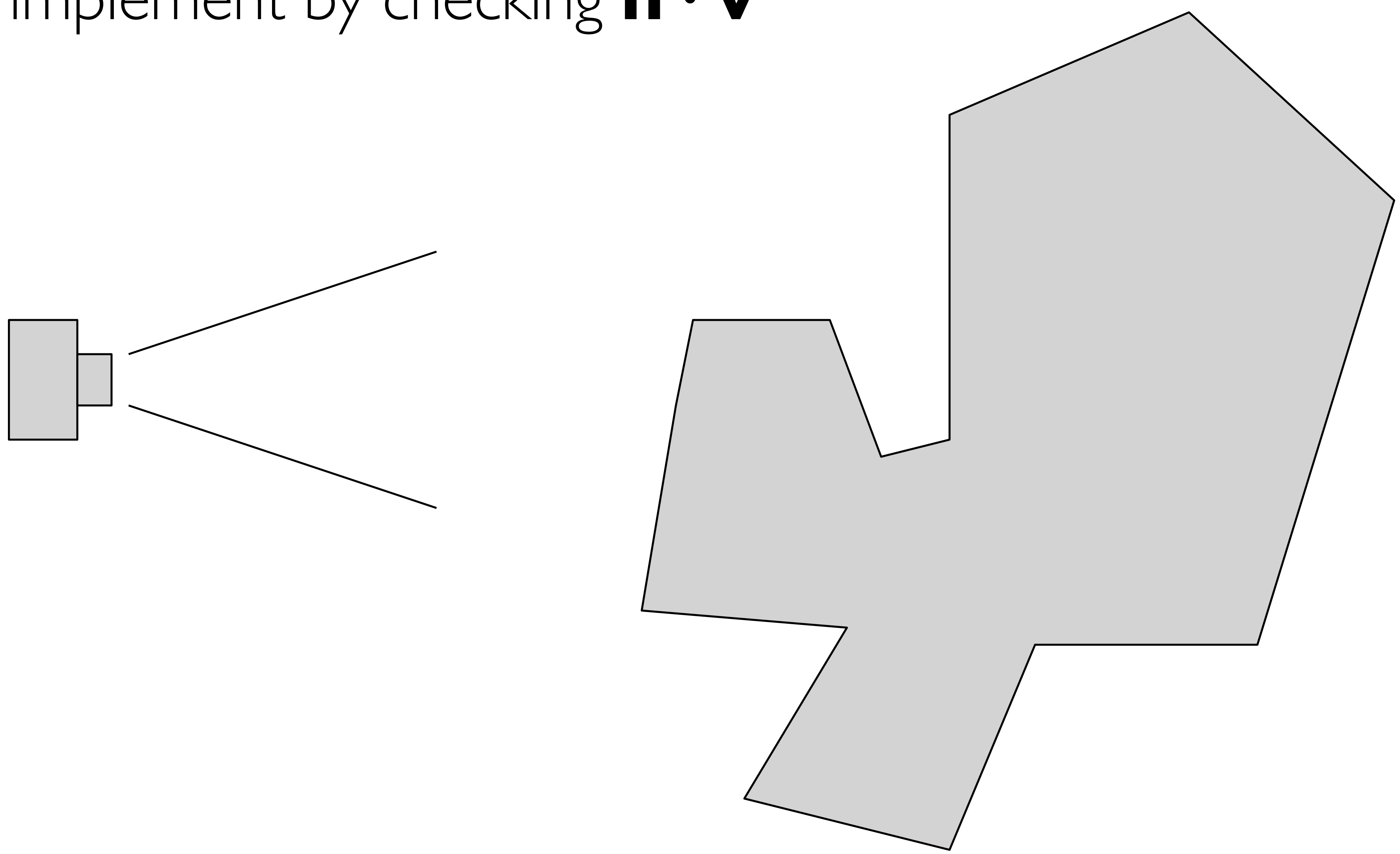
Hidden surface elimination

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



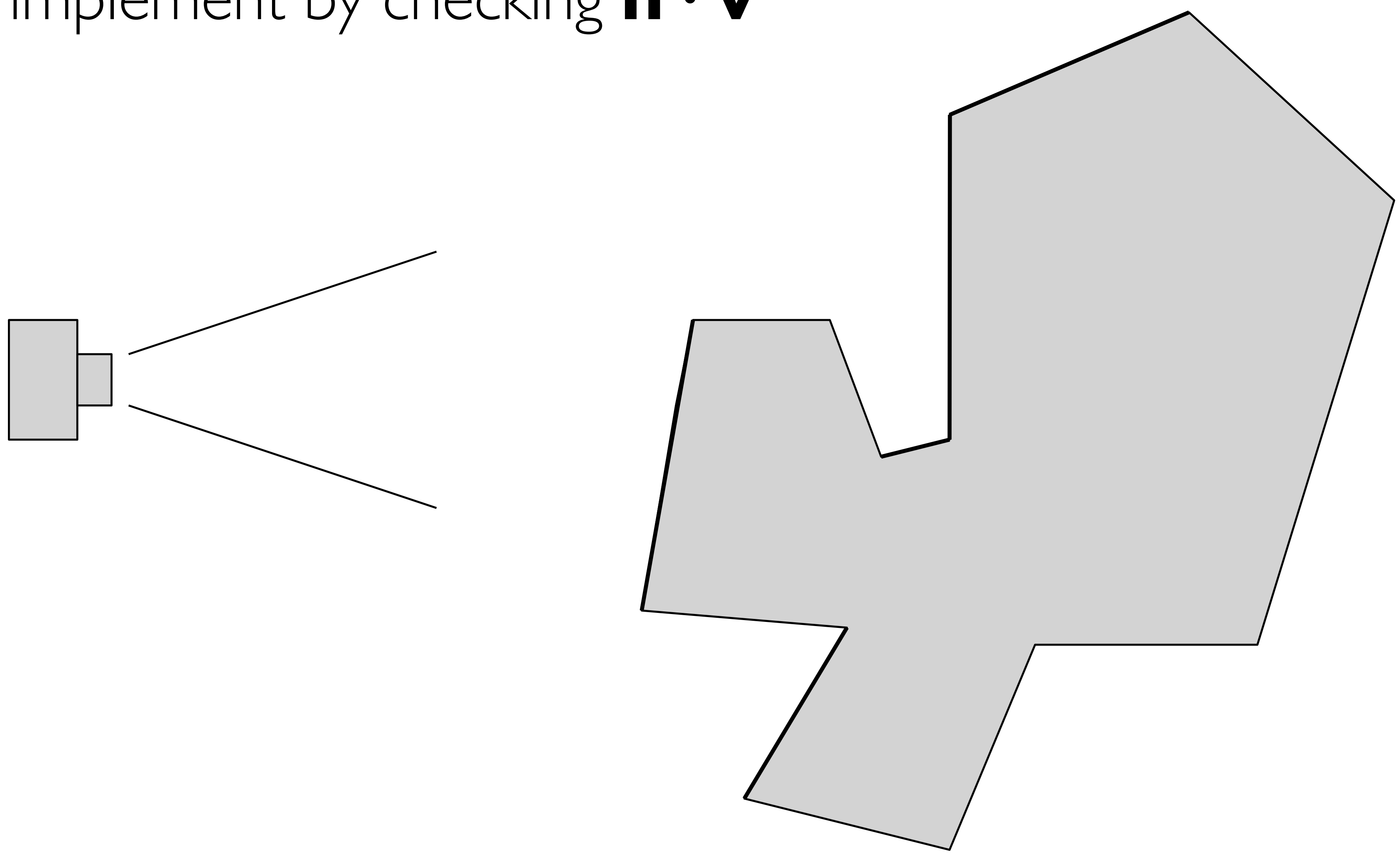
Back face culling

- **For closed shapes you will never see the inside**
  - therefore only draw surfaces that face the camera
  - implement by checking $\mathbf{n} \cdot \mathbf{v}$



