Computer Graphics 13 - Hidden surface removal and transparency

Tom Thorne

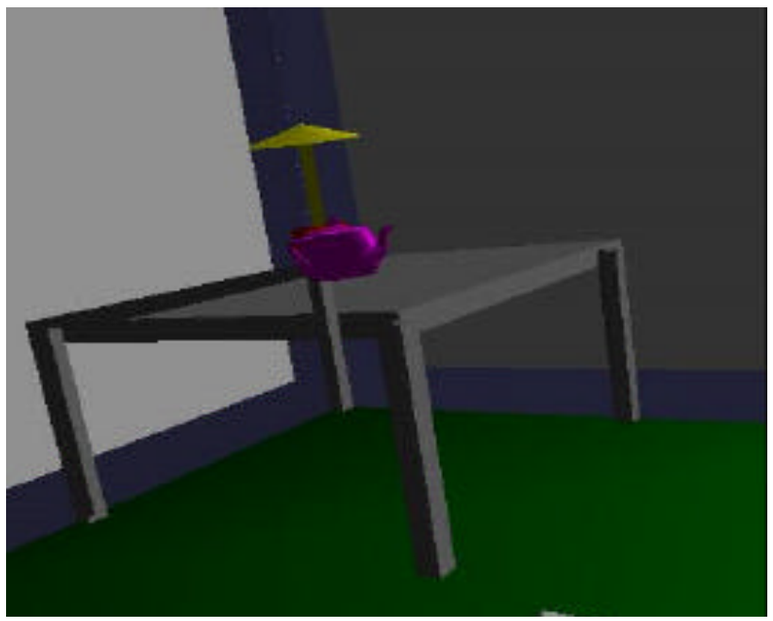
Slides courtesy of Taku Komura
www.inf.ed.ac.uk/teaching/courses/cg

Overview

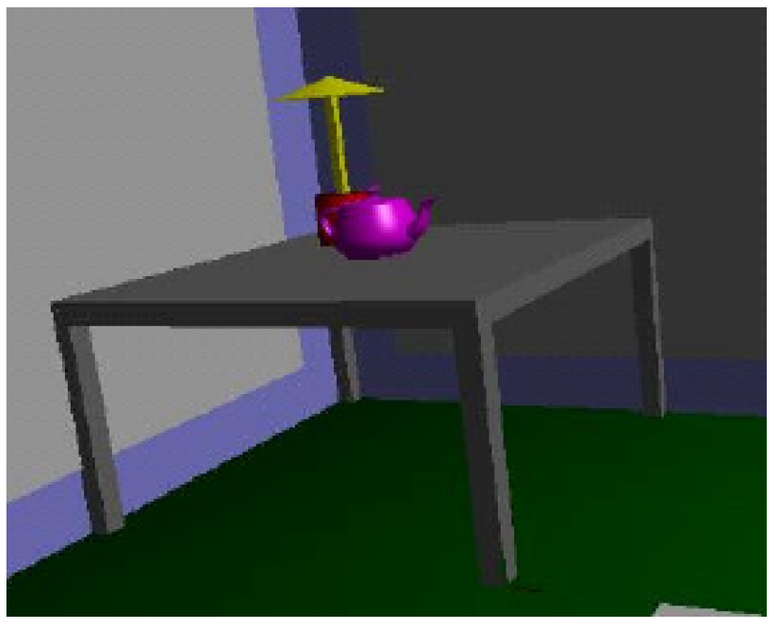
- **Hidden Surface removal**
  - Painter's algorithm
  - Z-buffer
  - BSP tree
  - Portal culling
  - Back face culling
- Transparency
  - Alpha blending
  - Screen door transparency

Why hidden surface removal

- Rendering correctly requires correct visibility calculations
- When multiple opaque polygons cover a space on the screen, only the closest one is visible



Incorrect visibility

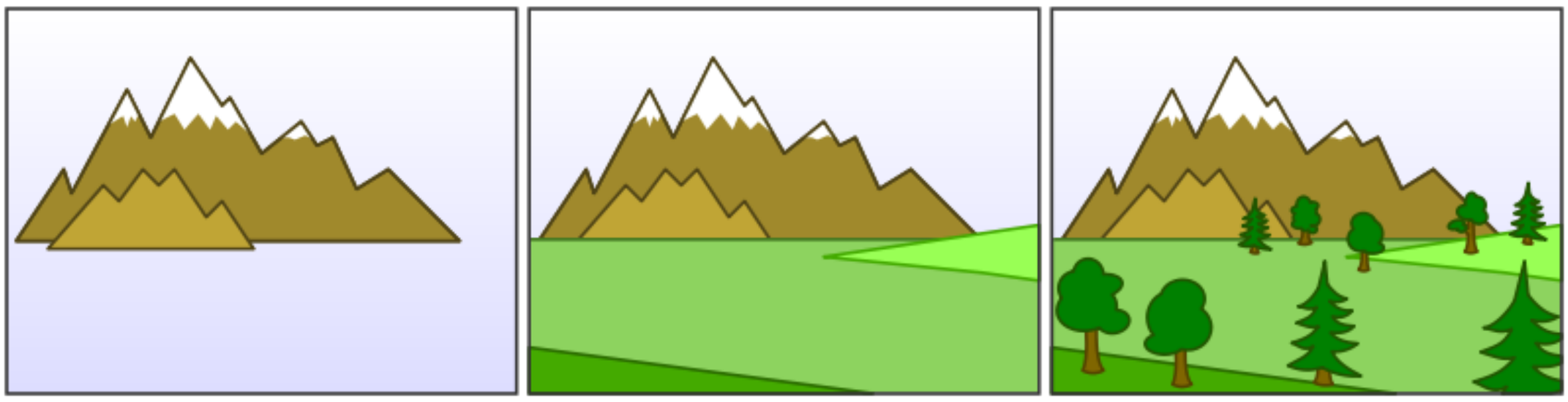


Correct visibility

Painter's algorithm

- Draw surfaces in back to front order, with nearer polygons 'painting' over farther ones

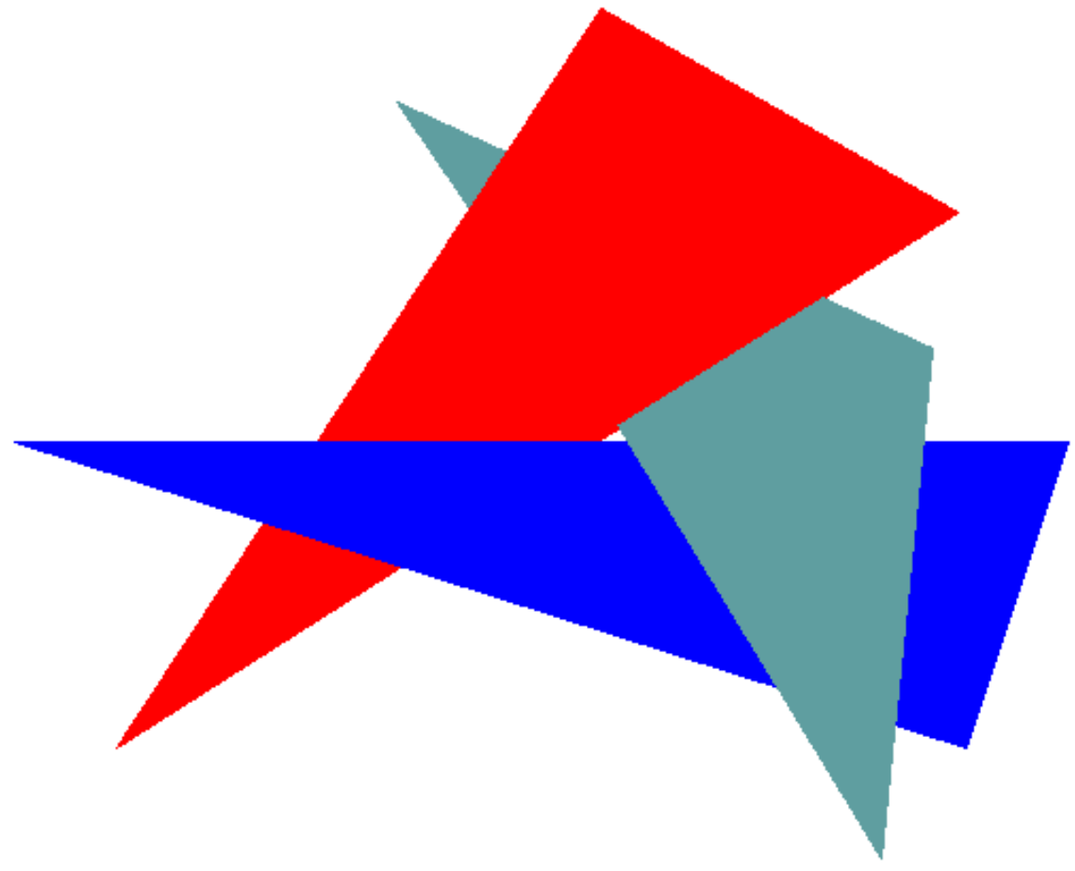
- Need to find the order to draw objects in



Painter's algorithm

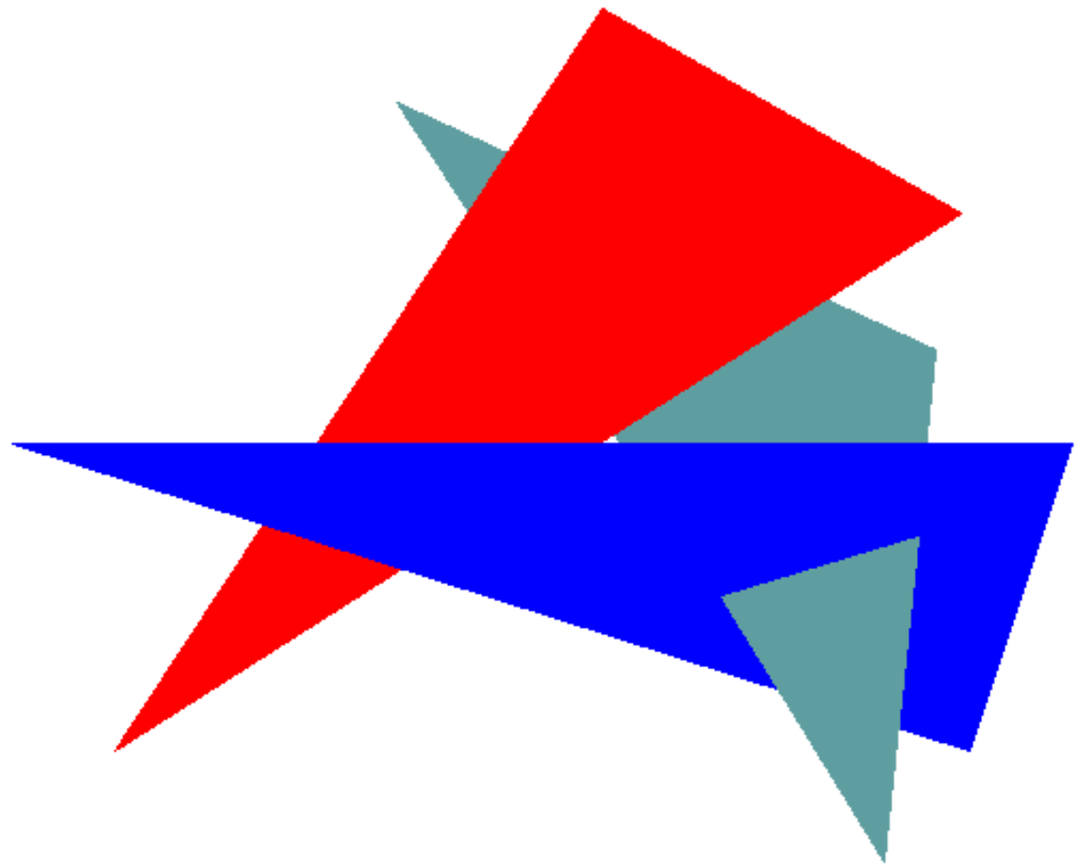
- Main issue is determining the order

- Doesn't always work



Painter's algorithm

- Another problem case
- Need to segment the triangles so that they can be sorted

