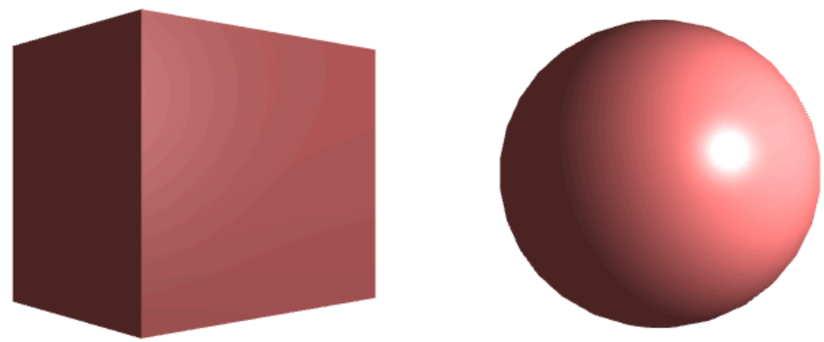


Computer Graphics

Lecture 22

The Graphics Pipeline

Announcements

- Wednesday's class will be a mini-lab. Please bring a laptop if you can!
- Other lab days coming up:
  - Line drawing: Tuesday 11/12
  - Splines: Wednesday 11/20 - Friday 11/22

~~Mid~~Lateterm Exam

- Take-home exam out Friday ~~11/4~~
  Due ~~Monday 11/7~~ Tuesday at 10pm.
- Similar to the homeworks, but no collaboration, no google, no chatgpt et al.
  - Book is ok. Writing code is ok.

Final Project

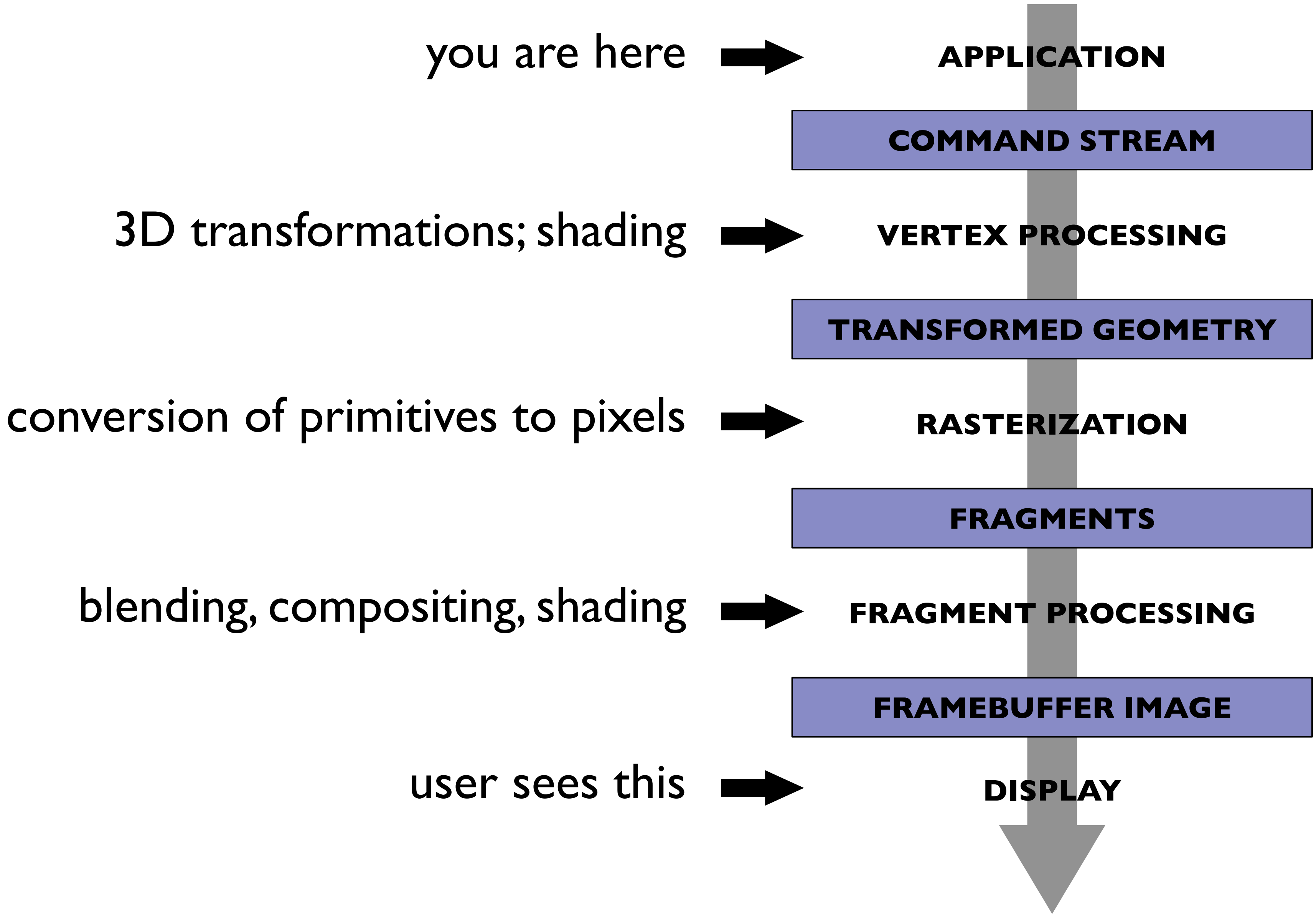
- Group formation due Wednesday
- Proposals due Friday

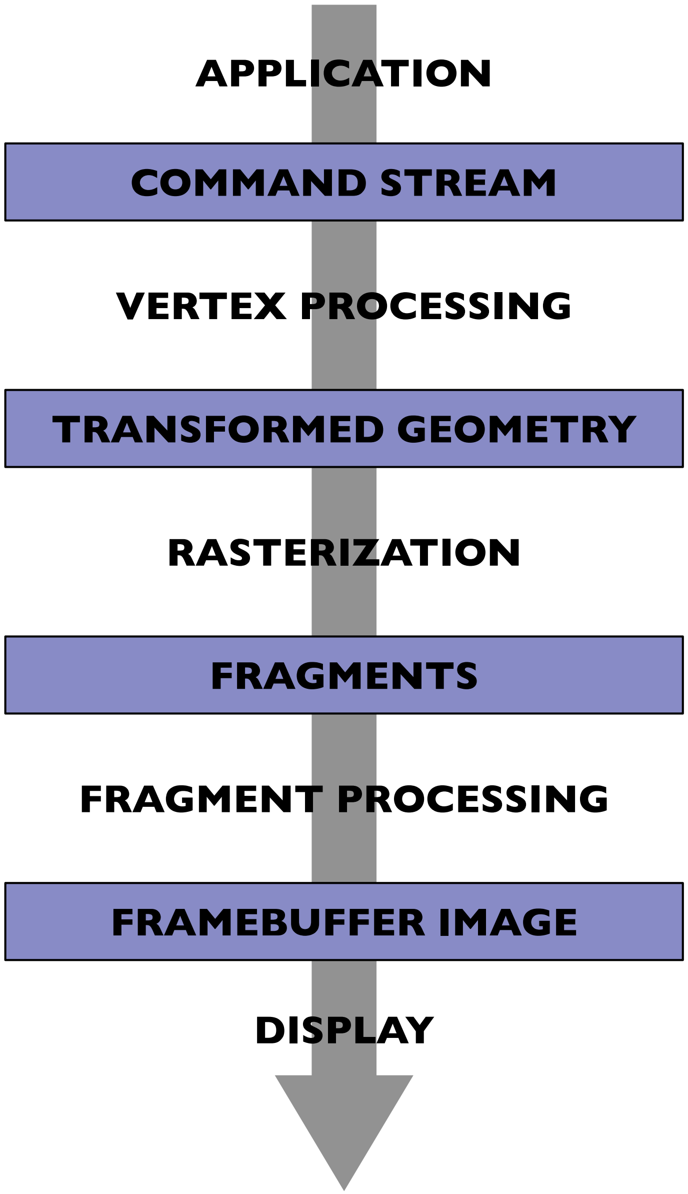
Questions?

Goals

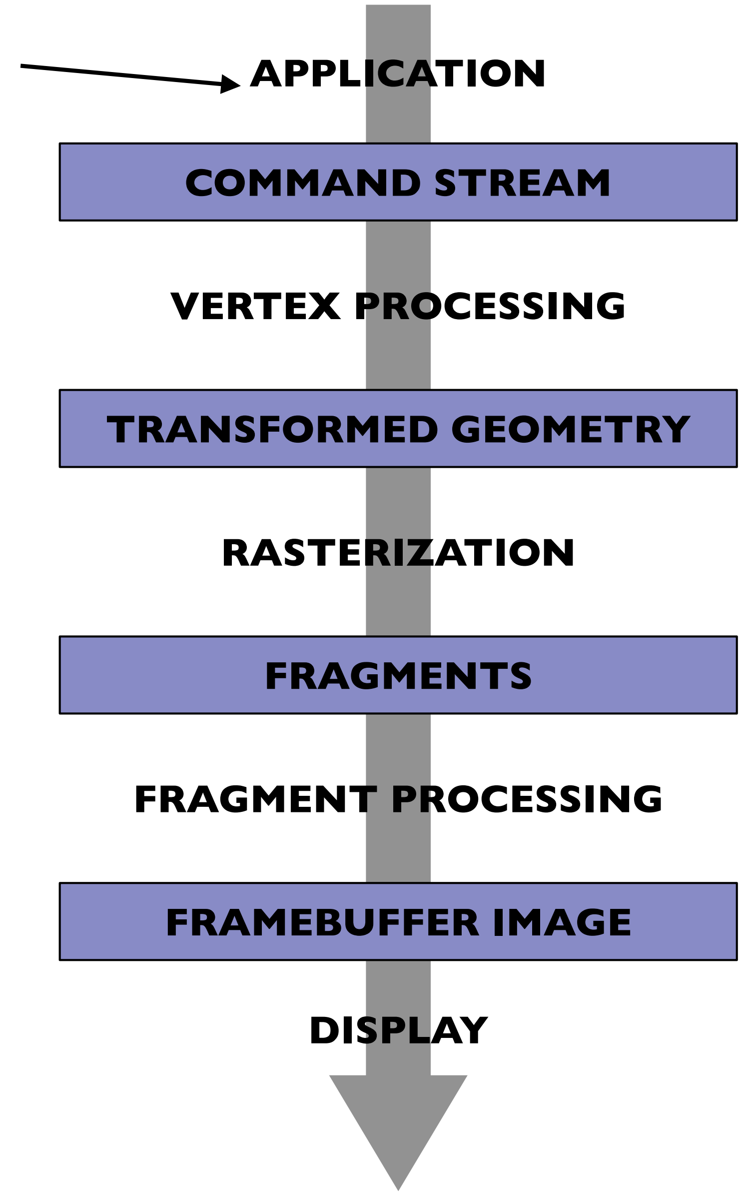
- Understand the basic phases of "The Graphics Pipeline"
- Know how to perform hidden surface removal
- Know how to use z-buffering to handle occlusion, and why this is used instead of the painter's algorithm.
- Know how the near and far planes affect z buffer precision, and why we use 1/z instead of z for interpolating.

Graphics Pipeline: Overview



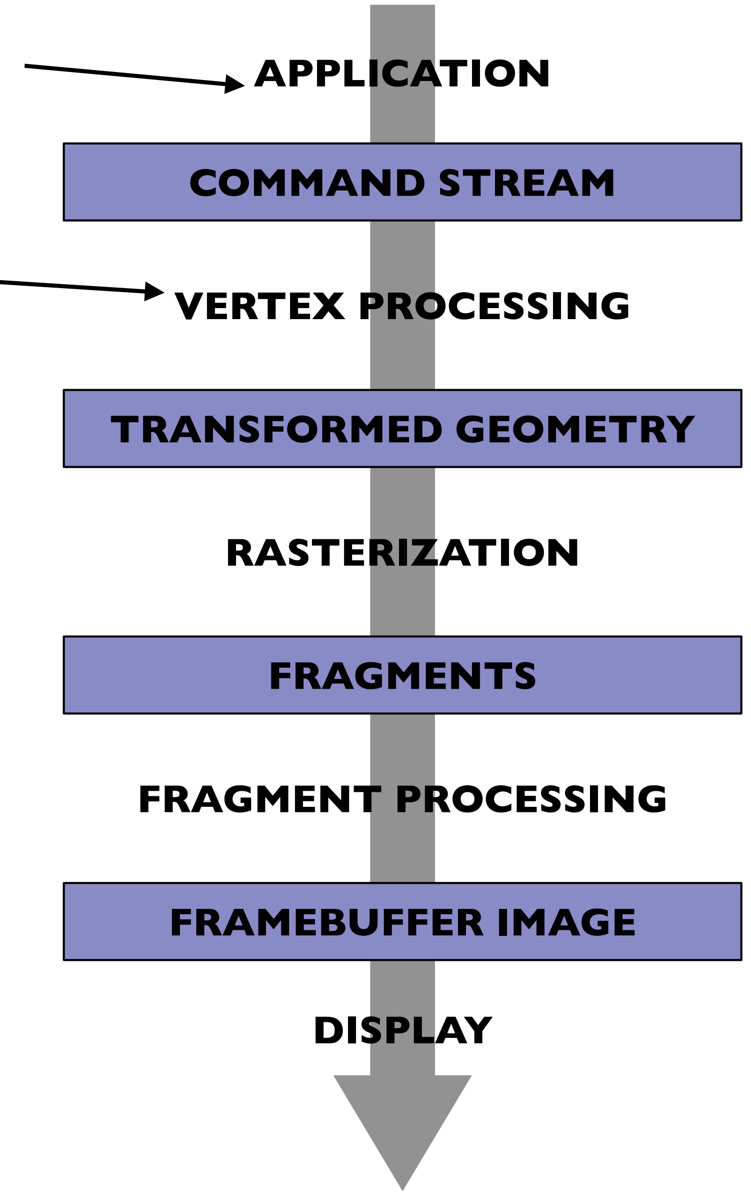

APPLICATION
COMMAND STREAM
VERTEX PROCESSING
TRANSFORMED GEOMETRY
RASTERIZATION
FRAGMENTS
FRAGMENT PROCESSING
FRAMEBUFFER IMAGE
DISPLAY

Application sends geometric primitives to renderer (e.g., to GPU)



Application sends geometric primitives to renderer (e.g., to GPU)

