# CSC 433/533 Computer Graphics

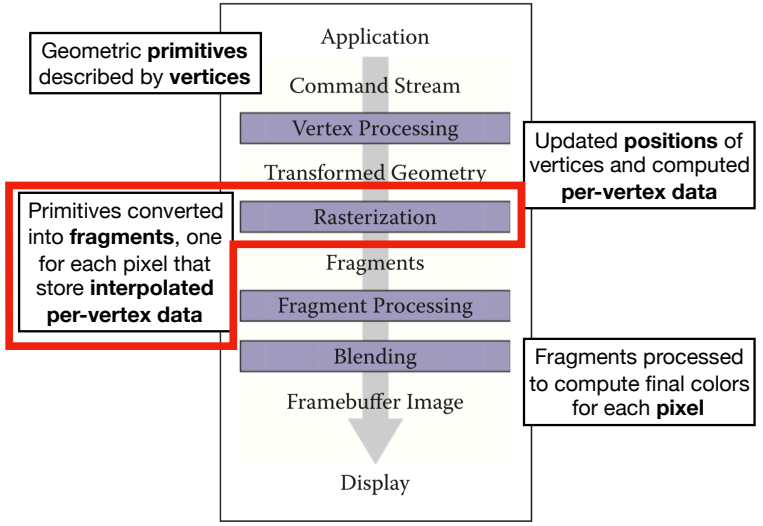
Joshua Levine
josh@email.arizona.edu

# Lecture 21 Graphics Pipeline 2

Nov. 13, 2019

# Today's Agenda

- Reminders:
  - A05 DUE. A06 posted
- Goals for today: finish discussing the graphics pipeline by talking about the operations before and after rasterization



# Recall: Barycentric Coordinates

- A coordinate system to write all points **p** as a weighted sum of the vertices

  $\mathbf{p} = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$

  $\alpha + \beta + \gamma = 1$

- Equivalently, $\alpha$, $\beta$, $\gamma$ are the proportions of area of subtriangles relative total area, A

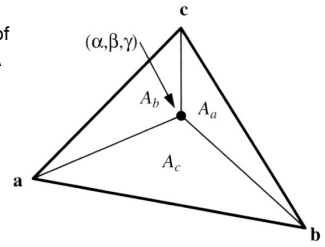
  $A_a / A = \alpha$

  $A_b / A = \beta$

  $A_c / A = \gamma$

- Triangle interior test:

  $\alpha > 0$, $\beta > 0$, and $\gamma > 0$




[Shirley 2000]

# Computing Barycentric Coordinates for Pixels

- We can rely on the equations for each edge of the triangle. Given points $(x_0,y_0)$, $(x_1,y_1)$, and $(x_2,y_2)$ on a triangle, we know:

$$f_{01}(x,y) = (y_o - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0$$
$$f_{12}(x,y) = (y_1 - y_2)x + (x_2 - x_1)y + x_1y_2 - x_2y_1$$
$$f_{20}(x,y) = (y_2 - y_0)x + (x_0 - x_2)y + x_2y_0 - x_0y_2$$

- And we can write the barycentric coordinate as:

$$\alpha = f_{12}(x,y) \,/\, f_{12}(x_0,y_0)$$
$$\beta = f_{20}(x,y) \,/\, f_{20}(x_1,y_1)$$
$$\gamma = f_{01}(x,y) \,/\, f_{01}(x_2,y_2)$$

# Acceleration: #1

- Only check x,y that are in the bounding box of the triangle

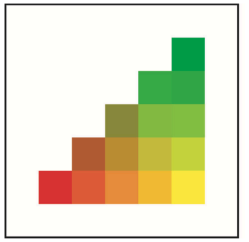
```
xmin = floor(min({x_0,x_1,x_2}))
xmax = ceiling(max({x_0,x_1,x_2}))
ymin = floor(min({y_0,y_1,y_2}))
ymax = ceiling(max({y_0,y_1,y_2}))

for all x in [xmin,xmax] {
  for all y in [ymin,ymax] {
  compute (α, β, γ) for (x,y)
  if (α ∈ [0,1] and β ∈ [0,1] and γ ∈ [0,1]) {
    draw(x,y);
  }
}
```

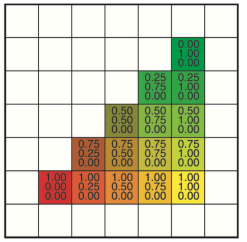
# Interpolating Color w/ Gouraud Shading

