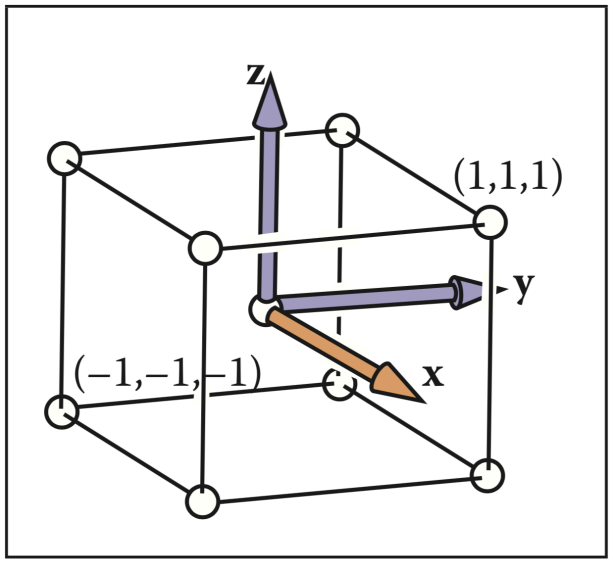
CSC 433/533
Computer Graphics

Joshua Levine
josh@email.arizona.edu

Lecture 25
Graphics Pipeline 2

Nov. 19, 2020

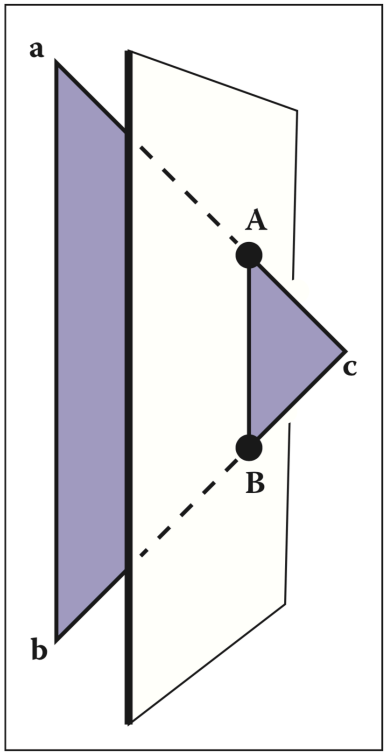
Clipping Planes



- Because we have not done the homogeneous divide, the view volume is actually a four dimensional object
- Math is a little bit tedious, but these equations use $l = b = n =$ -1 and $r = t = f = 1$
- Each equation defines a side of the box in the form $f(\mathbf{p}) = (\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0$, or equivalently: $f(\mathbf{p}) = \mathbf{n} \cdot \mathbf{p} + \mathbf{D} = 0$

$$
\begin{aligned}
-x + lw &= 0 \\
x - rw &= 0 \\
-y + bw &= 0 \\
y - tw &= 0 \\
-z + nw &= 0 \\
z - fw &= 0
\end{aligned}
$$

Clipping a Triangle Against a Plane



- Basic idea: Check if any of three line segments cross the plane.

- How? Solve for intersection points:

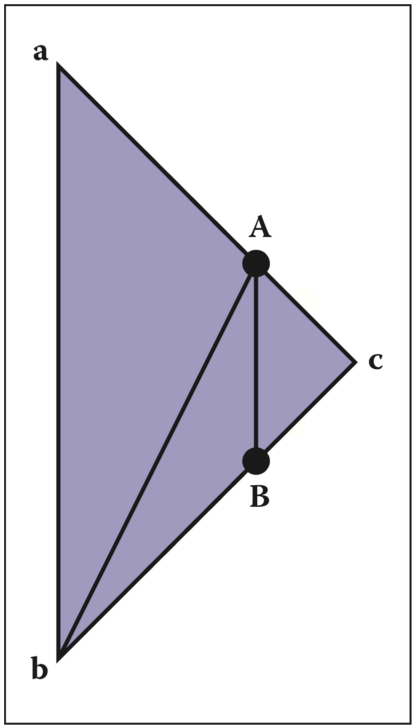
$$f(\mathbf{p}) = \mathbf{n} \cdot \mathbf{p} + \mathbf{D} = 0$$

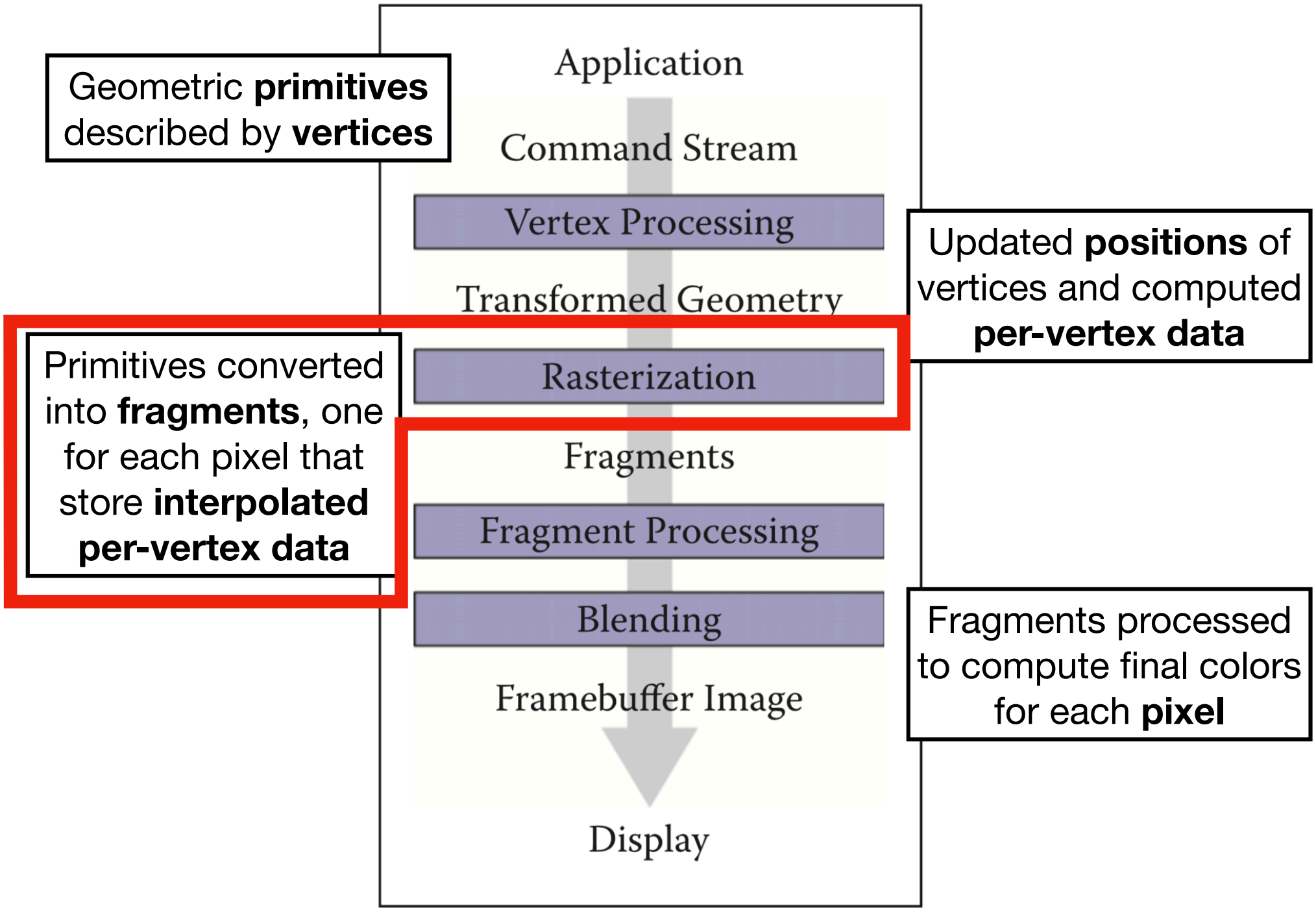
- By plugging in:

$$\mathbf{p} = \mathbf{a} + t(\mathbf{b} - \mathbf{a})$$

$$t = \frac{\mathbf{n} \cdot \mathbf{a} + D}{\mathbf{n} \cdot (\mathbf{a} - \mathbf{b})}$$

Note: $\mathbf{a}$, $\mathbf{b}$ are both 4D homogeneous points and $\mathbf{n}$ is a 4D vector




Geometric **primitives** described by **vertices**
Application
Command Stream
Vertex Processing
Transformed Geometry
Updated **positions** of vertices and computed **per-vertex data**
Primitives converted into **fragments**, one for each pixel that store **interpolated per-vertex data**
Rasterization
Fragments
Fragment Processing
Blending
Fragments processed to compute final colors for each **pixel**
Framebuffer Image
Display

Operations Before and After Rasterization

- Vertex Processing:
  - Role: Prepare the data necessary for rasterization.
- Fragment Processing:
  - Role: Combine the fragments and compute additional information to determine the final color for each pixel.

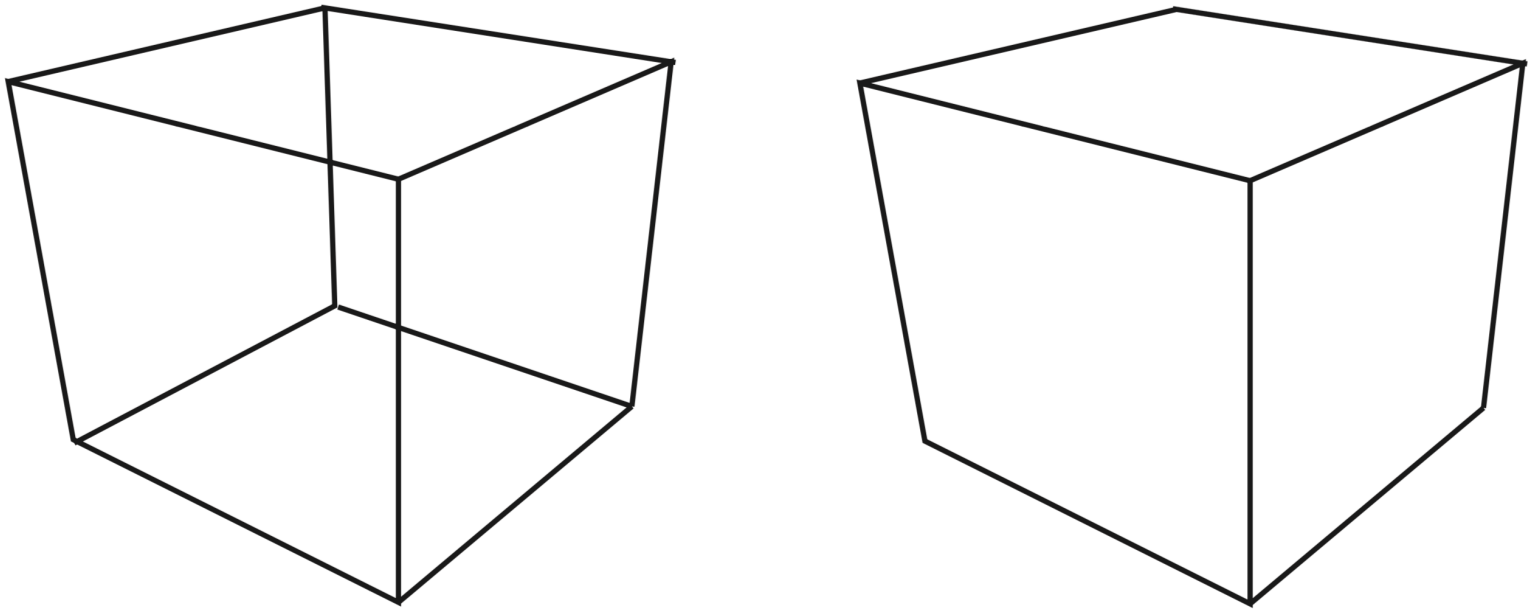
A Minimal Pipeline for Raster Graphics

- Vertex Processing
  - Transform vertex positions from object to screen space
  - Set color for each vertex based on primitive.
- Rasterizer
  - Enumerate fragments for each primitive and set their color.
- Fragment Processing
  - Write fragment colors to the appropriate pixel

Resolving Issues with Depth

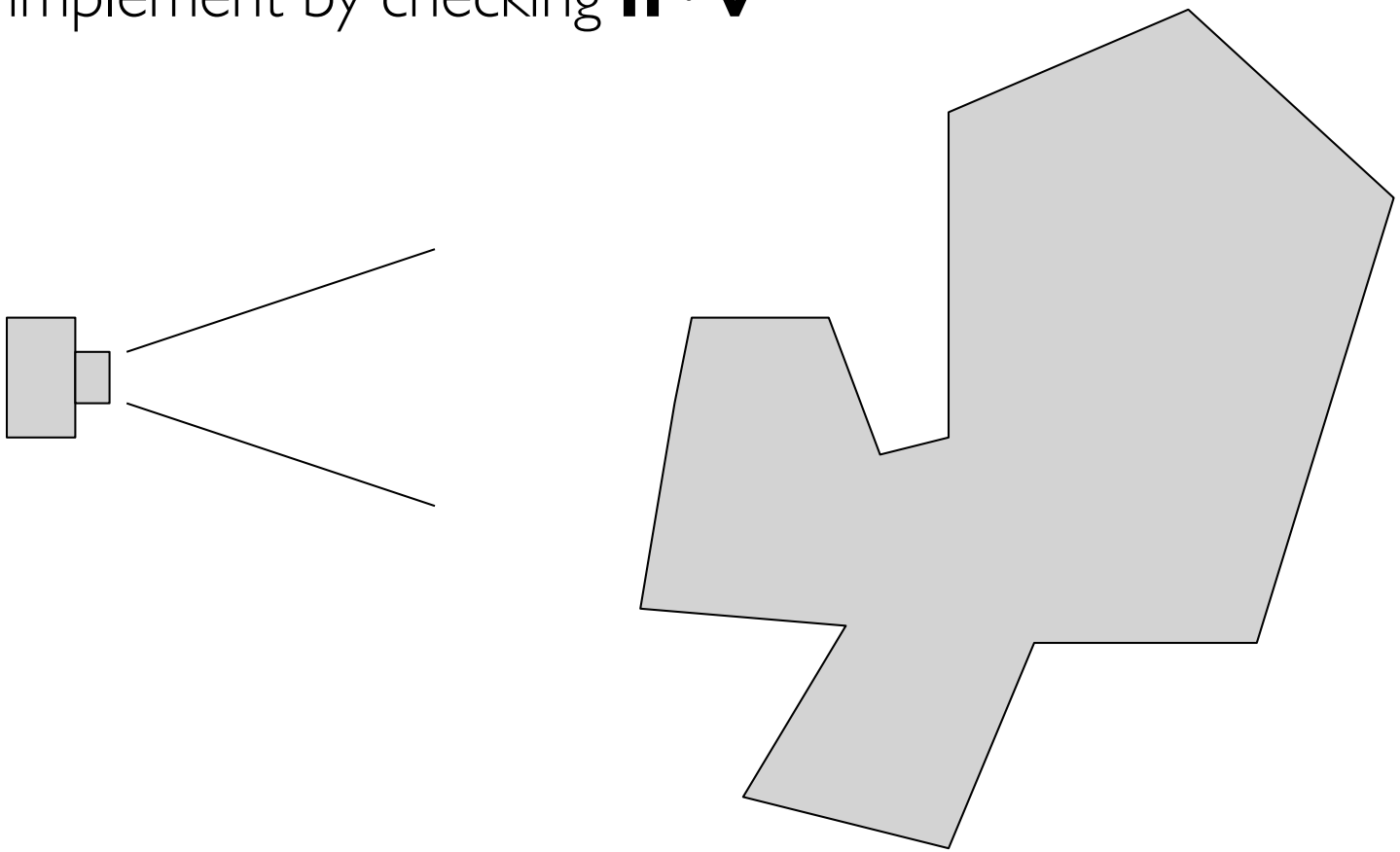
Hidden Surface Removal

- Perspective projection provides one important cue for 3D relationships
- Depth / Occlusion is equally important