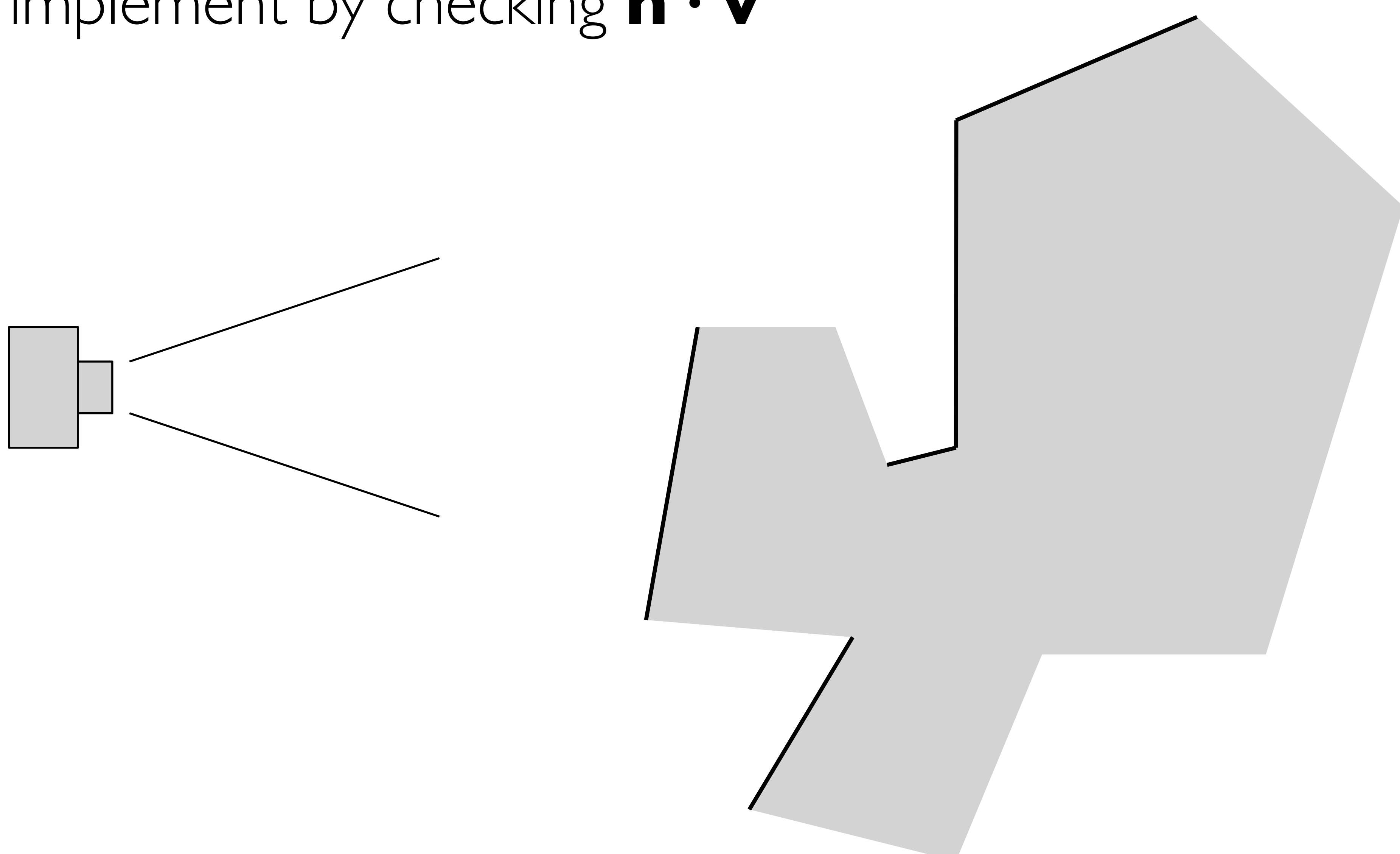
Back face culling

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



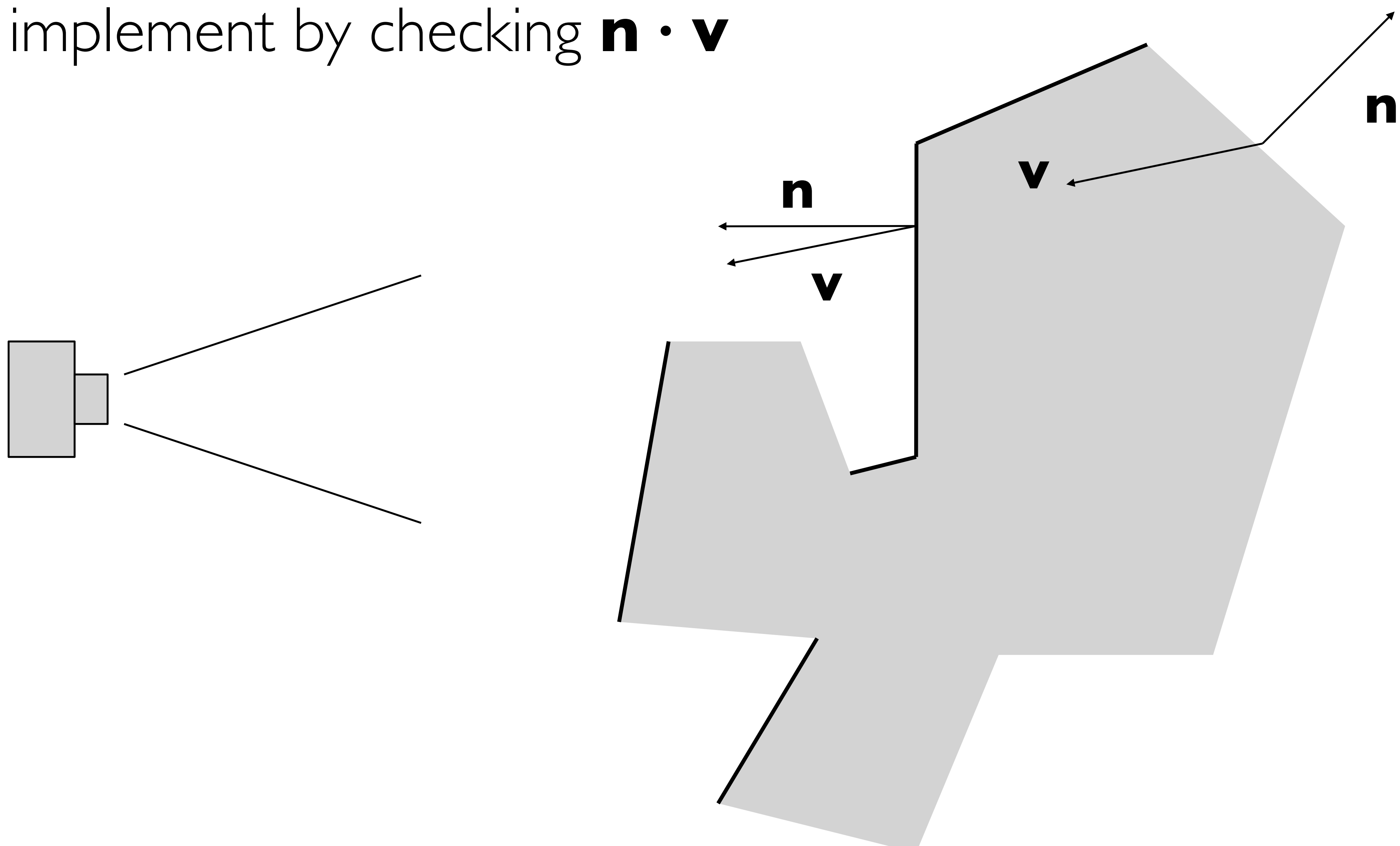
Back face culling

- **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

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

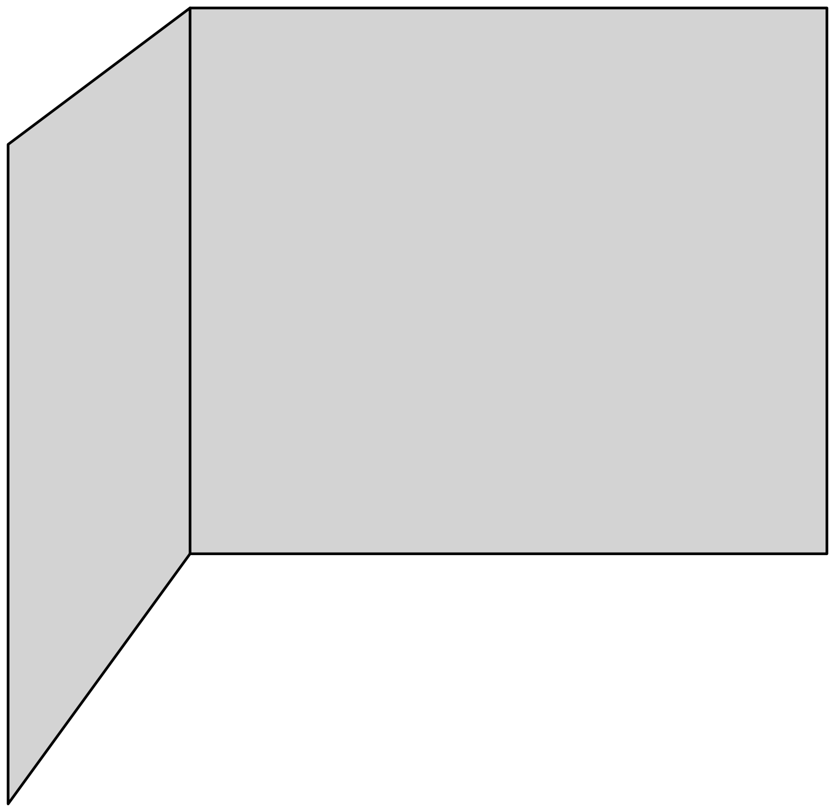


Painter's algorithm

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

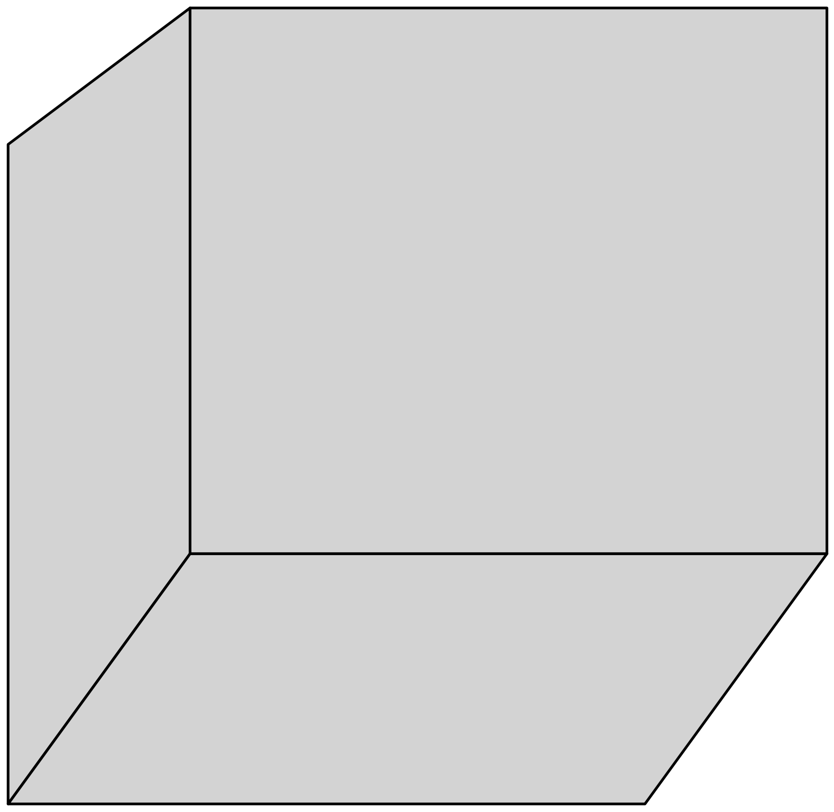
Painter's algorithm

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



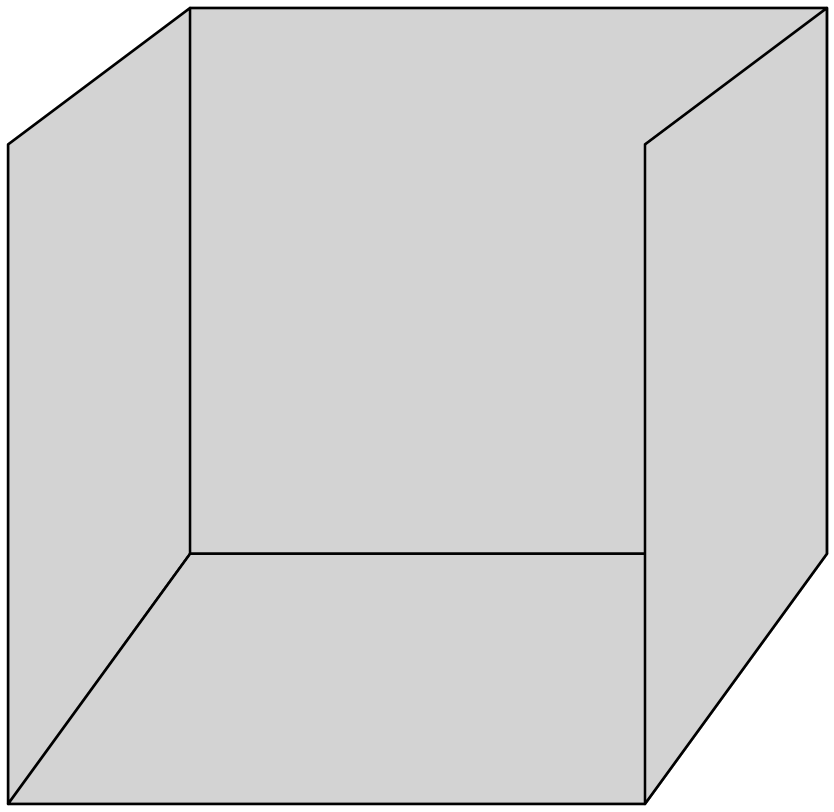
Painter's algorithm

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



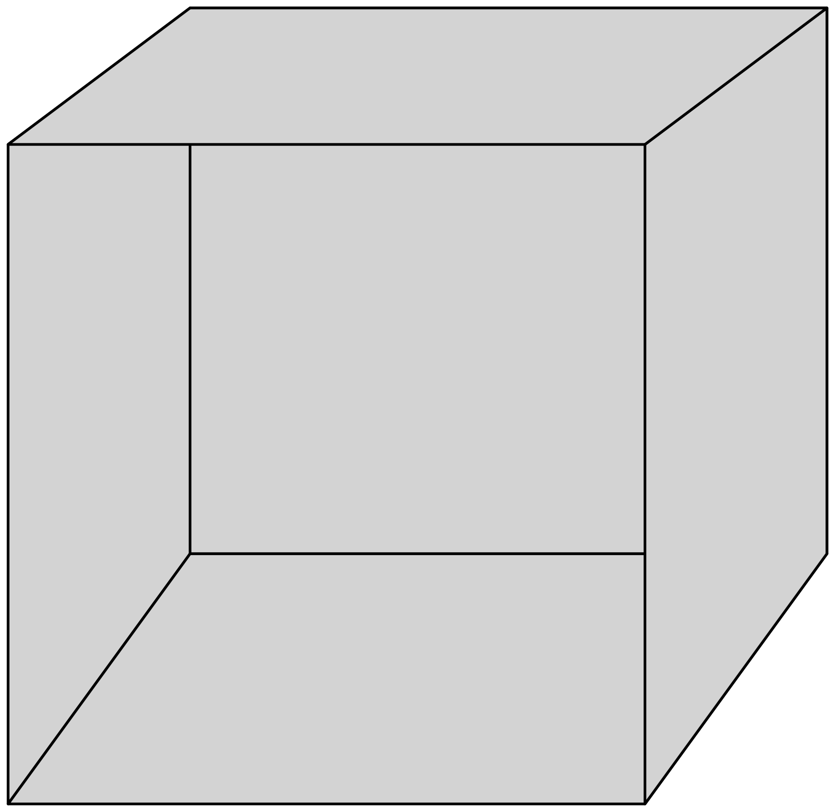