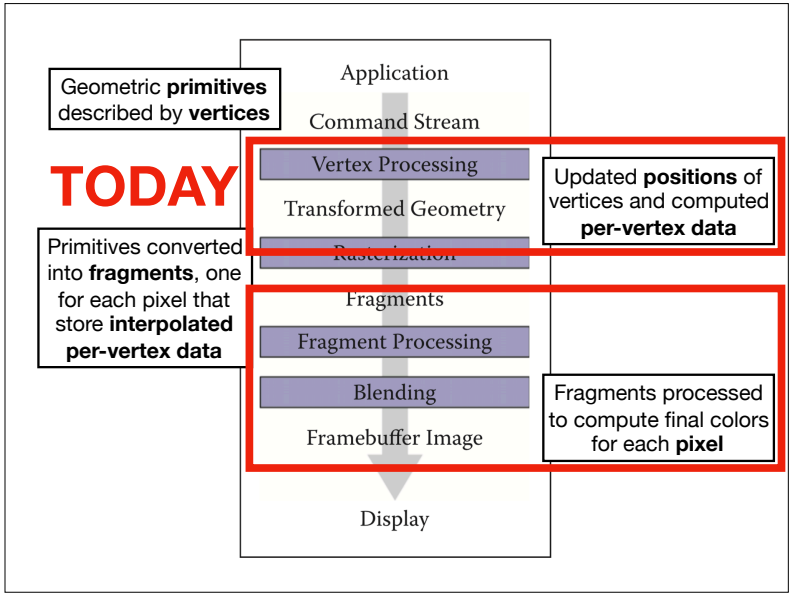
- Having the barycentric coordinates means that we can interpolate color:

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

```
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);
  }
}
```





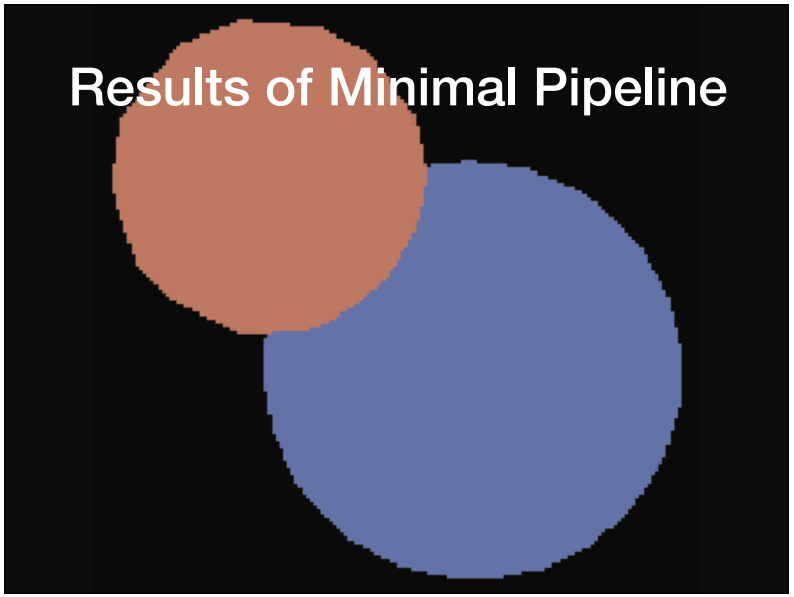



# Operations Before and After Rasterization

- 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

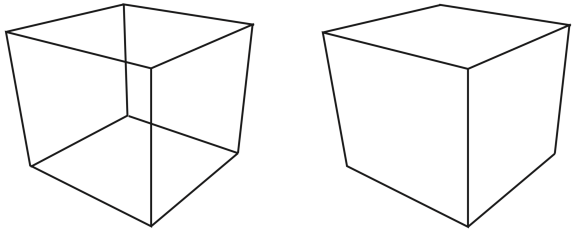
- Vertex Processing
  - Transform vertex positions from object to screen space
  - Set color for each vertex based on primitive.
- Rasterizer
  - Enumerate fragments for each primitive and set their color.
- Fragment Processing
  - Write fragment colors to the appropriate pixel