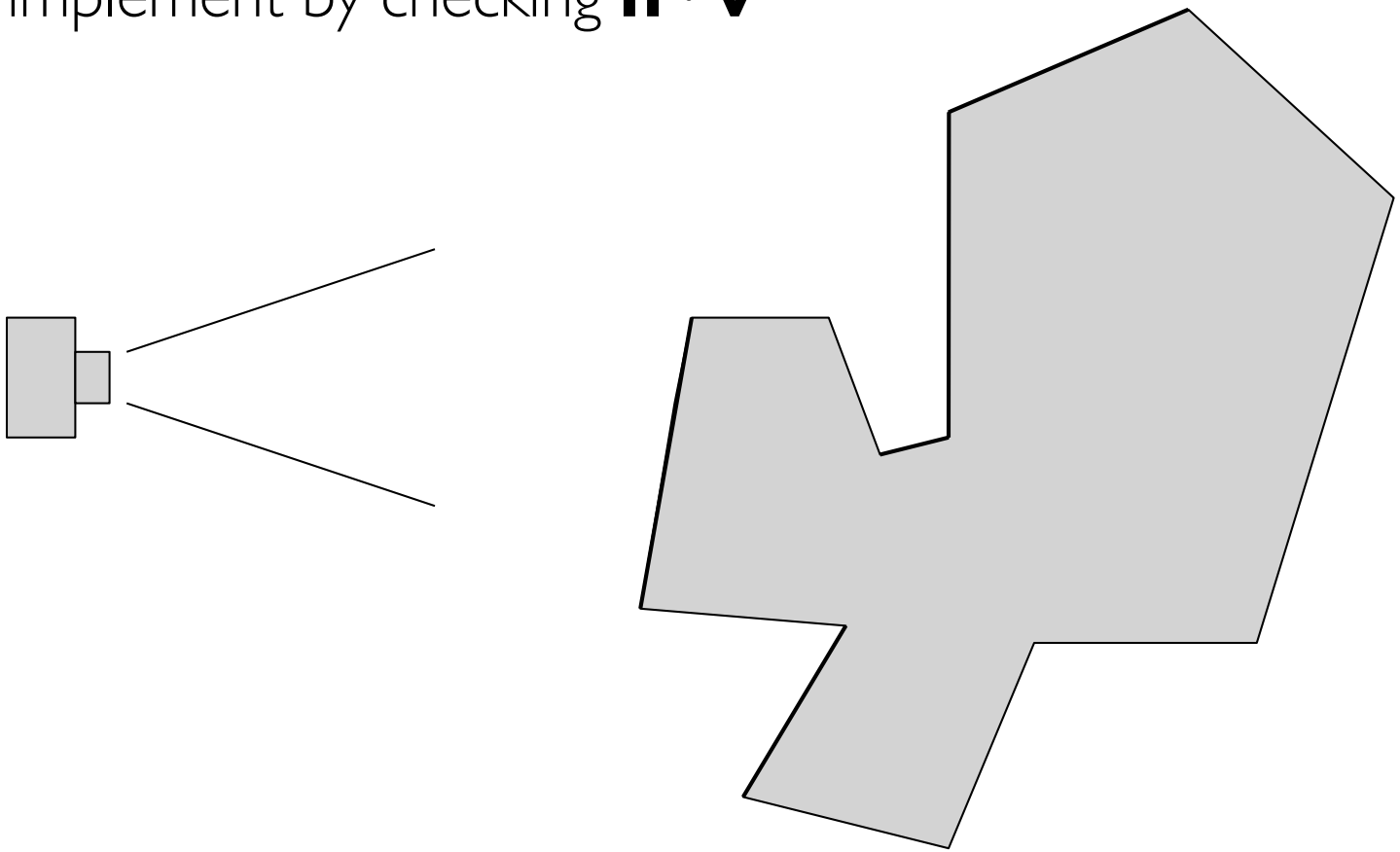
Back face culling

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



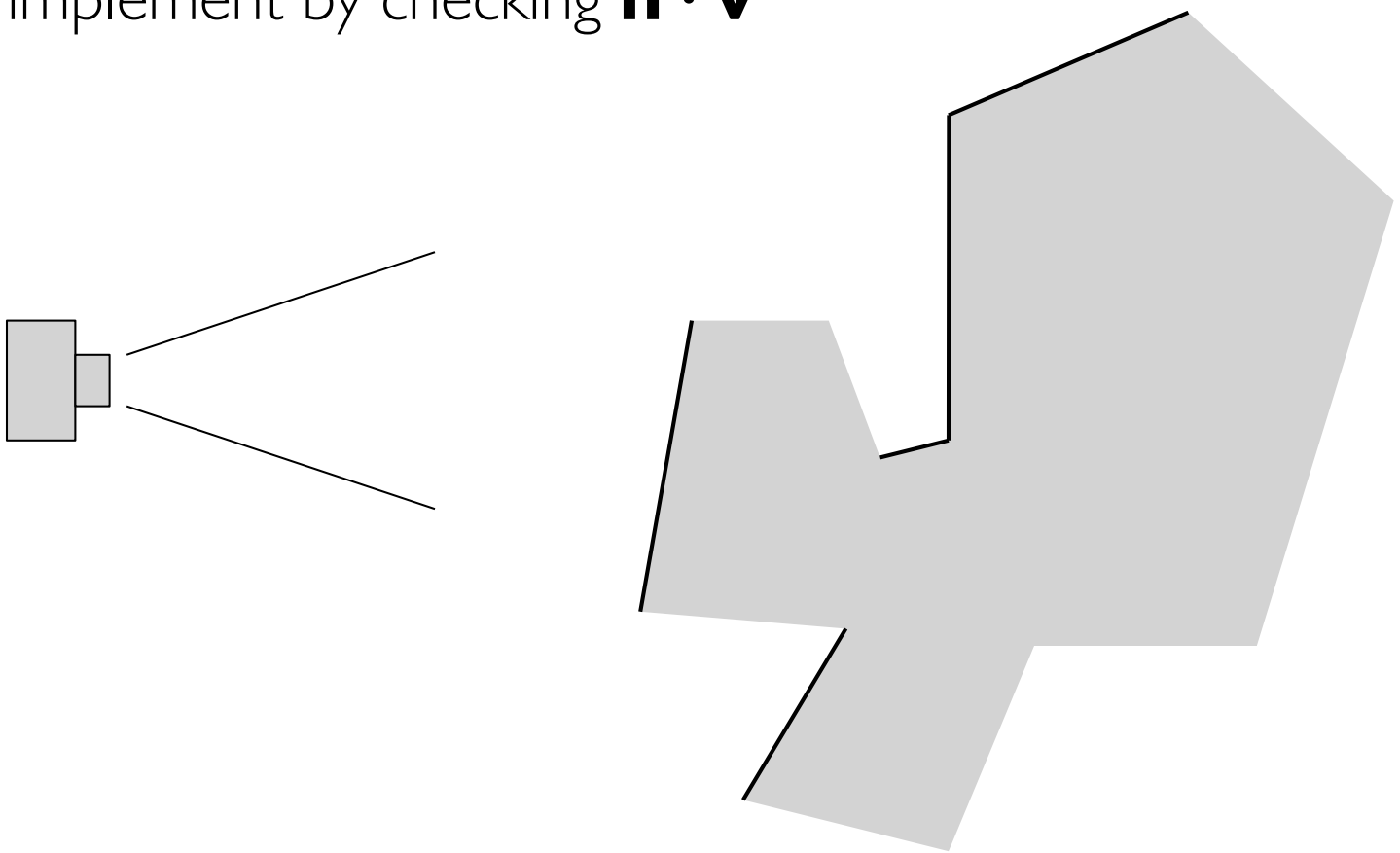
Back face culling

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



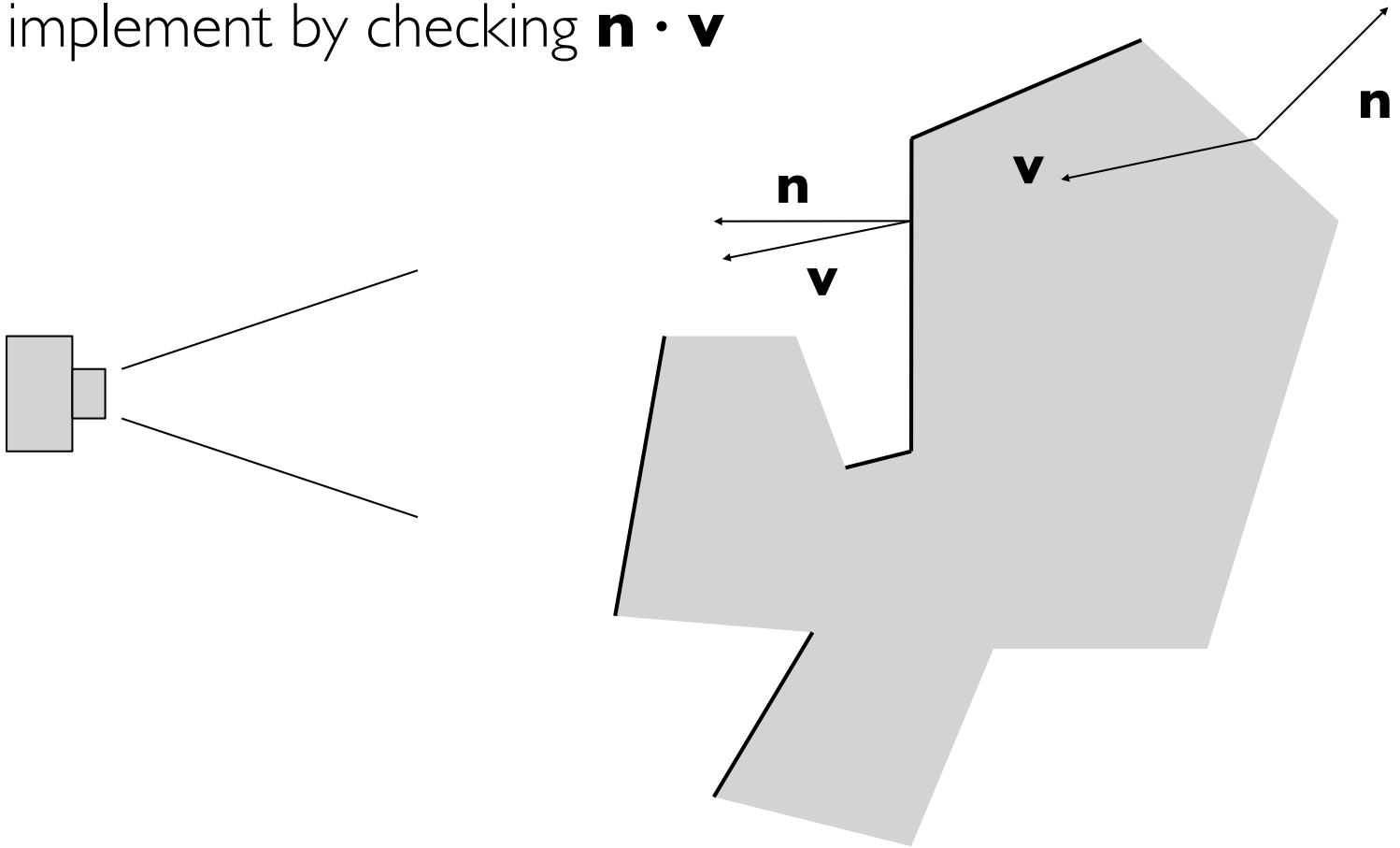
Back face culling

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



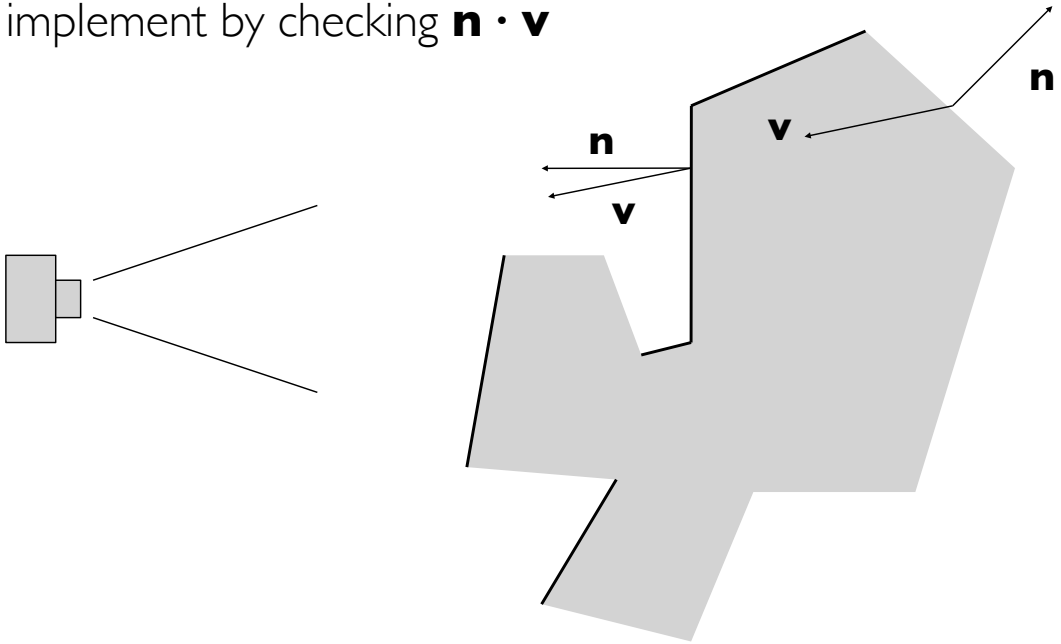
Back face culling

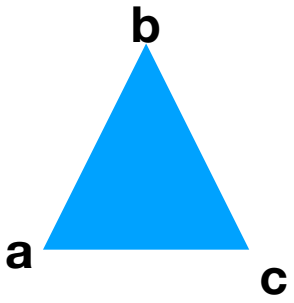
- **For closed shapes you will never see the inside**
  - therefore only draw surfaces that face the camera
  - implement by checking **n** · **v**



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**

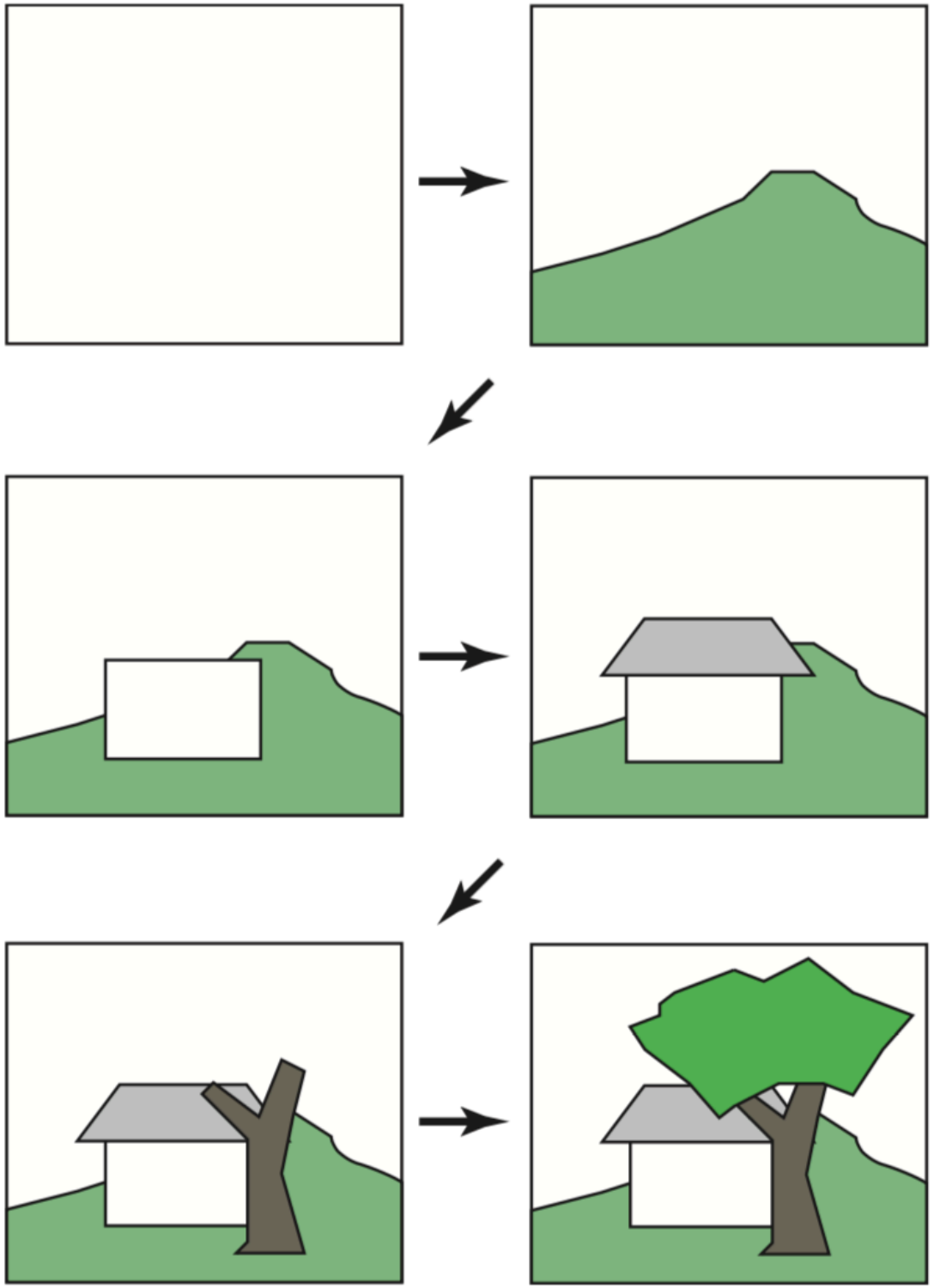




- This is one of the reasons that in the triangles meshes,
- The vertices of each triangle were given in CCW when watching from the OUTSIDE.
- This enables calculating $\overrightarrow{n} = (c - a) \times (b - a)$ points outwards