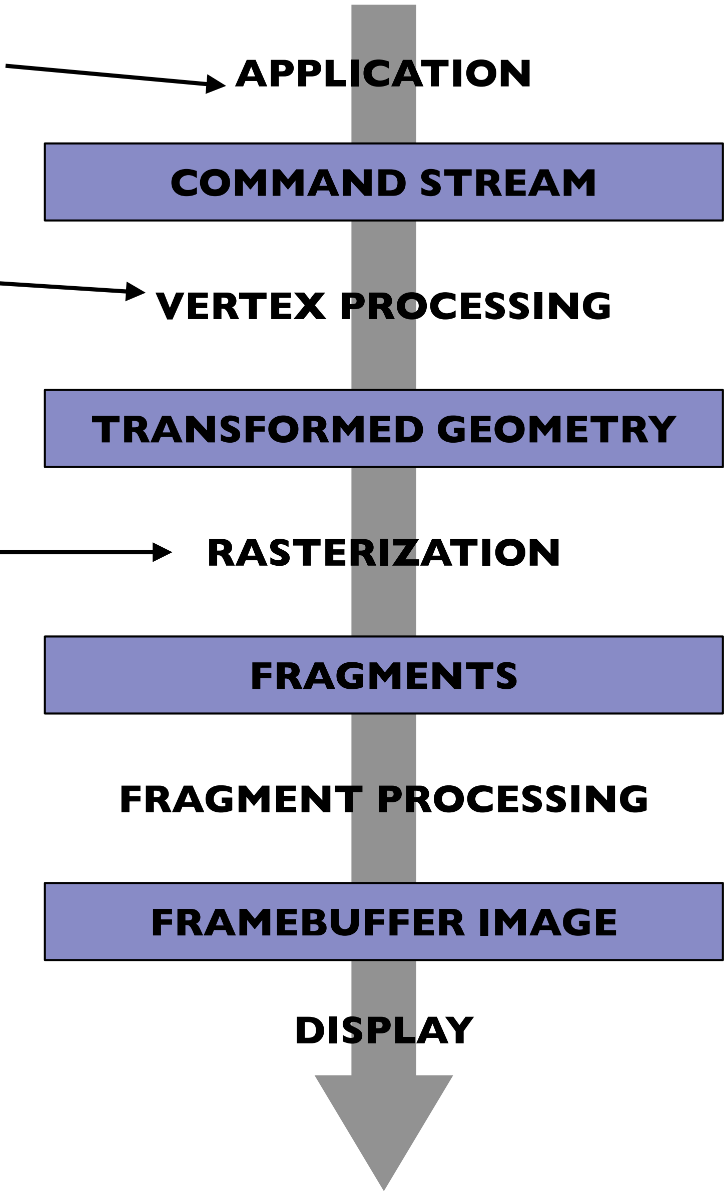
Vertices are transformed to image space (we've done lots of this!)



Application sends geometric primitives to renderer (e.g., to GPU)

Vertices are transformed to image space (we've done lots of this!)

Primitives are converted into pixel-shaped "fragments"; values are interpolated across primitives.

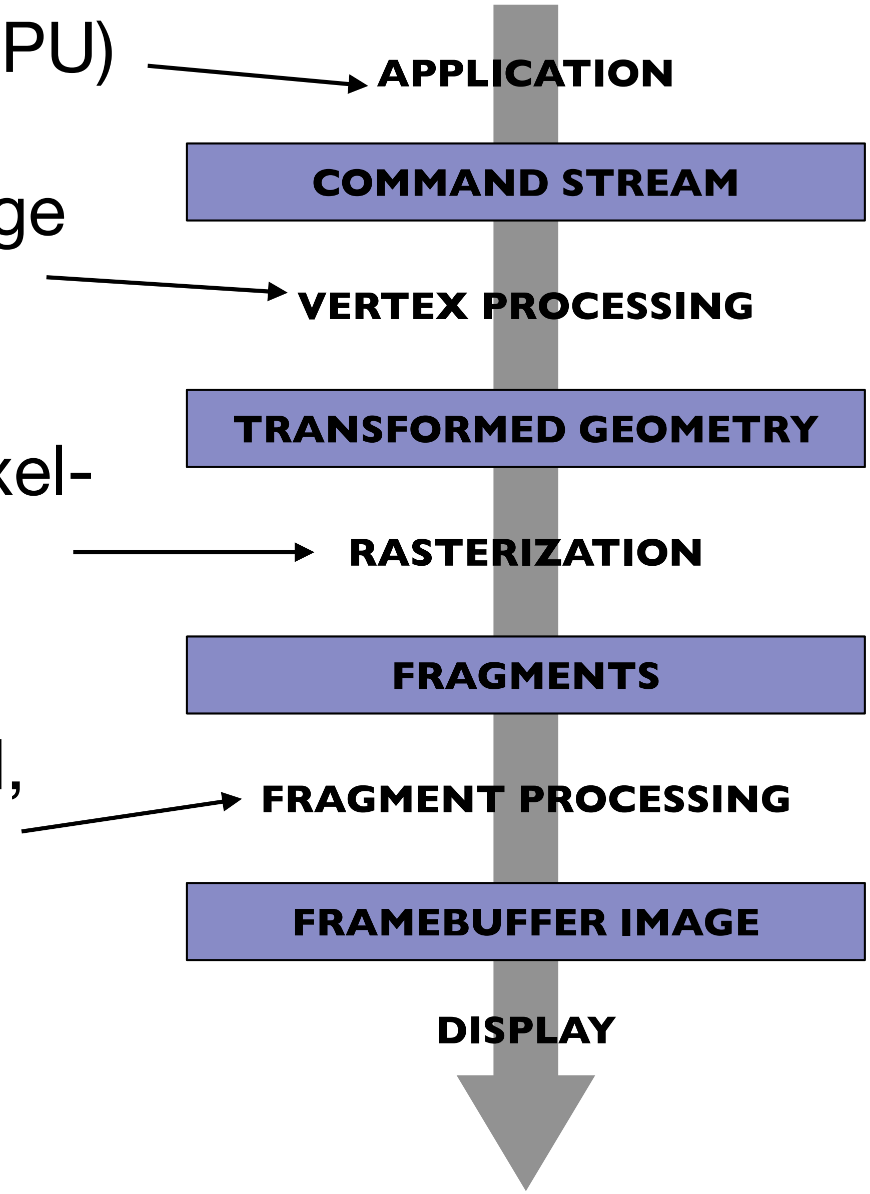


Application sends geometric primitives to renderer (e.g., to GPU)

Vertices are transformed to image space (we've done lots of this!)

Primitives are converted into pixel-shaped "fragments"; values are interpolated across primitives.

Fragments are shaded, blended, and composited to determine pixel colors.



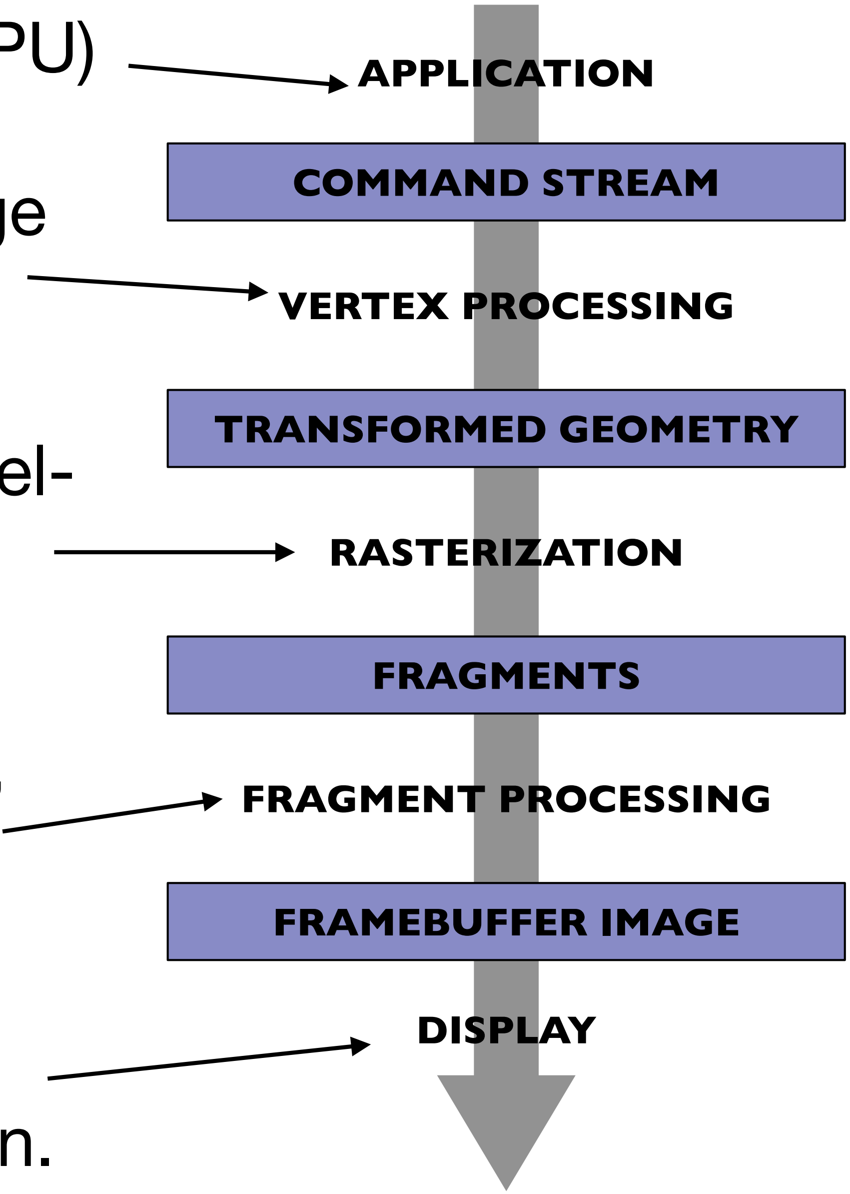
Application sends geometric primitives to renderer (e.g., to GPU)

Vertices are transformed to image space (we've done lots of this!)

Primitives are converted into pixel-shaped "fragments"; values are interpolated across primitives.

Fragments are shaded, blended, and composited to determine pixel colors.

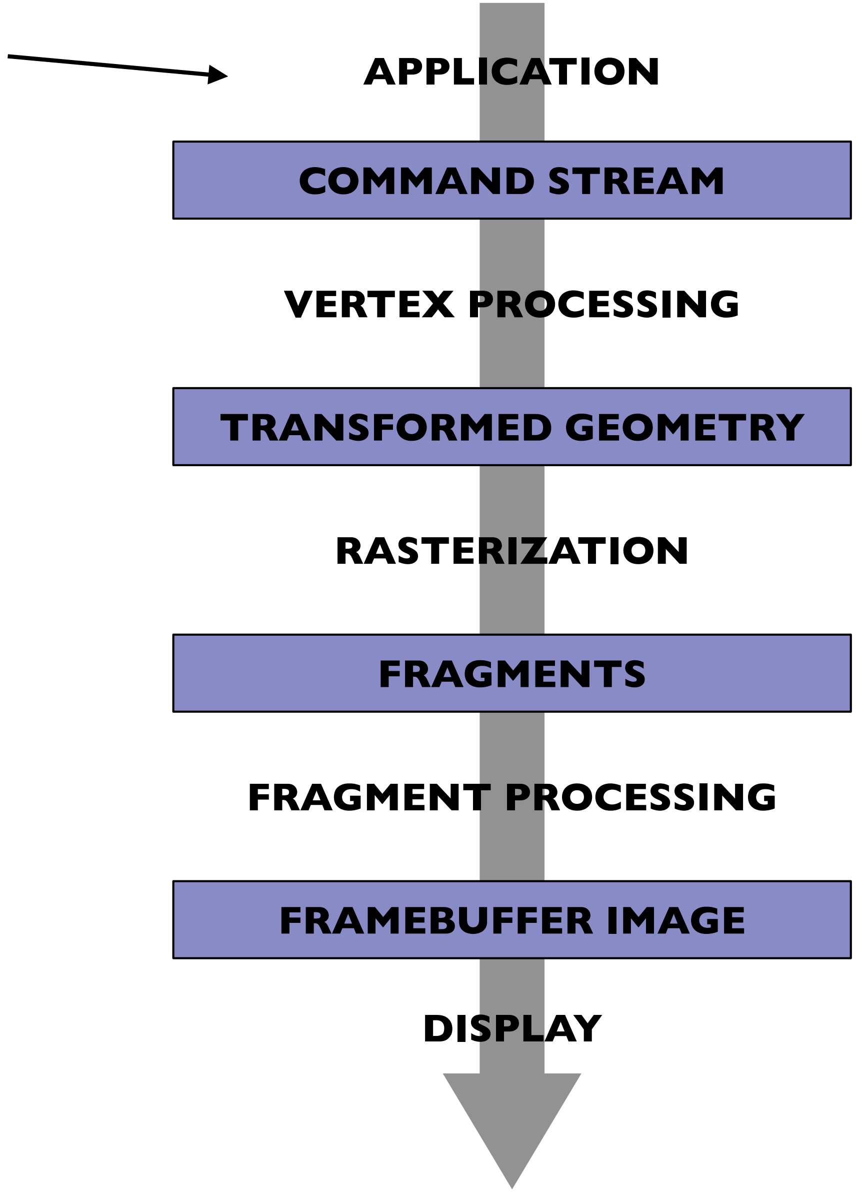
Pixel colors written to the framebuffer appear on the screen.



Command Stream

Application sends geometric primitives to renderer (e.g., to GPU)

What primitives?

