장진성

DirectX11 포트폴리오

프로젝트 개요

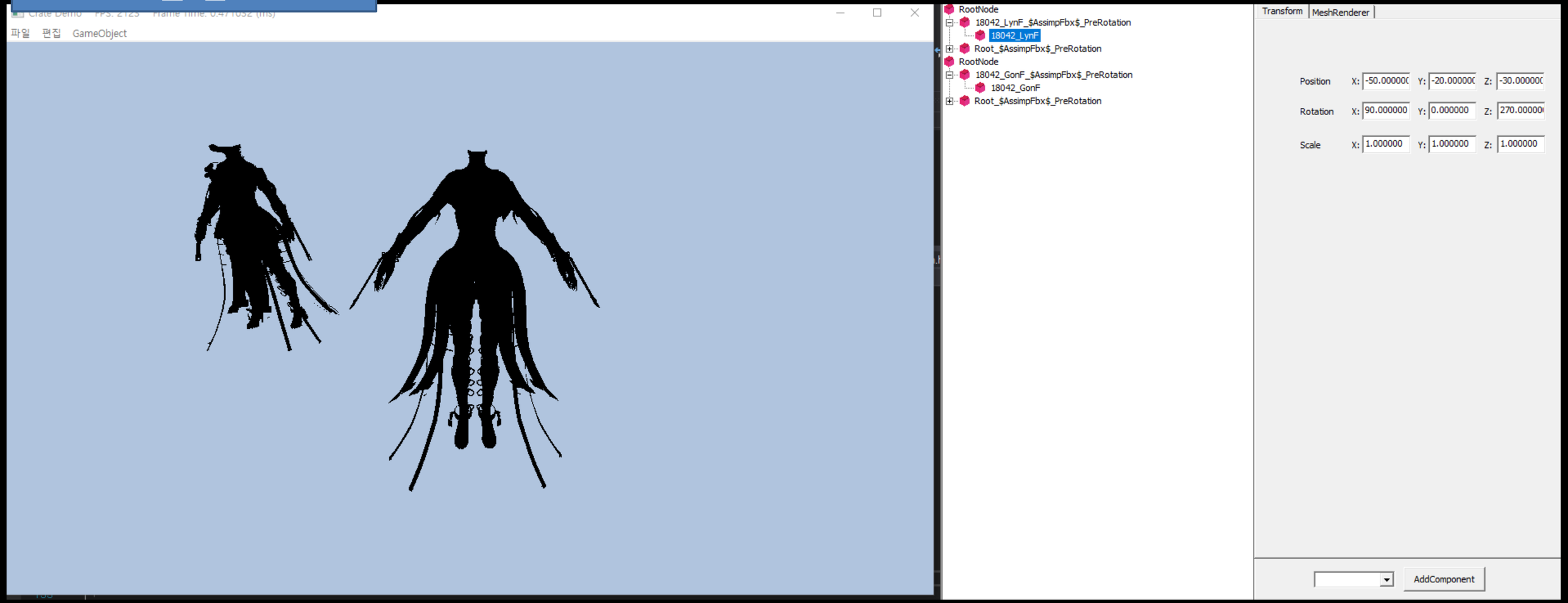
- 목표

- 유니티를 참고한 컴포넌트 기반의 렌더링 엔진 설계
- 쉐이더를 이용해 게임개발에 필요한 효과들 구현
- 각 기능들을 쉽게 사용할 수 있는 툴을 포함한 자체엔진 개발

- 목적

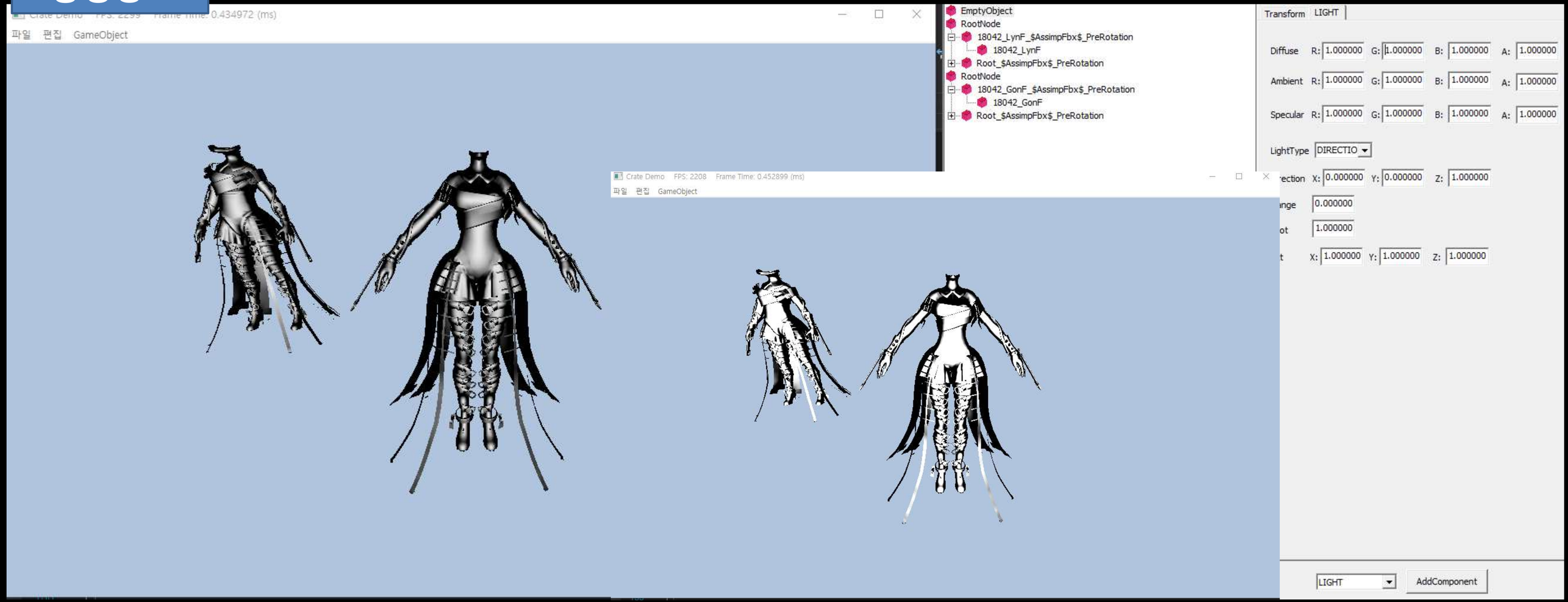
- 각 쉐이더 단계를 활용함으로써 공간과 렌더링파이프라인에 대한 이해
- 상용엔진에서 사용되는 기능들의 원리를 이해
- 프레임워크 설계 경험과 실력향상

- FBX 모델 임포트



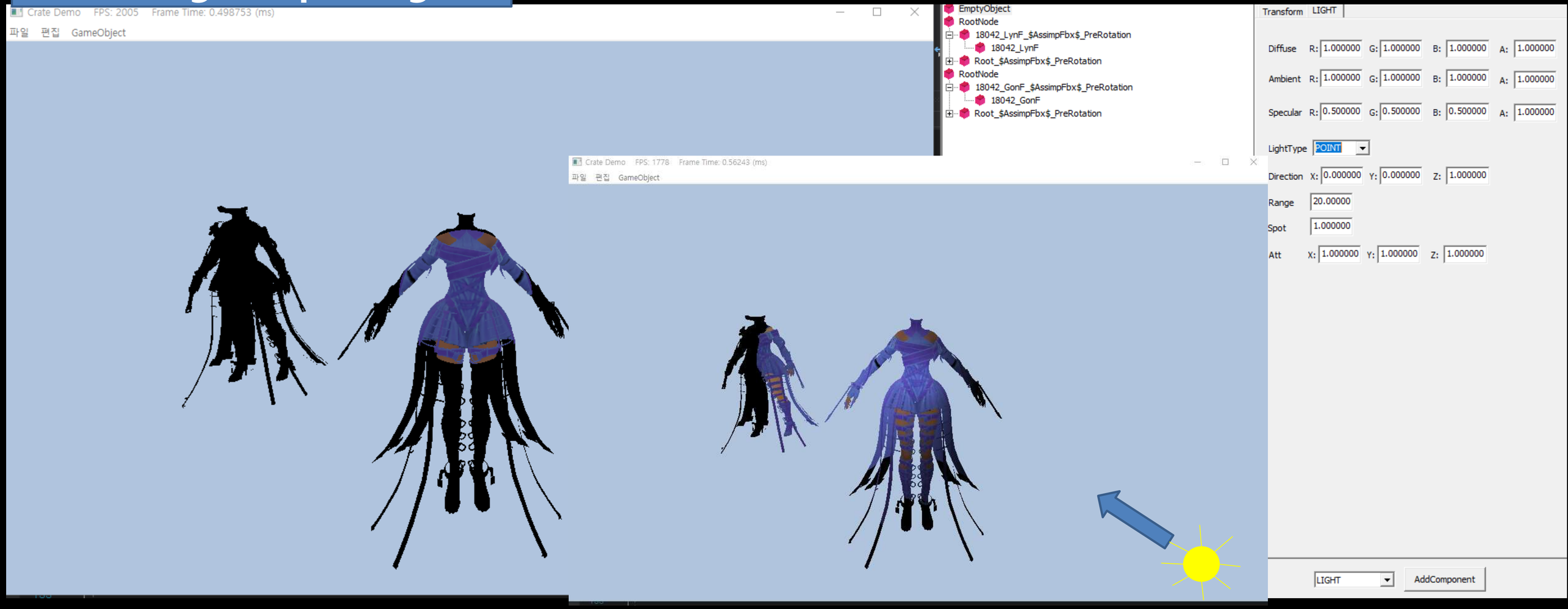
- Assimp Library로 fbx 파일을 읽어와 렌더링 합니다.
- 오브젝트는 컴포넌트의 집합으로 정의됩니다. 오브젝트는 Transform 컴포넌트를 기본으로 가지고 다른 컴포넌트들을 추가 할 수 있습니다.
- 새로운 기능을 구현할 때 Component 클래스를 상속한 클래스를 만들고, 추가하면 작동하도록 설계하였습니다.
- 렌더링은 MeshRenderer 컴포넌트를 오브젝트에 추가하여 실행됩니다.

- 평행광



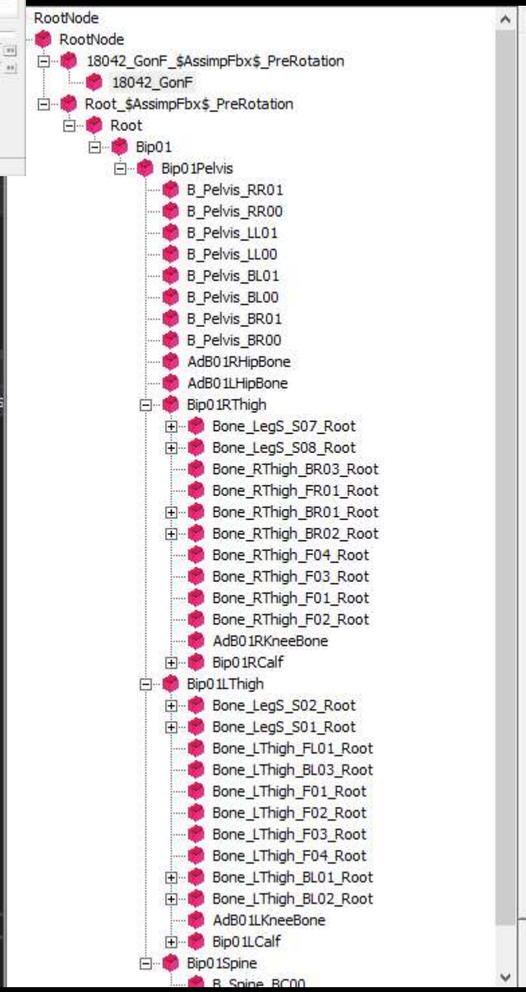
- 빈 오브젝트를 생성 후 Light Component를 추가 해 Directional Light를 조명으로 사용하였습니다.
- 좌측 사진은 Directional Light 1개, 우측 사진은 Directional Light 3개를 사용한 모습입니다.

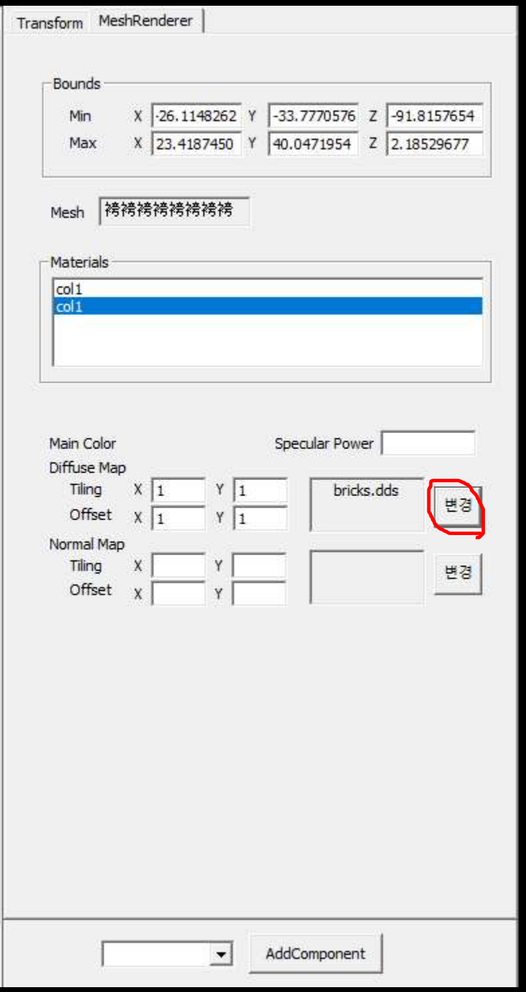
- Point Light, Spot Light



- Light 컴포넌트에서 LightType을 Point, Spot Light로 변경 할 수 있습니다.
- 조명의 방향, 범위 등 세부설정이 가능합니다.
- 좌측 사진은 Point Light, 우측 사진은 Spot Light를 사용한 모습입니다.

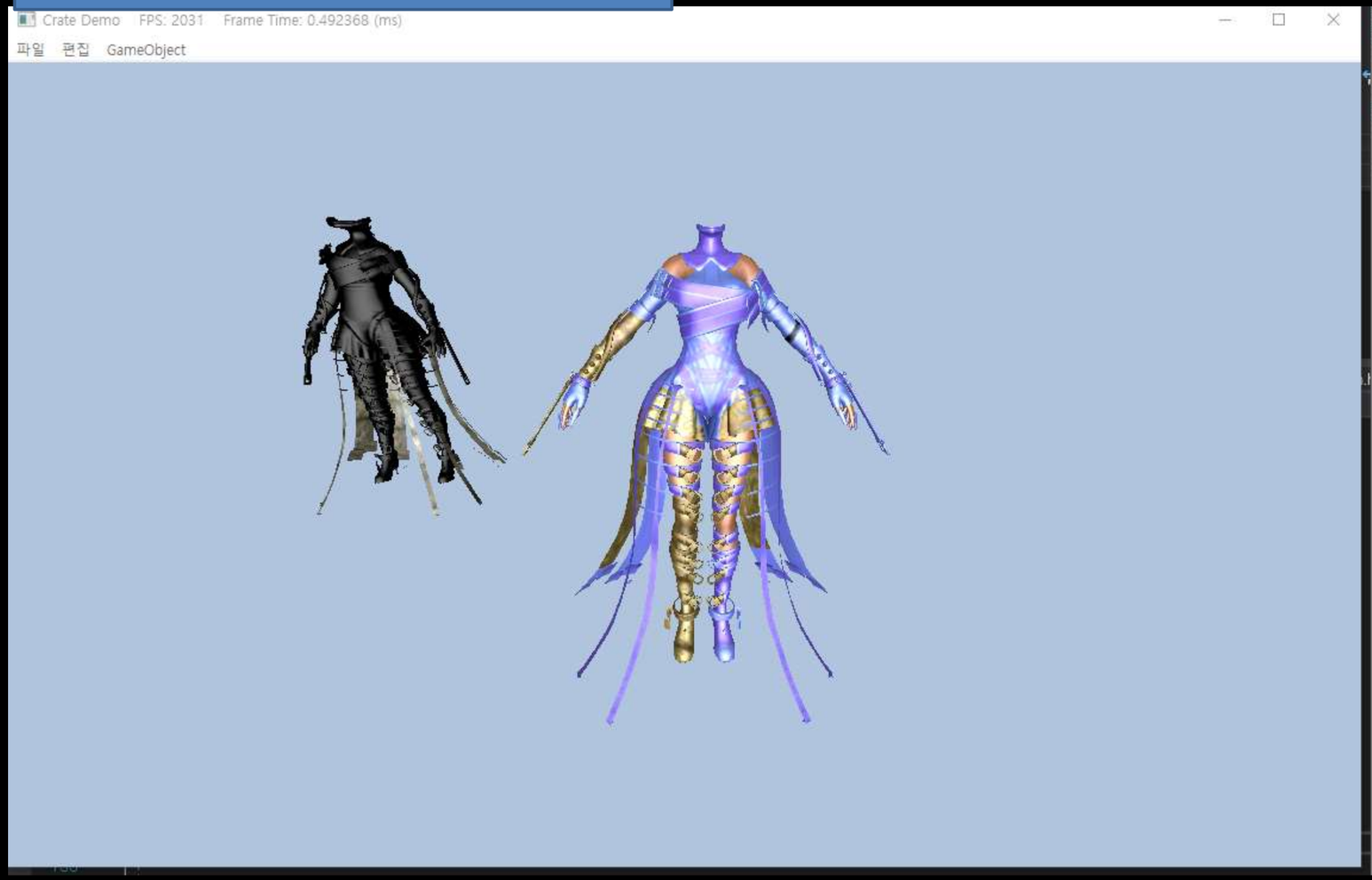
- 텍스쳐 적용

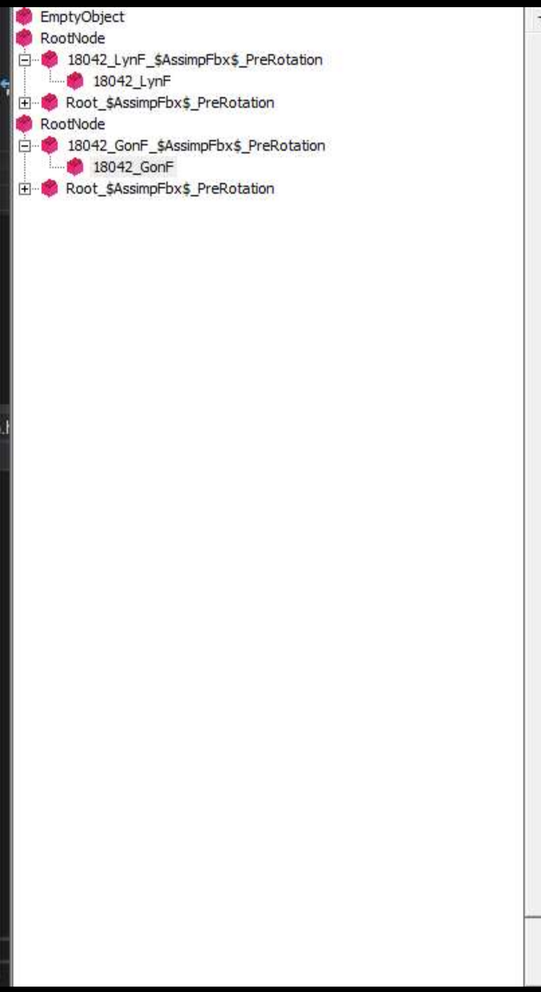


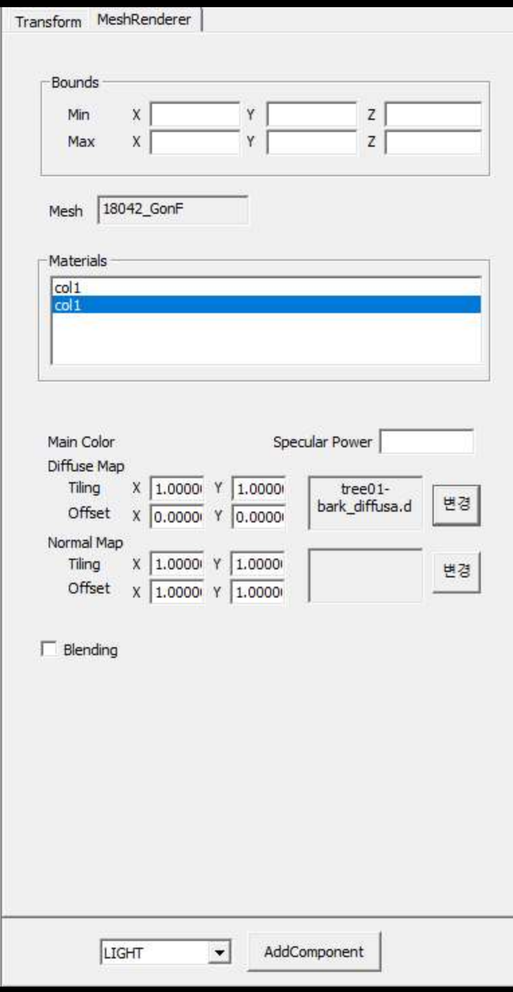


- 텍스쳐 파일을 불러와 Diffuse Map을 변경 할 수 있습니다.
- 명령패턴을 이용해 실행되며 Accelrator 단축키 설정을 이용해 Ctrl+z를 누르면 명령취소가 되게 구현했습니다.