# Resolving Issues with Depth

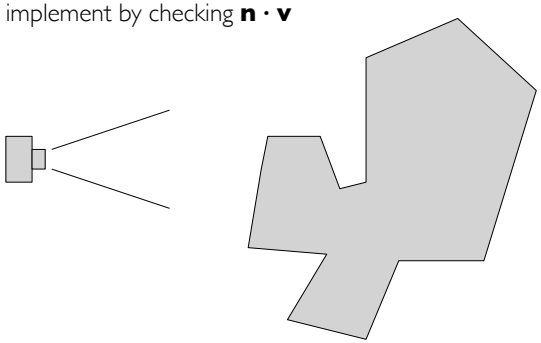
# Hidden Surface Removal

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



## Back face culling

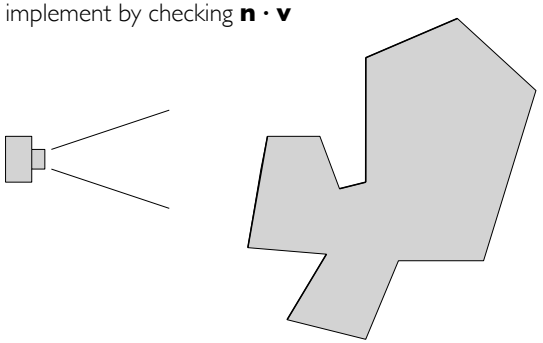
- **For closed shapes you will never see the inside**
  - therefore only draw surfaces that face the camera
  - implement by checking **n** · **v**





## Back face culling

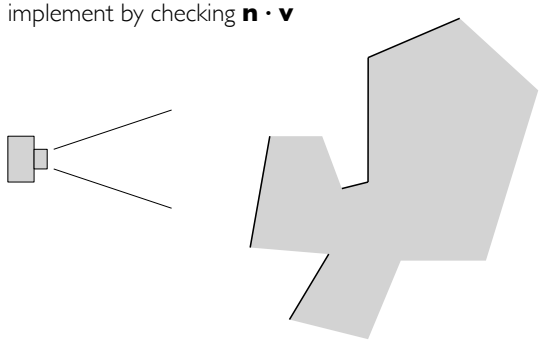
- **For closed shapes you will never see the inside**
  - therefore only draw surfaces that face the camera
  - implement by checking **n** · **v**



Cornell CS4620 Spring 2018 • Lecture 11 © 2018 Steve Marschner • 5

## Back face culling

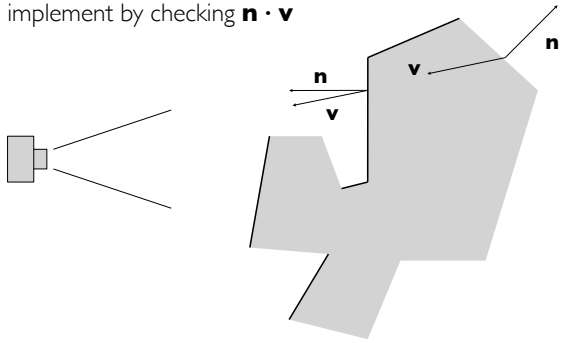
- **For closed shapes you will never see the inside**
  - therefore only draw surfaces that face the camera
  - implement by checking **n** · **v**





## Back face culling

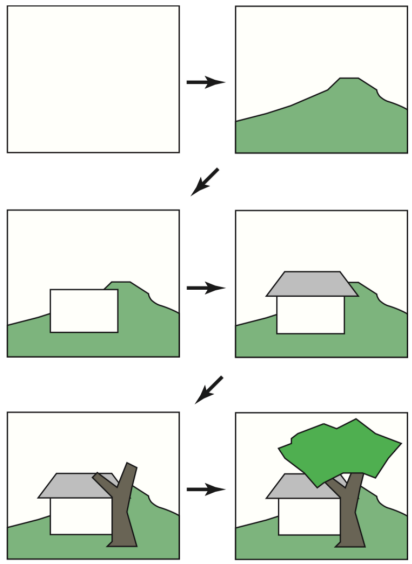
- **For closed shapes you will never see the inside**
  - therefore only draw surfaces that face the camera
  - implement by checking **n** · **v**






## Painter's Algorithm

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

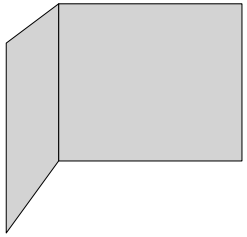


## Painter's Algorithm

- Draw one primitive at a time.

