높이 맵(Height Field)

높이 값만 가지고 지형을 구성하는 것을 높이 맵(Height Field)라 합니다. 높이맵은 위에서 수직으로 보았을 때 x와 z 축에 대한 격자가 동일해서 높이에 대한 정보만 가지고 있으면 지형을 표현할 수 있기 때문에 높이 맵이라 부릅니다.

장점은 높이 값만 가지고 있으므로 데이터의 크기가 작고, 높이 값은 포토샵 같은 그래픽 툴로 흑백 이미지로 표현해서 만들 수 있습니다. 따라서 별도의 툴이 없어도 지형을 손쉽게 만들 수 있습니다. 또한 거대한 지형은 블록 단위로 그리면 같은 인덱스 버퍼를 사용할 수 있어서 메모리를 절약할 수 있습니다. 이외에 오브젝트가 지형 위에 있을 때 산술 연산만으로 높이를 구할 수 있으며, 멀티 텍스처를 활용한 타일링 등의 효과를 표현하기가 쉽습니다.

이렇게 좋은 높이 맵이 장점만 있는 것은 아닙니다. 가장 큰 단점은 지형의 기울기가 급해졌을 때 급한 지형일수록 메쉬를 더 많이 넣어 부드럽게 표현 해 주어야 하는데 높이맵은 이것이 불가능하며 지형의 기울기가 낮으면 메쉬가 많이 필요하지 않음에도 높이맵은 메쉬의 정점을 줄이지 못합니다. 이러한 단점을 극복하기 위해 쿼드 트리(Quad Tree), ROAM 등이 개발 되었는데 특히 ROAM은 실외 지형에 대해서 메쉬의 표현력이 가장 우수해서 비행 시뮬레이션과 같은 게임에서 지형을 표현에서 많이 권장이 됩니다.

메쉬에 대한 폴리곤이 고정되어 있어 높이 맵이 단점이 많아 보이기도 하지만 요즘 그래픽 카드들은 정점의 처리만큼은 이미 최적화 단계는 지났다고 합니다. 따라서 어느 정도 낭비가 있어도 정점 처리에 대한 속도는 그다지 크지 않은 편이고, 사용하는데 장점이 많이 있어서 개발 단계에서의 테스트 용이나 학생 작품뿐만 아니라 MMORPG의 대규모 실외지형에서 단연 으뜸으로 사용되고 있는 기술입니다.

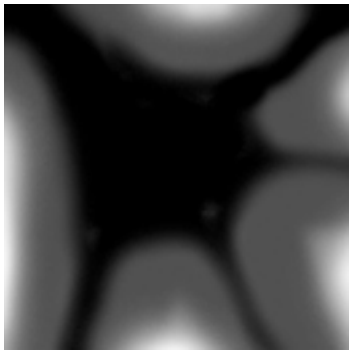
1. RAW 포맷

보통 게임 map은 형 툴이나 3d Max와 같은 3D Modeler 툴을 가지고 만들어야 하지만 높이맵은 x, z축 방향으로 간격이 일정하므로 각각의 정점의 높이를 이미지의 픽셀로 대응해서 구성할 수가 있습니다. 따라서 지형에 관련된 툴이 없어도 이미지로 작업하기 때문에 포토샵과 같은 그래픽 소프트웨어만 있어도 충분합니다.

높이맵으로 적합한 이미지 포맷은 먼저 압축이 없고, 파일의 헤더가 복잡하지 않고 흑백 픽셀을 지원하는 포맷이 좋습니다. 게임에서 사용되는 포맷 중에서 BMP, TGA 등이 좋은 포맷이 될 수 있고, 좀 더 간편한 것을 고려한다면 RAW 포맷을 생각할 수 있습니다. RAW는 파일의 헤더가 없고 1byte 색상을 저장하는 파일 포맷으로 파일의 크기가 곧 픽셀의 개수와 일치 합니다. 단순히 바이트 배열에 확장자 RAW(날 것)만 붙였다고 볼 수도 있습니다. RAW 포맷으로 저장하려면 포토샵을 사용할 경우 이미지의 높이와 너비는 같아야 하고 이미지 모드(image mode)는 흑백(gray scale)로 지정해야 바이트 단위로 저장됩니다.

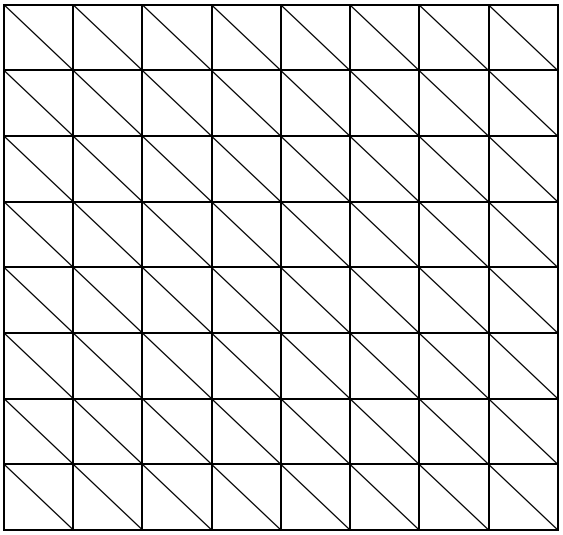
다음은 지형의 cell이 128 * 128 인 높이맵에 해당하는 높이 값을 흑백 이미지로 제작한 RAW 파일 예제 입니다. 주의할 것은 지형이 128 * 128 이면 이미지 파일의 너비와 높이는 129 * 129가 되어

야 합니다.

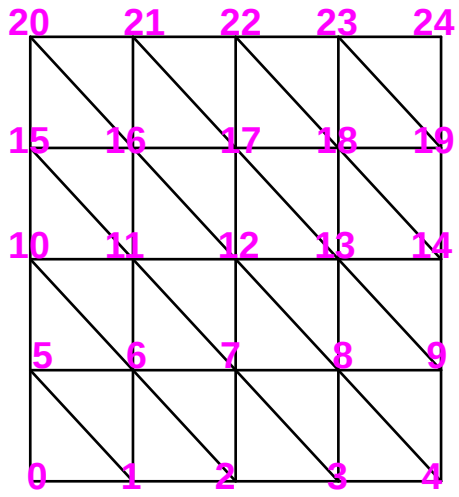


129 * 129 RAW 파일 Field_Height10.raw

우리는 칸(cell: 셀, 타일)의 수가 128 * 128인 지형을 제작할 것입니다. 따라서 정점은 (128+1) * (128+1)로 구성 되어 야 하고 RAW 파일의 이미지 크기도 129 * 129로 만들어야 합니다.
RAW은 이미 만들었다고 가정하고 이 RAW 파일의 픽셀을 가지고 높이 맵을 구성하기 위해서 먼저 다음과 같은 메쉬 모양을 프로그램 합니다.

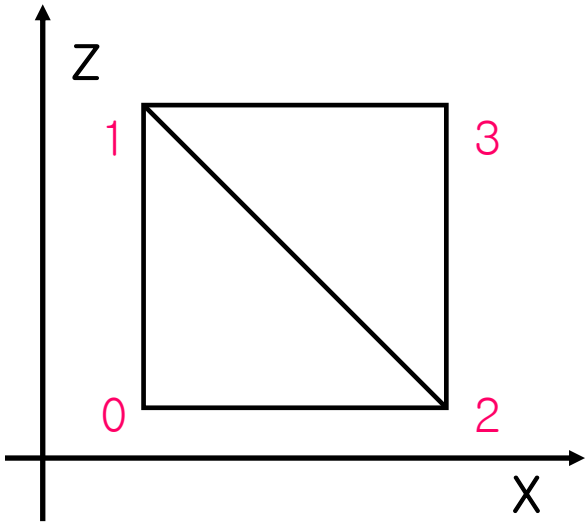


중복된 정점의 수를 줄이기 위해서 index 방식의 draw를 생각 하고 정점과 인덱스를 구성합니다.
여기서 메쉬는 사선이 왼쪽 상단에서 우측 하단으로 구성하도록 하는 것이 인덱스를 만들 때 유리 합니다.
정점과 인덱스를 구성하기 위해서 다음 그림과 같이 간략화 시켜 지형을 구성해보도록 하겠습니다.



앞의 그림에 따라 타일(셀)이 주어지면 삼각형 폴리곤의 숫자와 인덱스는 다음과 같이 됩니다.