- Material 개별 텍스쳐 적용







- 하나의 Mesh는 여러개의 Material로 구성될 수 있고 각각의 Material별로 Diffuse Map을 변경 할 수 있습니다.

렌더링 구조

파란색은 클래스명 / 검정색은 멤버변수 / 초록색은 멤버함수 입니다.

- 프로그램 설명

1. 기본적인 렌더링, 업데이트와 같은 게임로직은 Scene 클래스에서 이루어집니다.

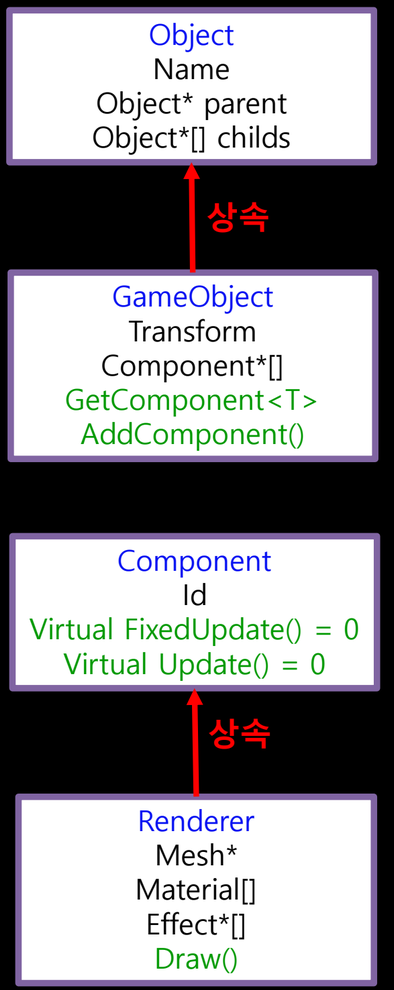
2. Scene에 있는 오브젝트들을 렌더링, 업데이트 합니다.

3. 오브젝트는 컴포넌트의 집합으로 이루어집니다.

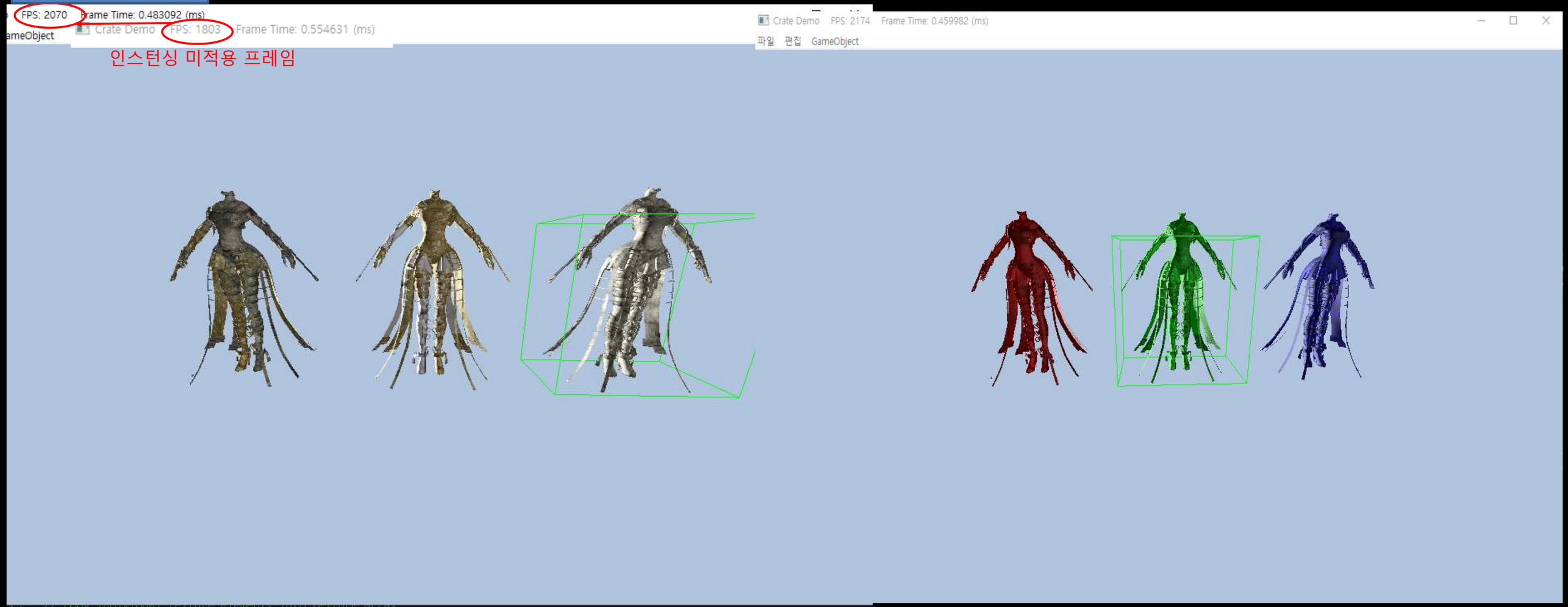
4. 컴포넌트 매니저에서 모든 컴포넌트를 관리합니다.

5. 컴포넌트 매니저의 모든 컴포넌트를 업데이트, 렌더링 합니다.

주요 클래스



- 인스턴싱

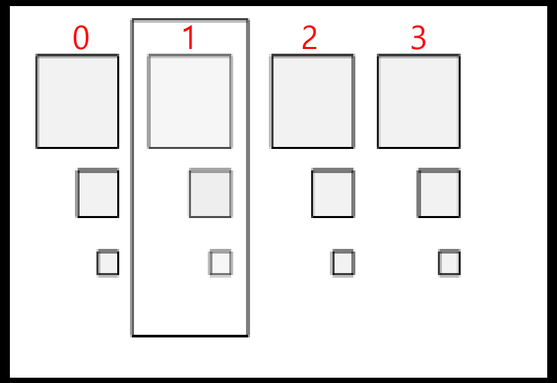


- 하드웨어 인스턴싱을 적용해 같은 메쉬를 사용하는 오브젝트를 한번의 그리기 호출로 렌더링하도록 하였습니다.
- 텍스쳐배열을 사용해 각 오브젝트, 각 메테리얼마다 다른 텍스쳐를 적용할 수 있습니다.

- 왼쪽사진은 3개의 텍스쳐를 사용해 3개의 오브젝트(6개의 메테리얼)에 조합해 적용한 모습입니다.
- 오른쪽사진은 인스턴싱버퍼에 위치,색상 데이터를 추가해 각기 다른 색상으로 렌더링 한 모습입니다.

- Instancing 코드1

```
//매개변수로 주어진 idx번째 arraySlice를 변경
for (UINT mipLevel = 0; mipLevel < minMipLevels; ++mipLevel)
{
    D3D11_MAPPED_SUBRESOURCE mappedTex2D;
    HR(context->Map(srcTex, mipLevel, D3D11_MAP_READ, 0, &mappedTex2D));

    context->UpdateSubresource(textureArr,
        D3D11CalcSubresource(mipLevel, destIdx, textureArrDesc.MipLevels),
        0, mappedTex2D.pData, mappedTex2D.RowPitch, mappedTex2D.DepthPitch);

    context->Unmap(srcTex, mipLevel);
}
```



- destIdx가 1이라면 텍스쳐배열의 1번째 ArraySlice를 수정

- 인스턴싱에서 같은 메쉬를 사용하는 오브젝트들은 같은 텍스쳐배열을 사용합니다.
- N번째 오브젝트의 텍스쳐를 변경했다면, 텍스쳐배열에서 N번째 텍스쳐(Array Slice)를 새 텍스쳐로 수정하는 코드입니다.

- Instancing 코드1

```cpp
void Renderer::InstancingUpdate()
{
    //instancing 중이 아니면 return
    if (!GetInstancing() || mesh == nullptr)
        return;

    //세계행렬 업데이트
    mesh->InstancingDatas[m_instancingIdx]->world = transform->m_world;
    //역전치행렬 업데이트
    XMStoreFloat4x4(&mesh->InstancingDatas[m_instancingIdx]->worldInvTranspose,
        MathHelper::InverseTranspose(
            XMLoadFloat4x4(&mesh->InstancingDatas[m_instancingIdx]->world)));
    //rgb 업데이트
    mesh->InstancingDatas[m_instancingIdx]->color = m_color;
    //현재 렌더러를 알려주는 인덱스
    mesh->InstancingDatas[m_instancingIdx]->RendererIdx = m_instancingIdx;

    //이번 프레임에 렌더링할 오브젝트로 등록
    mesh->enableInstancingIndexes.push_back(m_instancingIdx);
}
```

- 프레임마다 오브젝트의 인스턴싱 데이터를 업데이트 합니다. (위치, 행렬 등)

- 절두체 선별을 통해 선별 된 오브젝트만 해당 함수를 호출해 업데이트 합니다.

```cpp
void Mesh::InstancingUpdate(ID3D11DeviceContext* context)
{
    if (m_InstanceBuffer == nullptr)
        return;

    D3D11_MAPPED_SUBRESOURCE mappedData;
    context->Map(m_InstanceBuffer, 0, D3D11_MAP_WRITE_DISCARD, 0, &mappedData);

    InstancingData* dataView = reinterpret_cast<InstancingData*>(mappedData.pData);

    for (int i = 0; i < enableInstancingIndexes.size(); ++i)
    {
        if(InstancingDatas[enableInstancingIndexes[i]] != nullptr)
            dataView[i] = *InstancingDatas[enableInstancingIndexes[i]];
    }

    context->Unmap(m_InstanceBuffer, 0);
}
```

- 업데이트 한 인스턴싱 데이터들을 이용해 인스턴싱 버퍼에 데이터를 쓰는 함수입니다.

- 해당 메쉬를 인스턴싱 렌더링할 때 한 번 호출 됩니다.

- 반직선 교차 선택 코드

```cpp
Renderer * RayPicking::NearestOfIntersectRayAABB(D3D11_VIEWPORT* viewPort ,
    const std::vector<Renderer*>& objects, Camera * camera, float sx, float sy)
{
    Renderer* result = nullptr;

    XMVECTOR rayOrigin;
    XMVECTOR rayDir;

    //스크린좌표 -> 시야공간 반직선 계산
    ScreenToViewRay(&rayOrigin, &rayDir, sx, sy, viewPort, &camera->Proj());

    float tMin = INT_MAX;
    for (auto elem : objects)
    {
        std::pair<XMVECTOR, XMVECTOR> localRay;
        //시야공간 반직선 -> 국소공간 반직선 변환
        localRay = ViewToLocalRay(&rayOrigin, &rayDir, &camera->View(),
            &XMLoadFloat4x4(elem->GetTransform()->GetWorld()));

        //반직선과 AABB 교차판정
        float t = 0.0f;
        if (XNA::IntersectRayAxisAlignedBox(localRay.first, localRay.second, &elem->GetMesh()->GetAABB(), &t))
        {
            //교차했을 때 가장 가까운 오브젝트를 찾는다.
            if (t < tMin)
            {
                tMin = t;
                result = elem;
            }
        }
    }
    //교차하는 것이 없다면 nullptr 반환
    return result;
```

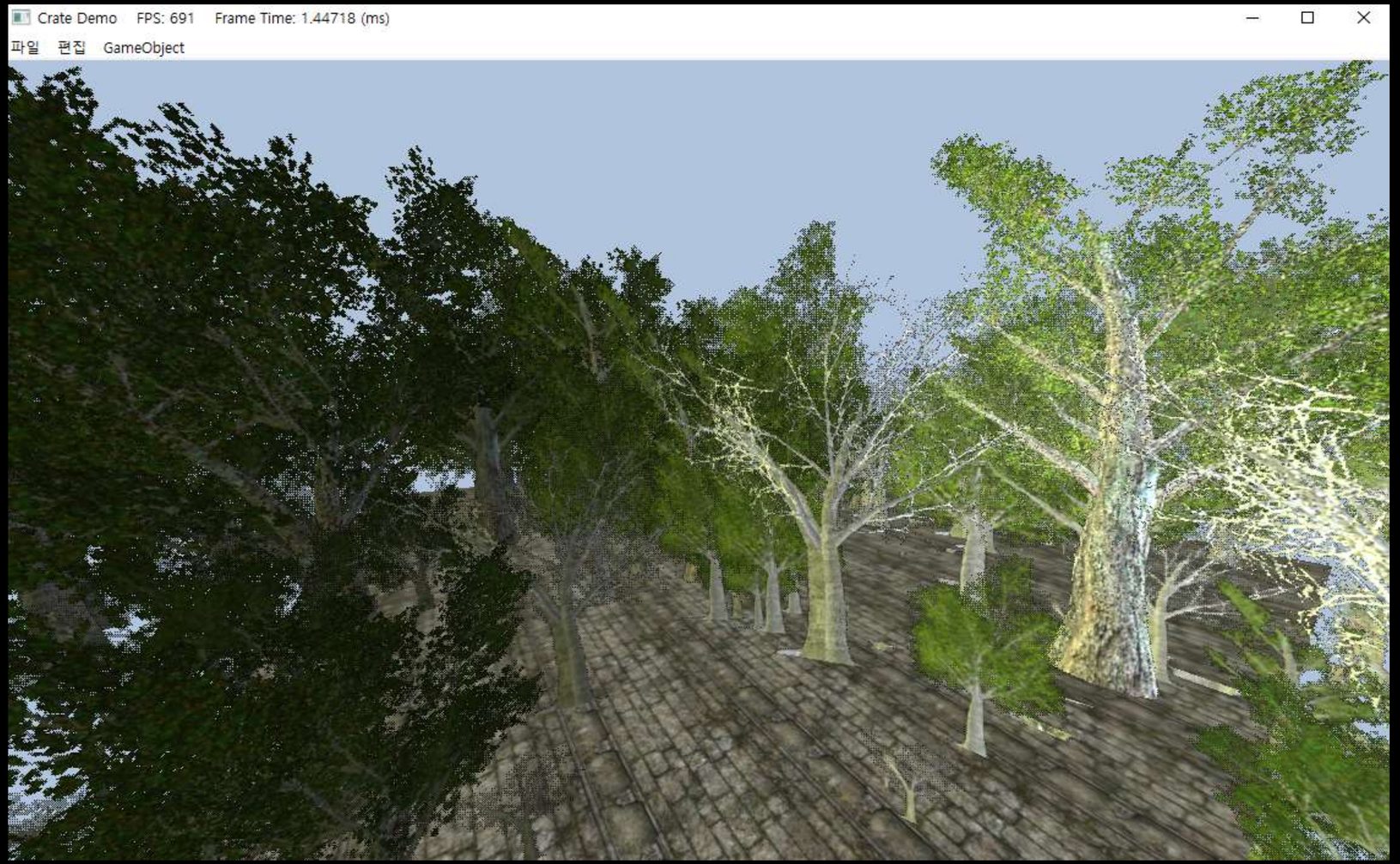
```cpp
class RayPicking
{
public:
    //스크린좌표 -> 시야공간에서의 반직선을 계산하는 함수
    static void ScreenToViewRay(XMVECTOR* rayOrigin, XMVECTOR* rayDir, float sx, float sy, D3D11_VIEWPORT * viewPort, XMMATRIX * projection
    //시야공간 반직선 -> 국소공간 반직선 변환 / 반환값 origin,dir
    static std::pair<XMVECTOR, XMVECTOR> ViewToLocalRay(XMVECTOR* rayOrigin, XMVECTOR* rayDir, XMMATRIX* view, XMMATRIX* world);
    //반직선교차 판정 후, 가장 가까운 물체를 반환
    static Renderer* NearestOfIntersectRayAABB(D3D11_VIEWPORT* viewPort,
        const std::vector<Renderer*>& objects,Camera* camera, float sx, float sy);

};
```

- 사용자가 클릭 한 뷰포트의 좌표를 입력받고 반직선을 쏴 오브젝트를 선택합니다.

- 반직선과 오브젝트의 AABB 교차판정을 수행하며, 교체한 오브젝트가 여러 개라면 가장 가까운 오브젝트를 선택하는 함수입니다.

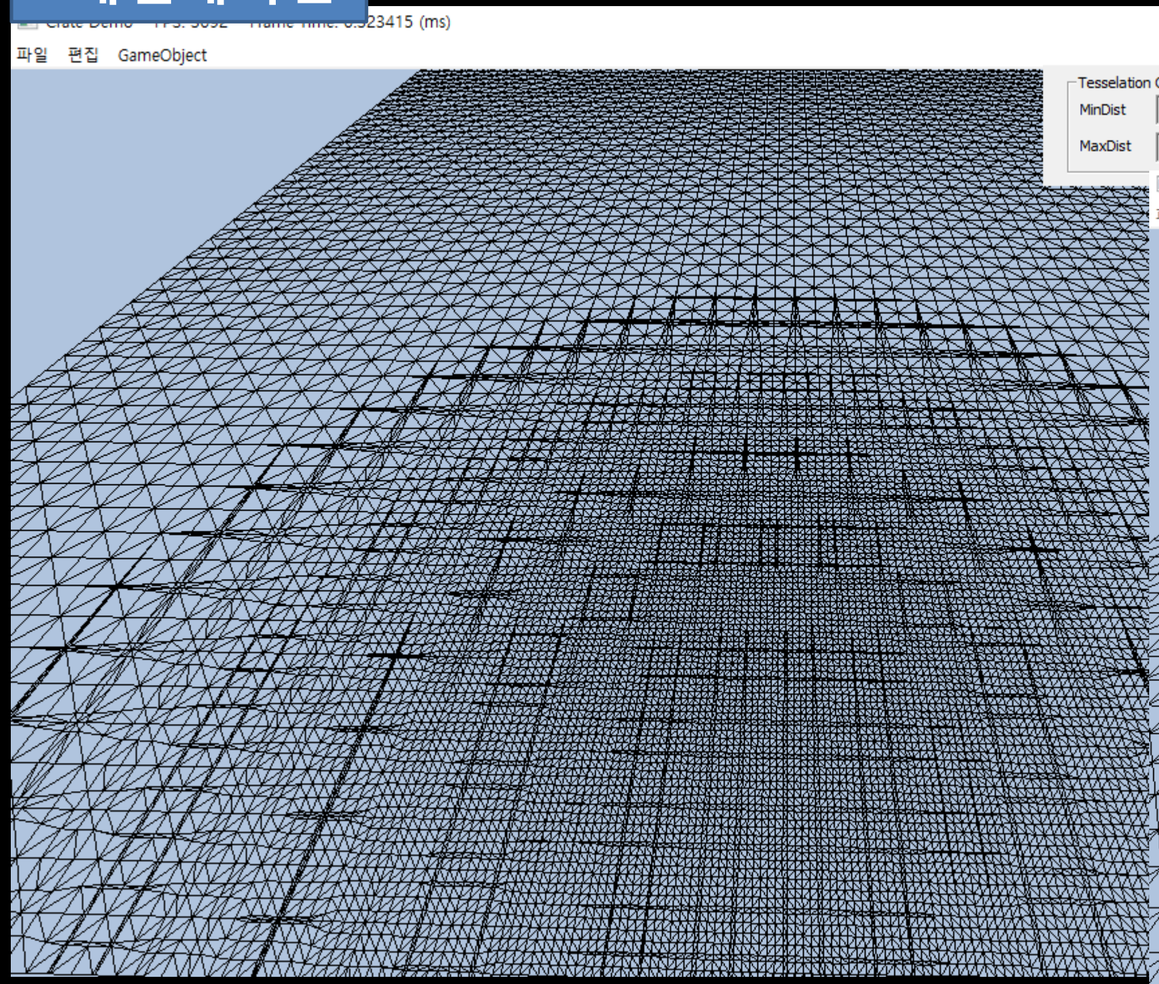
- 현재 선택된 오브젝트는 AABB를 렌더링합니다.

