빠른 렌더링을 위한 오브젝트 제외 기술

유영천

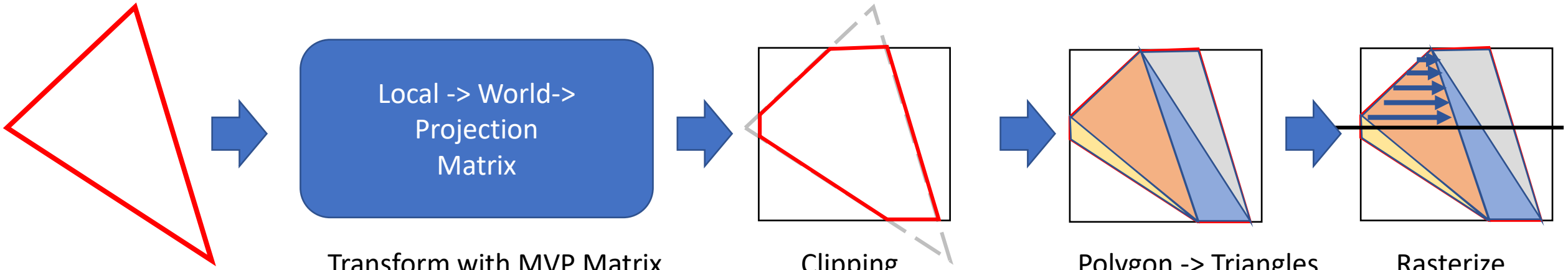
https://megayuchi.com

tw:@dgtman

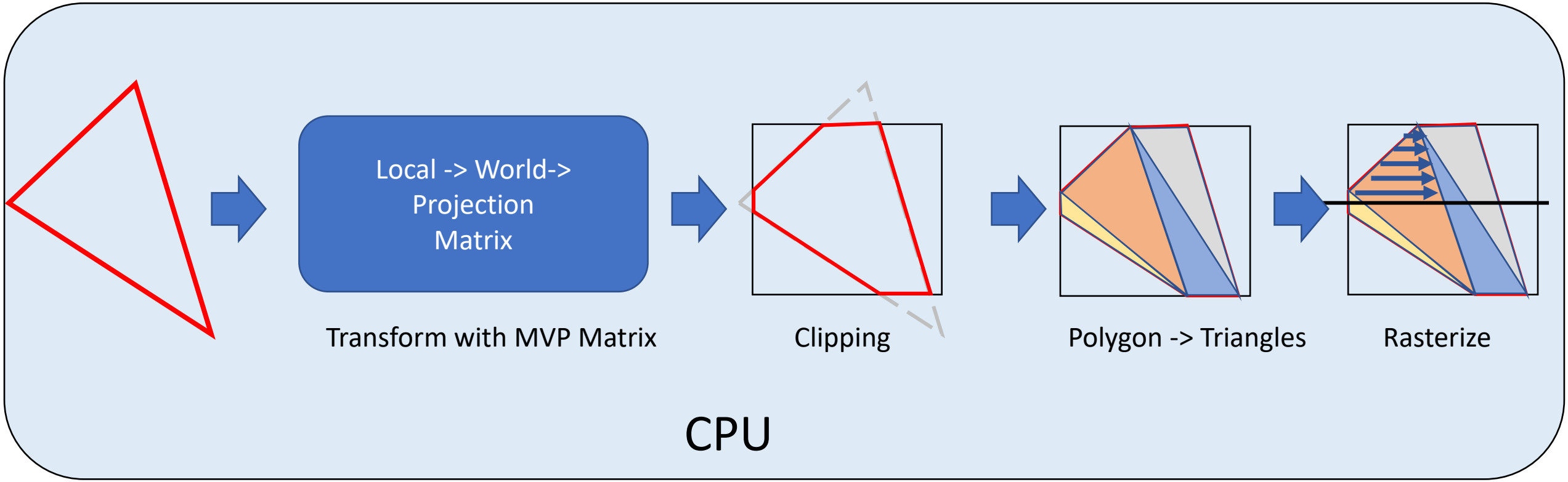
그래픽스 기본

삼각형이 화면에 그려지기까지...

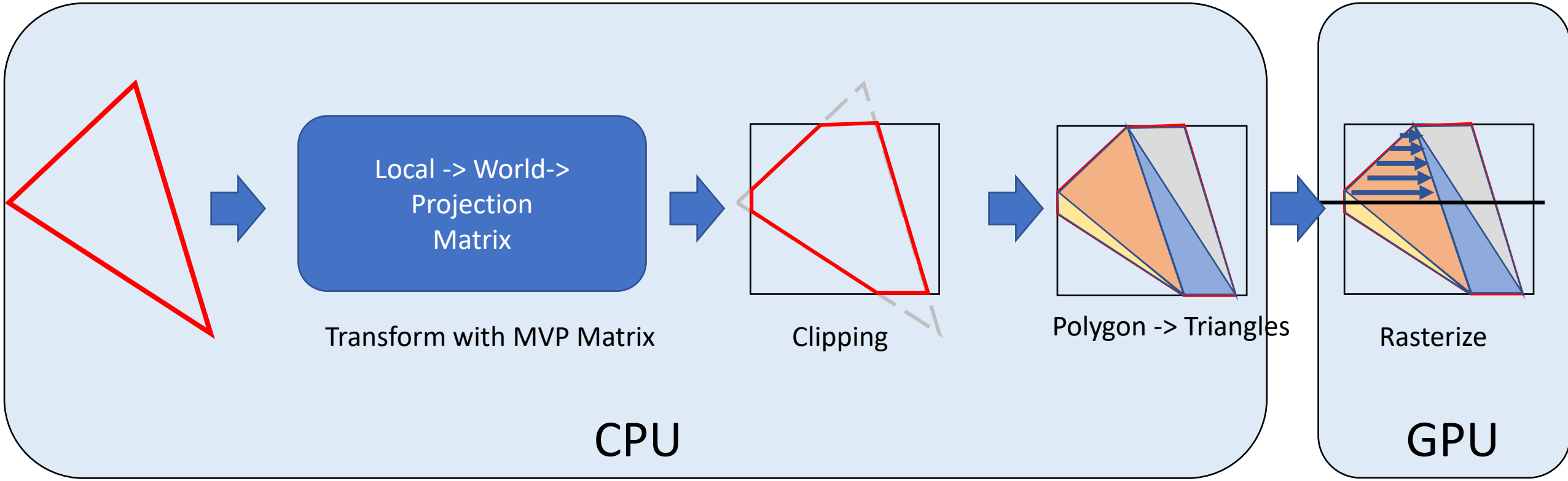
삼각형을 그리는 과정


Local -> World->
Projection
Matrix
Transform with MVP Matrix
Clipping
Polygon -> Triangles
Rasterize

