Chapter18-15 法線テクスチャの利用 [応用]

ジオメトリのライティングを正確におこなうためには相応の頂点が必要になり負荷が非常に高い。ゲームではリアルタイムのレンダリングを行うためにポリゴン数を減らす必要があり細かなポリゴンの作成は現実的ではない。そこで考えられた手法が法線テクスチャ（法線マップ）である。

法線テクスチャは本来ジオメトリの面（ポリゴン）ごとに存在する法線（面に垂直なベクトル）をテクスチャとして保存したもの。ピクセルシェーダーで法線テクスチャを参照することで平面にも細かな陰影をつけることが可能で少ないポリゴン数でクォリティの高い陰影の実現ではよく使用される手法である。

法線テクスチャを使用した簡易ライティングを紹介する。

Program18-15 Sample.hlsl

```
//! コンスタントバッファ
cbuffer cbSceneParam : register( b0 )
{
	float4		vecViewPos		: packoffset( c0 );
	matrix		mtxView			: packoffset( c1 );
	matrix		mtxProj			: packoffset( c5 );
};

cbuffer cbMeshParam : register( b1 )
{
	matrix		mtxWorld		: packoffset(c0);
	float4		colRevise		: packoffset(c4);
	float4		CoordsRevise		: packoffset(c5);
};

cbuffer cbMaterialParam : register( b2 )
{
	float4		matDiffuse		: packoffset( c0 );
	float4		matAmbient		: packoffset( c1 );
	float4		matSpeculer		: packoffset( c2 );
	float4		matEmissive		: packoffset( c3 );
	float		matPower		: packoffset( c4 );
};

cbuffer cbLightParam : register( b3 )
{
	float3		litDirection		: packoffset( c0 );
	float4		litDiffuse		: packoffset( c1 );
	float4		litAmbient		: packoffset( c2 );
	float4		litSpeculer		: 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;
	float2Tex		: TEXCOORD0;
};
```

```
//新規のテクスチャオブジェクトの作成
Texture2D txNormal : register( t1 );

//! 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 );

	outVert.Tex = inVert.Tex;
	return outVert;
}

//! ピクセルシェーダ
float4 RenderPS( OutputVS inPixel ) : SV_TARGET
{
	//法線をテクスチャから取得する
	//テクスチャの色では０～１なのでー１～１の範囲に変更する
	float4 n = normalize((2.0f * txNormal.Sample(samLinear, inPixel.Tex)) - 1.0f);

	//ハーフランバートライティング
	float l = dot(n, -litDirection);
	l = l * 0.5f + 0.5f;
	l *= l;

	//ライティング結果から色を計算する
	float3 diff = l * litDiffuse.xyz * matDiffuse.xyz;
	float4 LP = float4( saturate( diff ) , matDiffuse.w );
	return LP * txDiffuse.Sample(samLinear, inPixel.Tex);
}

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

ピクセルシェーダーにて法線テクスチャからサンプリングした色情報をライティングのための法線として利用している。テクスチャに保存される色情報は通常の R8G8B8A8 のテクスチャでは 0～255（０～１）の範囲しか保存されないため、そのままでは一方向の法線が実現できない、そこでサンプリングした色情報をー１～＋１の範囲に変換しなおすことで法線としてそのまま活用できる。

ゲーム側ではジオメトリのマテリアルに本来のテクスチャを設定して、本来のテクスチャとは別に法線テクスチャを読み込みシェーダー側に設定する。

Program18-15 GameApp.cpp

```
//カメラ
CCamera					gCamera;
//ライト
CDirectionalLight				gLight;
//メッシュ
LPGeometry				pGeometry;
//テクスチャ
```

```
CTexture                            gTexture;
//シェーダー
CShader                             gShader;
CShaderBind_3DPrimitiveBase         gShaderBind;

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);

        //ライト初期化
        gLight.SetDirection(CVector3(0, -1, -1));
        CGraphicsUtilities::SetDirectionalLight(&gLight);

