13. Visible Surface Determination

Visible Surface Determination (= Hidden Surface Removal)

> 💡 Ray tracing 은 ray 를 쏴서 맞은 곳만 신경 쓰기 때문에, Visible surface determination 문제가 저절로 해결됨.
> 하지만, Rasterization pipeline 은 render 하는 모든 primitive 에 대해 Rasterization 을 하기 때문에, Visible Surface Determination 문제가 생김

- **Visible Surface Determination**
  = Hidden Surface Removal
  = Visibility

1️⃣ **Object-space approach (물체별로 결정,** Per-object visibility**)**

1. **Painter's algorithm**
2. **BSP tree method**
3. **Octree method**

2️⃣ **Image-space approach (픽셀별로 결정,** Per-pixel visibility**)**

1. **Depth(z) buffer method**
2. **Scan line method**

2️⃣ Image-Space Approach

1. Depth(z) buffer method

- Normalized
  - 주로 **min depth: 0, max depth: 1**로 normalize 하여 이용

    > 💡 depth: 눈으로부터의 거리를 의미

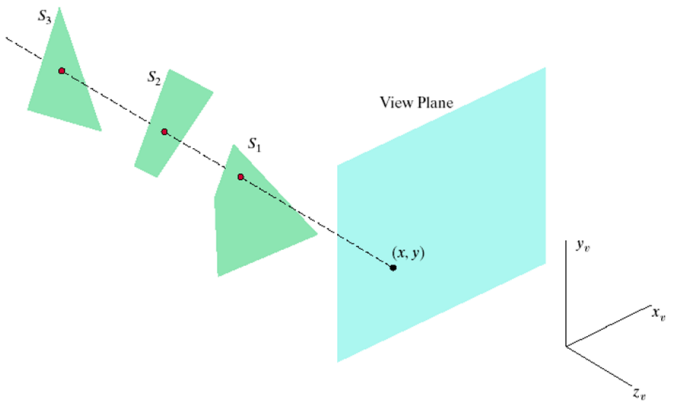
- Z-buffer 에 depth values 저장
  - (STEP 1) 모든 depth values 를 1로 initialize
    - 모든 물체가 viewpoint 에서 무한히 떨어져 있다고 초기화
  - (STEP 2) 각 다각형을 render 하면서 rasterize 된 pixel 의 depth 값을 비교

    > 💡 이때 rendering order 가 상관없음. (**Depth(z) buffer method** 의 장점)

  - (STEP 3) **z < depthBuff(x, y)**, depthBuff := z

    > 💡 z 가 z-buffer 에 있던 값보다 작으면, 더 가까운 픽셀이 발견되었다는 의미니까
    > (1) render 하고 (2) 기존 depth 를 갱신
    > 그렇지 않으면, pass

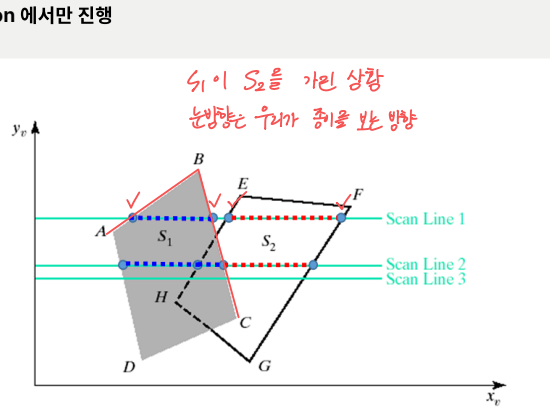
- GPU 를 이용하면 모든 pixel 에 대해 병렬적으로 계산하기에 매우 빠르다.
- (참고) depth 정보는 (x_h, y_h, z_h, h) → $(x_h/h, y_h/h, z_h/h, 1)$ 했을 때, z_h/h



2. Scanline Method

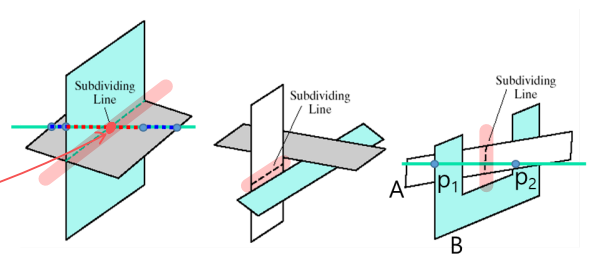
> 💡 Depth(z) buffer method 는 모든 pixel 에 대해 계산을 하는데, 이는 GPU 를 이용하여 병렬적으로 계산하기 때문에 가능
> 그런데 GPU 를 사용하지 못한다면 다른 방법이 필요 ⇒ **depth 비교를 모든 pixel 에서 하지 않고, scanline 과의 intersection 에서만 진행**

