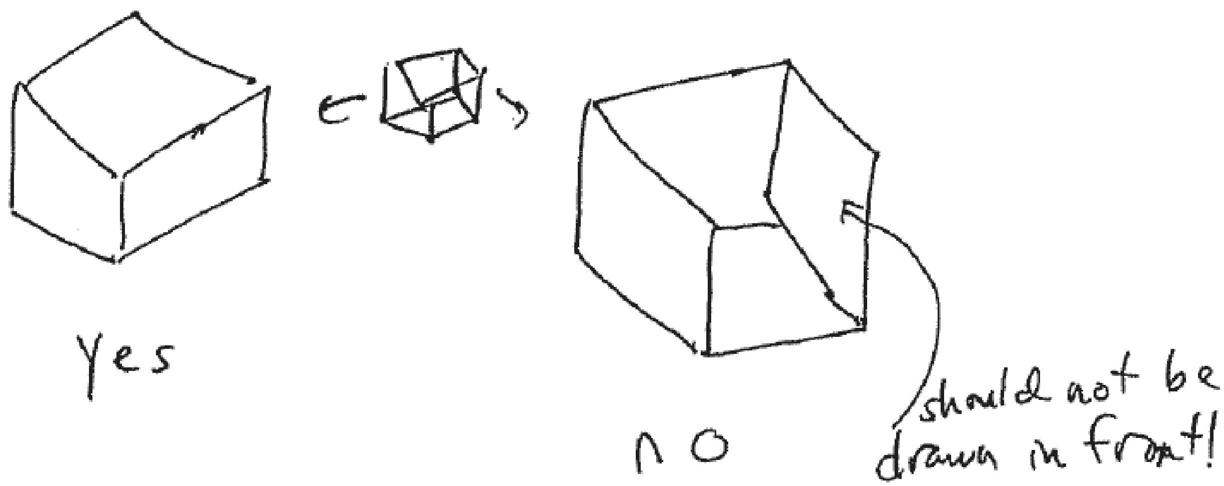
CS 428: Fall 2009

Introduction to Computer Graphics

Visibility

Visibility

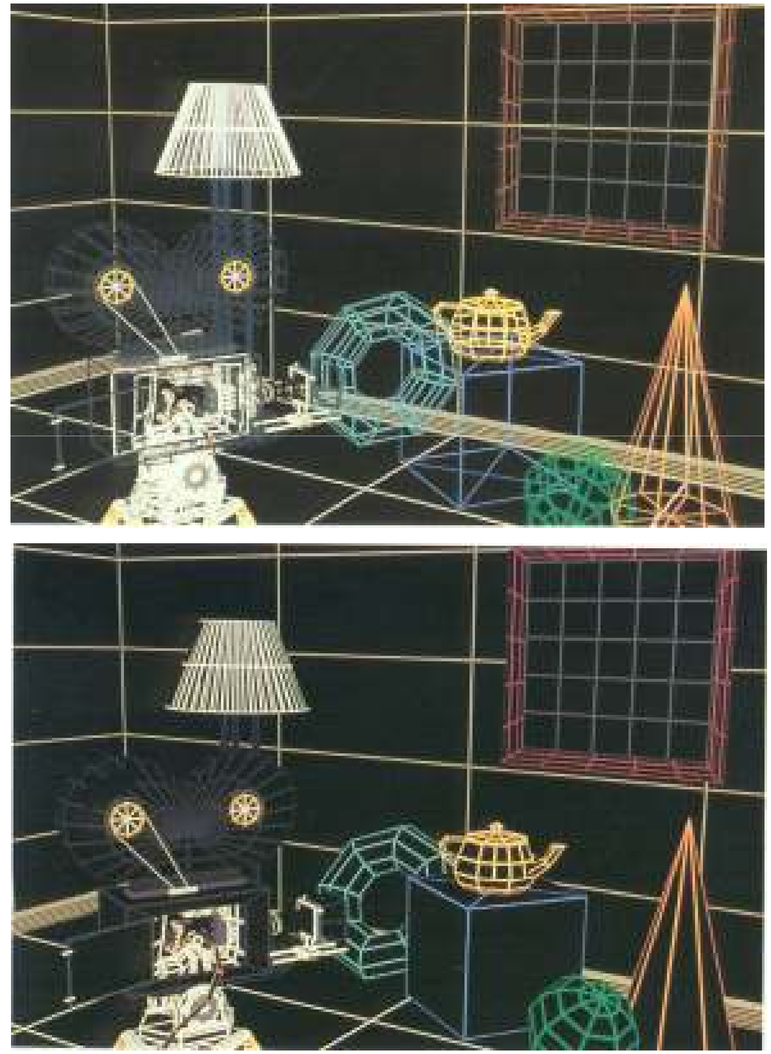
- Also known as **hidden surface removal**
  - Respect the nature of occlusion in visual scenes



  - Algorithms for determining which parts of the object/surfaces are **visible**
    - For now only **opaque** objects

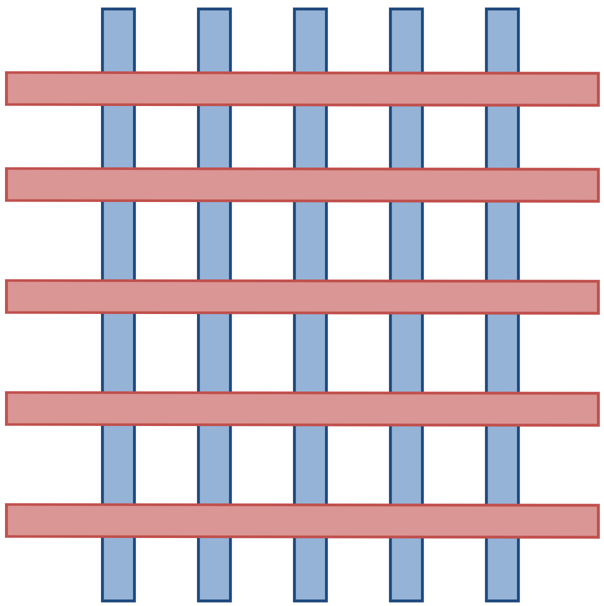
Visibility algorithms

- **Occlusion**
  - More than one point projects to the same point in the image
  - Obviously, the point closest to the observer is visible
  - Unless the closest point is (semi)transparent, in which case objects behind become visible



Visibility algorithms

- **Complexity**
  - Visibility computation is comparable to sorting
  - Worst case complexity is worse
  - Given a scene with **n** polygons, there might exist ~$\mathbf{n^2}$ visible parts
  - Worst case complexity is $\mathbf{O(n^2)}$



Visibility

- A variety of algorithms
  - Each work better (more efficiently) in different situations

