Hidden Surface Removal

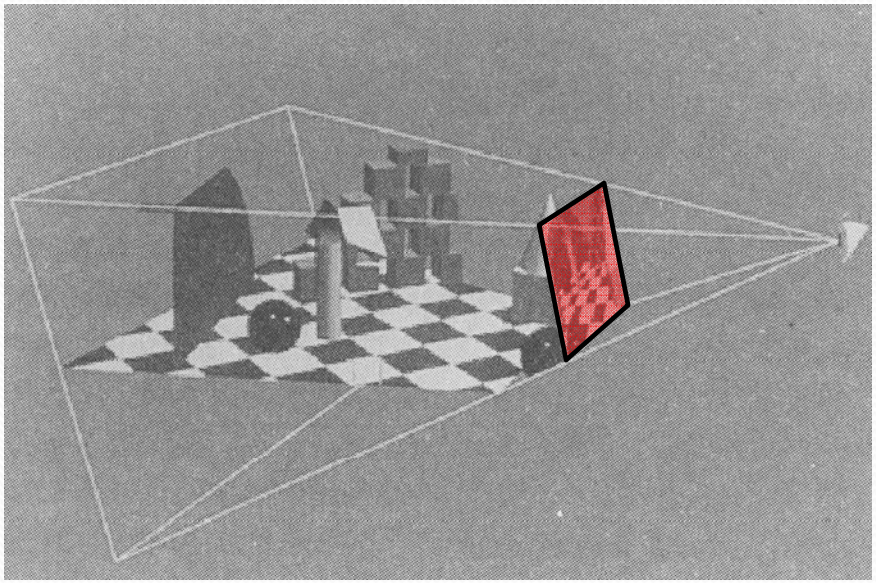
CS3241: Computer Graphics

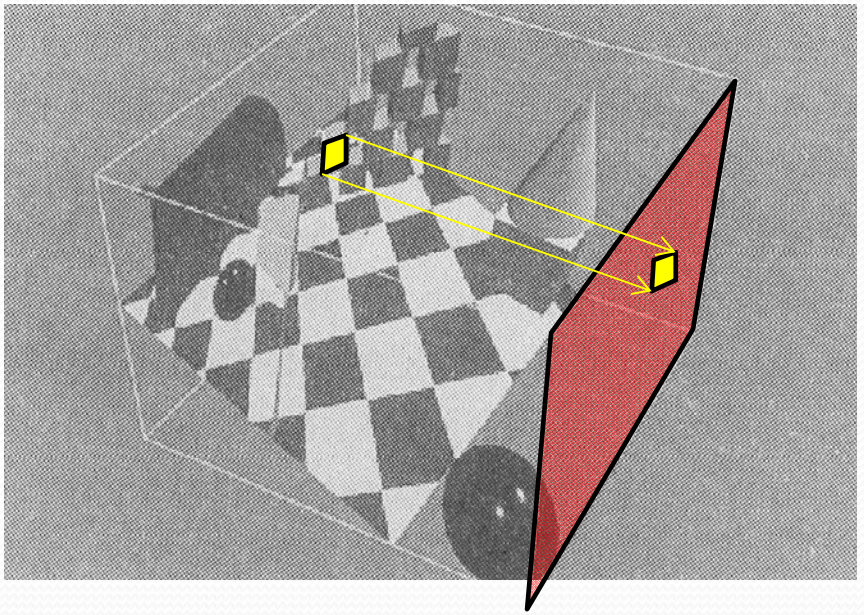
Where are we?

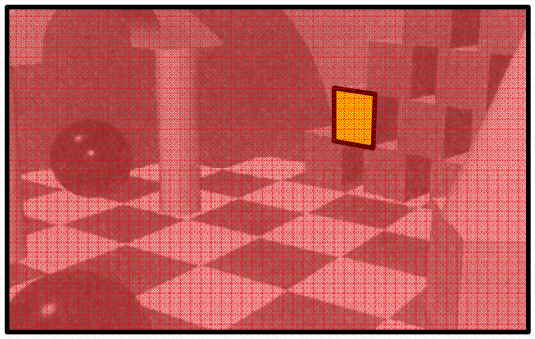
- All the objects are set up in the scene
  - And they are polygonal models
- A camera (view point) is set
- After all the transformations (including the **perspective transformation**)
  - So we now have many many polygons on the screen
  - Transform all the polygon vertices into the screen coordinates

