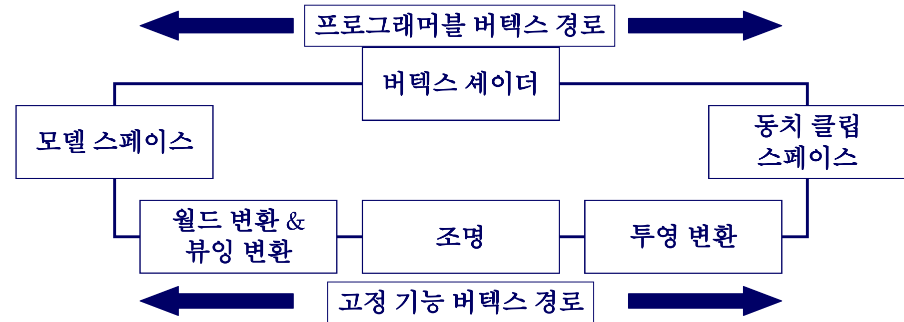
HLSL Vertex Shader

305890
2009년 봄학기
6/10/2009
박경신

버텍스 셰이더

- 그래픽 카드의 GPU에서 실행되는 프로그램
  - 하드웨어에서 지원하지 못하는 경우, 소프트웨어로 에뮬레이트 가능
- 고정 기능 파이프라인의 변환과 조명 단계를 대체



Overview

- 프로그래머블 파이프라인 내 버텍스 구조체의 요소를 정의하는 방법
- 버텍스 요소의 여러 가지 다른 이용법
- 버텍스 셰이더를 만들고 지정하고 제거하는 방법
- 버텍스 셰이더를 이용해 카툰 렌더링을 구현하는 방법

버텍스 선언

- 유연한 버텍스 포멧 (FVF)보다 훨씬 자세하고 강력함
- 버텍스 선언 기술하기

```
typedef struct _D3DVERTEXELEMENT9 {
   WORD Stream;
   WORD Offset;
   BYTE Type;
   BYTE Method;
   BYTE Usage;
   BYTE UsageIndex;
} D3DVERTEXELEMENT9;
```

  - 예)

```
D3DVERTEXELEMENT9 decl[] = {
 {0,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0},
 {0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL,   0},
 {0, 24, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL,   1},
 {0, 36, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL,   2},
 D3DDECL_END() };
```

버텍스 선언

- 버텍스 선언 만들기

```
HRESULT IDirect3DDevice9::CreateVertexDeclaration(
    CONST D3DVERTEXELEMENT9* pVertexElements,
    IDirect3DVertexDeclaration9** ppDecl
);
```

  - 예)

```
IDirect3DVertexDeclaration9* _decl;
hr = _device->CreateVertexDeclaration(decl, &_decl);
```

- 버텍스 선언 활성화하기

```
HRESULT IDirect3DDevice9::SetVertexDeclaration(
    IDirect3DVertexDeclaration9 *pDecl
);
```

  - 예) `_device->SetVertexDeclaration(_decl);`
  - 비교) `_device->SetFVF( MeshVertex::FVF );`

버텍스 데이터 이용

- 입력 구조체

버텍스 선언

버텍스 셰이더 입력 구조체

  - 예)

```
D3DVERTEXELEMENT9 decl[] = {
  {0,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0},
  {0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL,   0},
  {0, 24, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL,   1},
  {0, 36, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL,   2},
  D3DDECL_END()
};
```

의미 (: 용도[인덱스])

```
struct VS_INPUT
{
    vector position      : POSITION;
    vector normal        : NORMAL0;
    vector faceNormal1   : NORMAL1;
    vector faceNormal2   : NORMAL2;
};
```

인덱스를 생략하면 0을 지정한 것과 같은 의미

버텍스 데이터 이용

- 지원되는 버텍스 셰이더 입력 용도:

POSITION[n]	BLENDWEIGHTS[n]	BLENDINDICES[n]
NORMAL[n]	PSIZE[n]	DIFFUSE[n]
SPECULAR[n]	TEXCOORD[n]	TANGENT[n]
BINORMAL[n]	TESSFACTOR[n]	

  - **n** : [0, 15] 사이의 정수

- 출력 구조체

```
struct VS_OUTPUT
{
    vector position   : POSITION;
    vector diffuse    : COLOR0;
    vector specular   : COLOR1;
};
```

- 지원되는 버텍스 셰이더 출력 용도:

POSITION	PSIZE	FOG
COLOR[n]	TEXCOORD[n]	

버텍스 셰이더를 이용하기 위한 단계

1. 버텍스 셰이더를 작성하고 컴파일
2. 컴파일된 셰이더 코드를 기반으로 버텍스 셰이더를 나타내는 IDirect3DVertexShader9 인터페이스 생성
3. IDirect3DDevice9::SetVertexShader 메서드를 이용해 버텍스 셰이더 활성화
4. 이용이 끝난 뒤, 버텍스 셰이더 제거

버텍스 셰이더의 작성과 컴파일

- HLSL을 이용해 버텍스 셰이더 프로그램 작성
  - ASCII 텍스트 파일 (텍스트 편집기 이용)
- 버텍스 셰이더 컴파일
  - D3DXCompileShaderFromFile 함수 이용
    ➔ 컴파일된 셰이더 코드를 포함하는 ID3DXBuffer 포인터 리턴

```
HRESULT ID3DXCompileShaderFromFile (
        LPCSTR                  pSrcFile,
        CONST D3DXMACRO*            pDefines,
        LPD3DXINCLUDE           pInclude,
        LPCSTR                  pFunctionName,
        LPCSTR                  pTarget,
        DWORD                   Flags,
        LPD3DXBUFFER*           ppShader,
        LPD3DXBUFFER*           ppErrorMsgs,
        LPD3DXCONSTANTTABLE* ppConstantTable  );
```

버텍스 셰이더 만들기

- 버텍스 셰이더 만들기

```
HRESULT IDirect3DDevice9::CreateVertexShader(
   const DWORD *pFunction,
   IDirect3DVertexShader9** ppShader
);
```

- 예)

```
IDirect3DVertexShader9* DiffuseShader = 0;
ID3DXConstantTable* DiffuseConstTable = 0;
ID3DXBuffer* shader     = 0;
ID3DXBuffer* errorBuffer = 0;
hr = D3DXCompileShaderFromFile(
                "diffuse.txt",
                0,
                0,
                "Main",                // entry point function name
                "vs_1_1",
                D3DXSHADER_DEBUG,
                &shader,
                &errorBuffer,
                &DiffuseConstTable );
hr = Device->CreateVertexShader(
                (DWORD*)shader->GetBufferPointer(),
                &DiffuseShader );
```