Painter's Algorithm

- Simple way to do hidden surfaces
- Draw from back-to-front, overwriting directly on the image

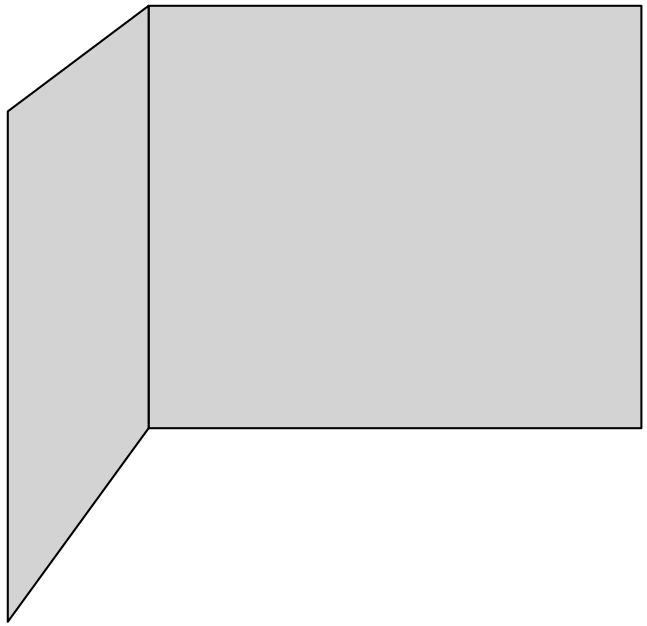


Painter's Algorithm

- Draw one primitive at a time.

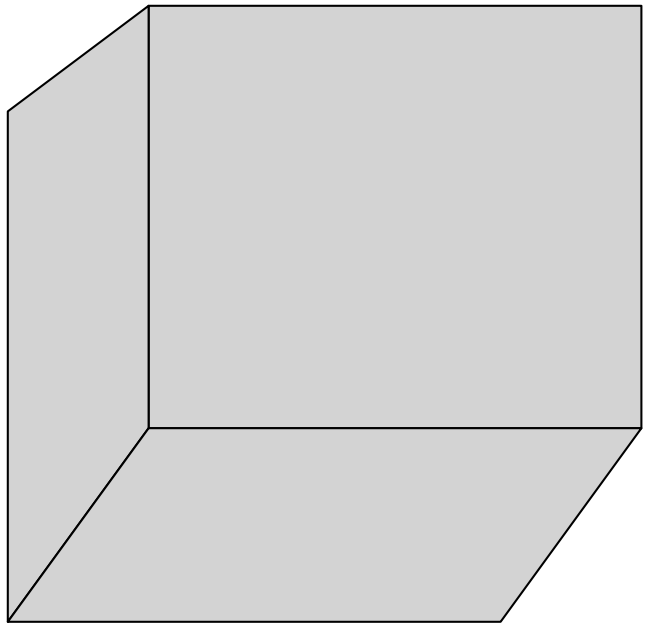
Painter's Algorithm

- Draw one primitive at a time.



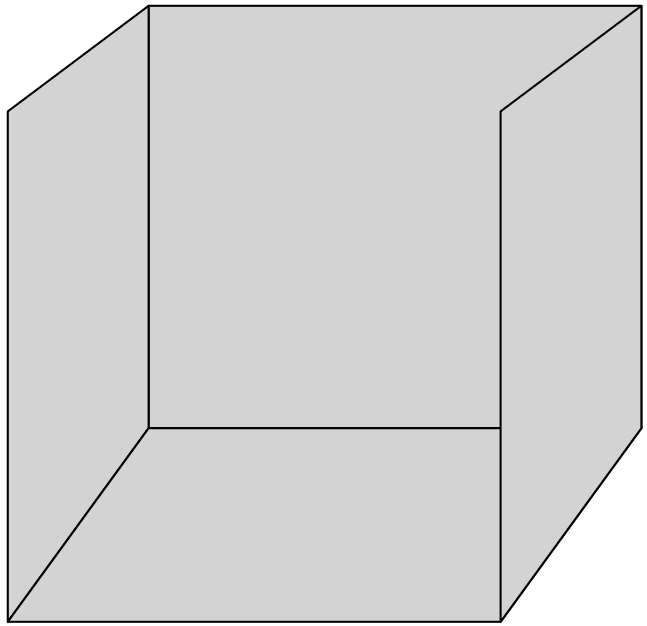
Painter's Algorithm

- Draw one primitive at a time.



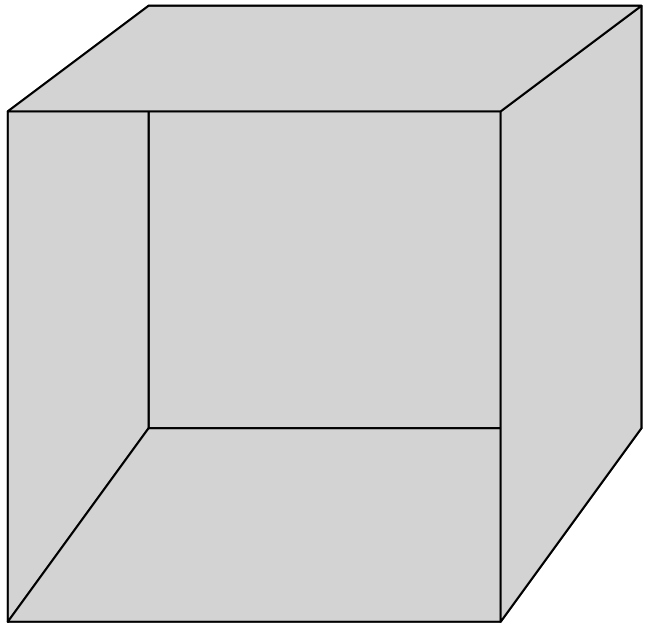
Painter's Algorithm

- Draw one primitive at a time.



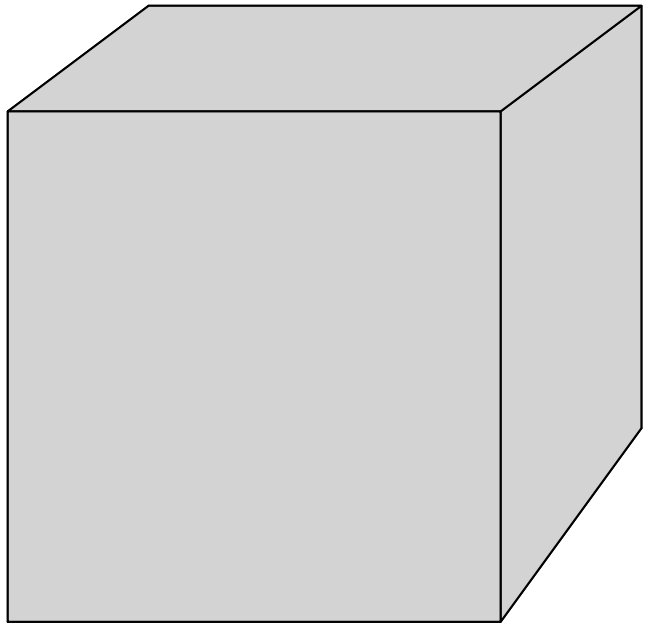
Painter's Algorithm

- Draw one primitive at a time.



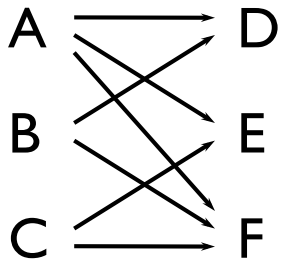
Painter's Algorithm

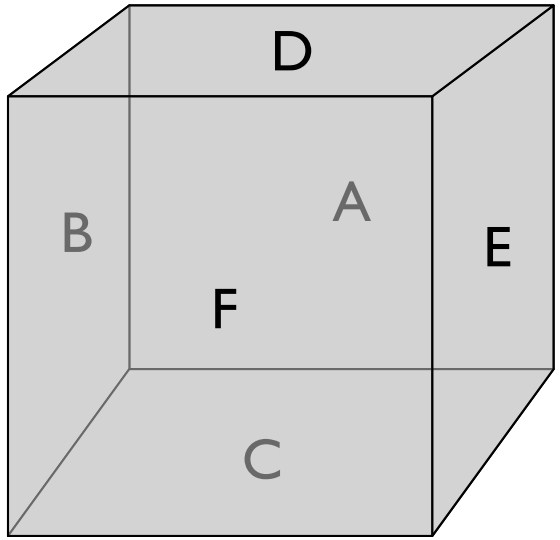
- Draw one primitive at a time.



Painter's algorithm

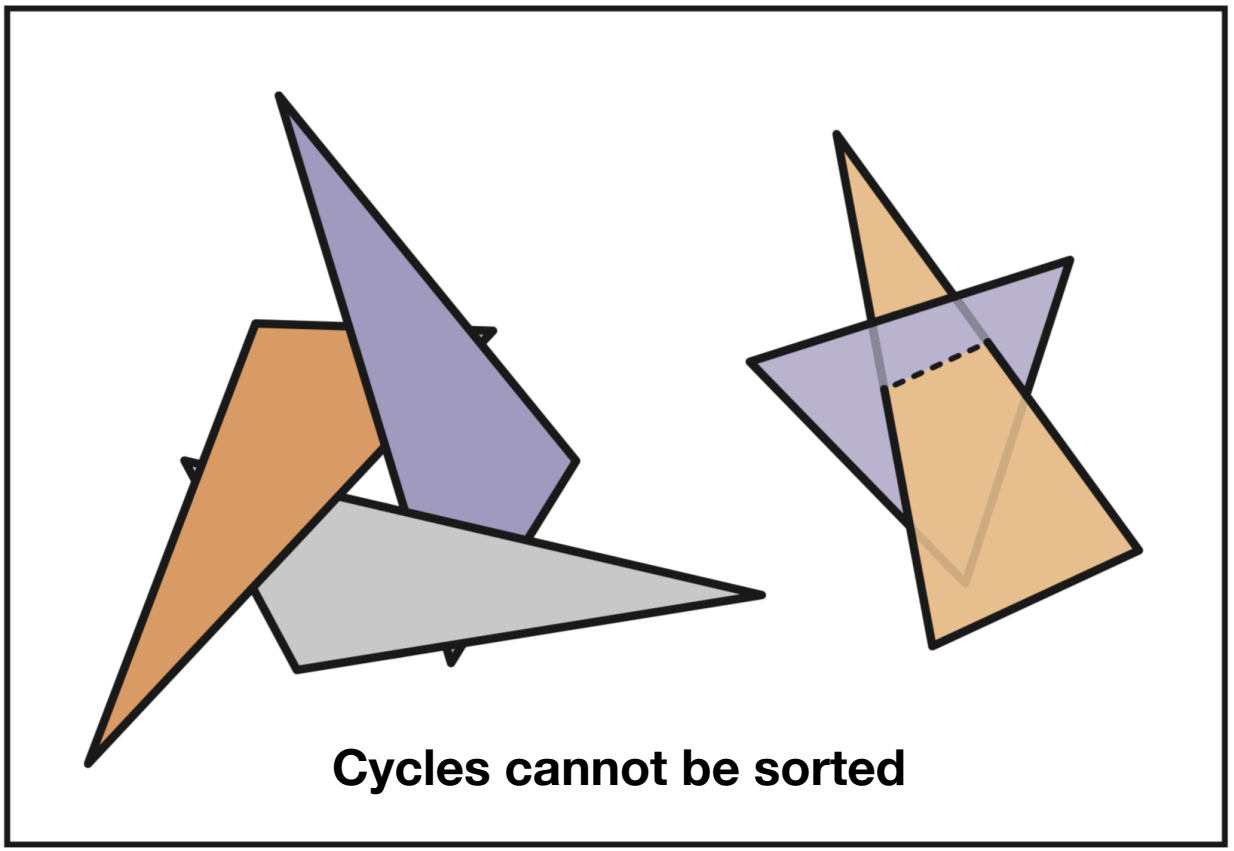
- **Amounts to a topological sort of the graph of occlusions**
  - that is, an edge from A to B means A sometimes occludes B
  - any sort is valid
    - ABCDEF
    - BADCFE
  - if there are cycles there is no sort





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

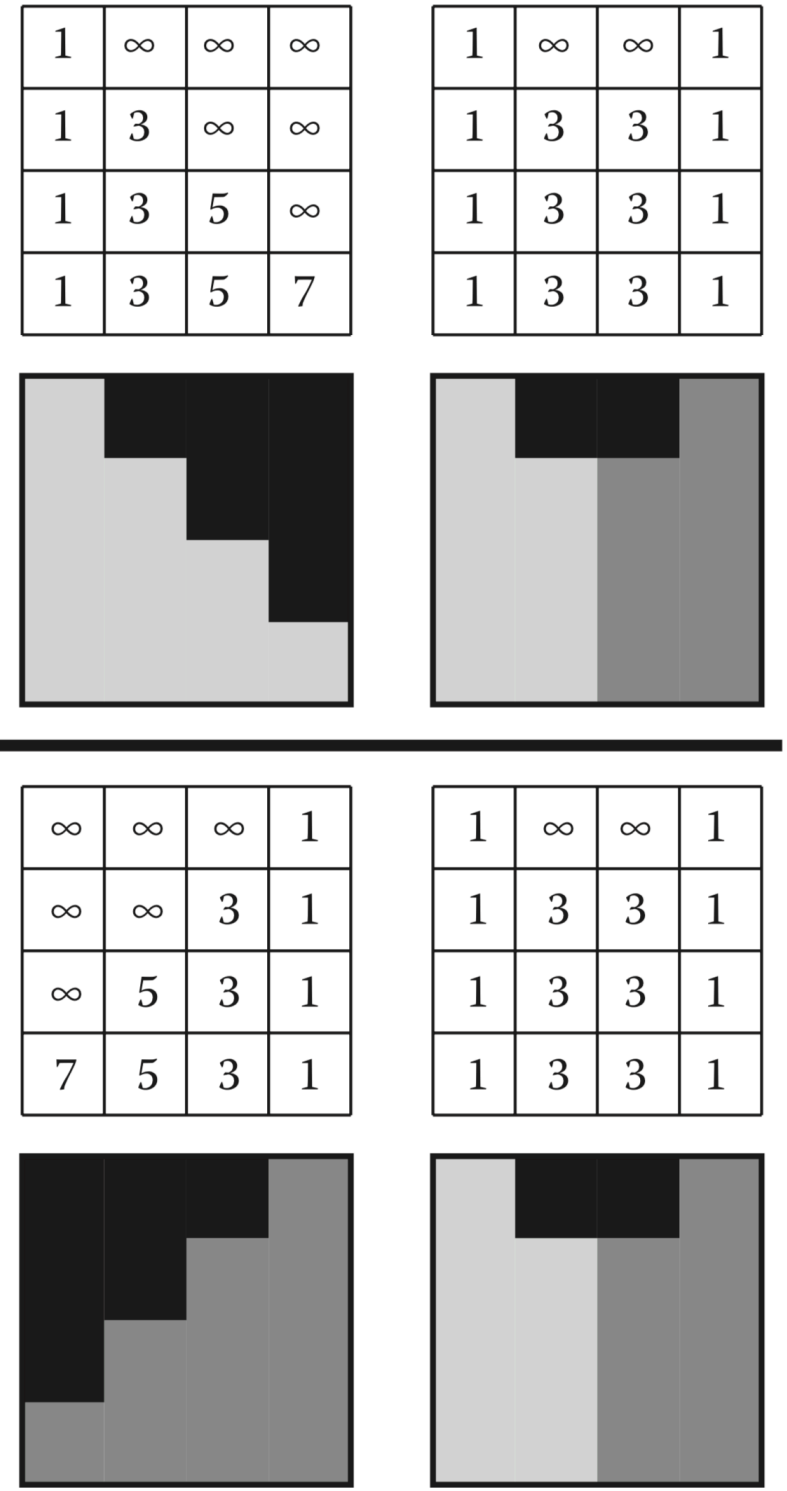


Using a z-Buffer for Hidden Surfaces

- Most of the time, sorting the primitives in z is too expensive and complex
- Solution: draw primitives in any order, but keep track of closest at the **fragment (pixel)** level
  - Method: use an extra data structure that tracks the closest fragment in depth.
  - When drawing, compare fragment's depth to closest current depth and discard the one that is further away

z-Buffers

- Example w/ drawing two triangles
- Order doesn’t matter
- Often depths are stored as integers to keep memory overhead low
- Memory-intensive, brute force approach, but it works and it is the standard because of its simplicity



