A primer for ArteOGL

TSBps

ComicMarket75 (2008.12)

目次。

act.1　はじめに

このドキュメントについて

このドキュメントは
　ArteOGL.frameworkを使って
　サンプル代わりになんか作ってみちゃおう。
　とりあえずインベーダーなんからくちんじゃね？
という内容のモノです。

読むとArteOGL.frameworkをフィーリングで使えるようになるかもしれません。

中身の流れ

とりあえず作ってみちゃおう系なので
開発中のフレームワークで実際に作ってみていきます。
どういう感じで動作するのか把握して、あとは応用で好き勝手にやってください。

要求するもの

読むにあたって、以下のものを用意しておくと良いです。

- Macintoshコンピュータ　1台以上
　　速い方がストレスなくていい。
- DeveloperTools及びXCode
　　インストールする事。
　　iPhone SDKに入ってるXCodeが無駄に強化されてラクチン。
- Objective-Cプログラマ　1人
　　C使えてオブジェクト指向わかってればOK。
　　少なくとも文法把握してればOK。

あると便利なもの

如何あるとよりよいと言えるでしょう。

- フォトショップ及び類するソフト
  - Adobe Photoshop
　　なんだかんだで最強。
　　もう少し安ければ。。。
  - GIMP
　　どうなんでしょ？

・Objective-Cに関する書物
　・ヒレガス本
　　持ってるけど読んでない。今回は関係ないかも。
　・HMDT本
　　持ってないし読んでないけど、読み易いのだろうと推測。
　・荻原本
　　これがあれば十分かと思う。
　　2.0の方が加筆されてて良い。
　　オススメ。

・英語技能
　　ソースコードのコメントは英語でつけてあるので読めないと辛いかも。
　　喋れなくてもいいので、技術者英語を理解してください。
　　残念ながら、単語力の無さには自信があります。

・言語経験
　　複数言語を同時に使う必要は無いのですが、
　　複数言語を知っていないと分からない事ってのがあると思います。
　　英語を喋れる必要は無いですが、外語の存在を知らないと残念なのと同じです。
　　Javaしか触った事が無い人だとObjective-Cってすごく気持ち悪いらしいですね。
　　あと友達の話だと、僕の書くコードはイミフらしいです。
　　コメント少なくてすみません。

オススメする読み方

慣れてる人：
　　ざっと読んで、概要把握して、あとは好き勝手に作ってみる。

慣れてない人：
　　一緒に作ってみる。

プログラム初めての人：
　　Objective-Cを少し触って読めるようにする。
　　とりあえず、動くものを作って喜ぶ。
　　＝＞作って、動かすの繰り返して楽しむ。

act.2　導入する

プロジェクトの開発計画を練る

まずはプロジェクトの開発計画を練ります。
本来ならば色々考えたりしなきゃいけないんですけど
どうせ同人だし趣味だし考えるだけ無駄だし、結局は、勢いです。
仕事じゃないんで、雑に決めてきます。
形式としては反復型開発にでも分類されるんでしょうかしらん。

今回の目的：
　　なんか作ってArteOGL.frameworkに慣れる。
　　（使いながらArteOGL.frameworkをテスト＆デバッグ）

今回の目標：
　　インベーダーゲームを作ってみる。
　　それでArteOGL.frameworkって使えるかもと思わせてみる。

今回の期限：
　　年末のコミケまでに仕上げる。
　　仕上がらなかったら、後日ネットワーク配信します。
　　すみません。ごめんなさい。先に謝っておきます。

今回の成果物；
　　じ・いんべーだー。
　　（The Invader）

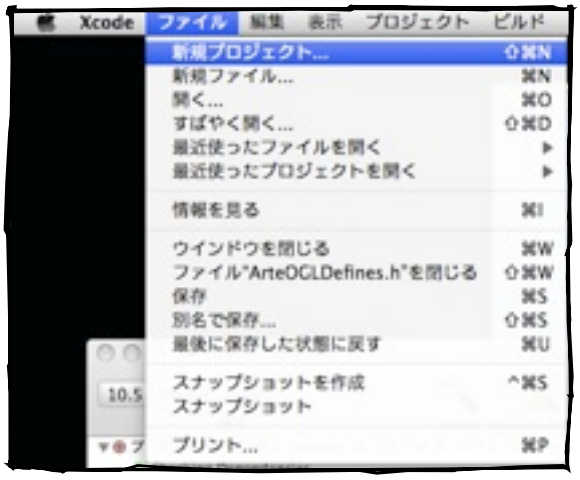
とりあえずは、こんなもので。

インターフェイスなどは既存のインベーダーのパク．．．オマージュする方向で
画像等は逐次、必要に応じて用意していきますので適当に。

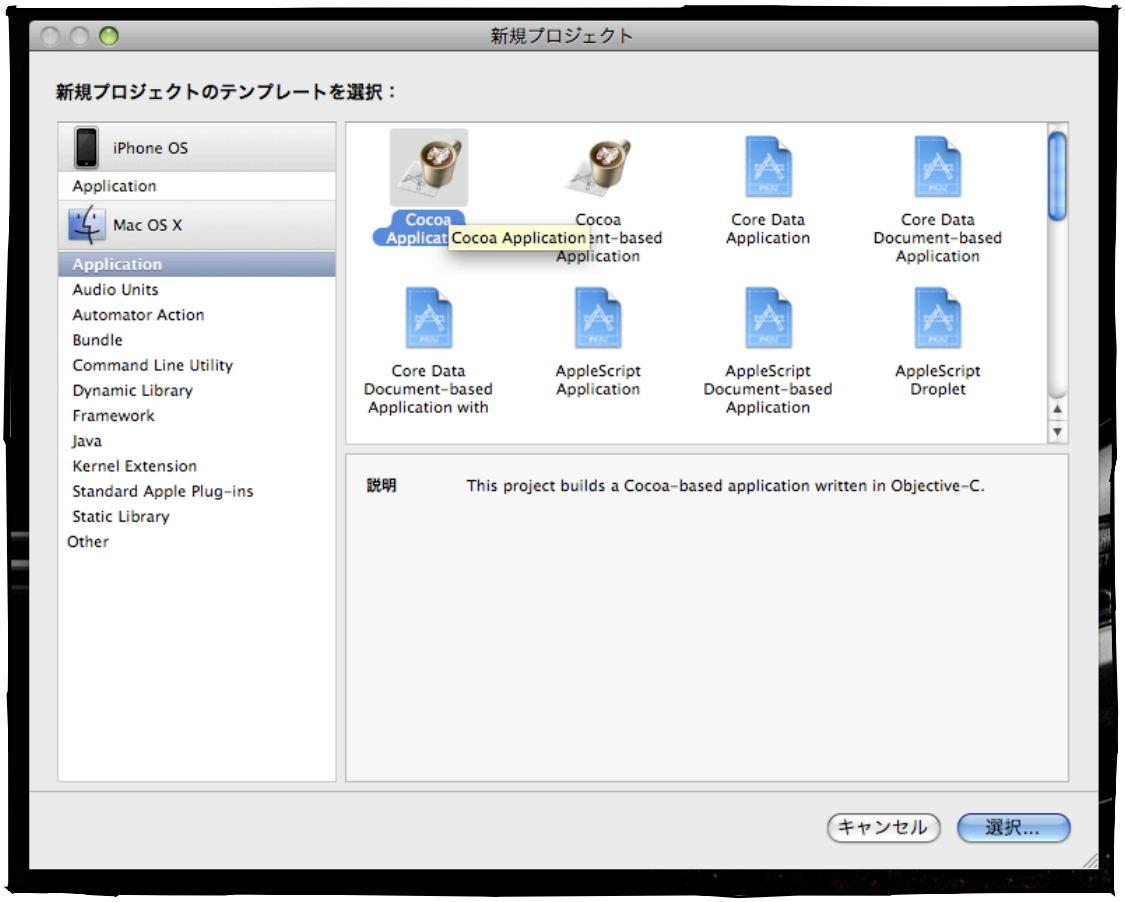
さぁ、始めましょうか。

新しいプロジェクトを作る

まず、Xcodeを起動し、新規プロジェクトを作成します。



プロジェクトの種類は、もちろん、「Cocoa Application」です。
保存場所は適当に指定してください。

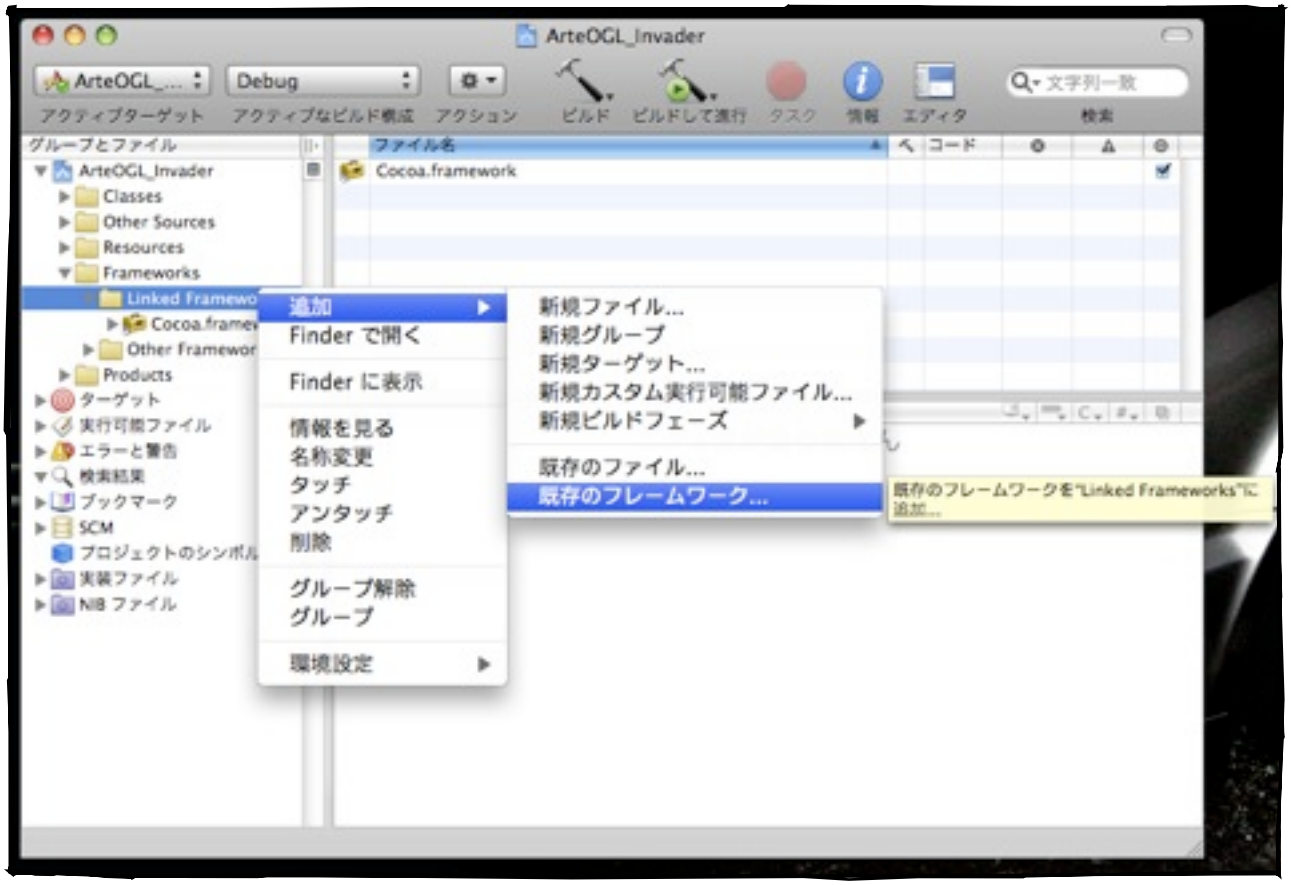


準備をする

出来上がったプロジェクトに対して、ArteOGLを使う準備をしましょう。
ArteOGL.frameworkをプロジェクトのディレクトリにコピーします。
（実は必ずしも必要ではない作業なのですが、やって損は無いでしょう。）

フレームワークを追加する

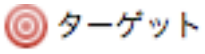
もう早速に、大事なところに突入です。
プロジェクトにArteOGLフレームワークを追加します。



