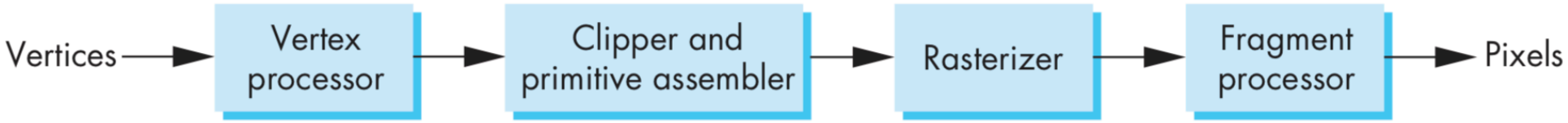
COMP3271 Computer Graphics

Hidden-Surface Removal

2019-20

Graphics Pipeline Overview



Hidden-Surface Removal — To determine which fragments are visible (i.e., those that are within the view volume and are not blocked by others as viewed from the camera)

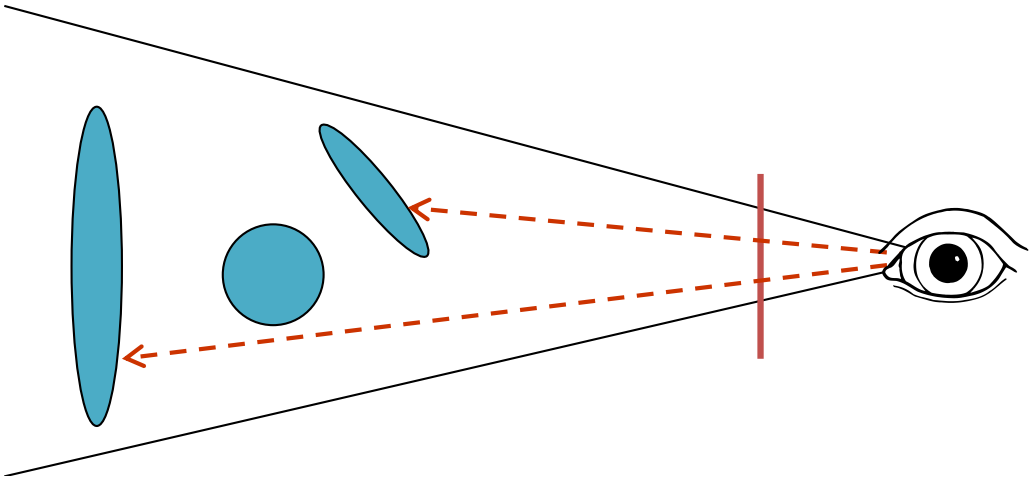
Hidden-Surface Removal

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**
  - Hidden-surface removal is also called visible surface determination, or back-face removal (culling)



The Z-Buffer Algorithm

A full screen buffer, called the **depth buffer** (or **z-buffer**), is used to store floating-point depth information for each pixel in the framebuffer.

Before a fragment's color is written into the framebuffer,

- Compares its depth with the value stored at the corresponding pixel of the depth buffer.
- If the depth is smaller than the existing value, overwrites the pixel in the framebuffer and update the depth buffer with new depth value.
- Otherwise, discard the fragment.

