Computer Graphics

Lecture 13
Hidden Surface Removal / Transparency

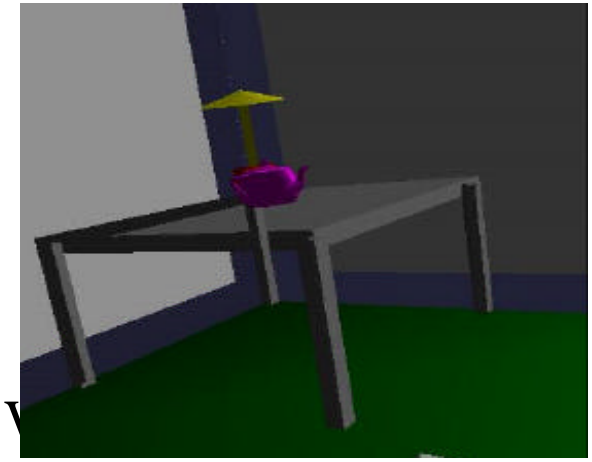
Taku Komura

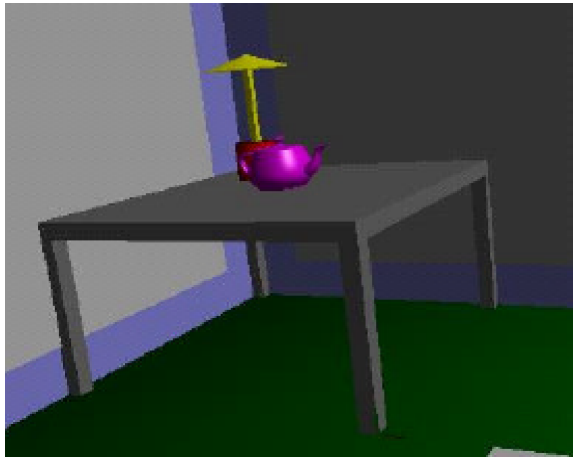
Overview

- Hidden Surface Removal
  - Painter's algorithm
  - Z-buffer
  - BSP tree
  - Portal culling
  - Back face culling
- Transparency
  - Alpha blending
  - Screen door transparency

Why Hidden Surface Removal?

- A correct rendering requires correct visibility calculations
- When multiple opaque polygons cover the same screen space, only the closest one is visible (remove the other hidden surfaces) → Painter's algorithm, Z-buffer





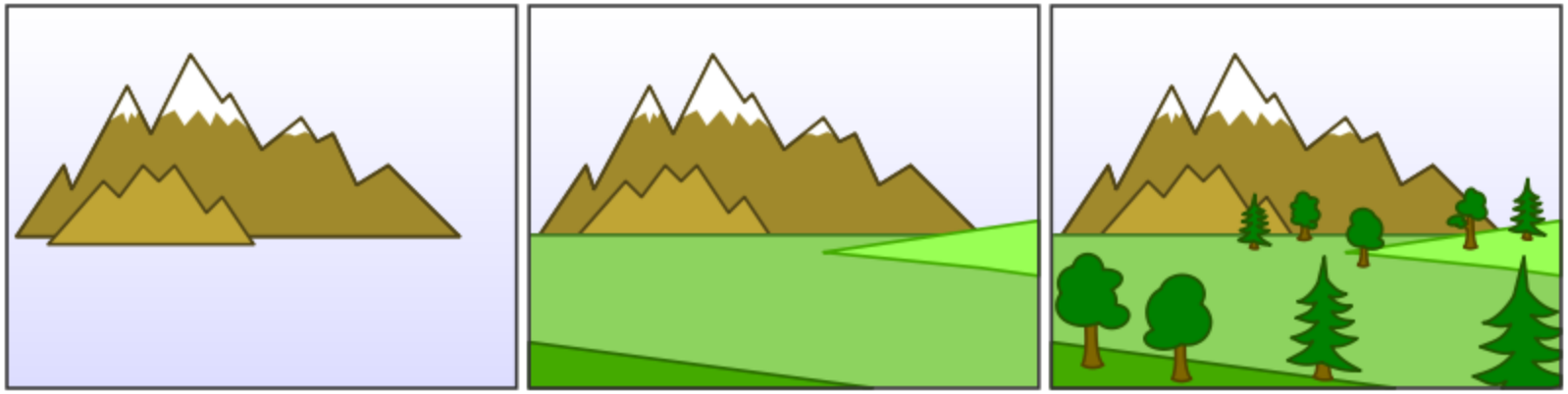
Overview

Hidden Surface Removal
- **Painter's algorithm**
- **Z-buffer**
- BSP tree
- Portal culling
- Back face culling

- Transparency
  - Alpha blending
  - Screen door transparency

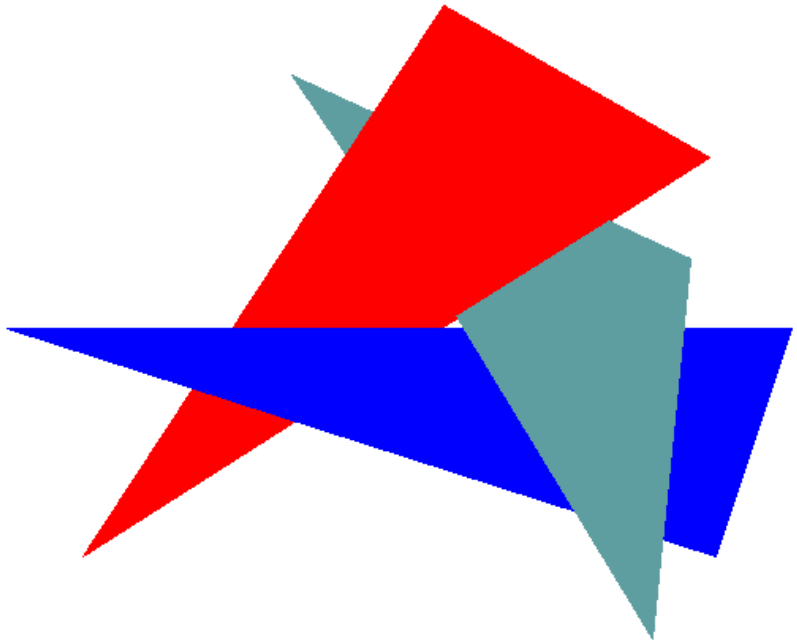
Painters algorithm

- Draw surfaces in back to front order – nearer polygons “paint” over farther ones.
- Need to decide the order to draw – far objects first



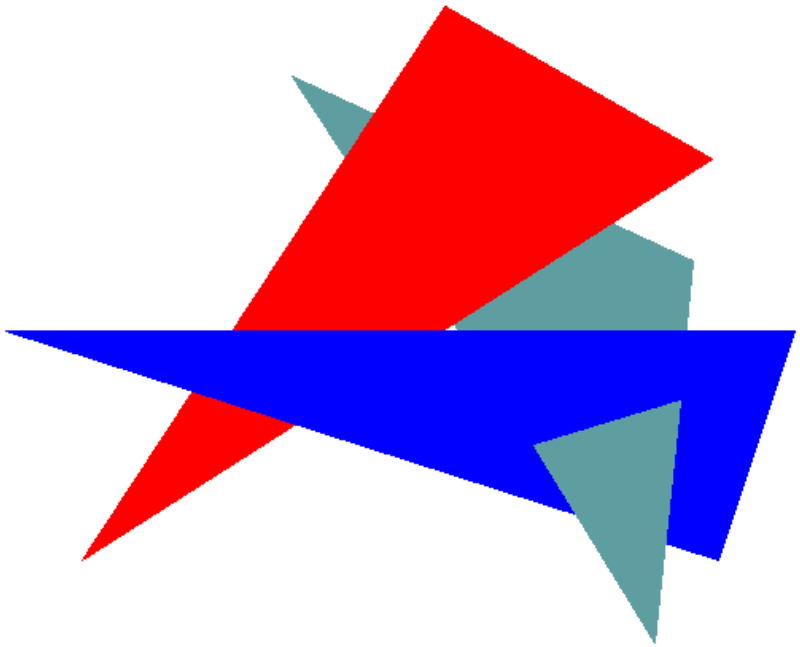
Painters algorithm

- Key issue is order determination.
- Doesn't always work – see image at right.



