Chapter 1

プログラマブルシェーダーとは

1.1 固定機能

シェーダーが生まれる前、DirectX9 までは固定機能パイプラインというものが存在していました。この固定機能パイプラインは DirectX10 で削除され、それ以降固定機能パイプラインは用意されなくなっています。これは OpenGL、OpenGLES、Sony や任天堂などが提供する専用 SDK(PS4、PS3、WiiU などで使用できる DirectX のようなもの)でも同じで固定機能はグラフィックプログラミングの世界では過去のものとなっています。では固定機能と呼ばれるものがどのようなものか見ていきましょう。下記は固定機能を使って 3D ポリゴンを表示しているコードです。

```
//-----------------------------------------------------------------------------
// ワールド*ビュー*プロジェクション行列を設定。
//-----------------------------------------------------------------------------
void SetupMatrices()
{
    // ワールド行列を設定。
    D3DXMATRIXA16 matWorld;
    D3DXMatrixIdentity( &matWorld );
    D3DXMatrixRotationX( &matWorld, timeGetTime() / 500.0f );
    g_pd3dDevice->SetTransform( D3DTS_WORLD, &matWorld );

    // ビュー行列を設定。
    D3DXVECTOR3 vEyePt( 0.0f, 3.0f,-5.0f );
    D3DXVECTOR3 vLookatPt( 0.0f, 0.0f, 0.0f );
    D3DXVECTOR3 vUpVec( 0.0f, 1.0f, 0.0f );
    D3DXMATRIXA16 matView;
    D3DXMatrixLookAtLH( &matView, &vEyePt, &vLookatPt, &vUpVec );
    g_pd3dDevice->SetTransform( D3DTS_VIEW, &matView );

    // プロジェクション行列を設定。
    D3DXMATRIXA16 matProj;
    D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI / 4, 1.0f, 1.0f, 100.0f );
    g_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );
}
//-----------------------------------------------------------------------------
// ライトを設定。
//-----------------------------------------------------------------------------
void SetupLights()
{
        //マテリアルを設定。
    D3DMATERIAL9 mtrl;
    ZeroMemory( &mtrl, sizeof( D3DMATERIAL9 ) );
    mtrl.Diffuse.r = mtrl.Ambient.r = 1.0f;
    mtrl.Diffuse.g = mtrl.Ambient.g = 1.0f;
    mtrl.Diffuse.b = mtrl.Ambient.b = 0.0f;
    mtrl.Diffuse.a = mtrl.Ambient.a = 1.0f;
    g_pd3dDevice->SetMaterial( &mtrl );

    // ディフューズライトの向きとカラーを設定。
    D3DXVECTOR3 vecDir;
    D3DLIGHT9 light;
```

```
        ZeroMemory( &light, sizeof( D3DLIGHT9 ) );
        light.Type = D3DLIGHT_DIRECTIONAL;
        light.Diffuse.r = 1.0f;
        light.Diffuse.g = 1.0f;
        light.Diffuse.b = 1.0f;
        vecDir = D3DXVECTOR3( cosf( timeGetTime() / 350.0f ),
                              1.0f,
                              sinf( timeGetTime() / 350.0f ) );
        D3DXVec3Normalize( ( D3DXVECTOR3* )&light.Direction, &vecDir );
        light.Range = 1000.0f;
        g_pd3dDevice->SetLight( 0, &light );
        g_pd3dDevice->LightEnable( 0, TRUE );
        g_pd3dDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
        // アンビエントライトを設定。
        g_pd3dDevice->SetRenderState( D3DRS_AMBIENT, 0x00202020 );
    }
    //-----------------------------------------------------------------------------
    // 描画
    //-----------------------------------------------------------------------------
    VOID Render()
    {
        // バックバッファとZバッファをクリア
        g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                             D3DCOLOR_XRGB( 0, 0, 255 ), 1.0f, 0 );

        // 描画開始。
        if( SUCCEEDED( g_pd3dDevice->BeginScene() ) )
        {
            // ライトとマテリアルを設定。
            SetupLights();
            // ワールドビュープロジェクション行列を設定。
            SetupMatrices();
            // 頂点バッファを設定。
            g_pd3dDevice->SetStreamSource( 0, g_pVB, 0, sizeof( CUSTOMVERTEX ) );
            // 頂点のフォーマットを指定。
            g_pd3dDevice->SetFVF( D3DFVF_CUSTOMVERTEX );
            // 描画。
            g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 2 * 50 - 2 );
            // 描画終了。
            g_pd3dDevice->EndScene();
        }

        // バックバッファの内容を表示。
        g_pd3dDevice->Present( NULL, NULL, NULL, NULL );
    }
```

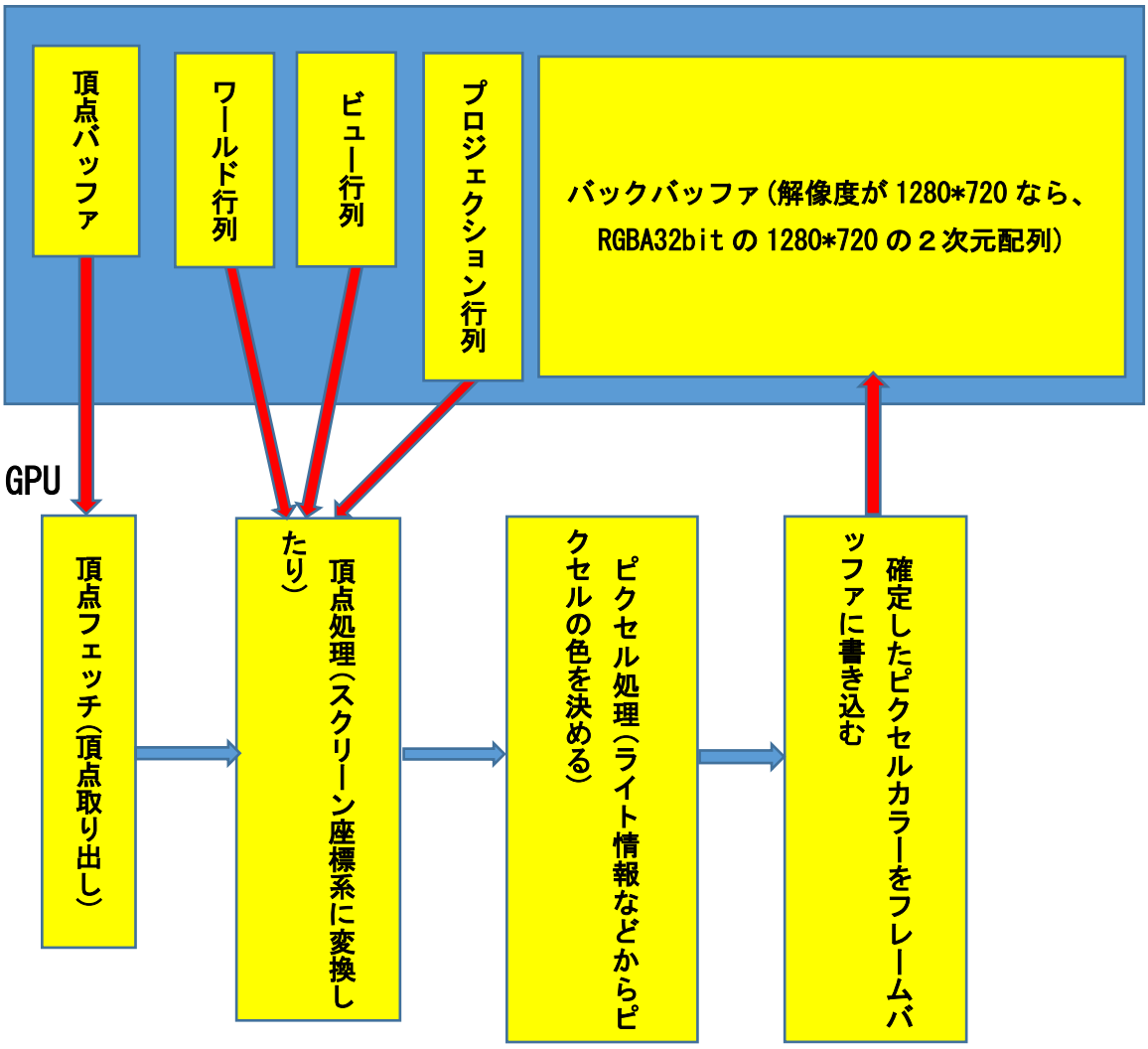
さて、上記のコードでどの部分が固定機能と呼ばれるものか分かりますか？このコードですと、ワールド行列、ビュー行列、プロジェクション行列、マテリアル、ライトの設定が固定機能になります。

では GPU で実行される処理を考えながら、固定機能とはなんなのかを考えていきましょう。

1.1.1 グラフィックスパイプライン

CPU から Draw 命令送られてくると、下記のようにセットされた頂点バッファから頂点をフェッチ(取り出して)して、その頂点に陰影処理を行いスクリーン座標系に変換して対応するピクセルの色を決定します。

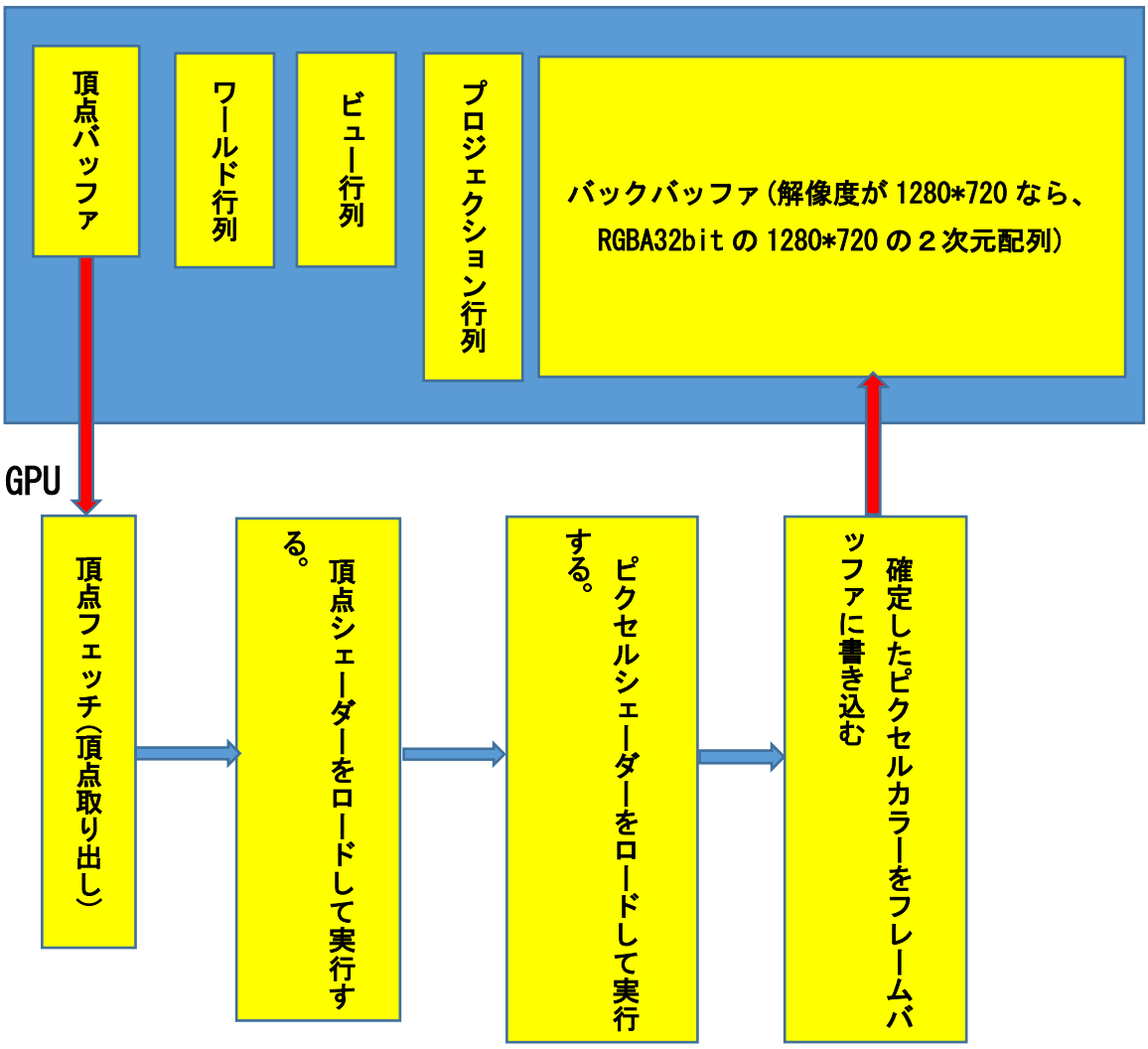
メモリ



このように３次元データから２次元画像を作り出すまでの処理をグラフィックスパイプラインといいます。現在失われた固定機能とはこの図では頂点処理、ピクセル処理がそれにあたります。つまり、現在の GPU は頂点をスクリーン座標に変換する機能とピクセルカラーの決定を自動的に行う機能を用意していないのです！それではどのようにして絵を出しているのでしょうか？頂点をスクリーン座標系に変換して、ポリゴンがスクリーン座標系でどこに位置するのかを決めて、ピクセルカラーを決めないと絵は描けません？そこでシェーダーが登場します。

1.2 シェーダー

シェーダーの導入で先ほど紹介したグラフィックパイプラインの頂点処理とピクセル処理を自分でプログラミングして、自由に頂点処理やピクセル処理を実装することができるようになりました。つまり、自分で頂点をスクリーン座標系へ変換したり、ピクセルカラーを決定するプログラムを書くことになります。つまり DirectX10 以降ではシェーダーを書かないと絵は表示できなくなりました。



この図のように、頂点処理とピクセル処理がシェーダーをロードして実行するという内容に変わっています。ではなぜ固定機能が削除されてシェーダーが登場したのでしょ

うか？せっかく用意されていたものがなくなって、同じものを作らないと絵を出せなくなったなんて面倒だと思いませんか？

1.3 シェーダーが生まれた経緯

固定機能しか存在していなかった DirectX7 まではマイクロソフトが用意したグラフィック表現しか行うことができませんでした。先ほどピクセルカラーを決める方法が固定されているといった話を思い出してみてください。DirectX9 ではせいぜいディフューズライト、スポットライトライト、ポイントライト、アンビエントライトくらいでしょうか？ではこれらの機能を使って下記のようなアニメ調のグラフィック表現が実現できるでしょうか？

アニメ調のグラフィックを実現するためには、特殊なライティングアルゴリズムを実装する必要があります。しかしシェーダーが生まれる前は新しいグラフィック表現を実現するためにはマイクロソフトがその処理を実装するまで待つ必要がありました。また、多数のゲーム開発者の要望に全て答えようとすると DirectX の API がどんどん膨らんで行くことにもなります。(ゲーム開発者というのは他とはことなるユニークな表現を行いたがるものなのです)その要望に答えるためにマイクロソフトは自分たちで処理を実装することを止めて、頂点処理、ピクセル処理を自由にプログラミングできるようにしました。これによりグラフィック表現の幅は大きく広がり、現在の高品質なフォトリアルな表現や、ナリティメットストームのようなノンフォトリアル表現まで多様な表現が実現できるようになったのです。

実はこのアニメ調の表現はプログラマブルシェーダーを書かなくても実現できます。CPU で頂点をロックすれば頂点を自由に加工することができますよね？また、ピクセルカラーも単なる 2 次元

配列に 32bit のピクセルカラーを描き込んでいるだけなので、こちらも CPU でプログラミング可能です。このように CPU でグラフィック処理を行うことをソフトウェアレンダリングといいます。ではなぜわざわざ GPU でプログラミングをするのか？その答えは浮動小数点計算において GPU は CPU に比べて圧倒的に高速に動作するからです。もし興味があれば、一度頂点のスクリーン座標変換を CPU と GPU の両方で実装してみて、速度を比較してみるといいでしょう。CPU の方は目も当てられないような動作速度になるはずです。

Chapter 2

ShaderTutorial_00(最もシンプルなシェーダープログラム)

では実際に簡単なサンプルプログラムを見てみて、シェーダーがどのようなものか見ていきましょう。下記のパスにプログラムを上げていますので Github からコードを pull してください。

https://github.com/KawaharaKiyohara/DirectXLesson/ShaderTutorial_00

Chapter3

ShaderTutorial_01(定数レジスタへの転送)

Chapter2 のサンプルではトランスフォーム済みの頂点(CPU でスクリーン座標系まで変換している頂点のこと)を GPU に送っているため、頂点シェーダーで頂点座標にワールド×ビュー×プロジェクション行列を乗算して、スクリーン座標系に変換するコードはありませんでした。しかし、PC、PS4、XBoxOne のような最新のゲームですと頂点数が 10 万を越えることはザラにあります。この頂点の座標変換を CPU で行うと、まともなパフォーマンスは出ません。そのため、ワールド、ビュー、プロジェクション行列などを転送して、GPU からアクセスできるようにする必要があります。下記のパスにデータの転送を行っているサンプルプログラムをアップしていますので、pull してください。

https://github.com/KawaharaKiyohara/DirectXLesson/ShaderTutorial_01

今回のサンプルでは、頂点カラーに乗算する g_color を CPU から転送しています。(まだワールドビュープロジェクション行列の転送は行っていないため、トランスフォーム済みの頂点で描画を行っています。頂点シェーダーでの座標変換は Chapter 4 で行います)

では、まず CPU からデータを転送する方法について見ていきましょう。

main.cpp

```
VOID Render()
{
    // Clear the backbuffer to a blue color
    g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0, 0, 255), 1.0f, 0);
    if (SUCCEEDED(g_pd3dDevice->BeginScene()))
    {
        //シェーダー適用開始。
        g_pEffect->SetTechnique("ColorPrim");
        g_pEffect->Begin(NULL, D3DXFX_DONOTSAVESHADERSTATE);
        g_pEffect->BeginPass(0);
        //定数レジスタに設定するカラー。
        D3DXVECTOR4 color( 1.0f, 0.0f, 0.0f, 1.0f);
        //シェーダー側のシェーダー定数の名前で、データの転送先を指定する。
        g_pEffect->SetVector("g_color", &color);
        //この関数を呼び出すことで、データの転送が確定する。描画を行う前に一回だけ呼び出す。
        g_pEffect->CommitChanges();
        g_pd3dDevice->SetStreamSource(0, g_pVB, 0, sizeof(SVertex));
        g_pd3dDevice->SetFVF(D3DFVF_CUSTOMVERTEX);
        g_pd3dDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);
        g_pEffect->EndPass();
        g_pEffect->End();
        // End the scene
        g_pd3dDevice->EndScene();
    }

    // Present the backbuffer contents to the display
    g_pd3dDevice->Present(NULL, NULL, NULL, NULL);
}
```

太字で書いている箇所が GPU へのデータの転送命令を行っている箇所になります。今回はシェーダー側に記述されているシェーダー定数名を指定してデータを転送する方法を採用しています。文字列で転送先の検索が行われるため重いのですが、今回は分かりやすさを重視しています。試しにローカル変数の color の値を変更してみてください。ポリゴンの色が変わるはずです。では続いてシェーダー側のソースを見てみましょう。

basic.fx

```
float4 g_color;          //カラー。これがシェーダー定数。CPU から値が転送されてくる。

struct VS_INPUT{
    float4    pos         : POSITION;
    float4    color       : COLOR0;
};

struct VS_OUTPUT{
    float4    pos         : POSITION;
    float4    color       : COLOR0;
};

/*!
 *@brief   頂点シェーダー。
 */
VS_OUTPUT VSMain( VS_INPUT In )
{
    VS_OUTPUT Out;
    Out.pos = In.pos;
    Out.color = In.color * g_color;   //頂点カラーに定数レジスタに設定されたカラーを乗算してみる。
    return Out;
}
```