버텍스 셰이더의 활성화와 제거

- 버텍스 셰이더의 활성화

```
HRESULT IDirect3DDevice9:: SetVertexShader(
   IDirect3DVertexShader9* pShader
);
```

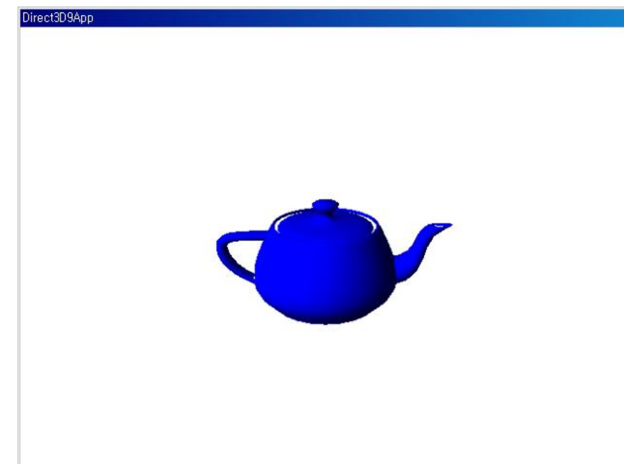
  - 예)

```
Device->BeginScene();

Device->SetVertexShader(DiffuseShader);

Teapot->DrawSubset(0);

Device->EndScene();
```

- 이용이 끝난 뒤에는 버텍스 셰이더를 제거
  - 예)

```
d3d::Release<IDirect3DVertexShader9*>(DiffuseShader);
d3d::Release<ID3DXConstantTable*>(DiffuseConstTable);
```

Sample: Diffuse Lighting



Shader: “diffuse.txt”

```
matrix ViewMatrix;
matrix ViewProjMatrix;

vector AmbientMtrl;
vector DiffuseMtrl;

vector LightDirection;
vector DiffuseLightIntensity = {0.0f, 0.0f, 1.0f, 1.0f};
vector AmbientLightIntensity = {0.0f, 0.0f, 0.2f, 1.0f};

struct VS_INPUT {
    vector position : POSITION;
    vector normal   : NORMAL;
};

struct VS_OUTPUT {
    vector position : POSITION;
    vector diffuse  : COLOR;
};
```

Shader: “diffuse.txt”

```
VS_OUTPUT Main(VS_INPUT input) {
    VS_OUTPUT output = (VS_OUTPUT)0;

    output.position  = mul(input.position, ViewProjMatrix);

    LightDirection.w = 0.0f;
    input.normal.w   = 0.0f;
    LightDirection   = mul(LightDirection, ViewMatrix);
    input.normal     = mul(input.normal,   ViewMatrix);

    float s = dot(LightDirection, input.normal);

    if( s < 0.0f )
        s = 0.0f;

    output.diffuse = (AmbientMtrl * AmbientLightIntensity) +
                (s * (DiffuseLightIntensity * DiffuseMtrl));

    return output;
}
```

Global Variables and Creating a Mesh

```
IDirect3DVertexShader9* DiffuseShader = 0;
ID3DXConstantTable* DiffuseConstTable = 0;

ID3DXMesh* Teapot           = 0;

D3DXHANDLE ViewMatrixHandle     = 0;
D3DXHANDLE ViewProjMatrixHandle = 0;

D3DXHANDLE AmbientMtrlHandle    = 0;
D3DXHANDLE DiffuseMtrlHandle    = 0;
D3DXHANDLE LightDirHandle       = 0;

D3DXMATRIX Proj;

bool Setup() {
    HRESULT hr = 0;

    D3DXCreateTeapot(Device, &Teapot, 0);
```

Compiling and Creating a VS

```
ID3DXBuffer* shader      = 0;
ID3DXBuffer* errorBuffer = 0;

hr = D3DXCompileShaderFromFile(
        "diffuse.txt",
        0,
        0,
        "Main", // entry point function name
        "vs_1_1",
        D3DXSHADER_DEBUG,
        &shader,
        &errorBuffer,
        &DiffuseConstTable);

if( errorBuffer ) {
        ::MessageBox(0, (char*)errorBuffer->GetBufferPointer(), 0, 0);
        d3d::Release<ID3DXBuffer*>(errorBuffer);
}
if(FAILED(hr)) {
        ::MessageBox(0, "D3DXCompileShaderFromFile() - FAILED", 0, 0);
        return false;
}
hr = Device->CreateVertexShader(
        (DWORD*)shader->GetBufferPointer(),
        &DiffuseShader);

if(FAILED(hr)) {
        ::MessageBox(0, "CreateVertexShader - FAILED", 0, 0);
        return false;
}
d3d::Release<ID3DXBuffer*>(shader);
```

Obtaining Handles and Initializing VS's Variables

```
ViewMatrixHandle    = DiffuseConstTable->GetConstantByName(0,
                                                  "ViewMatrix");
ViewProjMatrixHandle= DiffuseConstTable->GetConstantByName(0,
                                                  "ViewProjMatrix");
AmbientMtrlHandle   = DiffuseConstTable->GetConstantByName(0,
                                                  "AmbientMtrl");
DiffuseMtrlHandle   = DiffuseConstTable->GetConstantByName(0,
                                                  "DiffuseMtrl");
LightDirHandle      = DiffuseConstTable->GetConstantByName(0,
                                                  "LightDirection");

D3DXVECTOR4 directionToLight(-0.57f, 0.57f, -0.57f, 0.0f);
DiffuseConstTable->SetVector(Device, LightDirHandle, &directionToLight);
D3DXVECTOR4 ambientMtrl(0.0f, 0.0f, 1.0f, 1.0f);
D3DXVECTOR4 diffuseMtrl(0.0f, 0.0f, 1.0f, 1.0f);
DiffuseConstTable->SetVector(Device,AmbientMtrlHandle,&ambientMtrl);
DiffuseConstTable->SetVector(Device,DiffuseMtrlHandle,&diffuseMtrl);
DiffuseConstTable->SetDefaults(Device);
D3DXMatrixPerspectiveFovLH(
        &Proj,      D3DX_PI * 0.25f, (float)Width / (float)Height, 1.0f, 1000.0f);
```

Display ( )

```
bool Display(float timeDelta) {
    if( Device ){
        // Update the scene: code snipped ...

        D3DXVECTOR3 position( cosf(angle)*7.0f, height, sinf(angle)*7.0f );
        D3DXVECTOR3 target(0.0f, 0.0f, 0.0f);
        D3DXVECTOR3 up(0.0f, 1.0f, 0.0f);
        D3DXMATRIX V;
        D3DXMatrixLookAtLH(&V, &position, &target, &up);

        DiffuseConstTable->SetMatrix(Device, ViewMatrixHandle, &V);

        D3DXMATRIX ViewProj = V * Proj;
        DiffuseConstTable->SetMatrix(Device, ViewProjMatrixHandle, &ViewProj);

        Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                            0xffffffff, 1.0f, 0);
        Device->BeginScene();
        Device->SetVertexShader(DiffuseShader);
        Teapot->DrawSubset(0);
        Device->EndScene();
        Device->Present(0, 0, 0, 0);
    }
    return true;
}
```

