Cocoa はやっぱり! 出張版
2002.2
8. OpenGL を使う

■ 今回のテーマ

今回は、いつものサンプルコードの解説を一旦お休みして、**Cocoa アプリケーションから OpenGL を利用する方法**をテーマとして取り上げます。OpenGL というと、3D グラフィックス用という印象が強くて、一般のプログラマーには縁遠い感じもしますが、最近では様々なビジュアルエフェクトのために OpenGL を使用するというケースも増えてきています。

例えば、iPhoto のスライドショーや、Mac OS X に付属の Beach や Cosmos などのスライドショーを行うスクリーンセーバーでは、2 つの画像をクロスフェードする (2 つの画像の片方が消えながら、もう一方の画像が現れてくる) ビジュアルエフェクトが使われています。あれらは OpenGL の機能を使って実現しています。iTunes に標準搭載のビジュアルエフェクトも OpenGL によって実現されています。このように、3D グラフィックと関係のないソフトウェアでも OpenGL が使われるようになってきています。

そこで、本格的な 3D グラフィックの実現のための OpenGL の利用ではなく、**ビジュアルエフェクトのための OpenGL の利用**という観点からの説明を行っていきます。もちろん、3D グラフィックアプリケーションを Cocoa で作成するという方にも、第一歩を踏み出すのには役に立つのではないかと思います。

◎ 推奨環境

この解説は、以下の環境を前提にしていますので、ご確認ください。

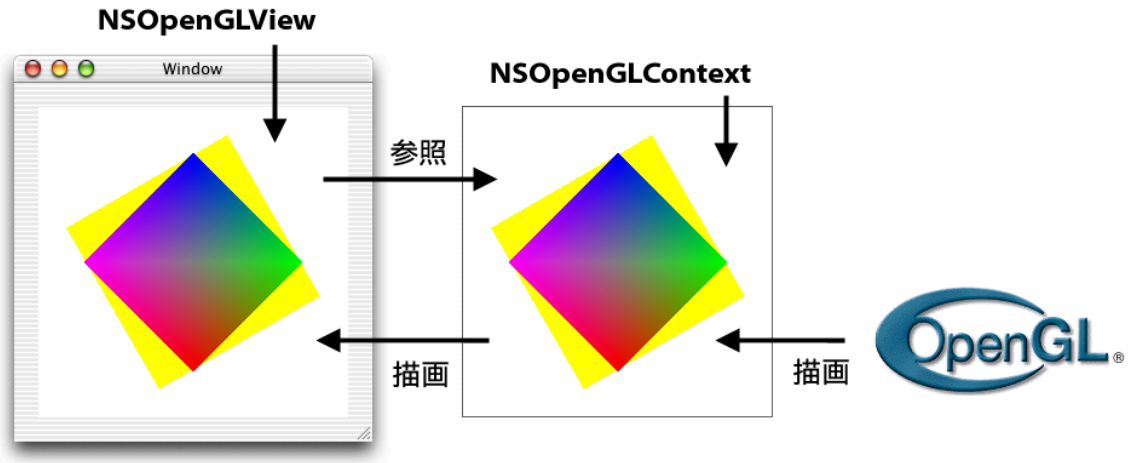
- Mac OS X 10.1.2 以降
- Project Builder 1.1.1 (December 2001 Developer Tools) 以降
- Interface Builder 2.2 (December 2001 Developer Tools) 以降

なお、OpenGL の実行結果は、搭載しているグラフィックスアクセラレーターカードによって変わることがありますのが、ご了承ください。こちらで確認しているのは、PowerBook G4 / 500 と iMac DV です。

■ Cocoa と OpenGL との関係

まずは、Cocoa と OpenGL の関係から見ていきます。Cocoa アプリケーションから OpenGL を使うための主要なクラスは、**`NSOpenGLView`** と **`NSOpenGLContext`** の 2 つです。`NSOpenGLView` は、`NSView` のサブクラスになっていまして、画面へのグラフィックの表示やイベント処理などを行います。`NSOpenGLContext` は、OpenGL のエンジンが使用するフレームバッファや描画の属性等を記憶するためのコンテキストを扱うためのクラスです。`NSOpenGLView` は、1 つの `NSOpenGLContext` を持っています。

そもそも、Cocoa (もしくは Mac OS X) と OpenGL は、全く独立に開発されたものですので、Cocoa と OpenGL はあまり密な関係にはありません。OpenGL には、OpenGL 自身が管理しているコンテキストに描画を行ってもらい、その結果を引き取って、Cocoa の環境に合わせて画面に表示するというようなイメージになります。Cocoa と OpenGL の間の仲介役として `NSOpenGLView` や `NSOpenGLContext` が存在しているわけです。



【図】Cocoa と OpenGL の関係

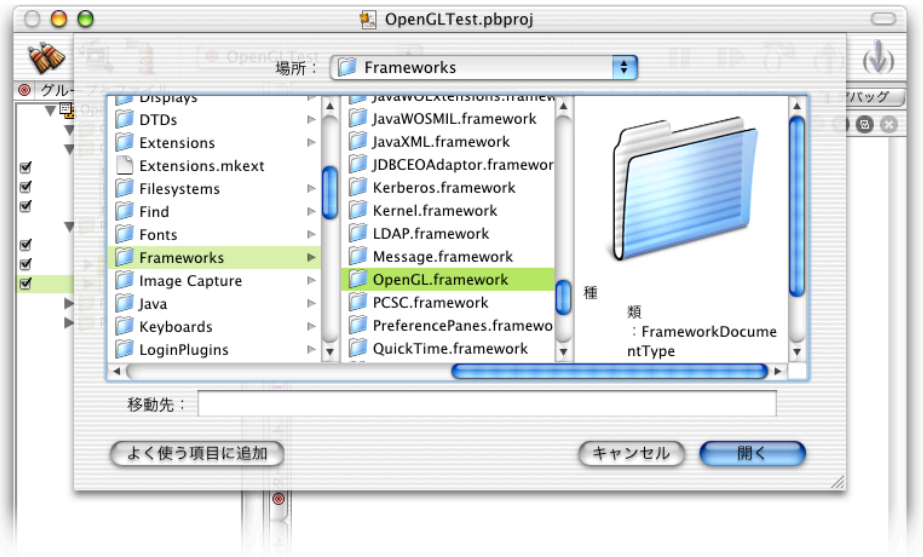
描画を行う場合は、**Cocoa アプリケーションから OpenGL の関数を直接呼び出す**ことになります。そうすると、カレントの OpenGL のコンテキストに対して描画が行われます。OpenGL の関数を呼ぶことになりますので、Cocoa のプログラミングとはいえ OpenGL の知識が必要になります。ただし、この記事の目的は OpenGL の解説ではありませんので、OpenGL については簡単な説明にとどめさせて頂きます。

