높이 맵(Height Field) 2

1. Texture Splatting

텍스처 스플래팅(Texture Splatting)은 다음 그림과 같이 둘 이상의 텍스처를 혼합하는 기술입니다.

이것을 수식으로 표현하면 다음과 같습니다.

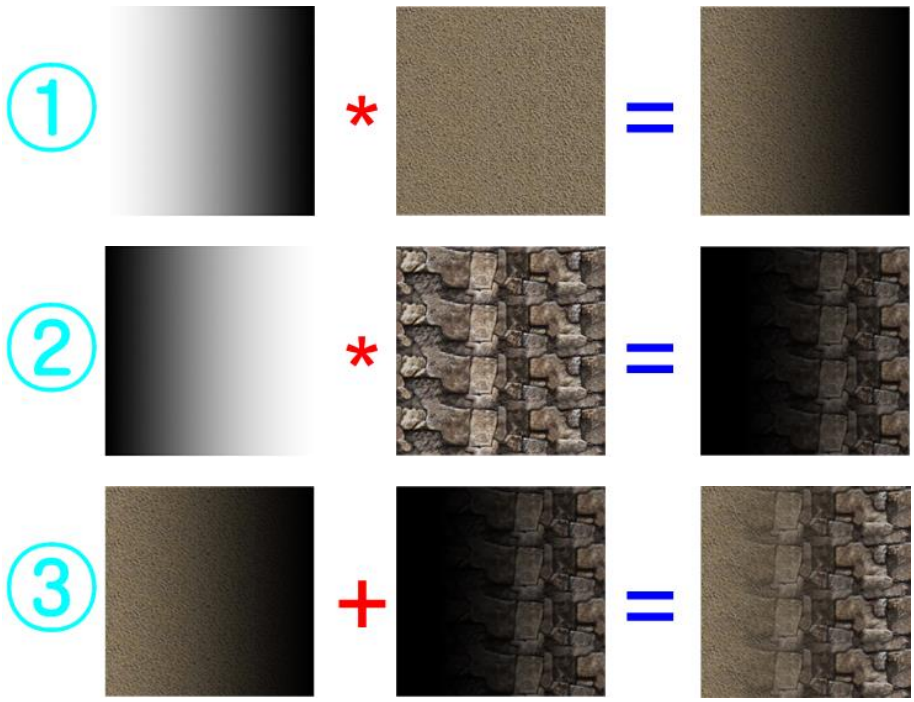
최종 색상 = $\sum \text{Texture}_i * \text{Weight}_i$

공식을 구현하기 위해서 Weight_i 값은 주로 알파(a)를 이용합니다.

위의 공식에서 Weight를 정하는 방법이 크게 두 가지가 있는데 하나는 알파 텍스처를 이용하는 것이고, 다른 하나는 정점의 알파 값을 직접 설정하는 것입니다. 만약 고정파이프라인에서 Weight 값을 설정하려고 한다면 여러 단계의 multi-texturing과 알파 블렌딩을 이용해야 합니다. 픽셀 쉐이더를 사용하는 경우라면 혼합하려는 텍스처의 숫자가 허용 이하라면 한 번의 DrawPrimitive 호출로 끝낼 수도 있습니다.
여기서는 알파 텍스처를 이용하는 방법, 정점의 알파 값을 이용하는 방법, 그리고 쉐이더를 이용하는 방법에 대해서 설명하겠습니다.

1. 알파 텍스처

알파 텍스처를 사용하는 방법을 그림으로 표현한다면 다음과 같습니다.



①, ② 그림에서처럼 각 텍스처에 알파 텍스처를 혼합합니다. 이것을 전부 더해 ③과 같이 최종 색상을 만들어 냅니다. (①, ②에서 알파텍스처의 알파 값은 검은색은 1, 흰색은 0입니다.)

만약 정점의 색상(diffuse)가 존재한다면 "최종 색상 = $\text{diffuse} * \sum \text{Texture}_i * \text{Weight}_i$"으로 구성될 것 이고 이것을 프로그램으로 적용하려면 각 단계 마다 diffuse 값을 곱해야 합니다.

최종 색상 = $\sum (\text{diffuse} * \text{Texture}_i * \text{Weight}_i)$

이 공식을 가지고 고정파이프라인에서 구현한다면 ①, ②에 대해서 다음과 같이 texture stage를 설정해야 합니다.

```
void CLcSplt::Render()
...
//0~3 stage에 대한 텍스처 설정
m_pDev->SetTexture(0, m_pTxB[0]);       // ① 과정 diffuse 텍스처
m_pDev->SetTexture(1, m_pTxA[0]);       // ① 과정 alpha 텍스처
m_pDev->SetTexture(2, m_pTxB[1]);       // ② 과정 diffuse 텍스처
m_pDev->SetTexture(3, m_pTxA[1]);       // ② 과정 alpha 텍스처
```

```
// 0-stage: 0-stage 텍스처 * 디퓨즈
m_pDev->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
m_pDev->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
m_pDev->SetTextureStageState(0, D3DTSS_COLOROP,   D3DTOP_MODULATE);

//① 과정: 0-stage 색상 * 1-stage 텍스처 => 임시 레지스터(D3DTA_TEMP)에 저장
m_pDev->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_CURRENT);
m_pDev->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_TEXTURE);
m_pDev->SetTextureStageState(1, D3DTSS_COLOROP,   D3DTOP_MODULATE);
m_pDev->SetTextureStageState(1, D3DTSS_RESULTARG, D3DTA_TEMP);

//2-stage: 2-stage 텍스처 * 디퓨즈 => current 레지스터(D3DTA_CURRENT)에 저장
m_pDev->SetTextureStageState(2, D3DTSS_COLORARG1, D3DTA_TEXTURE);
m_pDev->SetTextureStageState(2, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
m_pDev->SetTextureStageState(2, D3DTSS_COLOROP,   D3DTOP_MODULATE);
m_pDev->SetTextureStageState(2, D3DTSS_RESULTARG, D3DTA_CURRENT);

//② 과정: 3-stage 색상 * 3-stage 텍스처
m_pDev->SetTextureStageState(3, D3DTSS_COLORARG1, D3DTA_CURRENT);
m_pDev->SetTextureStageState(3, D3DTSS_COLORARG2, D3DTA_TEXTURE);
m_pDev->SetTextureStageState(3, D3DTSS_COLOROP,   D3DTOP_MODULATE);

//①+② 과정: 임시 레지스터 색상 * 3-stage 색상
m_pDev->SetTextureStageState(4, D3DTSS_COLORARG1, D3DTA_CURRENT);
m_pDev->SetTextureStageState(4, D3DTSS_COLORARG2, D3DTA_TEMP);
m_pDev->SetTextureStageState(4, D3DTSS_COLOROP,   D3DTOP_ADD);
…
for(…)
…
        //state texture 좌표는 0-stage texture 좌표 사용
        m_pDev->SetTextureStageState(1+i, D3DTSS_TEXCOORDINDEX, 0);

m_pDev->DrawPrimitiveUP(…);
```

이렇게 멀티 텍스처 처리를 설정하고 spt01_base1_one.zip을 실행하면 모래와 암석의 혼합이 연출됩니다.

Direct3D의 고정 파이프라인은 최대 8장의 텍스처를 설정할 수 있습니다. 앞의 예와 같이 한 번의 처리 과정으로 splatting을 구현할 수 있지만 이 구현은 최대 4장의 텍스처 혼합만 처리할 수 있습니다. 게임에서는 범프 맵, 라이팅 맵 등이 적용될 수 있으므로 실제로 2~3장의 혼합이 정도가 됩니다.
따라서 한 번에 처리하는 것보다 고전적인 방법인 여러 번 렌더링이 효과가 있으며 이것을 프로그램으로 구현하기도 수월합니다.
만약 여러 번 렌더링 하는 경우 렌더링 상태를 주의해야 하는데 같은 정점을 반복해서 그리게 되므로 렌더링 상태 중에서 depth test 의 함수는 less equal로 설정하고 알파 테스트는 통과하도록 하며 알파 블렌딩은 source * source alpha + dest * (1-source alpha)로 설정합니다.

```
// depth test: less equal
D3DCMPFUNC func = D3DCMP_LESSEQUAL;
m_pDev->SetRenderState(D3DRS_ZFUNC,    (DWORD)func);

// alpha test: disable
m_pDev->SetRenderState(D3DRS_ALPHATESTENABLE,    FALSE);

// alpha blending : source * source alpha + dest *(1.0 - source alpha)
m_pDev->SetRenderState(D3DRS_ALPHABLENDENABLE,   TRUE);
m_pDev->SetRenderState(D3DRS_SRCBLEND,  D3DBLEND_SRCALPHA);
m_pDev->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
```

여러 번 렌더링 하기 때문에 멀티 텍스처 단계는 색상은 색상 처리 과정으로, 알파는 알파 처리 과정으로 분리해서 합니다.
먼저 0-stage는 색상 값을 결정하는 단계로 텍스처와 정점의 디퓨즈를 곱한 값으로 결정합니다.
알파 값은 1-stage에서 결정할 것이므로 정점의 diffuse에 있는 알파 값을 그대로 사용합니다.

```
//0-stage: 색상 = 텍스처 * 정점 처리 후 diffuse 색상
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: 알파 = 정점 처리 후 diffuse 알파
m_pDev->SetTextureStageState(0, D3DTSS_ALPHAARG1, D3DTA_DIFFUSE);
m_pDev->SetTextureStageState(0, D3DTSS_ALPHAOP,   D3DTOP_SELECTARG1);
```

다음 1-stage는 알파 값을 결정하는 단계로 1-stage의 알파 텍스처의 알파 값을 최종 알파 값으로 결정합니다. 색상은 0-stage의 색상을 사용합니다.

```
//1-stage: 색상 = 0-stage 색상
m_pDev->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_CURRENT);
m_pDev->SetTextureStageState(1, D3DTSS_COLOROP,   D3DTOP_SELECTARG1);

//1-stage: 알파 = 1-stage 알파
m_pDev->SetTextureStageState(1, D3DTSS_ALPHAARG1, D3DTA_TEXTURE);
m_pDev->SetTextureStageState(1, D3DTSS_ALPHAOP,   D3DTOP_SELECTARG1);
```

이렇게 각각의 텍스처 처리에 대한 상태가 설정이 되면 GPU는 정점과 텍스처를 가지고 색상과 알파는 다음과 같이 결정합니다.