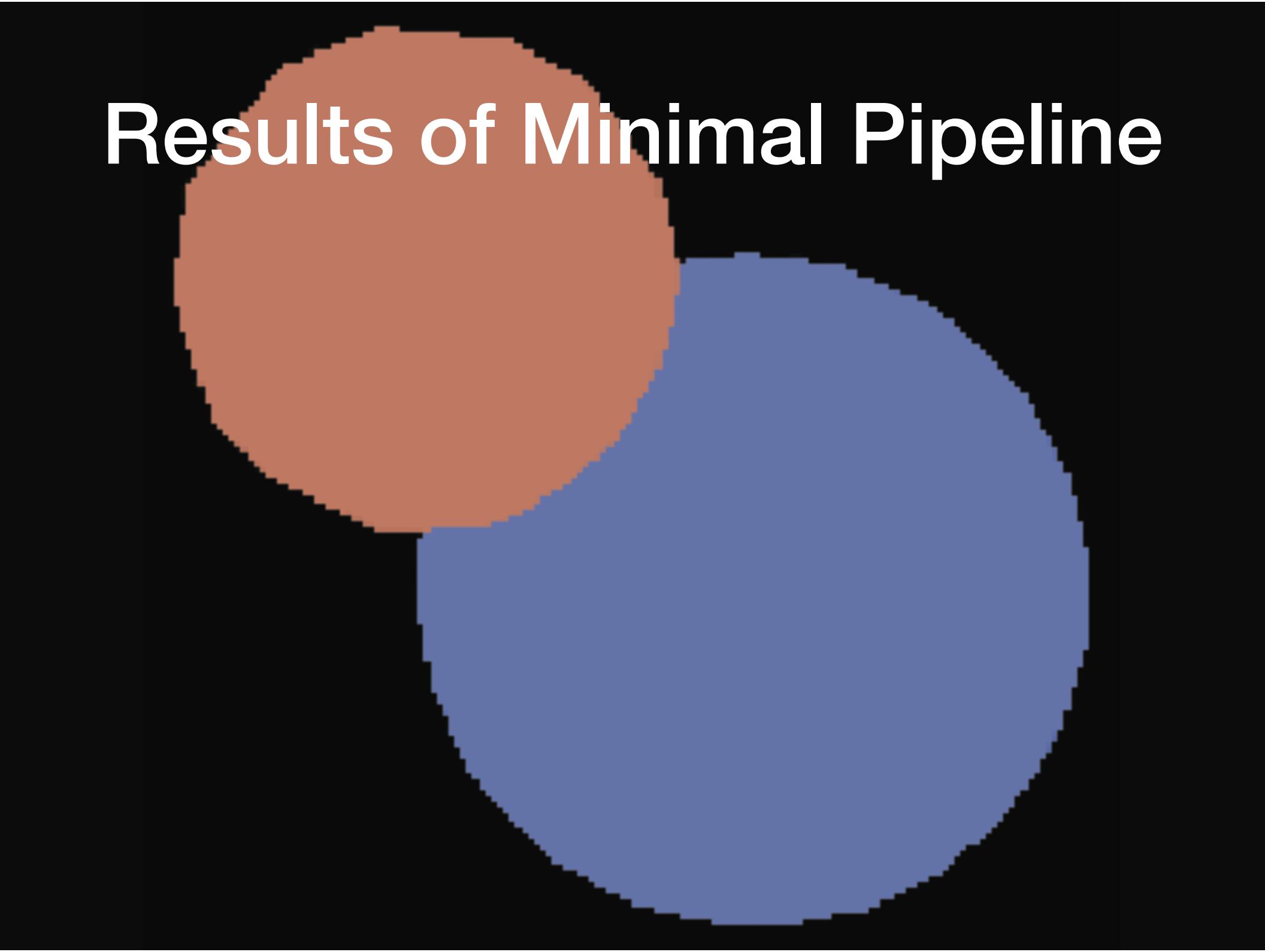
z-Buffer Precision

- If using integers, the precision is related to the distance between the near and far clipping planes
    - Coincidentally, this necessitates the existence of the near/far planes (unlike with ray tracing)
- Generally, use z values that are after transformation
- More info in the book on this topic!

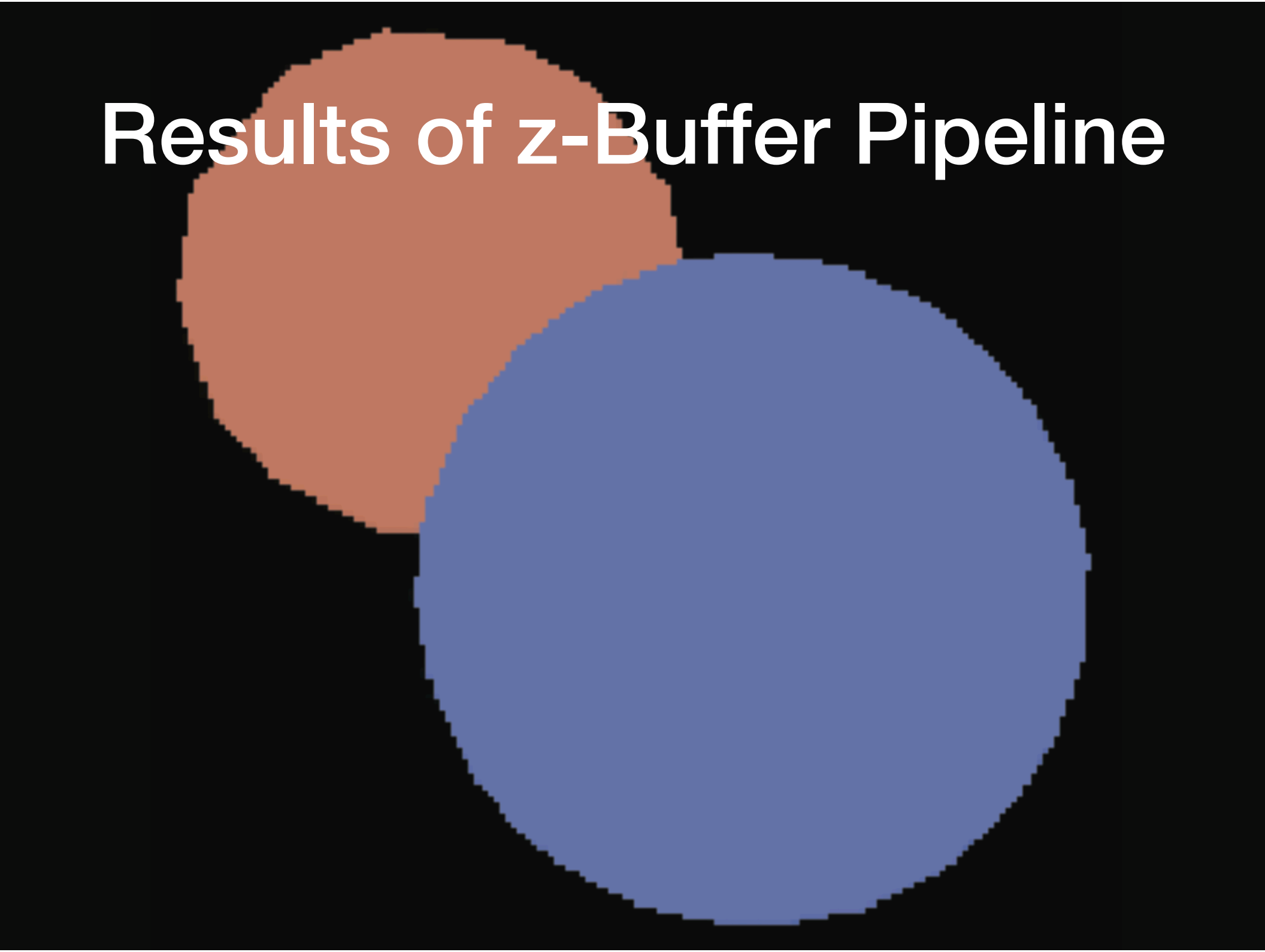
Pipeline for z-Buffer

- Vertex Processing
  - Transform vertex positions, set color for each vertex based on primitive.
- Rasterizer
  - Enumerate fragments for each primitive and set their color.
  - Interpolate the z-value for each fragment
- Fragment Processing
  - Write fragment colors only if interpolated z is closer

Results of Minimal Pipeline



Results of z-Buffer Pipeline



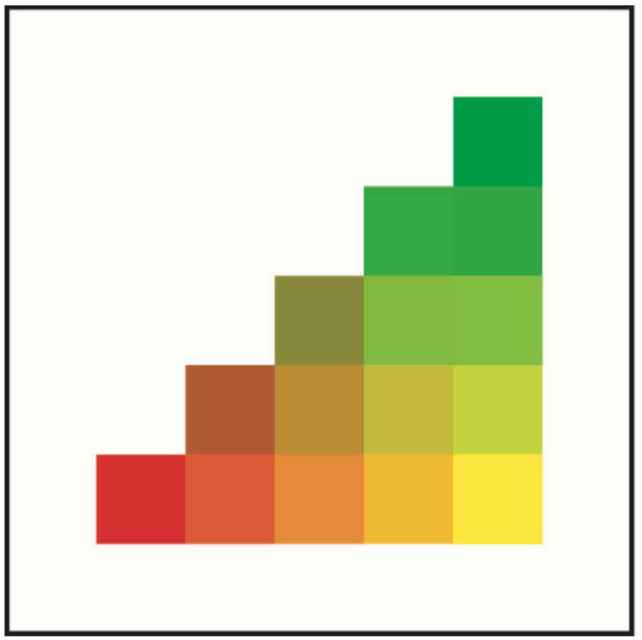
Resolving Issues with Interpolation

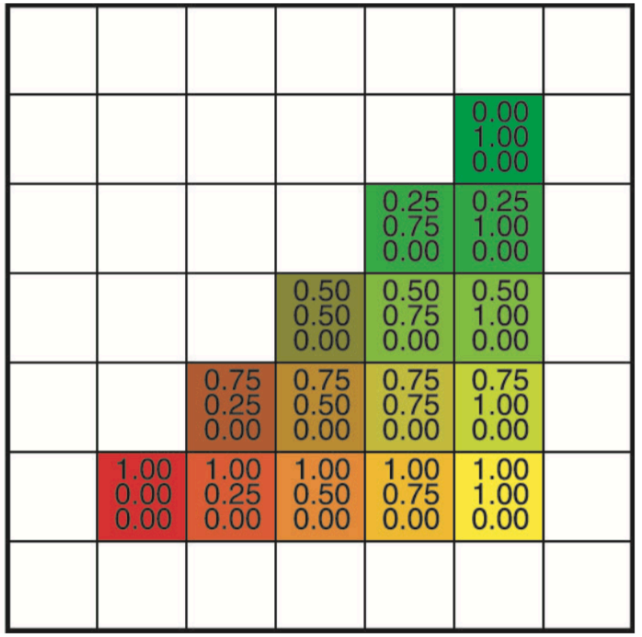
Recall: Screen Space Interpolation of Color

- Having the barycentric coordinates means that we can interpolate color:

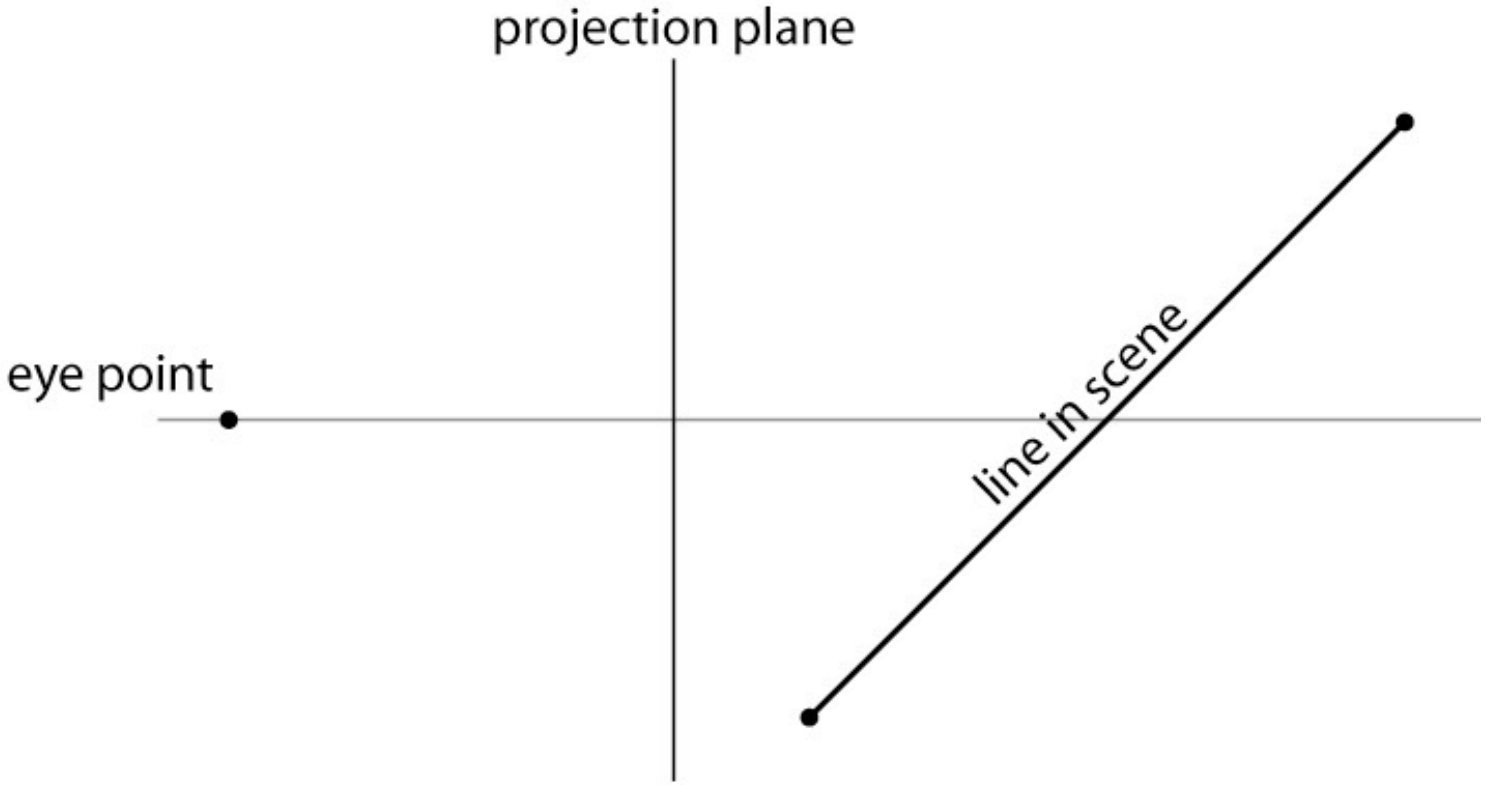
$$\mathbf{c}=\alpha\mathbf{c}_0+\beta\mathbf{c}_1+\gamma\mathbf{c}_2$$

```
for all x in [xmin,xmax] {
  for all y in [ymin,ymax] {
  compute (α, β, γ) for (x,y)
  if (α,β,γ ∈ [0,1]) {
    c = α*c0 + β*c1 + γ*c2
    draw(x,y);
  }
}
```



