나만의 게임엔진 개발하기

유영천

https://megayuchi.com

Tw:@dgtman

게임엔진 개요

게임 엔진?

- 게임을 빠르게 개발할 수 있게 해주는 기반 코드.

Megayuchi民이 생각하는 엔진

- 리소스를 로딩해서 렌더링하는 기능
- 사용자 입력에 의한 상호작용
- 편집 기능도 엔진에 포함시킨다.
- 게임과는 철저하게 분리한다.
- 이상적인 엔진 – 유니티 – 개인적으로는 싫지만

자체 개발 엔진

상용 엔진이 넘치기 전에는 엔진을 직접 개발할 능력이 없는 개발사는 아예 3D 게임을 만들 수 없었다. 그때는 엔진 프로그래머가 왕(사실은 아니었지만)과 같았다.
경영자들은 거만(자기들 눈에)해 보이는 엔진 프로그래머들을 필요악으로 여겼다.
상용엔진의 시대가 됐을때 경영자들은 기뻐했다. 이제 그 망할놈들의 목을 날릴 수 있어!

장점

- 로열티 안준다.
- 상용엔진이라도 안되는 기능은 안된다.
- 내 게임에서 중요한 기능은 내가 직접 만들어서 성능/품질을 높일 수 있다.
- 문제 해결에 남의 도움이 필요없다.
- 게임은 망해도 기술이 남는다!!!!!!

단점

- 새로 만들려면 오래 걸린다.
- 능력이 안되면 아예 못만든다.
- 능력이 안되면 문제를 해결할 수 없다.
- 특정 기능에 집중해도 그 특정 기능마저 상용엔진보다 못할 수 있다.

어떻게 만들까.

- 필요한 부품들을 하나하나 만들어간다
- 부품들을 최대한 재활용한다
- 개발이 진척될수록 흐르면 부품들간 연관성이 명확해진다.
- 그렇게 체계가 잡히다보면….Platform이라고 부를만한 소프트웨어 체계가 완성된다.

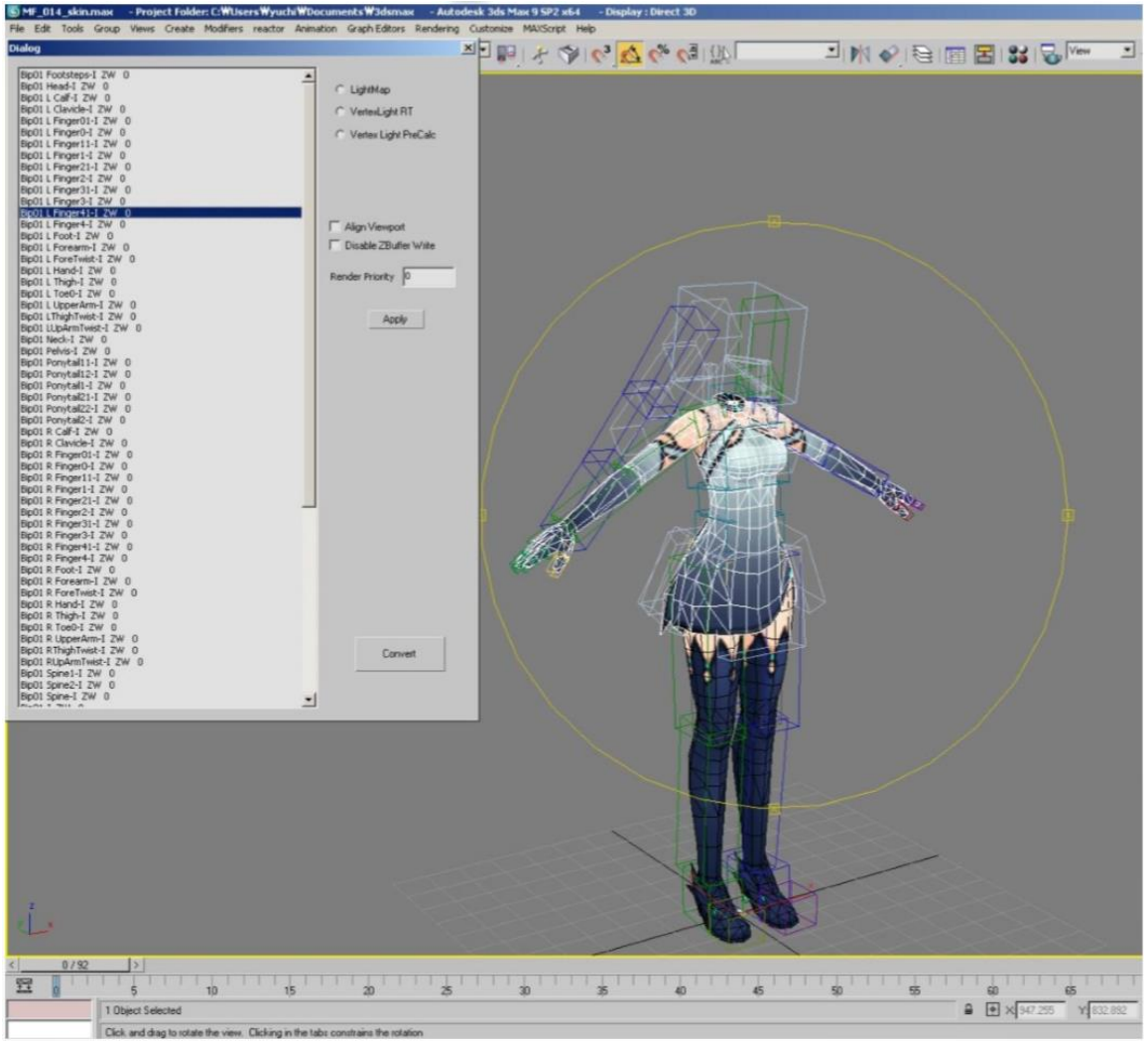
Client Side에서 시작 한다.

- 화면에 뭐가 보여야 진척이 된다.
- 캐릭터가 뛰어다니면 게임 다 만든것 같은 느낌을 받는다.

필요부품

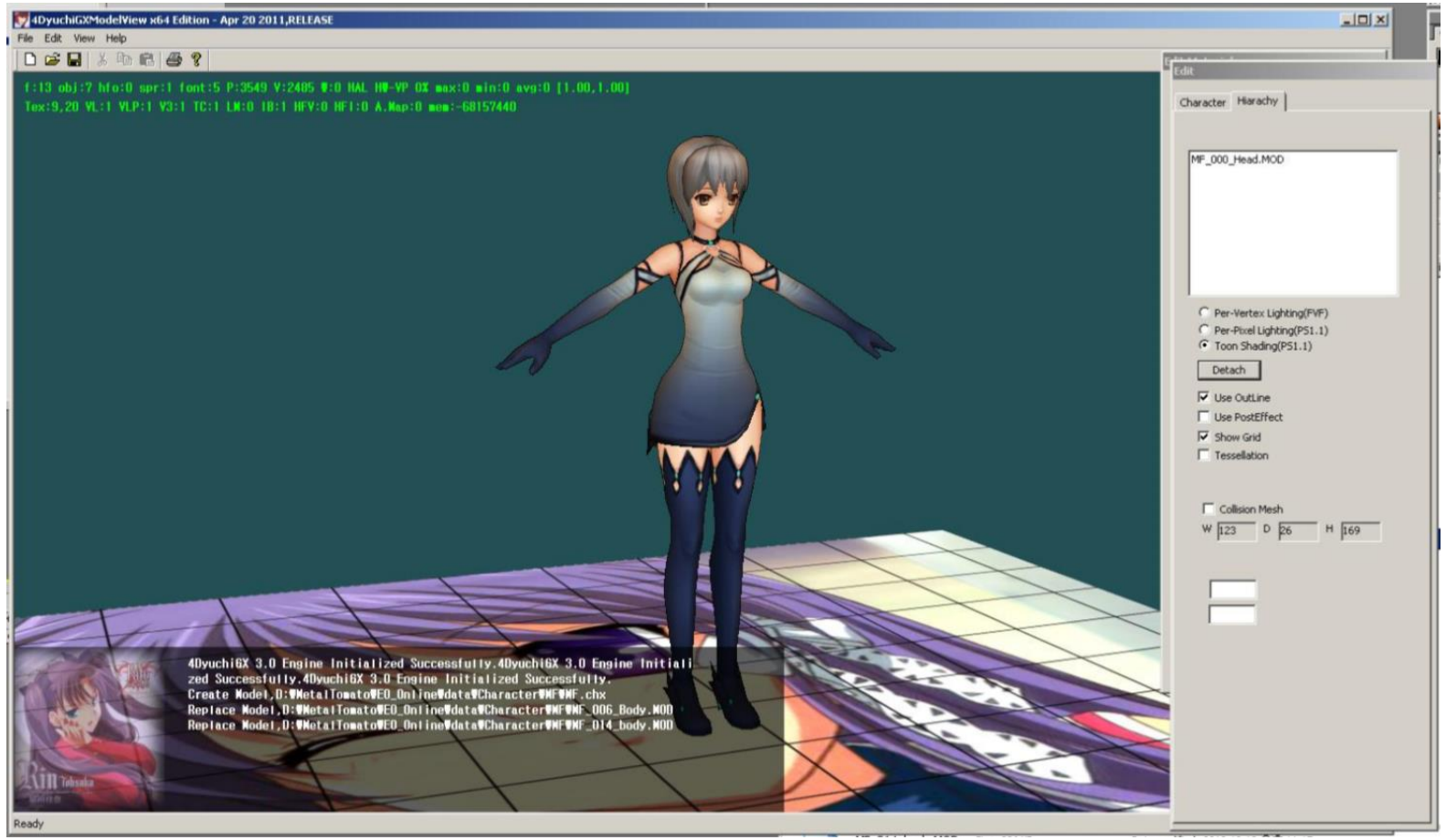
- 그래픽 툴(3dsmax등)으로부터 데이터 exporter
- 가장 기본적인 Renderer(3D엔진의 시작)
- Renderer를 이용한 Model Viewer
- Renderer를 이용한 최초의 Game Framework –> Game Client

Exporter



3ds max exporter

Model Viewer



세계를 자료구조로 구축하다 - Map

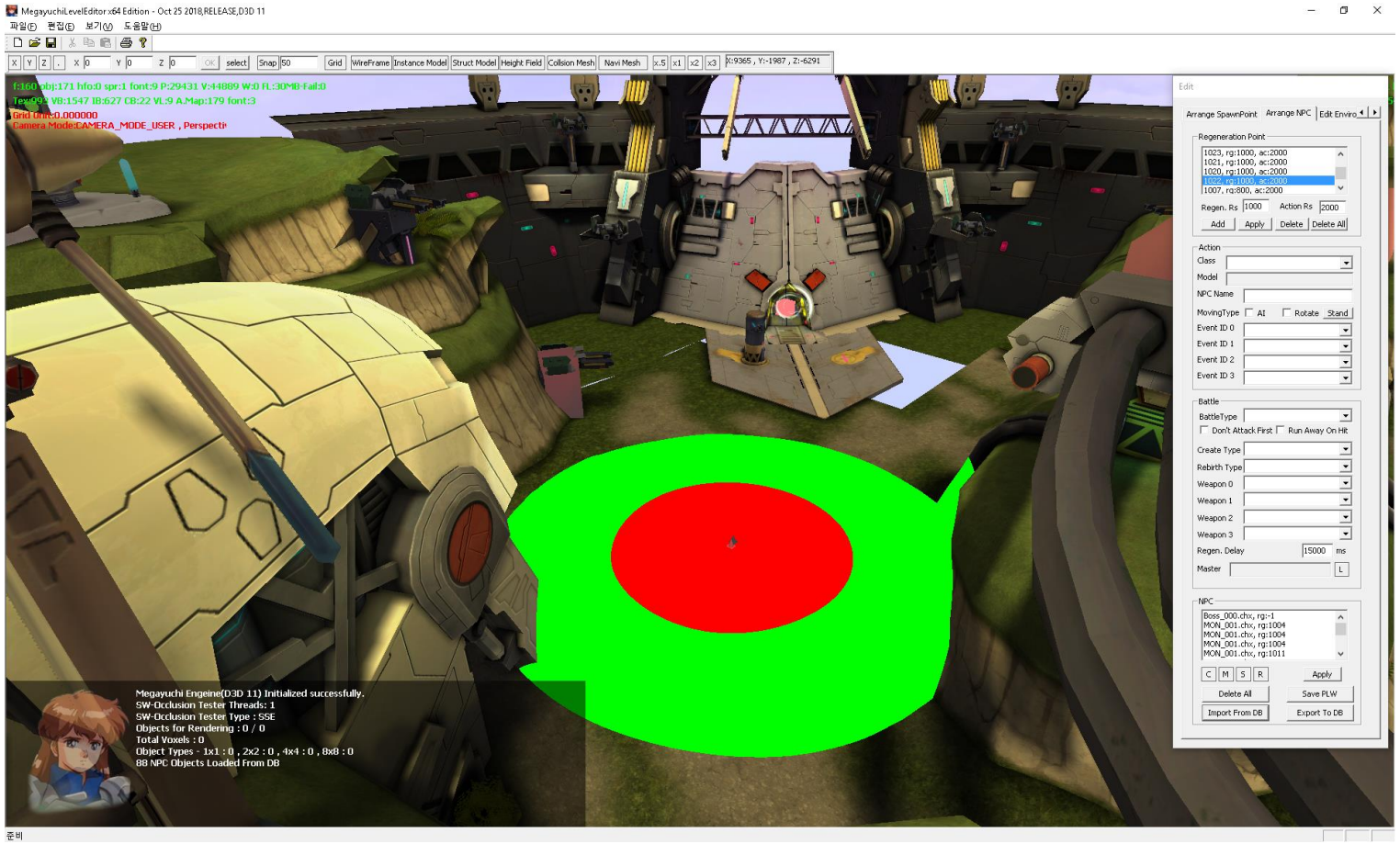
Map

- 그래픽적 관점으로는 배경이라 불린다.
- 게임 전체를 구성하는 자료구조
- 게임 전체적인 상태를 저장한다.

Map Tool

- 엔진과 함께 동시 개발.
- 맵툴 없는 엔진은 엔진이 아니다.
- 트리거, 인스턴스 오브젝트, NPC등을 배치한다.
- 그래픽적 요소를 편집할 수 있는 기능을 가진다.
- 클라이언트와 서버에서 필요한 자료구조를 생성한다.

Map Tool


MegayuchiLevelEditor:x64 Edition - Oct 25 2018,RELEASE,D3D 11
파일(F) 편집(E) 보기(V) 도움말(H)
Megayuchi Engeine(D3D 11) Initialized successfully.
SW-Occlusion Tester Threads: 1
SW-Occlusion Tester Type : SSE
Objects for Rendering : 0 / 0
Total Voxels : 0
Object Types - 1x1 : 0 , 2x2 : 0 , 4x4 : 0 , 8x8 : 0
88 NPC Objects Loaded From DB
Arrange SpawnPoint
Arrange NPC
Edit Enviro
Regeneration Point
Add
Apply
Delete
Delete All
Action
Class
Model
NPC Name
Event ID 0
Event ID 1
Event ID 2
Event ID 3
Battle
BattleType
Create Type
Rebirth Type
Weapon 0
Weapon 1
Weapon 2
Weapon 3
Regen. Delay
Master
NPC
Delete All
Save PLW
Import From DB
Export To DB
준비

최초의 Client

- Map Tool로 만들어낸 맵 데이터를 로드한다.
- 맵 위에 캐릭터 데이터를 인스턴싱해서 올린다.
- 유저의 입력을 받아서 캐릭터를 움직여준다.
- 되는건 많지 않지만 본격적인 개발이 시작된 것이다.

