Shading and Visibility

Connelly Barnes

CS 4810: Graphics

Acknowledgment: slides by Jason Lawrence, Misha Kazhdan, Allison Klein, Tom Funkhouser, Adam Finkelstein and David Dobkin

3D Rendering Pipeline (for direct illumination)

3D Primitives

↓ 3D Modeling Coordinates

Modeling Transformation

↓ 3D World Coordinates

Camera Transformation

↓ 3D Camera Coordinates

Lighting

↓ 3D Camera Coordinates

Projection Transformation

↓ 2D Screen Coordinates

Clipping

↓ 2D Screen Coordinates

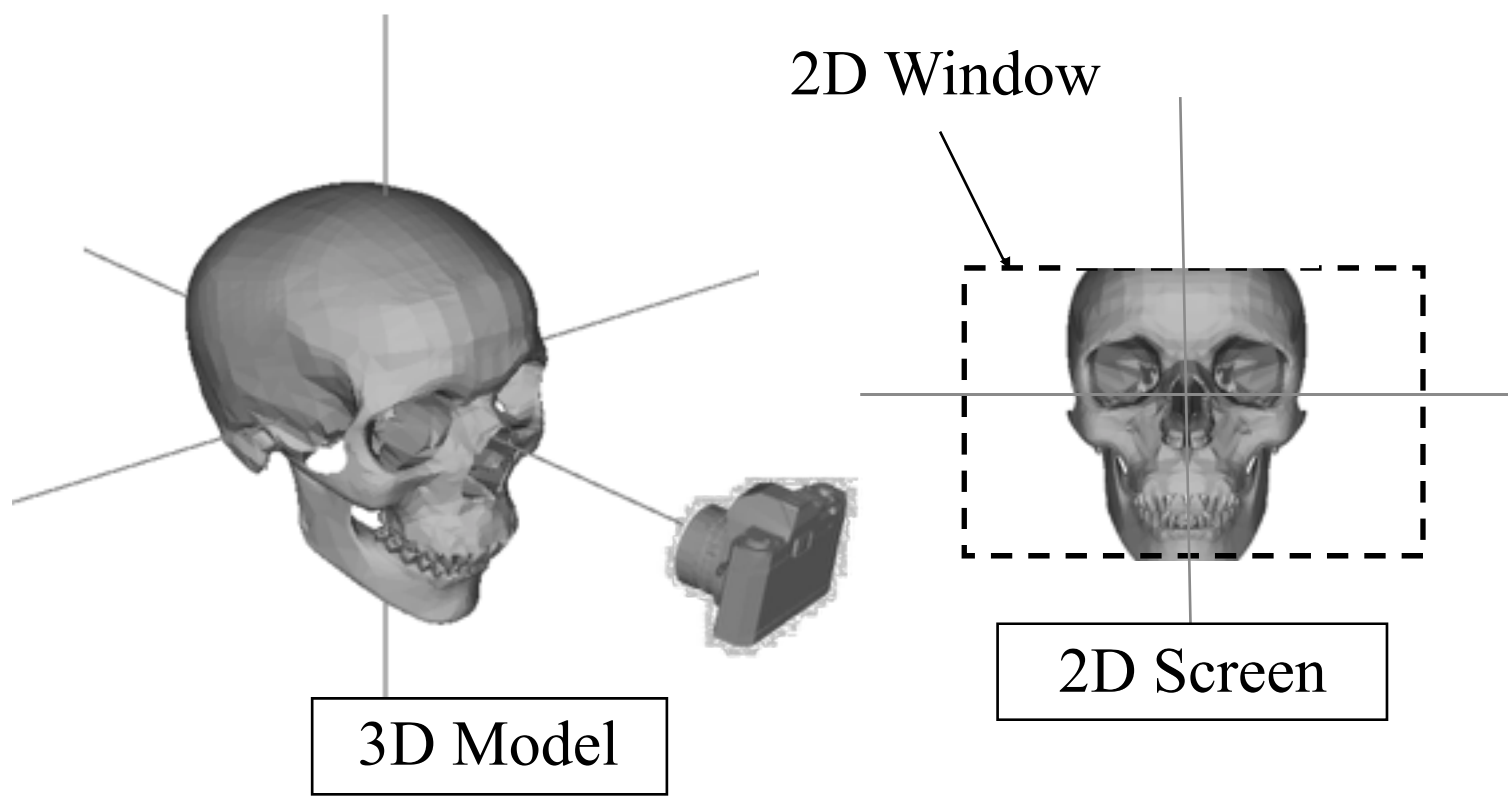
Viewport Transformation

↓ 2D Image Coordinates

Scan Conversion

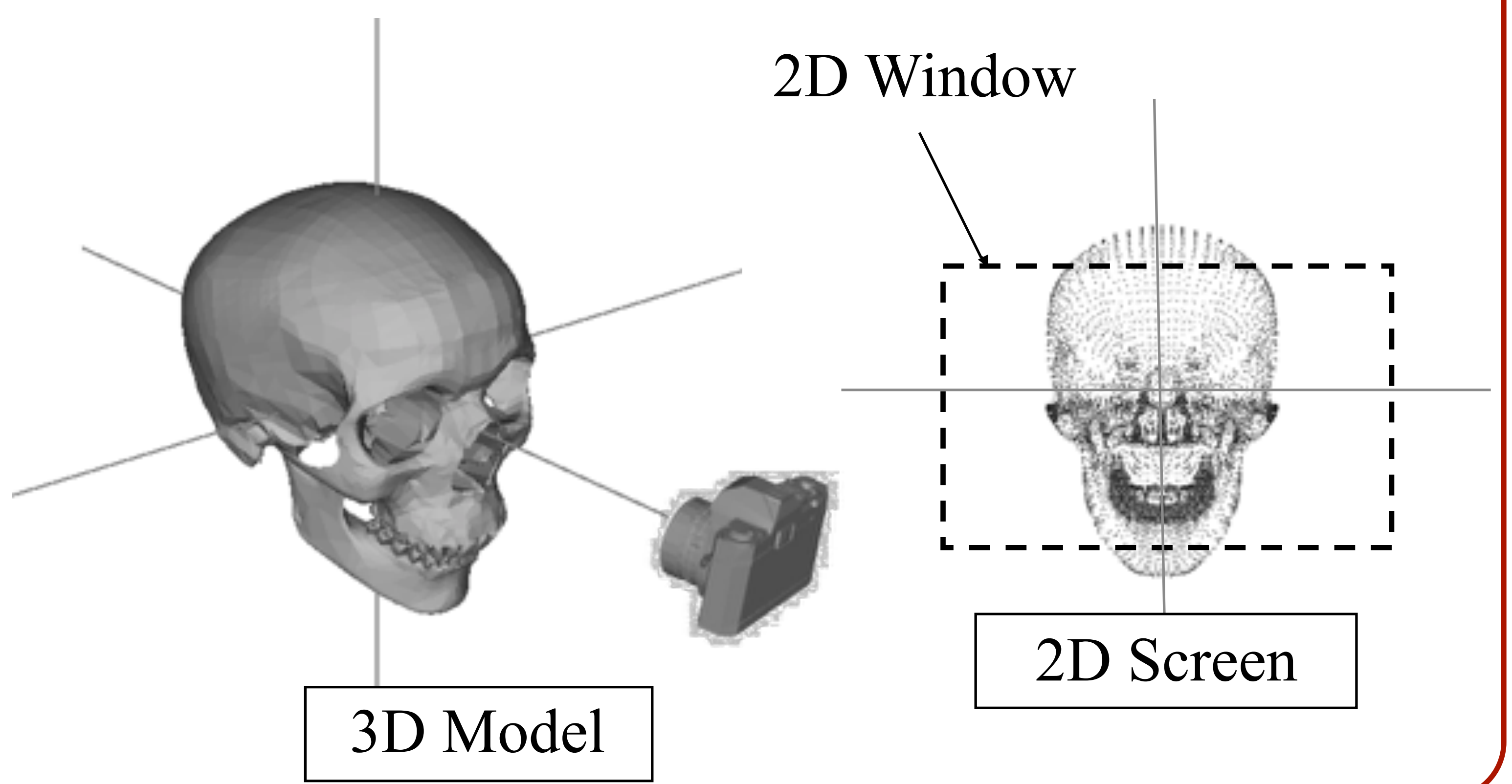
↓ 2D Image Coordinates

Image



3D Rendering Pipeline (for direct illumination)

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



3D Rendering Pipeline (for direct illumination)

3D Primitives

↓ 3D Modeling Coordinates

Modeling Transformation

↓ 3D World Coordinates

Camera Transformation

↓ 3D Camera Coordinates

Lighting

↓ 3D Camera Coordinates

Projection Transformation

↓ 2D Screen Coordinates

Clipping

↓ 2D Screen Coordinates

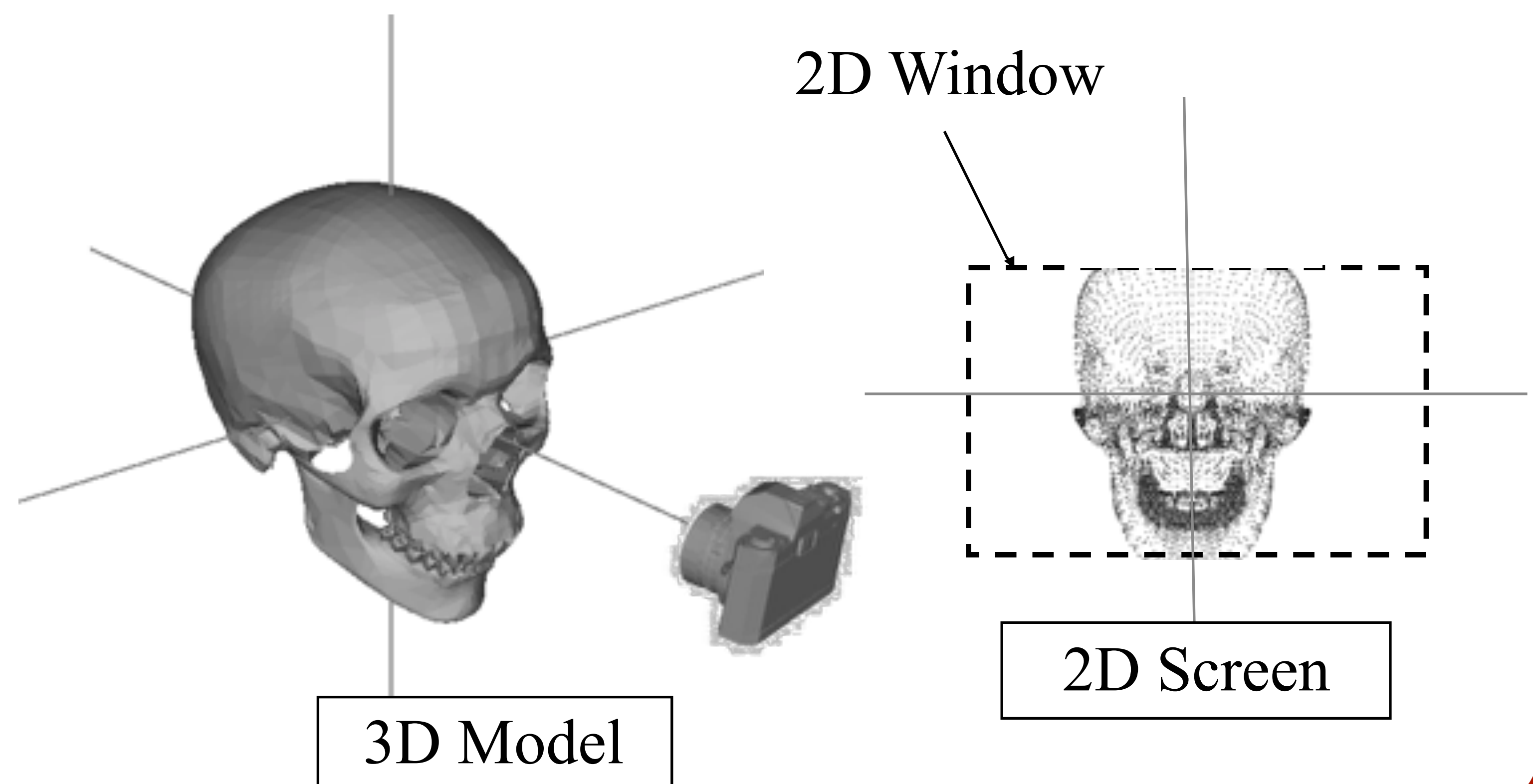
Viewport Transformation

↓ 2D Image Coordinates

Scan Conversion

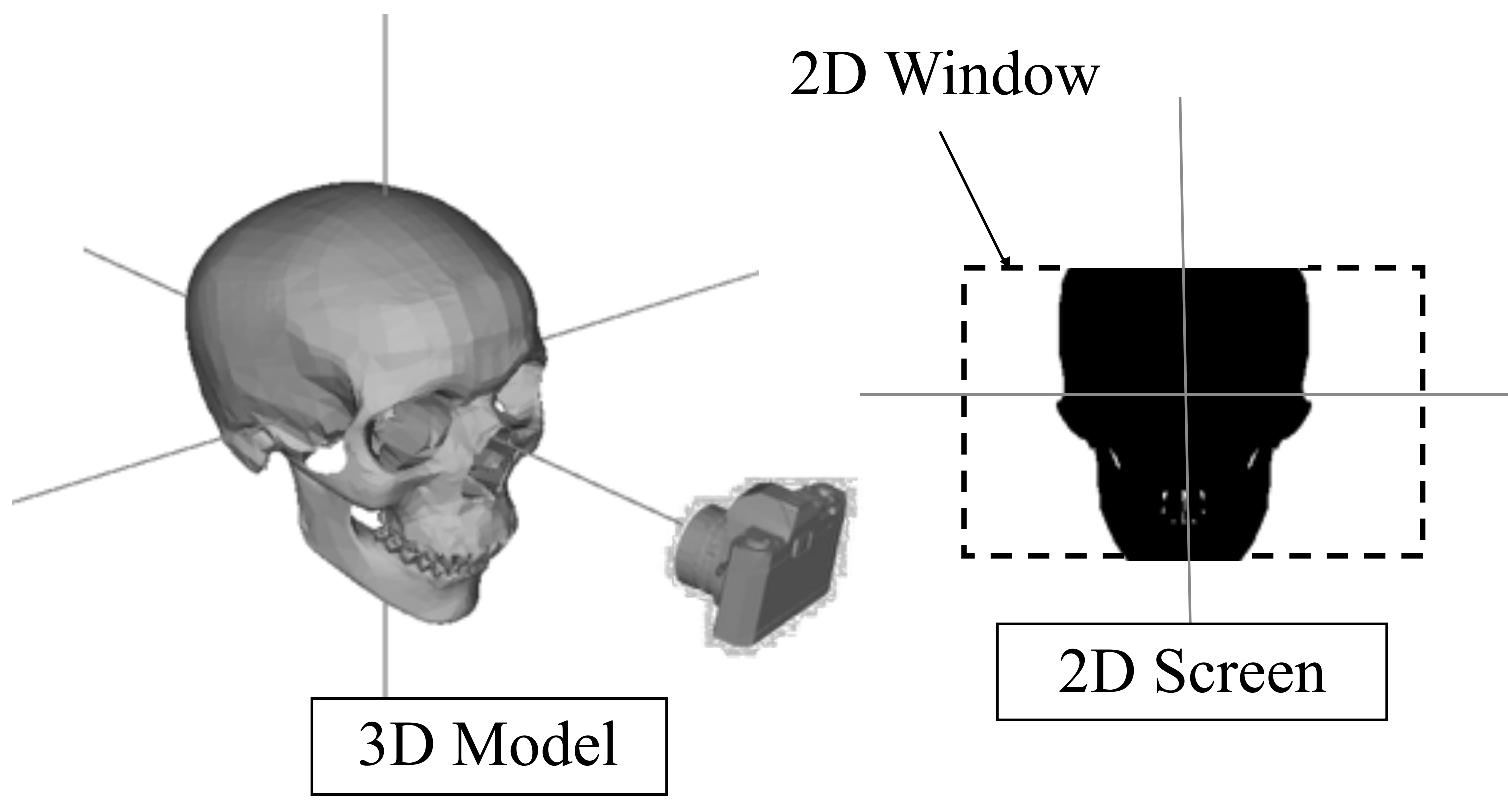
↓ 2D Image Coordinates

Image



3D Rendering Pipeline (for direct illumination)

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

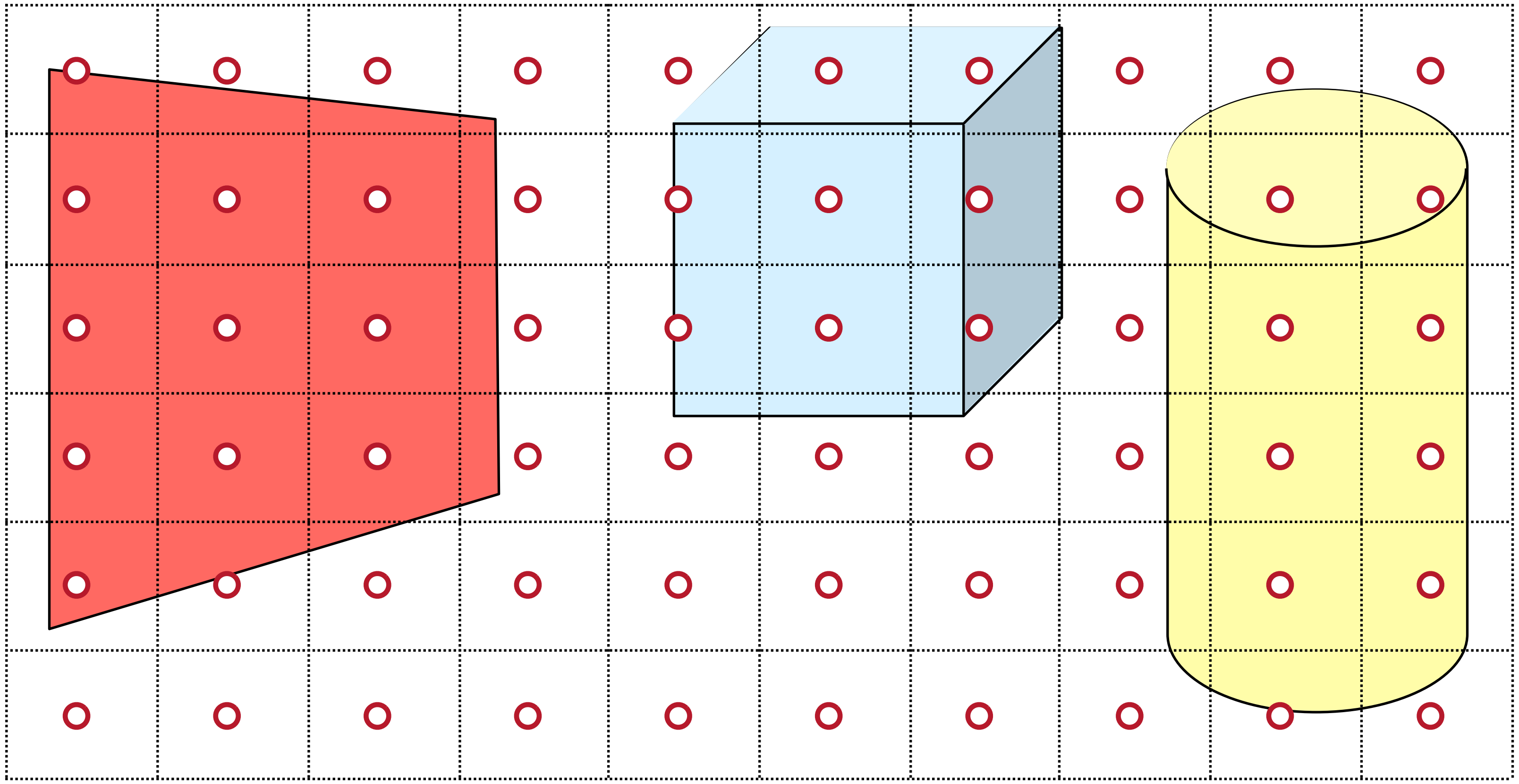


Overview

- 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

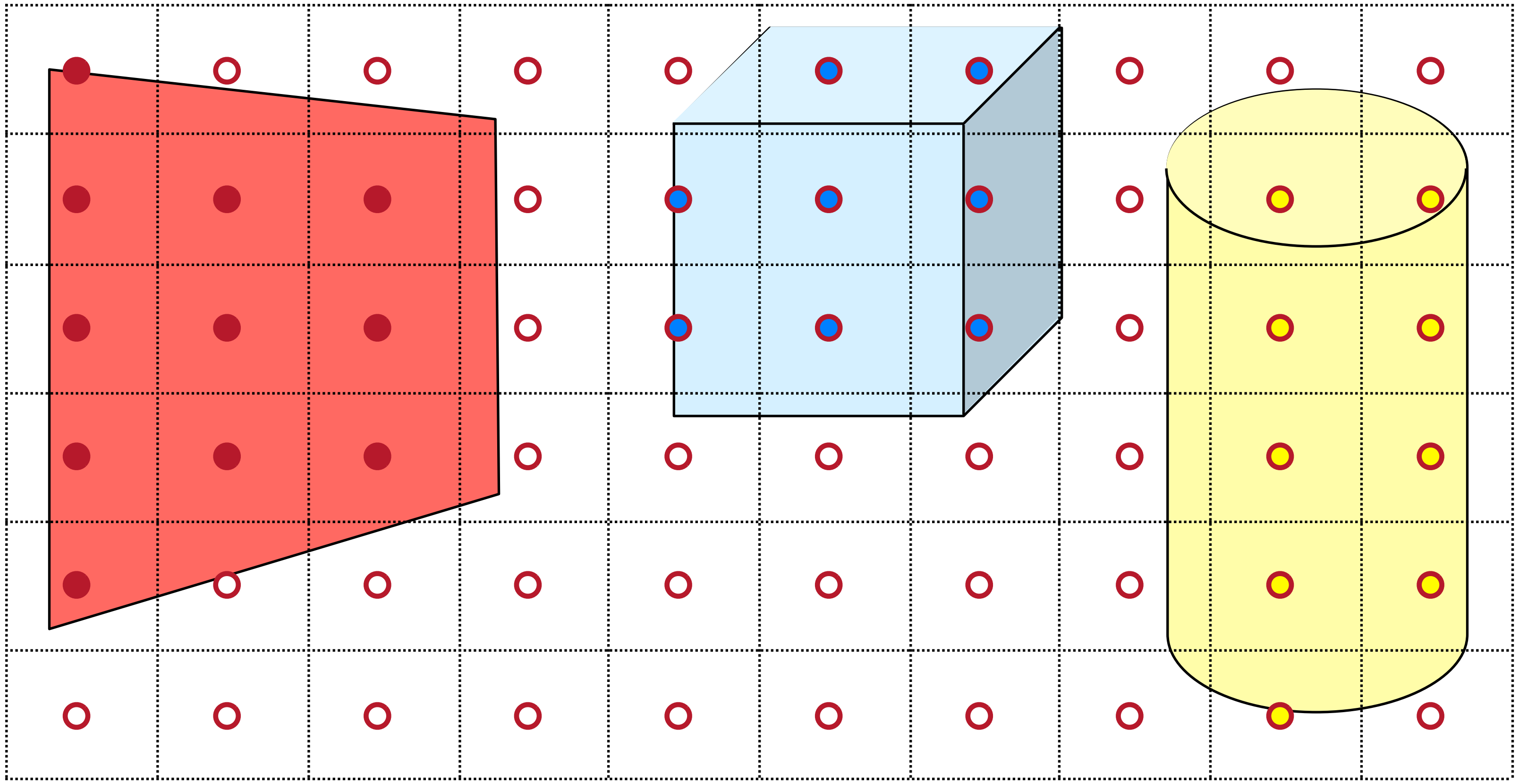
- Simplest shading approach is to perform independent lighting calculation for every pixel
  - When is this unnecessary?



$$I = I_E + K_A I_{AL} + \sum_i (K_D (N \bullet L_i) I_i + K_S (V \bullet R_i)^n I_i)$$

Polygon Shading

- Can take advantage of spatial coherence
  - Illumination calculations for pixels covered by same primitive are related to each other



$$I = I_E + K_A I_{AL} + \sum_i (K_D (N \bullet L_i) I_i + K_S (V \bullet R_i)^n I_i)$$

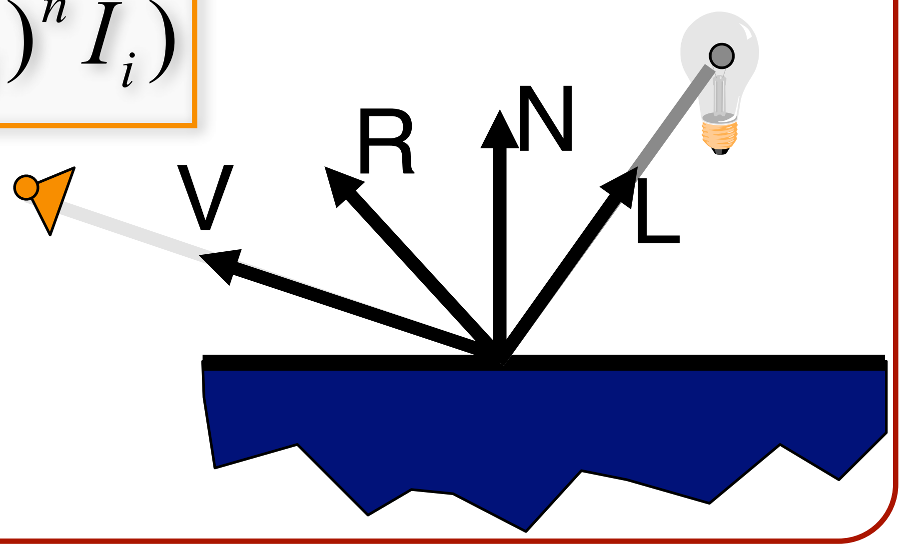
Polygon Shading Algorithms

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

Flat Shading

- Can take advantage of spatial coherence
  - Make the lighting equation constant over the surface of each primitive

$$I = I_E + K_A I_{AL} + \sum_i (K_D (N \bullet L_i) I_i + K_S (V \bullet R_i)^n I_i)$$



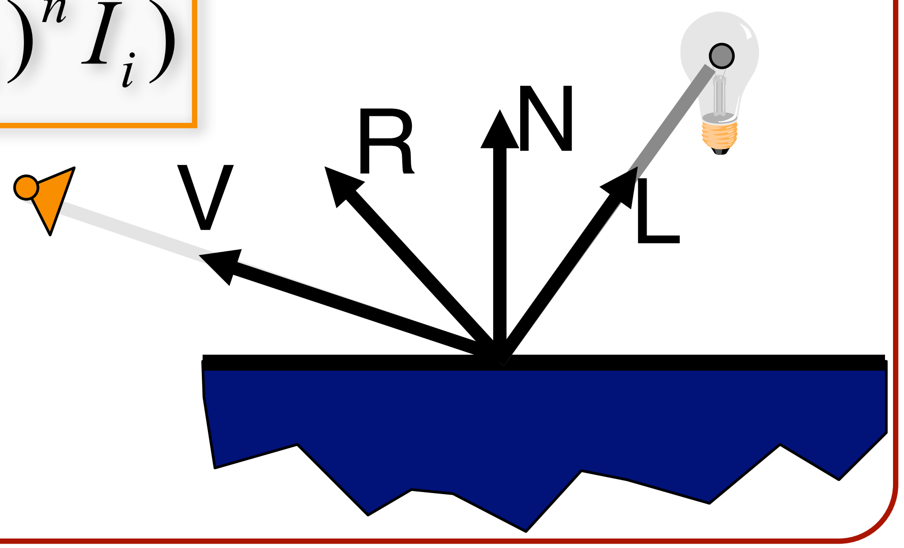
Flat Shading

- Can take advantage of spatial coherence
  - Make the lighting equation constant over the surface of each primitive

- 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

$$I = I_E + K_A I_{AL} + \sum_i (K_D (\boxed{N \bullet L_i}) I_i + K_S (V \bullet R_i)^n I_i)$$



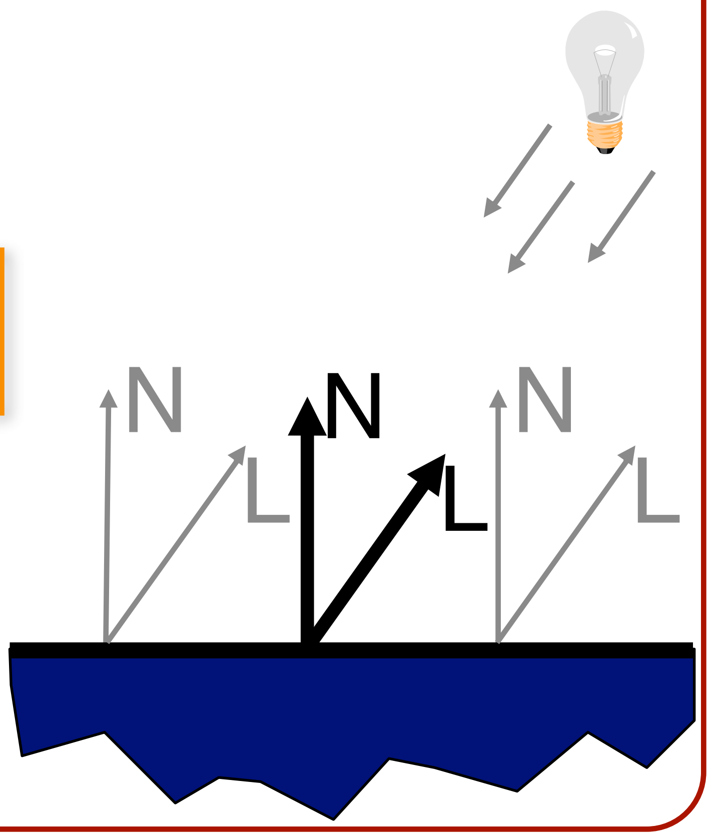
Flat Shading

- Can take advantage of spatial coherence
  - o Make the lighting equation constant over the surface of each primitive

- 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

$$I = I_E + K_A I_{AL} + \sum_i (K_D (\boxed{N \bullet L_i}) I_i + K_S (V \bullet R_i)^n I_i)$$



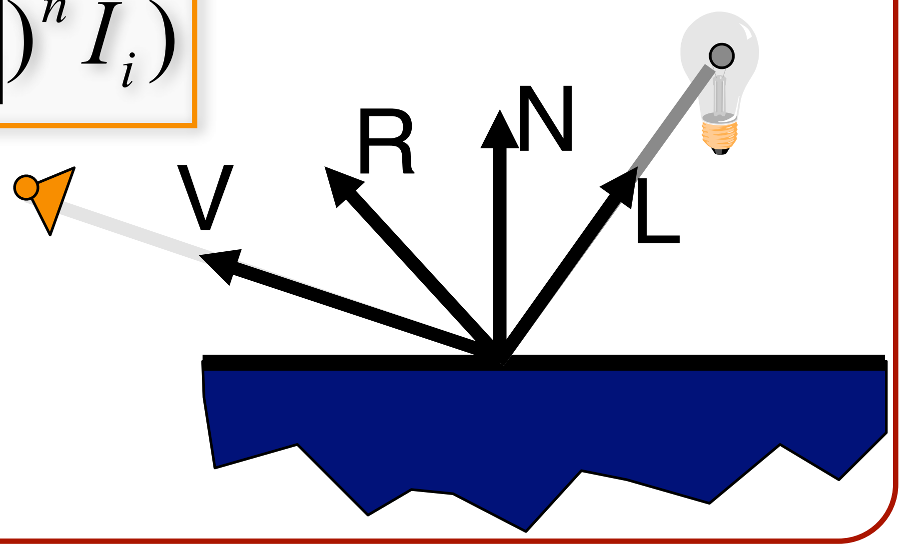
Flat Shading

- Can take advantage of spatial coherence
  - Make the lighting equation constant over the surface of each 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

$$I = I_E + K_A I_{AL} + \sum_i (K_D (N \bullet L_i) I_i + K_S (\boxed{V \bullet R_i})^n I_i)$$



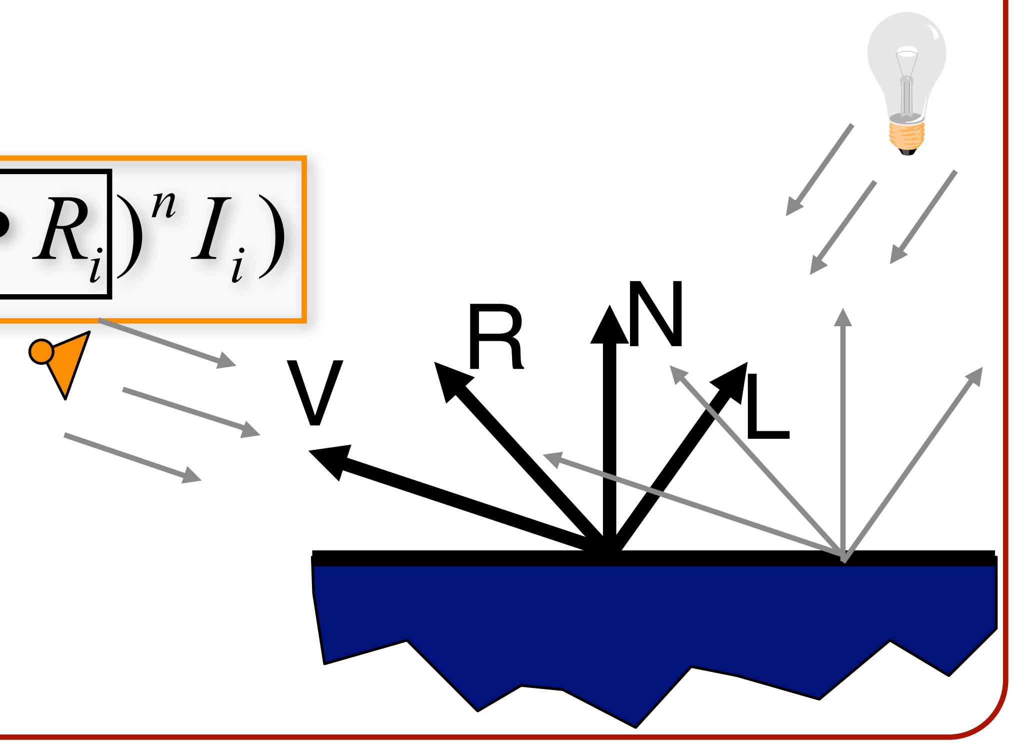
Flat Shading

- Can take advantage of spatial coherence
  - Make the lighting equation constant over the surface of each 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