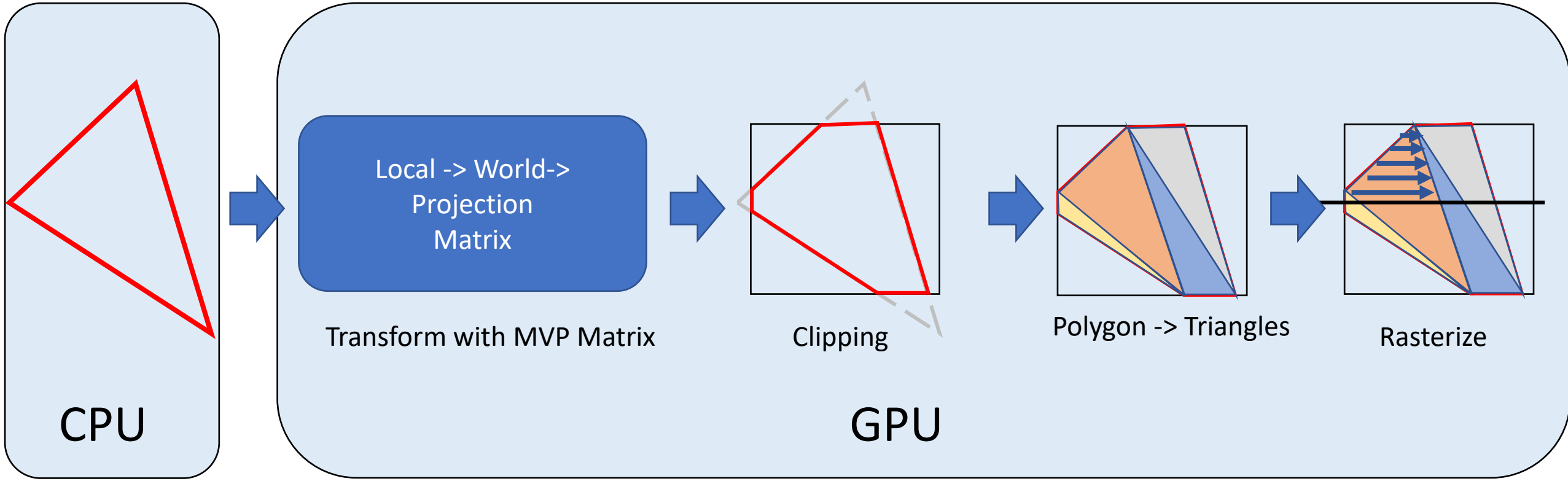
삼각형을 그리는 과정 – CPU로만 처리하던 시절



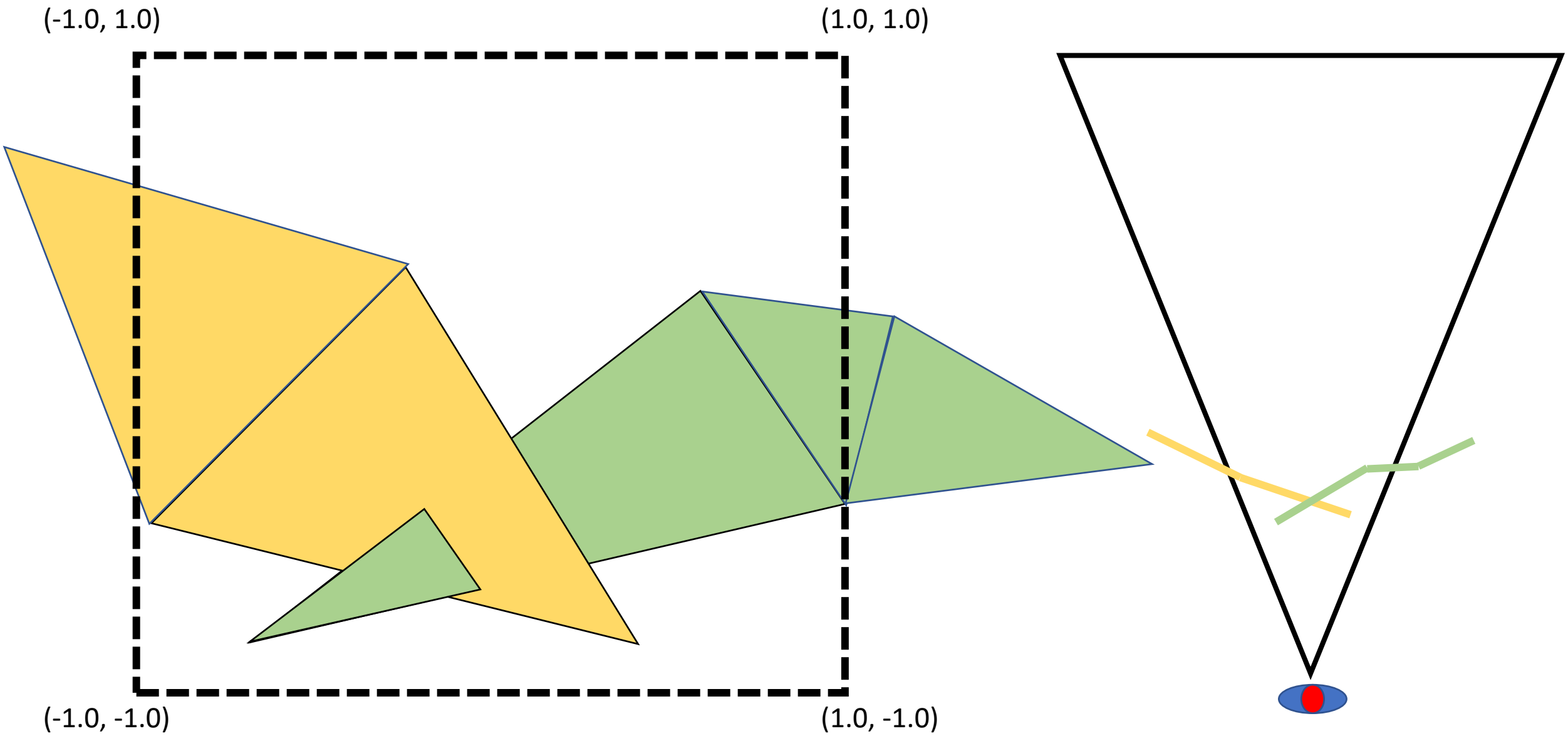
삼각형을 그리는 과정 – rasterize만 GPU로 처리하던 시절



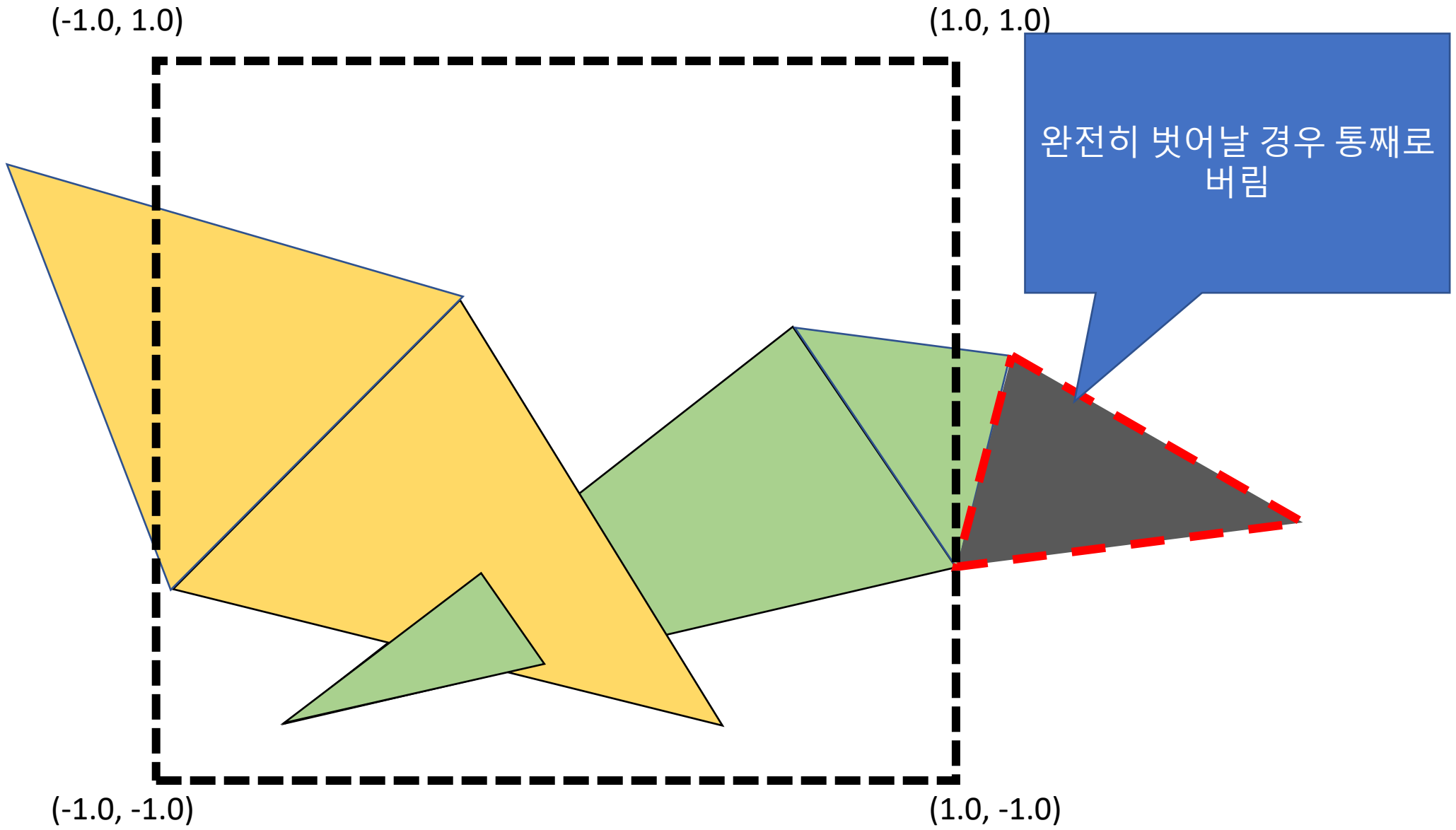
삼각형을 그리는 과정 – 대부분 GPU로 처리하는 시절



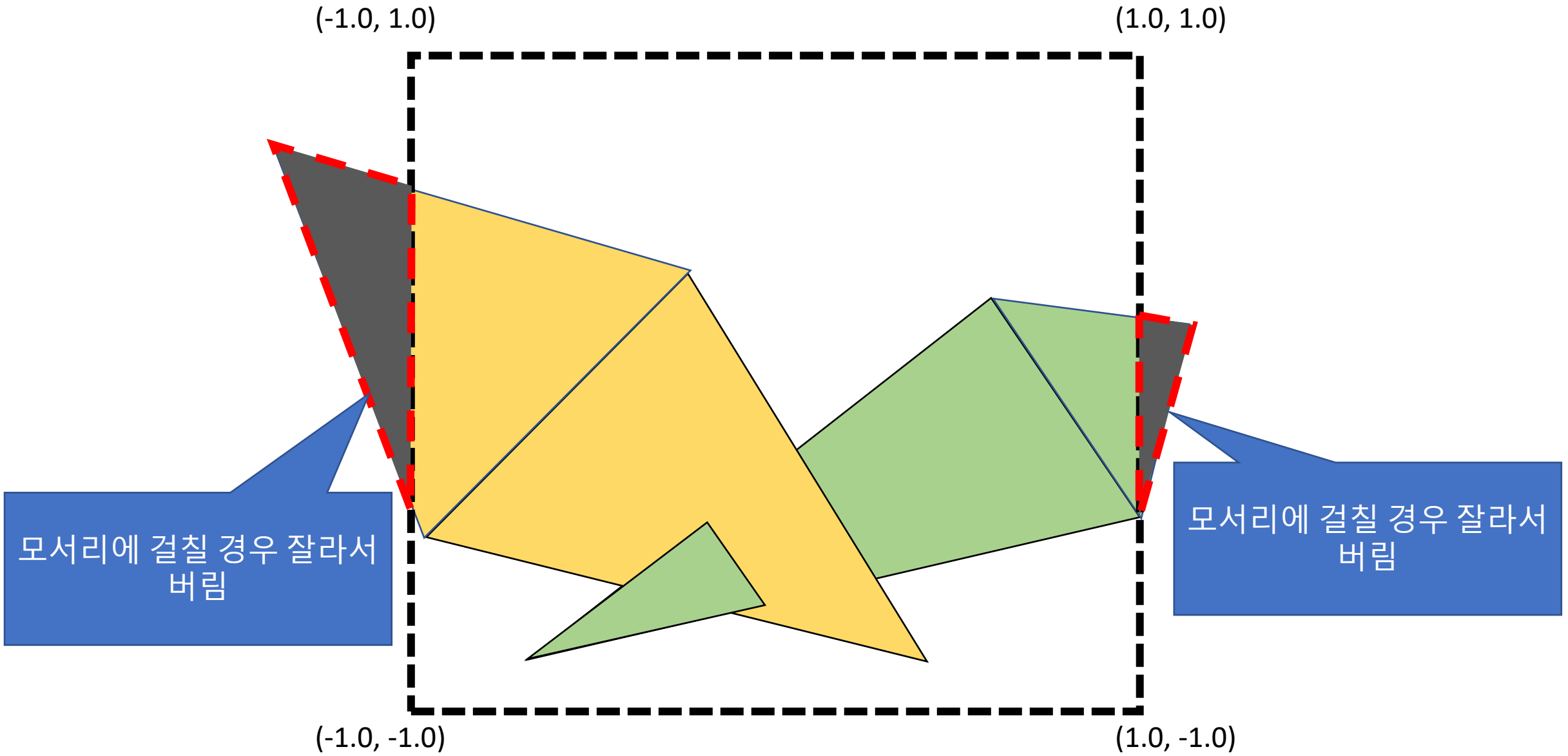
Rasterize과정에서 보이지 않는 삼각형 제거 과정

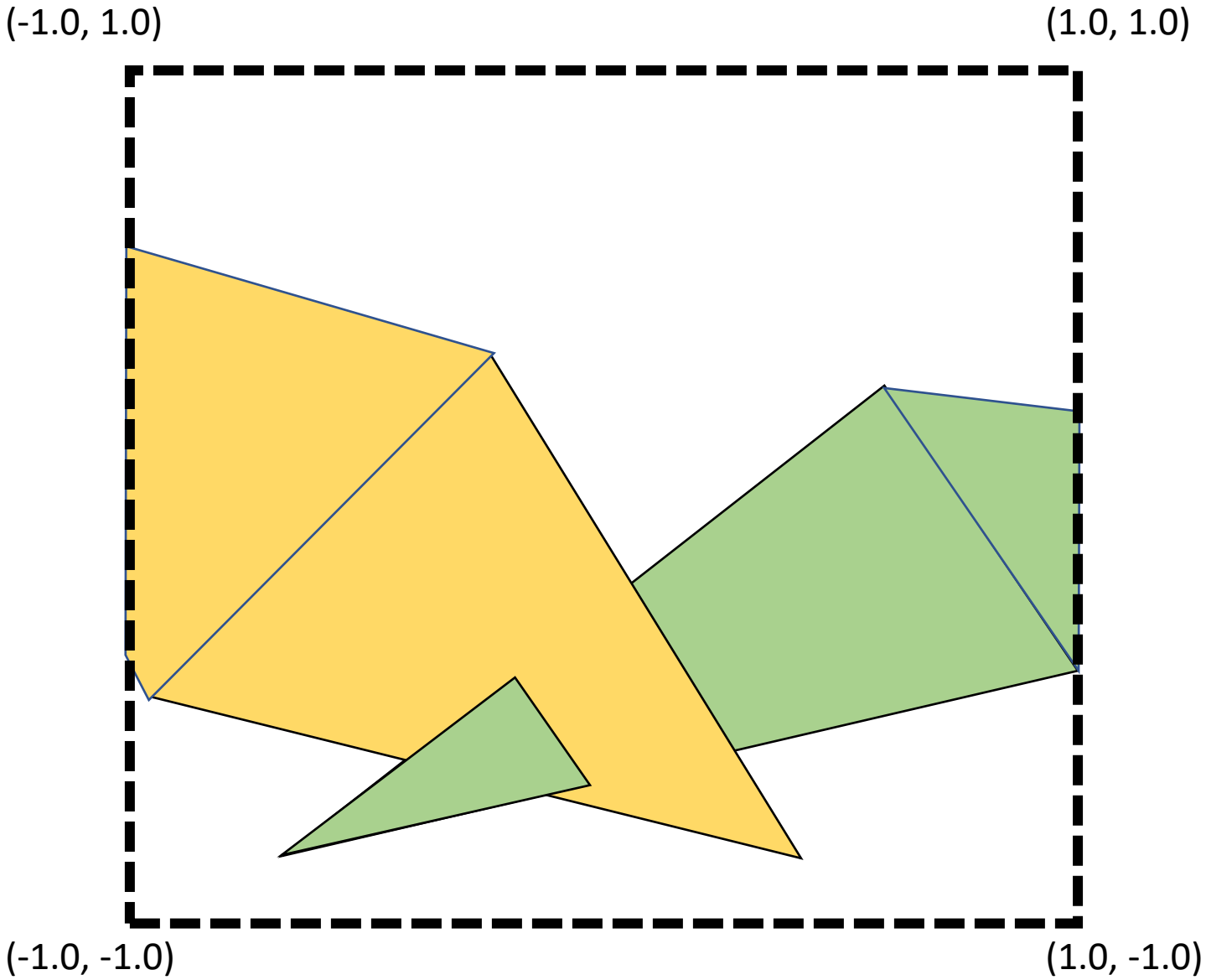


Rasterize과정에서 보이지 않는 삼각형 제거 과정



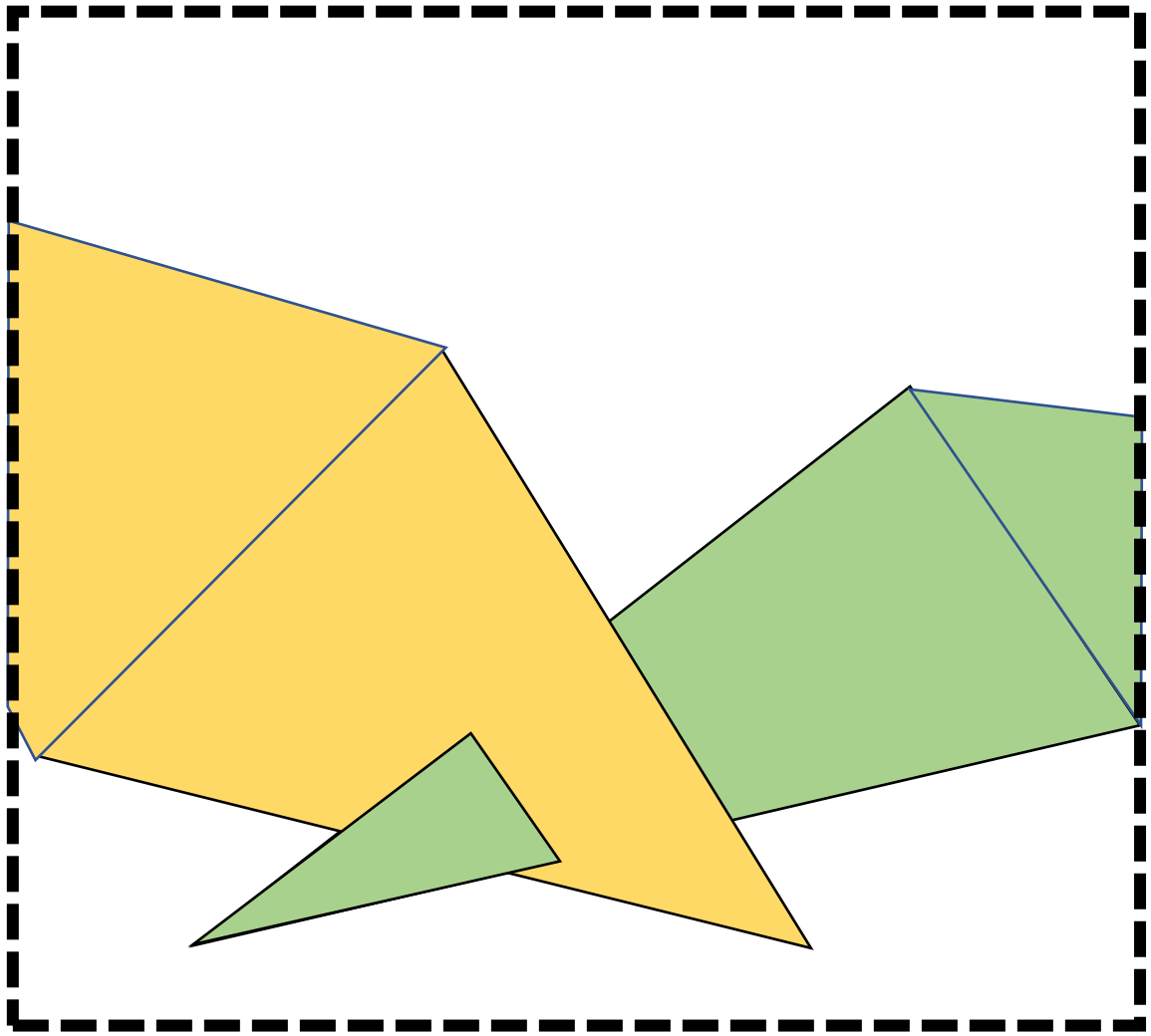
Rasterize과정에서 보이지 않는 삼각형 제거 과정




(-1.0, 1.0)
(1.0, 1.0)
(-1.0, -1.0)
(1.0, -1.0)

NDC

(-1.0, 1.0) (1.0, 1.0)



(-1.0, -1.0) (1.0, -1.0)

Z-Buffer

(0, 0) (Width, 0)

1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.45	0.45	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.44	0.44	0.45	0.46	0.46	1.0	1.0	1.0	1.0	1.0	1.0
0.46	0.45	0.46	0.46	0.45	1.0	1.0	1.0	0.48	0.48	0.48
0.46	0.46	0.46	0.45	0.44	1.0	1.0	0.47	0.48	0.48	0.48
0.47	0.46	0.46	0.45	0.44	0.43	0.46	0.47	0.47	0.48	0.48
0.47	0.46	0.45	0.44	0.43	0.43	0.46	0.46	0.47	0.48	0.48
1.0	1.0	0.42	0.43	0.43	0.43	0.43	0.45	0.46	1.0	1.0
1.0	1.0	0.41	0.42	0.43	0.43	0.42	0.41	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

(0, Height) (Width, Height)

GPU에 다 맡기면 되는데 무엇이 문제인가?

GPU측 부하

- 그래픽 리소스도 무거워진다 – 프레임 버퍼와 텍스처 해상도 UP - vertex processing보다 pixel processing이 문제
- Z-Buffer에 읽기/쓰기 병목발생

CPU측 부하

- 그래픽 API호출 비용이 생각보다 크다.
- 그래픽 API의 draw직전까지 셋업 비용 – SetBlendState…, SetDepthStencilState… , SetVertexShader…, SetTexture…,
  - 그래픽 API런타임 내부처리 비용
  - 드라이버의 커맨드 전송 비용

부하 줄이기 - 성능향상

프레임 레이트를 높이려면?

- 빠르게 그리기(오늘의 주제가 아님)
  - 같은 텍스처 사용하는 버텍스들 같은 draw call로 묶기
  - 상태변경 API 호출 최소화
  - LOD(Level of Detail) 사용
  - Shader 최적화
  - 기타 등등…

- 되도록 안그리기

되도록 안그리는 방법들

공간분할

- 어떤 방법이든간에 월드를 작은 단위의 공간으로 나눠야 한다.
- Grid, Tree등
- 장면관리를 위해서라도 반드시 필요하다 (3D그래픽스 엔진의 꽃이라 할 수 있다).

공간분할

- 삼각형 매시 기반의 BSP – 삼각형의 평면을 그대로 공간을 자르는 기준으로 사용 – Quake시리즈, Quake기술을 사용한 게임들
- KD-Tree – 축정렬된 2진트리
- Quad Tree – Y축 고려하지 않고 X,Z축으로 사각형 노드들로 분할
- Octree – Y축을 고려하여 X,Y,Z축으로 육면체 노드로 분할
- 2/3차원 Grid – 구조적으론 Quad Tree, Octree와 비슷. 배열을 사용한다는 점이 다르다.

공간분할에 종속적인 기법

- BSP / PVS – Quake시리즈 , Quake엔진 기술을 사용한 게임들
- Room / Portal – 언리얼, Project D Online(by megayuchi)
- PVS with GPU – Project D Online(next edition , by megayuchi)

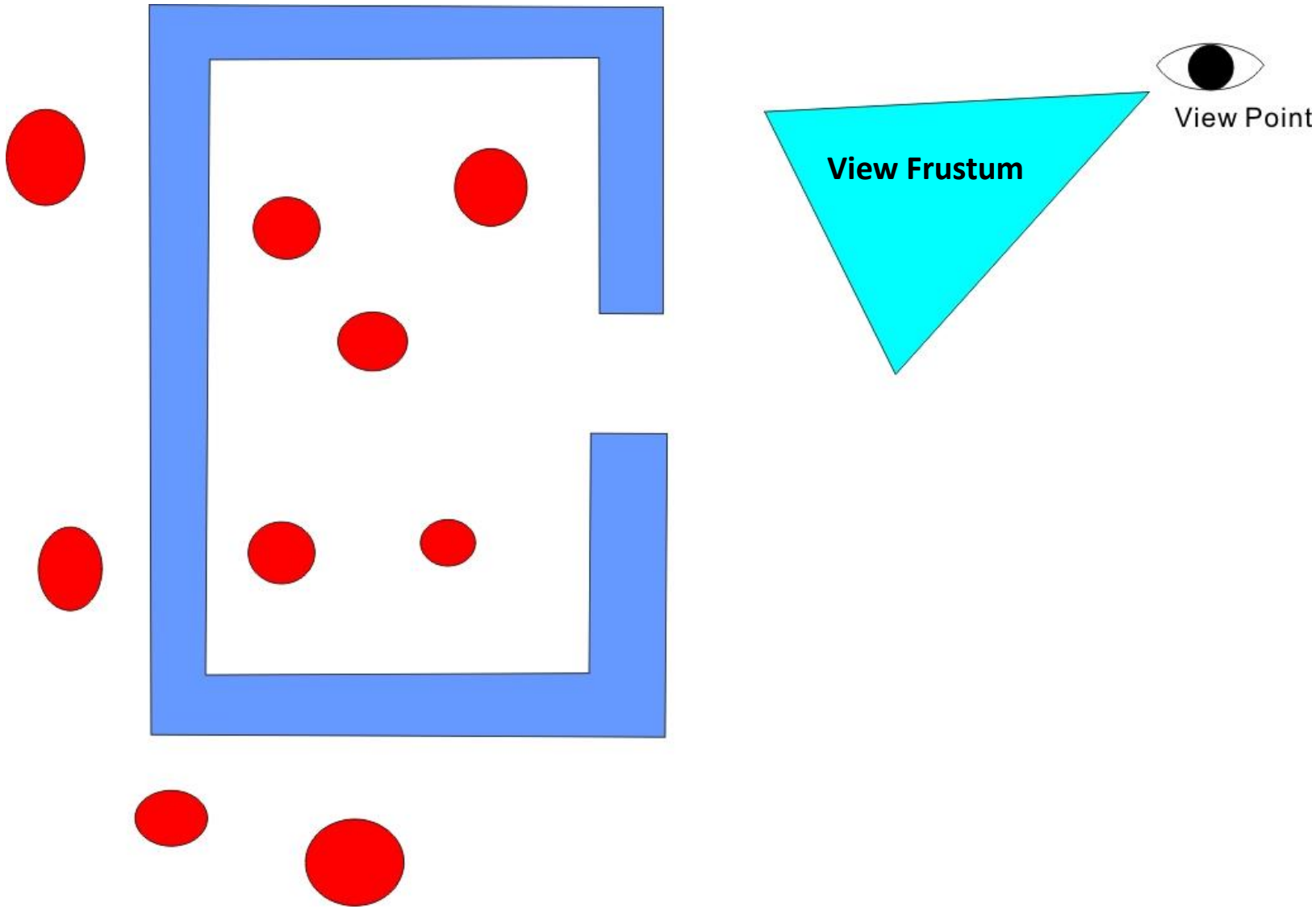
공간분할에 종속적이지 않은 기법

- View Frustum Culling
- 수동 Occlusion Culling
- S/W Occlusion Culling (Raster / Test)
- H/W Hierarchical Occlusion Culling
- D3DQuery

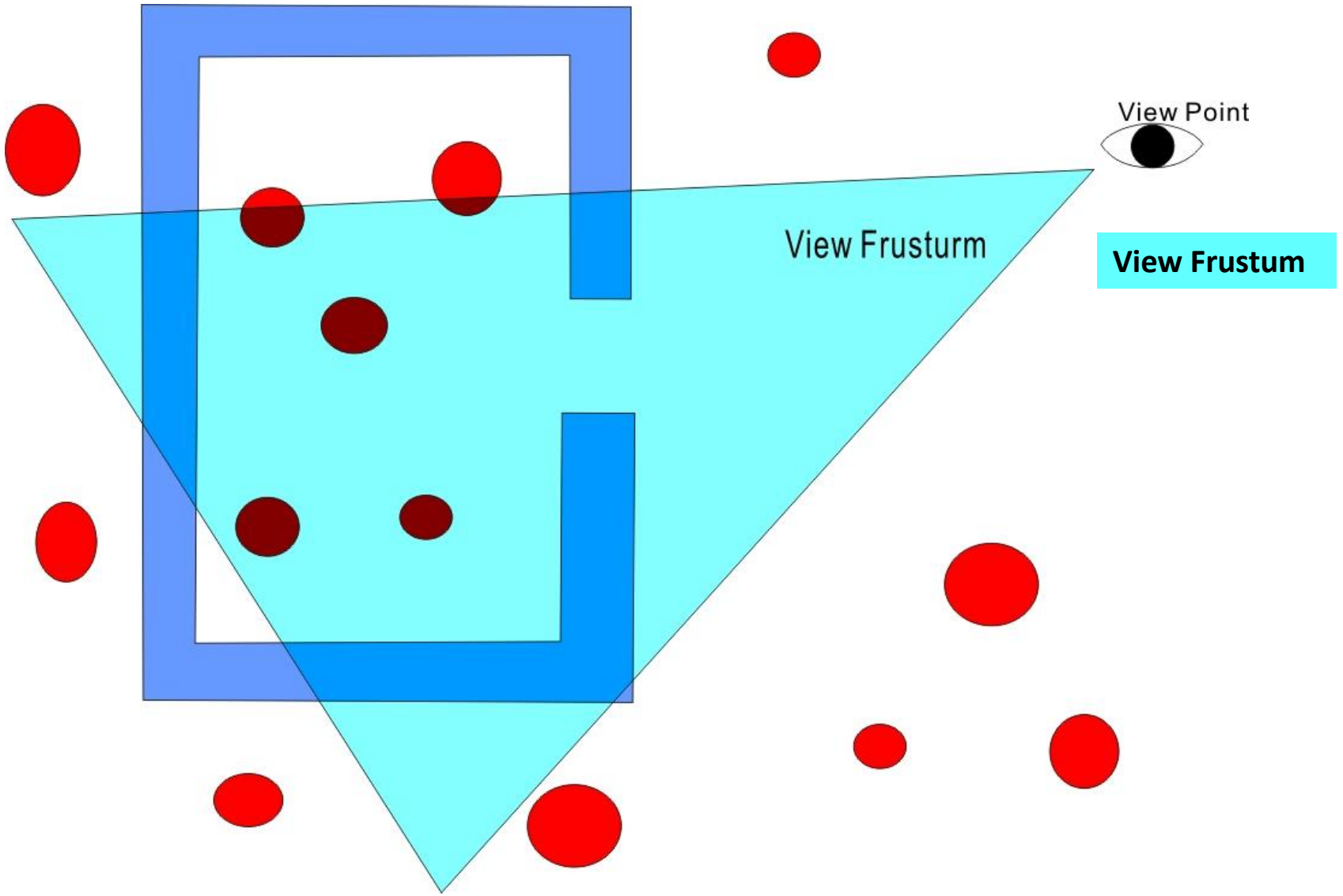
공간분할에 종속적이지 않은 기법도 사실은 공간분할 여부/방법에 따라 많은 성능 차이를 보인다.

오브젝트 제외기법 간단 요약

Culling 없음. GPU에 전적으로 의존. 문제없음. 단지 느릴 뿐.



공간분할 없이 View frustum Culling.



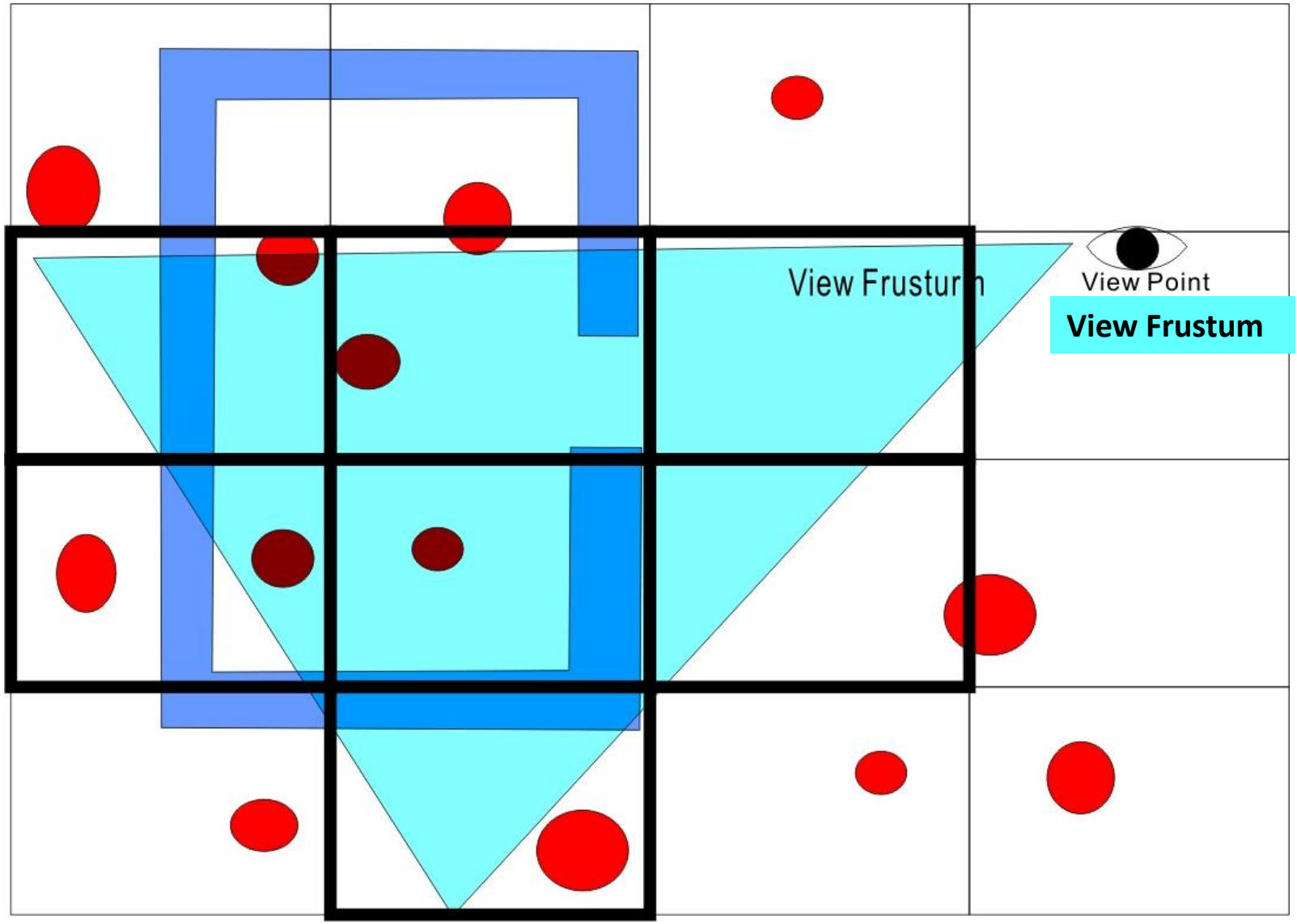
GPU부담 감소
그래픽 API호출 비용 감소

오브젝트 개수가 많으면 frustum culling 비용 자체가 증가

차폐물에 가려지는 오브젝트에 대해선 culling불가.

가장 만들기 쉽다.

View frustum culling + Quad Tree or Octree or KD-Tree



GPU부담 감소
그래픽 API호출 비용 감소

트리 덕분에 frustum culling비용 감소에 효과가 있다.

