Visible-Surface Determination

Introduction

- Not every part of every 3D object is visible to a particular viewer. We need an algorithm to determine what parts of each object should get drawn.
- Known as “hidden surface elimination” or “visible surface determination”.
- Hidden surface elimination algorithms can be categorized in three major ways:
  - Object space vs. image space
  - Object order vs. image order
  - Sort first vs. sort last

Object Space Algorithms

- Operate on geometric primitives
  - For each object in the scene, compute the part of it which isn't obscured by any other object, then draw.
  - Must perform tests at high precision
  - Resulting information is screen resolution-independent
- Complexity
  - Must compare every pair of objects, so $O(n^2)$ for n objects
  - For an m x m display, have to fill in colors for m^2 pixels.
  - Overall complexity can be $O(k_{obj}n^2 + k_{disp}m^2)$.
  - Best for scenes with few polygons or resolution-independent output
- Implementation
  - Difficult to implement!
  - Must carefully control numerical error

Image Space Algorithms

- Operate on pixels
  - For each pixel in the scene, find the object closest to the eye which intersects the projector through that pixel, then draw.
  - Perform tests at device resolution, result works only for that resolution
- Complexity
  - Naïve approach checks all *n* objects at every pixel.Then, $O(n\ m^2)$.
  - Better approaches check only the objects that *could* be visible at each pixel. Let's say, on average, *d* objects are visible at each pixel (a.k.a., depth complexity). Then, $O(d\ m^2)$.
- Implementation
  - Usually very simple!
  - Used a lot in practice.

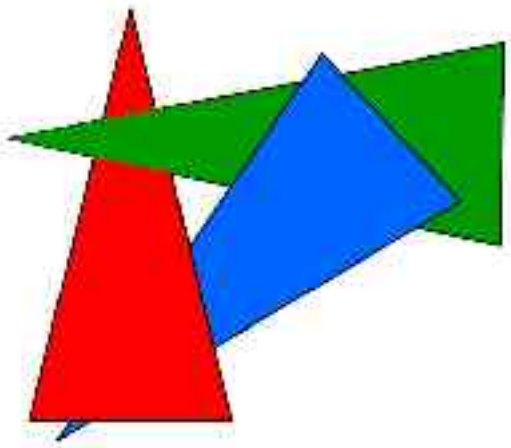
Object Order vs. Image Order

- Object order
  - Consider each object only once - draw its pixels and move on to the next object
  - Might draw the same pixel multiple times

- Image order
  - Consider each pixel only once - draw part of an object and move on to the next pixel
  - Might compute relationships between objects multiple times

Sort First vs. Sort Last

- Sort first
  - Find some depth-based ordering of the objects relative to the camera, then draw from back to front
  - Build an ordered data structure to avoid duplicating work
- Sort last
  - Determine depth observed at each pixel and draw the color corresponding to the closest depth
  - Can be done by considering all depths together or by “lazily” keeping track of depths as they arrive.



Some Definitions

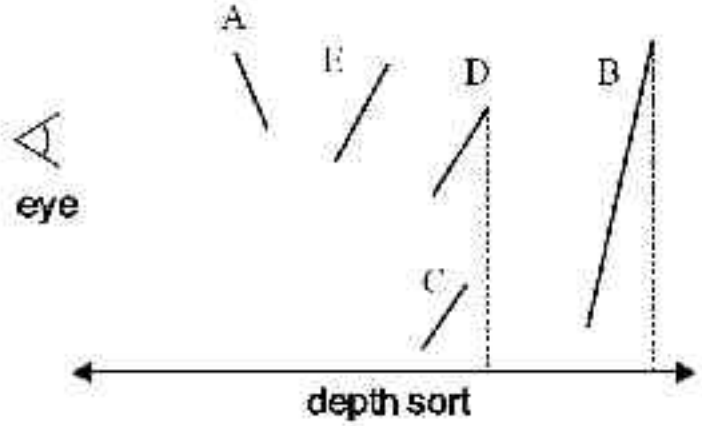
- An algorithm exhibits *coherence* if it uses knowledge about the continuity of the objects on which it operates
- An *online* algorithm is one that doesn't need all the data to be present when it starts running
- Example: insertion sort

Important Algorithms

- Painter`s Algorithm
- Binary space partitioning
- Back face culling
- Z-buffer
- Ray casting
- Scan-line algorithms
- A-buffer
- Area subdivision

Painter`s Algorithm

- Render just as oil painting:
  - Sort surfaces in order of decreasing maximum depth
  - Scan convert surfaces in back-to-front order
  - Not a complete solution

