Chapter18-6 コンスタントバッファ　［応用］

シェーダー内のデータの取得・出力にはGPU内のレジスタが使用され、**変数は全てローカル変数**（テンポラリレジスタ）のみ利用できる。
C言語のようなグローバル変数の記述は出力・共有用の指定のあるものを除き、全てが**定数**（定数レジスタ）に保存され**シェーダー内での編集・保存は原則的に不可能**になっている

Program18-6 エラープログラムの例

```
float t = 0.0f;
//! 頂点シェーダ
OutputVS RenderVS( InputVS inVert )
{
	OutputVS	outVert;

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

	t += 0.1f;		//定数のため変更がおこなえず、ここでエラーが発生する！！

	return outVert;
}
```

```
error X3025:
global variables are implicitly constant, enable compatibility mode to allow modification
```

このためシェーダーで描画ごとに表現を変化させるためには、ゲーム側（C++側）のプログラムから変数をシェーダーの定数レジスタに書き込む必要がある。

HLSLでは定数レジスタに効率的に書き込みができるように**定数バッファ（コンスタントバッファ）**が用意されている。コンスタントバッファはC言語の構造体のように複数の値をひとまとめに扱うことができ、DirectX11シェーダーバージョン４．０環境下では１６スロット（１６個のコンスタントバッファ、ただし内２つは内部利用用のため実質は１４個）に４０９６個までのレジスタ（４次元ベクトルを４０９６個まで）を作成できる。

コンスタントバッファはcbufferキーワードで作成できる。
　cbuffer 名前 : register(レジスタ番号)
registerの(レジスタ番号)は省略可能で利用するレジスタ（スロット）の番号をb#（０～１５）で保存するレジスタを指定できる。

コンスタントバッファ内部には通常の変数宣言と同じように記述できる。
　変数の型 名前 : packoffset(レジスタ番号)
packoffset(レジスタ番号)は省略可能で番号をc#でコンスタントバッファ内のどのレジスタに定数を展開するかを指定できる。
ＨＬＳＬで使用できる変数・定数の型は下記のものが存在する。

型	内容
bool	true,falseのみ、C言語のboolと同じ
int	符号あり32Bit整数型、C言語のintと同じ
uint	符号なし32Bit整数型、C言語のunsigned intと同じ
float	32Bit浮動小数型、C言語のfloatと同じ
float2,float3,float4	指定した数値の小数ベクトル型、数値分のfloat型の配列と同じ、それぞれMofライブラリのVector2,Vector3,Vector4に対応
int2,int3,int4	指定した数値の整数ベクトル型、数値分のint型の配列と同じ
vector< 型, 数 >	型にはfloatかintを指定可能

	記述した型の数を指定したベクトル型と同じ
float2x2, float2x3, float2x4, float3x2, float3x3, float3x4, float4x2, float4x3, float4x4	指定した数値の小数行列型、数値分の float 型の２次元配列と同じ float3x3 が Mof ライブラリの Matrix33 と、 float4x4 が Mof ライブラリの Matrix44 と対応
int2x2, int2x3, int2x4, int3x2, int3x3, int3x4, int4x2, int4x3, int4x4	指定した数値の整数行列型、数値分の int 型の２次元配列と同じ
matrix	float4x4 と同じ Mof ライブラリの Matrix44 が対応

Program18-6 コンスタントバッファ作成例

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

コンスタントバッファと同じ構造体をゲーム側でも作成することで、ゲーム側の変数をシェーダー側の定数レジスタに効率的に書きこむことが可能である。コンスタントバッファは書き込みの回数を減らすためにも書き込む頻度やタイミングに合わせて分ける方が効率がいい。

Program18-6 ゲーム側で同様の構造体の作成例

```
//! HLSL 側のコンスタントバッファと同様の構造体
struct cbSceneParam
{
	Vector4	vecViewPos;
	Matrix44	mtxView;
	Matrix44	mtxProj;
}
```

コンスタントバッファを作成する際にはアライメントに注意する必要がある。シェーダーを実行する GPU は SIMD（single instruction multiple data、シングルインストラクション マルチプルデータ）向きに最適化されており、４次元のベクトルデータの演算を一度に効率的におこなうことができる。