Painter's algorithm

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



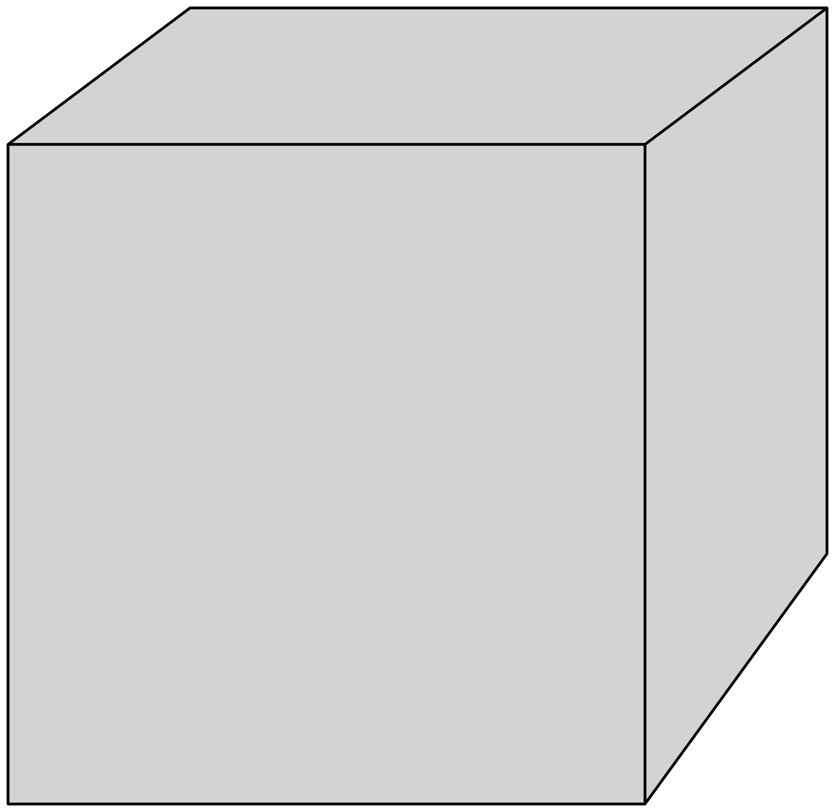
Painter's algorithm

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



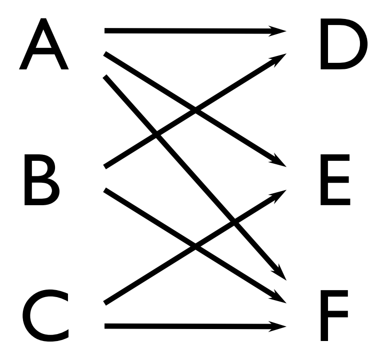
Painter's algorithm

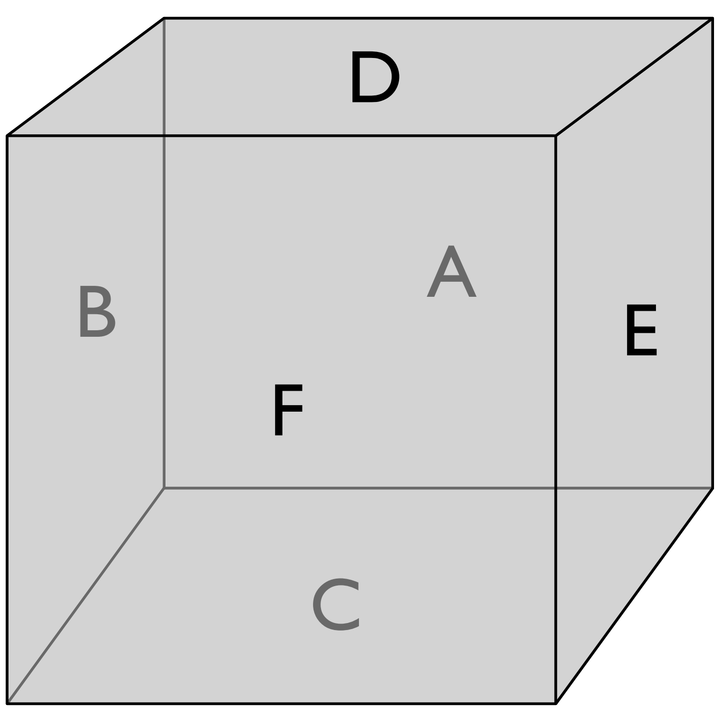
- **Simplest way to do hidden surfaces**
- **Draw from back to front, use overwriting in framebuffer**



Painter's algorithm

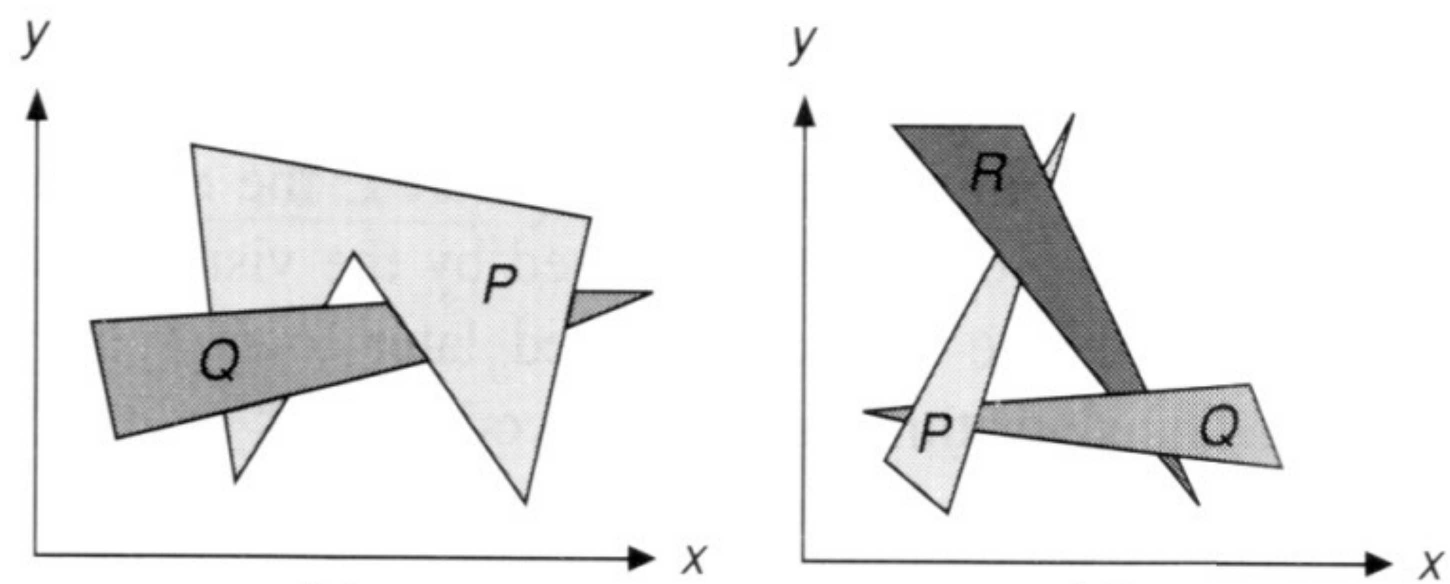
- **Amounts to a topological sort of the graph of occlusions**
  - that is, an edge from A to B means A sometimes occludes B
  - any sort is valid
    - ABCDEF
    - BADCFE
  - if there are cycles there is no sort





