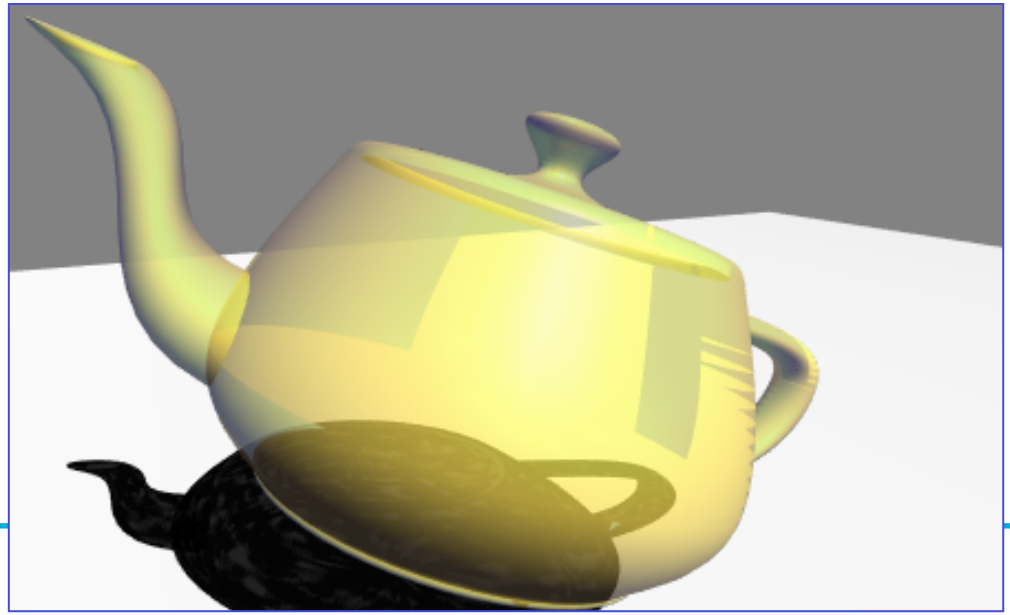
Lecture 8
Hidden Surface Removal

Lecture outline:

- Introduction to Hidden Surface Removal
- Method 1: Ray Casting
- Method 2: Z Buffer Method
- Method 3: Back face Culling
- Method 4: Potentially Visible Set (PVS)
- Hidden Surface Removal in OGL

What we have learnt so far

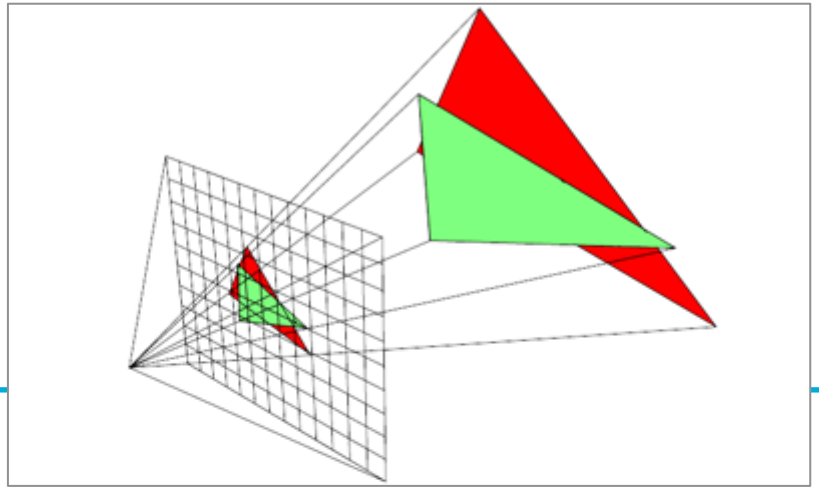
- Construct a 3D hierarchical geometric model
- Affine Transformation of the model in 3D
- Define a virtual camera and Project the 3D model accordingly
- Rasterize geometries in 3D space to pixels on the screen
- What's missing?



Introduction

- Hidden surface removal!!! Why we need it?

- Not every part of every 3D object is visible to a particular viewer. We need an algorithm to determine what parts of each object should be drawn.
- Known as "hidden surface removal/elimination" or "visible surface determination"
- Hidden surface removal algorithm can be classified in major ways:
  - Object order vs. Image order
  - Sort first vs. Sort last



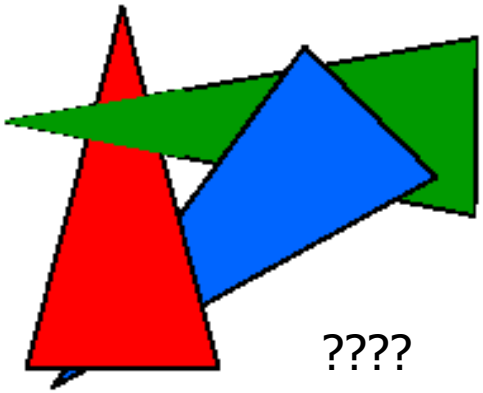
Object Order vs. Image Order

- Object order
  - Consider each 3D object (in the scene) only once - draw its pixels and move on to the next object
  - Might draw the same pixel multiple times
  - E.g. Z-buffer
- Image order
  - Consider each pixel only once - draw part of an object and move on to the next pixel
  - Might compute relationships between objects multiple times
  - E.g. Ray Casting