Painters algorithm

- Another situation it does not work
- In both cases, we need to segment the triangles and make them sortable



Z-buffer

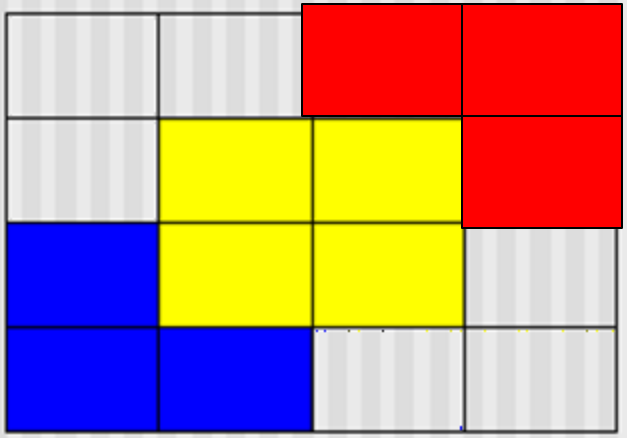
- An image-based method applied during the rasterization stage
- A standard approach implemented in most graphics libraries
- Easy to be implemented on hardware
- By Wolfgang Straßer in 1974

Z-buffer

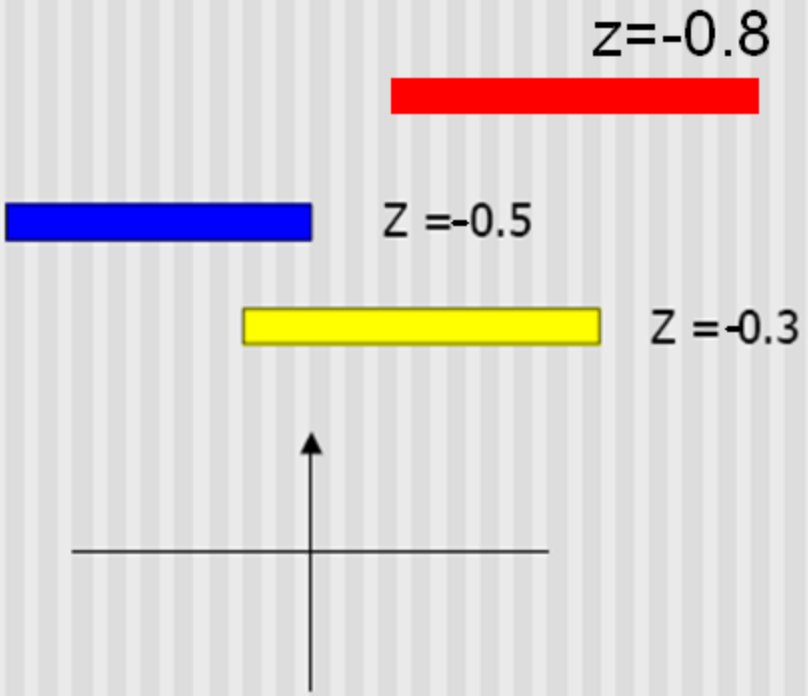
Basic Z-buffer idea:
For every input polygon

- For every pixel in the polygon interior, calculate its corresponding z value (by interpolation)
- Compare the depth value with the closest value from a different polygon (largest z) so far
- Paint the pixel with the color of the polygon if it is closer

Z buffer example



Correct Final image



Top View

Z buffer example

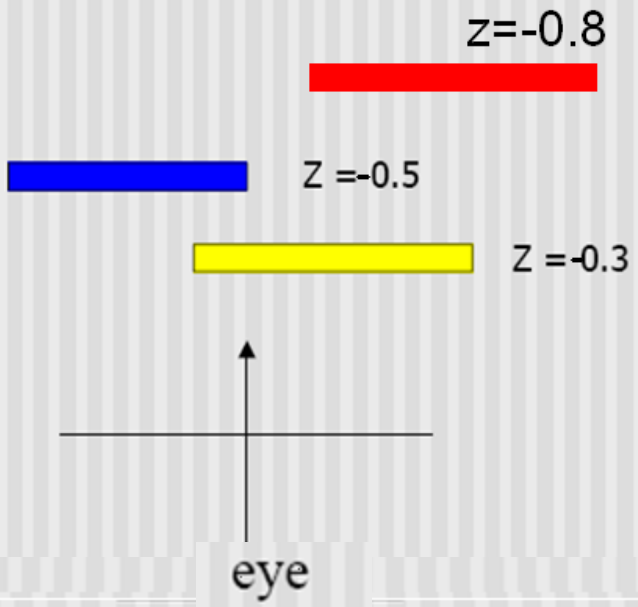
Step 1: Initialize the depth buffer

-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0

Z buffer example

Step 2: Draw the blue polygon (assuming the program draws blue polyon first – the order does not affect the final result any way).

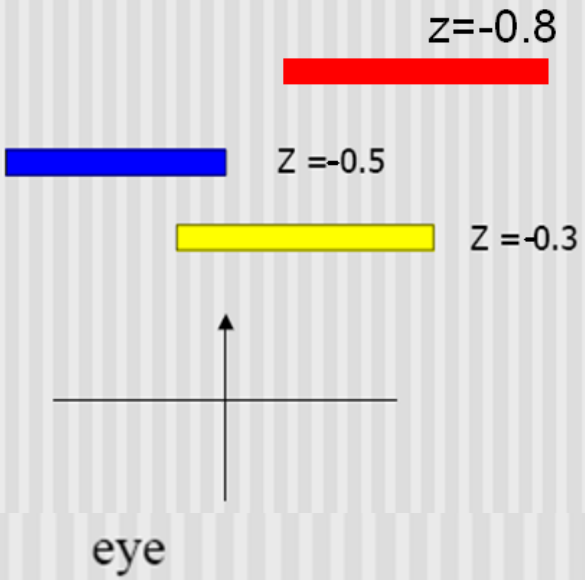
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-0.5	-0.5	-1.0	-1.0
-0.5	-0.5	-1.0	-1.0



Z buffer example

Step 3: Draw the yellow polygon

-1.0	-1.0	-1.0	-1.0
-1.0	-0.3	-0.3	-1.0
-0.5	-0.3	-0.3	-1.0
-0.5	-0.5	-1.0	-1.0

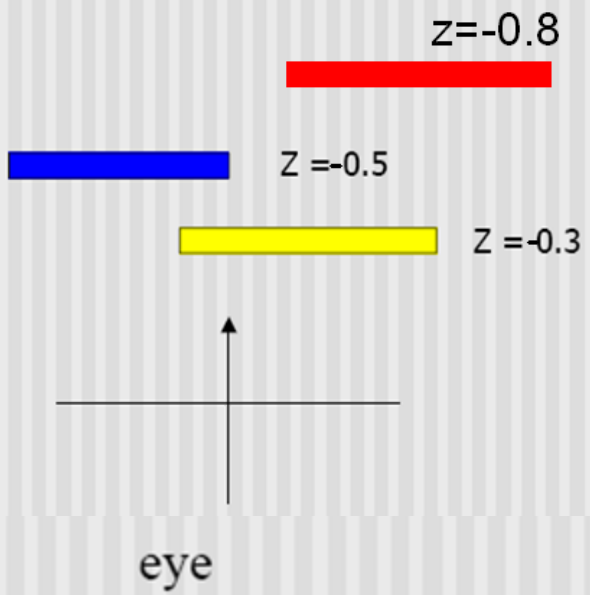


If the depth value is larger than that in the z-buffer, the pixel is coloured and value in the z-buffer is updated

Z buffer example

Step 4: Draw the red polygon

-1.0	-1.0	-0.8	-0.8
-1.0	-0.3	-0.3	-0.8
-0.5	-0.3	-0.3	-1.0
-0.5	-0.5	-1.0	-1.0



If the depth value is larger than that in the z-buffer, the pixel is coloured and value in the z-buffer is updated

Why is Z-buffering so popular ?

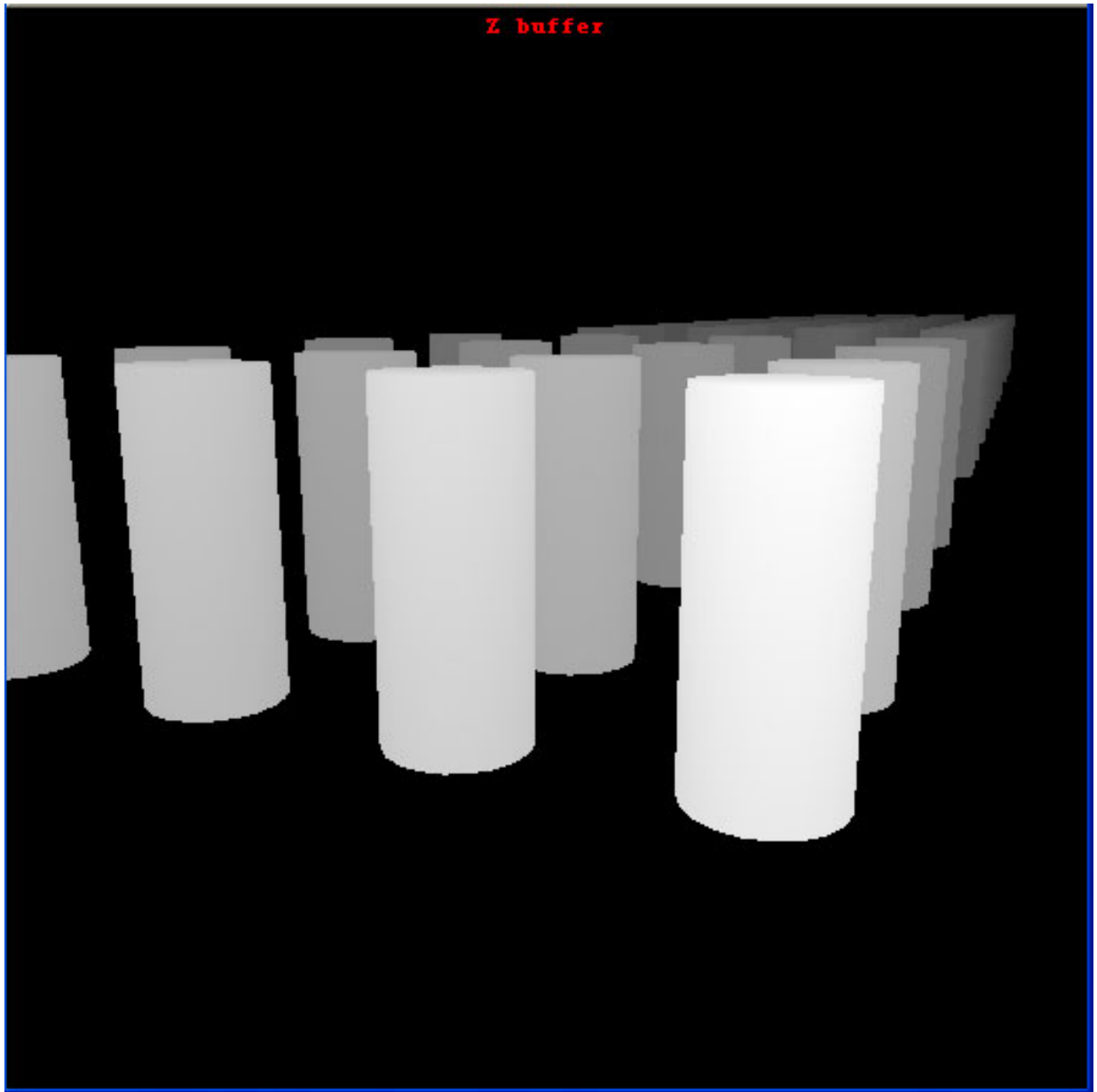
Advantage

- Simple to implement in hardware.
  - Memory for z-buffer is now not expensive
- Diversity of primitives – not just polygons.
- Unlimited scene complexity
- No need to sort the objects
- No need to calculate object-object intersections.