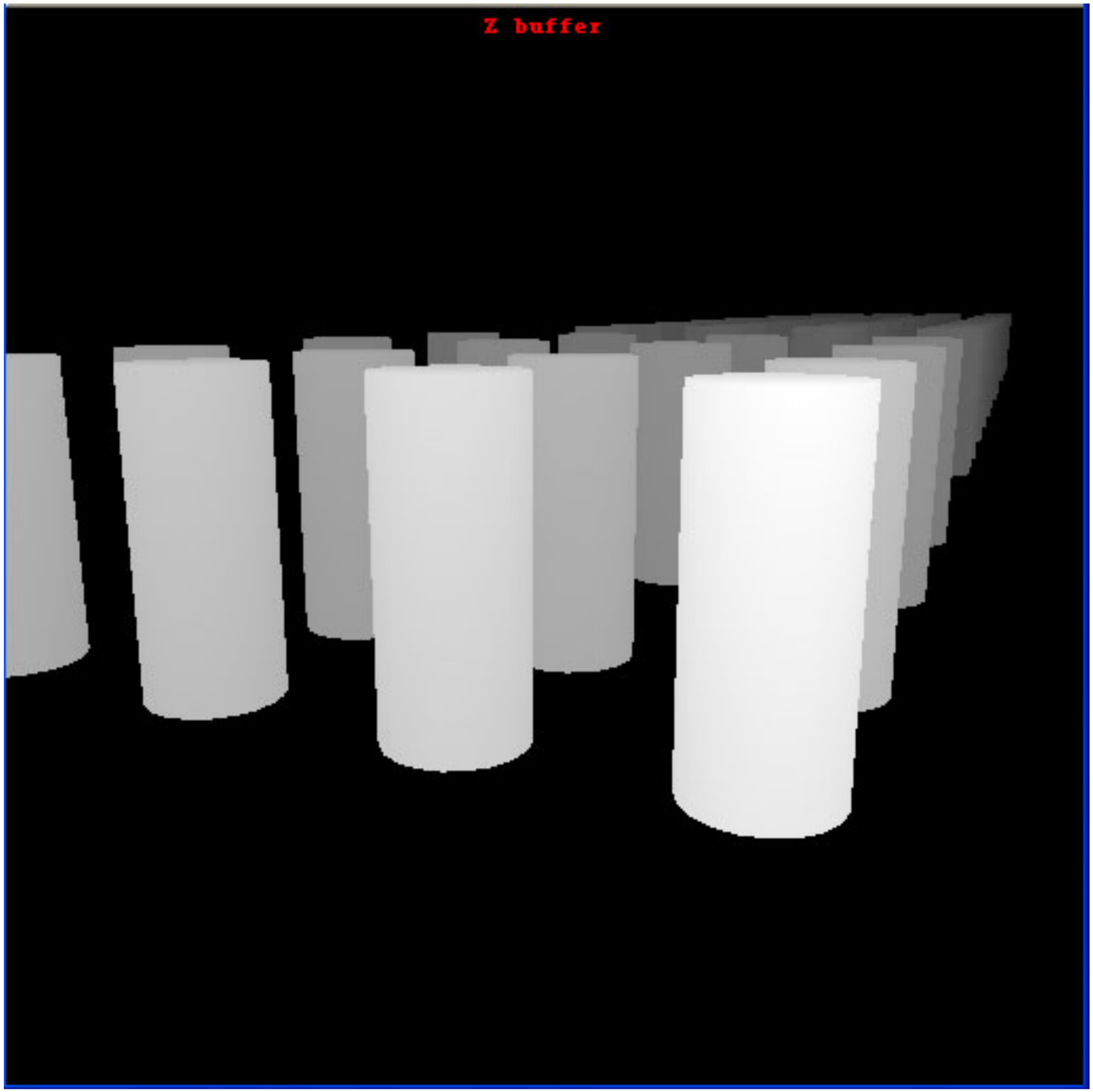


Z-buffer

- An image-based method applied during rasterisation
- Standard approach used in graphics hardware and libraries
- Easy to implement in hardware
- By Wolfgang Straßer in 1974

Z-buffer

- Advantages:
  - Simple to implement in hardware
  - Memory is relatively cheap
  - Works with any primitives
  - Unlimited complexity
  - No need to sort objects or calculate intersections
- Disadvantages:
  - Wasted time drawing hidden objects
  - Z-precision errors (aliasing)

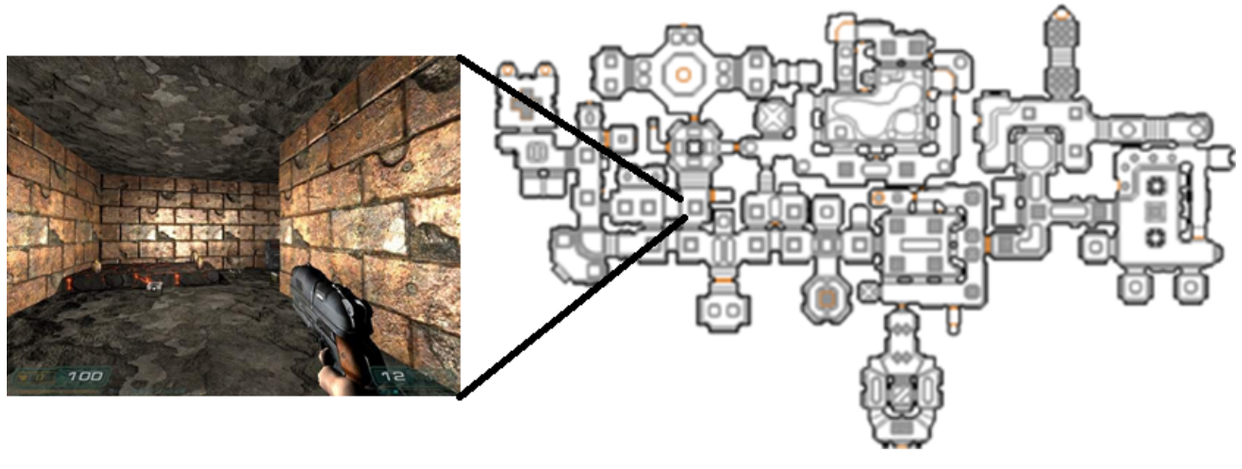

Z buffer

Z-buffer performance

- Memory overhead O(1)
- Visibility O(n) (n = number of polygons)
- Might need to be combined with other culling methods to reduce complexity

Rendering complex scenes

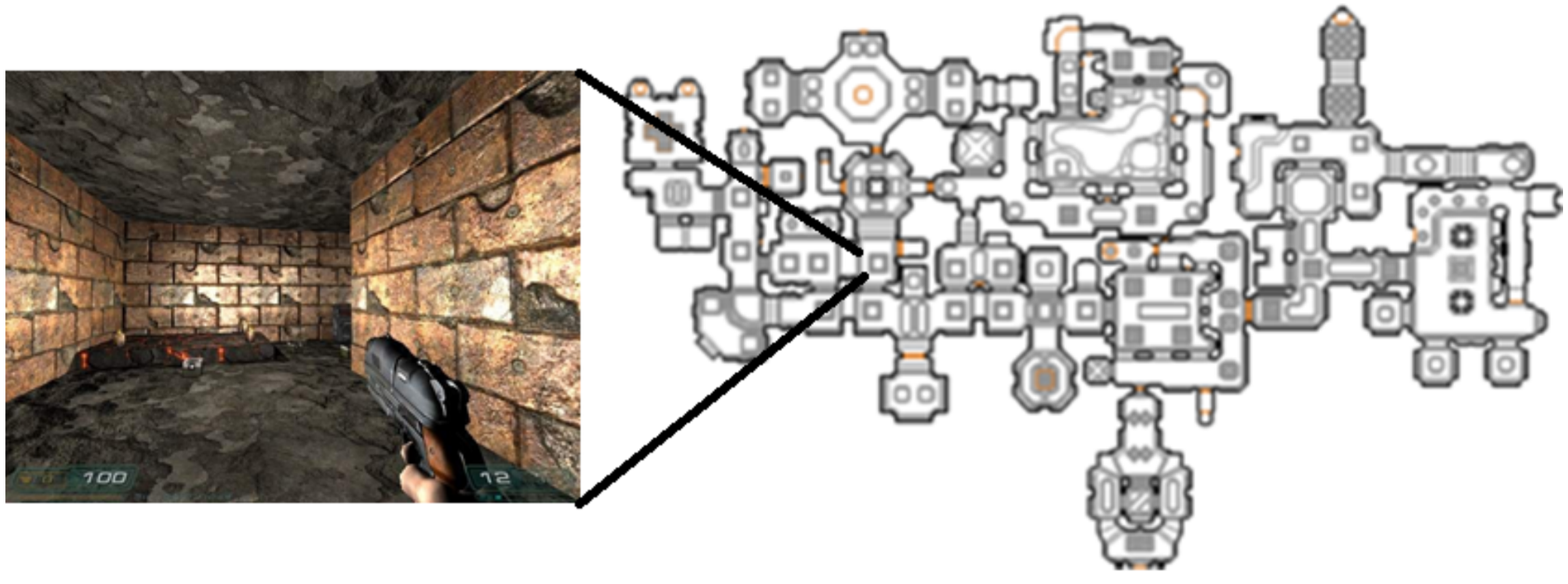
- Don't want to waste resources rendering triangles that don't contribute to the final image
- Drawing each triangle takes CPU/GPU cycles to calculate illumination etc



Rendering complex scenes

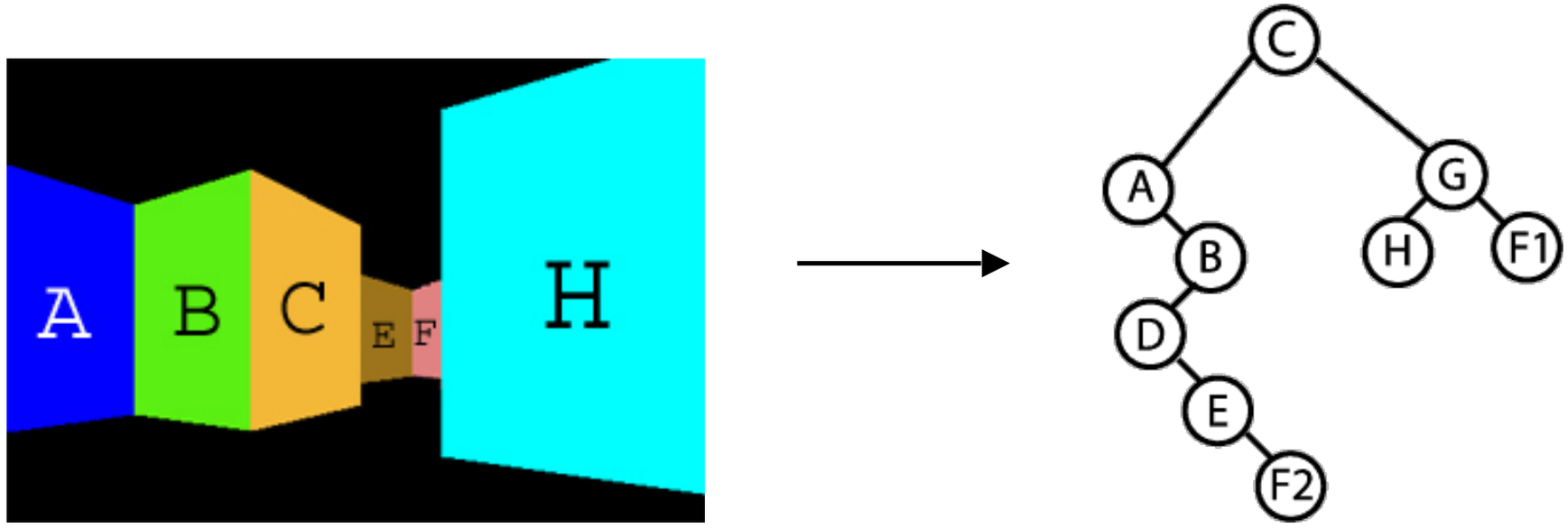
- Sort polygons according to depth and only draw those close to the viewer

