

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

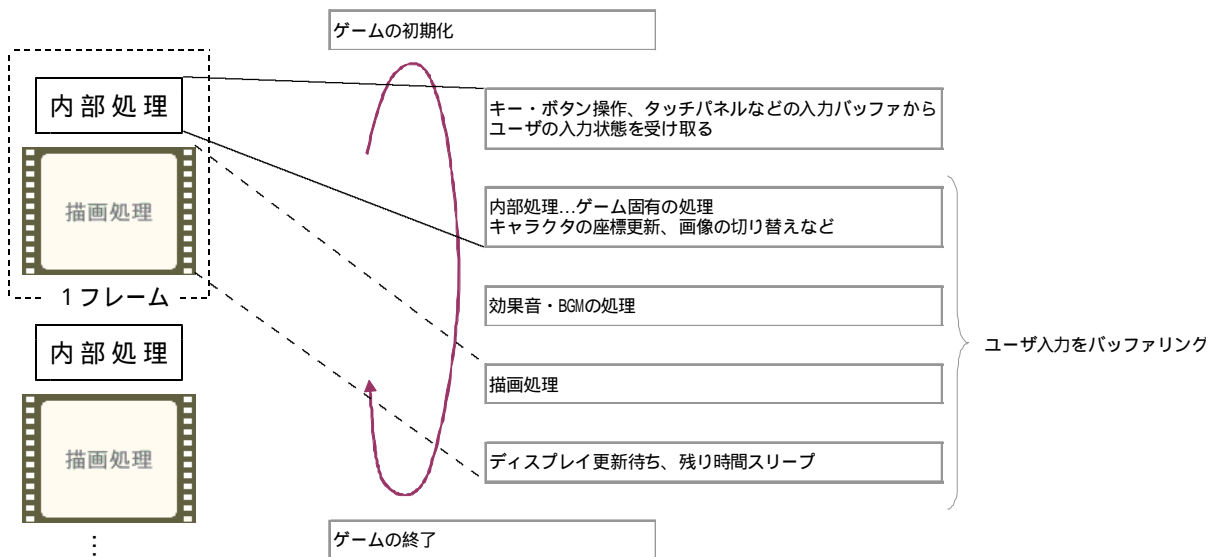
概要

ゲームプログラムは、キャラクターや背景など、ゲーム世界に存在するさまざまな「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。これを非常に単純にすると、

- ・「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つに分けることができます。この処理の繰り返しがゲームのメインループ(主処理)になります。ループを1回実行すると、ゲームが1コマ分進むことから「1フレーム」と呼びます。1フレーム内の処理では、1フレーム分ゲームを動かすための処理を行い、その結果を画面に表示するようにプログラムします。これを繰り返すことにより、ゲームが進んでいきます。

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 1秒間にゲームループが何回実行されるか
- ・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/UnloadContent | 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

ゲームを進めるための処理を記述します。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回実行されるようになっています。

Draw関数

ゲームの画面(Update関数の実行結果 = ゲームを進めた結果)を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で、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。

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、線に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。用語の意味からは、何角形でもかまいませんが、扱いやすさから、ほとんどの場合、三角形を基本単位としています。

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グラフィックボードやゲーム機の性能を表す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(ただし、1秒間に30または60フレーム必要なため、1フレームあたりのポリゴン数は割った数になります)。

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。

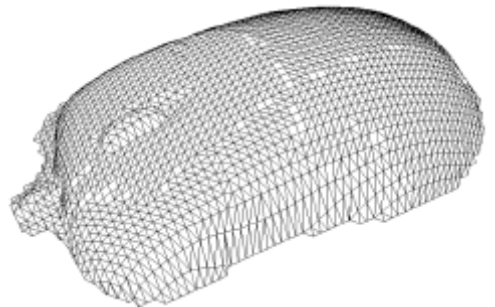
```
// プリミティブ描画(三角形を1つ描画)
GraphicsDevice.DrawUserPrimitives(PrimitiveType_TriangleList, v, 1, v[0].FVF());
```

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。

詳細なメッシュを作成するには、たくさんの頂点が必要になりますが、1つ1つ手作業で座標や色を指定するのは大変困難です。そこで、モデリングソフトと呼ばれる3Dオブジェクトを作成するツールを使って作成します。モデリングソフトを使うと、ビジュアル的にメッシュを作成し、頂点データを出力することができます。また、DirectXには、3Dグラフィクスのテストによく用いられる基本的な形状(ボックス、球、円柱、トーラス、ティーポット)のメッシュを生成する機能があります。



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、生成したメッシュ(モデルデータ)を保存する場所が必要なので、宣言して確保します。
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GameMain::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」という領域に保存されます。

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、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
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GameMain::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
- ・3Dグラフィックスのモデルの色は、光で照らし、反射することによって決められる
- ・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) = 陰の色

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) = ベース色

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) = ハイライト色

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(放射) = ベース色

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の適切な場所に追加しましょう。

```
// 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を担当しています。

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に追加しましょう。

```
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グラフィックスでは、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。

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GameMain::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。

3Dモデル描画の仕組み(ワールド変換)

