CS559: Computer Graphics

Lecture 12: Antialiasing & Visibility

Li Zhang

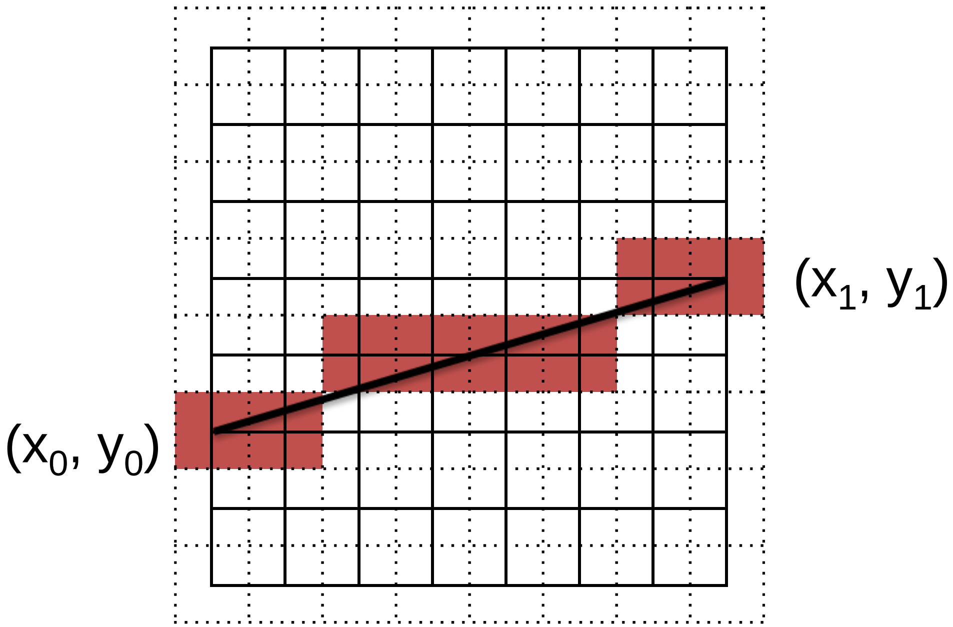
Spring 2008

Today

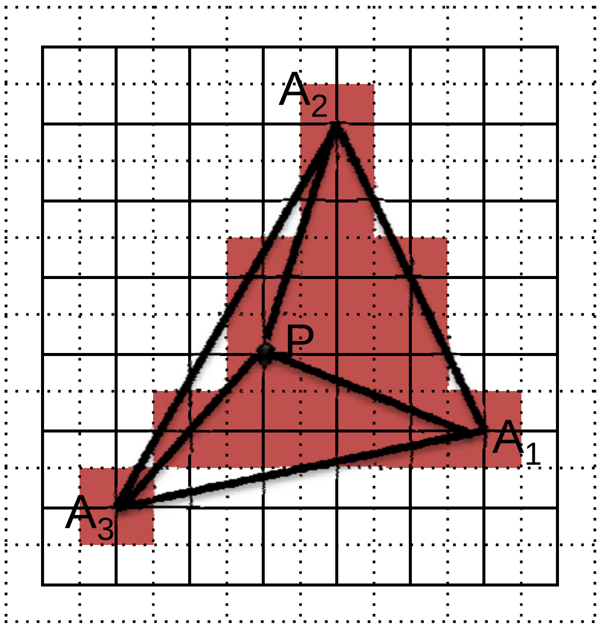
- Antialising
- Hidden Surface Removal

- Reading:
  - Shirley ch 3.7, 8
  - OpenGL ch 1

Last time



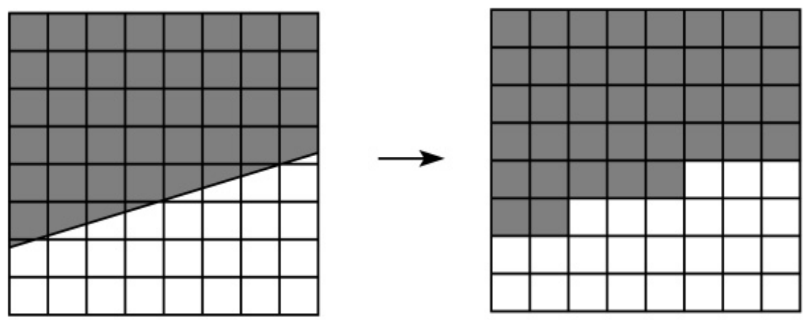
Line drawing



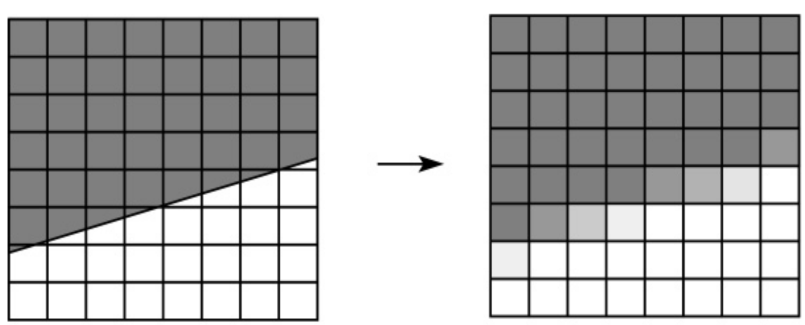
Triangle filling

Aliasing in rendering

- One of the most common rendering artifacts is the “jaggies”. Consider rendering a white polygon against a black background:

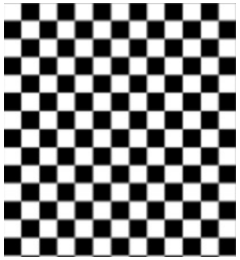


- We would instead like to get a smoother transition:

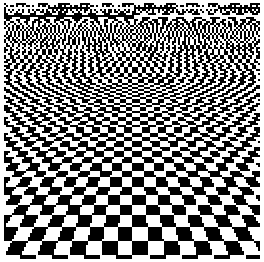


Other types of Aliasing

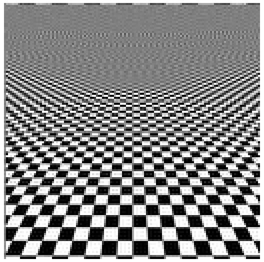
- Image warping



Original



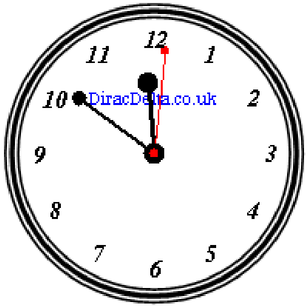
Aliased



Anti-Aliased

Images from answers.com

- Motion Aliasing



•If you were to only **look at the clock every 50 minutes** then the minute hand would appear to rotate anticlockwise.
•The hour hand would still rotate in the correct direction as you have satisfied nyquist.
•The second hand would jitter around depending on how accurate you were with your observations.

http://www.diracdelta.co.uk/science/source/a/l/aliasing/source.html

Anti-aliasing

- **Q**: How do we avoid aliasing artifacts?

1. Sampling:

   Increase sampling rate -- not practical for fixed resolution display.

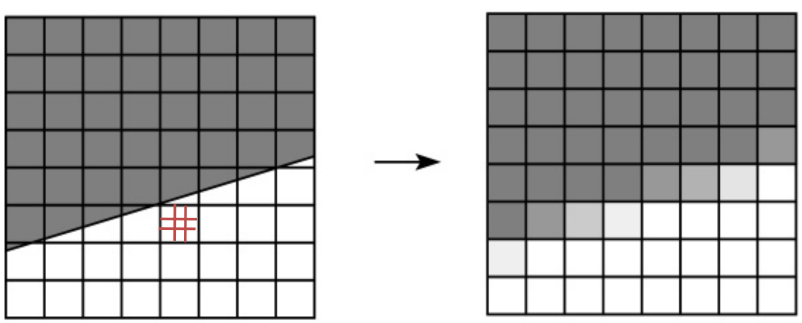
2. Pre-filtering:

   Smooth out high frequences analytically. Requires an analytic function.

3. Combination:

   Supersample and average down.

- Example - polygon:



Memory requirement?

Anti-aliasing

- **Q**: How do we avoid aliasing artifacts?

1. Sampling:

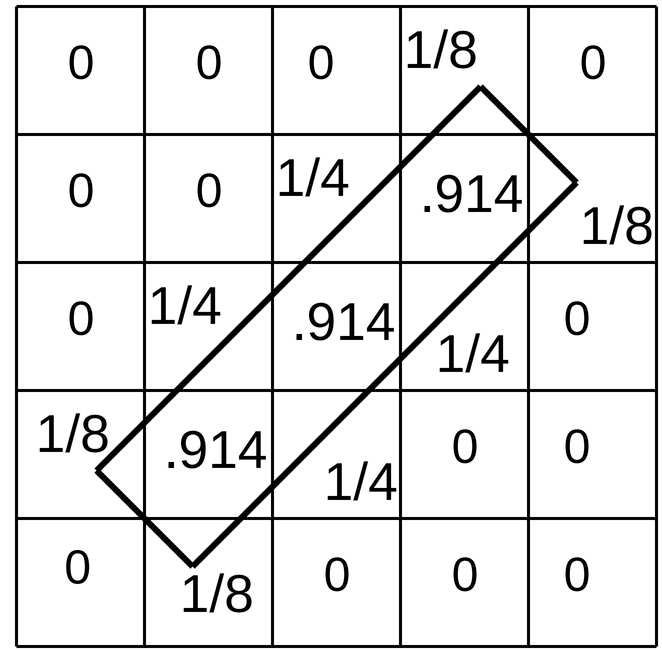
   Increase sampling rate -- not practical for fixed resolution display.

2. Pre-filtering:

   Smooth out high frequences analytically. Requires an analytic function.

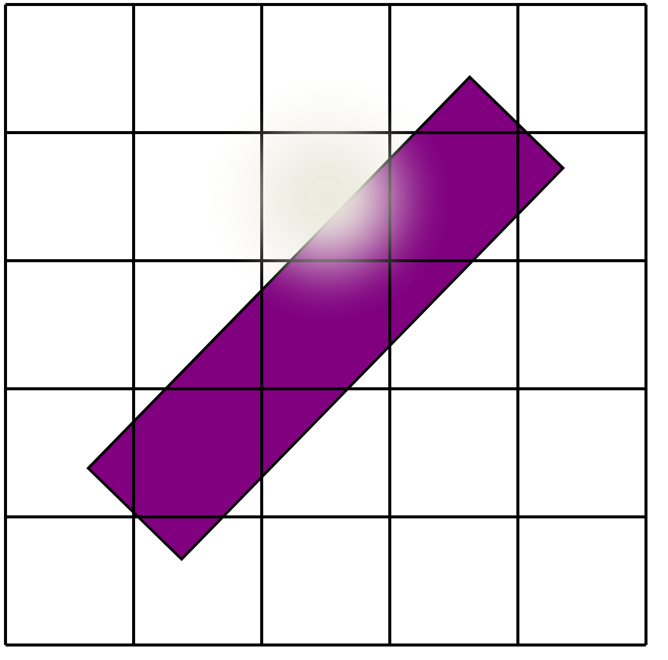
Box filter

- Consider a line as having thickness (all good drawing programs do this)
- Consider pixels as little squares
- Set brightness according to the proportion of the square covered by the line



Weighted Sampling

- Place the “filter” at each pixel, and integrate product of pixel and line
- Common filters are Gaussians



Anti-aliasing

- **Q**: How do we avoid aliasing artifacts?

1. Sampling:

   Increase sampling rate -- not practical for fixed resolution display.