        //メッシュの読み込み
        pGeometry = CGraphicsUtilities::CreatePlaneGeometry(3, 3, 1, 1, TRUE, Vector3(0, 0, 0));
        //テクスチャの読み込み
        gTexture.Load("isi2_n.png");
        LPTexture pTex = new CTexture();
        pTex->Load("isi2.png");
        pGeometry->GetMaterial()->GetTextureArray()->AddLast(pTex);

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

        //テクスチャオブジェクトを設定
        gShaderBind.CreateShaderResource("txNormal");
        return TRUE;
}
```

Chapter18-16 接ベクトルと従法線ベクトル　［応用］

法線テクスチャを利用したライティングは前の Capter で実現できているが、実際の３D モデル上で正確な凹凸を表現するためにはテクスチャが３D モデル上へのマッピングされているかにより法線の方向を変化させる必要がある。

そこで使用される情報が**接ベクトル**と**従法線ベクトル**である。接ベクトルと従法線ベクトルは頂点の法線ベクトルとテクスチャをマッピングするための UV 情報をもとに計算される、モデルのローカル空間での U 軸（接ベクトル＝テクスチャの横方向）と V 軸（従法線ベクトル＝テクスチャの縦方向）になる。

３D モデルの場合、MomView でモデルの頂点情報に接ベクトルと従法線ベクトルを含めることができる。ここではすでに従法線ベクトルと法線ベクトルが含まれているモデルを使用する。

Program18-16 Sample.hlsl

```
//! コンスタントバッファ
cbuffer cbSceneParam : register( b0 )
{
	float4	vecViewPos		: packoffset( c0 );
	matrix	mtxView			: packoffset( c1 );
	matrix	mtxProj			: packoffset( c5 );
};

cbuffer cbMeshParam : register( b1 )
{
	matrix	mtxWorld		: packoffset(c0);
	float4	colRevise		: packoffset(c4);
	float4	CoordsRevise		: packoffset(c5);
};

cbuffer cbMaterialParam : register( b2 )
{
	float4	matDiffuse		: packoffset( c0 );
	float4	matAmbient		: packoffset( c1 );
	float4	matSpeculer		: packoffset( c2 );
	float4	matEmissive		: packoffset( c3 );
	float	matPower		: packoffset( c4 );
};

cbuffer cbLightParam : register( b3 )
{
	float3	litDirection		: packoffset( c0 );
	float4	litDiffuse		: packoffset( c1 );
	float4	litAmbient		: packoffset( c2 );
	float4	litSpeculer		: packoffset( c3 );
};

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

//! 頂点属性
struct InputVS
{
	float4pos		: POSITION;
	float3	normal		: NORMAL;
	float3	Tangent		: TANGENT;
	float3	Binormal	: BINORMAL;
	float2Tex		: TEXCOORD;
	float4color		: COLOR0;
};
struct OutputVS
{
	float4pos		: SV_POSITION;
	float3	normal		: NORMAL;
```

```
	float3	Tangent		: TANGENT;
	float3	Binormal	: BINORMAL;
	float2Tex		: TEXCOORD0;
};

//新規のテクスチャオブジェクトの作成
Texture2D txNormalTex : register(t1);

//! 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 );

	outVert.normal = normalize(mul(inVert.normal, (float3x3)mtxWorld));
	outVert.Tangent = normalize(mul(inVert.Tangent, (float3x3)mtxWorld));
	outVert.Binormal = normalize(mul(inVert.Binormal, (float3x3)mtxWorld));

	outVert.Tex = inVert.Tex;
	return outVert;
}

//! ピクセルシェーダ
float4 RenderPS( OutputVS inPixel ) : SV_TARGET
{
	//法線をテクスチャから取得する
	//テクスチャの色では 0 ～ 1 なのでー 1 ～ 1 の範囲に変更する
	float3 n = normalize(2.0f * txNormalTex.Sample(samLinear, inPixel.Tex).xyz - 1.0f);
	//法線テクスチャからの結果をモデルのローカル空間に変換する
	n = float3(
		dot(float3(inPixel.Tangent.x,inPixel.Binormal.x,inPixel.normal.x),n),
		dot(float3(inPixel.Tangent.y,inPixel.Binormal.y,inPixel.normal.y),n),
		dot(float3(inPixel.Tangent.z,inPixel.Binormal.z,inPixel.normal.z),n)
	);

