Hardware Rendering

Brian Curless
CSE 557
Autumn 2017

Reading

Required:

- Shirley, Ch. 7, Sec. 8.2, Ch. 18

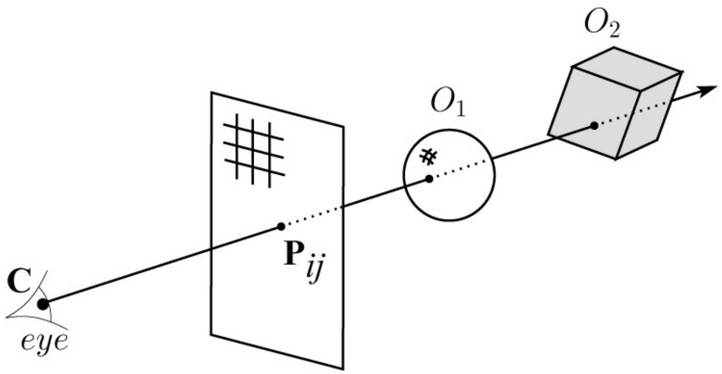
Further reading:

- Foley, et al, Chapter 5.6 and Chapter 6
- David F. Rogers and J. Alan Adams, *Mathematical Elements for Computer Graphics*, 2nd Ed., McGraw-Hill, New York, 1990, Chapter 2.
- I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, A characterization of ten hidden surface algorithms, *ACM Computing Surveys* 6(1): 1-55, March 1974.

Going back to the pinhole camera…

Recall that the Trace project uses, by default, the pinhole camera model.

If we just consider finding out which surface point is visible at each image pixel, then we are **ray casting**.



For each pixel center $\boldsymbol{P}_{ij}$

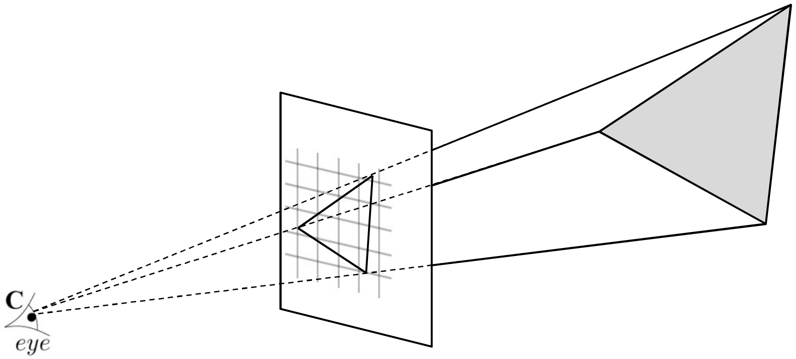
- Send ray from eye point (COP), $\mathbf{C}$, through $\boldsymbol{P}_{ij}$ into scene.
- For each object, intersect with the ray
- Select nearest intersection.

Alternative Approach

We could also flip the order of the loops:

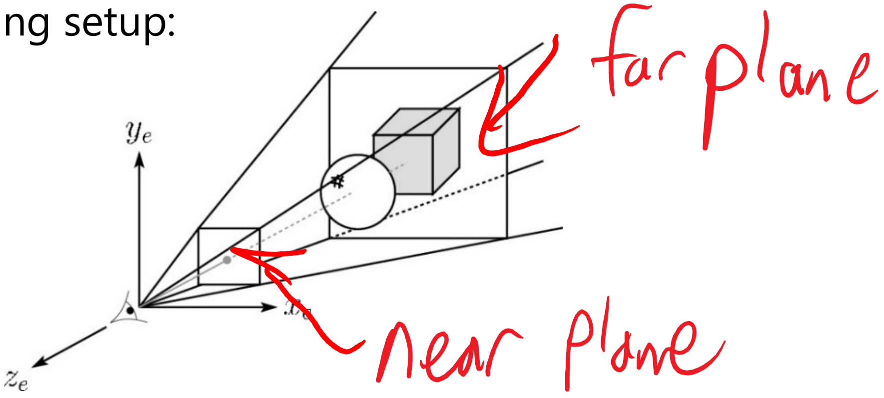
For each triangle in the scene,

- For each pixel, determine if the triangle projects onto it
- Update pixel if this triangle is the closest one so far



