

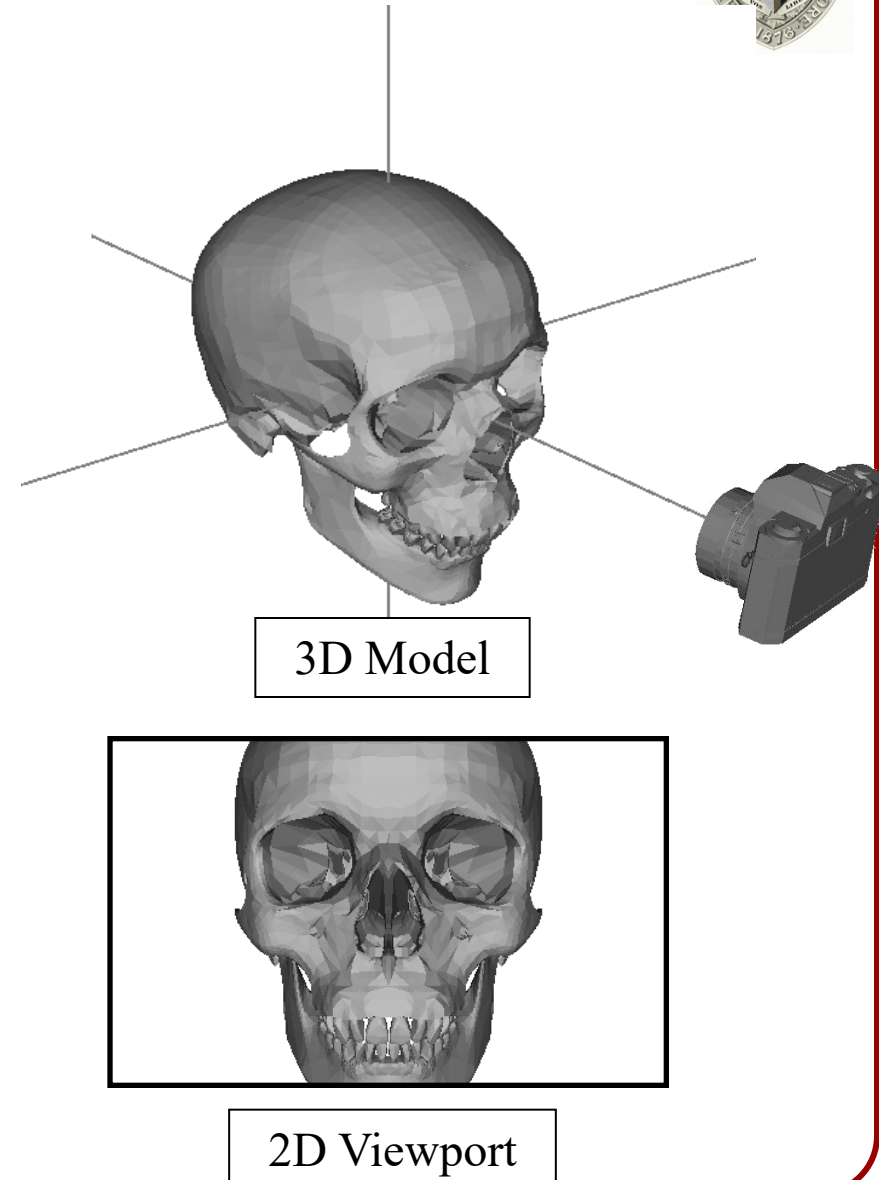
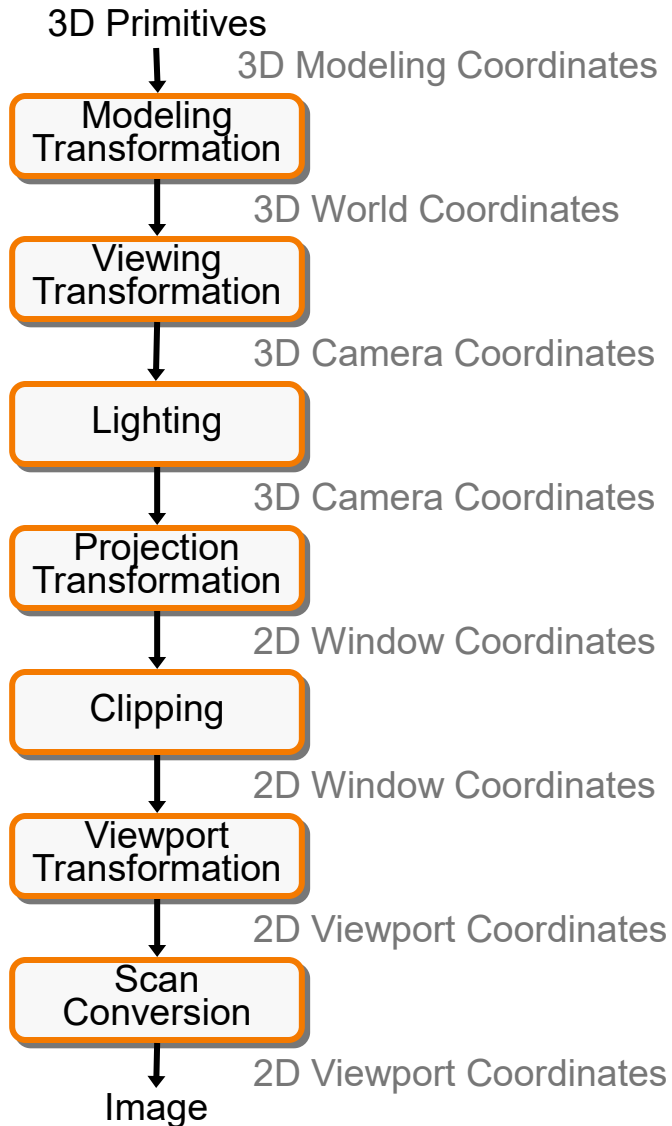


Scan Conversion

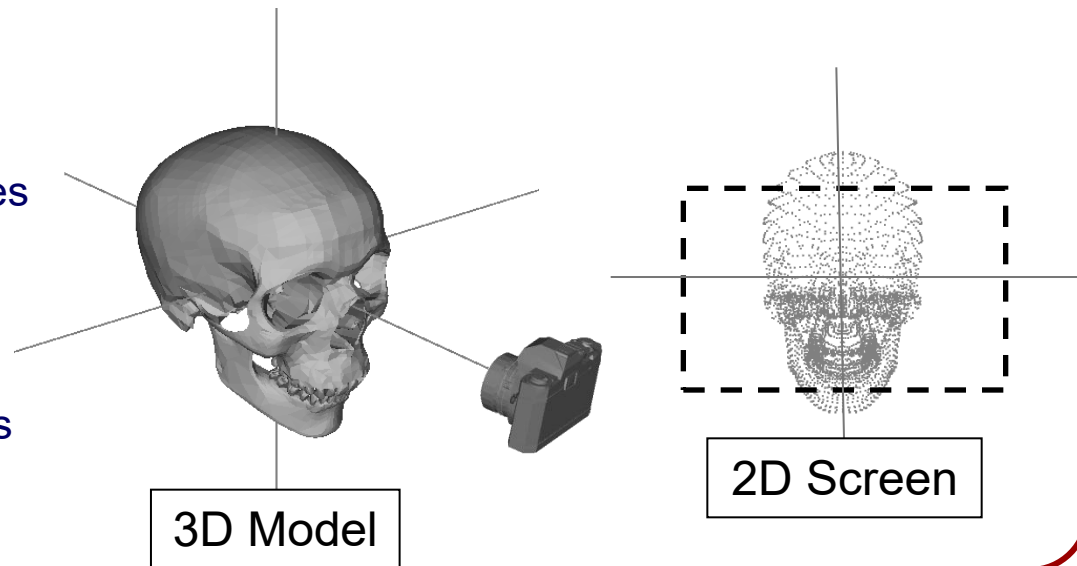
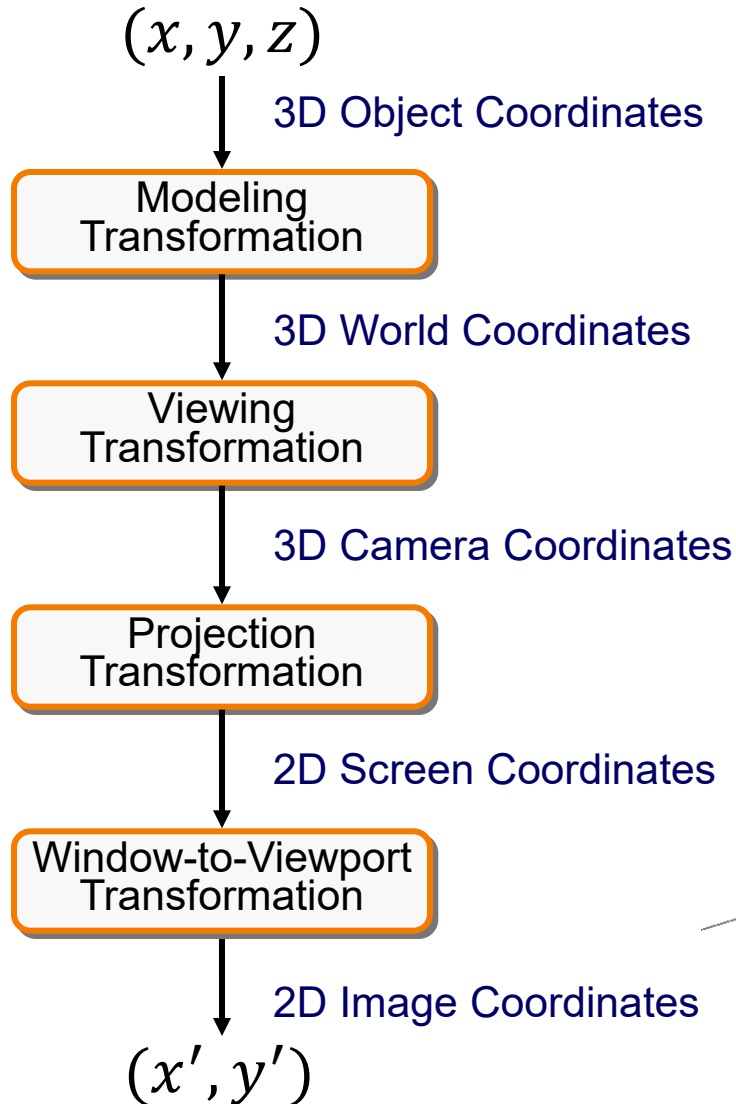
Misha Kazhdan

(601.457/657)

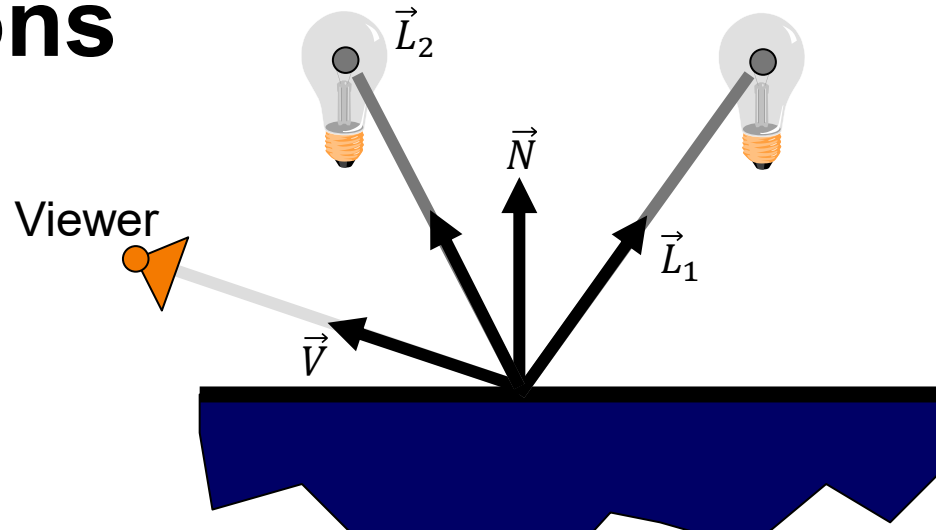
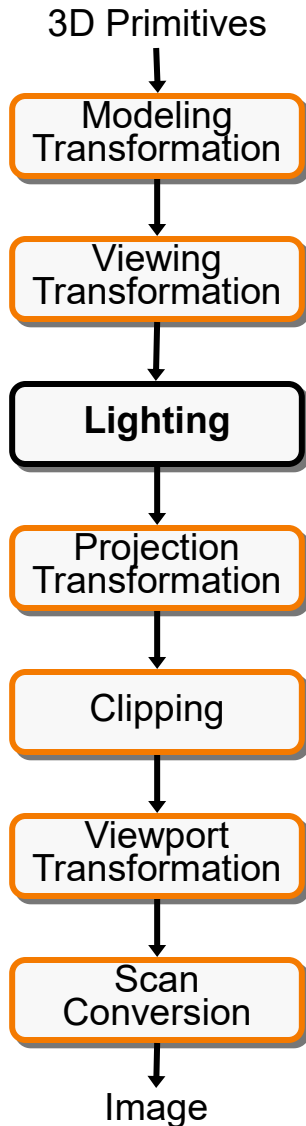
3D Rendering Pipeline (for direct illumination)



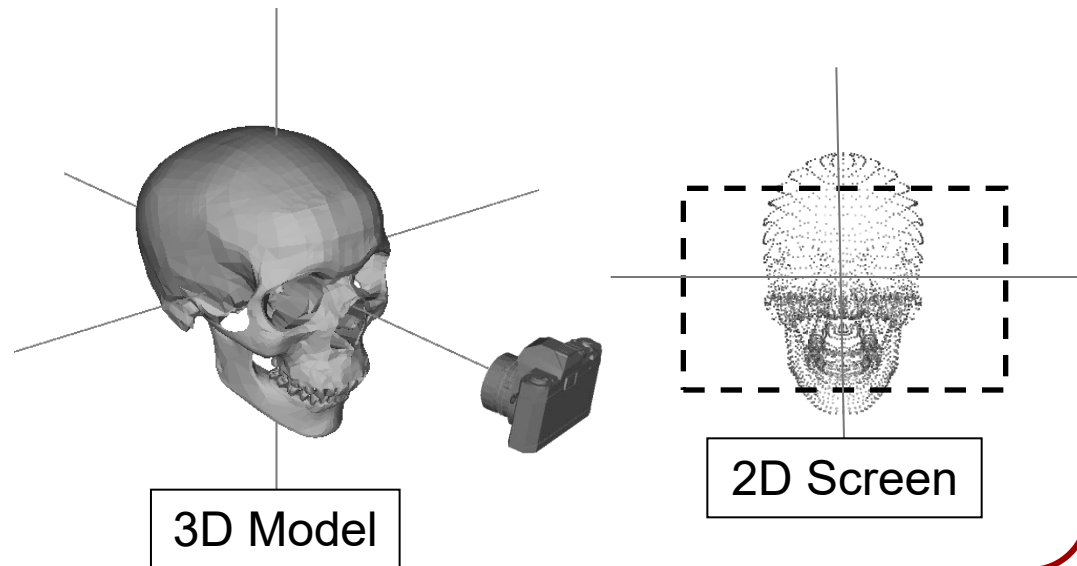
3D Rendering Pipeline (for direct illumination)



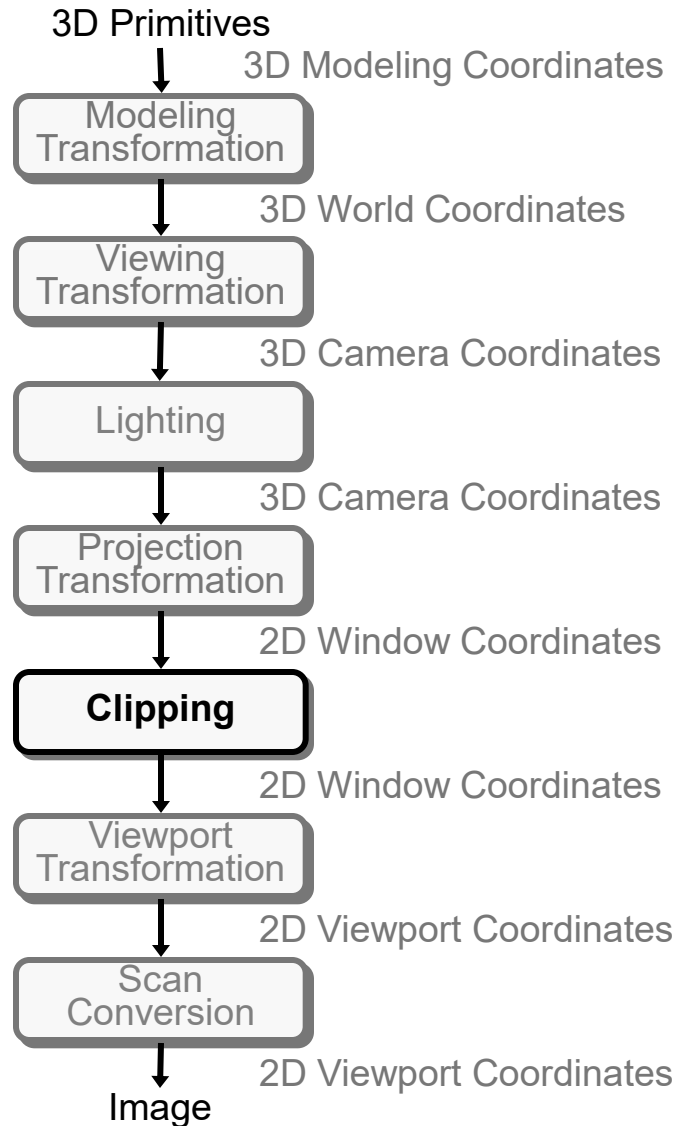
Transformations



$$I = I_E + \sum_L [K_A \cdot I_L^A + (K_D \cdot \langle \vec{N}, \vec{L} \rangle + K_S \cdot \langle \vec{V}, \vec{R}(\vec{L}) \rangle^{K_n}) \cdot I_L]$$

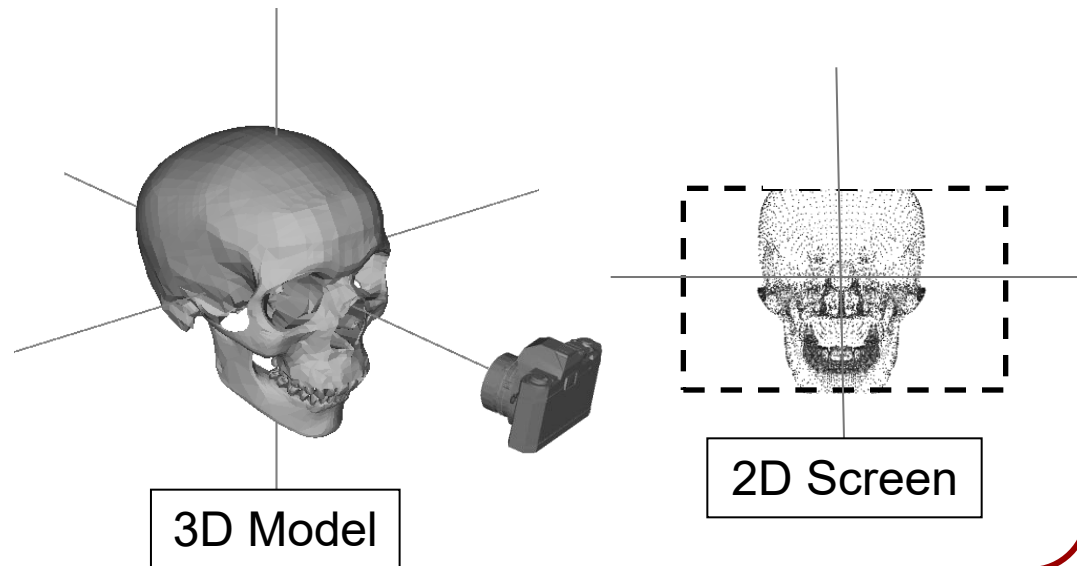


3D Rendering Pipeline (for direct illumination)



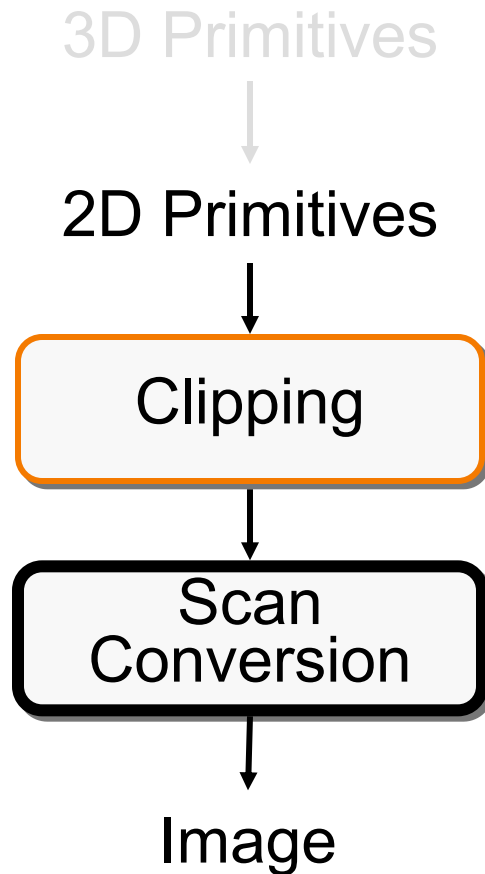
At this point we have the:

- 2D positions of the mesh vertices (with new vertices obtained through clipping)
 - Color at each vertex.
 - A list of clipped projected 2D polygons.
- Need to “fill in” the polygons.





2D Rendering Pipeline



Clip portions of geometric primitives residing outside the window

Fill pixels representing primitives in viewport coordinates



Overview

Scan conversion

Figure out which pixels to fill

Shading

Determine the color for each filled pixel

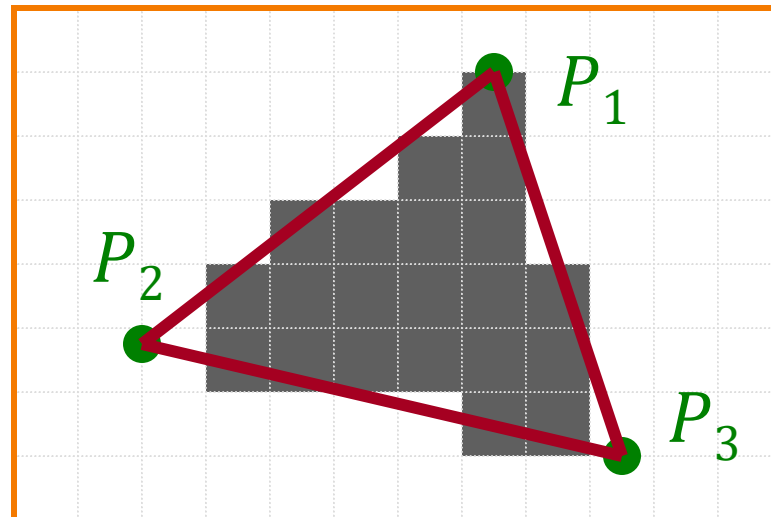
Depth test

Determine when pixel color should be overwritten



Scan Conversion

Render a polygon by setting interior pixel colors.

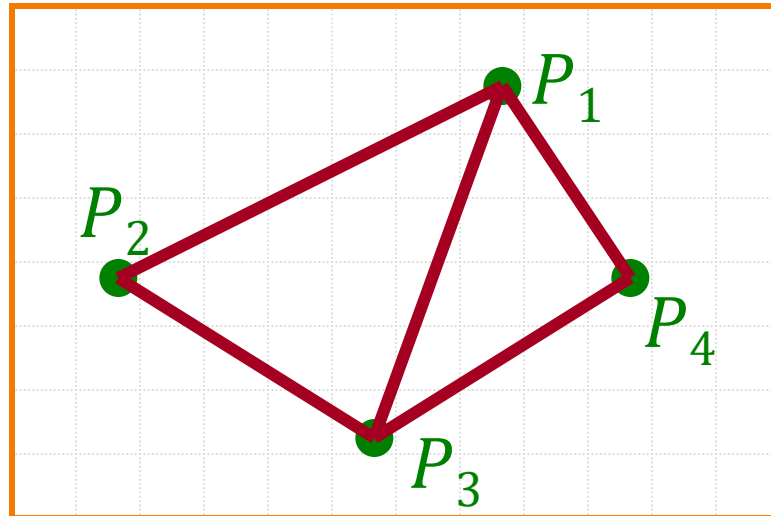




Triangle Scan Conversion

Properties of a good algorithm

- Must be fast

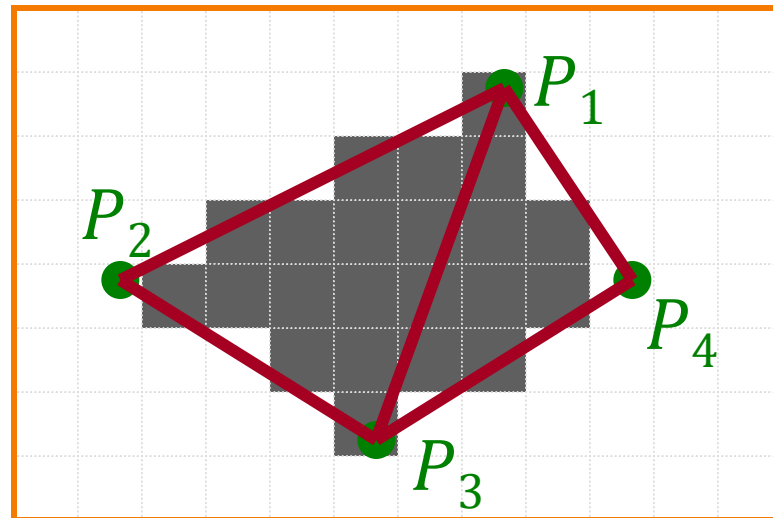




Triangle Scan Conversion

Properties of a good algorithm

- Must be fast
- Must be seamless

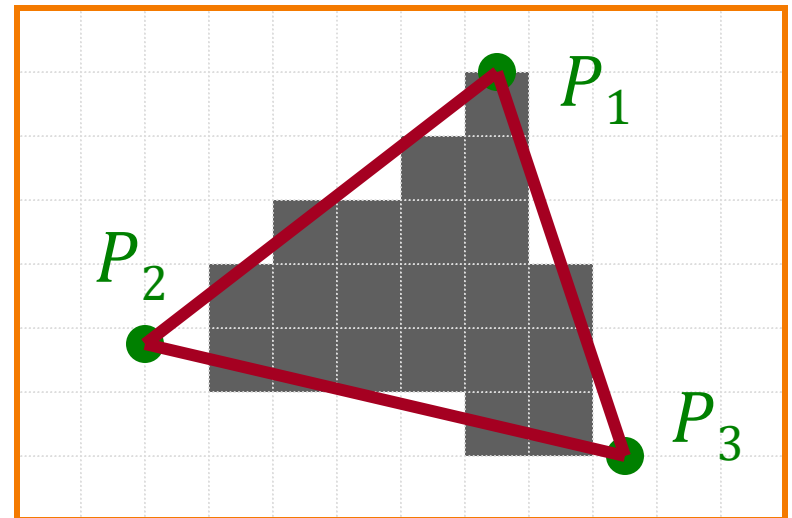




Simple Algorithm

Color all pixels inside triangle

```
void ScanTriangle( Triangle T , Color rgba )  
{  
    for each pixel center in image (x,y)  
        if( PointInsideTriangle( (x,y) , T ) )  
            SetPixel( x , y , rgba );  
}
```

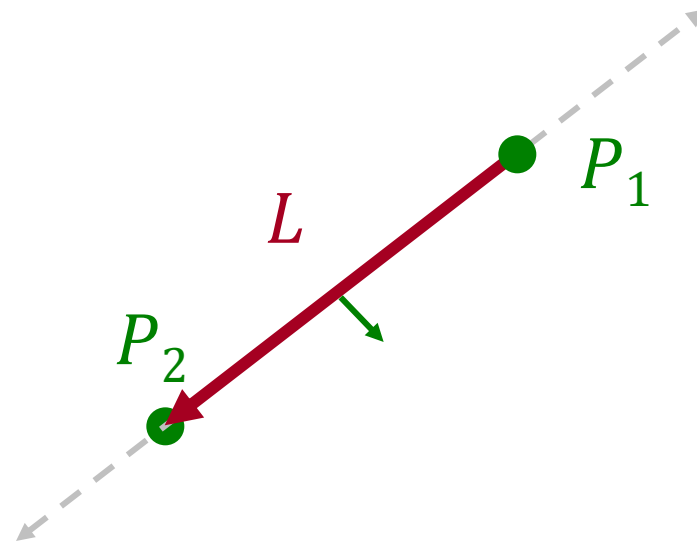




Line defines two halfspaces

Test: use implicit equation for a line

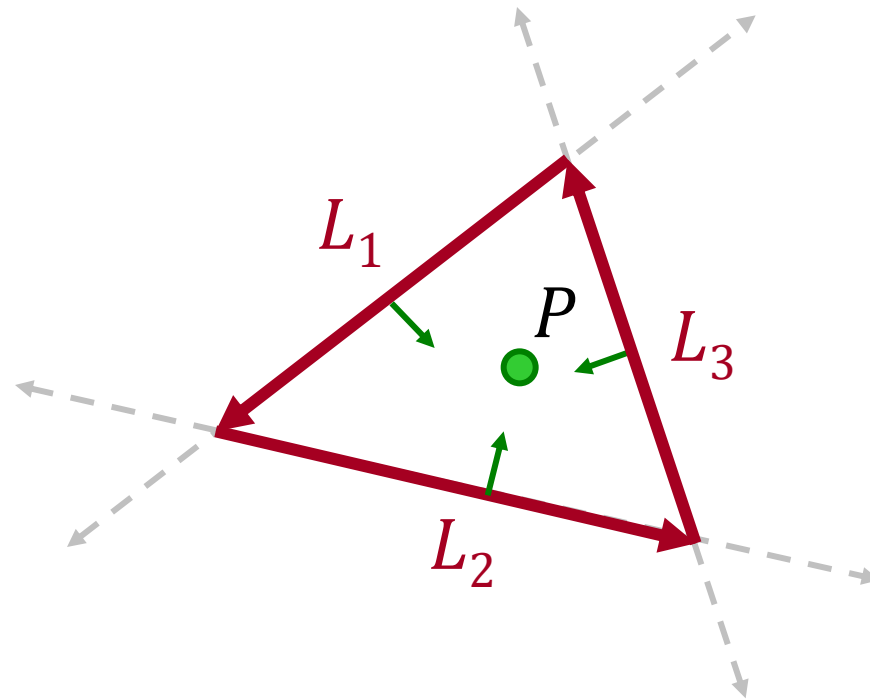
- On line: $ax + by + c = 0$
- To the right: $ax + by + c < 0$
- To the left: $ax + by + c > 0$





Inside Triangle Test

Assume triangle vertices are ordered counter-clockwise, since triangles are convex, an interior point must be to the left of every bounding line.