- 과정
  - (STEP 1) Scanline 과 만나는 edge 를 찾음 (**active edge**)
  - (STEP 2)
    - (scanline 1 의 경우)
      - s1 에서 scanline 과 처음 만나는 곳에서 depth 비교 ⇒ 이곳부터 보이는 물체는 s1
      - depth 비교 없이 s1 으로 채워나감
      - s1 polygon 이 끝난 시점에서 depth 비교 없이 끝냄
      - (s2 도 마찬가지)



- Surface subdivision 이 필요한 경우들 (Consistency 가 유지되도록 잘라야 함)
  - (경우 1) Surface cutting 이 있는 경우
    - 동일한 depth 를 가지도록 surface 를 자름
      그럼, 잘리면서 생긴 edge 부분도 depth checking
  - (경우 2) Cyclic overlap 이 생기는 경우
    - Cycle 이 생기지 않게 자름

  ⇒ 이렇게 예외 처리, 조건 처리가 들어가기에 병렬화가 어렵다.



1️⃣ Object-space approach

1. Painter's Algorithm

- 말 그대로 화가가 그림을 그리는 방식
- 눈으로부터 물체까지 거리를 계산하고, 가장 먼 물체부터 그린다.
- Algorithm
  1. Surface 를 decreasing depth 로 sort: depth 값이 큰 것부터 내림차순 정렬 (먼 것부터 정렬)
  2. Surface 가 순서대로 scan-converted 됨

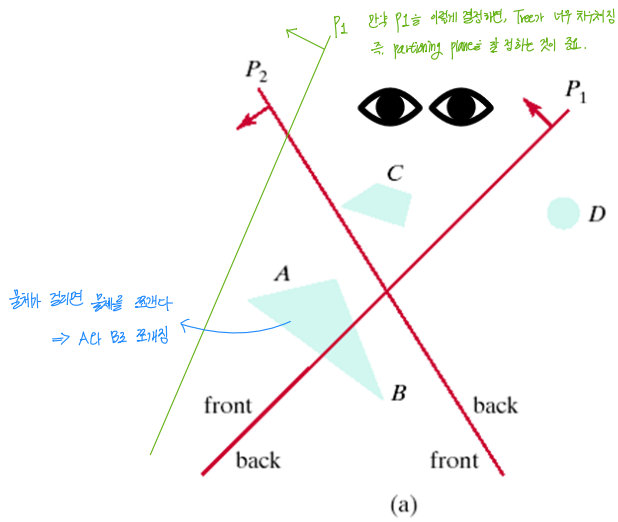
> 💡 (단점) cyclic loop 발생 시 infinite loop 발생

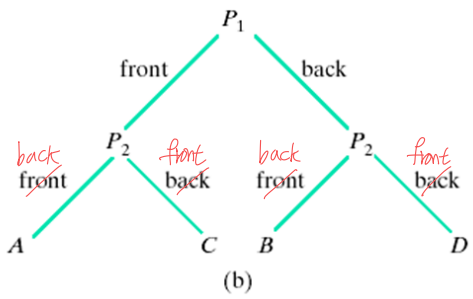
2. BSP (Binary Spatial Partitioning) tree Method

- **BSP-tree** (Binary Spatial Partitioning - tree)

> 💡
> - CG 분야에서 중요한 자료구조
> - Cycling overlap 같은 문제도 해결 가능
> - **공간을 두 개로 나눈다.**

- BSP-tree 예제 풀이
  1. P1 이 root 가 된다.
  2. P1 의 **수직 방향 법선 벡터에 속하는 부분을 front**, 그 **반대를 back** 이라 하자.
     - 이때 물체가 걸리면 물체를 쪼갠다.
  3. 그 결과 A, C 가 front // B, D 가 back 이 된다.
  4. 이제 P1 이 끝났으니, P2 에서 동일하게 반복한다.
- 공간을 나눌 때 Partitioning plane 을 잘 잡아야 한다. (예제에서는 P1, P2 로 이미 결정됨)
- BSP-tree Method 요약
  - 아이디어는 1. Painter's Algorithm 과 유사하다. (먼 물체부터 그리기)
  - 이때, 내 눈으로부터 먼 것이 어떤 물체인지 BSP-tree 가 알려준다.
- **BSP-tree Method**
  1. **BSP-tree 만들기**
  2. **Relabel: 현재 눈의 위치를 기준으로 눈이 있는 쪽이 front, 없는 쪽이 back 으로 재정의**
  3. **Render: back-to-front**. Root 부터 시작해서 back 부터 방문 (ex. **B→D→A→C**)