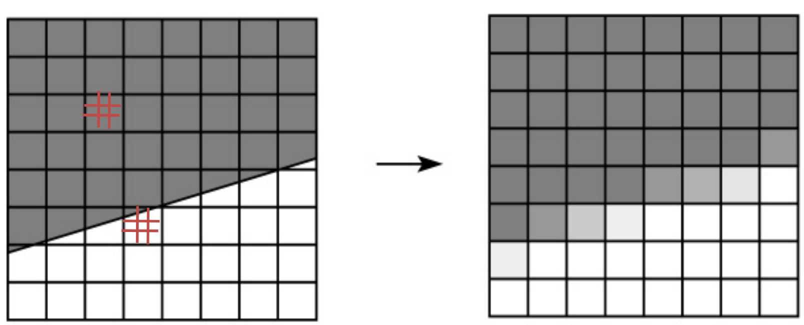
2. Pre-filtering:

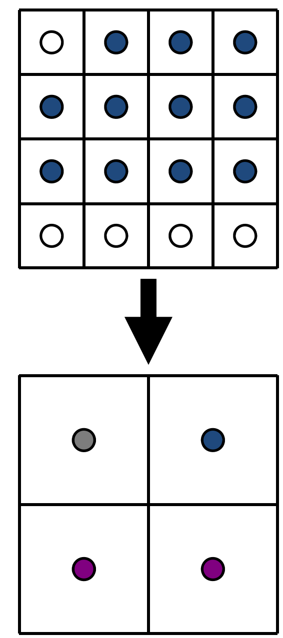
   Smooth out high frequences analytically. Requires an analytic function.

3. Combination:

   Supersample and average down.

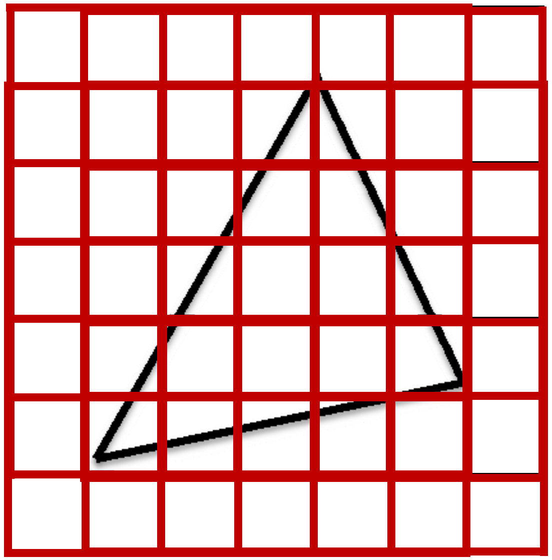
- Example - polygon:





Memory requirement?

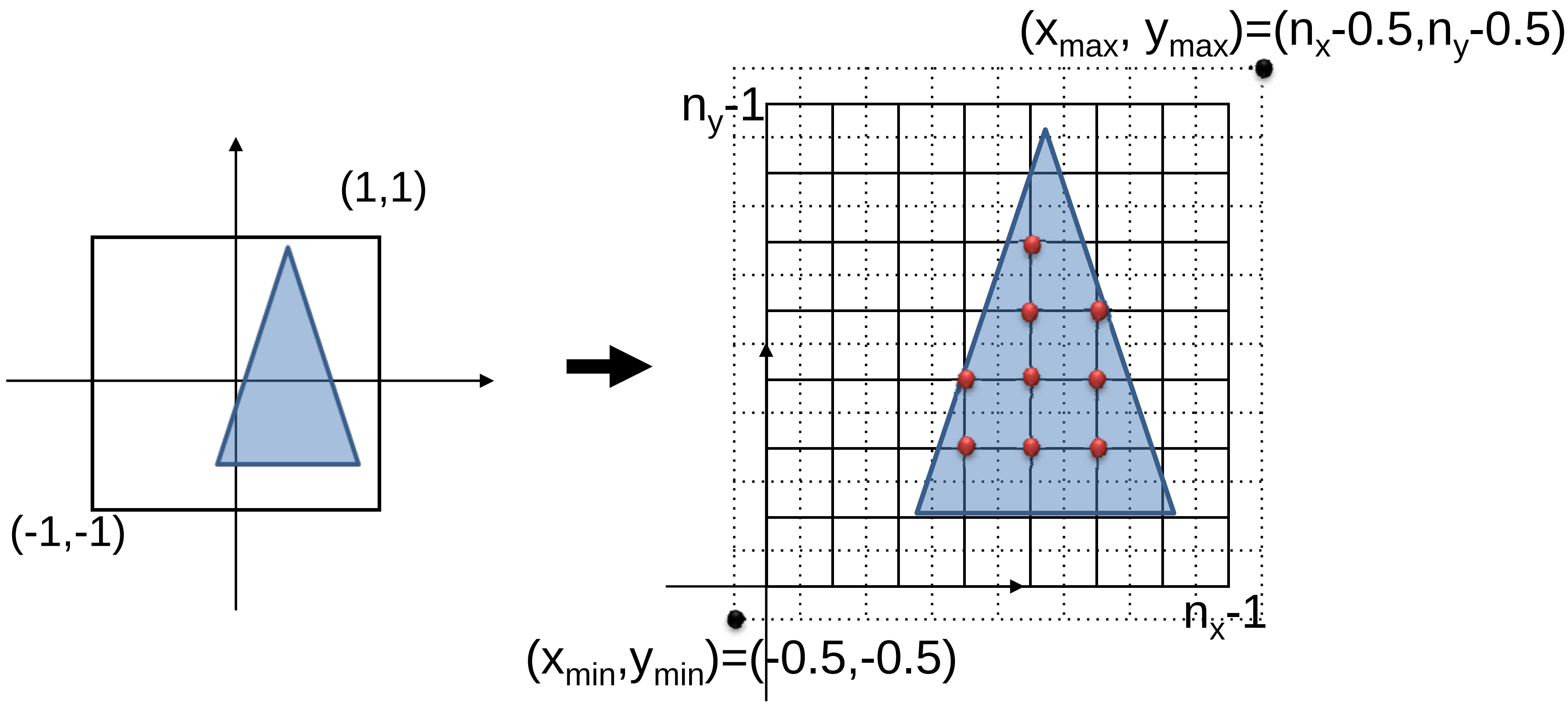
Implementing antialiasing



Assuming this is a 2X supersampling grid, how to achieve anti-aliasing without using 4X memory?