太字になっている箇所が CPU から送られたデータに関連する箇所になります。GPU には定数レジスタという高速にアクセスできるメモリがあり、CPU から送られたデータはこの定数レジスタにバインドされます。C++などのグローバル変数のような定義のされ方がしている変数が定数レジスタにバインドされる変数になります。定数レジスタには上限があり、ライトなども定数レジスタに送るため DirectX9 世代ではライトの本数などに上限が設けられていました。DirectX11 からは(10 からそうだったのかも？)ストラクチャバッファなどの機能が増え、事実上この上限はなくなっています。

では、下記の二つの実習を行ってみてください。

・カラーの定数 g_addColor を追加して、頂点シェーダーで加算合成を行う。

・カラーの定数 g_mulColor を追加して、頂点シェーダーで乗算合成を行う。

Chapter 4

ShaderTutorial02(ワールドビュープロジェクション行列による頂点変換)

Chapter3 でも予告していましたが、今度はワールドビュープロジェクション行列(以下 WVP 行列)を使用して、頂点シェーダーで頂点変換を行っていきます。サンプルプログラムを下記のパスにアップしていますので、pull を行ってください。

https://github.com/KawaharaKiyohara/DirectXLesson/ShaderTutorial_02

Chapter3 で学んだように、WVP 行列は CPU から GPU へ転送を行って、シェーダーで使用することになります。では CPU 側の転送命令を記述しているコードを見てみましょう。

main.cpp

```
VOID Render()
{
    // Clear the backbuffer to a blue color
    g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0, 0, 255), 1.0f, 0);
    if (SUCCEEDED(g_pd3dDevice->BeginScene()))
    {
        //シェーダー適用開始。
        g_pEffect->SetTechnique("ColorPrim");
        g_pEffect->Begin(NULL, D3DXFX_DONOTSAVESHADERSTATE);
        g_pEffect->BeginPass(0);
        //ワールド行列の転送。
        g_pEffect->SetMatrix("g_worldMatrix", &g_worldMatrix);
        //ビュー行列の転送。
        g_pEffect->SetMatrix("g_viewMatrix", &g_viewMatrix);
        //プロジェクション行列の転送。
        g_pEffect->SetMatrix("g_projectionMatrix", &g_projectionMatrix);
        //この関数を呼び出すことで、データの転送が確定する。描画を行う前に一回だけ呼び出す。
        g_pEffect->CommitChanges();
        g_pd3dDevice->SetStreamSource(0, g_pVB, 0, sizeof(SVertex));
        g_pd3dDevice->SetFVF(D3DFVF_CUSTOMVERTEX);
```

```
        g_pd3dDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);
        g_pEffect->EndPass();
        g_pEffect->End();
        // End the scene
        g_pd3dDevice->EndScene();
    }
    // Present the backbuffer contents to the display
    g_pd3dDevice->Present(NULL, NULL, NULL, NULL);
}
```

太字になっている箇所が GPU への転送命令を記述している箇所です。先ほどと大きな違いはないかと思います。転送するのが行列なので、関数名が SetMatrix になっている点が違うくらいでしょうか。ではシェーダー側のソースを見てみましょう。

basic.fx

```
float4x4 g_worldMatrix;         //ワールド行列。
float4x4 g_viewMatrix;          //ビュー行列。
float4x4 g_projectionMatrix;    //プロジェクション行列。

struct VS_INPUT{
    float4    pos         : POSITION;
    float4    color       : COLOR0;
};

struct VS_OUTPUT{
    float4    pos         : POSITION;
    float4    color       : COLOR0;
};

/*!
 *@brief   頂点シェーダー。
 */
VS_OUTPUT VSMain( VS_INPUT In )
{
    VS_OUTPUT Out;
    float4 pos;
    pos = mul( In.pos, g_worldMatrix );       //モデルのローカル空間からワールド空間に変換。
    pos = mul( pos, g_viewMatrix );           //ワールド空間からビュー空間に変換。
    pos = mul( pos, g_projectionMatrix );     //ビュー空間から射影空間に変換。
    Out.pos = pos;
    Out.color = In.color;
    return Out;
}
```

黒字になっている箇所が WVP 行列を使用したコードになります。Hlsl では float4x4 が行列の変数です。頂点シェーダーで mul 命令を使用して、頂点の座標変換を行っています。注意点としては実は mul 関数は下記のように記述することもできます。

pos = mul(g_worldMatrix, In.pos);

サンプルと比較して行列とベクトルの順番が変わっているのがわかりますでしょうか。この順番を入れ替えると異なる結果になるので注意が必要です。

第一引数にベクトルがある場合は行ベクトルとして計算されます。第二引数にベクトルがある場合は列ベクトルとして計算されます。今はとりあえず、結果が変わるということだけ覚えておいてください。今後シェーダーを記述していくことがあるかと思いますが、意図しない表示なっている場合などは、乗算する順番がおかしくなっていないか確認してみてく

ださい。
では下記の実習を行ってみてください。
・WVP 行列の計算を CPU で行って、GPU での計算コストを減らしてください。

Chapter 5

ShaderTutorial_03(シェーダーを使用して X ファイルを表示)

シェーダーを使用してモデルを表示するサンプルを下記のパスにアップしていますので pull を行ってください。
https://github.com/KawaharaKiyohara/DirectXLesson/ShaderTutorial_03
シェーダーを使用したモデル表示も今までの方法と大きな違いはないため、ここはサンプルの紹介にとどめておきます。

Chapter 6

ShaderTutorial_04(簡単なテクスチャ貼り付け)

Chapter5 のサンプルでは虎にテクスチャが貼られていませんでした。今回は虎にテクスチャを貼り付ける処理を説明します。今までは IDirect3DDevice9::SetTexture を実行すれば勝手にテクスチャが貼られていたと思いますが、シェーダーを使用する場合はテクスチャの貼り方までプログラミングする必要があります。ではサンプルを見ていきましょう。
まず、GPU にテクスチャのアドレスを転送しているコードです。

main.cpp

```
VOID Render()
{
    // Clear the backbuffer to a blue color
    g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, D3DCOLOR_XRGB(0, 0,
255), 1.0f, 0);
    static int renderCount = 0;
    if (SUCCEEDED(g_pd3dDevice->BeginScene()))
    {
        // Turn on the zbuffer
        g_pd3dDevice->SetRenderState(D3DRS_ZENABLE, TRUE);

        renderCount++;
        D3DXMATRIXA16 matWorld;
        D3DXMatrixRotationY( &g_worldMatrix, renderCount / 500.0f );
        //シェーダー適用開始。
        g_pEffect->SetTechnique("SkinModel");
        g_pEffect->Begin(NULL, D3DXFX_DONOTSAVESHADERSTATE);
        g_pEffect->BeginPass(0);
        //定数レジスタに設定するカラー。
        D3DXVECTOR4 color( 1.0f, 0.0f, 0.0f, 1.0f);
        //ワールド行列の転送。
        g_pEffect->SetMatrix("g_worldMatrix", &g_worldMatrix);
```

```
        //ビュー行列の転送。
        g_pEffect->SetMatrix("g_viewMatrix", &g_viewMatrix);
        //プロジェクション行列の転送。
        g_pEffect->SetMatrix("g_projectionMatrix", &g_projectionMatrix);
        g_pEffect->CommitChanges();
        // Meshes are divided into subsets, one for each material. Render them in
        // a loop
        for( DWORD i = 0; i < g_dwNumMaterials; i++ )
        {
            g_pEffect->SetTexture("g_diffuseTexture", g_pMeshTextures[i]);
            // Draw the mesh subset
            g_pMesh->DrawSubset( i );
        }

        g_pEffect->EndPass();
        g_pEffect->End();

        // End the scene
        g_pd3dDevice->EndScene();
    }

    // Present the backbuffer contents to the display
    g_pd3dDevice->Present(NULL, NULL, NULL, NULL);
}
```

黒字の部分が GPU へテクスチャアドレスの転送を行っているコードです。これは固定機能を使っている時と大差ないのではないでしょうか。ではシェーダーを見ていきます。

basic.fx

```
/*!
 *@brief      簡単なテクスチャ貼り付けシェーダー。
 */


float4x4 g_worldMatrix;              //ワールド行列。
float4x4 g_viewMatrix;               //ビュー行列。
float4x4 g_projectionMatrix;         //プロジェクション行列。

texture g_diffuseTexture;            //ディフューズテクスチャ。    ①
sampler g_diffuseTextureSampler =    //テクスチャサンプラ          ②
sampler_state
{
    Texture = <g_diffuseTexture>;
    MipFilter = NONE;
    MinFilter = NONE;
    MagFilter = NONE;
    AddressU = Wrap;
    AddressV = Wrap;
};

struct VS_INPUT{
    float4  pos     : POSITION;
    float4  color   : COLOR0;
    float2  uv      : TEXCOORD0;
};

struct VS_OUTPUT{
    float4  pos     : POSITION;
    float4  color   : COLOR0;
    float2  uv      : TEXCOORD0;

};

/*!
 *@brief      頂点シェーダー。
```

```
 */
VS_OUTPUT VSMain( VS_INPUT In )
{
	VS_OUTPUT Out;
	float4 pos;
	pos = mul( In.pos, g_worldMatrix );		//モデルのローカル空間からワールド空間に変換。
	pos = mul( pos, g_viewMatrix );			//ワールド空間からビュー空間に変換。
	pos = mul( pos, g_projectionMatrix );	//ビュー空間から射影空間に変換。
	Out.pos = pos;
	Out.color = In.color;
	Out.uv = In.uv;
	return Out;
}
/*!
 *@brief	頂点シェーダー。
 */
float4 PSMain( VS_OUTPUT In ) : COLOR
{
	return tex2D( g_diffuseTextureSampler, In.uv );	//③
}

technique SkinModel
{
	pass p0
	{
		VertexShader	= compile vs_2_0 VSMain();
		PixelShader		= compile ps_2_0 PSMain();
	}
}
```

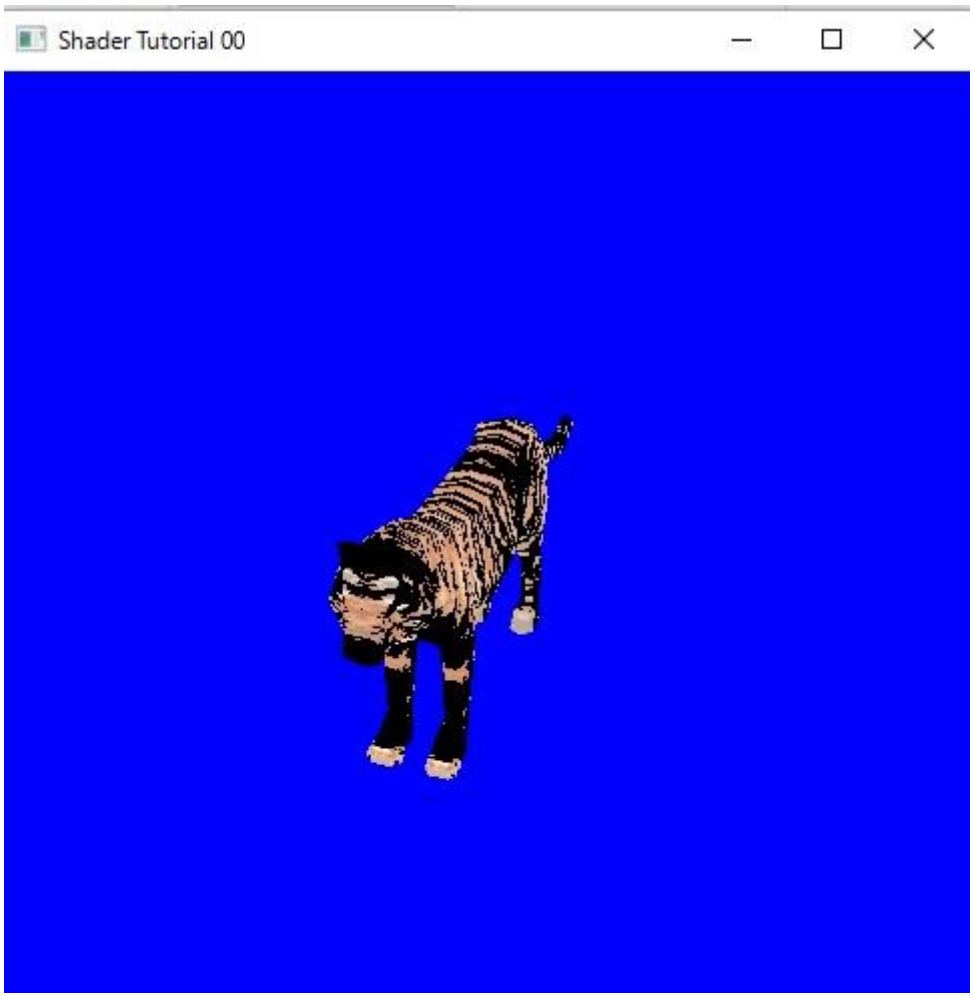
まず①の箇所が CPU から転送されたテクスチャのアドレスが格納された変数になります。そして②の箇所がテクスチャのサンプリング方法を記述したテクスチャサンプラと呼ばれるものです。テクスチャサンプラについてはここでは説明しません。今はこのように記述するものだと思っていてください。続いて③の箇所ですが、こちらが uv 座標を使用してテクスチャをサンプリングしているコードになります。

実習

① サンプリングしたテクスチャカラーの明るさが 1.0 以上であればテクスチャカラーを出力して、1.0 以下であれば黒を出力するようにピクセルシェーダーを改造してみてください。テクスチャカラーの明るさは下記のコードを使用して求めてください。

length(float3 color)

下記のような絵になります。



② トラのテクスチャが UV 座標の U 方向にスクロールするプログラムを実装してください。

実装した結果の実行ファイルを下記のパスにアップしていますので、こちらを参考に実装してみてください。

DirectXLesson¥ShaderTutorial_04¥UV スクロール

③ 虎を半透明で表示してみてください。

④ 虎を点滅させてください。

Chapter 7

深度テスト(Z テスト)

このチャプターではシェーダーではなく、3D グラフィックスのプログラミングでは欠かすことが出来ない、深度テスト(Z テスト)について解説をします。

7.1 深度バッファ(Z バッファ)

まず、深度テストについて説明をする前に深度バッファについて説明をします。みなさん DirectX9 で 3D モデルを表示する際に、デバイスの初期化で下記のような処理を記述していたのではないかと思います。

```
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory(&d3dpp, sizeof(d3dpp));
d3dpp.Windowed = TRUE;
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
d3dpp.BackBufferFormat = D3DFMT_UNKNOWN;
d3dpp.EnableAutoDepthStencil = TRUE;
d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
// Create the D3DDevice
if (FAILED(g_pD3D->CreateDevice(
        D3DADAPTER_DEFAULT,
        D3DDEVTYPE_HAL,
        hWnd,
        D3DCREATE_SOFTWARE_VERTEXPROCESSING,
        &d3dpp,
        &g_pd3dDevice
)))
{
        return E_FAIL;
}
```

d3dpp は CreateDevice に渡す引数なのですが、この赤字になっている箇所が深度バッファに関する設定になっています。この設定を渡すと 16 ビットの深度バッファがフレームバッファと同じ幅と高さで作成されます。例えば 1280×720 のフレームバッファを作成した場合は、下記のような深度バッファが作成されます。

```
short depthBuffer[720][1280];      //深度バッファ
```

深度バッファとは、フレームバッファに絵を書き込んだ際に、その絵の Z の座標を記録しておくためのバッファです。例えば図 A のような絵をフレームバッファに書き込んだ場合図 B のような深度バッファが作成されます。

図 A フレームバッファ

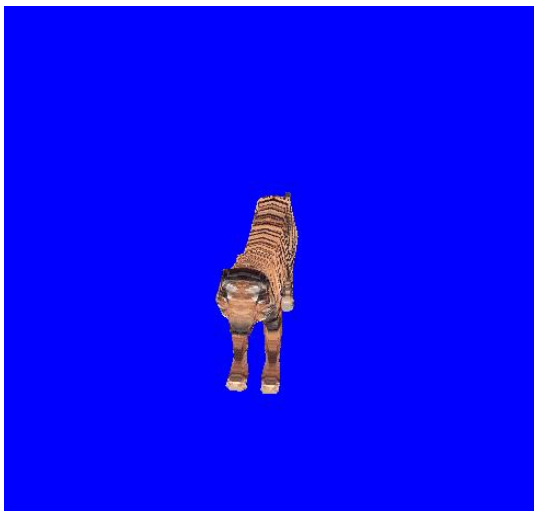
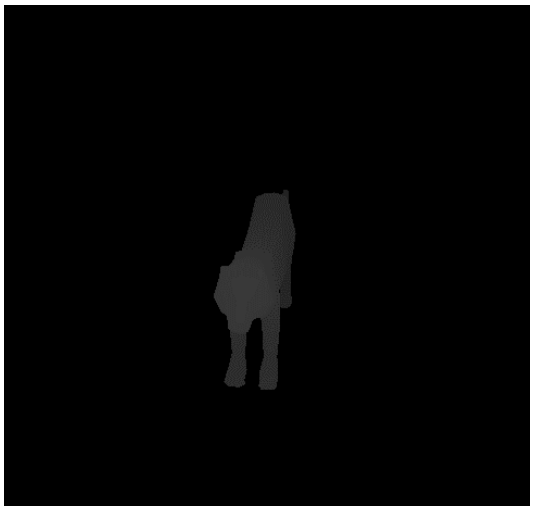


図 B 深度バッファ



7.2 深度テスト

深度テストとは頂点シェーダーとピクセルシェーダーの間で行われる処理で、深度バッファを参照して、新しく書き込もうとするピクセルが既に書き込まれているピクセルより手前にあるか、奥にあるかを判定するテストになります。これから書き込もうとしているピクセルが既に書き込まれているピクセルより奥にある場合、描きこむ必要がないため処理が破棄されます。この深度テストのおかげで 3D オブジェクトは正しい前後関係で描画することができるようになっています。

7.3 深度テストを有効にする方法

では DirectX9 で深度テストを有効にする方法を見ていきましょう。DirectX9 では各種レンダリングステートの設定を行う、IDirect3DDevice9::SetRenderState を使用することで深

度テストを有効にすることができます。下記のようなコードで深度テストを有効にできます。

```
g_pd3dDevice->SetRenderState(D3DRS_ZENABLE, TRUE);
```

7.4 Z ファイティング

深度バッファに保存できる Z 値の精度には限界があります。そのため非常に Z の値が近いポリゴンを重ねて描画すると、ポリゴンがチラ付く現象が発生します。これを Z ファイティングといいます。

PSP は 16bit の深度バッファしか作ることができずに、深度バッファの精度が低かったため、PSP のゲームではよくこの現象が起きていました。

下記のパスに Z ファイティングのサンプルプログラムをアップしています。
https://github.com/KawaharaKiyohara/DirectXLesson/Z ファイティングサンプル

7.5 ZFunc

実は Z テストは必ずしも手前にあるものだけを描画するわけではありません。実は ZFunc というレンダリングステートを変更すると奥にあるものを描画するということもできます。この ZFunc というのは Z テストの方法を指定するレンダリングステートで、「Z 値が大きいものが合格」や、「Z 値が小さければ合格」というふうに Z テストの方法を変更することができます。

下記に ZFunc に指定できる値を列挙します。

D3DCMP_NEVER
　　テストは常に失敗する。
D3DCMP_LESS
　　新しいピクセル値が、現在のピクセル値より小さいときに応じる。
D3DCMP_EQUAL
　　新しいピクセル値が、現在のピクセル値と等しいときに応じる。
D3DCMP_LESSEQUAL
　　新しいピクセル値が、現在のピクセル値以下のときに応じる。
D3DCMP_GREATER
　　新しいピクセル値が、現在のピクセル値より大きいときに応じる。
D3DCMP_NOTEQUAL
　　新しいピクセル値が、現在のピクセル値と等しくないときに応じる。
D3DCMP_GREATEREQUAL
　　新しいピクセル値が、現在のピクセル値以上のときに応じる。
D3DCMP_ALWAYS
　　テストは常にパスする。

