# 장진성

# DirectX11 포트폴리오

# 프로젝트 개요

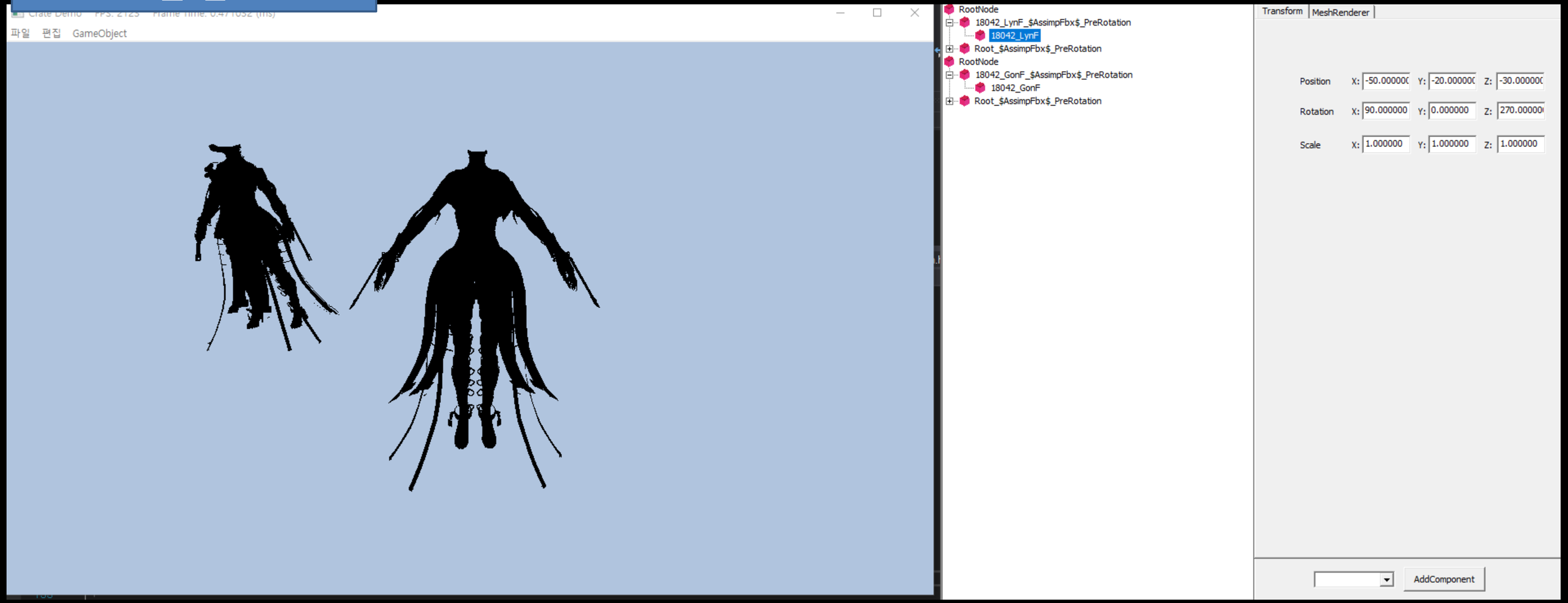
• 목표

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

• 목적

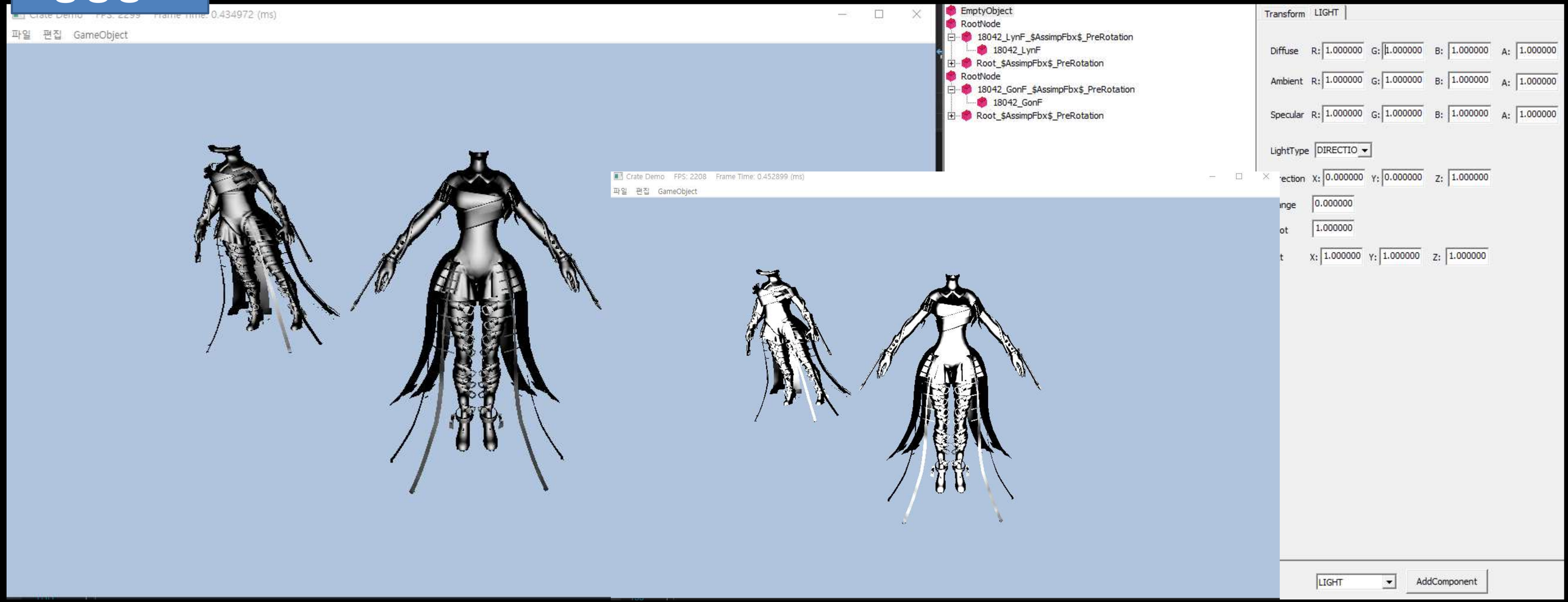
- 각 쉐이더 단계를 활용함으로써 공간과 렌더링파이프라인에 대한 이해
- 상용엔진에서 사용되는 기능들의 원리를 이해
- 프레임워크 설계 경험과 실력향상

# - FBX 모델 임포트



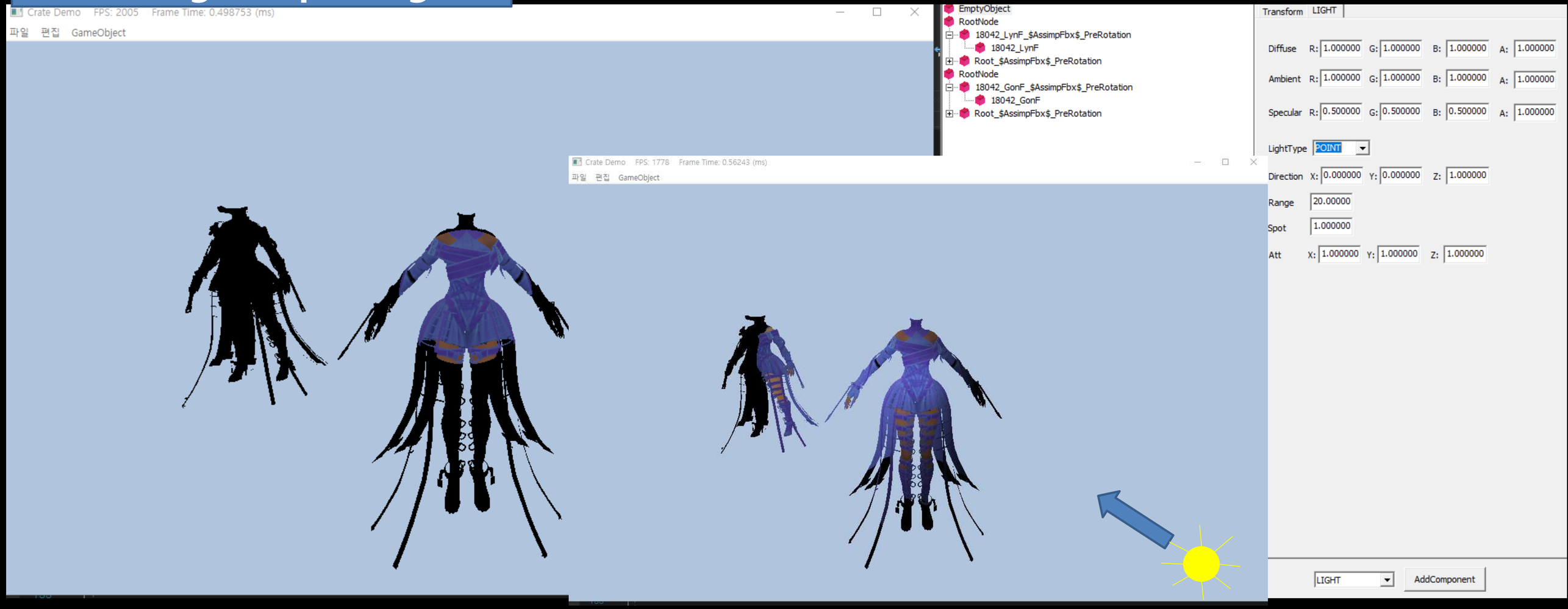

- Assimp Library로 fbx 파일을 읽어와 렌더링 합니다.
- 오브젝트는 컴포넌트의 집합으로 정의됩니다. 오브젝트는 Transform 컴포넌트를 기본으로 가지고 다른 컴포넌트들을 추가 할 수 있습니다.
- 새로운 기능을 구현할 때 Component 클래스를 상속한 클래스를 만들고, 추가하면 작동하도록 설계하였습니다.
- 렌더링은 MeshRenderer 컴포넌트를 오브젝트에 추가하여 실행됩니다.

