ESライブラリ＆＆
ゲームプログラミング

○　３Ｄゲームプログラミング入門　○

メインループ

・ゲームの仕組みはセルアニメーションと同じ
・１秒間に数十回程度のループを行う
・ループ内でゲーム固有の処理を行い、画面に描画する

概要

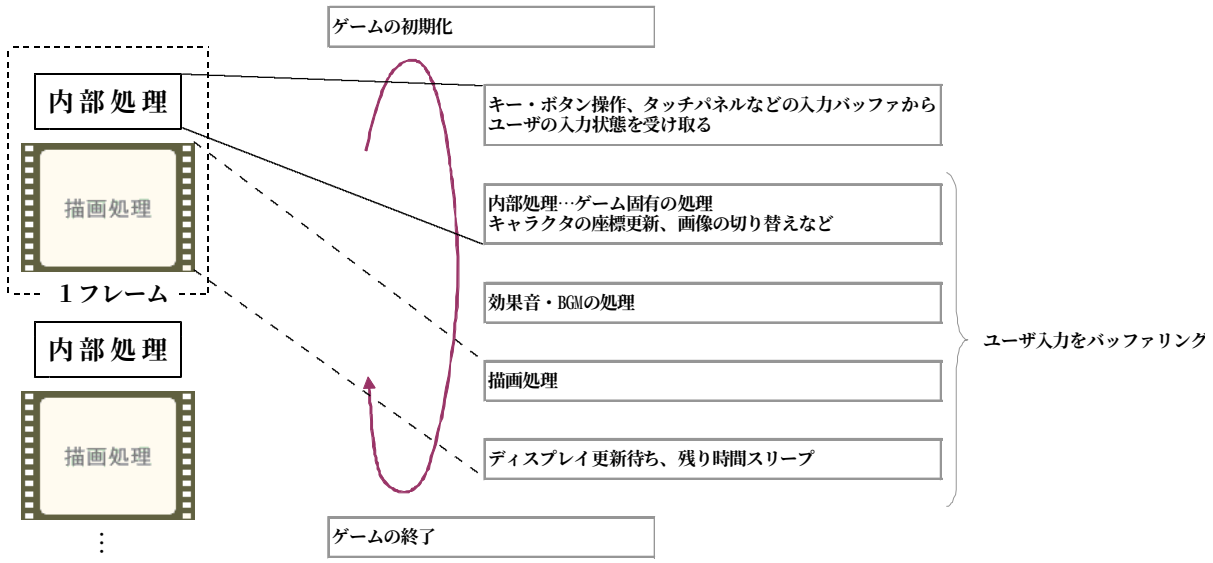
ゲームプログラムは、キャラクターや背景など、いろいろな「もの」の状態を時間にそって更新し、画面に描画するプログラムと考えることができます。これを非常に単純化すると、

・「もの」の状態を更新(内部処理 ＝ GameMain::Update)
・「もの」を画面に描画(描画処理 ＝ GameMain::Draw)

の２つにまとめることができます。この処理の繰り返しがゲームのメイン処理(メインループ)になります。ループ１回ぶんを「１フレーム」と呼び、アニメの１コマに相当します。１フレームごとに、キャラクターの位置や状態、背景などを少し更新して描画、という処理を繰り返しているということです。多くのゲームでは秒間６０フレームを採用し、滑らかな動きを実現しています。

▲「状態」を少しずつ変更して描画することにより、動いているように見せる

簡易図



フレームレート

・１秒間あたり何度画面が更新されるかを表す
・単位はfps(Frames Per Second)
・ディスプレイではリフレッシュレートと呼び、単位はHz

実例

・コンシューマ、アーケードは30または60fps
・携帯アプリは20～30fps
・映画は24fps、テレビアニメは8fps

GameMainクラスの機能

・Initialize	グラフィクス、サウンド以外の初期化
・LoadContent / UnloadConten	グラフィクス、サウンドなどリソースの初期化・解放
・Update / Draw	ゲームのメインループ

概要

プログラミングの核となるのがGameMainクラスです。Windowsで動作するゲームの最小の機能のみ定義してあります。このクラスには、Initialize, LoadContent, UnloadContent, Update, Drawの５つの機能(関数)のひな形が用意されています。