- BSP trees, portal culling



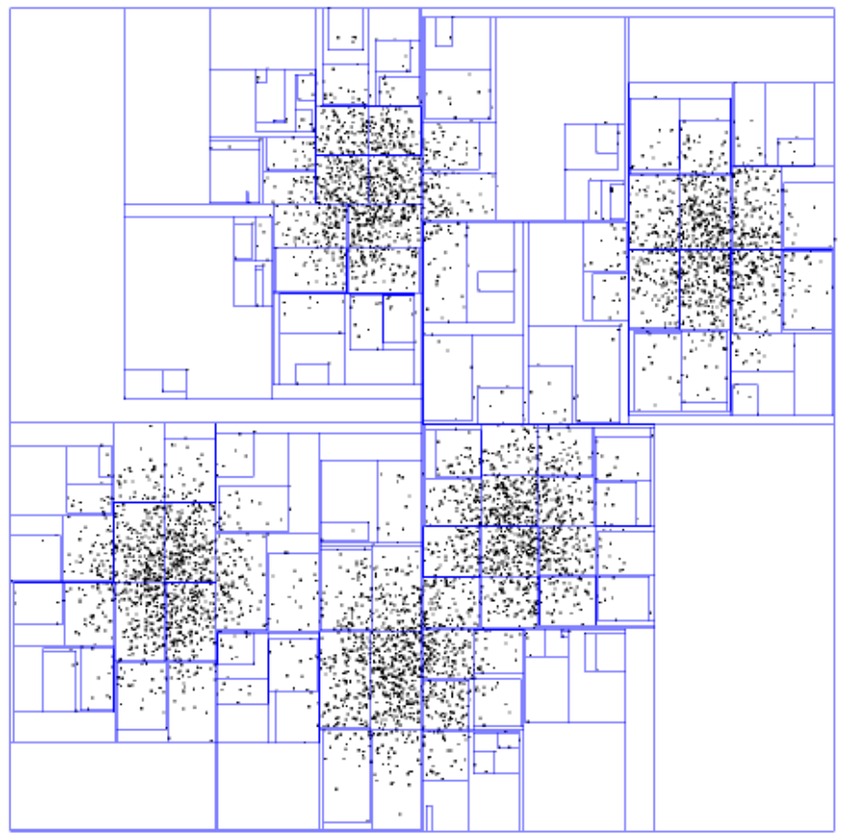
BSP trees

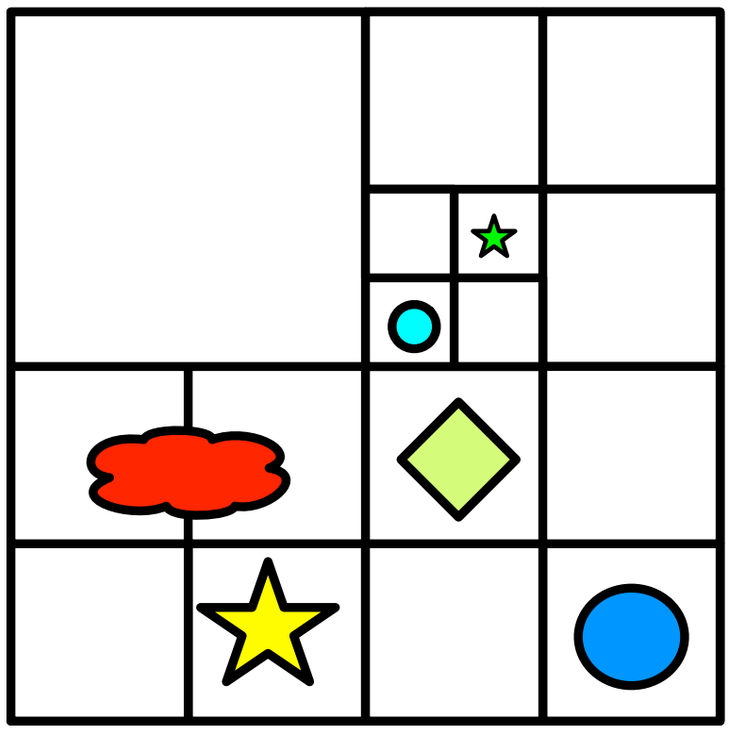
- Binary space partitioning tree
- Represents the scene with a tree
- Scene is drawn by traversing the tree
- Suitable for rendering static scenes



BSP trees

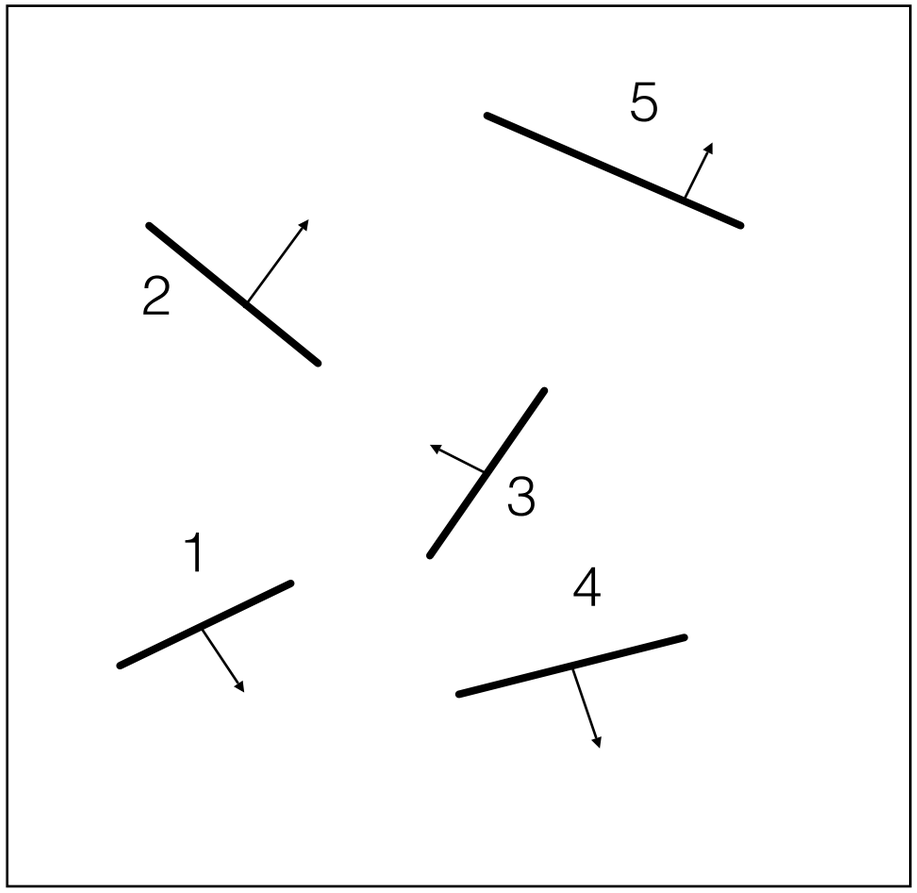
- Splitting schemes:
  - Polygon aligned
  - Axis aligned
- k-d trees
- Quadtrees, octrees





BSP trees

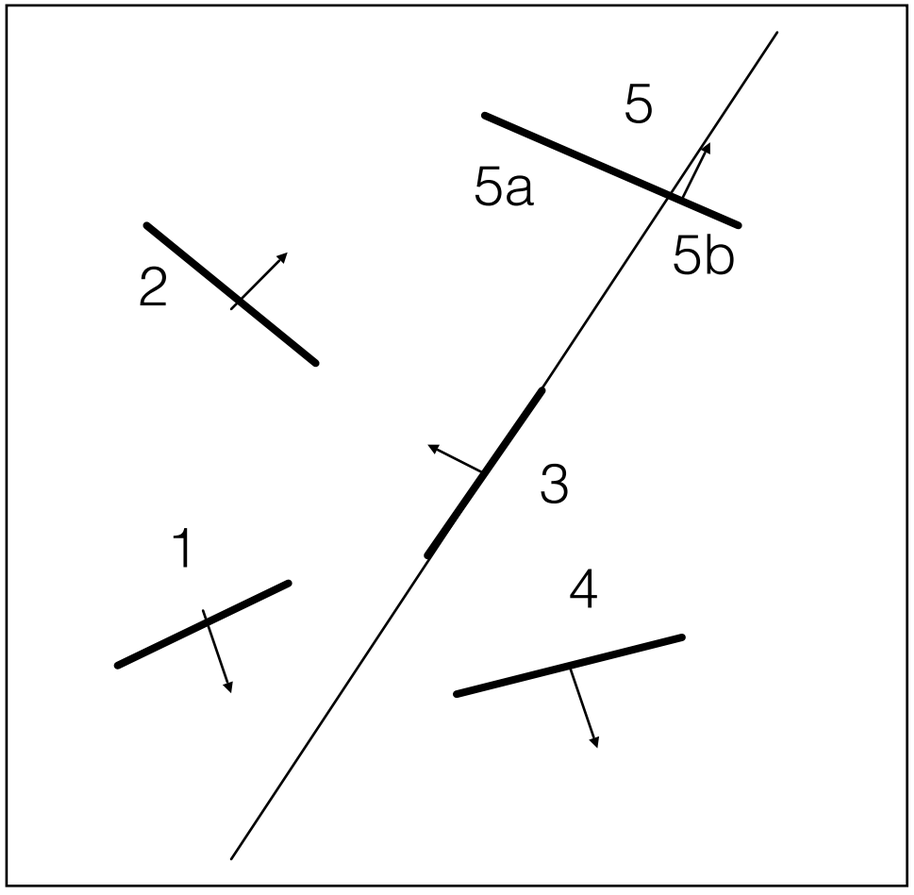
1. Choose polygon arbitrarily
2. Divide scene into front (relative to normal) and back half-spaces.
3. Split any polygon lying on both sides.
4. Choose a polygon from each side – split scene again.
5. Recursively divide each side until each node contains only 1 polygon.

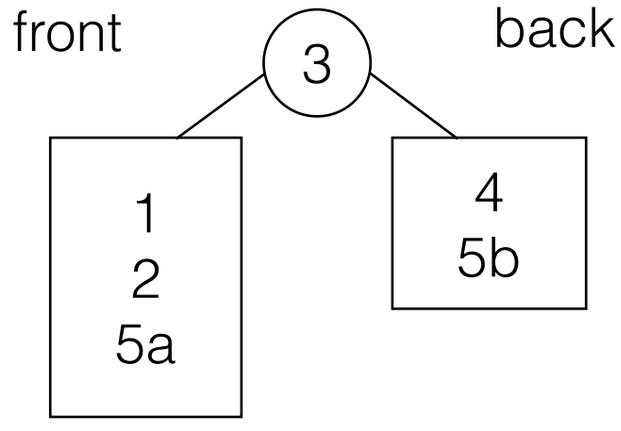


View of scene from above

BSP trees

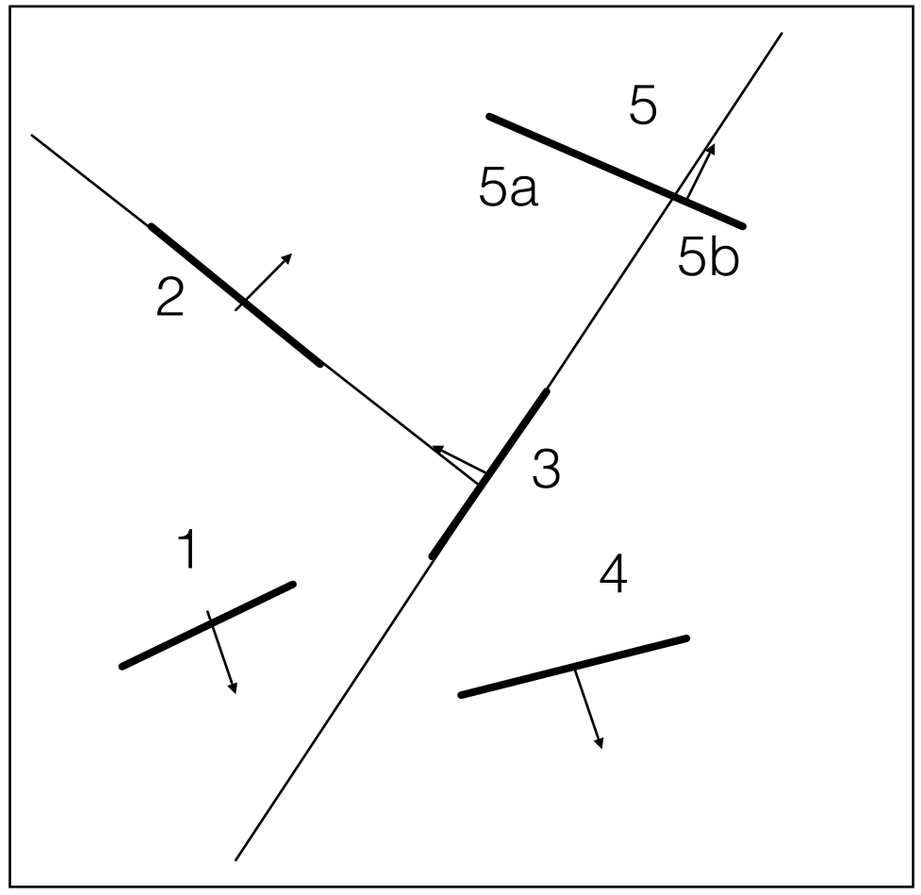
1. **Choose polygon arbitrarily**
2. **Divide scene into front (relative to normal) and back half-spaces.**
3. **Split any polygon lying on both sides.**
4. Choose a polygon from each side – split scene again.
5. Recursively divide each side until each node contains only 1 polygon.