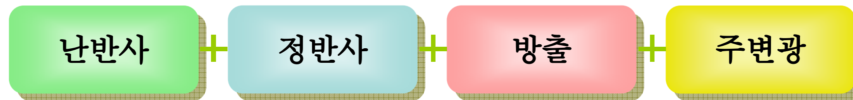
Cleanup ( )

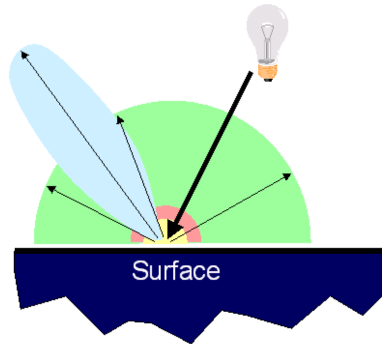
```
void Cleanup()
{
    d3d::Release<ID3DXMesh*>(Teapot);
    d3d::Release<IDirect3DVertexShader9*>(DiffuseShader);
    d3d::Release<ID3DXConstantTable*>(DiffuseConstTable);
}
```

Lab

- 정반사 조명(specular lighting)을 추가하시오.

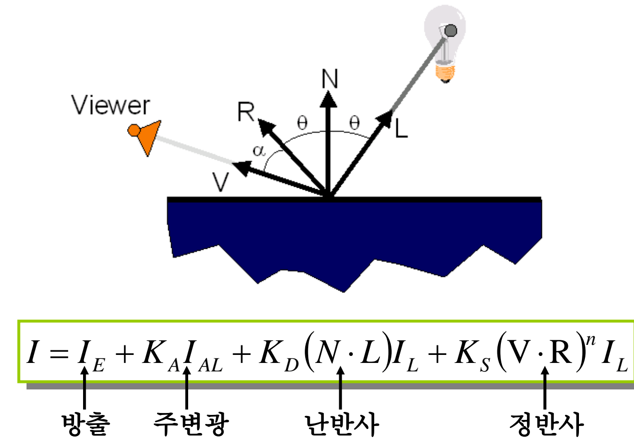
Phong의 조명 모델





렌더링 방정식

- 광원이 한 개인 경우:



$$I = I_E + K_A I_{AL} + K_D (N \cdot L) I_L + K_S (V \cdot R)^n I_L$$

방출 주변광 난반사 정반사

Shader: "specular.txt"

```
matrix ViewMatrix;
matrix ViewProjMatrix;

vector AmbientMtrl;
vector DiffuseMtrl;
vector SpecularMtrl;

vector LightDirection;

vector DiffuseLightIntensity = {0.0f, 0.0f, 1.0f, 1.0f};
vector AmbientLightIntensity = {0.0f, 0.0f, 0.2f, 1.0f};
vector SpecularLightIntensity = {1.0f, 1.0f, 1.0f, 1.0f};

// Input and Output structures.
struct VS_INPUT
{
    vector position : POSITION;
    vector normal   : NORMAL;
};

struct VS_OUTPUT
{
    vector position : POSITION;
    vector diffuse  : COLOR;
};
```

Shader: "specular.txt"

```
VS_OUTPUT Main(VS_INPUT input)
{
    VS_OUTPUT output = (VS_OUTPUT)0;

    output.position  = mul(input.position, ViewProjMatrix);

    LightDirection.w = 0.0f;
    input.normal.w   = 0.0f;
    LightDirection   = mul(LightDirection, ViewMatrix);
    input.normal     = mul(input.normal,   ViewMatrix);

    float s = dot(LightDirection, input.normal);
    if( s < 0.0f ) s = 0.0f;

    vector eyeToVertex  = normalize(mul(input.position, ViewMatrix));
    vector reflectLight = normalize(reflect(LightDirection, input.normal));

    float r = pow( saturate(dot(eyeToVertex, reflectLight)), 8 );

    output.diffuse = (AmbientMtrl * AmbientLightIntensity) +
                     (s * (DiffuseLightIntensity * DiffuseMtrl)) +
                         (r * (SpecularLightIntensity * SpecularMtrl));

    return output;
}
```

Global Variables

```
IDirect3DDevice9* Device = 0;

const int Width  = 640;
const int Height = 480;

IDirect3DVertexShader9* DiffuseShader = 0;
ID3DXConstantTable* DiffuseConstTable = 0;

ID3DXMesh* Teapot          = 0;

D3DXHANDLE ViewMatrixHandle     = 0;
D3DXHANDLE ViewProjMatrixHandle = 0;
D3DXHANDLE AmbientMtrlHandle    = 0;
D3DXHANDLE DiffuseMtrlHandle    = 0;
D3DXHANDLE SpecularMtrlHandle   = 0;
D3DXHANDLE LightDirHandle       = 0;

D3DXMATRIX Proj;
```

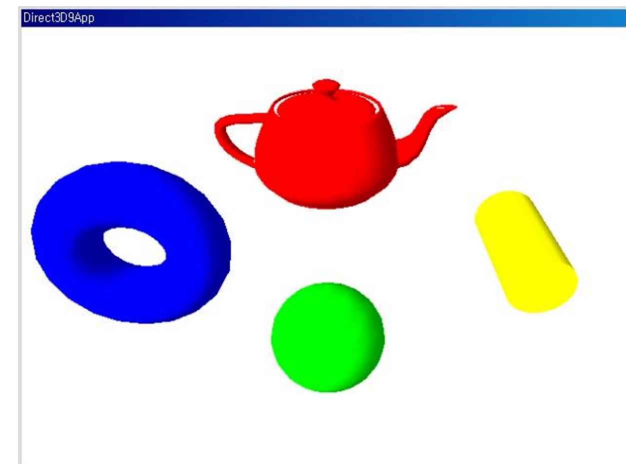
Setup ( ) – Compiling and Creating a VS

```
bool Setup() {
    HRESULT hr = 0;
    D3DXCreateTeapot(Device, &Teapot, 0);
    ID3DXBuffer* shader      = 0;
    ID3DXBuffer* errorBuffer = 0;
    hr = D3DXCompileShaderFromFile(
            "specular.txt",
            0,
            0,
            "Main", // entry point function name
            "vs_1_1",
            D3DXSHADER_DEBUG,
            &shader,
            &errorBuffer,
            &DiffuseConstTable);
    if( errorBuffer ) {
            ::MessageBox(0, (char*)errorBuffer->GetBufferPointer(), 0, 0);
            d3d::Release<ID3DXBuffer*>(errorBuffer);
    }
    if(FAILED(hr)) {
            ::MessageBox(0, "D3DXCompileShaderFromFile() - FAILED", 0, 0);
            return false;
    }
    hr = Device->CreateVertexShader(
            (DWORD*)shader->GetBufferPointer(),
            &DiffuseShader);
    if(FAILED(hr)) {
            ::MessageBox(0, "CreateVertexShader - FAILED", 0, 0);
            return false;
    }
    d3d::Release<ID3DXBuffer*>(shader);
```