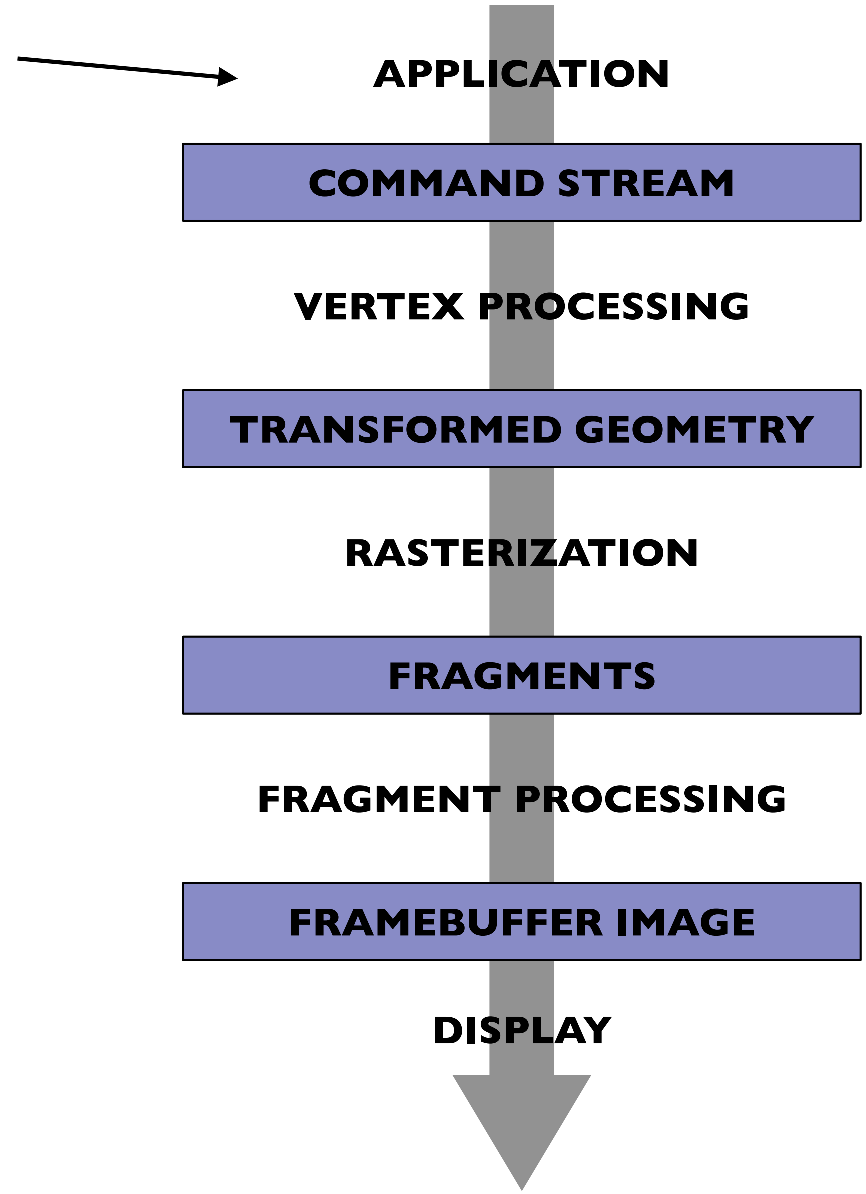


Command Stream

Application sends geometric primitives to renderer (e.g., to GPU)

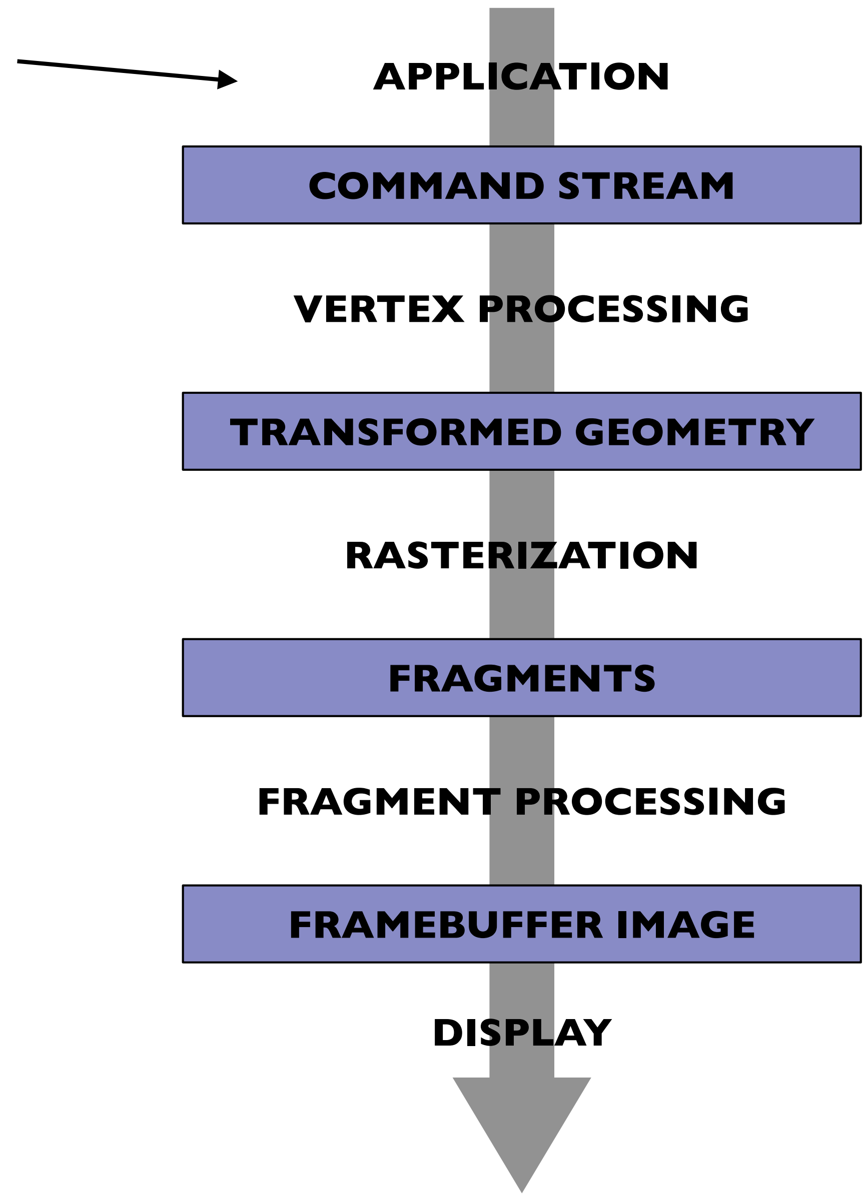
What primitives?

- Points



Command Stream

Application sends geometric primitives to renderer (e.g., to GPU)

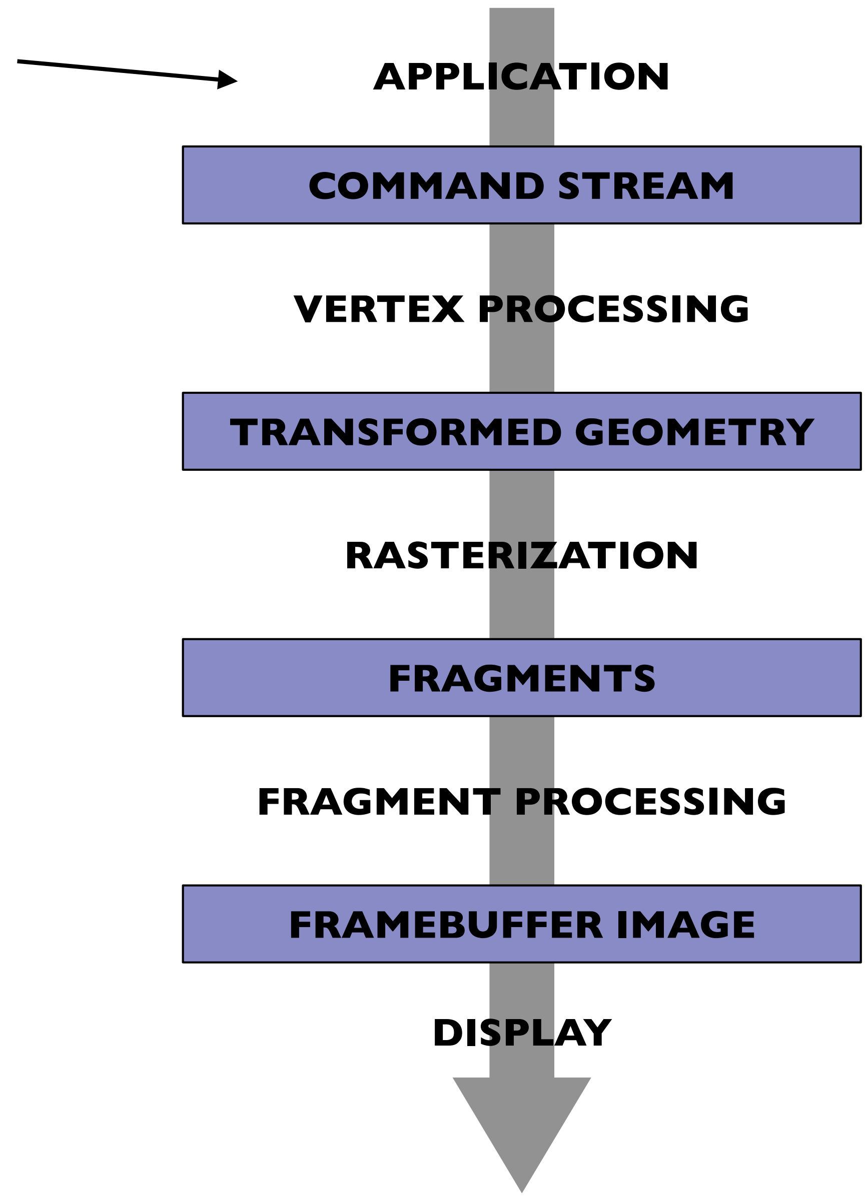


What primitives?

- Points
- Line segments
  - and chains of connected line segments

Command Stream

Application sends geometric primitives to renderer (e.g., to GPU)

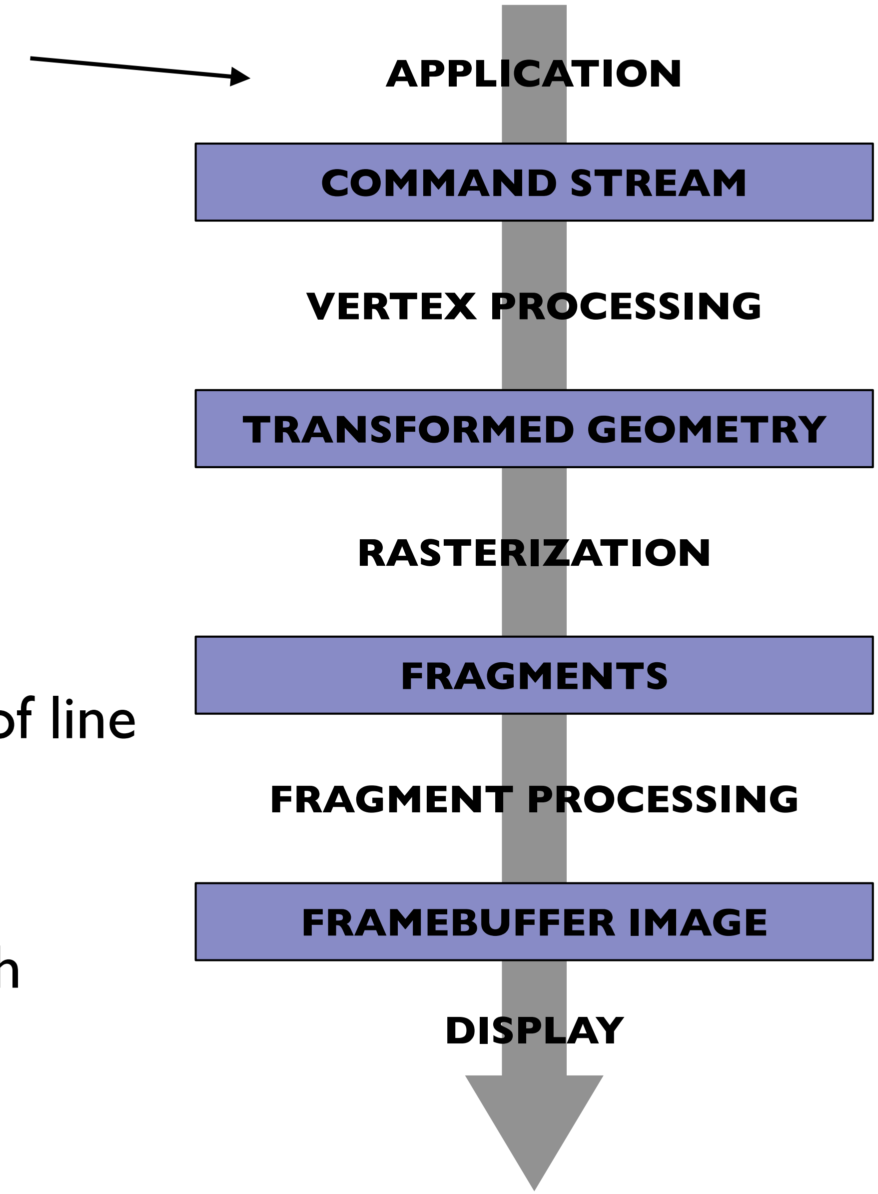


What primitives?

- Points
- Line segments
  - and chains of connected line segments
- Triangles

Command Stream

Application sends geometric primitives to renderer (e.g., to GPU)

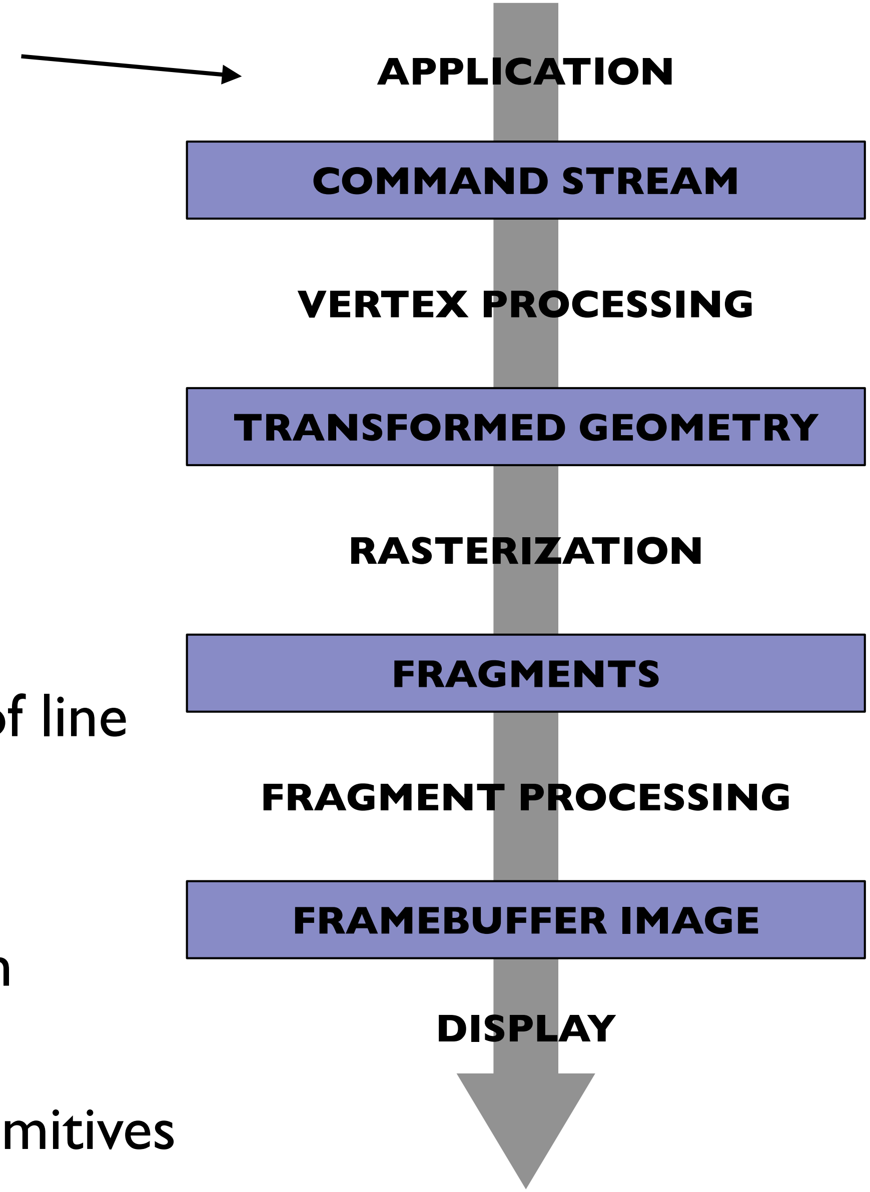


What primitives?

