iPhone OpenGL ESプログラミングガイド

2009-06-11

Apple Inc.
© 2009 Apple Inc.
All rights reserved.

本書の一部あるいは全部を Apple Inc. から書面による事前の許諾を得ることなく複写複製（コピー）することを禁じます。また、製品に付属のソフトウェアは同梱のソフトウェア使用許諾契約書に記載の条件のもとでお使いください。書類を個人で使用する場合に限り 1 台のコンピュータに保管すること、またその書類にアップルの著作権表示が含まれる限り、個人的な利用を目的に書類を複製することを認めます。

Apple ロゴは、米国その他の国で登録された Apple Inc. の商標です。

キーボードから入力可能な Apple ロゴについても、これを Apple Inc. からの書面による事前の許諾なしに商業的な目的で使用すると、連邦および州の商標法および不正競争防止法違反となる場合があります。

本書に記載されているテクノロジーに関しては、明示または黙示を問わず、使用を許諾しません。本書に記載されているテクノロジーに関するすべての知的財産権は、Apple Inc. が保有しています。 本書は、Apple ブランドのコンピュータ用のアプリケーション開発に使用を限定します。

本書には正確な情報を記載するように努めました。ただし、誤植や制作上の誤記がないことを保証するものではありません。

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
U.S.A.

アップルジャパン株式会社
〒163-1450 東京都新宿区西新宿
3 丁目20 番2 号
東京オペラシティタワー
http://www.apple.com/jp/

Apple, the Apple logo, iPod, Mac, Mac OS, Macintosh, Objective-C, Pages, Quartz, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

iPhone is a trademark of Apple Inc.

OpenGL is a registered trademark of Silicon Graphics, Inc.

Apple Inc. は本書の内容を確認しておりますが、本書に関して、明示的であるか黙示的であるかを問わず、その品質、正確さ、市場性、または特定の目的に対する適合性に関して何らかの保証または表明を行うものではありません。その結果、本書は「現状有姿のまま」提供され、本書の品質または正確さに関連して発生するすべての損害は、購入者であるお客様が負うものとします。

いかなる場合も、Apple Inc. は、本書の内容に含まれる瑕疵または不正確さによって生じる直接的、間接的、特殊的、偶発的、または結果的損害に対する賠償請求には一切応じません。そのような損害の可能性があらかじめ指摘されている場合においても同様です。

上記の損害に対する保証および救済は、口頭や書面によるか、または明示的や黙示的であるかを問わず、唯一のものであり、その他一切の保証にかわるものです。 Apple Inc. の販売店、代理店、または従業員には、この保証に関する規定に何らかの変更、拡張、または追加を加える権限は与えられていません。

一部の国や地域では、黙示あるいは偶発的または結果的損害に対する賠償の免責または制限が認められていないため、上記の制限や免責がお客様に適用されない場合があります。 この保証はお客様に特定の法的権利を与え、地域によってはその他の権利がお客様に与えられる場合もあります。

目次

序章 はじめに 9

この書類の構成 9

第 1 章 iPhone上のOpenGL ES 11

OpenGL ESとは 11
iPhoneのグラフィックスの概要 12
OpenGL ES入門 12
  OpenGL ESのオブジェクト 13
  フレームバッファ 14
EAGL 15
  EAGLContext 16
  EAGLSharegroup 16
  EAGLDrawableプロトコル 17

第 2 章 OpenGL ESの機能の判定 19

どのバージョンをターゲットとするべきか 19
  OpenGL ESのレンダリングコンテキストの作成 20
  アプリケーションを特定のバージョンのOpenGL ESに限定する 21
デバイスの機能の確認 21
  実装依存の値 21
  拡張機能を使用する前の確認 22
  エラー確認のためのglGetErrorの呼び出し 23

第 3 章 EAGLの利用 25

EAGLコンテキストの作成 25
フレームバッファオブジェクトの作成 25
  オフスクリーンのフレームバッファオブジェクト 26
  フレームバッファオブジェクトを使用したテクスチャへのレンダリング 27
  スクリーンへの描画 27
フレームバッファオブジェクトへの描画 30
結果の表示 30
sharegroup 31