# - 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.

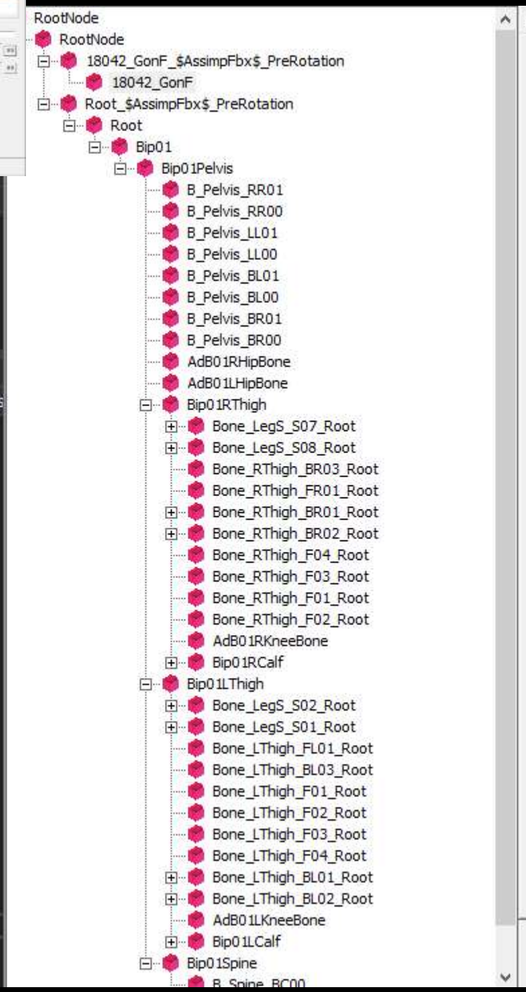
# - Point Light, Spot Light

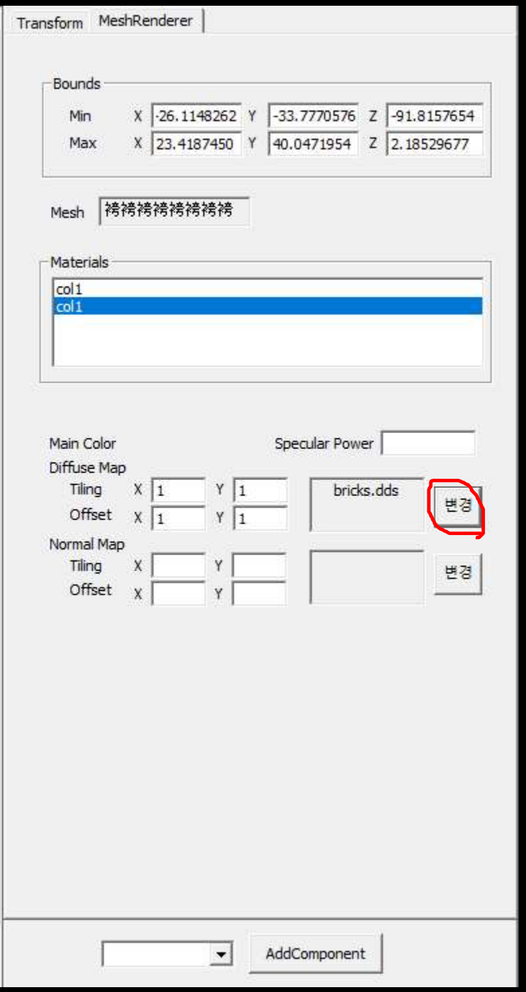



- Light 컴포넌트에서 LightType을 Point, Spot Light로 변경 할 수 있습니다.
- 조명의 방향, 범위 등 세부설정이 가능합니다.
- 좌측 사진은 Point Light, 우측 사진은 Spot Light를 사용한 모습입니다.