Initialize

キャラクターの座標やステータス、コンフィグやユーザーデータといった画像や音声以外の数値データの設定、初期化を行います。

LoadContent

グラフィクスやサウンドといったコンテンツデータの読み込みを行います。

UnloadContent

LoadContentで読み込んだグラフィクスやサウンドといったコンテンツデータの解放を行います。

Update

「内部処理」を記述するための部分です。キャラクターや背景の座標や状態変更、衝突判定、各種計算、ゲームの進行状況の管理など描画以外のプログラムを記述します。処理落ちがなければ、デフォルトで16.6ms毎、秒間60回実行されるようになっています。

Draw

「描画処理」を記述するための部分です。内部処理によって計算された情報をもとに、画面に描画するためのプログラムを記述します。ここも処理落ちがなければ、デフォルトで16.6ms毎、秒間60回呼ばれるようになっています。

ポリゴン

・3Dのオブジェクトは、ポリゴンと呼ばれる「板」を組み合わせて構成される
・ポリゴンは多角形という意味だが、三角形は必ず平面でもっとも扱いやすいため、
　ほとんどの場合は三角形が用いられている
・複雑なオブジェクトも、ポリゴンが組み合わされて造られている

ポリゴン

ポリゴン(polygon)とは、3Dグラフィクスで立体を表現するときに使われる多角形のことです。扱いやすさから、ほとんどの環境で三角形が使われています。

コンピュータで立体図形を扱う場合、物体表面を三角形に分割してデータ化します。分割数を増やせば増やすほど精細なモデルになりますが、比例して計算量が増えるため、描画に時間がかかります。

１秒間に処理できるポリゴン数がグラフィックカードやゲーム機の性能の指標として使われることがあります。以下の表のように、年代とともに飛躍的に能力が向上しています(注：任天堂の機種は実測値、それ以外は理論値)。

機種名	SS	PS	N64	DC	PS2	GC	Xbox	Xbox360	PS3	Wii
発売年	94年11月	94年12月	96年6月	98年11月	00年3月	01年9月	01年11月	05年12月	06年11月	06年12月
秒間ポリゴン数	90万	36万	10万	300万	7500万	1200万	1.25億	5億	3億	1600万

▲据え置き型ゲーム機のポリゴン描画能力

機種名	DS	PSP	3DS	PSVita
発売年	04年11月	04年12月	11年2月	11年12月
秒間ポリゴン数	12万	3300万	1530万	1.33億

▲携帯型ゲーム機のポリゴン描画能力

モデルデータ

どんな複雑なモデルでも、三角形が組み合わされて造られています。三角形は３つの頂点の集まりです。3Dグラフィクスでは、モデルをファイルに保存する際は、画像ではなく頂点の集まりとして保存しています。

モデルの保存形式は、画像のビットマップやJPEGのような標準フォーマットがなく、FBXやOBJなど、各社独自フォーマットが乱立していますが、保存形式の１つであるｘ形式(.x：DirectX Retained Mode形式)では、下記のようにモデルの基準点からの相対位置を頂点情報として、そのまま保存しています。

```
Mesh {
 4432;
 -34.720058;-12.484819;48.088928;,
 -25.565304;-9.924385;26.239328;,
 -34.612186;-1.674418;34.789925;,
 0.141491;7.622670;25.743210;,
 -34.612175;17.843525;39.827816;,
 -9.608727;27.597115;38.148296;,
              ⋮
```

このような頂点情報をモデリングツールを使って作成すれば、複雑な形状のモデルであってもプログラムでは読み込むだけで、簡単に描画することができます。

課　　題

モデルデータを読み込み、表示してみましょう。

１．まず、モデルのデータを保存・管理する場所が必要なので宣言します。以下のプログラムを"GameMain.h"の「// 変数宣言」の下に追加しましょう。

```
// 変数宣言
MODEL    town;
```

２．ｘ形式のファイルを上記"town"に読み込みます。以下のプログラムをGameMain::LoadContentに追加しましょう。

```
// モデル読み込み
town = GraphicsDevice.CreateModelFromX( TEXT("Town/Town.x") );
```