최초의 Client


EO_Client_01
67-Left,Up
68-Stop,Up
On Stop
69-Left,Up
70-Left,Up
71-Left,Up
72-Stop,Up
On Stop

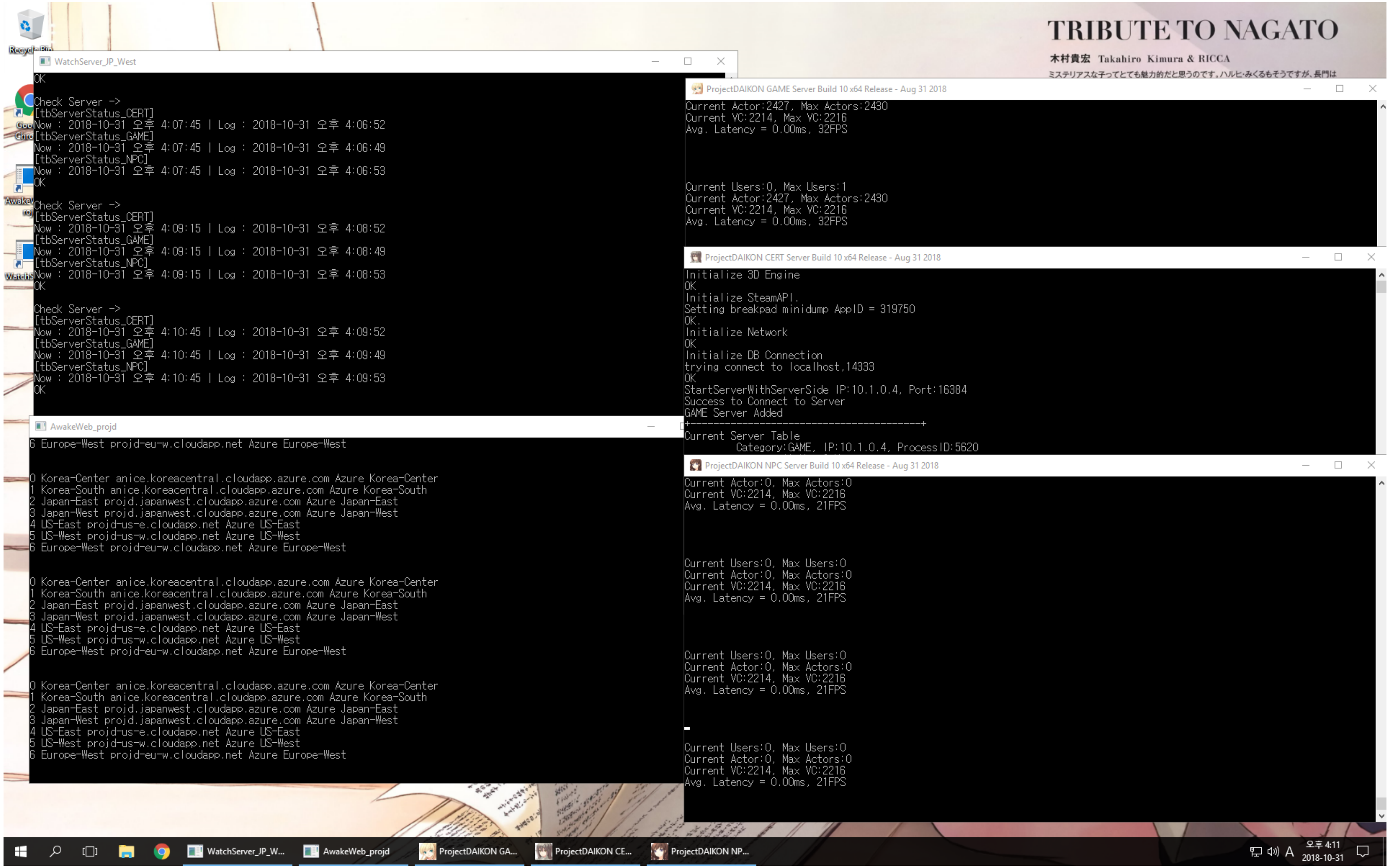
Game Engine

- 여기까지 오면 대충 엔진이라 부를 수 있다.
- 그러나 실제로는 훨씬 많은 기능이 필요하다.
- 공간 분할 (BSP Tree/PVS/Room-Portal/KD-Tree등)
- 오브젝트 검색 기능
- 충돌처리 기능
- 리소스 공유, 캐싱 기능

Server

- 이 모든 내용들은 서버에서 시뮬레이션 되어야한다.
- 클라이언트는 단지 터미널일 뿐이다. 입력을 받고 결과를 출력한다.
- 인증, 게임상에서 생성/변경된 데이터 저장
- 플레이어들의 위치 추적, 전투판정, 주기적인 이벤트 처리
- 엔진 잘 만들어놓으면 서버에서 공유가능.
  - 코딩량 줄어든다.
  - 안정성 검증.
  - 서버/클라이언트 동기화 50%는 먹고 들어감.

게임서버, NPC서버, 인증서버



필요 부품

- 네트워크 라이브러리
- DBMS
- DB미들웨어
- 게임엔진(클라이언트와의 엔진공유)

Network Library

- 일반적으로 Socket API사용. 당연히 TCP/IP
- 클라이언트와 서버측을 따로 만드는 경우가 많지만 하나 의 라이브러리로 양쪽을 커버하는 것이 좋다.
- 서버에서 사용할 때는 4000개 이상의 동시접속을 무난히 커버할 수 있어야한다.
- 1Gbps LAN을 꽉 채우는 정도의 부하에도 안정성을 보장 해야한다.

DBMS

- 데이터 저장, 복구, 백업을 위해 사용한다.
- 단순 쿼리문보다 Stored Procedure의 형태로 많이 사용 한다.
- MSSQL, MySQL등을 많이 사용한다.

DB Middleware

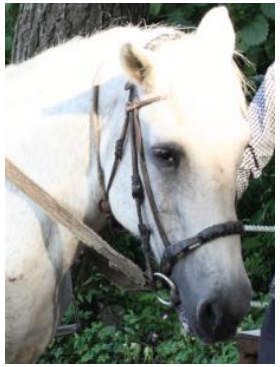
- DB는 상대적으로 응답시간이 느리고 처리시간을 정확히 예측할 수 없다.
- 게임서버가 DB의 처리를 무한정 기다릴 수 없으므로 비동기적으로 처리하고자 한다.
- 게임 특성에 맞는 DB처리가 필요할 때 미들웨어에 기능을 추가한다.
- 서버에서 DB에 접근할때, 혹은 게임관련 tool을 사용할때 편리하고 안전하게 DB에 접근할 수 있다.
- 보통 직접 개발한다.

서버와 클라이언트간의 부품 공유

- 클라이언트에서 사용하는 엔진을 서버에서도 사용한다.
- **자체 엔진 개발의 가장 큰 이득.**
- D3D나 OpenGL관련 코드들을 제거한 형태
- 서버와 클라이언트를 모두 고려해서 엔진을 설계해야 하는 것이 부담.
- 서버와 클라이언트의 동일한 시뮬레이션을 보장한다.
- 처음에는 개발기간이 늘어나는것 같지만 결과적으로는 훨씬 단축시켜준다.

처음 엔진 개발하던 시절
(since 2002)

는



이야

초기 개발 히스토리

- 3ds max 모델뷰어
- 애니메이션 파일을 분리
- 애니메이션을 적용
- Renderere, Geometry 등 DLL과 exe로 분리
- 화면에 판자 하나 깔고 그 위에 캐릭터 띄움
- 하나의 모델 여러 개의 인스턴스를 사용하려고 하니 문제가 됨. 특히 계층 구조 적용시
  - Shader가 없던 시절. Skining된 버텍스는 어딘가에 저장해야함.
- 모델 class에 context를 포함

초기 개발 히스토리

- Frustum culling
- Quad tree적용
- 제대로 된 공간분할 시도 – BSP/PORTAL , ROOM/PORTAL
- 공유되지 않는 정적 모델 데이터 생성
  - 잘려진 공간에 맞게 모델을 자르고 재구성

- ROOM/PORTAL방식 적용
  - 모델링/편집이 가능한 Level Editor 제작.
- 정적 모델 데이터에 light map적용
- MMOG특성상 Height Field의 필요성. Height Field 구현.
- Height Field에 per vertex lighting 적용

초기 개발 히스토리

- 월드 구조를 Scene이라는 구조로 재구성.
- Model Data + Index 를 캡슐화하는 ICharacterObject구현
- 엔진에서 다수의 Scene을 동시에 처리할 수 있도록 함(for Server)
  - 완전한 3D서버를 위해
- 충돌처리 코드 작성 -> 가상함수 인터페이스를 노출하는 DLL로 분리
- H/W Occlusion Culling등 가속화 알고리즘 추가 구현.
- 서버에서 실제로 사용하며 서버를 위한 기능 구현

megayuchi엔진

megayuchi엔진의 개발 목표

- 3D 필드 위를 자유롭게 돌아다닐 수 있을것.
- 온라인 프로젝트에서 서버/클라이언트가 공통 사용 가능할것
- 빠른 렌더링 속도.
- 빠른 로딩 속도.
- 적은 메모리 소모.
- 엔진을 업데이트 하면 게임이 자동으로 업데이트
- 엔진 코드의 변화로 인해 게임 빌드가 망가지면 안됨.
  - DirectX관련 데이터타입은 Renderer.dll프로젝트 내에서만 사용한다.

megayuchi 엔진의 특징

- 다수의 DLL로 쪼개져 있어 ‘빌드-디버깅’의 이터레이션 빠름
- 동일 exe, 다른DLL로딩으로 다른 기능 지원
- DX11,DX12,DXR 지원
- UWP(for XBOX지원)
- 서버/클라이언트 겸용
- 서버/클라이언트 네트워크 워크 동기화를 염두한 설계
- Voxel World 지원

모듈 구성

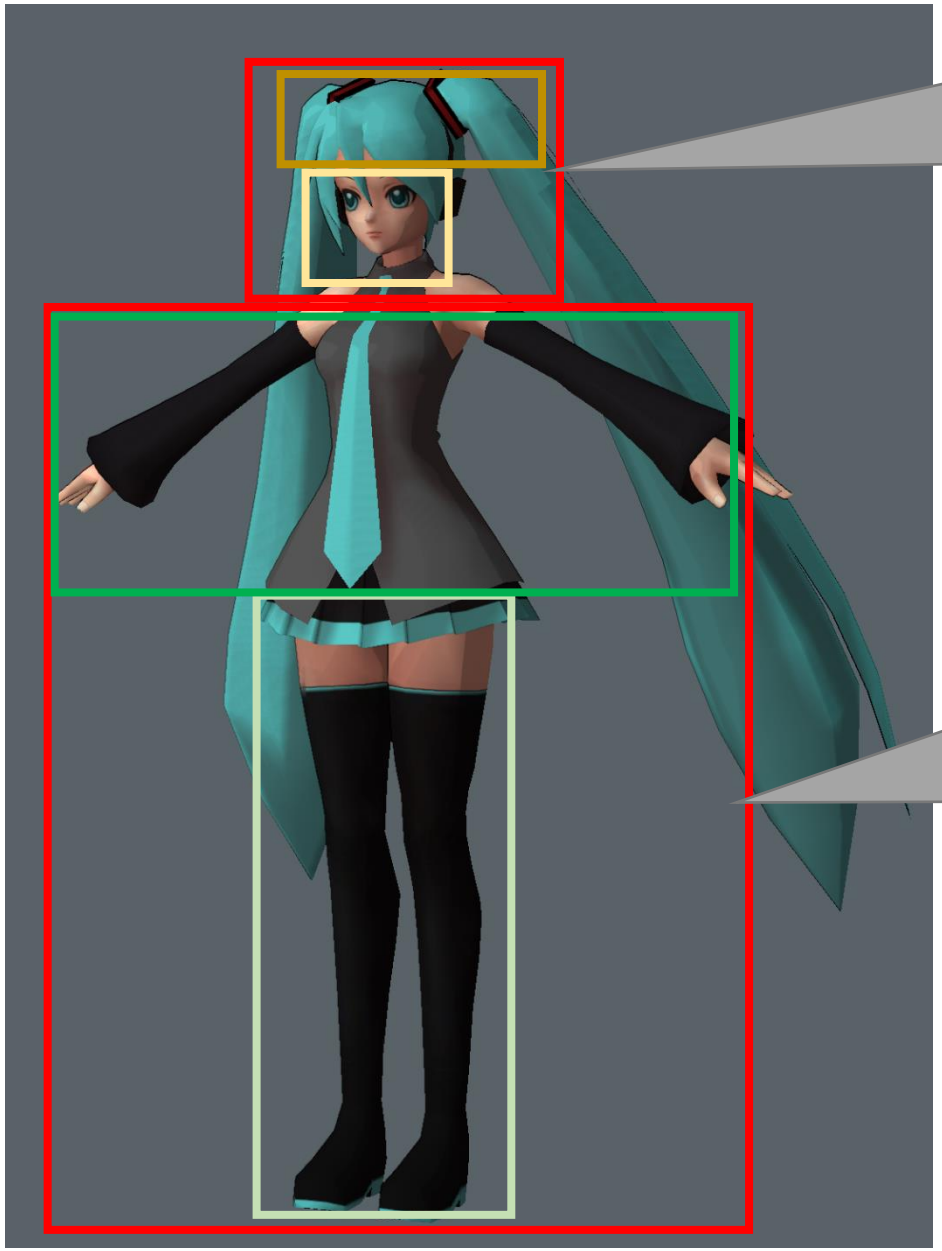
- MathLib.dll – 수학함수
- GenericLib.dll – 자료구조,검색,정렬,Memory Pool, Heap등.
- FileStorage.dll – pakcing된 파일과 일반파일을 동시 억세스 하기 위한 파일시스템.
- Renderer.dll – DirectX,OpenGL등 그래픽 API를 직접 호출하고 상위 레이어에 대해 독립적인 렌더링 기능을 제공한다.
- Geoemtry.dll – 오브젝트 관리,맵 관리, 트리거, 충돌처리 등등 직접 화면에 렌더링만 안하는 게임 엔진 본체
- Collision.dll – 충돌처리 및 오브젝트와 속도벡터를 넣었을때 최종 위치와 속도벡터를 산출해주는 이동처리 엔진. GPGPU지원을 위해 따로 분리했다.
- Network.dll – IOCP기반, 서버와 클라이언트 공용 네트워크 엔진.
- DBAccess.dll – OLEDB기반, SQL서버와 통신. 비동기/동기 모드 양쪽 모두 지원.