Chapter 8

アルファブレンディング

このチャプターでは半透明合成や、加算合成を行うために必要なアルファブレンディングについて説明します。

8.1 アルファブレンディングとは

アルファブレンディングとは、ピクセルシェーダーで計算されたカラー(RGBA)をフレームバッファにどのように描き込むのかを指定するものとなります。その描き込みの際にアルファ値を使用して描き込み方を決定するため、アルファブレンディングと言われます。アルファブレンディングは Z テストと同様に IDirect3DDevice9::SetRenderState を使用して設定することができます。

アルファブレンディングを有効にするには下記のようなコードを記述します。

```
g_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
```

8.2 半透明合成

アルファブレンディングにおいて、これから描きこもうとしているカラーをソースカラー(SRC)といいます。そして既にフレームバッファに描きこまれているカラーのことをデスティネーションカラー(DEST)といいます。半透明合成はソースアルファ(SRC_α)を使用してソースカラーとデスティネーションカラーを混ぜ合わせることで実現されています。

半透明合成の計算式は下記のようになります。

描き込まれるカラー ＝SRC×SRC_α+DEST×(1.0f－SRC_α)

半透明合成を行うためのレンダリングステートの設定は下記のようなコードを記述します。

```
//ソースカラーにはソースアルファを乗算する設定。
g_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
//デスティネーションカラーには 1.0f-ソースアルファを乗算する設定。
g_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
```

8.3 加算合成

光のエフェクト、炎のエフェクト、斬撃のエフェクトなど、光り輝くようなエフェクトは全て加算合成で実現されています。加算合成とは名前のとおり、色の加算になります。そのためポリゴンが重なれば重なるほど、白に近い色になっていきます。下記のようなエフェクトを実現するためには加算合成を行う必要があります。

加算合成は下記の計算式で実現されます。

描き込まれるカラー ＝SRC×1.0f+DEST×1.0f

加算合成を行うためのレンダリングステートの設定は下記のようになります。

```
//ソースカラーにはソースアルファを乗算する設定。
g_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);
//デスティネーションカラーには1.0f-ソースアルファを乗算する設定。
g_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);
```

8.4 実習

下記のサンプルプログラムを使用して、下記の実習を行いなさい。

https://github.com/KawaharaKiyohara/DirectXLesson/アルファブレンディング実習

① 重なって表示されているポリゴンを半透明合成できるようにしなさい。

② 重なって表示されているポリゴンを加算合成できるようにしなさい。

Chapter 9

ShaderTutorial_05(ディフューズライト)

このチャプターでは拡散反射光(ディフューズライト)を行います。ライティングの計算はシェーダープログラムの醍醐味の一つになります。ディフューズライトの計算式は下記のようになります。

ライトの方向　＝L(x,y,z)

ライトのカラー ＝C(r,g,b)

頂点の法線 ＝N(x,y,z)

内積を求める関数 ＝dot()

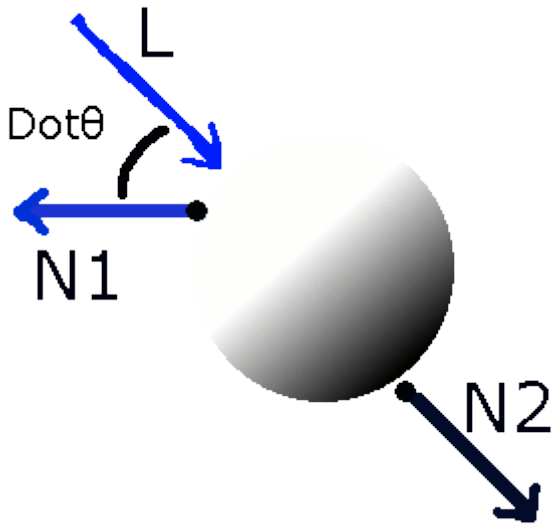
最大値を取得する関数 ＝max()

とした時に下記の計算式でライトのカラーが求まります。

max(0,dot(-L, N)) * C

まず、内積の性質について説明します。内積は長さ 1 のベクトル(単位ベクトル)同士で計算した場合、同じ向きを向いているベクトルで計算すると 1 という結果を返します。また、直行しているベクトル(なす角が 90 度)で計算をすると 0 という結果を返します。最後に真逆を向いているベクトルで計算すると-1 を返します。

つまり、ライトの向き*-1 と法線の向きが同じ場合(ライトを真正面から浴びている)は 1 を返し、ライトの向き*-1 と法線の向きが直行している場合は 0 を返し、ライトの向き*-1 と法線の向きが逆向きの場合は-1 を返します。この結果をライトのカラーに乗算すると下記のような見え方になります。



L がライトの方向、N が法線です。

今回のサンプルではライトを 4 本使用しています。ライトを複数用意する理由は下記の絵を見るとイメージしやすいのではないでしょうか。

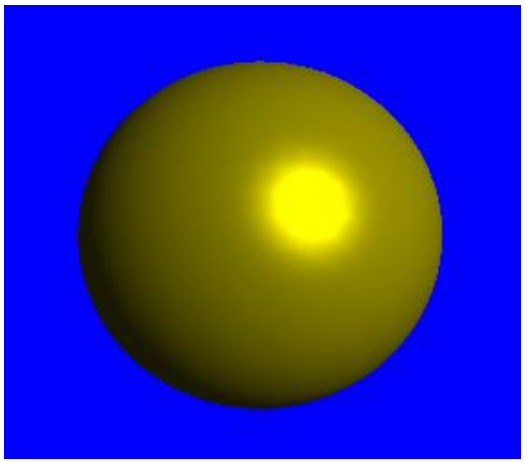
1

2　３Ｄモデルを綺麗に見せるためにライトの設定が重要なことが分かるかと思います。

3　アイドルや女優さんなどの写真集の撮影現場をイメージしてみてください。

Chapter 10 スペキュラライト

前節では太陽などのような平行光源からのライティングをシミュレーションするディフューズライティングを学びました。この節では金属や人肌や髪の毛のツヤなどをミュレーションするスペキュラライトを学びましょう。

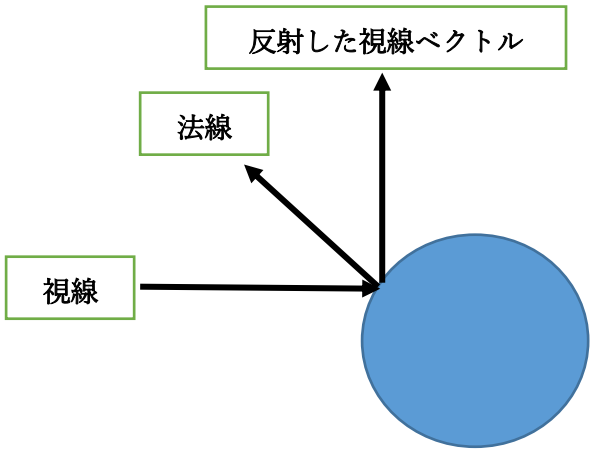


10.1 フォン鏡面反射光

スペキュラライトの計算の仕方はいくつかあるのですが、ここではフォン鏡面反射と呼ばれる手法を学びましょう。

ハイライトのような映り込みは、光源から入射した光が反射する角度から見た時が最も強く反射されます。つまり、視線が物体に当たって跳ね返った方向が、ライトの方向に近い時に明るく見えるのです。

図 1　視線が物体に反射したベクトル



では、視線ベクトルを反射させるベクトルの計算の仕方を求めてみましょう。反射ベクトルの計算に必要な要素は下記の二つになります。

・視線ベクトル

・当たった面の法線

まず、視線ベクトルを求めてみましょう。視線ベクトルは視点－頂点座標で求めたベクトルを正規化することで計算することができます。この視線ベクトルを E、当たった面の法線を N とすると、反射ベクトル R は下記の計算で求まります。

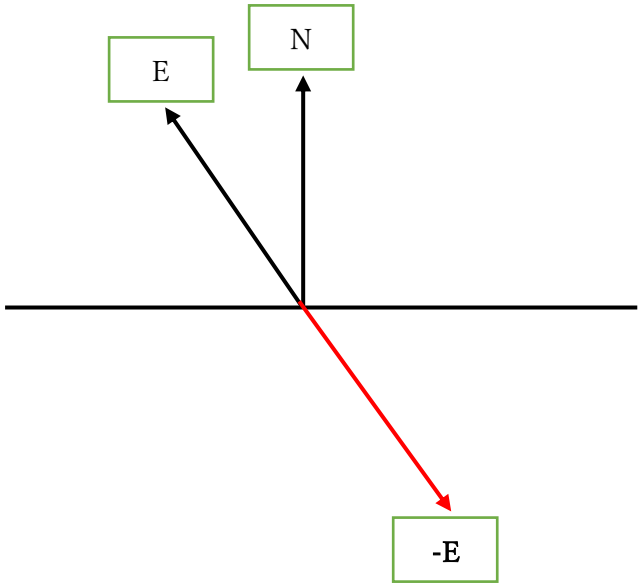
$$R = -E+2\times(N\cdot E)\times N$$

10.2 反射ベクトル

反射ベクトルの求め方は、先ほどの計算を使って求めればいいのですが、それほど難解な式ではありませんので、なぜあの式で反射ベクトルが求まるのか考えてみましょう。

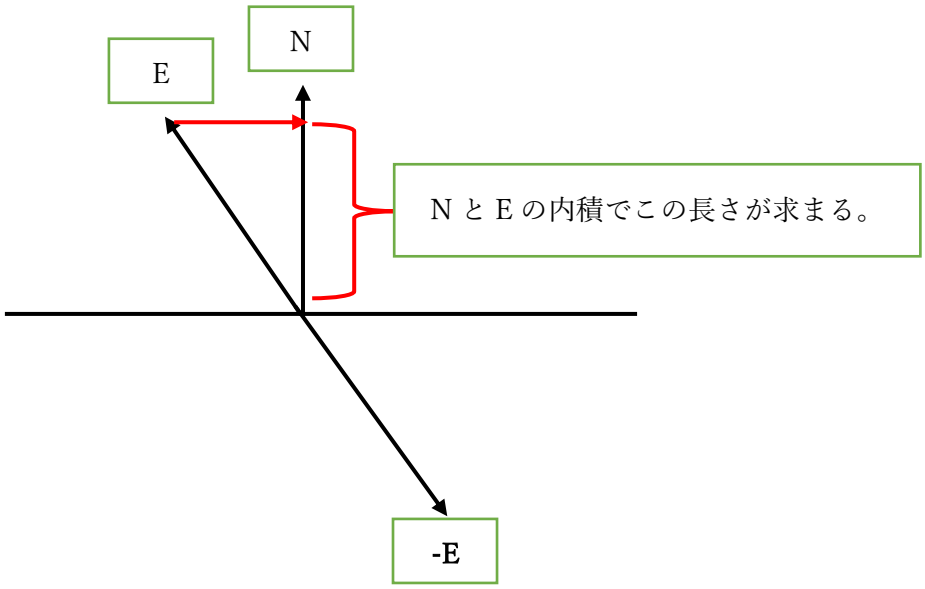
$$-E+2\times(N\cdot E)\times N$$

まず、網掛けになっている-E がどういうベクトルなのか、下記の図を見て確認してみて下さい。



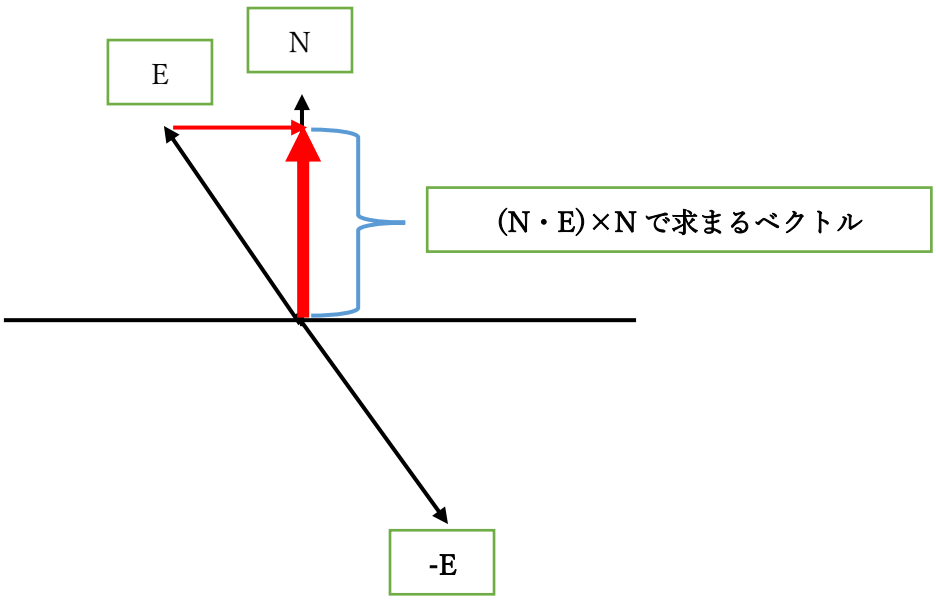
-E＋2×(N・E)×N

では続いて、網掛けになっている、(N・E)を見ていきましょう。N・E で法線と視線ベクトルの内積を求めています。法線は向きだけを保持している大きさ 1 の正規化されたベクトルです。正規化されたベクトルとの内積を計算すると、そのベクトルに射影した時の長さが求まります。



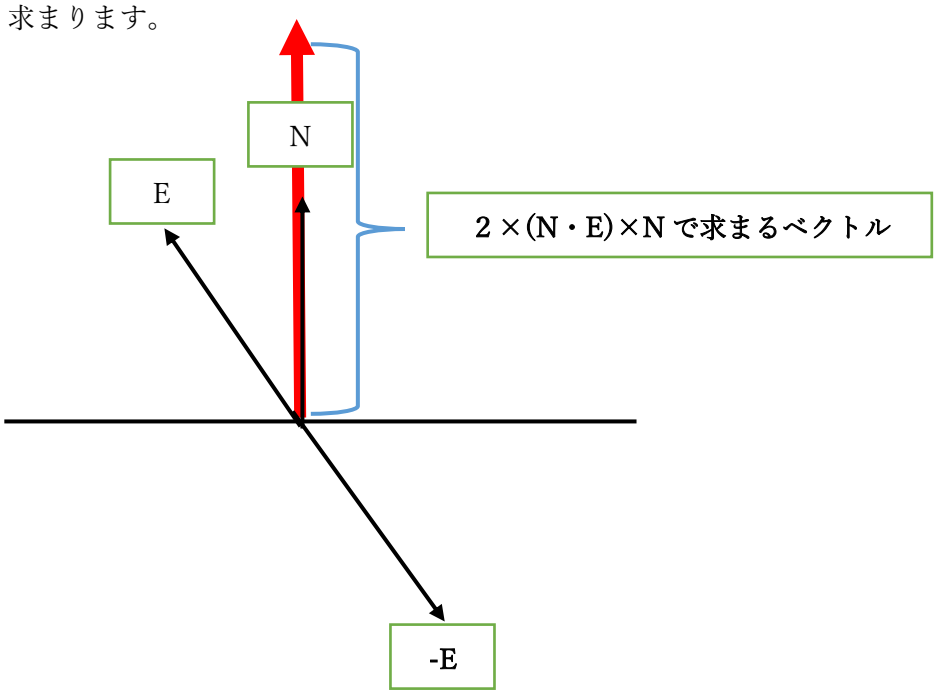
-E＋2×(N・E)×N

そして、**(N・E)×N** で、N・E で求めた長さを法線N に乗算していますので、下記のベクトルが求まります。

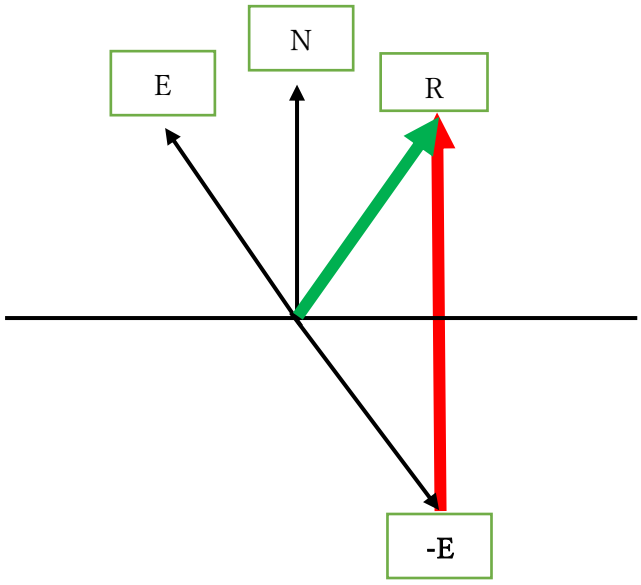


-E+2×(N・E)×N

そして、2×(N・E)×N から上で求めたベクトルを 2 倍しているので、下記のベクトルが求まります。



そして、上で求まったベクトルを-E に加算しているので、下記のベクトルが求まり、反射ベクトルが求まることになります。



10.3 スペキュラ反射の計算

では、本題のスペキュラ反射の計算に戻りましょう。スペキュラ反射は 10.2 で求めた視線の反射ベクトルがライトの方向と近いほど強く反射することになります。それをシミュレーションするために、下記の内積の性質を使用します。

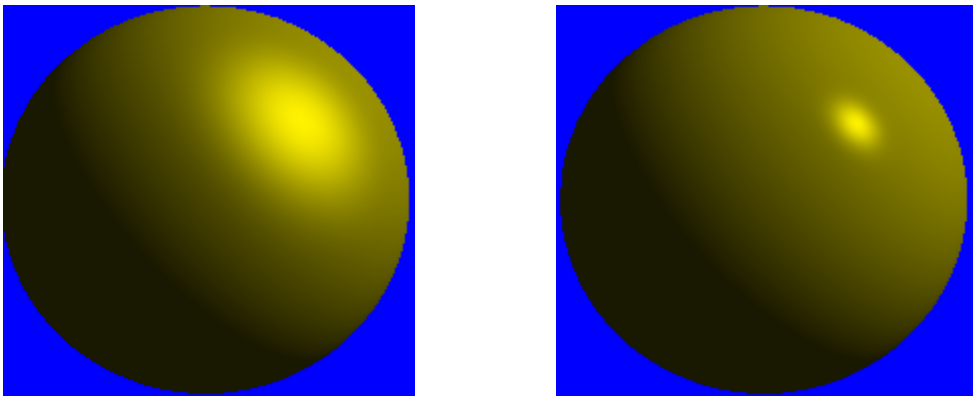
・正規化されたベクトル同士の内積を計算すると、同じ向きのベクトルなら 1 を返し、直交する場合は 0 を返す。

つまり、ライトの方向を L、視線の反射ベクトルを R とした場合、-L・R は向きが同じなら 1 を返して、ベクトルが離れていくほど小さい値を返すようになりますので、これをスペキュラ光として利用すればうまくいきそうですね。