Sort First vs. Sort Last

- Sort first
  - Find some depth-based ordering of the objects relative to the camera, **then** draw from back to front, e.g. Painter's algorithm
  - Build an ordered data structure to avoid duplicating work
- Sort last
  - Sort implicitly as more information becomes available, e.g. Z-buffer

Painter's algorithm
- Sort everything and
- Render from back to front

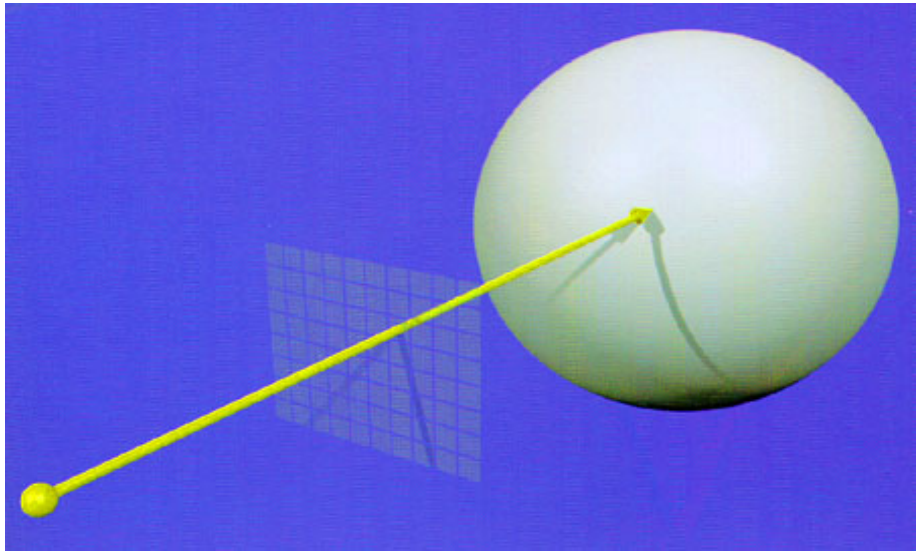


Important Algorithms

1. Ray casting
2. Z-buffer
3. Back face culling
4. PVS

(1) Ray Casting

- Partition the projection plane (PP) into pixels to match screen resolution
  - For each pixel p_i, construct ray (projector) from COP through PP at that pixel and into scene
  - Intersect the ray with every object in the scene, color the pixel according to the object with the closest intersection

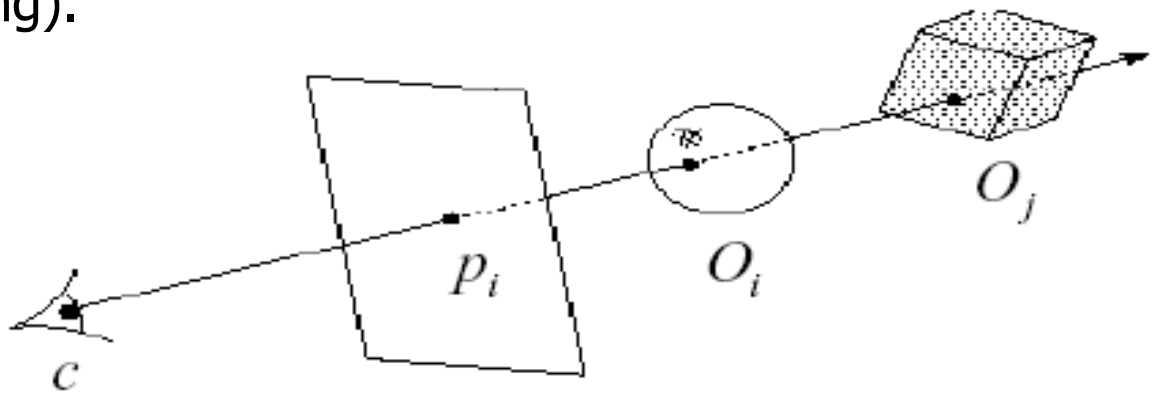


Ray Casting Implementation

- Parameterize the ray:

$$R(t) = (1\text{-}t)c + tp_i$$

  - If a ray intersects some object O_i, get parameter t such that first intersection with O_i occurs at $R(t_i)$
  - Which object owns the pixel?
- We will study ray-object intersections in more detail later (essential in ray tracing).