Disadvantage

- Waste time drawing hidden objects
- Z-precision errors (aliasing problems)

Z-buffer aliasing

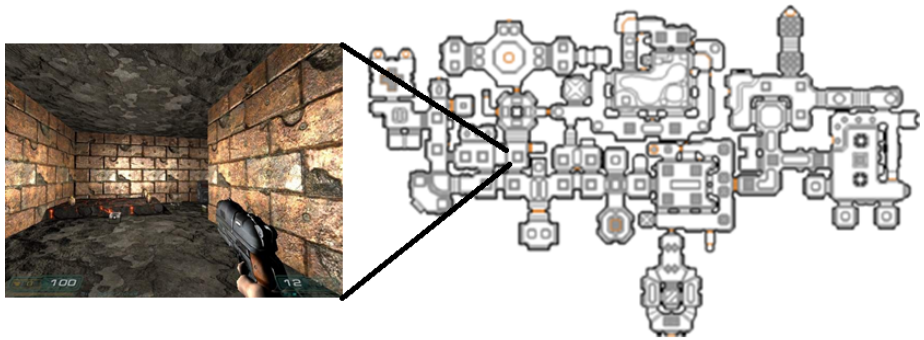


Z-buffer performance

- Memory overhead: O(1)
- Time to resolve visibility to screen precision: O(n)
  - n: number of polygons
  - Need to be combined with other culling methods to reduce complexity

Rendering Complex Scenes

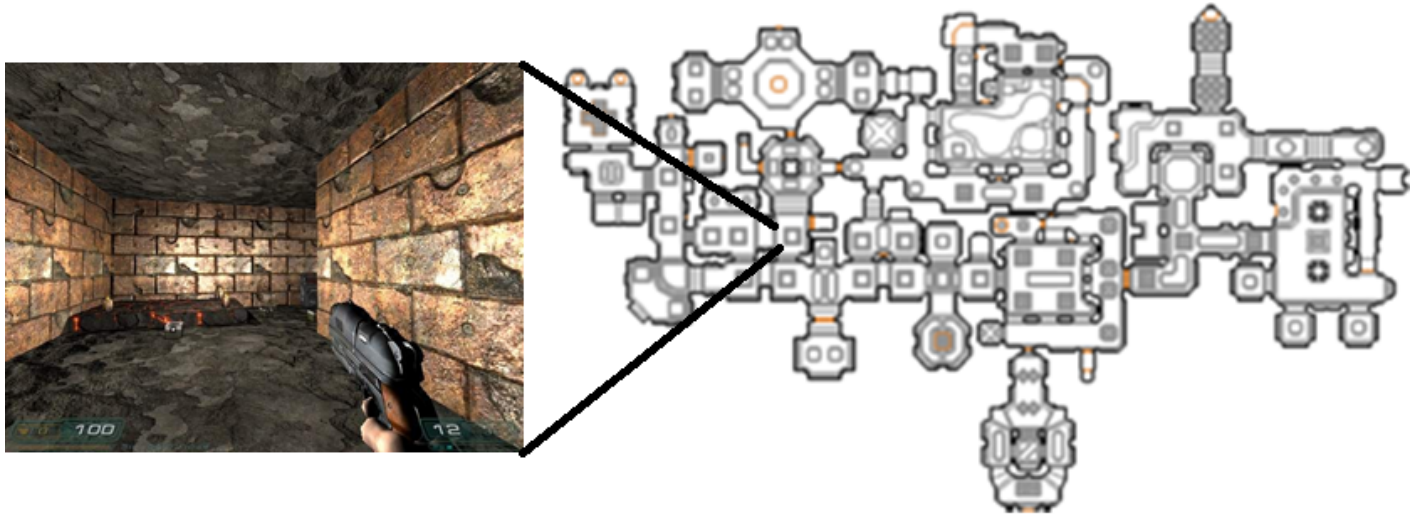
- We don't want to waste computational resources rendering primitives which don't contribute to the final image
  - Drawing polygonal faces on screen consumes CPU cycles
  - e.g. Illumination



Rendering Complex Scenes (2)

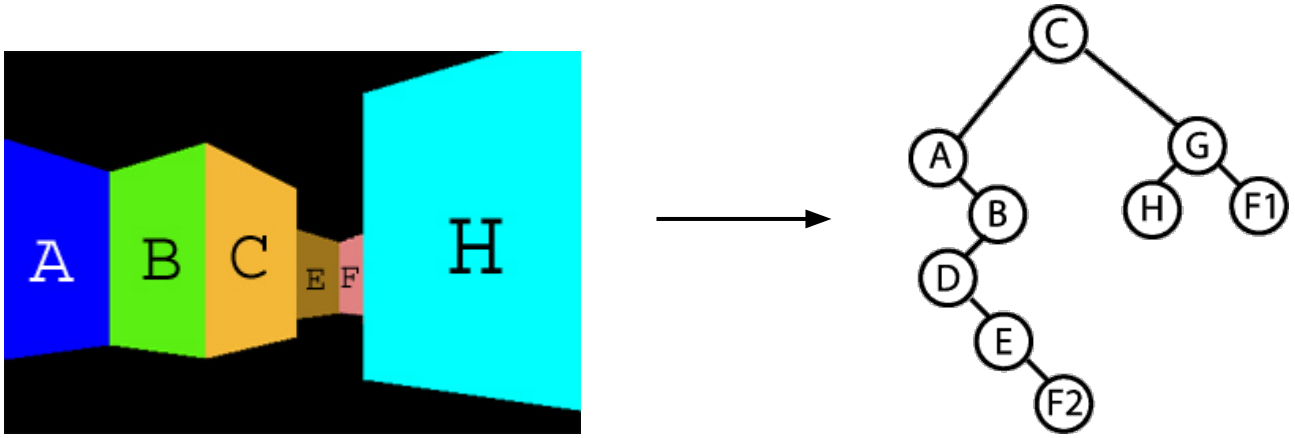
- We better sort the polygons according to the depth
  - coming back to the painter's algorithm
- And only draw those close to the viewer

⟶ BSP Tree, Portal culling



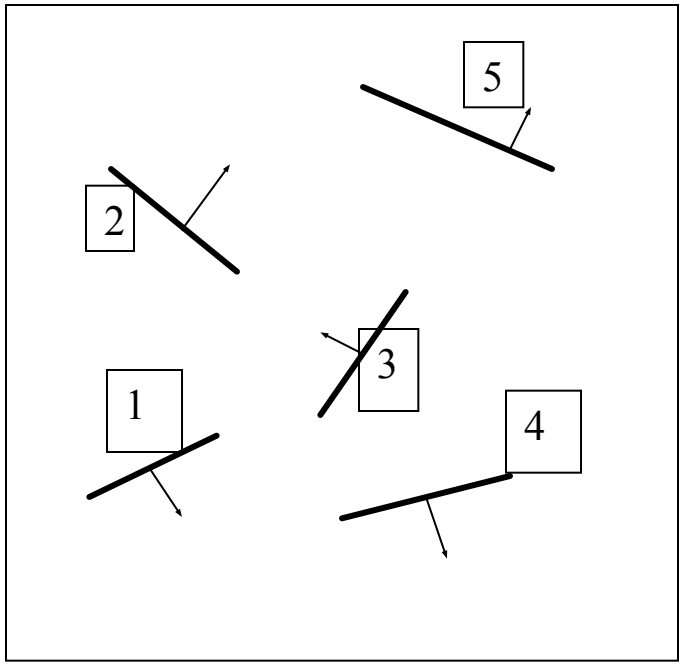
BSP Tree

- A tree structure that represents the scene
- Constructed by a detailed scene analysis
- Scene is then drawn by traversing the tree
- Suitable for static scenes



BSP (Binary Space Partitioning) Tree.

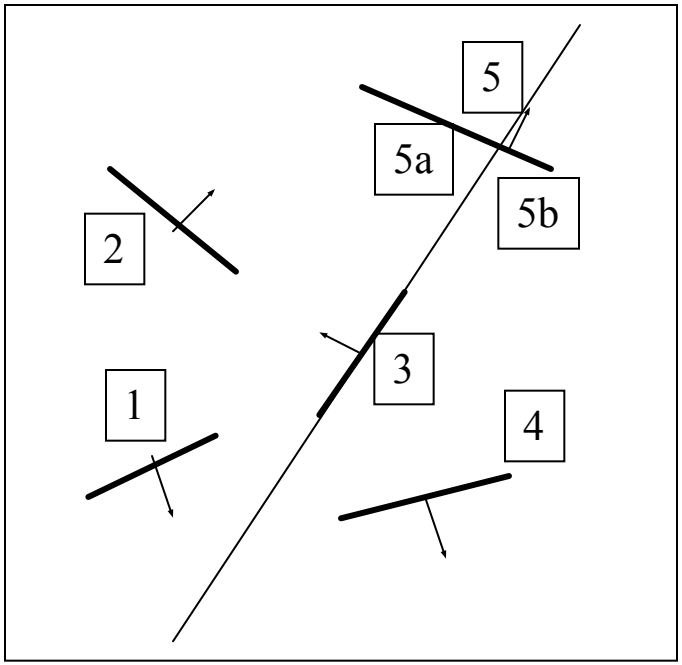
1. Choose polygon arbitrarily
2. Divide scene into front (relative to normal) and back half-spaces.
3. Split any polygon lying on both sides.
4. Choose a polygon from each side – split scene again.
5. Recursively divide each side until each node contains only 1 polygon.