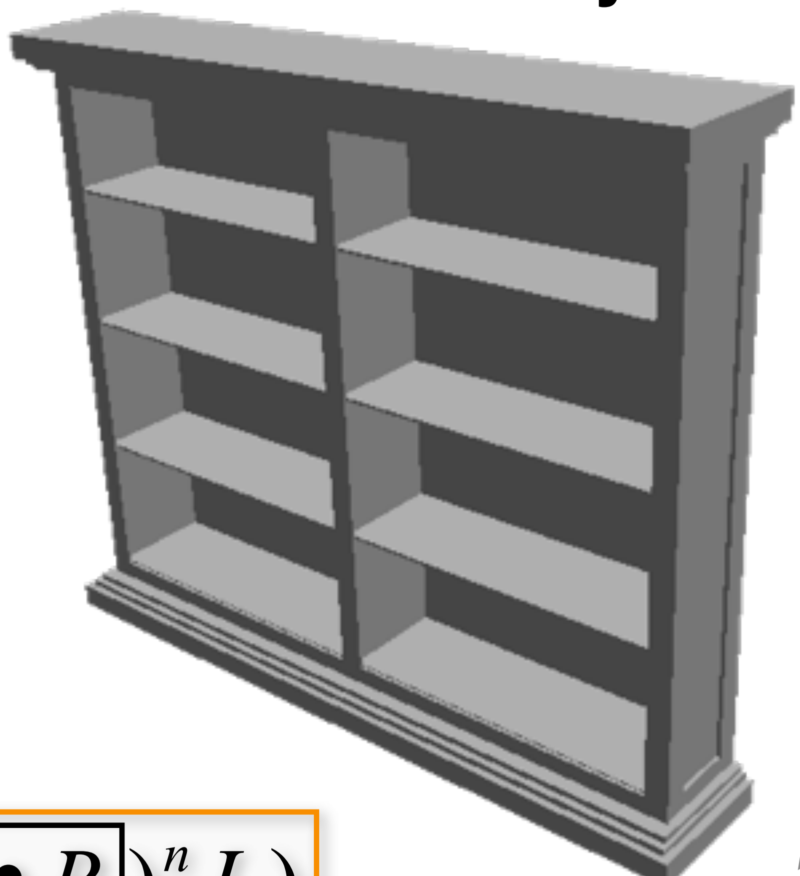
the specular component is the same for all points on the primitive

$$I = I_E + K_A I_{AL} + \sum_i (K_D (N \bullet L_i) I_i + K_S (\boxed{V \bullet R_i})^n I_i)$$

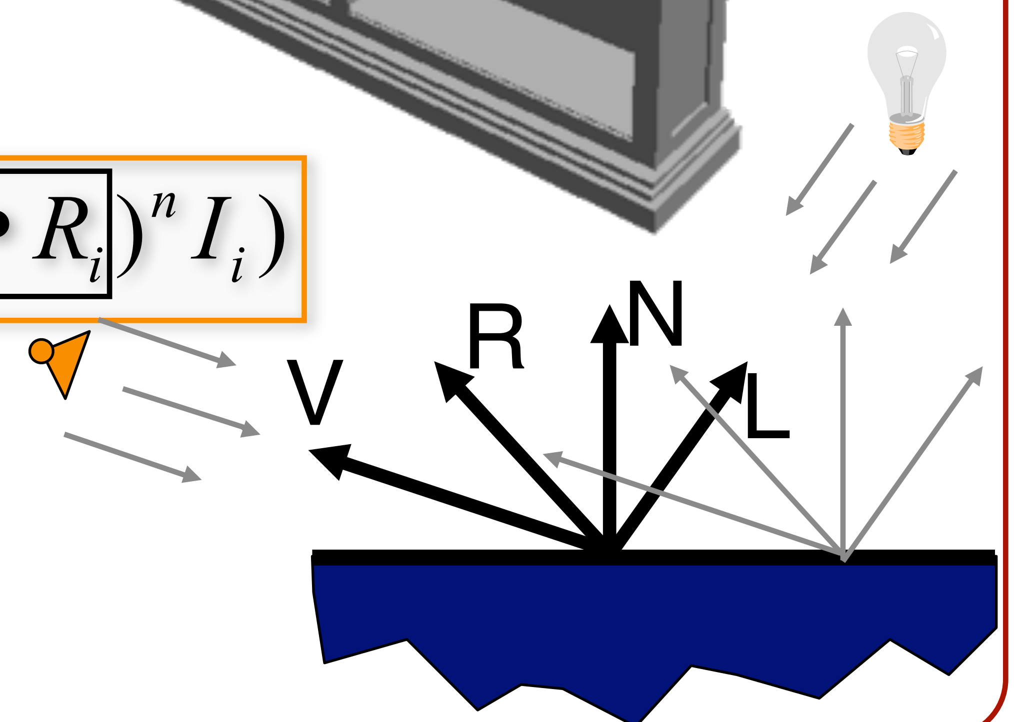


Flat Shading

- Illuminate as though all light sources are directional, the polygon is flat, and is viewed from infinitely far away
  - $N \cdot L_i$ constant over polygon
  - Attenuation function constant over polygon
  - $V \cdot R$ constant over surface

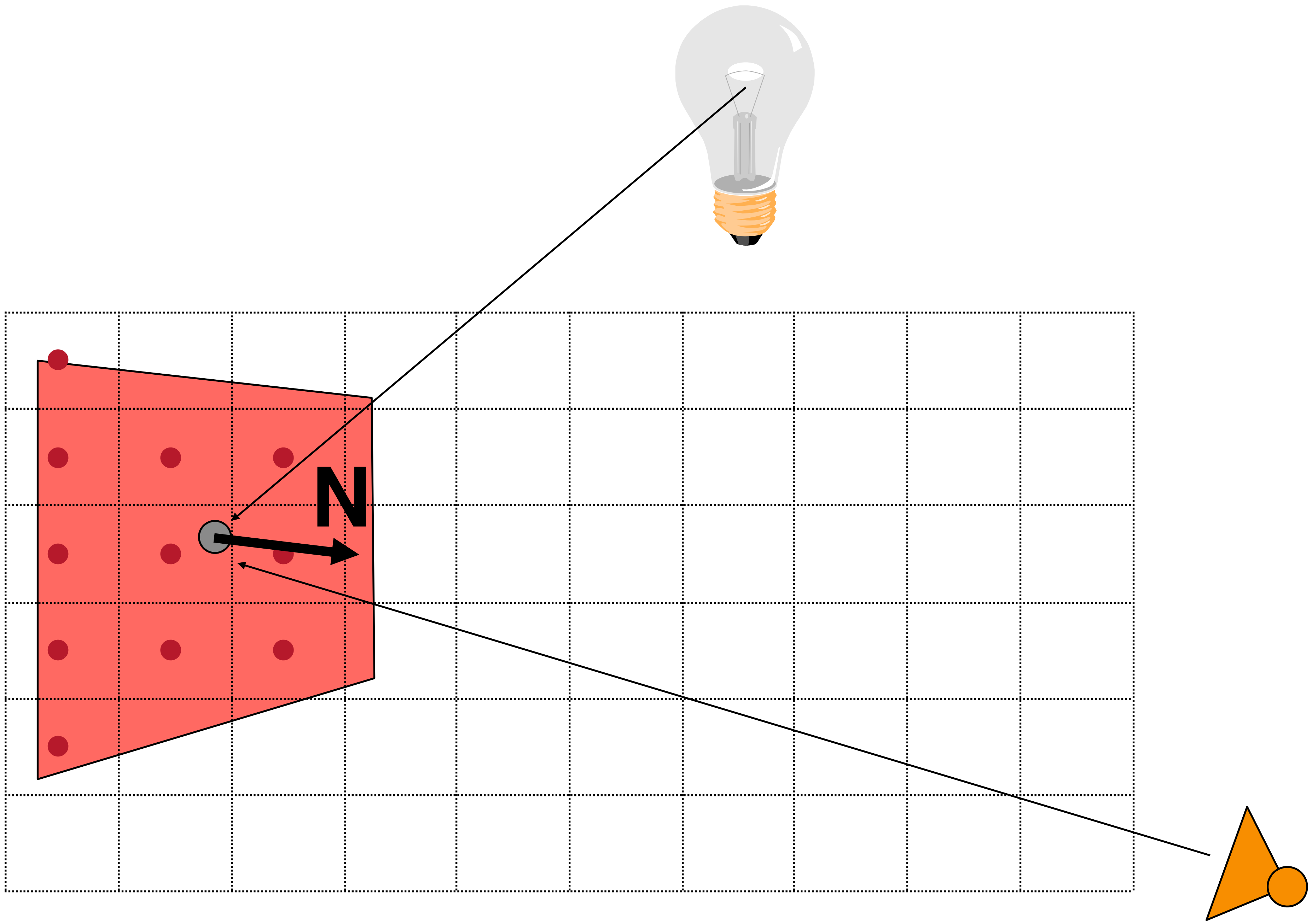


$$I = I_E + K_A I_{AL} + \sum_i (K_D (\boxed{N \bullet L_i}) I_i + K_S (\boxed{V \bullet R_i})^n I_i)$$



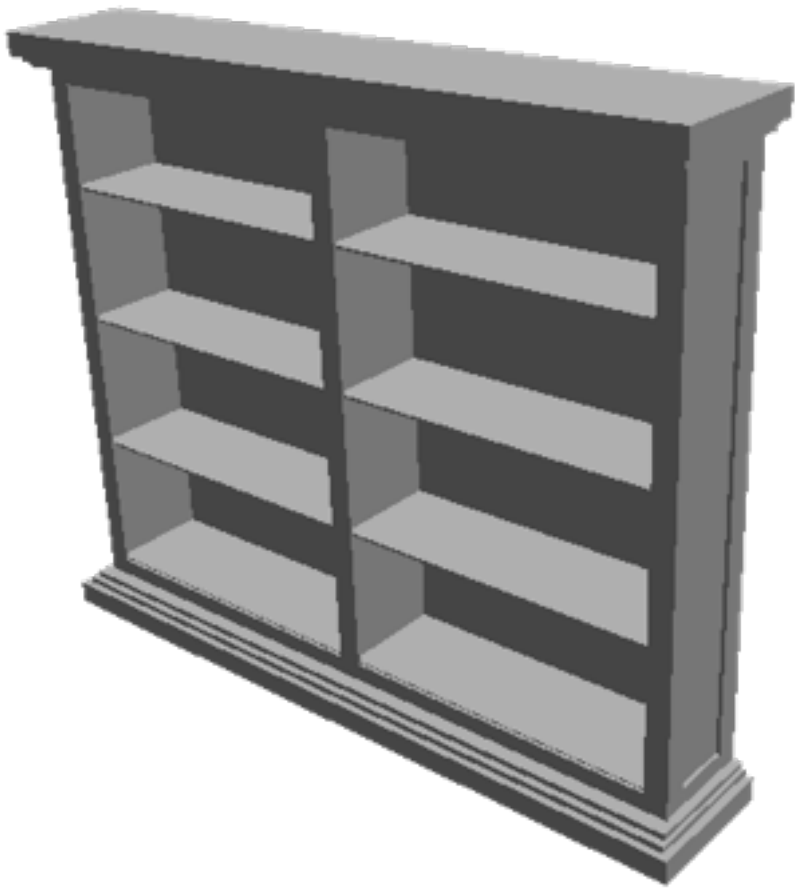
Flat Shading

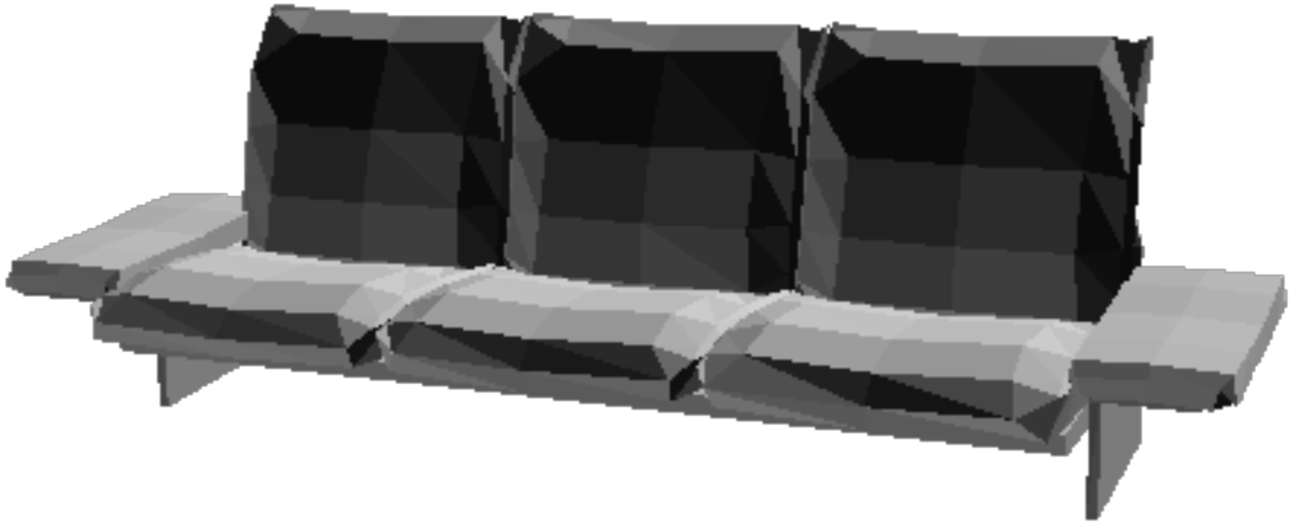
- One lighting calculation per polygon
  - Assign all pixels inside each polygon the same color



Flat Shading

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



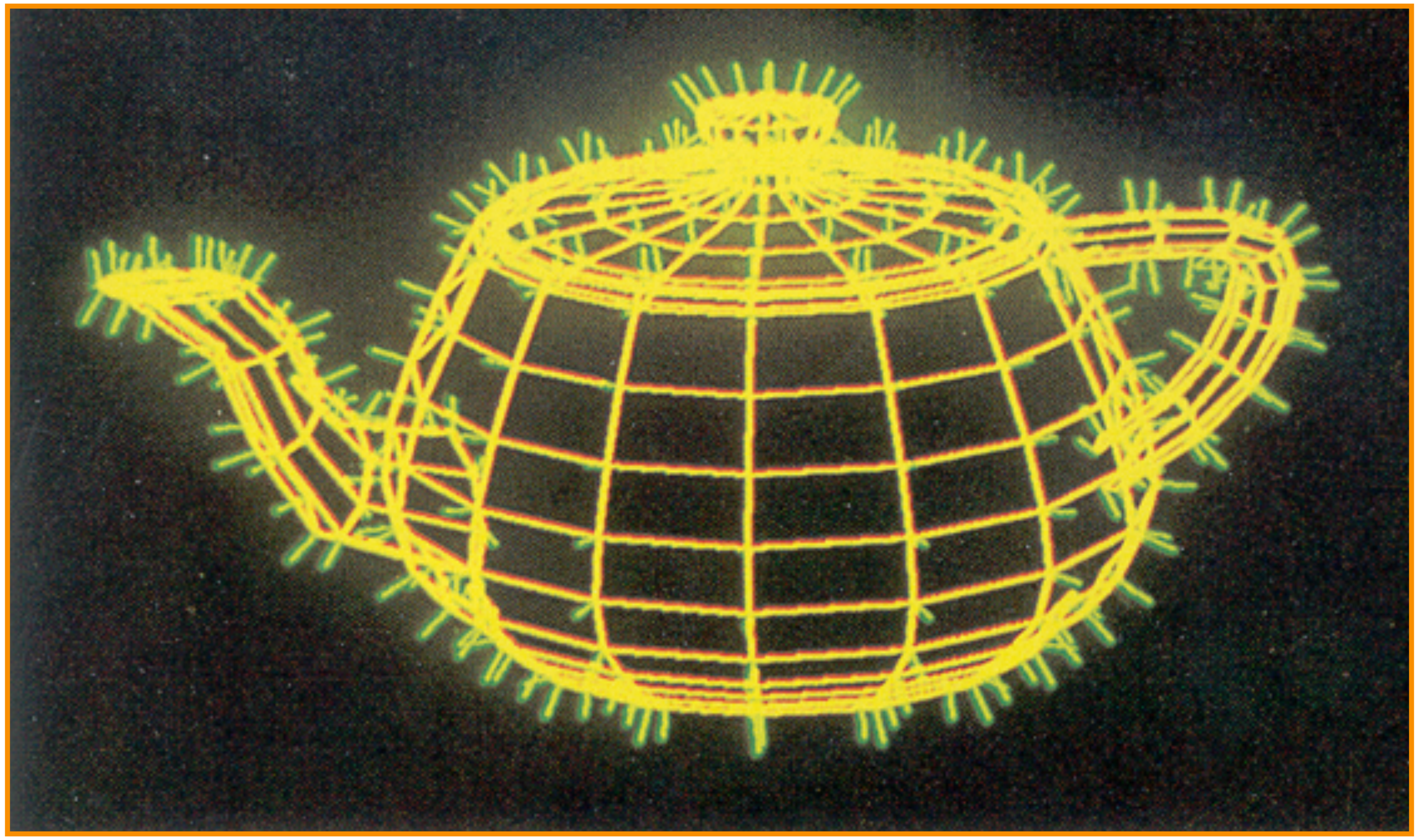


Polygon Shading Algorithms

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

Gouraud Shading

- What if smooth surface is represented by polygonal mesh with a normal at each vertex?

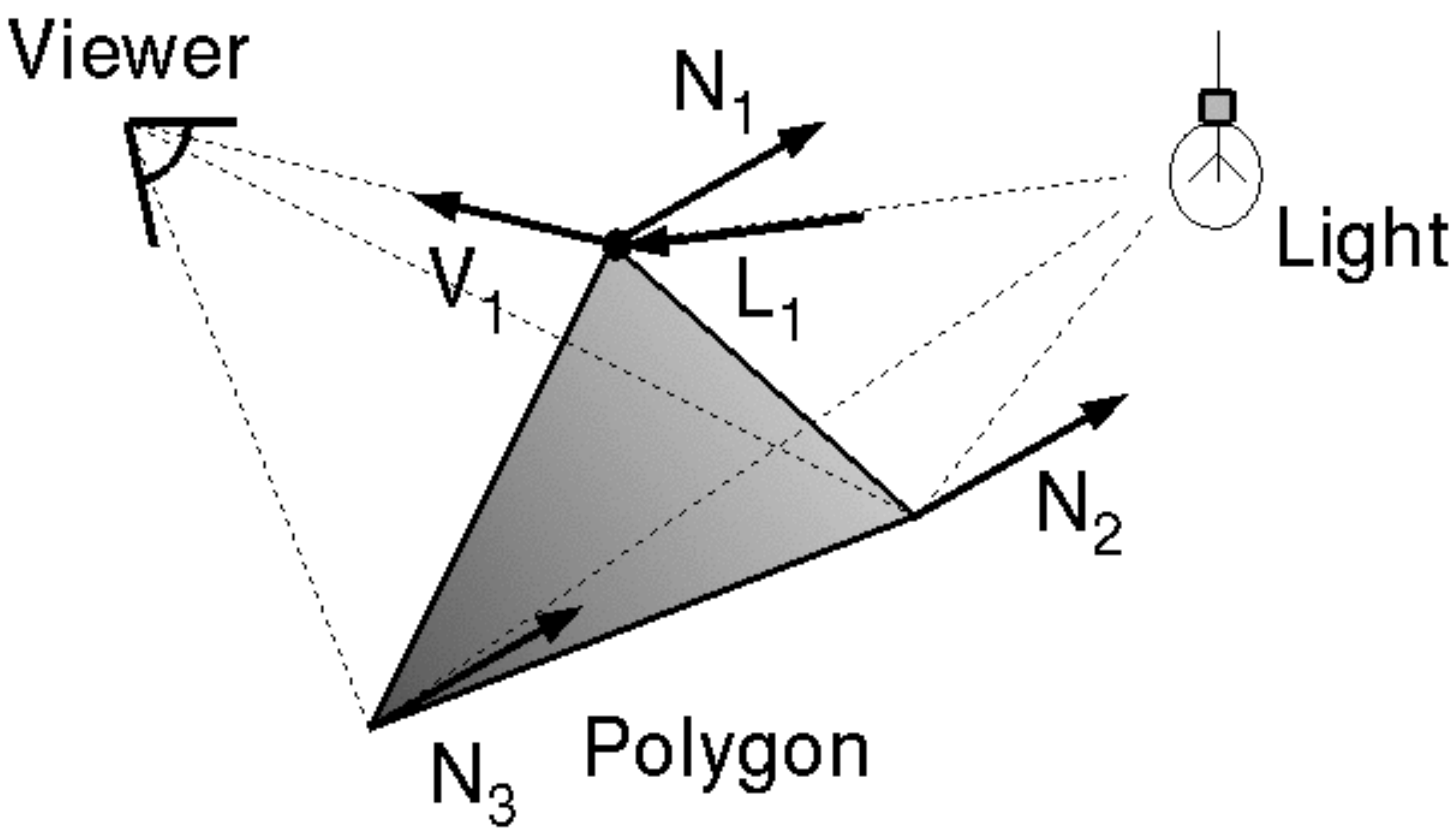


Watt Plate 7

$$I = I_E + K_A I_{AL} + \sum_i (K_D (N \bullet L_i) I_i + K_S (V \bullet R_i)^n I_i)$$

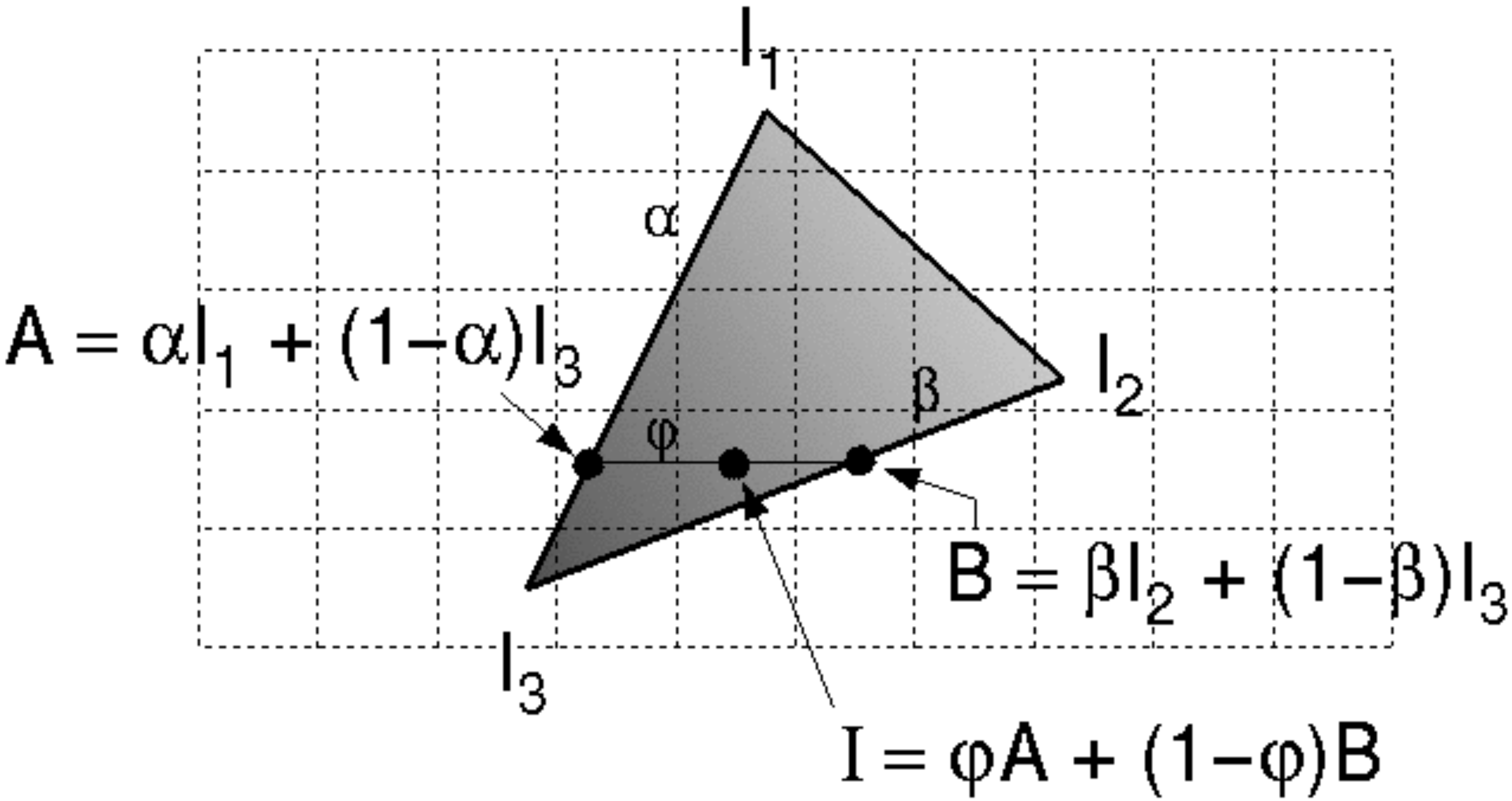
Gouraud Shading

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



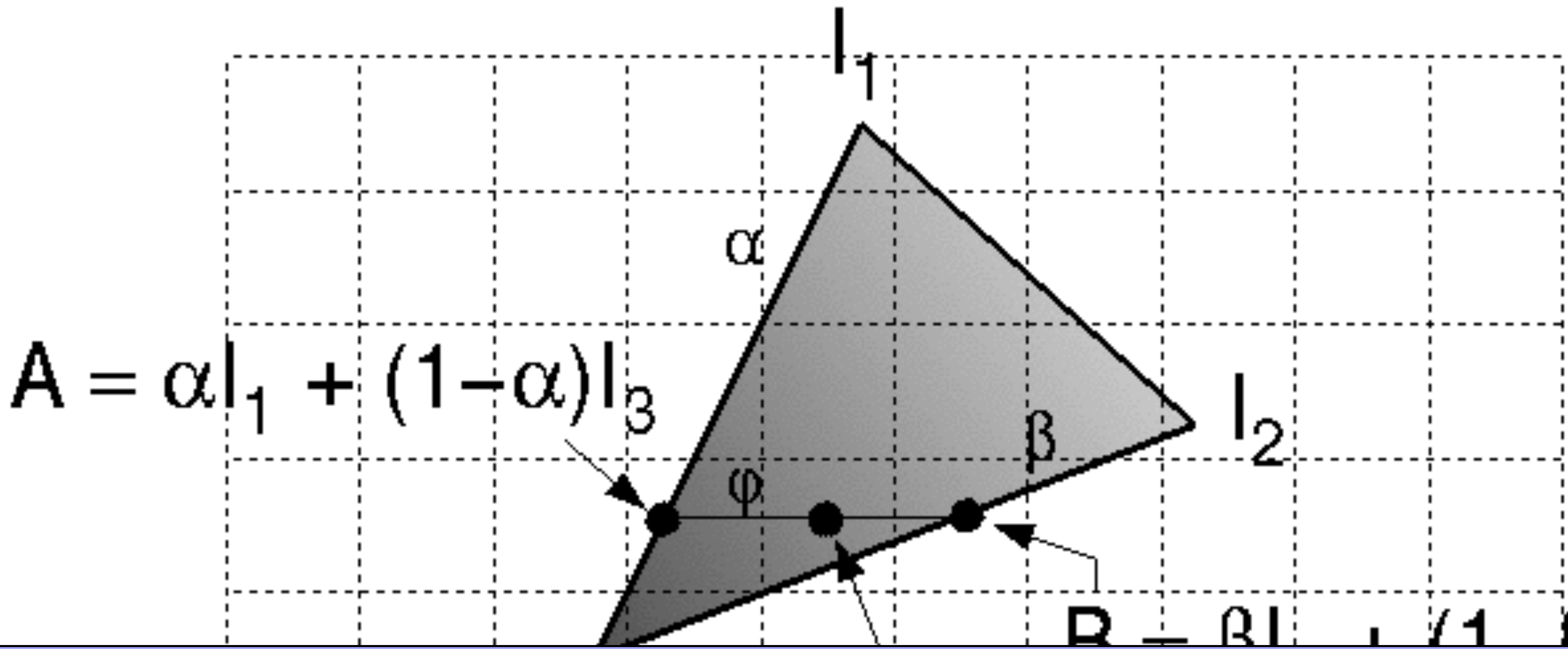
Gouraud Shading

- Bilinearly interpolate colors at vertices down and across scan lines



Gouraud Shading

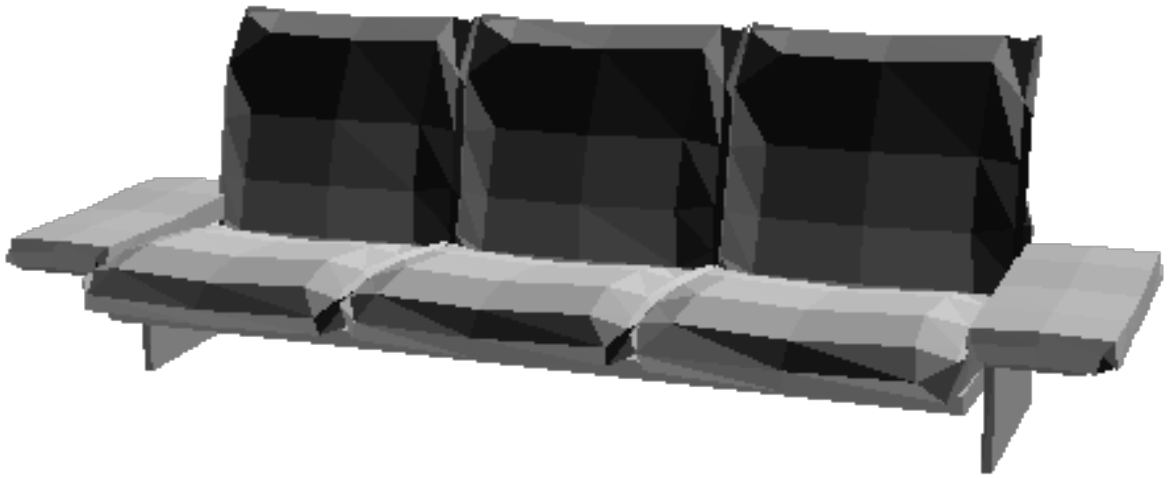
- Bilinearly interpolate colors at vertices down and across scan lines



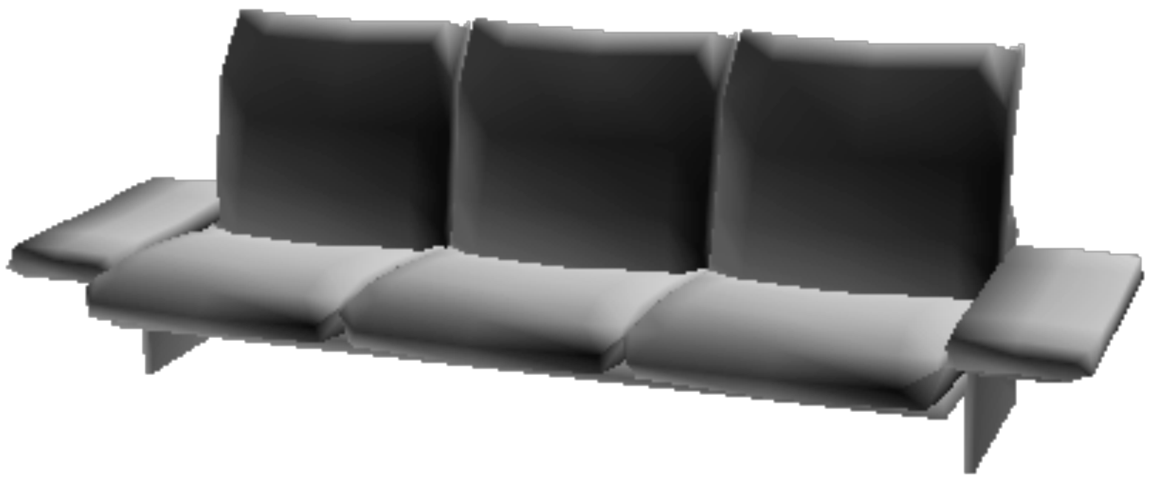
<u>Note</u>: The values of α and β only need to be updated as we move to the next scan-line. The value of φ needs to be updated as we advance along the scan-line.

Gouraud Shading

- Produces smoothly shaded polygonal mesh
  - Smooth shading over adjacent polygons
  - Need fine mesh to capture subtle lighting effects



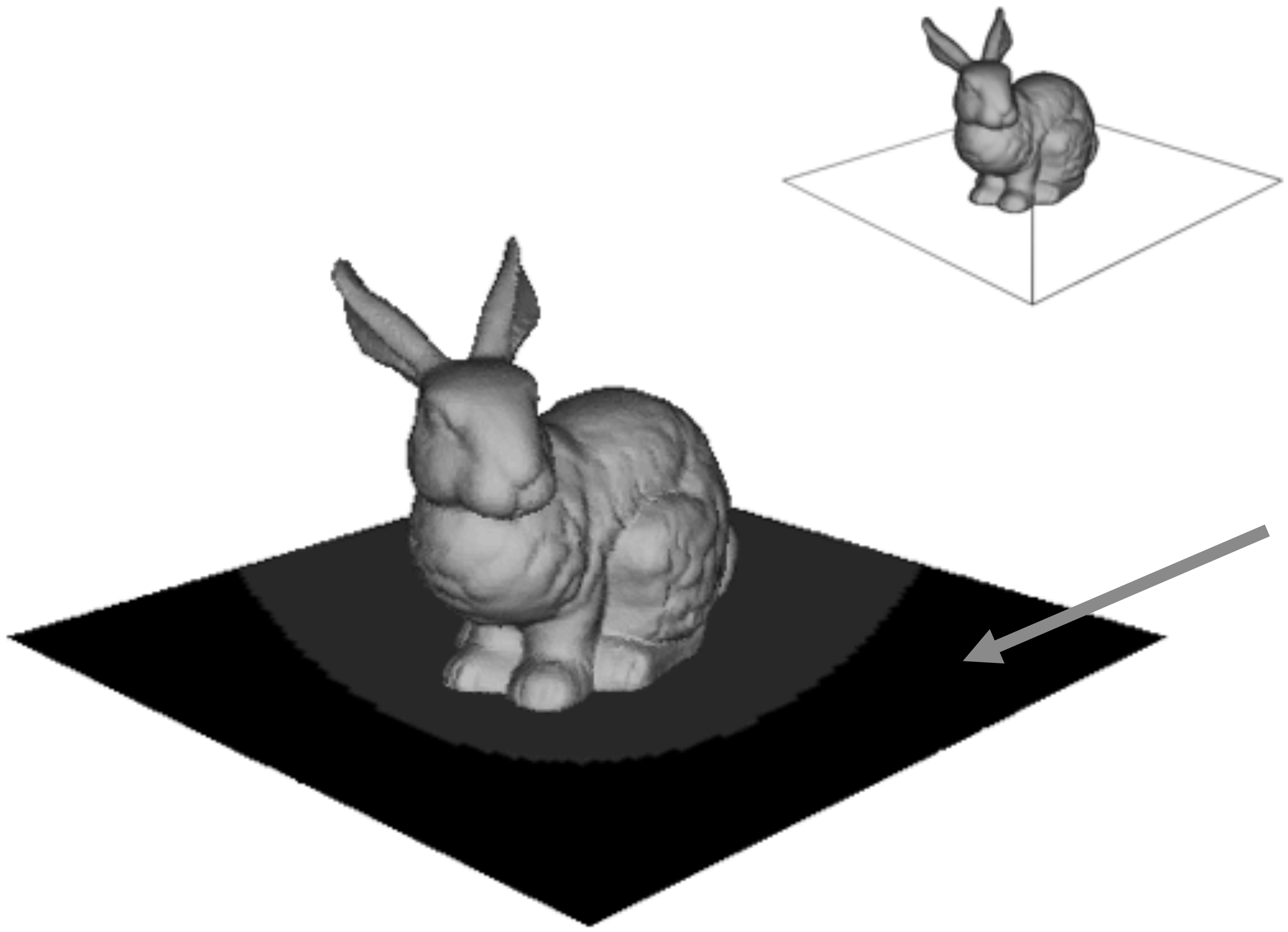
Flat Shading



Gouraud Shading

Gouraud Shading

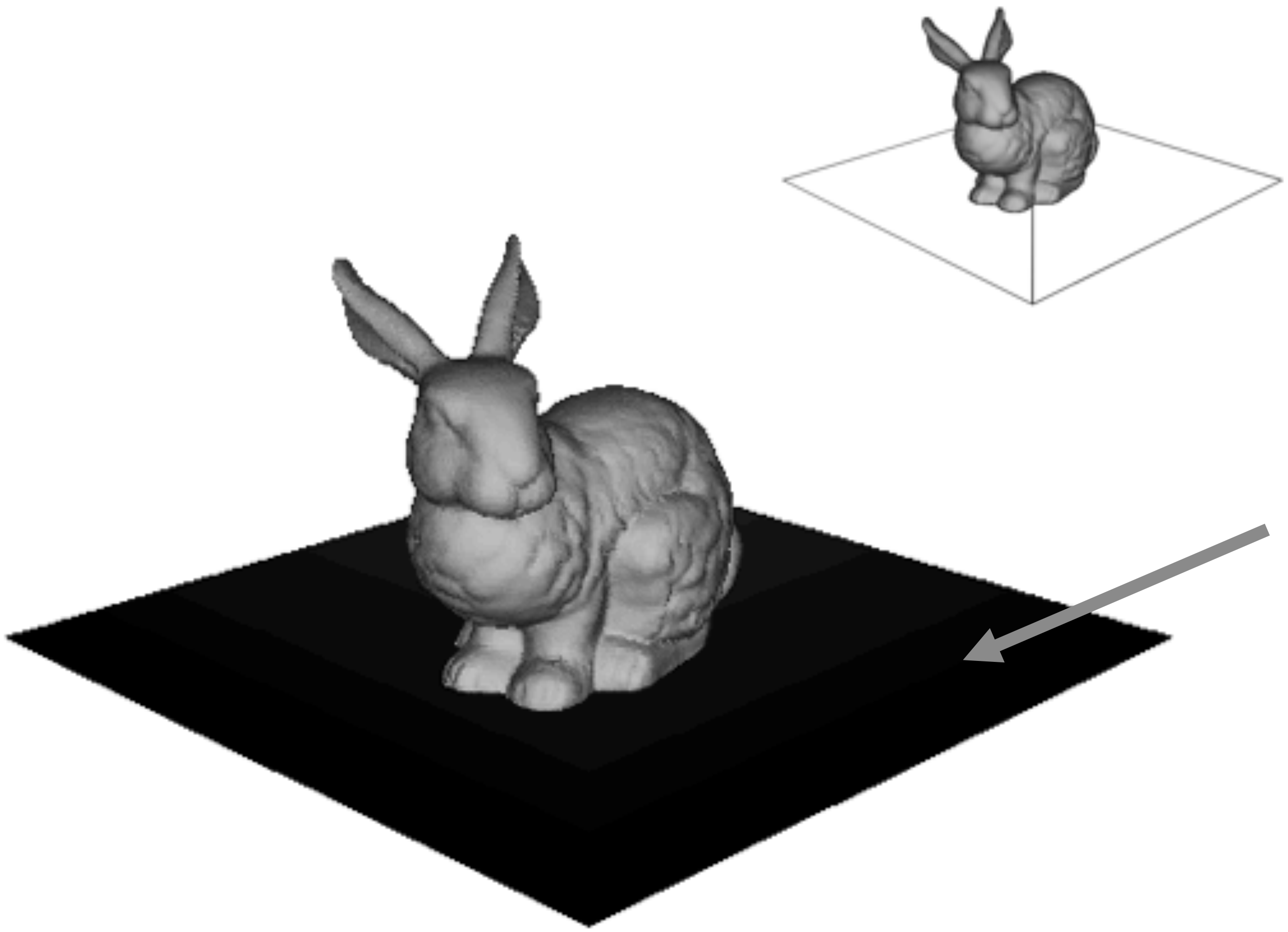
- Produces smoothly shaded polygonal mesh
  - Smooth shading over adjacent polygons
  - Need fine mesh to capture subtle lighting effects



What happens with large polygon & spotlight?