차폐물에 가려지는 오브젝트에 대해선 culling불가.

비교적 만들기 쉽다

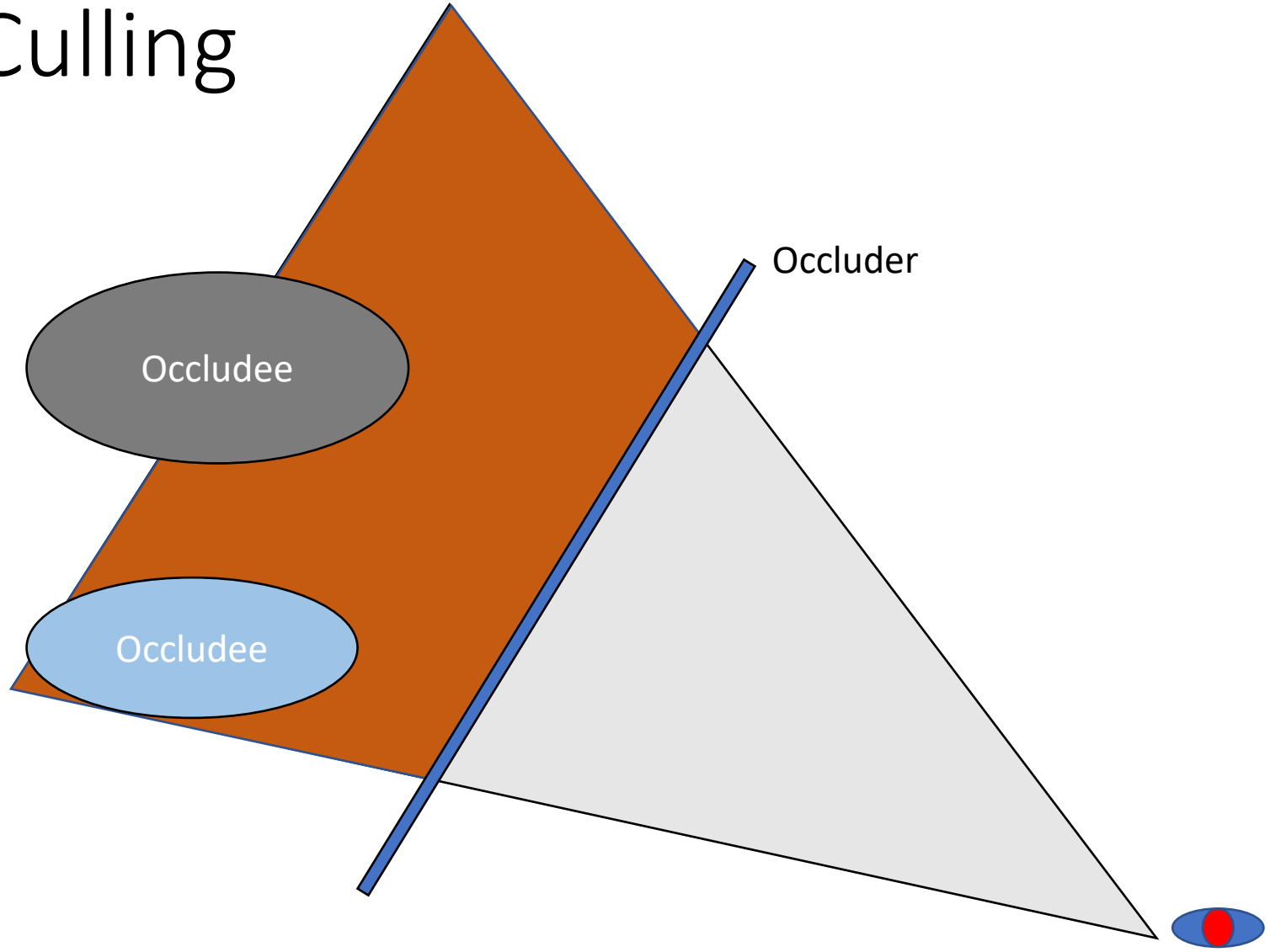
Occlusion Culling

- HW Occlusion Culling
  - D3DQuery
  - Hierarchical Occlusion Culling with Compute Shader

- SW Occlusion Culling
  - 앞서 설명한 Z-Buffer raster/test를 CPU에서 수행

- 수동 Occlusion Culling
  - 디자이너가 맵 위에 판대기를 직접 배치
  - 카메라의 eye로부터 판대기까지 View Frustum을 만들어서 여기 들어가는 오브젝트들은 렌더링 파이프라인에서 제외

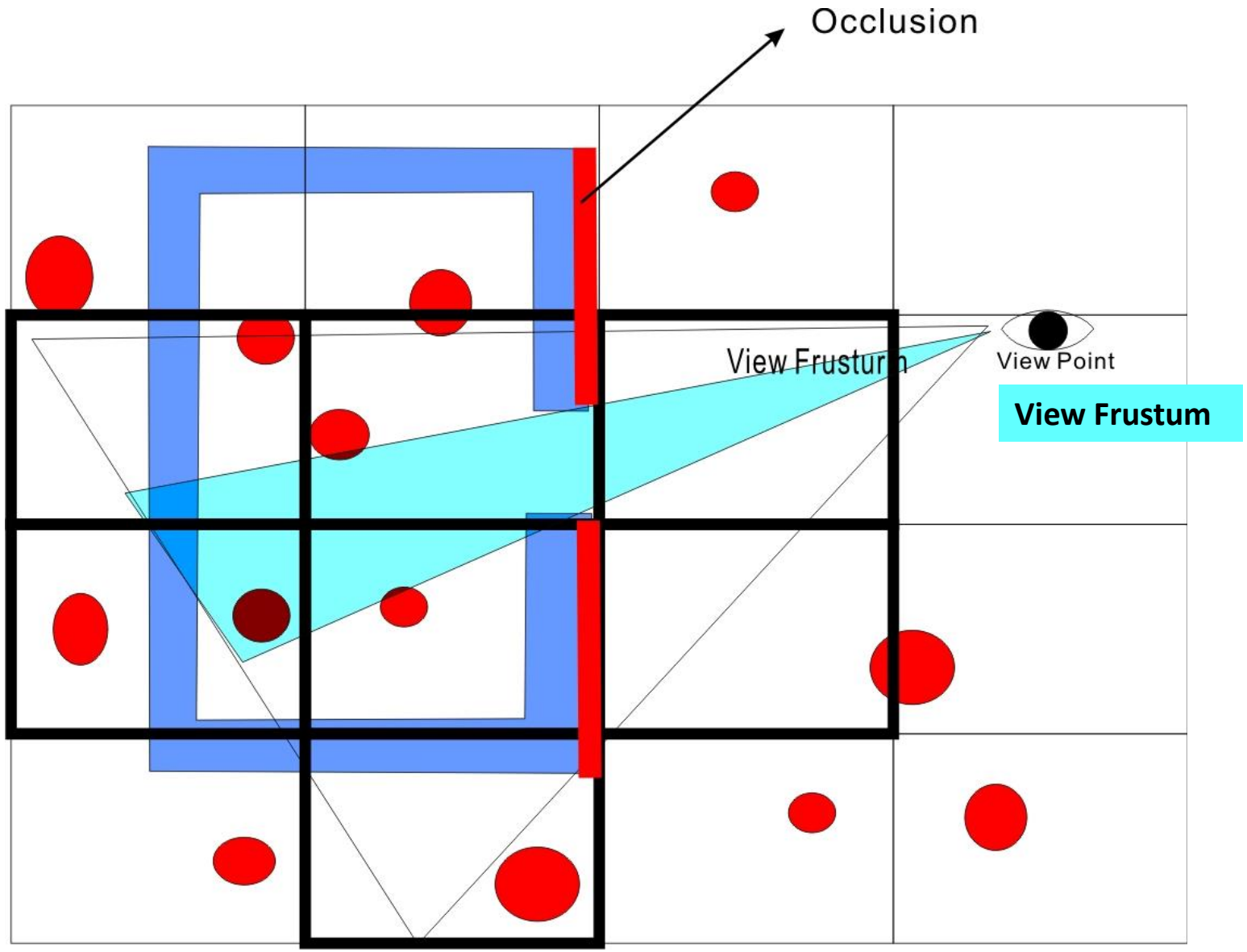
Occlusion Culling



수동 Occlusion Culling

- 디자이너가 맵에 Plane(판대기) 배치
- 런타임에 카메라의 Eye포인트와 판대기를 연결한 뷰프러스텀 계산-Near Plane은 Occlusion면이 되어야함.
- 이 뷰프러스텀에 들어가는 오브젝트는 렌더링 안함.
- 은근 실제 게임에 많이 쓰인다.

View frustum culling + Quad Tree or Octree + 수동 Occlusion



GPU부담 감소
그래픽 API호출 비용 감소

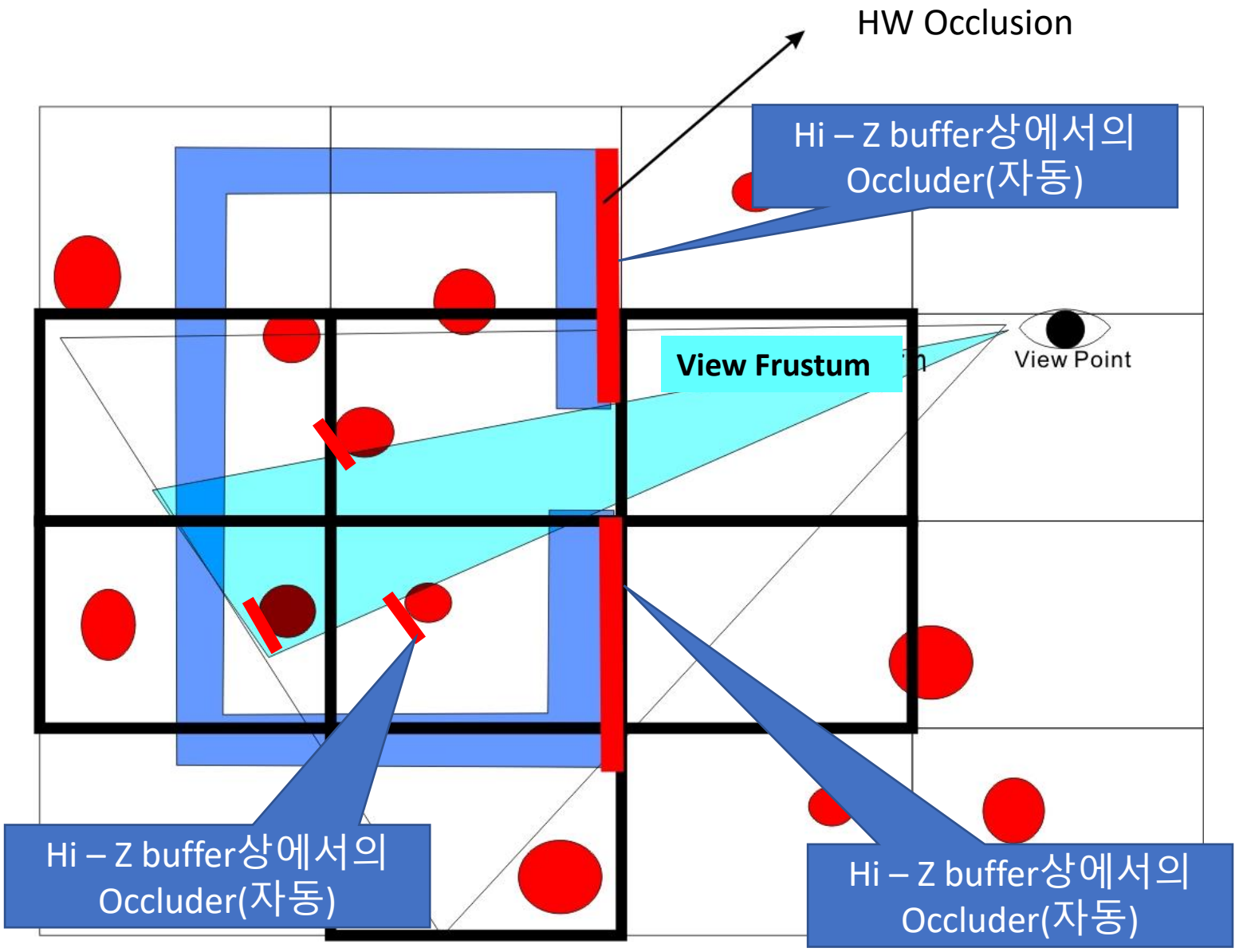
트리 덕분에 frustum culling비용 감소에 효과가 있다.

차폐물에 가려지는 오브젝트들을 어느 정도 culling가능하다.

프로그래머 입장에선 비교적 만들기 쉽다

아트 입장에선 짜증난다.

View frustum culling + Quad Tree or Octree + HW Occlusion Culling



GPU부담 감소
그래픽 API호출 비용 감소

트리 덕분에 frustum culling비용 감소에 효과가 있다.

차폐물에 가려지는 오브젝트들을 상당수 제외시킬 수 있다.

프로그래머 입장에서 구현이 어렵지는 않다.

아트 디자이너의 추가 작업 필요없다.

어느 정도 자동화된 Occlusion culling 제공.

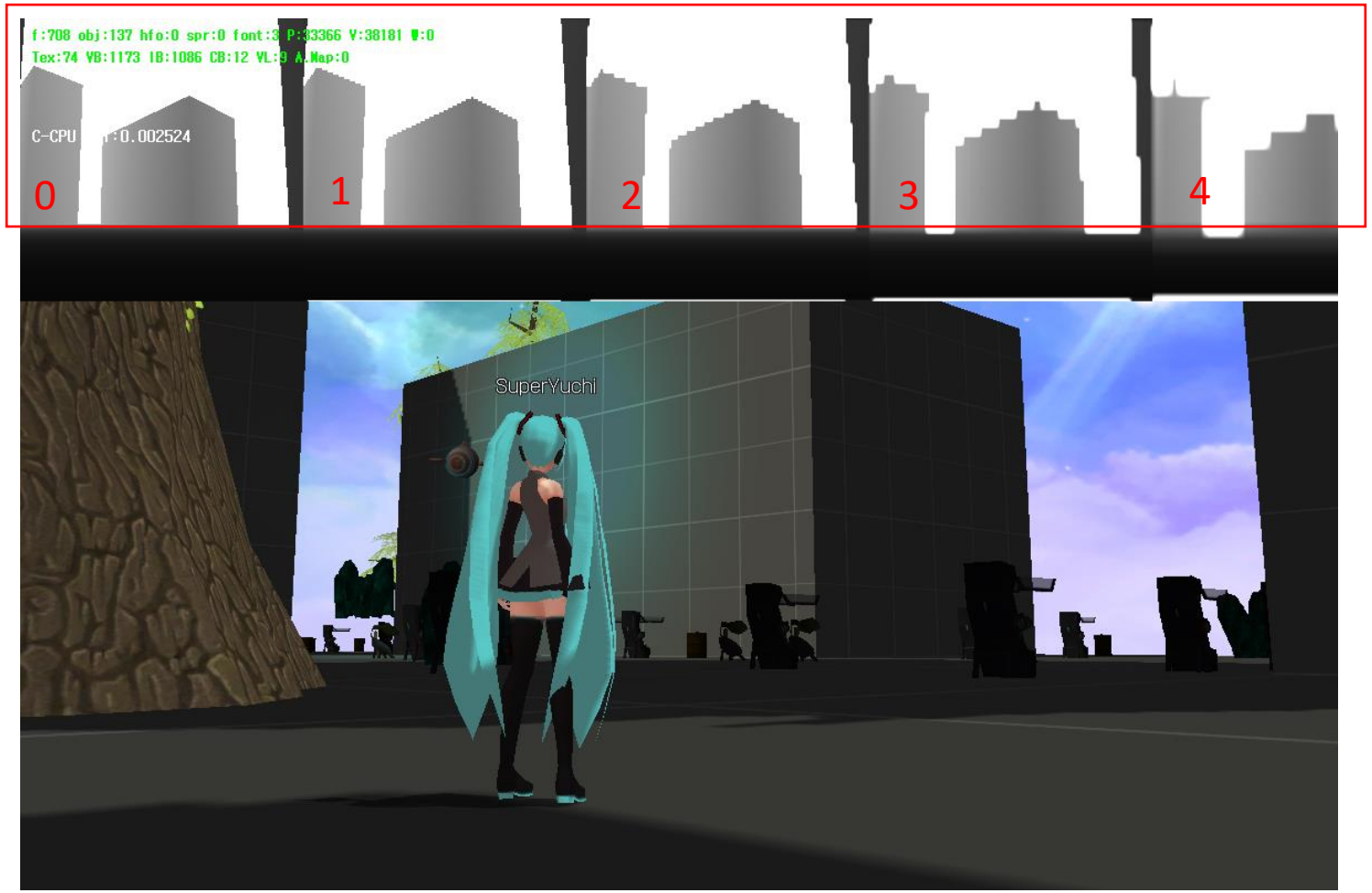
그래픽 API를 이용한 H/W Occlusion Culling

- Occluder(raw맵 데이타나 거의 꼭 맞는 바운딩매시)를 그려서 Z-Buffer를 구성한다.
- Occludee(아마도 캐릭터나 맵데이타,기타 배치 가능한 오브젝트)를 그리되 pixel은 write하지 않고 z-test만 한다.
- 그려지는 픽셀 수를 GPU측 메모리에 기록해둔다.
- 나중에 GPU로부터 픽셀수를 얻어온다.
- 보조수단으로만 권장

Hierarchical Z Map Occlusion Culling

- 정말 그려지는지 테스트-기본 아이디어는 같다.
- Occluder와 Occludee가 꼭 정밀해야하는가? 간단한 매쉬(경계구)로 대치하자.
- 그렇다면 Z-Buffer의 해상도도 정밀할 필요가 없다.
- 낮은 해상도의 Z-Buffer를 사용하면 Z-Buffer의 읽기/쓰기 시간을 줄일 수 있다.
- 경계구 가지고 테스트할건데 실제 그려볼 필요나 있을까? 아예 그리지도 말고 구의 한 점만 프로젝션 해서 depth값을 비교하자.
- 직접 버퍼에 그릴 필요가 없다면 Pixel Shader쓸 필요도 없으니 Compute Shader로 경계구 하나마다 스레드를 할당해서 depth테스트를 병렬화 시키자.

Z-buffer Texture mip chain

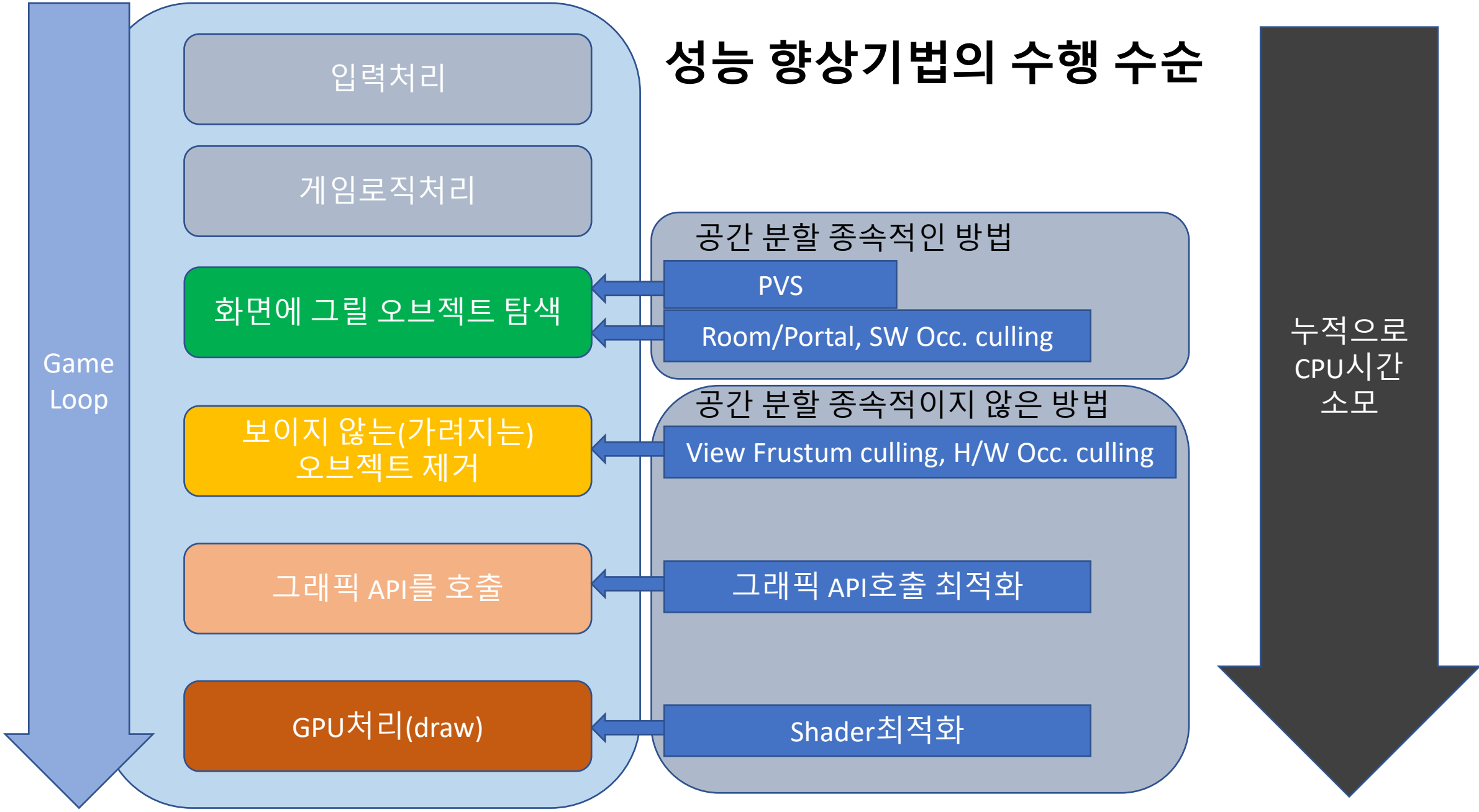


H/W Occlusion Culling

- 이미 발표했던 내용이라 링크로 대신합니다.
- https://www.slideshare.net/dgtman/hierachical-z-map-occlusion-culling

좀더 근본적인 방법

성능 향상기법의 수행 수순



앞 단계에서 많은 오브젝트들을 제거할 수록 성능이 향상될 가능성이 높다.