폴리곤 = (타일 + 1) * (타일 + 1)
삼각형(a,b,c) = 타일 * 타일 * 2 (단 삼각형 하나당 a, b, c 로 구성 됨)

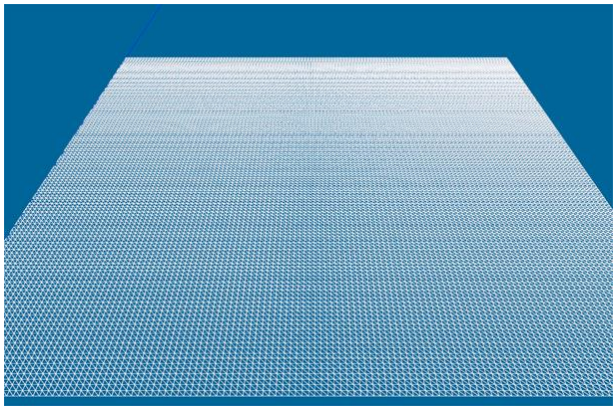


한 셀의 2개의 삼각형이 앞의 그림처럼 표현된다면 해당 인덱스는 다음과 같이 구성될 수 있습니다.

```
int _0 = (타일 +1)*(z+0) +x;
int _1 = (타일 +1)*(z+1) +x;
int _2 = (타일 +1)*(z+0) +x +1;
int _3 = (타일 +1)*(z+1) +x +1;
```

이에 따른 정점의 인덱스를 다음과 같이 간단하게 구성할 수 있습니다.

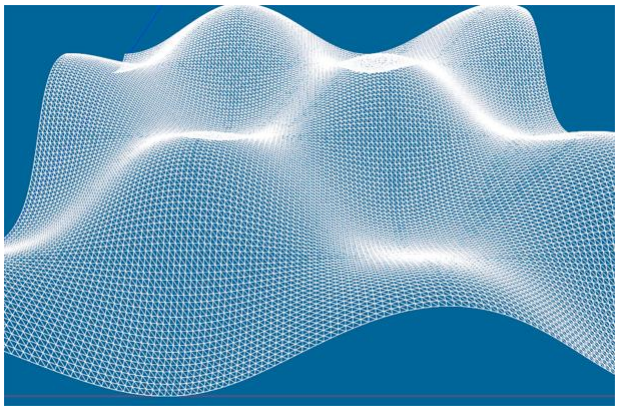
```
m_pFce[n] = VtxIdx(_0, _1, _2); ++n;
m_pFce[n] = VtxIdx(_3, _2, _1); ++n;
```



타일 크기 128*128: LcHg11_mesh.zip

높이 적용을 간단히 테스트할 수도 있습니다. 다음은 (cos 제곱 + 1)을 높이 값으로 사용한 예입니다.

```
...
for(z=0; z<=m_TileN; ++z)
{
	for(x=0;x<=m_TileN; ++x)
	{
		INT n = z * (m_TileN+1) + x;
		FLOAT _x = x * m_TileW;
		FLOAT _z = z * m_TileW;
		FLOAT _y = cosf(x/ FLOAT(m_TileN*0.1F) );
		_y = (_y*_y + 1.0F) * 40;
		m_pVtx[n].p = D3DXVECTOR3( _x, _y, _z);
	}
}
```



높이 값을 cos 제곱으로 설정 한 예: LcHg12_math.zip

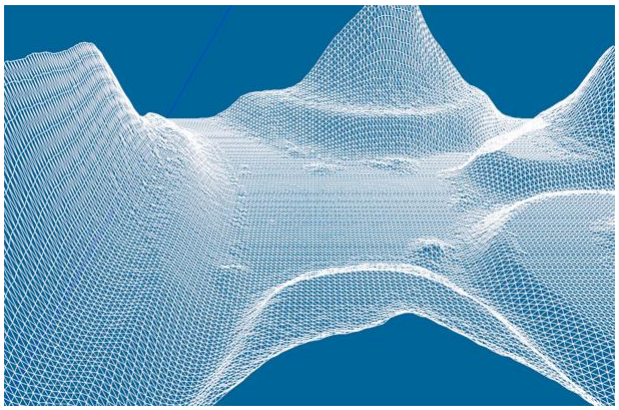
이제 마지막 단계로 RAW파일을 읽어와서 이를 높이 값(y)에 적용하는 일만 남았습니다. RAW 파일은 헤더가 없고, 픽셀을 높이와 1:1로 대응할 수 있기 때문에 높이 쓸 값을 바이트 단위로 파일의 크기만큼 읽어 옵니다.

```
FILE* fp = fopen(sRaw, "rb");
fseek(fp, 0, SEEK_END);
long lSize = ftell(fp);

fseek(fp, 0, SEEK_SET);
BYTE* pH = new BYTE[lSize];

fread(pH, lSize, 1, fp);
fclose(fp);
//높이 값 채우기
delete [] pH;
```

이렇게 읽어온 값을 높이 값으로 적용해야 하는데 주의할 점은 그림 파일은 좌상 단이 시작 이지만 우리가 만든 지형은 좌 하단이 시작입니다. 따라서 파일의 데이터를 그대로 적용하면 지형이 z 방향에 대해서 뒤집힌 형태로 표현이 됩니다. 이 부분은 공식이 필요한데 다음과 같은 것을 이용하면 됩니다.

```
for(z=0; z<=m_TileN; ++z)
{
        for(x=0; x<=m_TileN; ++x)
        {
                INT n1 = z * (m_TileN+1) + x;                    // 타일 인덱스
```

```
                INT n2 = (m_TileN - z) * (m_TileN+1) + x;       // 픽셀 인덱스

                FLOAT _x = x * m_TileW;
                FLOAT _z = z * m_TileW;
                FLOAT _y = pH[n2] * scaling;            // 높이 = 픽셀 * scaling

                m_pVtx[n1].p = D3DXVECTOR3( _x, _y, _z);
        }
}
```



RAW 파일의 픽셀을 높이 값으로 설정한 예: LcHg13_raw.zip

2. 정점 Diffuse, Detail Map, Diffuse Map

정점에 Diffuse 값과 텍스처를 적용하기 위해서 먼저 정점 데이터에 색상 값을 적용할 수 있도록 다음과 같이 구조체와 FVF 값을 바꾸어야 합니다.

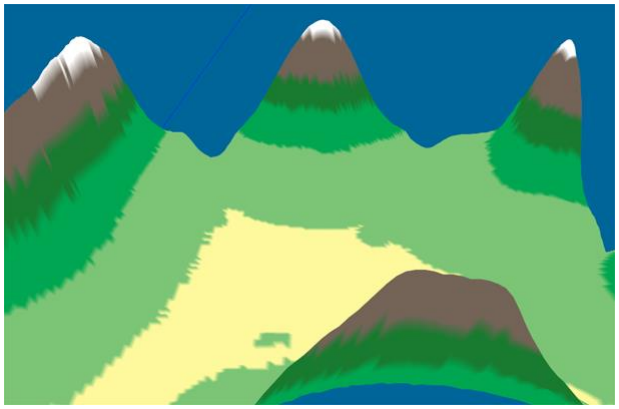
```
struct VtxD
{
        D3DXVECTOR3     p;      // position
        DWORD           d;      // diffuse color
...
        enum    {FVF = (D3DFVF_XYZ|D3DFVF_DIFFUSE),};   // FVF
};
```

만약 지형의 양면을 표현하려면 렌더링 상태 값 culling mode을 양면을 그릴 수 있도록 None로 설정합니다.

```
m_pDev->SetRenderState( D3DRS_CULLMODE,  D3DCULL_NONE); // cull mode  지정 안함
```

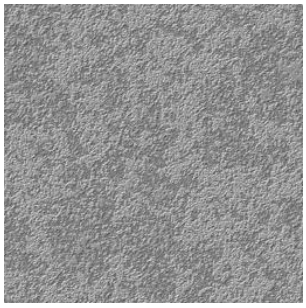
높이에 따라 정점의 Diffuse 색상을 적용해 보겠습니다. 다음은 모래, 풀밭, 숲, 암석, 눈의 색상을 적당히 선택해서 Diffuse 값을 적용한 예입니다.

```
FLOAT h = pH[n]*m_fHscl;
...
DWORD d = D3DCOLOR_XRGB(255, 255, 255);
if(h < 1.0f)            d = D3DCOLOR_XRGB(255, 249, 157);
else if(h < 45.0f)      d = D3DCOLOR_XRGB(124, 197, 118);
else if(h < 85.5f)      d = D3DCOLOR_XRGB(  0, 166,  81);
else if(h < 120.0f)     d = D3DCOLOR_XRGB( 25, 123,  48);
else if(h < 170.5f)     d = D3DCOLOR_XRGB(115, 100,  87);

m_pVtx[n].d = d;
```



LcHg21_diffuse.zip

디테일 맵(Detail map)의 크기는 작고 처리방법 또한 간편해서 3D 게임 초창기에 비디오 메모리가 크지 않았을 때 디테일 맵은 단조로운 지형의 색상에 효과적이었습니다. 현재도 시스템 메모리가 PC 보다 작은 일부 Embedded 기기에서도 일부 적용되기도 합니다.