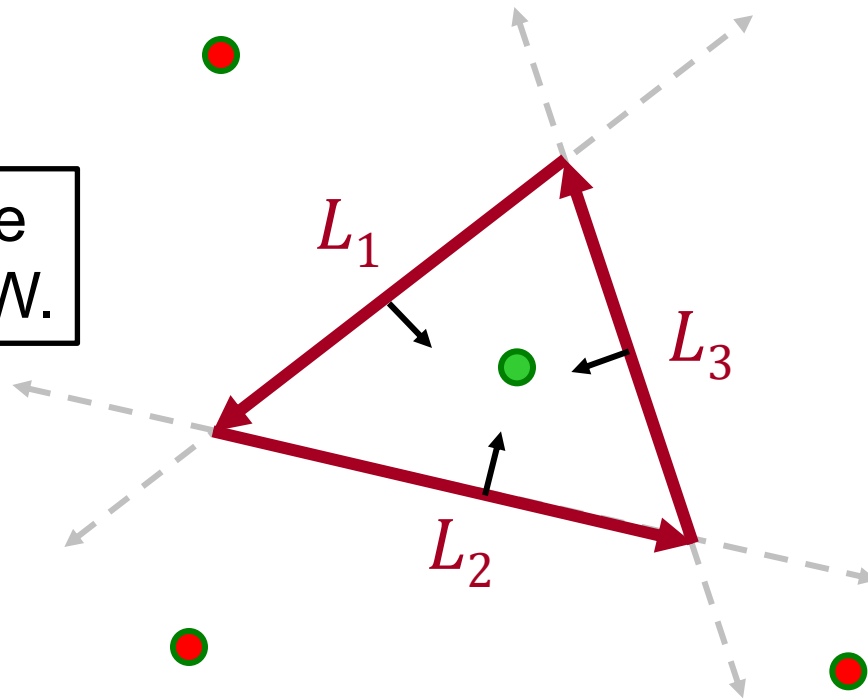




Inside Triangle Test

```
Boolean PointInsideTriangle( Point P , Triangle T )  
{  
  for each boundary line L of T  
  {  
    Scalar d = L.a*P.x + L.b*P.y + L.c;  
    if( d<0.0 ) return FALSE;  
  }  
  return TRUE;  
}
```

Assumes triangle
orientation is CCW.

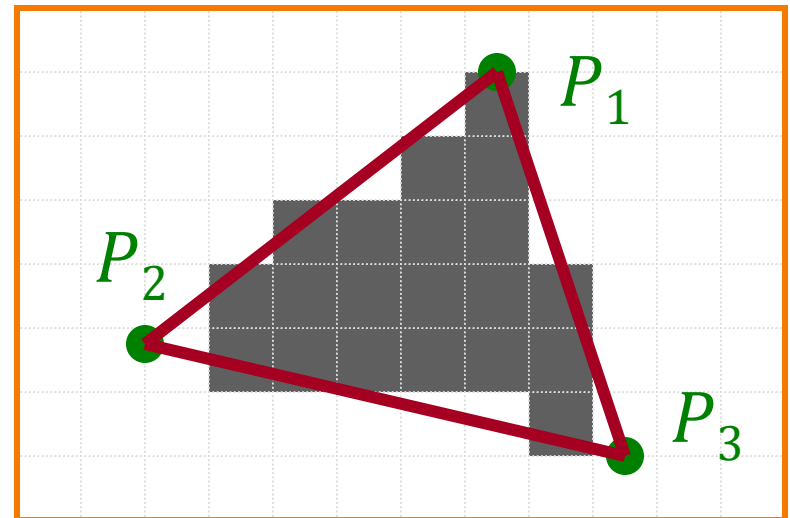




Simple Algorithm

What is bad about this algorithm?

```
void ScanTriangle( Triangle T , Color rgba )  
{  
    for each pixel center in image (x,y)  
        if( PointInsideTriangle( (x,y) , T ) )  
            SetPixel( x , y , rgba );  
}
```





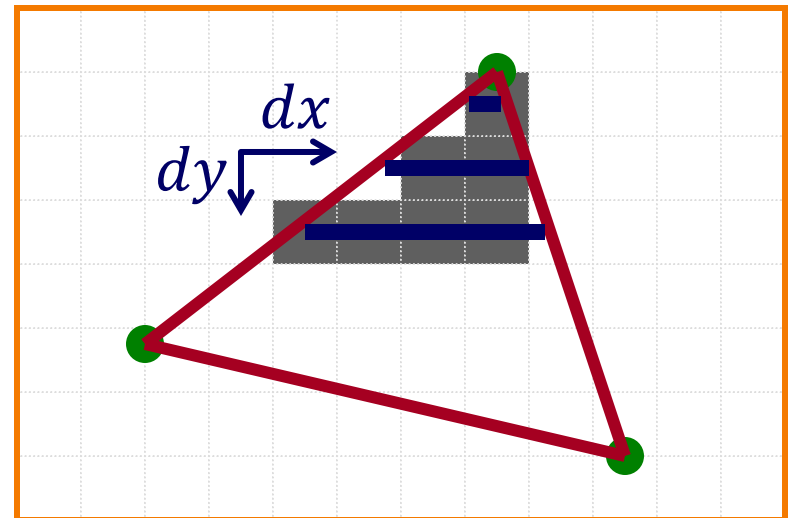
Triangle Sweep-Line Algorithm

Take advantage of spatial coherence

Per row, interior pixels are bounded by left/right edges.

Take advantage of edge linearity

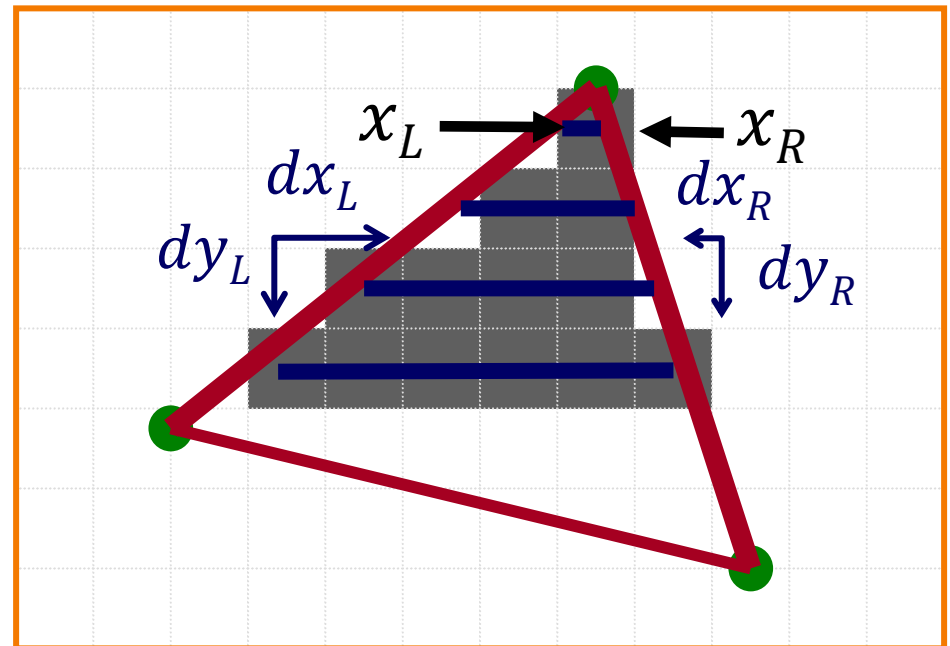
Moving from row to row, left/right boundary change is determined by the slope.





Triangle Sweep-Line Algorithm

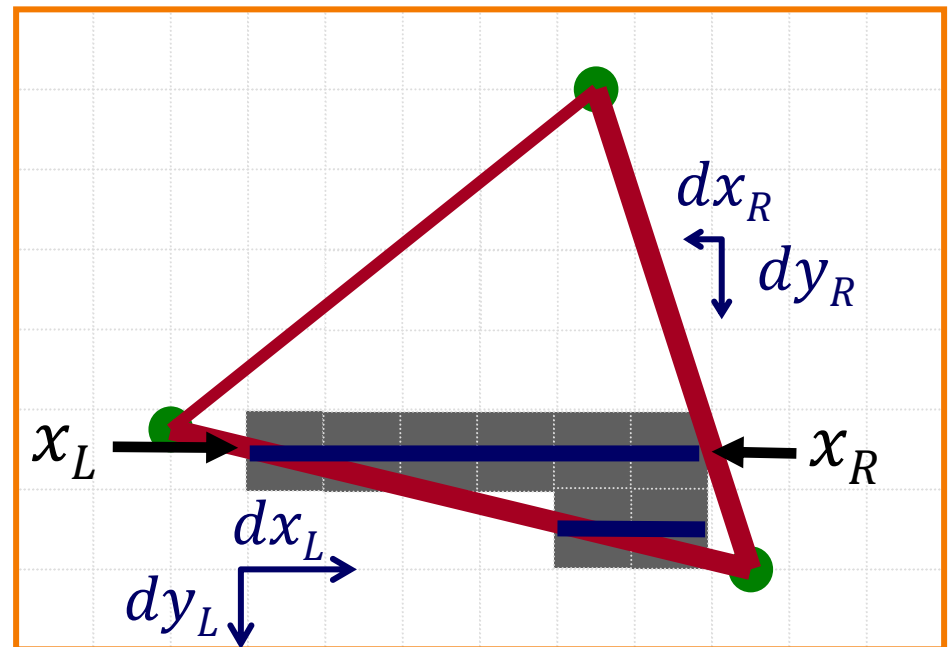
```
void ScanTriangle( Triangle T , Color rgba )
{
    for both edge pairs
    {
        initialize  $x_L$ ,  $x_R$ ,  $y$ ;
        compute  $dx_L/dy_L$  and  $dx_R/dy_R$ ;
        until  $y$  reaches the first end-point
            for( int  $x=x_L$  ;  $x \leq x_R$  ;  $x++$  ) SetPixel(  $x$  ,  $y$  , rgba );
             $x_L += dx_L/dy_L$ ;
             $x_R += dx_R/dy_R$ ;
             $y++$ ;
    }
}
```





Triangle Sweep-Line Algorithm

```
void ScanTriangle( Triangle T , Color rgba )  
{  
    for both edge pairs  
    {  
        initialize  $x_L$ ,  $x_R$ ,  $y$ ;  
        compute  $dx_L/dy_L$  and  $dx_R/dy_R$ ;  
        until  $y$  reaches the first end-point  
        for( int  $x=x_L$  ;  $x \leq x_R$  ;  $x++$  ) SetPixel(  $x$  ,  $y$  , rgba );  
         $x_L += dx_L/dy_L$ ;  
         $x_R += dx_R/dy_R$ ;  
         $y++$ ;  
    }  
}
```

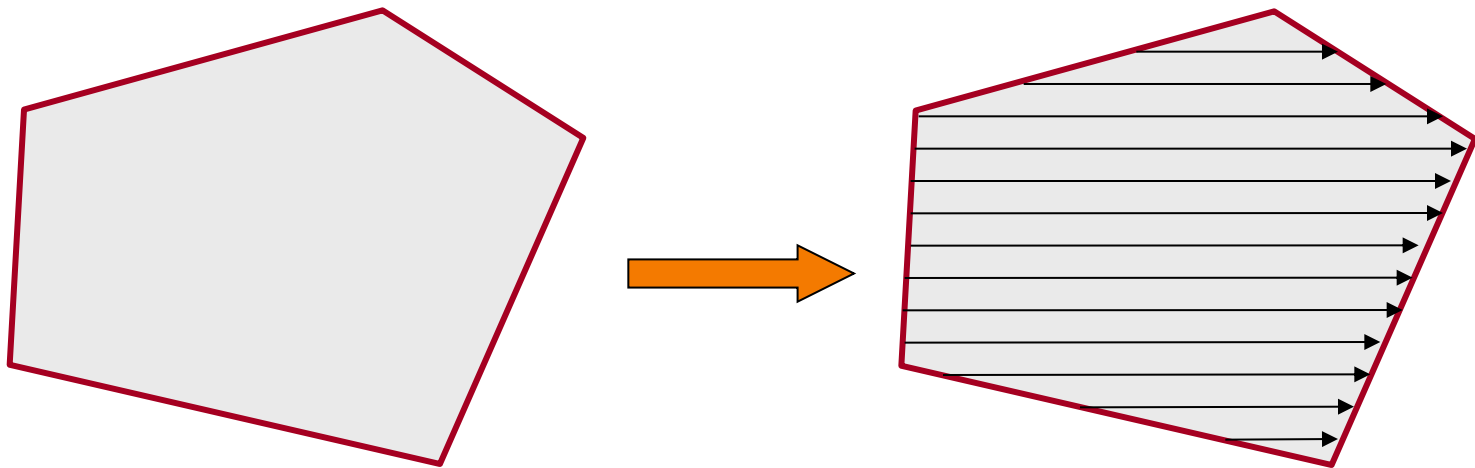




Polygon Scan Conversion

Will the method work for convex polygons?

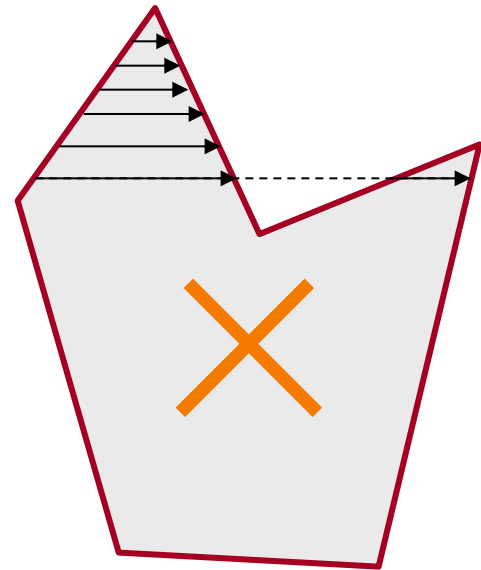
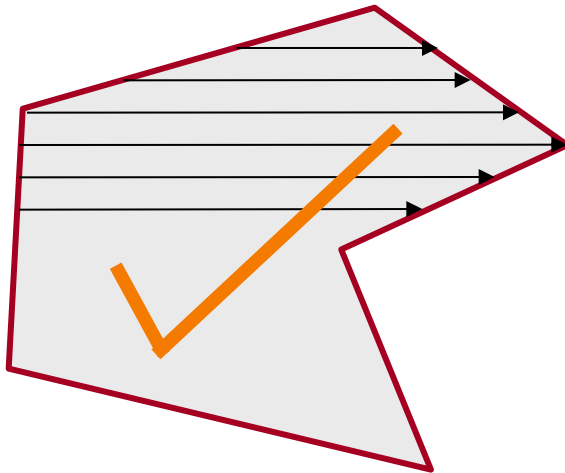
Yes, since each scan line will only intersect the polygon at two points.





Polygon Scan Conversion

How about these polygons?

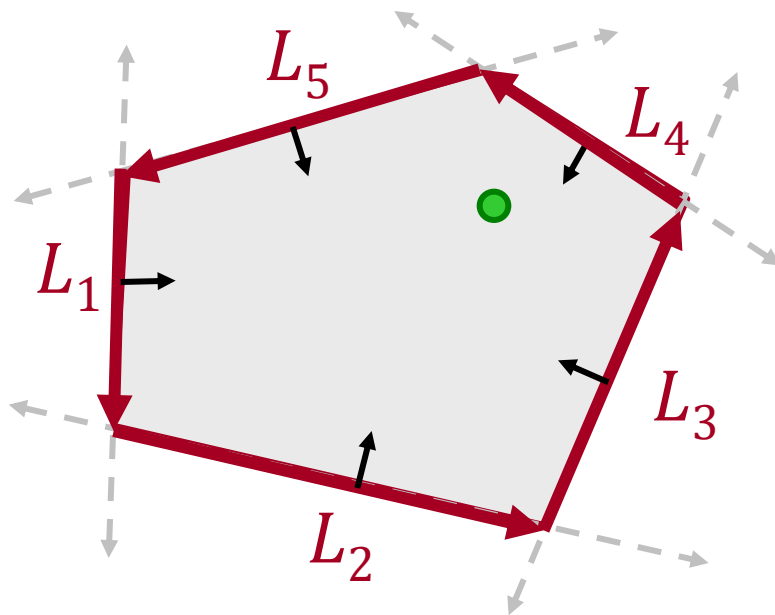




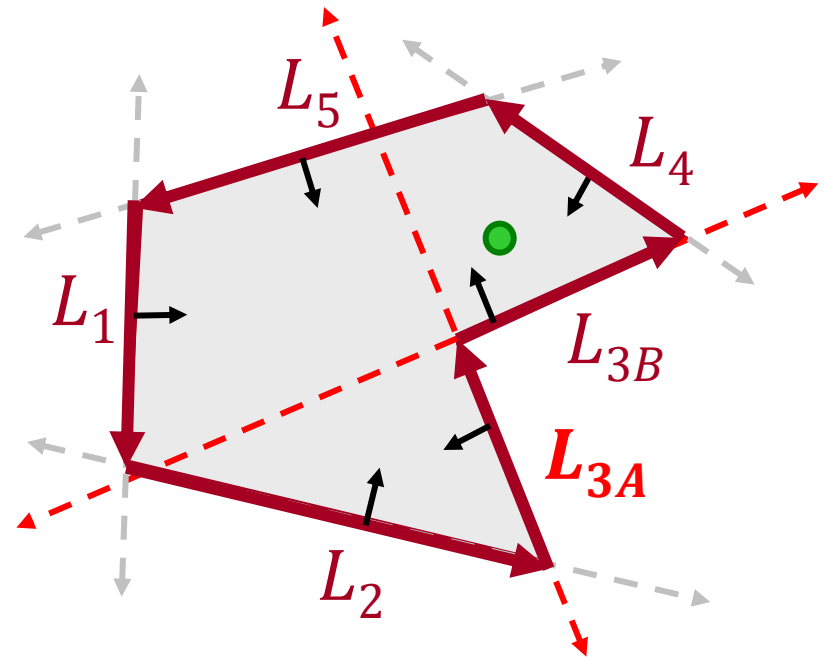
Polygon Scan Conversion

Need better test for points inside polygon

Triangle sweep-line algorithm only generalizes to convex polygons



Convex Polygon



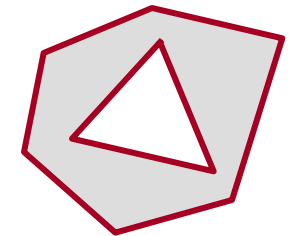
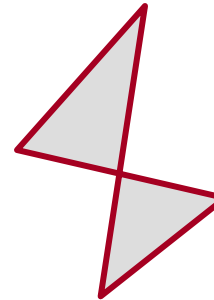
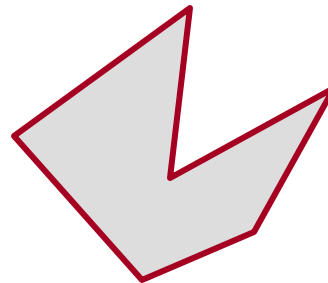
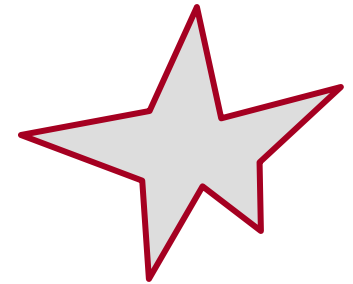
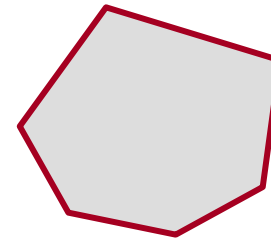
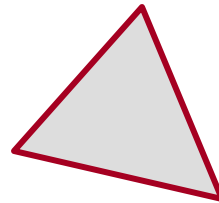
Concave Polygon



Polygon Scan Conversion

Fill pixels inside a polygon

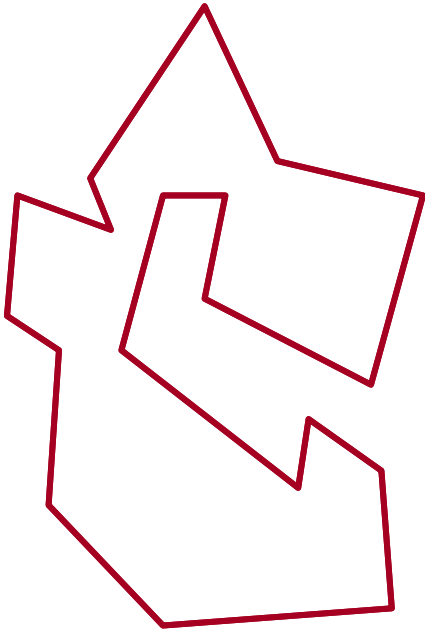
- Triangle
- Convex
- Star-shaped
- Concave
- Self-intersecting
- Holes



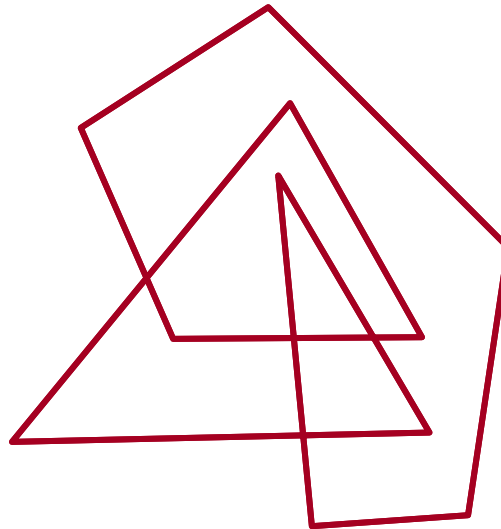


Inside Polygon Rule

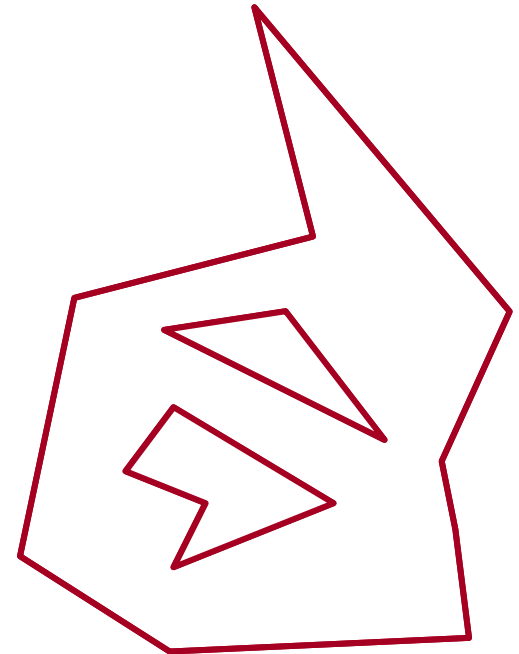
What is a good rule for which pixels are inside?



Concave



Self-Intersecting

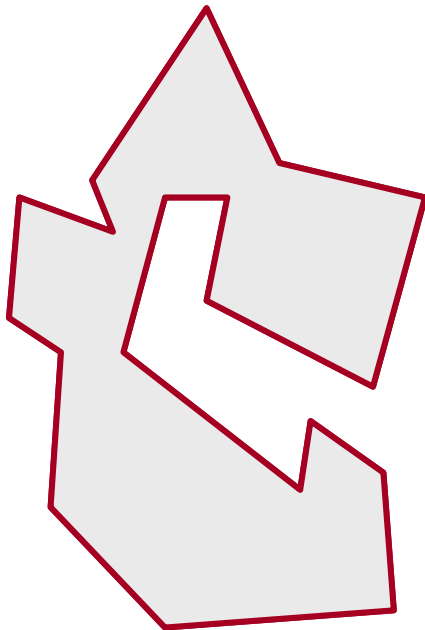


With Holes

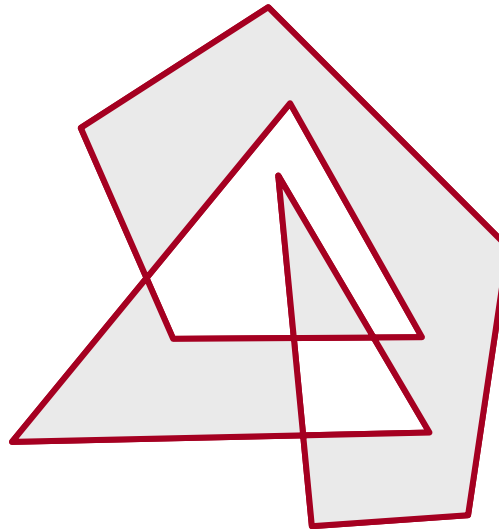


Inside Polygon Rule

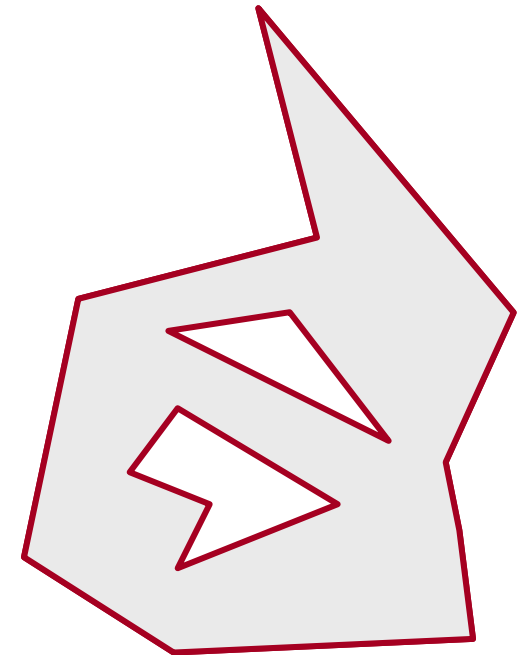
What is a good rule for which pixels are inside?