10.4 スペキュラの絞り

では、最後にスペキュラの絞りについて見てみましょう。スペキュラの絞りとはハイライトの大きさを調整するためのものとなります。

絞り 5　　絞り 50

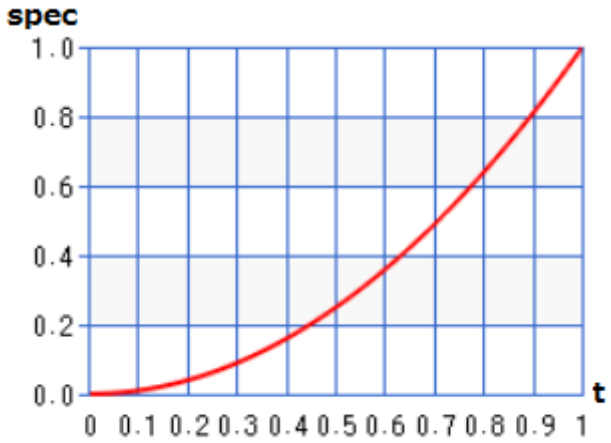


絞りは pow 関数を使用することでシミュレーションできます。10.3 で求めたスペキュラ光を下記のように pow 関数に渡してみましょう。

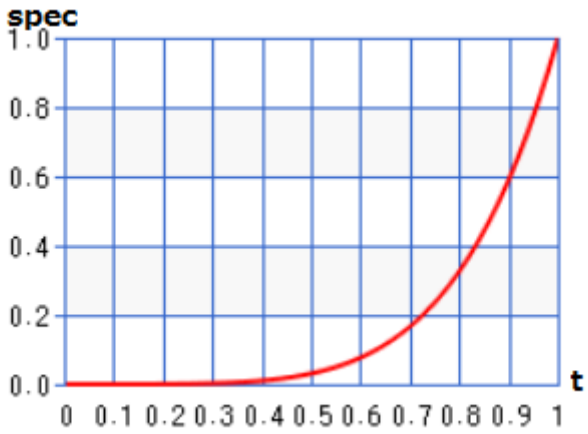
```
float t = max( 0.0f, dot( -L, R ) );　//10.3 で求めたスペキュラ光。
float3 spec = pow( t, 10.0f);　　//pow 関数を使用して、最終的なスペキュラ光を求める。
```

pow 関数は累乗を求める関数です。上記のコードですと、t^{10}を求めていることになります。
t は大きさ同士のベクトルで内積を計算しているので 0～1.0 を返してきます
では、t の累乗を求めるとどのような結果を返すのか下記のグラフをみて確認してみましょう。

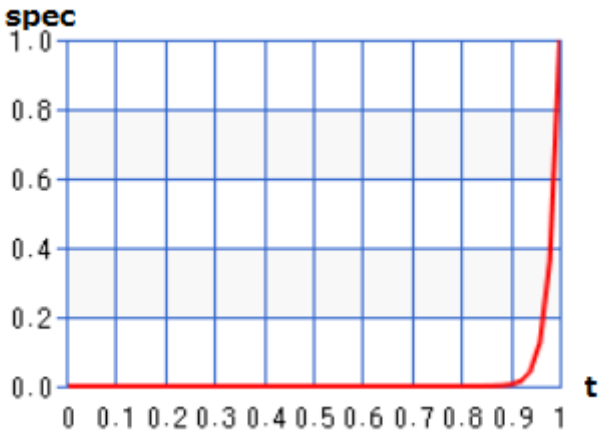
絞り 2 : pow(t,2.0f)



絞り 5 : pow(t, 5.0f)



絞り 50 : pow(t, 50.0f)



このように、絞りが大きいほど急なカーブを描くようになるため、ハイライトが小さくなります。

10.5 まとめ

スペキュラ反射を計算する手順を纏めます。

① 視点―ワールド頂点座標で求まるベクトルを正規化して、視線ベクトルを求める。（視点は eyePos、ワールド頂点座標は vertexPos とする）

```
float3 eye = normalize(eyePos – vertexPos);
```

② 視線ベクトルの反射ベクトルを求める。（ワールド頂点法線を vertexNormal とする）

```
float3 R = -eye + 2.0f * dot(vertexNormal, eye) * vertexNormal;
```

③ 反射ベクトルとライトのベクトルとの内積を計算してスペキュラ光を求める。（ライトの方向を L とする）

```
float3 spec = max( 0.0f, dot(R, -L ) );
```

④ 3 で求めたスペキュラ光に絞りを適用させる。

```
spec = pow(spec, 2.0f);
```

10.6 実習課題

DirextXLesson/ShaderTutorial_06/kadai/ShaderTutorial_06.sln を使用して、スペキュラライトを実装しなさい。

Chapter 11 オフスクリーンレンダリング

このチャプターではブルーム、被写界深度、SSSSS、影生成など最新の 3D グラフィックスになくてはならない表現を行うための基礎知識となるオフスクリーンレンダリングについて見ていきましょう。

11.1 レンダリングターゲット

レンダリングターゲットとは 3D モデルなどの描画先のことです。今まであまり意識してこなかったかもしれませんが、最終的に画面上に表示される絵はフレームバッファにレンダリングされた結果が表示されており、これもレンダリングターゲットになります。
オフスクリーンレンダリングとは、フレームバッファとは別に新しくレンダリングターゲットを作成して、そこにレンダリングを行う手法のことを言います。スクリーンに描画を行わないのでオフスクリーンレンダリングと呼ばれます。

11.2 レンダリングターゲットの作成

ではレンダリングターゲットの作成の仕方を見ていきましょう。まず、レンダリングターゲットとなるテクスチャを作成します。

```
//レンダリングターゲットとなるテクスチャを作成。
g_pd3dDevice->CreateTexture(
    1280,                       //テクスチャの幅
    720,                        //テクスチャの高さ
    1,                          //ミップマップの数。今は 1 でいい。
    D3DUSAGE_RENDERTARGET,      //このテクスチャをレンダリングターゲットとして使用することを明示する。
    D3DFMT_A8R8G8B8,            //テクスチャのフォーマット。ここでは A8R8G8B8 の 32bit を指定。
    D3DPOOL_DEFAULT,            //テクスチャメモリの確保の仕方。D3DPOOL_DEFAULT でいい。
    &m_texture,                 //テクスチャのアドレスを格納するポインタのアドレス。
    NULL                        //常に NULL を指定。
);
```

続いて、このテクスチャからサーフェイスを取得する。これはレンダリングターゲットを切り替えるときに必要となるもので、こういうものだと覚えておきましょう。

```
//サーフェイスの取得。
m_texture->GetSurfaceLevel(0,&m_surface);
```

最後にデプスステンシルバッファの作成を行います。このバッファは必ずしも必要なわけではありませんが、レンダリングを行う際に 3D モデルの前後関係の判定が必要な場合は作成しましょう。

```
g_pd3dDevice->CreateDepthStencilSurface(
    1280,                       //デプスステンシルバッファの幅。必ずテクスチャと同じ解像度にする。
    720,                        //デプスステンシルバッファの高さ。必ずテクスチャと同じ解像度にする。
    D3DFMT_D24X8,               //深度バッファのフォーマット。深度バッファに 24bit 使用する。
    D3DMULTISAMPLE_NONE,    //マルチサンプリングの設定。今回は D3DMULTISAMPLE_NONE でいい。
    0,                      //画像の品質レベル。今回は 0 でいい。
    TRUE,                   //TRUE でいい。
    &m_depthSurface,        //デプスステンシルサーフェイスのアドレスを格納するポインタのアドレス。
    NULL                    //NULL でいい。
);
```

11.3 レンダリングターゲットの切り替え

ではレンダリングターゲットを切り替えて、オフスクリーンレンダリングを行うコードを見ていきましょう。

まず、後でもとに戻す必要があるので切り替える前に、現在のレンダリングターゲットを記録しておきましょう。

```
LPDIRECT3DSURFACE9 renderTargetBackup;
LPDIRECT3DSURFACE9 depthBufferBackup;
//元々のレンダリングターゲットを保存。後で戻す必要があるので。
g_pd3dDevice->GetRenderTarget(0, &renderTargetBackup);
//元々のデプスステンシルバッファを保存。後で戻す必要があるので。
g_pd3dDevice->GetDepthStencilSurface(&depthBufferBackup);
```

続いてレンダリングターゲットを切り替えます。

```
g_pd3dDevice->SetRenderTarget(0, m_surface);
g_pd3dDevice->SetDepthStencilSurface(m_depthSurface);
//書き込み先を変更したのでクリア。
g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, D3DCOLOR_XRGB(255, 255, 255), 1.0f, 0);
```

では、このレンダリングターゲットに対して、何か描画を行ってみましょう。描画に関しては特別なことを行う必要はありません。

```
g_pTigerEffect->SetTechnique("SkinModel");
g_pTigerEffect->Begin(NULL, D3DXFX_DONOTSAVESHADERSTATE);
g_pTigerEffect->BeginPass(0);

//定数レジスタに設定するカラー。
D3DXVECTOR4 color( 1.0f, 0.0f, 0.0f, 1.0f);
D3DXMATRIX mTigerMatrix;
D3DXMatrixRotationY( &mTigerMatrix, renderCount / 10.0f );
//ワールド行列の転送。
g_pTigerEffect->SetMatrix("g_worldMatrix", &mTigerMatrix);
```

```
//ビュー行列の転送。
g_pTigerEffect->SetMatrix("g_viewMatrix", &g_viewMatrix);
//プロジェクション行列の転送。
g_pTigerEffect->SetMatrix("g_projectionMatrix", &g_projectionMatrix);
g_pTigerEffect->CommitChanges();

for( DWORD i = 0; i < g_dwNumMaterials; i++ )
{
        g_pTigerEffect->SetTexture("g_diffuseTexture", g_pMeshTextures[i]);
        // Draw the mesh subset
        g_pMesh->DrawSubset( i );
}
g_pTigerEffect->EndPass();
g_pTigerEffect->End();
```

最後にレンダリングターゲットを元に戻します。

```
g_pd3dDevice->SetRenderTarget(0, renderTargetBackup);          //戻す。
g_pd3dDevice->SetDepthStencilSurface(depthBufferBackup);       //戻す。
```

11.4 実習課題

ShaderTutorial_10/kadai を使用して、虎のオフスクリーンレンダリングを行い、レンダリングしたテクスチャを板ポリに張り付けなさい。板ポリの描画は CSprite クラスを使いなさい。

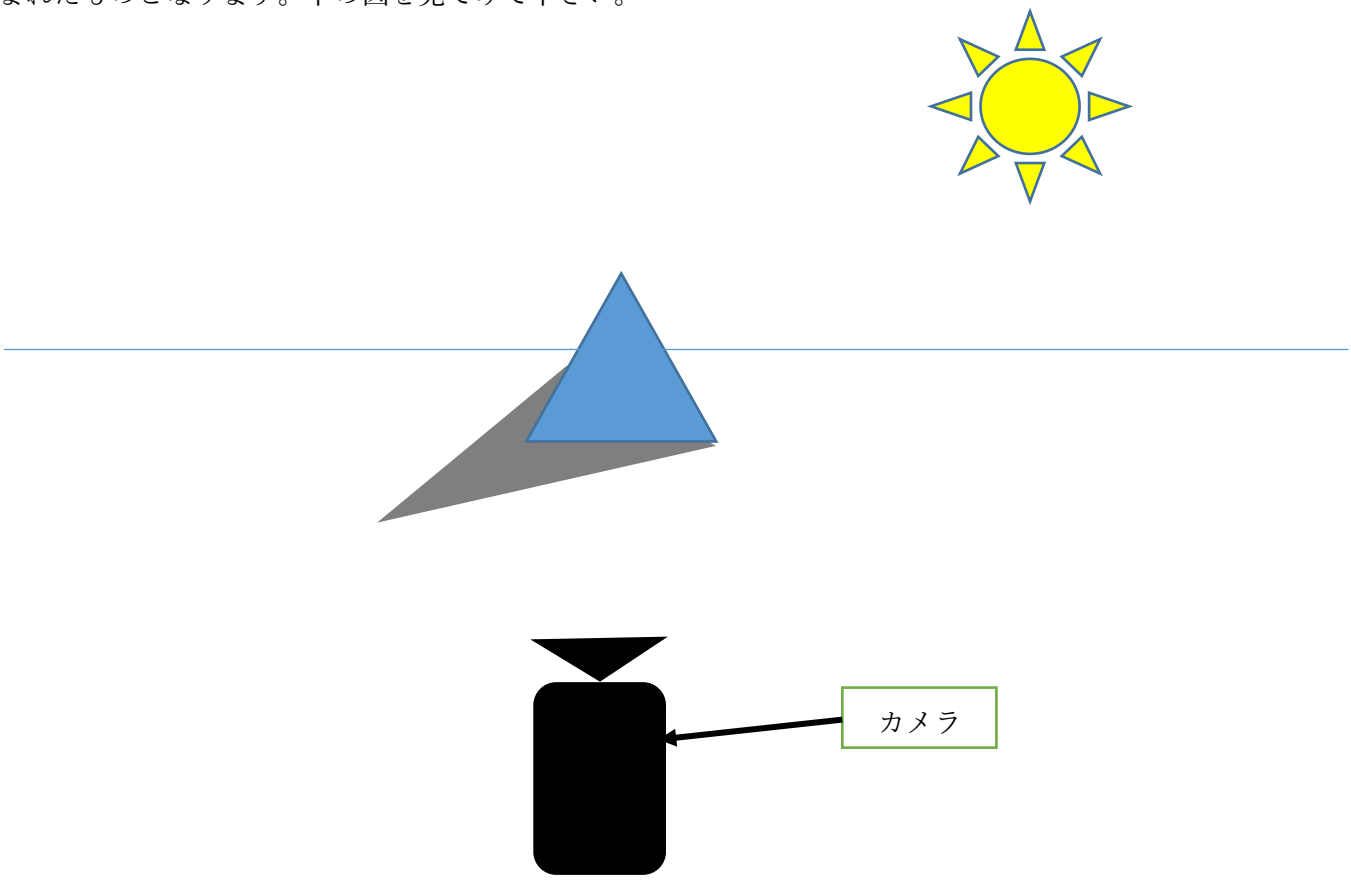
解答例となる実行ファイルは ShaderTutorial_10/kadai/ShaderTutorial_10/Answer.exe にあります。

Chapter 12 投影シャドウ

３Ｄゲームにおいて、 リアルなグラフィックを実現するために影は非常に重要な要素になります。また、グラフィック面だけではなく、影は３Ｄオブジェクトの空間上の位置をユーザーに教えるための重要な要素となります。
このチャプターでは、基本となる古典的な影生成アルゴリズムの投影シャドウについて見ていきます。

12.1 シャドウマップ

投影シャドウはシャドウマップと呼ばれるテクスチャを使用して、影を落とすアルゴリズムの一つになります。シャドウマップというのは影が落ちる場所がテクスチャに描きこまれたものとなります。下の図を見てみて下さい。



このように、影というのは物体によって光が遮られている箇所に発生するものだということが分かります。これは言い方を変えると、**光を放っているライトの位置から見たときに、手前に物体が存在していれば影が落ちる**と言うことができます。下記の絵を見てみて下さい。

キャラクターの影が岩に落ちているのが分かります。では、これをライトの方向から見た場合どうなるでしょうか？次の絵を見てみて下さい。

ライトの方向から見た絵

このように、ライトの位置から見てみると、影が落ちていた岩の部分は、キャラクターに遮られていることが分かります。この**ライトから見た絵**がシャドウマップです。

12.2　ShadowMap クラス

　では、シャドウマップはどうやって作るのか？勘のいい人ならすでに気付いているかもしれませんが、これはチャプター11 で勉強したオフスクリーンレンダリングを行うことで作成することが出来ます。つまり、ライトをカメラと見立てて影を生成したいオブジェクト(シャドウキャスターと呼ばれます)を、シャドウマップに対してレンダリングしてやればいいのです。では ShaderTutorial11 の ShadowMap.cpp と ShadowMap.h を開いてください。これはシャドウマップをクラス化したものになります。ShadowMap クラスには下記のメンバ変数があります。

ShadowMap.h

```
CRenderTarget     renderTarget;      //シャドウマップを書きこむレンダリングターゲット。
D3DXMATRIX        lightViewMatrix;   //ライトビューマトリクス。
D3DXMATRIX        lightProjMatrix;   //ライトプロジェクションマトリクス。
D3DXVECTOR3       iewPosition;       //ライトビューの視点。
D3DXVECTOR3       viewTarget;        //ライトビューの注視点。
```

シャドウマップを書きこむためのレンダリングターゲットや、ライトをカメラと見立てるためのパラメータがあるのが分かるかと思います。

では、ShadowMap::Update 関数を見てみましょう。

```
//ライトビュープロジェクション行列を更新。
```

```
//普通のカメラと同じ。
//カメラの上方向を決める計算だけ入れておく。
D3DXVECTOR3 tmp = viewTarget - viewPosition;
D3DXVec3Normalize(&tmp, &tmp);
if (fabsf(tmp.y) > 0.9999f) {
   //カメラがほぼ真上 or 真下を向いている。
   D3DXMatrixLookAtLH(&lightViewMatrix, &viewPosition, &viewTarget, &D3DXVECTOR3(1.0f, 0.0f, 0.0f));
}
else {
   D3DXMatrixLookAtLH(&lightViewMatrix, &viewPosition, &viewTarget, &D3DXVECTOR3(0.0f, 1.0f, 0.0f));
}
D3DXMATRIXA16 matProj;
D3DXMatrixPerspectiveFovLH(&lightProjMatrix, D3DXToRadian(60.0f), 1.0f, 0.1f, 100.0f);
```

カメラの処理と全く同じです。ライトの位置を視点、そして注視点を使うことでライトの向きも決まります。あとは、カメラ行列とプロジェクション行列を作成するだけです。

では、最後に ShadowMap::Draw を見てみましょう。

```
LPDIRECT3DSURFACE9 renderTargetBackup;
LPDIRECT3DSURFACE9 depthBufferBackup;
//元々のレンダリングターゲットを保存。後で戻す必要があるのでバックアップを行う。
g_pd3dDevice->GetRenderTarget(0, &renderTargetBackup);
//元々のデプスステンシルバッファを保存。後で戻す必要があるので。
g_pd3dDevice->GetDepthStencilSurface(&depthBufferBackup);

//レンダリングターゲットを変更する。
g_pd3dDevice->SetRenderTarget(0, renderTarget.GetRenderTarget());
g_pd3dDevice->SetDepthStencilSurface(renderTarget.GetDepthStencilBuffer());
//書き込み先を変更したのでクリア。
g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, D3DCOLOR_XRGB(255, 255,
255), 1.0f, 0);
//トラをシャドウマップにレンダリング。
g_tiger.Draw(&lightViewMatrix, &lightProjMatrix, true, false);

g_pd3dDevice->SetRenderTarget(0, renderTargetBackup);          //戻す。
g_pd3dDevice->SetDepthStencilSurface(depthBufferBackup);       //戻す。
```

チャプター11 で見た処理と大差ありません。レンダリングターゲットをシャドウマップに変更して、そこにシャドウキャスターをレンダリングしています。

実習課題

ここまでの内容を参考にして、ShaderTutorial11/kadai を使用しライトから見た虎をシャドウマップにレンダリングできるようにしなさい。シャドウマップにレンダリングができているかどうかの確認が取れないと思うので、できたはずと思った時点で指導教員に確認を取るようにして下さい。

12.3 シェーダーテクニック

さて、前節でシャドウキャスターをライトから見た絵をシャドウマップにレンダリングすることが出来るようになりました。下記のようなシャドウマップが作成できたはずです。

ライトの方向から見た虎がレンダリングされています。しかし、投影シャドウで使用するシャドウマップは下記のようにグレースケールになっている必要があります。
(この理由は後述します。)

この節ではトラをグレースケールでレンダリングするためにシェーダーテクニックの切り替えを行っています。レンダリングターゲットに描きこまれるピクセルカラーを決めているのはピクセルシェーダーでしたね？ つまり、ピクセルシェーダーを書き換えればグレースケールでレンダリングできることになります。
今回は虎をレンダリングするためのピクセルシェーダーに通常レンダリング用と、シャドウマップへ描きこむ用の二つを用意して、それを用途に合わせて切り替えてグレースケールの描画を行っています。