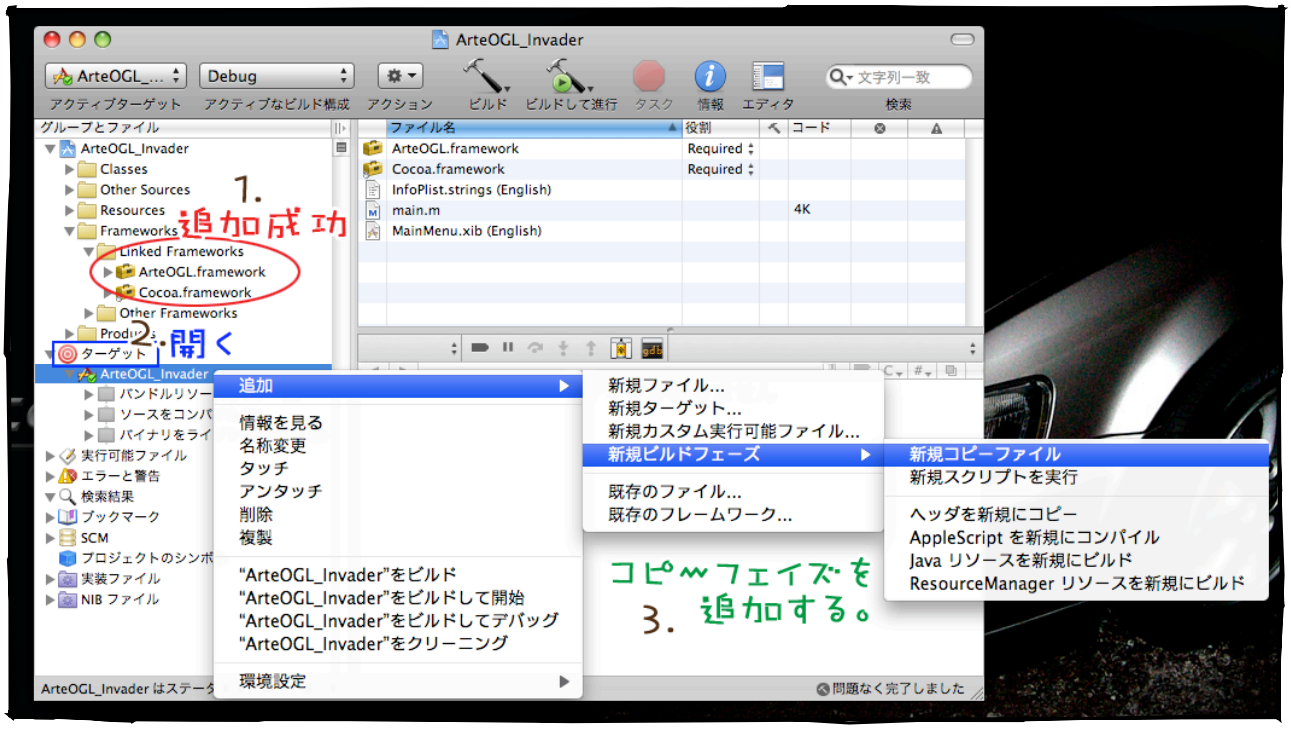
ポップアップメニューから、追加＞既存のフレームワークを選択します。
ダイアログでファイルの場所を問われるので、先ほどにプロジェクトのディレクトリにコピっておいたArteOGL.frameworkファイルを選択してください。
おそらく、ArteOGL.frameworkファイルをドラッグ＆ドロップするのでも大丈夫です。

フェイズを追加する

続いて、大事なところです。
ここまでで、フレームワークはプロジェクトに無事に追加されました。
しかしさらに、ビルドするときに自動でコピるようにXcodeに指示をしなければなりません。
これをやっとかないと、実行時に「フレームワークが見つからない！」とエラーが出て、強制終了させられます。
たしか、エラーコード 5だったはず。

まず、 

を開いて、プロジェクトターゲットを右クリック。
「追加＞新規ビルドフェーズ＞新規コピーファイル」を選択します。

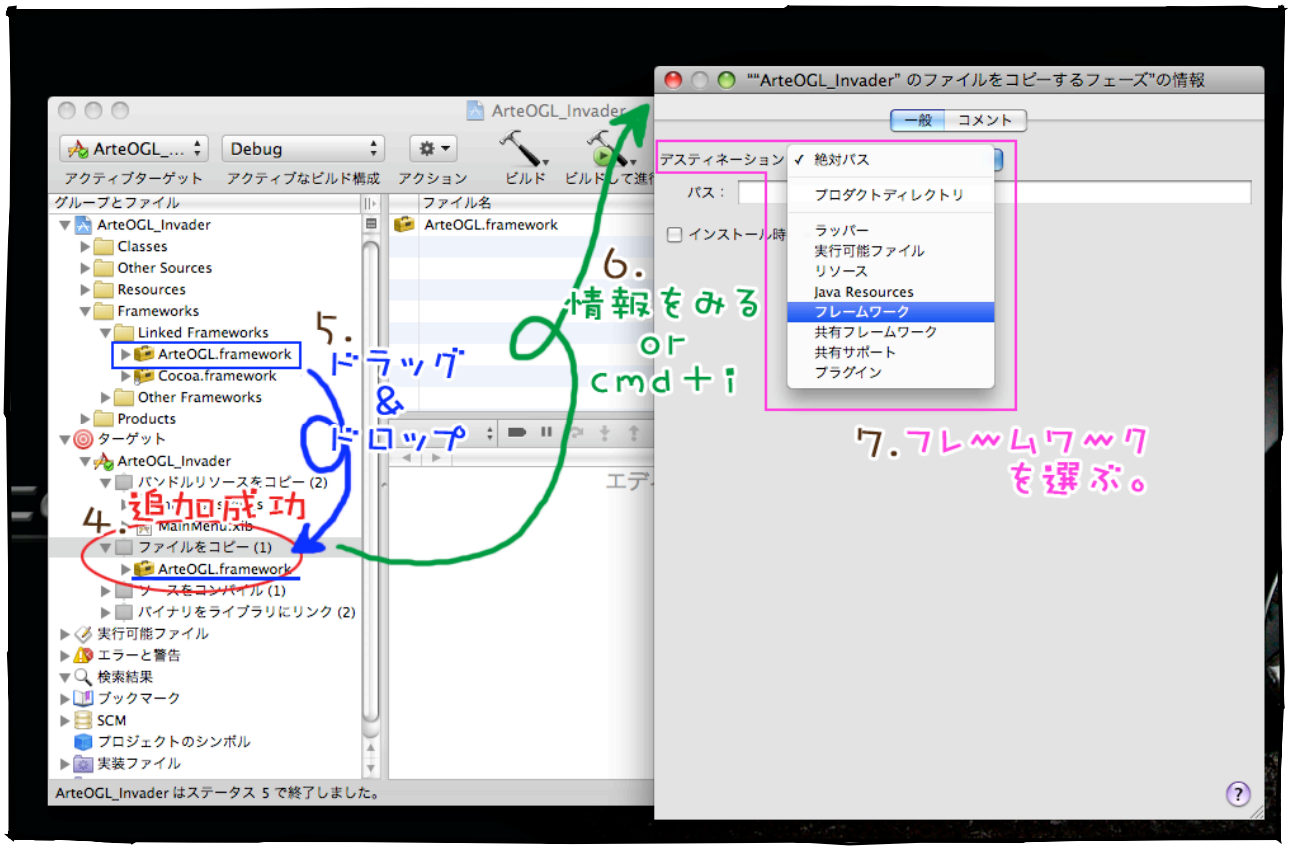


1. 追加成功の確認
   ArteOGL.frameworkが追加されているのを確認します。

2. ターゲットを開く
   ターゲットの左の三角をクリックして中身を展開

3. コピーフェイズを追加する
   右クリックでメニューをポップアップさせ、くいくいっと選んでいきます。

続いて、追加したフェイズを設定します。



4. 追加成功の確認

新しいコピーフェイズが追加されているのを確認します。
順番が不味そうな場合は、適当に並び替えてください。
「バンドルリソースをコピー」の次くらいが良さげです。

5. ドラッグ＆ドロップでフレームワークを登録

先ほどに追加した「ArteOGL.framework」をドラッグし
新しく作った「ファイルをコピー」フェイズにドロップします。

6. 「ファイルをコピー」のフェイズの情報を開きます

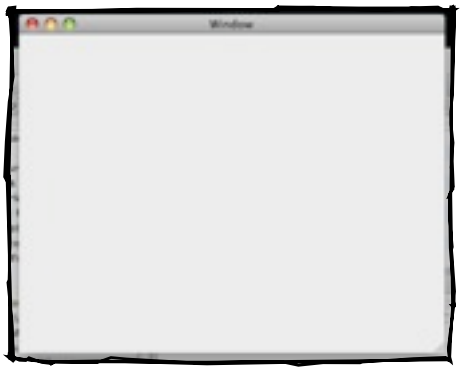
右クリックメニューの「情報をみる」か、コマンド+iでウインドウが開きます。

7. ディスティネーション：フレームワーク

「一般」タブの「ディスティネーション」を「フレームワーク」に設定してください。

以上の作業で、フレームワークの追加は完了です。

ついでに、動作確認を。



設定が終わったら、試しにビルドしてみましょう。

cmd + R、もしくはメニューから「ビルドと実行」を行います。成功すれば、右の様な空のウインドウが表示されるはずです。。

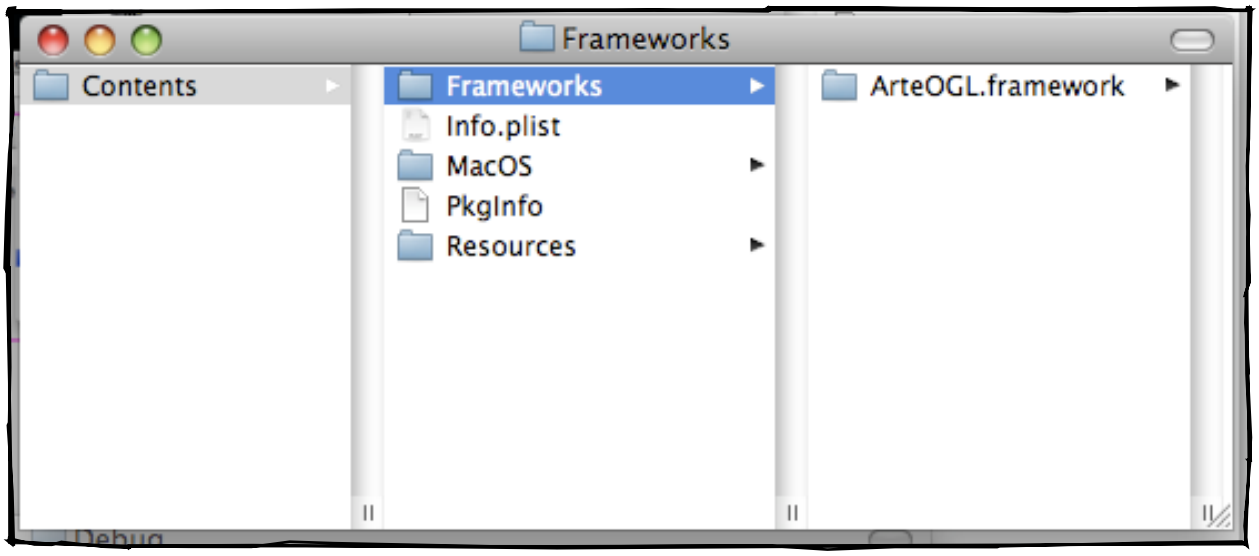
ウインドウが表示されず、エラー -5などと表示されていたら、それは失敗です。フレームワークがうまく追加出来ていないようです。
深呼吸をして落ち着いてから、もう一度やりなおしてみてください。
冷静に、冷静に、原因を考えてください。

ぐーぐる先生に聞いてみるのも良いかもしれません。

一段落。

以上の作業で、ビルド時に自動でフレームワークをコピってくれるようになりました。

この設定を終えた後にビルドすると、下の図の様に勝手に「Frameworks」フォルダが作られ、中に勝手にArteOGL.frameworkがコピられるようになっていると思います。