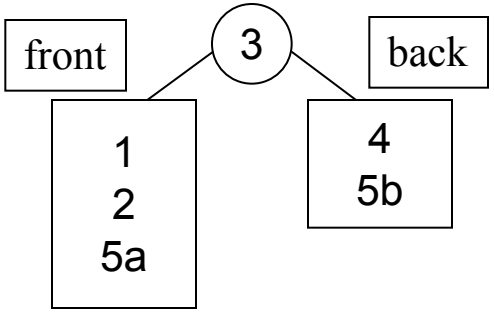


View of scene from above

BSP Tree.

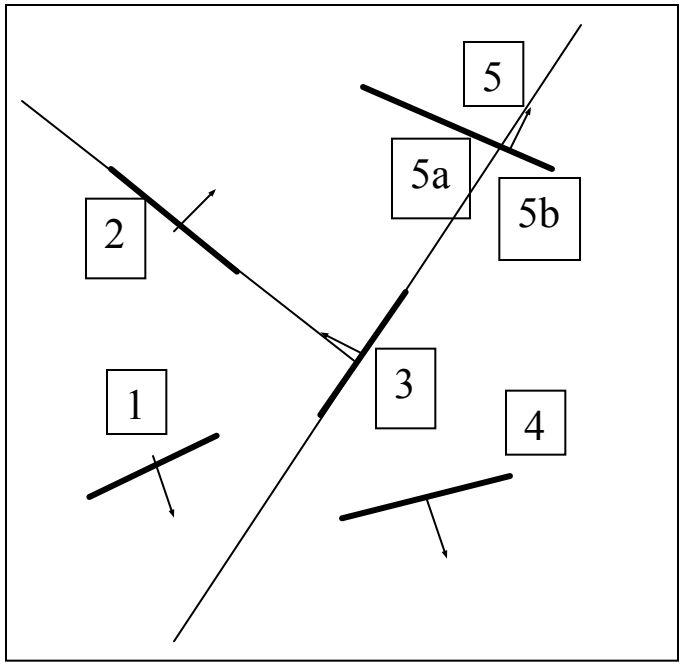
1. **Choose polygon arbitrarily**
2. **Divide scene into front (relative to normal) and back half-spaces.**
3. **Split any polygon lying on both sides.**
4. Choose a polygon from each side – split scene again.
5. Recursively divide each side until each node contains only 1 polygon.

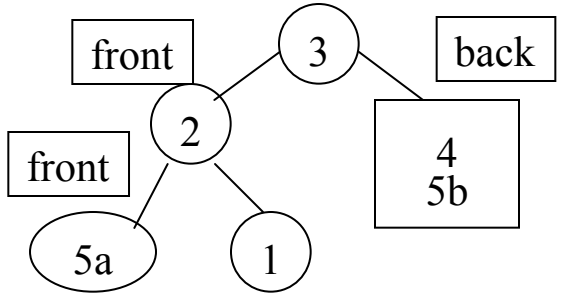




BSP Tree.

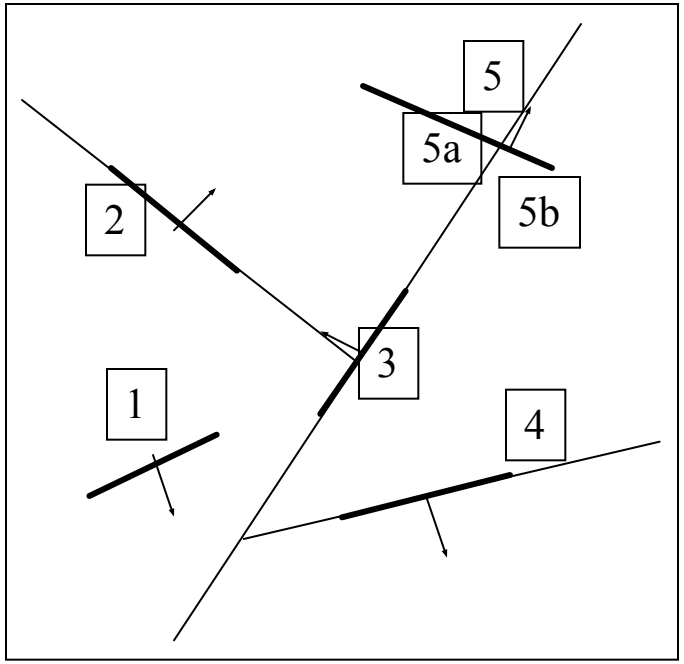
1. Choose polygon arbitrarily
2. Divide scene into front (relative to normal) and back half-spaces.
3. Split any polygon lying on both sides.
4. **Choose a polygon from each side – split scene again.**
5. Recursively divide each side until each node contains only 1 polygon.

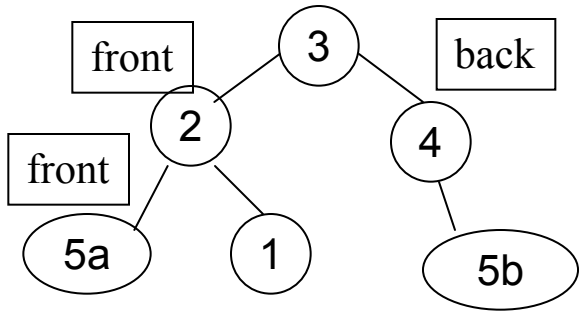




BSP Tree.

1. Choose polygon arbitrarily
2. Divide scene into front (relative to normal) and back half-spaces.
3. Split any polygon lying on both sides.
4. Choose a polygon from each side – split scene again.
5. **Recursively divide each side until each node contains only 1 polygon**.





Displaying a BSP tree.

- BSP tree can be traversed to yield a correct priority list for an arbitrary viewpoint.
  - •Back-to-front : same as painter's algorithm
  - •Front-to-back : a more efficient approach

Displaying a BSP tree : Back to Front

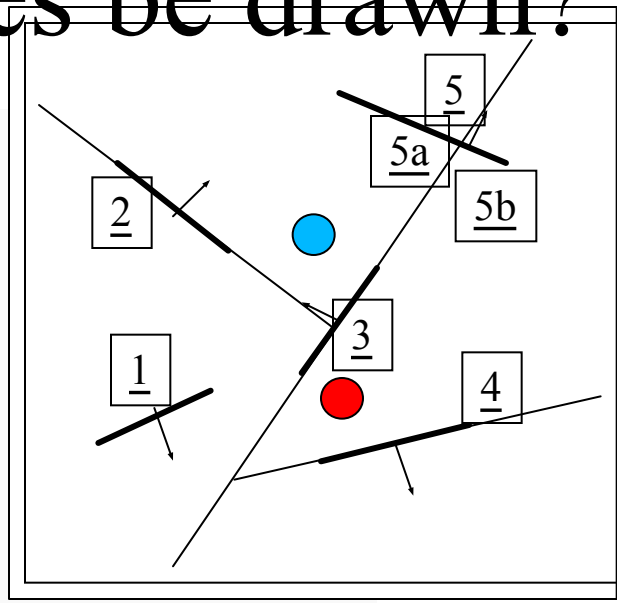
- Start at root polygon.
  - If viewer is in front half-space, draw polygons behind root first, then the root polygon, then polygons in front.
  - If viewer is in back half-space, draw polygons in front of root first, then the root polygon, then polygons behind.
  - If polygon is on edge – either can be used.
  - Recursively descend the tree.
- If eye is in rear half-space for a polygon can back face cull.
- Always drawing the opposite side of the viewer first

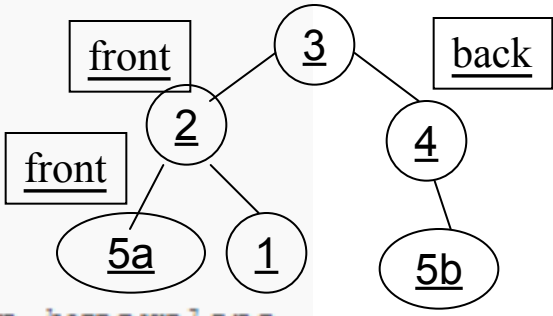
In what order will the faces be drawn?

```
traverse_tree(bsp_tree* tree, point eye)
{
  location = tree->find_location(eye);

  if(tree->empty())
    return;

  if(location > 0)        // if eye in front of location
  {
    traverse_tree(tree->back, eye);
    display(tree->polygon_list);
    traverse_tree(tree->front, eye);
  }
  else if(location < 0) // eye behind location
  {
    traverse_tree(tree->front, eye);
    display(tree->polygon_list);
    traverse_tree(tree->back, eye);
  }
  else                    // eye coincidental with partition hyperplane
  {
    traverse_tree(tree->front, eye);
    traverse_tree(tree->back, eye);
  }
}
```