第 4 章 頂点データを扱うための手法 33

ジオメトリを単純化する 33
定数を配列に格納するのは避ける 34
インターリーブされた頂点データを使用する 34

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

属性には可能な限り最小の型を使用する 34
インデックス付きの三角形を使用する 35
頂点バッファを使用する 35
頂点バッファの使用 37
頂点の構造を揃える 38

第 5 章 テクスチャデータを扱うための手法 39

テクスチャのメモリ使用量を削減する 39
テクスチャの圧縮 39
低精度のカラーフォーマットを使用する 39
適切なサイズのテクスチャを使用する 40
初期化中にテクスチャをロードする 40
複数のテクスチャをテクスチャアトラスまとめる 40
ミップマップを使用してメモリの帯域幅を削減する 41
マルチパスの代わりにマルチテクスチャを使用する 41

第 6 章 パフォーマンス 43

パフォーマンスのための一般的な推奨事項 43
必要なときだけシーンを再描画する 43
浮動小数点演算を使用する 43
使用しないOpenGL ES機能を無効にする 44
描画呼び出しの回数を最小にする 44
コンテキスト 45
メモリ 45
OpenGL ESの状態の読み書きを避ける 45
OpenGL ESの状態の照会を避ける 46
OpenGL ESの不要な状態変更は避ける 46
描画の順序 46
ライティング 47
OpenGL ES 2.0のシェーダ 47
初期化中にシェーダをコンパイルおよびリンクする 47
シェーダに関するハードウェア制限に従う 48
精度ヒントを使用する 48
ベクトル演算に注意する 49
シェーダ内の計算にはuniformまたはconstantを使用する 50
アルファテストとdiscardを避ける 51

第 7 章 プラットフォーム関連の情報 53

PowerVR SGXプラットフォーム 53
タイルベースの遅延レンダリング(TBDR) 53
PowerVR SGXでのベストプラクティス 54
PowerVR SGX上のOpenGL ES 2.0 54
PowerVR SGX上のOpenGL ES 1.1 55

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

PowerVR MBX 57
PowerVR MBXでのベストプラクティス 57
PowerVR MBX上のOpenGL ES 1.1 57
iPhone Simulator 59
iPhone Simulator上のOpenGL ES 2.0 59
iPhone Simulator上のOpenGL ES 1.1 60

付録 A texturetoolを使用したテクスチャの圧縮 61

texturetoolのパラメータ 61

改訂履歴 書類の改訂履歴 65

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

図、表、リスト

第 1 章 iPhone上のOpenGL ES 11

図 1-1 色、深度、およびステンシルの各バッファを持つフレームバッファ 15
図 1-2 2つのコンテキストによるOpenGLオブジェクトの共有 16

第 2 章 OpenGL ESの機能の判定 19

表 2-1 2つのバージョンに共通のOpenGL ESのハードウェア制限 21
表 2-2 OpenGL ES 1.1のハードウェア制限 22
表 2-3 OpenGL ES 2.0のシェーダの制限 22
リスト 2-1 同じアプリケーションでのOpenGL ES 1.1とOpenGL ES 2.0のサポート 20
リスト 2-2 OpenGL ES拡張の確認 23

第 3 章 EAGLの利用 25

図 3-1 Core AnimationとOpenGL ESとのレンダバッファの共有 28
表 3-1 フレームバッファの色アタッチメントを割り当てるための各種のメカニズム 30

第 4 章 頂点データを扱うための手法 33

リスト 4-1 OpenGL ES 1.1への頂点データの送信 35
リスト 4-2 OpenGL ES 1.1での頂点バッファの使用 36
リスト 4-3 さまざまな使用パターンを持つジオメトリ 37

第 6 章 パフォーマンス 43

リスト 6-1 シェーダのロード 47
リスト 6-2 フラグメントの色に低精度を使用する 49
リスト 6-3 ベクトル演算をうまく活用していない 49

付録 A texturetoolを使用したテクスチャの圧縮 61

リスト A-1 エンコードオプション 61
リスト A-2 画像をPVRTC圧縮フォーマットにエンコードする 63
リスト A-3 プレビューを作成するとともに画像をPVRTC圧縮フォーマットにエンコードする 63
リスト A-4 PVRTCデータをグラフィックチップにアップロードする例 63

