# ESライブラリ＆＆
# ゲームプログラミング
## ○　３Ｄ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

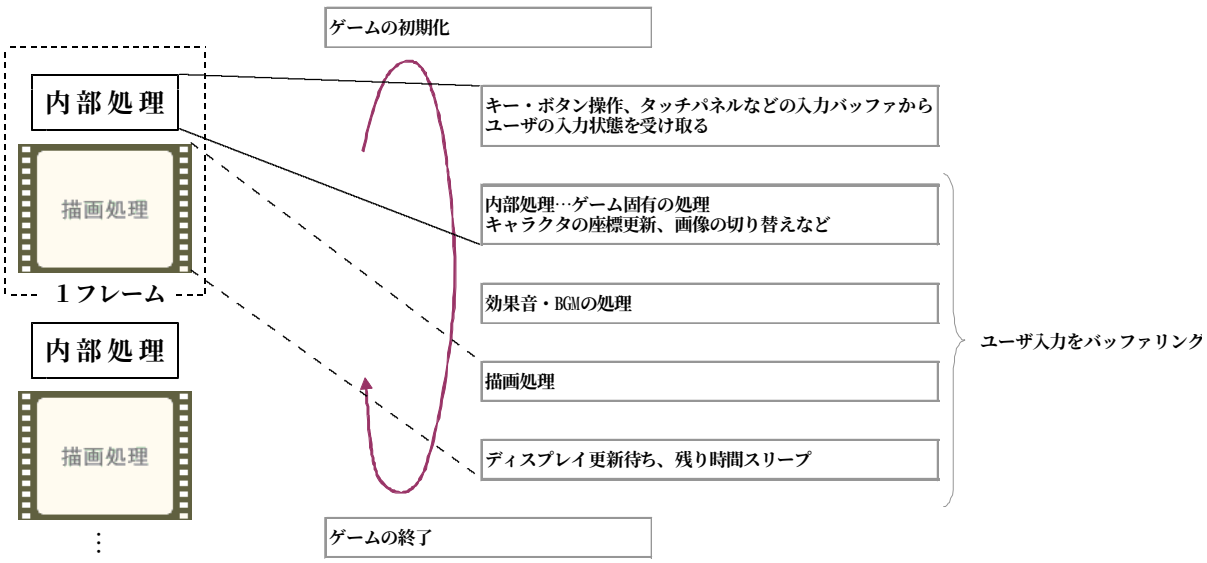
### 概要

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 ＝ CGameMain::Update)
・「もの」を画面に描画(描画処理 ＝ CGameMain::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ループになります。ループ１回ぶんを「１フレーム」と呼びます。１フレームごとに、キャラクターの位置や状態、背景やそのほかの表示状態などを更新し、その状態を反映して描画を行う、という処理を繰り返しています。

### 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関数

ゲームの状態変更や背景、キャラクターの座標更新、衝突判定など描画以外のゲーム処理を記述します。デフォルトでは16.6ms毎、秒間60回呼ばれるようになっています。

### Draw関数

画面に描画するための処理を記述します。

## プリミティブ

・プリミティブは、3Dオブジェクトを構成するための基本的な図形のこと
・プリミティブには、「点」「線」「三角形」がある
・複雑なモデルもプリミティブが組み合わされているが、主に三角形が使われている

### 概要

プリミティブとは、3Dオブジェクト(立体モデル)を構成するための基本的な図形のことです。3Dオブジェクトは、まずたくさんの点＝頂点を定義します。いくつかの頂点をつなげることにより、プリミティブが構成され、さらにプリミティブを組み合わせることにより、3Dオブジェクトが形成されます。

### プリミティブの描画

プリミティブの描画には、頂点の情報が必要です。頂点の「座標」や「色」といった情報を設定すれば、GraphicsDevice.DrawUserPrimitivesという機能を使って描画することができます。

# 課　題

プリミティブを画面に描画してみましょう。

１．頂点の情報を設定します。以下のプログラムをCGameMain::Draw関数に追加しましょう。

```
// 頂点設定(３個の頂点を格納できます)
ScreenVertex v[3];

// 頂点０
v[0].x     = 310.0f;                    // ｘ座標
v[0].y     =   3.0f;                    // ｙ座標
v[0].color = Color(1.0f, 0.0f, 0.0f);   // 色(左から赤、緑、青。0.0から1.0の範囲)
```