색상 = diffuse * diffuse 텍스처
알파 = alpha texture 알파

텍스처 처리에 대한 상태 설정이 끝났고 알파 처리 과정(1-stage)에서도 0-stage의 좌표를 사용하도록 설정 합니다.

```
m_pDev->SetTextureStageState(1, D3DTSS_TEXCOORDINDEX, 0);
```

이제 for문을 사용해서 diffuse 텍스처, alpha texture를 설정하고 반복해서 렌더링 하면 앞서 알파 블렌딩 옵션(source * source alpha + dest * (1-source alpha)에 의해서 ③의 과정인 "$\sum(diffuse * Texture_i * Weight_i)$" 공식을 따라 splatting이 완료가 됩니다.

```
m_pDev->SetFVF(…);
//같은 정점에 <텍스처, 알파>를 여러 번 반복해서 렌더링
for(i=0; i<m_nTex; ++i)
{
        m_pDev->SetTexture(0, m_pTxB[i]);       //0-stage: diffuse 텍스처
        m_pDev->SetTexture(1, m_pTxA[i]);       //1-stage: alpha 텍스처
        m_pDev->DrawPrimitiveUP(…);
}
```

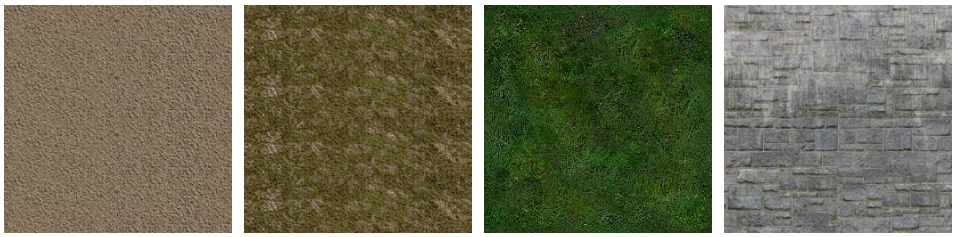
<여러 번 렌더링: spt01_base2_multi.zip>

이전의 예제는 이해를 돕기 위해서 RHW로 연습했습니다. 우리는 이것을 게임에 사용할 높이 맵에 적용하기 위해서 월드 공간의 3차원 좌표로 바꾸어 봅시다. 이 작업은 아주 쉬운데 월드 공간으로 바꾸기 위해서 D3DFVF의 RHW를 제거하고 위치에 대한 VECTOR4 타입을 VECTOR3로 바꾸고 적당한 월드 좌표([0,0,0]~[1024,0,1024])를 설정하면 끝입니다.

<월드 공간의 4각형: spt01_base3_world.zip>

2. 알파 텍스처

스플래팅의 기본 원리를 이해했다면 다음 단계로 알파 텍스처를 텍스처의 가중치에 따라 실시간으로 만드는 것입니다. 이 장에서는 32 * 32 타일에 다음과 같이 4장의 텍스처를 혼합해 보겠습니다.



<diffuse map>

알파 텍스처를 만드는 것이 목적이므로 실시간으로 텍스처의 인덱스를 계산하지 않고 다음과 같이 32*32 타일(정점: 33*33)의 각각의 정점에 대한 텍스처 인덱스가 다음과 같이 주어진다고 가정하겠습니다.

```
INT m_IdxLA[][33] =
{
        {0,0,0,1,1,1,1,1,    1,1,3,3,3,3,0,0, …
        {0,0,0,1,1,1,1,1,    1,1,3,3,3,3,0,0, …
        {0,0,0,1,1,1,1,1,    1,1,3,3,3,3,0,0, …
        {0,0,0,1,1,1,1,1,    1,1,3,3,3,3,0,0, …
        …
};
```

이 인덱스에 대한 알파 텍스처 생성은 D3DXCreateTexture() 함수를 사용하고 알파를 저장하기 위해서 픽셀 포맷을 D3DFMT_A8R8G8B8으로 설정합니다.

```
D3DXCreateTexture( m_pDev, nTile, nTile
                , 1, 0, D3DFMT_A8R8G8B8, D3DPOOL_MANAGED, &(m_pTex[i].pTxA));
```

2	2	3
3	3	3
1	1	3

간단한 비중 값(알파 값) 설정은 9-con sampling과 비슷하게 인접한 타일의 주변을 검색해서 비중 값을 설정 합니다. 그림처럼 중앙 cell의 인덱스가 3이고 주변의 인덱스를 조사하면 '3'인덱스가 총 5개가 있고 이를 전체 개수 '9'로 나누는 것입니다. 이 값을 3인덱스에 대한 알파 텍스처에 저장하고 스플래팅 처리에 적용하면 되는 것입니다.

만약 좀 더 넓은 범위가 된다면 시작 왼쪽, 오른쪽, 위, 아래에 대한 시작 인덱스를 정해서 하면 됩니다.

```
void CLcSplt::CalculateMapTile(int nTx, PDTX& xpTxA)
…
FLOAT   fAlpha;         // 알파 값 저장 변수
INT     nXBgn, nXEnd;   // x 방향 시작, 끝 인덱스
INT     nZBgn, nZEnd;   // z 방향 시작, 끝 인덱스

// 알파 텍스처 색상 포인터 얻기
xpTxA->GetLevelDesc(0, &sf);
```

```
xpTxA->GetSurfaceLevel(0, &pSf);

pSf->LockRect(&rc, 0, 0);

//32bit 색상 값으로 변경
DWORD* pPxl= (DWORD*)rc.pBits;

for (z=0; z<(int)sf.Height; ++z)
{
        for (x=0; x<(int)sf.Width; ++x)
        {
                //alpha 값 초기화
                fAlpha = 0.0f;
                …
                for(m=nZBgn; m<=nZEnd; ++m)
                {
                        for(n=nXBgn; n<=nXEnd; ++n)
                        {
                                //주어진 인덱스와 같다면 1 증가
                                if(m_IdxLA[z+m][x+n] ==nTx)
                                        fAlpha +=1.f;
                        }
                }

                // 인접한 인덱스 수
                fN = abs( (nXEnd-nXBgn) * (nZEnd-nZBgn) );

                fAlpha *=2.0f;
                fAlpha /= fN;
                if(fAlpha>1.0f)
                        fAlpha = 1.0f;
                …
                // 알파 채널에 저장
                pPxl[sf.Width*z + x] = D3DXCOLOR(1,1,1, fAlpha);
        }
}
…
```

이 함수가 올바르게 작동하는지 이들 알파 텍스처를 이미지 파일로 저장해서 다음 그림과 같이 볼 수 있다면 알파 텍스처 생성은 성공한 것입니다.

<alpha map>

텍스처 샘플링의 필터를 적용하지 않고 렌더링 하면 알파가 어느 정도 영향을 주었는지 대충 눈으로 확인할 있습니다.

<혼합: no filtering>

<filtering, fAlpha=2.0F>

spt02_alpha_tex_app.zip

함수의 지역 변수 fAlpha를 아주 크게하면 알파 값은 거의 0 아니면 1이 될 것입니다. 이 들을 비교해서 게임에서 자연스러운 연출을 만들어내는 값을 찾을 수 있도록 이후에 툴 프로그램 등에 추가 시킬 수도 있겠습니다.

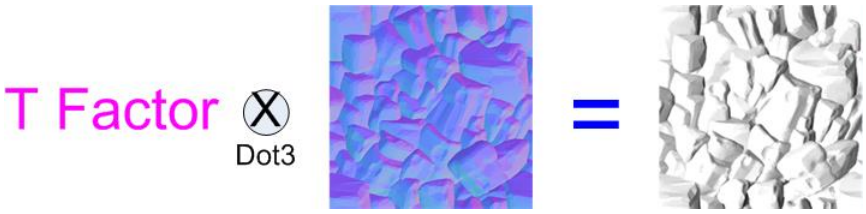
<fAlpha=1.0F>　　<fAlpha=1000.0F>

spt02_alpha_tex_app.zip

3. 범프 효과(Bump Effect)

스플래팅과 함께 범프 효과를 적용하면 좀 더 화려한 장면을 만들어 낼 수 있습니다. 현재 대부분 픽셀 쉐이더를 사용하는 하지만 고정 기능 파이프라인도 이것을 완벽하지는 않지만 구현해 낼 수는 있습니다.

범프 효과는 법선 벡터를 색상 RGB로 저장한 법선 맵(Normal map) 또는 범프 맵(Bump map) 이라는 텍스처의 색상을 처리 과정에서 다시 xyz 법선으로 변경하고 이것을 램버트 확산이나 퐁 반사를 구현하는 것입니다. D3D는 램버트 확산만 적용됩니다.
램버트 확산은 빛의 방향과 법선의 내적으로 반사의 크기가 결정이 됩니다. D3D는 멀티 텍스처 과정에서 색상 혼합 연산자 COLOROP 중에 D3DTOP_DOTPRODUCT3를 선택하면 법선 벡터를 법선 맵의 색상 rgb를 xyz로 변경해서 사용하고, 빛의 방향 벡터는 T-Factor(Texture factor)의 색상으로 입력받아 처리합니다.

