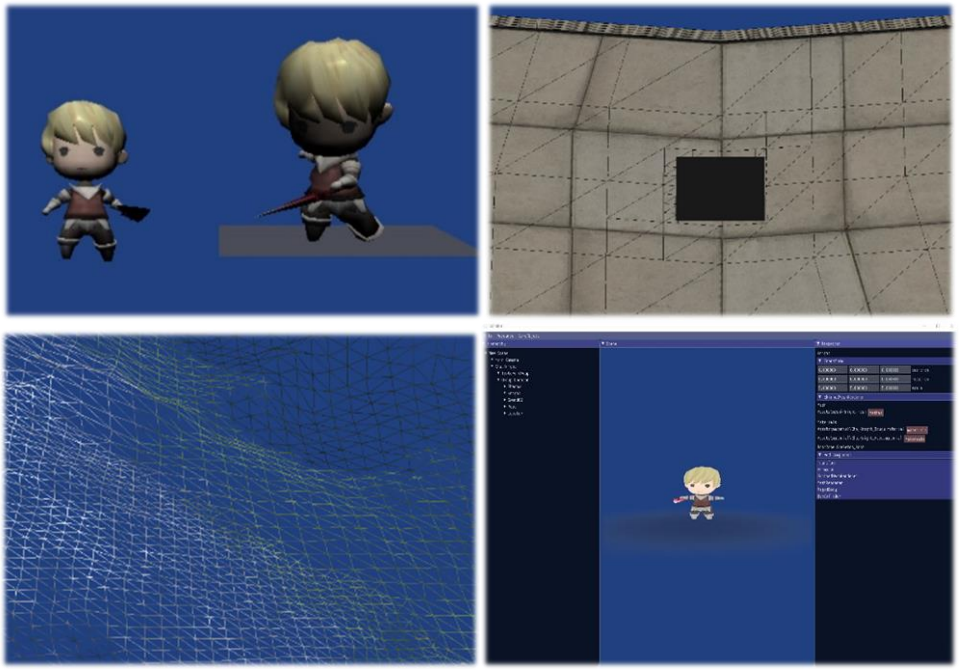
InwonEngine



개발 기간	2016.12 ~ 현재
개발 스펙	C++ / STL DirectX9 SDK FBX SDK 2017 Bullet3 Physics Boost Pool JsonCpp imGui
개발 내역	ECS 아키텍처를 이용한 게임엔진 설계 및 구현
요 약	게임엔진을 공부하기 위해 진행중인 프로젝트입니다.
성과 / 경과	졸업작품 심사 통과
소스 코드	https://github.com/InwonHwang/InwonEngine
영 상	https://www.youtube.com/watch?v=Wa7OsuCRW10&feature=youtu.be

간단 소개

엔진은 GameObect, Component, System기반으로 설계하였습니다. Component에는 정보를 저장하고 System에서는 로직을 수행합니다. GameObject는 Component를 갖고있는 컨테이너 역할을 합니다. 유니티 엔진처럼 계층구조를 지니고 있습니다.

FBX 로딩 및 애니메이션

FBX SDK 라이브러리를 사용하여 FBX내의 3D 모델 데이터 및 애니메이션 Json포맷으로 데이터를 저장하고 불러오는 작업을 하였습니다. 모든 리소스는 ResourceBase로부터 상속 받고 있으며 ResourceManager에 의해 생성됩니다.(내부적으로 Create함수를 호출).

```
class ResourceBase abstract
{
protected:
    shared_ptr<string>  _name;

public:
    ResourceBase() : _name(nullptr) { _name = make_shared<string>(); }
    virtual ~ResourceBase() {}

    virtual bool Create(void* pData, void* pResourceData, const string& name) abstract;
    virtual void Destroy() abstract;

    shared_ptr<string> GetName() const { return _name; }
};
```

<ResourceBase 클래스>

Animator 컴포넌트는 애니메이션 리소스와, 뼈대 정보를 갖고있으며 AnimationSystem에 의해 Aniamtion의 정보로 Transform 컴포넌트의 정보를 최신화합니다.

```
class Animator : public ComponentBase
{
    typedef vector<shared_ptr<Transform>> BoneVec;
    typedef vector<shared_ptr<Animation>> AnimationVec;

private:
    shared_ptr<ANIMATIONCALLBACKHANDLER> _animationHandler;
    shared_ptr<AnimationVec>             _animations;
    shared_ptr<BoneVec>                  _bones;
    shared_ptr<Animation>                _curAnim;
    shared_ptr<Animation>                _nextAnim;
```