この１行で、指定したｘ形式のファイルからモデルデータが読み込まれます。以降は、"town"に備えられた機能を使うことにより、モデルを制御することができます。

３．２．で作成した「model」を描画しましょう。

描画はGameMain::Drawで行います。以下のプログラムを適切な場所追加しましょう。

```
// モデル描画
town->Draw();
```

画面に無数の「点」が表示されれば成功です。

シェーディング

・オブジェクトに「陰」をあたえる演算
・光が当たらない暗い部分には濃い色を、光が当たる明るい部分には薄い色を付ける

概要

3Dグラフィクスは、視点(カメラ)、光源(ライト)、モデルの位置や向きから、表示すべき大きさや色が計算され、その結果が画面に出力されます。この一連の演算の中で、色づけを行う処理が「シェーディング」です。
演算能力が低かった80年代は、シェーディングに膨大な時間がかかっていたため、点を結んで線にした「ワイヤーフレーム」という手法で空間を構築していました。90年代にはいると、CPUやGPUの処理能力が大幅に向上したため、高度なシェーディングがリアルタイムに行えるようになり、ゲームでも広く使われるようになりました。

課　　題

モデルにシェーディング処理を施し、色を付けてみましょう。

シェーディング方法には大きく分けて２つあります。１つは、現在主流となっているプログラマブルシェーダーと呼ばれるプログラムによって色づけを行う方法です。もう１つは、レンダリングステートと呼ばれる、あらかじめ用意された計算方法の中から選択するというものです。モデルごとに、自由に切り換えて色づけを行うことができます。

１．レンダリングステートを変更し、「ワイヤーフレーム」でモデルを描画してみましょう。以下のプログラムをGameMain::Drawの適切な場所追加しましょう。

```
// ワイヤーフレームモードに設定
GraphicsDevice.SetRenderState(FillMode, FillMode_WireFrame);
```

２．「面」が構成されている部分を塗りつぶしてみましょう。１．のプログラムを以下のように変更しましょう。

```
// サーフェスモードに設定
GraphicsDevice.SetRenderState(FillMode, FillMode_Solid);
```

カメラ

・3Dグラフィクスでは、さまざまなモデルを配置し、仮想世界を構築する
・広大な仮想世界の「どこ」から「どの方向」を見るかの設定をするのが、カメラ(視点)

カメラ

3Dグラフィクスでは、オブジェクトを多数配置して仮想世界を構築します。次に行うのは、仮想世界の「どこ」から「どの方向」を見るのか、という「カメラ(視点)」の設定です。

3Dグラフィクスの仮想世界は、無限大の広大な領域を持っています。そのまますべての領域を画面に描画すると、処理時間が膨大に掛かるだけでなく、ほぼすべてのオブジェクトが１ピクセル以下の見えないものになってしまいます。そこで、広大な空間の中から、一部を切り取って画面に描画するようにします。この「切り取り方」を設定するのがカメラといえます。

課　　題

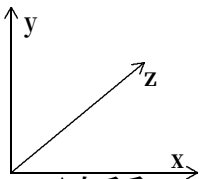
視点を変更し、モデルの中心から見た世界を画面に描画してみましょう。

１．カメラを座標(0.0, 0.0, 0.0)に配置し、向きを角度(0.0, 0.0, 0.0)に設定してみます。以下のプログラムをGameMain::Drawの適切な場所追加しましょう。

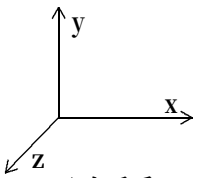
```
// カメラ設定
Camera->SetView(Vector3(0.0f, 0.0f, 0.0f), Vector3(0.0f, 0.0f, 0.0f));
                        ①                          ②
```

２．下線部①は座標、②はカメラの向き(度数法による角度)です。この値を変更してみましょう。

ヒント１：下線部の並びは、ｘ，ｙ，ｚの順です。それぞれの軸は以下の「左手系」になります。



▲左手系



▲右手系

ヒント２：下線部②は、それぞれの軸を中心としてカメラが回転します。このとき、正の値は反時計回り、負の値は時計回りに回転します。

テクスチャ

・テクスチャは、物体表面の質感を表現するための画像のこと
・ポリゴンの凹凸だけでは表現できない複雑な模様も簡単に表現できる
・単に画像を貼るだけではなく、テクスチャマップを応用して高低、凹凸を表現するバンプマップ、ディスプレースメントマップなど、さまざまなマッピング技術が開発されている

概要

テクスチャとは、物体表面の質感を表現するために、ポリゴンに貼り付ける画像のことをいいます。同じ立方体のモデルでも、金属のテクスチャを貼り付ければ金属片に見え、木目のテクスチャを貼り付ければ木片に見えます。