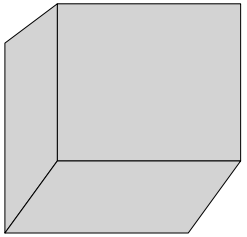
# Painter's Algorithm

- Draw one primitive at a time.



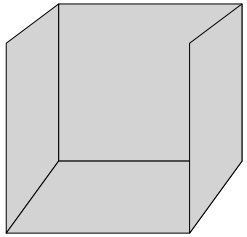
# Painter's Algorithm

- Draw one primitive at a time.



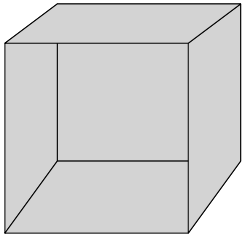
# Painter's Algorithm

- Draw one primitive at a time.



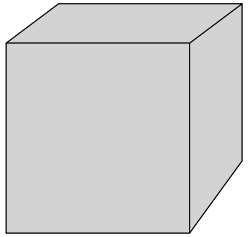
# Painter's Algorithm

- Draw one primitive at a time.



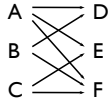
# Painter's Algorithm

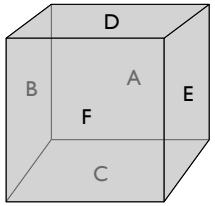
- Draw one primitive at a time.



## 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

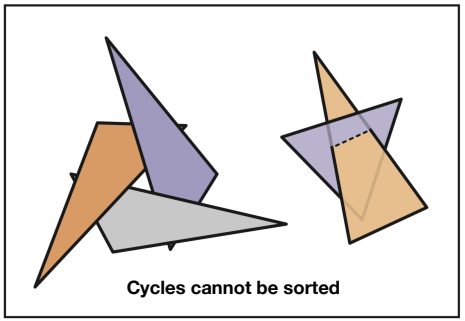





## Painter's algorithm

- **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



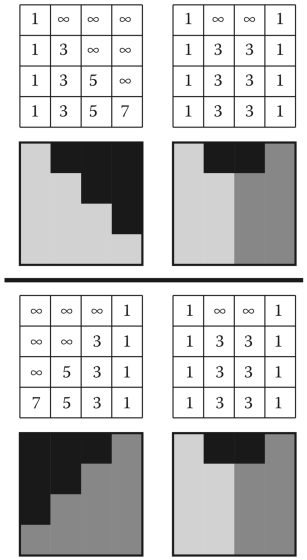



# Using a z-Buffer for Hidden Surfaces

- 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 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