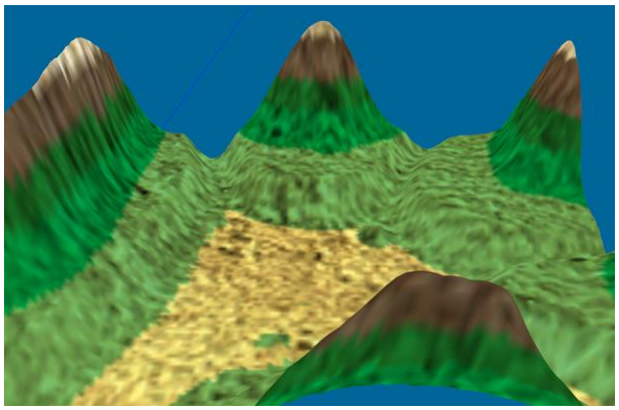
<Detail map>

디테일 맵은 정점에 텍스처를 입히는 것과 같기 때문에 텍스처의 UV 좌표가 설정될 수 있도록 다음과 같이 정점 구조체와 FVF를 구성합니다.

```
struct VtxDUV1
{
        D3DXVECTOR3     p;      // position
        DWORD           d;      // diffuse
        FLOAT           u,v;    // texture u,v
...
        enum    {FVF = (D3DFVF_XYZ|D3DFVF_DIFFUSE|D3DFVF_TEX1),};
};
```

지형 전체에 [0,1] 범위로 텍스처를 적용하려면 정점의 U,V 는 타일의 숫자로 나누면 됩니다. UV 설정에서 주의해야 할 것은 3D 지형에 적용되는 텍스처들은 대부분 z축으로 증가하면 UV좌표의 v 값을 감소시키기 위해서 RAW파일을 지형의 높이 적용에서 그림의 위아래 데이터를 바꾸어 주었듯이 UV의 v 값도 위 아래를 바꾸어 주어야 합니다. 바꾸는 방법은 1.0 - v 로 설정합니다.

```
m_pVtx[n].u = FLOAT(x)/m_TileN;
m_pVtx[n].v = 1.f - FLOAT(z)/m_TileN;
```



LcHg22_texture.zip

디테일 맵의 목적은 noise 등을 이용해서 질감의 자연스러움을 표현하는 것인데 UV의 [0,1] 범위 값은 적당하지 않음을 바로 알 수 있습니다. 계산된 UV의 [0,1] 범위 값은 이후에도 이용될 수 있으므로 디테일 맵의 반복 적용에 대한 적당한 셀 범위에 대한 변수 하나를 추가해서 UV 에 곱해서 사용하도록 하고, 여기서는 16.0f 정도의 값을 사용하겠습니다.

```
m_fUV   = 16.f;
m_pVtx[n].u = FLOAT(x)/m_TileN;
m_pVtx[n].v = 1.f - FLOAT(z)/m_TileN;
m_pVtx[n].u *= m_fUV;                    // 디테일 텍스처의 u 값을 증가시킴
m_pVtx[n].v *= m_fUV;                    // 디테일 텍스처의 v 값을 증가시킴
```

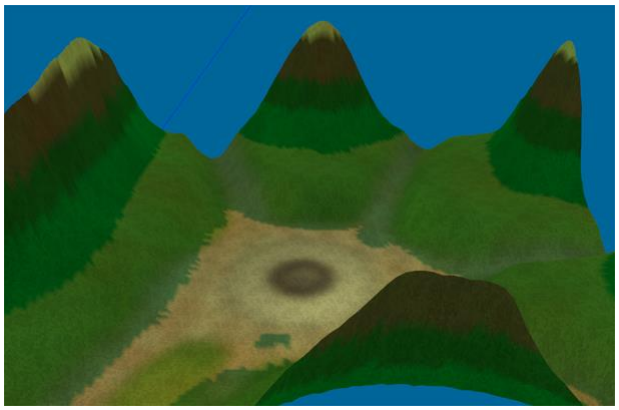
LcHg23_detail.zip

정점의 색상, 디퓨즈 맵, 디테일 맵을 적용하는 Multi-texturing을 위해서 2개의 텍스처 좌표가 사용할 수 있도록 정점 구조체와 FVF를 변경합니다. 여기서 정점 데이터의 u0, v0는 디퓨즈 맵의 좌표로 사용하고 u1, v1은 디테일 맵의 좌표로 사용하겠습니다.

```
struct VtxDUV2
{
        D3DXVECTOR3     p;      // position
        DWORD           d;      // diffuse
        FLOAT           u0, v0; // diffuse map uv
        FLOAT           u1, v1; // detail map uv
...
        enum    {FVF = (D3DFVF_XYZ|D3DFVF_DIFFUSE|D3DFVF_TEX2),};
};
```

디퓨즈 맵과 디테일 맵의 혼합은 곱셈 연산으로 처리합니다. 고정 기능 파이프라인에서 곱셈 연산에 대한 OP는 MODULATE, MODULATE2X, MODULATE4X 3종류가 있습니다. 곱셈 연산을 계속하면 색상은 점점 어두워지므로 먼저 MODULATE로 처리해 보고 어둡다고 생각되면 MODULATE2X나 MODULATE4X를 적용해서 색상의 밝기를 올리고 이중에서 가장 알맞은 OP를 선택합니다.

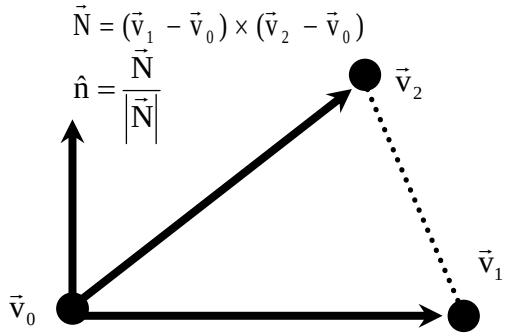
```
// 0-stage color operation
m_pDev->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
m_pDev->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
m_pDev->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE);
// 0-stage color operation
m_pDev->SetTextureStageState(0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE);
m_pDev->SetTextureStageState(0, D3DTSS_ALPHAARG2, D3DTA_DIFFUSE);
m_pDev->SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_MODULATE);

// 1-stage alpha operation
m_pDev->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_CURRENT);
m_pDev->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_TEXTURE);
m_pDev->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE2X);
// 1-stage alpha operation
m_pDev->SetTextureStageState(1, D3DTSS_ALPHAARG1, D3DTA_CURRENT);
m_pDev->SetTextureStageState(1, D3DTSS_ALPHAARG2, D3DTA_TEXTURE);
m_pDev->SetTextureStageState(1, D3DTSS_ALPHAOP, D3DTOP_MODULATE);
```



LcHg24_mutitexture.zip

3. 정점의 Normal Vector와 Lighting Map

지형의 입체감(volume)을 살리기 위해서 라이팅 효과가 필요하고 라이팅 효과를 적용하려면 정점의 법선 벡터(Normal Vector)가 필요합니다.
3 점만 있으면 외적과 정규화를 통해서 3점이 포함된 평면의 법선 벡터를 구할 수 있습니다.
을 사용해서 다음과 같이 구현 할 수 있습니다.