Warping space

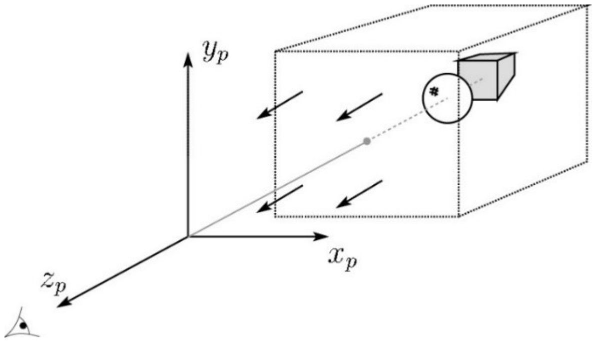
To determine which pixels a triangle projects onto, take this imaging setup:



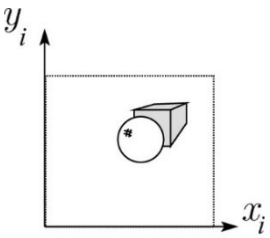
far plane

near plane

then warp all of space so that all the rays are parallel:



then just drop the z-coordinate to get pixel coordinates:

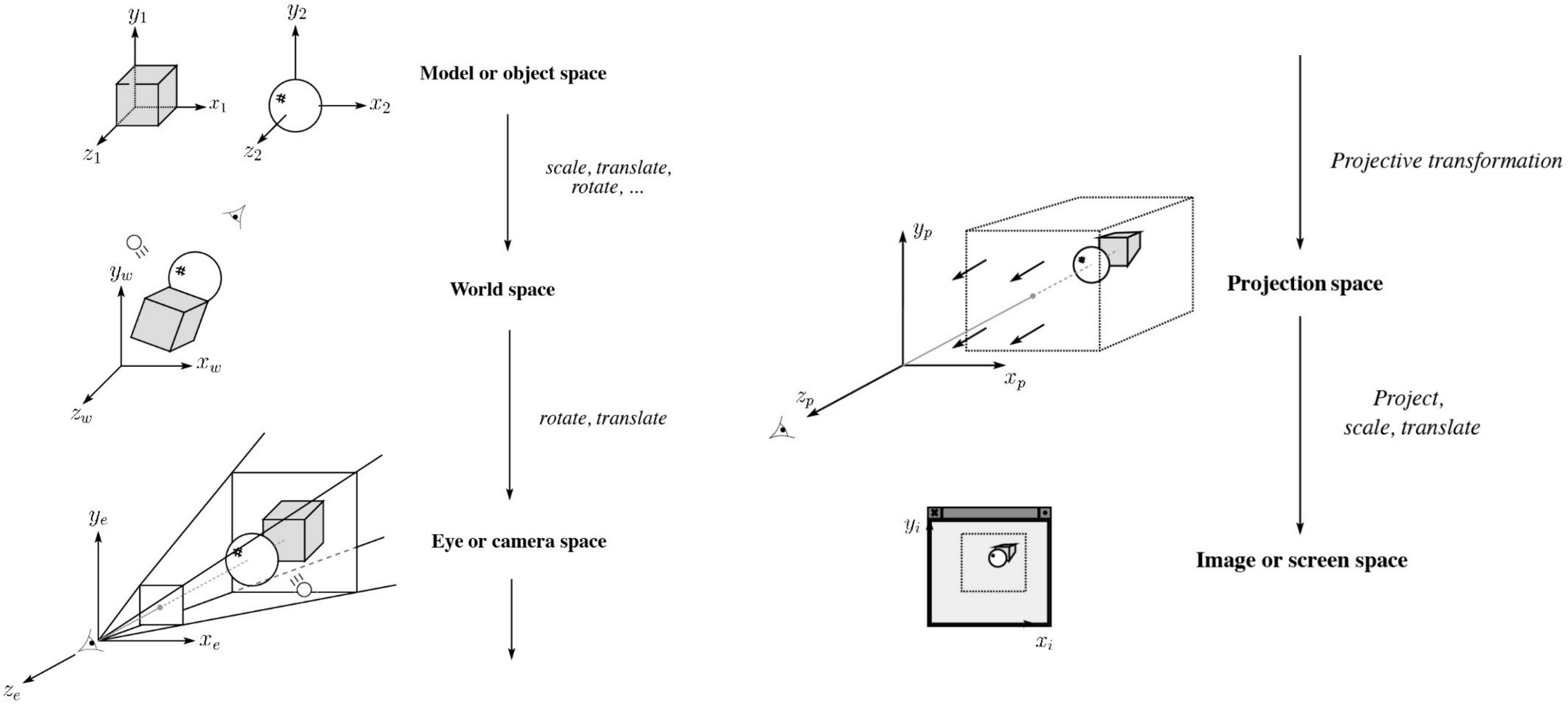


In practice, we keep track of the z-coordinate during drawing to determine visibility.

