

# GPU를 이용한 Voxelization

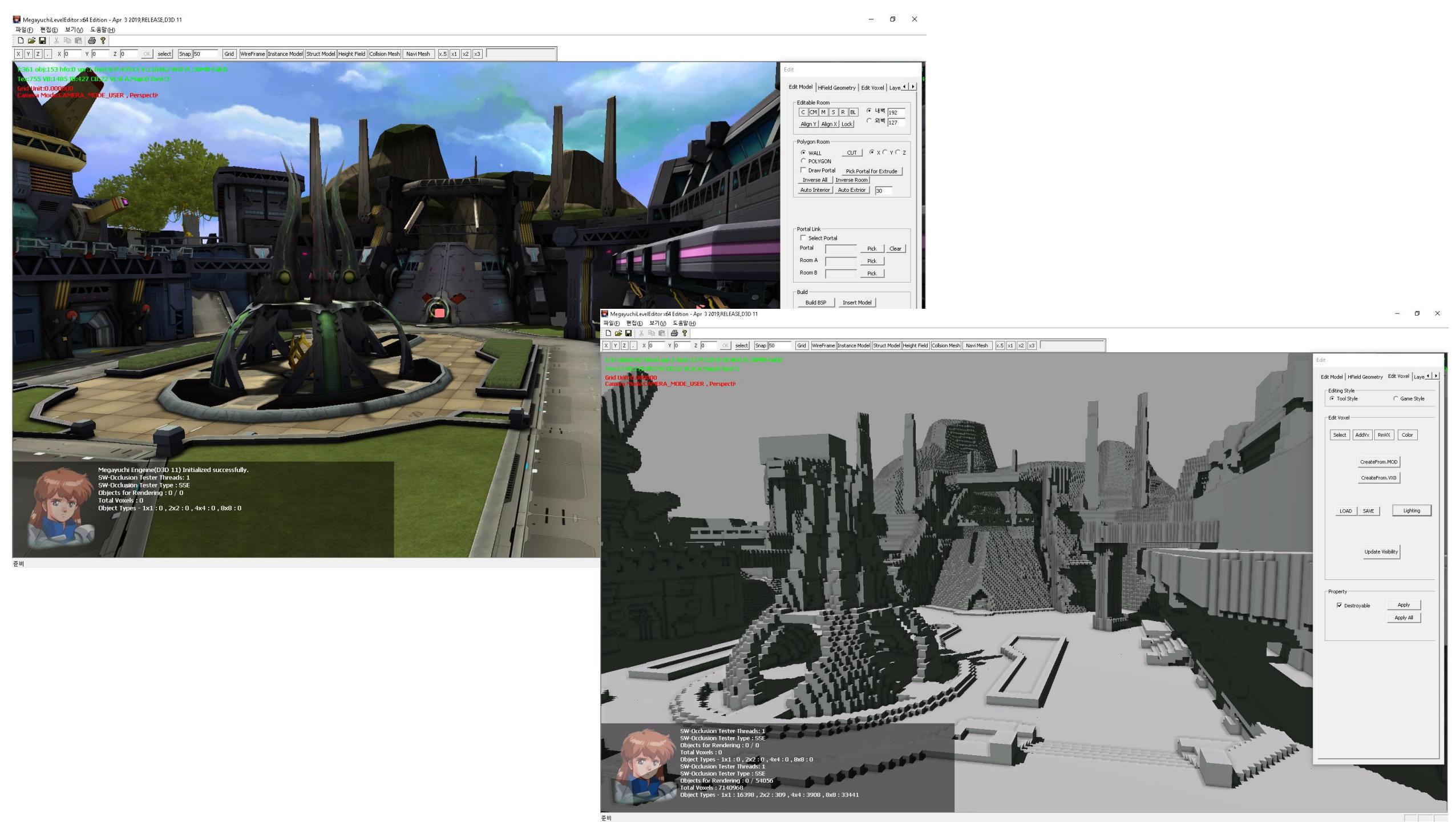
유영천

<https://megayuchi.com>

tw : @dgtman

# Voxelization

삼각형 매시를 voxel 데이터로 변환.





# Voxel 모델 데이터, 어디다 쓸까?

- Occlusion Culling을 위한 단순한 AABB 매시를 자동으로 생성.
- Voxel 기반 게임에서 월드 그 자체.
- Auto skinning – vertex weight계산.
- 지형편집
- 기타 등등....

# Surface Voxelization

- 삼각형과 교차하는 영역만 voxel로 변환.
- 이렇게 변환한 voxel 모델은 안이 비어있는 껍데기 형태가 된다.
- 이렇게 변환된 voxel 모델은 이후 'surface voxel'로 표기.
- 껍데기만으로는 그다지 쓸모가 없다.

# Solid Voxelization

- 삼각형 매시의 안을 꽉 채운 voxel 데이터로 변환
- 이렇게 변환해서 얻은 voxel데이터는 이후 'solid voxel'로 표기 함.
- 대부분의 경우 solid voxel 데이터가 필요하다.

Triangle mesh



Surface  
voxelization



Solid  
voxelization



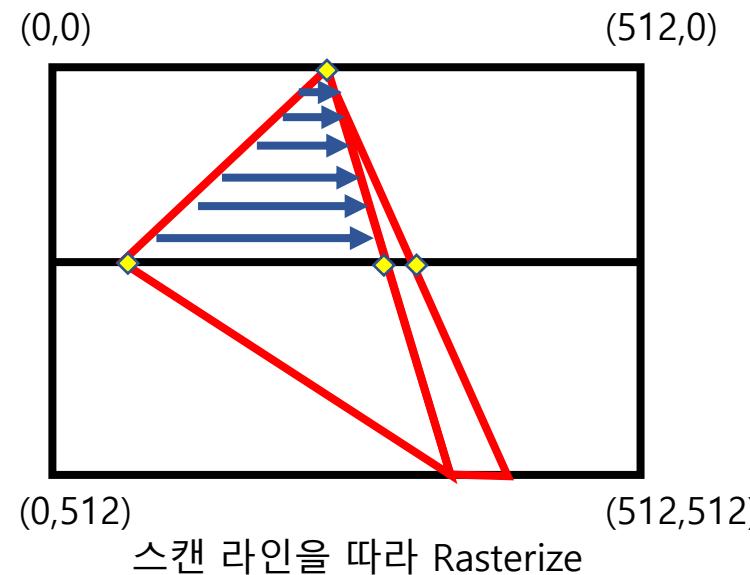
# Rasterization-based Voxelization

오늘의 주제가 아님. 소개만 할 뿐.

# Rasterization-based Voxelization

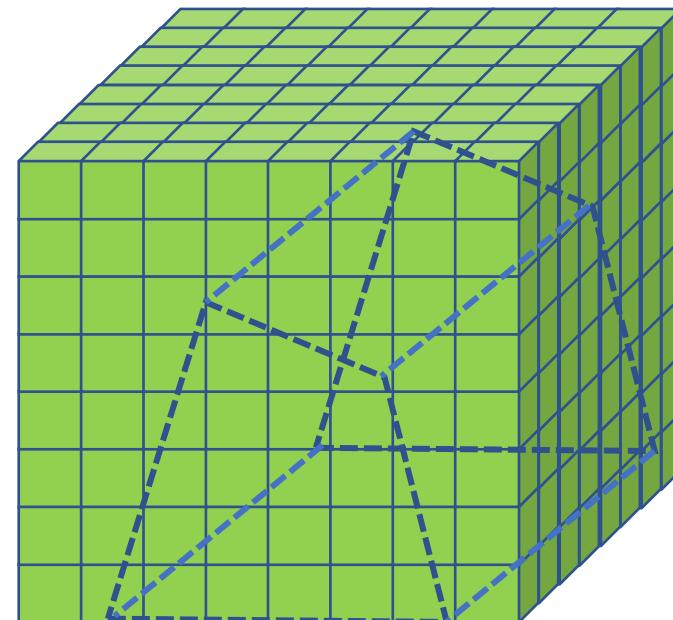
- Surface Voxelization을 먼저 수행.
- 이렇게 만들어진 surface voxel데이터를 기초로 Solid Voxelization을 수행.

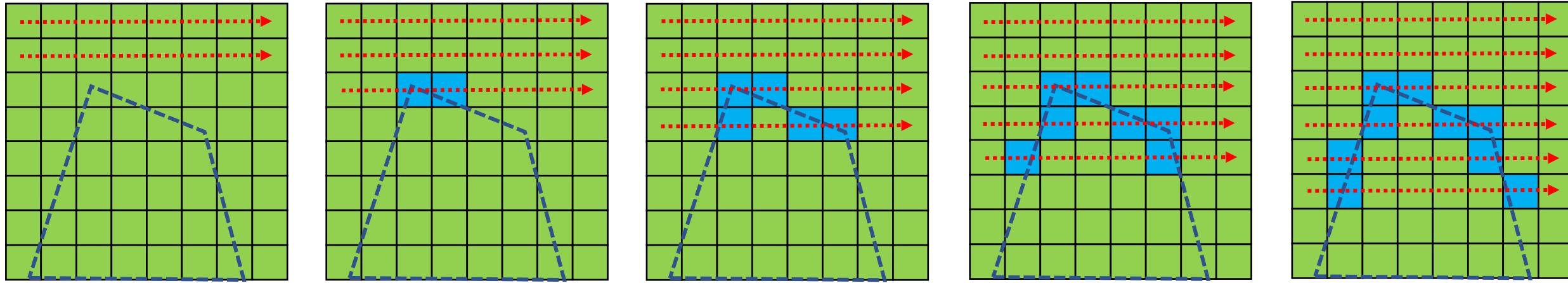
3D공간의 삼각형을 2D로  
투영해서 내부를 채워가는  
방법과 비슷하다. –  
Triangle rasterization.



# (1) Surface voxelization

- 월드를 꽉 찬 배열로 구성(3D배열). 각 cell은 empty cell로 표시해둔다.
- 삼각형 매시에 대해 배열의 cell마다 교차 테스트
- 교차하면 surface voxel로 표시.  
아니면 empty cell.

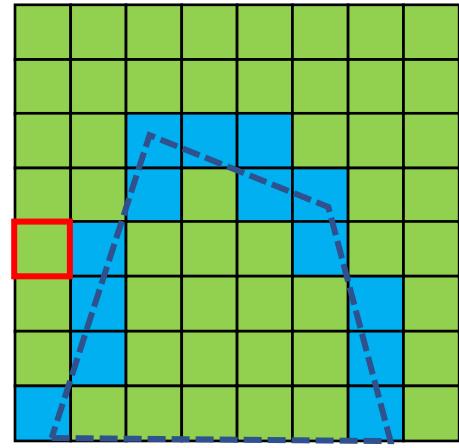




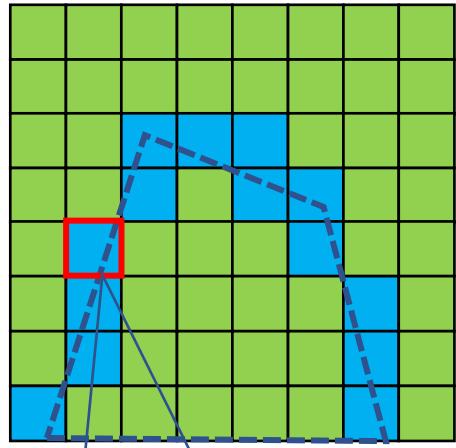
- 3D배열을 순회하면서 각 cell의 AABB를 구한다.
- 각 cell의 AABB와 삼각형이 교차하는지 테스트한다.
- 교차하면 Surface Voxel이다. 교차하지 않으면 empty cell이다.

## (2) Solid voxelization

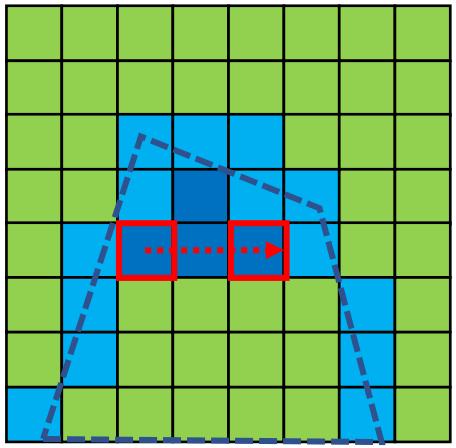
- +X축 방향으로 배열의 cell을 순회(어느 축을 기준으로 해도 상관없음).
- surface voxel을 만나면 교차하는 삼각형의 면 방향을 체크. 진행 방향에 대해 마주보고 있으면 solid voxel mode ON.
- 이후로 마주치는 empty cell은 solid voxel로 표시.
- 이후 surface voxel을 만나면 교차하는 삼각형의 면 방향을 체크. 진행 방향에 대해 뒤집혀 있으면 solid voxel mode OFF.
- 이후로 마주치는 empty cell은 그대로 empty cell로 표시.



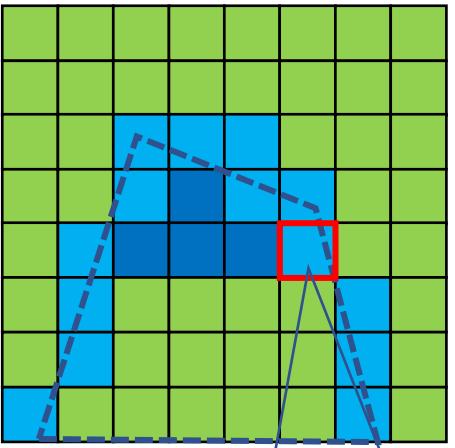
solid voxel mode  
OFF 상태로 empty  
cell을 만났으므로  
무시.



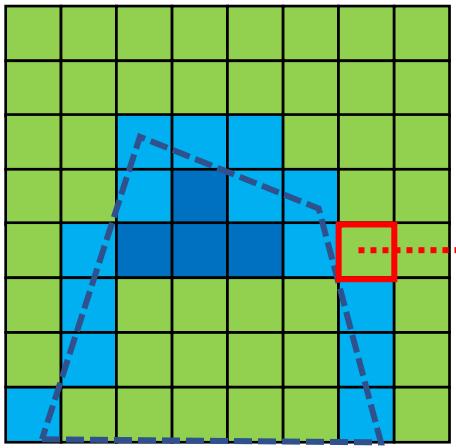
진행방향에 대해  
마주보고 있으므로  
solid voxel mode  
ON



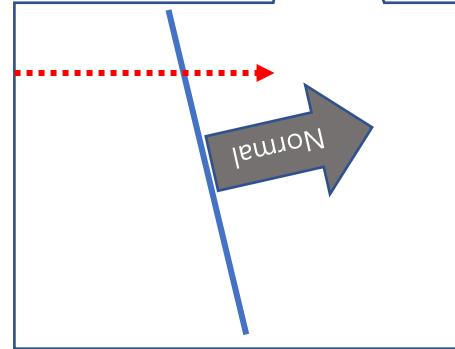
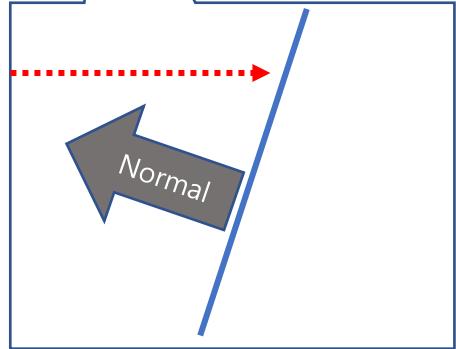
solid voxel mode  
ON 상태로 empty  
cell을 만났으므로  
solid voxel로 표시.



