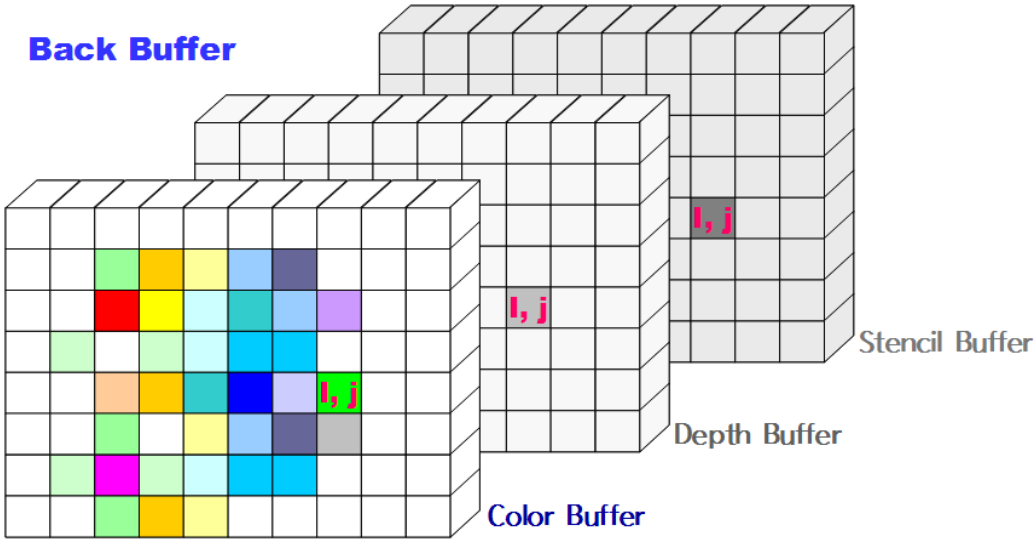
3D Game Programming Basic with Direct3D

9. Stencil

9.1 개요

스텐실 버퍼는 다음 그림처럼 후면 버퍼를 구성하는 3종 세트 중의 하나입니다. 스텐실 버퍼에는 스텐실 테스트를 위한 정수형 참조 값이 저장되어 있으며 스텐실 버퍼의 (i, j)에 해당 하는 픽셀은 깊이, 색상 버퍼의 (i, j) 와 정확하게 대응됩니다.
D3D는 스텐실 버퍼와 깊이 버퍼를 32비트로 구성하는 경우가 보통입니다. 이것은 지금까지의 프로그램 경험상으로 스텐실에 저장된 값이 0~255이면 다 쓰지도 못할 정도로 충분하다고 합니다. 그래서 보통 깊이-스텐실 버퍼를 묶어서 구성하며 깊이 버퍼에 대해서 가장 많이 사용되는 포맷은 이전 시간에 이야기 했던 D3DFMT_D24S8 입니다.



<후면 버퍼의 구성>

스텐실 버퍼가 스텐실 이름이 붙은 것은 마치 스텐실 판화처럼 동작하기 때문입니다. 초등학교 때 이미 감광지(빛느낌종이) 위에 물체를 올려 놓고 했던 실험이나 중고등학교 미술시간에 실크 스크린을 해 본 경험을 떠올리며 스텐실 테스트에 대한 프로그램을 구현해 봅시다.

9.2 스텐실 테스트 순서

9.2.1. 스텐실 버퍼의 생성, Clear

D3D에서 스텐실 테스트가 가능하기 위해서는 먼저 스텐실 버퍼가 있어야 하는데 이 버퍼는 디바이스의 생성 단계에서 만들어야 합니다.

D3DPRESENT_PARAMETERS 구조체의 변수 EnableAutoDepthStencil 과 AutoDepthStencilFormat 변수 값을 각각 다음과 같이 설정한 다음 D3D의 CreateDevice() 함수를 호출합니다.

```
D3DPRESENT_PARAMETERS d3dpp={0};

d3dpp.EnableAutoDepthStencil= TRUE;
d3dpp.AutoDepthStencilFormat= D3DFMT_D24S8;
…
pD3D->CreateDevice(…&d3dpp, &pDevice);
```

AutoDepthStencilFormat 은 D3DFMT_D24S8, D3DFMT_D24X4S4, D3DFMT_D15S1 중에 하나가 선택되어야 합니다. 가장 많이 선택하는 포맷은 D3DFMT_D24S8 입니다.
이렇게 스텐실 버퍼를 만들고 프로그램이 루프를 돌 때마다 디바이스를 Clear() 함수로 초기화 부분에 스텐실 버퍼도 같이합니다. Clear() 함수의 마지막 인수는 스텐실 버퍼를 기본적으로 채우는 값입니다.

```
m_pd3dDevice->Clear(…, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER|D3DCLEAR_STENCIL, …, 0L);
```

D3D는 후면 버퍼 각각의 버퍼를 각각 Clear 할 수 있으므로 다음과 같이 색상 버퍼와 깊이 버퍼는 이전 코드로 그대로 두고 스텐실 버퍼만 따로 Clear 해도 됩니다.

```
m_pd3dDevice->Clear(…, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER, …);
m_pd3dDevice->Clear(…, D3DCLEAR_STENCIL, …, 0L);
```

9.2.2. 스텐실 버퍼 이용과 참조

스텐실 테스트는 알파 테스트, 깊이 테스트를 통과한 픽셀에 마스킹을 스텐실 버퍼와 새로운 픽셀에 대한 참조 값에 적용해서 앞서 알파 또는 깊이 테스트에서 사용한 비교 함수를 통해서 새로운 픽셀의 RGBA는 색상 버퍼에, 깊이 값은 깊이 버퍼에 저장할 것이지 테스트 하는 것입니다. 스텐실 테스트는 다음과 같은 형식으로 합니다.

스텐실 테스트 = (참조 값 & 마스킹 값) 비교 함수 (스텐실 버퍼 값 & 마스킹 값)

여기서 참조 값은 알파 테스트처럼 프로그램에서 상태 함수에 설정해야 하는 값입니다. 마스킹 값

은 다른 테스트에는 없는 값으로 해당 스텐실의 값을 골라 내는 역할을 하며 비트 연산을 합니다. D3D는 기본적으로 스텐실 테스트를 안 합니다. 따라서 먼저 테스트 요청을 해야 합니다. 그 다음에 참조 값, 마스킹 값, 비교 함수를 디바이스의 상태 머신에 설정합니다.

```
pDevice->SetRenderState( D3DRS_STENCILENABLE, TRUE);          // 스텐실 테스트 활성화
pDevice->SetRenderState( D3DRS_STENCILREF, 0x1 );             // 참조 값 설정
pDevice->SetRenderState( D3DRS_STENCILMASK, 0x0000FFFF);      // 마스킹 값 설정
pDevice->SetRenderState( D3DRS_STENCILFUNC, D3DCMP_NOTEQUAL); // 비교 함수 설정

pDevice->DrawPrimitiveUP(…);
```

이 코드를 해석하면 다음과 같이 됩니다.

```
Test = (0x1 & 0x0000FFFF) != ("스텐실 버퍼 값" & 0x0000FFFF)
```

이 테스트의 결과는 스텐실 버퍼의 값이 0x1 이 아닌 모든 스텐실 버퍼가 됩니다.
참고로 스텐실 버퍼는 D3DFMT_D24S8 포맷에서 [0, 255] 값의 범위인데 마스킹 값은 32비트 입니다. 이것은 D3D의 상태 값들은 기본적으로 DWORD 형이기 때문입니다. 스텐실 마스킹의 디폴트 값은 0XFFFFFFFF 으로 되어 있습니다. 참조 값은 0x0 입니다.

비교 함수는 NEVER(항상 실패), ALWAYS(항상 성공), LESS('<'), EQUAL('='), GREATER('>'), LESSEQUAL(≤), GREATEREQUAL(≥) 등 다른 테스트와 동일하고 디폴트는 D3DCMP_ALWAYS 입니다.