<Animator Component>

애니메이션 재생할 때 자연스럽게 재생하기위해 재생되고 있는 애니메이션과 현재 재생할 애니메이션을 보간하는 함수를 제공합니다. 보간 방법은 재생중인 애니메이션의 행렬 + 현재 새로 재생될 애니메이션 행렬 / 2 결과로 만들어진 행렬로 뼈대 정보를 최신화 합니다.

```
float point = float(AppTimer->GetElapsedTime() - animHandler->_point) / float(length0);
float weight0 = 2.0f * point / animHandler->_fadeLength;
float weight1 = 2.0f - weight0;

D3DXMATRIX m, m0, m1;
Vector3 s, t;
Quaternion r;

int i = 0;
for (auto bone : *bones)
{
    auto animCurve0 = curAnim->GetAnimationCurve(i);
    auto animCurve1 = nextAnim->GetAnimationCurve(i++);

    getAnimatedMatrix(animCurve0, frame0, m0);
    getAnimatedMatrix(animCurve1, frame1, m1);

    m = (m0 * weight1 + m1 * weight0) / 2;

    D3DXMatrixDecompose(&s, &r, &t, &m);

    bone->SetLocalScale(s);
    bone->SetLocalRotation(r);
    bone->SetLocalPosition(t);
}
```

<AnimationSysem 내부 애니메이션 행렬 보간하는 코드 일부>

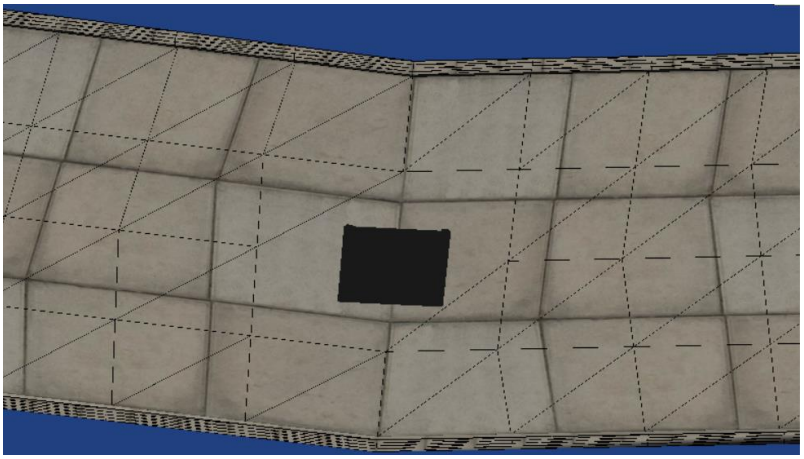
네비게이션메쉬

유니티의 NavgaionAgent 컴포넌트 기능을 제공하기 위해 구현한 기능입니다. 기본 알고리즘은 Mesh로부터 정점정보를 읽어와 네비게이션메쉬 데이터를 생성하고 장애물리스트를 등록하여 해당 장애물이 있는 지역의 폴리곤을 제거하는 방식으로 구현하였습니다. 폴리곤을 제거하는 방식은 장애물 오브젝트의 경계 박스 안에 폴리곤의 정점 중 1개라고 포함되면 제거하는 방식을 사용하였습니다.

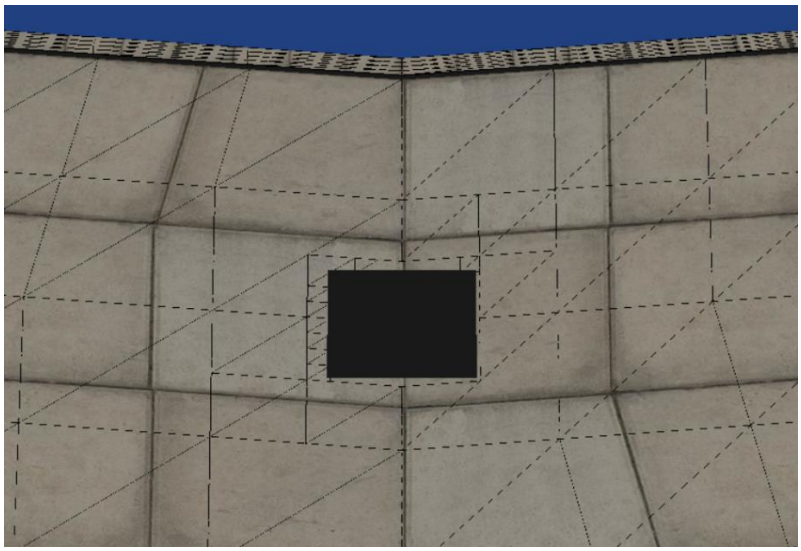
```
if ((v1.x < objectPos.x - boxSize.x || v1.z < objectPos.z - boxSize.z ||
    v1.x > objectPos.x + boxSize.x || v1.z > objectPos.z + boxSize.z) &&
    (v2.x < objectPos.x - boxSize.x || v2.z < objectPos.z - boxSize.z ||
        v2.x > objectPos.x + boxSize.x || v2.z > objectPos.z + boxSize.z) &&
        (v3.x < objectPos.x - boxSize.x || v3.z < objectPos.z - boxSize.z ||
            v3.x > objectPos.x + boxSize.x || v3.z > objectPos.z + boxSize.z))
{
    return eObtacleOut;
}



if (v1.x < objectPos.x + boxSize.x && v1.x > objectPos.x - boxSize.x &&
    v1.z < objectPos.z + boxSize.z && v1.z > objectPos.z - boxSize.z &&
    v2.x < objectPos.x + boxSize.x && v2.x > objectPos.x - boxSize.x &&
    v2.z < objectPos.z + boxSize.z && v2.z > objectPos.z - boxSize.z &&
    v3.x < objectPos.x + boxSize.x && v3.x > objectPos.x - boxSize.x &&
    v3.z < objectPos.z + boxSize.z && v3.z > objectPos.z - boxSize.z)
    return eObtacleIn;

return eObtaclePartiallyIn;
```