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

はじめに

OpenGL (Open Graphics Library)は、2Dおよび3Dのデータを視覚化するための、C言語ベースのクロスプラットフォームインターフェイスです。OpenGLは、公開標準に基づく多用途のグラフィックスライブラリで、2Dおよび3Dのデジタルコンテンツ作成、機械設計および建築設計、仮想プロトタイピング、フライトシミュレーション、ビデオゲームなどのアプリケーションをサポートします。OpenGLを利用するとアプリケーションデベロッパは、点、線、または多角形として指定したモデルに無数のシェーディング技法を適用して、希望どおりのレンダリング効果を生み出すことができます。OpenGLの関数は、レンダリング先の基盤ハードウェアにグラフィックスコマンドを送信します。この基盤ハードウェアはグラフィックスコマンドの処理専用のため、OpenGLによる描画は一般に非常に高速です。

OpenGL ES (OpenGL for Embedded Systems)は、モバイル機器向けに設計されたバージョンのOpenGLで、最新のグラフィックスハードウェアの利点を活用できます。OpenGL ESでは、OpenGLインターフェイスが簡素化され、ハードウェアでの実装が容易で、より覚えやすくなっています。

この文書は以下に該当するデベロッパの方を対象としています。

- OpenGL ESを使うのが初めてで、自身のアプリケーションで使用するプログラミングインターフェイスとしてOpenGL ESが適切かを調べたい方。
- OpenGLまたはOpenGL ESには精通していて、iPhoneでの使いかたを学びたい方。

この書類の構成

「iPhone上のOpenGL ES」（11 ページ）では、OpenGL ESの概要と、それがiPhone上のグラフィックスサブシステムにおいてどのような位置づけとなるのかを示します。

「OpenGL ESの機能の判定」（19 ページ）では、OpenGL ESのバージョンを選択する方法と、どのiPhone上でも安定したアプリケーション体験を提供できるように、実行時にその機能をテストする方法を推奨します。

「EAGLの利用」（25 ページ）では、EAGLを利用して、描画コマンドの送信先としての描画コンテキストおよびフレームバッファを作成する方法を説明します。

「頂点データを扱うための手法」（33 ページ）では、OpenGL ESのレンダリングパイプラインに効率的にジオメトリを送信する方法を説明します。

「テクスチャデータを扱うための手法」（39 ページ）では、テクスチャデータの作成および使用方法を説明します。

「パフォーマンス」（43 ページ）では、OpenGL ES 1.1およびOpenGL ES 2.0の両方について、アプリケーションのパフォーマンスを向上させる方法の一般的なガイドラインを示します。

「プラットフォーム関連の情報」（53 ページ）では、iPhone上で利用可能なMBXおよびSGXの各グラフィックスプロセッサだけでなく、iPhone Simulatorについての詳細情報を提供します。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

「texturetoolを使用したテクスチャの圧縮」（61 ページ）では、`texturetool`を使用してテクスチャのメモリ使用量を削減する方法を説明します。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

iPhone上のOpenGL ES

OpenGL ES (OpenGL for Embedded Systems)は、モバイル機器向けに設計されたバージョンのOpenGLで、最新のグラフィックスハードウェアを活用します。OpenGL ESは、ハードウェアに簡単に実装できる単一のプログラミングインターフェイスを提供するために、OpenGLの冗長な機能を取り除いて簡素化しています。

OpenGL ESとは

OpenGL ESは、アプリケーションで従来型の3Dグラフィックスパイプラインを設定できるAPIを提供します。パイプラインを設定したら、アプリケーションはOpenGLに頂点を送信します。OpenGLでは、それらの頂点が変形およびライティングされ、プリミティブへと組み立てられ、ラスタ化されて2D画像が作成されます。

現在、OpenGL ESには次の2つの異なるバージョンがあります。

- OpenGL ES 1.1は、明確に定義された**固定機能パイプライン (fixed-function pipeline)**を備えた標準的なグラフィックスパイプラインを実装しています。固定機能パイプラインは、従来型のライティングおよびラスタ化モデルを実装しています。これを利用すると、パイプラインのさまざまな部分を有効にして特定のタスクを実行するように設定したり、パフォーマンスを上げるために無効にしたりすることができます。
- OpenGL ES 2.0では、多くの関数がOpenGL ES 1.1と共通ですが、固定機能パイプラインを対象に動作する関数はすべて削除され、固定機能パイプラインは汎用的な**シェーダ**ベースのパイプラインに置き換えられています。シェーダを利用すると、独自の頂点属性を作成したり、独自の頂点関数やフラグメント関数をグラフィックスハードウェア上で直接実行することができます。これによって、アプリケーションは各頂点やフラグメントに適用される操作を完全にカスタマイズできます。