Rasterize shifted versions of the triangle on the original grid, accumulate the color, and divide the final image by the number of shifts

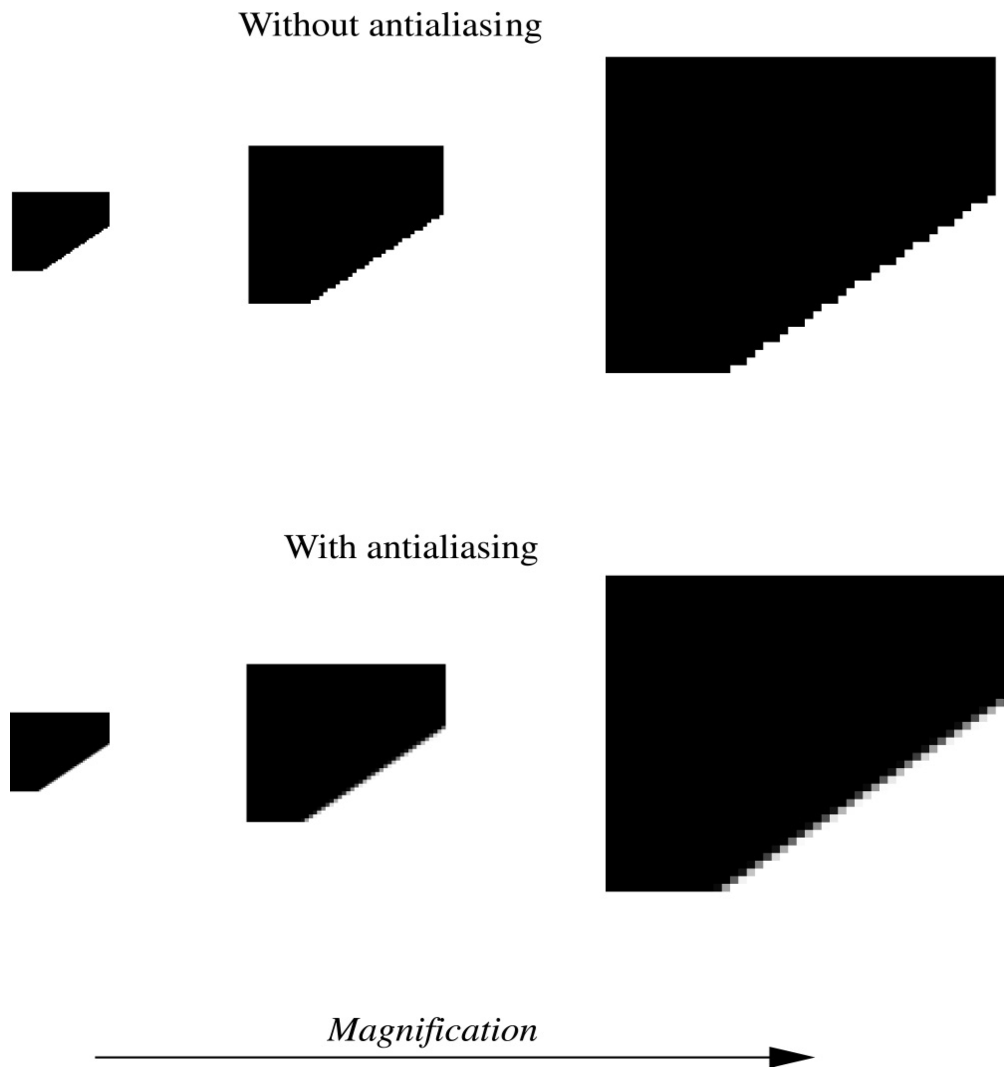
Canonical → Window Transform



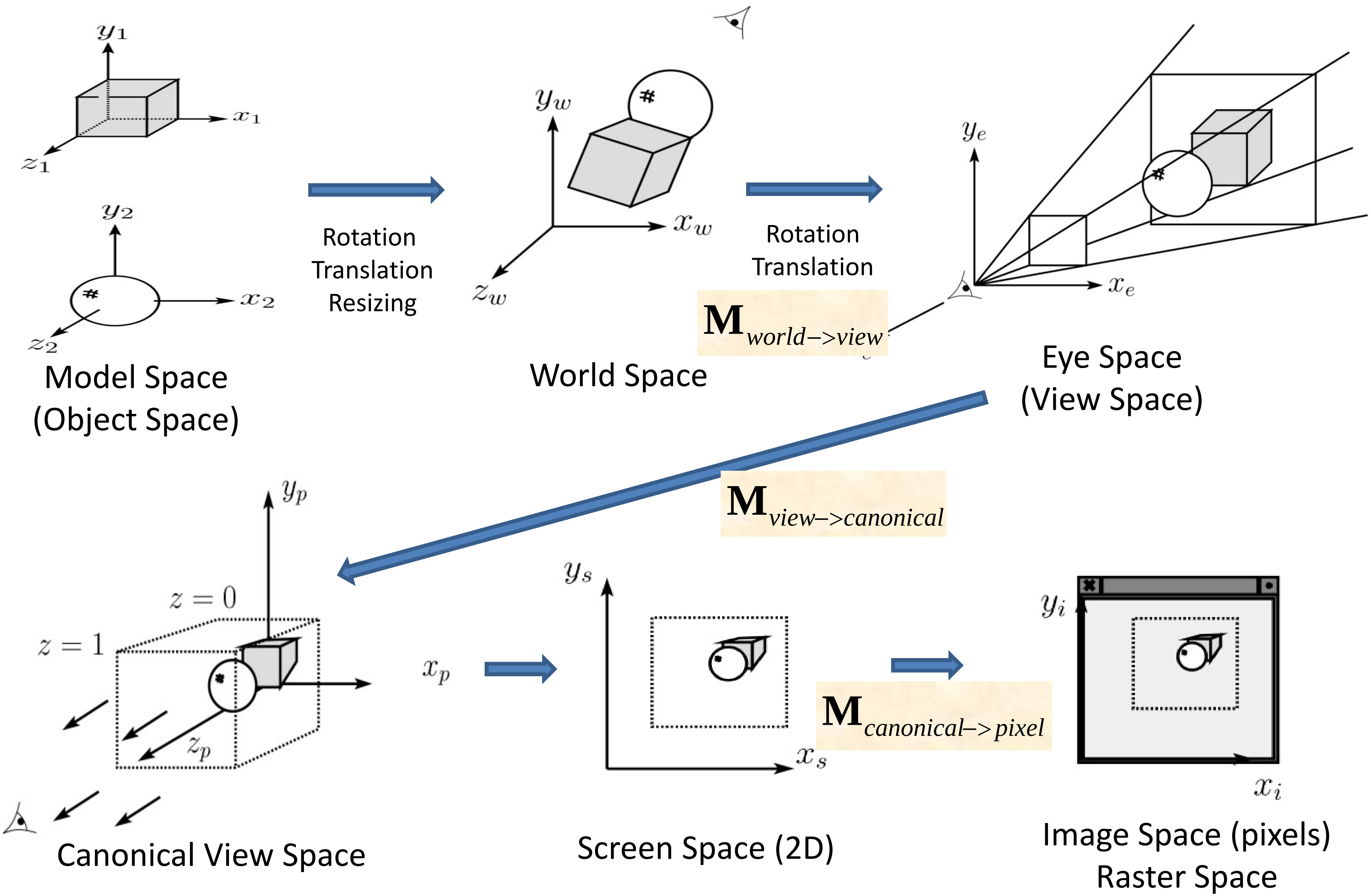
$$\begin{bmatrix} x_{pixel} \\ y_{pixel} \\ z_{pixel} \\ 1 \end{bmatrix} = \begin{bmatrix} (x_{\max} - x_{\min})/2 & 0 & 0 & (x_{\max} + x_{\min})/2 \\ 0 & (y_{\max} - y_{\min})/2 & 0 & (y_{\max} + y_{\min})/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{canonical} \\ y_{canonical} \\ z_{canonical} \\ 1 \end{bmatrix}$$

