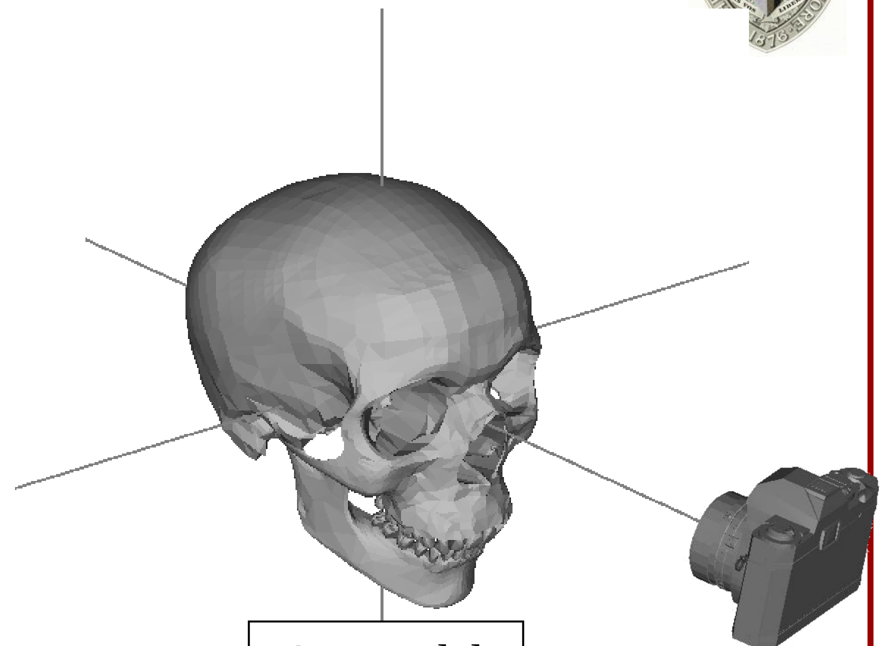
Scan Conversion

Michael Kazhdan

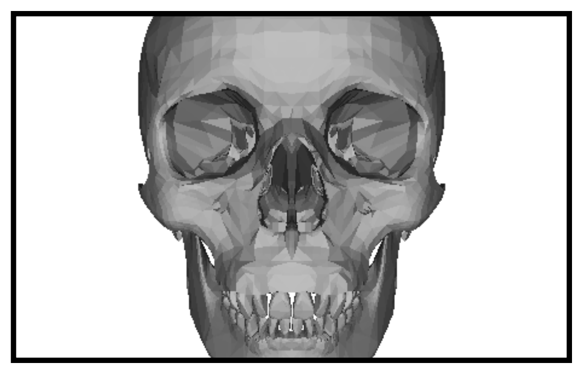
(601.457/657)

3D Rendering Pipeline (for direct illumination)

3D Primitives
↓ 3D Modeling Coordinates
Modeling Transformation
↓ 3D World Coordinates
Viewing Transformation
↓ 3D Camera Coordinates
Lighting
↓ 3D Camera Coordinates
Projection Transformation
↓ 2D Window Coordinates
Clipping
↓ 2D Window Coordinates
Viewport Transformation
↓ 2D Viewport Coordinates
Scan Conversion
↓ 2D Viewport Coordinates
Image



3D Model



2D Viewport

3D Rendering Pipeline (for direct illumination)

(x, y, z)

↓ 3D Object Coordinates

Modeling Transformation

↓ 3D World Coordinates

Viewing Transformation

↓ 3D Camera Coordinates

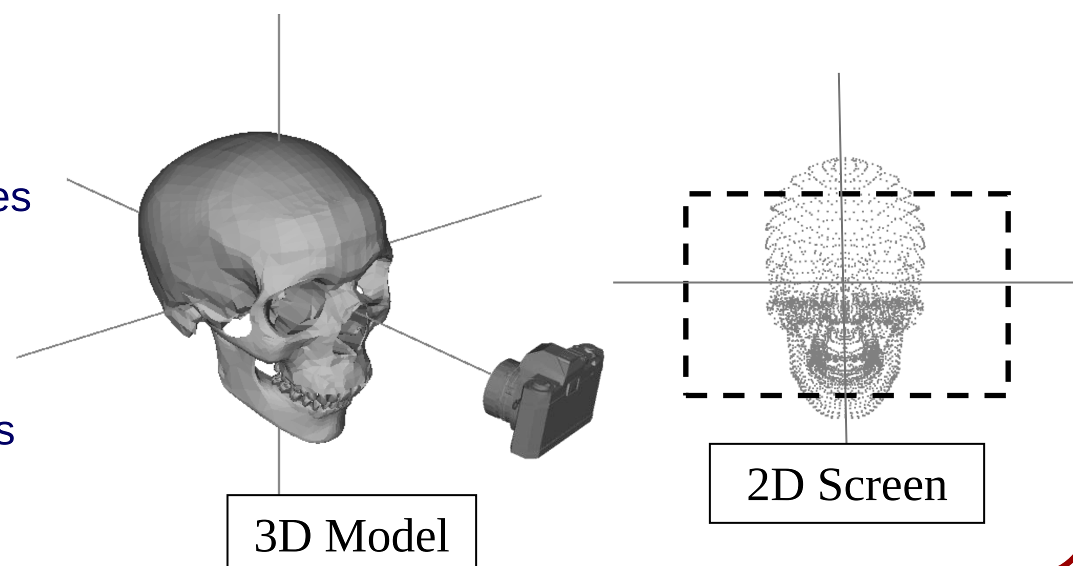
Projection Transformation

↓ 2D Screen Coordinates

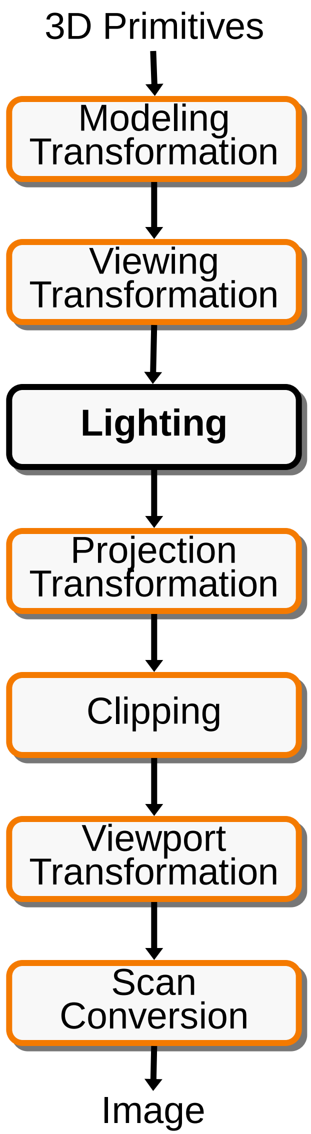
Window-to-Viewport Transformation

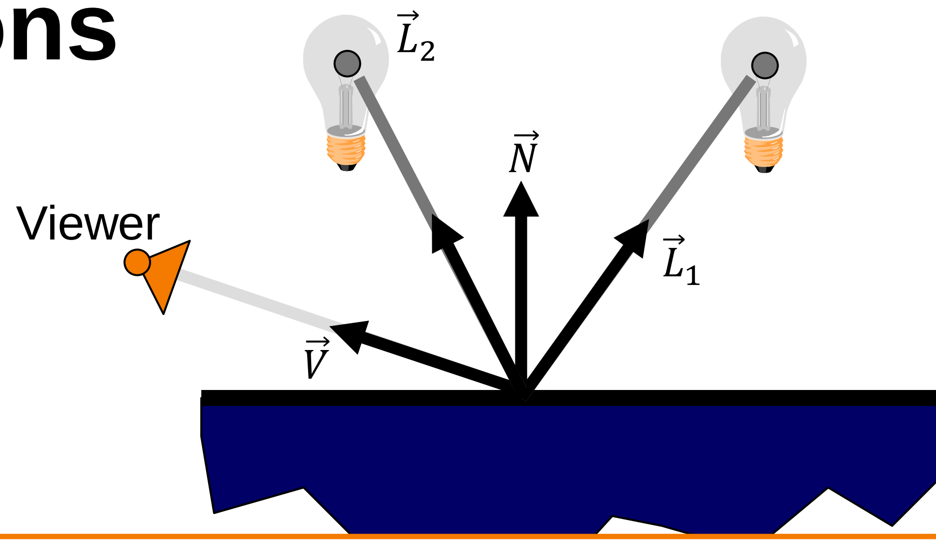
↓ 2D Image Coordinates

(x', y')

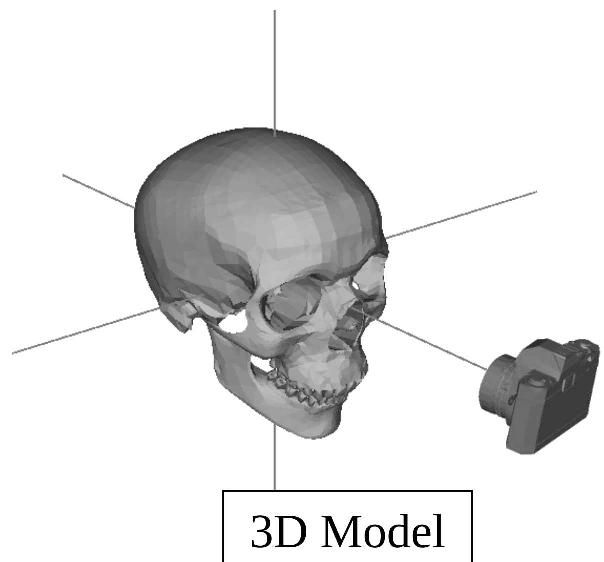


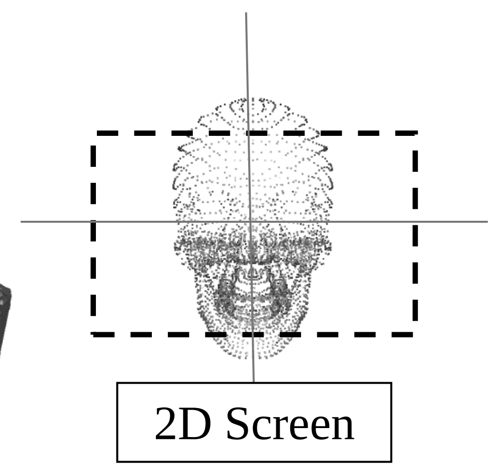
Transformations



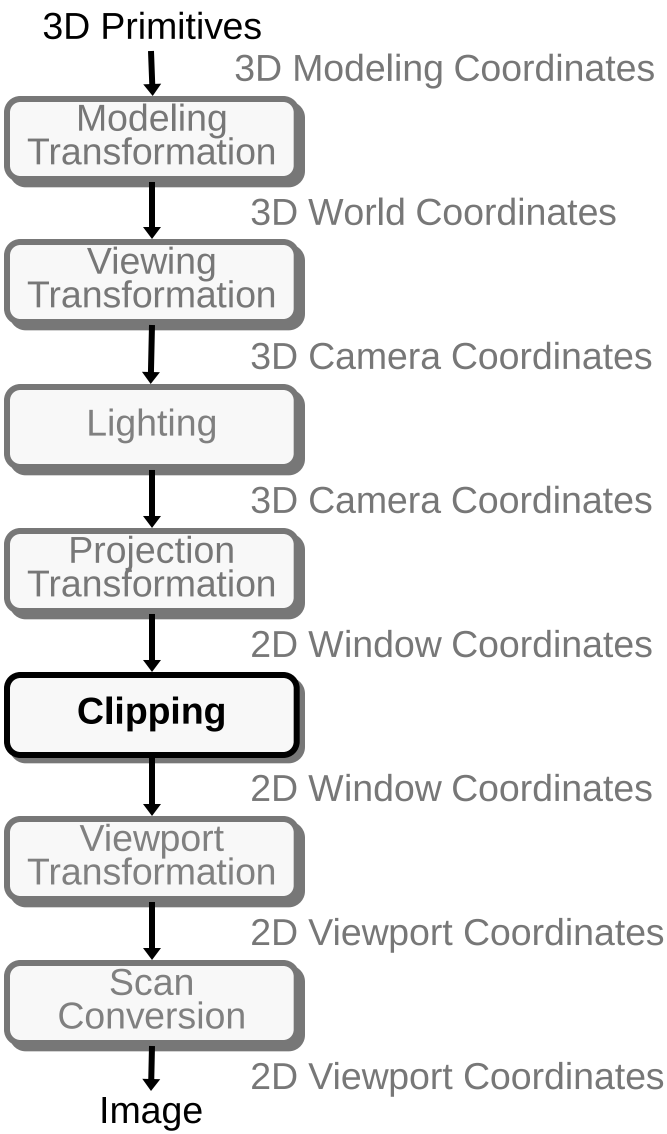


$$I = I_E + \sum_L \left[K_A \cdot I_L^A + \left(K_D \cdot \langle \vec{N}, \vec{L} \rangle + K_S \cdot \langle \vec{V}, \vec{R} \rangle^n\right) \cdot I_L\right]$$



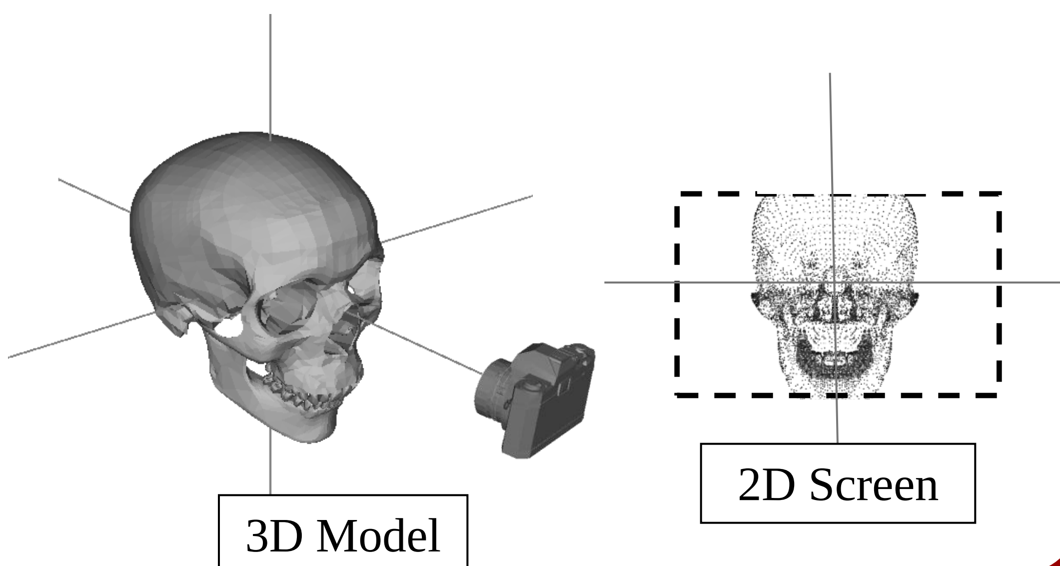


3D Rendering Pipeline (for direct illumination)



At this point we have the:

- Positions of the mesh vertices (including new vertices obtained through clipping)
- Color information at each vertex.
- A list of (possibly clipped) polygons describing the intersection of the projected 3D polygons with the window.



2D Rendering Pipeline

3D Primitives

2D Primitives

Clipping

Scan Conversion

Image

Clip portions of geometric primitives residing outside the window

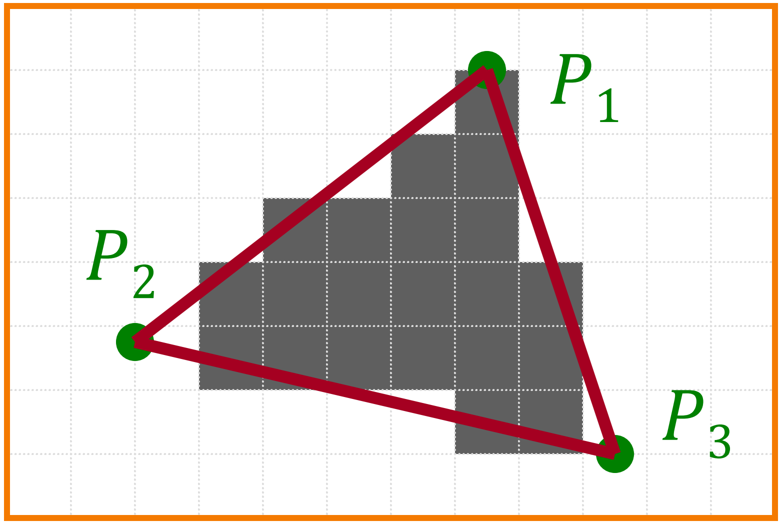
Fill pixels representing primitives in viewport coordinates

Overview

- Scan conversion
  - Figure out which pixels to fill
- Shading
  - Determine a color for each filled pixel
- Depth test
  - Determine when the color of a pixel should be overwritten

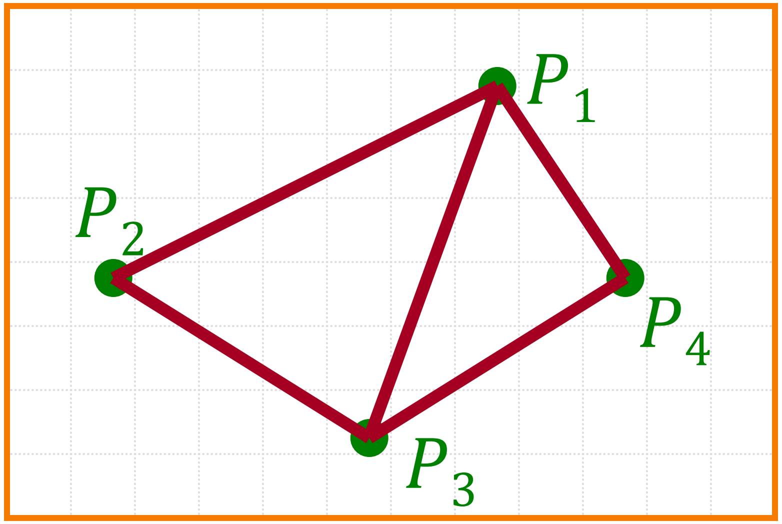
Scan Conversion

Render an image of a geometric primitive (specifically, a triangle) by setting interior pixel colors.



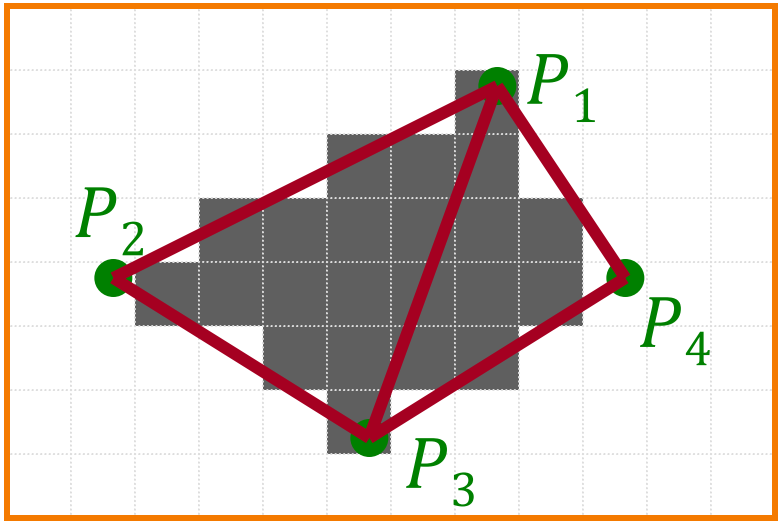
Triangle Scan Conversion

- Properties of a good algorithm
  - Must be fast



Triangle Scan Conversion

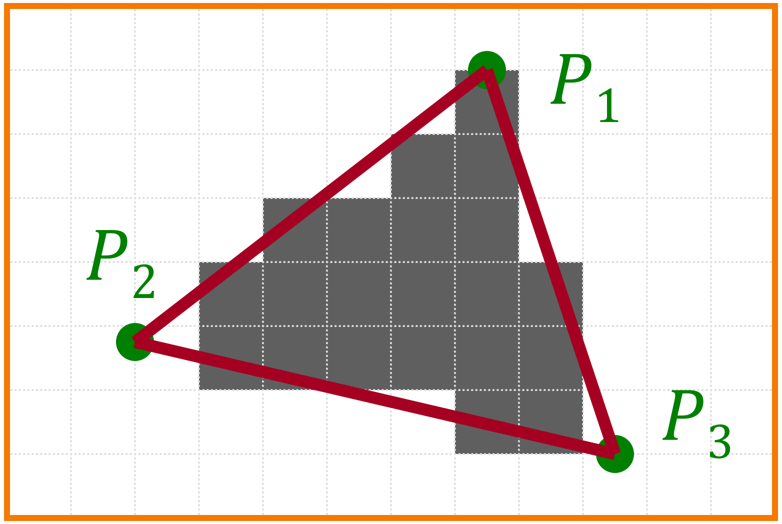
- Properties of a good algorithm
  - Must be fast
  - No cracks between adjacent primitives



Simple Algorithm

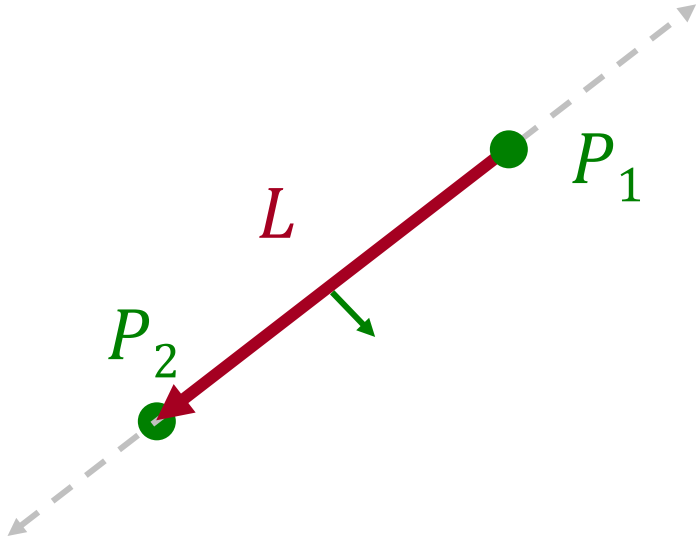
- Color all pixels inside triangle

```
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

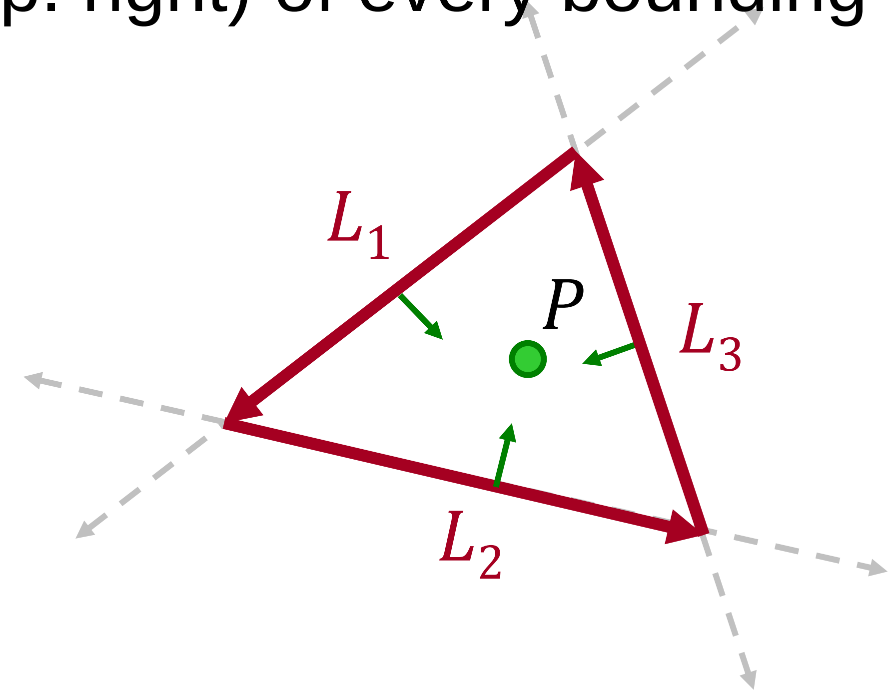
- Test: use implicit equation for a line
  - On line: $ax + by + c = 0$
  - To the right: $ax + by + c < 0$
  - To the left: $ax + by + c > 0$



Inside Triangle Test

Triangle vertices are ordered counter-clockwise (resp. clockwise):

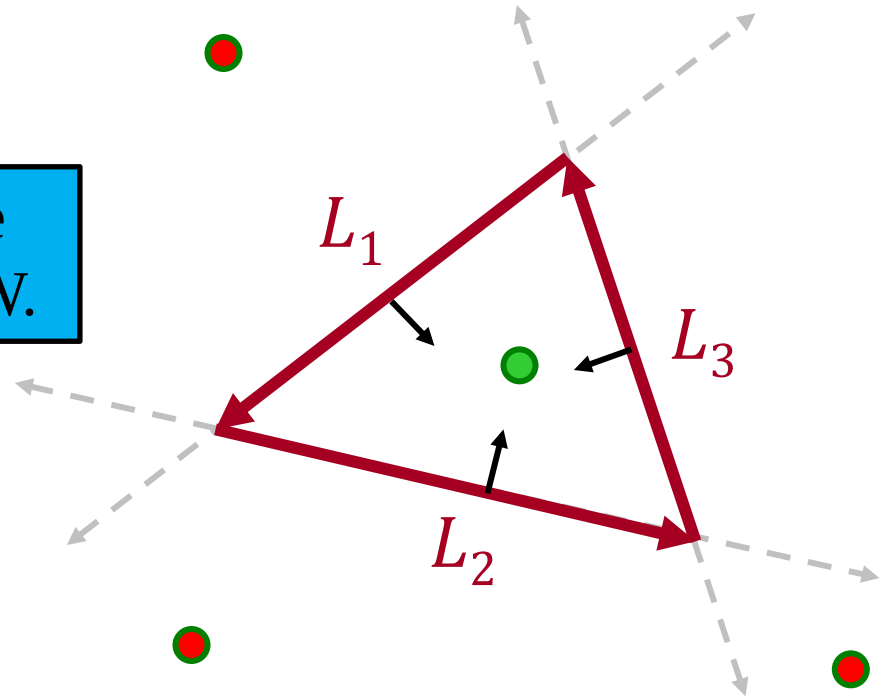
⇒ Since triangles are convex, an interior point must be to the left (resp. right) of every bounding line.



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;
}
```

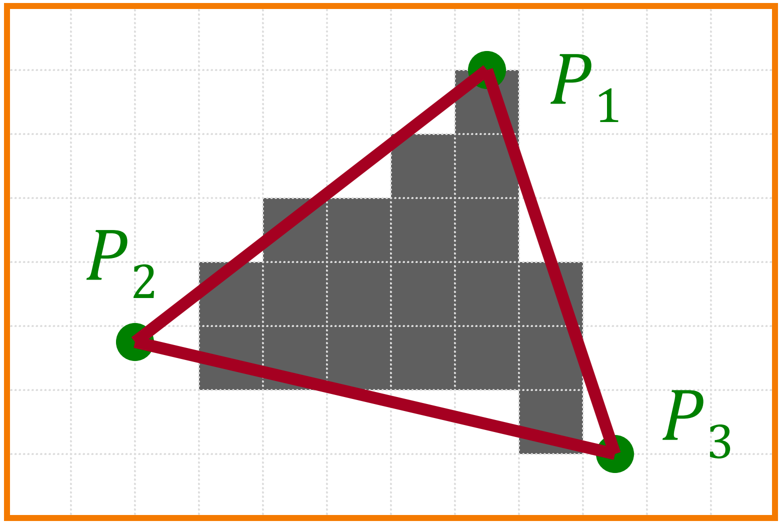
Assumes triangle orientation is CCW.



Simple Algorithm

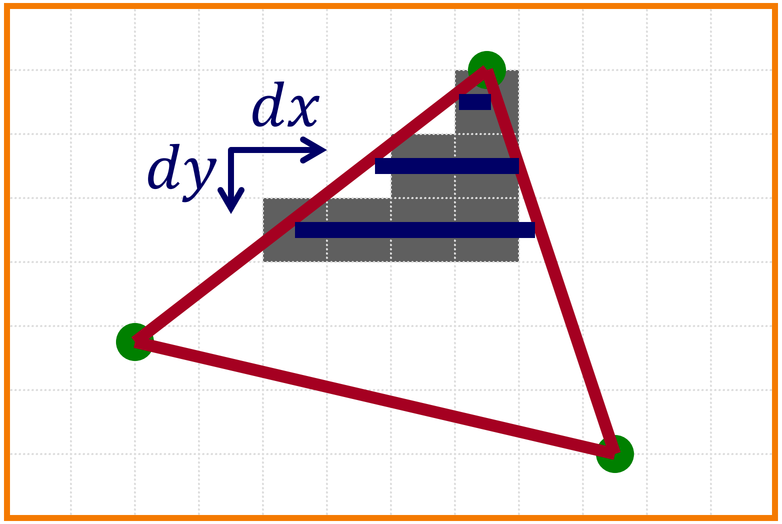
- What is bad about this algorithm?