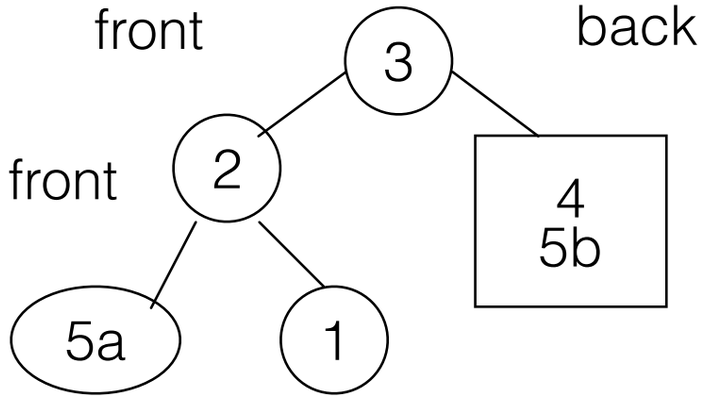




BSP trees

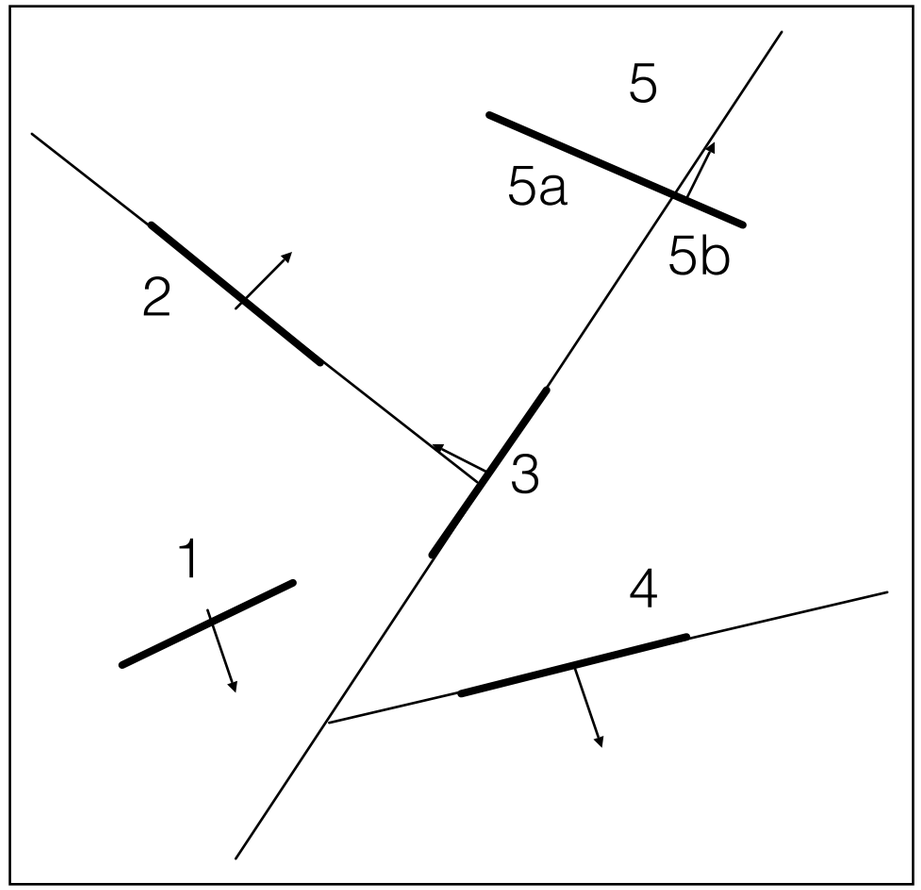
1. Choose polygon arbitrarily
2. Divide scene into front (relative to normal) and back half-spaces.
3. Split any polygon lying on both sides.
4. **Choose a polygon from each side – split scene again.**
5. Recursively divide each side until each node contains only 1 polygon.

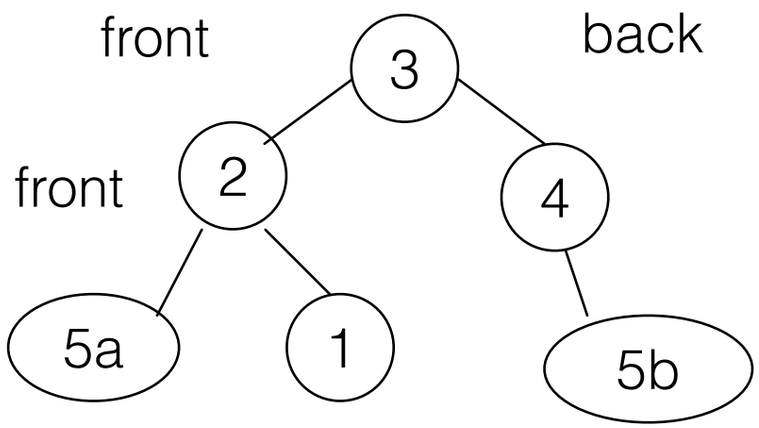




BSP trees

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

- The tree can be traversed to yield an ordering of the polygons for an arbitrary viewpoint
- Back to front using the painter's algorithm
- Front to back - more efficient

Displaying a BSP tree: back to front

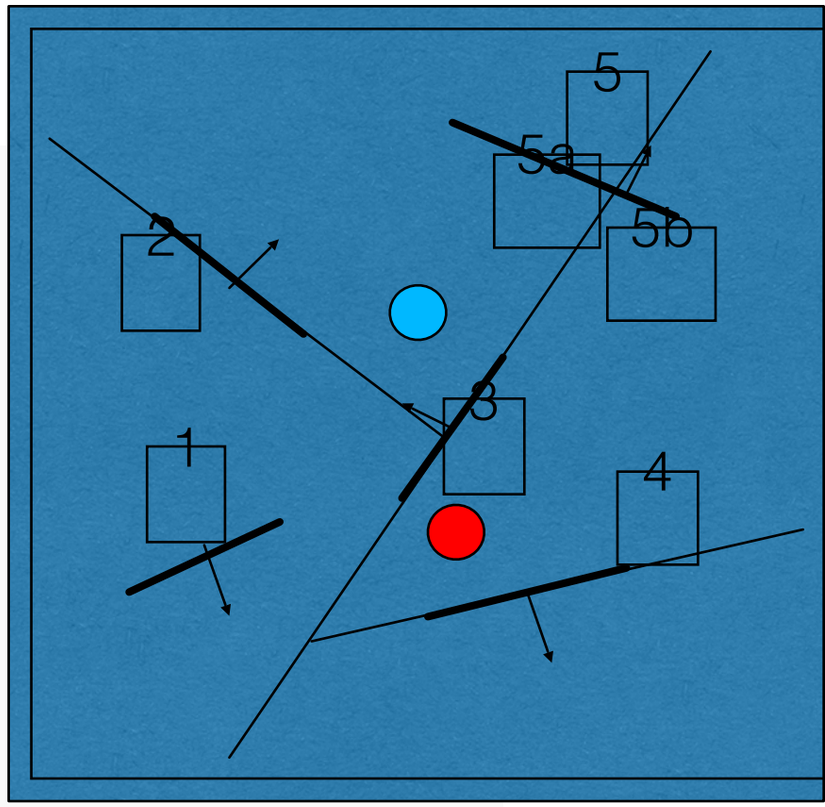
- Start at root polygon.
  - If viewer is in front half-space, draw polygons behind root first, then the root polygon, then polygons in front.
  - If viewer is in back half-space, draw polygons in front of root first, then the root polygon, then polygons behind.
  - Recursively descend the tree.
- If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

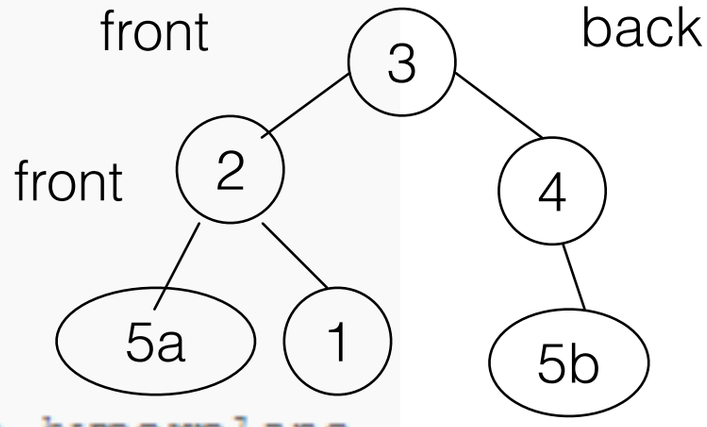
In what order will the faces be drawn?

```
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);
  }
}
```



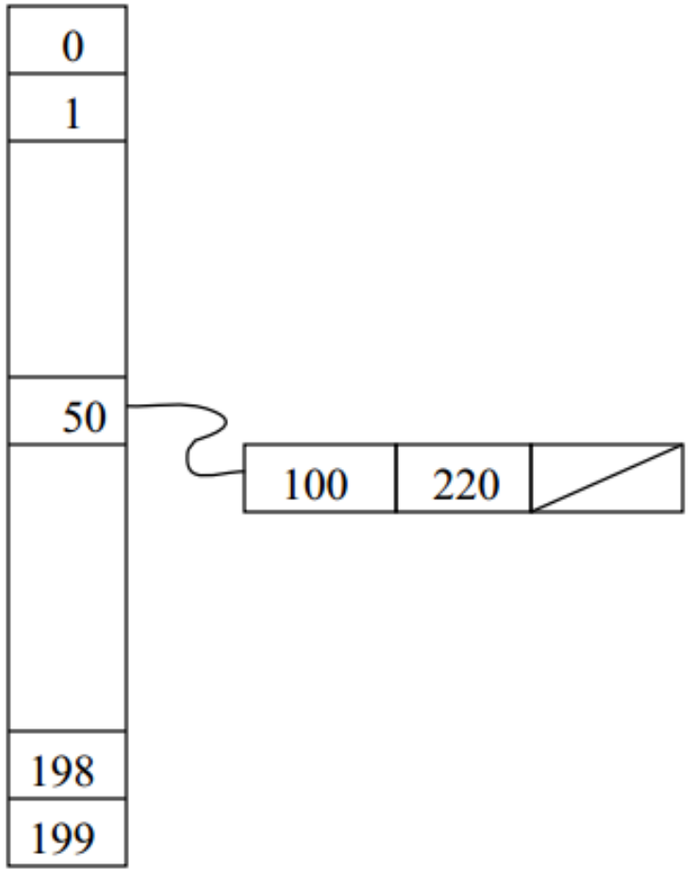


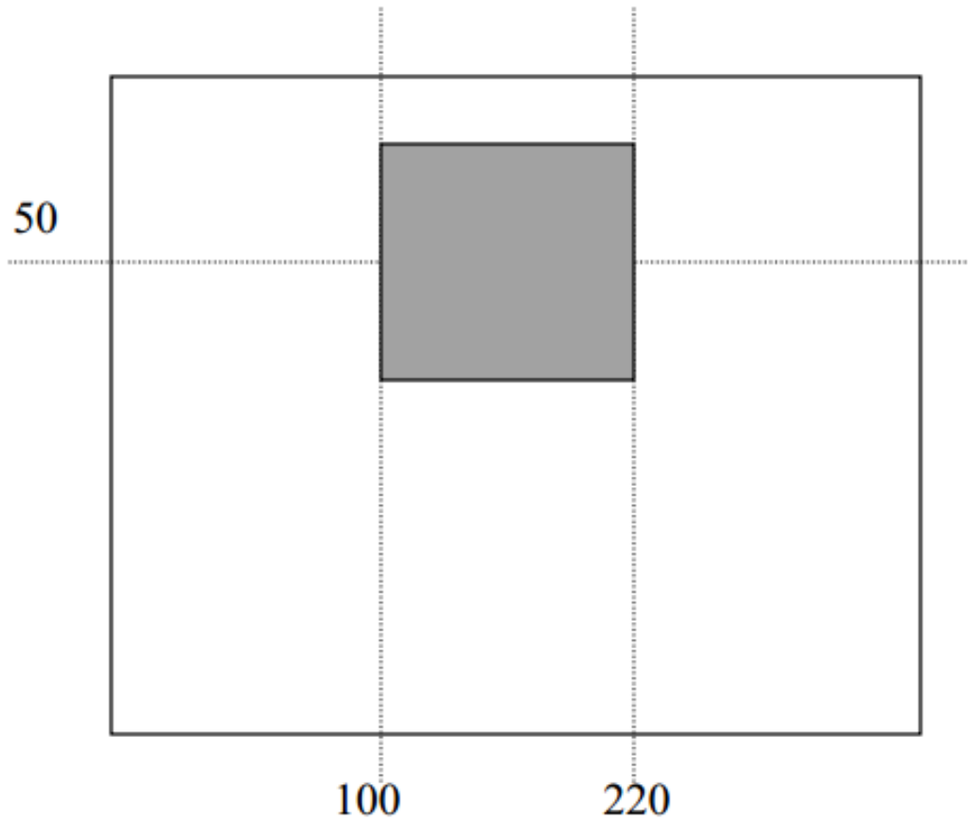
Displaying the BSP tree: front to back

- Back to front rendering will result in a lot of over-drawing
- Front to back traversal is more efficient
- Record which regions of the screen have been filled
- Finish when all regions are filled

Displaying the BSP tree: front to back

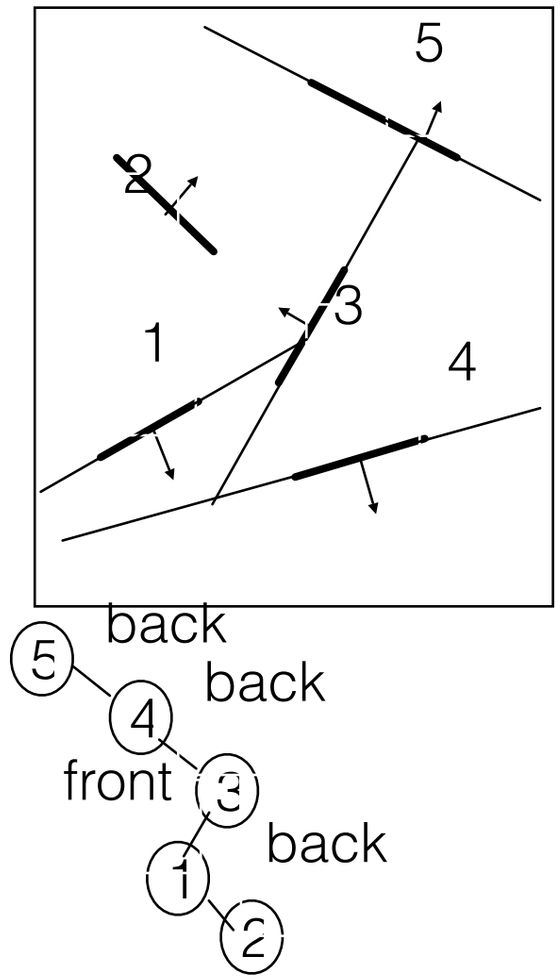
- Use the active edge table in a scanline algorithm
- Record pixels not filled in for each scanline