9.2.3. 테스트 후 스텐실 버퍼 처리(Stencil Operation)

스텐실 테스트는 다른 테스트와 다르게 테스트 결과에 따라 스텐실 버퍼의 내용을 변경할 수 있습니다. D3D는 머신의 상태를 스텐실 테스트 실패 때(D3DRS_STENCILFAIL), 스텐실 테스트는 성공했지만 깊이 테스트는 실패 했을 때(D3DRS_STENCILZFAIL), 스텐실 깊이 테스트 모두 성공 했을 때(D3DRS_STENCILPASS) 3종류로 나누고 스텐실 버퍼를 처리(Operation)합니다.

스텐실 버퍼 처리에 대한 방법은 _D3DSTENCILOP 에 다음과 같이 선언 되어 있습니다.

```
typedef enum _D3DSTENCILOP {
    D3DSTENCILOP_KEEP           = 1,
    D3DSTENCILOP_ZERO           = 2,
    D3DSTENCILOP_REPLACE        = 3,
    D3DSTENCILOP_INCRSAT        = 4,
```

```
    D3DSTENCILOP_DECRSAT        = 5,
    D3DSTENCILOP_INVERT         = 6,
    D3DSTENCILOP_INCR           = 7,
    D3DSTENCILOP_DECR           = 8,
} D3DSTENCILOP;
```

KEEP은 스텐실 버퍼의 값을 현재 값으로 유지합니다. ZERO는 0으로 설정합니다. REPLACE는 참조 값으로 변경합니다. INCRSAT는 값을 1 증가 시키되 하드웨어에서 지원되는 최대값에 도달하면 값을 안 올립니다. DECRSAT는 값을 1 감소 시키되 0까지 감소 합니다. INCRSAT, DECRSAT는 필터링의 클램프와 비슷합니다. INVERT는 비트 반전(0→1, 1→0)입니다. INCR은 값을 1증가 합니다. 최대값이 되면 0으로 됩니다. DECR는 값을 1 감소 합니다. 0보다 작으면 최대 값이 됩니다. INCR, DECR은 필터링의 Wrap과 비슷합니다.

D3D의 스텐실 버퍼 처리(Operation) FAIL, ZFAIL, PASS의 디폴트 값은 KEEP입니다.
D3D는 스텐실 처리를 하기 위해 쓰기용 마스킹 값을 설정해야 합니다.

```
pDevice->SetRenderState(D3DRS_STENCILWRITEMASK, "마스킹 값");
```

"D3DRS_STENCILMASK"는 읽기용 마스크이며, "D3DRS_STENCILMASK"는 쓰기용 마스크 상태 설정입니다. 쓰기용 마스크 값의 디폴트는 읽기와 마찬가지로 0xFFFFFFFF 입니다. 다음은 스텐실 버퍼 테스트 후에 스텐실 처리 코드 예입니다.

```
m_pDev->SetRenderState(D3DRS_STENCILWRITEMASK, 0xffffffff);
m_pDev->SetRenderState(D3DRS_STENCILZFAIL, D3DSTENCILOP_KEEP);
m_pDev->SetRenderState(D3DRS_STENCILFAIL,  D3DSTENCILOP_KEEP);
m_pDev->SetRenderState(D3DRS_STENCILPASS,  D3DSTENCILOP_KEEP);
```

스텐실의 테스트부터 스텐실 버퍼 처리까지의 코드를 모으면 다음과 같이 8줄을 기본 세트로 만들 수 있습니다.

```
pDevice->SetRenderState( D3DRS_STENCILENABLE, TRUE);
pDevice->SetRenderState( D3DRS_STENCILREF, "참조 값" );
pDevice->SetRenderState( D3DRS_STENCILMASK, "읽기 마스킹 값");
pDevice->SetRenderState( D3DRS_STENCILFUNC, "비교 함수");
pDevice->SetRenderState(D3DRS_STENCILWRITEMASK, "쓰기 마스킹 값");
pDevice->SetRenderState(D3DRS_STENCILFAIL,  "스텐실 실패 때 처리");
pDevice->SetRenderState(D3DRS_STENCILZFAIL, "스텐실 성공, 깊이 실패 때 처리");
```

```
pDevice->SetRenderState(D3DRS_STENCILPASS, "스텐실, 깊이 모두 성공했을 때 처리");
```

9.3 스텐실 테스트 연습

9.3.1. 스텐실 연습

다음의 왼쪽 그림은 스텐실 버퍼 사용의 가장 기초적인 예입니다. 이 그림처럼 만들기 위해서 오른쪽 그림처럼 화면의 스텐실 값을 설정해야 합니다.

<m3d_08_4stencil01_basic.zip>

프로그램은 스텐실을 설정은 참조 값을 0인 것부터 화면에 출력 합니다. 다음으로 참조 값을 1로 한 다음 오른쪽의 사각형을 그립니다. 마지막에 호랑이를 그릴 때는 마스킹과 비교 함수로 스텐실 값이 1로 저장되어 있는 버퍼의 픽셀만 다시 씁니다.

이것을 의사 코드(Pseudo -code)로 표현하면 다음과 같습니다.

1. 화면을 Clear하면 스텐실 버퍼의 값은 0으로 초기화 하므로 스텐실 설정 없이 렌더링 한다.
2. 오른쪽 사각형으로 인해 렌더링 후 스텐실 버퍼의 값을 1로 만들기 위해 테스트는 무조건 통과하도록 하고 스텐실 값을 참조 값으로 대치한다.

```
pDevice->SetRenderState(StencilRef, 1);
pDevice->SetRenderState(StencilCompareFunction, Always);
pDevice->SetRenderState(StencilPass, Replace)
```

3. 호랑이를 그릴 때 스텐실 버퍼가 1일 때만 그리게 한다.

```
pDevice->SetRenderState(StencilRef, 1);
pDevice->SetRenderState(StencilCompareFunction, Equal);
```

이것이 내용을 m3d_08_4stencil01_basic.zip의 CMcScene::Render() 함수에 다음과 같이 구현 되어 있습니다.

```
// 1. 왼쪽의 벽을 그린다.
…
// 2. 스텐실 적용을 위한 오른쪽 벽을 그린다.
m_pDev->SetRenderState(D3DRS_STENCILENABLE,    TRUE);
m_pDev->SetRenderState(D3DRS_STENCILFUNC,      D3DCMP_ALWAYS);
m_pDev->SetRenderState(D3DRS_STENCILREF,       0x1);
m_pDev->SetRenderState(D3DRS_STENCILMASK,      0xffffffff);
m_pDev->SetRenderState(D3DRS_STENCILWRITEMASK, 0xffffffff);
m_pDev->SetRenderState(D3DRS_STENCILZFAIL,     D3DSTENCILOP_KEEP);
m_pDev->SetRenderState(D3DRS_STENCILFAIL,      D3DSTENCILOP_KEEP);
m_pDev->SetRenderState(D3DRS_STENCILPASS,      D3DSTENCILOP_REPLACE);

m_pDev->SetTexture(0, m_pTxWall);
m_pDev->SetFVF(VtxUV1::FVF);
m_pDev->DrawPrimitiveUP(D3DPT_TRIANGLEFAN, 2, m_pVtxMirr, sizeof(VtxUV1));
m_pDev->SetRenderState(D3DRS_STENCILPASS,   D3DSTENCILOP_KEEP); // Default 값으로 재설정

// 3. 호랑이의 스텐실 설정
m_pDev->SetRenderState(D3DRS_STENCILENABLE,    TRUE);
m_pDev->SetRenderState(D3DRS_STENCILFUNC,       D3DCMP_EQUAL);
m_pDev->SetRenderState(D3DRS_STENCILREF,       0x1);
m_pDev->SetRenderState(D3DRS_STENCILMASK,      0xffffffff);
m_pDev->SetRenderState(D3DRS_STENCILWRITEMASK, 0xffffffff);
m_pDev->SetRenderState(D3DRS_STENCILZFAIL, D3DSTENCILOP_KEEP);
m_pDev->SetRenderState(D3DRS_STENCILFAIL,  D3DSTENCILOP_KEEP);
m_pDev->SetRenderState(D3DRS_STENCILPASS,  D3DSTENCILOP_KEEP);

// 호랑이를 그린다.
…
m_pDev->SetFVF(VtxUV1::FVF);
m_pDev->DrawIndexedPrimitiveUP(D3DPT_TRIANGLELIST, …);
```

오른 쪽 벽을 그릴 때 무조건 버퍼를 갱신하기 위해서 비교 함수를 ALWAYS로 하고 깊이 스텐실 모두 통과 됐을 때 참조 값 1로 대치 하는 REPLACE로 한 점이 중요합니다. 그리고 다시 스텐실 처리

(Operation)을 디폴트 값인 KEEP으로 설정 했습니다.
호랑이를 그릴 때는 참조 값을 1, 비교 함수를 EQUAL로 해서 오른쪽 사각형에 의해 만들어진 스텐실 버퍼 값1에만 후면 버퍼의 픽셀을 갱신하도록 합니다.