이것을 함수로 구현 하면 다음과 같습니다.

```
void CalculateNormal(D3DXVECTOR3* pOut
        , D3DXVECTOR3* v0, D3DXVECTOR3* v1, D3DXVECTOR3* v2)
...
D3DXVECTOR3 n;
D3DXVECTOR3 A = *v2 - *v0;
D3DXVECTOR3 B = *v1 - *v0;
D3DXVec3Cross(&n, &A, &B);
```

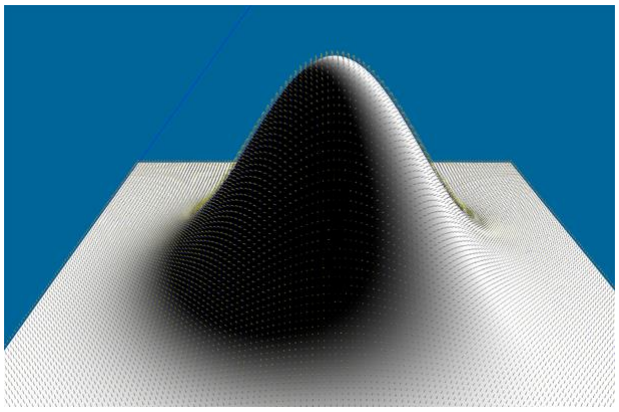
```
D3DXVec3Normalize(&n, &n);
*pOut = n;
...
```

높이 맵 정점의 법선 벡터는 정정에 인접한 삼각형의 법선 벡터의 평균 값을 사용합니다. 지형의 경계에 있는 정점들을 제외한 나머지 정점 들은 각각 6개의 삼각형에 인접해 있습니다. 따라서 6개 삼각형의 법선 벡터를 찾아 이들을 더한 평균 값을 정점의 법선 벡터로 설정해야 합니다.

```
void CMcField::SetupNormal()
...
for(z=1; z<m_TileN; ++z)
{
        for(x=1; x<m_TileN; ++x)
        {
                n = z * nVtxT + x;

                // 인접한 삼각형의 인덱스
                INT nIdx[6][3]=
                {
                        { n, n-1       , n-nVtxT   },
                        { n, n-nVtxT   , n-nVtxT+1 },
                        { n, n-nVtxT+1 , n+1       },
                        { n, n+1       , n+nVtxT   },
                        { n, n+nVtxT   , n+nVtxT-1 },
                        { n, n+nVtxT-1 , n-1       },
                };

                Normal = D3DXVECTOR3(0,0,0);
                // 법선 벡터 누적
                for(k=0; k<6; ++k)
                {
                        v0 = m_pVtx[  nIdx[k][0] ].p;
                        v1 = m_pVtx[  nIdx[k][1] ].p;
                        v2 = m_pVtx[  nIdx[k][2] ].p;
                        CalculateNormal(&Nor, &v0, &v1, &v2);
                        Normal +=Nor;
                }
```

```
                // 법선 벡터 정규화
                D3DXVec3Normalize(&Normal,&Normal);
                m_pVtx[n].n = Normal;
...
```

다음으로 인접한 삼각형의 수가 3개, 2개 또는 1개인 경계에 있는 정점들의 법선 벡터를 계산해서 SetupNormal() 함수를 완성합니다.

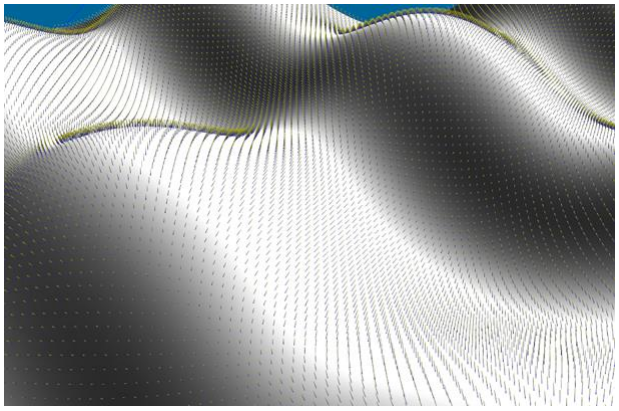


LcHg31_normalvector.zip

이렇게 정점의 법선 벡터를 계산해도 되지만 지형이 급격하게 변하지 않고 완만하게 변한다고 가정한다면 굳이 인접한 모든 삼각형의 법선 벡터를 찾아서 계산하지 않고 주변의 4점이나 아니면 증가하는 방향으로 인접한 삼각형의 법선 벡터를 그대로 사용해도 큰 차이는 없습니다.

```
void CMcField::SetupNormal()
...
D3DXVECTOR3     Normal(0,0,0);
D3DXVECTOR3     v0, v1, v2;

for(z=0; z<m_TileN; ++z)
{
        for(x=0; x<m_TileN; ++x)
        {
                n = z * (m_TileN+1) + x;

                v0 = m_pVtx[  n             ].p;
                v1 = m_pVtx[  n+1           ].p;
                v2 = m_pVtx[  n + m_TileN+1 ].p;
```

```
                CalculateNormal(&Normal, &v0, &v1, &v2);
                D3DXVec3Normalize(&Normal,&Normal);
                m_pVtx[n].n = Normal;
        }
}

// 맨 오른쪽: copy z-1 to z
for(z=0; z<m_TileN; ++z)
{
        n = z * (m_TileN+1) + m_TileN;
        m_pVtx[n].n = m_pVtx[n-1].n;
}

// 맨 위쪽: copy x-1 to x
for(x=0; x<=m_TileN; ++x)
{
        n = m_TileN * (m_TileN+1) + x;
        m_pVtx[n].n = m_pVtx[n-(m_TileN+1)].n;
}
...
```



LcHg31_normalvector2.zip

많은 실외 지형 툴들은 그 결과물을 대부분 이미지로 저장합니다. 그러한 이유는 프로그램 실행 중에 연산하는 것보다 메모리는 더 사용하지만 미리 계산된 값을 메모리에 올려서 사용하는 것이 렌더링 속도에 이득이 되기 때문입니다.
라이팅 맵은 미리 광원 효과를 계산해 놓은 텍스처를 적용하는 것으로 퀘이크3 게임에서 최초로 사용했고, 지금도 게임의 사실감을 높이는데 아주 유용하게 사용되는 기술입니다. 라이팅 맵 사용

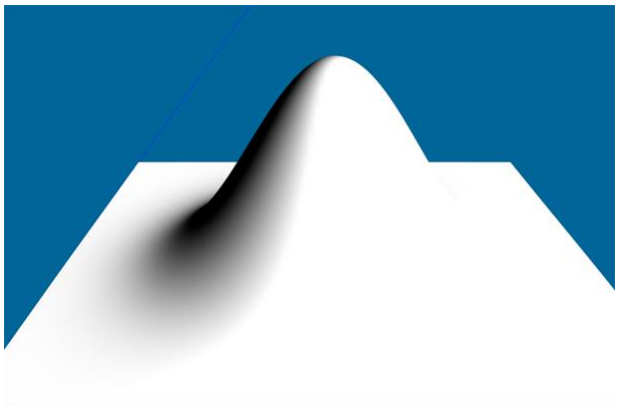
은 게임 프로그램의 메모리 사용을 증가시키지만 실시간 라이팅과 달리 Multi-texturing에서 처리되기 때문에 정점 처리에서 보다 부드러운 조명 효과를 만들어 냅니다. 특히, Embedded 기기와 같은 환경에서 실시간 조명보다 훌륭한 연출을 만들어 줍니다.

라이팅 맵을 만들기 위해서 먼저 프로그램 실행 중에 텍스처를 생성해야 합니다. 실시간 텍스처 생성은 D3DXCreateTexture 함수를 사용하거나 D3D Device 객체의 멤버 함수인 CreateTexture 함수를 사용합니다. 텍스처 생성해서 주의할 점은 DirectX는 텍스처의 서피스를 2 승수로 만든다는 점입니다. 앞의 예제가 129* 129의 폴리곤을 사용하는 지형이므로 텍스처는 256*256으로 설정해야 하겠지만 지형의 변화가 급격히 변하지 않는다고 가정하고 또한 타일의 숫자가 2의 승수로 맞추었다면 타일 * 타일 사이즈의 텍스처로 설정해도 됩니다.