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



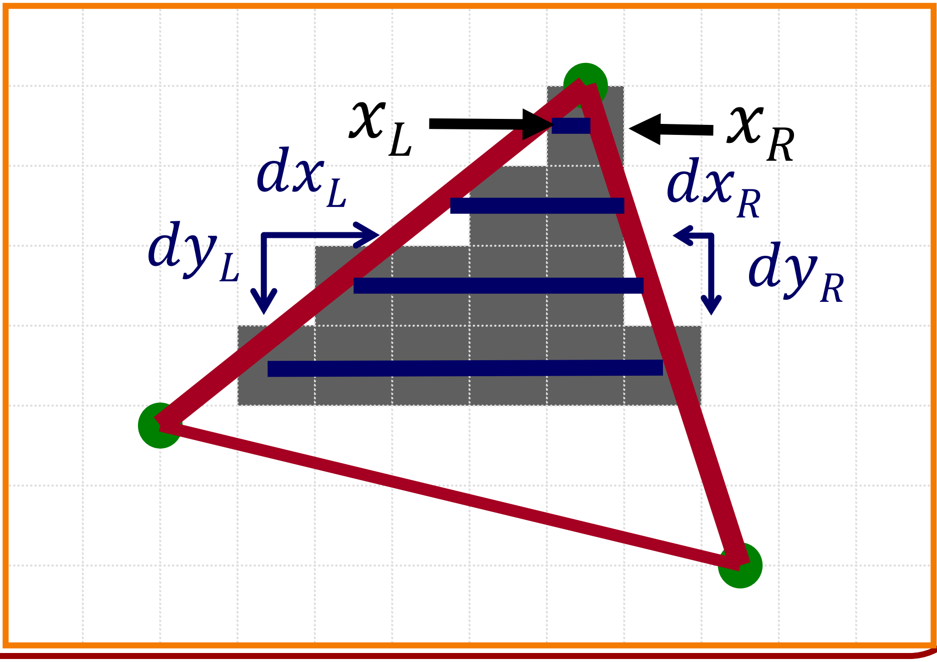
Triangle Sweep-Line Algorithm

- 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.



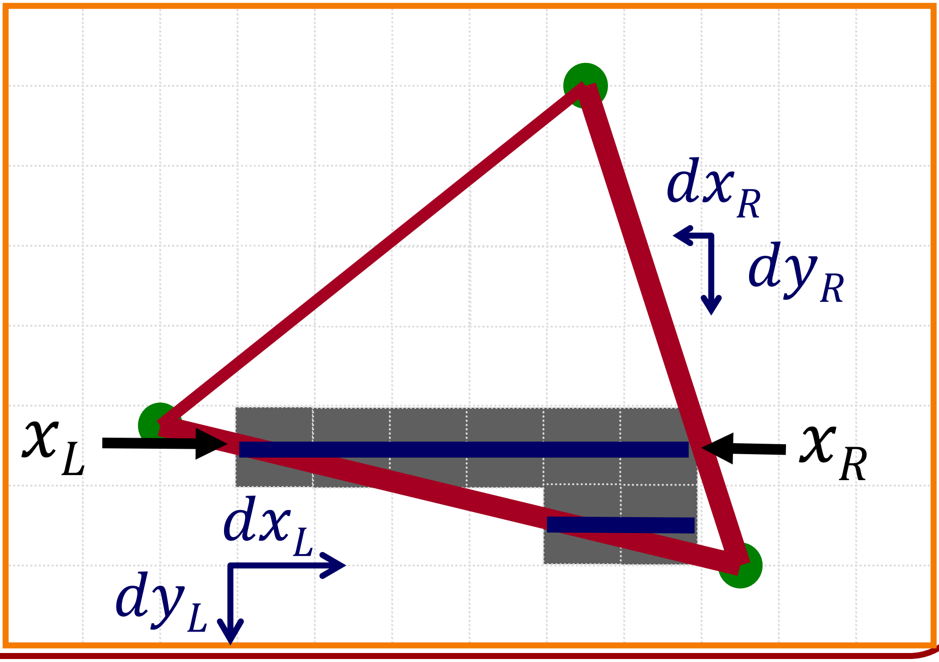
Triangle Sweep-Line Algorithm

```
void ScanTriangle( Triangle T , Color rgba )
{
    for both edge pairs
    {
        initialize xL, xR, y;
        compute dxL/dyL and dxR/dyR;
        until y reaches the first end-point
            for( int x=xL ; x<=xR ; x++ ) SetPixel( x , y , rgba );
            xL += dxL/dyL;
            xR += dxR/dyR;
            y++;
    }
}
```



Triangle Sweep-Line Algorithm

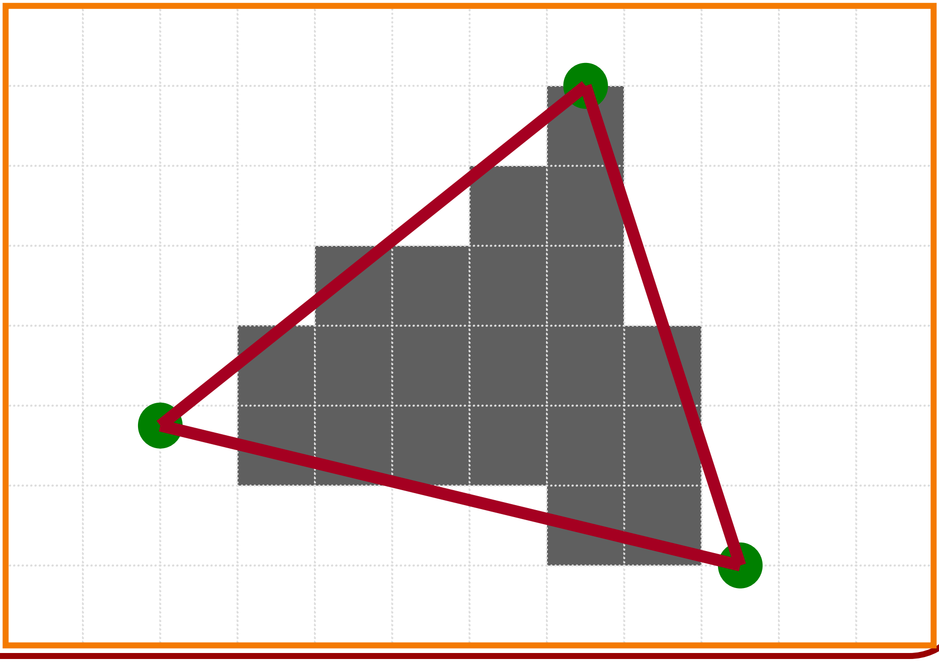
```
void ScanTriangle( Triangle T , Color rgba )
{
   for both edge pairs
   {
      initialize xL, xR, y;
      compute dxL/dyL and dxR/dyR;
      until y reaches the first end-point
         for( int x=xL ; x<=xR ; x++ ) SetPixel( x , y , rgba );
         xL += dxL/dyL;
         xR += dxR/dyR;
         y++;
   }
}
```



Triangle Sweep-Line Algorithm

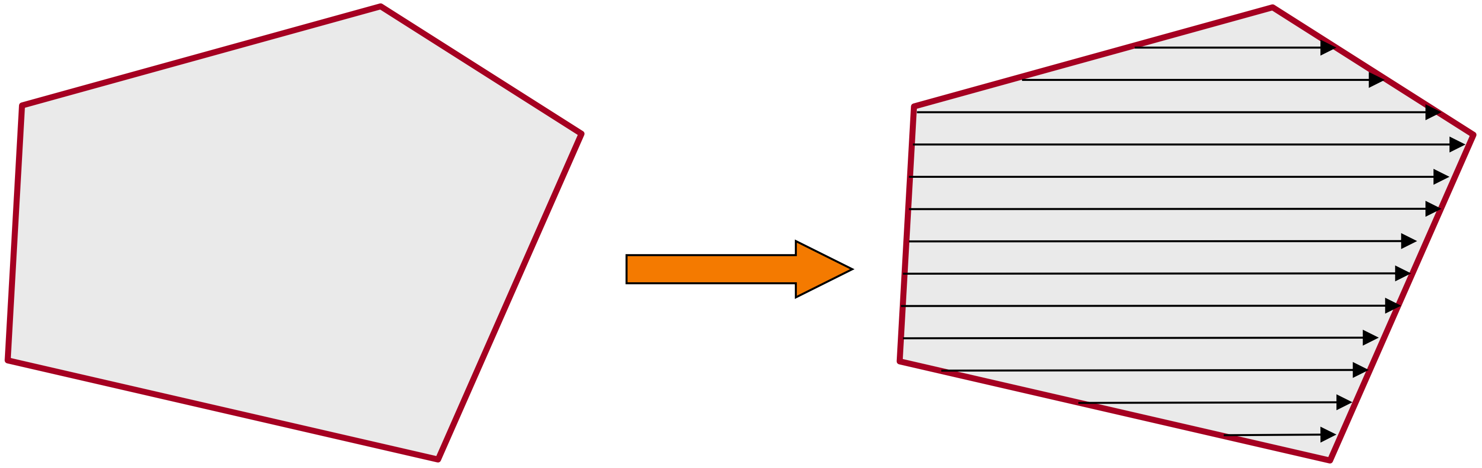
```
void ScanTriangle( Triangle T , Color rgba )
{
    for both edge pairs
    {
        initialize xL, xR, y;
        compute dxL/dyL and dxR/dyR;
        until y reaches the first end-point
            for( int x=xL ; x<=xR ; x++ ) SetPixel( x , y , rgba );
            xL += dxL/dyL;
            xR += dxR/dyR;
            y++;
    }
}
```

Bresenham's algorithm works similarly, but only requires integer arithmetic.



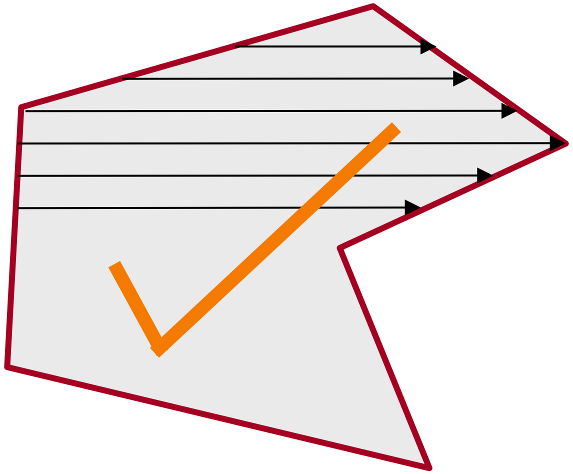
Polygon Scan Conversion

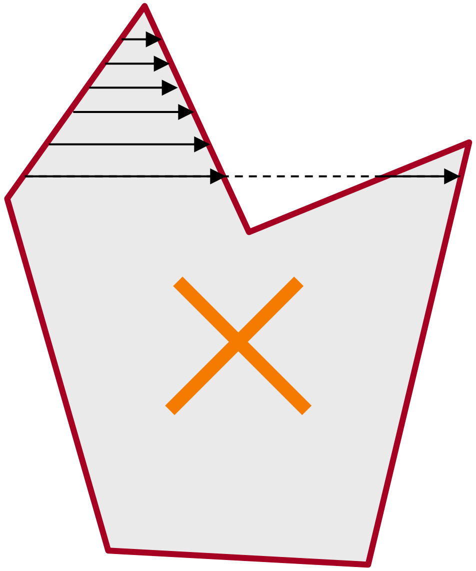
- Will this method work for convex polygons?
  - Yes, since each scan line will only intersect the polygon at two points.



Polygon Scan Conversion

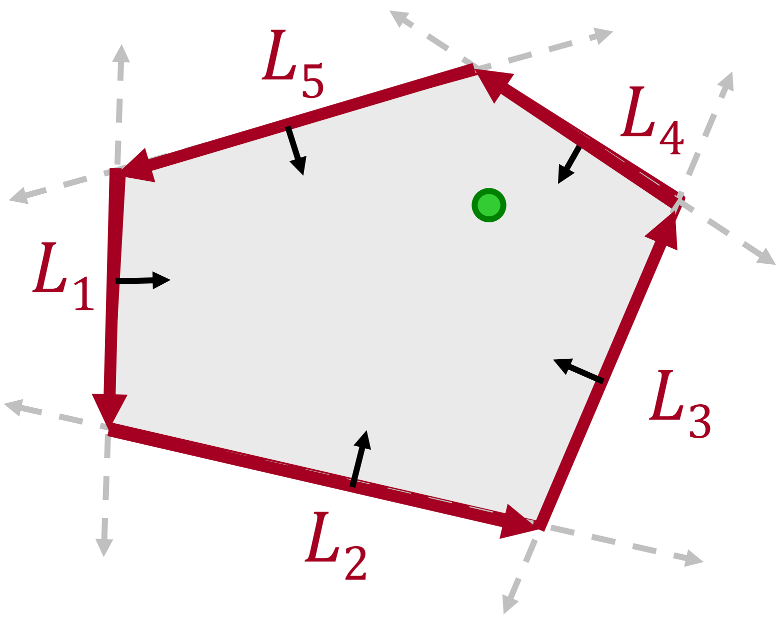
- How about these polygons?



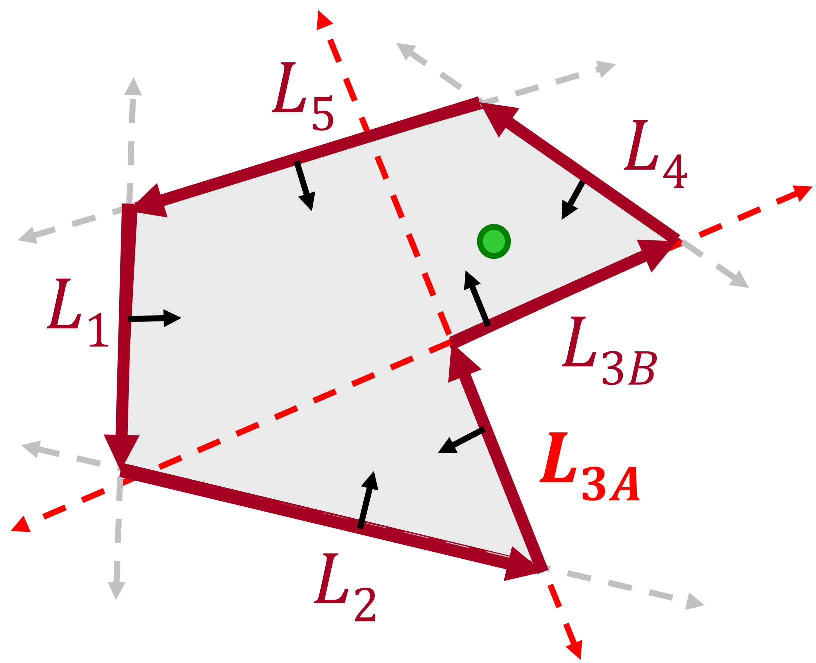


Polygon Scan Conversion

- Need better test for points inside polygon
  - Triangle sweep-line algorithm only generalizes to <u>convex</u> polygons



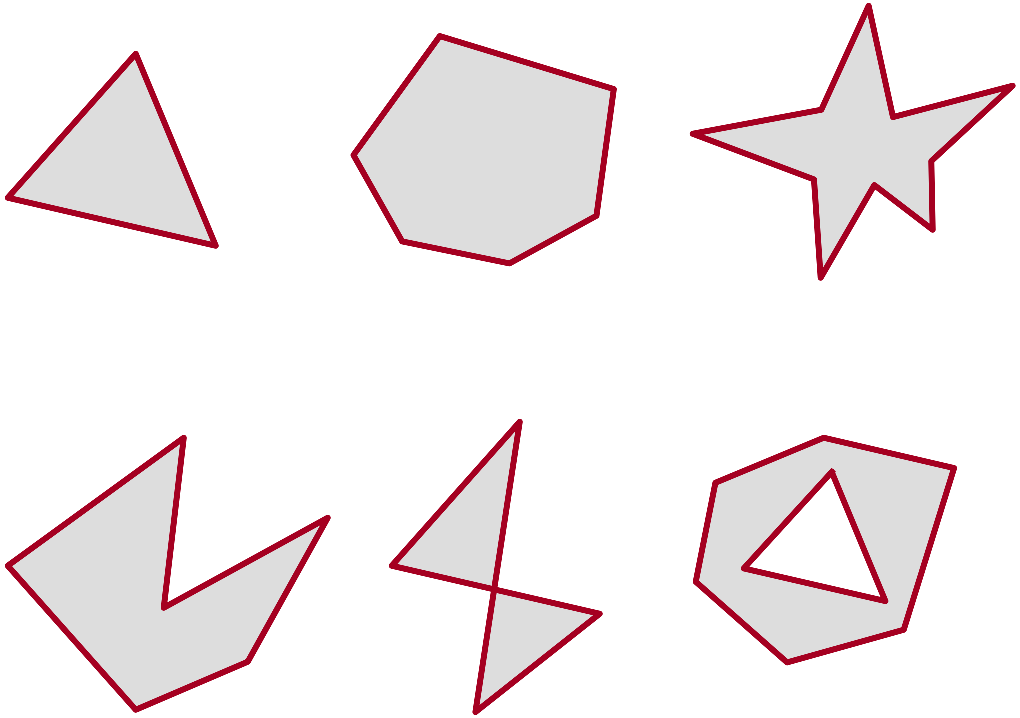
Convex Polygon



Concave Polygon

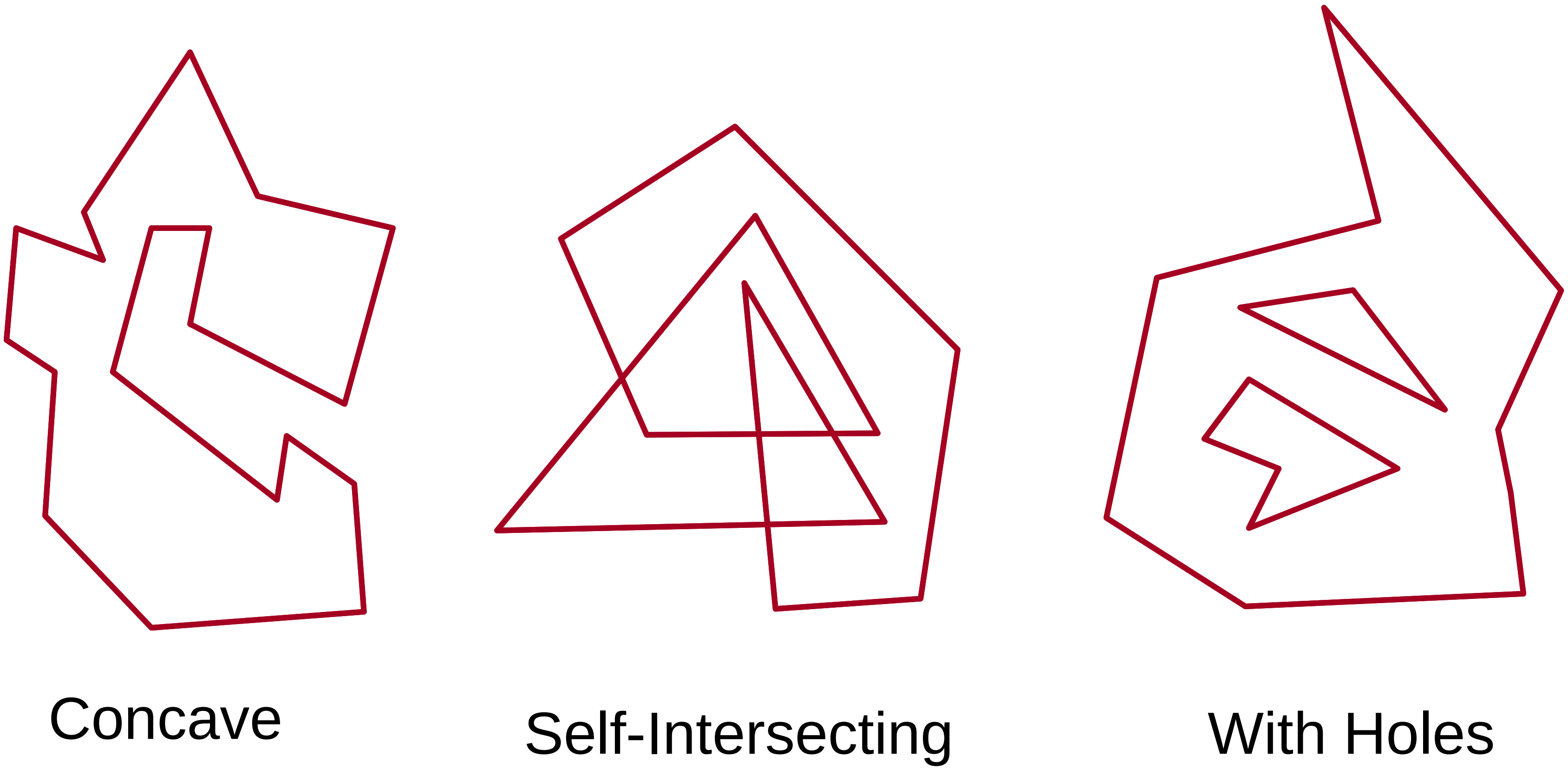
Polygon Scan Conversion

- Fill pixels inside a polygon
  - Triangle
  - Convex
  - Star-shaped
  - Concave
  - Self-intersecting
  - Holes



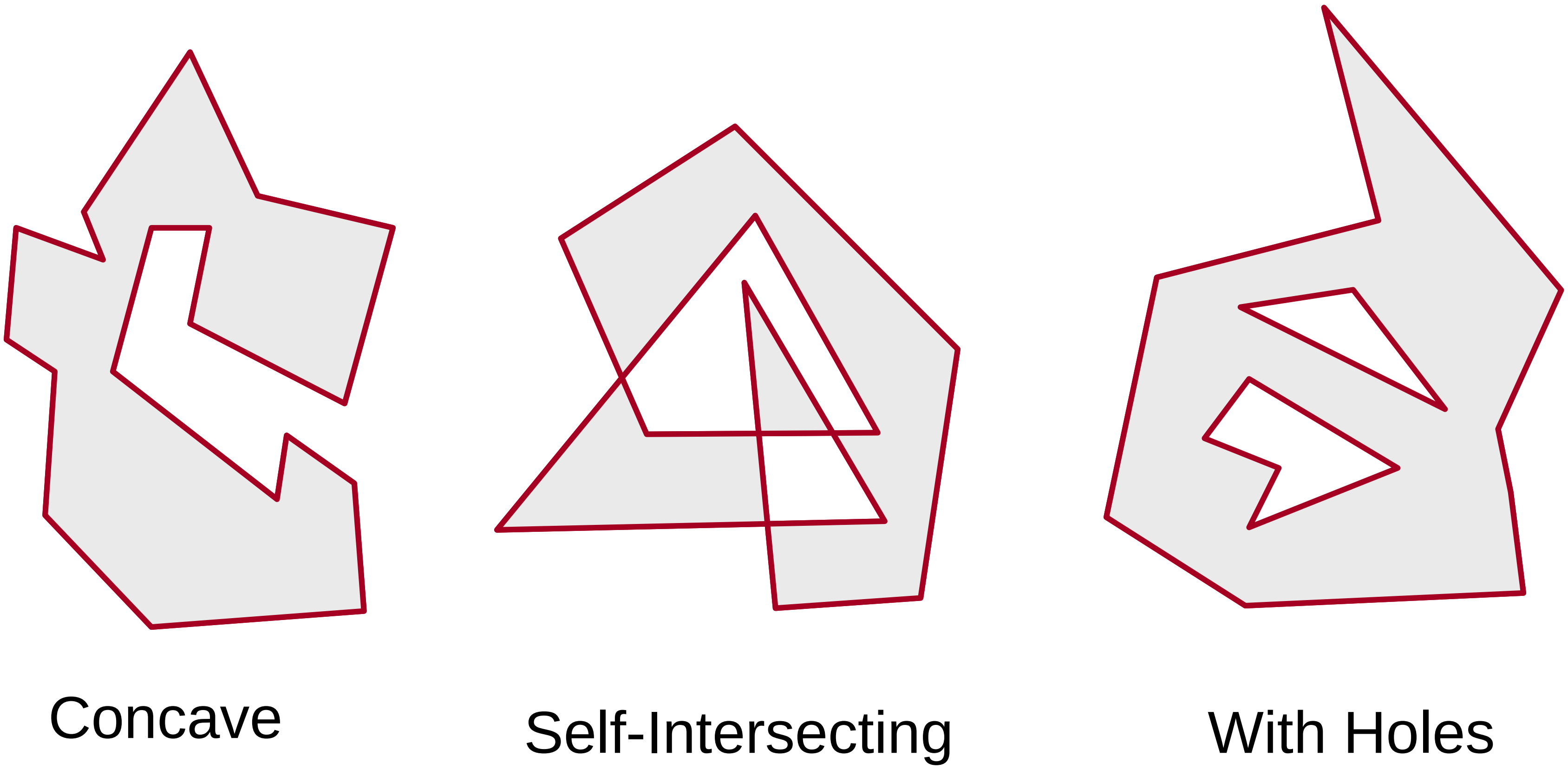
Inside Polygon Rule

- What is a good rule for which pixels are inside?



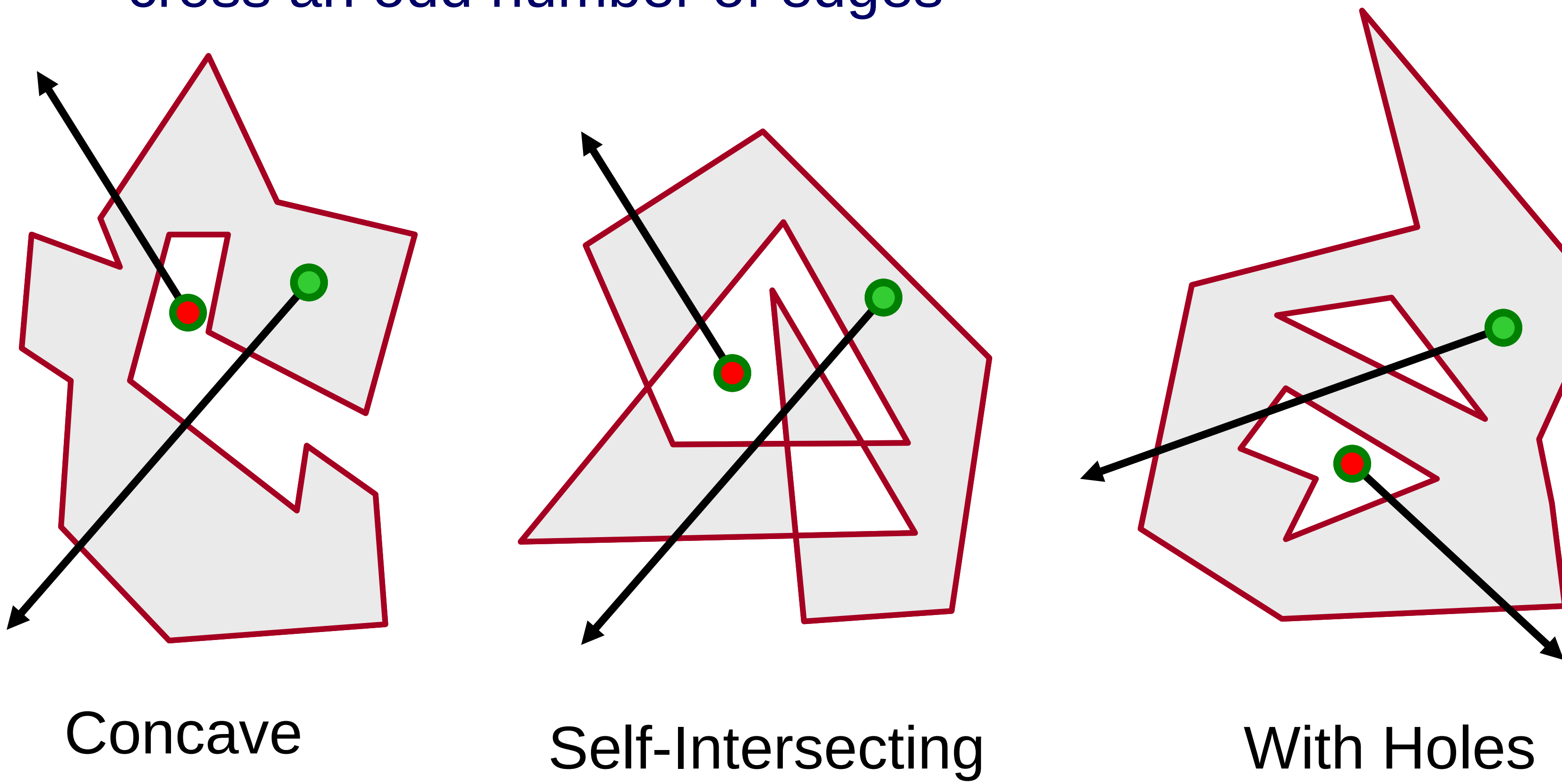
Inside Polygon Rule

- What is a good rule for which pixels are inside?



