Macintosh OpenGL Programming Guide

（1）導入

このドキュメントでは、Mac OS Xに実装されているOpenGLの機能について解説します。OpenGLは、広く業界で支持されているクロスプラットフォーム向けのオープンスタンダード3Dグラフィックライブラリです。OpenGLは「Open Graphics Library」の略で、Silicon Graphics社（SGI）のシステムで用いられていたIRISグラフィックライブラリから生まれ、v1.0が1992年に発表されています。OpenGL APIは、Mac OS XのCarbonとCocoa両アプリケーションから呼び出し利用することが可能です。

・なにゆえOpenGLを使うのか？

・Apple社が責任を持ってMac OS Xに実装しているため信頼性が高い。
・プラットフォーム非依存で、別システムからのアプリケーション移植も容易。
・業界標準であり、Mac OS X以外でもWindows, Linux, Irix, Solarisで利用可能。
・GPUの進化とともにOpenGL自身のパフォーマンスも大きく進歩している。
・ハードウェアがサポートする新機能をいちはやく導入できる仕組みを備えている。
・500を越える2Dや3Dに関するグラフィックルーチンを利用することができる。

（2）Macintoshメディアアーキテクチャ

Macintoshは、マルチメディア処理についてハイレベルなパフォーマンスを提供するように設計されています。この章では、こうしたパフォーマンスを最大限に生かすために、どのようなアキテクチャが形成されているのかを紹介します。ハイパフォーマンスの実現は、先進的なハードウェア設計と、Mac OS Xに実装されたグラフィックソフトウェアの両輪で達成されています。ハードウェア側で重要な部分は、2Dや3D描画の高速化を実現しているプログラマブルGPU（Graphics Processing Unit）ですが、されに重要なのは、その機種差をアプリケーション側から隠し、共通のAPI経由で機能を使えるようにしている「ハードウェア抽象化」という仕組みです。このハードウェア抽象化において最も重要な役割を担っているのがOpenGLとなります。

・Mac OS X メディアサービス

Mac OS Xでは様々なメデイアサービスをユーザに提供しています。そうした役割を担当しているシステムモジュールを上位階層から順に紹介してみます。

・QuickTime：Movieの作成や編集など最もハイレベルなメディアサービスを担当する
・Core Video：デジタルビデオフレームの処理に対してパイプラインモデルを提供する
・Core Image：ビデオや静止画像の処理やフィルタリングAPIを提供する
・Quartz 2D：Mac OS Xネイティブの2Dグラフィックライブラリ
・Quartz Compositor：Mac OS Xにおいてウィインドの合成と管理を担当する
・Quartz Services：Mac OS Xのウィンドウサーバ機能へのアクセスを可能にする
・OpenGL：ハードウエァに最も近いローレベルなグラフィックス処理を担当する

（1）QuickTime

QuickTimeはMac OS XとWindowsのどちらでも利用できるクロスプラットフォーム・マルチメディアアーキテクチャです。QuickTimeはシステムの機能拡張として提供されています。また、QuickTime Playerアプリケーション、QuickTime ActiveXコントロール、QuickTimeブラウザプラグインなどの単独ソフトも含まれています。

QuickTimeは、映像、サウンド、静止画、バーチャルリアリティ（VR）などの様々なメディアコンテンツの作成、制作、配布をサポートします。QuickTimeには、映像やサウンドのリアルタイム録画、録音、同期、メディアの各種変換、編集、合成、圧縮、配布、再生などに必要な機能がAPIとして盛り込まれています。

（2）Core Video

Core Videoは、Mac OS Xにおいてデジタルビデオのパイプライン処理を実行します。Core Videoを使えば、ディスプレイとの同期やデータの変換を気にすることなく個々のビデオフレームを操作することが可能です。

QuickTimeのMovie映像などから切り取られたフレーム画像は、OpenGLのコンテキストへ描画されます。通常コンテキストはウィンドウ内のビュー（View）ですが、それをオフスクリーンメモリ領域に設定することも可能です。

（3）Core Image

Core Imageは、Mac OS XにおいてGPUの能力を使い（機能がなければCPUを使う）ビ

デオ映像や静止画像のフィルタリングを実行します。また、独自のフィルタをユーザが用意することも可能です。Core Imageは、OpenGLの「OpenGL Shading Language」を用いたプログラミングモデルを用いることで、こうした画像処理を実現しています。

（4）Quartz 2D

Quartz 2Dは、Core GraphicsフレームワークにMac OS Xネイティブの2Dグラフィックライブラリです。Quartz 2Dは、解像度とデバイスに依存することなくアプリケーションから利用でき、業界標準のPostScriptとPDFをベースに、ColoSyncによるカラーマネージメントやATSによるフォント描画を実現しています。

Quartz 2Dには、ベジェ曲線描画、透明レイアの追加、パスベースの図形描画、オフスクリーンへの描画、アンチエイリアス描画、正確なカラーマネージメント、PDFドキュメント作成などの機能があります。開発者は、Quartz 2D APIのCGGLContextCreate()を使うことで、OpenGL描画対象コンテキストに対して、Quartz 2Dによる2D描画を実行することも可能です。

（5）Quartz Compositor

Quartz Compositorは、Mac OS Xにおいてスクリーン上のウィンドウの合成表示、オーバーラップ、透明度の調整、書き換えなどを担当しています。こうした処理はユーザの操作時に自動的になされます。

（6）Quartz Services

Quartz Servicesは、Mac OS Xウィンドウサーバの機能のうちディスプレイの機器構成と制御、Mac OS Xのリモートオペレーションへなどを提供します。Quartz Servicesは、旧システムの「Macintosh Display Manager」と「DrawSprockt」に置き換わるシステムモジュールです。

Quartz Services APIは、ディスプレイの解像度、階調、リフレッシュレートなどを制御でき、加えて、モニタ画面のフェードイン・アウト、ミラーリング、カラーコレクションの調整なども実行可能です。

（7）OpenGL

OpenGLは、Mac OS Xメディアサービスの最下層でグラフィック処理を担当します。ハードウェア抽象化により、ハードウェアに依存しない機能を上位階層に提供します。

・グラフィックツールの選択

Mac OS Xには、各階層に色々なグラフィックス用APIが用意されています。開発中のアプリケーションからどのAPIを利用するか迷う場合には、なるべくハイレベル階層のAPIを利用するようにします。各階層には、その目的が重複しているAPIが多々ありますが、そうした場合にも上位階層のAPIを選択します。下層レベルのAPIについては、本当にそのサービスが必要なとされているケースに限定して使用した方が良いでしょう。

（1）サンプル

どのような場合にどの階層のAPIを利用するかを、以下のサンプルアプリケーション開発のシナリオにそって調べてみます。開発中のアプリケーションは、複数の画像やMovieを映像として表示する必要があります。また、各フレームは個々に修正する必要があり、その表示先はウィンドウ上の3Dオブジェクトのサーフェス（面）です。また、オプションとしてアプリケーションを起動したままモニタの解像度を切り換える操作も可能とします。

複数画像やMovieを映像として表示するには、ファイルフォーマットや圧縮方法に注意しながらQuickTime APIを利用します。ビデオ映像を表示する直前のフレームのフィルタリングや修正については、オフスクリーンを使い、QuickTimeのコールバックルーチンで処理することができます。しかし、Core ImageかCore Videoを利用した方が簡単な場合もあり、コールバックルーチンの代わりにCore Videoのフレームパイプラインを用います。

画像のオーバーレイ表示にはQuart 2D APIを用い、Movie再生や画像のインポート処理ではQuickTime APIを使います。また、パイプライン処理やフィルタ処理などについては、Core ImageやCore Videoを使います。最終出力を回転している立方体などの3Dオブジェクトのサーフェス（面）に描画させるのには、QuickTimeやCore Videoの出力先をスクリーンへではなくオフスクリーン・テクスチャバッファに指定し、そのデータ内容をOpenGLのテクスチャとして利用します。

3Dオブジェクトのウィンドウへの表示、表示内容の更新、オーバーラップなどについては、Quartz Compositorが自動で実行します。この場合、不適当なピクセルフォーマットやグラフィックモードのまま画像合成が起こる状態は避けるようにします。アプリケーションを起動したままモニタの解像度を切り替えるには、Quartz Services APIを利用します。

（2）特定のアプリケーションタスク

以下の表は、アプリケーションでの様々な処理が、Mac OS Xのどのメディアサービスに対応しているのかを示しています。

処理	QuickTime	Core Video	Core Image	Quartze 2D, Services	OpenGL
簡単なユーザインターフェース	スプライト、アニメーション			アイコン、ボタン、etc	
2D描画			画像フィルタリング	ハイレベルな描画処理	ローレベルな描画処理
画像処理	キャプチャ、フォーマット変換		画像フィルタリング		レンダリング
映像編集	フルサービス	フレーム画像処理			
3D描画			画像フィルタリング		描画、レンダリング
ゲーム	メディア再生、VR	ビデオエフェクト		スクリーン全般の制御	描画、レンダリング

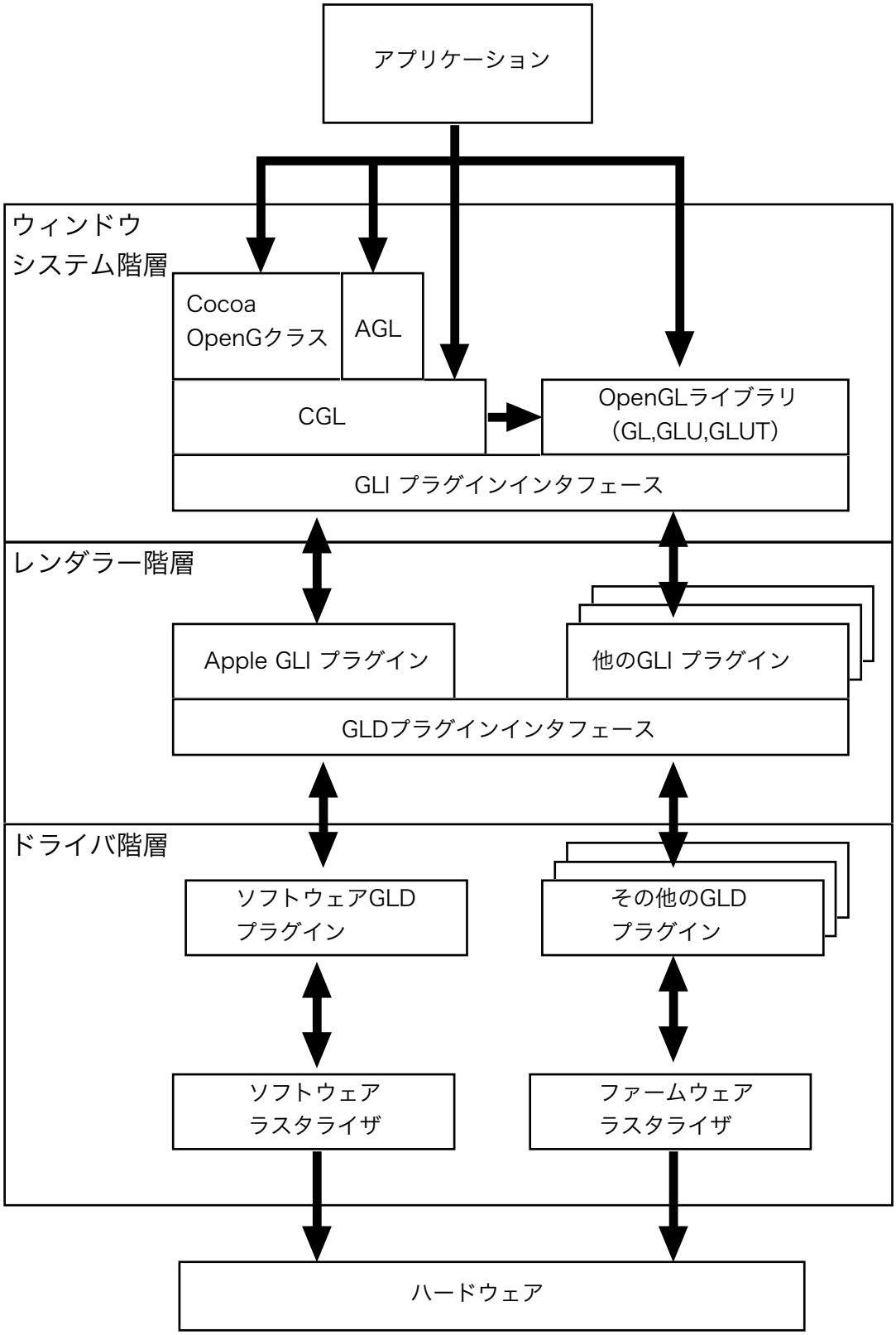
（3）MacintoshにおけるOpenGLの実装

Mac OS XのOpenGLは、OpenGLランタイムエンジンとその描画ソフトを含んだフレームワークのセットとして実装されています。これらのフレームワークは、プラットフォームに中立な仮想リソースを用い、できる限りハードウェアに依存しないプログラミングを可能にしています。Mac OS XのOpenGL APIは、マルチクライアント、マルチスレッド、マルチモニタを実現可能です。

