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;
```

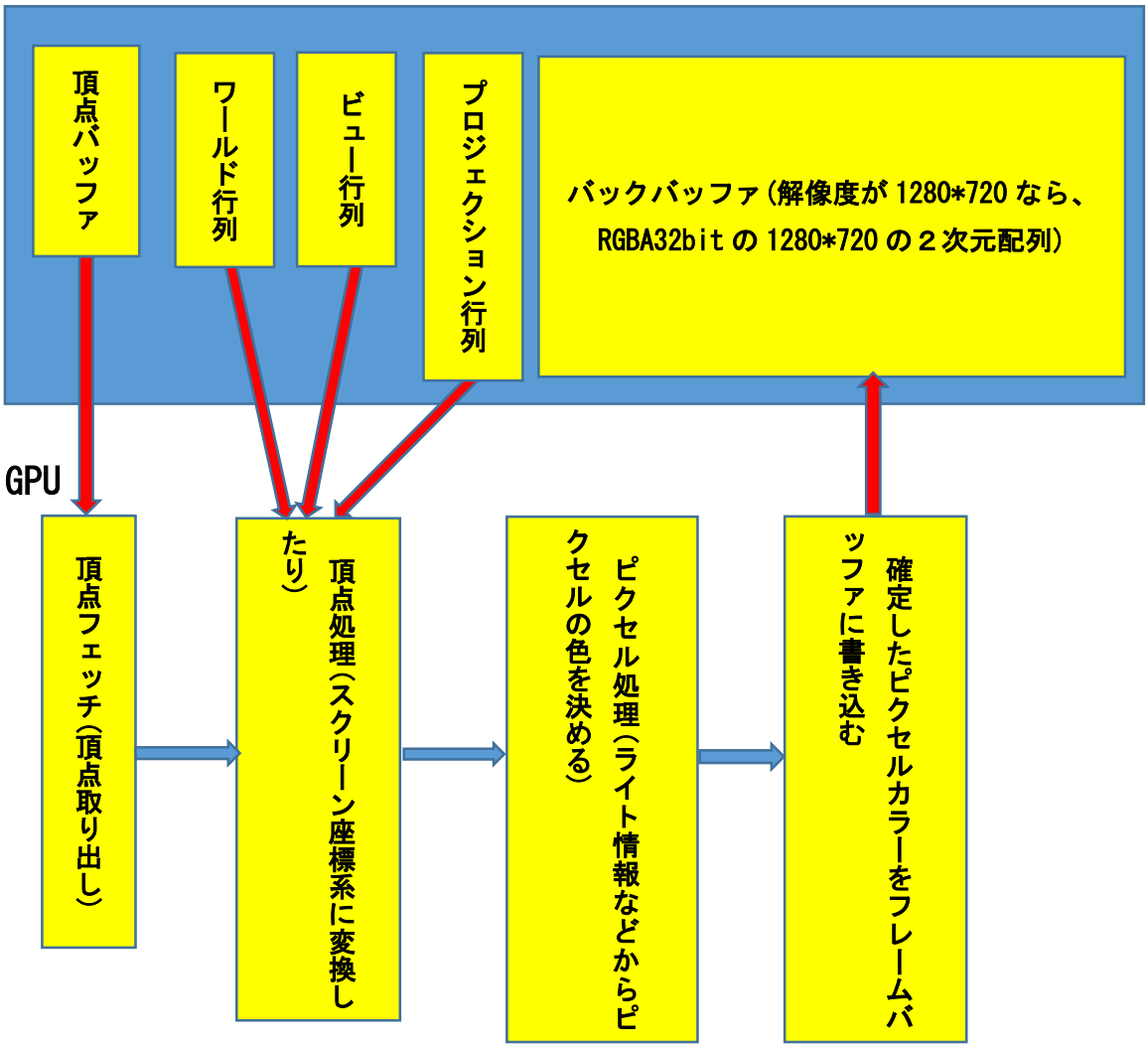
```
    ZeroMemory( &light, sizeof( D3DLIGHT9 ) );
    light.Type = D3DLIGHT_DIRECTIONAL;
    light.Diffuse.r = 1.0f;
    light.Diffuse.g = 1.0f;
    light.Diffuse.b = 1.0f;
    vecDir = D3DXVECTOR3( cosf( timeGetTime() / 350.0f ),
                          1.0f,
                          sinf( timeGetTime() / 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。

