Voxel기반 네트워크 게임 최적화기법

유영천

Blog: megayuchi.com

Tw :@dgtman

Voxel Horizon

- 개인 프로젝트
- 시작은 msx용 warroid(https://youtu.be/LyFoLY2m6Ic?t=5m56s)
- 지형 뒤에 숨는건 물론이고 지형을 뽀개면서 전투를 하자.
- 유도탄이 막 날라다니고(더티페어), 로켓이 터지면 그 자리에 구덩이가 패였으면 좋겠다. 강력한 무기를 사용하면 벽이 막 뚫리고...
- 디자이너가 없으니 플레이어들이 직접 맵을 만들 수 있게 하자.
  - 복셀 편집 기능 필요
- 전투만 할 순 없지. 당연히 마을에 모여서 놀기도 하고...
- 레벨업 할 수 있는 필드도 있어야 하고..
- 그렇다면 서버/클라이언트 방식이어야 한다.

기술적 목표

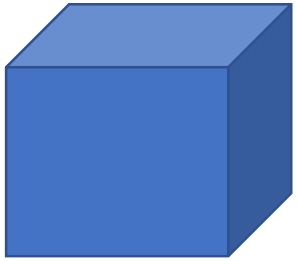
- 50cm x 50cm x 50cm 정도로 비교적 정밀한 편집이 가능할것.
  - 미술품 모델링, 폭탄 구덩이, 총알 구멍 등을 위해서는 보다 정밀할 필요가 있다.
- 복셀구조의 변형이 네트워크로 동기화 되어야함.
- 권장사양에서 120FPS로 렌더링 가능할것.
- 서버당 4000명 처리.
- 실시간으로 변경 가능한 라이팅.

https://youtu.be/2VJvHchHdNw
https://youtu.be/nCLN_pW__K0

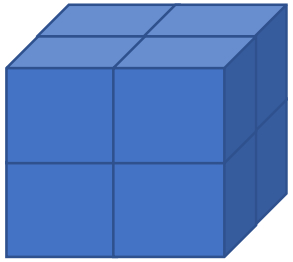
복셀 자료구조

복셀 자료구조

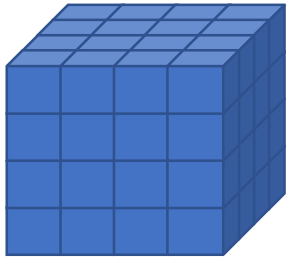
- 오브젝트 한개당 1x1x1 – 8x8x8까지 가변 정밀도
- 오브젝트 한개의 크기는 4m x 4m x 4m
- 기하구조 데이터와 텍스쳐 인덱스 데이터는 분리함.
  - (Vertex정보에 텍스쳐 인덱스 데이터 포함하지 않음)
- 기하구조는 복셀당 1 Bit, 텍스처 인덱스는 8 Bits



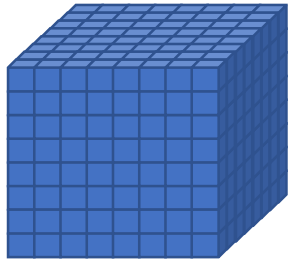
1x1x1 -> 1 bit -> 1 Byte



2x2x2 -> 8 bits -> 1 Bytes



4x4x4 -> 64 bits -> 8 Bytes



8x8x8 -> 512 bits -> 64 Bytes

복셀 자료구조 세부구현

- 기하구조(복셀 비트테이블) ,텍스쳐 인덱스 테이블 in System Memory
- 기하구조 -> Vertex Buffer
- 기하구조 -> 충돌처리용 삼각형리스트(system memory)
- 기하구조 -> 라이트맵 계산을 위한 사각형 패치 리스트(system memory)
- 텍스쳐 인덱스 테이블 -> Constant Buffer

기하구조 -> Vertex Buffer

- 복셀당 vertex 8개 필요.
- 초기에 Geometry Shader를 사용했지만 훨씬 느림.

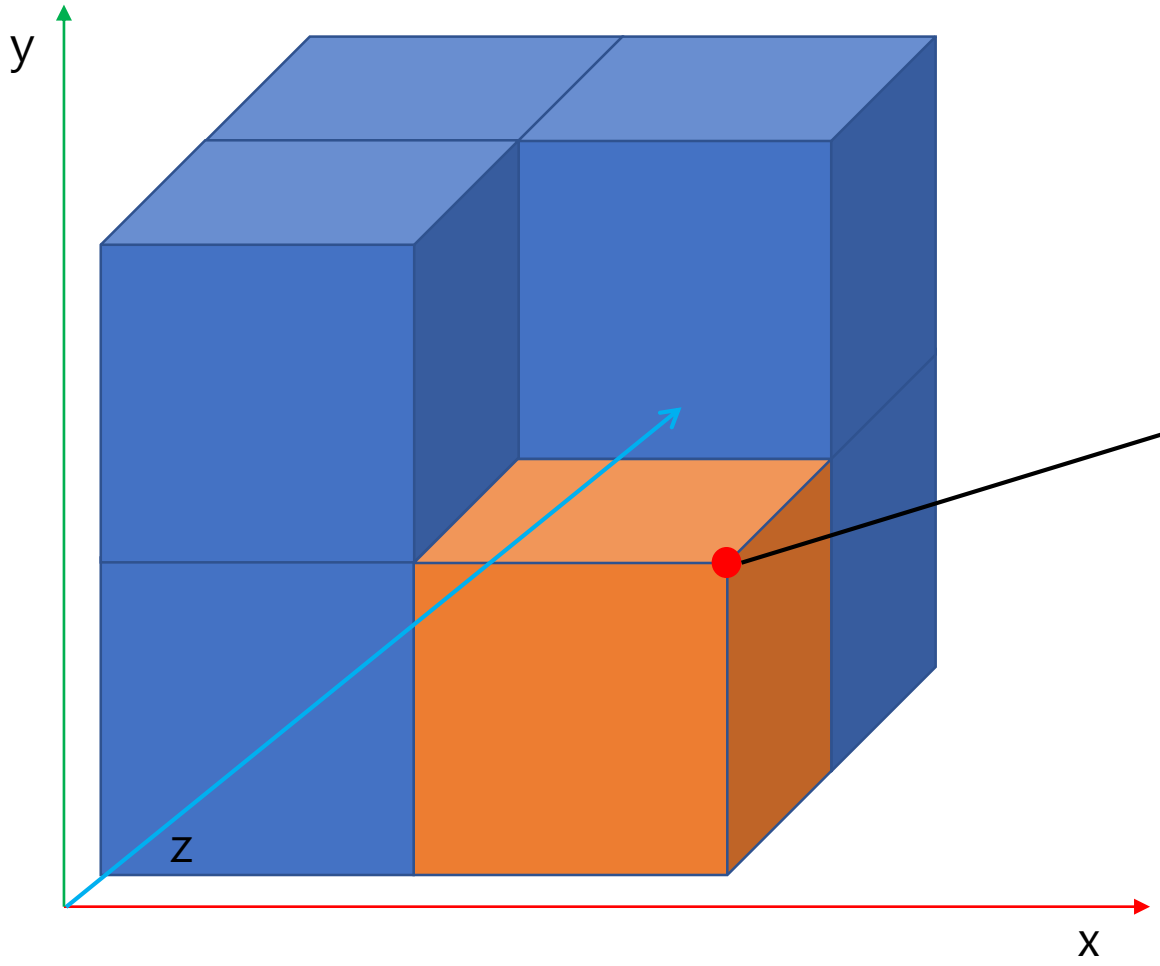
Vertex 성분

- 필요한 성분
  - 위치 좌표(4*3 = 12 bytes)
  - 법선벡터(4*3 = 12 bytes)
  - 라이트맵 텍스쳐 좌표 (4*2 = 8bytes)
  - 32 bytes -> 너무 크다.

Vertex 압축

- 점하나당 4 Bytes(32 bits)
- QuadIndex - 오브젝트 안에서의 사각형 인덱스(0 ~ 1023) , 4 bits
- oPos - 오브젝트 안에서의 좌표 (x,y,z = 0 ~7) ,3*3= 9 bits
- vPos – 복셀 안에서의 좌표 (x,y,z = 0 ~ 1), 2*3 = 6 bits
- qPos - 사각형 안에서의 좌표 (u,v = 0 ~1) = 2*2 = 4 bits
- N - 법선벡터 8방향 (0 ~ 7)= 3 bits

압축된 Vertex구조



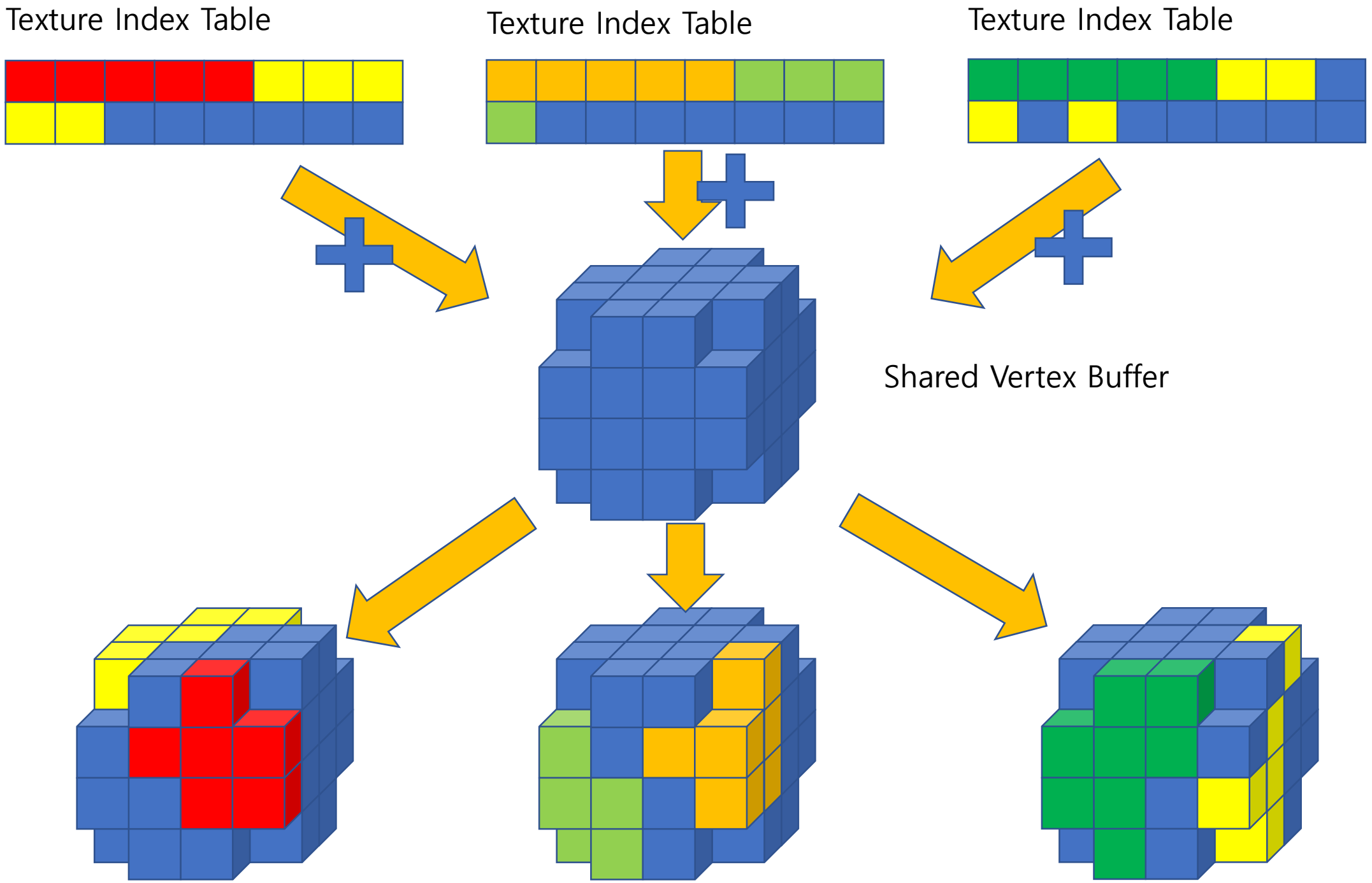
vPos : (1,1,0)
oPos : (1,0,0)
qPos : (1,1)
N : 5 (-Z)
QuadIndex : 1

vPos : (1,1,0)
oPos : (1,0,0)
qPos : (1,0)
N : 2 (+Y)
QuadIndex : 4

vPos : (1,1,0)
oPos : (1,0,0)
qPos : (0,1)
N : 0 (+X)
QuadIndex : 7

버텍스 버퍼 Sharing

- 8x8x8짜리 복셀 조합은 어마어마하게 많은 경우의 수를 만들어 낸다.
- 하지만 월드를 구성해보면 중복되는 패턴도 많이 나온다.
- 컬러(텍스처) 조합을 고려하면 훨씬 많은 조합이 있지만 이미 텍스처 인덱스 테이블을 분리했다!
- 따라서 8x8x8 복셀 조합은 중복되는 패턴들에 대해서 한번 만들어놓은 VertexBuffer는 그대로 재활용 가능.
- 상당한 GPU메모리 절약 효과가 있다.


Texture Index Table
Texture Index Table
Texture Index Table
Shared Vertex Buffer

충돌처리

복셀오브젝트에 대한 충돌처리

- 캐릭터가 복셀 지형지물 위에서 움직이기 위해 충돌처리를 한다.
- 또한 로켓탄을 쏴서 복셀을 파괴하거나 혹은 미끄러져 방향이 바뀌거나 하는 경우도 같은 코드를 사용한다.
- 3가지 액션이 있다. 충돌후 멈춤, 충돌후 미끄러짐, 충돌후 반사
- 위치/단축/장축을 가진 타원체와 속도와 방향을 나타내는 벡터로 충돌테스트를 할 삼각형을 수집한다.

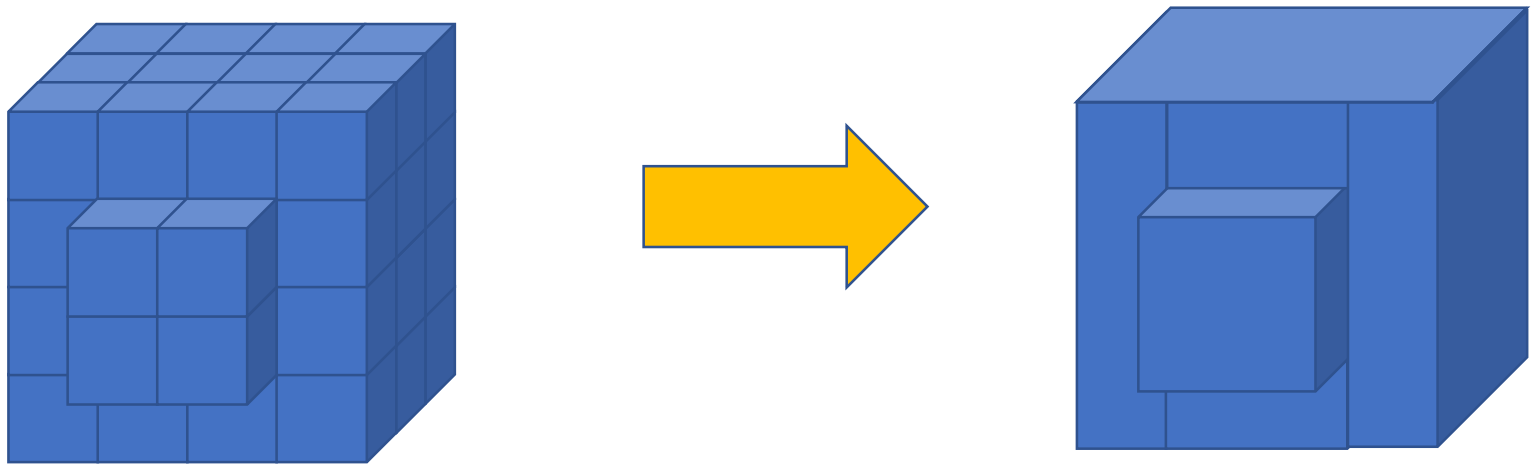
복셀오브젝트에 대한 충돌처리

- 월드에서 KD-Tree를 순회하여 타원체/ray에 대해 교차하는 오브젝트들을 선택.
- 각 오브젝트들에 대해 충돌처리 삼각형을 긁어온다.
- 삼각형들에 대해서 타원체를 충돌시키고 미끄러짐 벡터로 다음 위치를 조정. 속도벡터가 거의 0이 될때까지 반복.
- 서버와 클라이언트가 공유하는 코드의 대표적인 사례.