The Z-Buffer Algorithm

Create new framebuffer channel

- a depth component
- to go with our RGB channels

Records depth of pixel contents

- overwrite pixel that's farther away

This used to look pretty wasteful

- say 24 bits * number of pixels
- but memory is cheap now

Now most common method

- especially for hardware design

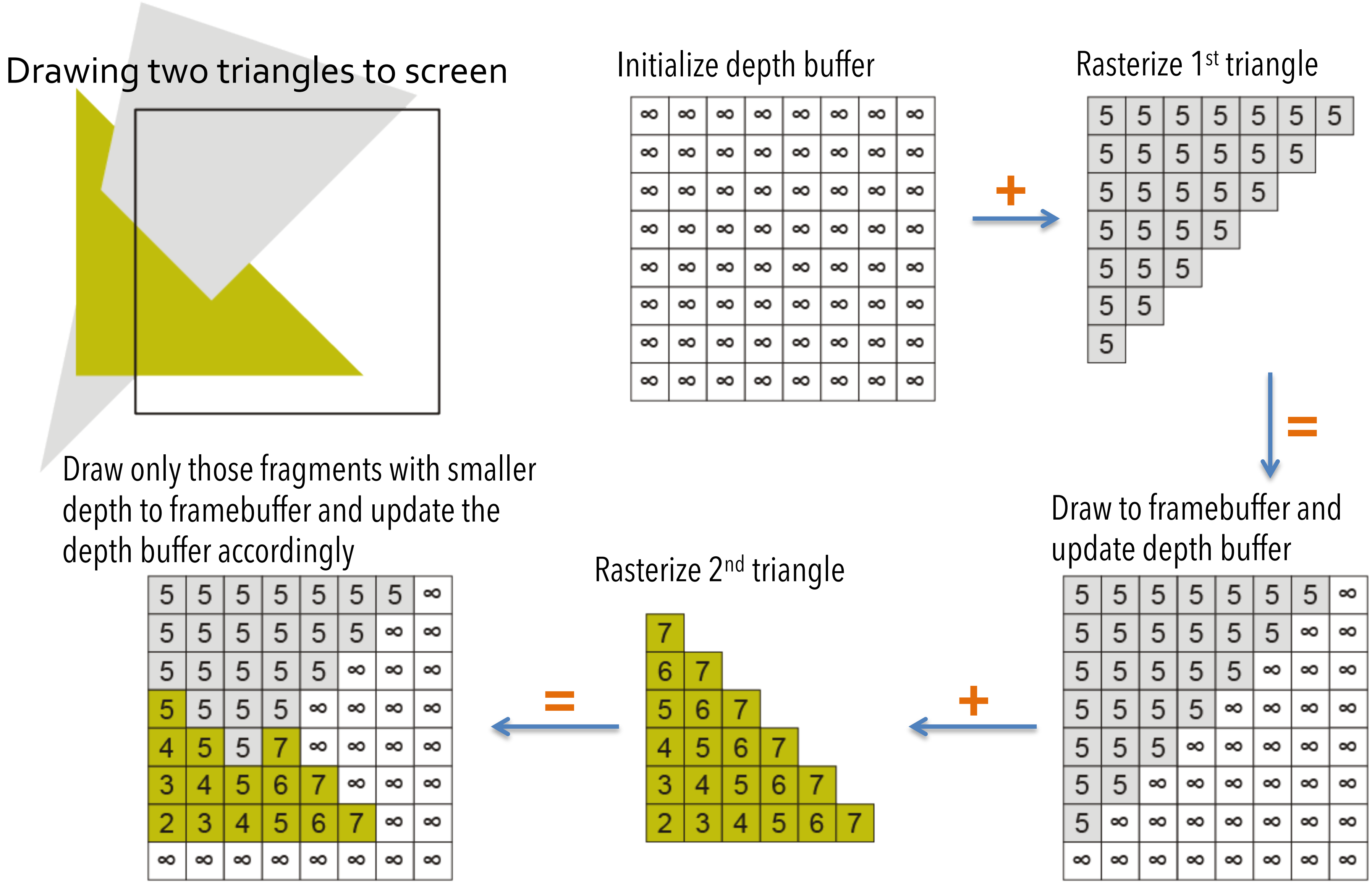
OpenGL — glEnable(GL_DEPTH_TEST)