現在、AppleではOpenGL ES 1.1とOpenGL ES 2.0の両方をサポートするハードウェアを提供しています。

OpenGL ESはKhronos Groupによって定義された公開標準です。OpenGL ES 1.1および2.0の詳細については、http://www.khronos.org/opengles/のWebページを参照してください。

iPhone OSベースのデバイス用にOpenGL ESアプリケーションを開発する場合は、次のリソースも参照してください。

- OpenGL ES API Registryは、OpenGL ES 1.1とOpenGL ES 2.0の仕様の公式リポジトリです。ここにはOpenGL ESのさまざまな拡張機能に関する文書も含まれています。後半の章の「プラットフォーム関連の情報」（53 ページ）では、iPhone OSで利用可能な拡張機能について説明しています。
- OpenGL ES 1.1 Reference Pagesでは、OpenGL ES 1.1の仕様の完全なリファレンス（アルファベット順のインデックス付き）が提供されています。
- OpenGL ES 2.0 Reference Pagesでは、OpenGL ES 2.0の仕様の完全なリファレンス（アルファベット順のインデックス付き）が提供されています。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

- 『*OpenGL ES Framework Reference*』では、OpenGL ESをiPhone OS上で使用できるようにするためにAppleが提供している関数およびクラスについて説明しています。

この章の以降のセクションでは、OpenGL ESの概要、iPhoneのグラフィックスモデル、およびこれら2つがどのように組み合わさるかを説明します。

iPhoneのグラフィックスの概要

Core AnimationはiPhoneのグラフィックスサブシステムの基礎です。アプリケーション内のすべての`UIView`がCore Animationレイヤを基盤にしています。さまざまなレイヤでコンテンツが更新されると、それらはCore Animationによってアニメーション化されて合成され、ディスプレイに表示されます。詳細については、『*iPhone Application Programming Guide*』の「Window and Views」を参照してください。

OpenGL ESは、iPhone上のその他のグラフィックスシステムと同様に、Core Animationのクライアントです。OpenGL ESを使用してスクリーンに描画するには、アプリケーションで特殊なCore Animationレイヤ(`CAEAGLLayer`)を持つ`UIView`クラスを作成します。`CAEAGLLayer`オブジェクトはOpenGL ESを認識し、Core Animationの一部として動作するレンダリングターゲットを作成するために使用できます。アプリケーションがフレームのレンダリングを完了したら、`CAEAGLLayer`オブジェクトのコンテンツを表示します。その結果、このコンテンツがその他のビューのデータと合成されます。

`CAEAGLLayer`の作成方法と、それを使用してレンダリングされた画像を表示する方法については、「EAGLの利用」（25 ページ）で説明します。

OpenGL ESレイヤとOpenGL ES以外の描画の両方を使用してシーンを合成できますが、実際には、OpenGL ESだけに限定した方が高いパフォーマンスを実現できます。それについては、後半の「結果の表示」（30 ページ）で詳しく説明します。

OpenGL ES入門

OpenGL ESは、ハードウェアで高速化されたレンダリングパイプラインにジオメトリを送信するための手続き型APIです。OpenGL ESのコマンドは**レンダリングコンテキスト**に送信され、そこで解釈されてユーザに表示可能な画像が生成されます。OpenGL ESのほとんどのコマンドは、次のいずれかのアクションを実行します。

- OpenGL ESコンテキストの現在の状態の読み取り。これは、OpenGL ES実装の機能を判定するために最もよく使われます。詳細については、「OpenGL ESの機能の判定」（19 ページ）を参照してください。
- OpenGL ESコンテキスト内の状態変数の変更。これは、通常、これから行う操作用にパイプラインを設定するために使われます。OpenGL ES 1.1では、状態変数は、光源や素材など、固定機能パイプラインに影響を与える値を設定するために広範囲に使われます。
- OpenGL ESオブジェクトの作成、変更、または破棄。OpenGL ES 1.1および2.0は共に、いくつかのオブジェクト（後で説明）を提供しています。
- レンダリングするジオメトリの送信。頂点データがパイプラインに送信されると、それらが処理されてプリミティブへと組み立てられ、**フレームバッファ**にラスタ化されます。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