# - 텍스쳐 적용

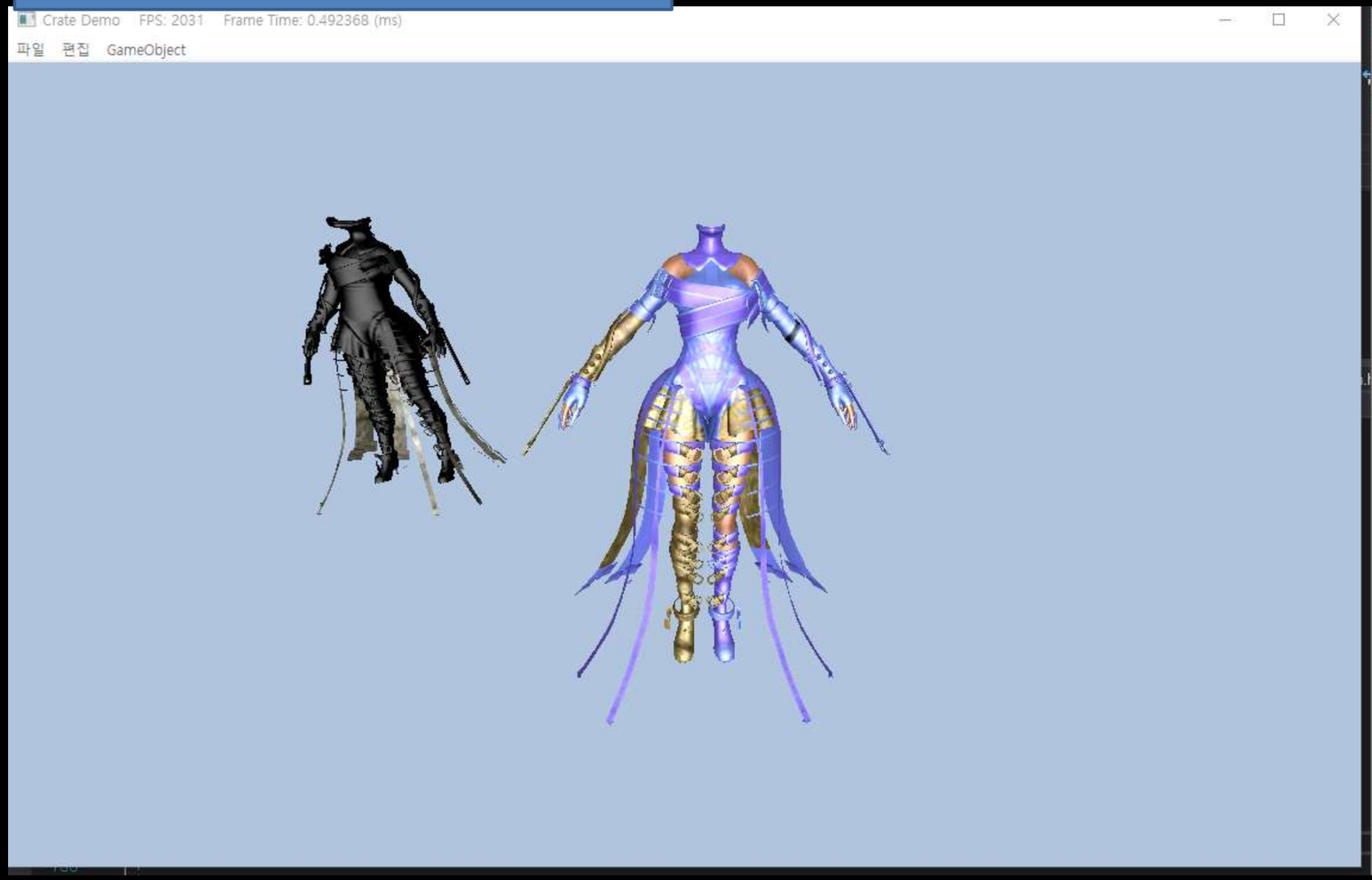


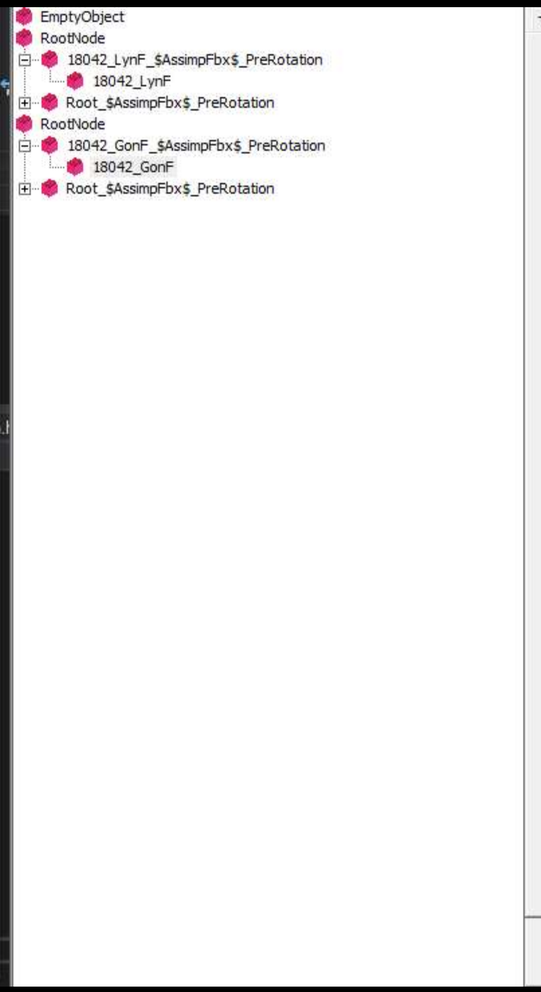



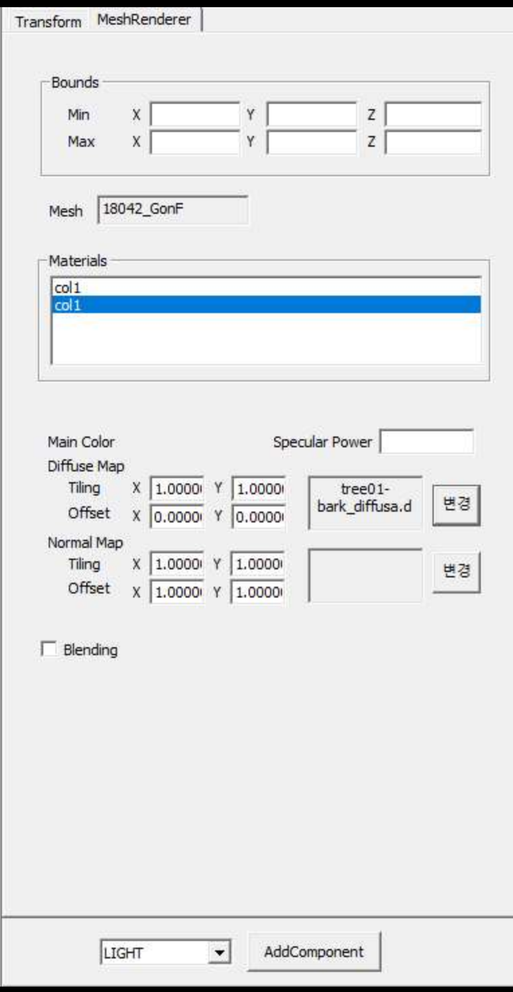




- 텍스쳐 파일을 불러와 Diffuse Map을 변경 할 수 있습니다.
- 명령패턴을 이용해 실행되며 Accelrator 단축키 설정을 이용해 Ctrl+z를 누르면 명령취소가 되게 구현했습니다.

# - Material 개별 텍스쳐 적용




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

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.

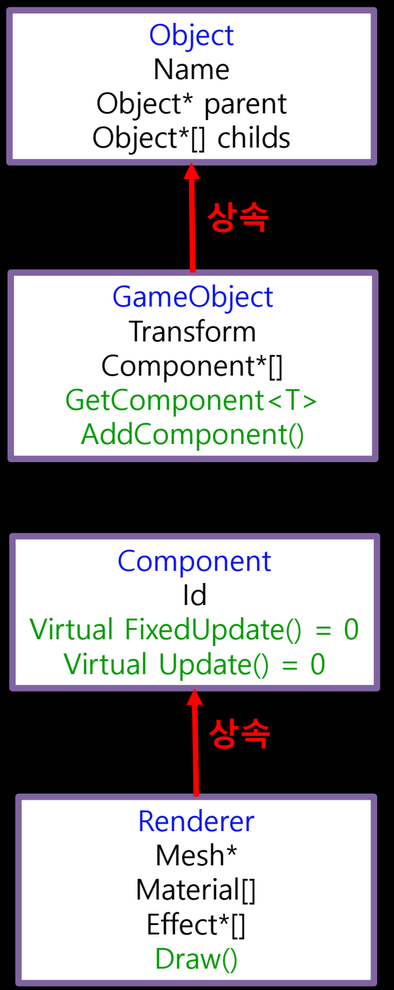
3. 오브젝트는 컴포넌트의 집합으로 이루어집니다.

4. 컴포넌트 매니저에서 모든 컴포넌트를 관리합니다.

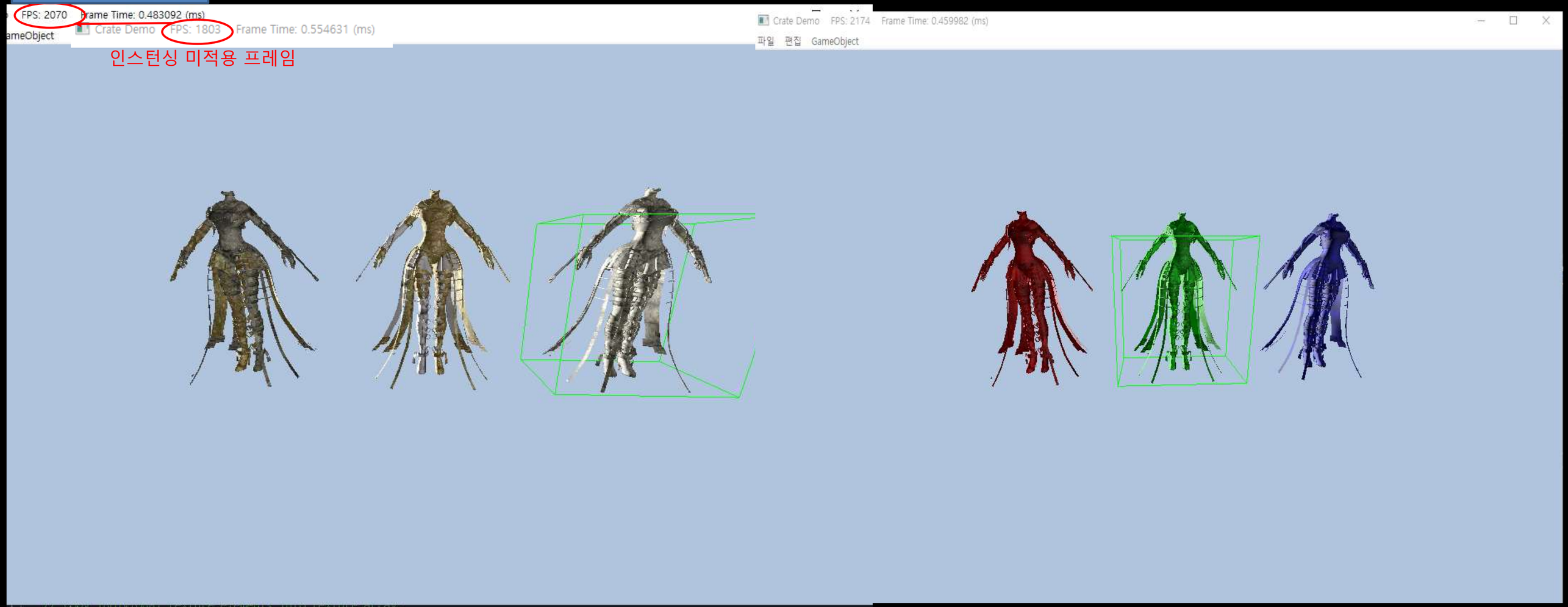
5. 컴포넌트 매니저의 모든 컴포넌트를 업데이트, 렌더링 합니다.

# 렌더링 구조


# 주요 클래스



# - 인스턴싱

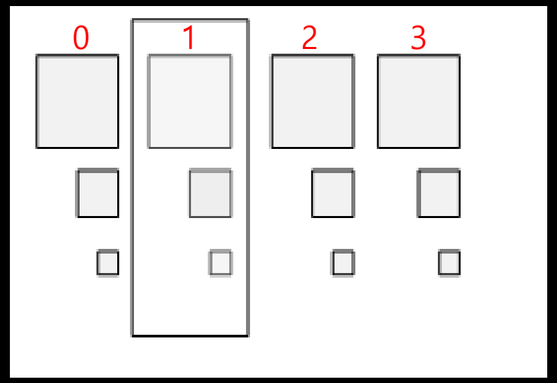



- 하드웨어 인스턴싱을 적용해 같은 메쉬를 사용하는 오브젝트를 한번의 그리기 호출로 렌더링하도록 하였습니다.
- 텍스쳐배열을 사용해 각 오브젝트, 각 메테리얼마다 다른 텍스쳐를 적용할 수 있습니다.