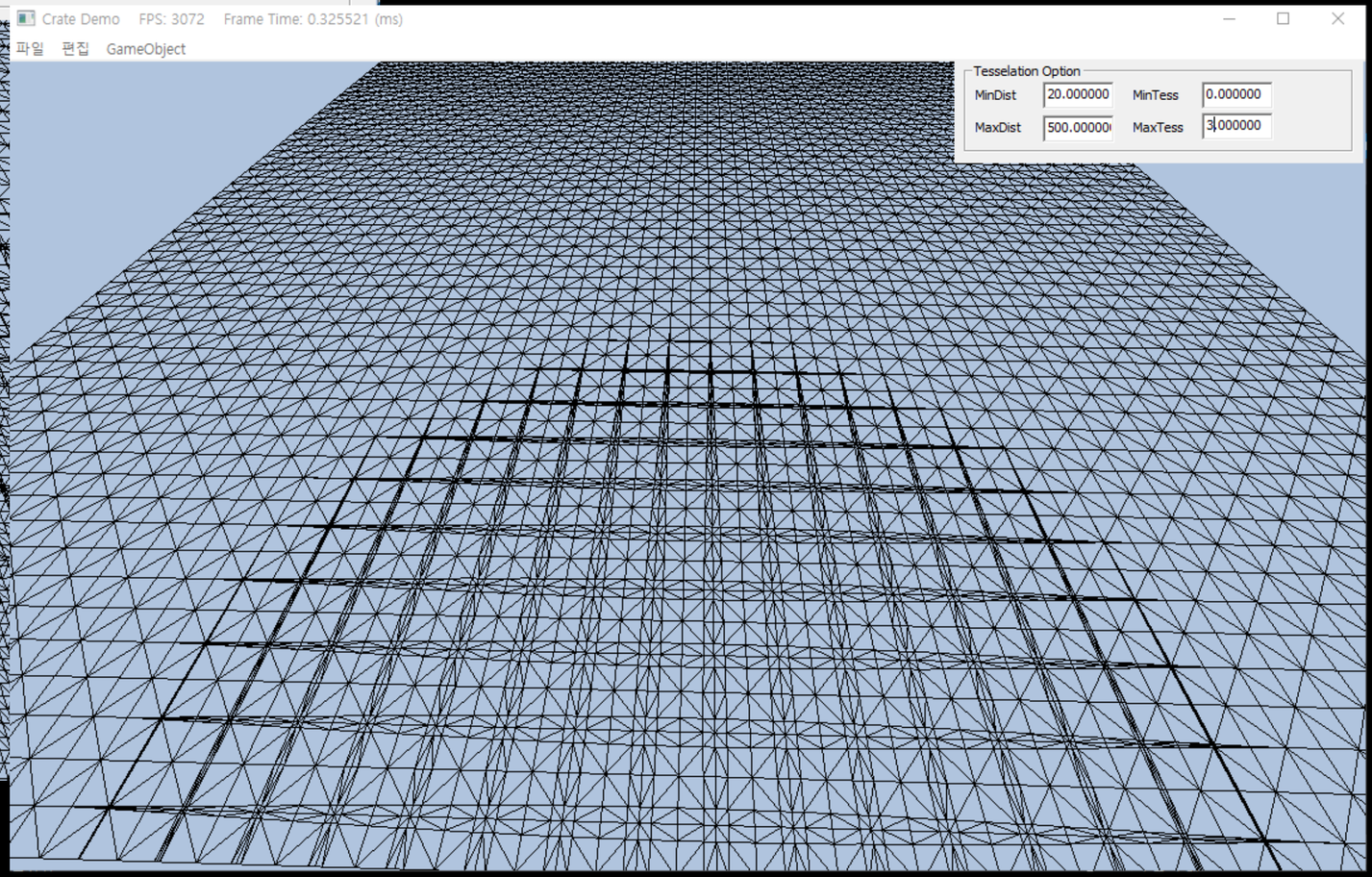
- 기하쉐이더



- 기하 쉐이더를 이용해 하나의 정점을 사각형으로 확장하고 텍스쳐를 입혀 나무를 구현하였습니다.
- 평면 사각형이 항상 카메라를 바라보게하여 사용자에게 나무처럼 보이게 합니다.

- 테셀레이션



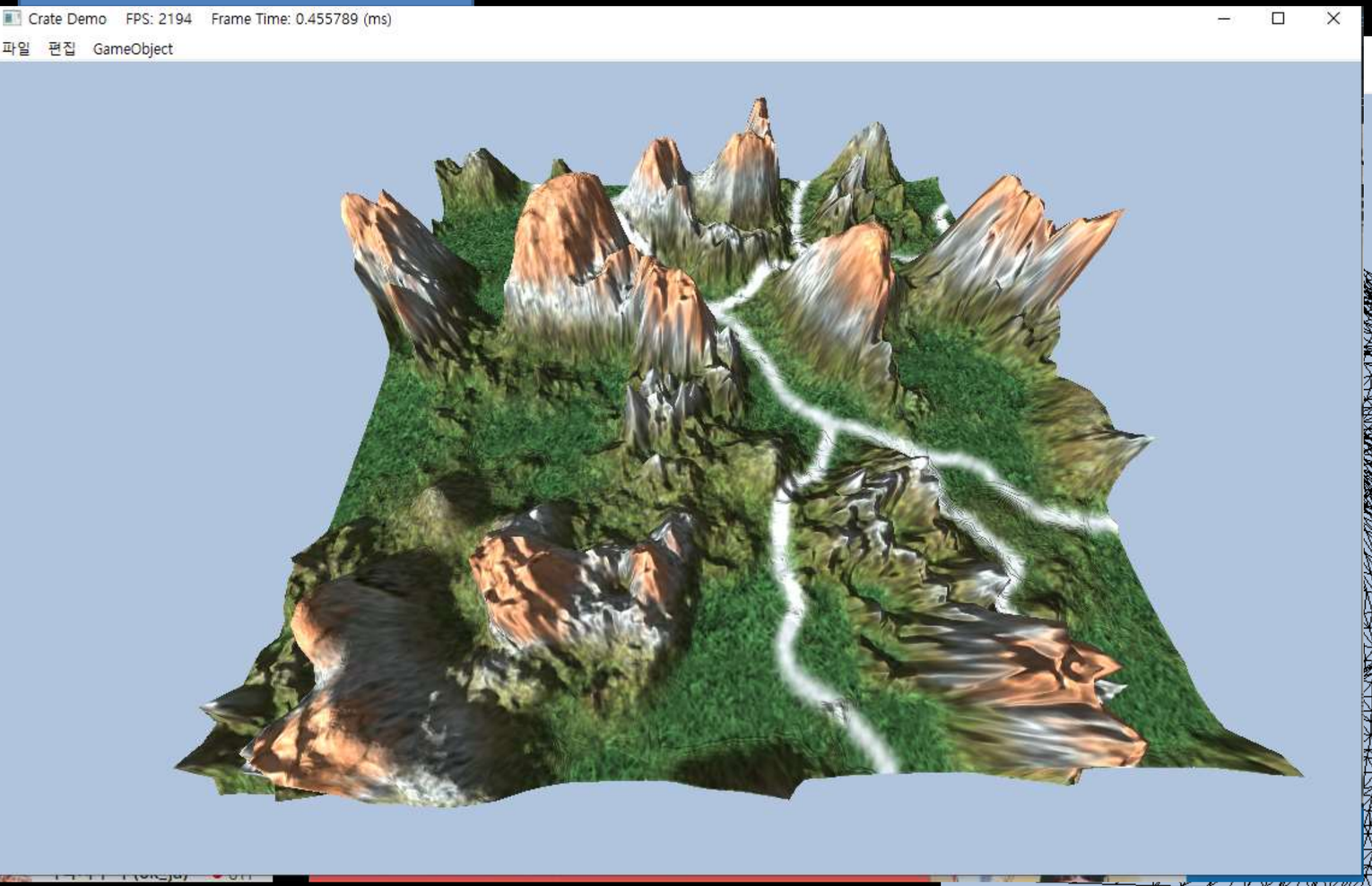


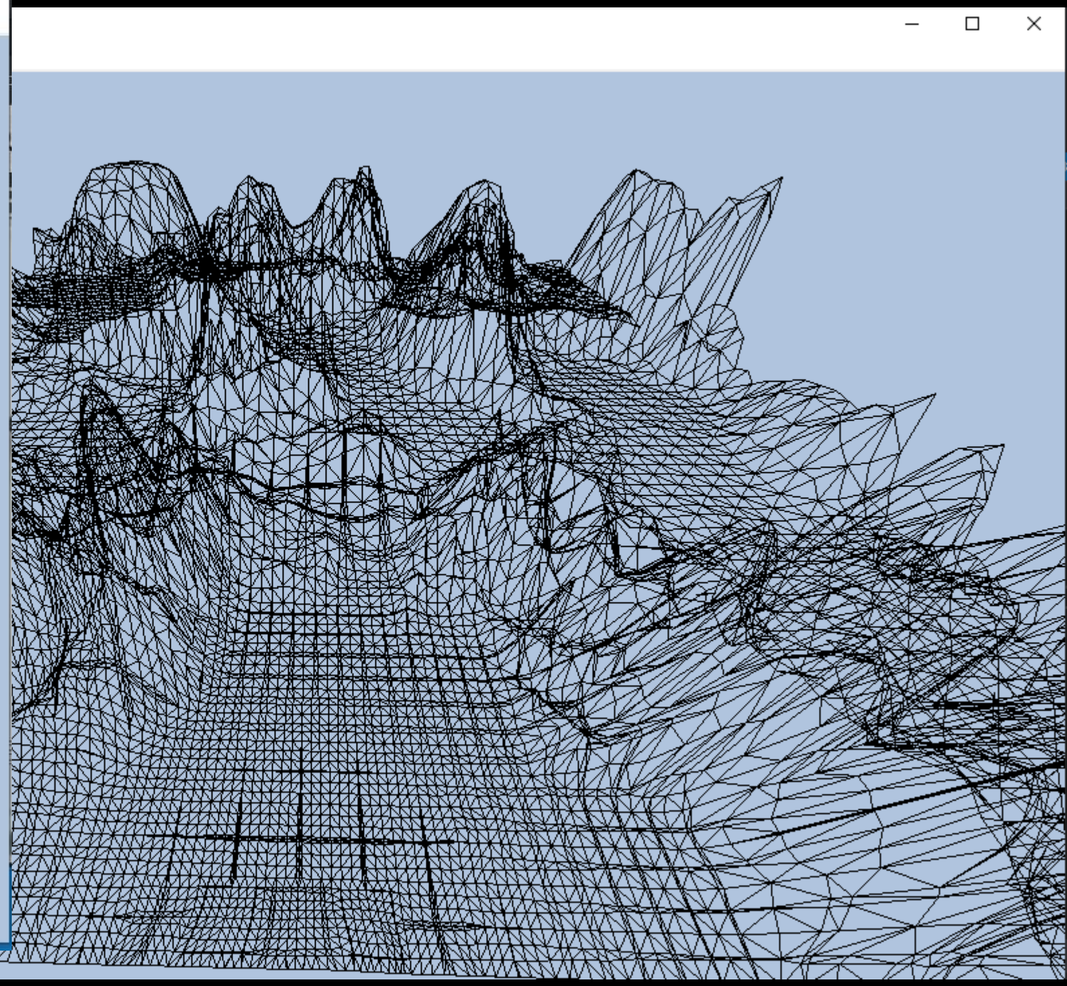
- 테셀레이션을 이용해 사각형 패치를 분할하였습니다. 카메라와 패치의 거리에 따라 테셀레이션 수준을 조정합니다.
- 카메라와 가까운 곳이 더 많이 분할되는 것이 나타납니다.
- 설정한 테셀레이션 계수 N에 따라 패치의 각 변을 최대 2^n만큼 분할합니다.
  왼쪽은 계수6, 오른쪽은 계수3을 적용한 모습입니다.

- 테셀레이션 지형

- 지형에 최대 5가지의 텍스쳐를 입히고 혼합 하도록 구현했습니다.
- BlendMap 텍스쳐의 해당 픽셀 r,g,b,a 색상값을 각 4개의 레이어텍스쳐 기여도로 사용해 보간하여 혼합합니다.

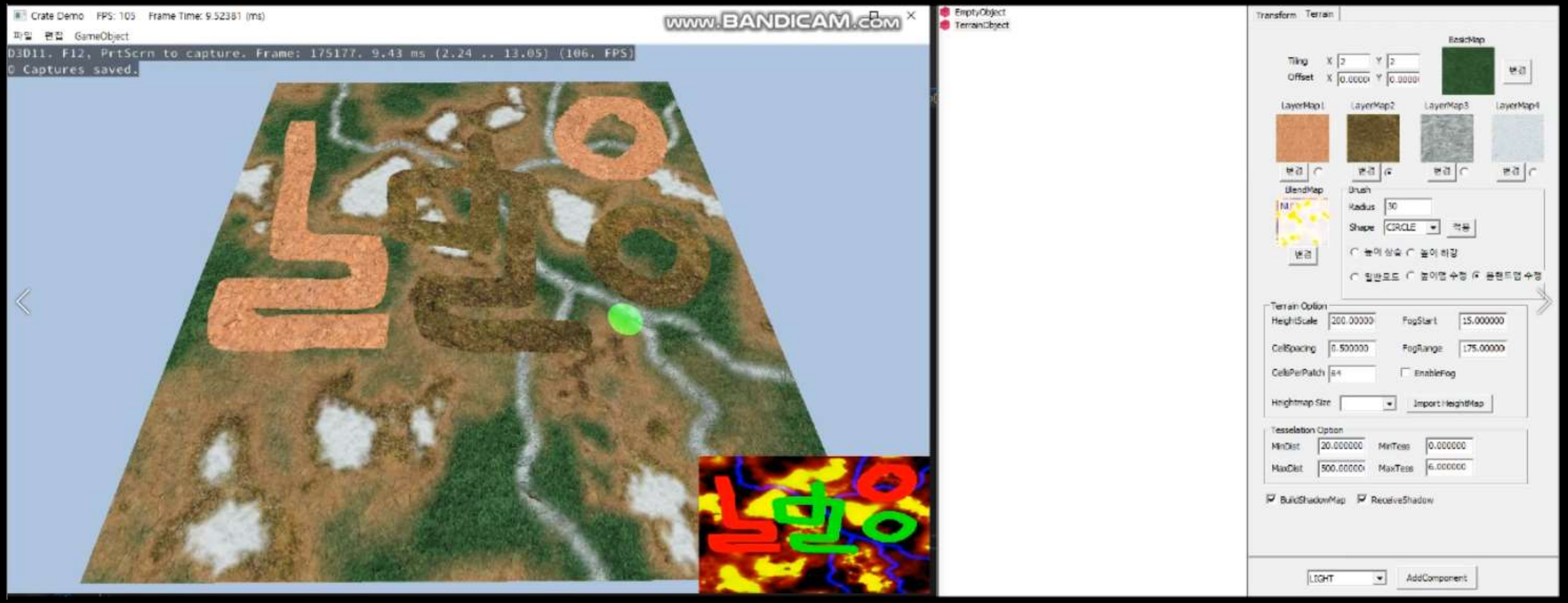
- 테셀레이션 지형





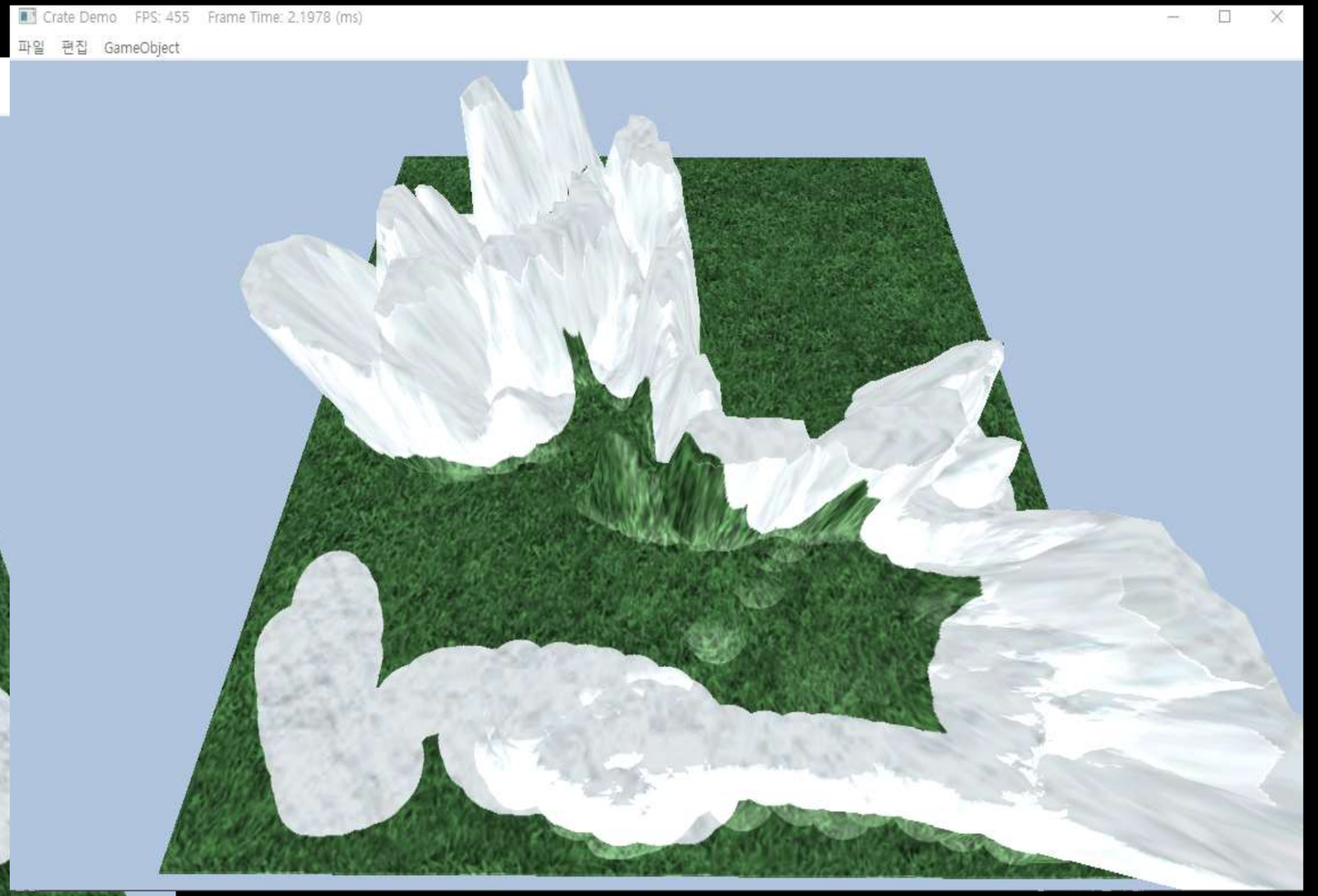
- 지형의 높이가 저장된 바이너리 파일을 읽어 텍스쳐를 생성하고 높이맵으로 사용하였습니다.
- Domain Shader에서 겹선형보간으로 텍스쳐 좌표를 구하고 높이맵의 해당좌표 픽셀값을 높이로 사용합니다.