- Points
- Line segments
  - and chains of connected line segments
- Triangles
- And that's all!
  - Curves? Approximate them with chains of line segments
  - Polygons? Break them up into triangles
  - Curved surfaces? Approximate them with triangles

Command Stream

Application sends geometric primitives to renderer (e.g., to GPU)

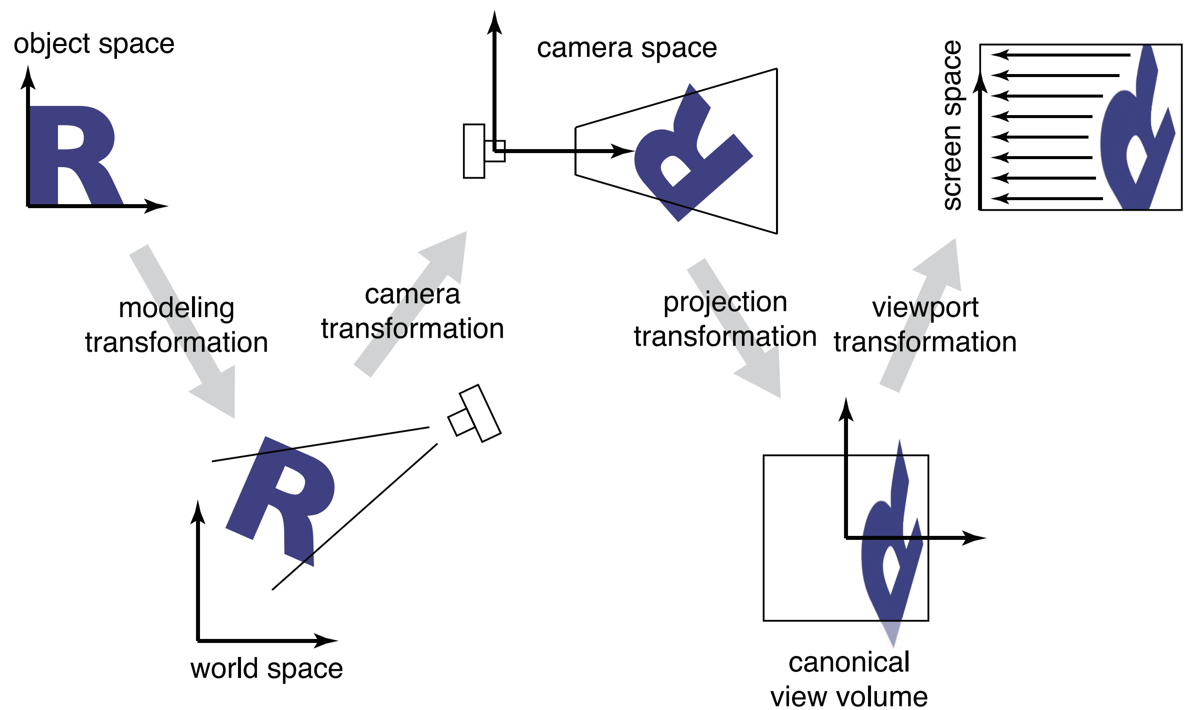


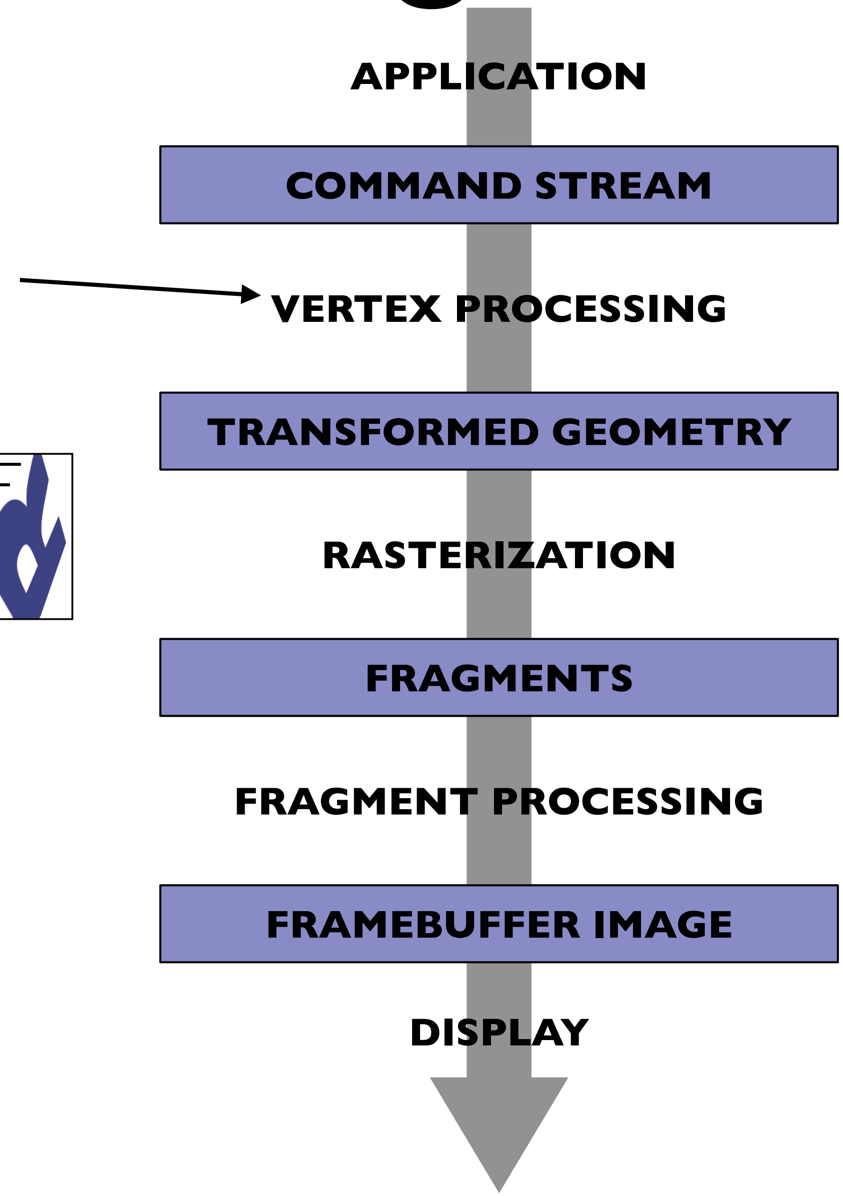
What primitives?

- Points
- Line segments
  - and chains of connected line segments
- Triangles
- And that's all!
  - Curves? Approximate them with chains of line segments
  - Polygons? Break them up into triangles
  - Curved surfaces? Approximate them with triangles
- Trend over the decades: toward minimal primitives
  - simple, uniform, repetitive: good for parallelism

Vertex Processing

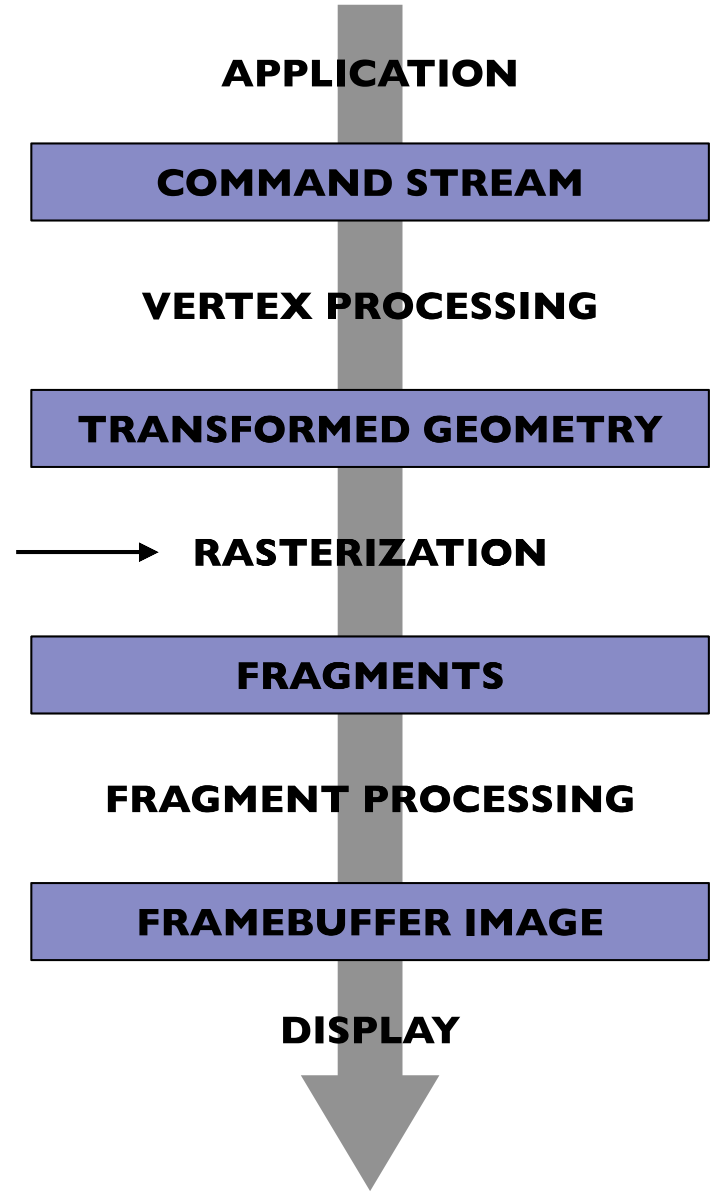
Vertices are transformed to clip space
Vertex values are computed
(we've done most of this!)





Rasterization

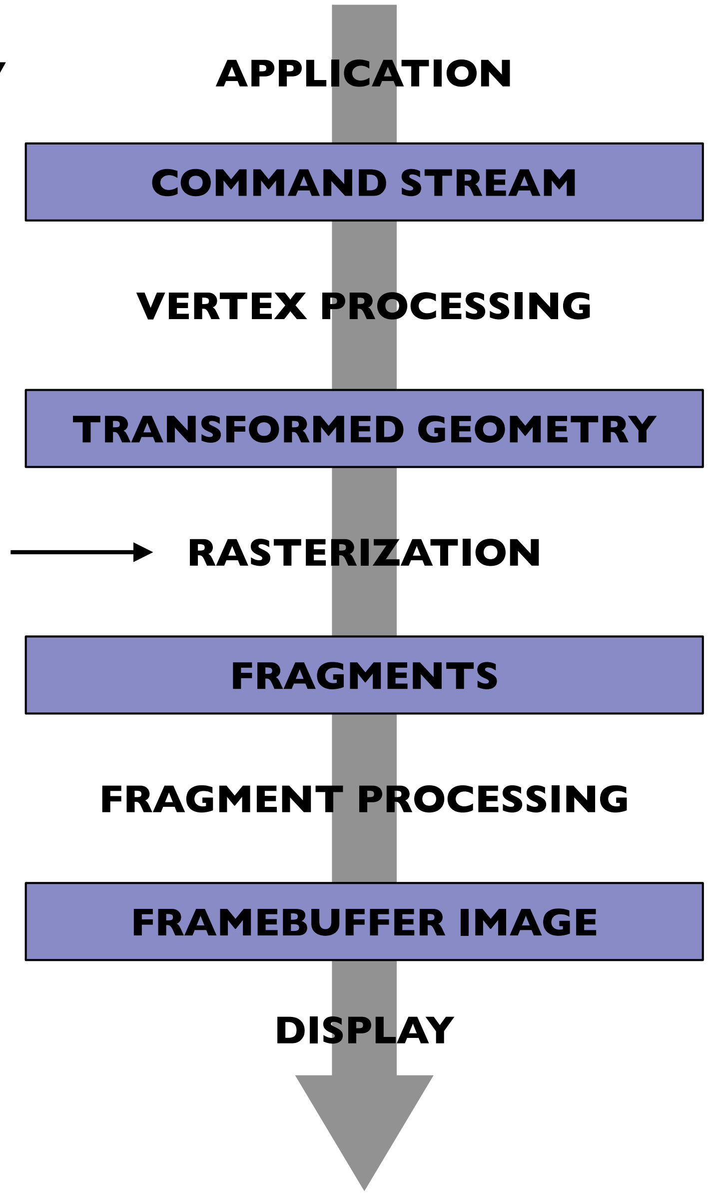
- First job: enumerate the pixels covered by a primitive
  - which pixels fall inside triangle?
  - includes "clipping" content outside view volume
- Second job: interpolate values across the primitive
  - e.g. colors computed at vertices
  - e.g. normals at vertices
  - e.g. texture coordinates



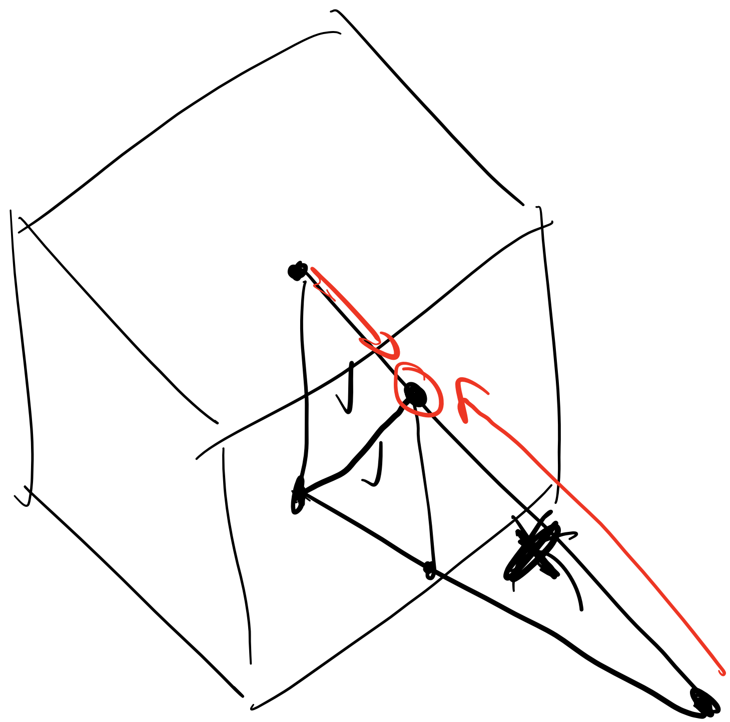
Rasterization

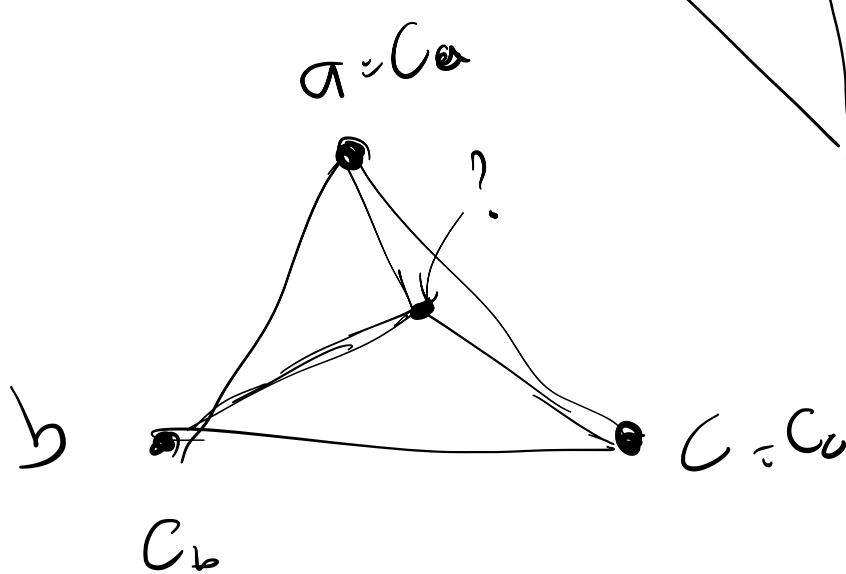
Rasterization algorithms: starting Friday