서버와 클라이언트의 코드 공유

Model data

모델뷰어와 엔진의 차이점을 가르는 기준 – 리소스 참조 기능의 여부

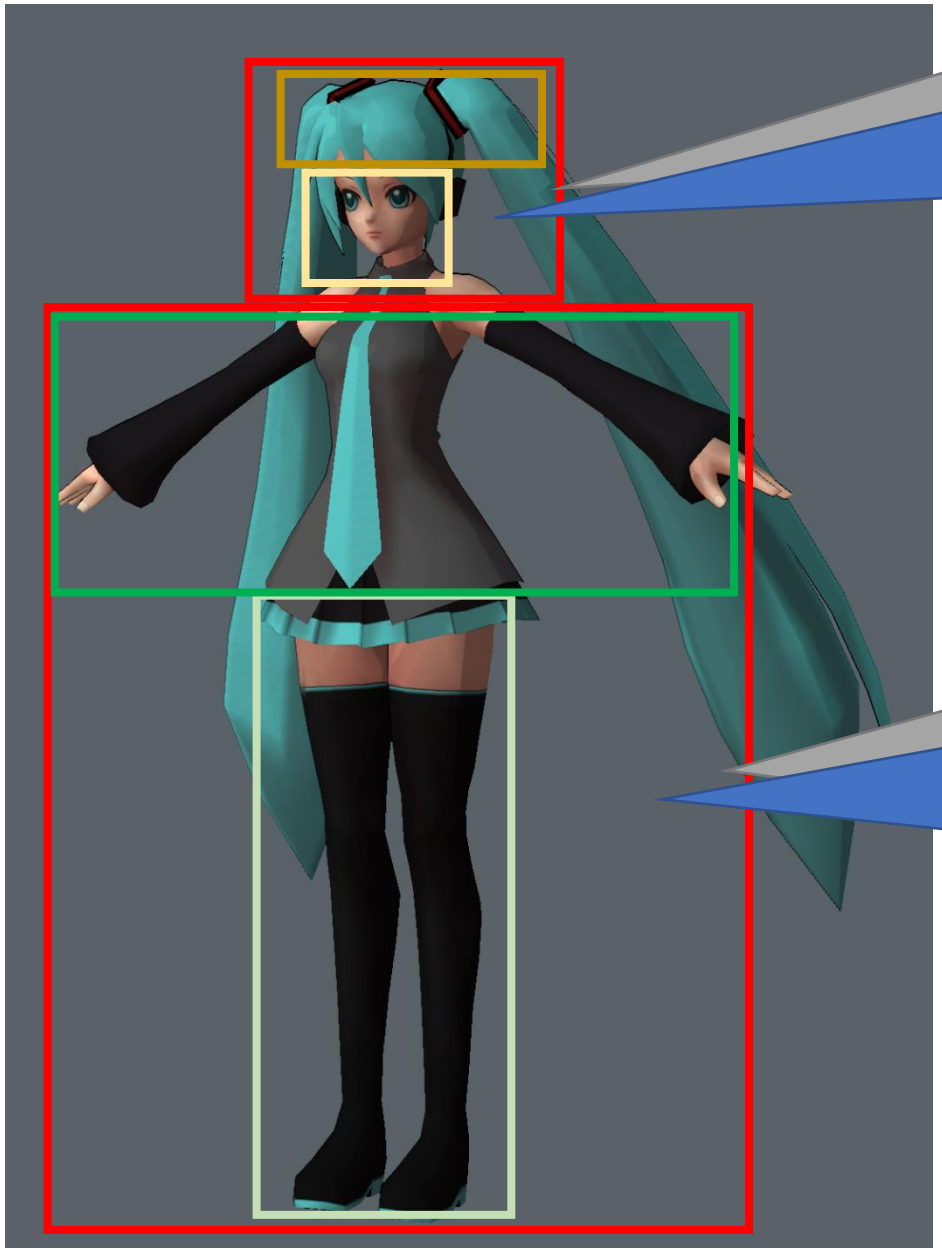
3dsmax



Object : “head”
Sub materials
+ #1: hair.dds
+ #2: face.dds

Object : “body”
Sub materials
+ #1: body_lower.dds
+ #2: body_upper.dds

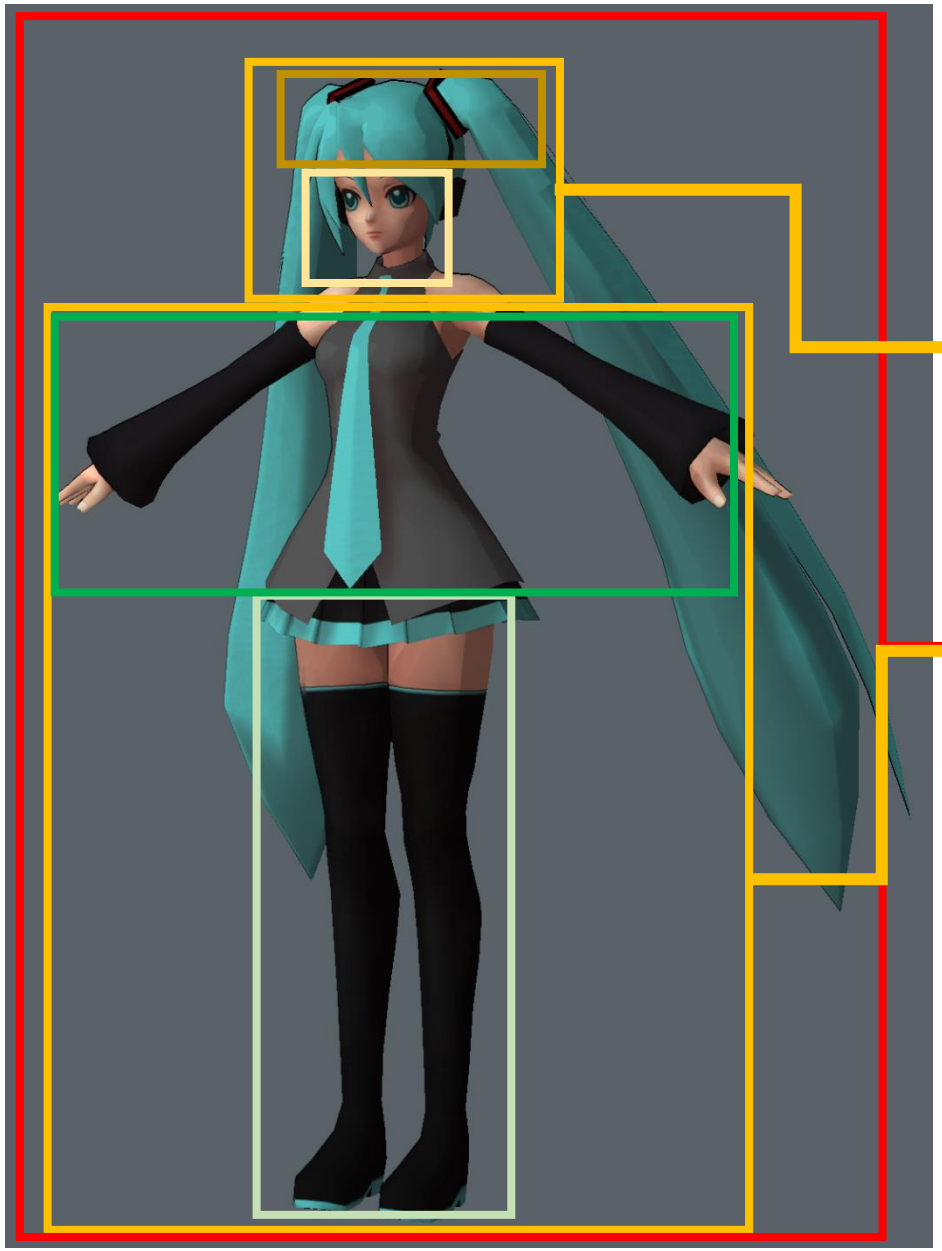
Megayuchi Engine



Object : “head”
+ Vertex Buffer
FaceGroup List
+ IndexBuffer , Mtl (hair.dds)
+ IndexBuffer, Mtl (face.dds)

Object : “body”
+ Vertex Buffer
FaceGroup List
+ IndexBuffer , Mtl (body_lower.dds)
+ IndexBuffer, Mtl (body_upper.dds)

Megayuchi Engine



Geometry

Model

CMeshObject
Name : “head”
IDIMeshObject Ptr

CMeshObject
Name : “body”
IDIMeshObject Ptr

Context Table

Index 0, TM:Pos/Scale/Rot Motion: 0, Frame :1
Index 1, Pos/Scale/Rot Motion: 4, Frame :3
Index 2, TM:Pos/Scale/Rot Motion: 16, Frame :10

Renderer

IDIMeshObject(CVLMesh)
+ Vertex Buffer

FaceGroup List
+ IndexBuffer , Mtl (hair.dds)
+ IndexBuffer, Mtl (face.dds)

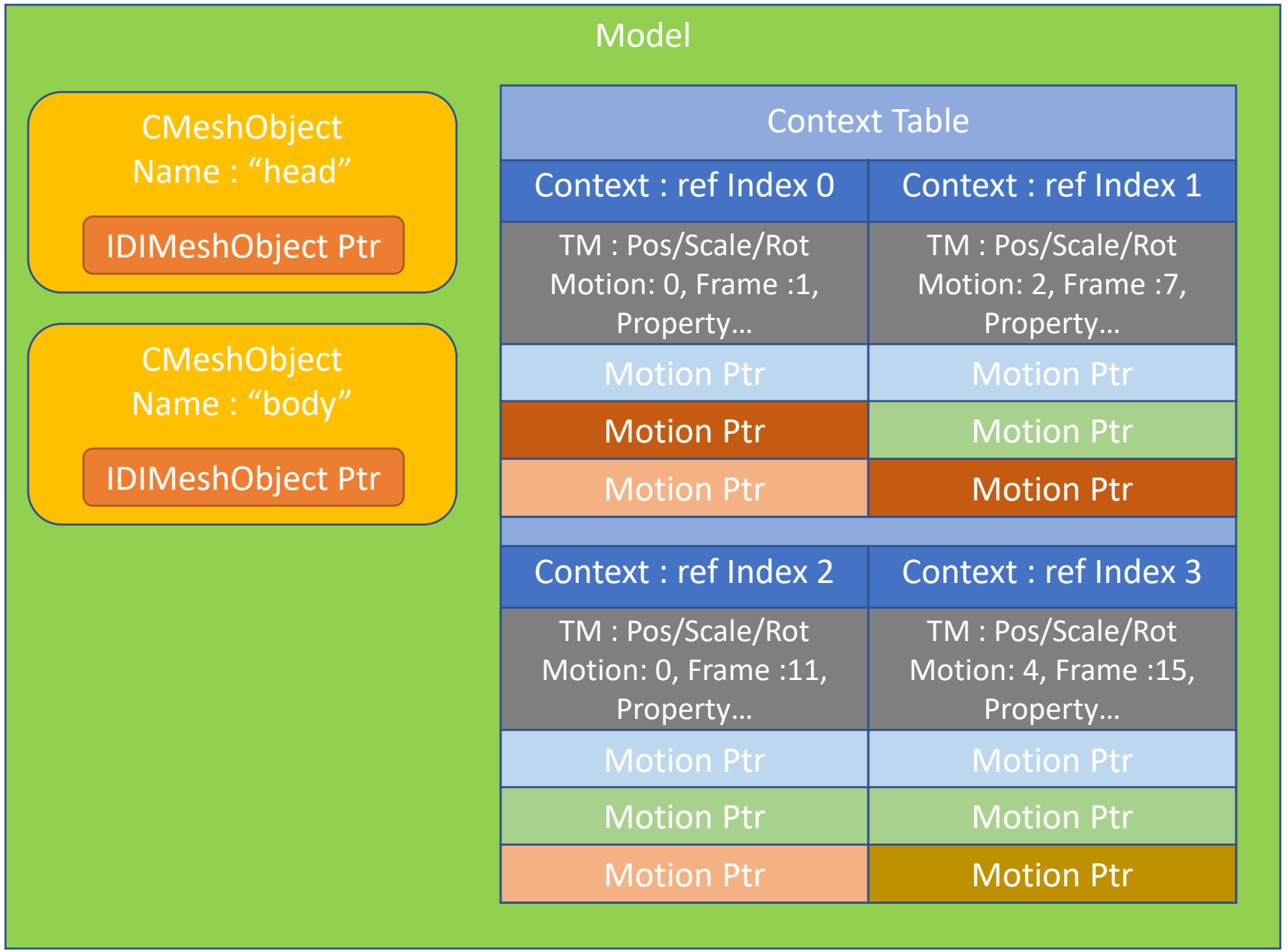
IDIMeshObject(CVLMesh)
+ Vertex Buffer

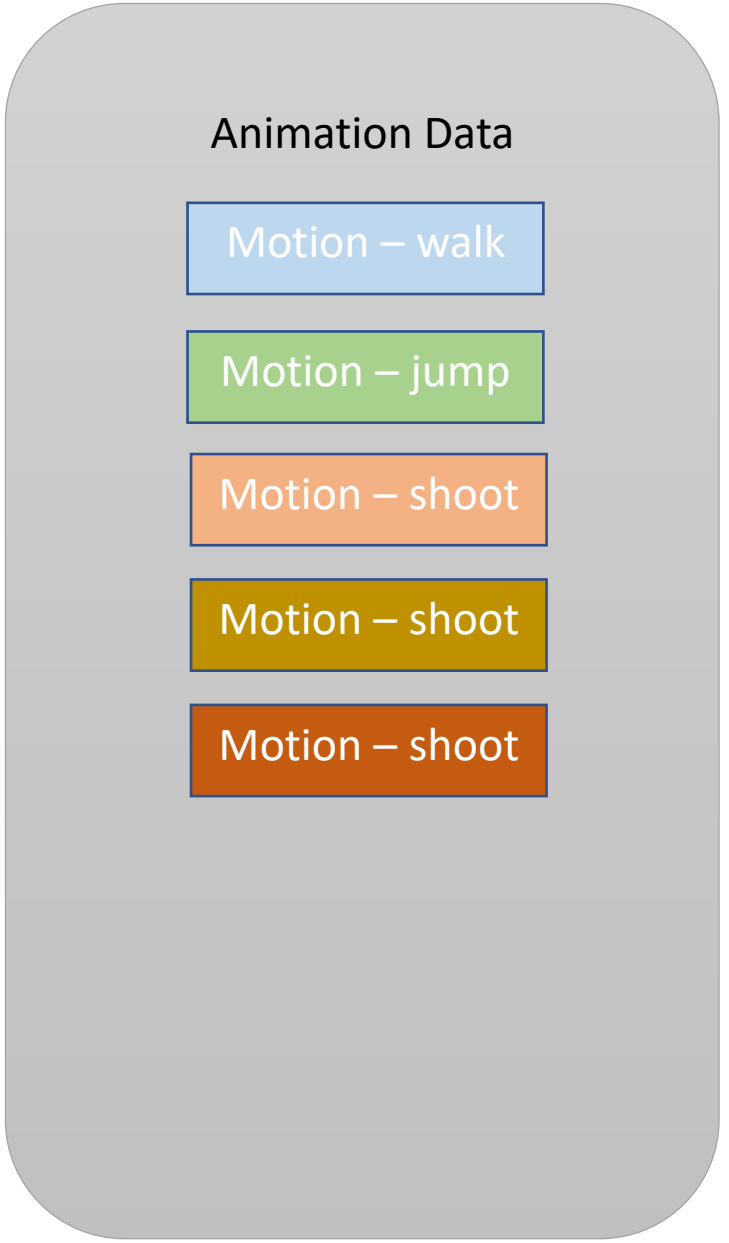
FaceGroup List
+ IndexBuffer , Mtl (body_lower.dds)
+ IndexBuffer, Mtl (body_upper.dds)