Inside Polygon Rule

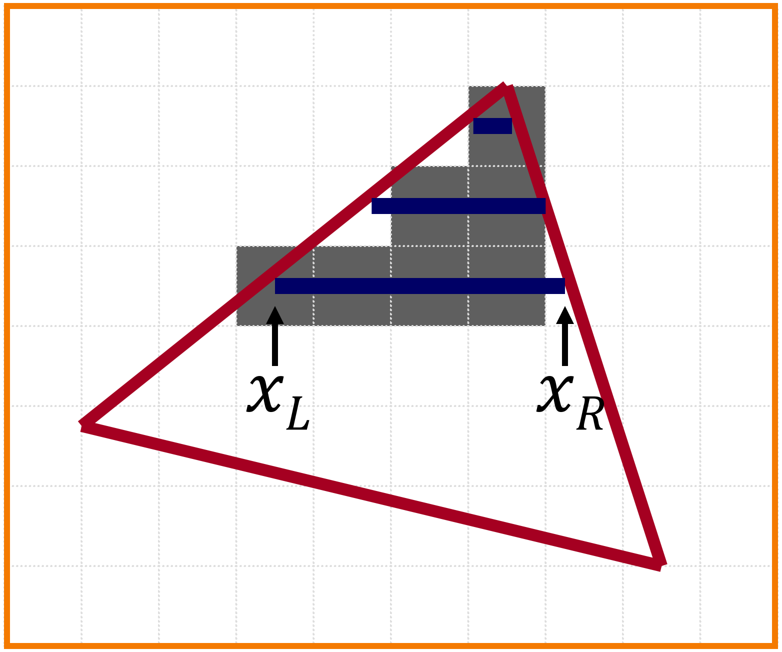
- Odd-parity rule
  - Any ray from inside the shape, out to infinity, must cross an odd number of edges



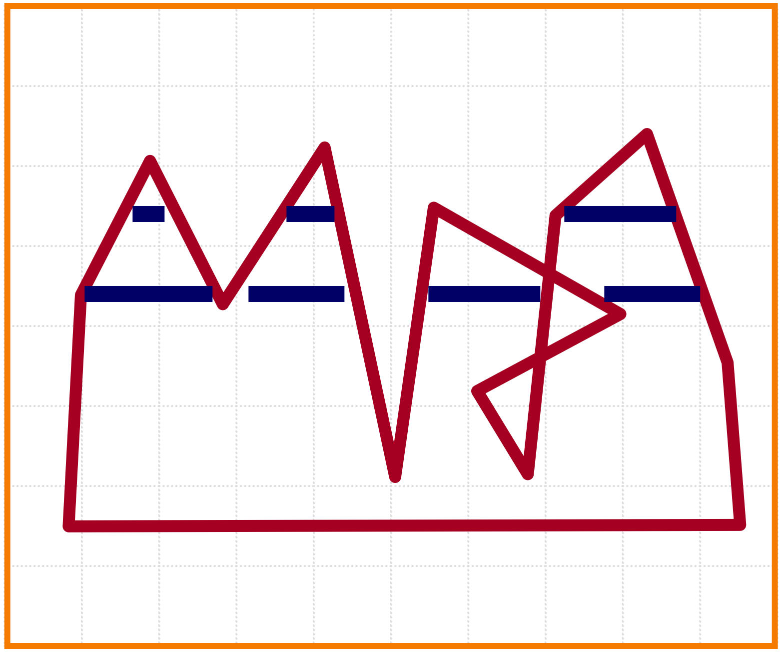
Polygon Sweep-Line Algorithm

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

- Takes advantage of scan line coherence



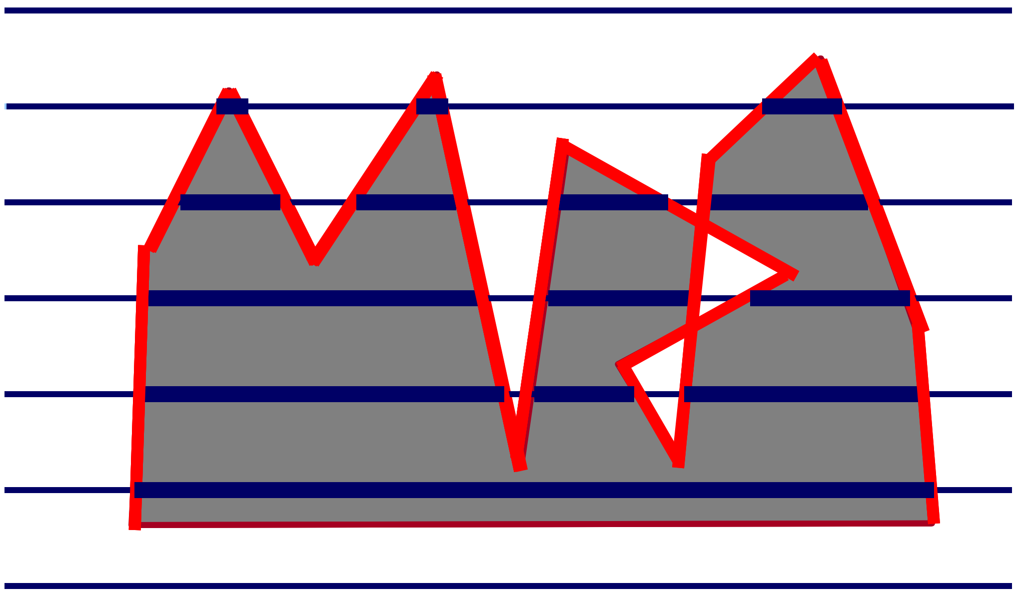
Triangle



Polygon

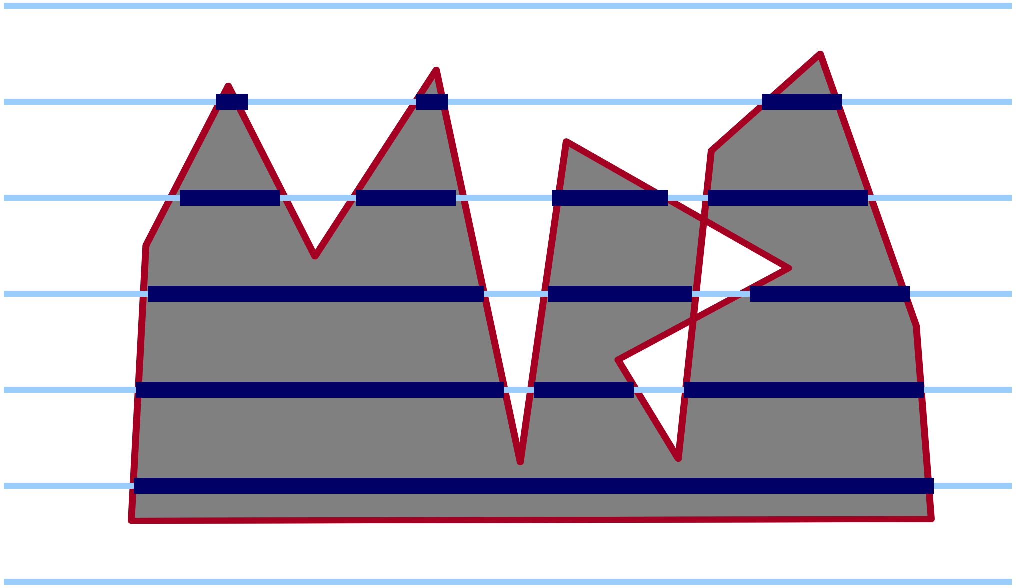
Polygon Sweep-Line Algorithm

```
void ScanPolygon( Polygon P , Color rgba )
{
    sort edges by maxy
    initialize active-edge-list as empty
    for each scanline ( top-to-bottom )
    {
        insert/remove edges from active-edge-list
        sort active edges by intersection with scanline
        for each succesive pair of edge-points (left-to-right)
            SetPixels( x_i , x_i+1 , y , rgba );
    }
}
```