- First job: enumerate the pixels covered by a primitive
  - which pixels fall inside triangle?
  - includes "clipping" content outside view volume
- Second job: interpolate values across the primitive
  - e.g. colors computed at vertices
  - e.g. normals at vertices
  - e.g. texture coordinates



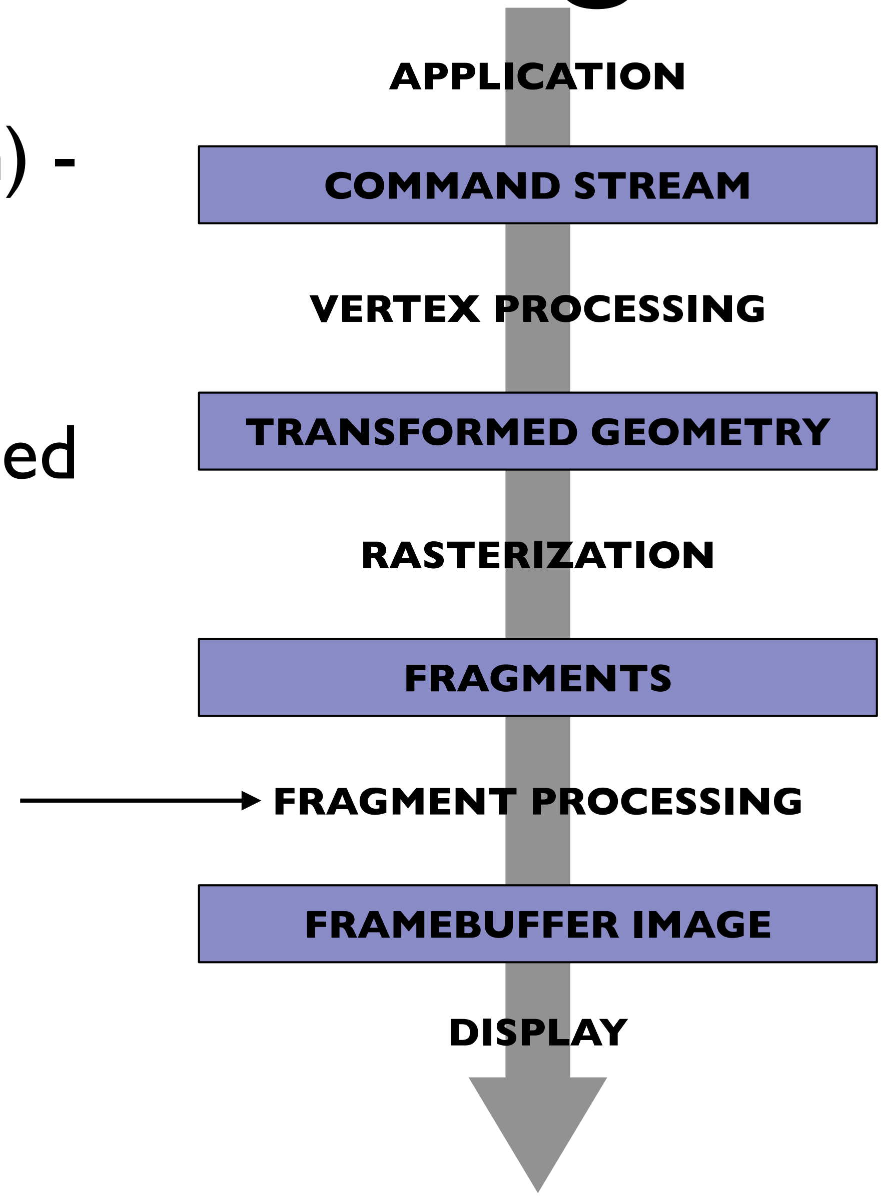
View volume clipping





Fragment Processing

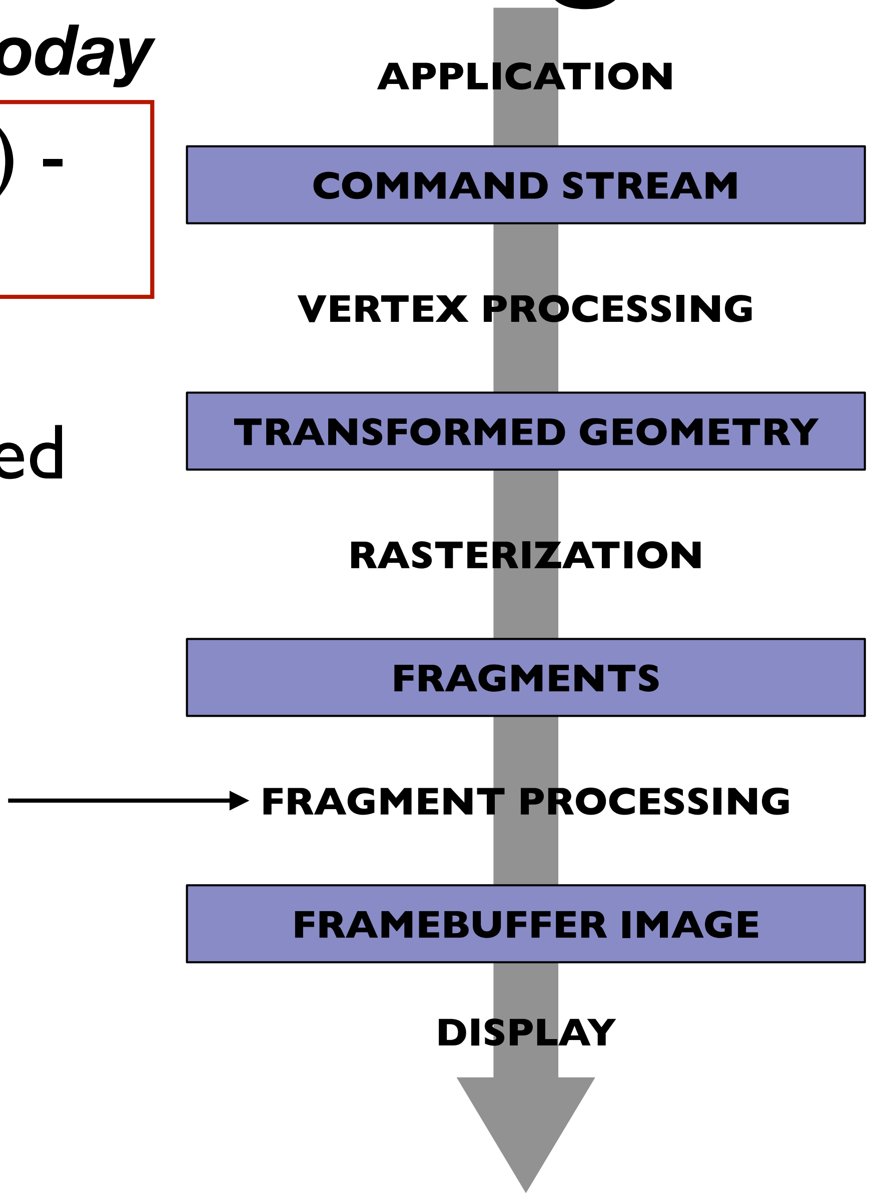
- Hidden surface removal (occlusion) - only the closest object is drawn
- Per-fragment shading:
  - determine color of the pixel based on a shading model
  - diffuse color might come from a texture
- Blending, compositing - e.g.:
  - anti-aliasing
  - transparency / alpha blending



Fragment Processing

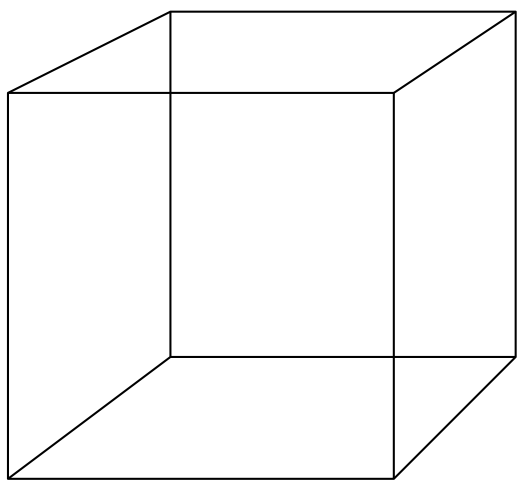
Painter's algorithm; Z buffering: today

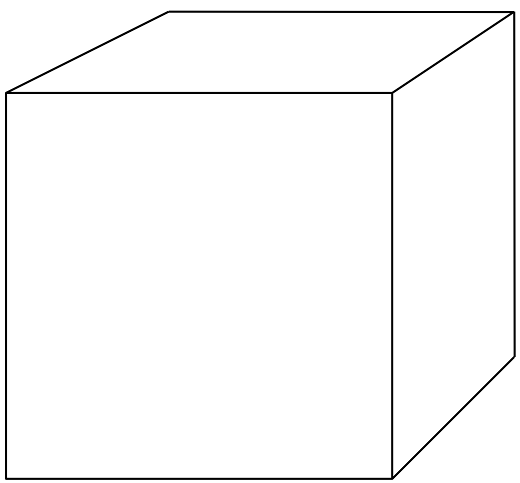
- Hidden surface removal (occlusion) - only the closest object is drawn
- Per-fragment shading:
  - determine color of the pixel based on a shading model
  - diffuse color might come from a texture
- Blending, compositing - e.g.:
  - anti-aliasing
  - transparency / alpha blending



Hidden Surface Removal

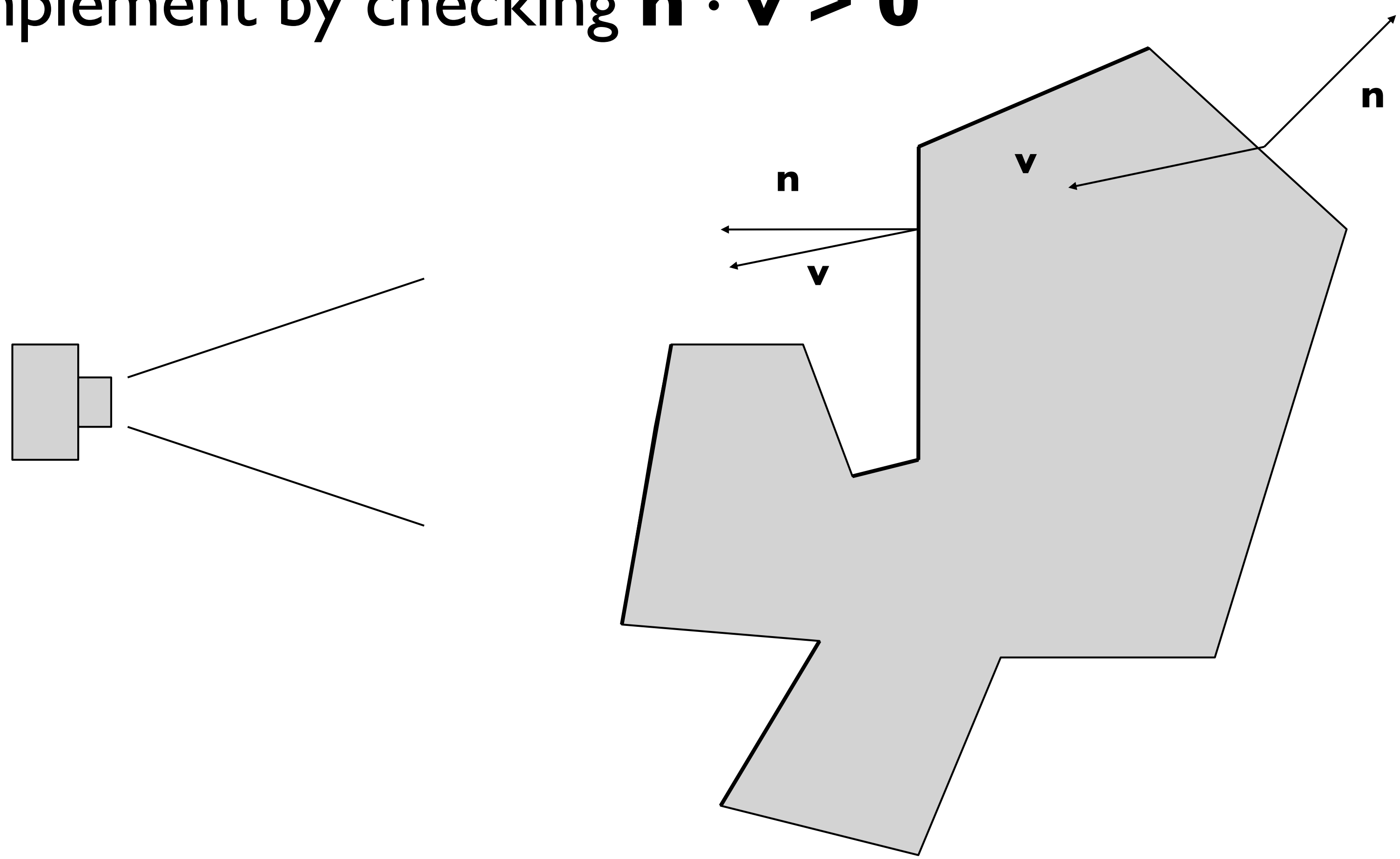
Two motivations: realism **and** efficiency





Back face culling

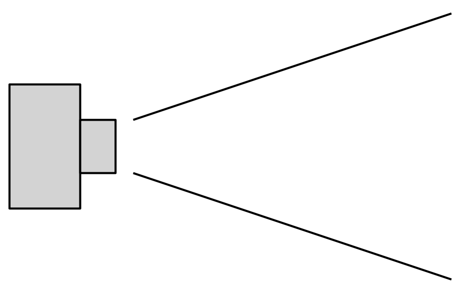
- For closed shapes you will never see the inside
  - therefore only draw surfaces that face the camera
  - implement by checking **n · v > 0**