Keeping the z Values

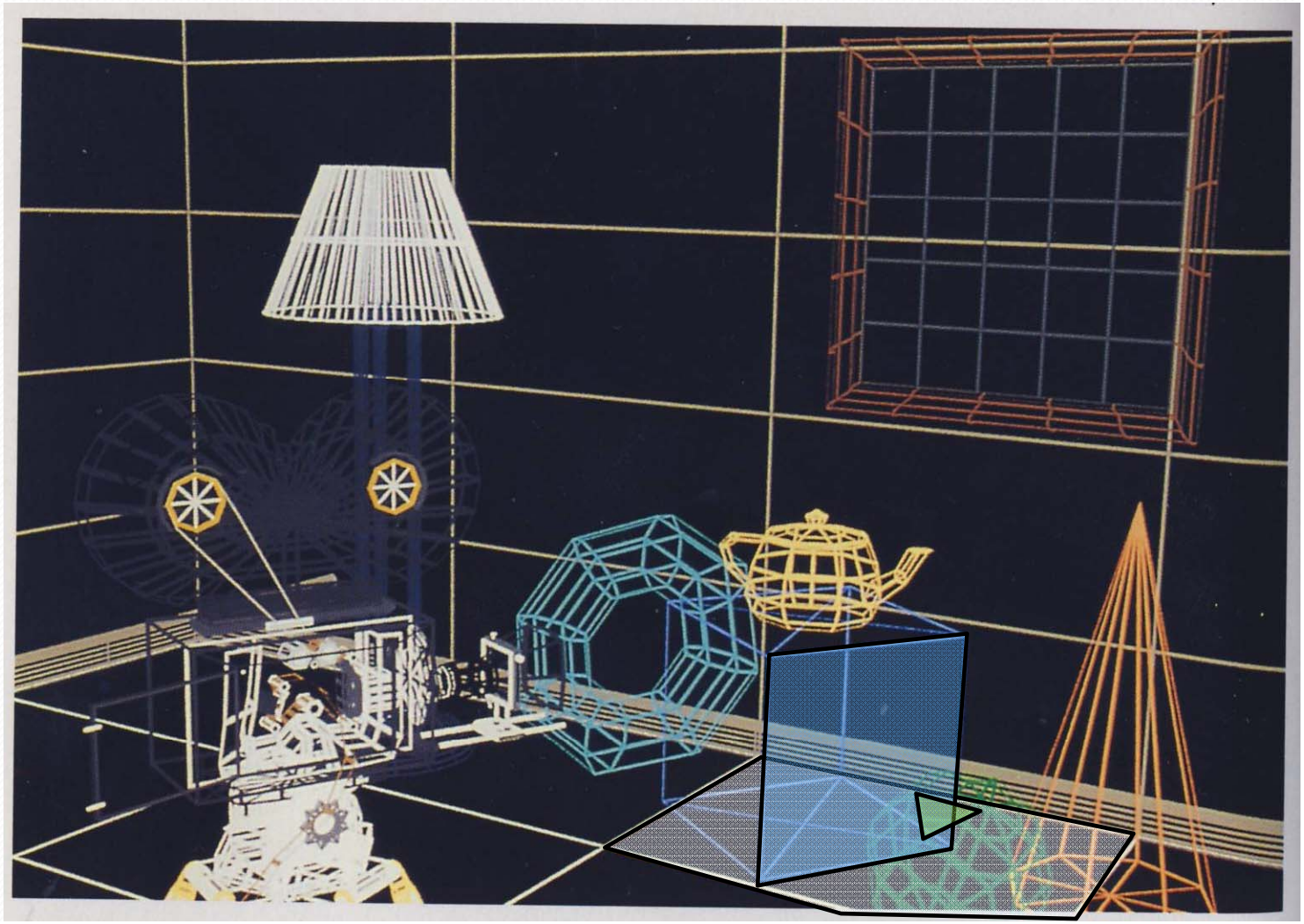
After Perspective Transformation





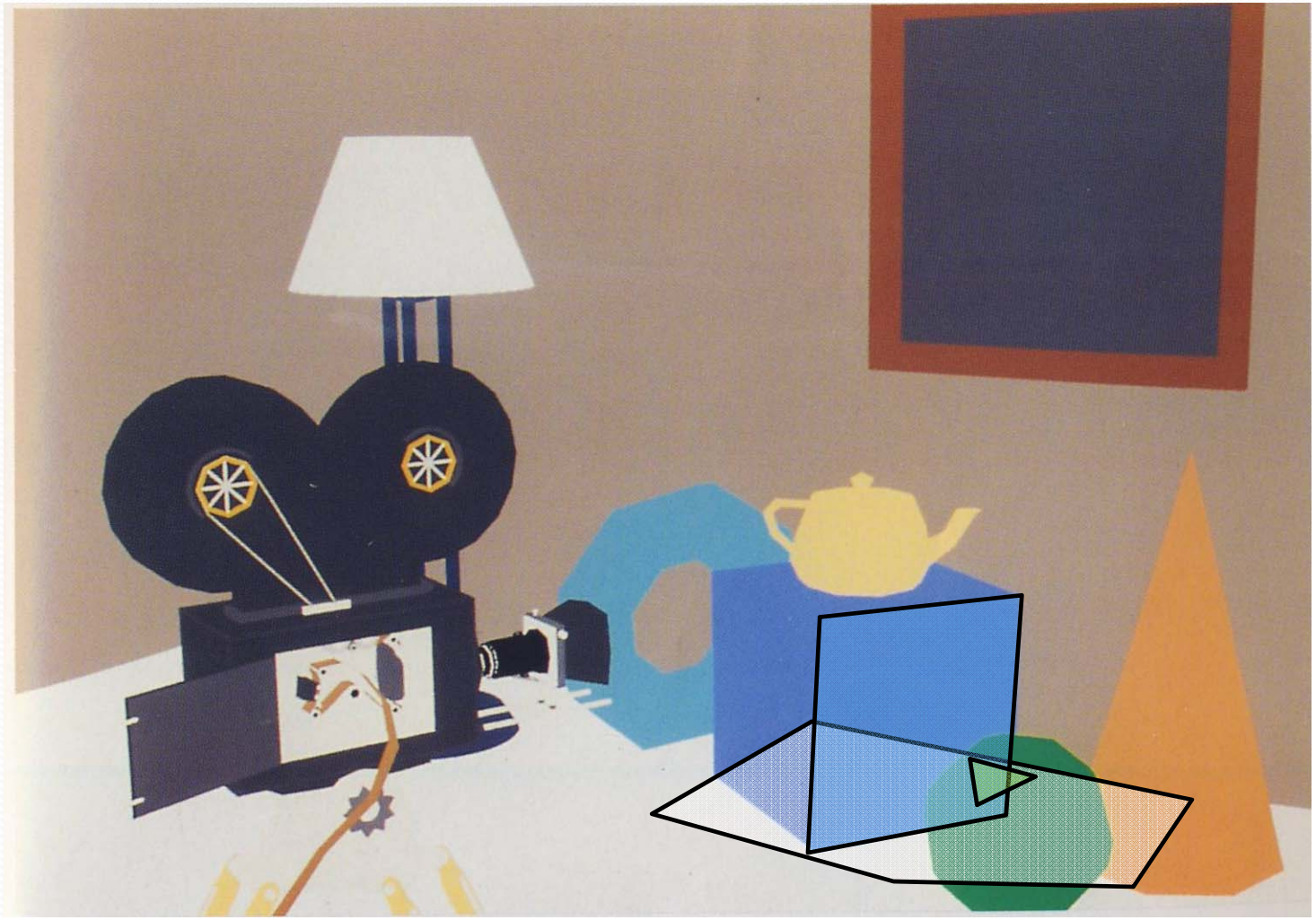


Transformation/Viewing

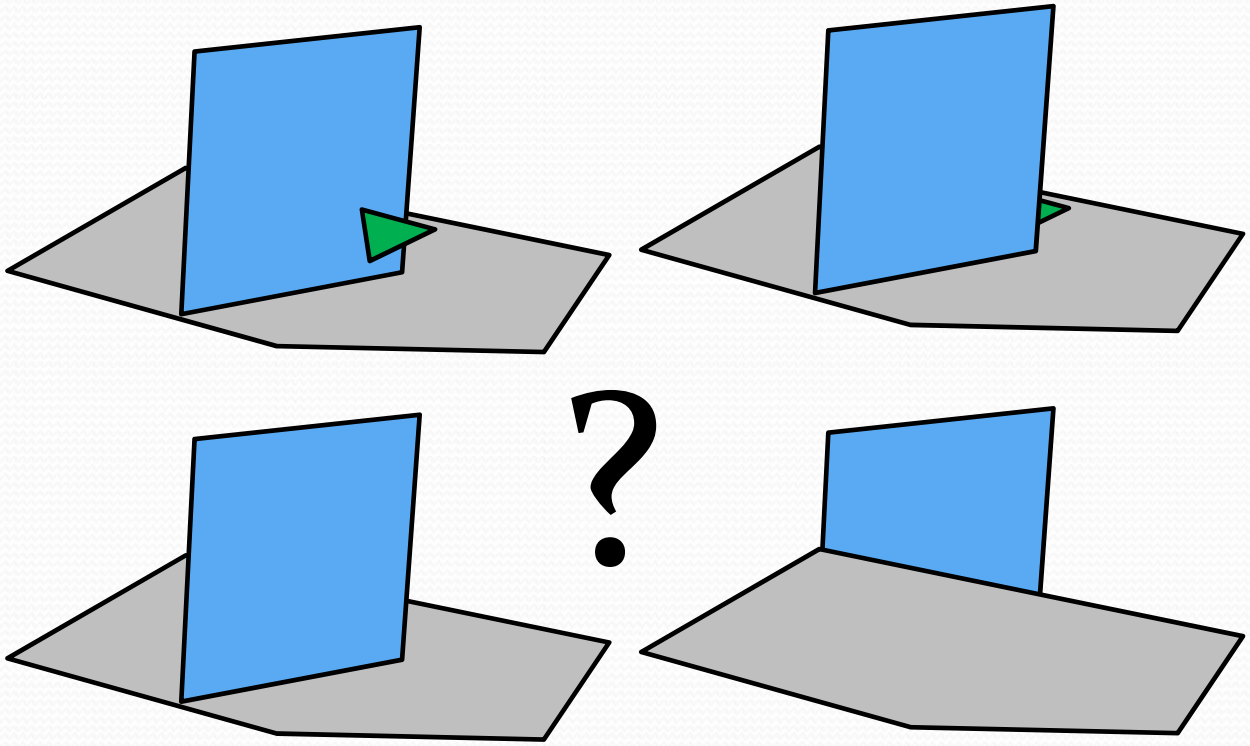


Scan Convert Algorithm

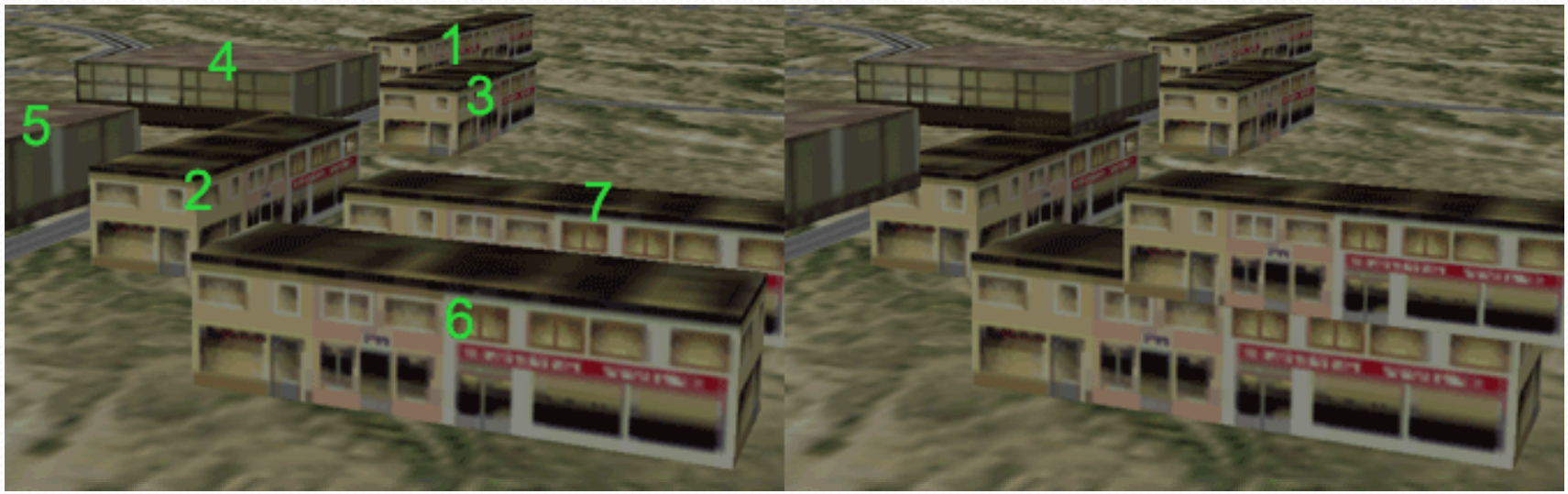
- But how do we know the order?



Drawing Order?

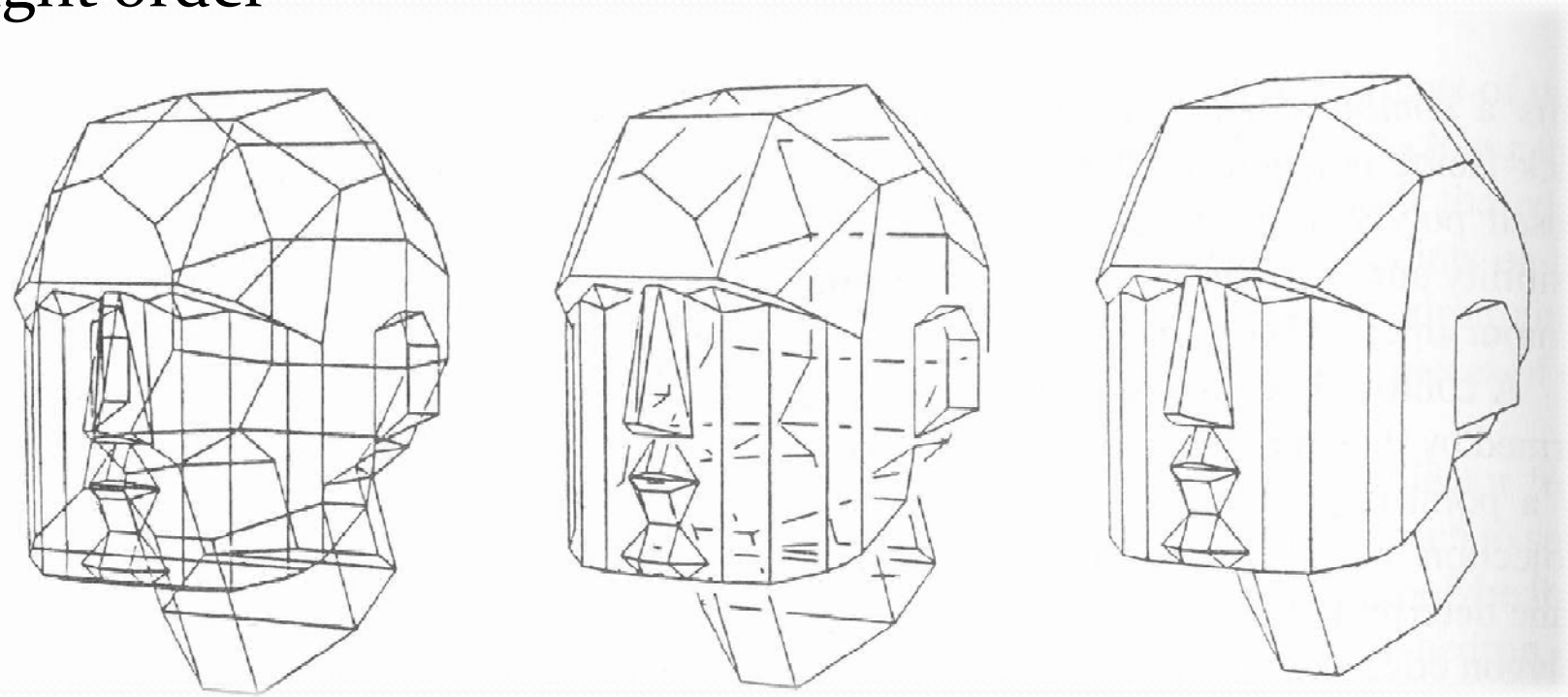


Wrong Drawing Order



Hidden Surface Removal (HSR)

- Managing the polygons so that they are drawn in the right order



Two Main Approaches

I. Control the drawing order of objects
- ***Object Precision Algorithm***
- Sort the order of polygons according to "depth"
- Draw polygons from furthest to nearest
- View dependent
  - The depth order changes when the viewpoint changes

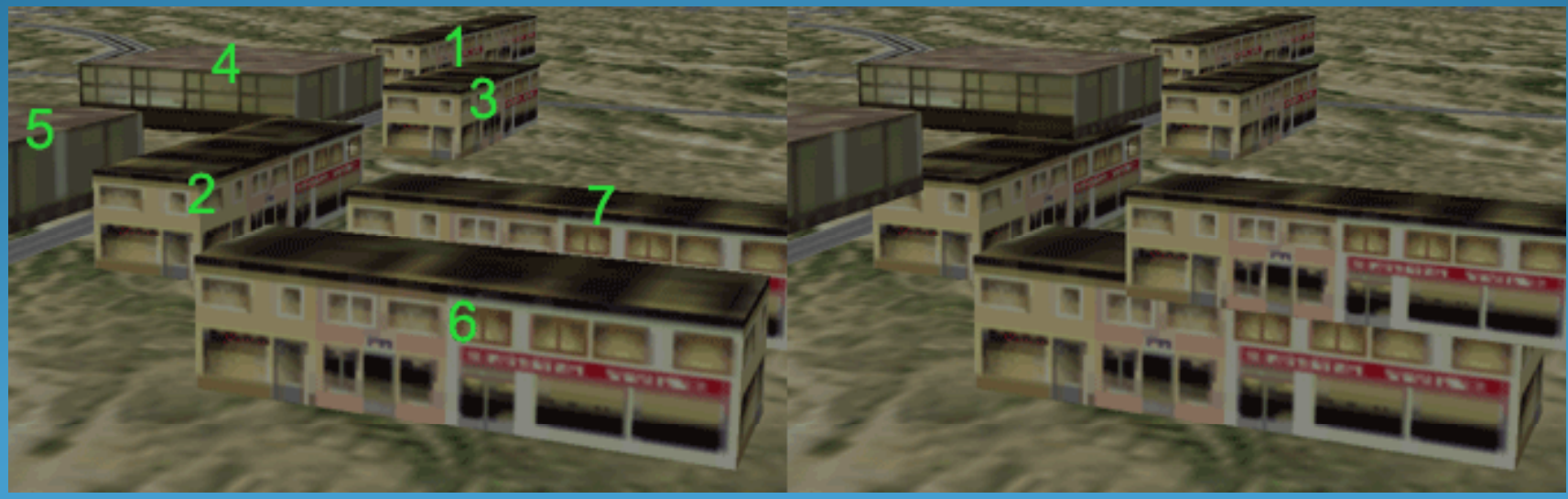
II. Draw objects in a random order
- ***Image Precision Algorithm***
- Then for each pixel drawn by a new object
  - If there is already some object drawn, decide if the new object should overwrite it or not

Various HSR Algorithms

- Type 1: (the "orderly" approach)
  - Depth Sorting
    - Weiler-Atherton Algorithm
  - Binary Space-Partitioning (BSP) Tree
- Type 2: (the "non-orderly" approach)
  - Z-buffer
  - Warnock's Algorithm
  - Ray-casting (in the lecture of ray tracing)
- …. etc.
- All of them are performed **after** perspective transformation
  - Except Ray-casting

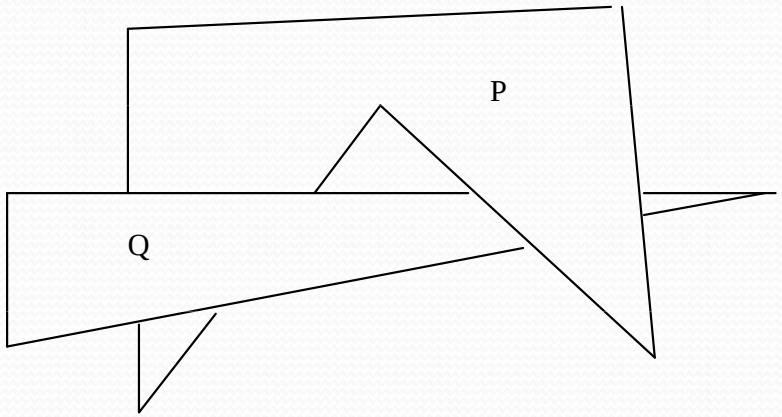
Type I

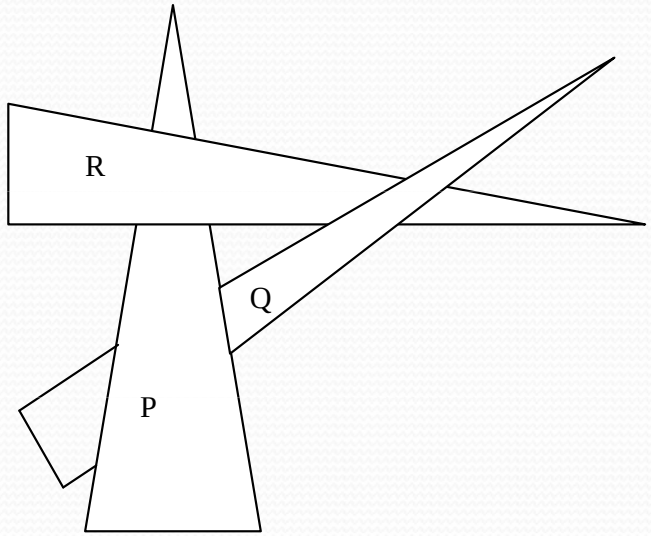
(Object-precision Algorithm)



Type 1: Controlling Drawing Order

- Goal: For a bunch of polygons, we sort them according to the "depth" and draw the "deepest" first
- However, sometime there is no possible correct order:

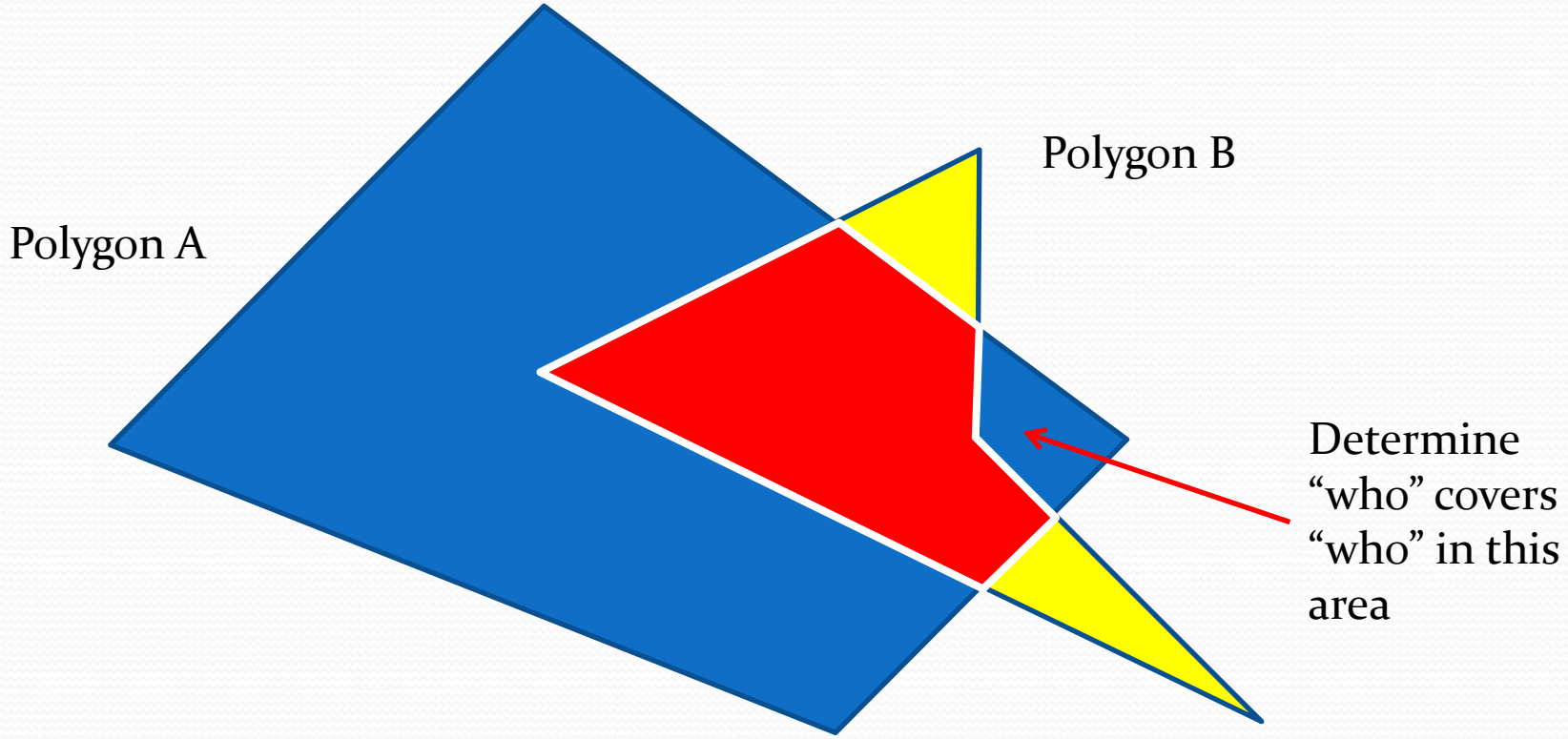




- So, we have to break the polygons

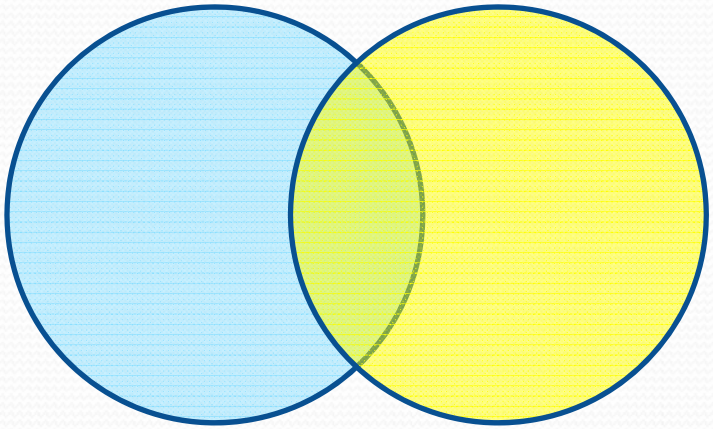
Type 1a: Depth Sort

- Breaking two polygons

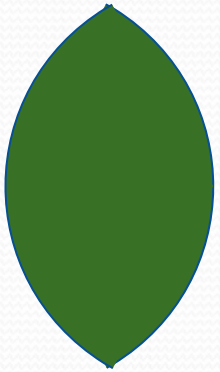


Type 1a:Depth Sort

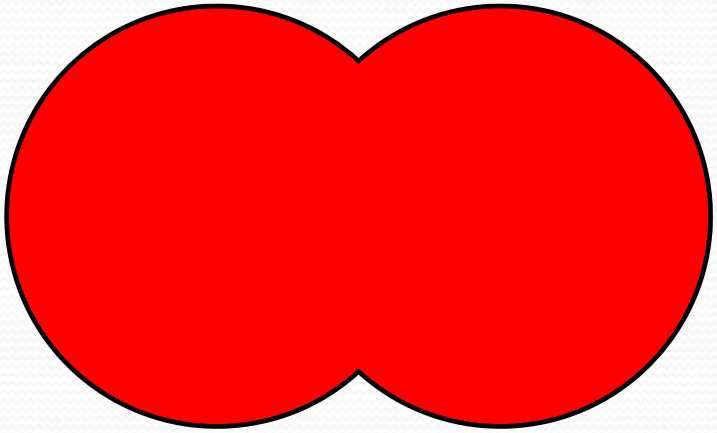
- Weiler-Atherton Algorithm
  - Computing intersections of polygons
  - A generic polygon clipping algorithm (detail omitted)
  - For two polygons and B, we can compute
    - the union of A and B
    - the intersection of A and B
    - A minus B
    - B minus A