Painter's algorithm

- **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

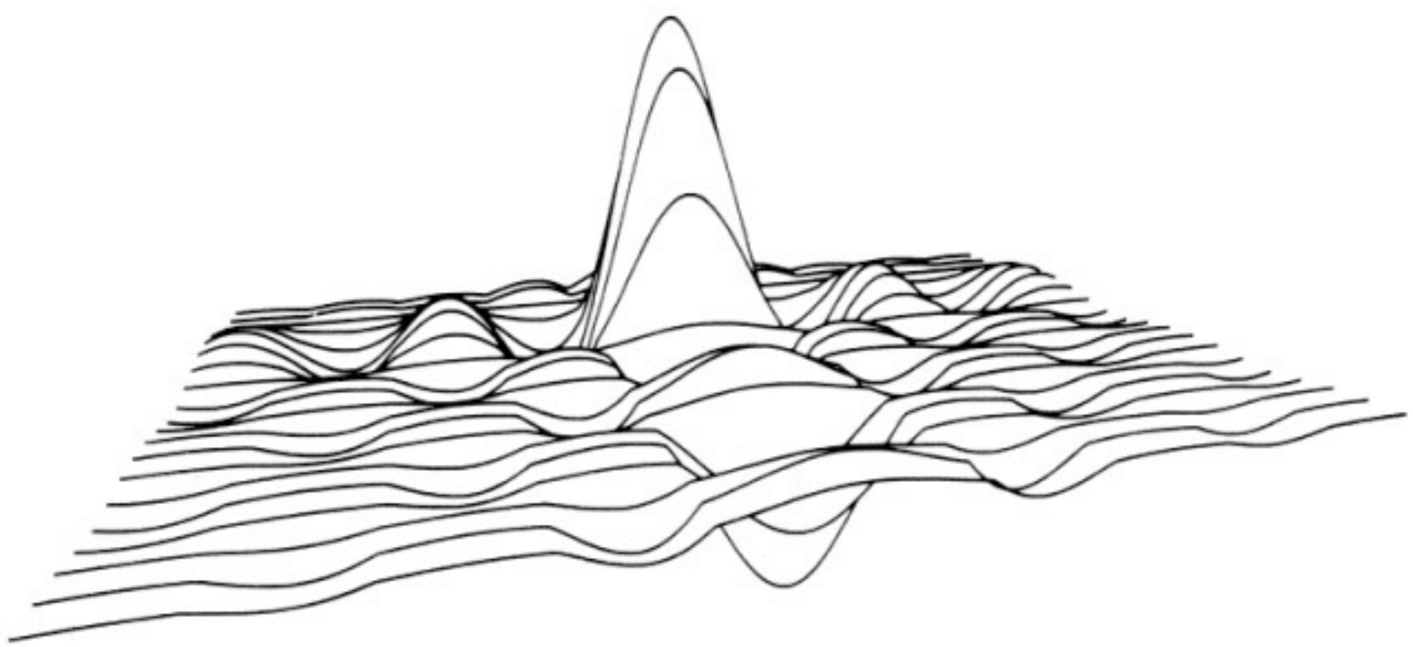


(b) (c)

[Foley et al.]

Painter's algorithm

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

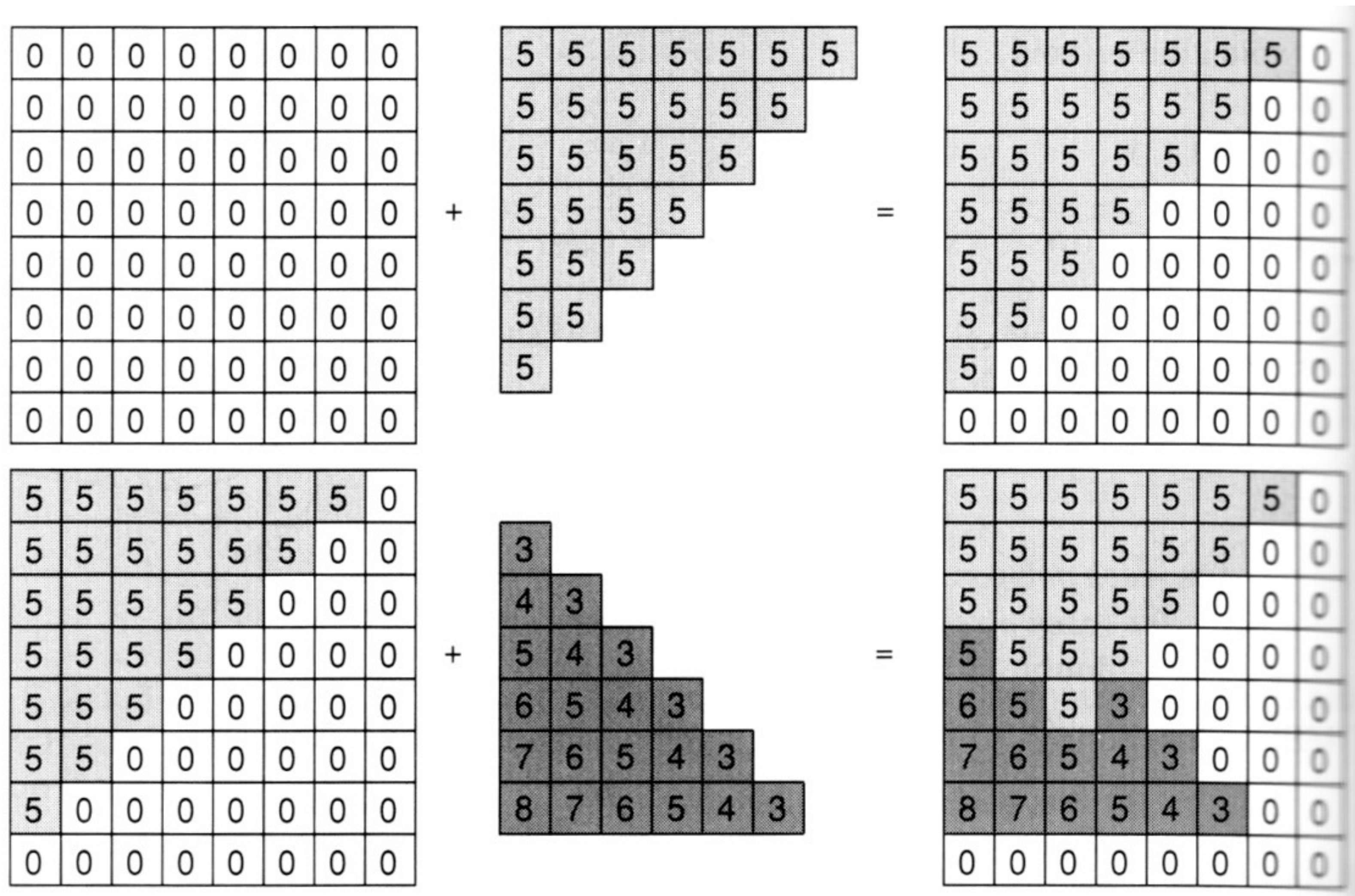


[Foley et al.]

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

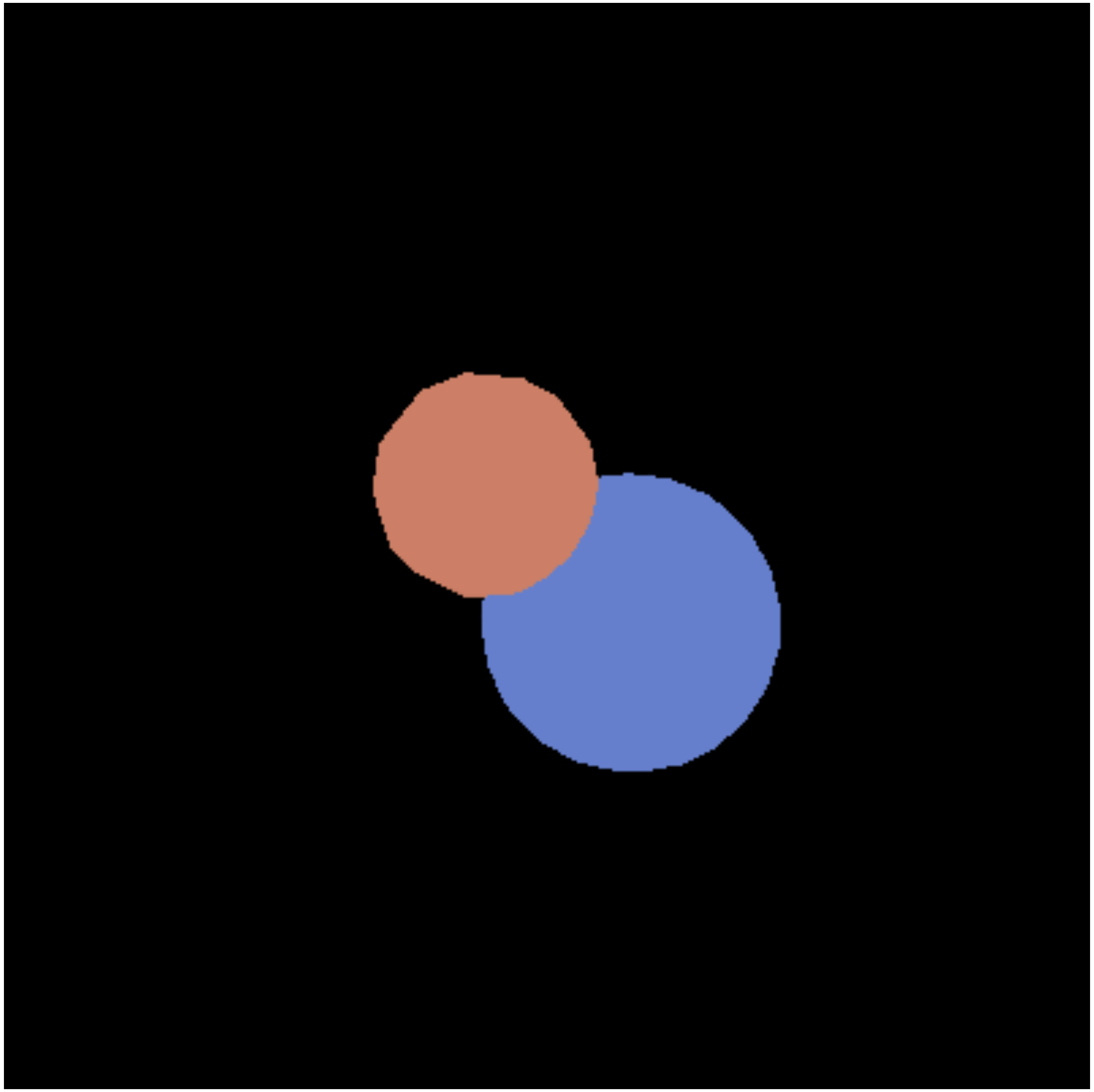
- **The precision is distributed between the near and far clipping planes**
  - this is why these planes have to exist
  - also why you can't always just set them to very small and very large distances
- **Traditionally use screen space depth (not eye-space) in *z* buffer**
  - screen space depth has to be interpolated linearly
    - and gives you more precision for near than far depths
  - eye-space depth has to be interpolated with perspective correction (a.k.a. *w* buffer)
    - and gives you uniform precision over [near, far]