では、ShaderTutorial11/basic.fx を見て下さい。96 行目～117 行目に下記のような記述が

あります。

```
/*!
 *@brief    通常描画用のテクニック
 */
technique SkinModel
{
    pass p0
    {
        VertexShader    = compile vs_3_0 VSMain();
        PixelShader     = compile ps_3_0 PSMain();
    }
}
/*!
 *@brief    シャドウマップ書き込み用のテクニック
 */
technique SkinModelRenderToShadowMap
{
    pass p0
    {
        VertexShader    = compile vs_3_0 VSMain();
        PixelShader     = compile ps_3_0 PSRenderToShadowMapMain();
    }
}
```

SkinModel テクニックでは頂点シェーダーに VSMain、ピクセルシェーダーに PSMain を使用すると記述されている。

SkinModel テクニックでは頂点シェーダーに VSMain、ピクセルシェーダーに PSRenderToShadowMapMain を使用すると記述されている。

このようにシェーダーテクニックが二つ記述されています。SkinModel が通常描画用のテクニック、SkinModelRenderToShadowMap がシャドウマップへの書き込み用のテクニックです。この二つのテクニックを見比べてみると、使用するピクセルシェーダーが異なっているのが分かります。では、PSRenderToShadowMap と PSMain を見比べてみましょう。

```
/*!
 *@brief    ピクセルシェーダー。
 */
float4 PSMain( VS_OUTPUT In ) : COLOR
{
    //テクスチャからカラーをフェッチして、それを出力している。
    float4 color = tex2D( g_diffuseTextureSampler, In.uv );
    return color;
}
/*!
 *@brief    シャドウマップ書き込み用のピクセルシェーダー。
 */
float4 PSRenderToShadowMapMain(VS_OUTPUT In) : COLOR
{
    //灰色のカラーを出力している。
    return float4(0.5f, 0.5f, 0.5f, 1.0f);
}
```

通常描画で使用されている PSMain はテクスチャカラーを出力していますが、シャドウマップへの書き込みで使用される PSRenderToShadowMapMain は灰色を出力しています。

これでシェーダー側の対応は完了です。次は CPU 側でどのようにシェーダーテクニックを切り替えているのか見てみましょう。ShaderTutorial11/Model.cpp の CModel::Draw 関数

を開いてください。

```
//シェーダー適用開始。
if (!isDrawShadowMap) {
    //通常描画
    g_pModelEffect->SetTechnique("SkinModel");
}else {
    //シャドウマップへの書き込みテクニック。
    g_pModelEffect->SetTechnique("SkinModelRenderToShadowMap");
}
```

ID3DXEffect::SetTechnique でテクニックを切り替えることができます。今回のサンプルでは、CModel::Draw 関数の引数の isDrawShadowMap を使用して、シェーダーテクニックの切り替えを行っています。

実習課題

シャドウマップをグレースケールで作成できるように変更しなさい。

12.4 シャドウマップの貼り付け

前節でグレースケールのシャドウマップが作成できました。これで投影シャドウに使用するシャドウマップは完成したことになります。あとは、影を落とすオブジェクト(シャドウレシーバーと言われます)にシャドウマップを張り付けてやればよいことになります。では、どうやって張り付けるのか見ていきましょう。

例えば下記の絵の岩を画面に描画する際は、**岩の頂点座標×ワールド行列×カメラ行列×射影行列**という計算を行ってスクリーン座標に変換します。シャドウマップを張り付ける場合は先ほどの変換とは別に**岩の頂点座標×ワールド行列×ライトカメラ行列×ライト射影行列**という計算を行って、ライトをカメラと見立てた座標系への変換も行う必要があります。つまり岩に影を落とす場合、下記の２パターンの座標変換を行うことになります。

カメラから見た絵

ライトから見た絵

では、具体的にコードを見てみましょう。ShaderTutorial_11/basic.fx の 54 行目～70 行目

を見て下さい。

```
VS_OUTPUT VSMain( VS_INPUT In )
{
    VS_OUTPUT Out = (VS_OUTPUT)0;
    float4 worldPos;
    worldPos = mul( In.pos, g_worldMatrix );            //モデルのローカル空間からワールド空間に変換。
    Out.pos = mul( worldPos, g_viewMatrix );            //ワールド空間からビュー空間に変換。
    Out.pos = mul( Out.pos, g_projectionMatrix );       //ビュー空間から射影空間に変換。
    Out.color = In.color;
    Out.uv = In.uv;
    if(g_isShadowReciever == 1){
        //シャドウレシーバー。
        //ワールド座標をライトカメラから見た射影空間に変換する。
        Out.lightViewPos = mul(worldPos, g_lightViewMatrix);
        Out.lightViewPos = mul(Out.lightViewPos, g_lightProjectionMatrix);
    }
    return Out;
}
```

g_isShadowRecieverの値が1になっている場合はOut.lightViewPosにライトカメラから見た射影空間での座標が計算されています。これが頂点座標をライトから見た頂点に変換しているコードになります。

(注意：lightViewPosのセマンティクスがTEXCOORD1になっているのは大丈夫なの？と思われるかもしれませんが。シェーダープログラマにとって、このセマンティクスは飾りです。TEXCOORDという名前ですが、UV座標以外の色々な計算結果をピクセルシェーダーに渡すときに利用されます。)

12.4.1 正規化座標系

では、ここで射影変換された頂点というのはどのような座標系になっているか復習してみましょう。例えば射影変換の結果、下記のような絵が描画された場合、頂点座標は下記のような縦横-1.0～1.0の座標系に変換されています。

(-1.0, 1.0)　　(1.0, 1.0)

(-1.0, -1.0)　　(1.0, -1.0)

このような座標系を正規化されたスクリーン座標系と呼ばれます。これがシャドウマップを張り付ける際に重要になってきますので、しっかりと抑えておいてください。

12.4.2　UV 座標に変換

では、シャドウマップの貼り付けの話に戻りましょう。先ほど頂点シェーダーでシャドウレシーバーのオブジェクトの頂点座標にライトビュープロジェクション行列を乗算して、座標変換を行いました。そうすると岩のオブジェクトは下記のように座標変換されます。

(-1.0, 1.0)　　(1.0, 1.0)

(-1.0, -1.0)　　(1.0, -1.0)

仮にこの人間のキャラクターをシャドウキャスターとした場合にどのようなシャドウマップが生成されているか見てみましょう。

(0.0, 0.0)　　(1.0, 0.0)

(0.0, 1.0)　　(1.0, 1.0)

このようなシャドウマップが生成されています。この二つの画像を見比べてみると、座標の範囲の違いはありますが、岩の影が落ちる部分はシャドウマップにマッピングするとちょうど灰色になっていることが分かります。
つまり、ライトビュープロジェクション行列で変換した座標(-1～1)を UV 座標(0～1)に変換して、それを使ってシャドウマップを張り付けてやればうまくいきそうです。

座標の変換は下記のように行います。
　ライトビュープロジェクション行列で変換された座標を V、計算結果の格納先を UV とした場合に。
　UV = V × 0.5　　// -1～1 の範囲に 0.5 をかけるので、-0.5～0.5 の範囲になる。
　UV = UV + 0.5　　//-0.5～0.5 の範囲に 0.5 を加算するので、0.0～1.0 の範囲になる。

では実際のコードを見ていきましょう。

```
float4 PSMain( VS_OUTPUT In ) : COLOR
{
    float4 color = tex2D( g_diffuseTextureSampler, In.uv );
    if(g_isShadowReciever == 1){
        //射影空間(スクリーン座標系)に変換された座標は w 成分で割ってやると(-1.0f～1.0)の
        //範囲の正規化座標系になる。
        //これを UV 座標系(0.0～1.0)に変換して、シャドウマップをフェッチするための UV として活用する。
        //この計算で(-1.0～1.0)の範囲になる。
        float2 shadowMapUV = In.lightViewPos.xy / In.lightViewPos.w;
        //この計算で(-0.5～0.5)の範囲になる。
        shadowMapUV *= float2(0.5f, -0.5f);
        //そしてこれで(0.0～1.0)の範囲になってＵＶ座標系に変換できた。
        shadowMapUV += float2(0.5f, 0.5f);
        //シャドウマップは影が落ちているところはグレースケールになっている。
        float4 shadowVal = tex2D(g_shadowMapTextureSampler, shadowMapUV);
        color *= shadowVal;
  }
  return color;
}
```

ShaderTutorial_11/basic.fx の 74 行目～87 行目までを見て下さい。
網掛けになっている箇所がシャドウマップの貼り付けを行っているコードです。シャドウマップからフェッチしてきたカラーを出力カラーに乗算しているコードを見てみて下さい。
影が落ちているところはグレースケールになっているため、出力カラーが暗くなるのが分かるかと思います。これがシャドウマップをグレースケールにした理由です。

実習課題
シャドウマップを地面に貼り付けて影を落としなさい。

Chapter 13 物理エンジンを活用した衝突判定

このチャプターでは BulletPhysics という物理エンジンを活用した衝突判定について見ていきます。

13.1 物理エンジン

3D ゲームにおいて衝突判定は非常に難しい分野のプログラムだったのですが、物理エンジンの登場によって、その難易度は大幅に下がりました。物理エンジンで有名なものは HavokPhysics、PhysX が挙げられます。ただし、基本的な使い方はどのエンジンも大きくは変わりません。この節では、衝突判定の話に行く前に物理エンジンとはどういうものなのか簡単に説明します。

物理エンジンにはいくつか機能的な分類があります。軟体物理、破壊物理などなど。ここでは剛体物理エンジンについて見ていきます。剛体とは形の変わらない物体のことを指します。剛体物理エンジンを使うためには物理ワールドを作成する必要があります。そして、ユーザーは剛体を生成して、そのインスタンスを物理ワールドに登録することで物理シミュレーションを行うことができます。下記は簡単な剛体シミュレーションの行い方です。

① 物理ワールドの初期化

```
//コリジョンコンフィギュレーションを作成。
collisionConfig = new btDefaultCollisionConfiguration();
//コリジョンディスパッチャーを作成。
collisionDispatcher = new btCollisionDispatcher(collisionConfig);
//ブロードフェーズの作成。
overlappingPairCache = new btDbvtBroadphase();
//コンストレイントソルバーの作成。
constraintSolver = new btSequentialImpulseConstraintSolver;
//物理ワールドの作成。
dynamicWorld = new btDiscreteDynamicsWorld(
        collisionDispatcher,
        overlappingPairCache,
        constraintSolver,
        collisionConfig
        );
//重力を設定。
dynamicWorld->setGravity(btVector3(0, -10, 0));
```

続いて、剛体を生成して物理ワールドに登録するやり方です。

② 剛体の生成と登録

```
//剛体の形状を表すオブジェクトを作成する。
//このサンプルでは半径 0.5 の球体コリジョンを作成。
btSphereShape shape = new btSphereShape(0.5f);
//座標と回転を使って剛体の初期位置を決める。
btTransform transform;
transform.setIdentity();
```

```
transform.setOrigin(btVector3(rbInfo.pos.x, rbInfo.pos.y, rbInfo.pos.z));
transform.setRotation(btQuaternion(rbInfo.rot.x, rbInfo.rot.y, rbInfo.rot.z, rbInfo.rot.w));
//モーションステートの生成。
myMotionState = new btDefaultMotionState;
//質量、モーションステート、コリジョン形状から剛体情報を用意して、剛体を作成。
btRigidBody::btRigidBodyConstructionInfo btRbInfo(rbInfo.mass, myMotionState, shape, btVector3(0, 0, 0));
//剛体を作成。
rigidBody = new btRigidBody(btRbInfo);
//作成した剛体を物理ワールドに追加。
dynamicWorld-> addRigidBody(rigidBody);
```

さて、初期化と剛体の登録までは終わりました。次は物理シミュレーションを行う方法を見ていきましょう。下記のコードを毎フレーム呼び出してください。

③ 物理シミュレーションの実行

```
//1 フレームの経過時間(単位は秒)を引数に渡して、物理シミュレーションを行う。
dynamicWorld->stepSimulation(1.0f / 60.0f );
```

では最後に物理シミュレーションを行った結果をゲームに引き渡すコードを見てみましょう。

④ ゲームオブジェクトへの引渡し

```
btTransform trans;
myMotionState->getWorldTransform(trans);
//ゲームのオブジェクトに位置と回転を引き渡す。
position = *(D3DXVECTOR3*)&trans.getOrigin();
rotation = *(D3DXQUATERNION*)&trans. getRotation();
```

簡単な剛体物理シミュレーションの使い方を説明しましたが、このチャプターの趣旨は物理エンジンを活用した衝突判定です。なので、実はこのチャプターでは必要なのは下記の３つの処理になります。

① 物理ワールドの初期化
② 剛体の生成と登録
③ 物理シミュレーションの実行

13.2 コリジョン

コリジョンは衝突判定で用いられる用語で、コリジョン＝衝突という意味になります。物体同士のコリジョンを調べるためにはその物体の形状を決める必要があります。形状とは例えば下記のようなものがあります。

・球体(SphereCollider)
・箱(BoxCollider)
・カプセル(CapsuleCollider)

・3D モデルと同じ形状をもったメッシュ(MeshCollider)

このようなコリジョンの形状のことをゲームエンジンの Unity では Collider(コライダー)と呼称しており、この教材でもその呼称を使用します。

多くのゲームでは Collider に複雑な形状をもった MeshCollider を使うことはほとんどなく、SphereCollider、BoxCollider、CapsuleCollider を組み合わせて複雑な形状の Collider を作成しています。その理由はモデルから生成した MeshCollider は非常に詳細な形状を持っており、衝突判定の処理に時間がかかってしまいます。また、複雑な形状の Collider との衝突判定は往々にして当たり抜けが発生しやすくなります。MeshCollider を使うとしても、ポリゴン数を減らした別データを用意することがほとんどです。

しかし、MeshCollider 以外は実装するのが非常に簡単ですので、今回はモデルから自動的に生成することが可能な MeshCollider の作り方を見ていきます。

13.3 MeshCollider

DirectXLesson/ CollisionDemo/game/Physics/ MeshCollider.cpp を開いてください。このソースファイルの 27 行目から始まっている CreateFromSkinModel 関数が MeshCollider を作成しているコートです。モデルを元に作成を行っているため、第一引数に SkinModel のインスタンスのアドレスを受け取っています。第二引数の offsetMatrix はモデルの頂点をワールド座標系に変換させるための行列です。

```
LPD3DXMESH mesh = model->GetOrgMeshFirst();
```

このコードでメッシュのアドレスを取得しています。

頂点バッファの作成

```
//頂点ストライドを取得。
DWORD stride = D3DXGetFVFVertexSize(mesh->GetFVF());
//頂点バッファを取得。
LPDIRECT3DVERTEXBUFFER9 vb;
mesh->GetVertexBuffer(&vb);
//頂点バッファの定義を取得する。
D3DVERTEXBUFFER_DESC desc;
vb->GetDesc(&desc);
//頂点バッファをロックする。
D3DXVECTOR3* pos;
vb->Lock(0, 0, (void**)&pos, D3DLOCK_READONLY);
VertexBuffer* vertexBuffer = new VertexBuffer;
int numVertex = mesh->GetNumVertices();
//当たりデータで使用する頂点バッファを作成。
for (int v = 0; v < numVertex; v++) {
		D3DXVECTOR3 posTmp = *pos;
		if (offsetMatrix) {
			D3DXVec3TransformCoord(&posTmp, pos, offsetMatrix);
```

```
                }
                vertexBuffer->push_back(posTmp);
                char* p = (char*)pos;
                p += stride;
                pos = (D3DXVECTOR3*)p;
}
vb->Unlock();
vb->Release();
vertexBufferArray.push_back(vertexBuffer);
```

34 行目から 60 行目まではメッシュコライダーを作成するための頂点バッファを作成するコードです。

インデックスバッファの作成

```
//続いてインデックスバッファを作成。
LPDIRECT3DINDEXBUFFER9 ib;
mesh->GetIndexBuffer(&ib);
D3DINDEXBUFFER_DESC desc;
ib->GetDesc(&desc);
int stride = 0;
if (desc.Format == D3DFMT_INDEX16) {
                //インデックスが 16bit
                stride = 2;
}
else if(desc.Format == D3DFMT_INDEX32){
                //インデックスが 32bit
                stride = 4;
}
//インデックスバッファをロック。
char* p;
HRESULT hr = ib->Lock(0, 0, (void**)&p, D3DLOCK_READONLY);
IndexBuffer* indexBuffer = new IndexBuffer;
for (int i = 0; i < (int)desc.Size / stride; i++) {
                unsigned int index;
                if (desc.Format == D3DFMT_INDEX16) {
                                unsigned short* pIndex = (unsigned short*)p;
                                index = (unsigned int)*pIndex;
                }
                else {
                                unsigned int* pIndex = (unsigned int*)p;
                                index = *pIndex;
                }

                indexBuffer->push_back(index);
                p += stride;
}
ib->Unlock();
ib->Release();
indexBufferArray.push_back(indexBuffer);
```

63 行目から 109 行目まではインデックスバッファを作成するためのコードです。

btIndexMesh の生成

```
        //インデックスメッシュを作成。
        btIndexedMesh indexedMesh;
        IndexBuffer* ib = indexBufferArray.back();
        VertexBuffer* vb = vertexBufferArray.back();
        indexedMesh.m_numTriangles = (int)ib->size() / 3;
        indexedMesh.m_triangleIndexBase = (unsigned char*)(&ib->front());
        indexedMesh.m_triangleIndexStride = 12;
        indexedMesh.m_numVertices = (int)vb->size();
        indexedMesh.m_vertexBase = (unsigned char*)(&vb->front());
        indexedMesh.m_vertexStride = sizeof(D3DXVECTOR3);
        stridingMeshInterface->addIndexedMesh(indexedMesh);

}
meshShape = new btBvhTriangleMeshShape(stridingMeshInterface, true);
```

そして最後の 99 行目から 112 行目までのコードで、作成した頂点バッファ、インデックスバッファを用いて BulletPhysics の btIndexMesh を生成して、btTriangleMeshShape を生成しています。これで MeshCollider の完成です。

13.4 RigidBody の登録

MeshCollider を作成することができたので、RigidBody を作成しましょう。ここでは bulletPhysics の btRigidBody を直接触らずに、ラッパークラス(*1)の RigidBody を使います。DirectXLesson/ CollisionDemo/game/MapChip.cpp の 50 行目から 58 行目にかけてサンプルとなるコードがあります。

```
//スキンモデルからメッシュコライダーを作成する。
D3DXMATRIX* rootBoneMatrix = modelData.GetRootBoneWorldMatrix();
meshCollider.CreateFromSkinModel(&model, rootBoneMatrix);
//続いて剛体を作成する。
//まずは剛体を作成するための情報を設定。
RigidBodyInfo rbInfo;
rbInfo.collider = &meshCollider;    //剛体のコリジョンを設定する。
rbInfo.mass = 0.0f;                 //質量を 0 にすると動かない剛体になる。
rbInfo.pos = position;
rbInfo.rot = rotation;
//剛体を作成。
rigidBody.Create(rbInfo);
```

(*1) ラッパークラスとは、既存のクラスのインスタンスを保持して、そのクラスを使いやすくなるようにインターフェースを定義したり、使用できる機能を制限したりすることが目的のクラスのことです。

```
//作成した剛体を物理ワールドに追加。
game->GetPhysicsWorld()->AddRigidBody(&rigidBody);
```

これで剛体を物理ワールドに登録することができました。剛体の質量を設定するときに 0.0 を設定していることに注意してください。質量を 0.0 にすることで、その剛体は静的になり、他の剛体と衝突しても動くことがなくなります。地面や、建物などのような動かない剛体は 0.0 を設定してください。