- 테셀레이션 블랜드맵 수정



- Ray Picking을 이용해 마우스 위치에 브러쉬 모양을 그립니다.
- 마우스 좌클릭을 통해 브러쉬 모양에 따라 블랜드맵을 수정합니다.
- 4개의 레이어중 하나를 선택해 그릴 수 있습니다.

- 테셀레이션 높이맵 수정



- 마우스 좌클릭을 통해 브러쉬 모양에 따라 높이맵을 수정합니다.
- 높이맵 상승, 하강을 선택 할 수 있으며 지형을 조절합니다.

- 높이맵 수정 코드1

```
//렌더링 자원으로 사용한 heightMap 텍스쳐를 얻어옴.

D3D11_MAPPED_SUBRESOURCE mappedData;
HR(m_texMgr.m_context->Map(m_hmapTex, D3D11CalcSubresource(0,0,1), D3D11_MAP_WRITE_DISCARD, 0, &mappedData));
HALF* heightMapData = reinterpret_cast<HALF*>(mappedData.pData);
D3D11_TEXTURE2D_DESC heightmapDesc;
m_hmapTex->GetDesc(&heightmapDesc);

UINT width = heightmapDesc.Width;

//브러쉬의 중심에서 정사각형 범위를 검사
float leftTopX, leftTopZ;
float rightBottomX, rightBottomZ;
//세계공간에서 radius 범위만큼 이동한 좌표 -> heightmap에 사상되는 좌표
GetLocalPosition(x-m_brush->radius, z+m_brush->radius, &leftTopX, &leftTopZ);
GetLocalPosition(x + m_brush->radius, z - m_brush->radius, &rightBottomX, &rightBottomZ);
```

- Ray Picking 으로 얻은 좌표를 중심으로 정사각형의 좌상단, 우하단 점 위치를 구합니다.
- 구한 두개의 위치를 세계공간에서 Terrain의 HeightMap 좌표로 변환합니다.

```
switch (m_brush->shape)
{
case BrushShape::SQUARE:
    for (int i = leftTopZ; i <= rightBottomZ; ++i)
    {
        for (int j = leftTopX; j <= rightBottomX; ++j)
        {
            idx = i * width + j;
            addVal = m_terrainData.HeightScale * RAISEDELTA;
            indices.push_back({ i,j,idx });

            //최대 높이를 넘을수 없게 함
            if (m_modifyMapOption == 0) //높이를 올림
            {
                mHeightmap[idx] = min(m_terrainData.HeightScale,
                    mHeightmap[idx] + addVal);
            }
            else //높이를 내림
            {
                mHeightmap[idx] = max(0,
                    mHeightmap[idx] - addVal);
            }

            //heightMapData[idx] = XMConvertFloatToHalf(mHeightmap[idx]);
        }
    }
    break;
```

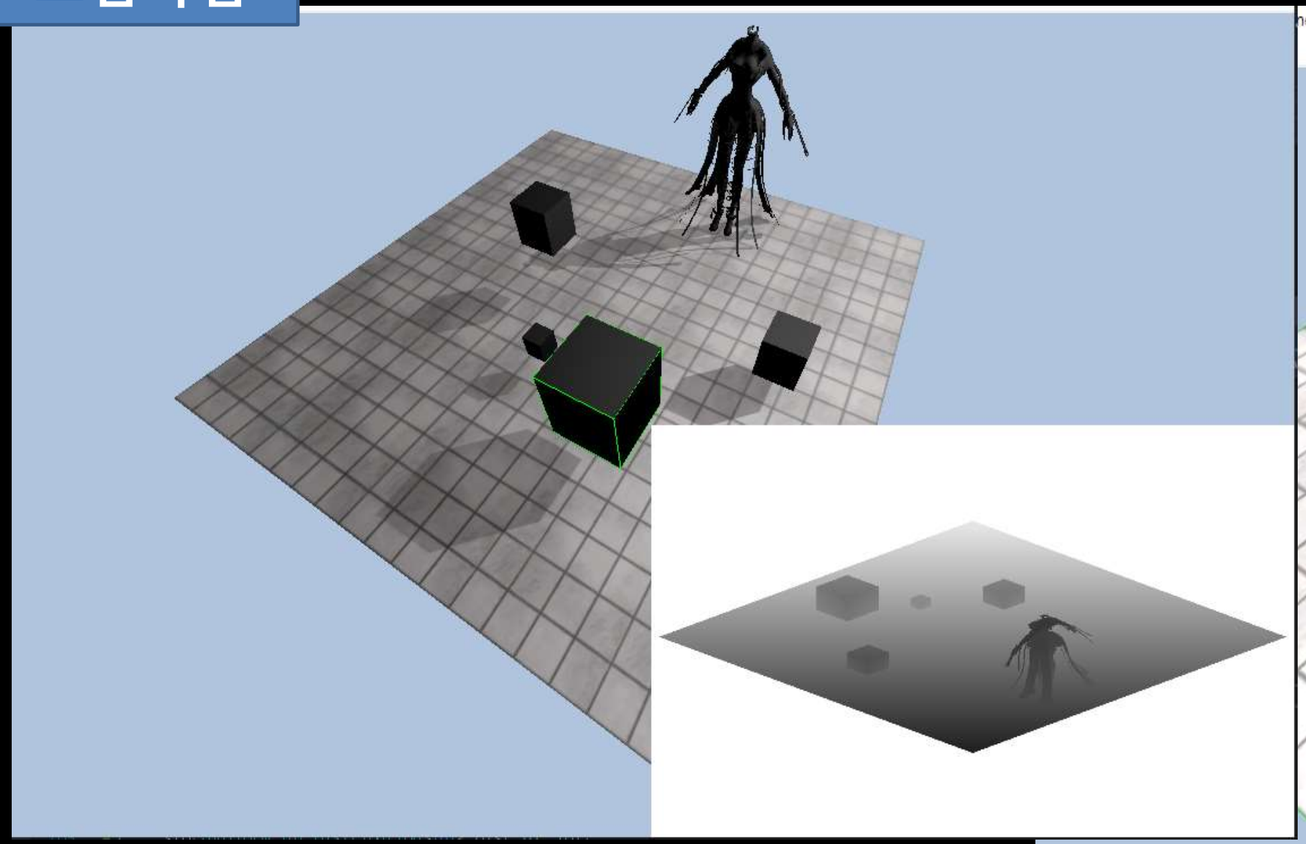
- 앞서 구한 사각형의 좌상단, 우하단의 좌표를 행과 열의 최소,최대치로 사용해 사각형 모양으로 검사하고, 인덱스로 변환해 텍스쳐에 접근합니다.
- 설정한 비율의 값만큼 높이맵을 상승,하강합니다.

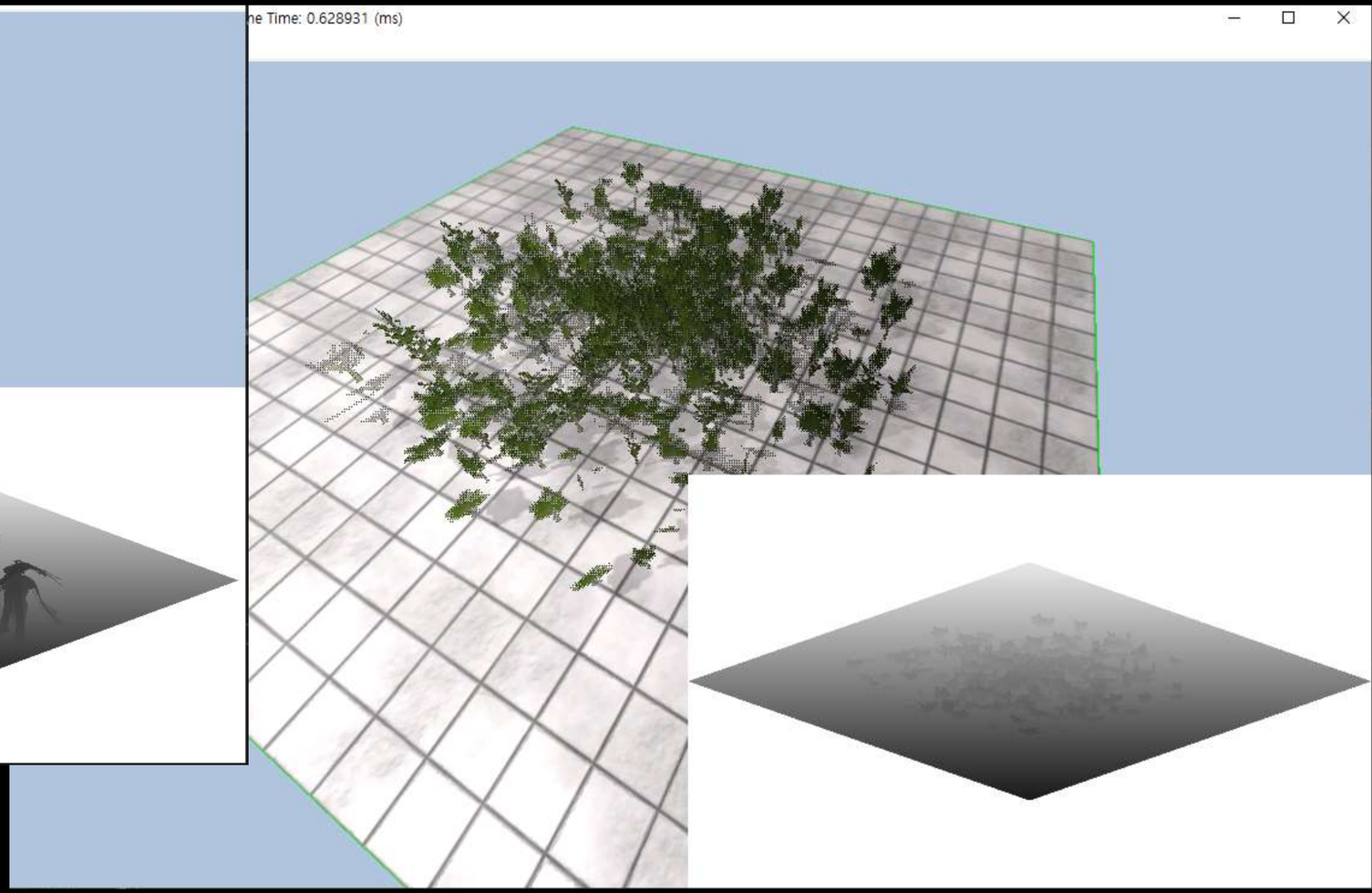
- 높이맵 수정 코드2

```
case BrushShape::CIRCLE:
    float yCoefficient = 0.0f;
    for (int i = leftTopZ; i <= rightBottomZ; ++i)
    {
        for (int j = leftTopX; j <= rightBottomX; ++j)
        {
            idx = i * width + j;

            //중심과의 거리
            float dist = sqrtf(pow((centerTexRow - i), 2) + pow((centerTexCol - j), 2));
            if (dist > radiusTex)
                continue;

            indices.push_back({ i,j,idx });
            //거리에 따라서 2차함수 모양으로 높이를 증가시킬 계수를 구함(0~1)
            yCoefficient = -(dist*dist) / (radiusTex*radiusTex) + 1;
            addVal = m_terrainData.HeightScale * RAISEDELTA * yCoefficient;
            if (m_modifyMapOption == 0)
            {
                mHeightmap[idx] = min(m_terrainData.HeightScale,
                    mHeightmap[idx] + addVal);
            }
            else
            {
                mHeightmap[idx] = max(0,
                    mHeightmap[idx] - addVal);
            }
            //최대 높이를 넘을수 없게 함

            //heightMapData[idx] = XMConvertFloatToHalf(mHeightmap[idx]);
        }
    }
    break;
```

- 사각형 범위와 마찬가지로 검사하지만, 중심과의 거리가 반지름보다 큰 좌표는 무시함으로써 원 모양으로 수정합니다.
- 2차 함수를 이용해 해당 좌표와 중심과의 거리에 따라 (-x*x) 2차함수 모양으로 상승시킵니다.

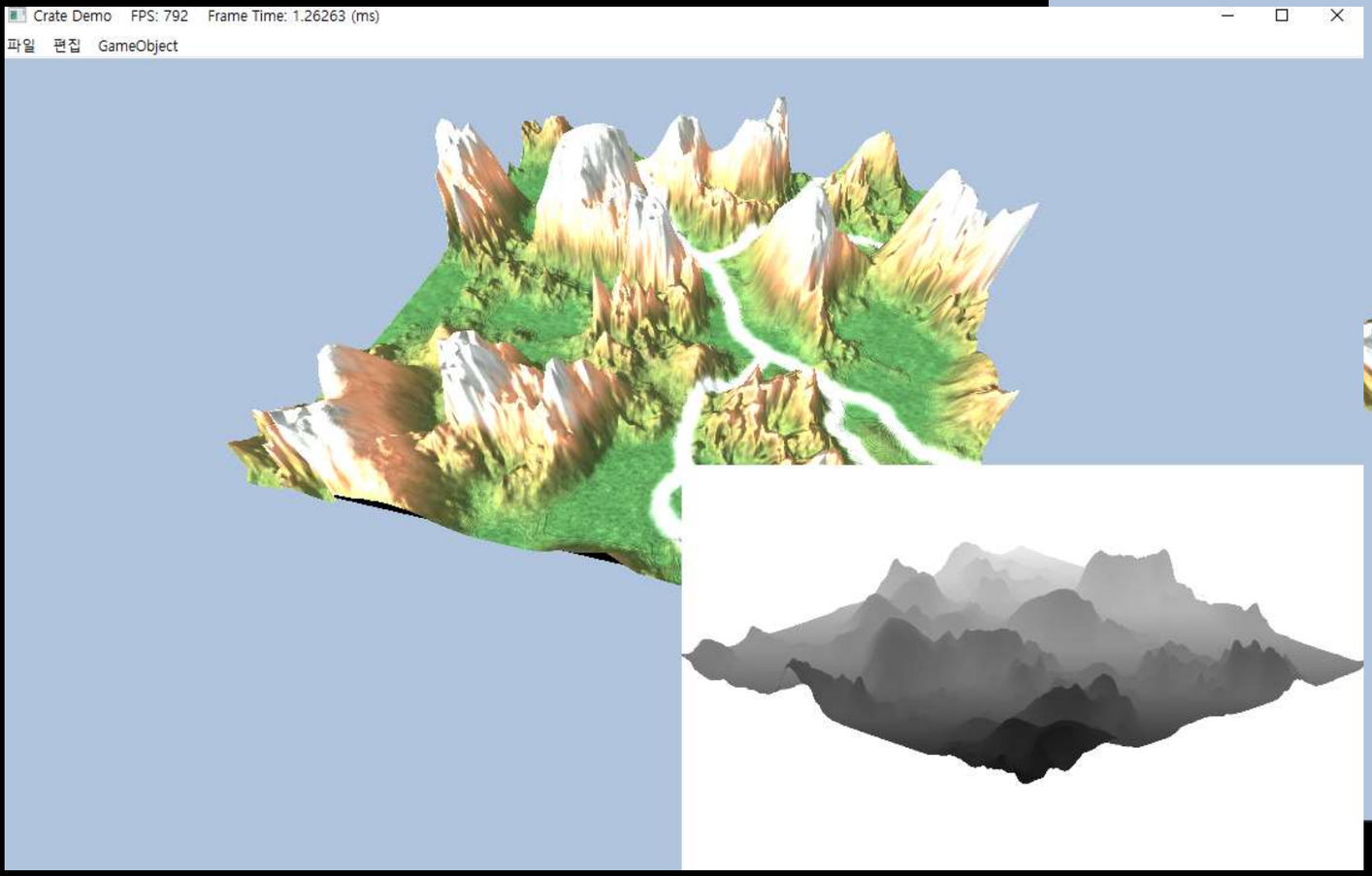
- 그림자맵

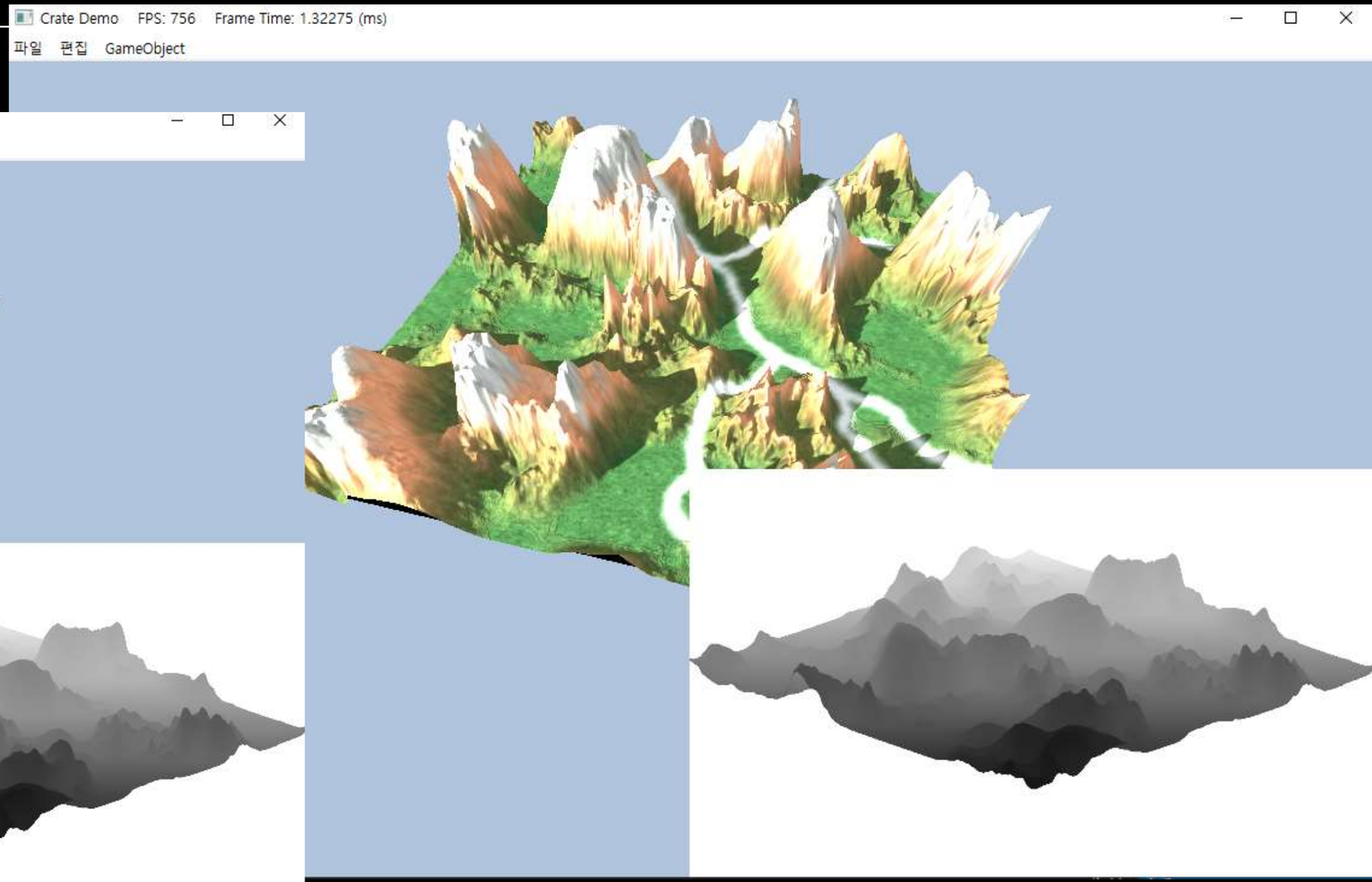




- 그림자맵을 이용해 그림자를 구현한 모습입니다.
- 각 사진의 우측 하단에 매 프레임마다 그림자맵 텍스쳐를 보여주고 있습니다.
- 오른쪽 사진은 기하 쉐이더를 이용해 나무의 그림자를 구현하였습니다.