$$\mathbf{M}_{canonical->pixel}$$

Polygon anti-aliasing



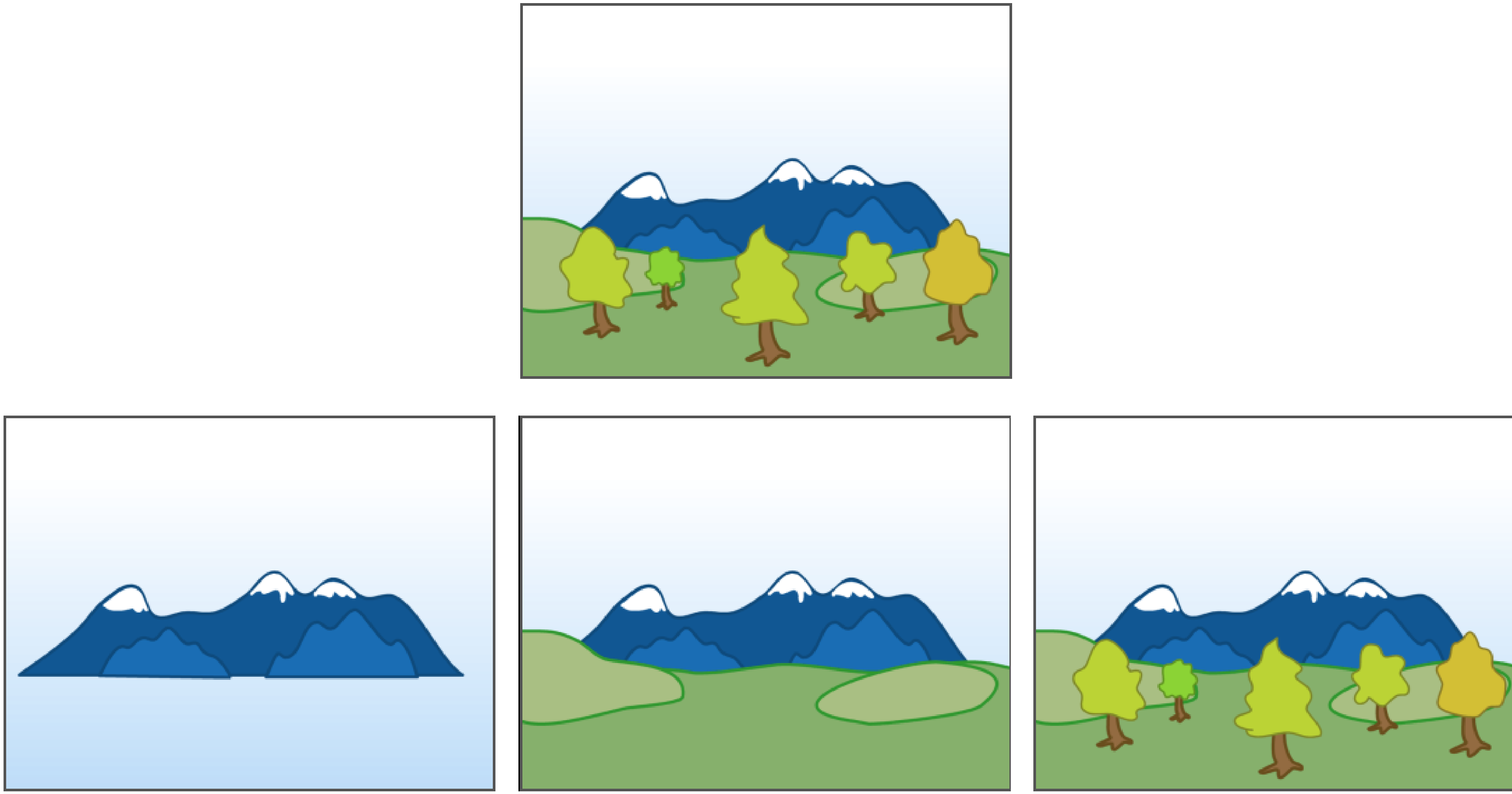
3D Geometry Pipeline



Visibility

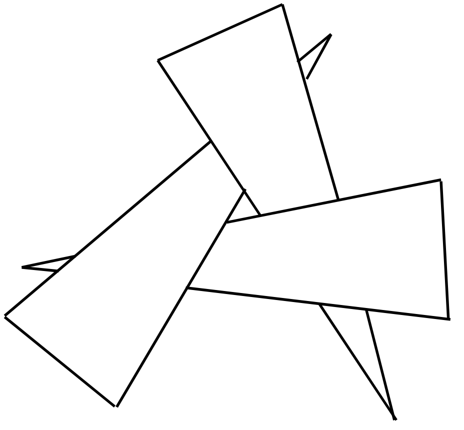
- Given a set of polygons, which is visible at each pixel? (in front, etc.). Also called *hidden surface removal*
- Very large number of different algorithms known. Two main classes:
  - Object precision
    - computations that operate on primitives
      - triangle A occludes triangle B
  - Image precision
    - computations at the pixel level
      - pixel P sees point Q

Painter's Algorithm



Draw objects in a back-to-front order

Painter's algorithm



Failure case

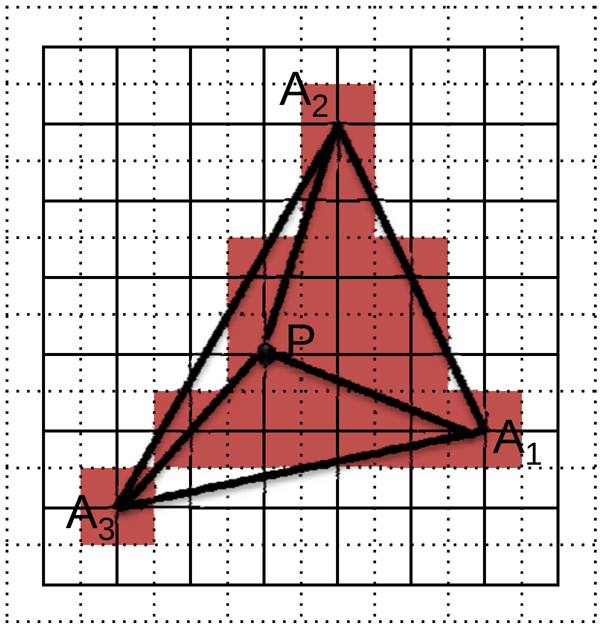
Z-buffer (image precision)

- The **Z-buffer** or **depth buffer** algorithm [Catmull, 1974] is probably the simplest and most widely used.
- For each pixel on screen, have at least two buffers
  - Color buffer stores the current color of each pixel
    - The thing to ultimately display
  - Z-buffer stores at each pixel the depth of the **nearest thing seen so far**
    - Also called the depth buffer

