멀티스레드
S/W Occlusion Culling

유영천

https://megayuchi.com

Tw: @dgtman

게임 성능에 대한 지표

- CPU코드의 병목으로 인해 GPU Queue에 패킷을 충분히 넣어주지 못한다 – GPU가 논다.
  - Draw call이 너무 많다
  - 렌더링에 진입하기 전에 비즈니스 로직 처리가 너무 길다.
- 너무 많은 삼각형, 너무 시간이 오래 걸리는 shader, 너무 큰 텍스처 – GPU에 과부하가 걸린다.
- GPU 점유율이 높다 -> GPU 부하가 크다 -> 더 좋은 GPU를 꽂으면 성능이 향상된다.
- CPU 점유율은 성능 지표가 되지 못한다.
  - 게임에서 멀티 코어를 100% 활용하는건 사실상 불가능하다.
  - CPU코드에서의 병목은 처리량이 아닌 응답성이다.

GPU점유율이 낮을 때(50% 이하)

- Draw call이 너무 많다.
- 오브젝트가 너무 잘게 쪼개져 있을 경우-draw call증가로 이어진다.
- 잘개 쪼개진 오브젝트를 그룹화 시켜서 draw call을 줄인다.
- 렌더링 직전까지 cpu코드에서 소모하는 시간을 줄이기
- 더 좋은 GPU를 꽂아도 성능은 거의 향상되지 않는다.

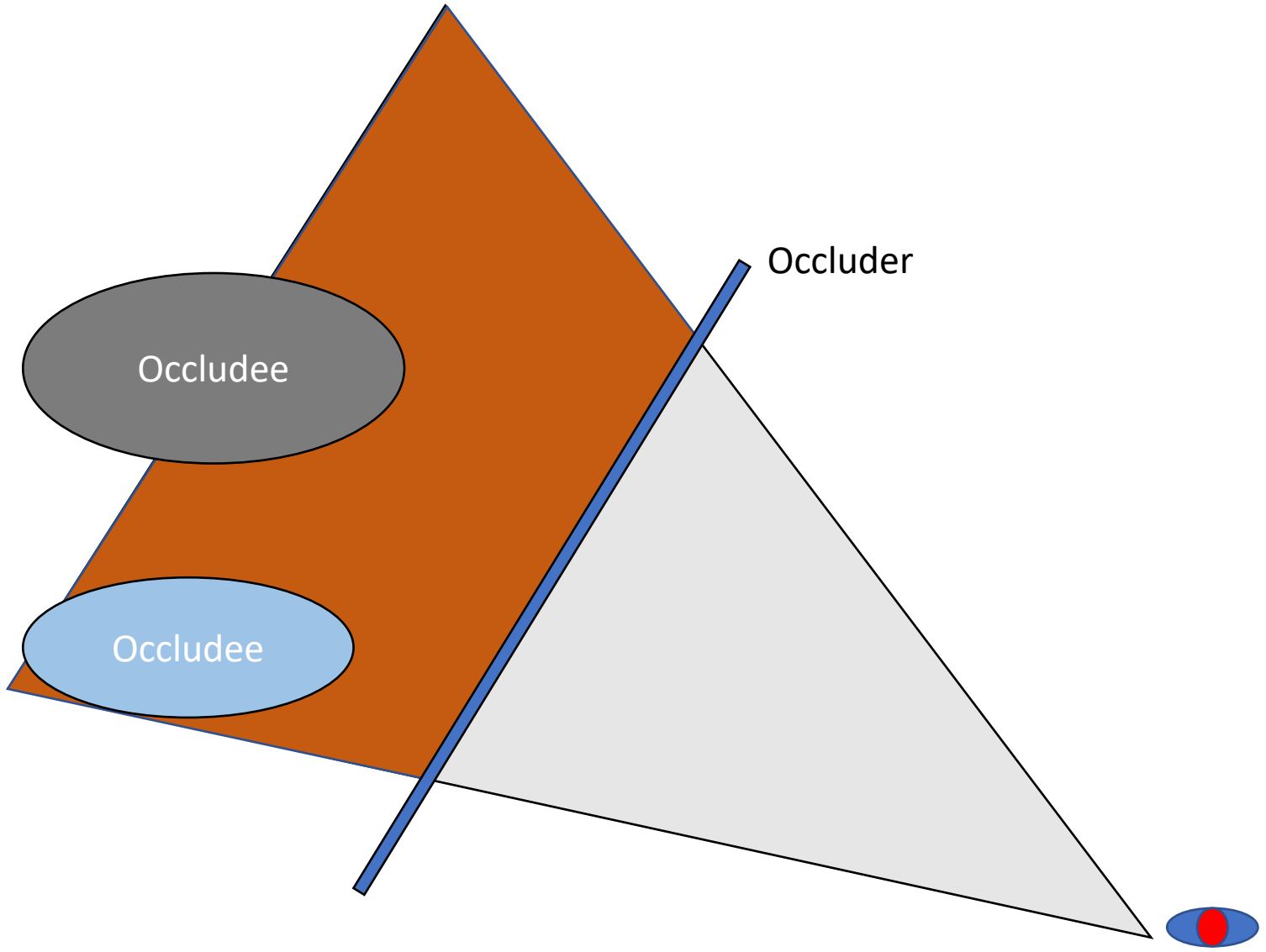
GPU점유율이 높을 때(90% 이상)

- 한번의 draw call당 다루는 리소스가 너무 클 경우
- 매시를 더 잘게 쪼개고 보이지 않는 오브젝트를 렌더링 하지 않는다.
- 쉐이더를 최적화 한다.
- 리소스 줄이기(삼각형 수 줄이기, 텍스처 사이즈 줄이기)
- 더 좋은 GPU를 꽂으면 성능이 향상된다.

Occlusion Culling

- 가려진 물체를 그리지 말자.
- Occluder가 Occludee를 가림
- Occludee가 가려지는지 어떻게 아느냐?
  - Z-buffer에다 그려보면 알지.
  - 이미 그래픽 API에서 삼각형을 그릴 때 늘 수행하고 있다.
  - 실제로 몇 픽셀 그렸는지 얻어낼 수도 있다.

- Occlusion Culling은 그래픽 API에 삼각형 데이터를 넘기기 전에 먼저 그려질지를 판별한다.


Occluder
Occludee
Occludee

HW Occlusion Culling

- 그려봐야 안다.
- 최대한 가볍게 그린다-GPU부하 감소
- 가벼운 렌더링을 먼저 수행해서 무거운 렌더링 렌더링을 최대한 피한다.
- 그려보기 위해서 draw call이 증가한다.
- GPU메모리부터 결과값을 읽어와야 한다.
  - D3D12에선 결과값을 읽어오지 않는 형태의 HW Occlusion Culling이 가능하다. 하지만 draw call증가는 막을 수 없다.
- GPU성능이 충분할 경우 되려 성능 저하
- 상대적으로 GPU 성능이 떨어질 때 좋은 효과를 얻을 수 있다.

SW Occlusion Culling

- 그려봐야 안다.
- 그래픽 API를 호출하기 전에 CPU코드로 최대한 단순하게 그려본다.
- CPU는 삼각형을 그리기 좋은 장치가 아니다. 느리다.
- Draw call 회수와 GPU부하를 줄이지만 draw call전까지 CPU시간을 너무 잡아 먹어서 느려진다.
- CPU클럭이 빠를수록 효과적.
- 그래픽API호출과 무관하게 작동한다.
- 처리량은 떨어지지만 응답성은 뛰어나다.
- 제한적 사용에 따라 성능향상을 기대할 수 있다.

SW Occlusion Culling의 재발견

- KD-Tree와 같은 자료구조를 순회하며 카메라에 가까운 오브젝트부터 탐색할 수 있다.
- 카메라에 가까운 오브젝트를 z-buffer에 그려가면서 다음 탐색할 노드가 z-buffer상에서 가려지는지를 판별할 수 있다.
- 부모 노드가 보이지 않는 것으로 판별되면 모든 자식 노드 또한 보이지 않는다.
- 카메라에 가까울수록 크게 그려진다.
- 따라서 모든 오브젝트를 그리고 테스트할 필요 없이 가까운 오브젝트 몇 개만 그리고 가까운 노드 몇 개만 테스트 해도 보이지 않는 오브젝트의 50%이상을 제외시킬 수 있다.

KD-Tree

- X,Y,Z 축에 정렬된 BSP Tree.
- 각 node를 3가지 축과 축방향의 거리 D값으로 표현할 수 있다.
- KD-Tree를 탐색할 때 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?

Occlusion Culling in KD-Tree

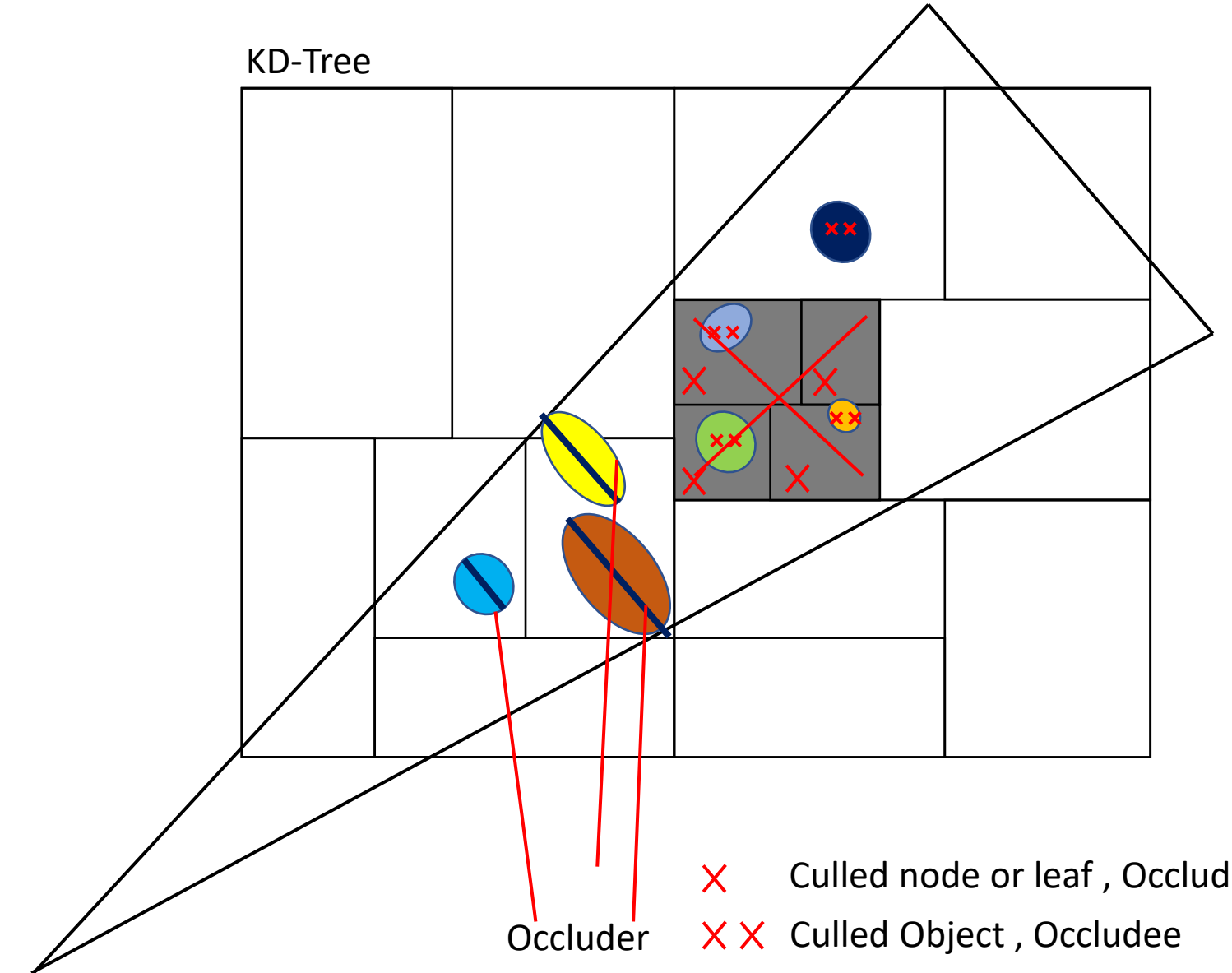
- node째로 culling하면 하위 node와 leaf에 포함된 오브젝트들도 같이 culling된다.
- KD-Tree탐색 단계에서 node에 대해서 Occlusion Culling을 수행하면 렌더링될 오브젝트들을 원천적으로 줄일 수 있다.
- 가까운 node에서 먼 node로 탐색해가면서 가까운 node(leaf)의 오브젝트를 z-buffe에 test& rasterize 한다.
- KD-Tree를 사용하면 가까운 node와 먼 node를 구분해서 가까운 node부터 탐색할 수 있다.(KD-Tree Traversal)
  - 가까운 곳의 물체는 크게 보인다.
  - 따라서 최초 몇번의 z-test만으로도 상당히 많은 수의 가려지는 오브젝트들을 걸러낼 수 있다.