9.3.2. 반사 효과

앞의 예제를 조금만 수정하면 다음과 같은 거울 반사 효과를 만들어 낼 수 있습니다.


<m3d_08_4stencil02_mirror.zip>

위 그림처럼 만들기 위해서 앞의 커다란 호랑이와 왼쪽의 사각형을 먼저 그립니다. 다음으로 스텐실을 적용해서 오른쪽 사각형을 먼저 그리고 반사된 호랑이를 그립니다.

반사된 호랑이를 그리기 위해서 DXSDK의 D3DXMatrixReflect() 함수를 이용해서 반사 행렬을 구합니다. 호랑이는 모델 좌표계를 사용하므로 거울의 법선과 반대 방향으로 호랑이를 이동 시켜야 합니다. 가장 중요한 것은 반사가 되면 삼각형을 그리는 순서가 반대로 된다는 점입니다. 따라서 Culling Mode가 CCW로 되어 있다면 반사된 호랑이는 뒷면이 출력될 것이고 Culling Mode를 CW로 고쳐야만 제대로 출력이 됩니다.

반사 행렬을 만드는 중요 부분은 다음과 같습니다.

```
D3DXMATRIX mtTrn; D3DXMATRIX mtRfc;
D3DXPLANE plane(0.0f, 0.0f, -1.0f, 0.0f); // xy plane
D3DXMatrixReflect(&mtRfc, &plane);
D3DXMatrixTranslation(&mtTrn, m_vcTiger.x, m_vcTiger.y+12, -m_vcTiger.z);

mtWld = mtRot * mtRfc * mtTrn;
```

```
// 반사 행렬에 의해 CW에서 CCW가 컬링 된다. 이것을 바꾸어야 한다.
m_pDev->SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);

// 호랑이를 그린다.
m_pDev->SetTransform(D3DTS_WORLD, &mtWld);
m_pDev->DrawIndexedPrimitive(…);
```

행렬의 곱(mtRot * mtRfc * mtTrn 부분)이 Rotation * Replection * Translation임을 주의하십시오. 이것은 호랑이가 지역 좌표계를 사용하기 때문에 먼저 회전을 적용한 것이고 회전이 적용된 결과를 반사, 이동 시키기 위해서 입니다.

9.3.3. 평면 그림자 1

평면 그림자 만들기는 스텐실 테스트 예제와 거의 같습니다. 단지 호랑이 정점을 이용해 그림자로 만들기 위해서 거울 예제처럼 행렬을 이용하고, 알파 블렌딩을 하는 점만 다를 뿐입니다.

<평면 그림자: m3d_08_4stencil03_plan.zip>

거울 예제에서 D3DXMatrixReflect() 함수로 정점의 위치를 반사면 반대로 이동시켰습니다. 평면 그림자는 D3DXMatrixShadow()함수를 월드 행렬에 적용해서 정점들을 납작한 평면에 모두 그리게 합니다.

m3d_08_4stencil03_plan.zip의 CMcScene::Render() 함수 안에 납작한 호랑이를 그릴 때 다음과 같은 행렬 설정을 볼 수 있습니다.

```
D3DXVECTOR4 vcLight(-2.8f, -1.f, 0.f, 0.f);
D3DXPLANE   dxPlane(  0.f, -1.f, 0.f, 0.f);

D3DXMATRIX mtShd;
D3DXMATRIX mtTrn;
```

```
D3DXVec4Normalize(&vcLight, &vcLight);
D3DXMatrixShadow(&mtShd, &vcLight, &dxPlane);
D3DXMatrixTranslation(&mtTrn, m_vcTiger.x, m_vcTiger.y, m_vcTiger.z);

mtWld = mtShd * mtRot * mtTrn;
```

D3DXMatrixShadow() 함수는 평면의 방정식과 빛의 방향 벡터를 주어야 합니다. 여기서 빛의 방향 벡터는 정규화 하고 평면의 방정식(ax + by + cz + d =0)에 대한 법선(a, b, c)의 방향을 (0, -1, 0) , d=0 으로 평면 방정식을 만들었습니다.
이 행렬을 호랑이에 적용할 때 호랑이에 정의된 색상을 버리고 디바이스에 설정된 색상 값을 그림자 색상으로 정한 다음 렌더링 합니다. 따라서 알파 블렌딩을 하면 그림자 색상과 배경 색상이 혼합 됩니다.

```
// 알파 블렌딩 활성화
m_pDev->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
m_pDev->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
m_pDev->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);

// 디바이스에 TFACTOR 색상을 정하고 이것을 호랑이 색상으로 설정한다.
m_pDev->SetRenderState(D3DRS_TEXTUREFACTOR, D3DXCOLOR(0.3F, 0.3F, 0.3F, 0.3F));
m_pDev->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_TFACTOR );
m_pDev->SetTextureStageState( 0, D3DTSS_COLOROP,   D3DTOP_SELECTARG2 );

// 호랑이를 그린다.
m_pDev->SetTransform(D3DTS_WORLD, &mtWld);
m_pDev->SetFVF(VtxUV1::FVF);
m_pDev->SetTexture(0, NULL);
m_pDev->DrawIndexedPrimitiveUP(…);
```

DXSDK에 있는 호랑이 폴리곤은 호랑이의 중심에서 모델링 되어 있습니다. 이것을 그대로 사용하면 그림자와 모델의 렌더링 위치가 잘 안 맞습니다. m3d_08_4stencil03_plan.zip의 CMcScene::Create(LPDIRECT3DDEVICE9 pDev) 함수에는 호랑이 모델의 정점을 시스템 메모리에 복사하고 다음과 같이 y 값을 수정해 사용했습니다.

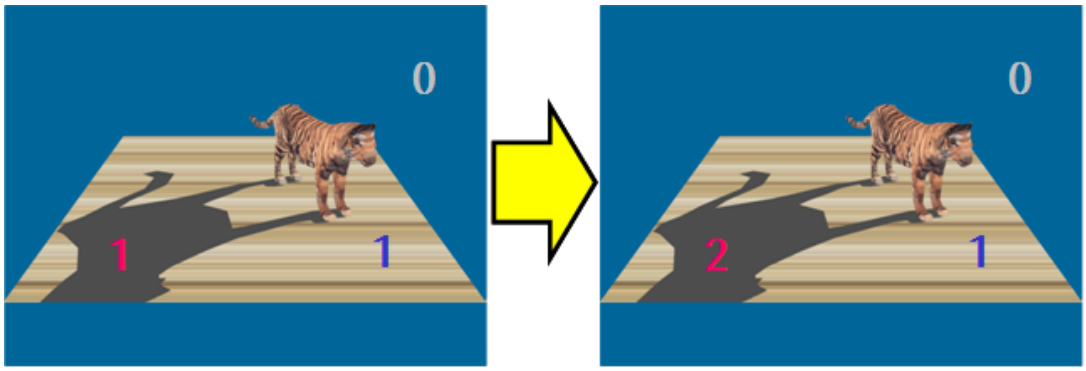
```
float fSCale=10; float  fMin=100000;
for(i=0; i<m_nVtx; ++i){
        m_pVtx[i].p *= fSCale;
```

```
        if(     m_pVtx[i].p.y<fMin)
                fMin= m_pVtx[i].p.y;
}
fMin = -fMin;
for(i=0; i<m_nVtx; ++i){
        m_pVtx[i].p.y +=fMin;
}
```

9.3.3. 평면 그림자 2

방금 보인 평면 그림자 1의 내용은 호랑이를 납작하게 만드는 것까지는 좋았습니다. 그런데 막상 실행 보면 그림자 색상과 배경 색상의 혼합이 잘 안 된다는 것을 볼 수 있습니다. 알파 블렌딩 연산 방법, 블렌딩 인수 등을 바꾸어도 위의 그림자 보다 약간 좋을 뿐 노력한 만큼 확실한 품질을 만들기 어렵습니다.