テクスチャを用いると、ポリゴンの組み合わせでは表現できない複雑な模様を簡単に表現することができます。また、少ないポリゴンでリアリティあるオブジェクトを表現することもできます。

課　　題

ポリゴンにテクスチャを貼り付けてみましょう。

テクスチャは、①どの画像を使うのか、②画像のどこをどの点に対応させるか、という設定が必要ですが、あらかじめモデリングツールで設定されています。そのため、所定の画像を所定の場所に置けば、モデルファイルの読み込み時に適用されます。

1．Textureフォルダのなかにテクスチャ画像が格納されています。どのような内容か確認してみましょう。

2．Textureフォルダのなかに入っているすべての画像を"Town.x"と同じフォルダに移動しましょう。

3．プログラムを実行し、結果を確認しましょう。

スプライト

・3D専用環境では、2Dグラフィクスの描画はすべてポリゴンとテクスチャを組み合わせて行う
・3D専用環境では、上記のことを「スプライト」と呼ぶことがある
・本来のスプライトはキャラクターをハードウェアで高速に表示する機能
・3Dの機能を用いるので「拡大縮小」「回転」「半透明」「z座標による重なり」をサポート

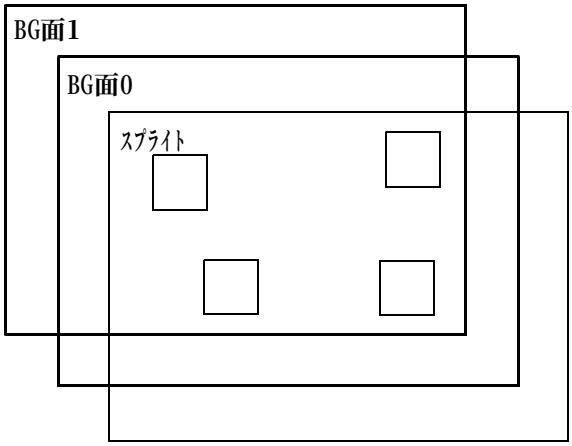
概要

スプライトとは、キャラクターを高速に表示するための機能のことで、背景などほかの画像を壊すことなく独立して制御でき、透過色や重なり順序を指定することができます。

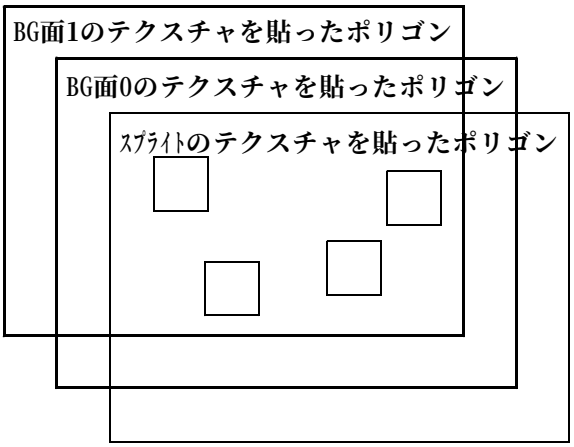
本来のスプライトは、ハードウェアで実現される機能で、背景画像(BG面)とは別に、多数の小さな画像を用意しておき、画面に合成して表示するものです。複数の背景やスプライトと重ね合わせて合成表示ができるようになっており、ほかのスプライトと重なったときに、どれを前面に表示するかが指定できるようになっています。CPUに負担をかけずにキャラクターを表示することができました。

現在では、CPUが高速化され、グラフィック機能も3D専用になっているため、ハードウェアでスプライトをサポートしている機種はほとんどなく、ポリゴンとテクスチャを用いて擬似的にスプライトを扱えるようにしています。

現在主流となっているゲーム機やPC環境では、2Dグラフィクスの機能(スーパーファミコンやゲームボーイアドバンスのような旧来のスプライトおよびBG機能)は実装されていないため、2D表示はポリゴンを平面的に並べ、テクスチャを貼り付けて擬似的に再現する疑似スプライトとなります。内部では面倒な作業を行っているように感じますが、3Dの機能を用いて描画(レンダリング)されるので「スケーリング」「回転」「アルファブレンディング」といった機能をGPUで高速に行うことができます。