Pipeline for minimal operation

Demo

- **Vertex stage (input: position / vtx)**
  - transform position (object to screen space)
- **Rasterizer**
  - nothing (extra) to interpolate
- **Fragment stage (output: color)**
  - write a fixed color to color planes
  - (color is a "uniform" quantity that is infrequently updated)

Result of minimal pipeline

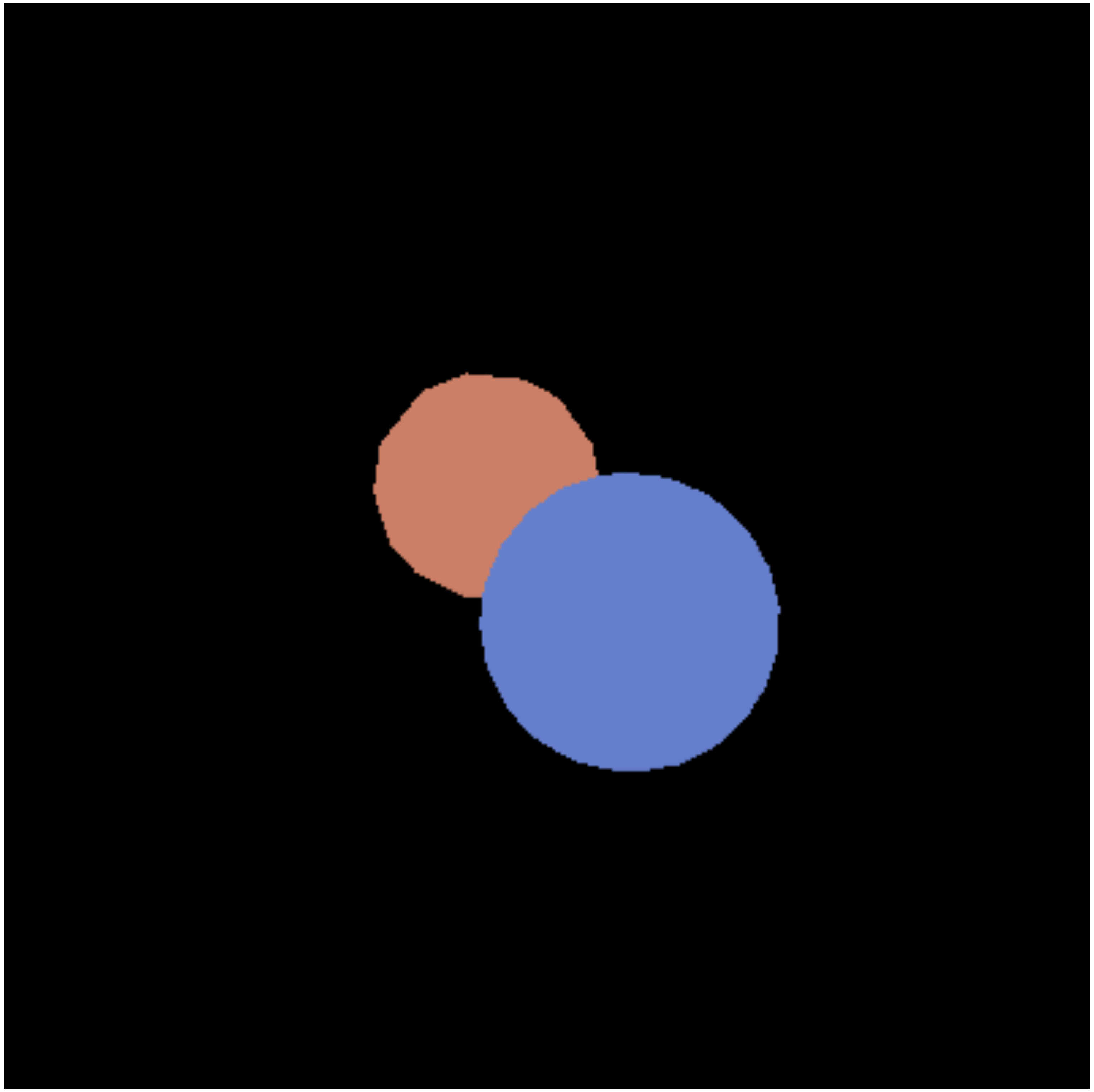


Pipeline for basic **z** buffer

Demo

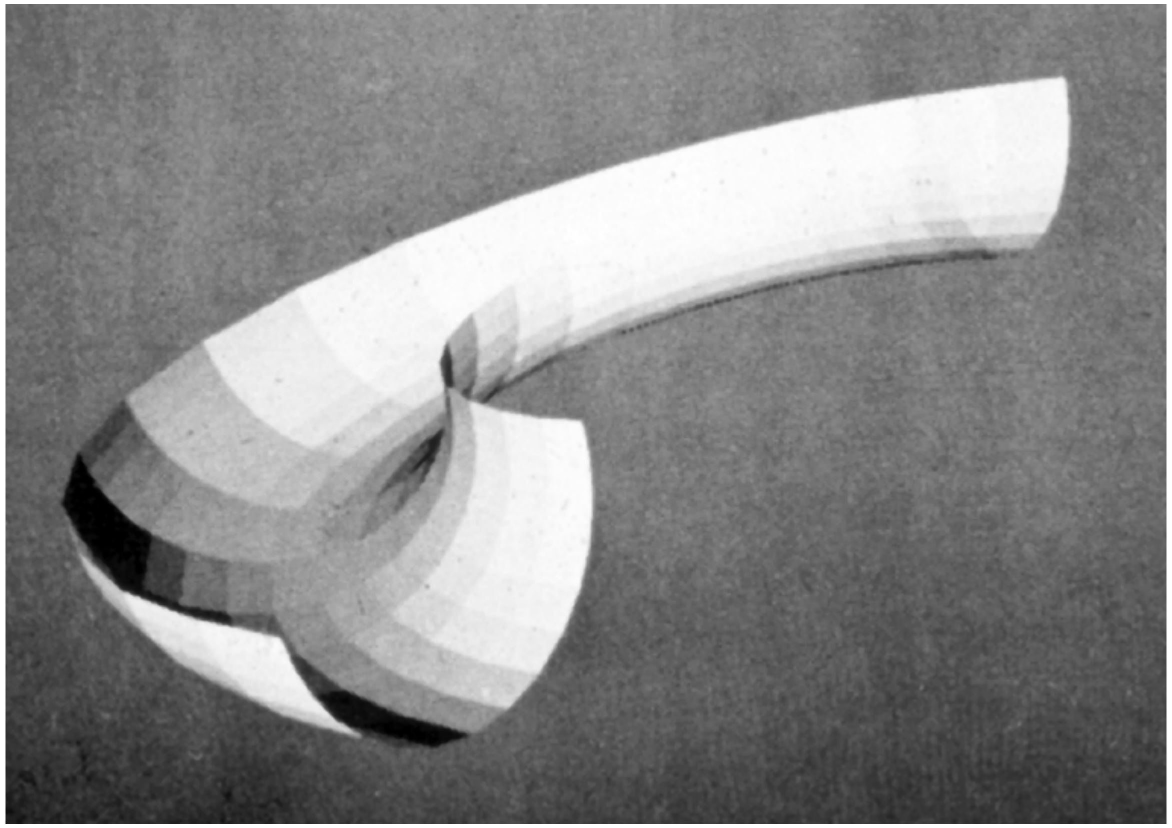
- **Vertex stage (input: position / vtx)**
  – transform position (object to screen space)
- **Rasterizer**
  – interpolated parameter: **z'** (screen **z**)
- **Fragment stage (output: color, *z'*)**
  – write fixed color to color planes only if interpolated **z'** < current **z'**