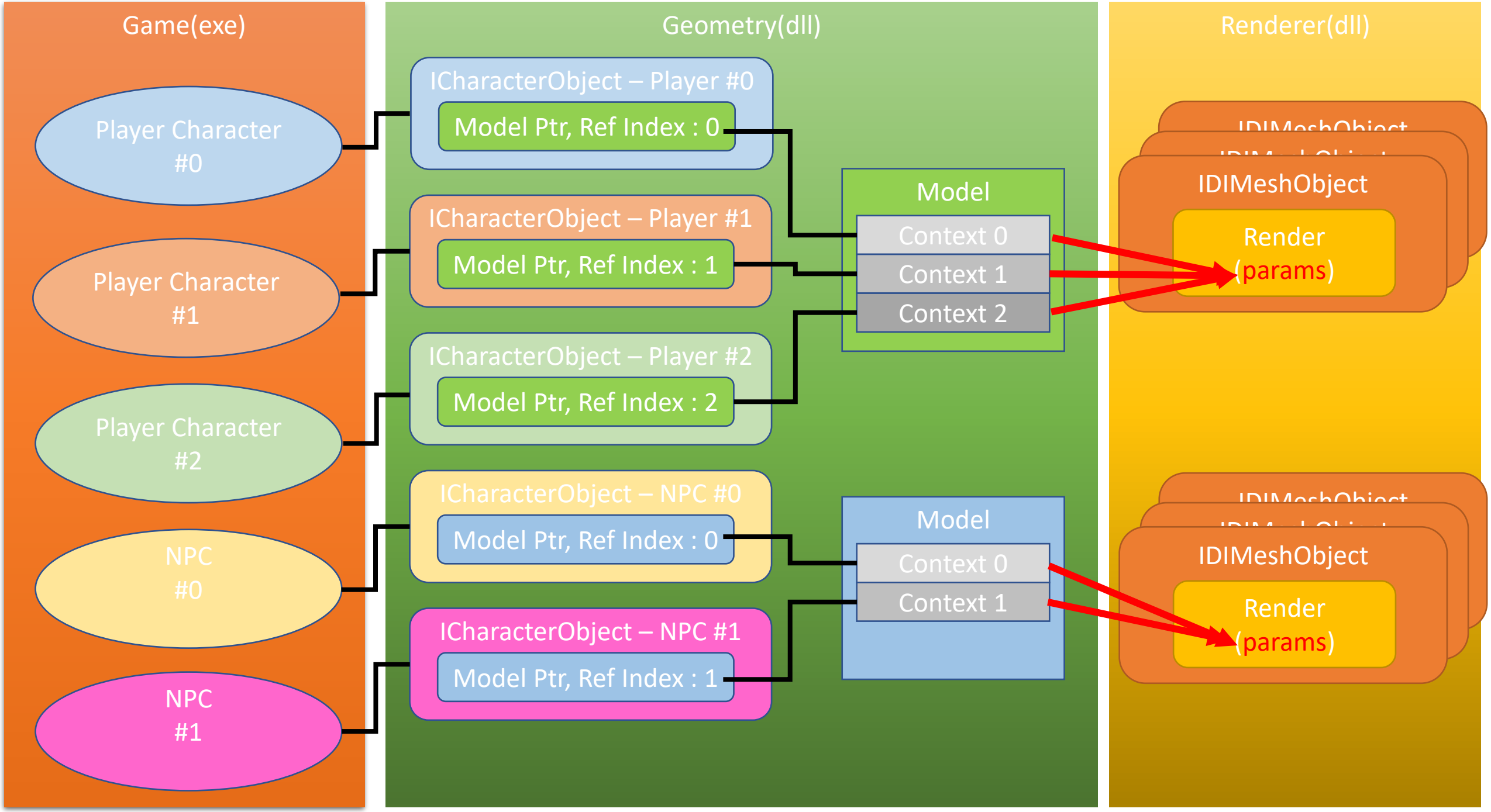
IDIMeshObject(CLMMesh)
+ Vertex Buffer

FaceGroup List
+ IndexBuffer , Mtl (wall.dds)

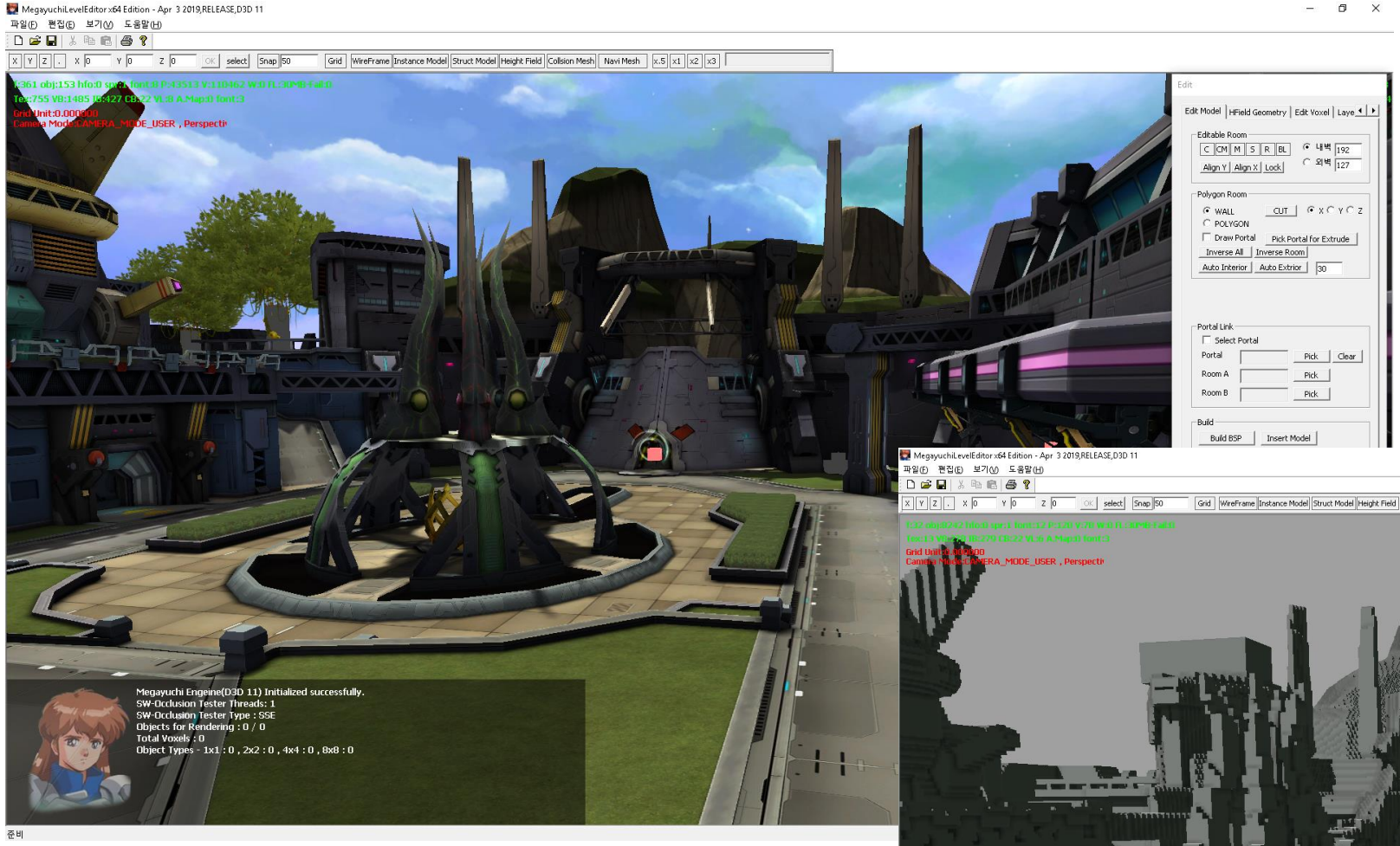
Model

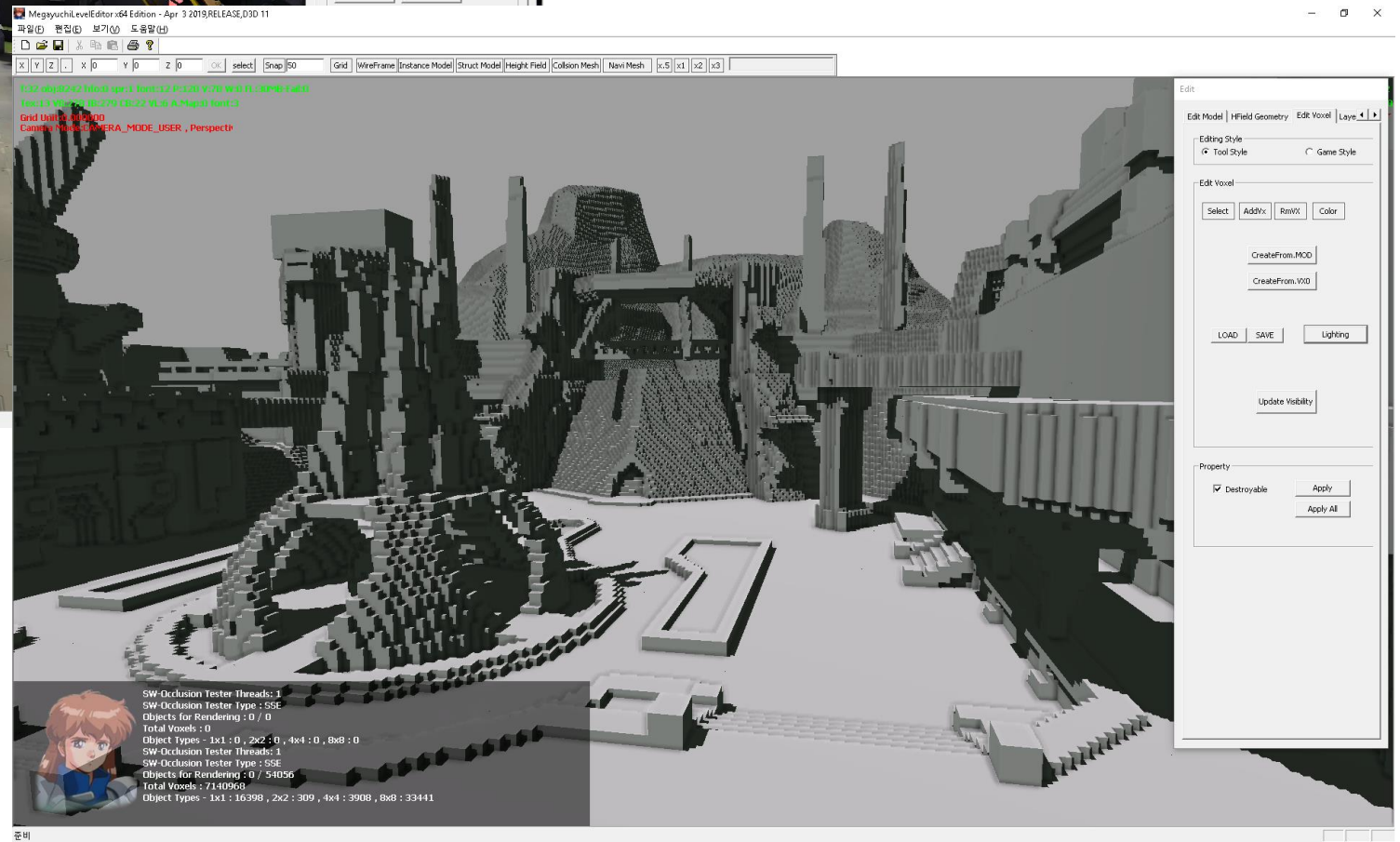

Model
CMeshObject
Name : “head”
IDIMeshObject Ptr
CMeshObject
Name : “body”
IDIMeshObject Ptr
Context Table
Context : ref Index 0
Context : ref Index 1
TM : Pos/Scale/Rot
Motion: 0, Frame :1,
Property…
TM : Pos/Scale/Rot
Motion: 2, Frame :7,
Property…
Motion Ptr
Motion Ptr
Motion Ptr
Motion Ptr
Motion Ptr
Motion Ptr
Context : ref Index 2
Context : ref Index 3
TM : Pos/Scale/Rot
Motion: 0, Frame :11,
Property…
TM : Pos/Scale/Rot
Motion: 4, Frame :15,
Property…
Motion Ptr
Motion Ptr
Motion Ptr
Motion Ptr
Motion Ptr
Motion Ptr


Animation Data
Motion – walk
Motion – jump
Motion – shoot
Motion – shoot
Motion – shoot


Game(exe)
Geometry(dll)
Renderer(dll)
Player Character #0
Player Character #1
Player Character #2
NPC #0
NPC #1
ICharacterObject – Player #0
Model Ptr, Ref Index : 0
ICharacterObject – Player #1
Model Ptr, Ref Index : 1
ICharacterObject – Player #2
Model Ptr, Ref Index : 2
ICharacterObject – NPC #0
Model Ptr, Ref Index : 0
ICharacterObject – NPC #1
Model Ptr, Ref Index : 1
Model
Context 0
Context 1
Context 2
Model
Context 0
Context 1
IDIMeshObject
Render (params)
IDIMeshObject
Render (params)

Map(scene)


MegayuchiLevelEditor x64 Edition - Apr 3 2019,RELEASE,D3D 11
파일(F) 편집(E) 보기(V) 도움말(H)
Megayuchi Engeine(D3D 11) Initialized successfully.
SW-Occlusion Tester Threads: 1
SW-Occlusion Tester Type : SSE
Objects for Rendering : 0 / 0
Total Voxels : 0
Object Types - 1x1 : 0 , 2x2 : 0 , 4x4 : 0 , 8x8 : 0
준비