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
```

The Z-Buffer Algorithm



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

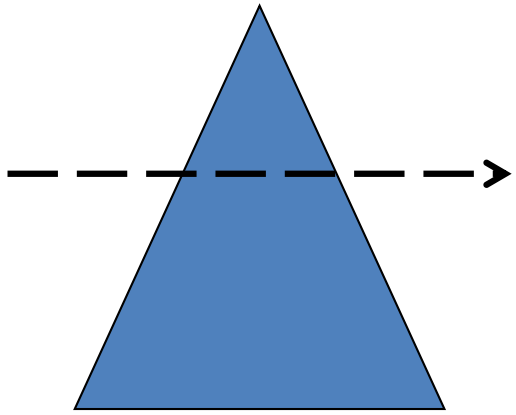
$$ax + by + cz + d = 0$$

- we could compute the depth as

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

- but taking account of coherence

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



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
  - consider drawing a closer transparent object first and then a farther opaque object
- needs intelligent selection of *znear* & *zfar* clipping planes
  - fixed bit precision mapped to range *znear..zfar*
  - Recall nonlinear z-mapping for perspective projection

Handling Transparency with Z-Buffering

With z-buffering, the scene can be drawn in any arbitrary order (and hence no need to sort a drawing order for the triangles), provided that there aren't any objects with transparency.

With transparent/semi-transparent objects

- first render all opaque objects with normal z-buffering
- Then disable write to depth buffer and render transparent objects, still perform depth buffer test to make sure transparent fragments that are behind opaque ones won't be drawn

Other Issues with Z-buffering

Z-fighting: Due to the lack of precision of depth buffer, two fragments from different objects that are close to each other (and far from the camera) will flicker back and forth on alternating frames.

- Alleviation: use 24-bit or 32-bit depth buffers or adjust near and far clipping planes for a smaller view frustum