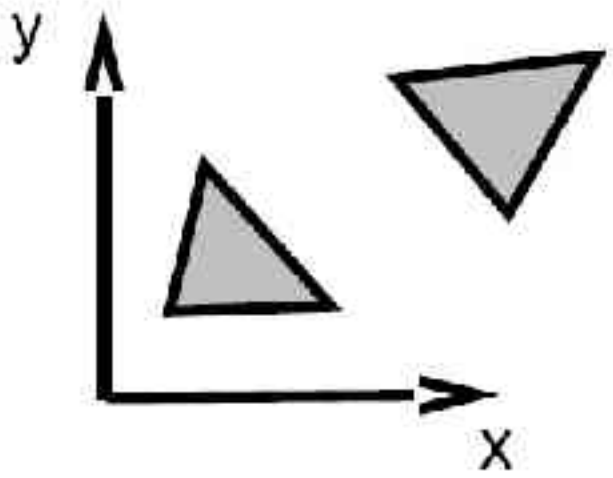


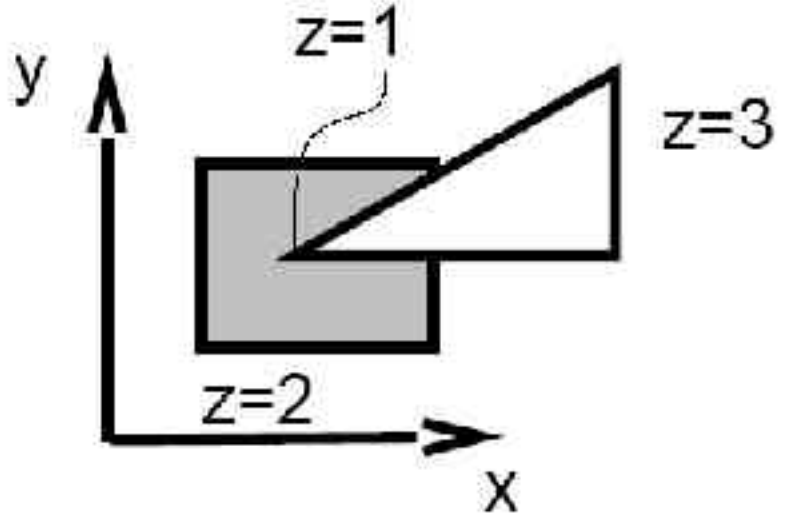
Depth sort comments

- O(n log n)
- Better with frame coherence?
- Implemented in software
- Render every polygon
- Often use BSP-tree or static list ordering

Painter`s Algorithm

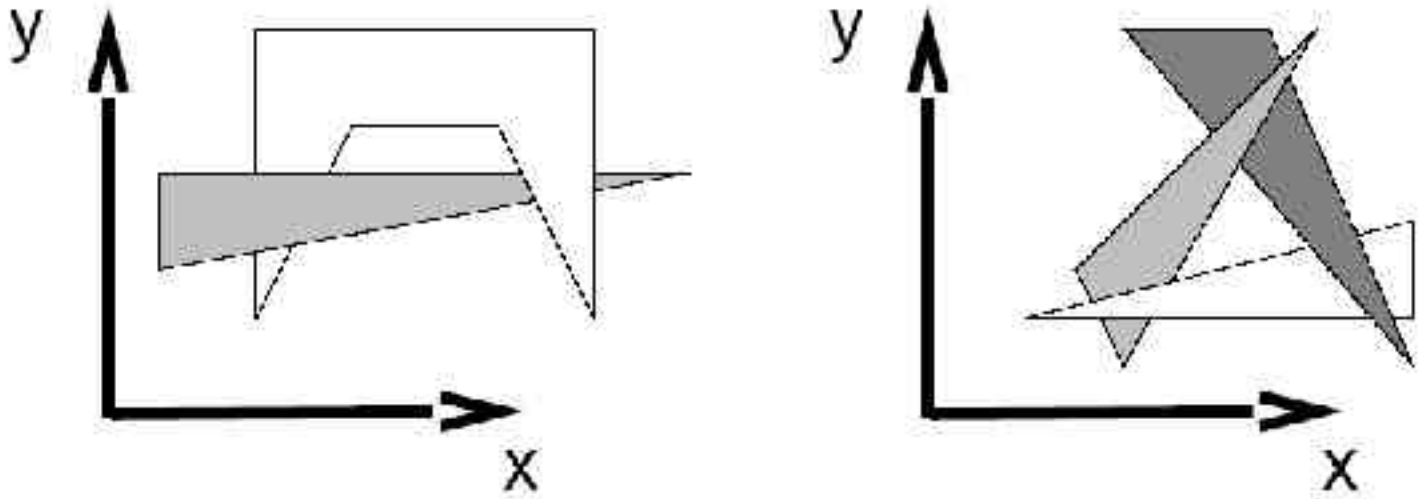
- Works correctly the following cases.
  Simple order arrangement, no penetration.





Painter`s Algorithm

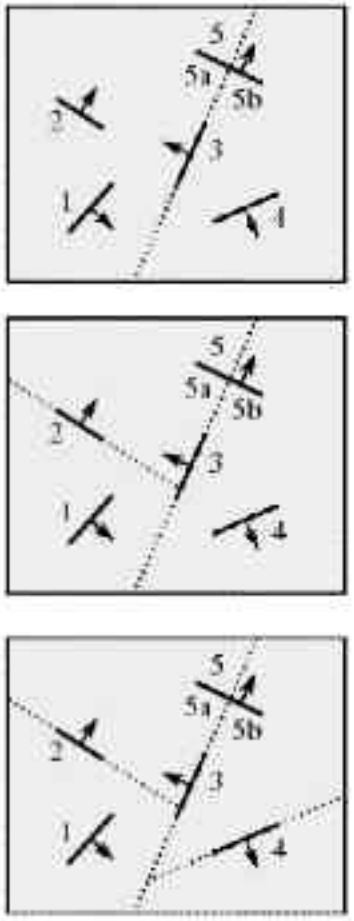
- But fails for the some cases (penetrating faces or cyclically overlapping)