(2) Z-buffer

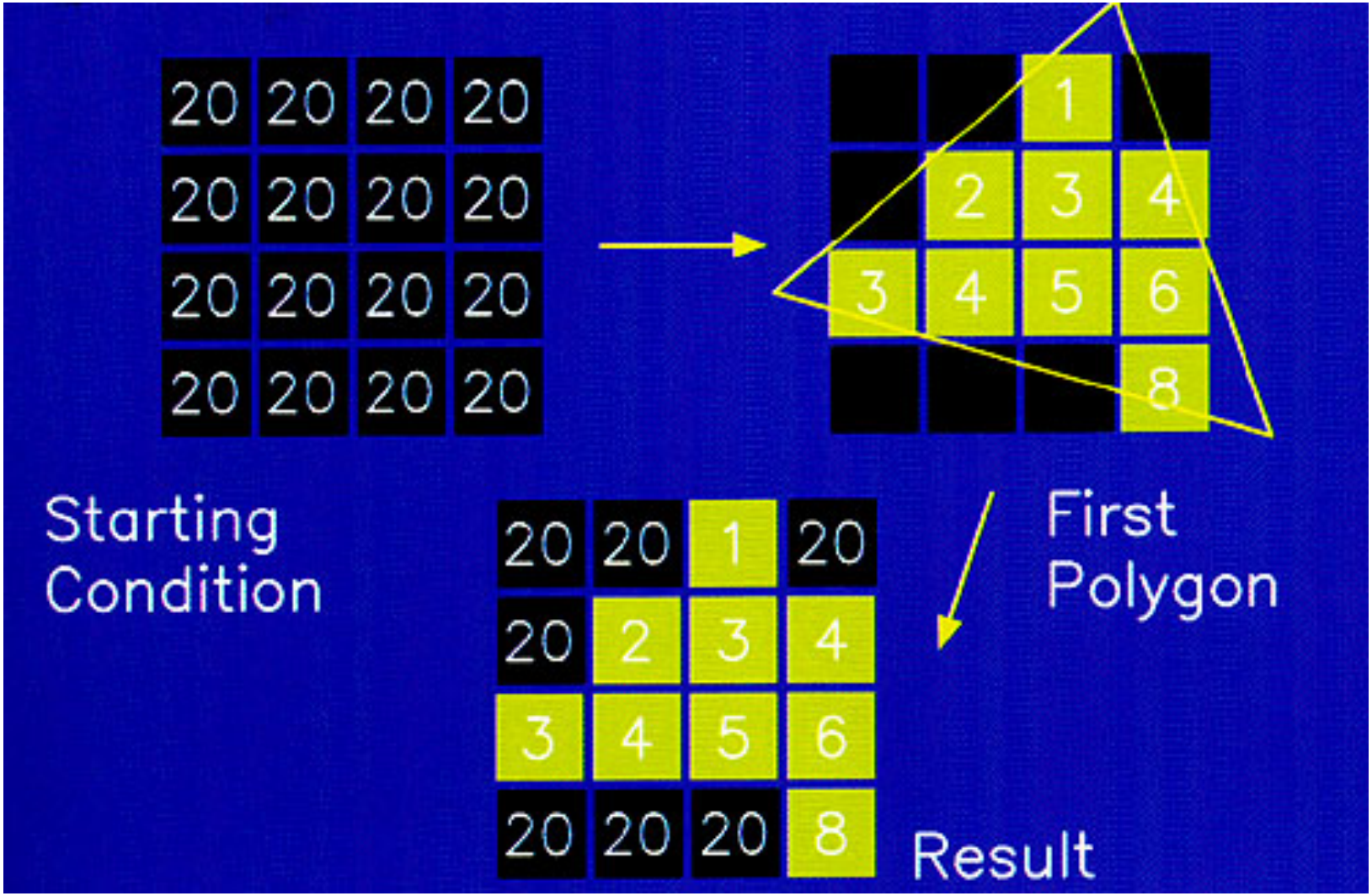
- **A book-keeping idea**: along with a pixel's red, green and blue values, maintain:
  - An additional channel in memory for the pixel's depth
  - Called the depth buffer or Z-buffer
- When the time comes to draw a pixel, compare its depth with the depth of what's already in the Z buffer. Replace only if it's closer
  - Very widely used in hardware from 90', e.g. SGI, NVidia, ATI, etc.
- History
  - Originally described as "brute-force image space algorithm"
  - Written off as impractical algorithm for huge memories (in 1974 by Catmull)
  - Today, done easily in hardware