Concave



Self-Intersecting



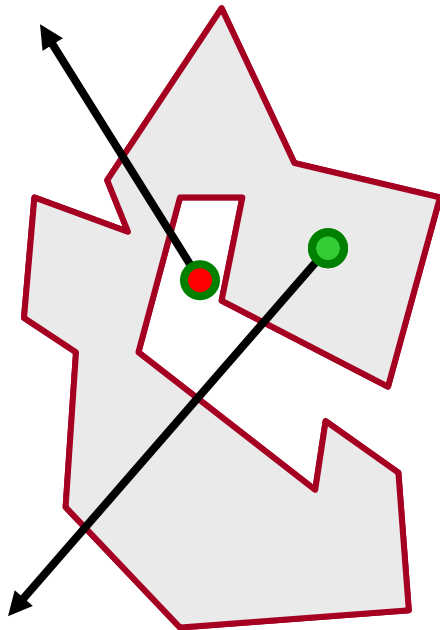
With Holes



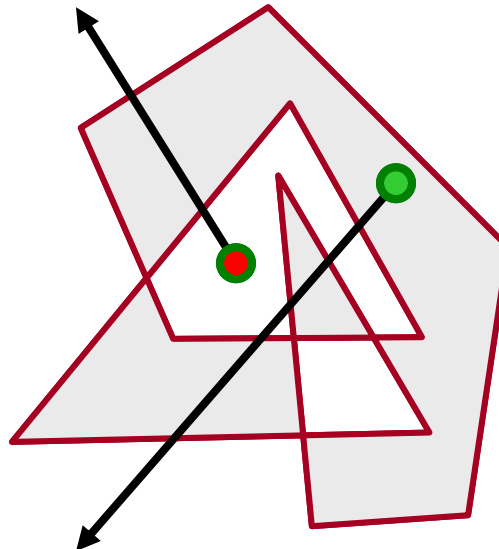
Inside Polygon Rule

Odd-parity rule

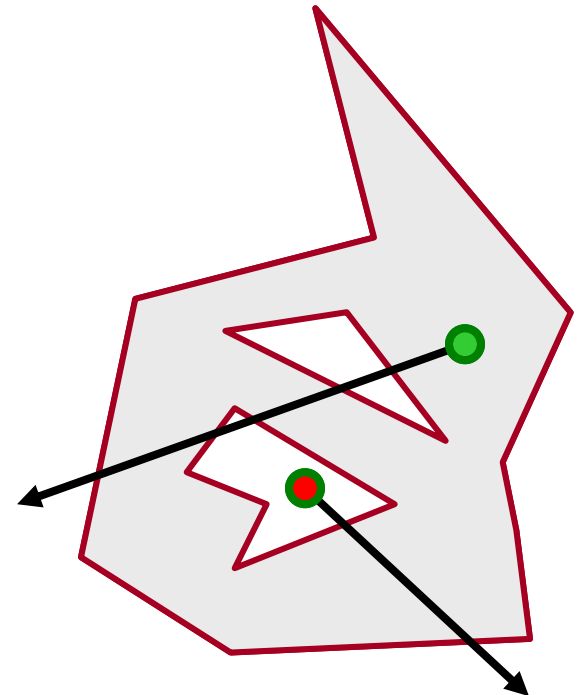
Any ray from inside the shape, out to infinity, must cross an odd number of edges



Concave



Self-Intersecting



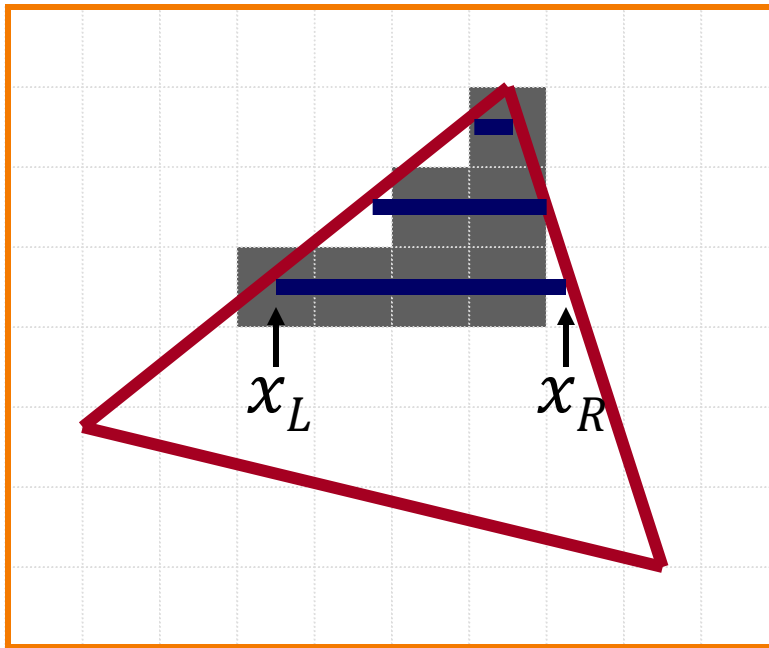
With Holes



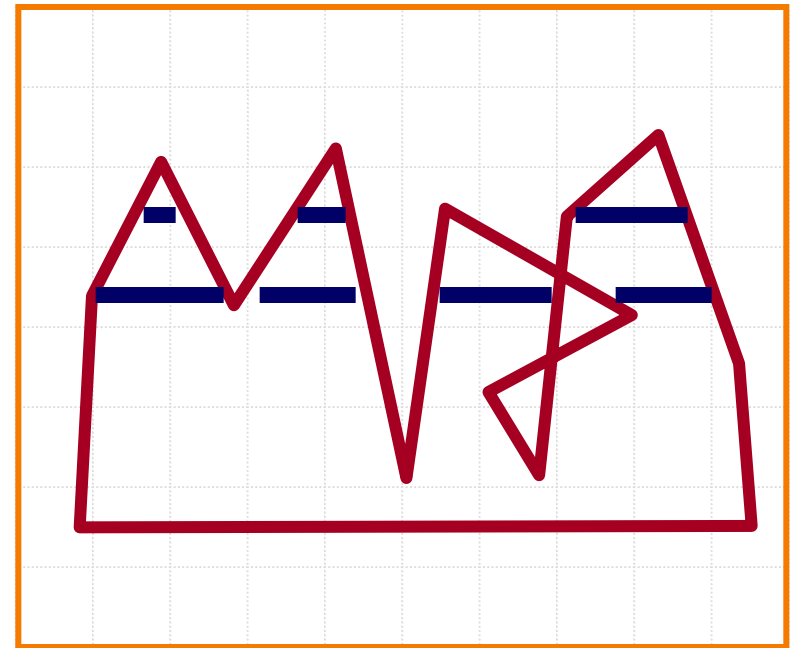
Polygon Sweep-Line Algorithm

- Use incremental algorithm to find spans
- Determine “insideness” with odd (horizontal) parity rule

Takes advantage of scan line coherence



Triangle

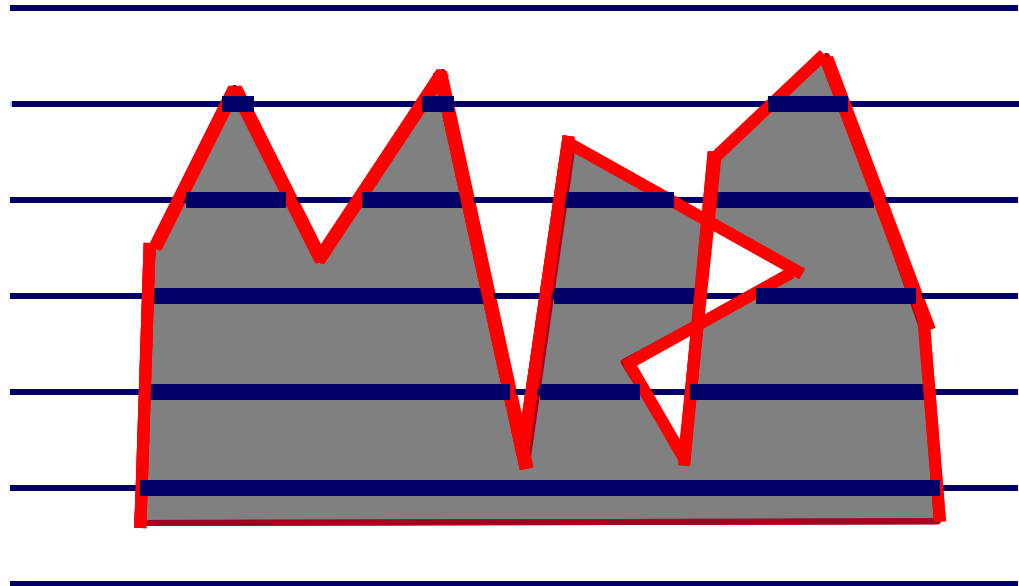


Polygon



Polygon Sweep-Line Algorithm

```
void ScanPolygon( Polygon P , Color rgba )  
{  
    sort edges by maxy  
    initialize sorted active-edge-list as empty  
    for each scanline ( top-to-bottom )  
    {  
        insert/remove/replace edges from active-edge-list  
        maintain sorted active edges intersecting the scanline  
        for each successive pair of edge-points (left-to-right)  
            SetPixels(  $x_{2i}$  ,  $x_{2i+1}$  ,  $y$  , rgba );  
    }  
}
```



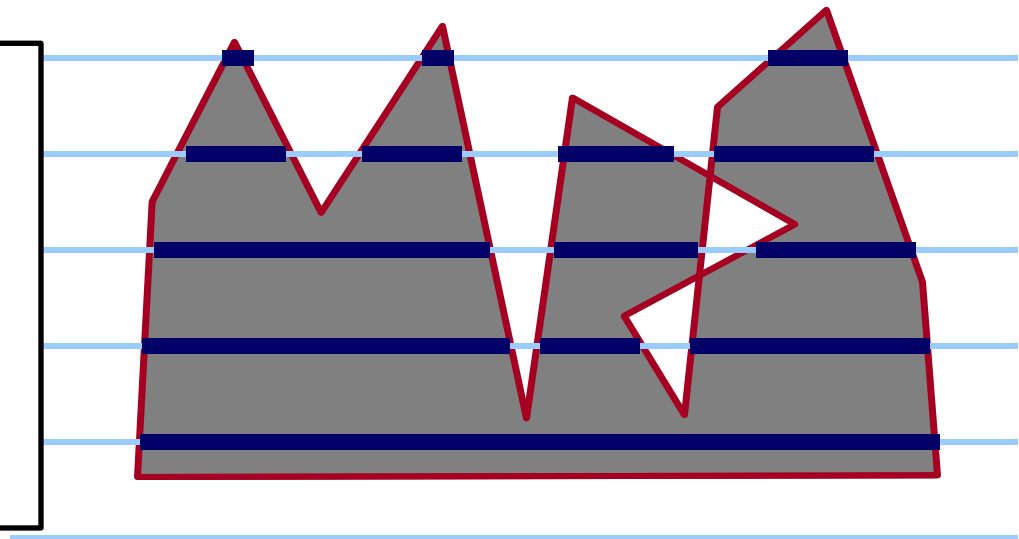


Polygon Sweep-Line Algorithm

```
void ScanPolygon( Polygon P , Color rgba )  
{  
    sort edges by maxy  
    initialize sorted active-edge-list as empty  
    for each scanline ( top-to-bottom )  
    {  
        insert/remove/replace edges from active-edge-list  
        maintain sorted active edges intersecting the scanline  
        for each successive pair of edge-points (left-to-right)  
            SetPixels(  $x_{2i}$  ,  $x_{2i+1}$  ,  $y$  , rgba );  
    }  
}
```

Observation:

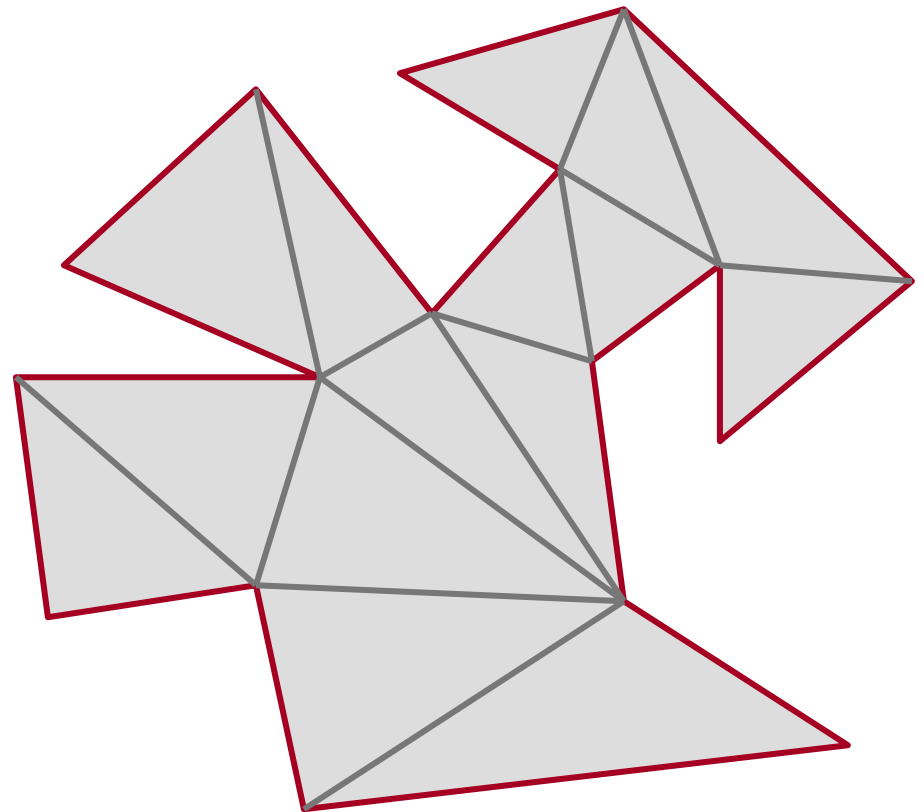
Don't have to do a full sort since ordering only changes when *adjacent* edges intersect.





Polygon Scan Conversion

- Triangulate the polygon
- Scan convert the triangles





Polygon Scan Conversion

Definition:

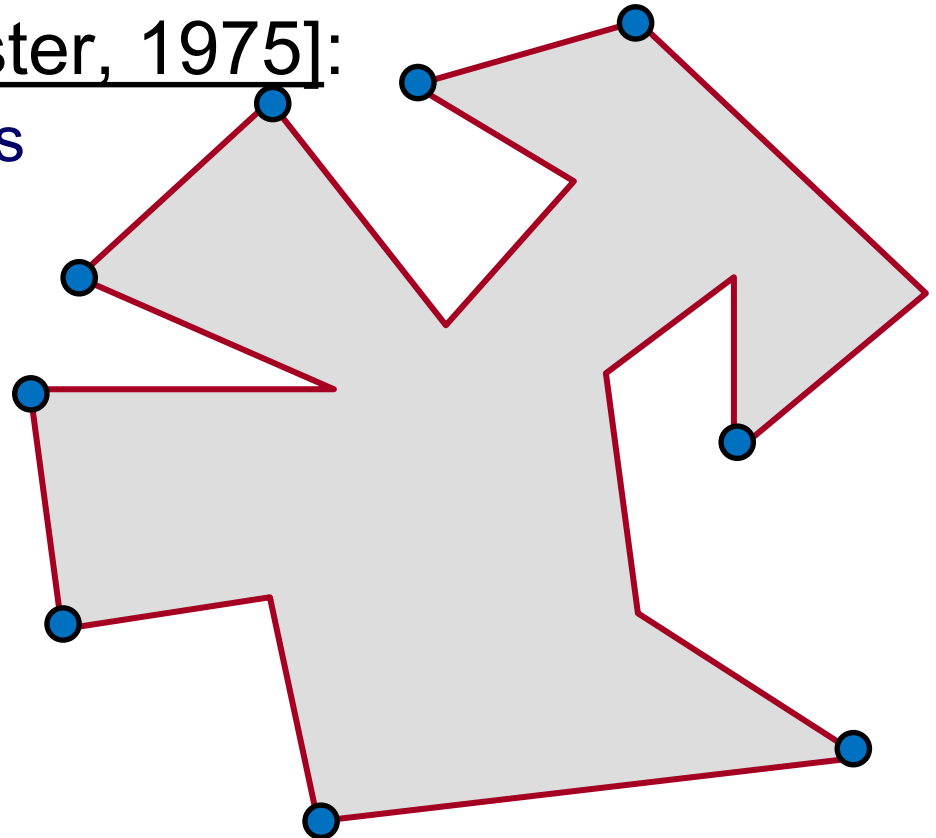
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.

Algorithm:

1. Clip off an ear
2. Repeat





Polygon Scan Conversion

Definition:

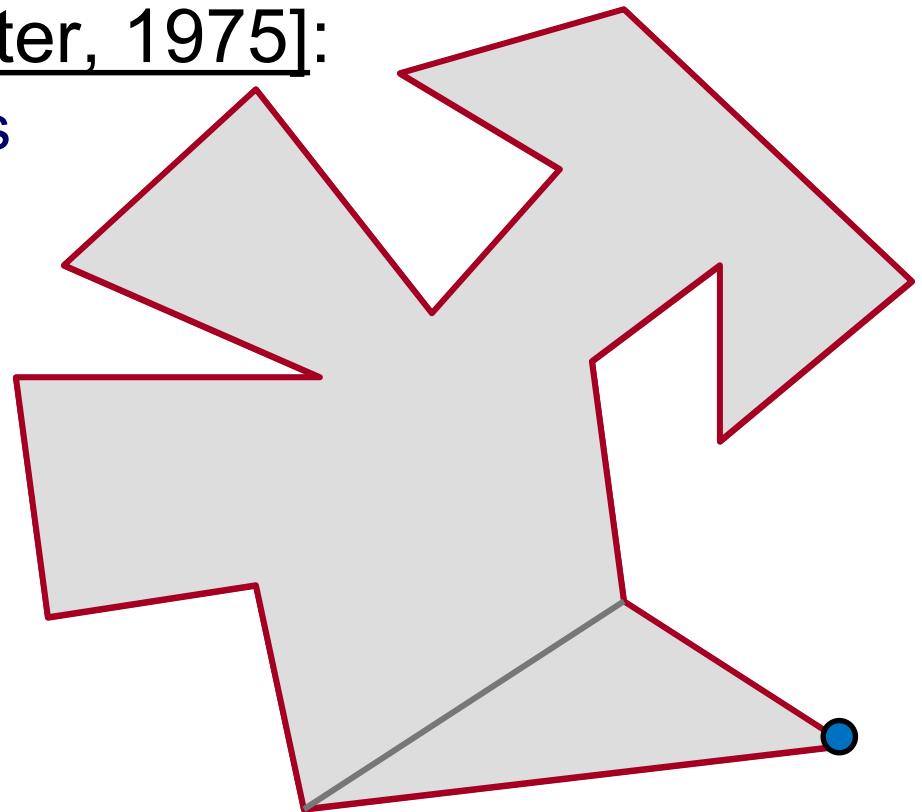
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.

Algorithm:

1. Clip off an ear
2. Repeat





Polygon Scan Conversion

Definition:

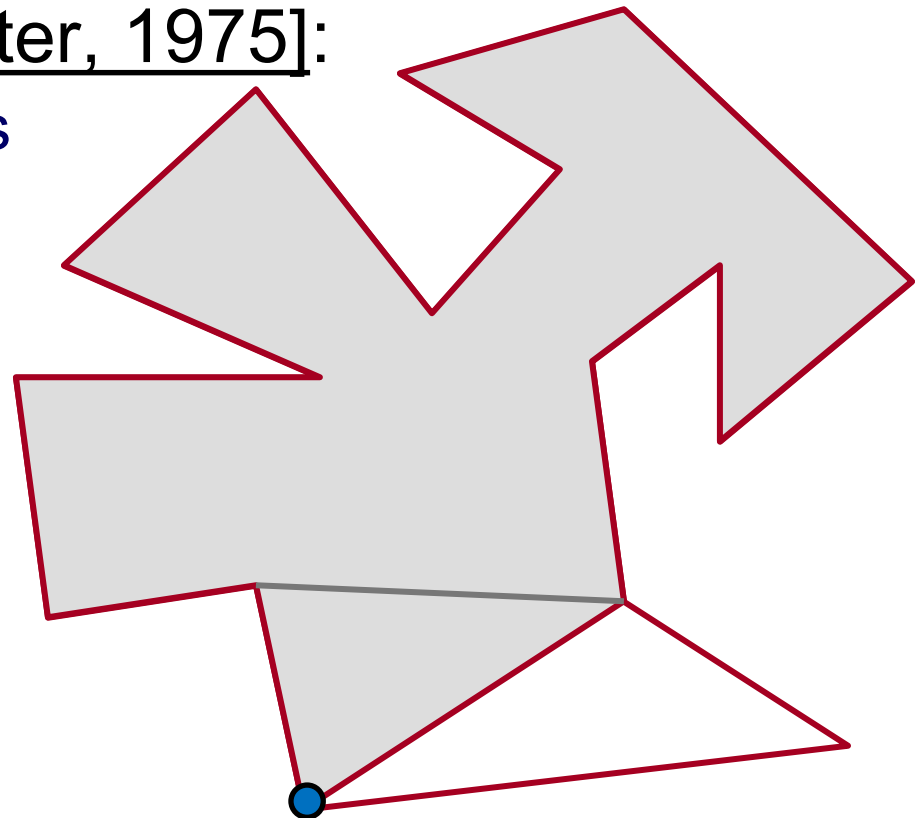
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.

Algorithm:

1. Clip off an ear
2. Repeat





Polygon Scan Conversion

Definition:

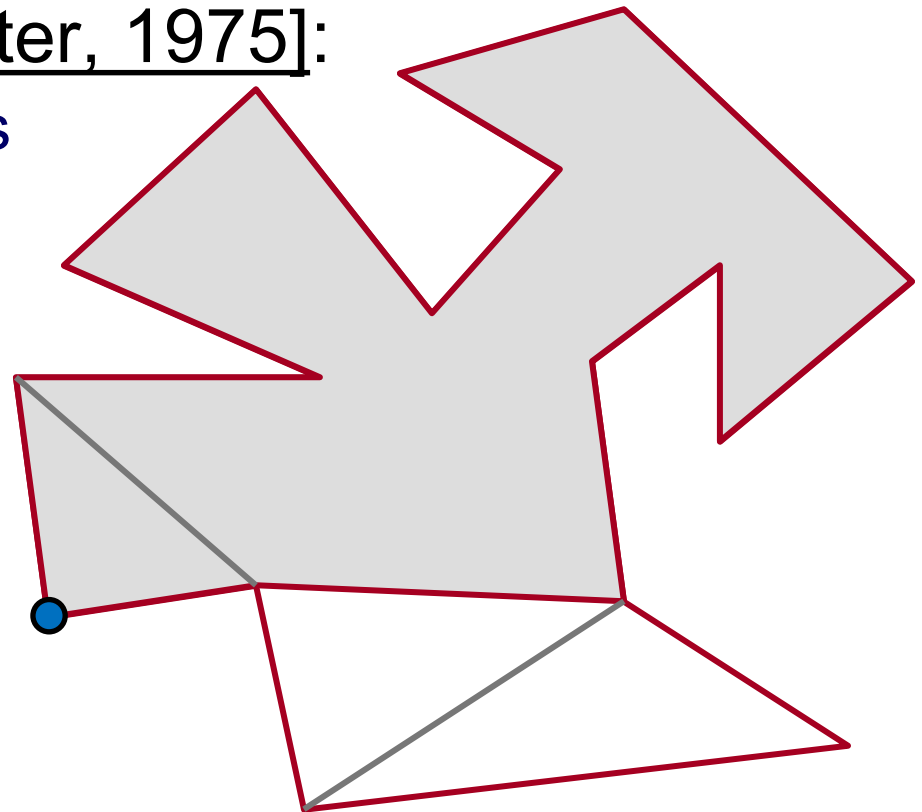
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.

Algorithm:

1. Clip off an ear
2. Repeat





Polygon Scan Conversion

Definition:

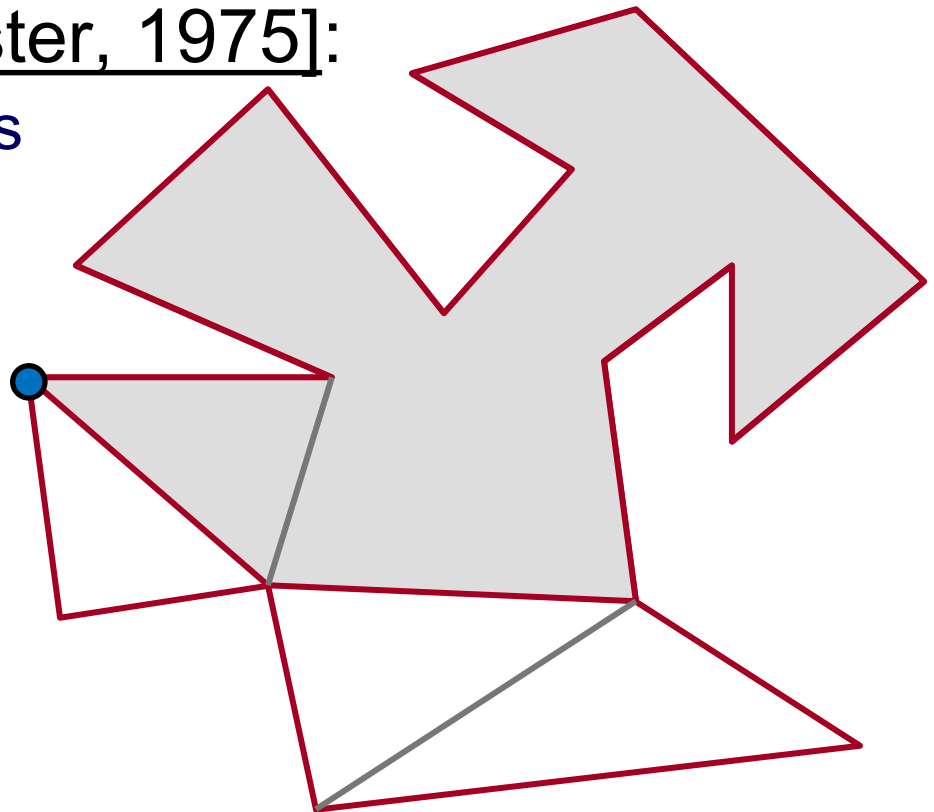
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.

Algorithm:

1. Clip off an ear
2. Repeat





Polygon Scan Conversion

Definition:

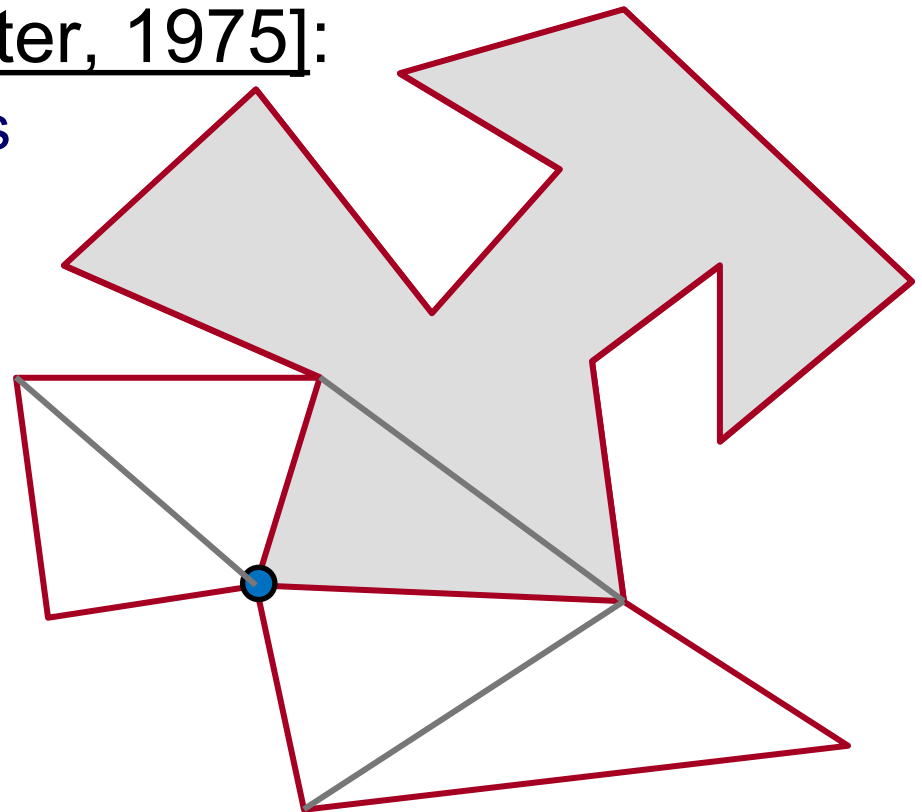
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.

Algorithm:

1. Clip off an ear
2. Repeat





Polygon Scan Conversion

Definition:

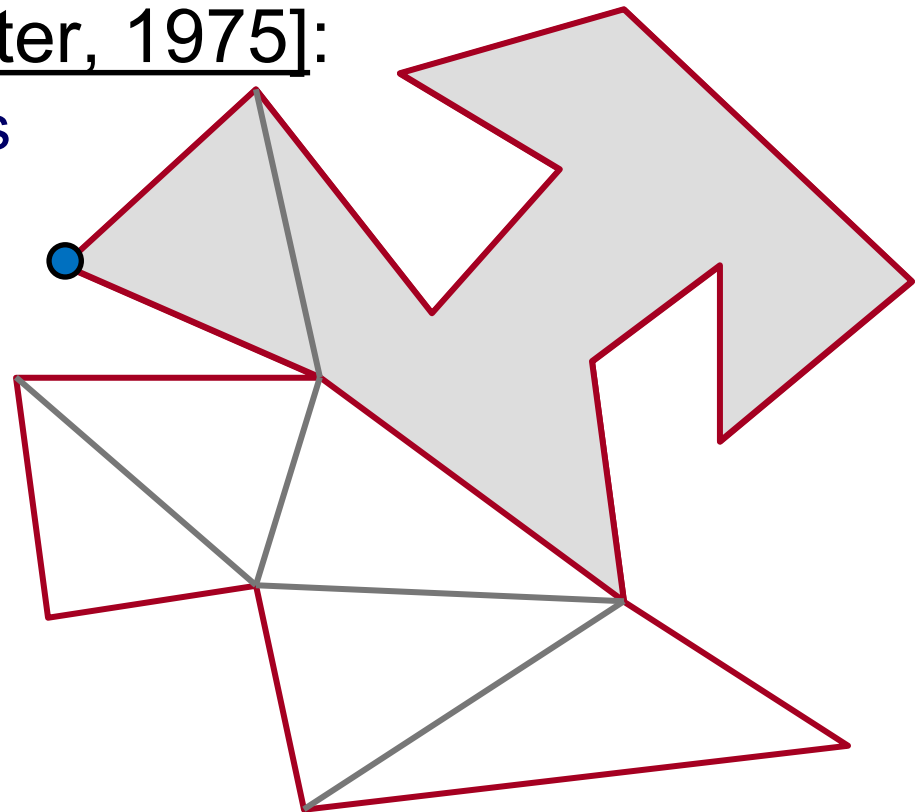
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.