3D Geometry Pipeline

Graphics hardware follows the “warping space” approach.

Before being turned into pixels, a piece of geometry goes through a number of transformations...



Z-buffer

The **Z-buffer** or **depth buffer** algorithm [Straßer, 1974][Catmull, 1974] can be used to determine which surface point is visible at each pixel.

Here is pseudocode for the Z-buffer hidden surface algorithm, for a viewer looking down the $-z$ axis (bigger – i.e., more positive – z's are closer):

```
for each pixel $(i, j)$ do
    Z-buffer $[i, j] \leftarrow$ FAR
    Framebuffer $[i, j] \leftarrow$ <background color>
end for
for each triangle $A$ do
    for each pixel $(i, j)$ in $A$ do
        Compute depth $z$ of $A$ at $(i, j)$
        color $\leftarrow$ shader($A, i, j$)
        if $z$ > Z-buffer $[i, j]$ then
            Z-buffer $[i, j] \leftarrow z$
            Framebuffer $[i, j] \leftarrow$ color
        end if
    end for
end for
```

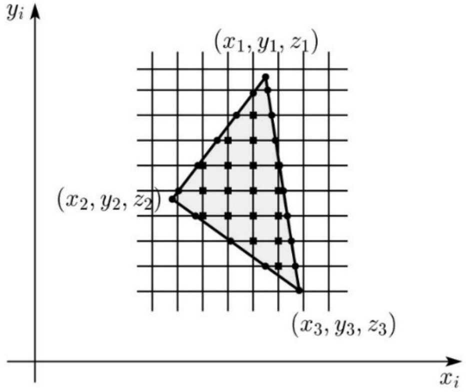
Q: What should *FAR* be set to? — ∞

Rasterization

We only need to compute the pixel coordinates of the vertices of the triangle – the interior pixels can be determined via interpolation.

This process called **rasterization**.

During rasterization, the z value can be computed incrementally (fast!).



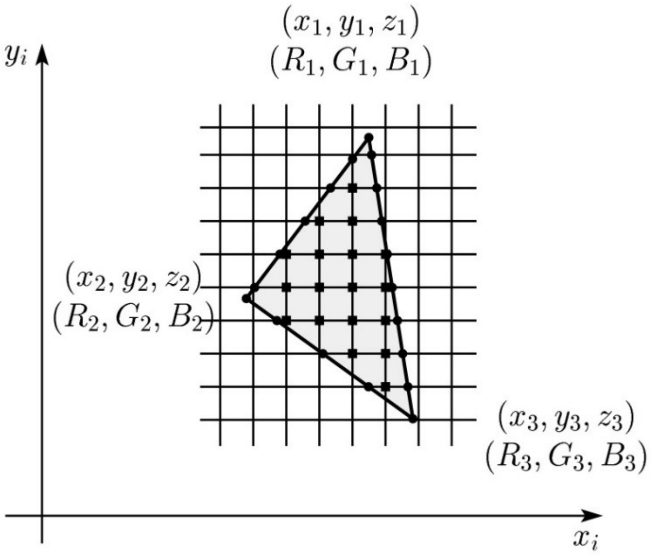
Curious fact:

- Described as the "brute-force image space algorithm" by [SSS]
- Mentioned only in Appendix B of [SSS] as a point of comparison for huge memories, but written off as totally impractical.

Today, Z-buffers are commonly implemented in hardware.