BSP trees

- Requires a lot of computation to generate the tree
  - Need to produce a balanced tree
  - Need to intersect polygons to split them
- Cheap to check visibility once the tree has been set up
- Efficient when the scene doesn't change often

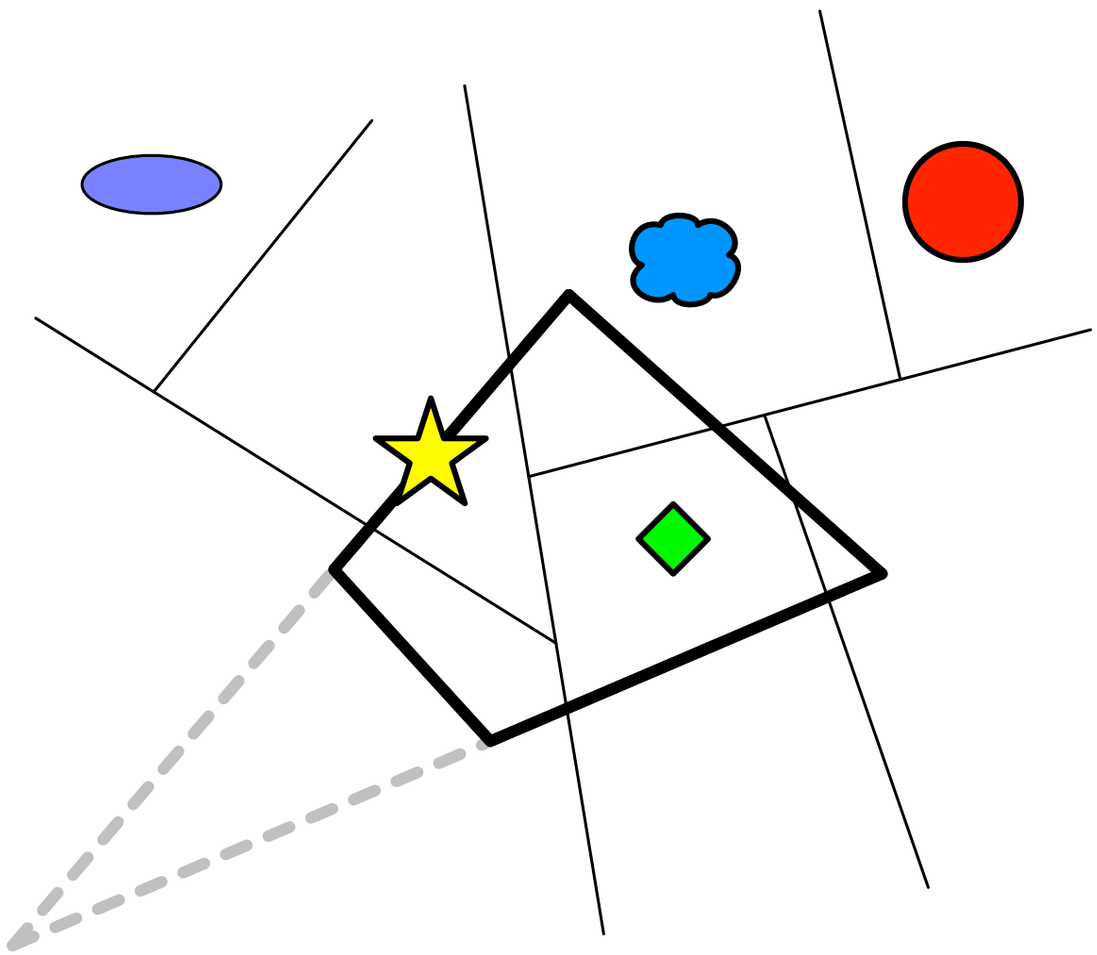


BSP trees

- Combine with Z-buffer
- Render static objects (front to back) with Z-buffer on
- Then draw dynamic objects

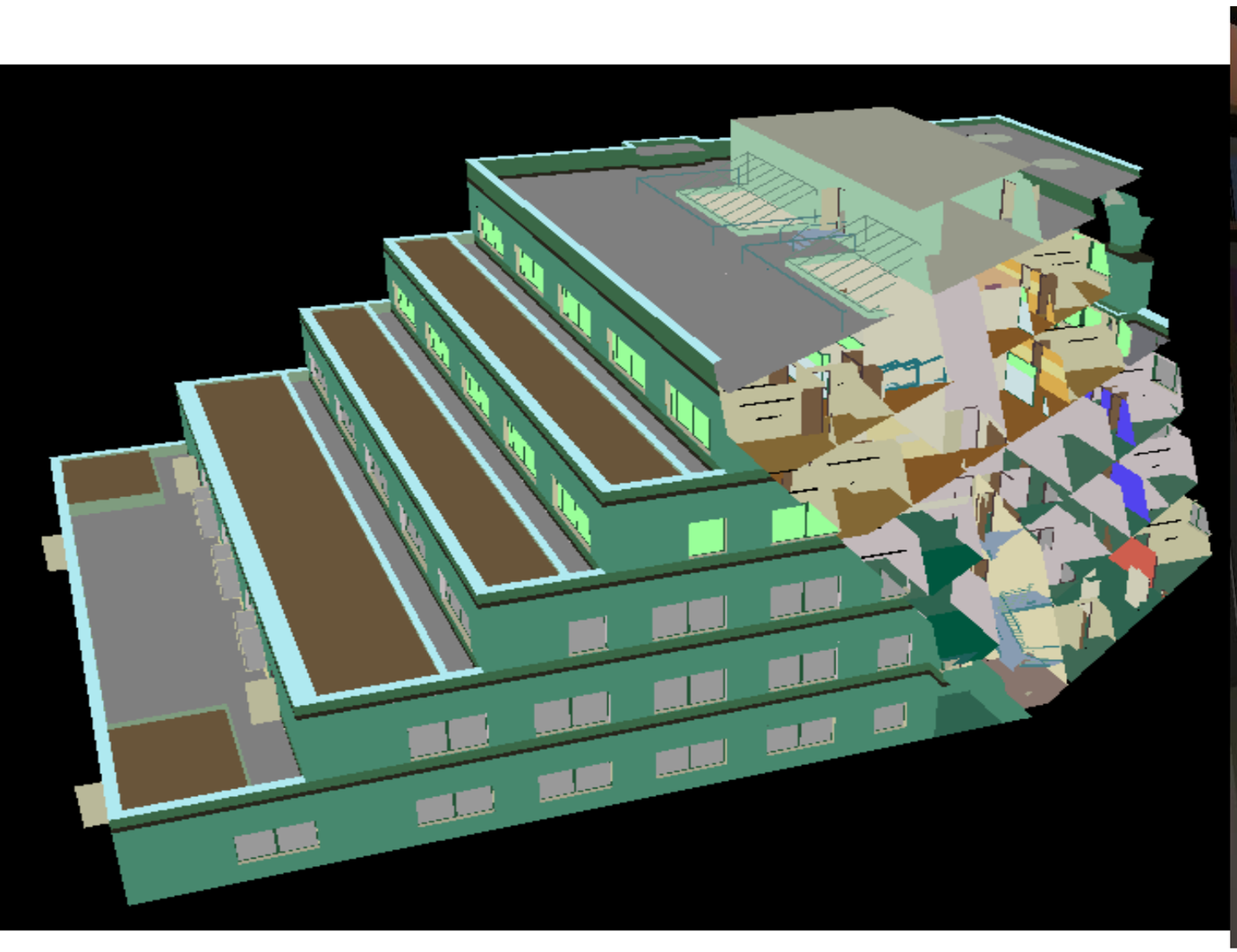
BSP tree visibility culling

- BSP trees can be used to cull polygons that fall outside of the viewing frustum
  - If plane intersects frustum, descend to both children
  - If frustum on one side of plane, cull objects on other side of plane
- Also possible with octrees



Example architectural scenes

- Can have an enormous amount of occlusion



Portal culling

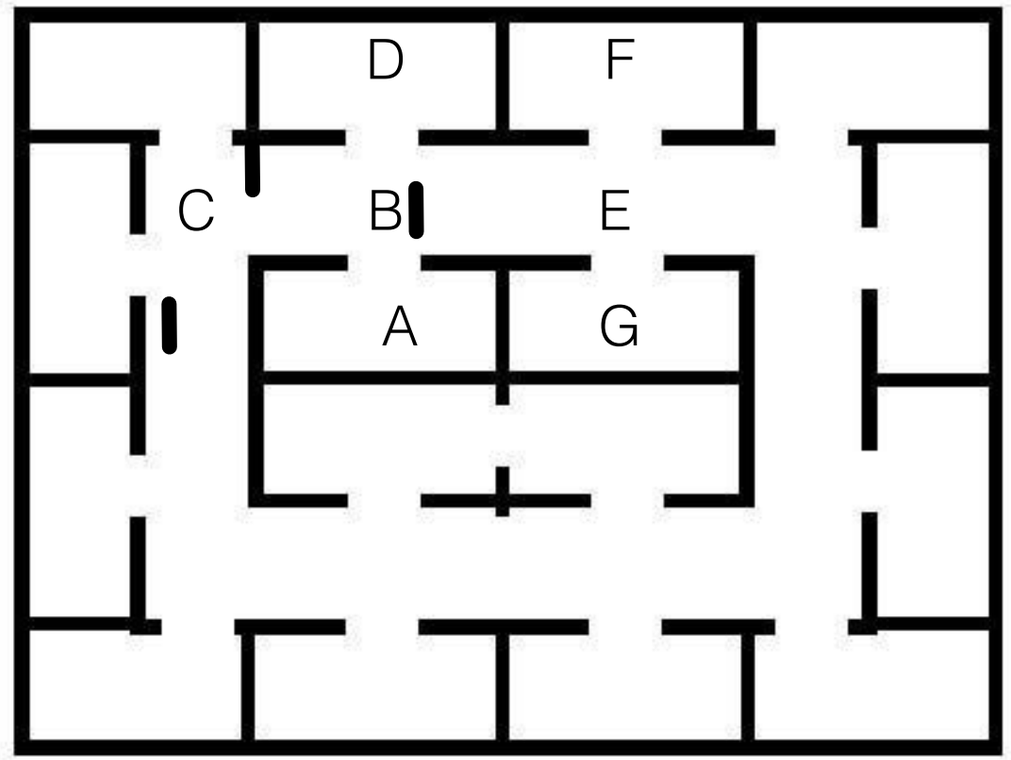
Model scene as a graph:

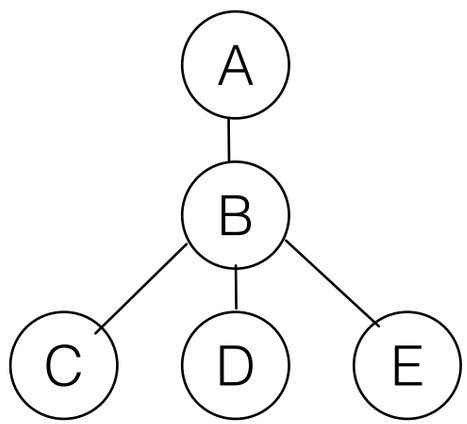
- Nodes: Cells (or rooms)
- Edges: Portals (or doors)

