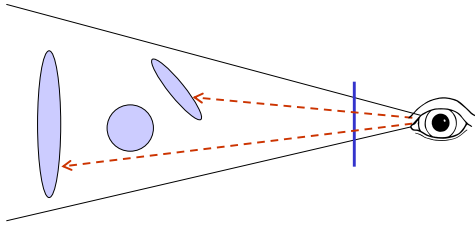


## Visible Surface Determination



### Rasterization will convert primitives to pixels in the image

- but we need to make sure we don't draw occluded objects

### For each pixel, what is the nearest object in the scene?

- this is the only thing we need to draw at this pixel
  - provided the object isn't transparent
- we need to determine the **visible surface**

## The Z-Buffer Algorithm

### Create new frame buffer channel

- a depth component
- to go with our RGB $\alpha$  channels

### Records depth of pixel contents

- overwrite pixel that's farther away

### This used to look pretty wasteful

- say 24 bits \* number of pixels
- doubles size of framebuffer
- but memory is cheap now

### Now most common method

- especially for hardware design

#### Z-Buffer Algorithm:

```
allocate z-buffer
initialize values to infinity
```

```
loop over all objects
  rasterize current object
  for each covered pixel
    (x, y)
      if z(x, y) < zbuffer(x, y)
        zbuffer(x, y) = z(x, y)
      write pixel
```

OpenGL — `glEnable(GL_DEPTH_TEST)`

## Looking at the Z-Buffer Algorithm

### It has some attractive strengths

- it's very simple, and easy to implement in hardware
- can easily accommodate any primitive you can rasterize
  - not just planar polygons

### But it does have a few problems

- it doesn't handle transparency well
- needs intelligent selection of *znear* & *zfar* clipping planes
  - z-buffers typically use integer depth values
  - fixed bit precision mapped to range *znear..zfar*

## Making Z-Buffers Efficient

### When we rasterize a polygon, we need *z* value at each pixel

- we could just compute it at every pixel
- but this is pretty expensive

### Can use the same incrementalization trick as in rasterization

- the projected polygon satisfies some plane equation

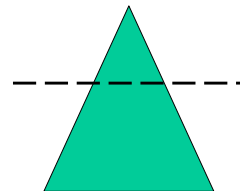
$$ax + by + cz + d = 0$$

- we could compute the depth as

$$z = \frac{-d - ax - by}{c}$$

- but taking account of coherence

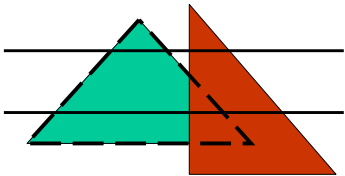
$$\Delta z = -\frac{a}{c} \Delta x \quad \text{for fixed values of } y$$



## Scanline Visibility

### Looks a lot like polygon rasterization

- maintains active edge table
- looks at one scanline at a time — no need to store entire image
  - nice if memory is scarce



#### Scanline Algorithm:

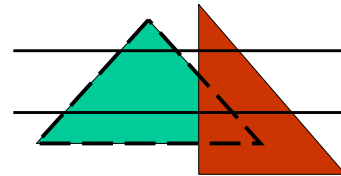
```
sort edges by ymin

loop over scanlines
  update active edge list
  sort active edges by x
  loop over x values
    find closest active edge
    write pixel
```

## Scanline Visibility

### Using Depth Coherence

1. Assume polygons do not penetrate each other.
2. Depth comparison only at the beginning of each span.



## Painter's Algorithm

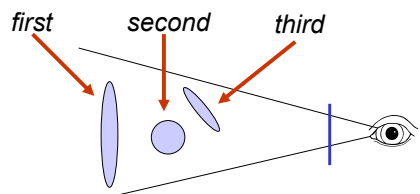
### Developed thousands of years ago

- probably by cave dwellers

### Draws every object in depth order

- from back to front
- near objects overwrite far objects

### What could be simpler?



#### Painter's Algorithm:

```
sort objects back to front

loop over objects
  rasterize current object
  write pixels
```

## But the Catch is in the Depth Sorting

### What do we sort by?

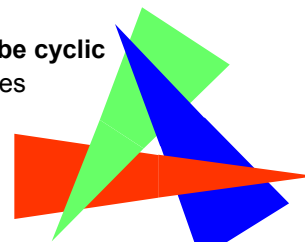
- minimum  $z$  value — no
- maximum  $z$  value — no



- in fact, there's no single  $z$  value we can sort by

### Worse yet, depth ordering of objects can be cyclic

- may need to split polygons to break cycles



## Looking at Painter's Algorithm

### It has some nice strengths

- the principle is very simple
- handles transparent objects nicely
  - just composite new pixels with what's already there

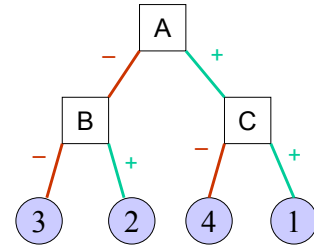
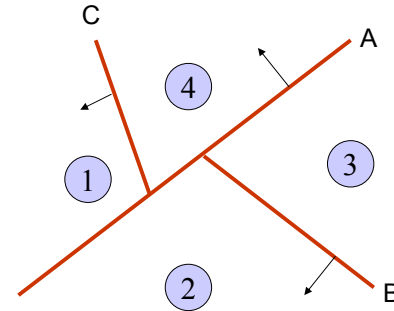
### But it also has some noticeable weaknesses

- general sorting is a little expensive — worse than  $O(n)$
- need to do splitting for depth cycles, interpenetration, ...
- and what if the objects aren't planar polygons?