Z-buffer Implementation

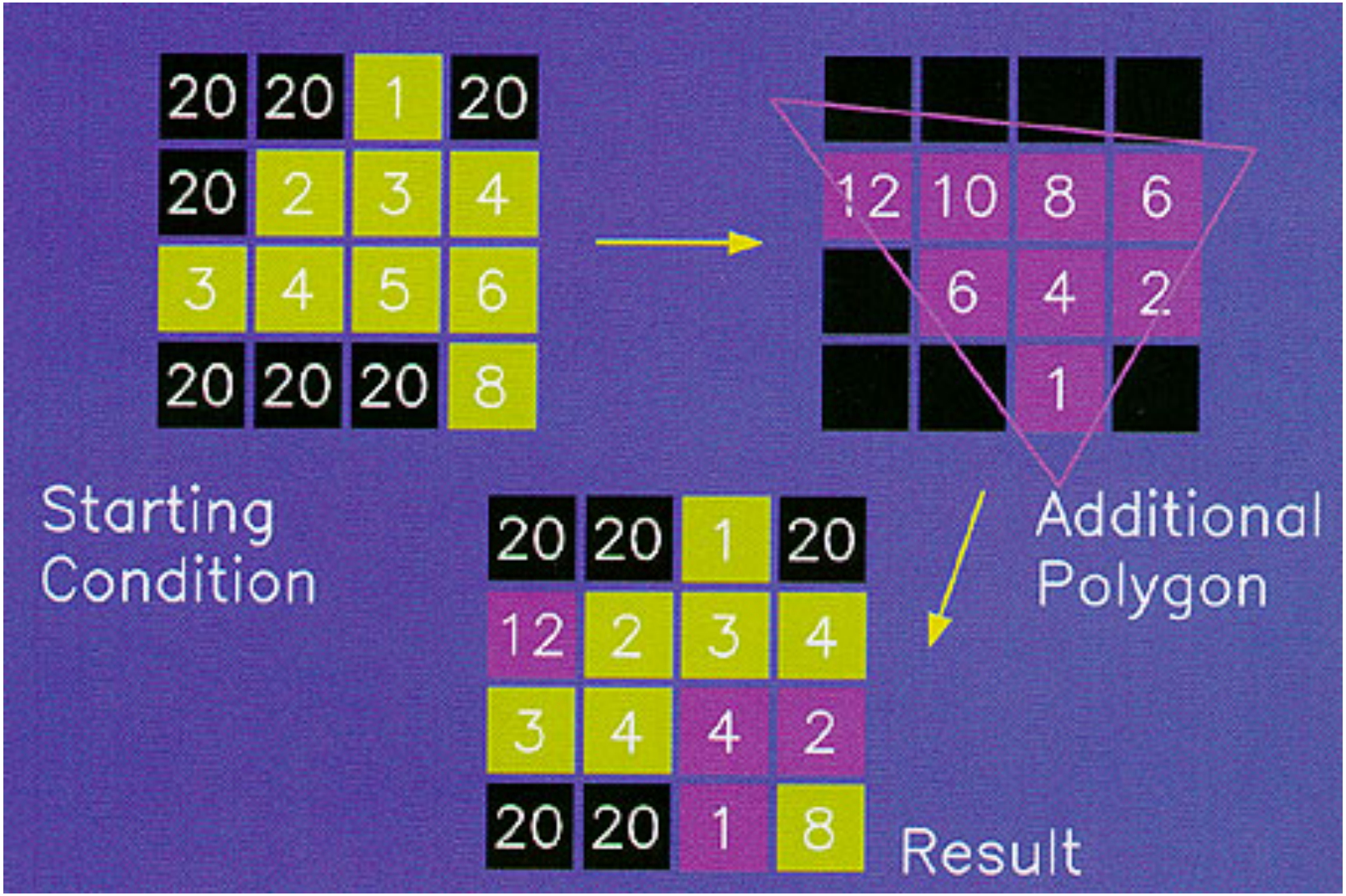
```
# Clear depth and frame (RGB) buffer
For each pixel pi
    Z-buffer     [ pi ] = FAR
    Frame-buffer [ pi ] = BACKGROUND_COLOR
End

# Render each polygon with Z buffer
For each polygon P
    For each pixel pi in rasterized P
        compute depth z and shade s of P at pi
        if z < Z-buffer [ pi ]
            Z-buffer     [ pi ] = z
            Frame-buffer [ pi ] = s
        End
    End
End
```