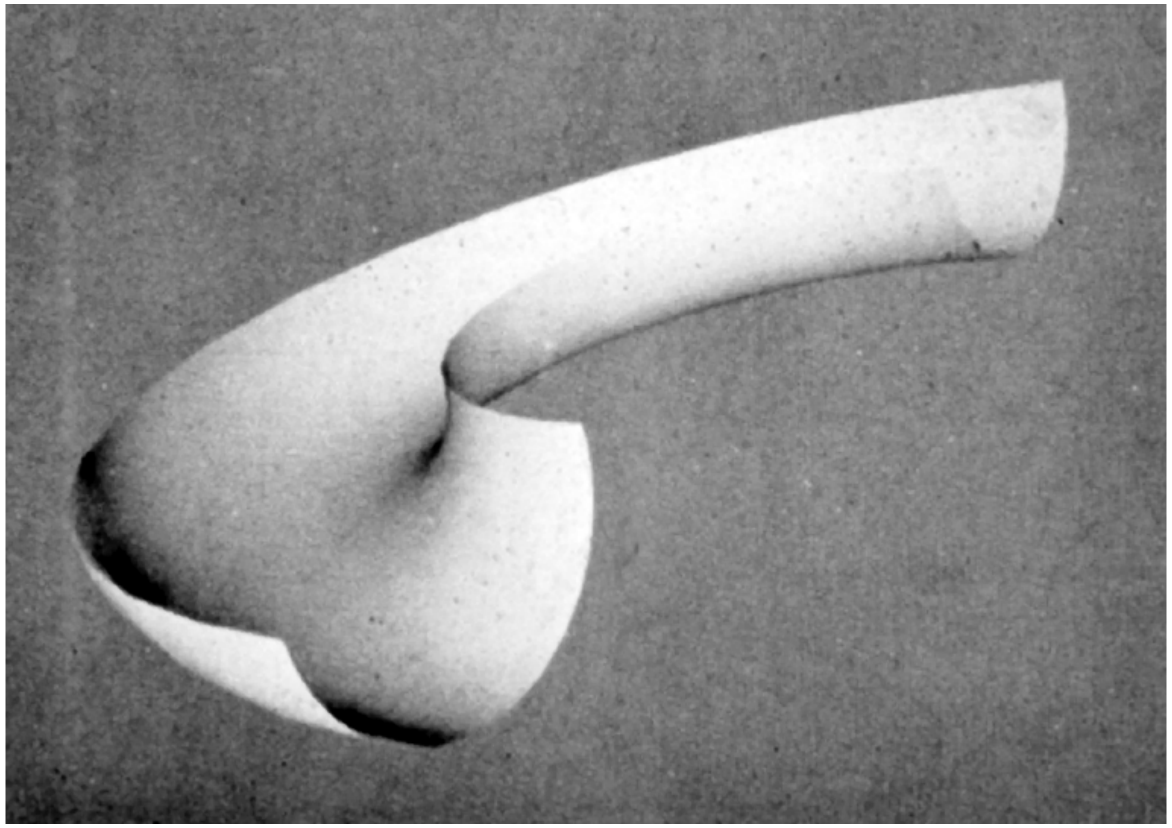
Result of *z*-buffer pipeline



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"



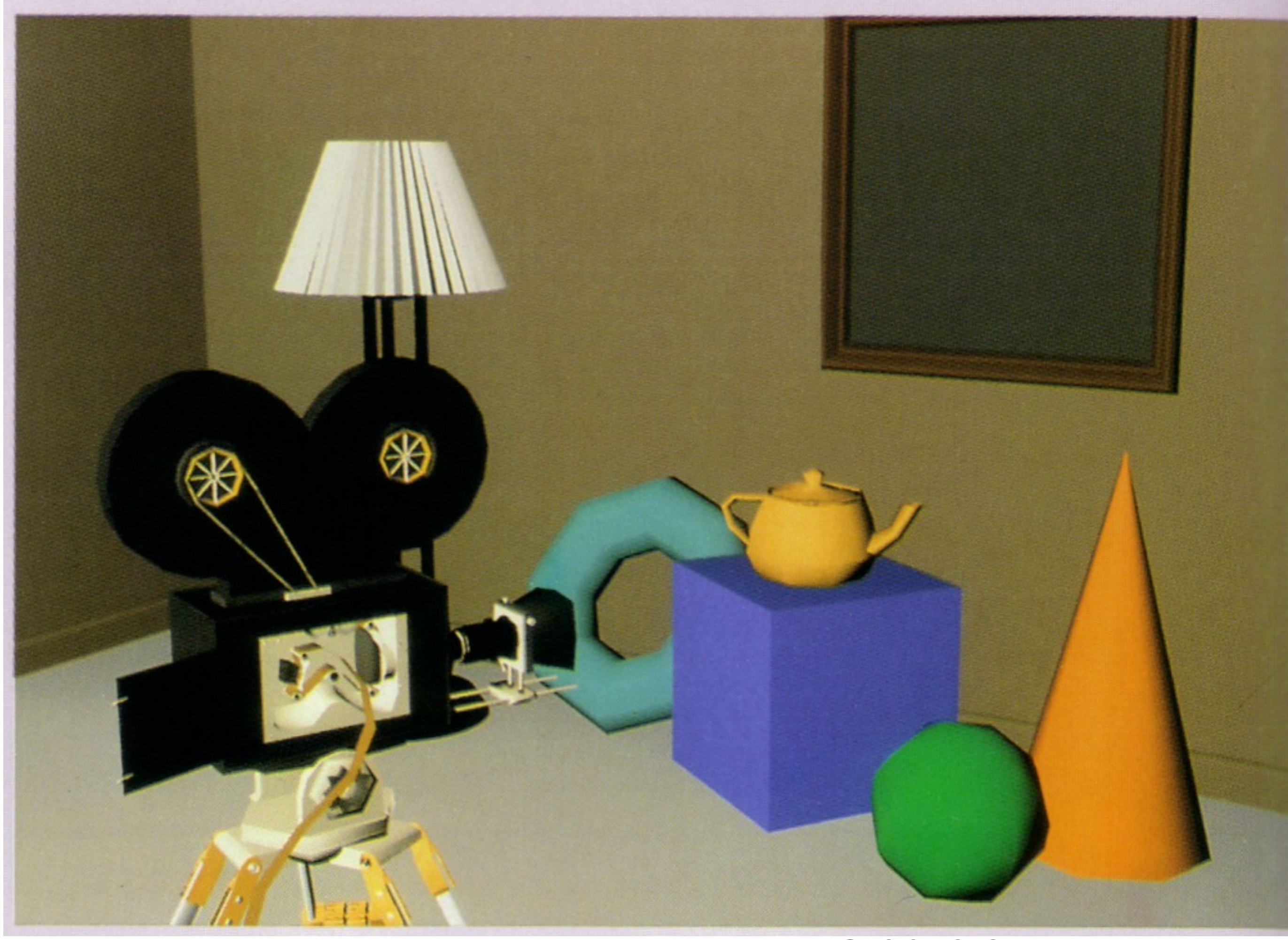


[Gouraud thesis]

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"

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.)



[Foley et al.]

Pipeline for Gouraud shading

Demo

- **Vertex stage (input: position and normal / vtx)**
  - transform position and normal (object to eye space)
  - compute shaded color per vertex (using fixed diffuse color)
  - 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'***