- 테셀레이션 지형 그림자





- 테셀레이션 지형의 그림자맵을 만들어 실시간 그림자를 구현하였습니다.
- 왼쪽 사진은 그림자를 적용하지 않은 모습, 오른쪽 사진이 그림자가 적용 된 모습입니다.

- 그림자맵 코드

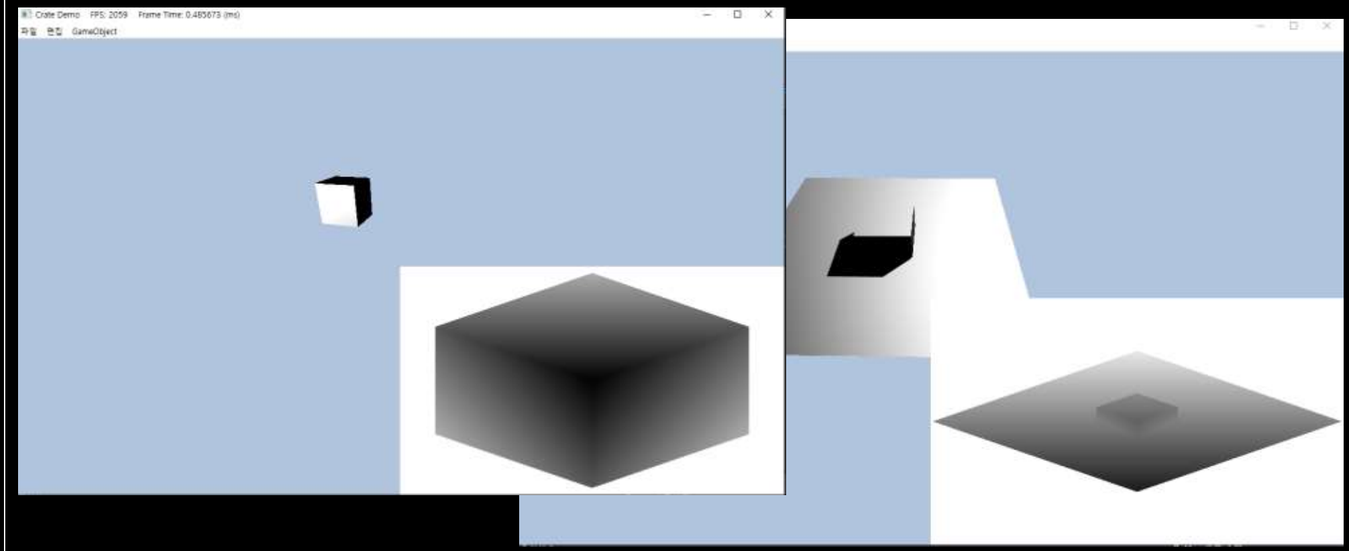
```
void ShadowMap::ComputeBoundingSphere(std::vector<Renderer*> renderers)
{
    XMFLOAT3 maxPosF(-MathHelper::Infinity, -MathHelper::Infinity, -MathHelper::Infinity);
    XMFLOAT3 minPosF(MathHelper::Infinity, MathHelper::Infinity, MathHelper::Infinity);

    XMVECTOR maxPosV = XMLoadFloat3(&maxPosF);
    XMVECTOR minPosV = XMLoadFloat3(&minPosF);


    for (auto renderer : renderers)
    {
        Mesh* mesh = renderer->GetMesh();
        XMMATRIX world = XMLoadFloat4x4(renderer->GetTransform()->GetWorld());

        //모든 오브젝트의 aabb를 검사해 최소정점과 최대정점을 구함.
        XNA::AxisAlignedBox& aabb = mesh->GetAABB();
        int xFactor, yFactor, zFactor;
        float xPos, yPos, zPos;
        for (int i = 0; i < 8; ++i)
        {
            xFactor = (i & 1) ? 1 : -1;
            yFactor = (i & 2) ? 1 : -1;
            zFactor = (i & 4) ? 1 : -1;

            xPos = aabb.Center.x + xFactor * aabb.Extents.x;
            yPos = aabb.Center.y + yFactor * aabb.Extents.y;
            zPos = aabb.Center.z + zFactor * aabb.Extents.z;

            XMVECTOR P = XMVector3TransformCoord(XMLoadFloat3(&XMFLOAT3(xPos,yPos,zPos)), world);
            maxPosV = XMVectorMax(maxPosV, P);
            minPosV = XMVectorMin(minPosV, P);
        }
```

- 오브젝트가 생성될 때마다 모든 오브젝트를 포함하는 경계구를 생성하였습니다.

- 모든 오브젝트의 AABB 정점을 검사해 최소정점과 최대정점을 구하고, 두 정점의 중점과 거리를 지름으로 사용해 경계구를 생성하였습니다.

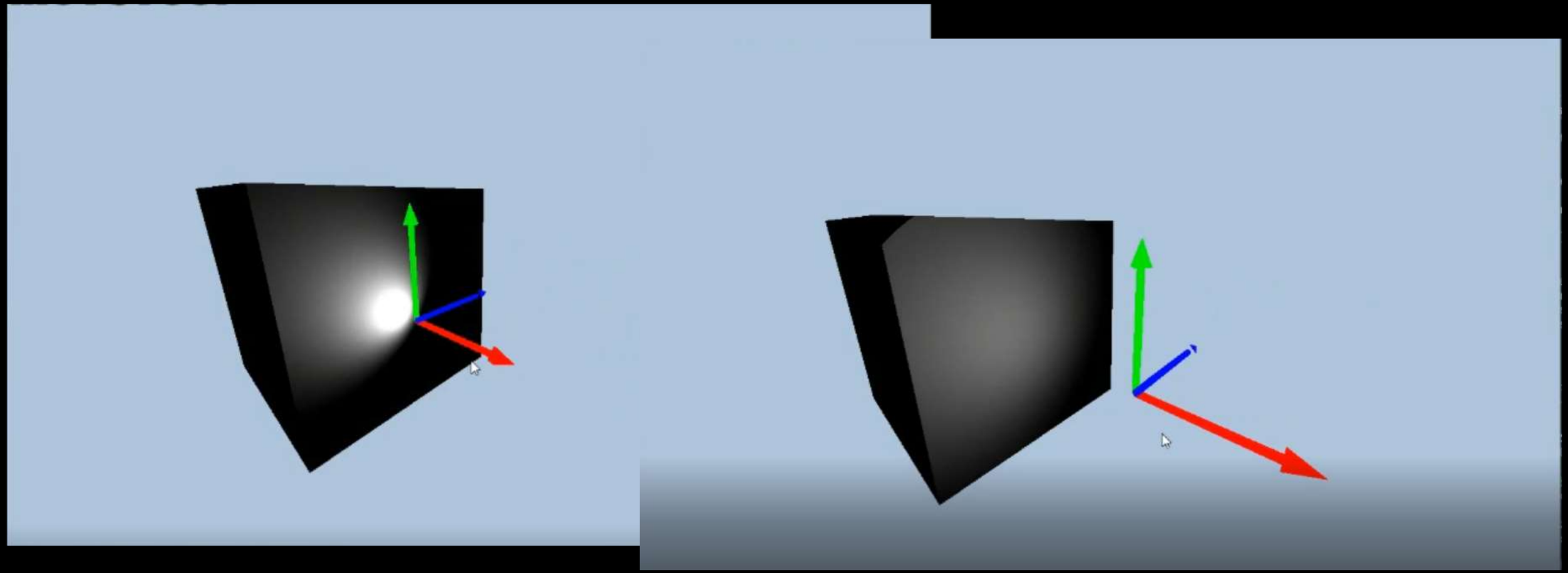
```
    //바라보는 점
    XMVECTOR targetPos = XMLoadFloat3(&m_boundingSphere.center);
    //평행광원의 위치(경계구의 중점에서 평행광방향의 반대방향*2 만큼 이동한 점)
    XMVECTOR lightPos = targetPos - 2.0f*m_boundingSphere.radius*lightDir;
    if(XMVector3Equal(lightDir, XMVectorZero()))
        return false;
```

- 평행광은 방향만을 가지고 있기때문에 (경계구 중심 + (경계구지름 * -평행광방향)을 위치로 사용하고, 평행광의 위치와 경계구 중심을 바라보는 점으로 이용해 광원 시야행렬을 만들었습니다.



- 새로운 오브젝트가 추가되면 모든 오브젝트를 포함하는 그림자맵이 그려지는 모습입니다.

- MoveTool



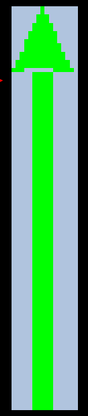
- 오브젝트를 선택했을 때 세계공간 기준으로 XYZ축 방향의 막대를 렌더링 했습니다.
- 마우스로 막대를 드래그하면 각 축의 방향으로 오브젝트를 이동 시킬 수 있습니다.
- 위 사진은 빈 오브젝트의 Light 컴포넌트를 추가 해 Point Light를 이동시킨 모습입니다.

MoveTool 렌더링

파란색은 클래스명 / 검정색은 멤버변수 / 초록색은 멤버함수 입니다.

MoveToolRenderer
ID3D11Buffer* mVB;
ID3D11Buffer* mInstanceBuffer;

```
struct InstancingWorldColor
{
    XMFLOAT4X4 world;
    XMFLOAT4 color; //기본색상
};
```



- mVB 버퍼에 y축막대 모양 메쉬의 정점을 저장합니다. (불변)

- mInstanceBuffer 버퍼에 세계행렬, 색상 정보를 저장합니다.
  총 3개의 축을 렌더링 하므로 3개의 행렬과 색상이 필요합니다.
  행렬은 매 프레임 업데이트 됩니다.

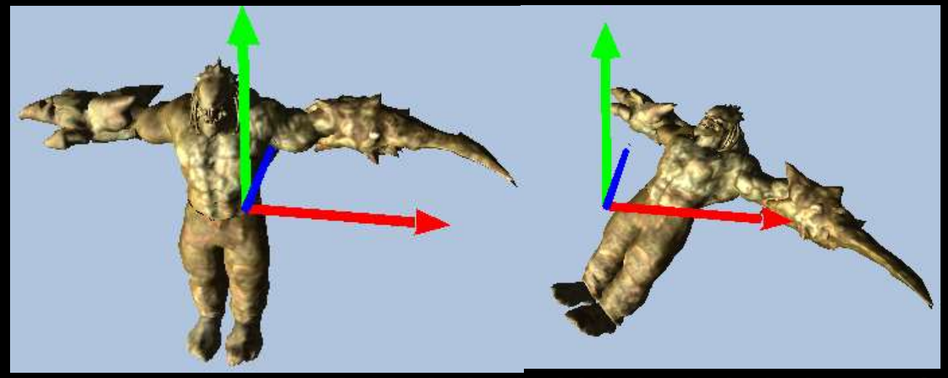
- 정점버퍼는 0번슬롯에 mVB, 1번슬롯에 mInstanceBuffer를
  사용합니다.

인스턴싱 버퍼(매 프레임 업데이트)		
Y축 데이터 World = Scale * 단위행렬 * Translate Color = (0.0f, 1.0f, 0.0f, 1.0f) = Green	X축 데이터 World = Scale * z축에대해 -90도 회전 * Translate Color = (1.0f, 0.0f, 0.0f, 1.0f) = Red	Z축 데이터 World = Scale * x축에대해 90도 회전 * Translate Color = (0.0f, 0.0f, 1.0f, 1.0f) = Blue

- 회전 부분은 항상 고정입니다.
- 매 프레임마다 각 축의 세계행렬을 업데이트하고
  인스턴싱 렌더링으로 3개의 축을 한번에 렌더링합니다.

MoveTool 렌더링 업데이트 코드

```
const XMFLOAT3& center = mAABBCenter;
XMMATRIX world;
m_gameObj->nodeHierarchy->GetFinalTransform(world, m_gameObj->GetID());
float x = world._41 + center.x;
float y = world._42 + center.y;
float z = world._43 + center.z;
XMMATRIX translate = XMMatrixTranslationFromVector({ x,y,z });
```



오브젝트의 회전과 관련없이 렌더링되는 모습

- 인스턴싱 버퍼의 world 행렬을 매 프레임마다 업데이트 합니다.
  (총 3개)

- MoveTool은 항상 세계공간의 xyz축을 표시하기 때문에
  오브젝트의 회전,크기와 관련이 없습니다. 따라서 오브젝트의
  세계행렬에서 이동변환 부분인 4행 1~3열을 추출합니다.
  (Translate 행렬 완료)

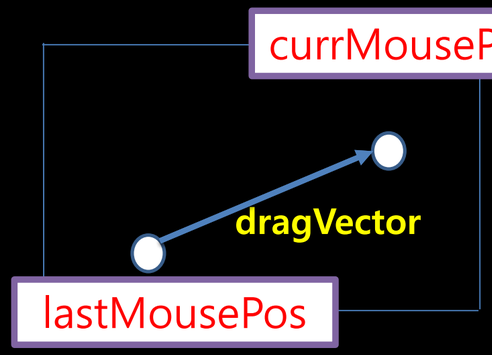
- y축을 z축에대해 -90도 회전하면 x축
  y축을 x축에대해 90도 회전하면 z축이므로
  함수내에 static 변수로 선언한 행렬을 사용합니다.
  (Rotation 행렬 완료)

```
//기본 메쉬인 y축에 곱할 단위행렬
static XMMATRIX identity = XMMatrixIdentity();
//y축을 z축에대해 -90도 회전하면 x축
static XMMATRIX makeXaxis =
    XMMatrixRotationAxis(XMVectorSet(0.0f, 0.0f, 1.0f, 0.0f), XMConvertToRadians(-90.0f));
//y축을 x축에대해 90도 회전하면 z축
static XMMATRIX makeZaxis =
    XMMatrixRotationAxis(XMVectorSet(1.0f, 0.0f, 0.0f, 0.0f), XMConvertToRadians(90.0f));
```

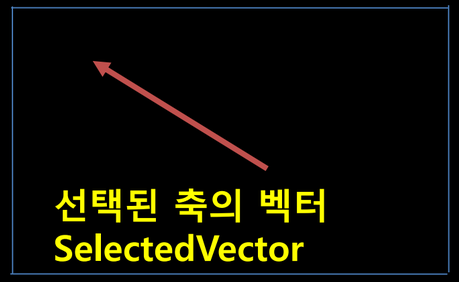
- MoveTool을 스크린에서 항상 같은 크기로 렌더링 하기 위한
  Scale 값을 구합니다. (Scale 행렬 완료)(뒤에 내용이 있습니다.)

- 이렇게 구한 행렬을 곱해 세계행렬을 완성합니다.
  WorldMatrix = Scale * Rotation * Translate

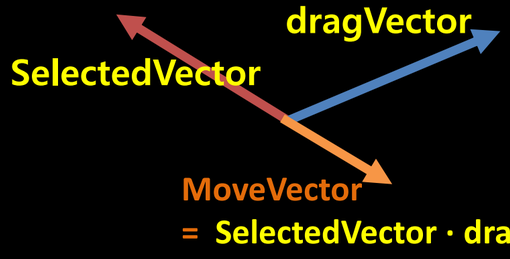
MoveTool 드래그 이동



- 저번 프레임의 마우스 위치가 lastMousePos,
  이번 프레임의 마우스 위치가 currMousePos때
  드래그 한 벡터 dragVector =
  lastMousePos - currMousePos 가 됩니다.

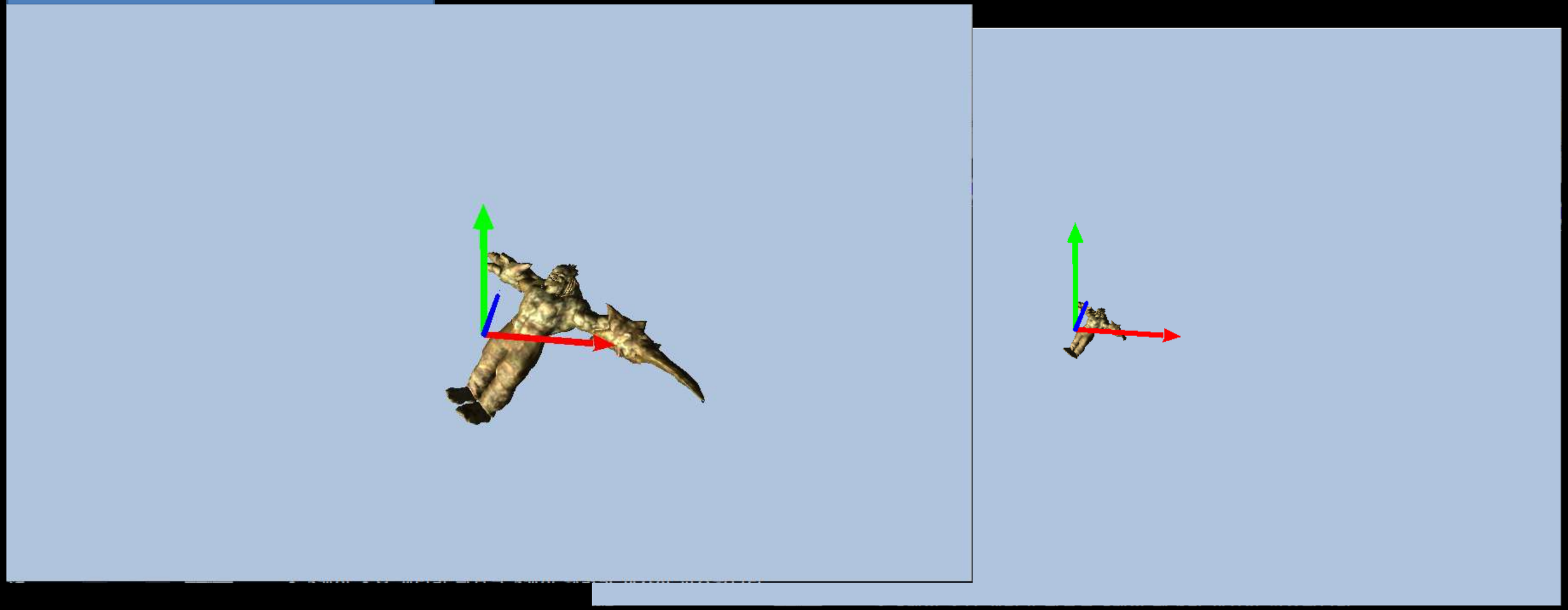


- 마우스를 클릭한 방향으로 반직선을 쏴 교차판정을 통해
  XYZ축 중 선택된 축 SelectedVector를 찾습니다.



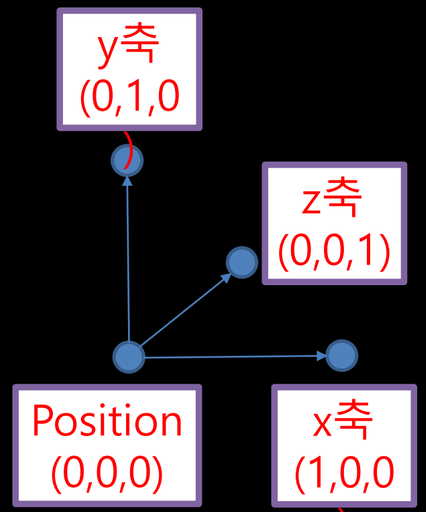
- dragVector에서 SelectedVector 방향으로의 벡터 MoveVector를 추출합니다.
- MoveVector = SelectedVector · dragVector로 내적을 사용해 구했습니다.
- 마지막으로 선택된 축의 방향으로 MoveVector의 크기만큼 이동시켜
  오브젝트를 이동시킵니다.

- MoveTool2

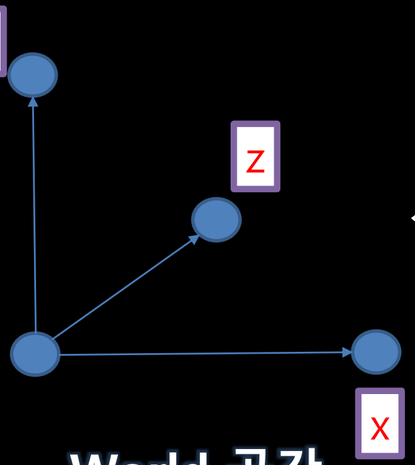


- MoveTool이 카메라의 거리와 상관없이 항상 같은 크기를 유지하도록 한 모습입니다.
- XYZ축 벡터를 미리 NDC 공간으로 변환시킨 뒤, 미리 정의해 놓은 원하는 크기 AXISSIZE로 조정합니다.
- AXISSIZE는 NDC공간을 기준으로 정의 됩니다.
  [-1,1]의 범위를 가진 NDC 공간에서 AXISSIZE를 0.5로 설정하면 화면의 4분의1정도 길이를 의미합니다.