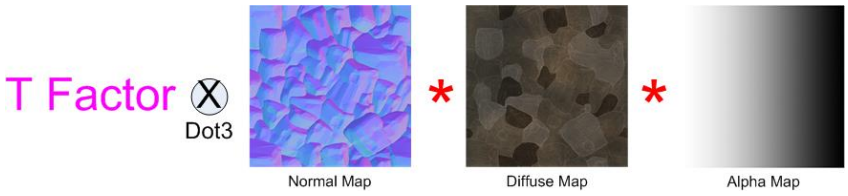


<T-factor와 범프 맵 색상에 대한 D3DTOP_DOTPRODUCT3 연산자 처리>

T-factor와 D3DTOP_DOTPRODUCT3 연산자로 얻어진 값을 다시 멀티텍스처 단계에 넣어서 diffuse 텍스처에 적용하면 입체감이 있는 연출을 볼 수 있습니다.

< D3DTOP_DOTPRODUCT3 연산 결과와 diffuse 텍스처 혼합(Modulate2X) 처리>

이들 연산은 멀티 텍스처 과정에서 한꺼번에 처리되므로 처리의 내용을 다음 그림처럼 표현할 수가 있습니다.



<Splatting과 범프 처리 과정>

그림을 보면 먼저 T-factor 값을 설정해야 하고 0-state는 Normal 맵, 1-stage는 Diffuse 맵, 2-stage는 Alpha 맵을 순서대로 설정하도록 하고 곱셈(Modulate)는 감산 연산이라 점점 어두워 지기 때문에 필요하다면 밝기를 2배 올린 Modulate2X나 4배 올린 Modulate4X를 사용합니다.

먼저 3차원 빛의 방향 벡터를 DWORD형으로 변경하는 함수 입니다.

```
void CLcSplt::VectorToRGB()
{
	// 빛의 방향 벡터 정규화
	D3DXVec3Normalize(&m_vcLgt, &m_vcLgt);

	DWORD dwR = (DWORD)(100.f * m_vcLgt.x + 155.f);
	DWORD dwG = (DWORD)(100.f * m_vcLgt.y + 155.f);
	DWORD dwB = (DWORD)(100.f * m_vcLgt.z + 155.f);

	// 3차원 벡터를 32비트 색상으로 변경
	m_dTFt = (DWORD)(0xff000000 + (dwR << 16) + (dwG << 8) + dwB);
}
```

이렇게 구한 T-factor 값은 Render State 함수를 사용해서 설정합니다.

```
m_pDev->SetRenderState(D3DRS_TEXTUREFACTOR, m_dTFt);
```

멀티 텍스처 0-stage는 단계는 범프 맵과 T-factor의 내적(dot3) 연산입니다. 알파 처리는 disable 시킵니다.

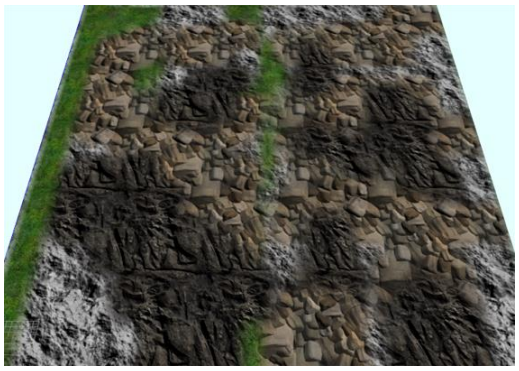
```
m_pDev->SetTextureStageState(0, D3DTSS_COLORARG1,  D3DTA_TEXTURE);
m_pDev->SetTextureStageState(0, D3DTSS_COLORARG2,  D3DTA_TFACTOR);
m_pDev->SetTextureStageState(0, D3DTSS_COLOROP   ,  D3DTOP_DOTPRODUCT3);
m_pDev->SetTextureStageState(0, D3DTSS_ALPHAOP   ,  D3DTOP_DISABLE);
```

1-state는 D3DTOP_DOTPRODUCT3 연산 결과와 diffuse 텍스처를 블렌딩 합니다. 이 때 곱셈(Modulate) 연산이 필요한데 곱셈 연산은 감산 연산이므로 Modulate2X 등도 사용해 봅니다.

```
m_pDev->SetTextureStageState(1, D3DTSS_COLORARG1,  D3DTA_CURRENT);
m_pDev->SetTextureStageState(1, D3DTSS_COLORARG2,  D3DTA_TEXTURE);
m_pDev->SetTextureStageState(1, D3DTSS_COLOROP   ,  D3DTOP_MODULATE2X);
m_pDev->SetTextureStageState(1, D3DTSS_ALPHAOP   ,  D3DTOP_DISABLE);
```

2-state는 알파 값을 결정하는 단계로서 색상은 1-stage를 그대로 사용하고 알파 값만 알파 텍스처에서 얻습니다.

```
m_pDev->SetTextureStageState(2, D3DTSS_COLORARG1,  D3DTA_CURRENT);
m_pDev->SetTextureStageState(2, D3DTSS_COLOROP   ,  D3DTOP_SELECTARG1);
m_pDev->SetTextureStageState(2, D3DTSS_ALPHAARG1,  D3DTA_TEXTURE);
m_pDev->SetTextureStageState(2, D3DTSS_ALPHAOP   ,  D3DTOP_SELECTARG1);
…
// 텍스처 설정
for(i=0; i<m_nTex; ++i)
{
        m_pDev->SetTexture(0, m_pTex[i].pTxN);          // 0-state Normal Texture
        m_pDev->SetTexture(1, m_pTex[i].pTxB);          // 1-state Diffuse Texture
        m_pDev->SetTexture(2, m_pTex[i].pTxA);          // 2-state Alpha Texture
        m_pDev->DrawIndexedPrimitiveUP(…);
}
```



<splatting-bump 적용: spt03_alpha_bump.zip>

이전의 코드에서 T-Factor와 법선 맵을 연산을 0-stage에 설정 했는데 이것은 조금이라도 멀티 텍

스처 처리 단계를 줄이기 위함입니다.

4. 디퓨즈 사용

알파 텍스처를 사용하는 방법이 직관적이지만 알파 텍스처 생성과 멀티 텍스처 단계의 설정이 초보자에게 어려움이 있습니다. 여기에 대한 간단한 대안으로 여러 번 그리는 것을 확장해서 정점의 diffuse 또한 텍스처 없이 렌더링 하고 splatting 처리는 정점의 디퓨즈 값을 이용하면 0-stage에서 모든 것을 처리할 수가 있습니다.

이해를 쉽게 하기 위해 앞서 알파 텍스처를 사용할 때처럼 다음과 같이 정점의 모든 데이터 중에서 알파 값만 차이 나는 간단한 구조의 정점 데이터를 이용하겠습니다.

```
//0번 텍스처에 적용할 정점
m_pVtx[0][0] = VtxDUV1(   0, 0, 1024,  0.0F, 0.0F,  0xFFFFFFFF);
m_pVtx[0][1] = VtxDUV1(1024, 0, 1024,  1.0F, 0.0F,  0x00FFFFFF);
m_pVtx[0][2] = VtxDUV1(1024, 0,    0,  1.0F, 1.0F,  0x00FFFFFF);
m_pVtx[0][3] = VtxDUV1(   0, 0,    0,  0.0F, 1.0F,  0xFFFFFFFF);
//1번 텍스처에 적용할 정점
m_pVtx[1][0] = VtxDUV1(   0, 0, 1024,  0.0F, 0.0F,  0x00FFFFFF);
m_pVtx[1][1] = VtxDUV1(1024, 0, 1024,  1.0F, 0.0F,  0xFFFFFFFF);
m_pVtx[1][2] = VtxDUV1(1024, 0,    0,  1.0F, 1.0F,  0xFFFFFFFF);
m_pVtx[1][3] = VtxDUV1(   0, 0,    0,  0.0F, 1.0F,  0x00FFFFFF);
```

위와 같이 구성된 정점은 렌더링에서 알파 텍스처를 사용하는 방법보다 좀 더 간결하게 작성이 되며 이것은 가장 보편적으로 사용되는 방법입니다.

```
//0-stage: 색상 = 텍스처 * diffuse
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: 알파 = 텍스처 * diffuse
m_pDev->SetTextureStageState( 0, D3DTSS_ALPHAARG1,   D3DTA_TEXTURE);
m_pDev->SetTextureStageState( 0, D3DTSS_ALPHAARG2,   D3DTA_DIFFUSE);
m_pDev->SetTextureStageState( 0, D3DTSS_ALPHAOP   ,   D3DTOP_MODULATE);
...
```

```
// 텍스처 수만큼 반복 렌더링
for (int i = 0; i < m_nTex; ++i)
{
        m_pDev->SetTexture(0, m_pTxB[i]); // diffuse 텍스처
        m_pDev->DrawPrimitiveUP(…);
}
```

spt05_diff.zip

5. 디퓨즈 응용

정점의 diffuse를 사용하면 코드의 구현은 쉽지만 각 텍스처마다 알파 맵처럼 알파 값을 저장할 버퍼가 필요합니다. 다음은 텍스처 포인터와 해당 텍스처의 비중 값을 저장하기 위한 구조체 입니다.

```
struct TexWgt
{
        PDTX    pTexB;  // 텍스처 포인터
        DWORD*  pTwgt;  // 텍스처 비중 값
};
```