- 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;
```

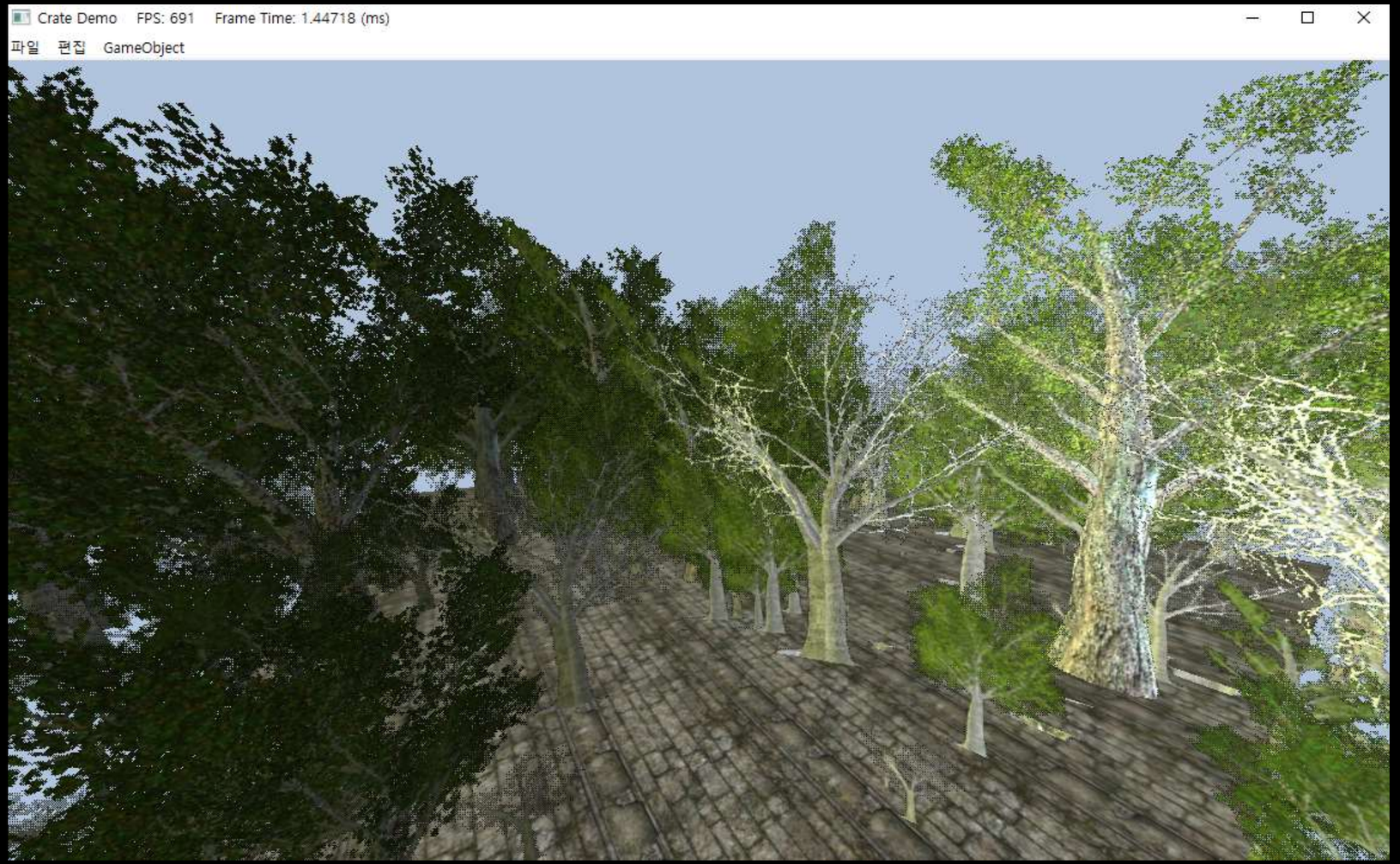
```cpp
class RayPicking
{
public:
    //스크린좌표 -> 시야공간에서의 반직선을 계산하는 함수
    static void ScreenToViewRay(XMVECTOR* rayOrigin, XMVECTOR* rayDir, float sx, float sy, D3D11_VIEWPORT * viewPort, XMMATRIX * projection
    //시야공간 반직선 -> 국소공간 반직선 변환 / 반환값 origin,dir
    static std::pair<XMVECTOR, XMVECTOR> ViewToLocalRay(XMVECTOR* 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.

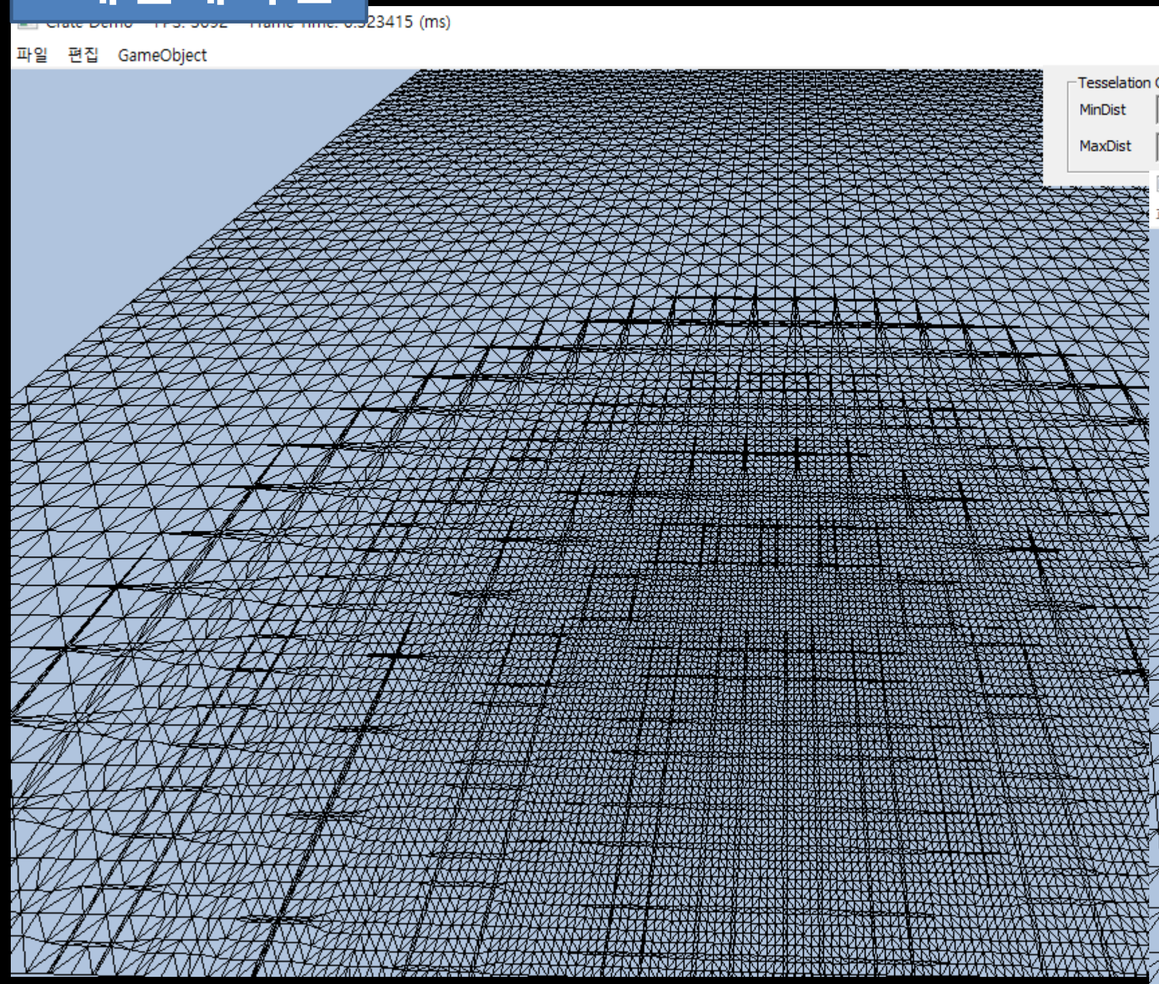
- 현재 선택된 오브젝트는 AABB를 렌더링합니다.