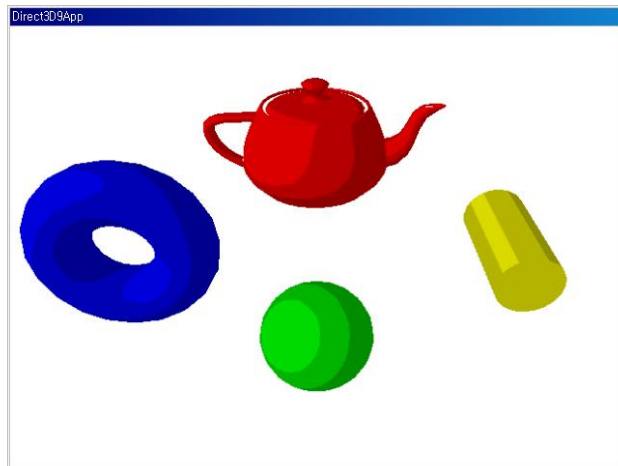
Setup ( ) – Obtaining Handles and Initializing VS's Variables

```
    ViewMatrixHandle    = DiffuseConstTable->GetConstantByName(0, "ViewMatrix");
    ViewProjMatrixHandle= DiffuseConstTable->GetConstantByName(0, "ViewProjMatrix");
    AmbientMtrlHandle   = DiffuseConstTable->GetConstantByName(0, "AmbientMtrl");
    DiffuseMtrlHandle   = DiffuseConstTable->GetConstantByName(0,  "DiffuseMtrl");
    SpecularMtrlHandle  = DiffuseConstTable->GetConstantByName(0,"SpecularMtrl");
    LightDirHandle      = DiffuseConstTable->GetConstantByName(0,   "LightDirection");

    D3DXVECTOR4 directionToLight(-0.57f, 0.57f, -0.57f, 0.0f);
    DiffuseConstTable->SetVector(Device, LightDirHandle, &directionToLight);

    D3DXVECTOR4 ambientMtrl(0.0f, 0.0f, 1.0f, 1.0f);
    D3DXVECTOR4 diffuseMtrl(0.0f, 0.0f, 1.0f, 1.0f);
    D3DXVECTOR4 specularMtrl(1.0f, 1.0f, 1.0f, 1.0f);

    DiffuseConstTable->SetVector(Device, AmbientMtrlHandle, &ambientMtrl);
    DiffuseConstTable->SetVector(Device, DiffuseMtrlHandle, &diffuseMtrl);
    DiffuseConstTable->SetVector(Device, SpecularMtrlHandle, &specularMtrl);
    DiffuseConstTable->SetDefaults(Device);

    D3DXMatrixPerspectiveFovLH( &Proj, D3DX_PI * 0.25f,
                        (float)Width / (float)Height, 1.0f, 1000.0f);
    return true;
}
```

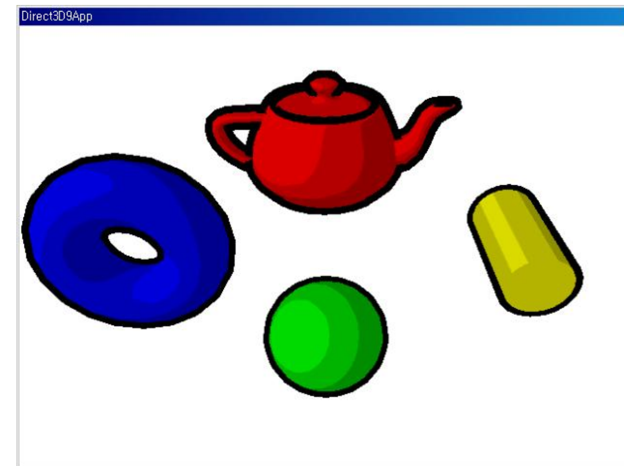
Sample: Shading with Diffuse



Sample: Cartoon 1



Sample: Cartoon 2

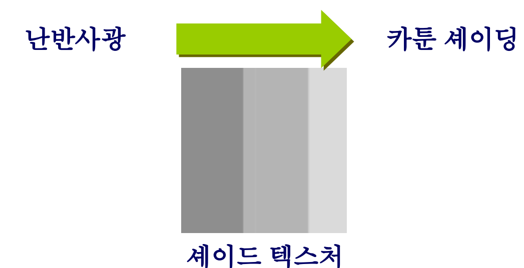


카툰 렌더링

- 비실사적 렌더링 방식 (non-photorealistic rendering)
- 스타일리스틱 렌더링
- 두 단계로 분리
  - 카툰 셰이딩 – 하나의 음영에서 다른 음영으로의 변환이 가파름 (단순한 음영 강도 – 예: 밝음, 중간, 어두움)
  - 실루엣 외곽선
  - ➔ 두 단계는 각각의 버텍스 셰이더가 필요!!

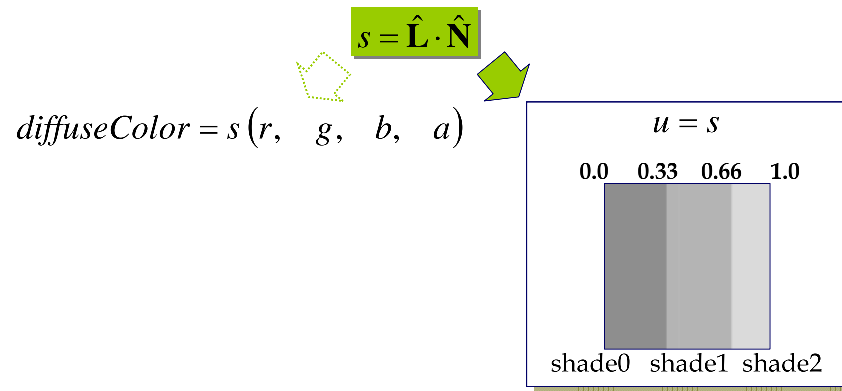
카툰 셰이딩

- "*Shades of Disney: Opaquing a 3D World*" (Game Developer 2000 3월호)와 같은 방법
- 그레이 스케일 명도 텍스처
  - 카툰 셰이딩에 이용할 음영 강도 (급격한 변환)
  - 왼쪽에서 오른쪽으로 음영의 강도가 증가