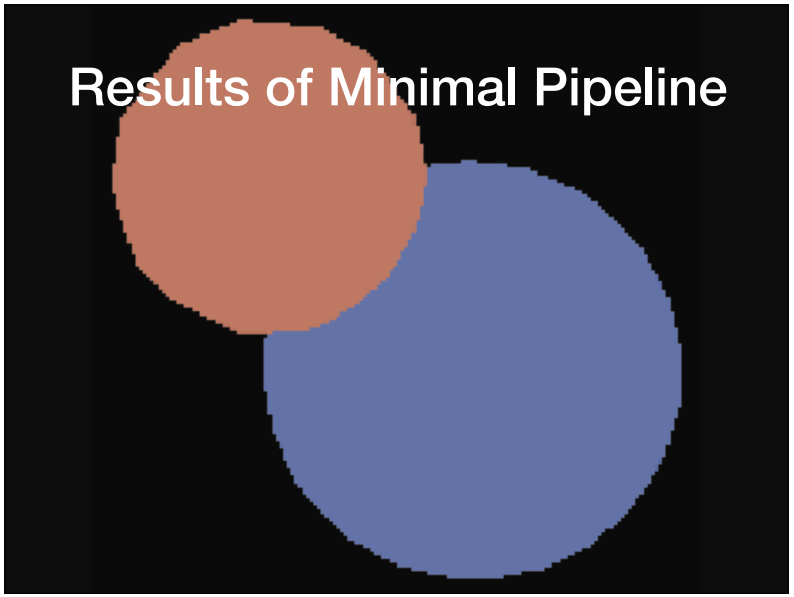



# z-Buffer Precision

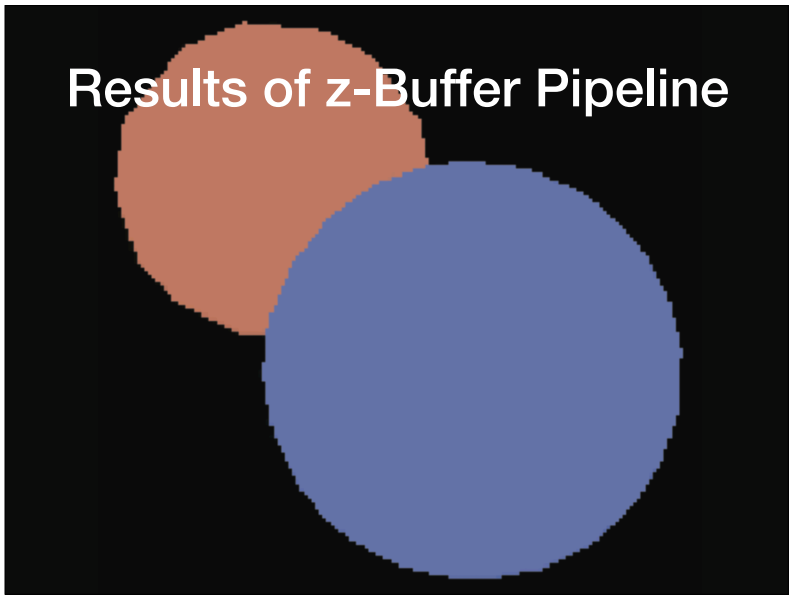
- If using integers, the precision is related to the distance between the near and far clipping planes
  - Coincidentally, this necessitates the existence of the near/far planes (unlike with ray tracing)
- Generally, use z values that are after transformation
- More info in the book on this topic!

# 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 z-Buffer Pipeline



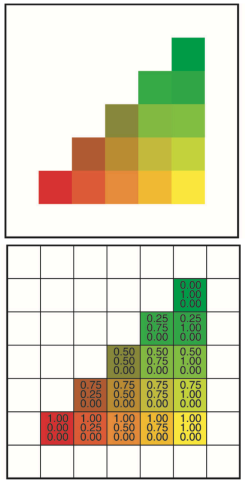
# Resolving Issues with Interpolation

## Recall: Screen Space Interpolation of Color

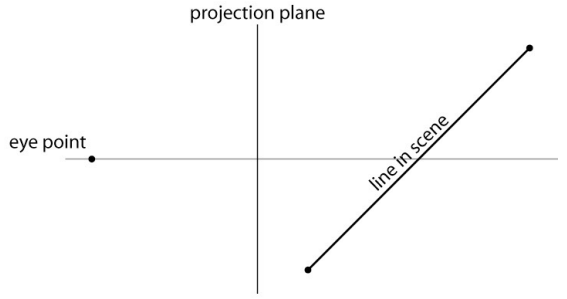
- Having the barycentric coordinates means that we can interpolate color:

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

```
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);
  }
}
```