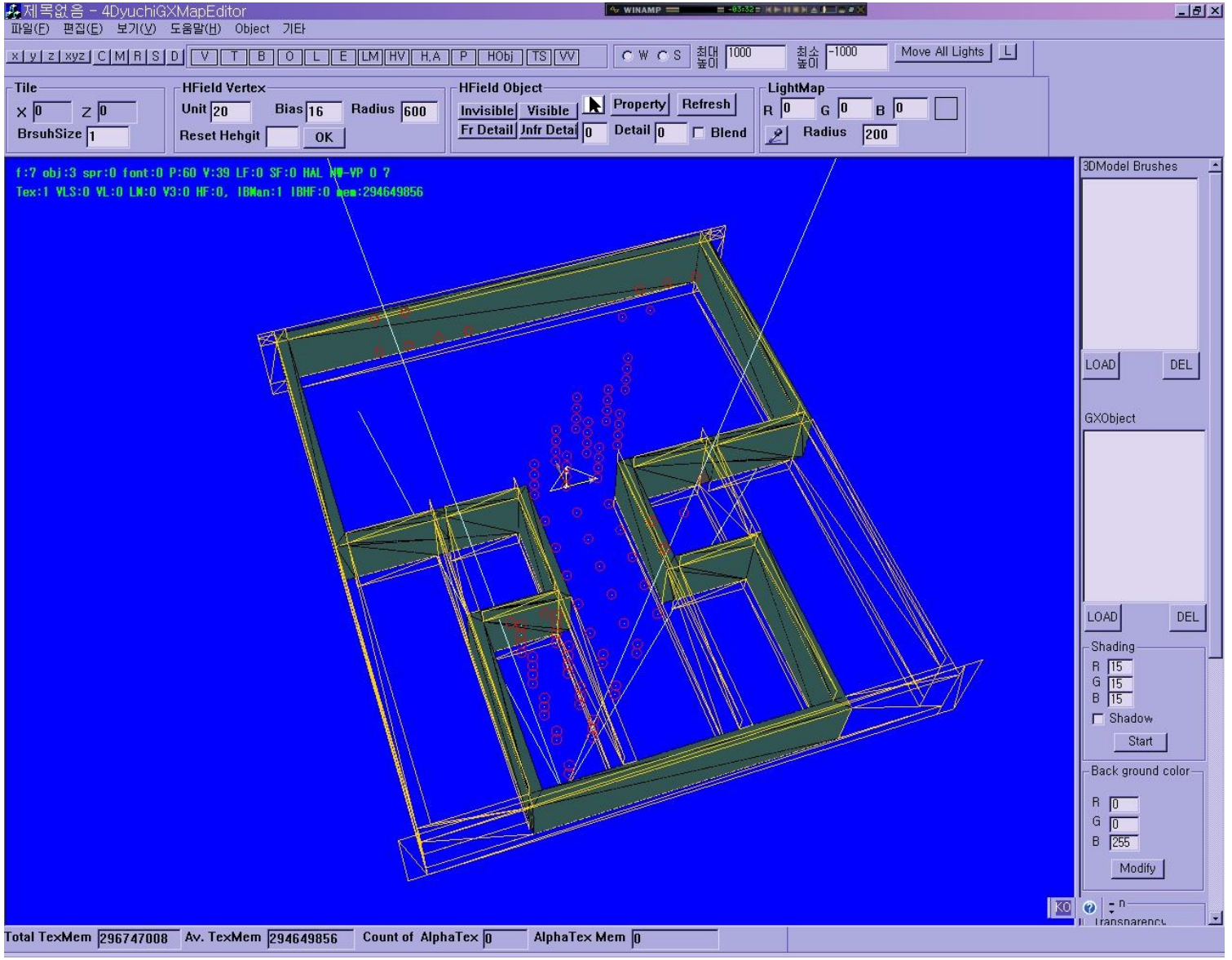
PVS(Potentially Visibility Set)

- 임의의 공간에서 월드상의 어떤 공간(혹은 오브젝트)들이 보이는지 미리 계산해둔다.
- 카메라의 위치가 주어지면 거의 상수시간 O(1)으로 보일 오브젝트들을 찾아낼수 있다.
- 고전적인 BSP/Portal방식을 쓰던, GPU를 사용하던 뭐가 됐든 미리 계산만 해둘 수 있으면 된다.

BSP/PVS (Quake엔진)

- 맵의 삼각형들을 그대로 사용해서-삼각형의 평면으로-월드를 잘라나간다.
- 자를때마다 왼쪽/오른쪽 노드로 다각형들을 나눠담는다.
- 최종적으로 모든 삼각형들은 임의의 공간(leaf)에 담기게 된다.
- 이렇게 만든 BSP트리로 월드 박스를 잘라나가면 PORTAL을 만들 수 있다.
- 모든 Leaf는 Portal을 통해서만 바깥을 볼 수 있다.
- Leaf -> Portal -> 다른 leaf의 가시성 검사를 기하하적으로 수행한다.
- 최종적으로 특정 leaf에서 보이는 leaf목록을 얻는다.

BSP/PORTAL/PVS + 뷰프러스텀컬링



BSP/Portal/PVS - 장점

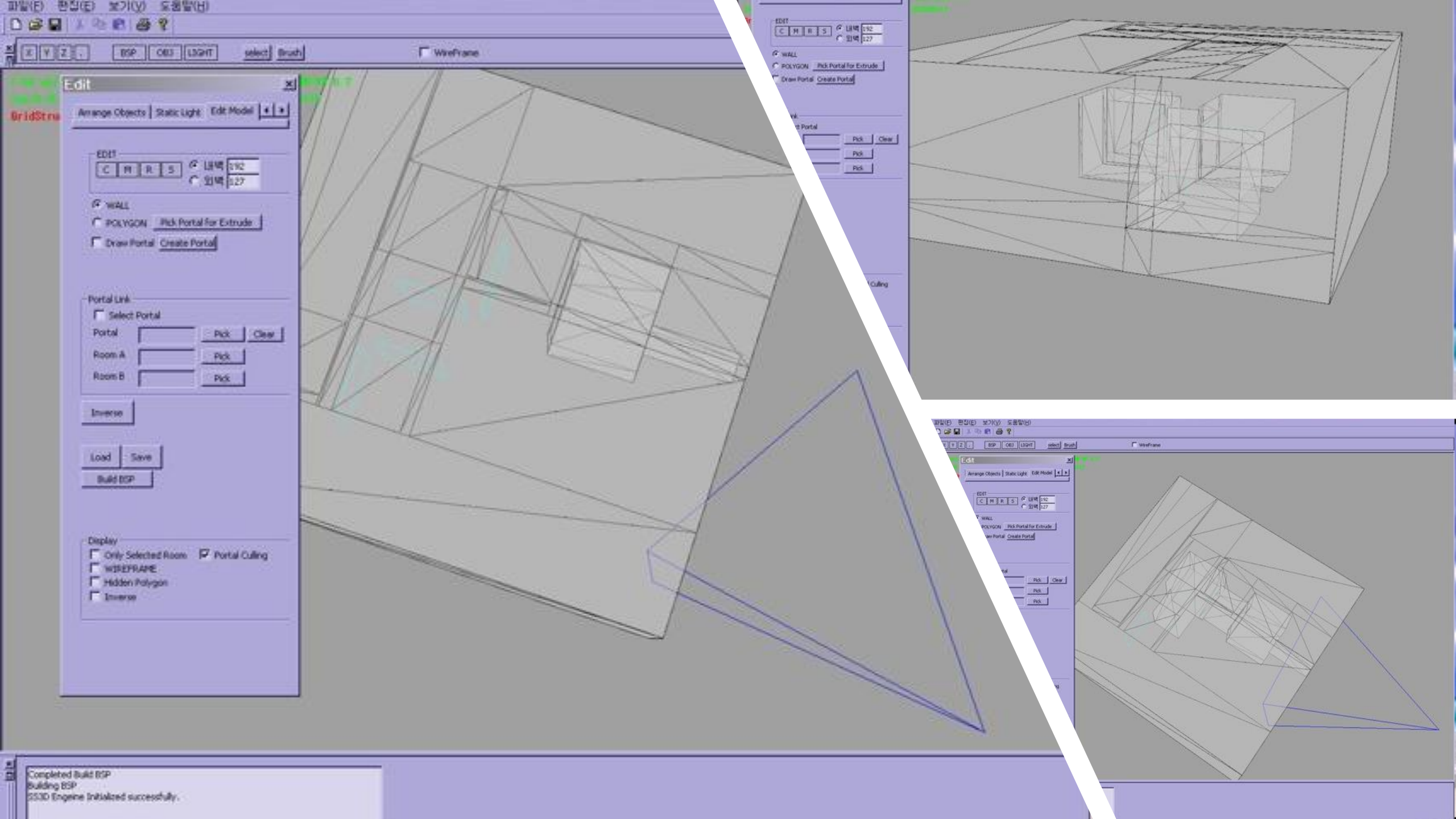
- 실내에 한해서 차폐물에 대한 컬링은 거의 완벽하다. 따라서 매우 훌륭한 퍼포먼스를 보여준다(Quake와 Quake엔진을 사용한 게임들에서 충분히 증명됐다

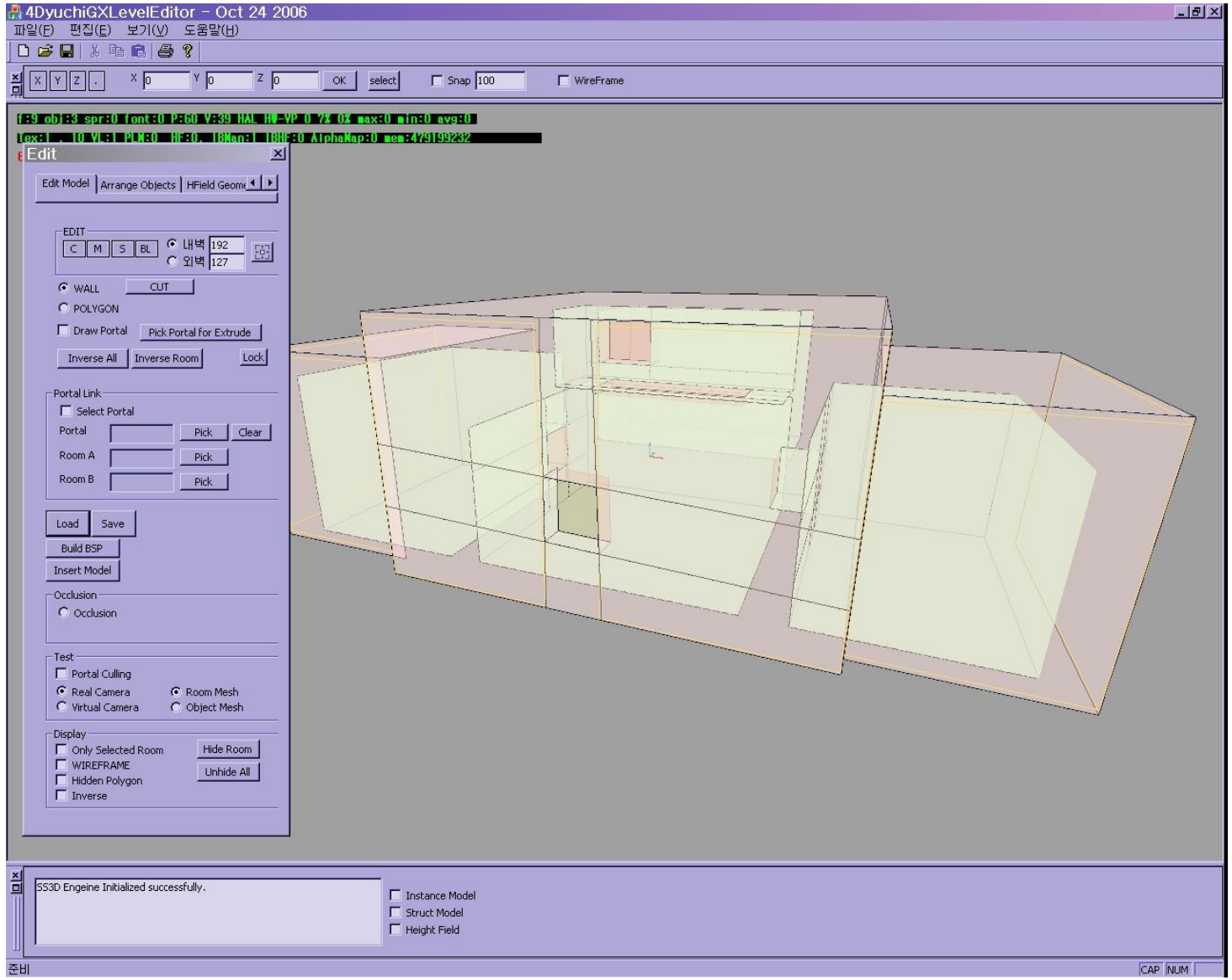
BSP/Portal/PVS - 단점

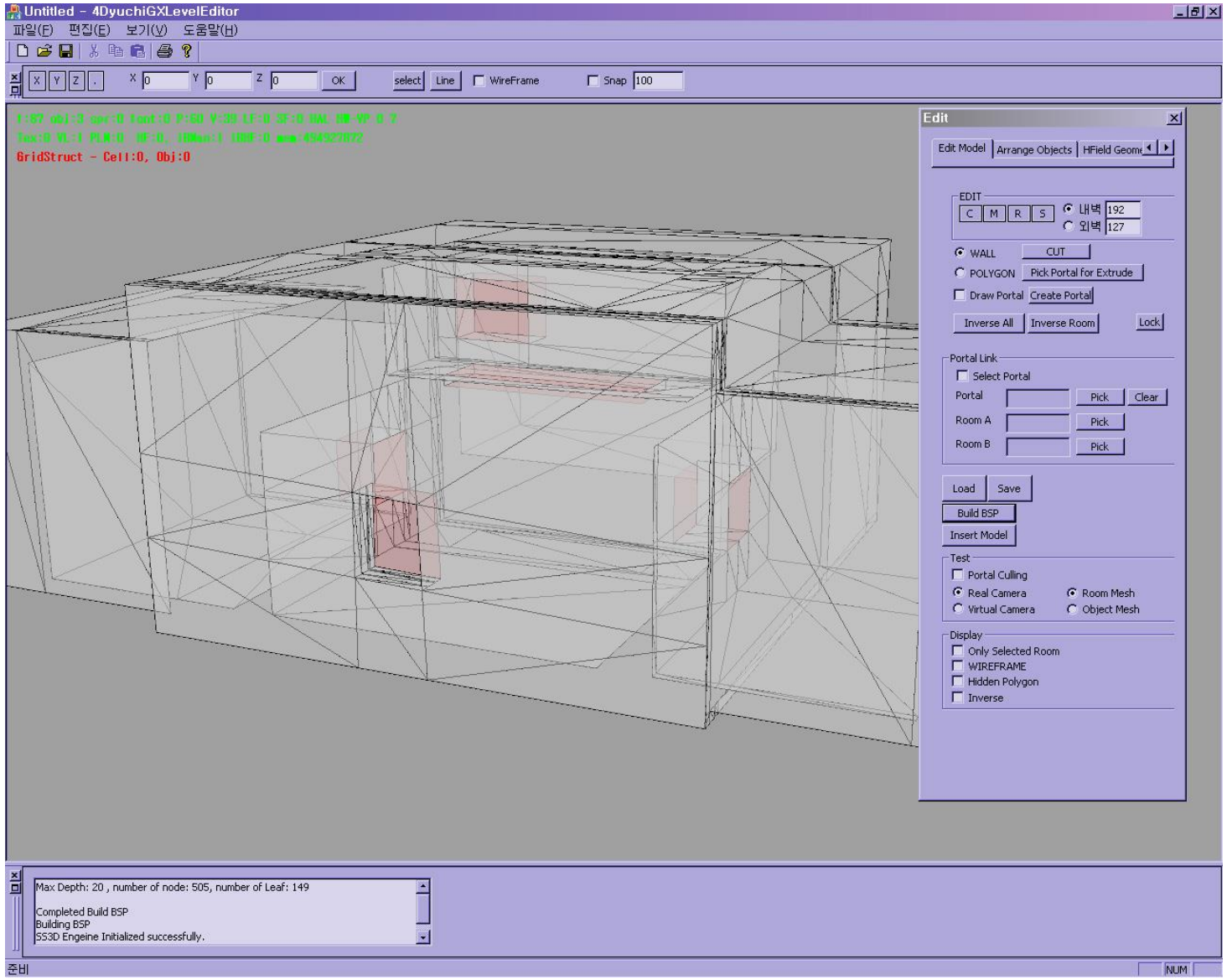
- 만들기 어렵다. (28각형 디버깅의 추억...)
- 닫힌 공간에서만 사용할 수 있다.따라서 실외처리가 어렵다.
- 실내가 아닌 실외(광대역 지형)에서는 거의 이득을 볼 수 없다.
- 복잡한 맵(삼각형이 많으면)을 만들기 어렵다. 기본은 단순하게 모델링하고, 복잡한 모양은 브러쉬로 갖다 붙여야 한다.

ROOM/PORTAL 방식

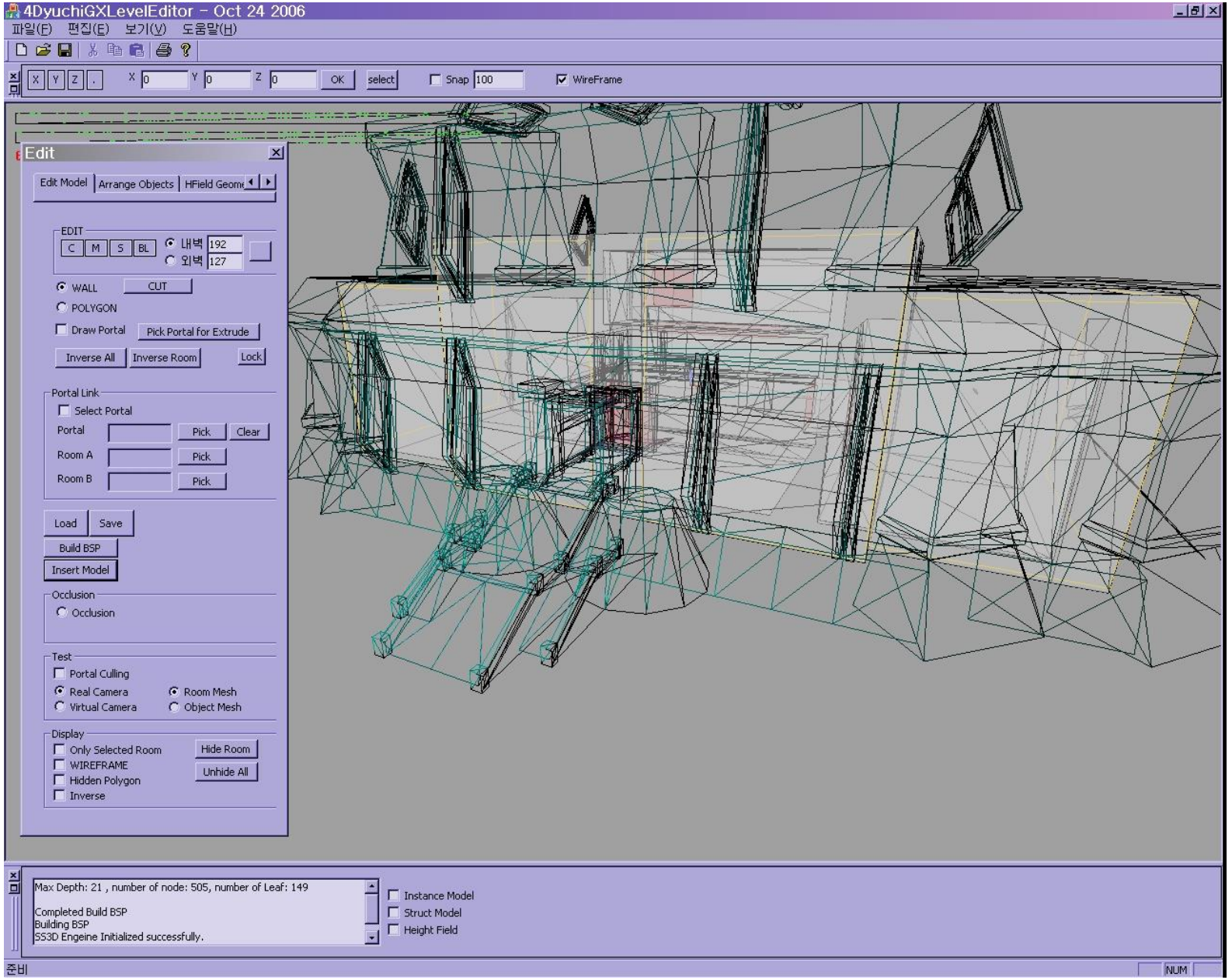
- Boolean연산이 가능한 전용 툴을 사용해서 방과 방을 복도로 연결한다. 복도와 방의 boolean연산을 수행할때 Portal이 만들어진다.
- 각 방에서는 다른 방을 볼때 Portal을 통해서만 볼 수 있다.
- 임의의 위치에서의 view frustum은 포탈을 통과할때 포탈의 크기에 맞게 잘릴 수 있다.
- 잘린 view frustum에 걸치는 오브젝트를 골라내기 때문에 Portal을 통과할수록 보이는 오브젝트의 수는 줄어든다.