Rasterization with color

During rasterization, colors can be smeared across a triangle as well:



Hardware Pipeline

Vertex processor

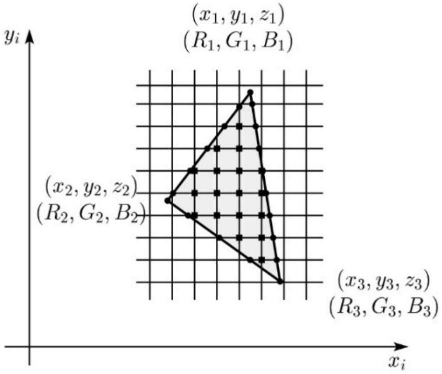
A vertex shader is run for each vertex, and outputs values to be interpolated across the triangle.

Primitive assembler

The vertices are grouped into triangles (or other primitives, e.g. lines) to be rasterized. A geometry shader is possibly run to generate more primitives.

Rasterizer

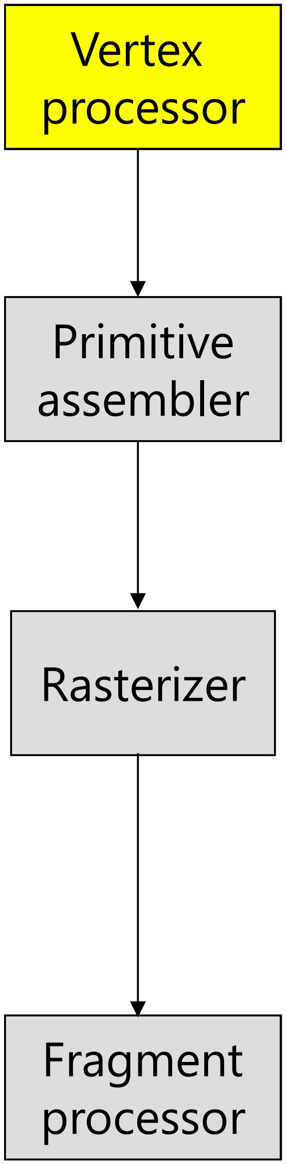
We iterate through scanlines, interpolating outputs from the vertex shader at each pixel.



Fragment processor

A fragment shader (or pixel shader) is called at each pixel in the primitive, which gets the interpolated values and outputs a final color to the framebuffer.

GLSL: Anatomy of a Vertex Shader


Vertex processor

Primitive assembler

Rasterizer

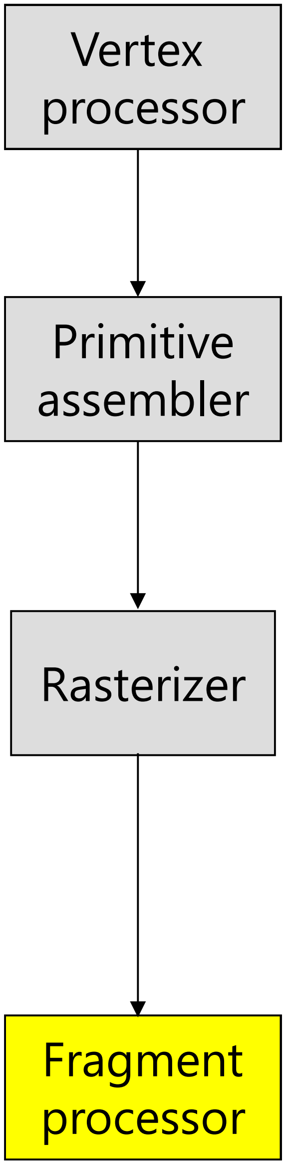
Fragment processor

```
#version 400

in vec3 position;
in vec3 vertex_color;

out vec3 color;

uniform mat4 modelview;
uniform mat4 projection;

void main() {
    color = vertex_color;
    gl_Position = projection * modelview * vec4(position, 1.0);
    // color = vec3(1.0, 0.0, 0.0);
    // gl_Position = vec4(1.0, -1.0, 0.0, -1.0);*
}
```

*NDC"

normalized device coordinates

GLSL: Anatomy of a Fragment Shader


Vertex processor

Primitive assembler

Rasterizer

Fragment processor

```
#version 400

in vec3 color;

out vec4 frag_color;

void main() {
    frag_color = color;
}
```

→ Same as V.S.

Out vec3 color

GLSL: Storage Qualifiers

`uniform`: Global value that is the same across all vertices and fragments (for this draw call).