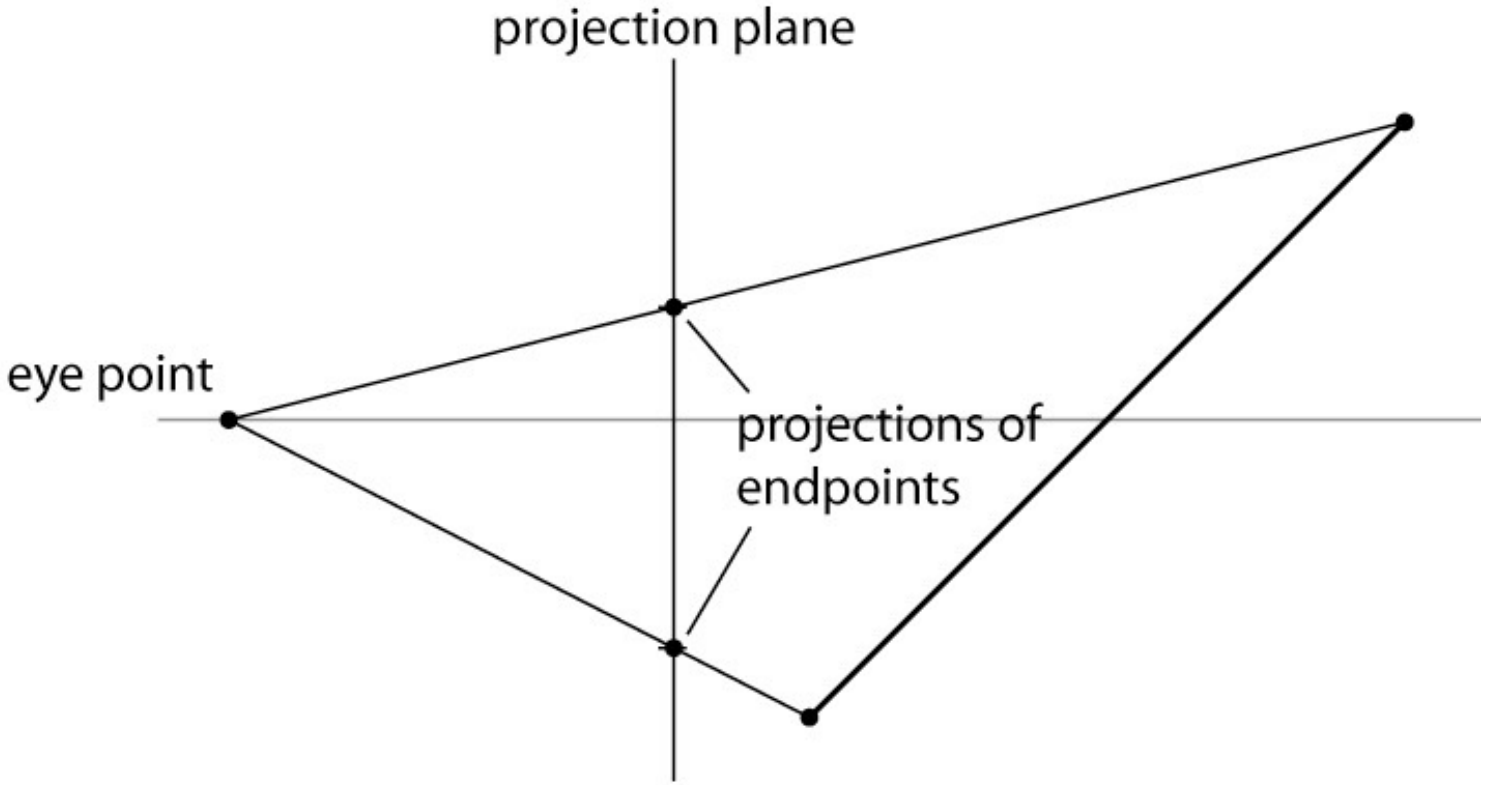


Interpolating in projection



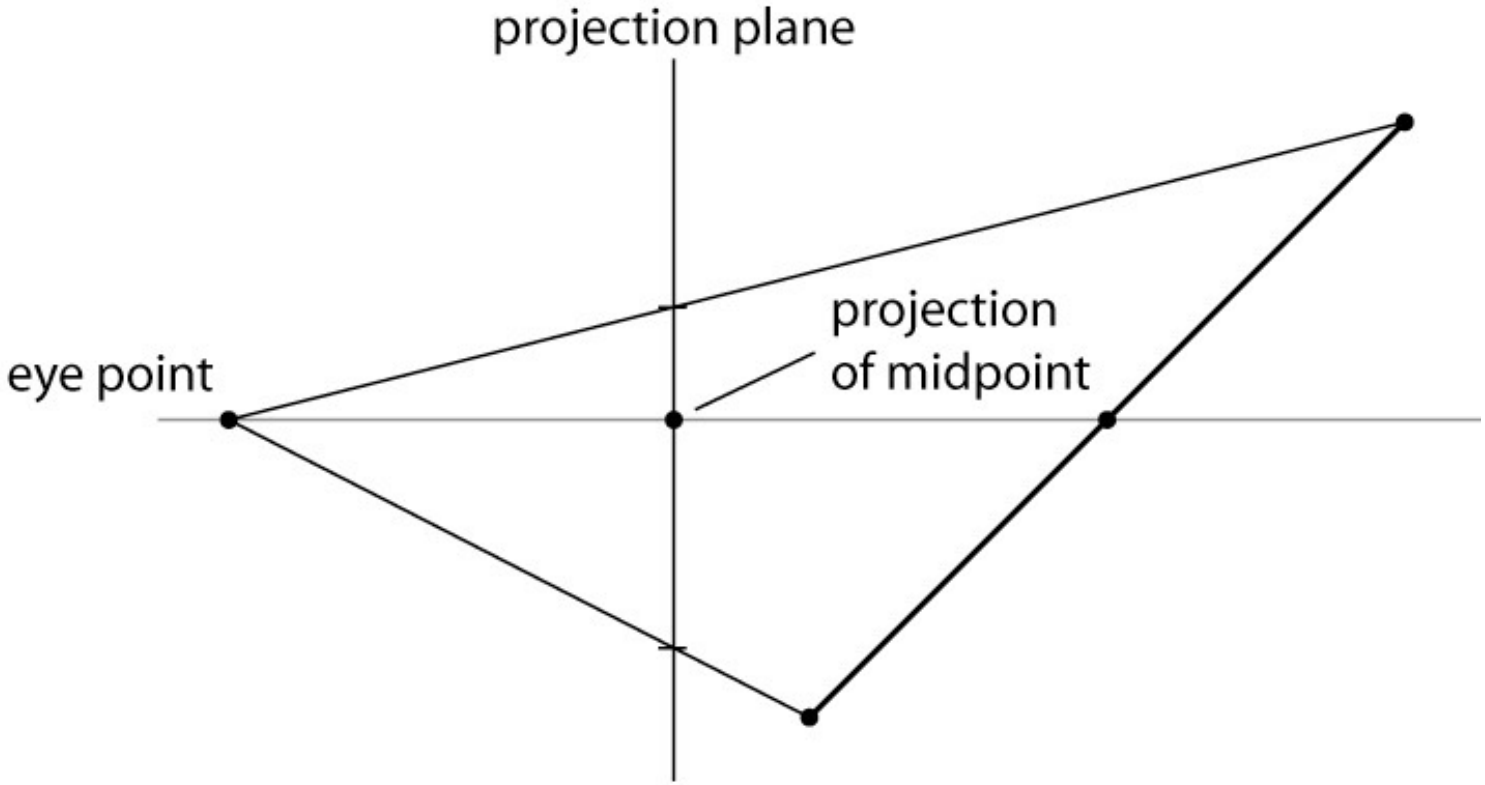
linear interp. in screen space ≠ linear interp. in world (eye) space

Interpolating in projection



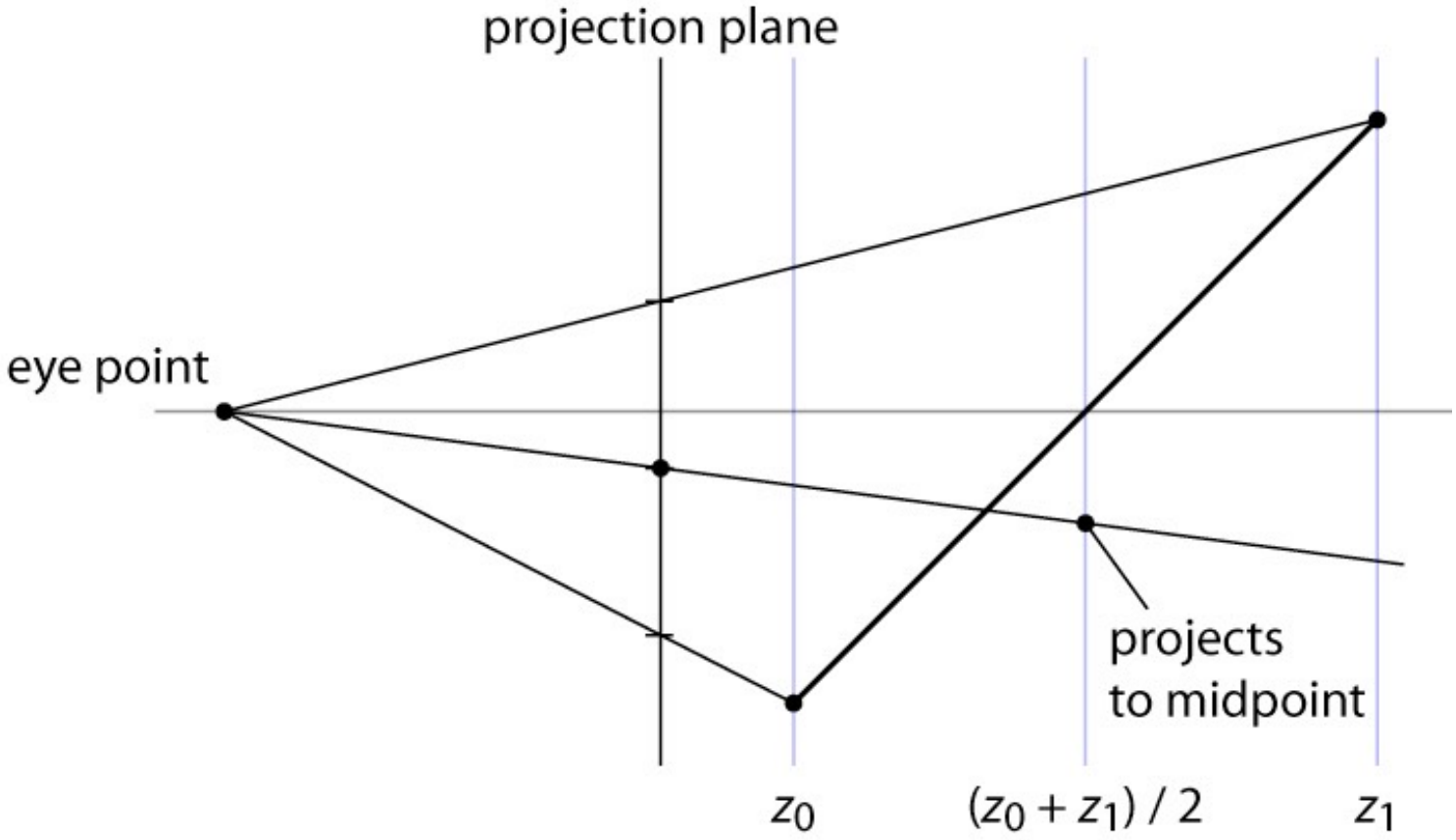
linear interp. in screen space ≠ linear interp. in world (eye) space

Interpolating in projection



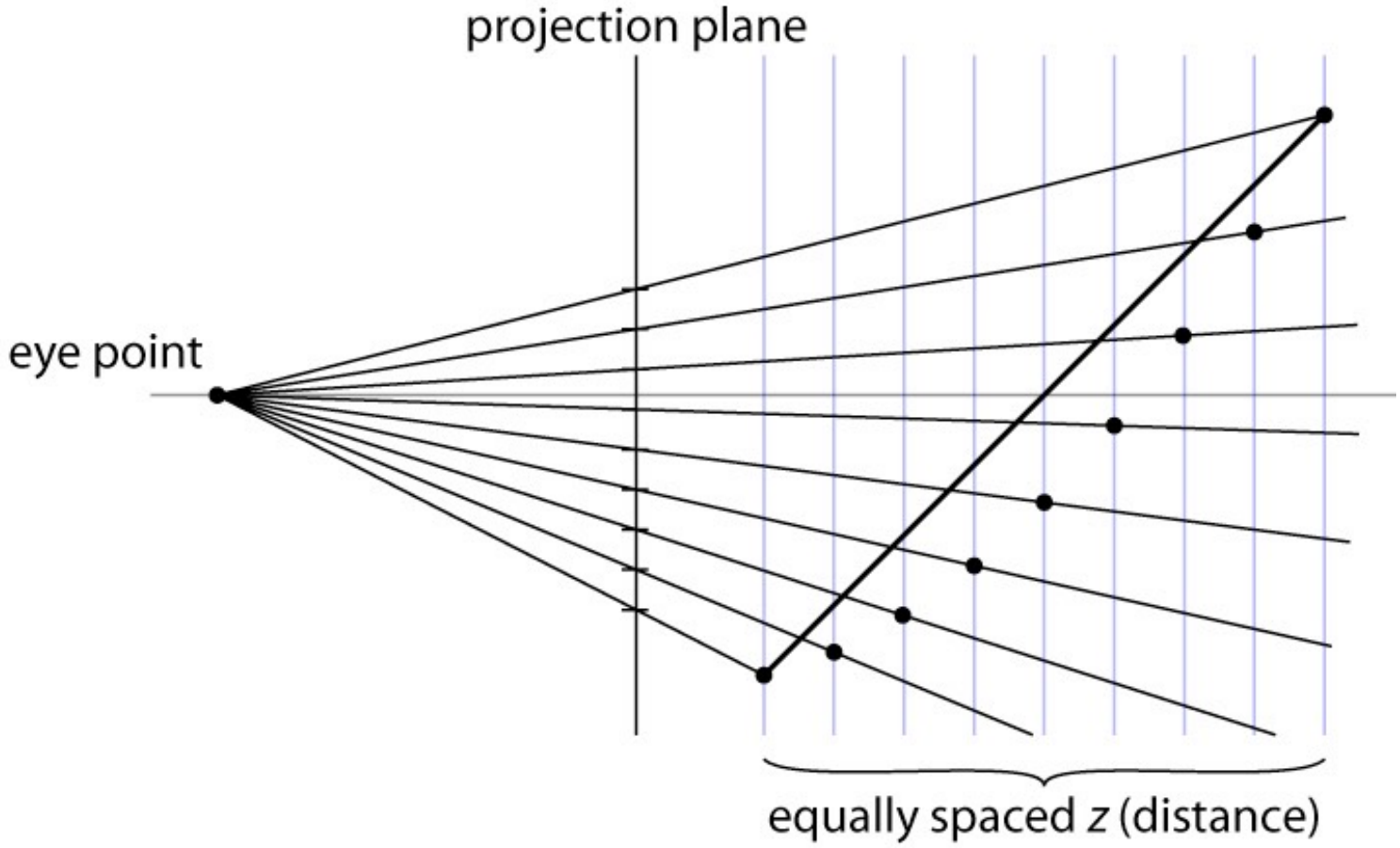
linear interp. in screen space ≠ linear interp. in world (eye) space