２．プリミティブを描画する準備ができたので、描画してみましょう。

以下のプログラムを１．の下に追加しましょう。

```
// プリミティブ描画
GraphicsDevice.DrawUserPrimitives(PrimitiveType_PointList, v, 1, v[0].FVF());
```

ウィンドウの中央やや左、上の方に赤い小さな点が１つ描画されます。

３．点の大きさを変えることもできます。以下のプログラムを２．の前に追加してみましょう。

```
GraphicsDevice.SetFloatRenderState(PointSize, 3.0f);
```

４．頂点を１つ増やしてみましょう。以下のプログラムを１．のあとに追加しましょう。

```
// 頂点１
v[1].x     = 87.0f;                     // ｘ座標
v[1].y     = 67.0f;                     // ｙ座標
v[1].color = Color(0.0f, 1.0f, 0.0f);   // 色
```

５．描画をするときに、２つ描画するようにします。２．のプログラムを以下のように変更しましょう。

```
// プリミティブ描画(点を２つ描画)
GraphicsDevice.DrawUserPrimitives(PrimitiveType_PointList, v, 2, v[0].FVF());
```

６．２つの点を結び、線として描画することができます。２．のプログラムを以下のように変更しましょう。

```
// プリミティブ描画(線を１つ描画)
GraphicsDevice.DrawUserPrimitives(PrimitiveType_LineList, v, 1, v[0].FVF());
```

## 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、物体表面を三角形のポリゴンに分割してデータ化します。ポリゴンの数を増やせば増やすほど表現が精細になりますが、計算量が増えるため、描画に時間がかかるようになります。

１秒間に処理できるポリゴンの数がビデオ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。PlayStation2は7,500万、PlayStation3やXbox 360では数億以上のポリゴンを１秒間に描画することができます。

## 課　　題

頂点の数を３つに増やし、ポリゴンを描画してみましょう。

１．頂点を３つ指定すると、それらが結ばれて三角形が形成され、ポリゴンとなります。このとき、頂点の並び順が時計回りの場合は表向き、反時計回りの場合は裏向きのポリゴンとして扱われます。

頂点は２つ設定されているので、あと１つ追加すると三角形にできます。頂点２として、ｘ座標5.5、ｙ座標0.5に設定してみます。
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。

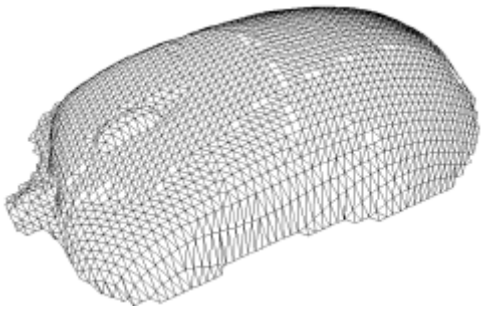
```
// 頂点２
v[2].x     = ※ここを考えてください※;  // ｘ座標
v[2].y     = ※ここを考えてください※;  // ｙ座標
v[2].color = Color(0.0f, 0.0f, 1.0f);   // 色
```

２．頂点を描画するときに、三角形として描画するようにします。描画のプログラムを以下のように変更しましょう。

```
// プリミティブ描画(三角形を１つ描画)
GraphicsDevice.DrawUserPrimitives(PrimitiveType_TriangleList, v, 1, v[0].FVF());
```

## メッシュ

三角形を多数作成すると、網のようになっていきます。これをメッシュと呼びます。3Dグラフィクスでは、立体モデルはメッシュで作られているともいえます。



複雑な形をしていて三角形だけでの表現が難しい場合は、モデルの表面に画像を貼ったり(テクスチャと呼びます)、画面からの距離によって三角形をさらに分割して細かくし、一部を持ち上げるなどを行って詳細な表現ができるようにしています。

## 課　　題

メッシュを表示してみましょう。

1．簡単な形状のメッシュであれば、プログラムで生成して表示することができます。まず、生成したモデルを保存する場所が必要なので、確保します。
　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    model;
```

2．ティーポットを生成させてみます。以下のプログラムをCGameMain::LoadContentに追加しましょう。

```
// ティーポットの生成
SimpleShape    shape;
shape.Type = Shape_Teapot;
model = GraphicsDevice.CreateModelFromSimpleShape(shape);
```

　以上でティーポットの頂点データが作成され、「model」に保存されます。

3．2．で作成した「model」を描画しましょう。

　描画はさきほどの頂点と同じように、CGameMain::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
model->Draw();
```