알파 맵과 마찬가지로 이 구조체의 필드 pTwgt 값도 같은 방식으로 채워야 합니다.

```
void CLcSplt::CalculateMapTile(int nTx, DWORD* &xpTxA)
…
for (z=0; z<m_iTile; ++z)
{
        for (x=0; x<m_iTile; ++x)
```

```
        {
                //alpha 값 초기화
                fAlpha = 0.0f;
                …
                for(m=nZBgn; m<=nZEnd; ++m)
                {
                        for(n=nXBgn; n<=nXEnd; ++n)
                        {
                                //주어진 인덱스와 같다면 1 증가
                                if(m_IdxLA[z+m][x+n] ==nTx)
                                        fAlpha +=1.f;
                        }
                }

                // 인접한 인덱스 수
                fN = abs( (nXEnd-nXBgn) * (nZEnd-nZBgn) );

                fAlpha *=2.0f;
                fAlpha /= fN;
                if(fAlpha>1.0f)
                        fAlpha = 1.0f;
                …
                // diffuse 버퍼에 저장
                xpTxA[m_iTile*z + x] = D3DXCOLOR(1,1,1, fAlpha);
        }
}
```

이렇게 만든 디퓨즈 버퍼의 알파 값은 렌더링 할 때마다 정점 diffuse 에 복사해야 합니다.

```
for(i=0; i<m_nTex;++i)
{
        m_pDev->SetTexture(0, m_pTex[i].pTexB);

        // 알파가 저장된 diffuse 버퍼의 알파 값을 정점 diffuse 에 복사
        for(m=0; m<m_iTile; ++m)
                for(n=0; n<m_iTile; ++n)
                        m_pVtx[m*m_iTile+n].d = m_pTex[i].pTwgt[m*m_iTile+n];
```

```
        m_pDev->DrawIndexedPrimitiveUP(…);
}
```

위의 for 문 안의 이중 for(m=0; m<m_iTile; ++m) 에서 정점의 디퓨즈 값에 비중 값을 설정하고 렌더링을 합니다. 이렇게 하면 원하는 만큼 텍스처 블렌딩이 가능합니다.
또한 정점의 색상이 주어진다고 가정한다면 이 작업 후에 다음과 같이 정점의 색상을 원래 값으로 설정한 다음 한번 더 렌더링 해서 모든 것을 완료합니다.

```
for(m=0; m<m_iTile; ++m)
        for(n=0; n<m_iTile; ++n)
                m_pVtx[m*m_iTile+n].d = 정점 버퍼 색상[m*m_iTile+n];
…
m_pDev->DrawIndexedPrimitiveUP(…);
```

<정점 diffuse 를 사용한 Splatting: spt06_diff_app.zip>

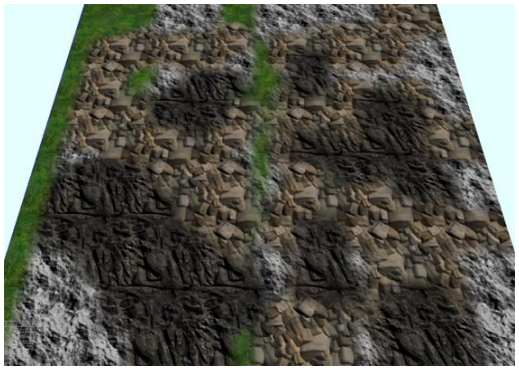
6. 디퓨즈 응용 범프 효과

텍스처 splatting은 알파 텍스처를 사용하거나 정점 디퓨즈를 사용하더라도 둘 다 비슷하게 표현이 됩니다. 따라서 범프 효과도 멀티 텍스처 stage 설정만 조금 차이가 있을 뿐입니다. 대부분 설명이 되었기 때문에 자세한 설명은 생략하겠습니다.

```
//0-stage: 색상 = 내적(T-factor, 범프 맵 텍스처)
m_pDev->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
m_pDev->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_TFACTOR);
m_pDev->SetTextureStageState(0, D3DTSS_COLOROP   , D3DTOP_DOTPRODUCT3);
m_pDev->SetTextureStageState(0, D3DTSS_ALPHAOP   , D3DTOP_DISABLE);
```

```
//1-stage 텍스처 좌표: 0-stage 텍스처 좌표 사용
m_pDev->SetTextureStageState(1, D3DTSS_TEXCOORDINDEX,   0);

//1-stage: 색상 = 0-stage 색상 * diffuse 텍스처 * 2
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: 알파 = 1-stage 텍스처 알파
m_pDev->SetTextureStageState(1, D3DTSS_ALPHAARG1, D3DTA_DIFFUSE);
m_pDev->SetTextureStageState(1, D3DTSS_ALPHAOP   , D3DTOP_SELECTARG1);

m_pDev->SetFVF(…);

// splatting 텍스처 수만큼 반복해서 렌더링
for(i=0; i<m_nTex; ++i)
{
        m_pDev->SetTexture(0, m_pTex[i].pTexN); // 0-stage bump 맵 텍스처
        m_pDev->SetTexture(1, m_pTex[i].pTexB); // 1-stage diffuse 맵 텍스처

        // 알파가 저장된 diffuse 버퍼의 알파 값을 정점 diffuse 에 복사
        for(int m=0; m<m_iTile; ++m)
                for(int n=0; n<m_iTile; ++n)
                        m_pVtx[m*m_iTile+n].d = m_pTex[i].pTwgt[m*m_iTile+n];

        m_pDev->DrawIndexedPrimitiveUP(…);
}

// 텍스처 설정 해제
m_pDev->SetTexture(0, NULL);
m_pDev->SetTexture(1, NULL);
```

알파 텍스처 사용에서와 마찬가지로 T-Factor 값을 먼저 계산한 후에 스플래팅을 적용합니다. 정점의 디퓨즈를 사용하면 알파 텍스처 보다 멀티 텍스처 단계는 줄었지만 렌더링은 한 번 더 늘어나게 되어 전체적으로 이 둘의 속도 차는 거의 없다고 볼 수 있습니다.



spt07_diff_bump.zip

7. 픽셀 쉐이더 처리

현재 대 부분의 그래픽 카드는 vertex shader, pixel shader를 지원합니다. 만약 픽셀 처리 과정에서 범프 효과, 스페큘러 효과 등을 사용을 안 한다면 고정 기능 파이프라인을 사용해도 충분하지만 shader를 사용하면 이전의 코드가 좀 더 간결해지고 이 후 확장성에도 유리한 점이 많습니다. 초창기 그래픽 카드들의 sampler 객체가 빈약했지만 ps1.4 버전 이후 6장 이상 처리할 수 있어서 가능하다면 shader를 사용하는 것이 유리하다고 할 수 있습니다.

앞서 보여준 예제들은 텍스처의 비중 값을 알파 텍스처에 저장 했거나 아니면 또 다른 버퍼를 두어서 저장한 다음 각각의 텍스처를 처리할 때마다 복사해서 사용했었습니다. 픽셀 쉐이더를 사용하더라도 텍스처에 대한 비중 값은 가지고 있어야 합니다. 한 번의 처리로 끝나야 하므로 정점 구조체를 선언할 때 사용하지 않은 영역에 비중 값(알파 값)을 저장합니다.

정점 구조체의 여러 필드 중에서 후보자가 될 수 있는 것은 픽셀 처리로 넘길 수 있는 필드 입니다. 정점의 diffuse, specular 색상과 텍스처 좌표가 그 대상입니다. 이들 중에서 텍스처의 비중 값으로 가장 많이 사용되는 필드는 텍스처 좌표 입니다.
diffuse는 사용할 수 있지만 정점 고유의 색상을 잃어버리기 때문에 제외가 됩니다. 남아 있는 것은 specular 색상과 텍스처 두 종류 인데 specular 색상은 게임에서 자주 사용되는 값이 아니고 4바이트로 구성되어 있어서 텍스처의 비중 값을 4종류까지 저장할 수가 있겠습니다.

대부분 텍스처 좌표를 사용하는데 텍스처 좌표는 diffuse 맵 이나 detail 맵을 사용하더라도 총 6개가 남아 있고 각각 최대 4차원으로 대충 5*4=20 가 되어 가장 탁월한 선택이라 할 수 있겠습니다.

이 장에서는 정점의 specular에 비중 값을 저장 하고 픽셀 쉐이더로 처리하는 방법을 먼저 소개하겠습니다. specular 를 사용하기 위해서 정점 구조체에 DWORD형 tw와 FVF_SPECULAR를 선언합니다.

```
struct VtxDSUV1
{
	VEC3	p;
	DWORD	d;
	DWORD	tw;
	FLOAT	u, v;
	enum {	FVF= (D3DFVF_XYZ|D3DFVF_DIFFUSE| D3DFVF_SPECULAR |D3DFVF_TEX1)	};
};
```

텍스처 비중 값 결정은 타일에 지정된 텍스처 인덱스를 순회하면서 정점의 specular 색상 값으로 변경합니다.

```
void CLcSplt::CalculateMap()
...
for (z=0; z<m_iTile; ++z)
{
	for (x=0; x<m_iTile; ++x)
	{
		int nTx = m_IdxLA[z][x];

		// 텍스처 비중 값 초기화
		D3DXCOLOR tw(0,0,0,0);

		// x: 0-stage 텍스처, y: 1-state 텍스처 비중 값
		// z: 2-stage 텍스처, w: 3-state 텍스처 비중 값
		if	(0 == nTx) tw.r = 1.0f;
		else if(1 == nTx) tw.g = 1.0f;
		else if(2 == nTx) tw.b = 1.0f;
		else if(3 == nTx) tw.a = 1.0f;

		// 비중 값 필드에 저장
		m_pVtx[m_iTile*z + x].tw = tw;
...
```

D3DXCOLOR 구조체는 DWORD형 캐스팅 연산자가 정의 되어 있어서 float4형을 32비트 색상 값으로 쉽게 변환 할 수 있습니다.