램버트 확산은 정점에서 광원의 방향 벡터와 법선 벡터의 내적을 반사의 크기로 설정합니다. 앞의 예제를 128* 128 크기에 램버트 확산을 적용한 라이팅 맵을 만들어 봅시다.

```
void CMcField::ExportLightingMap()
...
// 테스트 평행광의 방향을 임의로 설정, 반대 방향 벡터를 구성
D3DXVECTOR3 vcLght = D3DXVECTOR3(-1,-1,-1);     // test light direction
D3DXVECTOR3 vcL = -vcLght;                      // inversion the light direction

for(k=0; k<m_nVtx; ++k)
{
        D3DXVECTOR3 vcT = m_pVtx[k].n;          // 법선 벡터
        FLOAT fLight = D3DXVec3Dot(&vcT, &vcL); // 반사의 크기= dot(light, normal)

        if(fLight<0.f)
                fLight = 0.f;

        // 반사의 크기를 diffuse에 저장
        m_pVtx[k].d = D3DXCOLOR(fLight, fLight, fLight, 1);
}

// 라이팅 맵용 텍스처 생성
LPDIRECT3DTEXTURE9 pTxLgt;
D3DXCreateTexture(m_pDev, m_TileN, m_TileN
                , 0, 0, D3DFMT_X8R8G8B8, D3DPOOL_MANAGED, &pTxLgt);
```

```
// 텍스처의 서피스에서 색상 버퍼 얻어오기
D3DSURFACE_DESC Dsc;
LPDIRECT3DSURFACE9 pSf=NULL;
D3DLOCKED_RECT lockedRect;
pTxLgt->GetLevelDesc(0, &Dsc);
pTxLgt->GetSurfaceLevel(0, &pSf);
pSf->LockRect(&lockedRect, 0, 0);

// 색상 버퍼를 전부 0으로 초기화
DWORD* pImg = (DWORD*)lockedRect.pBits;
memset(pImg, 0, sizeof(DWORD) * Dsc.Width * Dsc.Height);

for(z=0; z<m_TileN; ++z)
{
	for(x=0; x<m_TileN; ++x)
	{
		INT	s = z * Dsc.Width + x;
		INT	n = z * nVtxT + x;
		DWORD	c = m_pVtx[n].d;

		// 픽셀에 색상 저장
		pImg[s] = c;
	}
}

pSf->UnlockRect();

// save image to bmp
D3DXSaveSurfaceToFile("data/Map_Lgt.bmp", D3DXIFF_BMP, pSf, NULL, NULL);

// 서피스 해제
pSf->Release(); pTxLgt->Release();
```



LcHg32_lightingmap.zip

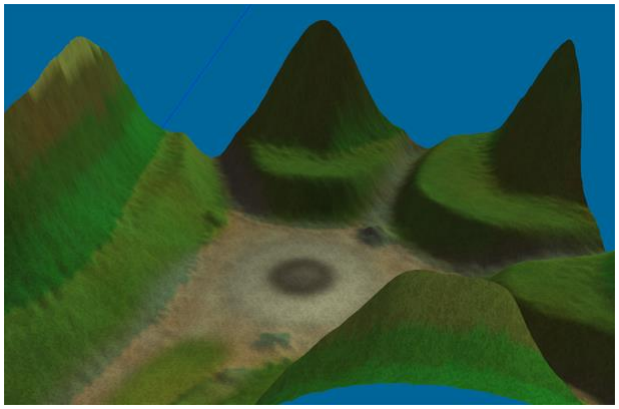
지금까지 설명한 높이 맵을 정리하는 차원에서 라이팅 맵은 사용하지 않고 실시간 라이팅과 디퓨즈 맵, 디테일 맵, 그리고 정점의 Diffuse 값을 혼합하는 지형을 표현해 봅시다. 먼저 정점 구조체와 FVF를 다음과 같이 구성합니다.

```
struct VtxNDUV2
{
        D3DXVECTOR3     p;              // position
        D3DXVECTOR3     n;              // normal
        DWORD           d;              // diffuse
        FLOAT           u0,v0;          // difuse map coordinate
        FLOAT           u1,v1;          // detail map coordinate
...
        enum    {FVF = (D3DFVF_XYZ|D3DFVF_NORMAL|D3DFVF_DIFFUSE|D3DFVF_TEX2),};
};
```

고정 파이프라인의 라이팅을 이용하기 위해서 라이팅과 재질을 설정합니다.

```
HRESULT CMain::Render()
...
// 광원과 광원의 방향 벡터 설정
D3DLIGHT9 d3Lght;
D3DXVECTOR3 vcLght( 1, 1, 1);

vcLght = -vcLght;
D3DXVec3Normalize(&vcLght, &vcLght);
memset( &d3Lght, 0, sizeof d3Lght);
```

```
d3Lght.Type = D3DLIGHT_DIRECTIONAL;
...
// 재질 설정
D3DMATERIAL9    d3Mtl;
...
// 디바이스에 재질, 광원, Ambient 설정
m_pd3dDevice->SetMaterial(&d3Mtl);
m_pd3dDevice->SetLight( 0, &d3Lght );
m_pd3dDevice->LightEnable( 0, TRUE );
m_pd3dDevice->SetRenderState( D3DRS_LIGHTING,TRUE);
m_pd3dDevice->SetRenderState( D3DRS_AMBIENT, 0xFF555555);
// 지형을그린다.
m_pField->Render();
```



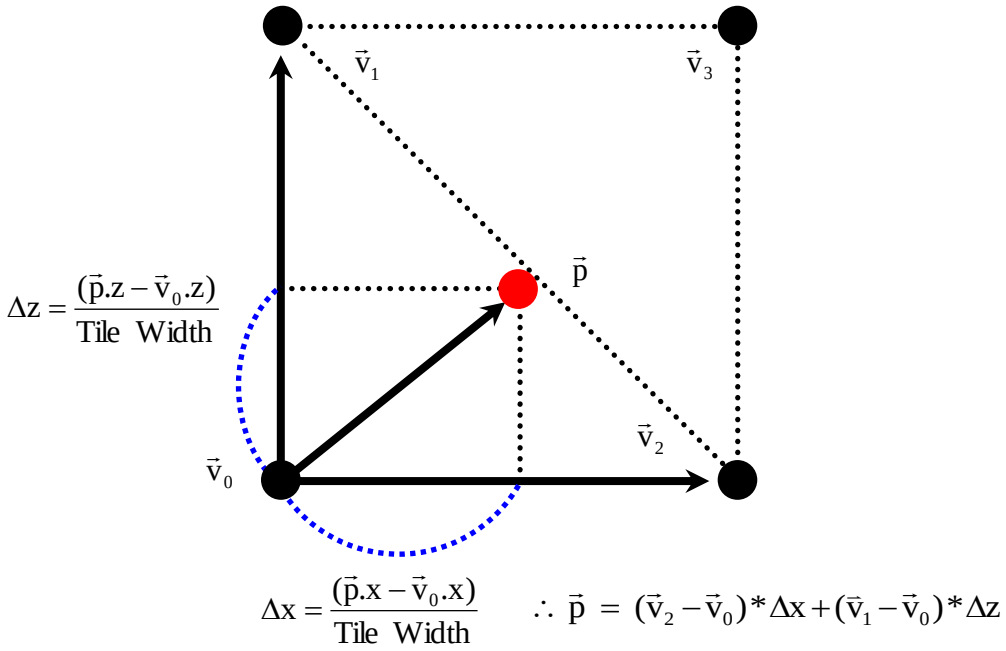
LcHg33_diffuse+normal.zip

4. 높이 맵 응용

4.1 높이 맵에서 높이 값

높이 맵의 장점 중에 하나가 타일 간격이 일정하기 때문에 지형 위에 있는 오브젝트의 y 값을 오브젝트와 타일의 위치로 간단한 산술 연산으로 빠르게 구할 수 있습니다.

다음 그림처럼 타일과 과 타일 위에 있는 오브젝트를 y축에서 내려다 본 방향에서 기하학적으로 분석해 보면 타일의 시작 점(v0)에서의 상대적인 거리 delta x, delta z와 v1, v2를 사용해서 v0에서 시작해서 오브젝트 위치까지의 벡터를 계산할 수 있습니다. 결국 최종 오브젝트의 위치는 이 벡터와 타일의 시작점 v0 를 합산한 결과로 계산이 됩니다.



타일의 정점 위치로 오브젝트의 위치를 계산하기 위해서 먼저 시작점 v1, v2를 구합니다. 높이 맵은 타일의 간격이 일정하기 때문에 오브젝트의 위치가 주어지면 이 위치를 타일의 폭으로 나눈 값을 int 형으로 변환하면 타일의 x와 z축 방향의 인덱스와 일치 합니다. 이 인덱스를 사용하면 시작점을 배열에서 바로 얻어 낼 수 있으며 v1, v2도 쉽게 얻게 됩니다.

<v0, v1, v2 의사 코드(Pseudo-code)>

```
x축 인덱스 = int(오브젝트 x / 타일 폭)
z축 인덱스 = int(오브젝트 z / 타일 폭)
시작점 인덱스 = z축 인덱스 * (타일 크기 + 1) + x축 인덱스
v0 = 버텍스 버퍼[시작점 인덱스]
v1 = 버텍스 버퍼[시작점 인덱스 + (타일 크기 + 1)]
v2 = 버텍스 버퍼[시작점 인덱스 + 1]
```

delta 값들을 구하고 최종 위치를 계산합니다.

<delta x, delta z, 최종 위치 의사 코드(Pseudo-code)>

```
delta x = (오브젝트 x - 시작점 x) / 타일 폭
delta z = (오브젝트 z - 시작점 z) / 타일 폭
오브젝트 위치 = v0 + (v2 - v0) * delta x + (v1 - v0) * delta z
```

이렇게 오브젝트의 위치를 정해도 게임 플레이에서 거의 문제가 안됩니다. 만약 좀 더 정교하게 작성 하려면 타일 안의 2개의 삼각형에서 오브젝트가 포함된 삼각형에서 위치를 구해야 됩니다.