Algorithm:

1. Clip off an ear
2. Repeat





Polygon Scan Conversion

Definition:

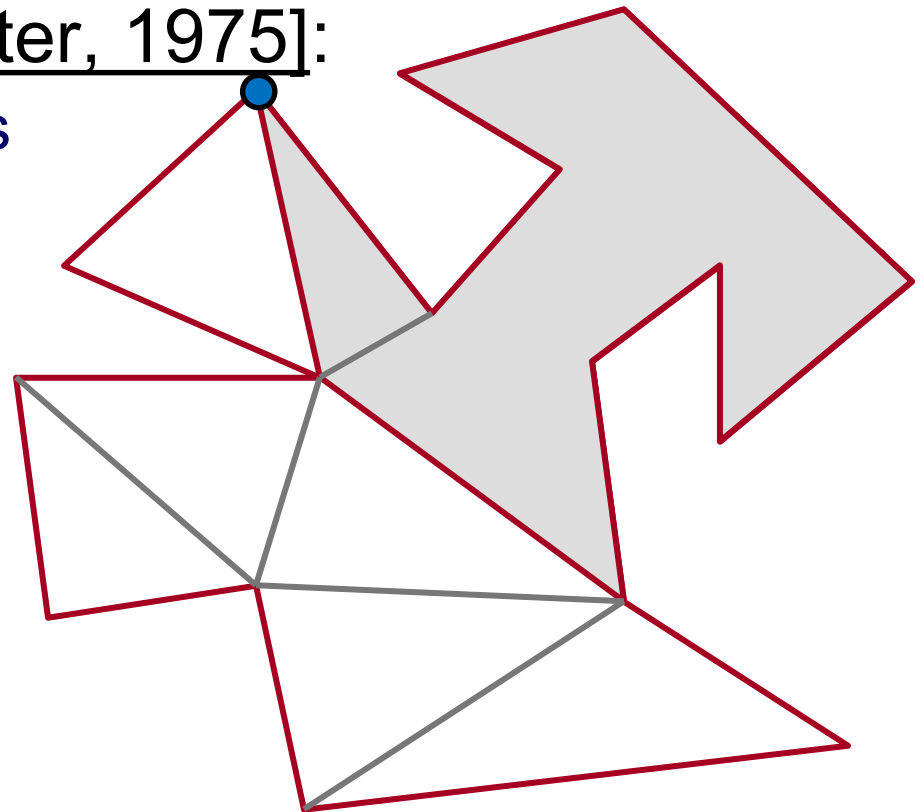
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.

Algorithm:

1. Clip off an ear
2. Repeat





Polygon Scan Conversion

Definition:

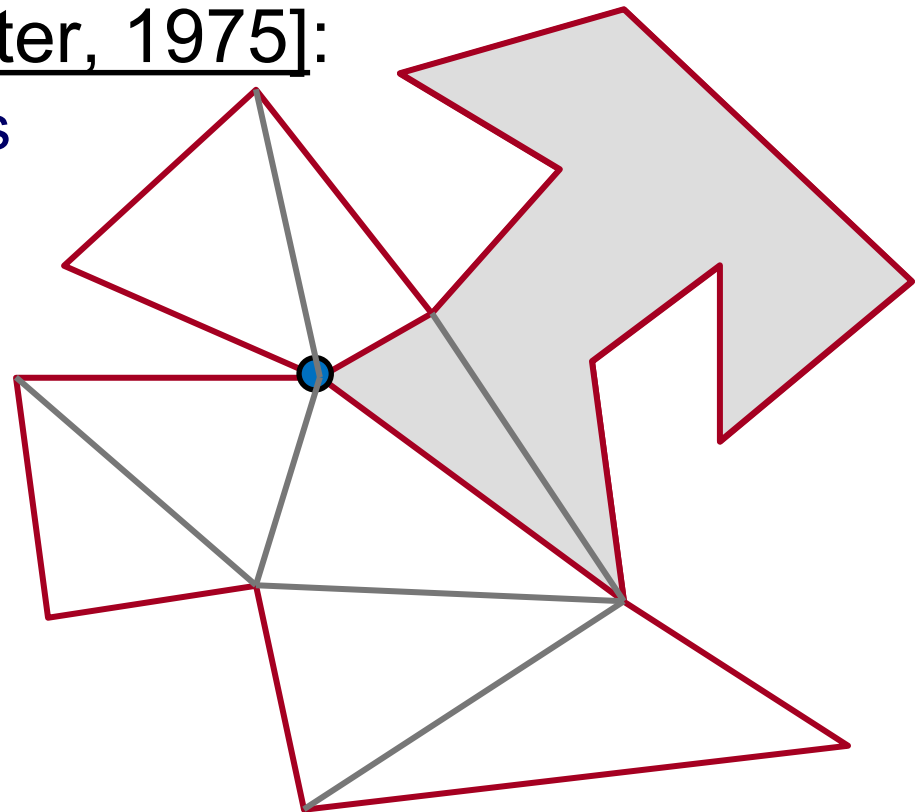
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.

Algorithm:

1. Clip off an ear
2. Repeat





Polygon Scan Conversion

Definition:

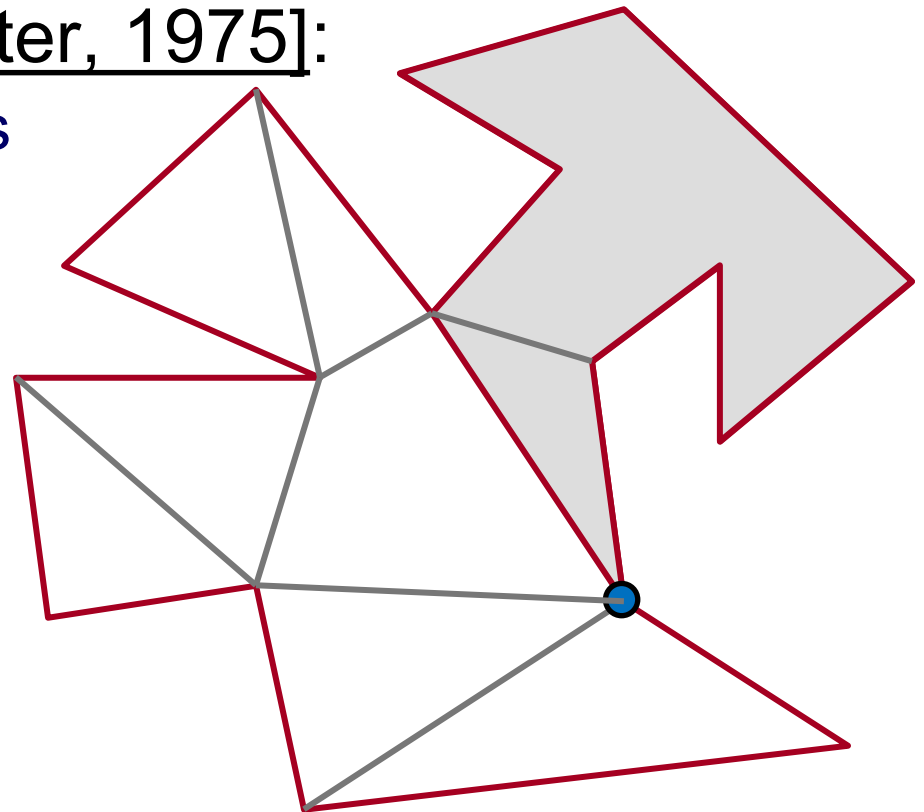
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.

Algorithm:

1. Clip off an ear
2. Repeat





Polygon Scan Conversion

Definition:

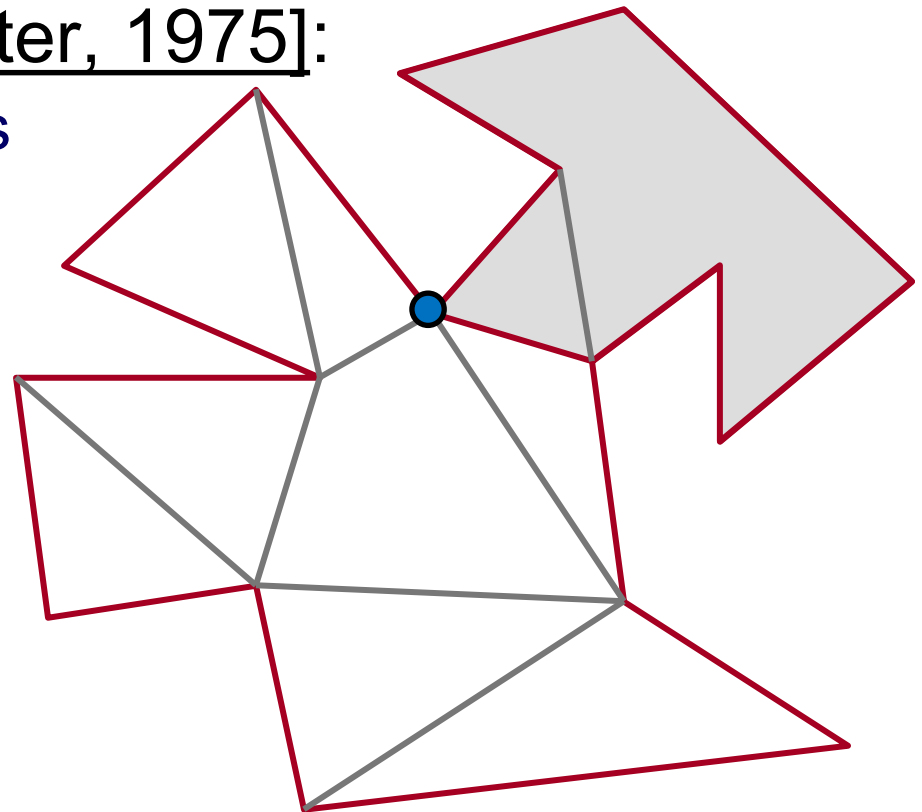
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.

Algorithm:

1. Clip off an ear
2. Repeat





Polygon Scan Conversion

Definition:

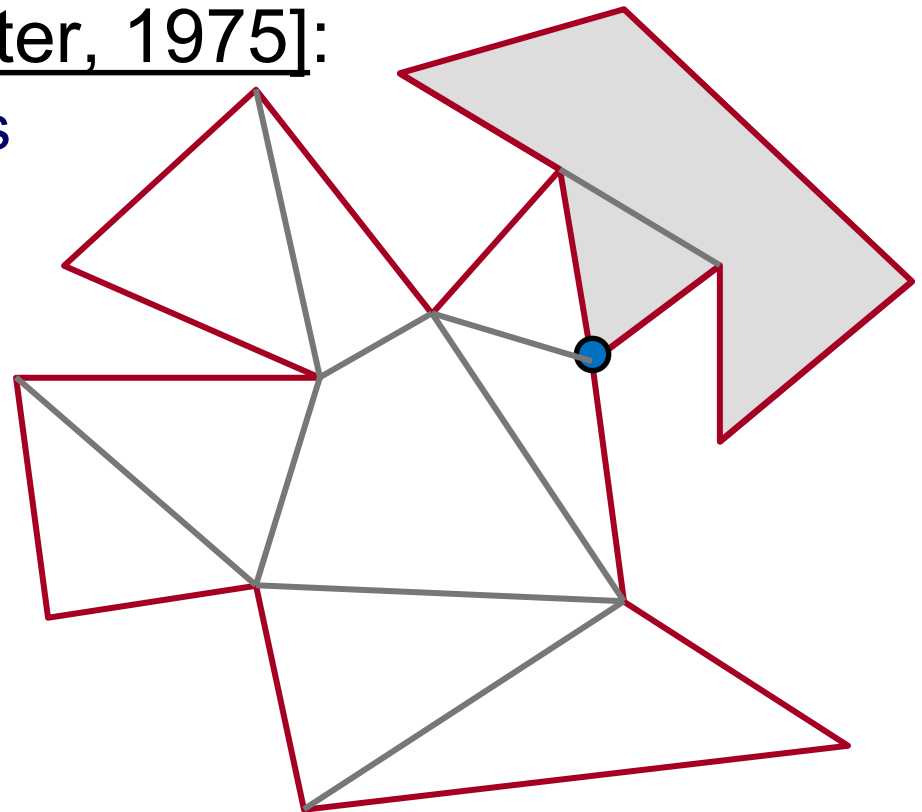
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.

Algorithm:

1. Clip off an ear
2. Repeat





Polygon Scan Conversion

Definition:

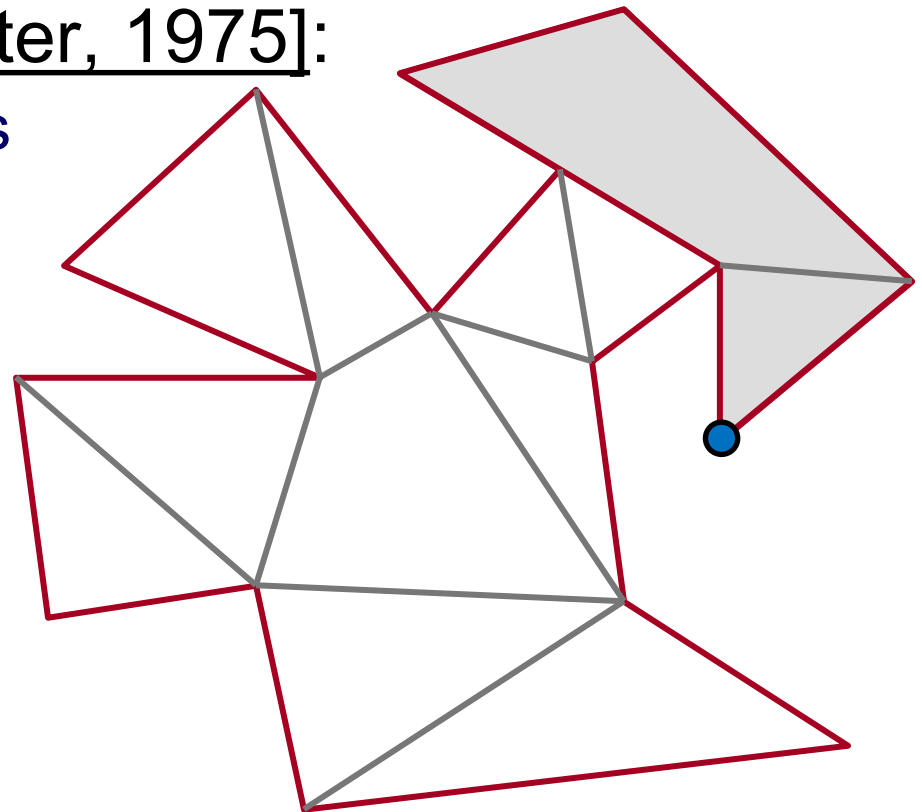
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.

Algorithm:

1. Clip off an ear
2. Repeat





Polygon Scan Conversion

Definition:

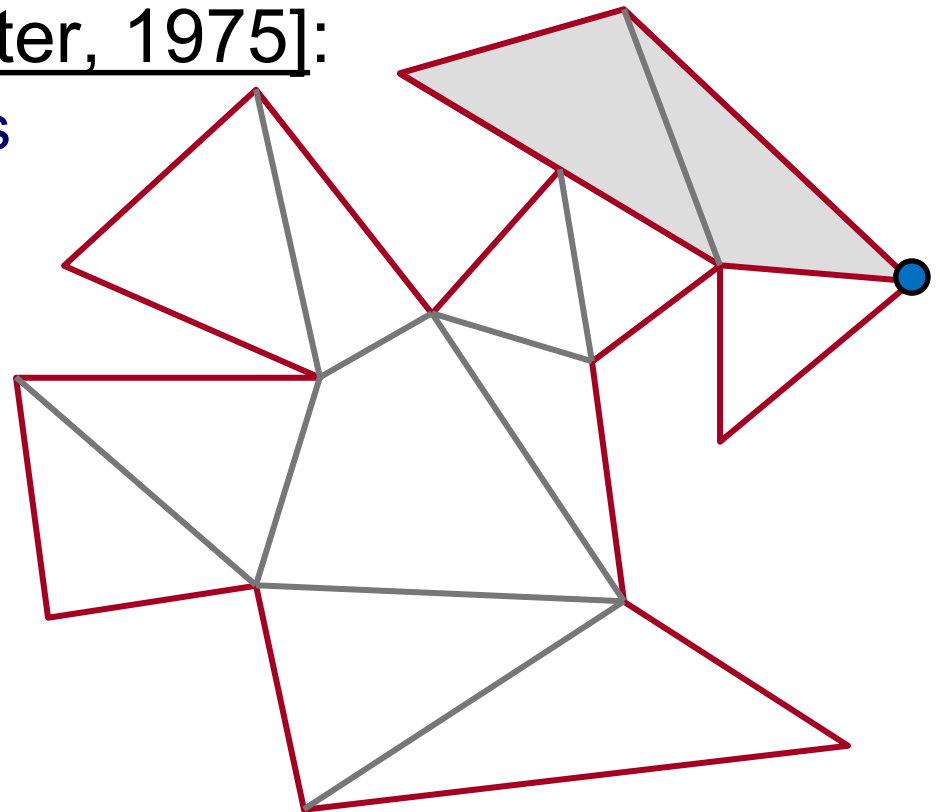
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.

Algorithm:

1. Clip off an ear
2. Repeat





Polygon Scan Conversion

Definition:

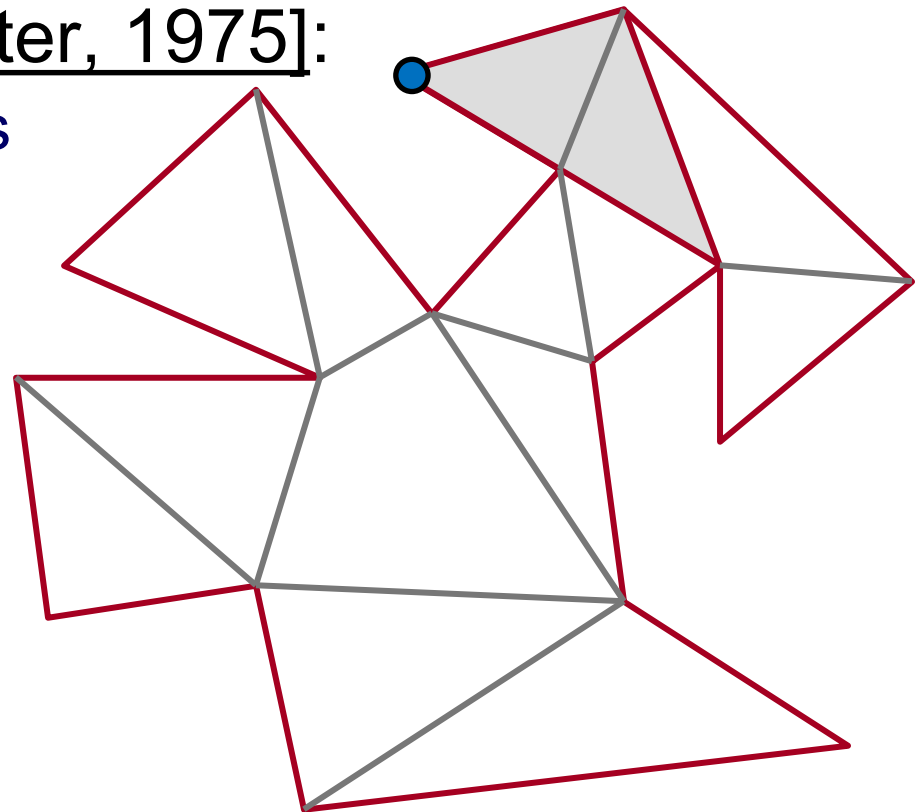
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.

Algorithm:

1. Clip off an ear
2. Repeat





Polygon Scan Conversion

Definition:

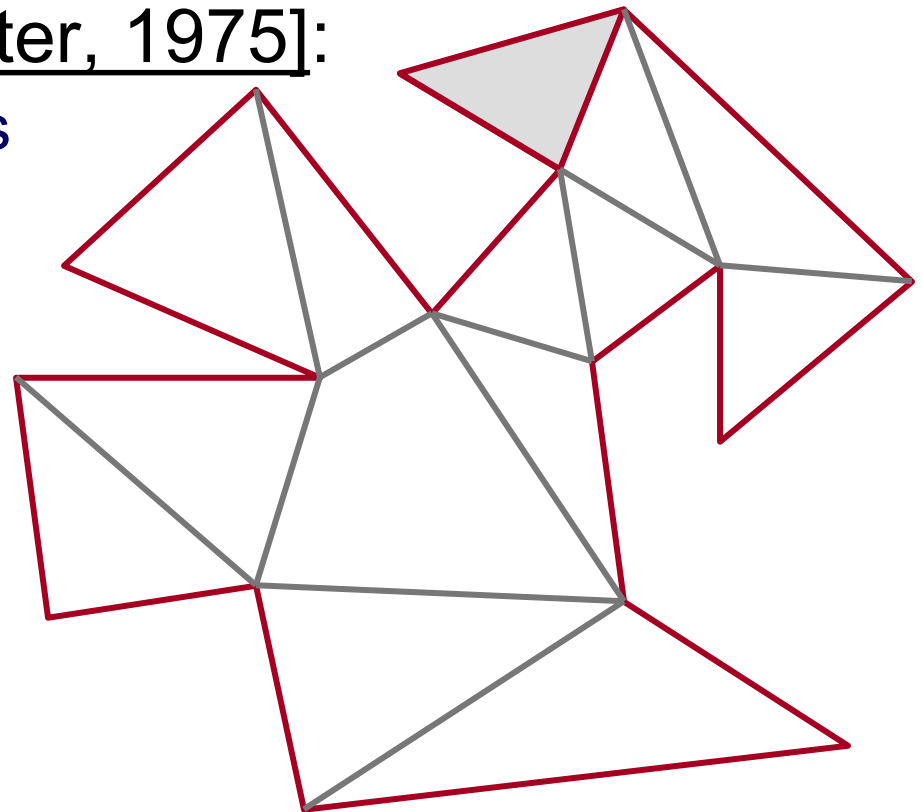
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.

Algorithm:

1. Clip off an ear
2. Repeat





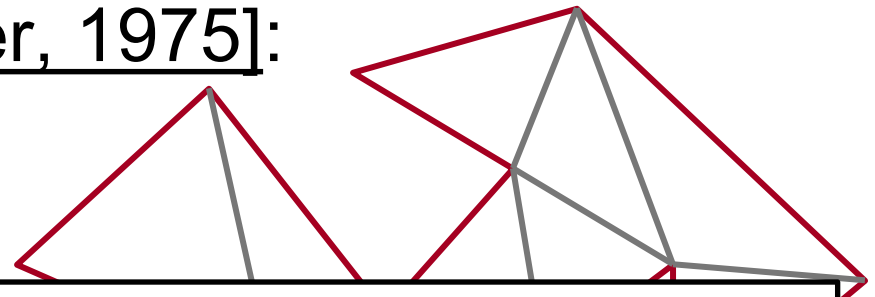
Polygon Scan Conversion

Definition:

A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.



Algo Note:

1. OpenGL does render polygons, but it assumes that
2. the polygon is *planar* and *convex*.

Recall:

Even if you only pass in triangles for rendering, OpenGL may still have to render polygons after the triangle is clipped.

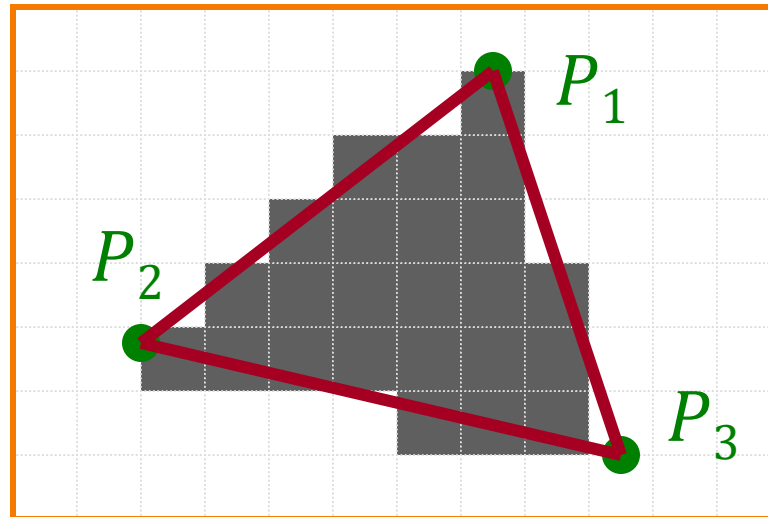
These are guaranteed to be *planar* and *convex*.



Scan Conversion

What about pixels on edges?

- If we set them either “on” or “off” we get aliasing or “jaggies” (similar to using nearest interpolation)



Scan Conversion



Example:



No Anti-Aliasing

Image courtesy of NVIDIA



Antialiasing Techniques

Display at higher resolution

Corresponds to increasing sampling rate

✖ Not always possible (fixed size monitors, etc.)

Modify pixel intensities

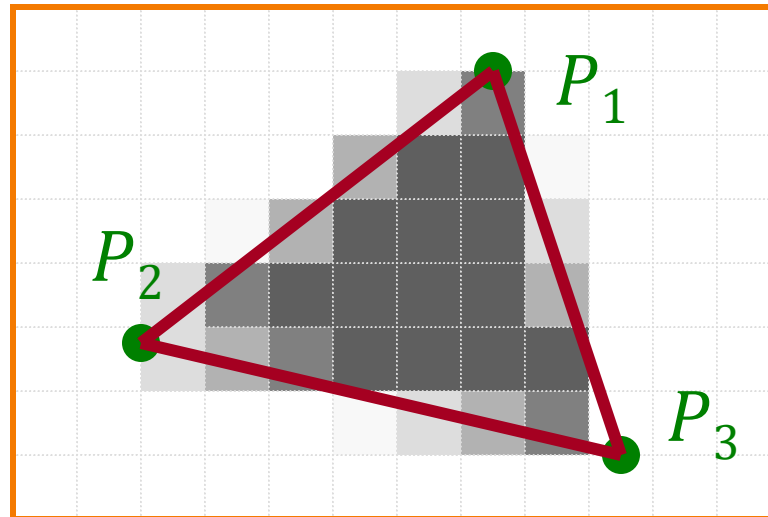
Vary pixel intensities along boundaries for antialiasing



Scan Conversion

What about pixels on edges?