다음 단계로 픽셀 쉐이더 코드를 다음과 같이 작성합니다.

```
char sPxlSh[] =

"ps_2_0             Wn"     // 픽셀 쉐이더 버전
// v0: 정점 처리 diffuse
// v1: 정점 처리 specular
// s0~s3: sampler
// t0~: 입력 텍스처 좌표
"dcl v0             Wn"     // diffuse color
"dcl v1.rgba        Wn"     // specular color
"dcl_2d s0          Wn"     // sampler 0
"dcl_2d s1          Wn"     // sampler 1
"dcl_2d s2          Wn"     // sampler 2
"dcl_2d s3          Wn"     // sampler 3
"dcl t0.xy          Wn"     // texture coord 0(x,y)

// sampling 0,1,2,3 stage pixel with texturecoord 0
"texld r0, t0, s0   Wn"     // r0 = sampling(s0, t0)
"texld r1, t0, s1   Wn"     // r1 = sampling(s1, t0)
"texld r2, t0, s2   Wn"     // r2 = sampling(s2, t0)
"texld r3, t0, s3   Wn"     // r3 = sampling(s3, t0)

// modulate by weight in texture 1 coordinate(x,y,z,w)
"mul r0, r0, v1.r   Wn"     // r0 *= specular .r
"mul r1, r1, v1.g   Wn"     // r1 *= specular .g
"mul r2, r2, v1.b   Wn"     // r2 *= specular .b
"mul r3, r3, v1.a   Wn"     // r3 *= specular .a

// integrate all pixel color
"add r0, r0, r1     Wn"     // r0 = r0 + r1
"add r2, r2, r3     Wn"     // r2 = r2 + r3
"add r1, r0, r2     Wn"     // r1 = r0 + r2
"mul r0, r1, v0     Wn"     // r0 = r1 * diffuse

// copy to output register
"mov oC0, r0        Wn";     // out = r0
```

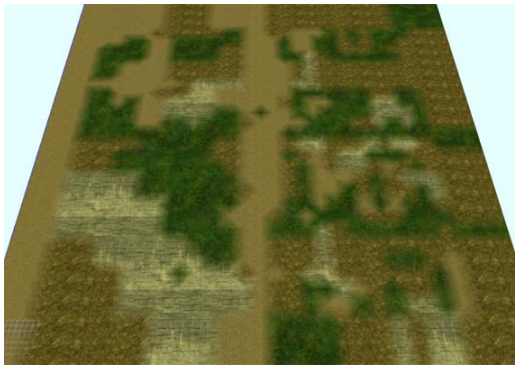
texld로 시작하는 부분은 sampler s0~s3가 주어진 텍스처 좌표 t0에 해당하는 색상을 추출해서 r0~r1에 저장하는 내용입니다. s[#]는 pDevice->SetTexture([#], pTex)에 1:1로 대응하도록 구성되어 있습니다. 예를 들어 s3은 SetTexture(3, pTex)의 3-stage에 연결된 텍스처의 색상을 가져오는 객체라 할 수 있습니다.

"mul r0, r0, v1.r" 구문은 r0 *=v1.r과 같으며 이것은 정점 처리 과정에서 끝난 specular 값(v1)에 텍스처에서 추출한 색상 값 r0를 곱한다는 의미입니다. 즉, 정점 specular 의 red 값에 텍스처 색상을 곱한 결과라 할 수 있습니다.
총 4장의 텍스처에 대해서 비중 값이 저장된 specular의 r, g, b, a를 각각 곱하고 저장했다면 남은 것은 이들의 총합을 구하는 것입니다.
add~mul 부분은 이전 과정에서 처리한 결과들을 전부 더하고 마지막 단계에서 정점 처리 과정에서 만들어진 diffuse 값을 곱해서 최종 색상을 만들어 내고 있습니다.
mov oC0, r0는 최종 단계로 임시 레지스터 r0에 저장된 색상 값을 출력 레지스터 oC0에 복사하는 명령어 입니다.

이렇게 작성한 쉐이더 코드를 빌드한 후에 다음과 같이 렌더링 코드를 작성하는데 이전 고정기능 파이프라인의 코드보다 상당히 줄어들 것을 볼 수 있습니다.

```
// 총 8-stage에 대한 텍스처 샘플링 상태 설정
for(i=0; i<8; ++i)
{
        m_pDev->SetSamplerState(i, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
        m_pDev->SetSamplerState(i, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
        m_pDev->SetSamplerState(i, D3DSAMP_MIPFILTER, D3DTEXF_LINEAR);
        m_pDev->SetSamplerState(i, D3DSAMP_ADDRESSU, D3DTADDRESS_WRAP);
        m_pDev->SetSamplerState(i, D3DSAMP_ADDRESSV, D3DTADDRESS_WRAP);
}

// 쉐이더 사용 시작 선언
m_pDev->SetVertexDeclaration(m_pFVF);
m_pDev->SetPixelShader(m_pPS);

// 각각의 stage에 텍스처 설정
for(i=0; i<m_nTex; ++i)
        m_pDev->SetTexture(i, m_pTex[i].pTexB);

m_pDev->DrawIndexedPrimitiveUP(…);
```

```
// 각각의 stage에 텍스처 해제
for(i=0; i<m_nTex;++i)
        m_pDev->SetTexture(i, NULL);

// 쉐이더 사용 해제
m_pDev->SetVertexShader(NULL);
m_pDev->SetPixelShader(NULL);
```



<정점 specular+ 픽셀 쉐이더: spt11_ps_spec.zip>

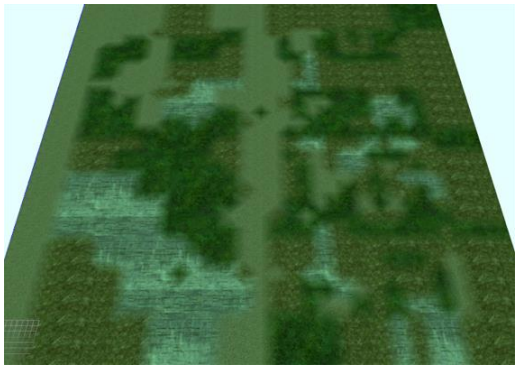
앞의 예제는 정점의 specular 값을 사용했었는데 텍스처가 4장이 넘어가거나 specular 값을 사용하게 되면 이 방법을 적용하는 것은 어려울 것입니다. 이러한 이유로 많은 프로그래머들이 텍스처의 좌표를 비중 값으로 사용하는 이유입니다.
이전 방법과 달라진 부분은 거의 없으며 정점 구조체와 픽셀 쉐이더 코드 일부가 변경되었습니다.
텍스처의 비중 값을 텍스처 좌표에 저장할 수 있도록 필드를 추가하고 FVF를 변경합니다.
FVF를 변경할 때 비중 값으로 사용할 텍스처의 좌표를 4차원(x,y,z,w)로 사용하겠다고 명시해야 합니다.
보통 default는 2차원이라 명시를 안 해도 되지만 2차원이 아니면 D3DFVF_TEXCOORDSIZE[#](텍스처-stage) 매크로를 사용해서 FVF를 결정해야 합니다.

```
struct VtxDUV1TW
{
        VEC3    p;
        DWORD   d;
        FLOAT   u, v;
        VEC4    tw;             // 텍스처 비중 값으로 사용될 텍스처 좌표
...
        enum {  FVF= (D3DFVF_XYZ | D3DFVF_DIFFUSE  | D3DFVF_TEX2 ₩
        | D3DFVF_TEXCOORDSIZE4(1)) // 1-stage의 텍스처 좌표가 4차원임을 명시
```

```
        };
};
```

쉐이더 코드는 비중 값이 specular(v1) 대신 텍스처 좌표(t1)로 저장 되었기 때문에 레지스터 선언과 곱셈 연산만 변경하면 됩니다. 참고로 예제의 "mul r0, r0, t1.x"은 이전 과정에서 처리된 r0에 1-stage texture 좌표 x(t1.x)를 곱한다는 의미입니다.

```
char sPxlSh[] =
...
// declaration
...
"dcl t0.xy          Wn"     // texture coord 0(x,y)
"dcl t1.xyzw        Wn"     // texture coord 1(x,y,z,w) v1대신 사용

// modulate by weight in texture 1 coordinate(x,y,z,w)
"mul r0, r0, t1.x   Wn"     // r0 *= texture 1.x
"mul r1, r1, t1.y   Wn"     // r1 *= texture 1.y
"mul r2, r2, t1.z   Wn"     // r2 *= texture 1.z
"mul r3, r3, t1.w   Wn"     // r3 *= texture 1.w
...
```



<정점 텍스처 좌표+ 픽셀 쉐이더: spt11_ps_texcoord.zip>

8. HLSL 이용

픽셀 쉐이더 사용도 충분하지만 쉐이더 자체가 저 수준 언어로 표현되므로 이 후에 범프 효과 등을 추가하고자 한다면 사용이 쉽지만 않습니다. HLSL은 고 수준 언어로 쉐이더를 작성할 수 있어서 처음 사용하는 사람도 쉽게 쉐이더를 만들 수가 있습니다. 또한 ID3DXEffect 객체를 사용하면

쉐이더의 상수 설정이 쉽고 하나의 파일 안에 여러 쉐이더 코드를 작성하고 필요에 따라 모을 수 습니다. 이 부분은 쉐이더 설명을 참고 하기 바랍니다.