Edit
Arrange Objects
Static Light
Edit Model
EDIT
C
M
R
S
WALL
POLYGON
Pick Portal for Extrude
Draw Portal
Create Portal
Portal Link
Select Portal
Portal
Pick
Clear
Room A
Room B
Inverse
Load
Save
Build BSP
Display
Only Selected Room
Portal Culling
WIREFRAME
Hidden Polygon
Inverse
BSP
OBJ
LIGHT
select
Brush
WireFrame
Completed Build BSP
Building BSP
3S3D Engine Initialized successfully.


4DyuchiGXLevelEditor - Oct 24 2006
파일(F) 편집(E) 보기(V) 도움말(H)
X Y Z
OK
select
Snap
WireFrame
Edit
Edit Model
Arrange Objects
EDIT
C M S BL
내벽 192
외벽 127
WALL
CUT
POLYGON
Draw Portal
Pick Portal for Extrude
Inverse All
Inverse Room
Lock
Portal Link
Select Portal
Portal
Pick
Clear
Room A
Pick
Room B
Pick
Load
Save
Build BSP
Insert Model
Occlusion
Occlusion
Test
Portal Culling
Real Camera
Virtual Camera
Room Mesh
Object Mesh
Display
Only Selected Room
WIREFRAME
Hidden Polygon
Inverse
Hide Room
Unhide All
S3D Engeine Initialized successfully.
Instance Model
Struct Model
Height Field
준비
CAP
NUM


Untitled - 4DyuchiGXLevelEditor
파일(F) 편집(E) 보기(V) 도움말(H)
X Y Z .
X 0 Y 0 Z 0 OK
select Line WireFrame Snap 100
GridStruct - Cell:0, Obj:0
Edit
Edit Model
Arrange Objects
HField Geome
EDIT
C M R S
내벽 192
외벽 127
WALL
CUT
POLYGON
Pick Portal for Extrude
Draw Portal
Create Portal
Inverse All
Inverse Room
Lock
Portal Link
Select Portal
Portal
Pick
Clear
Room A
Pick
Room B
Pick
Load
Save
Build BSP
Insert Model
Test
Portal Culling
Real Camera
Virtual Camera
Room Mesh
Object Mesh
Display
Only Selected Room
WIREFRAME
Hidden Polygon
Inverse
Max Depth: 20 , number of node: 505, number of Leaf: 149
Completed Build BSP
Building BSP
SS3D Engeine Initialized successfully.
준비
NUM

4. 맥스에서 만든 모델을 트리에 삽입



ROOM/PORTAL - 장점

- BSP/PVS/Portal방식보다 유연하다. 실외 공간에서도 사용할 수 있다.
- BSP/PVS/Portal방식만큼은 아니지만 그에 준하는 정도의 성능이 나온다.

ROOM/PORTAL - 단점

- Boolean연산 가능한 툴을 만드는게 보통 일이 아니다.
- 맵 모델링에 제약이 따른다. 벽의 두께가 10cm이상이어야 한다든지…
- 디자이너에게 추가 작업이 요구된다.

S/W Occlusion Culling

CPU에서 하던걸 GPU로

GPU에서 하던걸 CPU로

SW Occlusion Culling

- CPU 코드로 구현한 z-buffer
- Throughput은 HW방식에 비해 크~~~게 떨어진다. 대신 Draw Call 준비가 필요없고 응답성이 빠르다. CPU를 사용해서 z-buffer를 구현한다.
- Texturing이 없는 SW Rasterizer를 구현한다.
- GPU가 없던 시절 고대의 프로그래머들은 작은 사이즈의 buffer로 SW Occlusion Culling을 이미 사용했었다.
- 이후 SW Z-Buffer Rasterizer라 부르자.
- 물론 요새는 거의 사용하지 않는다…그러나..

KD-Tree

- X,Y,Z 축에 정렬된 BSP Tree.
- 각 node를 3가지 축과 축방향의 거리 D값으로 표현할 수 있다.
- KD-Tree를 탐색할때 카메라로부터 가까운 node와 먼 node를 찾을 수 있다.
- Ray의 교차판정 등에 많이 사용한다.
- Near node와 Far node를 구분하지 않는다면 Quad-Tree와 별 차이는 없다.
- 게임의 월드를 KD-Tree로 관리하는 경우가 많다.

KD-Tree를 이용한 오브젝트 수집

- node에 대해 view frustum culling. Frustum에 포함되지 않으면 다음 node로.
- Leaf이면 leaf가 포함하는 오브젝트 수집
- 오브젝트에 대해 view frustum culling.
- 다음 node로 진행

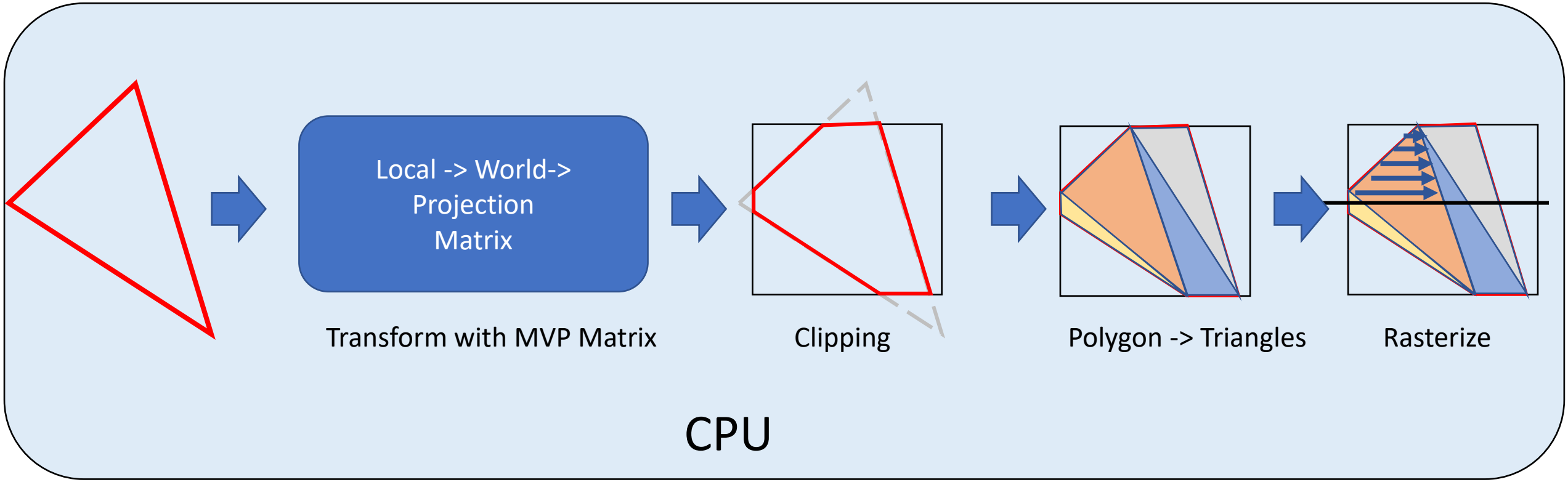
KD-Tree를 이용한 오브젝트 수집

- Tree 탐색중에 view frustum culling을 통과한 node라도 오브젝트에 가려져서 보이지 않을 수 있다. (사실은 이런 경우가 아주 많다.)
- Tree탐색중에 가려져서 보이지 않는 node를 바로 걸러낼 수 없나?
- Tree탐색중에 Z-buffe를 구성해가면 가능하겠네?

KD-Tree + S/W Occlusion Culling

- HW Occlusion Culling은 Occluder/Occludee를 렌더링하기 위해 준비 시간이 너무 길고 GPU로부터 결과를 얻어오는데도 많은 시간이 걸린다. 따라서 KD-Tree탐색중에는 사용할 수 없다.
- 삼각형 몇십 몇백개 정도를 rasterize & test하기 때문에 throughput이 크게 중요하진 않다.
- 응답성은 아주아주아주 중요!!!
- 따라서 KD-Tree탐색중에 사용할 Occlusion Culling은 SW방식을 사용한다.

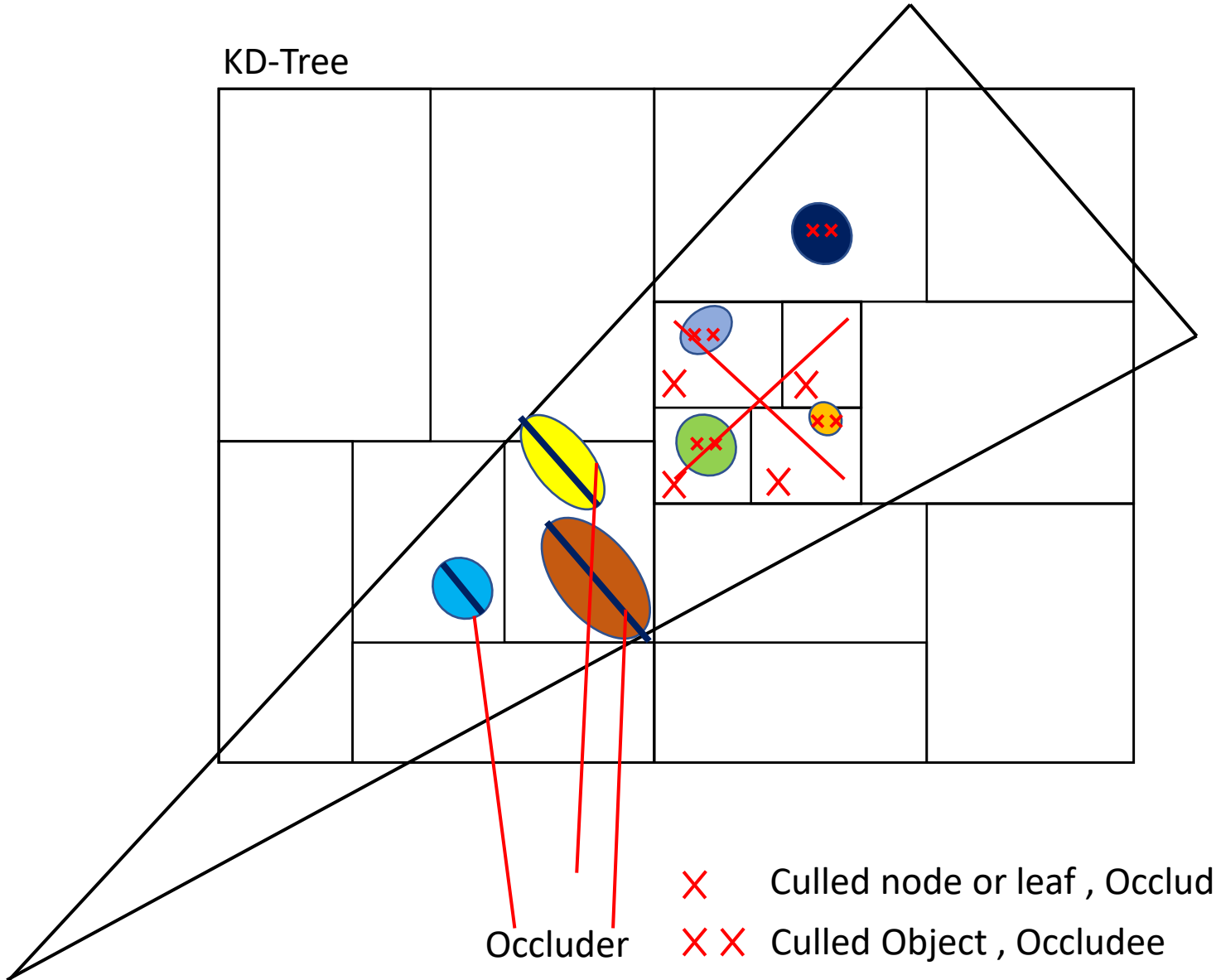
CPU로만 처리하던 시절을 기술을 꺼낸다 – 단 culling용으로만…



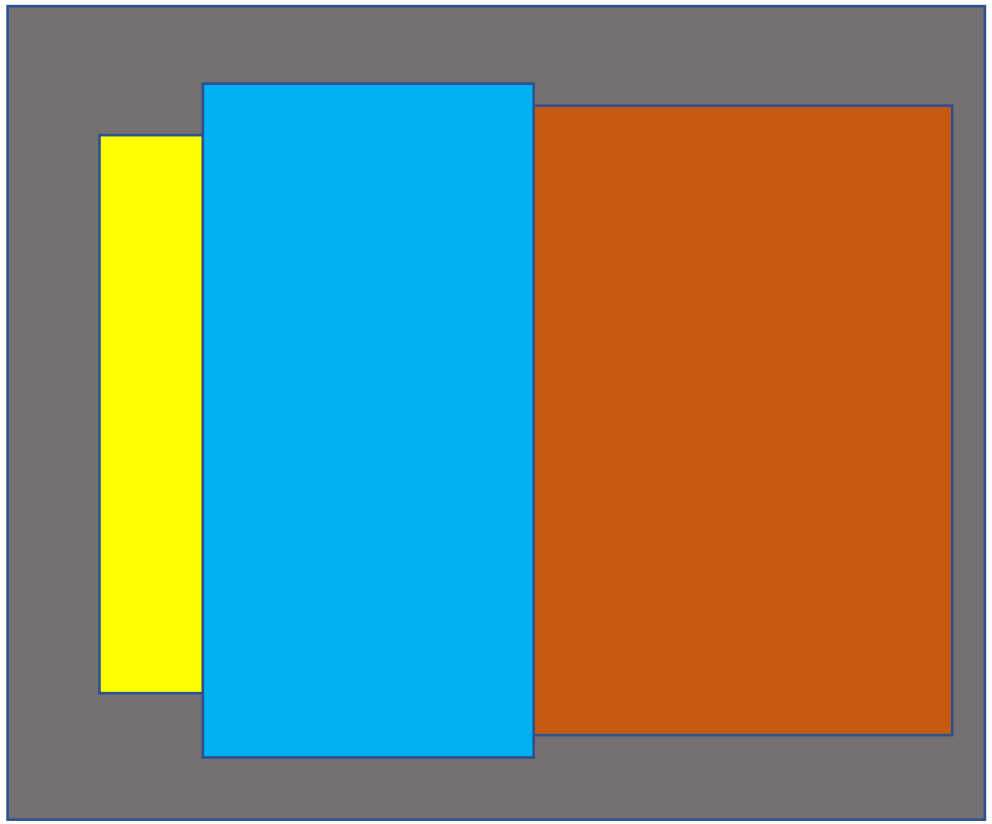
SW Occlusion Culling in KD-Tree

KD-Tree순회중에 먼저 발견한 오브젝트들(Occluder)을 z-buffer에 그린다. 이로 인해 다음번 순회할 node(or leaf)를 통째로 제외시킬수 있다.

KD-Tree



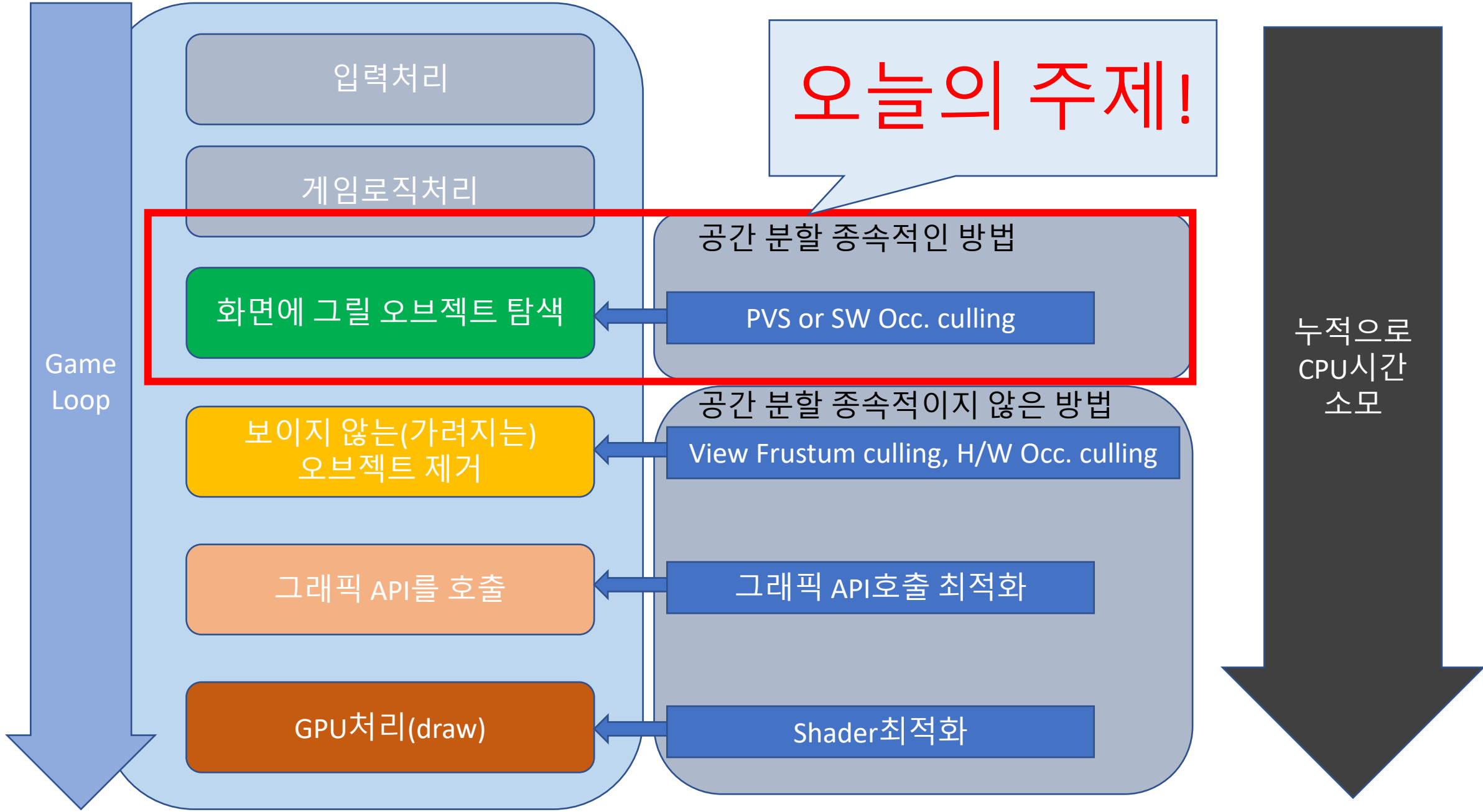
Frame Buffer or Z-Buffer



Culled node or leaf , Occludee

Culled Object , Occludee

기본은 여기까지 이제 응용을....



앞 단계에서 많은 오브젝트들을 제거할 수록 성능이 향상될 가능성이 높다.

(my)프로젝트의 기술적 목표

삼각형 베이스 맵 + 복셀 편집

이런거 만들건데요...

- 삼각형 베이스 맵이다. 그런데 실내외를 로딩없이 들락거린다.
- 지형이 변형되는 것이 너무나 당연한 Voxel 지형이 동시에 렌더링 된다.
- 아트 디자이너 자원이 부족하다(심지어 0명).

이런 경우라면...(해결책)

- 삼각형 베이스 맵이다. 그런데 실내외를 로딩없이 들락거린다.
  - GPU를 이용한 PVS를 사용한다. PVS를 미리 빌드하되 GPU를 이용해서 무식하게 처리하지만, 맵의 제약은 줄일 수 있다.
- 삼각형 베이스맵 위에 Voxel지형도 올라간다(지형이 변형된다.)
  - Voxel world에 대해선 PVS를 사용하지 않는다. S/W Occlusion Culling을 사용한다.
- 아트 디자이너 자원이 부족하다(심지어 0명).
  - 사람의 추가작업이 필요없도록 한다. GPU를 이용해서 무식(!)하게 계산할 것이므로 Boolean연산이 필요한 전용 맵툴을 사용하지 않는다.
  - 또한 S/W Occlusion Culling은 수동 Occlusion을 요구하지 않는다.

