Chapter18-10 簡易ぼかしシェーダー　［応用］

ポストエフェクトで利用例が多いものにぼかしシェーダーがある。代表的な例として紹介したHDRや被写界深度など非常に多くのエフェクトでぼかす手法が使われているここでは簡易版として4サンプリングのぼかしシェーダーと９サンプリングのぼかしシェーダーを紹介する。

Program18-10 Shader.hlsl

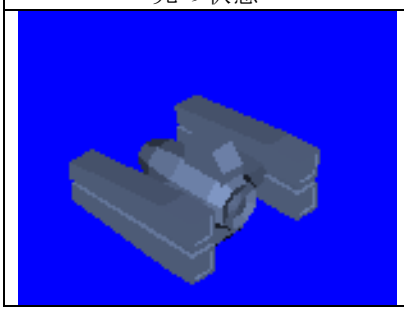
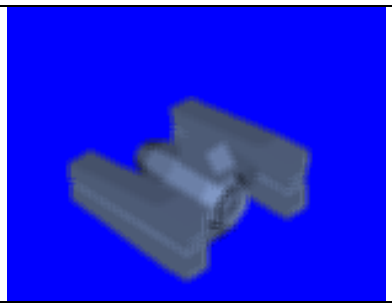
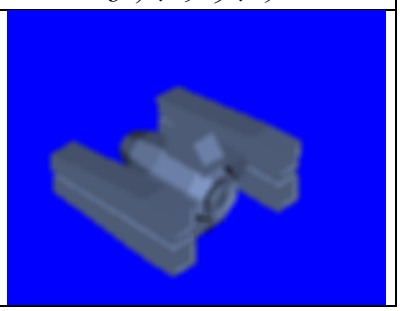
```
//! ピクセルシェーダ
float4 RenderPS( OutputVS inPixel ) : SV_TARGET
{
	float4 color = txDiffuse.Sample(samLinear, inPixel.Tex, float2(-1, -1));
	color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(1, -1));
	color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(-1, 1));
	color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(1, 1));
	color /= 4.0f;
	return color * inPixel.color;
}
```

４サンプリングのぼかしシェーダーはテクスチャの本来描画する周囲４点を参照して、その色を合計して平均をそのピクセルの色としている。

Program18-10 Shader.hlsl

```
//! ピクセルシェーダ
float4 RenderPS( OutputVS inPixel ) : SV_TARGET
{
	float4 color = txDiffuse.Sample( samLinear, inPixel.Tex, float2(-1,-1));
	color += txDiffuse.Sample( samLinear, inPixel.Tex, float2( 0,-1));
	color += txDiffuse.Sample( samLinear, inPixel.Tex, float2( 1,-1));
	color += txDiffuse.Sample( samLinear, inPixel.Tex, float2(-1, 0));
	color += txDiffuse.Sample( samLinear, inPixel.Tex, float2( 0, 0));
	color += txDiffuse.Sample( samLinear, inPixel.Tex, float2( 1, 0));
	color += txDiffuse.Sample( samLinear, inPixel.Tex, float2(-1, 1));
	color += txDiffuse.Sample( samLinear, inPixel.Tex, float2( 0, 1));
	color += txDiffuse.Sample( samLinear, inPixel.Tex, float2( 1, 1));
	color /= 9.0f;	return color * inPixel.color;
}
```

９サンプリングのぼかしシェーダーも考え方は同じで、ただ参照するピクセルを周囲９ピクセルに増やしているだけである。参照するピクセルが増えれば増えるほどぼかしは強くなるが当然、参照を増やせば処理は重くなる。自分が必要とする品質を出すためにいかに効率よく参照するピクセルを決めるかが重要になっている。

元の状態	４サンプリング	９サンプリング
		

Chapter18-11 色差エッジシェーダー　［応用］

ぼかしシェーダーと同じように周辺を参照しておこなうエフェクトとして各種のエッジ（輪郭線）出力用のシェーダーが存在する。３Ｄのレンダリングではトゥーンシェーディングに代表されるようなアニメ調の描画は専用のシェーダーが必要になる。特に輪郭線の抽出は様々な方法が存在する。
ここでは隣接するピクセルの色の差を元にポストエフェクトとしてエッジの抽出をおこなう方法を紹介する。