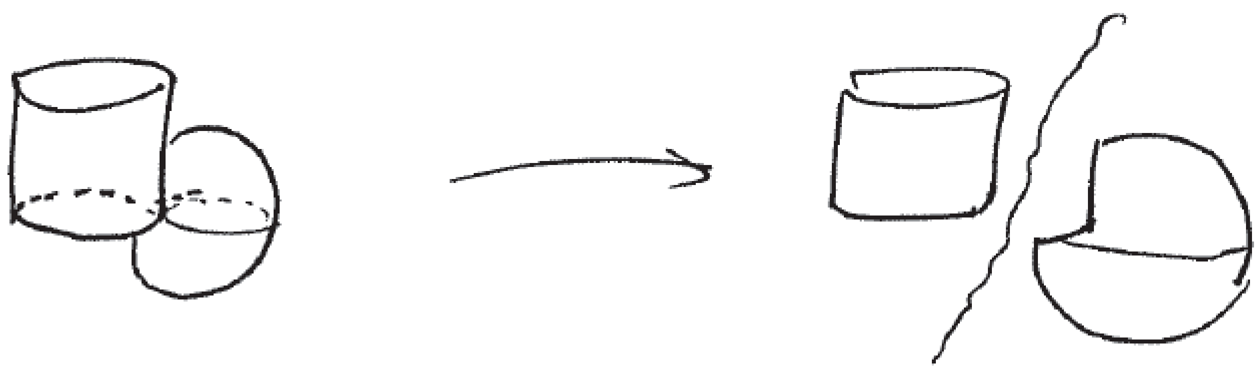
- Two main categories
  - Object precision algorithms
  - Image precision algorithms

Object precision

- Operate on geometric primitives
  - For every object in the scene
    - Compute visible part (not occluded by **any** other object in the scene) → needs **high** precision
    - Draw visible part
- Results are independent of display resolution
- Brute force algorithm is $\mathbf{O(n^2)}$
  - n = number of objects on screen
  - Can be improved (pre-computation) to **O(n log n)**

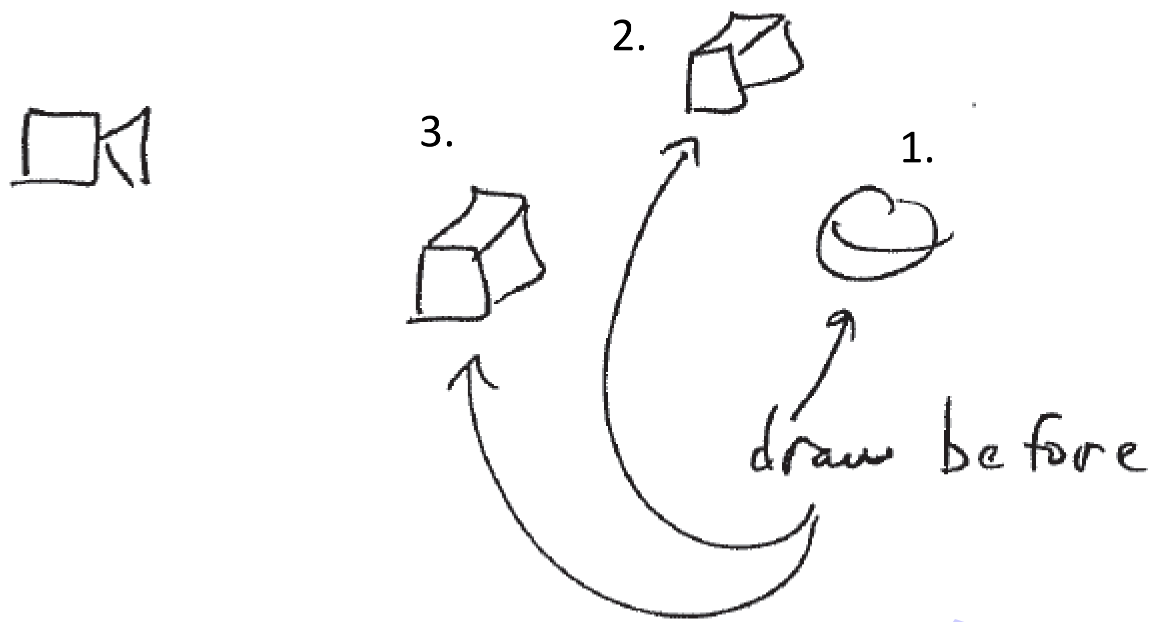
Object precision

- Hard(er) to implement
  - Due to numerical error
  - Due to tricky geometric computations (intersections, Boolean operations, etc.)

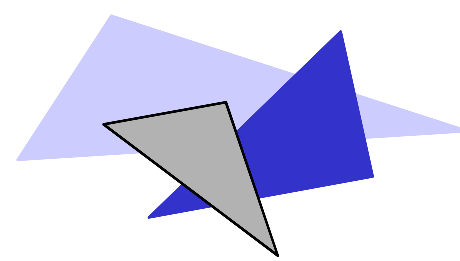


List priority methods

- Draw surfaces in **back-to-front** order

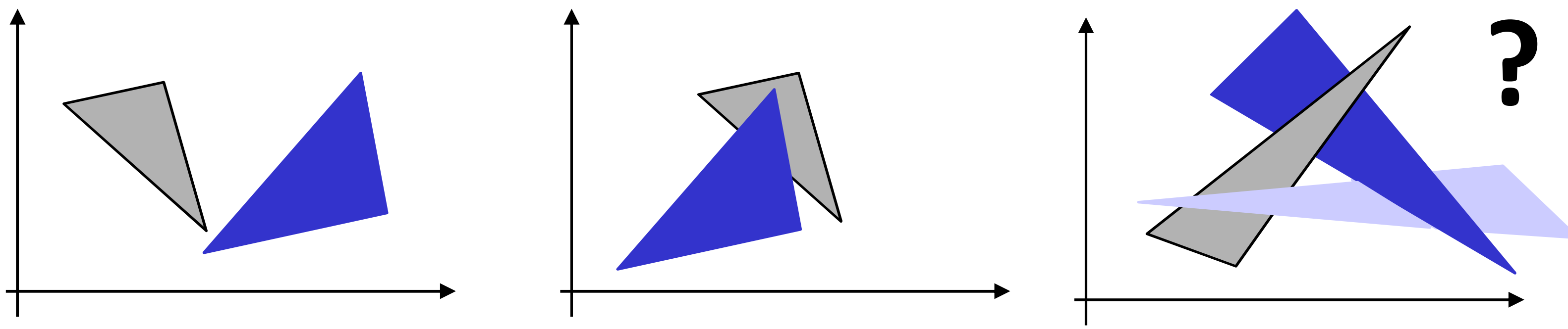


  - **Painters algorithm**



List priority methods

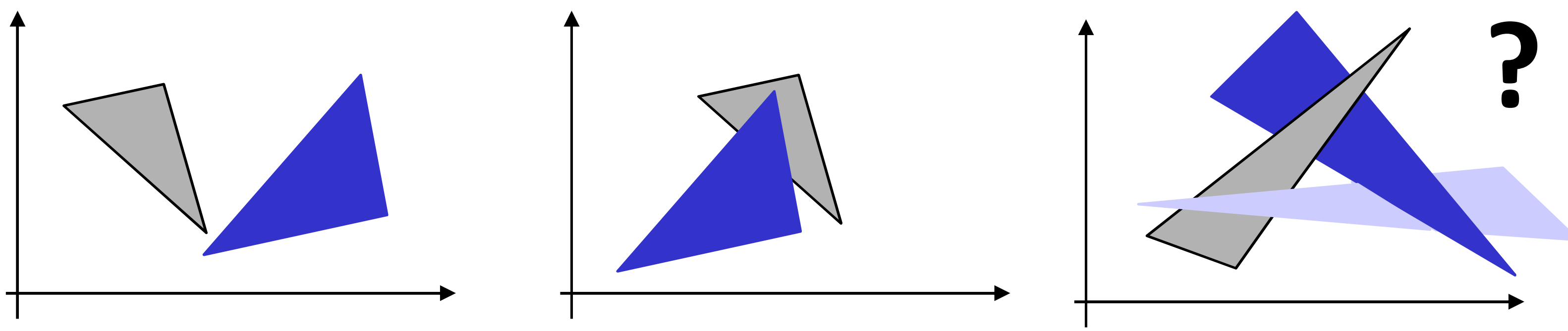
- **Problem:**
  such an ordering does not always exist



