# 3D게임프로그래밍
## -CHAPTER12-

SOULSEEK

# 목차

# LOD(LEVEL OF DETAIL)

# 1. LOD(LEVEL OF DETAIL)

- 메시 모델링 데이터의 정밀도를 단계별로 조정하는 기술
- 크게 정적 **LOD**와 동적 **LOD**로 나눌 수 있다.
- 지형 처리에서 사용되는 방법은 대부분 동적 **LOD**이다.

## LOD가 필요한 이유

- **LOD**기술은 속도와 질 사이의 타협 속에서 생겨난 기술이다. 질이 너무 좋은 화면은 속도가 느리게 되고, 반면 속도에만 너무 치중하게 되면 화면의 질이 떨어지는 게 당연한 것!質 을 적게 떨어뜨리면서 속도도 빠른 렌더링 기술을 개발하는 과정에서 태어난 기술이 **LOD**라는 기법이다.

- 매시 하나를 표현하는데도 매시를 몇 개로 분할하느냐에 따라 성능이 달라지는데, 카메라와 물체간의 거리를 측정하여 카메라와 가까운 거리에 있는 물체는 정밀한 매시를 사용하고, 거리가 멀어질 수록 낮은 단계의 매시를 사용하는 기법이 정적 **LOD**이다.

- 정적 **LOD**의 특징은 처음부터 매시의 정밀도가 정해져 있고, 이를 카메라와의 거리에 따라서 단계별로 바꿔치기 해가며 출력한다는 것이다. 이 방법의 장점은 연산이 간단하기 때문에 속도가 빠르다는 것, 그러나 여러 단계의 매시를 추가적으로 가지고 있어야 하기 때문에 메모리의 낭비가 심하고, 거리에 따라서 매시의 단계가 급격하게 변하기 때문에 튐 현상이 발생한다는 것이다. 이걸 해결한게 바로 동적 **LOD**이다.

## 동적 LOD 기법이란

- 카메라와 물체의 거리에 따라서 실시간으로 매시의 정밀도를 변화시키는 기법이다. 이 방법은 여러 가지 기법으로 분류되는데 일반적으로 매시 분할과 매시 간략화 기법으로 나뉜다.
- 장점은 거리에 따라서 자연스럽게 **LOD**가 이루어지기 때문에 튐 현상이 발생하지 않고, 낭비되는 메모리도 없다는 것이다. 그러나 매시 분할이나 간략화에 추가적인 연산이 필요하기 때문에 상대적으로 속도가 느리다는 것이 단점이다.

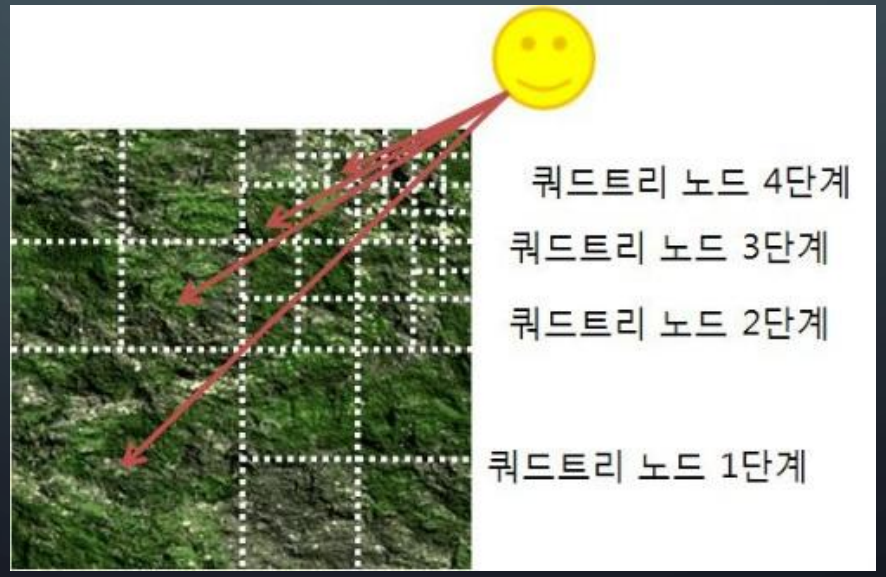
# 1. LOD(LEVEL OF DETAIL)