エッジの抽出は参照するピクセルの数や各ピクセルの比率（ウェイト・重み）によって様々なアルゴリズムが存在する。

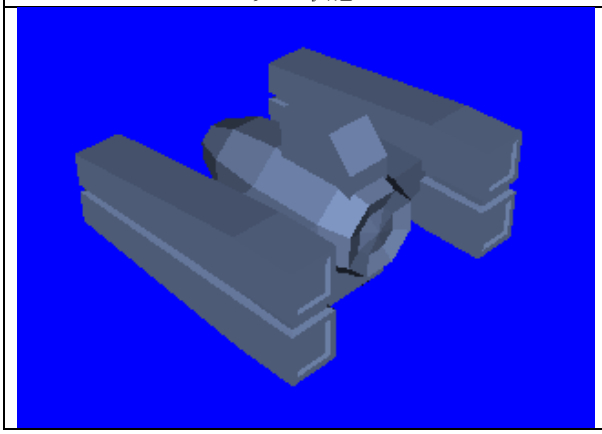
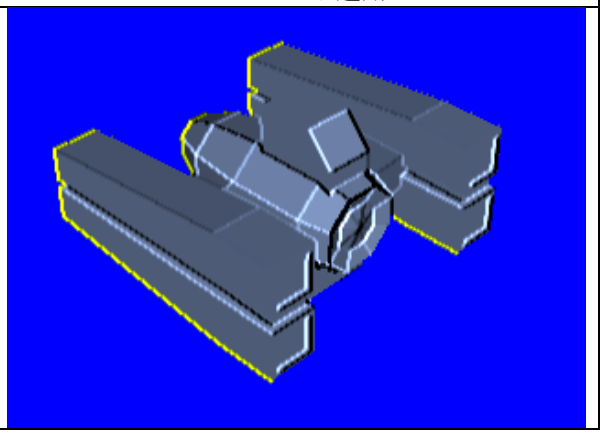
まずその中でも簡単で比較的効果の高い sobel（ゾーベル）フィルタを紹介する。
sobel フィルタは空間１次微分を計算しエッジを抽出する。エッジを出したい方向の重みを+2、その周囲を+1 として、逆方向の重みを－にすることで表現できる。
例えば右方向から左方向への輪郭を検出するためのテーブルは下記の表になる。

-1	0	1
-2	0	2
-1	0	1

シェーダー上でテーブル通りの参照をおこなった例を紹介する。

Program18-11 Shader.hlsl

```
//! ピクセルシェーダ
//sobel フィルタ
//    {     -1,         0,          1,          }
//    {     -2,         0,          2,          }
//    {     -1,         0,          1,          }
float4 RenderPS( OutputVS inPixel ) : SV_TARGET
{
    float4 color = float4(0,0,0,1);
    color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(-1, -1));
    color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(-1, 0)) * 2;
    color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(-1, 1));
    color.rgb += txDiffuse.Sample(samLinear, inPixel.Tex, float2(1, -1));
    color.rgb += txDiffuse.Sample(samLinear, inPixel.Tex, float2(1, 0)) * 2;
    color.rgb += txDiffuse.Sample(samLinear, inPixel.Tex, float2(1, 1));

    //元画像の追加
    color.rgb += txDiffuse.Sample(samLinear, inPixel.Tex);
    return color * inPixel.color;
}
```

元の状態	エフェクト適用
	

同じエッジ抽出のプログラムとして Laplacian（ラプラシアン）フィルタを紹介する。

Laplacian フィルタは空間２次微分を計算しエッジを抽出する。Sobel フィルタと違い一度のレンダリングで全ての方向へ万遍なくサンプリングをおこなうため、特定の方向にエッジを出すのではなく全体で輝度差が大きい位置にエッジが出力される特徴がある。

輪郭を抽出するためのテーブルは下記の表になる。今回は３×３の９サンプリングのものと、５×５の２５サンプリングのものの２種類のテーブルを紹介する。

3x3

-1	-1	-1
-1	8	-1
-1	-1	-1

5x5

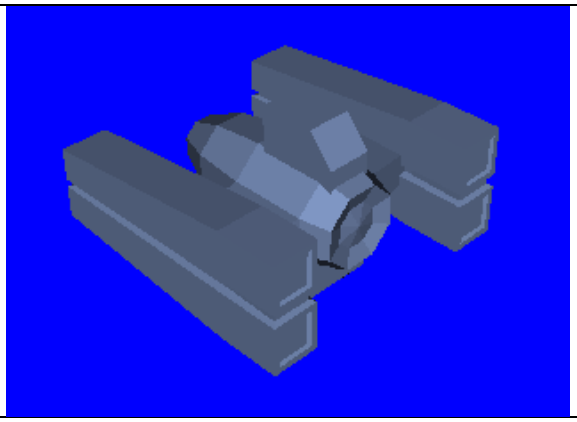
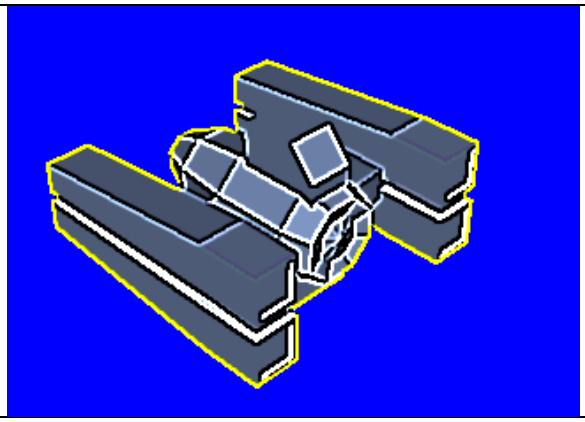
-1	-3	-4	-3	-1
-3	0	6	0	-3
-4	6	20	6	-4
-3	0	6	0	-3
-1	-3	-4	-3	-1