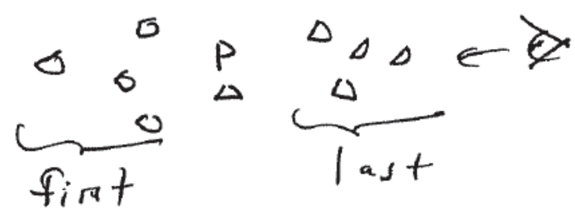
  - In such cases, polygons must be **split**
  - This can result in many split polygons (see worst case complexity)

List priority methods

- **Observation:**
  polygons are drawn in the correct order if



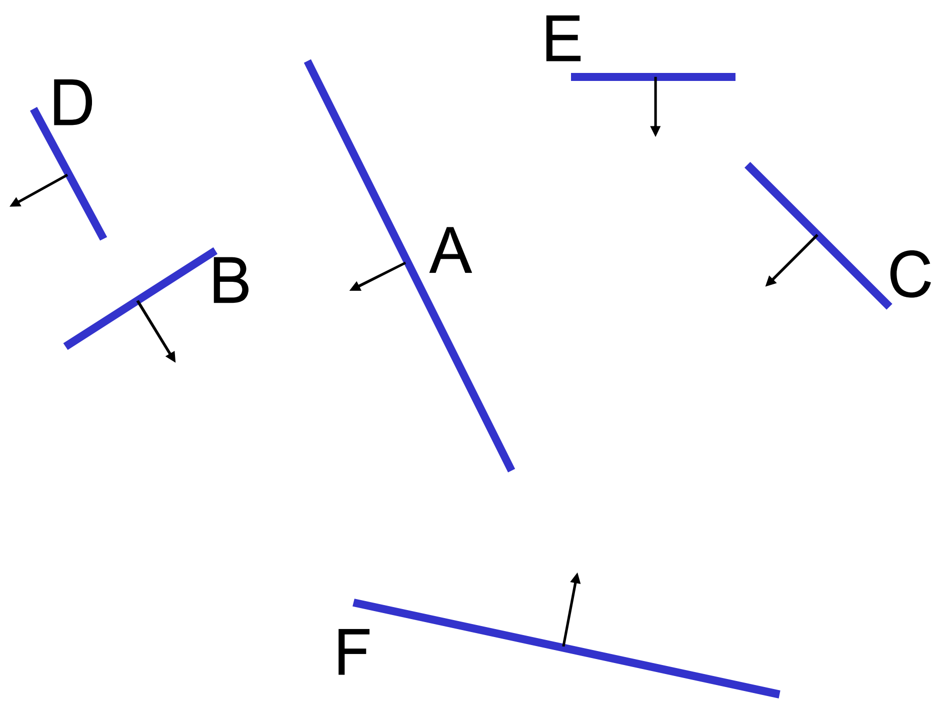
  - For every polygon part **P**
    - Draw everything behind **P**
    - Draw **P**
    - Draw everything in front of **P**



Binary space partion (BSP)

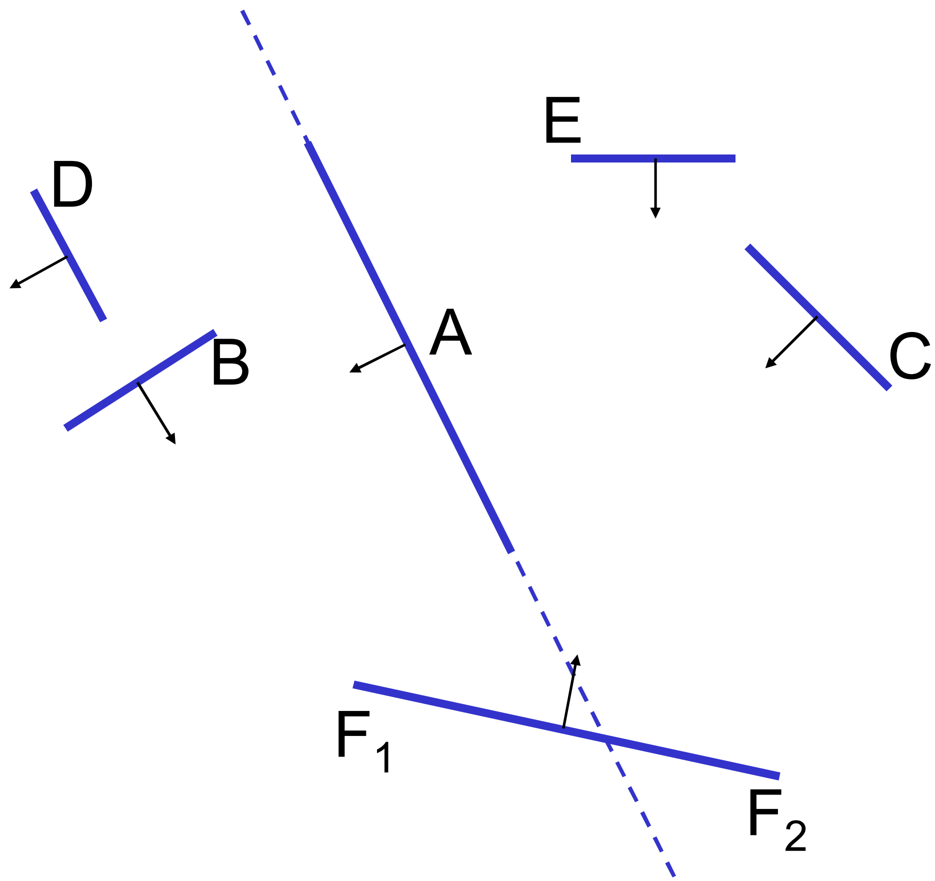
- BSP tree: binary spatial subdivision
- A tree that encodes **viewpoint-independent** and **relative** position/depth information
  - Every node is a splitting plane, which cuts space into two parts (two **half-spaces**)
  - Leads to an ordering with respect to every line in 2D (plane in 3D, etc.)
  - For visibility, the splitting (hyper)planes are defined by the scene geometry

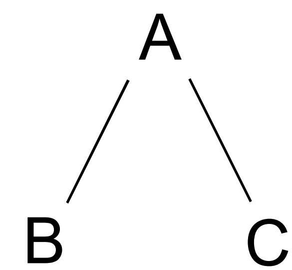
Binary space partion (BSP)