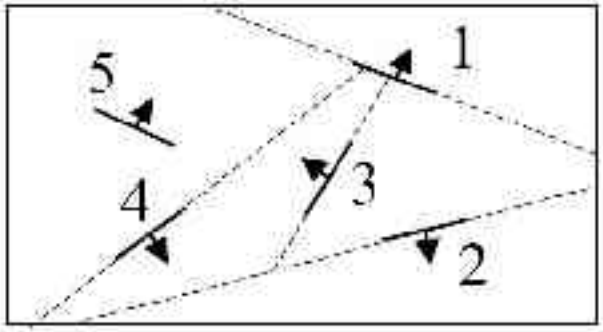
Binary Space Partitioning

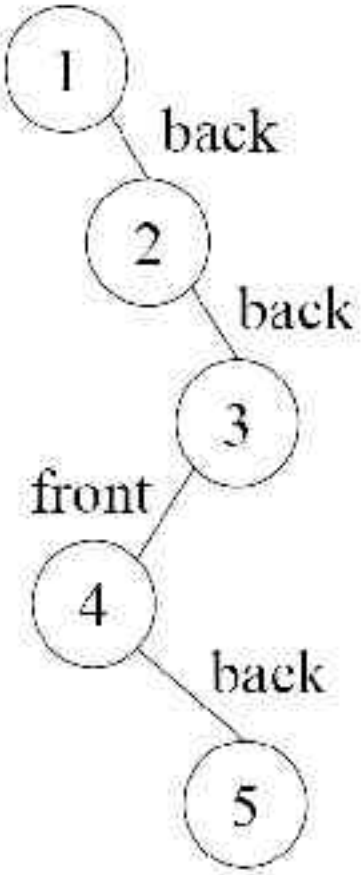
- **Goal**: build a structure that captures some relative depth information between objects. Use it to draw objects in the right order from any viewpoint.
- Called the binary space partitioning tree, or BSP tree
- **Key observation**: The polygons in the scene are painted in the correct order if for each polygon *P*,
  - Polygons on the far side of *P* are painted first
  - *P* is painted next
  - Polygons in front of *P* are painted last

Building a BSP Tree (in 2D)



Alternate BSP Tree





BSP Tree Construction

```
BSPtree makeBSP( L: list of polygons )
{
    if L is empty
    {
        return the empty tree
    }

    Choose a polygon P from L to serve as root
    Split all polygons in L according to P
    return new TreeNode(
        P,
        makeBSP( polygons on negative side of P ),
        makeBSP( polygons on positive side of P ))
    }
}
```

- **Note**: Performance is improved when fewer polygons are split --- in practice, best of ~ 5 random splitting polygons are chosen.
- **Note**: BSP is created in *world* coordinates. No projective matrices are applied before building tree

BSP Tree Display

```
showBSP( v: Viewer, T: BSPtree )
{
    if T is empty then return
    P := root of T

    if viewer is in front of P
    {
            showBSP( back subtree of T )
            draw P
            showBSP( front subtree of T )
    } else {
            showBSP( front subtree of T )
            draw P
            showBSP( back subtree of T )
    }
}
```

BSP Tree Applications

- Hidden surface removal
- Ray casting speedup
- Collision detection
- …

BSP Analysis

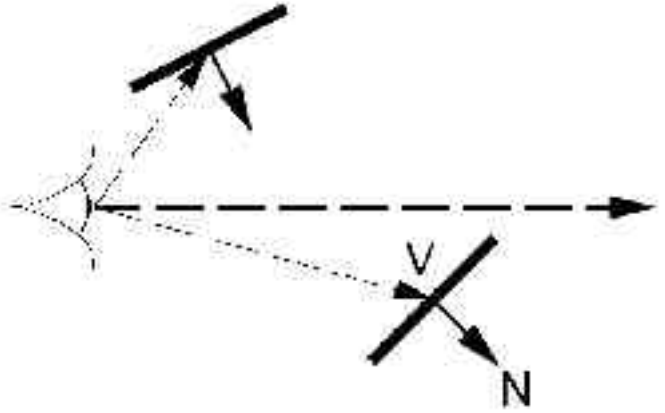
- Easy to implement?
- Hardware implementation?
- Pre-processing required?
- Incremental drawing calculations (uses coherence)?
- On-line (doesn’t need all objects before drawing begins)?
- Memory intensive?
- Handles transparency and refraction?
- Polygon-based?
- Extra work for moving objects?
- Extra work for moving viewer?
- Efficient shading?
- Handles cycles and self-intersections?

Back Face Culling

- Often, we don't want to draw polygons that face away from the viewer. So test for this and eliminate (cull) backfacing polygons before drawing

•Can be applied after world transformation:

if V.N < 0 front facing, else back facing



Back Face Culling

- Backface culling is not a complete solution:
  - If objects not convex, need to do more work.
  - Not suitable for transparent faces
  - If polygons two sided (i.e., they do not enclose a volume) then we can't use it.
- On the other hand,
  - A HUGE speed advantage if we can use it since the test is cheap and we expect at least half the polygons will be discarded.
  - Usually performed in conjunction with a more complete hidden surface algorithm.
  - Easy to integrate into hardware (and usually improves performance by a factor of 2).