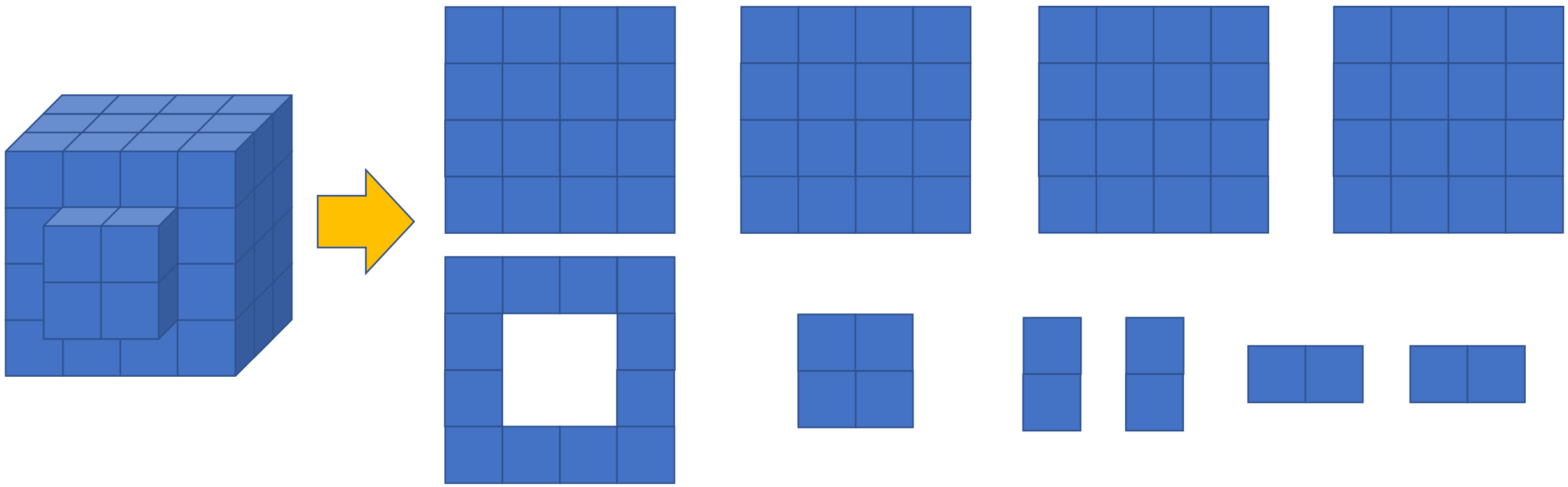
복셀의 기하구조 -> 충돌처리용 삼각형 매시

- 인접한 복셀을 병합해서 면을 줄일 수 있다.
- 인접한 복셀이 서로 다른 텍스쳐 인덱스를 가지는 경우 렌더링을 위해서는 병합해서는 안된다.
- 하지만 렌더링 용도가 아닌 경우, 기하구조의 충돌처리를 위해서라면 병합할 수 있다.
- 오히려 성능을 위해서는 병합해야한다.
- 효율적인 충돌처리와 picking을 위해서 최대한 복셀을 병합하여 시스템 메모리 상에 유지한다.

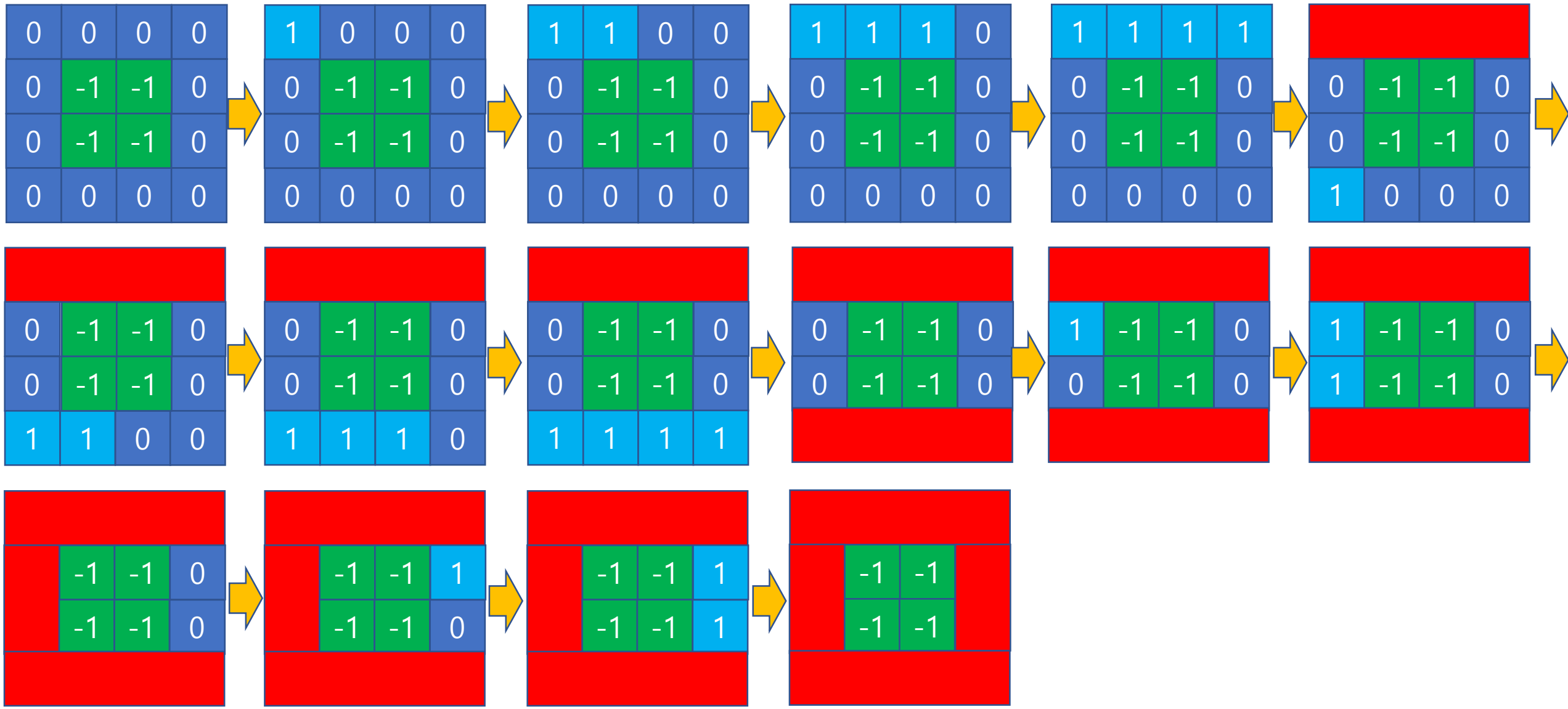
충돌처리를 위해 인접한 면들을 병합하여 최적화된 매시로 변환.



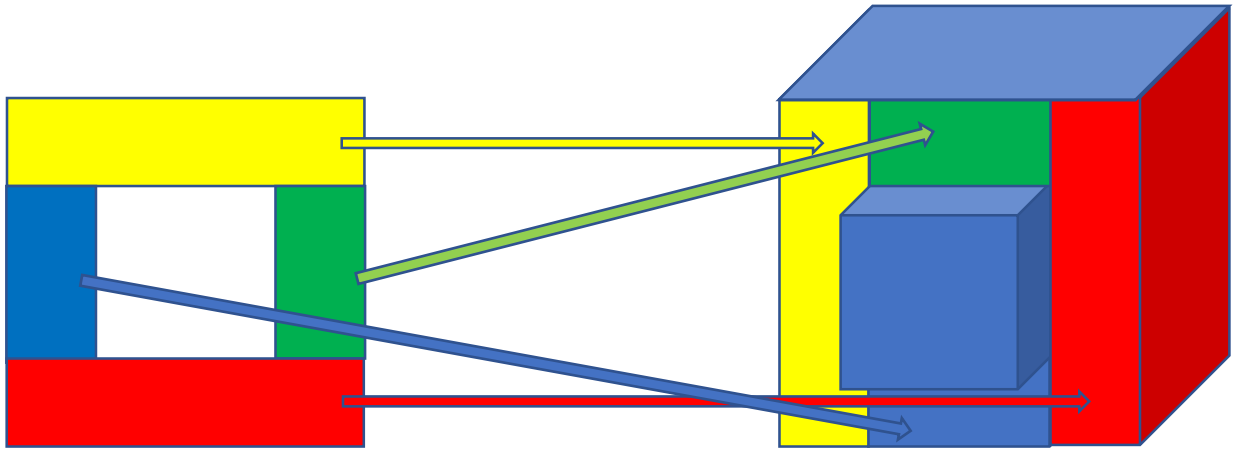
사각형들을 분해해서 같은 평면 방정식에 따라 그룹화



2D 비트맵상에서 x,y축으로 한칸씩 성장



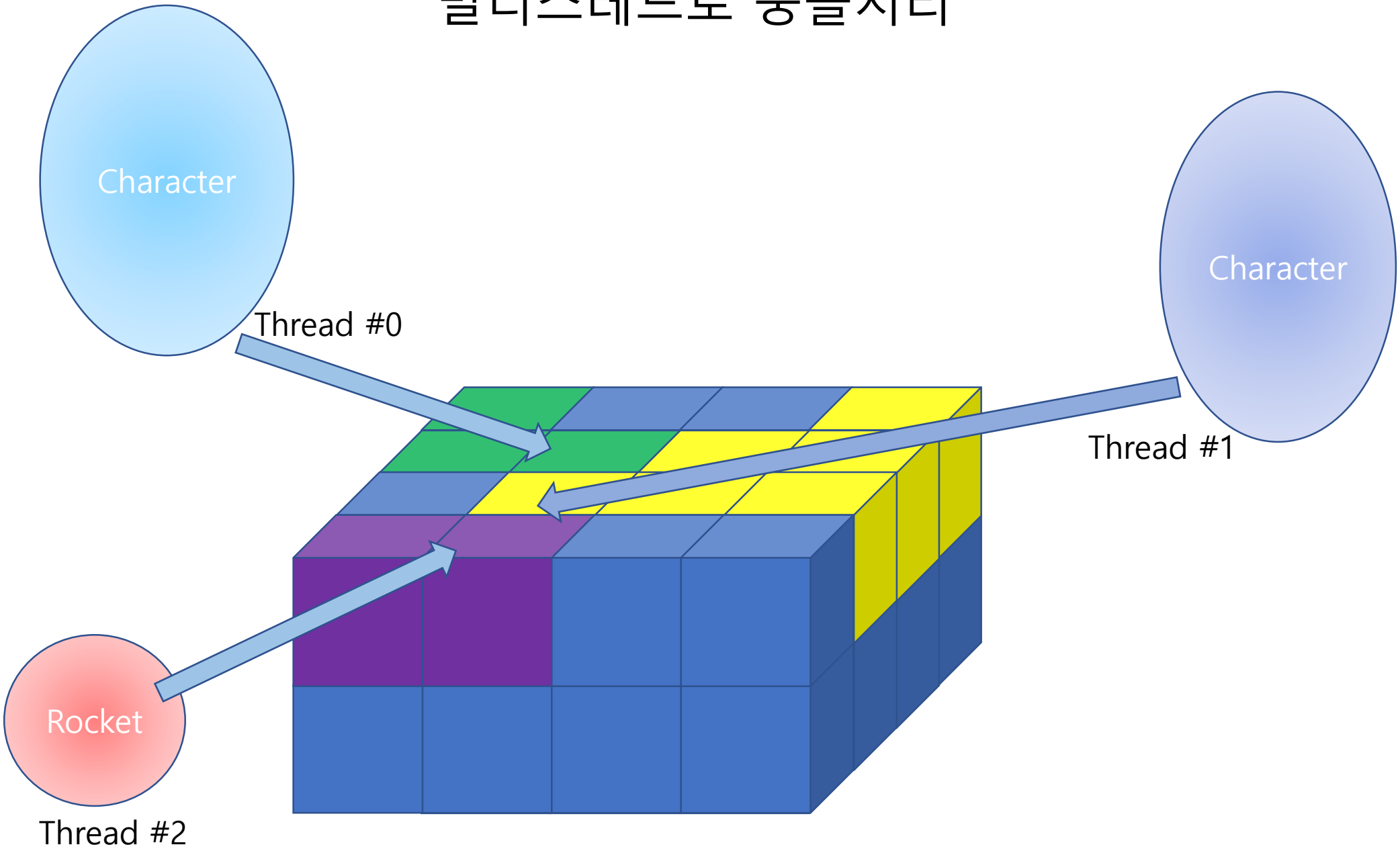
오브젝트의 3D위치, 분해된 사각형 그룹의 면방정식이 있으므로 2D 비트맵을 다시 3D 사각형 리스트로 변환할 수 있다.



멀티스레드로 충돌처리

- 서버와 클라이언트 모두 충돌처리를 멀티스레드로 처리하므로 완전히 충돌처리 코드와 KD-Tree코드는 thread safe하게 작성한다.
- Tree를 순회할때 필요한 Stack만 스레드별로 제공되면 thread safe하게 작성하는건 쉽다.
- 클라이언트와 서버 모두 멀티스레드를 사용하지만 서버에서 특히 유용하다.

멀티스레드로 충돌처리



Culling

Culling

- View Frustum Culling – (가장 기초적인 방법)
- BSP/PVS – Quake계열
- BSP/Room/Portal – 초기 Unreal 등
- Occlusion Culling
  - SW Occlusion Culling
  - HW Occlusion Culling with D3DQuery
  - Hierarchical Occlusion Culling with Compute Shader

Culling

- View Frustum Culling – (가장 기초적인 방법)
- BSP/PVS – Quake계열
- BSP/Room/Portal – 초기 Unreal 등
- Occlusion Culling
  - SW Occlusion Culling
  - HW Occlusion Culling with D3DQuery
  - Hierarchical Occlusion Culling with Compute Shader

Voxel 프로젝트에선 지형지물이 실시간으로 바뀌기 때문에 사용할 수 없음.

Voxel 프로젝트에선 오브젝트가 너무 많고 따라서 Draw call 또한 너무 많아 느려서 이 방식은 사용하지 않음.

Culling in Voxel Horizon

- View Frustum Culling
- Occlusion Culling
  - SW Occlusion Culling
  - Hierarchical Occlusion Culling with Compute Shader
- 카메라 방향에 상관없는 Culling
  - 가려진 복셀 Culling
  - 가려진 복셀 오브젝트 Culling

SW Occlusion Culling

- CPU 코드로 구현한 z-buffer
- Throughput은 HW방식에 비해 크~~~게 떨어진다. 대신 렌더링을 위한 셋업과정이 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?