実のところ、私自身もこの記事を書くために初めて OpenGL を勉強しましたので、まだ使い方がこなれていないところがあるかもしれませんが、間違いのご指摘などありましたらお願いいたします。Web 上で私が OpenGL 入門のために主に参考にしたのは、以下の **OpenGL プログラミングコース**というサイトです。サンプルを交えた丁寧な解説で PDF による 400 ページを超えるドキュメントがありますので、入門にはよいのではないかと思います。こちらと合わせて読んでいただけると、より理解が深まると思います。

OpenGL プログラミングコース
http://www.nk-exa.co.jp/mmtech/OpenGLEdu/

■ 簡単なプログラムを作る

では、簡単なプログラムを作って Cocoa アプリケーションから OpenGL を呼び出してみましょう。まず、Project Builder でプロジェクトを Cocoa Application として作成します。OpenGL の関数を使うためには **OpenGL.framework** というフレームワークが必要ですので、「 プロジェクト → フレームワークを追加... 」メニューでプロジェクトに追加します。



【図】OpenGL.framework の追加

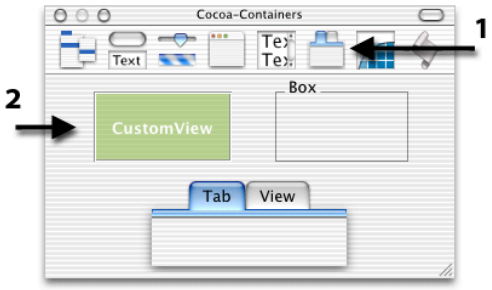
> **ちょっと補足**：今回は使用しませんが、**GLUT**（ OpenGL Utility Toolkit ）という、より簡単に OpenGL を扱うためのフレームワークも用意されています。GLUT を使う場合は、**GLUT.framework** もリンクしてください。ちなみに、GLUT のサンプルは、以下のディレクトリに沢山入っていますので参考になるでしょう。
>
> /Developer/Examples/GLUTExamples/

次に、Resources の中の MainMenu.nib をダブルクリックして Interface Builder を起動します。Cocoa-GraphicsViews の中にある OpenGL のロゴが書かれているものが、`NSOpenGLView` です。これをウィンドウの中にドロップして配置しますと、`NSOpenGLView` が使えるようになります。

【図】Cocoa-GraphicsViews の `NSOpenGLView`

自前のビューを作成する場合には、`NSView` のサブクラスを作成して、配置した Custom View にそのクラスを割り当てて、初期化メソッドの `initWithFrame :` や描画メソッドの `drawRect :` を書きます。自前の OpenGL のビューを作成する場合も、`NSOpenGLView` のサブクラスを作って...という流れは変わりません。

ウィンドウに配置するのは、Cocoa-GraphicsViews の `NSOpenGLView` でもよいですし、今までどおり、Cocoa-Containers の Custom View でも構いませんが、今回は、Custom View を使用します。



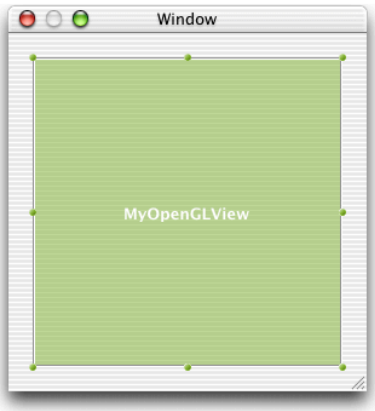
【図】Cocoa-Containers の Custom View

`NSOpenGLView` を使わないのは、**`NSOpenGLView` を配置した場合は、`initWithFrame :` メソッドが呼ばれないためです。`initWithFrame :` は、Custom View を配置した場合のみ呼ばれるようです。**このことについては、Interface Builder の FAQ の「 Why isn't my initWithFrame : method called? (なぜ私の initWithFrame : メソッドは呼ばれないのか?) 」に書かれています。FAQ は、Interface Builder の「 Help → FAQ 」メニューを選択すると表示されます。

いくつかの Apple のサンプルコードを見てみましたが、同様に Custom View が配置されていましたのでこの方法で問題はないと思われます。

> **ちょっと補足**：Apple のサイトのサンプルコードで 3D グラフィックスに関するものは、
> http://developer.apple.com/samplecode/Sample_Code/Graphics_3D.htm
> にあります。この中で、Cocoa での OpenGL 入門によさそうなものは、Simple AppKit と Cocoa InitGL です。

続いて、MainMenu.nib ウィンドウの Classes タブの中で `NSOpenGLView` のサブクラスを作成します。クラス名は `MyOpenGLView` とします。これを、先程の Custom View のクラスに割り当てて、ソースの出力を行います。この辺りの流れは、自前のビューを作成するのと同じです。ビューの形状ですが、今回は、以下のように正方形にしてください。



【図】正方形にビューを配置する

そうしましたらコーディングに入ります。まず、ビューの真ん中に OpenGL を使って黄色い四角形を書いてみましょう。これもいつもと同様に `drawRect :` メソッドに描画処理を書いていきます。