Z-buffer

- Here is pseudocode for the Z-buffer hidden surface algorithm:

for each pixel (*i,j*) **do**
    Z-buffer [*i,j*] ← *FAR*
    Framebuffer[*i,j*] ← <background color>
end for
for each polygon *A* **do**
    for each pixel in *A* **do**
        Compute depth *z* and shade *s* of *A* at (*i,j*)
        if z > *Z-buffer* [*i,j*] **then**
            Z-buffer [*i,j*] ← *z*
            Framebuffer[*i,j*] ← *s*
        end if
    end for
end for



Triangle filling

How to compute shades/color?

How to compute depth z?

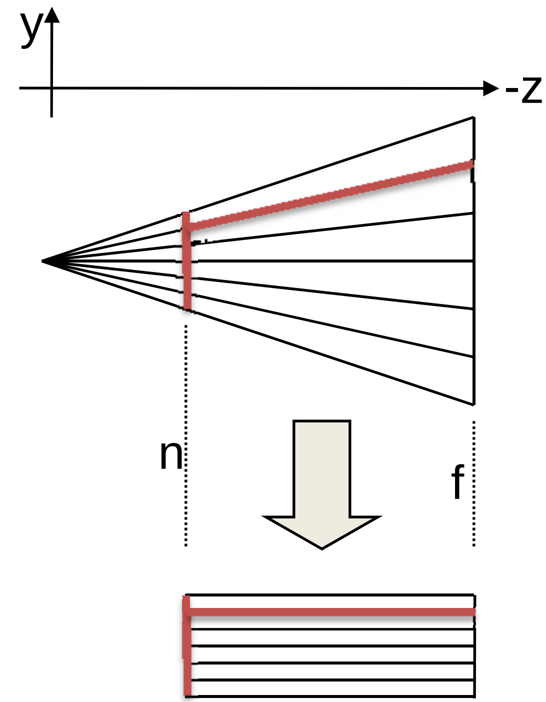
Precision of depth

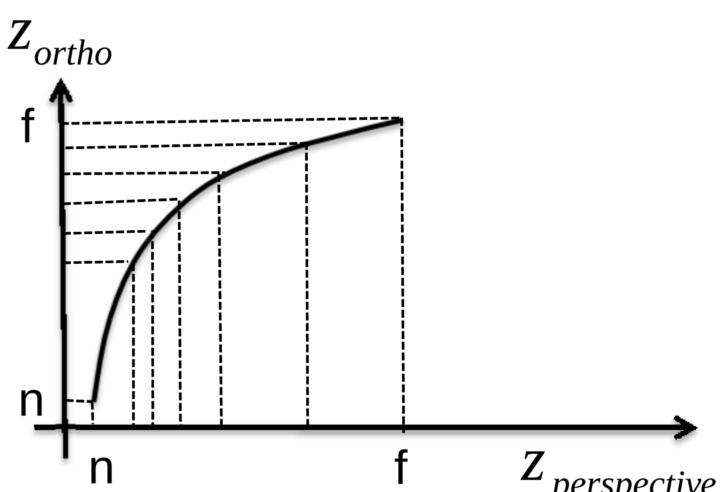
$$z_{ortho} = f + n - \frac{fn}{z_{perspective}}$$

$$\Delta z_{ortho} \approx \frac{fn}{z_{perspective}^2} \Delta z_{perspective}$$

$$\Delta z_{perspective} \approx \frac{z_{perspective}^2}{fn} \Delta z_{ortho}$$

$$\Delta z_{perspective}^{max} \approx \frac{f}{n} \Delta z_{ortho}$$





- Depth resolution not uniform
- More close to near plane, less further away
- Common mistake: set near = 0, far = infty. Don't do this. Can't set near = 0; lose depth resolution.