- Model/view/projection matrices, light parameters, material parameters (maybe), textures…

Vertex shader **`in`**: Per-vertex attributes (that were sent to the GPU)

Vertex shader **`out`**: Values to be interpolated at each fragment shader

Fragment shader **`in`**: Interpolated values of Vertex shader **`out`**’s

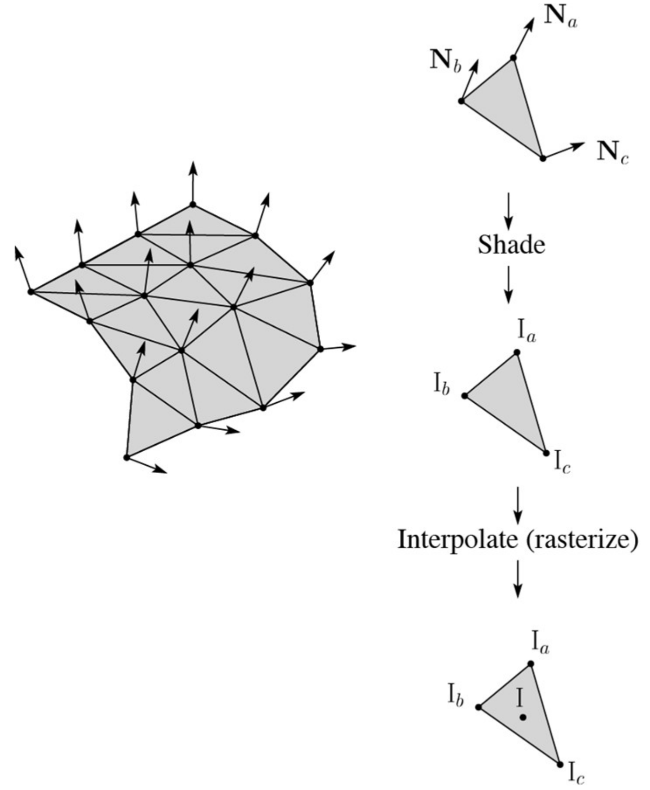
Fragment shader **`out`**: Value to be written to frame buffer

- Normals, positions, colors, material parameters (maybe), texture coordinates…

Gouraud interpolation

Recall from the shading lecture, rendering with per triangle normals leads to faceted appearance. An improvement is to compute per-vertex normals and use graphics hardware to do **Gouraud interpolation**:

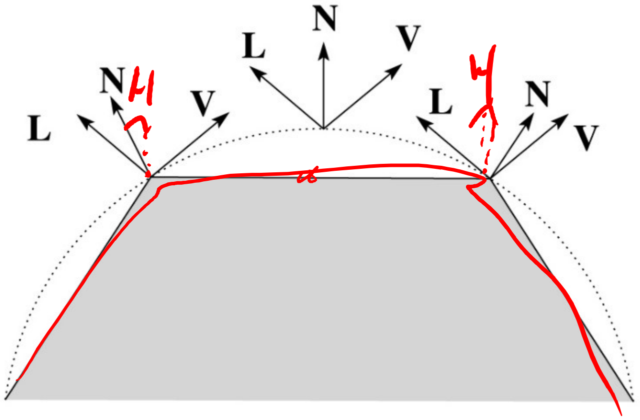
1. Compute normals at the vertices.
2. Shade only the vertices.
3. Interpolate the resulting vertex colors.



Gouraud interpolation artifacts

Gouraud interpolation has significant limitations.

1. If the polygonal approximation is too coarse, we can miss specular highlights.



2. We will encounter **Mach banding** (derivative discontinuity enhanced by human eye).

This is what graphics hardware does by default.

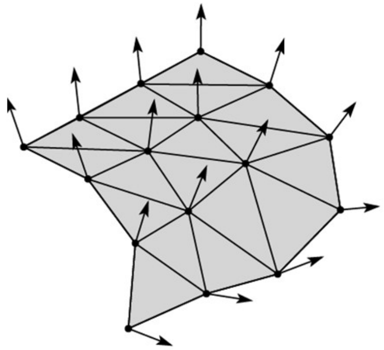
A substantial improvement is to do…

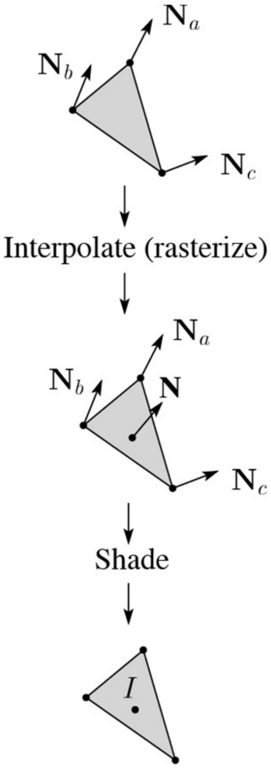
Phong interpolation

To get an even smoother result with fewer artifacts, we can perform **Phong *interpolation***.