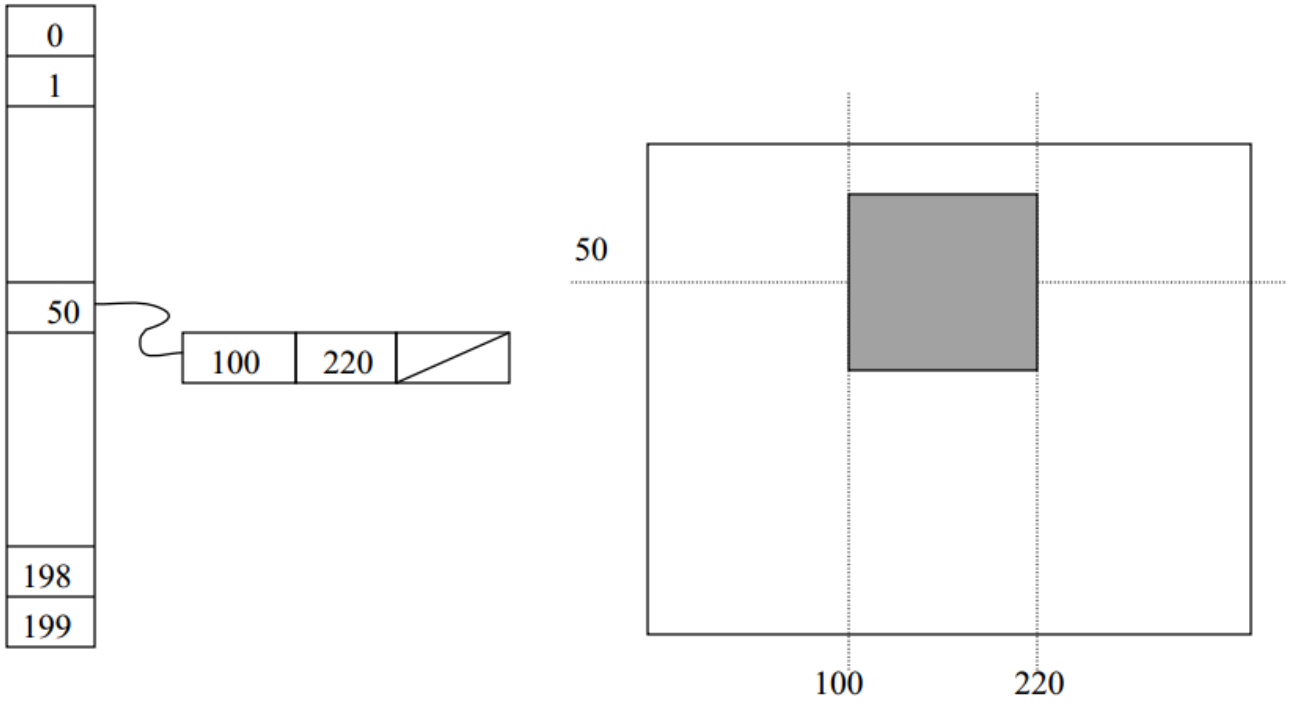


Displaying a BSP tree : Front to Back

- The back-to-front rendering will result in a lot of over drawing again
- Front-to-back traversal is more efficient (Chen and Gordon, 1991)
  - Record which region has been filled in already
  - Terminate when all regions of the screen is filled in

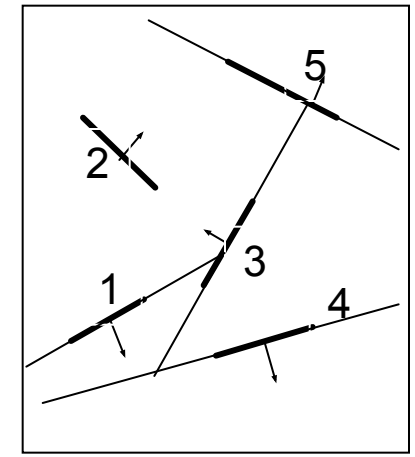
Displaying a BSP tree : Front to Back (2)

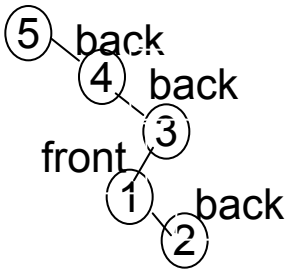
- To hold data on filled in pixels, use Active Edge Table
- Recording the pixels not filled in yet for each scan line



BSP Tree: Discussions

- A lot of computation required at start.
  - Need to produce a well balanced tree
  - Intersecting polygon splitting may also be costly
- Cheap to check visibility once tree is set up
- Efficient when objects don't change very often in the scene.



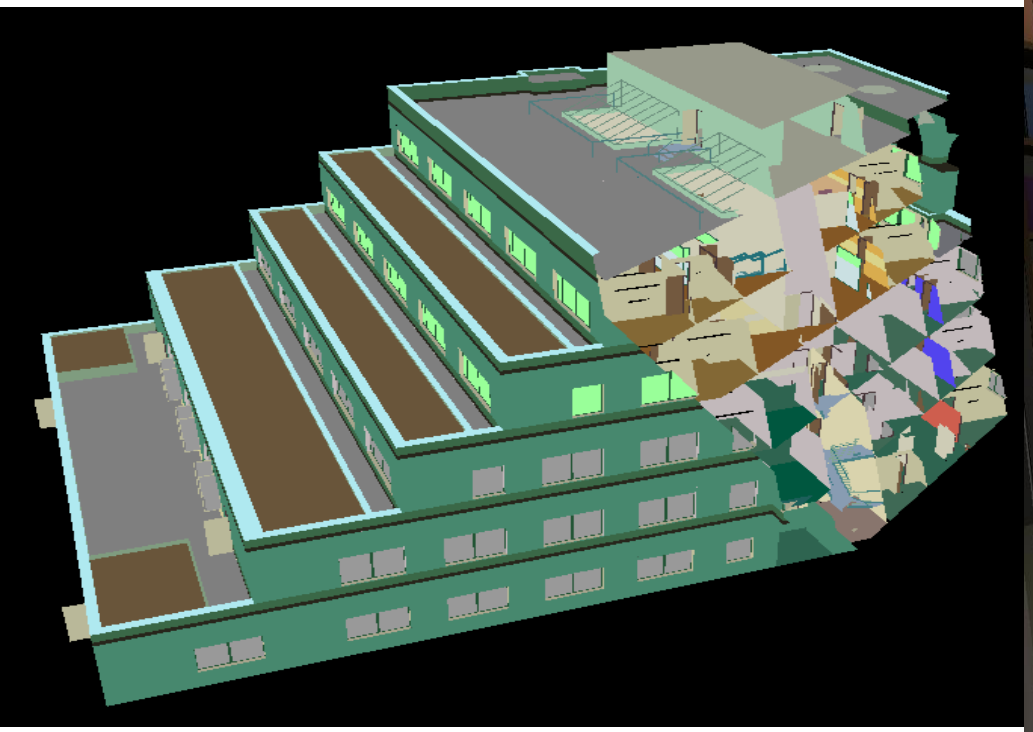


Alternate formulation starting at 5

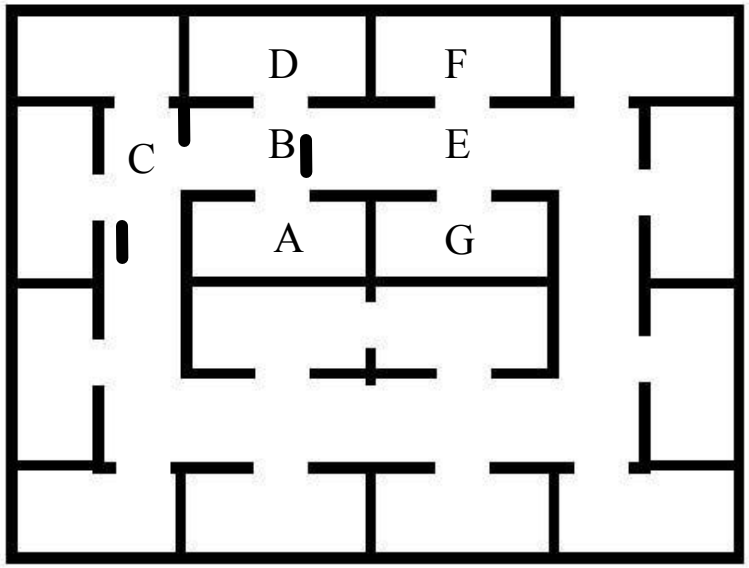
BSP Tree: Discussions (2)

- Good to combine with Z-buffer
- Render the static objects first (front-to-back) with the Z-buffer on,
- And then the dynamic objects (doors, characters)

Ex. Architectural scenes



Portal Culling



Model scene as a graph:

- Nodes: Cells (or rooms)
- Edges: Portals (or doors)

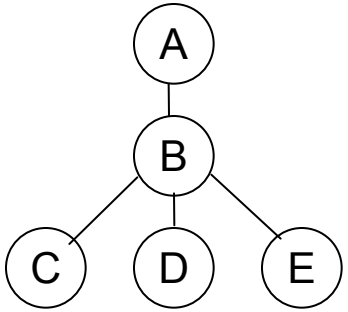
Graph gives us:

- Potentially visible set

1.Render the room

2.If portal to the next room is visible, render the connected room in the portal region

3.Repeat the process along the scene graph



Object space and Image space classification:

