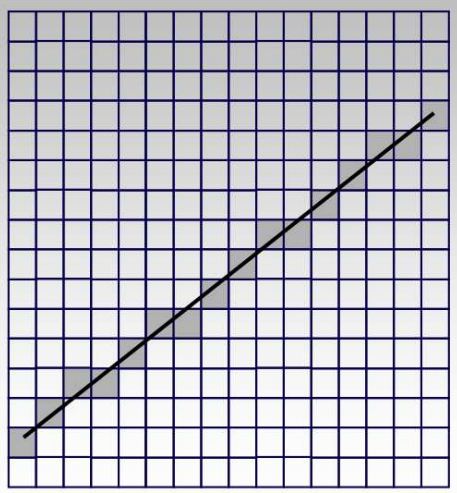
Hidden Surface Removal/ Visibility

CPSC 314

Scan Conversion - Lines



Scan Conversion of Lines - Digital Differential Analyzer

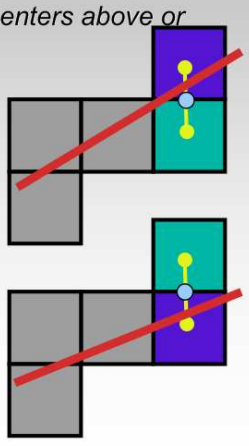
First Attempt:

```
dda( float xs, ys, xe, ye ) {
    // assume xs < xe, and slope m between 0 and 1
    float m= (ye-ys)/(xe-xs);
    float y= round( ys );
    for( int x= round( xs ) ; x<= xe ; x++ ) {
        drawPixel( x, round( y ) );
        y= y+m;
    }
}
```


Scan Conversion of Lines Midpoint Algorithm

Moving horizontally along x direction

- Draw at current y value, or move up vertically to y+1?
  - *Check if midpoint between two possible pixel centers above or below line*



Candidates

- Top pixel: (x+1,y+1)
- Bottom pixel: (x+1, y)

Midpoint: (x+1, y+.5)

Check if midpoint above or below line

- Below: top pixel
- Above: bottom pixel

Key idea behind Bresenham Alg.

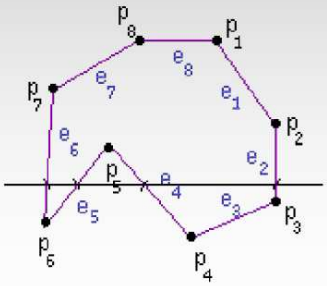
Scan Conversion of Lines

Bresenham Algorithm

```
Bresenham( int xs, ys, xe, ye ) {
    int y= ys;
    incrE= 2(ye - ys);
    incrNE= 2((ye - ys) – (xe-xs));
    for( int x= xs ; x<= xe ; x++ ) {
        drawPixel( x, y );
        if( d<= 0 ) d+= incrE;
        else { d+= incrNE; y++; }
    }
}
```

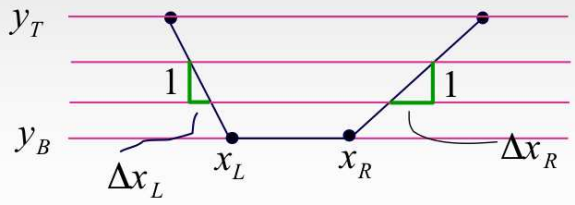

Scan Conversion of Polygons

- Works for arbitrary polygons
- Efficiency improvement:
  - *Exploit row-to-row coherence using "edge table"*



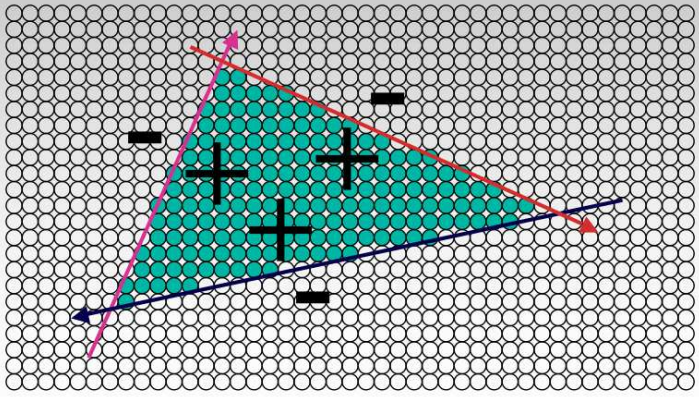
Edge Walking

```
for (y=yB; y<=yT; y++) {
    for (x=xL; x<=xR; x++)
     setPixel(x,y);
    xL += DxL;
    xR += DxR;
}
```



Modern Rasterization: Edge Equations

Define a triangle as follows:



Computing Edge Equations

Summary:

- Now we have only ONE equation

$$L(x,y) = -(y_e - y_s)(x - x_s) + (y - y_s)(x_e - x_s)$$

- Works for both cases
- Also works for vertical lines!

Plane Equation: Interpolating Vertex Attributes

Observation: Quantities vary linearly across image plane

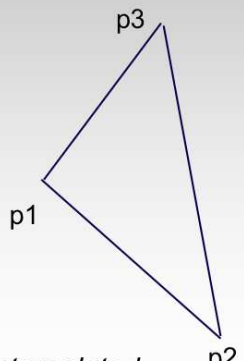
- E.g.: r = Ax + By + C
  - *r= red channel of the color*
  - *Same for g, b, Nx, Ny, Nz, z…*
- From info at vertices we know:

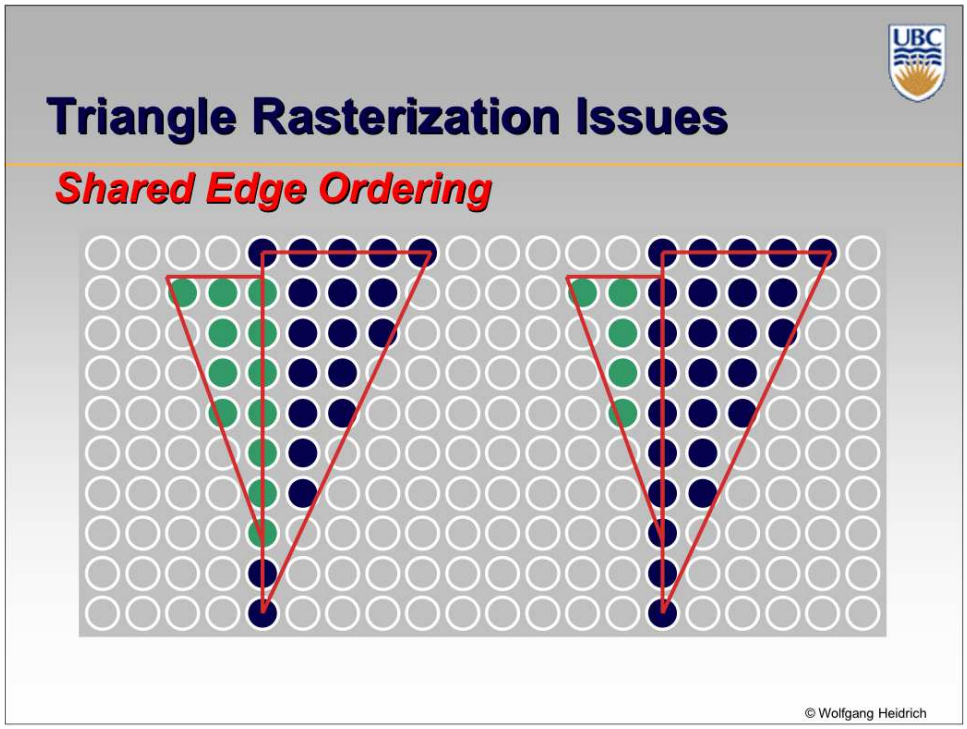
$$r_1 = Ax_1 + By_1 + C$$

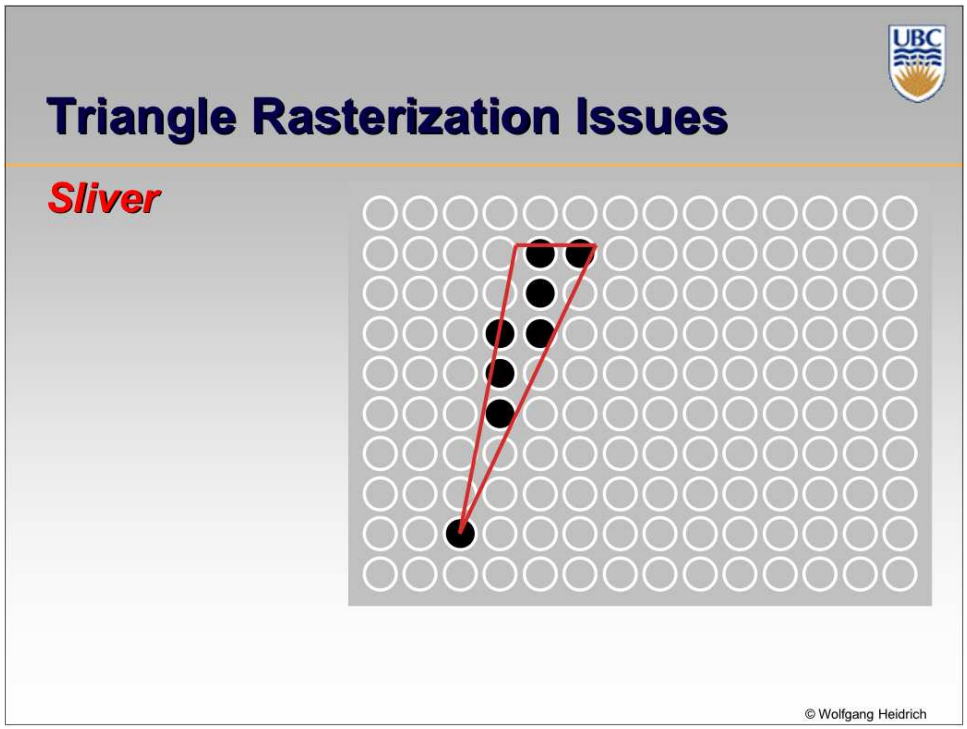
$$r_2 = Ax_2 + By_2 + C$$

$$r_3 = Ax_3 + By_3 + C$$

  - *Solve for A, B, C*
  - *One-time set-up cost per triangle and interpolated quantity*