MyOpenGLView.m > drawRect :

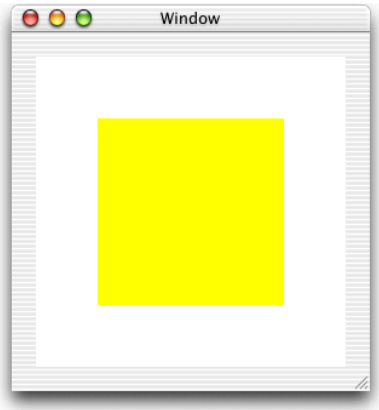
```
- (void) drawRect : (NSRect) rect {

  glClearColor( 1.0, 1.0, 1.0, 1.0 );                    // 背景色指定
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );  // 色とデプスバッファをクリア

  glColor4f( 1.0, 1.0, 0.0, 1.0 );  // 描画色を指定
  glRectf( -0.6, -0.6, 0.6, 0.6 );  // 矩形を描画

  glFinish();                               // 描画コマンド実行
  [ [ self openGLContext ] flushBuffer ]; // 画面更新

}
```

gl で始まっているのが OpenGL の関数です。このように、OpenGL に大きく依存したコードになります。そして、OpenGL.framework のヘッダーを読み込むために、MyOpenGLView.h に以下の記述を足してください。

MyOpenGLView.h

```
#import <OpenGL/OpenGL.h>
#import <OpenGL/gl.h>
#import <OpenGL/glu.h>
```

この 3 つが、必要なヘッダーです。これを実行すると、このようになります。



【図】黄色の四角の描画結果

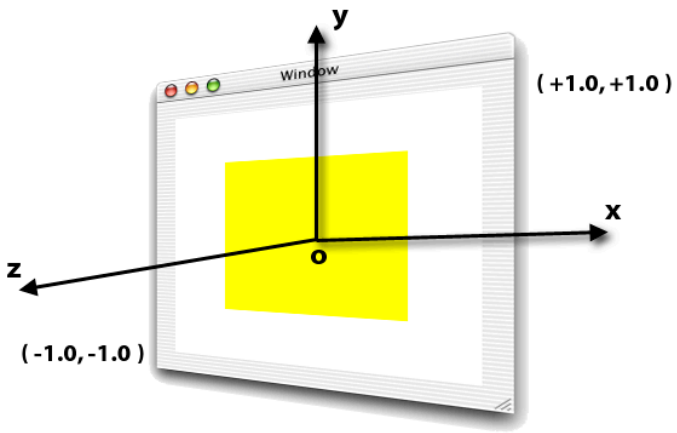
では、コードを簡単に説明します。**glClearColor で背景色を黒に指定**して、**glClear で実際にクリア**しています。glClearColor のパラメータは、4 つありますが、透明度を含んだ RGBA の 4 つを指定できるようになっています。値の範囲は、0.0〜1.0 です。

glClear のパラメータは、どのバッファをクリアするかという情報です。OpenGL は、画面に表示される各ピクセルの色情報を覚えるバッファ以外にもいくつかのバッファを持っています。陰面処理を行うためのデプ

スバッファ（Zバッファともいいます）や、全ピクセルに対する一括演算を行うためのアキュムレーションバッファなどです。今回は、**色情報のバッファ（GL_COLOR_BUFFER_BIT）**と**デプスバッファ（GL_DEPTH_BUFFER_BIT）**をクリアするために、これらの論理和を取ってパラメータとしています。

> **ちょっと補足**：OpenGL は、3D のグラフィックを描画する際の陰面処理をデプスバッファで行っています。デプスバッファとは各ピクセルの視点からの距離を記憶しているバッファです。ピクセルを描画するときには、そのピクセルの視点からの距離をデプスバッファへも書き込んでいきます。次回、同じピクセルへの書き込みが発生したときは、視点からの距離がそれよりも近い場合にのみ書き込むようにすることで陰面処理を行います。デプスバッファをクリアするということは、全ピクセルを最も遠い値で埋めることになります。また、Z バッファと呼ばれるのは、視線方向に Z 軸があるためです。

glColor4f では、描画色を指定します。RGBA の値をパラメータとします。**glRectf では、パラメータで指定された四角形を描画**します。デフォルト状態の座標系は、左下が（-1.0, -1.0）で、右上が（+1.0, +1.0）になっています。OpenGL は、3D グラフィックのエンジンですが、2D の座標指定も出来ます。2D で座標指定すると 3D 空間の中の特定の面に対しての描画になります。ちなみに、Z 軸は、ビューの面に垂直で手前に向かって伸びています。



【図】OpenGL のデフォルトでの座標軸

最後の **glFinish** で、これらの処理を実行させます。

OpenGL にはダブルバッファの機能がついていて（オフにすることも出来ます）、画面に表示されているフロントバッファと、OpenGL が直接描画するバックバッファの 2 つのバッファを持っています。バックバッファに描画しておいて、描画が一段落したところでバックバッファからフロントバッファへコピーする（バックバッファとフロントバッファをスイッチする場合もあるようです）ことで、描画過程を見せなくしたり、描画時のちらつきを抑えたりするわけです。

この、**バックバッファからフロントバッファへのコピーを行うのが、NSOpenGLContext クラスの flushBuffer メソッド**です。NSOpenGLView である self の **openGLContext メソッドで、OpenGL のコンテキストが取得できます**ので、それに対して flushBuffer を実行します。

NSOpenGLView : OpenGLのコンテキストを取得

書式
```
- (NSOpenGLContext *) openGLContext;
```
出力
返り値 : 割り付けられているOpenGLのコンテキストのインスタンス。