Gouraud Shading

- Produces smoothly shaded polygonal mesh
  - Smooth shading over adjacent polygons
  - Need fine mesh to capture subtle lighting effects



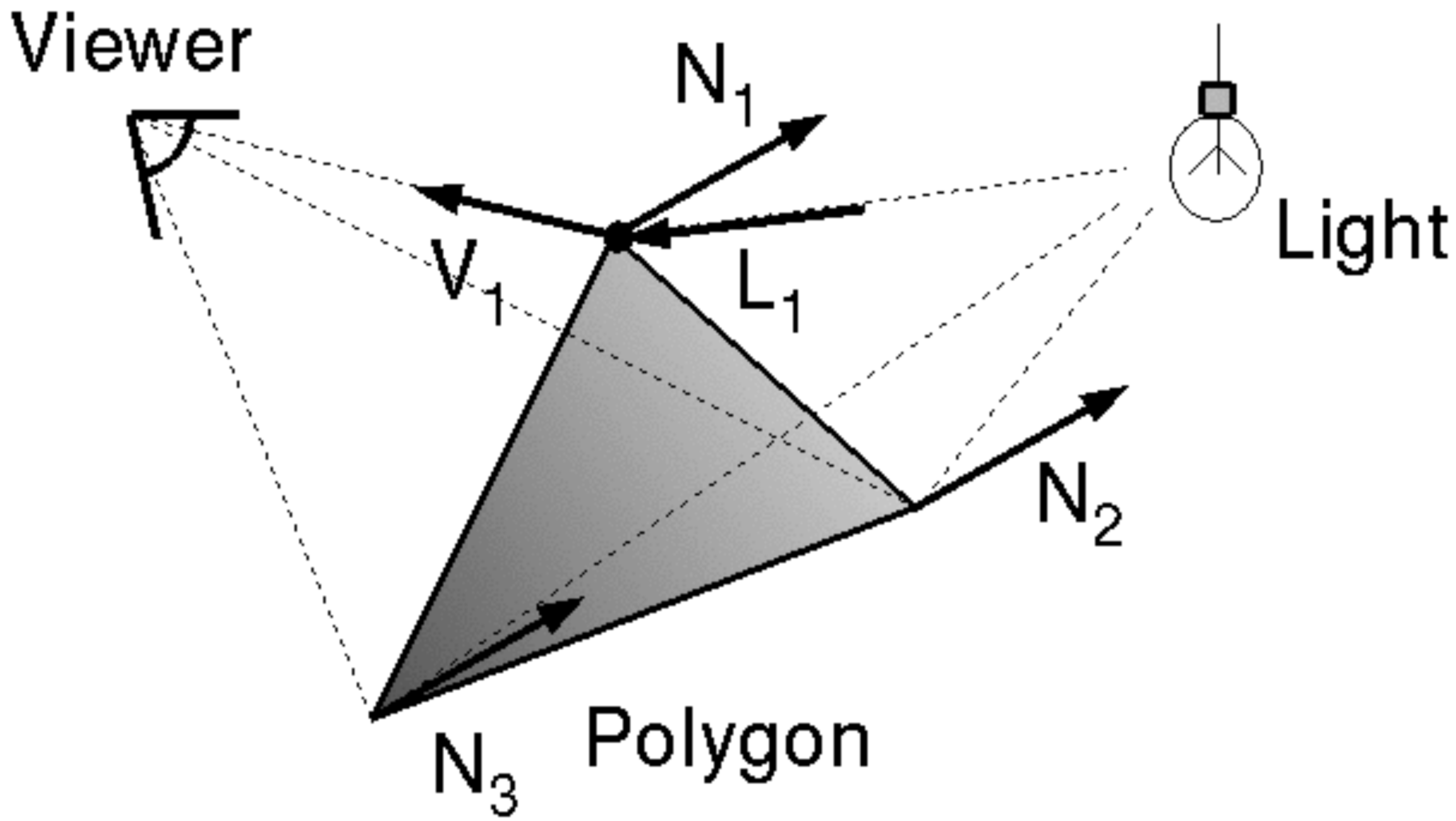
What happens with large polygon & spotlight?

Polygon Shading Algorithms

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

Phong Shading

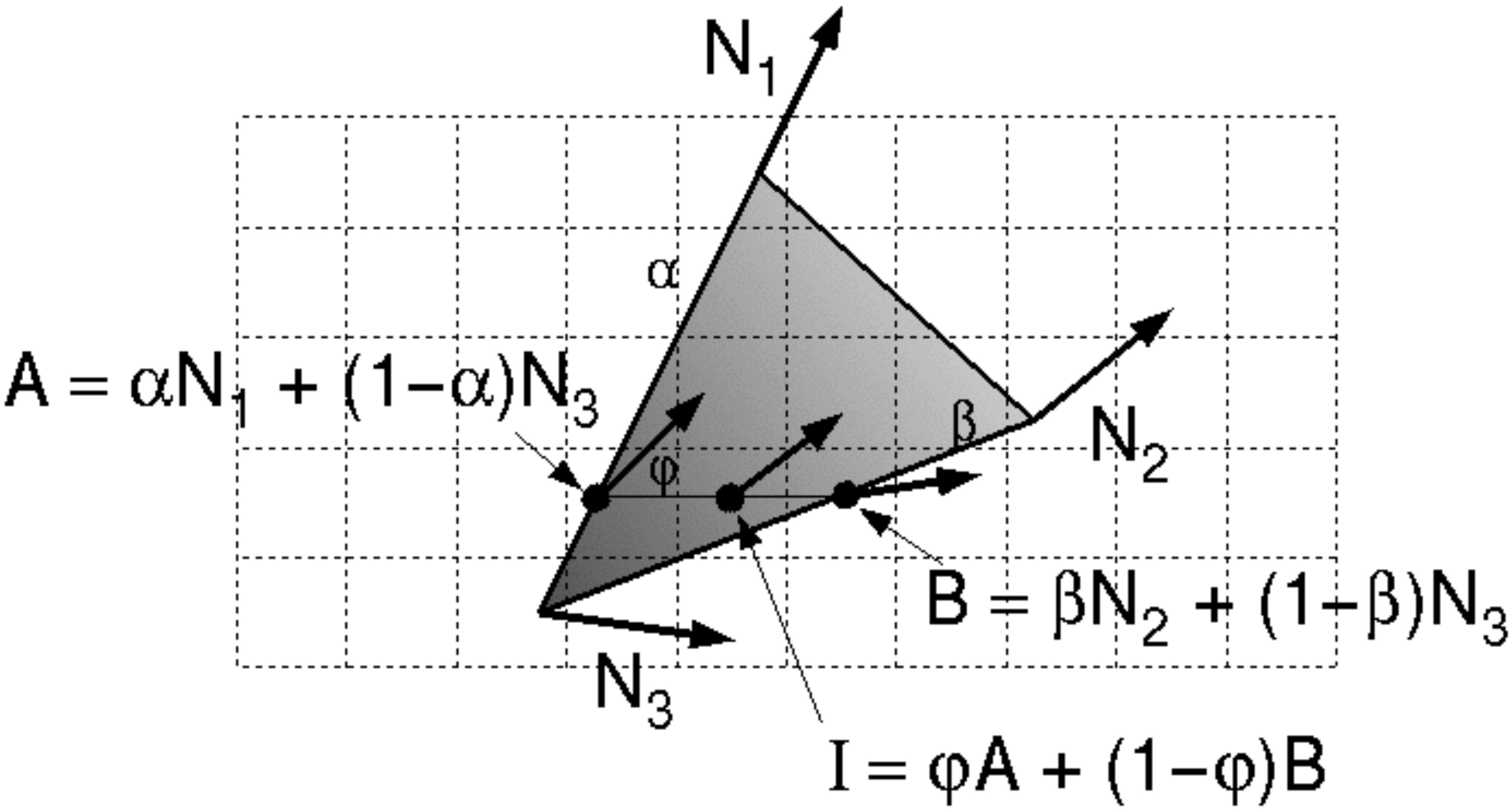
- One lighting calculation per pixel
  - Approximate surface normals for points inside polygons by bilinear interpolation of normals from vertices



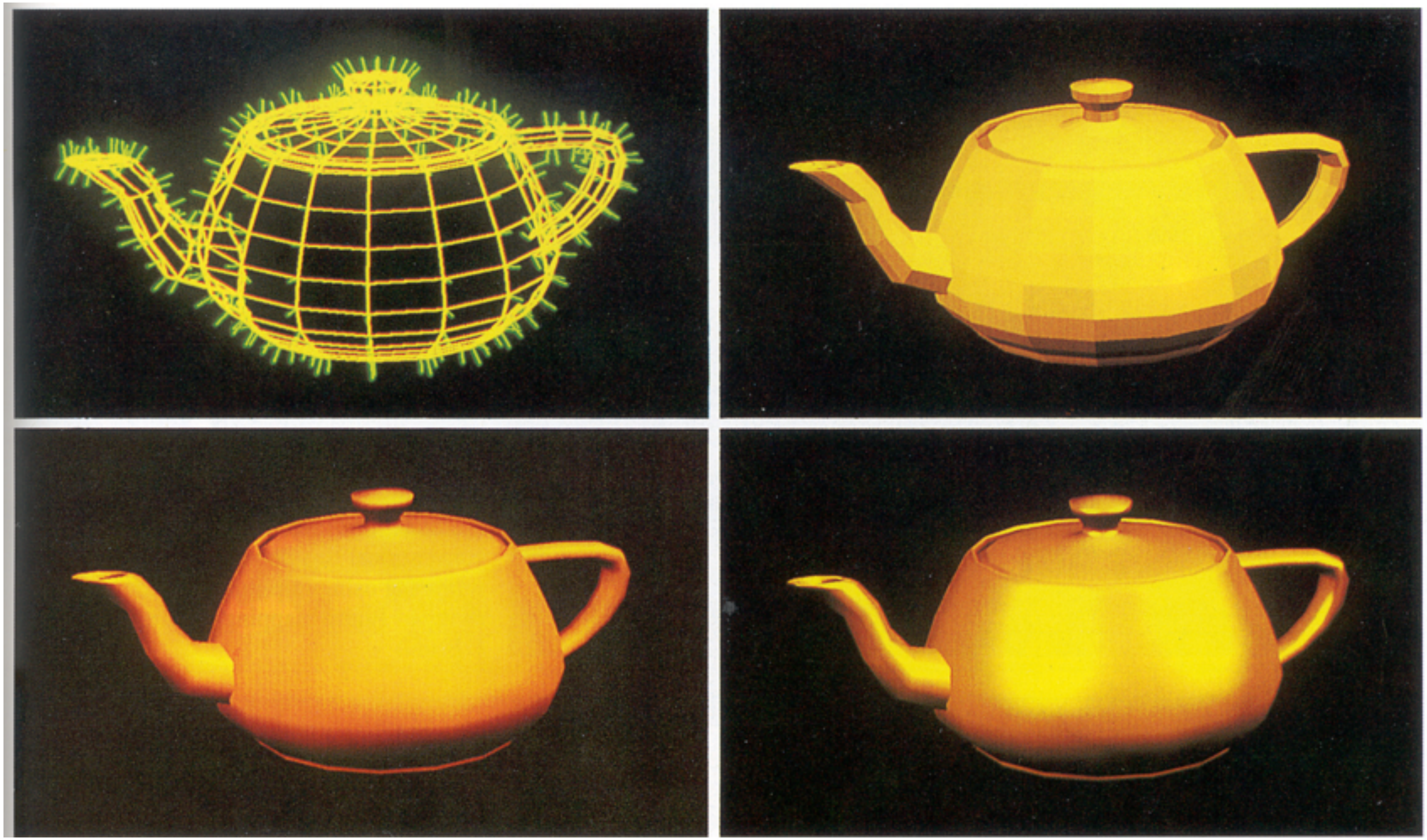
$$I = I_E + K_A I_{AL} + \sum_i (K_D (N \bullet L_i) I_i + K_S (V \bullet R_i)^n I_i)$$

Phong Shading

- Bilinearly interpolate surface normals at vertices down and across scan lines

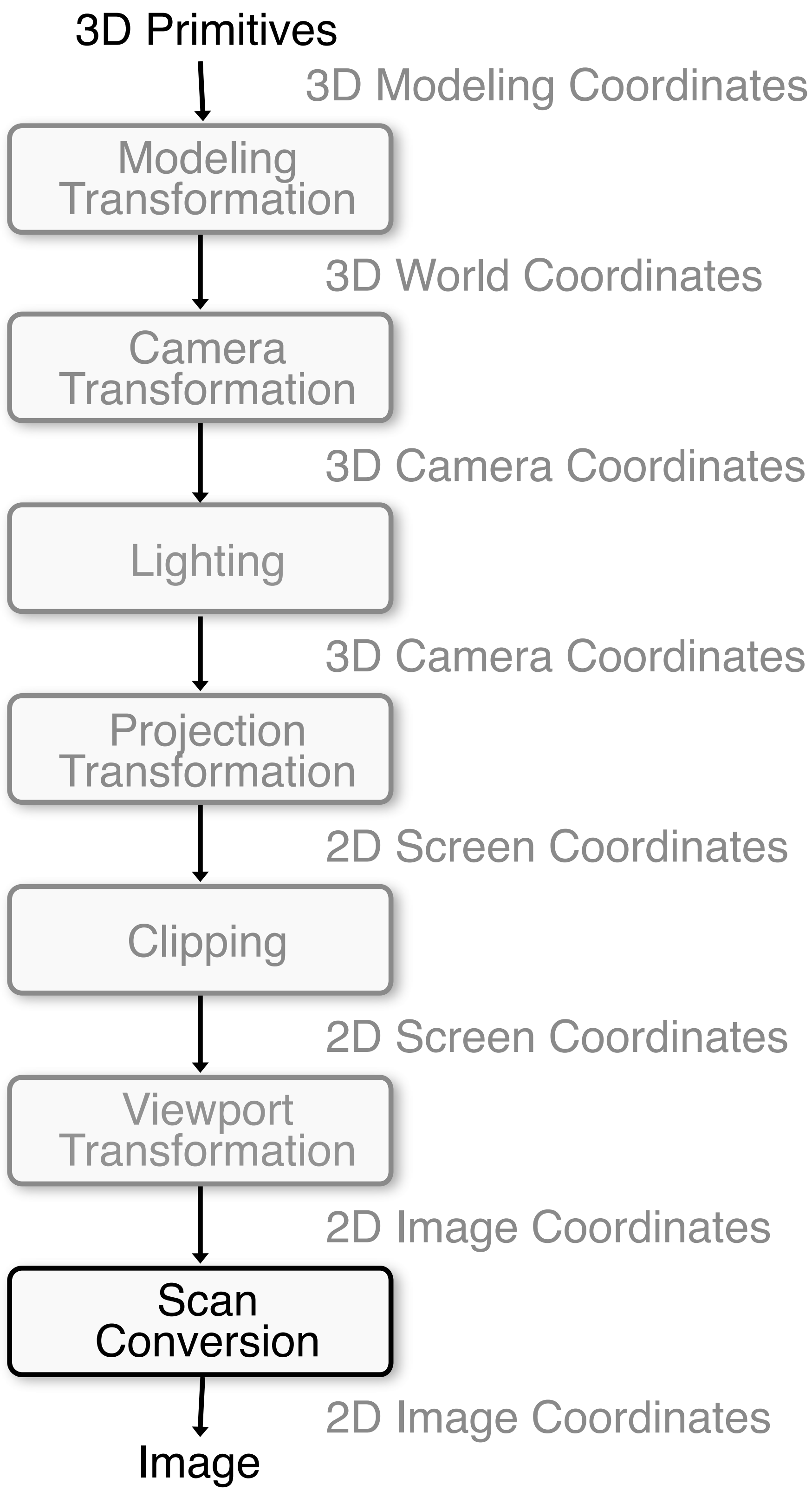


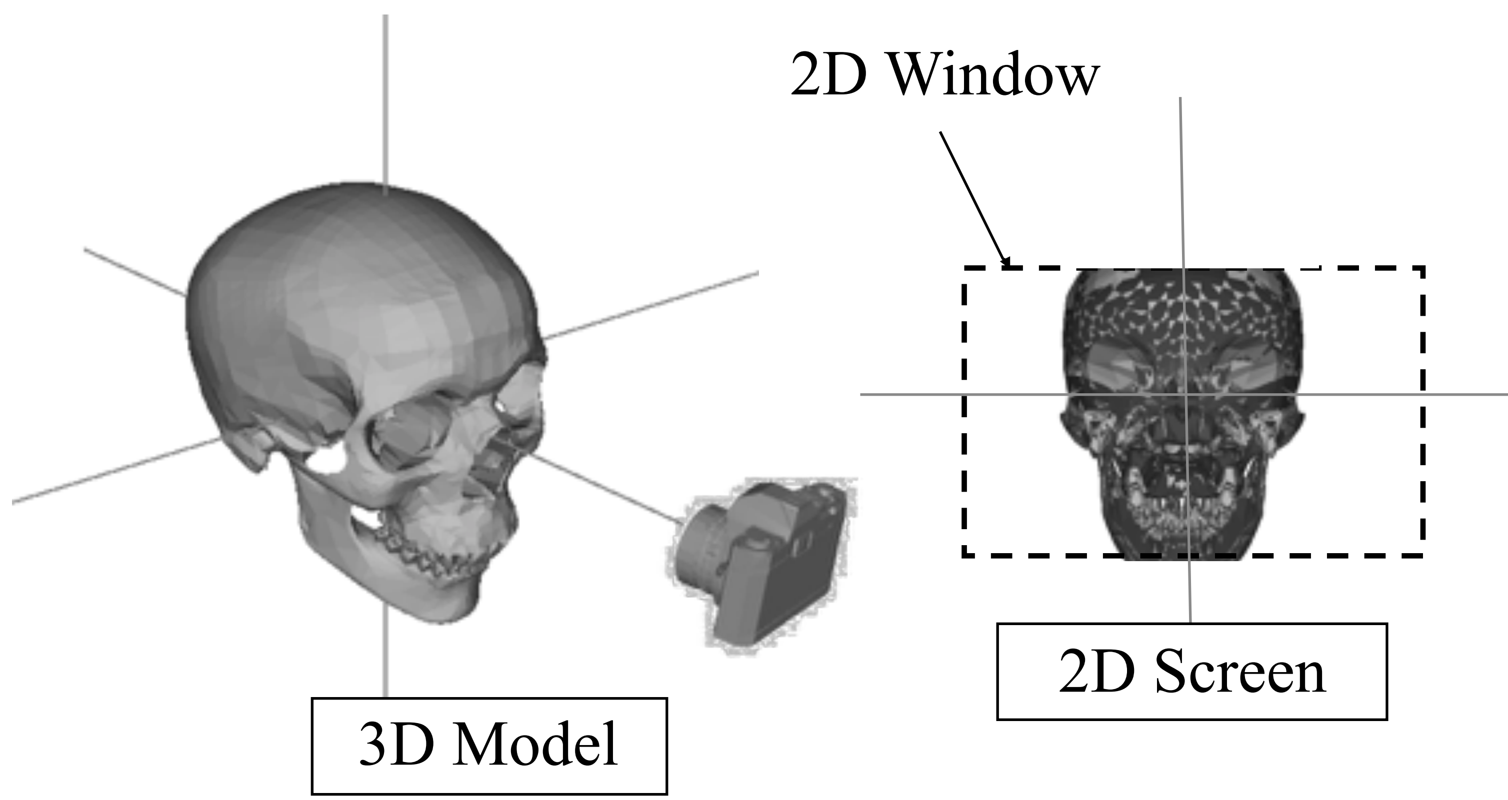
Polygon Shading Algorithms



Watt Plate 7

3D Rendering Pipeline (for direct illumination)





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

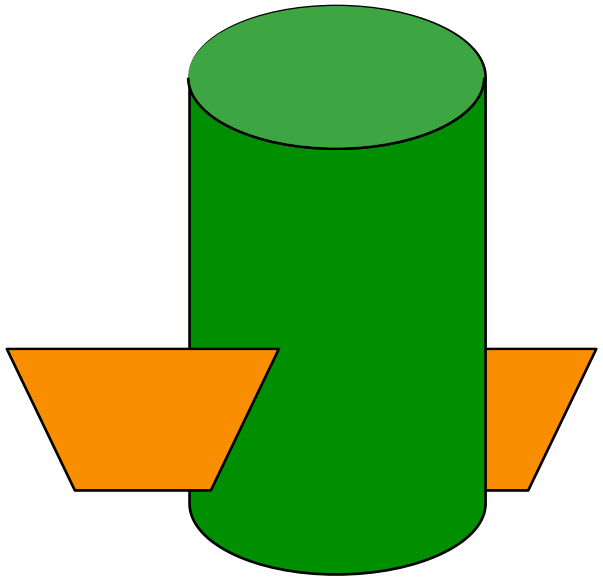
Hidden Surface Removal

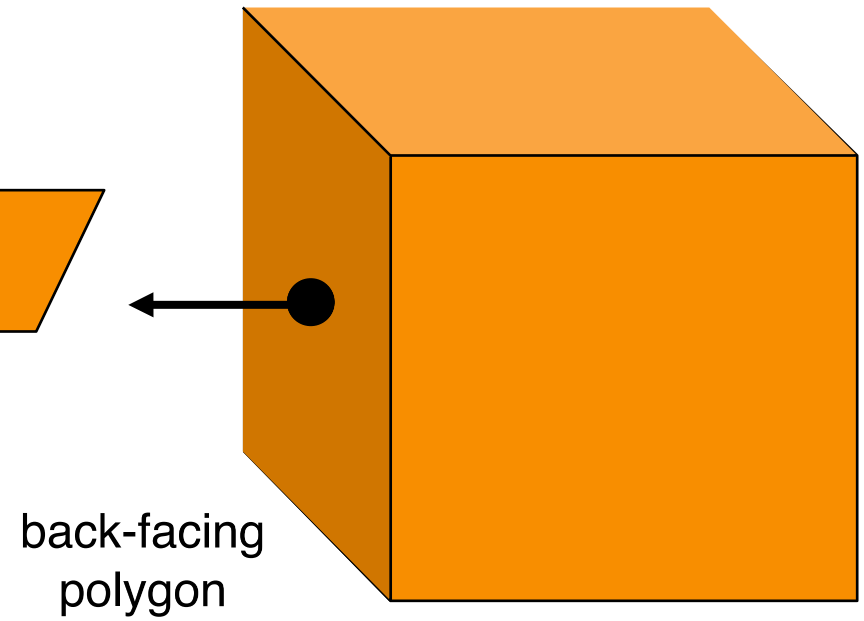
- Motivation
- Algorithms for HSR
  - Back-face detection
  - Depth sort
  - Ray casting
  - Z-buffer

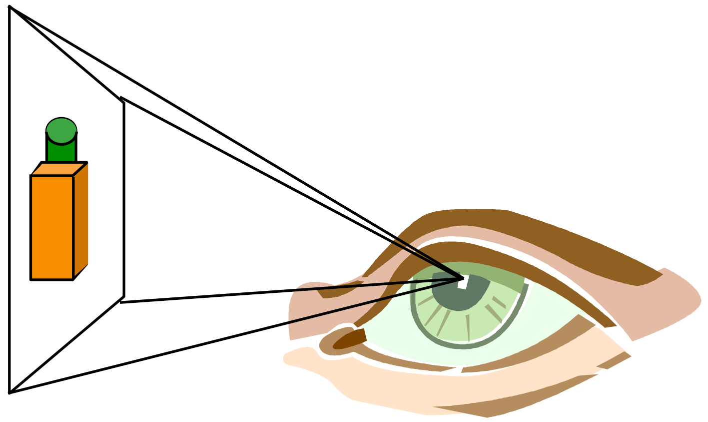
Motivation

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

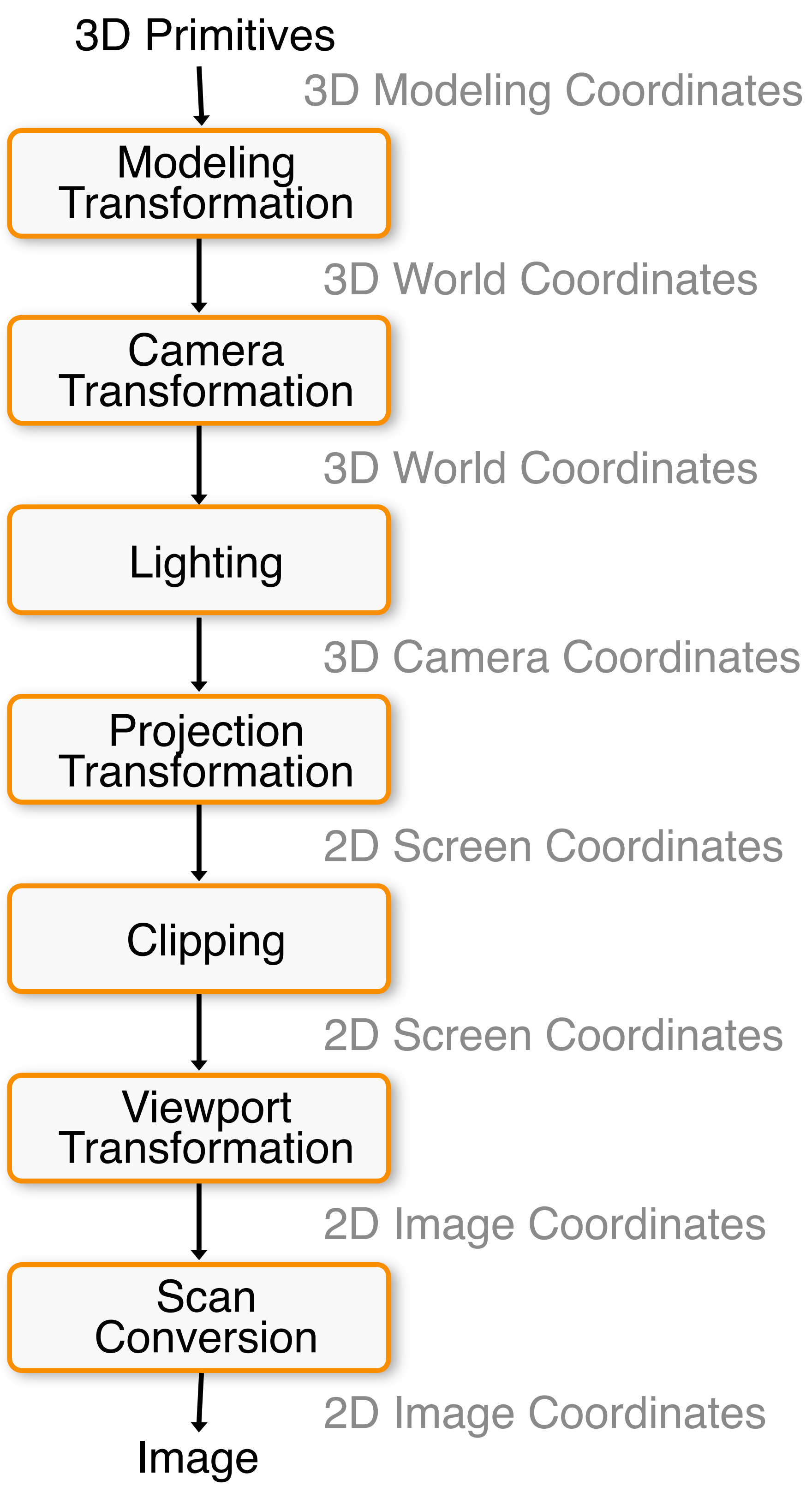
- Surfaces may be back-facing.
- Surfaces may intersect in 3D.
- Surfaces may intersect in the image plane.







3D Rendering Pipeline



Somewhere in here we have to decide which objects are visible, and which objects are hidden.

Overview

- Motivation
- Algorithms for HSR
  - Back-face detection
  - BSP-Trees
  - Ray casting
  - Z-buffer

Visibility algorithms

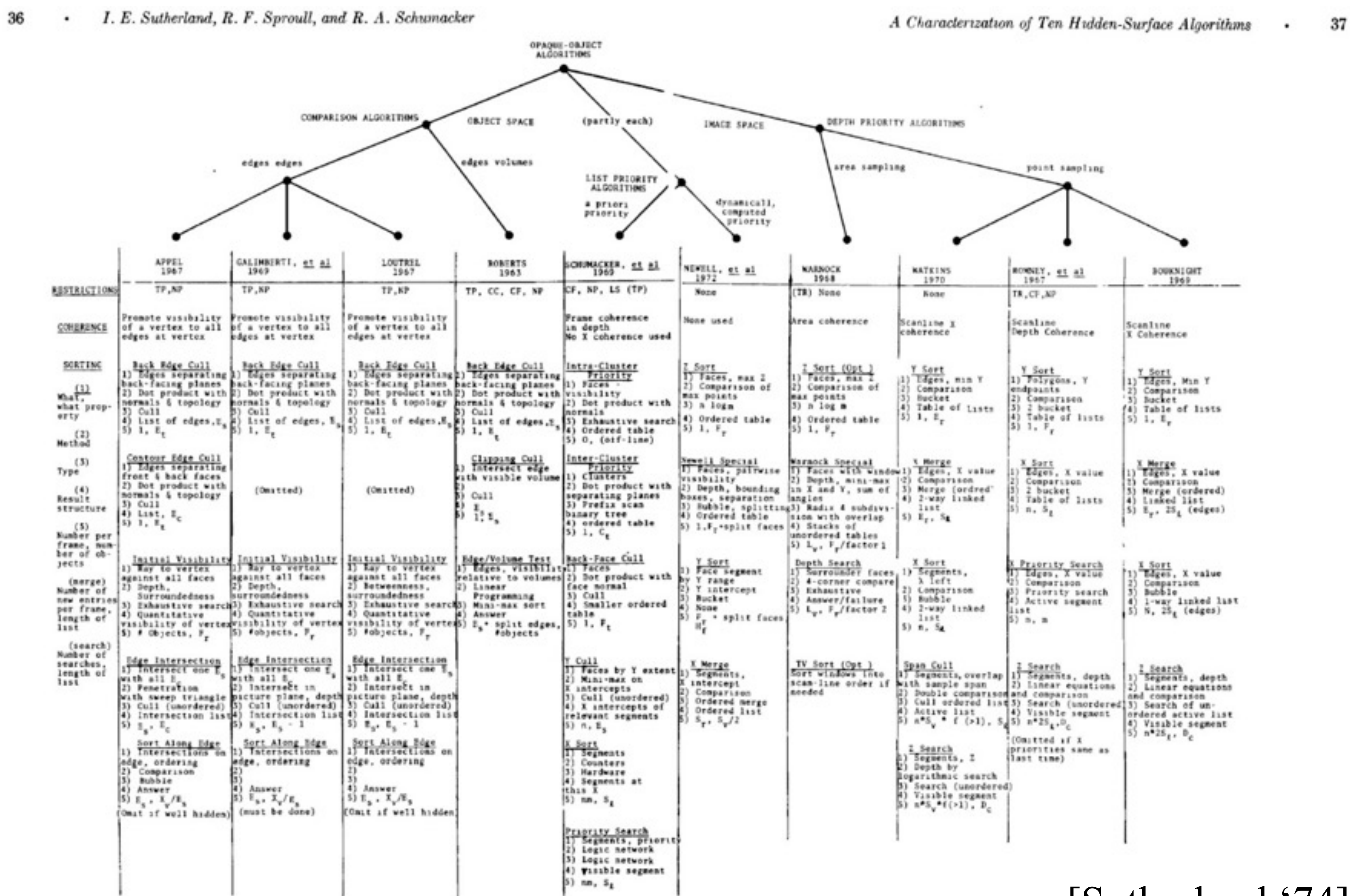
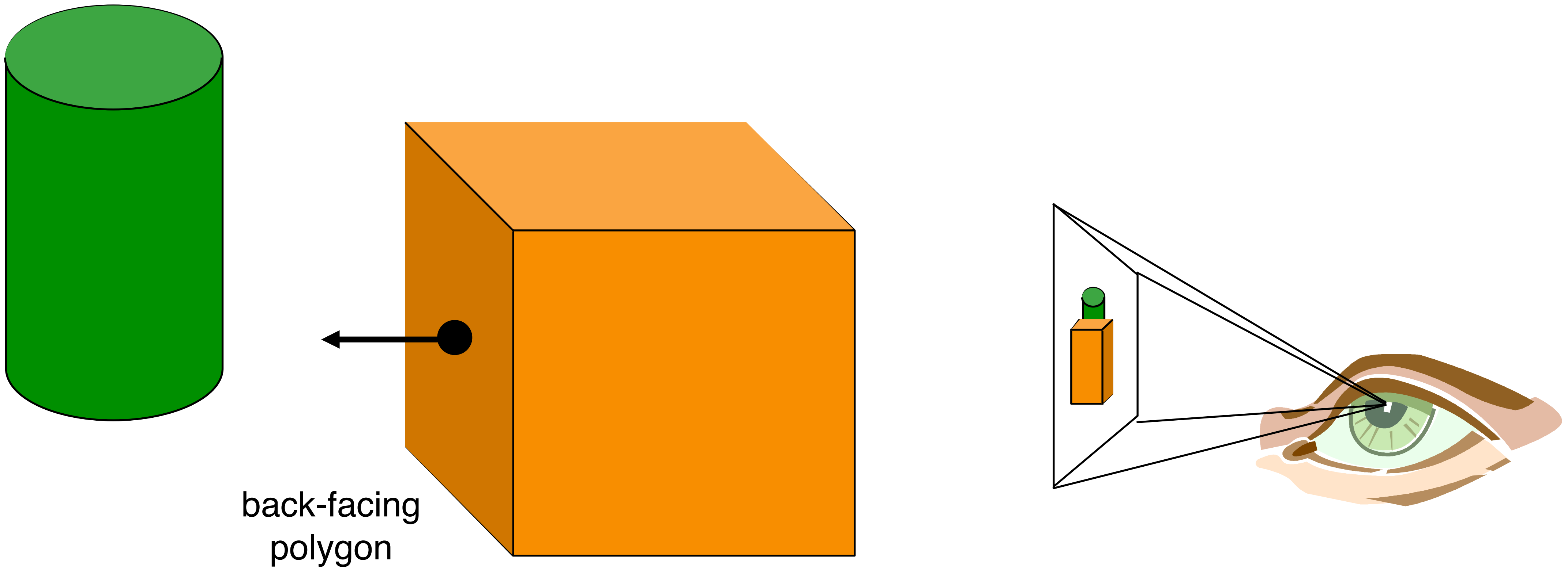


Figure 29. Characterization of ten opaque-object algorithms b. Comparison of the algorithms.