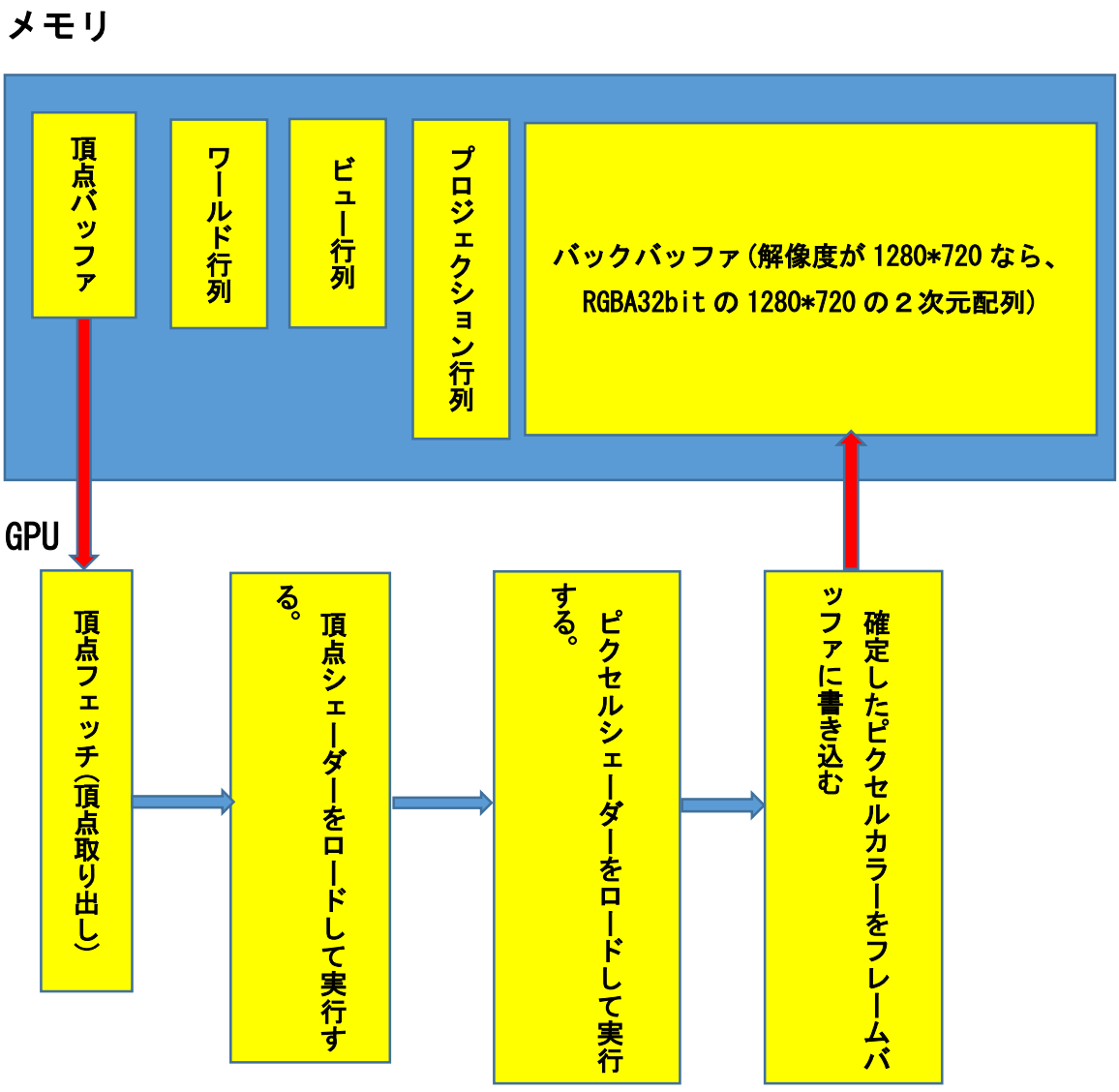
メモリ



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();
	}
}
```

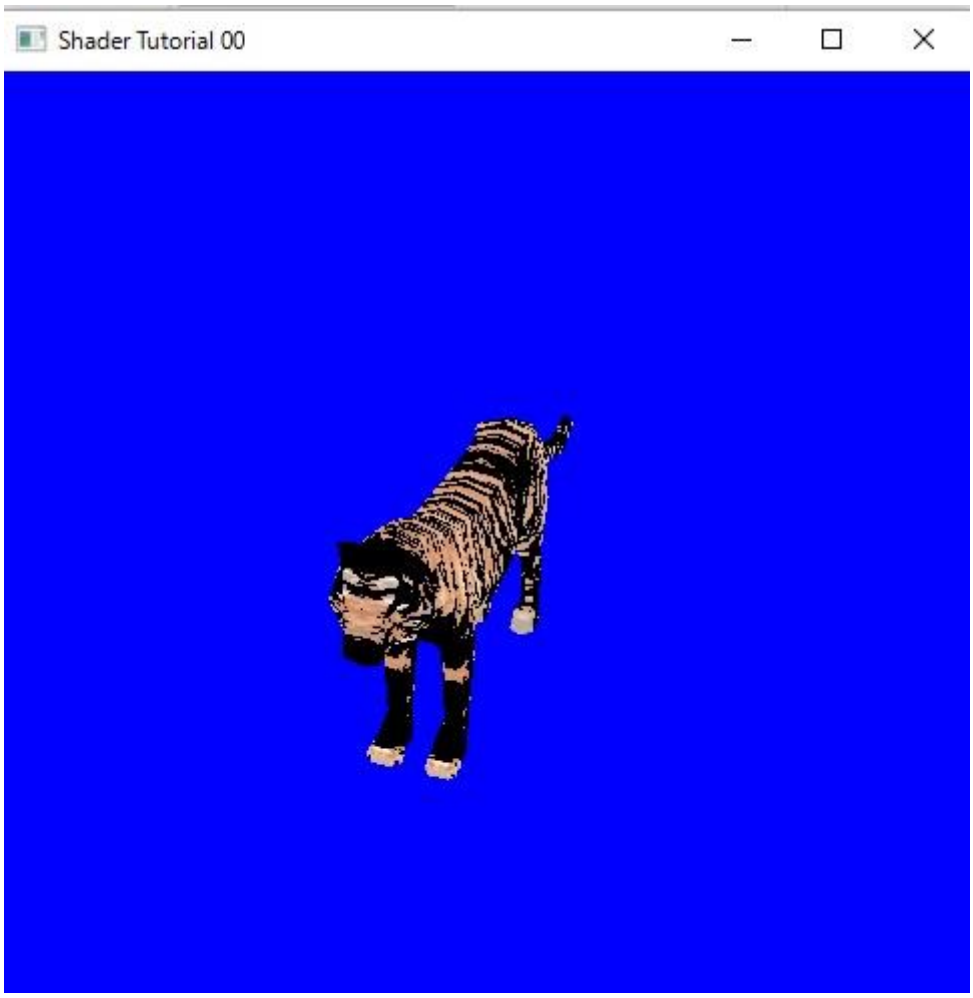
まず①の箇所が CPU から転送されたテクスチャのアドレスが格納された変数になります。そして②の箇所がテクスチャのサンプリング方法を記述したテクスチャサンプラと呼ばれるものです。テクスチャサンプラについてはここでは説明しません。今はこのように記述するものだと思っていてください。続いて③の箇所ですが、こちらが uv 座標を使用してテクスチャをサンプリングしているコードになります。

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

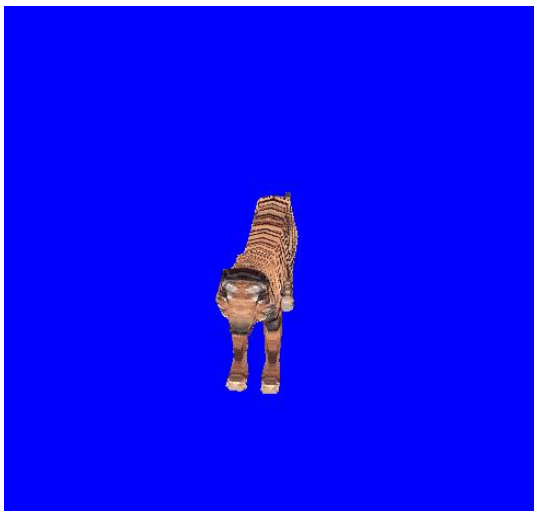
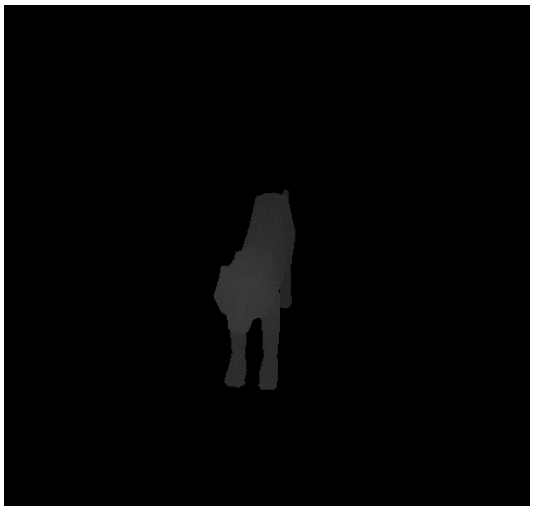


図 B 深度バッファ



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()