・Mac OS XにおけるOpenGLの構造

Mac OS Xは異なるグラフィックカードを用いたマルチモニタをサポートしており、その場合、それぞれのモニタを異なるレンダラーが担当しています。レンダラーとそれが担当

するディスプレイセットの結合を仮想スクリーンと呼びます。この仕組みを実現するために、Mac OS XのOpenGLは、ウィンドウシステム階層、レンダラー階層、ドライバ階層の3つの階層に分割されており、その上位2階層はプラグインインターフェースで結合されています。以下の図は、Mac OS Xに実装されているOpenGLの構造です。



（1）ウィンドウシステム階層

ほとんどのアプリケーションは、ウィンドウシステム階層のみとコミュニケーションを行います。この階層にはGL（OpenGLライブラリ）、フルスクリーンやオフスクリーン対応のCore OpenGL（CGL）、Carbon用のApple OpenGLフレームワーク（AGL）、Cocoa用のOpenGLクラスなどが含まれています。これらのライブラリには、Mac OS Xのウィンドウシステムに関係する関数やメソッドがあり、ウィンドウシステム階層での仕事は、ピクセルフォーマットの選択、コンテキストや描画対象オブジェクトの作成と破棄、バッファのスワップなどです。AGLとCocoa OpenGLクラスは、CGLの上位に実装されています。

フルスクリーンやオフスクリーンを用いたとしても、ウィンドウベースのアプリケーションはCarbonならAGLを、CocoaならCocoa OpenGLクラスを用います。よりダイレクトにシステムの機能にアクセスしたい場合には、その下位階層のCGLを用います。また、Mac OS 9の場合にはAGLのみが利用可能となります。

（2）レンダラー階層

レンダラー階層では、ピクセルフォーマットの選択、コンテキストや描画対象オブジェクトの作成と破棄、バッファのスワップなどに対してレンダラーを制御するコマンドが集められています。この階層には、GLIプラグインインターフェースに準拠した複数のGLIプラグインレンダラーが存在します。各プラグインは、それぞれが特定のソフトウェアやハードウエァに対応しており、GLIプラグインインターフェースの仕組みは、他のプラットフォーム用のドライバをMac OS Xにインプリメントする作業を容易にします。

（a）レンダリング環境のサポート

Mac OS Xは、色々な機能を持ったグラフィックタカードをサポートしています。それは、ハードウエァによるアクセラレーション機能があってもなくてもかまいません。GLIプラグインインターフェースは、ひとつのハードウエァに対して複数のレンダラーを働かせることが可能です。レンダラープラグインは簡単に追加や削除ができ、OpenGLにより用途に合わせて最適なものが選ばれて使用されます。

アプリケーションが起動されると、CGLによりすべてのレンダラーがレジストされます。適切なアトリビュートを選択しピクセルフォーマットを作成すると、CGLがそれに対して最適なレンダラーを選択します。コンテキストが描画対象オブジェクトに割り当てられると、CGLは関連するレンダラーを読み込み、てコンテキストに対して実行されたすべての

OepnGLコマンドをそれに送るようになります。また、新しいコンテキストが選択されれば、CGLはそれに適したレンダラーを再度読み込み使用します。

このように、アプリケーション側はレンダラー選択に対して注意をはらう必要はありません。もし、特定なレンダラーを呼び出して処理させたい場合には、レンダラーID番号を渡すことで、その機能情報を入手できるAPIがCGLやAGLに用意されています。

（b）マルチモニタのサポート

Mac OS XのOpneGLには、マルチモニタをまたいでレンダリングを実行する機能があります。Mac OS Xのウィンドウシステムは、単独やいくつかのモニタを使用した複数の仮想スクリーンをサポートしています。ユーザは、異なる階調や解像度のモニタ間でも描画対象となっているウィンドウをドラッグして表示することが可能です。

例えば、ひとつのビデオカードに2つのディスプレイが接続されていても、仮想スクリーンはひとつです（Dual-Head）。また、まったく別のビデオカードにそれぞれディスプレイが接続されていれば、仮想スクリーンは2つとなります。仮想スクリーンに対して割り当てられるレンダラーはひとつです。これは、ピクセルフォーマットを作成した段階で、その情報を吟味してOpenGLが選択します。

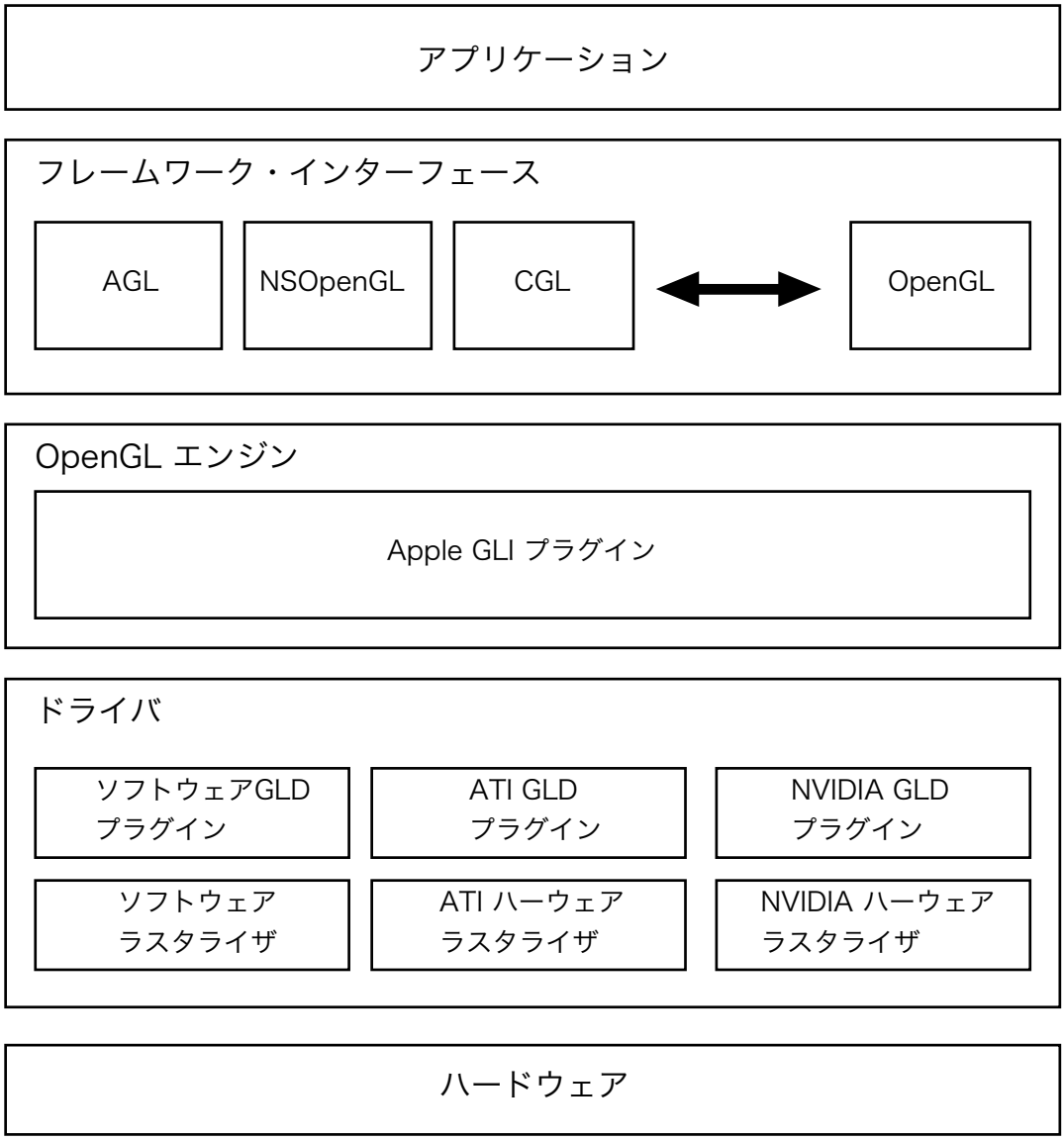
ウィンドウが2つの仮想スクリーンをまたいで表示されている場合、CGLは表示面積が大きい方の仮想スクリーンのレンダラーを用いて描画し、片方の仮想スクリーンの表示領域についてはバッファから画像をコピーすることで処理します。

（3）ドライバ階層

ドライバ階層では、ピクセルフォーマットの選択、コンテキストの作成と破棄、描画対象オブジェクト、バッファ、テクスチャの操作、バッファのスワップなどに対してハードウエァ特有のコマンドが集められています。この階層にはGDLプラグインインターフェースと複数のGDLプラグインが含まれています。ハードウエァ開発者は、独自のGDLプラグインを提供することが可能です。

・OpenGLのドライバモデル

Macintoshに実装されているOpenGLは、色々な階層を経由してアプリケーションと情報交換を行います。以下は、その階層を機能別に分けた図です。



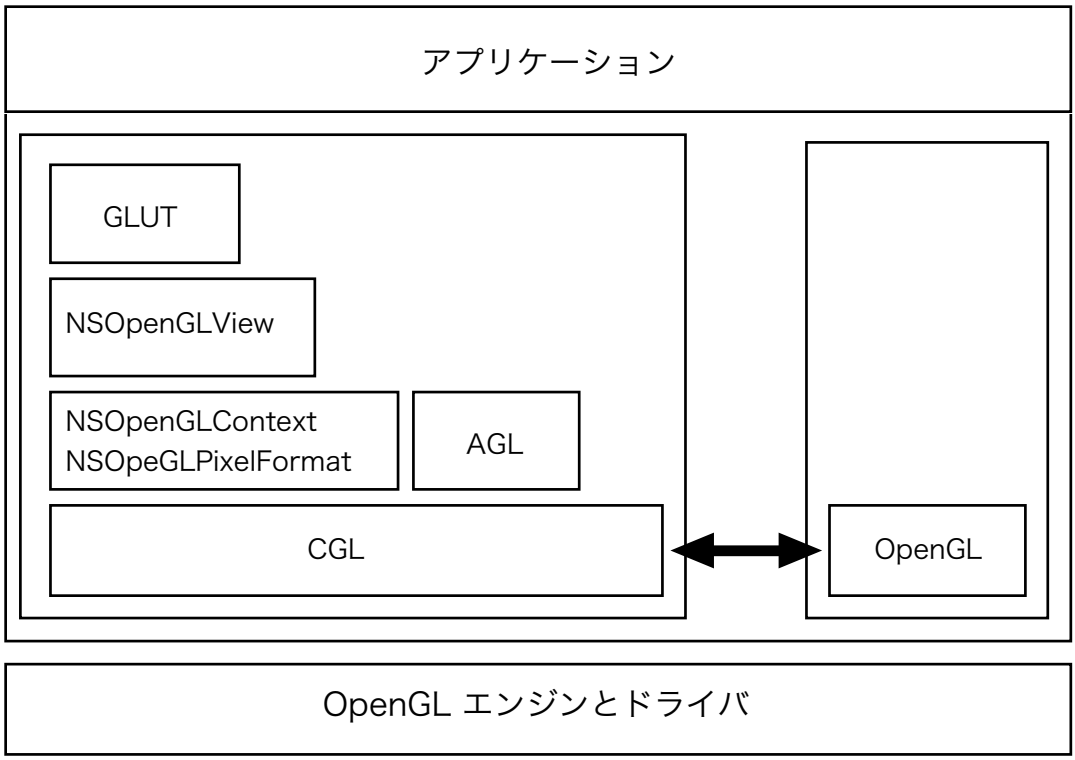
OpenGLのエンジンでは、複数アプリケーションから全リソースを利用可能です。つまり、複数のクライアントから同時に利用されることになります。Apple社のGLIプラグインは、すべての一般的なソフトウェアレンダリング用コードを含みます。ATIやNVIDIAのGLDプラグインは、それぞれのハードウエァにおいてラスタライズを実行するためのコードを含みます。OpenGLのグラフィック処理は特定のレンダラーにより制限を受けません。アプリケーション側では、ウィンドウが別モニタへドラッグされた時には、働くレンダラーが切り替わるケースがあることを考慮に入れておく必要があります。

・フレームワーク・インターフェース

Mac OS Xには、OpenGLの能力を利用するために幾つかのフレームワークが存在します。それを低い階層から並べると以下の様になります。

・Core OpenGL（CGL）：ハードウエァに一番近い階層です
・Apple OpenGL（AGL）：MacintoshのためのOpenGL拡張です（Carbon用）
・Cocoa OpenGLクラス：Cocoa用の様々なOpenGLクラスです。
・OpenGL Utility Toolkits（GLUT）：プラットフォーム非依存ライブラリです。