Z-Buffer Algorithm

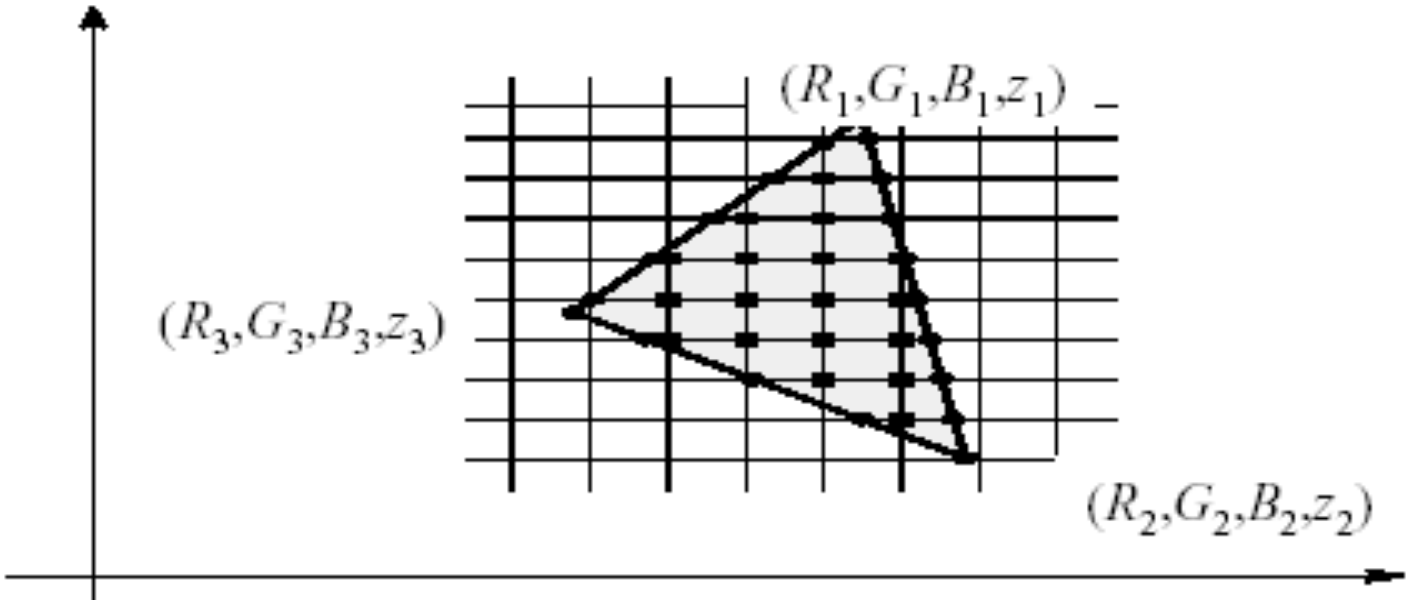


Z-Buffer Algorithm



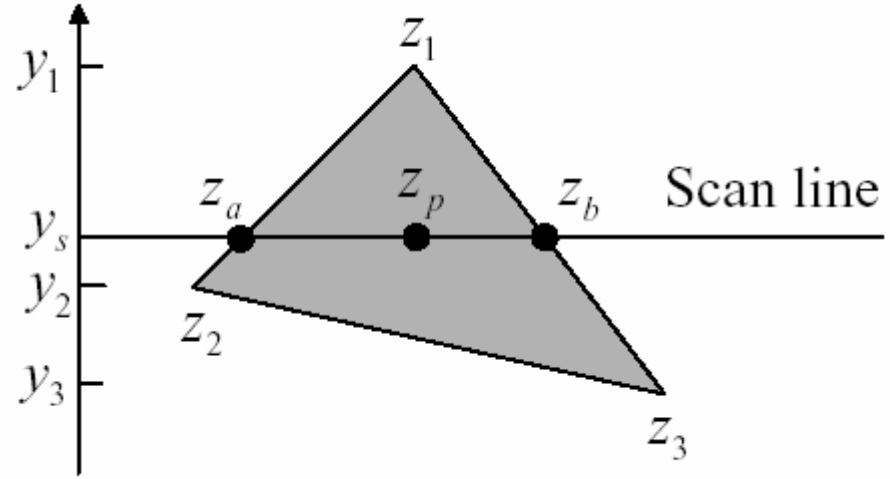
Z-buffer Tricks

- The shade of a triangle can be computed incrementally from the shades of its vertices (taking advantage of coherence)
- Note: remember the rasterization lecture?



Z Value Interpretation

- Method 1: **bilinear interpolation** or Method 2: **barycentric coordinates**
  See module 6 rasterization lecture.