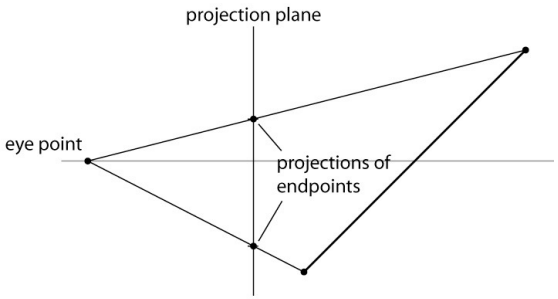


## Interpolating in projection




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

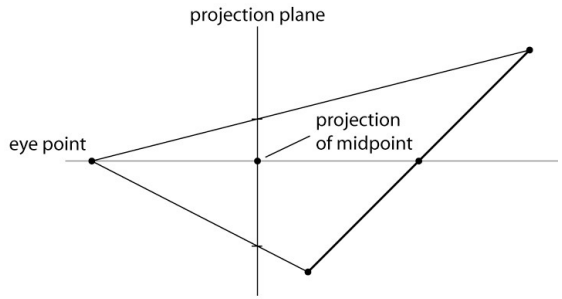
# Interpolating in projection




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



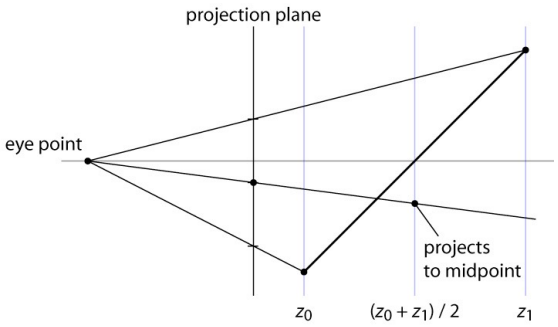
# Interpolating in projection




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



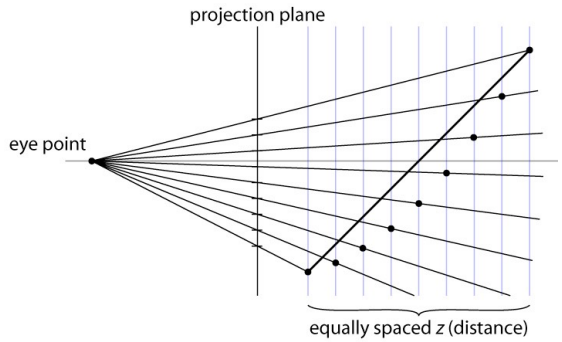
# Interpolating in projection




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

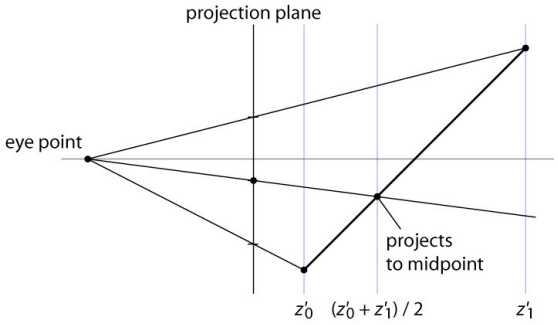


# Interpolating in projection




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

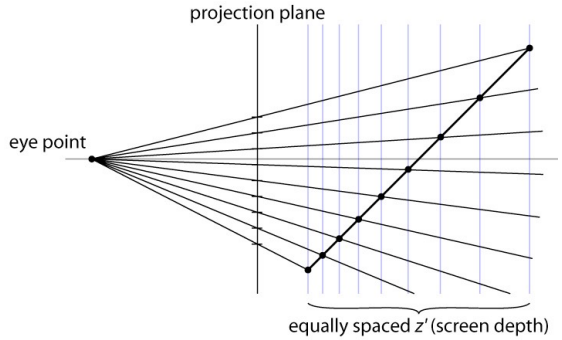
## Interpolating in projection




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



## Interpolating in projection

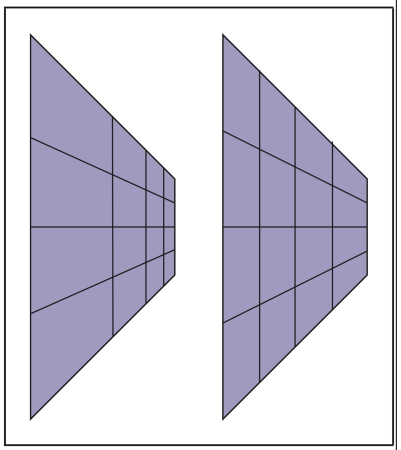



linear interp. in screen space ≠ linear interp. in world (eye) space



## Screen Space vs. World Space

- Linear variation in screen space does not yield linear variation in world space due to projection
- Consider a checkerboard at a steep angle; all squares are the same size on the plane, but not on screen



## Barycentric Interpolation in World Space

- We need to account for the homogenous rescaling in any function we interpolate:

```
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]) {
    z = α*z0/w0 + β*z1/w1 + γ*z2/w2
    s = α/w0 + β/w1 + γ/w2
    fragment.z = z/s
  }
}
```

- Section 11.2.4 of the textbook describes this in more detail

# Adding 3D Cues w/ Shading and Lighting

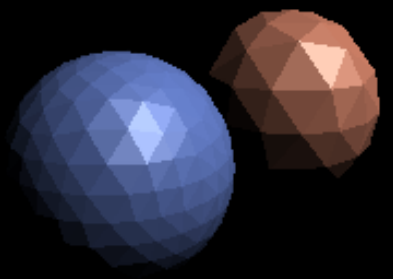
# Flat Shading

- 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

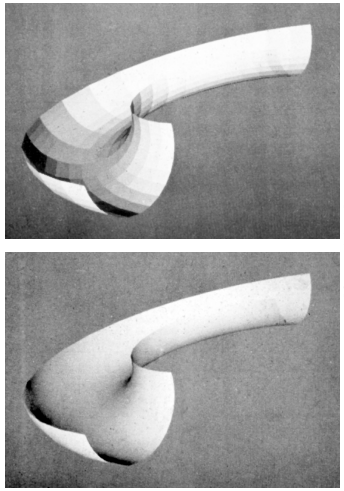
- Vertex Processing
  - Compute shaded color per triangle using triangle normal
  - Transform vertex positions
- Rasterizer
  - Interpolate the z-value for each fragment
  - Set fragment color based on its primitive color.
- Fragment Processing
  - Write fragment colors only if interpolated z is closer