各フレームワークの関係は以下のようになります。



もしアプリケーションがフルスクリーン表示を利用しているならば、フレームワークとしてCGLを採用します。これにより、もっともパフォーマンスの良い表示能力を得ることが可能です。アプリケーションがウィンドウベースであれば、Carbon環境であればAGLを、Cocoa環境であればNSOpenGLContextクラスやNSOpenGLPixelFormatクラスを用います。Cocoaを用いた簡単なアプリケーションであれば、NSOpenGLViewクラスを利用するのが最善です。GLUTは、ウィンドウシステム（プラットフォーム）非依存型のユーザインターフェース（メニュー表示やマウスクリックイベント等）を提供します。

（1）CGLとAGL

Core OpenGL（CGL）では、Mac OS XのOpenGLにアクセスする基本的な方法を提供します。CGLはフルスクリーン描画のみに特化されており、いかなるウィンドウシステムとも連動しません。ただし、ピクセルバッファは利用可能です。CGLには、ピクセルフォーマット、コンテキスト、描画対象オブジェクトの管理や、グローバルパラメータの操作やレンダラーの機能情報を入手するためのAPIが用意されています（別ドキュメントのCGL Referenceを参照）。

AGLは、Carbon環境におけるOpenGLインターフェースであり、ウィンドウ、オフスクリーンバッファ、フルスクリーンバッファ、ピクセルバッファなどを描画対象オブジェクトとすることが可能です。AGLのAPIは、CGLのAPIにダイレクトに一対一で対応しています。また、CocoaのNSOpenGLContextやNSOpenGLPixelFormatといったメソッドもCGL APIに準拠しています。

・OpenGLライブラリ

OpenGL対応アプリケーションは、OpenG Llibrary（GL）とOpenGL Utility Llibrary（GLU）とOpenGL Utility Toolkit （GLUT）の3つのライブラリにアクセスすることが可能です。

GLは、グラフィカルオブジェクトを定義するためのローレベルなAPIを提供しています。これらのAPIは、OpenGLの仕様を満たすための基本的なコマンドであり、バーテックスとピクセルイメージ（画像）という2つの基本的なグラフィックプリミティブの操作を提供しています。そのため、GL APIのみでは複雑なグラフィカルオブジェクトは扱えません。以下がGLがサポートしている機能一覧です。

・ジオメトリック（幾何学）とラスタ（ピクセル）プリミティブ
・テクスチャマッピング
・ビューとモデリングの座標変換
・クリッピングとカーリング（隠面処理）
・アルファ合成
・大気シミュレーション（もや、霧、煙）
・質感とライティングの実行
・スムーズシェーディング
・アンチエイリアス
・アキュムレーションバッファ
・RGBAディスプレイモード
・ステンシル面

GLUライブラリは、GL APIを有効に用いることで実現したより先進的なグラフィック機能をサポートしています。GLUライブラリの機能には以下の通りです。

・複雑なポリゴンの作成と操作
・Bスプライン（NURB）カーブの処理
・イメージのスケール処理
・タイリング

GLUTライブラリは、利用されているウィンドウシステム環境に準拠したクロスプラットフォーム用のAPIを提供します。ウィンドウへの表示や書き換え、メニュー表示やマウスクリックイベントの取り扱いなども提供しています。

例えば、CarbonアプリケーションからOpenGLの各ライブラリを利用するには、Xcodeのプロジェクトに OpenGL、AGL、GLUTの3つのフレームワークを登録しておき、ソースファイルの先頭で以下のヘッダファイルを呼び出すようにします。ちなみに、先んじてQuickTime.hを定義していれば、Carbon.hとOpenGL.hの定義は必要ありません。

```
#include <Carbon/Carbon.h>
#include <OpenGL/OpenGL.h>  // GL/GLU/CGL関連のAPI
#include <AGL/agl.h>        // AGL関連にAPI
#include <GLUT/glut.h>      // GLUT関連のAPI
```

・MacintoshにおけるOpenGLの機能

Mac OS XにおけるOpenGLには、そのオリジナル機能を補助もしくは拡張するために、Macintosh以外のプラットフォームでは利用できない幾つかの独自の特徴が追加されています。以下で、その特徴について解説します。

（1）仮想スクリーン

仮想スクリーンはMac OS Xに依存したOpenGLの機能拡張であり、OpenGLが画像描画を行うためのハードウェア、レンダラー、ピクセルフォーマットのコンビネーションです。仮想スクリーンを切り替えれば、通常レンダラーも切り替わります。

Mac OS X環境においては、1つのビデオカードでマルチモニタ（Dual-Head）を利用している時、その2つをまとめてひとつのスクリーンとして扱うことが可能です。片方のスクリーンから他方のスクリーンへウィンドウを移動しても、OpenGLによる描画が正しく

行われるのは、両方をまとめてひとつの仮想スクリーンとして管理しているためです。

OpenGLは、アプリケーションから指定されたピクセルアトリビュートを調べ、現在のシステムで利用可能な仮想スクリーンのリストを発生します。

（2）ピクセルアトリビュート

ピクセルアトリビュートは、Mac OS Xに依存したもうひとつのOpenGLの機能拡張であり、アプリケーションがピクセルフォーマットを作成する時に指定します。アトリビュートには2種類があり、そのひとつのレンダラーアトリビュートでは、レンダラーの能力、タイプ、ベンダーやモデルなどを指示可能で、もう片方のバッファアトリビュートでは、デプス（階調）などのサーフェス関連の定義を指示できます。こうしたアトリビュートの設定は、アプリケーションにおけるレンダリングのパフォーマンスや、どんな種類のイメージングを実行するのかに深く関係しています。

（3）ピクセルバッファ

ピクセルバッファ（PBBuffer）はOpenGLの描画対象オブジェクトであり、テクスチャバッファとして利用することが可能なオフスクリーンメモリ領域です。アプリケーションはPBBufferに対して画像を描画し、それを別バッファへコピーすることなく3Dオブジェクトなどのテクスチャとして利用できます。この時のPBBufferに対する描画では、ハードウェアアクセラレーションが有効となります。

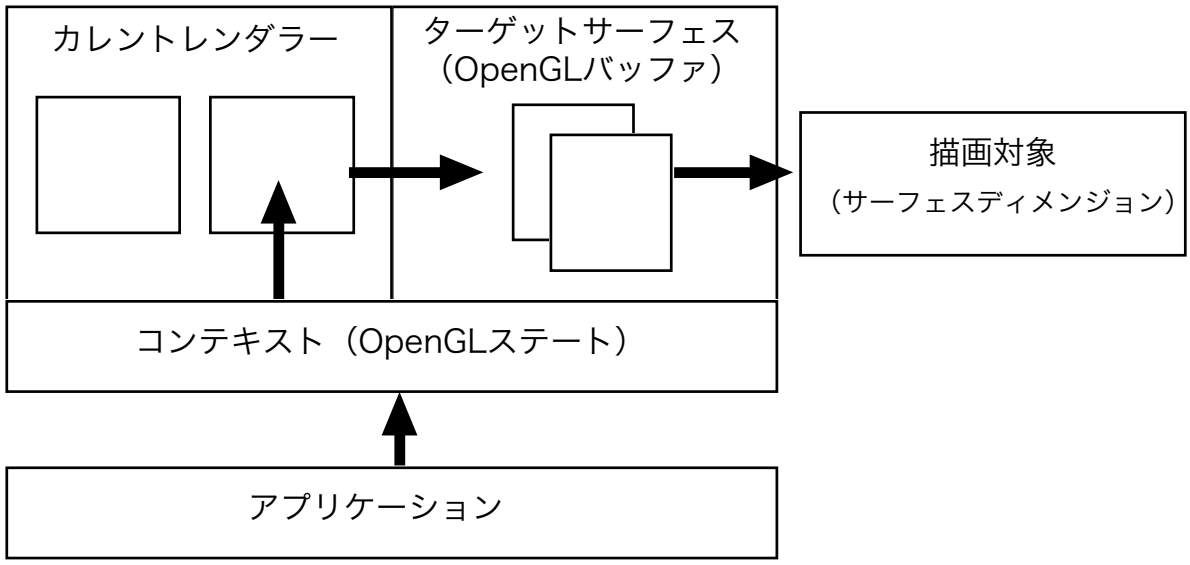
（4）コンテキスト

コンテキストは、アプリケーションのために、レンダリング先のOpenGLの各種状態（Mac OS Xに依存）を保持します。一度ピクセルフォーマットが作られると、アプリケーションはコンテキストを作成することが可能となります。この時、仮想スクリーンやピクセルバッファなど、すべてのOpneGLに関する状態情報などがコンテキストに移されます。その後、アプリケーションが実行したすべてのOpenGLコマンドは、カレントコンテキストに送られることになります。

（5）描画対象

描画対象とは、OpenGLの描画処理のターゲットを指します。描画対象の種類としてはは、ビュー（View）、ウィンドウ、ピクセルバッファ、オフスクリーンメモリ領域、フルスクリーンバッファなどがあります。描画対象をコンテキストに関連づけることで、OpenGLは自動で必要なバッファを割り当て適切なレンダラーを用意します。

アプリケーションから描画対象への物理的なデータの流れは以下のようになります。



（6）スレッディング

Mac OS Xはマルチスレッディングをサポートしています。OpenGL対応のアプリケーションでもマルチスレッディングを利用することは可能ですが、カレントコンテキストを保持する点に注意してください。スレッドを切り換える時にコカレントコンテキストも切り換えると、予期せぬ結果を招く場合があります。ゆえに、Apple社は、シングルスレッド・アプリケーションのみでOpenGLコマンドを実行することを推薦しています。

（7）OpenGLシェーディング・ランゲージ

MacintoshのOpenGLは、拡張機能として「ARB_shading_language_100」をサポートしており、ハードウエァアクセラレーションを利用した画像エフェクトなどを実現することができます。シェーディング・ランゲージについは、書籍として出版されている「OpenGL Shading Language」を別途参照してください。

（4）CによるOpenGLプログラミング

この章では、Carbon環境におけるOpenGLプログラミングとその基本的なテクニックを紹介します。プログラミングにはCを用います。

・OpenGLの機能を見つける

アプリケーションによっては、実際にOpenGLコマンドを実行する前に、そのバージョン番号を確認し、利用環境が目的に適合しているかどうかを確認する必要があります。

```
// バージョンのチェック：
    strVersion = glGetString (GL_VERSION);

// 拡張機能がサポートされているかどうかの確認：
    strExt = glGetString (GL_EXTENSIONS);
    fFence = gluCheckExtension ("GL_APPLE_fence", strExt);
    fShade = gluCheckExtension ("GL_ARB_shading_language_100", strExt);

// OpenGL機能の限界値を明らかにする：
    glGetIntegerv (GL_MAX_TEXTURE_SIZE, &maxTextureSize);
```

・イメージをテクスチャへ変換する

MacintoshのOpenGLでは、イメージ（画像）をテクスチャへ変換するために色々な方法が提供されています。例えば、CFURLRefを使えば、テクスチャを直接URL経由で得ることも可能です。また、テクスチャとしては、他の整数画像フォーマットと同様に浮動小数点フォーマットも利用でき、RGBフォーマットを経由することなく、こうした画像を浮動小数点フォーマットのテクスチャへ変換することも可能です。

・コンテキストの共有

MacintoshのOpenGLではリソースなどの共有が可能です。共有可能なオブジェクトとしては、テクスチャオブジェクト、バーテックスプログラム、ディスプレイリスト、バーテックスアレイオブジェクト、バッファオブジェクトなどがあります。ただし、コンテキストの共有は一般的な状態（カレントカラー、テクスチャ、座標系など）などは含まれま

せん。詳細な情報は「Apple Technical Q&A 1248」を参照してください。

```
GLint attrib[] = {AGL_RGBA, AGL_DOUBLEBUFFER,  AGL_FULL_SCREEN,
                  AGL_NONE};
GLint attrib2[] = {AGL_RGBA, AGL_DOUBLEBUFFER, AGL_NONE};

disp = GetMainDevice ();  // メインディバイスを得る
aglPixFmt=aglChoosePixelFormat (&disp, 1, attrib); // ピクセルフォーマットを得る
aglContext = aglCreateContext (aglPixFmt, NULL);  // コンテキストを作成する

// 同じディスプレイを利用する
aglPixFmt2 = aglChoosePixelFormat (&disp, 1, attrib2);
aglContext2 = aglCreateContext (aglPixFmt, aglContext);  //コンテキストの共有
```

・ピクセルバッファの使用

ピクセルバッファでは、OpenGLの機能拡張「GL_APPLE_pixel_buffe」により、ハードウェアアクセラレーションが有効となったオフスクリーンレンダリングが実行されます。ピクセルバッファを描画対象として選択するには、CGLのCGLSetPBuffer()を利用します。そして描画ターゲットとなるコンテキストにおいて実際のテクスチャとしてバインドするにはCGLTexImagePBuffer()を実行します。