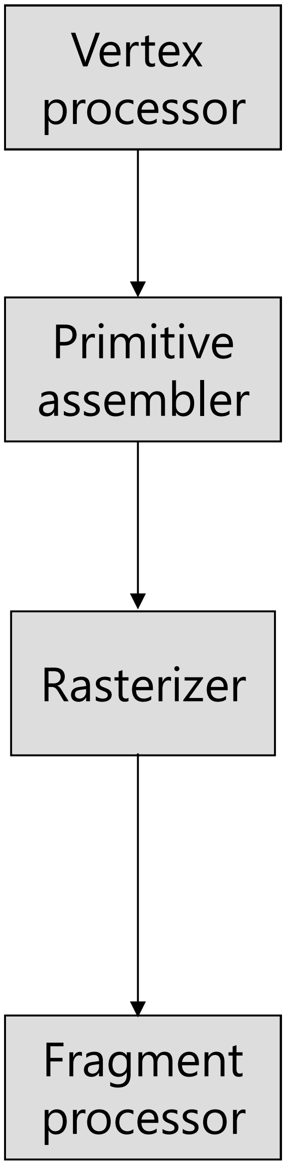
Here’s how it works:

1. Compute normals at the vertices.
2. Interpolate normals and normalize.
3. Shade using the interpolated normals.



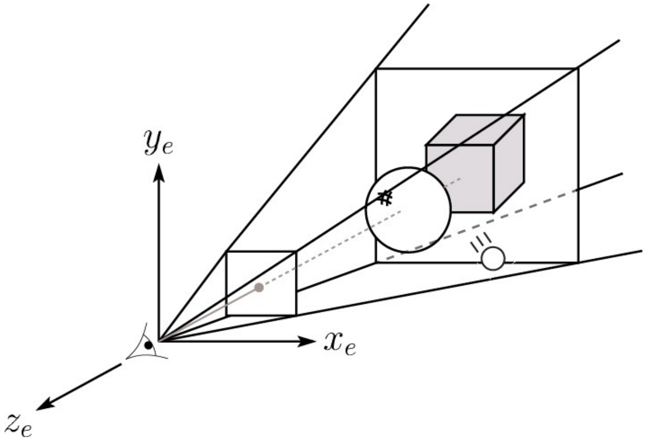


Old pipeline: Gouraud interpolation

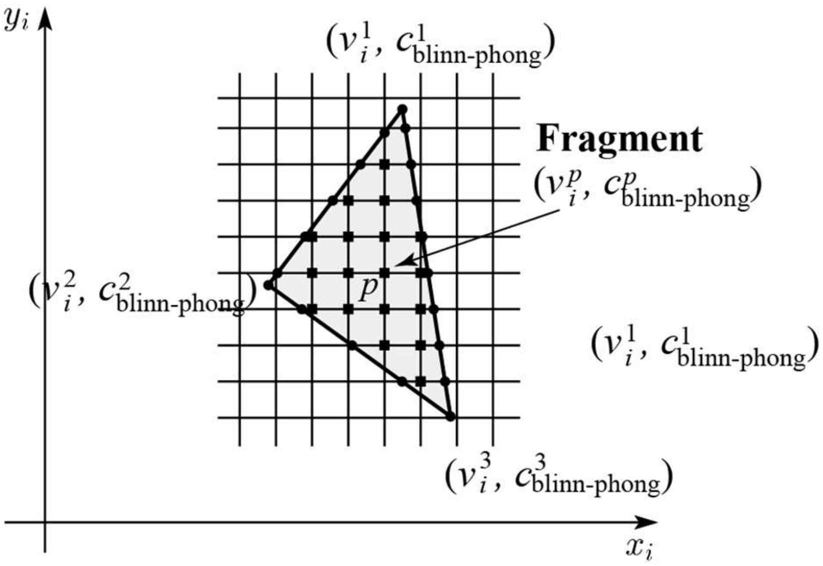


Default vertex processing:

```
$\mathbf{L} \leftarrow$ determine lighting direction
$\mathbf{V} \leftarrow$ determine viewing direction
$\mathbf{N} \leftarrow$ normalize($\mathbf{n}_e$)
$c_{\text{blinn-phong}} \leftarrow$ shade with $\mathbf{L}, \mathbf{V}, \mathbf{N}, k_d, k_s, n_s$