진행방향에 뒤집혀  
있으므로 solid  
voxel mode OFF

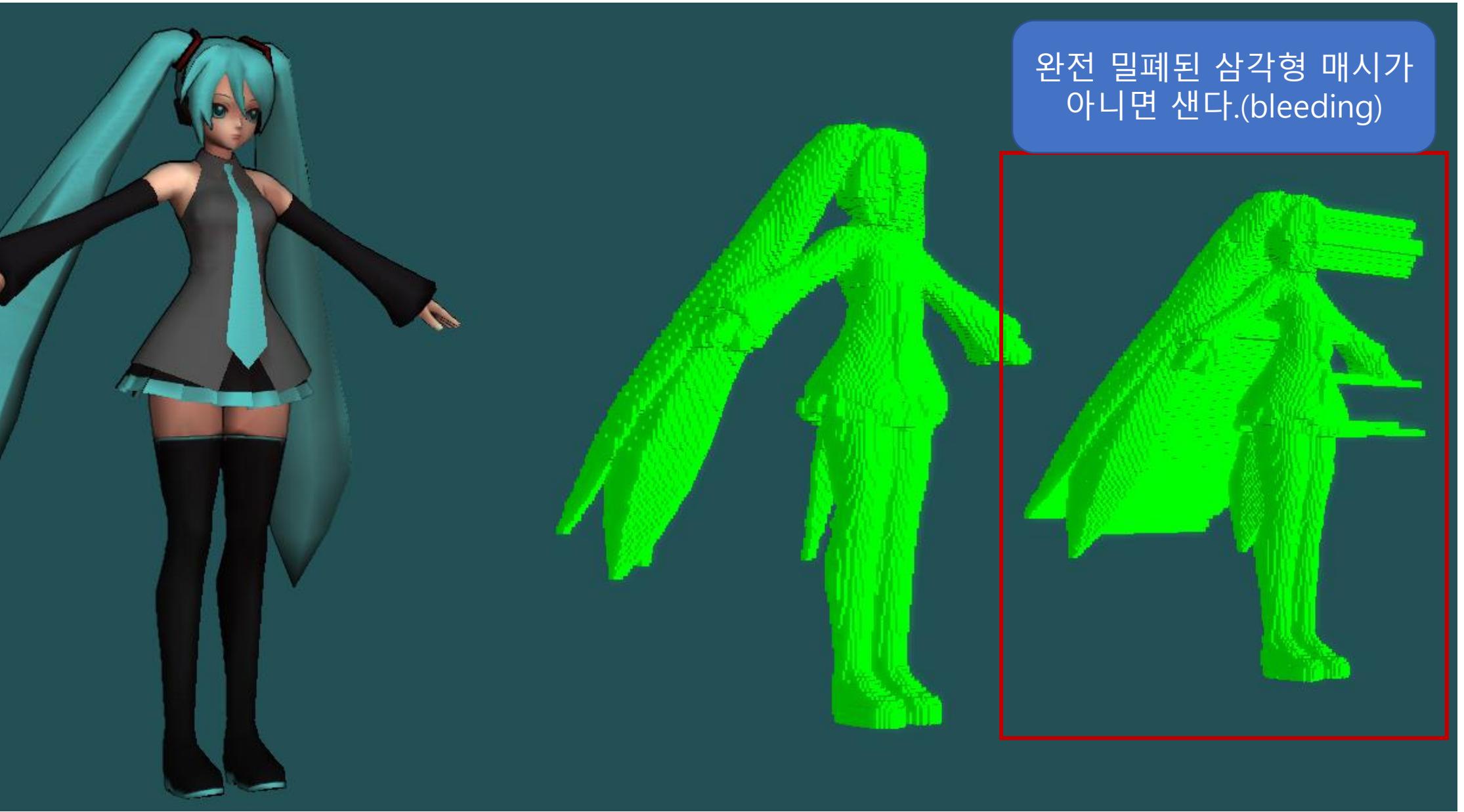


solid voxel mode  
OFF 상태로 empty  
cell을 만났으므로  
무시.



# 문제점

- 완전히 닫힌 매시가 아니라면 voxel이 샌다.(bleeding)
- 기본적으로 surface voxel을 먼저 생성하고, 이를 이용해서 solid voxel을 찾는 대부분의 알고리즘은 닫힌 매시(watertight)를 요구한다.
- 캐릭터라면 몰라도, 게임에서 사용할 월드 맵의 모델링 데이터는 닫혀있지 않을 가능성이 높다.
- 따라서 다른 방법이 필요하다.

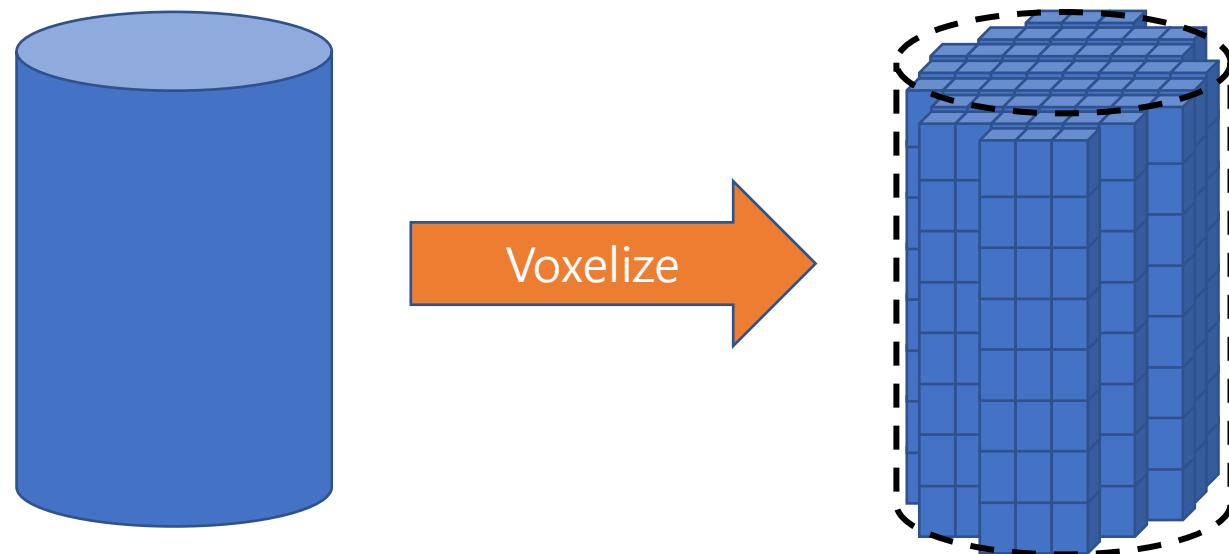


완전 밀폐된 삼각형 매시가  
아니면 샌다.(bleeding)

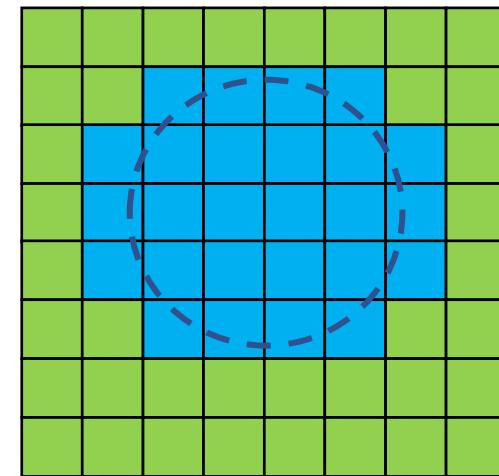
# Visibility-based voxelization

무식하다! 그러나 확실하다!

# Voxelization(하려는 것)

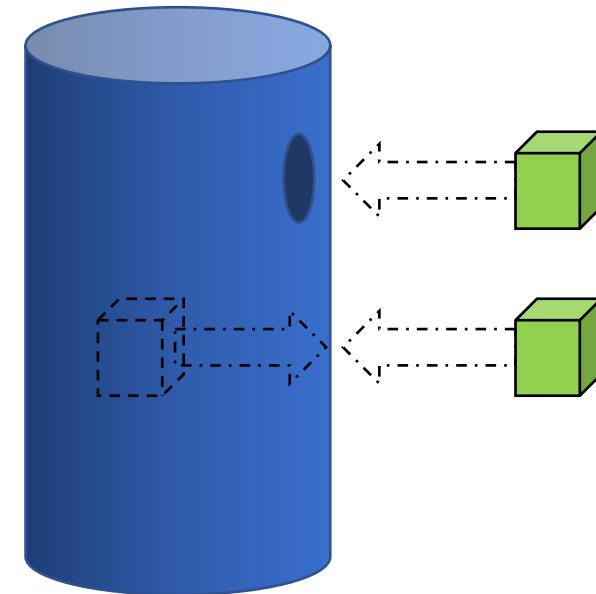


Top View



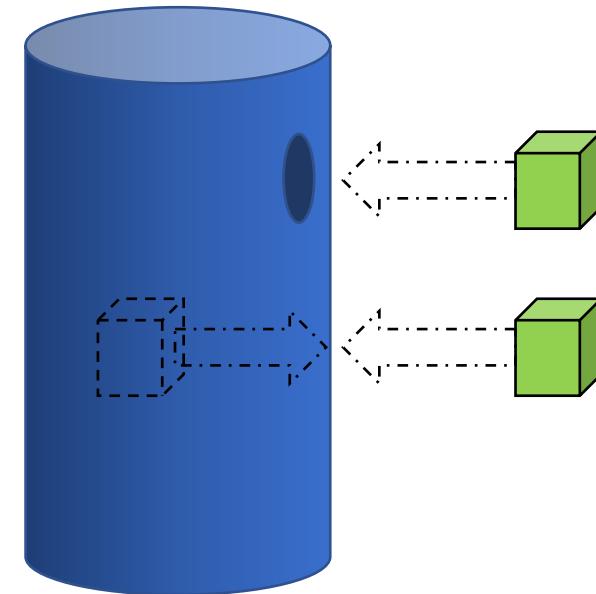
# 기본 개념

- 밀폐된 통 안에 갇혀서 주변을 둘러본다면 통 안쪽만 보인다.
  - 통 안에 있는 것이 확실하다!
- 반대로 통의 바깥쪽 면이 보인다면 통의 바깥에 있을 가능성이 아주 높다.
  - 통 바깥에 있을 가능성이 매우 높다.
- 통 바깥쪽 면도 보이긴 하지만 주로 통 안쪽만 보인다.
  - 통 안에 있을 가능성이 매우 높다!

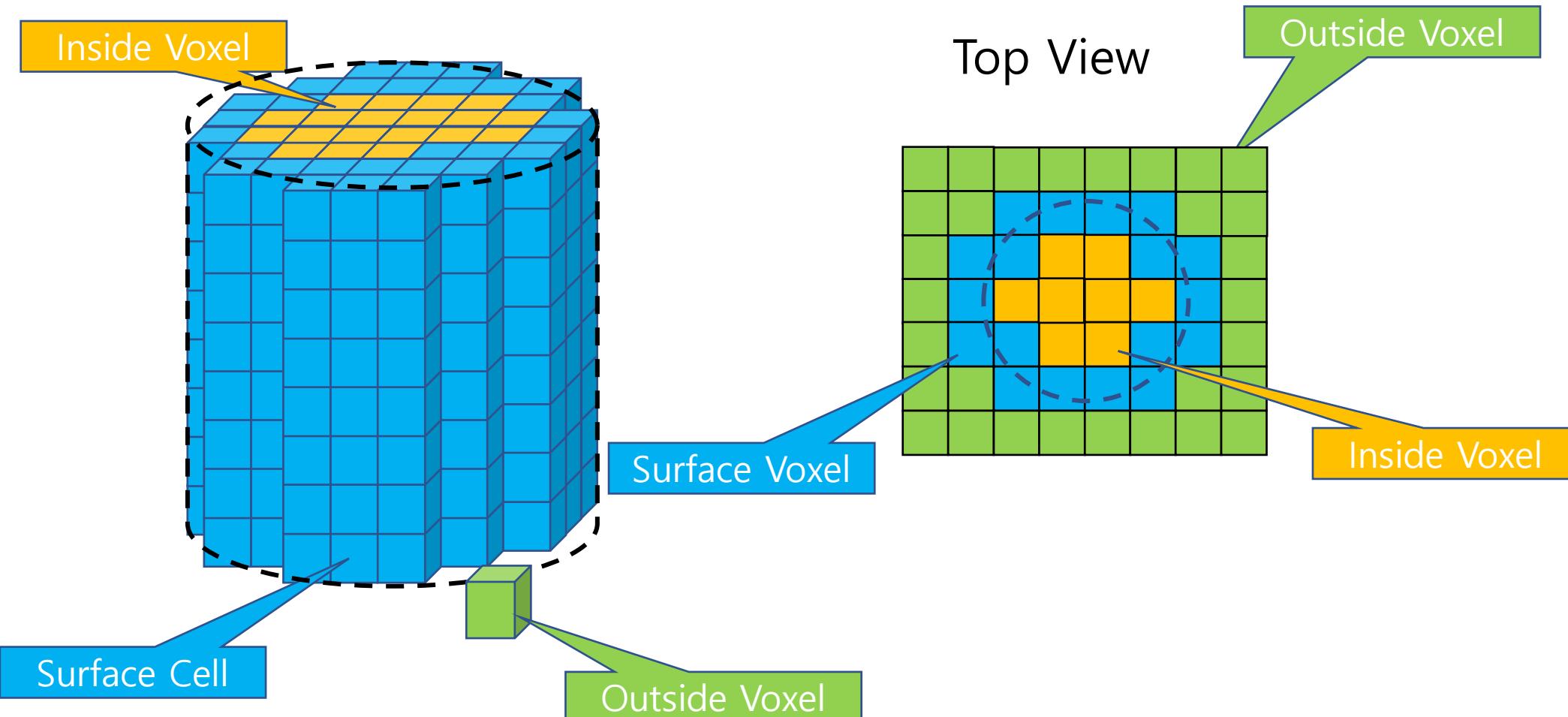


# 기본 개념

- 삼각형 매시 안에서 주변을 둘러본다면 삼각형의 뒤집힌 면이 보인다.
  - 삼각형 매시 안에 있는 것이 확실하다!
- 반대로 삼각형의 노말 방향 면이 보인다면 삼각형 매시의 바깥에 있을 가능성이 아주 높다.
  - 삼각형 매시 바깥에 있을 가능성이 매우 높다.
- 삼각형의 노말 방향 면도 보이지만 주로 삼각형의 뒤집힌 면이 보인다.
  - 삼각형 매시 안에 있을 가능성이 매우 높다!