Interpolating in projection



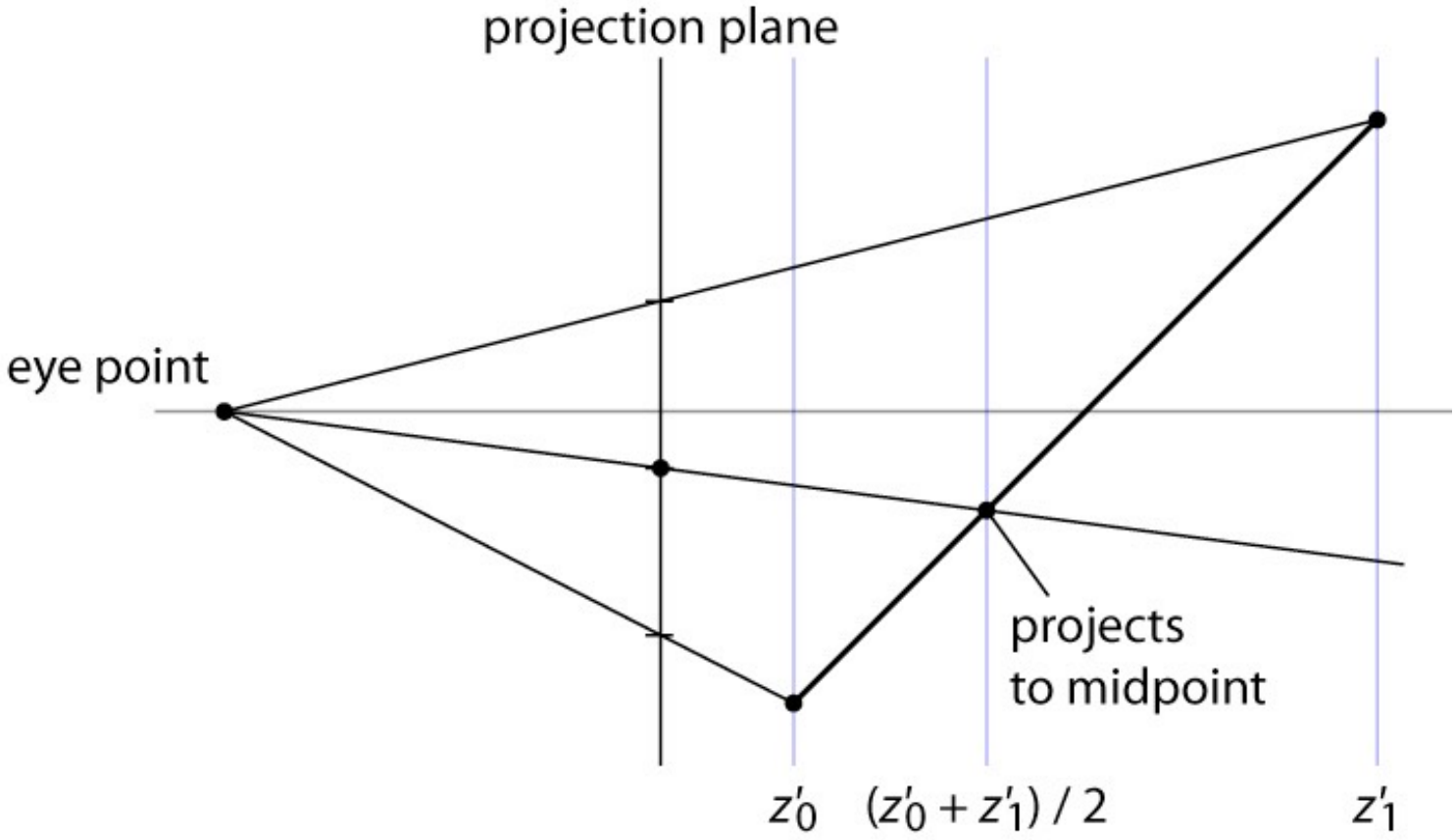
linear interp. in screen space ≠ linear interp. in world (eye) space

Interpolating in projection



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

Interpolating in projection



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

Adding 3D Cues w/ Shading and Lighting

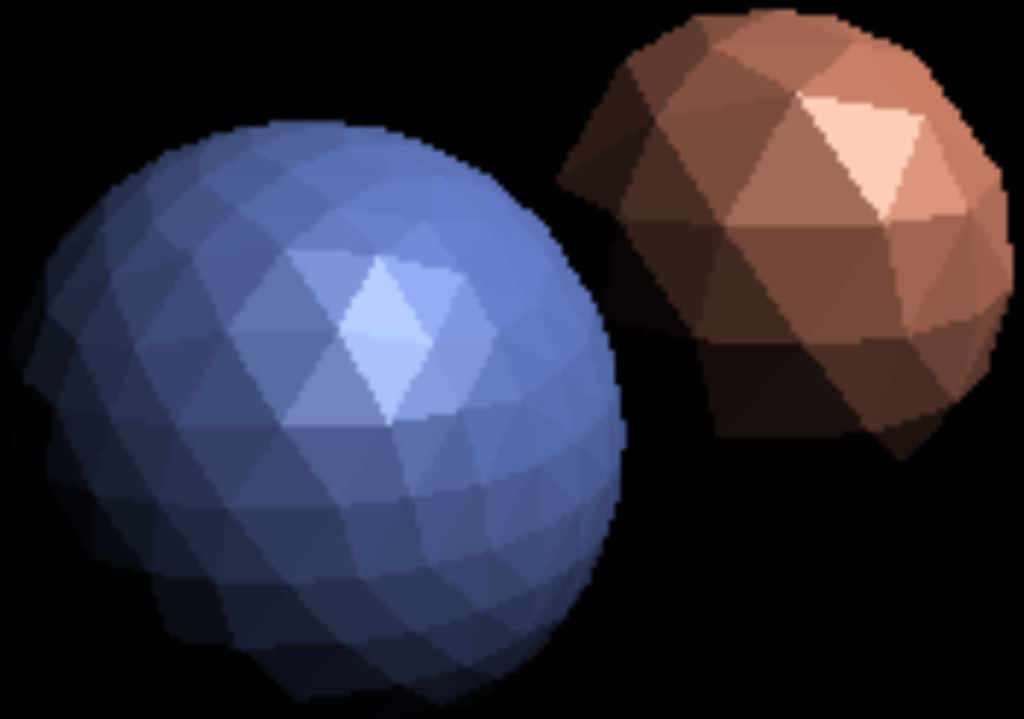
Flat Shading

- Use the normal of the primitive (triangle) to compute a shaded color based on the lights in the scene
- Creates a faceted appearance that highlights the mesh geometry

Pipeline for Flat Shading

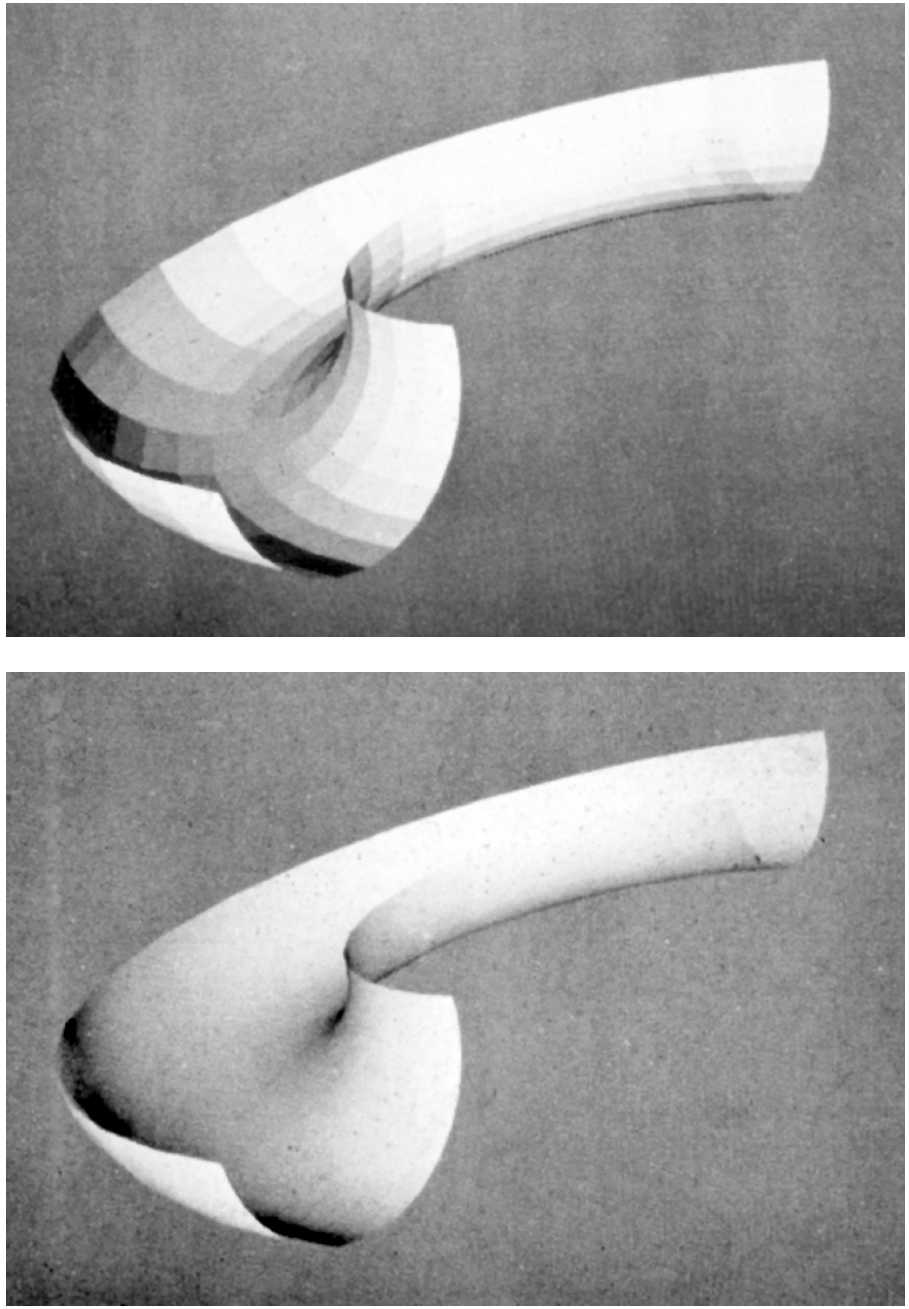
- Vertex Processing
  - Compute shaded color per triangle using triangle normal
  - Transform vertex positions
- Rasterizer
  - Interpolate the z-value for each fragment
  - Set fragment color based on its primitive color.
- Fragment Processing
  - Write fragment colors only if interpolated z is closer

Results of Flat Shading 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]

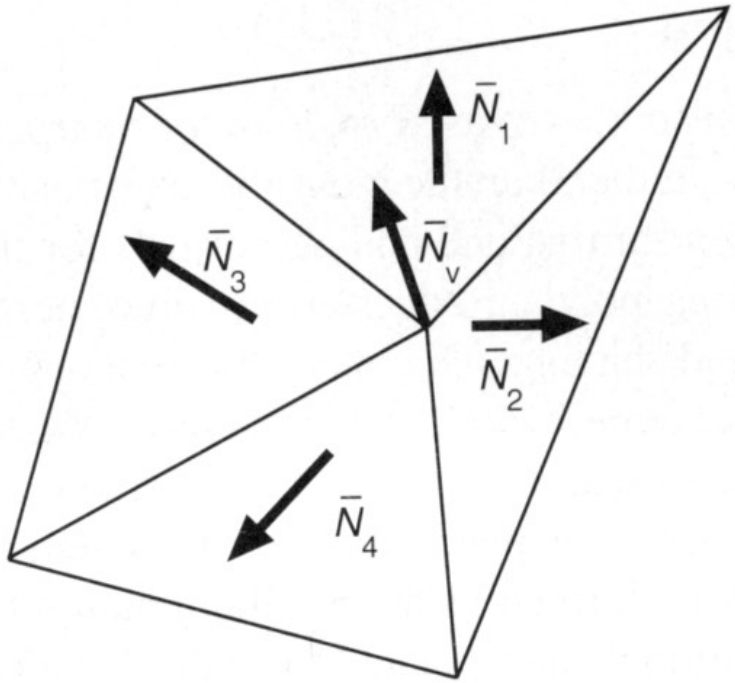
Pipeline for Gouraud Shading

- Vertex Processing
  - Compute shaded color per vertex using vertex normal
  - Transform vertex positions
- Rasterizer
  - Interpolate the z-value for each fragment as well as r,g,b colors
- Fragment Processing
  - Write fragment colors only if interpolated z is closer

Vertex normals

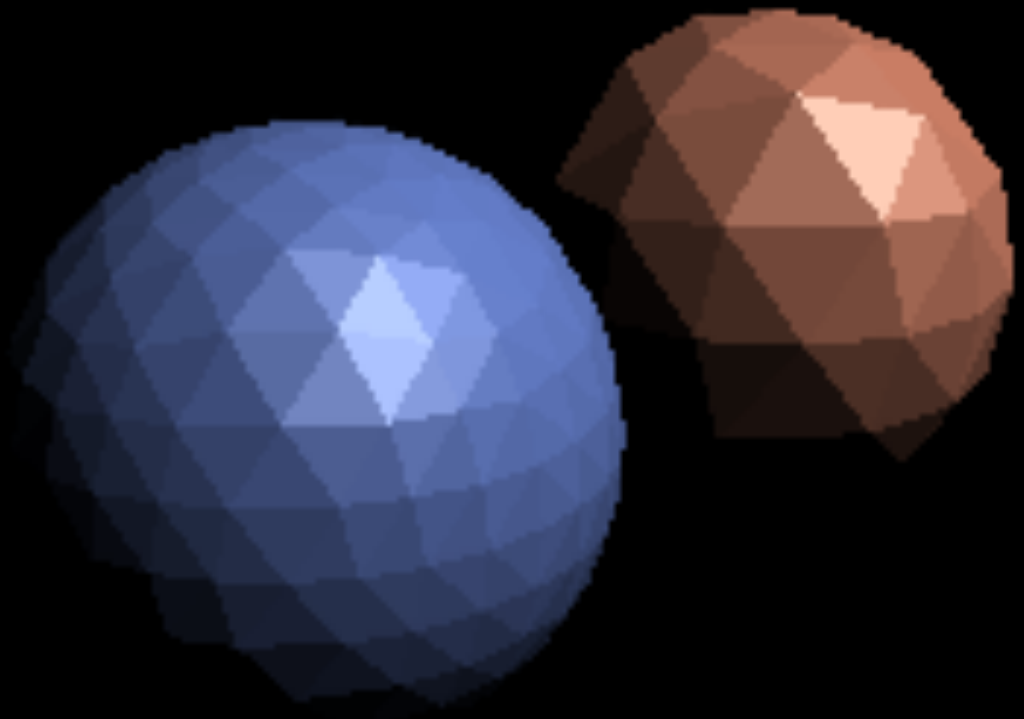
- **Need normals at vertices to compute Gouraud shading**
- **Best to get vtx. normals from the underlying geometry**
  - e. g. spheres example
- **Otherwise have to infer vtx. normals from triangles**
  - simple scheme: average surrounding face normals

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

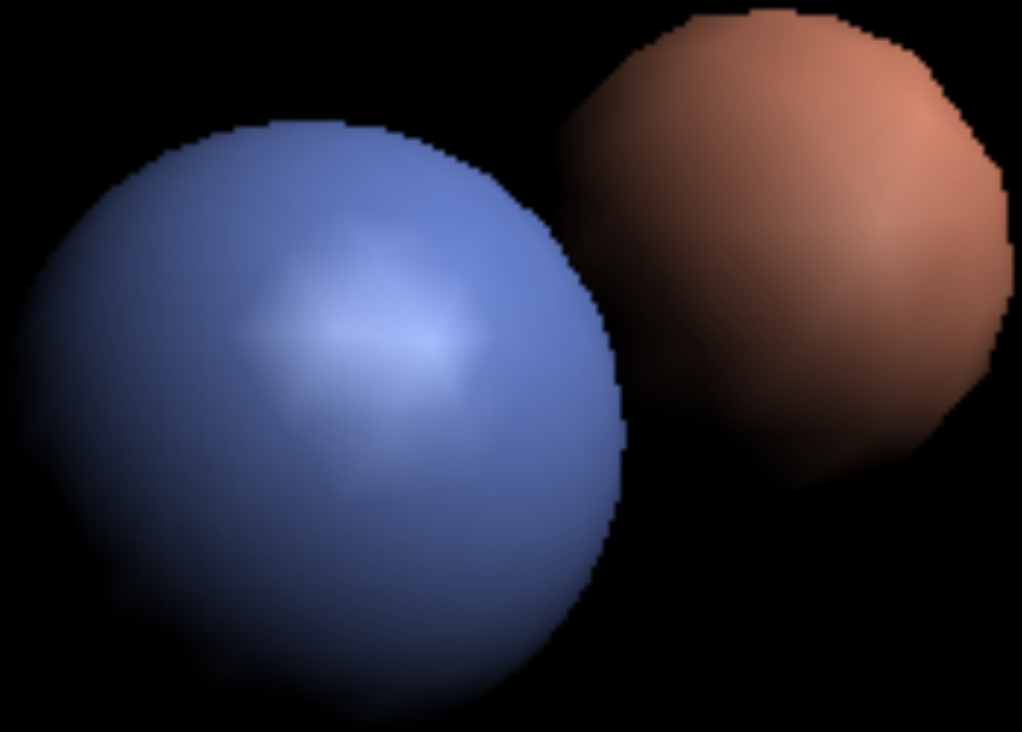


[Foley et al.]