これで背景などの静的オブジェクトの MeshCollider を作成することができるようになり、その Collider を設定した剛体を物理ワールドに登録することもできるようになりました。これで単純な剛体同士の衝突判定であれば、全て物理エンジンが計算を行ってくれるようになりました。しかし、ユーザーが操作するゲームキャラクターの挙動を全て物理シミュレーション任せにすることはできません。なぜなら、ゲームキャラクターが移動する方向と速度の決定はユーザーが決定します、このときキャラクターの挙動は物理的に正しくない挙動をすることが多々あります（物理的に正しい挙動は操作性が悪く感じることがあります）。また、物体に衝突した時のキャラクターの挙動もゲームの状況によって異なることもあるでしょう。反発する壁とかありそうですよね。そのため、ゲームキャラクターのコリジョン処理はアプリケーション側で実装したほうが都合の良い場合が多々あります。
次の節からは、キャラクターのコリジョン処理を実装するために必要な知識のコリジョン検出とコリジョン解決を見ていきましょう。

13.5 コリジョン検出とコリジョン解決

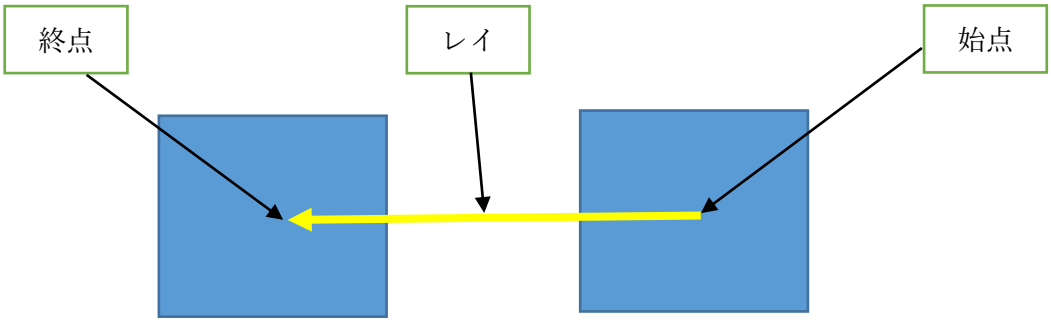
キャラクターコントローラーの話をする前に、コリジョン検出とコリジョン解決の話をします。コリジョン処理には大きく分類するとコリジョン検出とコリジョン解決に分けることができます。コリジョン検出は Collider 同士が衝突するかどうか、衝突する場合は衝突点を求めたり、衝突する面の法線を求めたりする処理のことをいいます。コリジョン解決は衝突する場合のめり込みの補正の処理を差します。

コリジョン検出は非常に難しいプログラムを実装することになるのですが、その難しい部分は BulletPhysics が実装を行ってくれているので、ありがたくそれを使いましょう。下記は BulletPhysics のレイキャストを使用した衝突判定のサンプルコードになります。

```
btTransform start;
start.setOrigin(btVctor3(10.0f, 10.0f, 10.0f)); //レイの始点を作成。
btTransform end;
end.setOrigin(btVector3(20.0f, 10.0f, 10.0f)); //レイの終点を作成。
ClosestConvexResultCallback cb              //衝突したときに呼ばれるコールバックの関数オブジェクト。
dynamicWorld-> ConvexSweepTest(
 shape,                                     //レイを飛ばすオブジェクトの形状
 start,                                     //レイの始点
 end,                                       //レイの終点
```

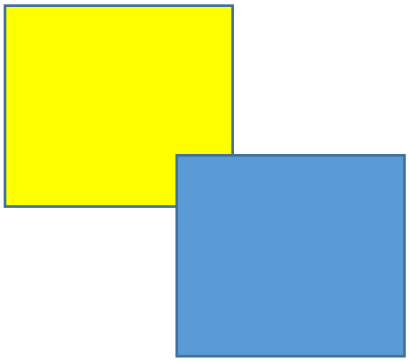
```
  cb                                          //コールバックオブジェクト
);
if(cb.hasHit()){
  //衝突した
}
```

ConvexSweepTest 関数にオブジェクトの形状を渡していることに注意してください。この形状は btCapsuleShape、btBoxShape、btSphereShape のインスタンスです。このレイキャストは下記の図のようなことをしていると考えて下さい。

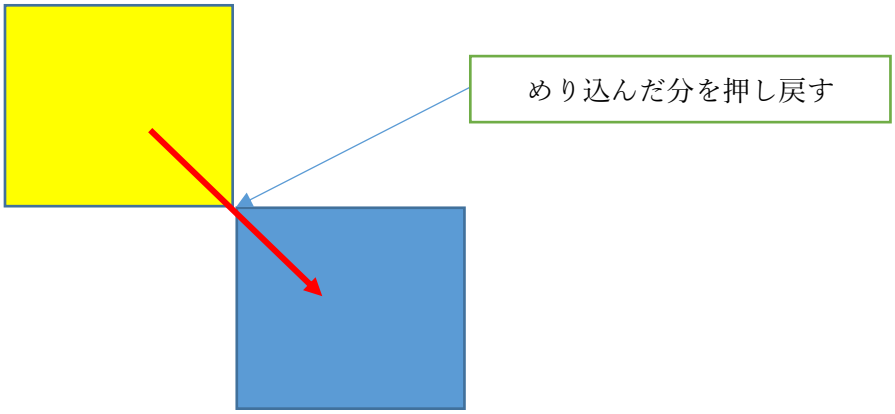


上の図の始点と終点の間に物理ワールドに登録されている剛体がある場合は、衝突しているという結果になります。

　続いてコリジョン解決を見ていきましょう。コリジョン解決とはコリジョン検出を行った結果、衝突していることが分かった場合に、その衝突を解決するものです。例えば下記のようなケースを考えてみて下さい。



物体がこのような位置関係になる場合にコリジョン検出を行うと、衝突しているという結果が返ってきます。多くのゲームではこのようなときは下記のような位置関係に補正するプログラムが実行されていると思います。



青いオブジェクトが赤い矢印の方向に押し戻されています。これがコリジョン解決です。

13.6 キャラクターコントローラー

最近の物理エンジンには前節で勉強した典型的なゲームキャラクターのコリジョン検出とコリジョン解決を行ってくれるキャラクターコントローラーというものがあります。しかし前述したように、キャラクターの挙動というのはゲームによって細かい調整が行われます。そのため、キャラクターコントローラーが行っている処理を理解するということは無駄ではありません。この節ではサンプルプログラムの CharacterController クラスを例にして、処理を説明していきます。

DirectXLesson/CollisionDemo/game/Physics/CharacterController.cpp の 128 行目から 285 行目までのプログラムを見てみて下さい。この関数がコリジョン検出、コリジョン解決を行っている関数です。

移動速度に従って、次の移動先を計算しているコード

```
//速度に重力加速度を加える。
m_moveSpeed.y += m_gravity * (1.0f / 60.0f);
//次の移動先となる座標を計算する。
D3DXVECTOR3 nextPosition = m_position;
//速度からこのフレームでの移動量を求める。オイラー積分。
D3DXVECTOR3 addPos = m_moveSpeed;
addPos *= 1.0f/60.0f;
nextPosition += addPos;
```

130 行目～137 行目までのコードは次の移動先を計算しているコードになります。
移動速度の単位は m/sec になっているために、移動速度に 1 フレームの経過時間(1/60 秒)を乗算しています。そして移動量が求まったら、それを nextPosition に加算しています。

XZ 平面でのコリジョン検出

```
//カプセルコライダーの中心座標 + 0.2 の座標を posTmp に求める。
D3DXVECTOR3 posTmp = m_position;
posTmp.y += m_height * 0.5f + m_radius + 0.2f;
//レイを作成。
```

```
btTransform start, end;
start.setIdentity();
end.setIdentity();
//始点はカプセルコライダーの中心座標 + 0.2 の座標を posTmp に求める。
start.setOrigin(btVector3(posTmp.x, posTmp.y, posTmp.z));
//終点は次の移動先。XZ 平面での衝突を調べるので、y は posTmp.y を設定する。
end.setOrigin(btVector3(nextPosition.x, posTmp.y, nextPosition.z));

SweepResultWall callback;
callback.me = m_rigidBody.GetBody();
callback.startPos = posTmp;
//衝突検出。
game->GetPhysicsWorld()->ConvexSweepTest((const btConvexShape*)m_collider.GetBody(), start, end, callback);
```

続いて、移動先が求まったので Y 成分を無視した XZ 平面での移動だけ考えます。移動前の座標を始点、移動後の座標を終点とするレイを作成してレイキャストを行ってコリジョン検出を行っています。

XZ 平面でのコリジョン解決

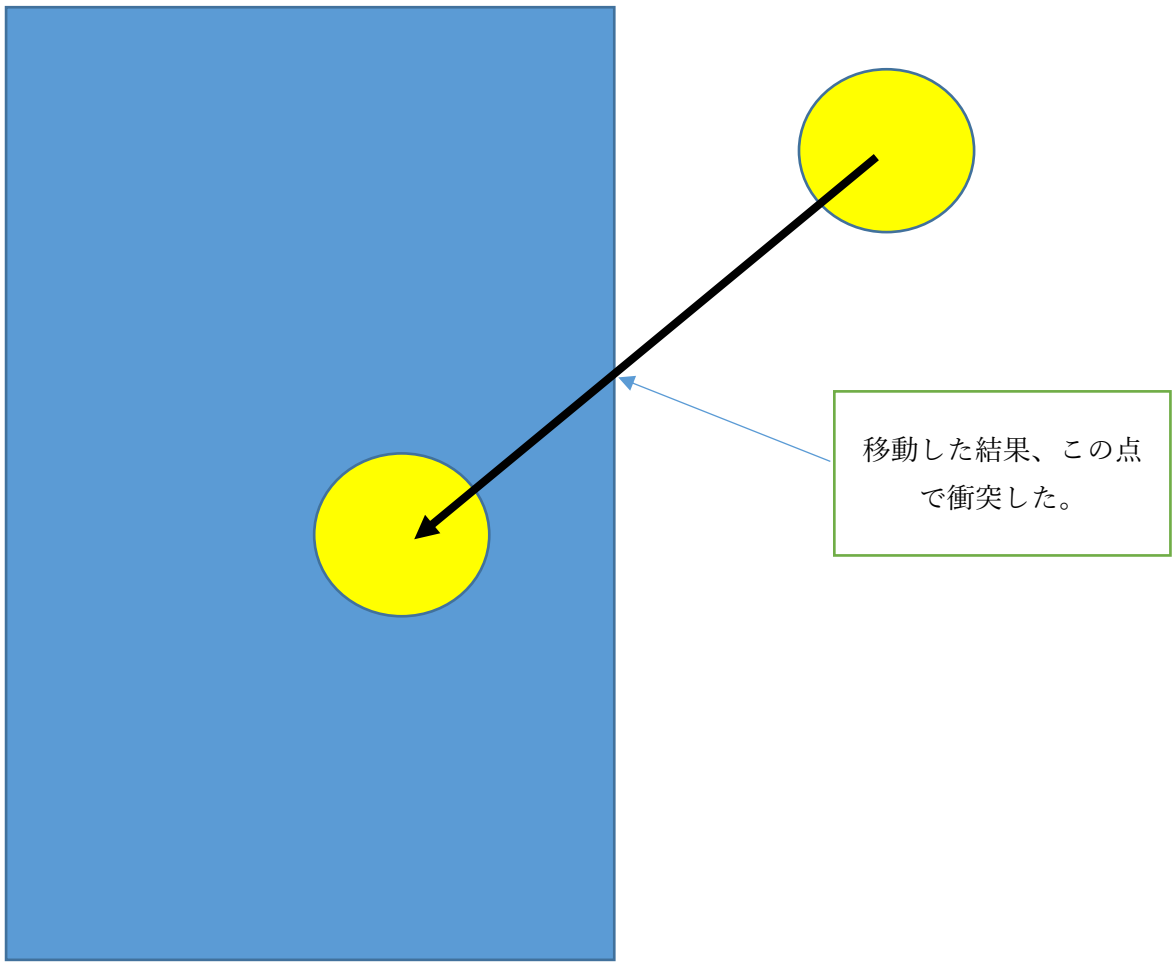
```
//当たった。
//壁。
D3DXVECTOR3 vT0, vT1;
//XZ 平面上での移動後の座標を vT0 に、交点の座標を vT1 に設定する。
vT0 = { nextPosition.x, 0.0f, nextPosition.z };
vT1 = {callback.hitPos.x, 0.0f, callback.hitPos.z};
//めり込みが発生している移動ベクトルを求める。
D3DXVECTOR3 vMerikomi;
vMerikomi = vT0 - vT1;
//XZ 平面での衝突した壁の法線を求める。。
D3DXVECTOR3 hitNormalXZ = callback.hitNormal;
hitNormalXZ.y = 0.0f;
D3DXVec3Normalize(&hitNormalXZ, &hitNormalXZ);
//めり込みベクトルを壁の法線に射影する。
float fT0 = D3DXVec3Dot(&hitNormalXZ, &vMerikomi);
//押し戻し返すベクトルを求める。
//押し返すベクトルは壁の法線に射影されためり込みベクトル+半径。
D3DXVECTOR3 vOffset;
vOffset = hitNormalXZ;
vOffset *= (-fT0 + m_radius);
nextPosition += vOffset;
D3DXVECTOR3 currentDir;
currentDir = nextPosition - m_position;
currentDir.y = 0.0f;
D3DXVec3Normalize(&currentDir, &currentDir);

if (D3DXVec3Dot( &currentDir, &originalXZDir ) < 0.0f) {
        //角に入った時のキャラクタの振動を防止するために、
        //移動先が逆向きになったら移動をキャンセルする。
        nextPosition.x = m_position.x;
        nextPosition.z = m_position.z;
        break;
}
```

この 176 行目～208 行目までがコリジョン解決を行っているコードです。この処理は**壁ずり**と呼ばれている壁に衝突した際の典型的なコリジョン解決となります。

13.6.1 壁ずり

では壁ずりについて見ていきましょう。下記の図のようにキャラクターが移動した場合を考えてみて下さい。



Chapter 14 パーティクル

14.1 ビルボード

ビルボードとは板ポリが常にカメラの方向を向く手法のことを言います。
常にカメラの方向を向くということは、カメラの回転行列を加えることになります。

カメラの回転行列は、カメラ行列の逆行列の平行移動成分を削除することで求めることができます。

```
D3DXMATRIX viewRotMatrix;
D3DXMatrixInverse(&viewRotMatrix, NULL, &viewMatrix); //カメラ行列の逆行列を求める。
//カメラの平行移動成分を 0 にする。
viewRotMatrix.m[3][0] = 0.0f;
viewRotMatrix.m[3][1] = 0.0f;
viewRotMatrix.m[3][2] = 0.0f;
viewRotMatrix.m[3][3] = 1.0fl
```

上のコードでカメラの回転行列を求めることができます。

あとは、この行列を描画したいオブジェクトのワールド行列に適用することで、ビルボードが完成します。

```
D3DXMatrix m;
m = viewRotMatrix * viewMatrix * projMatrix;
```

14.2 パーティクルに初速度を与えて動くようにする

Unity や UnrealEngine などのパーティクルエンジンにはエミッターに初速度を与えることができると思います。今、みなさんに触ってもらっているパーティクルのプログラムは板ポリが一枚だけ表示されているように見えるかもしれませんが、実は何枚も同じ場所に生成されています。そこで、このパーティクルエンジンンにも初速度のパラメータを追加して、パーティクルを動かせるようにしてみましょう。

まず、パーティクルのパラメータに初速度を追加します。

ParticleEmitter.h

```
struct SParicleEmitParameter{
   //初期化。
   void Init()
   {
       memset(this, 0, sizeof(SParicleEmitParameter));
   }
   const char* texturePath;                    //!<テクスチャのファイルパス。
   float w;                                    //!<パーティクルの幅。
   float h;                                    //!<パーティクルの高さ。
   float intervalTime;                         //!<パーティクルの発生間隔。
   D3DXVECTOR3 initSpeed;                      //初速度。
};
```

このパラメータを CParticle クラスのインスタンスに渡してやって、パーティクルを動かしてください。

恐らく、CParticle クラスに速度のメンバ変数や、座標のメンバ変数が必要になるはずです。そして。CParticle::Render 関数でパーティクルのワールド行列を計算する必要が生まれます。

今回はここまでのヒントで実習にチャレンジしてみて下さい。

14.2.1 初速度をパーティクルに引き渡す。

パーティクルをエミットされる際に、前節で追加した構造体を使用して初速度を渡しました。各粒子の描画、ワールド行列の更新を行っているのは CParticl クラスですので、この初速度を CParticle クラスに渡してやる必要があります。恐らく CParticle クラスに速度のメンバ変数を追加することになるでしょう。また、速度を使って位置も変位させていくはずですから、位置を表す座標のメンバ変数も必要になるはずです。

```
/*!
 * @briefパーティクル。
 */
class CParticle{
   CPrimitive                  primitive;      //!<プリミティブ。
   LPDIRECT3DTEXTURE9          texture;        //!<テクスチャ。
   ID3DXEffect*                shaderEffect;   //!<シェーダーエフェクト。
   D3DXVECTOR                  moveSpeed;      //!<速度。
   D3DXVECTOR3                 position;       //!<座標
public:
   CParticle();
   ~CParticle();
   void Init(const SParicleEmitParameter& param);
   void Update();
   void Render(const D3DXMATRIX& viewMatrix, const D3DXMATRIX& projMatrix) ;
};
```

速度を引き渡すことができたのであれば、その速度を使って座標を変位させていくだけです。今回は座標の変異は Update 関数で行いましょう。

```
void CParticle::Update()
{
    position += moveSpeed;
}
```

あとはこの座標を使用して平行移動行列を作成して、ワールド行列に適用すれば完了です。そこは自分で考えて実装してみてください。

14.3 初速度に乱数を加えてみよう。

今のままですと、決まった一定方向にパーティクルが飛んでいくだけになっています。もちろんそういうパーティクルもありますが、炎や煙のパーティクルはこれでは実装できません。炎や煙のような粒子のパーティクルは空気の流れなどを受けて均一の方向には流れていきません。そこでパーティクルに初速度に乱数を与えて、これを非常に簡単に近似してみましょう。

14.3.1 乱数アルゴリズム

パーティクルから話がずれるのですが、パーティクルだけではなく、ゲームのクオリティを上げるための乱数アルゴリズムについて少しだけ紹介します。

C 言語の標準関数の rand 関数は線形合同法と呼ばれるアルゴリズムを使用しています。このアルゴリズムはお世辞にも品質の高い乱数アルゴリズムとは言えません。そのため、ゲーム会社では自前で別の乱数アルゴリズムを使った乱数生成機を実装しています。Unity や UnrealEngine を使う場合は、エンジンが乱数生成機を実装しています。

今回、私の作成したパーティクルのデモではメルセンヌ・ツイスターと言われる乱数アルゴリズムを使った乱数生成機を実装しています。

14.3.2 初速度に乱数を加えてみよう。

では本題の初速度に乱数を加える処理を考えていきましょう。初速度に乱数を加えるので、初速度を引き渡す Init 関数で初速度を加工してやればいいことになります。では、私の作成したパーティクルデモのプログラムを参考までに記載します。

```
//初速度に乱数を加える。
//random.GetRandDouble は 0.0～1.0 を返してくる関数。これに-0.5 してから*2.0 しているので、
//-1.0～1.0 の乱数を取得している事になる。この乱数に対して、パラメータで渡された諸速度の速度のランダム幅を乗算してやることで
//速度をランダムにしている。
velocity = param.initVelocity;
velocity.x += (((float)random.GetRandDouble() - 0.5f) * 2.0f) * param.initVelocityVelocityRandomMargin.x;
velocity.y += (((float)random.GetRandDouble() - 0.5f) * 2.0f) * param.initVelocityVelocityRandomMargin.y;
velocity.z += (((float)random.GetRandDouble() - 0.5f) * 2.0f) * param.initVelocityVelocityRandomMargin.z;
```