Occlusion Culling in KD-Tree

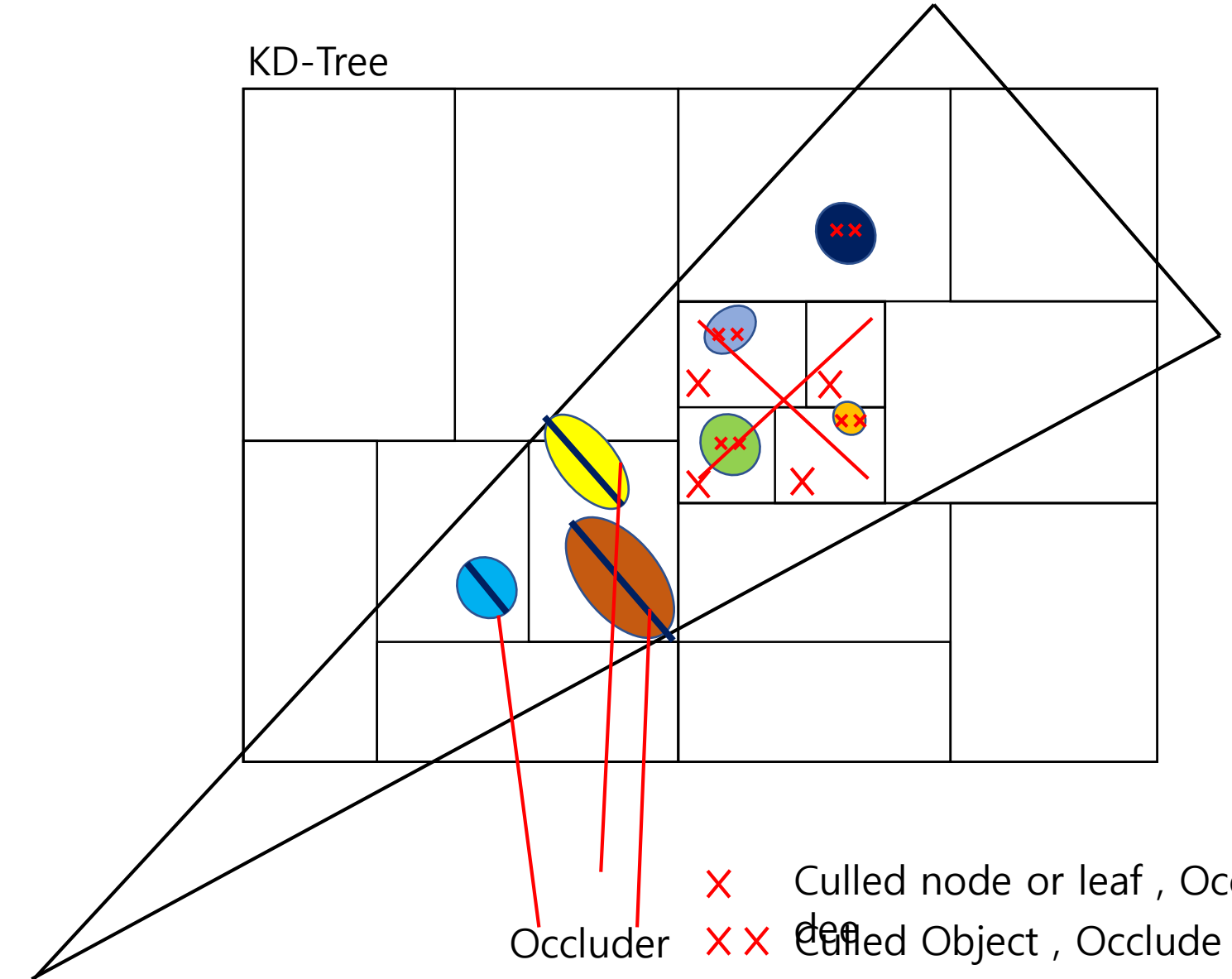
- node째로 culling하면 하위 node와 leaf에 포함된 오브젝트들도 같이 cullin된다.
- KD-Tree탐색 단계에서 node에 대해서 Occlusion Culling을 수행하면 렌더링될 오브젝트들을 원천적으로 줄일 수 있다.
- 가까운 node에서 먼 node로 탐색해가면서 가까운 node(leaf)의 오브젝트를 z-buffe에 test& rasterize 한다.
- KD-Tree를 사용하면 가까운 node와 먼 node를 구분해서 가까운 node부터 탐색할 수 있다.(KD-Tree Traversal)
  - 가까운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

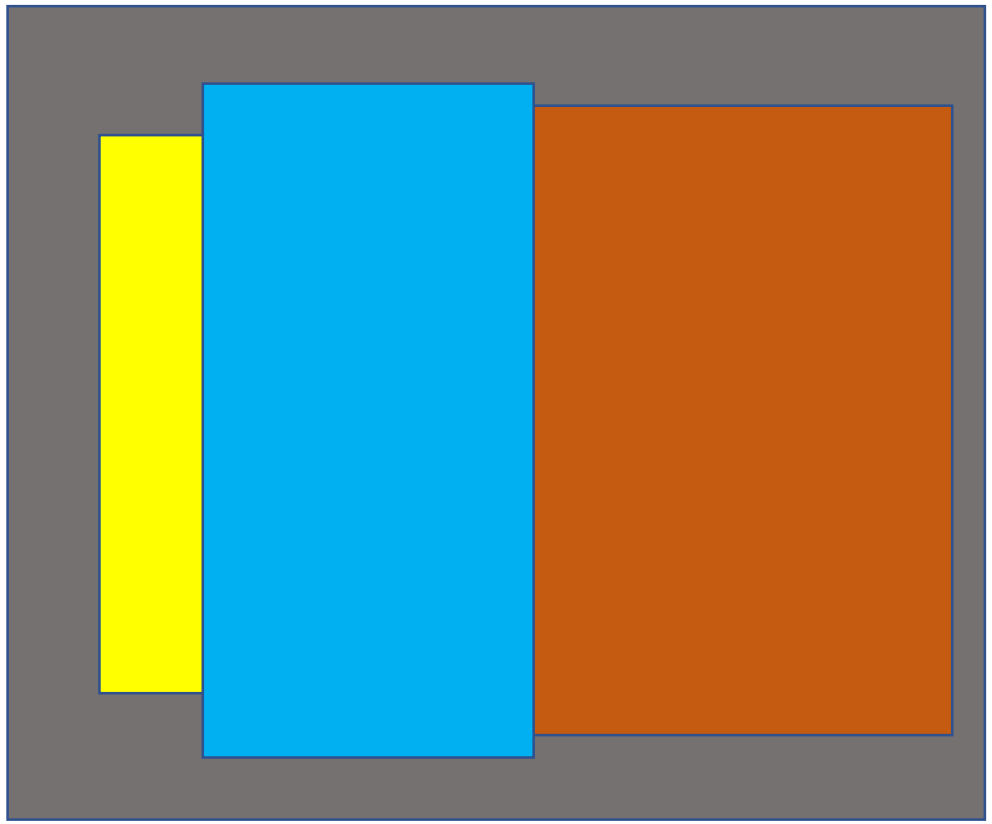
KD-Tree + SW Occlusion Culling을 이용한 오브젝트 수집

- 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-test & z-write수행. 렌더링되는 픽셀이 하나도 없으면 수집하지 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 + 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.


Voxel Horizon Client x64 Release D3D11 - D3D_FEATURE_LEVEL_11_1 - Oct 4 2017
f:950 obj:230 hfo:0 spr:3 font:10 P:5863 V:4663 W:0
Tex:41 VB:343 IB:324 CB:21 VL:7 A.Map:0 font:7
DEV. Version Build 1
Ping:T_T
f:950 obj:230 hfo:0
Tex:41 VB:343 IB:3
SuperYuchi
S/W Z Rasterizer
UI initialized sucessfully.
Trying load voxels from (C:₩Users₩megay₩AppData₩Local₩VoxelHorizon₩v101.vxl).

SW Z-Buffer Rasterizer요구사항

- 기본적으로 90년대 게임들의 SW Rasterizer와 크게 다르지 않다.
- Texturing은 필요하지 않음.
- 임의의 삼각형 리스트를 입력 받아 버퍼에 z값을 기록한다. GPU의 z-buffe와 기본적으로 같다.
- 화면의 left,right,top,bottom,near, far에 대해 클리핑 기능이 필요.
- Z-write & z-test, z-test only 두가지 함수를 노출한다.

SW Z-Buffer Rasterizer (Phase 1 - Transform)

1. 월드공간의 삼각형 -> N-Polygon으로 변환(클리핑시 억만조각 나는걸 막기 위해)
2. N-Polygon을 view 공간으로 변환
3. view 공간으로 변환된 N-Polygon을 near, far 평면에 대해서 절단. 여기서부터 삼각형이 아니게 된다.
4. view공간의 N-Polygon을 projection공간으로 변환 후 w로 나눠서 $-1 < x < 1$, $-1 < y < 1$, $0 < z < 1$ 좌표계로 변환.

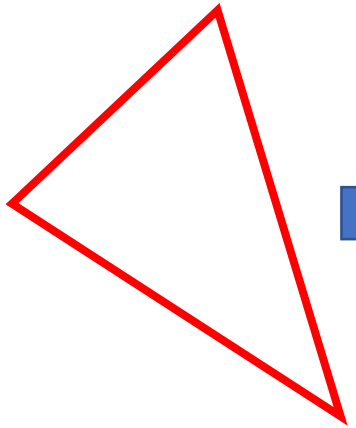
SW Z-Buffer Rasterizer (Phase 1 - Transform)

5. 정규좌표로 변환된 N-Polygon을 left,right,top,bottom 4개의 평면으로 절단. 이 과정을 거치면 3각형 -> 최대 8각형까지 증가.
6. N-Polygon -> N-2개의 삼각형으로 분리.
7. 각 삼각형을 y좌표가 같은 2개의 삼각형으로 분리.

SW Z-Buffer Rasterizer (Phase 2 – Rasterize)

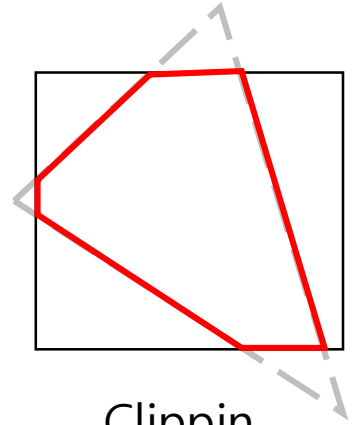
8. 삼각형의 왼쪽 오른쪽 두 변의 기울기를 구한다. 이제 y좌표가 주어질때 양쪽 변의 x0,x1좌표를 얻을 수 있다.
9. 삼각형의 윗쪽 꼭지점으로부터 한 라인씩 내려오며 x0부터 x1까지 스캔라인을 따라 z값을 보간해서 얻어온다.
10. 보간해서 얻은 z값을 스크린 버퍼로부터 얻어온 z값과 비교, 보간한 z값이 앞에 있을 경우 덮어 쓴다.

Near/far에 대해 한번, left/top/right.bottom에 한번 총 두번의 클리핑을 나눠서 하는 이유는 w값이 1이 아닌 상태에서의 클리핑이 까다롭기 때문이다. W <= 0 일 경우, 그러니까 카메라 안쪽에 넘어온 점이 있을 경우 계산 불능이 된다. 그러니까 -1 < x,y < 1의 좌표로 변환하기 전에 클리핑을 수행한다.