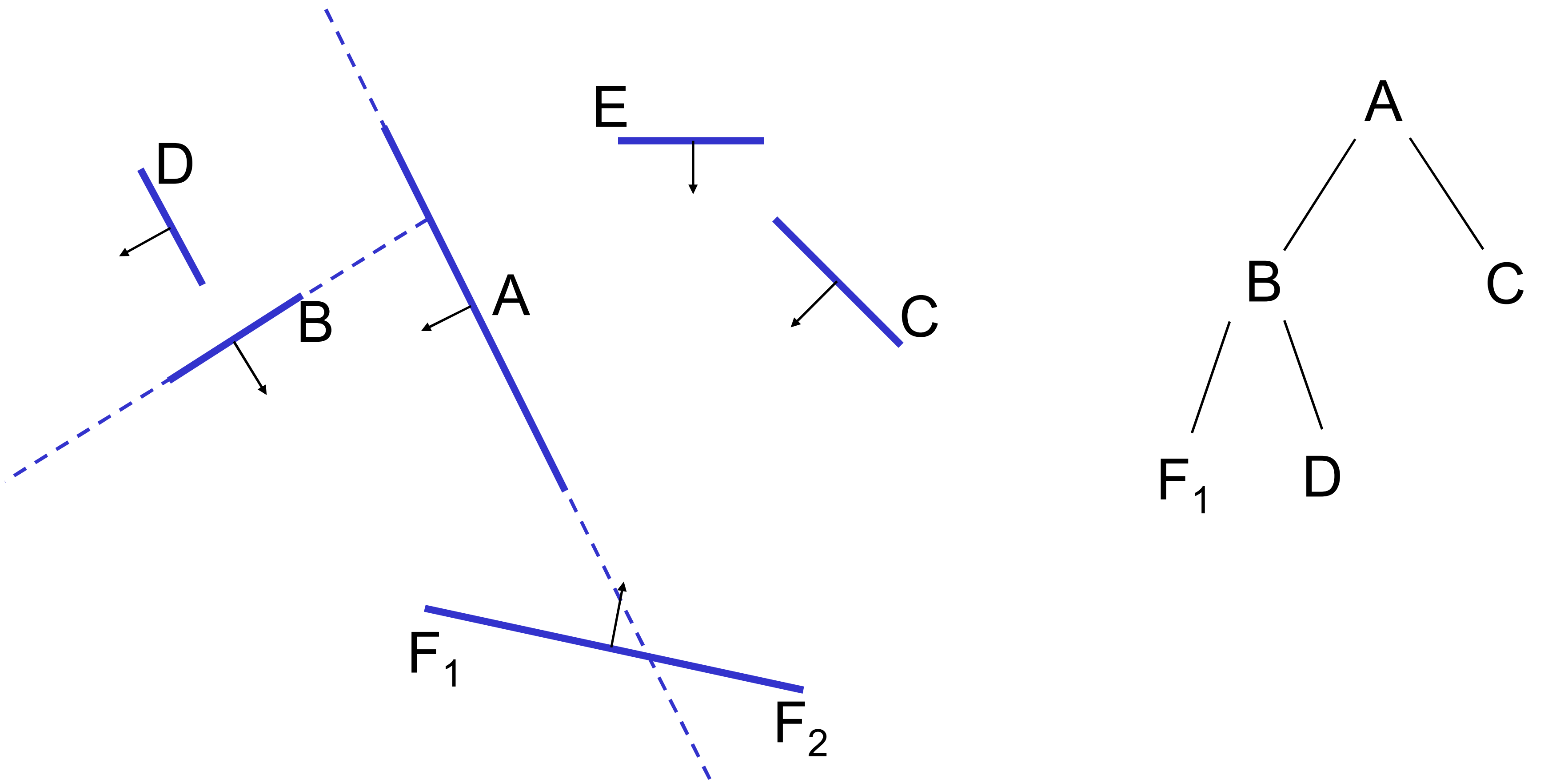
A

Binary space partion (BSP)