4．ポリゴンを塗りつぶしてみましょう。

　三角形が多数表示され、網のようになっています。これらの面を塗りつぶし、より立体に近づけることができます。
　以下のプログラムをCGameMain::Drawの適切な場所追加しましょう。

```
// 塗りつぶしモードに設定
GraphicsDevice.SetRenderState(FillMode, FillMode_Solid);
```

## 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
　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
　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

モデルにマテリアルを設定しましょう。

１．アンビエントを設定してみます。以下のプログラムをCGameMain::LoadContentの適切な場所な追加しましょう。

```
// マテリアル設定
Material    mat;
mat.Ambient = Color(0.3f, 0.1f, 0.1f);  // アンビエント
model->SetMaterial(mat);
```

　アンビエントは、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。

２．ディフューズを追加してみます。以下のプログラムを適切な場所な追加しましょう。

```
mat.Diffuse = Color(0.7f, 0.0f, 0.0f);  // ディフューズ
```

　ディフューズは、面の向きによって色に強弱が付きます。モデルの陰となる色を担当しています。

３．Color()の数を変更し、モデルの色を変更してみましょう。

　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

### モデルデータ

　複雑な形状をしているモデルでも、三角形が組み合わされて作られています。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、画像としてではなく頂点情報の集まりとして保存しています。

　モデルの保存形式の１つであるｘファイルでは、下記のように頂点情報を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

モデルを読み込んで描画してみましょう。

１．以下のプログラムをCGameMain::LoadContentの「// マテリアル設定」の下に追加しましょう。

```
// モデル読み込み
model = GraphicsDevice.CreateModelFromX( TEXT("ｘファイルのファイル名") );
```

この１行で、指定したｘファイルからモデルデータが読み込まれます。ティーポットに代わり、読み込んだモデルデータを制御することができるようになります。

## ワールド変換

・3Dグラフィクスは、モデルを3D空間に配置し、仮想世界を構築する
・「3D空間にモデルの配置を行う」には、3D空間の座標＝ワールド座標を指定する
・ワールド座標を直に指定して表示させることはできず、変換行列をとおさなければならない
・頂点座標の変換には４行４列の行列が使われる

### ワールド変換行列

3Dグラフィクスでは、複数のモデルを3D空間に配置し仮想世界を構築します。これは、それぞれのモデルを共通の3D空間の座標(ワールド座標)に配置することになります。

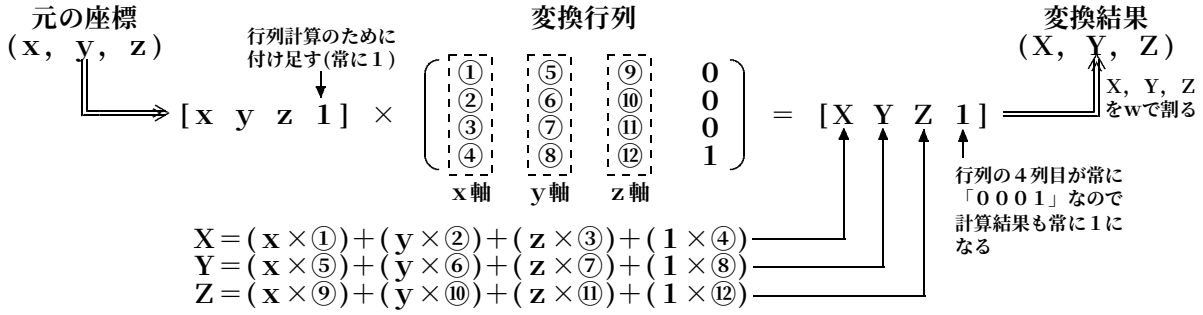
モデルをワールド座標に配置する場合、ワールド座標の値を直接指定することはできず、どこに、どのように配置するかを格納した「行列」を指定し、座標変換をさせて配置します。

座標変換は、変換行列によって行われます。モデルの頂点をワールド座標に変換する行列のことをワールド変換行列(ワールドトランスフォーム行列)と呼びます。

座標変換では、頂点座標はベクトルとして扱い、変換行列が掛けられ、新しい点に変換されます。3Dで座標変換に使用する行列は、同次座標を用いるため４×４の行列でなければなりません。3Dベクトルと４×４の行列を掛けることはできないので、3Dベクトルに４つ目の成分ｗを１としたものを加え、4Dベクトルに変換してから処理を行います。