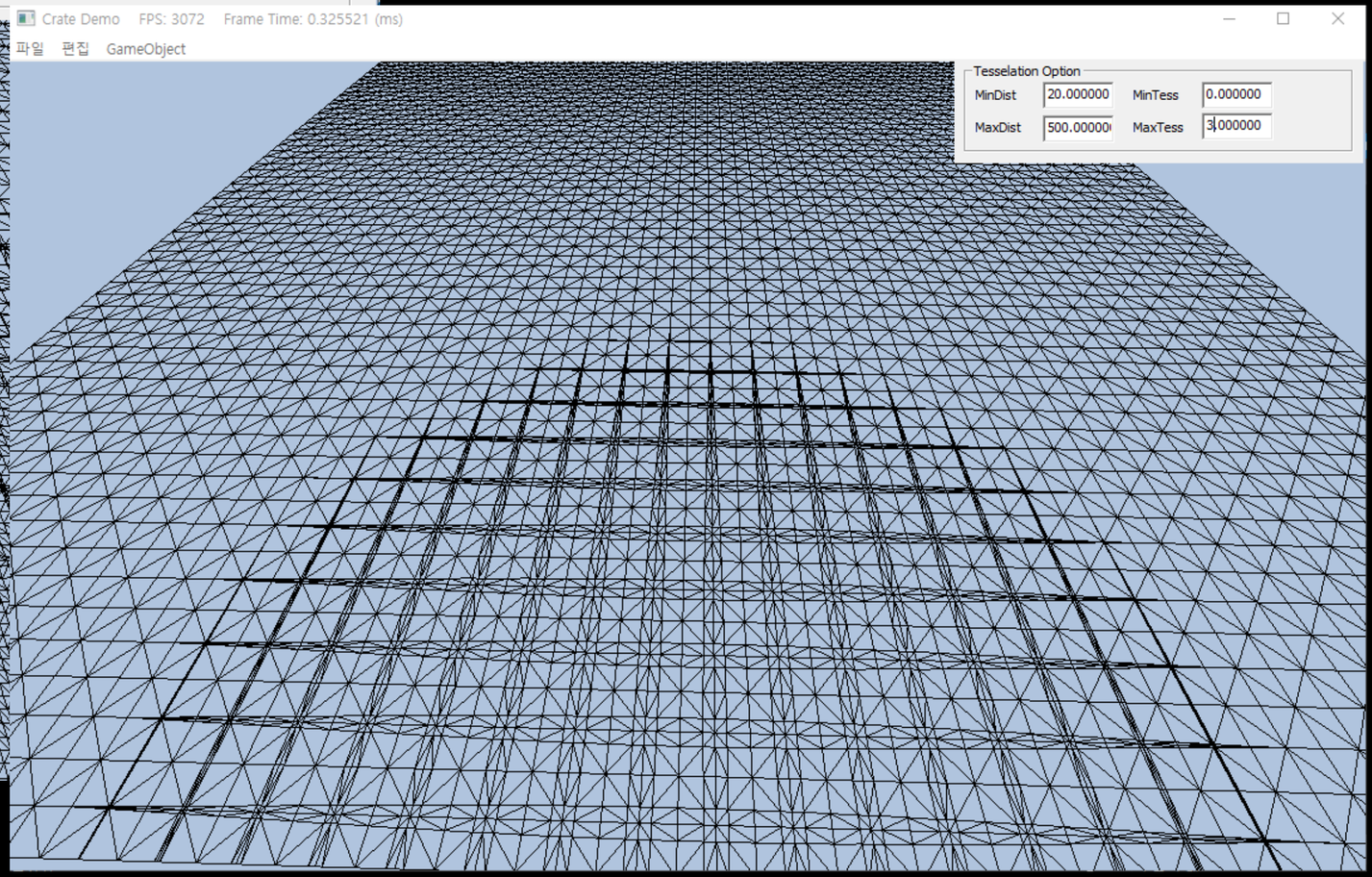
# - 기하쉐이더




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

# - 테셀레이션



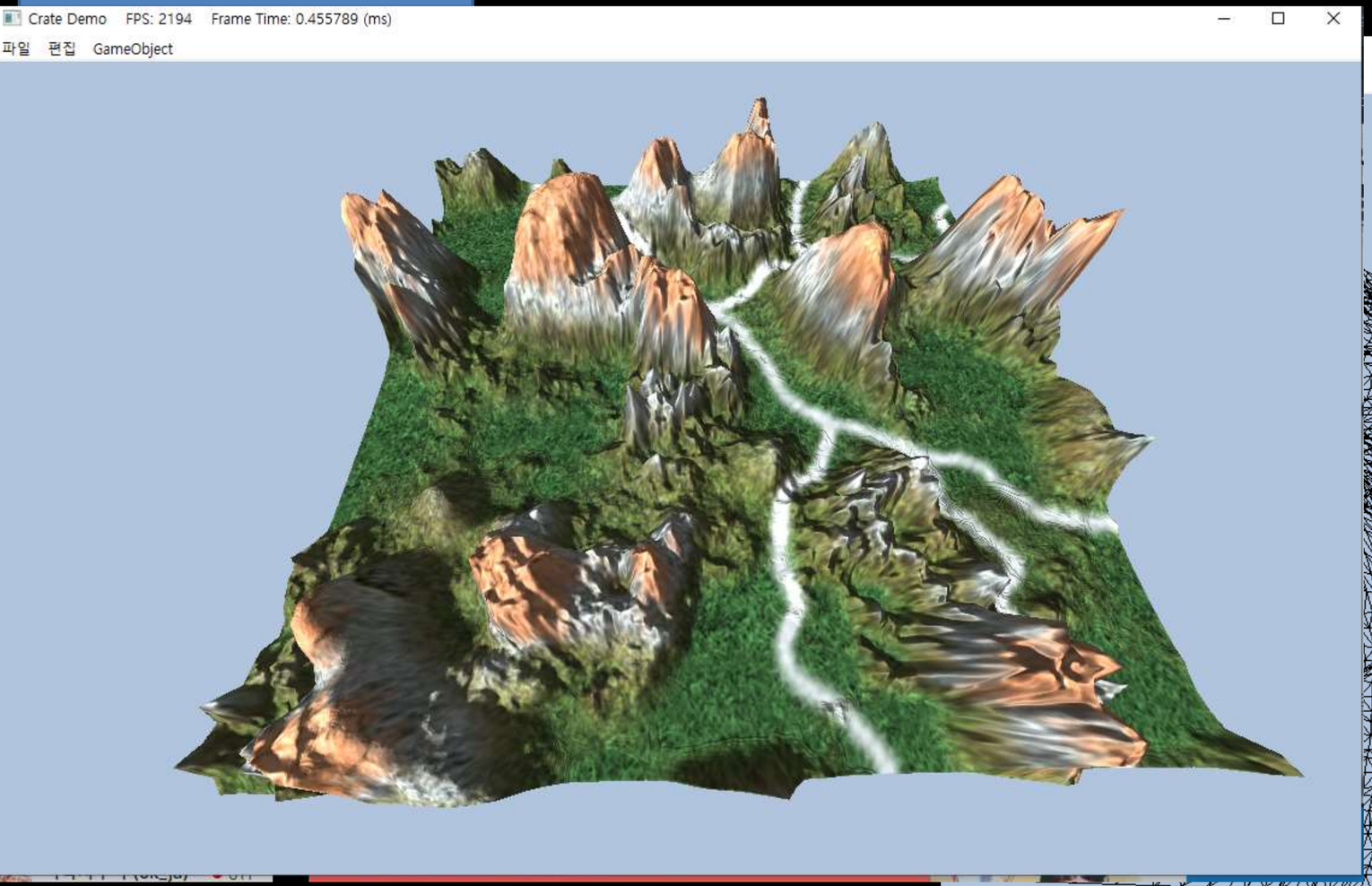



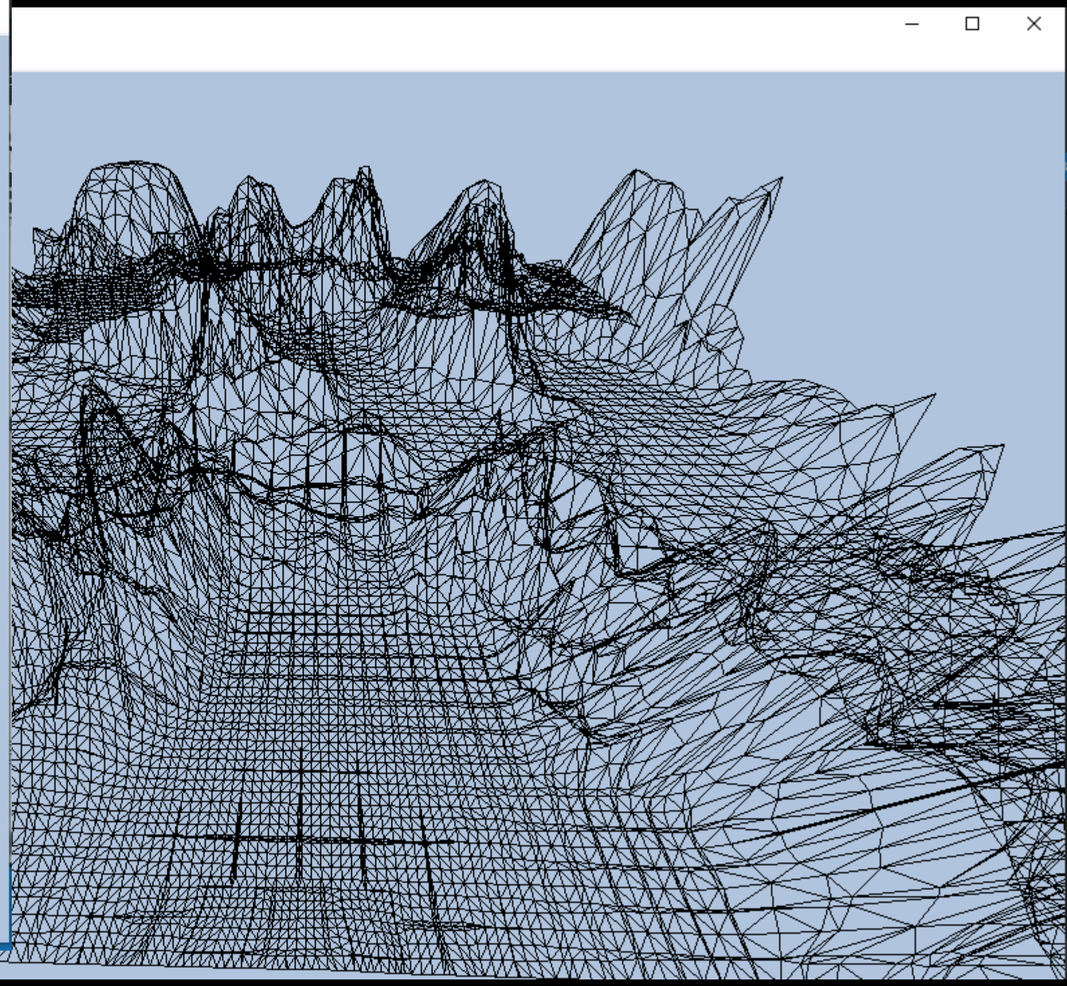

- 테셀레이션을 이용해 사각형 패치를 분할하였습니다. 카메라와 패치의 거리에 따라 테셀레이션 수준을 조정합니다.
- 카메라와 가까운 곳이 더 많이 분할되는 것이 나타납니다.
- 설정한 테셀레이션 계수 N에 따라 패치의 각 변을 최대 $2^n$만큼 분할합니다.
  왼쪽은 계수6, 오른쪽은 계수3을 적용한 모습입니다.