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





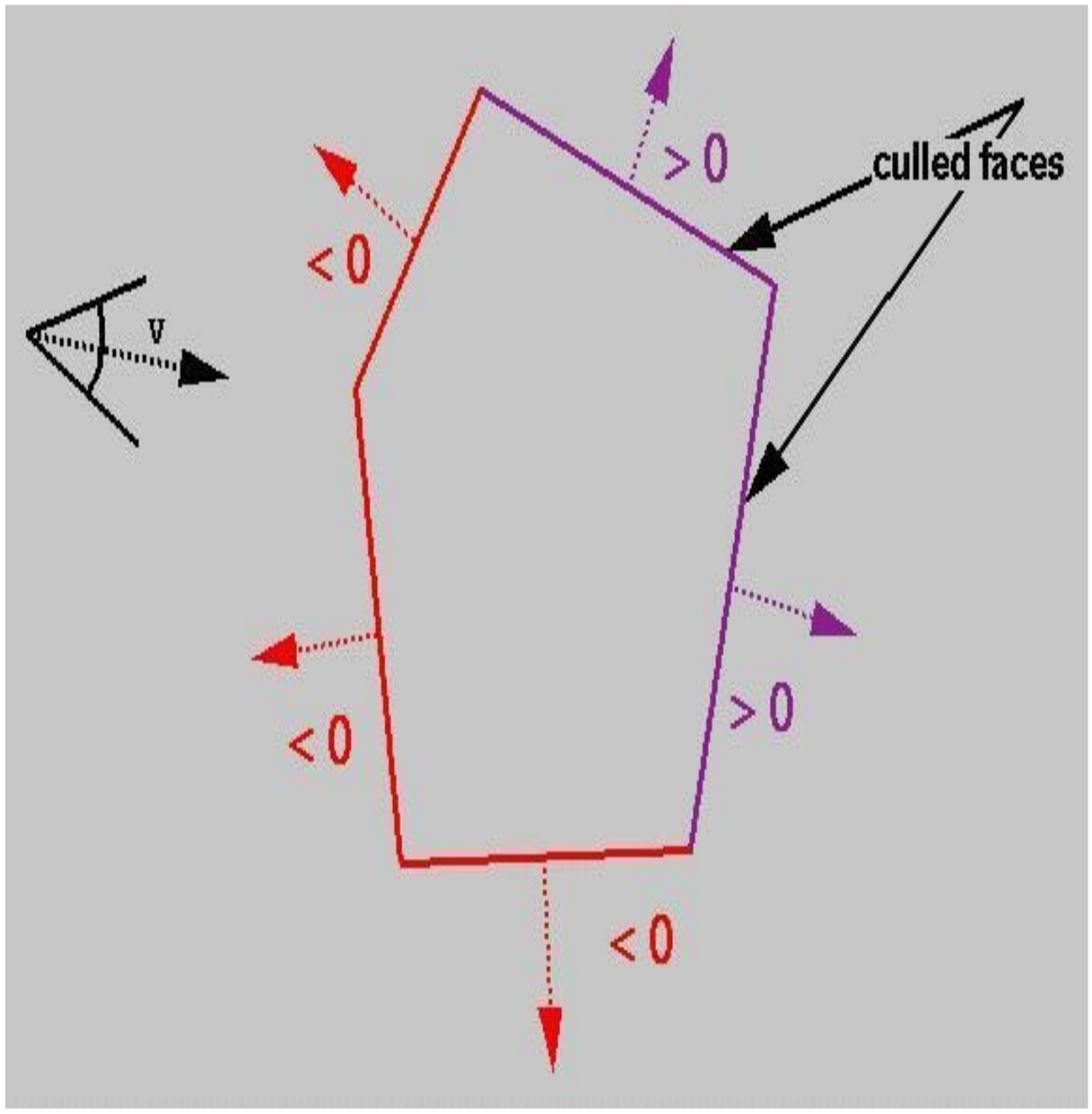
Object space and image space classification

- Object space techniques - applied to mesh geometry:
  - Painter's algorithm BSP trees, portal culling
- Image space techniques - applied when pixels are drawn:
  - 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.



Hidden surface removal summary

- Z-buffer is easy to implement in hardware and is a standard technique

- Need to combine Z-buffers with an object based approach when there are many polygons - BSP trees, portal culling

- Front to back traversal reduces the cost

Overview

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

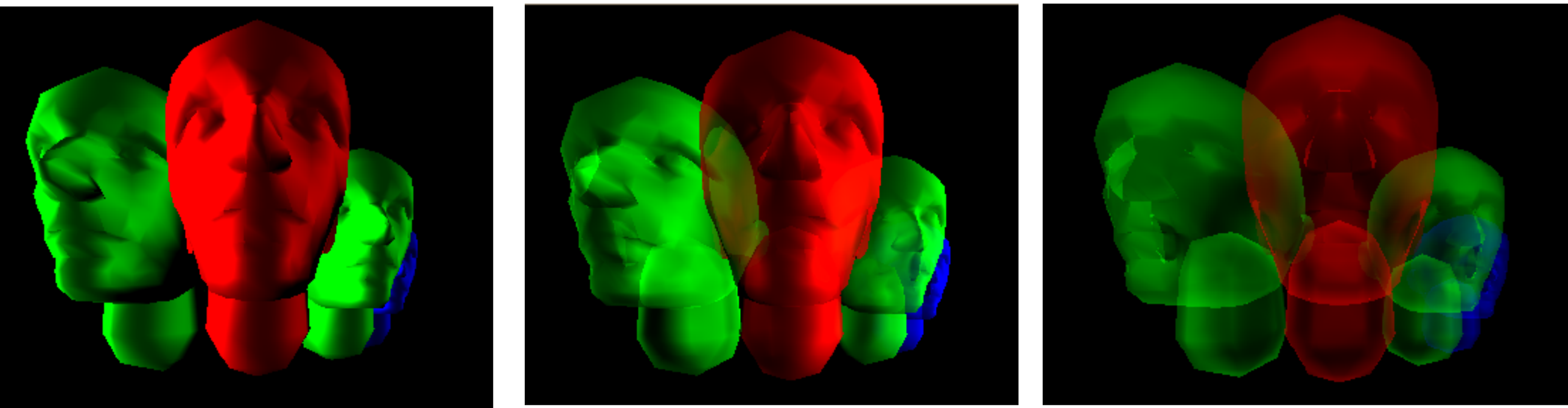
Transparency

- Sometimes we want to draw transparent objects
- We blend the colour of the objects visible at each pixel
- Alpha blending
- Screen-door transparency

Alpha blending

- Alpha values describes the opacity of an object
- 1 means fully opaque
- 0 means fully transparent

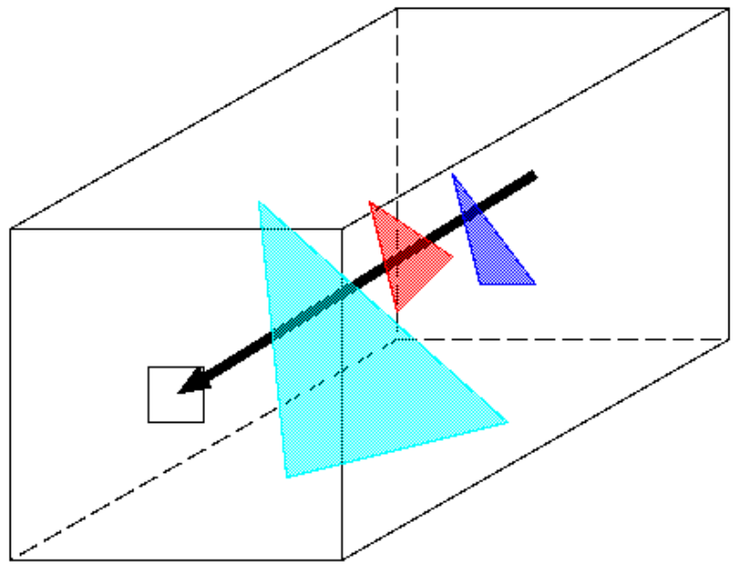
α=1.0 α=0.5 α=0.2



Sorting by depth

- The depth and colour of all fragments that will be projected onto the same pixel is stored in a list

- Blend the colours from back to front



Colour blending

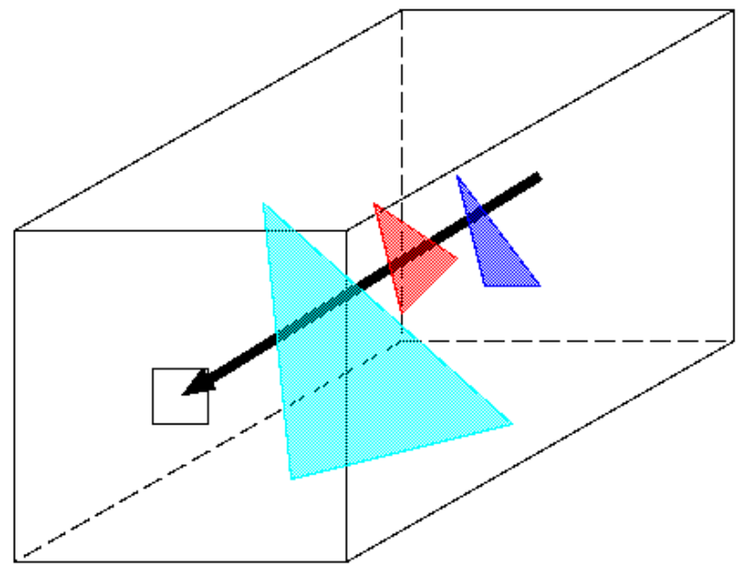
- Colours are blended as follows:

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

$$C_o = \text{New pixel colour}$$

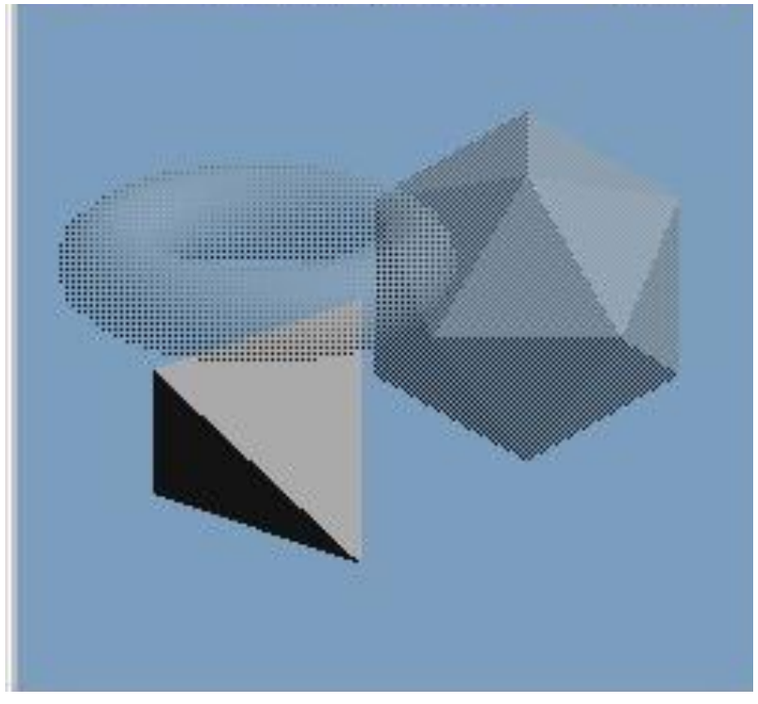
$$C_s = \text{Transparent object colour}$$

$$C_d = \text{Current pixel colour}$$



Sorting

- Sorting is expensive (BSP tree)
- Sorting per pixel is very expensive
- A faster solution - screen door transparency



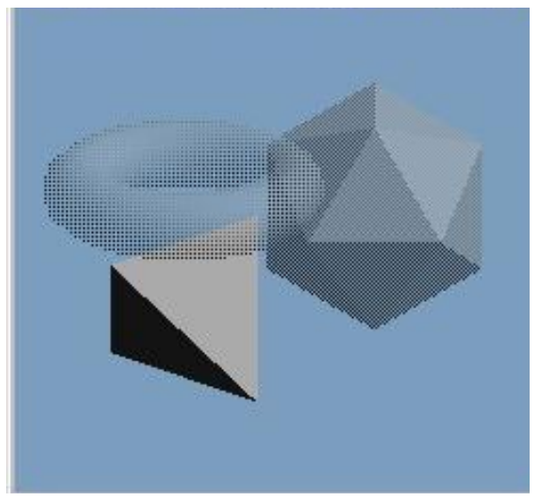
Screen-door transparency

- The object is solid but is drawn with holes using a stipple (checkboard) pattern like a screen-door

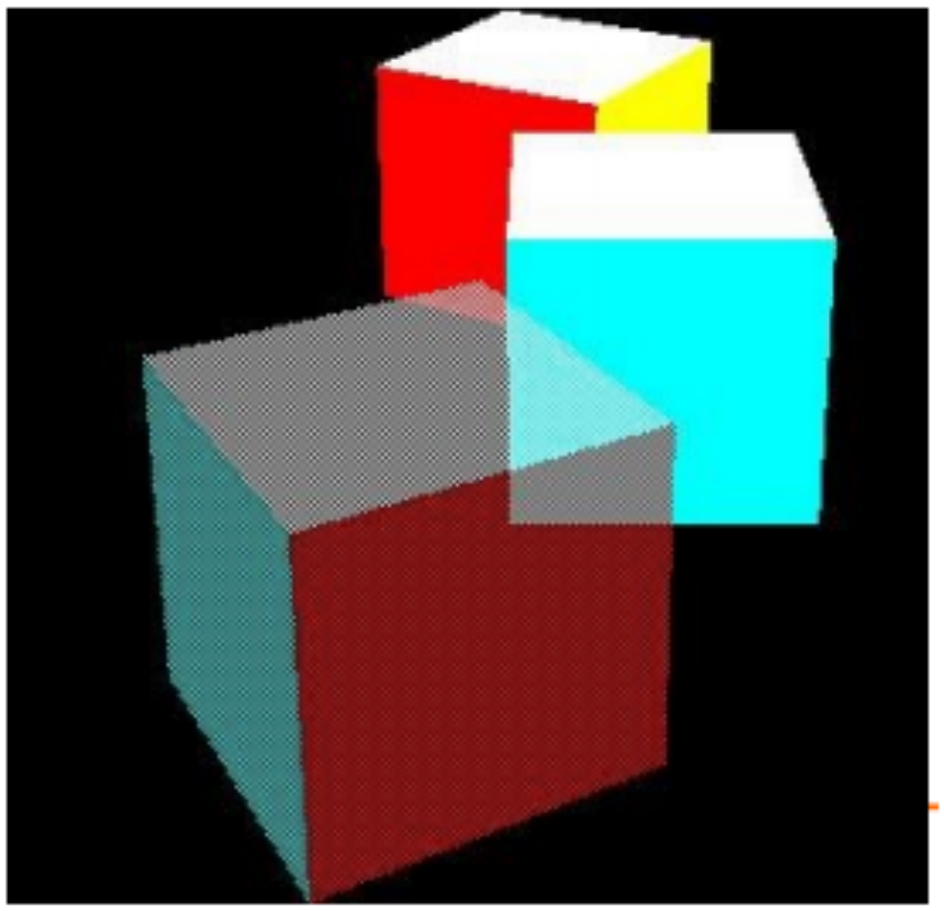
- The ratio of drawn pixels equals the alpha value