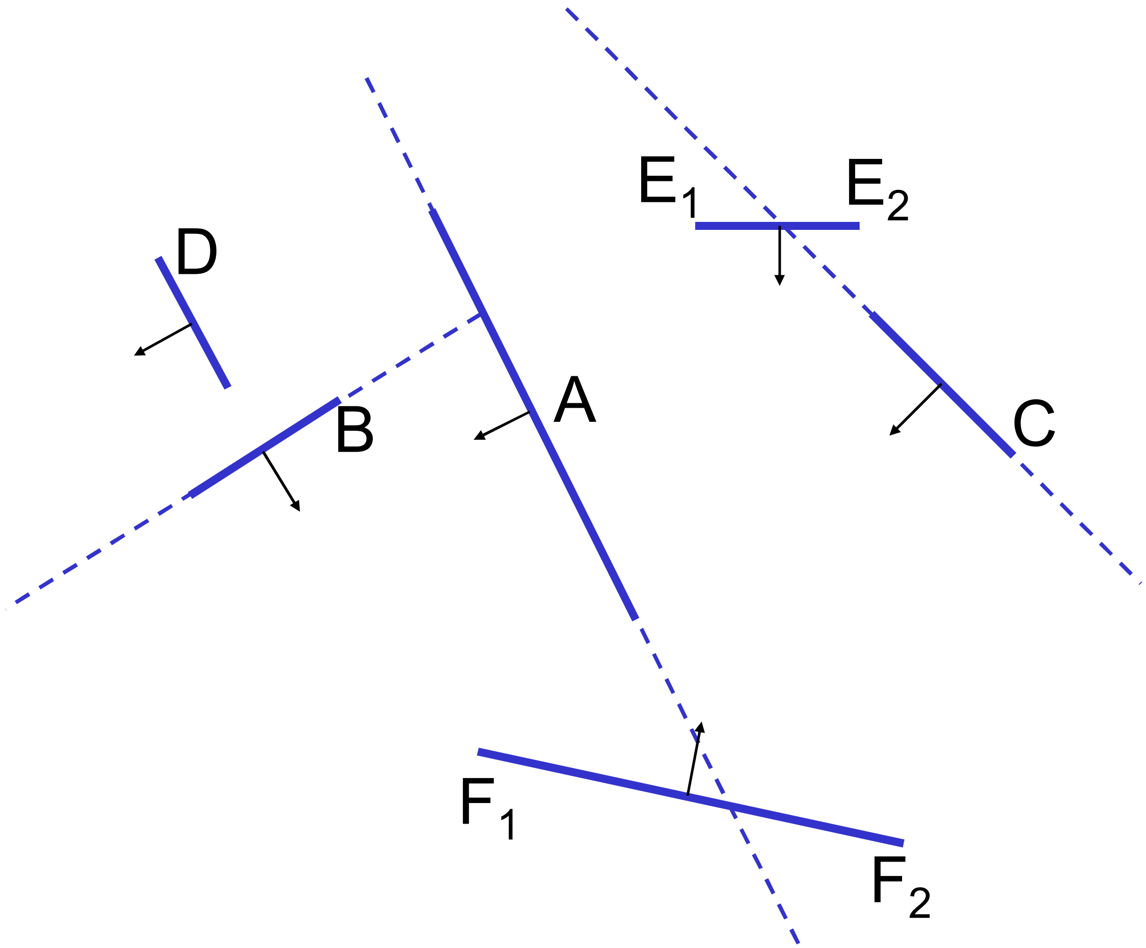


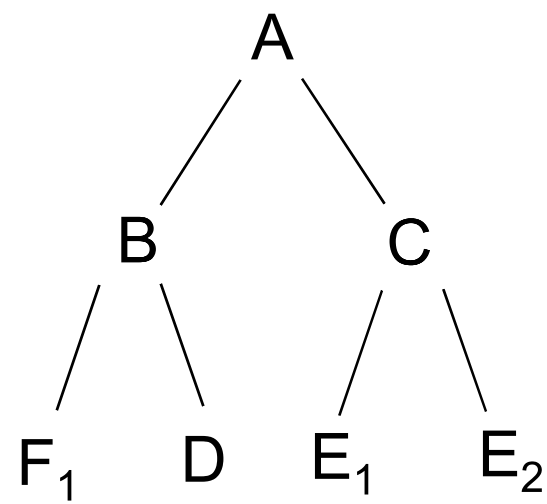


Binary space partion (BSP)

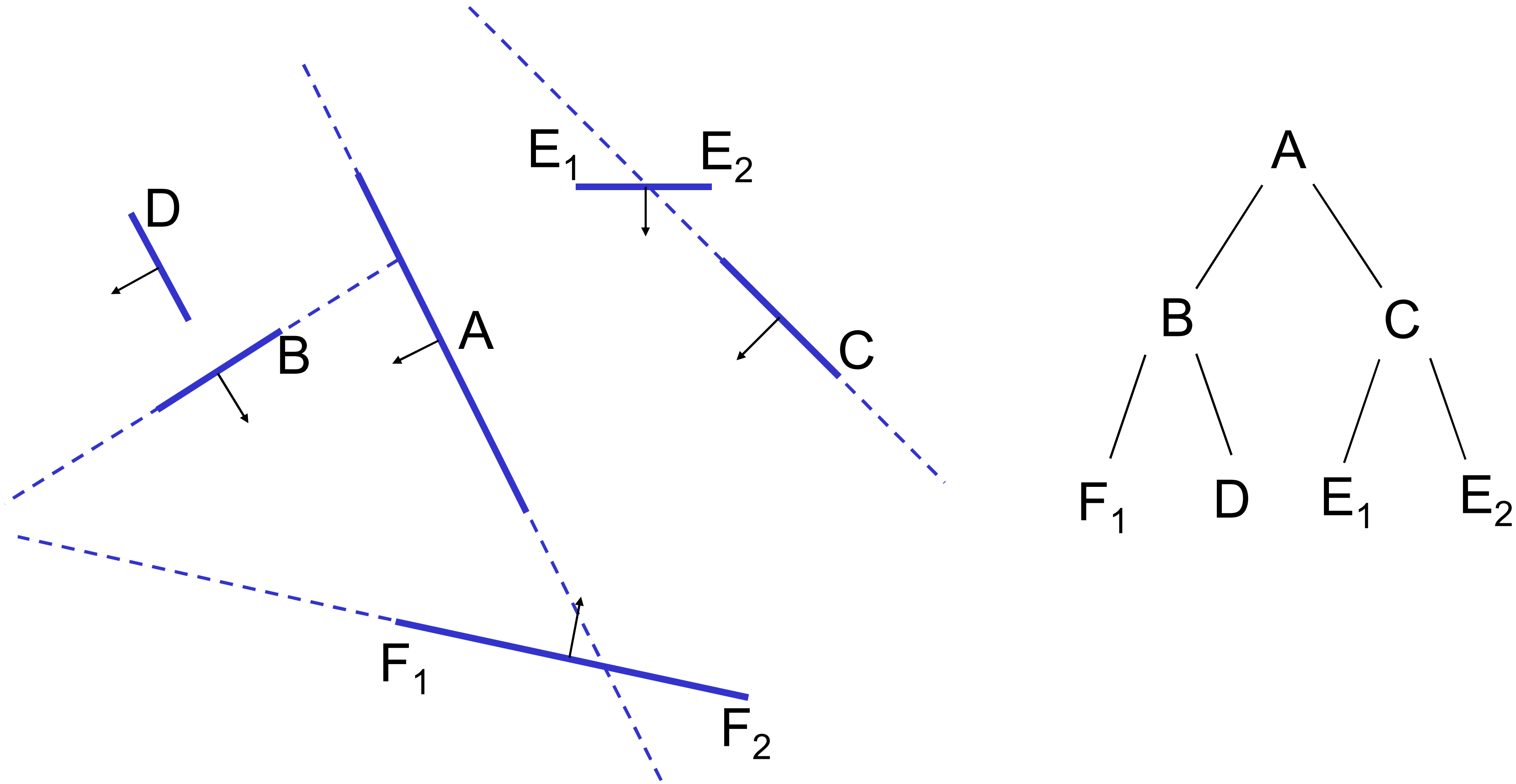


Binary space partion (BSP)