여기에 생각의 전환이 필요합니다. 그림자는 평면에 의해 만들어진 스텐실 버퍼 값과 비교해서 그립니다. 이것을 바꾸어서 그림자를 그리지 않고 스텐실 값을 한번 더 올리는 것입니다.
이렇게 하면 그림자 영역은 다음 그림처럼 스텐실의 값이 1에서 2로 증가 합니다. 이 증가한 값 2에 대해서 화면 전체를 그립니다.



화면 전체를 렌더링 하는 것은 그리 큰 부담은 아닙니다. 전체를 그릴 때 RHW로 정점의 색상에 알파를 넣어서 그리면 다음과 같은 장면을 만들 수 있습니다.

앞서 만든 예제에서 코드를 조금 수정해야 합니다. 그 부분은 스텐실 값을 2로 만드는 그림자 영역에는 스텐실 버퍼 값만 올리고 색상을 적용하지 않는 것입니다. 이전 알파 블렌딩에서 색상을 기록하지 않는 예제가 있었는데 다음과 같이 하면 색상 버퍼를 갱신 안 합니다.

```
m_pDev->SetRenderState(D3DRS_COLORWRITEENABLE, 0);
```

이것을 앞의 그림 스텐실 2를 기록할 때 사용합니다. 그리고 나서 전체 화면에 사각형을 다시 그리면 됩니다. 이점을 기억하고 구체적인 프로그램 내용을 봅시다.

이전 평면 그림자에 다음과 같은 정점 구조체를 만들고 이 정점을 설정합니다.

```
struct VtxDRHW
{
        D3DXVECTOR4     p;      // x,y,z, rhw
        DWORD           d;
        enum { FVF = (D3DFVF_XYZRHW|D3DFVF_DIFFUSE)};
};
```

디바이스에서 윈도우 핸들을 간접적으로 가져와서 위 구조체로 구성된 정점을 설정합니다. 이 부분은 CMcScene::Create() 함수의 마지막에 다음과 같이 구현 되어 있습니다.

```
D3DDEVICE_CREATION_PARAMETERS Parameters;
m_pDev->GetCreationParameters(&Parameters);
RECT rc;
GetClientRect(Parameters.hFocusWindow, &rc);
FLOAT   fScnW = FLOAT(rc.right - rc.left);
FLOAT   fScnH = FLOAT(rc.bottom- rc.top );
m_VtxShadow[0] = VtxDRHW(  0.f,     0, 0x880000FF);
m_VtxShadow[1] = VtxDRHW(fScnW,     0, 0x88FF0000);
m_VtxShadow[2] = VtxDRHW(fScnW, fScnH, 0x8800FF00);
m_VtxShadow[3] = VtxDRHW(  0.f, fScnH, 0x88FF00FF);
```

호랑이를 그림자 행렬을 적용해 그릴 때 스텐실 테스트 통과하면 값을 1증가시키도록 다음과 같은 코드를 넣습니다.

```
m_pDev->SetRenderState(D3DRS_STENCILENABLE,    TRUE);
m_pDev->SetRenderState(D3DRS_STENCILFUNC,      D3DCMP_EQUAL);
m_pDev->SetRenderState(D3DRS_STENCILREF,       0x1);
...
m_pDev->SetRenderState(D3DRS_STENCILPASS,      D3DSTENCILOP_INCRSAT);
```

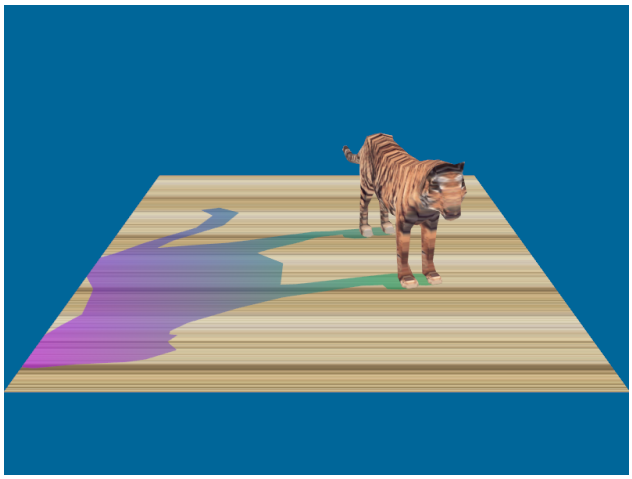
또한 호랑이를 그릴 때 색상 버퍼의 갱신을 막습니다. 깊이 버퍼도 안 씁니다.

```
m_pDev->SetRenderState(D3DRS_COLORWRITEENABLE, 0);
m_pDev->SetRenderState(D3DRS_ZWRITEENABLE, FALSE);      // 깊이도 안 쓴다.
…
m_pDev->DrawIndexedPrimitiveUP(…);
```

마지막에 화면 전체를 그립니다. 이 때 참조가 2와 같은 스텐실 버퍼만 찾아서 그립니다.

```
m_pDev->SetRenderState(D3DRS_STENCILENABLE,    TRUE);
m_pDev->SetRenderState(D3DRS_STENCILFUNC,      D3DCMP_EQUAL);
m_pDev->SetRenderState(D3DRS_STENCILREF,       0x2);
…
m_pDev->SetFVF(VtxDRHW::FVF);
m_pDev->DrawPrimitiveUP(D3DPT_TRIANGLEFAN, 2, m_VtxShadow, sizeof(VtxDRHW));
```

제공된 예제에서 알파 블렌딩은 선택사항입니다. 만약 알파 블렌딩을 적용하면 다음 그림과 같은 화면을 얻을 수 있습니다.



<향상된 그림자: m3d_08_4stencil04_screen.zip>

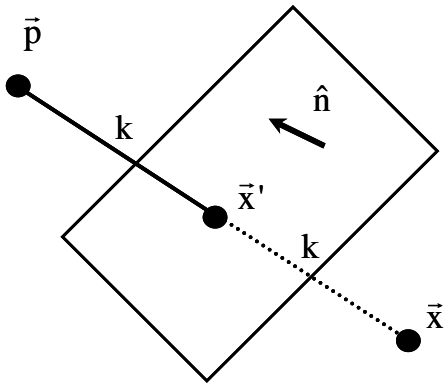
이 예제는 정점의 RHW 로 구성된 정점의 색상이 제대로 표현되는지 RED, Green, Blue, Magenta 색을 이용했습니다.

9.3.4. 반사 행렬, 그림자 행렬 구하기

이 부분은 반사 행렬과 그림자 행렬을 직접 구해보는 내용입니다. 넘어가도 상관 없습니다.