Results of Flat Shading Pipeline



Results of Gouraud Shading Pipeline

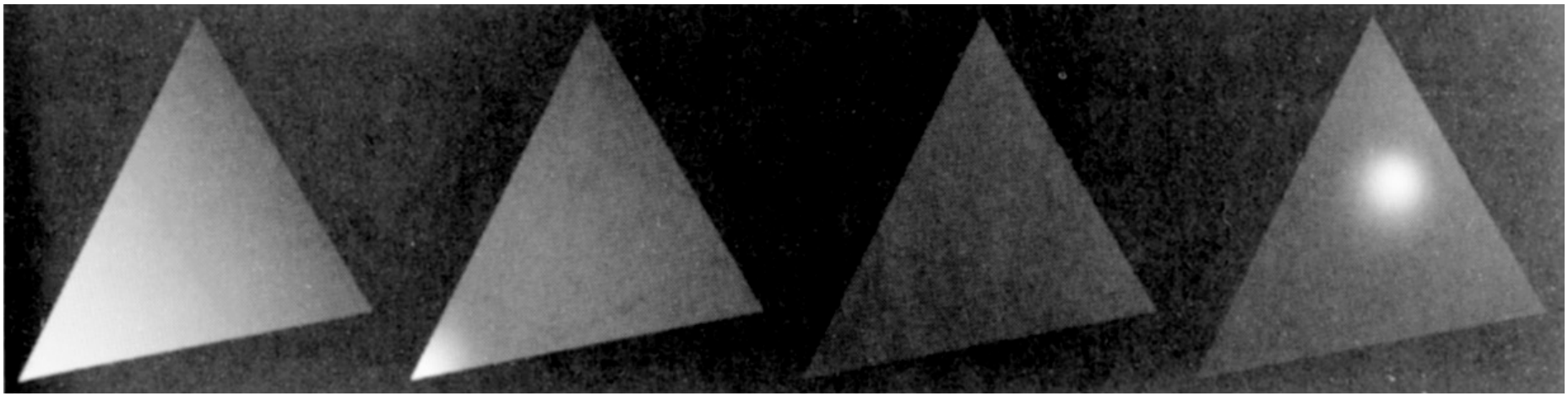


Problems w/ Gouraud Shading

- While you can use any shading model on the vertices (Gouraud shading is just an interpolation method!), typically using only diffuse color works best.
- Results tend to be poor with rapidly-varying models like specular color
    - In particular, when triangles are large relative to how quickly color is changing

Per-pixel (Phong) shading

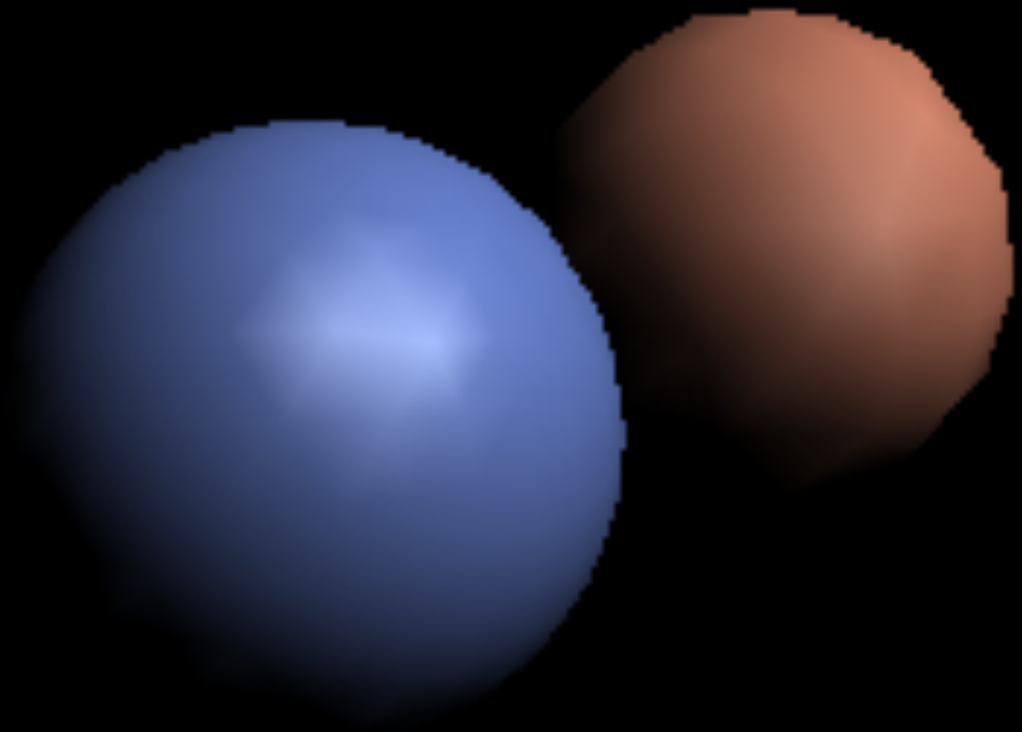
- **Get higher quality by interpolating the normal**
  - just as easy as interpolating the color
  - but now we are evaluating the illumination model per pixel rather than per vertex (and normalizing the normal first)
  - in pipeline, this means we are moving illumination from the vertex processing stage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



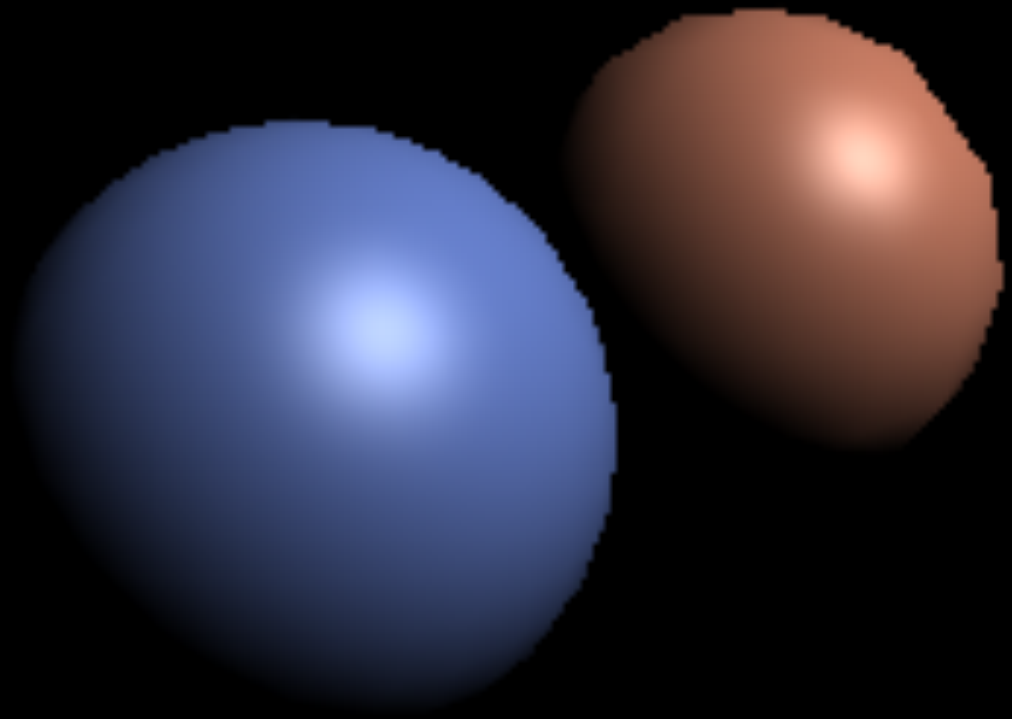
Pipeline for Phong Shading

- Vertex Processing
    - Transform vertex positions, compute normal for vertex based on primitive.
- Rasterizer
    - Interpolate world space position and **normal** of each fragment, and store the barycentric weights
- Fragment Processing
    - Compute shading using position and normal and fixed color for each primitive.
    - Write fragment colors only if interpolated z is closer

Results of Gouraud Shading Pipeline



Results of Phong Shading Pipeline

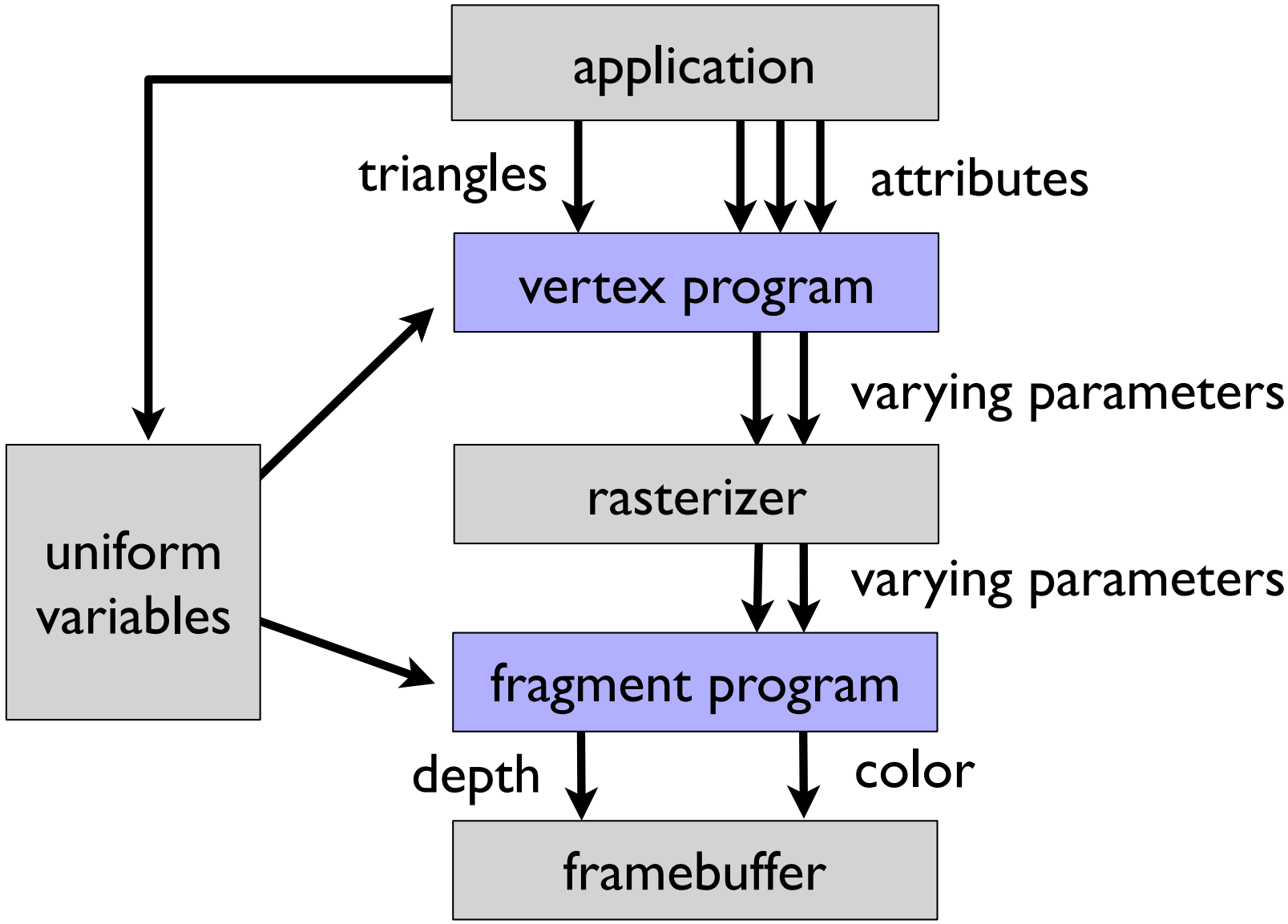


Some Comments on Hardware

Programming hardware pipelines

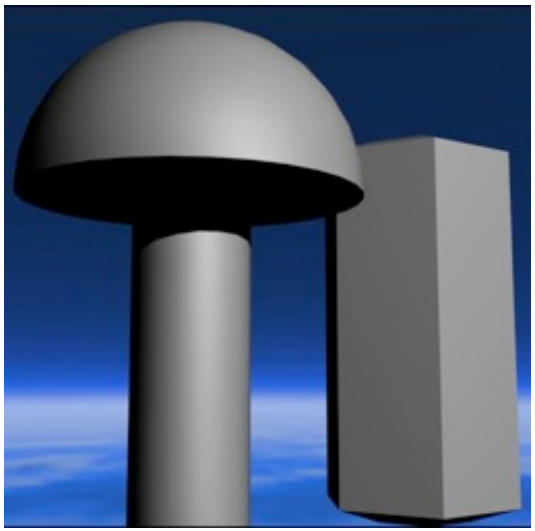
- **Modern hardware graphics pipelines are flexible**
  - programmer defines exactly what happens at each stage
  - do this by writing *shader programs* in domain-specific languages called *shading languages*
  - rasterization is fixed-function, as are some other operations (depth test, many data conversions, …)
- **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*