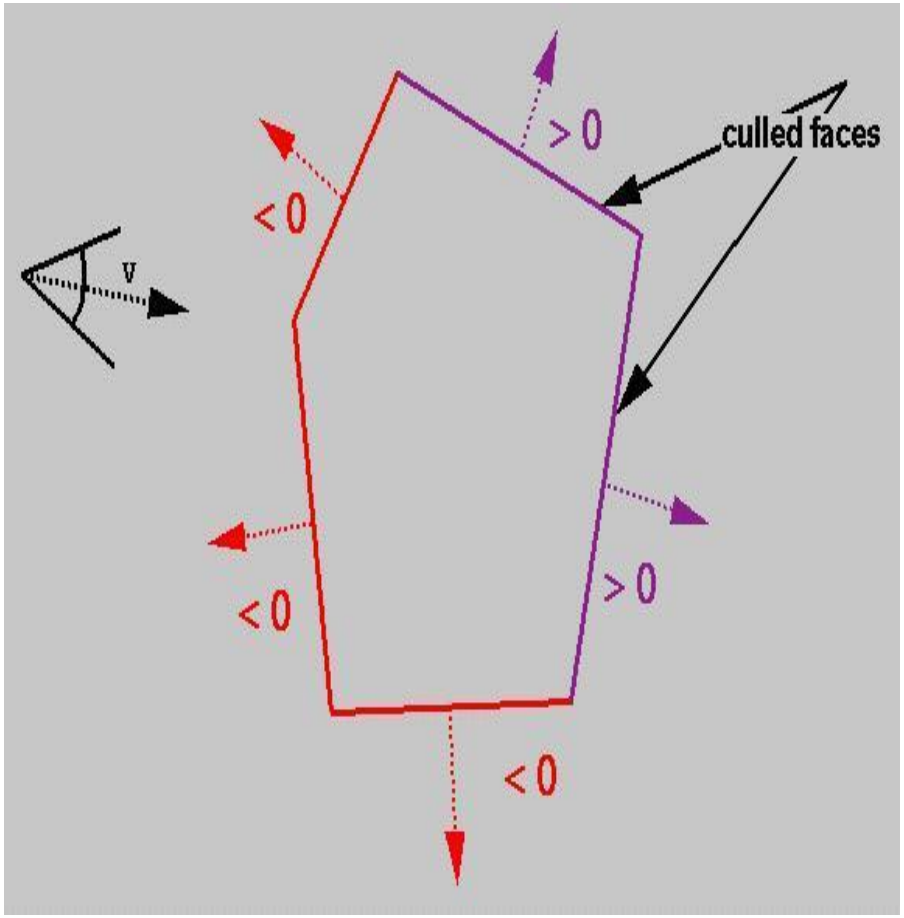
- Object space techniques: applied before vertices are mapped to pixels
  - Painter's algorithm, BSP trees, portal culling
- Image space techniques: applied while the vertices are rasterized
  - Z-buffering

Back Face Culling.

•We do not draw polygons facing the other direction

•Test z component of surface normals. If negative – cull, since normal points away from viewer.

•Or if N.V > 0 we are viewing the back face so polygon is obscured.



Summary for Hidden Surface Removal

- Z-buffer is easy to implement on hardware and is a standard technique for hidden surface removal
- We need to combine it with an object-based method especially when there are too many polygons BSP trees, portal culling
- Need to do the front-to-back traversal to reduce the cost

Overview

Hidden Surface Removal

- Painter's algorithm
- Z-buffer
- BSP tree
- Portal culling
- Back face culling

- **Transparency**
  - Alpha blending
  - Screen door transparency

Transparency

Sometimes we want to render transparent objects

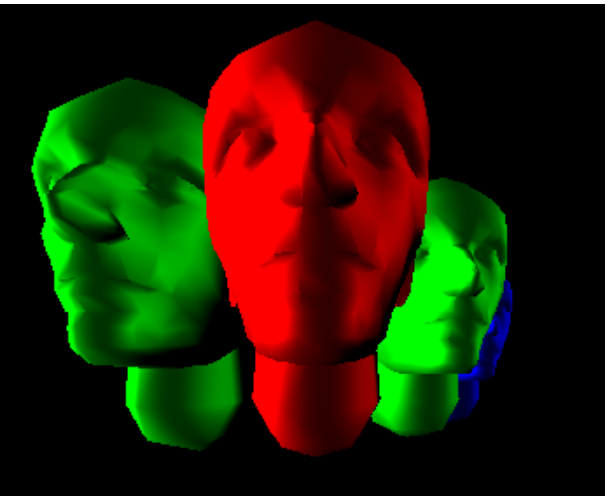
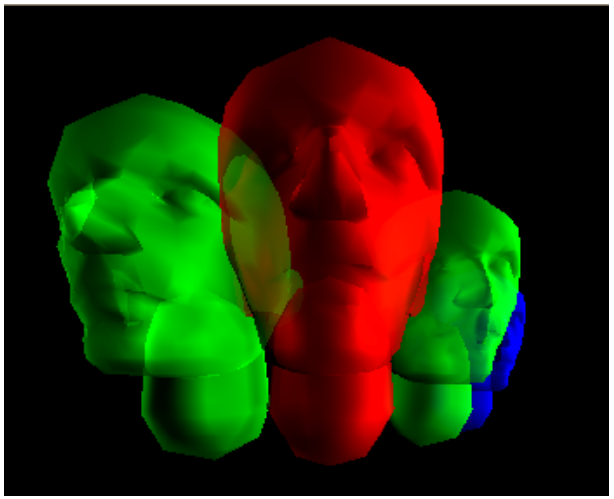
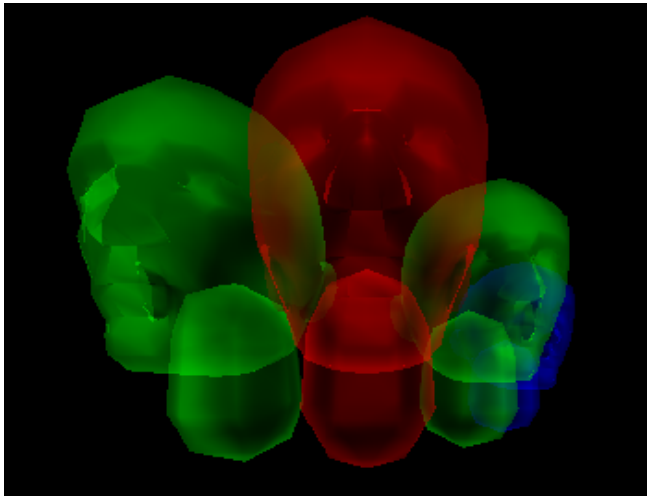
We blend the colour of the objects along the same ray

- Alpha blending
- Screen door transparency

Alpha Blending

Another variable called alpha is defined here
This describes the opacity
Alpha = 1.0 means fully opaque
Alpha = 0.0 means fully transparent

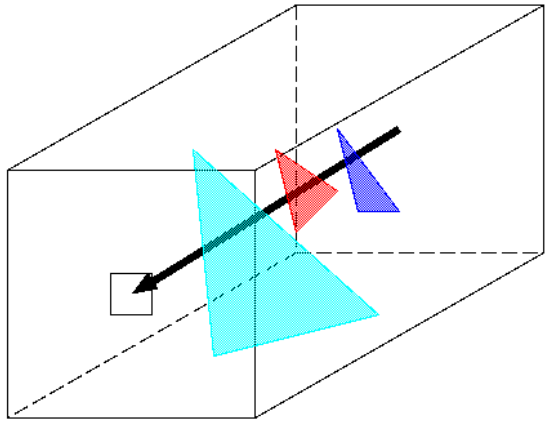
$\alpha = 1$ $\alpha = 0.5$ $\alpha = 0.2$

Sorting by the depth

First, you need to save the depth and colour of all the fragments that will be projected onto the same pixel in a list

Then blend the colour from back towards the front



Colour Blending

The colours of overlapping fragments are blended as follows:

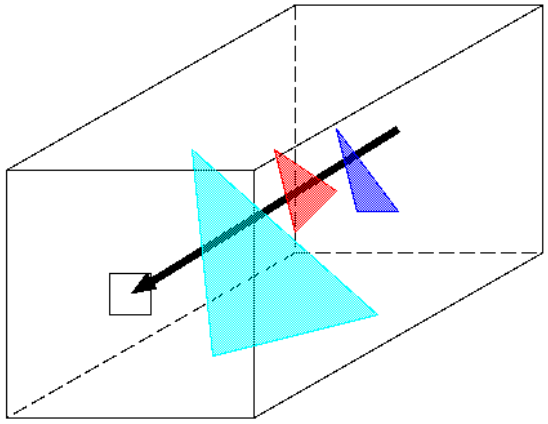
$C_o = \alpha C_s + (1-\alpha) C_d$

C_s : colour of the transparent object,

C_d is the pixel colour before blending,

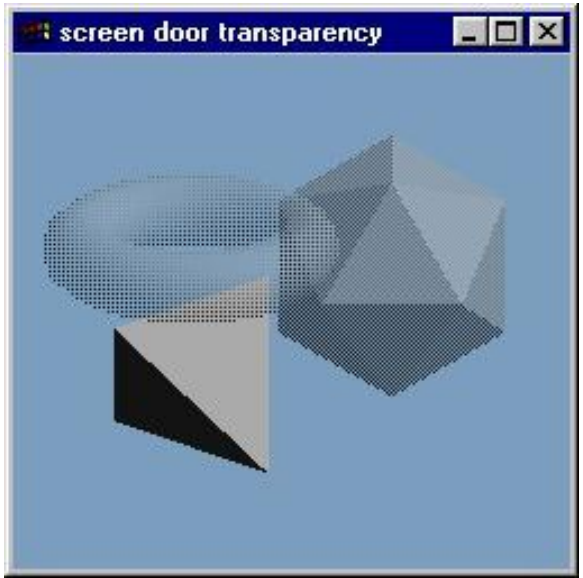
C_o is the new colour as a result of blending