2개의 삼각형 중에서 포함된 삼각형을 찾는 것은 delta x + delta z 값이 1보다 작으면 아래 삼각형을, 그렇지 않으면 위쪽 삼각형을 선택합니다.
만약 위쪽의 삼각형을 선택했을 때 시작점은 v3로 설정되어야 하고 delta x, delta z는 다음과 같이 수정되어야 합니다.

< (delta x + delta z)>=1 일 때 오브젝트 위치 의사 코드>

v3(시작점) = 버텍스 버퍼[시작점 인덱스 + 1 + (타일 크기 + 1)]

delta x = 1 - delta x

delta z = 1 - delta z

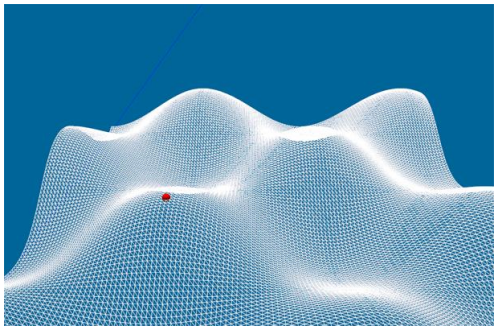
오브젝트 위치 = v3 + (v1 - v3) * delta x + (v1 - v3) * delta z

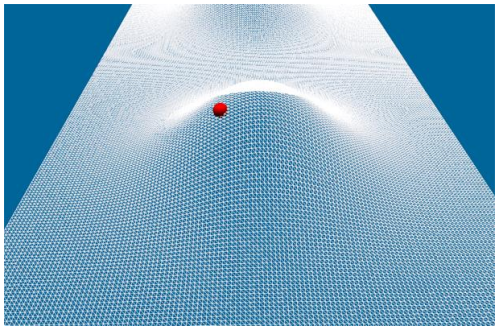
이것을 코드로 구현하면 다음과 같습니다.

```
INT CMcField::GetHeight(D3DXVECTOR3* pOut, const D3DXVECTOR3* pIn)
...
INT nX = INT( pIn->x/ m_TileW );
INT nZ = INT( pIn->z/ m_TileW );

// 실패: 오브젝트가 높이 맵 위에 없음
if(nX<0|| nX>=m_TileN || nZ<0|| nZ>=m_TileN)
        return -1;

//      1 ----- 3
//      . \     |
//      .    \  |
//      0 ----- 2

INT _0 = (nZ+0)*(m_TileN+1) + nX+0;
INT _1 = (nZ+1)*(m_TileN+1) + nX+0;
INT _2 = (nZ+0)*(m_TileN+1) + nX+1;
INT _3 = (nZ+1)*(m_TileN+1) + nX+1;

D3DXVECTOR3 v0 = m_pVtx[_0].p;
D3DXVECTOR3 v1 = m_pVtx[_1].p;
D3DXVECTOR3 v2 = m_pVtx[_2].p;
D3DXVECTOR3 v3 = m_pVtx[_3].p;
```

```
FLOAT   deltaX = (pIn->x - v0.x)/m_TileW;
FLOAT   deltaZ = (pIn->z - v0.z)/m_TileW;

D3DXVECTOR3 vcOut;

// 아래쪽 삼각형
if( (deltaX+deltaZ) <=1)
        vcOut = v0 + (v2- v0) * deltaX + (v1- v0) * deltaZ;

// 위쪽 삼각형
else
{
        deltaX = 1 - deltaX;
        deltaZ = 1 - deltaZ;
        vcOut = v3 + (v2- v3) * deltaX + (v1- v3) * deltaZ;
}

*pOut = vcOut;
...
```

다음의 예제는 키보드의 left, right, up, down 키를 이용해 오브젝트를 이동할 수 있으며 지형의 위에 오브젝트가 있을 때 이 오브젝트의 y 값을 결정하는 예제입니다.





LcHg41_getheight1.zip LcHg41_getheight2.zip

앞의 소스를 사용해서 높이 맵 위에 DirectX x 파일의 나무들을 심어 보겠습니다.

LcHg42_height+object.zip

연습 삼아 빌보드로 구성된 나무를 출력해 보겠습니다. 나무의 위에는 위치를 표시해 보겠습니다. 참고로 오브젝트의 위치에 글자를 출력 하려면 3차원 좌표를 2차원 화면 좌표로 바꾸어야 합니다. 2차원 화면 좌표로 바꾸는 방법은 그래픽 파이프라인의 변환 방식을 그대로 적용하면 됩니다.
그래픽 파이프라인에서 정점의 위치는 월드, 뷰, 투영, 뷰포트 변환을 거치므로 이들 행렬의 곱셈 결과를 오브젝트의 위치에 적용하면 2차원 화면 좌표가 됩니다.

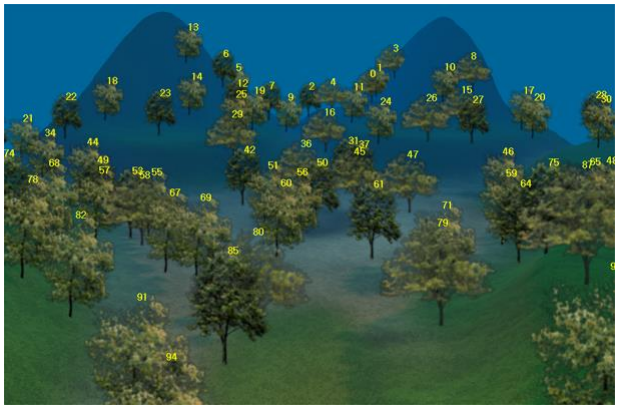
2차원 화면 좌표 = (3차원 좌표) * (월드 행렬 * 뷰 행렬 * 투영 행렬 * 뷰포트 행렬)

뷰포트 변환 행렬은 Direct3D의 경우 지원이 안되므로 다음과 같이 직접 계산 합니다.

```
void LcxMatrixViewport(D3DXMATRIX* pOut, const D3DVIEWPORT9* pV /*Viewport*/)
...
float   fW = 0, fH = 0, fD = 0, fY = 0, fX = 0;
float   fM = FLOAT(pV->MinZ);

fW = FLOAT(pV->Width)*.5f;
fH = FLOAT(pV->Height)*.5f;
fD = FLOAT(pV->MaxZ) - FLOAT(pV->MinZ);
fX = FLOAT(pV->X) + fW;
fY = FLOAT(pV->Y) + fH;

*pOut = D3DXMATRIX( fW,  0.f,  0, 0,
                    0.f,  -fH,  0, 0,
                    0.f,  0.f, fD, 0,
                    fX,    fY, fM, 1);
...
```

이렇게 얻은 뷰포트 행렬을 가지고 월드 행렬 * 뷰 행렬 * 투영 행렬 * 뷰포트 행렬의 곱셈을 계산해서 얻은 행렬을 오브젝트의 위치에 적용하면 2차원 좌표 변환은 완성이 됩니다.

```
D3DXMATRIX      mtView;         // 뷰행렬
D3DXMATRIX      mtProj;         // 투영행렬
D3DXMATRIX      mtVp;           // 뷰포트행렬

m_pDev->GetTransform(D3DTS_VIEW, &mtView);
m_pDev->GetTransform(D3DTS_PROJECTION, &mtProj);
m_pDev->GetViewport(&vp);
…
LcxMatrixViewport(&mtVp, &vp);// 뷰포트행렬계산

// 변환행렬= 월드행렬* 뷰행렬* 투영행렬* 뷰포트행렬
D3DXMATRIX  mtTMpt  = mtView * mtProj * mtVp;

// 오브젝트의 좌표 변환
D3DXVec3TransformCoord(&vcOut, &vcIn, &mtTMpt);
…
pFont->DrawText(…, outputString, …);
```