Polygon Sweep-Line Algorithm

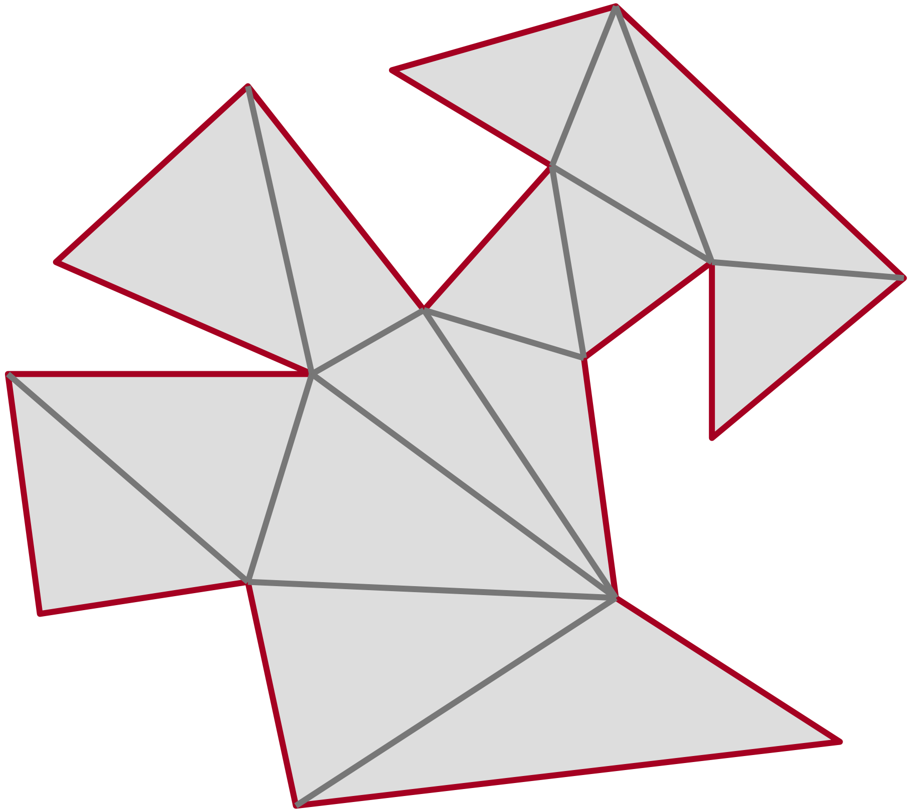
```
void ScanPolygon( Polygon P , Color rgba )
{
    sort edges by maxy
    initialize active-edge-list as empty
    for each scanline ( top-to-bottom )
    {
        insert/remove edges from active-edge-list
        sort active edges by intersection with scanline
        for each succesive pair of edge-points (left-to-right)
            SetPixels( x_i , x_i+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



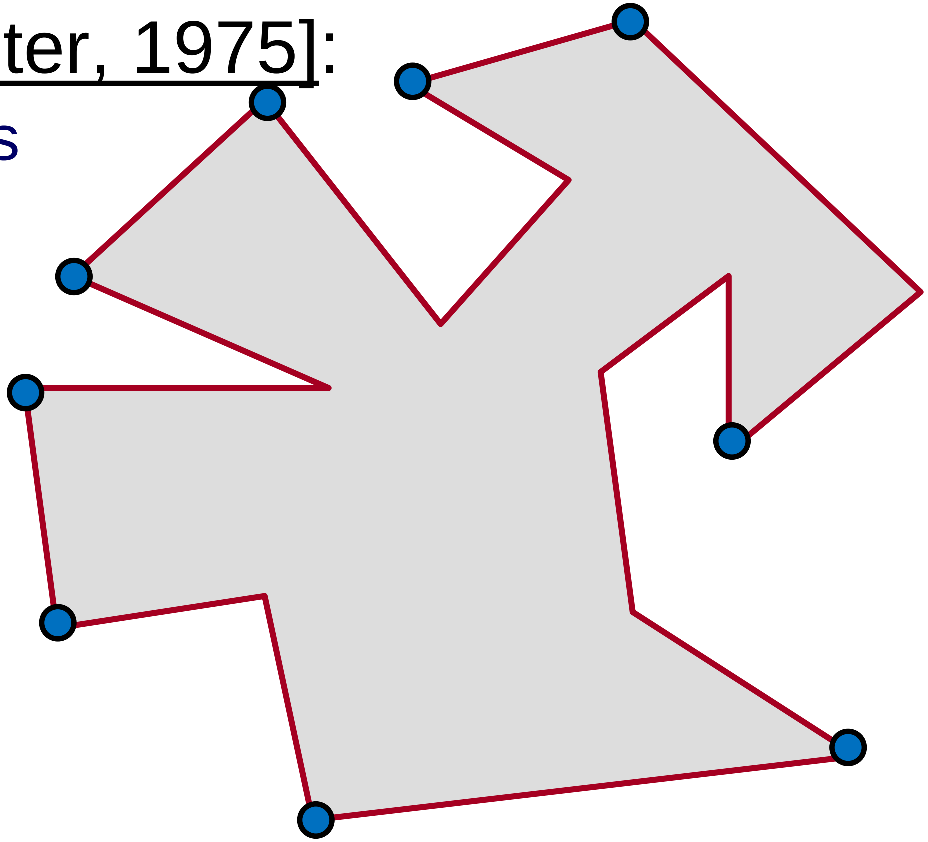
Polygon Scan Conversion

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:

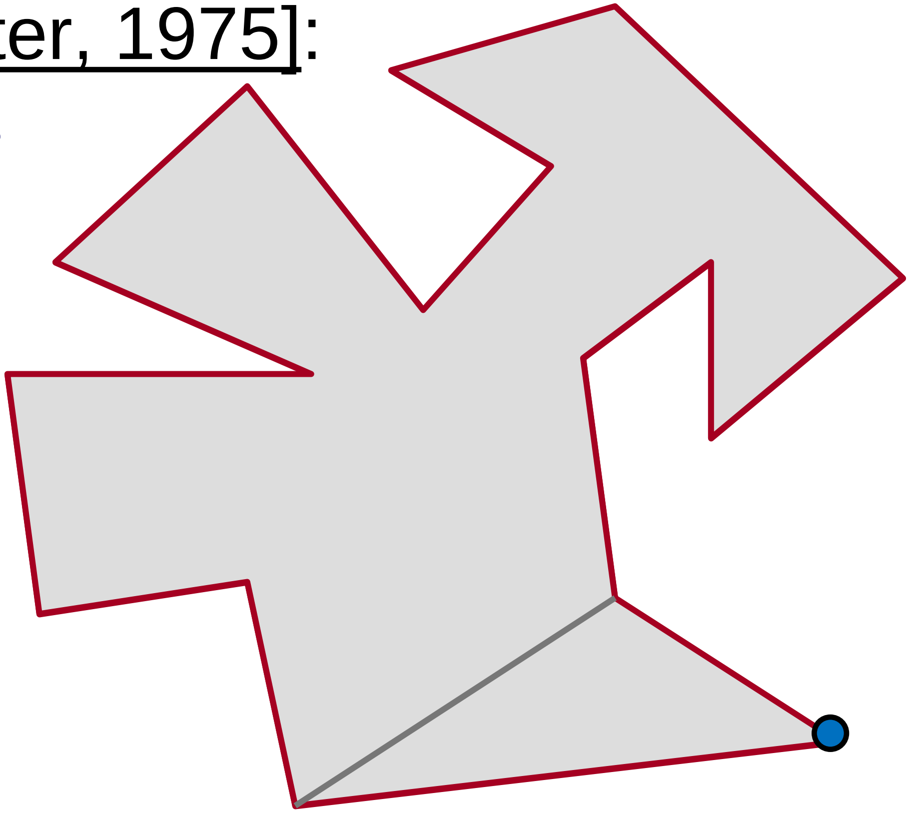
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

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:

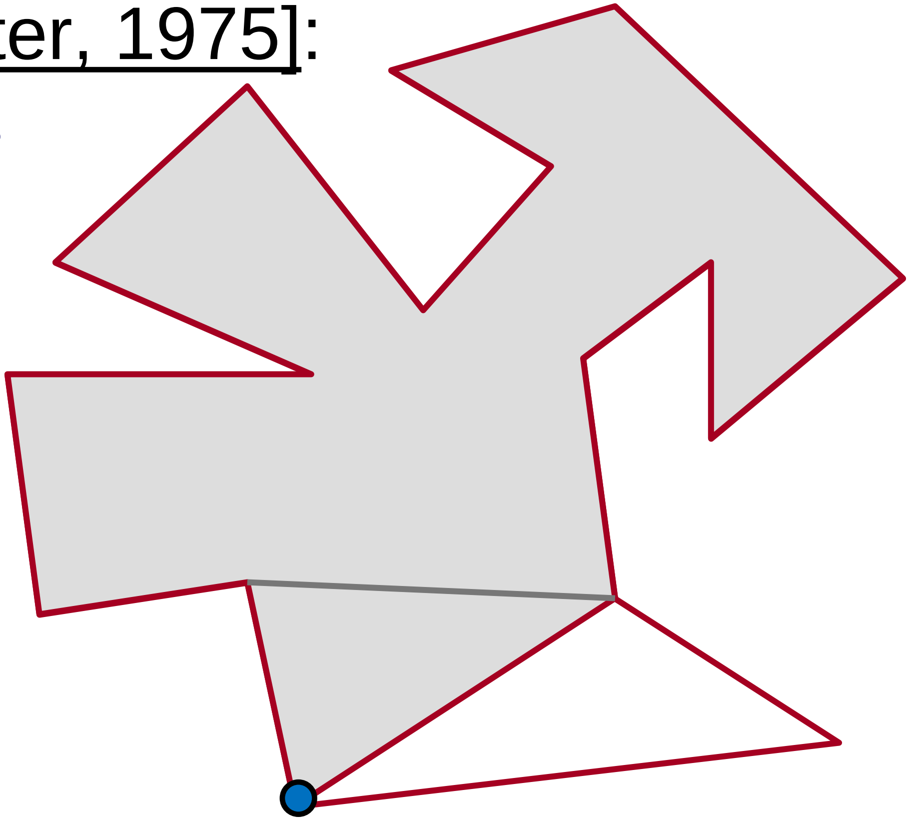
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

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:

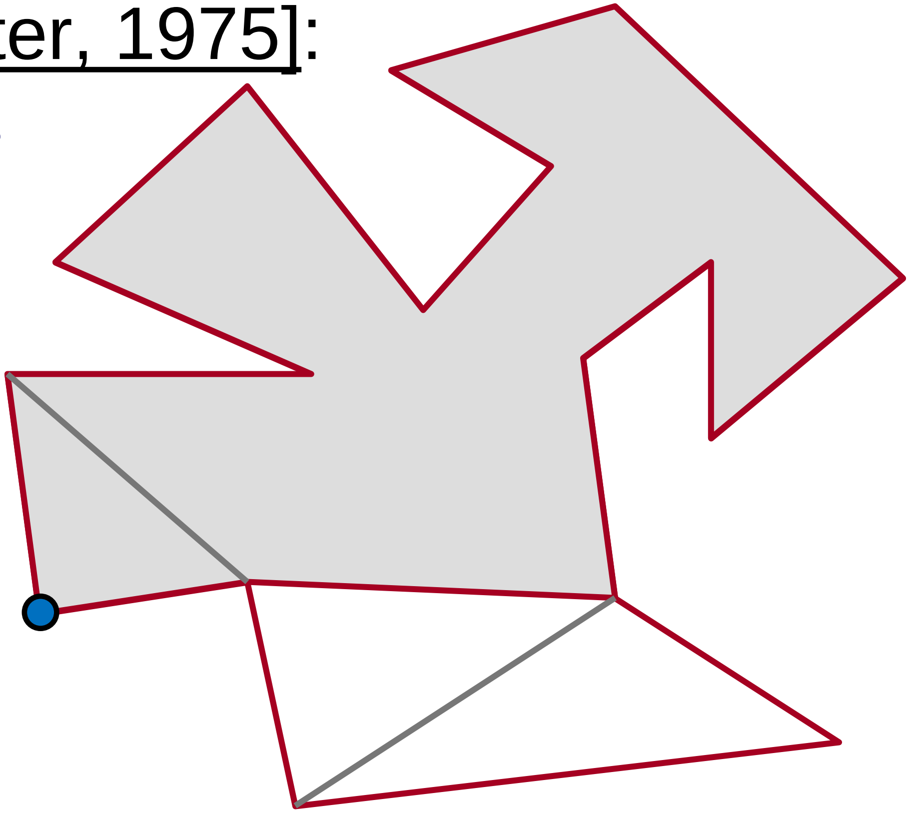
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

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:

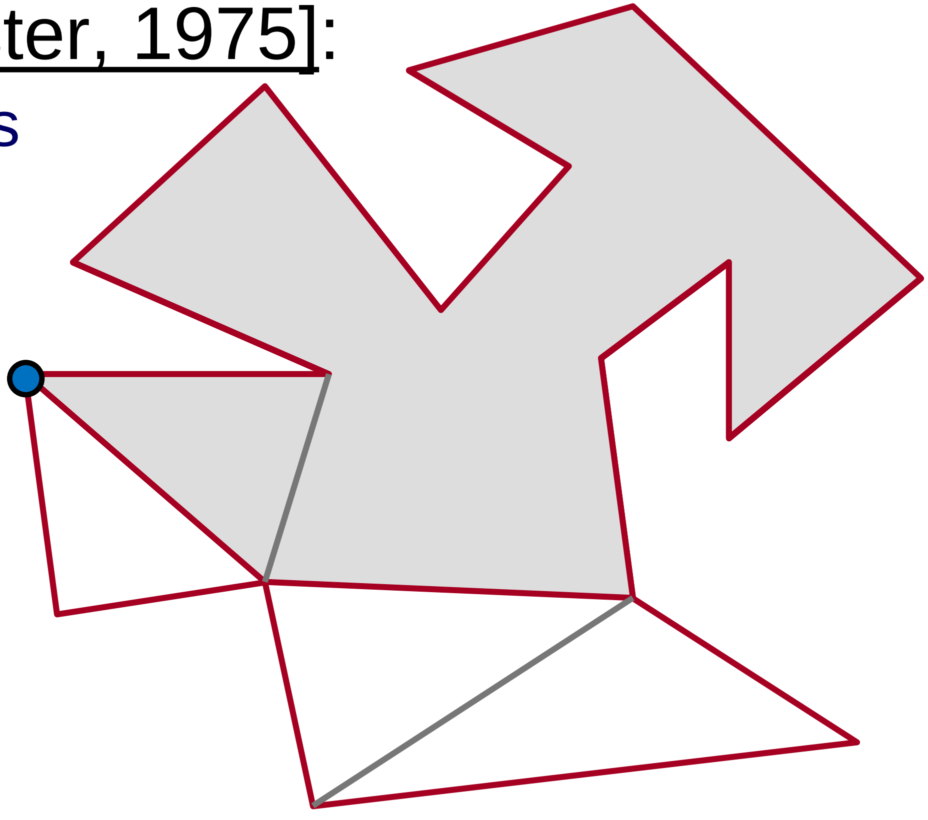
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

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:

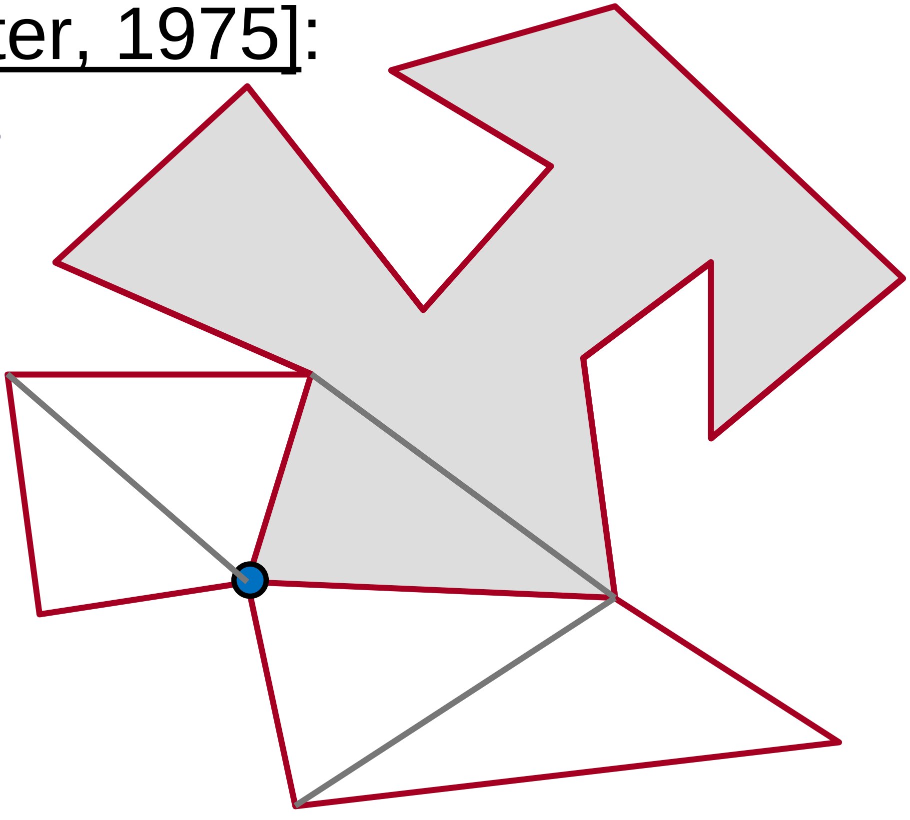
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

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:

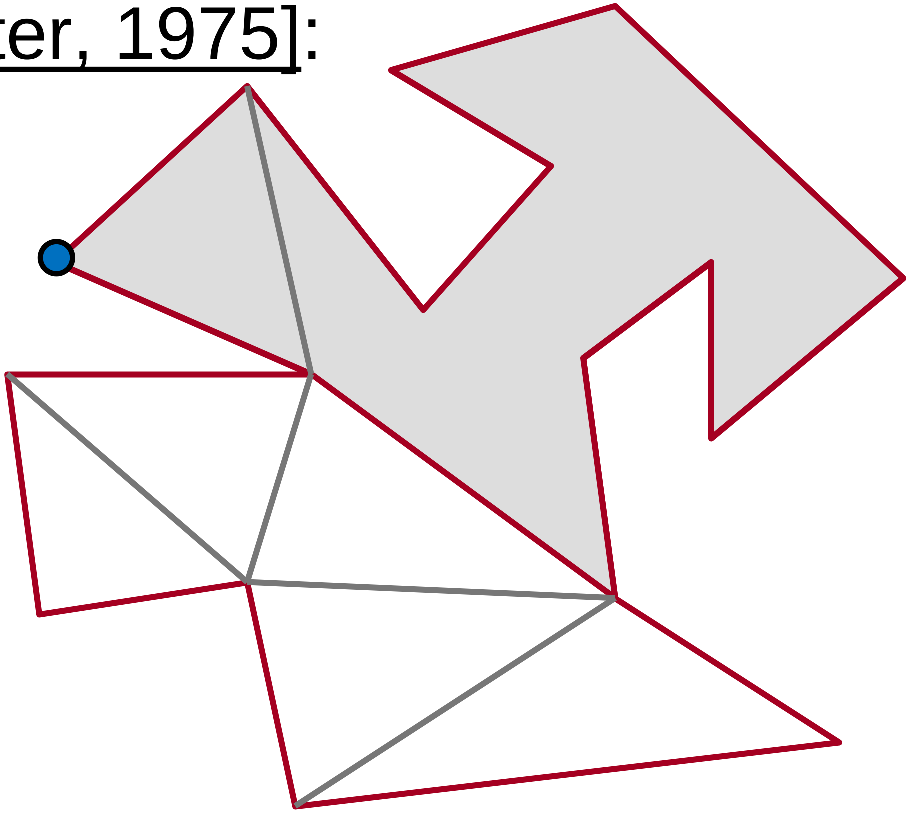
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.

Algorithm:

1. Clip off an ear
2. Repeat



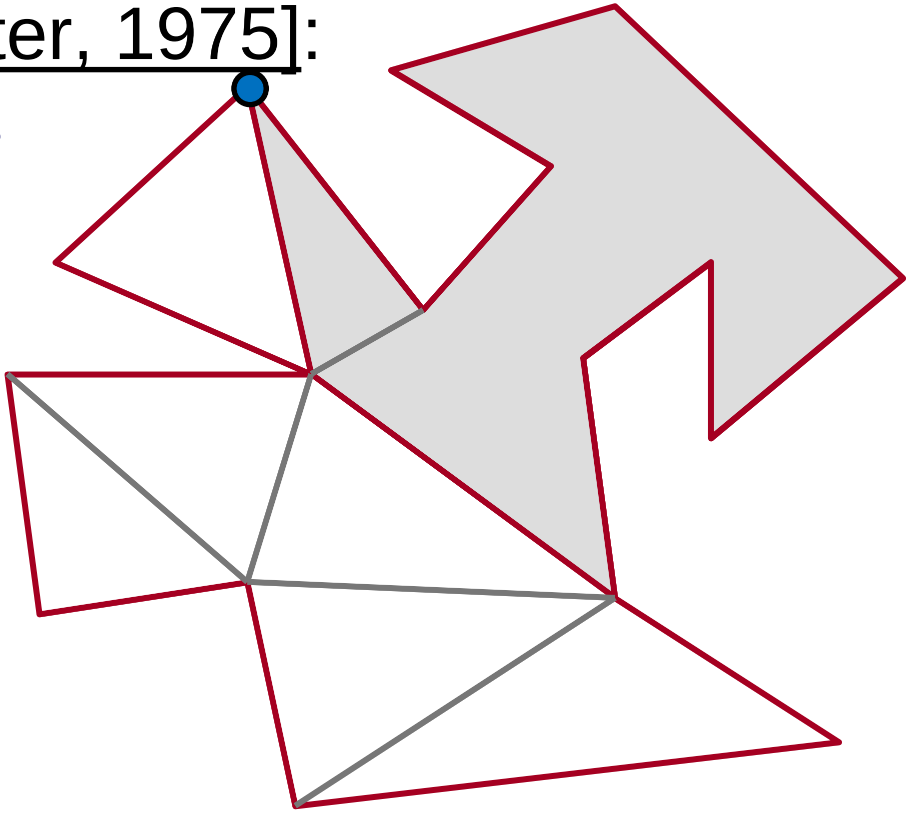
Polygon Scan Conversion

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:

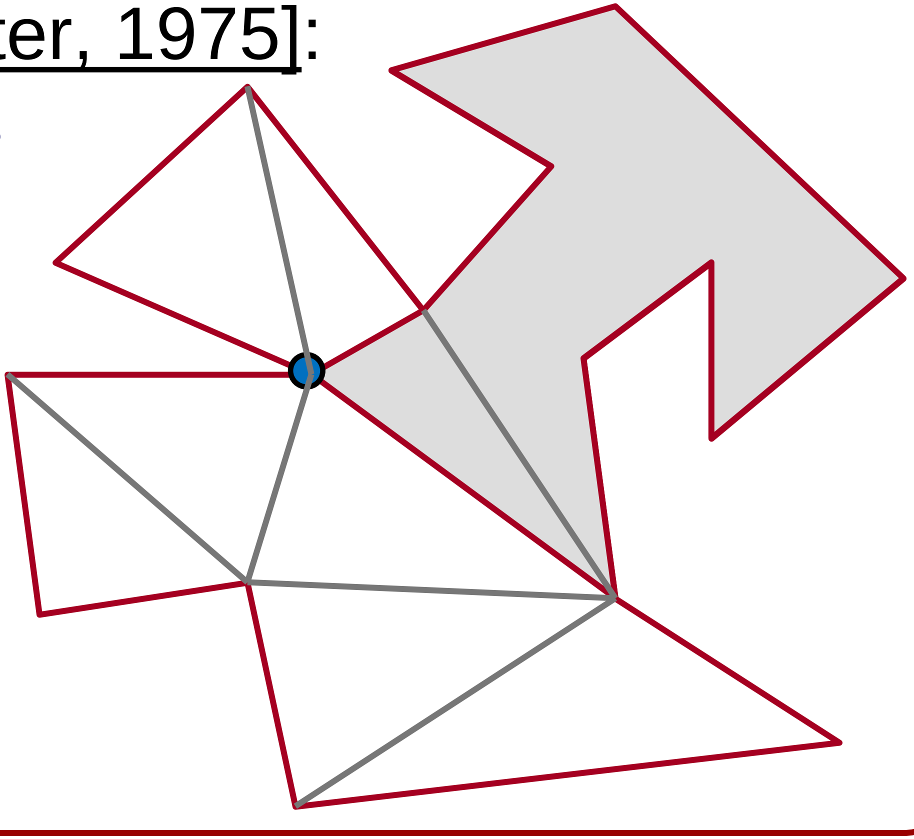
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

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:

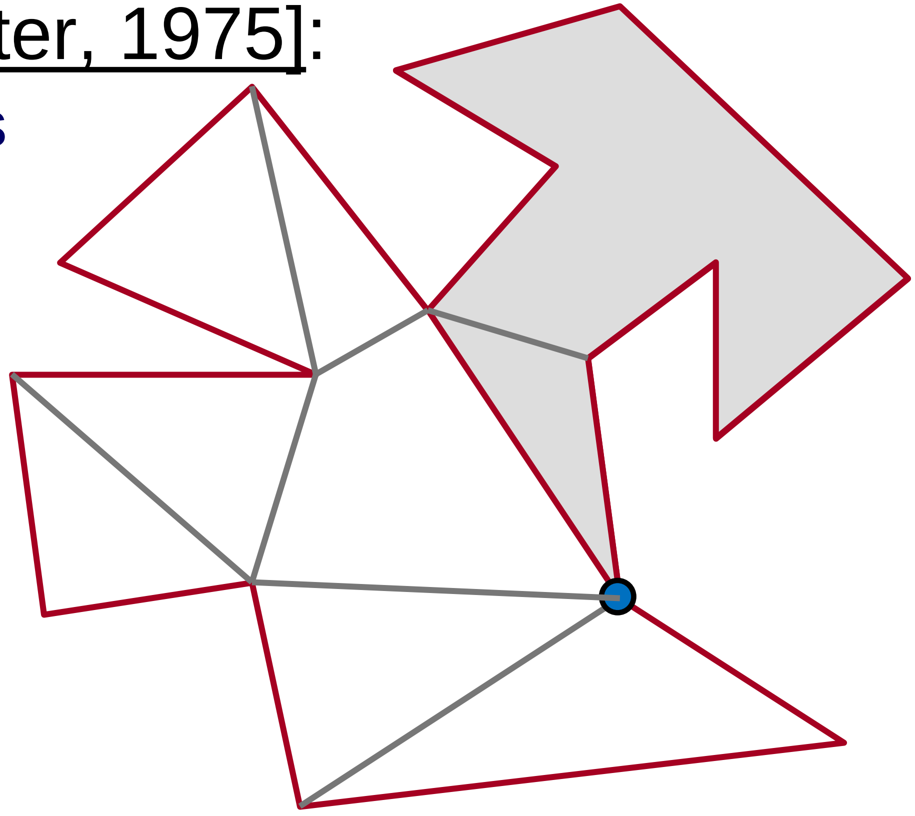
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.

Algorithm:

1. Clip off an ear
2. Repeat



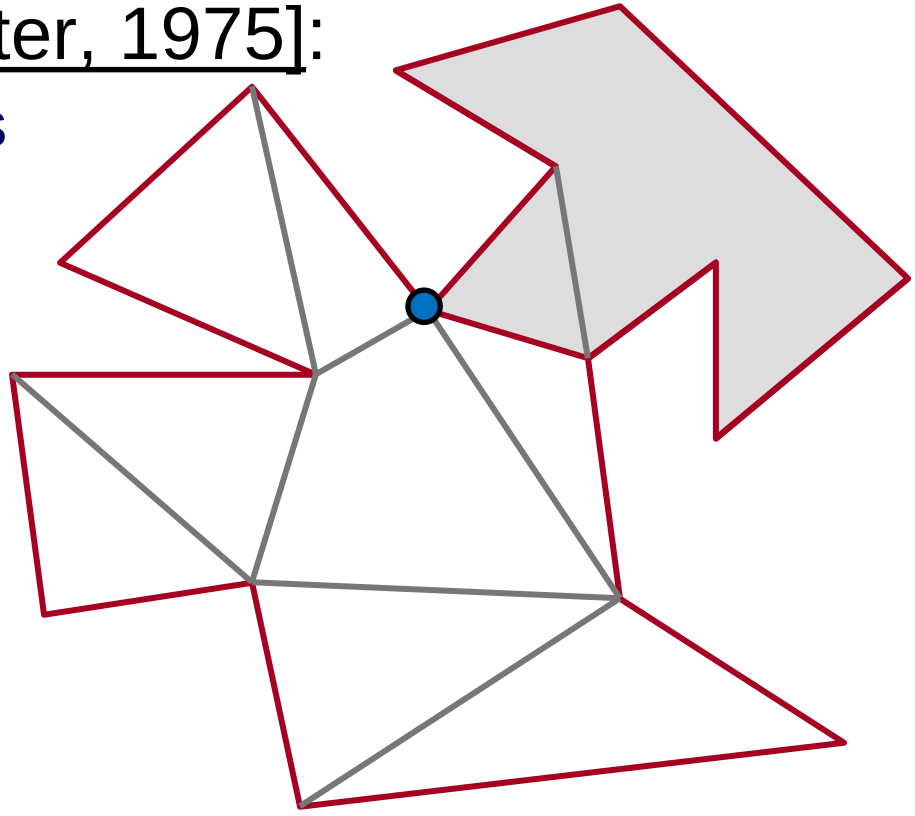
Polygon Scan Conversion

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:

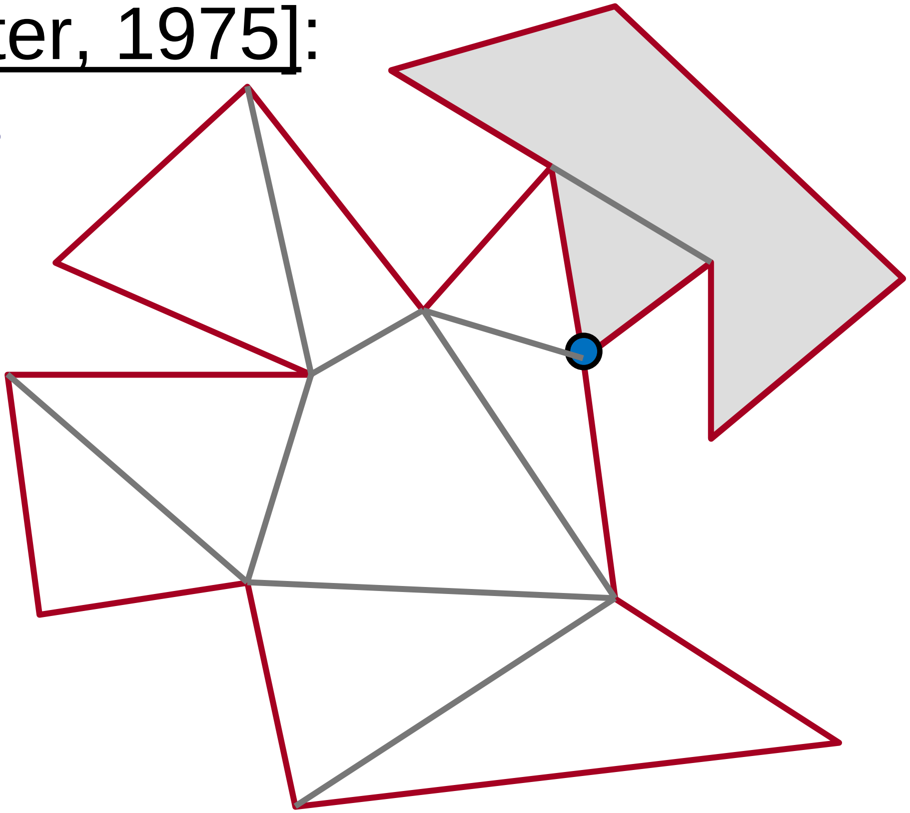
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

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:

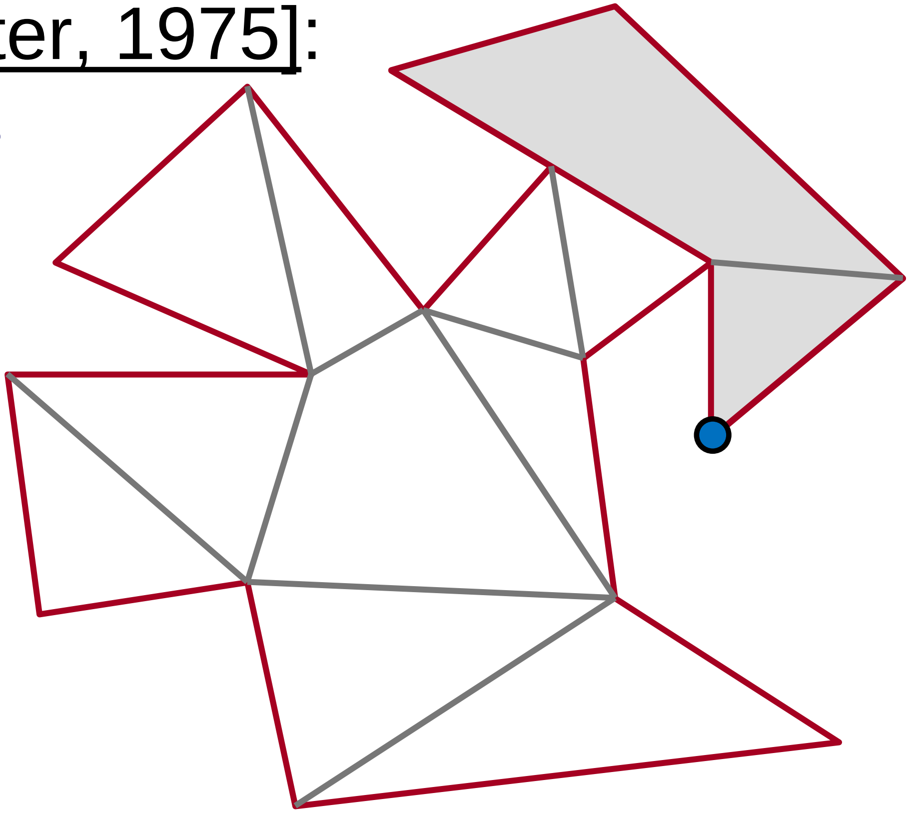
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

Two Ear Theorem [Meister, 1975]:

Every simple polygon has at least two ears.

Algorithm:

1. Clip off an ear
2. Repeat



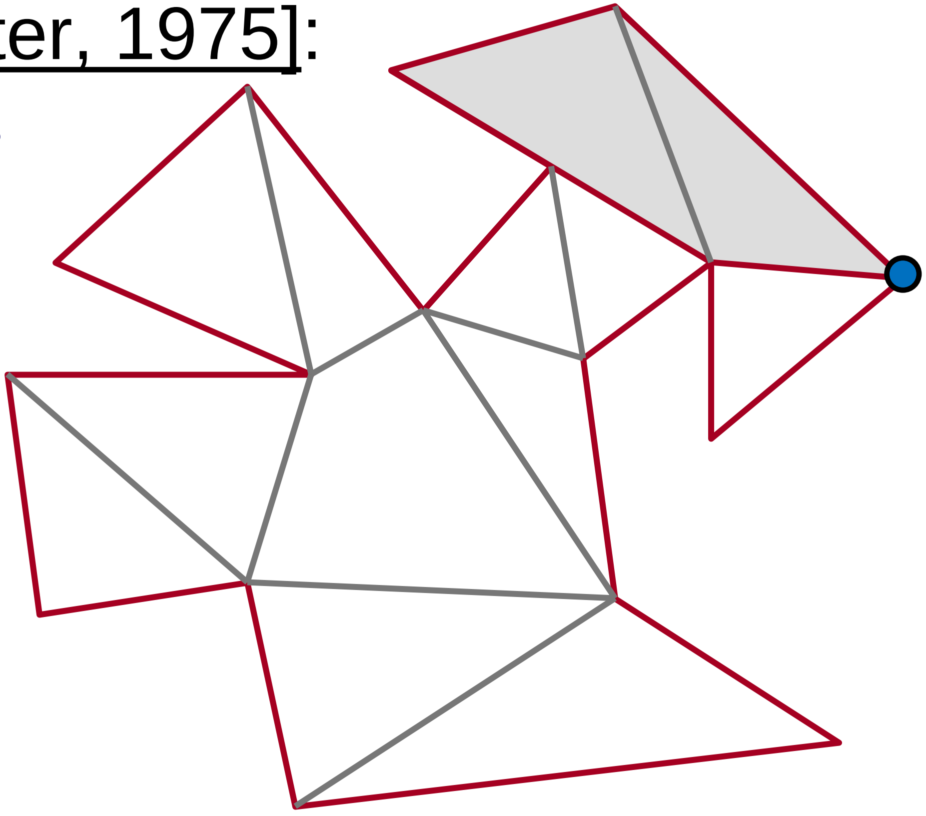
Polygon Scan Conversion

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

<u>Definition</u>:

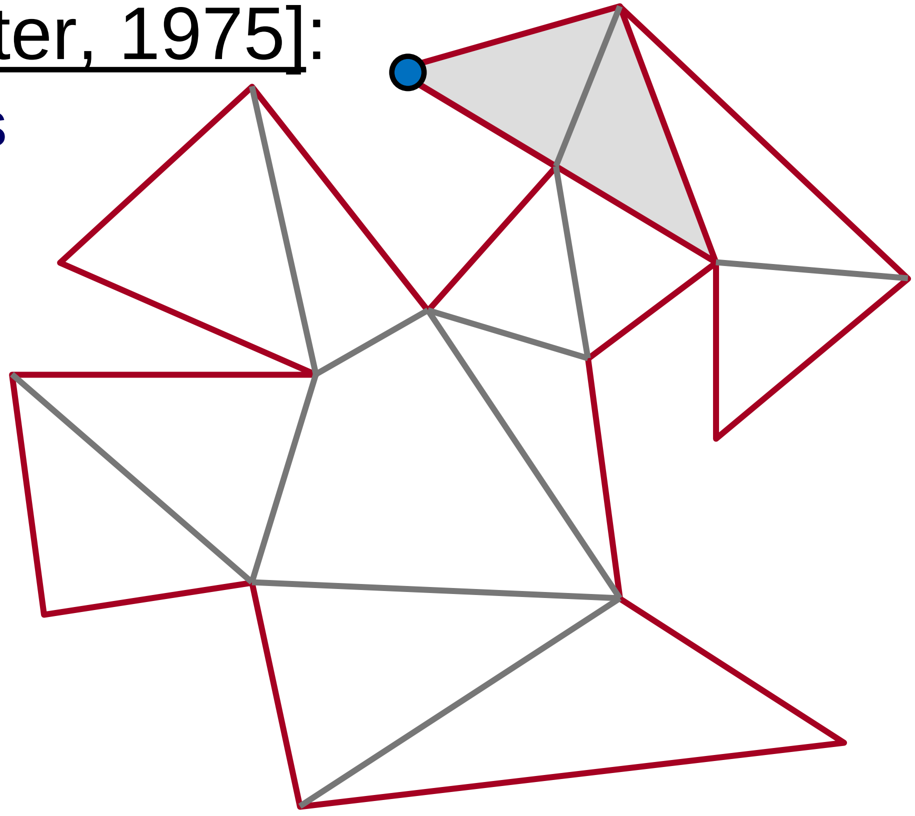
A vertex of a simple polygon is an *ear* if the edge connecting its neighbors is inside the polygon.

<u>Two Ear Theorem [Meister, 1975]</u>:

Every simple polygon has at least two ears.

<u>Algorithm</u>:

1. Clip off an ear
2. Repeat



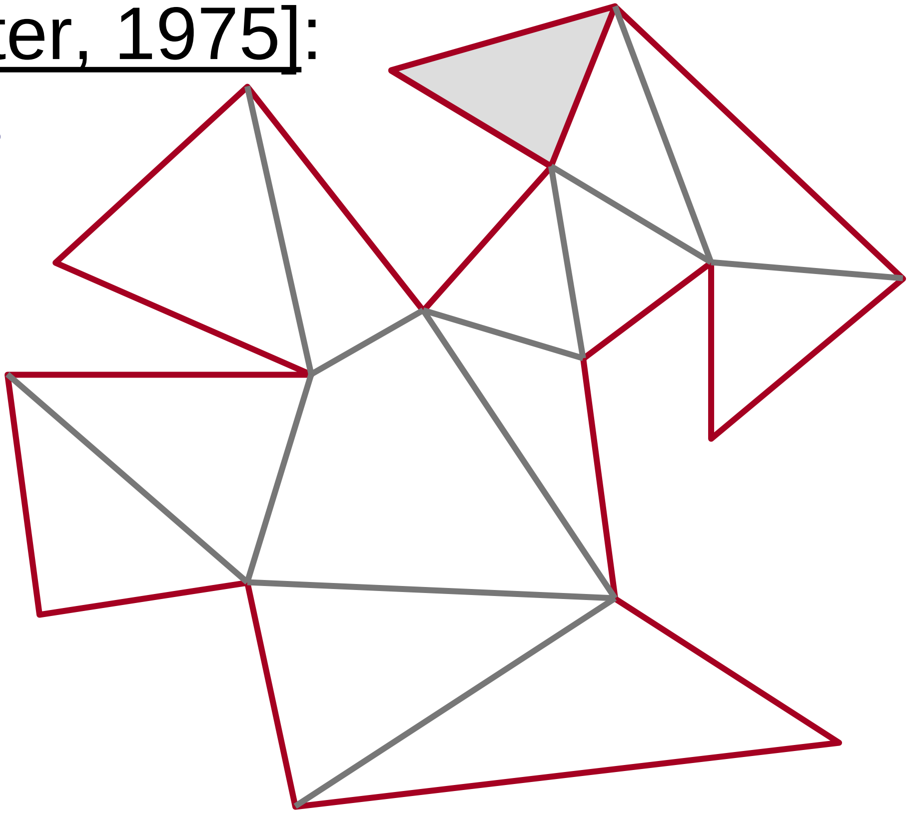
Polygon Scan Conversion

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

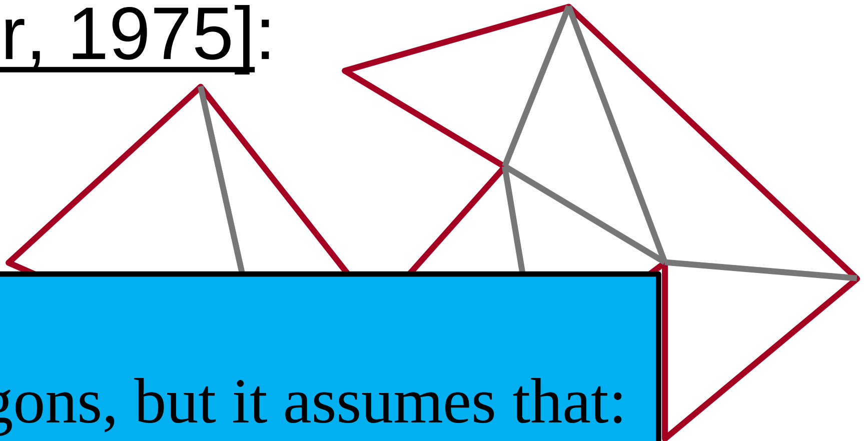
Polygon Scan Conversion

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.



Algor

1.

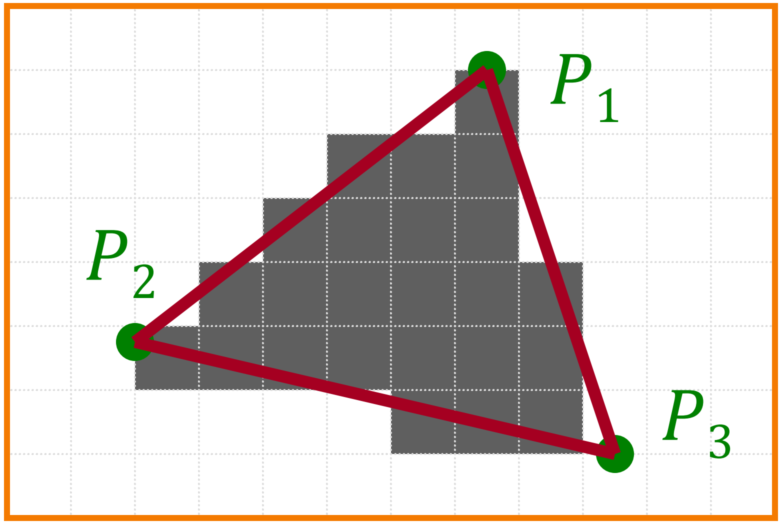
2.

Note:
OpenGL will render polygons, but it assumes that: 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. But 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

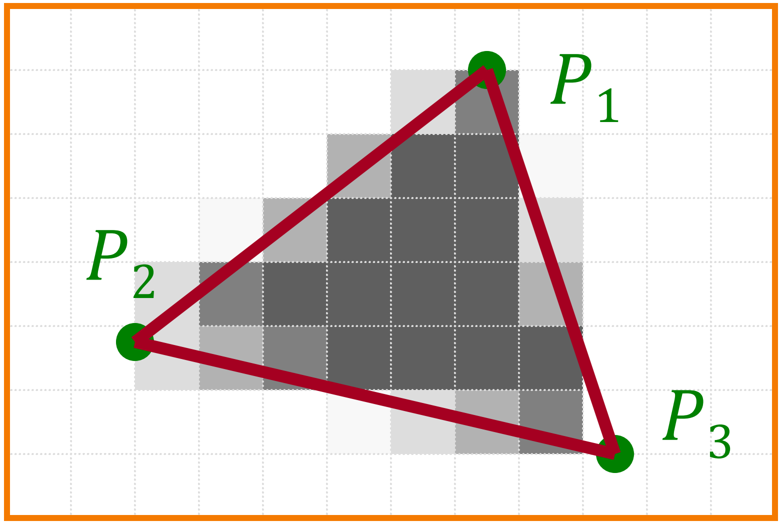
Image courtesy of NVIDIA

Antialiasing Techniques

- Display at higher resolution
  - Corresponds to increasing sampling rate
  - Not always possible (fixed size monitors, fixed refresh rates, etc.)
- Modify pixel intensities
  - Vary pixel intensities along boundaries for antialiasing

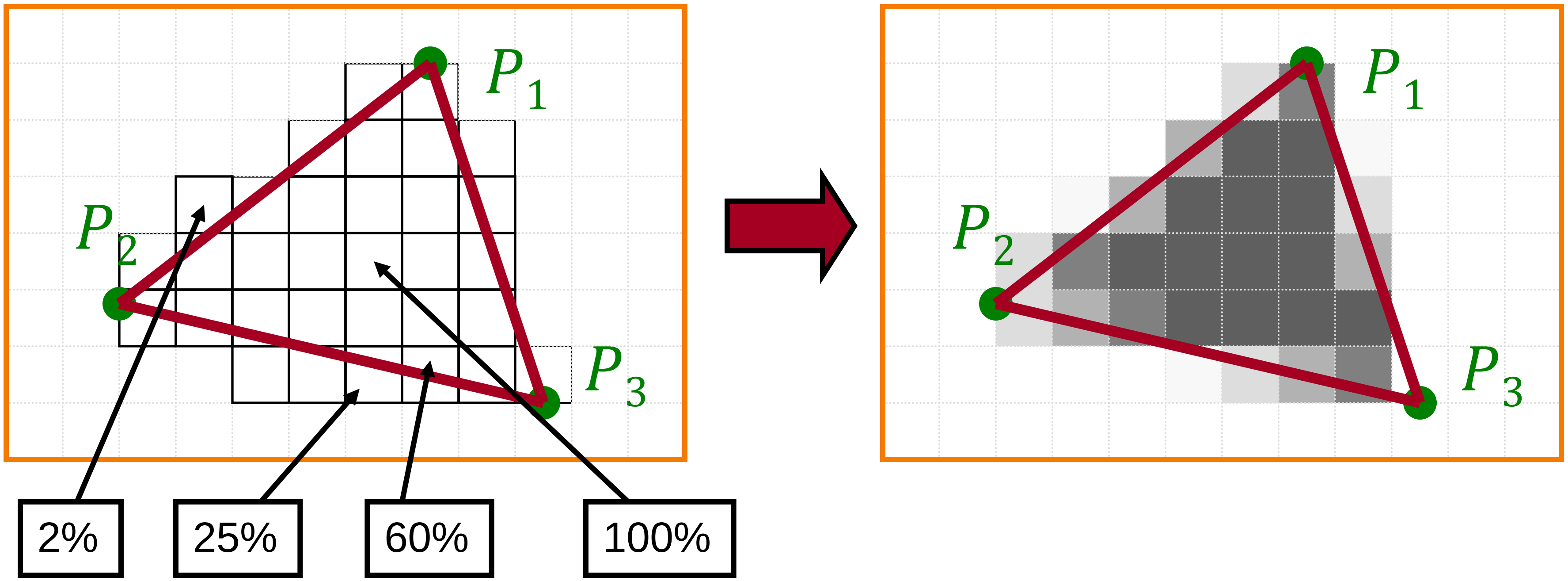
Scan Conversion

- What about pixels on edges?
  - Setting them either “on” or “off” we get aliasing/“jaggies”
  - **Antialias** by varying pixel intensities along boundaries



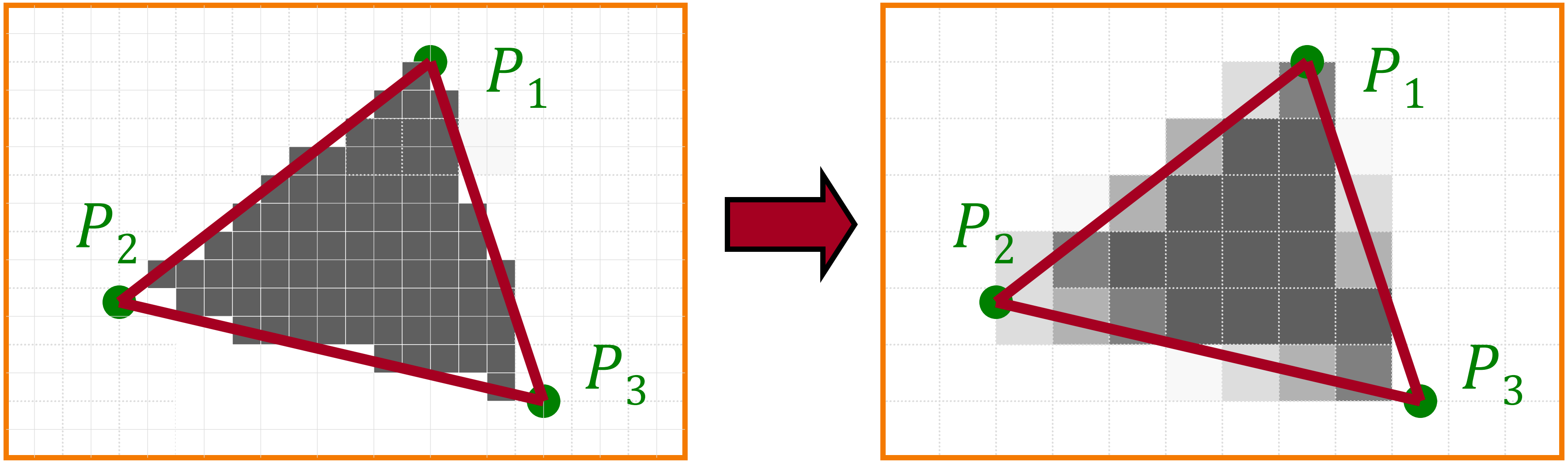
Antialiasing

- Ideally: Area sampling
  - Calculate percent of pixel covered by primitive
  - Multiply this percentage by desired intensity/color



Antialiasing

- 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)

3D Primitives

↓ 3D Modeling Coordinates

Modeling Transformation

↓ 3D World Coordinates

Viewing Transformation

↓ 3D Camera Coordinates

Lighting

↓ 3D Camera Coordinates

Projection Transformation

↓ 2D Window Coordinates

Clipping

↓ 2D Window Coordinates

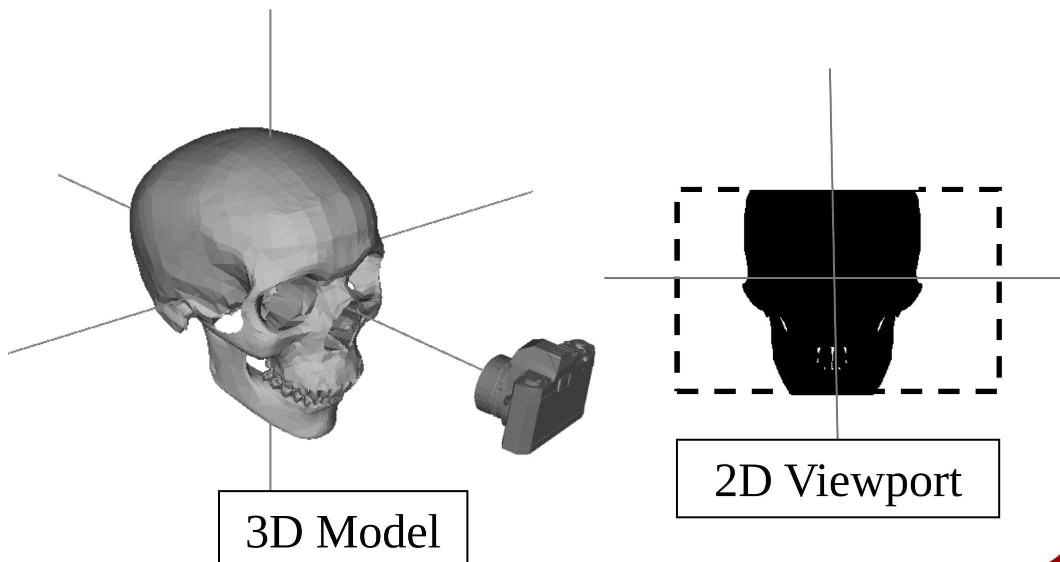
Viewport Transformation

↓ 2D Viewport Coordinates

Scan Conversion

↓ 2D Viewport Coordinates

Image

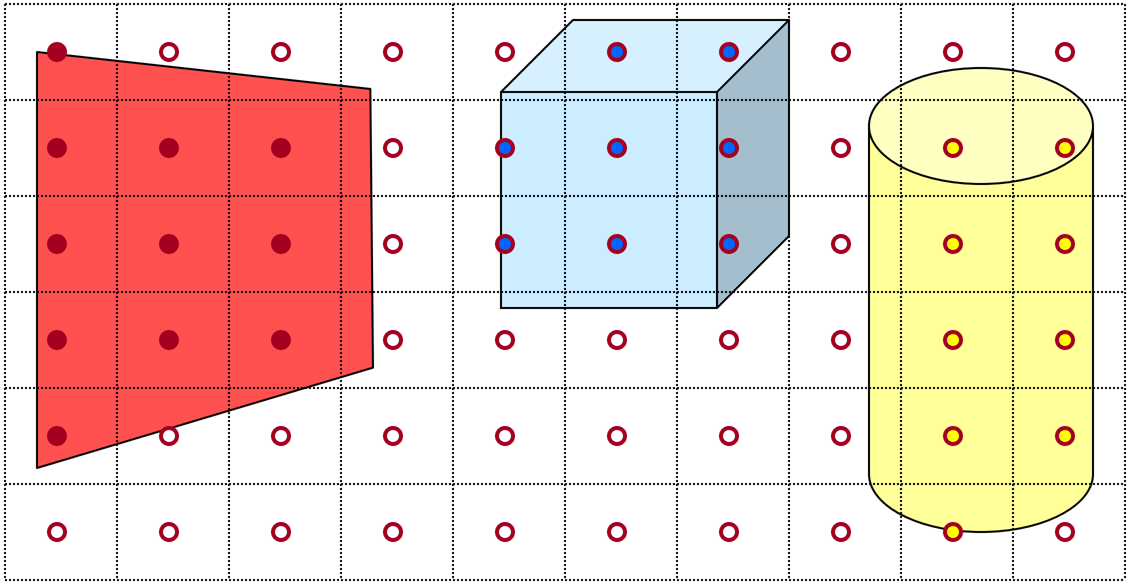


Overview

- Scan conversion
  - Figure out which pixels to fill
- Shading