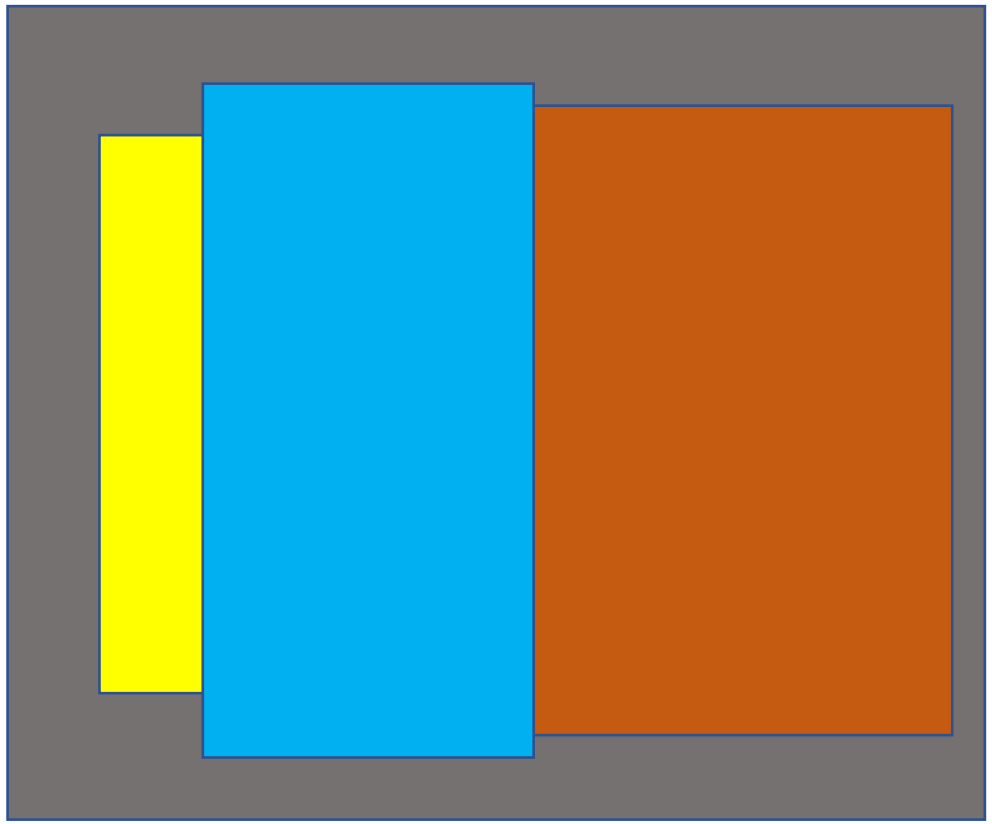
KD-Tree + SW Occlusion Culling을 이용한 오브젝트 수집

- Node에 대해 view frustum culling
- Node의 AABB-를 육면체(삼각형 12개)를 z-test. z-test결과 렌더링 되는 픽셀이 하나도 없으면 다음 node로.
- Node에 대해 view frustum culling. Frustum에 포함되지 않으면 다음 node로.
- Leaf이면 leaf가 포함하는 오브젝트 수집
- 오브젝트에 대해 view frustum culling.
- 오브젝트에 대해 z-write수행. 오차/성능 문제를 피하기 위해 z-test는 수행하지 않음
- 다음 node로 진행

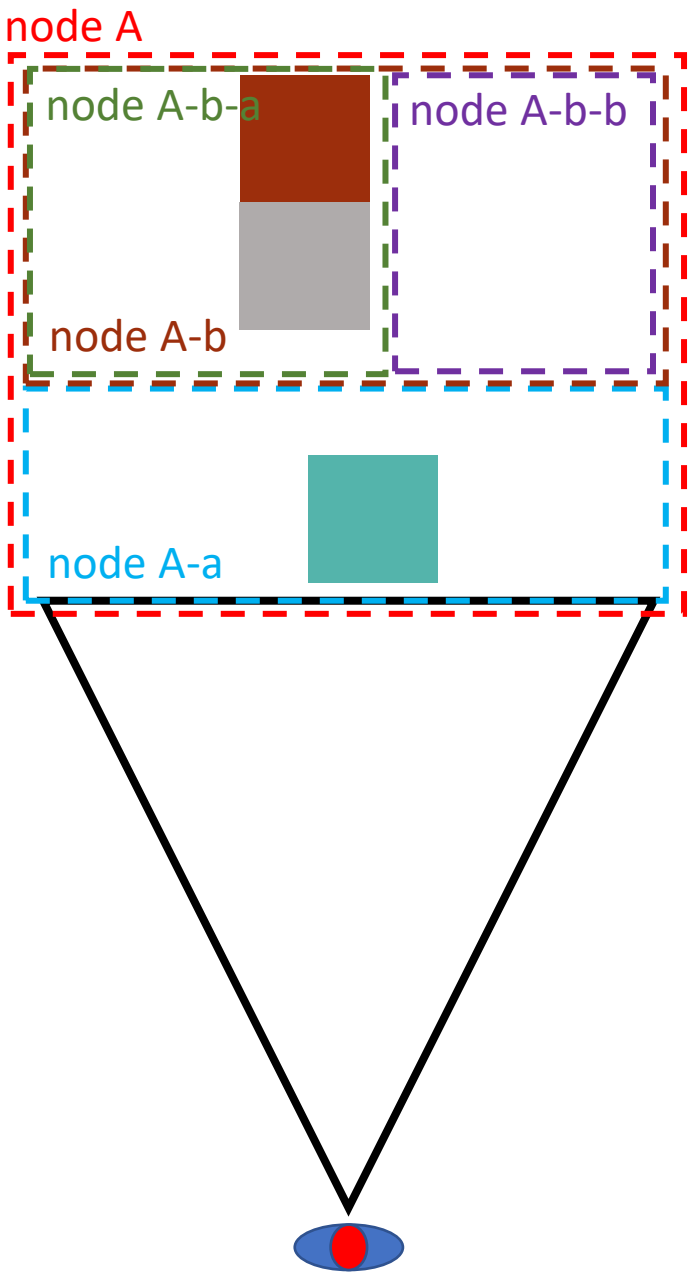
SW Occlusion Culling in KD-Tree

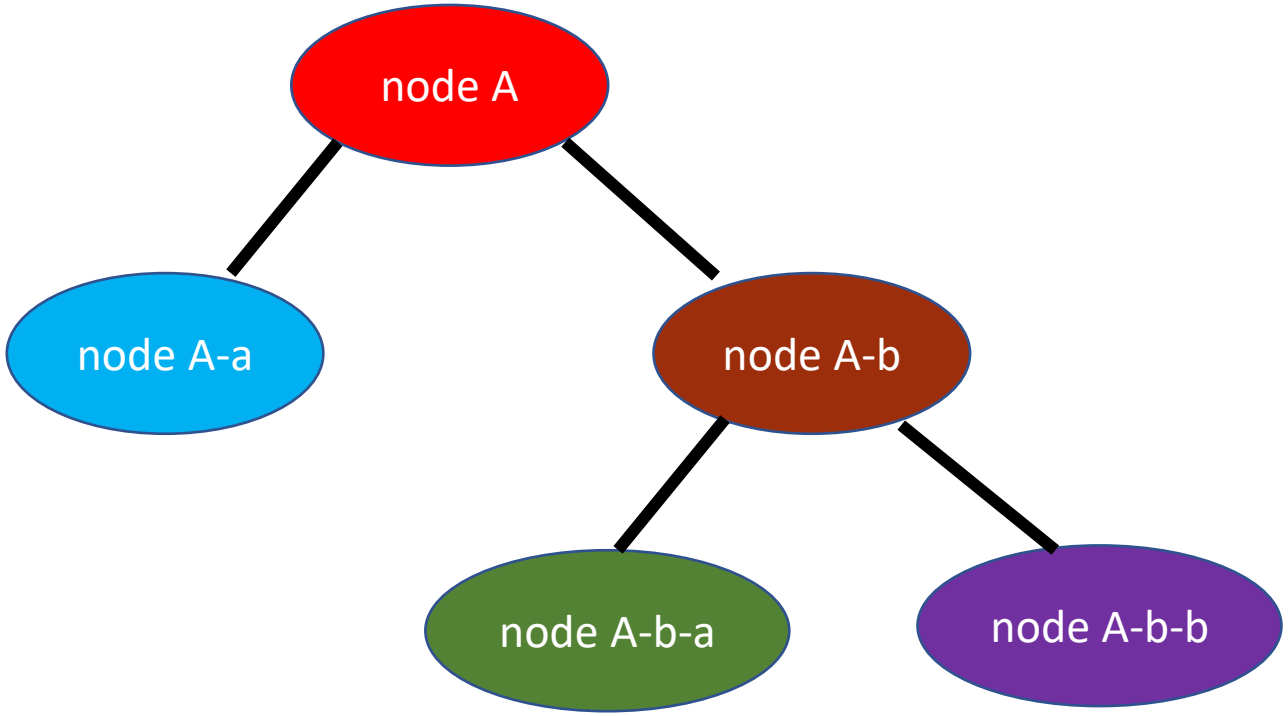
KD-Tree순회중에 먼저 발견한 오브젝트들(Occluder)을 z-buffer에 그린다. 이로 인해 다음번 순회할 node(or leaf)를 통째로 제외시킬수 있다.