Set A and Set B



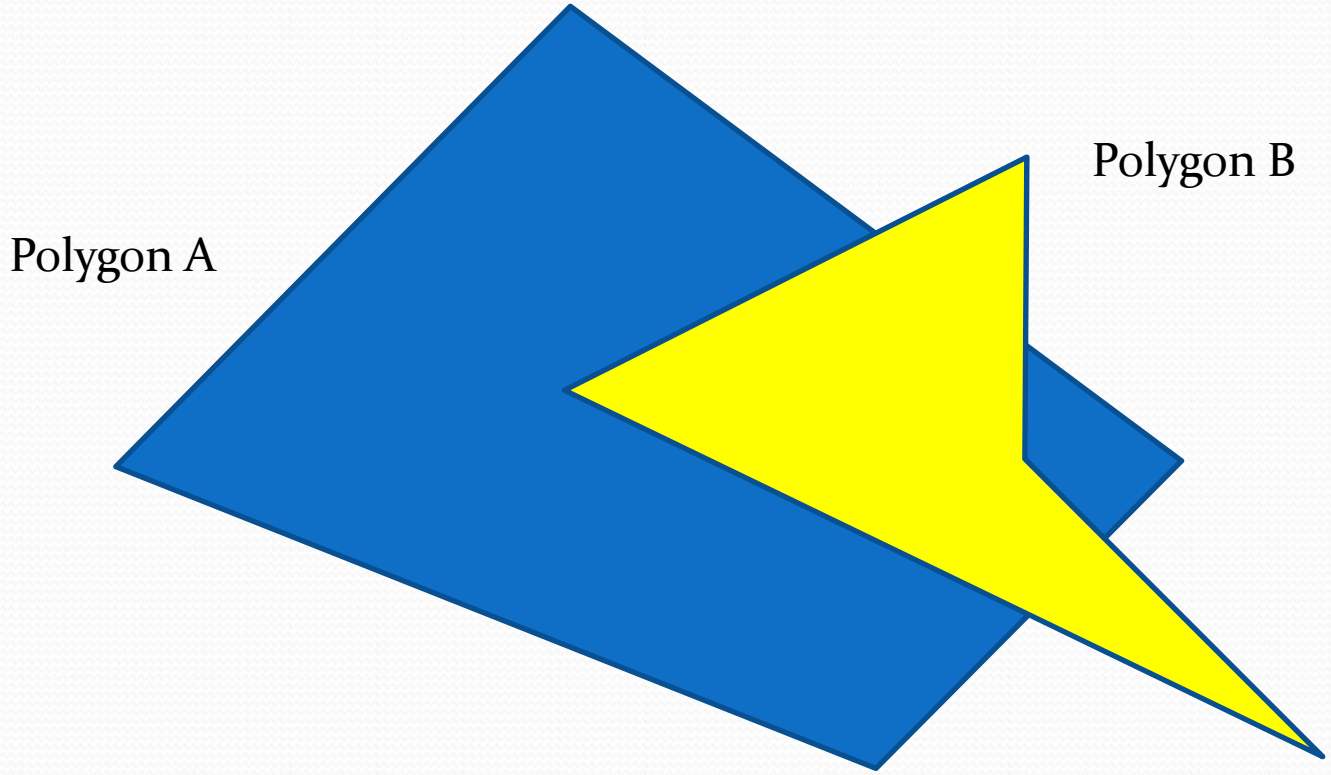
Intersection



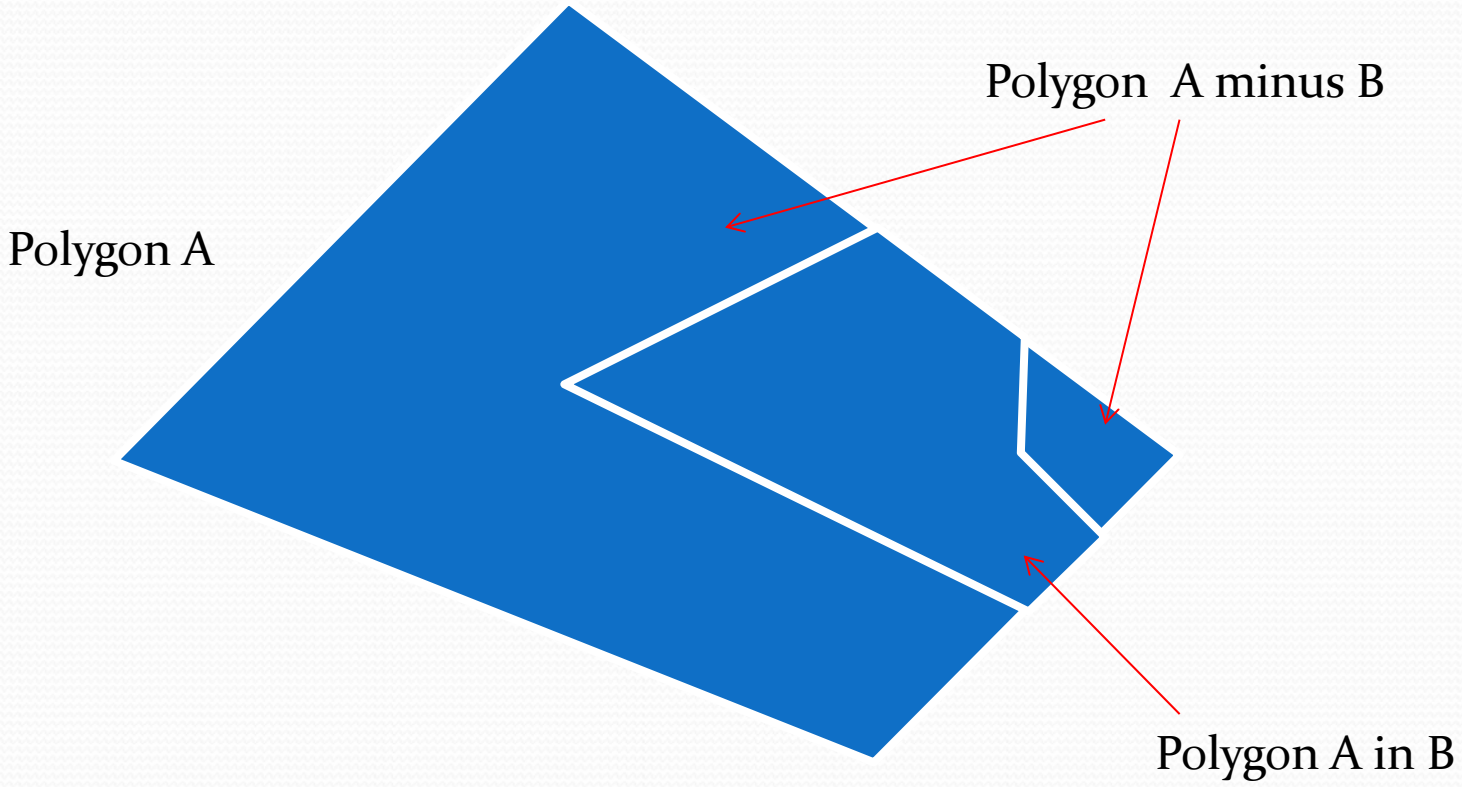
Union

A - B

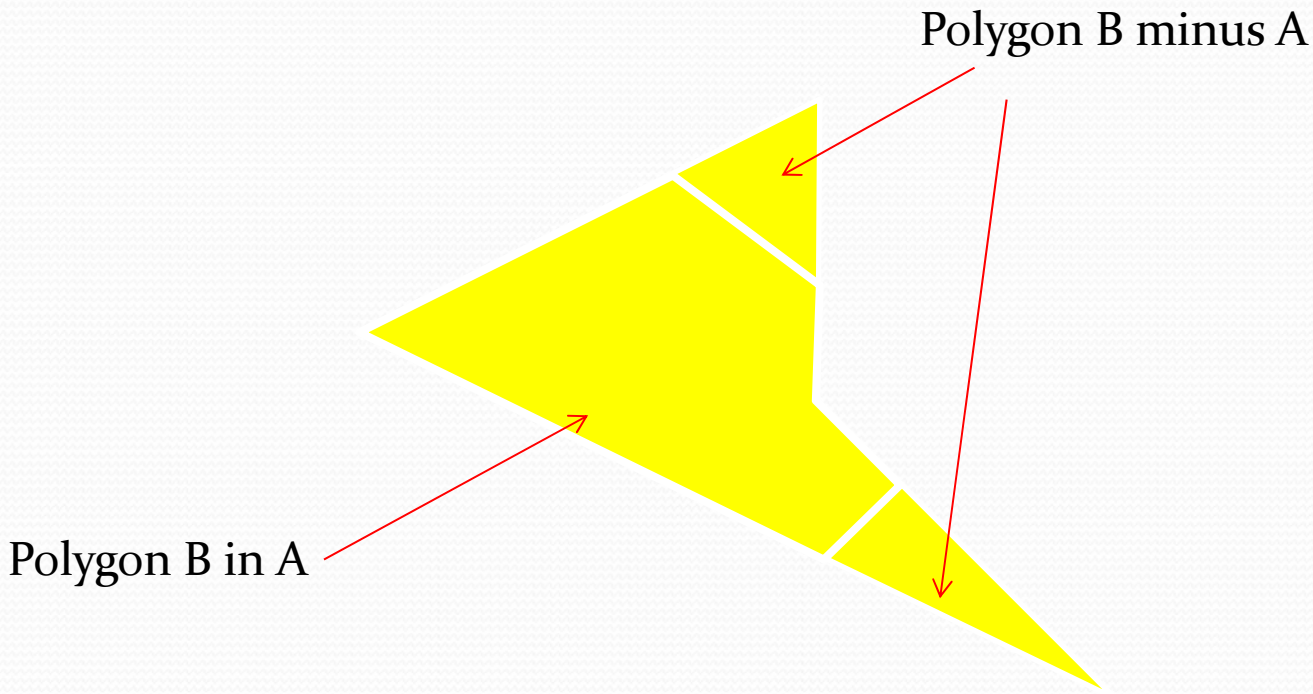
Type 1a: Depth Sort



Type 1a: Depth Sort

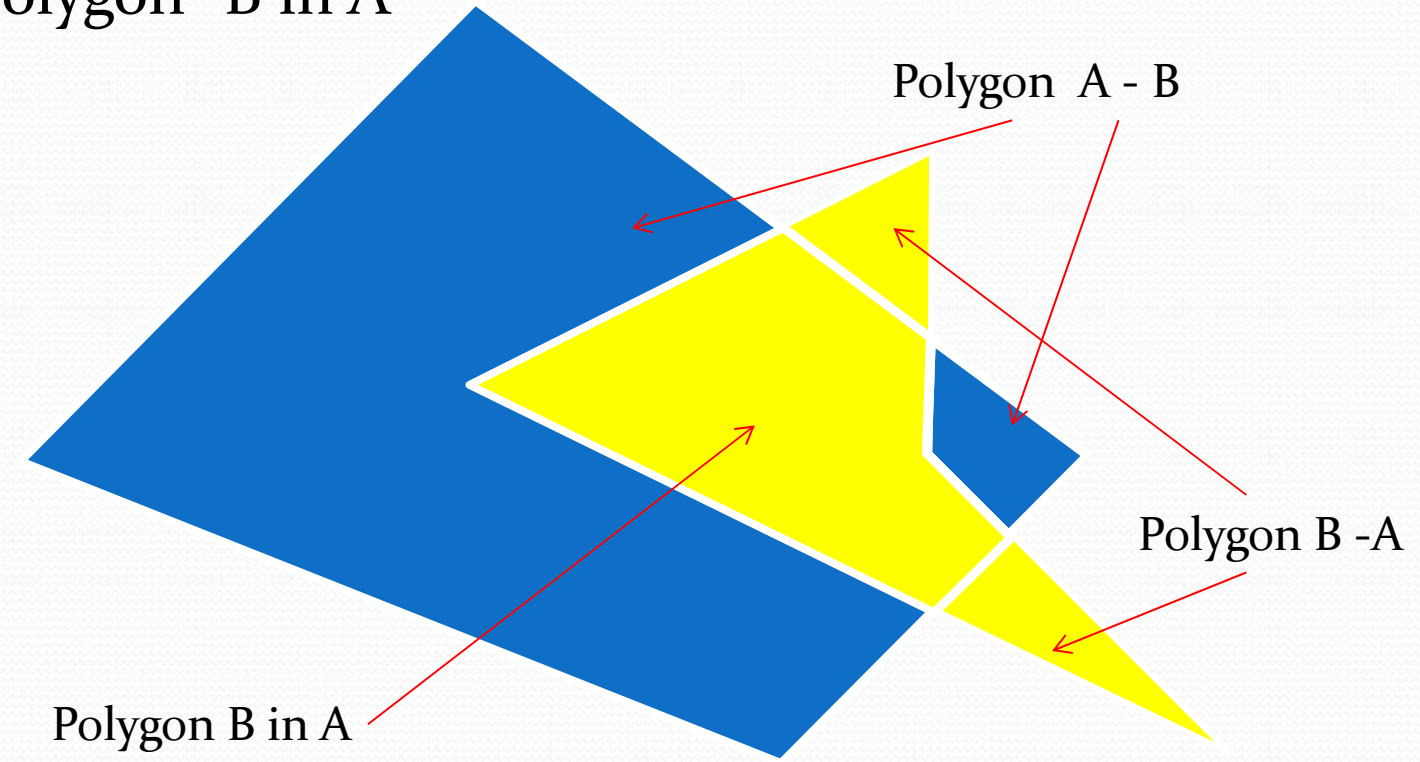


Type 1a: Depth Sort



Type 1a: Depth Sort

- Drawing order
  1. Polygons “A – B” and “B - A”
  2. Polygon “B in A”

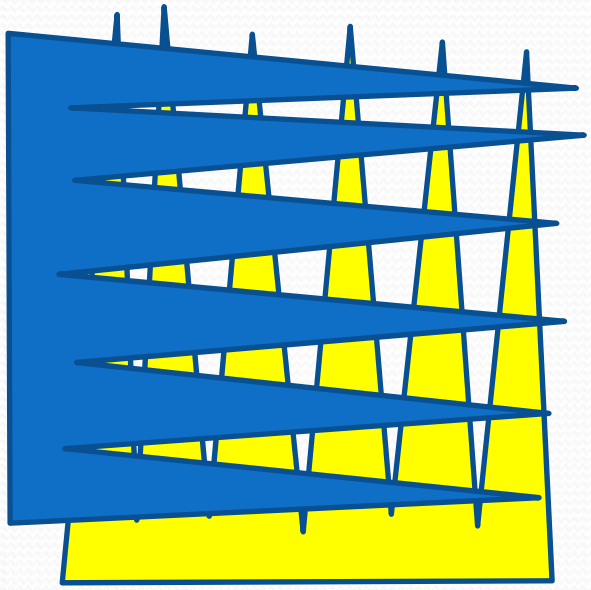


Type 1a: Depth Sorting

- Drawing order:
  - Determine which one to draw first
    - Polygon “A in B” or Polygon “B in A”
      - In this case, draw “A in B” first then draw “B in A” if polygon B is **transparent**
      - If B is not transparent, we can ignore or skip drawing “A in B” directly
  - Do not need to care about “A-B” or “B-A”
    - If they do not intersect other polygons in the screen

Type 1a: Depth Sorting

- **Advantages**
  - Handle transparency
  - Especially good if
    - The order is predictable and there is no need for polygon clipping
- **Disadvantages**
  - Need special care for penetrating polygons
  - Computational expensive
    - n polygons produce ${}_nC_2$ intersections
      - i.e. $O(n^2)$
    - And for each pair of polygon, we could produce $O(m^2)$ if each polygon has m edges



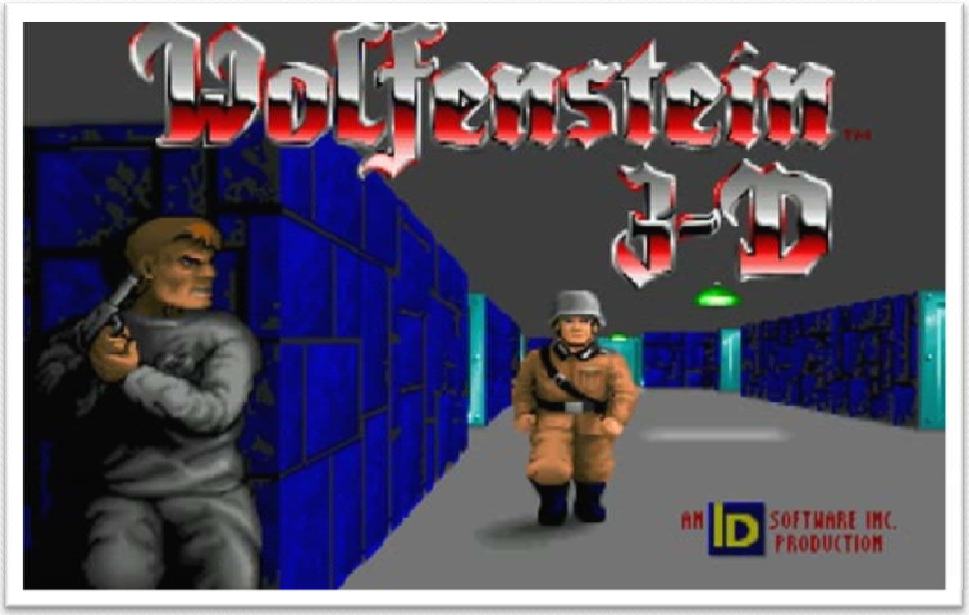
Binary Space Partitioning

Type Ib: Binary Space Partitioning

- An industrial standard for real time 3D games such as FPS
  - Doom, Counterstrike, Quake, etc…
- Input:
  - An environment modeled by polygons

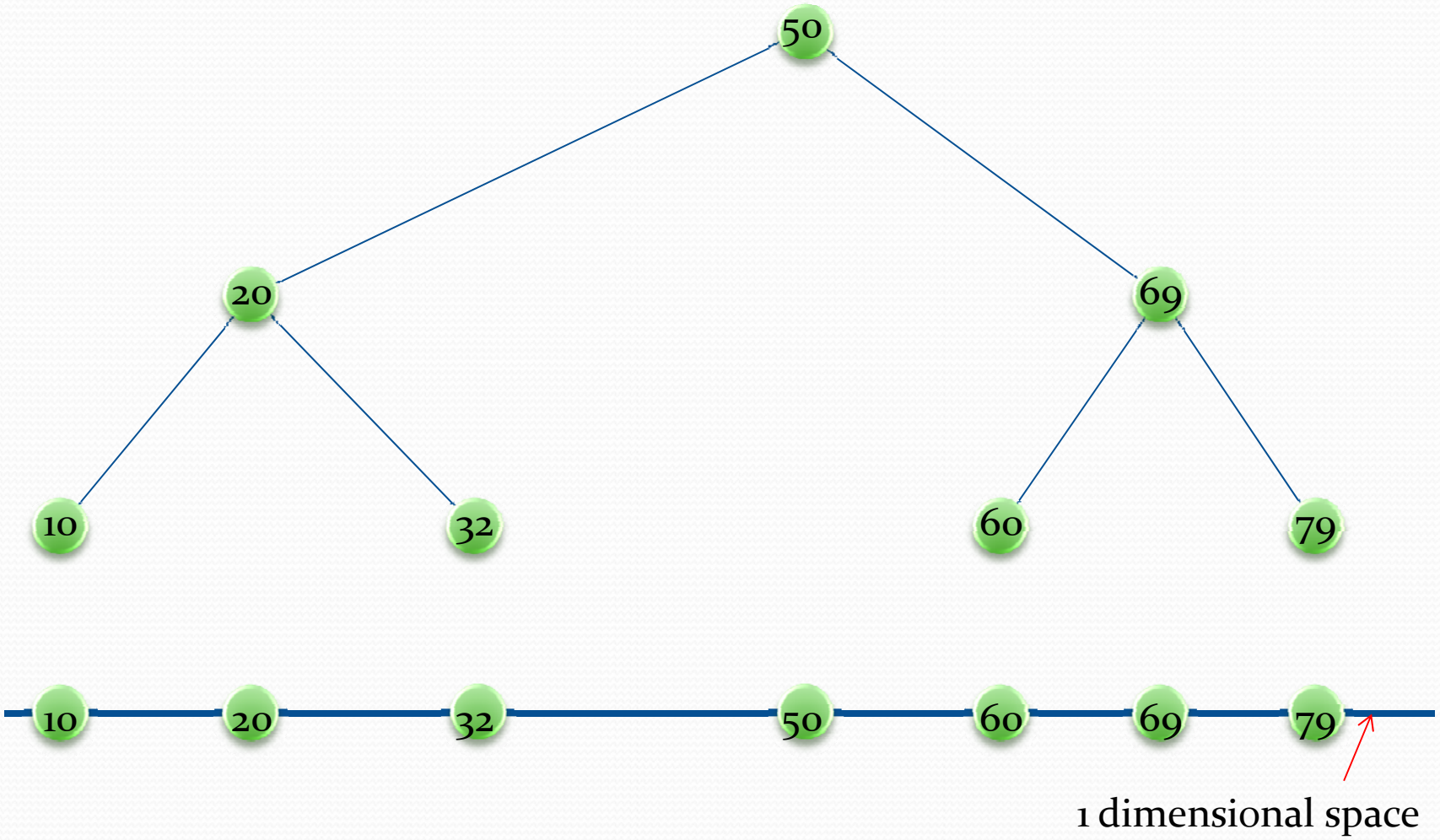
- Preprocess the environment and produce a BSP tree
- Output
  - Base on the BSP tree, output a drawing order from the furthest away to the nearest polygon

The First FPS

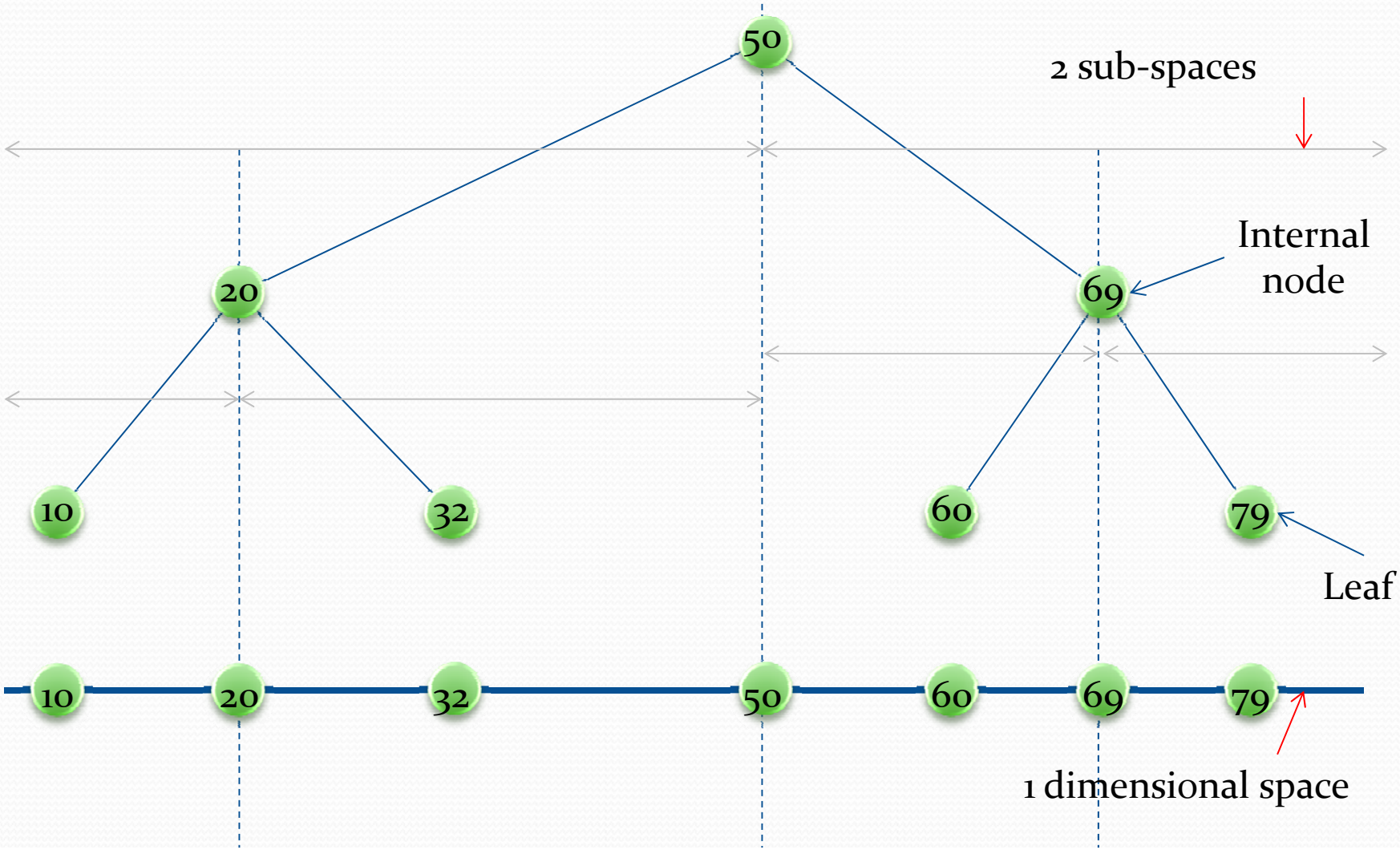


- There are free versions on web and iPad/iPhone
  - The iPad version is totally free on the Japanese iStore