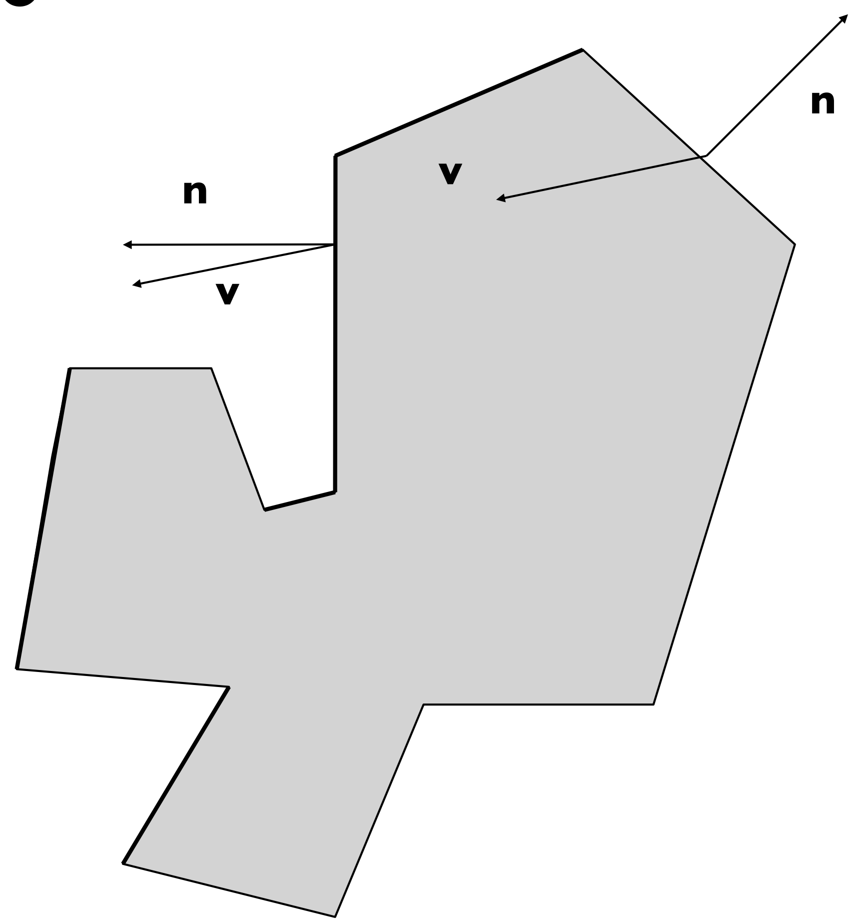


Back face culling

- For closed shapes you will never see the inside
  - therefore only draw surfaces that face the camera
  - implement by checking **n · v > 0**

Q: In which space would you prefer to do backface culling?





A: Model
B: World
C: Camera
D: Clip (/NDC/CVV)

Handling Occlusion

- What if multiple triangles are facing the viewer at different depths?

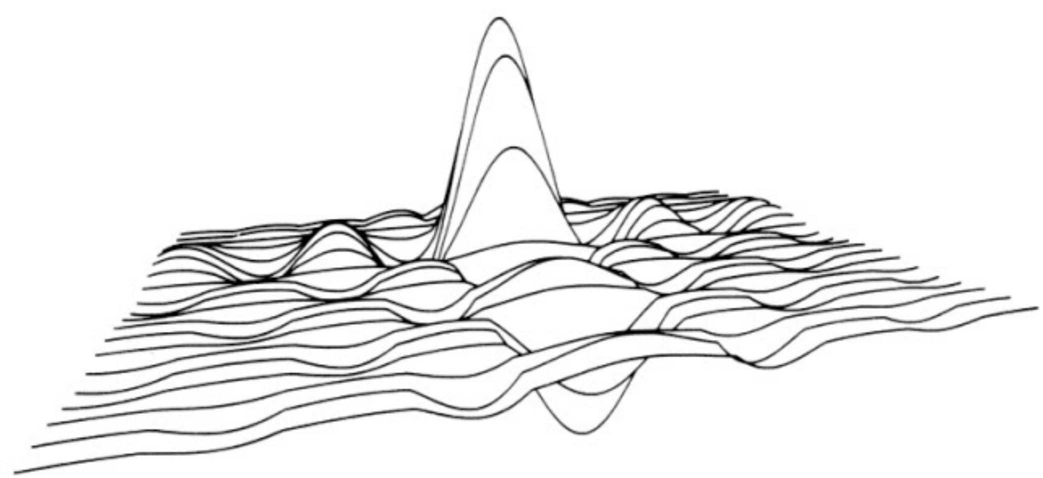
How would you deal with this?

Handling Occlusion

- What if multiple triangles are facing the viewer at different depths?
- **Painter's algorithm:** draw them back-to-front
- Topological sort on the occlusion graph:
  - if A ever occludes B, it must come after B in the drawing order

Handling Occlusion

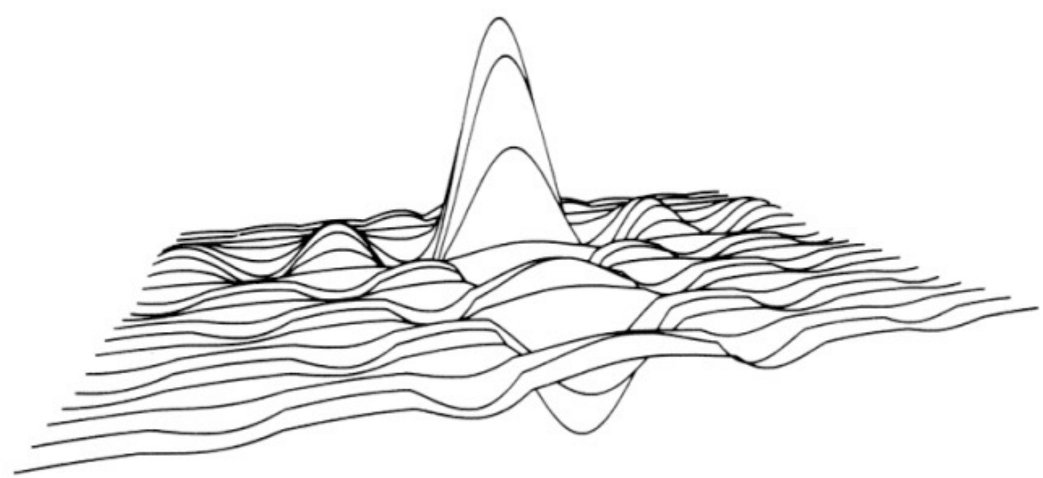
- What if multiple triangles are facing the viewer at different depths?
- **Painter's algorithm:** draw them back-to-front
- Topological sort on the occlusion graph:
  - if A ever occludes B, it must come after B in the drawing order



Handling Occlusion

- What if multiple triangles are facing the viewer at different depths?
- **Painter's algorithm:** draw them back-to-front
- Topological sort on the occlusion graph:
    - if A ever occludes B, it must come after B in the drawing order

Works great if the ordering is easy to find...

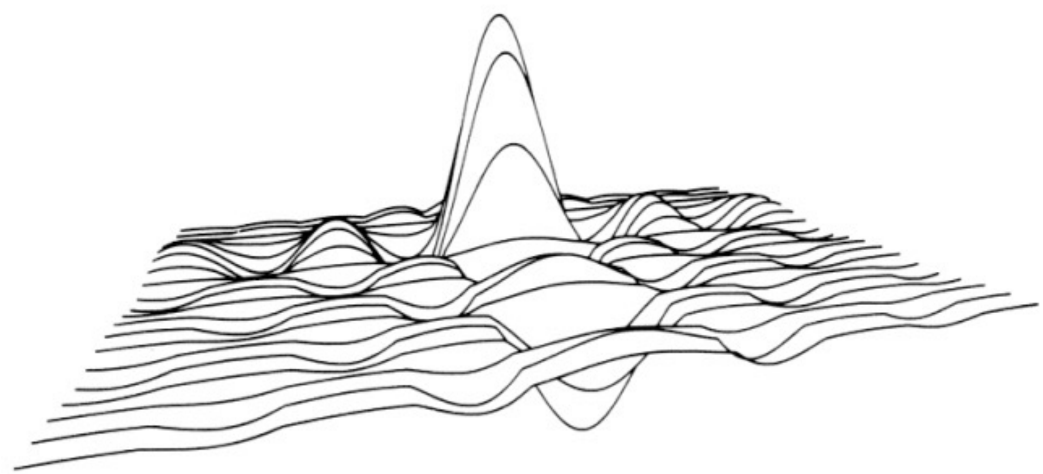


Handling Occlusion

- What if multiple triangles are facing the viewer at different depths?
- **Painter's algorithm:** draw them back-to-front
- Topological sort on the occlusion graph:
    - if A ever occludes B, it must come after B in the drawing order

Works great if the ordering is easy to find...

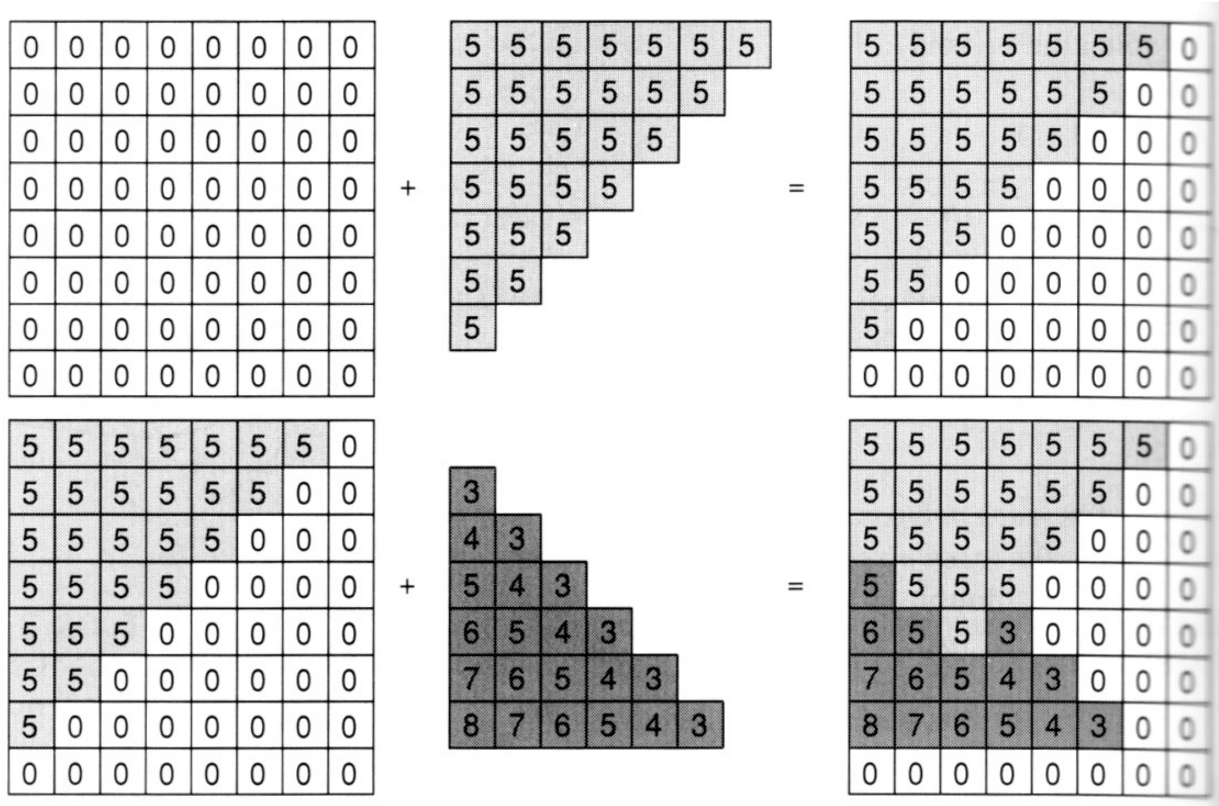
... but often it isn't.
Example: z.obj



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



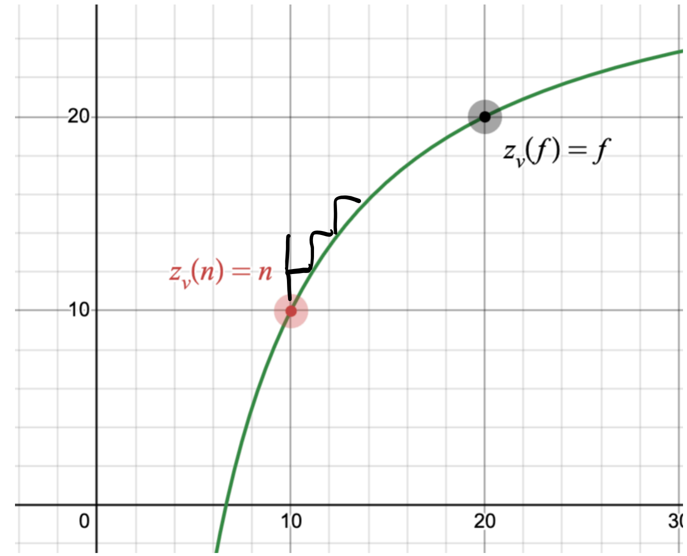
[Foley et al.]

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

– store z as an integer for speed and memory efficiency (at the expense of precision!)