- ・3Dグラフィックスは、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

3Dモデルをワールド空間に配置するための行列

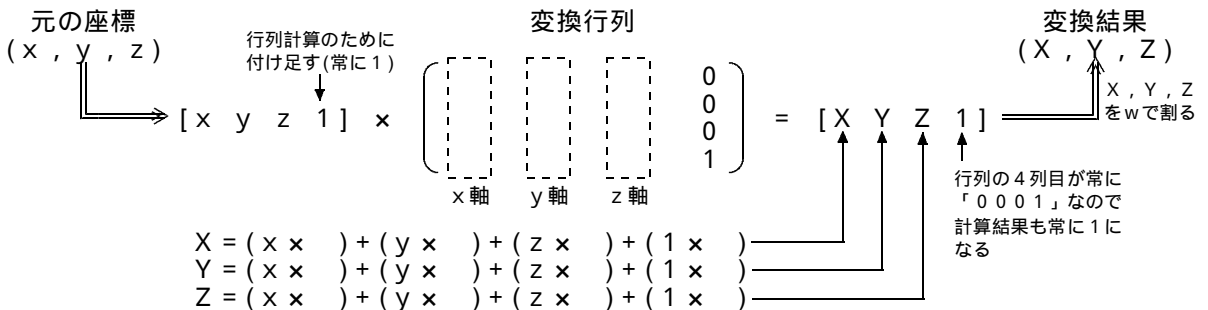
3Dグラフィックスでは、複数のモデルを3D空間に配置し仮想世界を構築します。これは、それぞれのモデルを共通の3D空間の座標(ワールド座標)に配置することになります。

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。

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{座標変換で使用する } 4 \times 4 \text{ の行列}$$



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きます。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。

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GameMain::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GameMain::Drawのモデル描画の前に追加しましょう。

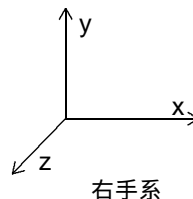
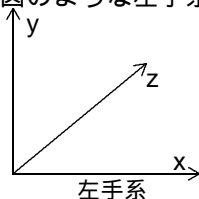
```
// モデル描画
model->SetPosition(x, y, z);    // ワールド座標の設定
model->Draw();
```

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、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うです。

ヒント2: 下図のような左手系が用いられています。



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

複数のモデルの表示

MODEL 1 つにつき 1 つモデルを制御することができます。複数のモデルを制御する場合、その分だけ MODEL が必要になります。MODEL は、それぞれの領域が独立しているため、あるモデルの座標や大きさを変えたとしても、ほかのモデルには影響がありません。

ゲームの世界に必要な分だけ MODEL を宣言し、データを読み込んで適切に表示することにより、仮想世界を構築することができます。

課 題

複数のモデルを表示してみましょう。

1 . MODEL をもう 1 つ宣言し、新しいモデルデータの領域を確保します。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。

```
MODEL    model2;
```

プログラムでは、このようなデータを格納する領域を「変数」と呼び、名前が重複しなければ、容量の許す限りいくつでも宣言することができます。

2. 変数model2にモデルデータを読み込みます。以下のプログラムをGameMain::LoadContentの適切な場所に追加しましょう。

```
// モデル2読み込み
model2 = GraphicsDevice.CreateModelFromX( TEXT("x ファイルのファイル名") );
```

3. 2. で作成した「model2」を描画しましょう。

描画はこれまでと同じように、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2 描画
model2->Draw();
```

4. 3. で描画したモデル2も、モデル(model)と同じように、座標や大きさ、角度を自由に設定できます。以下のプログラムを参考に、変更してみましょう。

```
model2->SetPosition(x座標, y座標, z座標); // 位置の設定
model2->SetScale(x方向の拡大率, y方向の拡大率, z方向の拡大率); // 大きさの設定
model2->SetRotation(x軸の角度, y軸の角度, z軸の角度); // 角度の設定
```

アニメーションモデル

モデルにアニメーションデータが設定されている場合は、アニメーションさせることができます。経過時間に合わせたモーションを計算する機能があるため、なめらかなアニメーションが実現できます。

課 題

アニメーション付きモデルを読み込み、アニメーションさせながら描画してみましょう。

1. アニメーションデータは、MODELでは扱えません(アニメーションデータが削除されてしまいます)。アニメーション付きモデルは、専用のANIMATIONMODELに格納します。
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。

```
ANIMATIONMODEL anime;
```

2. 以下のプログラムをGameMain::LoadContentの適切な場所に追加しましょう。

```
// アニメーションモデル読み込み
anime = GraphicsDevice.CreateAnimationModelFromX( TEXT("x ファイルのファイル名") );
```

3. 2. で作成した「anime」を描画しましょう。

描画はこれまでと同じように、GameMain::DrawでDraw()を実行すると描画できます。このとき、アニメーションの経過時間を指定すると、その時間に合わせたモーションを計算し、頂点データを変更するようになっています。

```
// モデル描画(0.016秒[1/60秒]経過)
anime->Draw(0.016);
```

位置、大きさ、角度はこれまでと同じように変更できます。