# - 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.

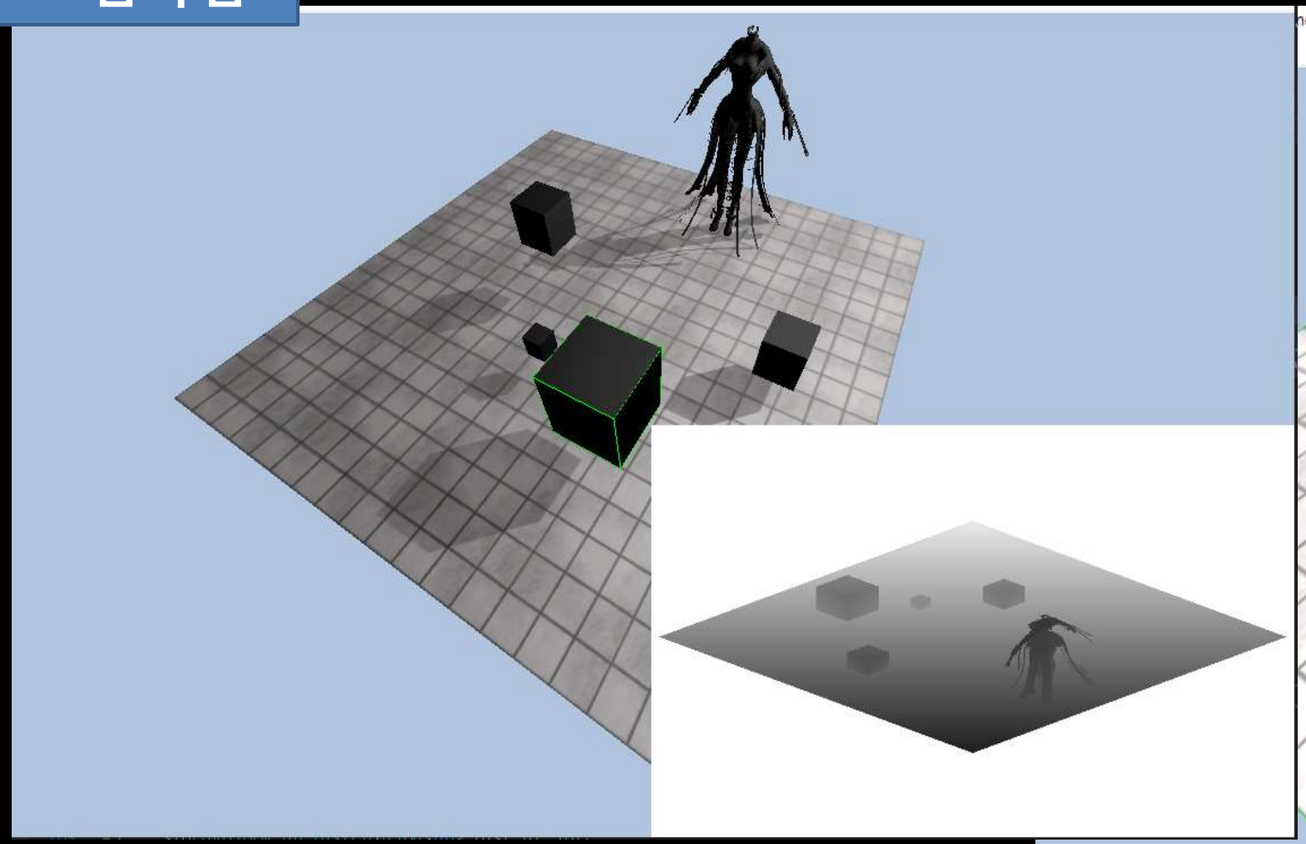
# - 테셀레이션 지형

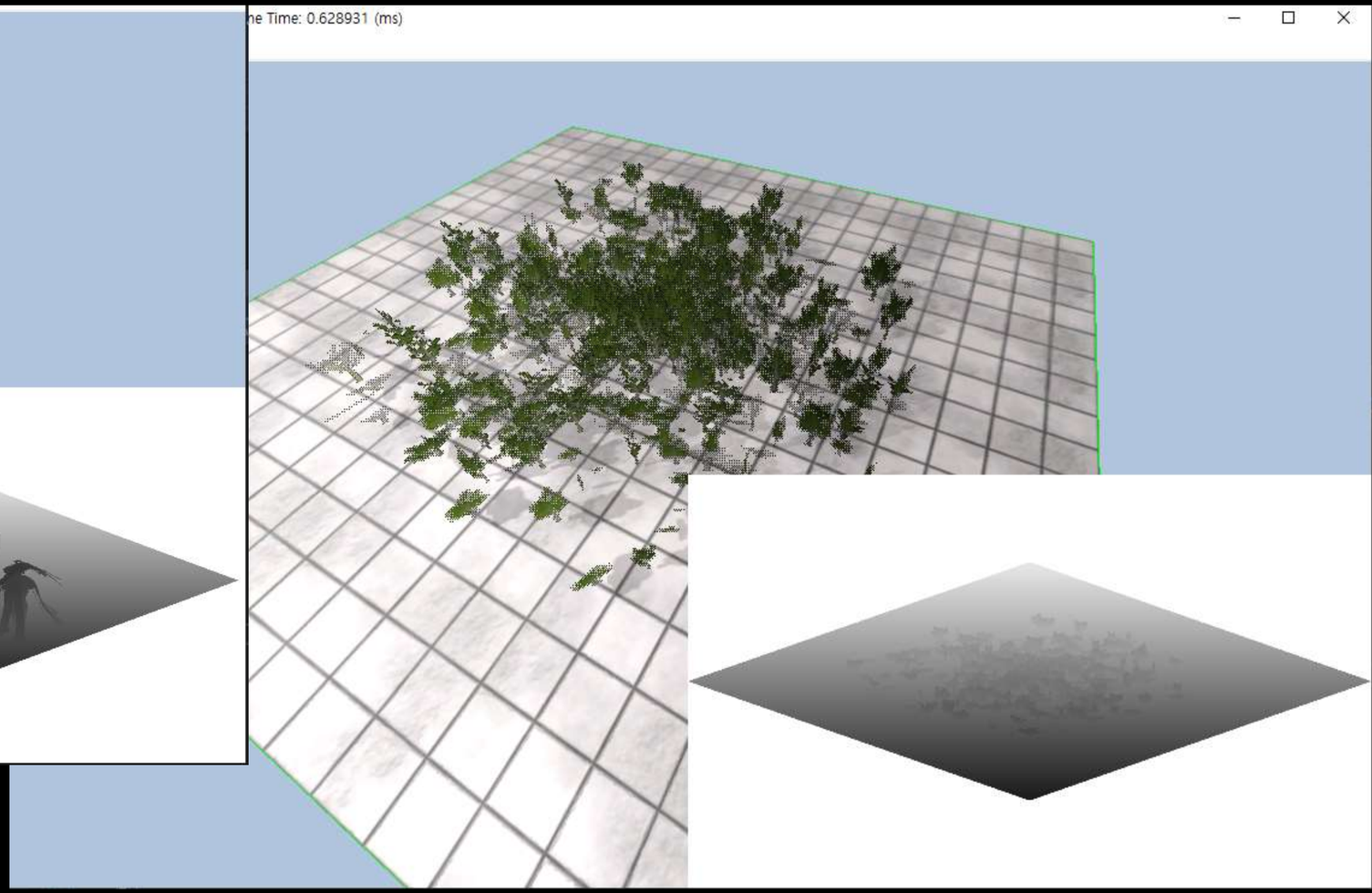


- 지형의 높이가 저장된 바이너리 파일을 읽어 텍스쳐를 생성하고 높이맵으로 사용하였습니다.
- Domain Shader에서 겹선형보간으로 텍스쳐 좌표를 구하고 높이맵의 해당좌표 픽셀값을 높이로 사용합니다.

