オブジェクト指向と
ゲームプログラミング

▶ DirectX Graphics 3D編 ― 第２回 頂点バッファへの書き込みと描画 ◀

頂点バッファへの書き込みと読み込み

頂点バッファに頂点データを書き込んだり、読み込んだりするには、IDirect3DVertexBuffer9::Lockメソッドを呼び出し、頂点バッファをロックしてバッファへのポインタを取得します。
得られたポインタをもとに、CopyMemory関数などのメモリ操作関数やポインタ操作を用いることにより、バッファへデータを書き込むことができます。頂点バッファが書き込み専用でない場合は、頂点データを読み取ることもできます。
頂点バッファへのアクセスが終了した場合は、IDirect3DVertexBuffer9::Unlockメソッドを呼び出してロックを解除します。ロックを解除しないと、レンダリングできません。

IDirect3DVertexBuffer9::Lockメソッド

―説明―
Lockメソッドは、頂点バッファをロックし、バッファへのポインタを取得します。

―書式―
HRESULT Lock(UINT *OffsetToLock*, UINT *SizeToLock*, VOID** *ppbData*, DWORD *Flags*);

―パラメーター―
１つ目の引数(OffsetToLock)は、ロックする領域のオフセット(先頭からのバイト数)です。頂点バッファ全体をロックするには、この引数と２つ目の引数の両方のパラメータに０を指定します。

２つ目の引数(SizeToLock)は、ロックするバイト数です。

３つ目の引数(ppbData)は、頂点バッファへのポインタを格納する変数のアドレスです。

４つ目の引数(Flags)は、ロックフラグです。以下のフラグの組み合わせを指定します。

D3DLOCK_DISCARD	ロックするすべての領域を書き込み専用にします。このフラグを用いた場合、つねに新しい領域のポインタが返されるので、(古い領域にあるデータの)レンダリングの停止およびレンダリング終了の待機といったオーバーヘッドがなくなります。ただし、バッファは動的なメモリでなければなりません
D3DLOCK_NOOVERWRITE	バッファ内のデータを上書きしないことが保証されます。古いデータをレンダリングしながら新しいデータを書き込むことができます
D3DLOCK_READONLY	読み込み専用としてロックします。Unlock処理の速度が向上する場合があります
D3DLOCK_NO_SYSLOCK	システムロックを使用しません。ロックが長時間に及ぶ場合、このフラグを指定すると他のアプリケーションへの悪影響を押さえます

―戻り値―
成功した場合はD3D_OK、それ以外はエラーコードを返します。

IDirect3DVertexBuffer9::Unlockメソッド

―説明―
Unlockメソッドは、頂点バッファのロックを解除します。

―書式―
HRESULT Unlock();

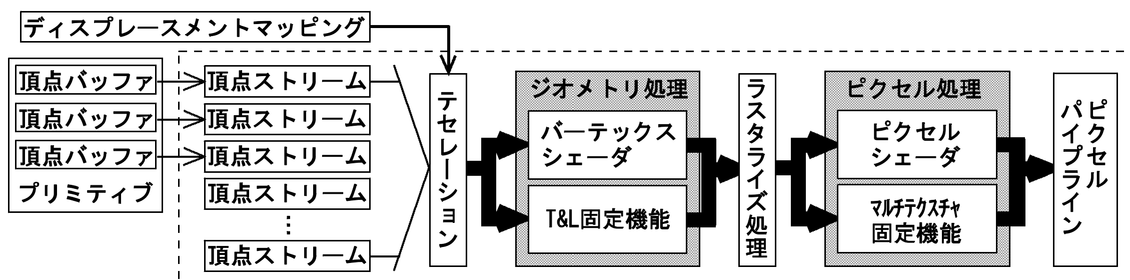
―戻り値―
成功した場合はD3D_OK、それ以外はD3DERR_INVALIDCALLを返します。

```
// 頂点バッファの書き込み(pVertexBufferは初期化済みのDirect3DVertexBuffer9オブジェクト)
// 頂点バッファをロックしてバッファへのポインタを得る
LPVOID   pBuffer;
pVertexBuffer->Lock(0, 0, &pBuffer, 0);

// 頂点データの書き込み
// 座標変換済み頂点
```

```
struct TLVERTEX {
    float       x, y, z;      // 頂点座標
    float       rhw;          // 法線
    D3DCOLOR    color;        // 頂点色
};

TLVERTEX*   vt = (TLVERTEX*)pBuffer;

vt[0].x     = 260.0f;
vt[0].y     = 180.0f;
vt[0].z     =   0.0f;
vt[0].rhw   =   1.0f;
vt[0].color = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f);

// ロック解除
pVertexBuffer->Unlock();
```