- ✗ Setting them either “on” or “off” we get aliasing/“jaggies”
- ✓ **Antialias** by varying pixel intensities along boundaries

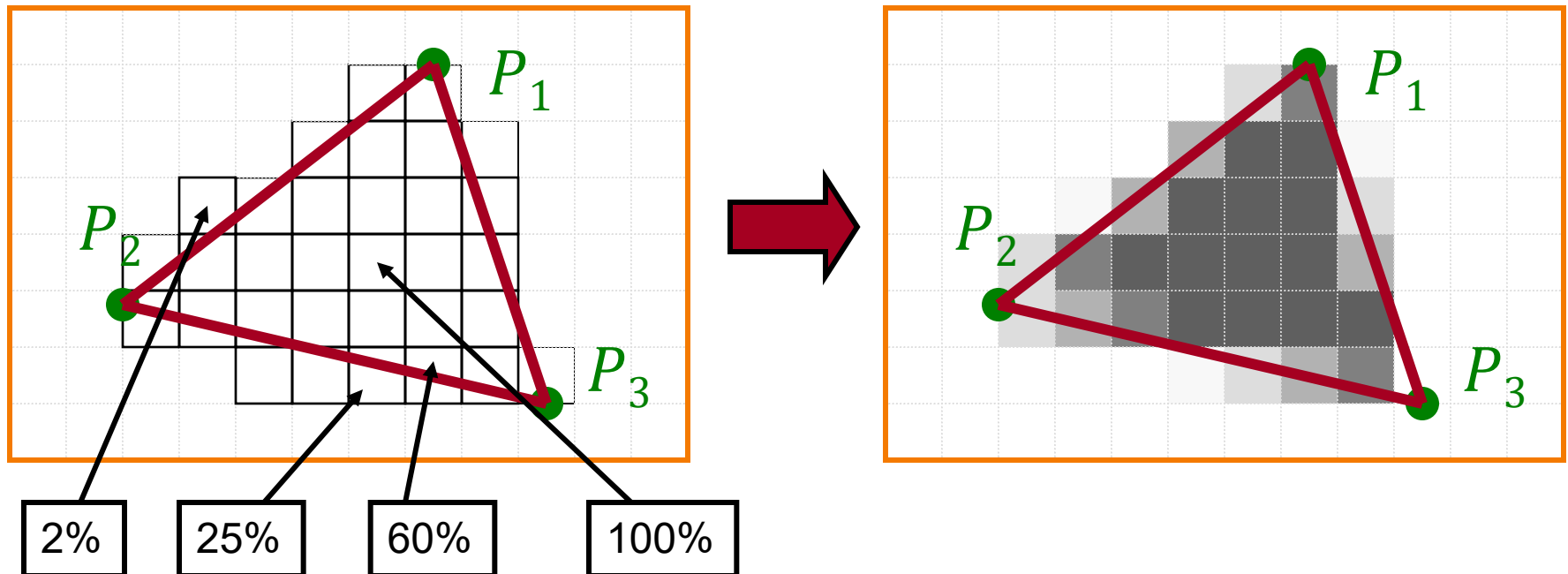




Antialiasing

Ideally – area sampling

- Calculate percent of pixel covered by primitive
- Multiply this percentage by desired intensity/color

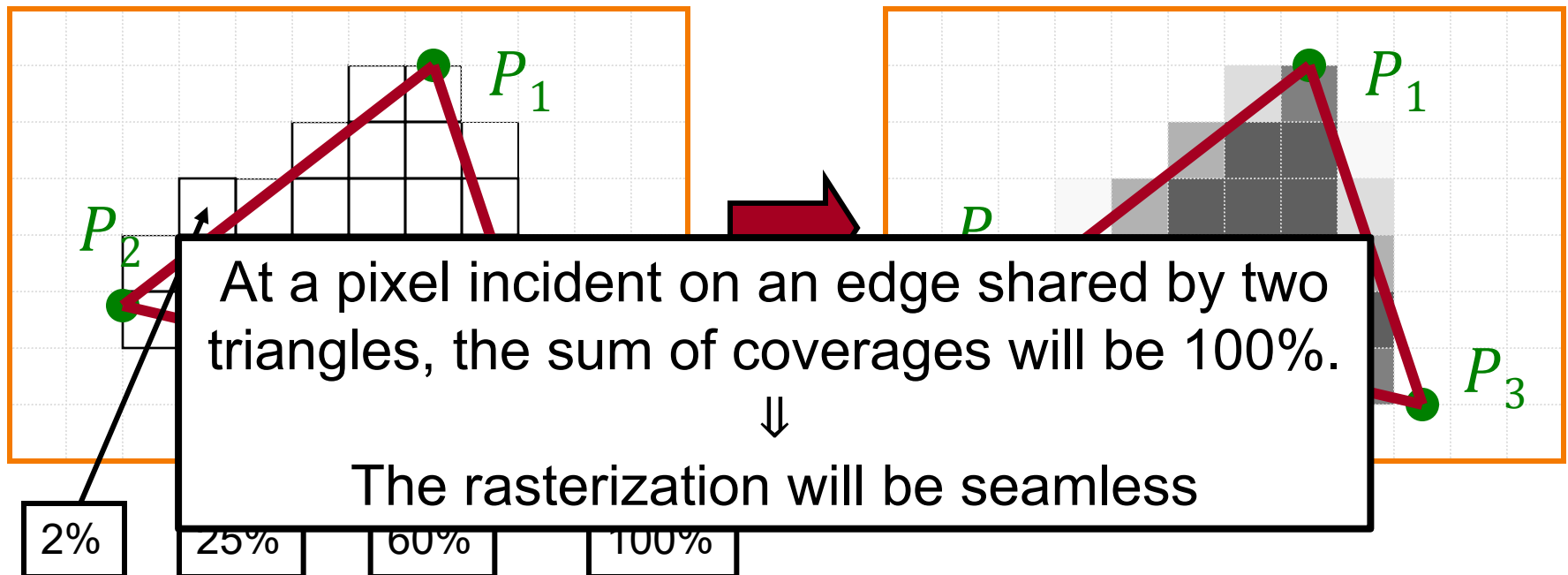




Antialiasing

Ideally – area sampling

- Calculate percent of pixel covered by primitive
- Multiply this percentage by desired intensity/color



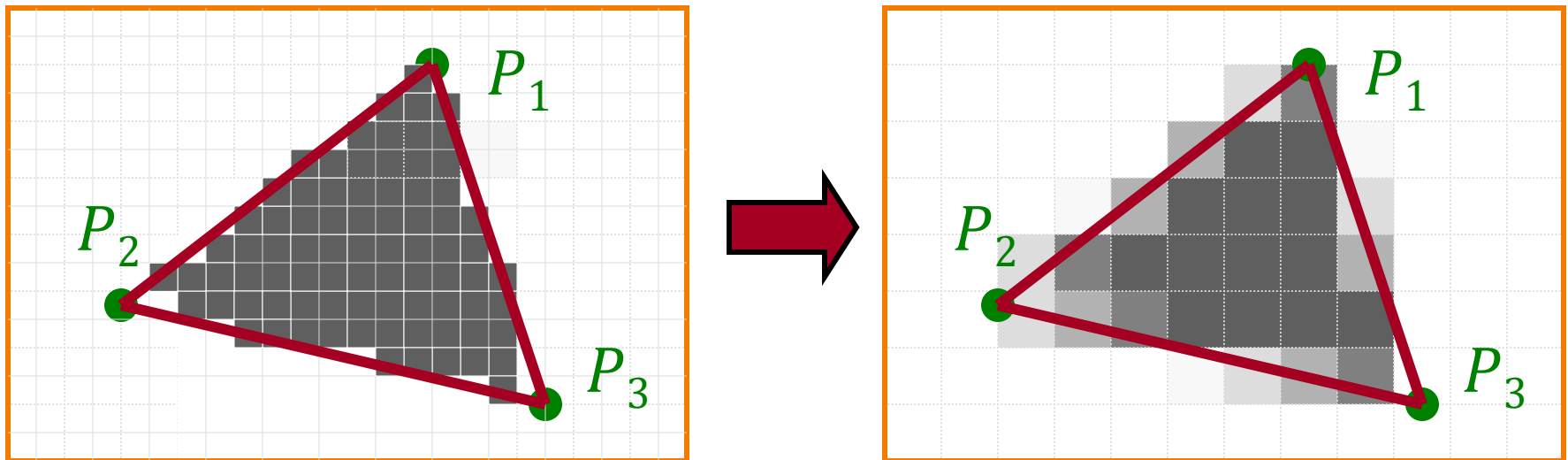


Antialiasing

In practice – supersampling (aka postfiltering)

- Sample as if screen were higher resolution
- Average multiple samples to get final intensity

This is done by rendering the scene multiple times with different (fractional) offsets and averaging



» The fractional value will be at a granularity determined by the number of rendering passes.

Scan Conversion



Example:



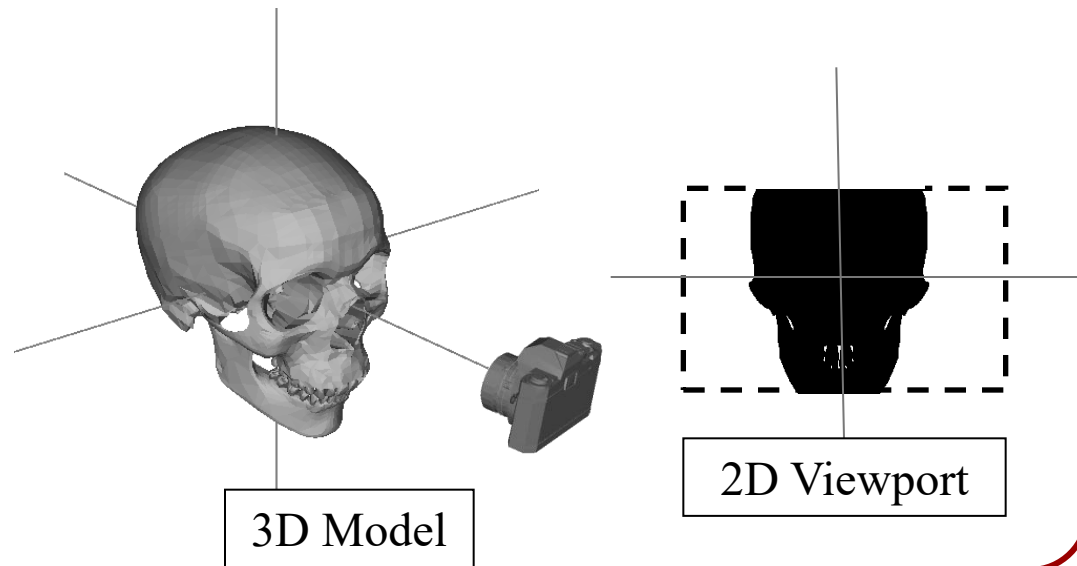
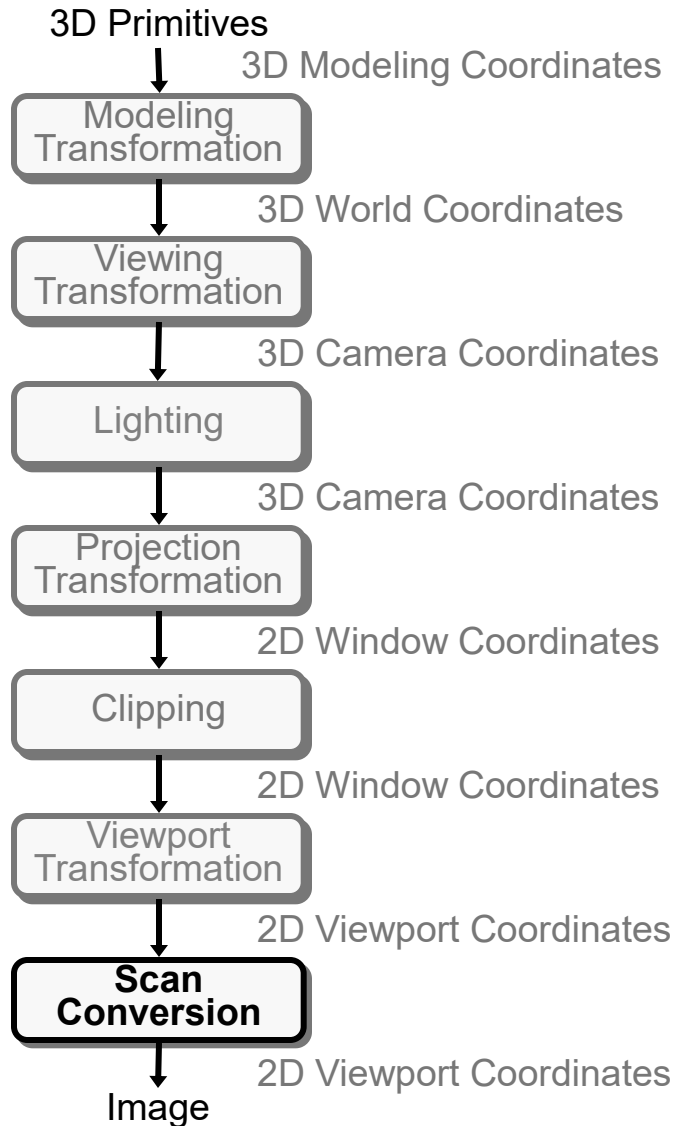
No Anti-Aliasing



4 x Anti-Aliasing

Images courtesy of NVIDIA

3D Rendering Pipeline (for direct illumination)





Overview

Scan conversion

Figure out which pixels to fill

Shading

Determine the color for each filled pixel

Depth test

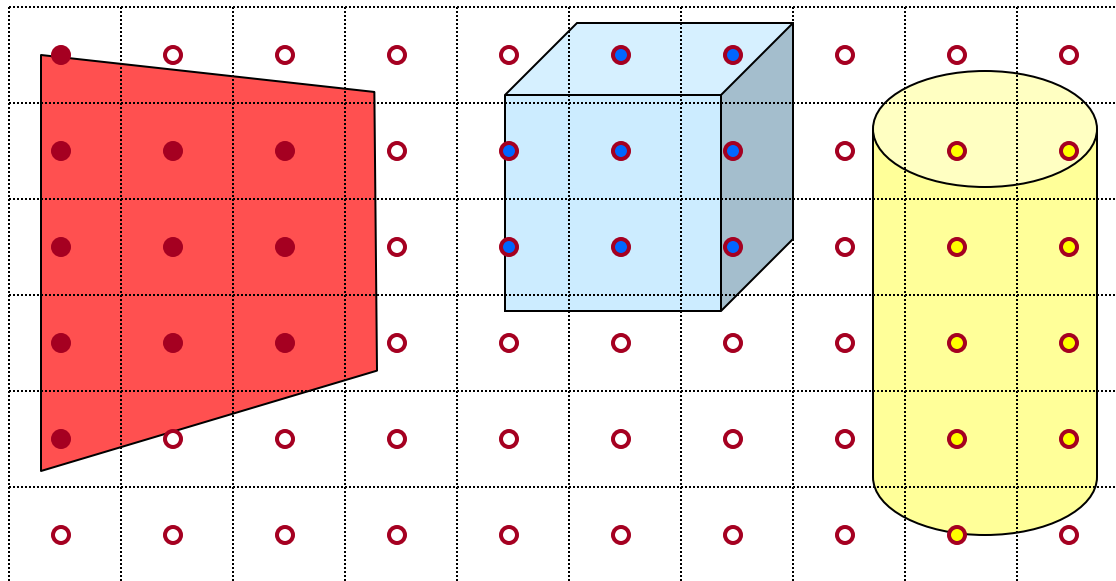
Determine when the color of a pixel comes from the front-most primitive



Polygon Shading

Take advantage of spatial coherence

Illumination calculations for pixels covered by same primitive are related to each other



$$I = I_E + \sum_L \left[K_A \cdot I_L^A + \left(K_D \cdot \langle \vec{N}, \vec{L} \rangle + K_S \cdot \langle \vec{V}, \vec{R}(\vec{L}) \rangle^{K_n} \right) \cdot I_L \right]$$

Polygon Shading Algorithms



Shading models:

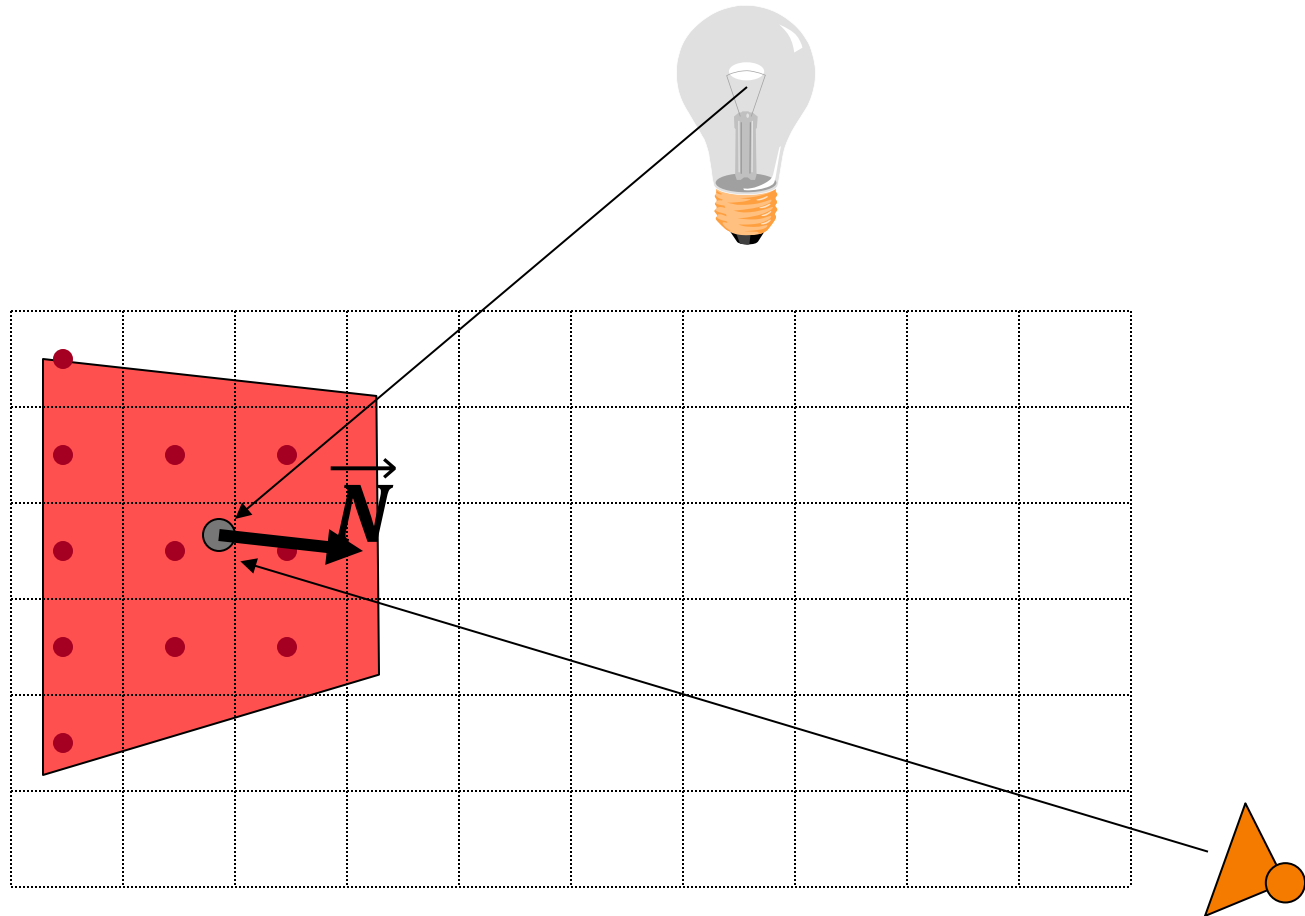
- Flat
- Gouraud
- Phong



Flat Shading

One lighting calculation **per polygon**

Assign all pixels inside each polygon the same color





Flat Shading

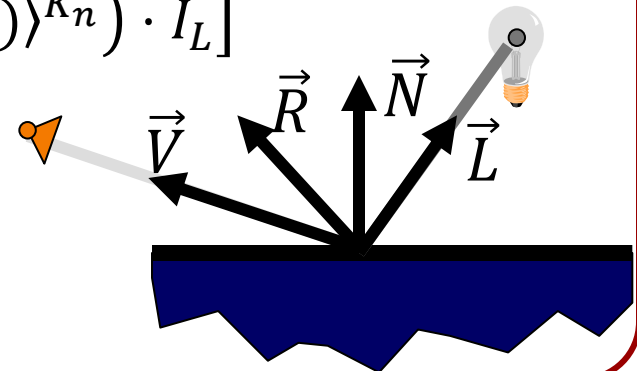
Take advantage of spatial coherence

Make the lighting equation constant
over the surface of each primitive

	Surface Normal	Light Direction	View Direction
Emissive	-	-	-
Ambient	-	-	-
Diffuse	+	+	-
Specular	+	+	+

$$I = I_E + \sum_L [K_A \cdot I_L^A + (K_D \cdot \langle \vec{N}, \vec{L} \rangle + K_S \cdot \langle \vec{V}, \vec{R}(\vec{L}) \rangle^{K_n}) \cdot I_L]$$

Assume that the emissive light, I_E , as well as material properties, K_A , K_D , K_S , and K_n , are constant across the triangle.





Flat Shading

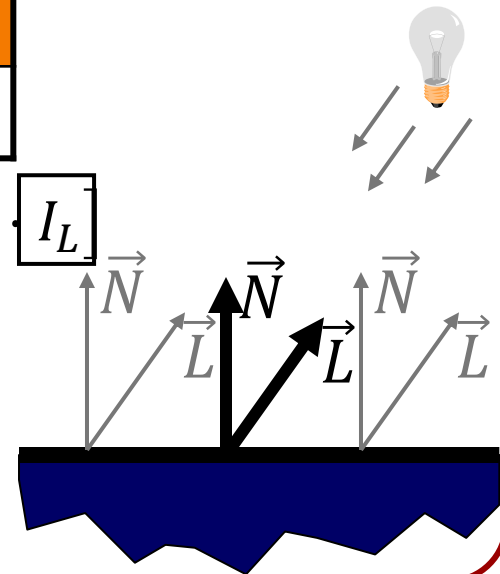
Take advantage of spatial coherence

- If the normal is constant over the primitive, and
- If the light is directional,
⇒ The diffuse component is the same for all points on the primitive

	Surface Normal	Light Direction	View Direction
Emissive	-	-	-
Ambient	-	-	-
Diffuse	+	+	-
Specular	+	+	+

$$I = I_E + \sum_L [K_A \cdot I_L^A + (K_D \cdot \langle \vec{N}, \vec{L} \rangle + K_S \cdot \langle \vec{V}, \vec{R}(\vec{L}) \rangle^{K_n}) \cdot I_L]$$

Assume that the emissive light, I_E , as well as material properties, K_A , K_D , K_S , and K_n , are constant across the triangle.





Flat Shading

Take advantage of spatial coherence

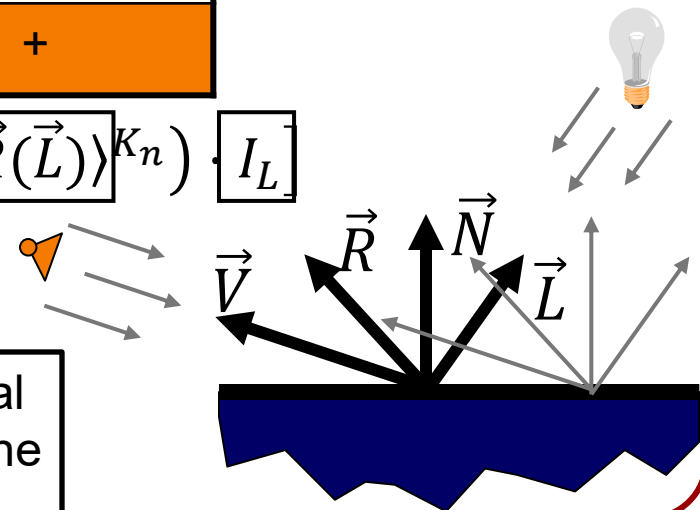
- If the normal is constant over the primitive, and
 - If the light is directional,
- ⇒ The diffuse component is the same for all points on the primitive

	Surface Normal	Light Direction	View Direction
--	----------------	-----------------	----------------

Emission	• If the normal is constant over the primitive, • If the light is directional, and • If the direction to the viewer is constant over the primitive ⇒ The specular component is the same for all points on the primitive		
Ambient			
Diffuse			
Specular	+	+	+

$$I = I_E + \sum_L [K_A \cdot I_L^A + (K_D \cdot \langle \vec{N}, \vec{L} \rangle + K_S \cdot \langle \vec{V}, \vec{R}(\vec{L}) \rangle^{K_n}) \cdot I_L]$$

Assume that the emissive light, I_E , as well as material properties, K_A , K_D , K_S , and K_n , are constant across the triangle.

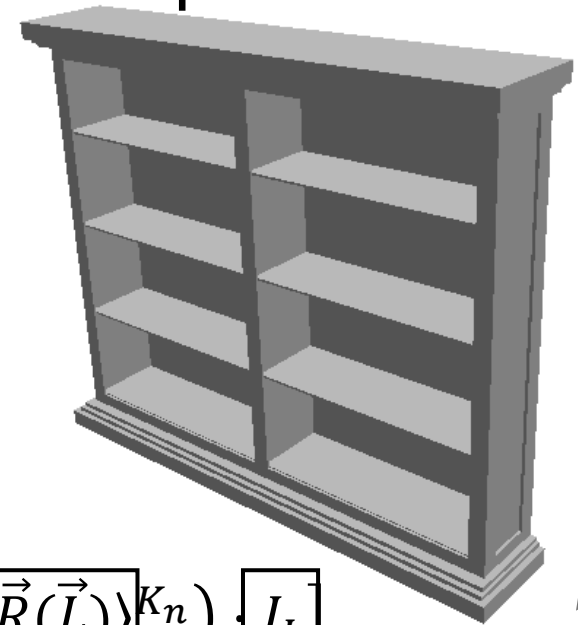




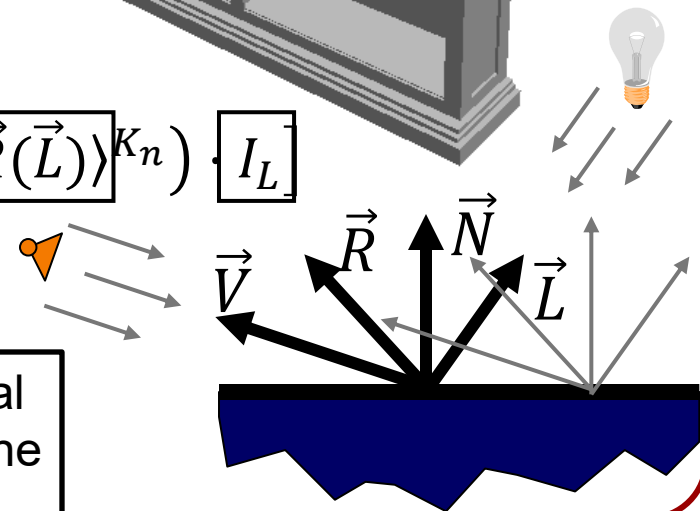
Flat Shading

Illuminates as though the lights are directional, the polygon is flat, and the camera uses parallel projection

- $\langle \vec{N}, \vec{L} \rangle$ constant over surface
- $\langle \vec{V}, \vec{R}(\vec{L}) \rangle$ constant over surface
- I_L constant over surface



$$I = I_E + \sum_L [K_A \cdot I_L^A + (K_D \cdot \langle \vec{N}, \vec{L} \rangle + K_S \cdot \langle \vec{V}, \vec{R}(\vec{L}) \rangle^{K_n}) \cdot I_L]$$



Assume that the emissive light, I_E , as well as material properties, K_A , K_D , K_S , and K_n , are constant across the triangle.

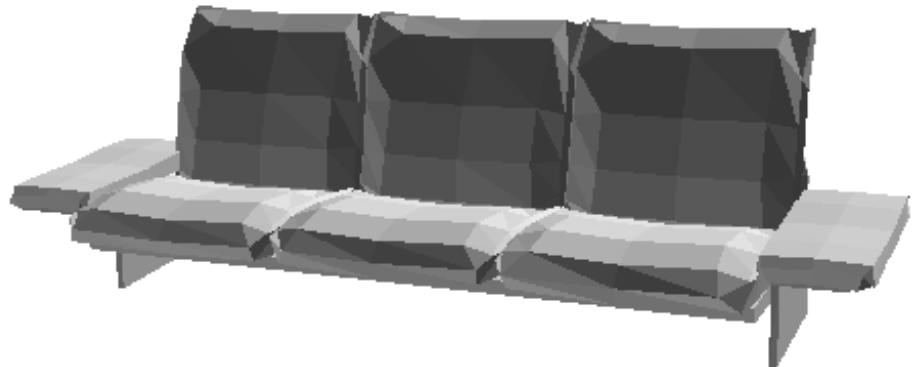
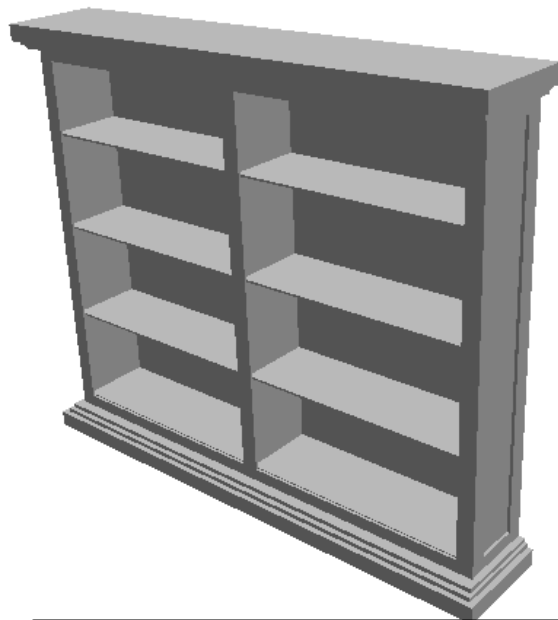


Flat Shading

Objects look like they are composed of polygons

OK for faceted objects

Not so good for smooth surfaces



Although this is the “simplest” lighting model, it is tricky to implement this on the graphics card.