そのためコンスタントバッファなど変数をゲーム側から受け渡す時は１６Byte（float 型４つ）にあわせる必要があり、

Program18-6 エラープログラムの例
※vecViewPos はカメラ座標で 3 次元ベクトルだがそのまま作成するとエラーになってしまう。

```
//シェーダー側
//! コンスタントバッファ
cbuffer cbSceneParam : register( b0 )
{
	float3	vecViewPos	: packoffset( c0 );
	matrix	mtxView		: packoffset( c1 );
	matrix	mtxProj		: packoffset( c5 );
}

//ゲーム側
//! HLSL 側のコンスタントバッファと同様の構造体
struct cbSceneParam
{
	Vector3	vecViewPos;
	Matrix44	mtxView;
	Matrix44	mtxProj;
```

```
}
```

ゲーム側のコンスタントバッファの生成でエラー
それを解決しても vecViewPos と mtxView の間にアライメントを埋めるための不要データが入ってしまい
データがずれる。

コンスタントバッファを作成して頂点シェーダーで頂点を操作するプログラムを紹介する。

Program18-6 Sample.hlsl

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

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

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

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

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

//! 頂点属性
struct InputVS
{
	float4		pos				: POSITION;
	float3		normal			: NORMAL;
	float2		Tex				: TEXCOORD;
	float4		color			: COLOR0;
};
struct OutputVS
{
	float4		pos				: SV_POSITION;
};

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

//! 頂点シェーダ
OutputVS RenderVS( InputVS inVert )
```

```
{
	OutputVS   outVert;

	matrix	mtxVP = mul( mtxView, mtxProj );
	float4 Pos = float4(inVert.pos.xyz + inVert.normal.xyz * sin(cbTime.x) , 1.0);
	Pos = mul(Pos, mtxWorld );
	outVert.pos = mul( Pos , mtxVP );

	return outVert;
}

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

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

ゲーム側でシェーダと同じ構造体を作成してシェーダー側に変数を設定する。

Program18-6 Sample.hlsl

```
//INCLUDE
#include    "GameApp.h"

//カメラ
CCamera                         gCamera;
//メッシュ
CMeshContainer                  gMesh;
//シェーダー
CShader                         gShader;
CShaderBind_3DPrimitiveBase     gShaderBind;

//シェーダー側のコンスタントバッファと同様の構造体
struct cbGameParam
{
	Vector4     cbTime;
};
//時間
float                           gTime = 0.0f;

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

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

	//メッシュの読み込み
```

```
	gMesh.Load("player.mom");

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

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

MofBool CGameApp::Update(void){
	//キーの更新
	g_pInput->RefreshKey();

	return TRUE;
}

MofBool CGameApp::Render(void){
	//描画処理
	g_pGraphics->RenderStart();
	//画面のクリア
	g_pGraphics->ClearTarget(0.0f,0.0f,1.0f,0.0f,1.0f,0);

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

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

	//プログラム側で時間経過を実行
	gTime += CUtilities::GetFrameSecond();
	//シェーダーに送るバッファを作成
	cbGameParam sb;
	sb.cbTime.x = gTime;
	//シェーダーにバッファを送る
	gShaderBind.GetShaderBuffer(0)->SetBuffer(&sb);

	//メッシュの描画
	CMatrix44 matWorld;
	gMesh.Render(matWorld, &gShader, &gShaderBind);

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

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

Mof ライブラリでコンスタントバッファの作成は CShaderBuffer クラスを作成しておこなう。初期化の際に CreateShaderBuffer 関数でシェーダーのコンスタントバッファと結びつけたバッファを作成する。

CShaderBind クラスのメソッド

MofBool CreateShaderBuffer(LPCMofChar pName, MofU32 size);	
コンスタントバッファの生成をおこなう	
引数	[in] pName　　バッファのシェーダー側の名前 [in] size　　作成するバッファのサイズ
戻り値	生成に成功した場合 TRUE,生成失敗した場合 FALSE

生成したバッファにゲーム側からデータを設定するには作成したバッファを GetShaderBuffer 関数で取得して、取得したバッファに対して SetBuffer 関数でデータを設定する。

CShaderBind クラスのメソッド

LPShaderBuffer GetShaderBuffer(MofU32 n);	
コンスタントバッファを取得する	
引数	[in] n　　バッファの番号
戻り値	コンスタントバッファ

LPShaderBuffer GetShaderBufferByName(LPCMofChar pName);	
コンスタントバッファを取得する	
引数	[in] pName　　バッファの名前
戻り値	コンスタントバッファ、存在しない場合 NULL