  - Determine a 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



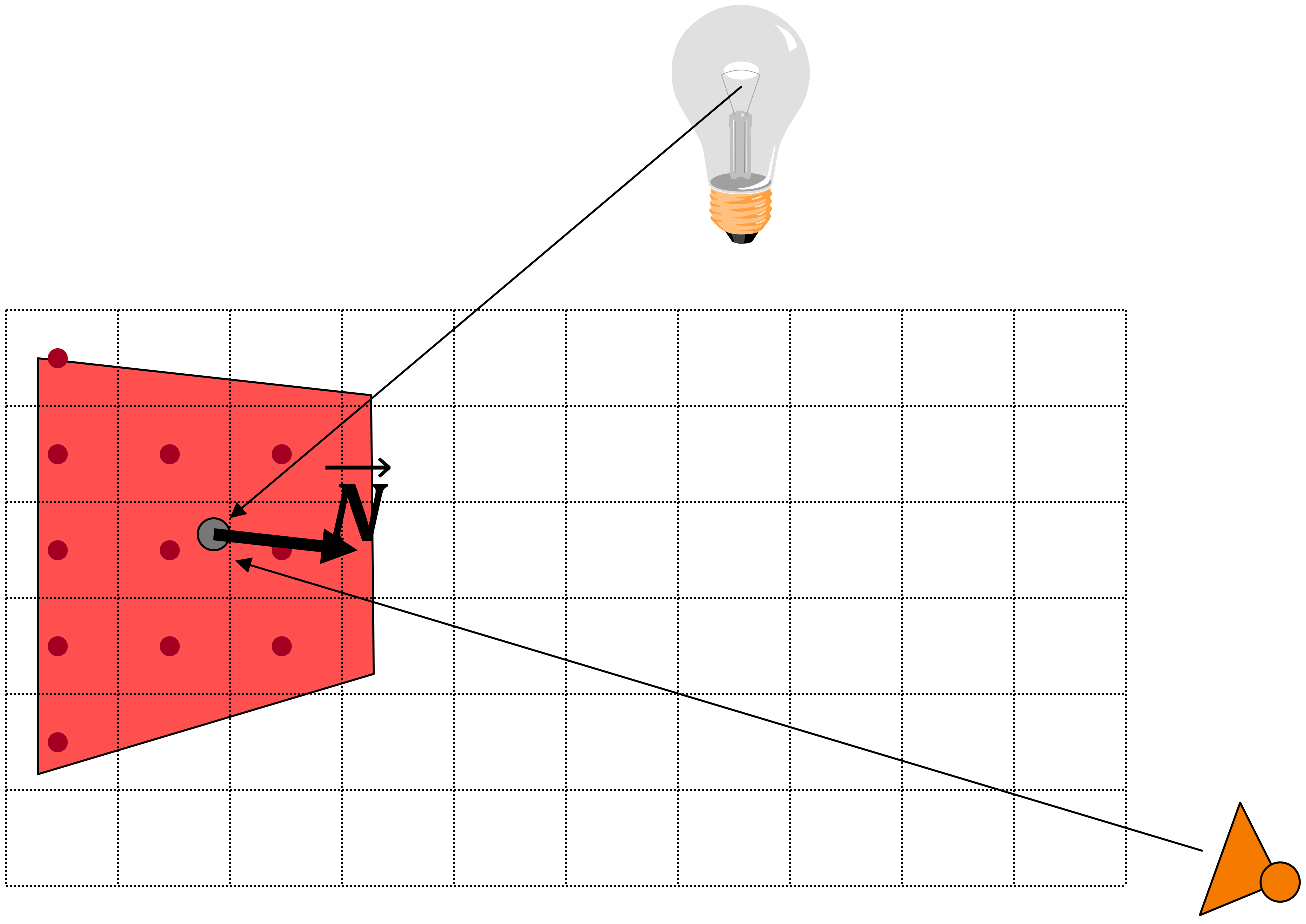
$$I = I_E + K_A I_{AL} + \sum_i \left(K_D \langle \vec{N}, \vec{L}_i \rangle + K_S \langle \vec{V}, \vec{R}_i \rangle^n \right) I_i$$

Polygon Shading Algorithms

- **Flat Shading**
- Gouraud Shading
- Phong Shading

Flat Shading

- One lighting calculation **per polygon**
  - Assign all pixels inside each polygon the same color

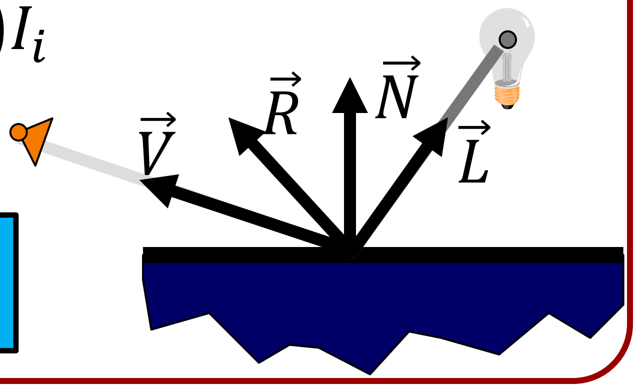


Flat Shading

- 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	+	+	+

$$I = I_E + K_A I_{AL} + \sum_i \left(K_D \langle \vec{N}, \vec{L}_i \rangle + K_S \langle \vec{V}, \vec{R}_i \rangle^n \right) I_i$$



In what follows, assume that the emissive light, I_E, as well as the material properties, K_A, K_D, K_s, and n, are constant

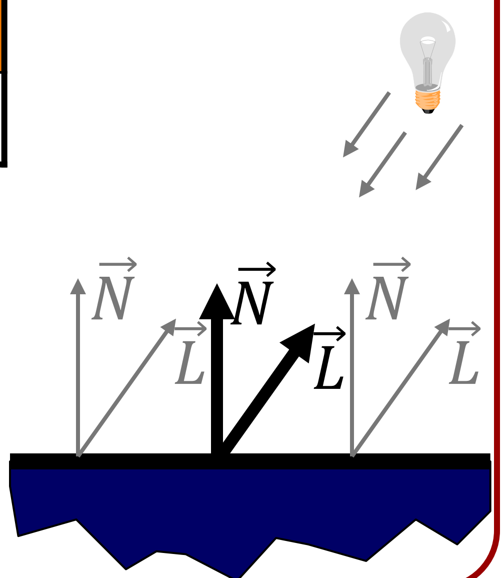
Flat Shading

- Take advantage of spatial coherence
  - Make the lighting equation constant

> - 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

Emissive	-	-	-
Ambient	-	-	-
Diffuse	+	+	-
Specular	+	+	+

$$I = I_E + K_A I_{AL} + \sum_i \left(K_D \boxed{\langle \vec{N}, \vec{L}_i \rangle} + K_S \langle \vec{V}, \vec{R}_i \rangle^n \right) I_i$$



> In what follows, assume that the emissive light, I_E, as well as the material properties, K_A, K_D, K_s, and n, are constant

Flat Shading

- Take advantage of spatial coherence
  - Make the lighting equation constant

- If the normal is constant over the primitive, and
- If the light is directional,

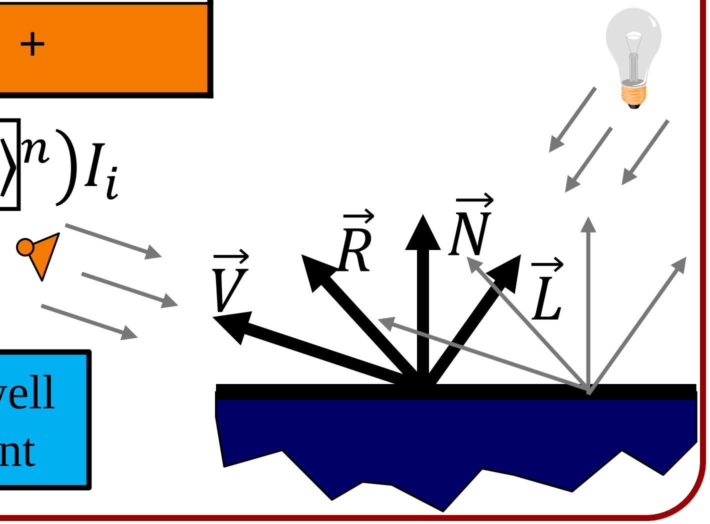
⇒ The diffuse component is the same for all points on the primitive

- 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

Emissive			
Ambient			
Diffuse			
Specular	+	+	+

$$I = I_E + K_A I_{AL} + \sum_i \left(K_D \langle \vec{N}, \vec{L}_i \rangle + K_S \boxed{\langle \vec{V}, \vec{R}_i \rangle}^n \right) I_i$$

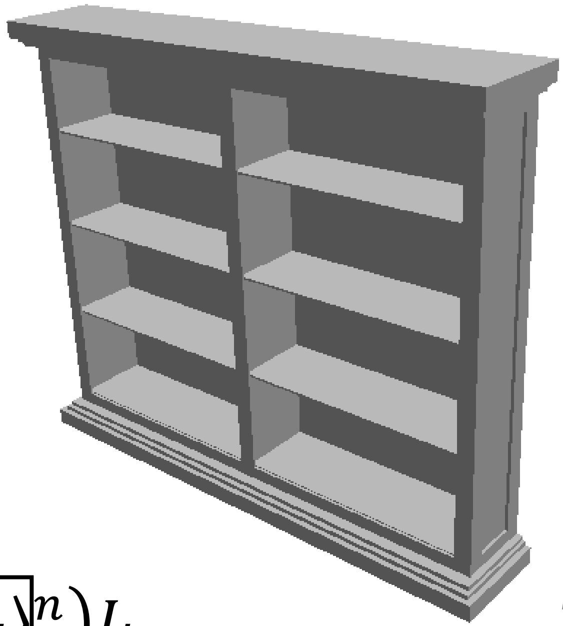


In what follows, assume that the emissive light, I_E, as well as the material properties, K_A, K_D, K_s, and n, are constant

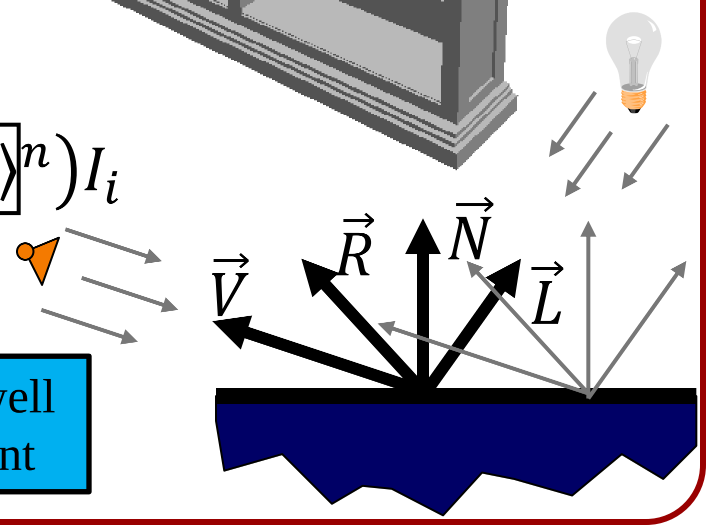
Flat Shading

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

- $\langle \vec{N}, \vec{L}_i \rangle$ constant over surface
- $\langle \vec{V}, \vec{R}_i \rangle$ constant over surface
- I_i constant over surface



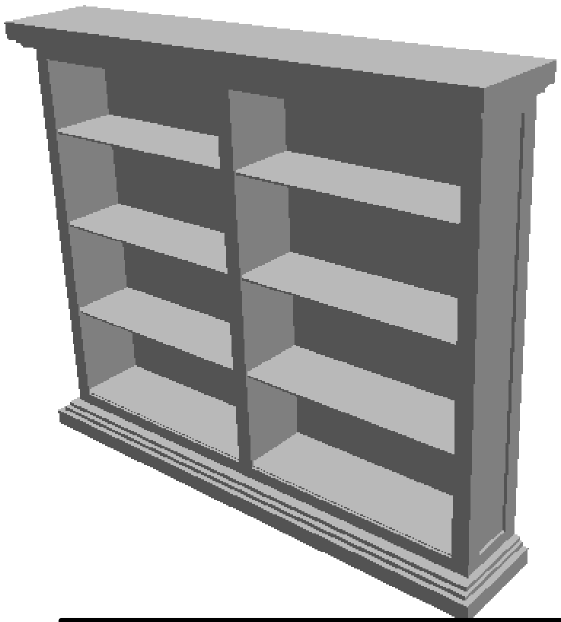
$$I = I_E + K_A I_{AL} + \sum_i \left(K_D \boxed{\langle \vec{N}, \vec{L}_i \rangle} + K_S \boxed{\langle \vec{V}, \vec{R}_i \rangle}^n \right) I_i$$

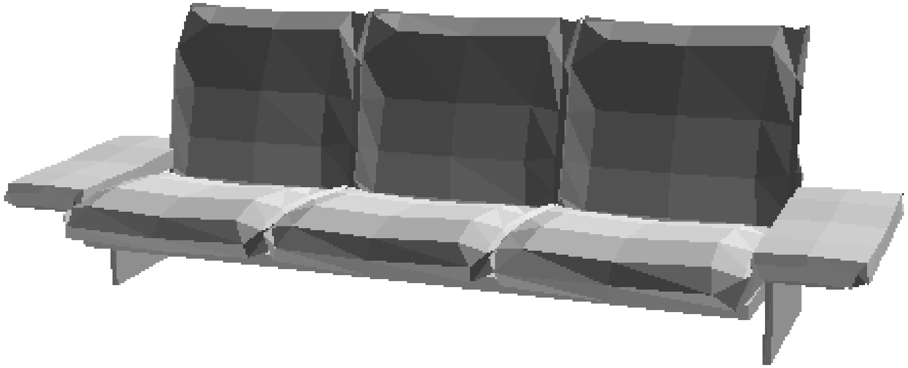


In what follows, assume that the emissive light, I_E, as well as the material properties, K_A, K_D, K_s, and n, are constant

Flat Shading

- Objects look like they are composed of polygons
  - OK for faceted objects
  - Not so good for smooth surfaces





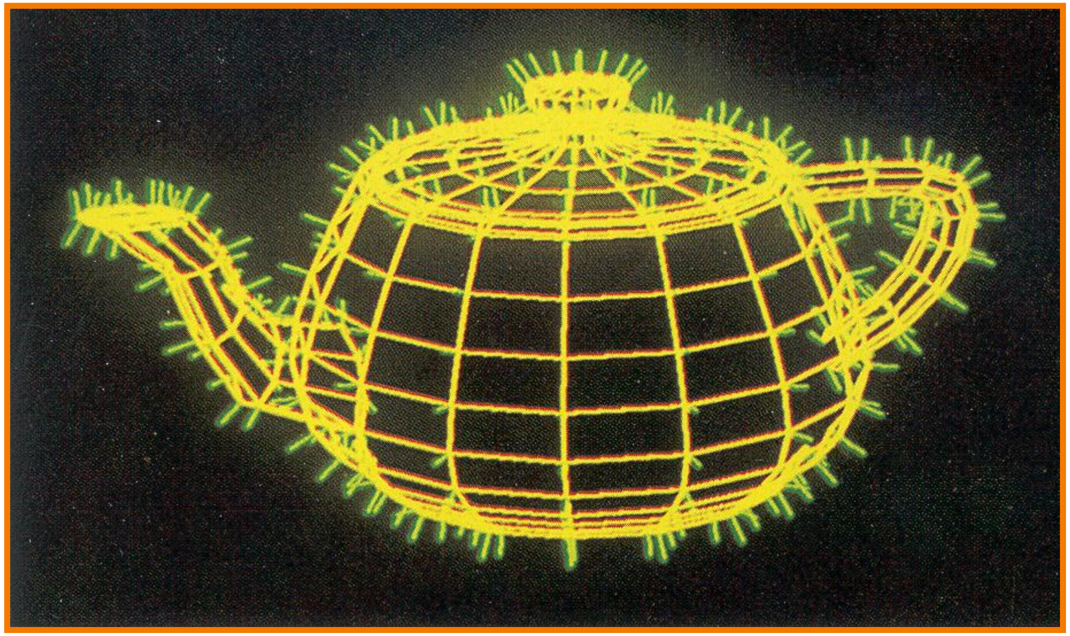
Although this is the "simplest" lighting model, it is tricky to implement this on the graphics card.

Polygon Shading Algorithms

- Flat Shading
- **Gouraud Shading**
- Phong Shading

Gouraud Shading

- Represent a polygonal mesh with a normal at each vertex

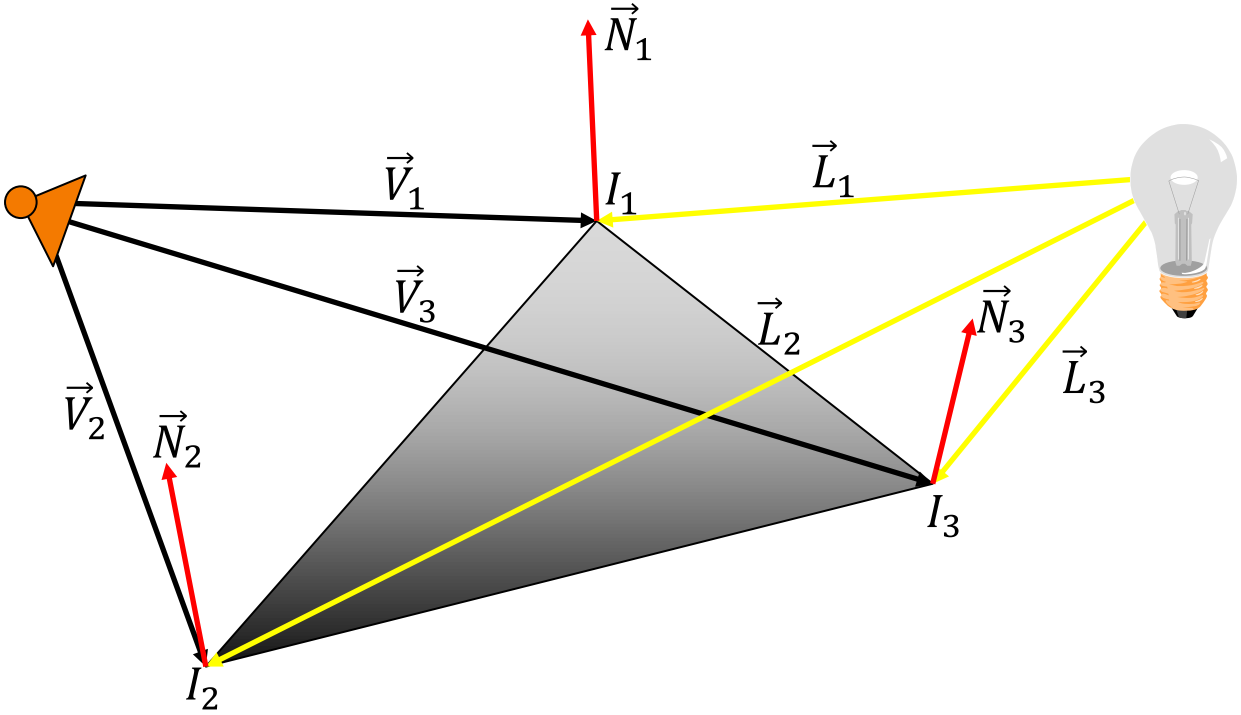


Watt Plate 7

$$I = I_E + K_A I_{AL} + \sum_i \left(K_D \langle \vec{N}, \vec{L}_i \rangle + K_S \langle \vec{V}, \vec{R}_i \rangle^n \right) I_i$$

Gouraud Shading

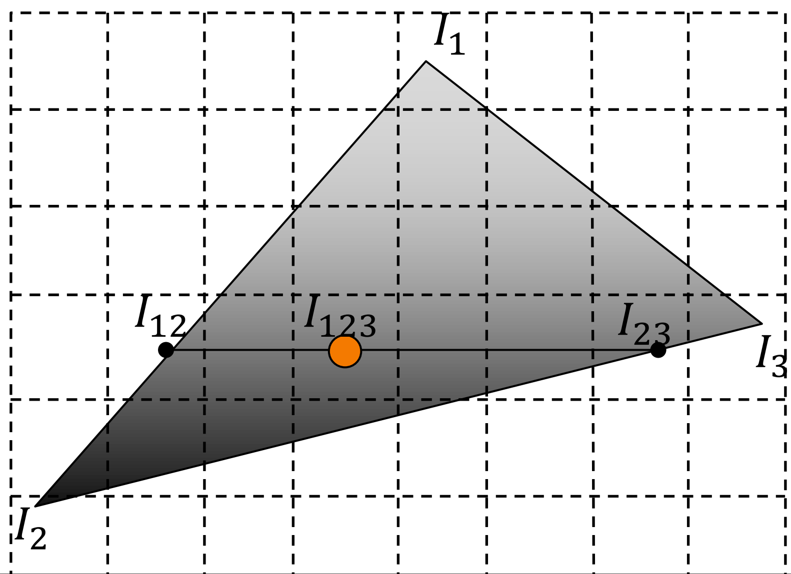
- One lighting calculation **per vertex**
  - Assign pixel colors inside polygon by interpolating colors computed at vertices



Gouraud Shading

- When rasterizing, linearly interpolate colors (first) across and (then) between edges:

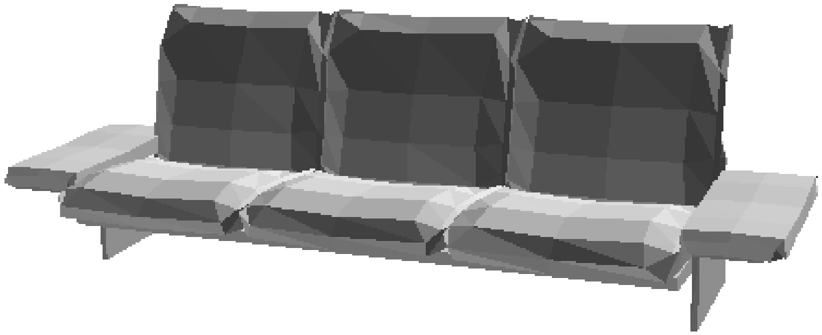
$$(I_1, I_2, I_3) \to (I_{12}, I_{23}) \to I_{123}$$



- I_1, I_2, and I_3 are constant per triangle.
- I_{12} and I_{23} (and I_{13}) are constant per scan-line.
- I_{123} varies across the scan-line

Gouraud Shading

- Produces smoothly shaded polygonal mesh
  - Continuous shading over adjacent polygons



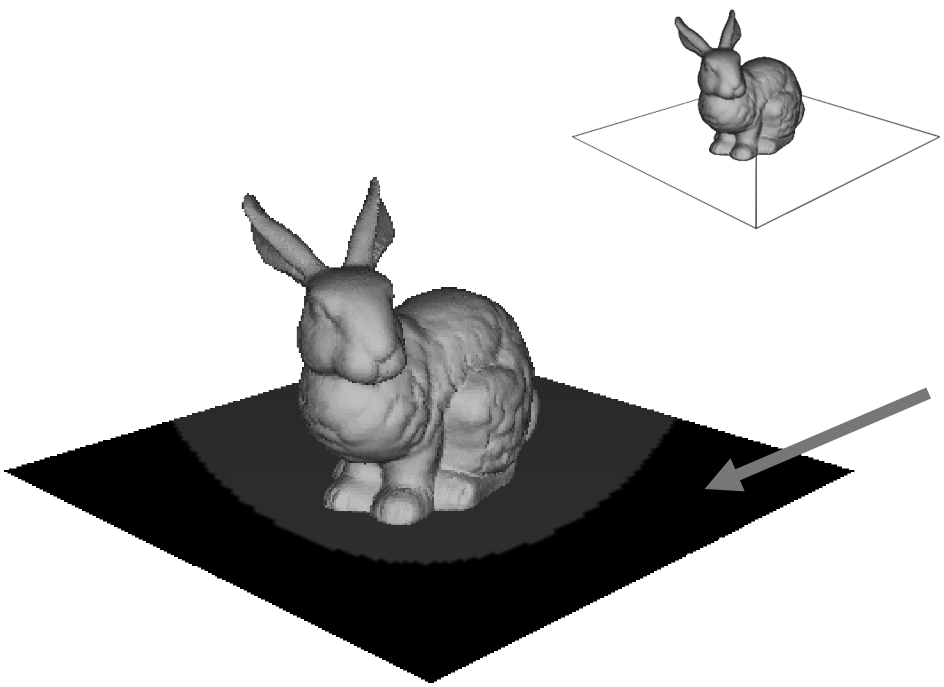
Flat Shading

Gouraud Shading

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

Gouraud Shading

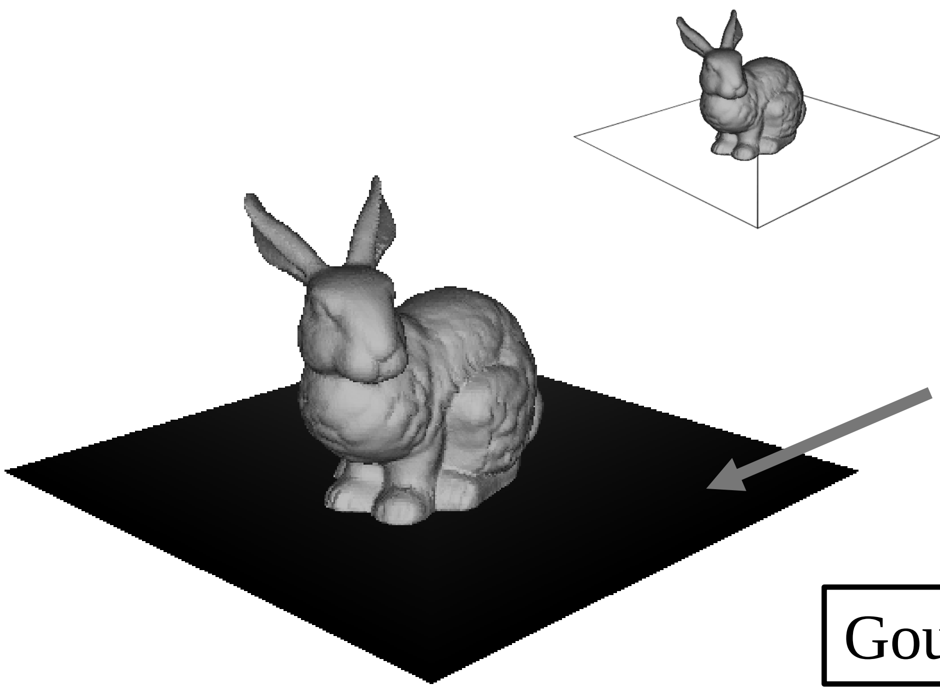
- Produces smoothly shaded polygonal mesh
  - Continuous shading over adjacent polygons



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?

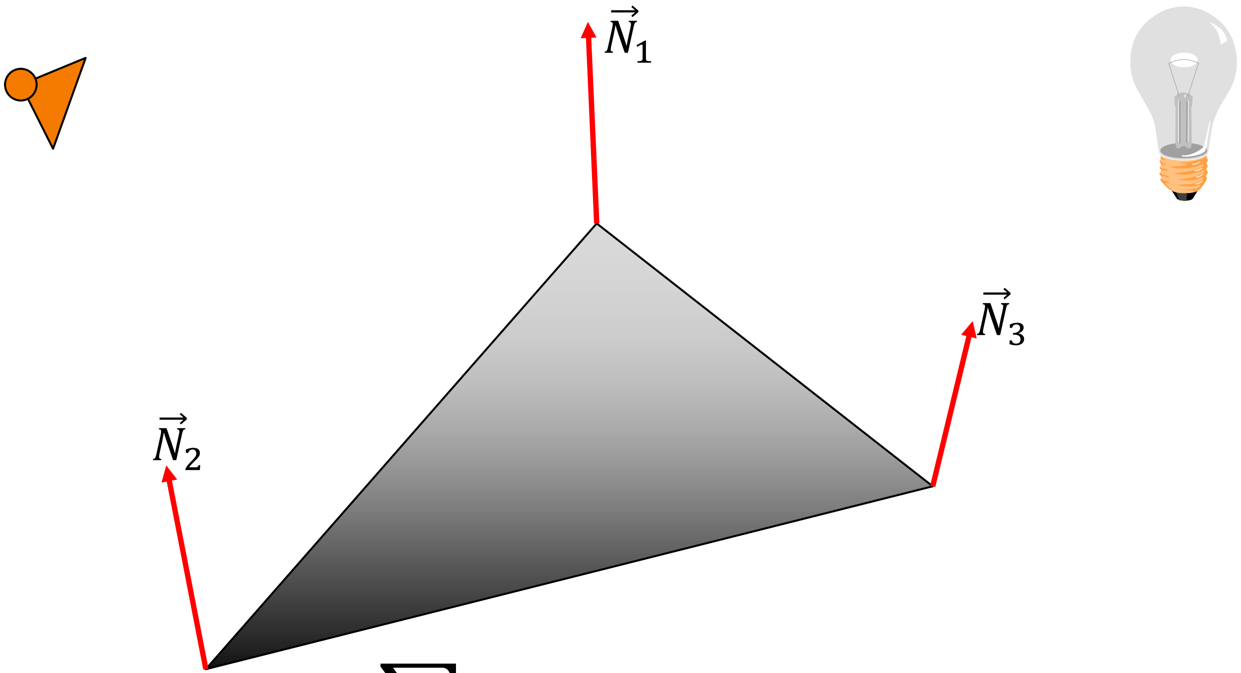
Gouraud Shading Demo

Polygon Shading Algorithms

- Flat Shading
- Gouraud Shading
- **Phong Shading**

Phong Shading

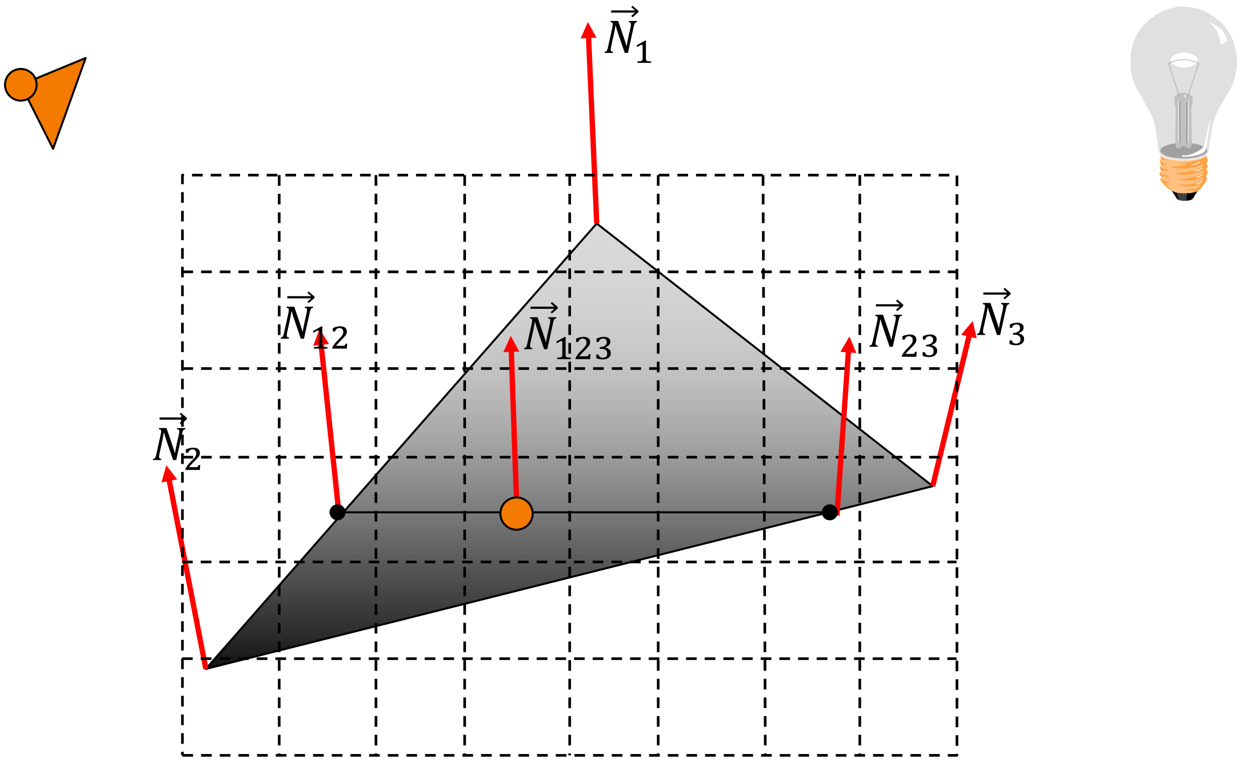
- One lighting calculation **per pixel/fragment**
  - Approximate surface normals for points inside polygons by linear interpolation of normals from vertices