頂点バッファのレンダリング

すべてのプリミティブは、以下の流れでターゲットに出力されます。



プリミティブを描画するには、頂点バッファを生成し、頂点データを設定します。次に、その頂点バッファをデータストリーム(頂点ストリーム)にバインド(結合)します。この状態でプリミティブのレンダリングを行うメソッドを呼び出すと、テセレーション以降が実行されてレンダリングターゲットにその結果が出力されます。このとき、すべての頂点ストリームがレンダリングされます。これを利用し、複数の頂点バッファをバインドしておき、一度のレンダリングで大量の頂点を出力したり、異なるストリーム間の頂点をブレンディングしたりすることができます。

頂点バッファをデータストリームにバインドするには、IDirect3DDevice9::SetStreamSourceメソッド、プリミティブのレンダリングは、IDirect3DDevice9::DrawPrimitiveメソッドで行います。
また、バーテックスシェーダを使用しない場合は、DrawPrimitiveを呼び出す前に、どのような頂点フォーマットなのかをSetFVFメソッドでシステムに設定しておく必要があります。

IDirect3DDevice9::SetStreamSourceメソッド

－説明－
SetStreamSourceメソッドは、頂点バッファをデータストリームにバインドします。

－書式－
HRESULT SetStreamSource(UINT *StreamNumber*, IDirect3DVertexBuffer9* *pStreamData*,
UINT *OffsetInBytes*, UINT *Stride*);

－パラメーター－
１つ目の引数(StreamNumber)は、０から始まる、バインドするストリームの番号です。

２つ目の引数(pStreamData)は、ストリームにバインドする頂点バッファのインタフェースを指定します。

３つ目の引数(OffsetInBytes)は、ストリームの先頭から頂点データの先頭までのオフセットです。

４つ目の引数(Stride)は、次の頂点までのバイト数です。

－戻り値－
成功した場合はD3D_OK、それ以外はエラーコードを返します。

IDirect3DDevice9::DrawPrimitiveメソッド

－説明－

DrawPrimitiveメソッドは、すべてのデータストリームを指定された方法でレンダリングします。

－書式－

```
HRESULT DrawPrimitive(D3DPRIMITIVETYPE PrimitiveType,
                      UINT StartVertex, UINT PrimitiveCount);
```

－パラメーター－

１つ目の引数(PrimitiveType)は、レンダリングの方法です。

D3DPT_POINTLIST	点リスト	D3DPT_TRIANGLELIST	三角形リスト
D3DPT_LINELIST	ラインリスト	D3DPT_TRIANGLESTRIP	三角形ストリップ
D3DPT_LINESTRIP	ラインストリップ	D3DPT_TRIANGLEFAN	三角形ファン

２つ目の引数(StartVertex)は、レンダリングする最初の頂点のインデックスです。この頂点を先頭にレンダリングが行われます。

３つ目の引数(PrimitiveCount)は、レンダリングするプリミティブの数です。

－戻り値－

成功した場合はD3D_OK、それ以外はエラーコードを返します。

```
// 頂点バッファのレンダリング(シーンの開始および終了は省略)
pD3DDevice->SetFVF(D3DFVF_XYZRHW | D3DFVF_DIFFUSE | D3DFVF_TEX1);   // FVF設定
pD3DDevice->SetStreamSource(0, pVertexBuffer, 0, Stride);           // ストリームに結合
pD3DDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 2);                // レンダリング
```

課　　題

頂点バッファクラスに、頂点バッファのロックおよびロック解除、レンダリングを行う機能を追加しましょう。

(１)頂点バッファをロックするLock関数と、ロックを解除するUnlock関数を作成します。以下のプロトタイプをIVertexBufferクラス定義の適切な場所に追加しましょう。

```
virtual LPVOID Lock() = 0;
virtual void Unlock() = 0;
```