シェーダー上で５×５のテーブル通りの参照をおこなった例を紹介する。

Program18-11 Shader.hlsl

```
//! ピクセルシェーダ
//ラプラシアンフィルタ
//    3x3
//    {     -1,         -1,         -1,         }
//    {     -1,         8,          -1,         }
//    {     -1,         -1,         -1,         }
//
//    5x5
//    {     -1,         -3,         -4,         -3,         -1,         }
//    {     -3,         0,          6,          0,          -3,         }
//    {     -4,         6,          20,         6,          -4,         }
//    {     -3,         0,          6,          0,          -3,         }
//    {     -1,         -3,         -4,         -3,         -1,         }
float4 RenderPS( OutputVS inPixel ) : SV_TARGET
{
    float4 color = float4(0,0,0,1);
    color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(-2, -2));
    color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(-2, -1)) * 3;
    color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(-2, 0)) * 4;
    color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(-2, 1)) * 3;
    color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(-2, 2));

    color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(-1, -2)) * 3;
    color.rgb += txDiffuse.Sample(samLinear, inPixel.Tex, float2(-1, 0)) * 6;
    color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(-1, 2)) * 3;

    color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(0, -2)) * 4;
    color.rgb += txDiffuse.Sample(samLinear, inPixel.Tex, float2(0, -1)) * 6;
    color.rgb += txDiffuse.Sample(samLinear, inPixel.Tex) * 20;
    color.rgb += txDiffuse.Sample(samLinear, inPixel.Tex, float2(0, 1)) * 6;
    color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(0, 2)) * 4;

    color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(1, -2)) * 3;
    color.rgb += txDiffuse.Sample(samLinear, inPixel.Tex, float2(1, 0)) * 6;
    color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(1, 2)) * 3;

    color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(2, -2));
    color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(2, -1)) * 3;
    color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(2, 0)) * 4;
    color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(2, 1)) * 3;
    color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(2, 2));

    //元画像の追加
    color.rgb += txDiffuse.Sample(samLinear, inPixel.Tex);
    return color * inPixel.color;
```

```
}
```

元の状態	エフェクト適用
	

エッジ抽出の活用の一例として鮮鋭化フィルタがある。鮮鋭化は周囲のピクセルを参照しエッジ部分を強調することで画像を鮮明にするフィルタ処理。

鮮鋭化をするためのテーブルは下記の表になる。

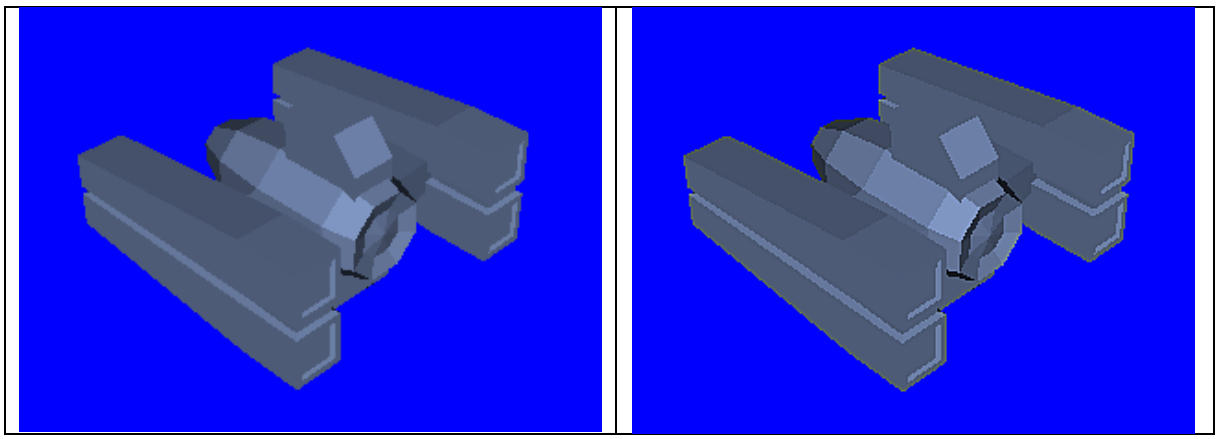
−1	−1	−1	÷ 8
−1	16	−1	
−1	−1	−1	

シェーダー上でテーブル通りの参照をおこなった例を紹介する。

Program18-11 Shader.hlsl

```
//! ピクセルシェーダ
//先鋭化フィルタ
//    {     -1,         -1,         -1,         }
//    {     -1,         16,         -1,         }         /         8
//    {     -1,         -1,         -1,         }
float4 RenderPS( OutputVS inPixel ) : SV_TARGET
{
        float4 color = float4(0,0,0,1);
        color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(-1, -1));
        color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(-1, 0));
        color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(-1, 1));
        color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(0, -1));
        color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(0, 1));
        color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(1, -1));
        color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(1, 0));
        color.rgb -= txDiffuse.Sample(samLinear, inPixel.Tex, float2(1, 1));

        color.rgb += txDiffuse.Sample(samLinear, inPixel.Tex) * 16;
        color.rgb /= 8;
        return color * inPixel.color;
}
```