Polygon Shading Algorithms



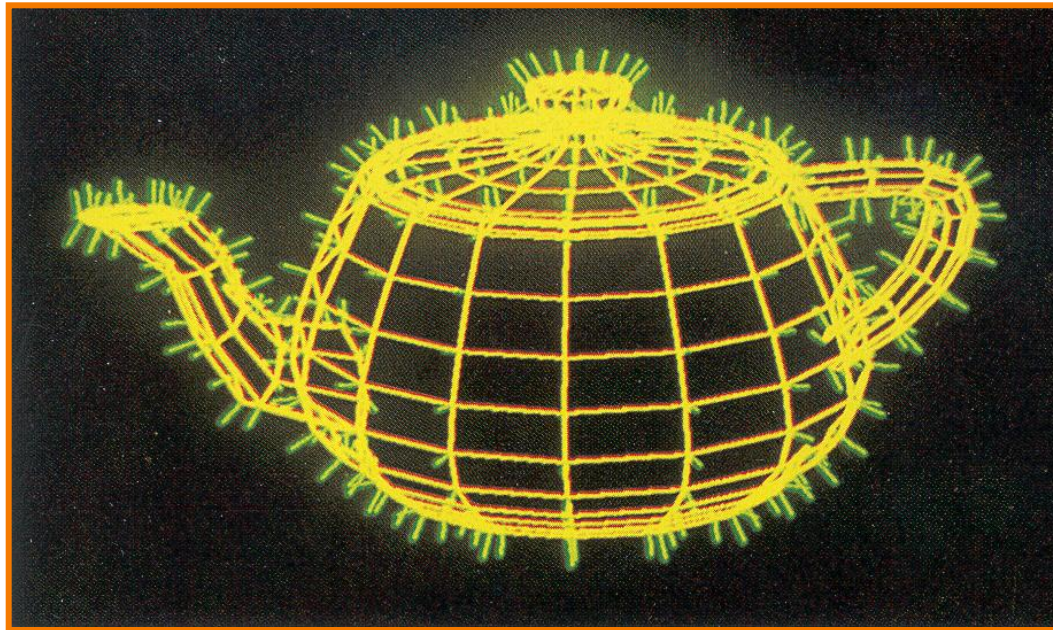
Shading models:

- Flat
- Gouraud
- Phong



Gouraud Shading

Represent a polygonal mesh with a normal at each vertex



Watt Plate 7

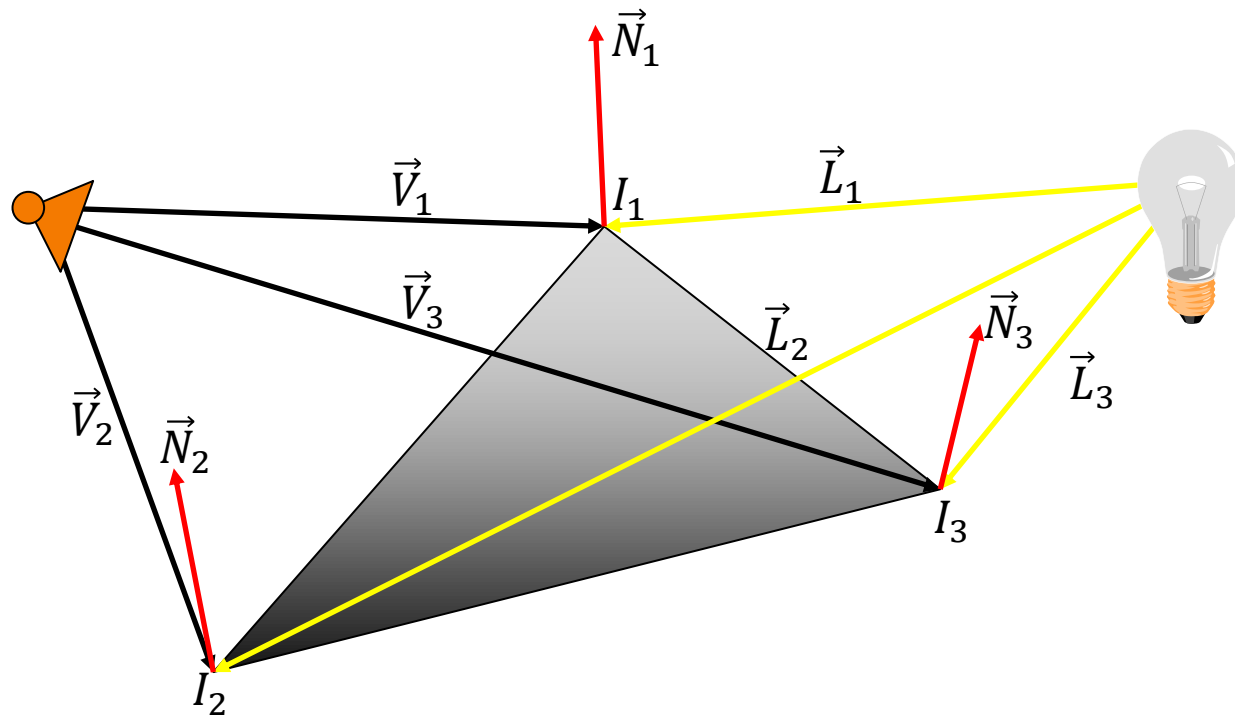
$$I = I_E + \sum_L [K_A \cdot I_L^A + (K_D \cdot \langle \vec{N}, \vec{L} \rangle + K_S \cdot \langle \vec{V}, \vec{R}(\vec{L}) \rangle^{K_n}) \cdot I_L]$$



Gouraud Shading

One lighting calculation **per vertex**

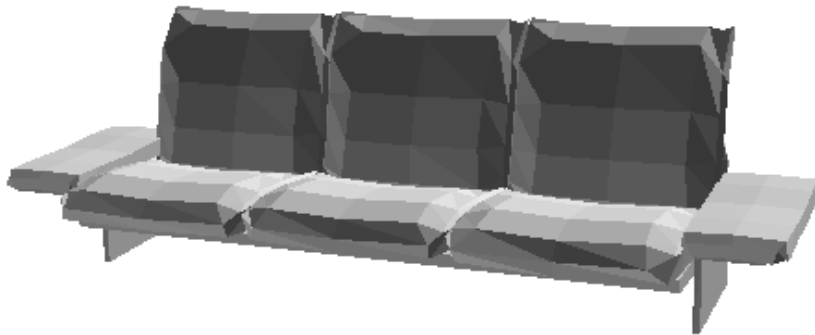
Assign pixel colors inside polygon by interpolating colors computed at vertices



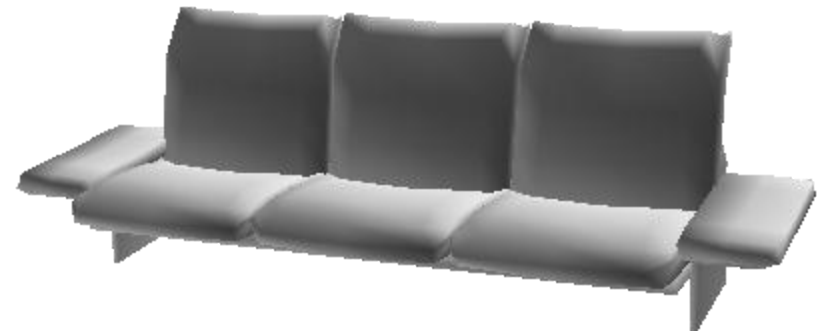


Gouraud Shading

Produces smoothly shaded polygonal mesh
Continuous shading over adjacent polygons



Flat Shading



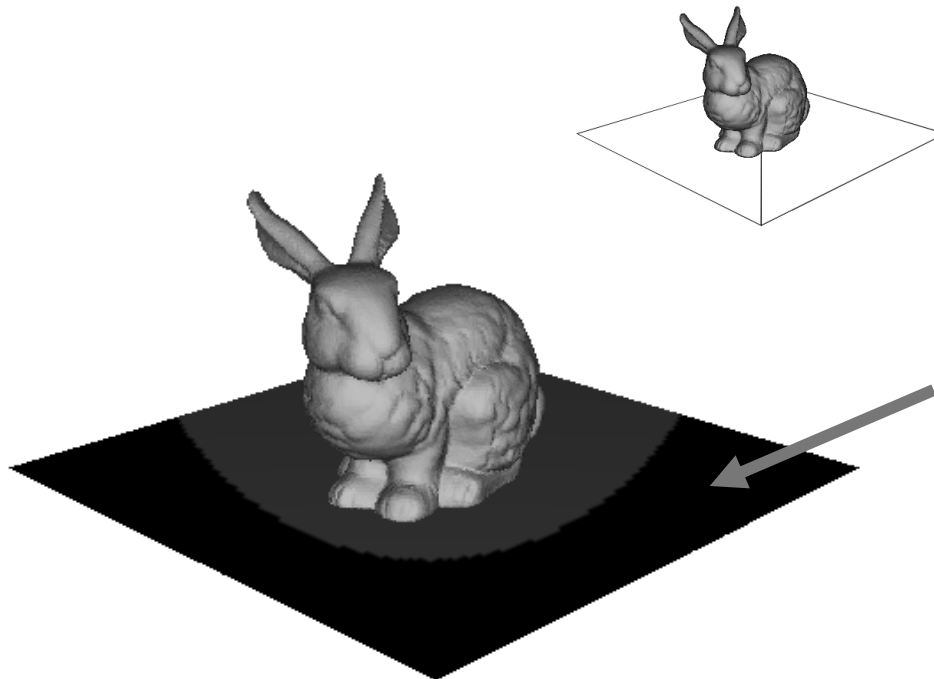
Gouraud Shading

This is the lighting model that is (was) implemented on the graphics card as part of the fixed pipeline.



Gouraud Shading

Produces smoothly shaded polygonal mesh
Continuous shading over adjacent polygons

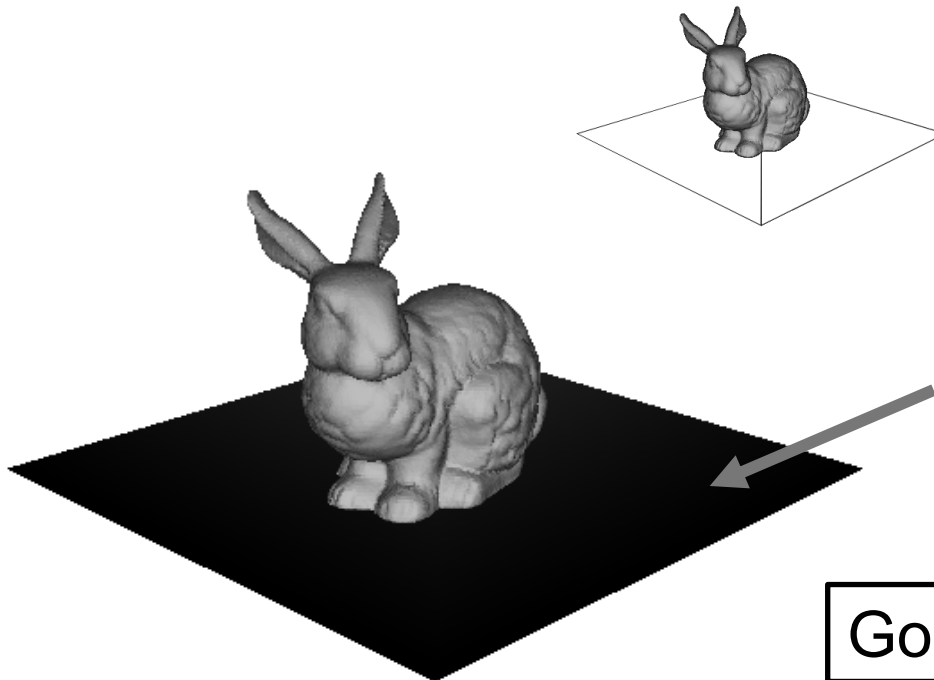


What happens with large polygon and spotlight / specular highlight?



Gouraud Shading

Produces smoothly shaded polygonal mesh
Continuous shading over adjacent polygons



What happens with
large polygon and
spotlight / specular
highlight?

Gouraud Shading Demo

Polygon Shading Algorithms



Shading models:

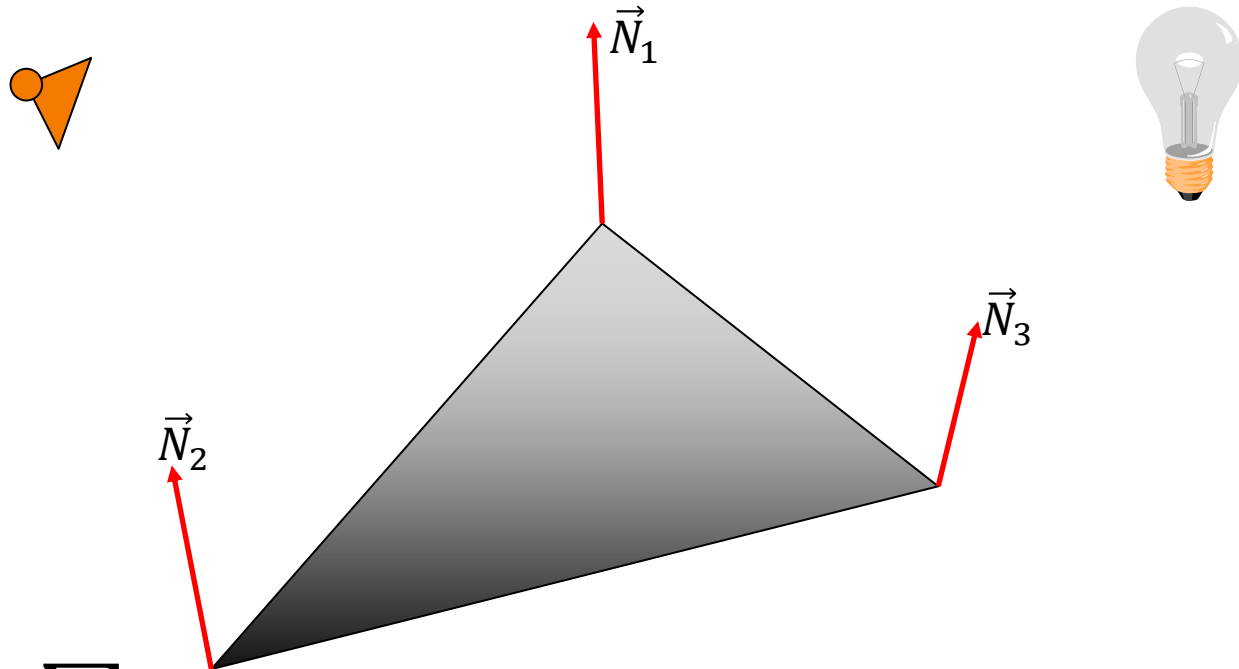
- Flat
- Gouraud
- Phong



Phong Shading

One lighting calculation **per pixel/fragment**

Approximate surface normals for points inside polygons by linear interpolation of normals from vertices

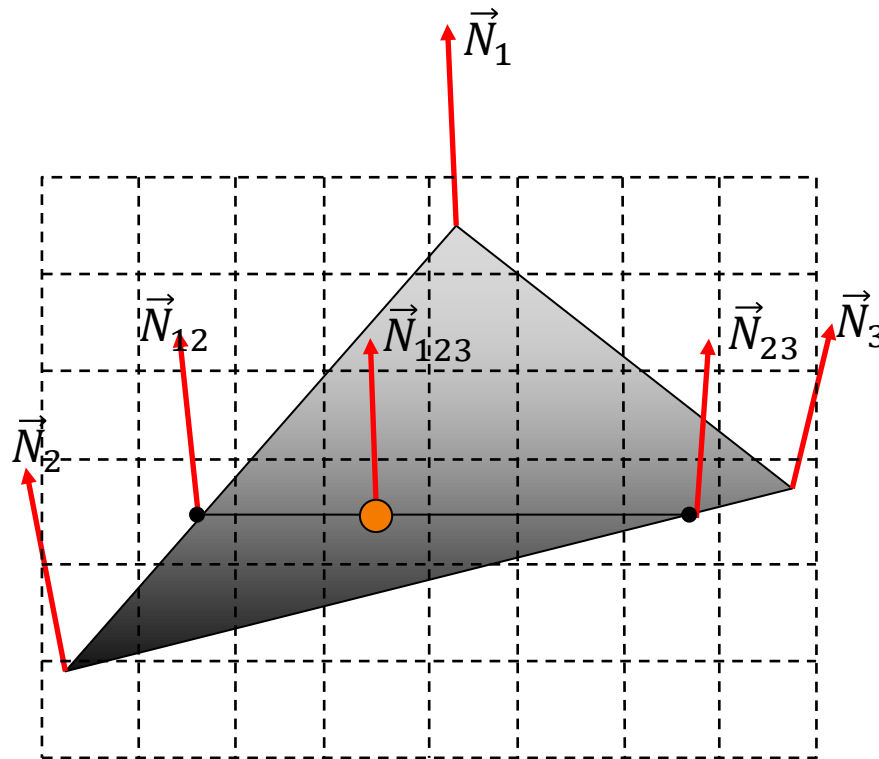


$$I = I_E + \sum_L \left[K_A \cdot I_L^A + \left(K_D \cdot \langle \vec{N}, \vec{L} \rangle + K_S \cdot \langle \vec{V}, \vec{R}(\vec{L}) \rangle^{K_n} \right) \cdot I_L \right]$$



Phong Shading

When rasterizing, interpolate normals down and across scan lines

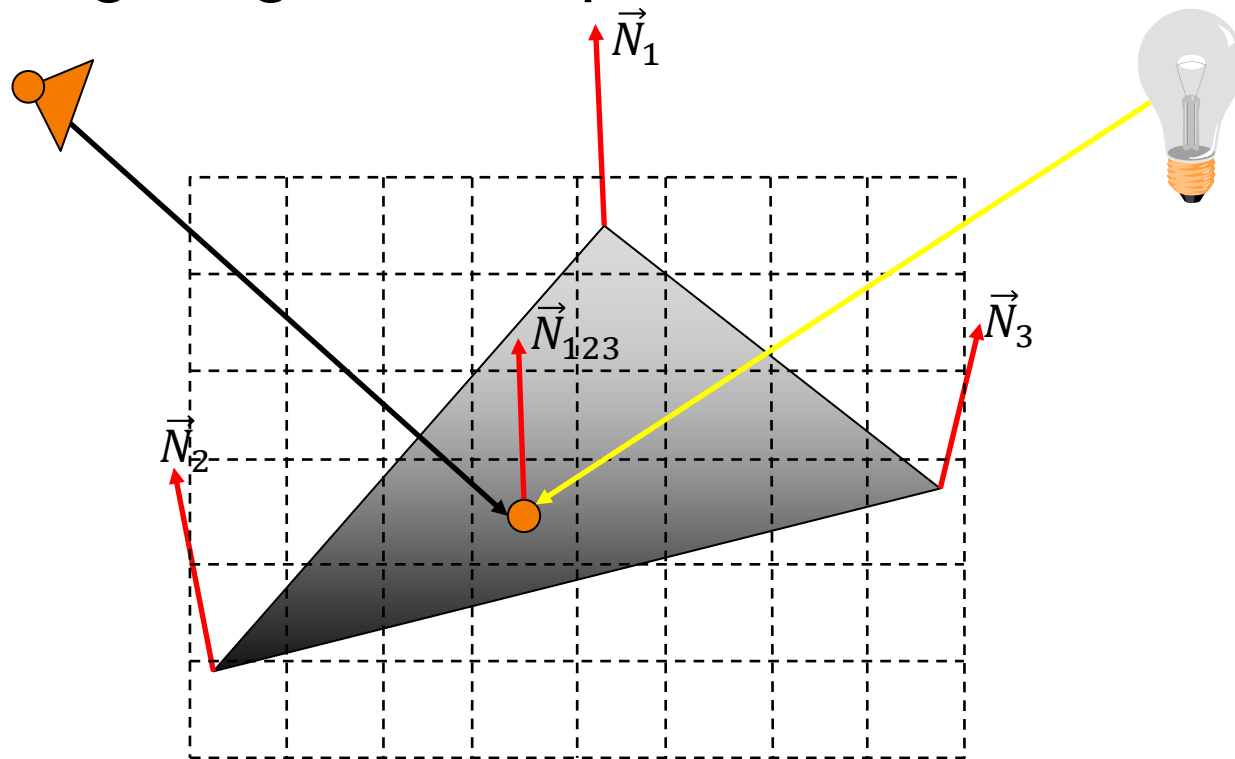




Phong Shading

When rasterizing, interpolate normals down and across scan lines

Compute lighting at each pixel

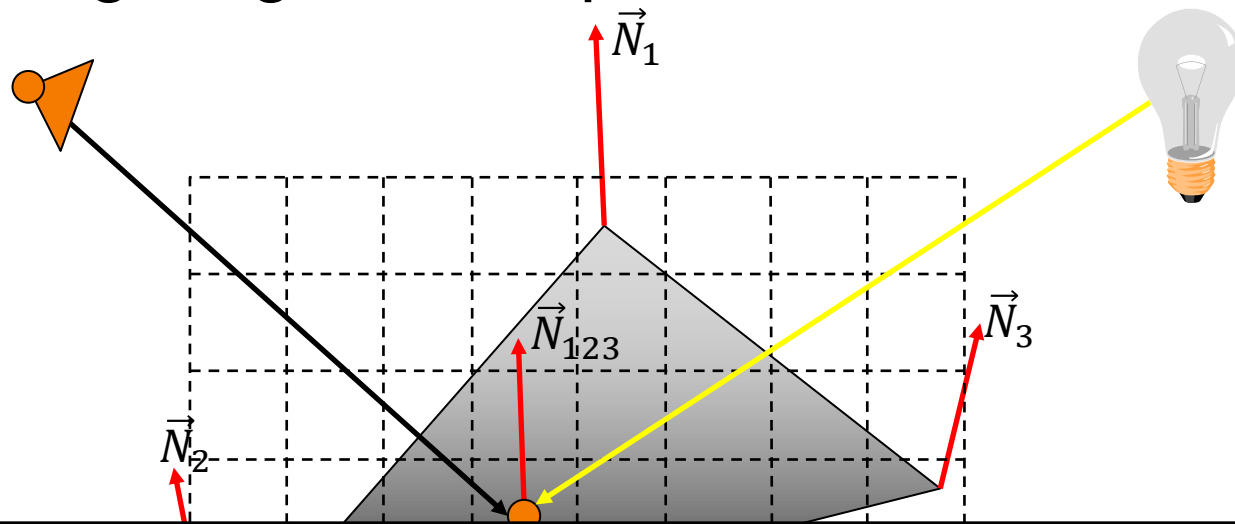




Phong Shading

When rasterizing, interpolate normals down and across scan lines

Compute lighting at each pixel



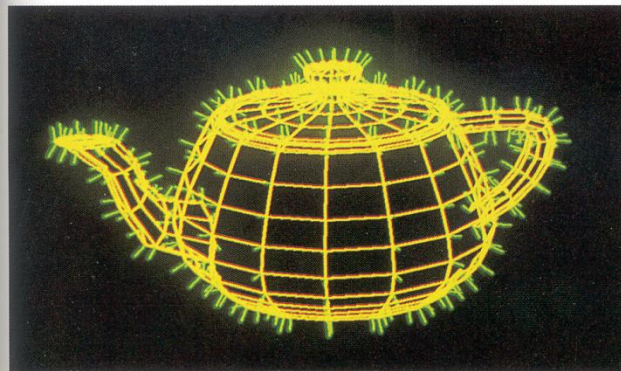
Wasn't supported in early generation graphic cards.
Now supported by the GPU's fragment shader.

Phong Shading Demo

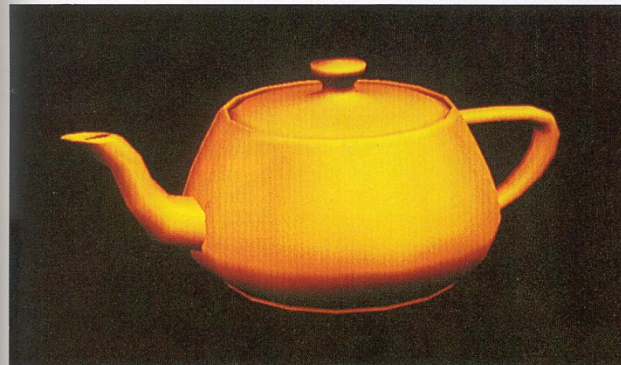
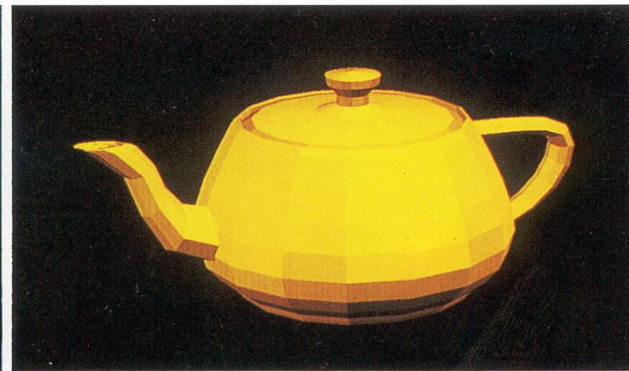
Polygon Shading Algorithms



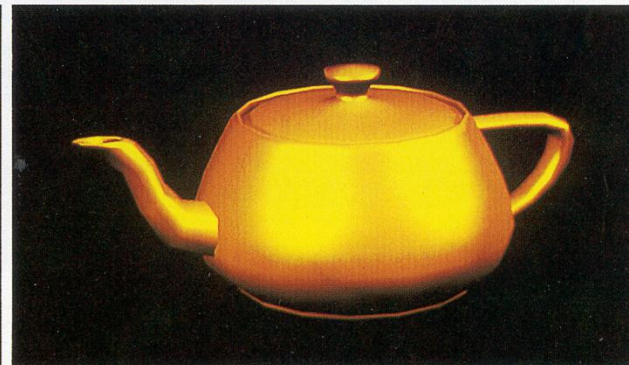
Wireframe



Flat

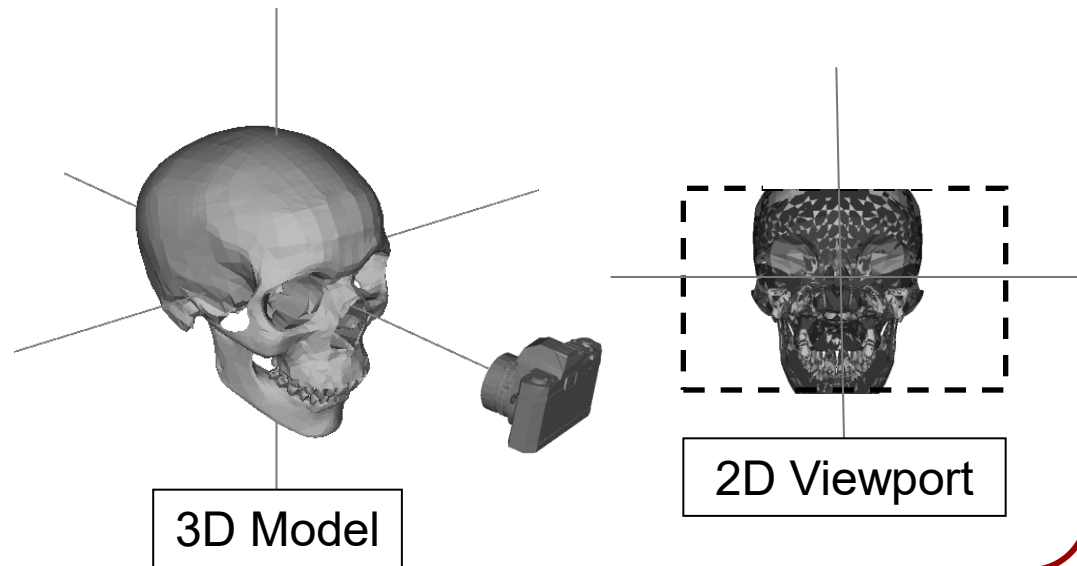
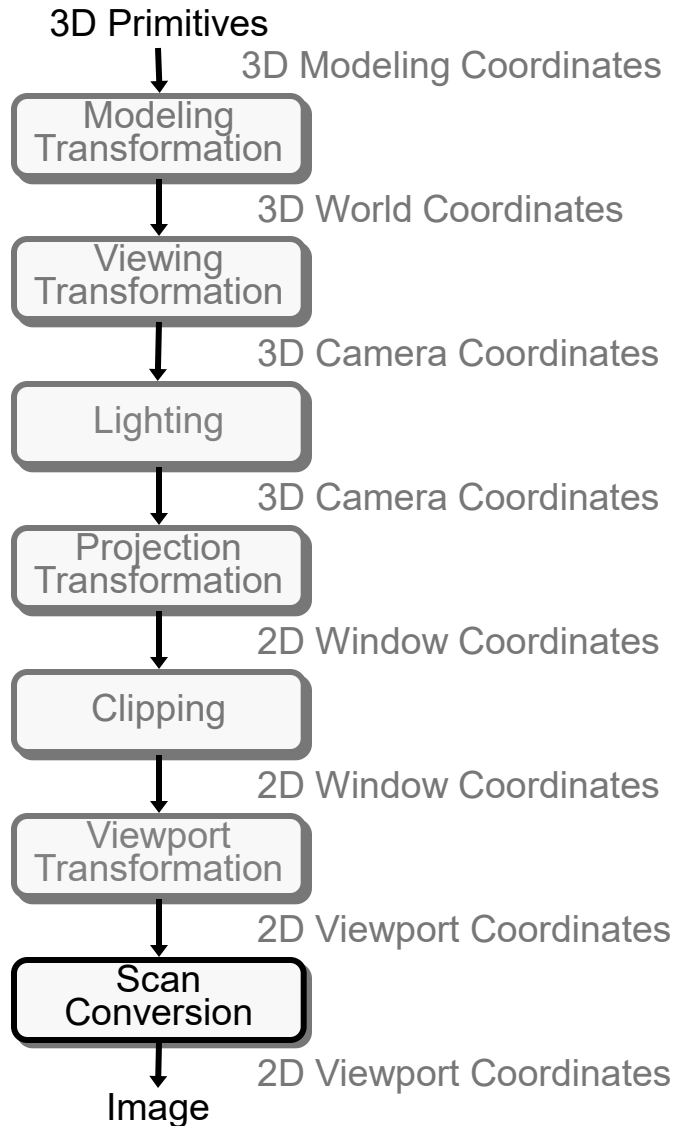