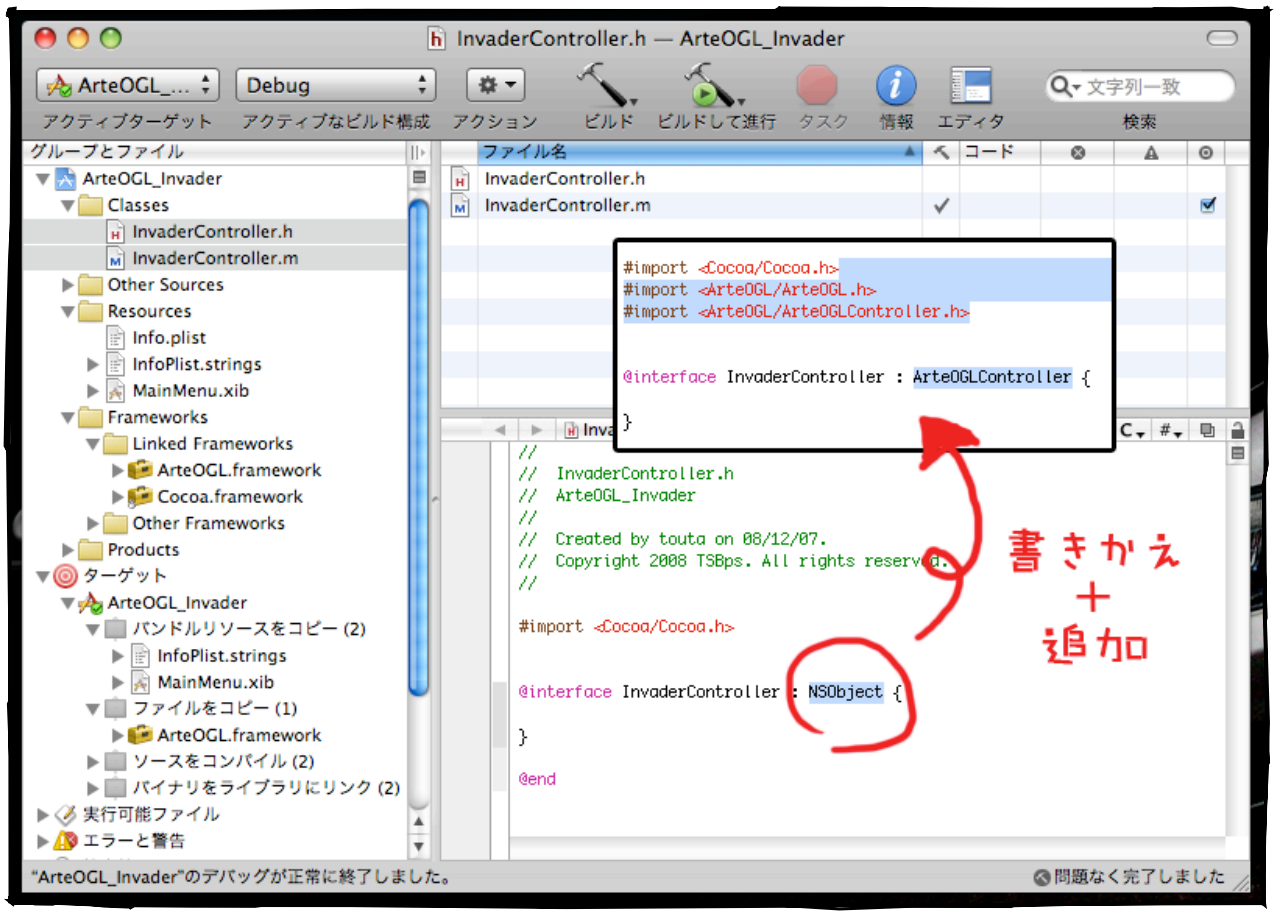
コントローラを作る

フレームワークの追加が完了したところで、
ゲームの制作に入ります。

まずは基本構造作り。
全てを統括するコントローラから作り始めましょう。

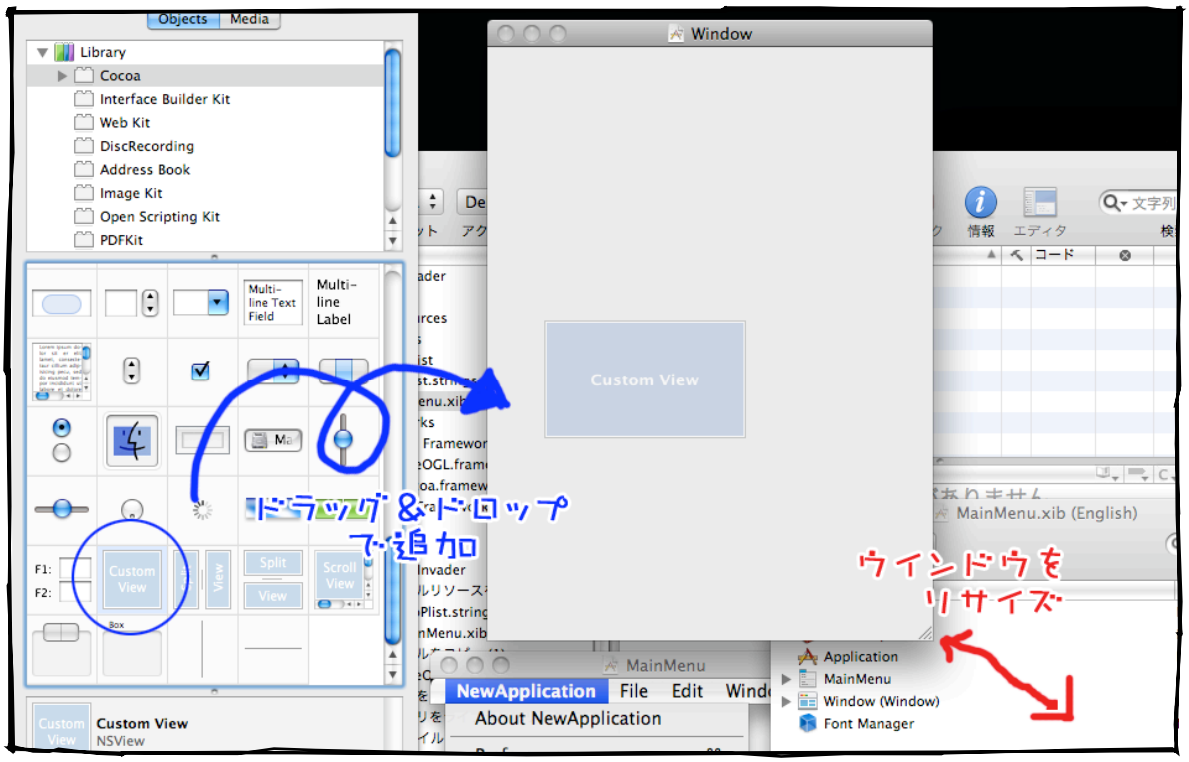
コントローラはNSObjectのサブクラスとして
ファイルを生成した後、親クラスを
「ArteOGLController」に書き換えます。
ArteOGLControllerは、ArteOGLの基礎っぽいところにあるコントローラな感じのクラスです。

また、ArteOGLControllerを使う為に「ArteOGL/ArteOGL.h」と
「ArteOGL/ArteOGLController,h」をimportしておきます。



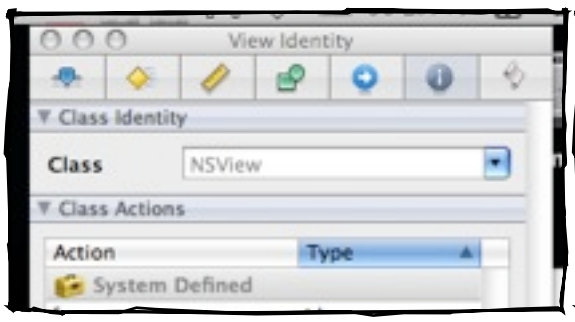
Xibファイルをいぢる

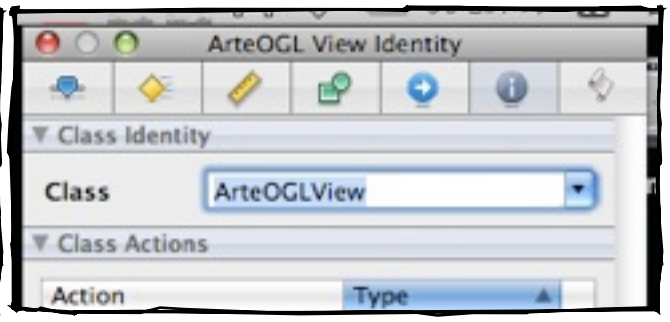
続いて、インターフェイスビルダーでxibファイルを編集します。



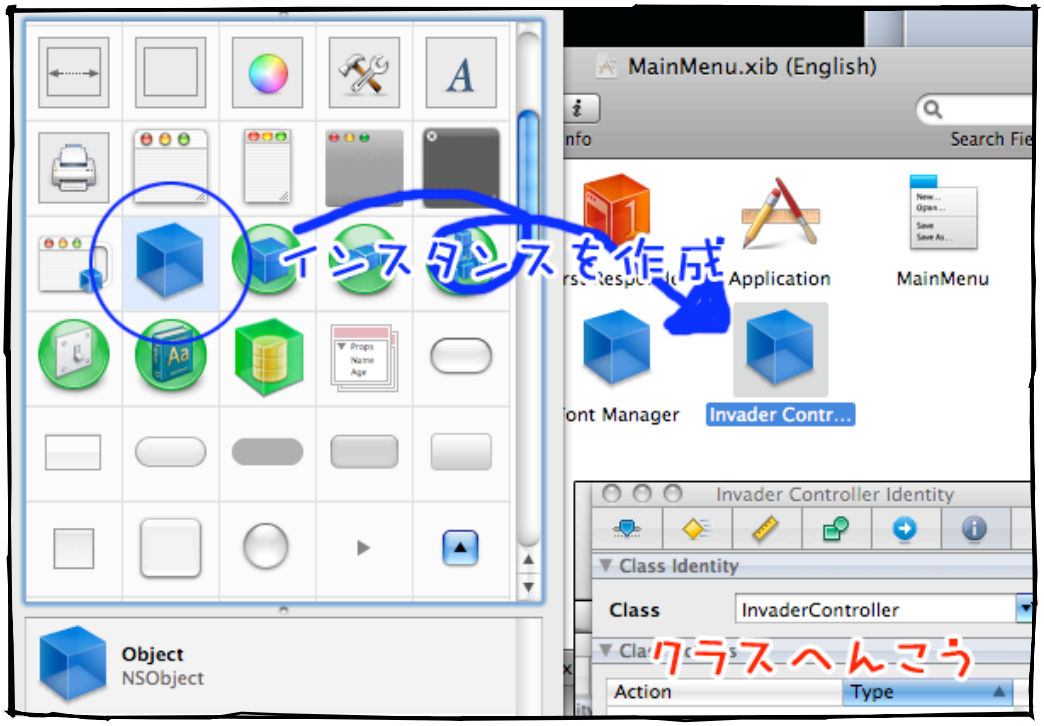
適当に縮めたウインドウにCustomViewを追加します。
ウインドウのサイズは縦横比が4:3になるように適当に設定しました。

次に追加したCustomViewのクラスを「ArteOGLView」に変更します。
File ＞ Read Class Files...から「ArteOGL.framework/Headers/ArteOGLView.h」を選択、クラスを読み込ませた後、フォーカスをビューに合わせ、インスペクタからクラスを変更します。