	//ハーフランバートライティング
	float l = dot(n, -litDirection);
	l = l * 0.5f + 0.5f;
	l *= l;

	//ライティング結果から色を計算する
	float3 diff = l * litDiffuse.xyz * matDiffuse.xyz;
	float4 LP = float4( saturate( diff ) , matDiffuse.w );
	return LP * txDiffuse.Sample(samLinear, inPixel.Tex);
}

//法線マップを使用しない描画
float4 RenderSimplePS(OutputVS inPixel) : SV_TARGET
{
	//ハーフランバートライティング
	float l = dot(inPixel.normal, -litDirection);
	l = l * 0.5f + 0.5f;
	l *= l;

	//ライティング結果から色を計算する
	float3 diff = l * litDiffuse.xyz * matDiffuse.xyz;
	float4 LP = float4( saturate( diff ) , matDiffuse.w );
	return LP * txDiffuse.Sample(samLinear, inPixel.Tex);
}

technique11 TShader
{
```

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

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

今回のシェーダーでは頂点の情報に接ベクトルと従法線ベクトルを含めている。接ベクトルと従法線ベクトルも法線ベクトルと同様にワールド返還を行いピクセルシェーダーに渡す。
ピクセルシェーダーでは受け取った接ベクトルを X 軸、従法線ベクトルを Y 軸、法線ベクトルを Z 軸としたローカル空間への変換行列として法線テクスチャからの法線を変換している。

また今回は描画結果の比較用に法線テクスチャの使用を行わないレンダリングも同じシェーダー内に追加している。ピクセルシェーダーのみを別の関数を実行するように差し替えるためピクセルシェーダーの関数のみを作成して、technique を定義している。technique は頂点シェーダーやピクセルシェーダーなどを一連の描画の流れを登録する機能で technique を作成しておけばゲーム側で簡単に切り替えて描画を実行できる。

ゲーム側ではシェーダー側の頂点に接ベクトルと従法線ベクトルが増えたため、シェーダーとのバインド用のクラスを変更している。

Program18-16 GameApp.cpp

```
//カメラ
CCamera                         gCamera;
//カメラ角度
CVector3                        gCamAngle;
//メッシュ
CMeshContainer                  gMesh;
//シェーダー
CShader                         gShader;
CShaderBind_BumpMapping         gShaderBind;
```

描画処理ではシェーダーの描画の変更をシェーダー内の technique を変更することで対応している。

Program18-16 GameApp.cpp

```
MofBool CGameApp::Render(void){
	//描画処理
	g_pGraphics->RenderStart();
	//画面のクリア
	g_pGraphics->ClearTarget(0.0f,0.0f,1.0f,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);

	//シェーダーエフェクトを使って描画
	if (g_pInput->IsKeyHold(MOFKEY_SPACE))
	{
		gShader.SetTechnique(0);
	}
	//シンプルなシェーダー描画
	else
	{
		gShader.SetTechnique(1);
	}
	//メッシュの描画
	CMatrix44 matWorld;
	gMesh.Render(matWorld, &gShader, &gShaderBind);

	CGraphicsUtilities::RenderString(10, 10, "'上''下''左''右'キーでカメラ移動");
	CGraphicsUtilities::RenderString(10, 34, "マウス左ドラッグでカメラ回転");
	CGraphicsUtilities::RenderString(10, 58, MOF_COLOR_WHITE, "SPACE キーでシェーダー使用");

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