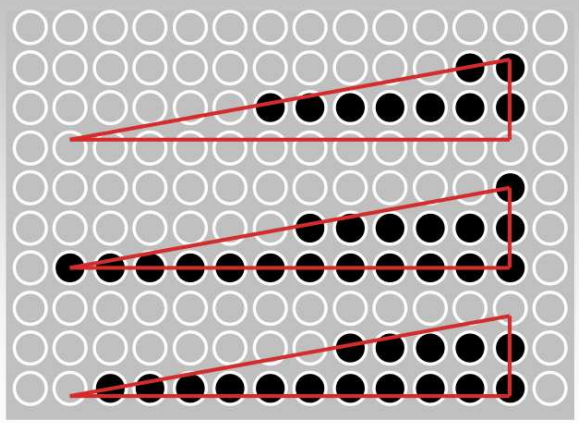



Triangle Rasterization Issues
Shared Edge Ordering
© Wolfgang Heidrich

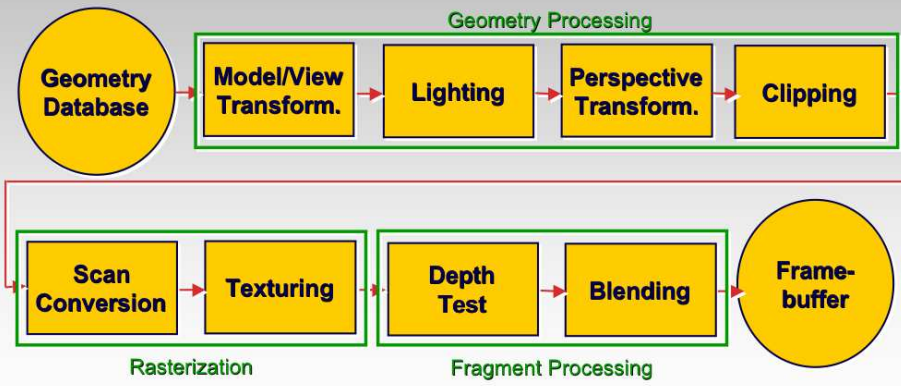

Triangle Rasterization Issues
Sliver
© Wolfgang Heidrich

Triangle Rasterization Issues

Moving Slivers

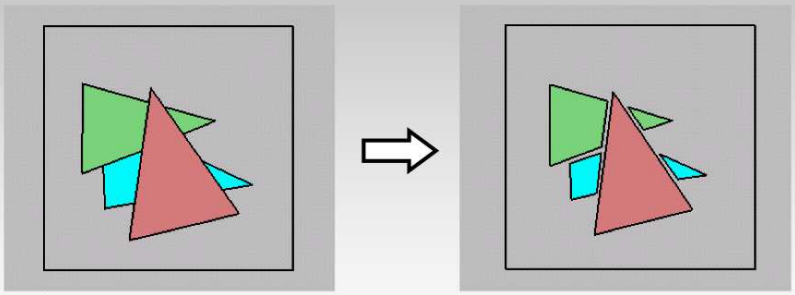


The Rendering Pipeline



Occlusion

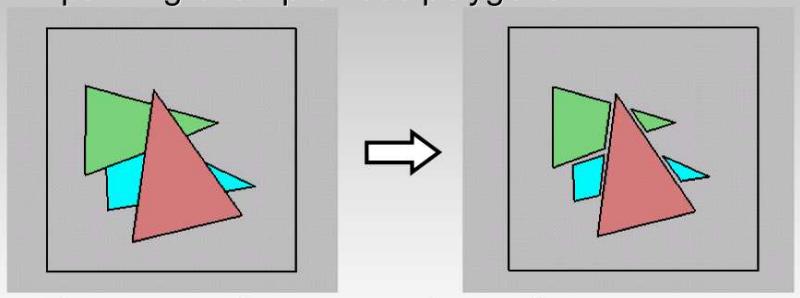
- For most interesting scenes, some polygons overlap



- To render the correct image, we need to determine which polygons occlude which

Painter's Algorithm

- Simple: render the polygons from back to front, "painting over" previous polygons



- Draw cyan, then green, then red

will this work in the general case?

Painter's Algorithm: Problems

- ***Intersecting polygons*** present a problem
- Even non-intersecting polygons can form a cycle with no valid visibility order:

Hidden Surface Removal

Object Space Methods:

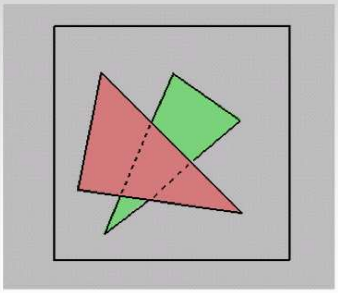
- Work in 3D before scan conversion
  - *E.g. Painter's algorithm*
- Usually independent of resolution
  - *Important to maintain independence of output device (screen/printer etc.)*

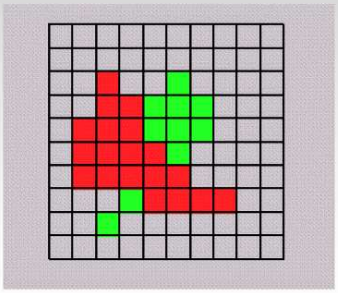
Image Space Methods:

- Work on per-pixel/per fragment basis after scan conversion
- Z-Buffer/Depth Buffer
- Much faster, but resolution dependent

The Z-Buffer Algorithm

- What happens if multiple primitives occupy the same pixel on the screen?
- Which is allowed to paint the pixel?





The Z-Buffer Algorithm

Idea: retain depth after projection transform

- Each vertex maintains z coordinate
  - *Relative to eye point*
- Can do this with canonical viewing volumes

The Z-Buffer Algorithm

Augment color framebuffer with Z-buffer

- Also called depth buffer
- Stores z value at each pixel
- At frame beginning, initialize all pixel depths to ∞
- When scan converting: interpolate depth (z) across polygon
- Check z-buffer before storing pixel color in framebuffer and storing depth in z-buffer
- don't write pixel if its z value is more distant than the z value already stored there

Z-Buffer

Store (r,g,b,z) for each pixel

- typically 8+8+8+24 bits, can be more

```
for all i,j {
 Depth[i,j] = MAX_DEPTH
 Image[i,j] = BACKGROUND_COLOUR
}
for all polygons P {
  for all pixels in P {
    if (Z_pixel < Depth[i,j]) {
      Image[i,j] = C_pixel
      Depth[i,j] = Z_pixel
    }
  }
}
```

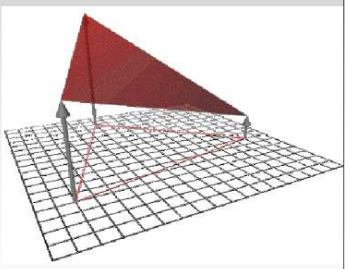
Interpolating Z

Edge walking

- Just interpolate Z along edges and across spans

Barycentric coordinates

- Interpolate z like other parameters
- E.g. color



The Z-Buffer Algorithm (mid-70's)

History:

- Object space algorithms were proposed when memory was expensive
- First 512x512 framebuffer was >$50,000!