Recap: Binary Search Tree



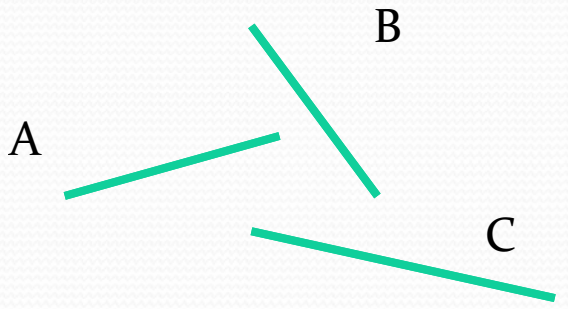
Recap: Binary Search Tree



Type 1b: Binary Space Partitioning

- Idea: Subdivide the entire space by a binary tree
  - Each internal node is a division of a partition/space
  - Each leaf is a part of the space with only **one** polygon

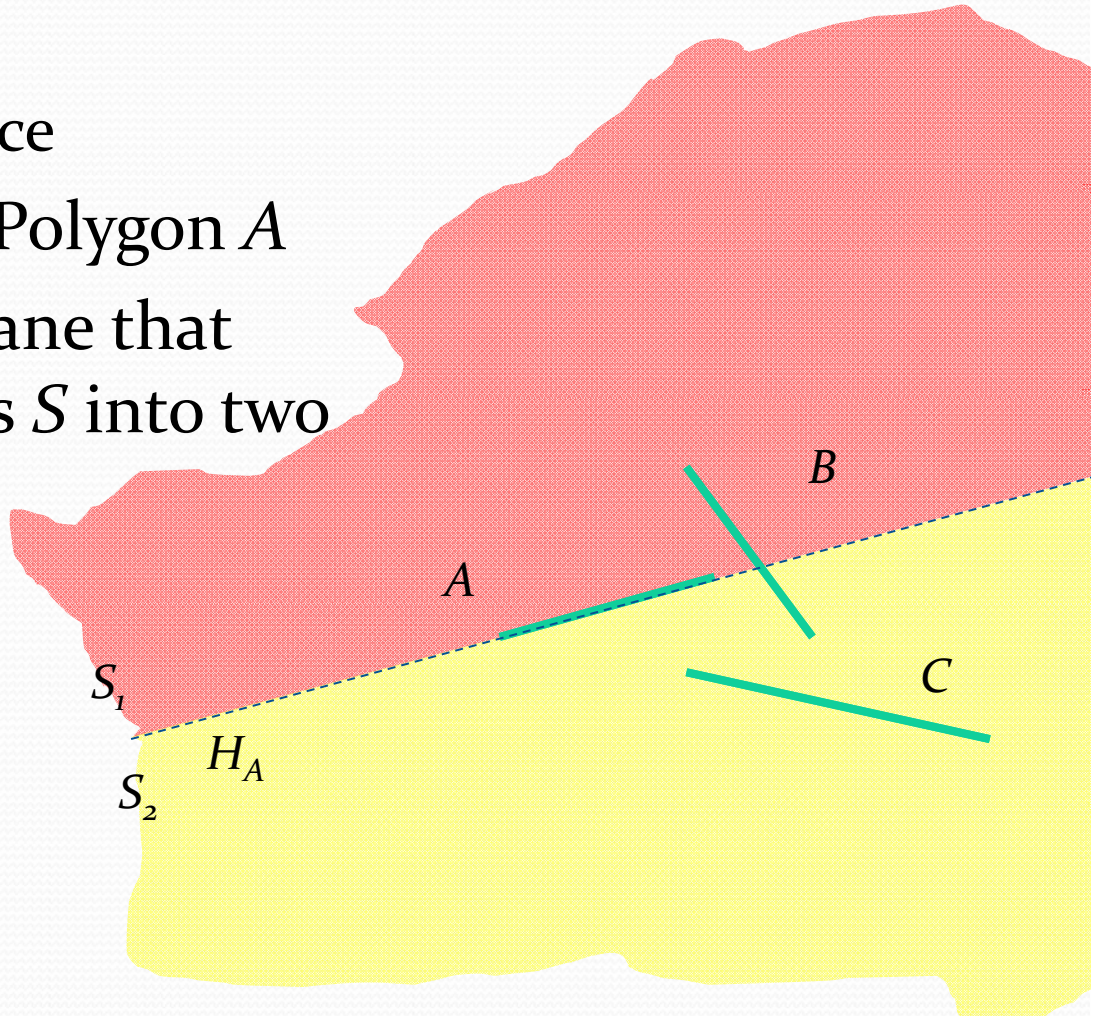
- Divide into two steps
  I. Preparation
  II. Rendering



(Use a bird's eye view for illustration)

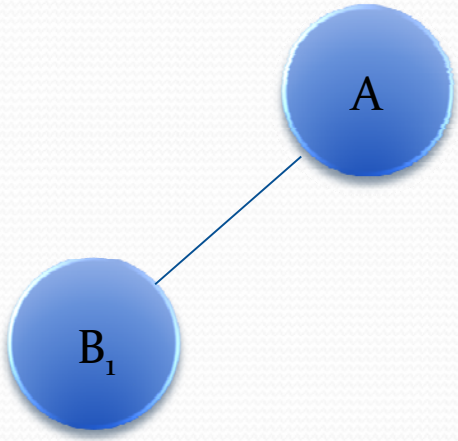
Type 1b: BSP-tree Preparation

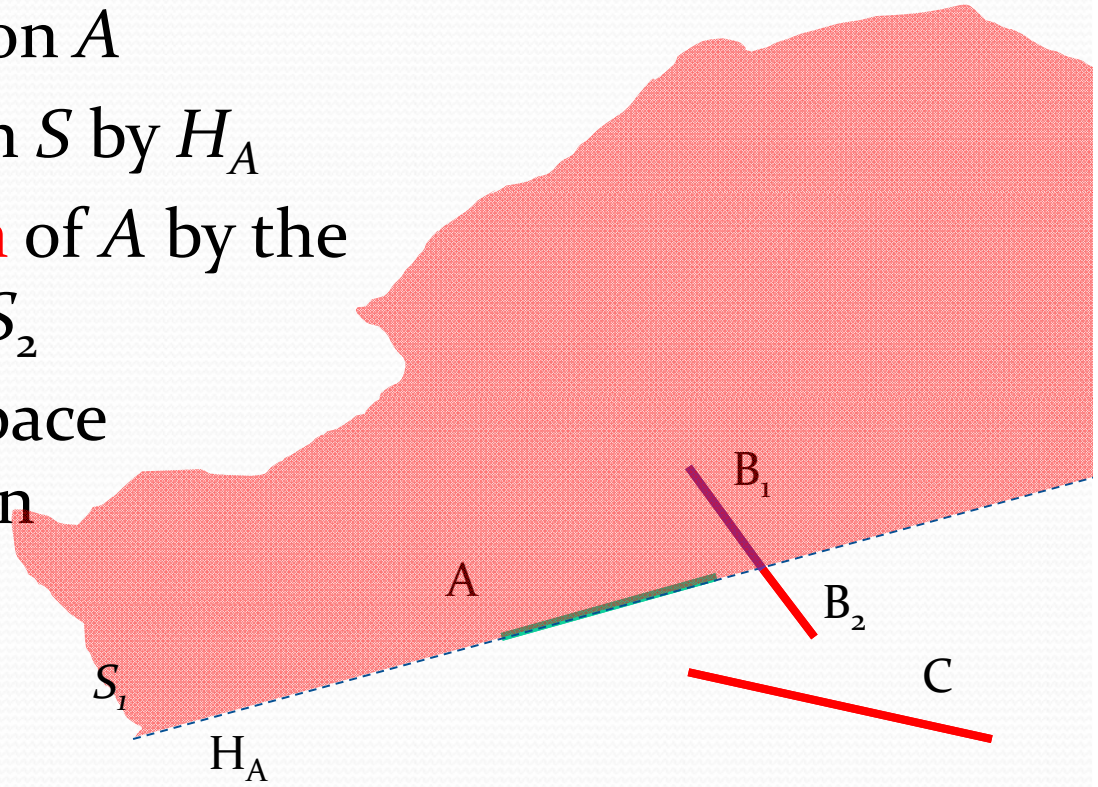
- Initialization
  - Let S = the entire space
- Pick any polygon, say Polygon A
- Let H_A be the hyperplane that contains A and divides S into two subspace S_1 and S_2



Type 1b: BSP-tree

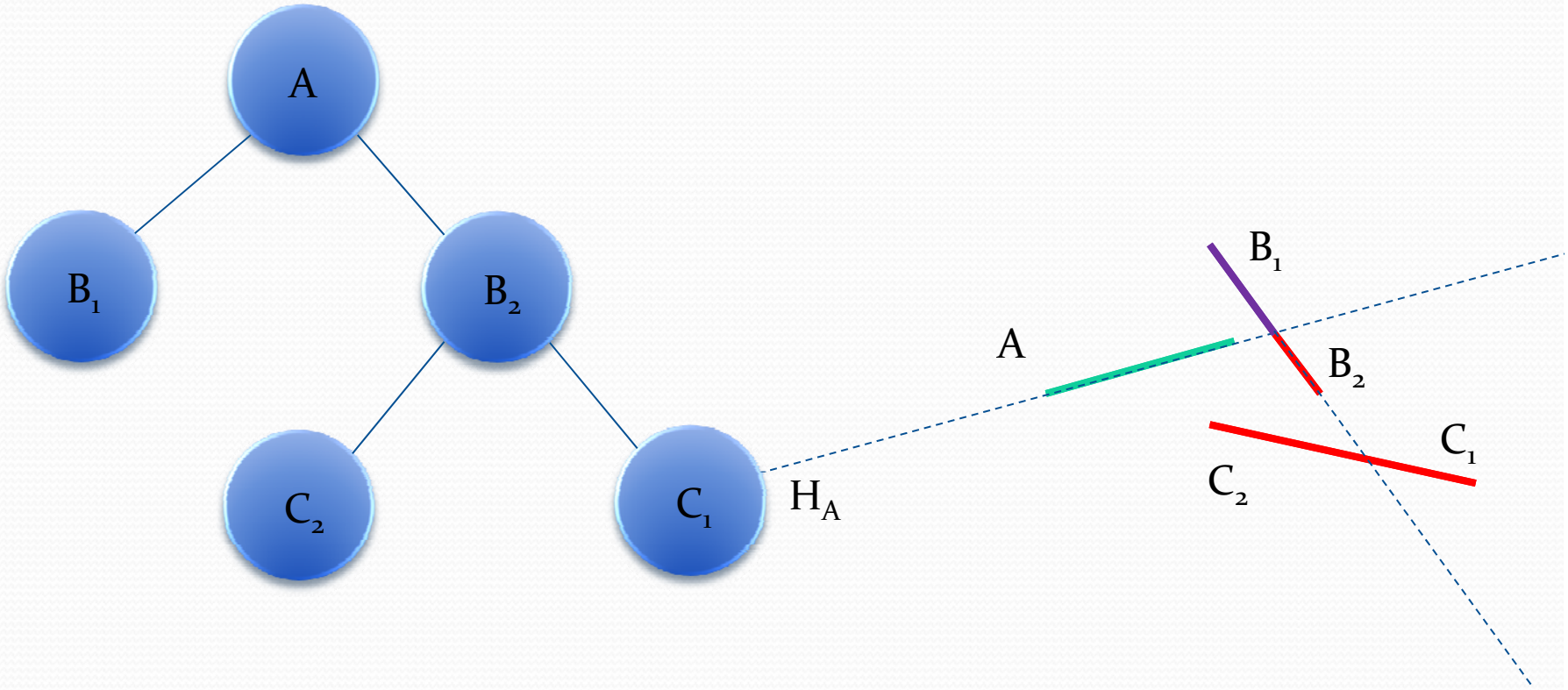
- Build a node by polygon A
- Break every polygon in S by H_A
- Build the two children of A by the two subspaces S_1 and S_2
- Repeat until the subspace contains only 1 polygon





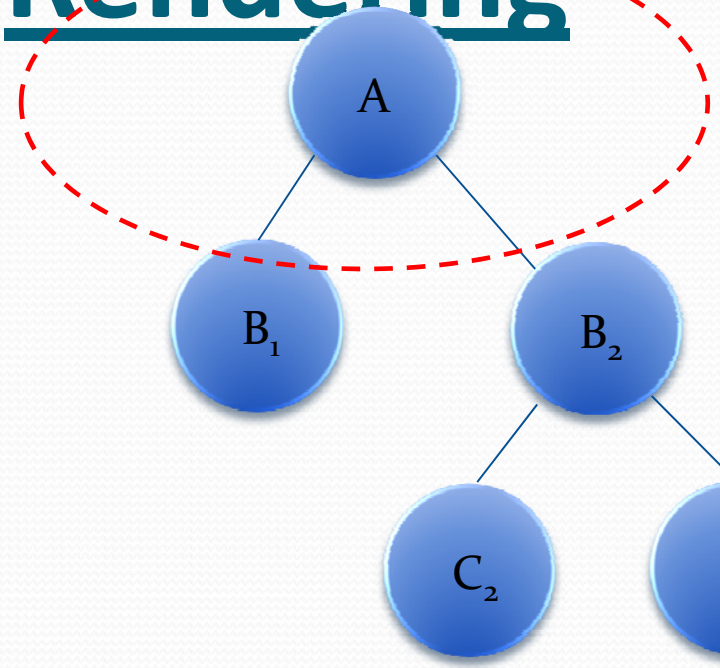
Type 1b: BSP-tree

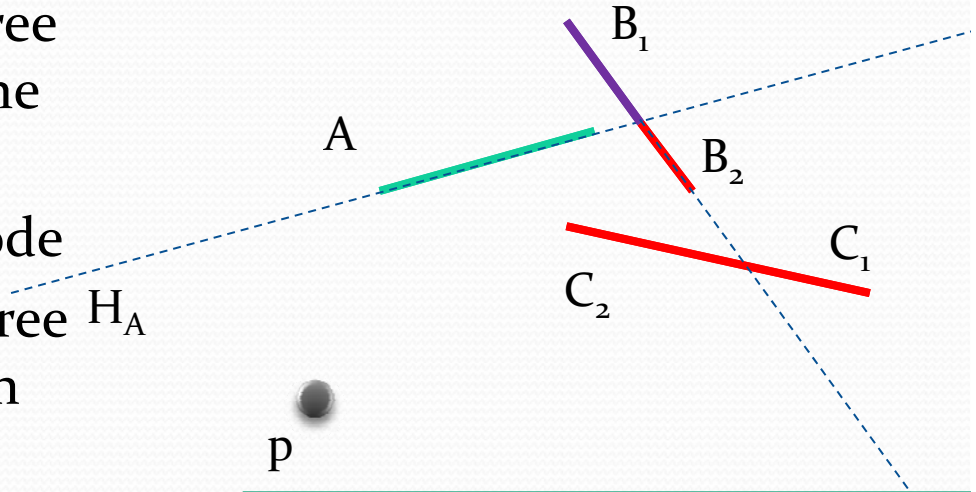
- Prepare the BSP-tree for rendering:



Type 1b: BSP-tree Rendering

- Depending on the viewpoint *p*
- Start from the root
  - For each node there is one polygon and two sub-spaces in the two children
    1. Recursively draw the sub-tree behind the polygon from the view point *p*
    2. Draw the polygon of the node
    3. Recursively, draw the sub-tree in front of the polygon from the view point *p*

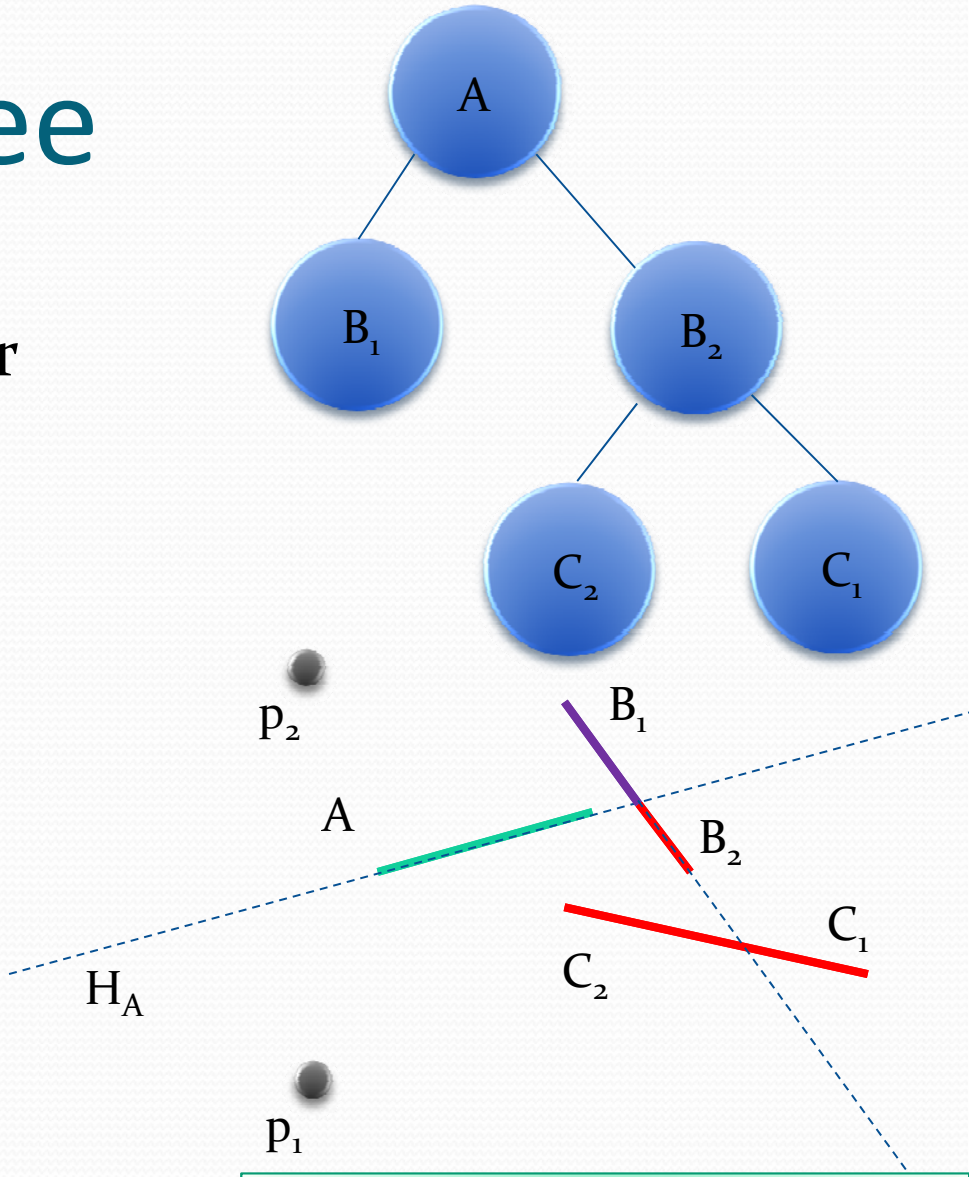




= in-order traversal of a BSP tree

Type 1b: BSP-tree

- For example, for the following viewpoints, their drawing order will be
  - p_1 : B_1, A, C_1, B_2 , C_2
  - p_2: C_1, B_2, C_2, A, B_1



Detailed exercises in tutorials

Type 1b: BSP-tree

- **Advantage**:
  - Once the tree is computed, the tree can handle all viewpoints, i.e. efficient
  - Handle transparency
  - Indeed, it's a standard format (.**BSP** files) to store the environment for many games
    - E.g. Quake, Half-life, Call of Duty, etc
- **Disadvantages**
  - Cannot handle moving/changing environments
  - Preprocessing time is long

Type II

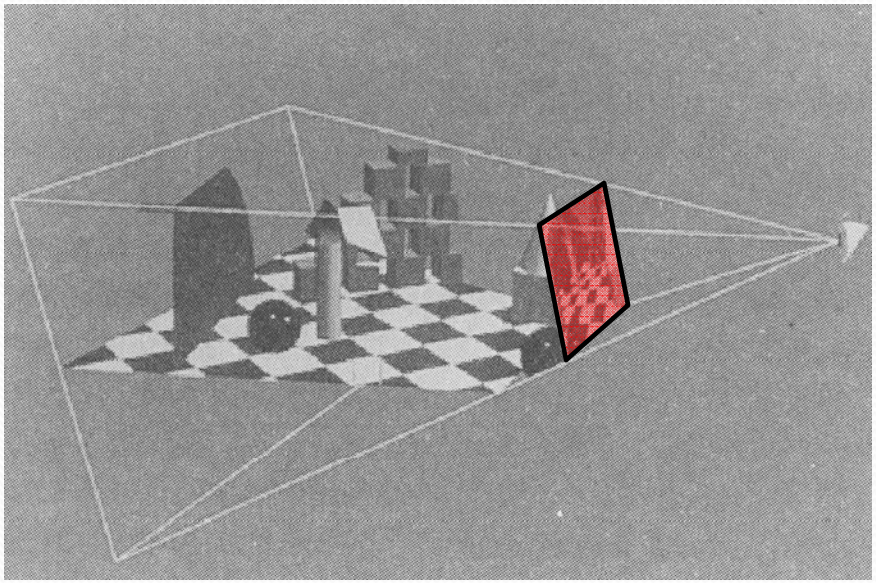
(Image-precision Algorithm)

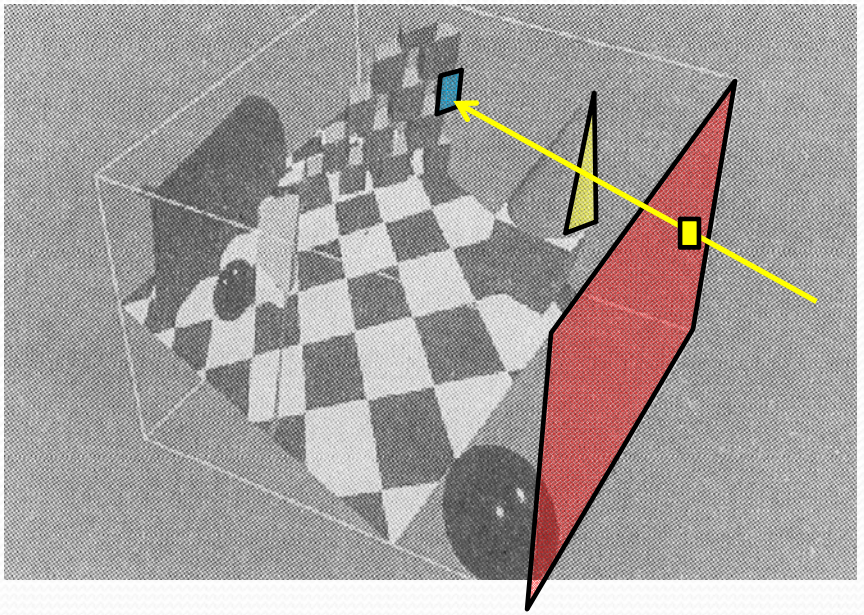
Type 2a: Z-buffer Algorithm

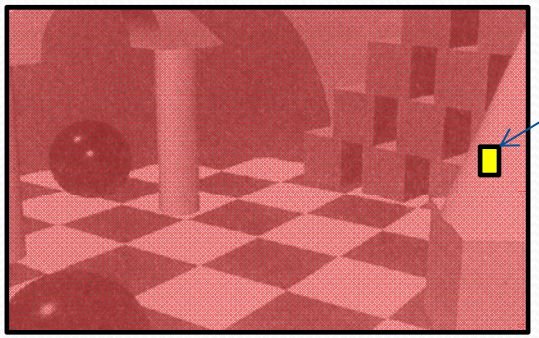
- When we draw a pixel on the screen
  - It is from a pixel by the SCA
  - We may overwrite the color onto the color buffer (memory)
  - The choices are:
    - To overwrite the original values already there, or
    - Do not write anything, i.e. ignoring the current pixel
- The main idea of *z*-buffer algorithm:
  - Allocate **another buffer** to store the *z* values of all the pixel drawn

Keeping the z Values

After Perspective Transformation





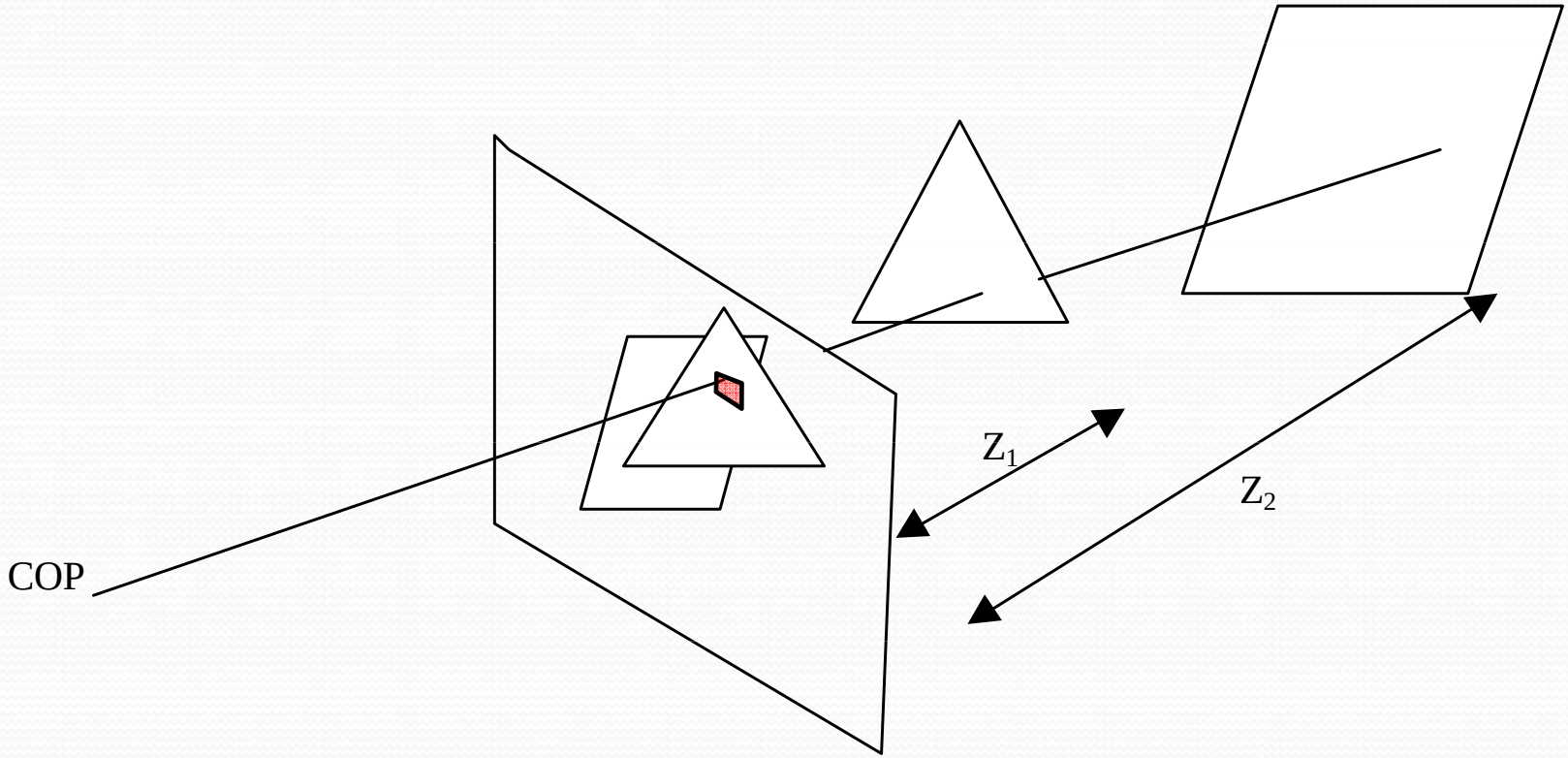


Yellow or blue?