**LOD 적용 방법**

- **LOD**를 지형처리에 적용할 경우 지형의 변화율을 사용하는 방법과 카메라와 거리를 사용하는 방법이 있는데, 우리는 카메라와의 거리만을 연구대상으로 하자
- 간단하게, 카메라와의 거리가 가까울 수록 정밀한 매시를 사용하고, 멀어질수록 간략화된 매시를 사용하는 것이다.
- 문제는, 하나의 단일 매시 속에 이렇게 여러 단계의 매시를 공존시켜야 한다는 것이다.
- 여러 단계의 매시가 공존하기 때문에, 서로 다른 단계의 매시와 매시간의 연결이 매끄럽지 못할 경우 균열현상이 생긴다.

**LOD를 쿼드트리에 적용하기**

- **LOD**에 의해 세그먼트 분할이 이루어지는 단계를 자세히 살펴보면 이것이 쿼드트리의 자식 노드 분할과 닮아있다는 걸 알 수 있다. 우리는 쿼드트리를 사용하고 있기 때문에 상당히 간단하게 **LOD**를 구현할 수 있다.



# 1. LOD(LEVEL OF DETAIL)

쿼드트리를 사용한 **LOD**의 구현은 **ZQuadTree.h**에 두개의 함수를 추가하는 것으로 해결된다.

**ZQuadTree.h 수정**

```
float _GetDistance(D3DXVECTOR3* pv1, D3DXVECTOR3* pv2)
{
	return D3DXVec3Length(&(*pv2 - *pv1));
}

// 카메라와 현재 노드와의 거리값을 기준으로 LOD값을 구한다
int _GetLODLevel(TERRAINVERTEX* pHeightMap, D3DXVECTOR3* pCamera, float fLODRatio)
{
	float d = _GetDistance((D3DXVECTOR3*)(pHeightMap+m_nCenter), pCamera);
	return max((int)(d*fLODRatio), 1);
}

// 현재 노드가 LOD등급으로 볼 때 출력이 가능한 노드인가?
BOOL _IsVisible(TERRAINVERTEX* pHeightMap, D3DXVECTOR3* pCamera, float fLODRatio)
{
	return ((m_nCorner[CORNER_TR] - m_nCorner[CORNER_TL]) <= _GetLODLevel(pHeightMap,
	pCamera, fLODRatio));
}
```

**_IsVisible()**함수는 카메라와 현재 노드와의 거리에 따른 **LOD** 등급을 구해서 쿼드트리의 출력 여부를 결정한다. 이때, **LOD** 등급을 구해주는 함수는 **_GetLODLevel()**이고, 카메라와 노드 중앙 정점과의 거리를 구해주는 함수는 **_GetDistance()**다. 이것을 추가하는 것만으로도 우리는 훌륭한 **LOD**지형을 구현할 수 있다.
문제는 **LOD**등급이 서로 다른 매시들끼리 공존하기 때문에 **LOD**등급이 다른 경계부분에서 균열이 발생하는 것을 막지 못하고 있다.

# 1. LOD(LEVEL OF DETAIL)

인자가 수정되는 곳은 연쇄적으로 다 수정해줘야한다.

**ZTerrain.h 수정**

```
HRESULT Create( LPDIRECT3DDEVICE9 pDev, D3DXVECTOR3* pvfScale, float fLODRatio, LPSTR
                lpBMPFilename, LPSTR lpTexFilename[MAX_TERRAIN_TEX] );
```

**ZTerrain.cpp 수정**

```
HRESULT ZTerrain::Create( LPDIRECT3DDEVICE9 pDev, D3DXVECTOR3* pvfScale, float
                fLODRatio, LPSTR lpBMPFilename, LPSTR lpTEXFilename[4] )
{
    m_pd3dDevice = pDev;
    m_vfScale = *pvfScale;
    m_fLODRatio = fLODRatio;
    if( FAILED( _BuildHeightMap( lpBMPFilename ) ) ) { _Destroy(); return E_FAIL; }
    if( FAILED( _LoadTextures( lpTEXFilename ) ) ) { _Destroy(); return E_FAIL; }
    if( FAILED( _CreateVIB() ) ) { _Destroy(); return E_FAIL; }
    m_pQuadTree = new ZQuadTree( m_cxDIB, m_czDIB );
    if( FAILED( _BuildQuadTree() ) ) { _Destroy(); return E_FAIL; }


    return S_OK;
}
```