Other issues of Z buffer

- Advantages:
  - Simple and now ubiquitous in hardware
    - A z-buffer is part of what makes a graphics card "3D"
  - Computing the required depth values is simple
- Disadvantages:
  - Depth quantization errors can be annoying
  - Can't easily do transparency

$$(\alpha_1 I_1 \ \text{ over } \ \alpha_2 I_2) \ \text{ over } \ \alpha_3 I_3$$

$$(\alpha_1 I_1 \ \text{ over } \ \alpha_3 I_3) \ \text{ over } \ \alpha_2 I_2$$

The A-buffer (Image Precision)

- Handles transparent surfaces and filter anti-aliasing
- At each pixel, maintain a pointer to a **list** of polygons sorted by depth

The A-buffer (Image Precision)

for each pixel (i,j) **do**

  Z-buffer $[i,j] \leftarrow$ *FAR*

  Framebuffer$[i,j] \leftarrow$ <background color>

end for

for each polygon *A* **do**

  for each pixel in *A* **do**

    Compute depth *z* and shade *s* of *A* at (i,j)

    ~~**if** $z >$ *Z-buffer* $[i,j]$ **then**~~

      ~~*Z-buffer* $[i,j] \leftarrow z$~~

      ~~*Framebuffer*$[i,j] \leftarrow s$~~

    ~~**end if**~~

  end for

end for

→

if polygon is opaque and covers pixel, insert into list, removing all polygons farther away

if polygon is transparent, insert into list, but don't remove farther polygons

A-Buffer Composite

For each pixel, we have a list of

$$(\alpha_1, I_1, z_1)\,(\alpha_2, I_2, z_2)\cdots(\alpha_N, I_N, z_N)$$

$$composite\{(\alpha_1, I_1, z_1)\,(\alpha_2, I_2, z_2)\cdots(\alpha_N, I_N, z_N)\}$$

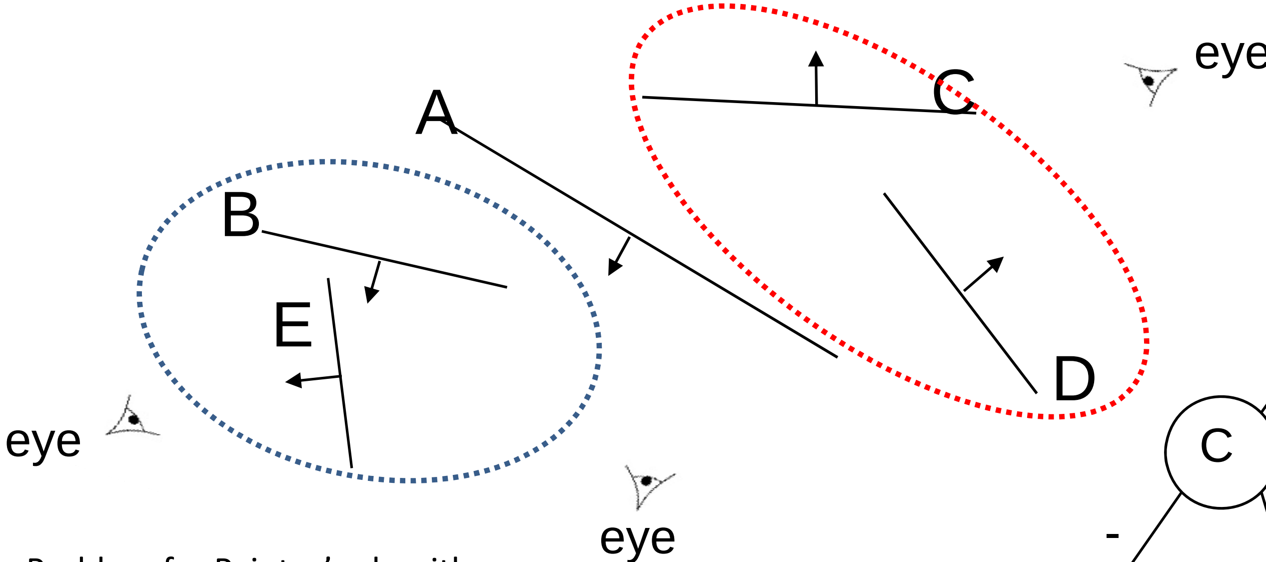
$$= composite\{(\alpha_1, I_1, z_1), composite\{(\alpha_2, I_2, z_2)\cdots(\alpha_N, I_N, z_N)\}\}$$

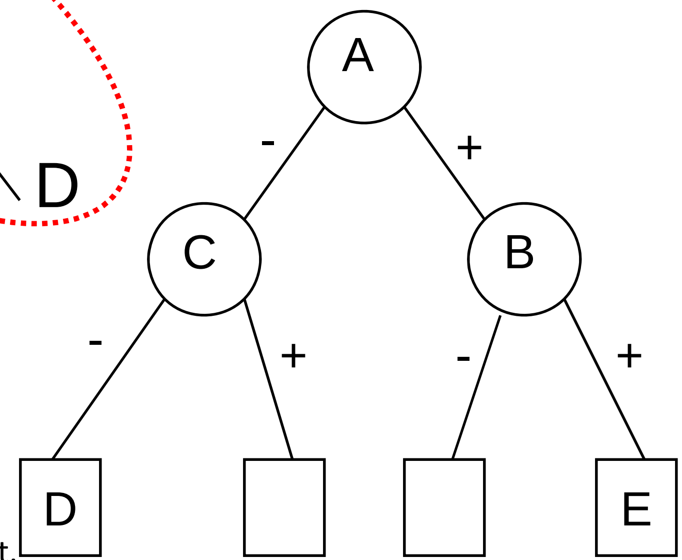
$$= \alpha_1 I_1 + (1-\alpha_1)\left(\alpha_2 I_2 + (1-\alpha_2)\left(\alpha_3 I_3 + \cdots \alpha_N I_N\right)\right)$$