Gouraud



Phong

3D Rendering Pipeline (for direct illumination)





Overview

Scan conversion

Figure out which pixels to fill

Shading

Determine the color for each filled pixel

Depth test

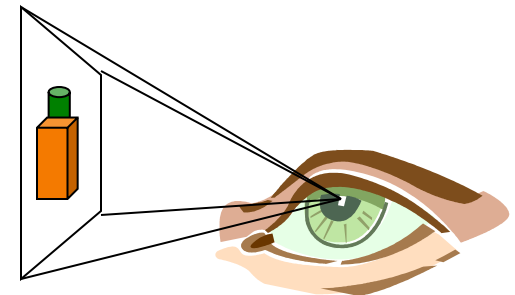
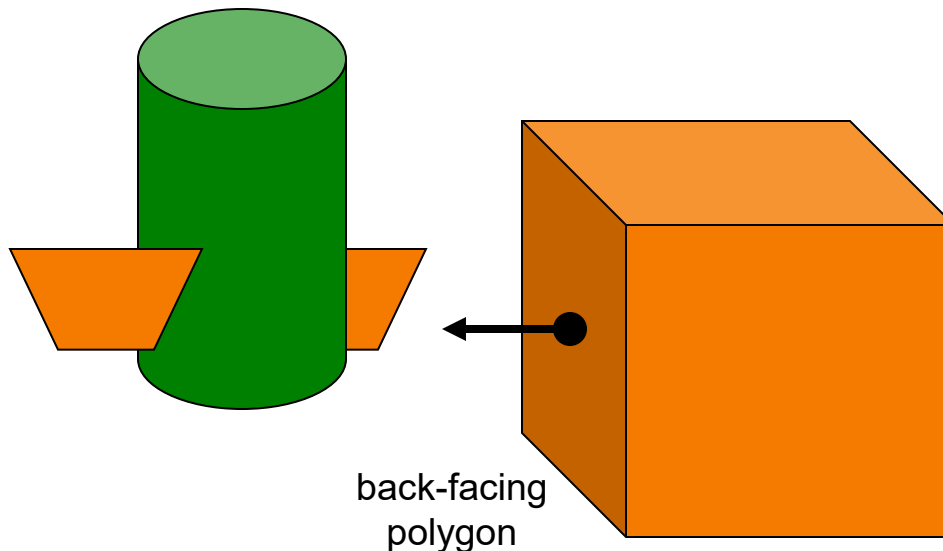
Determine when the color of a pixel comes from the front-most primitive



Motivation

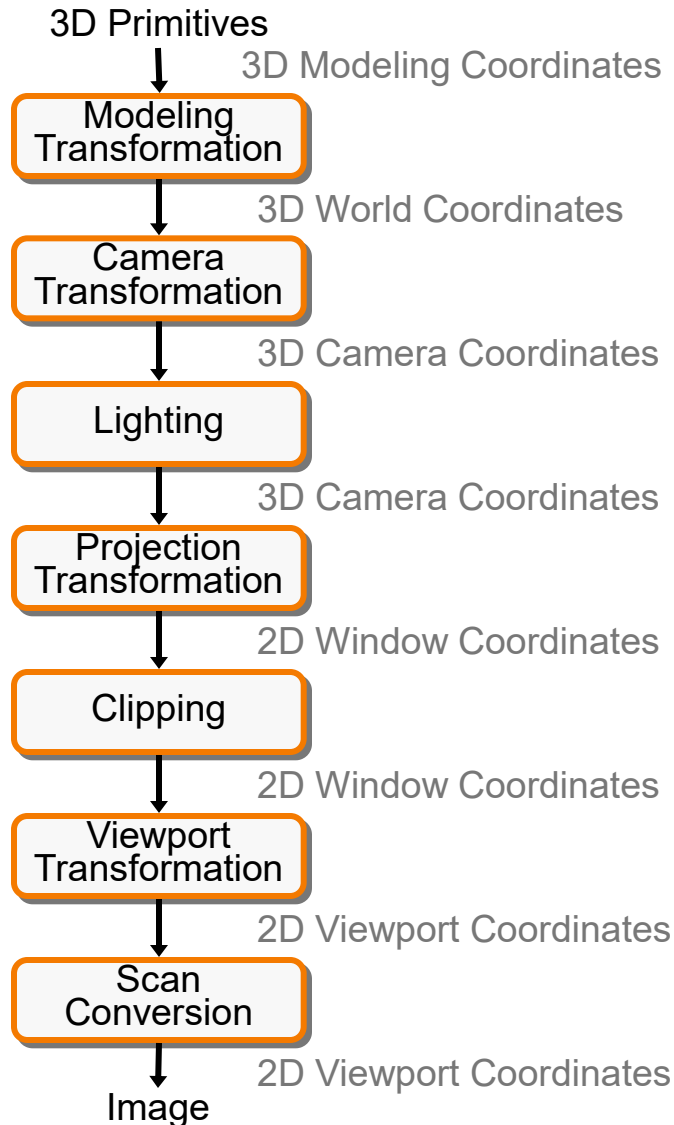
In general, don't want to draw surfaces that are not visible:

- Surfaces may be back-facing
- Surfaces may not project onto the window
- Surfaces may be covered in 3D
- Surfaces may be covered in the image plane





3D Rendering Pipeline



Somewhere, need to decide which objects are visible, and which are hidden (the sooner the better).

Visibility algorithms



36 • I. E. Sutherland, R. F. Sproull, and R. A. Schumacker

A Characterization of Ten Hidden-Surface Algorithms • 37

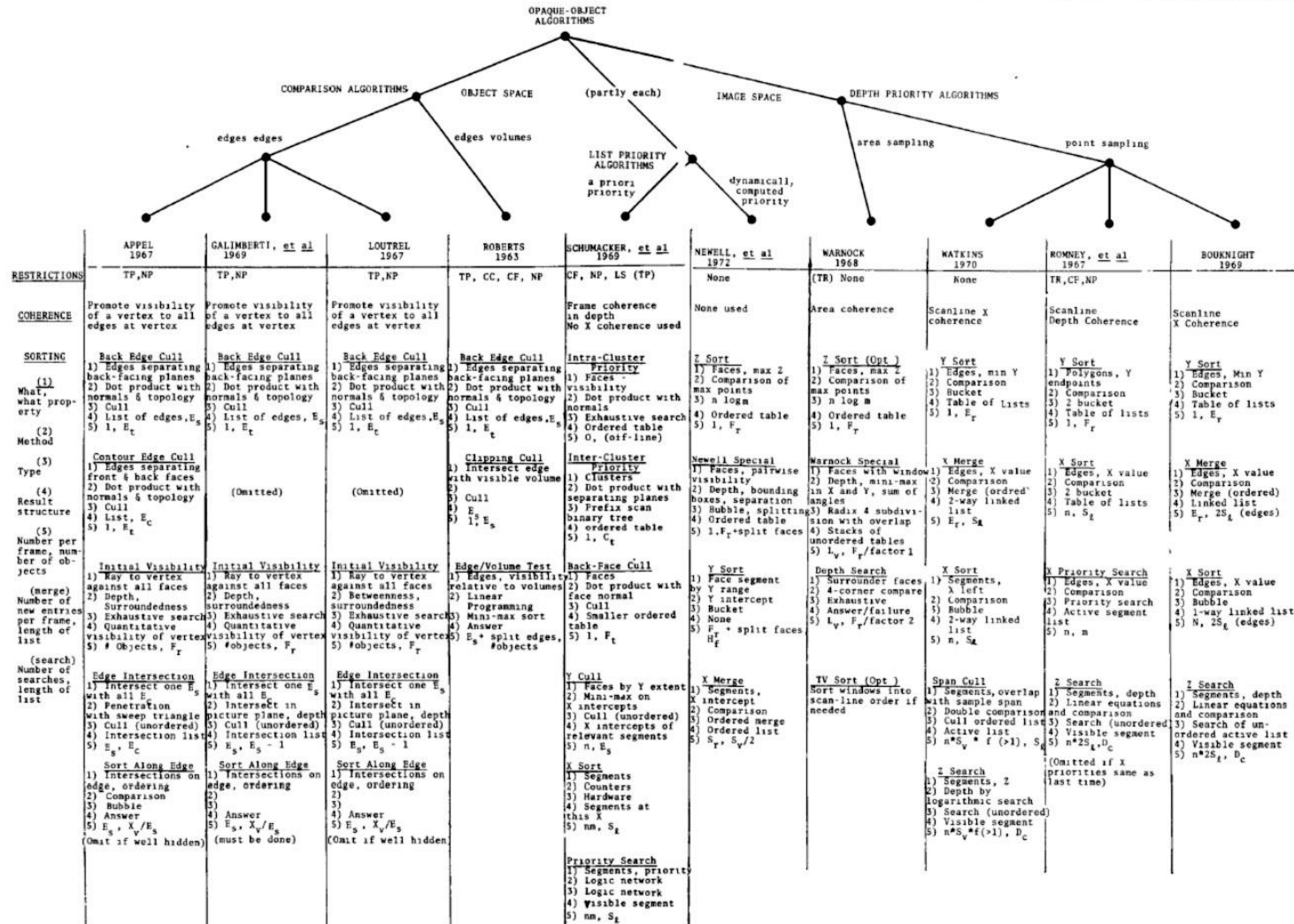


Figure 29. Characterization of ten opaque-object algorithms. b. Comparison of the algorithms.

[Sutherland '74]

Hidden Surface Removal (HSR)



Algorithms for HSR

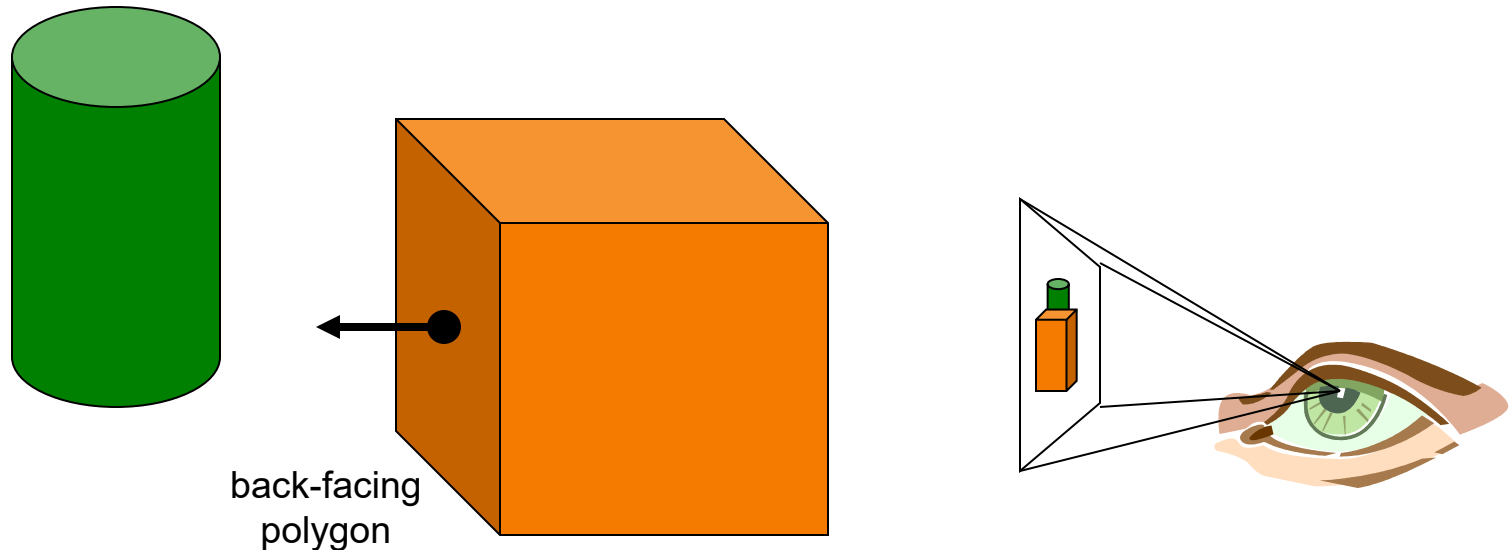
- Back-face detection
- View-frustum culling
- z-buffer



Back-face detection

Q: How do we test for back-facing polygons?

A: Dot product of the normal and view directions.



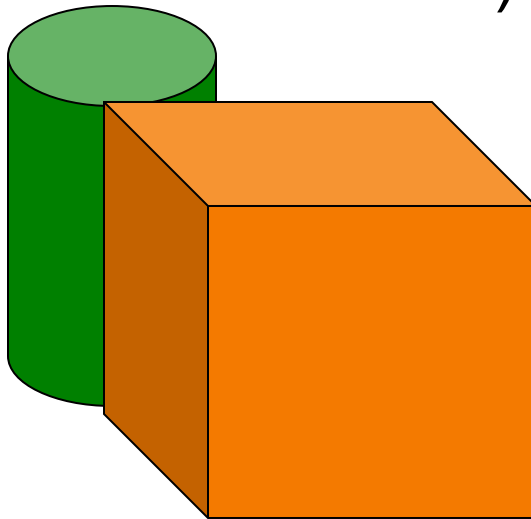
If $\langle \vec{V}, \vec{N} \rangle > 0$, then polygon is back-facing



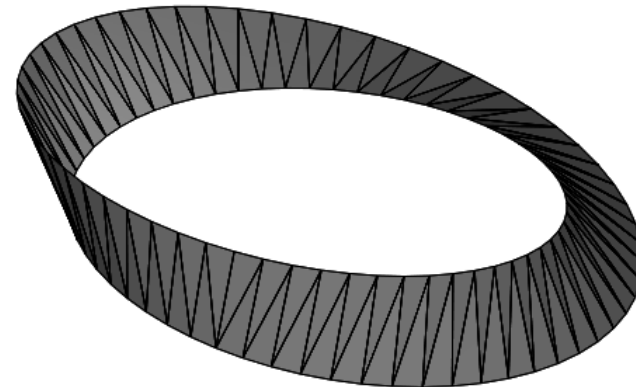
Back-face detection

This method:

- ✗ Does not eliminate shapes overlapping in 3D or 2D
- ✗ Requires surface to be **water-tight** and **orientable** (not all surfaces are)



Overlapping
Objects

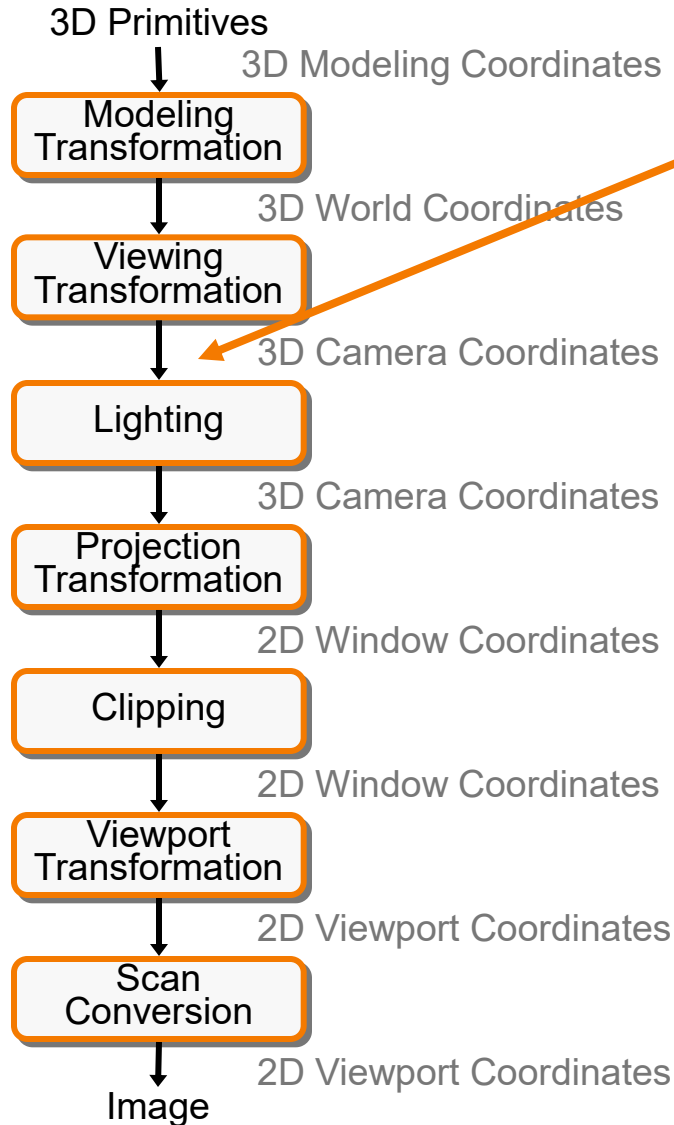


Non-water-tight
Non-orientable

In general, back-face expected to remove $\approx$ half of polygon surfaces from removal further visibility tests

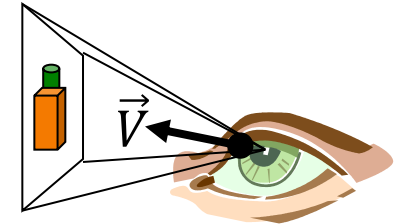
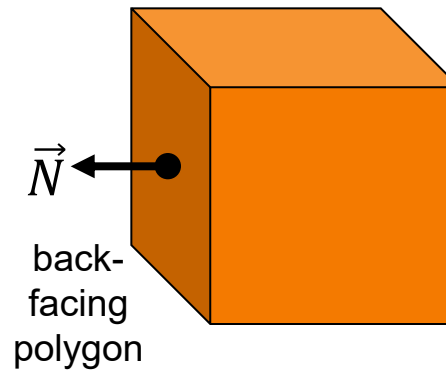


3D Rendering Pipeline



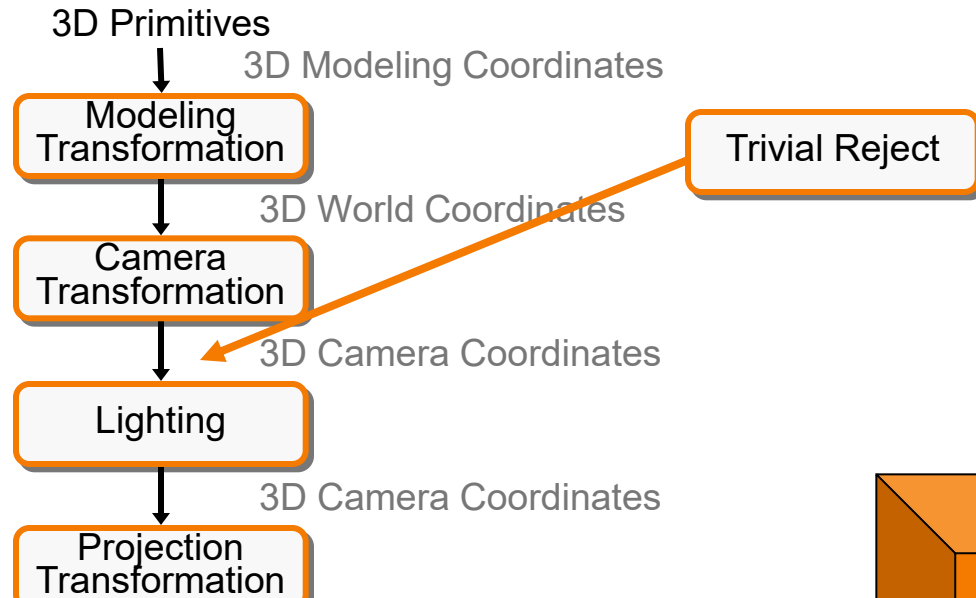
Trivial Reject

A polygon is back-facing if
 $\langle \vec{V}, \vec{N} \rangle > 0$

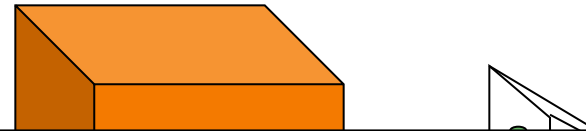




3D Rendering Pipeline



A polygon is back-facing if
 $\langle \vec{V}, \vec{N} \rangle > 0$



Note: When your graphics card does this, it does not use the rendering normals (used for lighting).

It uses the geometric normal – the cross-product of the triangle edges.

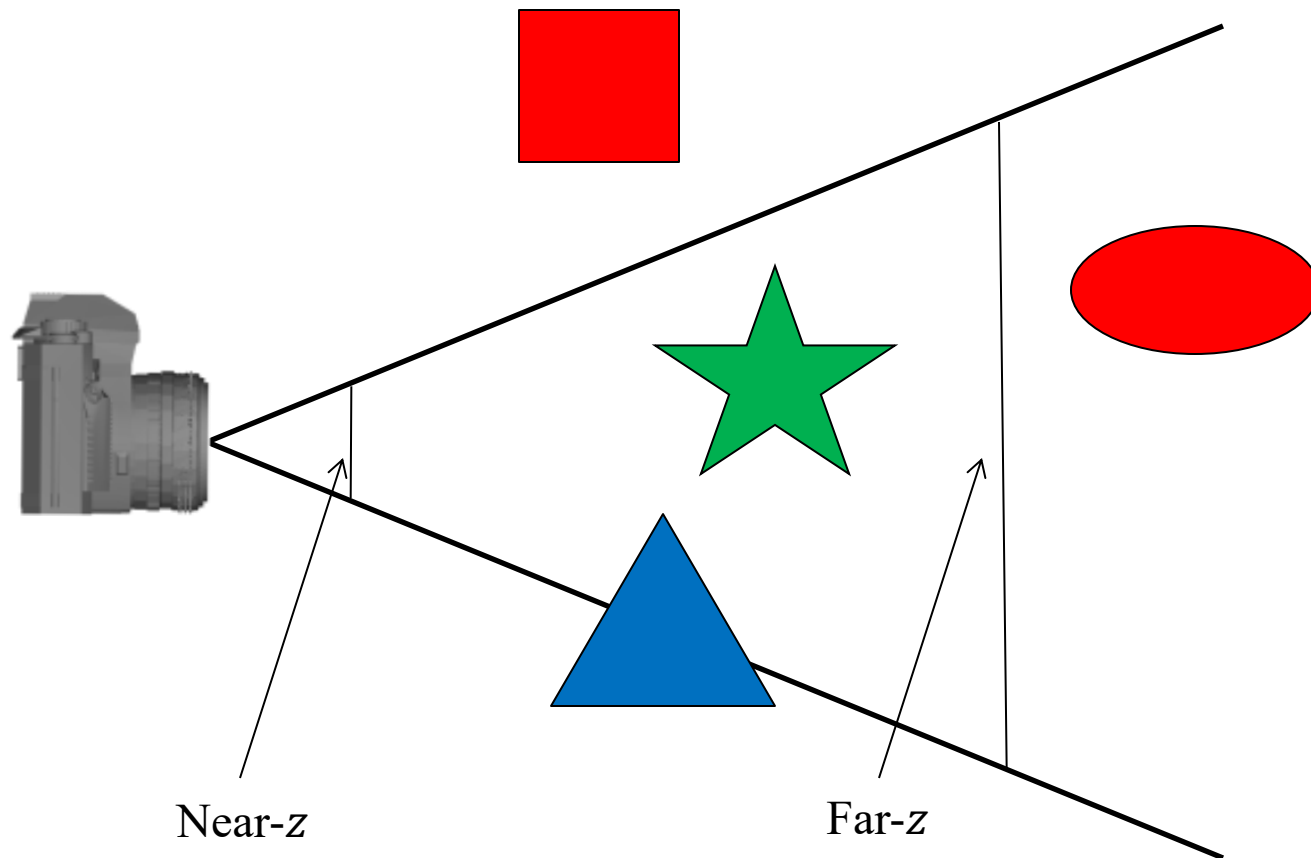
⇒ Make sure that the ordering of the vertices is consistent.

By default, triangles/polygons are **front-facing** if the vertices are in **counter-clockwise** order when viewed from the camera.



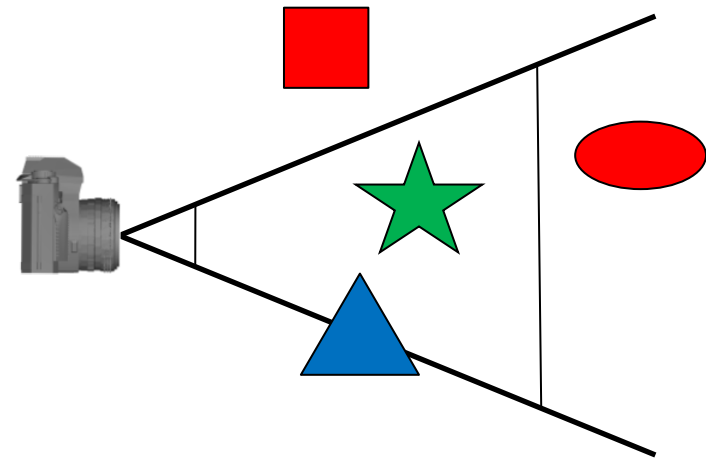
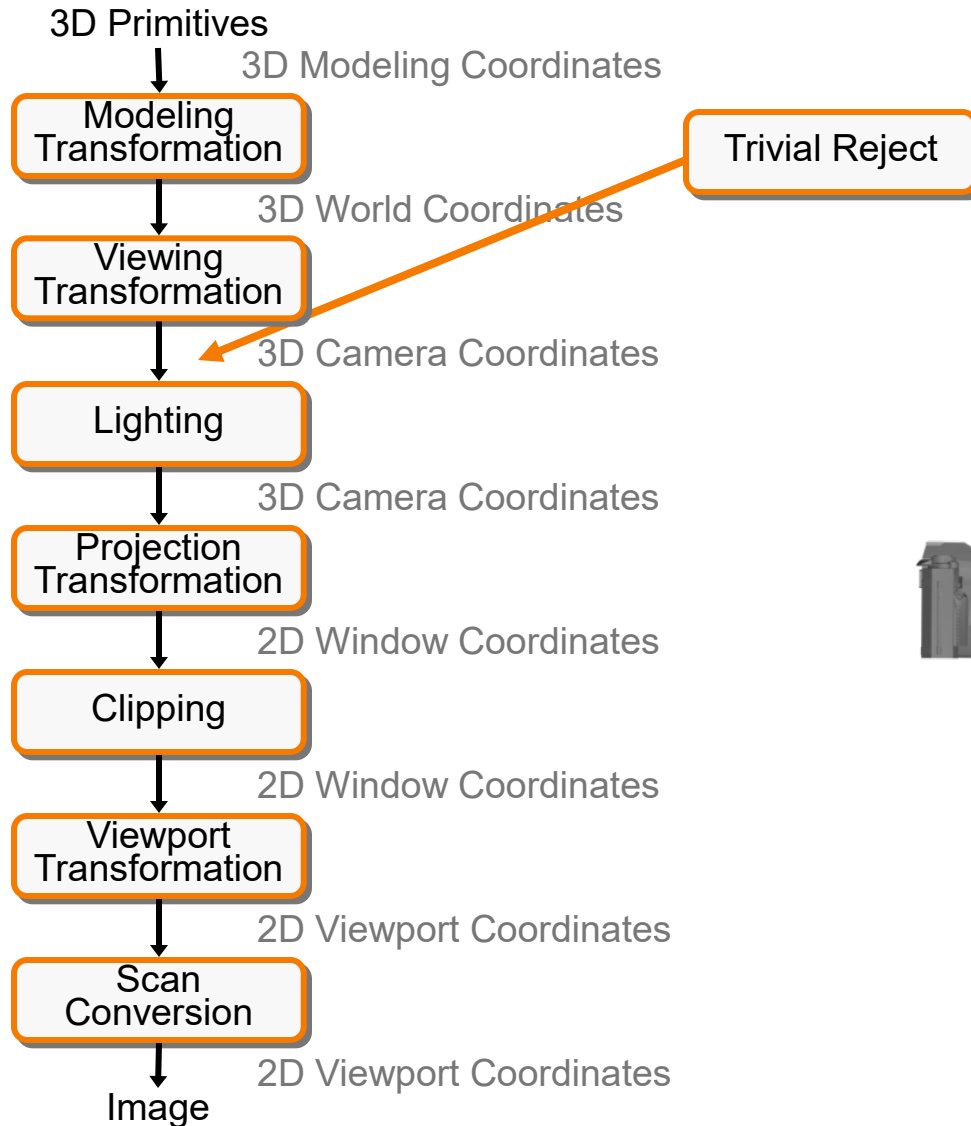
View-frustum culling

If the shape is outside the viewing volume, we don't need to draw it.