# 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]



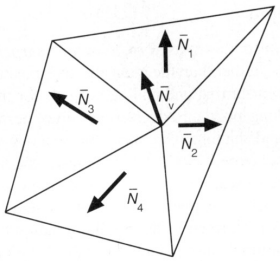
## Pipeline for Gouraud Shading

- 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

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

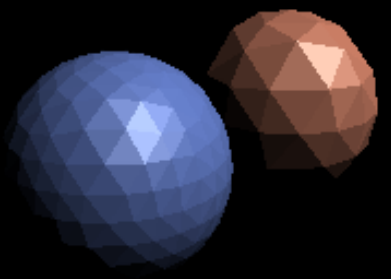



[Foley et al.]

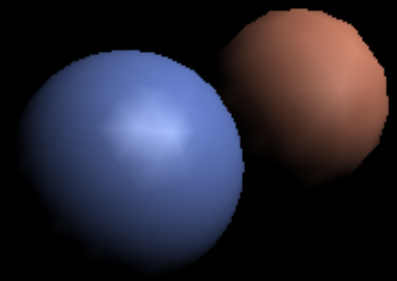


## Results of Flat Shading Pipeline



# Results of Gouraud Shading Pipeline

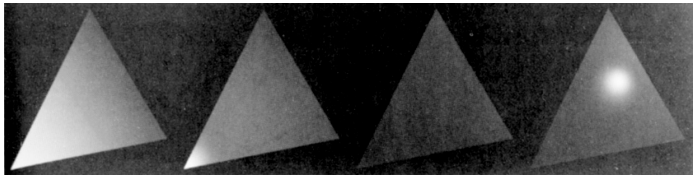


# Problems w/ Gouraud Shading

- 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

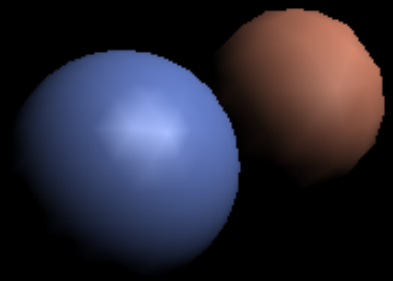


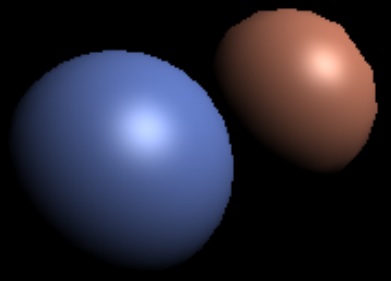
# Pipeline for Phong Shading

- 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

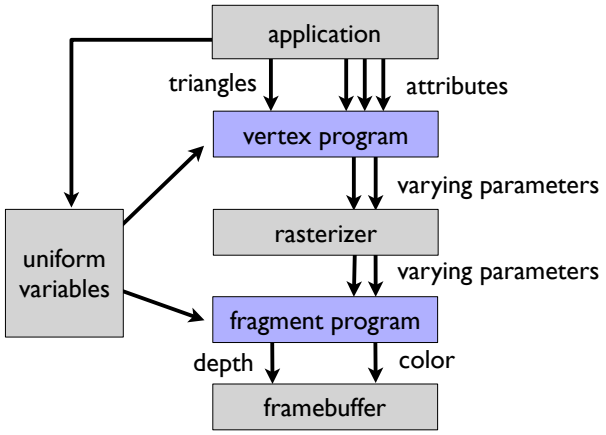


# Some Comments on Hardware

## 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






# Lec22 Required Reading

- FOCG, Ch. 11.1-11.14

# Reminder: Assignment 06

Assigned: Wednesday, Nov. 13
Written Due: Monday, Nov. 25, 4:59:59 pm
Program Due: Wednesday, Nov. 27, 4:59:59 pm