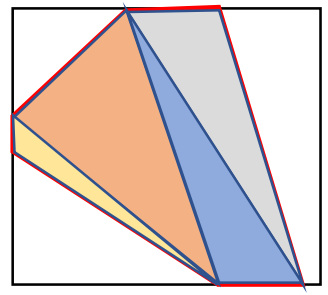


Local -> World-> Projection Matrix

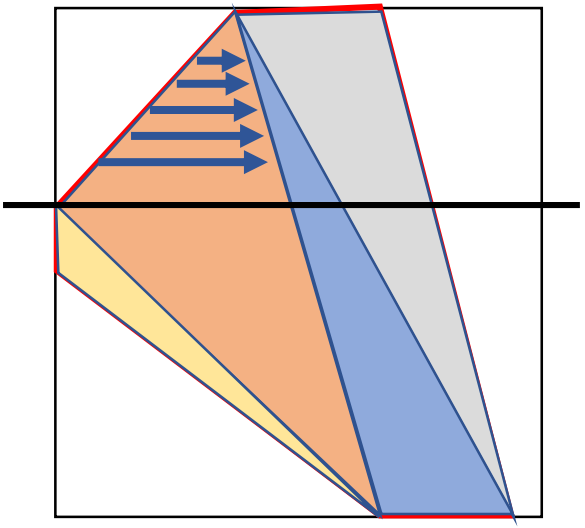
Transform with MVP Matrix

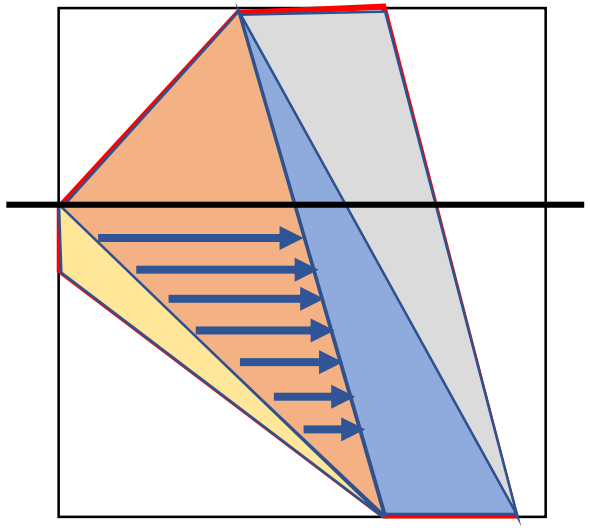


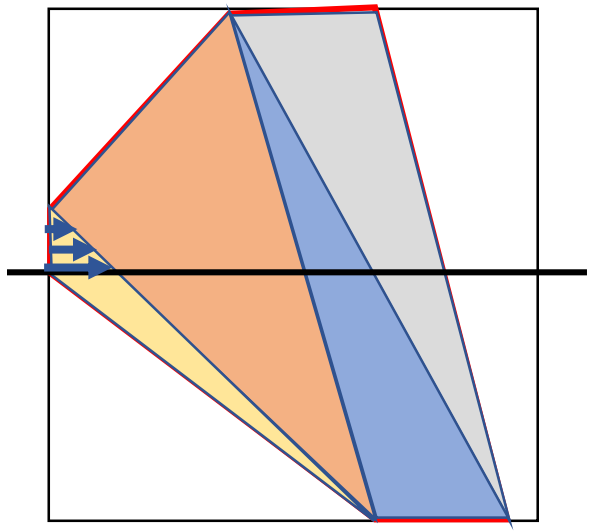
Clipping

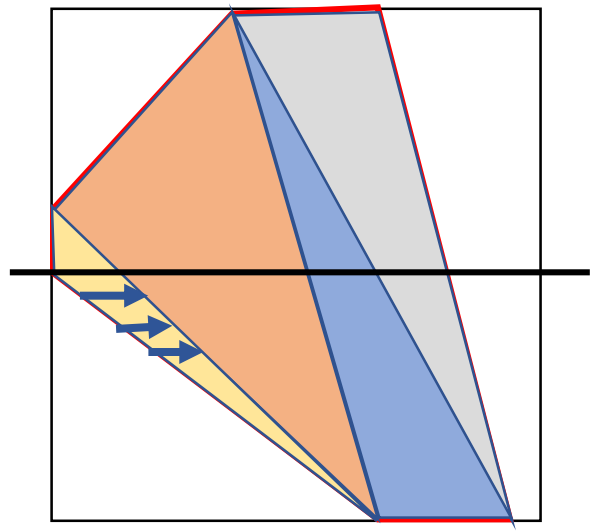


Polygon -> Triangles



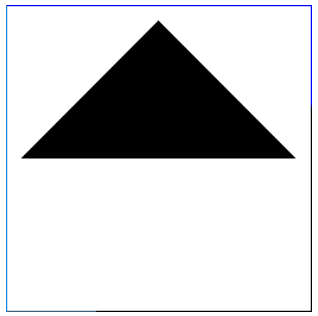




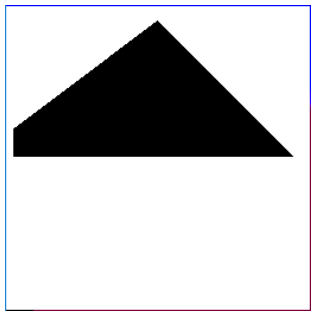


Rasterize

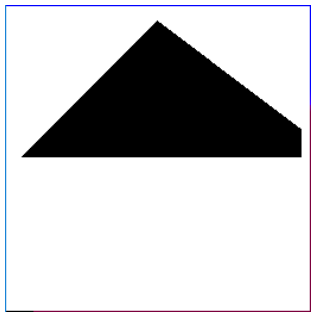
Clipping



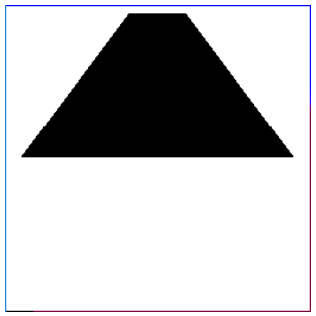
No clipping



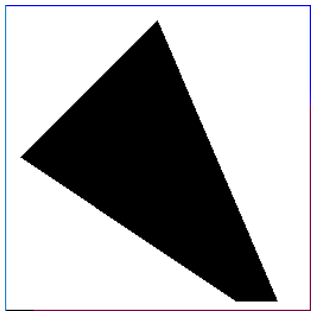
Left



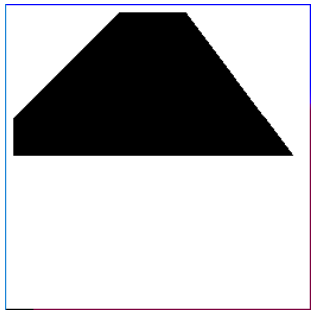
Right



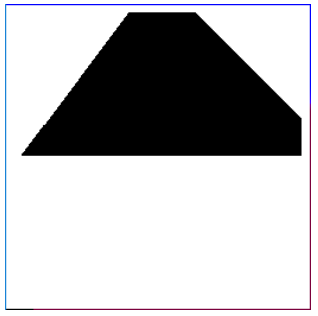
Top



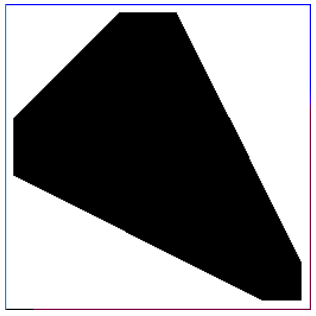
Bottom



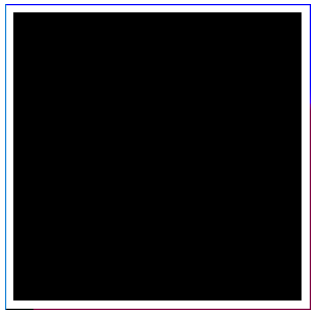
Left-top



Right-Top

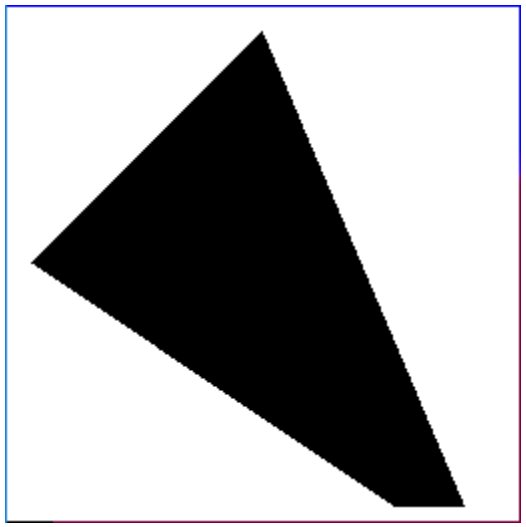


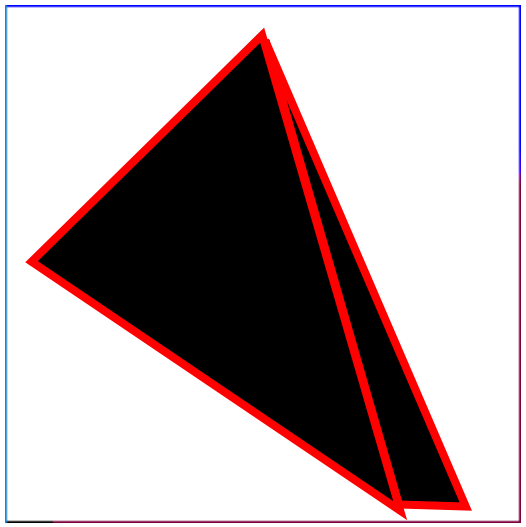
Left – Top – Right - Bottom



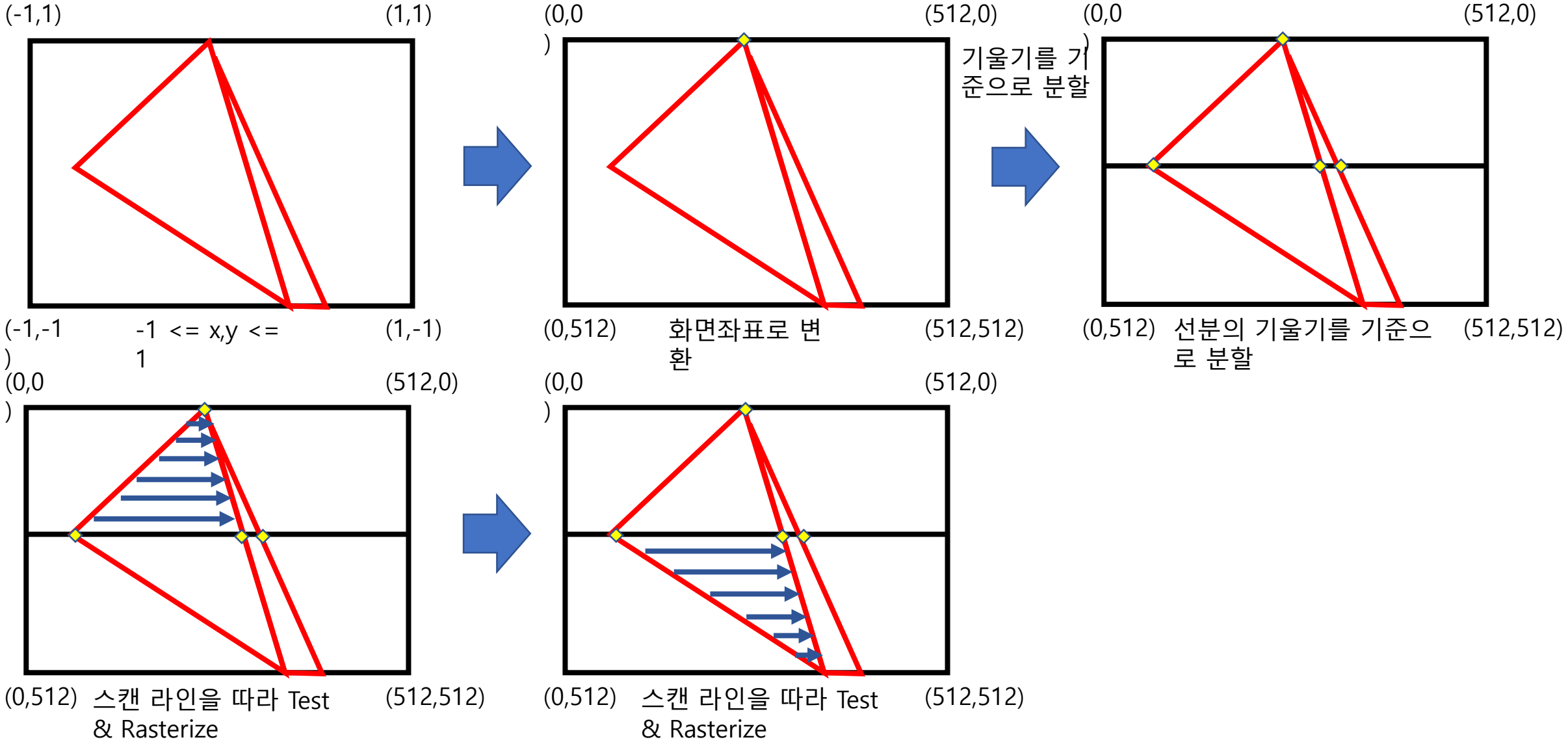
Left – Top – Right - Bottom

N Polygon -> Triangles

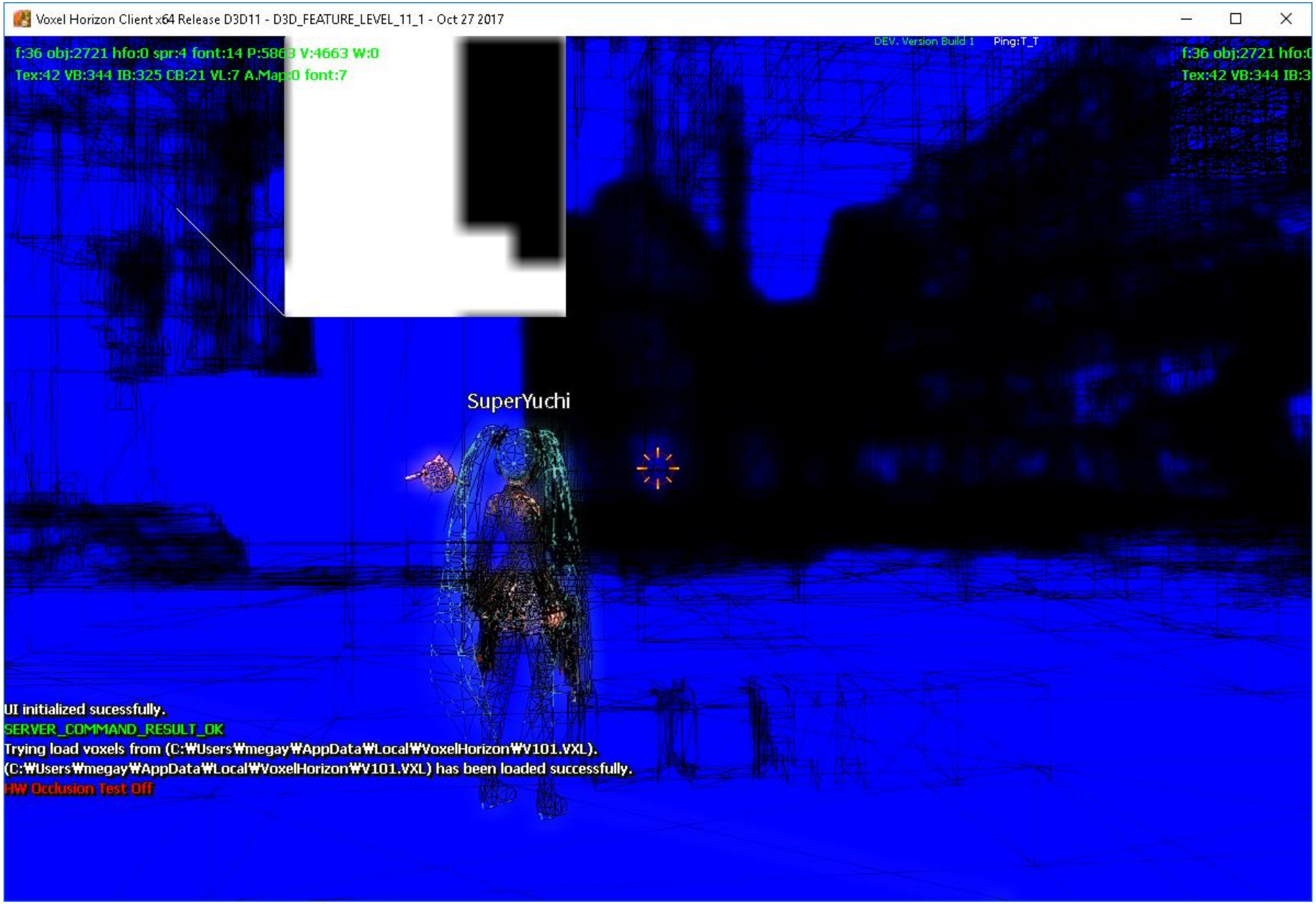




Rasterize(or test)



Only SW Occlusion Culling (rasterize 2ms, test 1ms 제한)



Hierarchical Occlusion Culling with Compute Shader

[CPU코드에서]

- KD-Tree View Frustum Culling으로 오브젝트들을 수집
- Occluder로 그릴 오브젝트들을 선별
- Occludee로 test할 오브젝트들을 선별
- Occluder 렌더링
- Z-Buffer를 1x1사이즈까지 리사이즈
- D3DQuery를 on하고 Occludee를 렌더링
- Occludee로 사용할 오브젝트의 Bounding Sphere를 CS로 전달.

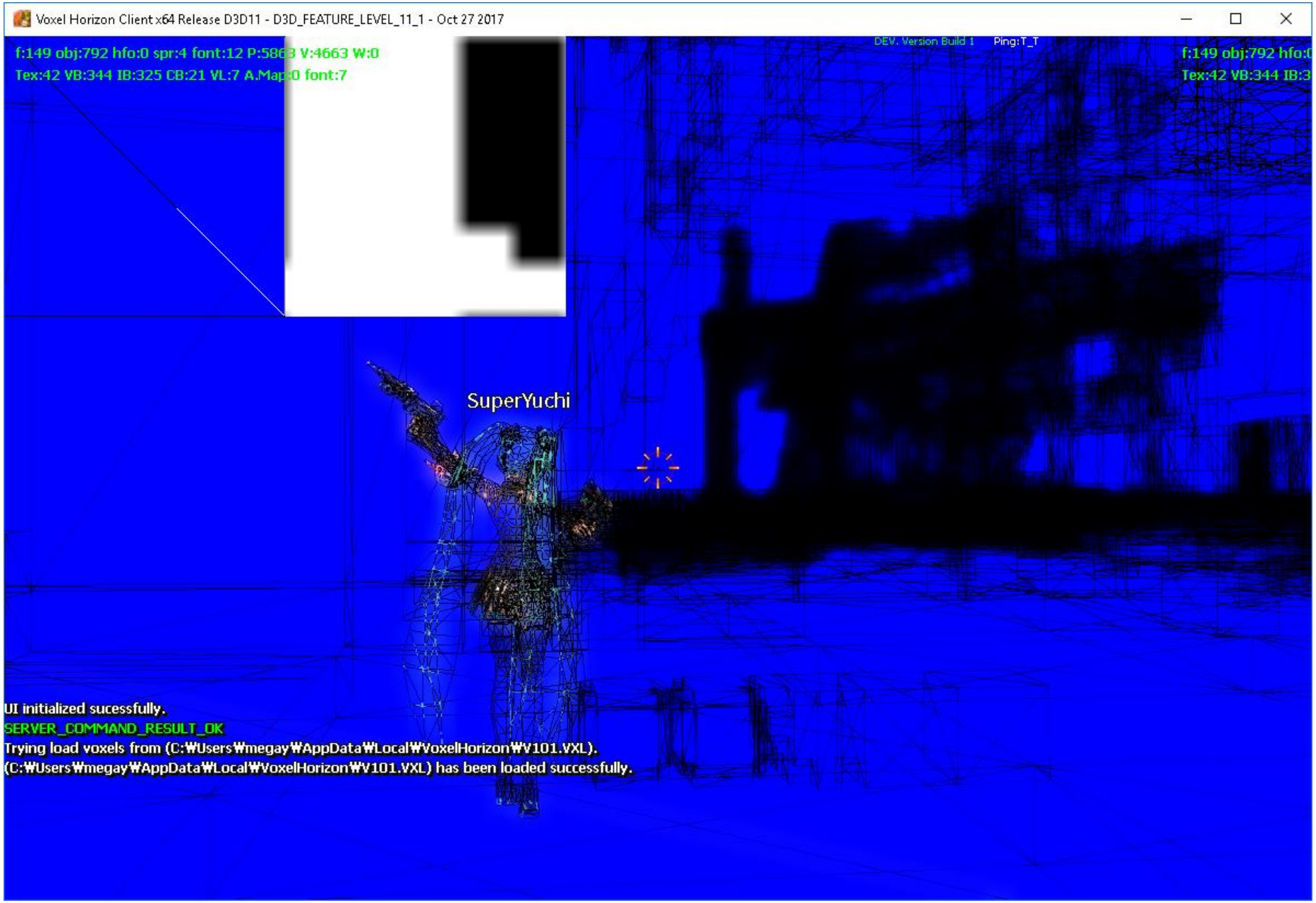
[Compute Shader에서]

- Bounding Shpere를 화면 좌표계로 변환하고 가장 1픽셀에 대응할 수 있는 zbuffer를 선택
- Z값 비교 후 결과를 SRW Buffer에 저장.