Precision in z buffer: Throwback

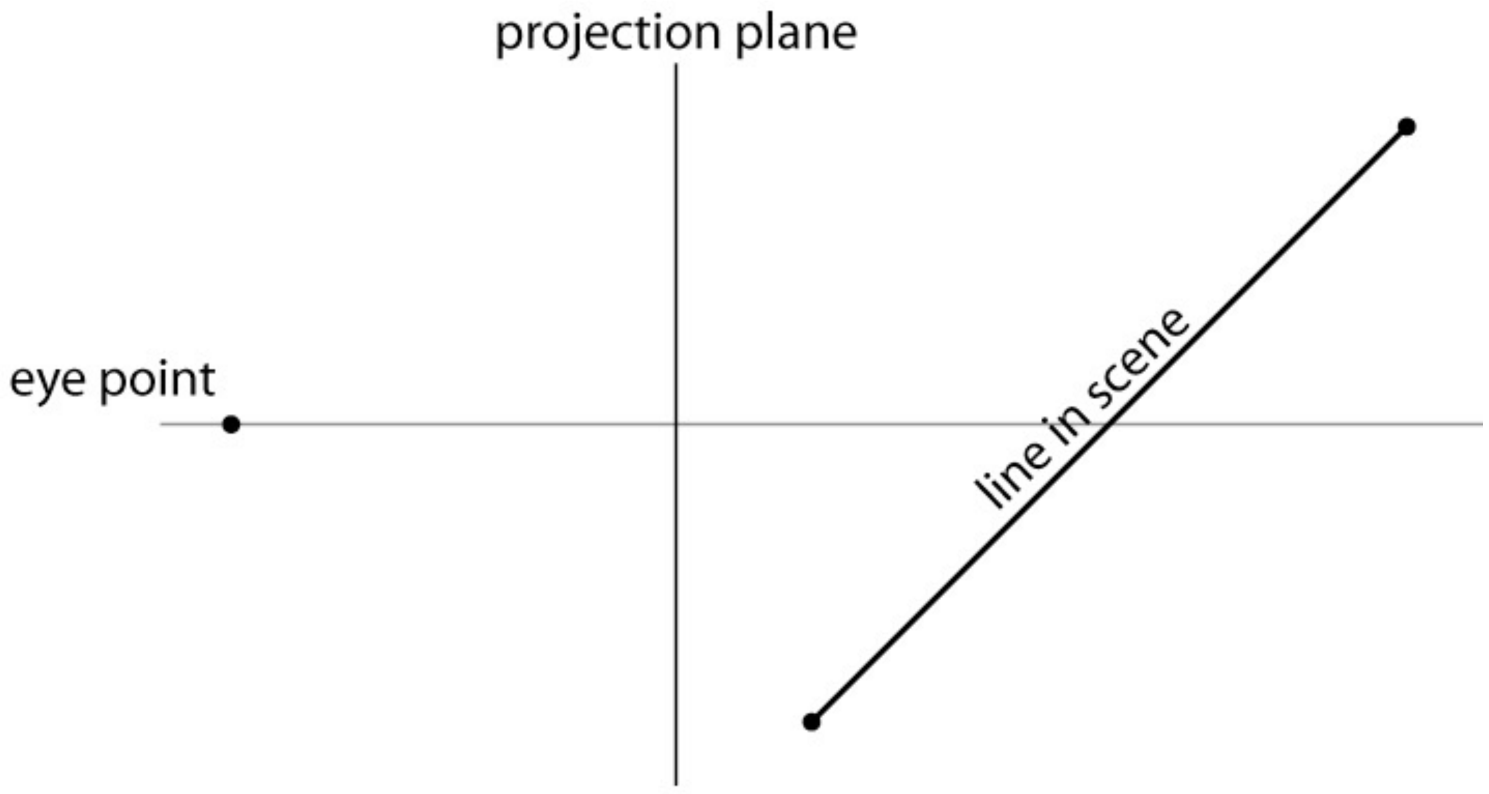
$$\mathbf{P} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}$$



Desmos Demo

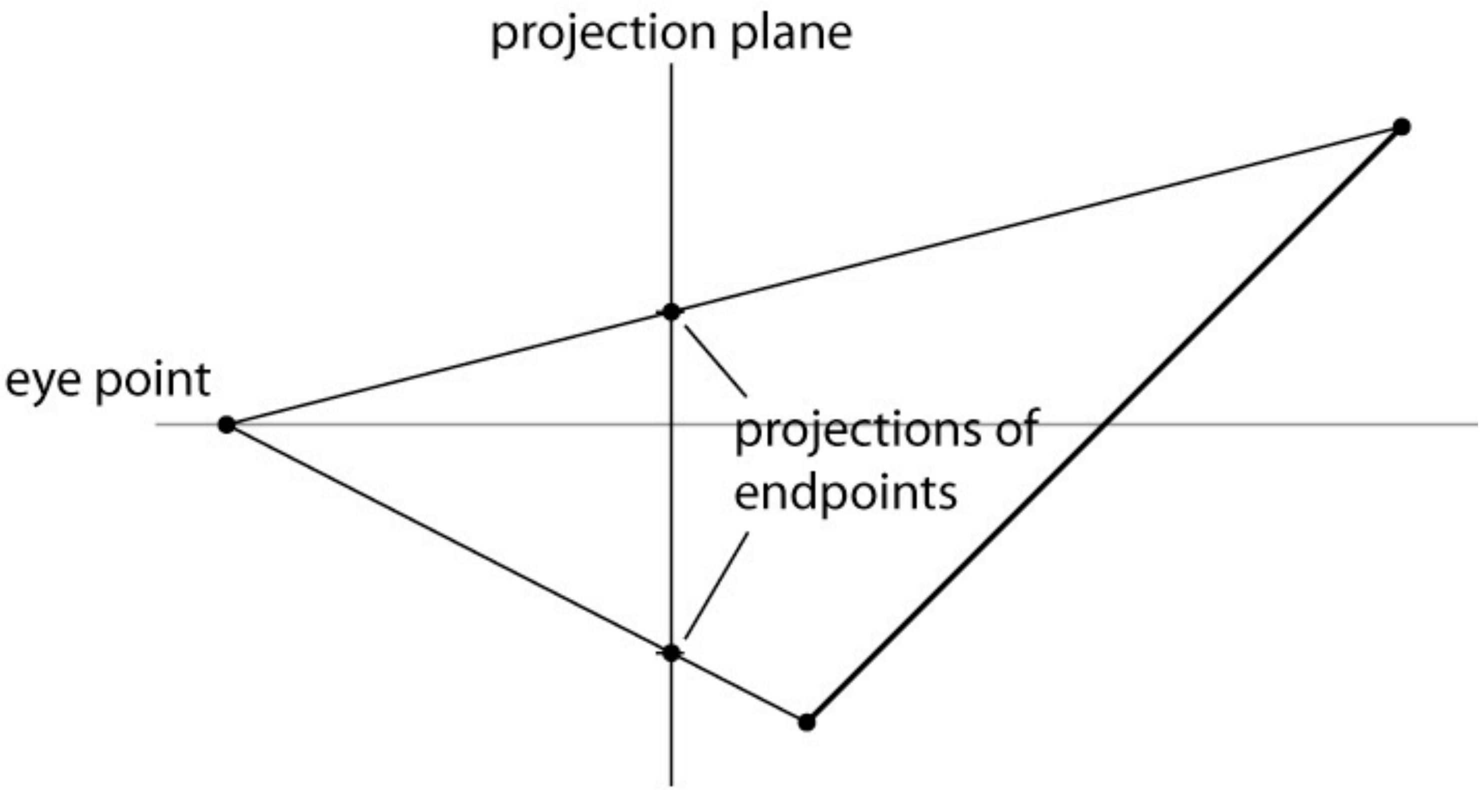
- The precision is distributed between the near and far clipping planes
  - this is why these planes have to exist
  - also why you can't always just set them to very small and very large distances
- Generally use z' (not world z) in z buffer, however...

Interpolating in projection



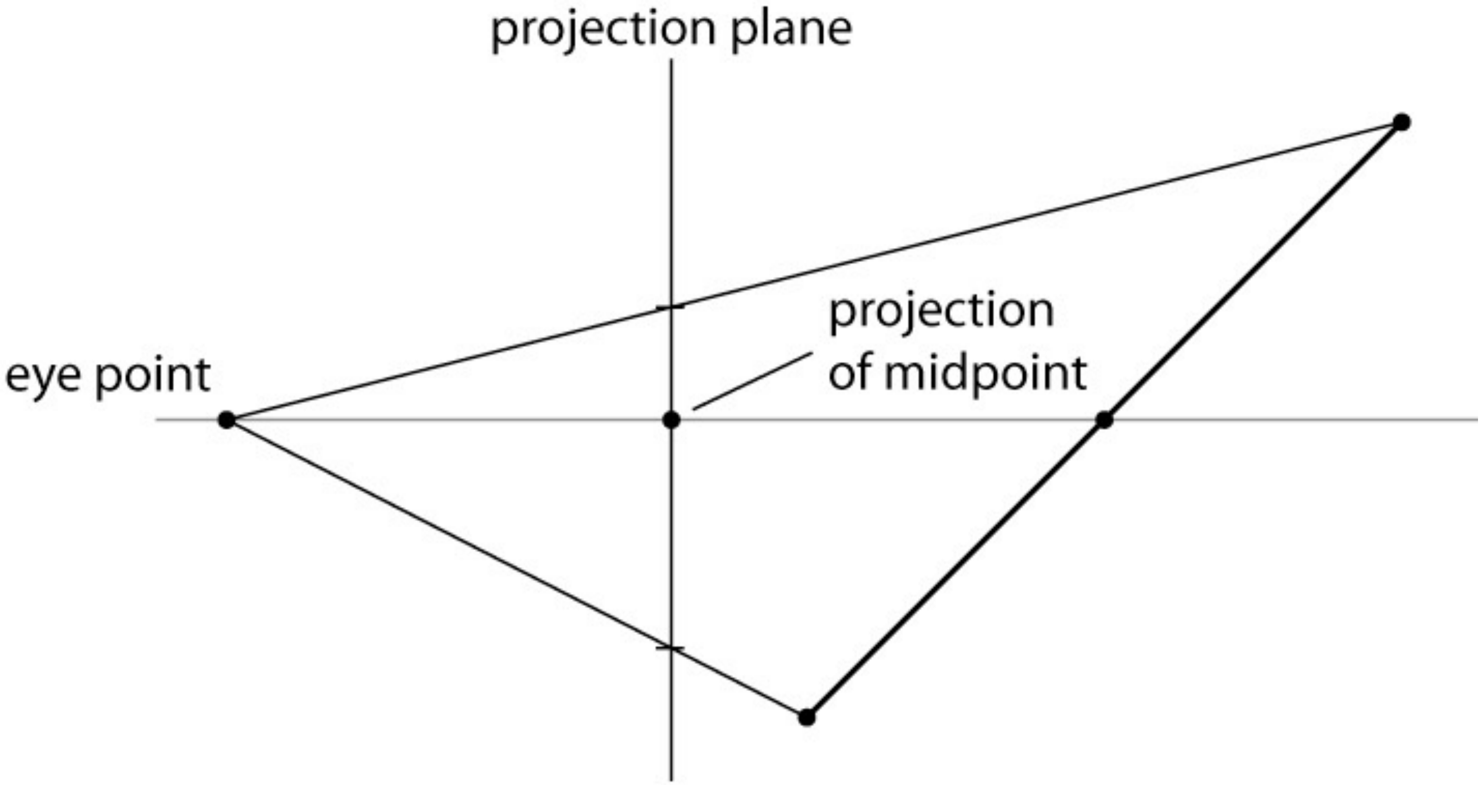
linear interp. in screen space ≠ linear interp. in world (eye) space

Interpolating in projection



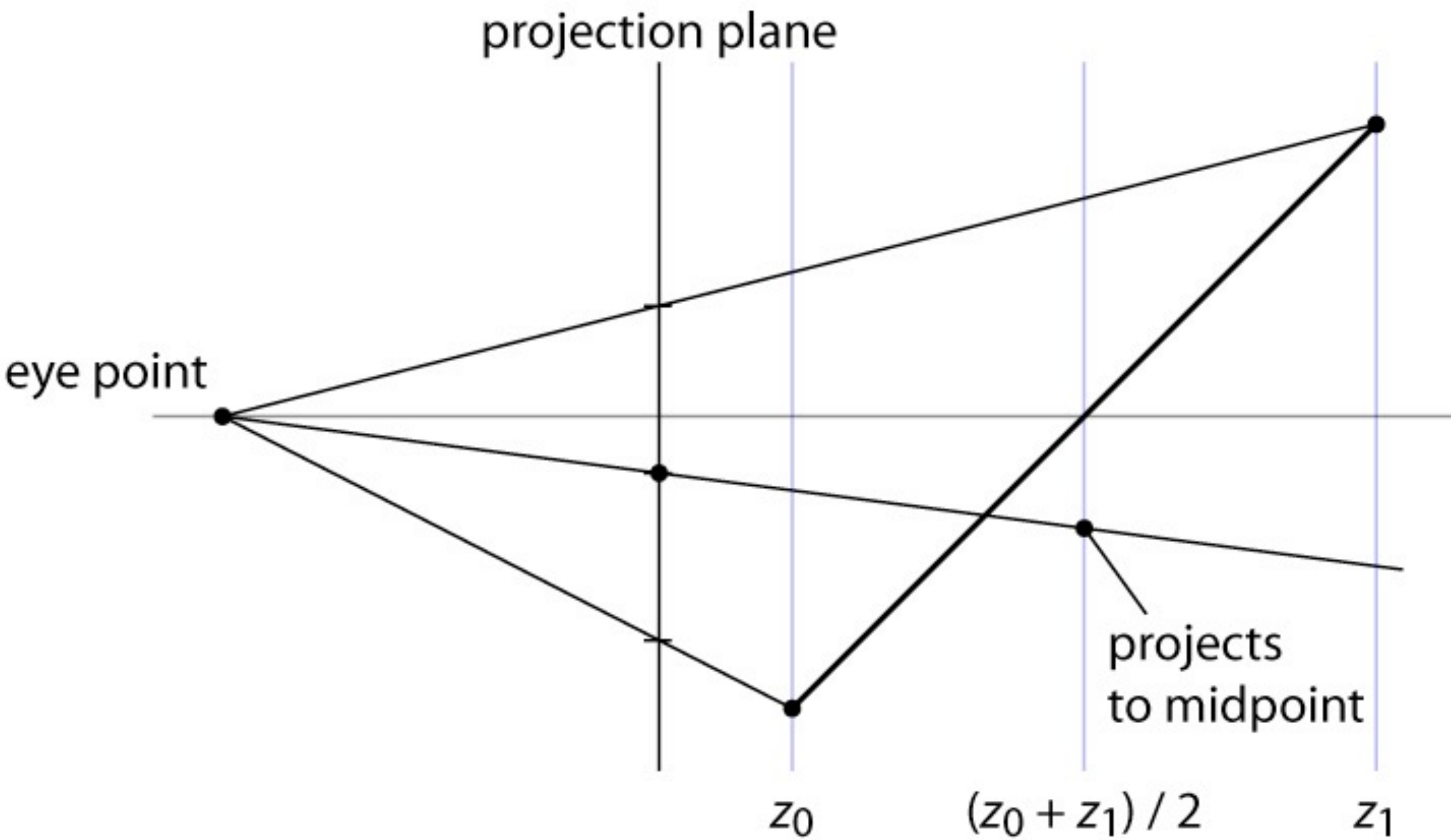
linear interp. in screen space ≠ linear interp. in world (eye) space