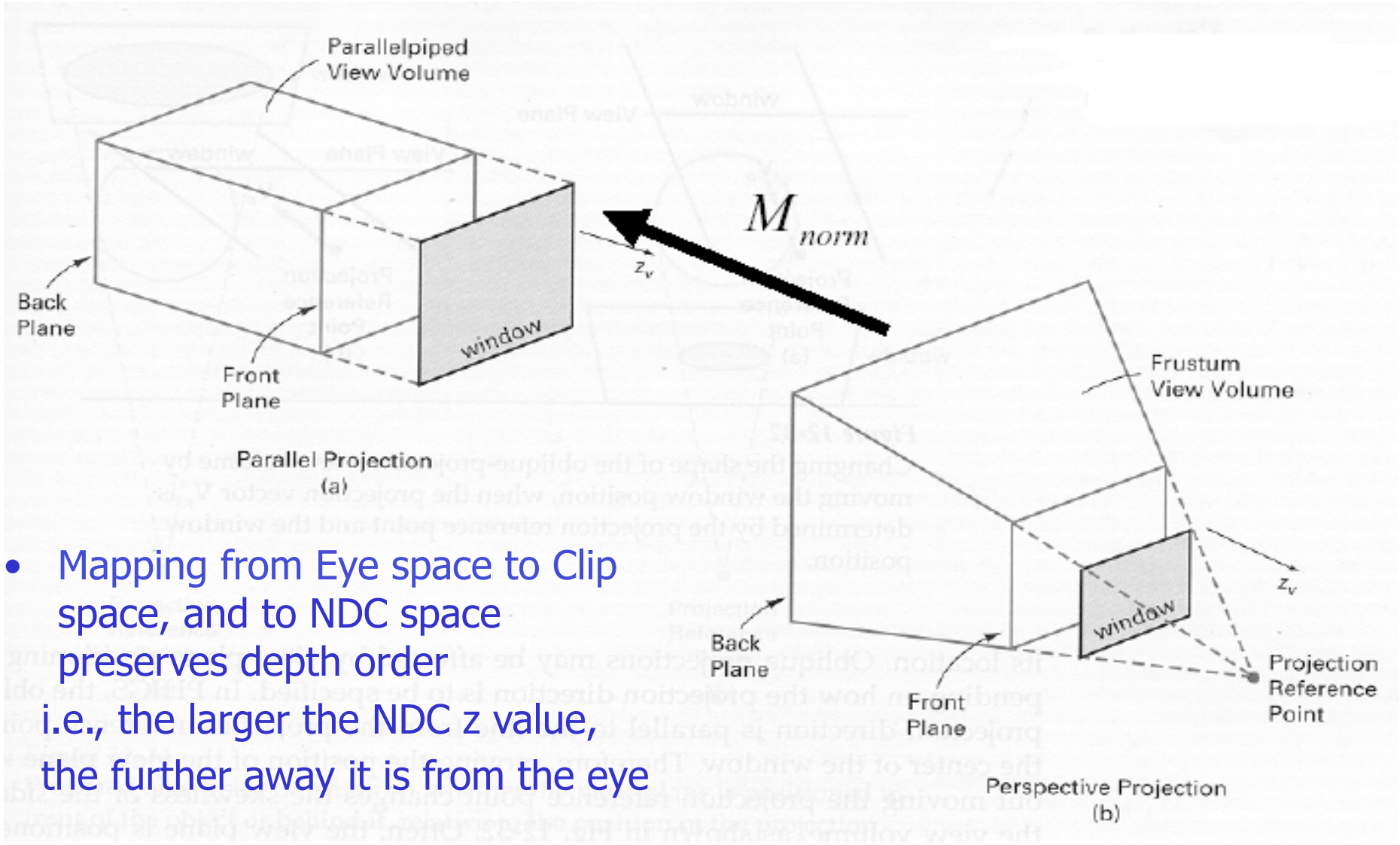
$$z_a = z_1 - (z_1 - z_2)\frac{y_1 - y_s}{y_1 - y_2}$$

$$z_b = z_1 - (z_1 - z_3)\frac{y_1 - y_s}{y_1 - y_3}$$

$$z_p = z_b - (z_b - z_a)\frac{x_b - x_p}{x_b - x_a}$$

Note: interpolation is done in the 2D screen domain. Is it alright?
Actually, we interpolate using 1/z instead of z
http://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/visibility-problem-depth-buffer-depth-interpolation

Depth Preserving Conversion: Projection Mechanism



- Mapping from Eye space to Clip space, and to NDC space preserves depth order

  i.e., the larger the NDC z value, the further away it is from the eye

Z Buffer: Analysis

- Advantages:
  - Running time is roughly proportional to the number of objects
  - Very fast! "Object-order" and "Sort-Last"
  - Simple to implement on hardware:
    De facto standard in most graphics systems.

- Disadvantage:
  - Suffers from aliasing, e.g., z-fighting
  - Cannot render objects with transparency

Don't make put the *far plane* too far and the *near plane* too near.
https://www.opengl.org/archives/resources/faq/technical/depthbuffer.htm#0040

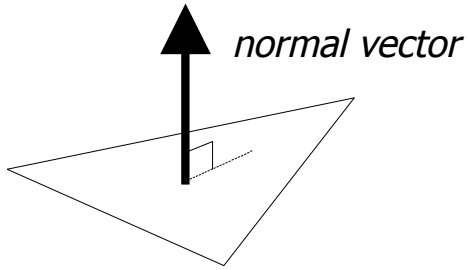
(3) Back Face Culling

- Used in conjunction with polygon-based algorithms
  - Given a SOLID geometry, we don't need to draw polygons that face away from the viewer. So we can eliminate (cull) back-facing polygons in the drawing process
  - Can be used with ray casting and Z-Buffer algorithms

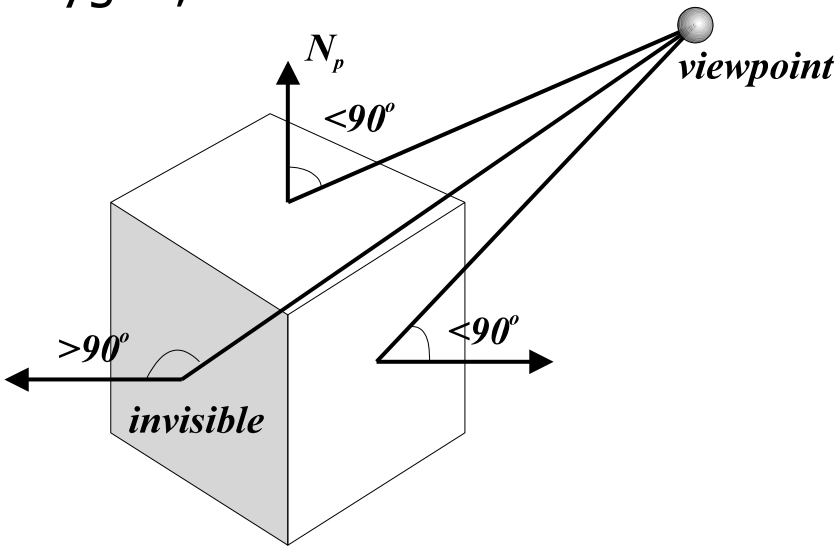
- How can we test for this?