카툰 셰이딩

- 난반사광: 부드러운 → 급격한 음영 전환
  - s 는 컬러 벡터의 크기 변경을 위해 사용하는 대신, 명도 텍스처 좌표 u 로 사용



Shader: toon.txt

```
extern matrix WorldViewMatrix;
extern matrix WorldViewProjMatrix;
extern vector Color;
extern vector LightDirection;

struct VS_INPUT {
    vector position : POSITION;
    vector normal   : NORMAL;
};
struct VS_OUTPUT {
    vector position : POSITION;
    float2 uvCoords : TEXCOORD;
    vector diffuse  : COLOR;
};
VS_OUTPUT Main(VS_INPUT input) {
    VS_OUTPUT output = (VS_OUTPUT)0;

    output.position  = mul(input.position, WorldViewProjMatrix);

    LightDirection.w = 0.0f;
    input.normal.w   = 0.0f;
    LightDirection   = mul(LightDirection, WorldViewMatrix);
    input.normal     = mul(input.normal,   WorldViewMatrix);

    float u = dot(LightDirection, input.normal);
    if( u < 0.0f )   u = 0.0f;

    float v = 0.5f;
    output.uvCoords.x = u;
    output.uvCoords.y = v;

    output.diffuse = Color;

    return output;
}
```

Global Variables

```
IDirect3DDevice9* Device = 0;

const int Width  = 640;
const int Height = 480;

IDirect3DVertexShader9* ToonShader = 0;
ID3DXConstantTable* ToonConstTable = 0;

ID3DXMesh*  Meshes[4] = {0, 0, 0, 0};
D3DXMATRIX  WorldMatrices[4];
D3DXVECTOR4 MeshColors[4];

D3DXMATRIX ProjMatrix;

IDirect3DTexture9* ShadeTex  = 0;

D3DXHANDLE WorldViewHandle     = 0;
D3DXHANDLE WorldViewProjHandle = 0;
D3DXHANDLE ColorHandle         = 0;
D3DXHANDLE LightDirHandle      = 0;
```

Setup ( ) – Loading a Texture

```
bool Setup()
{
    // creating geometry of four meshes
    // computing world matrix and color for each mesh
    // compiling and creating a vertex shader : code snipped ...

    //
    // loading textures
    //
    D3DXCreateTextureFromFile(Device, "toonshade.bmp", &ShadeTex);

    Device->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_POINT);
    Device->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_POINT);
    Device->SetSamplerState(0, D3DSAMP_MIPFILTER, D3DTEXF_NONE);

    // obtaining the handles
    // initializing the variables of a vertex shader : code snipped ...

    return true;
}
```

Cleanup ( )

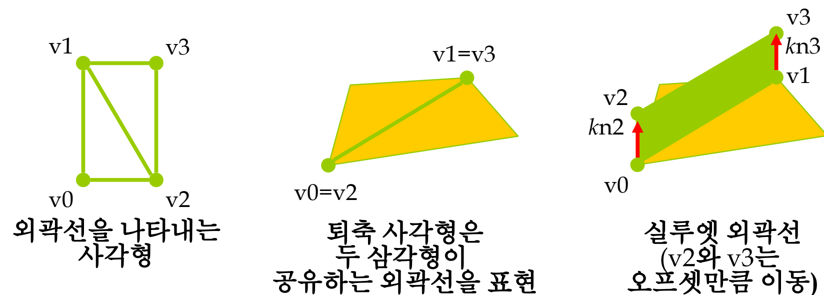
```
void Cleanup()
{
    for(int i = 0; i < 4; i++)
    {
        d3d::Release<ID3DXMesh*>(Meshes[i]);
    }
    d3d::Release<IDirect3DTexture9*>(ShadeTex);
    d3d::Release<IDirect3DVertexShader9*>(ToonShader);
    d3d::Release<ID3DXConstantTable*>(ToonConstTable);
}
```

Display ( )

```
bool Display(float timeDelta) {
    if( Device )     {
        // Update the scene : code snipped ...
        Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                    0xffffffff, 1.0f, 0);
        Device->BeginScene();
        Device->SetVertexShader(ToonShader);
        Device->SetTexture(0, ShadeTex);
        D3DXMATRIX WorldView;
        D3DXMATRIX WorldViewProj;
        for(int i = 0; i < 4; i++) {
            WorldView     = WorldMatrices[i] * view;
            WorldViewProj = WorldMatrices[i] * view * ProjMatrix;
            ToonConstTable->SetMatrix(Device,
                    WorldViewHandle,
                    &WorldView);
            ToonConstTable->SetMatrix(Device,
                    WorldViewProjHandle,
                    &WorldViewProj);
            ToonConstTable->SetVector(Device,
                    ColorHandle,
                    &MeshColors[i]);
            Meshes[i]->DrawSubset(0);
        }
        Device->EndScene();
        Device->Present(0, 0, 0, 0);
    }
    return true;
`
```

실루엣 외곽선

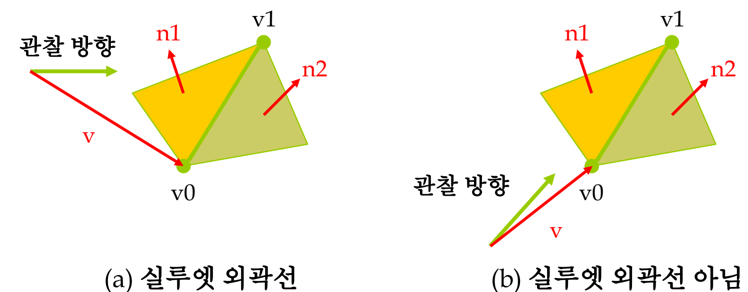
- 경계선 표현 ➔ 하나의 사각형
  - 사각형 넓이를 조정하여 손쉽게 외곽선의 두께를 변경
  - 외곽선을 감추기 위해 퇴축 사각형(degenerate quads)을 렌더링 (버텍스 버퍼에서 제거하지 않음)
- 실루엣 외곽선 ➔ 버텍스 법선 벡터 방향으로 버텍스 위치를 일정한 스칼라 만큼 이동