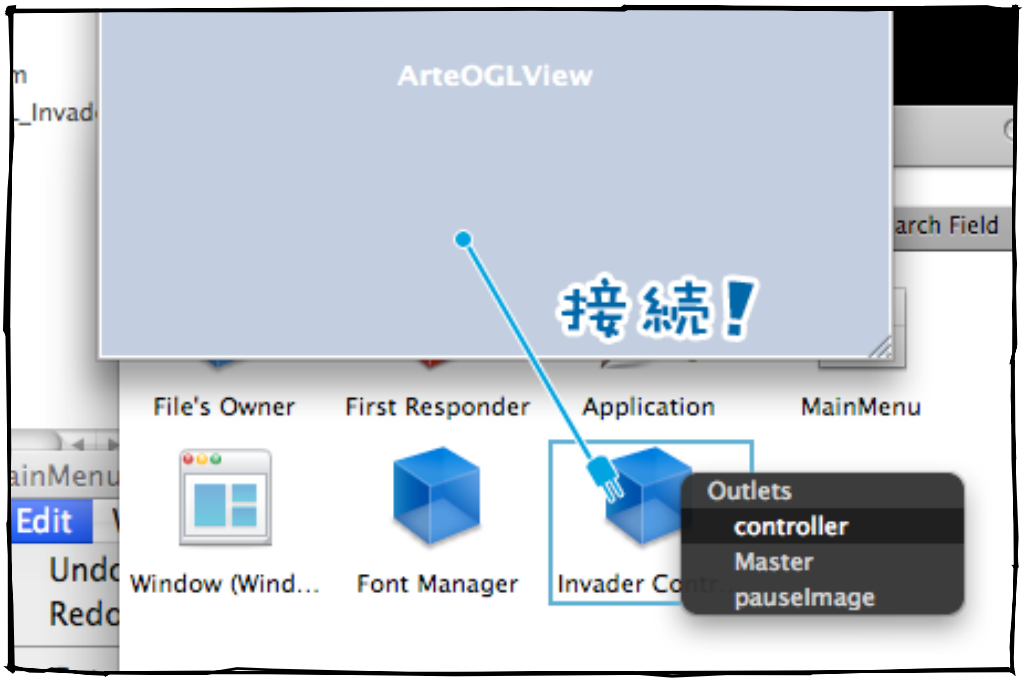




先ほど作成したコントローラクラスをインスタンス化します。



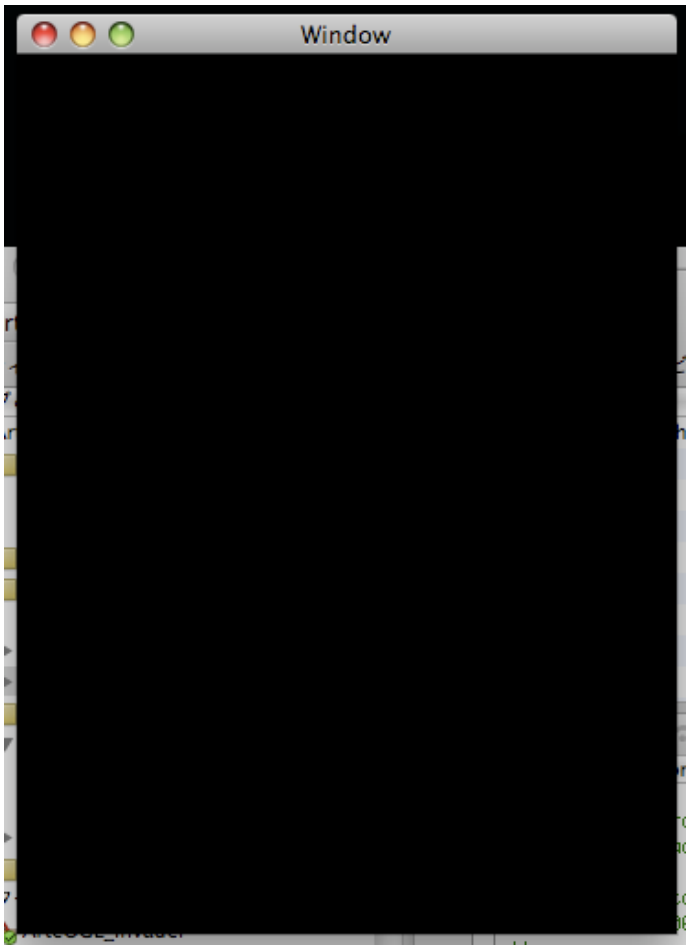
アウトレットとして接続します。
ついでに忘れていたビューの最大化もやっておきます。
ウインドウと同じサイズにしておきます。



コントローラのクラスをcontrollerとして接続します。
（図だと関係の無いアウトレットが入ってますが、気にしないでください。）

動かしてみる。

以上を終えたところで、Xcodeに戻り、ビルドと実行をしてみましょう。

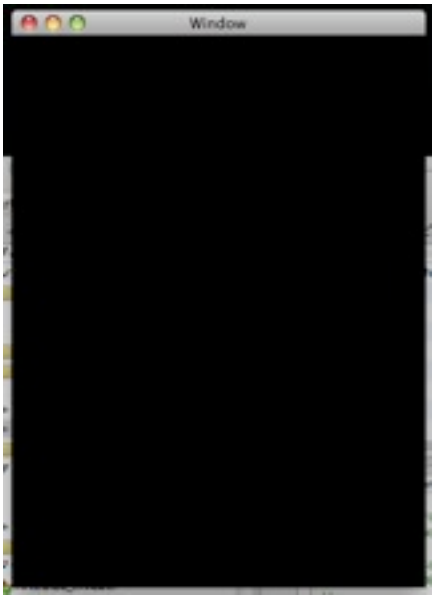
動作成功。
 こんな真っ黒のウインドウが現れていたら成功です。 この黒は、ArteOGLViewの背景色です。

以上で導入編が終了です。
次の章からは、これを使って遊んでいきます。

act.3　使ってみる

画像を追加する

導入編はいかがでしたか？ここからがこのドキュメントの本番になります。
「図＞＞＞文字」だった配置バランスが「コード＞文字＞＞＞図」になり
読みにくく、しかし書き易くなります。図を用意するのって、大変なんですよ。



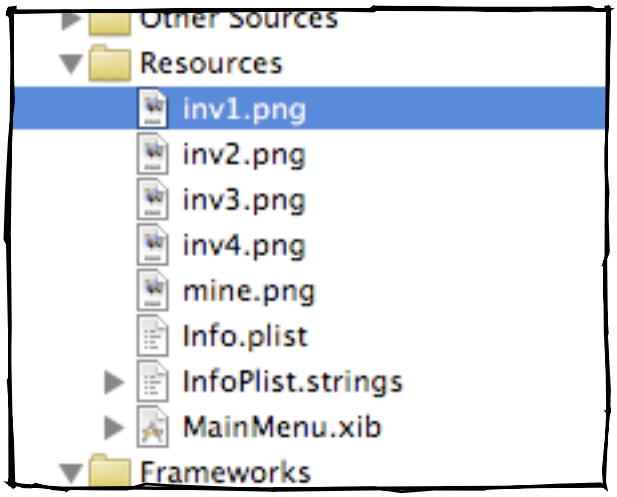
復習すると前回、真っ黒なウインドウを出しました。

まずはこの漆黒の中に、燦然と煌めく感じに画像を浮かせる所から始めてみましょう。

その為に、画像を用意し、プロジェクトに追加します。

作るのはインベーダーゲームなので、それらしいのをさらっと描いてみます。ここでPhotoshopの登場ですね。

これをプロジェクトに追加します。



ArteOGLで使える画像フォーマットは
『NSImageが読めれば何でもOK』です。

ArteOGLとしてはPNGとTIFF（拡張子は.pngと.tif）を推奨します。

画像を表示する

コントローラにコードを追加し、画像を表示させます。

まずArteOGLControllerの `registerAsDelegate` を継承します。

```
- (void)ArteOGL:(id <ArteOGLViewProtocol>)view
          registerAsDelegate:(id)controller;
```

これは本来はControllerにViewを登録するためのものですが、テストにちょうど良いのでこれを用います。このメソッドは、ViewがawakeFromNib:のタイミングでdelegateに対して発行します。

続いて、これにコードを足していきます。

InvaderController.m

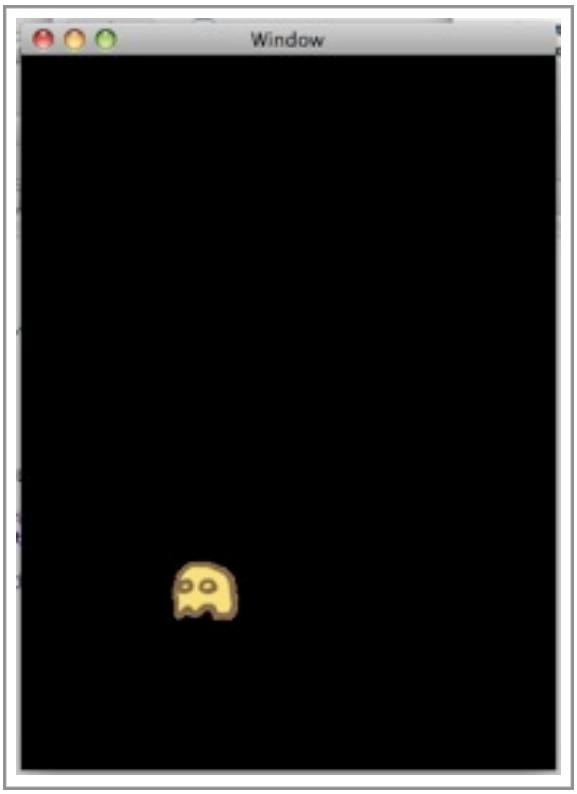
```
#import "InvaderController.h"

@implementation InvaderController

- (void)ArteOGL:(id <ArteOGLViewProtocol>)view
                 registerAsDelegate:(id)controller{

    [super ArteOGL:view registerAsDelegate:controller];

    id obj;

    obj = [ArteOGLImage objectWithImageNamed:@"inv1.png"];

    [view setObject:obj];
}

@end
```

```
[super ArteOGL:view registerAsDelegate:controller];
  // 本来の処理を行うため、親クラスを呼び出します。


id obj;
obj = [ArteOGLImage objectWithImageNamed:@"inv1.png"];
  //変数を宣言し、画像オブジェクトを代入します


[view setObject:obj];
  //オブジェクトをビューにセットします。
```

見て分かるように、説明するまでもないようなコードですが
これがArteOGL.frameworkの基礎になります。

オブジェクト作成
↓
ビューにセット

右が実行結果です。



inv1.pngが画面に表示されています。

inv1.png

ここでコードに

```
NSLog([obj description]);
```

を追加し、デスクリプションを確認すると

```
ArteOGLImage: inv1.png
XYZ(100.000/0.000,100.000/0.000,0.000)
S(0.000,0.000) A(0.000,0.000)
F(0.000,0.000)
R(0.000,0.000,0.000)
```

このオブジェクトが inv1.png を表すArteOGLImageクラスのインスタンスであり、
(x,y) = (100, 100)の位置にあることがわかります。
この位置は、ArteOGLImageクラスの初期設定座標です。

ArteOGLImageは画像を表示する為のクラスで、通常 `objectWithImageNamed:`で画像オブジェクトを生成します。
今回は画像名を拡張子を含めて指定しましたが `objectWithImageNamed:` は
.tifと.pngに限り、拡張子を省略する事も出来ます。

次に、このオブジェクトに対し、属性を設定します。

```
[obj setX:40];
[obj setY:300];
```

この場合のデスクリプションは

```
ArteOGLImage: inv1.png
XYZ(40.000/0.000,300.000/0.000,0.000)
S(0.000,0.000) A(0.000,0.000)
F(0.000,0.000)
R(0.000,0.000,0.000)
```

となります。

ArteOGLは左下座標系を採用しているため、
設定するyの値が左下からの数字になります。

左下座標系については、
『左下が原点、右方向にX座標、上方向にY座標』の
数学でおなじみのグラフ、と言えば分かってもらえるかと思います。

ここまでのまとめ：

InvaderController.m

```
#import "InvaderController.h"

@implementation InvaderController

- (void)ArteOGL:(id <ArteOGLViewProtocol>)view
                 registerAsDelegate:(id)controller{

    [super ArteOGL:view registerAsDelegate:controller];

    id obj;
    obj = [ArteOGLImage objectWithImageNamed:@"inv1.png"];

    [obj setX:40];
    [obj setY:300];

    NSLog([obj description]);

    [view setObject:obj];
}

@end
```

文字を表示する

画像の表示に成功したところで、次は文字を表示しようと思います。
文字の表示には、ArteOGLStringクラスを用います。

InvaderController.m 追記分

```
id string = [ArteOGLString objectWithString:@"The Invader"
                                 grayscale:1.0
                                  fontsize:32];

[string setXY:NSMakePoint(60, 400)];

[view setObject:string];
```

右の実行結果が示す通り。

なんとこれだけで、文字が表示出来ました。

引数に注目すると
`grayscale` は白黒値。
0.0で黒、0.5で灰色、1.0で白と変化。
`fontsize` はフォントサイズ。
単位はpt（ポイント）。

ちなみに32ptは この大きさ。

今回に用いた
`objectWithString: grayscale: fontsize:`
は文字を簡単に生成するためのメソッドなので、
実際に使う場合には

```
- (id)initWithString:(NSString *)str
  attributes:(NSDictionary *)att;
```

を呼んだ方が無難かと思います。

四角形を表示する

ついでに四角形の表示もやっておきます。
四角形は、ArteOGLRectangleクラスです。

InvaderController.m 追記分

```
id rect = [ArteOGLRectangle
            objectWithRectFrame:NSMakeRect(200,50,60,10)];

[rect setColor:[NSColor brownColor]];

[view setObject:rect];
```