HLS은 쉽기 때문에 이전 4장까지 적용한 것을 8장으로 늘려보겠습니다. 이를 위해서 정점 구조체에 텍스처 좌표를 저장할 수 있는 필드를 추가하고 FVF를 변경합니다.

```
struct VtxDUV1TW
{
        VEC3    p;
        DWORD   d;
        FLOAT   u, v;
        VEC4    tw0;    // 0~3 텍스처 비중 값으로 사용될 텍스처 좌표
        VEC4    tw1;    // 4~7 텍스처 비중 값으로 사용될 텍스처 좌표
...
        enum {  FVF= (D3DFVF_XYZ | D3DFVF_DIFFUSE | D3DFVF_TEX3 ₩
                | D3DFVF_TEXCOORDSIZE4(1) ₩     // 1-stage 텍스처 좌표는 4차원
                | D3DFVF_TEXCOORDSIZE4(2))      // 2-stage 텍스처 좌표는 4차원
        };
};
```

하나의 커다란 버퍼에 인덱스를 저장 할 수 없어서 텍스처 좌표를 2개로 나누어서 사용해야 하고 인덱스를 채우는 함수도 변경해야 합니다.

```
void CLcSplt::CalculateMap()
...
for (z=0; z<m_iTile; ++z)
{
        for (x=0; x<m_iTile; ++x)
        {
                int nTx = m_IdxLA[z][x];

                // 텍스처 비중 값 초기화
                VEC4 tw0(0,0,0,0);
                VEC4 tw1(0,0,0,0);

                // tw0.x: 0-stage 텍스처, tw0.y: 1-state 텍스처 비중 값
                // tw0.z: 2-stage 텍스처, tw0.w: 3-state 텍스처 비중 값
```

```
                // tw1.x: 4-stage 텍스처, tw1.y: 5-state 텍스처 비중 값
                // tw1.z: 6-stage 텍스처, tw1.w: 7-state 텍스처 비중 값
                if     (0 == nTx) tw0.x = 1.0f;
                else if(1 == nTx) tw0.y = 1.0f;
                else if(2 == nTx) tw0.z = 1.0f;
                else if(3 == nTx) tw0.w = 1.0f;
                else if(4 == nTx) tw1.x = 1.0f;
                else if(5 == nTx) tw1.y = 1.0f;
                else if(6 == nTx) tw1.z = 1.0f;
                else if(7 == nTx) tw1.w = 1.0f;

                // 비중 값 필드에 저장
                m_pVtx[m_iTile*z + x].tw0 = tw0;
                m_pVtx[m_iTile*z + x].tw1 = tw1;
...
```

렌더링 상태 설정은 픽셀 쉐이더와 거의 같으며 ID3DXEffect 객체를 사용할 경우 Technique, pass 등을 지정해야 합니다.

```
// 필터링, 어드레싱, 텍스처 설정
for(i=0; i<m_nTex;++i)
{
        m_pDev->SetSamplerState(i, D3DSAMP_MINFILTER,   D3DTEXF_LINEAR);
        m_pDev->SetSamplerState(i, D3DSAMP_MAGFILTER,   D3DTEXF_LINEAR);
        m_pDev->SetSamplerState(i, D3DSAMP_MIPFILTER,   D3DTEXF_LINEAR);
        m_pDev->SetSamplerState(i, D3DSAMP_ADDRESSU ,   D3DTADDRESS_WRAP);
        m_pDev->SetSamplerState(i, D3DSAMP_ADDRESSV ,   D3DTADDRESS_WRAP);

        m_pDev->SetTexture(i, m_pTex[i].pTexB); // diffuse map
}

// 쉐이더, 테크닉 사용 선언
m_pDev->SetVertexDeclaration(m_pFVF);
m_pEffct->SetTechnique(m_hdTech);

// 테크닉의 패스 지정
m_pEffct->Begin(NULL, 0);
```

```
m_pEffct->BeginPass(0);

m_pDev->DrawIndexedPrimitiveUP(…);

m_pEffct->EndPass();
m_pEffct->End();

// 쉐이더 사용 해제
m_pDev->SetVertexDeclaration(NULL);
m_pDev->SetVertexShader(NULL);

// 텍스처 설정 해제
for(i=0; i<m_nTex;++i)
        m_pDev->SetTexture(i, NULL);
```

HLSL 코드 작성에서 제일 먼저 선언해야 할 것은 sampler 입니다. "sampler"는 register 명령어를 사용해서 쉐이더 s#를 지정할 수 있습니다. 따라서 pDevice->SetTexture(…) 함수로 지정된 텍스처를 샘플링 하게 됩니다.

```
sampler smp0 : register(s0);    // 0 - stage
…
sampler smp7 : register(s7);    // 7 - stage
```

HLSL 쉐이더 메인 함수는 정점 diffuse, 텍스처 좌표, 그리고 텍스처 비중 값을 텍스처 좌표로 받아와야 합니다. 함수의 반환은 색상이므로 float4 형으로 정하고 입력은 semantic을 사용해서 다음과 같이 작성합니다.

```
float4 PxlPrc(    float4 Dff  : COLOR0          // diffuse
                , float2 Tx0  : TEXCOORD0       // 0- stage texture coordinate
                , float4 nTx0 : TEXCOORD1       // 1- to texture weight
                , float4 nTx1 : TEXCOORD2       // 2- to texture weight
                ) : COLOR0
{
        float4 Out={0,0,0,0};

        Out += tex2D( smp0 , Tx0 ) * nTx0.x;
…
```

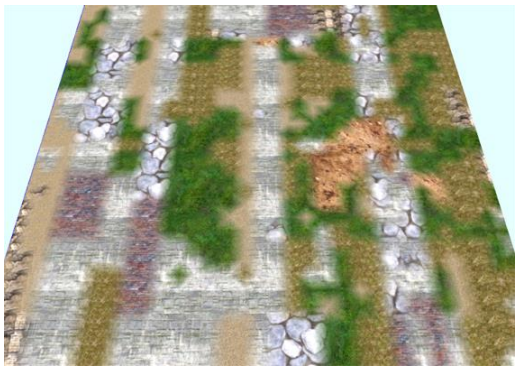
```
        Out += tex2D( smp7 , Tx0 ) * nTx1.w;
        Out *= Dff;
        Out *= 1.5;

        return Out;
}
```

tex2D()함수는 샘플러와 텍스처 좌표를 사용해서 2차원 텍스처를 추출하는 함수 입니다. PxlPrc 함수는 tex2D() 함수로 s0~s7 샘플러를 사용해서 텍스처를 추출하고 텍스처 비중 값인 nTx0.x ~ nTx.w에 곱을 한 후 순차적으로 더하고 이 결과에 diffuse 값을 곱한 다음 출력하고 있습니다. 밝기를 조금 올리기 위해서 반환 전에 1.5 시키고 있습니다.

HLSL 파일의 마지막 technique 설정은 픽셀 쉐이더 버전을 2.0으로 컴파일 하고 정점 쉐이더 없이 픽셀 쉐이더만 사용하고 있음을 볼 수 있습니다.

```
technique TShader {
    pass P0 {
        PixelShader  = compile ps_2_0 PxlPrc();
} }
```

전체 쉐이더 코드는 spt12_hlsl_samp_reg.zip의 data/hlsl.fx 파일을 참고하고 이 예제는 총 8장의 텍스처를 스플래팅 할 수 있게 만들었습니다.
아직 쉐이더가 어렵다면 이 후에 익숙해지고 나서 쉐이더 예제로 스플래팅을 꼭 구현해 보기 바랍니다.

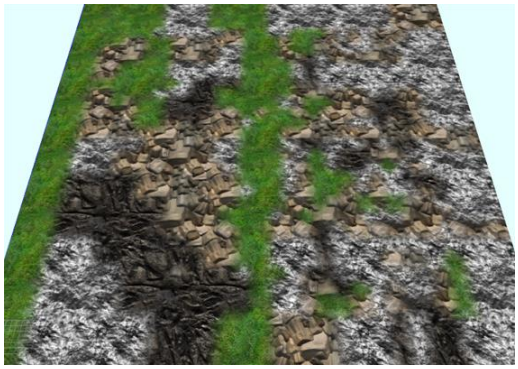


< HLSL를 사용 8장의 텍스처를 혼합한 Splatting: spt12_hlsl_samp_reg.zip>

HLSL을 사용하면 고정기능 파이프라인보다 좀 더 다양한 반사 효과를 구현할 수 있습니다. 범프 효과 또한 좀 더 유연하게 사용자가 원하는 방식으로 처리할 수가 있습니다. spt13_hlsl_bump.zip

은 범프 효과에 대한 예제인데 픽셀 처리 함수 PxlPrc()는 범프 맵의 색상을 법선 벡터 xyz로 변경하고 이 벡터를 빛의 방향 벡터와 내적을 한 다음 power 함수로 명도 대비를 높이고 이 결과 값을 splatting 에 적용하고 있습니다.

```
float4 PxlPrc(    float4 Dff  : COLOR0          // diffuse
                , float2 Tx0  : TEXCOORD0       // 0- stage texture coordinate
                , float4 nTx1 : TEXCOORD1       // 1- to texture weight
                ) : COLOR0
{
        float3 L = normalize(m_vcLgt);          // lighting direction
...
        N = 2.0 * tex2D( smpNor0, Tx0 ).xyz-1.0; // rgb를 추출해서 법선 벡터로 만듦
        C0 = dot(N, L)+1.f;                     // 빛의 방향 벡터와 내적 + 1
        C0 = pow(C0, fIgt1) * fIgt2;            // pow 함수로 명도 대비를 높임
...
        // 스플래팅에 적용
        Out += tex2D( smpDif0 , Tx0 )*nTx1.x * C0;
        Out += tex2D( smpDif1 , Tx0 )*nTx1.y * C1;
        Out += tex2D( smpDif2 , Tx0 )*nTx1.z * C2;
        Out += tex2D( smpDif3 , Tx0 )*nTx1.w * C3;
...
        return Out;
}
```