NSOpenGLContext : バックバッファを画面に表示する

書式
```
- (void) flushBuffer
```
備考
ダブルバッファを使用しているときのみ有効。

OpenGL の関数の名前は、**gl** で始まっていることはお話しましたが、関数名の末尾の部分にも意味のある文字が付いています。glColor**4f** の末尾の **4f は、パラメータが４つの浮動小数値（ Float ）である**ということ表しています。ここには出てきていませんが、glColor**3i** という関数もあります。こちらは、透明度を指定しない RGB の **３つの整数値（ Integer ）で色を指定**することになります。このように、OpenGL には同じ機能でパラメータだけが違うという関数が沢山あります。そして、関数名からそれらを見分けることが出来ます。

■ グラデーション付きの描画

先程のサンプルはちょっと地味でしたので、もう少し派手なものにしてみましょう。グラデーション付きの四角形を書いてみます。以下のメソッドを新たに書いて、drawRect : メソッド内から [self drawGradRect]; で呼び出してください。

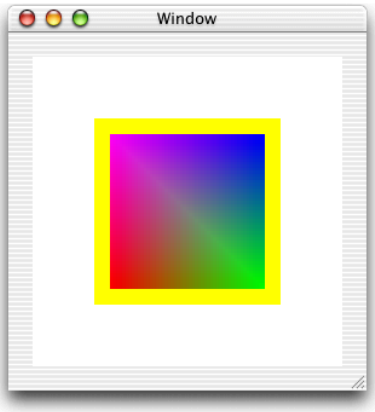
MyOpenGLView.m > drawGradRect

```
- (void) drawGradRect {

  glBegin( GL_QUADS ); { // 四角形列の定義

    glColor4f(   1.0,  0.0,  0.0,  1.0 );  // 赤
    glVertex3f( -0.5, -0.5,  0.0 );        // 左下の頂点

    glColor4f(   0.0,  1.0,  0.0,  1.0 );  // 緑
    glVertex3f(  0.5, -0.5,  0.0 );        // 右下の頂点

    glColor4f(   0.0,  0.0,  1.0,  1.0 );  // 青
    glVertex3f(  0.5,  0.5,  0.0 );        // 右上の頂点

    glColor4f(   1.0,  0.0,  1.0,  1.0 );  // 紫
    glVertex3f( -0.5,  0.5,  0.0 );        // 左上の頂点

  } glEnd();

}
```

glRectf よりももっと凝った四角形を描く場合は、**glBegin(GL_QUADS); 〜 glEnd();** を使用します。この 2 つの間で様々な指定が出来ます。**glVertex3f は、各頂点の座標を指定**しています (Vertex とは頂点のこと)。3f ですので、3D での座標指定になります。glVertex2f を使えば、2D での指定も出来ます。頂点の色指定は、頂点の座標指定の前に行います。glColor4f がそれです。頂点毎に別の色を指定することでグラデーションを作ることが出来ます。

GL_QUAD**S** と複数形になっているのは、複数個の四角形を一気に描けることを意味しています。もし、直方体などの立体を描画したい場合は、glBegin(GL_QUADS); 〜 glEnd(); の間に座標指定を 4 つずつ増やして 6 つの四角形を指定すればよいのです。GL_QUADS 以外にも、**GL_POINTS**、**GL_LINES**、**GL_POLYGON**、**GL_TRIANGLES など**があり、様々な図形を描画することが出来ます。

実行結果は以下のようになります。



【図】グラデーション付きの四角の描画結果

ちょっと補足：四角形の頂点の指定の順番は、このサンプルでは左回りになっています。OpenGL では、左回りに見える側が表（おもて）側になります。立方体のような閉じた物体を描画する場合は、裏面を描画することはあり得ないので、裏面を描画しないことで高速化を行うという描画オプションがあります。

■ 座標変換

Cocoa の Application Kit には、NSAffineTransform という 2 次元の座標変換のクラスがありますが、OpenGL にも 3 次元の座標変換の機能があります。これを使って、平行移動・拡大縮小・回転 (ロール・ヘッド・ピッチ) が簡単にできます。これらを実現するためのメソッドは以下のようになります。ただし、座標変換の関数は実行の順番で結果が変わりますので、汎用のメソッドというわけではありません。

MyOpenGLView.m > transformMoveX : moveY : moveZ : zoomX : zoomY : zoomZ : ...

```
- (void) transformMoveX : (float) mx
                 moveY : (float) my
                 moveZ : (float) mz
                 zoomX : (float) zx
                 zoomY : (float) zy
                 zoomZ : (float) zz
                  roll : (float) roll
                  head : (float) head
                 pitch : (float) pitch {

  glMatrixMode( GL_MODELVIEW );  // 操作する座標系をモデルにする
  glLoadIdentity();              // 座標系をリセットする

  glTranslatef( mx, my, mz );    // 移動
  glScalef( zx, zy, zz );        // 拡大縮小

  glRotatef( roll , 0, 0, 1 );   // ロール
  glRotatef( head , 0, 1, 0 );   // ヘッド
  glRotatef( pitch, 1, 0, 0 );   // ピッチ

}
```

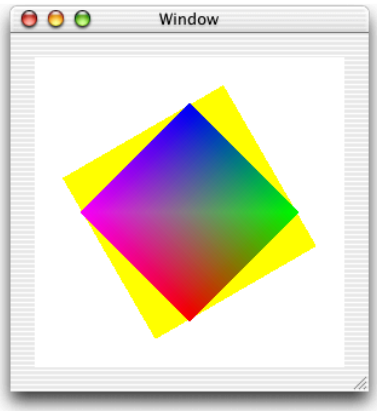
まず、**座標変換を行う対象を glMatrixMode で選択**します。**GL_MODELVIEW で、これから描画するオブジェクトの座標が変換される**ことになります。**glLoadIdentity で、座標変換を行うマトリックスを初期化**します。そして、**glTranslatef で平行移動**、**glScalef で拡大縮小**、**glRotatef で回転**になります。