四角形もやっぱり簡単ですね。

実行結果は右の通り。

無事に茶色（brown color）の枠（frame）が浮かんでいます。

四角形には
- 枠
- 塗り潰し
- 消去
- 明るく
- ネガポジ反転

の種類があります。

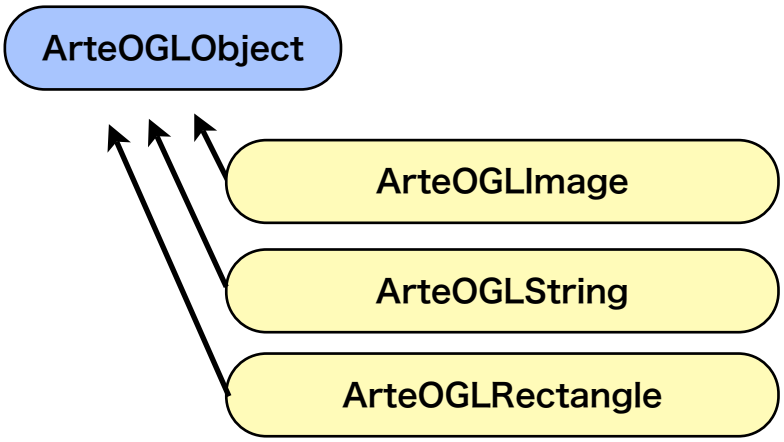
上手く使ってあげてください。

act.4　サブクラスをつくる

継承

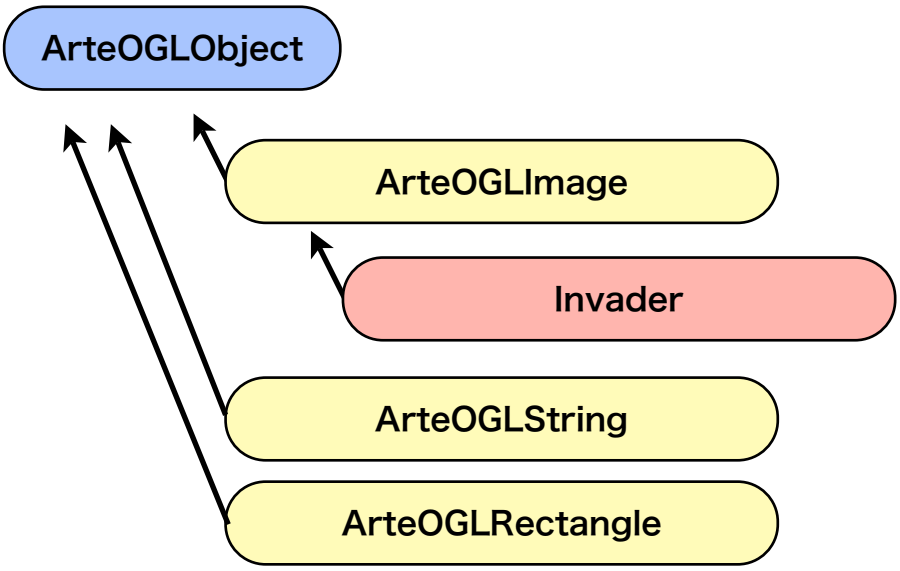
前の章で、Image, String, Rectangleの3つの基礎となるクラスを使ってみました。

これらは、下のような継承関係を持っています。



（ここでArteOGLObjectなどというクラスが初出しました。
これにはArteOGLを構成する為の仕組みが色々詰まってます。
が、ここでは扱いません。詳しくは別のドキュメントかリファレンスをどうぞ。）

さて。
これらのクラスは表示の基本としては十分ですが、これだけではインベーダーゲームを作るのが大変です。
なので、オブジェクト指向言語らしくサブクラスを作ります。



本来はArteOGLObjectから継承すべきなのですが、表示とか面倒くさいので、ArteOGLImageを親にして「Invader」クラスを作成します。

インベーダークラス

ざっと、こんな感じで。

Invader.h

```
#import <Cocoa/Cocoa.h>
#import <ArteOGL/ArteOGL.h>

@interface Invader : ArteOGLImage {
}

- (id)initWithColorNo:(NSInteger)no;

@end
```

Invader.h

```
#import "Invader.h"

@implementation Invader

- (id)initWithColorNo:(NSInteger)no{

    // check arg
    if(no <= 0 || no > 4) return nil;
            // inv1.png ~ inv4.png

    if(self = [super initWithImageNamed:[NSString stringWithFormat:@"inv
%d",no]]){
            texspeed.x = 1;
            moveCount = 0;
    }
    return self;
}

- (void)OGLcalc:(id <ArteOGLViewProtocol>)arteview{
    if(++moveCount>50){
            // step down and turn every 50 frames
            texvirtual.y -= 16;
            texspeed.x *= -1;
            moveCount = 0;
    }
    [super OGLcalc:arteview];
}

@end
```

インベーダークラスは

- 初期化時に色を設定します。(1〜4)
- 毎フレーム、1ピクセル横に動きます。
- 移動回数を覚えておき、50回に1回、特別な動作をします。
    - 縦に16pxほど動きます。
    - 横方向の移動の向きを変えます。

を満たします。

ここで注目すべきは`texspeed`と`texvirtual`、それに`OGLcalc:`でしょうか。

InvaderクラスがArteOGLImageクラスを継承しているように
ArteOGLImageもまたArteOGLObjectクラスを継承しています。
つまり、InvaderはArteOGLObjectを祖父に持ちます。
`texspeed`、`texvirtual`、`OGLcalc:`はその祖父、ArteOGLObjectが持っている値とメソッドです。

まず値について言えば、ArteOGLObjectは以下を持っています。

texpoint	オブジェクトの座標@スクリーン座標
texvirtual	仮想座標（毎フレーム、座標に変換/加算）
texspeed	速度（毎フレーム、座標に加算）
texaccela	加速度（毎フレーム、速度に加算）

texspeedは、そのまま速度なので分かり易いかと思います。
先ほどのコードではtexspeed.x = -1;としていたので
これは毎フレーム、X座標が-1されるということになります。
またtexspeed.x *= -1;とすれば正負が反転し、進行方向が逆転します。

texvirtualは、仮想座標です。今回の場合は座標軸に実軸を使っている（無指定なのでデフォルト値として実軸が選択されます）ので直接にtexpointをいぢっても良いのですが
ここはArteOGLのマナーとしてあえてtexvirtualを操作しています。
変更対象をtexvirtualにしておくことで、座標軸を変更した場合にもコード変更なしに対応出来るようになります。
texvirtualに対して行った変更は座標のモードを問わず、次回の座標変換時に適用されます。

サブクラスの利用

続いて作成したInvaderクラスを、利用・表示してみます。

まず、ヘッダに#importを追加し

InvaderController.h 追加分

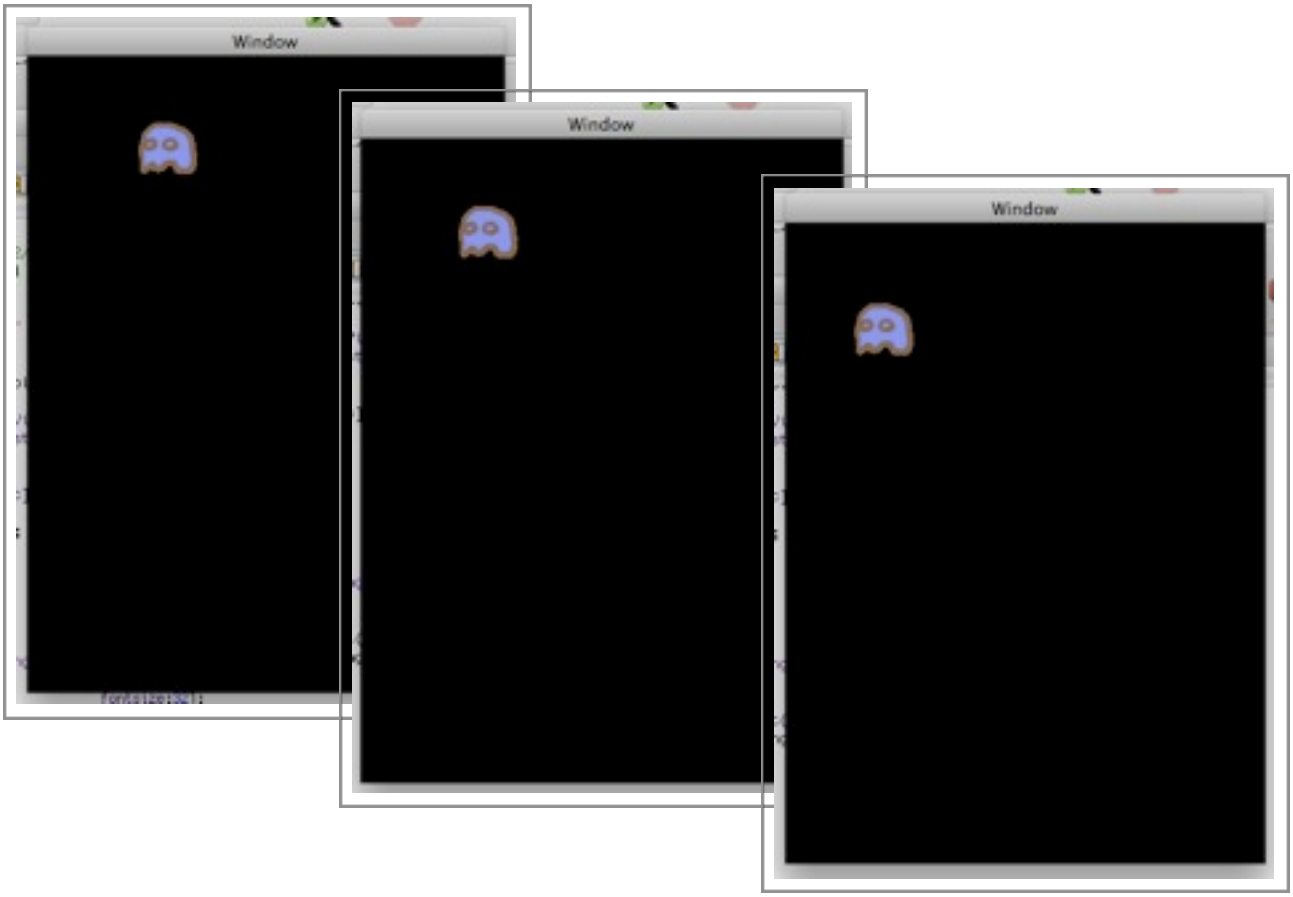
```
#import "Invader.h"
```

前章のテスト用のコードを書き換えます。

InvaderController.m 差し替え分

```
- (void)ArteOGL:(id <ArteOGLViewProtocol>)view
            registerAsDelegate:(id)controller{

    [super ArteOGL:view registerAsDelegate:controller];

    id inv = [[[Invader alloc] initWithColorNo:3 ] autorelease];
    [inv setY: 400 ];
    [arteview setObject:inv];

}
```

実行すると、、、動きました。



・・・ううむ。動きが若干に早いですね。
フレームレートを落として対抗してみましょう。
ArteOGLfps:をオーバーライドしてFPSを15に変更します。
デフォルト値が60なので、1/4の速度になるはずです。

InvaderController.m 追加分

```
- (uint)ArteOGLfps:(id <ArteOGLViewProtocol>)view{
    return 15;
}
```

今回はインベーダーゲームの古くささを演出する為にあえてフレームレートを落としてみましたが、普通はフレームレートではなく、移動幅を変更する気もします。

同じく1/4の速度にしたい場合は

```
texspeed.x = 1;
  ↓
texspeed.x = 0.25;
```

のように変更するとか

あるいは、OGLcalc:の先頭に

```
if([arteview framecount] & 0x3) return;
```

などといったコードを負荷し、4回に1回しか実行させなくするなどが良いでしょう。

また、C言語くさいコードを書くと云う意味では

```
if(++moveCount>50){
  ↓
if([arteview framecount] & 0x40){
```

とかすると、メモリが削減出来て良いですね、sizeof(int)ほど。
これだともとよりさらに14フレーム分ほど遅くなりますが
特に厳密なものでもないので別に構わないかと思います。

閑話休題。
1匹だと寂しいのでたくさん並べてみましょう。

InvaderController.m

```
#import "InvaderController.h"


@implementation InvaderController

- (void)ArteOGL:(id <ArteOGLViewProtocol>)view registerAsDelegate:
(id)controller{
    [super ArteOGL:view registerAsDelegate:controller];

    [self readyGame];
}

- (void)readyGame{

    //title
    id string = [ArteOGLString objectWithString:@"The Invader"
                                                        grayscale:1.0
                                                         fontsize:32];
    [string setXY:NSMakePoint(60, 400)];
    [arteview setObject:string];

    //invaders
    id inv;
    int r,c;
    r=0;
    while (r<8) {
            c=0;
            while (c < 6) {
                  inv = [[[Invader alloc] initWithColorNo: 1 + r%4 ]
autorelease];
                  [inv setX: 60 + 48 * c ];
                  [inv setY: 150 + 40 * r ];
                  [arteview setObject:inv];
                  c++;
            }
            r++;
    }
}

- (uint)ArteOGLfps:(id <ArteOGLViewProtocol>)view{
    return 15;
}

@end
```