Back Face Culling

- What is a normal vector? (see the lecture note in 2D/3D Math.)
- It is a vector pointing in an orthogonal direction of a plane.

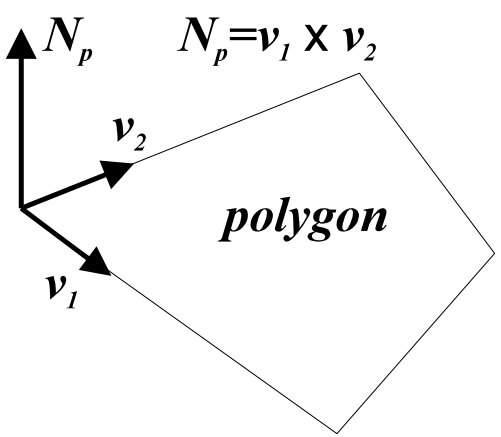


- A polygon is front facing when θ, the angle between the viewer and the normal to the polygon, lies between -90^{o} to $+ 90^{o}$ or $\cos \theta \geq 0$



Back Face Culling

(right hand rule)

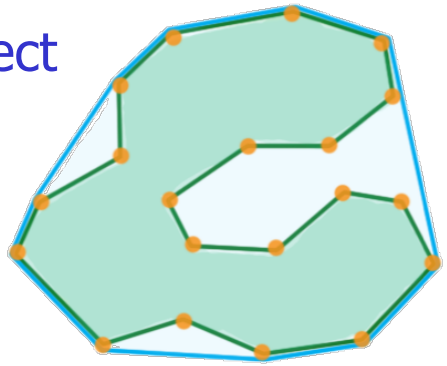


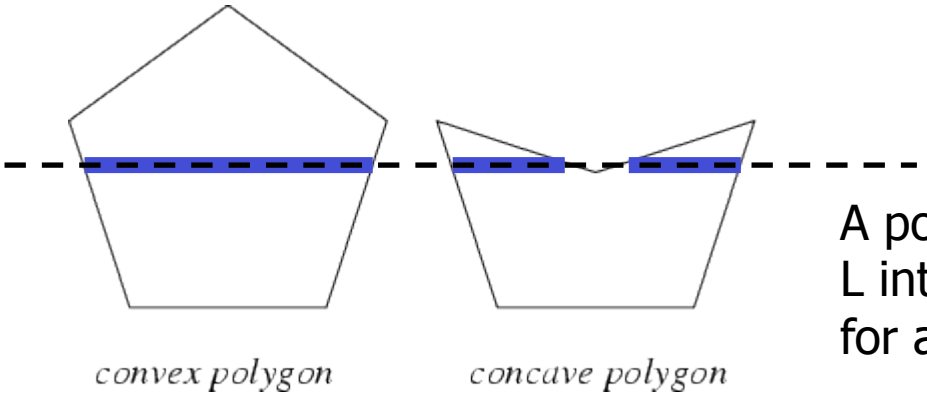
- Use dot product to test for visibility.
- Visibility test: $N_p \cdot V > 0$
  where N_p is the polygon normal
  V is the line of sight (viewing) vector
- How to calculate polygon normal?

- A polygon is front facing when θ, the angle between the viewer and the normal to the polygon, lies between -90^o to $+ 90^o$ or $\cos \theta \geq 0$
- All vertices of polygons must be listed in the same order (either clockwise or anticlockwise)

Back Face Culling

- Eye space is the most convenient space in which to 'cull' polygon. Why?
- Remove all polygons that face away from the viewer
- For a scene that consists of only one single closed convex object, it solves the hidden surface problem *completely*.
- Not sufficient for scenes with multiple objects
- Cannot remove all unnecessary polygons of a concave object
- In most cases, back face culling is a preprocessing step to reduce the number of polygons for Z-buffer or ray casting



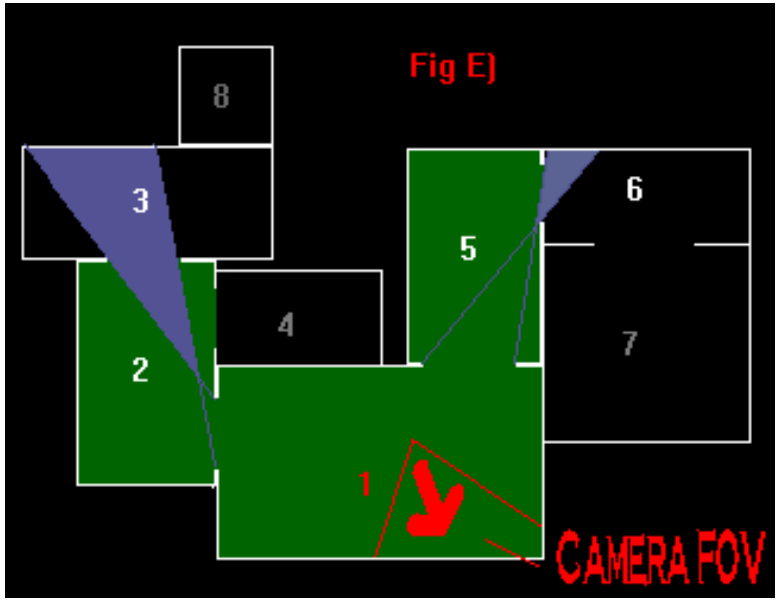


A polygon P is convex iff P and L intersects with one segment for all lines L in space