The A-buffer (2)

- Advantage:
  - Can do more than Z-buffer
  - Alpha can represent partial coverage as well
- Disadvantages:
  - Not in hardware, and slow in software
  - Still at heart a z-buffer: depth quantization problems
- But, used in high quality rendering tools

Binary-space partitioning (BSP) trees





- <u>Problem for Painter's algorithm:</u>
  - Order is view dependent

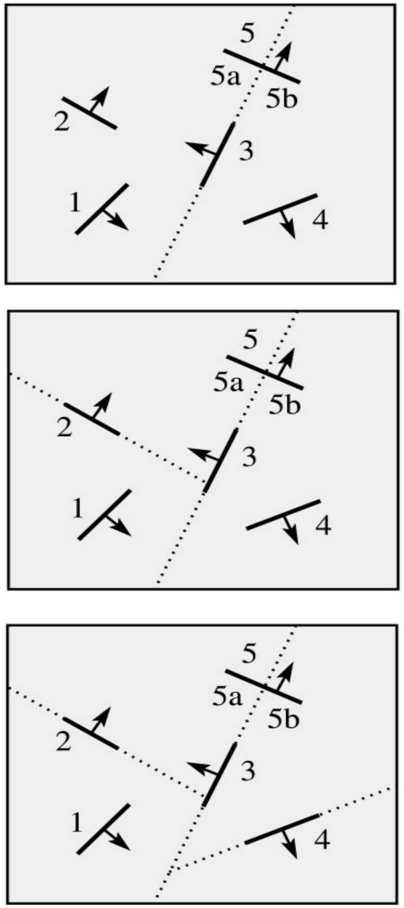
- <u>Idea:</u>
  - Do extra preprocessing to allow quick display from <u>any</u> viewpoint.

- <u>Key observation:</u> A polygon *A* is painted in correct order if
  - Polygons on far side of *A* are painted first
  - *A* is painted next
  - Polygons on near side of *A* are painted last.

- Solution: build a tree to recursively partition the space and group polygons

- Why it works? What's the assumption?

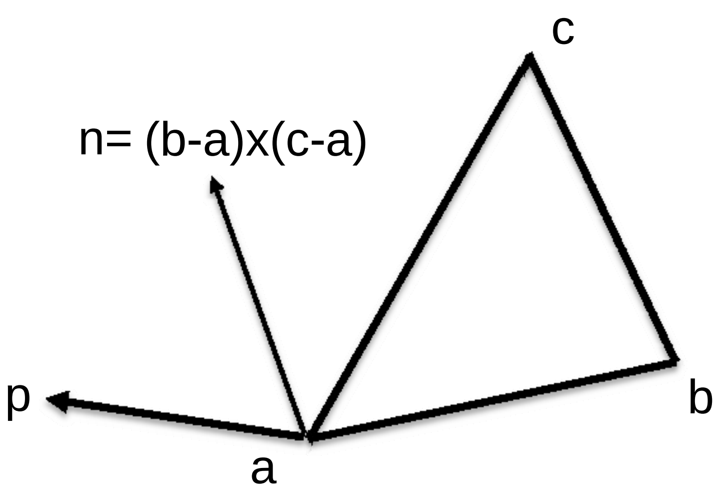
BSP tree creation



- **procedure** *MakeBSPTree*:
- **takes** *PolygonList L*
- **returns** *BSPTree*
- Choose polygon *A* from *L* to serve as root
- Split all polygons in *L* according to *A*
- node ← *A*
- *node.neg* ← *MakeBSPTree*(Polygons on neg. side of A)
- *node.pos* ← *MakeBSPTree*(Polygons on pos. side of A)
- **return** node
- **end** procedure

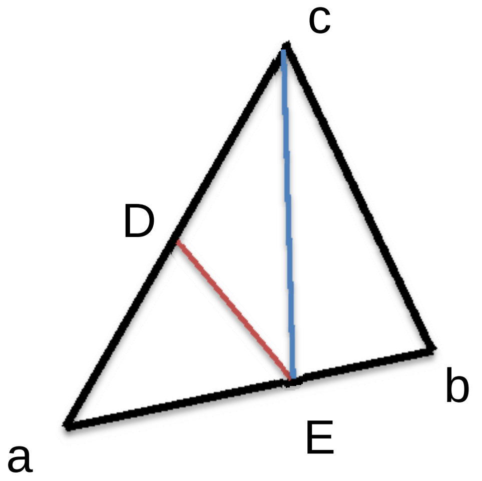
Plane equation

Plane equation: $f(p) = n^T(p-a)$



Positive side $f(p) > 0$
Negative side $f(p) < 0$

Split Triangles



abc => aED, Ebc, EcD

BSP tree display

- **procedure** *DisplayBSPTree:*
- **Takes** *BSPTree T*
- **if** *T* is empty **then return**
- **if** viewer is in front (on pos. side) of *T.node*
- *DisplayBSPTree(T. ______)*
- *Draw T.node*
- *DisplayBSPTree(T.______)*
- **else**
- *DisplayBSPTree(T. ______)*
- *Draw T.node*
- *DisplayBSPTree(T. ______)*
- **end if**
- **end procedure**

Performance Notes

- Does how well the tree is balanced matter?
  - No
- Does the number of triangles matter?
  - Yes
- Performance is improved when fewer polygons are split --- in practice, best of ~ 5 random splitting polygons are chosen.
- BSP is created in world coordinates. No projective matrices are applied before building tree.

BSP-Tree Rendering (2)

- Advantages:
  - One tree works for any viewing point
  - transparency works
    - Have back to front ordering for compositing
  - Can also render front to back, and avoid drawing back polygons that cannot contribute to the view
    - Major innovation in *Quake*
- Disadvantages:
  - Can be many small pieces of polygon

3D Geometry Pipeline