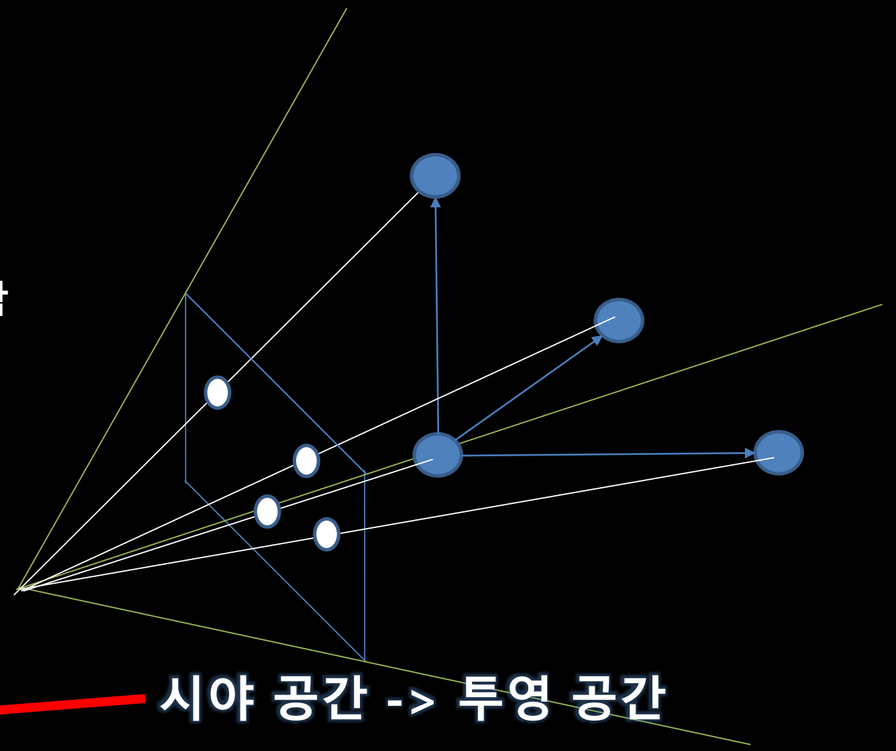
MoveTool 크기조정



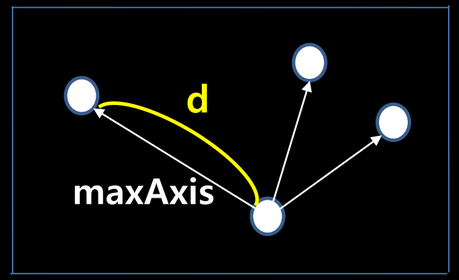
세계행렬 곱함
(Scale은 1고정)



시야,투영행렬 곱함



동차나누기



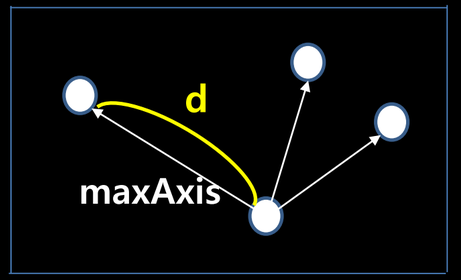
- NDC 공간으로 변환된 4개의 정점을 이용해서
  XYZ축을 나타내는 3개의 벡터를 구합니다.

- 3개의 벡터 중 크기가 가장 큰 벡터 maxAxis를 찾습니다.

- 이 maxAxis의 크기를 d라고 할때, d를 항상 일정한 값으로 유지시켜
  카메라 위치와 거리에 관계없이 화면에서 항상 같은크기로 보이게 합니다.

MoveTool 크기조정 코드

```
void MoveToolRenderer::GetMoveToolAxes(XMVECTOR & posW, XMMATRIX & viewProj)
{



    //세계공간에서의 각 축의 끝점
    //현재 오브젝트의 위치인 posW에서 각 축 방향으로 이동
    XMVECTOR xAxisEndPosW = XMVector3TransformCoord(posW, XMMatrixTranslation(1.0f, 0.0f, 0.0f));
    XMVECTOR yAxisEndPosW = XMVector3TransformCoord(posW, XMMatrixTranslation(0.0f, 1.0f, 0.0f));
    XMVECTOR zAxisEndPosW = XMVector3TransformCoord(posW, XMMatrixTranslation(0.0f, 0.0f, 1.0f));

    //동차절단공간에서의 각 축의 끝 점을 구함
    XMVECTOR xAxisEndPosH = XMVector3TransformCoord(xAxisEndPosW, viewProj);
    XMVECTOR yAxisEndPosH = XMVector3TransformCoord(yAxisEndPosW, viewProj);
    XMVECTOR zAxisEndPosH = XMVector3TransformCoord(zAxisEndPosW, viewProj);

    //ndc 공간으로 변환
    XMVECTOR posH = XMVector3TransformCoord(posW, viewProj);
    XMVECTOR desetNDCpos = posH / XMVectorGetW(posH);
    XMVECTOR xAxisEndPosNDC = xAxisEndPosH / XMVectorGetW(xAxisEndPosH);
    XMVECTOR yAxisEndPosNDC = yAxisEndPosH / XMVectorGetW(yAxisEndPosH);
    XMVECTOR zAxisEndPosNDC = zAxisEndPosH / XMVectorGetW(zAxisEndPosH);

    //ndc공간에서 xyz 축의 벡터
    XMVECTOR AxisDir[3];
    AxisDir[0] = xAxisEndPosNDC - desetNDCpos;
    AxisDir[1] = yAxisEndPosNDC - desetNDCpos;
    AxisDir[2] = zAxisEndPosNDC - desetNDCpos;

    //ndc공간에서 각 축의 크기
    float AxisLen[3];
    AxisLen[0] = XMVectorGetX(XMVector2Length(AxisDir[0]));
    AxisLen[1] = XMVectorGetX(XMVector2Length(AxisDir[1]));
    AxisLen[2] = XMVectorGetX(XMVector2Length(AxisDir[2]));

    UINT maxIdx = AxisLen[0] > AxisLen[1] ? 0 : 1;
    maxIdx = AxisLen[maxIdx] > AxisLen[2] ? maxIdx : 2;

    scaleFactor = AXISSIZE / AxisLen[maxIdx];
}
```



NDC 공간

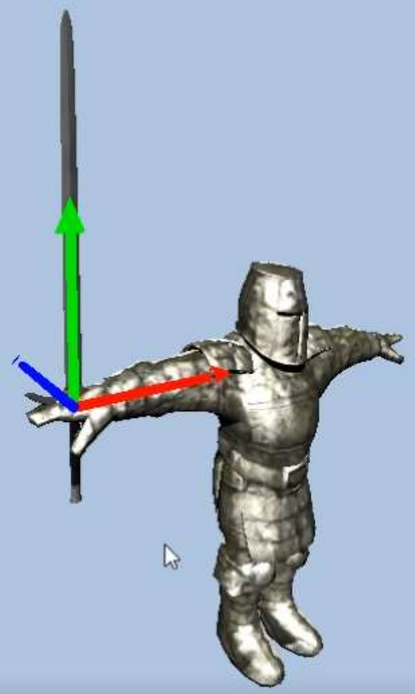
/* NDC 공간에서 일정하게
유지하기 원하는 크기 */
#define AXISSIZE 0.5

- NDC 공간에서 가장 큰 축 벡터의 크기가 d이고,
d를 항상 AXISSIZE 값으로 유지함으로써
카메라 거리에 관계없이 스크린에서 같은 크기로
보이게 합니다.

- [d * scaleFactor = AXISSIZE]가 되는
scaleFactor값을 찾습니다.
[scaleFactor = AXISSIZE / d] 가 됩니다.

- 위에서 구한 scaleFactor를 세계행렬의 크기변환 값으로
사용하면 NDC 공간으로 변환하면서
가장 긴 축의 길이가 항상 AXISSIZE가 됩니다.

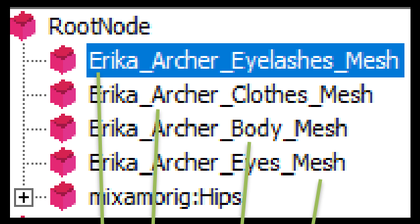
- 계층구조



- 계층구조를 따라 자식 오브젝트는 부모 오브젝트를 거슬러 올라가면서
  부모공간으로 변환을 반복하고 루트공간으로 변환됩니다.
- RightHand 오브젝트의 자식으로 LongSword를 설정해 검을 착용한 효과를 나타냈습니다.
- 애니메이션에 의해 부모 오브젝트의 위치가 변함에 따라 LongSword의 위치도 변화하는 모습입니다.

계층구조 코드1

파란색은 클래스명 / 검정색은 멤버변수 / 초록색은 멤버함수 입니다.



하나의 객체 공유

NodeHierarchy

NodeHierarchy

```cpp
class NodeHierarchy
{
public:
    UINT mTechType;
public:
    NodeHierarchy() : m_animator(std::make_shared<Animator>()),
        mTechType(TechniqueType::Light | TechniqueType::DiffuseMap)
        {}

    std::shared_ptr<Animator> m_animator;

    //node id와 index 매핑
    std::map<std::string, int> nodeIdIdx;
    std::vector<int> parentIndices;
    std::vector<std::weak_ptr<Transform>> toParents; //각 노드에서 부모로의 변환
```

- 오브젝트 추가 시 index를 배정하고 계층구조를 만듭니다.

- 오브젝트의 Transform을 weak_ptr로 참조하여 해당 오브젝트 위치 변경시 자동으로 계층구조의 위치 계산을 수행합니다.

GameObject

```cpp
class GameObject : public Object
{

public:
    GameObject() : Object(), transform(std::make_sh
    GameObject(const gameObjectID& id) : Object(id)
    GameObject(const gameObjectID& id, const std::w
    //복사 생성자
    GameObject(const GameObject& other);
    ~GameObject();

    std::shared_ptr<Transform> transform;
    std::shared_ptr<NodeHierarchy> nodeHierarchy;
```

- 오브젝트는 본인이 어떤 계층구조에 속하는지 확인하기 위해 NodeHierarchy 클래스를 가집니다.

- 부모 오브젝트의 자식 오브젝트를 추가할 때 같은 계층구조를 공유하기 위해 Shared_Ptr을 이용합니다.

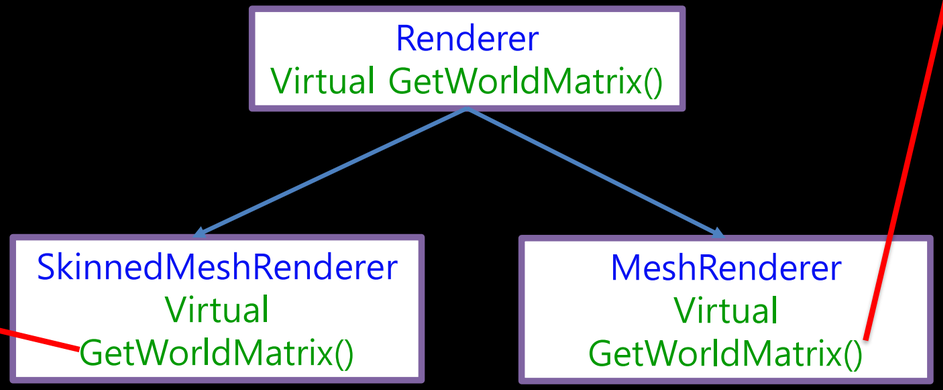
계층구조 코드2

파란색은 클래스명 / 검정색은 멤버변수 / 초록색은 멤버함수 입니다.

```
void NodeHierarchy::GetRootWorldTransform(XMMATRIX & dest)
{
    std::shared_ptr<Transform> currTransform = toParents[0].lock();
    if (!currTransform)
        return;
    //루트노드의 부모변환 행렬(세계행렬과 같음)
    dest = XMLoadFloat4x4(currTransform->GetWorld());
}
```

- 항상 Root 노드의 변환행렬을 반환하는 함수입니다.

- SkinnedMeshRenderer에서 정점 위치는 애니메이션의 영향을 받으므로 RootNode의 위치변경으로만 오브젝트 위치를 설정하도록 합니다.



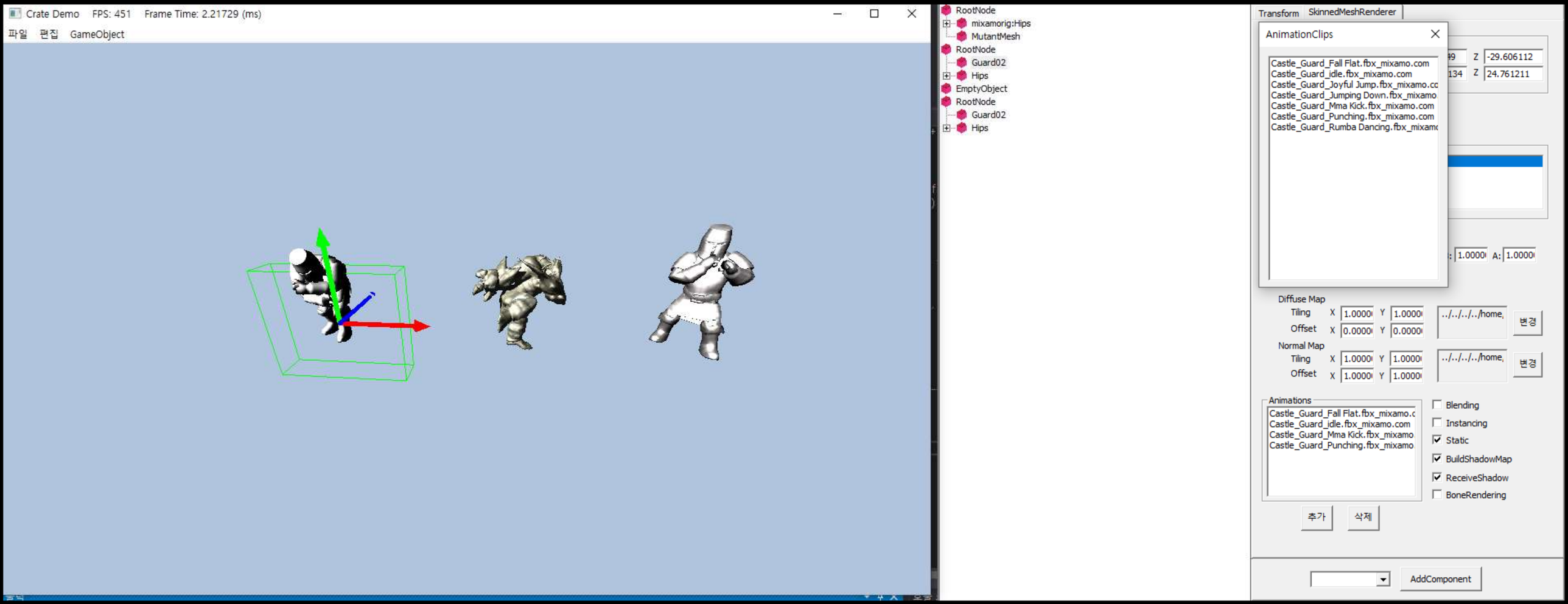

```
void NodeHierarchy::GetFinalTransform(XMMATRIX & dest, std::string & nodeId)
{
    int nodeIdx = nodeIdIdx[nodeId];
    int parentIdx = parentIndices[nodeIdx];
    std::shared_ptr<Transform> currTransform = toParents[nodeIdx].lock();
    if (!currTransform)
        return;

    //현재 노드의 부모변환 행렬
    dest = XMLoadFloat4x4(currTransform->GetWorld());

    //부모 인덱스가 -1은 루트노드, -2는 삭제된 노드
    while (parentIdx >= 0)
    {
        currTransform = toParents[parentIdx].lock();
        if (currTransform)
            dest = dest * XMLoadFloat4x4(currTransform->GetWorld());

        parentIdx = parentIndices[parentIdx];
    }
}
```

- 해당 노드의 최종변환행렬을 구하는 함수입니다.

- 재귀적으로 부모 공간으로의 변환행렬을 곱해 Root 공간으로의 변환행렬을 구합니다.

- MeshRenderer에서 해당 오브젝트의 위치와 부모 오브젝트의 위치에따라 결정됩니다.

- 애니메이션



- Assimp 라이브러리를 이용해 Bone 구조와 애니메이션을 로딩하고 스키닝 애니메이션을 적용하였습니다.
- 매 프레임마다 애니메이션 업데이트 후 계산쉐이더를 이용해 정점 위치를 계산해 렌더링 하였습니다.
- 우측 Inspector 창에서 애니메이션을 불러오고 변경할 수 있습니다.

애니메이션 적재/변환

파란색은 클래스명 / 검정색은 멤버변수 / 초록색은 멤버함수 입니다.

FBX 파일

1. 애니메이션 적재
Assimp 라이브러리를 사용해 FBX 파일의 애니메이션을 적재합니다.

AssimpLoader
AssimpAnimation

2. 애니메이션 변환
적재한 애니메이션을 변환하여 매니저 클래스에 저장합니다.

AnimationManager
(SingleTon)
MyAnimationClip

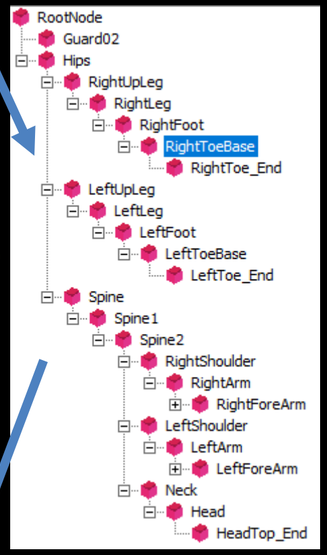
3. 애니메이션 사용
렌더러 컴포넌트의 애니메이터에 클립을 복사합니다.
각 렌더러에서 Update마다 Animator의 현재 클립이 업데이트 됩니다.

SkinnedMeshRenderer
Animator

```
NodeStruct* currNode;
//모든 노드를 bfs로 탐색하면서 BoneName과 일치하는 노드가 있으면 뼈 정보 추가
while (!q.empty())
{
    parentIdx = q.front().first;
    currNode = q.front().second;
    q.pop();

    it = assimpBones.find(currNode->GetName());
    //현재 노드가 Bone Name과 일치할 때
    if (it != assimpBones.end())
    {
        parents.push_back(parentIdx);
        offsets.push_back(it->second.offsetMat);
        boneParentMatrix.push_back(it->second.toParentMat);

        m_boneNameIdx[it->first] = parents.size() - 1;
        parentIdx = parents.size() - 1;
    }

    for (auto& child : currNode->childs)
    {
        q.push({ parentIdx, &child });
    }
```

- 해당 모델의 모든 노드를 BFS로 탐색하면서 뼈의 계층구조를 만듭니다.
- 해당 노드와 뼈의 이름이 같을 때 부모의 인덱스를 가지는 vector 컨테이너를 이용해 계층구조를 표현합니다.

계층구조 적용 전 Bone
{RightFoot, Hips, RightUpLeg, RightToeBase...}



계층구조 적용

계층구조 적용 Bone
{Hips, RightUpLeg, RightFoot, RightToeBase...}

부모 인덱스 Parents
{-1, 0, 1, 2...}

- Parents에서 1번째 인덱스의 값이 0이면, RightUpLeg의 부모가 Hips인 것을 나타냅니다.
- -1은 해당 노드가 Root Bone임을 나타냅니다.