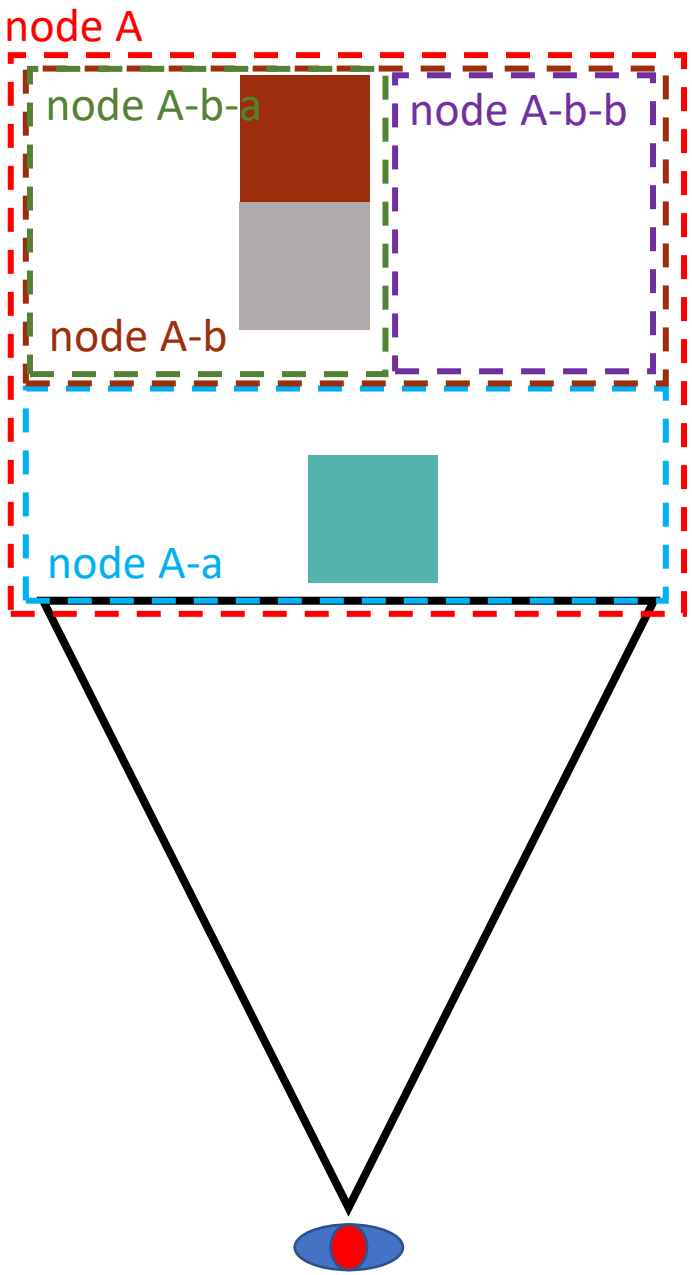


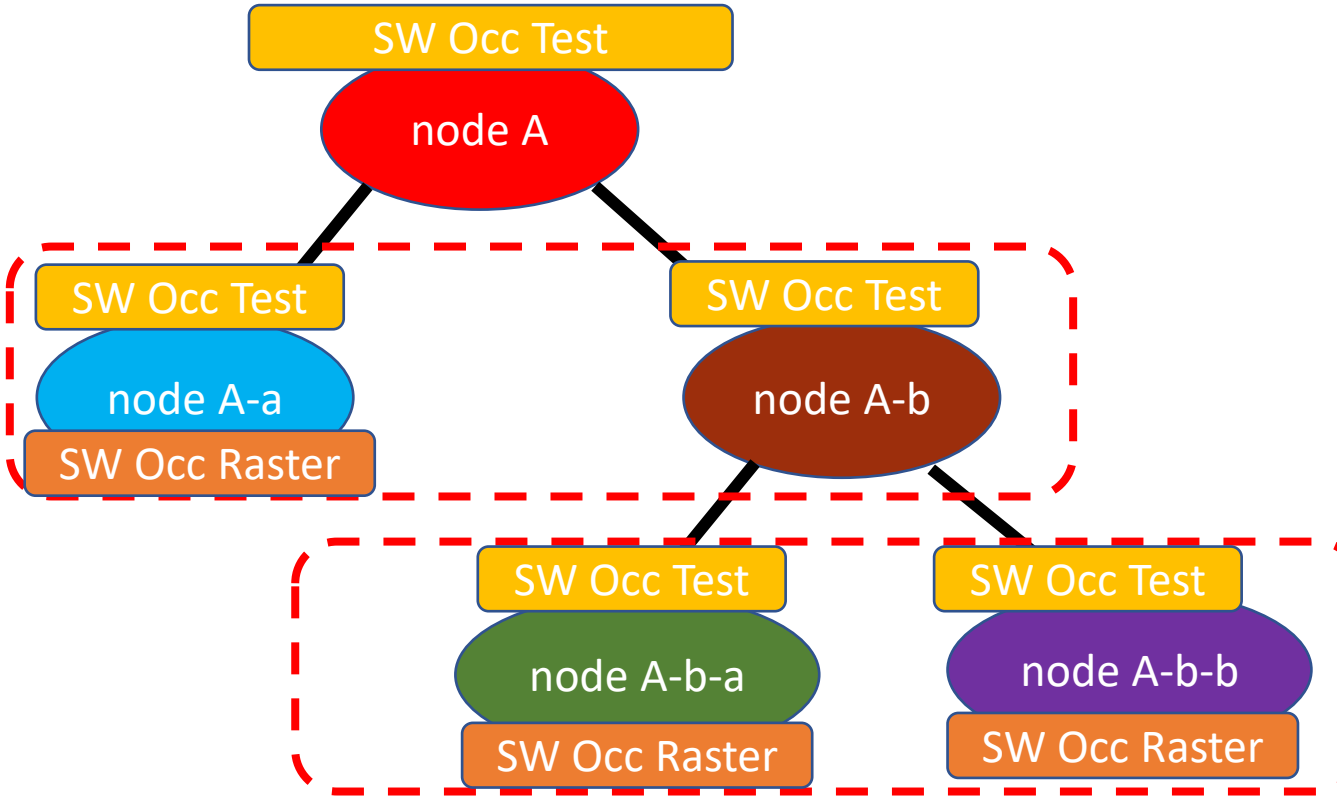
KD-Tree Traversal





KD-Tree Traversal + Raster/Test





SWOcc Test를 통과한 노드 A방문 : 노드 A-a, 노드A-b를 test
노드 A-a를 방문 : 자식 노드가 없으므로 leaf, 가지고 있는 오브젝트를 raster
노드 A-b를 방문 : 노드 A-b-a와 노드 A-b-b를 test
노드 A-b-a를 방문 : 자식 노드가 없으므로 가지고 있는 오브젝트를 raster
노드 A-b-b를 방문 : 자식 노드가 없으므로 가지고 있는 오브젝트를 raster

멀티스레드 적용

SW Occlusion Culling에 멀티스레드 적용

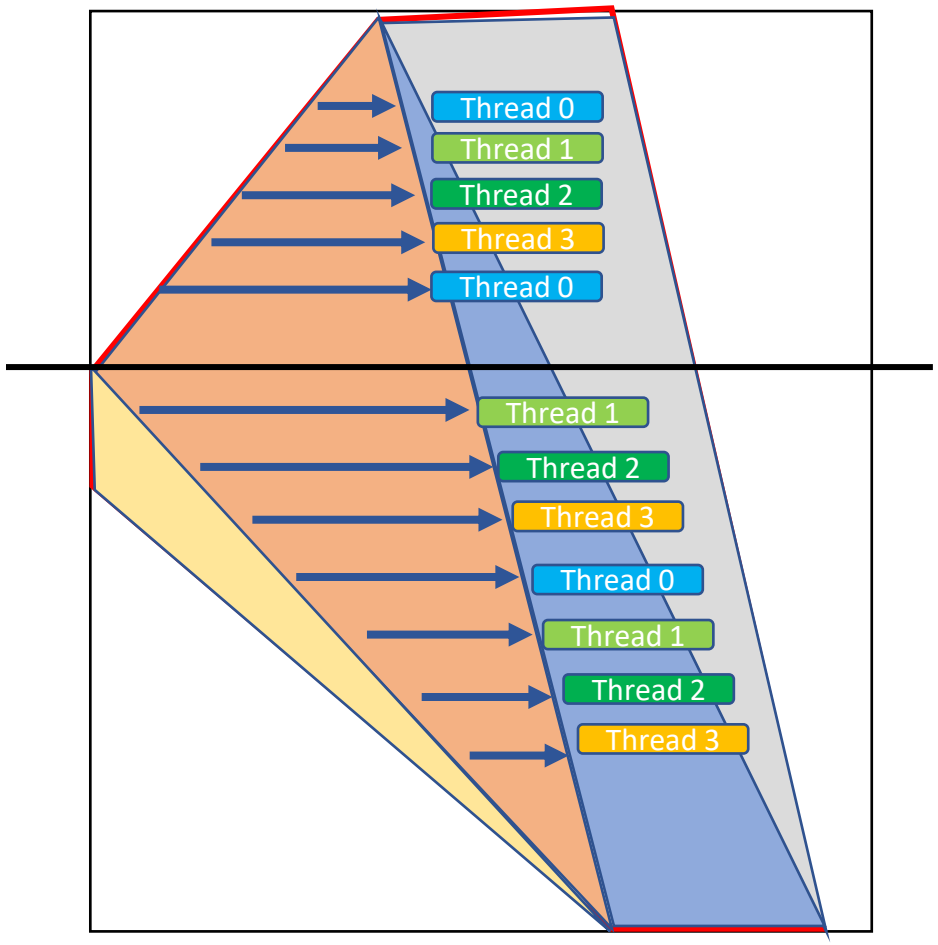
- 가까운 오브젝트 몇 개만 그리고 가까운 노드 몇 개만 테스트한다 해도 CPU코드로는 여전히 느리다.
- CPU클럭이 엄청나게 빠르면 큰 도움이 되지만 클럭주파수 향상은 거의 정체되어 있다.
- 따라서 멀티 코어를 활용해서 SW Occlusion Culling 성능을 높일 방법을 찾아야한다.