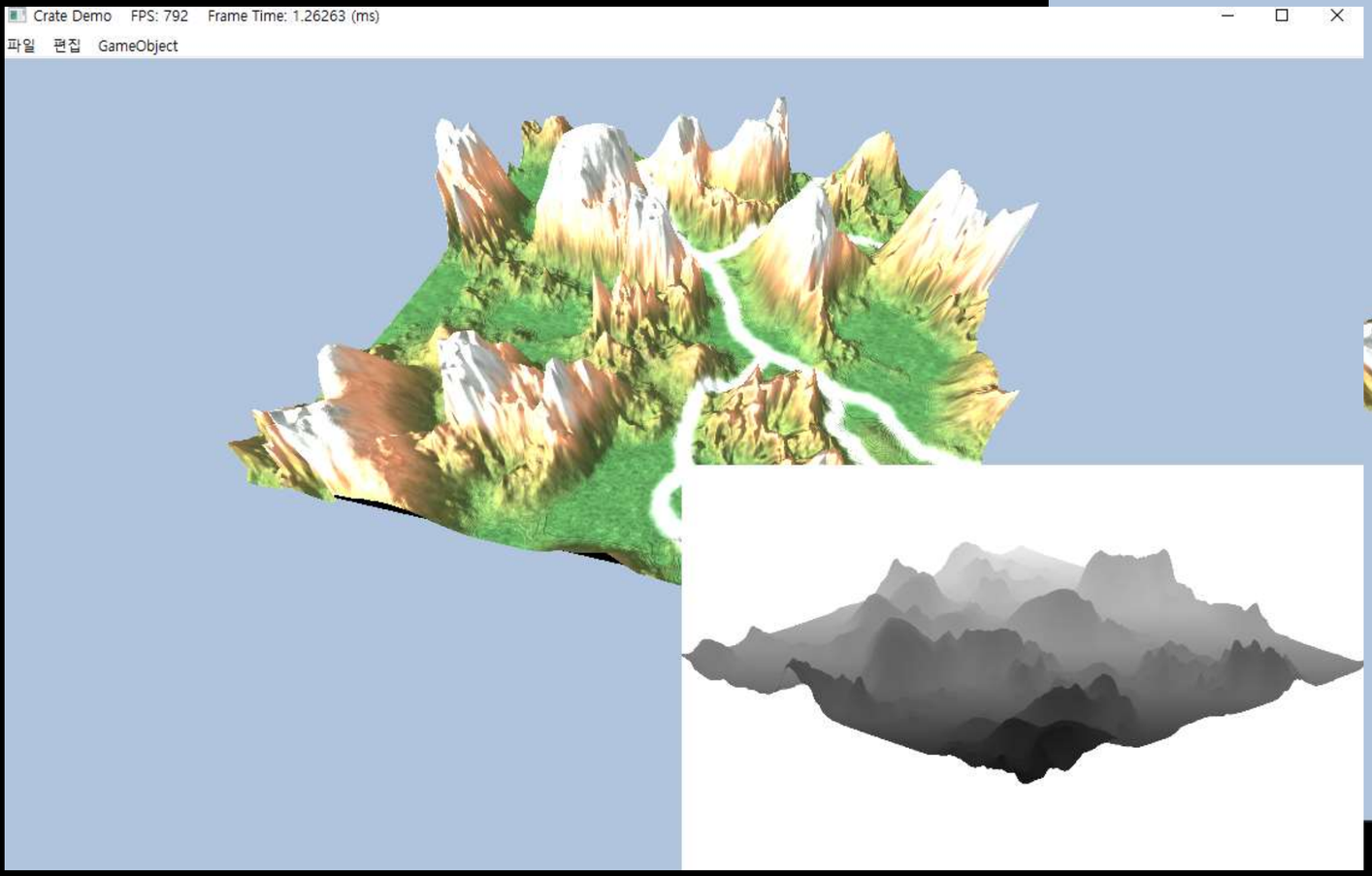
# - 그림자맵

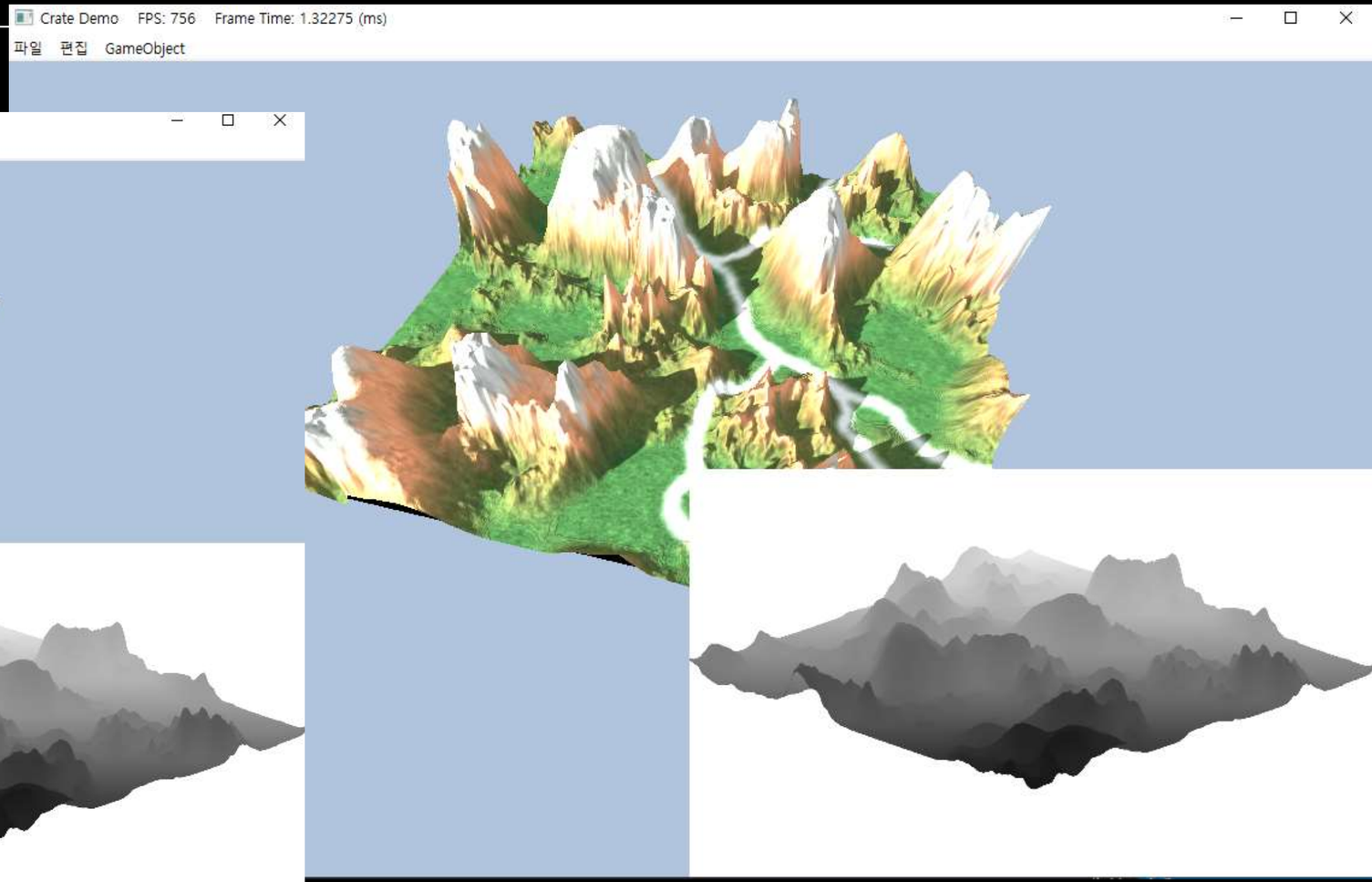





- 그림자맵을 이용해 그림자를 구현한 모습입니다.
- 각 사진의 우측 하단에 매 프레임마다 그림자맵 텍스쳐를 보여주고 있습니다.
- 오른쪽 사진은 기하 쉐이더를 이용해 나무의 그림자를 구현하였습니다.