元の状態	エフェクト適用



Chapter18-12 モザイクシェーダー　［応用］

ぼかしシェーダーと同じように周辺の色を参照しておこなうエフェクトにモザイクシェーダーが存在する。

Program18-12 Shader.hlsl

```
cbuffer cbGameParam : register(b4)
{
	float2	Screen;			//x : 画面幅, y : 画面高さ
	int		nCount;			//参照数
};
float4 RenderPS( OutputVS inPixel ) : SV_TARGET
{
	int AllPixel = nCount * nCount;
	//平均化するための基準矩形座標を求める
	int2 stex = int2(
		int(floor(inPixel.Tex.x * Screen.x / nCount)) * nCount,
		int(floor(inPixel.Tex.y * Screen.y / nCount)) * nCount
	);
	float2 tex = stex / Screen;

	float4 color = float4(0, 0, 0, 0);
	//基準座標から参照数分のピクセルをサンプリングして平均化する
	for (int x = 0; x < nCount; ++x)
	{
		for (int y = 0; y < nCount; ++y)
		{
			color += txDiffuse.Sample(samLinear, tex + float2(x, y) / Screen);
		}
	}
	color /= AllPixel;
	return color * inPixel.color;
}
```

ゲーム側のコードで今回は必要な変数が３つなため、コンスタントバッファと同様の構造体にダミーを入れている。

Program18-12 GameApp.cpp

```
//シェーダー側のコンスタントバッファと同様の構造体
struct cbGameParam
{
	Vector2	Screen;
	int		nCount;
	float	d[1];		//アライメント調整用のダミー
};
int gcnt = 5;
```

```
MofBool CGameApp::Initialize(void){
	//リソース配置ディレクトリの設定
	CUtilities::SetCurrentDirectory("Resource");

	//カメラ初期化
	gCamera.SetViewPort();
	gCamera.LookAt(Vector3(-2.0f,2.0f,-2.0f),Vector3(0,0,0),Vector3(0,1,0));
	gCamera.PerspectiveFov(MOF_ToRadian(60.0f),1024.0f / 768.0f,0.01f,1000.0f);
	gCamera.Update();
	CGraphicsUtilities::SetCamera(&gCamera);

	//メッシュの読み込み
	gMesh.Load("player.mom");

	//描画ターゲットを作成する
	MofU32 sw = g_pGraphics->GetTargetWidth();
	MofU32 sh = g_pGraphics->GetTargetHeight();
	gTarget.CreateTarget(sw, sh, PIXELFORMAT_R8G8B8A8_UNORM,
BUFFERACCESS_GPUREADWRITE);

	//シェーダーの読み込み
	gShader.Load("Shader.hlsl");
	gShaderBind.Create(&gShader);
	//シェーダーに２D 描画用の行列を設定する
	CGraphicsUtilities::SetScreenSize(sw, sh, &gShaderBind);
	//シェーダーとのバッファの連携
	BOOL re = gShaderBind.CreateShaderBuffer("cbGameParam", sizeof(cbGameParam));
	return TRUE;
}
```

プログラム中でダミーデータを削除してアライメントを崩すとエラーが発生する。これはコンスタントバッファのアライメントを１６Byte にそろえる必要があるにもかかわらず、cbGameParam のサイズが Vector2(float×２)＋int の１２Byte しかないため生成に失敗している。

Program18-12 エラープログラムの例

```
//シェーダー側のコンスタントバッファと同様の構造体
struct cbGameParam
{
	Vector2	Screen;
	int		nCount;
};
```

```
BOOL re = gShaderBind.CreateShaderBuffer("cbGameParam", sizeof(cbGameParam));
上記の戻り値が FALSE に変化してしまう。
```

描画処理では今回は参照数を左右のキーで変化させるプログラムを追加している。

Program18-12 GameApp.cpp

```
MofBool CGameApp::Render(void){
	//描画処理
	g_pGraphics->RenderStart();

	//元の描画ターゲットを取得する
	LPRenderTarget pold = g_pGraphics->GetRenderTarget();

	//作成したテクスチャを描画ターゲットとして設定する
	//深度バッファは元の情報をそのまま使用する
	g_pGraphics->SetRenderTarget(gTarget.GetRenderTarget(), g_pGraphics->GetDepthTarget());

	//画面のクリア
	g_pGraphics->ClearTarget(0.0f,0.0f,1.0f,1.0f,1.0f,0);

	//深度バッファ有効化
```

```
	g_pGraphics->SetDepthEnable(TRUE);

	//メッシュの描画
	CMatrix44 matWorld;
	gMesh.Render(matWorld);

	//描画ターゲットを元に戻す
	g_pGraphics->SetRenderTarget(pold, g_pGraphics->GetDepthTarget());
	g_pGraphics->SetDepthEnable(FALSE);

	//描画したターゲットをテクスチャとして画面に描画する
	if (!g_pInput->IsKeyHold(MOFKEY_SPACE))
	{
		CGraphicsUtilities::RenderTexture(0, 0, &gTarget);
	}
	else
	{
		//シェーダーに送るバッファを作成
		cbGameParam sb;
		sb.Screen.x = 1024;
		sb.Screen.y = 768;
		sb.nCount = gcnt;
		//シェーダーにバッファを送る
		gShaderBind.GetShaderBuffer(0)->SetBuffer(&sb);
		CGraphicsUtilities::RenderTexture(0, 0, &gTarget, &gShader, &gShaderBind);
	}
	CGraphicsUtilities::RenderString(10, 10, MOF_COLOR_WHITE, "SPACE キーでシェーダー使用");
	CGraphicsUtilities::RenderString(10, 32, MOF_COLOR_WHITE,
		"左右キーで参照数変更 [現在の参照数 : %d]", gcnt);
	if (g_pInput->IsKeyPush(MOFKEY_LEFT) && gcnt > 1)
	{
		gcnt--;
	}
	if (g_pInput->IsKeyPush(MOFKEY_RIGHT))
	{
		gcnt++;
	}

	//描画の終了
	g_pGraphics->RenderEnd();
	return TRUE;
}
```