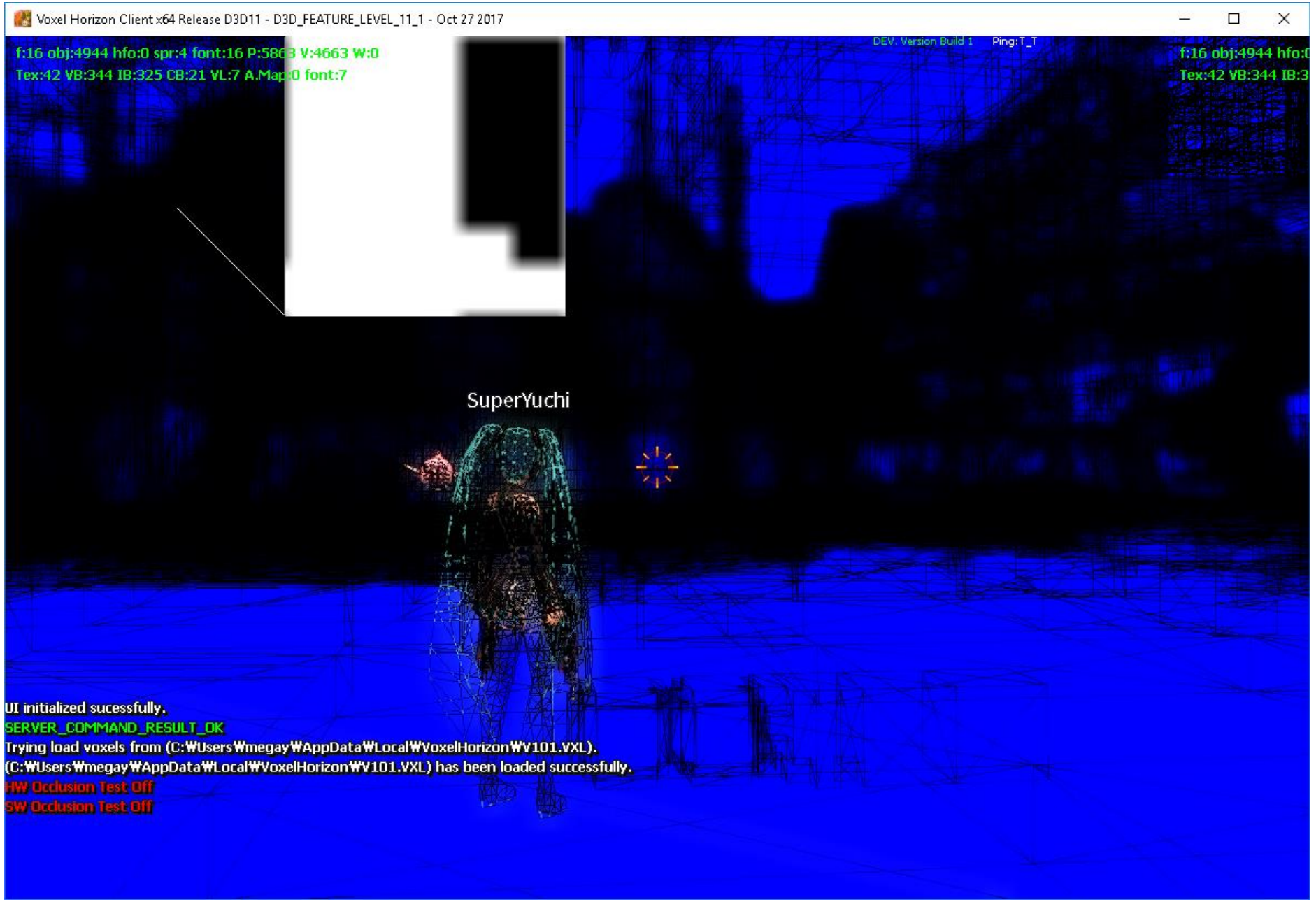
Hierarchical Occlusion Culling with Compute Shader

- Occluder를 그리는데 걸리는 시간은 D3Dquery방식과 똑같음.
- 추가적으로 z-buffer resize시간이 필요함.
- Occludee를 그리는 대신 CS를 호출. 이 부분에선 D3DQuery보다 명백히 빠름.
- Culling에 걸리는 시간 자체는 대개 D3DQuery보다 50%이상 빠름. 대신 정밀도는 떨어짐.
- https://megayuchi.com/2016/03/13/d3d12%ec%97%94%ec%a7%84%ea%b0%9c%eb%b0%9c-hierachical-z-map-occlusion-culling/

HW Occlusion Culling on / SW Occlusion Culling on (rasterize 2ms, test 1ms 제한)



HW Occlusion Culling off / SW Occlusion Culling off (rasterize 2ms, test 1ms 제한)

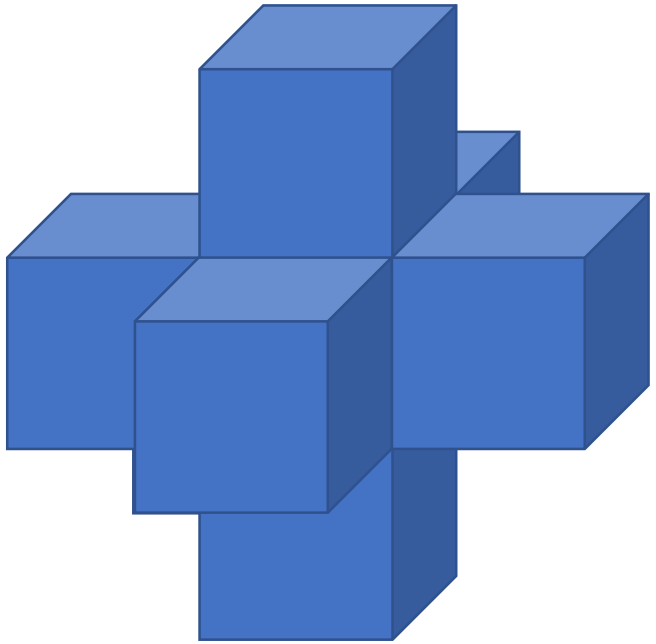


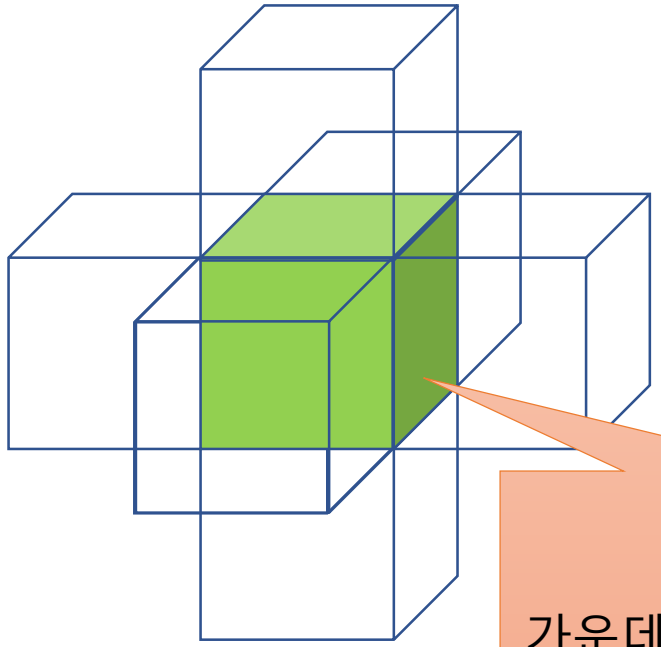
카메라 방향에 상관없는 culling

- 숨겨진 복셀 제거
- 숨겨진 복셀 오브젝트 제거

숨겨진 복셀 제거

- 오브젝트 내에서 임의의 복셀에 대하여 인접한 6면에 복셀이 존재한다면(bit가 1이라면) 이 복셀은 보이지 않는다.



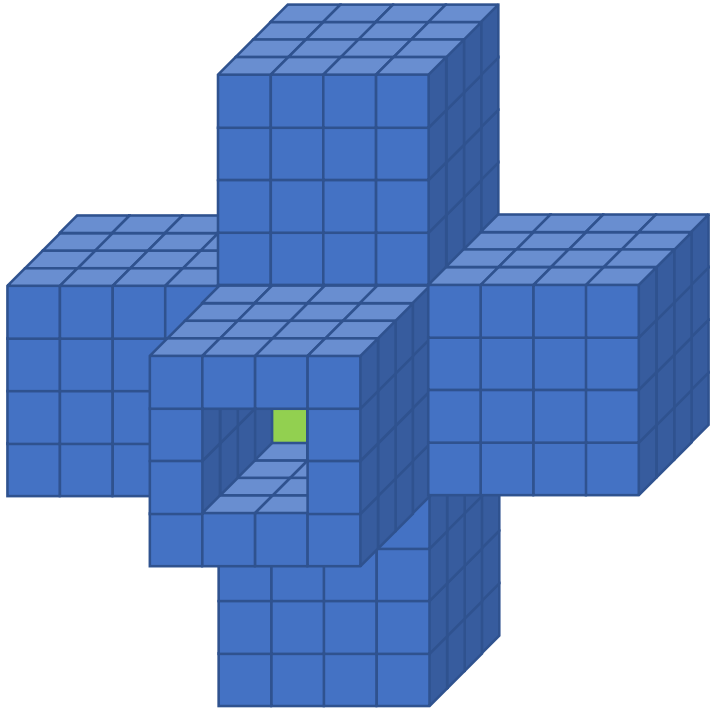


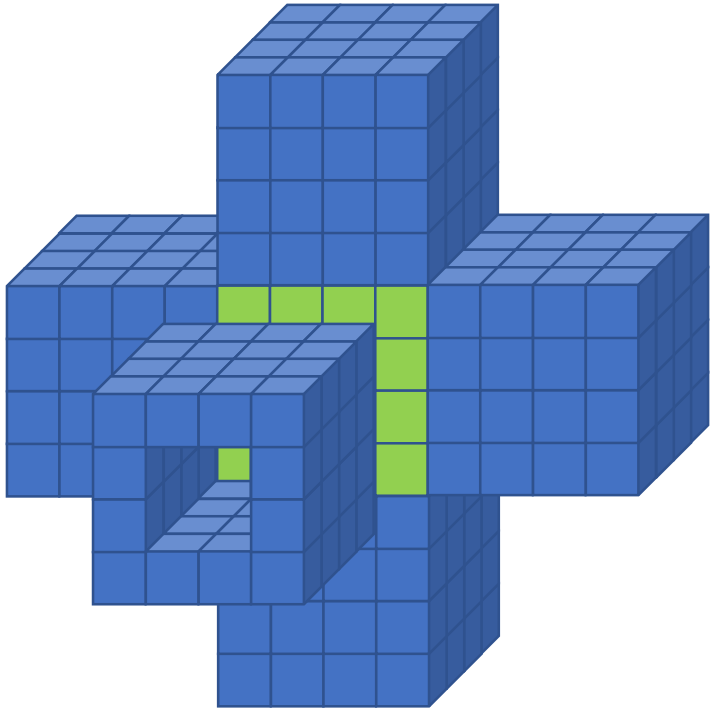
가운데 있는 복셀은 카메라 방향에 상관없이 보이지 않는다.

숨겨진 복셀 오브젝트 제거

- 카메라 방향에 대해 A복셀 오브젝트가 B복셀 오브젝트의 앞에 있고,
- A복셀 오브젝트는 카메라 방향으로 구멍이 뚫려있지 않다면,
- B복셀 오브젝트는 보이지 않는다.
- 이후로 이를 터널찾기 라고 부르자.
- 모든 오브젝트는 인접한 오브젝트의 터널상태에 따라 bits flags 값을 가진다.

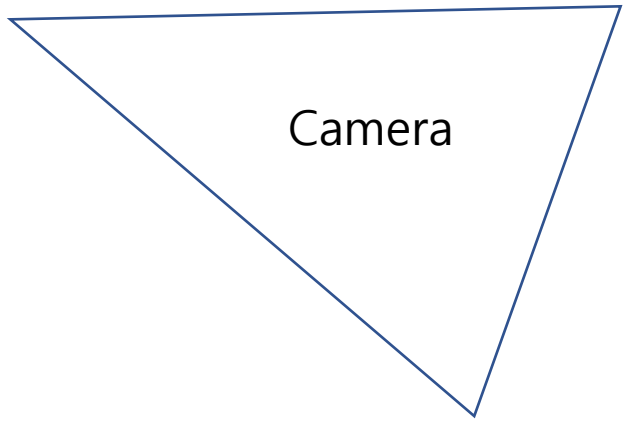
- 카메라와 상관없이 오브젝트가 보일지 안보일지를 결정할 수 있다.
- 추가적으로 카메라 ray의 x,y,z성분과 bits flags변수에 따라 보일지 보이지 않을지를 빠르게 판단할 수 있다.
- 오브젝트에 변형이 생길때마다 인접한 오브젝트들에 대해서 수행해줘야한다.

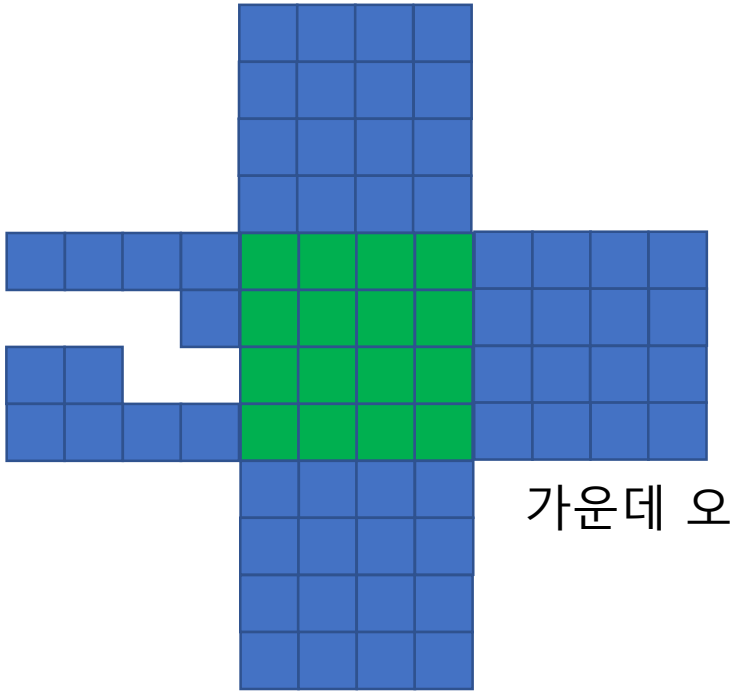




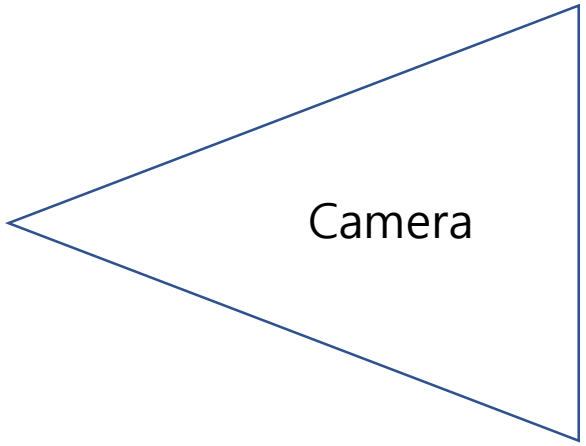
인접한 6면에 다른 브젝트들이 존재한다. 하지만 그중 하나는 구멍이 뚫려있으므로 카메라 위치에 따라 가운데 오브젝트가 보일수 있다.

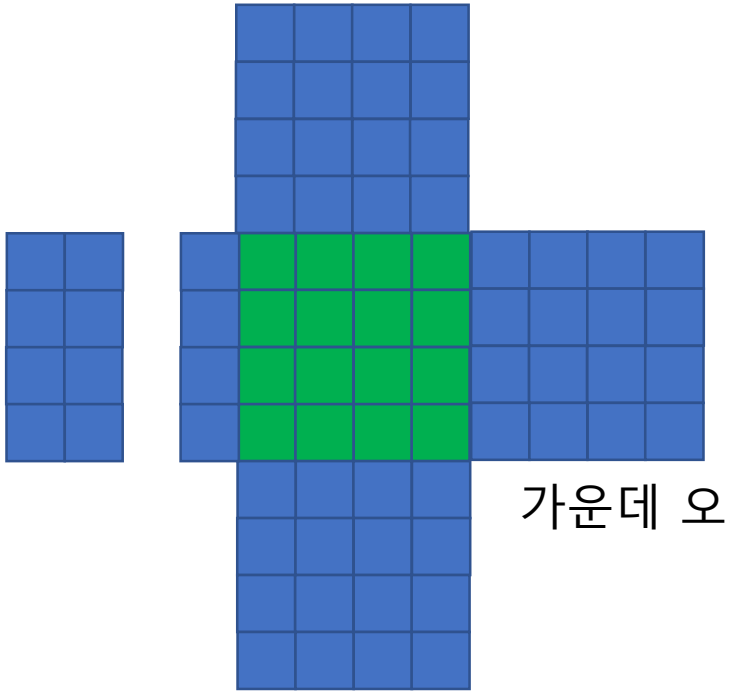
TOP View에서 본 단면





가운데 오브젝트 보임





가운데 오브젝트 보이지 않음

네트워크

World Sector Table

- 전체 월드를 특정 간격 (30m x 30m) 로 자른 그리드.
- 패킷 전송의 지역적 구분 단위가 된다.
- 이벤트가 발생한 섹터 + 주변 8섹터가 패킷 전송 범위.
- 모든 플레이어와 NPC는 위치에 따라 섹터에 저장된다.
- 플레이어나 NPC가 움직일때 즉시 섹터 테이블도 갱신된다.
- 모든 MMO게임은 비슷한 구조를 갖추고 있다.
- 더 이상 언급 안함. 너무 당연한 내용이라...