OpenGL ESの仕様には、各関数の正確な振る舞いが定義されています。

OpenGL ES実装では、最低限の要件よりも上限を高くしたり（より大きなテクスチャを利用できるようにするなど）、**拡張メカニズム**を通じてAPIを拡張することにより、OpenGL ESの仕様を拡張することが許されています。Appleでは、拡張メカニズムを使用して、iPhone上でのパフォーマンス向上に役立つ重要な拡張機能をいくつか提供しています。たとえば、Appleでは、テクスチャ圧縮の拡張機能を提供しています。これにより、テクスチャをiPhone上のメモリに簡単に収めることができます。iPhone OSによる制限および拡張機能は、ハードウェアによって異なる可能性があります。アプリケーションは、実行時に機能を確認して、利用可能な機能に合わせて動作を変更しなければなりません。詳細については、「OpenGL ESの機能の判定」（19 ページ）を参照してください。

OpenGL ESのオブジェクト

前述のとおり、OpenGL ESにはいくつかのオブジェクトがあります。これらを作成して設定することによってシーンの作成に役立てることができます。これらすべてのオブジェクトはOpenGL ESによって管理されます。最も重要なオブジェクトタイプには以下のものがあります。

- **テクスチャ**は、グラフィックスパイプラインによってサンプリング可能な画像の1つです。通常、これはカラー画像をジオメトリ上にマッピングするために使用しますが、その他のデータをジオメトリ上にマッピングするために使用することもできます（法線、ライティング情報など）。「テクスチャデータを扱うための手法」（39 ページ）では、AppleのOpenGL ES実装でテクスチャを使用するための重要なトピックについて説明します。
- **バッファ**は、OpenGL ESが所有する一連のメモリで、アプリケーションはこのメモリとの間でデータの読み書きを行うことができます。バッファは、アプリケーションがグラフィックスハードウェアに送信する頂点データを保持するために、最もよく使われます。このバッファはOpenGL ES実装が所有しているため、頂点をより効率的に処理するために、このバッファ内のデータの配置や形式を最適化できます。特に、データがフレーム間で変化しない場合は効果的です。バッファを使用して頂点データを管理することによって、アプリケーションのパフォーマンスを大幅に向上させることができます。
- **シェーダ**もオブジェクトです。OpenGL ES 2.0のアプリケーションはシェーダを作成し、コードをシェーダにコンパイルおよびリンクしてから頂点データとフラグメントデータの処理をシェーダに割り当てます。
- **レンダバッファ**は指定されたフォーマットの単純な2Dグラフィックスです。このフォーマットは、カラーデータとして定義できますが、そのデータは深度やステンシルの情報の場合もあります。レンダバッファは、通常単独では使用されず、複数のバッファを集めてフレームバッファの一部として使用されます。
- **フレームバッファ**は、グラフィックスパイプラインの最終地点です。フレームバッファオブジェクトは、実際には、最終的なレンダリング先を作成するためにテクスチャとレンダバッファがアタッチされる単なるコンテナです。フレームバッファオブジェクトは、OpenGL ES 2.0標準の一部です。Appleでは`OES_framebuffer_object`拡張によって、OpenGL ES 1.1にもフレームバッファを実装しています。フレームバッファは、iPhoneでは広範囲に使われます。詳細については、この後で説明します。後の章の「EAGLの利用」（25 ページ）では、iPhone上でフレームバッファを作成、使用する方法について説明します。

OpenGL ESの各オブジェクトはそれぞれを操作するための固有の関数を持ちますが、すべてのオブジェクトは次に示す共通の標準モデルを採用しています。

1. オブジェクト識別子を生成する。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

アプリケーションで作成するオブジェクトごとに、識別子を生成する必要があります。識別子はポインタに似ています。アプリケーションがオブジェクトを操作する場合は常に、この識別子を使用して操作対象のオブジェクトを指定します。