・ピクセルフォーマットについて

CGLのピクセルフォーマット作成では、各種バッファとコンテキストのアトリビュートを指示します。カラーバッファ、アルファバッファ、デプスバッファ、ステンシルバッファ、補助バッファ、アキュムレータバッファのプロパティや、その他レンダラーに関する仕様はピクセルフォーマット作成時に渡すアトリビュート配列で決定されます。ピクセルフォーマットはコンテキストを作成する時に用いられ、その後破棄します。

ピクセルフォーマットのためのアトリビュートには、単独でその機能のON/OFFを指示するタイプと、後ろに何らかのパラメータを必要とするタイプの2種類があります。またアトリビュート配列の最後には、項目リストのターミネータとしてNULLを代入する必要があります。ピクセルフォーマットの作成にはCGLChoosePixelFormat()を使用します。もし正常なピクセルフォーマットが作成できなかった場合には、オブジェクトにNULLが代入されて返ります。

以下はフルスクリーンと1bitのステンシルバッファを指示したアトリビュート配列です。

```
CGLPixelFormatAttribute attribs[] =
{
    kCGLPFAFullScreen,
    kCGLPFAStencilSize, 1,
    0
} ;
```

このアトリビュート配列を渡して、ピクセルフォーマットにNULLが返された場合には、ステンシルバッファがサポート不可であることを意味しますので、以下のように配列を変更して再トライしてみます。

```
GLPixelFormatAttribute attribs[] =
{
    kCGLPFAFullScreen,
    0
} ;
```

こうした処理を連続して実行するようなソースコードを記述してみしょう。

```
CGLPixelFormatAttribute attribs[] =
{
    kCGLPFAFullScreen,
    kCGLPFAStencilSize, 1,
    0
} ;

CGLPixelFormatObj pixelFormatObj ;
long numPixelFormats ;
long value ;

CGLChoosePixelFormat( attribs, &pixelFormatObj, &numPixelFormats ) ;
                                    // アトリビュートよりピクセルフォーマットを作成
if( pixelFormatObj == NULL ) {
    // ステンシルバッファは確保できないのでフルスクリーンのみに変更
    attribs[1] = NULL;
    CGLChoosePixelFormat( attribs, &pixelFormatObj, &numPixelFormats ) ;
}                                   // 再度ピクセルフォーマットを作成
```

```
if( pixelFormatObj == NULL ) {
    // フルスクリーンも利用できないのですべての処理は中止する
}

// ステンシルバッファを使えるかどうかの情報をピクセルフォーマットから得る
CGLDescribePixelFormat( pixelFormatObj, 0, kCGLPFAStencilSize, &value ) ;
if( value == 0 ) {
    // ステンシルバッファが利用できない場合に対応する
}
```

CGLDescribePixelFormat()により、ピクセルフォーマットのアトリビュートの状態を調べてから次に続く処理を切り分けます。CGLにおいて最低限設定しなければいけないアトリビュートは、kCGLPFAFullScreenかkCGLPFAOffScreenです。

・フルスクリーンコンテキストの作成

フルスクリーンOpenGLコンテキストを作成する時には、最初にCore Graphics Direct Display APIを利用して、対象となるディスプレイをキャプチャ（占有）します。一度ディスプレイをキャプチャすると、そのディスプレイは他のアプリケーションやサービスからの処理を受け付けなくなります。他のアプリケーションは、ディスプレイの状態が変化したことについて何も知りませんし、デスクトップ上のウィンドウやアイコンの移動も不可能です。Core Graphics Direct Display APIをについては、 CGDirectDisplay.hを参照してください。

1. メインディスプレイをキャプチャ（占有）する：

```
CGDisplayCapture(kCGDirectMainDisplay);
```

2. ピクセルフォーマットを選択する：

```
CGOpenGLDisplayMask displayMask =
        CGDisplayIDToOpenGLDisplayMask( kCGDirectMainDisplay ) ;

CGLPixelFormatAttribute attribs[] =
{
    kCGLPFAFullScreen,
    kCGLPFADisplayMask,
    displayMask,
    NULL
```

} ;

CGLPixelFormatObj pixelFormatObj ;
long numPixelFormats ;

CGLChoosePixelFormat(attribs, &pixelFormatObj, &numPixelFormats);

3. フルスクリーンコンテキストをセットアップする：

CGLContextObj contextObj ;
CGLCreateContext(pixelFormatObj, NULL, &contextObj) ;

CGLDestroyPixelFormat(pixelFormatObj) ;

CGLSetCurrentContext(contextObj) ;
CGLSetFullScreen(contextObj) ;

4.アプリケーションのメインループを開始する：

5. コンテキストを破棄する：

CGLSetCurrentContext(NULL) ;
CGLClearDrawable(contextObj) ;
CGLDestroyContext(contextObj) ;

6. キャプチャ（占有）したディスプレイをリリースする：

CGReleaseAllDisplays();

幾つかのOpenGLレンダラー（ソフトウェアレンダラーなど）はフルスクリーンモードに対応していません。もし、CGLChoosePixelFormat()がピクセルフォーマットオブジェクトとしてNULLを返したら、フルスクリーンコンテキストは作成できなかったことを意味しています。

（1）ディスプレイモードの調整

スクリーンの解像度やリフレッシュレイト（周波数）を変更するために、Direct Displayは2つのルーチンを提供しています。ひとつは解像度と階調のみを設定し、もうひとつは解像度、階調、リフレッシュレイトを変更します。これらのルーチンは、対応するディス

プレイの物理制限のため、指定された状態に正確に切り替わったかどうかは保証されまっせん。また、アプリケーション終了時にディスプレイの設定を復帰させる必要はありません。Direct Displayは、この処理を自動的に行います。

Direct Displayには、現在のディスプレイの設定情報を得るためのルーチンもあります。また、CGLSetFullScreen()はOpenGLのビューポートをディスプレイの矩形枠に設定し直します。

1. ディスプレイをキャプチャ（占有）し32bit階調で1024x768サイズに設定する

```
CGDisplayCapture( kCGDirectMainDisplay ) ;
CGDisplaySwitchToMode( kCGDirectMainDisplay,
    CGDisplayBestModeForParameters( kCGDirectMainDisplay,
        32, 1024, 768, 0 ) ) ;
```

（2）画面書き換え（Vertical Blanking Loop）との同期

アプリケーションのスクリーンアップデートは、表示の荒れを防ぐために、画面書き換え周波数（VBL）に同期させるべきです。これを達成するには、ダブルバッファとそれの切り換えによる表示方法を利用します。以下は、「フルスクリーンコンテキストの作成」のソースコードの次なるステップで、アトリビュートにkCGLPFADoubleBufferを加えた場合の処理を示しています。

```
CGLContextObj contextObj ;
long swapInterval ;

CGLCreateContext( pixelFormatObj, NULL, &contextObj ) ;
CGLDestroyPixelFormat( pixelFormatObj ) ;

swapInterval = 1 ;
CGLSetParameter( contextObj, kCGLCPSwapInterval, &swapInterval ) ;
                                            // 描画をVBLと同期するように指示
CGLSetCurrentContext( contextObj ) ;
CGLSetFullScreen( contextObj ) ;
```

この処理により、CGLFlushDrawable(contextObj)を実行すると、同期を取ってバックバッファからフロントバッファへの画像コピーが実行されるようになります。この件に関する詳細については、テクニカルノートのTN2093「UnderstandingVSYNCH」を参照してみてください。

・オフスクリーンコンテキストの作成

CGLでオフスクリーンコンテキストを作成する方法は、フルスクリーンコンテキストの場合とほぼ同じです。以下のサンプルでは、32bit 1024x786サイズのオフスクリーンコンテキストを作成しています。

1. ピクセルフォーマットを選択する：

```
CGLPixelFormatAttribute attribs[] =
{
    kCGLPFAOffScreen,
    kCGLPFAColorSize, 32,
    NULL
} ;

CGLPixelFormatObj pixelFormatObj ;
long numPixelFormats ;

CGLChoosePixelFormat( attribs, &pixelFormatObj, &numPixelFormats ) ;
```

2. コンテキストをセットアップする：

```
CGLContextObj contextObj ;
CGLCreateContext( pixelFormatObj, NULL, &contextObj ) ;
CGLDestroyPixelFormat( pixelFormatObj ) ;
CGLSetCurrentContext( contextObj ) ;
```

3. バッファ用のメモリ領域を確保する：

```
void* memBuffer = ( void* ) malloc( 1024 * 768 * 32 / 8 ) ;
```

4. バッファにコンテキストを割り当てる：

```
CGLSetOffScreen( contextObj, 1024, 768, 1024 * 4, memBuffer ) ;
```

5. アプリケーションのメインループを開始する：

6. コンテキストを破棄する：

```
CGLSetCurrentContext( NULL ) ;
CGLClearDrawable( contextObj ) ;
CGLDestroyContext( contextObj ) ;
```

オフスクリーンコンテキストを作成した後に、OpenGLコマンドを使いglFinish()を呼ぶと、memBufferとして確保されているメモリ領域にレンダリングが実行されます。現状、CGLSetOffscreen()経由のレンダリンについてはハードウエァアクセラレーションは有効になりません。もし、AGLやCocoaのOpneGLクラスで、ハードウエァアクセラレーションが有効のオフスクリーン描画を実行したい場合には、描画対象として隠した（Hideさせた）ウィンドウをコンテキストに割り当て、レンダリング後のバッファデータをglReadPixels()で読み込むようにします。

・マルチサンプル・ピクセルフォーマットの利用

CGLは、ハードウエァを使い低解像度のビジュアルの画質を格段に向上することができるピクセルフォーマットオプションも提供します。フルスクリーン・アンチエイリアシング（FSAA）には多くのビデオカードが対応しており、それが利用可能かどうかは、拡張機能のGL_ARB_multisampleを探すことで判断できます。もし利用可能であれば、ピクセルフォーマットのアトリビュートにkCGLPFASampleBuffersとkCGLPFASamplesを加えることで、マルチサンプルレンダリングが可能となります。

最初のkCGLPFASampleBuffersアトリビュートでは、一度に何枚のマルチサンプルバッファを確保するのかを指定します。現在の拡張機能の仕様では、利用できるルチサンプルバッファはひとつだけです。

次のkCGLPFASamplesアトリビュートでは、1ピクセルについていくつのサンプルを要求するのかを指示します。この数値の決定は、画質、メモリ消費量、処理スピードなどに影響します。通常は2か4に設定しますが、最大値はCGLDescribePixelFormat()により調べることが可能です。処理スピードと消費メモリを重視するなら2を、画質重視であれば4を設定してください。1ピクセルにおけるサンプル数を大きくするとメモリ消費量が増え、メモリ不足が発生するようだとバッファの作成は失敗します。

以下のサンプルは、1ピクセルあたり4サンプルを指定したピクセルフォーマット・アトリビュートです。

```
CGLPixelFormatAttribute attribs[] =
{
    kCGLPFAFullScreen,
```

```
    kCGLPFASampleBuffers,
    1,
    kCGLPFASamples,
    4,
    NULL
} ;
```

GL_ARB_multisampleなどのフルスクリーンのアンチエイリアシングに関する機能拡張については、以下のURLを別途参照してください。特定のハードウエァでは、マルチサンプルフィルタの処理速度や画質を向上するためのGL_NV_multisample_filter_hint拡張機能がサポートされています。

「OpenGL extensions registry」http://oss.sgi.com/projects/ogl-sample/registry/

・OpenGLマクロの利用

OpenGLマクロは、コマンド呼び出しのオーバーヘッドを解消します。もし、アプリケーションのソースコードの中で相当量のOpneGLコマンドが呼び出されている場合には、このマクロを使うことで、処理パフォーマンスを上げることができます。

CGLとCocoaでマクロを使う場合には、CGLマクロのヘッダファイル（cglMacro.h）を、AGLとCarbonの場合にはAGLマクロのヘッダファイル（aglMacro.h）をインクルードし、ローカル変数としてcgl_ctxもしくはagl_ctxを定義しておきます。

以下は、Carbon環境でAGLマクロを利用する場合のサンプルです。

```
#include <AGL/aglMacro.h>            // ヘッダファイルをインクルード

AGLContext agl_ctx = myContext;   // カレントコンテキストのセット
glBegin (GL_QUADS);                    // コマンドとしてマクロが使用される
                                                 // ここでプリミティブの描画 を行う
glEnd ();
```

・Direct Display APIについて

Direct Display APIを用いれば、メインスクリーンを占有して独占的にアクセスできるようになり、その階調、解像度、リフレッシュレイトなども変更可能です。加えて、ディスプレイ情報の入手、マウス操作、スクリーンのフェードインやフェードアウト、カラーパ