Radical new approach at the time

- The big idea:
  - *Resolve visibility **independently at each pixel***

Depth Test Precision

- Reminder: projective transformation maps eye-space z to generic z-range (NDC)
- Simple example:

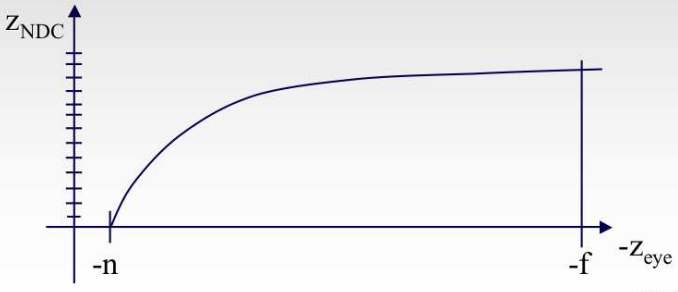
$$T\left(\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}\right) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Thus:

$$z_{NDC} = \frac{a \cdot z_{eye} + b}{z_{eye}} = a + \frac{b}{z_{eye}}$$

Depth Test Precision

- Therefore, depth-buffer essentially stores 1/z, rather than z!
- Issue with integer depth buffers
  - *High precision for near objects*
  - *Low precision for far objects*



Depth Test Precision

- Low precision can lead to depth fighting for far objects
  - *Two different depths in eye space get mapped to same depth in framebuffer*
  - *Which object "wins" depends on drawing order and scan-conversion*
- Gets worse for larger ratios $f{:}n$
  - Rule of thumb: $f{:}n < 1000$ for 24 bit depth buffer
- With 16 bits cannot discern millimeter differences in objects at 1 km distance

Z-Buffer Algorithm Questions

- How much memory does the Z-buffer use?
- Does the image rendered depend on the drawing order?
- Does the time to render the image depend on the drawing order?
- How does Z-buffer load scale with visible polygons? with framebuffer resolution?

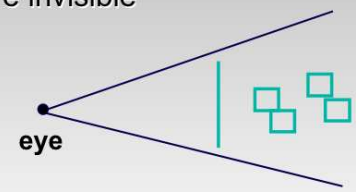
Z-Buffer Pros

- Simple!!!
- Easy to implement in hardware
  - *Hardware support in all graphics cards today*
- Polygons can be processed in arbitrary order
- Easily handles polygon interpenetration

Z-Buffer Cons

Poor for scenes with high depth complexity

- Need to render all polygons, even if most are invisible



Shared edges are handled inconsistently

- *Ordering dependent*

Z-Buffer Cons

Requires lots of memory

- (e.g. 1280x1024x32 bits)

Requires fast memory

- Read-Modify-Write in inner loop

Hard to simulate transparent polygons

- We throw away color of polygons behind closest one
- Works if polygons ordered back-to-front
  - *Extra work throws away much of the speed advantage*

Object Space Algorithms

Determine visibility on object or polygon level

- Using camera coordinates

Resolution independent

- Explicitly compute visible portions of polygons

Early in pipeline

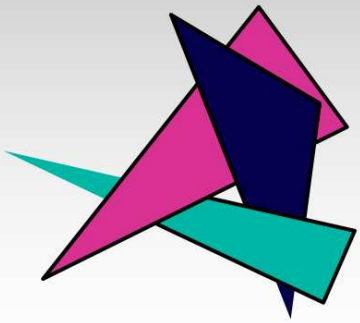
- After clipping

Requires depth-sorting

- Painter's algorithm
- BSP trees

Object Space Visibility Algorithms

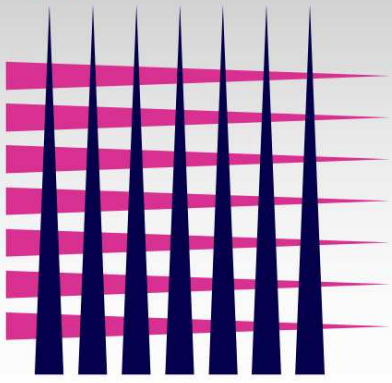
- Early visibility algorithms computed the set of visible ***polygon fragments*** directly, then rendered the fragments to a display:



Object Space Visibility Algorithms

What is the minimum worst-case cost of computing the fragments for a scene composed of *n* polygons?

Answer:
O(n^2)



Object Space Visibility Algorithms

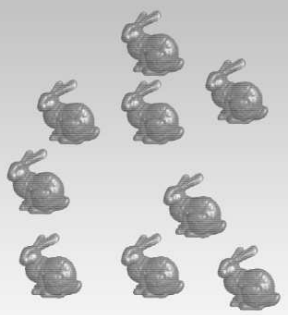
- So, for about a decade (late 60s to late 70s) there was intense interest in finding efficient algorithms for hidden surface removal
- We'll talk about one:
  - **Binary Space Partition (BSP) Trees**
  - *Still in use today for ray-tracing, and in combination with z-buffer*

Binary Space Partition Trees (1979)

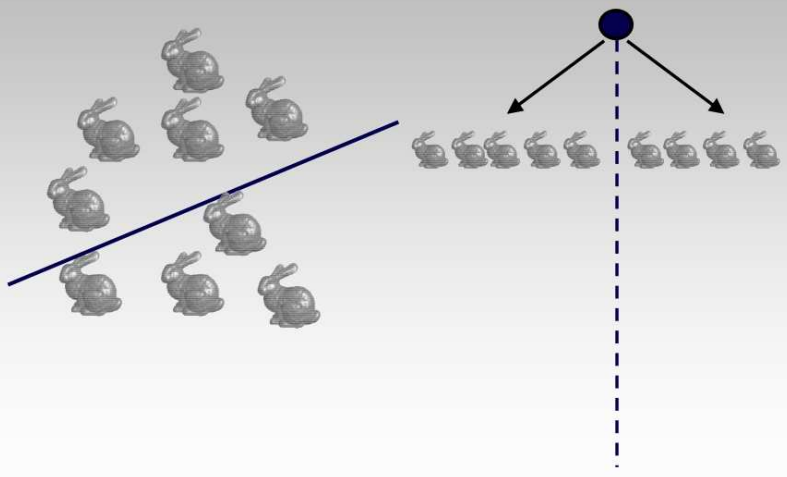
BSP Tree: partition space with binary tree of planes

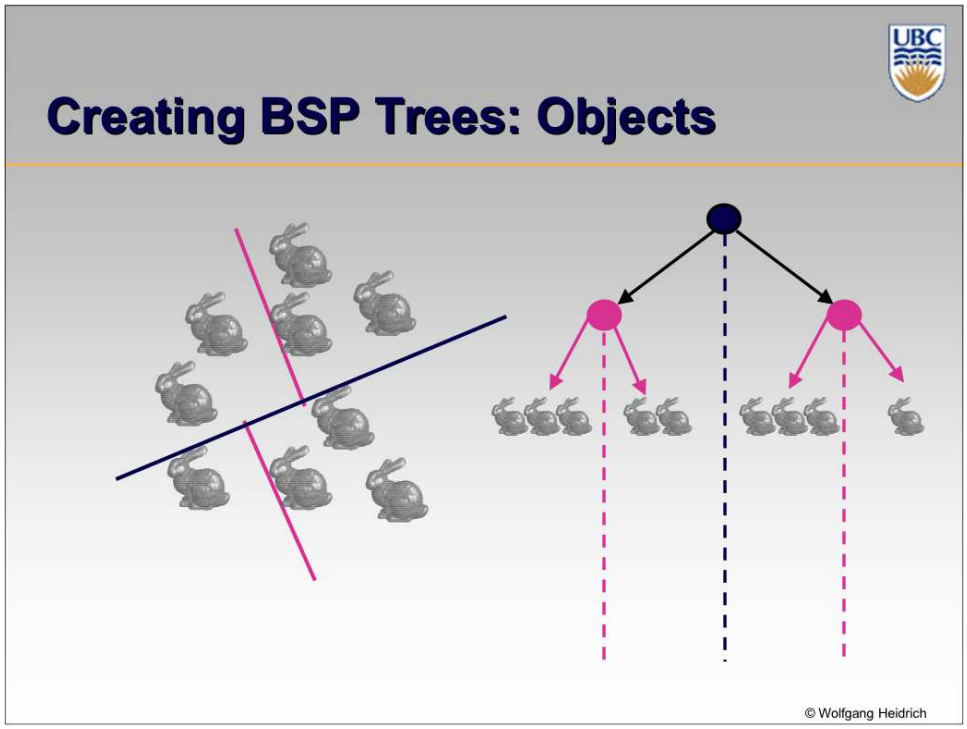
- Idea: divide space recursively into half-spaces by choosing splitting planes that separate objects in scene
- Preprocessing: create binary tree of planes
- Runtime: correctly traversing this tree enumerates objects from back to front