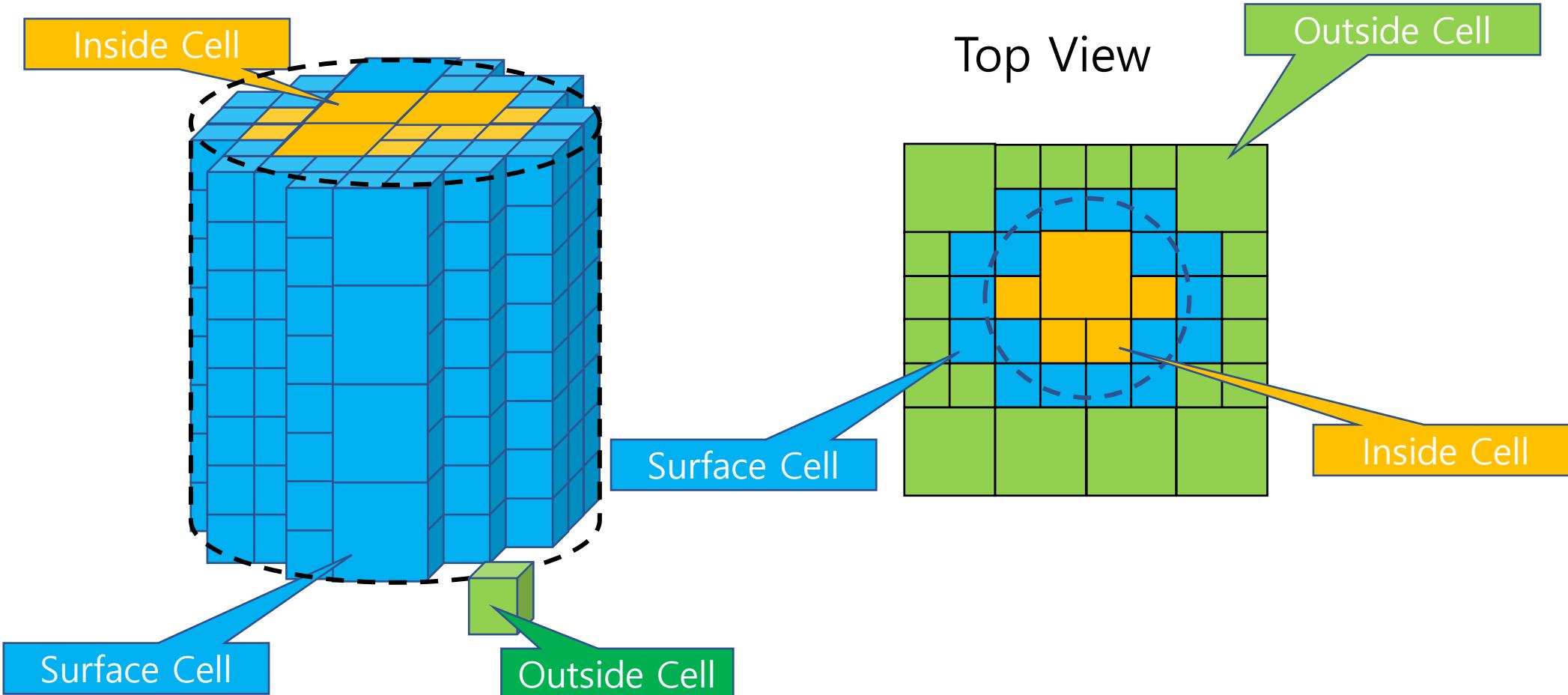
- Inside Voxel – 삼각형과 교차하지 않고 삼각형 매시의 안쪽에 위치함.
- Surface Voxel – 삼각형과 교차하는 voxel
- Outside Voxel – 삼각형과 교차하지 않고 삼각형 매시의 바깥쪽에 위치함.



# Visibility 검사를 하기 위한 사전작업

- 삼각형 매시 전체의 월드 영역을 분할해서 Octree를 만든다.
- 삼각형과 교차하면 sub-node로 분할.
- node가 최소 사이즈에 도달하면 분할하지 않고 leaf로 처리.
- 삼각형과 교차하지 않으면 분할하지 않고 leaf로 처리.
- 삼각형과 교차하지 않는 leaf를 pure cell로 부른다.

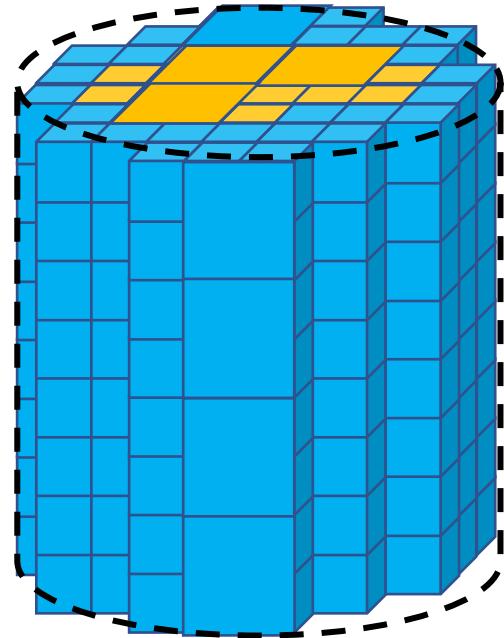
- 삼각형 매시 전체의 월드 영역을 분할해서 Octree를 만든다.
- 삼각형과 교차하면 sub-node로 분할.
- node가 최소 사이즈에 도달하면 분할하지 않고 leaf로 처리.
- 삼각형과 교차하지 않으면 분할하지 않고 leaf로 처리.
- 삼각형과 교차하지 않는 leaf를 pure cell로 부른다.



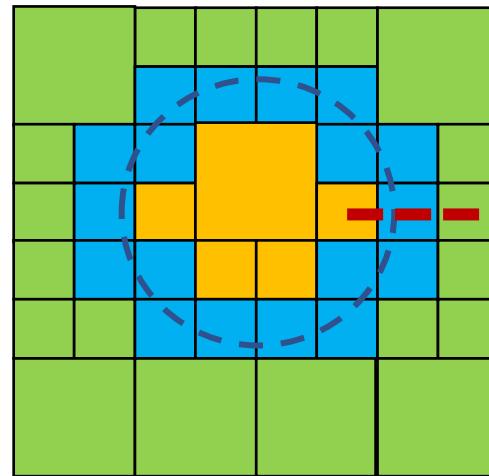
# Visibility 검사

- pure cell의 6면을 카메라 방향과 fov로 간주하고 360도 카메라를 만든다. 즉 6개의 matrix가 나온다.
- 6개의 카메라 맞춰서 6개의 frame buffer와 z-buffer를 준비한다.
- 6면에 대해 렌더링한다. 즉 cell의 중심점에서 360도 렌더링 하는 것이다.
- 삼각형 면 방향이 cell의 중심점을 바라보고 있으면 파란색으로, 중심점에 대해 뒤집혀 있으면 빨간색으로 렌더링한다.

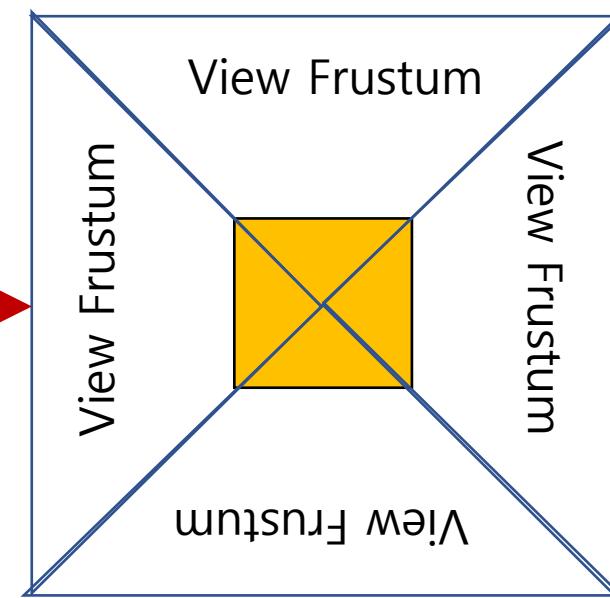
- pure cell의 6면을 카메라 방향과 fov로 간주하고 360도 카메라를 만든다. 즉 6개의 matrix가 나온다.
- 6개의 카메라 맞춰서 6개의 frame buffer와 z-buffer를 준비한다.
- 6면에 대해 렌더링한다. 즉 cell의 중심점에서 360도 렌더링 하는 것이다.
- 삼각형 면 방향이 cell의 중심점을 바라보고 있으면 파란색으로, 중심점에 대해 뒤집혀 있으면 빨간색으로 렌더링한다.



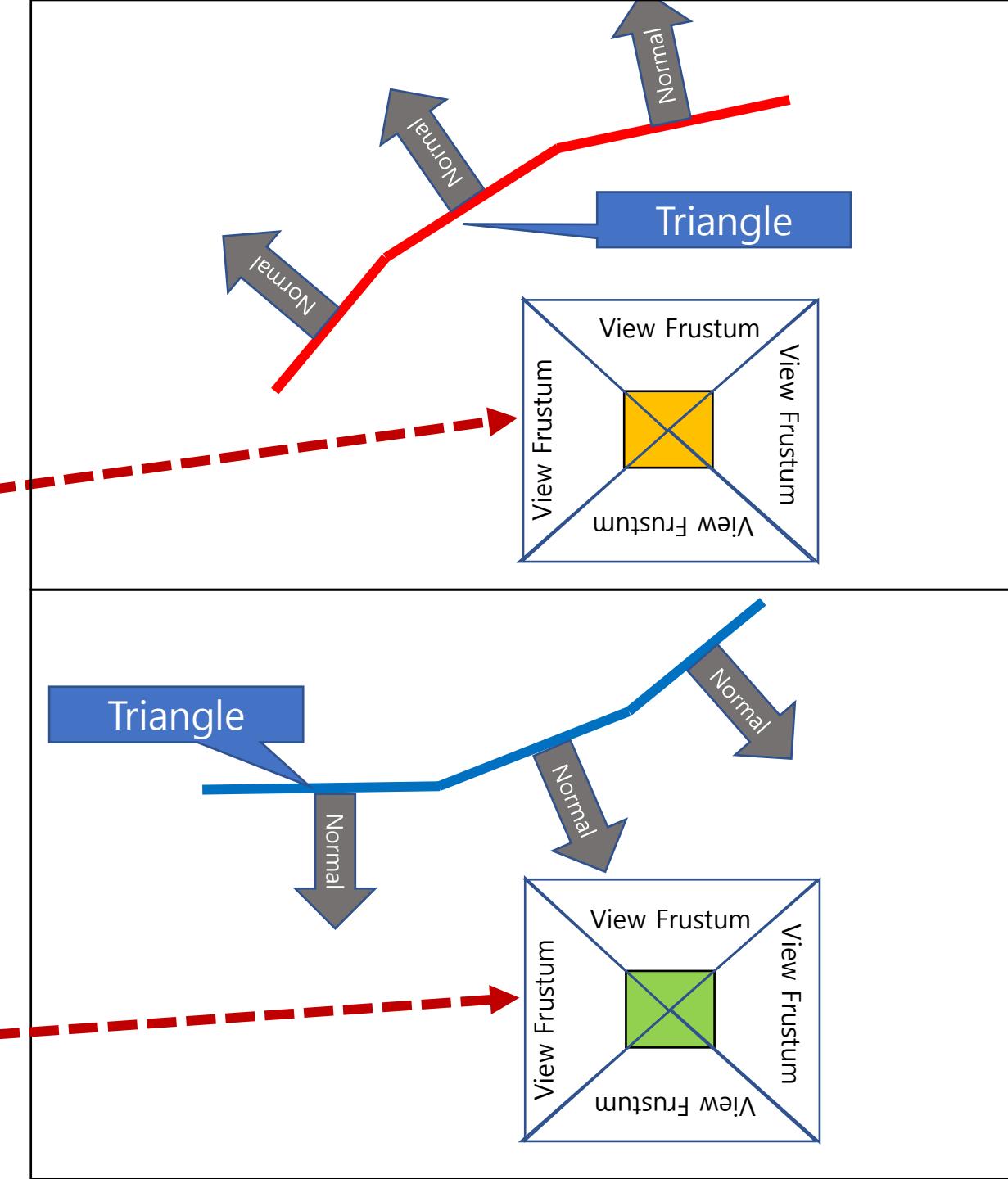
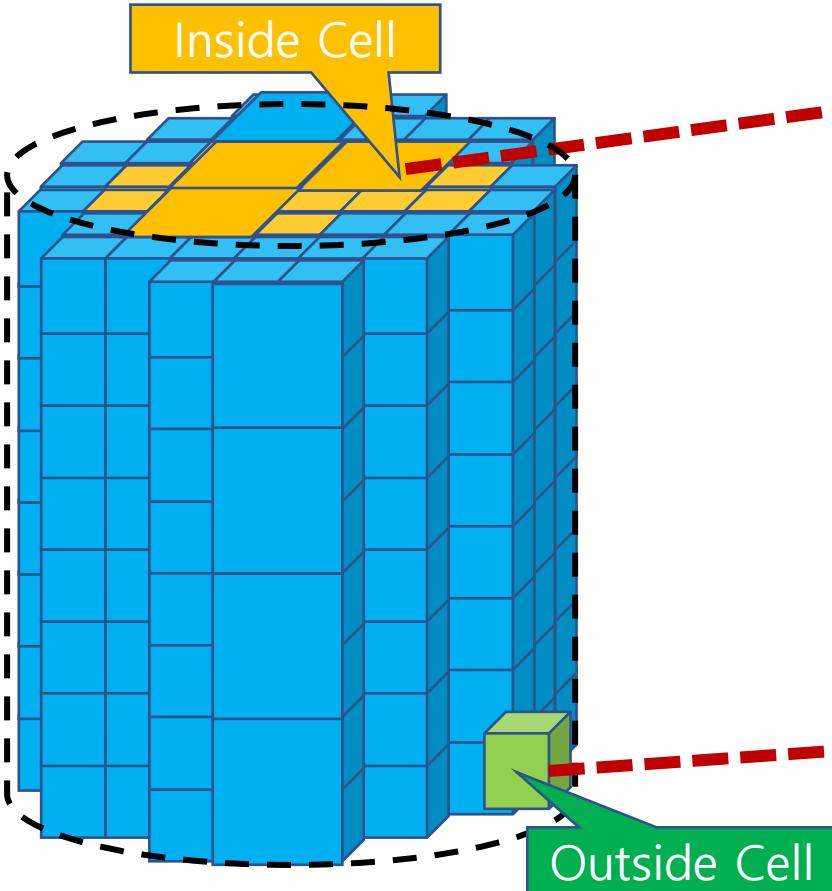
Top View