Sector Table per Player

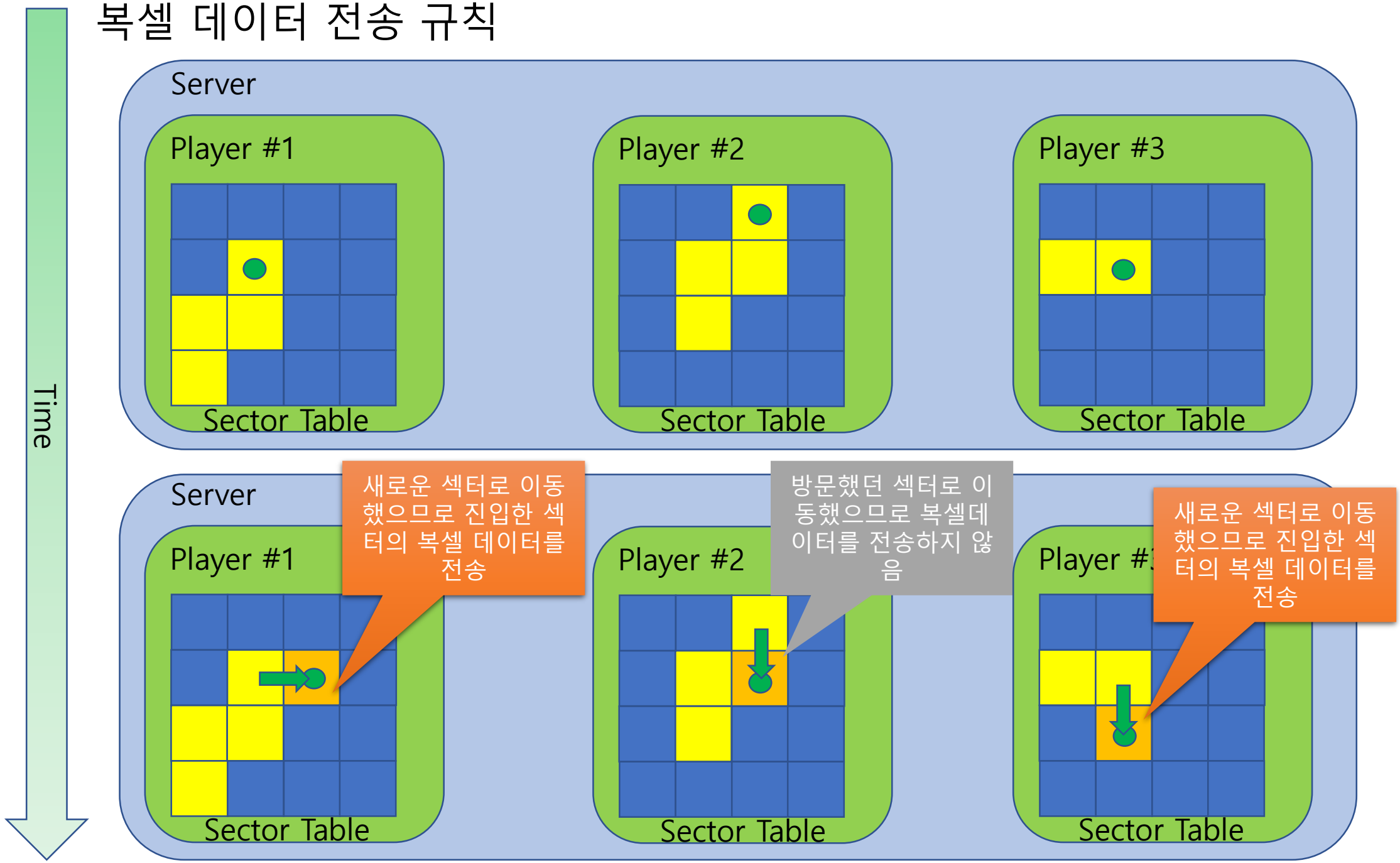
- 월드의 어느 섹터를 방문했는지 표시. 방문한적이 있으면 1, 없으면 0으로 설정한다.
- 서버측의 플레이어 인스턴스마다 갖고 있다.
- 클라이언트에서도 가지고 있다.
- 플레이어가 처음 로그인하거나 이동했을때 새로 진입한 섹터를 방문했는지 확인한다. 방문한적이 없을 경우 bit를 1로 바꾸고 해당 섹터의 복셀 지형을 플레이어에게 전송한다.
- World Sector Table과 간격은 같다.

복셀 데이터 스트리밍

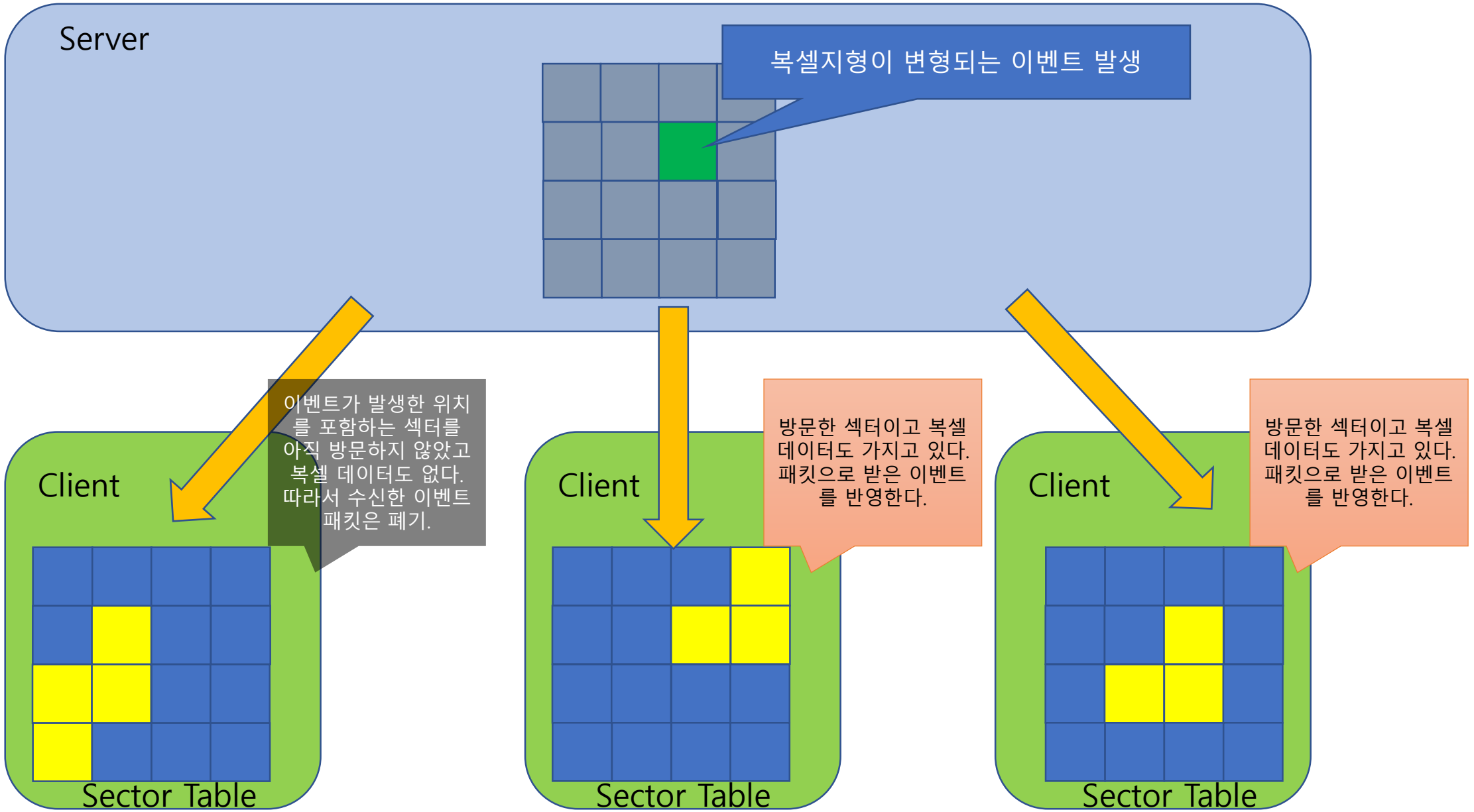
- 서버는 복셀지형이 변형되는 이벤트를 월드 전역에 브로드캐스팅한다.
- 클라이언트는 자신이 방문했던 섹터에 대한 패킷만을 처리한다.
- 클라이언트는 방문한적이 없는, 따라서 복셀지형을 가지고 있지 않은 섹터에서 발생한 복셀변형 이벤트는 그냥 폐기한다.

- 방문한 적이 없어서 해당 섹터의 복셀 데이터를 수신하면 그것이 서버의 최신 데이터이다.
- 이후로 복셀지형이 변형되는 이벤트를 수신하면 클라이언트에서 바로 반영하므로 복셀 데이터는 최신으로 유지된다.
- 방문한 적이 없는 섹터에서 발생한 복셀 변형 이벤트는 폐기한다.

복셀 데이터 전송 규칙


Time
Server
Player #1
Sector Table
Player #2
Sector Table
Player #3
Sector Table
Server
Player #1
Sector Table
새로운 섹터로 이동했으므로 진입한 섹터의 복셀 데이터를 전송
Player #2
Sector Table
방문했던 섹터로 이동했으므로 복셀데이터를 전송하지 않음
Player #
Sector Table
새로운 섹터로 이동했으므로 진입한 섹터의 복셀 데이터를 전송

복셀 데이터가 변형되는 이벤트가 발생했을 때 패킷 처리



복셀 스트리밍 데모

패킷 줄이기

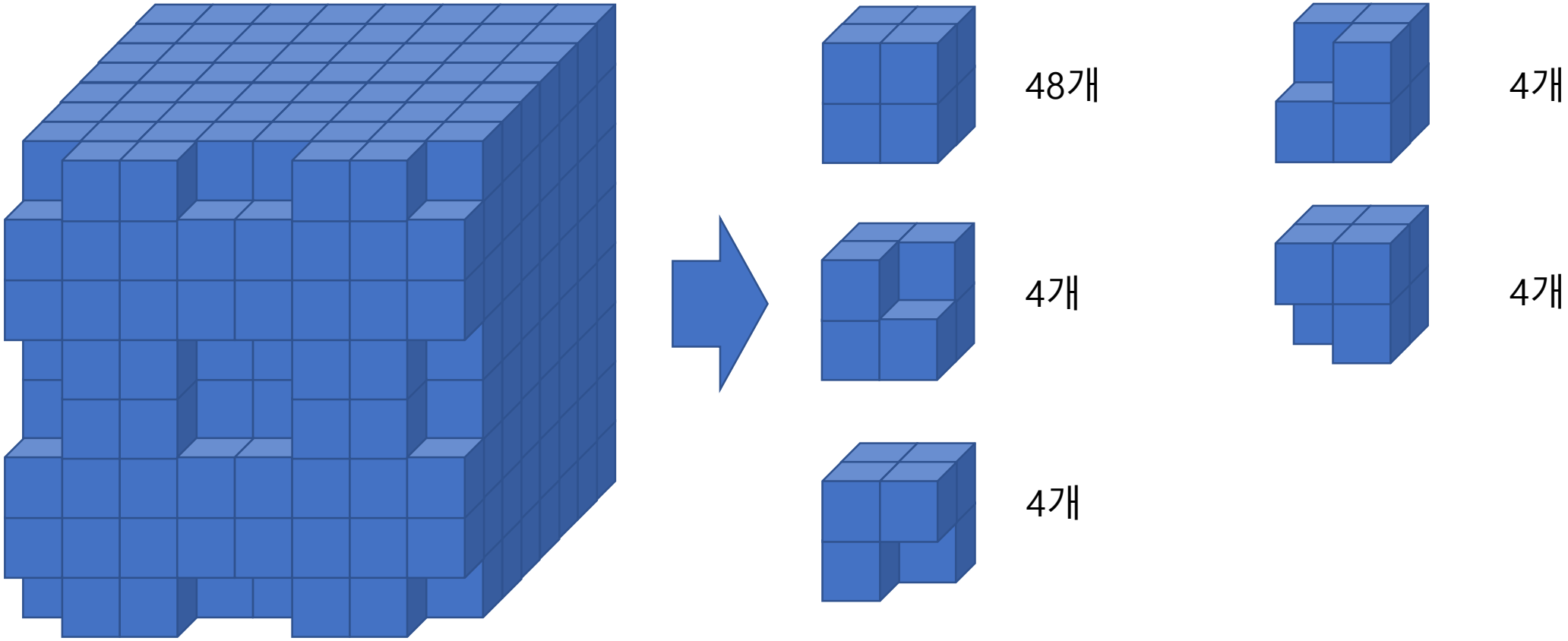
복셀 데이터 압축

- 복셀이 변형되는 이벤트의 패킷은 사이즈가 상당히 작아서 압축할 필요가 없다.
- 방문하지 않은 섹터로 이동한 경우 적지 않은 사이즈의 복셀데이터를 전송하게 된다.
- 로그인시에는 9섹터 분량의 복셀 데이터를 수신해야한다. 대량의 패킷 전송이 발생한다.
- 압축해서 패킷 사이즈를 줄일 필요가 있다.
- Zlib를 써봤지만 느리고 압축률은 너무 낮다.

복셀 데이터 압축

1. 8x8x8 복셀 오브젝트에서 2x2x2블럭으로 쪼개면 2x2x2블럭의 패턴은 8 bits, 최대 256가지.
2. 최대 패턴 개수 16개로 제한을 건다. 오브젝트 한개당 16개 패턴을 초과할 경우 압축불가로 처리.
3. 8 bits x 16 = 128 bits = 16 bytes, 이것이 패턴 팔레트 사이즈
4. 2x2x2블럭 하나는 16가지중 하나의 패턴을 가지게 되므로 각 블럭은 4 bits로 표현 가능.

5. 8x8x8오브젝트에서 2x2x2블럭의 개수는 64개. 4 bits x 64 = 256 bits = 32 bytes. 이것이 압축된 복셀 데이터 바디 사이즈
6. 패턴 팔레트16 bytes + 바디 32 bytes = 48 bytes.
7. 압축하지 않은 8x8x8복셀오브젝트의 복셀 데이터 사이즈는 64 bytes.
8. 패턴이 16개인 오브젝트는 64bytes -> 48 bytes 로 25% 사이즈를 줄일 수 있음.
9. 패턴이 8개 이하일때는 2x2x2블럭 하나를 3 bits로 표현 가능. 이 경우 패턴 팔레트 사이즈 8 bytes , 바디 사이즈 3 bits x 64 = 192 bits = 24 bytes , 합쳐서 32 bytes로 압축가능.
10. 패턴이 4개 이하일때 2x2x2블럭 하나를 2 bits로 표현 가능. 패턴 팔레트 사이즈 4 bytes , 바디 사이즈 2 bits x 64 = 128 bits = 16 bytes , 합쳐서 20 bytes로 압축가능.



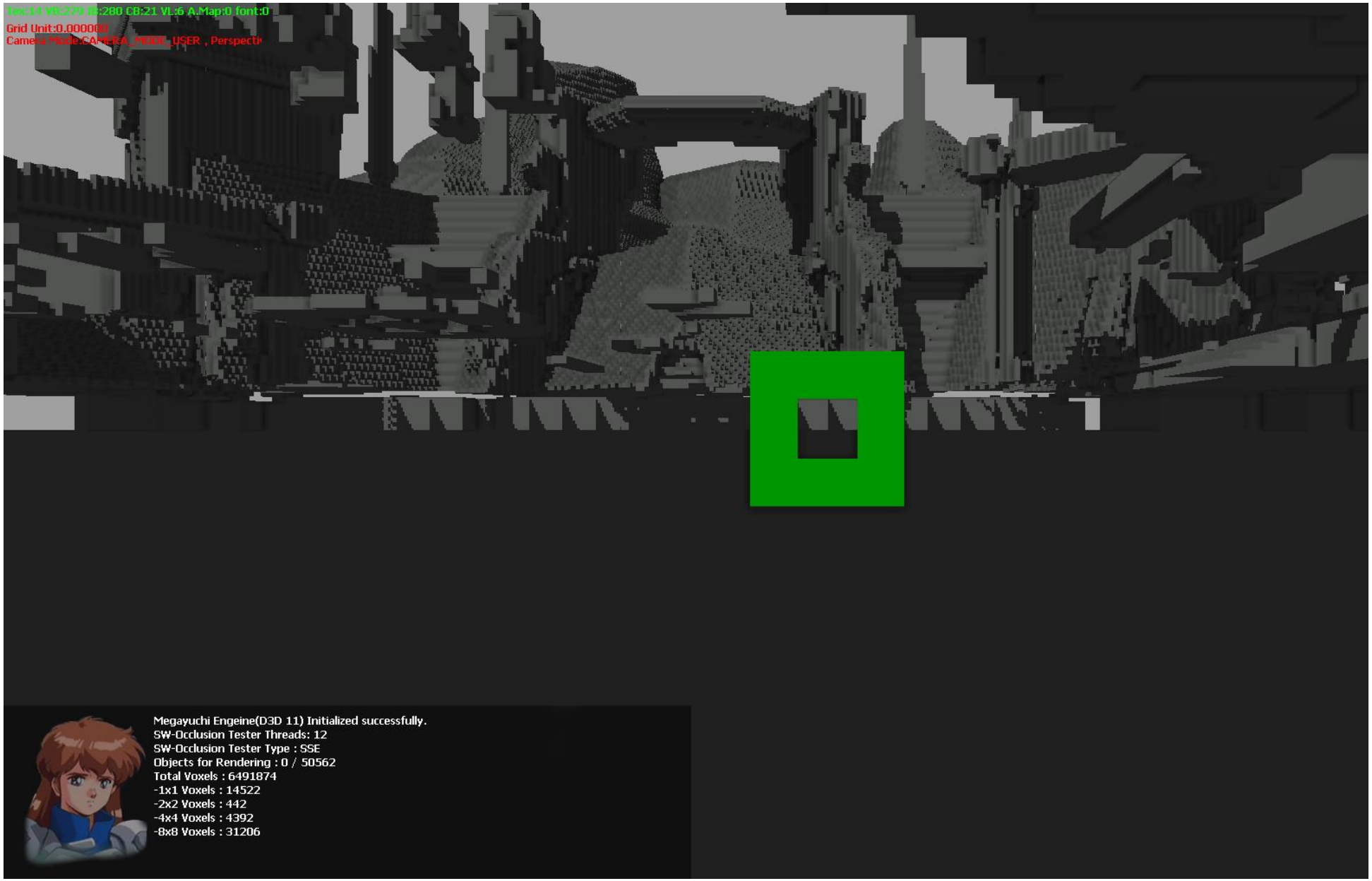
패턴 5개, 8개 미만이므로 패턴 인덱스는 3 Bits
3 Bits x 64 = 192 Bits = 24 Bytes
8x8x8복셀 오브젝트의 기본 사이즈는 512 Bits = 64 Bytes
24 / 64 = 37.5% -> 37.5%로 사이즈 감소