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

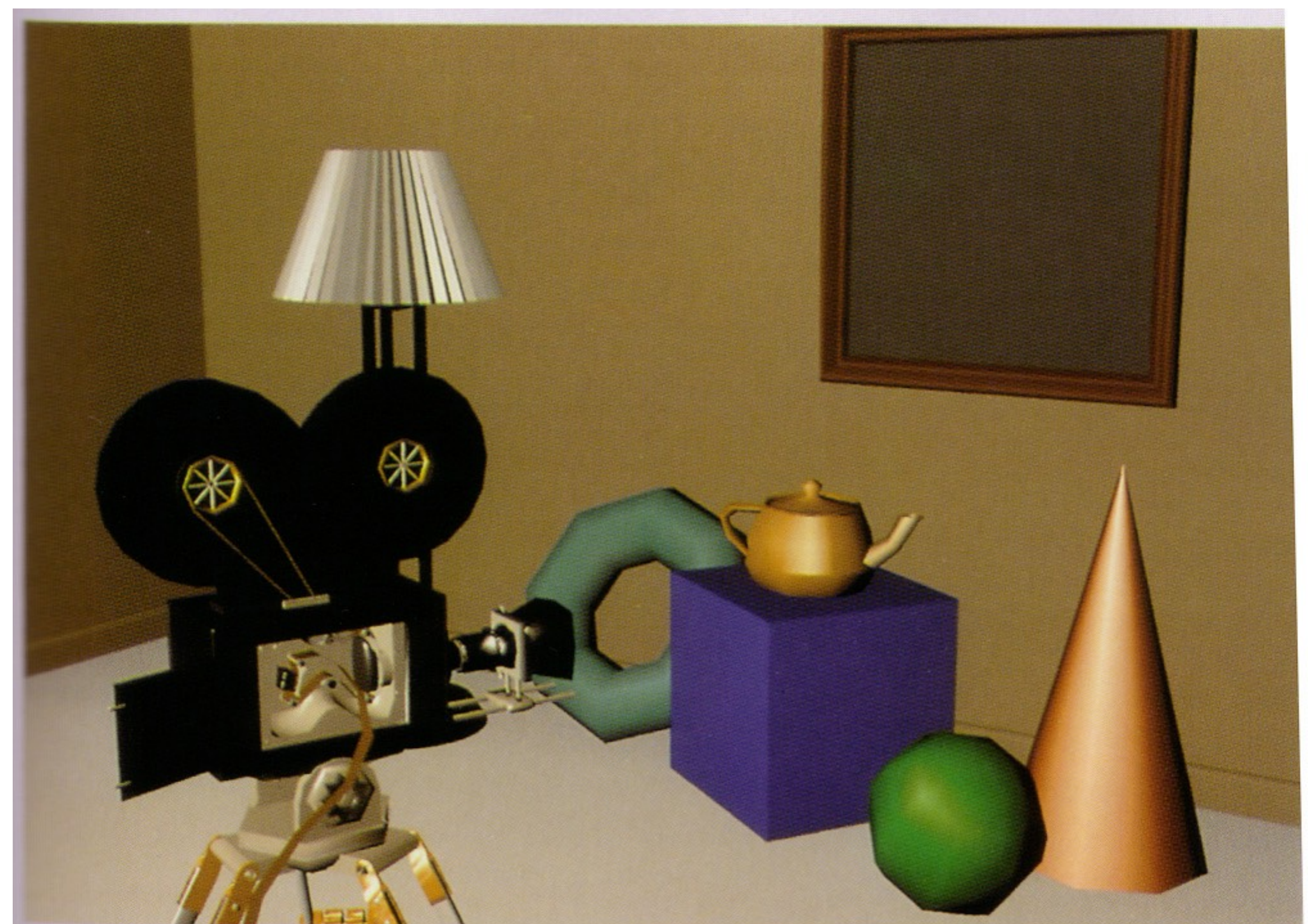
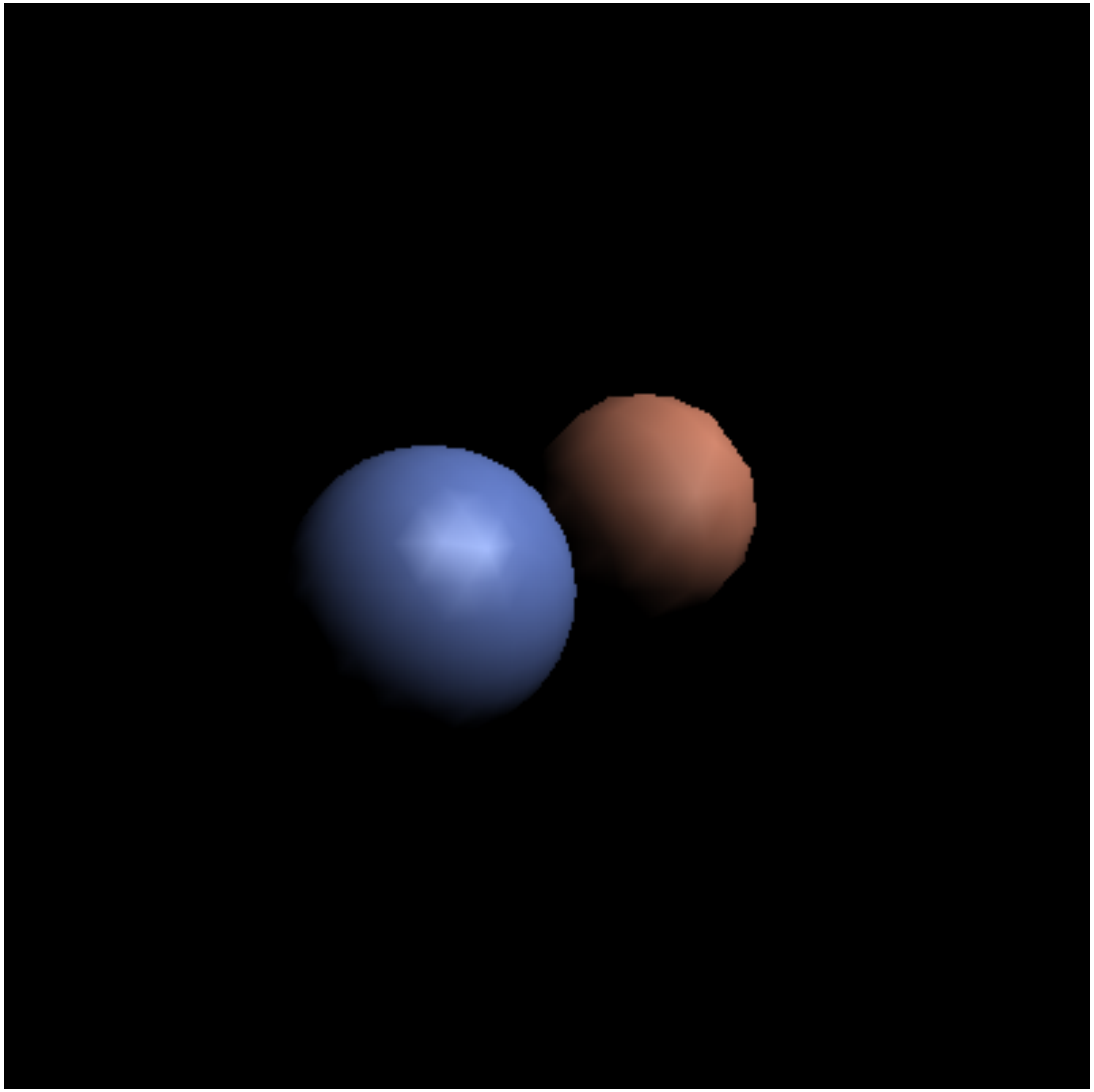


Plate II.31 *Shutterbug.* Gouraud shaded polygons with specular reflection (Sections 14.4.4 and 16.2.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

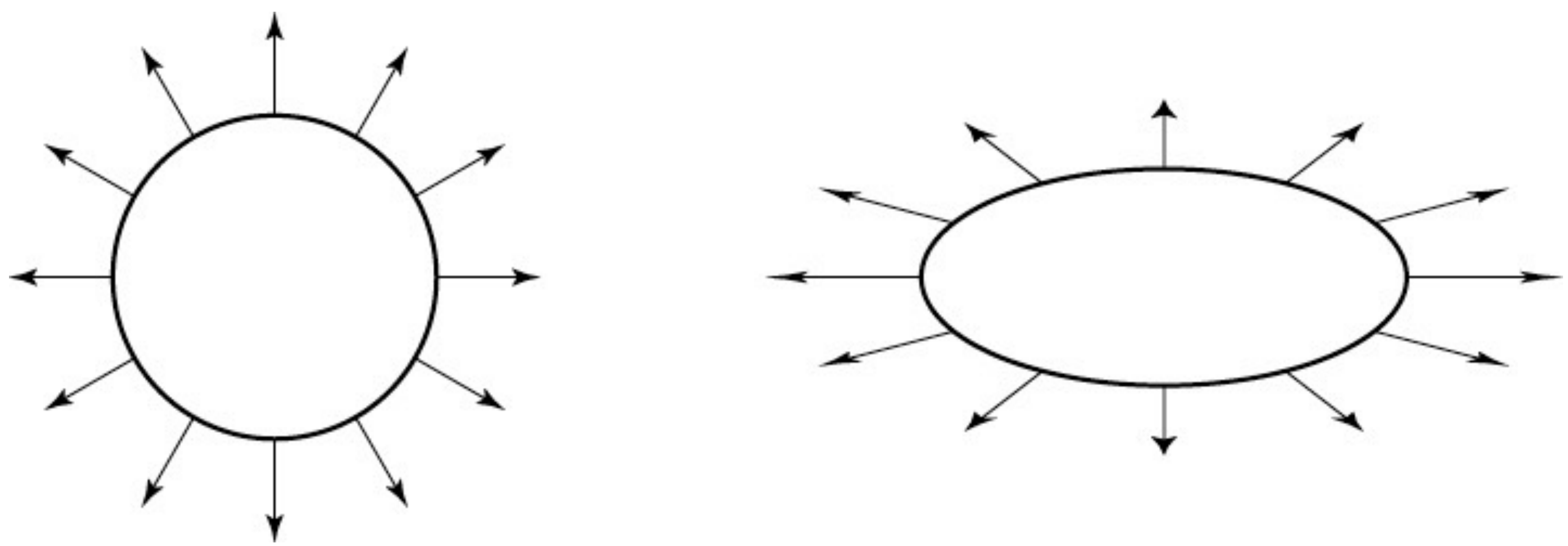
[Foley et al.]

Result of Gouraud shading pipeline



Transforming normal vectors

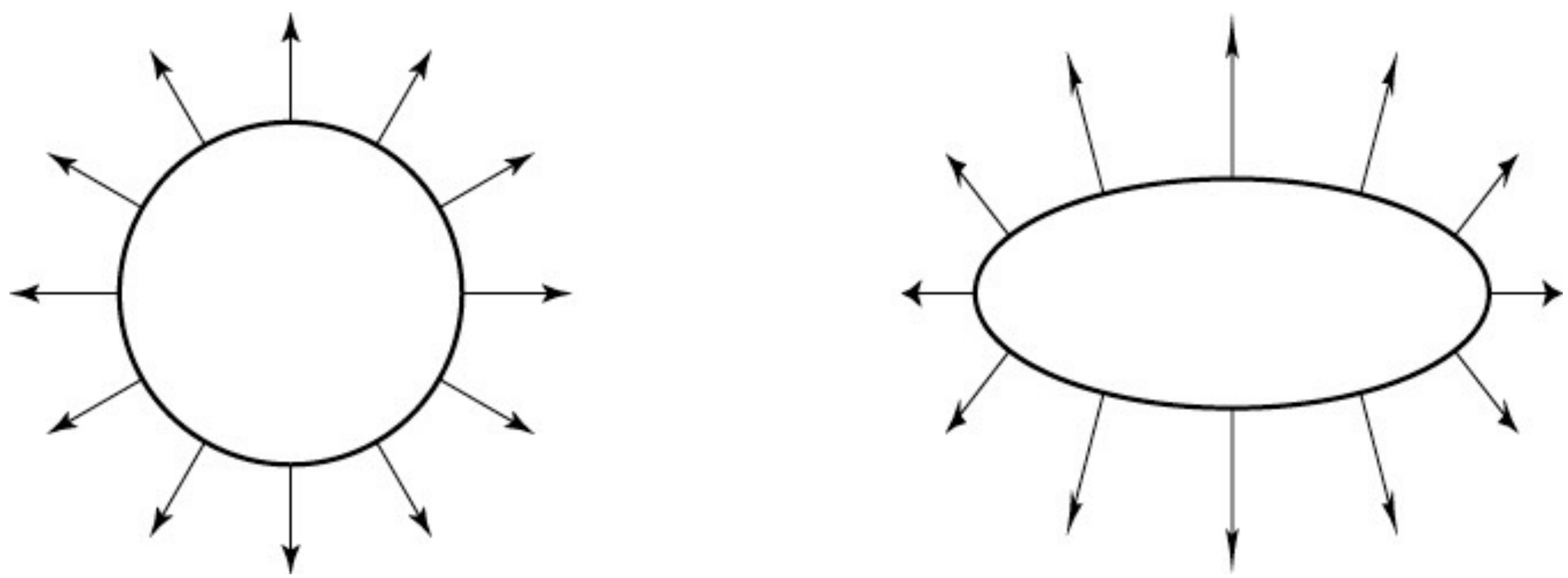
- **Transforming surface normals**
  - differences of points (and therefore tangents) transform OK
  - normals do not --> use inverse transpose matrix



have: $\mathbf{t} \cdot \mathbf{n} = \mathbf{t}^T \mathbf{n} = 0$
want: $M\mathbf{t} \cdot X\mathbf{n} = \mathbf{t}^T M^T X \mathbf{n} = 0$
so set $X = (M^T)^{-1}$
then: $M\mathbf{t} \cdot X\mathbf{n} = \mathbf{t}^T M^T (M^T)^{-1} \mathbf{n} = \mathbf{t}^T \mathbf{n} = 0$