Interpolating in projection



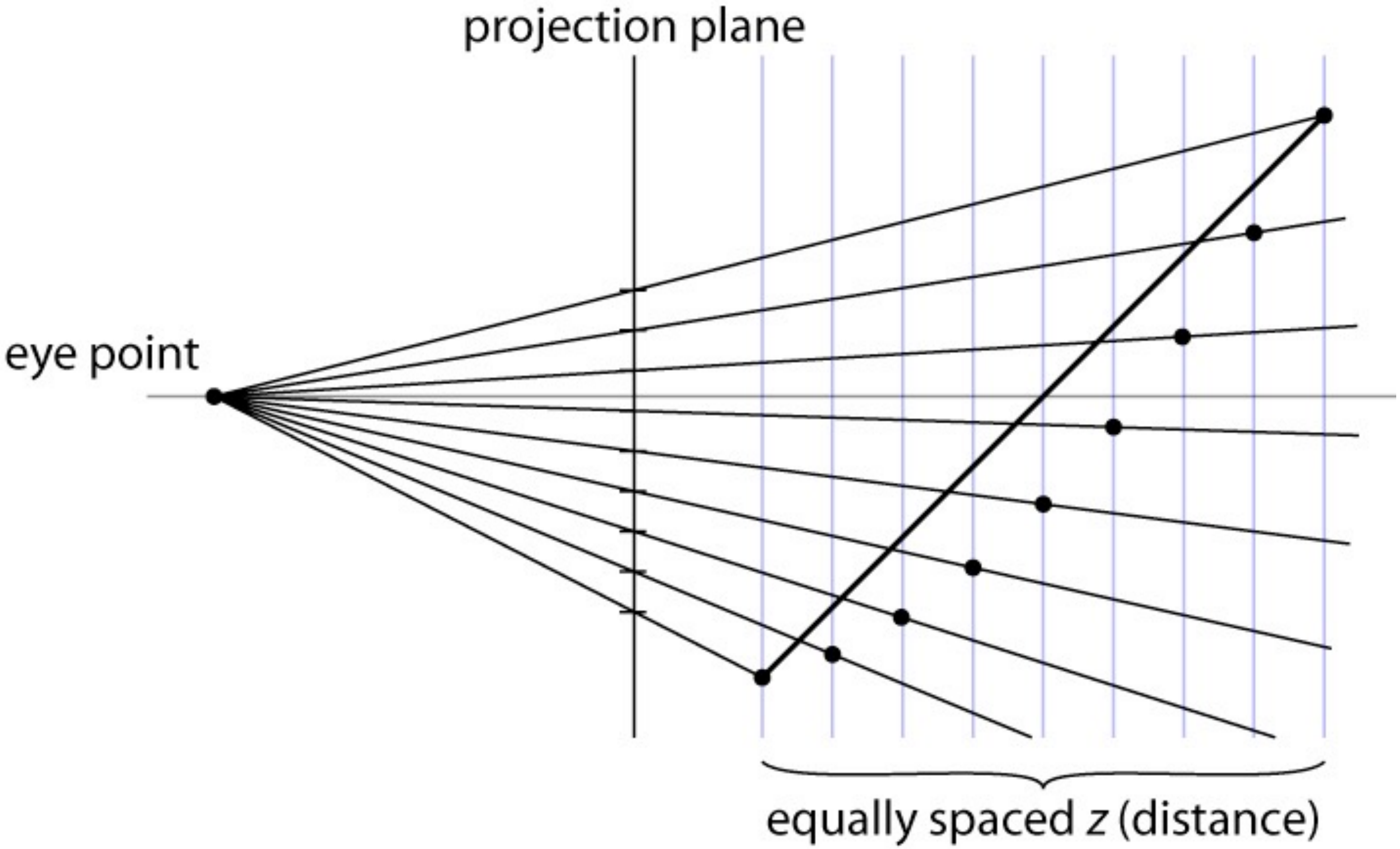
linear interp. in screen space ≠ linear interp. in world (eye) space

Interpolating in projection



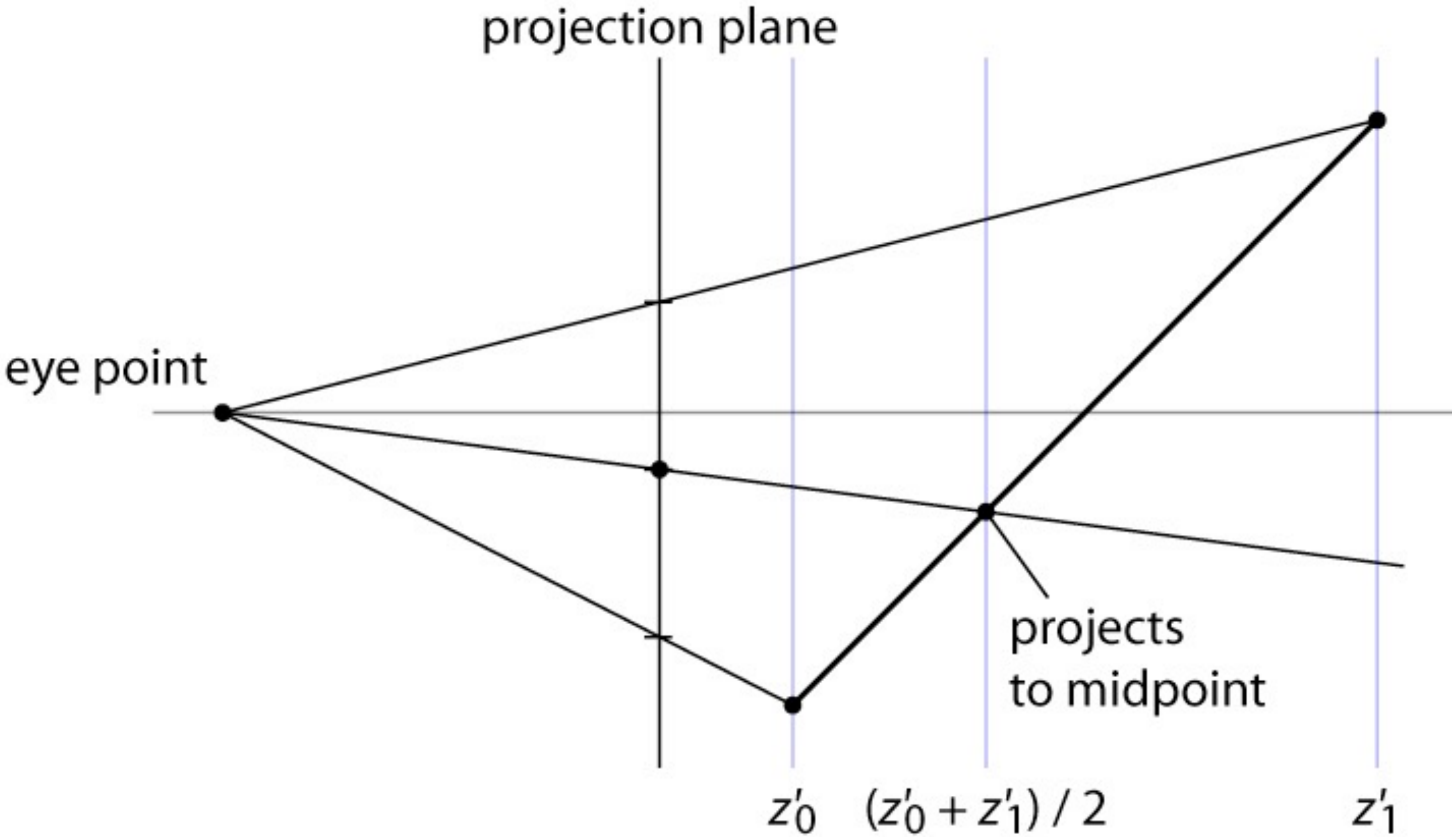
linear interp. in screen space ≠ linear interp. in world (eye) space

Interpolating in projection



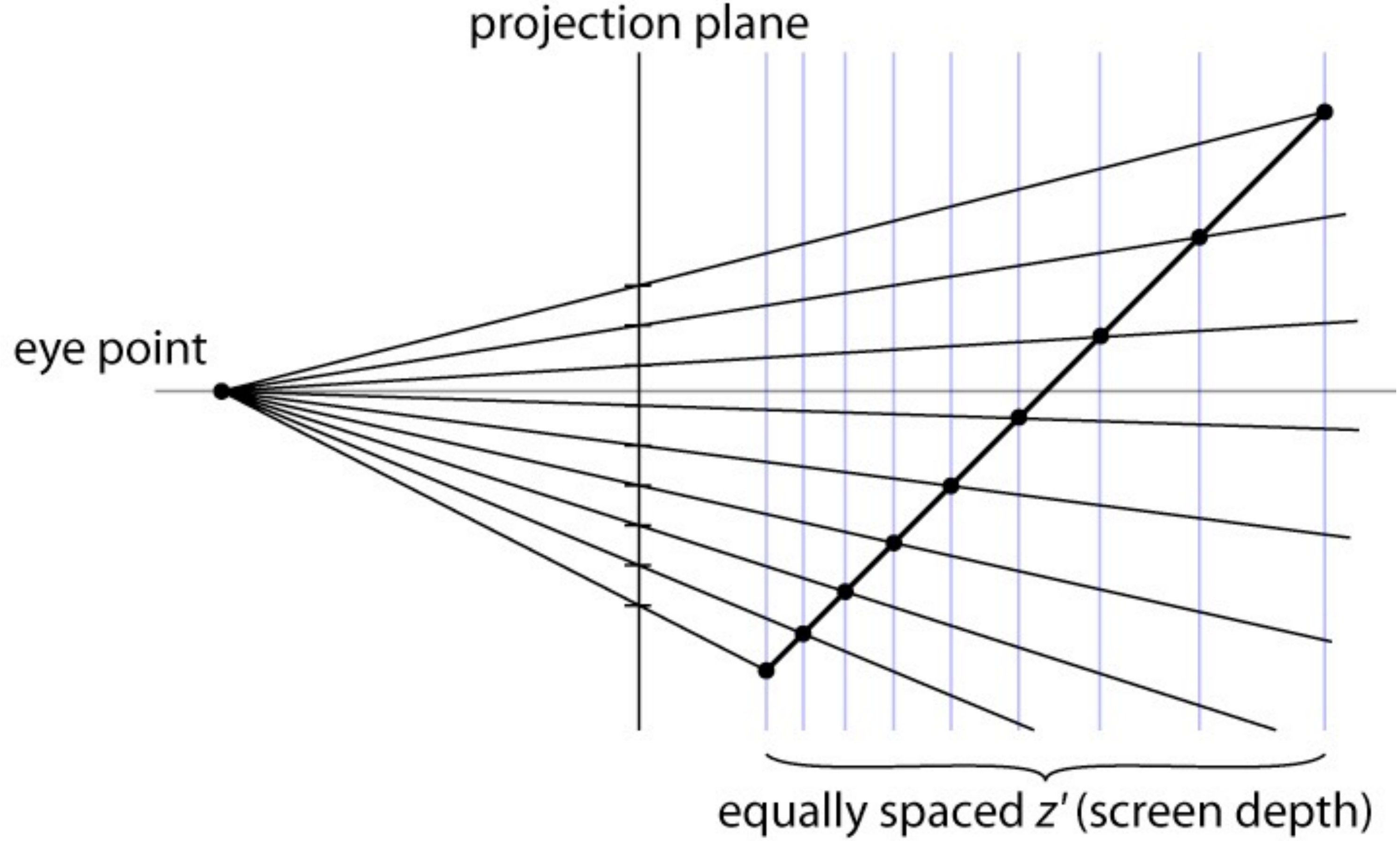
linear interp. in screen space ≠ linear interp. in world (eye) space

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

How did this happen?

screen space depth

$$z' = f + n - \frac{fn}{z}$$

camera space depth

$$\mathbf{P} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

How did this happen?

$$\mathbf{P} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

screen space depth

$$z' = f + n - \frac{fn}{z}$$

camera space depth

$$z' = k_1 - k_2 \frac{1}{z}$$

How did this happen?

$$\mathbf{P} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

screen space depth

$$z' = f + n - \frac{fn}{z}$$

camera space depth

$$z' = k_1 - k_2 \frac{1}{z}$$

$$z' \propto \frac{-1}{z}$$

How did this happen?

$$\mathbf{P} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

screen space depth

camera space depth

$$z' = f + n - \frac{fn}{z}$$

$$z' = k_1 - k_2 \frac{1}{z}$$

$$z' \propto \frac{-1}{z}$$

$$z \propto \frac{-1}{z'}$$

How did this happen?

$$\mathbf{P} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

screen space depth → z'; camera space depth → z

$$z' = f + n - \frac{fn}{z}$$

$$z' = k_1 - k_2 \frac{1}{z}$$

$$z' \propto \frac{-1}{z}$$

$$z \propto \frac{-1}{z'}$$

instead of using the smallest z',

use the **largest** $\frac{1}{z'}$

How did this happen?

$$\mathbf{P} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

screen space depth

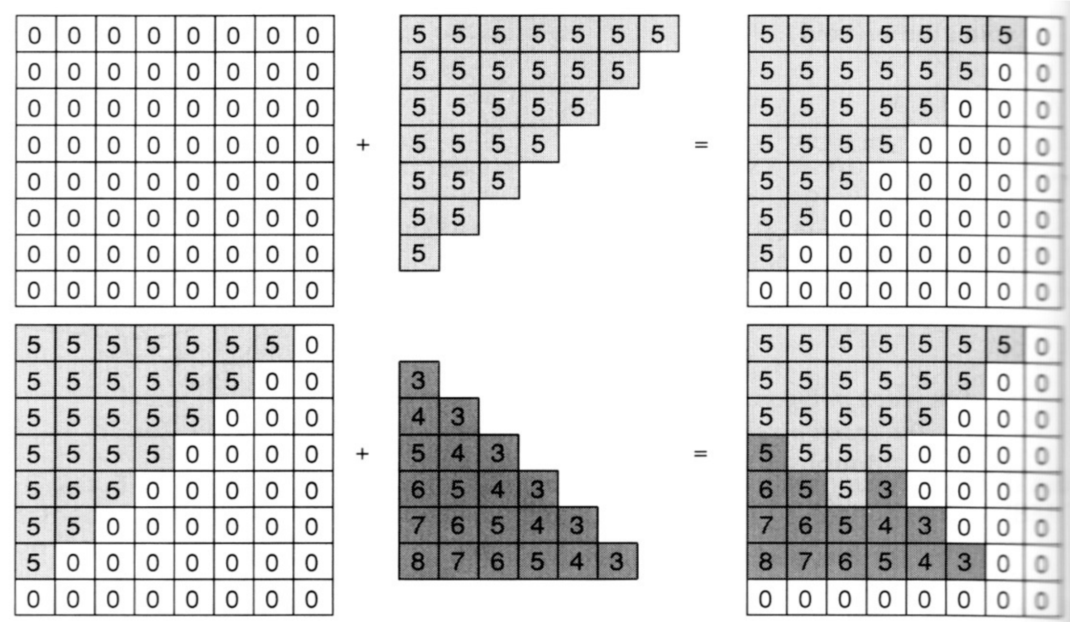
camera space depth

$$z' = f + n - \frac{fn}{z}$$

$$z' = k_1 - k_2\frac{1}{z}$$

$$z' \propto \frac{-1}{z}$$

$$z \propto \frac{-1}{z'}$$



instead of using the smallest z',

use the **largest** $\frac{1}{z'}$

Graphics Pipeline: Overview