Chapter18-13 深度出力 ［応用］

レンダリングターゲットを活用することによって、色を参照する以外にも様々な効果が考案されている。その代表的な例として深度を保存しておこなうエフェクトがあげられる。

３D のレンダリングでは深度バッファを参照して、描画するジオメトリの前後関係を反映しているが、深度バッファはプログラムからリアルタイムで高速なアクセスは難しい。そこでレンダリングターゲットに色の代わりに深度の値（ビュー変換した Z 座標）を出力することでシェーダー上で高速に深度を読み取りエフェクトを実現している。

深度をレンダリングターゲットに出力するサンプルを紹介する。

Program18-13 Shader.hlsl

```
//! コンスタントバッファ
cbuffer cbSceneParam : register( b0 )
{
	float4vecViewPos		: packoffset( c0 );
	matrix	mtxView		: packoffset( c1 );
```

```
	matrix	mtxProj		: packoffset( c5 );
};

cbuffer cbMeshParam : register( b1 )
{
	matrix	mtxWorld		: packoffset(c0);
	float4colRevise		: packoffset(c4);
	float4CoordsRevise	: packoffset(c5);
};

cbuffer cbMaterialParam : register( b2 )
{
	float4matDiffuse	: packoffset( c0 );
	float4matAmbient		: packoffset( c1 );
	float4matSpeculer	: packoffset( c2 );
	float4matEmissive	: packoffset( c3 );
	float	matPower		: packoffset( c4 );
};

cbuffer cbLightParam : register( b3 )
{
	float3litDirection : packoffset( c0 );
	float4litDiffuse		: packoffset( c1 );
	float4litAmbient		: packoffset( c2 );
	float4litSpeculer : packoffset( c3 );
};

Texture2D txDiffuse : register( t0 );
SamplerState samLinear : register( s0 );

//! 頂点属性
struct InputVS
{
	float4pos		: POSITION;
	float3normal		: NORMAL;
	float2Tex		: TEXCOORD;
	float4color		: COLOR0;
};
struct OutputVS
{
	float4pos		: SV_POSITION;
	float4	VPos		: COLOR0;
};

//! 新規のコンスタントバッファ作成
cbuffer cbGameParam : register( b4 )
{
	float4	cbProjection		: packoffset(c0);
};

//! 頂点シェーダ
OutputVS RenderVS( InputVS inVert )
{
	OutputVS	outVert;

	matrix	mtxVP = mul( mtxView, mtxProj );
	float4 Pos = mul(inVert.pos, mtxWorld);
	outVert.pos = mul( Pos , mtxVP );

	//深度算出用にビュー変換のみをおこなった座標を出力
	outVert.VPos = mul(Pos, mtxView);

	return outVert;
}
```

```
//! ピクセルシェーダ
float4 RenderPS( OutputVS inPixel ) : SV_TARGET
{
	return float4(inPixel.VPos.z / cbProjection.x, 0.0f, 0.0f, 1.0f);
}

technique11 TShader
{
	pass P0
	{
		SetVertexShader( CompileShader( vs_4_0, RenderVS() ) );
		SetGeometryShader( NULL );
		SetHullShader( NULL );
		SetDomainShader( NULL );
		SetPixelShader(  CompileShader( ps_4_0, RenderPS() ) );
		SetComputeShader( NULL );
	}
}
```

シェーダー側では通常の３D ジオメトリの描画と同じ手順での描画だが、頂点シェーダーからピクセルシェーダーへの出力にビュー変換までを実行した座標を加えている。

ピクセルシェーダーでは頂点シェーダーから出力されたビュー座標を出力する色の赤色成分としている。ここで Z 値をそのまま出力すると範囲によっては大きな値になってしまい、テクスチャの色数によっては出力しきれない可能性がある。また深度によるエフェクトを適用する場合、奥の方のピクセルは画面上でもあまり細かく見えないため不要なことが多い。