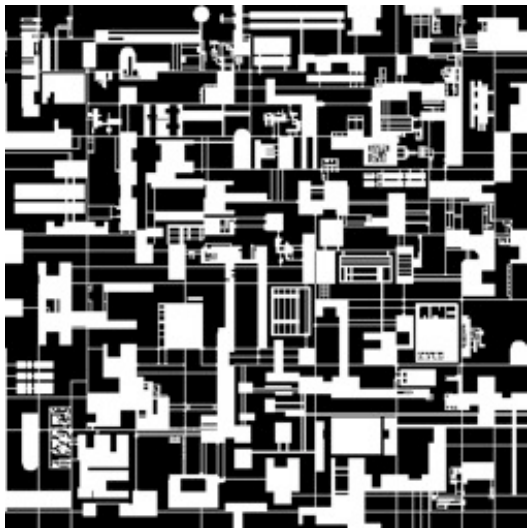
GLSL Shaders

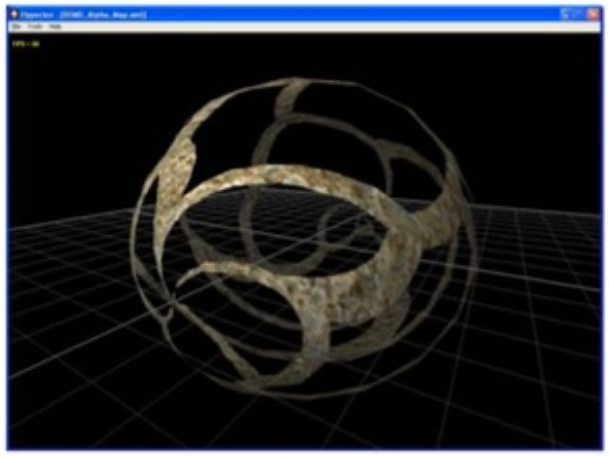


Textures

Opacity mapping

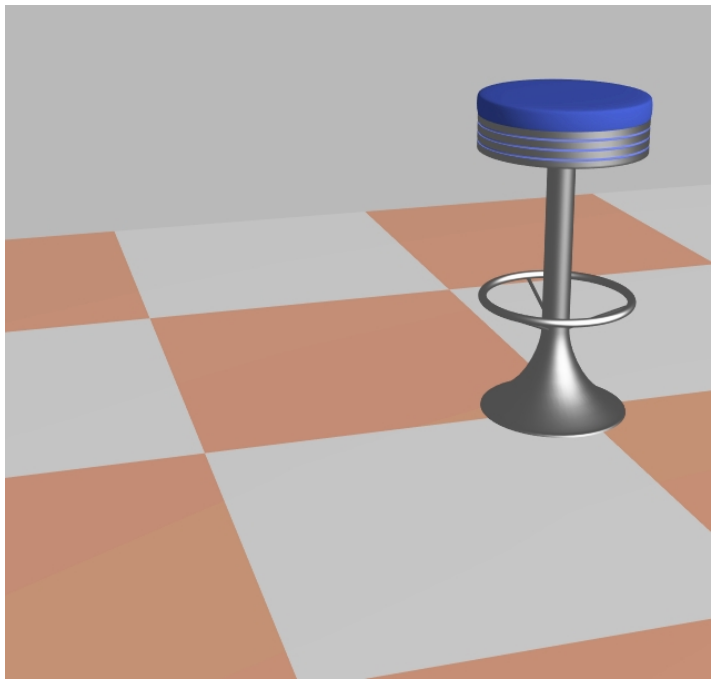


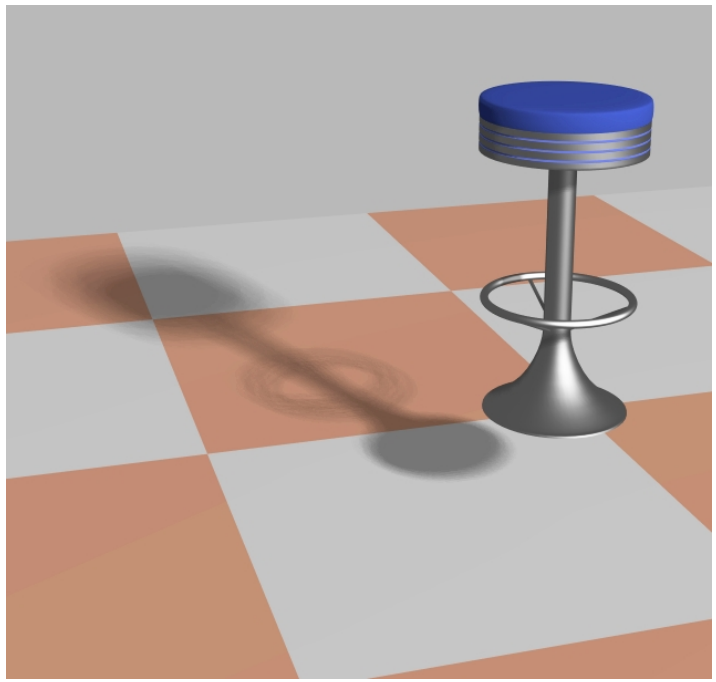




http://lectrablog.lectralizard.com/page/2

Shadows





- Valuable cue of spatial relationships
- Increases realism

Akenine-Moeller and Haines

Three Spaces

- Just like we have mappings from world space to image, we use ϕ as another mapping

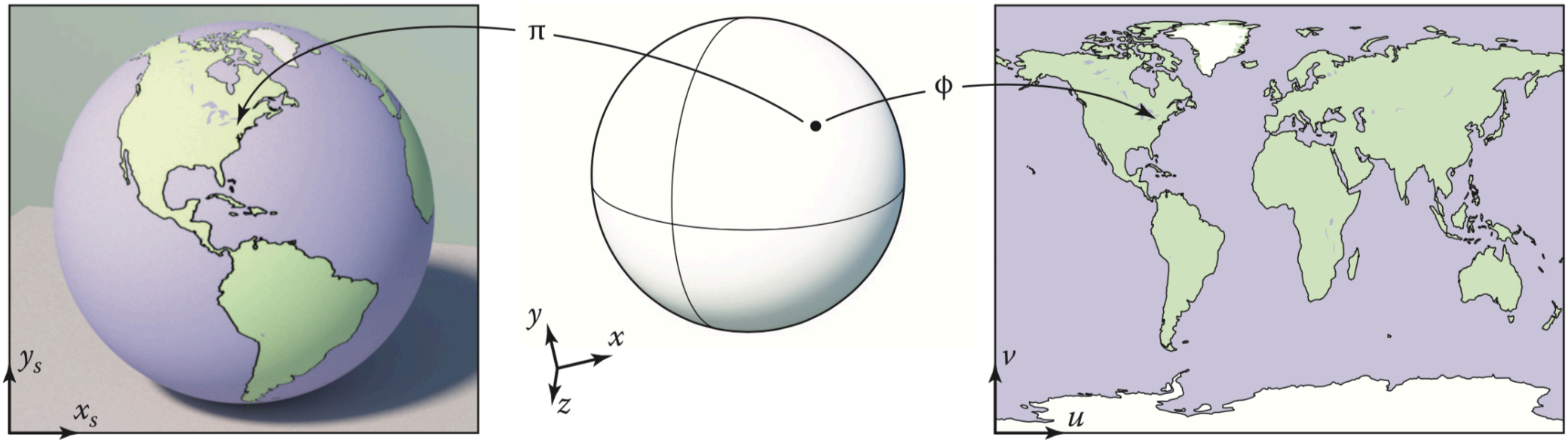


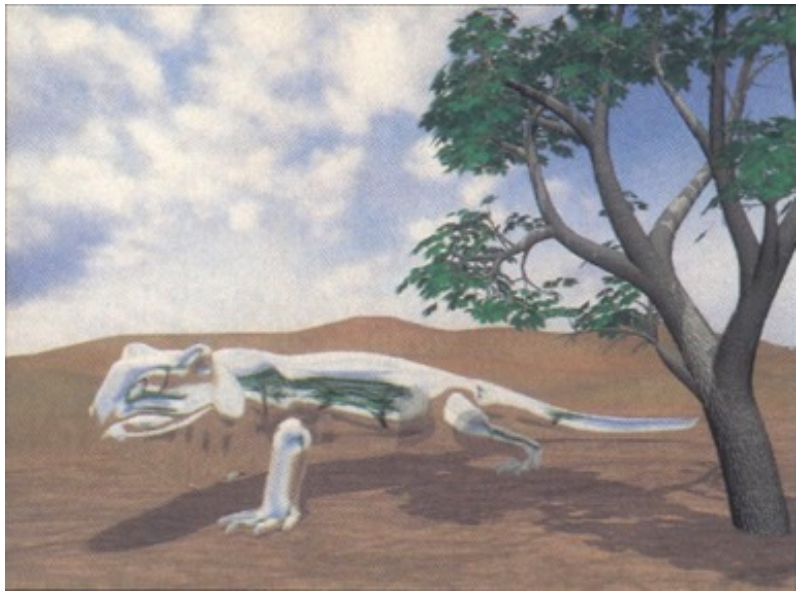
Image space | Surface S in world space | Texture space, T

Reflection mapping

- Render the scene from a single point inside the reflective object. Store rendered images as textures.
- Map textures onto object. Determine texture coordinates by reflecting view ray about the normal.

Terminator 2

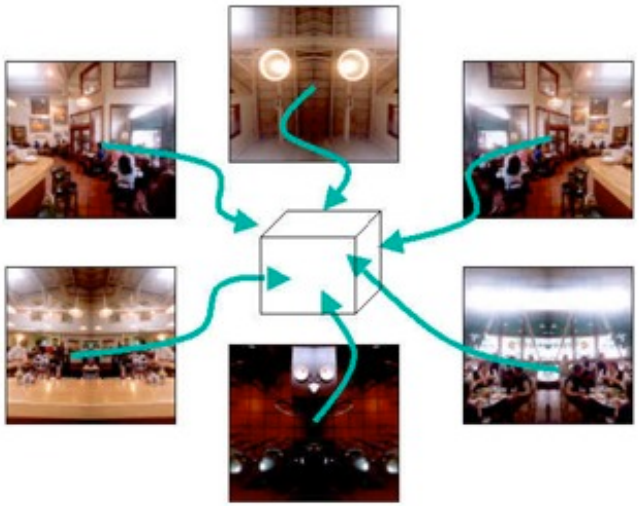
Cube mapping

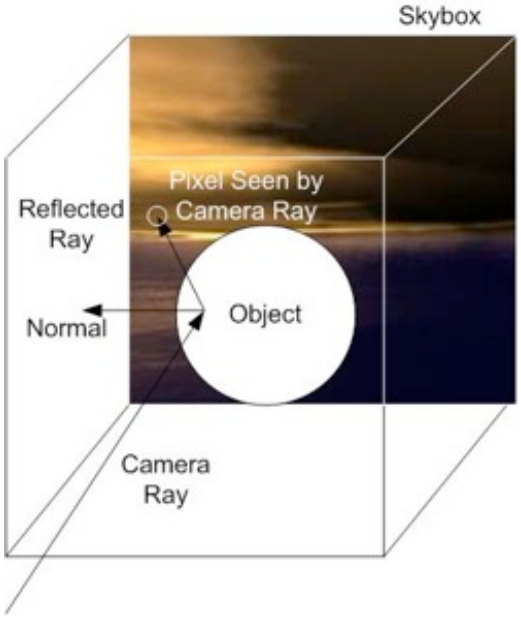


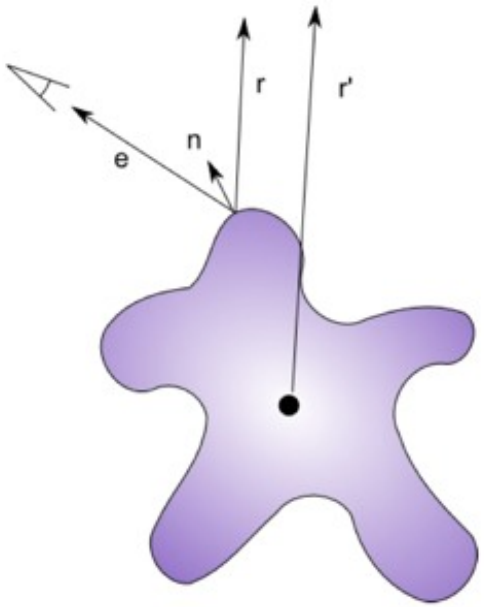
- Render the scene six times, through six faces of a cube, with 90-degree field-of-view for each image.
- Store images in six textures, which represent an omni-directional view of the environment

Greene, 1986

Cube mapping







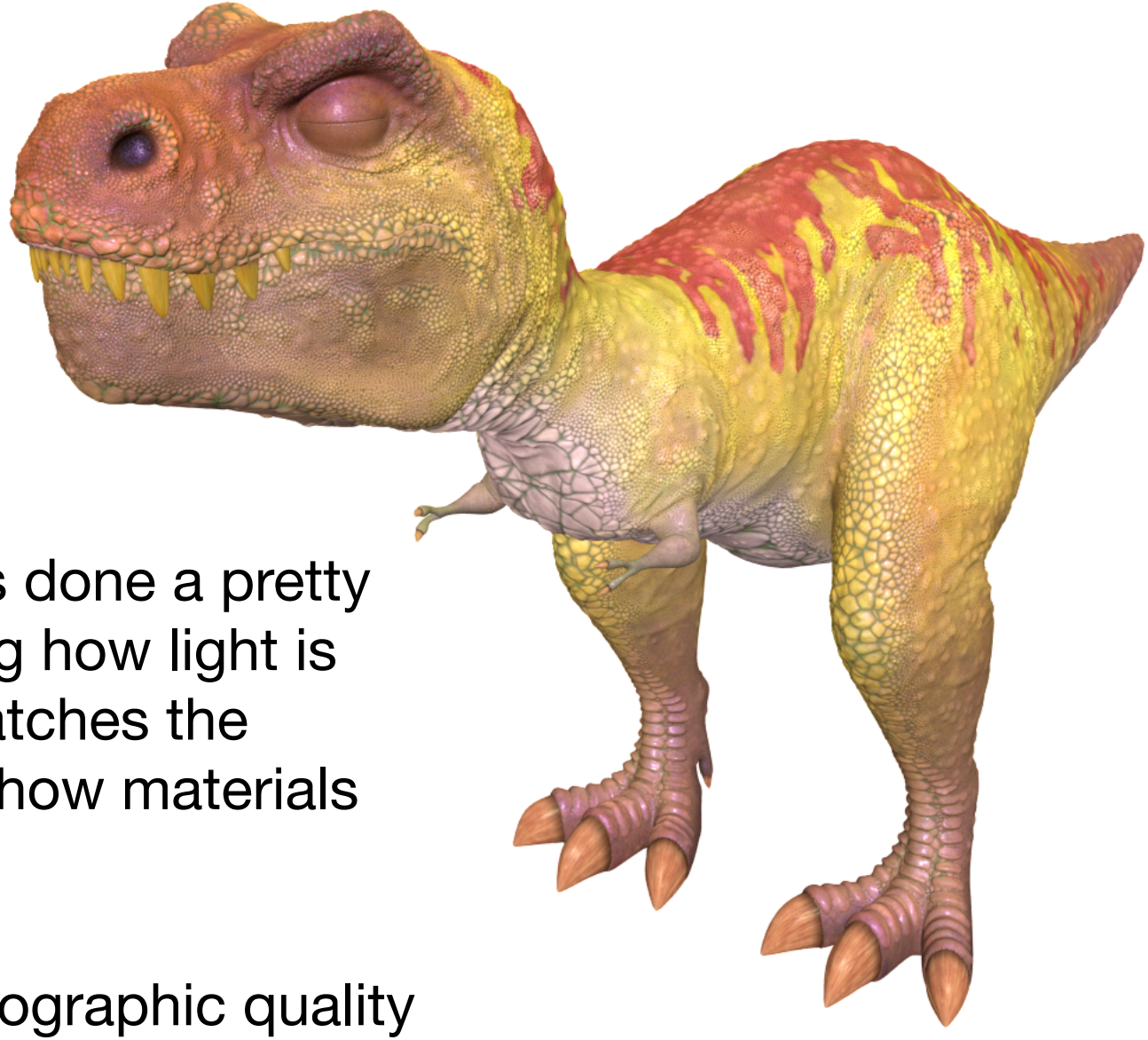
- To compute texture coordinates, reflect the view vector **v** about the normal **n**:

$$\mathbf{r} = 2(\mathbf{v} \cdot \mathbf{n})\mathbf{n} - \mathbf{v}$$

- The highest (in absolute value) coordinate of r identifies which of the six maps we need. The texture coordinates in this map are obtained by normalizing the other two coordinates of r.

http://developer.nvidia.com/object/cube_map_ogl_tutorial.html; TopherTG (Wikipedia)

Challenge: Real World Surfaces Have Complex Materials



- While ray tracing has done a pretty good job of capturing how light is modeled, it only scratches the surface at modeling how materials look

- Goal: Replicate photographic quality by varying shading parameters?

http://ptex.us/ptexpaper.html

Texture Mapping

Texture Mapping

- Models attributes of surfaces that **vary as position changes**, but do not affect the shape of the surface.
- Examples: wood grain, wrinkles in skin, woven structures in cloth, defects in metal surfaces, patterns (in general), …

Texture Maps

- Idea: model this variation using an image, called a **texture map** (or, sometimes “texture image” or just “texture”)
- The texture map stores the surface details
  - Typically, shading parameters like k_d and k_s
- Can be used in lots of interesting ways to achieve complex effects

Tiling and Wrapping

- Can be achieved by modifying the mapping to cycle around in various ways (similar to boundary conditions for image processing)
- Could also just clamp values