초기 접근

- 멀티스레드로 작동하는 SWOcclusion Tester(DLL엔진) 개발
- 입력 : 삼각형 리스트
- 출력 : z-buffer, rasterizaition으로 그려진 픽셀 개수, test로 통과한 픽셀 개수
- 트리 순회시 노드를 방문할 때마다 SWOcclusion 스레드가 wait/awake를 반복한다.

라인 당 스레드

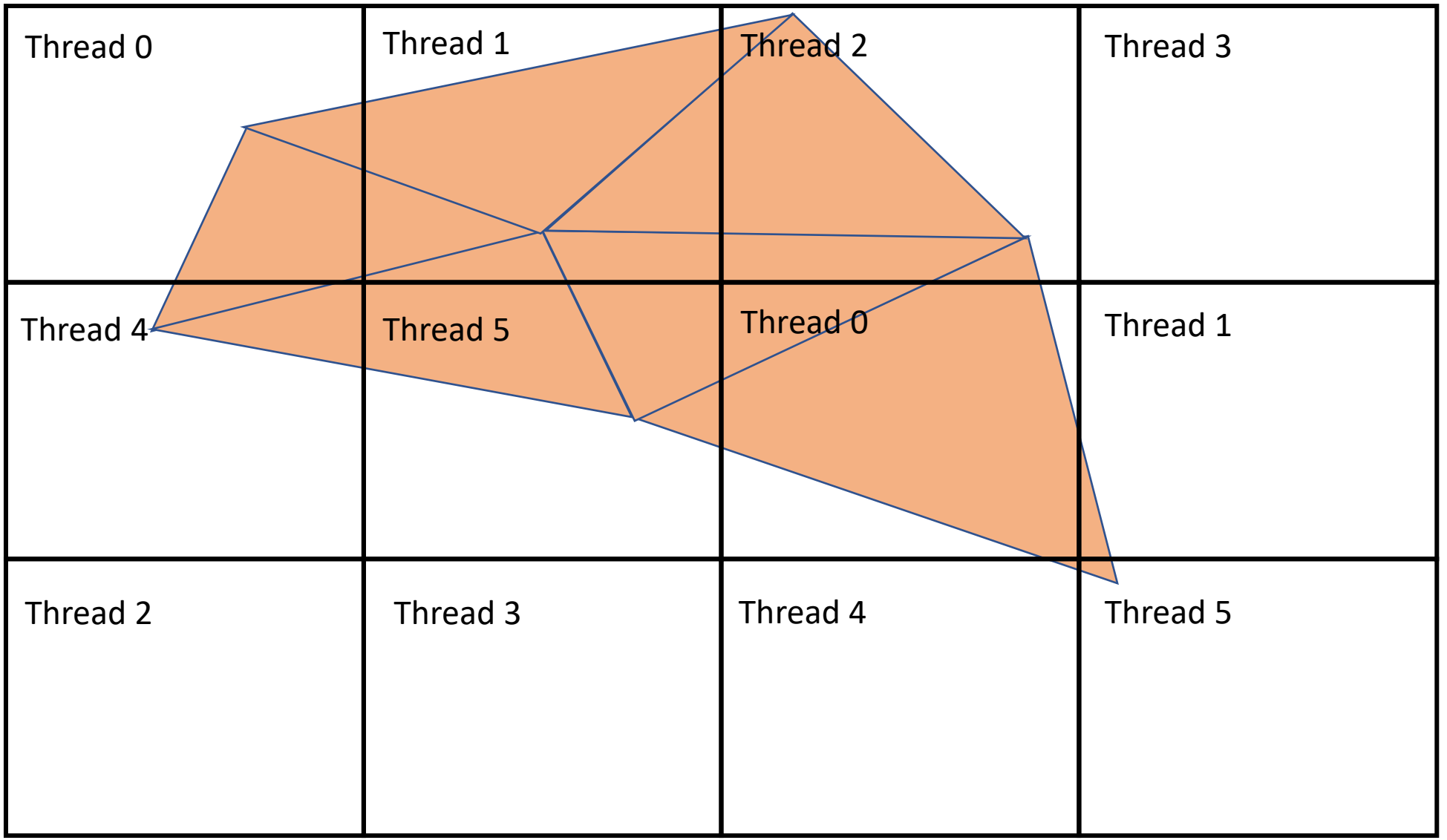
- Z-buffe의 동일 픽셀에 여러 스레드가 접근할 수 있다
- Lock(atomic operation까지 포함해서)이 필요하다.
- Lock때문에 느려짐.
- Lock을 줄이기 위해 스레드 인덱스x스레드 개수만큼 스캔라인(y좌표)를 할당해서 스레드당 한 라인씩 독립적으로 그려넣고 테스트한다.
- 삼각형 변환에선 멀티스레드 적용을 못 받는다.
- 결론적으로 느리다....


Thread 0
Thread 1
Thread 2
Thread 3
Thread 0
Thread 1
Thread 2
Thread 3
Thread 0
Thread 1
Thread 2
Thread 3

Rasterize

화면을 분할하고 무게중심좌표를 이용하여 raster/test

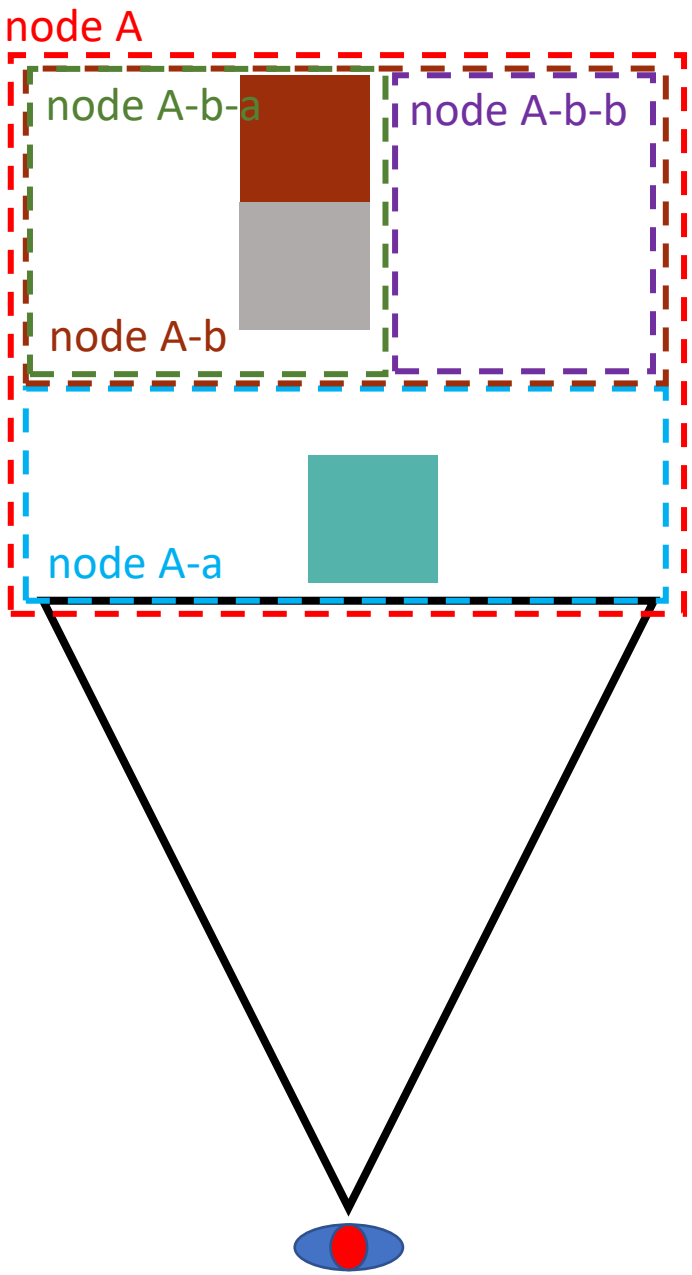
- 버퍼를 스레드 개수만큼의 공간으로 분할
- 삼각형 입력이 들어오면 분할된 공간에 삽입
- 각 스레드는 자신의 영역 안에서 픽셀들이 삼각형 안에 포함되는지 테스트.
- Lock이 필요없다.
- SIMD를 사용하면 비례해서 성능 향상
- 오버헤드 : 분할된 공간에 삼각형 삽입
- 오버헤드 : 불필요한 픽셀들에 대한 테스트 비용
- 결론적으로 싱글스레드보다 느리다.
  SSE < AVX <= 싱글스레드였으니까 AVX256까지 사용한다면 싱글스레드보다 빠를지도…

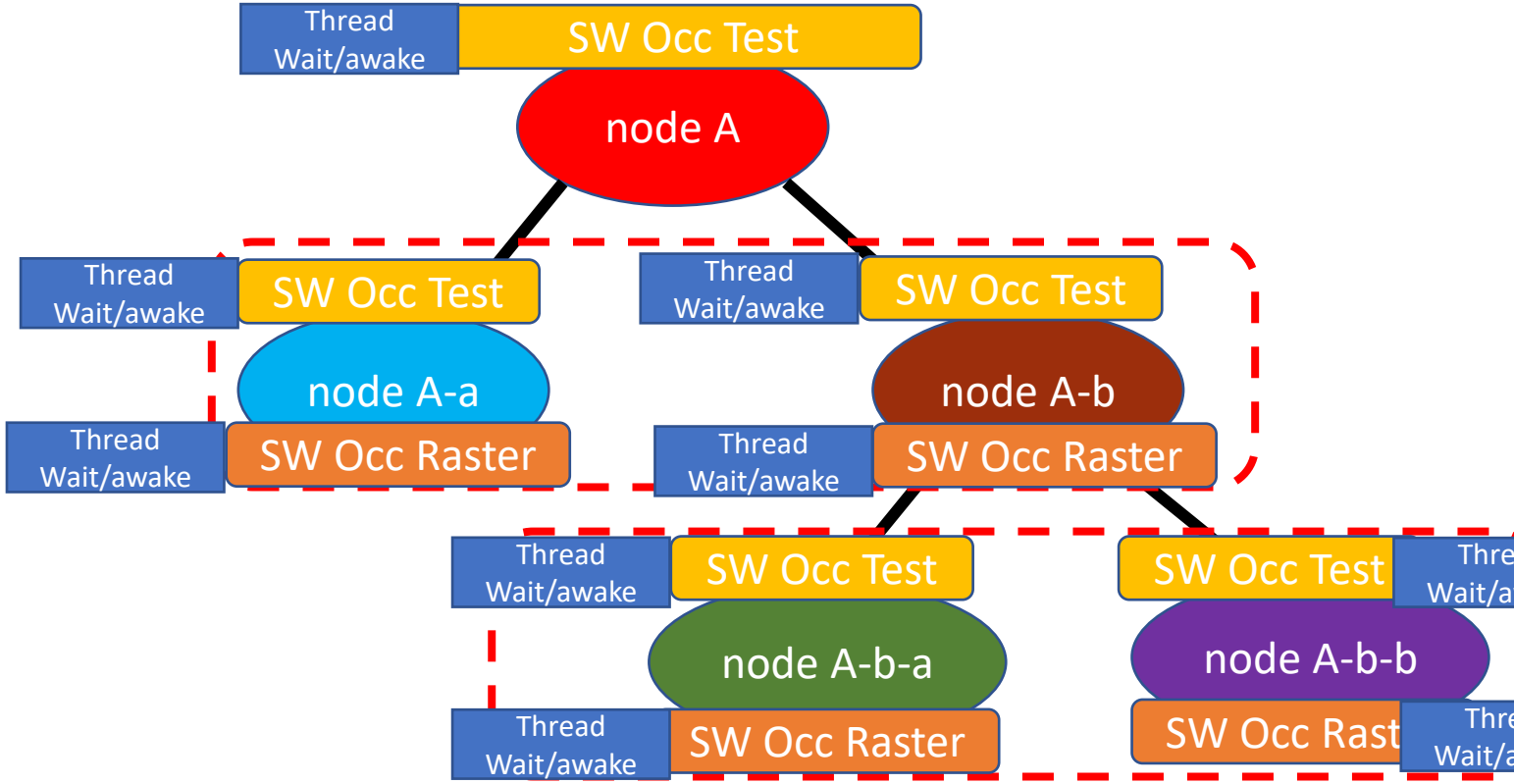

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5

왜 느린가?