レットの再構築、スクリーンピクセルやディスプレイのビームポジションへのアクセスなど、数多くの機能が提供されています。加えて、「Mac OS Core Graphics's Remote Operation API」はマウス操作に関する沿革操作機能を提供します。この章では、Direct Displayに関する様々な機能を、サンプルソースコードと共に紹介します。

（1）ディスプレイとディスプレイモード

Direct Display APIであるCGGetActiveDisplayList()、CGDisplaySwitchToMode()、CGDisplayBestModeForParameters()などは、Core Foundationフレームワークで定義されているCFArrayRefやCFDictionaryRefを引数として受け取ります。これらに関しての詳細については、CFArray.hやCFDictionary.hを別途参照してください。

すべてのディスプレイIDを含んだ配列は、CGGetActiveDisplayList()で得ることが出来ます。配列中から目的のディスプレイIDを見つけるには、CGGetDisplaysWithPoint()、CGGetDisplaysWithRect()、rCGGetDisplaysWithOpenGLDisplayMask()などを利用します。また、CGGetDisplaysWithPoint()やCGGetDisplaysWithRect()については、目的のディスプレイを選択するためのユーザインターフェースを提供しています。各ディスプレイがサポートしているモードの配列は、CGDisplayAvailableModes()で得ることが可能です。

以下のサンプルコードは、リストの最後のディスプレイをキャプチャ（占有）し、そのディスプレイがサポートしている最初のディスプレイモードへと切り換えます。

```
#define MAX_DISPLAYS 32

CGDirectDisplayID lastDisplay, displayArray[MAX_DISPLAYS] ;
CGDisplayCount numDisplays ;

CFArrayRef displayModeArray ;
CFDictionaryRef displayMode ;

CFNumberRef number ;
long height, width ;

// ディスプレイの配列を得て最後のディスプレイをキャプチャ（占有）する
CGGetActiveDisplayList( MAX_DISPLAYS, displayArray, &numDisplays ) ;
lastDisplay = displayArray[ numDisplays - 1 ] ;

CGDisplayCapture( lastDisplay ) ;
```

```
// ディスプレイモードの配列を得て最初のモードを採用する
displayModeArray = CGDisplayAvailableModes( lastDisplay ) ;
displayMode = (CFDictionaryRef) CFArrayGetValueAtIndex(
                                                    displayModeArray, 0 ) ;

// ディスプレイの幅と高さを得てプリントする
number = CFDictionaryGetValue( displayMode, kCGDisplayWidth ) ;
CFNumberGetValue( number, kCFNumberLongType, &width ) ;

number = CFDictionaryGetValue( displayMode, kCGDisplayHeight ) ;
CFNumberGetValue( number, kCFNumberLongType, &height ) ;

printf( "Switching to display mode with width: %ld and height: %ld \n",
                                                          width, height ) ;

// 新しいディスプレイモードに切り換える
CGDisplaySwitchToMode( lastDisplay, displayMode ) ;

// イベントループを開始する

// キャプチャ（占有）したディスプレイを開放する
CGReleaseAllDisplays();
```

もしアプリケーションが正確な階調と解像度を必要とするなら、適切なパラメータをCGDisplayBestModeForParameters()に渡すことで、適切なモード情報を得ることが出来ます。また、CGDisplayBestModeForParametersAndRefreshRate()では、追加パラメータとしてリフレッシュレートが用いられます。両APIとも、もし適切なモードがなければ現在のモードが返されますので、CGDisplayCurrentMode()で得たモードと比較して確認できます。またルーチンからはBoolean値も返り、もしそれがTRUEであれば要求されたディスプレイモードに完全に一致したことを意味します。

以下のサンプルでは、メインディスプレイを640x480、32bit、60Hzに切り換えています。また、返されたモードの解像度とリフレッシュレートはプリントされます。

```
CFDictionaryRef displayMode ;
CFNumberRef number ;
long height, width, depth, freq ;
boolean_t exactMatch ;
```

```
// メインディスプレイをキャプチャ（占有）する
CGDisplayCapture( kCGDirectMainDisplay ) ;

// ディスプレイモードとして640x480、32bit、60Hzを要求
displayMode =
    CGDisplayBestModeForParametersAndRefreshRate( kCGDirectMainDisplay,
        32,
        640, 480,
        60,
        &exactMatch ) ;

// もし現在のディスプレイモードが返ると要求モードは存在しない
// この場合、CGDisplaySwitchToMode()は何も実行しない
if( displayMode != CGDisplayCurrentMode( kCGDirectMainDisplay ) ) {
    // 要求を満たすディスプレイモードに対応可能なのかどうか？
    if( exactMatch ) printf( "Direct Display found an exact match.\n" ) ;

    // 解像度と階調とリフレッシュレートをプリント
    number = CFDictionaryGetValue( displayMode, kCGDisplayHeight ) ;
    CFNumberGetValue( number, kCFNumberLongType, &height ) ;
    number = CFDictionaryGetValue( displayMode, kCGDisplayWidth ) ;
    CFNumberGetValue( number, kCFNumberLongType, &width ) ;
    number = CFDictionaryGetValue( displayMode, kCGDisplayBitsPerPixel ) ;
    CFNumberGetValue( number, kCFNumberLongType, &depth ) ;
    number = CFDictionaryGetValue( displayMode, kCGDisplayRefreshRate ) ;
    CFNumberGetValue( number, kCFNumberLongType, &freq ) ;
    printf( "Switching to display mode with width: %ld, height: %ld, bpp: %ld,
                            and refresh rate: %ld \n", width, height, depth, freq ) ;
}
else {
    printf( "Direct Display could not find a suitable mode. \n" ) ;
}

// 新しいディスプレイモードに切り換える
CGDisplaySwitchToMode( kCGDirectMainDisplay, displayMode ) ;

// イベントループを開始する
```

```
// キャプチャ（占有）したディスプレイを開放する
CGReleaseAllDisplays() ;
```

Direct Displayには、ディスプレイモードのプロパティを簡単に入手するためのアクセッサが用意されています。このため、プロパティを得るためにCFDictionaryGetValue()をいちいち呼ぶ必要はありません。以下のサンプルソースコードでは、全てのディスプレイ（最大は32まで）の現在のモードにおける全プロパティをプリントします。

```
#define MAX_DISPLAYS 32

CGDirectDisplayID displayArray[MAX_DISPLAYS] ;
CGDisplayCount numDisplays ;

CFNumberRef number ;
long height, width, refresh, mode, bpp, bps, spp, rowBytes, gui, ioflags ;
int i ;

// ディスプレリストの配列を得て、その個数をプリントする
CGGetActiveDisplayList( MAX_DISPLAYS, displayArray, &numDisplays ) ;
printf( "Displays installed: %d\n", numDisplays ) ;

// ディスプレイの現在のモードのプロパティをプリントする
for(i = 0; i < numDisplays; i++) {
    width = CGDisplayPixelsWide( displayArray[i] ) ;
    height = CGDisplayPixelsHigh( displayArray[i] ) ;
    bpp = CGDisplayBitsPerPixel( displayArray[i] ) ;
    bps = CGDisplayBitsPerSample( displayArray[i] ) ;
    spp = CGDisplaySamplesPerPixel( displayArray[i] ) ;
    rowBytes = CGDisplayBytesPerRow( displayArray[i] ) ;
    number = CFDictionaryGetValue( CGDisplayCurrentMode( displayArray[i] ),
            kCGDisplayMode ) ;
    CFNumberGetValue( number, kCFNumberLongType, &mode ) ;
    number = CFDictionaryGetValue( CGDisplayCurrentMode( displayArray[i] ),
            kCGDisplayRefreshRate ) ;
    CFNumberGetValue( number, kCFNumberLongType, &refresh ) ;
    number = CFDictionaryGetValue( CGDisplayCurrentMode( displayArray[i] ),
            kCGDisplayModeUsableForDesktopGUI ) ;
    CFNumberGetValue( number, kCFNumberLongType, &gui ) ;
    number = CFDictionaryGetValue( CGDisplayCurrentMode( displayArray[i] ),
```

```
                kCGDisplayIOFlags ) ;
    CFNumberGetValue( number, kCFNumberLongType, &ioflags ) ;
    printf("\n--\n") ;
    printf("Current mode (%ld) of display #%d\n", mode, i) ;
    printf("\tResolution width: %ld height: %ld\n", width, height) ;
    printf("\tRefresh rate: %ld\n", refresh) ;
    printf("\tBits per pixel: %ld\n", bpp) ;
    printf("\tBits per sample: %ld Samples per pixel: %ld\n", bps, spp) ;
    printf("\tBytes per row: %ld\n", rowBytes) ;
    if( gui )
        printf("\tMode usable for a desktop GUI: Yes\n") ;
    else
        printf("\tMode usable for a desktop GUI: No\n") ;
    printf("\tIOFlags: %ld\n", ioflags) ;
}
```

（2）フェードインとフェードアウト

より滑らかに通常表示からフルスクリーンへ切り換えるためには、一度スクリーンを黒にフォードアウトさせてからモードを切り換え、その後フォードインして表示を復帰させる方法を取ります。こうした用途のために、Direct Displayでは、ディスプレイのガンマ値を変更する機能が提供されています。この場合、各カラーコンポーネントのガンマを0.0から1.0までの値として指示します。

```
double fadeValue ;
CGGammaValueredMin, redMax, redGamma,
            greenMin, greenMax, greenGamma,
            blueMin, blueMax, blueGamma ;

// 現在のガンマ値をユーザのColorSync設定として保存する
CGGetDisplayTransferByFormula( kCGDirectMainDisplay,
            &redMin, &redMax, &redGamma,
            &greenMin, &greenMax, &greenGamma,
            &blueMin, &blueMax, &blueGamma ) ;

// オリジナルガンマ値から黒へフェードアウトする
for( fadeValue = 1.0; fadeValue >= 0.0; fadeValue -= 0.01 ) {
    CGSetDisplayTransferByFormula( kCGDirectMainDisplay,
            redMin, fadeValue*redMax, redGamma,
```

```
            greenMin, fadeValue*greenMax, greenGamma,
            blueMin, fadeValue*blueMax, blueGamma ) ;
}

// ディスプレイモードを変更しCGLフルスクリーンコンテキストを設定する
// イベントループを開始する

// 黒い背景からオリジナルガンマ値へフェードインする
for( fadeValue = 0.0; fadeValue <= 1.0; fadeValue += 0.01 ) {
    CGSetDisplayTransferByFormula( kCGDirectMainDisplay,
            redMin, fadeValue*redMax, redGamma,
            greenMin, fadeValue*greenMax, greenGamma,
            blueMin, fadeValue*blueMax, blueGamma ) ;
}

// ユーザのColorSync設定を復帰させる
CGDisplayRestoreColorSyncSettings() ;
```

上記コードでは滑らかなフェードを実現するためにループを用いていますがて、フェード間隔を機種依存させないより正確かな方法としてタイマーを使うことをお勧めします。

（3）カーソルとマウスコントロール

通常、ゲームなどフルスクリーンを用いるアプリケーションでのマウスコントロールは一般的なアプリケーションのそれとは大きく異なります。 Direct DisplayとRemote Operationでは、そうした要求に対応する様々なサービスを提供しています。

マウスカーソルの消去と再表示には、Direct Display APIのCGDisplayHideCursor()とCGDisplayShowCursor()を使います。これらのAPIの引数はディスプレイIDのみです。

CGGetLastMouseDelta()では、マウスのX方向とY方向の最後の移動距離を得ることができます。CGAssociateMouseAndMouseCursorPosition()では、マウスの動きとマウスカーソルの表示位置を連動させるかどうかを選択できます。

CGDisplayMoveCursorToPoint()では、マウスカーソルをディスプレイIDで指示されたディスプレイの指定位置へ移動させます。もし、ディスプレイ外へ移動させたい場合には、マウスカーソルを消去してから実行してください。

以下のサンプルソースコードは、メインディスプレイ上のマウスカーソルを消去し、マウスの移動とカーソル位置の連動を停止させまることで、マウスドラッグを別の処理に利用できるようにしています。

1. マウスカーソルを消去してからメインディスプレイの原点へポイントを移動させる：

```
CGDisplayHideCursor( kCGDirectMainDisplay ) ;
CGDisplayMoveCursorToPoint( kCGDirectMainDisplay, CGPointZero ) ;
```

2.マウスの移動とマウスカーソル位置の連動を停止する：

```
CGAssociateMouseAndMouseCursorPosition( FALSE ) ;
```

3. メインループで実行することでマウスの移動距離を得る ：

```
CGMouseDelta dx, dy ;
CGGetLastMouseDelta( &dx, &dy ) ;
```

4. マウスの移動とマウスカーソル位置の連動を再開する：

```
CGAssociateMouseAndMouseCursorPosition( TRUE ) ;
```