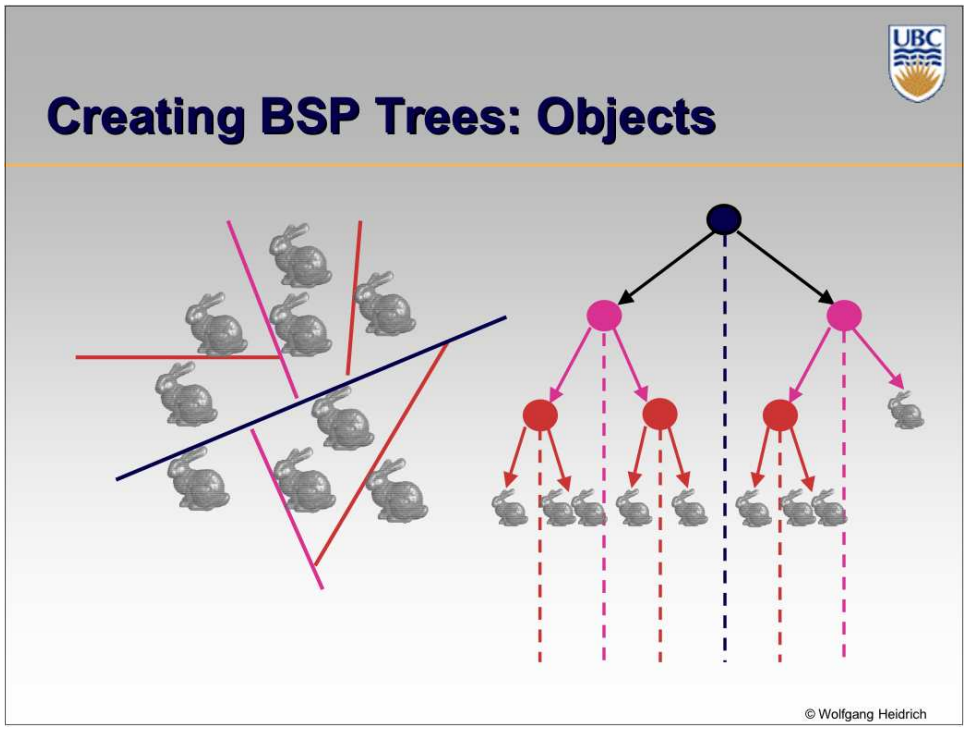
Creating BSP Trees: Objects



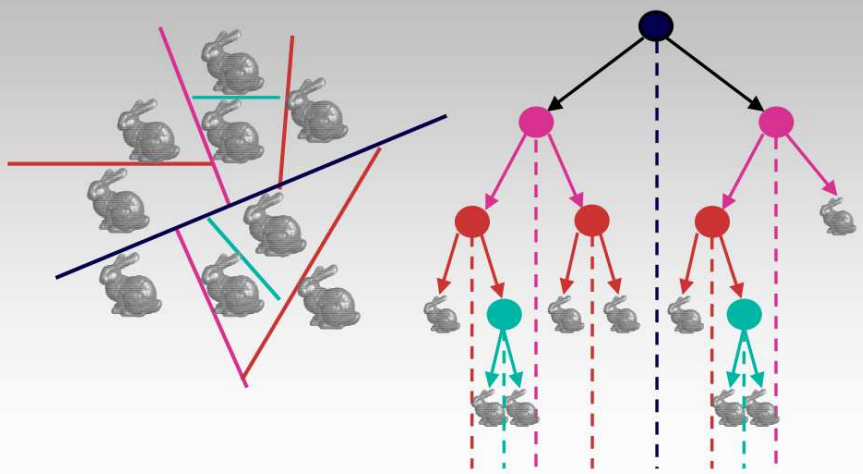
Creating BSP Trees: Objects




Creating BSP Trees: Objects
© Wolfgang Heidrich


Creating BSP Trees: Objects
© Wolfgang Heidrich

Creating BSP Trees: Objects

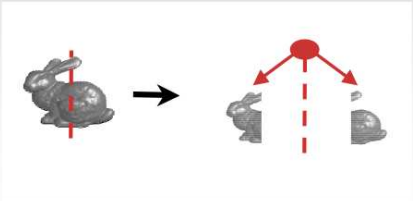


Splitting Objects

No bunnies were harmed in previous example

But what if a splitting plane passes through an object?

- Split the object; give half to each node



Traversing BSP Trees

Tree creation independent of viewpoint

- Preprocessing step

Tree traversal uses viewpoint

- Runtime, happens for many different viewpoints

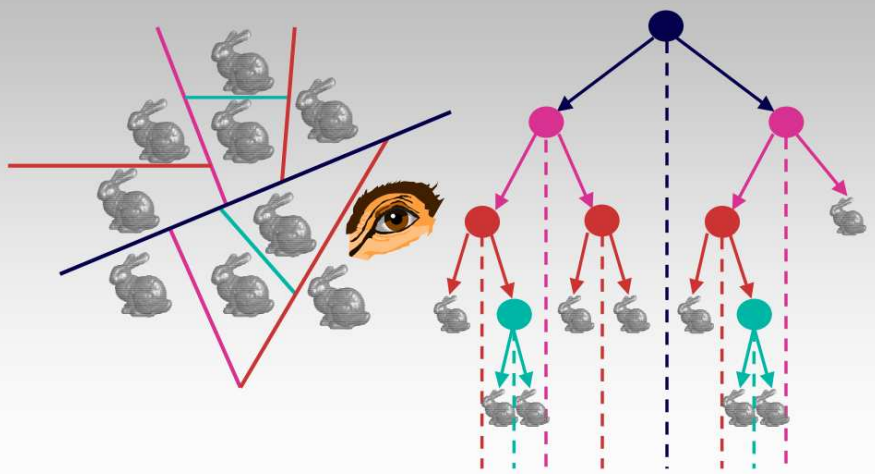
Each plane divides world into near and far

- For given viewpoint, decide which side is near and which is far
  - *Check which side of plane viewpoint is on independently for each tree vertex*
  - *Tree traversal differs depending on viewpoint!*
- Recursive algorithm
  - *Recurse on far side*
  - *Draw object*
  - *Recurse on near side*

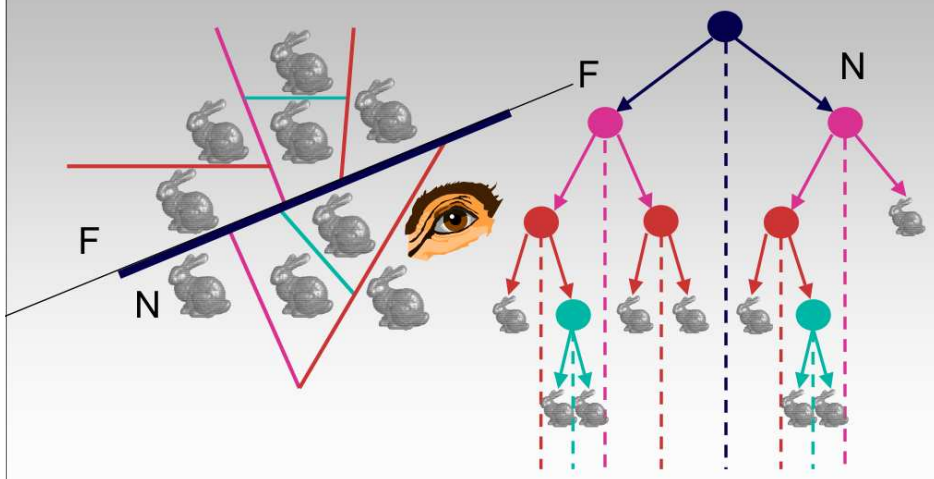
Traversing BSP Trees

```
renderBSP(BSPtree *T)
     BSPtree *near, *far;
     if (eye on left side of T->plane)
      near = T->left; far = T->right;
     else
      near = T->right; far = T->left;
     renderBSP(far);
     if (T is a leaf node)
      renderObject(T)
   renderBSP(near);
```