- Lock(atomic 연산 포함)을 깔끔하게 제거할 방법이 없다.
- SWOcclusion Tester내부의 스레드 작동 방식이나 삼각형 그리기 방법을 어떻게 바꿔도 더 느려질 뿐이다. 처리량은 향상되지만 응답성은 결코 향상되지 않는다.
- 삼각형 처리 소요시간보다 스레드가 깨어나서 스케쥴링 될 때까지의 딜레이가 더 크다.

트리 순회중 스레드 스케쥴링 딜레이

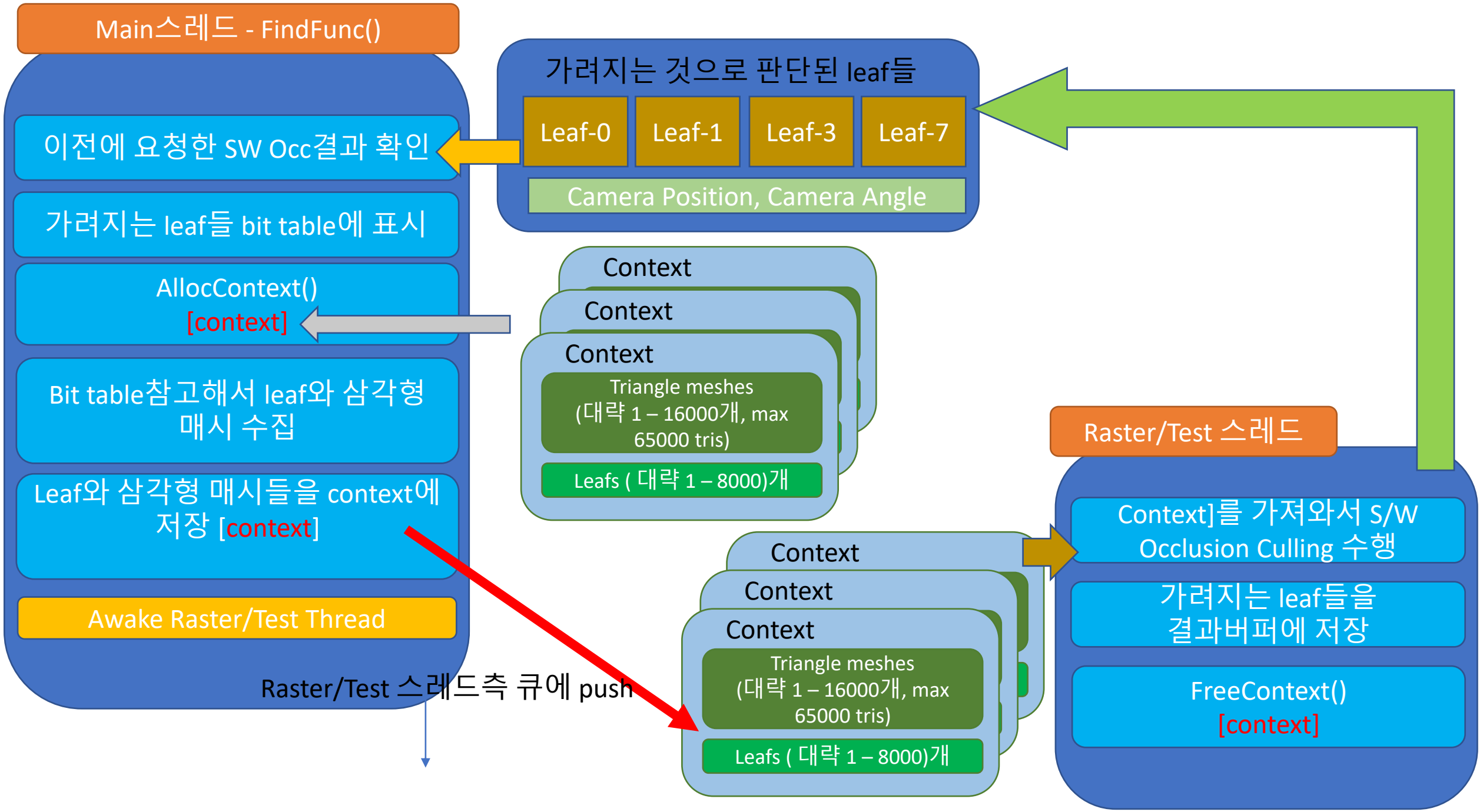




SWOcc Test를 통과한 노드 A방문 : 노드 A-a, 노드A-b를 test
노드 A-a를 방문 : 자식 노드가 없으므로 leaf, 가지고 있는 오브젝트를 raster
노드 A-b를 방문 : 노드 A-b-a와 노드 A-b-b를 test
노드 A-b-a를 방문 : 자식 노드가 없으므로 가지고 있는 오브젝트를 raster
노드 A-b-b를 방문 : 자식 노드가 없으므로 가지고 있는 오브젝트를 raster

비동기 SW Occlusion Culling

- 최근까지 사용한 방법.
- 현재 프레임의 씬 상태를 기준으로 비동기 SW Occluion Manager에 SWOcc작업을 요청-> 백그라운드에서 처리
- SW Occlusion Culling 작업 자체는 싱글 스레드로 처리
- 다음 프레임에 이전 결과를 바탕으로 렌더링
- 60FPS 기준으로 16ms 의 비교적 여유있는 작업 시간을 확보가능.
- 적어도 렌더링 속도를 까먹지는 않는다.


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]

비동기 SW Occlusion Culling의 단점

- 이전 프레임에 대한 Occlusion Culling이므로 오차가 크다.
- 특히 화면 모서리 부분이 크게 문제가 됨.
- 오차문제를 해결하기 위해 온갖 꼼수를 동원-> 코드가 복잡해짐
- 비동기 처리 자체만으로 코드가 대단히 복잡해짐.
- CPU코어 개수가 많아져도 성능향상이 없음.

절두체 분할 멀티 스레드 렌더링

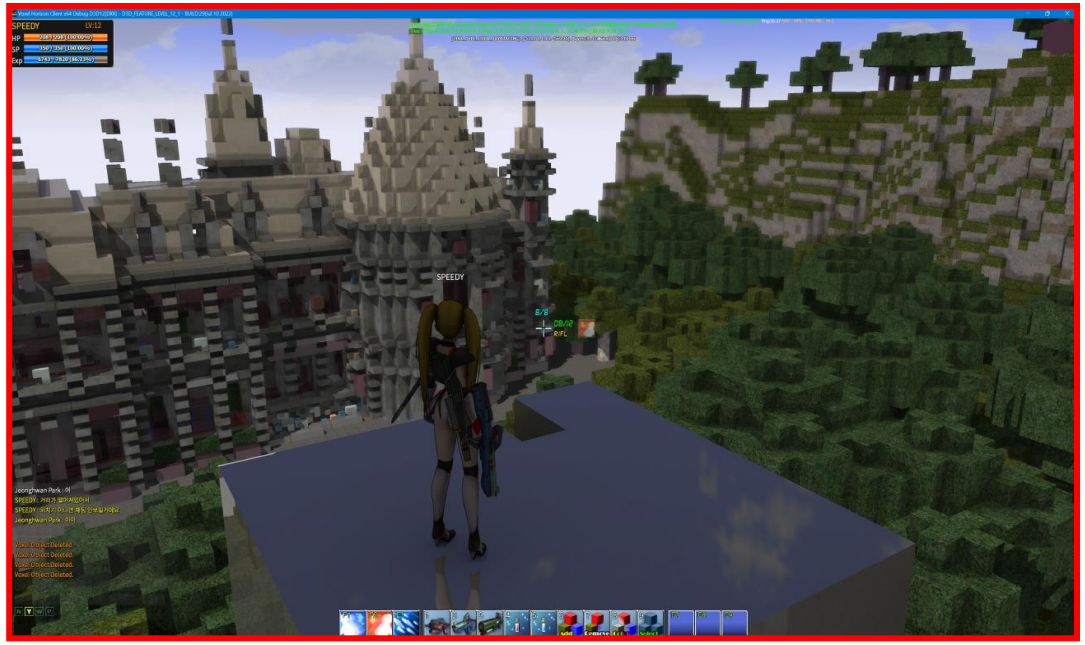
탐색단계부터 멀티 스레드를 적용해볼까?