패턴 개수	오브젝트 개수
3	8417
4	8682
5	2370
6	3498
7	1902
8	1686
9	1481
10	1175
11	586
12	402
13	272
14	205
15	153
16	111

총 오브젝트 개수 : 50562

압축 가능한 오브젝트 수 : 30941

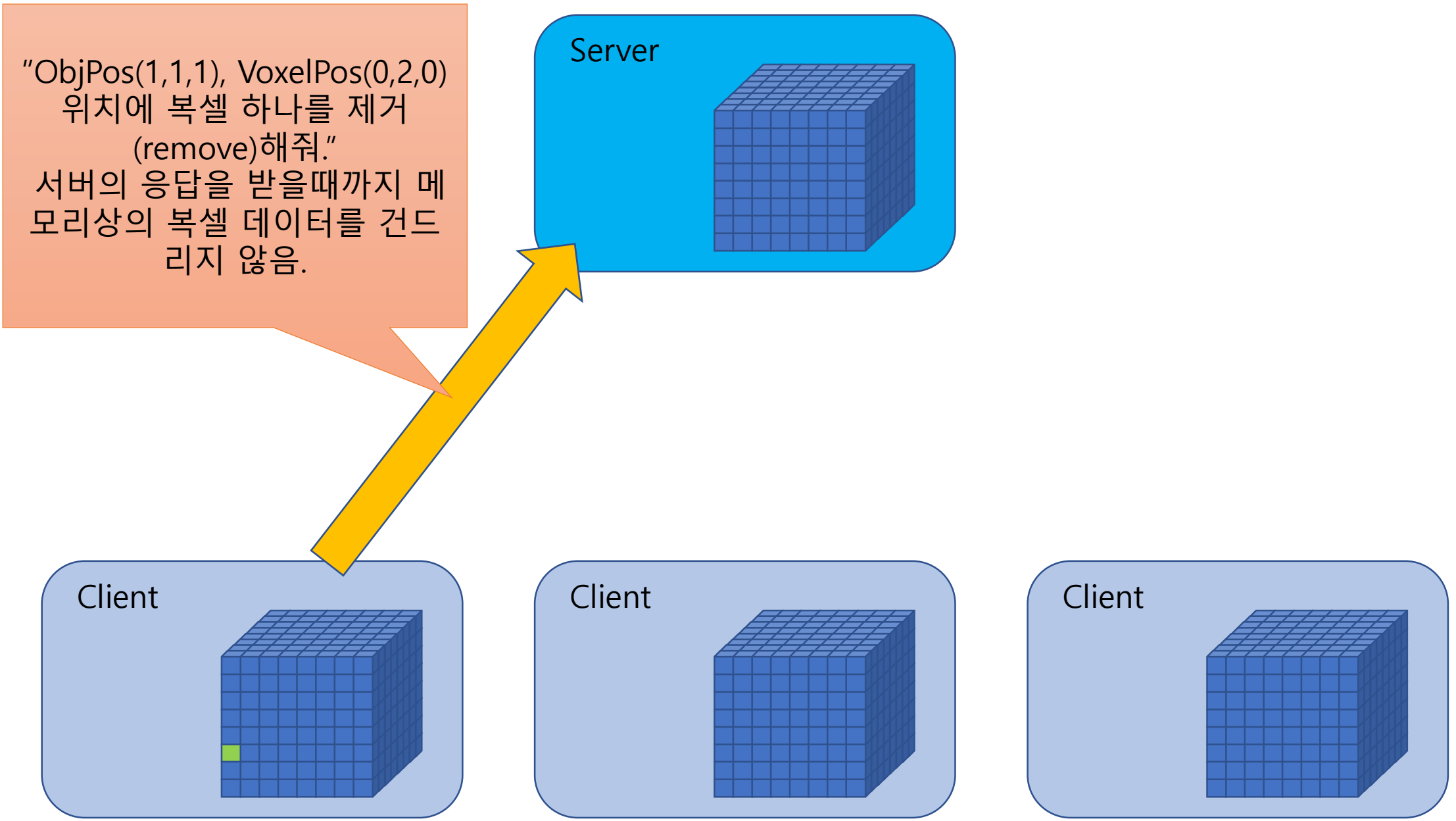
8x8x8오브젝트 개수 : 31206



스트리밍시 평균적으로 61%정도의 오브젝트에 대해 25% 패킷사이즈 감소.

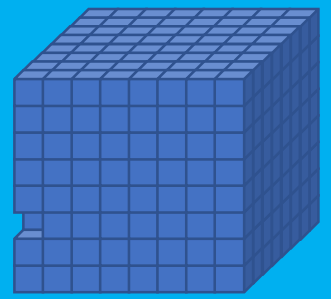
네트워크 이벤트에 의한 기하구조 변형

- 클라이언트에서 복셀 기하구조를 직접 전송하지 않는다.
- 서버에서도 복셀의 기하구조를 직접 전송하지 않는다.(플레이어가 새로운 섹터 진입시 VOXEL_APPEAR패킷으로 전달하는 경우는 제외)

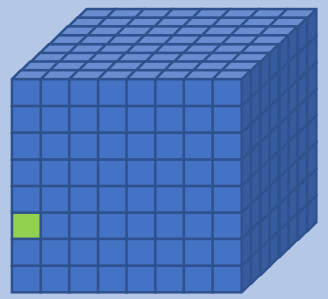

"ObjPos(1,1,1), VoxelPos(0,2,0) 위치에 복셀 하나를 제거(remove)해줘."
서버의 응답을 받을때까지 메모리상의 복셀 데이터를 건드리지 않음.
Server
Client
Client
Client

ObjPos(1,1,1), VoxelPos(0,2,0) 위치에 복셀이 실제로 존재하나?
Player#1 은 이 복셀 오브젝트에 대해 편집권한을 가지고 있나?
OK라면 서버측의 복셀 데이터 변경

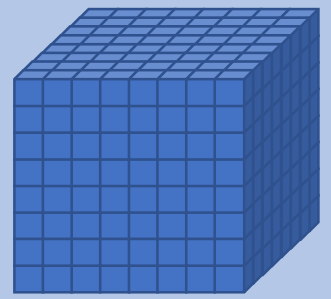
Server



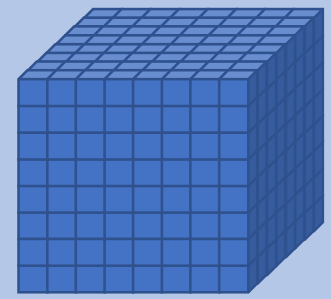
Client

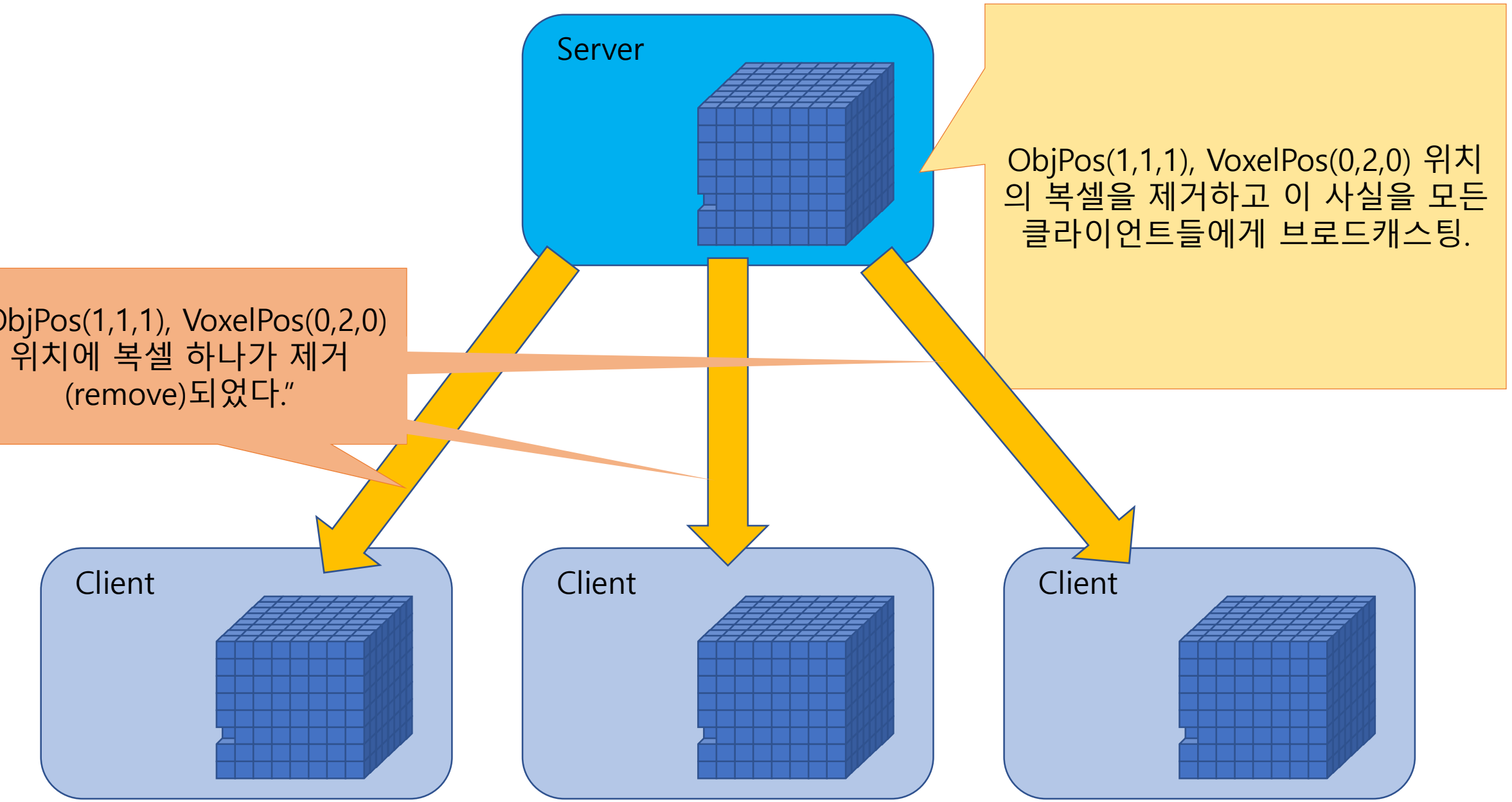


Client



Client



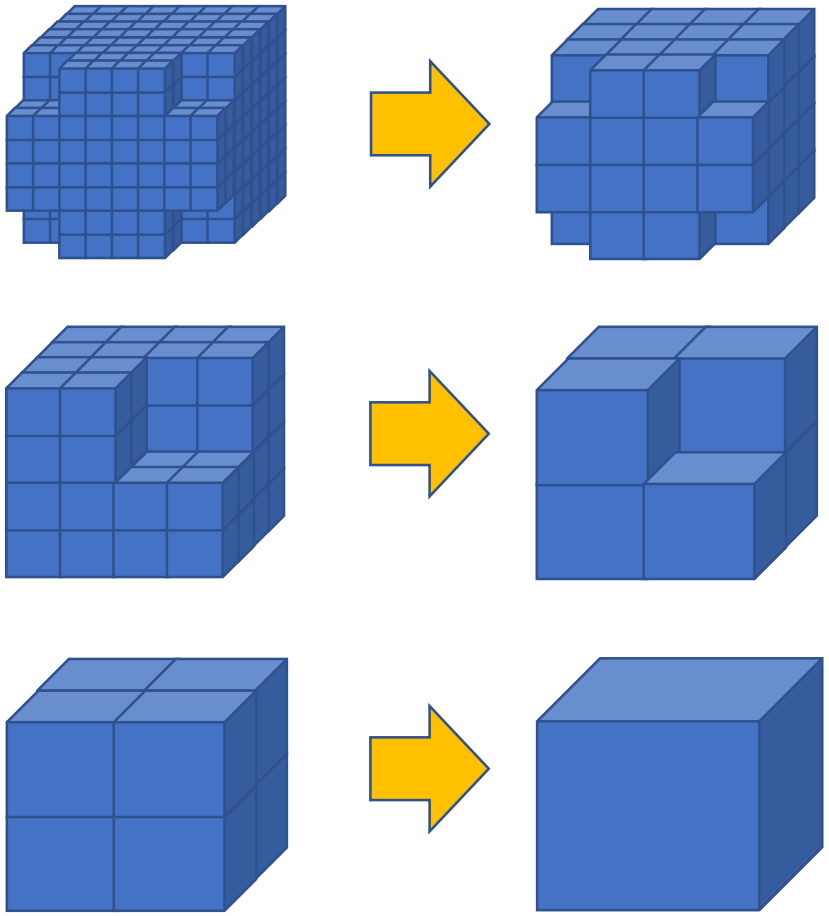

Server
ObjPos(1,1,1), VoxelPos(0,2,0) 위치의 복셀을 제거하고 이 사실을 모든 클라이언트들에게 브로드캐스팅.
bjPos(1,1,1), VoxelPos(0,2,0) 위치에 복셀 하나가 제거(remove)되었다."
Client
Client
Client

복셀 편집 데모

성능과 메모리절약을 복셀 최적화

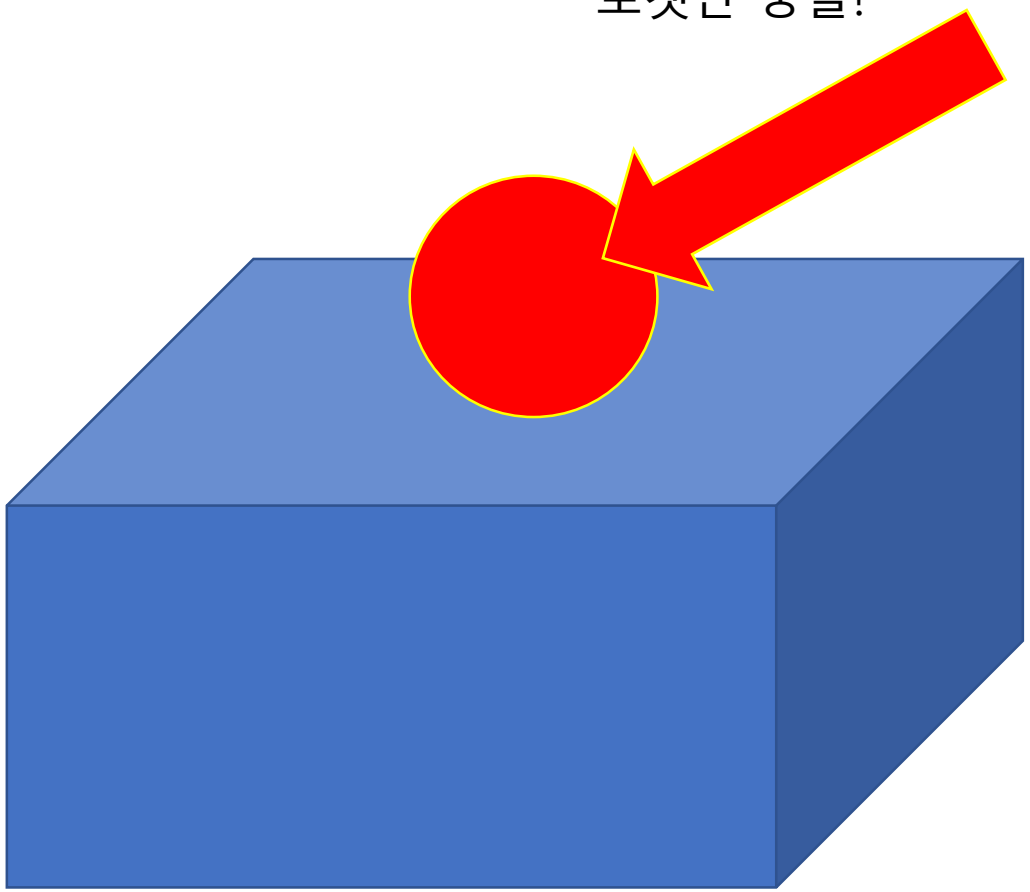
- 복셀 데이터의 정밀도는 가능한 낮게 유지한다.
- 서버 스타트 후 복셀 데이터 로드 후 복셀 데이터를 최적화.
- 모양이 변형되지 않는 선에서 최대한 낮은 정밀도로 변환.
- 50cm 단위로 복셀을 편집하거나 로켓탄이 터져서 8x8x8정밀도가 필요한 상황이 되면 즉시 8x8x8정밀도로 변환.
- 변환후에도 복셀의 기하구조를 직접 전송하지 않음. 변환 룰은 명백하므로 클라이언트에서도 알아서 변환.