Back Face Culling - OpenGL

- OpenGL may utilize back face culling. And it`s applied after the projection transformation and primitive assembly, just before the rasterization takes place.
- Signed area of the triangle computed using the projected x,y coords are used to determine front and back facing polygons
- Back face culling eliminates unnecessary rasterization (and thus shading).
- OpenGL has some states controlling the back face culling such as:
  - glFrontFace(GL_CCW);
  - glCullFace (GL_FRONT);
  - glEnable(GL_CULL_FACE);

Z-buffer

- Idea: along with a pixel's color values, maintain some notion of its *depth*
  - An additional channel in memory, like alpha
  - Called the depth buffer or Z-buffer

- When the time comes to draw a pixel, compare its depth with the depth of what's already in the framebuffer Replace only if it's closer

- Very widely used
- History
  - Originally described as "brute-force image space algorithm"
  - Written off as impractical algorithm
  - Today easily done in hardware

Z-buffer Implementation

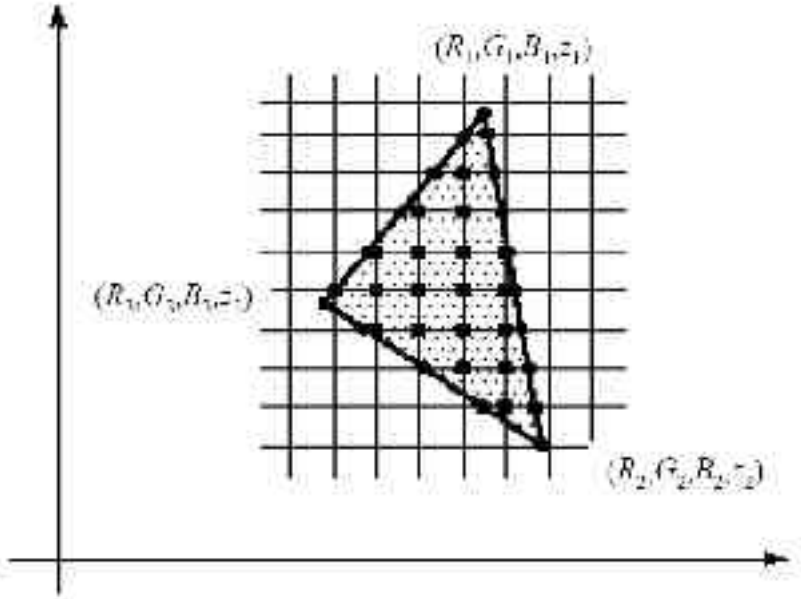
```
for each pixel pi
{
    Z-buffer[ pi ] = FAR
    Fb[ pi ] = BACKGROUND_COLOR
}

for each polygon P
{
    for each pixel pi in the projection of P
    {
        Compute depth z and shade s of P at pi
        if z < Z-buffer[ pi ]
        {
            Z-buffer[ pi ] = z
            Fb[ pi ] = s
        }
    }
}
```

Cost of Z-buffering

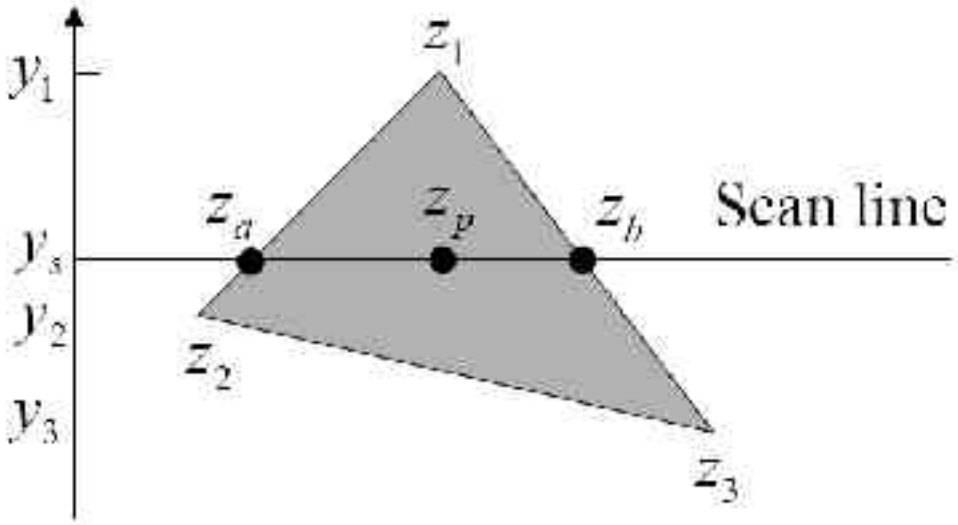
- Z-buffering is ***the*** algorithm of choice for hardware rendering, so let's think about how to make it run as fast as possible…
- The steps involved in the Z-buffer algorithm are:
  - Send a triangle to the graphics hardware.
  - Transform the vertices of the triangle using the modeling matrix.
  - Shade the vertices.
  - Transform the vertices using the projectionmatrix.
  - Set up for incremental rasterizationcalculations
  - Rasterize and update the framebufferaccording to *z*.

Z-buffer Rasterization

- The shade of a triangle can be computed by linearly interpolating the shades of its vertices
- Can do the same with depth values of the vertices