(２)CVertexBufferクラスにLock関数およびUnlock関数を追加します。これらの関数のプロトタイプを(１)をもとに作成し、適切な場所に追加しましょう。

(３)Lock関数とUnlock関数を実装します。実装します。以下のプログラムを完成させ、ソースファイルに追加しましょう。

```
/*****************************************************************************************/
/*                                        ロック                                          */
/*****************************************************************************************/
LPVOID CVertexBuffer::Lock()
{
    LPVOID   pBuffer;
    m_pVertexBuffer->????(0, 0, &pBuffer, 0);

    return pBuffer;      // 得られたポインタを返す
}

/*****************************************************************************************/
/*                                      ロック解除                                        */
/*****************************************************************************************/
void CVertexBuffer::Unlock()
{
    ※ここは各自考えましょう;
}
```

Null Objectパターンを用いているため、CVertexBufferクラスの関数では、m_pVertexBufferがNULLかどうかを調べる必要はありません(NULLの場合は、CNullVertexBufferオブジェクトが生成されるようになっています)。

(４)CNullVertexBufferクラスに「何もしない」Lock関数およびUnlock関数を実装します。以下のプログラムを適切な場所に追加しましょう。

```
virtual LPVOID Lock() { return NULL; }
virtual void Unlock() {}
```

(５)頂点バッファをレンダリングするRender関数を作成します。以下のプロトタイプをIVertexBufferクラス定義の適切な場所に追加しましょう。

```
virtual void Render(const D3DPRIMITIVETYPE inType, const UINT inStart, const UINT inCount) = 0;
```

Render関数の仕様は、以下のとおりです。

Render 描画

頂点バッファをデータストリーム０に結合し、格納されている頂点データをレンダリングします。

書式 void Render(const D3DPRIMITIVETYPE inType, const UINT inStart, const UINT inCount);

inType	プリミティブのレンダリング方法。以下のフラグを指定します D3DPT_POINTLIST D3DPT_LINELIST D3DPT_LINESTRIP D3DPT_TRIANGLELIST D3DPT_TRIANGLESTRIP D3DPT_TRIANGLEFAN
inStart	レンダリングする最初の頂点のインデックス。この頂点を先頭にレンダリングが行われます
inCount	レンダリングするプリミティブ数

(６)CVertexBufferクラスにRender関数を追加します。Render関数のプロトタイプを(５)をもとに作成し、適切な場所に追加しましょう。

(７)Render関数を実装します。実装します。以下のプログラムを完成させ、ソースファイルに追加しましょう。

```
/**************************************************************************************************/
/*                                        レンダリング                                             */
/**************************************************************************************************/
void CVertexBuffer::Render(const D3DPRIMITIVETYPE inType, const UINT inStart, const UINT inCount)
{
    // 描画
    m_pD3DDevice->??????(m_FVF);                                  // FVF設定
    ※ここは各自考えましょう;                                      // データストリーム０にバインド
    ※ここは各自考えましょう;                                      // レンダリング
}
```

(８)CNullVertexBufferクラスに「何もしない」Render関数を実装します。以下のプログラムを適切な場所に追加しましょう。

```
virtual void Render(const D3DPRIMITIVETYPE inType, const UINT inStart, const UINT inCount) {}
```

(９)プリミティブを描画してみましょう。CTestSceneクラスに、以下のメンバを追加します。

```
IVertexBuffer*    m_pVertex;
```

(10)頂点バッファ生成し、頂点データを設定します。CTestSceneクラスのコンストラクタを以下のように変更しましょう。

```
CTestScene::CTestScene()
{
    // ここに、機能テストの初期化処理を記述します
    // 頂点バッファの生成(トランスフォーム済みライティング済みの頂点を１００個格納するバッファ)
    m_pVertex = DXGraphics().CreateVertexBuffer(sizeof(DXGTLVERTEX) * 100, DXGFVF_TLVERTEX,
                                                sizeof(DXGTLVERTEX));
```

```
    // 頂点バッファをロックし、ポインタを得る
    DXGTLVERTEX*   vt = (DXGTLVERTEX*)m_pVertex->Lock();

    // 頂点０
    vt[0].x     = 260.0f;
    vt[0].y     = 180.0f;
    vt[0].z     =   0.0f;
    vt[0].color = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f);

    // ロック解除
    m_pVertex->Unlock();

    FPSTimer().Reset();     // タイマリセット
}
```