<HLSL를 이용한 범프 효과+Splatting: spt13_hlsl_bump.zip>

2. Using Volume Texture

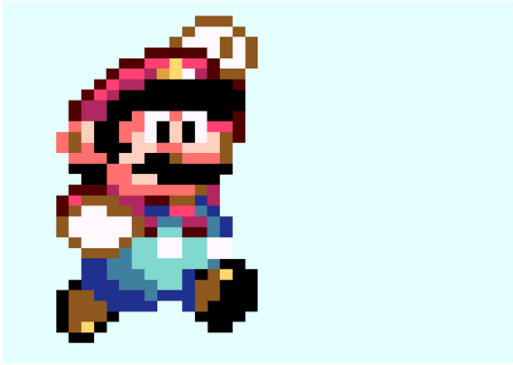
텍스처는 보통 2차원으로 구성 되어 있는 픽셀을 집합입니다. 그런데 volume 텍스처는 깊이 값을 가지고 있는 3차원 픽셀 집합입니다. 간단하게 비교한다면 포토샵의 레이어로 생각하면 됩니다. 즉, 깊이는 2차원 텍스처의 레이어 숫자와 같다고 볼 수 있습니다.

이것을 지형에 적용하면 의외로 좋은 효과를 볼 수 있습니다. 볼륨 텍스처는 DirectX SDK의 utility에 DxTex.exe로 제작이 가능하고, dds로 저장 됩니다. 이 볼륨 텍스처를 읽기 위해서 D3DXCreateVolumeTextureFromFileEx() 함수를 사용해야 하는데 이 함수의 인수들을 설정하기 위해서 D3DXGetImageInfoFromFile() 함수로 이미지 정보를 먼저 이 정보를 가지고 볼륨 텍스처를 생성합니다.

```
// 이미지 정보 얻기
hr = D3DXGetImageInfoFromFile("Texture/mario.dds", &m_pImg );

// 볼륨 텍스처 생성
hr = D3DXCreateVolumeTextureFromFileEx( m_pDev, "Texture/mario.dds",
        …, m_pImg.Depth, m_pImg.MipLevels, …, &m_pTxv );
```

볼륨 텍스처는 2D 애니메이션에서도 활용 할 수 있는데 단순히 w 값만 증가시켜서 간단히 구현됩니다.

```
INT CLcVolTx::FrameMove()
…
FLOAT d= m_pImg.Depth * 1.f;
…
if(dEnd> dBgn+100)
…
        w+=1.f;
        float d2 = w/d;
        m_pVtx[0].w = d2;
        m_pVtx[1].w = d2;
        m_pVtx[2].w = d2;
        m_pVtx[3].w = d2;
…
```



<볼륨 텍스처를 사용한 2D 애니메이션: vol01_2d.zip>

간단히 텍스처의 w 좌표만 변화시켜 캐릭터 애니메이션을 연출하는 것과 마찬가지로 w 값을 높이에 대응되도록 하면 자연스럽게 높이에 따라 텍스처 블렌딩이 만들어지게 됩니다.

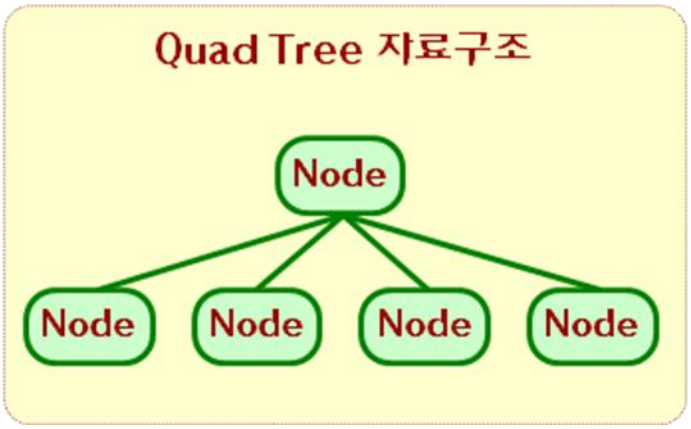
```
INT CLcVolTx::Create(PDEV pDev)
…
// Create a volume texture
hr = D3DXCreateVolumeTextureFromFileEx(…);
…
m_pVB->Lock(0, 0, (void**)&pVB, 0);
for (j = 0; j < 33; j++)
…
        // 지형의 높이(hr)에 w 값을 대응
        w= h/300.f * imageinfo.Depth +0.5f;
        pVB[idx].w = w/imageinfo.Depth;
…
```

전체 코드는 vol02_field.zip에 포함되어 있고 이 예제의 지형 높이에 사용된 함수는 로렌쯔 곡선입니다.

<볼륨 텍스처: vol02_field.zip>

3. 쿼드 트리(Quad Tree)

지형에서 정점에 대한 LOD(Level of Detail)를 적용하는 방법 중의 하나가 쿼드 트리입니다. 쿼드 트리는 카메라 위치로부터 거리나 기울기 등의 가파른 정도에 따른 레벨을 두어 높은 레벨일수록 정점을 많이 표현하고 낮은 레벨은 적게 표현해서 장면에 필요한 정점의 숫자를 조금이라도 줄이려는 알고리듬입니다.



쿼드 트리는 지형을 2차원(Top view)으로 바라 본 형태에서 4개의 자식 노드로 분할 합니다. 또한 각각의 노드들은 레벨에 따라 더 분할 할 수 있으면 마찬가지로 4개의 자식 노드를 재귀호출을 통해서 분할합니다. 이렇게 레벨에 따라 노드를 분할 다음 정점의 메쉬를 구성하고, 크랙이 발생한 부분에 대해서 패치를 진행하는 것이 기본 원리입니다.
쿼드 트리는 정점을 줄일 수 있다는 장점이 있으나 몇 가지 단점이 존재하는데 가장 큰 단점은 실시간으로 정점을 구성하므로 트리를 구성하는 부하가 더 걸릴 수 있습니다. 게다가 멀티 텍스처를 이용할 경우 정점의 UV를 설정하는 부분에서 문제가 발생할 수 있습니다. 이외에 오브젝트가 지형 위에 있을 때 높이 찾는 방법도 높이 맵과 다른 방법으로 구성해야 하고, 지형이 카메라의 위치에 따라 꾸물거리는 현상이 있어 요즘은 그리 크게 호평 받는 기술이 아닙니다. 만약 쿼드 트리를 사용할 경우라면 개발자들은 오히려 ROAM을 많이 추천하는 편입니다. 그래도 자료구조의 연습으로 좋은 예제 이기에 한 번 구현 해봅시다.

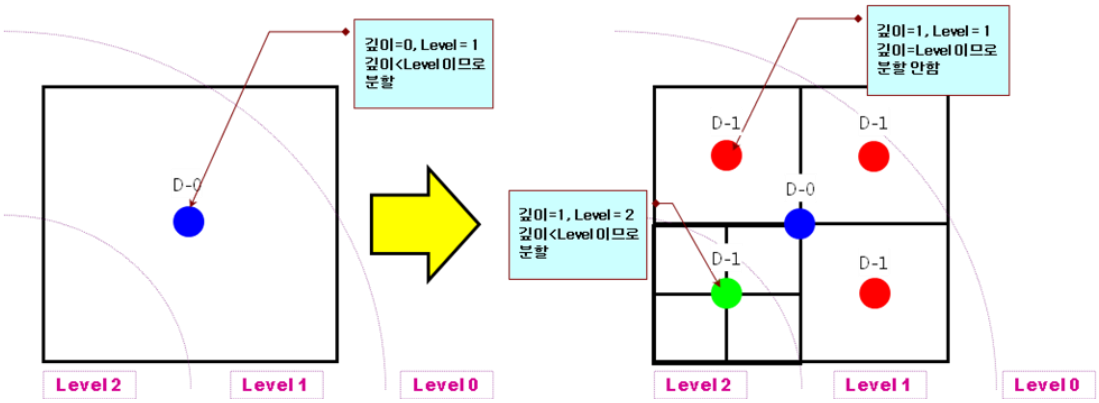
쿼드 트리의 첫 번째 단계는 레벨과 4개의 자식 노드를 가진 자료구조를 구성하는 것입니다. 간단하게 쿼드 트리의 자료구조는 4개의 자식 노드와 한 개의 부모 노드로 만들 수 있습니다.

```
struct CMcQuad
{
        CMcQuad*    m_pRH;              // right - header
```

```
        CMcQuad*    m_pLH;              // left - header 노드
        CMcQuad*    m_pLT;              // left - tail 노드
        CMcQuad*    m_pRT;              // right - tail 노드
        CMcQuad*    m_pP;               // 부모 노드

        INT     m_iDeph;                // 자신의 깊이 값
        INT     m_iLvl;                 // 자신의 레벨
        VEC3    m_vP;                   // 위치
...
};
```

노드 뿐만 아니라 자식 노드의 분할 여부를 판단할 깊이 값과 현재의 레벨 값이 필요하고, 기타 렌더링에 필요한 자료를 추가합니다.