# - 테셀레이션 지형 그림자




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

# - 그림자맵

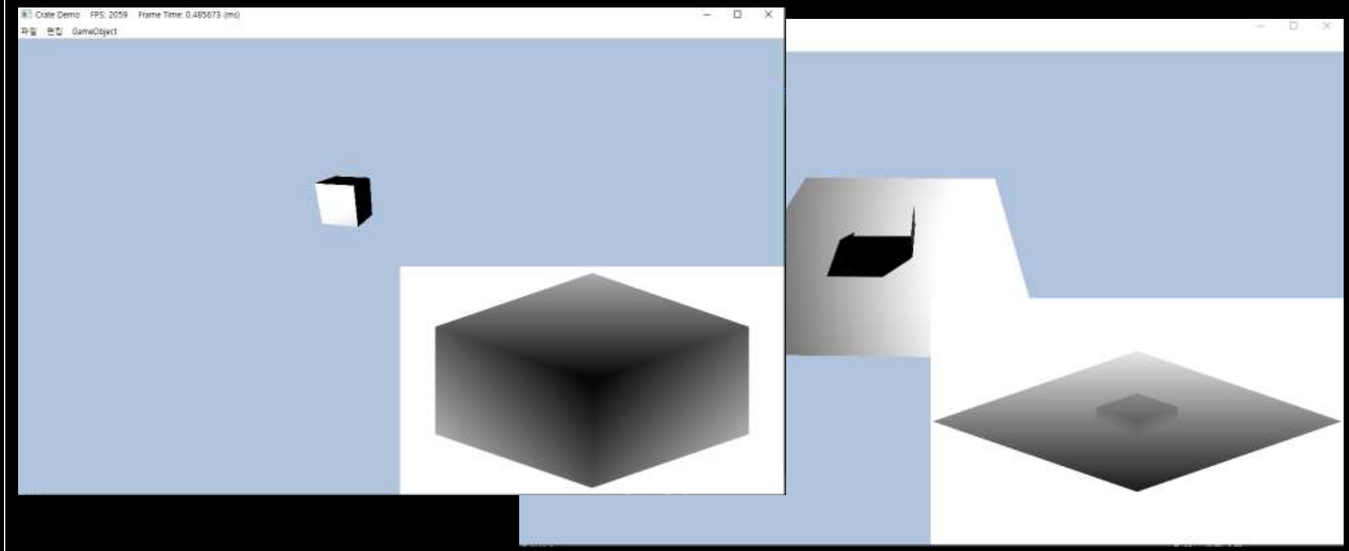
```
void ShadowMap::ComputeBoundingSphere(std::vector<Renderer*> renderers)
{
    XMFLOAT3 maxPosF(-MathHelper::Infinity, -MathHelper::Infinity, -MathHelper::Infinity);
    XMFLOAT3 minPosF(MathHelper::Infinity, MathHelper::Infinity, MathHelper::Infinity);

    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.

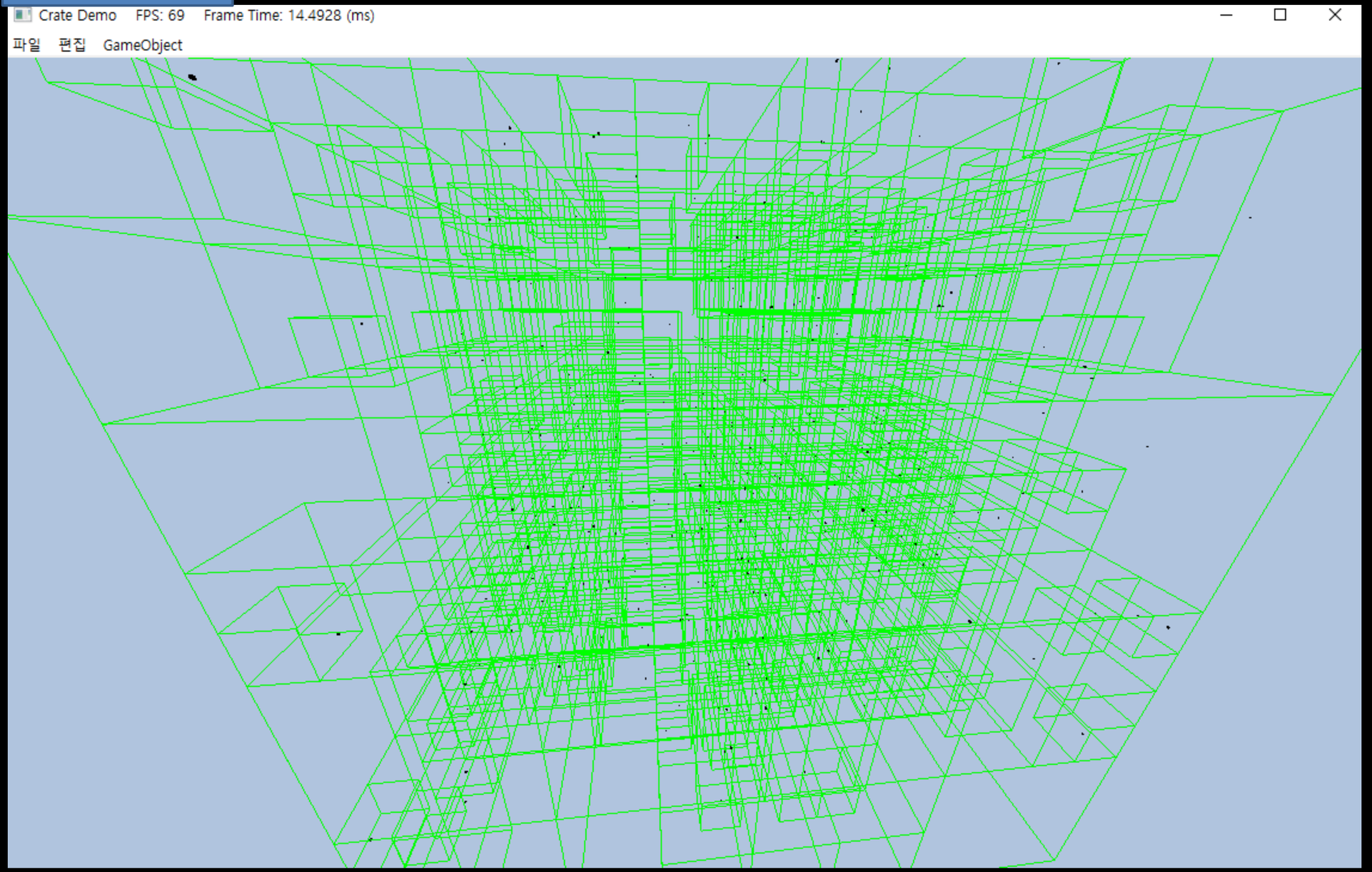
```
    //바라보는 점
    XMVECTOR targetPos = XMLoadFloat3(&m_boundingSphere.center);
    //평행광원의 위치(경계구의 중점에서 평행광방향의 반대방향*2 만큼 이동한 점)
    XMVECTOR lightPos = targetPos - 2.0f*m_boundingSphere.radius*lightDir;
    if(XMVector3Equal(lightDir, XMVectorZero()))
        return false;
```

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



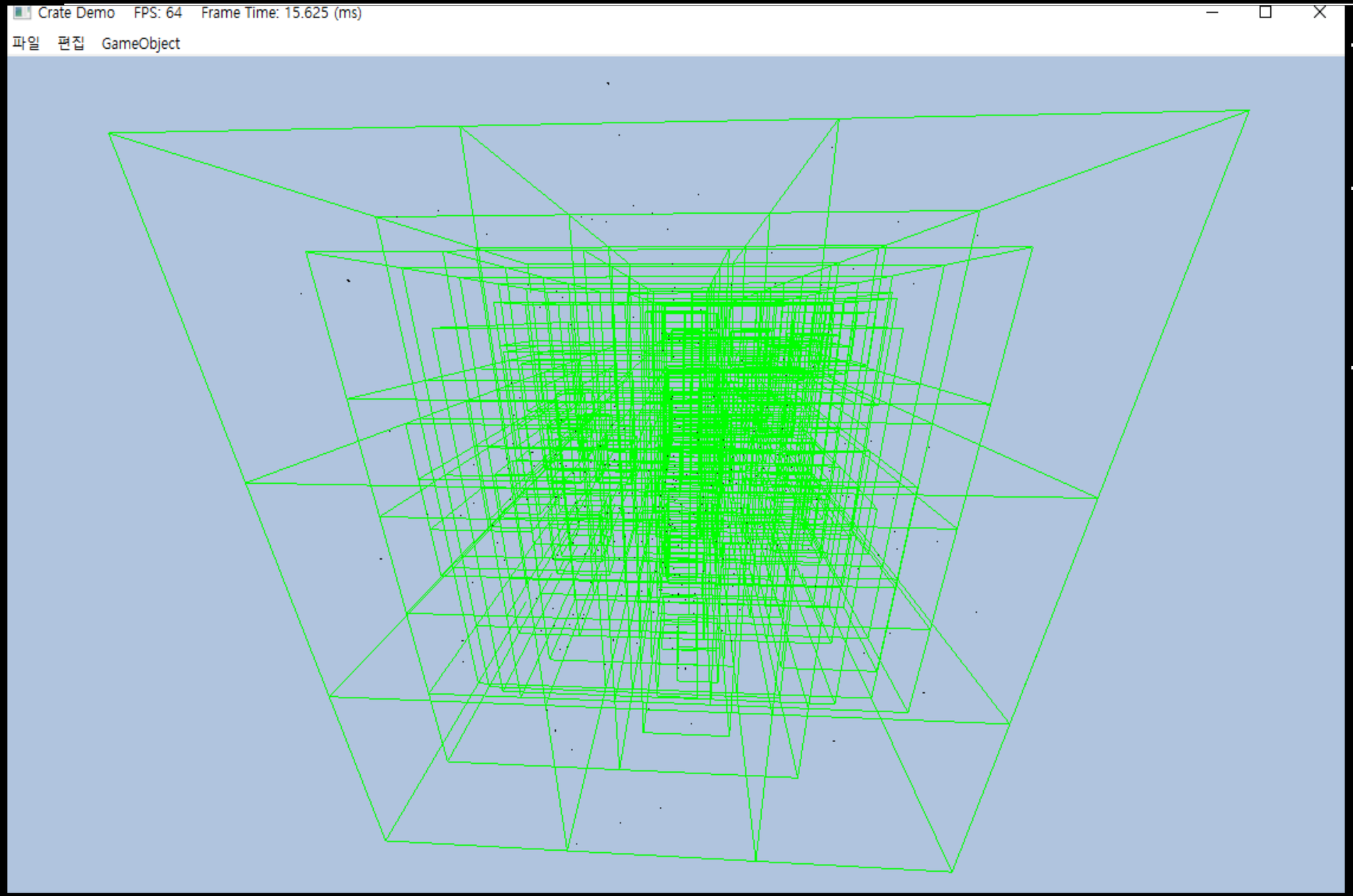
- 새로운 오브젝트가 추가되면 모든 오브젝트를 포함하는 그림자맵이 그려지는 모습입니다.

# - 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.