(11)CTestSceneクラスのデストラクタを以下のように変更し、テストシーン終了時に生成したすべての頂点バッファが解放されるようにしましょう。

```
CTestScene::~CTestScene()
{
    // ここに、機能テストの解放処理を記述します
    DXGraphics().ReleaseAllVertexBuffers();
}
```

(12)CTestScene::Active関数を以下のように変更し、プリミティブを描画してみましょう。

```
int CTestScene::ActiveProc()
{
    // ここに、機能テストのメイン処理を記述します

    // ここに、機能テストの描画処理を記述します
    if(FPSTimer().IsSkip())
        return 0;

    DXGraphics().Clear();                          // 画面クリア

    DXGraphics().BeginScene();                     // シーン開始

    m_pVertex->Render(D3DPT_POINTLIST, 0, 1);   // 頂点バッファのレンダリング

    DXGraphics().EndScene();                       // シーン終了

    DXGraphics().UpdateFrame();                    // フレーム更新

    return 0;
}
```

(13)点を２つ描画してみます。以下のプログラムをCTestScene::CTestScene関数の適切な場所に追加しましょう。

```
    // 頂点１
    vt[1].x     = 380.0f;
    vt[1].y     = 300.0f;
    vt[1].z     =   0.0f;
    vt[1].color = D3DXCOLOR(0.0f, 1.0f, 0.0f, 1.0f);
```

(14)以下のようにレンダリングするプリミティブ数を２にすれば、点が２つ描画されます。

```
    m_pVertex->Render(D3DPT_POINTLIST, 0, 2);    // 頂点バッファのレンダリング
```

(15)２点を結んで線にしてみましょう。頂点バッファのレンダリングを以下のように変更すると、頂点が結ばれて線になります。

```
    m_pVertex->Render(D3DPT_LINELIST, 0, 1);     // 頂点バッファのレンダリング
```

(16)頂点を３つ定義し、３点を結ぶ三角形にしてみます。すでに２つの頂点が定義されているので、あと１つ頂点を追加する必要があります。以下のプログラムをCTestScene::CTestScene関数の適切な場所に追加しましょう。

```
// 頂点２
vt[2].x     = 260.0f;
vt[2].y     = 300.0f;
vt[2].z     =   0.0f;
vt[2].color = D3DXCOLOR(0.0f, 0.0f, 1.0f, 1.0f);
```

(17)三角形では、線が３本必要になります。以下のようにプログラムを変更し、線が３本レンダリングされるようにしましょう。

```
m_pVertex->Render(D3DPT_LINELIST, 0, 3);     // 頂点バッファのレンダリング
```

(18)線を３本レンダリングするとき、ラインリストの場合は、２つの頂点を開始点と終了点のペアで扱うので、頂点は６つ必要になります。以下のようにレンダリングの方法を変更し、ラインストリップにしてみましょう。

```
m_pVertex->Render(D3DPT_LINESTRIP, 0, 3);        // 頂点バッファのレンダリング
```

(19)線をレンダリングするとき、ラインストリップの場合は「レンダリングする線の数＋１」の頂点が必要になります。以下のプログラムをCTestScene::CTestScene関数の適切な場所に追加し、頂点をさらに１つ増やしましょう。

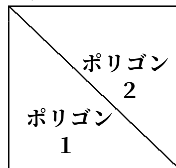
```
// 頂点３
vt[3] = vt[0];
```

(20)レンダリング方法をラインストリップから三角形リストに変更してみます。以下のようにレンダリングの方法を変更しましょう。

```
m_pVertex->Render(D3DPT_TRIANGLELIST, 0, 1);     // 頂点バッファのレンダリング
```

三角形リストの場合は、３つの頂点が結ばれて１つの三角形になります。

(21)以下のように頂点を定義し、三角形を２つ合わせて四角形を描画しましょう。



発展問題　頂点バッファへの書き込み効率を向上させるための機能強化を行いましょう。

頂点バッファに書き込みを多く行う場合、動的なメモリを使用するようにするとレンダリングと書き込み操作を同時に行えるようになります。また、頂点バッファをVRAMより書き込み効率の良いメモリに配置させれば、より効率が向上します。