14.3.3 速度に重力加速度を加えてみよう。

加速度というのは速度を変位させていくものになります。例えば、重力加速度が9.8m/sec^2 だというのは、中学生あたりで勉強したと思います。9.8m/sec^2 というのは物体の落下速度が 1 秒ごとに 9.8m 加速するという意味になります。

3D ゲームでは速度は向きと大きさを表現できるベクトルで考えることがほとんどです。例えば、マリオのようなジャンプアクションゲームで走りながらジャンプした場合、下記のような速度になります。

```
moveSpeed = D3DXVECTOR3(1.0f, 5.0f, 0.0f); // X 方向に 1m/frame、Y 方向に 5.0m/frame の速度。
```

では、この速度を使用してプレイヤーの座標を動かしてみましょう。

```
player.position += moveSpeed;   //座標を変位させる。
```

このようなコードを記述することで、プレイヤーは 3D 空間上を動くことができます。

では、加速度について考えていきましょう。加速度というのは速度を変位させていくものになります。つまり、重力落下を実装したい場合は速度に対して加速度を適用する計算をすればいいことになります。

では先ほどの moveSpeed に重力加速度を加えるコードを追加してみましょう。

```
D3DXVECTRO3 gravity = D3DXVECTOR3(0.0f, -0.16f, 0.0f); //Y 方向に-0.16m/frame^2 の重力加速度。
moveSpeed += gravity;   //重力加速度を moveSpeed に適用する。
```

では、ここまでの説明を参考にしてパーティクルに重力落下を追加してみて下さい

14.4 加算合成

では、パーティクルの実習の最後にパーティクルに加算合成を行える機能を追加してみましょう。

加算合成はチャプター 8 で実装を行いましたので、その復習になります。

14.4.1 レンダリングステート

加算合成とはアルファブレンディングの一種で、これからフレームバッファに描きもうとしているカラーを、既に書き込まれているカラーと加算して描きこむことでした。
カラーの描き込み方は IDirect3DDevice9::SetRenderState 関数を使用することで設定できます。

サンプルコード

```
//アルファブレンディングを有効にする。
g_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
g_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);
g_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);
```

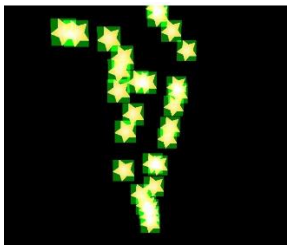
IDirect3DDevice9::`DrawIndexedPrimitive` は現在のレンダリングステートの状態を使用してプリミティブを描画します。上記のレンダリングステートのコードは `DrawIndexedPrimitive` の直前に記述すれば、加算合成を確認できるはずです。

14.4.2 レンダリングステートの切り替えによるオーバーヘッド

当然ですがレンダリングステートの切り替えというのはタダではありません。関数呼び出しのオーバーヘッドは当然かかりますし、PS4 などのハードでは不要なレンダリングステートの切り替えは GPU にも悪影響を与えます。そのため、レンダリングステートの切り替えというのはラッパークラスを作成して、IDirect3DDevice9 をカプセル化して制御する方がベターな場合があります。今回はわかりやすさのために `DrawIndexedPrimitive` の直前で毎回コールしていますが、まだまだ改善の必要のあるコードになっています。

14.4.3 シェーダーの変更

さて、前節でレンダリングステートを加算合成を行うように変更しました。しかし、それだけでは下記のような見え方になってしまい、まだ問題があります。



このようにアルファで抜けて欲しいはずの部分が正しく抜けていません。
では particleDemo/ShaderTutorial_04/ColorTexPrim.fx を開いてみてください。このシェーダーファイル
加算用のピクセルシェーダーの PSMainAdd 関数の 51 行目を見てみましょう。

```
return float4(tex.xyz * g_alpha, 1.0f/g_brightness );
```

この行がフレームバッファに描きこむソースカラーを返している処理になるのですが、実は星の画像はαで色を抜いている箇所にもカラーが埋め込まれています。そのためそのカラーの情報が加算合成されるために、上のような見え方になっていました。この行を下記のように変更してください。

```
return float4(tex.xyz*tex.a, 1.0f/g_brightness );
```

これできれいにαの部分が抜けるようになったはずです。

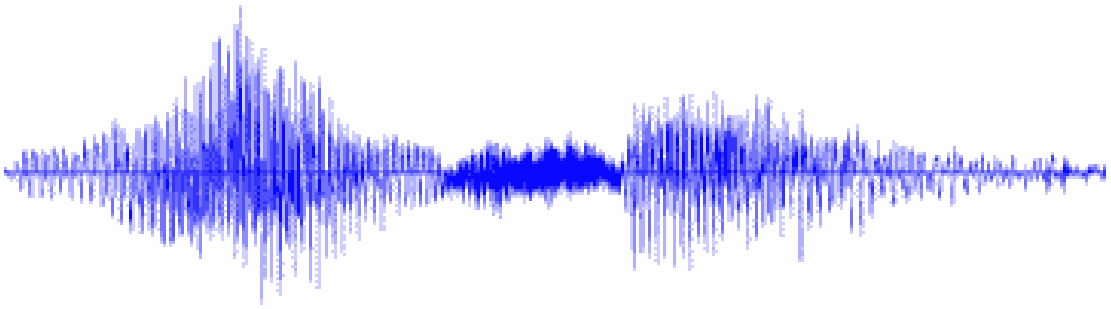
Chapter 15 サウンド

15.1 XAudio2

このチャプターでは Microsoft が提供している、Windows および Xbox 向けの低レベルのオーディオ API である XAudio2 を使用したサウンドプログラミングについて解説をします。

15.2 PCM データ(波形データ)

音とは空気を振動して伝わってきます。そして、その新動が鼓膜に伝わり鼓膜を揺らすことで私たちは音を認識することができます。音色はこの音の振動速さに依存します。振動は一秒間の振動の回数で測定され、単位をヘルツ(Hz)といいます。下のような図を見たことがあるのではないでしょうか。



コンピューターでは音を再生するためには、上記の音の振動のデジタル化を行う必要があり、デジタル化したデータを PCM データ、波形データなどと呼ばれます。では簡単な波形データを見てみましょう。

```
char waveData[] = { 128, 96, 64, 32, 0, -32, -61. -89, -126, -113, -89, -48, -12, 24, 65};
```

これが波形データです。0 が振動なし、そして 128〜-128 の間で値が揺れ動いていると思います。これが振動になります。コンピューターは PCM データを流し込まれて、そのデータ通りに振動を起こすことによって、音を再生しているのです。

15.3 色々なサウンドファイル

サウンドファイルには色々なフォーマットがあるが、最も多く使われているのは wave、mp3、ogg などだと思います。このうちの wave フォーマットは PCM データをそのまま保存しているフォーマットになります。そのため音質の劣化がないため、品質は最も高くなります。続いて mp3 や ogg といったフォーマットは PCM データを非可逆圧縮したフォーマットになります。そのため音質は wave フォーマットに比べると若干劣化していることにな

ります(ただし、まず気づきません)。mp3 や ogg は wave と比べると 1/10 以下のファイルサイズになるため、外部記憶メディアの容量が足りない場合などはこれらを活用すべきでしょう。ただし、音を再生するためには圧縮されている PCM データを復元する必要があるため、CPU 不可は増大します。

15.4 オンメモリ再生とストリーミング再生

音を再生するためには PCM データをメモリにロードして、そのデータをサウンドデバイスに流し込む必要があります。オンメモリ再生とは、再生したいサウンドファイルのデータを全てメモリ上にロードして再生する方法になります。一方ストリーミング再生とは全てのデータをメモリに乗せるのではなく、少しずつデータをロードして再生を行い、再生が完了したデータは逐次破棄していく手法となります。そのため、メモリの節約ができメインメモリの少ないハードウェアでも BGM などのようなサイズの大きなサウンドデータを再生することが可能です。一般的にサイズの小さな SE などのデータはオンメモリ再生、サイズの大きな BGM などのデータはストリーミング再生を行うことが多いです。では、次の節から実際に XAudio2 を活用したサウンド再生について見ていきましょう。

15.5 XAudio の初期化

tkEngine/Sound/tkSoundEngine.cpp、tkEngine/Sound/SoundEngine.h で XAudio の初期化を行っています。CSound::Init 関数ではマスターボリューム、リバーブエフェクト、サブミックスボイス、3D サウンドの初期化などを行っています。

15.6 サウンドソース

tkEngine/Sound/tkSoundSource.cpp、tkSoundSource.h を見てみてください。このクラスはサウンドソース(音源)の処理を記述しています。音源であるため、位置を記録する CVector3 型のメンバー変数も保持しています。初期化関数にはオンメモリ再生用の初期化関数の CSoundSource::Init と、ストリーミング再生用の初期化関数の CSoundSource::InitStreaming 関数が用意されています。

15.7 サウンドリスナー

サウンドソース(音源)があるのであれば、その音を聞くサウンドリスナーが必要です。サウンドリスナーの処理は tkEngine/Sound/tkSoundEngine.cpp に記述されています。
tkSoundEngine::Update 関数では登録されているサウンドソースが 3 D サウンドであれば、サウンドリスナーとの位置関係によって、3D サウンドエフェクトを計算するプログラムが記述されています。

15.8 wave ファイル

tkEngine/Sound/ tkWaveFile.cpp には wave ファイルを読み込むためのコードが書かれています。このクラスは XAudio の機能は使っていません。Wave ファイルから PCM データを読み込むコードが記述されています。Wave ファイルの読み込みは同期読み込みの CWave::Read 関数と、非同期読み込みの CWave::ReadAsync 関数が用意されています。CWave::ReadAsync 関数は内部で読み込みスレッドを起動して、メインスレッドと並列して PCM データの読み込みを行っています。メインスレッドをブロックしないため、ストリーミング再生の時に使用されています。

16 法線マップ

16.1 概要

このチャプターでは法線情報をテクスチャに書き込んで、低ポリゴンで細かなディテールを表現するための技術の法線マップについて見ていきます。

16.2 法線マップの登場

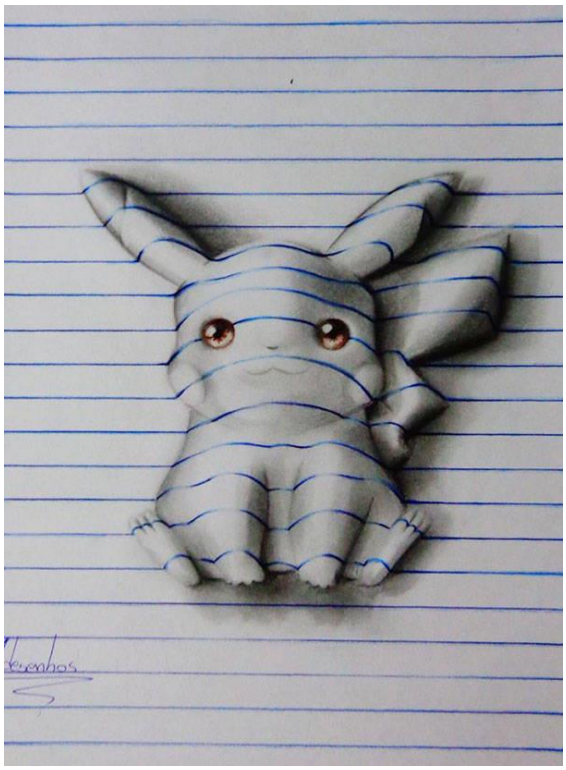
コンソールゲーム機で法線マップが本格的に使われだしたのは、PS3、Xbox360 が登場してからになります。法線マップはプログラマブルシェーダーが登場してから生まれた技術のため、固定パイプラインには存在しません。そのためプログラマブルシェーダーを本格的に使用することが可能になった PS3、Xbox360 の世代でコンソールゲーム機でも法線マップが活用されるようになりました。そして、現在は PS4、XboxOne、PC、スマートフォンなどでリリースされる多くの 3D ゲームではあって当たり前の技術になっています。

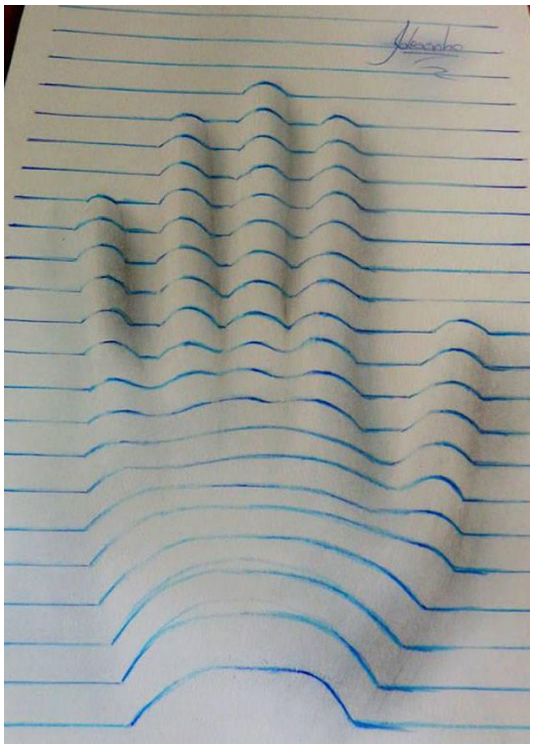
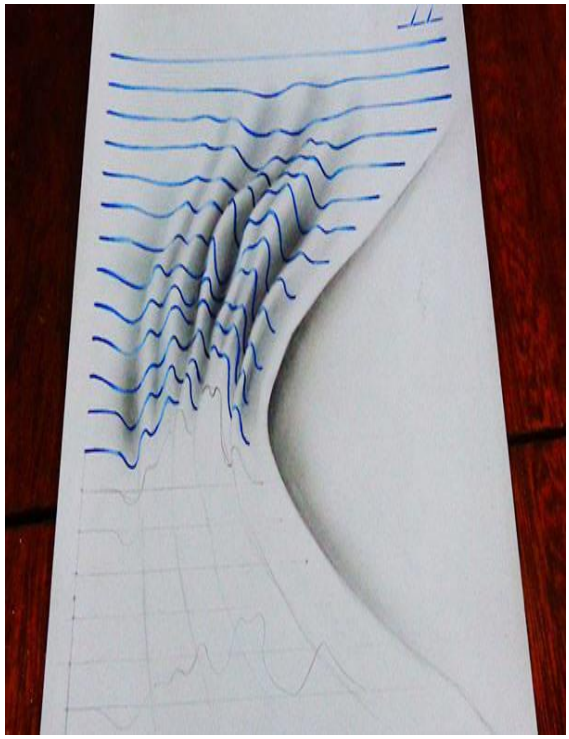
16.3 細かいディテールの表現

法線マップの目的は低ポリゴンで細かなディテールを表現することです。では、それを具体的に考えていきましょう。例えば道路のアスファルトを考えてみてください。アスファルトを近くでよく観察をしてみると細かな凹凸があることが分かります。もしこの凹凸をポリゴンで表現しようとすると非常に多くのポリゴンが必要になります。多すぎるポリゴンは GPU のパフォーマンスを低下させ、多くのメモリを使用してしまうことになります。このような細かいディテールの表現をポリゴンで行うことには多くの問題がありました。そこでこれらを高いパフォーマンスと少ないメモリ使用量で実現するために法線マップという技術が生まれました。

16.4 法線をテクスチャに書き込む

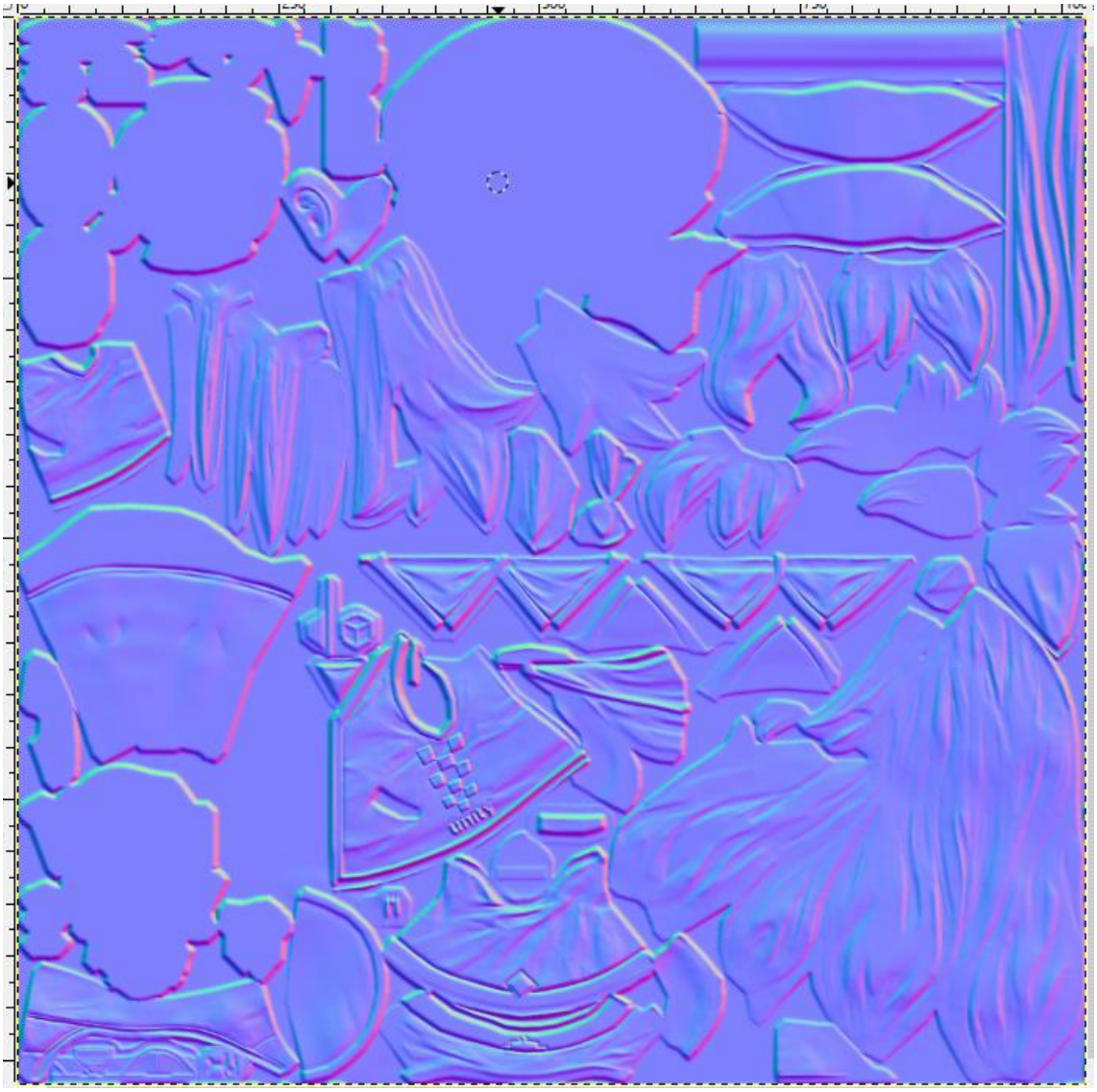
引き続き先ほどのアスファルトの凹凸で考えていきましょう。私たちはどうやってアスファルトの凸凹を認識しているのでしょうか？ 実は凸凹を認識するために一番重要な情報は光の陰影になります。下記のだまし絵を見てみてください。

立体的に見えると思います。しかしこれらは全て平らなノートに書かれた絵であって実際には凹凸はありません。このように人間は実際に凹凸がなくても、陰影をつけることで実際に凹凸があるように錯覚します。では、ここでディフューズライトやスペキュラライトの計