View-frustrum culling

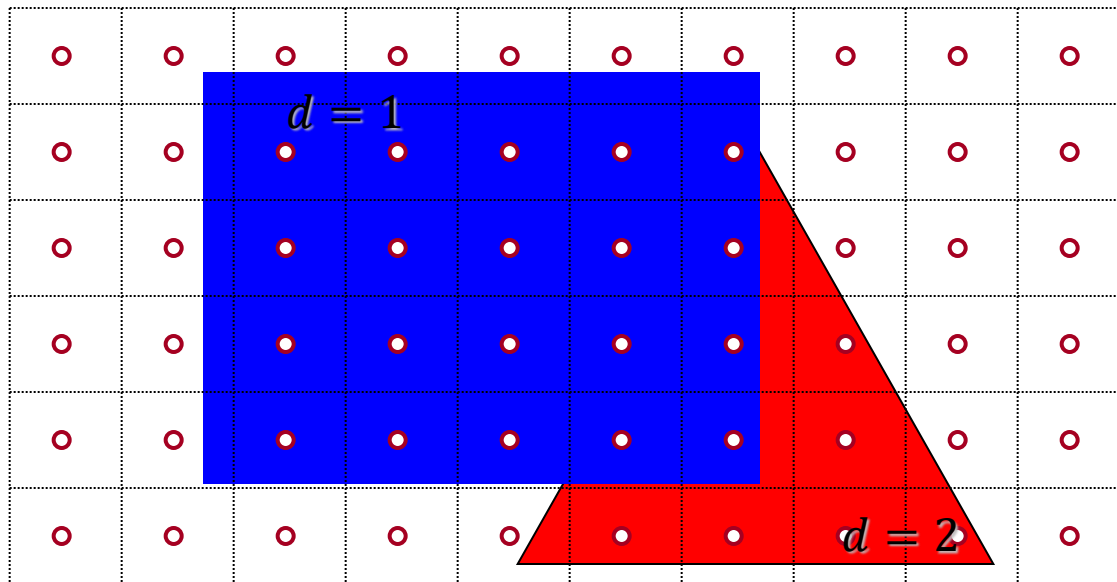




z-Buffer

Store color & **depth** of closest object at each pixel

- Initialize depth of each pixel in the z-buffer to ∞
- Only update pixels from a primitive when the depth is closer than what's stored in the z-buffer

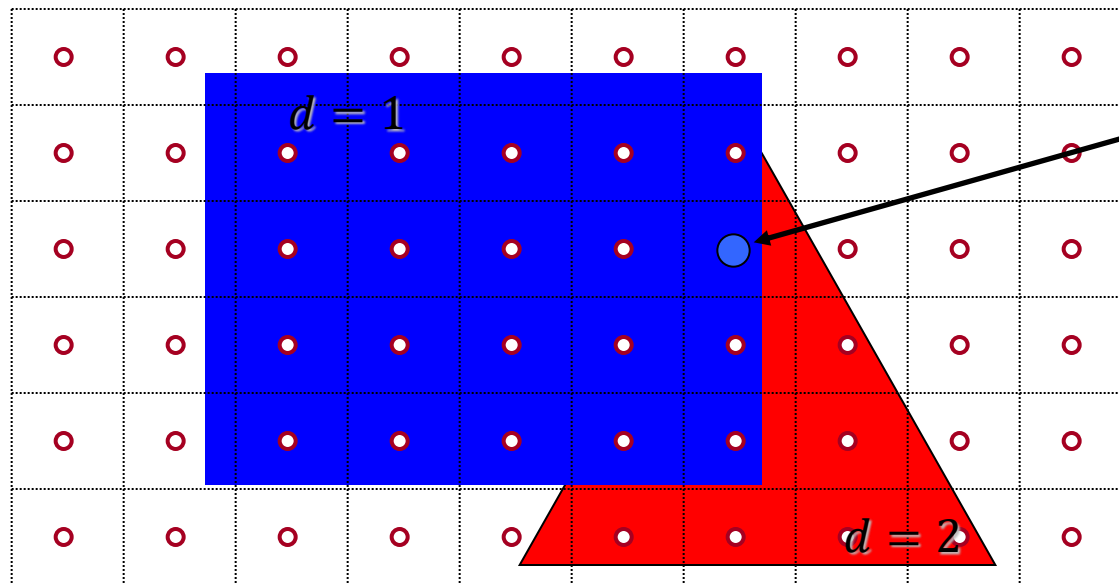




z-Buffer

Store color & **depth** of closest object at each pixel

- Initialize depth of each pixel in the z-buffer to ∞
- Only update pixels from a primitive when the depth is closer than what's stored in the z-buffer



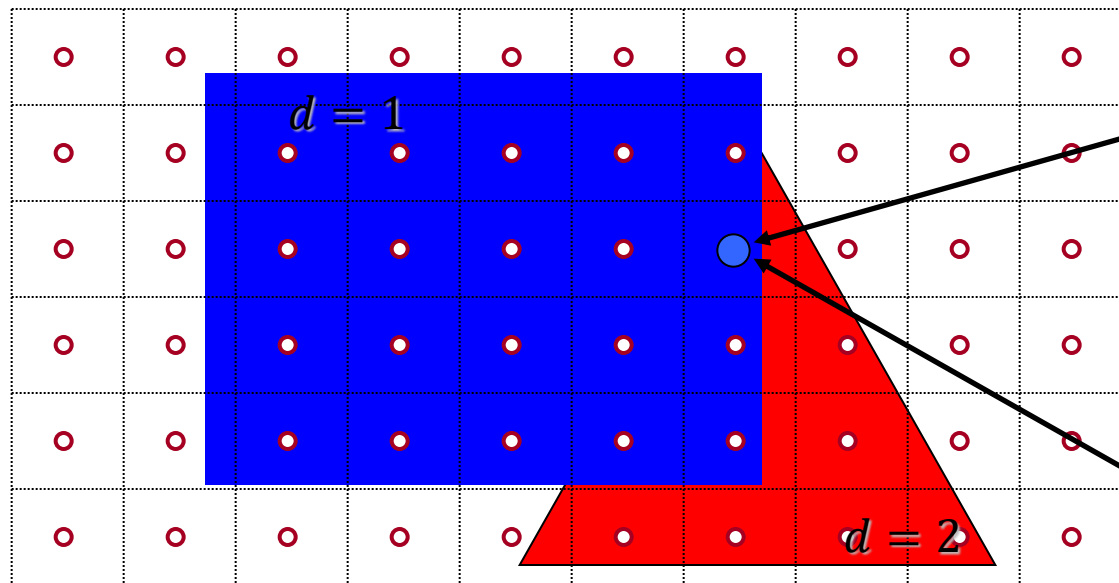
Case 1 (Blue before Red):
Blue $\rightarrow (d = 1) < (d = \infty)$:
Set $RGB = (0,0,1), d = 1$
Red $\rightarrow (d = 2) > (d = 1)$:
Don't change pixel



z-Buffer

Store color & **depth** of closest object at each pixel

- Initialize depth of each pixel in the z-buffer to ∞
- Only update pixels from a primitive when the depth is closer than what's stored in the z-buffer



Case 1 (Blue before Red):

Blue $\rightarrow (d = 1) < (d = \infty)$:

Set $RGB = (0,0,1), d = 1$

Red $\rightarrow (d = 2) > (d = 1)$:

Don't change pixel

Case 2 (Red before Blue):

Red $\rightarrow (d = 2) < (d = \infty)$:

Set $RGB = (1,0,0), d = 2$

Blue $\rightarrow (d = 1) < (d = 2)$:

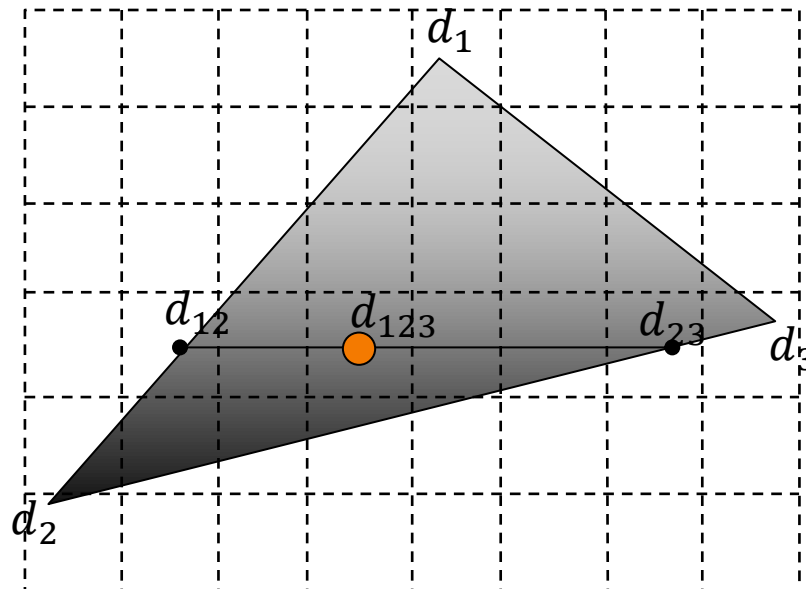
Set $RGB = (0,0,1), d = 1$



z-Buffer

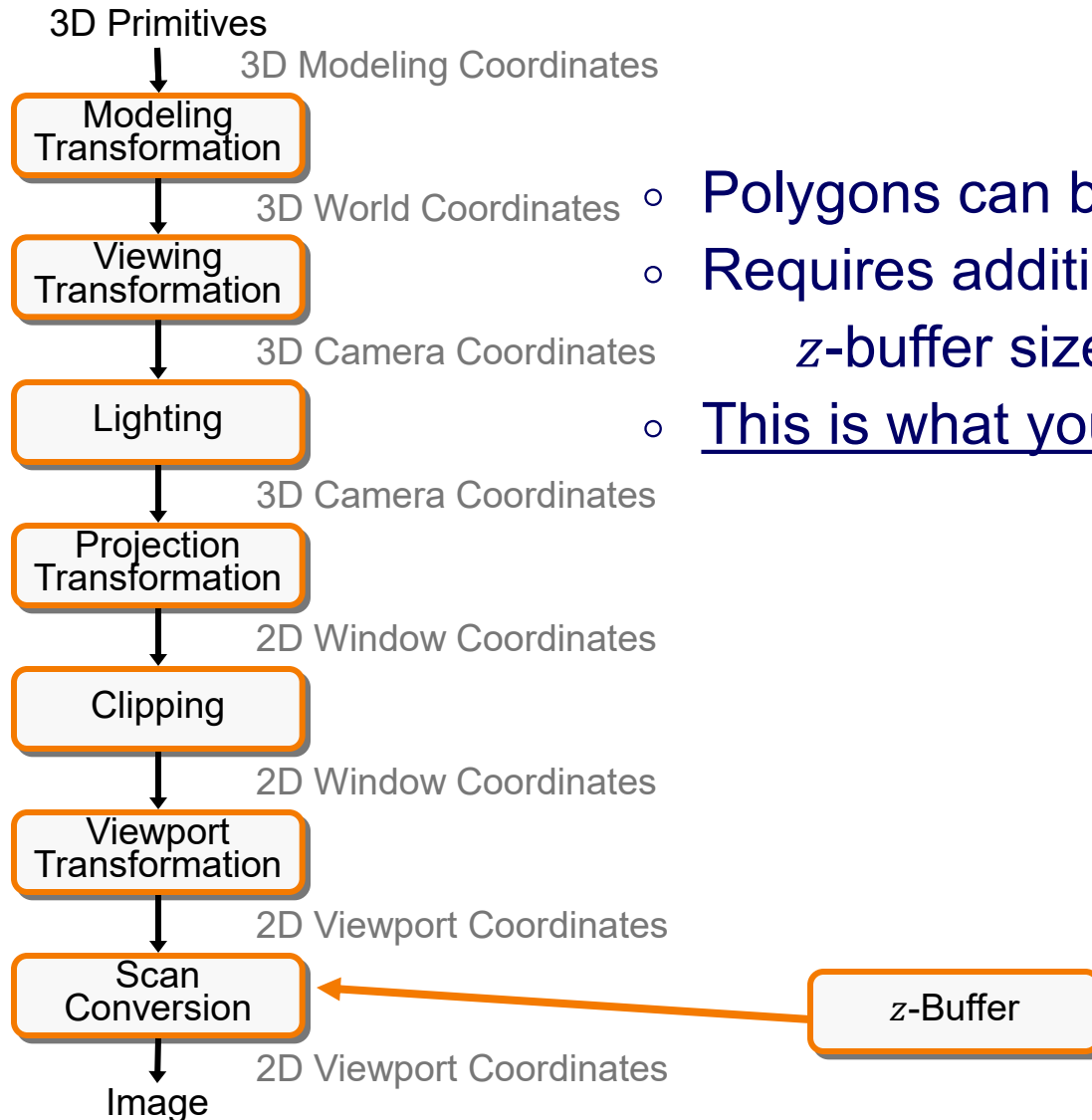
Store color & **depth** of closest object at each pixel

- Initialize depth of each pixel in the z-buffer to ∞
- Only update pixels from a primitive when the depth is closer than what's stored in the z-buffer
- Depths are interpolated from vertices, just like colors



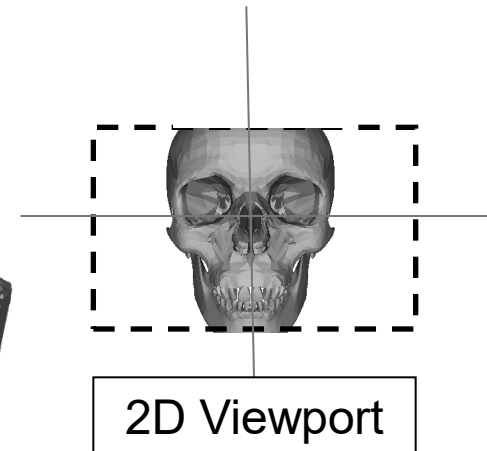
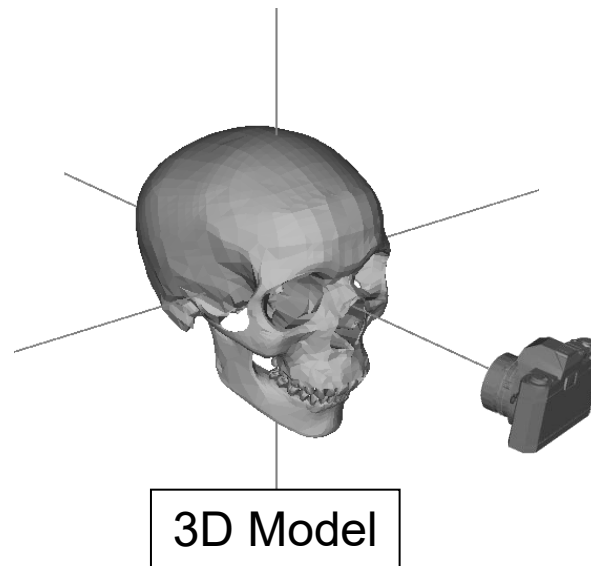
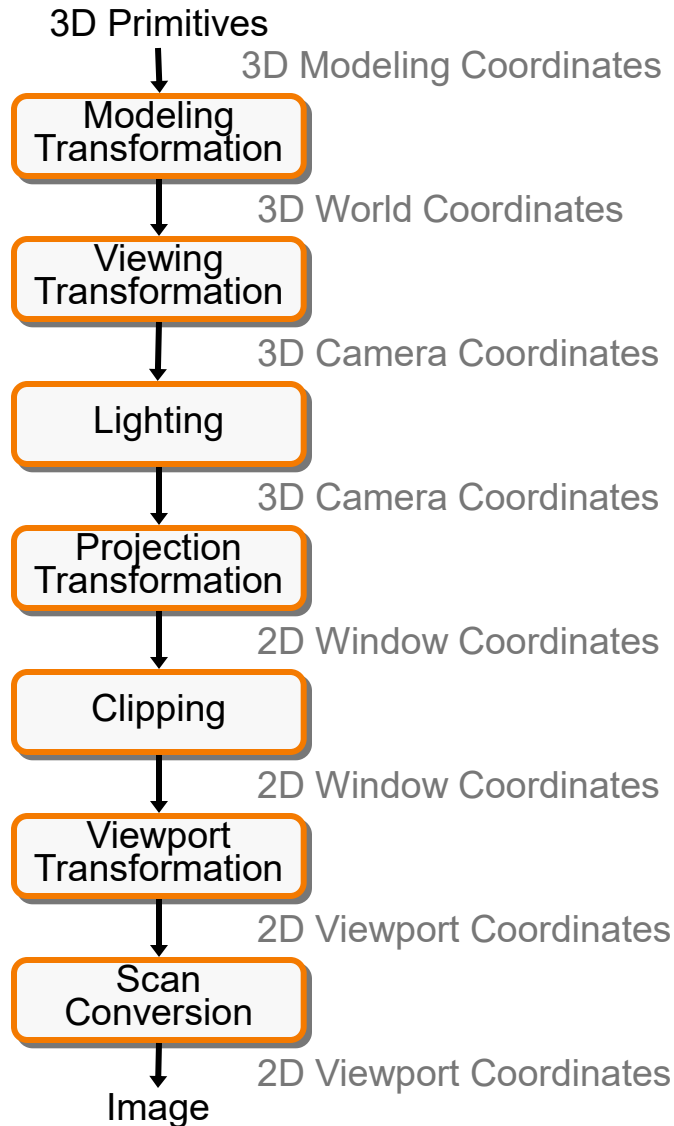


3D Rendering Pipeline



- Polygons can be rasterized in any order
- Requires additional memory
 - $z\text{-buffer size} \approx \text{frame buffer}$
- This is what your graphics card does

3D Rendering Pipeline (for direct illumination)





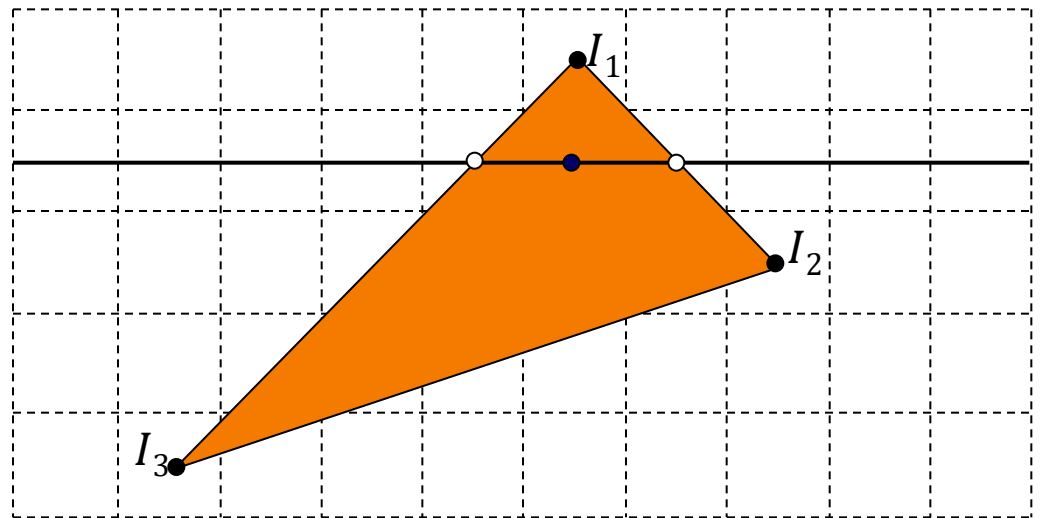
Scan Conversion

How should we average information (e.g. color, normal, depth) from the vertices of a triangle?

- Interpolate using screen space (2D) weights
- Interpolate using world space (3D) weights

It's easier to do the interpolation in 2D.

Is there a difference?





Scan Conversion

Projective transformations (recall)

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Properties of projective transformations:

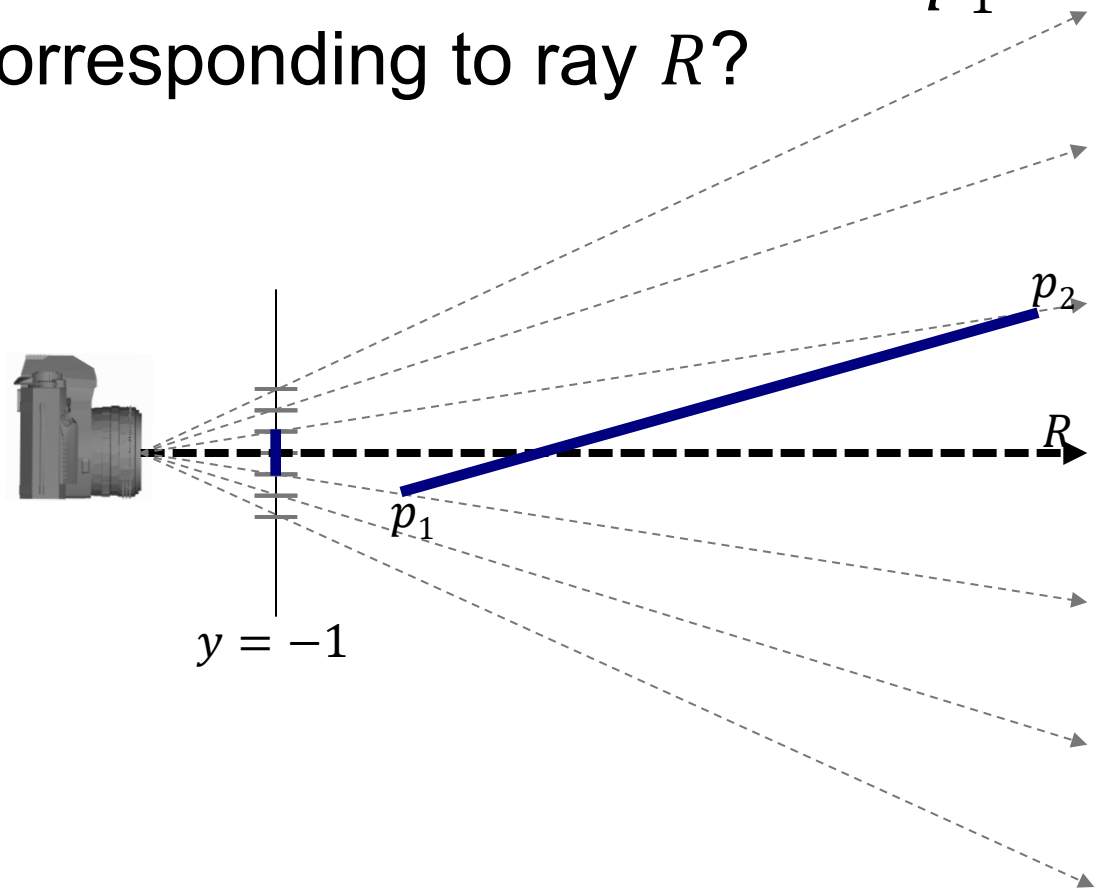
- Origin does not necessarily map to origin
- Lines map to lines
- (Weighted) average is not necessarily preserved
- Parallel lines do not necessarily remain parallel
- Closed under composition



Scan Conversion Example

A line segment in 2D projected onto a 1D window.

How to interpolate the information from vertices p_1 and p_2 at the pixel corresponding to ray R ?

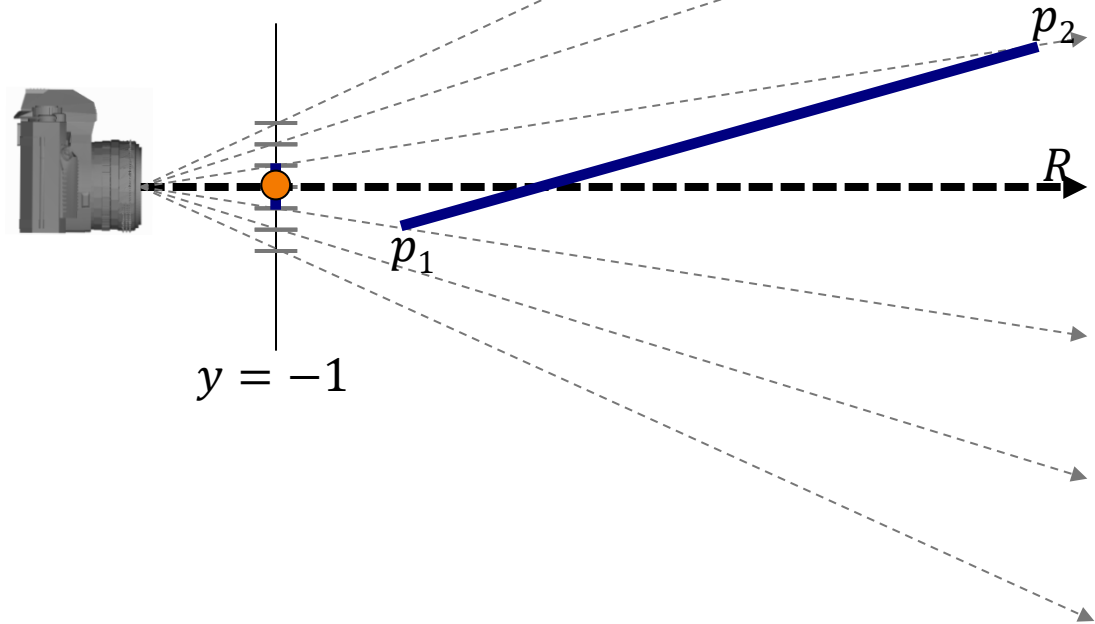




Scan Conversion Example

A line segment in 2D projected onto a 1D window.

1. [1D] The projected intersection of the ray with the line segment is directly between the projections of p_1 and p_2 :
⇒ Use equal contributions from p_1 and p_2 .

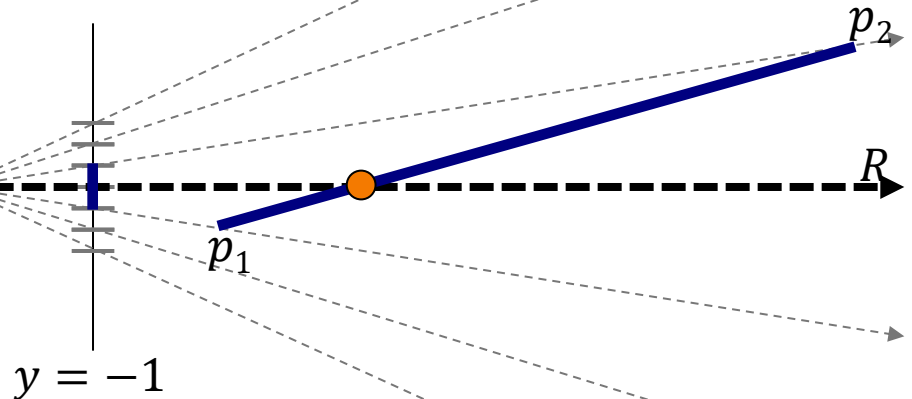
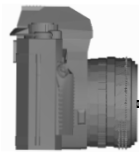




Scan Conversion Example

A line segment in 2D projected onto a 1D window.

1. [1D] The projected intersection of the ray with the line segment is directly between the projections of p_1 and p_2 :
⇒ Use equal contributions from p_1 and p_2 .
2. [2D] The intersection of the ray with the line segment is closer to p_1 than to p_2 :
⇒ Use more information from p_1 than from p_2 .





Scan Conversion Example

Q: How do we interpolate correctly at $(x, -1)$?

If $p_1 = (x_1, z_1)$ and $p_2 = (x_2, z_2)$, to find the blending value α for a pixel falling at position x in the screen we need to solve:

$$(1 - \alpha)(x_1, z_1) + \alpha(x_2, z_2) \sim (x, -1)$$



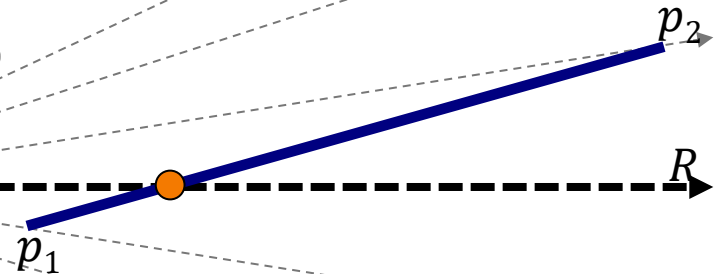
$$((1 - \alpha)x_1 + \alpha x_2, (1 - \alpha)z_1 + \alpha z_2) \sim (x, -1)$$



$$\frac{(1 - \alpha)x_1 + \alpha x_2}{(1 - \alpha)z_1 + \alpha z_2} = \frac{x}{-1}$$



$z = -1$





Scan Conversion Example

Q: How do we interpolate correctly at $(x, -1)$?

If $p_1 = (x_1, z_1)$ and $p_2 = (x_2, z_2)$, to find the blending value α for a pixel falling at position x in the screen we need to solve:

$(1 - \alpha)$ To compute the interpolation weights, perform a perspective divide:

$$\frac{(1 - \alpha)x_1 + \alpha x_2}{(1 - \alpha)z_1 + \alpha z_2} = \frac{x}{-1}$$

This is different than solving for the blending value in the image plane:

$$(1 - \alpha) \frac{x_1}{z_1} + \alpha \frac{x_2}{z_2} = \frac{x}{-1}$$