そこで自らが必要とする深度の範囲をシェーダーへ渡し、その値で除算することで値を決めている。

ゲーム側ではまず深度の出力用のレンダリングターゲットを作成する。作成するターゲットは深度のみを出力するため今までのレンダリングターゲットのように RGBA の４色ある必要がない。そのためテクスチャのフォーマットに PIXELFORMAT_R32_FLOAT を指定している。このフォーマットは通常 R8G8B8A8 の３２Bit で一つのピクセルを構成しているテクスチャを、一つのピクセルを R32（赤色）の 32Bit のみで構成することで同じテクスチャサイズで赤のみに多くの階調を出力することができる。深度はゲームにもよるが８Bit（０～２５５）の２５６階調では不足することが多いためこのフォーマットを選択している。

Program18-13 GameApp.cpp

```
//カメラ
CCamera                         gCamera;
//カメラ角度
CVector3                        gCamAngle;
//メッシュ
CMeshContainer                  gMesh;
//描画ターゲット
CTexture                        gTarget;
//シェーダー
CShader                         gShader;
CShaderBind_3DPrimitiveBase     gShaderBind;

//シェーダー側のコンスタントバッファと同様の構造体
struct cbGameParam
{
	Vector4	cbProjection;
};

MofBool CGameApp::Initialize(void){
	//リソース配置ディレクトリの設定
	CUtilities::SetCurrentDirectory("Resource");

	//カメラ初期化
	gCamera.SetViewPort();
	gCamera.LookAt(Vector3(0.0f, 0.0f, -2.0f), Vector3(0, 0, 0), Vector3(0, 1, 0));
```

```
	gCamera.PerspectiveFov(MOF_ToRadian(60.0f),1024.0f / 768.0f,0.01f,1000.0f);
	gCamera.Update();
	CGraphicsUtilities::SetCamera(&gCamera);

	//メッシュの読み込み
	gMesh.Load("player.mom");

	//シェーダーの読み込み
	gShader.Load("Shader.hlsl");
	gShaderBind.Create(&gShader);

	//シェーダーとのバッファの連携
	gShaderBind.CreateShaderBuffer("cbGameParam", sizeof(cbGameParam));

	//深度描画ターゲットを作成する
	MofU32 sw = g_pGraphics->GetTargetWidth();
	MofU32 sh = g_pGraphics->GetTargetHeight();
	gTarget.CreateTarget(sw, sh, PIXELFORMAT_R32_FLOAT,
BUFFERACCESS_GPUREADWRITE);
	return TRUE;
}
```

描画処理で作成したレンダリングターゲットに深度値の出力をおこなう。深度の出力をおこなうためにはターゲットの変更後に作成した深度出力シェーダーを指定してモデルの描画をおこなうだけだが、ここでは深度の出力のみでこのままでは画面への出力はない。

そこで深度を別で出力する場合は**レンダリングターゲットを戻した後に通常のモデルのレンダリングもおこなう必要がある**。そのため**単純に描画回数が２倍になる**。

全ての３D モデルの描画が終わった後に深度を描画したテクスチャを確認用に画面左上に表示している。３D オブジェクトが奥に行くほど赤色が強くなっている深度情報を確認できる。

Program18-13 GameApp.cpp

```
MofBool CGameApp::Render(void){
	//描画処理
	g_pGraphics->RenderStart();

	//深度バッファ有効化
	g_pGraphics->SetDepthEnable(TRUE);

	//カメラを設定
	gShaderBind.SetCamera(&gCamera);

	//元の描画ターゲットを取得する
	LPRenderTarget pold = g_pGraphics->GetRenderTarget();

	//作成したテクスチャを描画ターゲットとして設定する
	//深度バッファは元の情報をそのまま使用する
	g_pGraphics->SetRenderTarget(gTarget.GetRenderTarget(), g_pGraphics-
>GetDepthTarget());

	//画面のクリア
	g_pGraphics->ClearTarget(0.0f,0.0f,0.0f,0.0f,1.0f,0);

	//シェーダーに送るバッファを作成
	cbGameParam sb;
	sb.cbProjection.x = 10;
	//シェーダーにバッファを送る
	gShaderBind.GetShaderBuffer(0)->SetBuffer(&sb);

	//メッシュの描画(深度を出力する)
	CMatrix44 matWorld;
	gMesh.Render(matWorld, &gShader, &gShaderBind);
```

```
	//描画ターゲットを元に戻す
	g_pGraphics->SetRenderTarget(pold, g_pGraphics->GetDepthTarget());

	//画面のクリア
	g_pGraphics->ClearTarget(0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0);

	//メッシュの描画(通常の描画)
	gMesh.Render(matWorld);

	//2D 描画に変更
	g_pGraphics->SetDepthEnable(FALSE);
	g_pGraphics->SetBlending(BLEND_NONE);

	//確認用に画面右上に深度を表示する
	gTarget.Render(CRectangle(0, 0,
		g_pGraphics->GetTargetWidth() / 4, g_pGraphics->GetTargetHeight() / 4));

	//描画の終了
	g_pGraphics->RenderEnd();
	return TRUE;
}

MofBool CGameApp::Release(void){
	gMesh.Release();
	gTarget.Release();
	//シェーダーの解放
	gShaderBind.Release();
	gShader.Release();
	return TRUE;
}
```