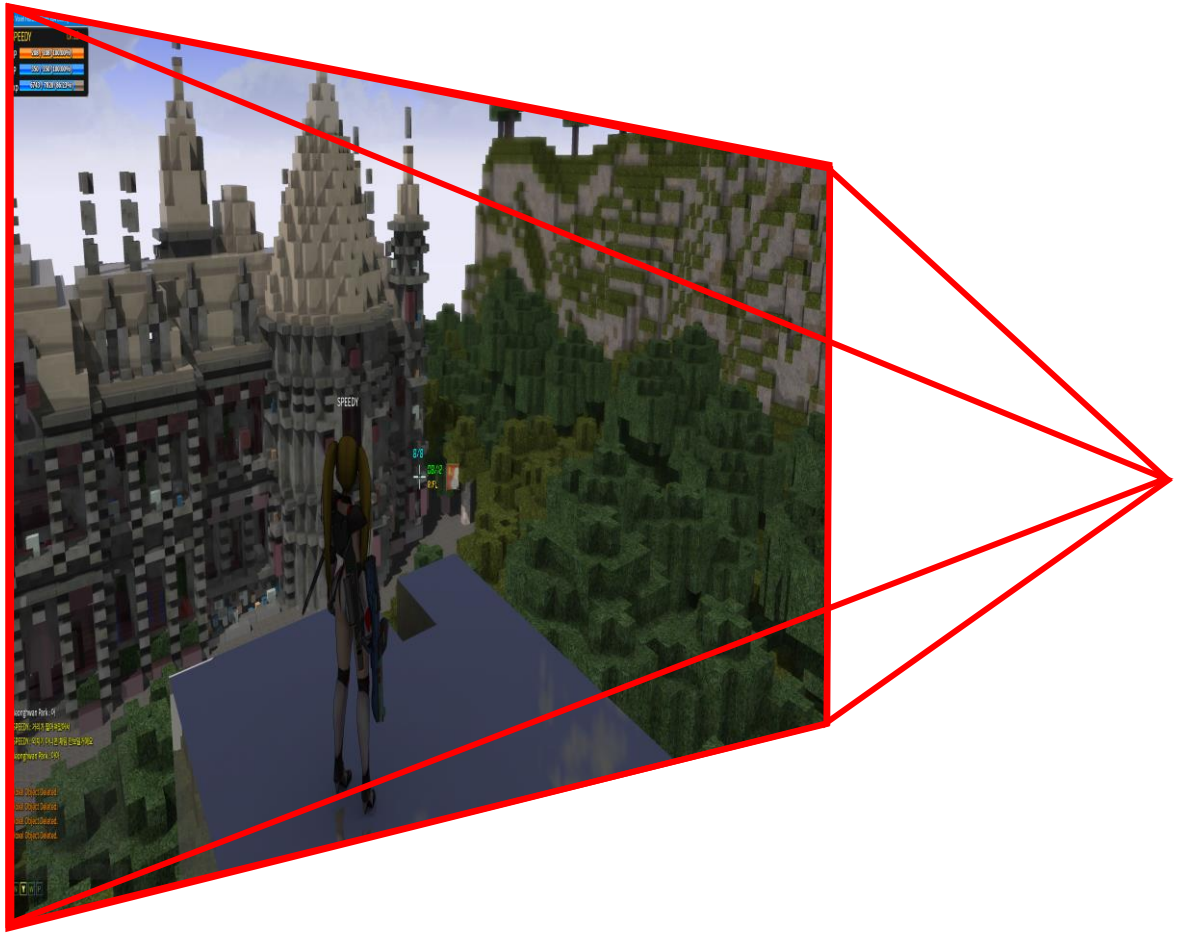
- 새벽에 운동하다 생각남.
- SWOcclusion Tester가 멀티스레드 동작할 것이 아니라 트리를 멀티스레드로 탐색하면 안되나?

화면 분할(절두체 분할) 멀티스레드 렌더링

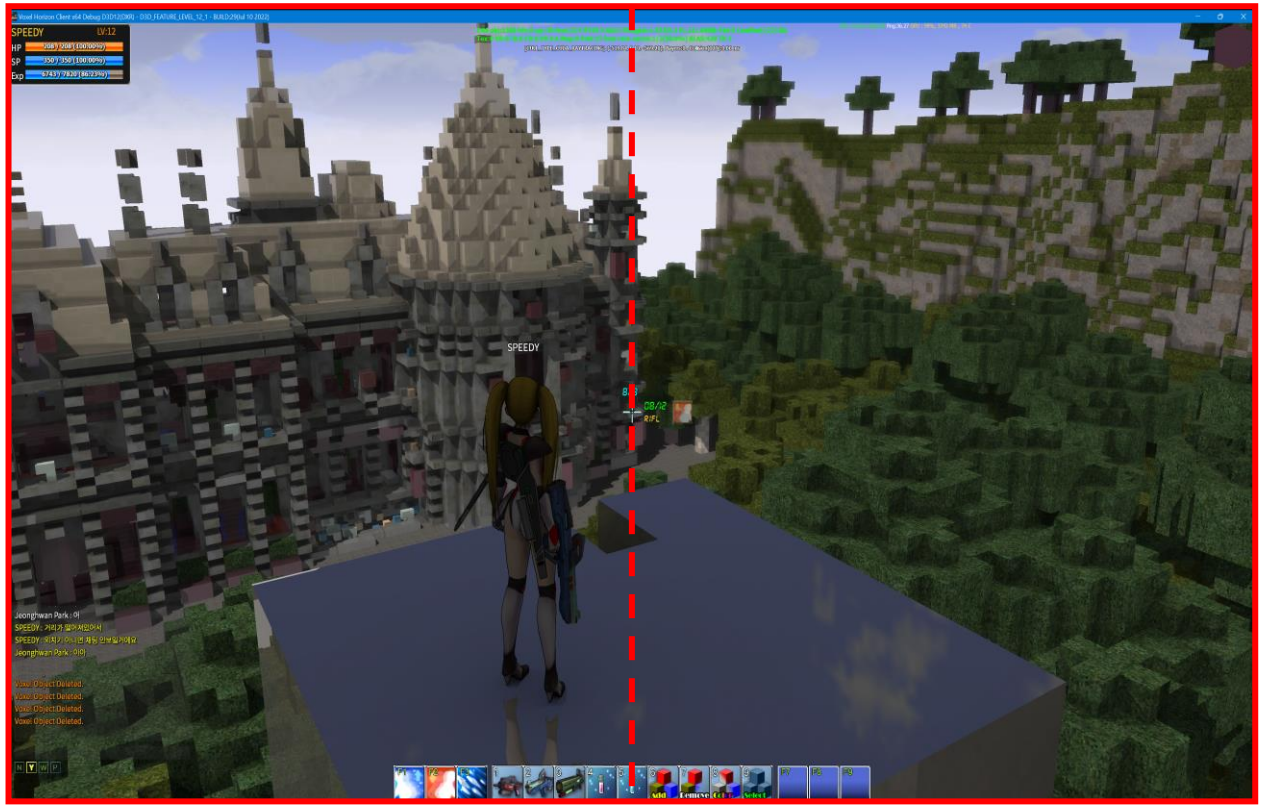
- 월드 공간상에서 카메라 영역(view frustum)이 작아지면
  - 트리 탐색 시간이 줄어든다.
  - 테스트할 오브젝트와 노드 개수가 줄어든다.
  - 대응하는 z-buffer사이즈가 줄어든다.
  - z-buffer사이즈가 줄어들었으므로 raster/test 시간도 줄어든다.
- 트리탐색/SW Occlusion Culling을 하나의 스레드가 처리한다.
- 화면에 대응하는 뷰프러스텀을 분할한다. 분할된 각 프러스텀에 스레드 하나씩을 할당해서 트리탐색과 함께 SW Occlusion Culling을 수행한다.

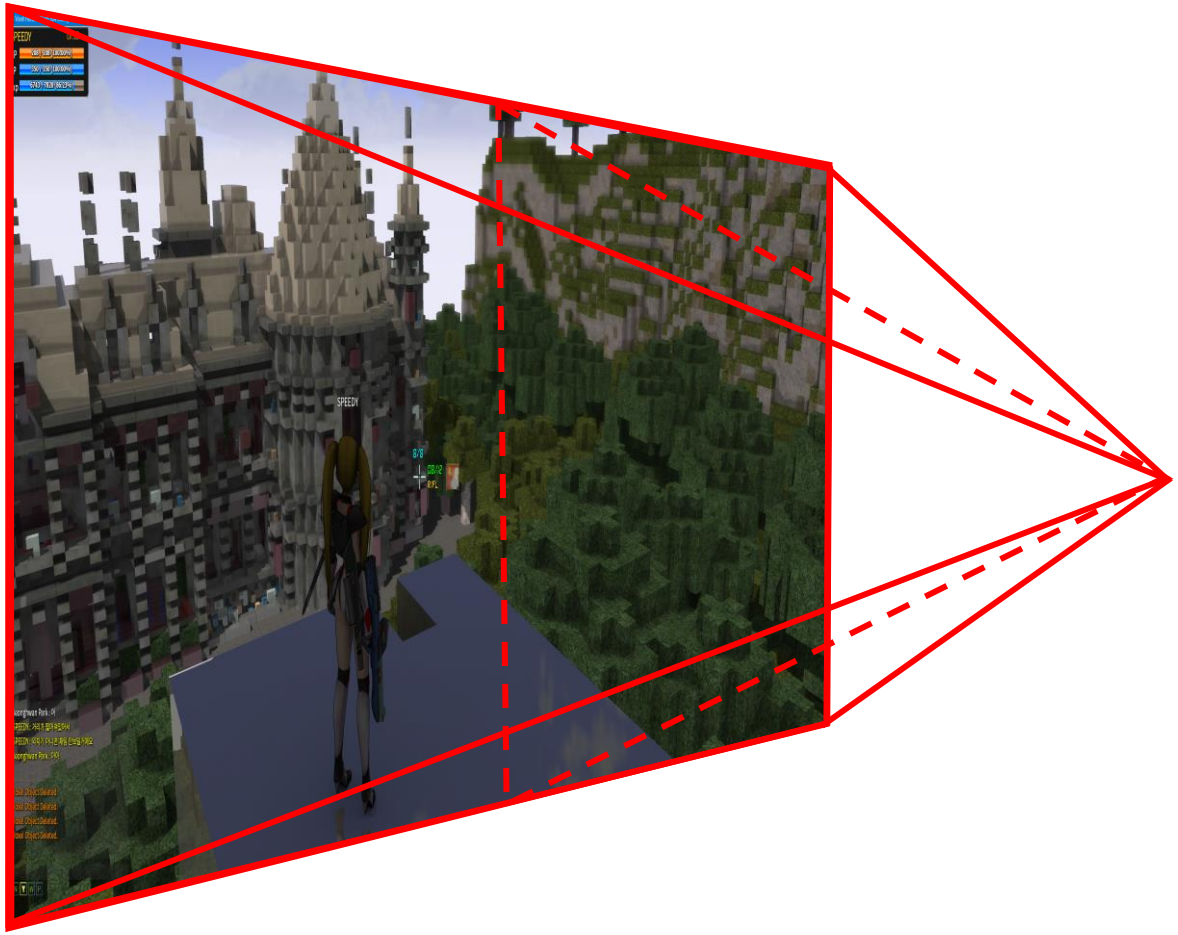
절두체 분할 없음





절두체 2분할 – 2 스레드





SWOcclusion Tester는 스레드 세이프해야한다.