Z value interpolation



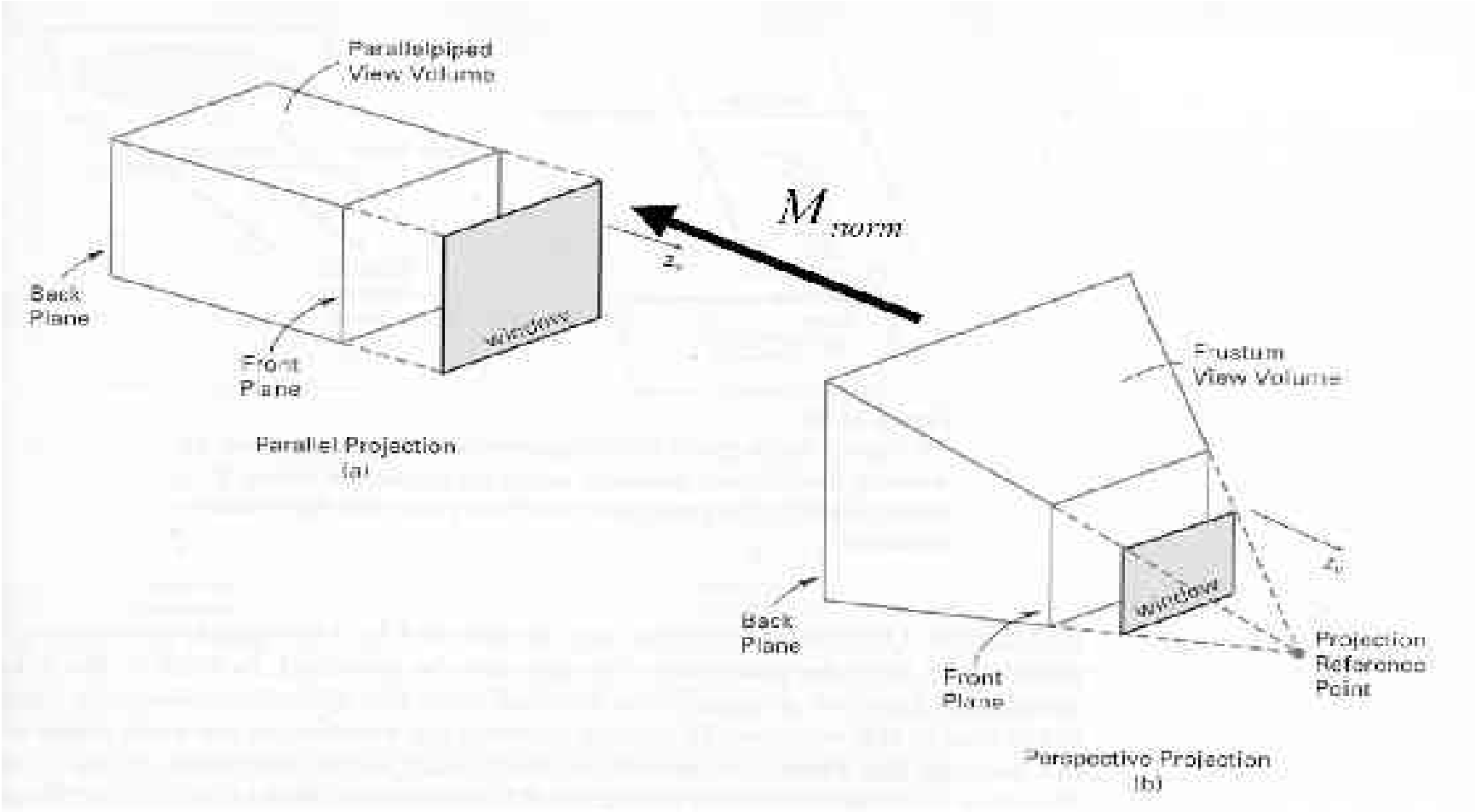
$$z_a = z_1 - (z_1 - z_2)\frac{y_1 - y_s}{y_1 - y_2}$$

$$z_b = z_1 - (z_1 - z_3)\frac{y_1 - y_s}{y_1 - y_3}$$

$$z_p = z_b - (z_b - z_a)\frac{x_b - x_p}{x_b - x_a}$$

- *Devise equations for the incremental computation ! Hint: read the book*

Depth Preserving Conversion to Parallel Projection



Z-buffer Analysis

- Easy to implement?
- Hardware implementation?
- Pre-processing required?
- Incremental drawing calculations (uses coherence)?
- On-line (doesn’t need all objects before drawing begins)?
- Memory intensive?
- Handles transparency and refraction?
- Polygon-based?
- Extra work for moving objects?
- Extra work for moving viewer?
- Efficient shading?
- Handles cycles and self-intersections?

Z-buffer Precision

- Precision is increased using more depth bits per pixel. Typically 24, 16 or 32 bits for hardware implementations.
- After of the homogeneous division (z component, after z/w), the depth precision is not uniform for the scene:
  - Closer points have more depth precision than distant points.
  - This may be a problem for large scenes with detailed objects. Typically depth fighting occurs in such cases

- We can use the w component instead of depth for each pixel instead. W values are uniform. This is also called as w buffering and works better for large scenes

Z-Buffering - OpenGL

- OpenGL`s hidden surface removal heavily depends of z buffering.
- Z buffer is a separate buffer with the same dimensions of color buffer. A depth buffer should be created before using it. Such as:
  - glutInitDisplayMode (GLUT_RGB | **GLUT_DEPTH | …**);

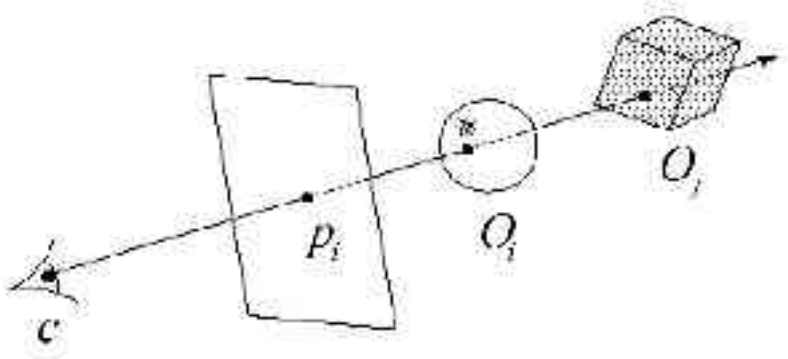
- Z buffer writes and tests can be enabled independently from the color buffer.
- Z buffer is controlled using some GL states such as:
  - glEnable(GL_DEPTH_TEST);
  - glDepthFunc(GL_LESS_OR_EQUAL);
  - glDepthMask (GL_TRUE);

Z-Buffering - Questions

- Should we draw from back to front or front to back, or is it a good idea to sacrifice the “online” nature of the z-buffer ?
- How can I find good occluders ?
- How can I increase the depth precision ?
- Is it able to handle transparent surfaces correctly ?
- In which step of the pipeline should we perform the Z buffer test ?