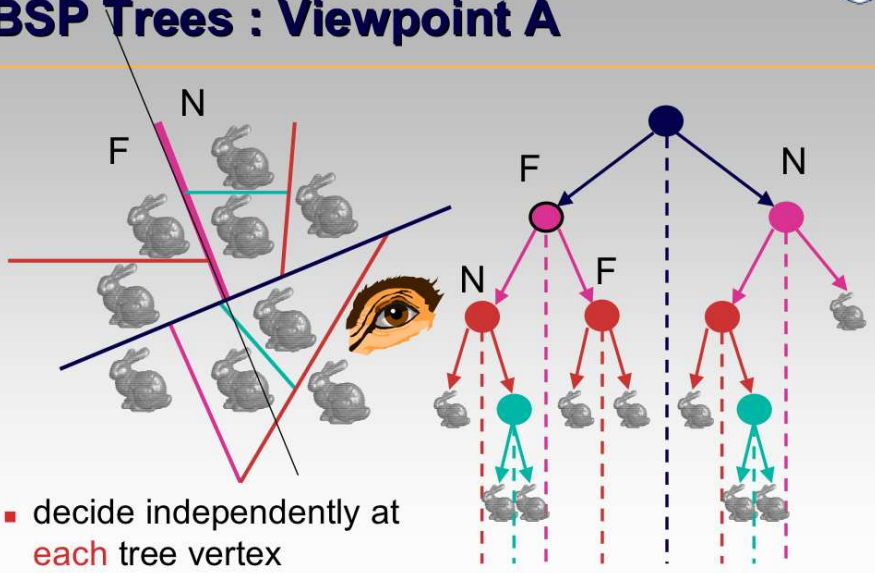
BSP Trees : Viewpoint A



BSP Trees : Viewpoint A

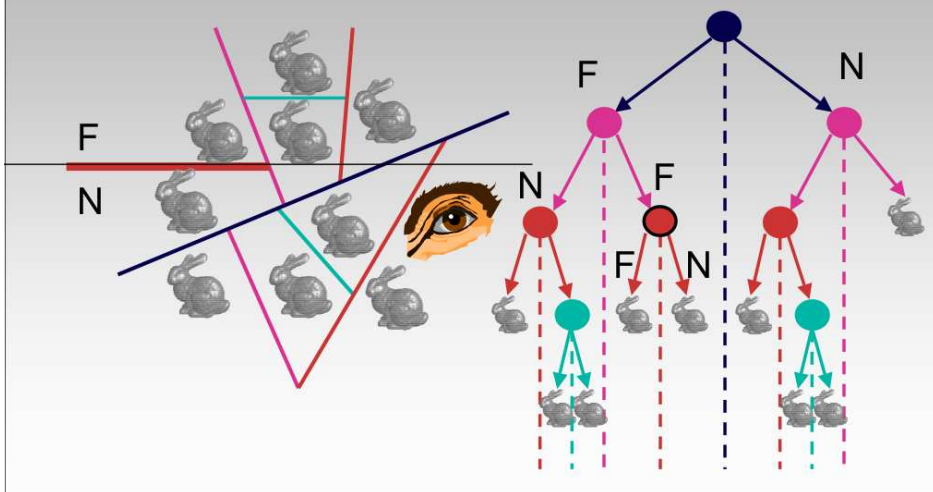


BSP Trees : Viewpoint A

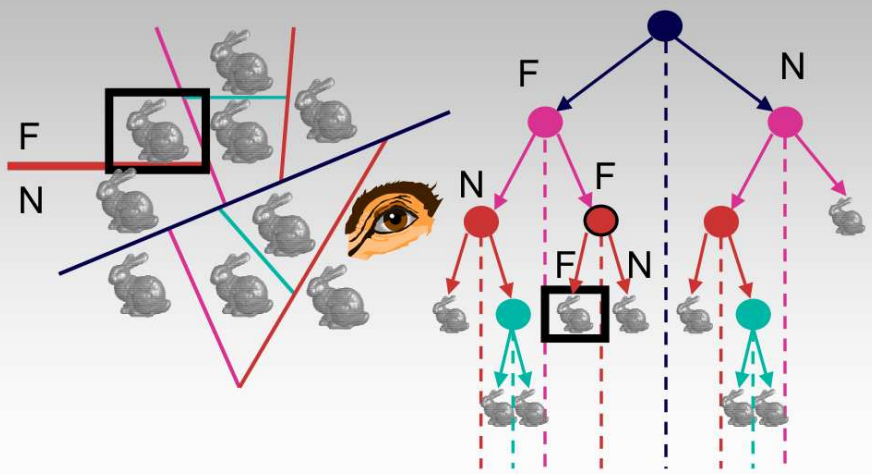


- decide independently at each tree vertex
  - not just left or right child!