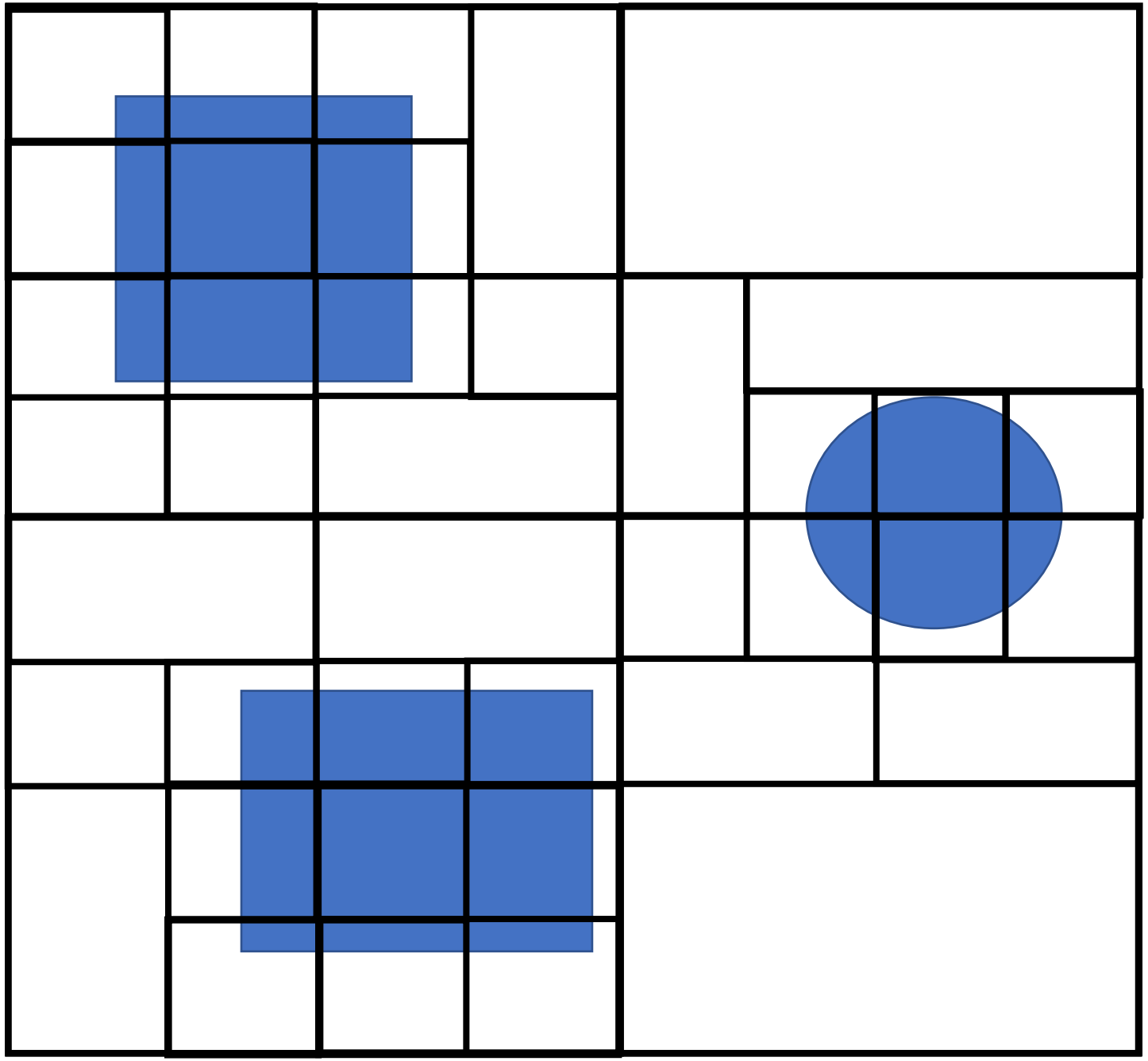
완벽하지 않더라도 힘을 합치면!

KD-Tree + PVS with GPU + S/W Occlusion culling + H/W Occlusion culling

PVS with GPU

Potentially visible set

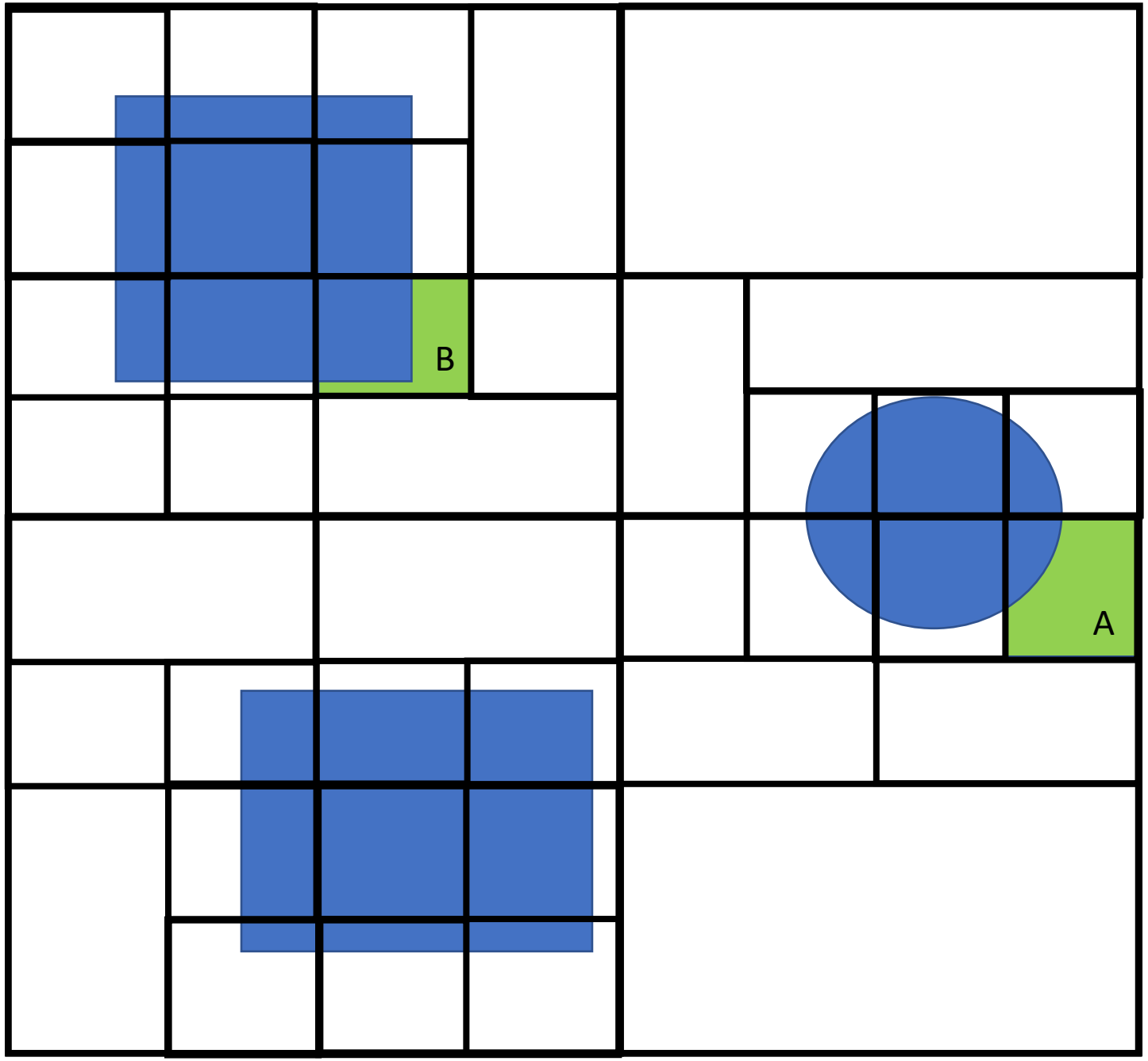
특정 공간에서 보이는 공간들의 목록을 미리 계산해둔다.



월드 공간을 분할한다(뭐든 상관없지만 여기서는 KD-Tree)

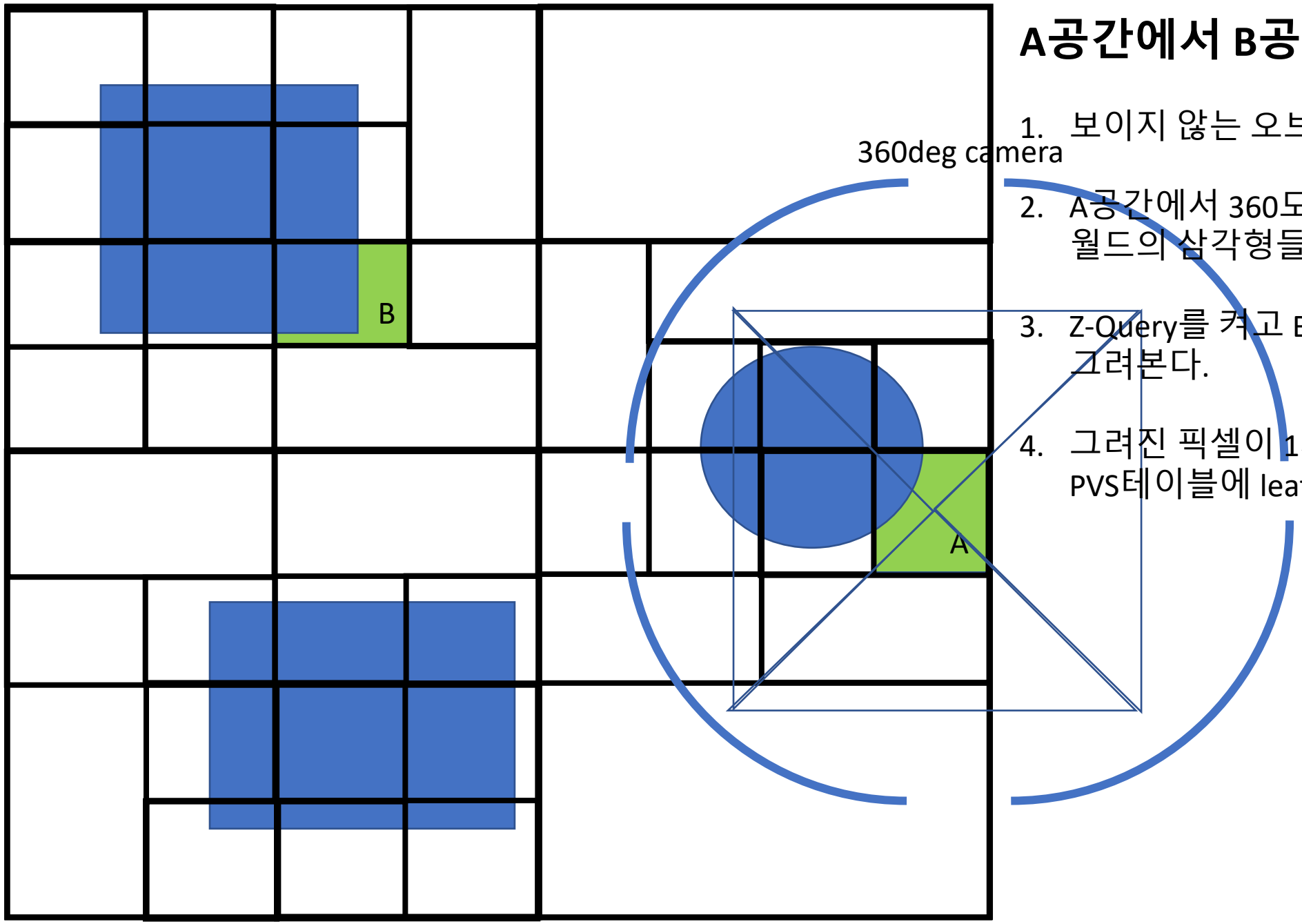
삼각형에 교차하지 않는 Node는 leaf가 된다.
또는 node의 사이즈가 일정 이하면 leaf가 된다.

트리의 leaf는 오브젝트 및 지형지물의 삼각형이 위치하는 공간이 된다(개념적으로 ROOM이라고 생각하면 된다.)



A공간에서 B공간이 보일까?

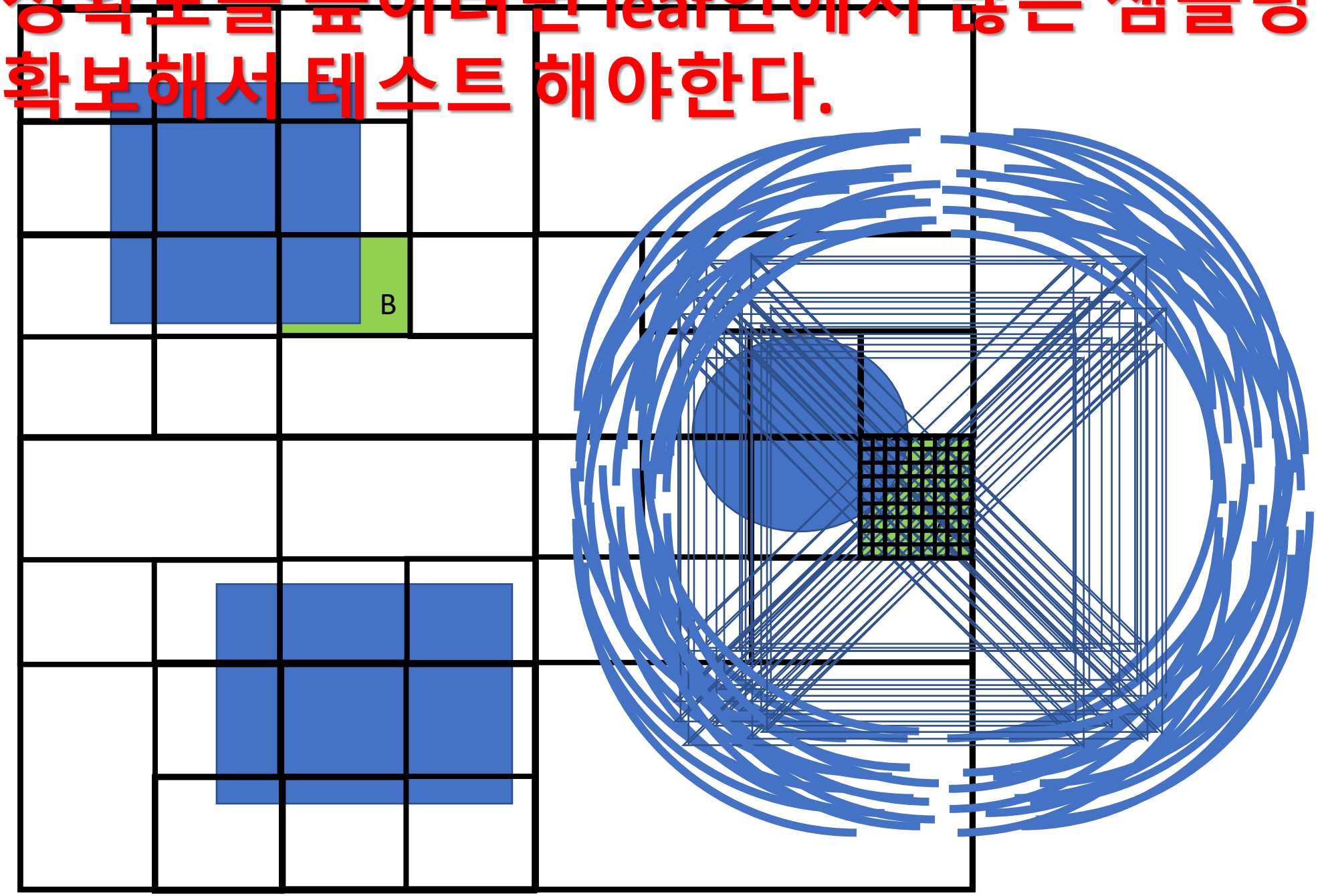
1. 보이지 않는 오브젝트는 z-test에 걸러진다.
2. A공간에서 360도를 카메라를 만들어서 월드의 삼각형들을 몽땅 그린다.
3. Z-Query를 켜고 B공간의 leaf 바운딩 박스를 그려본다.
4. 그려진 픽셀이 1개 이상이면 leaf A의 PVS테이블에 leaf B의 인덱스를 추가.



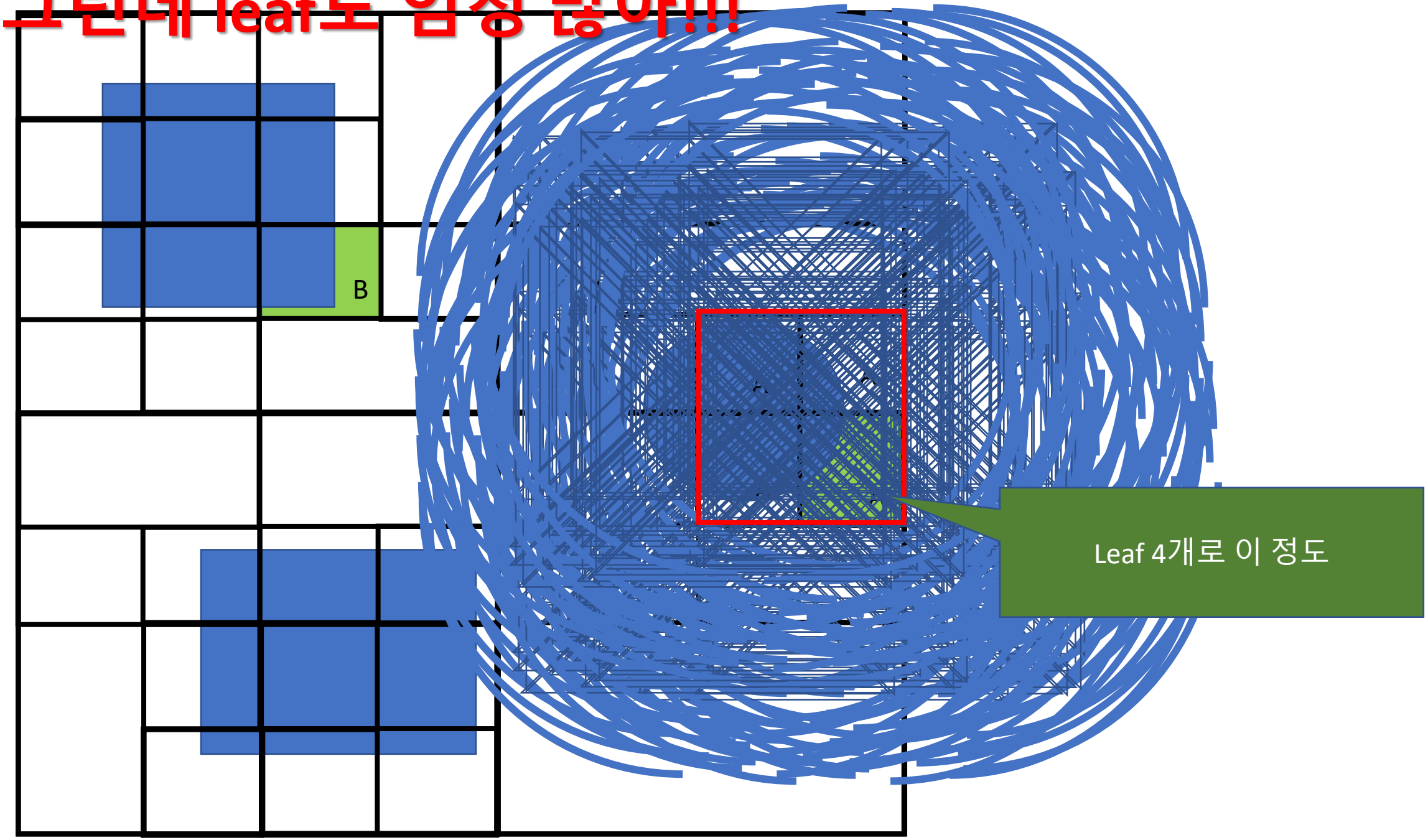
A공간에서 B공간이 보일까?

1. 보이지 않는 오브젝트는 z-test에 걸러진다.
2. A공간에서 360도를 카메라를 만들어서 월드의 삼각형들을 몽땅 그린다.
3. Z-Query를 켜고 B공간의 leaf 바운딩 박스를 그려본다.
4. 그려진 픽셀이 1개 이상이면 leaf A의 PVS테이블에 leaf B의 인덱스를 추가.

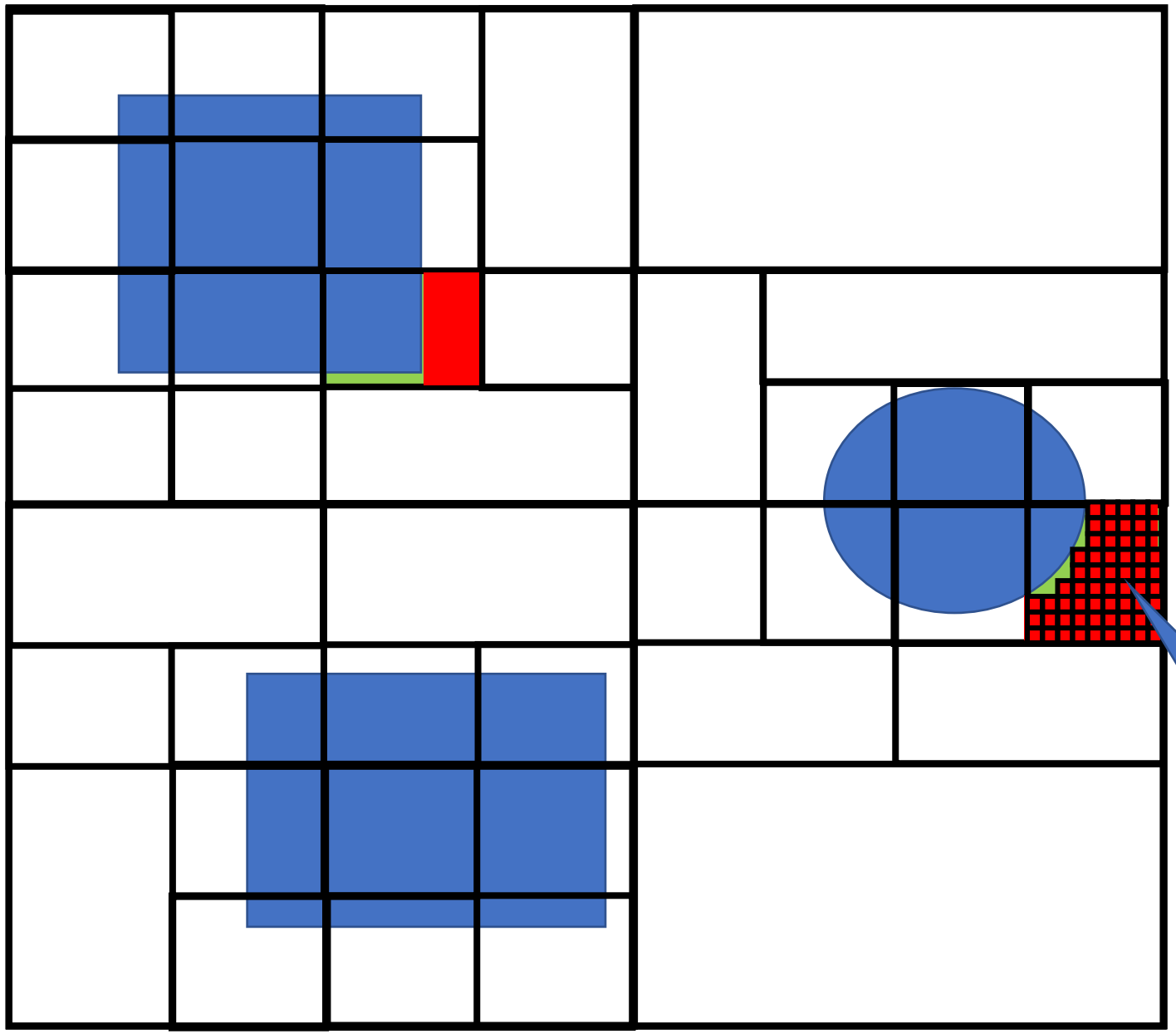
정확도를 높이려면 leaf안에서 많은 샘플링 포인트를 확보해서 테스트 해야한다.



