Visibility

H&B chapter 9 (similar to notes)

- Of these polygons, which are visible? (in front, etc.)
- Very large number of different algorithms known. Two main (very rough) classes:
  - Object precision: computations that decompose polygons in world coordinates
  - Image precision: computations at the pixel level
- Depth order in standard view box is same as depth order in 3D, so can work with the box.

Visibility

H&B chapter 9 (similar to notes)

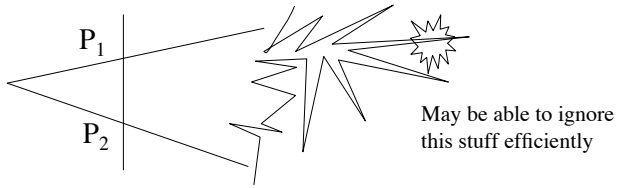
- Essential issues:
  - must be capable of handling complex rendering databases.
  - in many complex worlds, few things are visible
  - efficiency - don't render pixels many times.
  - accuracy - answer should be right, and behave well when the viewpoint moves
  - aliasing

Image Precision

- Typically simpler algorithms (e.g., Z-buffer, ray cast)
- Pseudocode (conceptual!)
  - For each pixel
    - Determine the closest surface which intersects the projector
    - Draw the pixel the appropriate color

Image Precision

- "Image precision" means that we can save time not computing precise intersections of complicated objects



- But the algorithms are subject to aliasing problems, and the sampling needs to be redone when the view changes, even if only a simple window resize

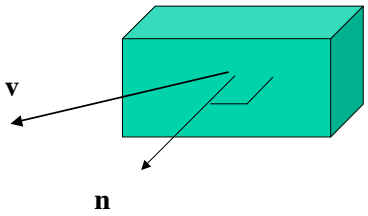
Object Precision

- The algorithms are typically more complex
- Pseudocode (conceptual)
  - For each object
    - Determine which parts are viewed without obstruction by other parts of itself of other objects
    - Draw those parts the appropriate color

Visibility - Back Face Culling

- Simple, preliminary step, to reduce the amount of work.
- Polygons from solid objects have a front face and back face
- If the viewer sees the back face, then the plane can be culled.
- Why would the viewer see the back face?
  - Because they are on the back side of the plane of the polygon.

Visibility - Back Face Culling



v is direction from any point on the plane to the center of projection (the eye).

If $\mathbf{n}.\mathbf{v} > 0$, then display the plane

Note that we are calculating which side of the plane the eye is on.

Question: How do we get **n**? (e.g., for the assignment)

Visibility - Back Face Culling

Question: How do we get **n**? (e.g., for the assignment)

Answer

When you render the parallelepiped, you have to create the faces which are sequences of vertices.

To compute **n** from vertices, use cross product.

You need to store vertices consistently so that you can get the sign of n. Consider storing them so that you can get the sign of **n** by RHR.

Depending on the situation you may find it easier to

1) compute **n** early on, and transform it using the correct formula
2) recompute it from transformed vertices.

Visibility - Back Face Culling

Question: In which coordinate frames can/should we do this?

Visibility - Back Face Culling

Question: In which coordinate frames can/should we do this?

Answer

All of them. It is perhaps more natural to attempt do this in the standardized view box where perspective projection has become parallel projection.

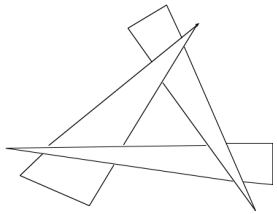
Here, **e**=(0,01) (why?), so the test **n.e**>0 is especially easy (n_z>0).

But be careful with the transformation which lead to **n**!

Also, an efficiency argument can be made for culling before division.

Visibility - painters algorithm

- Algorithm
  - Choose an order for the polygons based on some choice (e.g. depth to a point on the polygon)
  - Render the polygons in that order, deepest one first
- This renders nearer polygons over further.
- Works for some important geometries (2.5D - e.g. VLSI, mazes--but more efficient algorithms exist)
- Doesn't work in this form for most geometries (see figure)



The Z - buffer

- For each pixel on screen, have a second memory location - called the z-buffer
- Set this buffer to a value corresponding to the furthest point
- As a polygon is filled in, compute the depth value of each pixel
  - if depth < z buffer depth, fill in pixel and new depth
  - else disregard
- Typical implementation: Compute Z while scan-converting. A ∂Z for every ∂X is easy to work out.