[Sutherland ‘74]

Back-face detection

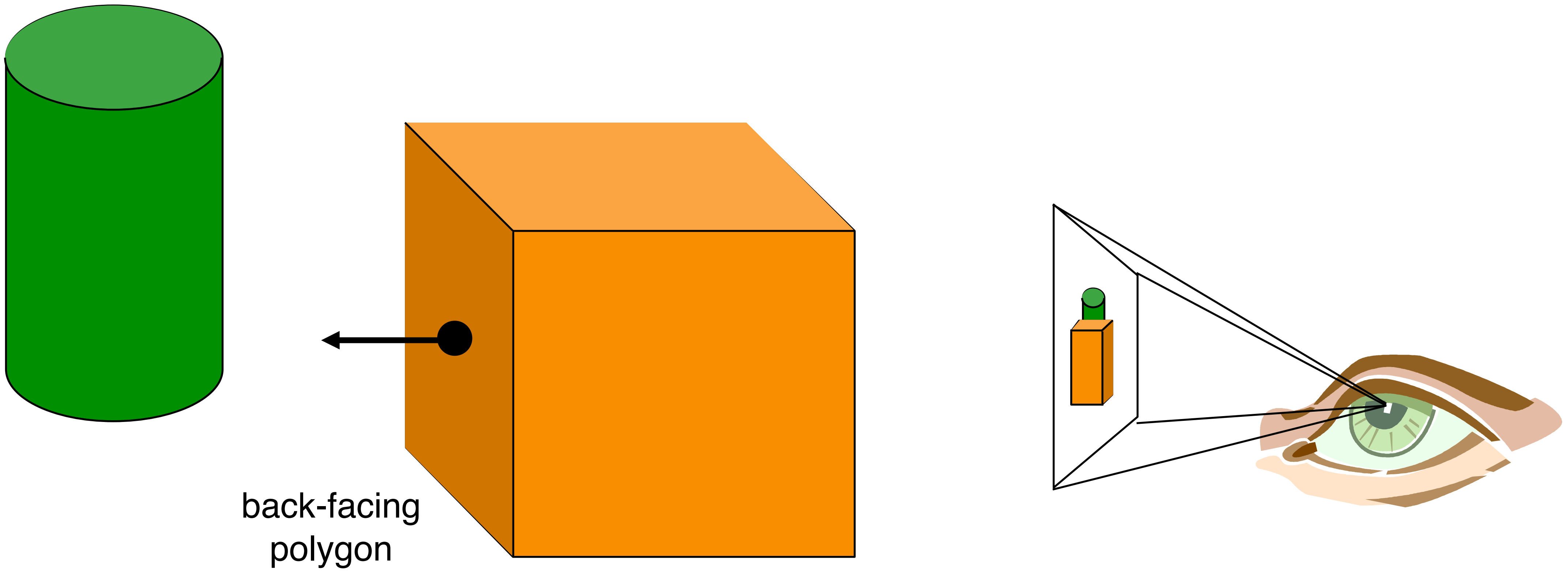
Q: How do we test for back-facing polygons?



Back-face detection

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

A: Dot product of the normal and view directions.

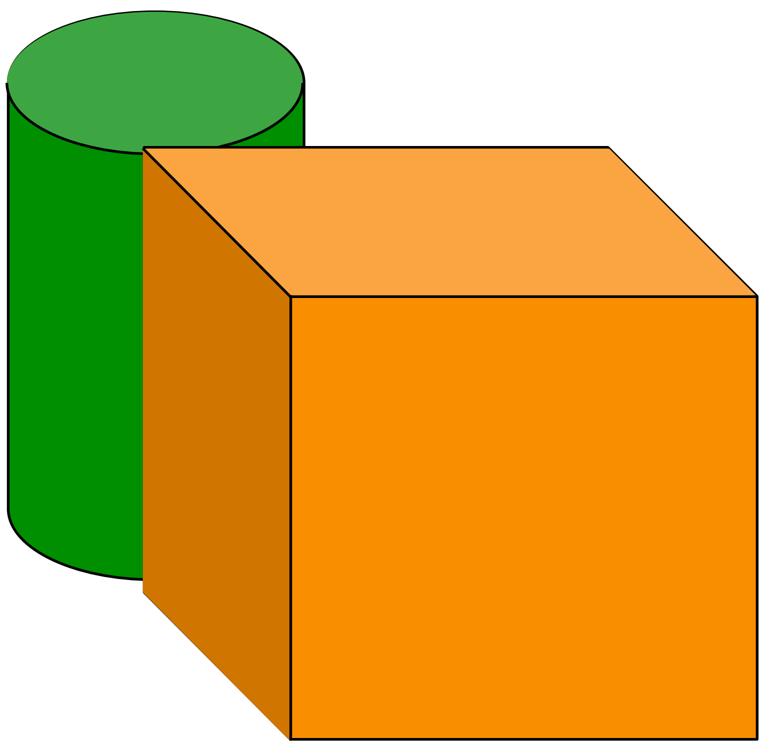


If V·N > 0, then polygon is back-facing

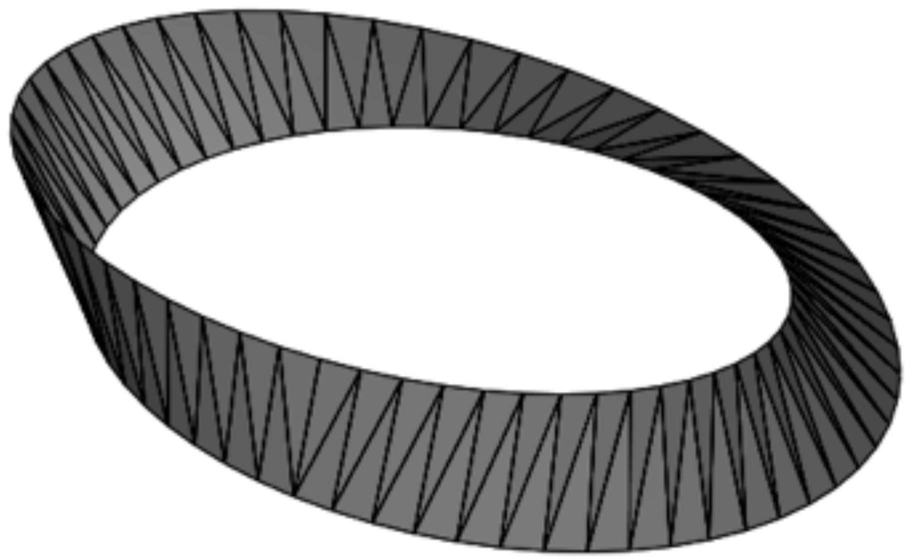
Back-face detection

This method breaks down for:

- Overlapping primitives
- Non-solid models and/or models without a well defined orientation.



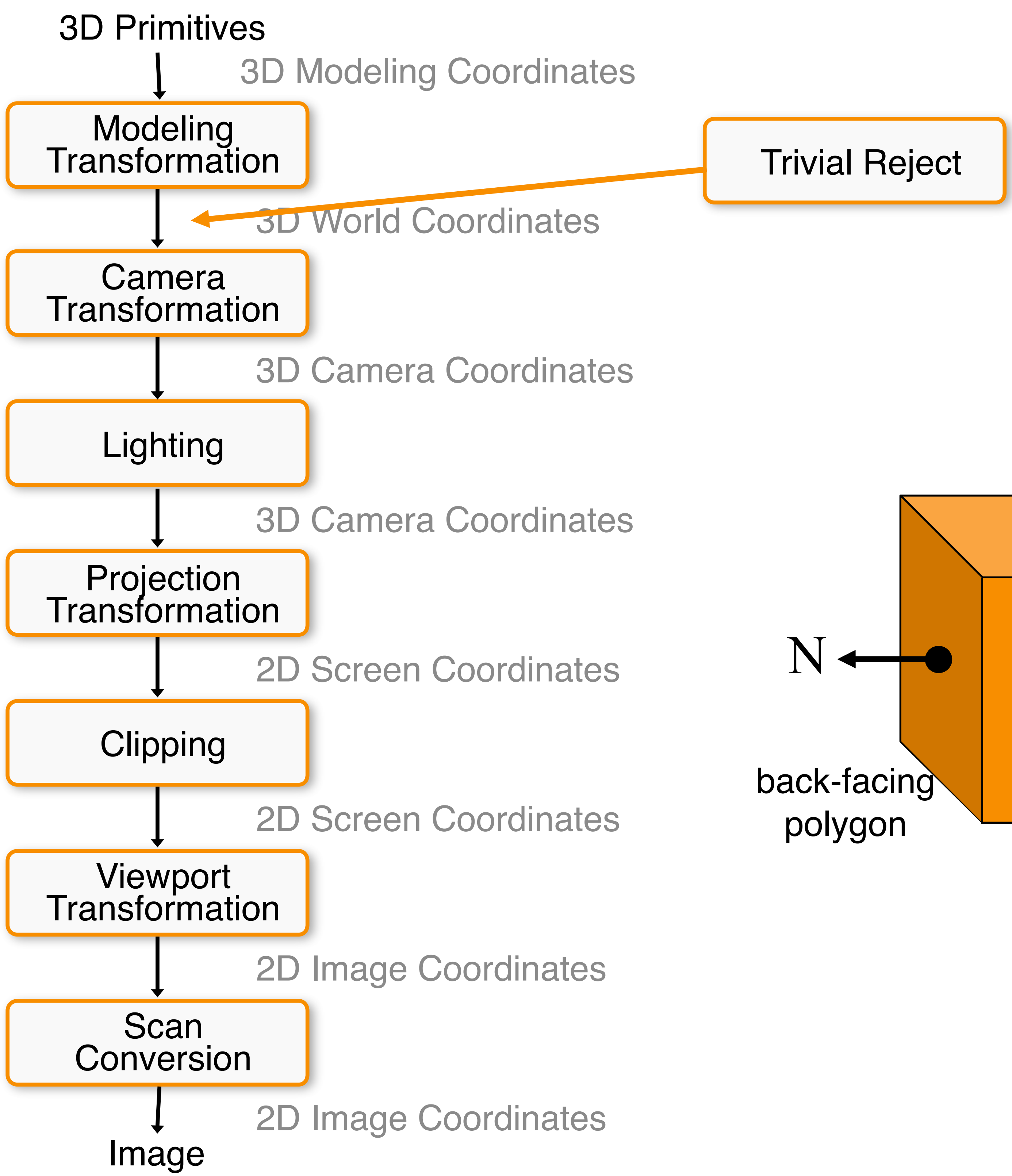
Overlapping Objects



Non-Solid Objects

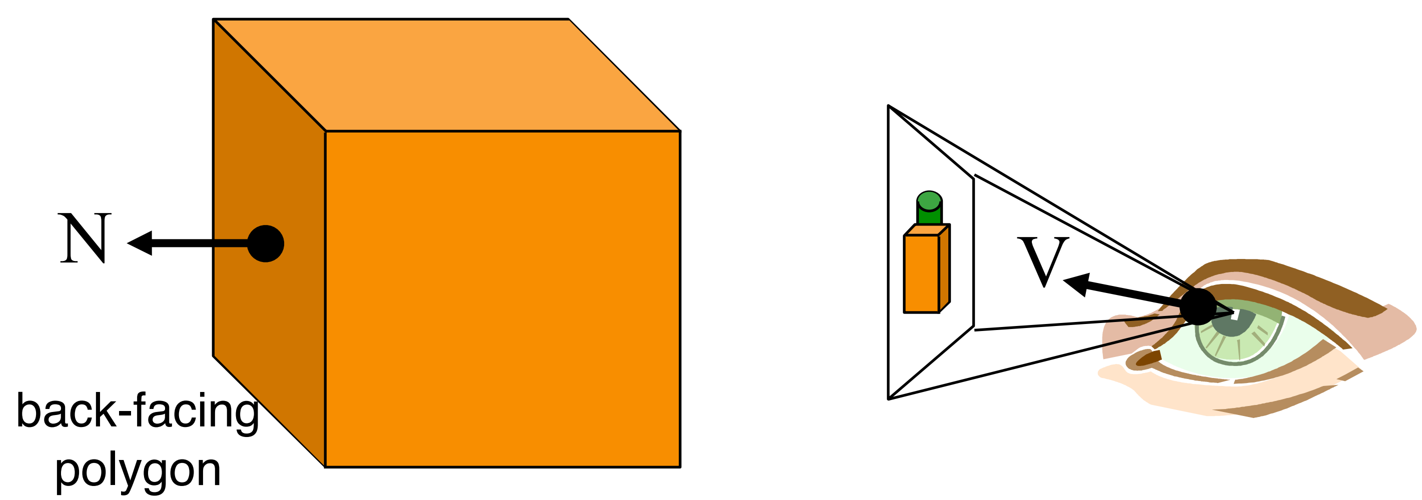
In general, back-face removal expected to remove ≈ half of polygon surfaces from further visibility tests

3D Rendering Pipeline

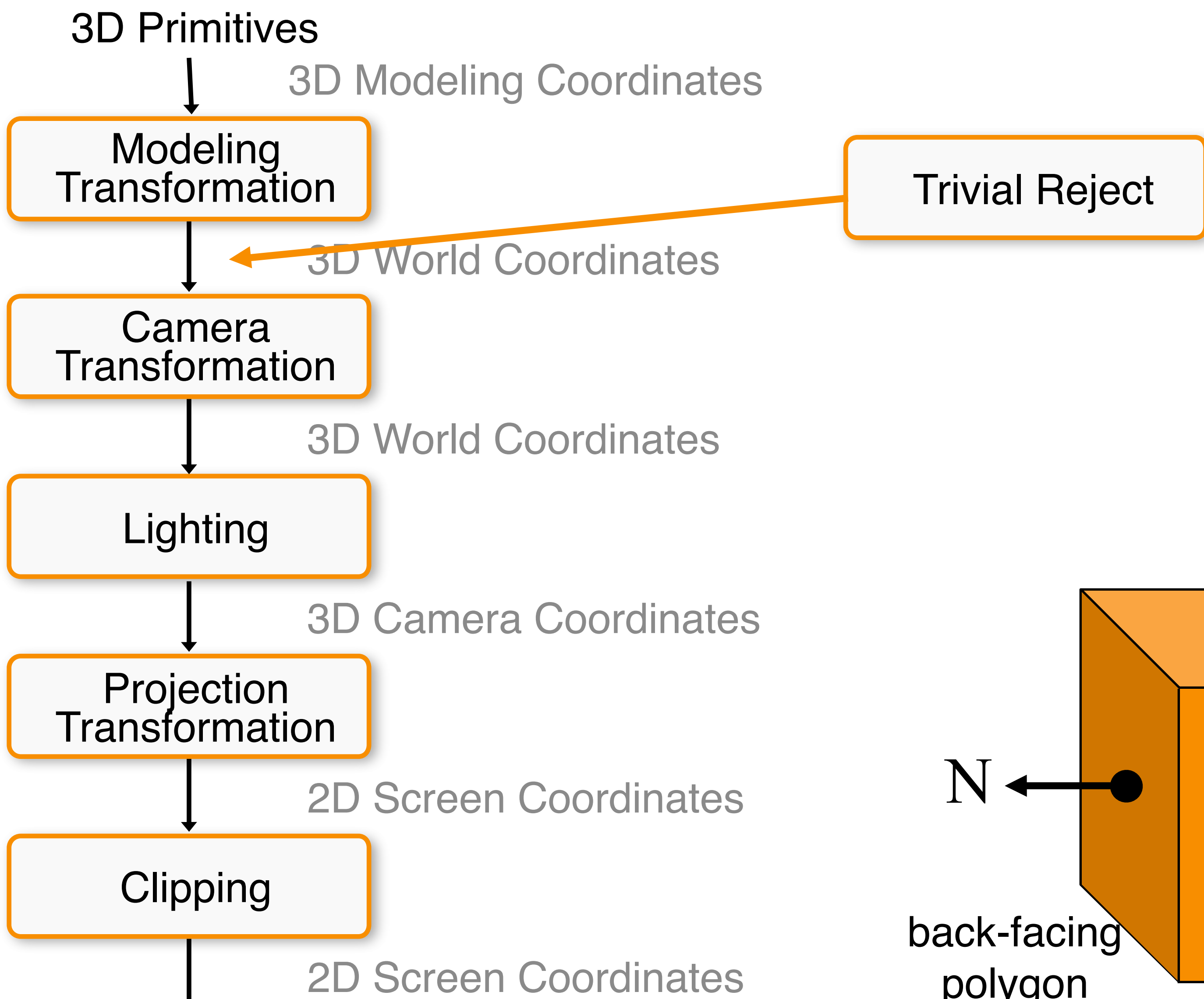


A polygon is backfacing if

$$V \bullet N > 0$$

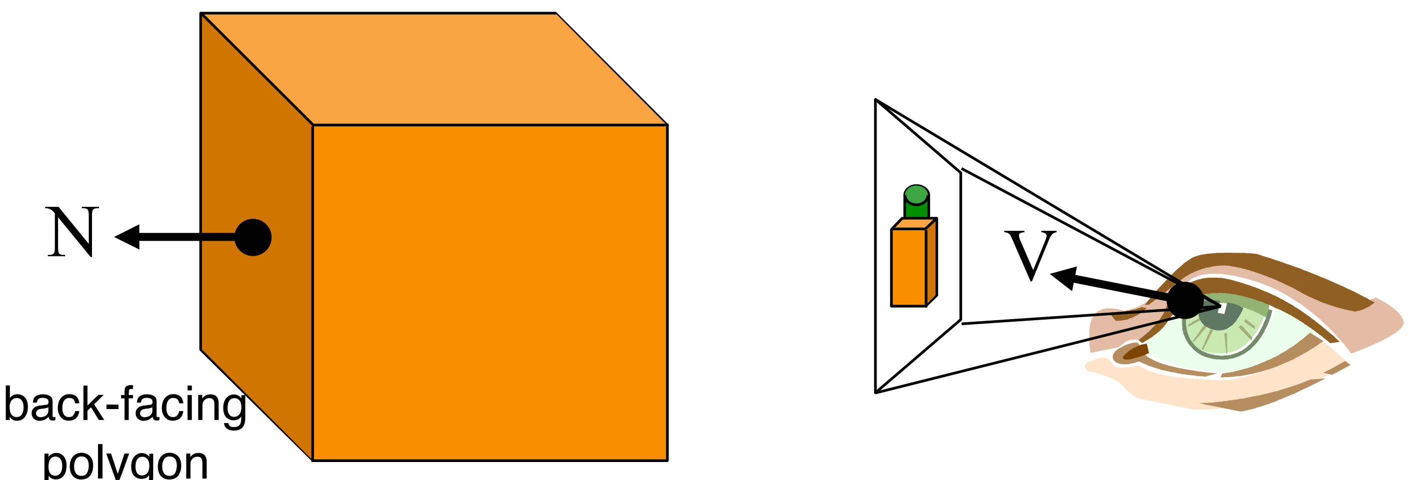


3D Rendering Pipeline



A polygon is backfacing if

$$V \bullet N > 0$$



Note: When your graphics card does this, it does not use the normals you provide at the vertices.
Instead it uses the cross-product of the triangle vertices, so make sure that the ordering of the vertices is consistent (e.g. CCW)

Ideal Solution

Painter’s Algorithm:

- Sort primitives front to back and draw the back ones first, over-writing pixel values with information from the front primitives as they are processed.

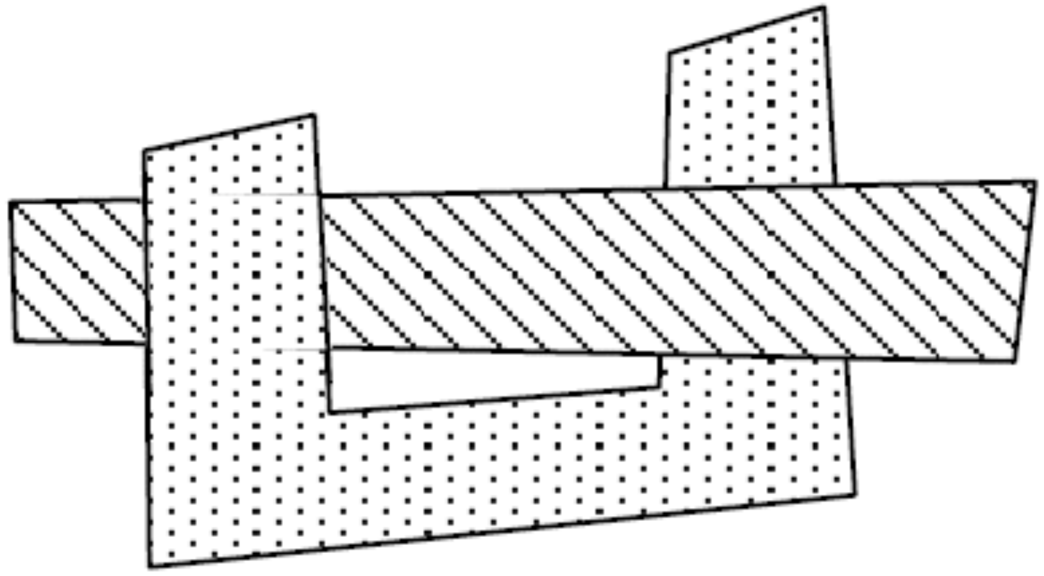
Ideal Solution

Painter's Algorithm:

- Sort primitives front to back and draw the back ones first, over-writing pixel values with information from the front primitives as they are processed.

Problem:

- You can't always sort the primitives.



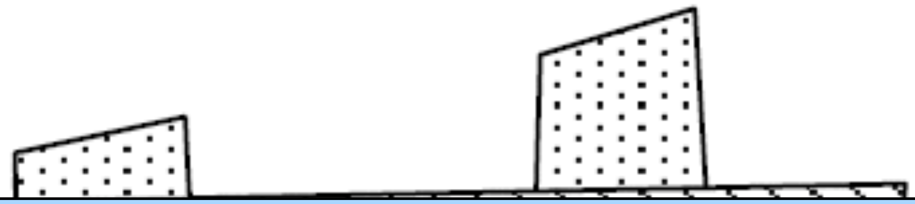
Ideal Solution

Painter's Algorithm:

- Sort primitives front to back and draw the back ones first, over-writing pixel values with information from the front primitives as they are processed.

Problem:

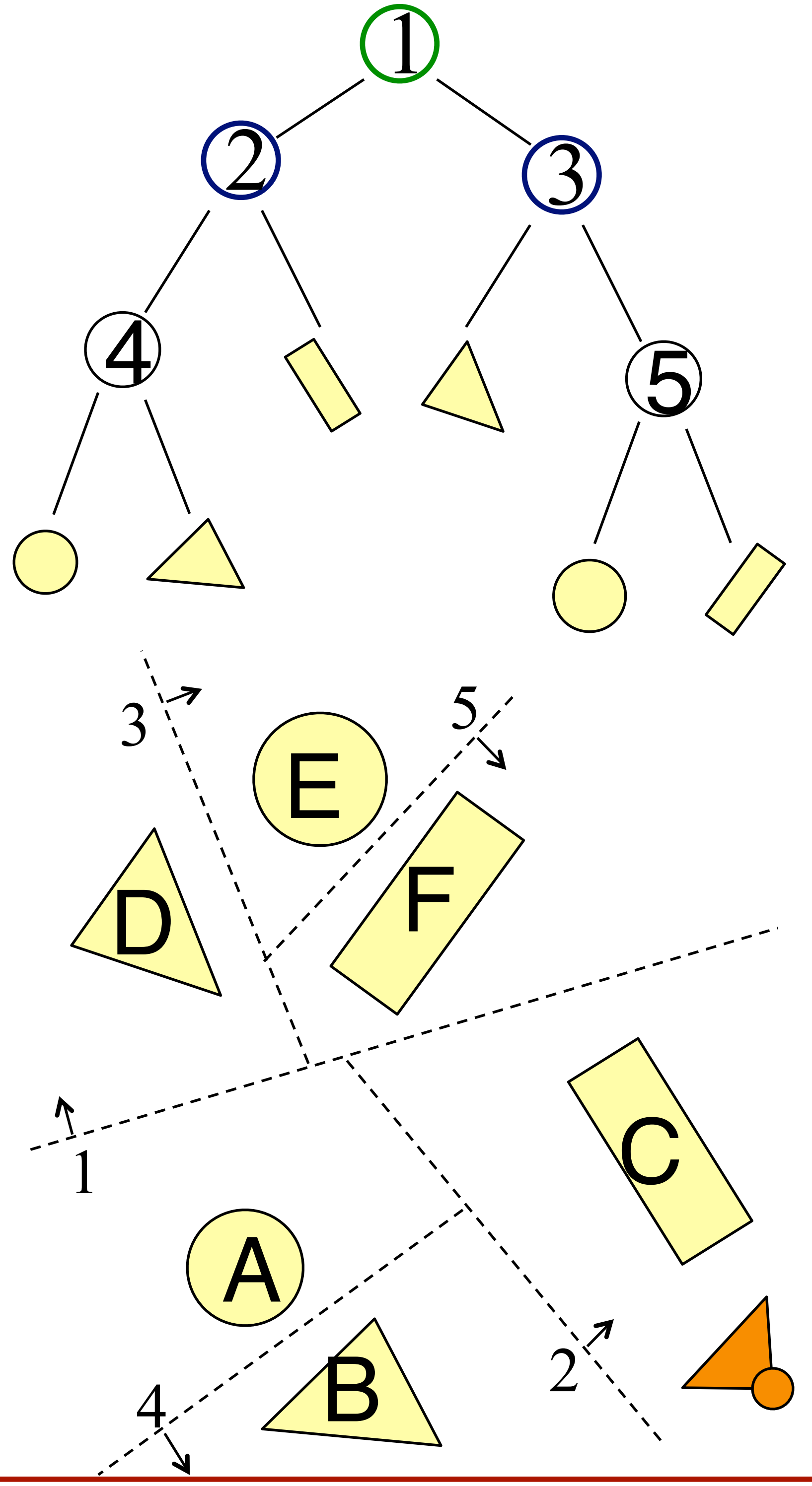
- You can't always sort the primitives.



However, in some cases you can sort the primitives – e.g. if all the vertices of one primitive are in front of all the vertices of the second.

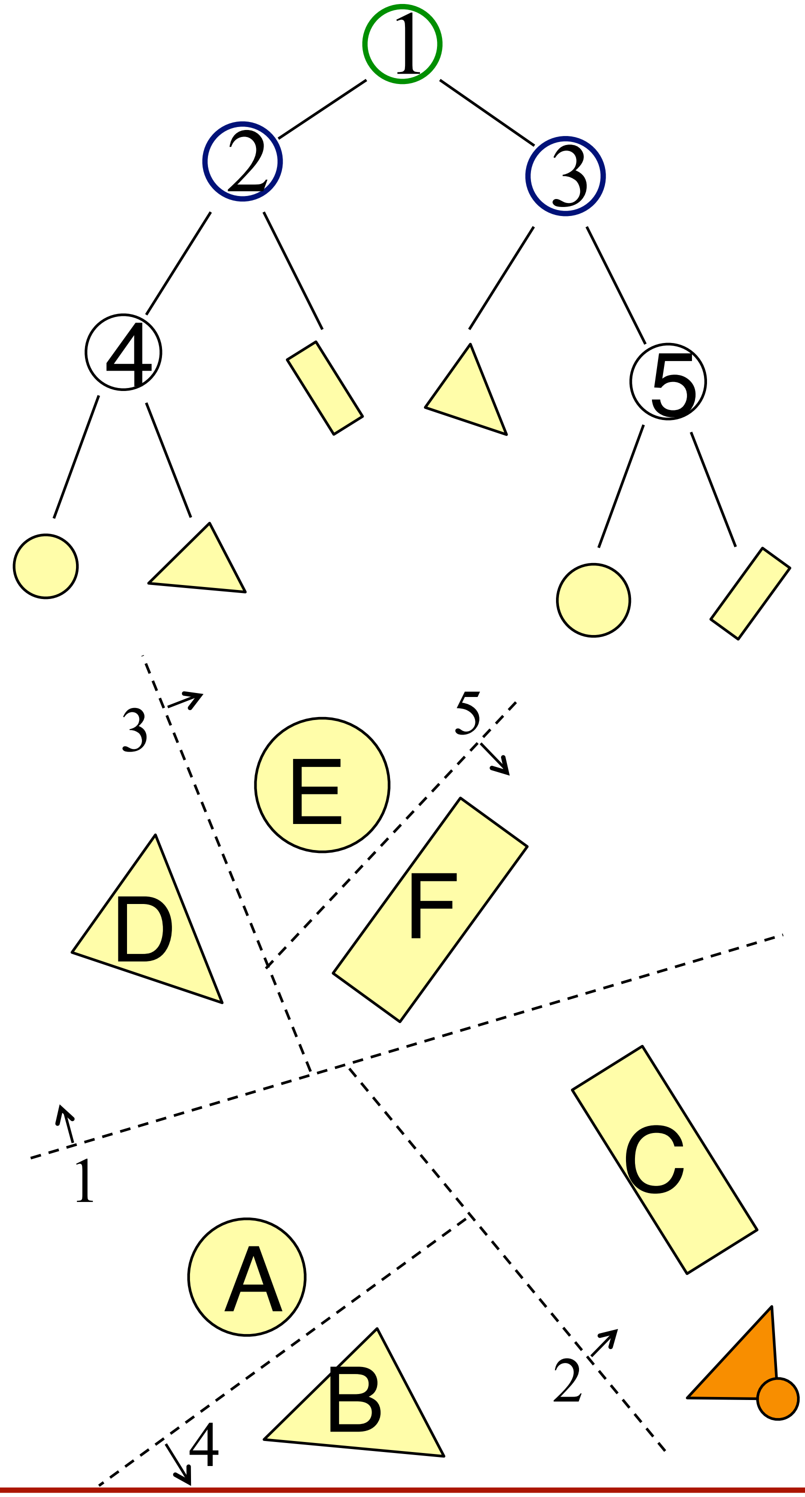
BSP-Tree Rendering (Object Precision)

- BSP-Trees recursively partition space by planes
  - Given two primitives on either side of a plane, the one on the opposite side from the camera will always be further away.
  - Draw the further side first, and then draw the closer one



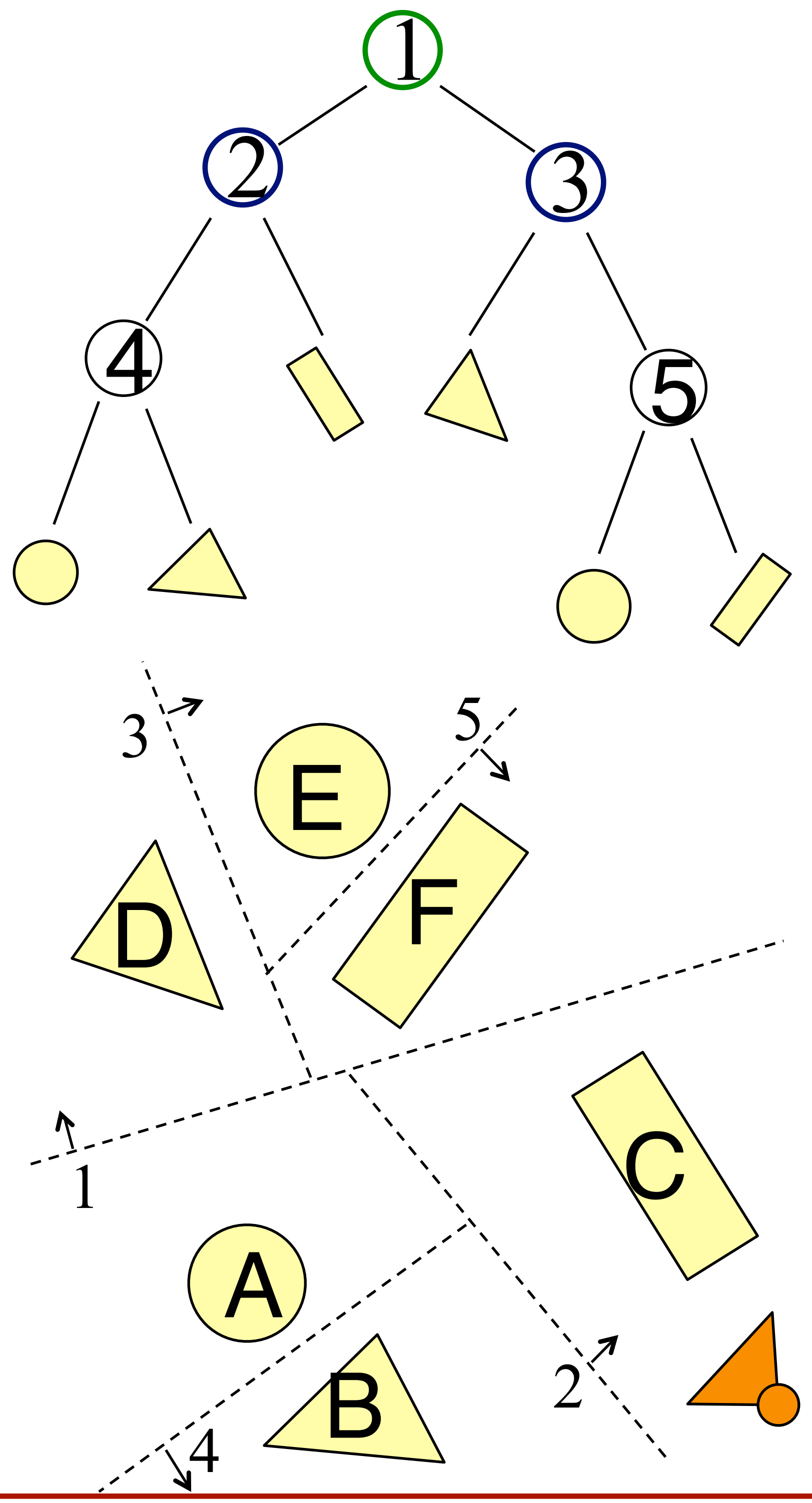
BSP-Tree Rendering (Object Precision)