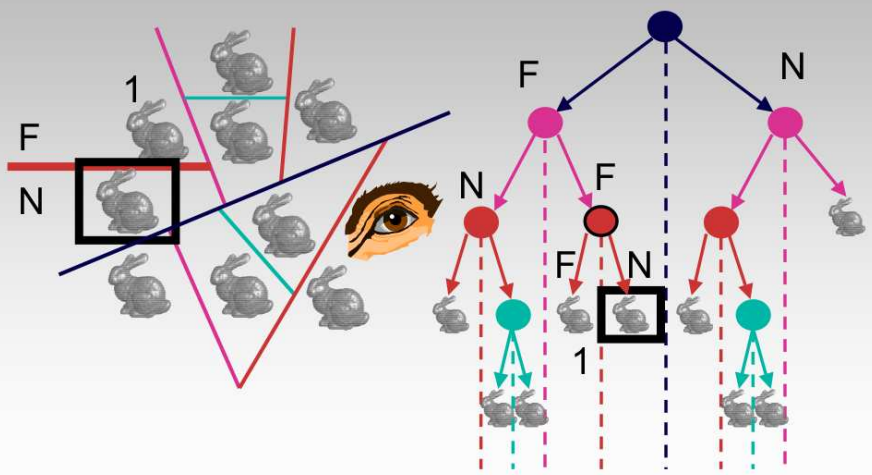
BSP Trees : Viewpoint A

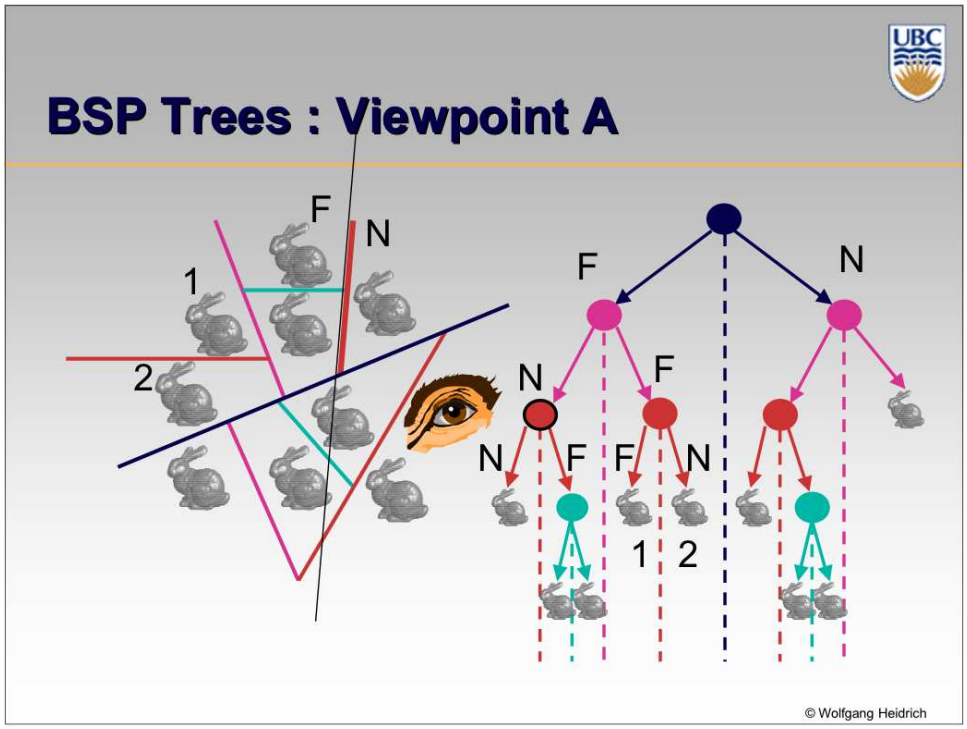


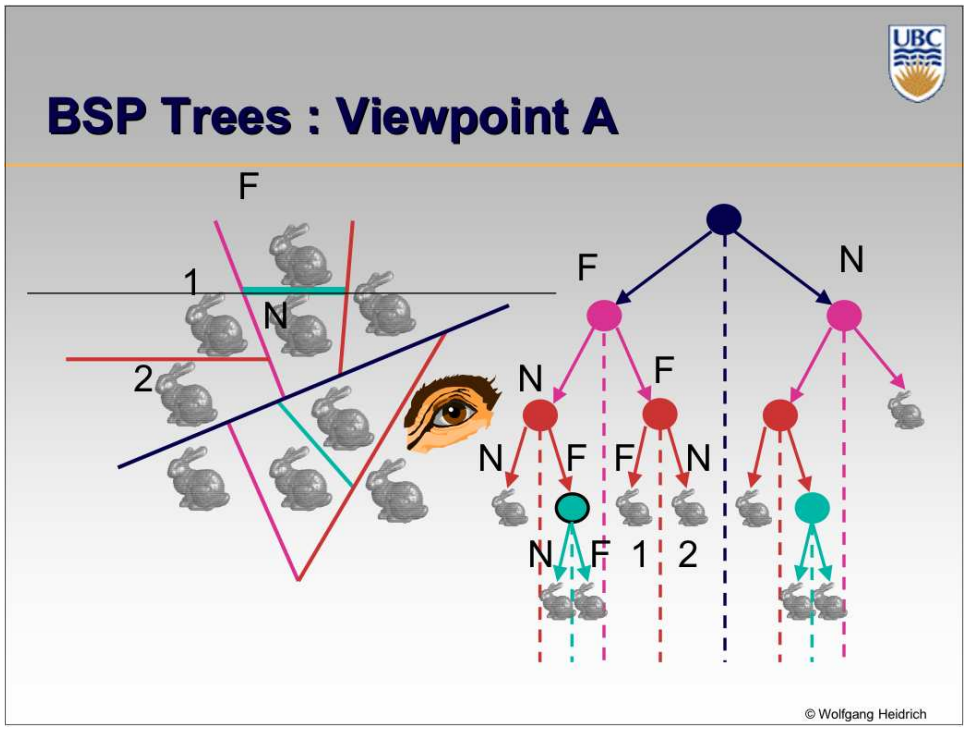
BSP Trees : Viewpoint A

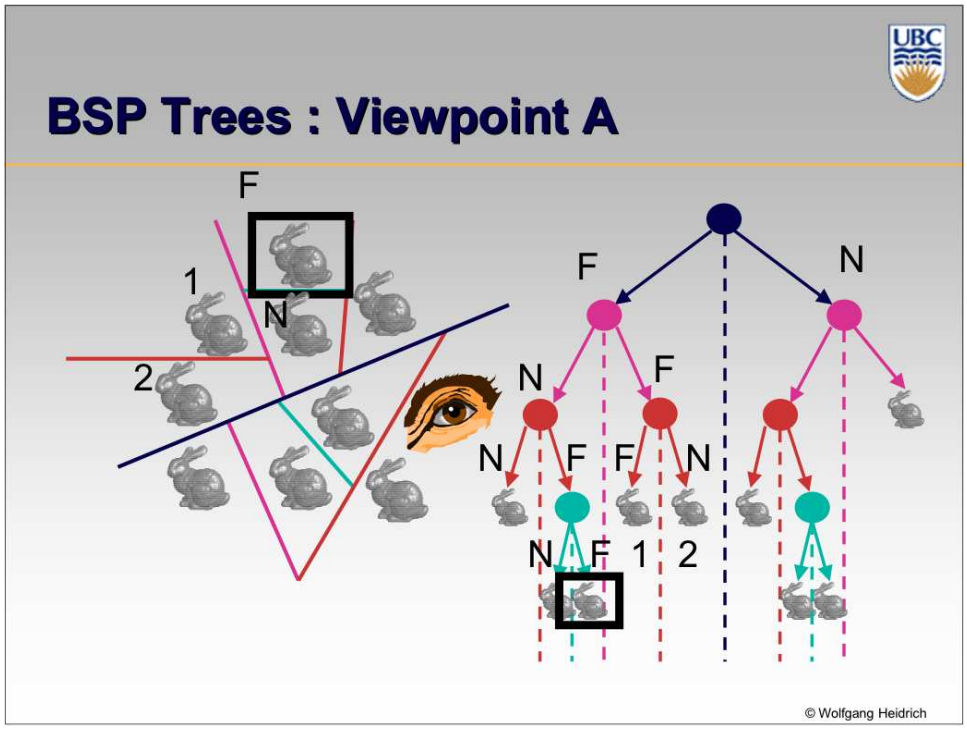


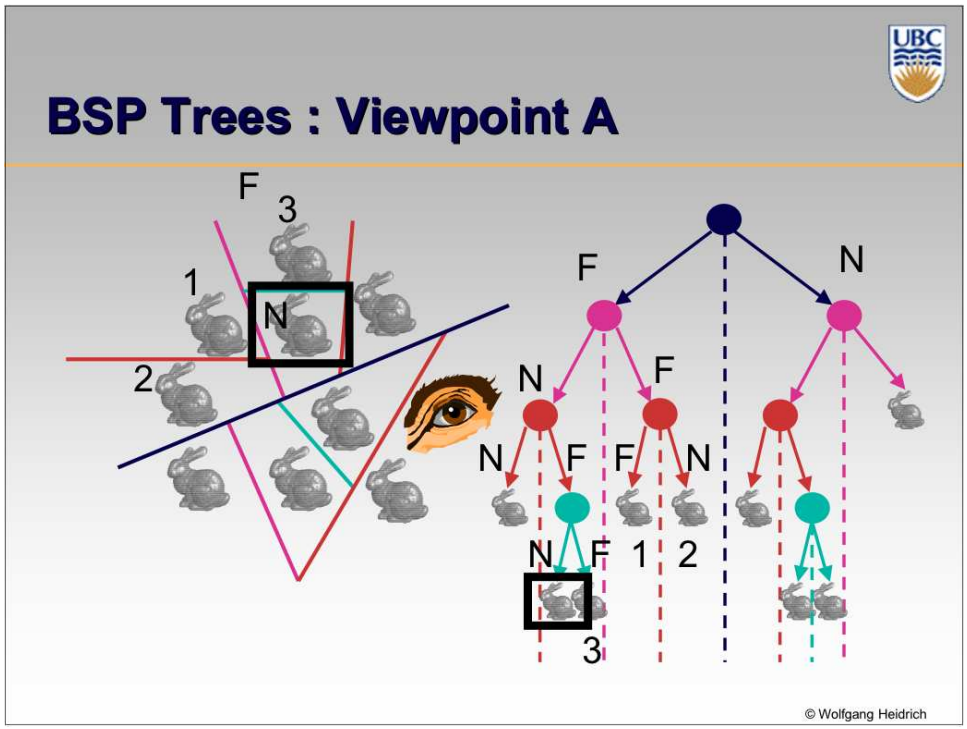
BSP Trees : Viewpoint A



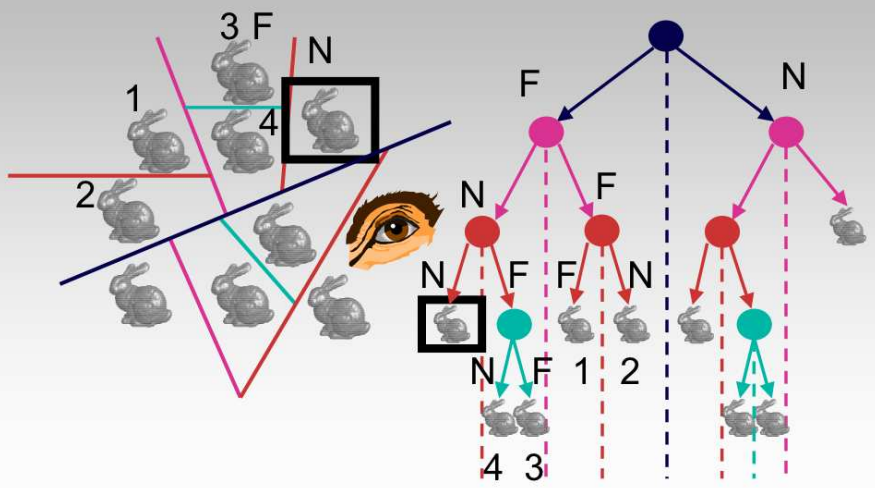

BSP Trees : Viewpoint A
F
N
1
2
F
N
N
F
N
F
F
N
1
2
© Wolfgang Heidrich


BSP Trees : Viewpoint A
F
1
N
2
F
N
N
F
N
F
F
N
N
F
1
2
© Wolfgang Heidrich

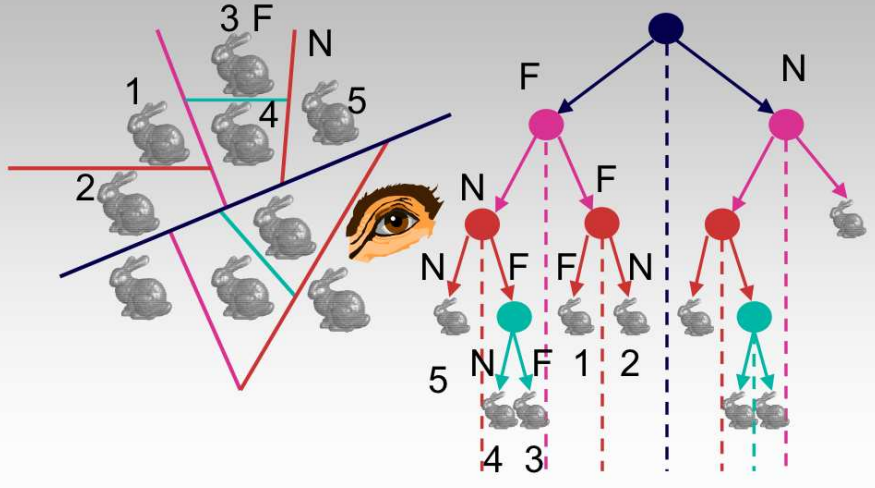

BSP Trees : Viewpoint A
F
1
N
2
F
N
N
F
N
F
F
N
N
F
1
2
© Wolfgang Heidrich


BSP Trees : Viewpoint A
F
3
1
N
2
F
N
N
F
N
F
F
N
N
F
1
2
3
© Wolfgang Heidrich