실루엣 외곽선

- 실루엣 경계 테스트
  - 만약 한 모서리를 공유하는 두 개의 면이 관찰 방향에서 서로 다른 쪽에 있다면, 실루엣 외곽선임

$$(\mathbf{v}\cdot\mathbf{n}_0)(\mathbf{v}\cdot\mathbf{n}_1)<0$$

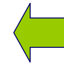


(a) 실루엣 외곽선 (b) 실루엣 외곽선 아님

실루엣 외곽선

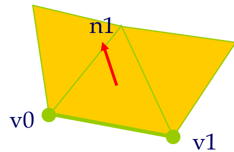
- 새로운 버텍스 선언
  - 인접하는 두 면의 법선 벡터 필요

```
struct VS_INPUT
{
    vector position        : POSITION0;
    vector normal          : NORMAL0;
    vector faceNormal1   : NORMAL1;
    vector faceNormal2   : NORMAL2;
};
```



```
struct EdgeVertex
{
    D3DXVECTOR3 position;
    D3DXVECTOR3 normal;
    D3DXVECTOR3 faceNormal1;
    D3DXVECTOR3 faceNormal2;
};
```

- 하나의 삼각형 만을 가지는 모서리
  ➔ 항상 실루엣 외곽선으로 정의



faceNormal2 = -faceNormal1;

Shader: outline.txt

```
extern matrix WorldViewMatrix;
extern matrix ProjMatrix;

static vector Black = {0.0f, 0.0f, 0.0f, 0.0f};

struct VS_INPUT
{
    vector position    : POSITION;
    vector normal      : NORMAL0;
    vector faceNormal1 : NORMAL1;
    vector faceNormal2 : NORMAL2;
};

struct VS_OUTPUT
{
    vector position : POSITION;
    vector diffuse  : COLOR;
};
```

Shader: outline.txt

```
VS_OUTPUT Main(VS_INPUT input) {
    VS_OUTPUT output = (VS_OUTPUT)0;

    input.position = mul(input.position, WorldViewMatrix);

    vector eyeToVertex = input.position;

    input.normal.w      = 0.0f;
    input.faceNormal1.w = 0.0f;
    input.faceNormal2.w = 0.0f;

    input.normal      = mul(input.normal,      WorldViewMatrix);
    input.faceNormal1 = mul(input.faceNormal1, WorldViewMatrix);
    input.faceNormal2 = mul(input.faceNormal2, WorldViewMatrix);

    float dot0 = dot(eyeToVertex, input.faceNormal1);
    float dot1 = dot(eyeToVertex, input.faceNormal2);

    if( (dot0 * dot1) < 0.0f ) input.position += 0.1f * input.normal;

    output.position = mul(input.position, ProjMatrix);
    output.diffuse = Black;

    return output;
}
```

Silhouette Edges (Header)

```
struct EdgeVertex {
    D3DXVECTOR3 position;
    D3DXVECTOR3 normal;
    D3DXVECTOR3 faceNormal1;
    D3DXVECTOR3 faceNormal2;
};

struct MeshVertex {
    D3DXVECTOR3 position;
    D3DXVECTOR3 normal;
    static const DWORD FVF;
};

class SilhouetteEdges {
public:
    SilhouetteEdges( IDirect3DDevice9* device,
                     ID3DXMesh* mesh,
                     ID3DXBuffer* adjBuffer );
    ~SilhouetteEdges();

    void render();
```

Silhouette Edges (Header)

```
private:
    IDirect3DDevice9*           _device;
    IDirect3DVertexBuffer9*     _vb;
    IDirect3DIndexBuffer9*      _ib;
    IDirect3DVertexDeclaration9* _decl;

    UINT _numVerts;
    UINT _numFaces;

private:
    bool createVertexDeclaration();
    void getFaceNormal( ID3DXMesh* mesh,
                        DWORD faceIndex,
                        D3DXVECTOR3* faceNormal );
    void getFaceNormals( ID3DXMesh* mesh,
                         ID3DXBuffer* adjBuffer,
                         D3DXVECTOR3* currentFaceNormal,
                         D3DXVECTOR3 adjFaceNormals[3],
                         DWORD       faceIndex );
    void genEdgeVertices( ID3DXMesh* mesh,
                          ID3DXBuffer* adjBuffer );
    void genEdgeIndices( ID3DXMesh* mesh );
};
```

Silhouette Edges

```
const DWORD MeshVertex::FVF = D3DFVF_XYZ | D3DFVF_NORMAL;

SilhouetteEdges::SilhouetteEdges( IDirect3DDevice9* device,
                                  ID3DXMesh* mesh,
                                  ID3DXBuffer* adjBuffer ) {
    _device   = device;
    _vb       = 0;
    _ib       = 0;
    _decl     = 0;
    _numVerts = 0;
    _numFaces = 0;
    genEdgeVertices(mesh, adjBuffer);
    genEdgeIndices(mesh);
    createVertexDeclaration();
}

SilhouetteEdges::~SilhouetteEdges() {
    d3d::Release<IDirect3DVertexBuffer9*>(_vb);
    d3d::Release<IDirect3DIndexBuffer9*>(_ib);
    d3d::Release<IDirect3DVertexDeclaration9*>(_decl);
}
```

Generating Edge Vertices

```
void SilhouetteEdges::genEdgeVertices( ID3DXMesh* mesh,
                                       ID3DXBuffer* adjBuffer )
{
    // 3 edges per face and 4 vertices per edge
    _numVerts = mesh->GetNumFaces() * 3 * 4;
    _device->CreateVertexBuffer(
            _numVerts * sizeof(EdgeVertex),
            D3DUSAGE_WRITEONLY,
            0, // using vertex declaration
            D3DPOOL_MANAGED,
            &_vb,
            0);
    MeshVertex* vertices = 0;
    mesh->LockVertexBuffer(0, (void**)&vertices);

    WORD* indices = 0;
    mesh->LockIndexBuffer(0, (void**)&indices);

    EdgeVertex* edgeVertices = 0;
    _vb->Lock(0, 0, (void**)&edgeVertices, 0);

    for(int i = 0; i < mesh->GetNumFaces(); i++)
    {
            D3DXVECTOR3 currentFaceNormal;
            D3DXVECTOR3 adjFaceNormals[3];
            getFaceNormals(mesh, adjBuffer, &currentFaceNormal, adjFaceNormals, i);
```

Generating Edge Vertices

```
        // get the indices for this face
        WORD index0 = indices[i * 3];
        WORD index1 = indices[i * 3 + 1];
        WORD index2 = indices[i * 3 + 2];

        // get the vertices for this face
        MeshVertex v0 = vertices[index0];
        MeshVertex v1 = vertices[index1];
        MeshVertex v2 = vertices[index2];

        // A        B
        // *--------*
        // |  edge  |
        // *--------*
        // C        D
        // note, C and D are duplicates of A and B respectively,
        // such that the quad is degenerate.  The vertex shader
        // will un-degenerate the quad if it is a silhouette edge.

        // compute edge0 v0->v1, note adjacent face
        // normal is faceNormal0
        EdgeVertex A0, B0, C0, D0;
        A0.position    = v0.position;
        A0.normal      = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
        A0.faceNormal1 = currentFaceNormal;
        A0.faceNormal2 = adjFaceNormals[0];
```