- Draw further half first, then the closer one.
  - Draw right side of **1**
  - Draw left side of **1**



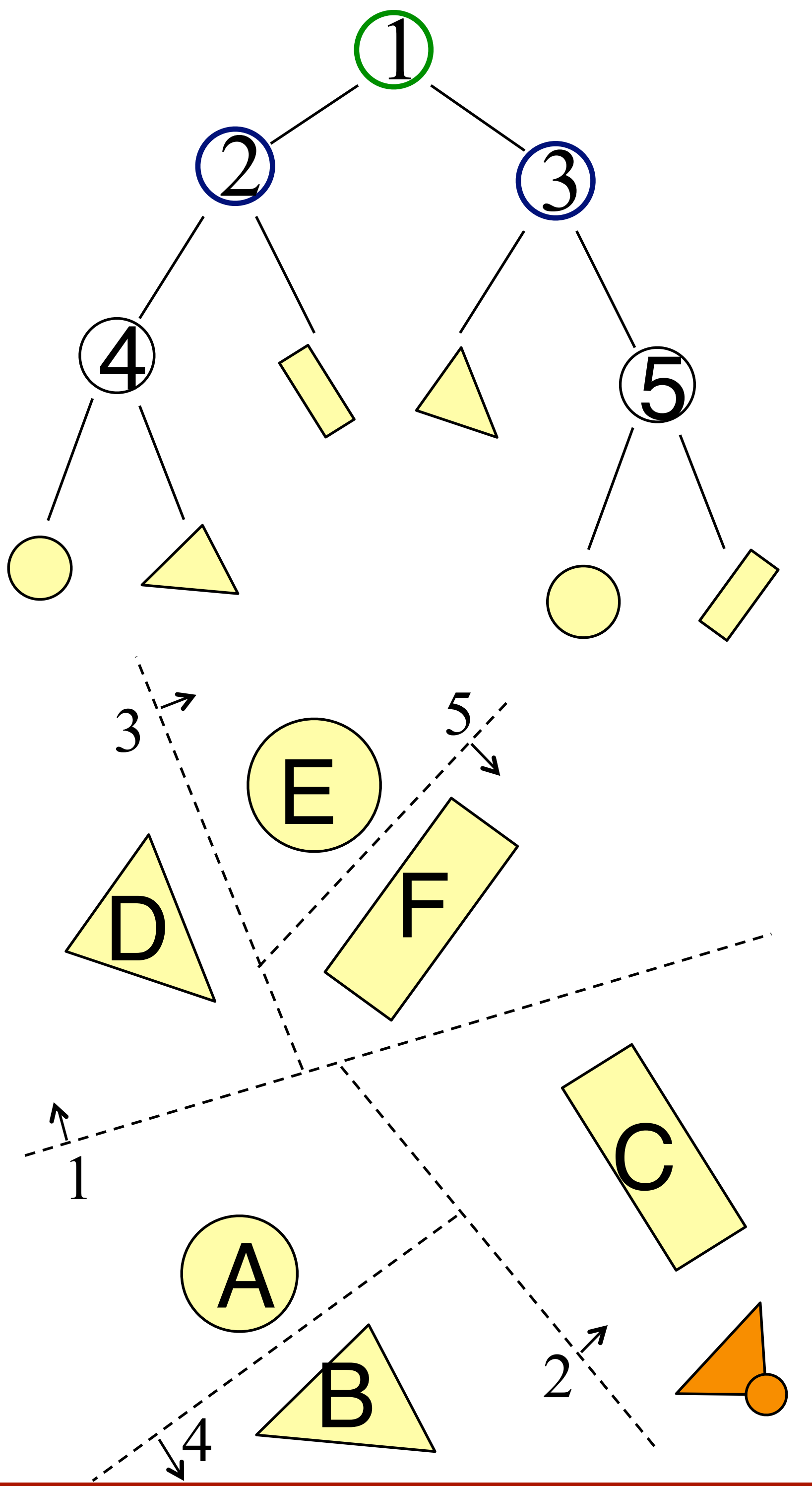
BSP-Tree Rendering (Object Precision)

- Draw further half first, then the closer one.
  - Draw right side of **1**
    - Draw left side of **3**
    - Draw right side of **3**
  - Draw left side of **1**



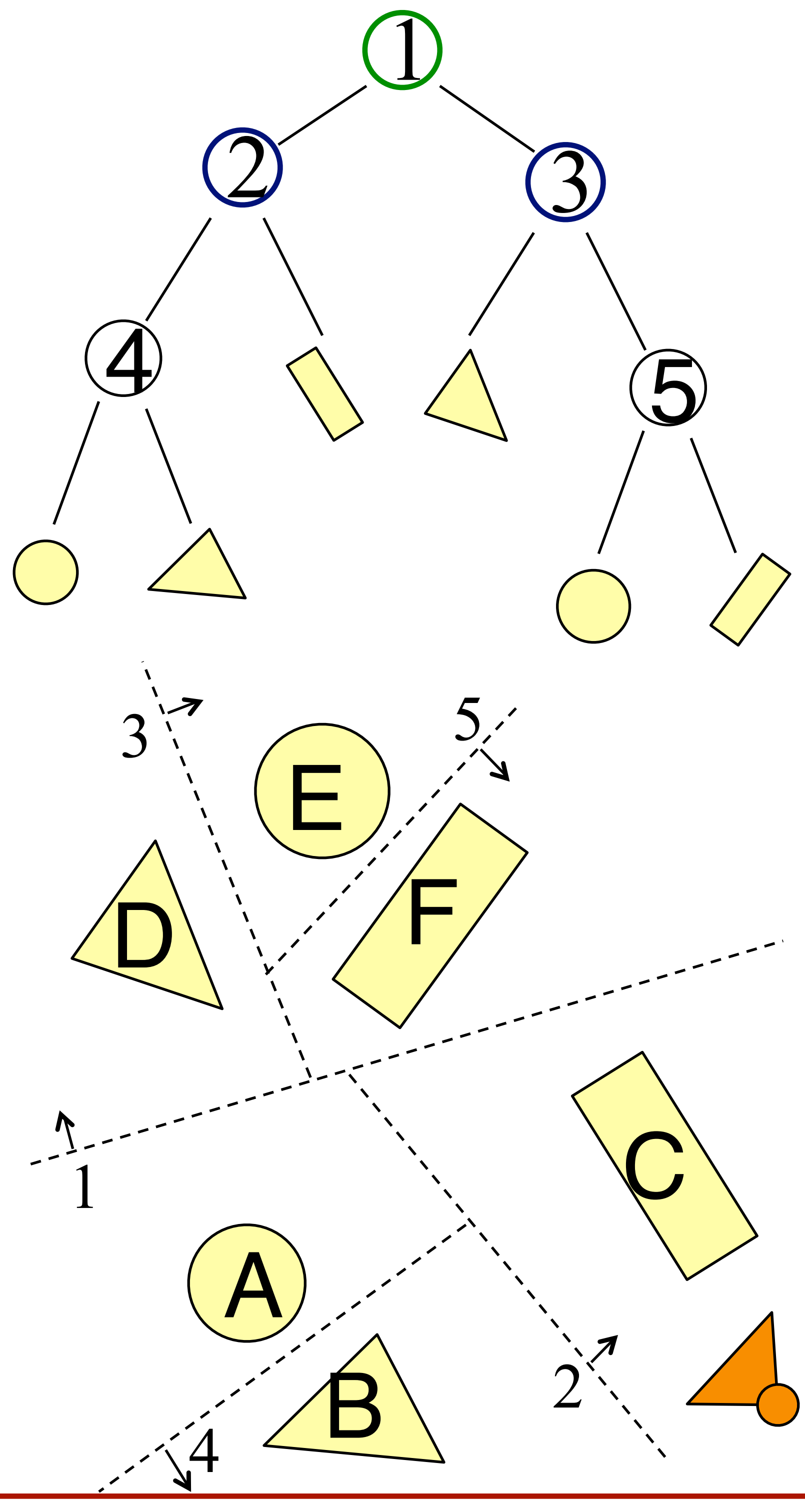
BSP-Tree Rendering (Object Precision)

- Draw further half first, then the closer one.
  - Draw right side of **1**
    - Draw left side of **3**
      - Draw **D**
    - Draw right side of **3**
  - Draw left side of **1**



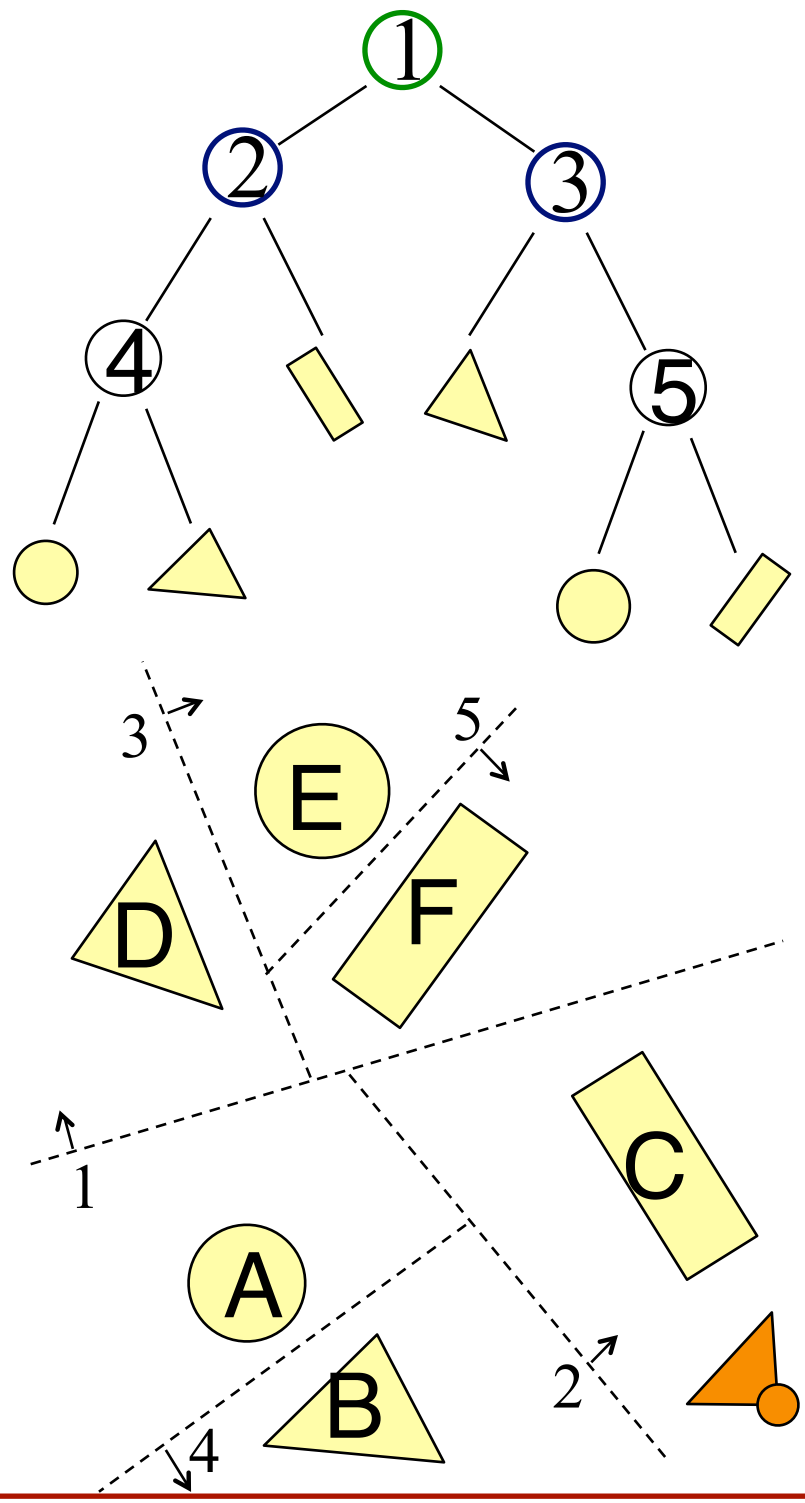
BSP-Tree Rendering (Object Precision)

- Draw further half first, then the closer one.
  - Draw right side of **1**
    - Draw left side of **3**
      - Draw **D**
    - Draw right side of **3**
      - Draw left side of **5**
      - Draw right side of **5**
  - Draw left side of **1**



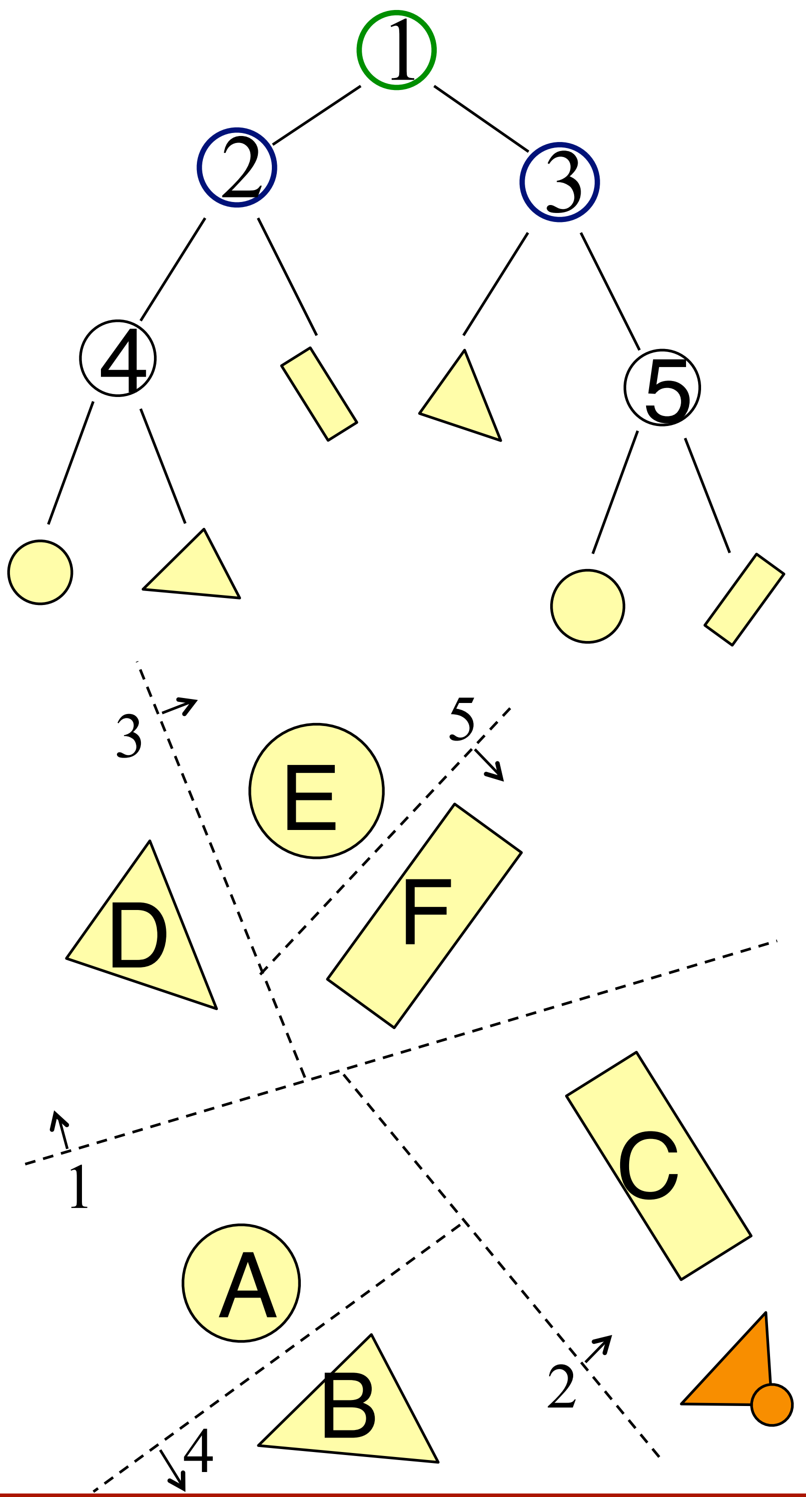
BSP-Tree Rendering (Object Precision)

- Draw further half first, then the closer one.
  - Draw right side of **1**
    - Draw left side of **3**
      - Draw **D**
    - Draw right side of **3**
      - Draw left side of **5**
        - Draw **E**
      - Draw right side of **5**
  - Draw left side of **1**



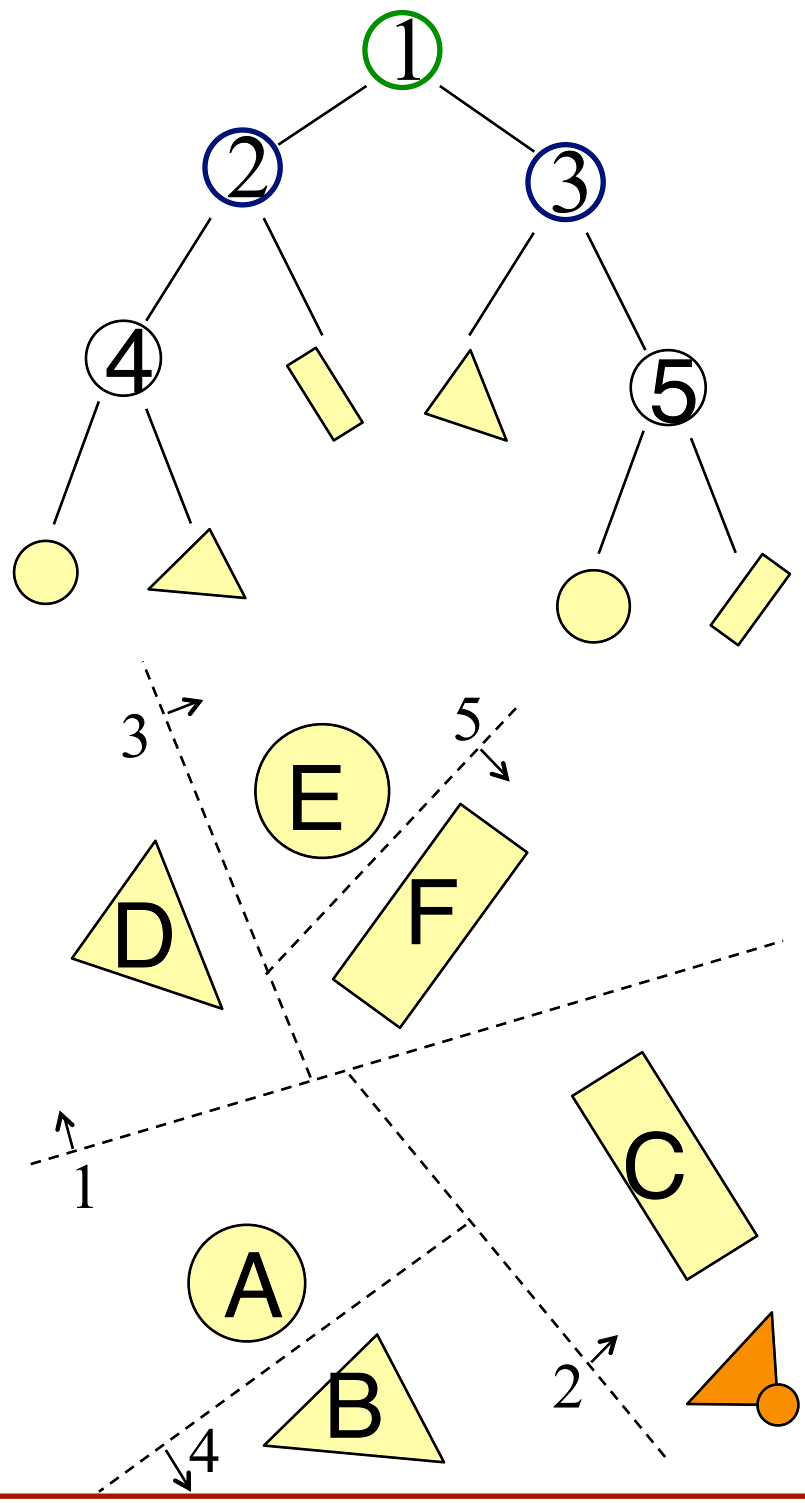
BSP-Tree Rendering (Object Precision)

- Draw further half first, then the closer one.
  - Draw right side of **1**
    - Draw left side of **3**
      - Draw **D**
    - Draw right side of **3**
      - Draw left side of **5**
        - Draw **E**
      - Draw right side of **5**
        - Draw **F**
  - Draw left side of **1**



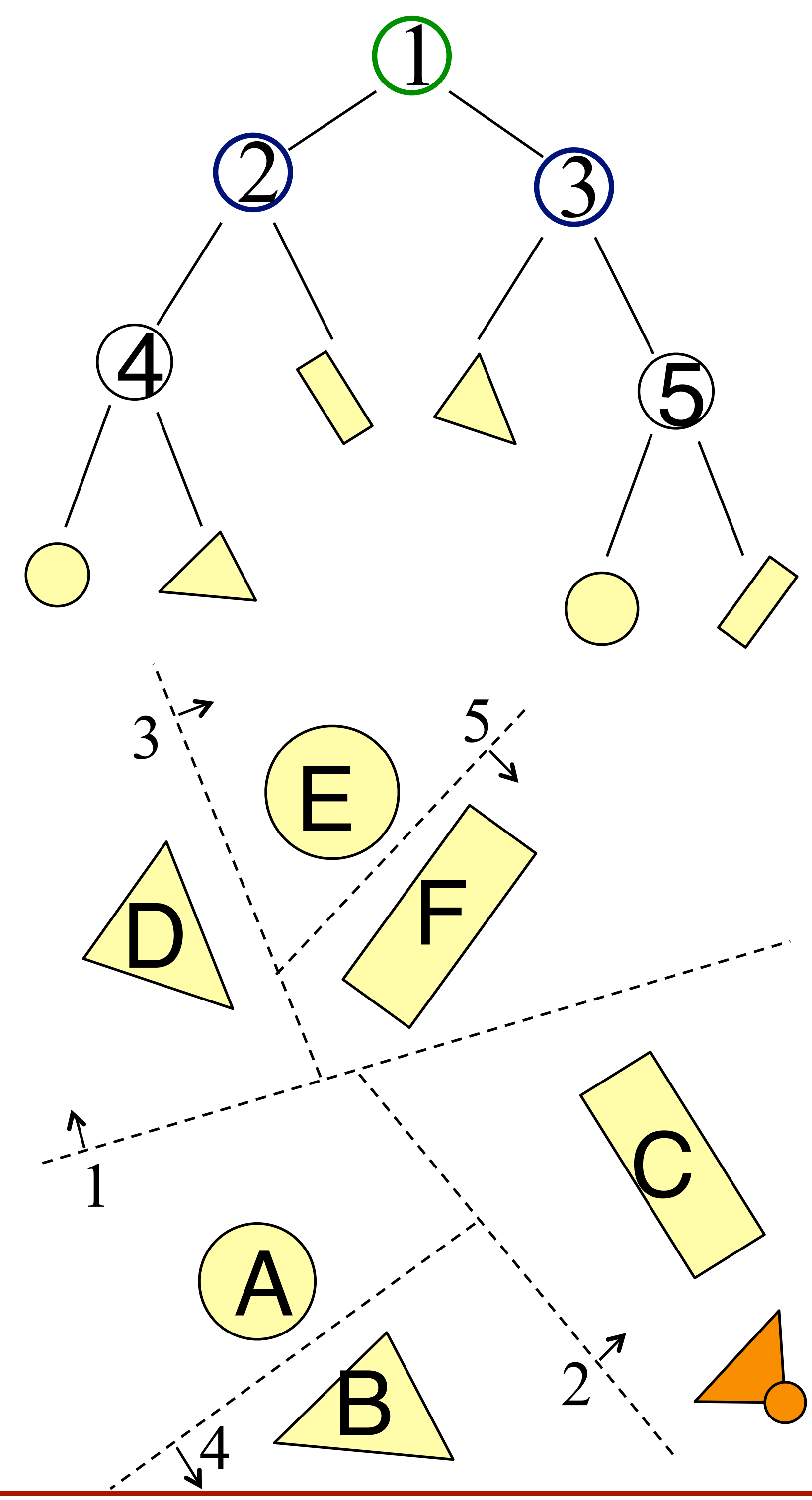
BSP-Tree Rendering (Object Precision)

- Draw further half first, then the closer one.
  - Draw right side of **1**
    - Draw left side of **3**
      - Draw **D**
    - Draw right side of **3**
      - Draw left side of **5**
        - Draw **E**
      - Draw right side of **5**
        - Draw **F**
  - Draw left side of **1**
    - Draw left side of **2**
    - Draw right side of **2**



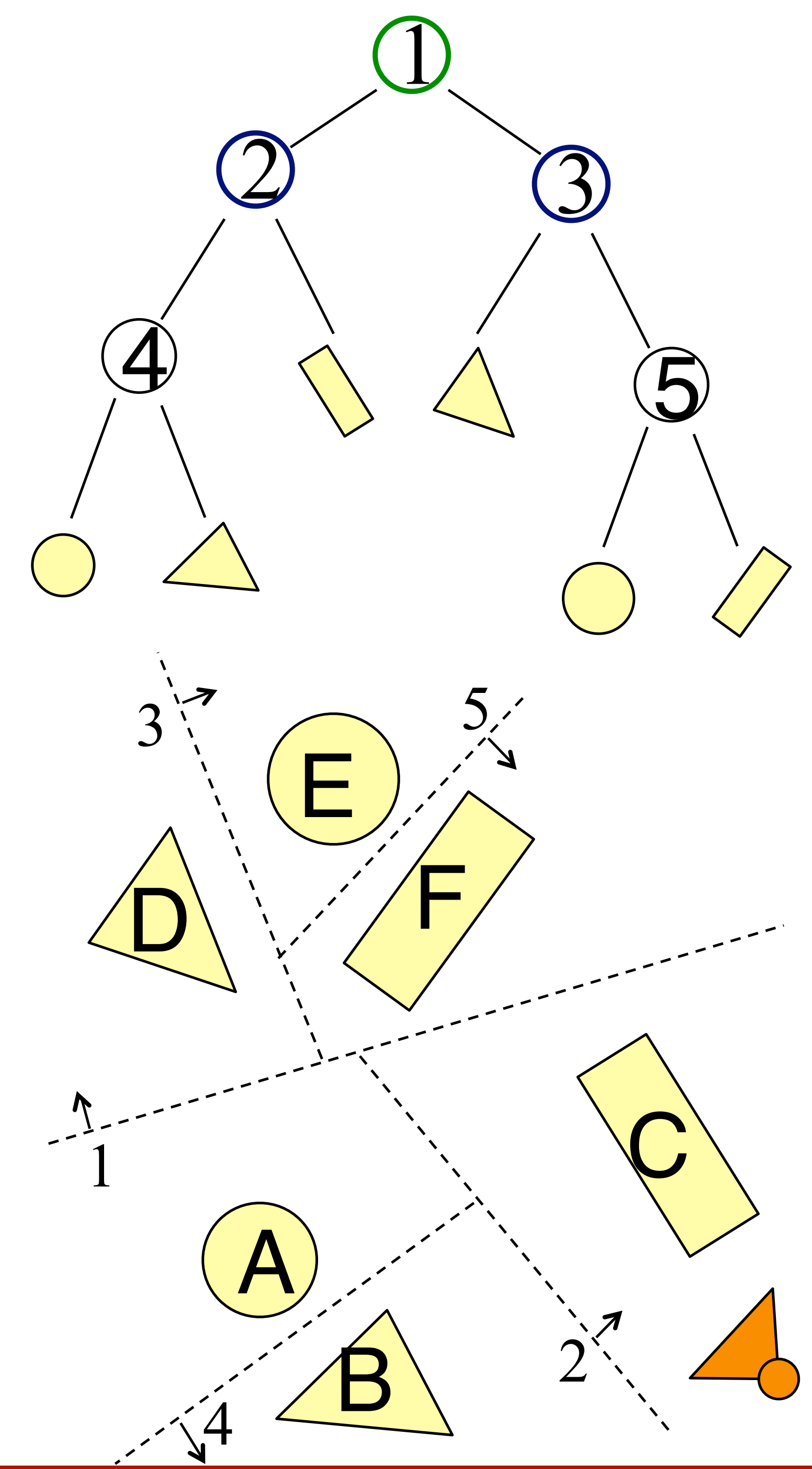
BSP-Tree Rendering (Object Precision)

- Draw further half first, then the closer one.
  - Draw right side of **1**
    - Draw left side of **3**
      - Draw **D**
    - Draw right side of **3**
      - Draw left side of **5**
        - Draw **E**
      - Draw right side of **5**
        - Draw **F**
  - Draw left side of **1**
    - Draw left side of **2**
      - Draw left side of **4**
      - Draw right side of **4**
    - Draw right side of **2**



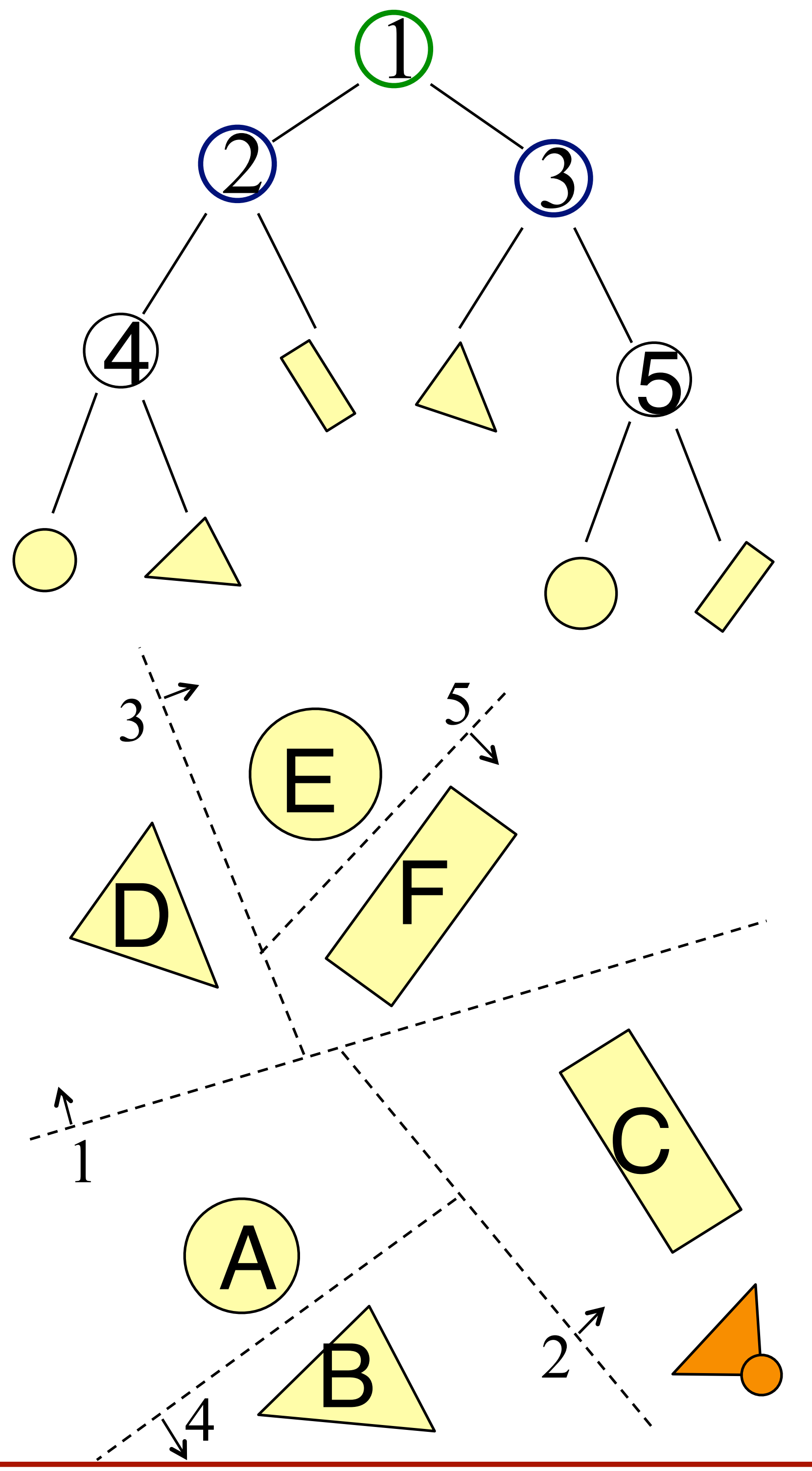
BSP-Tree Rendering (Object Precision)

- Draw further half first, then the closer one.
  - Draw right side of **1**
    - Draw left side of **3**
      - Draw **D**
    - Draw right side of **3**
      - Draw left side of **5**
        - Draw **E**
      - Draw right side of **5**
        - Draw **F**
  - Draw left side of **1**
    - Draw left side of **2**
      - Draw left side of **4**
        - Draw **A**
      - Draw right side of **4**
    - Draw right side of **2**