Animator 구조

파란색은 클래스명 / 검정색은 멤버변수 / 초록색은 멤버함수 입니다.


```cpp
class Animator
{
private:
    bool AnimatedPerFrame;
public:
    Animator() : timePos(0.0f), AnimatedPerFrame(0)  {}
    Animator(const Animator& other);
    Animator& operator=(const Animator& other);
    void Update(float deltaTime);

private:
    //애니메이터에서 참조하는 뼈 구조
    BoneDatas boneDatas;
    //현재 애니메이터에서 실행할 수 있는 클립들
    std::map<std::string, MyAnimationClip> clips;
    //현재 실행하는 클립이름
    std::string currClipName;
    //현재 실행중인 시간위치
    float timePos;
```

- Animator 클래스는 애니메이션 클립들과 Bone 구조를 가집니다.
- 매 프레임마다 deltaTime에 따라 애니메이션을 업데이트합니다.

```cpp
class MyAnimationClip
{
public:
    double duration;
    std::string m_clipName; //animation clip의 이름
    std::vector<BoneFrames> m_bones;

    void Interpolate(float time, std::vector<XMFLOAT4X4>& toParents);
};
```

- 하나의 애니메이션 클립을 나타내는 클래스 입니다.
- 각 뼈의 움직임을 나타내는 BoneFrames 클래스를 뼈의 개수만큼 벡터에 저장합니다.

```cpp
//시간, 해당 키 프레임의 시간, 벡터
typedef std::pair<float, XMFLOAT3> frameKey3;
typedef std::pair<float, XMFLOAT4> frameKey4;

//뼈 하나에 대해 저장된 KeyFrame들
//애니메이션으로 보간된 행렬은 부모변환 행렬로 사용됨.
class BoneFrames
{
public:
    std::wstring m_boneName;
    BoneFrames(const std::wstring& name) : m_boneName(name) {}
private:
    std::vector<frameKey3> scaleKeys;
    std::vector<frameKey4> quaternionKeys;
    std::vector<frameKey3> translateKeys;
public:
    void Interpolate(float time, XMFLOAT4X4& dest);
```

- 하나의 뼈의 시간에 따른 움직임을 나타내는 크기, 회전, 이동 정보를 가지고 업데이트시 보간합니다.

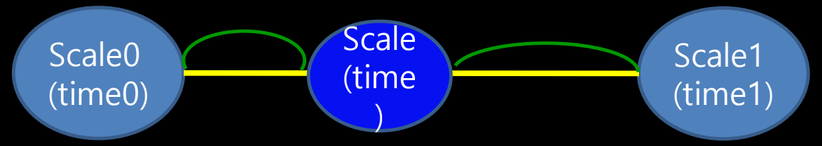
애니메이션 업데이트 코드

파란색은 클래스명 / 검정색은 멤버변수 / 초록색은 멤버함수 입니다.

```
void Animator::Update(float deltaTime)
{
    //여러 렌더러에서 공유하는 Animator일 경우
    //중복 업데이트 방지
    if (AnimatedPerFrame)
        return;

    UpdateBody

    //이번 프레임에 업데이트 완료
    AnimatedPerFrame = true;

}
```

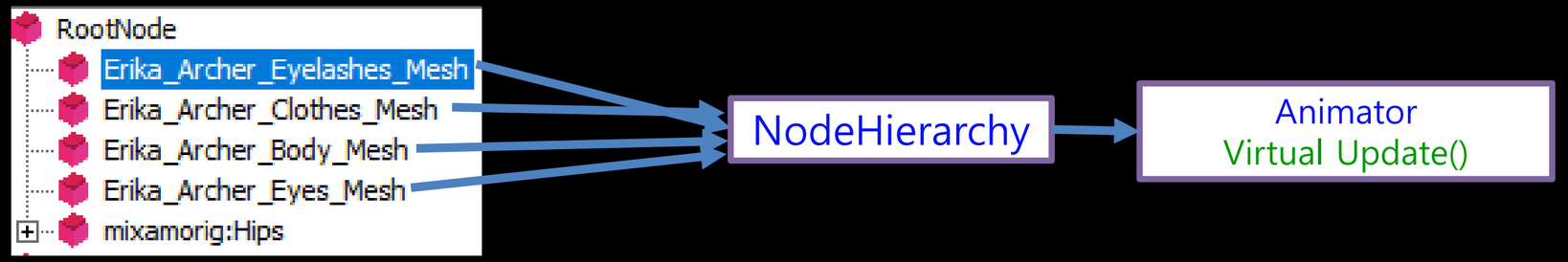
- 같은 애니메이터를 공유하는 렌더러에서 중복 업데이트를 방지하기 위해 Bool AnimatedPerFrame 변수를 플래그로 사용합니다.

- 이번 프레임에서 업데이트를 완료한 애니메이터는 플래그를 true로 설정합니다.
- 다른 애니메이터에서 업데이트 시 플래그가 true면 업데이트 하지 않습니다.




```
//time보다 크거나 같은 키프레임을 찾아 보간한다.
auto it = std::lower_bound(scaleKeys.begin(), scaleKeys.end(), time, compKey3);

lerpPercent = (time - (it-1)->first) / (it->first - (it-1)->first);
XMVECTOR scale0 = XMLoadFloat3(&(it-1)->second);
XMVECTOR scale1 = XMLoadFloat3(&it->second);
scale = XMVectorLerp(scale0, scale1, lerpPercent);
```

- 현재 time의 양쪽에 해당하는 원소를 이분탐색으로 구합니다.
- 현재 time에 해당하는 Scale 값을 보간해서 구합니다.
- 회전과 이동에 동일하게 수행한 뒤 Scale * Quaternion * Translation 계산을 수행해 이번 프레임에서 해당 뼈의 행렬을 구합니다.



- 하나의 오브젝트가 여러개의 메쉬로 이루어진 경우 하나의 애니메이터를 공유하고, 중복 업데이트를 방지합니다.

애니메이션 쉐이더 변수

초록색은 변수를 나타냅니다.

Float4x4 **gBoneTransforms[BoneSize] 뼈 최종변환 행렬**

RootBone	Bone1	Bone2	Bone3	Bone4	Bone5	Bone6	Bone7	Bone8	...

StructuredBuffer<vertex> **gVertices 모델의 정점 정보**

Vertex0	Vertex1	Vertex2	Vertex3	...

StructuredBuffer<skinData> **gInputSkinData 모델 스킨 정보**

SkinData0	SkinData1	SkinData2	SkinData3	...

- i 번째 vertex와 i 번째 skinData는 매핑됩니다.
- skinData는 정점이 참조하는 뼈의 인덱스와 가중치를 가집니다.

StructuredBuffer<resultVertex> **gVertices 최종 정점 정보**

Vertex0	Vertex1	Vertex2	Vertex3	...

groupshared float4x4 **gBoneCache[BONESIZE] 공유메모리**

```
//모든 뼈 최종행렬을 공유메모리에 저장
if (groupThreadID.x < BONESIZE)
    gBoneCache[groupThreadID.x] = gBoneTransforms[groupThreadID.x];

//동기화
GroupMemoryBarrierWithGroupSync();
```

- 속도 향상을 위해 **뼈 최종변환 행렬**을 공유메모리에 저장

계산쉐이더 정점위치 계산

```
[numthreads(N, 1, 1)]
void Skinning(int3 dispatchThreadID : SV_DispatchThreadID,
    int3 groupThreadID : SV_GroupThreadID)
```

- 정점의 개수 N길이의 1차원 계산쉐이더를 수행합니다.
- dispatchThreadID의 값을 정점버퍼의 인덱스로 사용해 각 스레드마다 하나의 정점변환을 수행합니다.

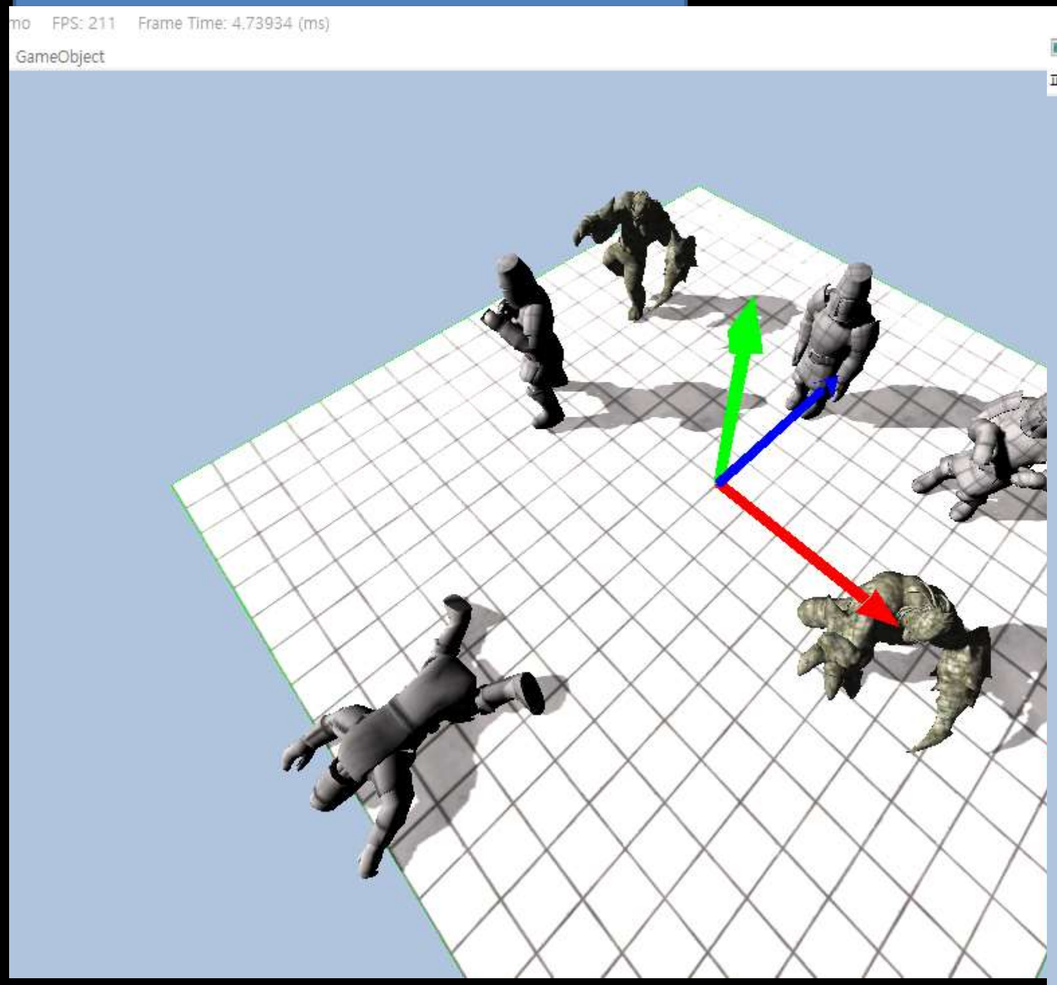
Int **boneIdx[4]** = gInputSkinData[dispatchThreadID].indices

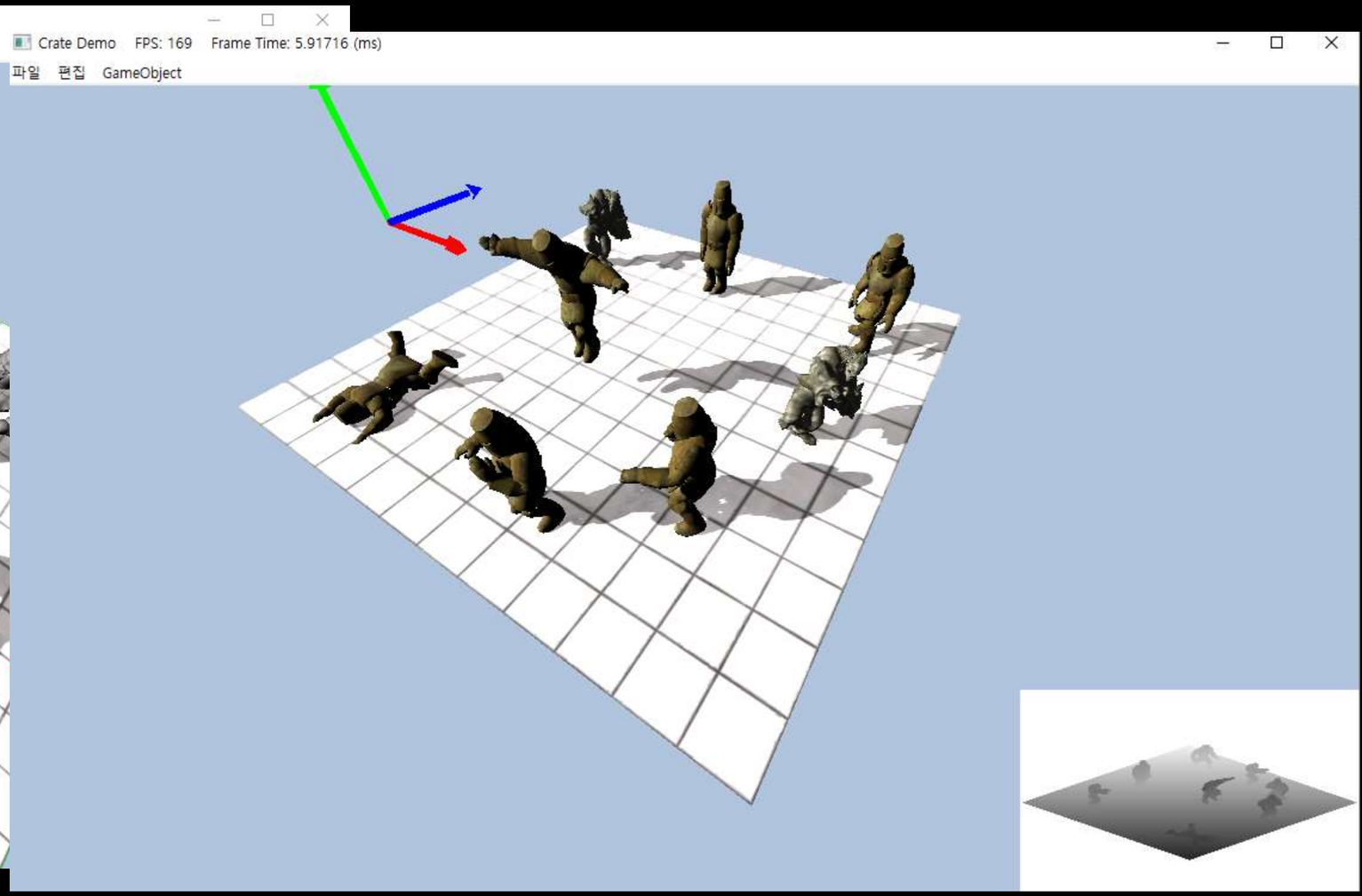
float **weights[4]** = gInputSkinData[dispatchThreadID].weight

```
posL += weights[i] * mul(float4(gVertices[dispatchThreadID.x].PosL,
    gBoneCache[boneIdx[i]]).xyz;
```

- 최대 4개의 뼈를 참조해 가중치에 따라 위치를 계산합니다.

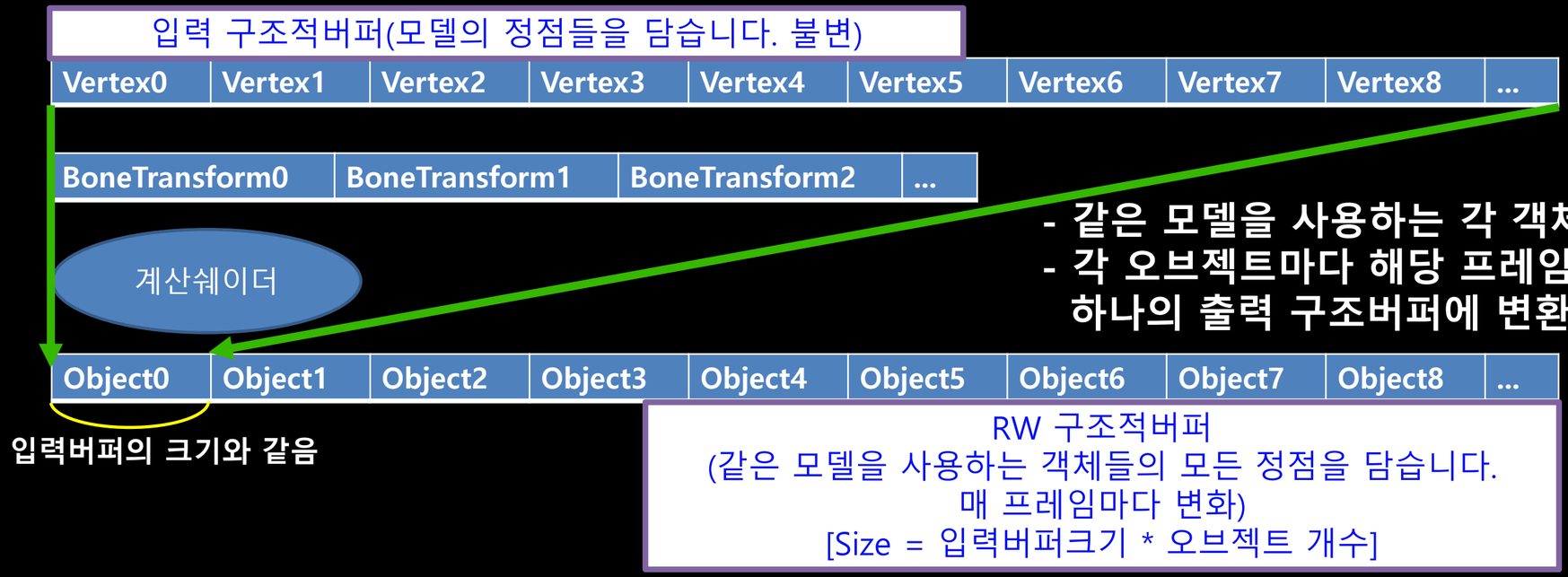
- 애니메이션 그림자





- 계산쉐이더에서 애니메이션의 최종변환 행렬을 이용해 정점들의 위치를 버퍼에 저장합니다.
- 버퍼에 저장된 정점 위치를 이용해 추가계산 없이 그림자맵에 오브젝트를 그릴 수 있습니다.
- 위 사진은 각 오브젝트마다 다른 애니메이션을 적용하고 그림자를 그린 모습입니다.

애니메이션 인스턴싱 / 그림자



- 같은 모델을 사용하는 각 객체들에 대해 계산쉐이더를 수행합니다.
- 각 오브젝트마다 해당 프레임에서 업데이트 된 뼈 변환 행렬을 사용하고 하나의 출력 구조버퍼에 변환된 정점 데이터를 저장합니다.