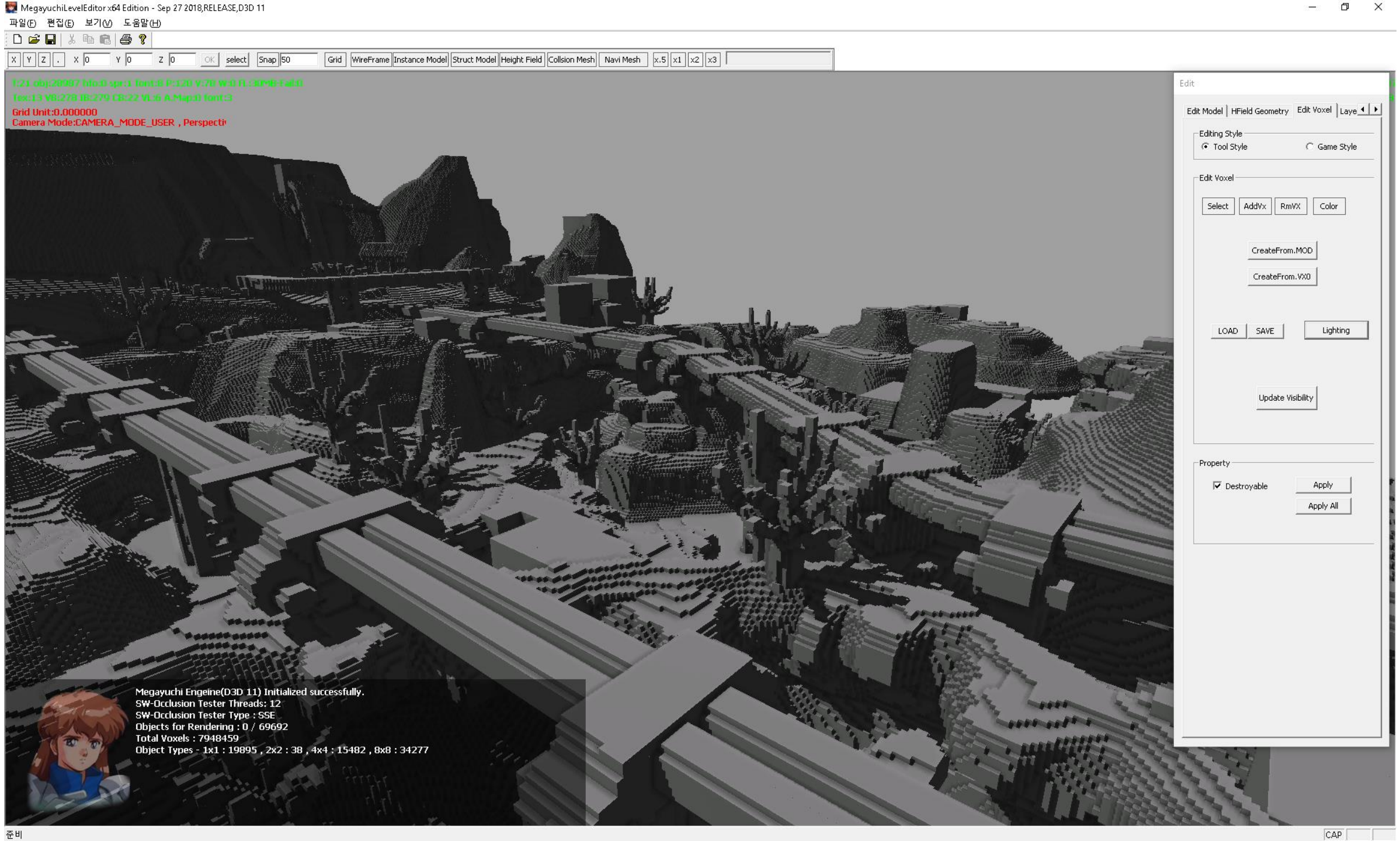

MegayuchiLevelEditor x64 Edition - Apr 3 2019,RELEASE,D3D 11
파일(F) 편집(E) 보기(V) 도움말(H)
SW-Occlusion Tester Threads: 1
SW-Occlusion Tester Type : SSE
Objects for Rendering : 0 / 0
Total Voxels : 0
Object Types - 1x1 : 0 , 2x2 : 0 , 4x4 : 0 , 8x8 : 0
SW-Occlusion Tester Threads: 1
SW-Occlusion Tester Type : SSE
Objects for Rendering : 0 / 54056
Total Voxels : 7140960
Object Types - 1x1 : 16398 , 2x2 : 309 , 4x4 : 3908 , 8x8 : 33441
준비

Map-Scene

- 기본 자료구조는 KD-Tree
- 3ds max에서 모델링한 데이터를 익스포트
- 모델 데이터를 사용해서 KD-Tree빌드
- KD-Tree에 맞춰서 모델링 데이터를 잘라낸 후 재구성
  - RoomMeshObject라는 단위로 새롭게 오브젝트 생성
- 동적 오브젝트는 맵툴에서 배치
- 다수의 scene로드 및 관리 가능
- 다수의 scene에 대해서 충돌처리 가능

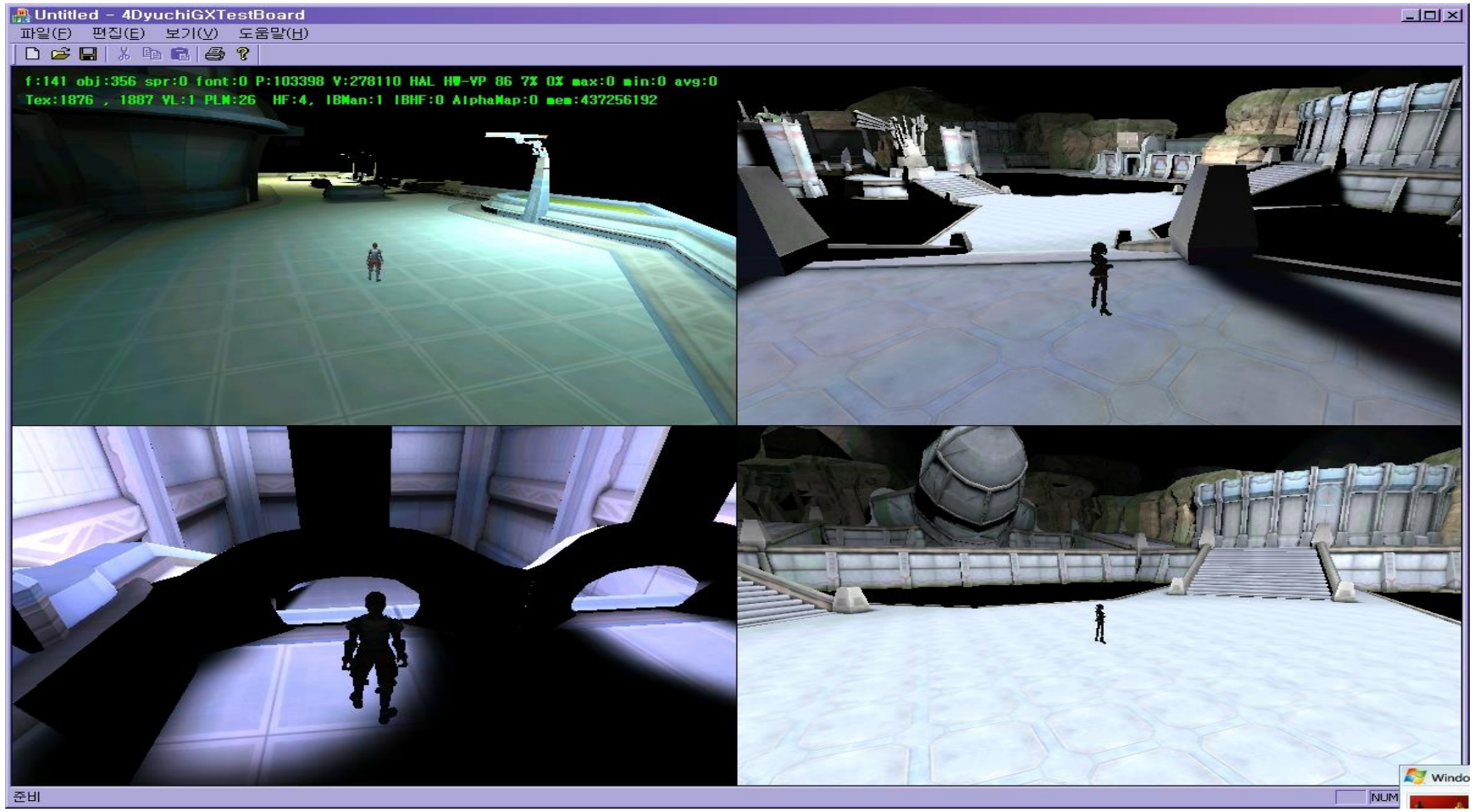
Map-Scene

- voxel 맵은 삼각형 베이스 맵으로부터 변환
- Voxel world인 경우 전용의 KD-Tree사용
- Voxel 맵도 인스턴싱 가능


MegayuchiLevelEditor x64 Edition - Sep 27 2018,RELEASE,D3D 11
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 Collsion Mesh Navi Mesh x.5 x1 x2 x3
Grid Unit:0.000000
Camera Mode:CAMERA_MODE_USER , Perspecti
Megayuchi Engeine(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
Edit
Edit Model
HField Geometry
Edit Voxel
Laye
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
준비
CAP

다수의 맵을 동시에 로드

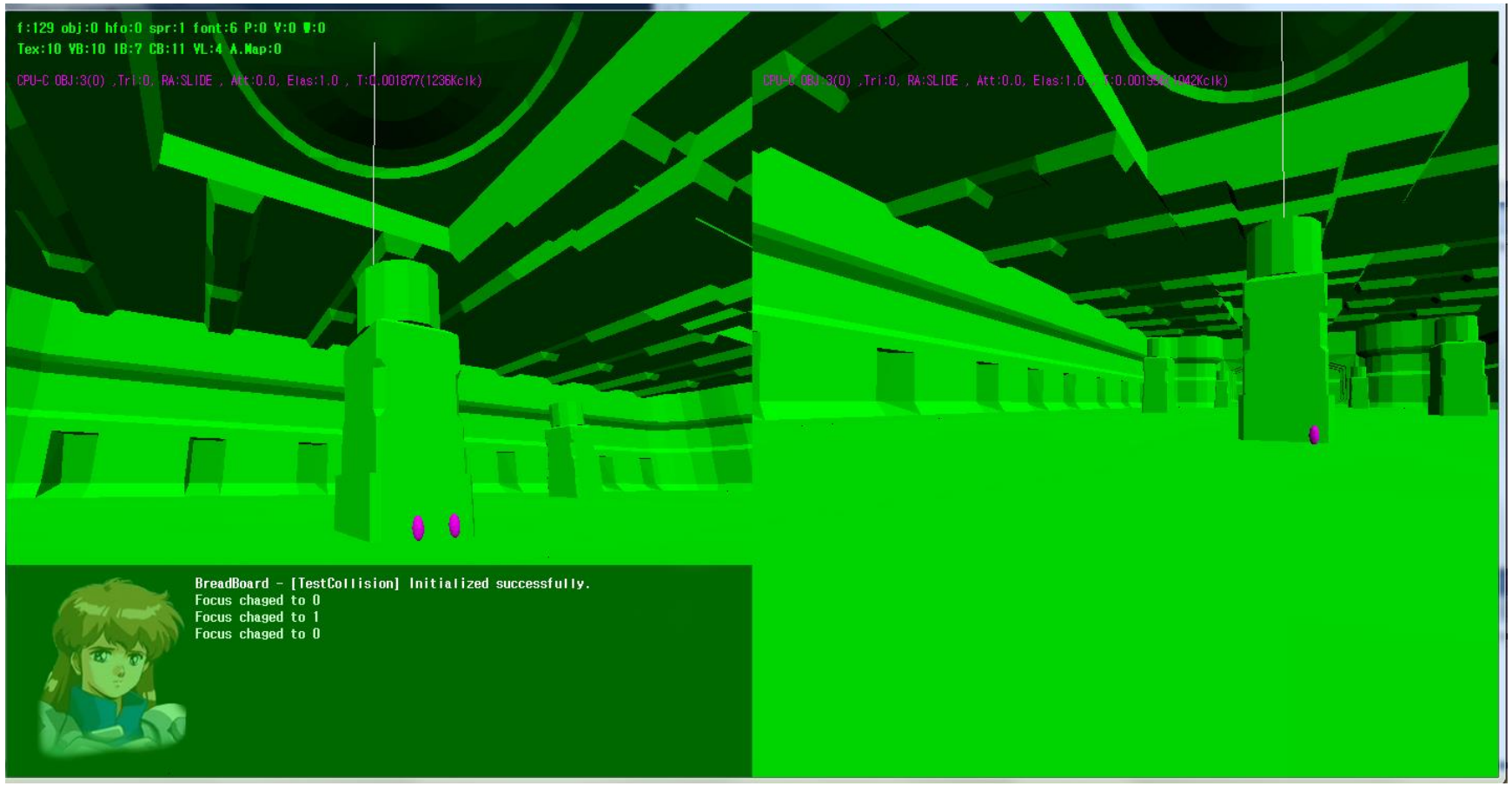
- 클라이언트는 한번에 1개의 맵을 로드한다.
- 서버는 모든 맵을 다 로드한다.



하나의 엔진에서 동시에 여러개의 맵을 로드한 경우

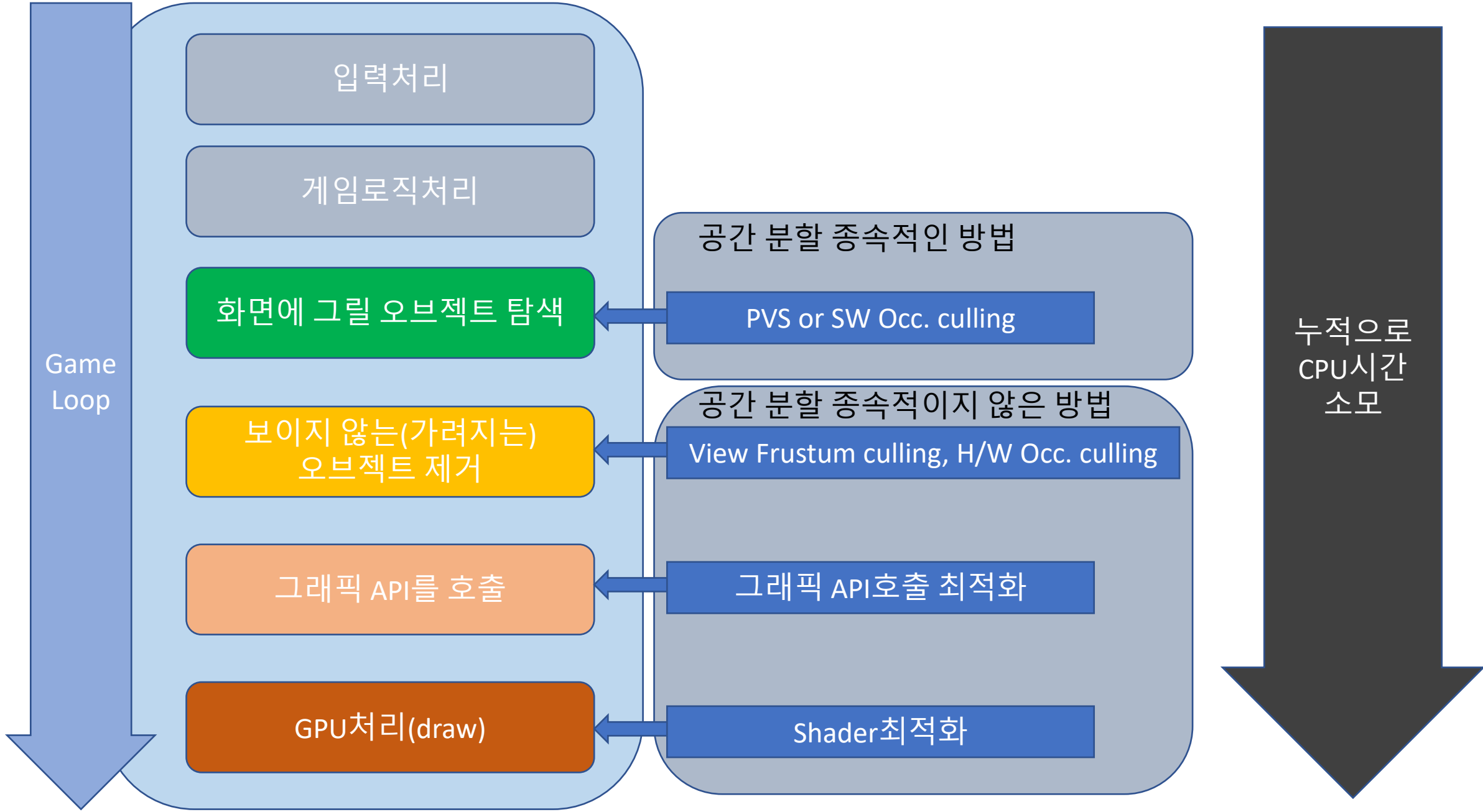
게임 맵 인스턴싱

- 하나의 리소스로 여러 개의 맵을 생성한다.
- 채널 사용을 위해선 필수.
- 그 밖에도 수백 수천개의 인스턴스 맵이 생성될 수 있다.
- 메모리를 절약하고 성능을 높이려면 반드시 필요하다.