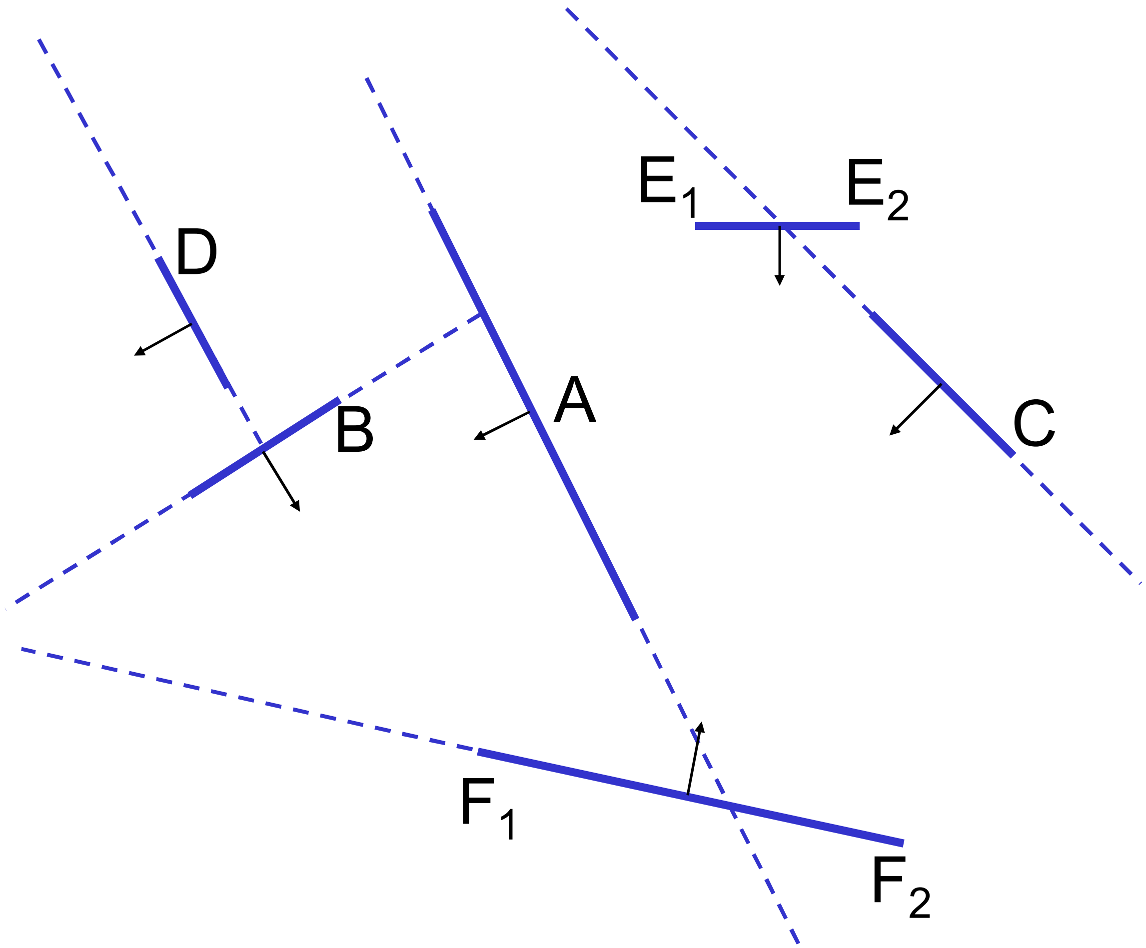


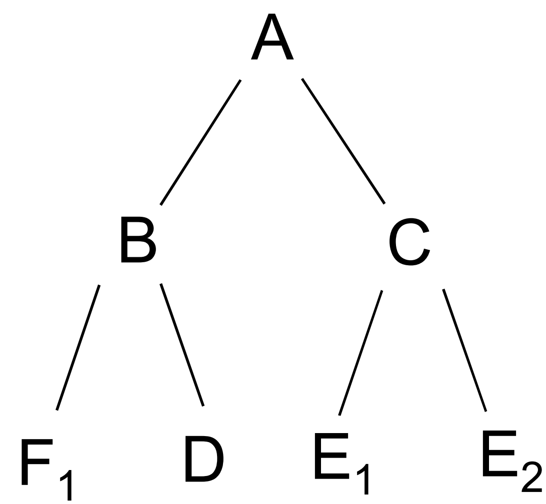


Binary space partion (BSP)

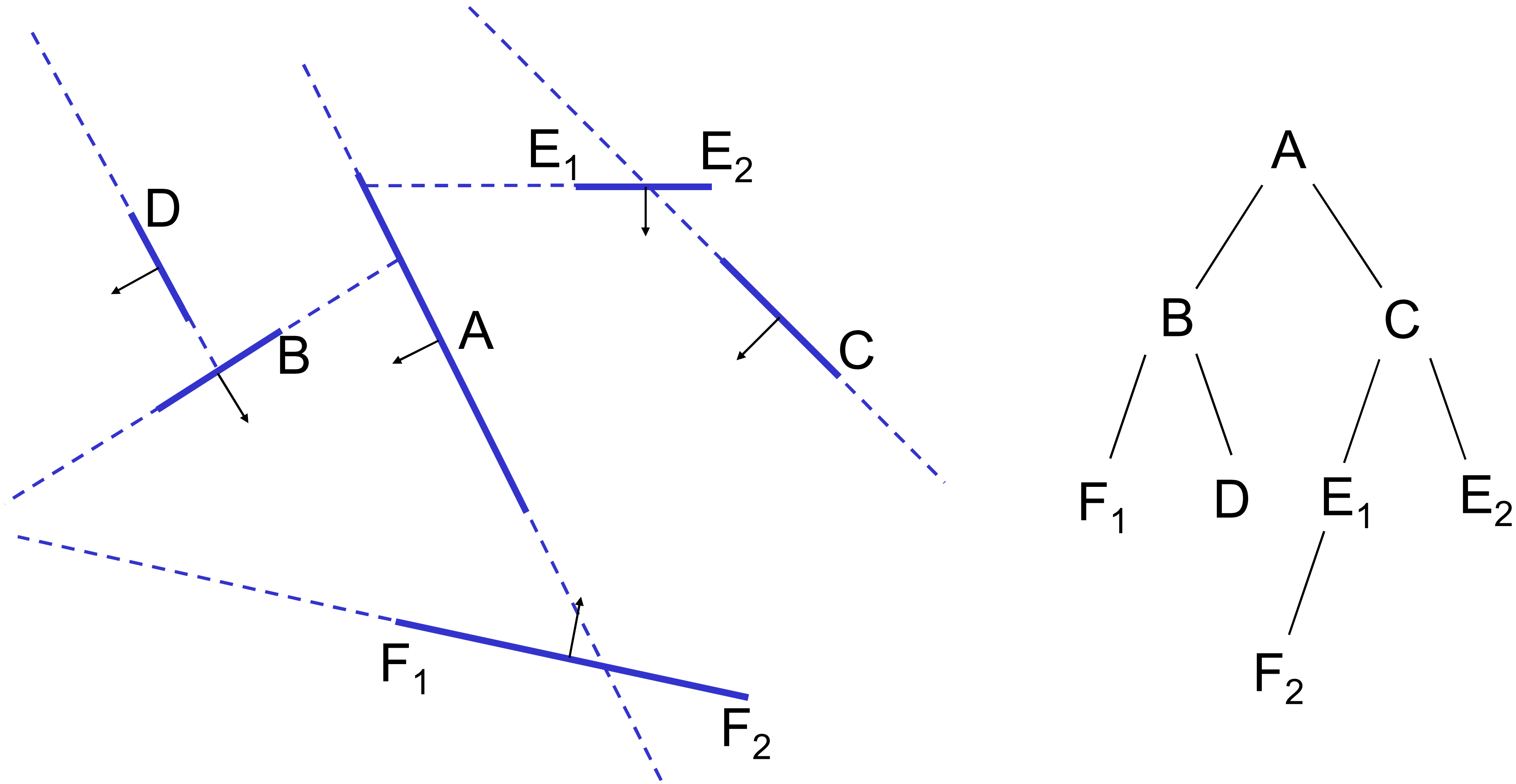


Binary space partion (BSP)