と、勝手に追加したメソッドをヘッダにも追加しましょう。

InvaderController.h

```
- (void)readyGame;
```

ArteView:registerAsDelegate:にあまりだらだらと書き連ねるのもエレガントさに欠けるので、ゲームの初期化コードということで、別メソッドに定義を移しました。

これを実行してやると、右になります。
なかなかに圧巻・・・というか、ちょっとキモい？

この辺りで気付いたのですが、インベーダーって、たしか触覚生えてましたよね。
触覚のないコイツらって、インベーダーじゃなくパックマンの敵キャラなんじゃ、、、、なんて、げふんげふん。

コード解説は、するまでもなく、単純に二重のwhileループで6x8個、並べています。
色については行の番号をmod 4したのに+1して1〜4の数字を出してます。

ループがforじゃなくwhileなのは好みです。
カウンタのrとcはrowとcolumnです。

今回、基本的に勢いが信条の開発をしていますので座標等は基本的に目分量と感覚です。
40 + 48 *cとか、150 + 40 *rの根拠は、テキトウという奴です。

蠢く姿を堪能した後は、次のステップ。イベント処理です。

act.5 イベント処理

キー入力を受け付ける

蠢いている虫達…もとい、インベーダーを駆除する為に、自機を作成しましょう。
必要な物は後から追加する事にして、 とりあえずは表示だけ出来れば十分です。

Fighter.h

```
#import <Cocoa/Cocoa.h>
#import <ArteOGL/ArteOGL.h>

@interface Fighter : ArteOGLImage {
}

@end
```

Fighter.m

```
#import "Fighter.h"

@implementation Fighter

- (id)init{
    if(self = [super initWithImageNamed:@"mine"]){
            ;
    }
    return self;
}

- (void)OGLcalc:(id <ArteOGLViewProtocol>)arteview{
    if(texpoint.x<10) texpoint.x = 10;
    else if(texpoint.x>300) texpoint.x = 300;
    [super OGLcalc:arteview];
}

@end
```

またコントローラにも追加します。

InvaderController.h 追加

```
#import "Fighter.h"

@interface InvaderController : ArteOGLController {
    Fighter *fighter;
}
```

readyGameに表示用のコードを追加し

InvaderController.m 追加

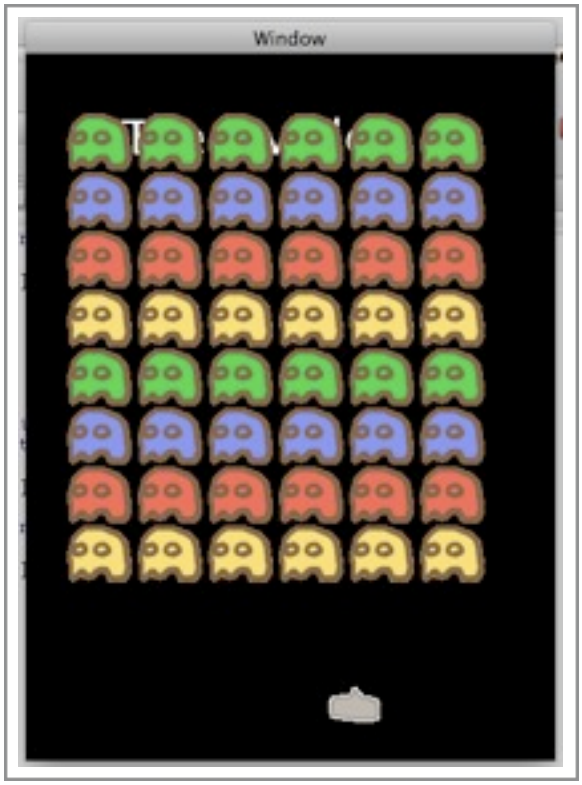
```
    fighter = [[Fighter alloc] init];
    [fighter setXY:NSMakePoint(160, 20)];
    [arteview setObject:fighter];
```

おまけでdeallocも作っておきます。

InvaderController.m 追加

```
- (void)dealloc{
    [fighter release], fighter = nil;
    [super dealloc];
}
```

右が結果です。
若干に貧相な気もしますが、自機が追加されました。



コードを確認すると、表示だけと言いながら、計算部分で移動幅に制限をつけています。

次に、動かす事を考えます。
入力キーと操作の対応を表にしてみます。

キー	操作
左矢印	左へ移動
右矢印	右へ移動
スペース	発射

矢印キーは押されている間だけ
スペースは押した瞬間に作用するのがミソです。

ArteOGLのキーイベントはArteOGL:pressKey:とArteOGL:releaseKey:で拾えます。
名前の通り、押したときと、放したときにイベントが発生します。

ではメソッドをオーバーライドし、コントローラをキー入力に対応させてみましょう。

InvaderController.m　追加

```
- (void)ArteOGL:(id <ArteOGLViewProtocol>)view keyPressed:(unichar)key{
    if(key == NSLeftArrowFunctionKey){
            // Move left
            [fighter setSpeedX:-5];

    }else if(key == NSRightArrowFunctionKey){
            // Move right
            [fighter setSpeedX:+5];

    }else if(key == ' '){
            // Fire

    }
}

- (void)ArteOGL:(id <ArteOGLViewProtocol>)view keyReleased:(unichar)key{
    if(key == NSLeftArrowFunctionKey){
            // Move left
            [fighter addSpeedX:+5];

    }else if(key == NSRightArrowFunctionKey){
            // Move right
            [fighter addSpeedX:-5];

    }else if(key == ' '){
            // Fire

    }
}
```

どうですか？
若干にフレームレートが足りてないようでカクカク気味ですが、ちゃんと自機左右に動いていると思います。

コードを追うに、ただのif文です。変わってるところと云えば、 `NSLeftArrowFunctionKey` と `NSRightArrowFunctionKey`。これらはNSEventで定義されているFunction-Key Unicodeという奴です。詳しくはNSEventのドキュメントを参照してください。
ここは内部的にNSEventを引っ張ってきているので、これらをそのまま使います。

当たり判定

続いて、弾を宣言していきます。

弾を表すBulletクラスは初期化時にインベーダーかファイターを引数に取ります。

Bullet.h

```
#import <Cocoa/Cocoa.h>
#import <ArteOGL/ArteOGL.h>

#import "Invader.h"
#import "Fighter.h"

@interface Bullet : ArteOGLRectangle {

}
- (id)initWithInvader:(Invader *)inv;
- (id)initWithFighter:(Fighter *)fig;

@end
```

わざわざ分けてはいますが、2つの差異は、出現位置と移動方向だけです。

Bullet.m

```
#import "Bullet.h"

@implementation Bullet

- (id)initWithInvader:(Invader *)inv{
    self = [super initWithRect:NSMakeRect([inv x]+10, [inv y]-3, 4, 8 )];
    texspeed.y = -5;
    arteteam = [inv arteTeam];
    atkrect.size = texsize;
    attackFlag = YES;
    return self;
}

- (id)initWithFighter:(Fighter *)fig{
    self = [super initWithRect:NSMakeRect([fig x]+20, [fig y]+30, 4, 8 )];
    texspeed.y = +5;
    arteteam = [fig arteTeam];
    atkrect.size = texsize;
    attackFlag = YES;
    return self;
}

@end
```

またこれに合わせて、InvaderとFighterも変更をして

Invader.m　初期化コードに追加

```
arteteam = ArteBetaTeam;
exirect.size = texsize;
targetFlag = YES;
```

Fighter.m　初期化コードに追加

```
arteteam = ArteAlphaTeam;
exirect.size = texsize;
targetFlag = YES;
```

コントローラに発射処理を加えます。

InvaderController.m　キーイベント処理に追加

```
// Fire
if(!fighterBullet){
        fighterBullet = [[Bullet alloc] initWithFighter:fighter];
        [arteview setObject:fighterBullet];
}
```

さて、ここで新しいモノがいくつか出たので確認します。

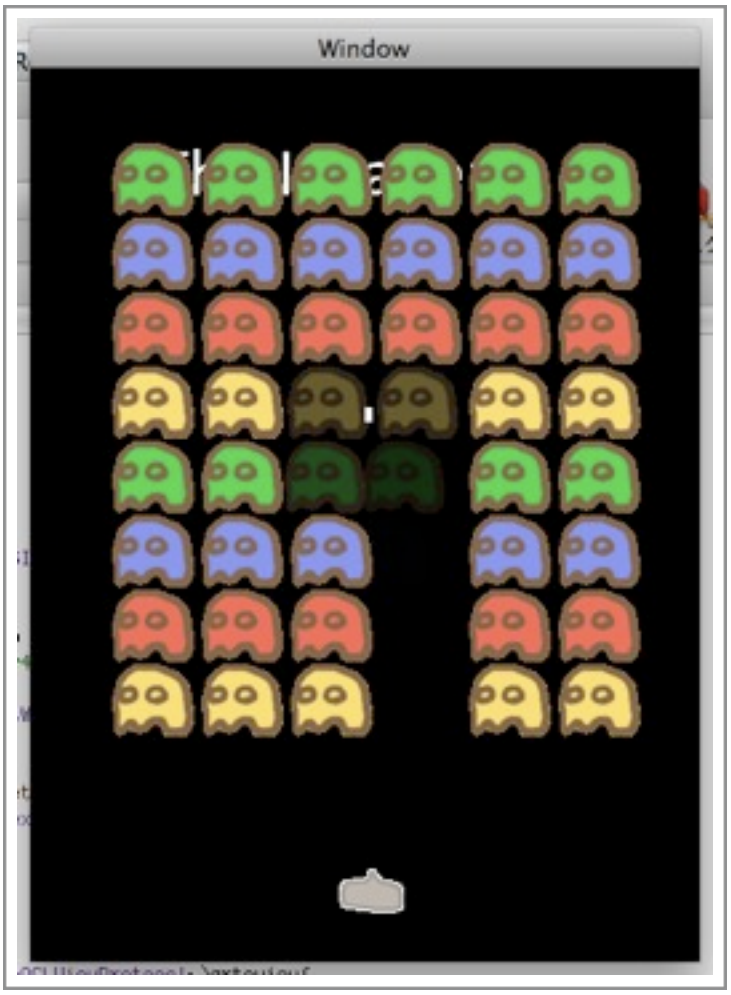
arteteam	チーム分け
attackFlag	攻撃判定フラグ
atkrect	攻撃判定エリア
targetFlag	目標判定フラグ
exirect	目標判定エリア

これらは、当たり判定を行うのに必要な値です。

ArteOGLの当たり判定は全てのオブジェクトに対して総当たりで行います。
全てのattackFlag:YESなオブジェクトが、targetFlag:YESのオブジェクトについて
互いの所属するチームが異なる場合のみに攻撃の可否を調査します。具体的には
自身のatkrectから算出される攻撃範囲と、相手のexirectから算出される目標範囲が
重なっているかを評価し、重なっていれば攻撃成功（＝当たり）という判定になります。

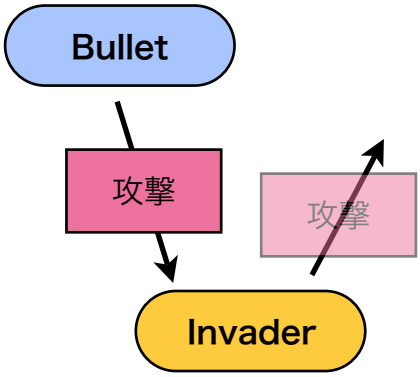
これらを設定する事で、後は勝手に当たり判定がなされます。

これを実行してみると・・・



あれ？なんかよくわからないことになってますね。
確かに当たり判定は動いていますが、
貫通してたくさん消えていっています。

これは、攻撃判定が弾側にしかないからです。
弾の攻撃判定はインベーダーに有効ですが、インベーダーに攻撃判定が無いため、弾に反撃出来ません。
結果、インベーダーが一方的にやられていきます。



これに対処するには、インベーダーに当たり判定を持たすのが手っ取り早いのですが
今回は別の手法でやってみたいと思います。

手を加えるのは、Invader.m。

Invader.m　追加

```
- (void)OGLhitBy:(ArteOGLObject *)obj{
    [super OGLhitBy:obj];
    [obj kill];
}
```

攻撃を受けた場合のデフォルトの挙動は、死ぬ事（[self die];）ですが
それに加えて、相手を殺す([obj kill];)するようにしてみました。
これで敵に当たった弾は消えてくれます。

objectDied, objectOvered

現状にはまだ不満が残ります。今のコードだと、弾が1発しか撃てません。
インベーダーゲームでは撃った弾を回収してからしか、次の弾が撃てないのです。

ので、回収します。

InvaderController.m　追加

```
- (void)ArteOGL:(id <ArteOGLViewProtocol>)view objectDied:(id)obj{
    if(obj == fighterBullet) [fighterBullet release], fighterBullet = nil;
}
```