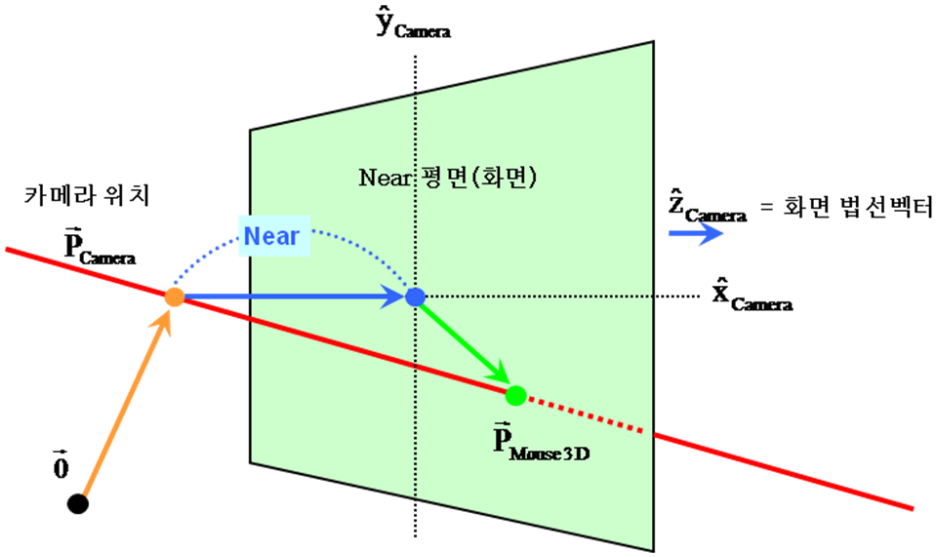
LcHg43_3d_to_2d.zip

4.2 Picking

3차원에서 피킹(Picking)은 2차원 마우스의 위치를 3차원으로 환원해서 이 3차원 좌표와 카메라의 위치를 지나는 직선을 충돌 직선으로 설정하고 이 직선에 충돌하는 오브젝트를 찾아내는 작업입니다.

물론 3차원 오브젝트 좌표를 2차원으로 변환해서 마우스 좌표와 충돌 검사를 해도 되는데 이 방법은 여기서 다루지 않겠습니다.

피킹을 위해서 먼저 마우스의 위치를 3차원으로 바꾸어야 합니다. 방법은 뷰 체적(Viewing Volume) 안에 있는 정점은 투영 변환 후에 [-1, 1] 범위 안의 값으로 됩니다. 화면 좌표의 마우스 위치는 3차원 공간의 점이 화면 좌표로 변환 되었다고 생각한다면 그림처럼 3차원으로 환원할 마우스의 위치를 near 평면에 있는 한 점과 카메라의 위치로 구성된 직선의 임의의 점으로 생각할 수 있습니다.



< near 평면 위에서 마우스의 3차원 위치>

그림에서 near 평면 위의 점은 다음 식으로 간단히 계산이 됩니다.

$$\vec{P}_{Mouse3D} = \vec{P}_{Camera} + Near * \hat{z}_{Camera} + \vec{P}_{UnTMmouse}$$

여기서 $\vec{P}_{UnTMmouse}$은 정규 변환 단계로 환원한 마우스 좌표 입니다.

이 수식을 단계적으로 완성해 봅시다. 먼저 P(UnTMmouse)를 구할 것인데 정규 변환(투영 변환) 후에 좌표는 [-1,1] 이고 이 값이 뷰포트 변환(Viewport transform) 후에 [화면 좌상단, 화면 우하단] 값으로 변환이 되므로 화면의 마우스 위치를 가지고 원래의 정규 변환 값을 구하는 공식을 완성할 수 있습니다.

정규 변환 x = 2 * 마우스 x / 화면 너비 - 1
정규 변환 y = 1 - 2 * 마우스 x / 화면 높이
카메라 x 방향 = (정규 변환 x * 카메라 x축 방향) * near
카메라 y 방향 = (정규 변환 y * 카메라 y축 방향) * near
카메라 z 방향 = (카메라 z축 방향) * near

최종 위치 = 카메라 위치 + 카메라 x 방향 + 카메라 y 방향 + 카메라 z 방향

이 방법에서 near를 사용하지 않고 1 로 설정해도 됩니다. 또한 이 방법은 역 변환에 대한 수식을 풀어 쓴 것과 동일 합니다.

3차원 화면 좌표 = (2차원 좌표) * (뷰포트 역 행렬 * 투영 역 행렬 * 뷰 역 행렬)

물론 연산의 속도를 위해서 (뷰 행렬 * 투영 행렬 * 뷰포트 행렬)의 역 행렬을 적용해도 됩니다.
이 두 가지 방법은 Direct3D Extended 함수 중에 D3DXVec3Unproject() 함수를 사용해서 간단히 구할 수 있습니다.
재미 삼아 이 두 가지 방법과 D3DX 함수를 구현해서 비교해 보도록 하겠습니다.

첫 번째 방법을 코드로 구현하면 다음과 같습니다.

```
D3DXVECTOR3 GetMouse3DPosition(…)
…
// 뷰포트, 뷰행렬, 투영행렬
pDev->GetViewport(&vp);
pDev->GetTransform(D3DTS_VIEW, &mtViw);
pDev->GetTransform(D3DTS_PROJECTION, &mtPrj);

// 뷰행렬의 역 행렬
D3DXMatrixInverse(&mtViwI, NULL, &mtViw);

// 카메라 위치, x, y, z 축
vcCamP = D3DXVECTOR3(mtViwI._41, mtViwI._42, mtViwI._43);
vcCamX = D3DXVECTOR3(mtViwI._11, mtViwI._12, mtViwI._13);
vcCamY = D3DXVECTOR3(mtViwI._21, mtViwI._22, mtViwI._23);
vcCamZ = D3DXVECTOR3(mtViwI._31, mtViwI._32, mtViwI._33);

// 뷰포트에서 화면 너비, 높이 구함
FLOAT fW    = (FLOAT)vp.Width;          // 화면너비
FLOAT fH    = (FLOAT)vp.Height;         // 화면높이
```

```
FLOAT w     =  mtPrj._11;
FLOAT h     =  mtPrj._22;
FLOAT fNear = -mtPrj._43/mtPrj._33;     // Near 값

// [-1,1]로 정규화
FLOAT viw_x =  ( 2.f * mouseX / fW - 1 ) / w;
FLOAT viw_y = -( 2.f * mouseY / fH - 1 ) / h;
FLOAT viw_z =  fNear;

// 카메라 공간으로 변환 하기 위해 Near 값에 비례해서 크기를 늘림.
// Near 값이 1이면 이 과정은 필요 없음
viw_x *= fNear;
viw_y *= fNear;

//최종 위치
vcPick = vcCamP + viw_x * vcCamX + viw_y * vcCamY + viw_z * vcCamZ;
…
```

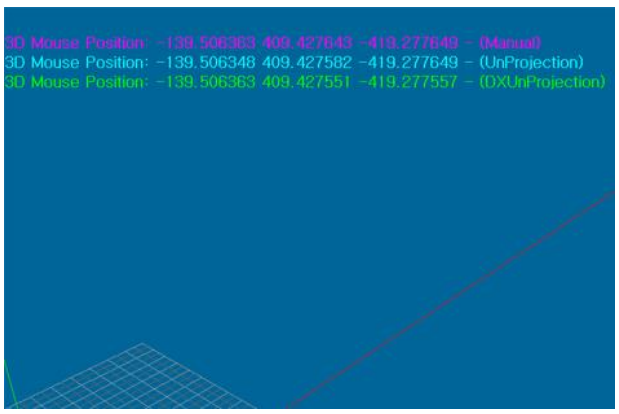
두 번째 방법은 각각의 변환 행렬을 구하고 이들을 순서대로 곱한 행렬의 역 행렬을 화면 좌표에 적용합니다.

```
D3DXVECTOR3 GetMouse3DPositionUnTM(…)
…
pDev->GetViewport(&vp);
D3DXMatrixViewport(&mtVpt, &vp);

pDev->GetTransform(D3DTS_VIEW, &mtViw);
pDev->GetTransform(D3DTS_PROJECTION, &mtPrj);

// 변환행렬= 월드행렬* 뷰행렬* 투영행렬* 뷰포트행렬
D3DXMATRIX  mtTMpt  = mtViw * mtPrj * mtVpt;
D3DXMATRIX  mtTMptI;

// 변환 행렬의 역 행렬
D3DXMatrixInverse(&mtTMptI, NULL, &mtTMpt);
```

```
// 스크린 좌표 설정
// Near 평면의 점들은 투영변환 후 깊이가 0(=z)
vcPick = D3DXVECTOR3(mouseX, mouseY, 0);

//최종 위치
D3DXVec3TransformCoord(&vcPick, &vcPick, &mtTMptI);
…
```