Ray Casting

1. Partition the projection plane into pixels to match screen resolution:
2. For each pixel *pi*, construct ray from COP through PP at that pixel and into scene
3. Intersect the ray with every object in the scene
4. Color the pixel according to the object with the closest intersection



Ray Casting, cont.

- Parameterize each ray:

  $\mathbf{r}(t) = \mathbf{c} + t\,(\boldsymbol{P}ij - \mathbf{c})$

- Each object *Oi* returns t*i* >1 such that first intersection with O*i* occurs at **r**(*ti*).

Q: Given the set {*ti*} what is the first intersection point?

Ray Casting Analysis

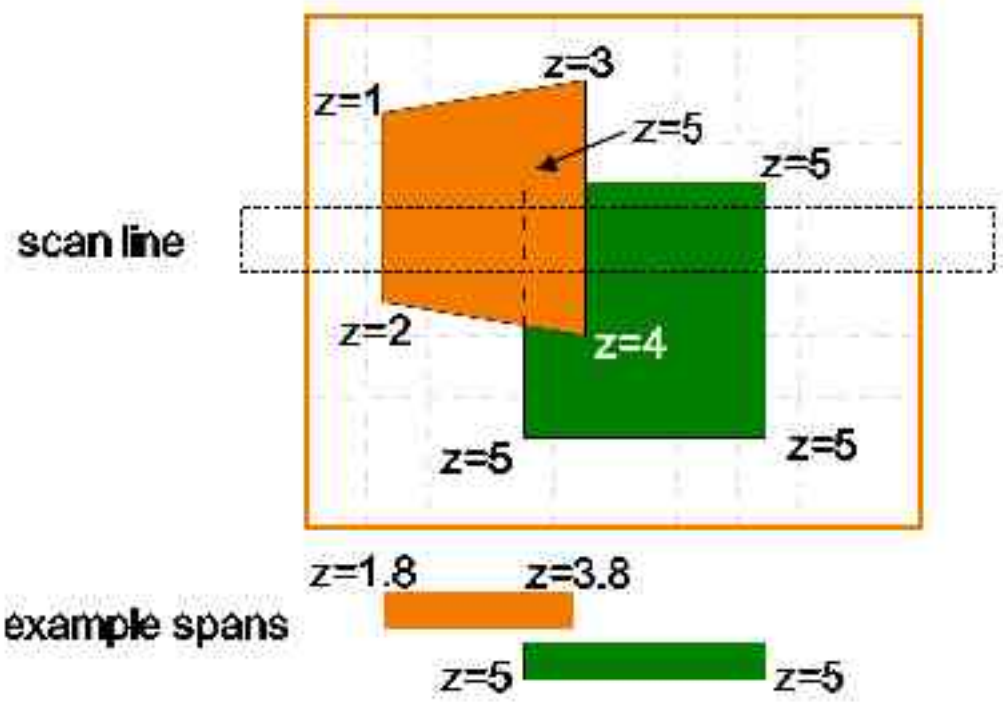
- Easy to implement?
- Hardware implementation?
- Pre-processing required?
- Incremental drawing calculations (uses coherence)?
- On-line (doesn’t need all objects before drawing begins)?
- Memory intensive?
- Handles transparency and refraction?
- Polygon-based?
- Extra work for moving objects?
- Extra work for moving viewer?
- Efficient shading?
- Handles cycles and self-intersections?

Scan Line Method

- Is an extension of the scan-line algorithm for filling polygon interiors. So this is an image-space method
- Instead of filling one surface, we now deal with multiple surfaces
- As each scan line is processed, all polygon surfaces intersecting that line are examined to determine which are visible
- Across each scan line, depth calculations are made for each overlapping surface segment to determine which is closer. Closer segment is shaded and written to the frame buffer

Scan Line Method

- For each scan line, construct spans
  - Sort by depth



A Buffer

- Z buffer works incorrectly for transparent faces:
  - To solve the problem:
    - render opaque faces in the first phase, and render transparent faces from back to front in the second phase
  - Still not correct for penetrating transparent faces. And Z buffering is no more online for transparent faces
- A Buffer represents *antialiased, area-averaged, accumulation buffer* method developed by Lucasfilm for implementation in the REYES rendering system.
- A Buffer store extra information aside from depth for each pixel

A Buffer

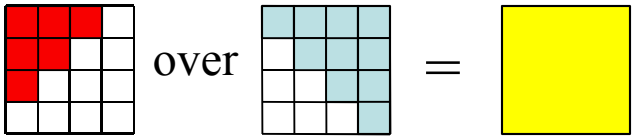
- Extra information for each pixel may include:
  - RGB intensity components
  - Opacity
  - Depth
  - Area coverage percent
  - Surface id
  - Normals, tangents …

A Buffer

- Algorithm: Drawing pass (do not directly display the result)
  - if polygon is opaque and covers pixel, insert into list, removing all polygons farther away
  - if polygon is transparent or only partially covers pixel, insert into list, but don’t remove farther polygons

A Buffer

- Algorithm: Rendering pass
  - At each pixel, traverse buffer using polygon colors and coverage masks to composite:
- Advantage:
  - Can do more than Z-buffer
  - Coverage mask idea can be used in other visibility algorithms