## A Quick Look at BSP Trees

### Recursively partition space with planes

- this defines a **binary space partitioning** tree
- need to split objects hit by planes



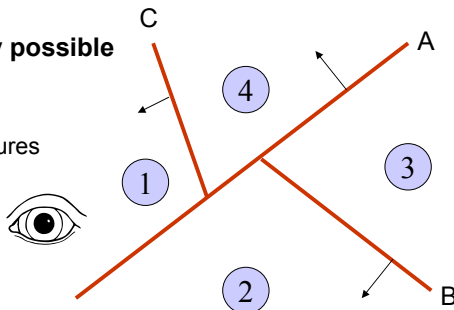
## A Quick Look at BSP Trees

### Can use this to draw scene in order (back to front for instance)

- start at root plane
- figure out which side the viewpoint is on
- descend on the opposite first
- do this recursively

### Can use one BSP tree for any possible viewpoint.

View Independent Data Structures



## A Quick Look at BSP Trees

### Can use this to draw scene in order (back to front for instance)

- start at root plane
- figure out which side the viewpoint is on
- descend on the opposite first
- do this recursively

### Originally developed in early 80's

- for Painter's Algorithm, for instance
- resurrected by PC game programmers in the early 90's
  - e.g., by John Carmack for Doom
- it's quite handy if you don't have a z-buffer

## Ray Casting

### This is a very general algorithm

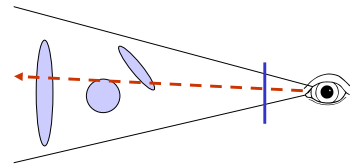
- works with any primitive we can write intersection tests for
- but it's hard to make it run fast

### We'll come back to this idea later

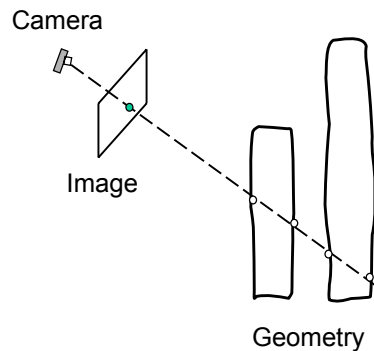
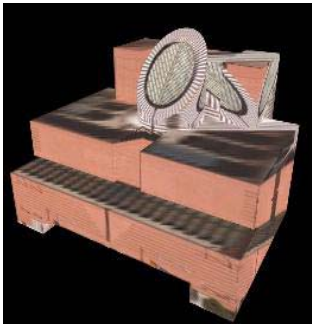
- can use it for much more than visibility testing
- shadows, refractive objects, reflections, motion blur, ...

#### Ray Casting:

```
loop over every pixel (x,y)
  shoot ray from eye through (x,y)
  intersect with all surfaces
  find first intersection point
  write pixel
```



## Visibility Processing for Image-Based Rendering



Like to attach real photographs onto geometry.  
Consider each camera position as a viewpoint.

## A Classification of Visibility Algorithms

### Image-space

Z-buffer

Scanline

Ray-casting

### Object-space

Painter's algorithm

BSP

They are all view-dependent except BSP trees.

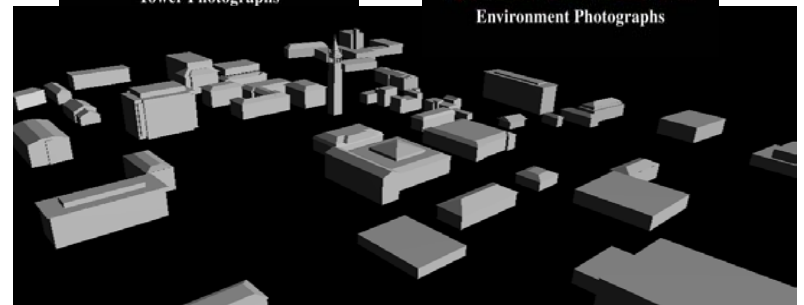
## Input Photographs and Geometric Model



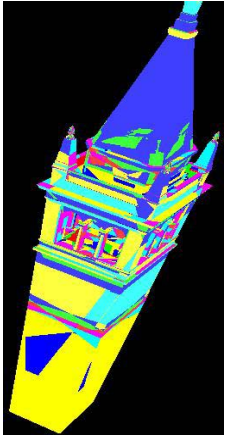
Tower Photographs



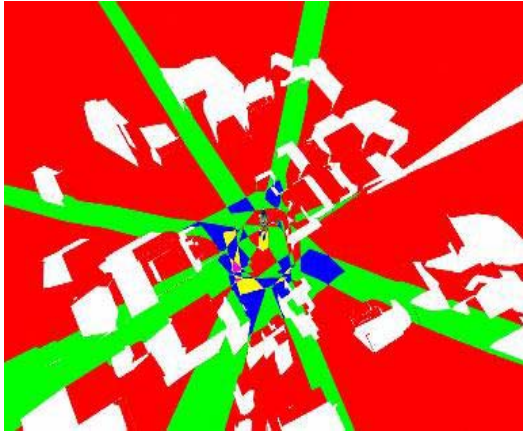
Environment Photographs



## Visibility Processing Results



The tower



The rest of the campus

## Synthetic Renderings