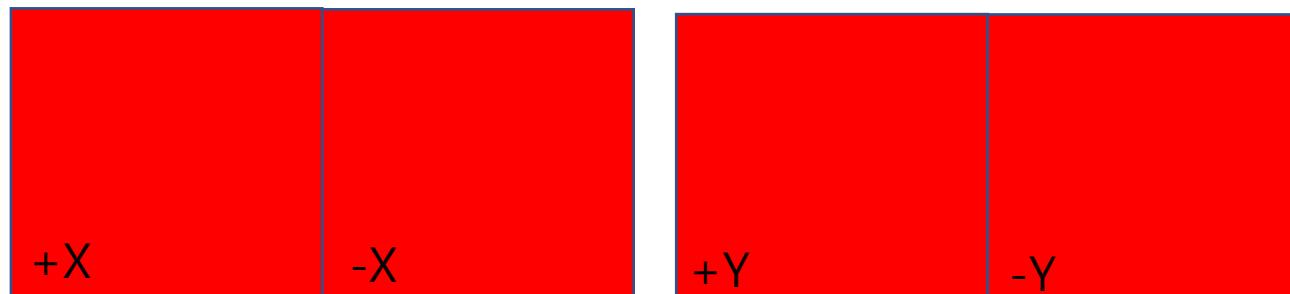
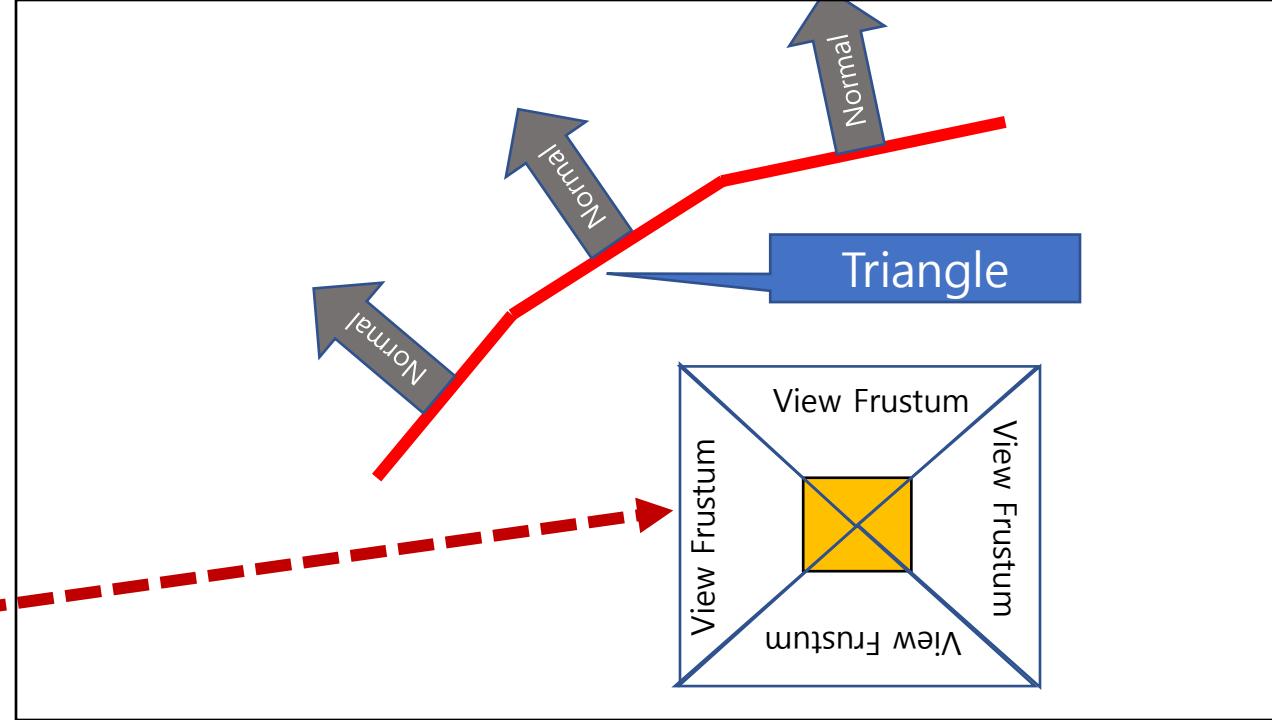
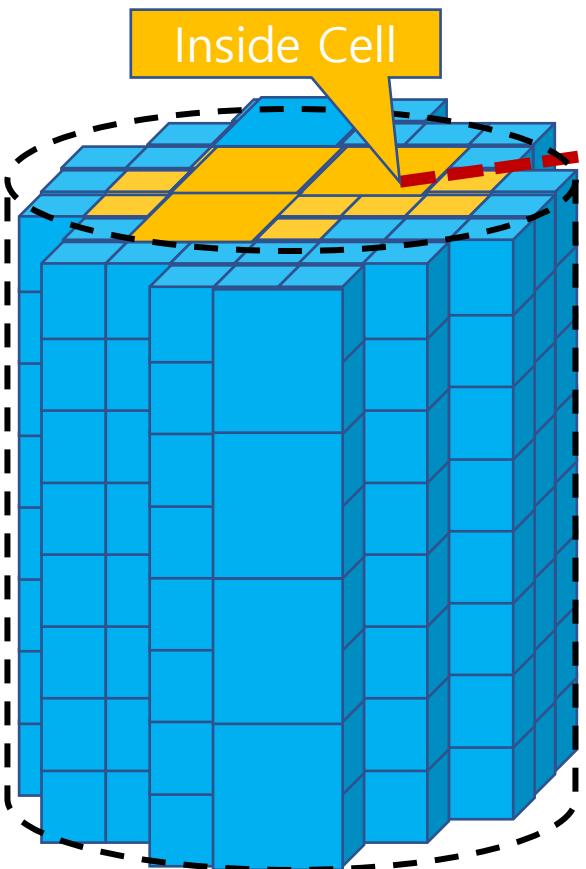
360deg camera



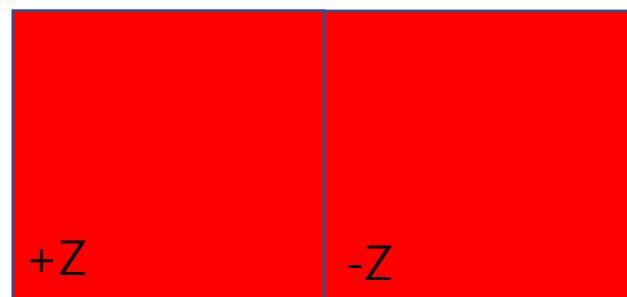
- 아직 Inside Cell과 Outside Cell을 판별할 수 없다.  
알고 있는 것은 Surface Cell뿐이다.
- Surface Cell을 제외한 모든 pure cell을 순회하면서  
6개의 카메라로 삼각형 매시를 렌더링한다.
- 삼각형 면이 카메라를 바라보고 있으면 파란색.  
면이 뒤집혀있는 경우 빨간색으로 렌더링한다.



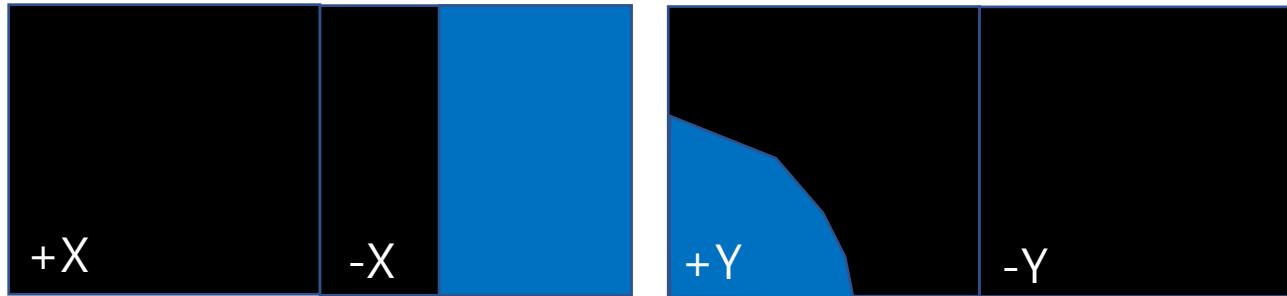
# Inside Cell일 경우



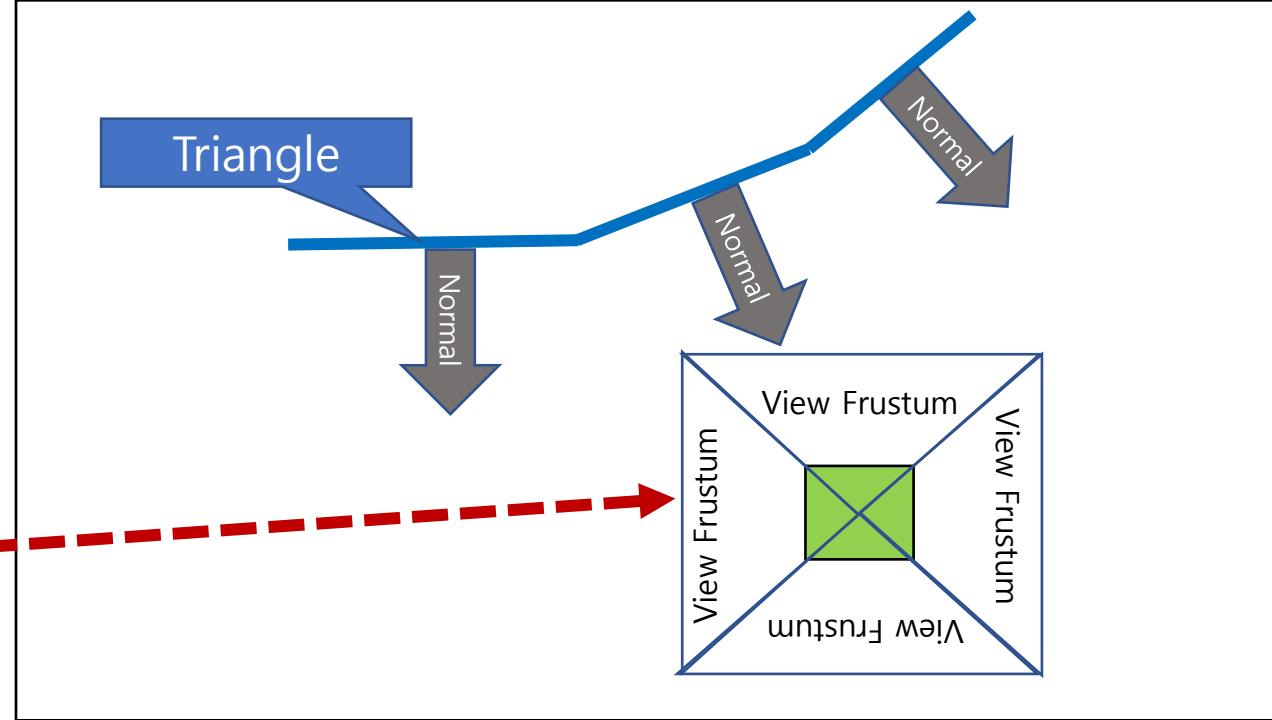
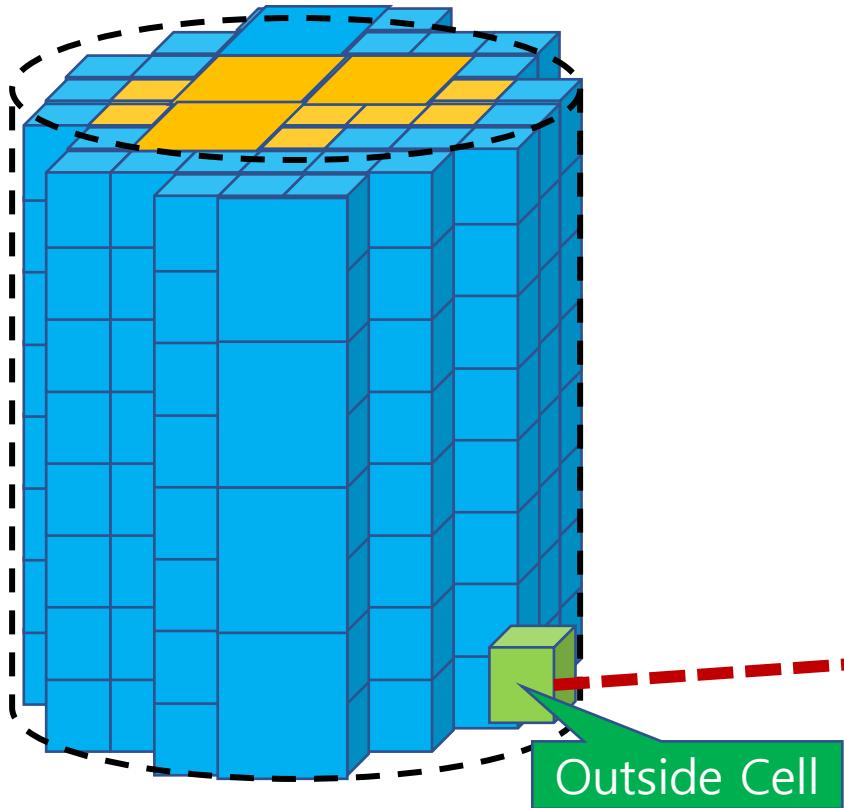
6방향 렌더링한 결과



# Outside Cell일 경우

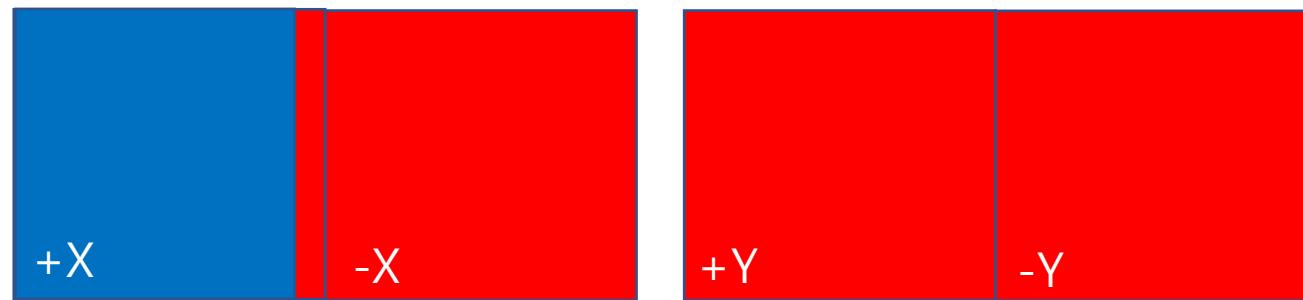


6방향 렌더링한 결과

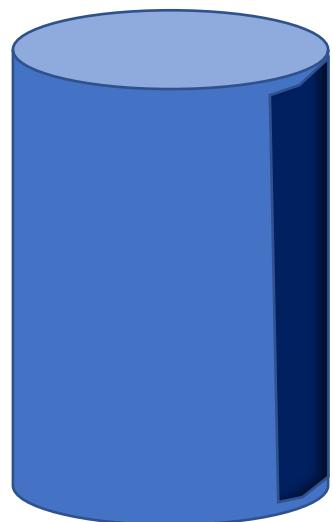


# Inside Cell 판별 기준.

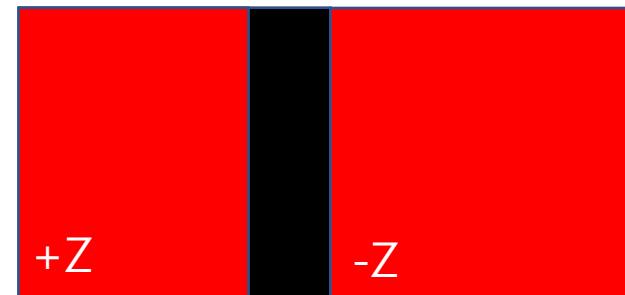
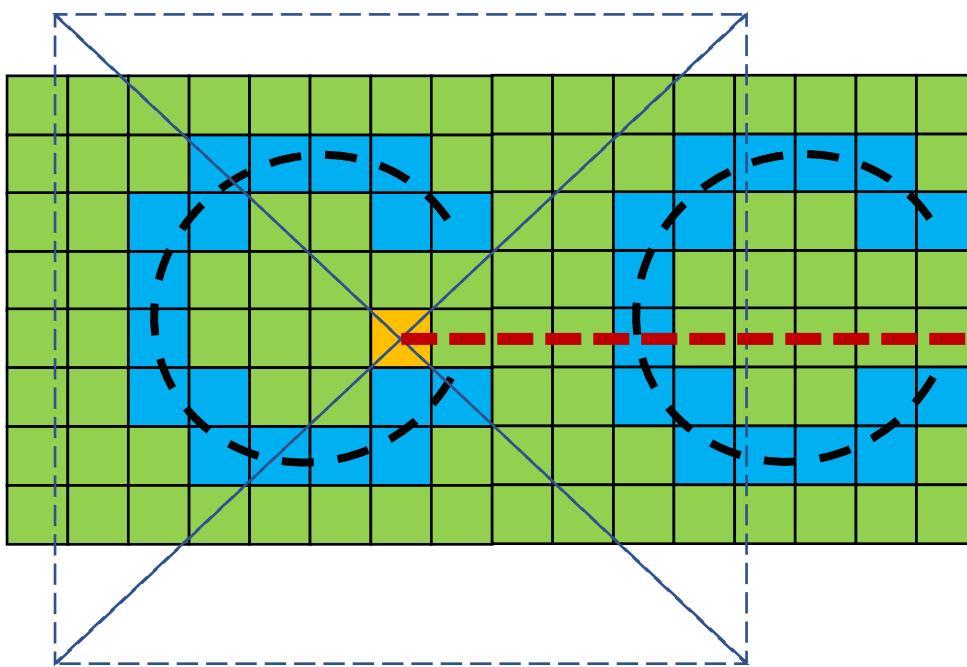
- 픽셀 개수가 일정 개수 미만(ex: 4개)면 무효처리.
- Inside의 경우 : 적어도 6면중 4면 이상에서 red pixel을 발견할 것.
- Outside의 경우 : red pixel을 하나도 발견하지 못할것.
- Inside Cell을 그대로 voxel 배열에 맵핑하면 순수한 Solid Voxel 이 된다. Occlusion Culling을 위해 사용할 경우 순수 Solid voxel만 사용한다.
- Inside Cell + Surface Cell을 합치면 삼각형과 교차하는 영역까지 포함한 Voxel 데이터가 된다.