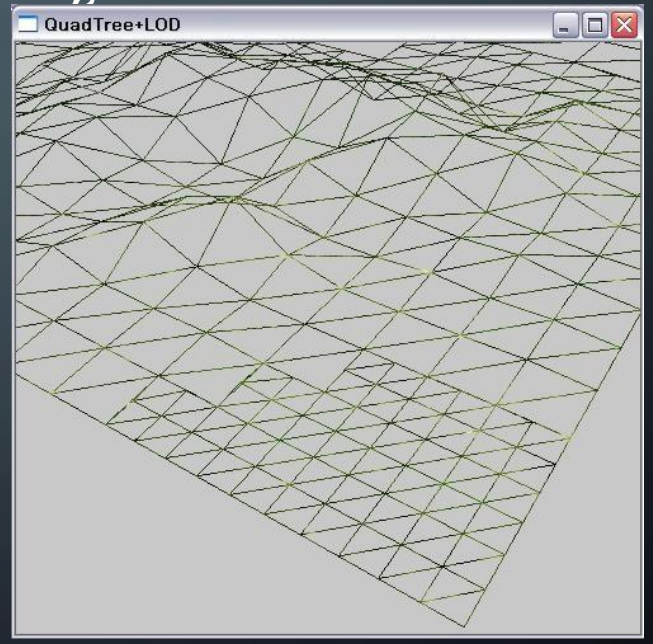
# 1. LOD(LEVEL OF DETAIL)

**ZQuadTree.cpp 수정**

```
int ZQuadTree::_GenTriIndex( int nTris, LPVOID pIndex, TERRAINVERTEX* pHeightMap, ZFrustum*
pFrustum, float                                    fLODRatio )
{
    // 컬링된 노드라면 그냥 리턴
    if( m_bCulled )
    {
        m_bCulled = FALSE;
        return nTris;
    }

    // 현재 노드가 출력되어야 하는가?
    if( _IsVisible( pHeightMap, pFrustum->GetPos(), fLODRatio ) )
    {
        #ifdef _USE_INDEX16
        LPWORD p = ((LPWORD)pIndex) + nTris * 3;
        #else
        LPDWORD p = ((LPDWORD)pIndex) + nTris * 3;
        #endif
        *p++ = m_nCorner[0];
        *p++ = m_nCorner[1];
        *p++ = m_nCorner[2];
        nTris++;
        *p++ = m_nCorner[2];
        *p++ = m_nCorner[1];
        *p++ = m_nCorner[3];
        nTris++;

                                                    // 자식 노드들 검색
                                                    if( m_pChild[CORNER_TL] ) nTris = m_pChild[CORNER_TL]->
                                                    _GenTriIndex( nTris, pIndex, pHeightMap, pFrustum, fLODRatio );
                                                    if( m_pChild[CORNER_TR] ) nTris = m_pChild[CORNER_TR]->
                                                    _GenTriIndex( nTris, pIndex, pHeightMap, pFrustum, fLODRatio );
                                                    if( m_pChild[CORNER_BL] ) nTris = m_pChild[CORNER_BL]->
                                                    _GenTriIndex( nTris, pIndex, pHeightMap, pFrustum, fLODRatio );
                                                    if( m_pChild[CORNER_BR] ) nTris = m_pChild[CORNER_BR]->
                                                    _GenTriIndex( nTris, pIndex, pHeightMap, pFrustum, fLODRatio );

                                                    return nTris;
                                                }
```