모양을 유지하는 선에서 최적화

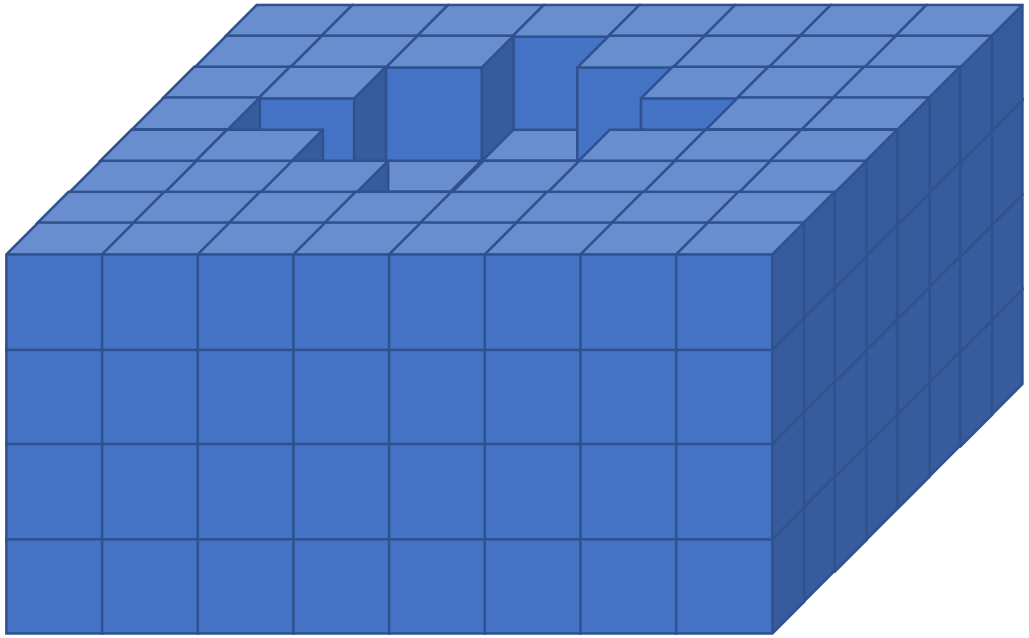


모양을 유지하는 선에서 최적화

로켓탄 충돌!



8x8x8로 변환



최대한 낮은 정밀도로 유지

Lighting

라이트맵 - 장점

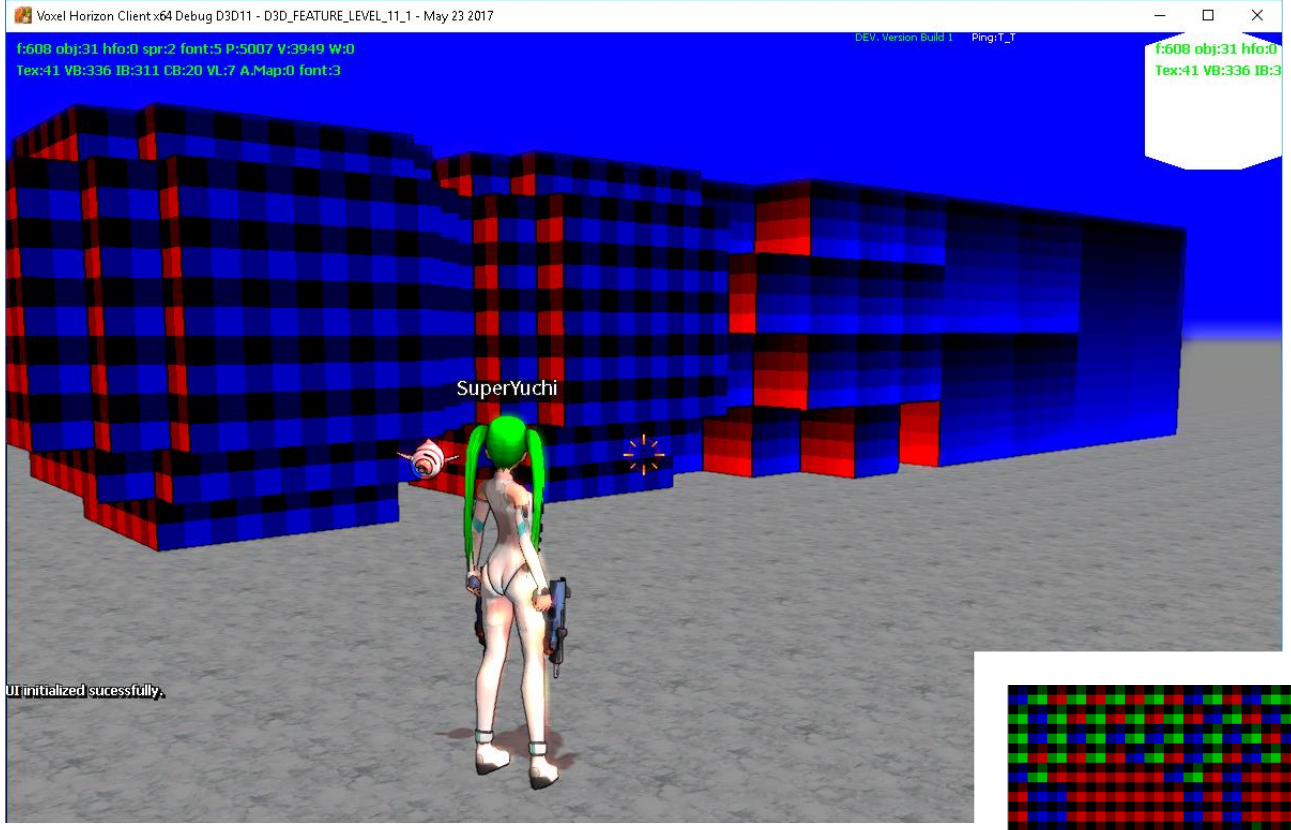
- 일단 구워놓으면 렌더링 중에 비용이 0에 가깝다.
- 그림자 처리시 카메라 방향에 따른 품질변화가 없다.

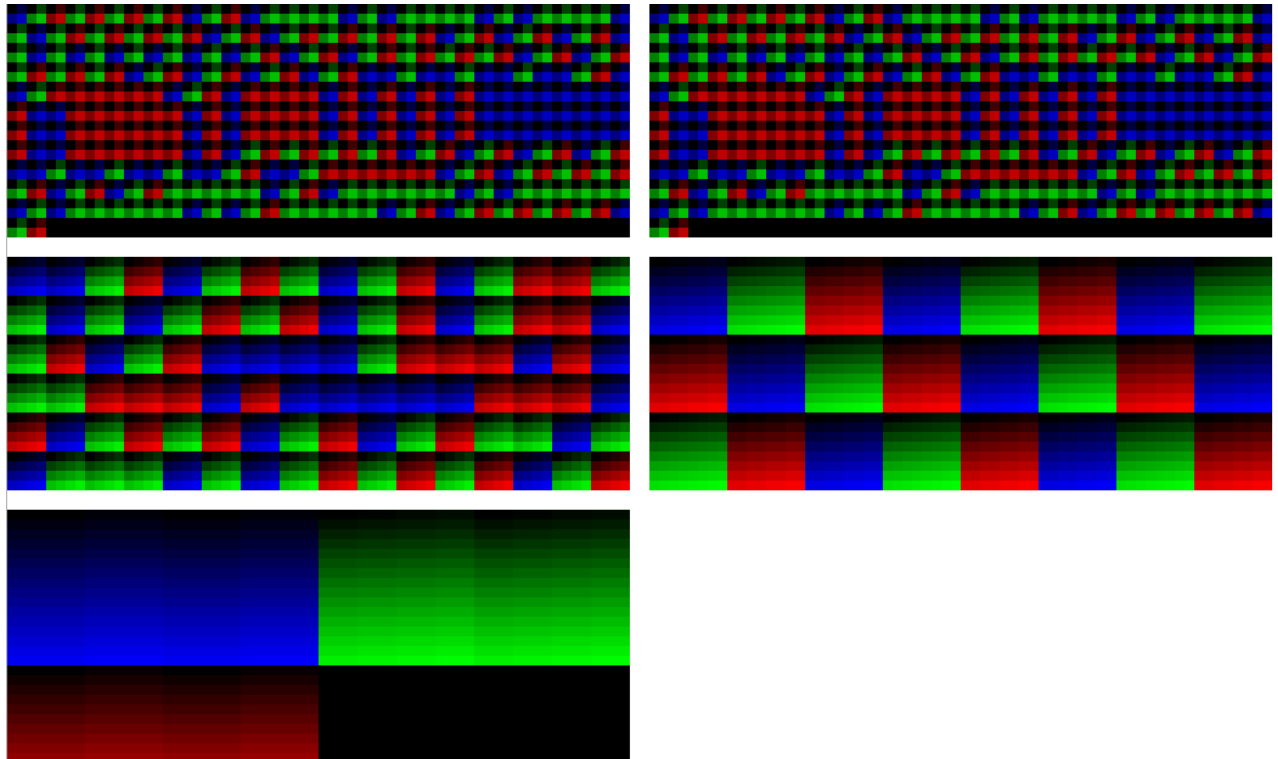
라이트맵 - 단점

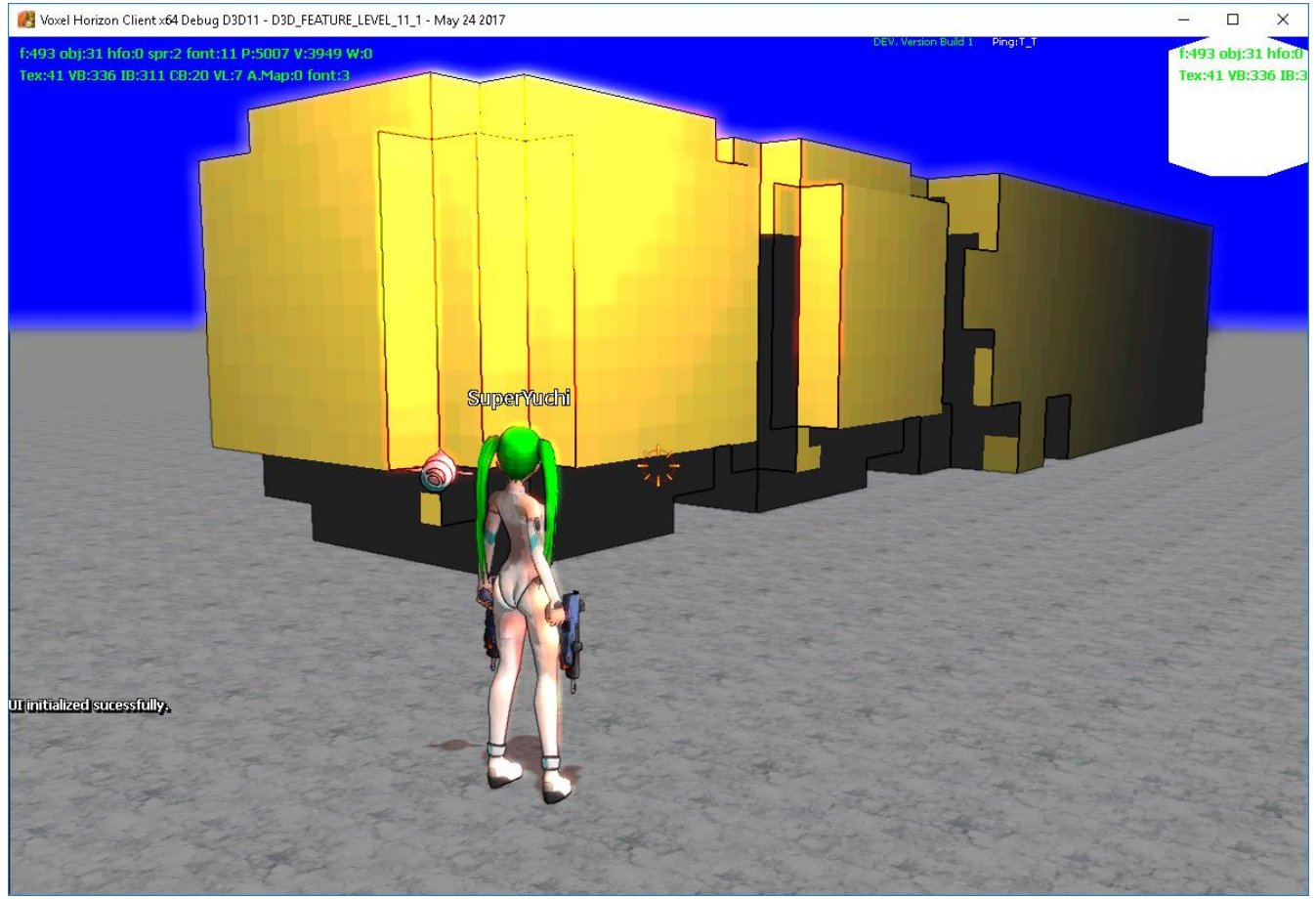
- 굽는데 오래 걸림.
- 게임 플레이중에 갱신하려면 patch 메모리를 계속 들고 있어야 함.

라이트맵 구현

- 오브젝트당 텍스쳐 한장
- 복셀의 한면-사각형 당 2x2 텍셀 -> 전개해서 팩킹
- vertex shader 안에서 라이트맵 텍스쳐의 좌표를 계산. Vertex압축에서 언급했던 QuadIndex와 qPos를 사용한다.


Voxel Horizon Client x64 Debug D3D11 - D3D_FEATURE_LEVEL_11_1 - May 23 2017
f:608 obj:31 hfo:0 spr:2 font:5 P:5007 V:3949 W:0
Tex:41 VB:336 IB:311 CB:20 VL:7 A.Map:0 font:3
DEV. Version Build 1 Ping:T_T
f:608 obj:31 hfo:0
Tex:41 VB:336 IB:3
SuperYuchi
UI initialized sucessfully.




Voxel Horizon Client x64 Debug D3D11 - D3D_FEATURE_LEVEL_11_1 - May 24 2017
f:493 obj:31 hfo:0 spr:2 font:11 P:5007 V:3949 W:0
Tex:41 VB:336 IB:311 CB:20 VL:7 A.Map:0 font:3
DEV. Version Build 1
Ping:T_T
SuperYuchi
UI initialized sucessfully.

실시간 라이트맵 베이킹

- 라이트맵이 갱신되어야할 오브젝트들을 큐에 넣음.
- 매 프레임마다 큐를 체크
- Thread Pool에 대기중이던 다수의 스레드가 큐에 들어있는 오브젝트들에 대해 라이트맵 베이킹 및 라이트 텍스쳐 갱신. 이 때 메인스레드 대기.
- 베이킹중 일정 시간을 초과하면 베이킹을 중단하고 게임 루프를 속행.
- 네트워크로 복셀지형을 스트리밍하는 경우 충분히 빠름.
- CUDA를 사용하면 성능 대폭 향상 가능.
  https://youtu.be/IxvbqsW8D2A

결론

- 메모리 겁나 먹어요. 무조건 최대한 아끼고 압축한다!
- 무조건 느려요. 최대한 빠르게! 빠르게! 빠르게!
- 열심히 합시다.

Q/A