옆이 뚫린 경우?



Top View

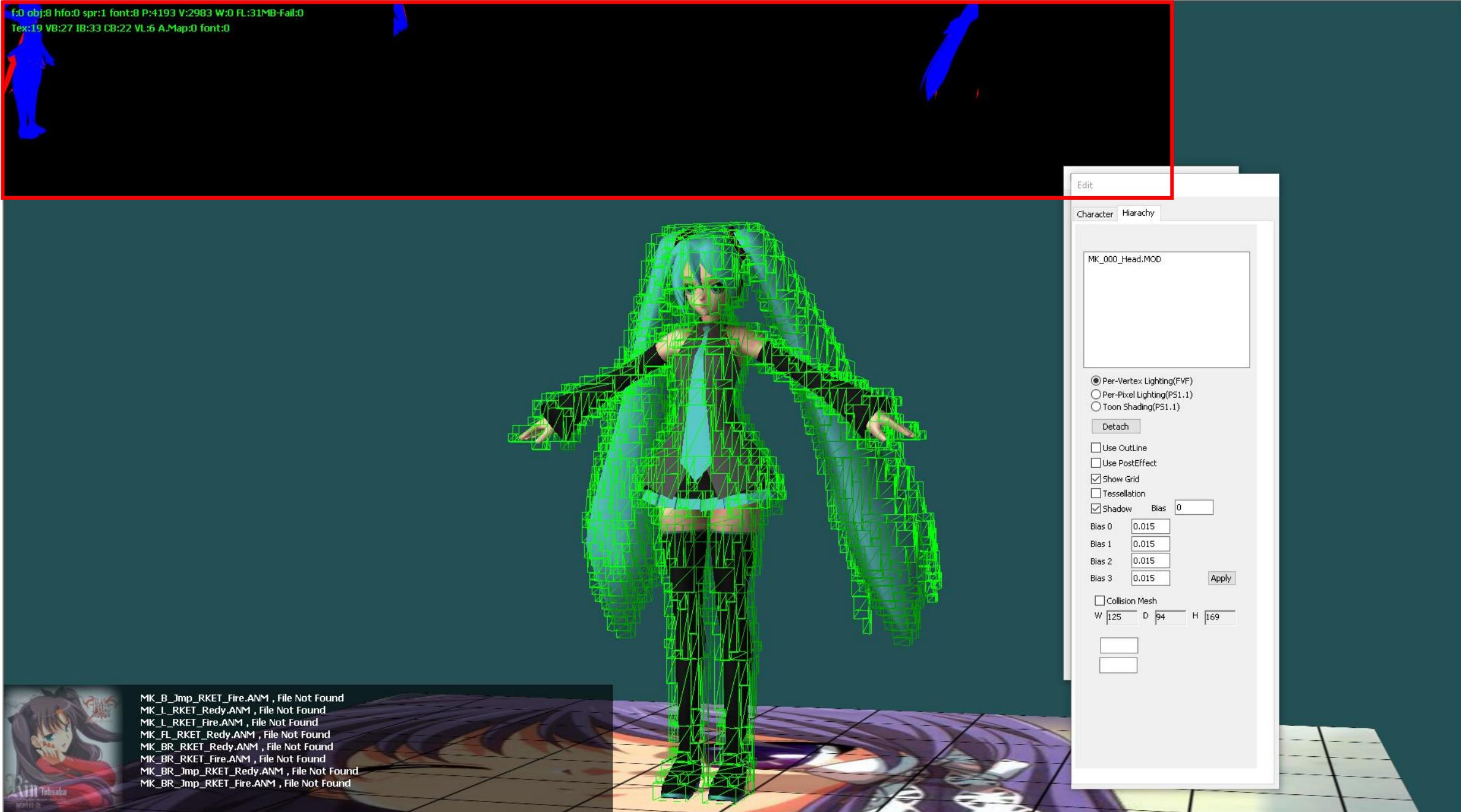


Red faces >  
threshold(ex:4) ?  
-> OK

Inside Cell!



f0 obj:8 hfo:0 spr:1 font:8 P:4193 V:2983 W:0 FL:31MB-Fail:0  
Tex:19 VB:27 IB:33 CB:22 VL:6 A.Map:0 font:0

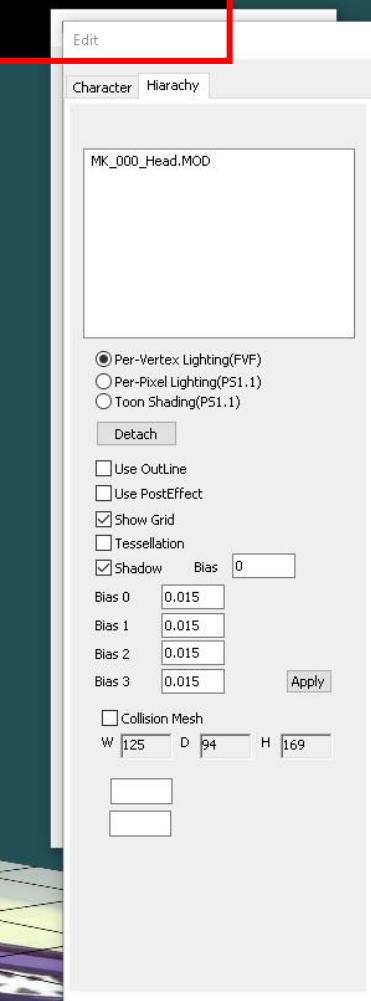
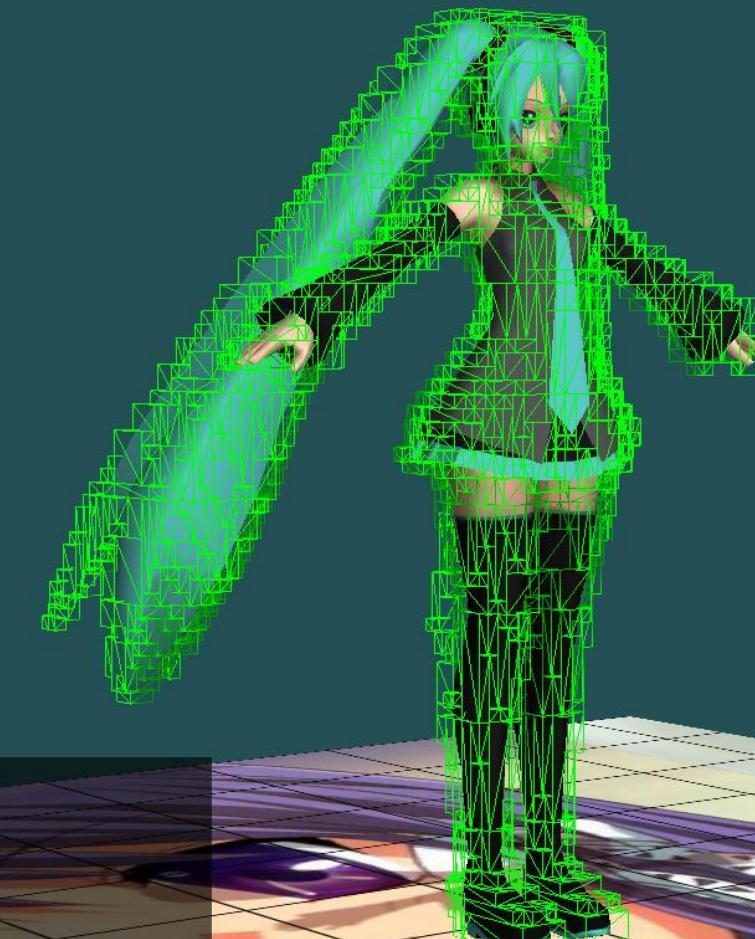




f:23 obj:8 hfo:0 spr:1 font:8 P:4193 V:2983 W:0 FL:31MB Fail:0  
Tex:19 VB:27 IB:33 CB:22 VL:6 A.Map:0 font:0

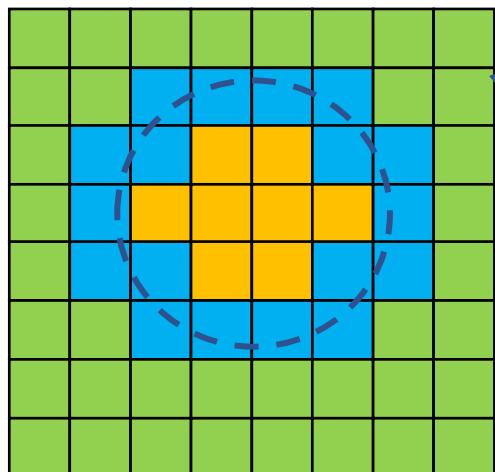


MK\_B\_Jmp\_RKET\_Fire.ANM , File Not Found  
MK\_L\_RKET\_Redy.ANM , File Not Found  
MK\_L\_RKET\_Fire.ANM , File Not Found  
MK\_FL\_RKET\_Redy.ANM , File Not Found  
MK\_BR\_RKET\_Redy.ANM , File Not Found  
MK\_BR\_RKET\_Fire.ANM , File Not Found  
MK\_BR\_Jmp\_RKET\_Redy.ANM , File Not Found  
MK\_BR\_Jmp\_RKET\_Fire.ANM , File Not Found

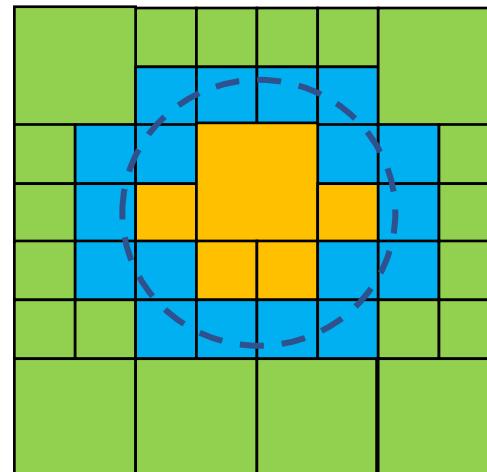


# 균일하게 분할하지 않고 Octree로 공간을 나누는 이유

- 필요 이상으로 조밀하게 공간을 나눌 경우 visibility 검사의 비용이 크게 증가한다.
- 삼각형과 교차하지 않는 최대한 큰 육면체(pure cell)가 필요하다.
- 이 최대한 큰 육면체(pure cell)들을 얻기 위해 Octree를 만든다.



Pure cell 42개  
렌더링할 cell이 42개



Pure cell 25개  
レン더링할 cell이 25개

# 성능개선

그대로는 느려서 못쓴다. 성능 개선이 반드시 필요하다.

# 성능 개선의 여지

- 6면을 매번 따로 렌더링할 필요가 있는가? 한번에 6면을 렌더링 할 수 없는가?
- GPU메모리의 프레임버퍼에 lock을 걸어서 컬러값을 읽어오는 건 매우 느리다. 더 빠른 방법은 없나?
- 월드 지형을 voxelize하는 경우 아주 먼 거리의 삼각형들까지 렌더링 할 필요가 없다. 즉 적당히 공간을 분할해서 근접한 삼각형들만 렌더링하면 훨씬 빠를 것이다.
- GPU에 막대한 부하가 걸릴것 같지만 실제로는 그 반대다. 그래픽 API 호출에 상당한 CPU시간을 소비하고 그 동안 GPU는 평평 논다. GPU를 꽉 채워서 사용할 수 없는가?

# 6면을 한번에 렌더링 하기

- Cell당 Draw Call을 6번 호출하는 대신 6개분의 리소스를 전달하고 Draw call을 한번으로 줄인다.
- Shader에 6개의 matrix와 6칸짜리 Texture Array를 전달한다.
- Geometry Shader에서 6개의 matrix로 transform한다.
- Pixel Shader에서 각각의 입력 스트림을 Texture Array에 그린다.

# Geometry Shader

```
■ cbuffer ConstantBufferCubemap : register(b4)
{
    matrix      matCubeViewList[6];
    matrix      matCubeProjList[6];
    matrix      matCubeViewProjList[6];
    float4     CamEyePos;      // x,y,z = eyepos , w = 1/far
};
```

자료구조 선언

```
■ struct PS_CUBEMAP_INPUT
{
    float4 Pos : SV_POSITION;      // Projection coord
    float4 Diffuse : COLOR0;
    uint    RTIndex : SV_RenderTargetArrayIndex;
};
```

```
[maxvertexcount(18)]
void gsCubeMapFrontBack(triangle GS_INPUT input[3], inout TriangleStream<PS_CUBEMAP_INPUT> CubeMapStream)
{
    float    rcp_far = CamEyePos.w;

    float4  colorFront = float4(0, 0, 1, 1);
    float4  colorBack = float4(1, 0, 0, 1);

    for (int f = 0; f < 6; ++f)
    {
        uint3  index = float3(0, 1, 2);
        float4  diffuseColor = colorFront;
        bool    flip = IsBackFace((float3)input[0].Pos, (float3)input[1].Pos, (float3)input[2].Pos, (float3)CamEyePos);
        if (flip)
        {
            index = uint3(0, 2, 1);
            diffuseColor = colorBack;
        }

        PS_CUBEMAP_INPUT      output[3];
        for (uint i = 0; i < 3; i++)
        {

            float4  Pos = mul(input[i].Pos, matCubeViewProjList[f]);

            output[i].RTIndex = f;
            output[i].Diffuse = diffuseColor;
            output[i].Pos = Pos;
        }
        CubeMapStream.Append(output[index.x]);
        CubeMapStream.Append(output[index.y]);
        CubeMapStream.Append(output[index.z]);
        CubeMapStream.RestartStrip();
    }
}
```

## Geometry Shader

# 파랑/빨강 픽셀의 개수를 빠르게 얻어오기.

- Lock(GPU메모리를 시스템 메모리에 맵핑) 걸고 GPU Memory에 있는 Texture Array 내용을 System Memory 카피하는 작업은 느린다.
- 대신 Compute Shader를 사용해서 픽셀 개수만 얻어온다.