Chapter18-14 被写界深度 [応用]

深度を利用したポストエフェクトの代表的なものとして被写界深度（DepthOfField）があげられる。

被写界深度はピント（焦点）があっており、はっきりと見える範囲のことで、写真ではピントがあっていない部分はぼやけて写る。シェーダーエフェクトで被写界深度といわれる場合は、その効果を３Ｄレンダリング結果のシーンに適用する。これにより、遠近感が生まれ、リアリティが増す。

サンプルでは深度が一定より大きい場合にぼかしフィルタを掛ける簡単な処理で実装をおこなっている。

Program18-14 Dof.hlsl

```
//! コンスタントバッファ
cbuffer cbSceneParam : register(b0)
{
	matrix	mtxView		: packoffset( c0 );
	matrix	mtxProj		: packoffset( c4 );
};

cbuffer cbMeshParam : register( b1 )
{
	float4cbvOffset		: packoffset( c0 );
	float4cbvSize		: packoffset( c1 );
	float4cbtOffset		: packoffset( c2 );
	float4cbtSize		: packoffset( c3 );
	float4cbColor		: packoffset( c4 );
	matrix	mtxWorld	: packoffset( c5 );
};
```

```
Texture2D txDiffuse : register( t0 );
Texture2D txDepth : register( t1 );
SamplerState samLinear : register( s0 );

//! 頂点属性
struct InputVS
{
	float4pos		: POSITION;
	float2Tex		: TEXCOORD;
};
struct OutputVS
{
	float4pos		: SV_POSITION;
	float2Tex		: TEXCOORD0;
	float4color		: COLOR0;
};

//! 頂点シェーダ
OutputVS RenderVS( InputVS inVert )
{
	OutputVS   outVert;

	float3 pv = mul( float4(inVert.pos * cbvSize.xyz + cbvOffset.xyz, 1), mtxWorld );
	float4 Pos = mul(float4(pv, 1), mtxView);
	outVert.pos = mul(Pos, mtxProj);

	outVert.color = cbColor;

	outVert.Tex = inVert.Tex * cbtSize.xy + cbtOffset.xy;

	return outVert;
}

//! ピクセルシェーダ
float4 RenderPS( OutputVS inPixel ) : SV_TARGET
{
	//深度値の取得
	float d = txDepth.Sample(samLinear, inPixel.Tex).x;
	//深度が閾値以上の場合
	if (d >= 1)
	{
		//25 サンプリングのぼかし
		float4 color = txDiffuse.Sample(samLinear, inPixel.Tex, float2(-2, -2));
		color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(-2, -1));
		color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(-2, 0));
		color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(-2, 1));
		color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(-2, 2));

		color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(-1, -2));
		color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(-1, -1));
		color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(-1, 0));
		color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(-1, 1));
		color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(-1, 2));

		color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(0, -2));
		color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(0, -1));
		color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(0, 0));
		color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(0, 1));
		color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(0, 2));

		color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(1, -2));
		color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(1, -1));
		color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(1, 0));
		color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(1, 1));
		color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(1, 2));
```

```
            color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(2, -2));
            color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(2, -1));
            color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(2, 0));
            color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(2, 1));
            color += txDiffuse.Sample(samLinear, inPixel.Tex, float2(2, 2));
            color /= 25.0f;
            return color * inPixel.color;
        }
        //通常出力
        return txDiffuse.Sample(samLinear, inPixel.Tex) * inPixel.color;
}

technique11 TShader
{
        pass P0
        {
                SetVertexShader(CompileShader(vs_4_0, RenderVS()));
                SetGeometryShader(NULL);
                SetHullShader(NULL);
                SetDomainShader(NULL);
                SetPixelShader(CompileShader(ps_4_0, RenderPS()));
                SetComputeShader(NULL);
        }
}
```

本来はピントより手前の場合もぼかしが必要であり、またぼかしフィルタが高精度であればあるほど、ピント位置とぼかしの境界部分の補間が重要になる。

ゲーム側では深度出力のシェーダーと被写界深度のシェーダーの二つのシェーダーを読み込む。レンダリングターゲットも深度出力とぼかし用の画面の二つが必要でそれぞれの初期化をおこなっている。