- Disadvantages:
  - Not in hardware, and slow in software
  - Still at heart a z-buffer: Over-rendering and depth quantization problems
- But, used in high quality rendering tools

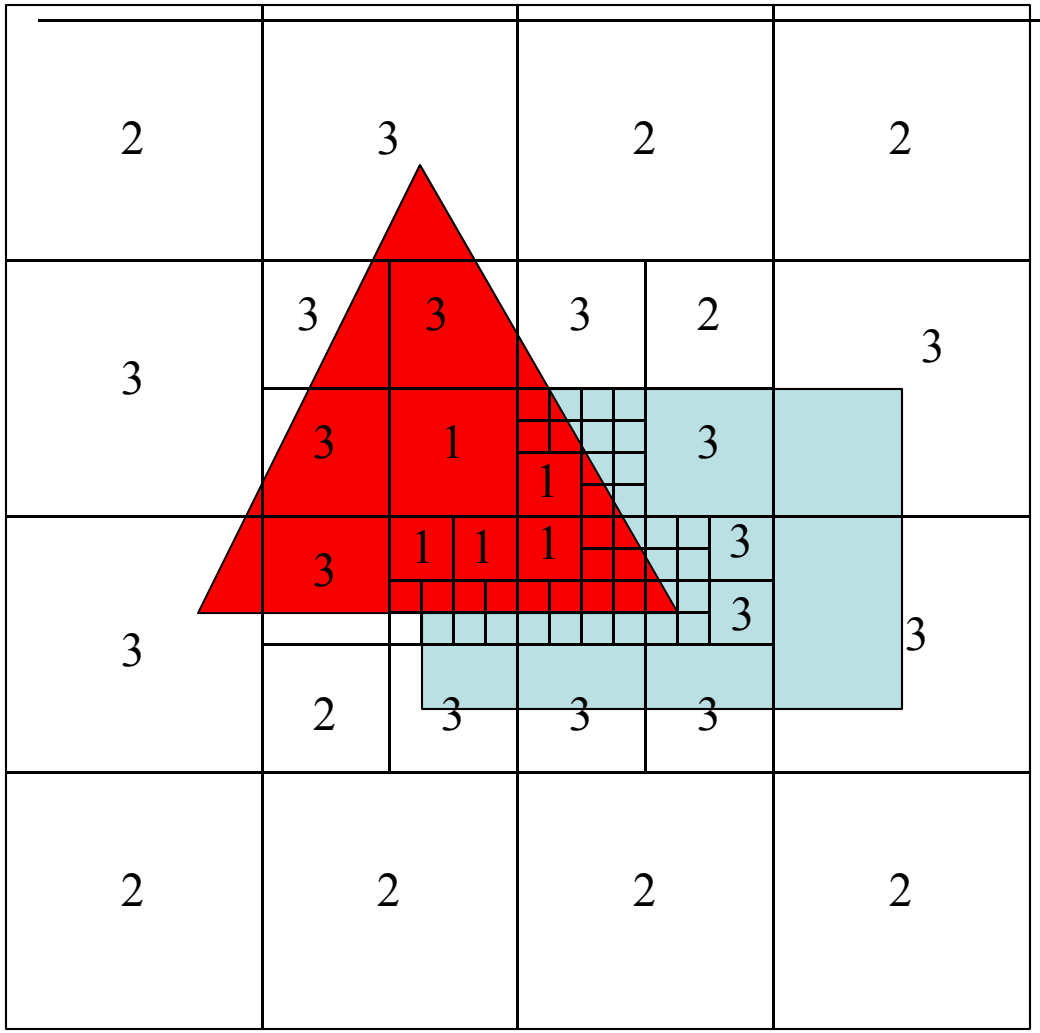
Area Subdivision

- Exploits area coherence: Small areas of an image are likely to be covered by only one polygon
- Three easy cases for determining what's in front in a given region:
  1. a polygon is completely in front of everything else in that region
  2. no surfaces project to the region
  3. only one surface is completely inside the region, overlaps the region, or surrounds the region

Warnock's Area Subdivision

- Start with whole image (image space technique)
- If one of the easy cases is satisfied (previous slide), draw what's in front
- Otherwise, subdivide the region and recurse
- If region is single pixel, choose surface with smallest depth
- Advantages:
  - No over-rendering
  - Anti-aliases well - just recurse deeper to get sub-pixel information
- Disadvantage:
  - Tests are quite complex and slow

Warnock`s Algorithm



- Regions labeled with case used to classify them:
  1) One polygon in front
  2) Empty
  3) One polygon inside, surrounding or intersecting
- Small regions not labeled
- Note it's a rendering algorithm and a HSR algorithm at the same time
  - Assuming you can draw squares

Simple Comparison of Some Algorithms

- Backface culling fast, but insucient by itself
- Painter's algorithm device independent, but details tough, and algorithm is slow
- Z-buffer is online, fast, and easy to implement, but device dependent and memory intensive. Algorithm of choice for hardware

- *Think of a more detailed comparison !*

View Frustum Culling

- All previous techniques works on primitives (triangles in most cases) sent thru the pipeline
- View frustum culling groups primitives using simpler higher level constructs and tries to eliminate primitives in a higher level before sending them thru the pipeline.