- 2 Cases:
  - Draw blue first, then overwritten by yellow
  - Draw yellow first, then the blue pixel is skipped when the blue polygon is drawn

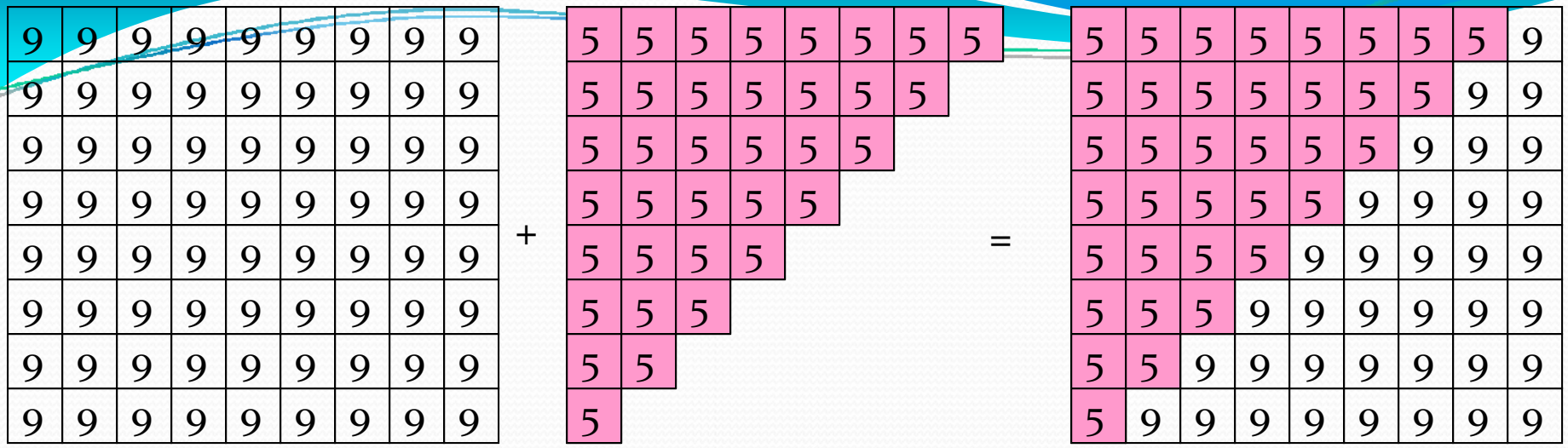
Type 2a: Z-buffer Algorithm

- For every pixel of a polygon (during SCA) will (usually) have a different *z* values

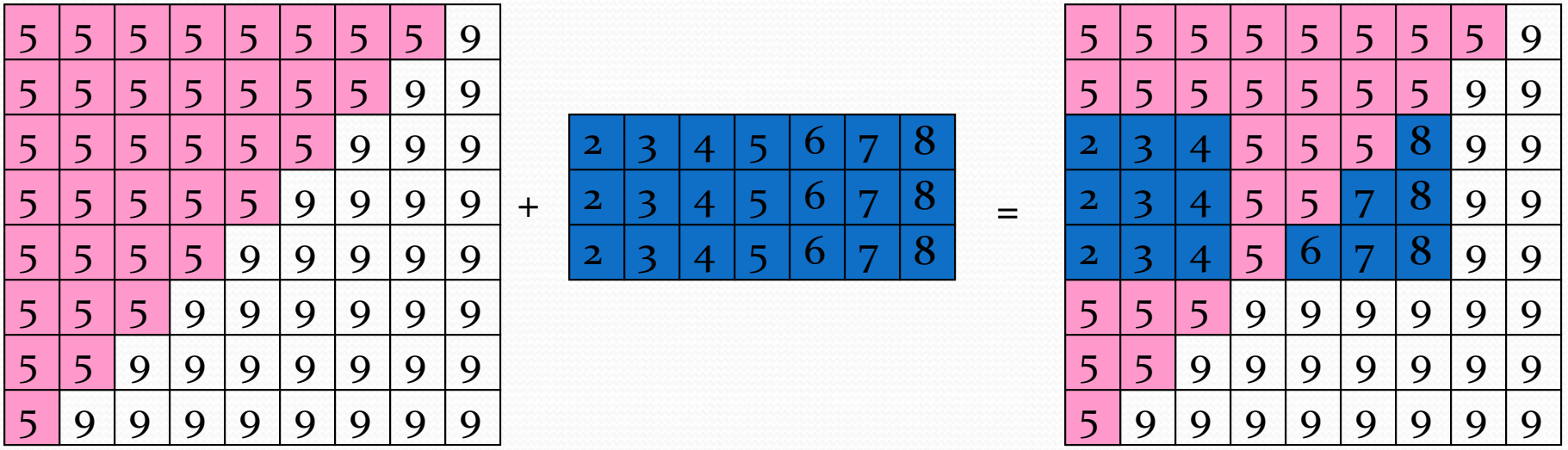


Type 2a: Z-buffer Algorithm

- The main idea of *z*-buffer algorithm:
  - Allocate **another buffer** to store the *z* values of all the pixel drawn, namely z-buffer
  - E.g. a 1024 x 768 screen has a memory size of 1024 x 768 array of "double" values
- Before drawing, initialize every value to be infinity
- When we draw a pixel, we compare its z value with the one on the z-buffer
  - If the depth is smaller, overwrite
  - Else, do nothing, namely, do NOT draw this pixel



Initial Screen z-buffer
(Let 9 be the maximum z value)

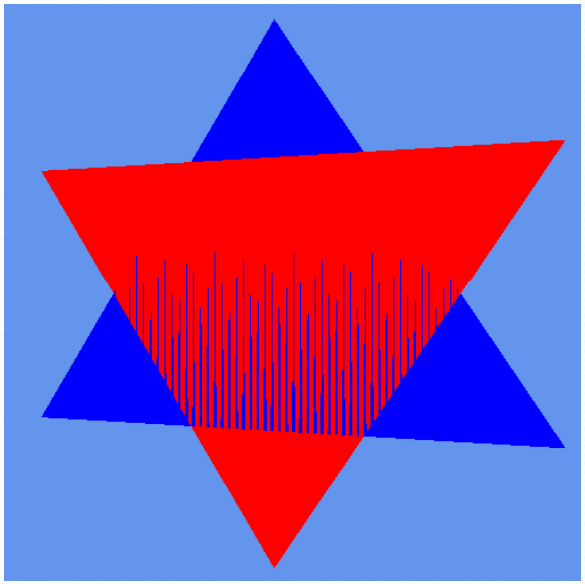


Type 2a: Z-buffer Algorithm

- **Advantages**
  - No pre-sorting is necessary,
    - no explicit intersection/object-object comparisons are required, i.e.
  - Fast and simple
  - Handles cyclic and penetrating polygons
  - Can make use of Δz to speed up in the incremental SCA
- **Disadvantages**
  - Can handle only opaque objects, namely, transparency is not handled
    - Well… can somehow fix it partially…
  - A single point sampling process, i.e. severe **aliasing**.
    - E.g. two polygons on the same plane but with different shapes
    - "Z fighting"
  - Problem in a sub-pixel level
    - The z-buffer's finite precision does not provide adequate resolution for scene with small objects separated far apart (in z) and for intersections of distant objects. (i.e. 2 pts close together may become touching each other due to rounding)

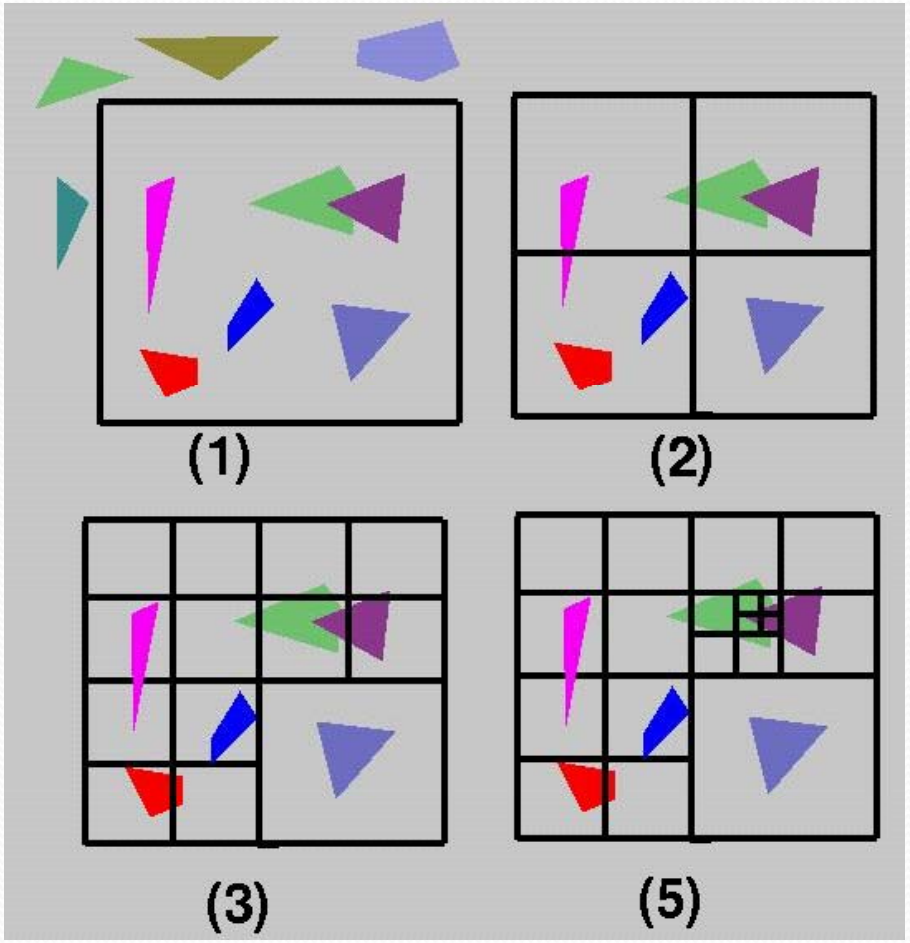
Z-fighting

- When two polygons are contained in the same plane and overlapping with each other
  - Theoretically, their z values are exactly the same at every pixel
  - But, practically, numerical errors



Type 2b: Warnock's Algorithm

- Subdivide the screen until
  - The area only contains one polygon, or
  - The area is one pixel big
- For the one pixel big area
  - Determine which polygon to draw by intersecting a line to all the polygons in that pixel



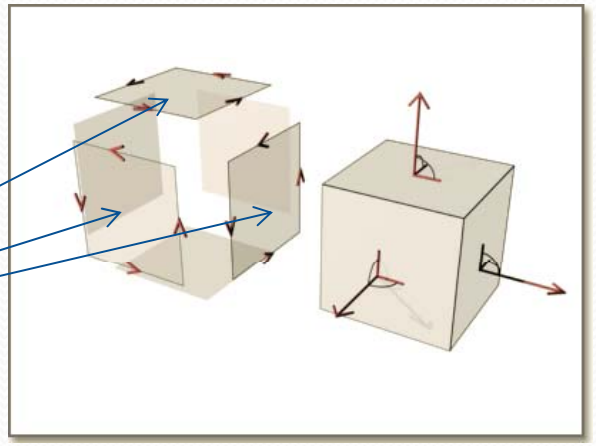
Speed-up Enhancement

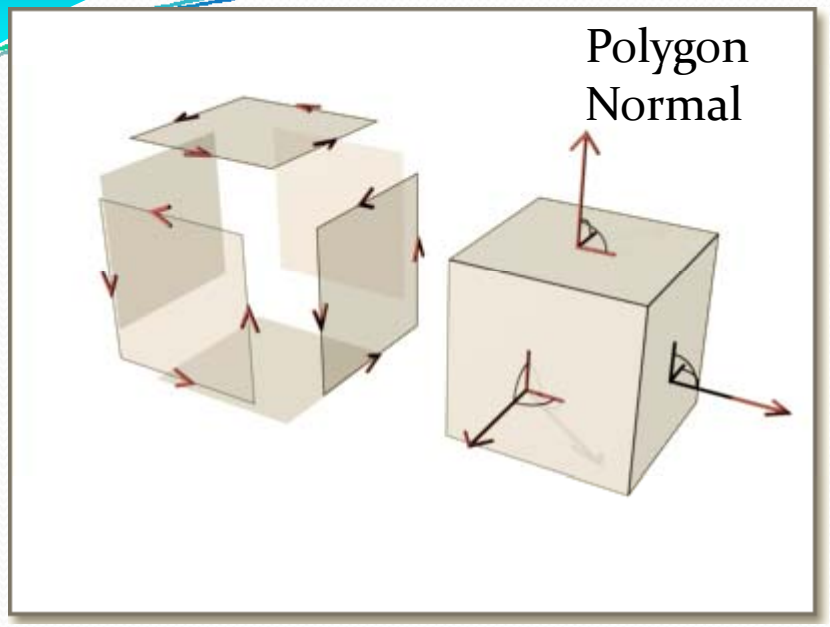
(Reducing the number of polygons rendered)

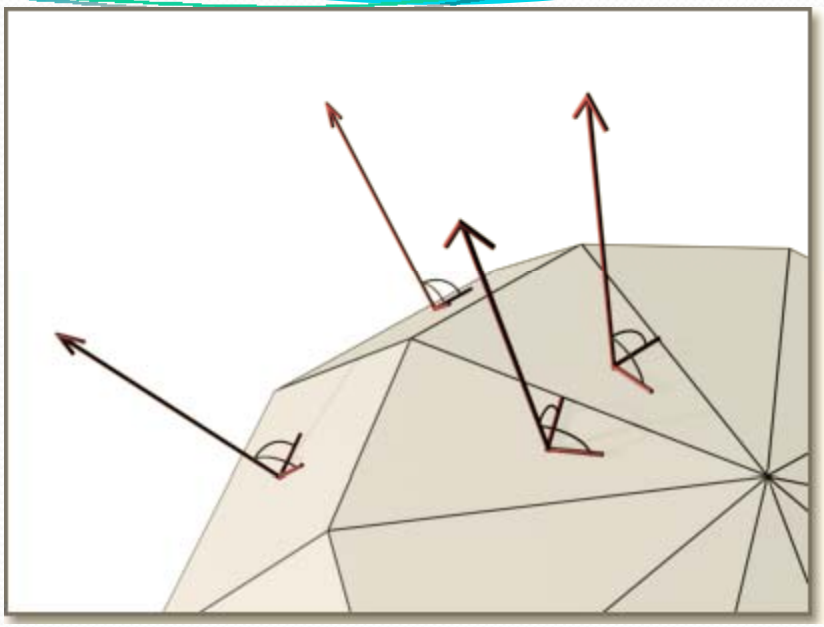
Speed-up Enhancements

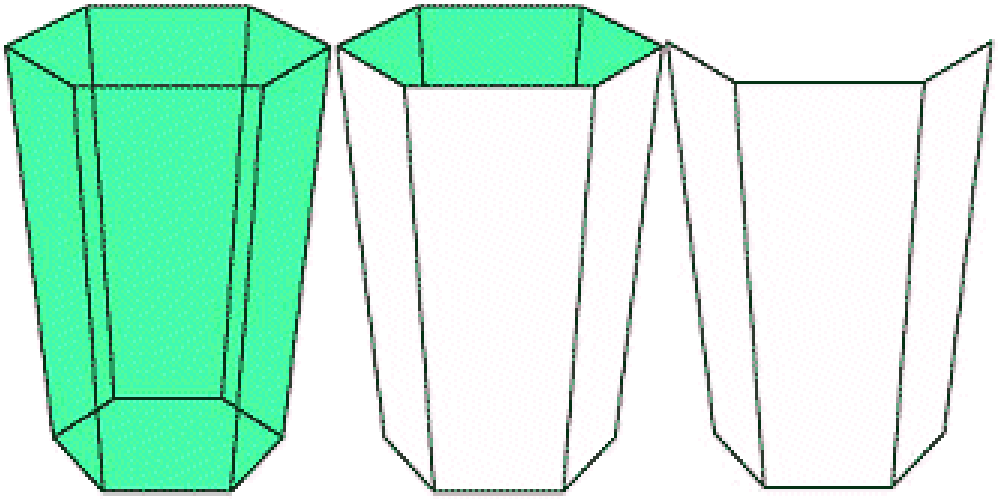
1. Skipping (eliminating) objects outside the view frustum
   - Many many methods…
2. **Back face culling**
   - For each polygon, define one side is "outside" and the other "inside"
   - Only draw the polygon if the "outside" face is facing the viewer