このメソッドを drawRect : 内から以下のように呼び出します。

MyOpenGLView.m > drawRect :

```
[ self transformMoveX :   0  moveY :  0   moveZ :  0
                zoomX :   1  zoomY :  1   zoomZ :  1
                 roll :  30   head :  0   pitch :  0 ];  // 30度ロール

glColor4f( 1.0, 1.0, 0.0, 1.0 );  // 描画色を指定
glRectf( -0.6, -0.6, 0.6, 0.6 );  // 矩形を描画

[ self transformMoveX :   0  moveY :  0   moveZ :  0
                zoomX :   1  zoomY :  1   zoomZ :  1
                 roll :  45   head :  0   pitch :  0 ];  // 45度ロール

[ self drawGradRect ]; // グラデーションの四角形を描画
```

すると、以下のような結果になります。



【図】座標変換したときの実行結果

ちょっと補足：ロールは「 手を前に伸ばして手首を回転させたときの動き 」、ヘッドは「 首を左右に振ったときの動き 」、ピッチは「 お辞儀をしたときの動き 」に相当します。

■ 初期化について

先程、initWithFrame : メソッドのために Custom View を配置するようにしましたが、実装しなくてもとりあえず OpenGL を呼び出して描画を行うことが出来ました。これは、NSOpenGLView の初期化メソッドがデフォルトの状態を決めて初期化を行ってくれているためです。

ただし、このデフォルトの状態だと必要のないバッファが取られている可能性もありますし、逆に必要なものが取られていない可能性もあります。そのため、自分で初期化を行いたい場合もあるでしょう。その場合は、initWithFrame : を記述していきます。サンプルを以下に書きます。

MyOpenGLView.m > initWithFrame :

```
- (id) initWithFrame : (NSRect) frameRect {

  NSOpenGLPixelFormatAttribute attr[] = {
    NSOpenGLPFADoubleBuffer,      // ダブルバッファを使う
    NSOpenGLPFAAccelerated ,      // ハードウェアアクセラレーションを使う
    NSOpenGLPFAStencilSize , 32,  // ステンシルバッファのビット数を32bitにする
    NSOpenGLPFAColorSize   , 32,  // 画像用バッファのビット数を32bitにする
    NSOpenGLPFADepthSize   , 32,  // デプスバッファのビット数を32bitにする
    0                             // ターミネータ
  };
  NSOpenGLPixelFormat* pFormat;

  pFormat = [ [ [ NSOpenGLPixelFormat alloc ] initWithAttributes : attr ]
                                              autorelease ];
  self    = [ super initWithFrame : frameRect
                      pixelFormat : pFormat ];  // スーパークラスによる初期化

  [ [ self openGLContext ] makeCurrentContext ];  // カレントコンテキストを変更
  glClearColor( 1.0, 1.0, 1.0, 1.0 );             // 背景色を黒にする

  return( self );

}
```

最初で **NSOpenGLFormatAttribute** の配列を定義しています。OpenGL でどういうバッファを使用するかとか、どういう属性を持つかの定数を定義しているのが NSOpenGLFormatAttribute です (enum です)。沢山の指定がありますので、この定数のうち必要なものを配列に詰めて、それをパラメータとしてメソッドに渡すという形になっています。

```
  NSOpenGLPixelFormatAttribute attr[] =  {
    NSOpenGLPFADoubleBuffer,      // ダブルバッファを使う
    NSOpenGLPFAAccelerated ,      // ハードウェアアクセラレーションを使う
    NSOpenGLPFAStencilSize , 32,  // ステンシルバッファのビット数を32bitにする
    NSOpenGLPFAColorSize   , 32,  // 画像用バッファのビット数を32bitにする
    NSOpenGLPFADepthSize   , 32,  // デプスバッファのビット数を32bitにする
    0                             // ターミネータ
  };
```

NSOpenGLPFADoubleBuffer のように定数単独のものと、NSOpenGLPFColorSize のように数値を伴うものがありますが、前者は論理値の指定になります。存在していると YES、存在しない場合は NO という指定になります。配列の最後には、ターミネータとして 0 を入れます。なお、NSOpenGLPixelFormatAttribute の定数の意味を理解するには、OpenGL の知識が必要になりますので、ここでは使用方法のみの説明にとどめます。

この配列を使って、NSOpenGLPixelFormat クラスのインスタンスを以下のように生成します。

```
NSOpenGLPixelFormat* pFormat;

pFormat = [ [ [ NSOpenGLPixelFormat alloc ] initWithAttributes : attr ]
                                            autorelease ];
```

そして、このインスタンスを使って、NSOpenGLView を初期化するという流れになります。

```
self    = [ super initWithFrame : frameRect
                    pixelFormat : pFormat ];  // スーパークラスによる初期化
```

NSOpenGLContext : ピクセルフォーマット指定での初期化

書式
```
- (id) initWithAttributes : (NSOpenGLPixelFormatAttribute *) attribs
```
入力
attribs : ピクセルフォーマット。
出力
返り値 : NSOpenGLContextのインスタンス。

NSOpenGLView : ピクセルフォーマット指定での初期化

書式
```
- (id) initWithFrame : (NSRect) frameRect
         pixelFormat : (NSOpenGLPixelFormat *) format
```
入力
frameRect : ビューのサイズ指定。
format : ピクセルフォーマット。
出力
返り値 : NSOpenGLViewのインスタンス。

NSOpenGLView には、**defaultPixelFormat というデフォルトのピクセルフォーマットを返すメソッド**があります。デフォルトで構わない場合は、これを使うと今までの部分を次の一文に置き換えられます。

```
self = [ super initWithFrame : frameRect
                 pixelFormat : [ NSOpenGLView defaultPixelFormat ] ];
```

NSOpenGLView : デフォルトのピクセルフォーマット

書式
```
+ (NSOpenGLPixelFormat *) defaultPixelFormat
```
出力
返り値 : デフォルトのピクセルフォーマットのインスタンス。