그런데 leaf도 엄청 많아!!!



검사 대상 샘플링 포인트 개수 줄이기



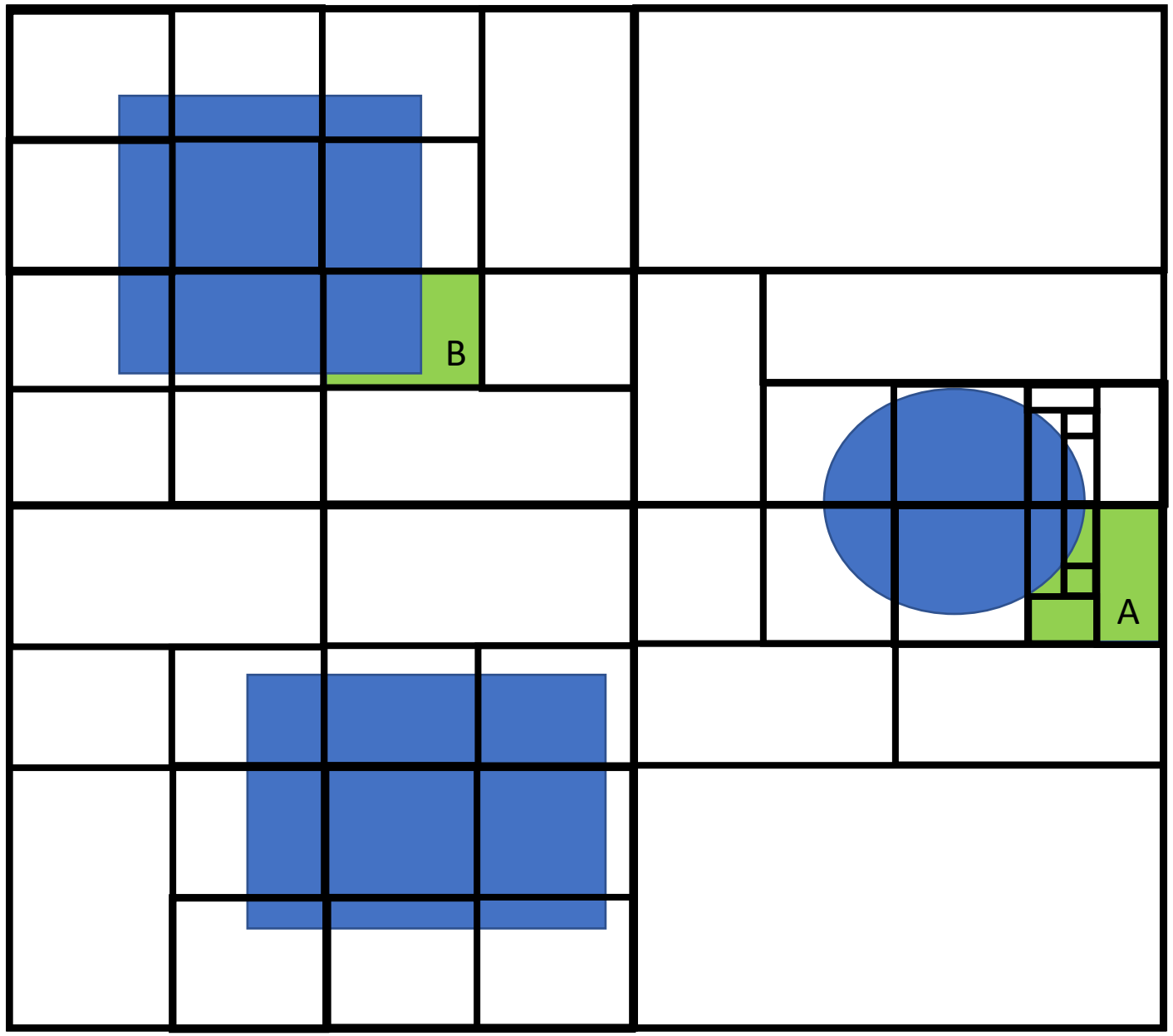
1. Leaf의 내부를 균등분할한것이 샘플링 포인트이므로 지형에 정밀하게 들어맞는 leaf(총부피 감소)를 사용하면 샘플링 포인트 개수도 줄어든다.

2. 유효하지 않은 공간(벽 안쪽)은 샘플링 포인트에서 제외한다.

3. 유효하지 않은 공간(벽 안쪽)은 가시성 검사 대상에서도 제외한다.

Leaf의 총부피가 줄어들면 샘플링 포인트의 개수는 줄어든다.

검사 대상 샘플링 포인트 개수 줄이기



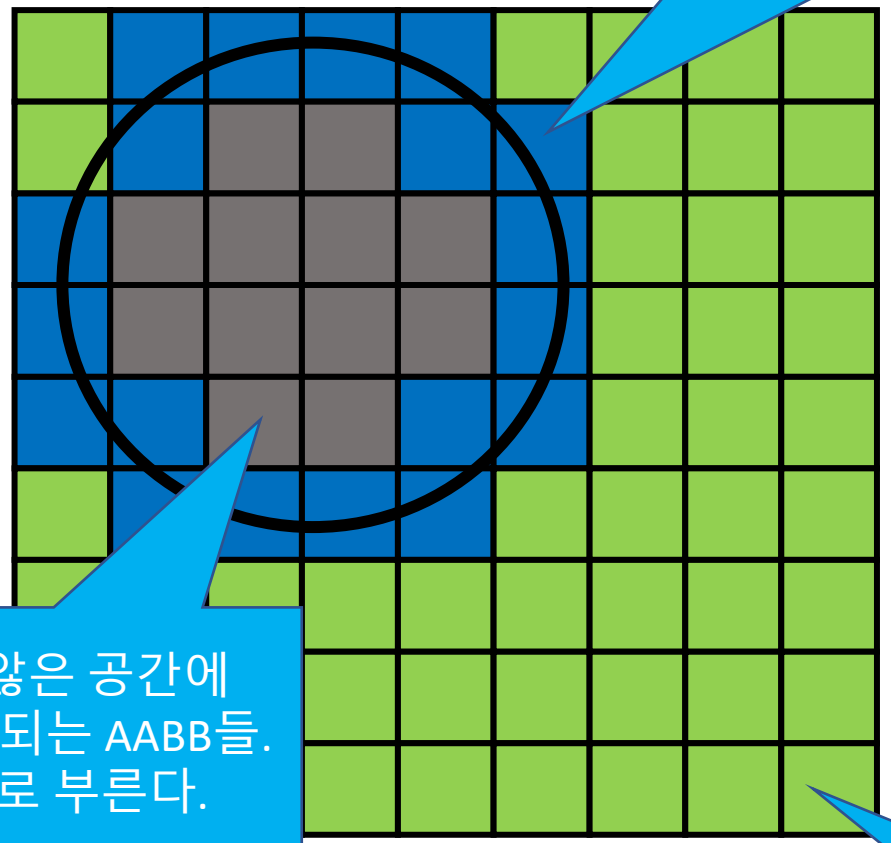
1. 1차 KD-Tree의 leaf안에서 다시 sub-tree를 빌드한다. 여기서는 leaf가 될 수 있는 최소 사이즈를 더 작게 설정한다.

2. 좀더 정밀하게 삼각형 지형과 들어맞는 leaf들을 얻는다.

3. Leaf들로부터 육면체 AABB들을 얻는다.

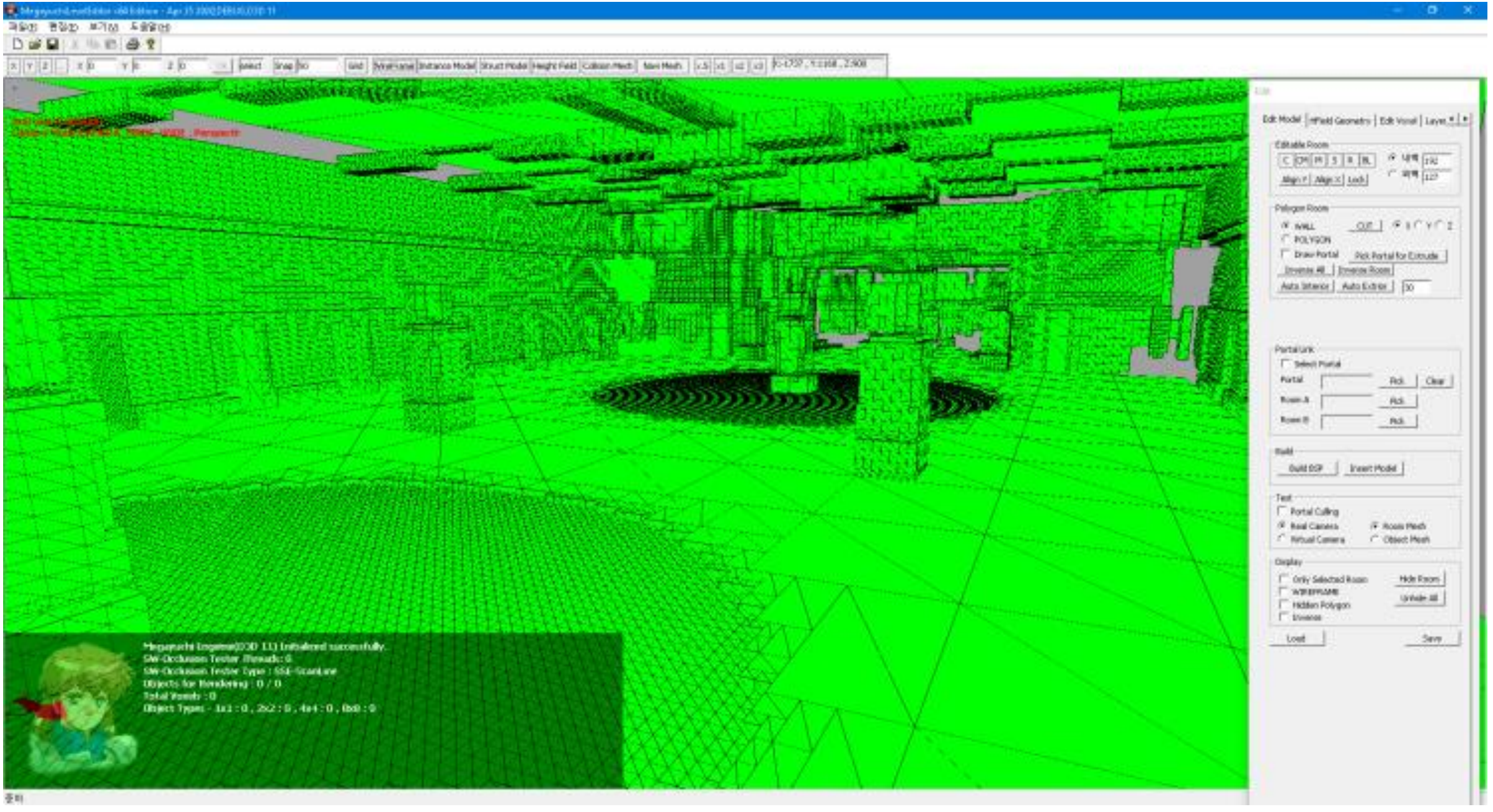
4. 이 AABB들 중에 명백하게 유효하지 않은 공간(벽 안쪽)에 들어가는 녀석들은 버린다. (https://www.slideshare.net/dgtman/voxelization-with-gpu 참고)

검사 대상 leaf줄이기

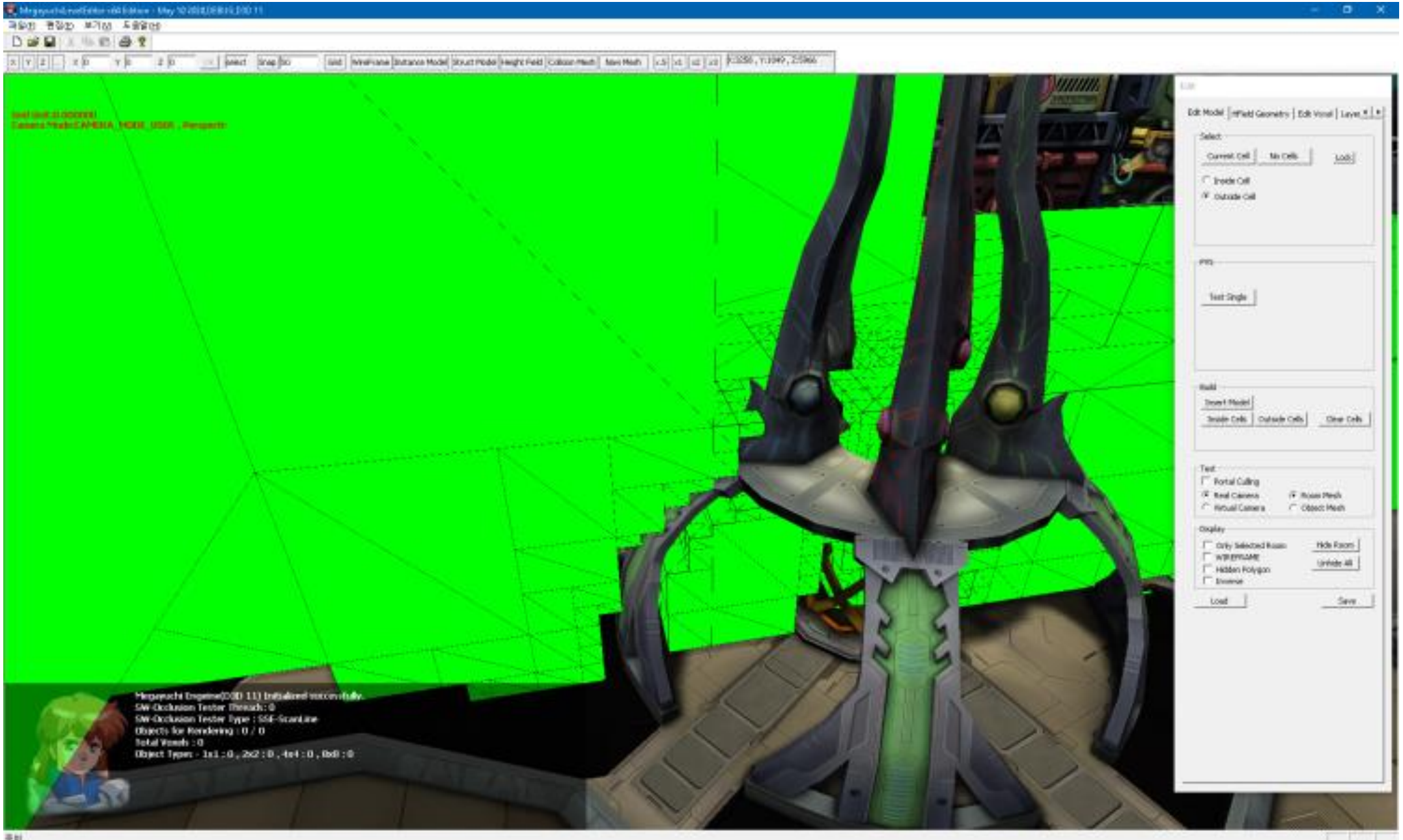


1. 1차 KD-Tree의 leaf안에서 다시 sub-tree를 빌드한다. 여기서는 leaf가 될 수 있는 최소 사이즈를 더 작게 설정한다.
2. 좀더 정밀하게 삼각형 지형과 들어맞는 leaf들을 얻는다.
3. Leaf들로부터 육면체 AABB들을 얻는다.
4. 이 AABB들 중에 명백하게 유효하지 않은 공간(벽 안쪽)에 들어가는 Inside cell들은 버린다.
   (https://www.slideshare.net/dgtman/voxelization-with-gpu 참고)
5. Unknown cell과 Outside cell들만 샘플링 포인트로 사용한다.

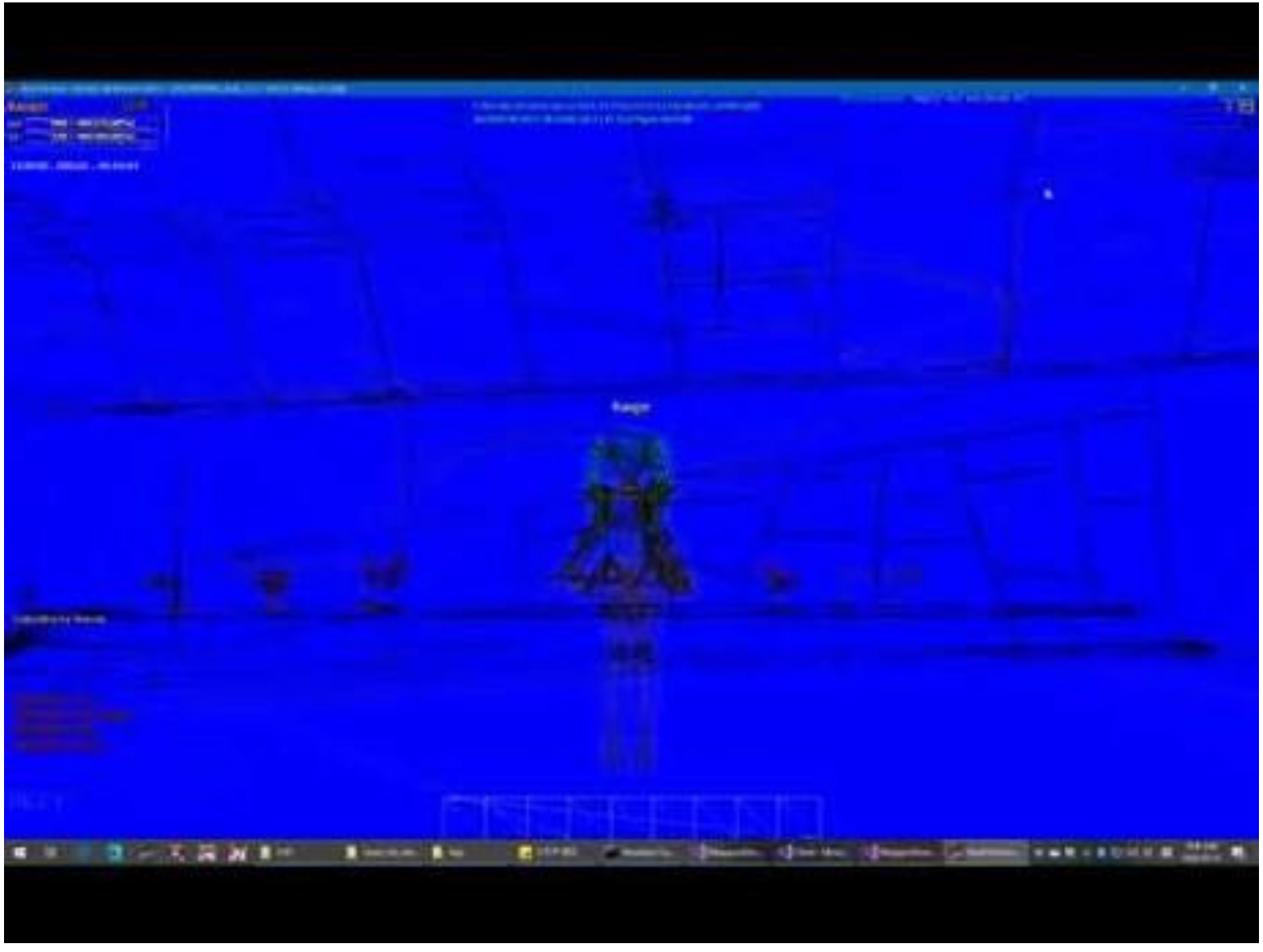
1차 KD트리 + 2차 Sub트리 빌드 후 Inside Cells + Unknown Cells



Outside Cells – (샘플링 포인트로 사용)



GPU를 이용한 PVS의 실제 적용



GPU를 이용한 PVS - 장점

- 런타임에 Culling에 필요한 작업이 거의 없다. CPU시간 대폭 절약
- 꽤 많은 수의 보이지 않은 오브젝트(정확히는 leaf단위)들을 렌더링 파이프라인에서 제거할 수 있다.
- Culling코드가 엄청 단순해진다(PVS빌드 코드를 짜느라 고생한 대가)

GPU를 이용한 PVS - 단점

- 만들기 빡시다 (Quake의 BSP/Portal/PVS 만들기보단 쉽다).
- 미리 계산할때 시간이 꽤 필요하다.
- 정확한 결과를 얻지 못할수도 있다(보여야 될 오브젝트(공간)이 안보인다). – 해결은 가능.