C_o becomes C_d for the next round



Sorting is expensive

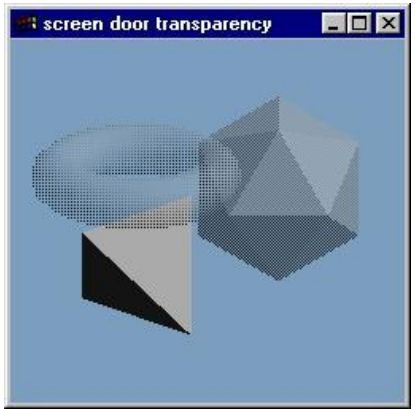
- Need to use BSP Tree
  - Soring per-pixel is very expensive
- Any faster solution ?
  –Screen-door transparency



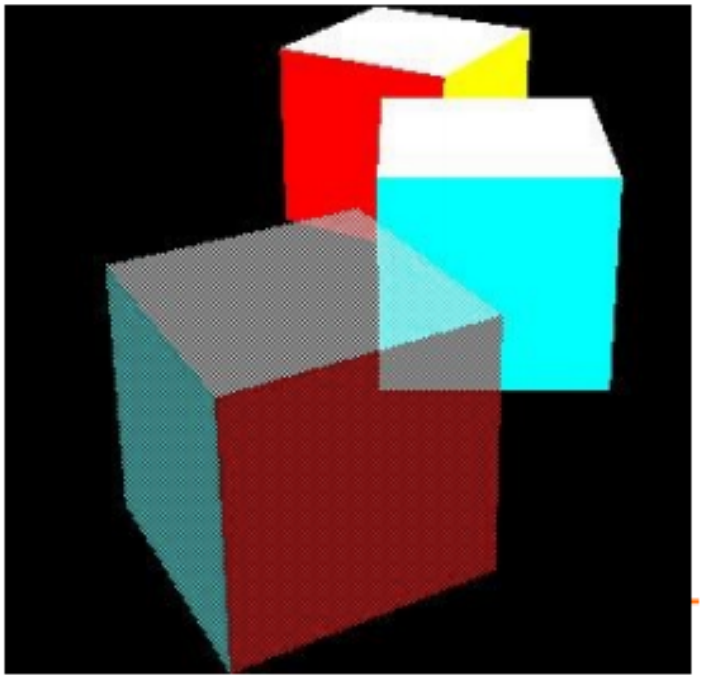
Screen-door Transparency

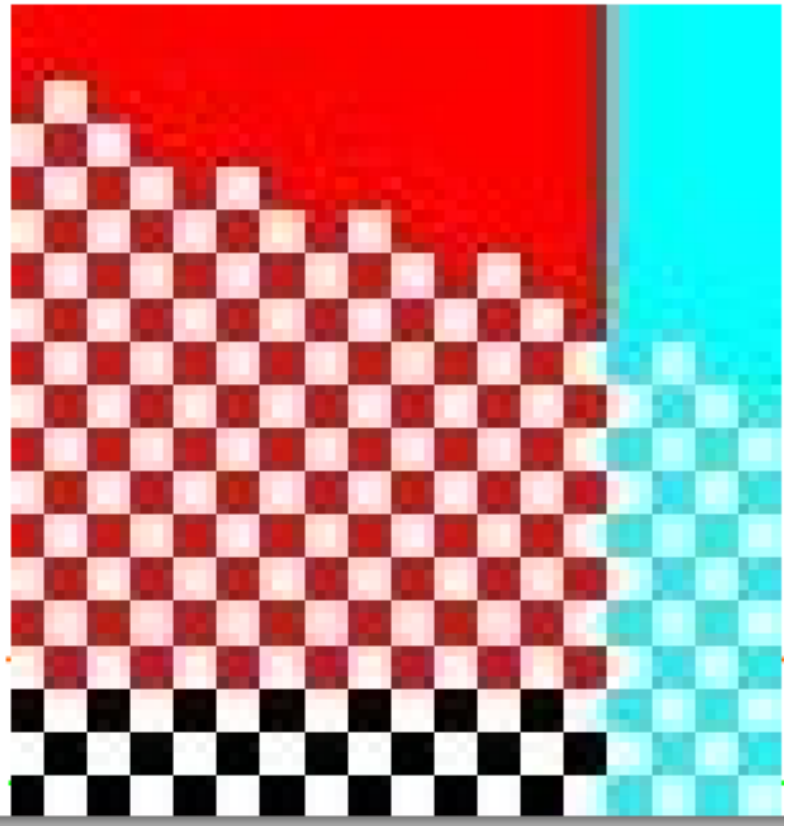
- The object is solid but holes in it
  - like a screen door
- Using a stipple pattern (like a checkboard pattern)

- The ratio of blocked pixels equal to alpha
- No need of sorting : objects can be drawn in any order
- Z-buffer can handle the overlaps of translucent surfaces