$$I = I_E + K_A I_{AL} + \sum_i \left(K_D \langle \vec{N}, \vec{L}_i \rangle + K_S \langle \vec{V}, \vec{R}_i \rangle^n \right) I_i$$

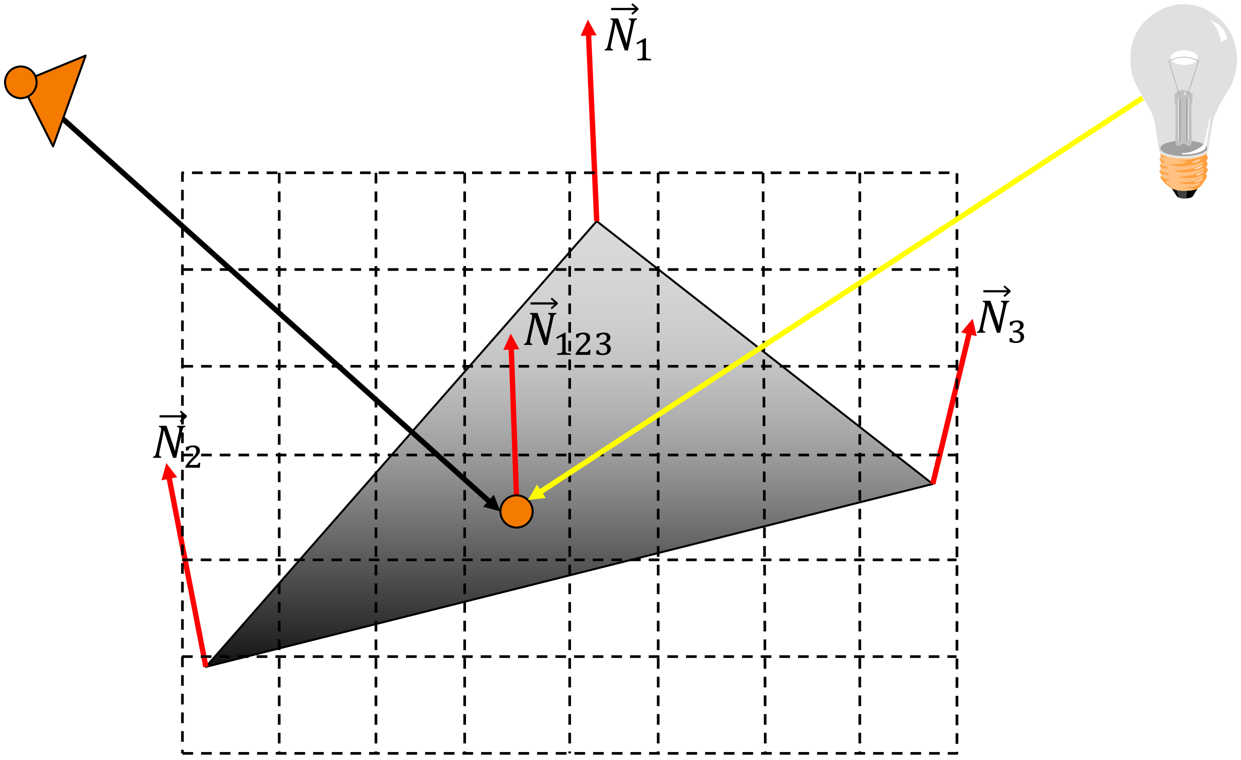
Phong Shading

- When rasterizing, interpolate vertex normals (first) down and (then) across scan lines



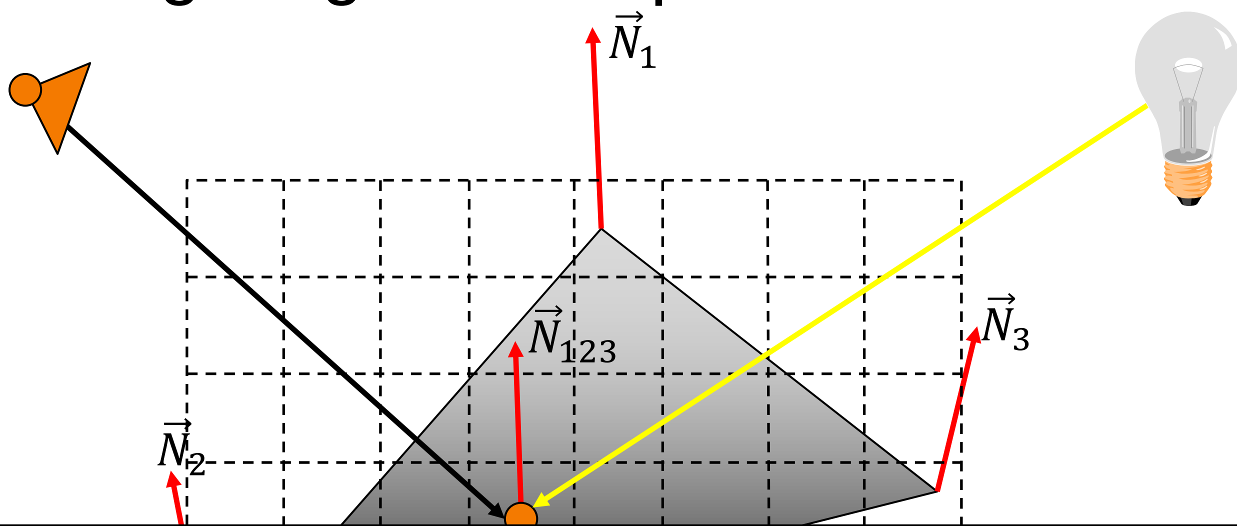
Phong Shading

- When rasterizing, interpolate vertex normals (first) down and (then) across scan lines
- Compute lighting at each pixel



Phong Shading

- When rasterizing, interpolate vertex normals (first) down and (then) across scan lines
- Compute lighting at each pixel

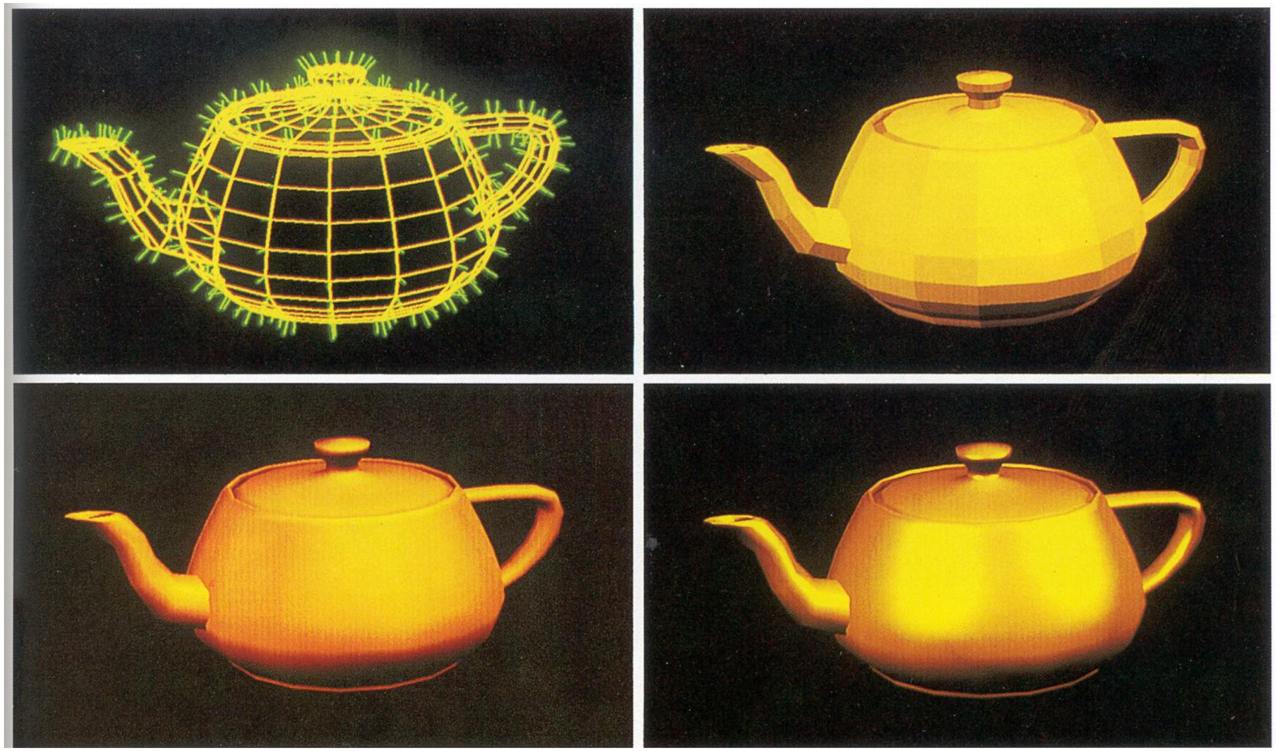


This was not supported in early generation graphic cards but can now be implemented in the <u>fragment shader</u> of the GPU.

Phong Shading Demo

Polygon Shading Algorithms

Wireframe Flat



Gouraud Phong

Watt Plate 7

3D Rendering Pipeline (for direct illumination)

3D Primitives

↓ 3D Modeling Coordinates

Modeling Transformation

↓ 3D World Coordinates

Viewing Transformation

↓ 3D Camera Coordinates

Lighting

↓ 3D Camera Coordinates

Projection Transformation

↓ 2D Window Coordinates

Clipping

↓ 2D Window Coordinates

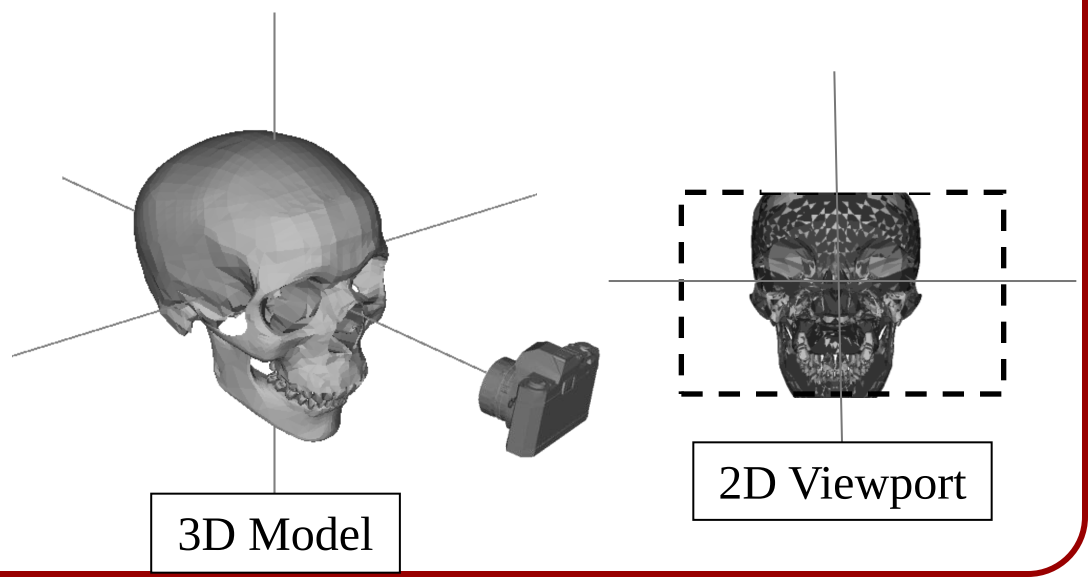
Viewport Transformation

↓ 2D Viewport Coordinates

Scan Conversion

↓ 2D Viewport Coordinates

Image



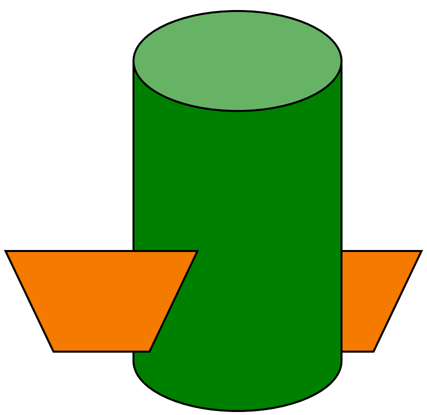
Overview

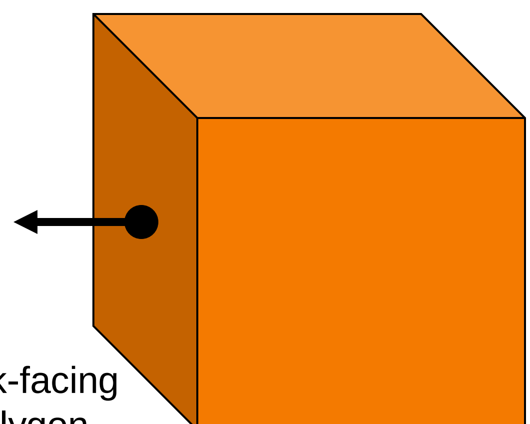
- Scan conversion
  - Figure out which pixels to fill
- Shading
  - Determine a color for each filled pixel
- Depth test
  - Determine when the color of a pixel comes from the front-most primitive

Motivation

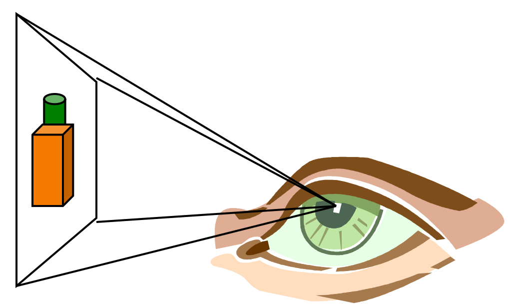
In general, we don't want to draw surfaces that are not visible to the viewer:

- Surfaces may be back-facing
- Surfaces may be covered in 3D
- Surfaces may be covered in the image plane

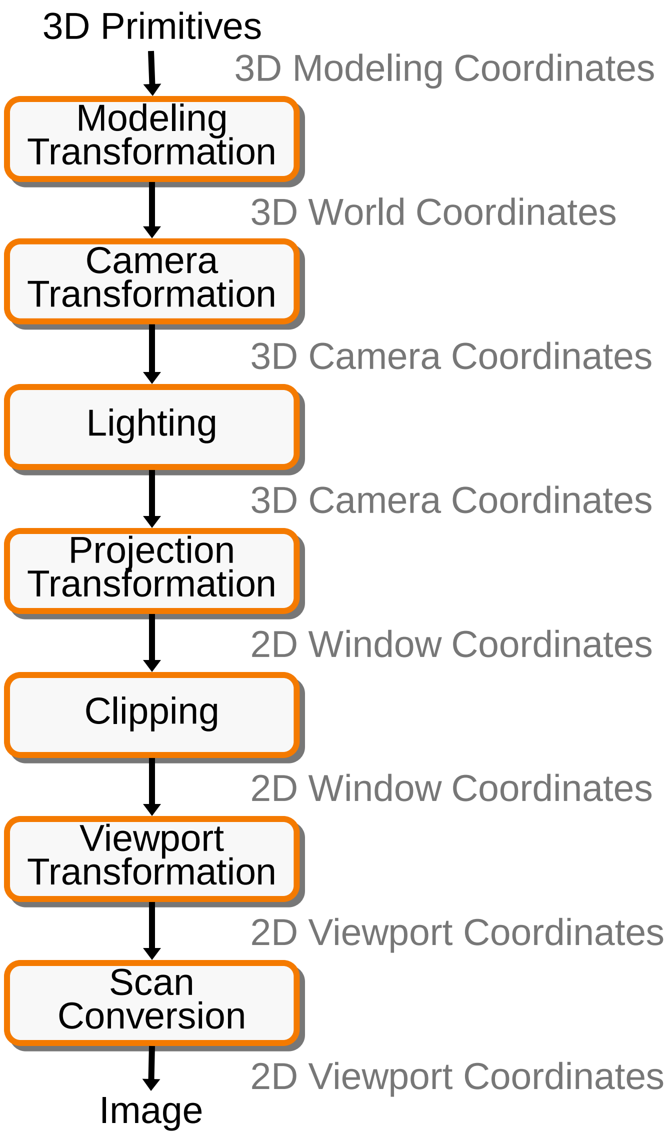




back-facing polygon

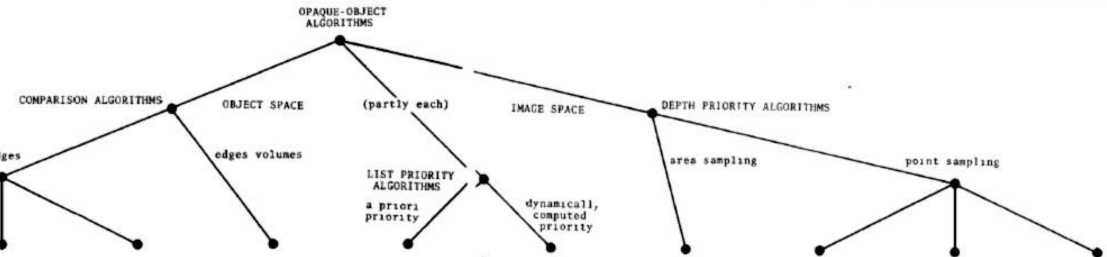


3D Rendering Pipeline



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

Visibility algorithms



	APPEL 1967	GALIMBERTI, et al 1969	LOUTREL 1967	ROBERTS 1963	SCHUMACKER, et al 1969	NEWELL, et al 1972	WARNOCK 1968	WATKINS 1970	ROMNEY, et al 1967	BOUKNIGHT 1969
RESTRICTIONS	TP, NP	TP, NP	TP, NP	TP, CC, CF, NP	CF, NP, LS (TP)	None	(TR) None	None	TR, CF, NP	
COHERENCE	Promote visibility of a vertex to all edges at vertex	Promote visibility of a vertex to all edges at vertex	Promote visibility of a vertex to all edges at vertex		Frame coherence in depth No X coherence used	None used	Area coherence	Scanline X coherence	Scanline Depth Coherence	Scanline X Coherence
SORTING (1) What, what property (2) Method (3) Type (4) Result structure (5) Number per frame, number of objects	Back Edge Cull 1) Edges separating back-facing planes 2) Dot product with normals & topology 3) Cull 4) List of edges, E_s 5) 1, E_t	Back Edge Cull 1) Edges separating back-facing planes 2) Dot product with normals & topology 3) Cull 4) List of edges, E_s 5) 1, E_t	Back Edge Cull 1) Edges separating back-facing planes 2) Dot product with normals & topology 3) Cull 4) List of edges, E_s 5) 1, E_t	Back Edge Cull 1) Edges separating back-facing planes 2) Dot product with normals & topology 3) Cull 4) List of edges, E_s 5) 1, E_t	Intra-Cluster Priority 1) Faces - visibility 2) Dot product with normals 3) Exhaustive search 4) Ordered table 5) 0, (off-line)	Z Sort 1) Faces, max Z 2) Comparison of max points 3) n log m 4) Ordered table 5) 1, F_r	Z Sort (Opt.) 1) Faces, max Z 2) Comparison of max points 3) n log m 4) Ordered table 5) 1, F_r	Y Sort 1) Edges, min Y 2) Comparison 3) Bucket 4) Table of Lists 5) 1, E_r	Y Sort 1) Polygons, Y endpoints 2) Comparison 3) 2 bucket 4) Table of lists 5) 1, F_r	Y Sort 1) Edges, Min Y 2) Comparison 3) Bucket 4) Table of lists 5) 1, E_r
	Contour Edge Cull 1) Edges separating front & back faces 2) Dot product with normals & topology 3) Cull 4) List, E_c 5) 1, E_t	(Omitted)	(Omitted)	Clipping Cull 1) Intersect edge with visible volume 2) 3) Cull 4) E_s 5) 1^s, E_s	Inter-Cluster Priority 1) Clusters 2) Dot product with separating planes 3) Prefix scan binary tree 4) ordered table 5) 1, C_t	Newell Special 1) Faces, pairwise visibility 2) Depth, bounding boxes, separation 3) Bubble, splitting 4) Ordered table 5) 1, F_r+split faces	Warnock Special 1) Faces with window 2) Depth, mini-max in X and Y, sum of angles 3) Radix 4 subdivision with overlap 4) Stacks of unordered tables 5) L_v, F_r/factor 1	X Merge 1) Edges, X value 2) Comparison 3) Merge (ordred) 4) 2-way linked list 5) E_r, S_ℓ	X Sort 1) Edges, X value 2) Comparison 3) 2 bucket 4) Table of lists 5) n, S_ℓ	X Merge 1) Edges, X value 2) Comparison 3) Merge (ordered) 4) Linked list 5) E_r, $2S_\ell$ (edges)