オブジェクト識別子を作成しても実際にオブジェクトが割り当てられるのではなく、単にオブジェクトの参照が割り当てられるだけです。

2. オブジェクトをOpenGL ESコンテキストにバインドする。

OpenGL ESの各オブジェクトタイプには、オブジェクトをコンテキストにバインドするためのメソッドがあります。一度に操作できるオブジェクトはタイプごとにそれぞれ1つだけであり、オブジェクトをコンテキストにバインドすることによってそのオブジェクトを選択します。最初にオブジェクト識別子にバインドしたときに、OpenGL ESはメモリを割り当ててそのオブジェクトを初期化します。

3. オブジェクトの状態を変更する。

コマンドは、暗黙のうちに現在バインドされているオブジェクトを操作します。オブジェクトをバインドしたら、アプリケーションはそのオブジェクトを設定するために1つ以上のOpenGL ES呼び出しを行います。たとえば、テクスチャをバインドすると、アプリケーションは実際にそのテクスチャ画像をロードするための呼び出しを行います。

4. レンダリングのためにオブジェクトを使用する。

オブジェクトを作成して設定したら、ジオメトリの描画を始めることができます。頂点を送信する際は、現在バインドされているオブジェクトが出力のレンダリングに使われます。シェーダの場合は、現在のシェーダが最終結果の計算に使われます。その他のオブジェクトが、パイプラインのさまざまな段階で関与する場合もあります。

5. オブジェクトを削除する。

最後に、オブジェクトを使い終わったらアプリケーションはそれを削除します。オブジェクトが削除されるとその内容とオブジェクト識別子はリサイクルされます。

iPhoneでは、OpenGL ESのオブジェクトは**sharegroup**オブジェクトによって管理されます。2つ以上のレンダリングコンテキストが同じsharegroupを使用するように設定することもできます。これによって、同じデータ（たとえばテクスチャ）を使用する必要がある2つのレンダリングコンテキストが、実際に1つのテクスチャオブジェクトを共有できます。sharegroupsについては、後で詳しく説明します。

フレームバッファ

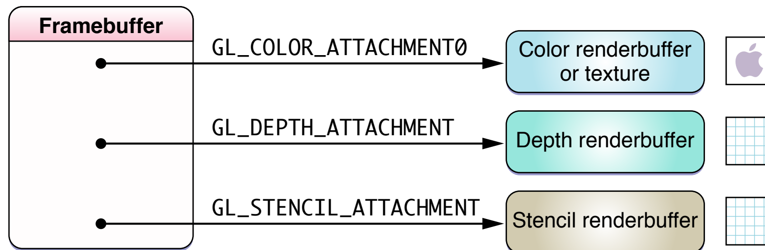
フレームバッファオブジェクトは、すべてのレンダリングコマンドのターゲットです。従来、OpenGL ESではフレームバッファはプラットフォーム定義のインターフェイスを使用して作成されていました。それぞれのプラットフォームが、スクリーンに描画可能なフレームバッファを作成するための固有の関数を提供しています。`OES_framebuffer_object`は、OpenGL ESを拡張して、オフスクリーンのレンダバッファやテクスチャにレンダリングするフレームバッファを作成、設定するための標準メカニズムを提供します。

Appleでは、フレームバッファオブジェクトを作成するためのプラットフォームインターフェイスは提供していません。代わりに、すべてのフレームバッファオブジェクトは`OES_framebuffer_object`拡張を使用して作成されます。OpenGL ES 2.0では、これらの関数がコア仕様に含まれています。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

フレームバッファオブジェクトは、画像をフレームバッファにアタッチすることで、色、深度、およびステンシルデータ用のストレージを提供します（図 1-1 （15 ページ）を参照）。最も一般的な画像アタッチメントはレンダバッファです。ただし、テクスチャをフレームバッファのカラーアタッチメントにアッタチすることもできます。これによって、画像を描画した後でほかのジオメトリにテクスチャをマッピングすることができます。

図 1-1 色、深度、およびステンシルの各バッファを持つフレームバッファ



典型的なフレームバッファ作成手順を次に示します。

1. フレームバッファオブジェクトを作成してバインドします。
2. 画像を作成し、バインドして設定します。
3. その画像をフレームバッファにアッタチします。
4. ほかの画像に対しても手順2と3を繰り返します。
5. フレームバッファの完全性をテストします。フレームバッファの完全性は、フレームバッファとそのアタッチメントが明確に定義されていることを保証する一連のルールによって成立します。完全性のためのルールは仕様において定義されています。