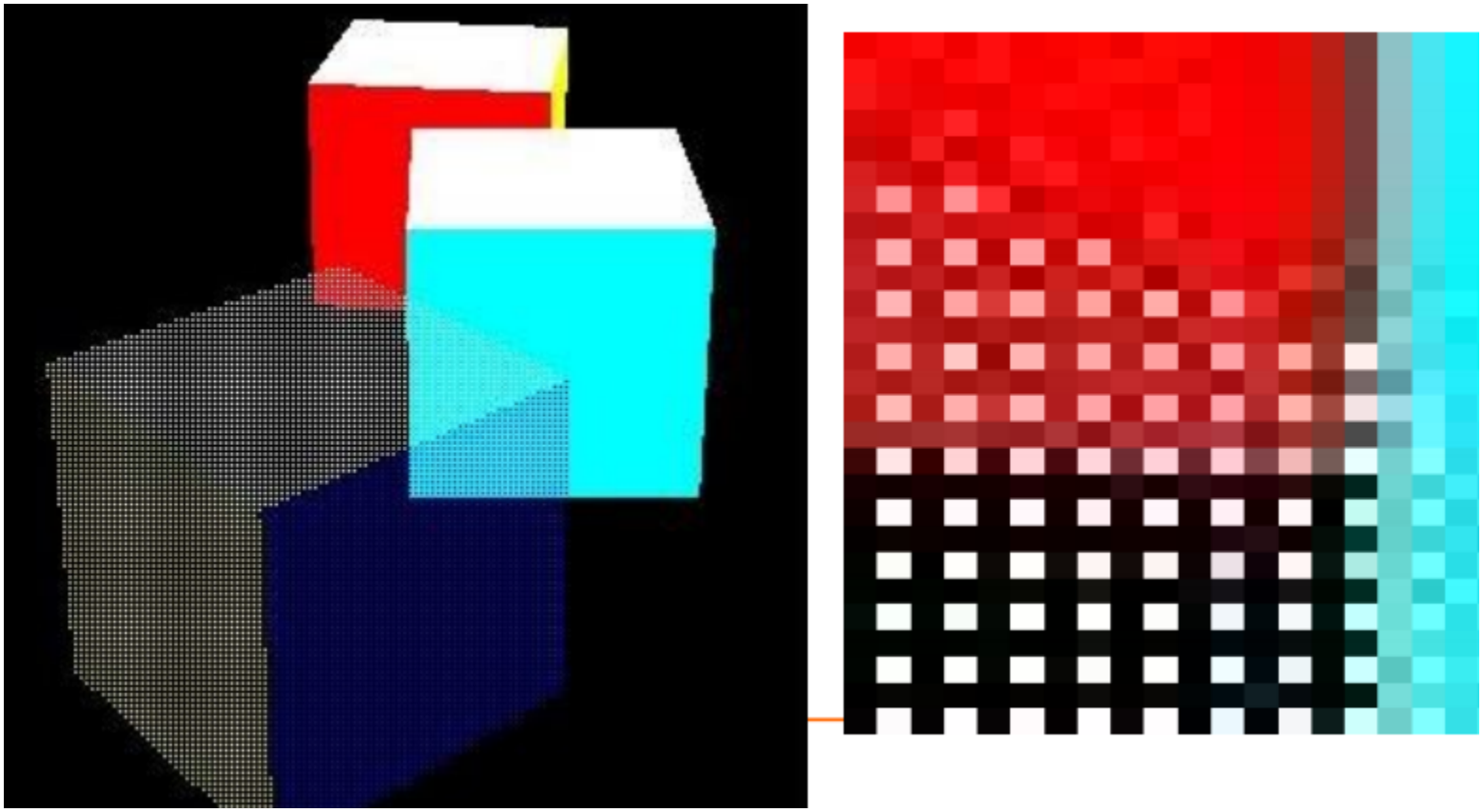


alpha = 0.5



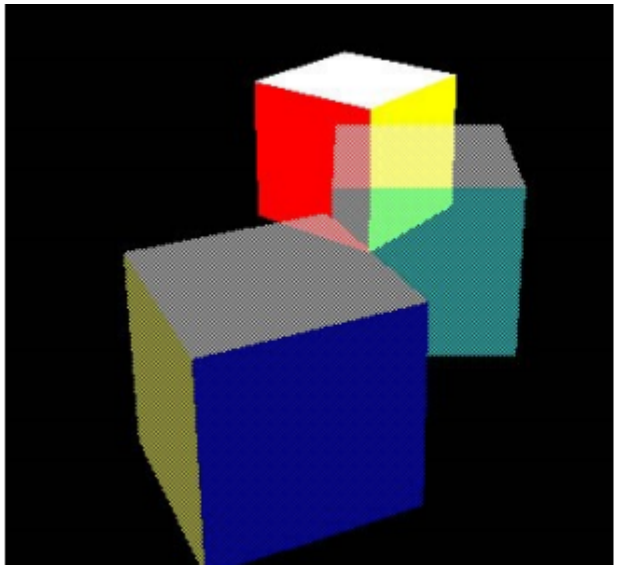


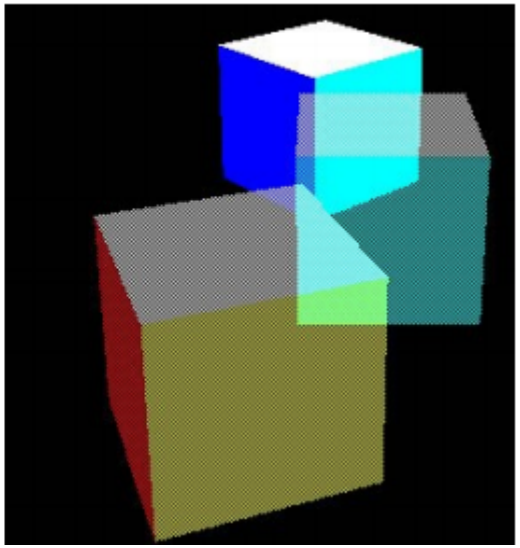
alpha = 0.25



Screen-door Transparency: Issues

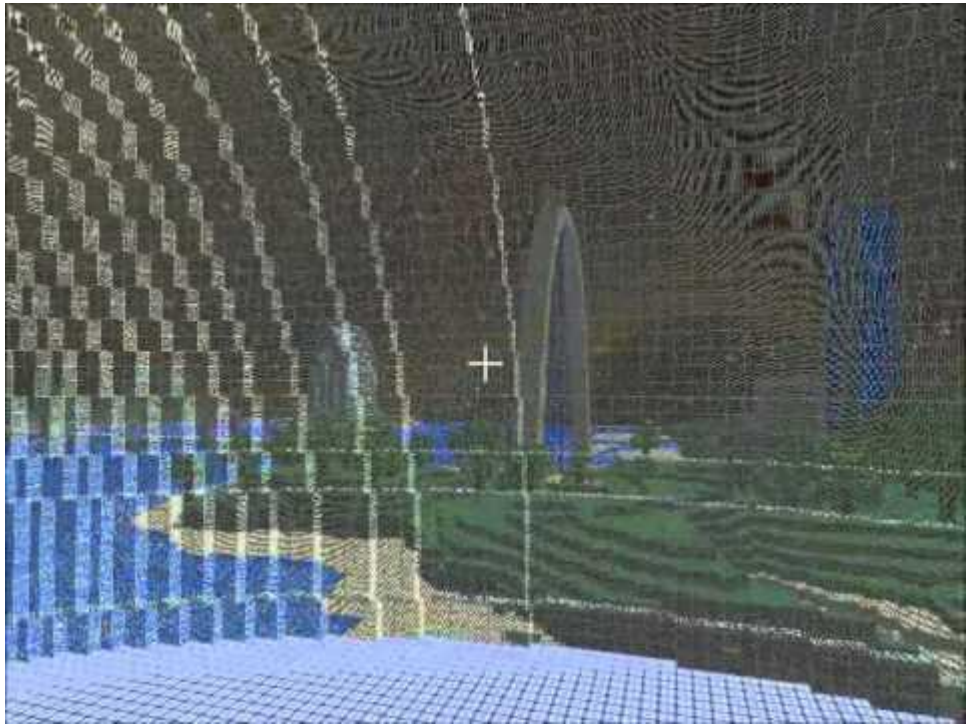
- Transparent object over another transparent object can block everything behind
  - –When fixed patterns are used





Screen-door Transparency: Issues 2

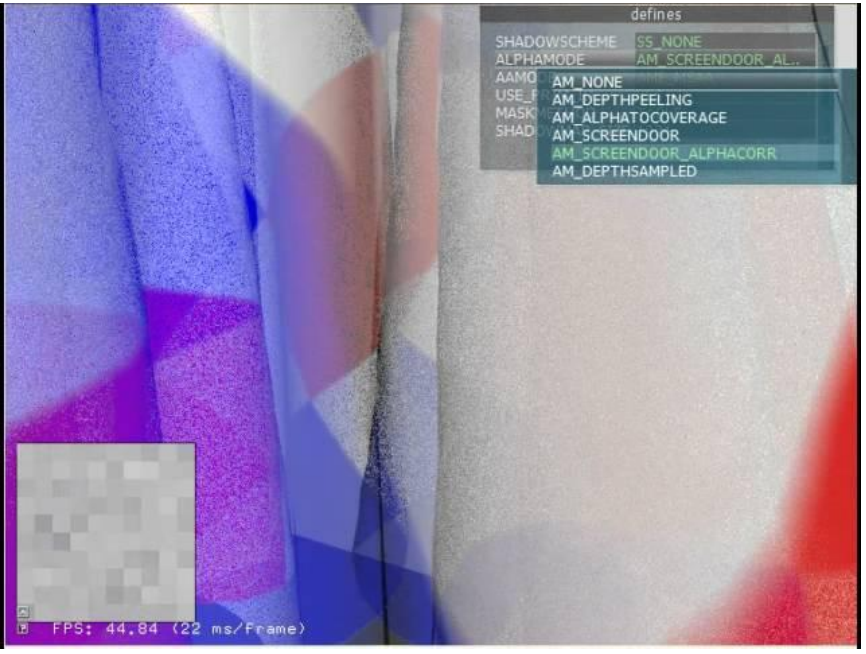
- Stipple patterns must be set in the screen space – otherwise suffer from aliasing
  - –Why?
  - –See this



http://www.youtube.com/watch?v=gMsmJfiApCs

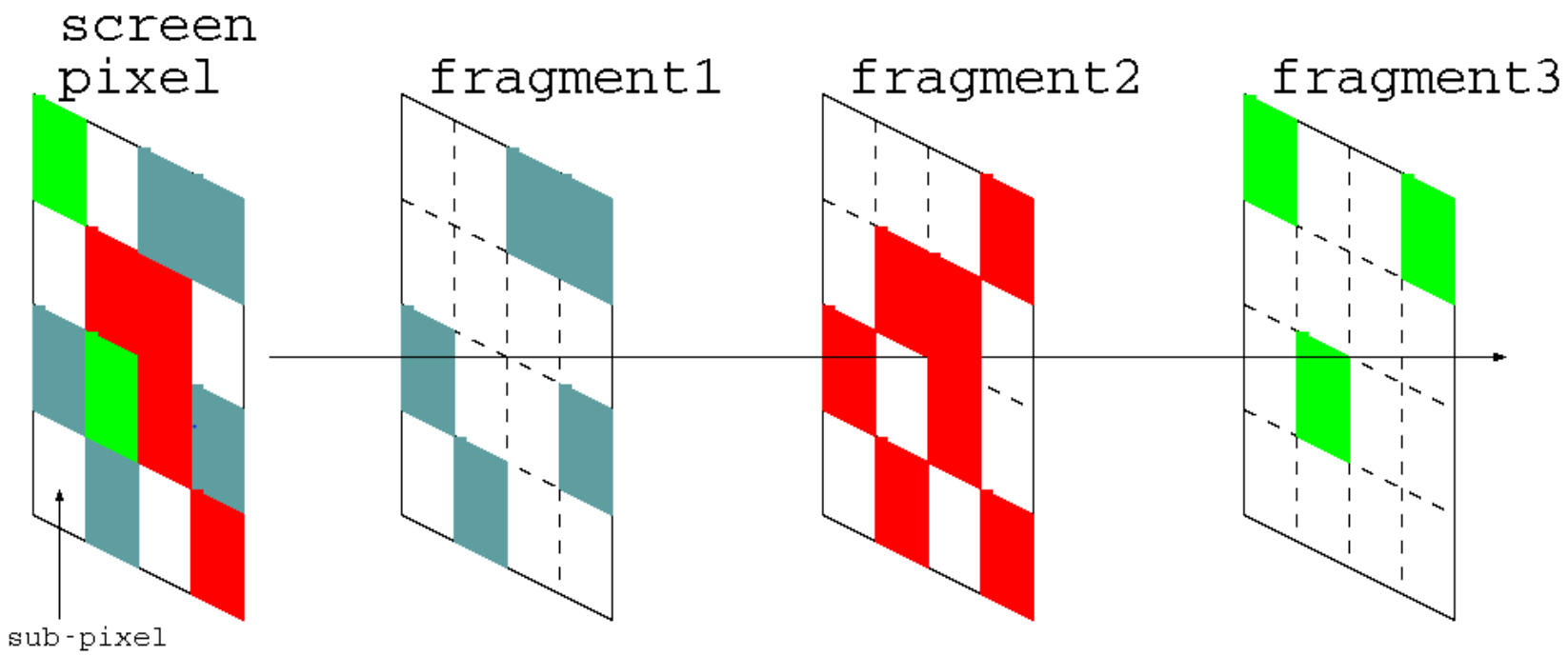
Stochastic Transparency

- Multi-sampling : subpixels are produced and the pixel colour is computed by averaging their colour
- Random sub-pixel stipple pattern



Stochastic Transparency

- No sorting needed
- The final color of the pixel is computed by averaging those of the subpixels



References for hidden surface removal

- Foley et al. Chapter 15, all of it.
- Introductory text, Chapter 13, all of it
- Or equivalents in other texts, look out for:
  - (as well as the topics covered today)
  - Depth sort – Newell, Newell & Sancha
  - Scan-line algorithms
- S. Chen and D. Gordon. "Front-to-Back Display of BSP Trees." IEEE Computer Graphics & Algorithms, pp 79–85. September 1991.

http://research.nvidia.com/publication/stochastic-transparency