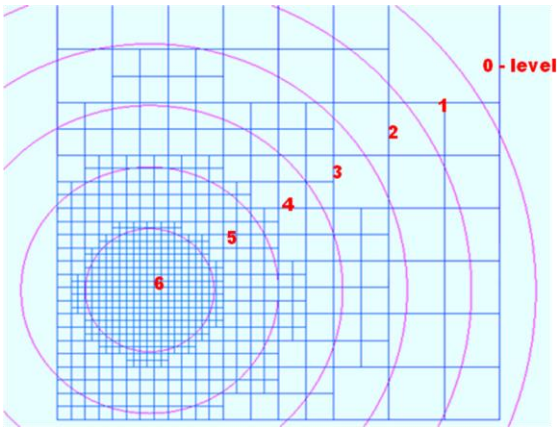
자식 노드의 분할은 먼저 부모 노드의 깊이 값을 가져와서 자신은 부모보다 한 단계 높은 값으로 변경합니다.
만약 거리에 따라 깊이 값의 레벨이 결정 된다면 노드를 분할할 때 위치를 먼저 계산해야 합니다.
이 계산된 위치 값으로 자신의 현재 레벨을 설정합니다.
만약 현재 레벨이 깊이 값보다 작거나 같으면 자식 노드를 재귀적으로 만들어 갑니다.



<쿼드 트리의 자식 노드 분할 과정>

```
void CMcQuad::Build(CMcQuad* _pPrn, INT iDepth, INT iX,INT iZ)
...
m_pP    = _pPrn;                // 부모 노드 설정
m_iDph  = (iDepth+1);           // 깊이 값 1 증가

// 부모노드의 위치에서 자신의 위치 계산
```

```
m_vcP   = m_pP->m_vcP + D3DXVECTOR3(iX*m_iBrd, 0, iZ*m_iBrd);
…
// 레벨을 결정하기 위해 카메라에서부터 거리 계산
VEC3    vcD     =  m_vcP - g_pPos;
INT     iDist   = sqrtl(vcD.x*vcD.x + vcD.z*vcD.z)-m_iBrd*1.41421 …;

m_iLvl = GetDepth(iDist); //자신의 레벨 설정

//자신의 깊이가 레벨 보다 작으면 자식 노드를 분할
if(m_iDph <= m_iLvl)
{
        m_pRH = new CMcQuad;    m_pRH->Build(this, m_iDph,  1, 1);
        m_pLH = new CMcQuad;    m_pLH->Build(this, m_iDph, -1, 1);
        m_pLT = new CMcQuad;    m_pLT->Build(this, m_iDph, -1,-1);
        m_pRT = new CMcQuad;    m_pRT->Build(this, m_iDph,  1,-1);
}
…
```

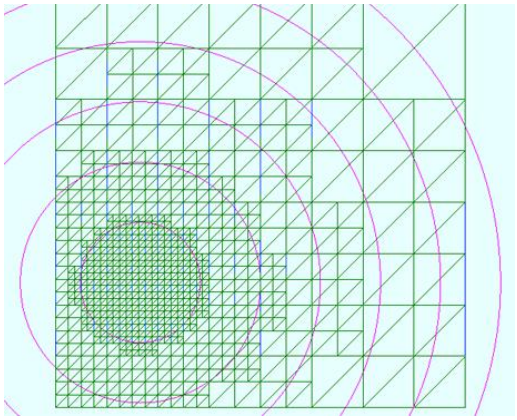


<쿼드 트리와 레벨: qdt01_basic.zip>

실질적으로 쿼드 트리의 자료 구조 내용은 여기까지 입니다. 자식 노드가 완성이 되면 화면의 렌더링 메쉬를 구성할 차례입니다.
만약 자신의 노드가 완전히 분할이 되었다면 메쉬를 만들고 그렇지 않으면 자식 노드를 재귀적으로 호출해서 자식 노드의 메쉬를 구성합니다. 이 때 각각의 메쉬는 쿼드 트리의 사각형을 기준으로 구성합니다.

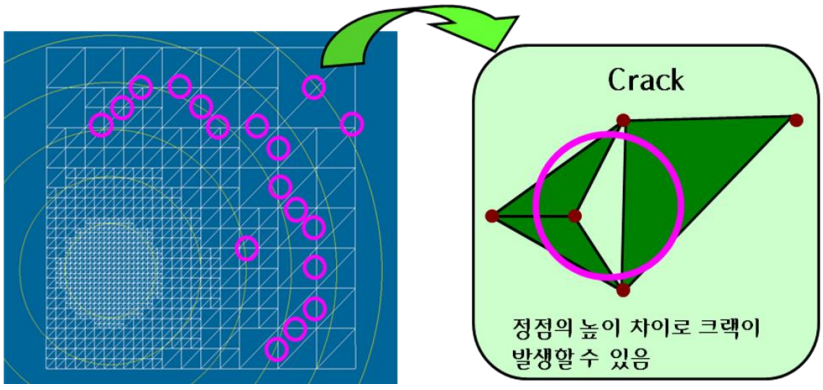
```
void CMcQuad::BuildMesh()
…
```

```
if(m_bEnd)      // 자신의 분할이 완전히 완료 되면 자신의 메쉬를 구성
...
else            // 자식 노드의 메쉬를 구성
{
        if(m_pRH)       m_pRH->BuildMesh();
        if(m_pLH)       m_pLH->BuildMesh();
        if(m_pLT)       m_pLT->BuildMesh();
        if(m_pRT)       m_pRT->BuildMesh();
}
...
```



<쿼드 트리+메쉬: qdt02_mesh.zip>

이 단계에서 모든 것이 끝나면 좋겠지만 정점의 높이 차이로 인해 크랙(crack)이 발생합니다. 이 크랙이 발생한 부분에 대해서 패치를 진행 해야 합니다.

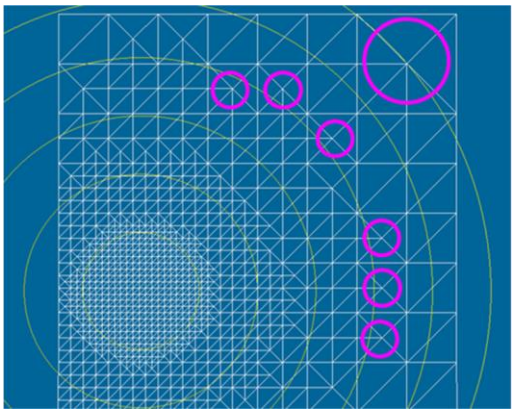


<높이와 레벨 차이로 인한 크랙 발생>

크랙의 원인은 자신의 노드와 인접한 노드의 레벨 차이 때문에 발생합니다. 따라서 인접한 노드와 비교해 보는 것이 타당하지만 이렇게 하면 부모 노드를 거쳐서 부모 노드의 자식노드들을 검사해

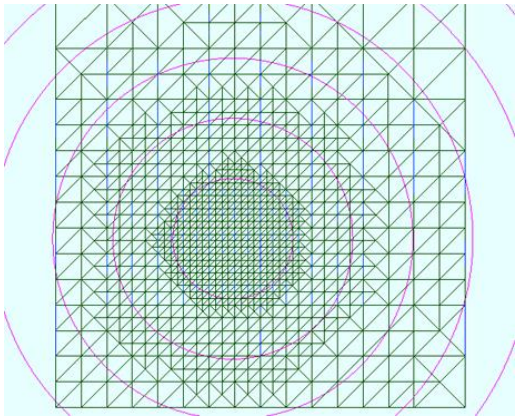
야 합니다.
만약 카메라의 거리에 레벨이 결정된다면 좀 더 쉬운 방법이 존재하는데 자신의 노드에 또 다른 깊이가 존재할 수 있는지 계산해 보는 것입니다. 상하좌우 테스트를 실행해서 레벨 차이가 있으면 크랙이 발생한 것이고 이 부분을 패치 합니다.

```
void CMcQuad::BuildMesh()
…
if(m_bEnd)
…
// 자식노드의 위치를 찾아서 레벨을 계산하고 patch 결정
for(…)
        for(…)
                // 거리 계산
                vcD   =  m_vcP - g_pPos + 2*VEC3(m_iBrd * x,0, m_iBrd * z);
                iDist = sqrtl(vcD.x*vcD.x + vcD.z*vcD.z)-m_iBrd*1.41421;

                // 레벨 계산
                iLvl  = GetDepth(iDist);

                // patch 결정
                if(iLvl>m_iLvl && iLvl>=m_iDph)
                {
                        bX |=x; bZ |=z;
                }
…
```



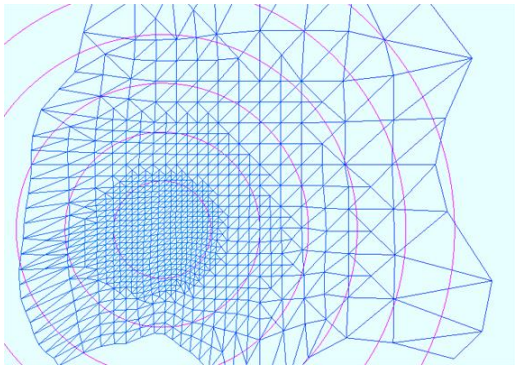
<쿼드 트리 패치: qdt03_meshpatch.zip>

크랙에 대한 패치를 완료하고 나서 카메라를 이동시켜 실시간으로 쿼드 트리가 만들어지는지 확인 합니다.



<런타임: qdt04_realtime.zip>

카메라 이동에 대해서 쿼드 트리가 실시간으로 계산이 되고 있다면 마지막으로 지형에 적용하는 일만 남아 있습니다. 높이 맵에 사용되었던 높이 값을 적용하고 텍스처를 매핑 합니다. 만약 높이 맵에 인덱스를 쿼드 트리로 구성한다면 속도가 향상 될 수 있습니다.



<높이 값 적용: qdt05_appheight.zip>