Appleでは、フレームバッファオブジェクトを拡張して、Core Animationレイヤと共有できるよう色レンダバッファのストレージが割り当てられるようにしています。このデータが提供されると、ほかのCore Animationデータと合成されてスクリーンに表示されます。詳細については、「EAGLの利用」（25 ページ）を参照してください。

EAGL

OpenGL ESのどの実装にも、レンダリングコンテキストを作成し、それを使用してスクリーンに描画するためのプラットフォーム固有のコードが必要になります。iPhone OSでは、これをObjective-Cインターフェイスの1つであるEAGLを通じて実現しています。このセクションでは、EAGL APIのクラスとプロトコルに焦点を当てて説明します。EAGL API の詳細については、「EAGLの利用」（25 ページ）を参照してください。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

EAGLContext

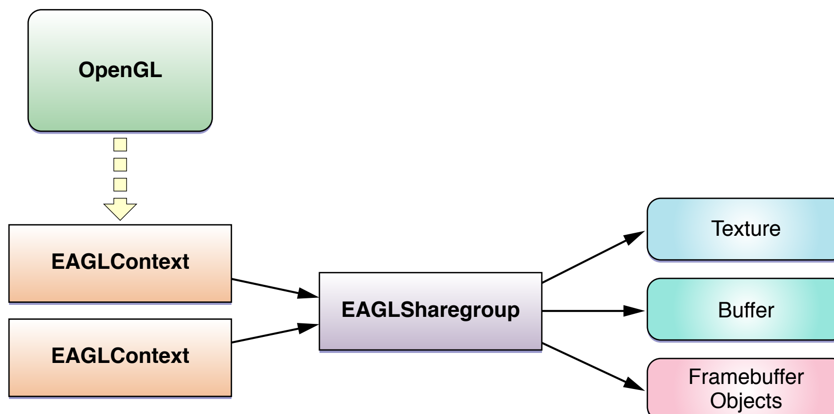
EAGLContextクラスは、OpenGL ESのすべてのコマンドのターゲットとなるレンダリングコンテキストを定義しています。アプリケーションは、EAGLContextオブジェクトを作成して初期化し、それをコマンドのカレントターゲットとして設定します。OpenGL ESコマンドが実行される際には、通常、このコンテキストが管理するキューにコマンドが格納されます。その後で、コマンドが実行されて最終的な画像がレンダリングされます。

EAGLContextには、表示のためにCore Animationに画像を提供するメソッドもあります。

EAGLSharegroup

どのEAGLContextオブジェクトにも、EAGLSharegroupオブジェクトの参照が含まれています。OpenGL ESによってコンテキストにオブジェクトが割り当てられると、実際にはsharegroupによってそのオブジェクトが割り当てられて管理されます。この責任分担は役に立ちます。それは、同じsharegroupを使用するコンテキストを複数作成できるからです。このシナリオでは、あるレンダリングコンテキストによって割り当てられたオブジェクトを別のコンテキストから使用できます（図 1-2（16 ページ）を参照）。

図 1-2 2つのコンテキストによるOpenGLオブジェクトの共有



sharegroupを使用する利点は、同じデータを必要とする2つのコンテキストが、そのデータを複製せずに済む点です。モバイル機器上のリソースは、デスクトップハードウェア上のリソースよりも限られています。テクスチャ、シェーダ、その他のオブジェクトを共有することによって、アプリケーションは利用可能なリソースを有効に活用できます。

複数のフレームバッファオブジェクトを作成し、1つのコンテキスト内でそれらを切り替えることによって、同様のリソース共有メカニズムを実装することもできます。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

EAGLDrawableプロトコル

アプリケーションが、EAGLDrawableプロトコルをオブジェクトに直接実装することはありません。EAGLContextはこのプロトコルを使用して、オブジェクトがレンダバッファにストレージを割り当てるために使用できるオブジェクトであることを認識します。このレンダバッファは後でユーザが表示できます。そのように表示できるレンダバッファは、このDrawableオブジェクトを使用して割り当てられたレンダバッファだけです。

iPhone OSでは、このプロトコルは、OpenGL ESのレンダバッファとCore Animationグラフィックスシステムを関連付けるために、CAEAGLLayerクラスによってのみ実装されています。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