# 1. LOD(LEVEL OF DETAIL)

```
// 삼각형의 인덱스를 만들고, 출력할 삼각형의 개수를 반환한다.
int ZQuadTree::GenerateIndex( LPVOID pIndex, TERRAINVERTEX* pHeightMap, ZFrustum*
                              pFrustum, float fLODRatio )
{
    // 먼저 프러스텀 컬링을 해서 컬링될 노드들을 배제한다.
    _FrustumCull( pHeightMap, pFrustum );

    // 출력할 폴리곤의 인덱스를 생성한뒤, 폴리곤의 개수를 리턴한다.
    return _GenTriIndex( 0, pIndex, pHeightMap, pFrustum, fLODRatio );
}
```

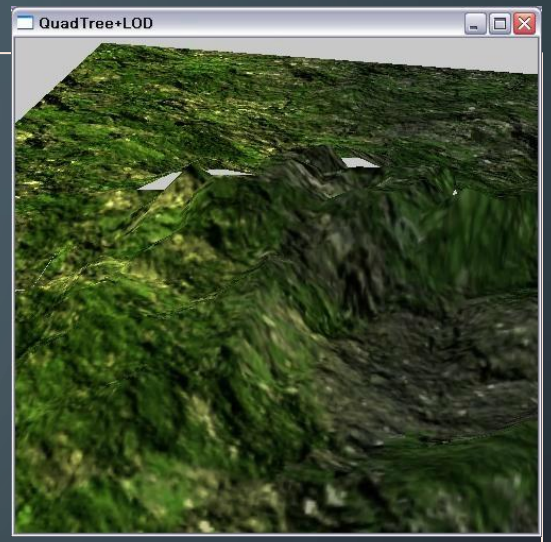



HeightMap.cpp 수정

```
HRESULT InitObjects()
{
    LPSTR tex[4] = { "tile2.tga", "", "", "" };
    D3DXVECTOR3 vScale;


    vScale.x = vScale.z = 1.0f; vScale.y = 0.1f;
    g_pLog = new ZFLog( ZF_LOG_TARGET_WINDOW );
    g_pCamera = new ZCamera;
    g_pFrustum = new ZFrustum;
    g_pTerrain = new ZTerrain;
    g_pTerrain->Create( g_pd3dDevice, &vScale, 0.1f, BMP_HEIGHTMAP, tex );


    return S_OK;
}
```

# 균열방지

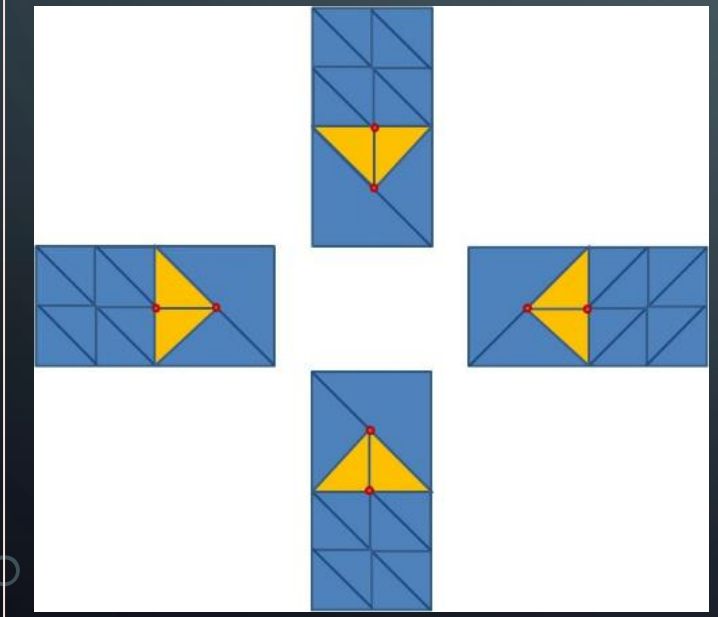
# 2. 균열방지




- **LOD**를 적용하면서 균열이 생긴걸 봤다.
- 결국은 **LOD** 단계가 다른 메시들이 만나는 경계부분에 때우기용
- 분할 삼각형을 추가해주면 된다.