▲スーパーファミコンやセガサターンはBG面とスプライトをハードウェアで合成して画面に表示



▲3D専用環境ではすべて「ポリゴン」＋「テクスチャ」で表示。重なり順序は描画する順序やｚ座標で指定

課　題

背景画像を読み込み、描画してみましょう。

1．まず、背景画像を保存・管理する場所が必要なので宣言します。以下のプログラムを"GameMain.h"の「// 変数宣言」の下に追加しましょう。

```
// 背景
SPRITE    sky;
```

2．画像を上記"sky"に読み込みます。以下のプログラムをGameMain::LoadContentに追加しましょう。

```
// 画像読み込み
sky = GraphicsDevice.CreateSpriteFromFile( TEXT("Town/BlueSky.jpg") );
```

この１行で、指定した画像ファイルが"sky"に読み込まれます。以降は、"sky"に備えられた機能を使うことにより、画像を制御することができます。

3．２．で読み込んだ「sky」を描画しましょう。

描画はGameMain::Drawで行います。以下のプログラムを適切な場所追加しましょう。

```
// 背景描画
SpriteBatch.Begin();
SpriteBatch.Draw(sky, Vector3(0.0f, 0.0f, 1.0f));
SpriteBatch.End();
```

ポイント：描画の順番は、3D環境の場合は、手前から描画するようにします。3D環境では、奥の画像は描画されないため、あらかじめより手前のものを描画しておくことにより、奥になるモデルや画像のピクセル描画をキャンセルさせることができます。画像の上書きが行われなくなる分だけ高速になります。

マテリアル

・マテリアルは、モデルの「光の反射」を設定する
・モデルの色は、光の反射によって決定される
・設定できる反射は「ディフューズ」「アンビエント」「スペキュラー」「エミッシブ」

概要

マテリアルは、表面属性ともいわれるもので、モデル表面の質感を表現する機能です。マテリアルは、光の反射と放射という属性を持ちます。光の反射には「ディフューズ反射」「アンビエント反射」「スペキュラー反射」があります。モデルの色は、これらの反射によって決定されます。また、光の放射には「エミッシブ」というモデル自体を発光させる属性があります。

ディフューズ反射(拡散反射)
ディフューズ反射は、光を一定の方向に反射し陰影を与えます。光の当たる角度で明るさが決まり、光に向いているところほど明るくなります。モデルの陰の色は、この反射の影響を大きく受けます。

アンビエント反射(環境光)
アンビエント反射は、光の反射方向がない反射で、モデル全体に均等に色を与えます。モデルの基本色となります。

スペキュラー反射(鏡面反射)
スペキュラー反射は、表面上での乱反射を再現し、モデルに質感を与えます。物体の光沢を表す反射で、モデルのもっとも明るい部分(ハイライト)の色は、この反射の影響を大きく受けます。

エミッシブ(自己発光)
エミッシブは、モデル自体から光を放射する機能です。物体自体が蛍光塗料を塗られたように光るような効果が得られます。光で照らさなくてもモデルに色をつけられます。

課　　題

背景とマテリアルを変更し、夕方のような雰囲気にしてみましょう。

1．夕方用の背景画像"RedSky.jpg"が読み込まれ、描画されるようにプログラムを変更してください。

2．モデルのマテリアルを変更します。以下のプログラムをGameMain::LoadContentの「モデル読み込み」以降の適切な場所に追加しましょう。

```
// マテリアル設定
Material    mtrl(town);
mtrl.Emissive = Color(1.0f, 0.0f, 0.0f);     // エミッシブ設定
town->SetMaterial(mtrl);
```

エミッシブは、光源や面の向きを考慮せず色を付けます。モデル全体のベースとなる色の担当です。

3．Color()の数を変更し、よりリアルな雰囲気になるようにしましょう。

ヒント１：Colorには、左から赤、緑、青の順で色を指定します。
ヒント２：0.0から1.0の範囲で指定します。
ヒント３：0.0は色なし、1.0は最も明るくなります。
ヒント４：光の三原色なので、すべてが0.0は黒、すべてが1.0は白になります

フォグ

・フォグは、モデルと視点の距離に応じて霧がかかったようにぼやけさせる処理
・モデルが霧で包まれたような効果が得られる
・距離によって徐々に現れたり、消えるような表現ができる
・ただし、フォグ自体は見えない

概要

フォグとは、ポリゴンを視点との距離に応じて霧がかかったようにぼやけて見えるようにする効果のことです。フォグをかけることで、遠くのオブジェクトを霧に包まれたようにしたり、近づくと徐々に現れるような効果を得ることができます。