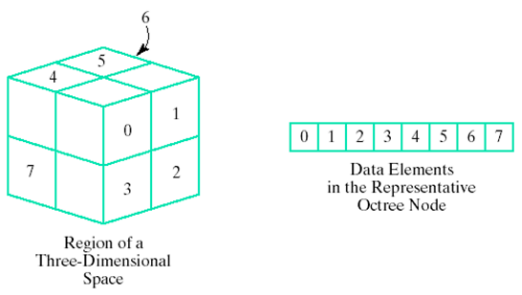
> 💡 **BSP-tree Method 장점**
>
> : 눈의 위치에 따라 edge 에 있는 label 은 바뀔 수 있으나, 기본적으로 (대부분의 물체는 고정이므로) BSP-tree 의 topology 가 바뀌지는 않는다.
>
> ⇒ 물체는 static, viewpoint 는 dynamic 하게 바뀔 때 자주 사용하는 방법 (ex. 3D FPS 게임)

3. Octree Method

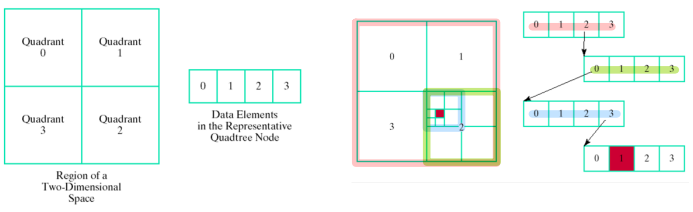
- **Octree** in 3D (Oct = 8)
  - BSP-tree 보다도 더 중요한 자료구조
  - 이전에 Ray tracing 의 Acceleration 중에서, 공간을 균일하게 나누는 Spatial subdivision 배웠음

> 💡 Spatial subdivision은 물체가 있든 없든 같은 크기로 나누다 보니까, 쓸데없는 곳 (물체가 없는 곳) 까지도 나누었음 → memory 비효율적
>
> ⇒ 이를 해결한 게 Octree

  - **Non-uniform spatial subdivision** 을 위한 hierarchical tree structure
    - 8로 계속 쪼개는데, 모든 child 를 쪼개지 않고 필요할 때만 쪼갠다. ⇒ 비균등 분할
  - 각 node = octant 는 3D 공간의 영역을 의미



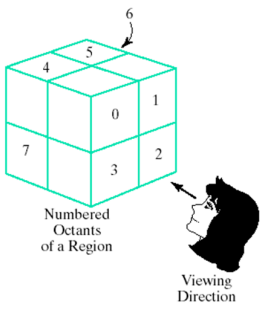
- **Quadtree** in 2D (Quad = 4)

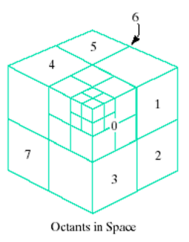


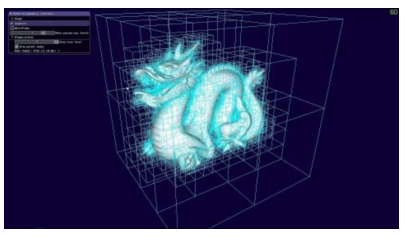
- **Octree Method**

> 💡 back to front 로 기껏 멀리 있는 물체부터 렌더링을 마쳤더니, 나중에 앞쪽 물체에 의해 가려질 수 있음
>
> 결과적으로 보이지도 않을 픽셀들을 불필요하게 처리하는 문제 발생
>
> ⇒ **front to back**: 가까운 물체부터 그리기

  - **front to back**. Depth First Traversal (DFS)
    - Depth first = 0 노드가 분할되어 있다면, 그 노드의 모든 하위 노드를 먼저 다 그린 다음에야 다음 노드 1로 넘어간다.