Take a look at the example: https://youtu.be/9AcCrF_nX-I

Z-buffering is done on a pixel-by-pixel basis. If an entire object is blocked by another, it still needs to undergo all the lighting calculations and rasterization before all its pixels are discarded.

- Solutions: object culling algorithms such as BSP (binary spatial partitioning), occlusion volumes, etc.

Painter's Algorithm

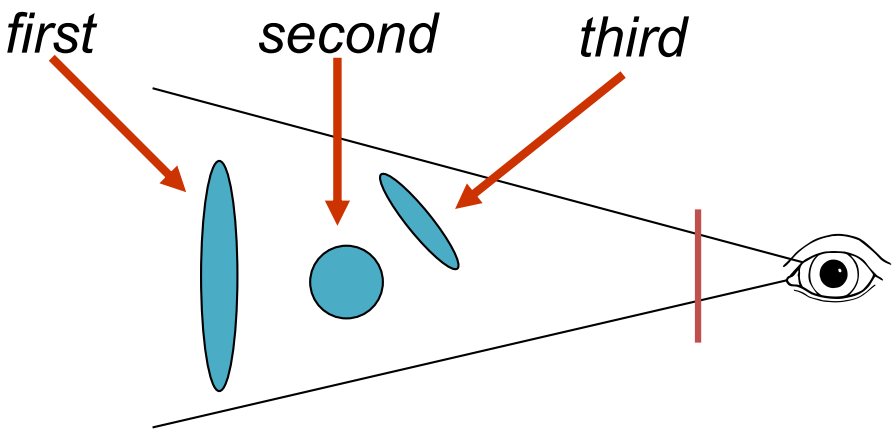
Developed thousands of years ago

- 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?

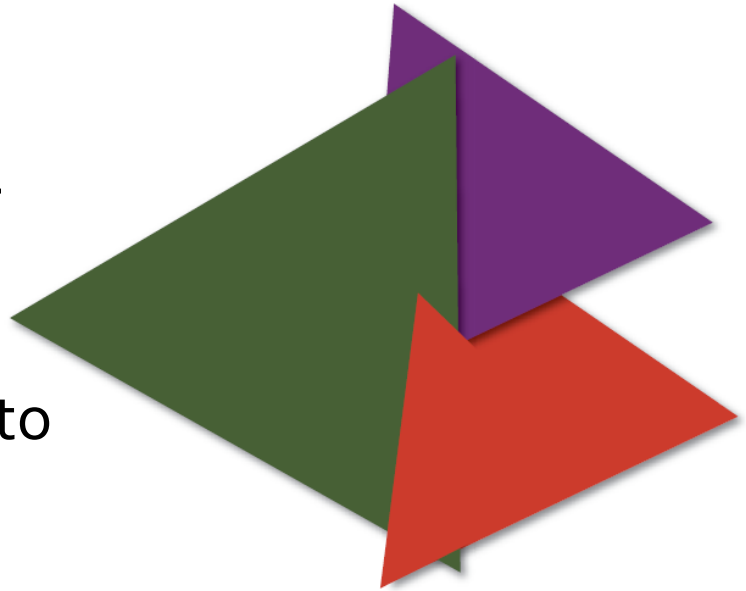
- minimum z value — no

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

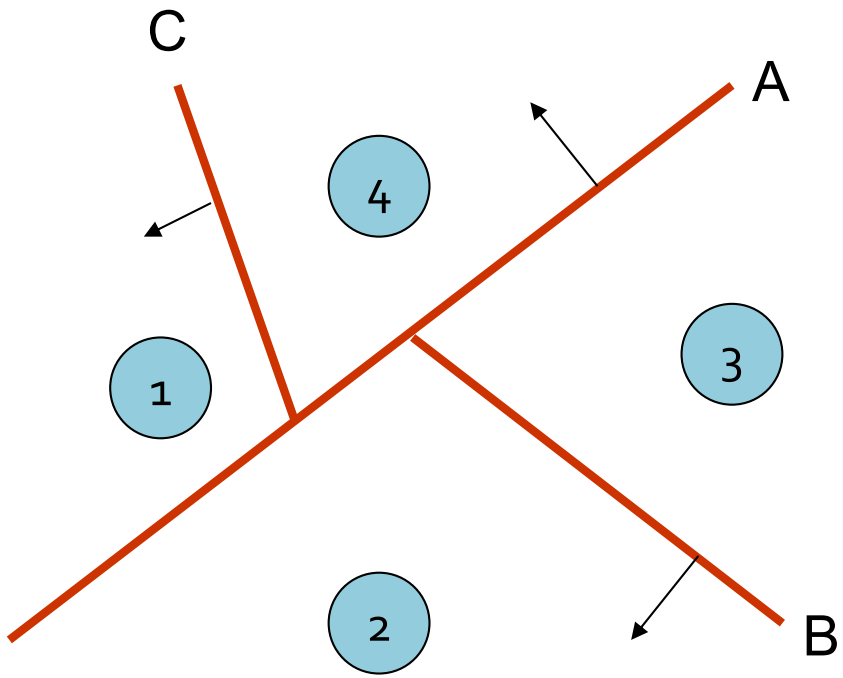
But it also has some noticeable weaknesses

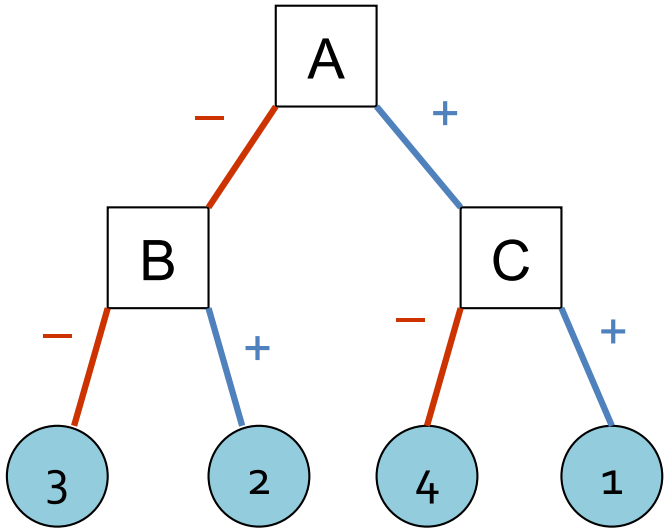
- general sorting is a little expensive — worse than *O(n)*
- need to do splitting for depth cycles, interpenetration, …

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

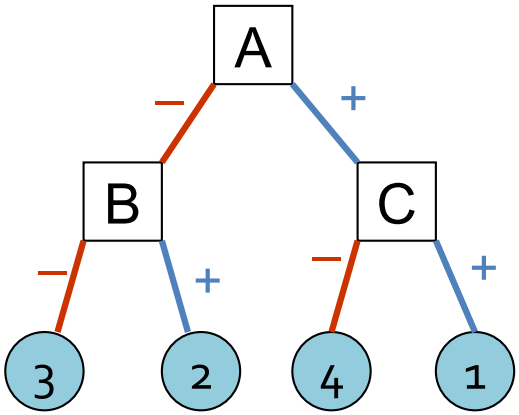




A Quick Look at BSP Trees

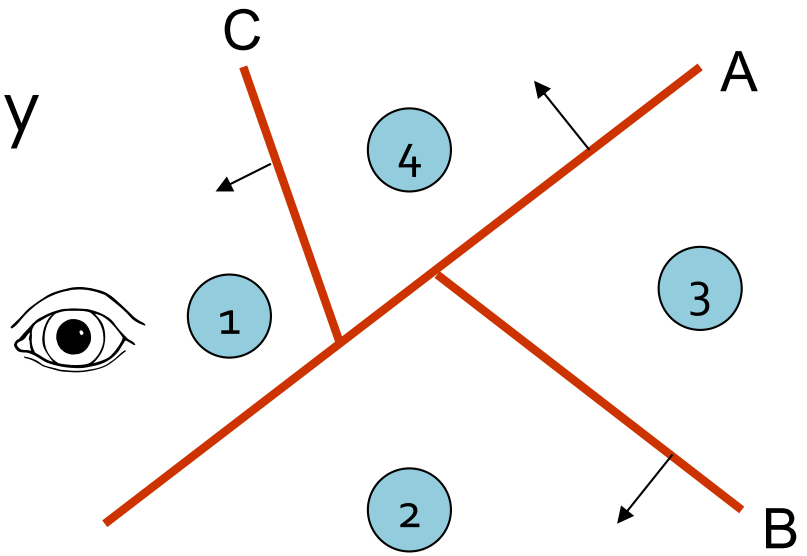
Can use this to draw scene in order (back to front for instance)

- 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

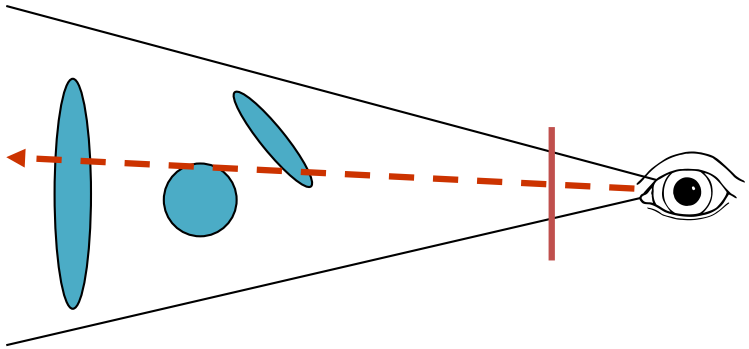
- works with any primitive we can write intersection tests for
- but it's hard to make it run fast

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
```



A Classification of Visibility Algorithms

Image-space

- Z-buffer
- Ray-casting

Object-space

- Painter's algorithm
- BSP

They are all view-dependent except BSP trees.