- 여러 스레드가 동시에 진입할 것이므로 스레드 세이프 해야한다.
- 스레드별로 컨텍스트를 가진다.
  - z-buffer메모리(읽기/쓰기)
  - 결과 버퍼 메모리(쓰기)
  - 카메라 상태/삼각형 리스트(읽기전용->스레드간 공유 가능)
  - Voxel World Tree/Tri Mesh Tree(읽기전용 -> 스레드간 공유 가능)
- 각 스레드는 자신의 컨텍스트만 억세스하므로 lock없이 간단히 구현 가능.

ndc좌표계로 절두체-6개의 평면방정식 얻기

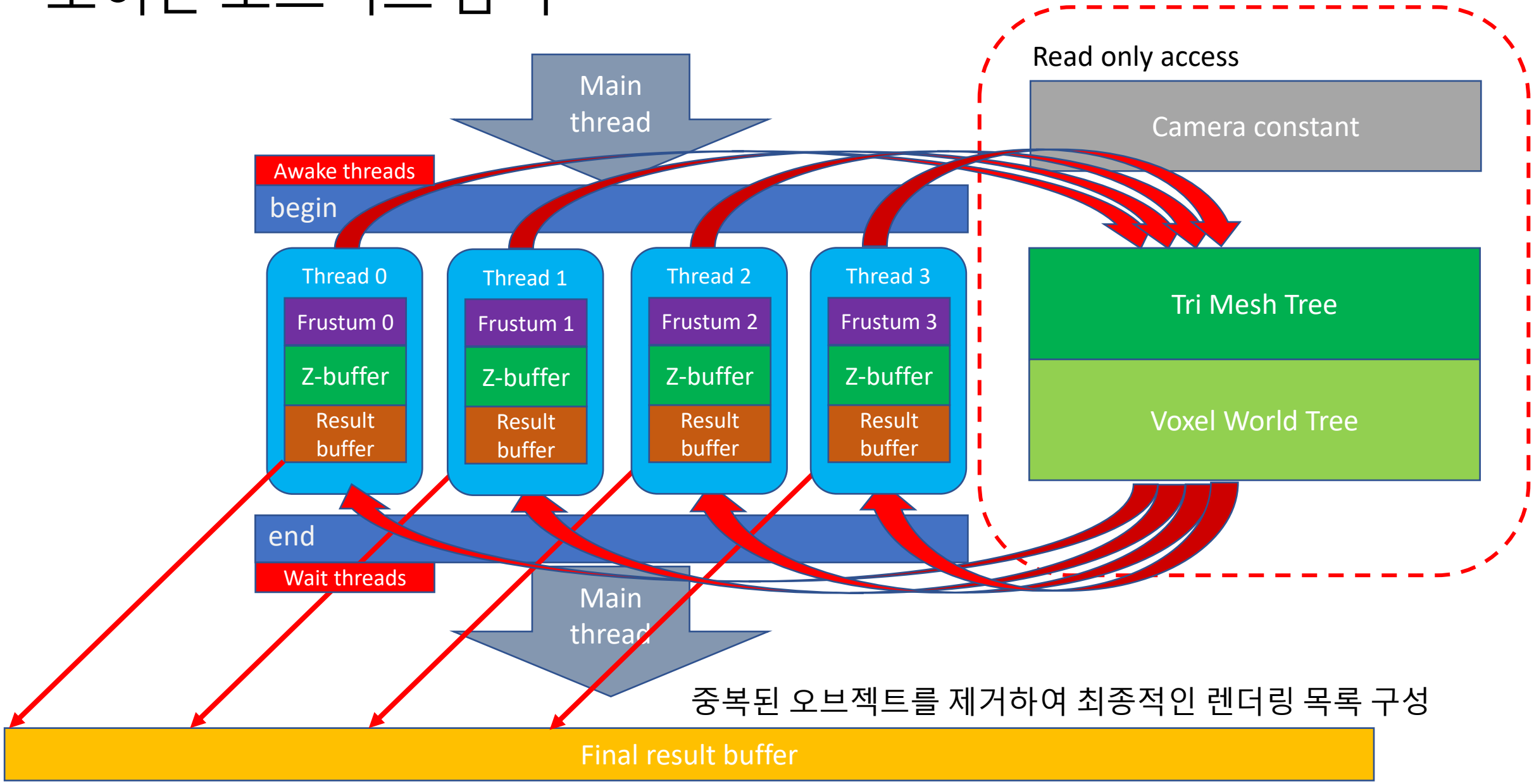
- (x,y,z,1)좌표를 View x Proj 로 변환 후 w로 나누면 ndc좌표를 얻는다.
  - 화면 귀퉁이의 가장 먼 점 : (-1,-1,near) , (1, -1,near) , (1,1,near) , (-1 ,1,near)
  - 화면 귀퉁이의 가장 가까운 점: (-1,-1,far) , (1, -1,far) , (1,1,far) , (-1 ,1,far)
- Proj 행렬의 각 성분들을 역으로 곱해서 view공간의 점 8개를 구한다.
- View 공간의 점 8개 x view역행렬 = 월드 공간에서의 점 8개
- 8개 점으로 6개의 평면방정식을 구할 수 있다.

절두체 분할

- 같은 방법으로
  - (-1,-1,near) , (0.5, -1,near) , (0.5,1,near) , (-1 ,1,near) : 화면 $\frac{1}{2}$ 의 가까운 점
  - (-1,-1,far) , (0.5, -1,far) , (0.5,1,far) , (-1 ,1,far) : 화면 $\frac{1}{2}$ 의 먼 점
  - (-0.5,-1,near) , (1, -1,near) , (1,1,near) , (-0.5 ,1,near) : 화면$\frac{2}{2}$의 가까운 점
  - (-0.5,-1,far) , (1, -1,far) , (1,1,far) , (-0.5 ,1,far) : 화면$\frac{2}{2}$의 먼 점

2분할 절두체의 평면 방정식을 각각 구할 수 있다.

보이는 오브젝트 탐색


Read only access
Camera constant
Main thread
Awake threads
begin
Thread 0
Frustum 0
Z-buffer
Result buffer
Thread 1
Frustum 1
Z-buffer
Result buffer
Thread 2
Frustum 2
Z-buffer
Result buffer
Thread 3
Frustum 3
Z-buffer
Result buffer
Tri Mesh Tree
Voxel World Tree
end
Wait threads
Main thread
중복된 오브젝트를 제거하여 최종적인 렌더링 목록 구성
Final result buffer

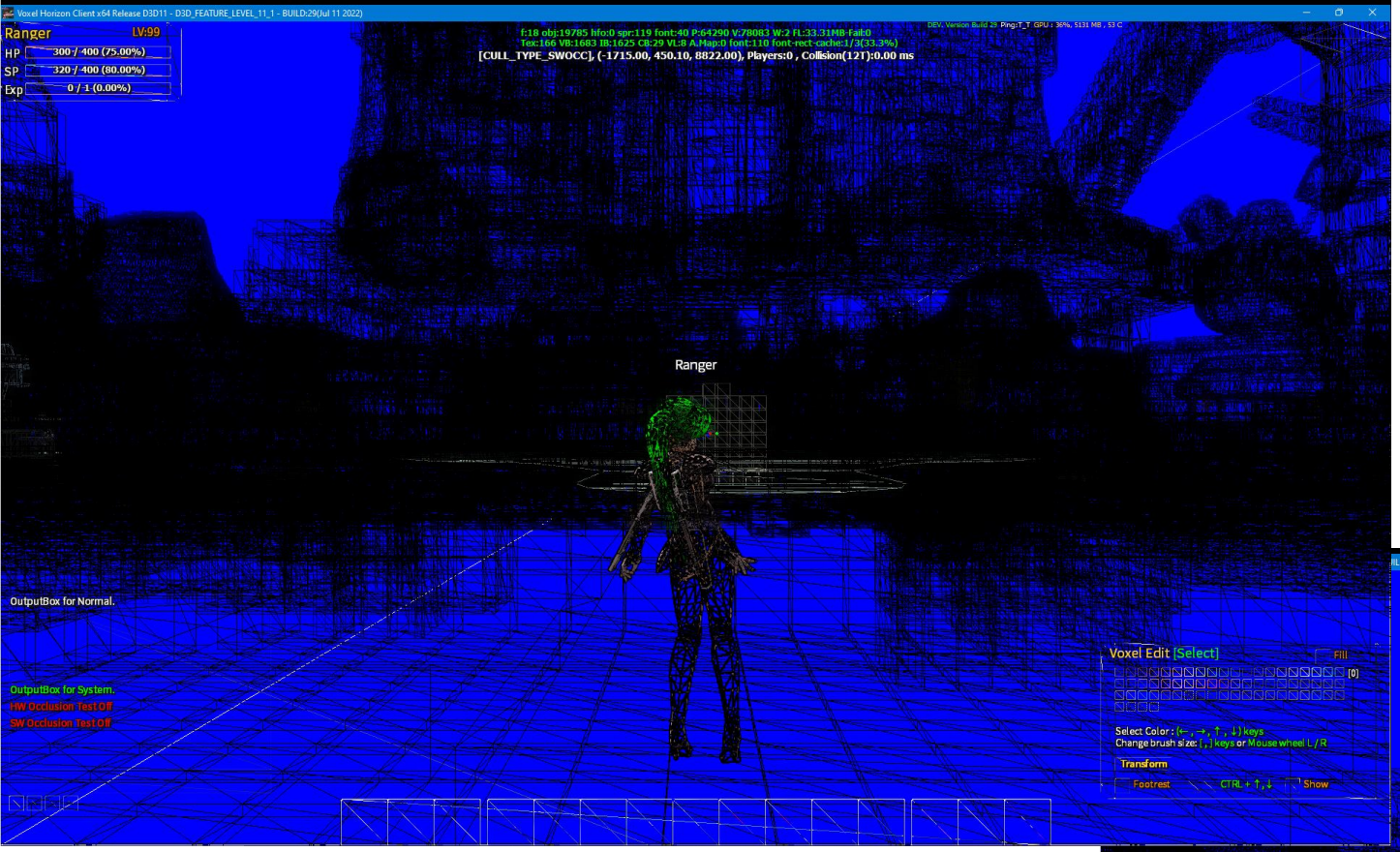
장점

- Lock이 필요 없다.
- 코드가 대폭 간단해 진다.
- SW Occlusion Culling을 사용하지 않을 때도 트리 탐색 시간을 줄일 수 있다.
- 스레드 개수에 따라 선형에 가까운 성능 증가(가장 중요)