# Compute Shader

```
#define MAX_COUNT_PIXEL_THREAD_NUM_X    16
#define MAX_COUNT_PIXEL_THREAD_NUM_Y    16

cbuffer CONSTANT_BUFFER_PIXEL_COUNT : register(b0)
{
    uint      Width;
    uint      Height;
    uint      Reserved0;
    uint      Reserved1;
};

struct CUBEMAP_PIXEL_COUNT
{
    uint      RedPixels;
    uint      BluePixels;
    uint      Reserved0;
    uint      Reserved1;
};

Texture2DArray    cubeMap           : register(t0);
RWStructuredBuffer<CUBEMAP_PIXEL_COUNT> BufferOut : register(u0);
SamplerState PointSampler : register(s0);

groupshared uint g_RedPixelCount[6];
groupshared uint g_BluePixelCount[6];
```

자료구조 선언

# Compute Shader

Shared memory와 global memory의  
변수 초기화

```
[numthreads(MAX_COUNT_PIXEL_THREAD_NUM_X, MAX_COUNT_PIXEL_THREAD_NUM_Y, 1)]
void csCountPixel(uint3 GroupId           : SV_GroupID,
                  uint3 GroupThreadID : SV_GroupThreadID,
                  uint3 DispatchThreadId : SV_DispatchThreadID,
                  uint GroupIndex : SV_GroupIndex)
{
    uint     sx = DispatchThreadId.x;
    uint     sy = DispatchThreadId.y;
    uint     f = GroupId.z;

    uint     oldValue;
    if (DispatchThreadId.x == 0 & DispatchThreadId.y == 0)
    {
        BufferOut[f].BluePixels = 0;
        BufferOut[f].RedPixels = 0;
    }

    if (GroupThreadID.x == 0 && GroupThreadID.y == 0)
    {
        g_RedPixelCount[f] = 0;
        g_BluePixelCount[f] = 0;
    }
    GroupMemoryBarrierWithGroupSync();
    DeviceMemoryBarrierWithGroupSync();
```

# Compute Shader

```
    uint      oldRedPixelCount, oldBluePixelCount;
    if (sx < Width && sy < Height)
    {
        uint4  nCoords = uint4(sx, sy, f, 0);
        float4 color = cubeMap.Load(nCoords);

        uint      r = (color.r > 0.5);
        uint      b = (color.b > 0.5);

        InterlockedAdd(g_RedPixelCount[f], r, oldValue);
        InterlockedAdd(g_BluePixelCount[f], b, oldValue);
    }

    GroupMemoryBarrierWithGroupSync();

    if (GroupThreadID.x == 0 && GroupThreadID.y == 0)
    {

        InterlockedAdd(BufferOut[f].RedPixels, g_RedPixelCount[f], oldValue);
        InterlockedAdd(BufferOut[f].BluePixels, g_BluePixelCount[f], oldValue);
    }
}
```

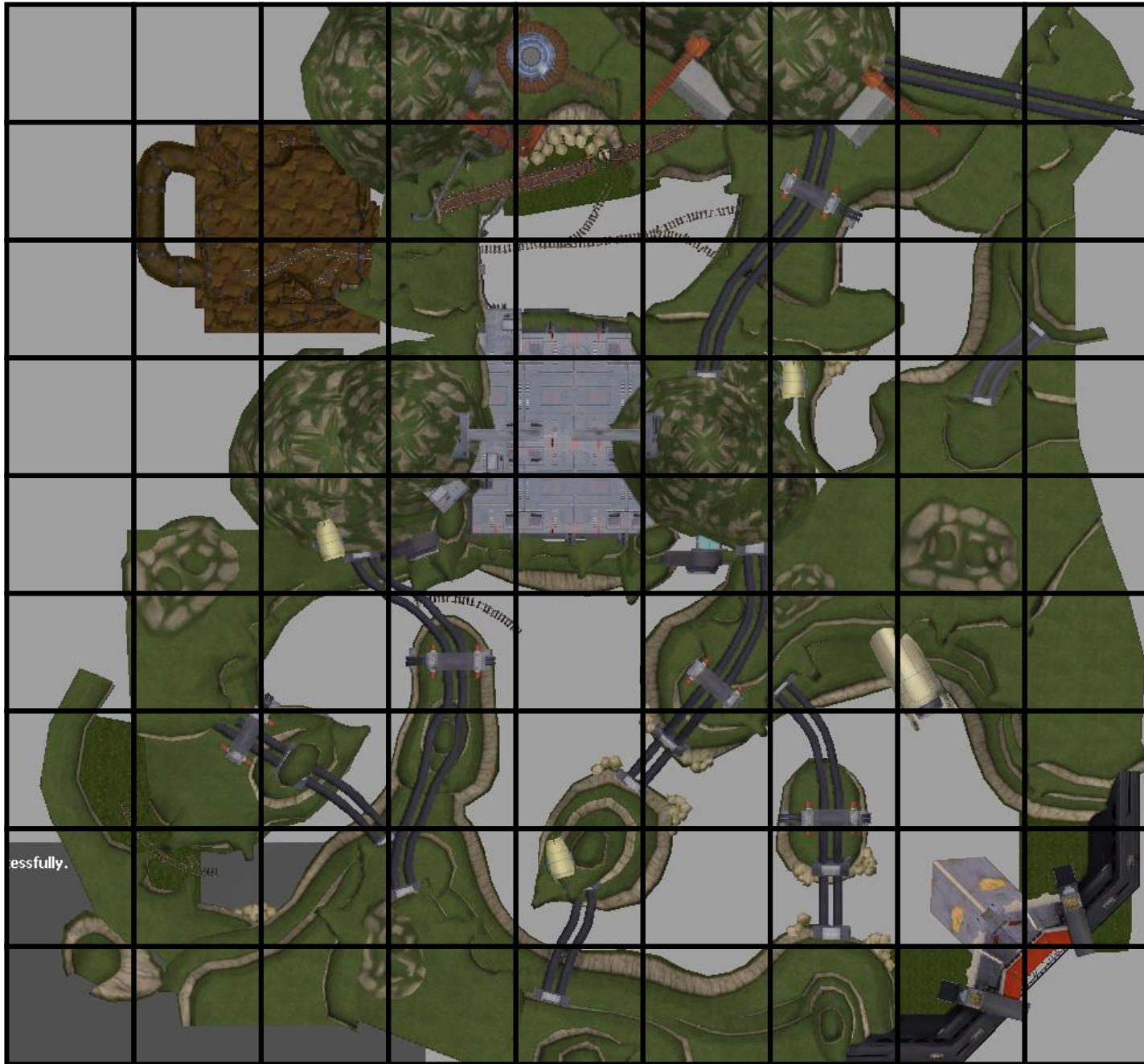
GPU Thread별로 프레임버퍼의 컬러값을 카운팅.  
thread group안에서 먼저 카운트 한 후 global  
memory(Device Memory)의 변수를 업데이트.

# 멀리 떨어진 삼각형 매시 culling

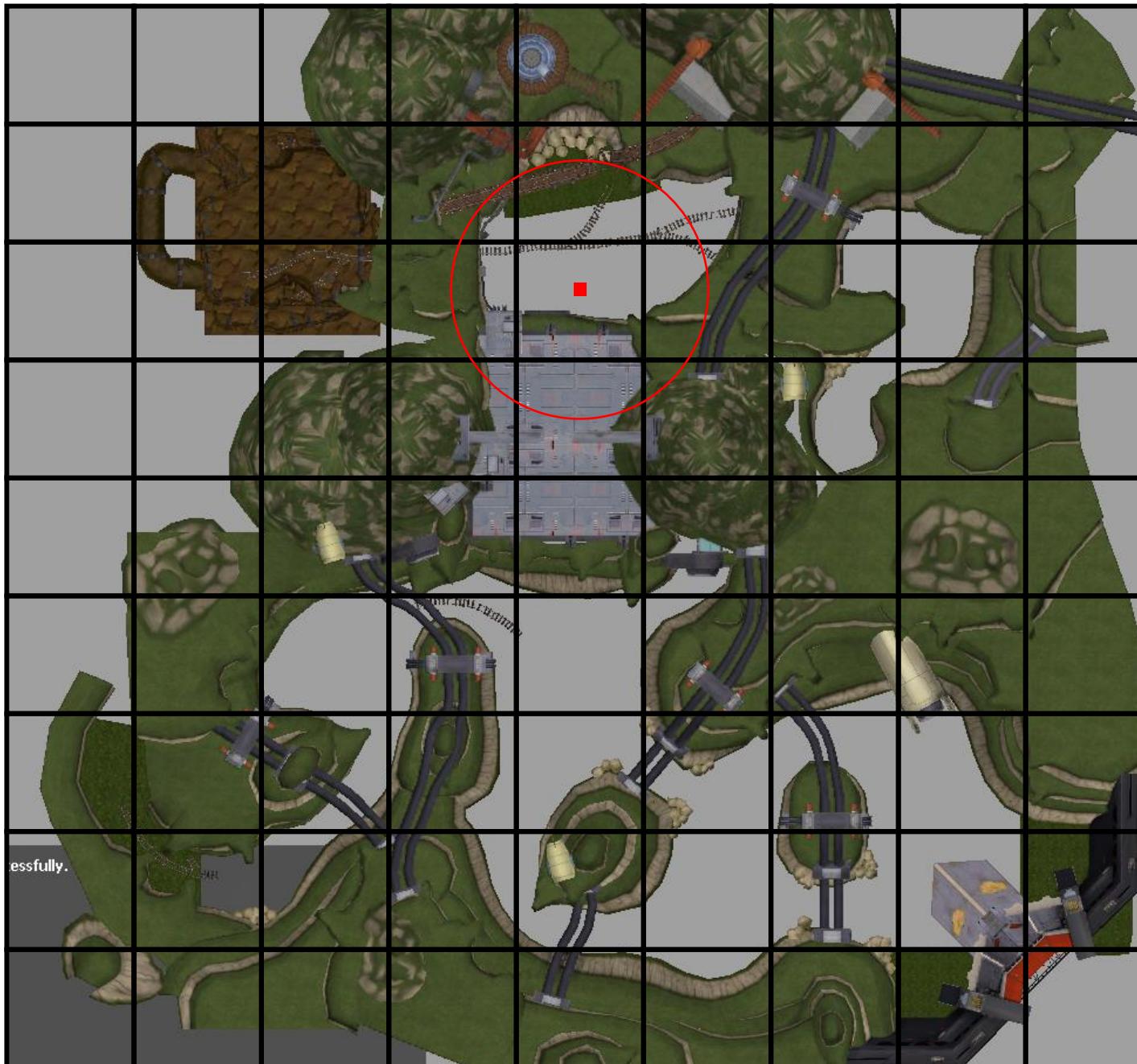
- 너무 멀리 떨어진 삼각형에 대해서 visibility 검사를 할 필요는 없다.
- Pure cell을 만들기 위한 Octree와는 별도로 적당히 공간을 분할 한다(Grid, Octree, KD-Tree,BSP Tree...) .
- 삼각형 매시를 잘라서 분할된 leaf에 담는다.
- Cell기준으로 삼각형 매시를 렌더링할때 멀리 떨어진 오브젝트 (삼각형매시)는 렌더링 파이프라인에서 제외한다.
- 최대 벽 두께를 감안해서 최대 렌더링 범위를 설정한다.



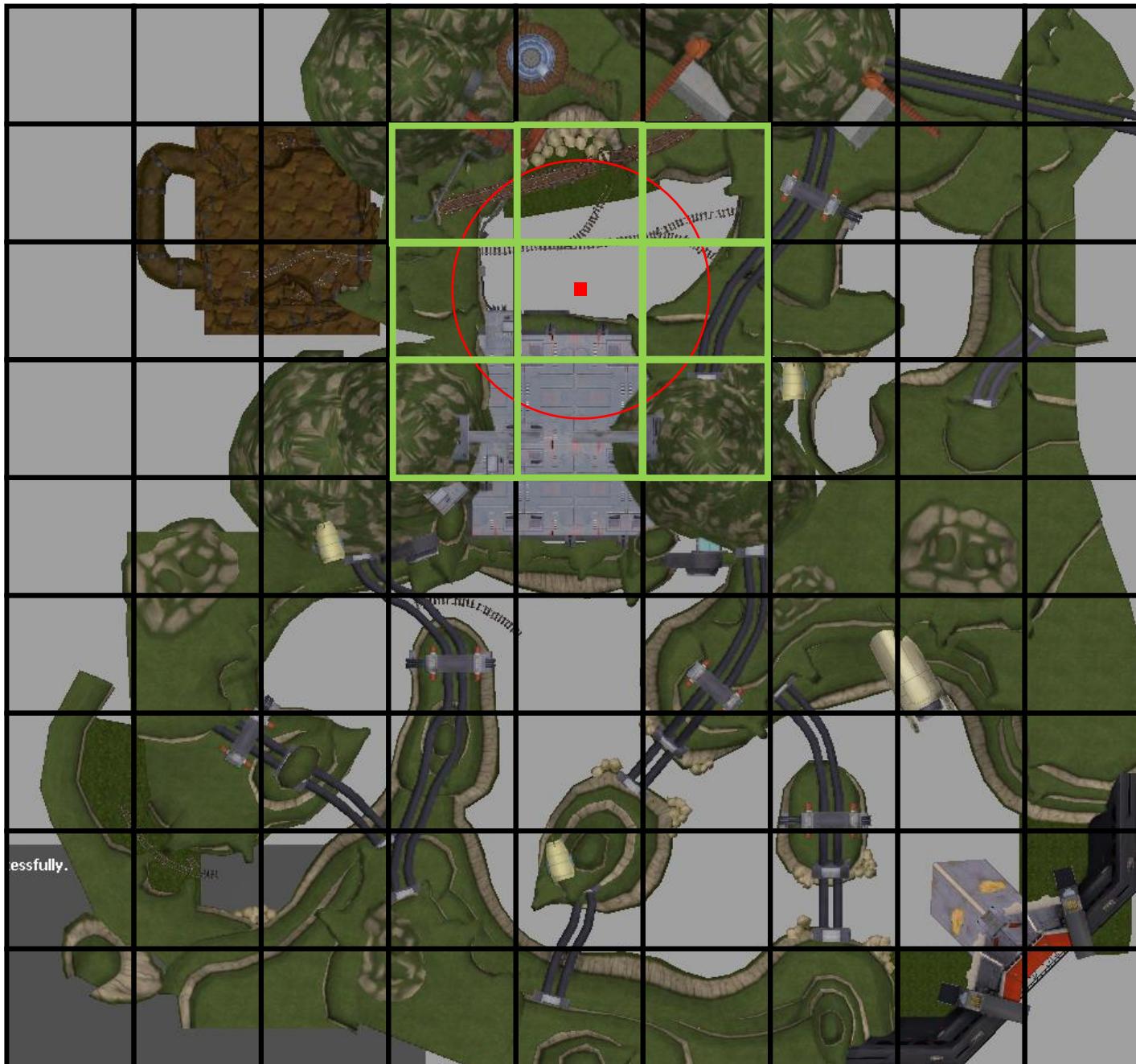
이 정도 크기의 cell의 visibility 테스트를 위해서 월드 전체의 삼각형 매시를 모두 그려야 할 필요가 있을까?