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

◀座標変換で使用される４×４の行列




$$X = (x \times ①) + (y \times ②) + (z \times ③) + (1 \times ④)$$
$$Y = (x \times ⑤) + (y \times ⑥) + (z \times ⑦) + (1 \times ⑧)$$
$$Z = (x \times ⑨) + (y \times ⑩) + (z \times ⑪) + (1 \times ⑫)$$

3Dの開発環境の整備が進み、ベクトルや行列の知識がなくてもモデルの制御を簡単に行えるようになっています。プログラムからは、ワールド座標や回転角を指定するだけで、その位置までモデルを移動させることができますが、内部では頂点座標を4Dベクトルに変換し、さらに４行４列の変換行列を作って掛け合わせ、3Dの仮想世界に配置しています。頂点の分だけこのような処理が必要なので、１秒間に何十億という計算が行われていることになります。座標変換しても、このままでは単なる点なので、さらに線で結んで三角形にし、囲まれた領域を塗りつぶして色づけを行い、画面に表示していることになります。

# 課　　題

モデルを3D空間内で移動させてみましょう。

１．モデルの座標を保存する領域(変数といいます)を確保します。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   model;
float   x, y, z;     // モデルのx, y, z座標
```

２．変数ｘ，ｙ，ｚに最初の数を設定します。以下のプログラムをCGameMain::Initializeの一番下に追加しましょう。

```
// モデル初期座標
x = 0.0f;
y = 0.0f;
z = 0.0f;
```

変数ｘ，ｙ，ｚには、数値を代入したり、四則演算などの計算をして数値を増減することができます。

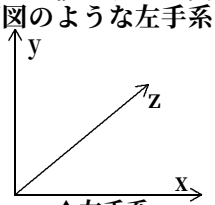
３．モデルの描画の前に、ｘ，ｙ，ｚを反映させます。以下のプログラムをCGameMain::Drawのモデル描画の前に追加しましょう。

```
// モデル描画
model->SetPosition(x, y, z);     // ワールド座標の設定
model->Draw();
```

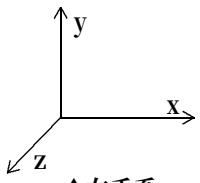
SetPositionでmodelにワールド座標を与えることができます。ワールド座標を設定しておくと、描画時に指定座標へのワールド変換行列を作成して適用するようになっています。

４．２．の初期座標を変更してみましょう。

ヒント１：大きな値にすると画面の外に飛び出してしまいます。
ヒント２：下図のような左手系が用いられています。



▲左手系



▲右手系

５．キーボードで動くようにしてみます。以下のプログラムをCGameMain::Updateに追加しましょう。

```
int CGameMain::Update()
{
    // TODO: Add your update logic here
    // キーボード入力を取得
    KeyboardState   key = Keyboard->GetState();
    if(key.IsKeyDown(Keys_Up))
        z += 0.02f;
    if(key.IsKeyDown(Keys_Down))
        z -= 0.02f;

    return 0;
}
```

このプログラムにより、カーソルキーの上下で画面の奥行き方向に移動できるようになります。

６．上記を参考に、カーソルキーの左右でモデルが左右に移動するプログラムを作成しましょう。

ヒント１：←はKeys_Left、→はKeys_Rightになります。
ヒント２：ｘ座標は'x'という変数が管理しています。

７．'A'キーを押したら上に、'Z'キーを押したら下にモデルが移動するプログラムを作成しましょう。

ヒント１：'A'キーはKeys_A、'Z'キーはKeys_Zになります。
ヒント２：ｙ座標は'y'という変数が管理しています。

### モデルの回転・拡大縮小

ワールド変換では、移動のほかに回転と拡大縮小も行えます。本来は、ｘ、ｙ、ｚ軸それぞれの回転行列、拡大縮小のための行列を作成し、合成する必要があります。PCやXbox360では、数値を指定するだけで目的の行列を作成する機能があるため、簡単にモデルの制御を行うことができます。

## 課　　題

モデルの回転と拡大縮小をしてみましょう。

１．モデルの回転は、SetRotationで行えます。以下のプログラムをmodel->Draw()の前に挿入し、実行結果を確認しましょう。