5. マウスカーソルを表示する：

```
CGDisplayShowCursor( kCGDirectMainDisplay ) ;
```

（5）CocoaによるOpenGLプログラミング

この章では、CocoaのOpenGLクラスを紹介し、「Green Triangle」と命名したサンプルアプリケーションを作成してみます。最初はウィンドウに緑色の三角形のみを描画してみます。その後、マウスドラッグ処理、フェードインとアウトのアニメーション、ダブルバッファの使用など、順次機能を拡張していきます。

「Green Triangle」アプリケーションを開発する手順は以下の通りです。

1. Xcodeで新規プロジェクトを作る。
2. インターフェースとCocoa OpenGL Viewクラスのサブクラスを作る。
3. サブクラスに描画メソッドを実装しOpenGLにより緑の三角形を描画する。
4. サブクラスにマウスイベント処理メソッドを実装する。
5. サブクラスのイニシャライザにタイマーを作成し周期的に色変更するメソッドを実装
6. ダブルバッファアトリビュートを設定したピクセルフォーマットを作成する

・「Green Triangle」のインターフェースを作る

まず最初に「Green Triangle」のインターフェースを作成します。

1. Xcodeでアプリケーション用のプロジェクトを新規作成する。

2. nibファイルをオープンする 。

3. アプリケーションのウィンドウをカスタマイズする。

4. Cocoa NSOpenGLViewクラスのサブクラスを作成する 。

5. ウィンドウにそのサブクラスを追加する 。

（1）アプリケーションのプロジェクトを作成

まず最初に「Green Triangle」のインターフェースを作成します。

1. Xcodeを起動する。

2. ファイルメニューから「新規プロジェクト...」を選択する。

3. 新規プロジェクトパネルから「Cocoa Application」を選び「次へ」をクリック。

4. プロジェクト名を「Green Triangle」とする。

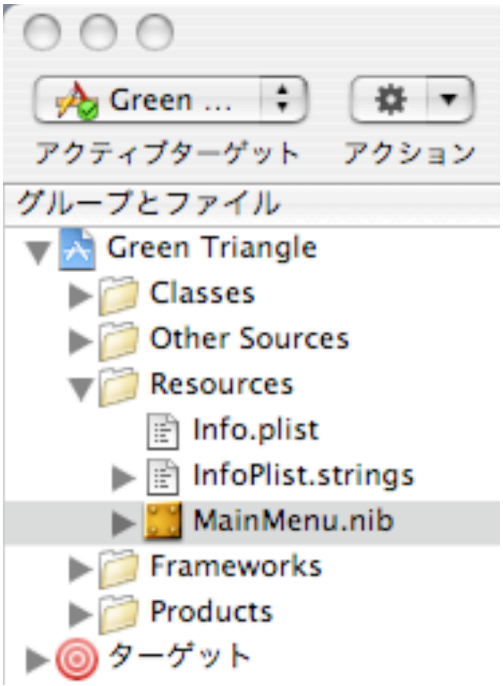
5. 「選択...」でプロジェクトの保存場所を変更することもできる。

6. 「完了」をクリックする。

（2）アプリケーションウィンドウの改良

アプリケーションのメインウィンドウの名称変更、リサイズ、パラメータ設定をします。

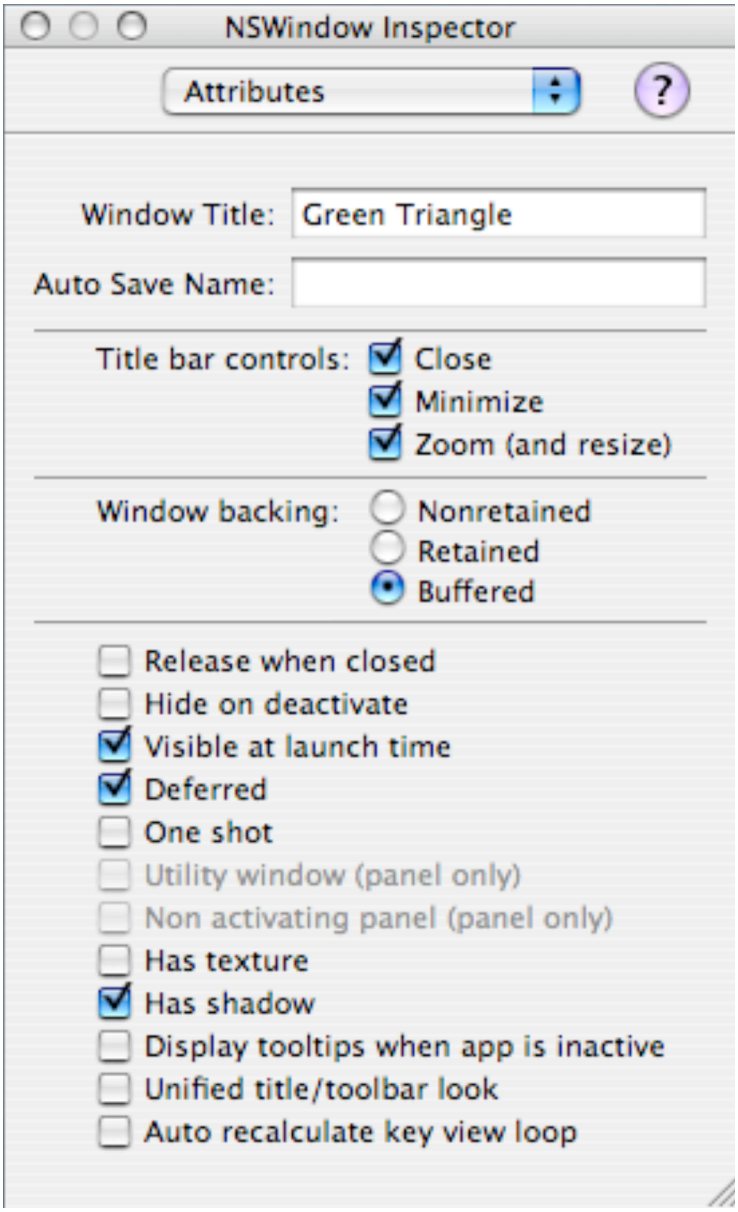
1. 「グループとファイル」ブラウザで「 Resources」を開きます。



2. 「 MainMenu.nib」をダブルクリックしてInterface Builderを起動します。

3. Toolsメニューの「Show Inspector」を選びInspectorウィンドウを表示する。

4. MainMenu.nibウィンドウから「Window」を選ぶと以下のパラメータが表示される。



5. ポップアップメニューから「Attributes」を選択する。

6. ウィンドウタイトルを「Green Triangle」に変更。

7. Title bar controlsの「Close」と「Zoom (and resize)」のチェックを外す。

8. ポップアップメニューから「Size」を選択する

9. Content Rectangle:の下のポップアップメニューから「Width/Height」を選ぶ。

10. ウィンドウのサイズを決定するためw:に320をh:に240を代入する。

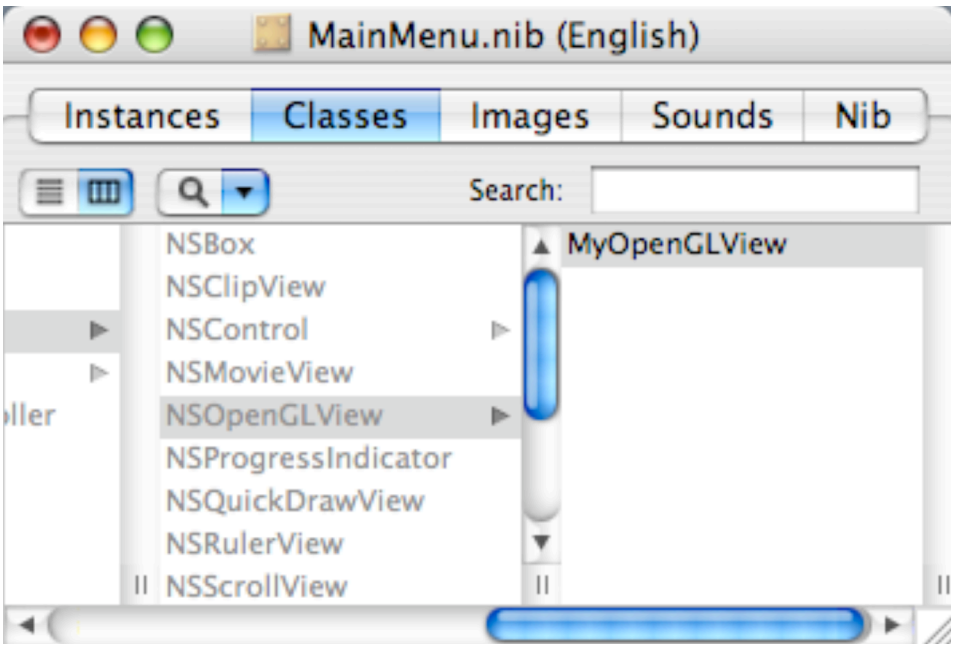
（3）OpenGL Viewのサブクラスを作成

クラスを定義するために「MainMenu.nib」ウィンドウでクラスを表示します。

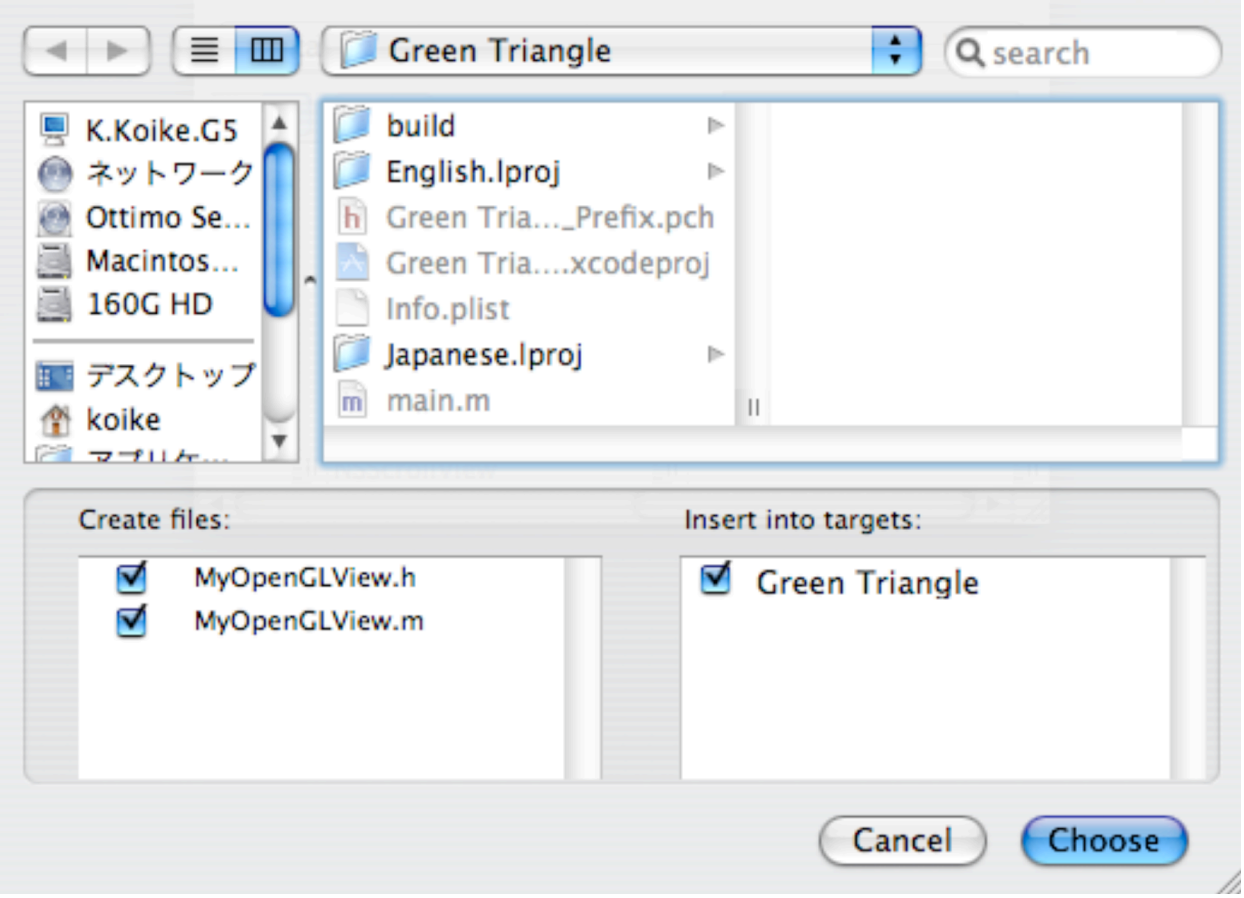
1. 「MainMenu.nib」ウィンドウの「Class」タブを選択します。

2. Search:で「NSOpenGLView」を検索したらそれをクリックして選択する。

3. Classesメニューで「Subclass NSOpenGLView」を選びMyOpenGLViewを作成。



4.Classesメニューから「Create Files for MyOpenGLView」を選択する



5. Create files:でのMyOpenGLView.hとMyOpenGLView.mのチェックを確認する。

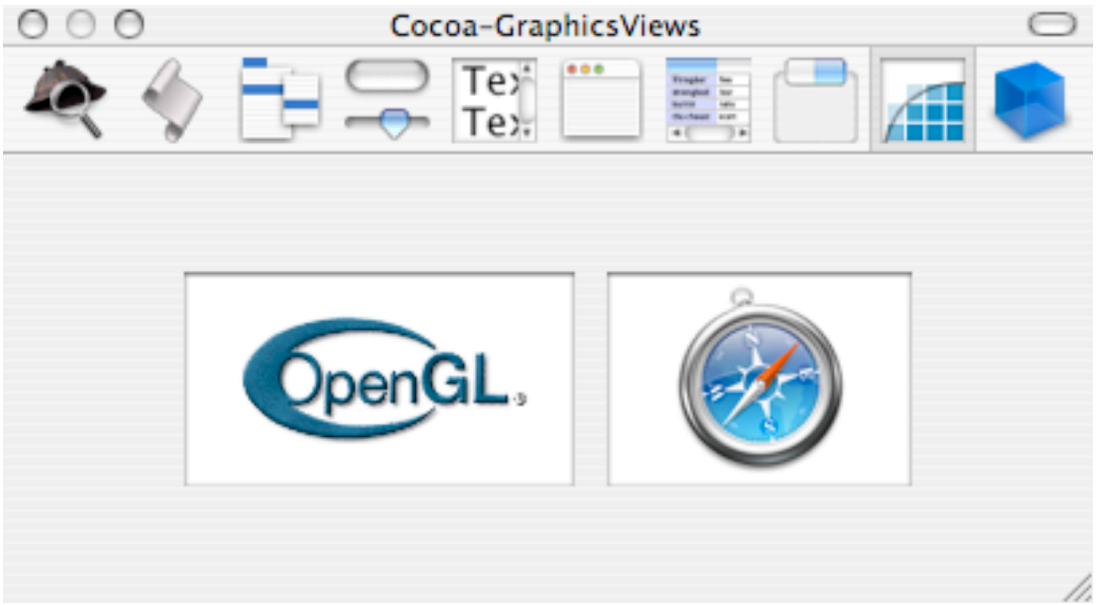
6. Insert into targets:でのGreen Triangleのチェックを確認する。

7. 「Choose」ボタンをクリックする。

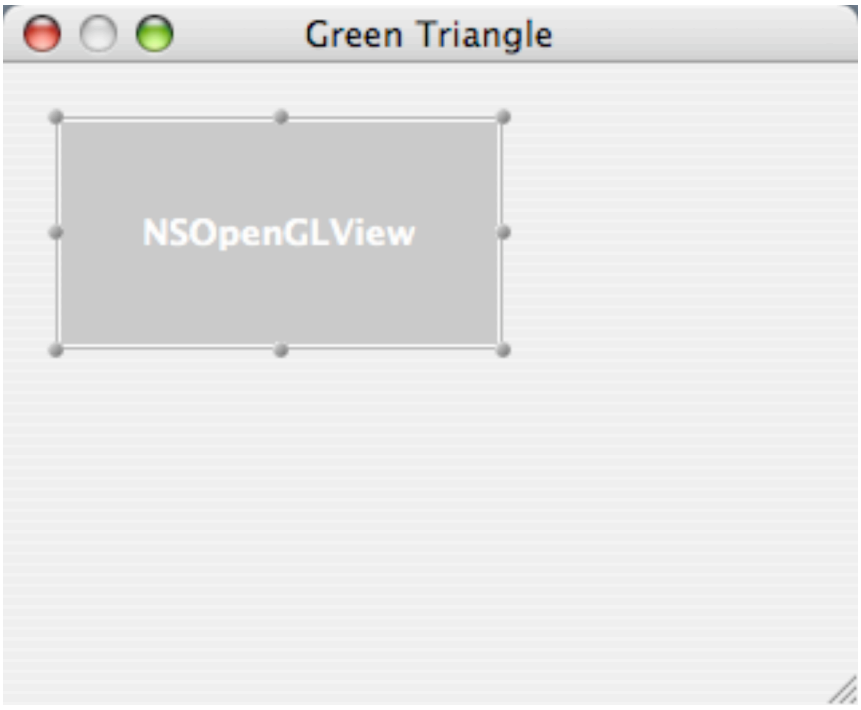
（4）カスタムView サブクラスをウィンドウに加える

ウィンドウにViewを追加するためにCocoaパレットを利用します。

1. Cocoaパレットの「Graphics Views」を選択する。



2. 「Green Triangle」ウィンドウ上にOpenGL Viewをドラッグする。



3. NSOpenGLView Inspectorのポップアップメニューから「Size」を選択。

4. 上のポップアップメニューから「Top/Left」下から「Width/Height」を選ぶ。

5. 「Top/Left」のx:とy:カラムには両方ともゼロを代入する。

6. 「Width/Height」のw:とh:カラムには320と240を代入する。

7. NSOpenGLView Inspectorのポップアップメニューから「Custom Class」を選択。

8. MyOpenGLViewを選択するとウィンドウ上のOpenGLViewの名称も変わる。

9. nibファイルを保存する。

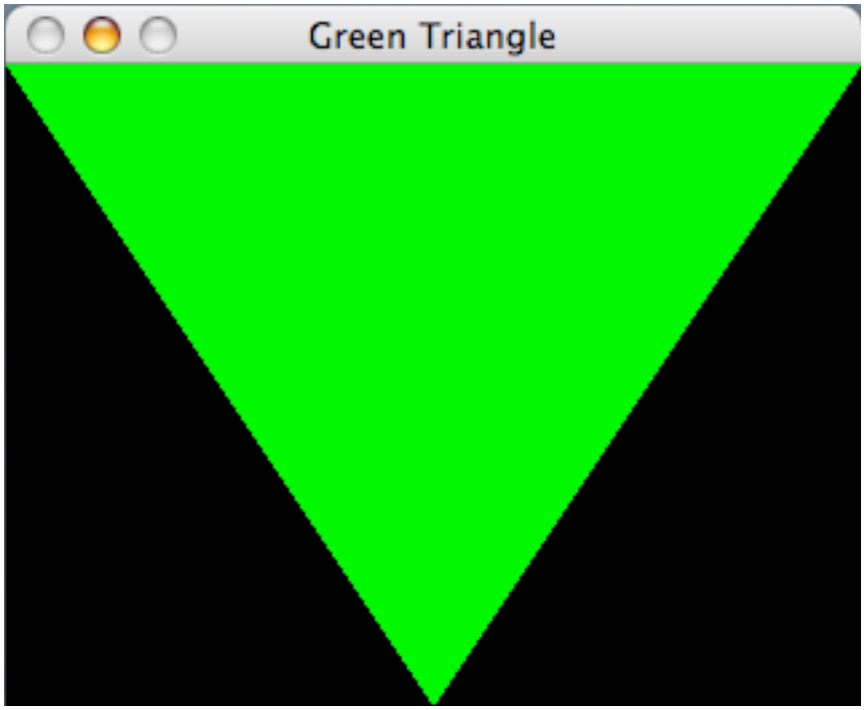
・OpenGLの描画メソッドを実装する

次の行程により、アプリケーションのメインウィンドウにはOpenGLで緑色の三角形が描画されます。Cocoaがウィンドウ上のViewを表示する時、そのViewに対してdrawRect:メッセージが送られます。ここでは、前の行程で作成したサブクラスに対するdrawRect:メソッドを実装します。以下は、完成したアプリケーションを起動した状態です。

1. プロジェクトにOpenGL Frameworkを追加する

2. drawRect:に対するメソッド宣言をする

3. drawRect:に対するメソッド定義をする



（1）プロジェクトのOpenGLフレームワークを追加

1. 「グループとファイル」ブラウザで「Frameworks」グループを開く。

2. 「Linked Frameworks」グループを選択する。

3. アクションメニューから「追加/既存のフレームワーク...」を選択する。

4. ファイル選択シートが表示されるので「OpenGL.framework」を選択する。

5. もうひとつシートが表示されるので「追加」をクリックする。