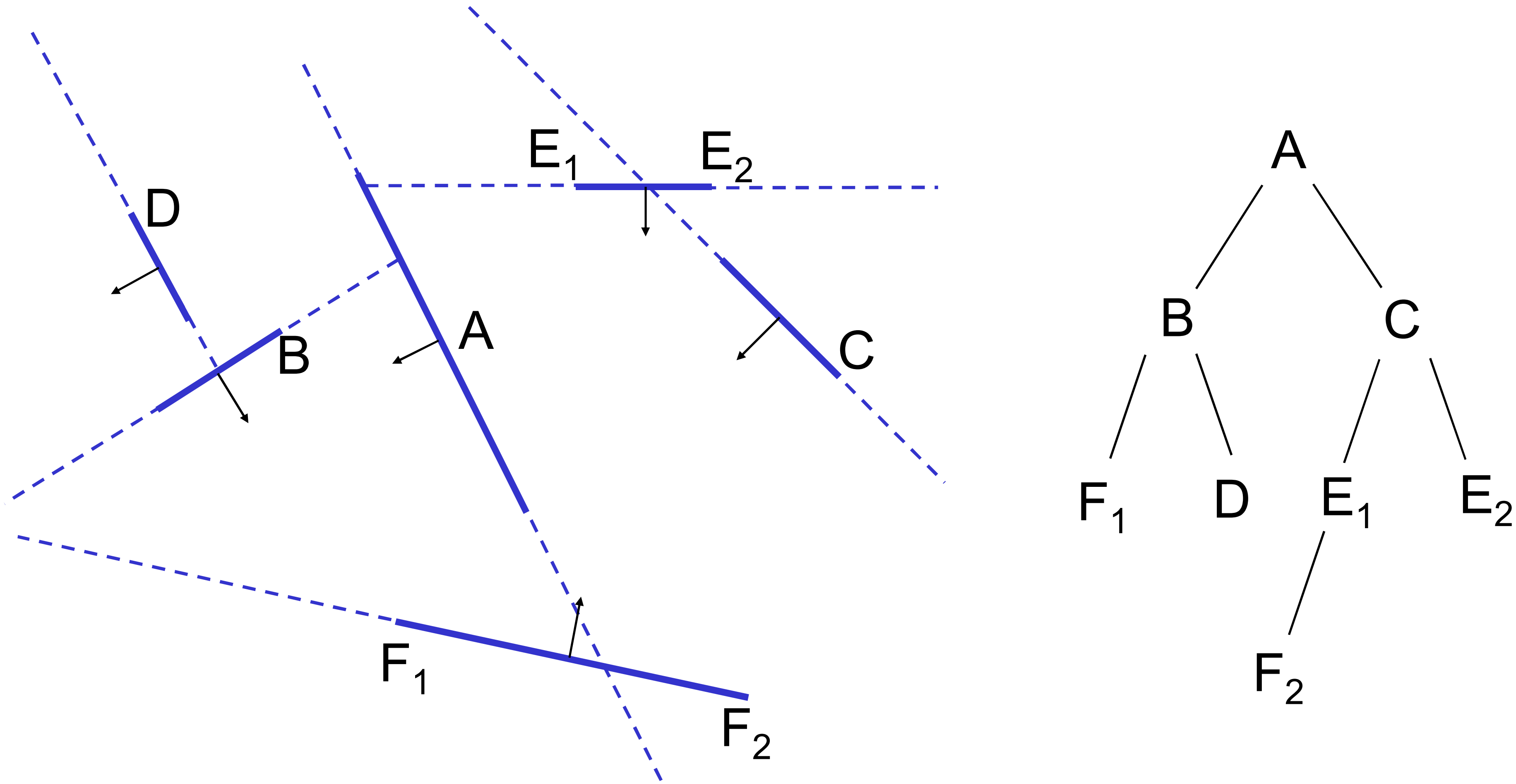




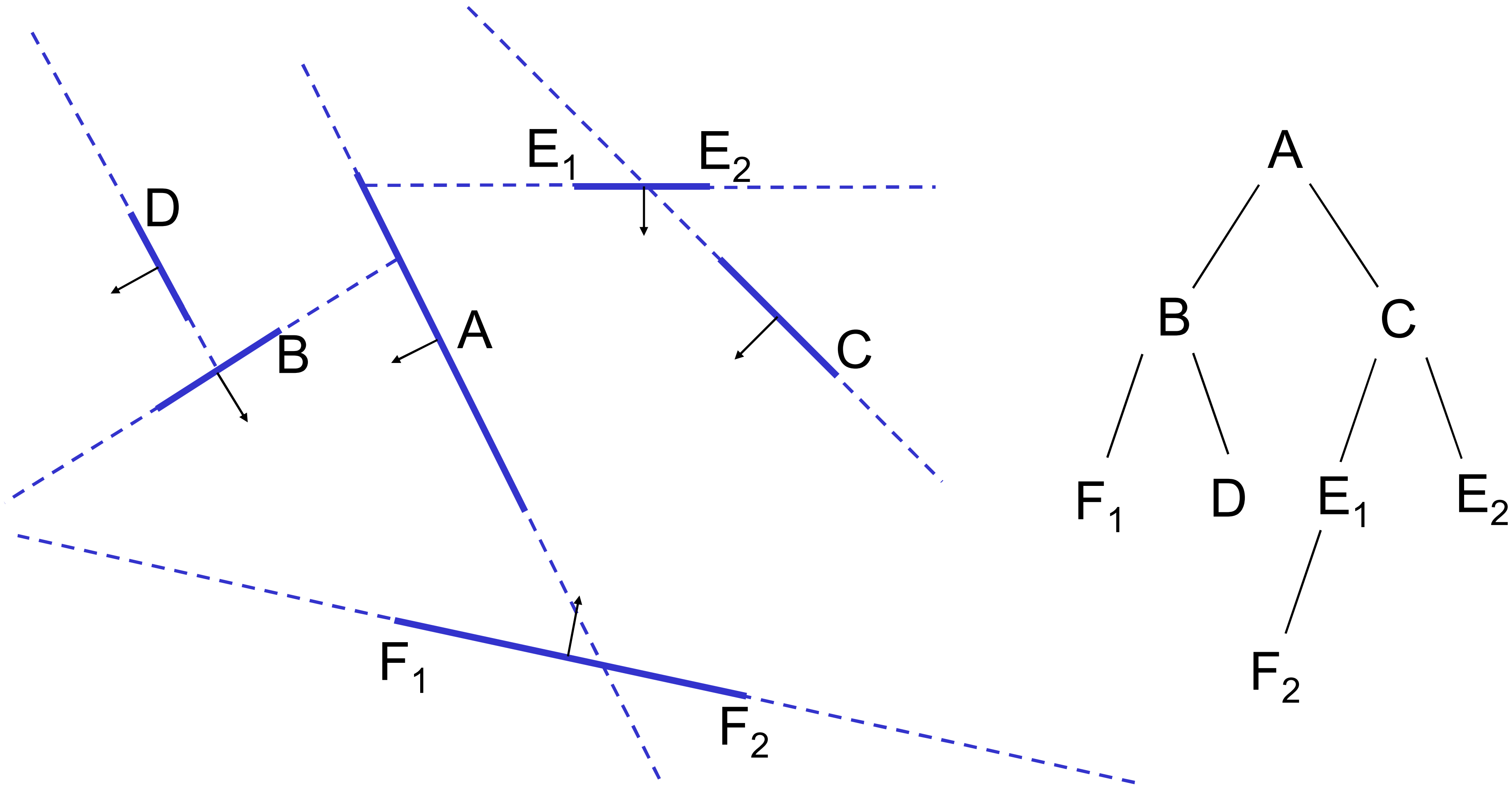
Binary space partion (BSP)



Binary space partion (BSP)



Binary space partion (BSP)