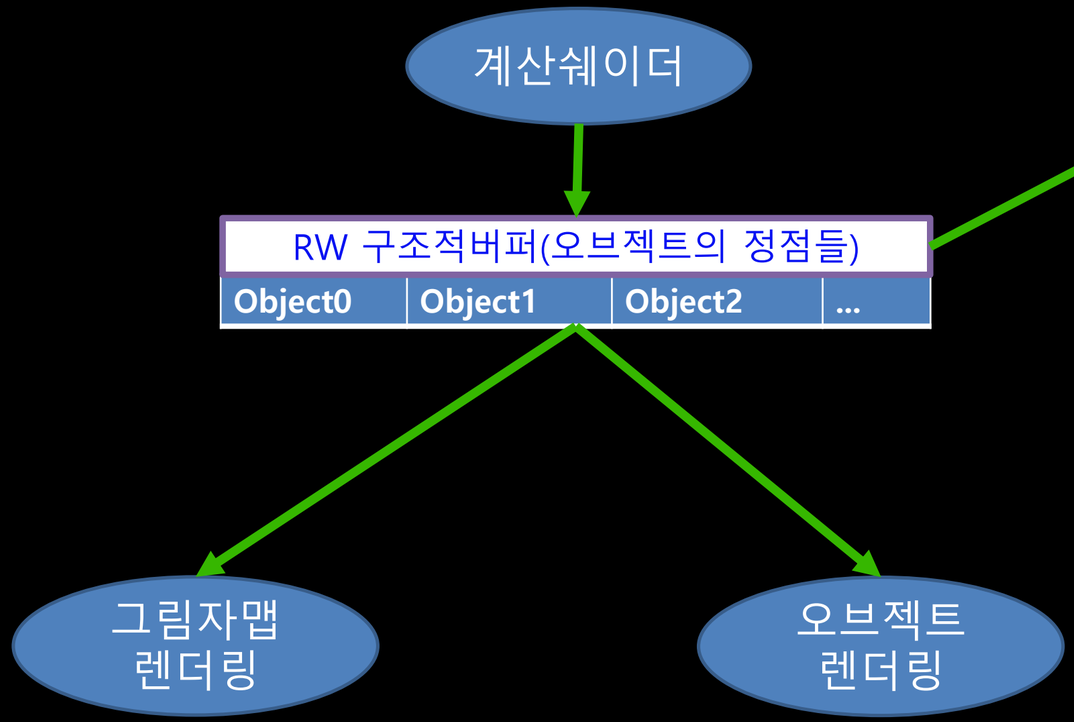
```
//해당 메쉬의 정점 개수를 구함
static uint vertexSrcSize = 0;
static uint bufferStride = 0;
gVertices.GetDimensions(vertexSrcSize, bufferStride);

uint vertexStart = vertexSrcSize * instanceID;
```

- 출력 버퍼의 시작지점을 구하는 코드입니다.
- 해당 모델의 정점 개수 * 이번 객체의 ID
- 모델정점개수 = 100, 5번째 객체의 시작 인덱스(vertexStart)는 500이 됩니다.

```
gDestVertices[vertexStart + dispatchThreadID.x].PosL = posL;
```

- i번째 스레드는 모델의 i번째 정점을 계산하므로 해당 정점의 출력버퍼 위치는 vertexStart + dispatchThreadID가 됩니다.
- 같은 모델을 사용하는 모든 객체의 애니메이션 변환이 완료된 정점들을 하나의 버퍼에 담게됩니다.

애니메이션 인스턴싱 / 그림자 2



- 계산쉐이더에서 출력버퍼로 사용해 애니메이션 업데이트 된 오브젝트 정점들을 계산합니다.

- 계산쉐이더를 이용한 계산 한번으로 이후에 애니메이션 관련 계산이 필요하지 않습니다.

- 그림자맵과 오브젝트 렌더링 등에서 입력버퍼로 재사용 할 수 있습니다.

인스턴싱 렌더링 코드

```
StructuredBuffer<vertex> gVertices;

cbuffer MeshInfo
{
    uint vertexBufferLen;
};

VertexOut SkinningInstancingVS(SkinnedInstanceVertexIn vin,
    uint vertexID : SV_VertexID, uint instanceID : SV_InstanceID)
{
    VertexOut vout;
    uint resultVertexID = vertexBufferLen * instanceID + vertexID;
    vout.PosH = gVertices[resultVertexID].PosH;
    vout.PosW = gVertices[resultVertexID].PosW;
```

- gVertices 버퍼에 인스턴싱 할 모든 정점들이 담겨있습니다.

- vertexBufferLen 변수는 인스턴싱 할 모델의 정점개수를 나타냅니다.

- 모델의 정점 개수, 인스턴싱ID, 정점ID를 이용해 이번 정점쉐이더에서 참조하는 gVertices 버퍼의 인덱스를 구할 수 있습니다.

- 이미 계산쉐이더에서 완료된 결과를 출력변수에 전달합니다.

- Bone Rendering

- 애니메이션을 실행할 때 뼈의 움직임을 확인 할 수 있도록 렌더링 한 모습입니다.
- 매 프레임마다 애니메이션에 따라 정점버퍼를 업데이트해 렌더링 합니다.

Bone Rendering

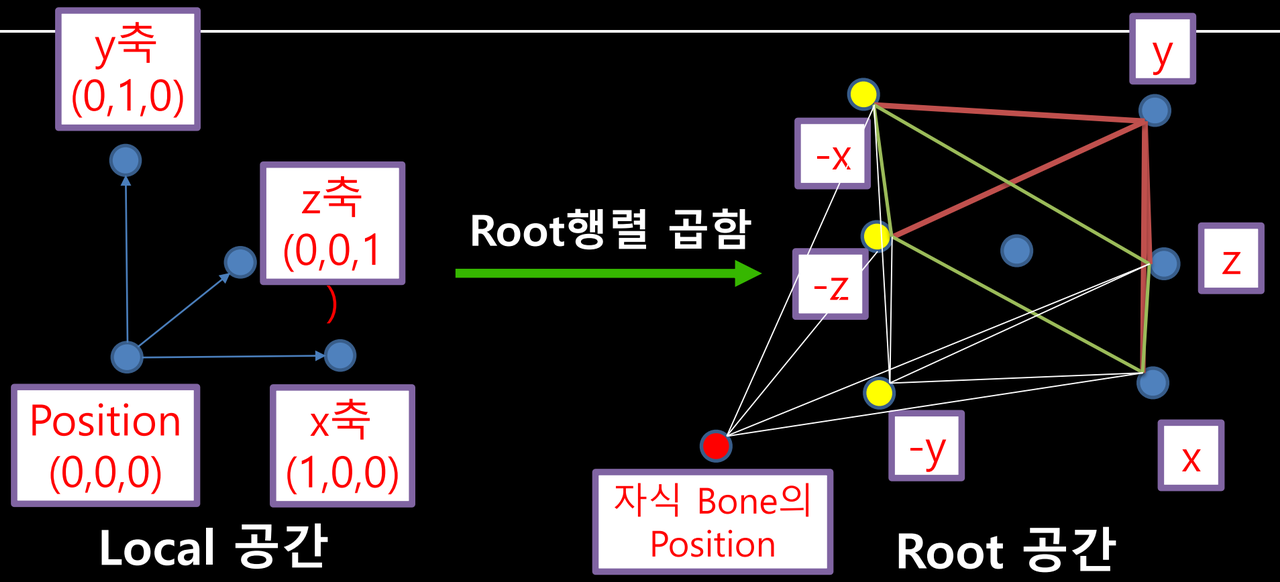
파란색은 클래스명 / 검정색은 멤버변수 / 초록색은 멤버함수 입니다.

Animator
Vector<4x4행렬> toRoots
Vector<UINT> parentIndices;

```
//뼈 하나당
for (int i = 0; i < boneSize; ++i)
{
    UINT firstIdx = i * 7;
    int parentIdx = m_Animator->boneDatas.m_parentIndices[i];
    UINT nextVertex = parentIdx * 7;
```

- 모든 뼈를 순회하면서 뼈의 부모를 참조하고,
  부모에서 현재 뼈로 연결하도록
  인덱스버퍼를 초기화 합니다.

- 각 뼈마다 7개의 정점을 가지고 있기때문에
  i번째 뼈의 시작 인덱스 firstIdx = i*7이 되고
  [i*7, i*8) 범위의 인덱스로 부모 뼈의 모든 정점에
  접근할 수 있습니다.



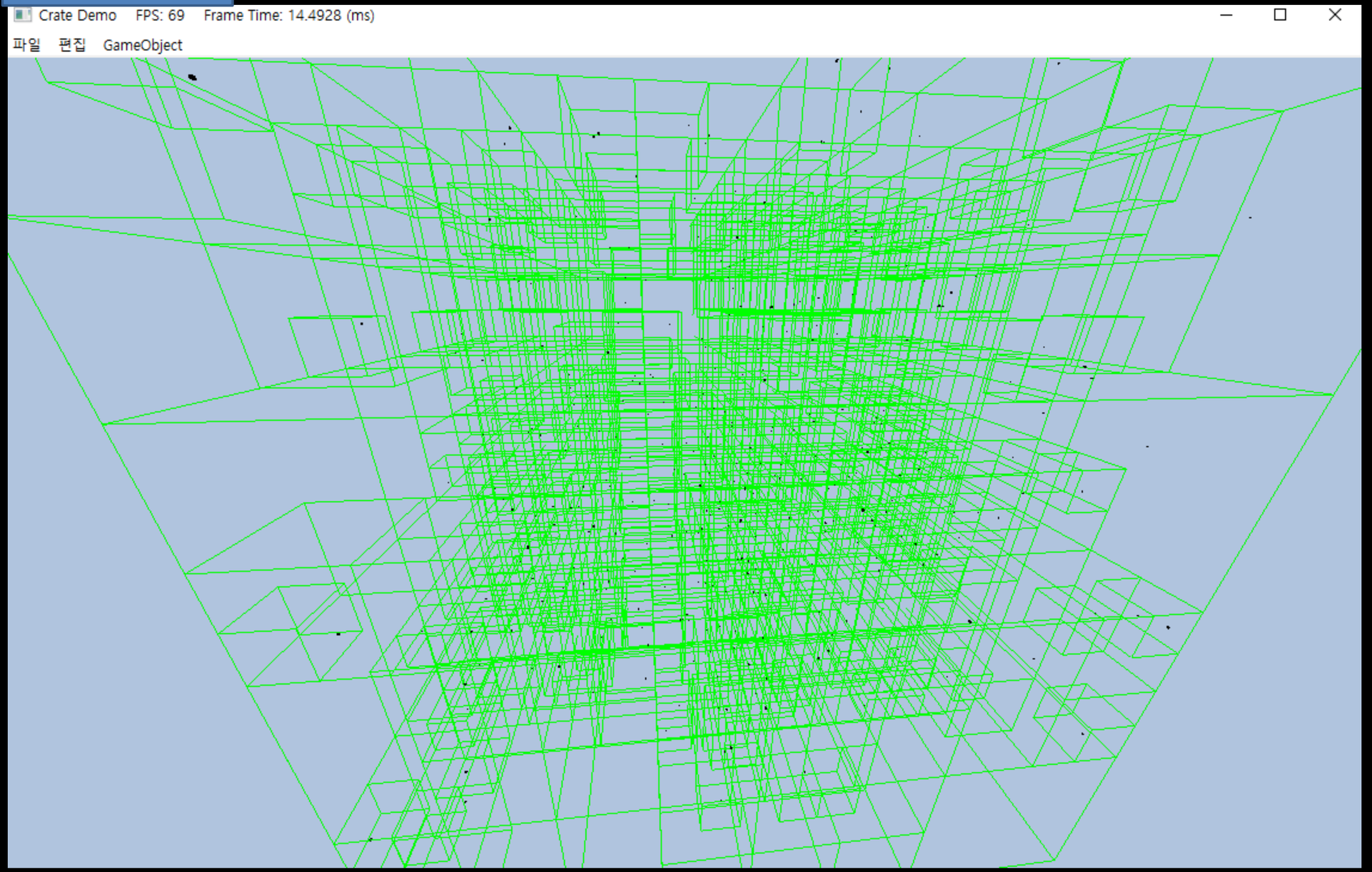
- Animator에서 이번 프레임에 업데이트 된 toRoot 행렬을 곱해
  모든 뼈대를 같은 Root 공간에 위치시켜 렌더링 합니다.

- Local 공간에서 임의로 설정한 xyz축을 Root 공간으로 변환합니다.

- 변환된 축의 정점 3개, 축의 반대방향 정점 3개, 변환된 원점 1개
  총 7개의 정점을 사용해 크리스탈 모양을 렌더링 해 뼈를 표현합니다.

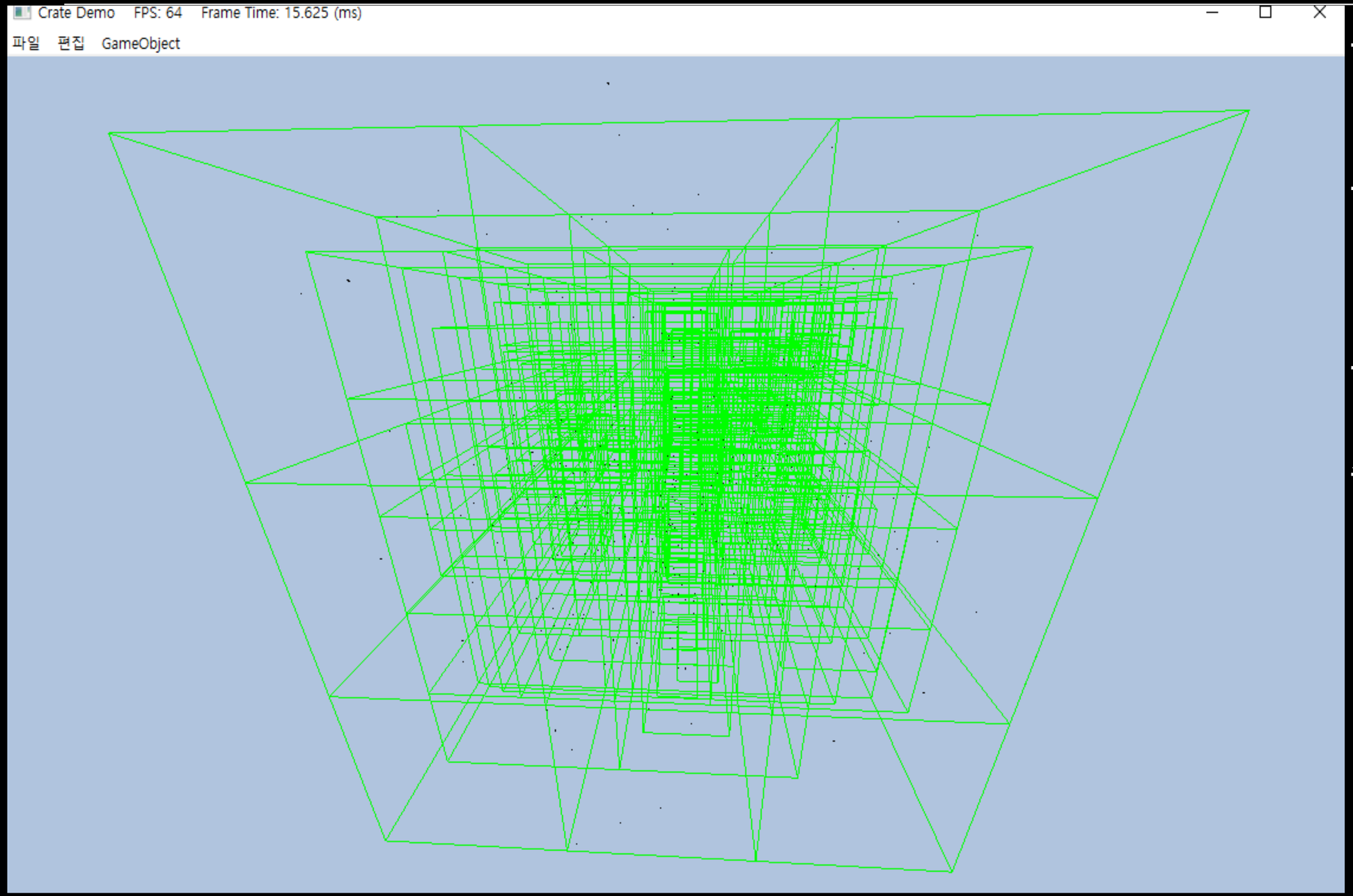
- 중간 평면에 사용하는 정점 4개 {x,-x,z,-z}에서 현재 Bone의
  자식Bone 위치로 연결해 Bone 사이의 움직임을 표현합니다.

- Octree



- Octree를 구현 해 각 노드들에 대해 절두체 선별하고 오브젝트를 렌더링하였습니다.

- 왼쪽 사진은 랜덤한 위치에 생성된 1000개의 큐브를 Octree에 넣고 렌더링 한 모습입니다.



- 일반 Octree가 아닌 느슨한 Octree를 이용해 렌더링 한 모습입니다.

- 큰 오브젝트가 Octree의 자식노드에 들어가지 못하는 상황을 개선하기 위해 구현하였습니다.

- 마찬가지로 랜덤한 위치에 생성된 1000개의 큐브에 대해 수행하였습니다.

- Octree 구현 코드

```
OctreeNode * Octree::BuildOctree(OctreeNode* parent, Renderer * renderer, int depth)
{
    if (depth > DEPTH_LIMIT)
    {
        parent->AddObject(renderer);
        return parent;
    }


    XNA::AxisAlignedBox objAABB = renderer->GetMesh()->GetAABB();
    //x,y,z축으로 원점에서 더해 줄 값
    XMFLOAT3 offset;
    float fStep = parent->m_radius*0.5;
    //자식노드들 순회
    for (int i = 0; i < 8; ++i)
    {
        OctreeNode* child = parent->m_children[i];
        if (child != nullptr)
        {
            //AABB가 자식오브젝트에 완전히 들어가는지 검사
            if (inNode(renderer, child->GetAABB()))
            {
                return BuildOctree(child, renderer, depth + 1);
            }
        }
        else //자식노드가 없는 경우
        {
            offset.x = (i & 1) ? -fStep : fStep;
            offset.y = (i & 4) ? -fStep : fStep;
            offset.z = (i & 2) ? -fStep : fStep;

            //새로 만들 자식노드의 AABB를 구한다.
            XNA::AxisAlignedBox nodeAABB;
            const XNA::AxisAlignedBox& parentAABB = parent->GetAABB();
            CreateNodeAABB(&nodeAABB,
                { parentAABB.Center.x + offset.x , parentAABB.Center.y + offset.y,
                parentAABB.Center.z + offset.z }, fStep);

            //자식노드에 오브젝트가 들어가는 경우 새로운 자식을 만든다.
            if (inNode(renderer, nodeAABB))
            {
                OctreeNode* childNode = new OctreeNode(nodeAABB, fStep);
                parent->m_children[i] = childNode;
                parent->m_children[i]->SetParent(parent);
                //생성한 노드의 AABB를 렌더러에 추가
                m_OctreeRenderer->AddBoundingBox(nodeAABB);
                return BuildOctree(childNode, renderer, depth + 1);
            }
        }
    }
    //어떤 자식노드에도 오브젝트가 포함되지 못하면 현재노드에 포함.
    parent->AddObject(renderer);

    return parent;
}
```

- 옥트리의 노드를 재귀적으로 탐색합니다.

- 현재 노드의 자식노드가 있다면 자식노드에 대해 현재 오브젝트가 들어 갈 수 있는지 검사한 뒤, true라면 자식노드를 탐색합니다.

- 자식노드가 없다면 자식노드를 생성 후 검사,탐색합니다.

- 어떤 자식노드에도 오브젝트가 들어갈 수 없다면 현재 노드에 오브젝트를 추가하고 반환합니다.

- 중점적으로 생각했던 부분

Object는 Component의 포인터만을 가지고
있고, 실제 Component는 Manager에서
벡터컨테이너로 관리해 Cache Hit를 높혔습니다.

```cpp
template<typename compType>
inline Component* ComponentMgr::SwapEnable(std::vector<compType>& vec, int & enableCount, int idx)
{
    //비활성화 컴포넌트인지 검사
    assert(idx >= enableCount);


    //비활성화된 컴포넌트를 제일 앞에 있는 비활성화된 컴포넌트와 바꿈
    std::swap(vec[enableCount], vec[idx]);

    //id와 index를 매핑하는 해쉬맵 업데이트
    idMap[vec[enableCount].id] = enableCount;
    idMap[vec[idx].id] = idx;

    //활성화된 카운트 수 증가
    enableCount++;

    return &vec[enableCount - 1];
}
```

Component를 활성화 시키는 함수입니다.
활성화된 컴포넌트의 개수보다 인덱스가 작으면 활성화입니다.
항상 앞쪽에 활성화된 컴포넌트를 모아두고,
렌더링이나 업데이트시 활성화된 앞쪽만 동작합니다.

```cpp
class ComponentMgr
{
private:
    std::vector<MeshRenderer> meshRenderers;
    std::vector<SkinnedMeshRenderer> skinnedMeshRenderers;

    //component의 id와 배열 index 매핑
    std::unordered_map<std::string, int> idMap;
    //component의 type 매핑
    std::unordered_map<std::string, ComponentType> typeMap;

private:
    //Component를 만들때 사용할 id넘버
    int creatingIdNum;
    //활성화 된 컴포넌트의 개수
    int enableCount_meshRenderer;
    int enableCount_skinnedMeshRenderer;
```

Component Manager는 각 컴포넌트를 vector로
관리합니다.
각 컴포넌트마다 활성화된 컴포넌트의 개수를 가지고
있습니다.