```
model->SetRotation(0.0f, 90.0f, 0.0f);
```

２．SetRotationのカッコ内の数値を変更し、モデルがどのように回転されるか確認しましょう。
ヒント：カッコ内の数値は回転角で、左からｘ，ｙ，ｚの順に並んでいます

３．モデルの拡大縮小は、SetScaleで行えます。以下のプログラムをmodel->Draw()の前に挿入し、実行結果を確認しましょう。

```
model->SetScale(0.5f, 0.5f, 0.5f);
```

４．SetScaleのカッコ内の数値を変更し、モデルがどのように拡大されるか確認しましょう。
ヒント：カッコ内の数値は拡大率で、左からｘ，ｙ，ｚの順に並んでいます

## 影

・影(Shadow)は、物体が光を遮ることでできる
・陰(Shade)は、ライトで照らして描画すればできるが、影はできない(光自体も描画されない)
・リアルタイムに生成する影は負荷が高く完全ではないため、さまざまな方法が考案されている
・影は、臨場感が増すだけでなく遮蔽物の存在を確認できるなど、空間把握の手助けにもなる

### 概要

影は、物体が光を遮ることでできます。3Dグラフィクスでは、光は物体を照らして陰をつけることはしますが、光そのものは描画されません。影も同様に、描画されることはありません。

説得力の高い、臨場感あふれる映像を再現するには、影は欠かすことのできない要素の１つです。しかし、完全な影をリアルタイムに生成し描画する方法は、まだ考案されていません。また、リアルな影を再現しようとすると、負荷が高くなってしまいます。

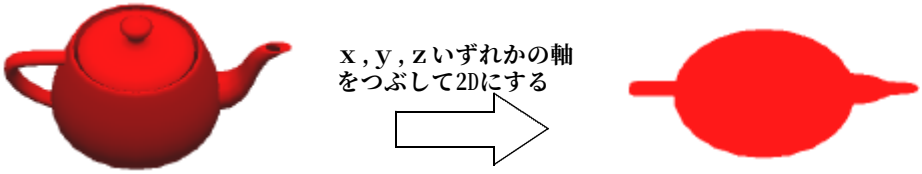
影を生成する主な方法には、まる影、平面投影、投影テクスチャマッピング、シャドウボリューム、シャドウマップなどがあります。

## まる影

丸い影の画像をモデルの足下に描く方法です。負荷が非常に低く、位置関係を立体的に伝えることができます。まる影も発展しており、まる影１つを描画するものから、モデルの姿勢によって伸縮するもの、モデルのパーツごとにまる影を連結して描画するもの、光源の方向を考慮するものもあります。
あらゆるモデルの陰が"まる"をもとに描かれるため、リアリティはほとんどありませんし、セルフシャドウ(自分自身への影付け)もできませんが、負荷の低さから現在でも使われることがあります。また、リアルな影よりもシンボル的な影が良い場合もあります。

## 平面投影シャドウ

モデルを平面に投影したものを影として描画する方法です。基本的な考え方は、「影は立体を平面にしたもの」というものです。3Dから次元をひとつ落として2Dにし、それを影とする方法です。




## 課　　題

平面投影シャドウでモデルに影をつけましょう。

１．影用モデルを保存しておく領域を宣言します。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。

```
// 影用モデル
MODEL    shadow;
```

２．影用のモデルを読み込みます。今回は、描画用モデルをそのまま流用します。以下のプログラムをCGameMain::LoadContentの「// モデル読み込み」以降に追加しましょう。

```
// 影用モデルの読み込み
shadow = GraphicsDevice.CreateModelFromX( TEXT("ｘファイルのファイル名") );
```

３．「光が上から照らしている」という設定にします。ｙ軸の拡大率を０にして描画すれば、モデルの高さがなくなり、「幅(ｘ軸)」と「奥行き(ｚ軸)」しかなくなるので立体は平面となって描画されます。以下のプログラムを２．のあとに追加しましょう。

```
shadow->SetScale(0.5f, 0.0f, 0.5f);
```

４．影を描画します。拡大率の設定はされているので、座標を指定して描画するだけです。以下のプログラムをCGameMain::Drawの「// モデル描画」のあとに追加し、プログラムを実行してみましょう。

```
// 影描画
shadow->SetPosition(0.0f, -3.0f, 0.0f);
shadow->Draw();
```

５．モデルと影が一緒に動くようにしましょう。