1. 2D배열, KD-Tree, BSP Tree, Quad Tree, Octree 뭐든 좋으니 공간을 분할한다.
2. Leaf에 삼각형 매시를 나눠 담는다.



1. Cell에 대해 visibility 테스트를 할 때 최대 범위를 정한다.
2. 최대 범위
  - 최소값 = cell을 감싸는 구의 반지름
  - 최대값 = 최소값 + 월드상에서 예상되는 가장 두꺼운 벽 두께.

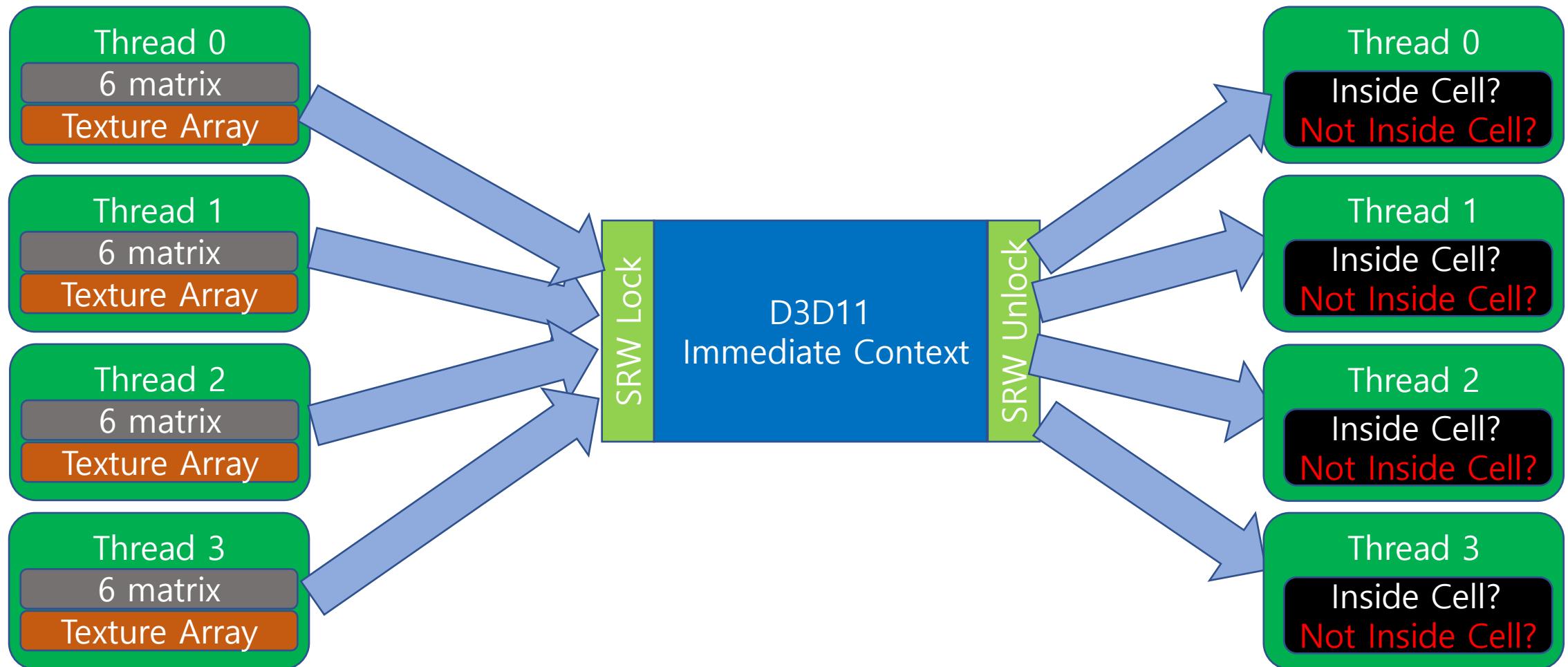


1. Cell의 위치와 최대범위로 걸치는 삼각형 매시를 탐색.
2. 찾아낸 삼각형 매시들로 visibility 테스트.

# 멀티 스레드 적용해서 GPU 사용률 높이기

- 그래픽 API를 호출할 때까지 CPU 시간이 꽤 소모된다.
- 그래픽 API를 호출하고 나서도 실제로 GPU에 렌더링 커맨드가 전달될 때까지 CPU 시간이 꽤 소모된다.
- 이 동안 GPU는 그냥 논다.
- CPU가 렌더링을 준비하는 중에도 GPU를 쉴 틈 없이 사용해야 한다. 그래서 멀티스레드로 렌더링을 한다.

# Multi threading in D3D11

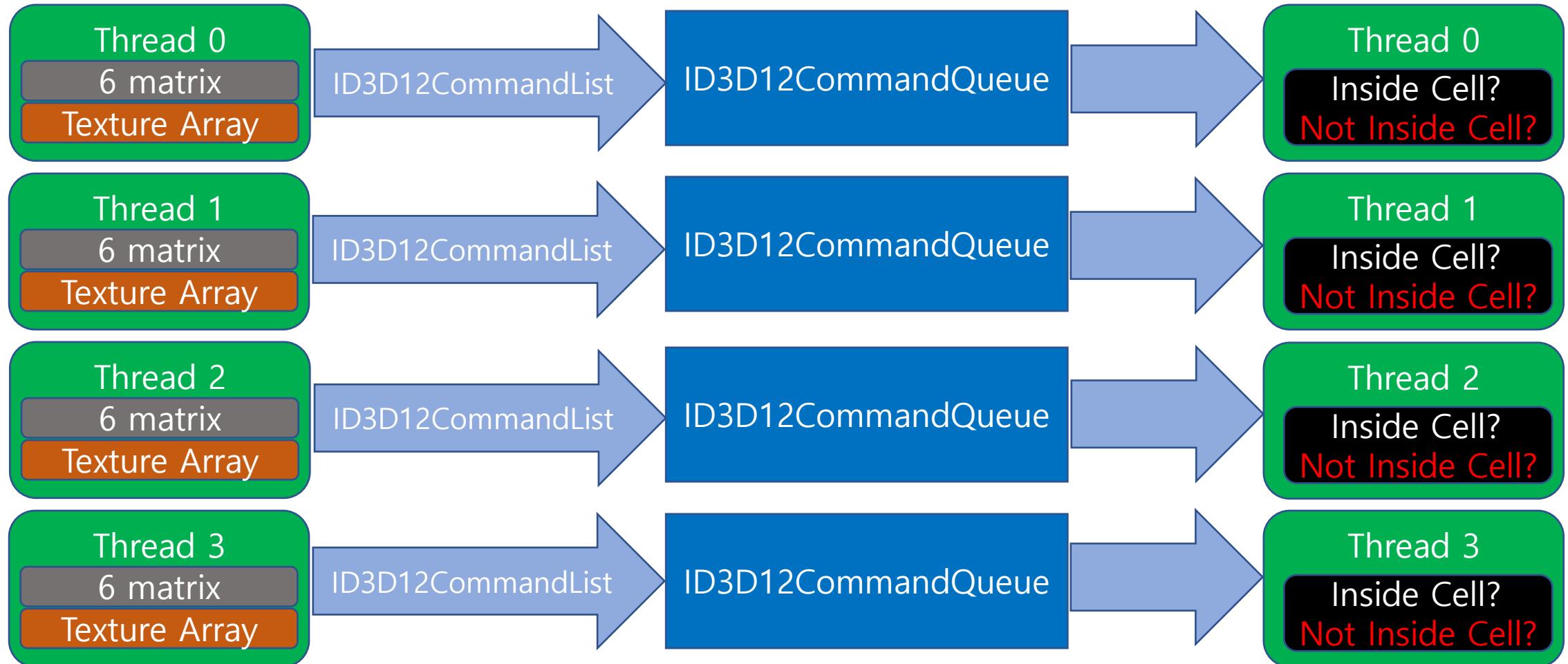


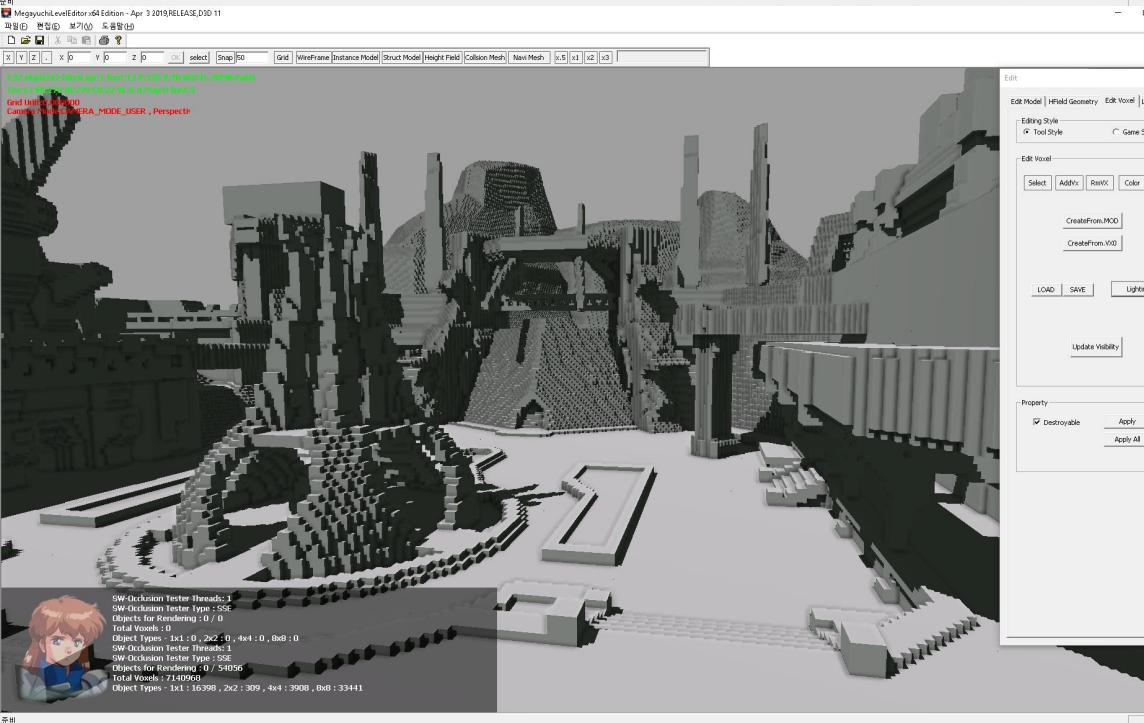
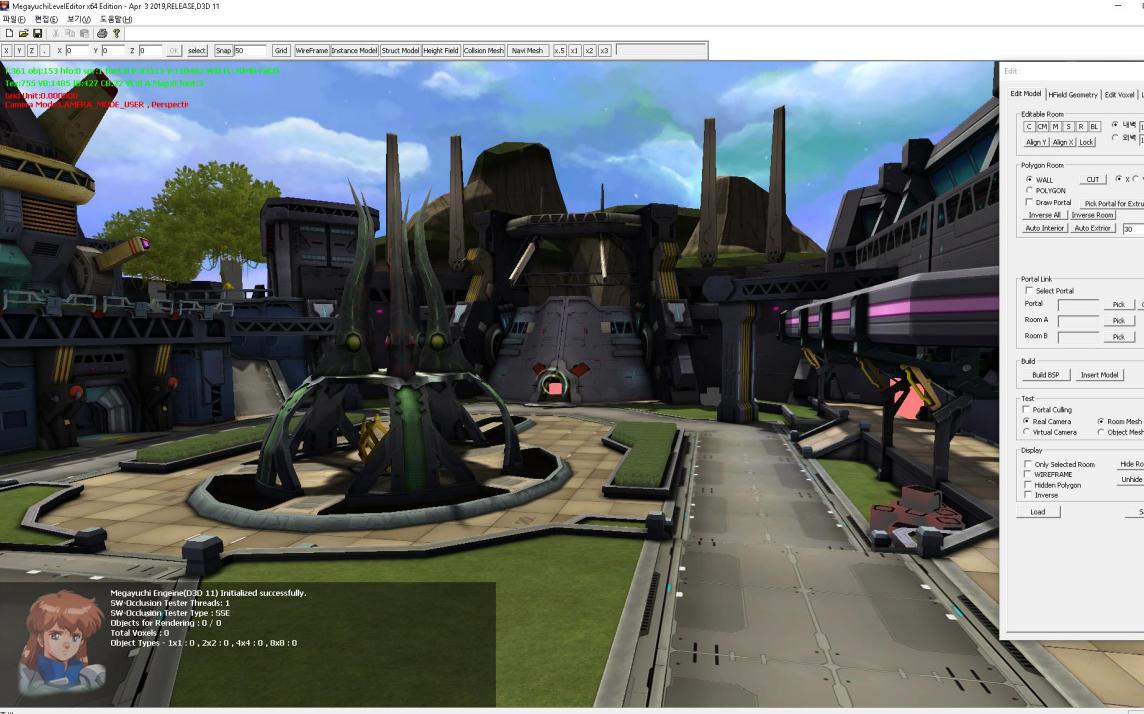
DirectX 11에서 Immediate Context를 사용할 때 Lock을 건다.

# Lock(동기화 객체)없이 멀티 스레드 적용하기.

- DirectX12는 완전히 thread safe하다.
- ID3D11DeviceContext에 해당하는 것이 ID3D12CommandQueue다.
- 워커 스레드 개수만큼 ID3D12CommandQueue를 생성한다.
- 워커 스레드 개수만큼 Texture Array와 Constant Buffer를 준비 한다.
- 각각의 스레드는 각기 다른 Command Queue에 렌더링 커맨드를 전송한다.

# Multi threading in D3D12





# 성능비교

## I7 8700K , nvidia GTX970

1. D3D12 – 12 threads
  - 503704ms , GPU 점유율 : 99% - 100%
2. D3D11 - 12 threads
  - 541170ms , GPU 점유율 : 80% – 90%
3. D3D11 - 1 thread
  - 544856ms , GPU 점유율 : 75% -85%
4. D3D12 - 1 thread
  - 661013ms , GPU 점유율 : 65%-75%

## I7 8700K , nvidia GTX1160

1. D3D12 – 12 threads
  - 464443ms , GPU 점유율 : 99% - 100%



파일(F) 편집(E) 보기(V) 도움말(H)



X Y Z . X 0 Y 0 Z 0 OK select Snap 50 Grid WireFrame Instance Model Struct Model Height Field Collision Mesh Navi Mesh x.5 x1 x2 x3

f6 obj:10801 hfo:0 spr:1 font:8 P:120 V:78 W:0 FL:30MB:Fail:0

Tex:13 VB:278 IB:279 CB:22 VL:6 A.Map:0 font:3

Grid Unit:0.000000

Camera Mode:CAMERA\_MODE\_USER , Perspective



파일(F) 편집(E) 보기(V) 도움말(H)



X Y Z . X 0 Y 0 Z 0 OK select Snap 50 Grid WireFrame Instance Model Struct Model Height Field Collision Mesh Navi Mesh x.5 x1 x2 x3

f1.obj:20429 Mod: spr:1 font:12 P:120 V:20 W:0 F:130M8-Fa00

Dec:13 VB:278 IB:279 CB:22 VL:6 A:Map01 font:3

Grid Unit:0.000000

Camera Mode:CAMERA\_MODE\_USER , Perspective

Edit Model HField Geometry Edit Voxel Layer



파일(F) 편집(E) 보기(V) 도움말(H)



X Y Z . X 0 Y 0 Z 0 OK select Snap 50 Grid WireFrame Instance Model Struct Model Height Field Collision Mesh Navi Mesh x.5 x1 x2 x3

t21.obj(20987 hfds:0 spr:1 font:0 P:120 V:20 W:0 F:13098-Bad:0

Dec:13 VR:279 DR:279 CB:32 VL:6 A:Map01 font:3

Grid Unit:0.000000

Camera Mode:CAMERA\_MODE\_USER , Perspective



Megayuchi Engine(D3D 11) Initialized successfully.