Transforming normal vectors

- **Transforming surface normals**
  - differences of points (and therefore tangents) transform OK
  - normals do not --> use inverse transpose matrix

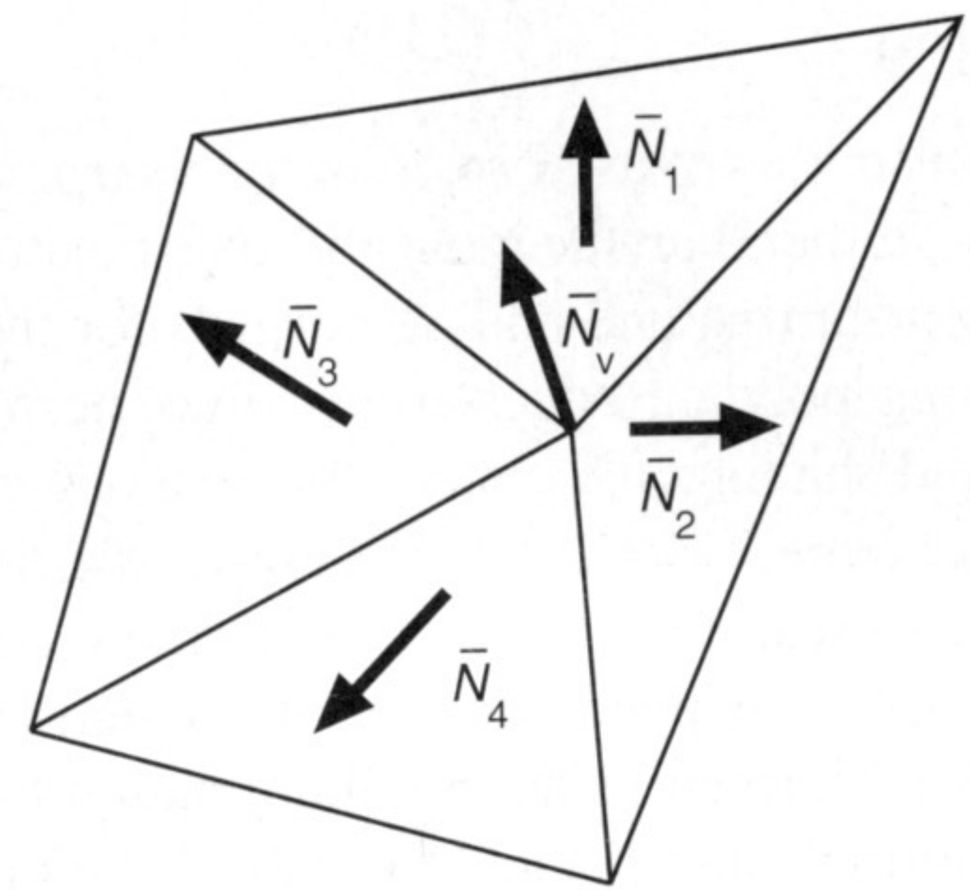


have: $\mathbf{t} \cdot \mathbf{n} = \mathbf{t}^T \mathbf{n} = 0$
want: $M\mathbf{t} \cdot X\mathbf{n} = \mathbf{t}^T M^T X \mathbf{n} = 0$
so set $X = (M^T)^{-1}$
then: $M\mathbf{t} \cdot X\mathbf{n} = \mathbf{t}^T M^T (M^T)^{-1} \mathbf{n} = \mathbf{t}^T \mathbf{n} = 0$

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

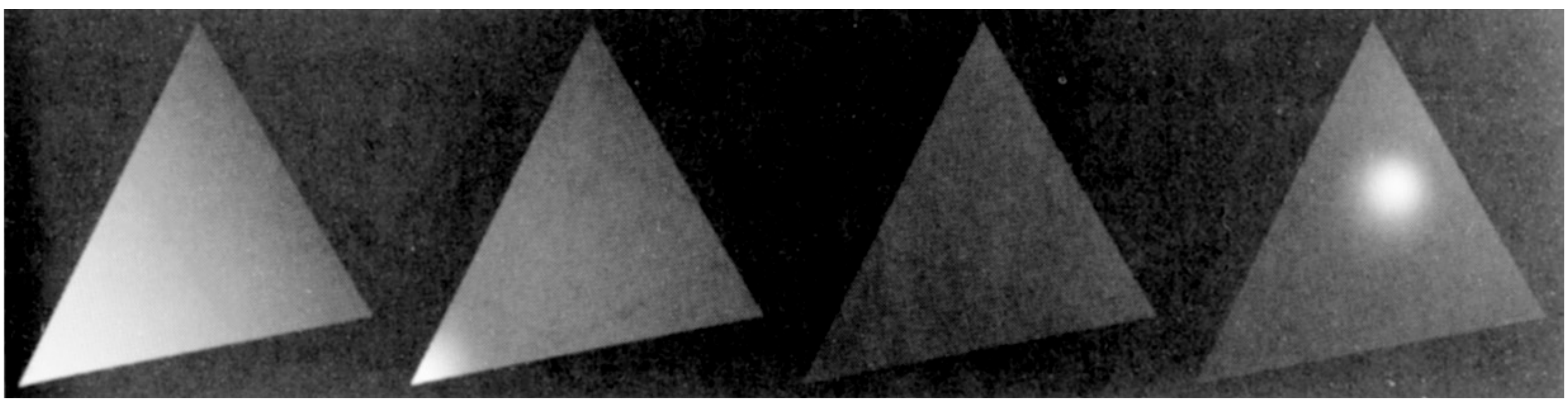
$$N_v = \frac{\sum_i N_i}{\| \sum_i N_i \|}$$



[Foley et al.]

Per-pixel (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



Per-pixel (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.)

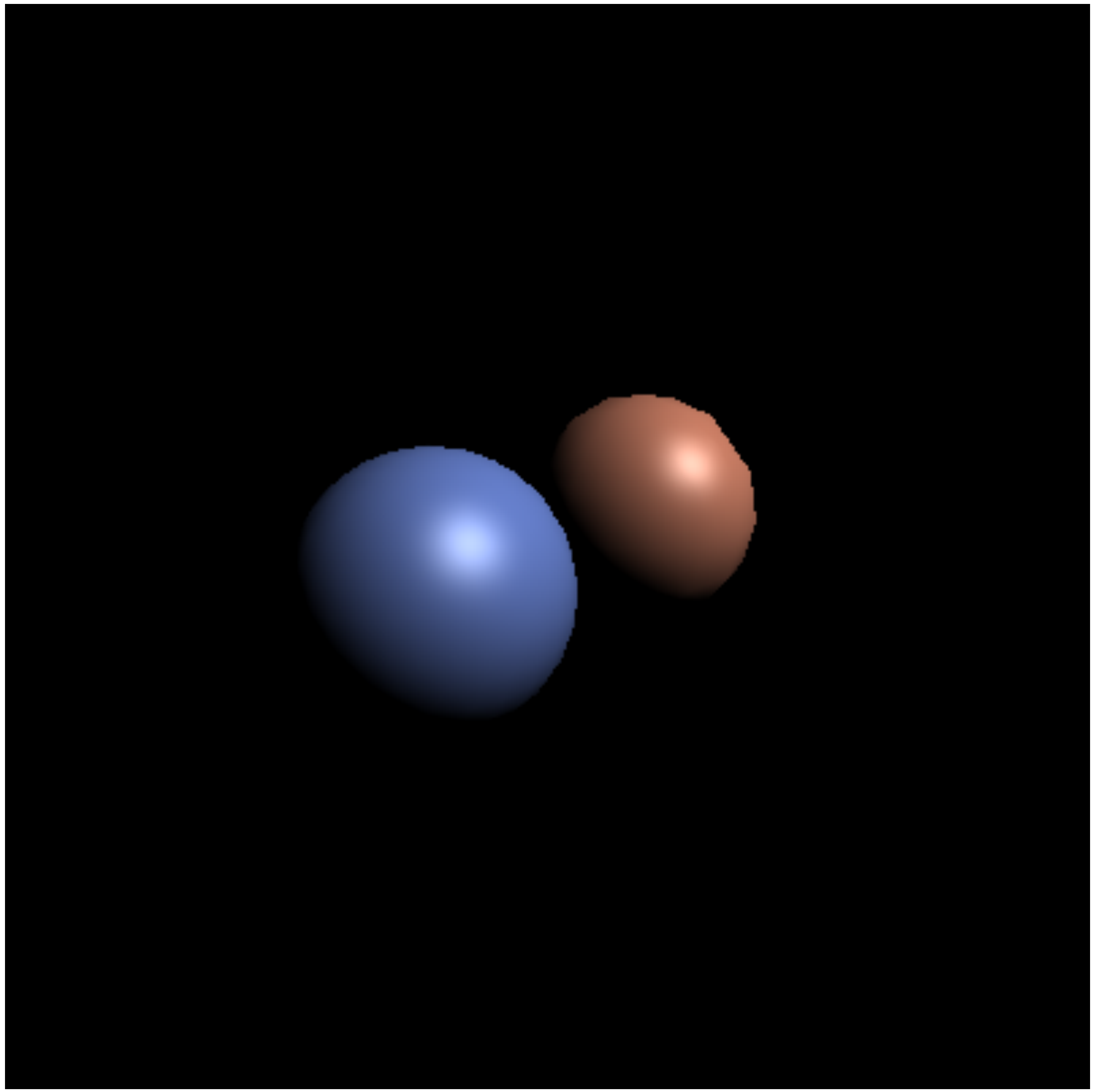
[Foley et al.]

Pipeline for per-pixel shading

Demo

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

Result of per-pixel shading pipeline



Programming hardware pipelines

- **Modern hardware graphics pipelines are flexible**
  - programmer defines exactly what happens at each stage
  - do this by writing *shader programs* in domain-specific languages called *shading languages*
  - rasterization is fixed-function, as are some other operations (depth test, many data conversions, …)
- **One example: OpenGL and GLSL (GL Shading Language)**
  - several types of shaders process primitives and vertices; most basic is the *vertex program*
  - after rasterization, fragments are processed by a *fragment program*

GLSL Shaders