

## Visibility

### How to avoid rendering polygons

- real scenes can have hundreds of millions of polys
- view frustum culling
  - trivial reject if all vertices “outside” with respect to any single plane of the viewing frustum
  - apply to groups of polygons by using bounding boxes, bounding spheres, or env grid cells
- back-face culling
  - cull if the eyepoint lies on the “backside” of a polygon
  - applies to closed solid objects (50% of polys!)

© Michiel van de Panne

## Visibility

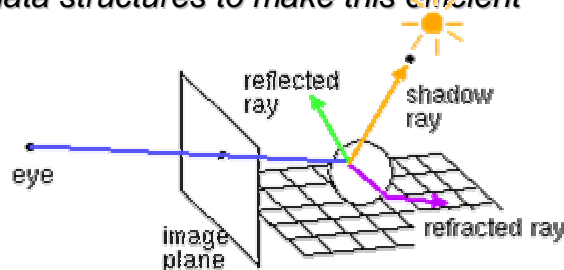
- visibility tables
  - store a list of visible cells
- horizon maps
  - for terrain models
- levels of detail
  - ‘coarse’ and ‘fine’ models
  - texture-mapped ‘billboards’

© Michiel van de Panne

## Visibility

### use ray casting (instead of projective rend.)

- cast a ray through each pixel
- requires efficient intersection tests
  - walk along ray until first intersection
  - use data structures to make this efficient



© Michiel van de Panne

## Ray Tracing

```

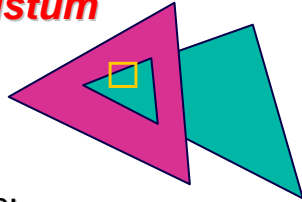
for each pixel on screen {
  determine ray from eye through pixel
  colour = raytrace(ray)
  set pixel to colour
}

colour raytrace(ray){
  find closest intersection of ray with an object
  reflect_colour = raytrace(reflected_ray)
  refract_colour = raytrace(refracted_ray)
  local_colour = lighting_computation()
  return k1*reflect_colour + k2*refract_colour
    + k3*local_colour
}
  
```

© Michiel van de Panne

# Visibility

## ... inside the view frustum



- image space algorithms:
  - operate on pixels or scan-lines
  - visibility resolved to the precision of the display
  - e.g.: Z-buffer
- object space algorithms
  - explicitly compute visible portions of polygons
  - painter's algorithm: depth-sorting, BSP trees

# Z-buffer

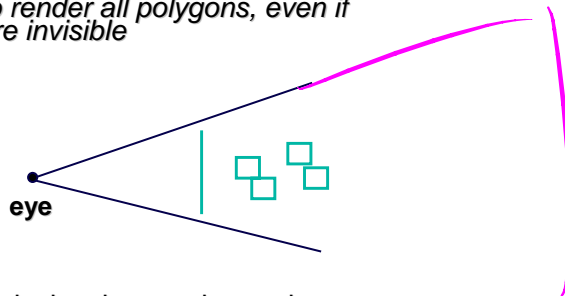
## store (r,g,b,z) for each pixel

- 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 {
  project vertices into screen-space, i.e., DCS
  for all pixels in P {
    if (Z_pixel < Depth[i,j]) { or <=
      Image[i,j] = C_pixel
      Depth[i,j] = Z_pixel
    }
  }
}
  
```

# Z-buffer

- hardware support in graphics cards
- poor for high-depth-complexity scenes
  - need to render all polygons, even if most are invisible

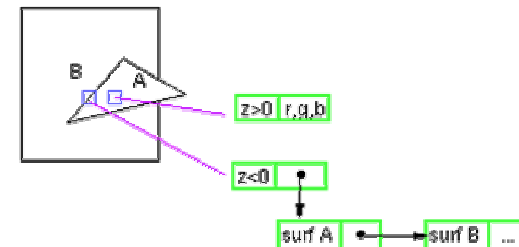


- "jaggies": pixel staircase along edges

aliasing

# The A-Buffer

- antialiased, area-averaged accumulation buffer
  - z-buffer: one visible surface per pixel
  - A-buffer: linked list of surfaces



- data for each surface includes
  - RGB, Z, area-coverage percentage, ...

# Painter's Algorithm

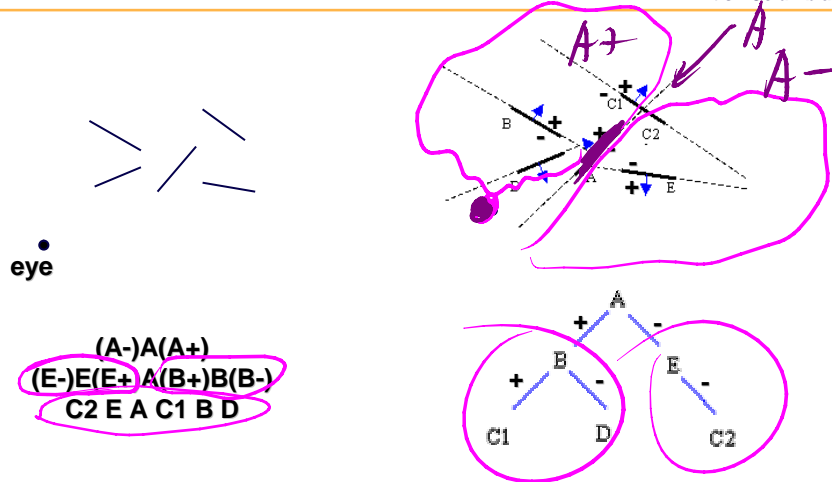
- scan-convert polygons from back to front
- but how to sort?

# BSP trees

## Binary Space Partitions

- object-space method
- produces a back-to-front ordering
- build the BSP tree once
- traverse the BSP in a view-dependent fashion

# BSP trees (example)



# Building a BSP tree

```

BSPtree *BSPmaketree(polygon list) {
  choose a polygon as the tree root
  for all other polygons {
    if polygon is in front, add to front list
    if polygon is behind, add to behind list
    else split polygon and add one part to each list
  }
  BSPtree = BSPcombinetree(BSPmaketree(front list),
    root, BSPmaketree(behind list) )
}
    
```



# Using a BSP tree

## *producing a back-to-front ordering*

```
DrawTree(BSPtree) {  
  if (eye is in front of root) {  
    DrawTree(BSPtree->behind)  
    DrawPoly(BSPtree->root)  
    DrawPoly(BSPtree->front)  
  } else {  
    DrawTree(BSPtree->front)  
    DrawPoly(BSPtree->root)  
    DrawTree(BSPtree->behind)  
  }  
}
```