(4) Potentially Visible Set (PVS)

- It is an acceleration technique
- It is a data structure (usually pre-computed) that defines from-region visibility, i.e., a set of objects that may be visible from a region (no matter where you put the virtual camera in the region).
- Often used in 3D games
- Details:
  - Divide the scene into cells (regions)
  - Precompute the objects that may be visible in each cell

PVS for region 1:



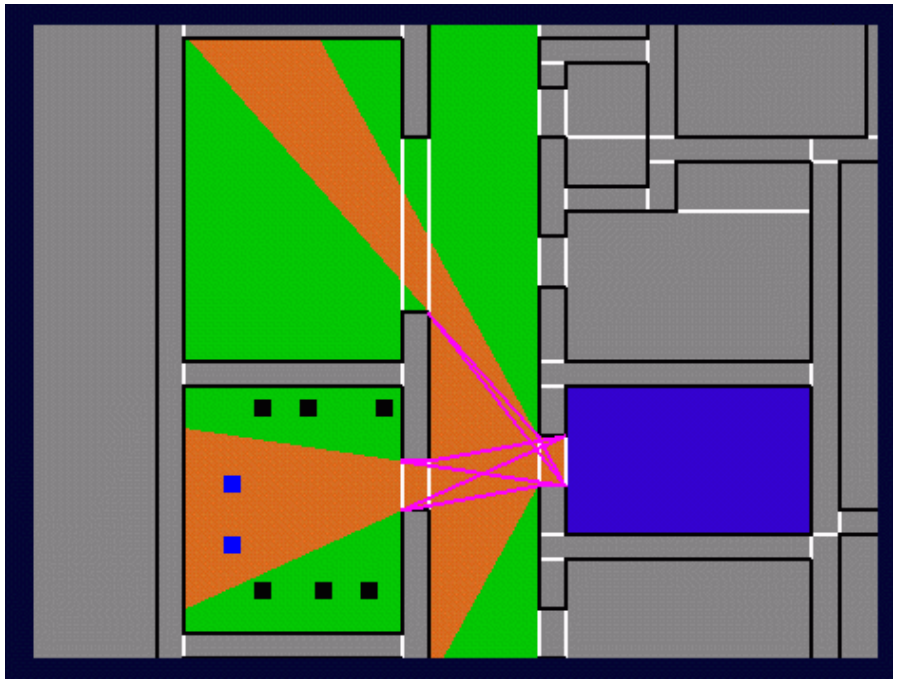
(4) Potentially Visible Set (PVS)

Advantages?

- Speed up!!! Avoid the rendering of large amount of 3D objects

Disadvantages?

- Need additional storage (memory)
- Preprocessing time may be long
- Dynamic scene (?)



Hidden Surface Removal in OGL

- Three related modules:
  1. View Frustum Clipping
     - remove objects outside camera view
     - see previous lecture module "camera, projection, and clipping"
  2. Back-face Culling
  3. Depth Buffering
- Ray casting can be implemented on your own (e.g., using shader programming).

Hidden Surface Removal in OGL

Note on Back-face Culling:

- This is done before the "rasterization" step
- Three related OGL functions:
  - glEnable (GL_CULL_FACE) ; # default: disabled

    OR

    glDisable (GL_CULL_FACE) ;
  - glFrontFace (mode) ; # GL_CCW or GL_CW

    - specifiy which side is the front face: counter-clockwise or clockwise
  - glCullFace (mode) ; # default: GL_BACK

    - cull which side? front face or back face: GL_FRONT or GL_BACK, etc.

Hidden Surface Removal in OGL

Note on Depth Buffering:

- You need to request the depth buffer when you create the window, e.g. glutInitDisplayMode (… | GL_DEPTH).
- Remember to clear the depth buffer (with the max. depth (default)) before you render the scene again, e.g. glClear and glClearDepth.
- Need to enable the depth test, glEnable (GL_DEPTH_TEST).
- glDepthFunc defines the test condition for pixel fragment.
  - Default and normal argument is GL_LESS, meaning that incoming pixel fragment with smaller Z value should pass. But we can play tricks using other settings.

Let's check the sample code in assignment 1!

Summary

- Classification of hidden surface algorithms
- Understand ray casting algorithms
- Understand Z-buffer
- Back face culling and PVS *accelerate*
- How OGL handles hidden surface removal