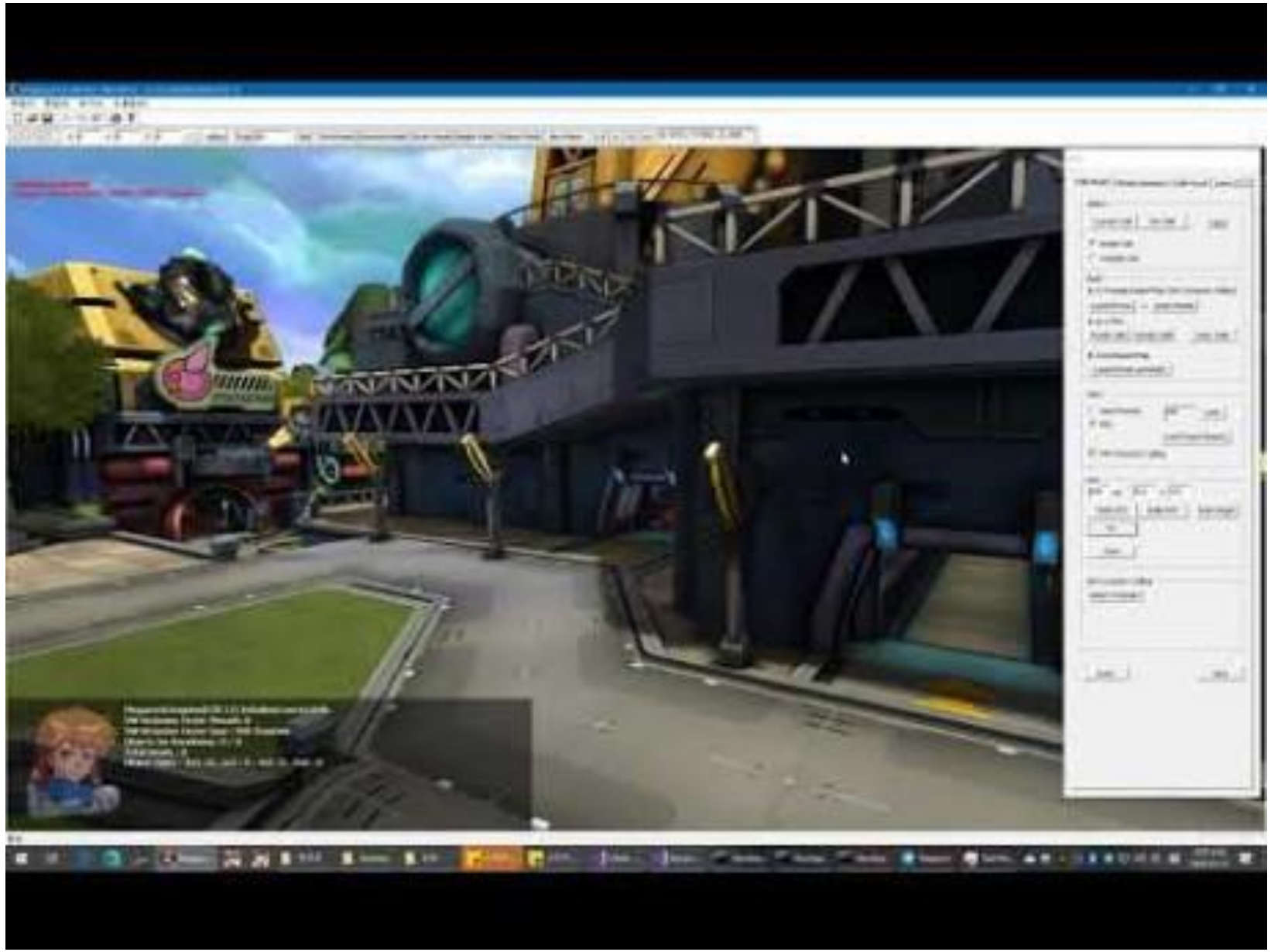
오류

- 안보일 공간이 보인다고 처리된건 큰 문제가 아니다.
- 보일 공간인데 안보인다고 처리된건 큰 문제다.

오류처리

- 정밀도를 높여서 전체 월드를 다시 계산.
- 정밀도를 높여서 해당 공간(leaf)만 재계산.
- 일반 H/W Occlusion Culling이나 S/W Occlusion Culling만 켜고 보이는 공간 목록을 얻어서 현재 공간(leaf)의 PVS테이블에 추가.
- 아예 개발기간 내내 테스트 플레이를 할 때마다 자동으로 PVS테이블을 갱신하도록 한다.

오류 해결 in Level Editor



오류 해결 in Client

GPU를 이용한 PVS 적용 팁

- 삼각형 베이스 맵의 게임일 경우 PVS사용 가능.
- 개발중엔 PVS는 거의 잊고 살아도 좋다.
- 1개월에 한번, 3개월에 한번 정도는 PVS빌드-테스트.
- 개발팀 내에서 테스트중, 특정 leaf에서 특정 leaf가 반드시 보이거나, (통계적으로)한번도 보이지 않는다는 정보 수집 가능 -> PVS테이블 업데이트
- 최종적으로 유저가 플레이할때 도움이 됨(1프레임이라도 더 나오면 유저는 좋아한다-성질을 덜 낸다).

Async S/W Occlusion Culling

- S/W Occlusion Culling은 대단히 부하가 많이 걸리는 작업이기 때문에, 이를 매 프레임마다 수행하면 오히려 전체적인 응답성을 떨어뜨릴 수 있다.
- S/W Occlusion Culling으로 이득을 보되, 최악의 경우에도 S/W Occlusion Culling때문에 느려지는 사태를 방지한다.

Async S/W Occlusion Culling

- S/W Occlusion Culling을 별도 스레드로 분리한다.
- S/W Occlusion Culling을 비동기적으로 수행한다고 하는 것은, 항상 이전 프레임(적어도 1프레임 전)의 S/W Occlusion Culling결과를 사용하게 된다는 뜻이다.
- 카메라 상태가 변화하지 않았다면 이전 프레임의 결과를 그대로 사용해도 문제가 없다. 대부분의 경우 1프레임 사이에 카메라 상태가 크게 변하지는 않는다. 따라서 일정 정도의 변화량 미만이면 카메라의 상태가 변화하지 않았다고 간주한다.

스레드별 작업 수순

메인스레드

1. 렌더링될 복셀 오브젝트를 탐색하는 함수로 진입.
2. Raster/Test스레드가 완료시킨 SW Occlusion Culling결과가 있는지 확인한다. 없으면 5로.
3. 완료된 SW Occ 결과가 있으면 SW Occlusion Culling을 수행할 당시의 카메라 상태와 현재 카메라 상태 사이에 유의미한 변화가 있었는지 확인한다.
4. 유의미한 변화가 있었다면 Raster/Test스레드가 저장한 culling테이블을 참조해서 이번 프레임에 보이지 않을 leaf들의 인덱스를 bit table에 표시한다.
5. KD-Tree를 탐색해서 렌더링될 leaf들과 leaf에 속한 복셀 오브젝트들을 수집한다. 이때 위에서 작성한 bit table에 표시된 leaf에 포함된 오브젝트는 제외시킨다.
6. 렌더링된 오브젝트들을 탐색하는 과정에서 복셀 오브젝트의 삼각형 매시도 함께 수집한다.
7. 비동기 S/W Rasterizer/Tester에 수집한 삼각형 매시들을 전달하고 Raster/Test스레드를 깨운다.

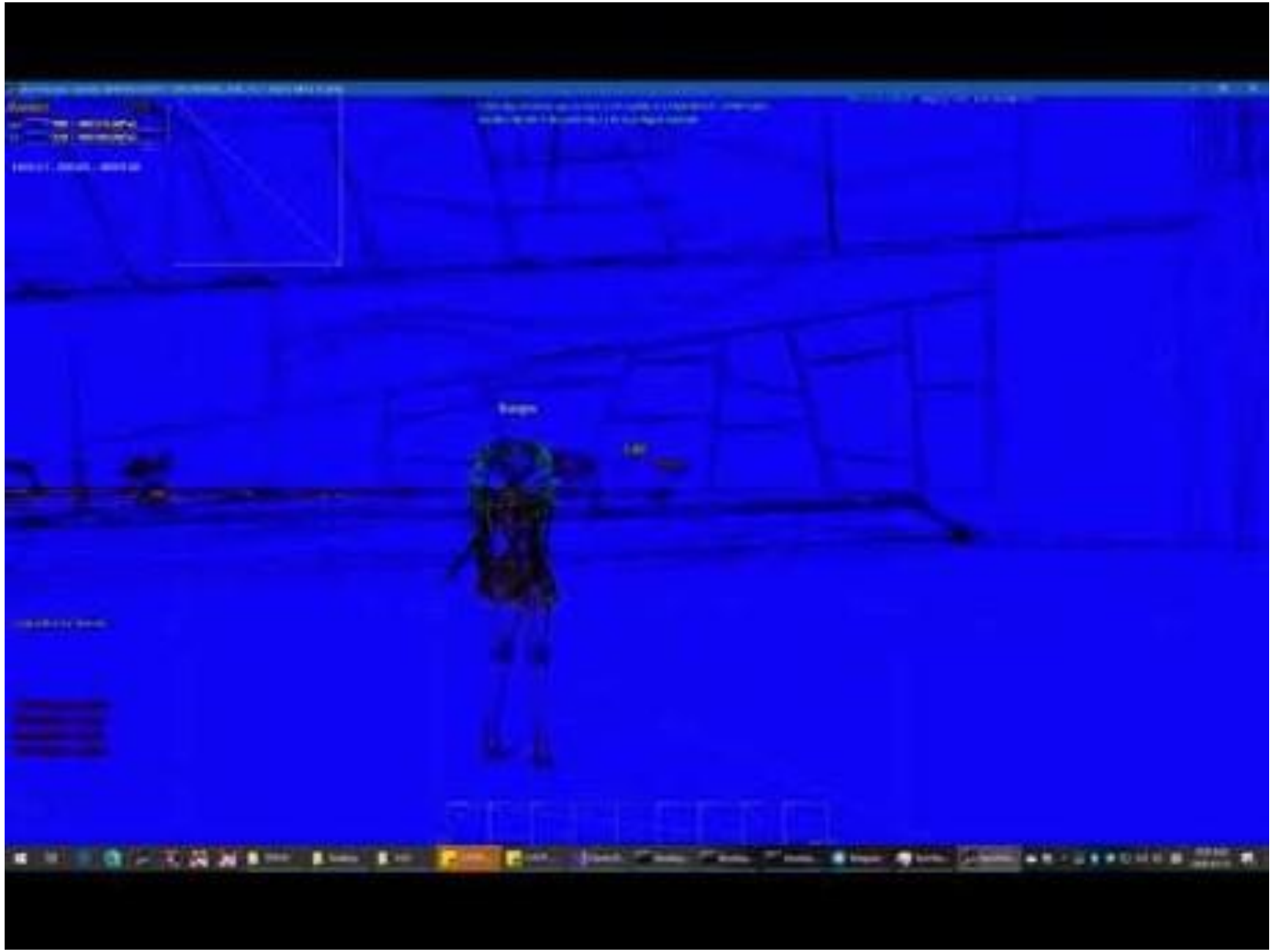
Raster/Test스레드

1. 메인스레드에 의해 깨어난다.
2. 메인스레드로부터 전달받은 삼각형 매시들(Voxel오브젝트로부터 얻은, Occluder로 사용할 삼각형)을 S/W Z-Buffer에 그린다.
3. 전달달받은 leaf들의 AABB가 S/W Z-Buffer에 그려지는지 테스트한다. depth test를 통과한, 보인다고 판정된 leaf들만 결과 버퍼에 저장한다.
4. wait상태로 진입

Async SW Occlusion Culling의 적용 - 삼각형 베이스 맵에서

- 지형이 변형되지 않는다고 가정
- 부분적인 변형이 일어나는 지형매시의 경우 Occluder삼각형으로는 사용하지 않는다.
- Occluder로 사용할 삼각형들을 Tree에 알맞게 넣으면 된다.

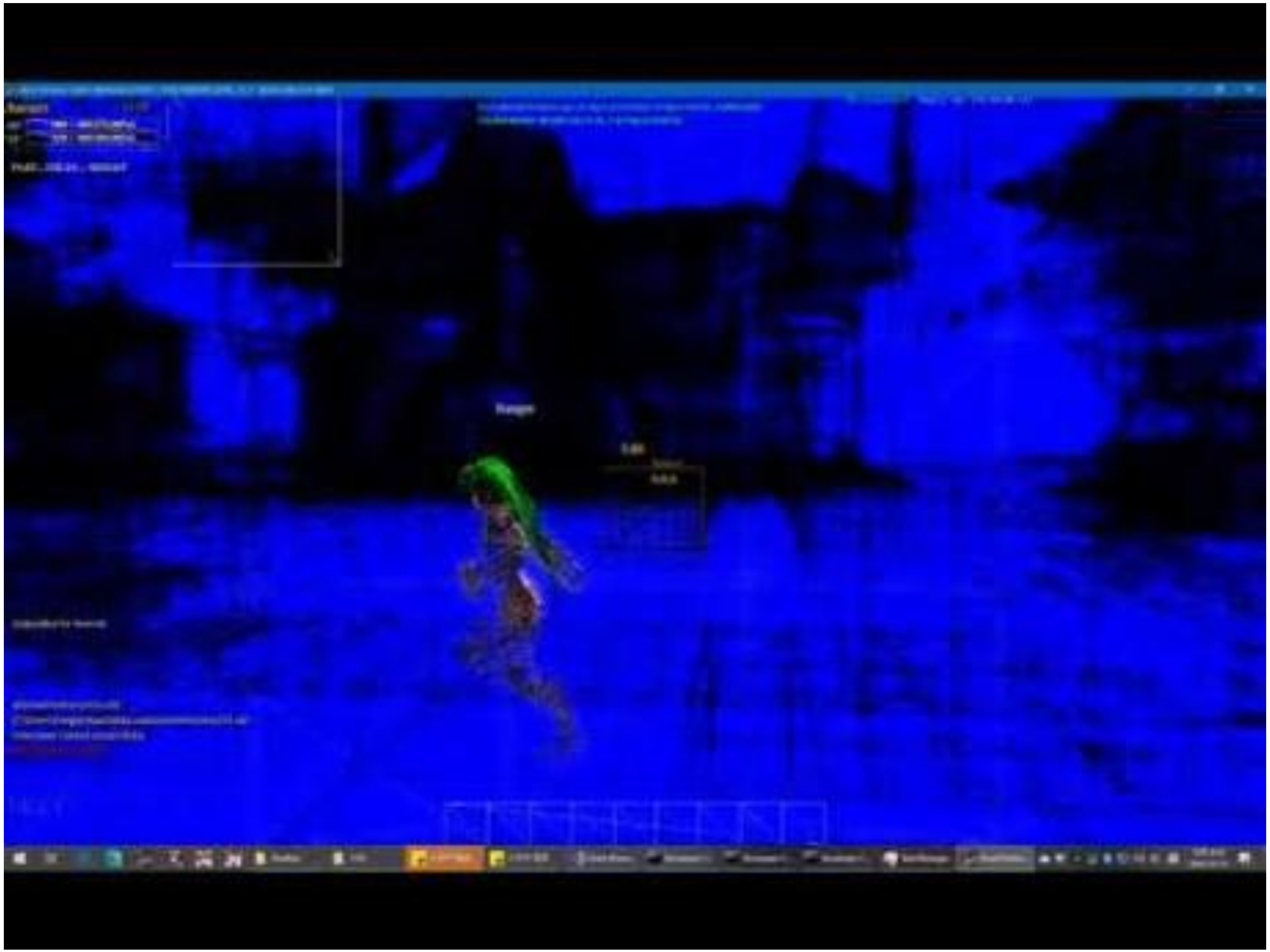
S/W Occlusion Culling in Triangles based map

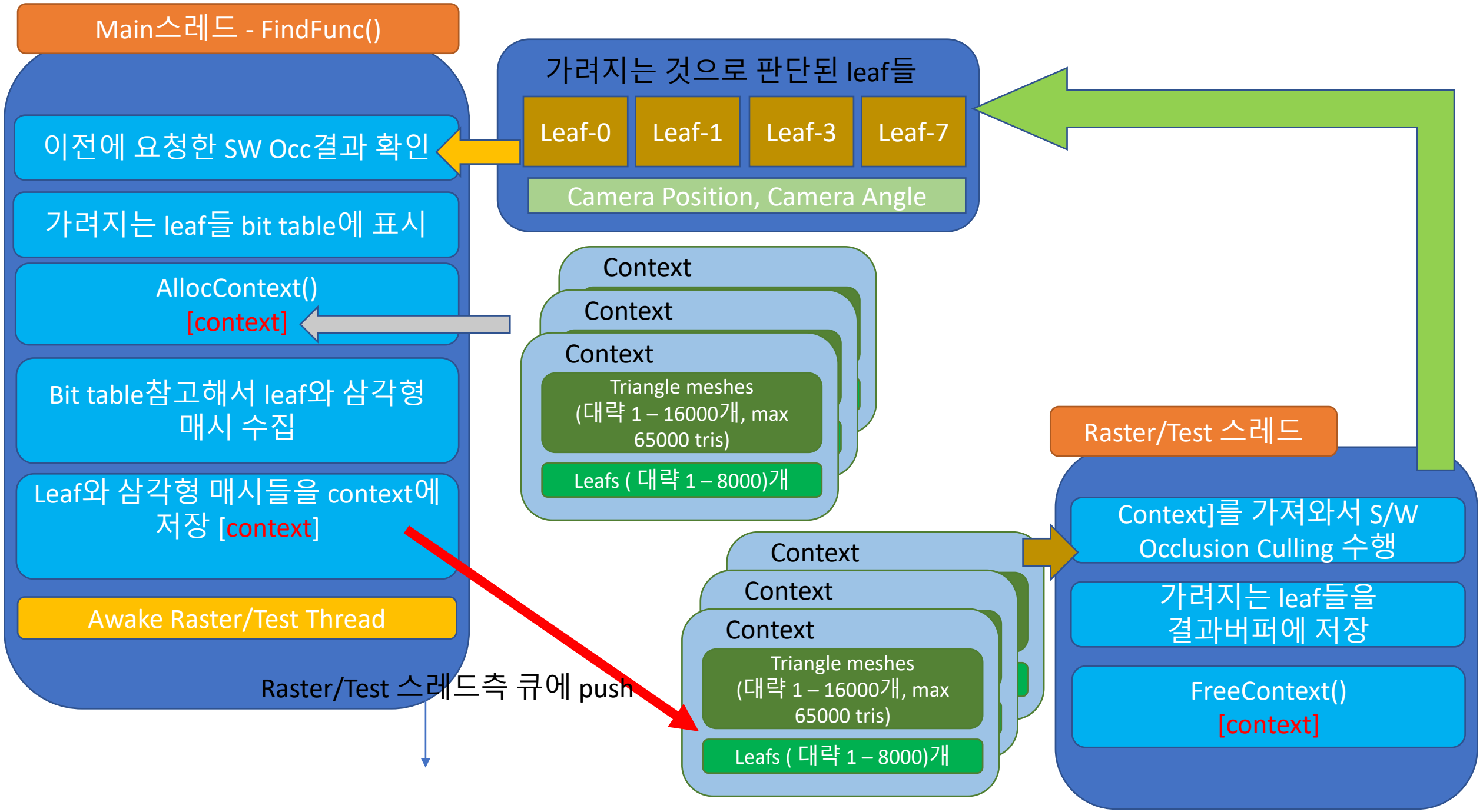


Async SW Occlusion Culling의 적용 - 복셀 베이스 맵에서

- 50cm x 50cm x 50cm 복셀맵의 삼각형은 정~~~~말 많다.
- 삼각형 수집 비용만 해도 상당하다.
- 삼각형 수집비용을 줄여야 한다.
- 스레드간 삼각형 전달 비용도 줄여야한다.
- 거슬리는 문제들(보여야할 오브젝트가 안보이는 현상 등)을 줄이기 위해 화면 주변부에 그려지는 leaf들에 대해서는 SW Occlusion Culling을 생략한다.

S/W Occlusion Culling in Voxel World




Main스레드 - FindFunc()
이전에 요청한 SW Occ결과 확인
가려지는 leaf들 bit table에 표시
AllocContext()
[context]
Bit table참고해서 leaf와 삼각형 매시 수집
Leaf와 삼각형 매시들을 context에 저장 [context]
Awake Raster/Test Thread
Raster/Test 스레드측 큐에 push
가려지는 것으로 판단된 leaf들
Leaf-0
Leaf-1
Leaf-3
Leaf-7
Camera Position, Camera Angle
Context
Context
Context
Triangle meshes
(대략 1 – 16000개, max 65000 tris)
Leafs (대략 1 – 8000)개
Context
Context
Context
Triangle meshes
(대략 1 – 16000개, max 65000 tris)
Leafs (대략 1 – 8000)개
Raster/Test 스레드
Context]를 가져와서 S/W Occlusion Culling 수행
가려지는 leaf들을 결과버퍼에 저장
FreeContext()
[context]

생각해보기

- Async S/W Occlusion Culling을 애초에 뭐하러 해? 어차피 Async로 할거면 이전 프레임의 GPU측 Z-Buffer값을 읽으면 되지 않나?
- Async SWOcc일때 비동기로 GPU를 이용(CUDA/OpenCL/Compute Shader)해서 Raster/Test를 수행하면?

결론

- H/W Hierarchical Occlusion Culling은 무조건 사용할것. – 써서 안좋은 케이스를 본적이 없다.
- KD-Tree + Sync모드 SW Occlusion Culling는 GPU성능이 나쁠 때 최소한의 안전장치는 될 수 있다.
- Sync모드 SW Occlusion Culling은 최대프레임을 저하시킬 수 있다. 최대 프레임 향상을 위해서는 Async SW Occlusion Culling를 사용하는게 좋다.
- PVS를 사용하면 런타임에는 무조건 이득. 그러나 어떤 방식의 PVS든간에 개발기간중 별도의 빌드시간을 소모하므로 개발 기간 내내 사용할 필요는 없다.

참고자료

- https://megayuchi.com
- https://www.slideshare.net/dgtman/voxelizaition-with-gpu
- https://www.slideshare.net/dgtman/sw-occlusion-culling
- https://www.slideshare.net/dgtman/hierachical-z-map-occlusion-culling