弾が死んだ場合（＝敵に当たって殺された場合）に、次弾が撃てるよう準備します。

これで敵に当たった弾は回収出来るようになりました。
でも当たらなかった弾は？

どこか、画面の外へ飛んでいってしまったのを捕獲する為の仕組みを稼働させます。

InvaderController.m　差し替え

```
- (void)ArteOGL:(id <ArteOGLViewProtocol>)view
            registerAsDelegate:(id)controller{
    [super ArteOGL:view registerAsDelegate:controller];

    [arteview setStageBoundToScreenSize];

    [self readyGame];
}
```

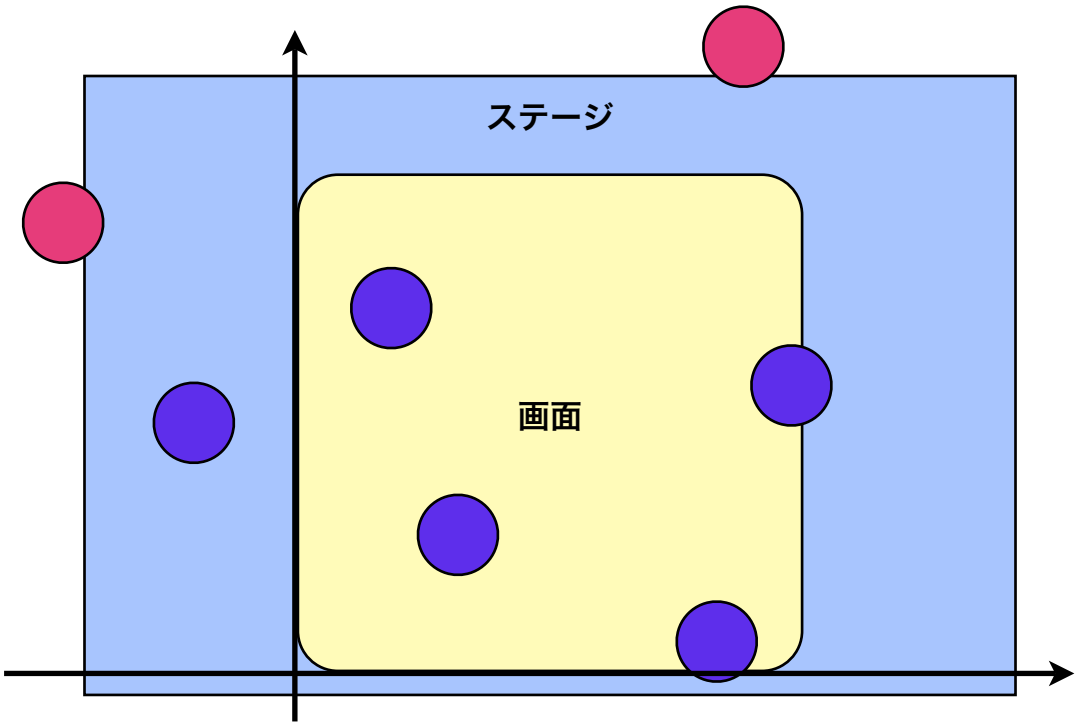
setStageBoundToScreenSizeはステージの大きさを画面サイズと同じに設定します。
ステージが設定された場合、その範囲を超えた位置にあるオブジェクトは
描画の度に通知されるようになります。

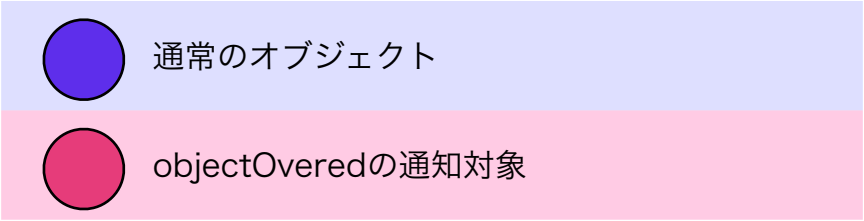
そして、通知を受け取ります。

InvaderController.m　追加

```
- (void)ArteOGL:(ArteOGLView *)view objectOvered:(id)obj{
    if(obj == fighterBullet) [fighterBullet release], fighterBullet = nil;
}
```

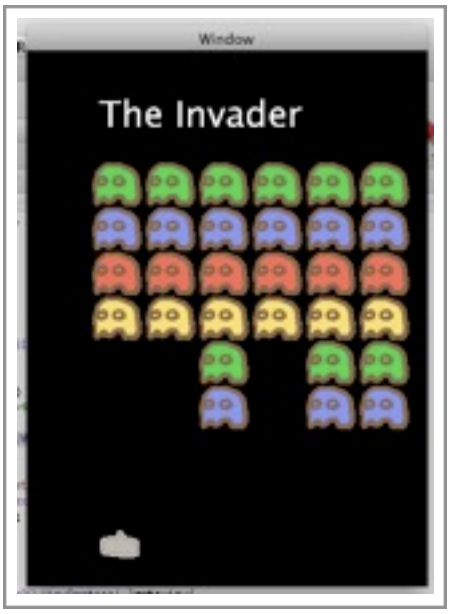
ステージの概念図。





ステージは仮想軸上に展開されるので、画面サイズより大きくても、小さくても良い。
今回は仮想軸を使わないので、素直に ステージサイズ＝画面サイズ とする。

以上で、自機が動くところまで来ました。



次は、ゲームシステムを構築します。
敵の攻撃と、勝利条件の設定です。

act.6　ゲームにする

インベーダーゲームにする

ここまでだいぶ駆け足で来ましたが、大丈夫でしょうか。
見た目がゲームらしくなったところで、ゲームシステムを構築し、ゲームにします。

現在の状況は、ただ動くだけの敵を撃つだけで、インベーダーゲームとは別の物になってしまっています。これを、ちゃちゃっと直します。

良い子の皆さんはちゃんと最初から設計してあげてください。

InvaderController.h　変更

```
@interface InvaderController : ArteOGLController {
    NSMutableArray *invaderQueue;
    Fighter *fighter;
    Bullet *fighterBullet;
    ArteOGLString *msg;
}
- (void)readyGame;
- (void)checkGame;
- (void)clearGame;
- (void)failGame;

@end
```

まずヘッダから。
列上に並んだインベーダーを保持するinvaderQueue、
メッセージ表示用のmsg、をインスタンス変数に追加定義。
ゲームクリアのチェックを行うcheckGameと
ゲームクリア/オーバー時の処理を行うclearGameとfailGameを用意。

流れとして
readyGame
 | -> CheckGame
 |　　v

 | -> clearGame
failGame
を構築する。

続いて、Invaderを生成後に保持するように変更。
またその手間を減らす為に横列単位生成から縦列単位で生成に変更。

Invaderは弾の発射の都合から縦列単位で管理する。

InvaderController.m　変更

```
- (void)readyGame{
    [arteview removeAllObjects];

    //title
    id string = [ArteOGLString objectWithString:@"The Invader"
                                                        grayscale:1.0
                                                         fontsize:32];
    [string setXY:NSMakePoint(60, 400)];
    [arteview setObject:string];

    //invaders
    id inv;
    [invaderQueue release], invaderQueue = [[NSMutableArray array] retain];
    int r,c;
    c=0;
    while (c<6) {
            r=0;
            [invaderQueue addObject:[NSMutableArray array]];
            while (r < 8) {
                  inv = [[[Invader alloc] initWithColorNo: 1 + r%4 ]
autorelease];
                  [inv setX: 60 + 48 * c ];
                  [inv setY: 150 + 40 * r ];
                  [[invaderQueue lastObject] addObject:inv];
                  [arteview setObject:inv];
                  r++;
            }
            c++;
    }

    fighter = [[Fighter alloc] init];
    [fighter setXY:NSMakePoint(160, 20)];
    [arteview setObject:fighter];
}
```

追加メソッドを書く。

InvaderController.m　追加

```
- (void)checkGame{
    NSMutableArray *currentline;
    if(!invaderQueue) return;
    int c=[invaderQueue count];
    while(--c>=0){
            currentline = [invaderQueue objectAtIndex:c];
            while([currentline count]>0){
                  if(![[currentline objectAtIndex:0] livingFlag])
                        [currentline removeObjectAtIndex:0];
                  else break;
            }
            if([currentline count] == 0)
                  [invaderQueue removeObject:currentline];
    }
    if([invaderQueue count] == 0) [self clearGame];
}

- (void)clearGame{
    [arteview removeAllObjects];
    [invaderQueue release], invaderQueue = nil;
    [msg release];
    msg = [[ArteOGLString objectWithString:@"!!! clear !!!"
                                     grayscale:1.0
                                      fontsize:32] retain];
    [msg setXY:NSMakePoint(90, 0)];
    [msg setSpeedY:2.0];
    [arteview setObject:msg];
}

- (void)failGame{
    [arteview removeAllObjects];
    [invaderQueue release], invaderQueue = nil;
    [msg release];
    msg = [[ArteOGLString objectWithString:@"... game over ..."
                                         grayscale:1.0
                                          fontsize:32] retain];
    [msg setXY:NSMakePoint(50, 0)];
    [msg setSpeedY:2.0];
    [arteview setObject:msg];
}
```

checkGameでは各縦列について、死んだインベーダーを取り除いていき
インベーダーの居なくなった縦列は、これも取り除く。
全ての縦列がなくなったら、ゲームクリア。

clearGame、failGameではメッセージを生成、縦に流す。

敵弾発射と敵侵略

InvaderController.m　追加

```
- (void)ArteOGLdidCalculate:(id <ArteOGLViewProtocol>)view{
    int c=[invaderQueue count];
    id inv;
    while (--c>=0) {
            inv = [[invaderQueue objectAtIndex:c] objectAtIndex:0];
            if(![inv livingFlag])continue;
            if([inv y] < 50) [self failGame];
            if((random() % 200)==0){
                  id bul = [[[Bullet alloc] initWithInvader:inv]
autorelease];
                  [arteview setObject:bul];
            }
    }
}

- (void)ArteOGL:(id <ArteOGLViewProtocol>)view objectDied:(id)obj{
    if(obj == fighterBullet){
            [fighterBullet release], fighterBullet = nil;
    }else if(obj == fighter){
            [self failGame];
            return;
    }
    [self checkGame];
}

- (void)ArteOGL:(ArteOGLView *)view objectOvered:(id)obj{
    if(obj == fighterBullet){
            [view removeObject:obj];
            [fighterBullet release], fighterBullet = nil;
    }else if(obj == msg){
            [view removeObject:obj];
            [self readyGame];
            return;
    }else [arteview removeObject:obj];
}
```

ArteOGLdidCalculate:をオーバーライドし、計算後のタイミングでインベーダーの攻撃処理及び侵略チェックを行う。各縦列の先頭のインベーダーが1/200の確率で弾を発射。
また先頭のy座標が50を下回ると侵略成功と看做し、ゲームオーバーへ。

ArteOGL:objectDied:では自機が死亡した場合にゲームオーバーへ。

ArteOGL: objectOvered:ではメッセージが行き過ぎた場合にゲームリセット。
それ以外のオブジェクト（敵弾等）はビューから取り除く。

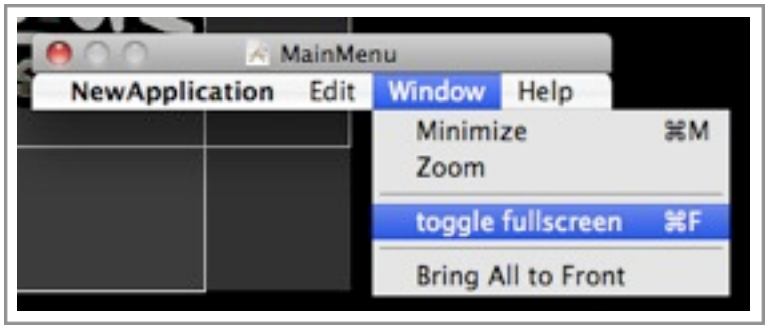
調整する

おおよそ出来上がったところで、雑なところを調整する。

・xibを調整

ウインドウのタイトルを変更し
Close, Resize, Minimizeのボタンを外す。

フルスクリーンをメニューに追加する



InvaderController.m　変更

```
- (uint)ArteOGLfps:(id <ArteOGLViewProtocol>)view{
    return 30;
}
```