(１)頂点バッファを「書き込み専用」として生成できるようにします。CDXGraphics9::CreateVertexBuffer関数に以下のように変更しましょう(※２カ所)。

```
IVertexBuffer* CreateVertexBuffer(const UINT inSize, const DWORD inFVF, const UINT inStride,
                                  const bool inWriteOnly = false);
```

追加された新しい引数inWriteOnlyは、書き込み専用を示すフラグです。この引数がfalseの場合は、これまでと同じ読み書きできる頂点バッファを生成します。trueの場合は、書き込み専用バッファを生成し、書き込み効率を向上させます(ただし、レンダリング速度が若干低下する場合があります)。

（２）頂点バッファの生成方法を書き込み専用フラグによって変更します。CDXGraphics9::CreateVertexBuffer関数本体を以下のように変更しましょう。

```
/*************************************************************************************************/
/*                                        頂点バッファ生成                                        */
/*************************************************************************************************/
IVertexBuffer* CDXGraphics9::CreateVertexBuffer(const UINT inSize, const DWORD inFVF, const UINT inStride)
{
#ifdef _DEBUG
    if(m_pD3DDevice == NULL) {
        ::OutputDebugString("*** Error - Direct3DDevice9未初期化(CDXGraphics9_CreateVertexBuffer)\n");
        return NULL;
    }
#endif
    // フラグ設定
    DWORD     Usage;
    D3DPOOL   Pool;
    if(inWriteOnly == false) {
        Usage = 0;                                              ※追加
        Pool  = D3DPOOL_MANAGED;
    } else {
        Usage = D3DUSAGE_DYNAMIC | D3DUSAGE_?????????;
        Pool  = D3DPOOL_SYSTEMMEM;
    }

    // 頂点バッファ生成
    IVertexBuffer*            pVertex;
    IDirect3DVertexBuffer9*   pBuffer;
    if(m_pD3DDevice->??????????????????(inSize, Usage, inFVF, Pool, &pBuffer, NULL) == D3D_OK) {
        pVertex = new CVertexBuffer(m_pD3DDevice, pBuffer, inFVF, inStride, inWriteOnly);
    } else {
        ::OutputDebugString("*** Error - 頂点バッファ生成失敗(CDXGraphics9_CreateVertexBuffer)\n");
        pVertex = new ??????????????????();  // 生成に失敗した場合は、NULLオブジェクトを生成
    }
    SafeRelease(pBuffer);

    m_VertexBuffer.push_back(pVertex);      // 頂点バッファリストに追加する

    return pVertex;
}
```

（３）CVertexBufferクラスにロックするときのフラグを格納するメンバ変数を追加します。以下のメンバを適切な場所に追加しましょう。

```
DWORD   m_LockFlags;  // ロックフラグ
```

書き込み専用の頂点バッファは、生成しただけでは効果が現れません。バッファをロックするときに正しいフラグを指定して初めて効果が現れます。m_LockFlagsメンバには、ロックするときに指定するフラグを格納します。

（４）CVertexBufferクラスのコンストラクタを以下のように変更し、書き込み専用バッファかどうかを受け取るようにしましょう(※２カ所)。

```
CVertexBuffer(IDirect3DDevice9* pD3DDevice, IDirect3DVertexBuffer9* pVertexBuffer,
              const DWORD inFVF, const UINT inStride, const bool inWriteOnly);
```

（５）CVertexBufferクラスのコンストラクタ本体を変更し、m_LockFlagsメンバの値を設定するようにします。以下のプログラムを完成させ、適切な場所に追加しましょう。

```
// ロックフラグ設定
if(inWriteOnly == false)
    m_LockFlags = 0;
else
    m_LockFlags = D3DLOCK_??????? | D3DLOCK_???????????;
```

（６）バッファをロックするときにm_LockFlagメンバを用いるようにします。ロック処理を以下のように変更しましょう。

```
m_pVertexBuffer->????(0, 0, &pBuffer, m_LockFlags);
```