$v_i \leftarrow$ project $v$ to image
out $c_{\text{blinn-phong}}$
out $v_i$
```



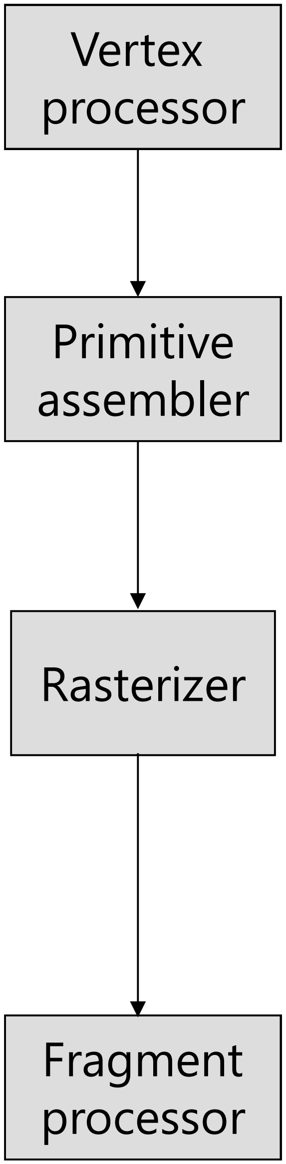
$v_i^1, v_i^2, v_i^3 \rightarrow$ triangle



Default fragment processing:

color $\leftarrow c^p_{\text{blinn-phong}}$

Programmable pipeline: Phong-interpolated normals!



Vertex processor

Primitive assembler

Rasterizer

Fragment processor

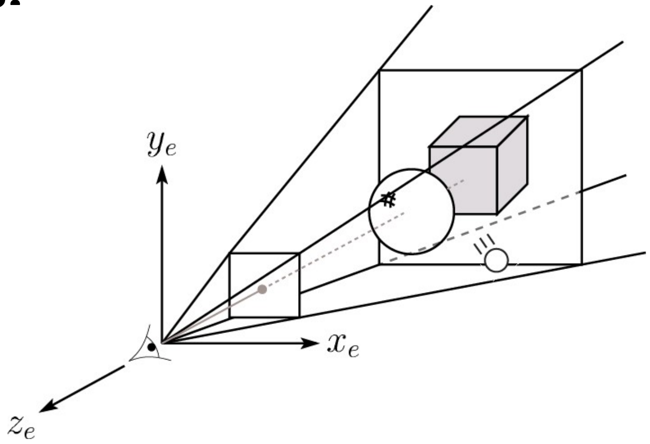
Vertex shader:

$v_i \leftarrow$ project v to image

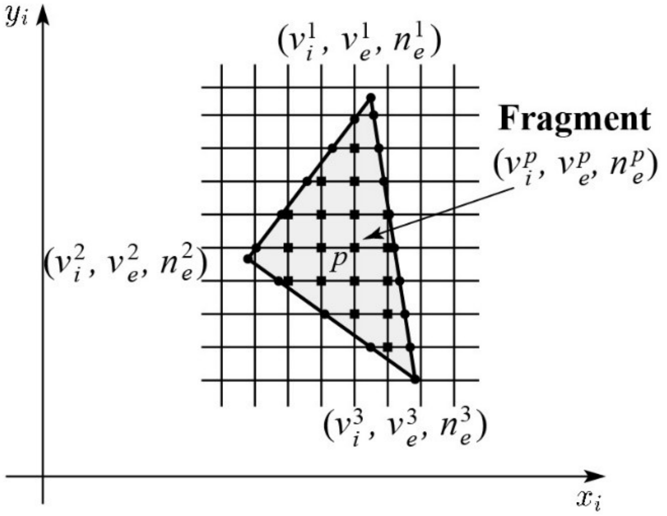
`out` $\mathbf{n}_e$

`out` v_e

`out` v_i



$v_i^1, v_i^2, v_i^3 \rightarrow$ triangle



Fragment shader:

$\mathbf{L} \leftarrow$ determine lighting direction (using v_e^p)

$\mathbf{V} \leftarrow$ normalize$\left(-v_e^p\right)$

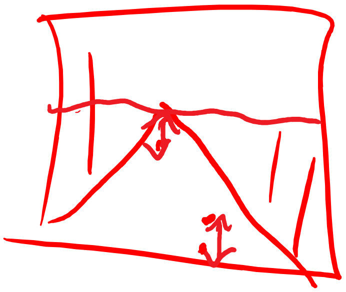
$\mathbf{N} \leftarrow$ normalize$(\mathbf{n}_e^p)$

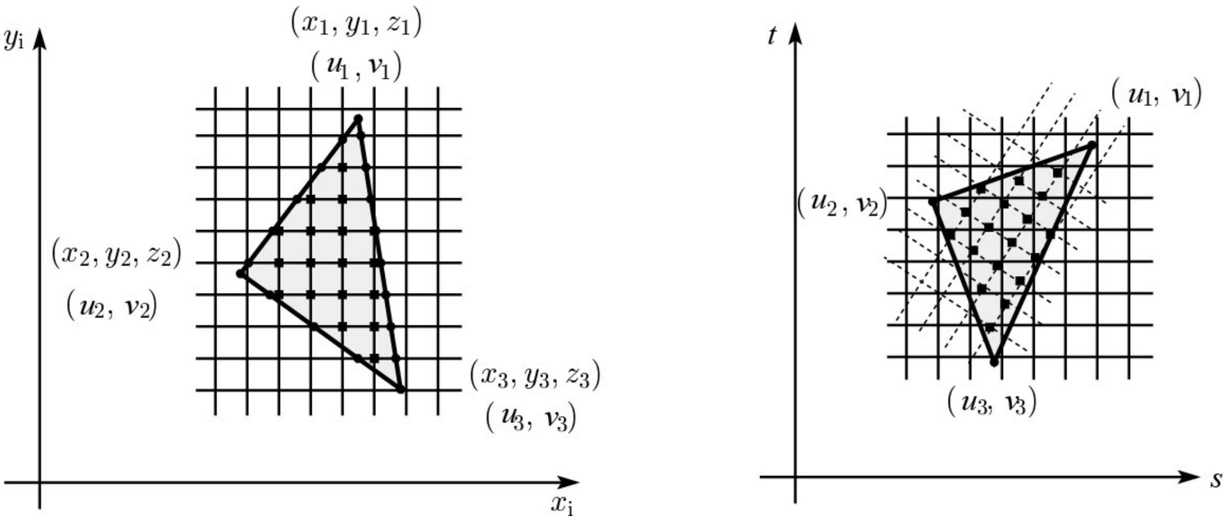
color $\leftarrow$ shade with $\mathbf{L}, \mathbf{V}, \mathbf{N}, k_d, k_s, n_s$

Texture mapping and the z-buffer

Method:

- Supply per-vertex texture coordinates
- Scan conversion is done in screen space, as usual
- Texture coordinates are interpolated, as usual
- Supply a **`uniform`** with the texture data
- Each pixel is colored by looking up the texture at the interpolated coordinates





Note: Mapping is more complicated to handle perspective correctly! (OpenGL does this by default)

Rasterization vs Raycasting

Fundamental loop: For each pixel and triangle, determine if they intersect

- Observation: Adjacent pixels often hit the same triangle.
    - In raycasting, you throw away this knowledge!
    - In rasterization, you don't even need to compute the intersection at interior pixels
- In raycasting, you accelerate by culling triangles, while in rasterization, you cull pixels instead
    - Culling triangles requires an acceleration data structure storing the whole scene
    - Traversing this data structure causes branching
    - But, rasterization might do more unnecessary work
- Rasterization doesn't naturally generalize to recursive (multi-bounce) effects like reflections and shadows
    - There are plenty of hacks (as you'll see for shadows)