- 반사 행렬

반사 행렬은 다음그림과 같이 정점을 평면 $(\hat{n}, d)$의 법선의 반대 방향으로 이동하는 것입니다.
이것을 다음 그림을 가지고 반대 방향 $\vec{x}$에 대해서 다음과 같은 수식으로 구할 수 있습니다.



$$\vec{x} = \vec{p} + k\vec{L}$$
$$\hat{n} \cdot \vec{x}' + d = 0$$
$$\hat{n} \cdot \vec{p} = \hat{n} \cdot (\vec{x}' + k\hat{n})$$
$$= -d + k$$
$$k = \hat{n} \cdot \vec{p} + d$$

$$\vec{x} = \vec{p} - 2k\hat{n}$$
$$= \vec{p} - 2(\hat{n} \cdot \vec{p} + d)\hat{n}$$
$$= \vec{p} \cdot (\vec{I} - 2\hat{n}\hat{n}) - 2d\hat{n}$$

$$\vec{x} = \vec{p} \cdot \vec{M}_{Reflect}$$

$$\therefore \vec{M}_{Reflect} = \begin{pmatrix} 1-2n_x n_x & -n_x n_y & -n_x n_z & 0 \\ -2n_y n_x & 1-2n_y n_y & -n_y n_z & 0 \\ -2n_z n_x & -n_z n_y & 1-2n_z n_z & 0 \\ -2dn_x & -2dn_y & -2dL_z & 1 \end{pmatrix}$$

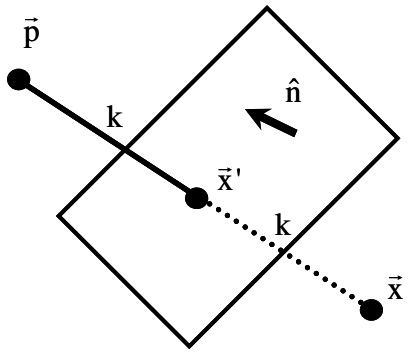
이 결과를 다음과 같은 코드로 만듭니다.

```
void LcMatrixReflect(void* pOutMatrix, const void* Plane, BOOL bNormalized=FALSE)
{
        D3DXPLANE P((FLOAT*)Plane);     D3DXMATRIX mtS;
        if(FALSE == bNormalized)        // Normalize Plane
                D3DXPlaneNormalize(&P, &P);

        mtS  = D3DXMATRIX       // Setup Output Value
        (
        1 - 2 * P.a * P.a,   - 2 * P.a * P.b,   - 2 * P.a * P.c,   - 2 * P.a * P.d,
          - 2 * P.b * P.a, 1 - 2 * P.b * P.b,   - 2 * P.b * P.c,   - 2 * P.b * P.d,
          - 2 * P.c * P.a,   - 2 * P.c * P.b, 1 - 2 * P.c * P.c,   - 2 * P.c * P.d,
          - 2 * P.d * P.a,   - 2 * P.d * P.b,   - 2 * P.d * P.c, 1 - 2 * P.d * P.d
        );
        *((D3DXMATRIX*)pOutMatrix) = mtS;       // Copy Output value
}
```

- 그림자 행렬

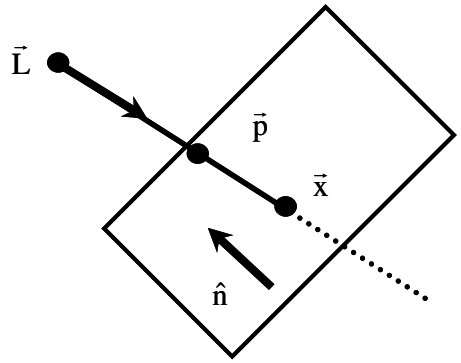
그림자 행렬은 한 점에서 오는 빛과 평행하게 오는 빛 두 가지로 계산한 다음 결합합니다.
평행한 빛은 다음처럼 간단히 구할 수 있습니다.



$$\vec{x} = \vec{p} + k\vec{L}$$
$$\hat{n}\cdot\vec{x} + d = 0$$
$$\hat{n}\cdot\vec{p} + k(\hat{n}\cdot\vec{L}) + d = 0$$
$$\therefore \vec{x} = \vec{p} - \frac{d + \hat{n}\cdot\vec{p}}{\hat{n}\cdot\vec{L}}\vec{L}$$
$$= \frac{\vec{p}\cdot((\hat{n}\cdot\vec{L})\vec{I} - \hat{n}\vec{L}) - d\vec{L}}{\hat{n}\cdot\vec{L}}$$
$$\vec{x} = \vec{p}\cdot\vec{\vec{M}}_{Shadow}$$

$$\therefore \vec{\vec{M}}_{Parallel} = \begin{pmatrix} \hat{n}\cdot\vec{L} - n_x L_x & -n_x L_y & -n_x L_z & 0 \\ -n_y L_x & \hat{n}\cdot\vec{L} - n_y L_y & -n_y L_z & 0 \\ -n_z L_x & -n_z L_y & \hat{n}\cdot\vec{L} - n_z L_z & 0 \\ -dL_x & -dL_y & -dL_z & \hat{n}\cdot\vec{L} \end{pmatrix}$$

한 점에서 오는 빛은 빛의 위치가 중요합니다. 이 빛의 위치를 이용해서 다음과 같이 구합니다.



$$\vec{x} = \vec{L} + k(\vec{p} - \vec{L})$$
$$\hat{n}\cdot\vec{x} + d = 0$$
$$\hat{n}\cdot\vec{L} + k(\hat{n}\cdot\vec{p} - \hat{n}\cdot\vec{L}) + d = 0$$
$$\therefore \vec{x} = \vec{L} + \frac{d + \hat{n}\cdot\vec{L}}{-\hat{n}\cdot(\vec{p} - \vec{L})}(\vec{p} - \vec{L})$$
$$= \frac{-\vec{L}(\hat{n}\cdot\vec{p}) + \vec{L}(\hat{n}\cdot\vec{L}) + (\hat{n}\cdot\vec{L})\vec{p} + d\vec{p} - \vec{L}(\hat{n}\cdot\vec{L}) - d\vec{L}}{-\hat{n}\cdot\vec{p} + \hat{n}\cdot\vec{L}}$$
$$= \frac{\vec{p}\cdot((d + \hat{n}\cdot\vec{L})\vec{I} - \hat{n}\vec{L}) - d\vec{L}}{-\hat{n}\cdot\vec{p} + \hat{n}\cdot\vec{L}}$$
$$\vec{x} = \vec{p}\cdot\vec{\vec{M}}_{Shadow}$$

$$\therefore \vec{\vec{M}}_{Spot} = \begin{pmatrix} d + \hat{n}\cdot\vec{L} - n_x L_x & -n_x L_y & -n_x L_z & -n_x \\ -n_y L_x & d + \hat{n}\cdot\vec{L} - n_y L_y & -n_y L_z & -n_y \\ -n_z L_x & -n_z L_y & d + \hat{n}\cdot\vec{L} - n_z L_z & -n_z \\ -dL_x & -dL_y & -dL_z & \hat{n}\cdot\vec{L} \end{pmatrix}$$

평행한 빛, 한 점에서 오는 빛을 합치기 위해서 수식을 다시 수정하면 다음과 같이 됩니다.

$$\vec{P}(a,b,c,d) = (n_x, n_y, n_z, d)$$
$$\vec{L} = (L_x, L_y, L_z, L_w)$$
$$d + \hat{n}\cdot\vec{L} = \vec{P}\cdot\vec{L} = D$$
$$\hat{n}\cdot\vec{L} = D - P_d L_w$$

$$\vec{M}_{Shadow} = \begin{pmatrix} D-P_aL_x & -P_aL_y & -P_aL_z & -P_aL_w \\ -P_bL_x & D-P_bL_y & -P_bL_z & -P_bL_w \\ -P_cL_x & -P_cL_y & D-P_cL_z & -P_cL_w \\ -P_dL_x & -P_dL_y & -P_dL_z & D-P_dL_w \end{pmatrix}$$
$$\text{or } \vec{M}_{Shadow} = D\vec{I} - \vec{P}\vec{L}$$

이 것을 다음과 같은 함수로 만들 수 있습니다.

```
void LcMatrixShadow(void* pOutMatrix, const void* Light, void* Plane
                , BOOL bNormalized=FALSE)
{
        D3DXPLANE   P((FLOAT*)Plane);
        D3DXVECTOR4 L((FLOAT*)Light);
        D3DXMATRIX  mtS;
        FLOAT D =0;

        if(FALSE == bNormalized)        // Normalize Plane
                D3DXPlaneNormalize(&P, &P);

        D = D3DXVec4Dot((D3DXVECTOR4*)&P, (D3DXVECTOR4*)&L);    // Dot(Plane, Light)
        mtS  = D3DXMATRIX       // Setup Output Value
        (
                D - P.a * L.x,    - P.a * L.y,    - P.a * L.z,    - P.a * L.w,
                  - P.b * L.x,  D - P.b * L.y,    - P.b * L.z,    - P.b * L.w,
                  - P.c * L.x,    - P.c * L.y,  D - P.c * L.z,    - P.c * L.w,
                  - P.d * L.x,    - P.d * L.y,    - P.d * L.z,  D - P.d * L.w
        );

        // Copy Output value
        *((D3DXMATRIX*)pOutMatrix) = mtS;
}
```

DXSDK 도움말을 보면 D3DXMatrixShadow() 함수 구현에 대한 설명이 있습니다. 설명대로 함수 만들면 잘 안됩니다.