Only these three polygons are actually drawn by SCA, the rest are skipped



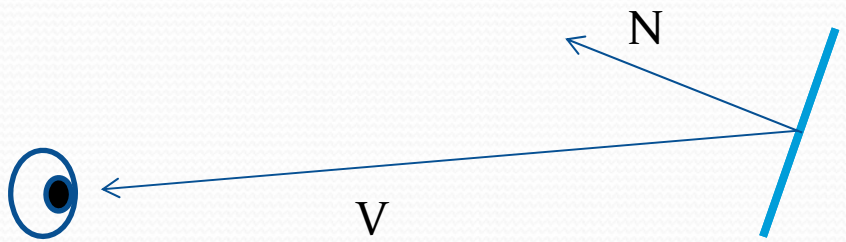

Polygon
Normal




All surfaces
Hidden surface removed
Back faces culled

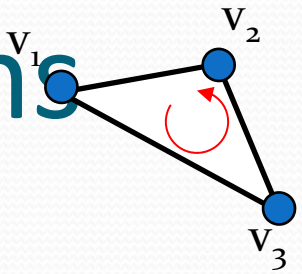
Back face culling

- For each polygon, define one side is "outside" and the other "inside"
- Only draw the polygon if the "outside" face is facing the eye by testing if the **dot product** of the two vectors *V* and *N* is greater than zero
  - *V*: eye position minus a point on the polygon
  - *N*: normal vector of the polygon

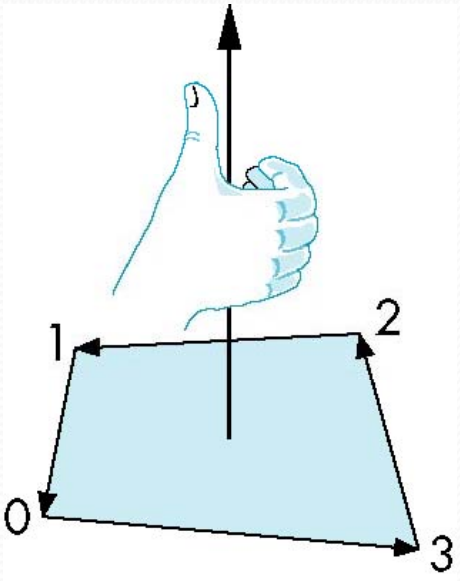


That's why you need to know the polygon orientation in Lecture 4

Recap:Polygon Orientations

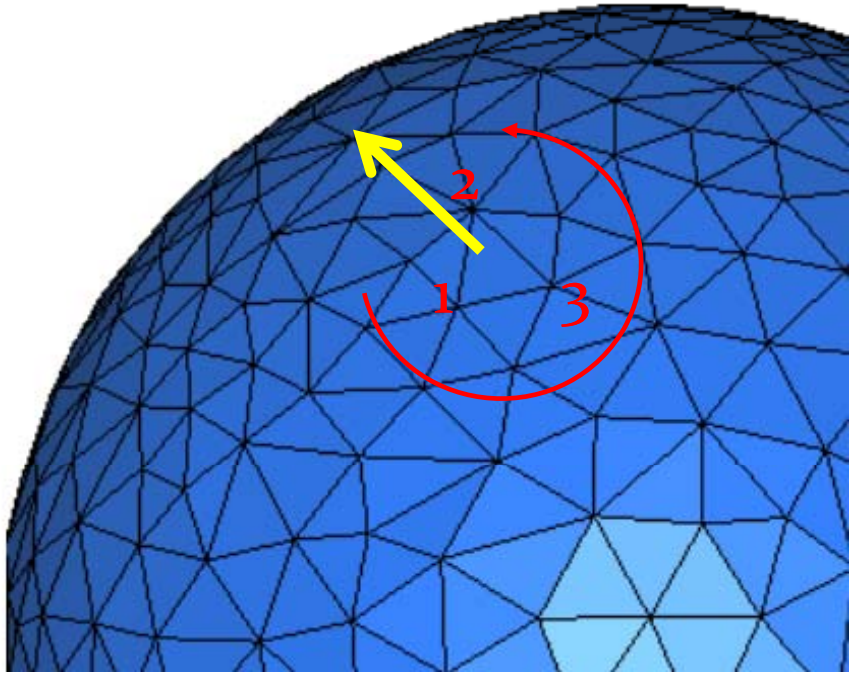


- For a triangle with three vertices v_1, v_2 and v_3
  - The order (v_1, v_3, v_2) and (v_3, v_2, v_1) are considered to be equivalent during drawing, e.g. `glVertex`
- But the order (v_1, v_2, v_3) is considered to be different (or "opposite")
- (v_1, v_3, v_2) and (v_3, v_2, v_1) describe a **facing direction** of a polygon/triangle
  - Right-hand rule
  - The order (v_1, v_2, v_3) creates an opposite facing direction
- So, geometry is the same, but normal vector for that polygon will be different
  - Usually we follow the "natural" order in the polygon list to define the "outward" direction of an object

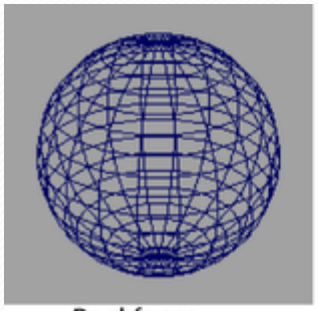


Vertex List and Polygon List

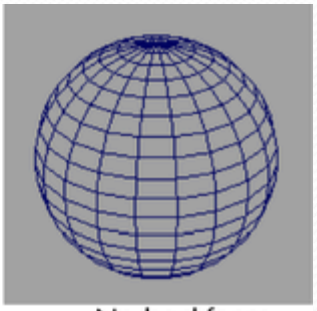
- For example, if it is a “solid” ball, the triangle below must be stored in the order of
  - (v1, v3, v2) or (v3,v2,v1) or (v2,v1,v3)
  - But NOT (v1,v2,v3) or some other orders



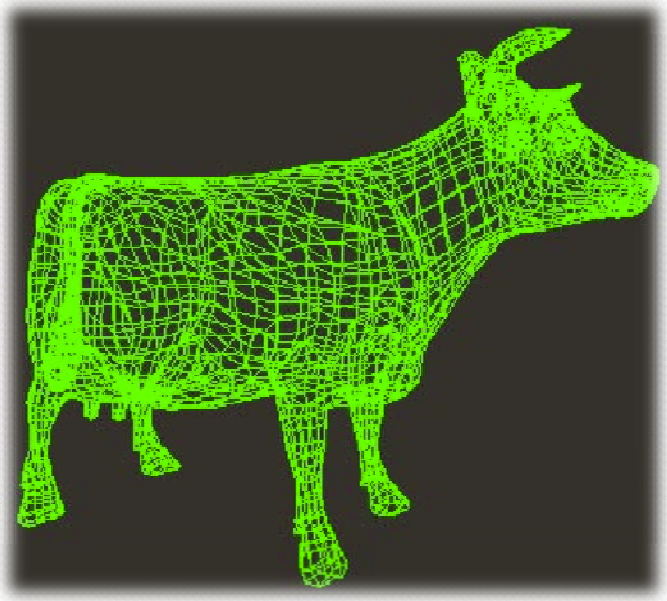
Back Face Culling

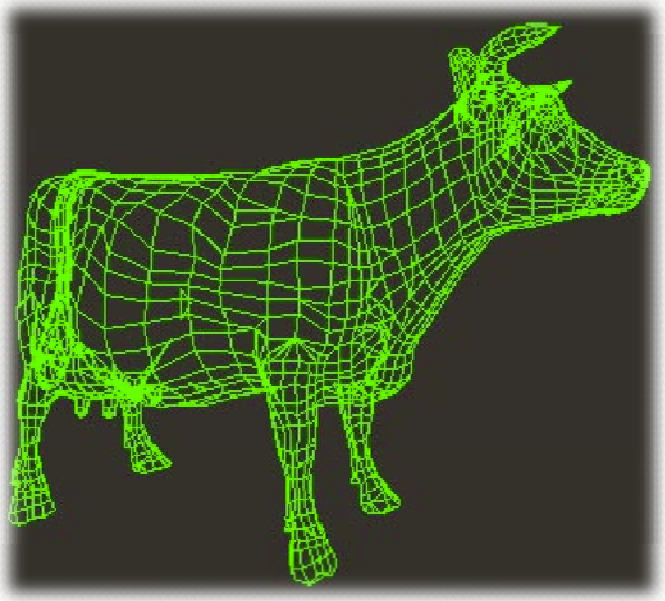


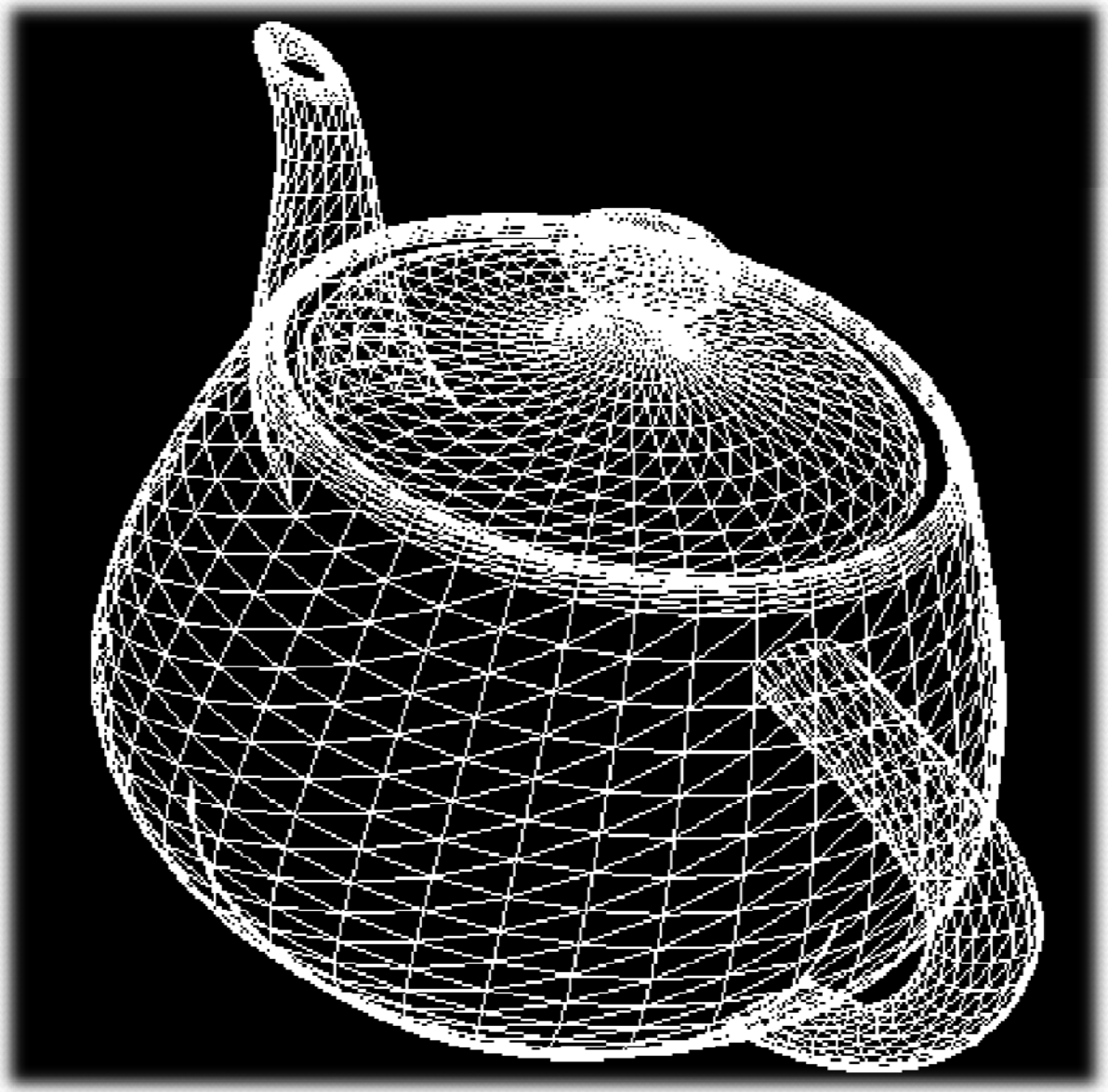
Backfaces



No backfaces







OpenGL

- Z-buffer (depth buffer)
  - Setup:

    ```
    glEnable(GL_DEPTH_TEST);
    glDepthMask(GL_TRUE);
    ```

  - Every time clearing the screen

    ```
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    ```

- Back face culling
  - Setup

    ```
    glEnable(GL_CULL_FACE);
    glCullFace(GL_FRONT);
    ```

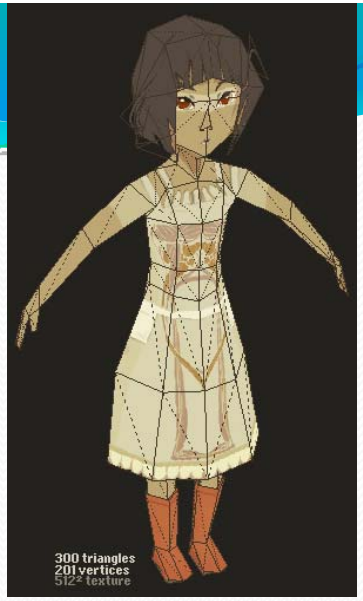
    or

    ```
    glCullFace(GL_BACK);
    ```

  - Then draw all polygons in an anti-clockwise or a clockwise order

Outline Effect

- A polygonal model
- Draw the outline
  - The division between polygon being “backface culled” and not culled



Conclusion

- This all COMPUTATIONAL GEOMETRY!
- Z-buffer is not superior,
  - but most “user friendly”
  - Most suitable for hardware implementation
- Other algorithms are more suitable for software implementation
- Performance depends on
  - The number of polygons
  - Size of the output
- Those who are interested:
  - Quake’s HSR techniques:
    - http://downloads.gamedev.net/pdf/gpbb/gpbb66.pdf

Admin

- HW1
  - Please check your marks
- HW2 Deadline 1st March
- Midterm **next next** lecture in class
  - 14th March
  - Start sharply at 10 am
  - 1 hour
  - Open Book
  - Scope: everything (lecture, tutorial, hw) before 11th March
    - See IVLE lesson plan for details