その後、**カレントコンテキストを makeCurrentContext メソッドで自分自身に変更**して、glClearColor

で背景色を設定しています。背景色をずっと変更しないということであれば、drawRect : の中ではなく、初期化時に実行する方が効率的ということで、こちらに移動しました。

```
  [ [ self openGLContext ] makeCurrentContext ];  // カレントコンテキストを設定
  glClearColor( 1.0, 1.0, 1.0, 1.0 );  // 背景色を黒にする
```

Apple のサンプルプログラムの Simple AppKit のコードを見ると、この部分をちょっと特殊な実装をしています。initGL という OpenGL の初期化を行うメソッドを定義して、これを一度だけ drawRect : メソッド内から呼ぶようになっています。

Simple AppKit > GLView.m > initWithFrame :

```
- (id) initWithFrame: (NSRect) frameRect {
    : 省略
  processFunc = @selector( initGL ); // OpenGLの初期化メソッドを記憶しておく

  return self;
}
```

initWithFrame : メソッド内で processFunc に initGL を覚えさせておいて、drawRect : メソッドから実行して、nil に置き換えるという方法です。nil にすることで、一度だけ実行されるようになります。

Simple AppKit > GLView.m > drawRect :

```
- (void) drawRect : (NSRect) rect {

  if ( processFunc ) {
    [ self performSelector : processFunc ]; // initGLを呼ぶ
    processFunc = nil;                      // 一度だけ呼ばれるようにする
  }
    : 省略
}
```

描画メソッドから初期化メソッドを呼ぶというのは、ちょっと変則的な感じがします。このようにしているのは、drawRect : が、このビューに対しての描画環境が整った状態で呼ばれるためだと思われます。例えば、drawRect : が呼ばれたときには、この NSOpenGLView の OpenGL のコンテキストはカレントになっているために、カレントのコンテキストの切り替えを意識しなくてよいというようなメリットがあります。そういった準備を Cocoa のフレームワーク側でやってくれているため安全ということでしょう。

この記事では先程説明した OpenGL のコンテキストを切り替えて初期化するというスタイルを採用しています。この方法で問題ないかの裏はまだ取れていないのですが、問題は今のところ発生していません。もしかすると、[self lockFocus] 〜 [self unlockFocus] で初期化処理を挟むのが正解なのかもしれません。

■ 画像ファイルの表示

続いては、画像の表示の方法です。OpenGL で画像を表示する方法はいくつかありますが、ここでは、テクスチャーとして画像を読み込んで、四角形にテクスチャーをマッピングして表示することにします。この方法を採用すると、マッピングをした四角形を回転すると画像も一緒に回転しますし、四角形の透明度を変えることでフェーディングもできるようになります。

◎ テクスチャーの読み込み

まずは、画像の読み込み部分から。loadImageToTexture : というメソッドを作ります。画像のファイルパスを渡すとテクスチャーを作成して、OpenGL が管理しているテクスチャーの ID を返してくれるというものです。

以下のソースを見ると、前半は、Cocoa のフレームワークを主に使っていて、後半は OpenGL の関数を主に使っていることが分かります。前半では、NSImage に画像を読み込んで、OpenGL へそのビットマップデータを渡すための加工を行っています。特に、**OpenGL のテクスチャーは、ピクセル数が（２の n 乗）×（２の n 乗）の値しか取れない（ n は整数 ）**ということもあって、ここでは、固定的に 512×512 ドットに変形しています。

MyOpenGLView.m > loadImageToTexture :

```
#define TEX_SIZE 512 // テクスチャーのサイズ

- (GLuint) loadImageToTexture : (NSString*) imgPath {

  NSImage*          imgFile;   // ファイルから読み込んだ画像
  NSImage*          imgTex;    // テクスチャー用に変形した画像
  NSBitmapImageRep* imgTexRep; // テクスチャーのビットマップ抽出用
  GLuint            texId;     // テクスチャーID

  // テクスチャーサイズのNSImageを作成 ( Cocoaを主に使用 )

  imgFile = [ [ [ NSImage alloc ] initWithContentsOfFile: imgPath ]
                                  autorelease ];  // ファイルから読み込み
  imgTex  = [ [ [ NSImage alloc ] initWithSize :
                 NSMakeSize( TEX_SIZE, TEX_SIZE ) ] autorelease ];
  [ imgTex lockFocus ];
  [ imgFile drawInRect : NSMakeRect( 0, 0, TEX_SIZE, TEX_SIZE )
              fromRect : NSZeroRect
             operation : NSCompositeSourceOver
              fraction : 1.0 ]; // imgFileをimgTex内にテクスチャーサイズで描画
  [ imgTex unlockFocus ];

  imgTexRep = [ [ NSBitmapImageRep alloc ] initWithData :
                  [ imgTex TIFFRepresentation ] ];
```

```
  // テクスチャーのセットアップ ( OpenGLを主に使用 )

  glPixelStorei( GL_UNPACK_ALIGNMENT, 1 );  // テクスチャーのフォーマットを指定
  glGenTextures( 1, &texId );               // 空のテクスチャーを作成
  glBindTexture( GL_TEXTURE_2D, texId );    // 対象のテクスチャーを指定
  glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA,
                TEX_SIZE, TEX_SIZE,         // テクスチャーのサイズ (幅、高さ)
                0,                          // テクスチャーの枠の太さ
                [ imgTexRep hasAlpha ] ? GL_RGBA : GL_RGB,
                GL_UNSIGNED_BYTE,
                [ imgTexRep bitmapData ] ); // テクスチャーのビットマップを作成
  glBindTexture( GL_TEXTURE_2D, 0 );        // 対象のテクスチャーをなしに指定

  return( texId );

}
```

initWithContentsOfFile : メソッドを使って imgFile という NSImage のインスタンスに画像ファイルを読み込んだ後、512×512 ドットの NSImage のインスタンス imgTex を生成して、そこに imgFile を変形して描画しています。NSImage の中に何かを描画するには、描画先を lockFocus してから描画系のメソッドを実行します。描画が終わったら unlockFocus します。