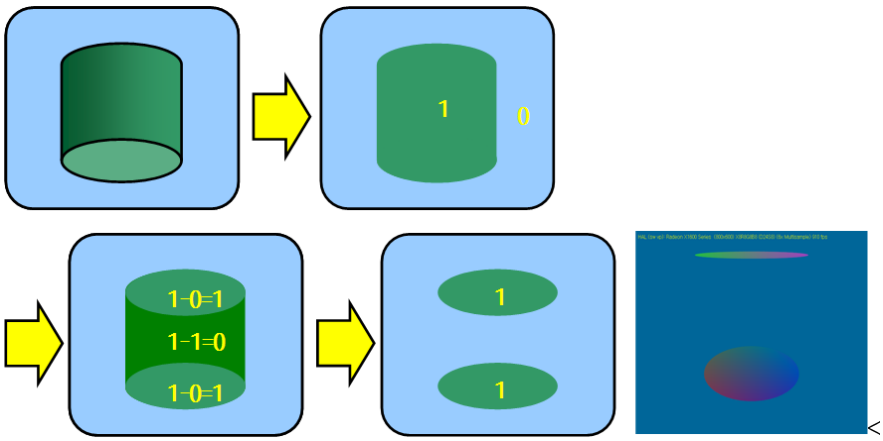
9.4 체적 그림자(Volumetric Shadow)

이 부분은 이론 보다는 응용이라서 3D를 처음 접하는 분들은 지금은 넘어가도 됩니다.

3D 게임에서 그림자를 표현 기법은 여러 가지가 있습니다. 그 중에서 전통적으로 가장 많이 사용되는 방식이 그림자 체적(Volume)을 만드는 체적 그림자며 이 방식은 스텐실 버퍼를 이용합니다. 일단 그림자 체적(Shadow Volume)을 만드는 방법은 잠시 뒤로 미루고 먼저 체적 그림자 어떻게 만드는지 살펴 봅시다.

9.4.1 스텐실을 이용한 그림자 표현 기초

체적 그림자의 원리는 의외로 간단 합니다. 다음 그림을 보면 앞면을 그릴 때 스텐실을 증가시키고, 반대로 뒷면을 그릴 때는 스텐실을 감소 시킵니다. 그리고 마지막에 화면 전체를 그립니다.



< 스텐실 사용 그림자 기초>

이 내용을 코드로 구현 하면 목적이 스텐실 버퍼의 내용만 갱신하는 것이므로 깊이 버퍼와 색상버퍼를 쓰지 않도록 디바이스의 상태를 설정 합니다.

```
m_pDev->SetRenderState( D3DRS_ZWRITEENABLE,  FALSE );
m_pDev->SetRenderState( D3DRS_COLORWRITEENABLE, FALSE);
```

다음으로 원통의 앞면을 그리기 위한 스텐실 관련 상태를 준비합니다. 앞면은 무조건 1이 되어야 하므로 스텐실 테스트를 항상 통과하도록 ALWAYS로 합니다. 물론 스텐실 테스트는 활성화 해야 합니다.

```
m_pDev->SetRenderState( D3DRS_STENCILENABLE, TRUE );          // 스텐실 활성화
m_pDev->SetRenderState( D3DRS_STENCILREF, 0x1 );
```

```
m_pDev->SetRenderState( D3DRS_STENCILMASK, 0xffffffff );
m_pDev->SetRenderState( D3DRS_STENCILWRITEMASK, 0xffffffff );
m_pDev->SetRenderState( D3DRS_STENCILFUNC, D3DCMP_ALWAYS );     // 테스트는 무조건 통과
```

테스트를 통과한-물론 항상 통과합니다- 경우는 스텐실 값을 1 증가 시킬 수 있도록 INCR 또는 INCRSAT로 설정합니다. 다른 테스트 결과는 만약 지형이 있는 경우도 고려한다면 ZFAIL에서는 아무것도 안해야 합니다. 따라서 테스트 실패의 경우 디폴트 상태(KEEP)로 유지합니다.

```
m_pDev->SetRenderState( D3DRS_STENCILZFAIL,     D3DSTENCILOP_KEEP);
m_pDev->SetRenderState( D3DRS_STENCILFAIL,      D3DSTENCILOP_KEEP);
m_pDev->SetRenderState( D3DRS_STENCILPASS,  D3DSTENCILOP_INCR); // 스텐실 값을 1 증가
```

은면 제거 모드를 CCW로 설정하고 원통의 앞면을 그립니다.

```
m_pDev->SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
…
m_pDev->DrawPrimitiveUP(D3DPT_TRIANGLESTRIP, 2*50-2, m_pVtx, sizeof(Vtx));
```

뒷면 또한 테스트를 항상 통과하도록 ALWAYS로 합니다. 그리고 테스트를 통과한 경우에는 스텐실 값을 1감소 하도록 DECR 또는 DECRSAT를 사용합니다.

```
m_pDev->SetRenderState( D3DRS_STENCILFUNC, D3DCMP_ALWAYS);
m_pDev->SetRenderState( D3DRS_STENCILZFAIL, D3DSTENCILOP_KEEP);
m_pDev->SetRenderState( D3DRS_STENCILFAIL, D3DSTENCILOP_KEEP);
m_pDev->SetRenderState( D3DRS_STENCILPASS, D3DSTENCILOP_DECR);  // 스텐실 값을 1 감소
```

은면 제거를 CW로 설정하면 뒷면을 그릴 수 있습니다. CW로 상태 머신을 설정하고 원통을 그립니다.

```
m_pDev->SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
…
m_pDev->DrawPrimitiveUP(D3DPT_TRIANGLESTRIP, 2*50-2, m_pVtx, sizeof(Vtx));
```

원통을 이용해 스텐실 버퍼의 내용을 바꾸었습니다. 스텐실 버퍼에 쓰여져 있는 내용을 화면에 출력하기 위해 색상 버퍼 쓰기와 깊이 버퍼 쓰기를 다시 활성화 합니다.

```
m_pDev->SetRenderState( D3DRS_COLORWRITEENABLE, 0xF);
```

```
m_pDev->SetRenderState( D3DRS_ZWRITEENABLE,     TRUE );
```

은면 제거 모드도 CCW로 다시 설정합니다.

```
m_pDev->SetRenderState( D3DRS_CULLMODE,  D3DCULL_CCW );
```

화면 전체를 RHW 정점들로 그립니다. 이 때 스텐실 버퍼에 의 값이 0x1보다 큰 값을 찾기 위해 LessEqual을 사용합니다. (0x1 ≤ 스텐실 버퍼 값)

```
m_pDev->SetRenderState( D3DRS_STENCILENABLE,    TRUE );
m_pDev->SetRenderState( D3DRS_STENCILREF,  0x1 );
m_pDev->SetRenderState( D3DRS_STENCILFUNC, D3DCMP_LESSEQUAL);   // 1 ≤ 스텐실 버퍼 값
m_pDev->SetRenderState( D3DRS_STENCILPASS, D3DSTENCILOP_KEEP );

...
m_pDev->DrawPrimitiveUP( D3DPT_TRIANGLESTRIP, 2, m_pRhw, sizeof(VtxRHWD) );
```

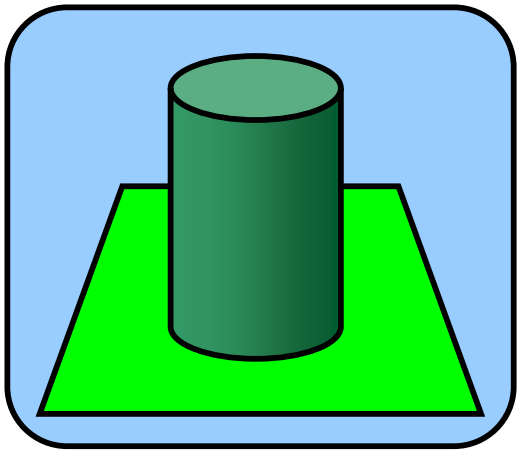
스텐실을 더 이상 이용하지 않으므로 비활성화 시킵니다.

```
m_pDev->SetRenderState( D3DRS_STENCILENABLE,    FALSE );
```

전체 코드는 m3d_08_5shd_v_00.zip을 참고 바랍니다.