フォグは、以下のいずれかの方法で計算させることができます。

(線形公式)

線形公式は、以下の式で計算されます。始点と終点の間で距離に比例してフォグが強くなります。

$$\text{フォグの強さ} = \frac{end - d}{end - start}$$

start = フォグ効果が始まる距離(開始点)
end = フォグ効果がこれ以上増加しなくなる距離(終了点)
d = 視点からオブジェクトまでの距離

線形のほか指数的にフォグを強くする方法があり、それぞれ以下の式で計算されます。

(指数公式)

$$\text{フォグの強さ} = \frac{1}{e^{(d \times density)}}$$

(指数公式(２乗))

$$\text{フォグの強さ} = \frac{1}{e^{(d \times density)^2}}$$

e = 自然対数(約2.71828)
d = 視点からオブジェクトまでの距離
density = フォグ密度(0.0～1.0の任意の値)

課　題

フォグを設定してみましょう。

１．フォグの設定を行う以下のプログラムをGameMain::Drawの適切な場所追加しましょう。

```
// フォグ設定
GraphicsDevice.SetRenderState(FogEnable, TRUE);                    // フォグを有効にする
GraphicsDevice.SetRenderState(FogVertexMode, FogMode_Linear);      // 線形フォグ
GraphicsDevice.SetRenderState(FogColor, Color(1.0f, 0.0f, 0.0f));  // フォグの色
GraphicsDevice.SetFloatRenderState(FogStart,  1.0f);               // フォグ開始点
GraphicsDevice.SetFloatRenderState(FogEnd,  250.0f);               // フォグ終了点
```

２．下線部 Color()の数を変更し、フォグの色を仮想世界の雰囲気にあったものにしましょう。

３．フォグの開始点や終了点も変更してみましょう。

カメラワーク

・カメラの動きを適切に制御することにより、シーンをより効果的に見せられる
・現実世界のカメラワークの技術が応用できる

概要

カメラの基本的な操作は、右の図のように６つあります。これらの操作を的確に用いると、同じシーンでもよりよく見せることができます。現実世界の写真や映画などのカメラワークのテクニックが応用できるようになっています。

ただし、実際の撮影と違い、カメラは概念的な存在のため、その姿を見ることはできません。よって、どの位置にあり、どの方向を向いているのか、といったことを直接見ることはできません。また、カメラの設定を間違えると、何も描画されなくなることがあります。

ドリー	カメラの前後移動
トラック	カメラの左右移動
クレーン	カメラの上下移動
パン	カメラの左右の振り
チルト	カメラの上下の振り
ズーム	レンズ操作による画角変更

課　　題

カメラをキーボード入力によって動かし、仮想世界を自由に動き回ってみましょう。

１．キーボードの「右」が押されたらカメラが右に回転するようにしてみます。以下のプログラムをGameMain::Updateに追加しましょう。

```
int GameMain::Update()
{
    // TODO: Add your update logic here
    // キーボード入力を取得
    KeyboardState    keyboard = Keyboard->GetState();

    // カメラ操作－パン
    if(keyboard.IsKeyDown(Keys_Right)) {
        Camera->Rotation(0.0f, -1.0f, 0.0f);
    }

    // カメラ更新
    Camera->Update();

    return 0;
}
```

２．上記のプログラムを参考に、キーボードの「左」が押されたらカメラが左に回転するプログラムを作成しましょう。

ヒント１：カーソルキーの←はKeys_Left、→はKeys_Rightになります。
ヒント２：Camera->Rotation()の数の並びは、回転させる角度で、ｘ，ｙ，ｚの順です。それぞれの軸を中心軸として回転します。

３．キーボードの「上」が押されたらカメラが前方向に進むようにしてみます。以下のプログラムをGameMain::Updateの適切な場所に追加しましょう。

```
    // カメラ操作－ドリー
    if(keyboard.IsKeyDown(Keys_Up)) {
        Camera->Move(0.0f, 0.0f, 0.25f);
    }
```

４．キーボードの「下」が押されたらカメラが後方向に進むプログラムを作成しましょう。

ヒント１：カーソルキーの↑はKeys_Up、→はKeys_Downになります。
ヒント２：Camera->Move()の数の並びは、進ませる量で、ｘ，ｙ，ｚの順です。左手系の軸の向きに進みます。

５．左右、上下に移動とｘ軸、ｚ軸を中心とする回転もできるプログラムを作成してみましょう。