단점

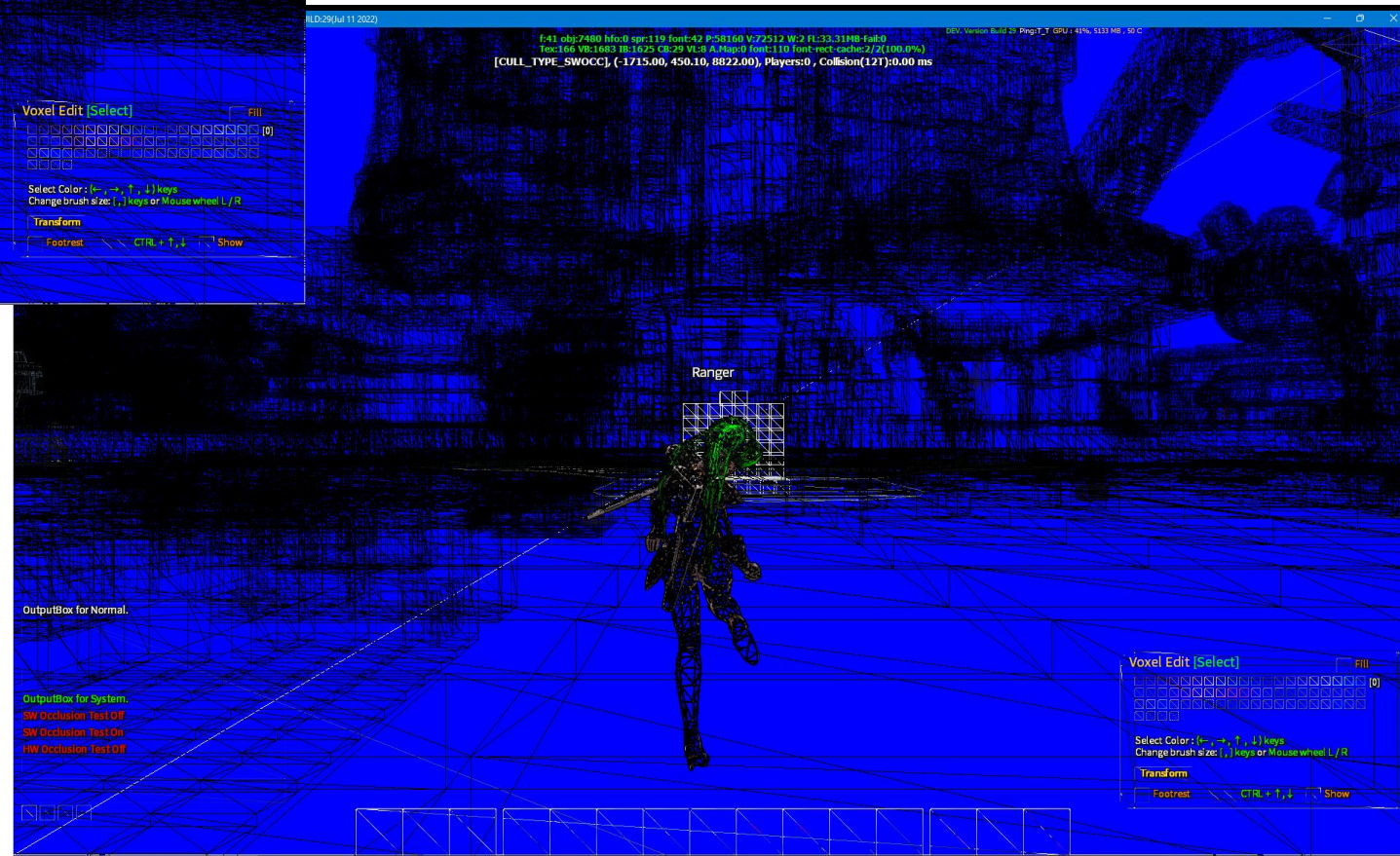
- 스레드 개수가 증가한다고 무한정 빨라지는 것은 아니다.
- 한 프레임 늦게 결과를 반영하는 비동기 SW Occlusion Culling보다 명백하게 빠른 것은 아니다.
- 코어 개수가 2개 이하인 경우 기존 방식보다 성능이 높지 않다.

성능 테스트



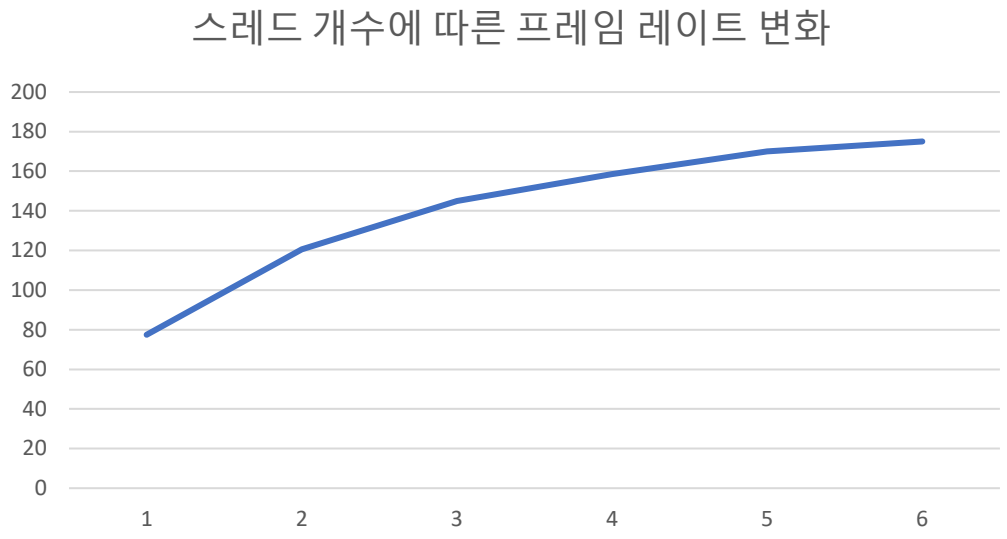
SW Occlusion Culling Off
렌더링 되는 오브젝트 : 19785개

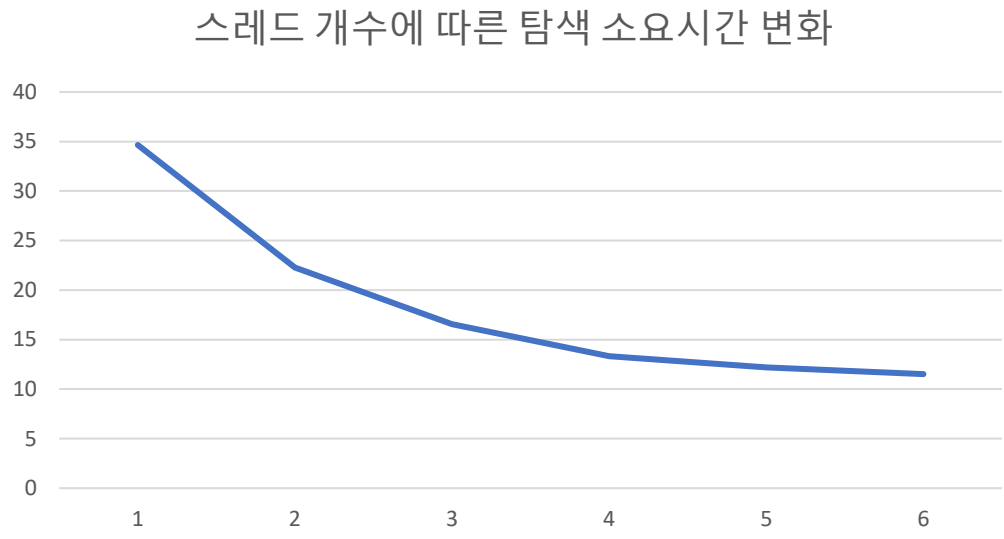
SW Occlusion Culling On
렌더링 되는 오브젝트 : 7480개

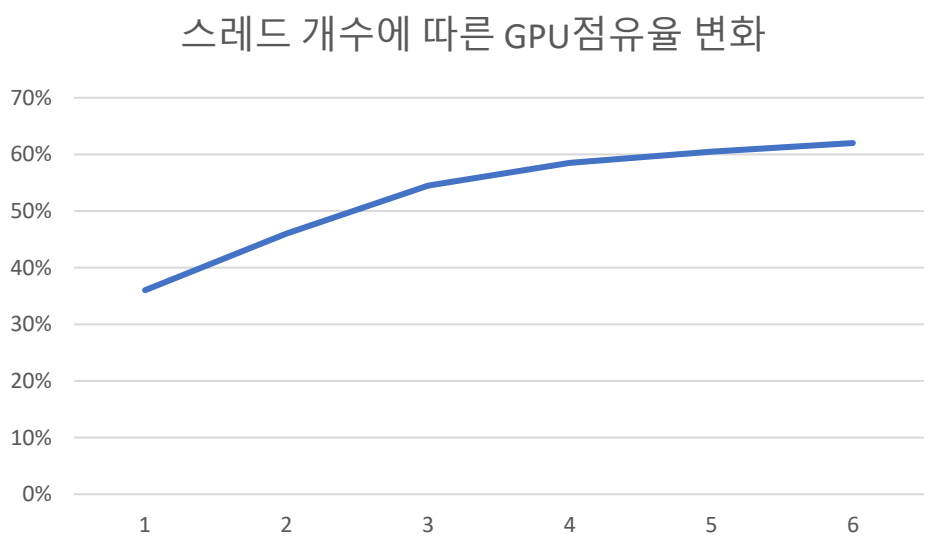


성능 테스트

i7 8700K(6 cores), 1920x1200							
threads	SW Occ(x)	1	2	3	4	5	6 CUDA Raytracing
objs	19798	7677	7660	7762	7741	7543	7678 5697
FPS	53	77.5	120.5	145	158.5	170	175 280
GPU Uasge	37%	36%	46%	54.50%	58.50%	60.50%	62% 94%
elapsed time(ms)		34.65	22.26	16.56	13.33	12.21	11.53


스레드 개수에 따른 프레임 레이트 변화
200
180
160
140
120
100
80
60
40
20
0
1
2
3
4
5
6


스레드 개수에 따른 탐색 소요시간 변화
40
35
30
25
20
15
10
5
0
1
2
3
4
5
6


스레드 개수에 따른 GPU점유율 변화
70%
60%
50%
40%
30%
20%
10%
0%
1
2
3
4
5
6

성능과 스레드 개수의 관계가 정비례하지 않는 이유

1. 코어개수가 많이 활성화될수록 코어당 클럭이 낮아진다.
2. 절두체 경계면에서 중복탐색이 많아진다.
3. 메인스레드는 탐색 스레드들이 작업을 끝내기를 기다린다.
   스레드 개수가 많아지면 대기 시간이 길어진다.

Tip

- 가로/세로 균등 분할보다 가로 분할이 더 효과적이다.
  - 일반적으로 지평선 위쪽으로 하늘(아무것도 없다)이기 때문. 이 경우 윗쪽에 배치된 스레드는 하는 일이 없고 아랫쪽에 배치된 스레드에 대부분의 작업이 몰린다.
- 타겟이 될 CPU의 성능을 명확히 알 수 없다.
  최악의 경우 SW Occlusion Culling으로 전체 성능을 크게 저하시킬 수 있다. 따라서 트리 탐색 중 기준 시간을 초과하면(ex:16ms) 즉시 SW Occlusion Culling을 끈다.
- 코어 개수가 충분히 많은 경우 z-buffer해상도를 높인다. 이것으로 렌더링 오차를 줄일 수 있다.
- 카메라의 위치/방향이 미세하게 바뀐 경우 이전 프레임의 탐색 정보를 그대로 사용해도 된다.

결론

- 멀티스레드 SW Occlusion Culling은 늘어나는 코어 개수에 대응할 수 있다. 이 점이 가장 큰 장점이다.
- CUDA Raytracing이 제일 빠르다.
- ‘할 수 있다면’ CUDA Raytracing을 Compute Shader로 구현하는 쪽이 최선의 방법이다.
- CPU만으로 작동하기 때문에 여전히 무난하고 안전한 방법이다.