(merge) Number of new entries per frame, length of list	Initial Visibility 1) Ray to vertex against all faces 2) Depth, Surroundedness 3) Exhaustive search 4) Quantitative visibility of vertex 5) # Objects, F_r	Initial Visibility 1) Ray to vertex against all faces 2) Depth, surroundedness 3) Exhaustive search 4) Quantitative visibility of vertex 5) #objects, F_r	Initial Visibility 1) Ray to vertex against all faces 2) Betweenness, surroundedness 3) Exhaustive search 4) Quantitative visibility of vertex 5) #objects, F_r	Edge/Volume Test 1) Edges, visibility relative to volumes 2) Linear Programming 3) Mini-max sort 4) Answer 5) E_s+ split edges, #objects	Back-Face Cull 1) Faces 2) Dot product with face normal 3) Cull 4) Smaller ordered table 5) 1, F_t	Y Sort 1) Face segment by Y range 2) Y intercept 3) Bucket 4) None 5) F_r + split faces H_f^r	Depth Search 1) Surrounder faces 2) 4-corner compare 3) Exhaustive 4) Answer/failure 5) L_v, F_r/factor 2	X Sort 1) Segments, X left 2) Comparison 3) Bubble 4) 2-way linked list 5) n, S_ℓ	X Priority Search 1) Edges, X value 2) Comparison 3) Priority search 4) Active segment list 5) n, m	X Sort 1) Edges, X value 2) Comparison 3) Bubble 4) 1-way linked list 5) N, $2S_\ell$ (edges)
(search) Number of searches, length of list	Edge Intersection 1) Intersect one E_s with all E_c 2) Penetration with sweep triangle 3) Cull (unordered) 4) Intersection list 5) E_s, E_c	Edge Intersection 1) Intersect one E_s with all E_c 2) Intersect in picture plane, depth 3) Cull (unordered) 4) Intersection list 5) E_s, E_s - 1	Edge Intersection 1) Intersect one E_s with all E_c 2) Intersect in picture plane, depth 3) Cull (unordered) 4) Intersection list 5) E_s, E_s - 1		Y Cull 1) Faces by Y extent 2) Mini-max on X intercepts 3) Cull (unordered) 4) X intercepts of relevant segments 5) n, E_s	X Merge 1) Segments, X intercept 2) Comparison 3) Ordered merge 4) Ordered list 5) S_r, S_v/2	TV Sort (Opt.) Sort windows into scan-line order if needed	Span Cull 1) Segments, overlap with sample span 2) Double comparison 3) Cull ordered list 4) Active list 5) n*S_v * f (>1), S_ℓ	Z Search 1) Segments, depth 2) Linear equations and comparison 3) Search (unordered) 4) Visible segment 5) n*$2S_\ell$, D_c (Omitted if X priorities same as last time)	Z Search 1) Segments, depth 2) Linear equations and comparison 3) Search of unordered active list 4) Visible segment 5) n*$2S_\ell$, D_c
	Sort Along Edge 1) Intersections on edge, ordering 2) Comparison 3) Bubble 4) Answer 5) E_s, X_v/E_s (Omit if well hidden)	Sort Along Edge 1) Intersections on edge, ordering 2) 3) 4) Answer 5) E_s, X_v/E_s (must be done)	Sort Along Edge 1) Intersections on edge, ordering 2) 3) 4) Answer 5) E_s, X_v/E_s (Omit if well hidden)		X Sort 1) Segments 2) Counters 3) Hardware 4) Segments at this X 5) nm, S_ℓ			Z Search 1) Segments, Z 2) Depth by logarithmic search 3) Search (unordered) 4) Visible segment 5) n*S_v*f(>1), D_c		
					Priority Search 1) Segments, priority 2) Logic network 3) Logic network 4) Visible segment 5) nm, S_ℓ					

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

[Sutherland '74]

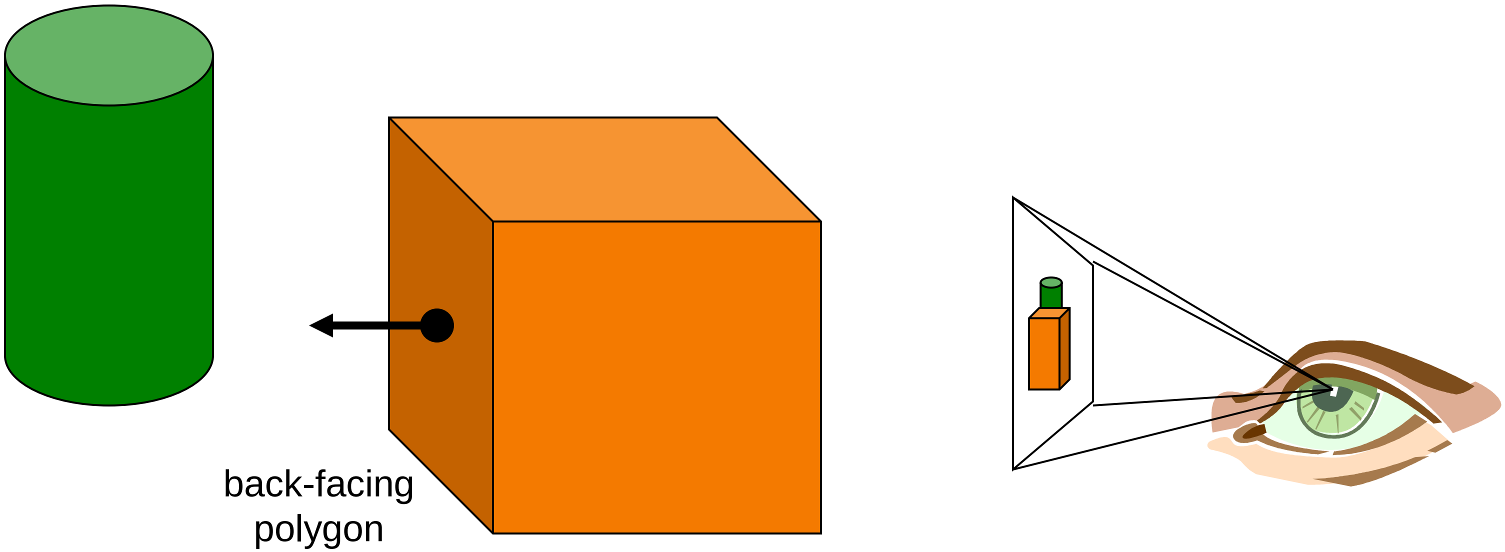
Hidden Surface Removal (HSR)

- Algorithms for HSR
  - Back-face detection
  - View-frustrum culling
  - z-buffer

Back-face detection

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

A: Dot product of the normal and view directions.

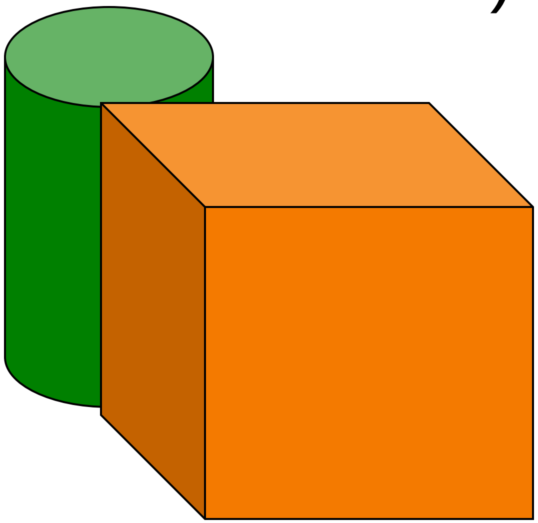


If $\langle \vec{V}, \vec{N} \rangle > 0$, then polygon is back-facing

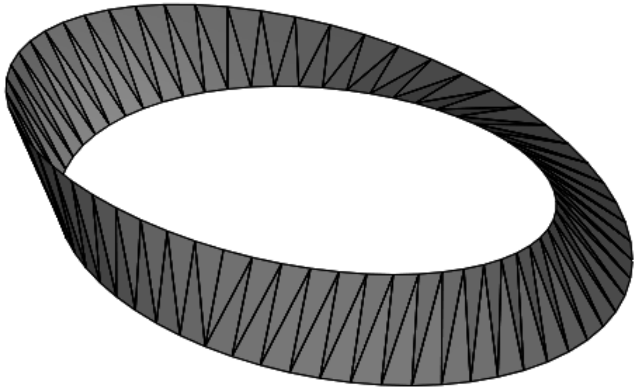
Back-face detection

This method:

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



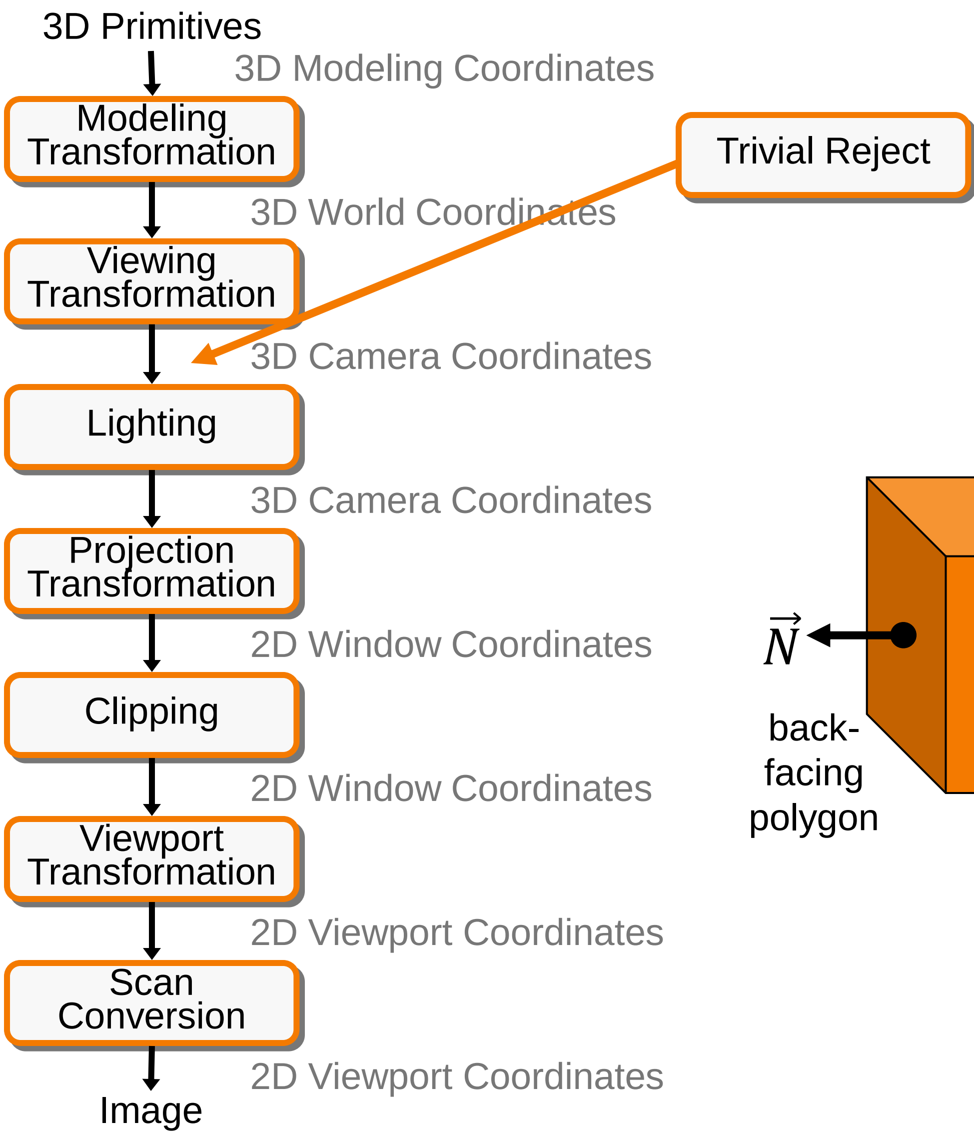
Overlapping
Objects



Non-water-tight
Non-orientable

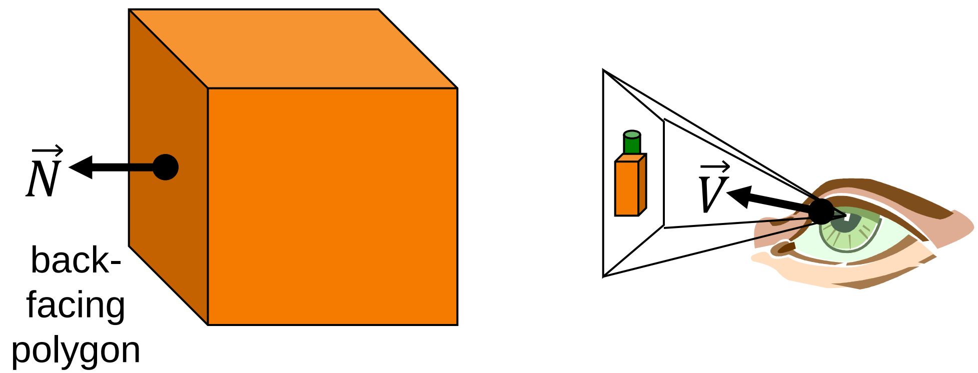
In general, back-face expected to remove ≈ half of polygon surfaces from removal further visibility tests

3D Rendering Pipeline



A polygon is back-facing if

$$\langle \vec{V}, \vec{N} \rangle > 0$$



3D Rendering Pipeline

3D Primitives

↓ 3D Modeling Coordinates

Modeling Transformation

↓ 3D World Coordinates

Camera Transformation

↓ 3D Camera Coordinates

Lighting

↓ 3D Camera Coordinates

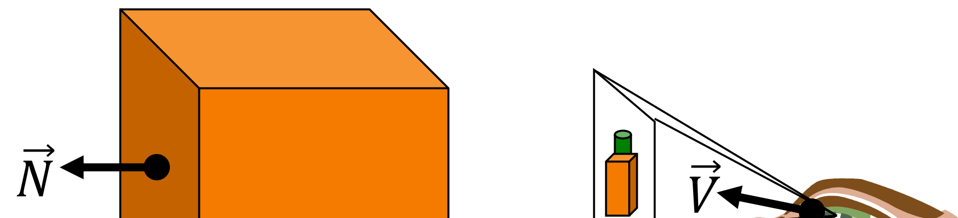
Projection Transformation

↓ 2D Window Coordinates

Trivial Reject

A polygon is back-facing if

$$\langle \vec{V}, \vec{N} \rangle > 0$$

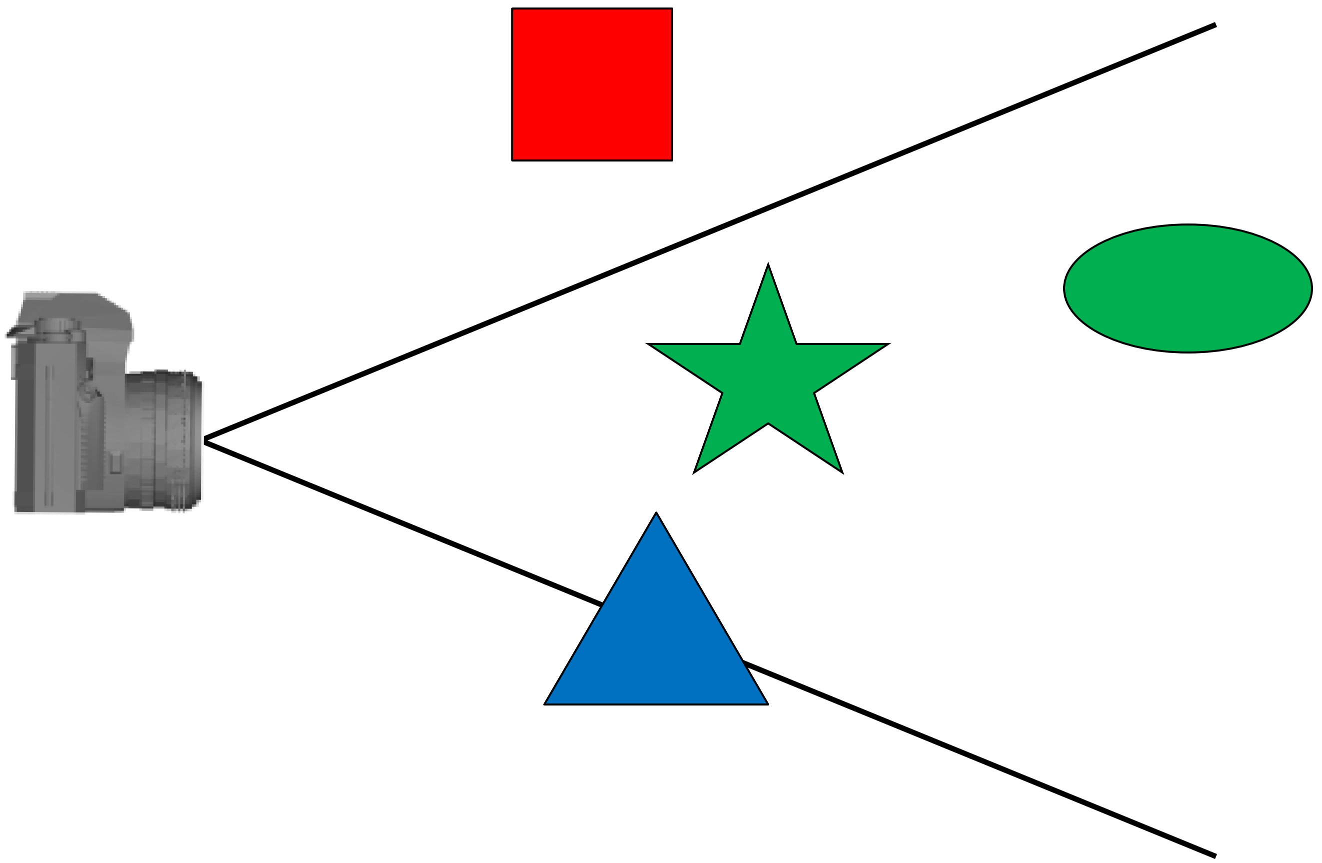


<u>Note</u>: When your graphics card does this, it does <u>not</u> use the rendering normals you provide at the vertices for lighting.