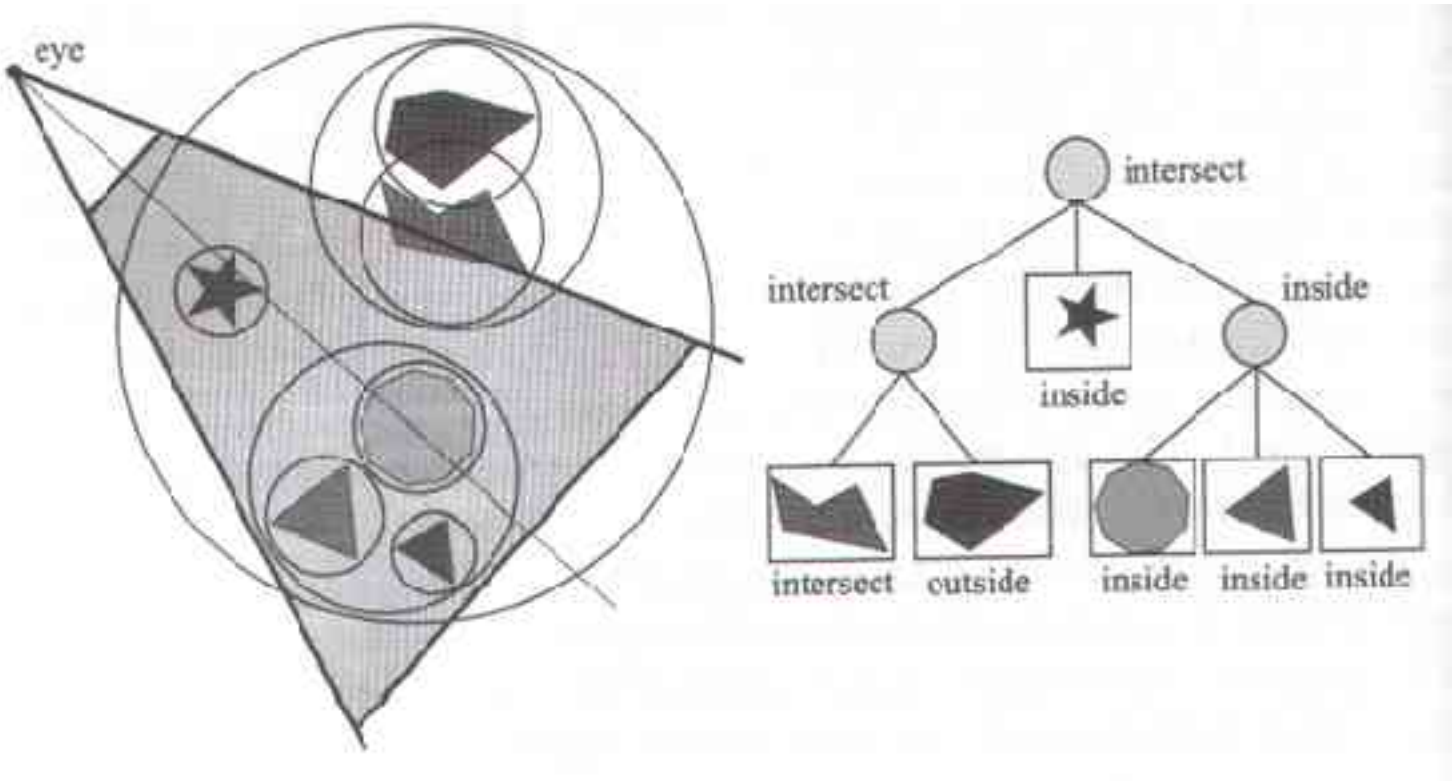
View Frustum Culling

- One way to speed up the rendering process is to compare bounding volume (box, sphere, simplified convex hull) of each object to the view frustum. If the BV is outside the frustum, then the geometry it encloses can be omitted from rendering before sending to the pipeline.
- If instead the BV is inside or intersecting the frustum, then the contents of that BV *may* be visible and must be sent thru the pipeline.

Hierarchical View Frustum Culling

- By using a spatial data structure this kind of culling can be applied hierarchically
- For a bounding volume hierarchy a preorder traversal from the root does the job.
- Each node with a BV is tested against the frustum. If the BV is outside the frustum then the node is not processed further and the tree is pruned since BV`s subtree is outside the view
- If the BV intersects the frustum, then the traversal continues and its children are tested. When a leaf node is found to intersect or inside its geometry is sent thru the pipeline
- If BV is fully inside the frustum its contents must be all inside the frustum. Traversal continues to draw but no further frustum testing is needed for the rest of the subtree

Hierarchical View Frustum Culling



View Frustum Culling

- View frustum culling operates before entering to the pipeline
- For large scenes or certain camera views only a fraction of scene might be visible and needs to be sent thru the rendering pipeline
- View frustum techniques exploit spatial coherence in a scene, since objects located near to each other can be enclosed in a BV, and nearby BVs may be clustered hierarchically
- Other spatial data structures than the bounding volume hierarchy for view frustum culling include octrees and BSPs. They are good for static scenes and can perform better that bounding volume hierarchies

References

These slides are based on:

- *University Of Washington CSE 457* lecture slides:
  - *http://www.cs.washington.edu/education/courses/457/04au/slides/cse457-09-hida*
- *Princeton University COS 426* lecture slides. Follow the link:
  - *http://www.cs.princeton.edu/courses/archive/spring03/cs426/#Coursework/*
- *University of Waterloo CS 488* lecture slides. Follow the link:
  - *http://www.student.cs.uwaterloo.ca/~cs488/*
- *University of Wisconsin CS559* course notes. Follow the link:
  - *http://www.cs.wisc.edu/~cs559-1/*
- *"Real Time Rendering"* book by Tomas Akenine-Moller, Eric Haines
  - *http://www.realtimerendering.com*