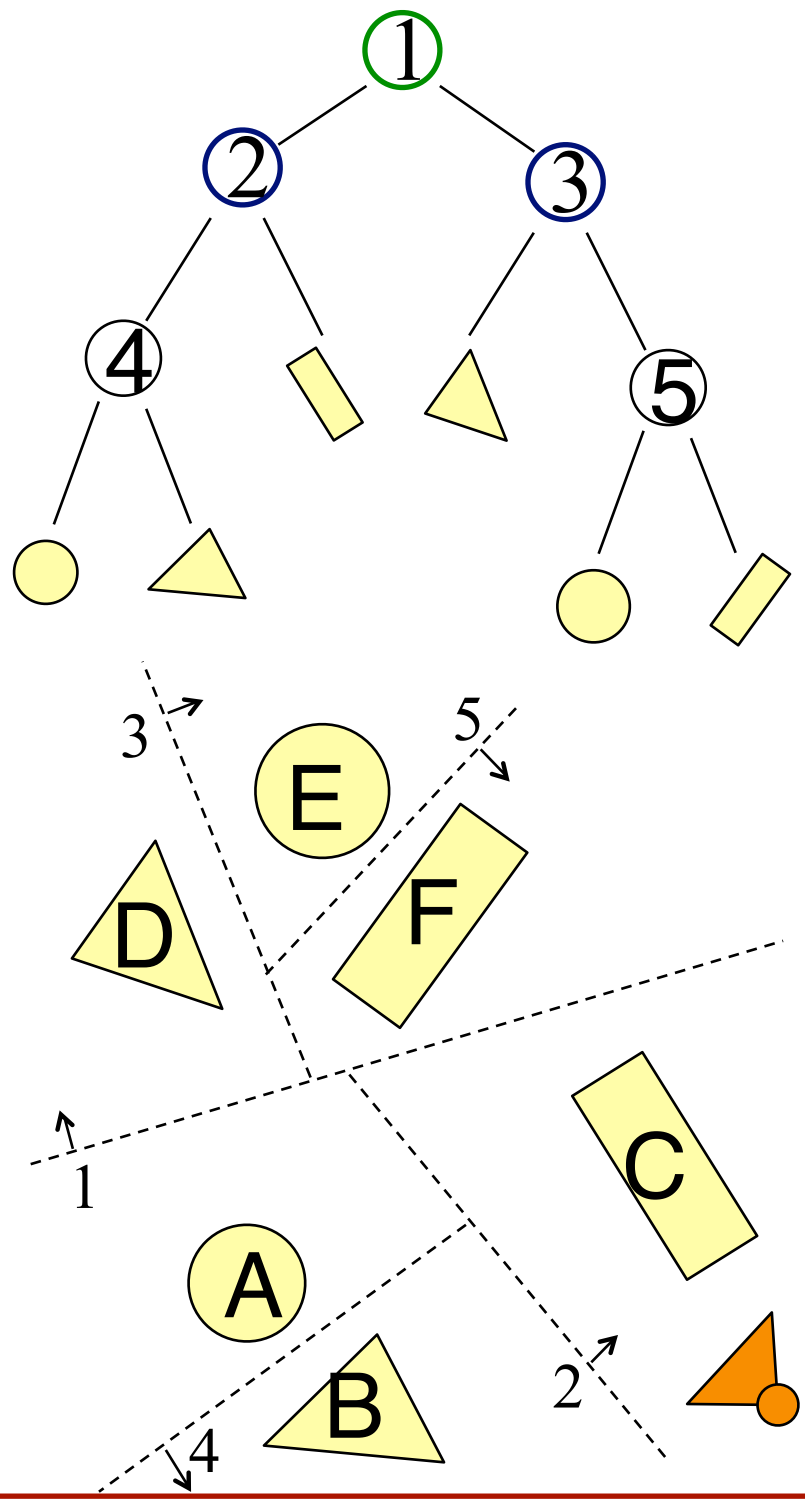
BSP-Tree Rendering (Object Precision)

- Draw further half first, then the closer one.
  - Draw right side of **1**
    - Draw left side of **3**
      - Draw **D**
    - Draw right side of **3**
      - Draw left side of **5**
        - Draw **E**
      - Draw right side of **5**
        - Draw **F**
  - Draw left side of **1**
    - Draw left side of **2**
      - Draw left side of **4**
        - Draw **A**
      - Draw right side of **4**
        - Draw **B**
    - Draw right side of **2**



BSP-Tree Rendering (Object Precision)

- Draw further half first, then the closer one.
  - Draw right side of **1**
    - Draw left side of **3**
      - Draw **D**
    - Draw right side of **3**
      - Draw left side of **5**
        - Draw **E**
      - Draw right side of **5**
        - Draw **F**
  - Draw left side of **1**
    - Draw left side of **2**
      - Draw left side of **4**
        - Draw **A**
      - Draw right side of **4**
        - Draw **B**
    - Draw right side of **2**
      - Draw **C**



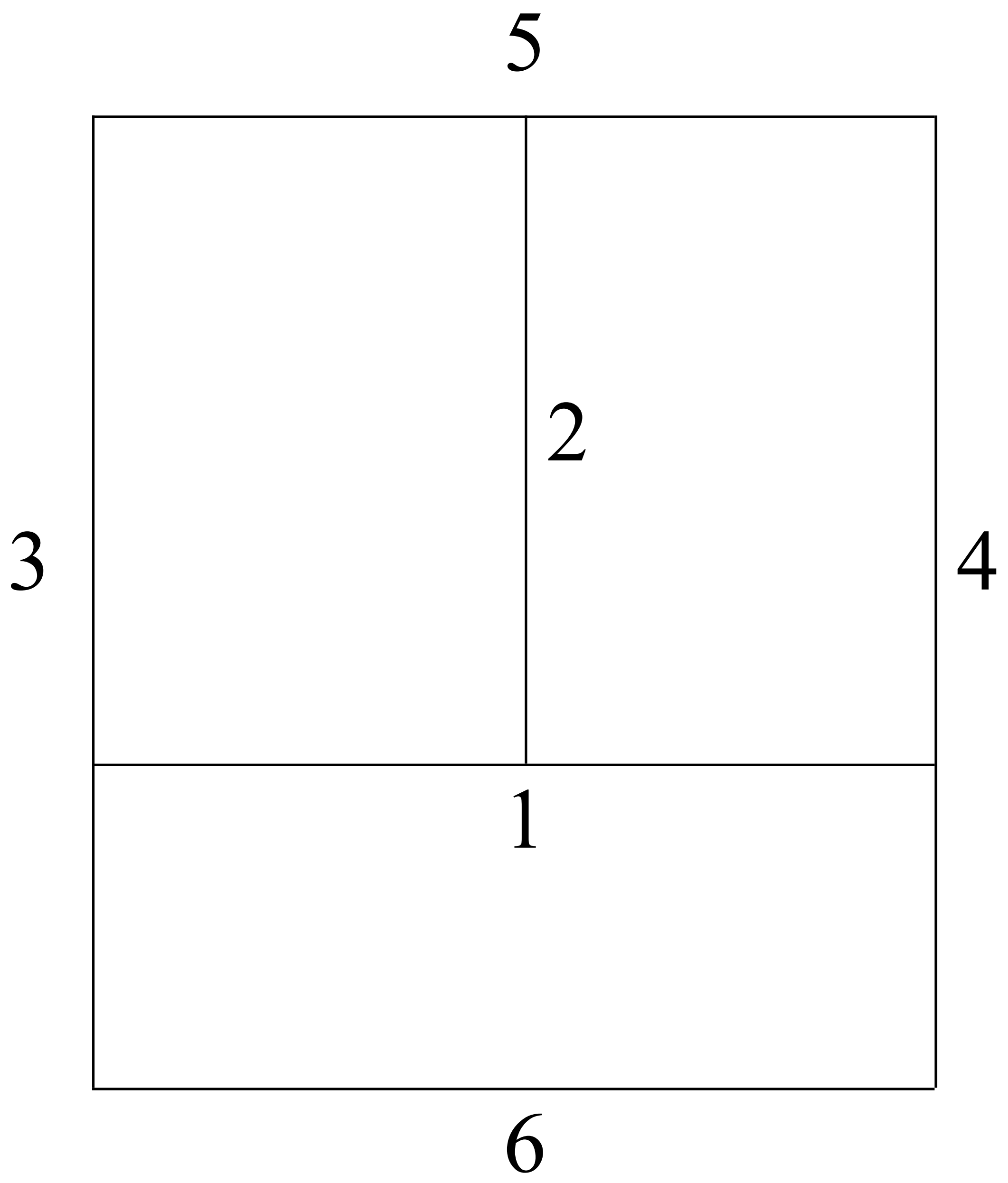
Building BSP-Trees

- Choose polygon (arbitrary)
- Split its cell using plane on which polygon lies
  - May have to chop polygons in two (Clipping!)
- Continue until each cell contains only one polygon fragment
- Splitting planes could be chosen in other ways, but there is no efficient optimal algorithm for building BSP trees
  - Optimal means minimum number of polygon fragments in a balanced tree

Slide courtesy UWisconsin CS559

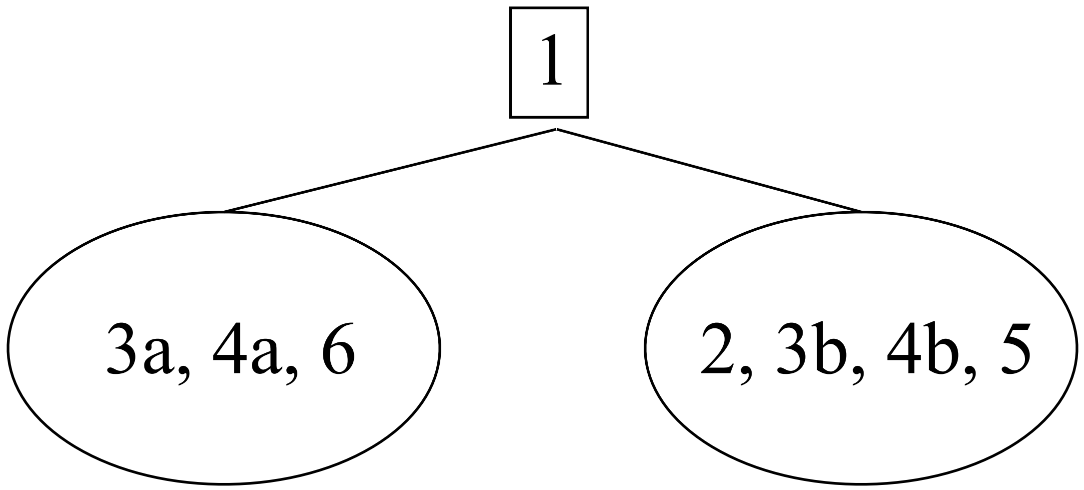
Building Example

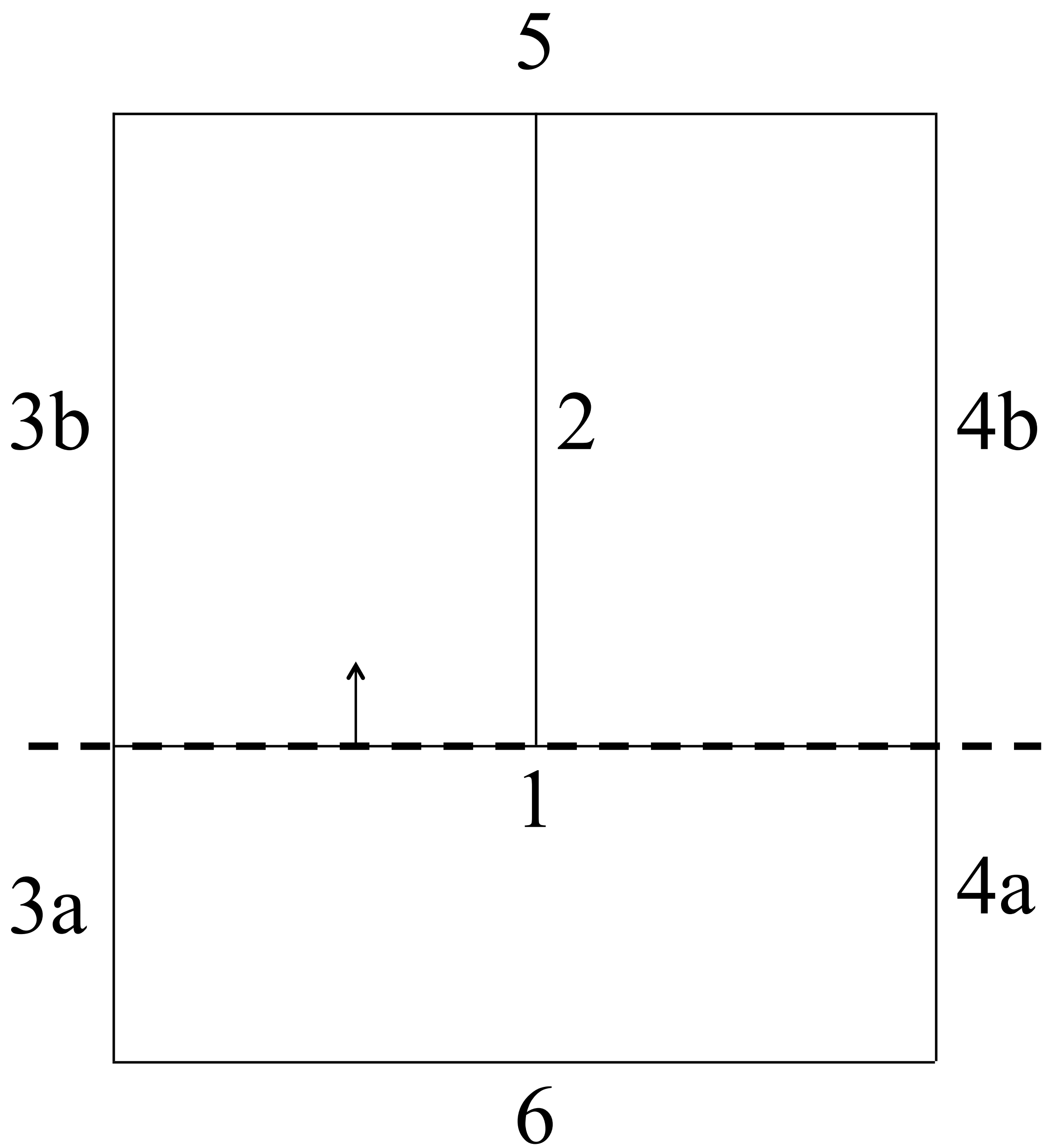
- We will build a BSP tree, in 2D, for a 3 room building



Slide courtesy UWisconsin CS559

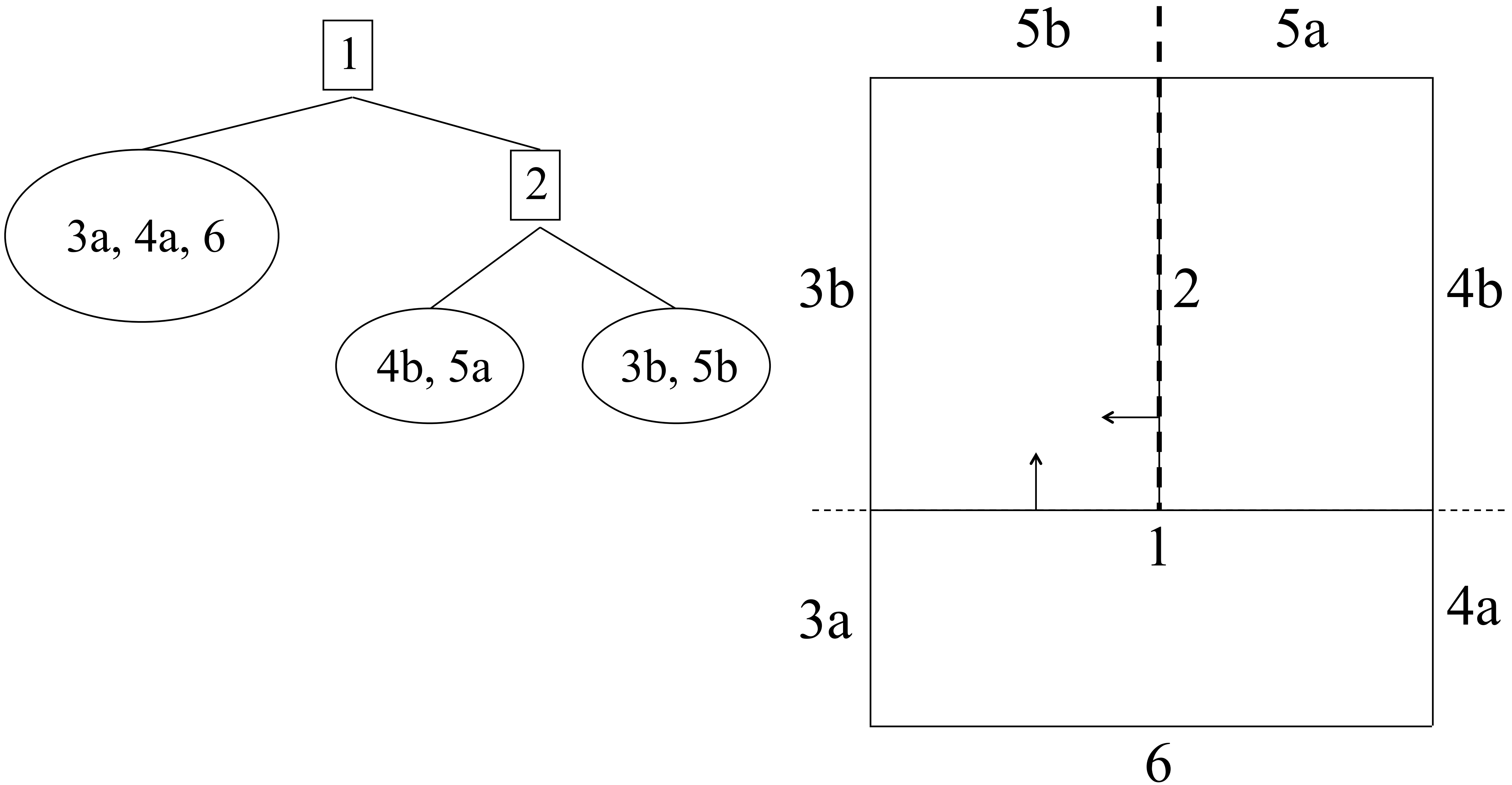
Building Example (1)





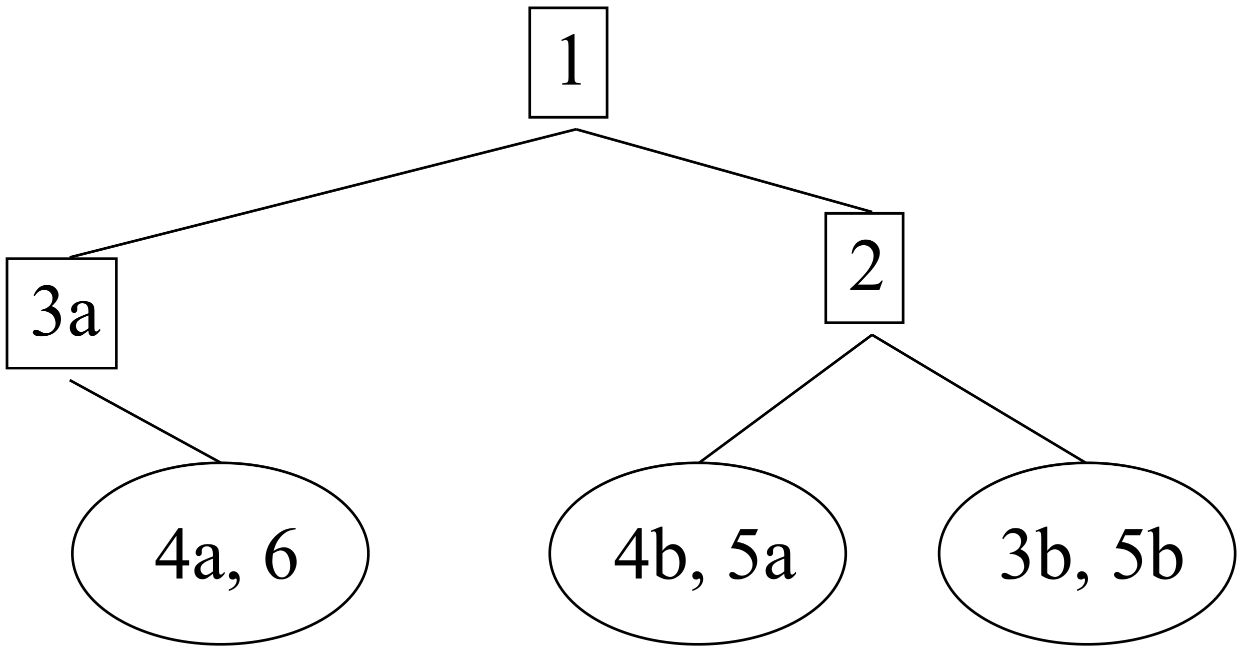
Slide courtesy UWisconsin CS559

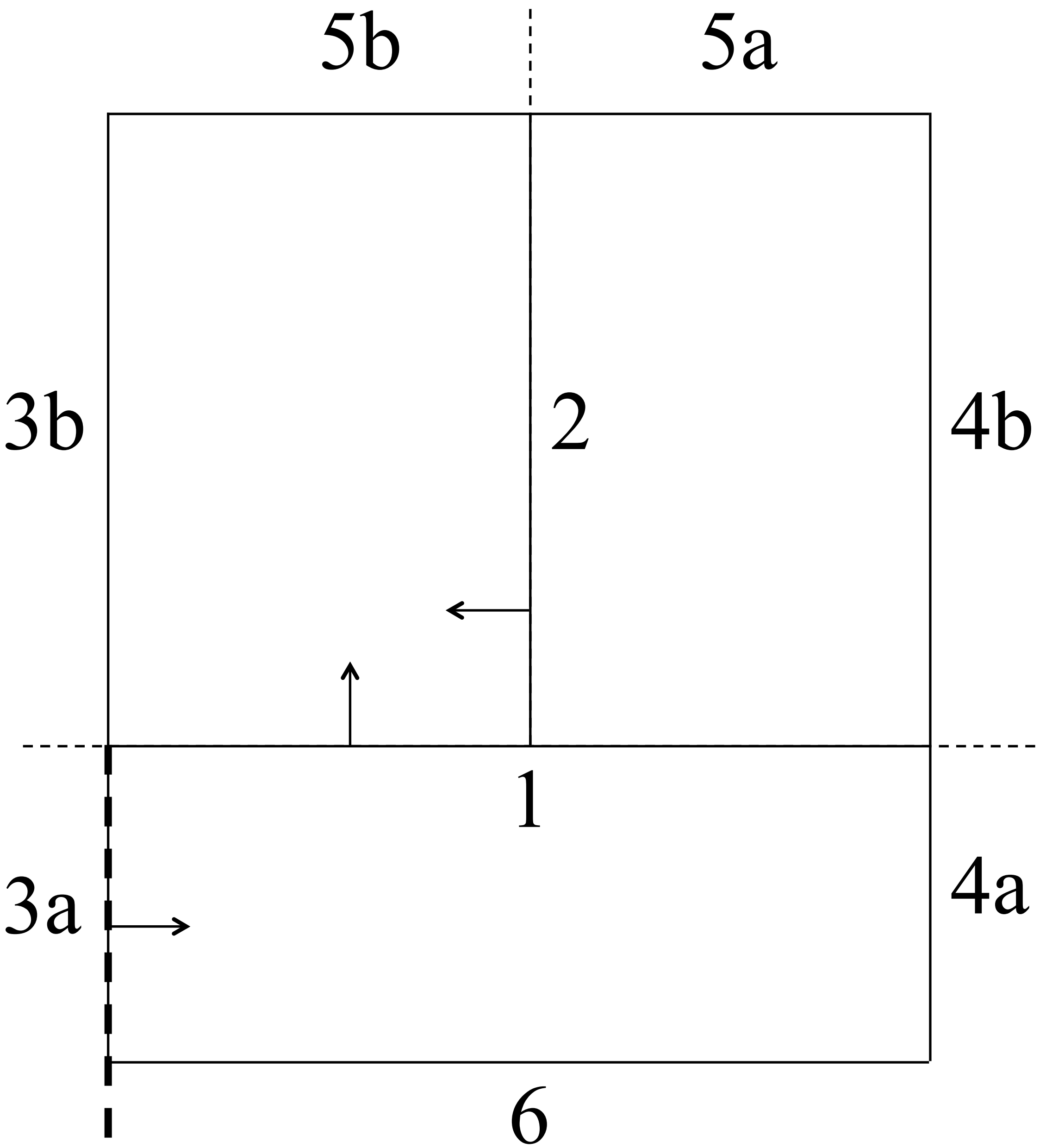
Building Example (2)



Slide courtesy UWisconsin CS559

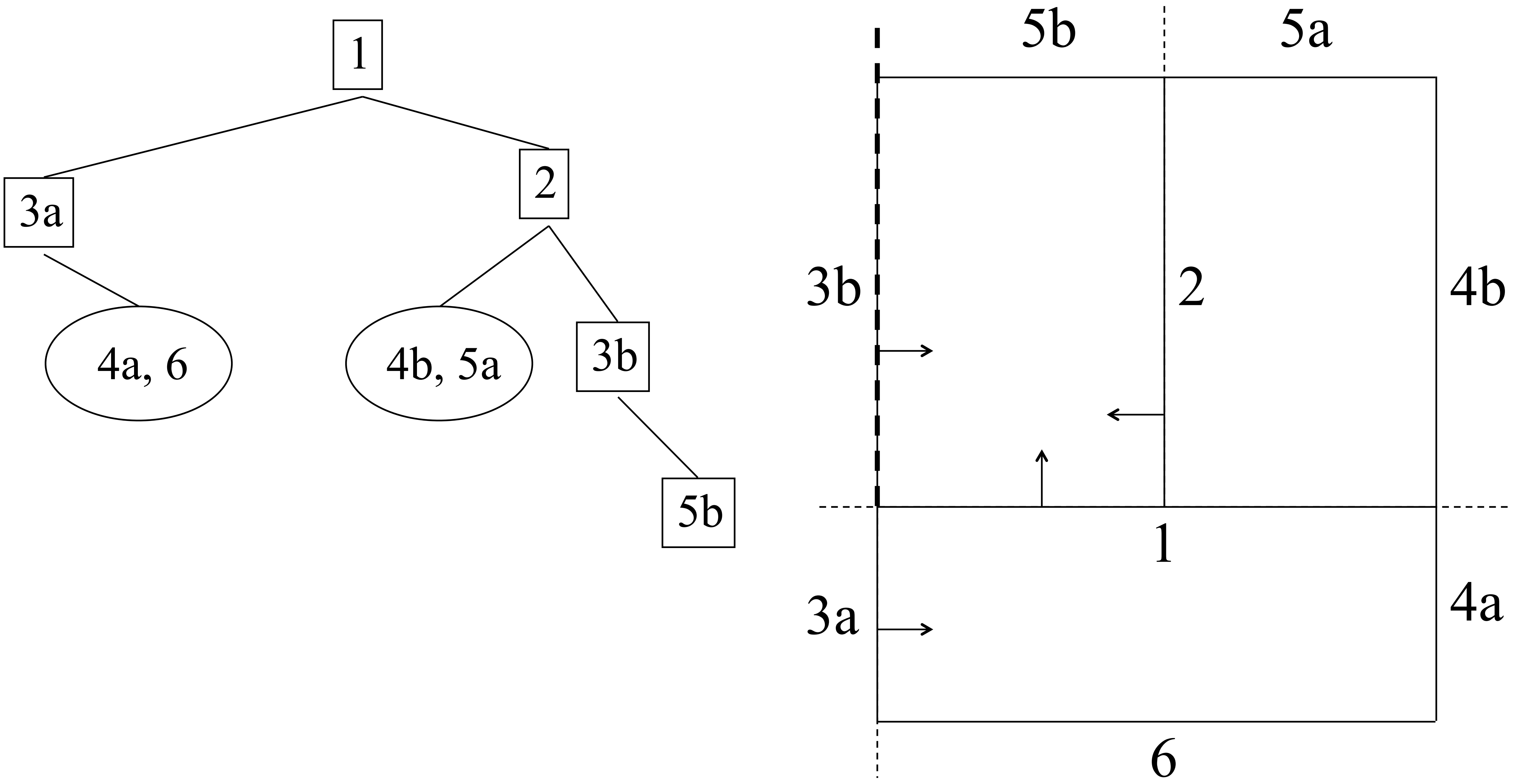
Building Example (3)





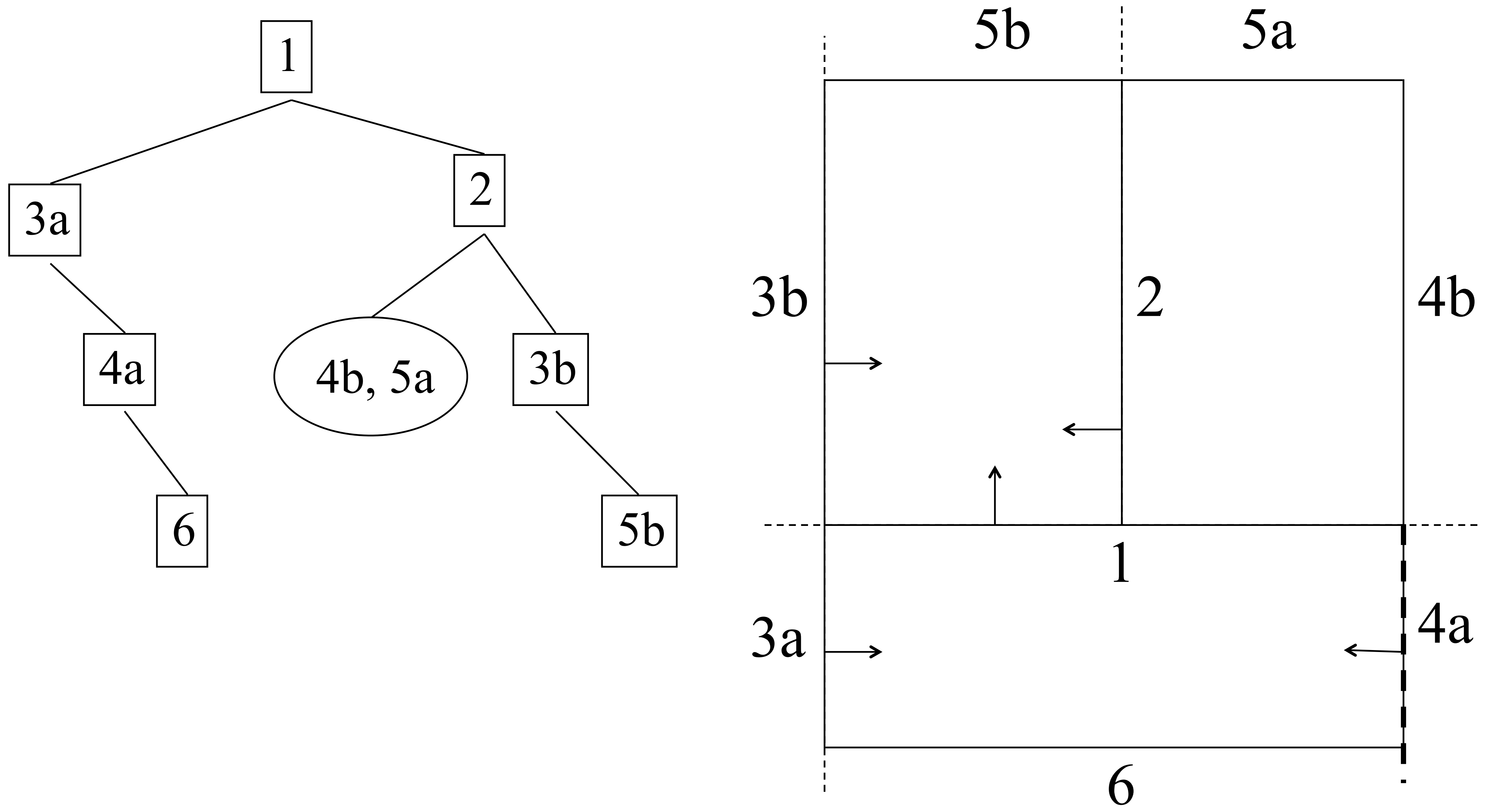
Slide courtesy UWisconsin CS559

Building Example (4)



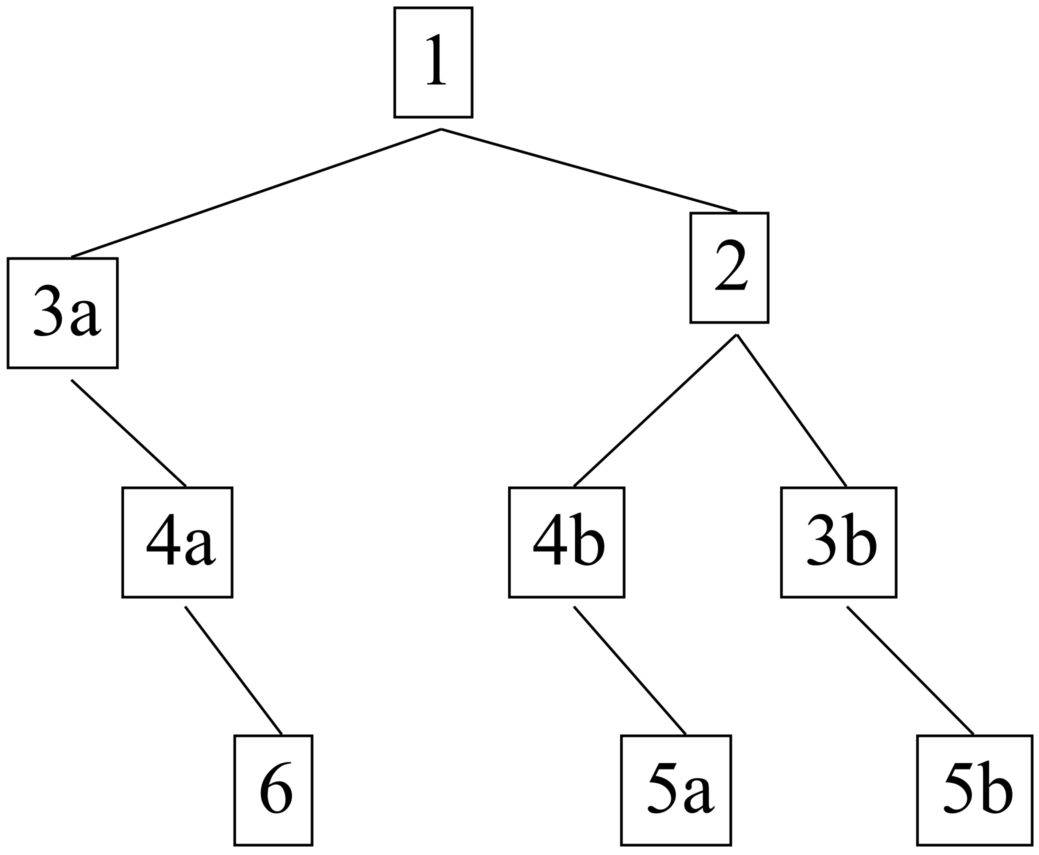
Slide courtesy UWisconsin CS559

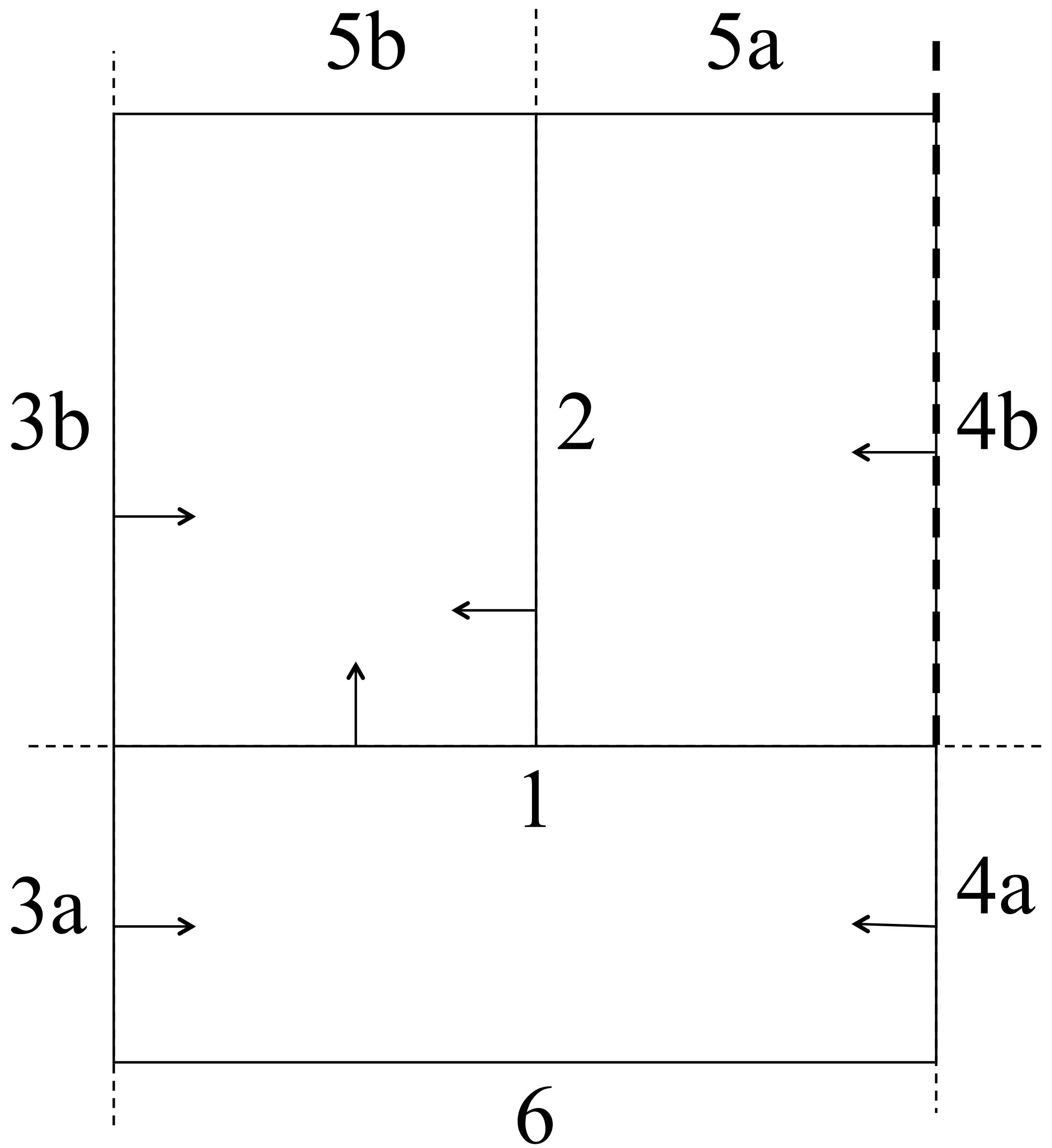
Building Example (5)