算式を思い出してください。ライトによって生成される陰影はライトの方向とモデルの頂点の法線によって決まっていました。つまり、ポリゴン数が少なくてもモデルの法線さえ詳細であれば、細かなディテールは表現できることになります。いわゆるだまし絵です。そこで、モデルの法線をテクスチャに書き込んでしまって、ライティングの時にはその法線を使用しようという考えが生まれます。これが法線マップです。下記はユニティちゃんの法線マップです。



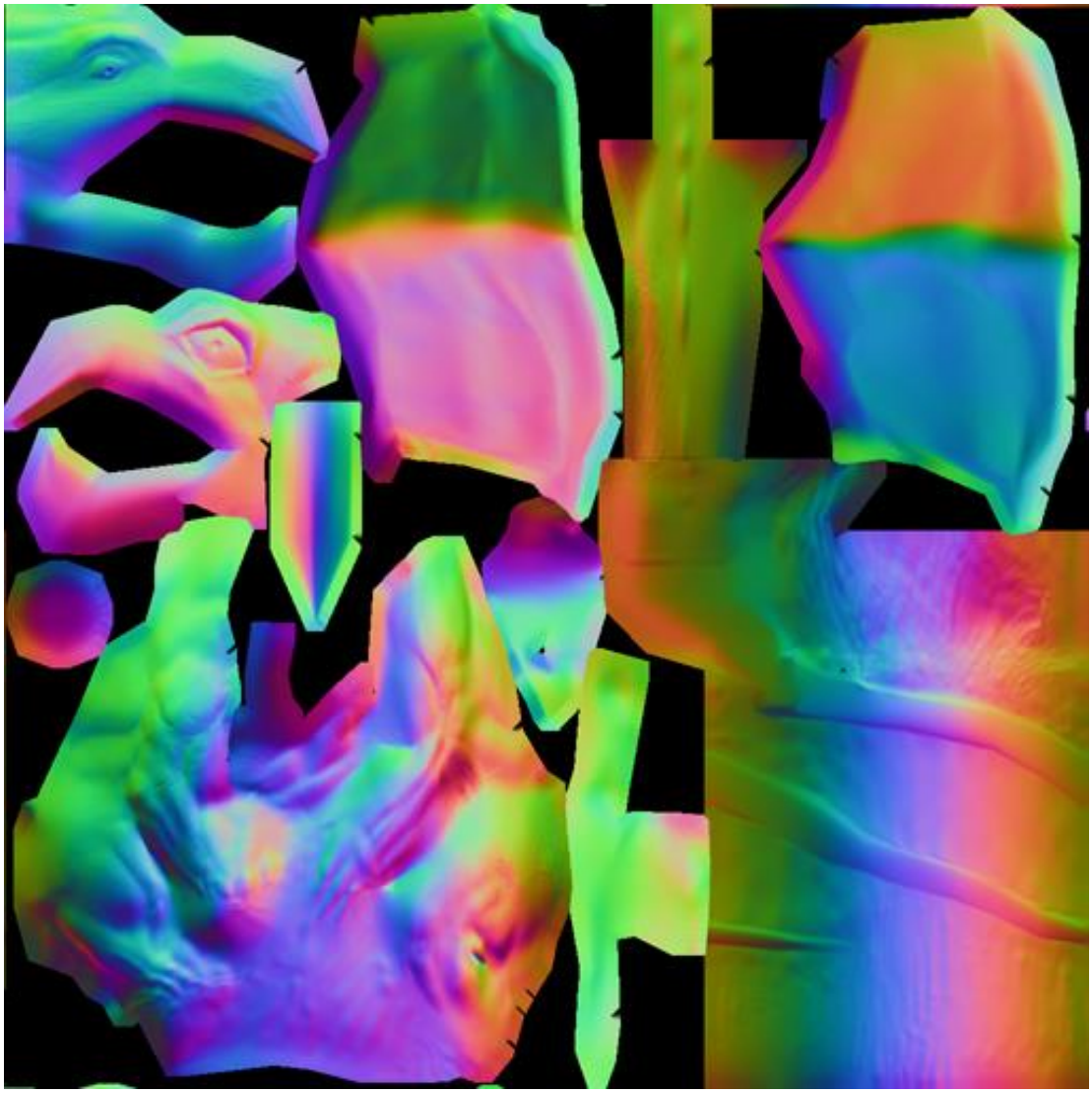
テクスチャの RGB 成分に、法線の XYZ 成分を書き込んでテクスチャにしたものです。

16.5 オブジェクトスペース法線マップとタンジェントスペース法線マップ

法線マップには**オブジェクトスペース法線マップ**と**タンジェントスペース法線マップ**という二種類のデータ形式があります。ではこの二つについて見ていきましょう。

16.5.1 オブジェクトスペース法線マップ

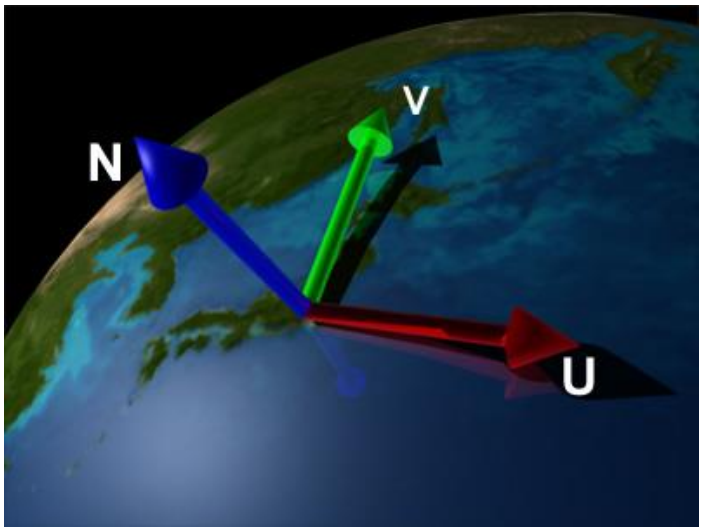
オブジェクトスペースはタンジェントスペースと比べるとイメージがしやすい法線マップです。この法線マップは単純にオブジェクトの法線の XYZ をテクスチャの RGB に書き込んでいるだけのものです。オブジェクトスペース法線マップは下記のようなデータになります。



先ほどのユニティちゃんの法線マップとは少し違った感じのデータに見えるかと思います。実は先ほどのユニティちゃんの法線マップはタンジェントスペース法線マップです。では続いてタンジェントスペース法線マップについて見ていきましょう。

16.5.2 タンジェントスペース法線マップ

タンジェントスペース法線マップは単純にモデルの法線を埋め込んだものではなく、**法線マップを貼り付けるポリゴンの法線座標系から見た法線**を書き込んでいることになります。ポリゴンの法線座標系というのはポリゴンの法線を Z 軸、ポリゴンの法線と直交している接ベクトル(タンジェント)を X 軸、法線と接ベクトルの外積で求めた従法線ベクトルを Y 軸とした空間のことです。



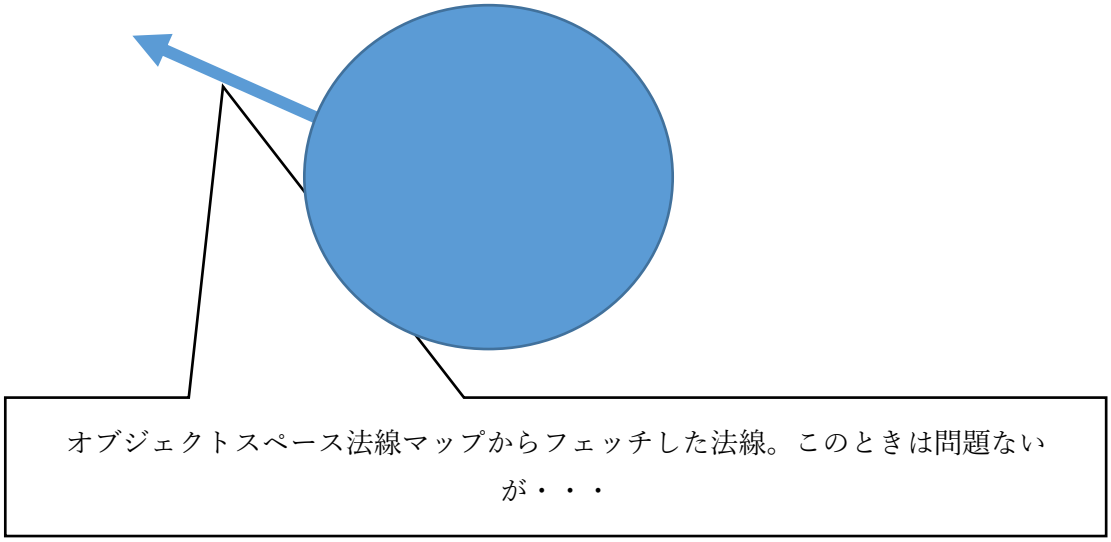
この画像ですと、N が法線、U が接ベクトル、V が従法線ベクトルになります。この 3 軸を基底軸とした空間がタンジェントスペースと呼ばれるものです。

例えば、N が(0.707, 0.0, 0.707)というベクトルだった場合、この N と同じ向きのベクトル(0.707, 0.0, 0.707)をタンジェントスペースに変換すると(0, 0, 1)になります。このように法線マップを貼り付けるポリゴンの法線と同じ向きのベクトルはタンジェントスペースに変換すると(0, 0, 1)に変換されます。これをテクスチャの RGB に変換して書き込むため、(0, 0, 255)という青いカラーが書き込まれることになります。多くのオブジェクトで法線マップに書き込む法線の方向は、貼り付けるポリゴンの法線から大きく変化することはありません。そのため、タンジェントスペース法線マップは青の成分が強く出る画像データになります。

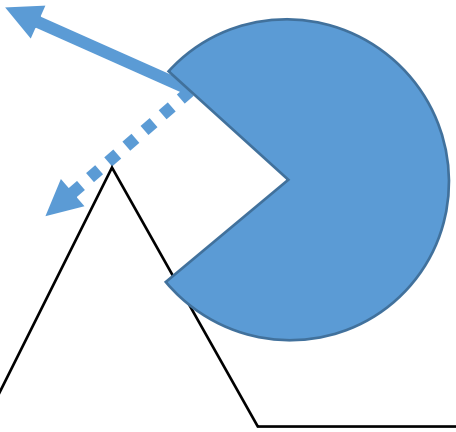
16.5.3 主流はタンジェントスペース法線マップ

現在主流となっているのはタンジェントスペース法線マップです。特に断りなく法線マップと言った場合はタンジェントスペースを指していると考えてください。
では、なぜタンジェントスペースが主流なのでしょうか。オブジェクトスペースの方が単純で分かりやすく感じたはずです。実はオブジェクトスペースには頂点の変形に対応できないという欠点があり、その欠点はタンジェントスペースであれば解決できるためです。

今の 3D ゲームではモーフィング、クロスシミュレーションなどといったオブジェクトの頂点を変形させる技術が使われています。その場合にオブジェクトスペース法線マップを使っていると下記のような問題が発生します。



オブジェクトスペース法線マップからフェッチした法線。このときは問題ないが・・・



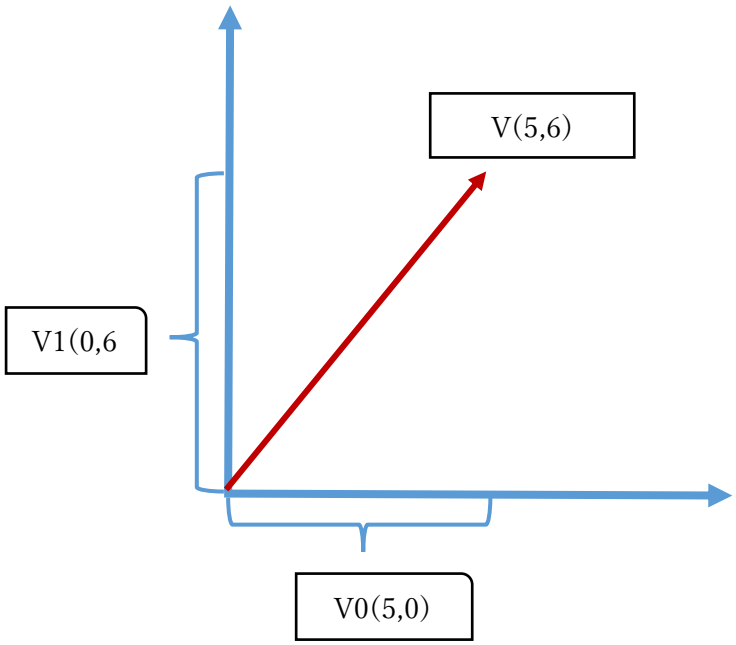
オブジェクトが変形しているので、本来は点線の法線が正しいのだが、オブジェクトスペースの場合は実線の法線になっていまい、見た目がおかしくなる。

これがオブジェクトスペースの欠点です。一方タンジェントスペースはどうでしょうか？タンジェントスペースの法線はライティングの計算を行うときに、ワールドスペースに変換を行う必要があります。このワールドスペースへの変換を行う時に貼り付けるモデルの法線を使って変換を行うことになります。つまり、オブジェクトが変形した時にポリゴンの法線さえを正しく計算を行っておけば変形後の法線を求めることができます。

オブジェクトスペース法線マップは非常にシンプルな考え方になるため、タンジェントスペース法線マップに比べると GPU パフォーマンスの面では軍配があがります。しかし、頂点の変形に対応できないというデメリットを抱えることとなります。

16.5.4 タンジェントスペースからワールドスペースへの変換

ライティングを行うためにはワールドスペースへ変換する必要があります。ではその変換の仕方について考えていきましょう。話を簡単にするために 2 次元で考えてみましょう。例えば、基底軸 ex(1,0)、ey(0,1)の座標系で考えてみましょう。



V は二次元上のベクトルです。このベクトルは X 軸方向に 5、Y 軸方向に 6 の大きさを持っています。ベクトル V は下記のような計算が成り立ちます。

V = V0 + V1

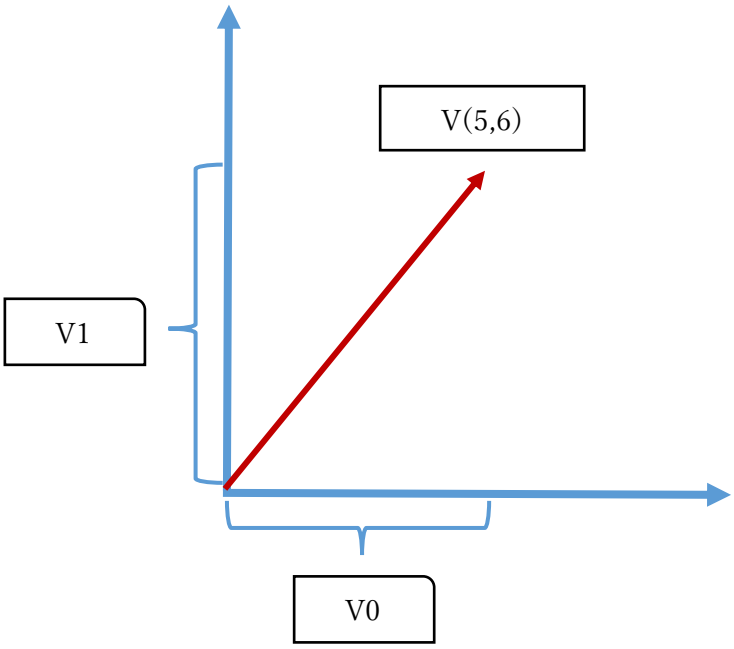
また、V0 と V1 は下記の計算が成り立ちます。

V0 = V.x * ex

V1 = V.y * ey

ex と ey はこの座標系の基底軸です。

この基底軸が例えば ex(0.707, -0.707), ey(0.707, 0.707)の場合を考えましょう。



グラフは先ほどと同じですが、基底軸が違うことに注意してください。V はワールドスペースでのベクトルではなく、基底軸 ex(0.707, -0.707), ey(0.707, 0.707)の座標系でのベクトルです。これを基底軸 ex(1, 0), ey(0,1)のワールド座標系に変換することを考えます。まず、V は V0 と V1 の合算で求まります。そこでワールド座標系で V0 と V1 がどのようなベクトルになるか求めてみましょう。V0 と V1 は下記の計算でも止まります。

V0 = V.x * ex

V1 = V.y * ey

つまり、V0 = 5×(0.707, -0.707)

= (3.535, −3.535)・・・・・・・・・①

V1 = 6×(0.707, 0.707)

=(3.535, 3.535) ・・・・・・・・・②

そして V は下記の計算で求まります。

V = V0 + V1

よって、①と②から

V = (3.535, −3.535) + (3.535, 3.535)

= (7.07, 0.0)

よって、(7.07, 0.0)が V をワールド空間に変換したベクトルということになります。

では、タンジェントスペースの法線をワールドスペースに変換する話に戻しましょう。タンジェントスペースの基底軸は ex が接ベクトル、ey が従法線、ez が法線です。各ベクト

を下記で表すこととします。

接ベクトル

tangent

従法線

biNormal

法線

normal

法線マップからフェッチしたタンジェントスペースでの法線

localNormal

この時にワールドスペースでの法線 worldNormal は下記の計算で求まります。

worldNormal = localNormal.x × tangent + localNormal.y × binormal + localNormal ×normal

16.6 サンプルコード

では、この節ではサンプルコードを使って法線マップの実装の仕方を見ていきましょう。サンプルコードは DireceXLesson/ShaderTutorial_09 です。

まず、main.cpp の LoadTexture 関数で法線マップのロードを行っています。

```
void LoadNormalMap()
{
    if( FAILED( D3DXCreateTextureFromFileA( g_pd3dDevice,
                                            "normal.jpg",
                                            &g_pNormalMap ) ) )
    {
        std::abort();
    }
    if (FAILED(D3DXCreateTextureFromFileA(g_pd3dDevice,
                                          "diffuse.jpg",
                                          &g_pDiffuseMap)))
    {
         std::abort();
    }
}
```

main.cpp の Render 関数でロードした法線マップを GPU へ転送しています。

```
g_pEffect->SetTexture("g_normalMap", g_pNormalMap);
```

続いてシェーダーを見ていきましょう basic.fx を開いてください。basic.fx に法線マップ用のテクスチャとテクスチャサンプラが追加されています。

```
texture g_normalMap;                    //法線マップ
sampler g_normalMapSampler =
sampler_state
{
    Texture = <g_normalMap>;
    MipFilter = LINEAR;
    MinFilter = LINEAR;
    MagFilter = LINEAR;
    AddressU = Wrap;
    AddressV = Wrap;
};
```

ピクセルシェーダーで法線マップからタンジェントスペース法線のフェッチとワールドスペース法線への変換を行っています。

```
//法線マップからタンジェントスペース法線をフェッチ。
float3 localNormal = tex2D( g_normalMapSampler, In.uv );
//法線と接ベクトルの外積を求めて従法線を求める。
float3 biNormal = normalize( cross( In.tangentNormal, In.normal) );
//-1.0～1.0 の範囲にマッピングする。
localNormal = (localNormal * 2.0f)- 1.0f;
//ワールドスペースの法線を求める。
float3 worldNormal = In.tangentNormal * localNormal.x
                   + biNormal * localNormal.y
                   + In.normal * localNormal.z;
```

localNormal に 2.0 をかけてあとで-1.0 を行っている箇所に注意してください。テクスチャの各要素 8bit の RGB にベクトルの情報を書き込んでいるため、各成分に 0-255 の値しか書き込めません。そのため、負数を書き込むために 0～127 を負数、128～255 を正数というようにマッピングしています。2.0 をかけた後で 1.0 減算を行うことで localNormal の値を-1.0～1.0 に変換しているのです。