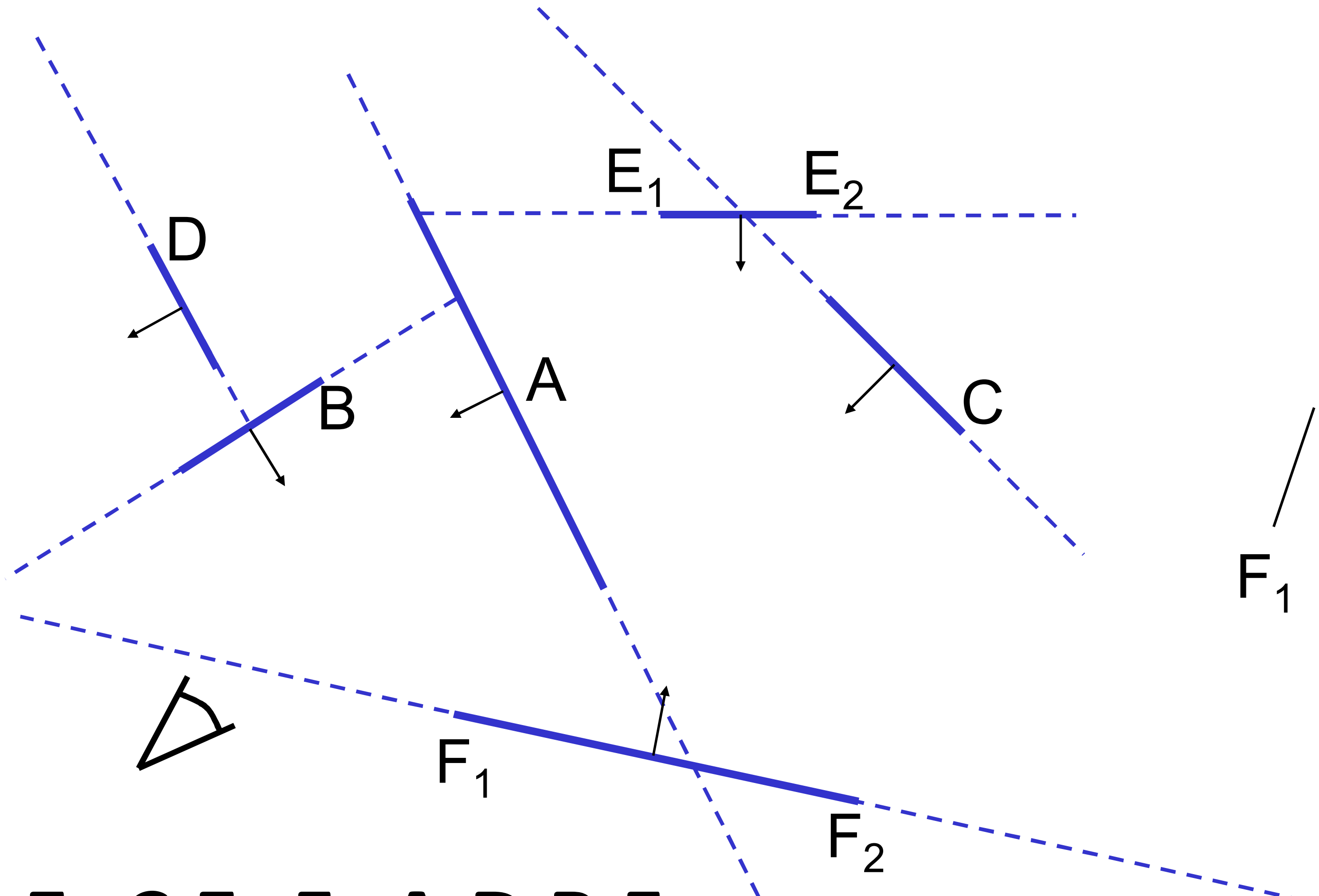


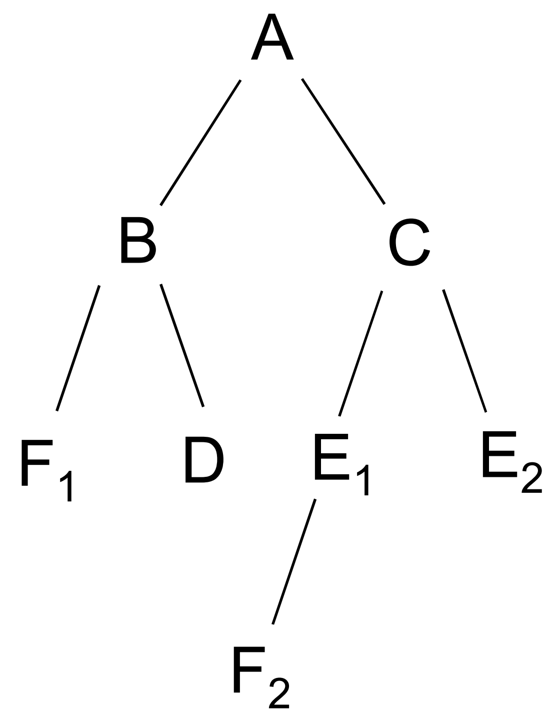
Binary space partion (BSP)

- Ordered list of polygons by traversal
- Identify half-space **H** of eye position

- Traversal ordering
  - Other half-space
  - Polygon (node)
  - Containing half-space **H**

Binary space partion (BSP)





E_2,C,E_1,F_2,A,D,B,F_1

Binary space partion (BSP)

- Some issues
  - Which plane to chose as the splitting plane in each step?
  - How to balance the tree?
  - How to avoid excessive polygon splitting?
- Solution
  - Re-run the algorithm!
    - "Perfect" BSP is in NP (exponential complexity)
    - Randomized version works well, has good expected performance

Image precision

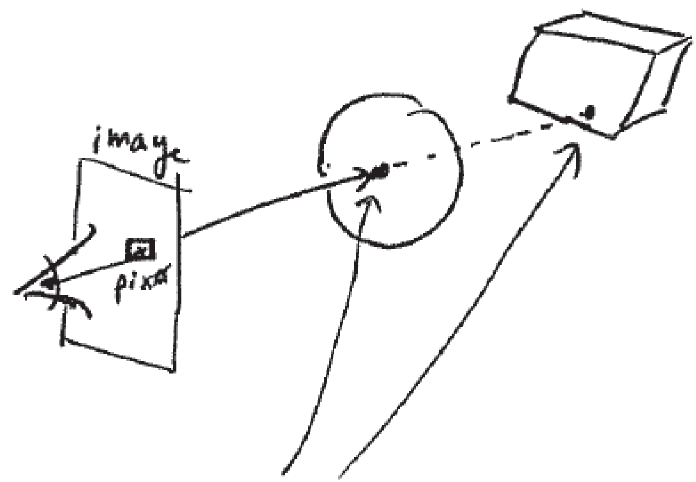
- You already know one: **z-buffer**
- Z-buffer algorithm is an **output sensitive** algorithm (only looks at rendered pixels)
- Brute force
  - For each pixel
    - Find object closest to camera which projects to here
    - Draw that object
- Complexity is **O(nP)**, while z-buffer is **$O(nP_R)$**

Image precision

- You already know one: **z-buffer**
- Z-buffer algorithm is an **output sensitive** algorithm (only looks at rendered pixels)
- Z-Buffer
  - Initialize depth image **D** to farthest distance
  - For each pixel **p** of each polygon with depth **d**
    - If d(x,y) < D(x,y)
      - Replace D(x,y) with d(x,y) and write color of **p** into image

Ray casting

- Preview for next lecture
- Associate a ray with each pixel



- Find object-ray intersection points
- Choose closest point to the camera