Program18-14 GameApp.cpp

```
//カメラ
CCamera                         gCamera;
//カメラ角度
CVector3                        gCamAngle;
//メッシュ
CMeshContainer                  gMesh;
//描画ターゲット
CTexture                        gTarget;
CTexture                        gDepthTarget;
//シェーダー
CShader                         gShader;
CShaderBind_3DPrimitiveBase     gShaderBind;

//シェーダー側のコンスタントバッファと同様の構造体
struct cbGameParam
{
        Vector4     cbProjection;
};

//シェーダー
CShader                         gDofShader;
CShaderBind_SpriteBase          gDofShaderBind;

MofBool CGameApp::Initialize(void){
        //リソース配置ディレクトリの設定
        CUtilities::SetCurrentDirectory("Resource");

        //カメラ初期化
        gCamera.SetViewPort();
        gCamera.LookAt(Vector3(0.0f, 0.0f, -2.0f), Vector3(0, 0, 0), Vector3(0, 1, 0));
```

```
	gCamera.PerspectiveFov(MOF_ToRadian(60.0f),1024.0f / 768.0f,0.01f,1000.0f);
	gCamera.Update();
	CGraphicsUtilities::SetCamera(&gCamera);

	//メッシュの読み込み
	gMesh.Load("player.mom");

	//シェーダーの読み込み
	gShader.Load("Shader.hlsl");
	gShaderBind.Create(&gShader);

	//シェーダーとのバッファの連携
	gShaderBind.CreateShaderBuffer("cbGameParam", sizeof(cbGameParam));

	//深度描画ターゲットを作成する
	MofU32 sw = g_pGraphics->GetTargetWidth();
	MofU32 sh = g_pGraphics->GetTargetHeight();
	gDepthTarget.CreateTarget(sw, sh, PIXELFORMAT_R32_FLOAT,
BUFFERACCESS_GPUREADWRITE);

	//通常の描画ターゲットを作成する
	gTarget.CreateTarget(sw, sh, PIXELFORMAT_R8G8B8A8_UNORM,
BUFFERACCESS_GPUREADWRITE);
	//シェーダーの読み込み
	gDofShader.Load("Dof.hlsl");
	gDofShaderBind.Create(&gDofShader);
	//シェーダーに２Ｄ描画用の行列を設定する
	CGraphicsUtilities::SetScreenSize(sw, sh, &gDofShaderBind);
	//テクスチャオブジェクトを設定
	gDofShaderBind.CreateShaderResource("txDepth");
	return TRUE;
}
```

描画処理ではまず深度を出力し、通常の描画は色出力用のターゲットに切り替えて描画をおこなう。色と深度の二つのテクスチャが作成し、その２枚を使用して被写界深度のシェーダーを実行する。描画で標準のテクスチャ以外のテクスチャを使用するために深度テクスチャをシェーダー側に設定し描画を実行する。

Program18-14 GameApp.cpp

```
MofBool CGameApp::Render(void){
	//描画処理
	g_pGraphics->RenderStart();

	//元の描画ターゲットを取得する
	LPRenderTarget pold = g_pGraphics->GetRenderTarget();

	//作成したテクスチャを描画ターゲットとして設定する
	//深度バッファは元の情報をそのまま使用する
	g_pGraphics->SetRenderTarget(gDepthTarget.GetRenderTarget(),
		g_pGraphics->GetDepthTarget());

	//画面のクリア
	g_pGraphics->ClearTarget(0.0f,0.0f,0.0f,0.0f,1.0f,0);

	//深度バッファ有効化
	g_pGraphics->SetDepthEnable(TRUE);

	//カメラを設定
	gShaderBind.SetCamera(&gCamera);

	//シェーダーに送るバッファを作成
	cbGameParam sb;
	sb.cbProjection.x = 3;
```

```
	//シェーダーにバッファを送る
	gShaderBind.GetShaderBuffer(0)->SetBuffer(&sb);

	//メッシュの描画(深度を出力する)
	CMatrix44 matWorld;
	gMesh.Render(matWorld, &gShader, &gShaderBind);


	//描画ターゲットを色出力用に切り替える
	g_pGraphics->SetRenderTarget(gTarget.GetRenderTarget(), g_pGraphics->GetDepthTarget());

	//画面のクリア
	g_pGraphics->ClearTarget(0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0);

	//メッシュの描画(通常の描画)
	gMesh.Render(matWorld);

	//描画ターゲットを元に戻す
	g_pGraphics->SetRenderTarget(pold, g_pGraphics->GetDepthTarget());

	//2D 描画に変更
	g_pGraphics->SetDepthEnable(FALSE);
	g_pGraphics->SetBlending(BLEND_NONE);

	//画面のクリア
	g_pGraphics->ClearTarget(0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0);

	//描画したターゲットをテクスチャとして画面に描画する
	if (!g_pInput->IsKeyHold(MOFKEY_SPACE))
	{
		//画面を描画する
		CGraphicsUtilities::RenderTexture(0, 0, &gTarget);
		//確認用に画面右上に深度を表示する
		gDepthTarget.Render(CRectangle(0, 0,
			g_pGraphics->GetTargetWidth() / 4, g_pGraphics->GetTargetHeight() / 4));
	}
	else
	{
		//テクスチャの設定
		gDofShaderBind.GetShaderResource(0)->SetResource(&gDepthTarget);
		CGraphicsUtilities::RenderTexture(0, 0, &gTarget, &gDofShader, &gDofShaderBind);
	}
	CGraphicsUtilities::RenderString(10, 10, MOF_COLOR_WHITE, "SPACE キーでシェーダー使用");

	//描画の終了
	g_pGraphics->RenderEnd();
	return TRUE;
}

MofBool CGameApp::Release(void){
	gMesh.Release();
	gTarget.Release();
	gDepthTarget.Release();
	//シェーダーの解放
	gShaderBind.Release();
	gShader.Release();
	gDofShader.Release();
	gDofShaderBind.Release();
	return TRUE;
}
```