Generating Edge Vertices

```
        B0.position    = v1.position;
        B0.normal      = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
        B0.faceNormal1 = currentFaceNormal;
        B0.faceNormal2 = adjFaceNormals[0];

        C0 = A0;
        C0.normal = v0.normal;

        D0 = B0;
        D0.normal = v1.normal;

        *edgeVertices = A0; ++edgeVertices;
        *edgeVertices = B0; ++edgeVertices;
        *edgeVertices = C0; ++edgeVertices;
        *edgeVertices = D0; ++edgeVertices;

        // compute edge0 v1->v2, note adjacent face
        // normal is faceNormal1 : code snipped ...
        // compute edge0 v0->v2, note adjacent face
        // normal is faceNormal2: code snipped ...
    }
    _vb->Unlock();
    mesh->UnlockVertexBuffer();
    mesh->UnlockIndexBuffer();
}
```

Generating Face Normals

```
void SilhouetteEdges::getFaceNormals( ID3DXMesh*   mesh,
                                      ID3DXBuffer* adjBuffer,
                                      D3DXVECTOR3* currentFaceNormal,
                                      D3DXVECTOR3  adjFaceNormals[3],
                                      DWORD        faceIndex )
{
    MeshVertex* vertices = 0;
    mesh->LockVertexBuffer(0, (void**)&vertices);

    WORD* indices = 0;
    mesh->LockIndexBuffer(0, (void**)&indices);

    DWORD* adj = (DWORD*)adjBuffer->GetBufferPointer();

    // Get the face normal.
    getFaceNormal(mesh, faceIndex, currentFaceNormal);

    // Get adjacent face indices
    WORD faceIndex0 = adj[faceIndex * 3];
    WORD faceIndex1 = adj[faceIndex * 3 + 1];
    WORD faceIndex2 = adj[faceIndex * 3 + 2];
```

Generating Face Normals

```
    D3DXVECTOR3 faceNormal0, faceNormal1, faceNormal2;

    if( faceIndex0 != USHRT_MAX ) // is there an adjacent triangle?
    {
        WORD i0 = indices[faceIndex0 * 3];
        WORD i1 = indices[faceIndex0 * 3 + 1];
        WORD i2 = indices[faceIndex0 * 3 + 2];

        D3DXVECTOR3 v0 = vertices[i0].position;
        D3DXVECTOR3 v1 = vertices[i1].position;
        D3DXVECTOR3 v2 = vertices[i2].position;

        D3DXVECTOR3 edge0 = v1 - v0;
        D3DXVECTOR3 edge1 = v2 - v0;
        D3DXVec3Cross(&faceNormal0, &edge0, &edge1);
        D3DXVec3Normalize(&faceNormal0, &faceNormal0);
    }
    else {
        faceNormal0 = -(*currentFaceNormal);
    }
```

Generating Face Normals

```
    if( faceIndex1 != USHRT_MAX ) // is there an adjacent triangle?
    {
        // code snipped ...
    }
    else {
        faceNormal1 = -(*currentFaceNormal);
    }

    if( faceIndex2 != USHRT_MAX ) // is there an adjacent triangle?
    {
        // code snipped ...
    }
    else {
        faceNormal2 = -(*currentFaceNormal);
    }

    // save adjacent face normals
    adjFaceNormals[0] = faceNormal0;
    adjFaceNormals[1] = faceNormal1;
    adjFaceNormals[2] = faceNormal2;

    mesh->UnlockVertexBuffer();
    mesh->UnlockIndexBuffer();
}
```

Generating a Face Normal

```
void SilhouetteEdges::getFaceNormal( ID3DXMesh* mesh,
                                     DWORD faceIndex,   D3DXVECTOR3* faceNormal )
{
    MeshVertex* vertices = 0;
    mesh->LockVertexBuffer(0, (void**)&vertices);

    WORD* indices = 0;
    mesh->LockIndexBuffer(0, (void**)&indices);

    // Get the triangle's indices
    WORD index0 = indices[faceIndex * 3];
    WORD index1 = indices[faceIndex * 3 + 1];
    WORD index2 = indices[faceIndex * 3 + 2];

    // Now extract the triangles vertices positions
    D3DXVECTOR3 v0 = vertices[index0].position;
    D3DXVECTOR3 v1 = vertices[index1].position;
    D3DXVECTOR3 v2 = vertices[index2].position;

    // Compute face normal
    D3DXVECTOR3 edge0, edge1;
    edge0 = v1 - v0;
    edge1 = v2 - v0;
    D3DXVec3Cross(faceNormal, &edge0, &edge1);
    D3DXVec3Normalize(faceNormal, faceNormal);

    mesh->UnlockVertexBuffer();
    mesh->UnlockIndexBuffer();
}
```

Generating Edge Indices

```
void SilhouetteEdges::genEdgeIndices(ID3DXMesh* mesh)
{
    DWORD numEdges = mesh->GetNumFaces() * 3;

    _numFaces = numEdges * 2;

    _device->CreateIndexBuffer(
            numEdges * 6 * sizeof(WORD), // 2 triangles per edge
            D3DUSAGE_WRITEONLY,
            D3DFMT_INDEX16,
            D3DPOOL_MANAGED,
            &_ib,
            0);

    WORD* indices = 0;

    _ib->Lock(0, 0, (void**)&indices, 0);
```

Generating Edge Indices

```
    // 0        1
    // *--------*
    // |  edge  |
    // *--------*
    // 2        3

    for(UINT i = 0; i < numEdges; i++)
    {
            // Six indices to define the triangles of the edge,
            // so every edge we skip six entries in the
            // index buffer.  Four vertices to define the edge,
            // so every edge we skip four entries in the
            // vertex buffer.
            indices[i * 6]     = i * 4 + 0;
            indices[i * 6 + 1] = i * 4 + 1;
            indices[i * 6 + 2] = i * 4 + 2;
            indices[i * 6 + 3] = i * 4 + 1;
            indices[i * 6 + 4] = i * 4 + 3;
            indices[i * 6 + 5] = i * 4 + 2;
    }

    _ib->Unlock();
}
```