9.4.2 그림자 연습

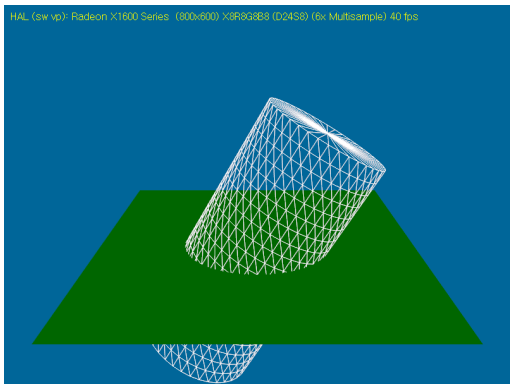
이것을 구현 했다면 그림자 체적 프로그램의 절반 정도를 했다고 봐도 됩니다. 몇 가지 더 손을 보자면 먼저 이 렌더링 오브젝트는 위 아래가 뚫려 있는 원통입니다. 실제 오브젝트는 위 아래 막혀 있는 것이 대부분 입니다. 만약 위 아래가 완전히 막혀 있는 오브젝트를 위와 같은 방식으로 렌더링 하면 양쪽 따 +1하고 -1하기 때문에 아무 것도 화면에 나오지 않습니다.

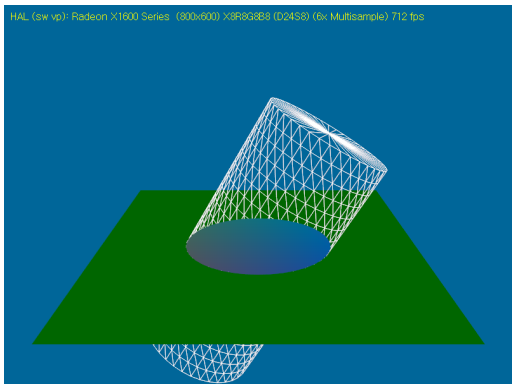


따라서 그림자를 제대로 만들고 있는지 확인 하기 위해 그림처럼 평면 하나를 더 추가해서 연습을 해야 합니다.

완전히 막힌 원통을 그래픽의 도움을 받으면 좋겠지만 현실적으로 어려우므로 여기서는 DXSDK의 확장 객체 Cylinder를 이용하겠습니다.

m3d_08_5shd_v_01.zip 을 컴파일 해서 실행하면 다음 왼쪽 그림처럼 초록색 배경 지형에 원통이 출력됩니다. 여기에 앞서 보인 스텐실 코드를 그대로 옮기면 오른쪽 그림과 같은 화면을 얻을 수 있습니다. 코드는 거의 같아 설명을 하지 않겠습니다. 코드는 m3d_08_5shd_v_02.zip의 CMcScene::Render를 참고 하기 바랍니다.





<스텐실 적용 전, 후>

위의 그림 오른쪽과 같은 화면을 얻을 수 있으면 나중에 그림자에 알파 블렌딩을 적용하면 됩니다. 앞면 뒷면 총 두 번을 렌더링 했습니다. 그런데 요즘은 스텐실에 대해서 양면을 동시에 처리할 수 있는 그래픽 카드들이 많이 있고 또한 양면을 동시에 처리하면 렌더링 파이프라인을 한 번만 거치게 되므로 속도에 이득이 있습니다.

양면(2-Side)를 처리하기 위해서 먼저 하드웨어가 지원하는 지 다음과 같이 확인해야 합니다.

```
D3DCAPS9        caps;
m_pDev->GetDeviceCaps(&caps);
m_bTwoSide = caps.StencilCaps & D3DSTENCILCAPS_TWOSIDED;
```

D3D는 양면 지원 모드의 디폴트가 FALSE로 되어있습니다 양면 지원의 되면 양면을 활성화 해야 합니다.

```
m_pDev->SetRenderState( D3DRS_TWOSIDEDSTENCILMODE, TRUE );
```

양면을 다 그리기 위해 은면 제거를 안 합니다.

```
m_pDev->SetRenderState( D3DRS_CULLMODE,  D3DCULL_NONE);
```

앞면 설정은 이전 코드와 동일 합니다.

```
m_pDev->SetRenderState( D3DRS_STENCILREF, 0x1 );
...
m_pDev->SetRenderState( D3DRS_STENCILFUNC, D3DCMP_ALWAYS );     // 테스트는 무조건 통과
...
m_pDev->SetRenderState( D3DRS_STENCILPASS,  D3DSTENCILOP_INCR); // 스텐실 값을 1 증가
```

뒷면 설정은 이전에 뒷면을 처리하는 부분에 D3DRS_CCW 로 다음과 같이 접두어만 바꾸면 됩니다.

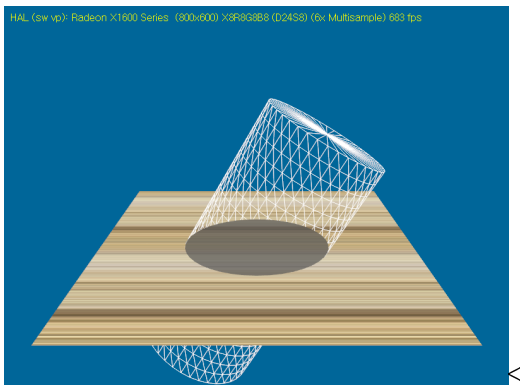
```
// 3. 뒷면 설정
m_pDev->SetRenderState( D3DRS_CCW_STENCILFUNC, D3DCMP_ALWAYS);
m_pDev->SetRenderState( D3DRS_CCW_STENCILZFAIL, D3DSTENCILOP_KEEP );
m_pDev->SetRenderState( D3DRS_CCW_STENCILFAIL,  D3DSTENCILOP_KEEP );
m_pDev->SetRenderState( D3DRS_CCW_STENCILPASS, D3DSTENCILOP_DECR );

// Object Rendering
m_pDev->SetTransform(D3DTS_WORLD, &m_mtObject);
m_pMesh->DrawSubset(0);
```

양면 지원에 대한 스텐실을 다 사용했으면 양면 지원을 해제합니다.

```
m_pDev->SetRenderState( D3DRS_TWOSIDEDSTENCILMODE, FALSE);
```

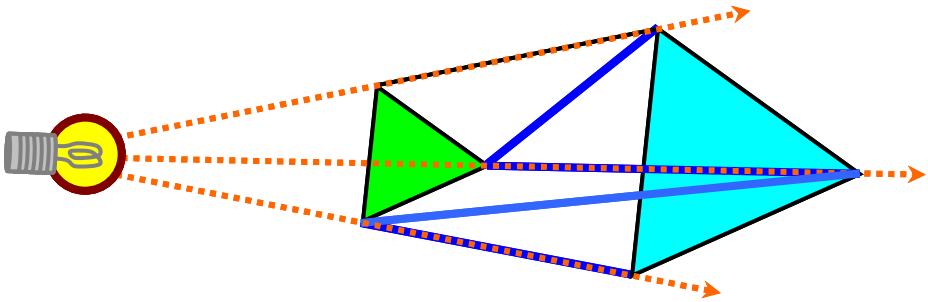
전체 코드는 m3d_08_5shd_v_02_ts.zip을 참고하기 바랍니다.