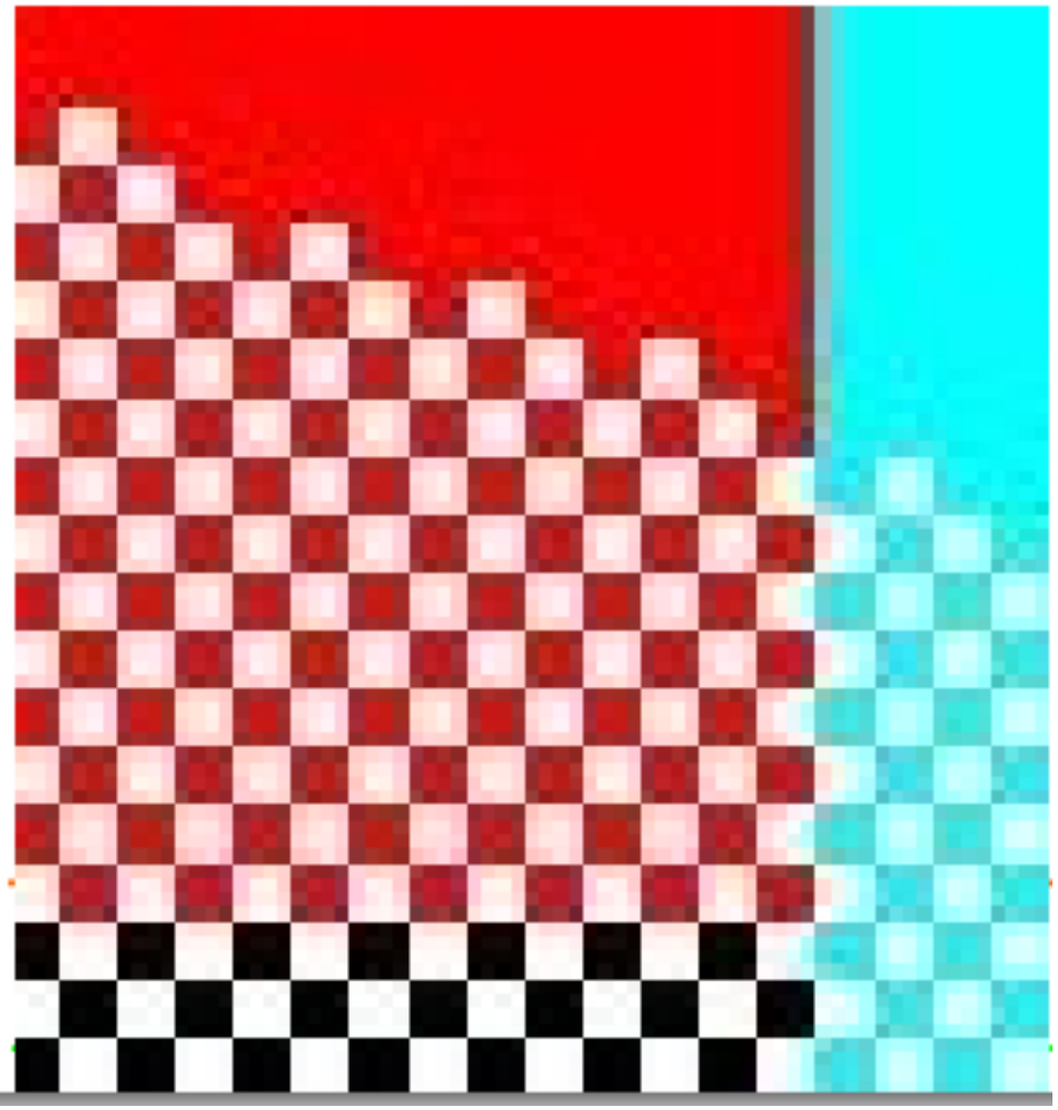
- No need to perform sorting, objects can be drawn in any order