    - Leaf node 에 도달했으면 rendering
- **GPU-accelerated occlusion query**
    - Leaf node 에 도달했으면 rendering 할 때, GPU 를 통해 occlusion query 를 call 해서 현재 rendering 하려는 부분이 보이는지 안 보이는지 확인이 가능하다.

> 💡 (참고) 결국 가려지는 물체를 render 하지 않는 것이 목표인데, 가려지나 안 가려지나는 render 를 하고 나야 판단 가능하다. 그럼, occlusion query 는 어떻게 한 것일까?
>
> **⇒ 물체 자체를 rendering 하는 것이 아니라, octree node (octant) 를 rendering 하는 것**
>
> Zero cost 까지는 아니지만, 육면체의 면 6개만 render 하면 되므로 물체 rendering 보다는 시간이 적게 걸린다.

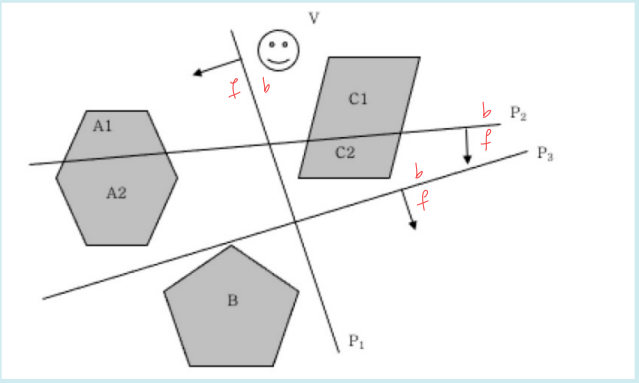
Comparisons

> 💡 Depth Buffer 방법이 가장 자주 이용된다. (GPU 가 이 방법을 지원하기 때문에)

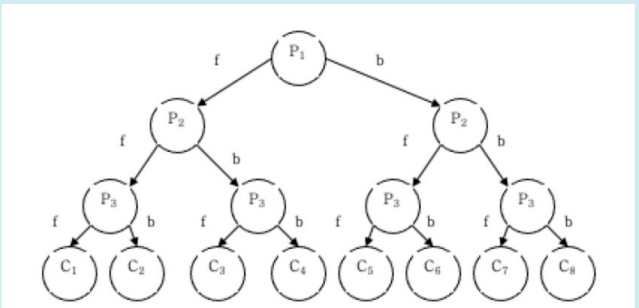
- **Very little depth overlap** → 물체들이 가리는 게 적으면 object space approach
    - Painter's algorithm or BSP-tree method
- **Few depth overlap** → 물체들이 가리는 게 좀 있으면 image space approach
    - Scan-line method
- **Few surfaces**
    - Painter's algorithm, BSP-tree method, scan-line method
- **Many surfaces**
    - Depth buffer method, octree method
- **Multiple views** → 물체는 static, viewpoint 는 dynamic 하게 바뀔 때 BSP-tree
    - BSP-tree method

Quiz) Visible Surface Determination

There exist three objects, A, B, and C in 3D space. Three planes P_1, P_2, P_3 partition the space and further split A into A1 and A2, and C into C1 and C2, as shown below.



The BSP-tree induced by the above situation can be described as below:

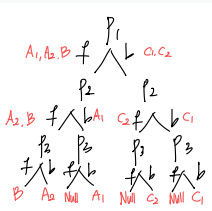


문제 1
정답
총 8.00 점에서 8.00 점 할당
문제 표시

Fill the leaf nodes from C_1 to C_8 in the BSP tree with objects, A1, A2, B, C1, C2, and NULL.

C_1	B ✔
C_2	A2 ✔
C_3	NULL ✔
C_4	A1 ✔
C_5	NULL ✔
C_6	C2 ✔
C_7	NULL ✔
C_8	C1 ✔

정답 : C_1 → B, C_2 → A2, C_3 → NULL, C_4 → A1, C_5 → NULL, C_6 → C2, C_7 → NULL, C_8 → C1



문제 2
정답
총 5.00 점에서 5.00 점 할당
문제 표시

When the viewpoint is located at V, what is the rendering order for the objects using the BSP tree?

A1	3 ✔
A2	2 ✔
B	1 ✔
C1	5 ✔
C2	4 ✔

정답 : A1 → 3, A2 → 2, B → 1, C1 → 5, C2 → 4