（2）メソッドの宣言を追加

1. Xcodeへ戻る。

2. 「グループとファイル」ブラウザで「Classes」グループをオープンする。

3. 「Other Sources」にあるMyOpenGLView.hとMyOpenGLView.mをそちらへ移動。

4. MyOpenGLView.hをエディタでオープンする。

5. drawRect:の宣言を挿入する。

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

@interface MyOpenGLView : NSOpenGLView
{
}
- (void) drawRect: (NSRect) bounds ;

@end
```

（3）メソッドの定義を追加

1. MyOpenGLView.mをエディタでオープンする。

2. 以下のようにdrawRect:メソッドを定義する。

```
#import "MyOpenGLView.h"
#include <OpenGL/gl.h>

@implementation MyOpenGLView
- (void) drawRect: (NSRect) bounds
{
    glClearColor( 0, 0, 0, 0 );                 // クリアーカラー（黒）
    glClear( GL_COLOR_BUFFER_BIT );  // カラーバッファをクリア
    glColor3f( 0.0f, 1.0f, 0.0f );              // 描画カラー（緑）
    glBegin( GL_TRIANGLES );                    // トライアングルバーテックス定義開始
    {
        glVertex3f( -1.0f, 1.0f, 0.0f );        // 1点目
        glVertex3f( 1.0f, 1.0f, 0.0f );         // 2点目
        glVertex3f( 0.0f, -1.0f, 0.0f );        // 3点目
    }
    glEnd();                                    // 定義終了
    glFlush();                                  // 描画
}
@end
```

3. ファイルを保存してプロジェクトウィンドウから「ビルドして実行」を選択する。

・マウスインターフェースを実装する

次の行程により、アプリケーションのウィンドウに表示された緑色の三角形の下の頂点をマウスドラッグで移動させることが可能となります。Viewに対するマウス関連のメッセージには、mouseDown:、mouseUp:、mouseDragged:、mouseMoved:などがあります。ここでは、mouseDown:とmouseDragged:メソッド、加えて初期化ルーチンとしてのinitWithCoder:メソッドを実装します。

1. mouseDown:とmouseDragged:とinitWithCoder:メソッドの宣言をする

2. 三角形の頂点を保存するインスタンス変数を宣言します。

3. mouseDown:とmouseDragged:とinitWithCoder:メソッドを定義します。

4. 描画メソッドをマウスカーソルの座標位置を使うように修正します。

（1）メソッドの宣言を追加

1. Xcodeへ戻る。

2. MyOpenGLView.hをエディタでオープンする。

3. mouseDown:とmouseDragged:とinitWithCoder:のメソッド宣言を追加する。

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

@interface MyOpenGLView : NSOpenGLView
{
}
- (id) initWithCoder: (NSCoder *) coder ;

- (void) mouseDown: (NSEvent *) theEvent ;
- (void) mouseDragged: (NSEvent *) theEvent ;

- (void) drawRect: (NSRect) bounds ;
@end
```

（2）マウスポイントのインスタンス変数を宣言する

1. マウスカーソル座標位置を示すインスタンス変数のmousePointを宣言する。

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

@interface MyOpenGLView : NSOpenGLView
{
    NSPoint mousePoint ; // インスタンス変数
}
- (id) initWithCoder: (NSCoder *) coder ;

- (void) mouseDown: (NSEvent *) theEvent ;
- (void) mouseDragged: (NSEvent *) theEvent ;
- (void) drawRect: (NSRect) bounds ;
@end
```

（3）メソッドの定義を追加

1. MyOpenGLView.mをエディタでオープンする。

2. 以下のようにmouseDown: メソッドを定義する。

```
- (void) mouseDown: (NSEvent *) theEvent
{
    [self mouseDragged: theEvent] ;    // mouseDragged:を呼ぶだけ
}
```

3. 以下のようにmouseDragged:メソッドを定義する。

```
- (void) mouseDragged: (NSEvent *) theEvent
{
    mousePoint = [theEvent locationInWindow] ;  // マウス位置を保存する
    [self setNeedsDisplay: YES] ;               // Viewに描画指示を送る
}
```

4. 以下のようにinitWithCoder:メソッドを定義する。

```
- (id) initWithCoder: (NSCoder *) coder
{
    self = [super initWithCoder: coder] ;          // superclassの初期化

    mousePoint.x = [self bounds].size.width / 2.0 ;  // Viewの左右の中心
    mousePoint.y = 0.0 ;                             // Viewの一番下

    return self ;
}
```

（4）描画メソッドのアップデート

1. MyOpenGLView.mをdrawRect:メソッドを更新する

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