- **하나의 리소스 여러개의 맵 인스턴스**
  고정적인 지형지물은 하나의 리소스를 참조해서 사용.
  움직이는 오브젝트들은 새로 생성해서 사용.

Object Culling

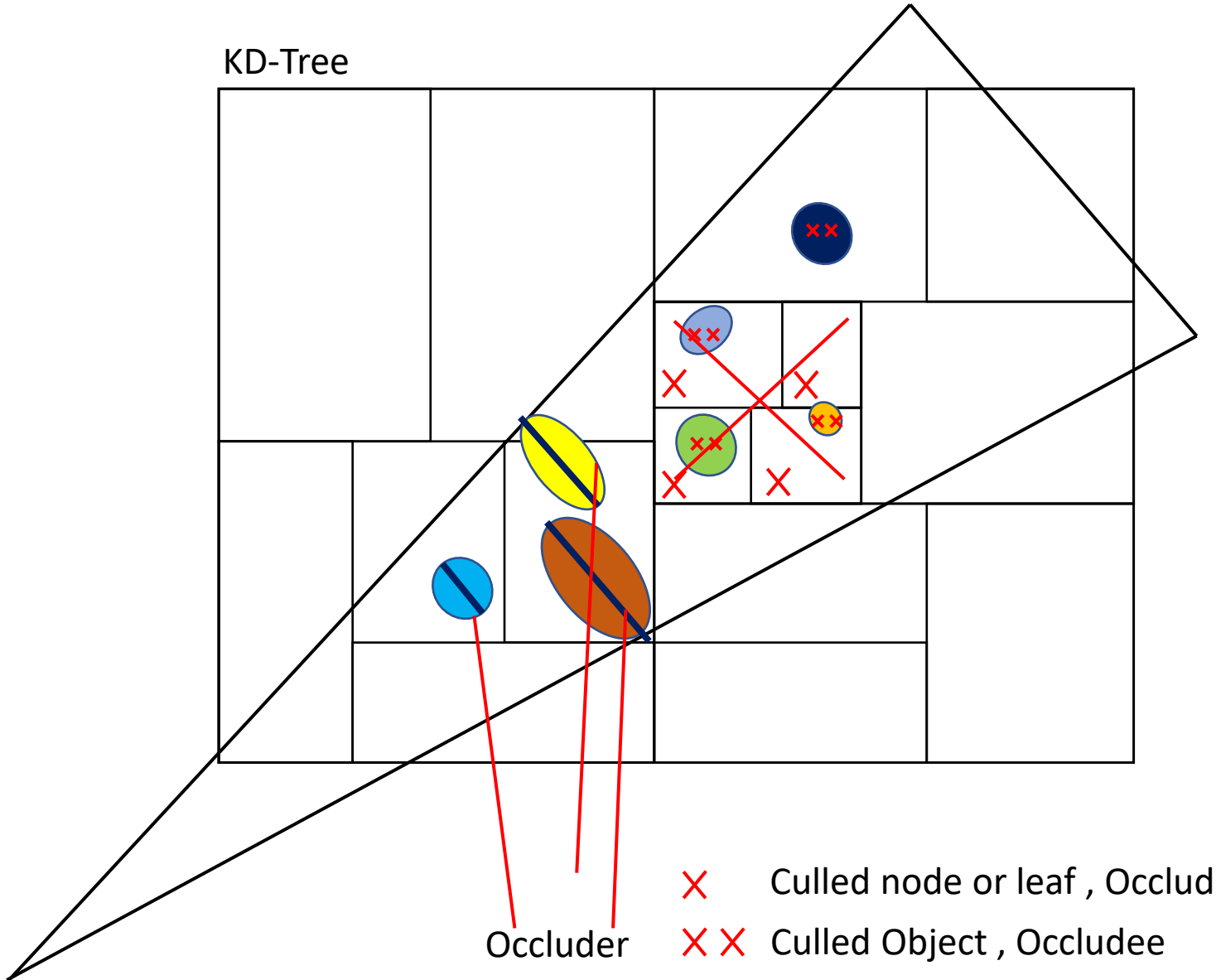


앞 단계에서 많은 오브젝트들을 제거할 수록 성능이 향상될 가능성이 높다.