It uses the geometric normal – the cross-product of the triangle edges – so make sure that the ordering of the vertices is consistent.

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

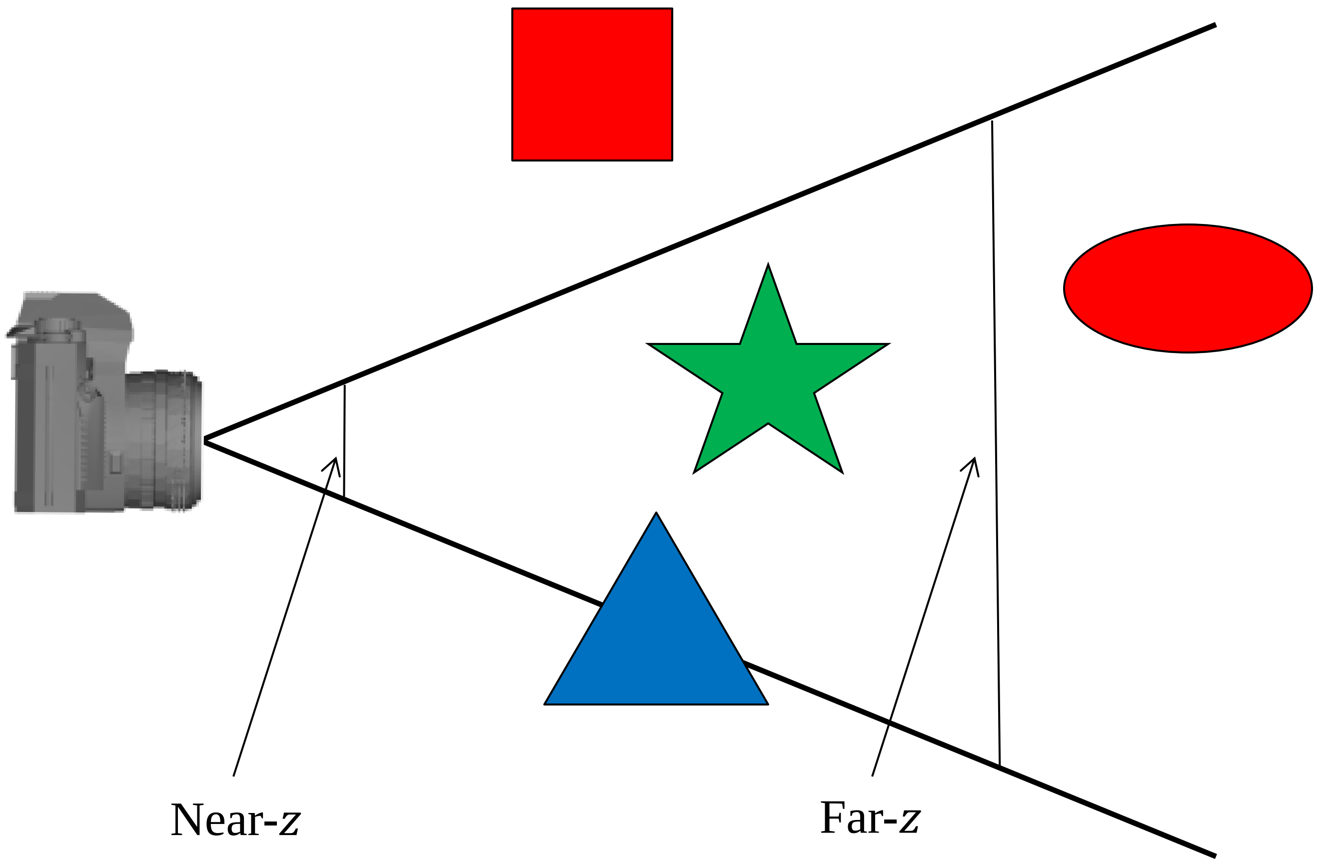
View-frustrum culling

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

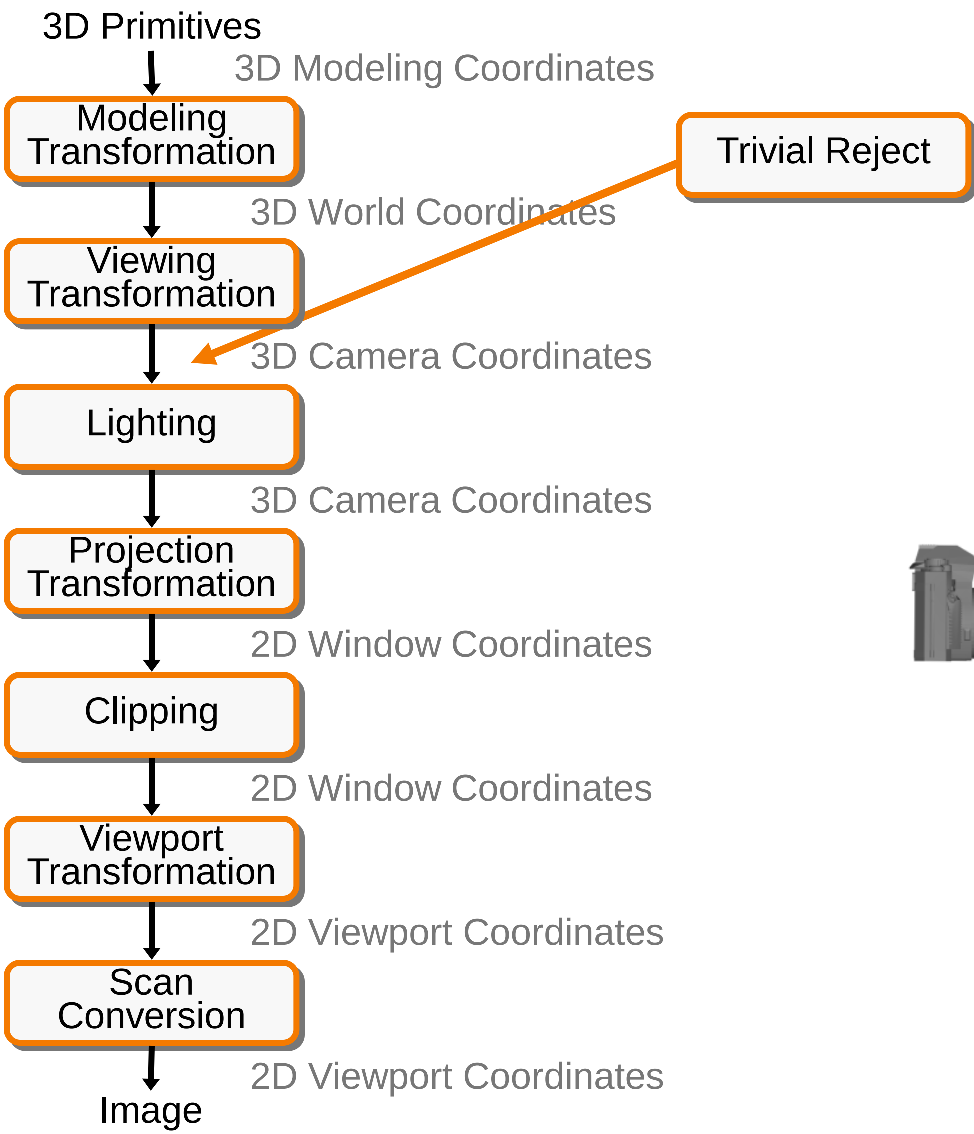


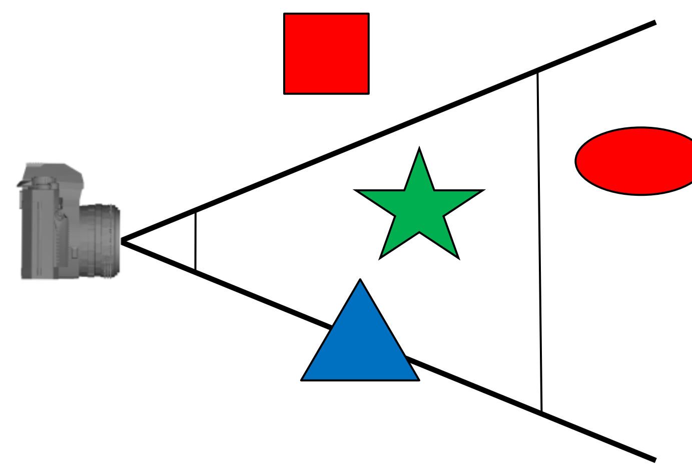
View-frustrum culling

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



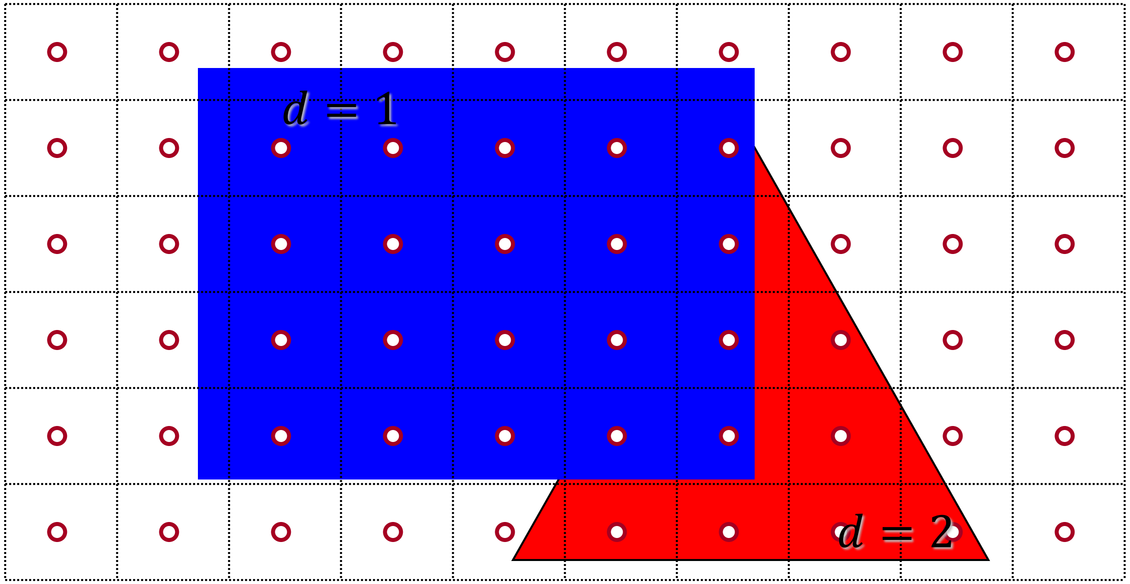
View-frustrum culling





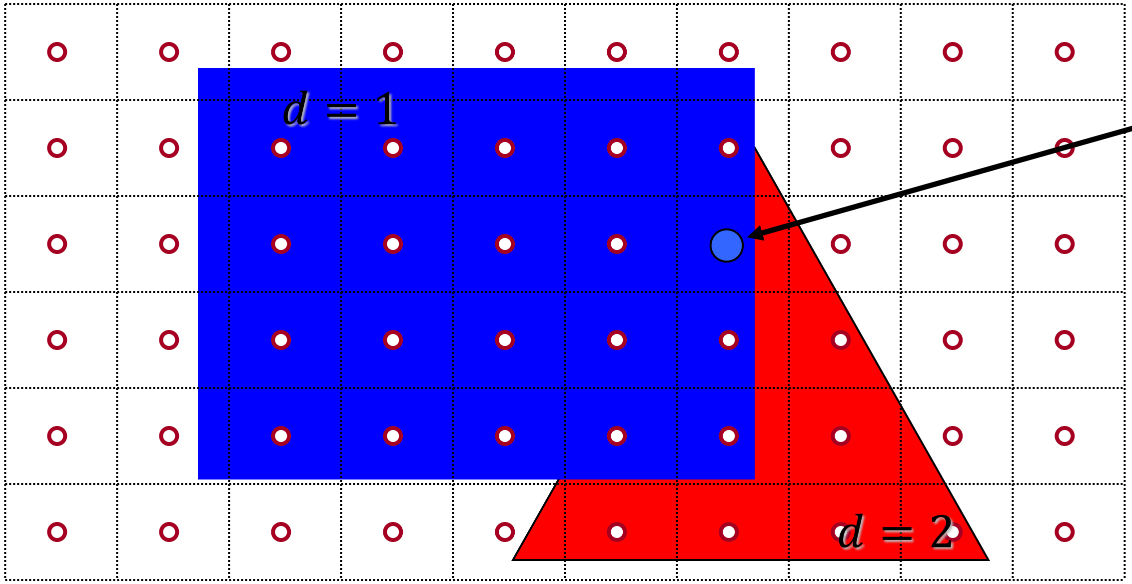
z-Buffer

- 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 what's stored in the z-buffer



z-Buffer

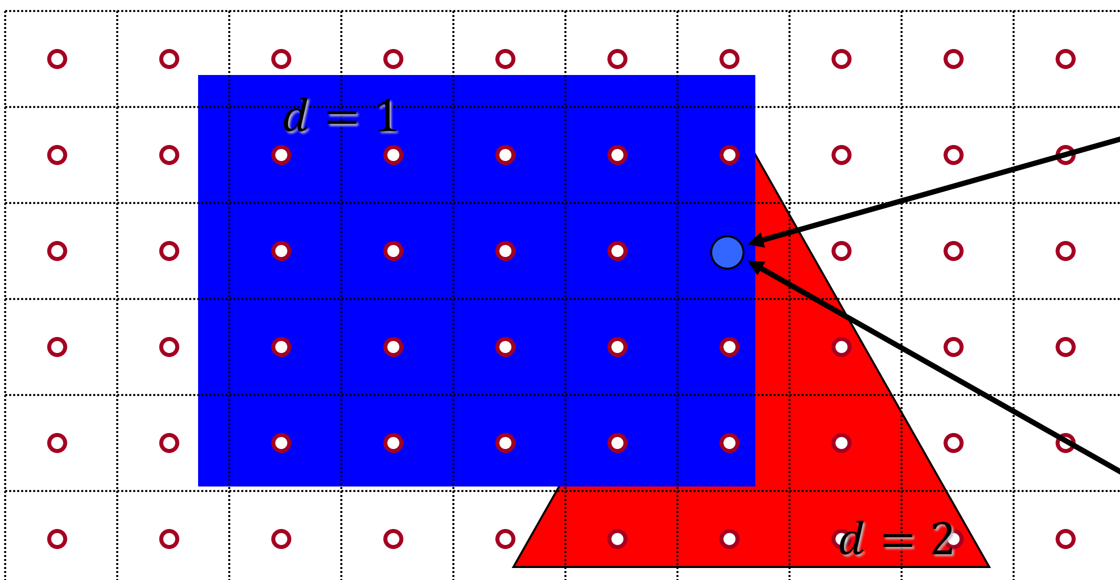
- Store color **& depth** of closest object at each pixel
  - Initialize depth of each pixel in the z-buffer to ∞
  - Only update pixels from a primitive when the depth is closer what's stored in the z-buffer



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 what's stored in the z-buffer

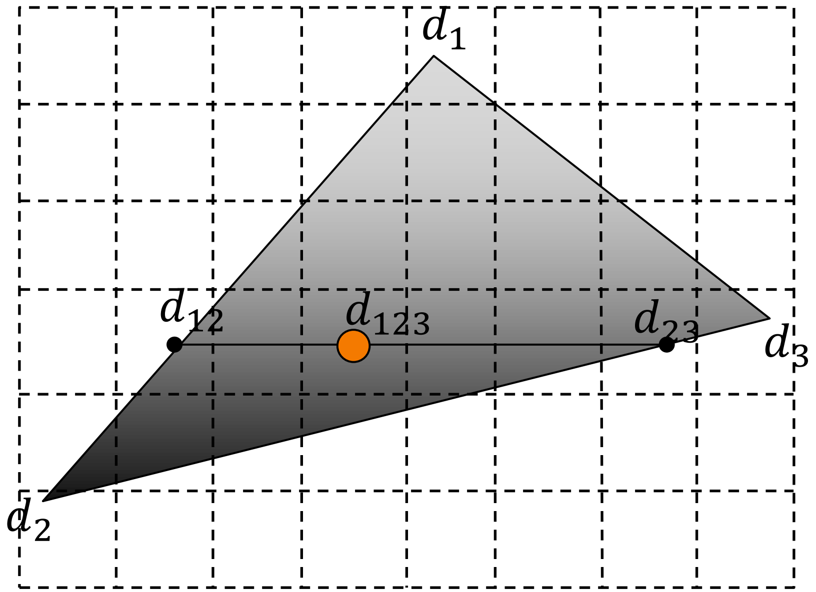


<u>Case 1 (Blue before Red)</u>:
Blue $\rightarrow (d = 1) < (d = \infty)$:
  Set $RGB = (0,0,1), d = 1$
Red $\rightarrow (d = 2) > (d = 1)$:
  Don't change pixel

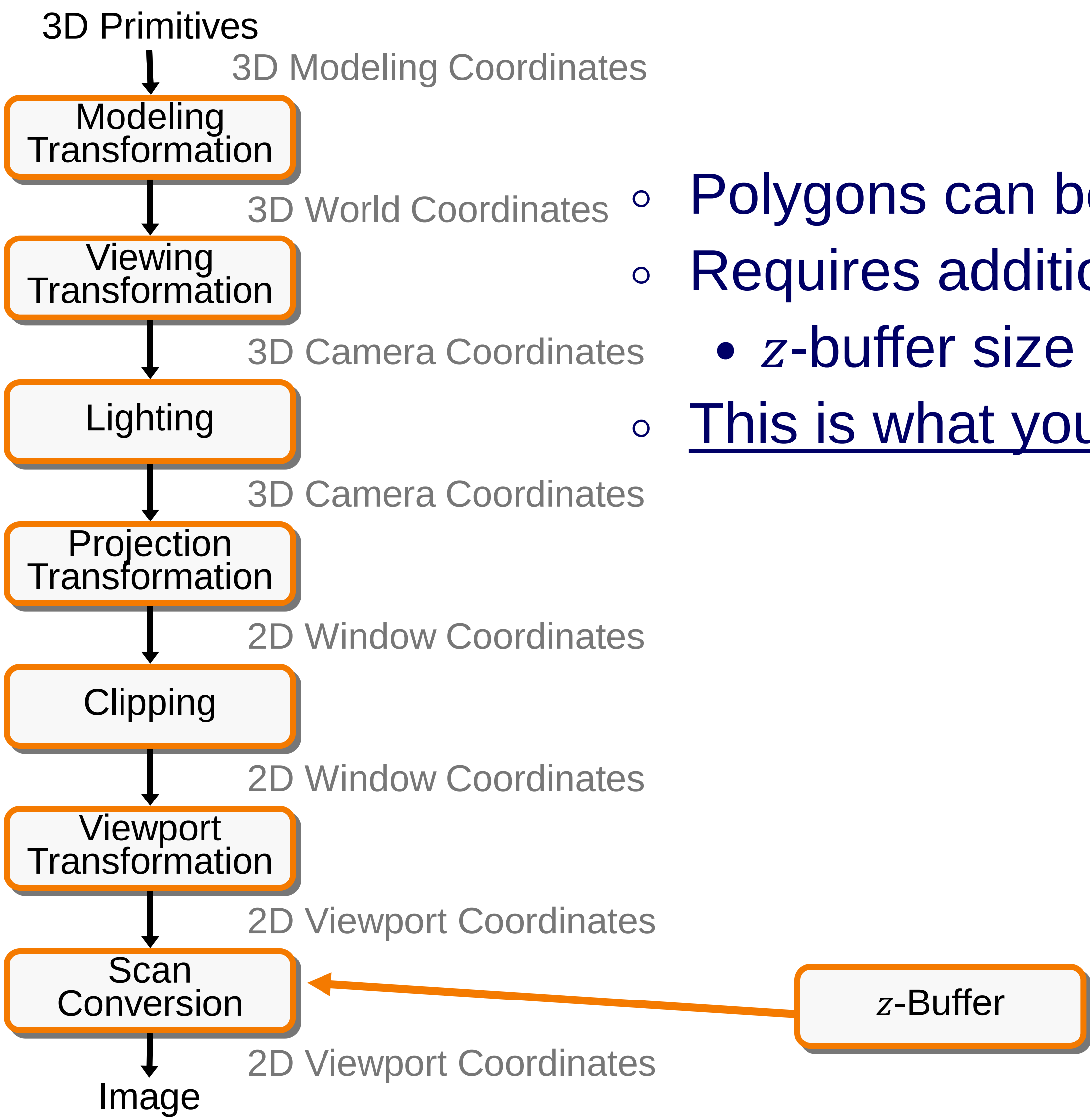
<u>Case 2 (Red before Blue)</u>:
Red $\rightarrow (d = 2) < (d = \infty)$:
  Set $RGB = (1,0,0), d = 2$
Blue $\rightarrow (d = 1) < (d = 2)$:
  Set $RGB = (0,0,1), d = 1$

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 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-buffer size $\approx$ frame buffer
- <u>This is what your graphics card does!</u>

3D Rendering Pipeline (for direct illumination)

3D Primitives

↓ 3D Modeling Coordinates

Modeling Transformation

↓ 3D World Coordinates

Viewing Transformation

↓ 3D Camera Coordinates

Lighting

↓ 3D Camera Coordinates

Projection Transformation

↓ 2D Window Coordinates

Clipping

↓ 2D Window Coordinates

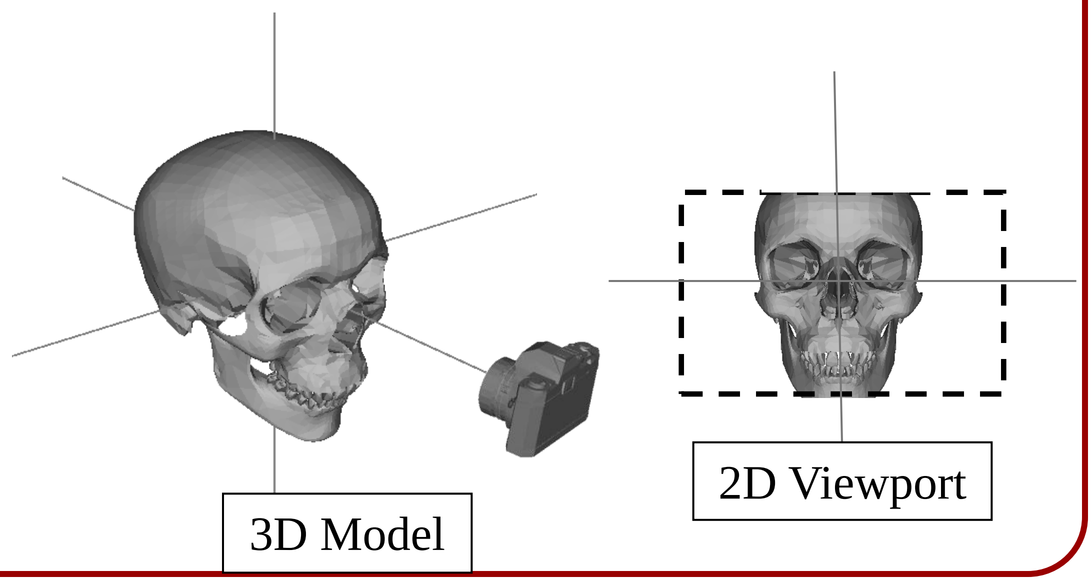
Viewport Transformation

↓ 2D Viewport Coordinates

Scan Conversion

↓ 2D Viewport Coordinates

Image



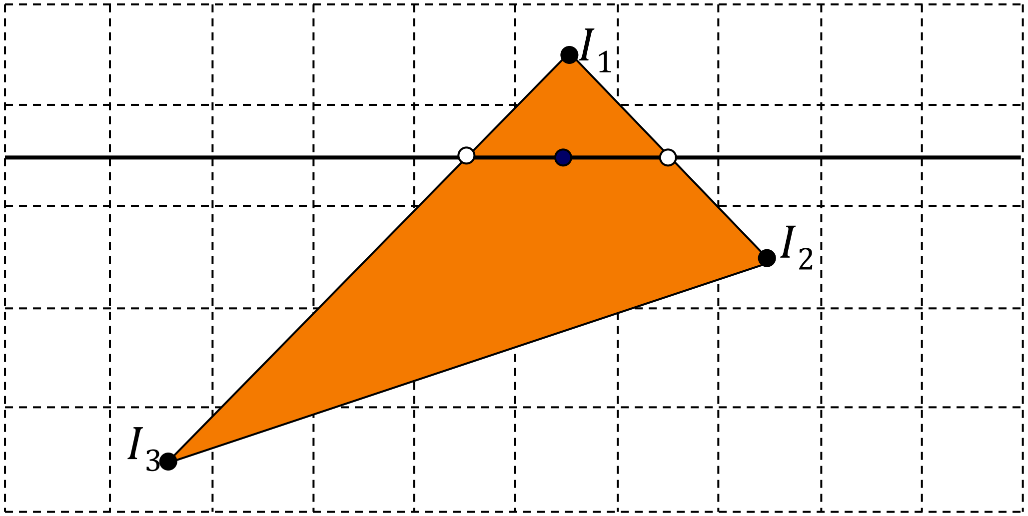
Scan Conversion

How do we average information (e.g. color, normal, depth) from the three 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)

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

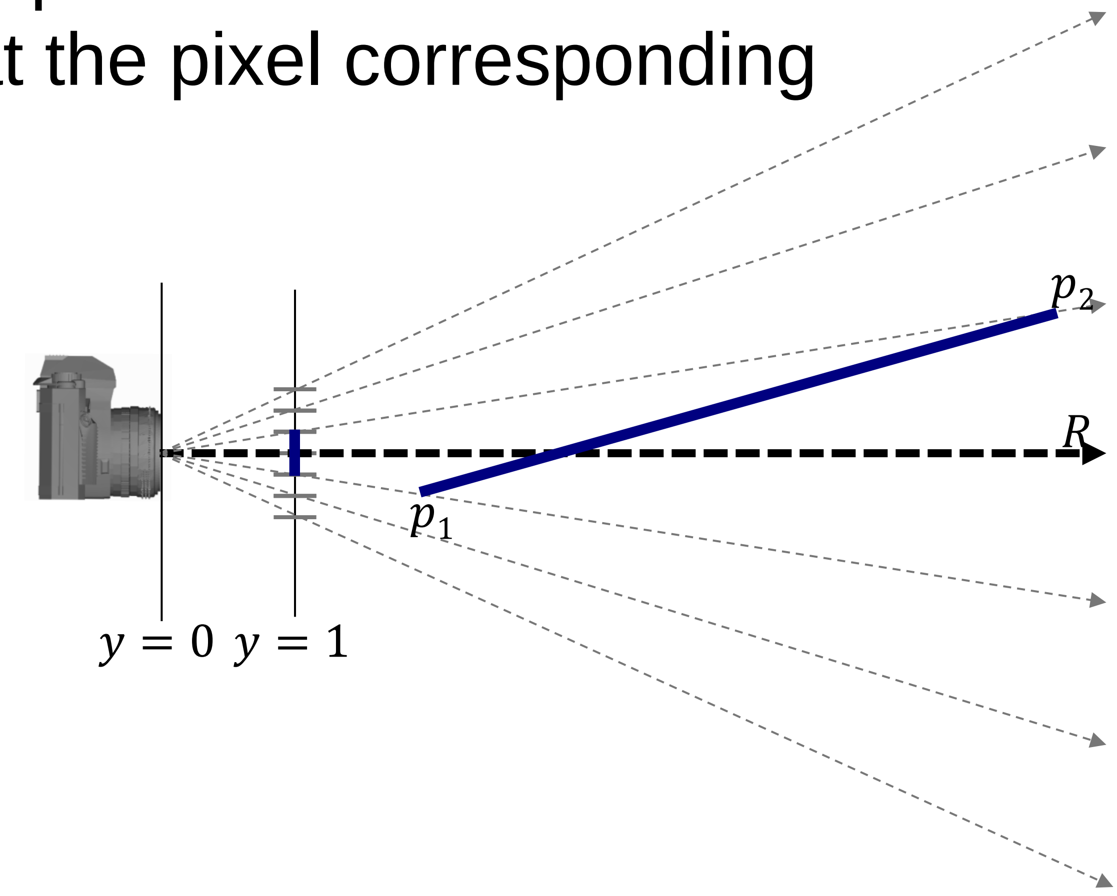
Properties of projective transformations:

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

Scan Conversion Example

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

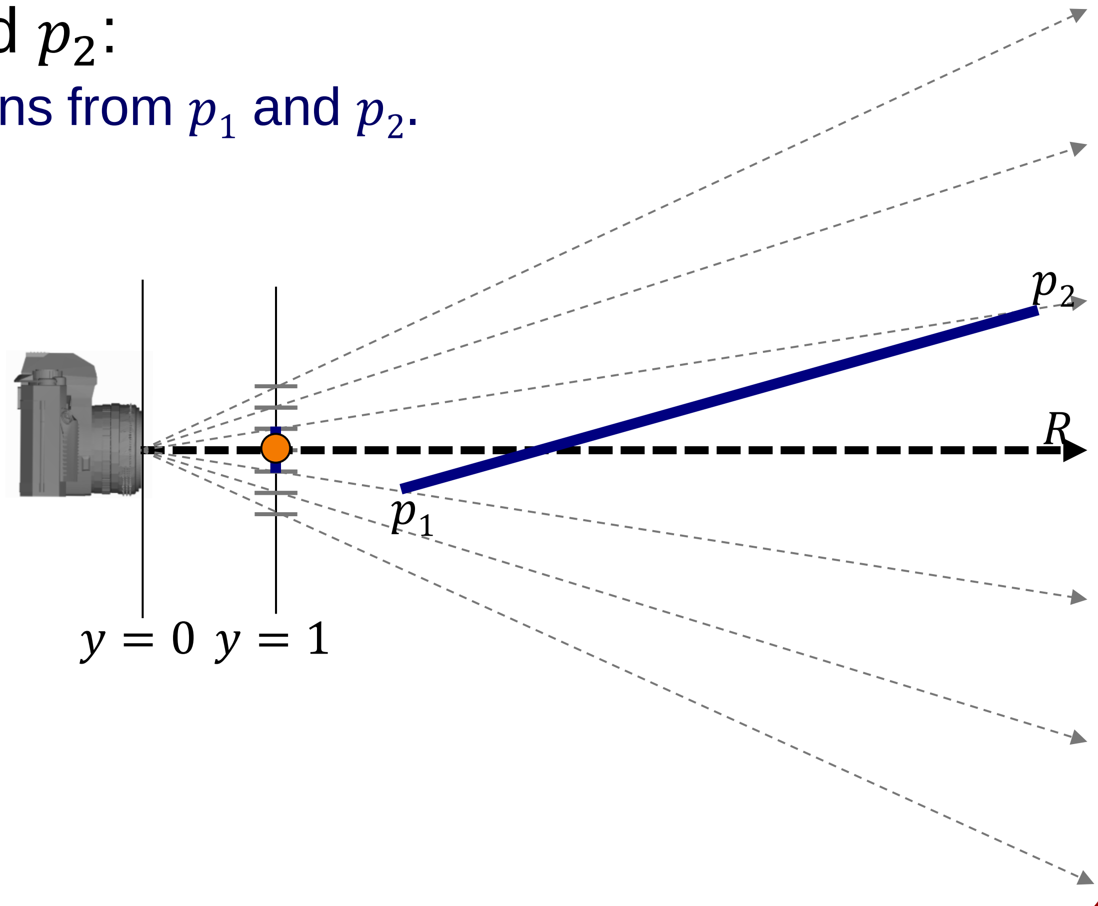
How should we interpolate the information from vertices p_1 and p_2 at the pixel corresponding to ray R?



Scan Conversion Example

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

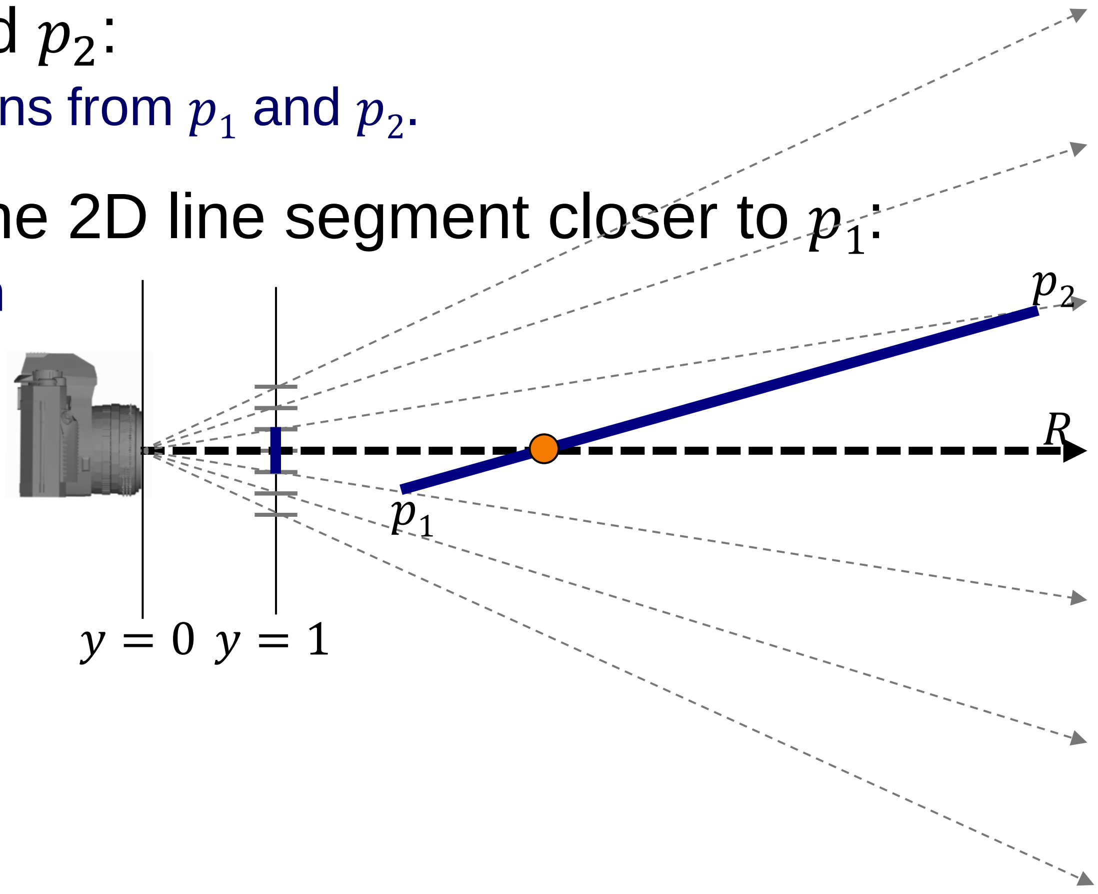
1. The ray intersects the window directly between the projections of p_1 and p_2:
   $\Rightarrow$ Use equal contributions from p_1 and p_2.



Scan Conversion Example

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

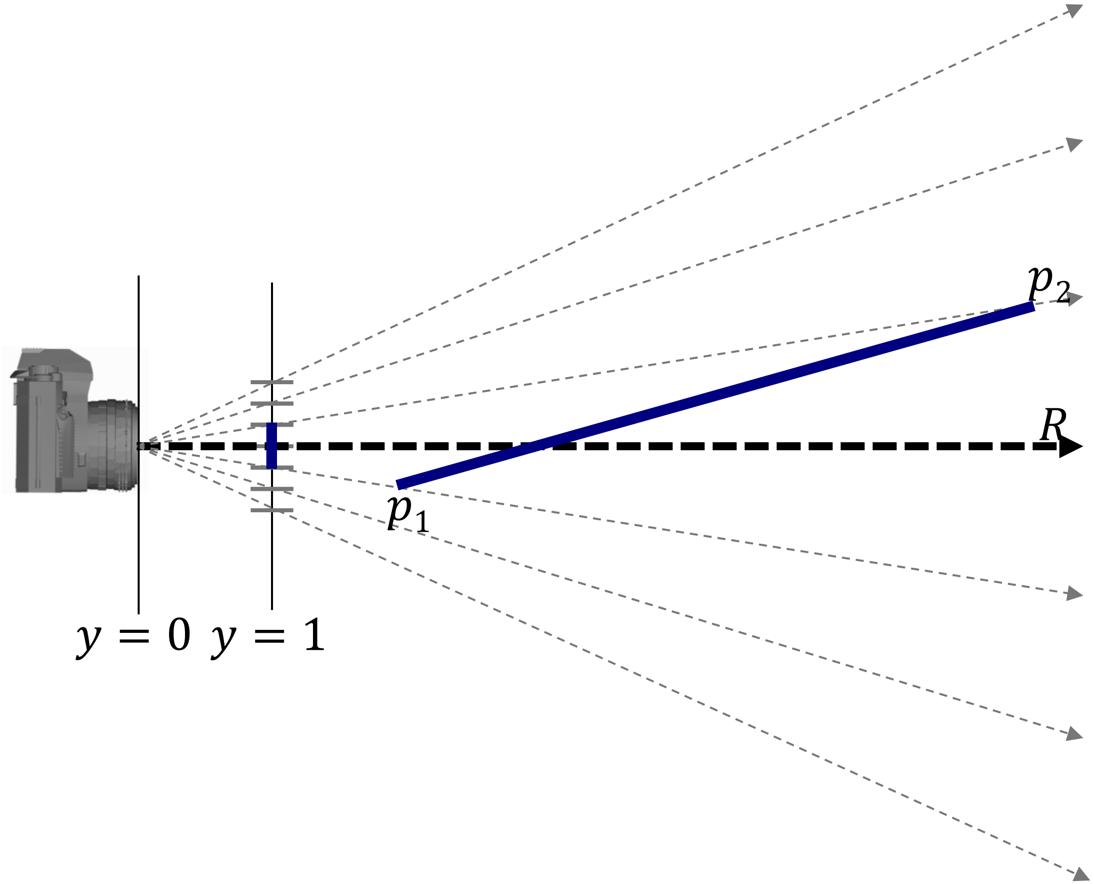
1. The ray intersects the window directly between the projections of p_1 and p_2:
   ⇒ Use equal contributions from p_1 and p_2.
2. The ray intersects the 2D line segment closer to p_1:
   ⇒ Use more information from p_1 than from p_2.



Scan Conversion Example

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

- How do we interpolate correctly?



Scan Conversion Example

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

- How do we interpolate correctly?

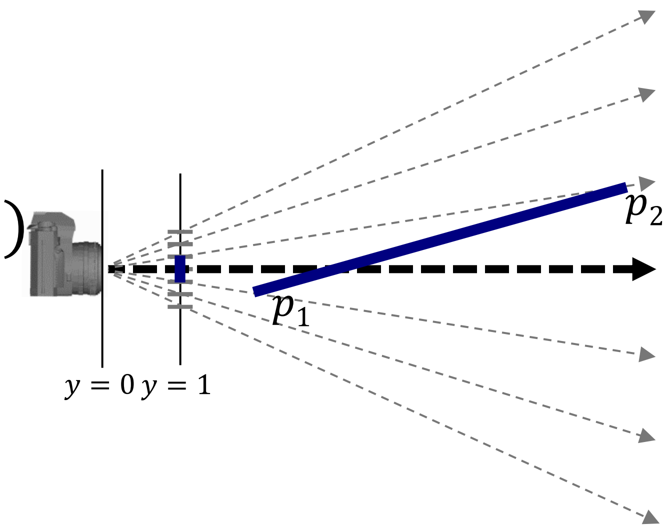
Recall: The 2D point (x, y) maps to the point (x/y) in 1D.

If $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, to find the blending value α for a pixel falling at position x in the screen we need to solve:

$$(1-\alpha)(x_1, y_1) + \alpha(x_2, y_2) \sim (x, 1)$$

$$\big((1-\alpha)x_1 + \alpha x_2, (1-\alpha)y_1 + \alpha y_2\big) \sim (x, 1)$$

$$\frac{(1-\alpha)x_1 + \alpha x_2}{(1-\alpha)y_1 + \alpha y_2} = \frac{x}{1}$$



Scan Conversion Example

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

- How do we interpolate correctly?

<u>Recall</u>: The 2D point (x, z) maps to the point (x/z) in 1D.

If $p_1 = ($ … value α for a pixe… …d to solve:

$(1-\alpha)$ …

$((1-\alpha)x_1$ …

$(1-\alpha)z_1 + \alpha z_2 \quad 1$

To compute the interpolation weights, perform a <u>perspective divide</u>:

$$\frac{(1-\alpha)x_1 + \alpha x_2}{(1-\alpha)y_1 + \alpha y_2} = \frac{x}{1}$$

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

$$(1-\alpha)\frac{x_1}{y_1} + \alpha\frac{x_2}{y_2} = \frac{x}{1}$$