Creating Vertex Declaration

```
bool SilhouetteEdges::createVertexDeclaration()
{
    HRESULT hr = 0;

    D3DVERTEXELEMENT9 decl[] =    {
            // offsets in bytes
            {0,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                                        D3DDECLUSAGE_POSITION, 0},
            {0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                                        D3DDECLUSAGE_NORMAL,   0},
            {0, 24, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                                        D3DDECLUSAGE_NORMAL,   1},
            {0, 36, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                                        D3DDECLUSAGE_NORMAL,   2},
            D3DDECL_END()
    };

    hr = _device->CreateVertexDeclaration(decl, &_decl);
    if(FAILED(hr))
    {
            ::MessageBox(0, "CreateVertexDeclaration() - FAILED", 0, 0);
            return false;
    }
    return true;
}
```

Rendering Silhouette Edges

```
void SilhouetteEdges::render()
{
    _device->SetVertexDeclaration(_decl);
    _device->SetStreamSource(0, _vb, 0, sizeof(EdgeVertex));
    _device->SetIndices(_ib);

    _device->DrawIndexedPrimitive(
        D3DPT_TRIANGLELIST, 0, 0, _numVerts, 0, _numFaces);
}
```

Global Variables

```
IDirect3DDevice9* Device = 0;

const int Width  = 640;
const int Height = 480;

IDirect3DVertexShader9* ToonShader = 0;
ID3DXConstantTable* ToonConstTable = 0;

ID3DXMesh*  Meshes[4] = {0, 0, 0, 0};
D3DXMATRIX  WorldMatrices[4];
D3DXVECTOR4 MeshColors[4];

D3DXMATRIX ProjMatrix;

IDirect3DTexture9* ShadeTex  = 0;

D3DXHANDLE ToonWorldViewHandle     = 0;
D3DXHANDLE ToonWorldViewProjHandle = 0;
D3DXHANDLE ToonColorHandle    = 0;
D3DXHANDLE ToonLightDirHandle = 0;

SilhouetteEdges* MeshOutlines[4] = {0, 0, 0, 0};
IDirect3DVertexShader9* OutlineShader = 0;
ID3DXConstantTable* OutlineConstTable = 0;

D3DXHANDLE OutlineWorldViewHandle = 0;
D3DXHANDLE OutlineProjHandle = 0;
```

Setup ( ) – Creating Outline

```
bool Setup() {
    // creating geometry of four meshes  computing world matrix and color for each
    mesh : // code snipped ... allocating mesh outlines
    MeshOutlines[0] = new SilhouetteEdges(Device, Meshes[0], adjBuffer[0]);
    MeshOutlines[1] = new SilhouetteEdges(Device, Meshes[1], adjBuffer[1]);
    MeshOutlines[2] = new SilhouetteEdges(Device, Meshes[2], adjBuffer[2]);
    MeshOutlines[3] = new SilhouetteEdges(Device, Meshes[3], adjBuffer[3]);

    d3d::Release<ID3DXBuffer*>(adjBuffer[0]);
    d3d::Release<ID3DXBuffer*>(adjBuffer[1]);
    d3d::Release<ID3DXBuffer*>(adjBuffer[2]);
    d3d::Release<ID3DXBuffer*>(adjBuffer[3]);

    // compiling and creating cartoon shader : code snipped ...
    // compiling and creating outline shader
    ID3DXBuffer* outlineCompiledCode = 0;
    ID3DXBuffer* outlineErrorBuffer  = 0;

    hr = D3DXCompileShaderFromFile(
        "outline.txt",
        0,
        0,
        "Main", // entry point function name
        "vs_1_1",
        D3DXSHADER_DEBUG,
        &outlineCompiledCode,
        &outlineErrorBuffer,
        &OutlineConstTable);
```

Setup ( ) – Creating Outline

```
    if( outlineErrorBuffer )       {
        ::MessageBox(0, (char*)outlineErrorBuffer->GetBufferPointer(), 0, 0);
        d3d::Release<ID3DXBuffer*>(outlineErrorBuffer);
    }

    if(FAILED(hr))    {
        ::MessageBox(0, "D3DXCompileShaderFromFile() - FAILED", 0, 0);
        return false;
    }

    hr = Device->CreateVertexShader(
        (DWORD*)outlineCompiledCode->GetBufferPointer(),
        &OutlineShader);

    if(FAILED(hr))    {
        ::MessageBox(0, "CreateVertexShader - FAILED", 0, 0);
        return false;
    }
    d3d::Release<ID3DXBuffer*>(outlineCompiledCode);

    // loading textures
    // obtaining the handles : code snipped ...
    OutlineWorldViewHandle = OutlineConstTable->GetConstantByName(0,
        "WorldViewMatrix");
    OutlineProjHandle      = OutlineConstTable-
    >GetConstantByName(0,  "ProjMatrix");
```

Setup ( ) – Creating Outline

```
    // Set shader constants:
    // Light direction:
    D3DXVECTOR4 directionToLight(-0.57f, 0.57f, -0.57f, 0.0f);

    ToonConstTable->SetVector(
            Device,
            ToonLightDirHandle,
            &directionToLight);

    ToonConstTable->SetDefaults(Device);
    OutlineConstTable->SetDefaults(Device);

    // Calculate projection matrix.
    D3DXMatrixPerspectiveFovLH(
            &ProjMatrix, D3DX_PI * 0.25f,
            (float)Width / (float)Height, 1.0f, 1000.0f);

    return true;
}
```

Cleanup ( )

```
void Cleanup()
{
    for(int i = 0; i < 4; i++)      {
            d3d::Release<ID3DXMesh*>(Meshes[i]);
            d3d::Delete<SilhouetteEdges*>(MeshOutlines[i]);
    }
    d3d::Release<IDirect3DTexture9*>(ShadeTex);
    d3d::Release<IDirect3DVertexShader9*>(ToonShader);
    d3d::Release<ID3DXConstantTable*>(ToonConstTable);
    d3d::Release<IDirect3DVertexShader9*>(OutlineShader);
    d3d::Release<ID3DXConstantTable*>(OutlineConstTable);
}
```

Display ( )

```
bool Display(float timeDelta)
{
    if( Device )        {
            // Updating the scene
            // Drawing Cartoon : code snipped ...
            // Drawing Outlines
            Device->SetVertexShader(OutlineShader);
            Device->SetTexture(0, 0);
            Device->SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE);
            for(int i = 0; i < 4; i++)          {
               worldView = WorldMatrices[i] * view;

               OutlineConstTable->SetMatrix( Device,
                                             OutlineWorldViewHandle,
                                             &worldView );
               OutlineConstTable->SetMatrix( Device,
                                             OutlineProjHandle,
                                             &ProjMatrix );
               MeshOutlines[i]->render();
            }
            Device->SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
            Device->EndScene();
            Device->Present(0, 0, 0, 0);
    }
    return true;
}
```