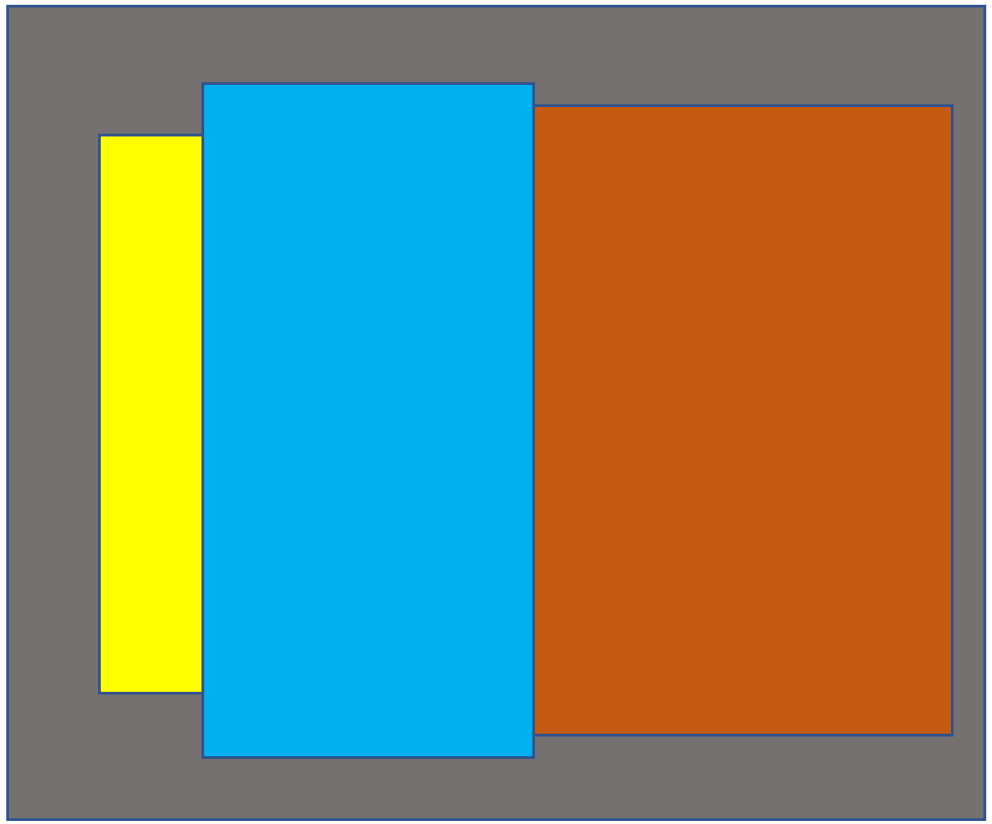
SW Occlusion Culling in KD-Tree

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

KD-Tree

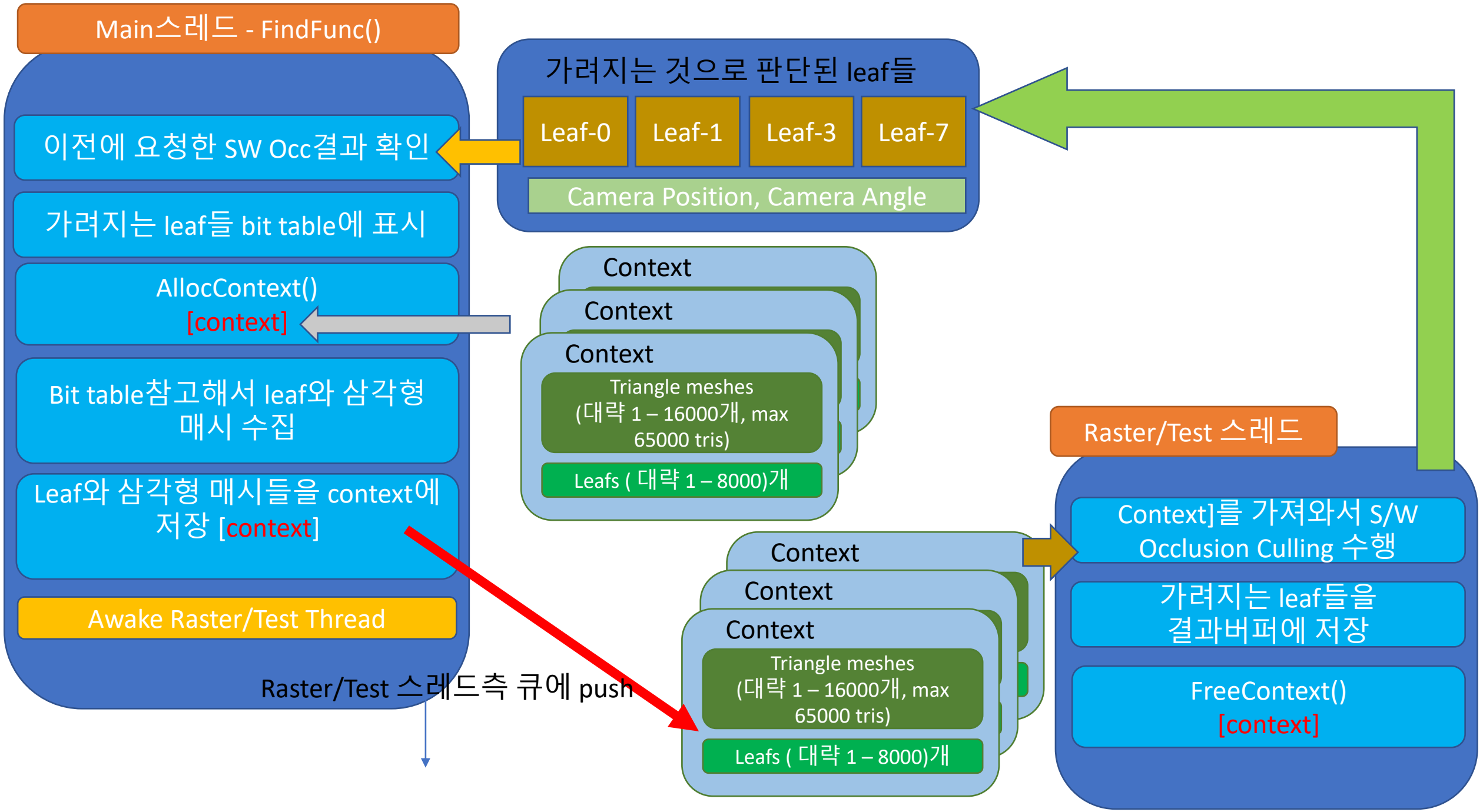


Frame Buffer or Z-Buffer

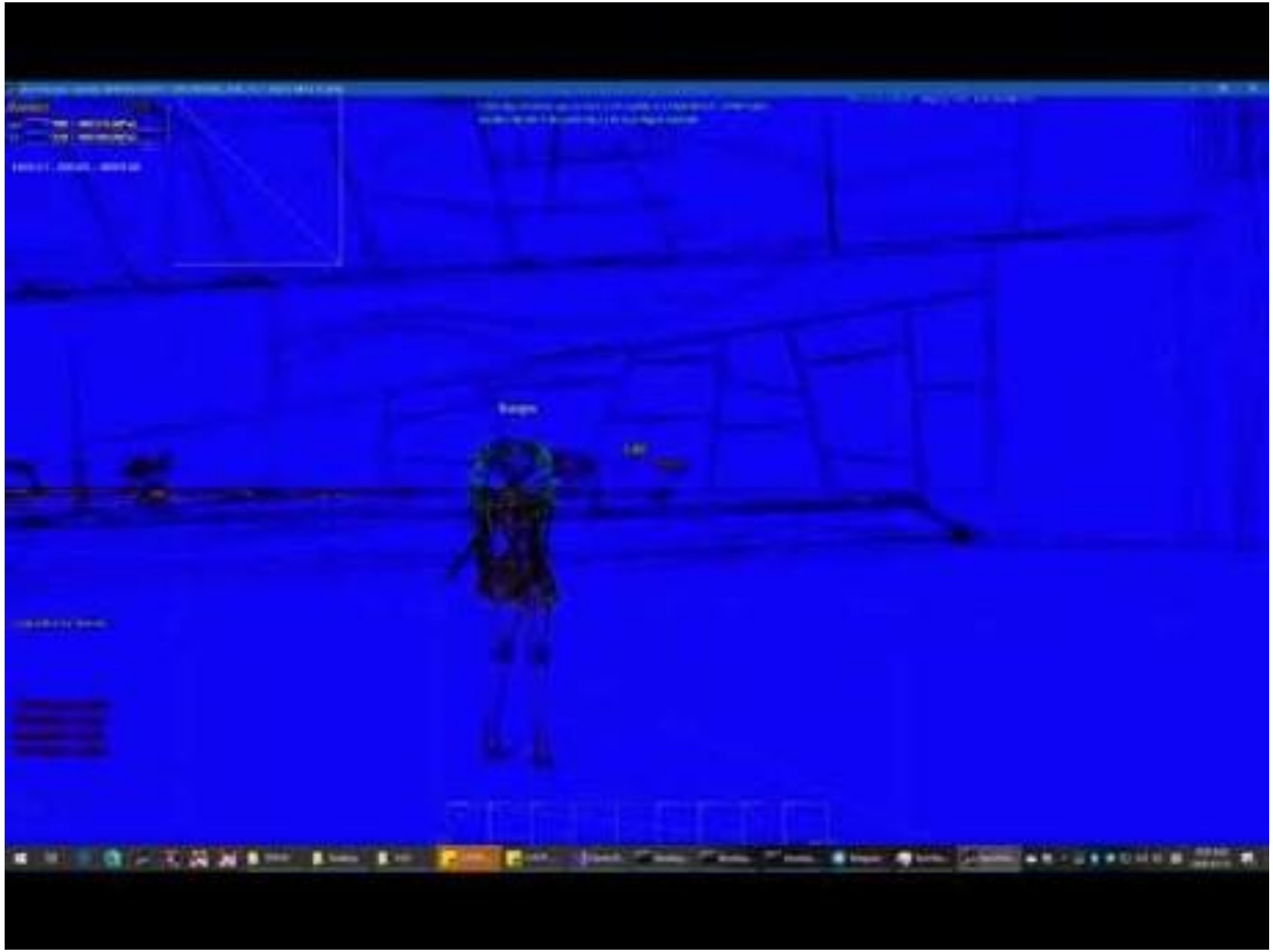


Culled node or leaf , Occludee

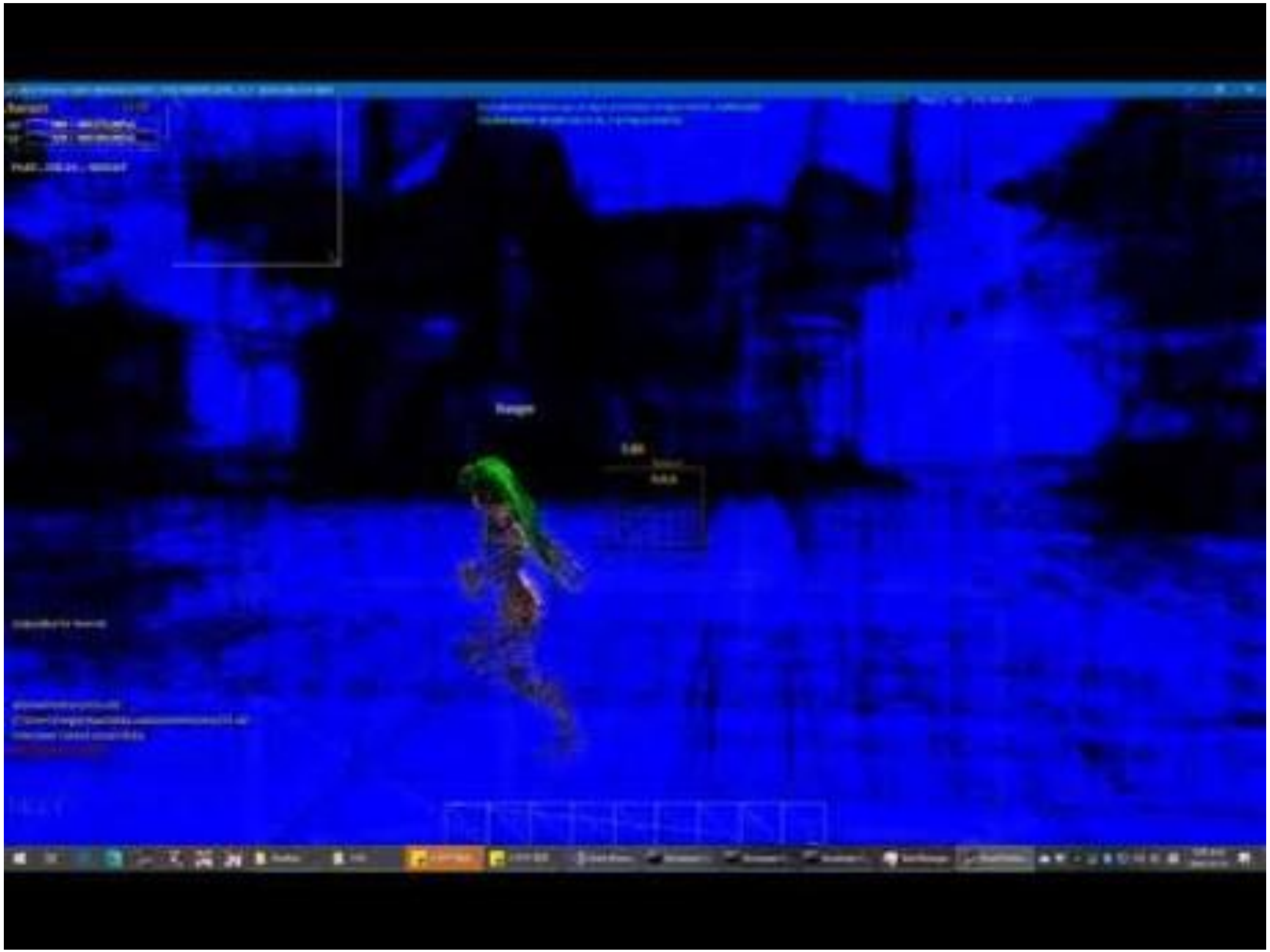
Culled Object , Occludee


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]

S/W Occlusion Culling in Triangles based map



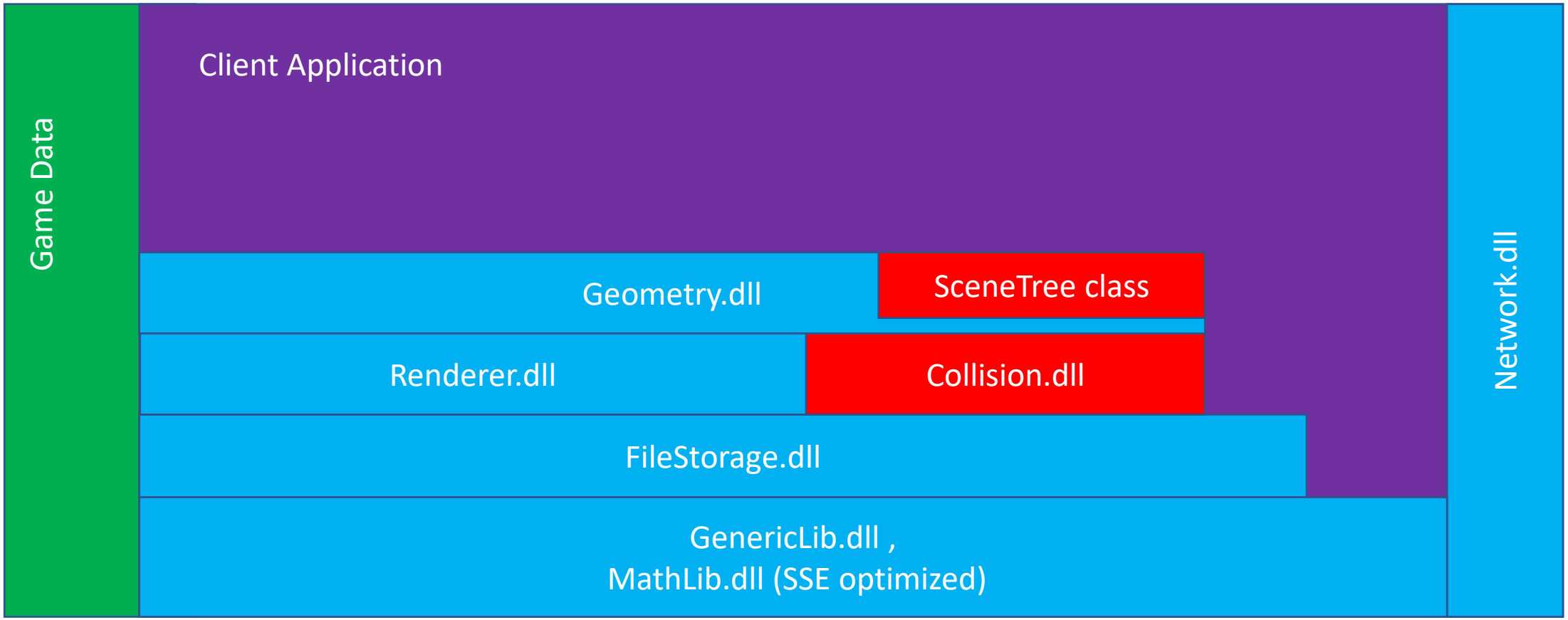
S/W Occlusion Culling in Voxel World



충돌처리

이동&충돌&타격 판정의 중요 모듈

1. Collision.dll (충돌처리 엔진)
2. SceneTree class (오브젝트 픽킹 및 검색 엔진)



Collision.dll (충돌처리 엔진)

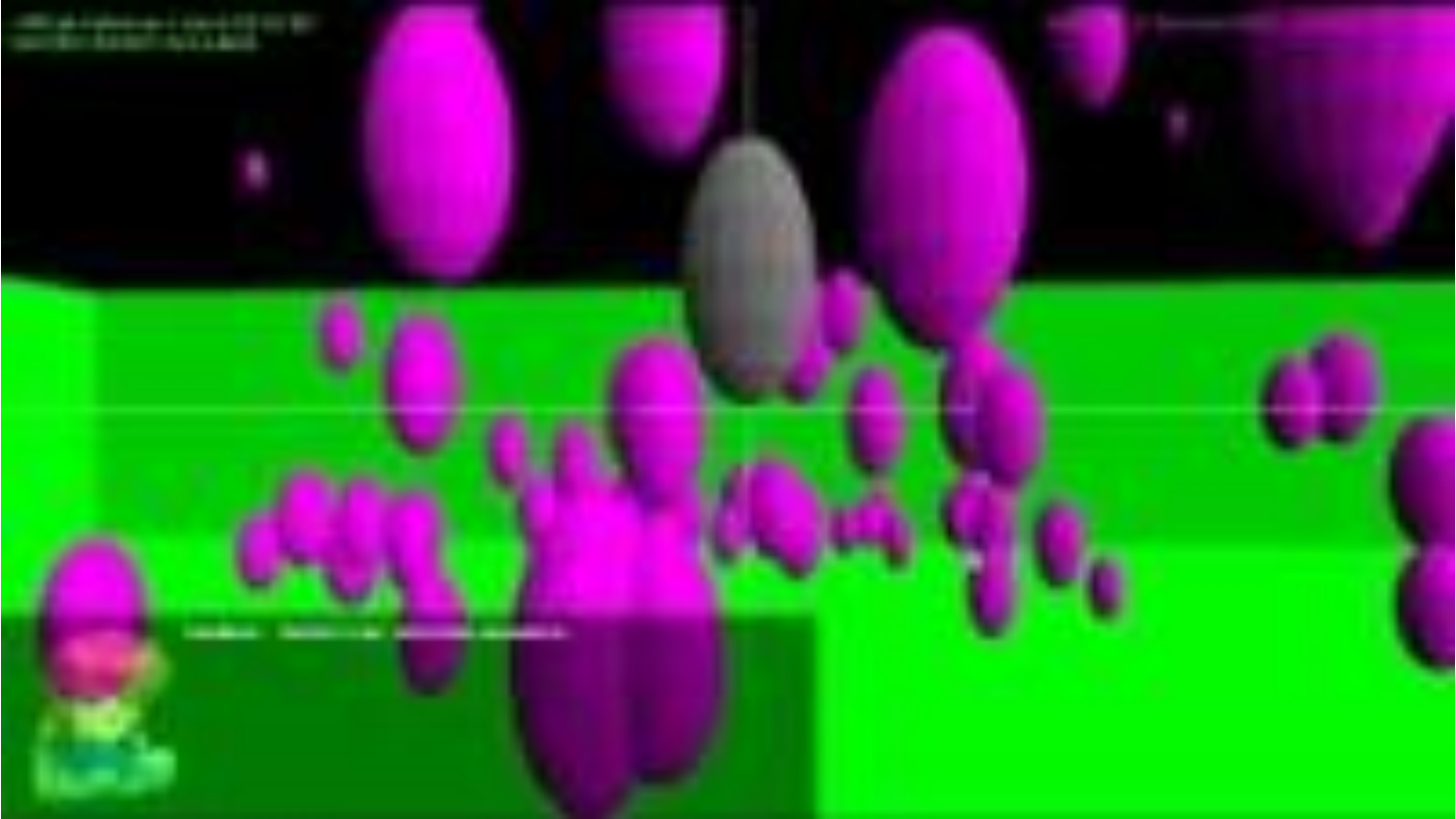
- 3차원 그리드 자료구조 – KD Tree를 사용하지 않고 그리드구조를 사용한 이유는 GPGPU최적화 때문
- 멀티 스레드
- 삼각형에 충돌했는지 타원체에 충돌했는지 비트플래그 리턴
- 최종 속도벡터 리턴
- 로켓탄이 어딘가에 충돌했을때 타원체 충돌했다는 비트가 켜지면 로켓탄의 데미지 범위로 SceneTree로부터 캐릭터 오브젝트 탐색.

충돌처리 기본 컴포넌트

- 움직이는 타원체 vs 삼각형
- 움직이는 타원체 vs 타원체
- 움직이는 타원체 vs 움직이는 타원체

충돌처리 기본 리액션

- 충돌시 미끄러짐
- 충돌시 정지
- 충돌시 반사

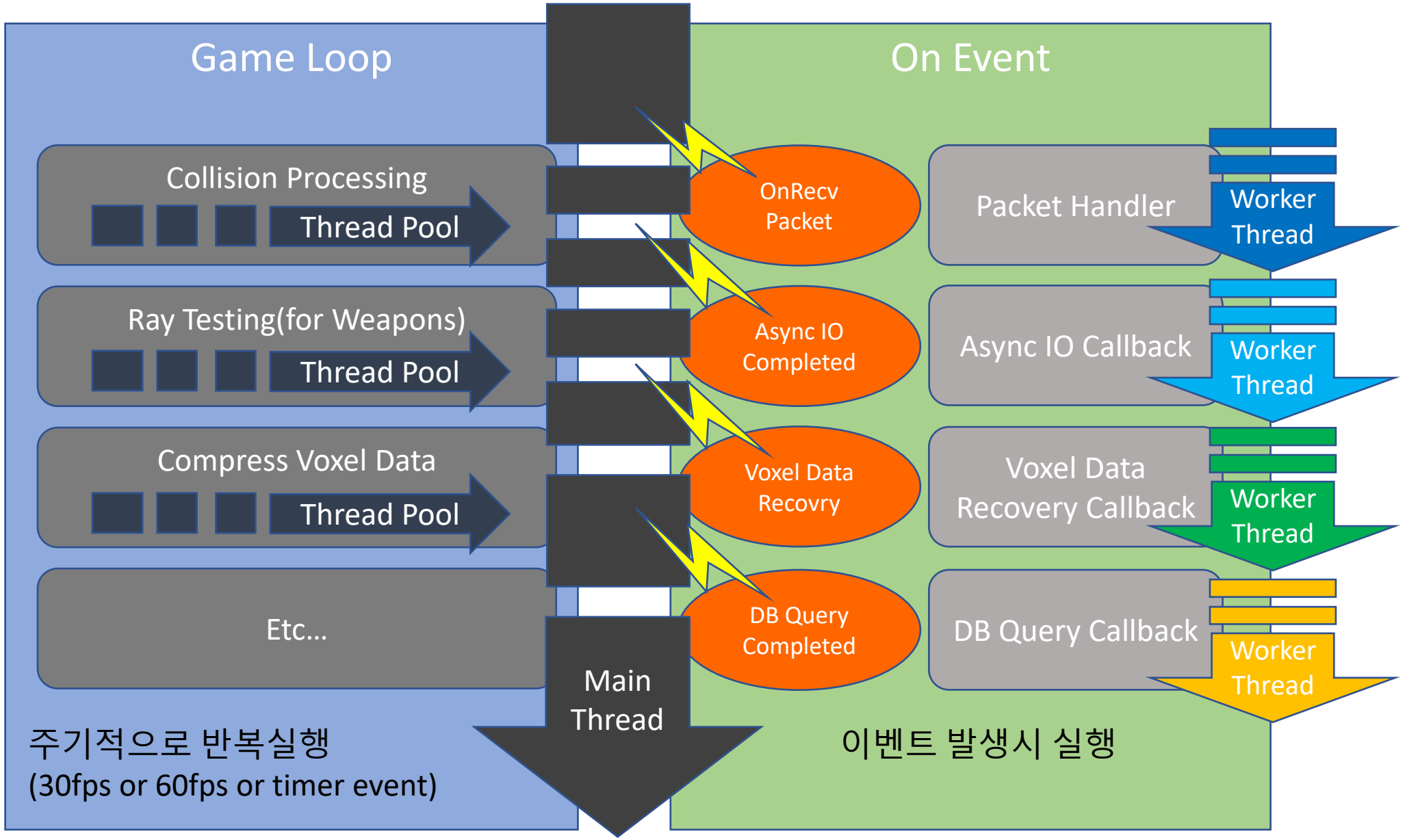


네트워크

네트워크 라이브러리

- IOCP기반
- .DLL엔진. 서버와 클라이언트 모두 동일한 바이너리 사용.
- 서버에서 사용할 경우 자체 스케쥴링.
- 클라이언트에서 사용할 경우 windows message와 interop.