SW-Occlusion Tester Threads: 12

SW-Occlusion Tester Type : SSE

Objects for Rendering : 0 / 69692

Total Voxels : 7948459

Object Types - 1x1 : 19895 , 2x2 : 38 , 4x4 : 15482 , 8x8 : 34277

준비

Edit

Edit Model | HField Geometry | Edit Voxel | Layer

Editing Style  
 Tool Style  Game Style

Edit Voxel

Select AddVx RmVx Color

CreateFrom.MOD  
CreateFrom.VX0

LOAD SAVE Lighting

Update Visibility

Property

Destroyable

CAP

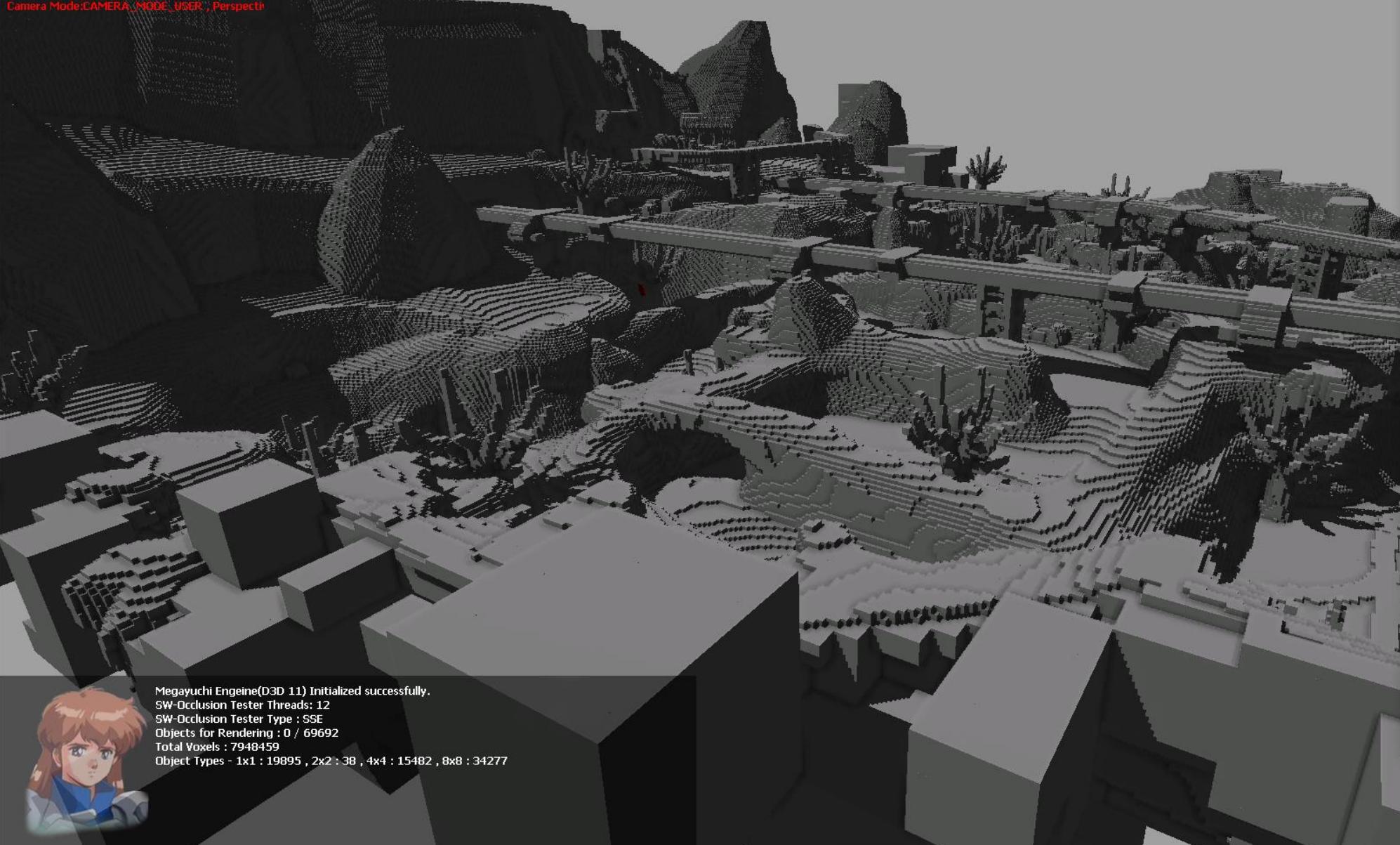
파일(F) 편집(E) 보기(V) 도움말(H)



X Y Z . X 0 Y 0 Z 0 OK select Snap 50 Grid WireFrame Instance Model Struct Model Height Field Collision Mesh Navi Mesh x.5 x1 x2 x3

15 obj:3654 mod:0 spr:1 font: Pez20 v:76 W:0 H:0 0.00MB 540.0  
Dec14 VB:229 IB:200 FB:22 VL:6 A:Map01 font:0

Grid Unit:0.000000  
Camera Mode:CAMERA\_MODE\_USER , Perspective



Edit

Edit Model | HField Geometry | Edit Voxel | Layer ▶ ▷

Editing Style  
 Tool Style       Game Style

Edit Voxel

Select AddVx RmVx Color

CreateFrom.MOD  
CreateFrom.VX0

LOAD SAVE Lighting

Update Visibility

---

Property

Destroyable       Apply  
 Apply All

파일(F) 편집(E) 보기(V) 도움말(H)



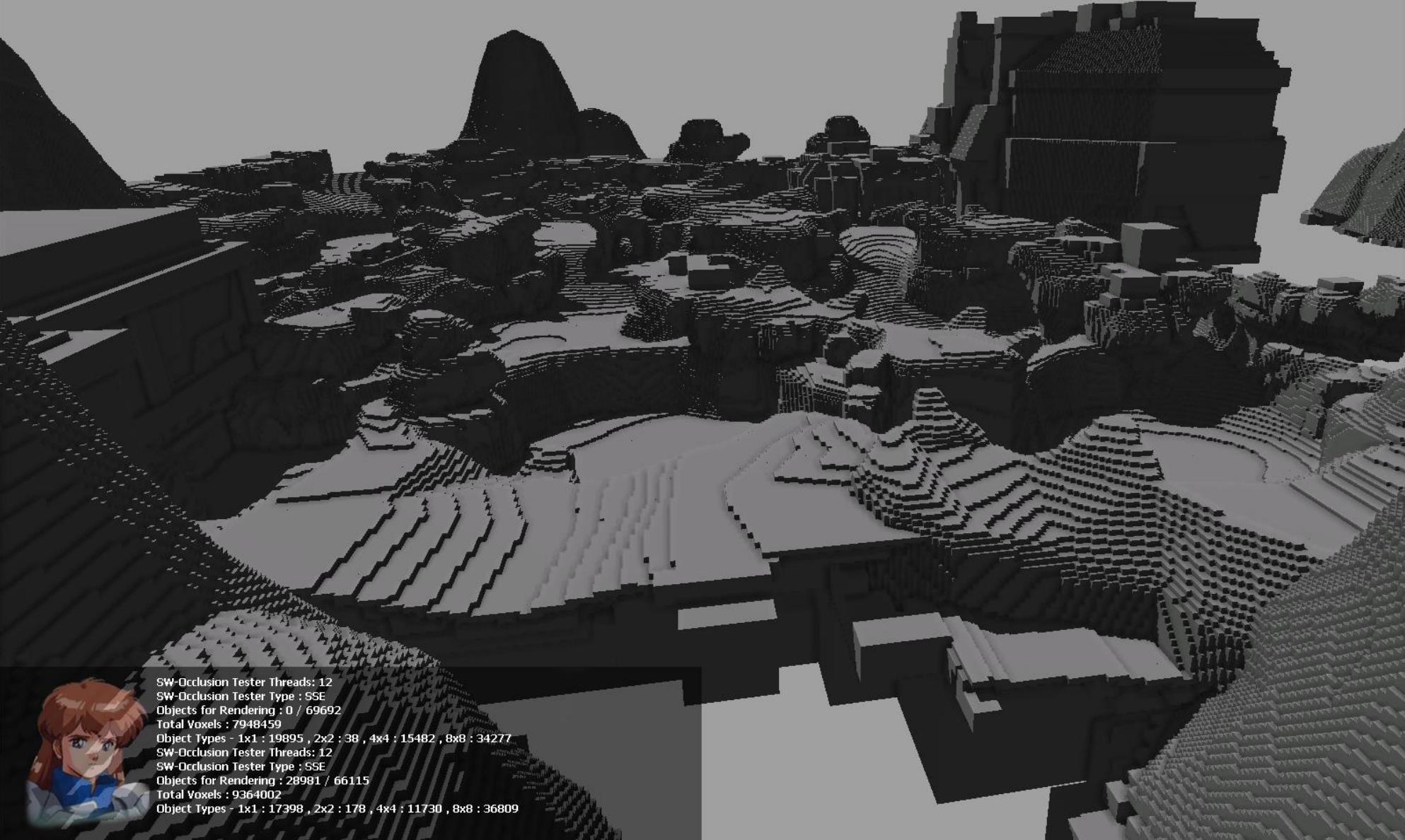
X Y Z . X 0 Y 0 Z 0 OK select Snap 50 Grid WireFrame Instance Model Struct Model Height Field Collision Mesh Navi Mesh x.5 x1 x2 x3

fbx obj:34206 mod spr:1 font:12 P:120 V:10 W:0 F:138M:5a:f0

Dec:13 V:0.279 IB:279 CB:22 VL:6 A:Map01 font:3

Grid Unit:0.000000

Camera Mode:CAMERA\_MODE\_USER , Perspective



SW-Occlusion Tester Threads: 12  
 SW-Occlusion Tester Type : SSE  
 Objects for Rendering : 0 / 69692  
 Total Voxels : 7948459  
 Object Types - 1x1 : 19895 , 2x2 : 38 , 4x4 : 15482 , 8x8 : 34277  
 SW-Occlusion Tester Threads: 12  
 SW-Occlusion Tester Type : SSE  
 Objects for Rendering : 28981 / 66115  
 Total Voxels : 9364002  
 Object Types - 1x1 : 17398 , 2x2 : 178 , 4x4 : 11730 , 8x8 : 36809

Edit

Edit Model | HField Geometry | Edit Voxel | Layer ▶

Editing Style  
 Tool Style  Game Style

Edit Voxel

Select AddVx RmVx Color

CreateFrom.MOD  
CreateFrom.VX0

LOAD SAVE Lighting

Update Visibility

Property

Destroyable Apply  
Apply All

## 기타 등등...

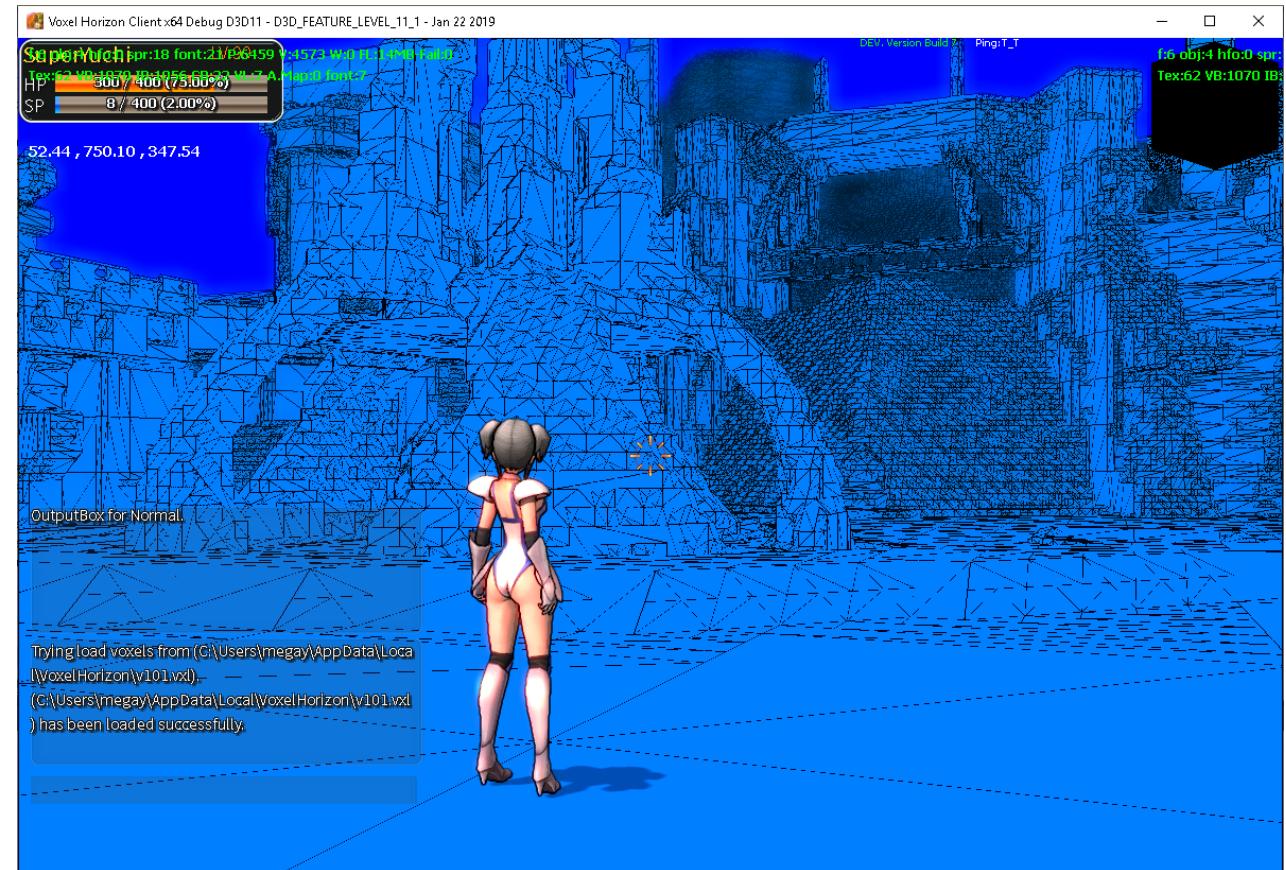
- 추가적인 성능 개선 아이디어?

D3D12의 Multiple GPU Support 사용

<https://developer.nvidia.com/explicit-multi-gpu-programming-directx-12>

# 기타 등등...

- Voxel Model -> Triangle Mesh?  
Marching Cube 알고리즘으로 간단하게 구현 가능. 속도도 빠르다.



# Reference

- <http://blog.wolfire.com/2009/11/Triangle-mesh-voxelization>
- [http://www-evasion.imag.fr/Membres/Franck.Hetroy/Teaching/ProjetsImagne/2005//haumont\\_warzee-jgt2002.pdf](http://www-evasion.imag.fr/Membres/Franck.Hetroy/Teaching/ProjetsImagne/2005//haumont_warzee-jgt2002.pdf)