세 번째, D3DX 함수 사용에 대한 코드 구현입니다.

```
D3DXVECTOR3 GetMouse3DPositionDXUnProjection(…)
…
pDev->GetViewport(&vp);
D3DXMatrixViewport(&mtVpt, &vp);

pDev->GetTransform(D3DTS_VIEW, &mtViw);
pDev->GetTransform(D3DTS_PROJECTION, &mtPrj);

// 스크린 좌표 설정
vcPick = D3DXVECTOR3(mouseX, mouseY, 0);

// 최종 위치
D3DXVec3Unproject(&vcPick, &vcPick, &vp, &mtPrj, &mtViw, NULL);
```

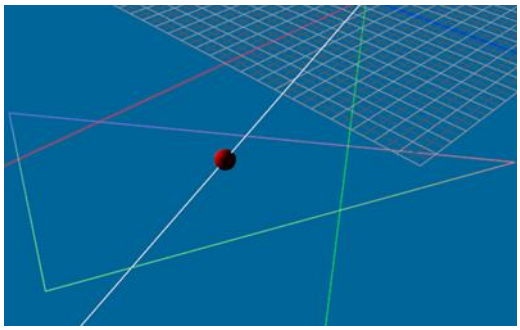


< 3차원 마우스 위치: Manual, 역 변환 행렬, D3DXVec3Unproject: LcHg44_mouse_to_3d.zip

다음 단계는 카메라의 위치와 월드 공간의 near 평면 위에 있는 마우스 위치를 지나는 직선과 지형의 삼각형 충돌을 검사합니다.

D3DX의 D3DXIntersectTri() 함수는 직선과 삼각형의 충돌을 빠르게 검사하고 충돌 위치를 간단한 산술 연산으로 계산할 수 있도록 관련된 정보를 반환하는 함수로 사용방법은 다음과 같습니다.

```
D3DXVECTOR3  vcRayPos, vcRayDir;        // 직선의 시작점, 방향
D3DXVECTOR3  vcPick, V0, V1, V2;
FLOAT   U, V, D;                        // D: 충돌 지점에서 직선 시작점까지 거리
if(D3DXIntersectTri( &V0, &V1, &V2, &vcRayPos, &vcRayDir, &U, &V, &D))
{
        // Picking 위치
        vcPick = V0 + U * (V1-V0) + V * (V2-V0);
}
```

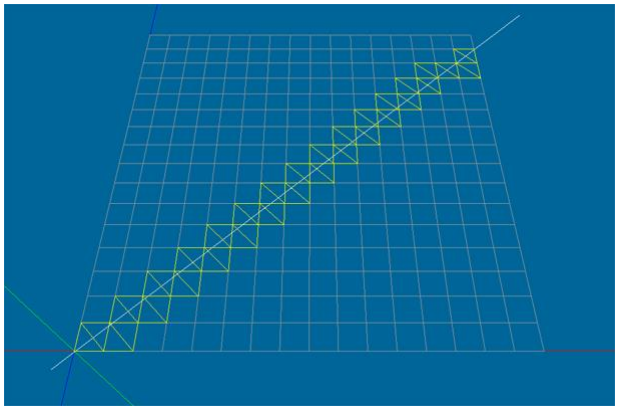


<D3DXIntersectTri 함수 사용 예>

D3DXIntersectTri 함수의 구현 원리는 충돌 부분을 참고 하기 바랍니다. 이제 남아 있는 지형의 삼각형에 대한 피킹을 완료할 단계입니다.
가장 간단한 방법은 지형의 모든 삼각형을 D3DXIntersectTri() 함수로 검사하는 방법입니다. 그런데 이 방법은 타일의 수가 증가할수록 검색의 부담이 커지는 단점이 있습니다. 좀 더 효율 적인 방법은 피킹 직선과 충돌하는 타일만 골라서 처리하는 방법입니다. 이 방법은 다음 그림과 같이 먼저 충돌 직선을 x, z 평면으로 투영하고 이 직선과 충돌하는 타일들을 찾아내야 합니다.
마우스가 움직일 때마다 직선의 방향이 달라지므로 이들 타일을 모으는 것은 stl의 vector 자료구조를 사용하는 것이 좋습니다.
그림의 테스트 코드를 요약하면 다음과 같습니다.

```
std::vector<LcRect> m_vRc;      // 충돌한 타일

INT CLcPickHgt::FrameMove()
...
// 벡터 컨테이너 clear
m_vRc.clear();
```

```
for(…)
…

// 직선의 기울기로 타일을 찾아내서 컨테이너에 추가
m_vRc.push_back(rc);
```



LcHg51_picking1.zip

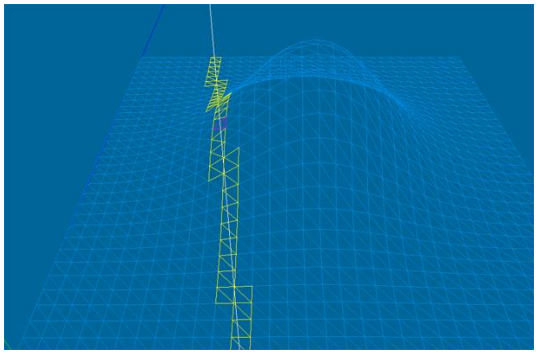
지금까지 조금 길었는데 이제 높이 맵의 충돌 삼각형을 찾아내는 일만 남았군요.
앞의 테스트 코드를 확장해서 직선의 시작점은 카메라의 위치로 설정하고 직선의 방향은 near 평면의 3차원 마우스 위치에서 카메라 위치를 뺀 값으로 설정합니다. 그리고 x, z로 투영(y=0)한 직선에 충돌하는 타일(2개의 삼각형)들을 vector 컨테이너에 추가합니다.
vector 컨테이너의 요소를 순회하면서 D3DXIntersectTri()로 충돌을 검사하고 충돌하는 삼각형을 충돌 컨테이너에 타일의 <인덱스, 거리, 직선과 충돌 위치> 값을 추가합니다.
충돌 컨테이너의 요소를 거리에 따라 오름차순으로 정렬한 후에 제일 첫 번째 요소를 가져오면 피킹은 완료가 됩니다.

```
INT CLcPickHgt::FrameMove()
…
D3DXVECTOR3 vcCamPos;   // 카메라 위치
D3DXVECTOR3 vcRayDir;   // 픽킹 직선의 방향

// 카메라 위치, 픽킹 직선의 방향 설정
vcCamPos = pCamera->GetCamPos();
…
pCamera->GetPickLayDirection(&vcRayDir, fMouseX, fMouseY);
```

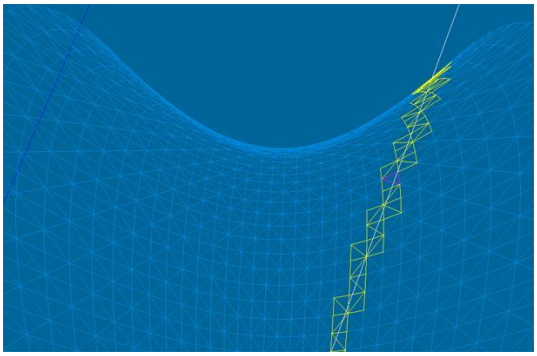
```
// 직선의 끝점을 지형을 벗어날 정도로 큰 값(여기서는 100000)으로 설정
// x, z에 투영하기 위해 y=0으로 설정
m_pLine[0].p = vcCamPos;
m_pLine[1].p = vcCamPos + vcRayDir*100000;
m_pLine[0].p.y =0;
m_pLine[1].p.y =0;

//충돌 타일 컨테이너 초기화
m_vRc.clear();
…
// x, z 방향으로 충돌 타일을 컨테이너에 추가
for(…)
{
        if(…)
                m_vRc.push_back(rc);
}

// 충돌 삼각형이 포함된 타일 리스트 초기화
m_vVc.clear();

INT iSize = m_vRc.size();

// 충돌 타일에서 충돌 삼각형이 포함된 타일을 다시 추가
for(int i=0; i<iSize; ++i)
{
        if(D3DXIntersectTri(…, & vcCamPos, & vcRayDir, &U, &V, &D))
        {
                // Pick Position
                Pck.vcP = V0 + U * (V1-V0) + V * (V2-V0);
                …
                // 충돌 삼각형이 포함된 타일 리스트에 추가
                m_vVc.push_back( Pck );
        }
}

// 충돌 삼각형이 포함된 타일 리스트를 거리에 따라 정렬
sort(m_vVc.begin(), m_vVc.end(), TsrtG<LcPck >());
```

...

전체 코드는 LcHg51_picking2.zip, LcHg51_picking3.zip 예제를 참고 하기 바랍니다.



LcHg51_picking2.zip



LcHg51_picking3.zip