- Z-buffer can handle the overlaps of translucent surfaces

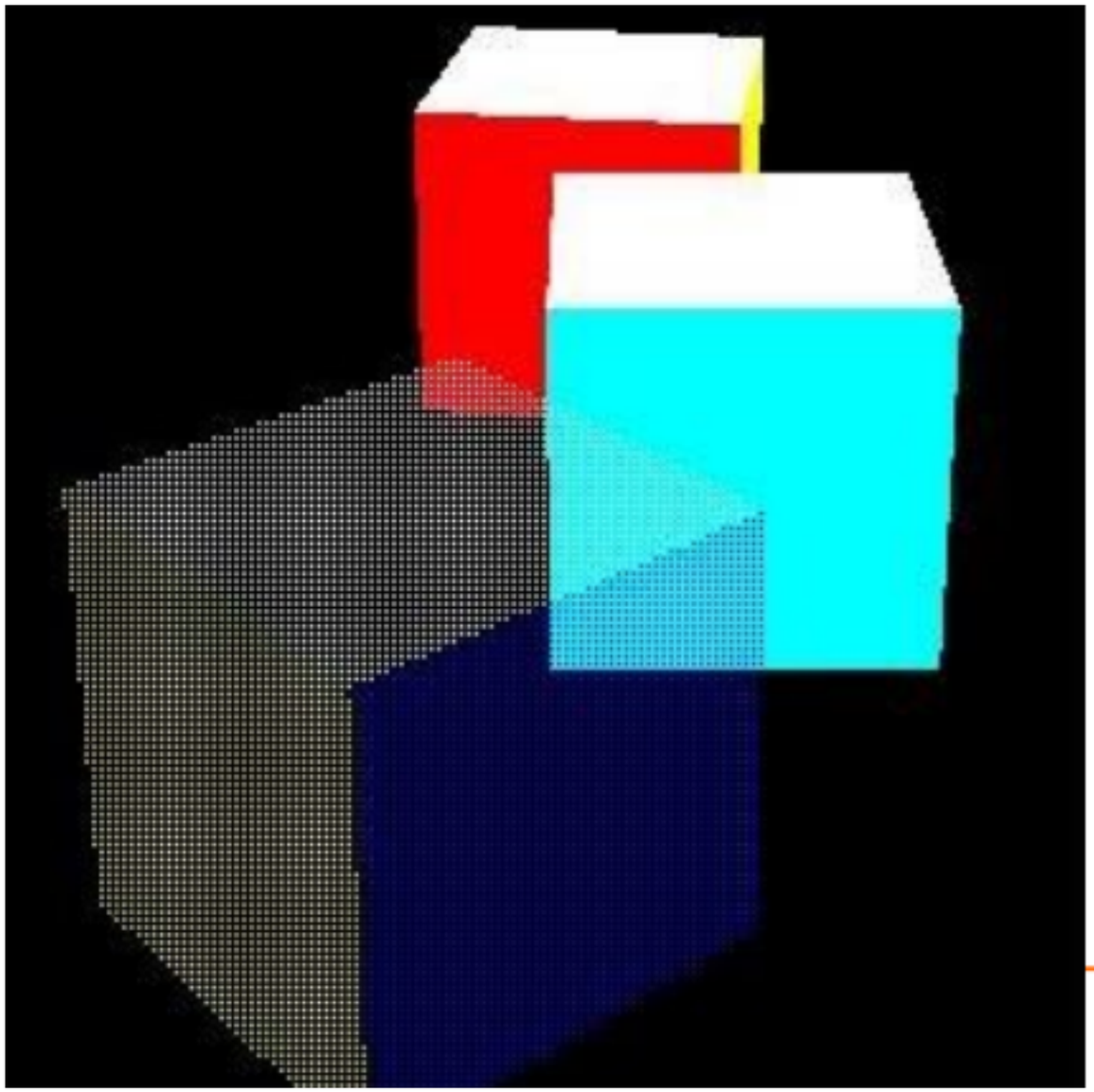


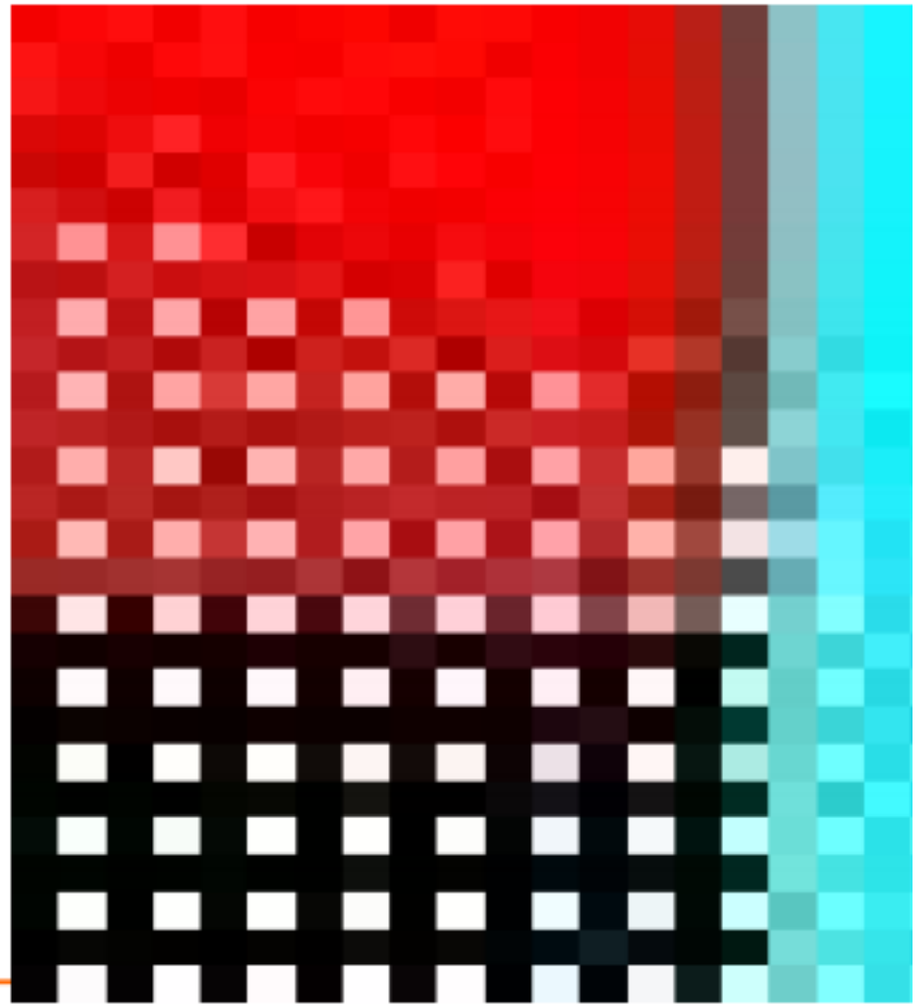
alpha = 0.5





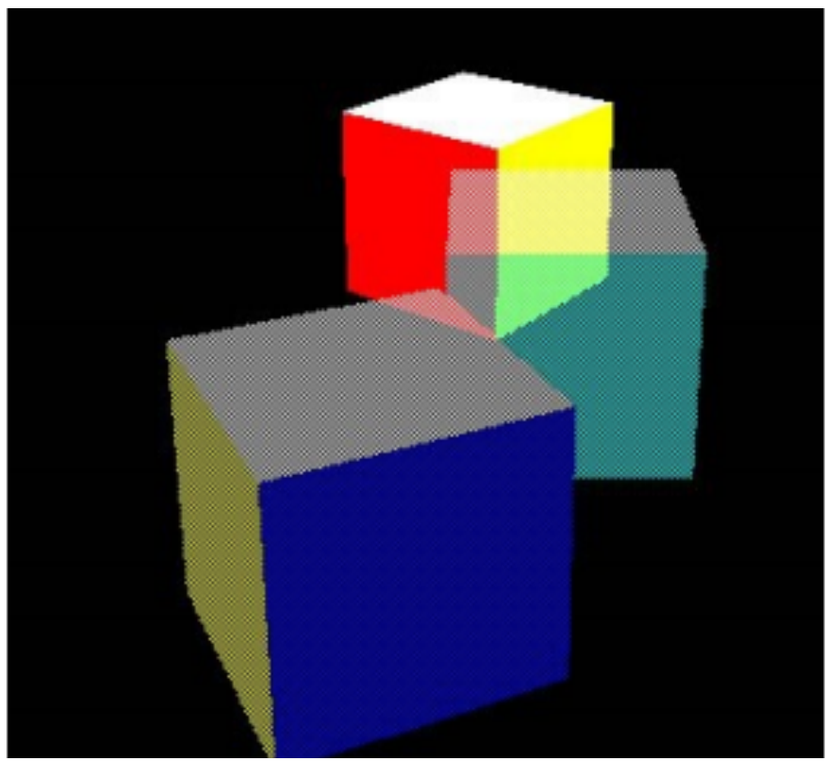
alpha = 0.25

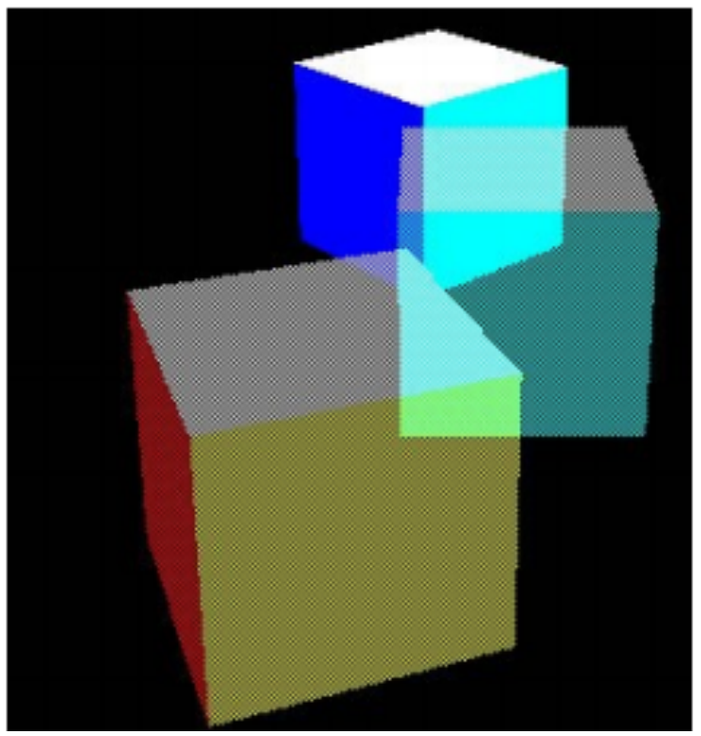




Screen-door transparency

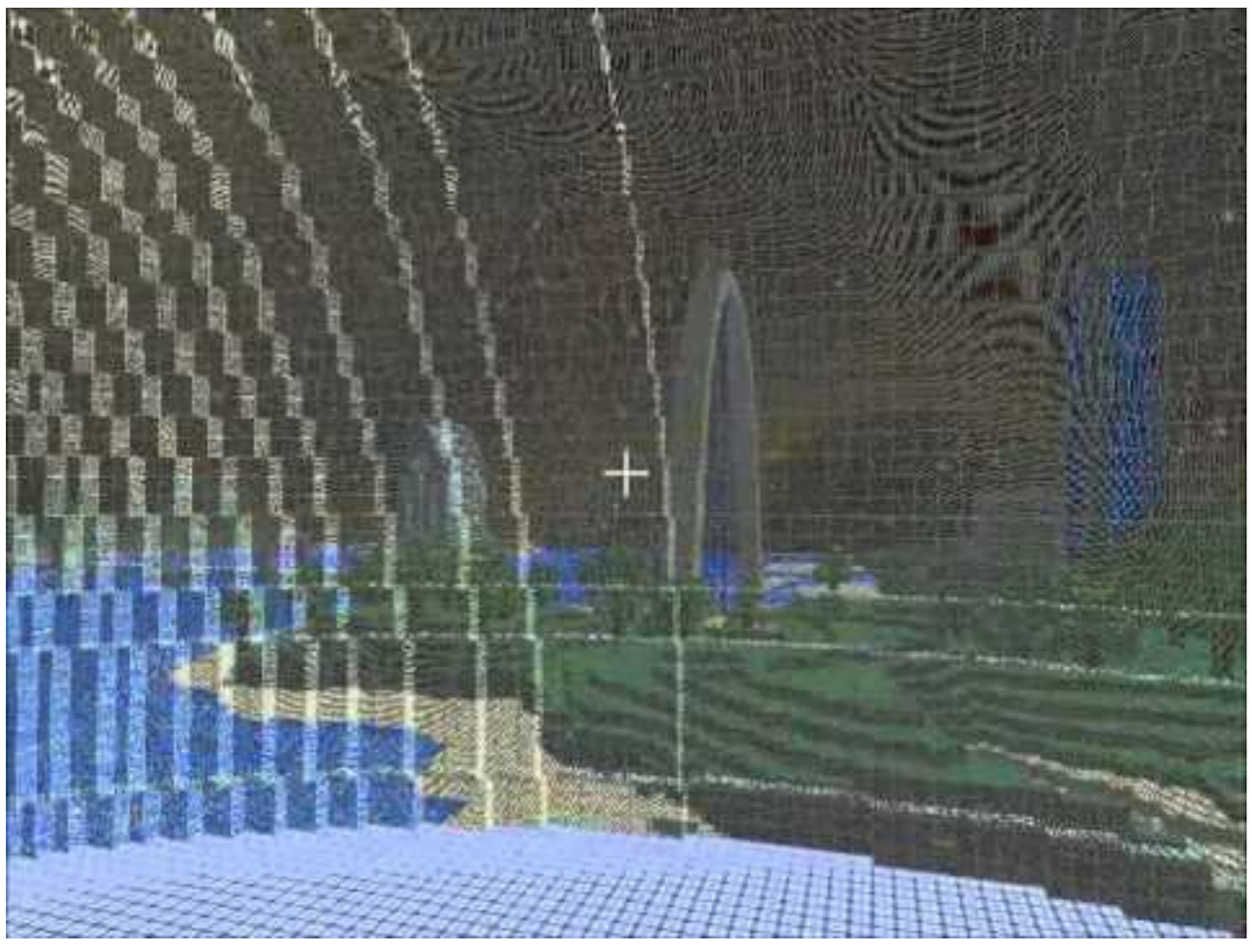
- With a transparent object over another, the transparent object can block everything behind it when the same fixed stipple patterns are used





Screen-door transparency

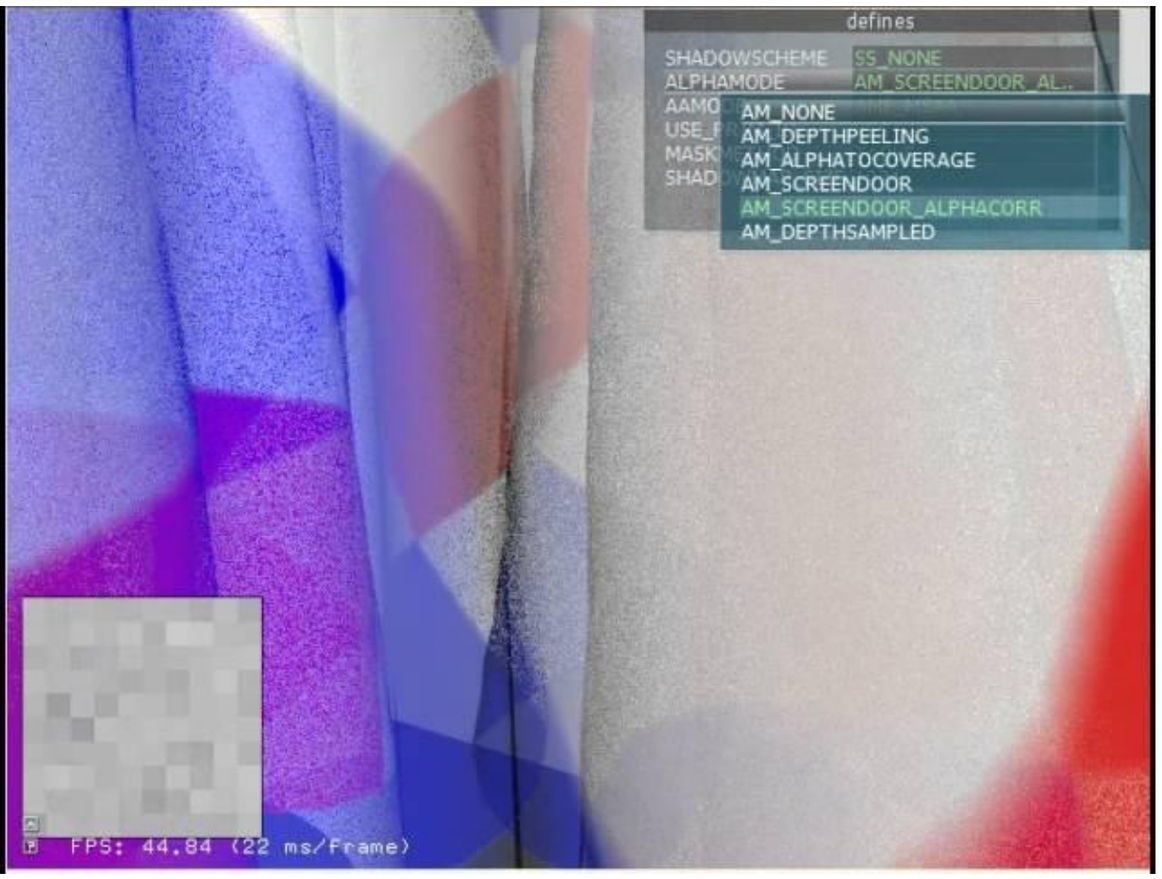
- Stipple patterns need to be set in screen space, otherwise aliasing occurs



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

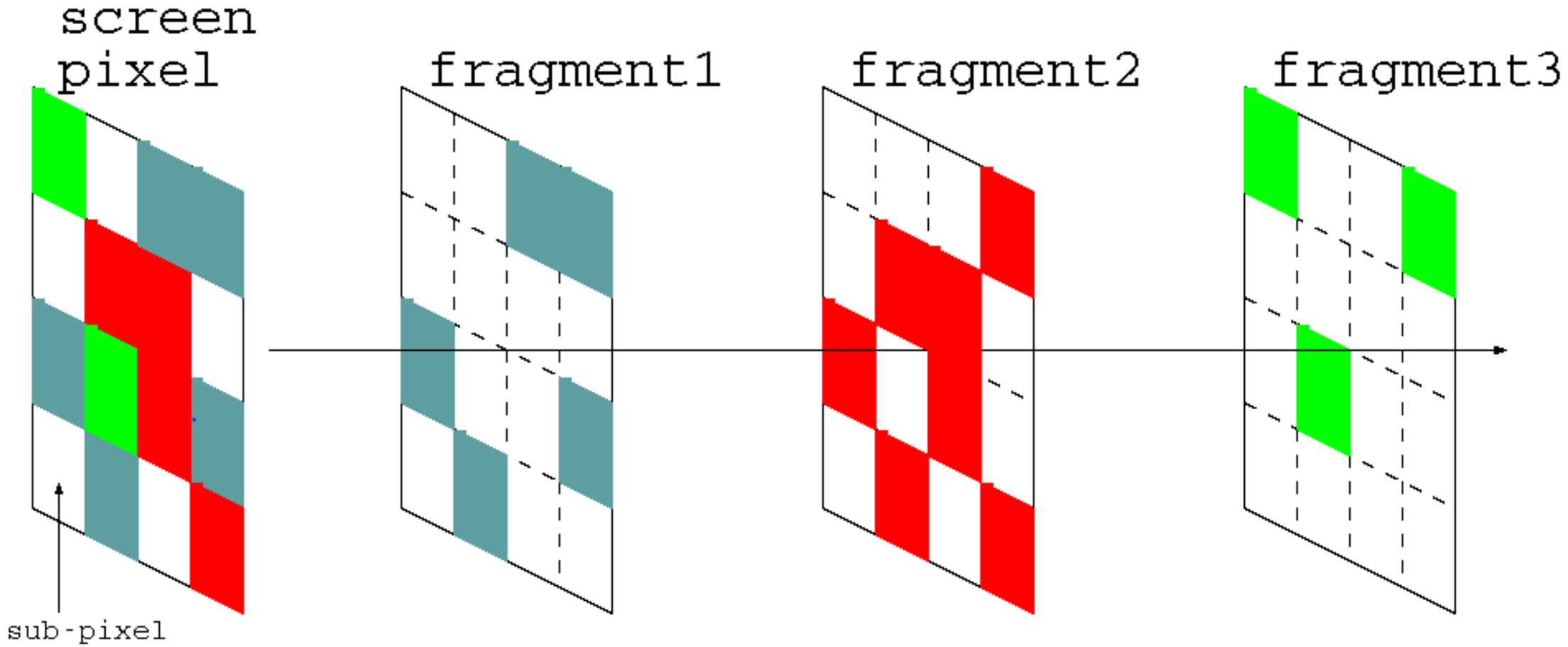
Stochastic transparency

- Using multisampling, sub-pixels are drawn and the pixel colour is computed by averaging their colour
- Uses a random sub-pixel stipple pattern



Stochastic transparency

- No sorting needed
- Final colour of a pixel is calculated by averaging sub-pixel colours



References

- Shirley Chapter 12.4 (BSP trees for visibility)
- Akenine-Möller Chapter 14.1.2 (BSP trees)
- Shirley Chapter 3.4 (Alpha compositing)
- Akenine-Möller Chapter 5.7 (Transparency, Alpha and Compositing)
- Foley, Chapter 15.4, 15.5.1, 15.5.2
- http://research.nvidia.com/publication/stochastic-transparency