Slide courtesy UWisconsin CS559

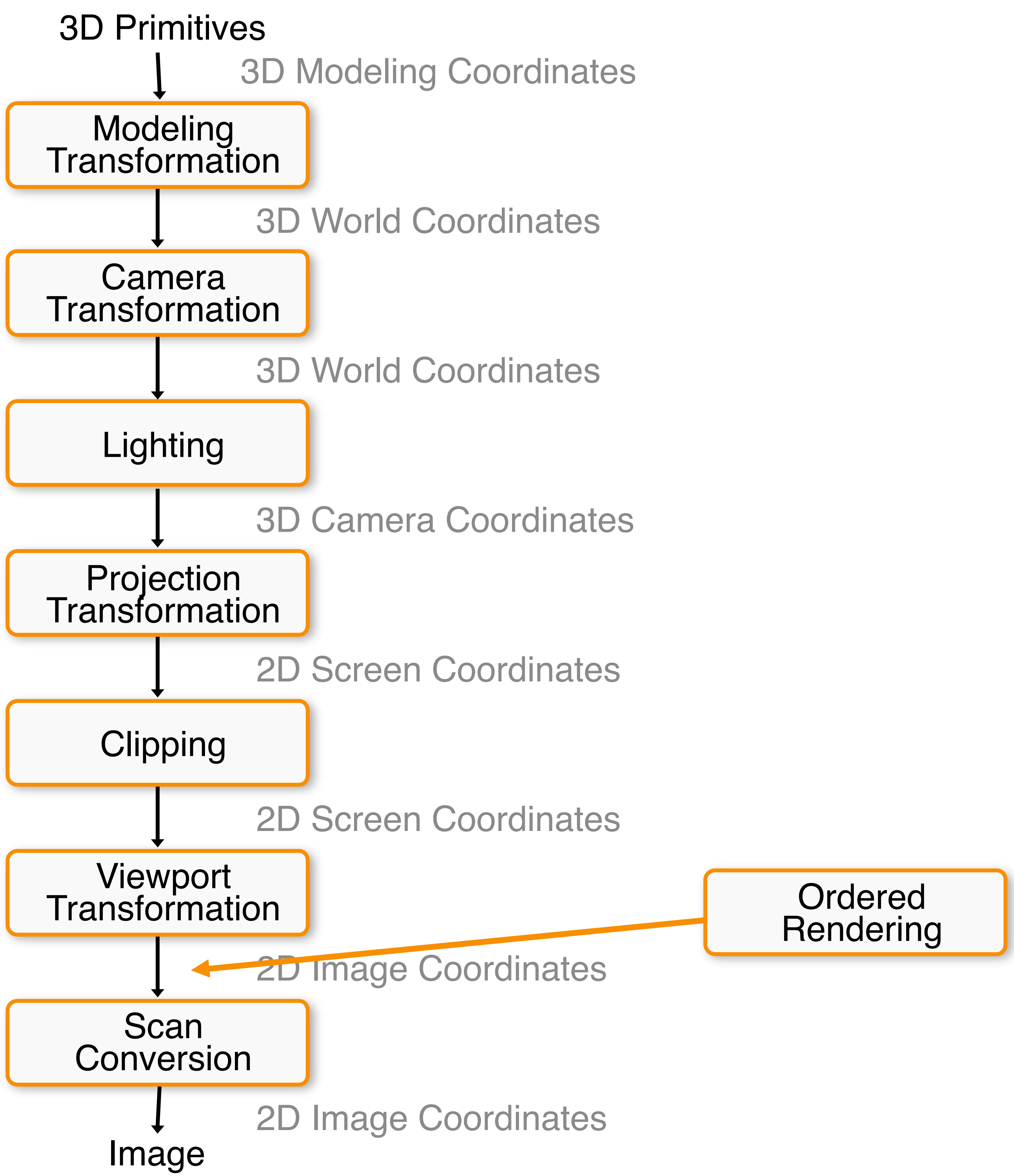
Building Example (Done)





Slide courtesy UWisconsin CS559

3D Rendering Pipeline

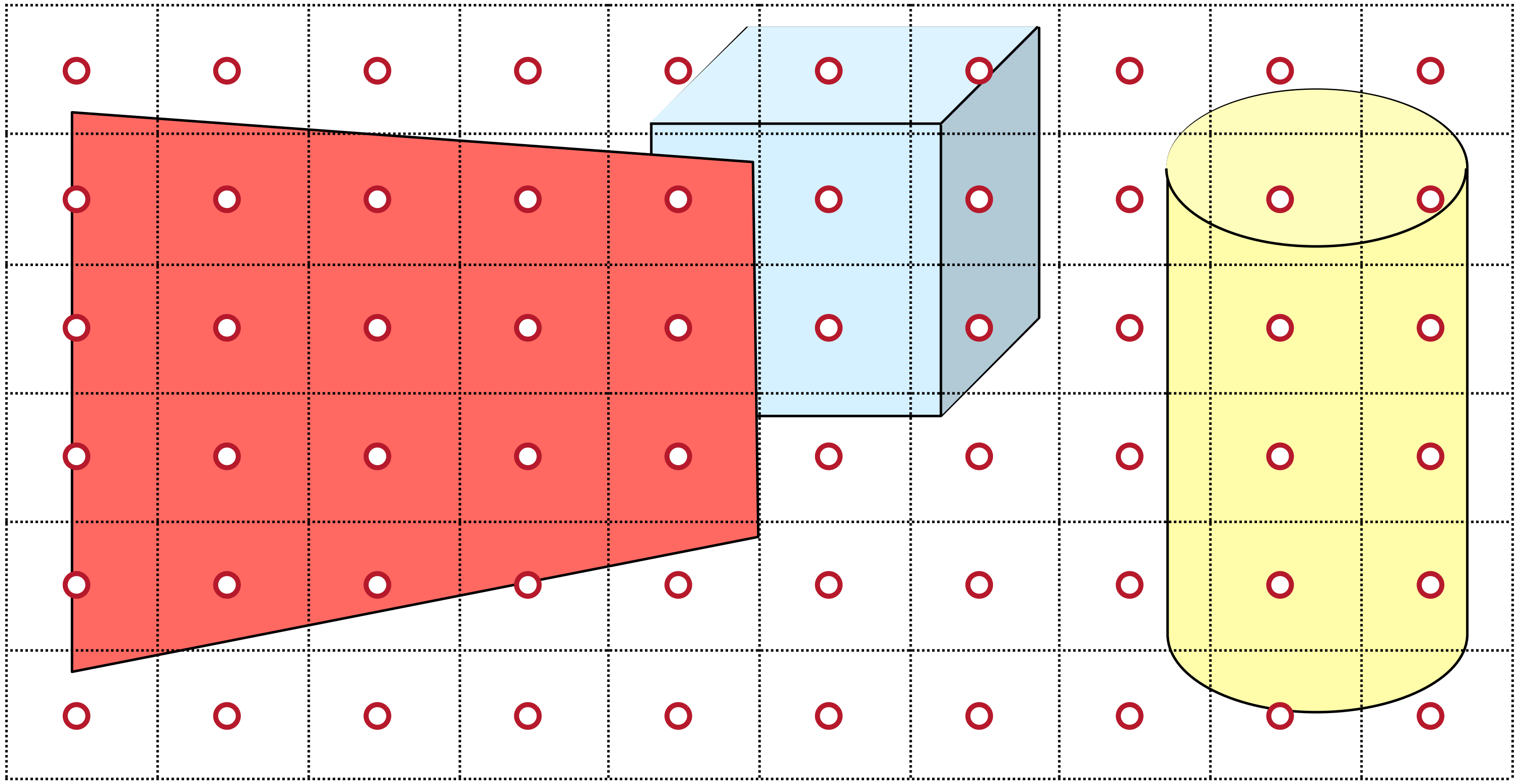


Binary Space Partition:

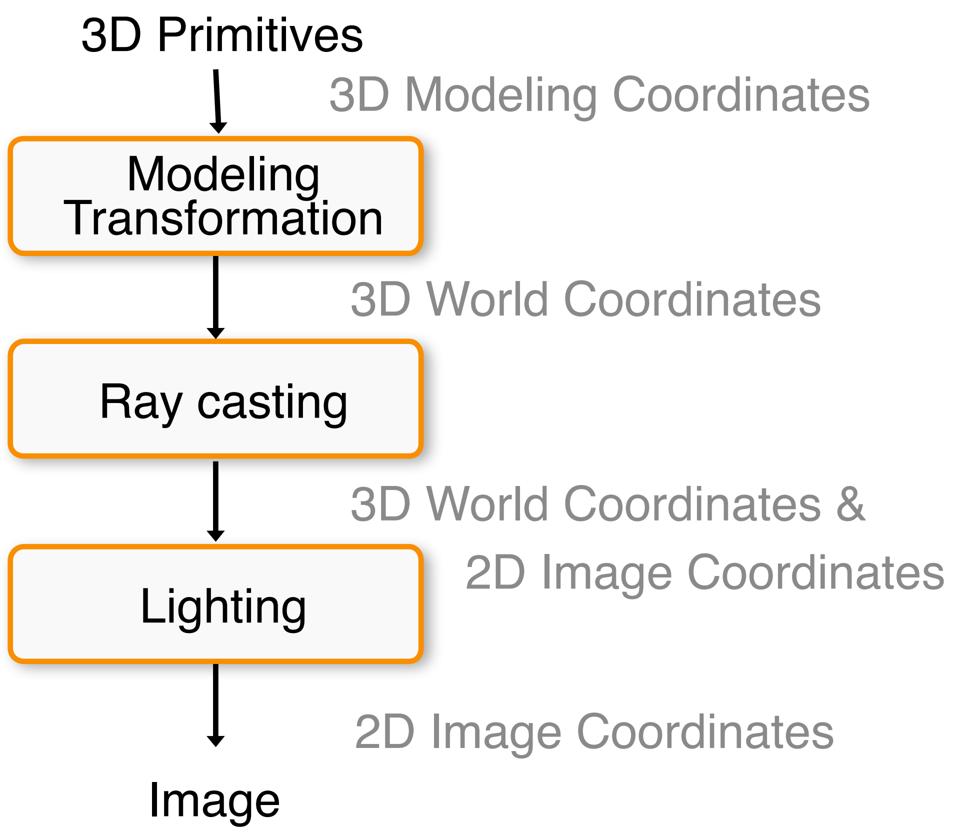
- View Independent
- Linear-time depth sort

Ray Casting

- Fire a ray for every pixel
  - If ray intersects multiple objects, take the closest



Ray Casting Pipeline

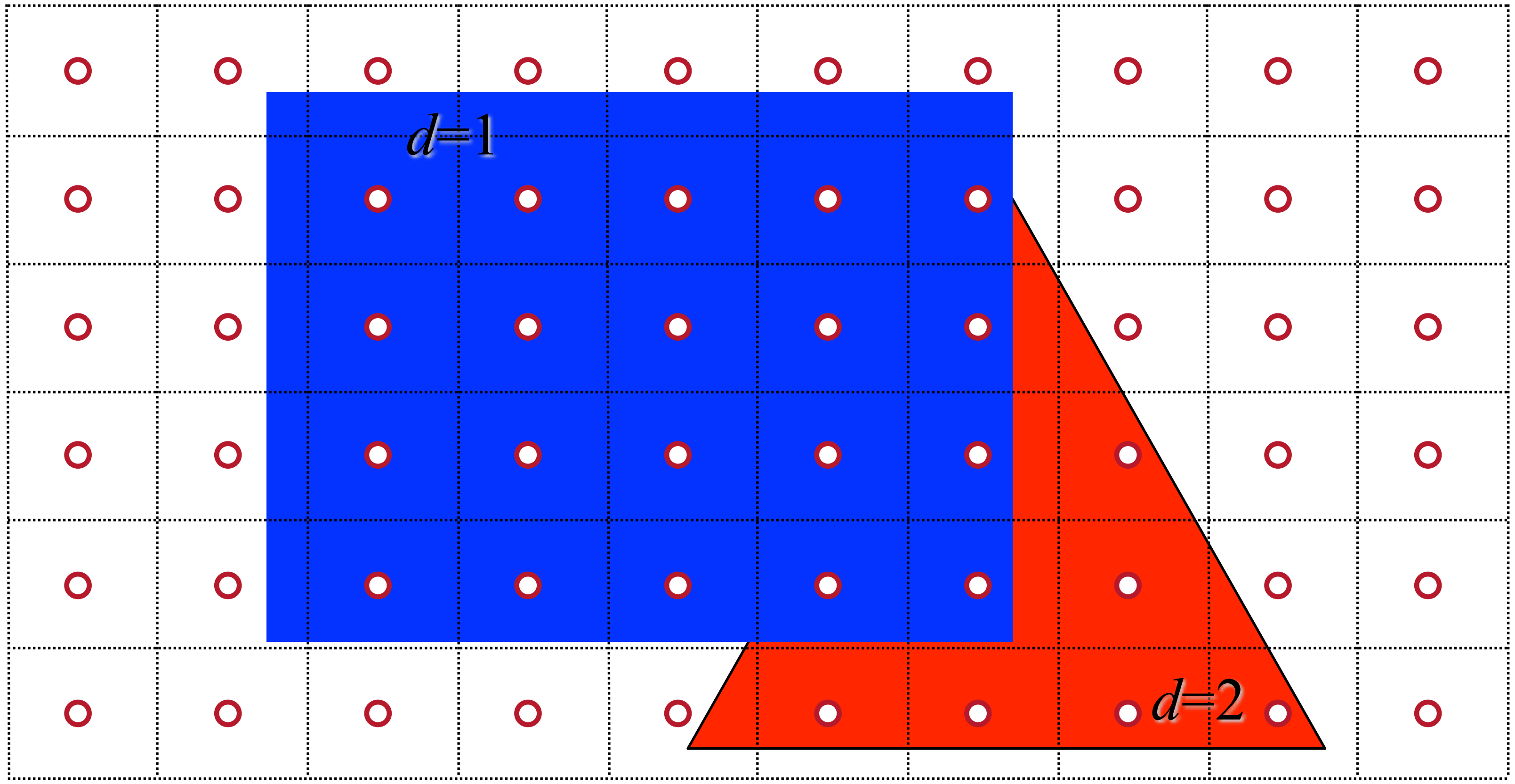


Ray casting comments

- O(p log n) for p pixels
- May (or not) use pixel coherence
- Simple, but generally not used

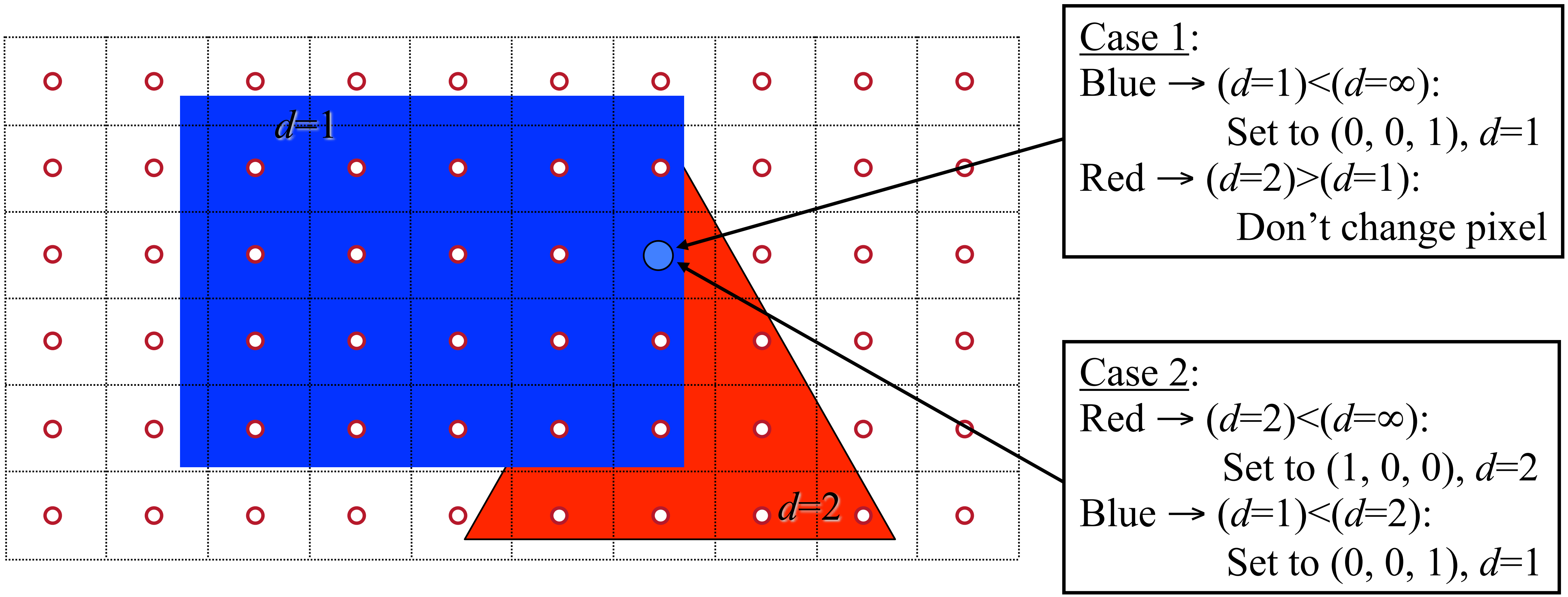
Z-Buffer

- Store color & depth of closest object at each pixel
  - Initialize depth of each pixel to ∞
  - Update only pixels whose depth is closer than in buffer



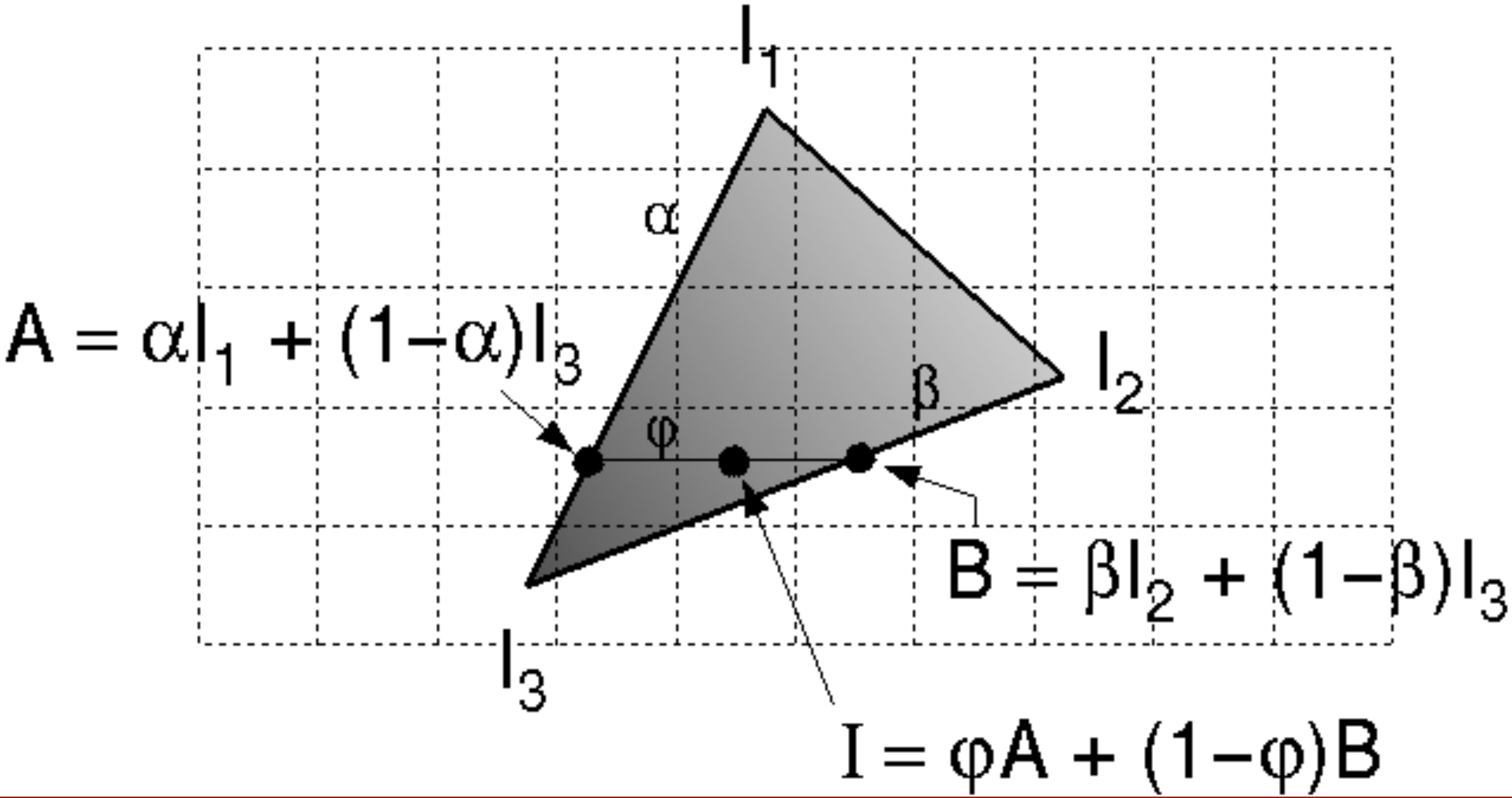
Z-Buffer

- Store color & depth of closest object at each pixel
  - Initialize depth of each pixel to ∞
  - Update only pixels whose depth is closer than in buffer



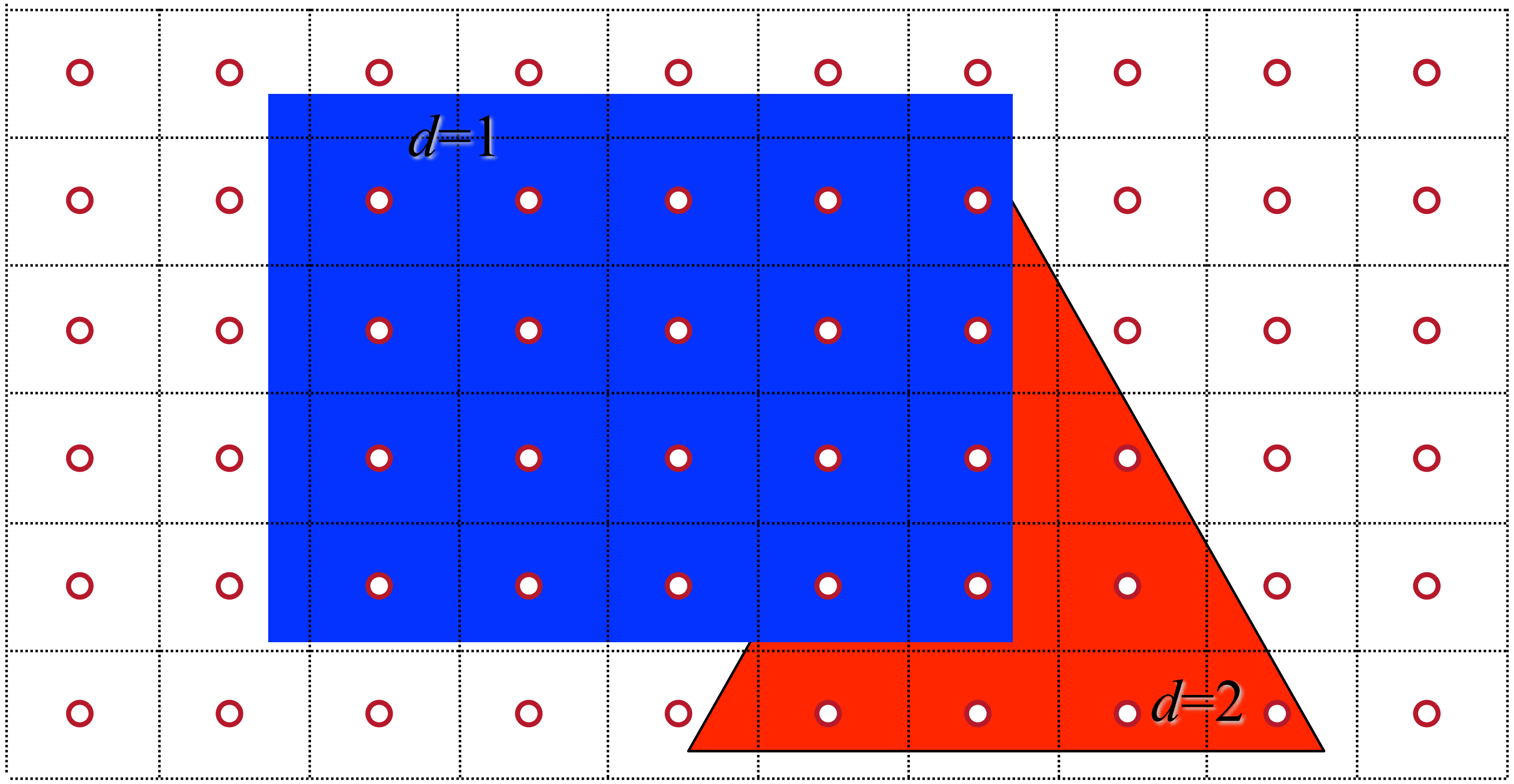
Z-Buffer

- Store color & depth of closest object at each pixel
  - Initialize depth of each pixel to ∞
  - Update only pixels whose depth is closer than in buffer
  - Depths are interpolated from vertices, just like colors

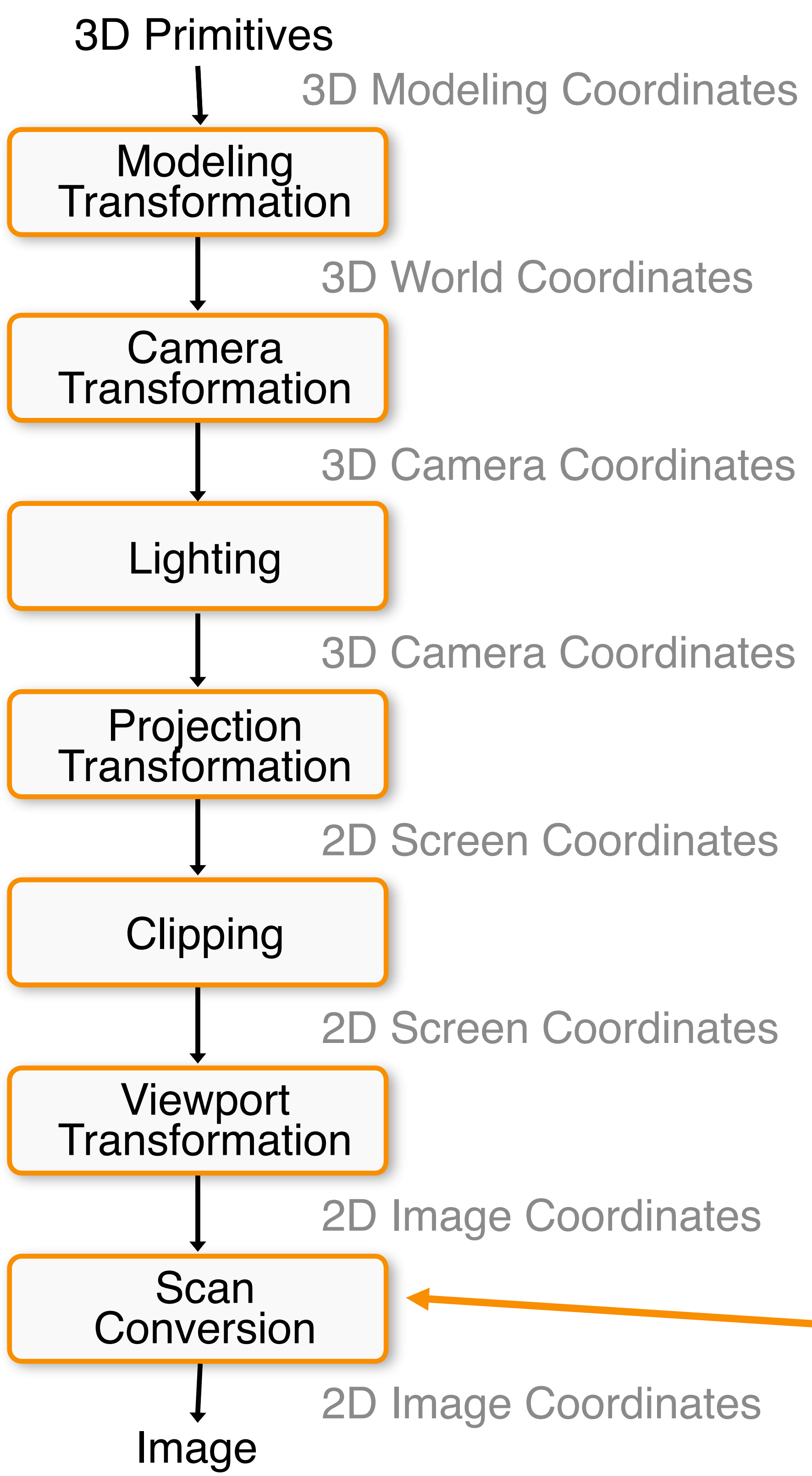


A-Buffer

- Alpha values can cause problems:
  - Z-buffer can only find one visible surface at each pixel
  - A-buffer supports linked list of surfaces at each pixel for better transparency support
  - A-buffer also helps with anti-aliasing



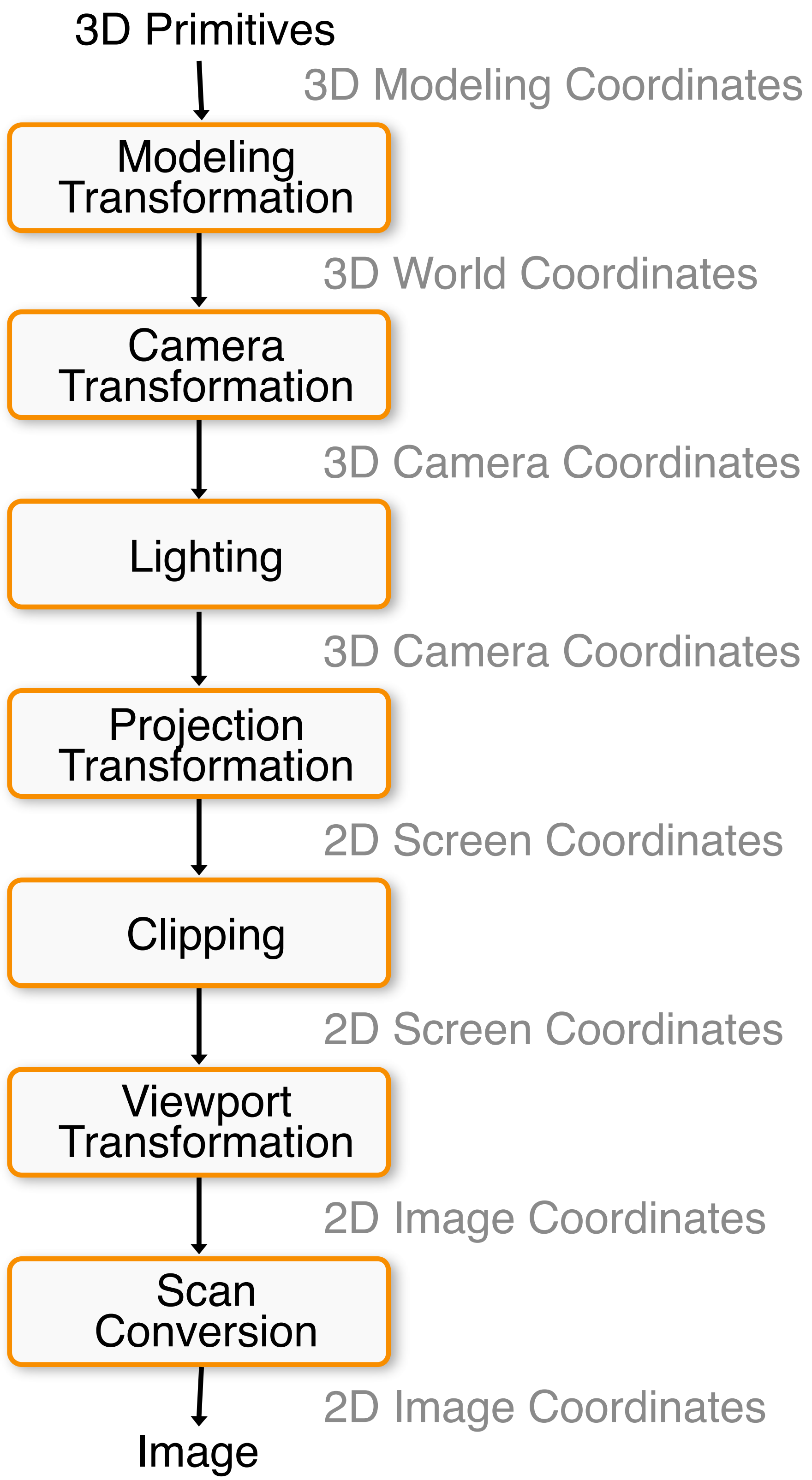
3D Rendering Pipeline

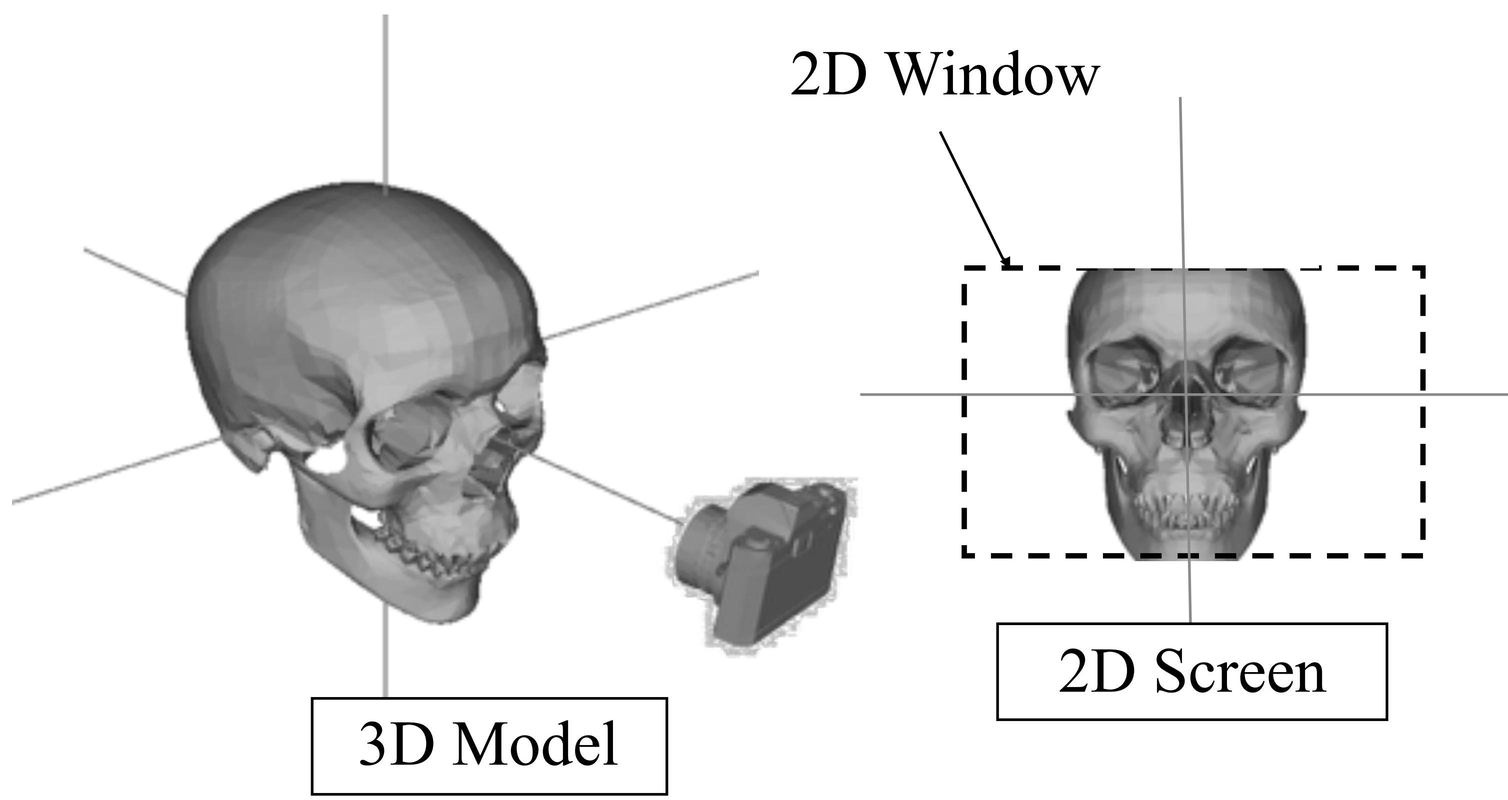


Z-buffer comments

- Polygons rasterized in any order
- Requires additional memory
  - Z-buffer size ≈ frame buffer
  - Was expensive, cheap now
- Has problems with Alpha (A-buffer)
- Very common in hardware