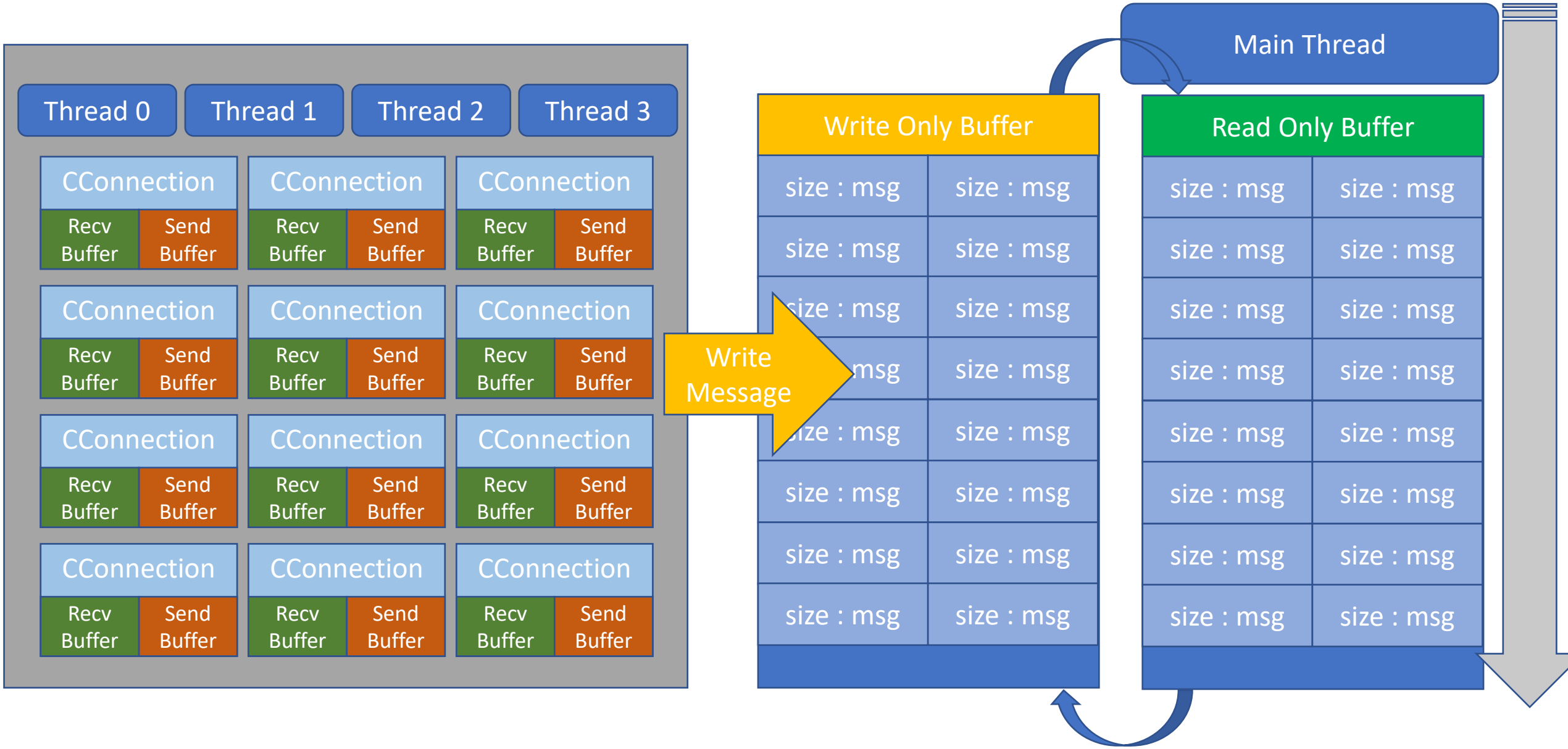
서버의 작업 스케쥴링


Game Loop
On Event
Collision Processing
Thread Pool
Ray Testing(for Weapons)
Thread Pool
Compress Voxel Data
Thread Pool
Etc...
Main Thread
OnRecv Packet
Async IO Completed
Voxel Data Recovry
DB Query Completed
Packet Handler
Async IO Callback
Voxel Data Recovery Callback
DB Query Callback
Worker Thread
Worker Thread
Worker Thread
Worker Thread
주기적으로 반복실행
(30fps or 60fps or timer event)
이벤트 발생시 실행

패킷 수신 -> 메시지 처리

- 최소 패킷 구조 = size(4 bytes) + body(N bytes)
- I/O 워커 스레드의 메시지 수집과 메인 스레드의 경쟁 상태를 줄인다.
- Double buffering
  - 하나 이상의 패킷이 수집되면 Network측 Worker thread가 쓰기 버퍼에 수집된 패킷을 써넣는다.
  - Main Thread는 패킷 수신이 통보되면 쓰기 버퍼와 읽기 버퍼의 포인터를 swap한다(가벼운 lock사용).
  - Main Thread는 읽기 버퍼의 쌓인 패킷을 처리한다.
  - 처리가 완료되면 쓰기 버퍼와 읽기 버퍼의 포인터를 swap한다.

패킷 수신 -> 메시지 처리


Thread 0
Thread 1
Thread 2
Thread 3
CConnection
Recv Buffer
Send Buffer
Write Message
Write Only Buffer
Read Only Buffer
size : msg
Main Thread

DB미들웨어

DB 미들웨어

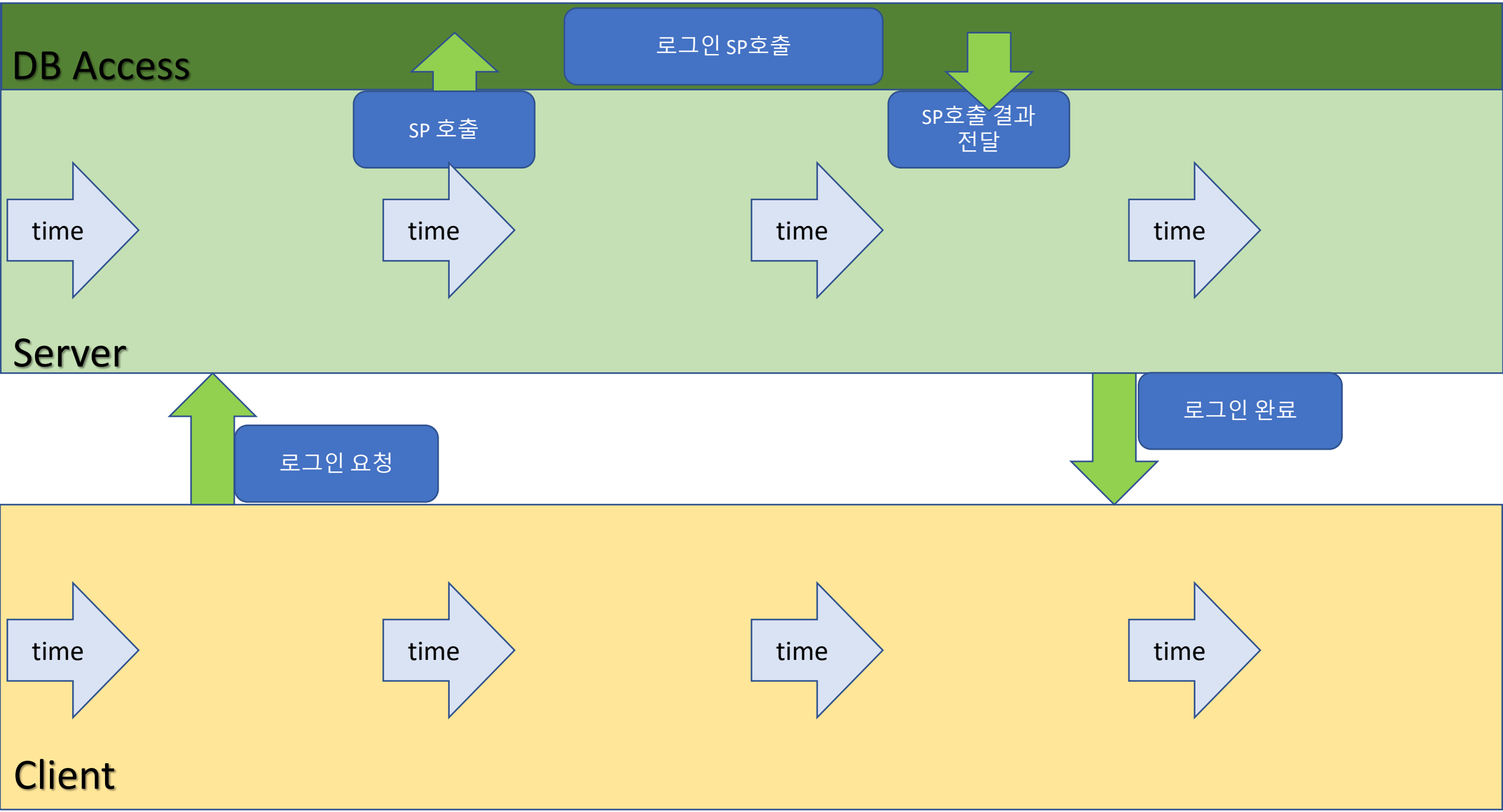
- 비동기 쿼리를 위한 중간 계층
- Insert, update, delete, select, Stored Procedure지원
- Virtual function interface노출
- MegayuchiDBAccess.dll – C++/OLEDB 기반
  - Sqlserver사용을 위한 빌드.
- MegayuchiDBAccessCLI.dll - C++/CLI - SqlClient기반
  - Sqlexpress localdb사용을 위한 빌드.
- 두가지 DLL모두 동일한 인터페이스로 기능상 차이 없이 사용 가능

비동기 DB쿼리

- DB에 쿼리 후 응답 수신 후 메모리에 업데이트
  - 로그인, 아이템 획득 등

- 서버의 메모리 업데이트 후 DB에 쿼리(저장)
  - 총알 소모, 캐릭터 데이터 세이브, 기타 등등

- 어느쪽이든 비동기 처리
- 절.대.로 DB의 게임서버의 패킷 처리 스레드가 DB의 응답을 대기해서는 안된다.


DB Access
로그인 SP호출
SP 호출
SP호출 결과
전달
time
time
time
time
Server
로그인 완료
로그인 요청
time
time
time
time
Client

Sound Library

Sound Library

- .dll엔진
- C++ Virtual function interface 노출
- 내부적으로 fmod사용. 언제든 교체 가능.
- 과거에는 Direct Sound를 사용했었다.
- Null device 사용 가능.

megayuchi 엔진의 현재

- 공간분할 및 culling
  - KD-Tree, S/W Occlusion Culling + H/W Occlusion Culling
- Voxel world지원 – voxel기반 게임 출시한지 1년됐음(망했음)
- 충돌처리
  - 다수의 맵과 다수의 동적 오브젝트를 처리할 수 있는 별도 DLL엔진
  - 타원 vs 타원, 타원 vs 삼각형 , 타원 vs 복셀지형
  - 자체 공간 분할
  - 멀티 스레드

megayuchi 엔진의 현재

- DX11/ DX12 / DX12-DirectX Raytracing 지원
- UWP API지원(for XBOX)
- 3D지형에서의 길찾기 기능 추가
  - 네비게이션 매시 편집 기능 추가

megayuchi 엔진의 현재

- 계속 이 물건으로 게임 개발중.
- 엔진도 업데이트중
- 한 템포, 두 템포씩 늦지만 그래도 화면발도 꾸준히 좋아지고 있음.

Tool chain

- Model Viewer
- Level Editor
- Game Settings Editor
- Packaging Tool
- Converting Tool
  - Triangled mesh to voxels

Megayuchi엔진의 철학

모듈화

- 바이너리 레벨에서의 격리 없이는 모듈화도 없다.
- 3ds max sdk, DirectX API가 모범사례
- C++ virtual function으로 억세스
- D3D나 OpenGL등의 API특화된 타입은 해당 렌더러에서만 사용. 상위 계층에선 이들의 존재를 모르게 할것.

빠른 이터레이션

- 코드 작성(수정) -> 빌드 -> 테스트 -> 디버깅 -> 코드 작성(수정) -> 빌드 -> 테스트 -> 디버깅 -> .....
- 빌드가 느린가?
- 테스트하고자 하는 기능을 구현할때까지 로딩이 너무 오래 걸리는가?
- 테스트 하고자 하는 기능을 테스트할때까지 진입 과정이 너무 긴가? 또는 복잡한가?

버그부터 잡는다.

- 기능 추가를 하려고 하기 전에 버그가 있으면 버그부터 잡는다.
- 알고 있는 버그가 0여도 버그는 있다.
- 하물며 알고 있는 버그가 n개 있으면 모르는 버그가 n x 10개 이상 있다.

자체 엔진을 개발하시겠다구요?

하고 싶으면 하는거지...

- **판대기 하나 위에 박스만 하나 그릴 수 있으면 그것도 엔진이다.**
  물론 원하는 게임이 그걸로 충분할 경우에만...
- 하려고만 하면 할 수 있다. 자료가 널렸으니까.
- 유감스럽게도 취업에 도움 된다고는 말 못한다.
- **하고 싶으면 하는거지...**

목표설정

- 이 물건으로 게임이 돌고 있는가?
  - 게임을 만들 수 없는 엔진이란건 아무 쓸모도 없다.
  - 종종 모델뷰어를 엔진과 착각한다.
- 잘 작동하는가?
  - 엔진을 신뢰할 수 있어야 컨텐츠를 올릴 수 있다.
- 쾌적한가?
  - 만족스러운 성능이 나오지 않으면 개발 과정 전체의 병목을 유발한다.
- 화면을 봐줄만한가?
  - 차차 개선해나간다. (혼자 개발하는 엔진의 화면발이 좋을리가 없잖아!)

설계와 구조

- 설계따위 하지마라!
  - 기술적인 이해가 충분하지 않으면 반드시 쓰레기같은 설계를 하게 된다.
  - 기술적 이해가 충분해졌을때 설계를 한다.
  - 기술적 이해가 깊어졌다면 설계는 자동으로 된다.
- 철학은 필요하다.
  - 무엇에 주안점을 둘 것인가? - 자체 엔진개발의 묘미

설계와 구조

- 정말 커다란 그림만 그려놓는다.
  - 게임 exe 하나 + 엔진 DLL 3개 - 이 정도.
- 기능 구현을 해보면 내가 한 설계 따윈 강아지 간식도 못된다는걸 곧 깨닫게 된다.
- 기능 구현을 한 후에는 이런저런 문제가 있음을 깨닫게 된다.
- 문제를 다 해결하고나서 한걸음 떨어져서 바라보자.
  - 어떤 구조를 가져야 할지 윤곽이 보인다.
  - 뒤집어 엎는다!

기능 구현 중에는 코드를 이쁘게 짜려고 하지 않는다.

- 기능 구현 최우선
- 디버깅
- 정말 잘 작동하나?
- 현자타임 -> 리팩토링

유지보수 원칙

- Debug / release /x86/x64 빌드에 문제가 없어야한다.
- Debug / release /x86/x64 빌드의 실행에 문제가 없어야한다.
- 종료시점에서 CRT heap, D3d resource의 누수가 없어야한다.
  - D3D Debug runtime테스트.
  - 주기적으로 gflags나 Application verifier로 heap 체크.
- 어차피 프로세스가 종료될때 모든 리소스는 반환되는데 뭐하러?
  - 완벽하게 종료해도 버그가 있을 수 있는데 종료 시점에서 무결성을 검증할 수 없으면 버그가 없다고 장담할 수 있을까?
  - 버그는 어떻게 찾지? 티 안나면 버그 없는건가?

유지보수 원칙

- 특정 코드그룹의 빌드 시간이 너무 긴가?
  - 오래 걸리는 코드를 찾아서 별도 프로젝트(바이너리)로 분리한다.