The Z - buffer

- Advantages:
  - simple; hardware implementation common
  - efficient z computations are easy.
  - ok with lots of surfaces (if there are lots, they tend to be small, and not much difference to this algorithm)
- Disadvantages:
  - over renders - can be slow for very large collections of polygons - may end up scan converting many hidden objects
  - quantization errors can be annoying (not enough bits in the buffer)
  - doesn't help with transparency, or filtering for anti-aliasing.

The A - buffer

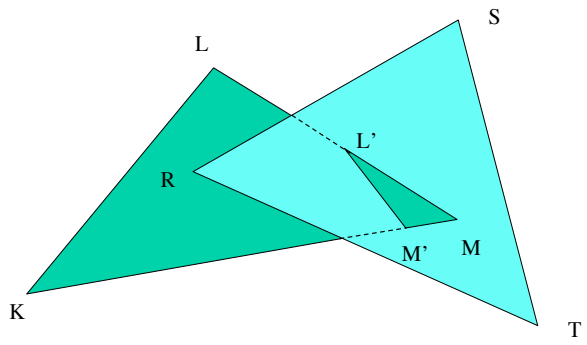
- For transparent surfaces and filter anti-aliasing:
- Algorithm: filling buffer
  - at each pixel, maintain a pointer to a list of polygons sorted by depth.
  - when filling a pixel:
    - if polygon is opaque and covers pixel, insert into list, removing all polygons farther away
    - if polygon is opaque and only partially covers pixel, insert into list, but don't remove farther polygons
- Algorithm: rendering pixels
  - at each pixel, traverse buffer using brightness values in polygons to fill.
  - values are used either in transparency or for filtering for aliasing

Scan line algorithm

- Assume polygons do not intersect one another.
- Observation: on any given scan line, the visible polygon can change only at an edge.
- Algorithm:
  - fill all polygons simultaneously at each scan line, have all edges that cross scan line in AEL
  - keep record of current depth at current pixel - use to decide which is in front in filling span when an edge is encountered

Scan line algorithm

- To deal with penetrating polygons, split them up



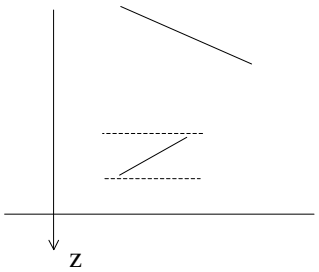
Scan line algorithm

- Advantages:
  - potentially fewer quantization errors (typically more bits available for depth, but this depends)
  - filter anti-aliasing can be made to work.
- Disadvantages:
  - invisible polygons clog AEL, ET (can get expensive for complex scenes).

Depth sorting

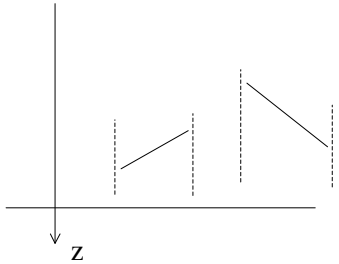
- Sort in order of decreasing depth
  - use closest point
- Render in sorted order
- For surface S with greatest depth
  - if no depth overlaps (z extents intersect) with other surfaces, then render (like painter's algorithm), and remove surface from list
  - if a depth overlap is found, test for problem overlap in image plane
  - if S, S' overlap in depth and in image plane, swap and try again
  - if S, S' have been swapped already, split one across plane of other (like clipping) and reinsert
- Testing image plane problem overlaps (test get increasingly expensive):
  - xy bounding boxes do not intersect
  - *or* S is behind the plane of S'
  - *or* S' is in front of the plane of S
  - *or* S and S' do not intersect
- Advantages:
  - filter anti-aliasing works fine
  - no depth quantization error
  - works well if not too much depth overlap (rarely get to expensive cases)
- Disadvantages:
  - gets expensive with lots of depth overlap (over-renders)

Depth sorting (2D illustrations)



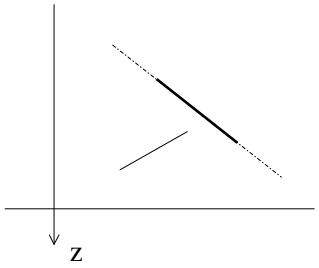
No depth overlap between furthest object and rest of list--paint it.