NSImage : 描画準備処理を行う

書式
```
- (void) lockFocus
```
備考
描画前に実行すること

NSImage : 描画完了処理を行う

書式
```
- (void) unlockFocus
```
備考
lockFocusとペアで描画後に実行すること

NSImage を描画するには、drawInRect : fromRect : operation : fraction : を使っています。このメソッドの第 1 パラメータで描画先の矩形を指定できますので、これで変形が出来ます。第 2 パラメータの fromRect : のパラメータは、画像の部分切り出しの指定ですが、NSZeroRect を指定することで、画像全体の指定になります。

NSImage : 指定位置に指定透明度、指定合成方法で画像を表示する

書式

```
- (void) drawInRect : (NSRect) inRect
            fromRect : (NSRect) fromRect
           operation : (NSCompositingOperation) op
            fraction : (float ) delta
```

入力

```
inRect   : 表示位置
fromRect : 画像の切り出し範囲 ( NSZeroRectで全体を指定 )
op       : 合成方法
delta    : 透明度
```

その後、texImgRep という NSBitmapImageRep クラスのインスタンスを生成していますが、これは、この NSBitmapImageRep を経由することで、圧縮などが行われていない**生のビットマップデータを bitmapData メソッド取り出す**ことが出来るためです。テクスチャー生成のためには、OpenGL に生のビットマップデータを渡す必要があるのです。

glPixelStorei でメモリへの格納形式を指定し、**glGenTextures で空のテクスチャーを生成**します。ここでテクスチャーの ID が返ってきます。**glBindTexture はテクスチャー操作関数の対象を変更する関数**なので、生成したテクスチャーに変更します。そして、**glTexImage2D でテクスチャーのビットマップイメージを送り込みます**。

glTexImage2D のパラメータのところに [imgTexRep **hasAlpha**] ? GL_RGBA : GL_RGB という記述があります。TIFF フォーマットのようにアルファチャンネル (透明度情報) を画像に含んでいる場合がありますので、その場合のために、データの並びをこのような記述で切り替えています。

NSImageRep : アルファチャンネルを持っているか

書式

```
- (BOOL) hasAlpha
```

出力

返り値 : アルファチャンネルを持っている場合 — YES、ない場合 — NO

最後のパラメータで、生のビットマップデータを渡しますが、これは、NSBitmapImageRep の **bitmapData** メソッドを使います。

NSBitmapImageRep : ビットマップの生データを取得

書式

```
- (unsigned char *) bitmapData
```

出力

返り値 : ビットマップデータの生データの存在する先頭のアドレスを返す

なお、アルファチャンネルを画像に含んでいる場合は、テクスチャー上データもアルファチャンネルを持つことになり、そのテクスチャーが貼られた四角形上のピクセルも、そのアルファチャンネルに従って透明になります。Mac OS X に標準添付の Aqua Icons スクリーンセーバーでは、アプリケーションアイコンが飛び交っていますが、ここに書いているのと同様の方法で実現していると思われます。

◎ テクスチャーの描画

続いては、テクスチャーを貼り付けた四角形の描画です。以下のような、drawRectTextureId : alpha : というメソッドを定義します。指定した ID のテクスチャーを指定の透明度で表示するものです。

MyOpenGLView.m > drawRectTextureId : alpha :

```
- (void) drawRectTextureId : (GLuint) texId
                     alpha : (float ) alpha {

  glColor4f( 1.0, 1.0 ,1.0, alpha );     // 四角の色を透明度付きで指定
  glBindTexture( GL_TEXTURE_2D, texId ); // テクスチャー指定

  glBegin( GL_QUADS ); {

    glTexCoord2f(  0.0,  1.0 );  // テクスチャーの左上をマッピング
    glVertex2f(   -1.0, -1.0 );  // 左下の頂点

    glTexCoord2f(  1.0,  1.0 );  // テクスチャーの右上をマッピング
    glVertex2f(   +1.0, -1.0 );  // 右下の頂点

    glTexCoord2f(  1.0,  0.0 );  // テクスチャーの右下をマッピング
    glVertex2f(   +1.0, +1.0 );  // 右上の頂点

    glTexCoord2f(  0.0,  0.0 );  // テクスチャーの左下をマッピング
    glVertex2f(   -1.0, +1.0 );  // 左上の頂点

  } glEnd();

  glBindTexture( GL_TEXTURE_2D, 0 ); // テクスチャー指定解除

}
```

glColor4f の 4 つ目のパラメータでこの四角形の透明度を指定します。そして、対象のテクスチャーを glBindTexture で指定してから四角形の描画になります。グラデーション付きの四角形の時には、各頂点の色を指定していましたが、テクスチャーの場合は、**テクスチャー上のどこの座標をこの頂点に割り付けるかを glTexCoord2f で指定します**。テクスチャーのイメージ全体には、左下 (0.0, 0.0) 〜 右上 (1.0, 1.0) という座標系が割り当たっています。

よく見ると、テクスチャーを上下反転してマッピングしていることが分かると思います。実は、そのままマッピングすると画像が上下反転して表示されてしまうためです。多分、ビットマップのデータの上下の並びが、OpenGL が期待しているものと NSBitmapImageRep の bitmapData メソッドが返すものとが逆だからだと思いますが、このようにして回避しています。