<양면 스텐실 테스트: m3d_08_5shd_v_02_ts.zip>

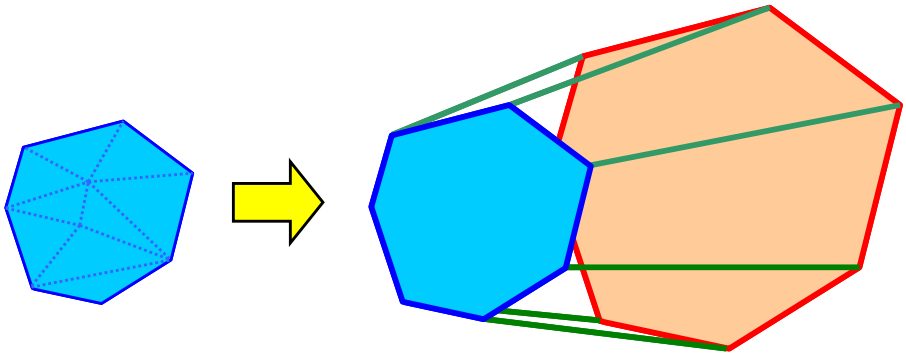
9.4.3 그림자 체적(Volumetric Shadow)

여기까지 완성이 되면 이제 남아 있는 것은 그림자 체적(Shadow Volume)을 만드는 일이 남아 있습니다. 그림자 체적은 다음 그림처럼 삼각형의 변을 늘려서 만들 수 있습니다.



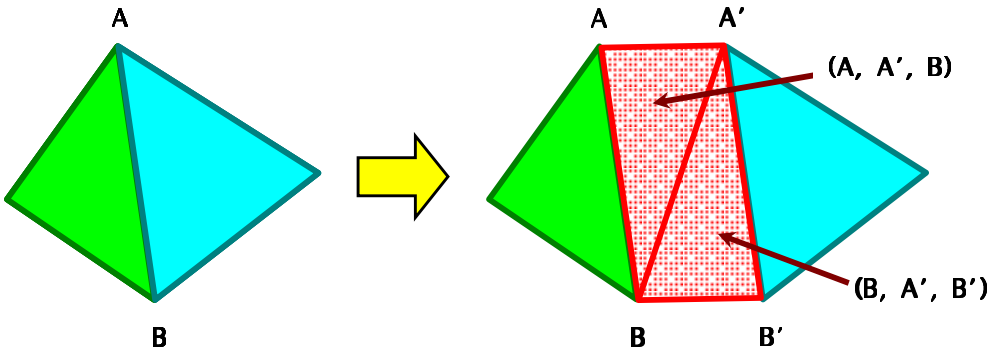
<간단한 그림자 체적 만들기>

만약 폴리곤이 여러 개로 구성된 렌더링 오브젝트의 그림자 체적에 이 방법이 적용되려면 그림자에 대한 외곽선을 찾아야 됩니다.



<외곽선 추출과 2개의 Cap>

외곽선을 구성하는 폴리곤을 2개로 구성하고 경계부분의 삼각형들은 다음 그림처럼 늘어난 부분의 사각형을 삼각형 2개로 만들어 줍니다.

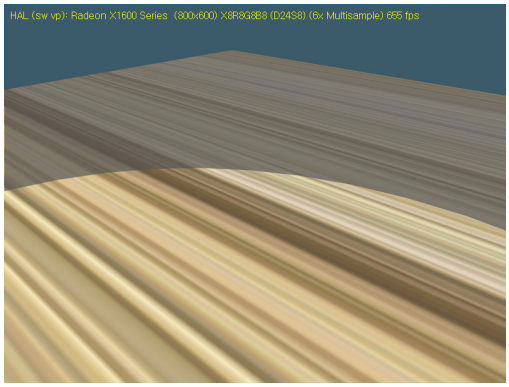


<경계 부분의 인접한 삼각형>

이렇게 외곽선을 만들고 그림자 체적을 만드는 방법을 "Silhouette Edge" 그림자 체적 만들기라 합니다.
이 방법은 DXSDK의 예제에 잘 소개되어 있습니다. 여기서는 코드의 구현은 넘어가고 DXSDK의 코드를 가지고 다음과 같이 일반 높이 지형에 구현된 것만 보여주는 것으로 끝내겠습니다.

<그림자 체적: m3d_08_5shd_v 03.zip>

9.4.4 Depth Fail



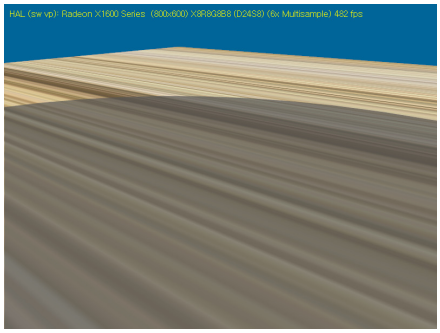
지금까지 설명한 그림자 방법은 그림자 안쪽 즉 그림자 체적 안으로 들어가면 옆의 그림처럼 그림자처리를 제대로 하지 못하는 경우가 발생합니다. 또한 오브젝트의 그림자가 여러 개 겹쳐도 이와 비슷한 문제가 발생합니다. 이것을 해결하는 방법중의 하나가 Z-검사가 실패했을 때만 그려주는 방법이 있습니다. Z-Failed(실패) 방법은 CW로 그려주는 삼각형에 대해서 스텐실 테스트는 통과했지만 Z-검사가 실패하는 경우 다음 코드처럼 스텐실 버퍼의 값을 1증가 하고 CCW로 그려주는 삼각형에 대해서는 반대로 1감소 합니다.

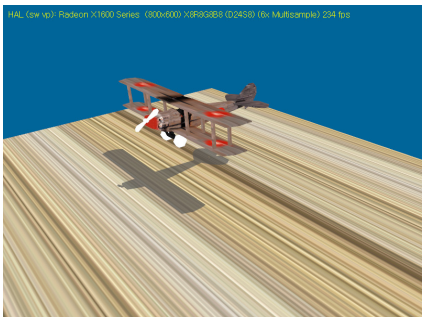
```
// 2-side 활성화
m_pDev->SetRenderState( D3DRS_TWOSIDEDSTENCILMODE, TRUE );
…
// 2. 앞면(CW로 그리는 삼각형) 설정
m_pDev->SetRenderState( D3DRS_STENCILFAIL, D3DSTENCILOP_KEEP );
m_pDev->SetRenderState( D3DRS_STENCILPASS,  D3DSTENCILOP_KEEP );
m_pDev->SetRenderState( D3DRS_STENCILZFAIL, D3DSTENCILOP_INCR);// Z-FAIL만 값을 1 증가

// 3. 뒷면(CCW로 그리는 삼각형) 설정
```

```
m_pDev->SetRenderState( D3DRS_CCW_STENCILFAIL, D3DSTENCILOP_KEEP );
m_pDev->SetRenderState( D3DRS_CCW_STENCILPASS, D3DSTENCILOP_KEEP );
m_pDev->SetRenderState( D3DRS_CCW_STENCILZFAIL,D3DSTENCILOP_DECR );// Z-FAIL만 값을 1 감소
// Object Rendering
...
```

이 코드는 이전의 코드와 거의 같으며 단지 STENCILPASS에서 값을 올리거나 내렸던 것을 ZFAILD 바꾼 것뿐입니다. m3d_08_6shd_d_01.zip을 실행하면 다음과 같은 화면을 볼 수 있습니다.





m3d_08_6shd_d_01.zip
m3d_08_6shd_d_02.zip
m3d_08_6shd_d_03.zip
<Z-failed에 의한 그림자 체적 렌더링>

과제) 게임에서 사용하는 그림자의 종류를 찾아 보시오.