Depth sorting (2D illustrations)



Overlap in depth, but lack of overlap in image plane can be resolved by bounding boxes. \

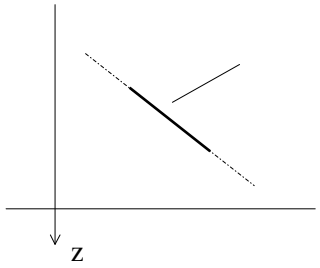
Depth sorting (2D illustrations)



It is safe to paint the furthest, but figuring this out requires observing that the near one is all on the same side of the plane (line in the 2D figure) of the furthest.

I.e, the near polygon is in front of the plane of the far polygon.

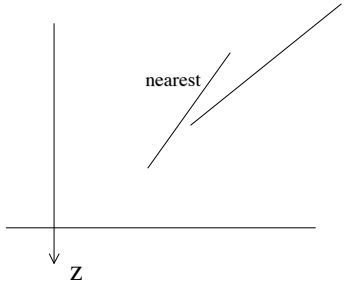
Depth sorting (2D illustrations)



It is safe to paint the furthest, but figuring this out requires observing that it is all on the other side of the plane (line in the 2D figure) of the nearest plane.

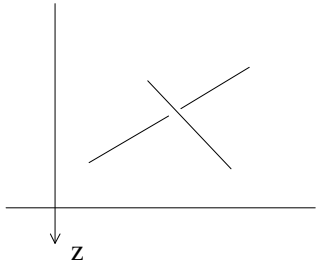
I.e, the far polygon is behind the plane of the near polygon.

Depth sorting (2D illustrations)



The furthest (as defined by the closest point) obscurs the "nearest". It is safe to paint the "nearest", but figuring this out requires reversing the nearest and furthest, and then reapplying one of the previous tests.

Depth sorting (2D illustrations)



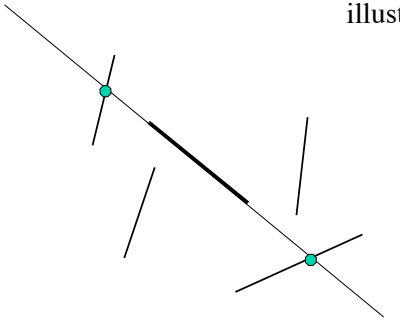
If the preceding tests fail, we have to split the far polygon (line in the drawing) with the plane of the near polygon (**basically a clip operation**), and put the pieces into the list, and carry on.

BSP - trees

- Construct a tree that gives a rendering order
- Tree recursively splits 3D world into cells, each of which contain at most one piece of polygon.
- Constructing tree:
  - Choose polygon (arbitrary)
  - split its cell using plane on which polygon lies
  - continue until each cell contains only one polygon

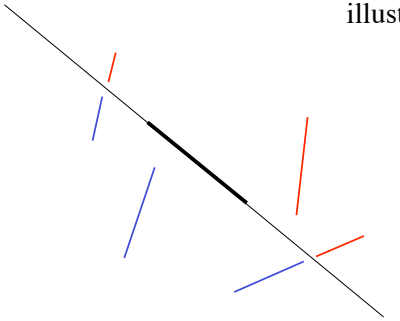
BSP - trees

2D version for illustration



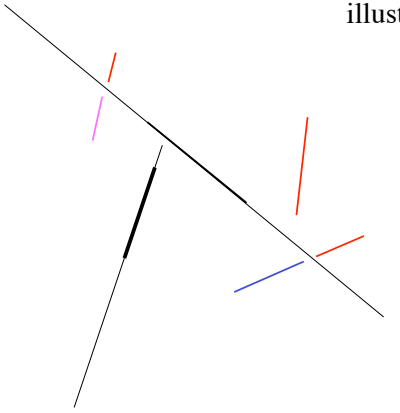
BSP - trees

2D version for illustration



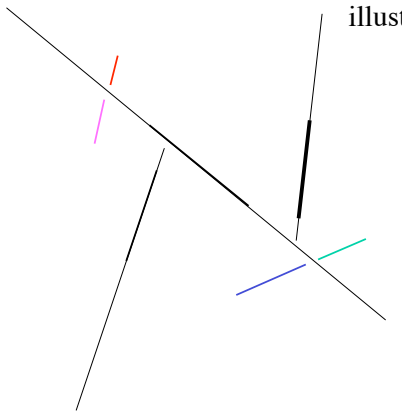
BSP - trees

2D version for illustration



BSP - trees

2D version for illustration

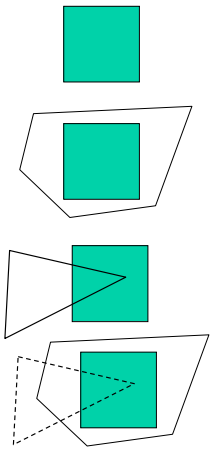


BSP - trees

- Rendering tree:
  - recursive descent
  - render back, node polygon, front
- Disadvantages:
  - many small pieces of polygon (more splits than depth sort!)
  - over rendering (does not work well for complex scenes with lots of depth overlap)
  - hard to get balanced tree
- Advantages:
  - one tree works for all focal points (good for cases when scene is static)
  - filter anti-aliasing works fine, as does transparency
  - data structure is worth knowing about