◎ テクスチャーとアルファチャンネルのための初期化

テクスチャーとアルファチャンネル を使うためには、OpenGL に対していくつかの初期化が必要です。そこ

で、initWithFrame : の末尾に以下のコードを追加します。

MyOpenGLView.m > initWithFrame :

```
- (id) initWithFrame : (NSRect) rect {
    : 省略

  // テクスチャー関連

  glEnable( GL_TEXTURE_2D );  // 2Dテクスチャーを使用可能する
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR ); // 拡大
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR ); // 縮小

 // 半透明関連

  glEnable( GL_BLEND );  // アルファブレンディングを使用する
  glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );  // ブレンド時の演算を選択

  return( self );

}
```

glEnable は、OpenGL の機能のどれかをオンにするための関数です。パラメータに **GL_TEXTURE_2D** にすることでテクスチャー機能が使えるようになります。**glDisable** というオフにする関数もあります。**glTexParameteri は、テクスチャーに関する設定を行う関数**で、ここでは、テクスチャーの拡大や縮小で用いる画像補間アルゴリズムを設定しています。**GL_TEXTURE_MAG_FILTER で拡大時の指定**、**GL_TEXTURE_MIN_FILTER で縮小時の指定**が出来ます。**GL_LINEAR は直線補間**になります。**GL_NEAREST を使うと一番近いピクセルを使って補間することができます**。このアルゴリズムの場合、画質は低下しますが、描画は高速になります。用途やマシンスピードに応じて使い分けをすることができます。

glEnable(**GL_BLEND**) でアルファブレンディング (透明度を考慮した画像描画処理) を可能にします。**glBlendFunc** は、ブレンド処理の時の演算種別を指定します。通常はこのままで構いません。そして、この実行結果のサンプルは以下のようになります。カエルの絵をテクスチャーとして貼り付けた四角形を描いています。

【図】テクスチャー付きの四角の描画結果

アルファチャンネルを持たせてあるので、カエルの形状に切り抜かれた四角形として描画されています。

ここまでの内容で、画像のクロスフェードまではできるようになります。次に現れるべき画像を最初に描いておいて、消えていく画像を透明度を時間とともに高くしながらかぶせて描画していくだけです。回転なども先程のメソッドで出来ます。

■ アニメーション

クロスフェードさせるためには、OpenGL でアニメーションを表示させる必要があります。この場合は、単純に display メソッドを繰り返し呼ぶだけで実現できます。例えば、以下のようになります。

MyOpenGLView.m > drawAnimation : 〜 setupTimer

```
- (void) drawAnimation : (NSTimer*) aTimer {
    : アニメーションのための処理
  [ self display ]; // 表示要求
}

NSTimer* animTimer; // アニメーションタイマー

- (void) setupTimer {
  animTimer = [ NSTimer scheduledTimerWithTimeInterval : 0.03
                                        target : self
                                      selector : @selector( drawAnimation : )
                                      userInfo : nil
                                       repeats : YES ];
}
```

drawAnimation : メソッドで、アニメーションをワンステップ進める処理 (透明度を変えるなど) を書いて、その後に display メソッドを呼ぶようにしておきます。それを NSTimer を使って定期的に呼びます。setupTimer メソッドは、タイマーの初期化メソッドですので、initWithFrame : から呼ぶようにします。このサンプルでは、タイマーの間隔が 0.03 秒になっていますので、約 1/30 秒刻みで呼ばれますが、描画処理がさほど重くなければ、これくらい頻繁に呼び出しても問題なくなめらかに動いてくれます。

■ ビューのサイズが変わったとき

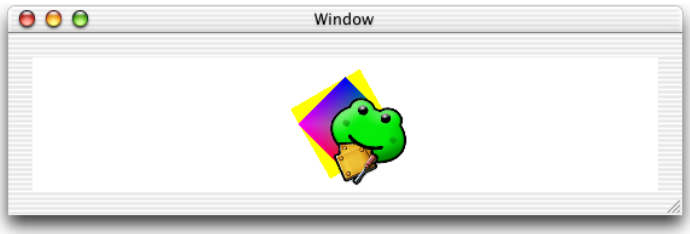
ビューのサイズが変化したときには、**glViewport** 関数を使うことで、OpenGL に描画エリアが変化したことを伝えます。具体的には、以下の resizeView : メソッドに書いてあるように、ビューの位置を渡すだけです。

MyOpenGLView.m > drawAnimation : 〜 setupTimer

```
- (void) resizeView : (NSRect) rect {

  // 画面上の描画範囲を変更
  glViewport( (GLint) rect.origin.x   , (GLint) rect.origin.y,
              (GLint) rect.size.width, (GLint) rect.size.height );

  { // 描画する座標の範囲を変更
    float fRate = rect.size.width / rect.size.height;
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    if      ( 1 <= fRate ) gluOrtho2D ( -fRate, fRate, -1, 1 );
    else if ( 0 <  fRate ) gluOrtho2D ( -1, 1, -1 / fRate, 1 / fRate );
  }

}
```

ただし、ビュー内の OpenGL の座標系は、 (-1.0, -1.0) 〜 (+1.0, +1.0) のままですので、ビューが横に伸びた場合、表示されるイメージも横に伸びてしまいます。OpenGL の座標系を変えるには、**gluOrto2D**、**glOrtho** などの関数を呼びます。このサンプルでは、短い方の辺の座標の範囲を −1.0 〜 +1.0 に固定して、長い方を比率で計算して求めています。

このメソッドを drawRect : の先頭で呼んでおけばよいでしょう。以下が実行結果です。



【図】リサイズ時の実行結果

■ おわりに

最近では、OpenGL は主要 OS の大半に標準で搭載されていますが、他の OS でも OpenGL の扱いは Cocoa と同様で、OpenGL の関数を直接呼ぶものが多いようです。そのため、Cocoa 以外の OS の OpenGL のコードでも OpenGL の部分はそのまま流用できます。これは、サンプルコードは大量に入手できることを意味し

ますし、Cocoa 上で OpenGL を勉強したとしても、OpenGL の知識は他のプラットホーム上でのそのまま通用することを意味します。

冒頭でも触れましたが、OpenGL が使われるケースというのは 3D ソフトに限られなくなってきました。最近の PC にはほとんど 3D アクセラレーターが積まれるようになっていることも要因でしょう。従来では困難だったグラフィックの表現が比較的簡単にできますし、Cocoa からも簡単に使用できることも分かっていただけたと思います。他のソフトとの差別化にも、OpenGL は強力な味方になってくれるでしょう。