<해당 정점이 경계박스 내에 있는지 판단하는 함수 일부>

장애물이 존재하는 곳의 폴리곤을 없애는 경우 움직임이 제한되는 영역이 넓어지는 문제가 있었습니다. 문제를 해결하기 위해 삭제될 폴리곤을 세분화하고 다시 네비게이션메쉬에서 삭제될 폴리곤인지 판정하는 과정을 여러 번 수행하여 해결하였습니다.



<문제점>



<세분화 후>

터레인

3D 게임 프로그래밍 책을 보면서 추가했던 기능입니다.

Furstum Culling

Frustum 클래스는 오브젝트 또는 터레인이 절두체 안에 있는지 밖에 있는지 판단합니다. 카메라 컴포넌트내에서 생성합니다.

```cpp
bool Frustum::IsIn(const Vector3& position) const
{
    assert(_pPlanes && "plane is not created");

    for (int i = 0; i < 6; i++)
    {
        if (D3DXPlaneDotCoord(&_pPlanes[i], &position) > PLANE_EPSILON)
        {
            return false;
        }
    }
    return true;
}

bool Frustum::IsInSphere(const Vector3& position, float radius) const
{
    assert(_pPlanes && "plane is not created");

    for (int i = 0; i < 6; i++)
    {
        if (D3DXPlaneDotCoord(&_pPlanes[i], &position) > radius + PLANE_EPSILON)
        {
            return false;
        }
    }
    return true;
}
```

<Frustum>

QuadTree

지형을 빠르게 검색할 수 있다.(지형 위에 있는 오브젝트의 렌더링 여부를 빠르게 알 수 있다.)

```cpp
bool QuadTree::buildQuadTree(shared_ptr<HeightMapVec> pHeightMapVec)
{
    assert(pHeightMapVec && "null reference: pHeightMapVec");

    if (subDivide())
    {
        // 좌측상단과, 우측 하단의 거리를 구한다.
        Vector3 v = pHeightMapVec->data()[_corner[0]] - pHeightMapVec->data()[_corner[3]];
        // v의 거리값이 이 노드를 감싸는 경계구의 지름이므로,
        // 2로 나누어 반지름을 구한다.
        _radius = D3DXVec3Length(&v) / 2.0f;
        _pChild[0]->buildQuadTree(pHeightMapVec);
        _pChild[1]->buildQuadTree(pHeightMapVec);
        _pChild[2]->buildQuadTree(pHeightMapVec);
        _pChild[3]->buildQuadTree(pHeightMapVec);
    }
    return true;
}
```

<QuadTree 일부>

LOD

```
int QuadTree::getLODLevel(const shared_ptr<HeightMapVec> pHeightMapVec, const Vector3& cameraPos, float LODRatio) const
{
    assert(pHeightMapVec && "null reference: pHeightMapVec");
    float d = Vector3::GetDistance(pHeightMapVec->data()[_center], cameraPos);
    return max((int)(d * LODRatio), 1);
}

bool QuadTree::isVisible(const shared_ptr<HeightMapVec> pHeightMapVec, const Vector3& cameraPos, float LODRatio) const
{
    assert(pHeightMapVec && "null reference: pHeightMapVec");
    return (_corner[eTopRight] - _corner[eTopLeft] <= getLODLevel(pHeightMapVec, cameraPos, LODRatio));
}
```