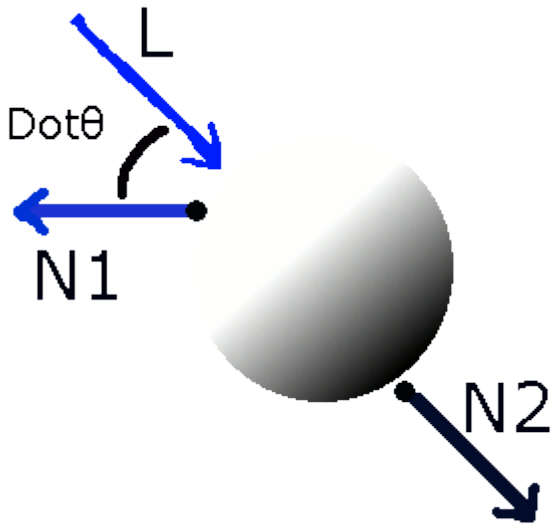
最大値を取得する関数 ＝max()

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。

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。

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 パーティクル

10.1 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;
```

10.2 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。

10.2.1 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。

10.3 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。

10.3.1 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。

10.3.2 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;
```

10.3.3 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

10.4 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。

10.4.1 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);
```

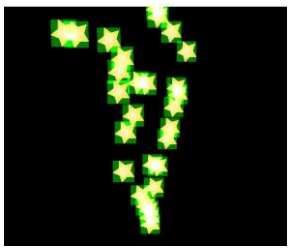
IDirect3DDevice9::`DrawIndexedPrimitive` は現在のレンダリングステートの状態を使用してプリミティブを描画します。上記のレンダリングステートのコードは `DrawIndexedPrimitive` の直前に記述すれば、加算合成を確認できるはずです。

10.4.2 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。

10.4.3 シェーダーの変更

さて、今回のサンプルはレンダリングステートを加算合成を行うように変更した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 );
```