```
    float transformX, transformY ;

    transformX = 2.0 * mousePoint.x / [self bounds].size.width - 1.0 ;
    transformY = 2.0 * mousePoint.y / [self bounds].size.height - 1.0 ;
    // マウス位置をViewの中心が（0,0）であるOpenGLの座標系にへと変換する

    glClearColor( 0, 0, 0, 0 ) ;
    glClear( GL_COLOR_BUFFER_BIT ) ;
    glColor3f( 0.0f, 1.0f, 0.0f ) ;
    glBegin( GL_TRIANGLES ) ;
    {
        glVertex3f( -1.0f, 1.0f, 0.0f ) ;
        glVertex3f( 1.0f, 1.0f, 0.0f ) ;
        glVertex3f( transformX, transformY, 0.0f ) ;  //マウス座標位置を利用
    }
    glEnd() ;
    glFlush() ;
}
```

2. ファイルを保存してプロジェクトウィンドウの「ビルドして実行」を選択する。

・タイマーの追加

次の行程により、アプリケーションのウィンドウに表示された緑色の三角形の緑色がタイマーを利用することで点滅するようになります。これを実現するために、アプリケーションが起動してからの時間経過をベースに三角形の色の強さを変えるメソッドを追加します。加えてタイマーを作成して、その中から定期的に色を点滅させるためのメソッドを呼びます。

1. pulseColor:とdealloc:メソッドを宣言をする。

2. 色の強さ、開始時間、タイマーを保存する各インスタンス変数を宣言する。

3. pulseColor:とdealloc:メソッドを定義をする

4. 色の初期化やタイマー作成などを加えるため初期化メソッドを更新します。

5. 色が点滅するように描画メソッドを更新します。

（1）メソッドの宣言を追加

1. Xcodeへ戻る。

2. MyOpenGLView.hをエディタでオープンする。

3. pulseColor:とdealloc:メソッドを宣言を追加する。

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

@interface MyOpenGLView : NSOpenGLView
{
    NSPoint mousePoint ;
}
- (id) initWithCoder: (NSCoder*) coder ;
- (void) dealloc ;

- (void) pulseColor ;

- (void) mouseDown: (NSEvent *) theEvent ;
- (void) mouseDragged: (NSEvent *) theEvent ;

- (void) drawRect: (NSRect) bounds ;

@end
```

（2）インスタンス変数を宣言する

1. MyOpenGLView.hをエディタして新しいインスタンス変数を追加する。

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

@interface MyOpenGLView : NSOpenGLView
{
    NSPoint mousePoint ;  // マウス位置
    float colorIntensity ;    // 緑色の強さ
    NSDate* startTime ;     // 開始時間
```

```
    NSTimer* timer ;          // タイマー
}
- (id) initWithCoder: (NSCoder*) coder ;
- (void) dealloc ;

- (void) pulseColor ;

- (void) mouseDown: (NSEvent *) theEvent ;
- (void) mouseDragged: (NSEvent *) theEvent ;

- (void) drawRect: (NSRect) bounds ;

@end
```

（3）メソッドの定義を追加

1. MyOpenGLView.mをエディタでオープンする。

2. dealloc:メソッドを追加する。

```
- (void) dealloc
{
    [timer invalidate] ;
    [timer release] ;
    [startTime release] ;
    [super dealloc] ;
}
```

3. pulseColor:メソッドを追加する。

```
- (void) pulseColor
{
    colorIntensity = fabs( sin( 2.0 * M_PI * [startTime timeIntervalSinceNow] ) ) ;
    [self setNeedsDisplay: YES] ;  // 0.0から1.0の間で数値が変化する
}
```

（4）イニシャライザーのアップデート

1. MyOpenGLView.mのinitWithCoder:メソッドを更新する

```
- (id) initWithCoder: (NSCoder *) coder
{
    SEL theSelector ;
    NSMethodSignature* aSignature ;
    NSInvocation* anInvocation ;

    self = [super initWithCoder: coder] ;

    mousePoint.x = [self bounds].size.width / 2.0 ;
    mousePoint.y = 0.0 ;

    colorIntensity = 1.0 ;            // 緑色の強さの初期値

    startTime = [NSDate date] ;  //開始時間
    [startTime retain] ;

    theSelector = @selector( pulseColor );
    aSignature = [MyOpenGLView
                instanceMethodSignatureForSelector: theSelector] ;
    anInvocation = [NSInvocation
                invocationWithMethodSignature: aSignature] ;
    [anInvocation setSelector: theSelector] ;
    [anInvocation setTarget: self] ;

    timer = [NSTimer
          scheduledTimerWithTimeInterval: 0.05
          invocation: anInvocation
          repeats: YES] ;  // タイマーを作成する
    [timer retain] ;

    return self ;
}
```

(5) 描画メソッドのアップデート

1. MyOpenGLView.mのdrawRect:メソッドを更新する

```
- (void) drawRect: (NSRect) bounds
```

```
{
    float transformX, transformY ;
    transformX = 2.0 * mousePoint.x / [self bounds].size.width - 1.0 ;
    transformY = 2.0 * mousePoint.y / [self bounds].size.height - 1.0 ;

    glClearColor( 0, 0, 0, 0 ) ;
    glClear( GL_COLOR_BUFFER_BIT ¦ GL_DEPTH_BUFFER_BIT ) ;

    glColor3f( 0.0f, colorIntensity, 0.0f ) ;  // インスタンス変数から緑成分を得る

    glBegin( GL_TRIANGLES ) ;
    {
        glVertex3f( -1.0f, 1.0f, 0.0f ) ;
        glVertex3f( 1.0f, 1.0f, 0.0f ) ;
        glVertex3f( transformX, transformY, 0.0f ) ;
    }
    glEnd() ;
    glFinish() ;
}
```

2. ファイルを保存してプロジェクトウィンドウの「ビルドして実行」を選択する。

・カスタムピクセルフォーマットの作成

次の行程により、アプリケーションはダブルバッファコンテキストを利用できるようになります。初期化メソッドでダブルバッファを指定したピクセルフォーマットを作成し、そのスワップインターバルに1を設定することで描画時のフリッカーを抑えます。また、描画メソッドの最後には、バッファ切り換え用の処理を追加します。

1. カスタムピクセルフォーマットを作成するように初期化メソッドを更新します。

2. ダブルバッファのスワップを行うように描画メソッドを更新します。

（1）イニシャライザーのアップデート

1. MyOpenGLView.mのinitWithCoder:メソッドを更新する

```
- (id) initWithCoder: (NSCoder *) coder
```

```
{
    SEL theSelector ;
    NSMethodSignature* aSignature ;
    NSInvocation* anInvocation ;

    NSOpenGLPixelFormatAttribute attrs[] =
    {
        NSOpenGLPFADoubleBuffer,  //   アトリビュートでダブルバッファを指定
        nil
    } ;
    NSOpenGLPixelFormat* pixFmt ;

    long swapInterval ;

    self = [super initWithCoder: coder] ;
    mousePoint.x = [self bounds].size.width / 2.0 ;
    mousePoint.y = 0.0 ;

    colorIntensity = 1.0 ;

    startTime = [NSDate date] ;
    [startTime retain] ;

    pixFmt = [[NSOpenGLPixelFormat alloc] initWithAttributes: attrs] ;
    [self setPixelFormat: pixFmt] ;    // ピクセルフォーマットを作成

    swapInterval = 1 ;
    [[self openGLContext]
        setValues: &swapInterval
        forParameter: NSOpenGLCPSwapInterval ] ; // スワップインターバルの設定

    theSelector = @selector( pulseColor );
    aSignature = [MyOpenGLView
                instanceMethodSignatureForSelector: theSelector] ;
    anInvocation = [NSInvocation
                invocationWithMethodSignature: aSignature] ;
    [anInvocation setSelector: theSelector] ;

    [anInvocation setTarget: self] ;
```

```
    timer = [NSTimer
            scheduledTimerWithTimeInterval: 0.05
            invocation: anInvocation
            repeats: YES] ;
    [timer retain] ;

    return self ;
}
```

（2）描画メソッドのアップデート

1. MyOpenGLView.mのdrawRect:メソッドを更新する

```
- (void) drawRect: (NSRect) bounds
{
    float transformX, transformY ;

    transformX = 2.0 * mousePoint.x / [self bounds].size.width - 1.0 ;
    transformY = 2.0 * mousePoint.y / [self bounds].size.height - 1.0 ;

    glClearColor( 0, 0, 0, 0 ) ;
    glClear( GL_COLOR_BUFFER_BIT ) ;
    glColor3f( 0.0f, colorIntensity, 0.0f ) ;
    glBegin( GL_TRIANGLES ) ;
    {
        glVertex3f( -1.0f, 1.0f, 0.0f ) ;
        glVertex3f( 1.0f, 1.0f, 0.0f ) ;
        glVertex3f( transformX, transformY, 0.0f ) ;
    }
    glEnd() ;
    glFinish() ;
    [[self openGLContext] flushBuffer] ; // バックバッファからフロントへコピー
}
```

2. ファイルを保存してプロジェクトウィンドウの「ビルドして実行」を選択する。

（6）追加情報

・関連ドキュメント

http://developer.apple.com/経由で参照することが可能です。

「CGL Reference 」（CGL_OpenGL.pdf）CGL APIリファレンス
「AGL Programming Guide」（CAGL_PG.pdf）Carbon環境AGLの使用方法
「AGL Reference 」（AGL_OpenGL.pdf）AGL APIリファレンス
「OpenGL Extensions Guide」拡張されたOpenGLコマンドについて
「OpenGL」Cocoa関連のNSOpenGLV、NSOpenGLContextなどについて

・関連テクニカルノートとQ&A

http://developer.apple.com/経由で参照することが可能です。

TN2131「OpenGL Release Highlights - Mac OS X 10.3 Panthe」
TN2093「OpenGL Performance Optimization : The Basic」
TN2080「Understanding and Detecting OpenGL Functionality」
TN2014「Insights on OpenGL」（日本語訳）
TN2007「The CGDirectDisplay API」（日本語訳）

QA1248「Context Sharing Tips」(日本語訳)
QA1166「「NSOpenGLView redraw problems after a window is closed and re-opened」（Cocoa関連）
QA1167「「OpenGL Sample Code」
QA1167「Using Interface Builder's NSOpenGLView or Custom View objects for an OpenGL application」
QA1158「glFlush() vs. glFinish() 」
QA1168「How do I determine how much VRAM is available on my video card?」
QA1168「NSTimers and Rendering Loops 」
QA1385「Using Cocoa timers (NSTimer) to drive a rendering loop」
QA1325「Creating an OpenGL texture from an NSView」」 (日本語訳)
QA1334「OpenGL Driver Monitor Decoder Ring 」 (日本語訳)
QA1268「Sharpening Full Scene Anti-Aliasing Details」

QA1248「Context Sharing Tips 」 (日本語訳)
QA1269「Mac OS X OpenGL Interfaces 」(日本語訳)
QA1209「Updating OpenGL Contexts 」(日本語訳)
QA1258「OpenGL and 3D Graphics Changes in Mac OS X v10.2.5 」(日本語訳)
QA1239「OpenGL and 3D Graphics Changes in Mac OS X v10.2.4 」(日本語訳)
QA1229「OpenGL and 3D Graphics Changes in Mac OS X v10.2.3」(日本語訳)
QA1225「Finding Missing OpenGL CFM Entry Points 」(日本語訳)
QA1222「Using Clip Region and Buffer Rectangles with OpenGL Carbon」
QA1218「How do I tell if a particular display is being hardware accelerated by Quartz Extreme?」(日本語訳)
QA1188「GetProcAdress and OpenGL Entry Points」(日本語訳)
QA1086「Setting the preferred CMM programatically? 」
QA1042「Menus & Hardware Accelerated OpenGL under Mac OS 9 Carbon」
QA1031「OpenGL Texture Sharing Between Contexts」
OGL02「Correct Setup of an AGLDrawable 」

・OpenGLに関連するドキュメント

http://www.opengl.org経由で参照することが可能です。

「OpenGL 2.0 Specification」OpenGLの公式仕様について
「OpenGL State Machine」OpenGLのステートマシンとグラフィックパイプライン
「OpenGL Reference」GLライブラリのリファレンス
「OpenGL GLU Reference」GLUライブラリのリファレンス
「OpenGL GLUT Reference」GLUTライブラリのリファレンス

・関連書籍

「OpenGL Programming Guide,Version 1.1」（赤本）12,000円
邦題：「OpenGL プログラミング ガイド」
「OpenGL Programming Guide,Version 1.1」（青本）8,300円
邦題：「OpenGL リファレンスマニュアル」
「OpenGL Programming for the X Window System」（緑本）8,500円
発行：Addison-Wesley Publishers Japan Ltd　販売：星雲社
「OpenGL Programming Guide,Version 1.4」（最新の赤本）$59.99
「OpenGL Shading Language」（オレンジ本）$59.99
発行：Addison-Wesley

本ドキュメントの履歴

オリジナル2005年11月9日　要約2005年12月15日 v1.00