3D Rendering Pipeline (for direct illumination)





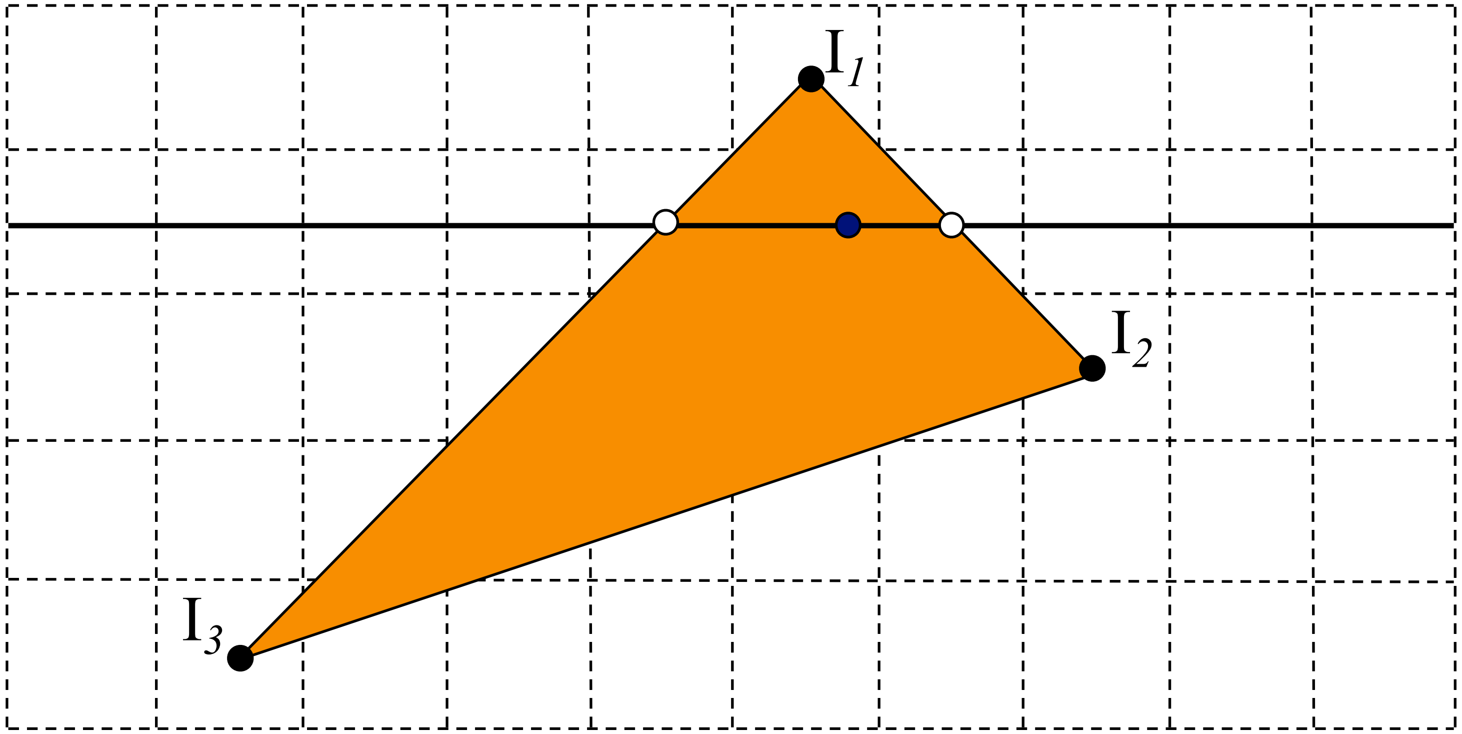
Scan Conversion

How do we average information from the three vertices of a triangle?

- Interpolate using weights determined by the screen space projection?
- Interpolate using weights determined by the 3D locations?

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

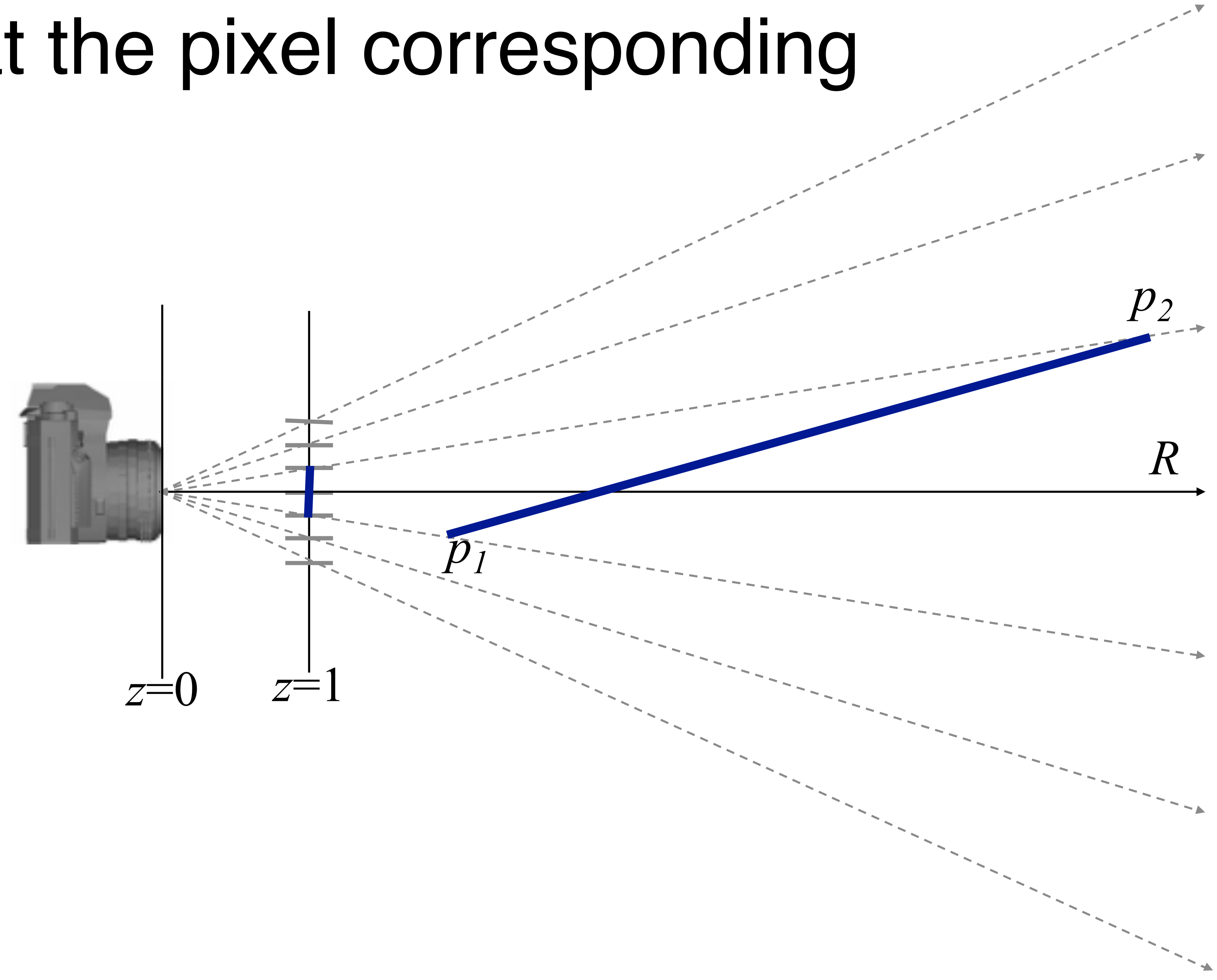
Is there a difference?



Scan Conversion Example

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

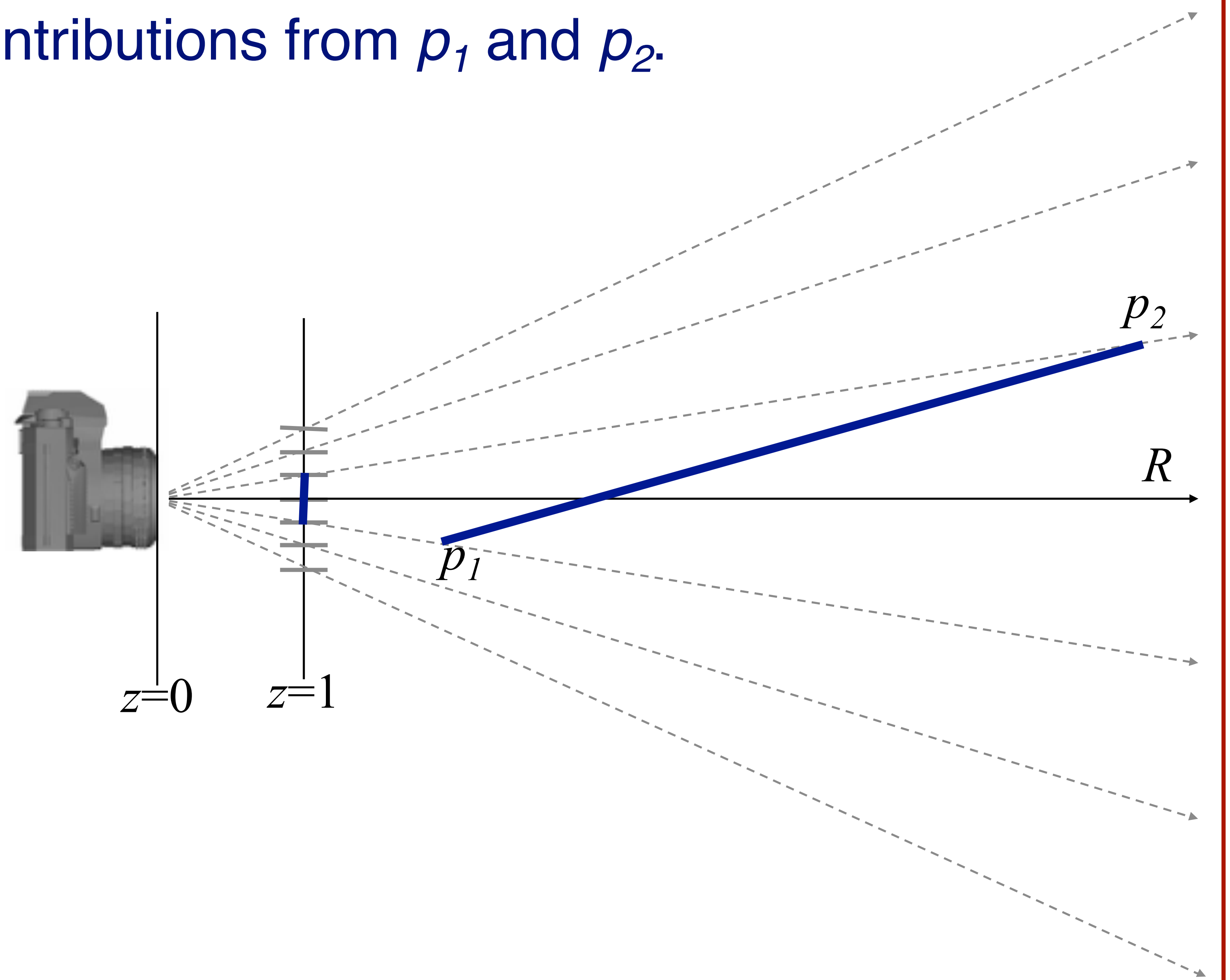
How should we interpolate the information from vertices p_1 and p_2 at the pixel corresponding to ray R?



Scan Conversion Example

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

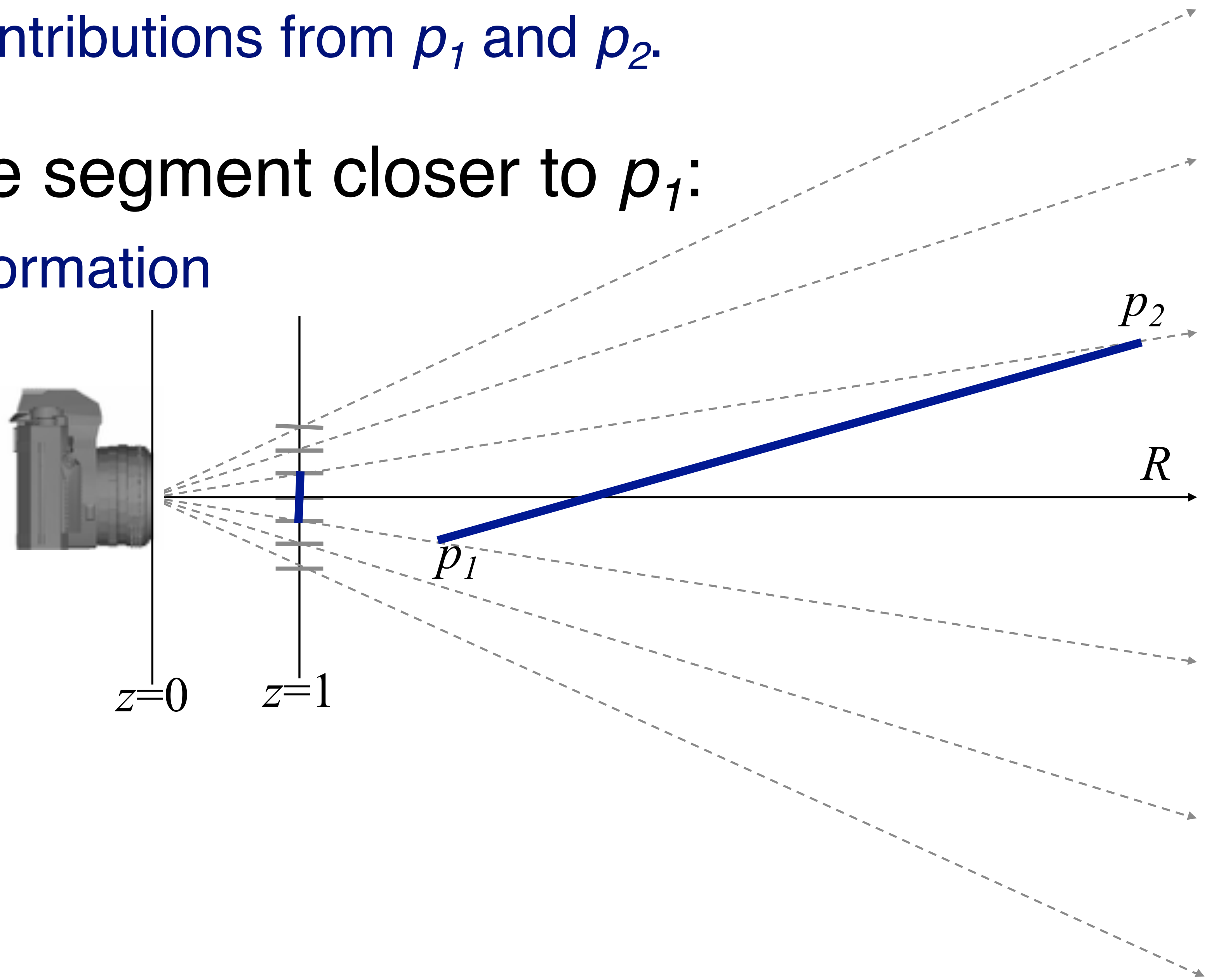
- *R* intersects the projected line segment in the middle:
  - We should use equal contributions from p_1 and p_2.



Scan Conversion Example

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

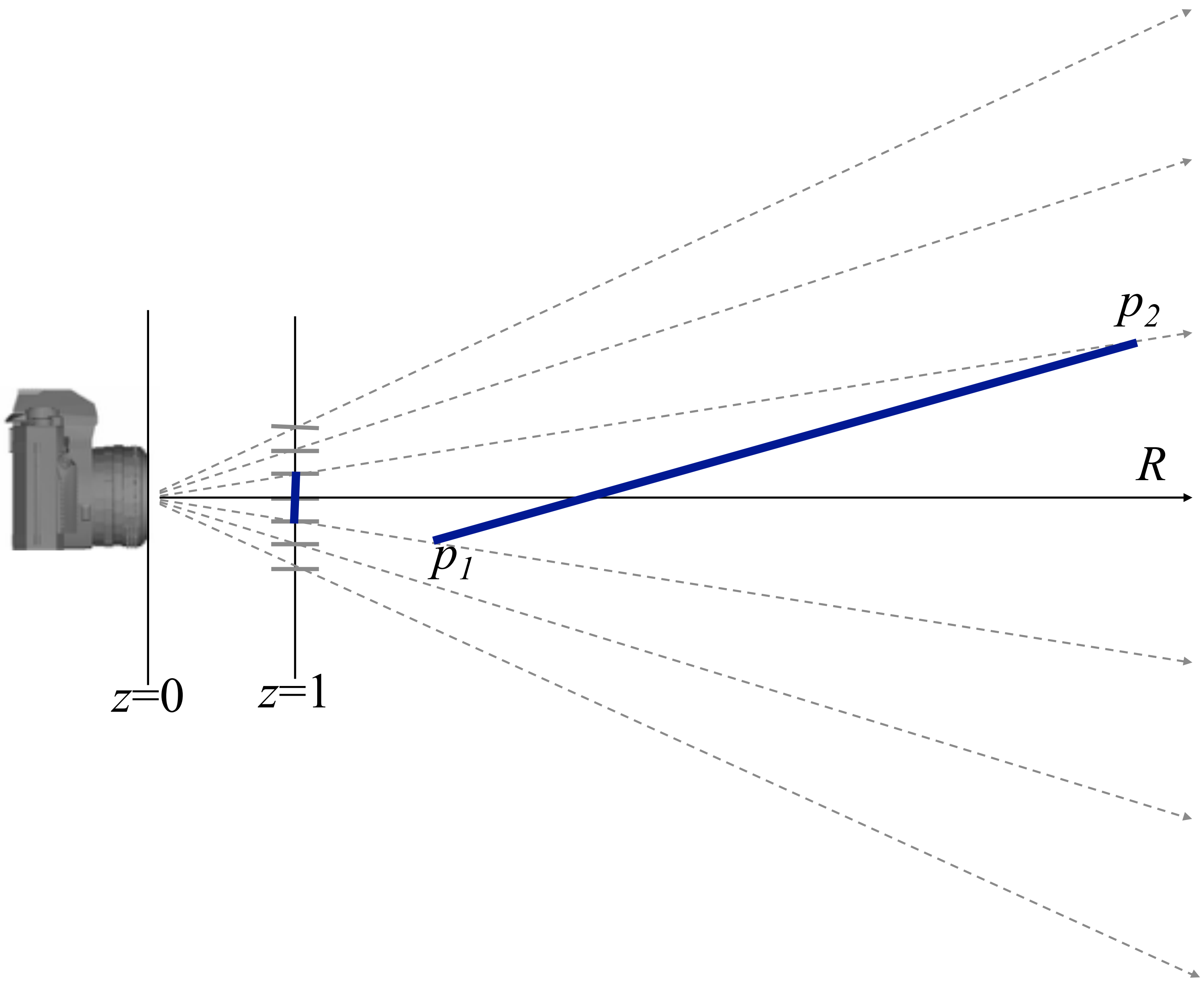
- *R* intersects the projected line segment in the middle:
  - We should use equal contributions from p_1 and p_2.
- *R* intersects the 2D line segment closer to p_1:
  - We should use more information from p_1 than from p_2.



Scan Conversion Example

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

- How do we interpolate correctly?

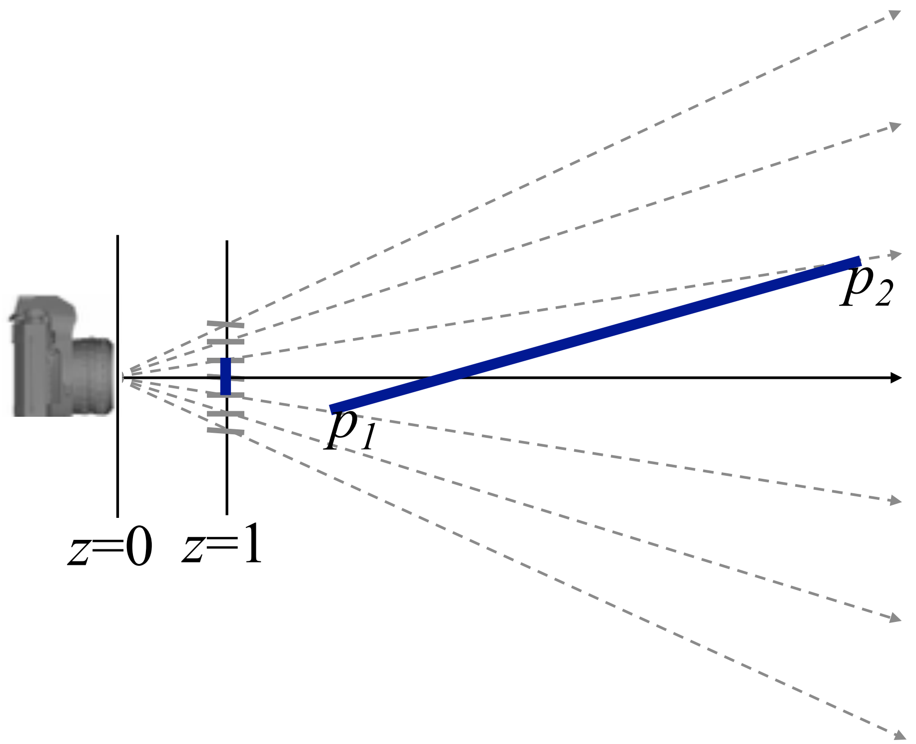


Scan Conversion Example

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

- How do we interpolate correctly?

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



Scan Conversion Example

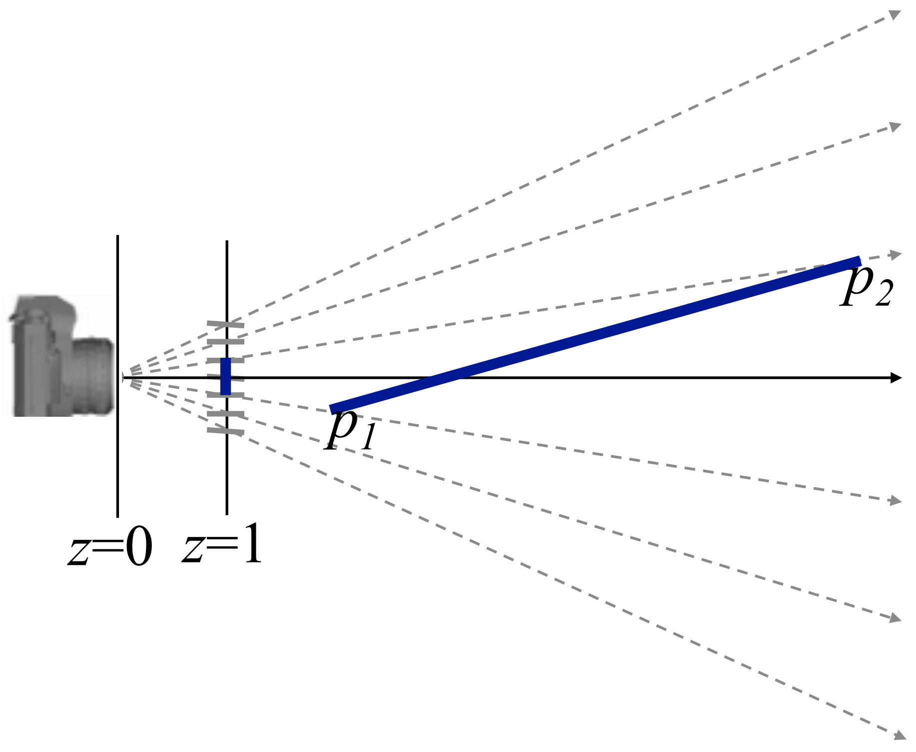
A line segment in 2D projected onto a 1D screen.

- How do we interpolate correctly?

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

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

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



Scan Conversion Example

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

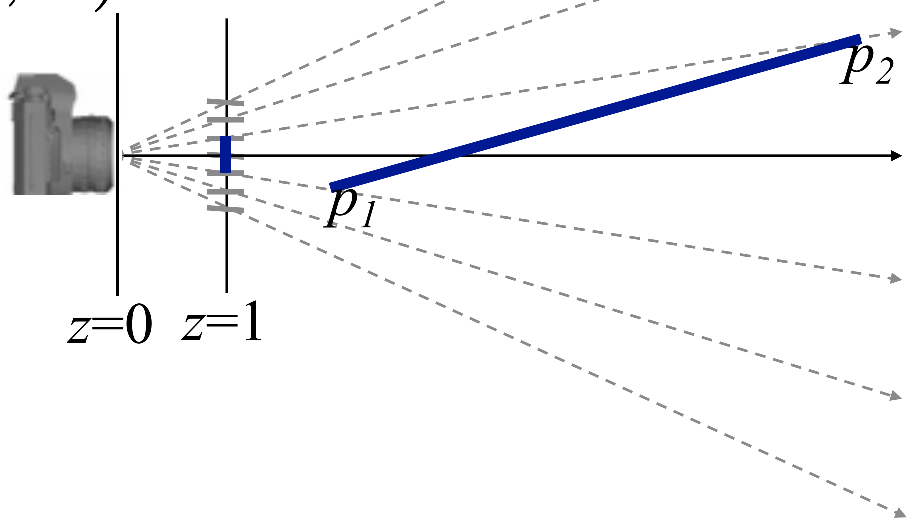
- How do we interpolate correctly?

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

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

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

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



Scan Conversion Example

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

- How do we interpolate correctly?

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

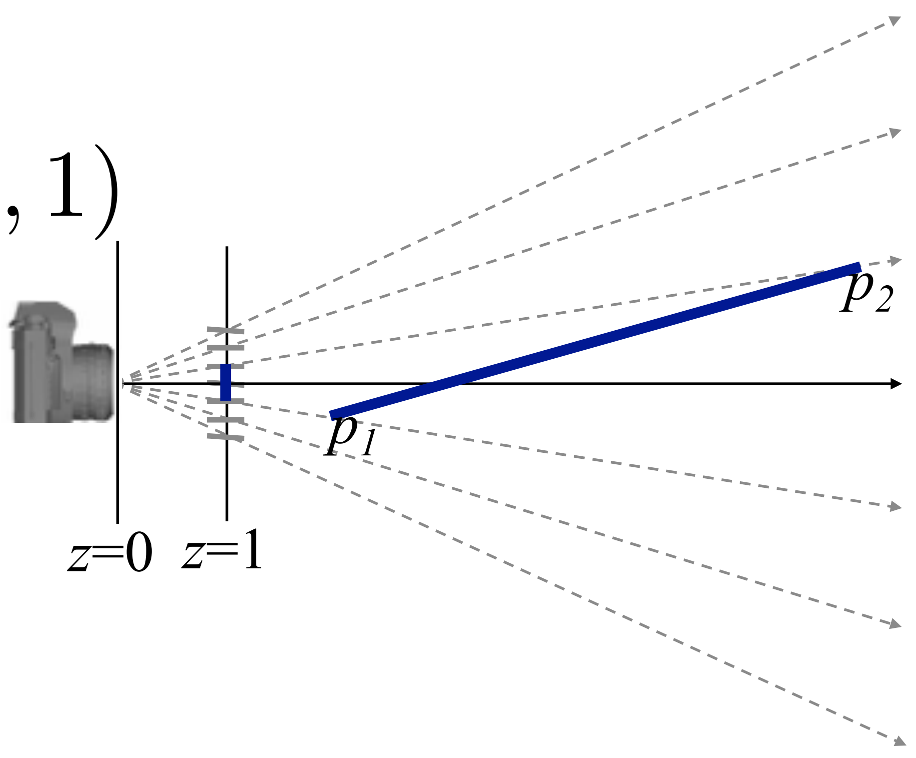
If $p_1=(x_1,z_1)$ and $p_2=(x_2,z_2)$, to find the blending value for a pixel at position x in the screen we need to solve for α s. t.:

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

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

$$\Updownarrow$$

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



Scan Conversion Example

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

- How do we interpolate correctly?

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

If *p*

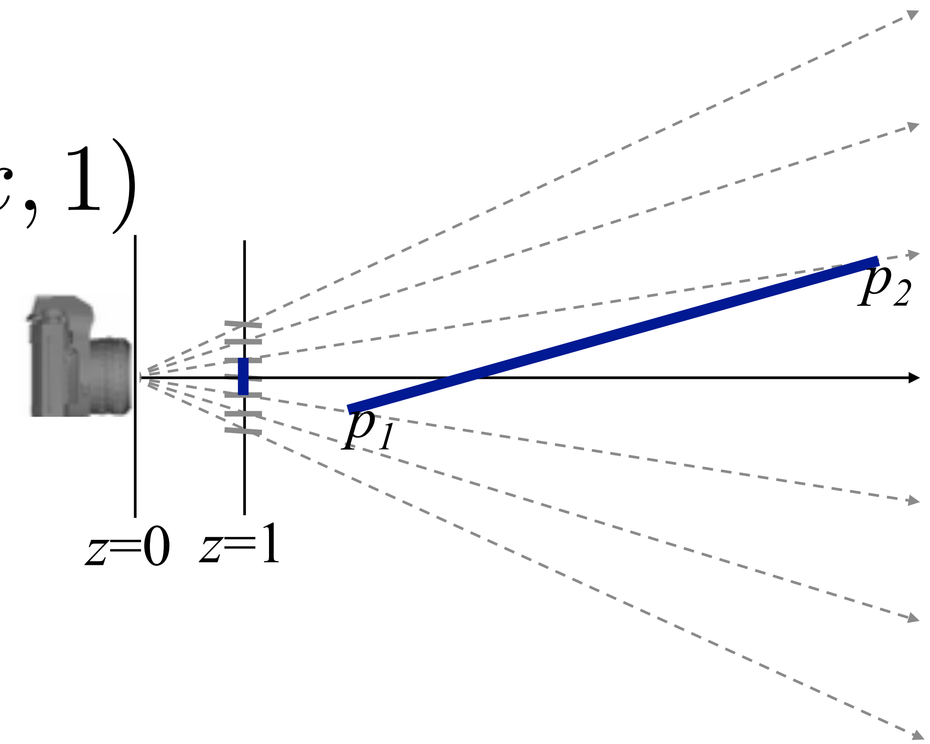
To compute the interpolation weights correctly, we need to perform a perspective divide!

pixel at position x in the screen we need to solve for α s. t.:

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

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

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



Scan Conversion Example

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

- How do we interpolate correctly?

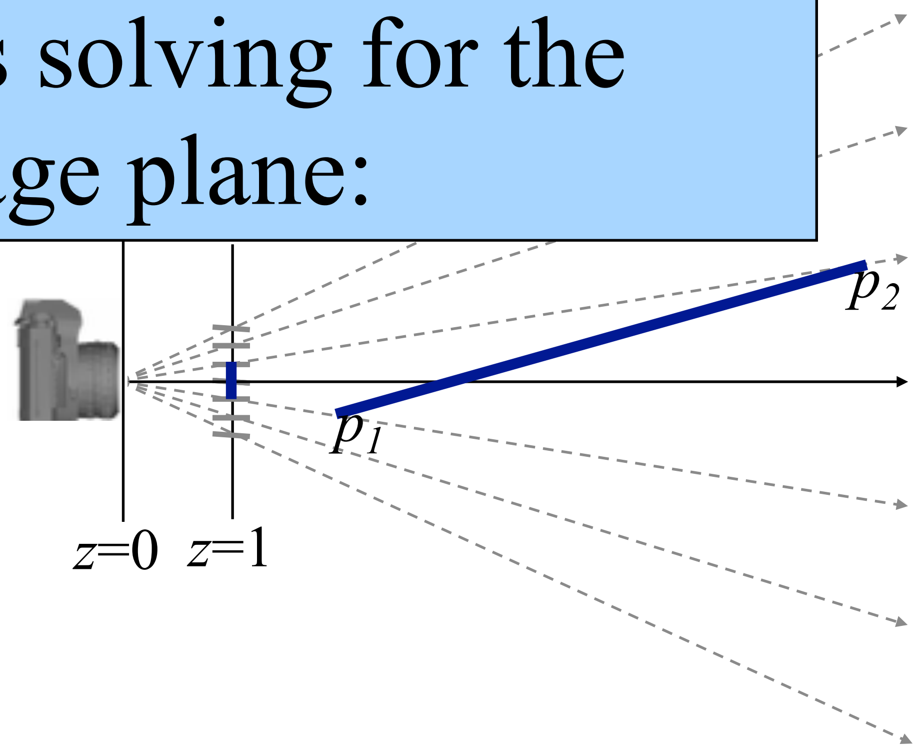
<u>Recall</u>: The 2D point (x, z) maps to the point (x/z) in 1D

If p

To compute the interpolation weights correctly, we need to perform a <u>perspective divide</u>!

pixel at position x in the screen we need to solve for α s. t.:

Note that this is not the same as solving for the blending value in the image plane:



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