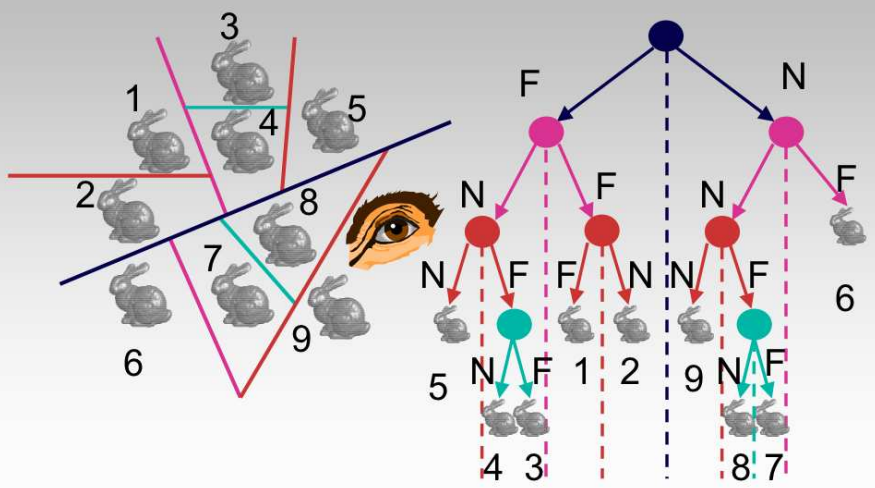
BSP Trees : Viewpoint A



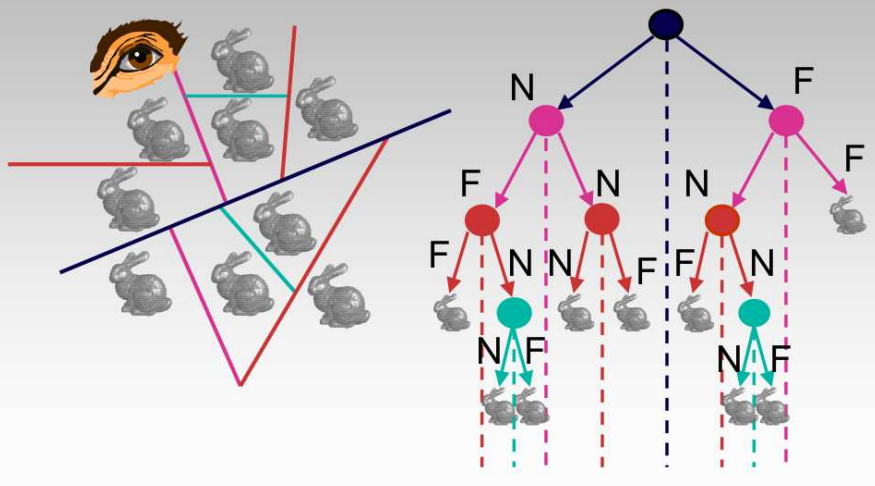
BSP Trees : Viewpoint A



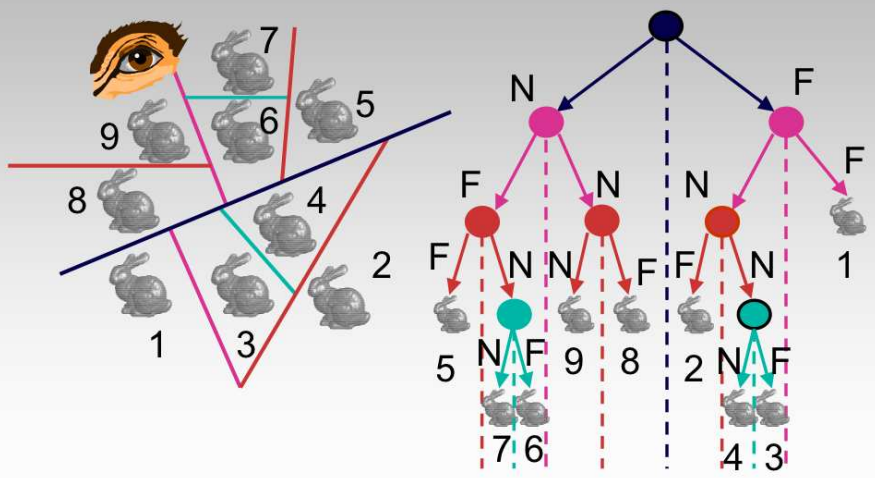
BSP Trees : Viewpoint A



BSP Trees : Viewpoint B

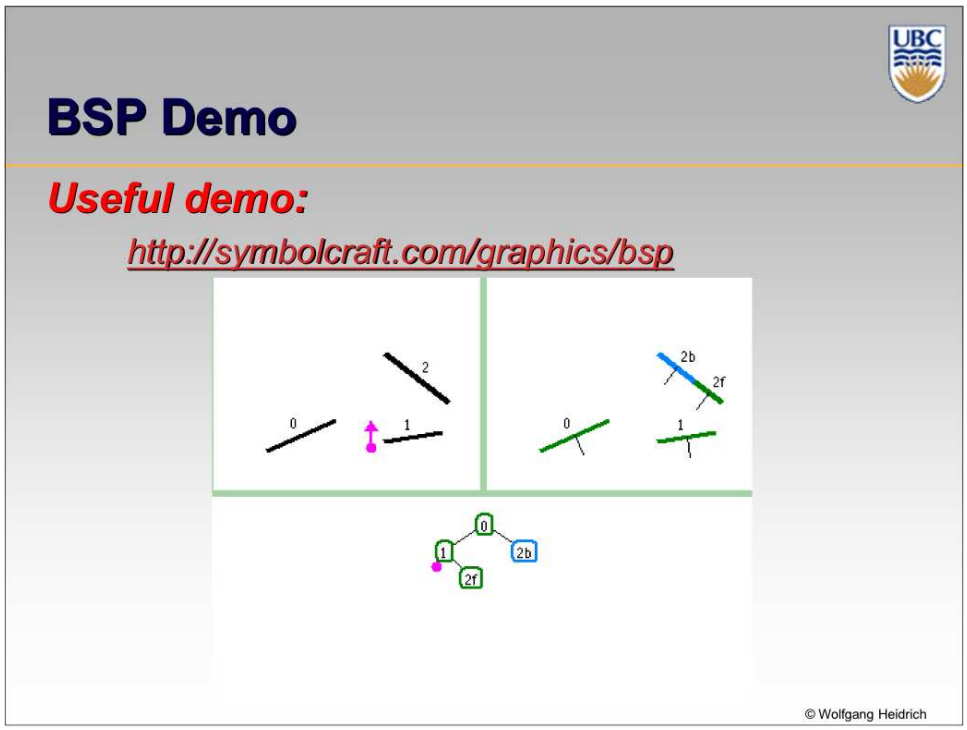


BSP Trees : Viewpoint B



BSP Tree Traversal: Polygons

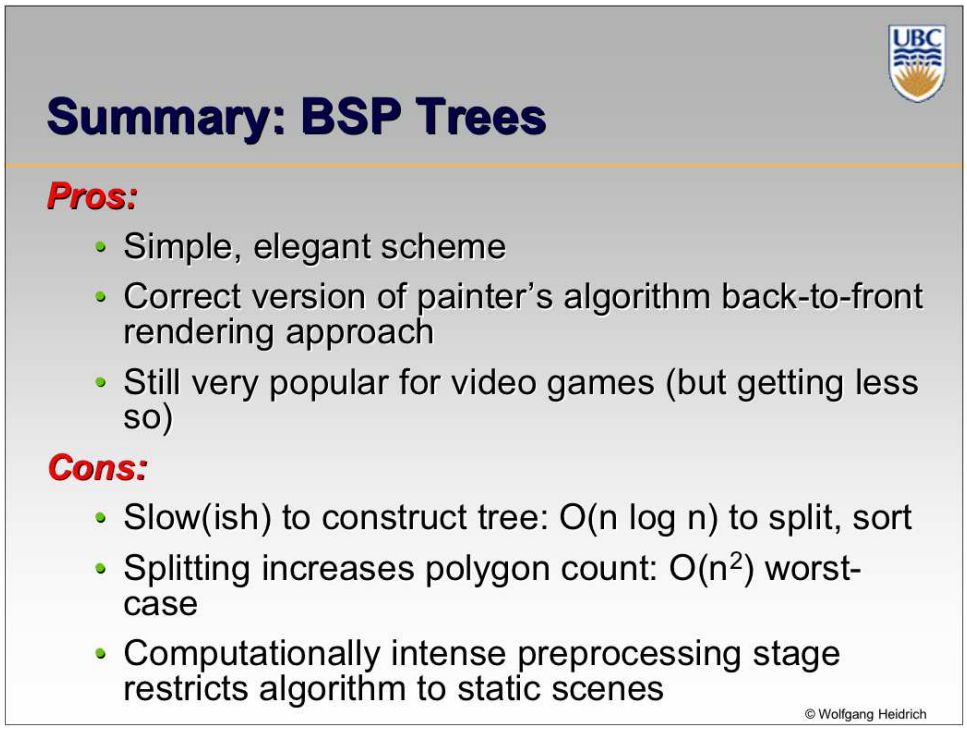
- Split along the plane defined by any polygon from scene
- Classify all polygons into positive or negative half-space of the plane
  - *If a polygon intersects plane, split polygon into two and classify them both*
- Recurse down the negative half-space
- Recurse down the positive half-space