やっぱり30fpsにしてみる。

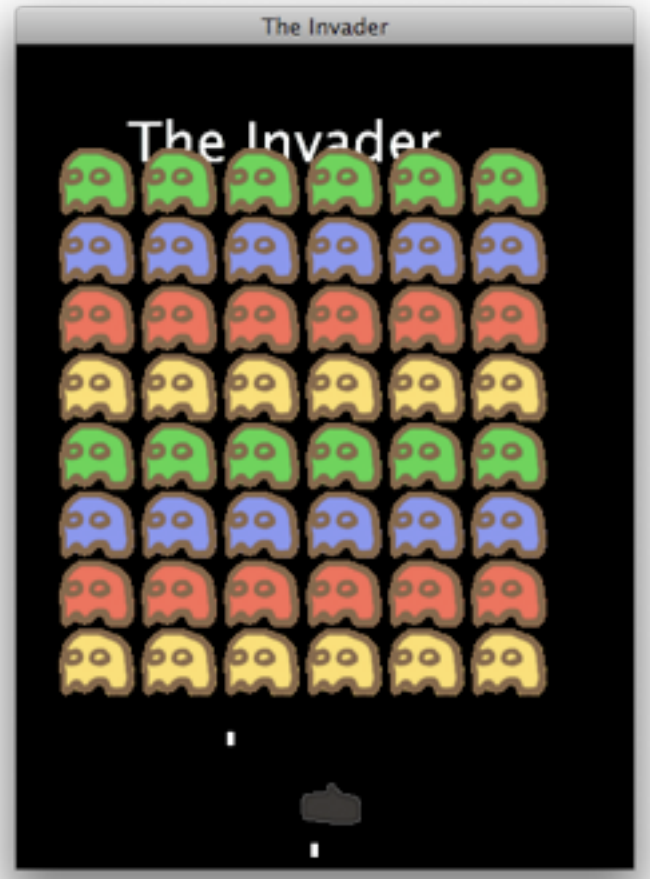
Fighter.m　変更

```
- (id)init{
    if(self = [super initWithImageNamed:@"mine"]){
            arteteam = ArteAlphaTeam;
            exirect = NSMakeRect(6,2,texsize.width-12, texsize.height/2);
            targetFlag = YES;
    }
    return self;
}
```

標的領域を調整して小さくしてみる。

できあがり

というような感じで、インベーダーゲームが出来ました。



防御のブロックがなかったり、残機1機ですぐゲームオーバーだったり
結構に突っ込みどころはあるのですけども
その辺は各自で機能追加して遊んでみてください。

act.7　UniversalBinaryにする

最後に、Universal Binaryについて少し。

MacOS X v10.3 <Panther>

ただUniversalBinaryにするだけならば簡単ですが
MacOS X v10.3対応にするのが、大変です。

理由
・MacOS X v10.4でCocoa.frameworkが大きく整理された。
・MacOS X v10.5で追加されたObjective-C 2.0のラクチン文法が使えない。

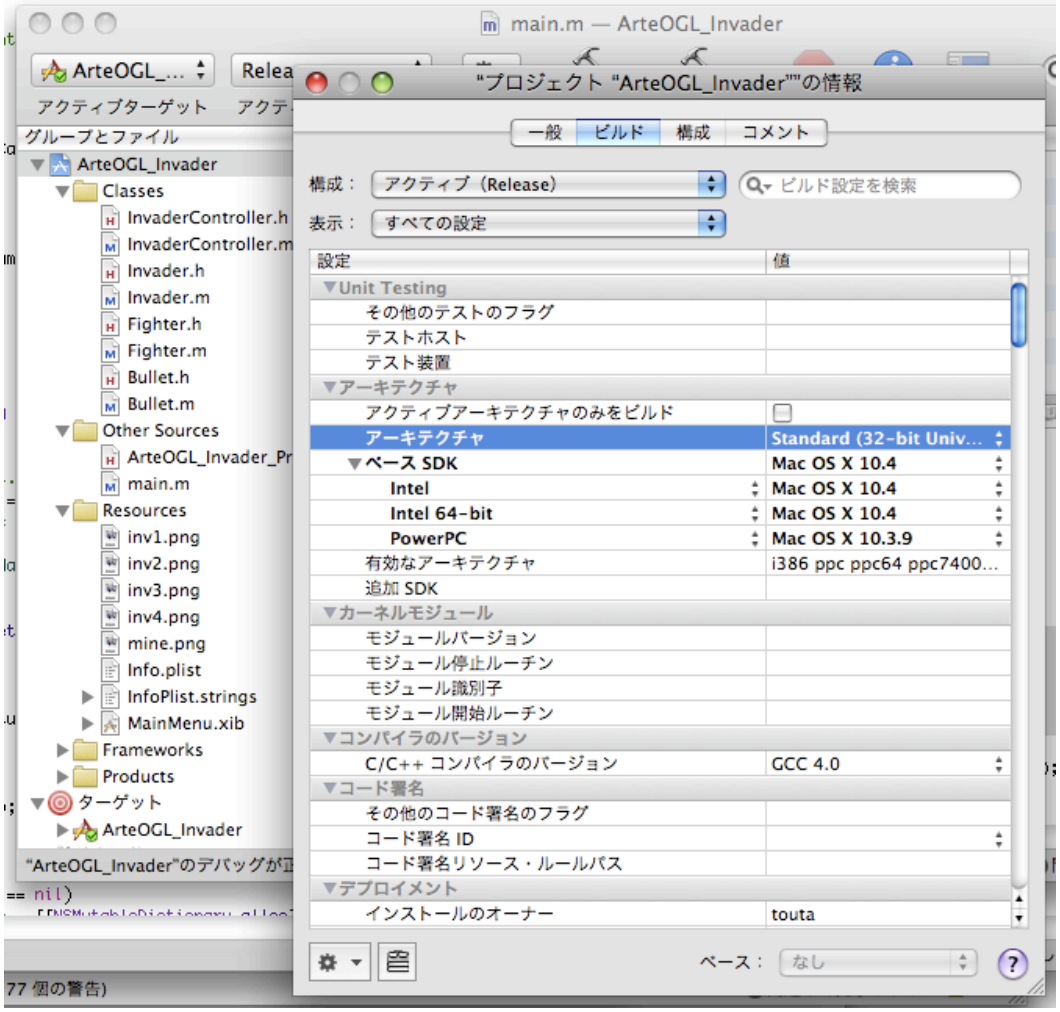
具体的には
・使えるメソッドを選ばなきゃいけない。
　・Deprecated in Mac OS X v10.4
　・Available in Mac OS X v10.4 and later.
　　の両方が使えない。
　　便利なメソッドは大抵v10.4から追加された
・NSInteger, CGFloatなどが使えない
　　int、floatを使わなきゃいけない。
　　せっかく癖が付き始めたのに。
・for(id obj in array)が使えない
　　もうwhile文書くのだるいー。

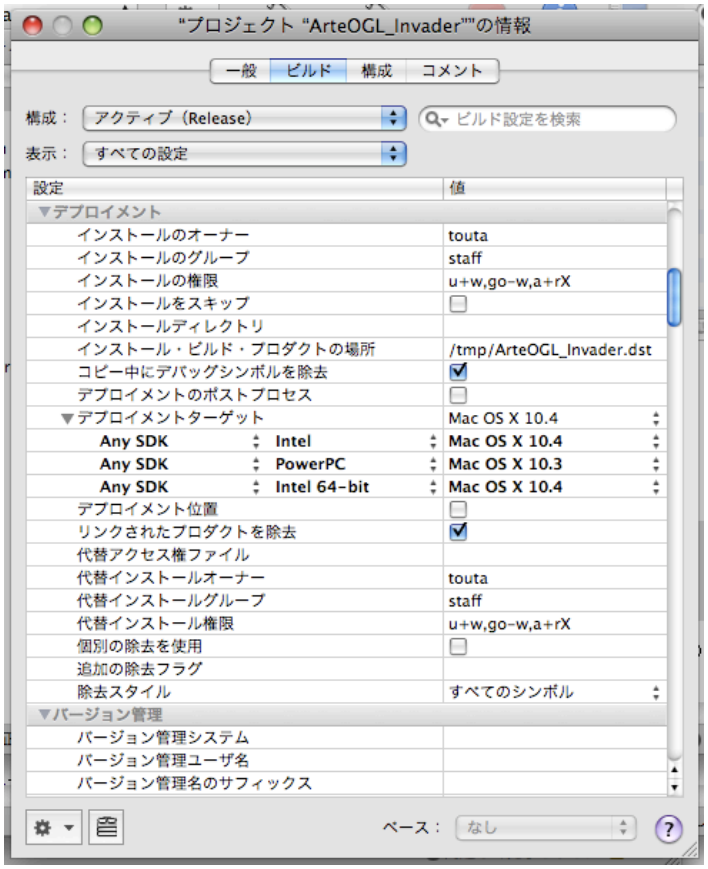
以上を心得た上で、XCodeでの設定方法。

1. プロジェクトのインスペクタを開く
2. 「ビルド」タブを開く
3. 「ベースSDK」を選択
4. 左下の＊から「ビルド設定条件を追加」を実行 x 3回
5. 各々に設定する。

Intel	MacOS X v10.4
Intel 64-bit	MacOS X v10.4
PowerPC	MacOS X v10.3.9

6. 下方へスクロールし「デプロイメントターゲット」についても
　同様に「ビルド設定条件を追加」 x 3回 ＋ 各々に設定。


main.m — ArteOGL_Invader
ArteOGL_...
アクティブターゲット
グループとファイル
ArteOGL_Invader
Classes
InvaderController.h
InvaderController.m
Invader.h
Invader.m
Fighter.h
Fighter.m
Bullet.h
Bullet.m
Other Sources
ArteOGL_Invader_Pr
main.m
Resources
inv1.png
inv2.png
inv3.png
inv4.png
mine.png
Info.plist
InfoPlist.strings
MainMenu.xib
Frameworks
Products
ターゲット
ArteOGL_Invader
"ArteOGL_Invader"のデバッグが正
77 個の警告)
"プロジェクト "ArteOGL_Invader""の情報
一般
ビルド
構成
コメント
構成：
アクティブ（Release）
ビルド設定を検索
表示：
すべての設定
設定
値
Unit Testing
その他のテストのフラグ
テストホスト
テスト装置
アーキテクチャ
アクティブアーキテクチャのみをビルド
アーキテクチャ
Standard (32-bit Univ...
ベース SDK
Mac OS X 10.4
Intel
Mac OS X 10.4
Intel 64-bit
Mac OS X 10.4
PowerPC
Mac OS X 10.3.9
有効なアーキテクチャ
i386 ppc ppc64 ppc7400...
追加 SDK
カーネルモジュール
モジュールバージョン
モジュール停止ルーチン
モジュール識別子
モジュール開始ルーチン
コンパイラのバージョン
C/C++ コンパイラのバージョン
GCC 4.0
コード署名
その他のコード署名のフラグ
コード署名 ID
コード署名リソース・ルールパス
デプロイメント
インストールのオーナー
touta
ベース：
なし


"プロジェクト "ArteOGL_Invader""の情報
一般
ビルド
構成
コメント
構成：
アクティブ（Release）
ビルド設定を検索
表示：
すべての設定
設定
値
デプロイメント
インストールのオーナー
touta
インストールのグループ
staff
インストールの権限
u+w,go-w,a+rX
インストールをスキップ
インストールディレクトリ
インストール・ビルド・プロダクトの場所
/tmp/ArteOGL_Invader.dst
コピー中にデバッグシンボルを除去
デプロイメントのポストプロセス
デプロイメントターゲット
Mac OS X 10.4
Any SDK
Intel
Mac OS X 10.4
Any SDK
PowerPC
Mac OS X 10.3
Any SDK
Intel 64-bit
Mac OS X 10.4
デプロイメント位置
リンクされたプロダクトを除去
代替アクセス権ファイル
代替インストールオーナー
touta
代替インストールグループ
staff
代替インストール権限
u+w,go-w,a+rX
個別の除去を使用
追加の除去フラグ
除去スタイル
すべてのシンボル
バージョン管理
バージョン管理システム
バージョン管理ユーザ名
バージョン管理名のサフィックス
ベース：
なし

7. SDK設定を「プロジェクト設定を使用」
8. 構成をReleaseに設定する

9. ビルド
10.出たエラーを直す。
11.MacOS X v10.3.9の動いてる実機で動作テスト。

以上、終わり。

見た目大丈夫そうでも、実際に動かしてみないと動かなかったりします。
ちょっとした設定ミスが命取り。

奥付

A primer for ArteOGL	
著者	とうた
発行者	TSB program system
発行日	2008/12/27