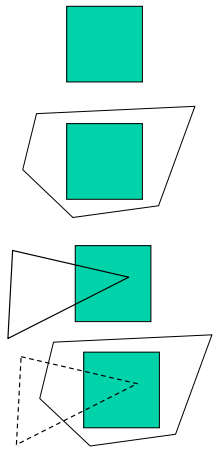
Area subdivision

- Four tractable cases for a given region in image plane:
  - no surfaces project to the region
  - only one surface completely surrounds the region
  - only one surface is completely inside the region or overlaps the region
  - a polygon is completely in front of everything else in that region determined by considering depths of the polygons at the corners of the region



Area subdivision

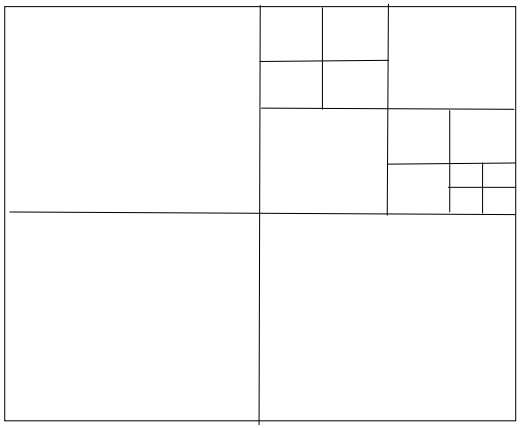
- Four tractable cases for a given region in image plane:
  - no surfaces project to the region (paint background if needed, otherwise do nothing)
  - only one surface completely surrounds the region (paint surface)
  - only one surface is completely inside the region or overlaps the region (paint background if applicable and then scan convert region)
  - a polygon is completely in front of everything else in that region determined by considering depths of the polygons at the corners of the region (paint surface)



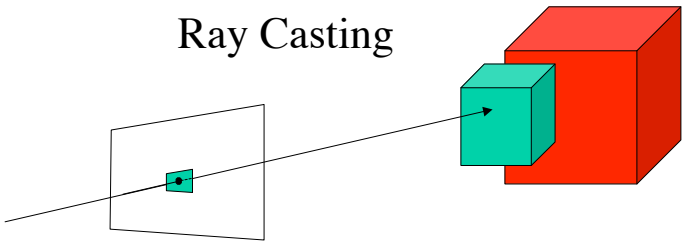
Area subdivision

- Algorithm:
  - subdivide each region until one of these cases is true or until region is very small
  - if case is true, deal with it
  - if region is small, choose surface with smallest depth.
  - determining cases quickly makes use of the same ideas in depth sort (depth sorting, bounding boxes, tests for which side of the plane), and the difficult cases are deferred by further subdivision.
- Advantages:
  - can be very efficient
  - no over rendering
  - anti-aliases well (subdivide a bit further)

One Subdivision Strategy



Ray Casting



- Image precision algorithm
- For each pixel cast a ray into the world
  - For each surface
    - determine intersection point with ray
  - Render pixel based on closest surface

Ray Casting

- First step in ray tracing algorithm
- Expensive
- Good performance usually requires clever data structures such as bounding volumes for object groups or storing world occupancy information in octrees.
- Other main problem is computing intersection.
- For polygons, we can use the standardized orthographic space where we can work in 2D.
- Spheres are easy (a bit more difficult in perspective space).
- Useful for “picking”--not expensive here (why?)

Ray Tracing--teaser

- Idea is very simple--follow light around
- Following **all** the light around is intractable, so we follow the light that makes **the most** difference
- Work backwards from what is seen
- Simple ray tracer
  - Cast a ray through each pixel (as in ray casting for visibility)
  - From intersection point cast additional rays to determine the color of the pixel.
    - For diffuse component, must cast rays to the lights
    - We may also add in some “ambient” light
    - For mirrors, must cast ray in mirror direction (recursion--what is the stopping condition)