BSP Demo

Useful demo:

http://symbolcraft.com/graphics/bsp



Summary: BSP Trees

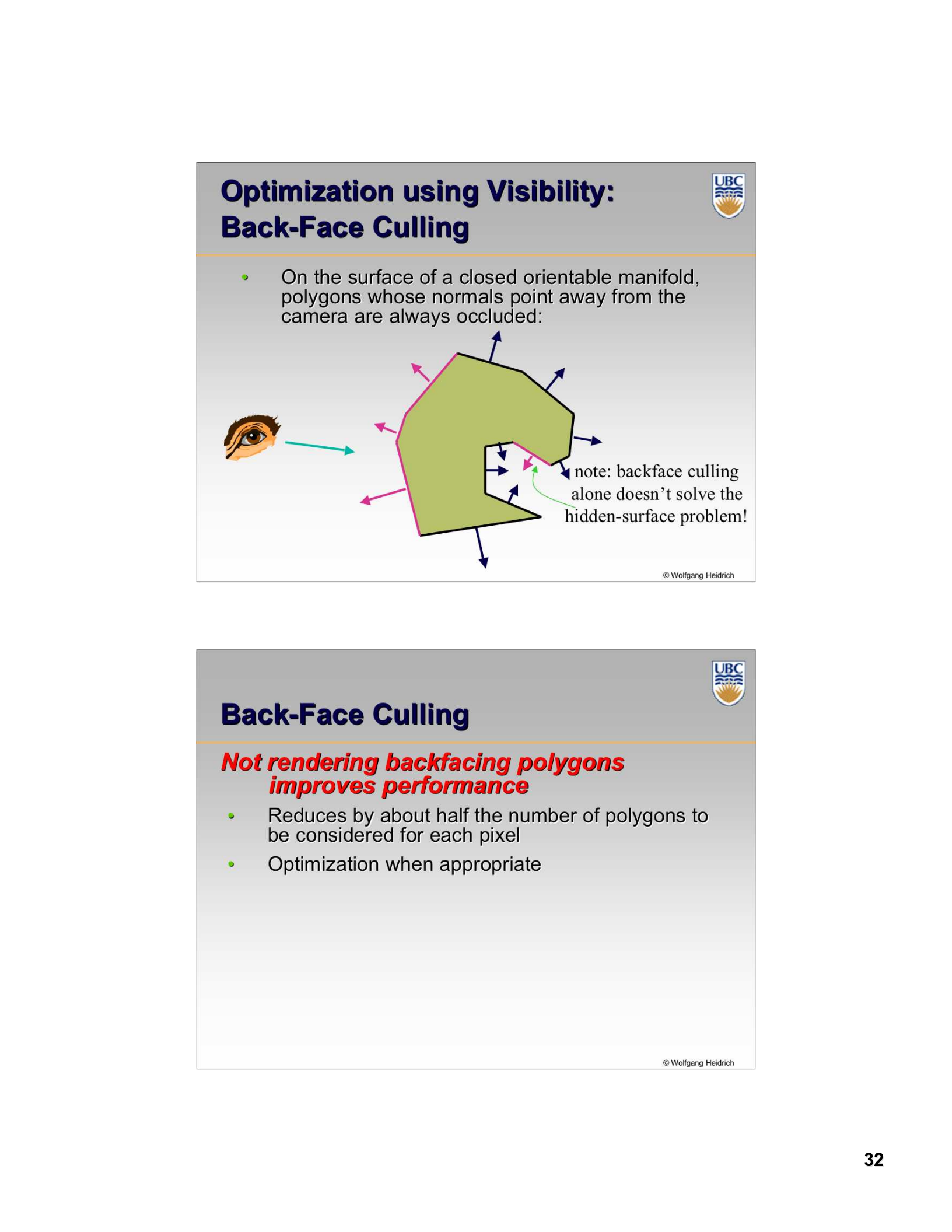
Pros:

- Simple, elegant scheme
- Correct version of painter's algorithm back-to-front rendering approach
- Still very popular for video games (but getting less so)

Cons:

- Slow(ish) to construct tree: O(n log n) to split, sort
- Splitting increases polygon count: $O(n^2)$ worst-case
- Computationally intense preprocessing stage restricts algorithm to static scenes

Optimization using Visibility: Back-Face Culling



- On the surface of a closed orientable manifold, polygons whose normals point away from the camera are always occluded:

Back-Face Culling

Not rendering backfacing polygons improves performance

- Reduces by about half the number of polygons to be considered for each pixel
- Optimization when appropriate

Back-Face Culling

Most objects in scene are typically "solid"

rigorously: orientable closed manifolds

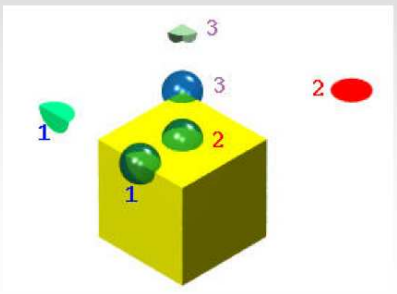
- Orentable: must have two distinct sides
  - *Cannot self-intersect*
  - *A sphere is orientable since has two sides, 'inside' and 'outside'.*
  - *A Mobius strip or a Klein bottle is not orientable*
- Closed: surface encloses a volume
  - *Sphere is closed manifold*
  - *Plane is not*

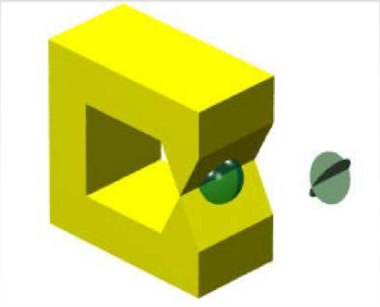
Back-Face Culling

Most objects in scene are typically "solid"

Rigorously: orientable closed manifolds

- Manifold: local neighborhood of all points isomorphic to disc
- Boundary partitions space into interior & exterior

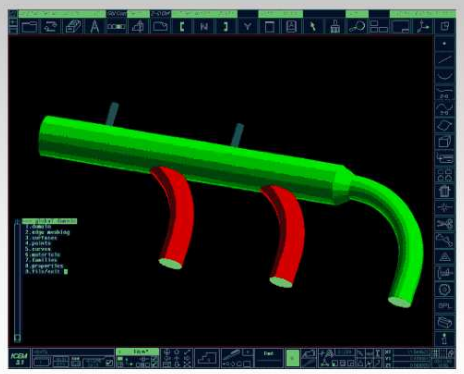




Manifold

Examples of* manifold *objects:

- Sphere
- Torus
- Well-formed CAD part

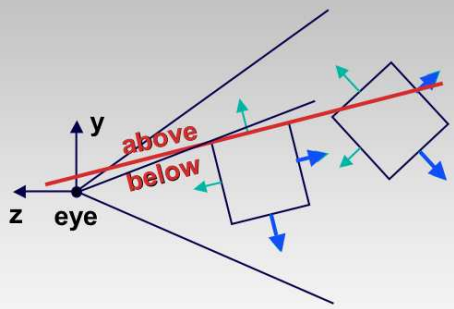


Back-Face Culling

Examples of non-manifold objects:

- A single polygon
- A terrain or height field
- Polyhedron w/ missing face
- Anything with cracks or holes in boundary
- One-polygon thick lampshade

Back-face Culling: VCS



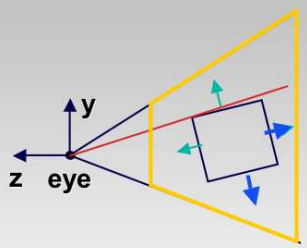
first idea:
cull if $N_z < 0$

sometimes
misses polygons that
should be culled

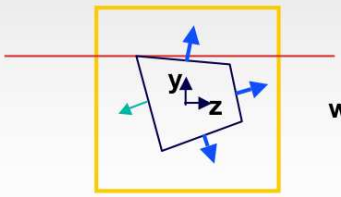
better idea:
cull if eye is below polygon plane

Back-face Culling: NDCS

VCS



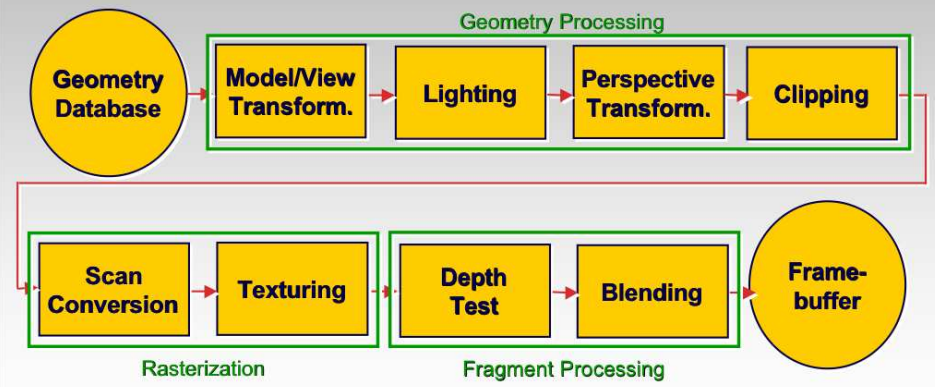
NDCS



eye

works to cull if $N_z > 0$

The Rendering Pipeline



Coming Up:

Tuesday

- Quiz 1 Solutions (Brad)

Thursday:

- Picking, Alpha Blending, Double Buffer (Brad)