**균열부위 땜질하기**

균열된 부분을 해결하려면 경계부분에 추가로 삼각형을 몇개 넣으면 되는데 다음과 같은 경우를 생각해보자

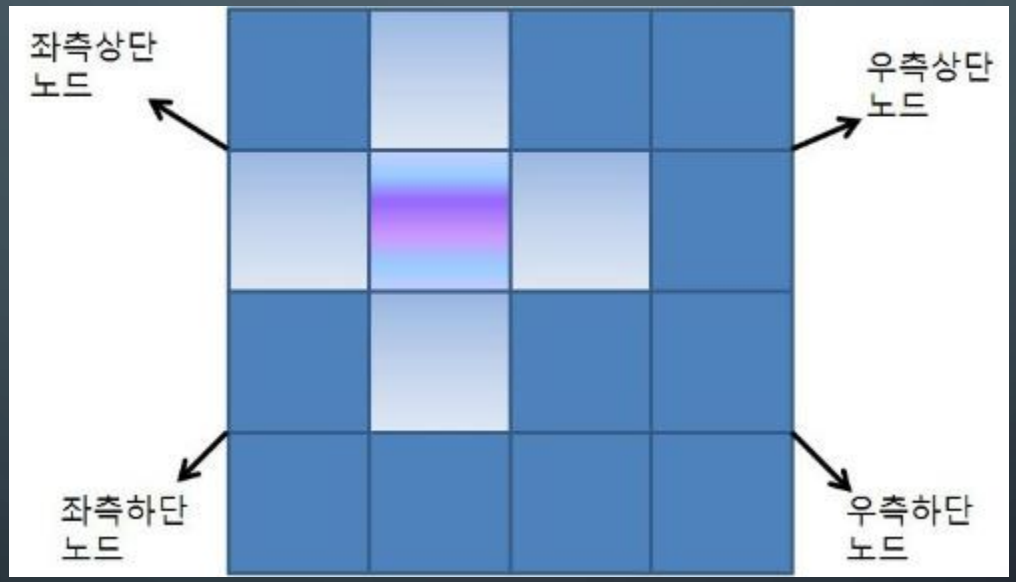
- 균열이 생길 수 있는 경계선 부분이 땜질 삼각형으로 덮이기 때문에 균열이 발생하지 않는다.
- 이런 땜질 삼각형을 LOD단계가 달라지는 모든 노드에 상하좌우 네 방향으로 만들어주면 된다.
- 균열부위를 땜질하려면 현재 자신의 노드 이외의 이웃한 노드들의 정보도 있어야 하는데, 그러기 위해서 4가지 함수가 추가되었다.

# 2. 균열방지

```
/// 이웃노드를 만든다.(삼각형 찢어짐 방지용) - 주변의 4개 노드값을 얻어 이웃노드를 만든다
void _BuildNeighborNode( ZQuadTree* pRoot, TERRAINVERTEX* pHeightMap, int cx )
{
	int n;
	int _0, _1, _2, _3;

	for( int i=0 ; i<4 ; i++ )
	{
		_0 = m_nCorner[0];
		_1 = m_nCorner[1];
		_2 = m_nCorner[2];
		_3 = m_nCorner[3];
		// 이웃노드의 4개 코너값을 얻는다.
		n = _GetNodeIndex( i, cx, _0, _1, _2, _3 );
		// 코너값으로 이웃노드의 포인터를 얻어온다.
		if( n >= 0 ) m_pNeighbor[i] = pRoot->_FindNode( pHeightMap, _0, _1, _2, _3 );
	}

	// 자식노드로 재귀호출
	if( m_pChild[0] )
	{
		m_pChild[0]->_BuildNeighborNode( pRoot, pHeightMap, cx );
		m_pChild[1]->_BuildNeighborNode( pRoot, pHeightMap, cx );
		m_pChild[2]->_BuildNeighborNode( pRoot, pHeightMap, cx );
		m_pChild[3]->_BuildNeighborNode( pRoot, pHeightMap, cx );
	}
}
```

# 2. 균열방지

- 출력할 삼각형의 인덱스를 만들면서 상하좌우 노드의 **LOD**값을 검색한 뒤**,** 현재 노드의 **LOD**값과 다른노드가 있는 경우 그 방향으로 땜질삼각형을 분할 생성해주면 된다**.**




**그림은 서로다른 자식노드에 이웃노드가 걸쳐있는 경우 그림기준으로 노드들은 상하좌우 2x2**