OpenGL ESの機能の判定

iPhone上のOpenGL ESは、固定機能グラフィックスパイプラインを実装したOpenGL ES 1.1と、シェーダパイプラインをサポートするOpenGL ES 2.0の両方をサポートしています。OpenGL ES 2.0はOpenGL ES 1.1のスーパーセットでは*ありません*。OpenGL ES 1.1の固定機能グラフィックスパイプラインは、簡素化されたインターフェイスをグラフィックスハードウェアに提供するために削除されました。

OpenGL ES 1.1もOpenGL ES 2.0も、すべての実装がサポートしなければならない最小限の機能を定義しています。ただし、OpenGL ESの仕様ではハードウェアの実装をその機能だけに制限してはいません。OpenGL ESの実装では、この最小限の機能を拡張したり（たとえば、テクスチャの最大サイズを増やすなど）、OpenGL ES拡張メカニズムを利用してOpenGL ESの機能を追加することができます。Appleでは、この両方のメカニズムを使用して各種のiPhoneモデルにさまざまな機能を提供しています。後半の章の「プラットフォーム関連の情報」 （53 ページ）では、これらの特定の機能について詳しく説明します。しかしAppleはさまざまなバージョンのOpenGL ESを提供しており、同じバージョン内であっても提供する機能が異なるため、アプリケーション側でデバイスの機能を確認し、その機能に合わせてOpenGL ESの動作を調整する必要があります。アプリケーション側で実行時にOpenGL ESの機能を確認しなければ、アプリケーションがクラッシュするか実行できなくなり、ユーザに不快な体験をさせてしまう可能性があります。

どのバージョンをターゲットとするべきか

OpenGL ESのアプリケーションの設計時に最初に明確にしておかなければならないのは、アプリケーションでOpenGL ES 1.1とOpenGL ES 2.0のどちらをサポートするか、または両方をサポートするかということです。

OpenGL ES 1.1の固定機能パイプラインは、頂点の変形とライティングから、最終的なピクセル群とフレームバッファとのブレンドまで、3Dグラフィックスパイプラインとしての優れた基本動作を提供します。OpenGL ES 2.0アプリケーションを実装する選択をした場合は、この機能を再現する必要があります。一方、OpenGL ES 2.0はOpenGL ES 1.1よりも柔軟性に富んでいます。OpenGL ES 1.1を使用すると、独自の頂点操作やフラグメント操作を実装するのは困難あるいは不可能ですが、OpenGL ES 2.0のシェーダを利用するといとも簡単に実装できます。OpenGL ES 1.1アプリケーションで独自の操作を実装すると、一般に、マルチレンダリングパスとOpenGL ES状態の複雑な変更が必要になり、コードの意図がわかりにくくなります。アルゴリズムが複雑になるにつれて、シェーダの方がこれらの操作を明確かつ簡潔に表現でき、パフォーマンスでも優れています。

すべてのiPhoneおよびiPod touchをサポートする場合、アプリケーションはOpenGL ES 1.1をターゲットとするべきです。OpenGL ES 2.0のシェーダの表現力豊かな機能を活用する場合、アプリケーションはOpenGL ES 2.0をターゲットとするべきです。同じアプリケーションで両方を実装して、OpenGL ES 1.1のデバイス上では基本動作を提供し、OpenGL ES 2.0をサポートするデバイス上では表現力豊かな体験を提供することもできます。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

OpenGL ESのレンダリングコンテキストの作成

iPhone OSでは、アプリケーションがOpenGL ESのレンダリングコンテキスト(EAGLContext)を初期化するときに、どのバージョンのOpenGL ESを使用するかを決定します。特定のコンテキストで使用するOpenGL ESのバージョンを、アプリケーションで指定します。たとえば、OpenGL ES 1.1のコンテキストを作成するには、アプリケーションでEAGLContextを作成し、次のように初期化します。

```
EAGLContext* myContext = [[EAGLContext alloc]
initWithAPI:kEAGLRenderingAPIOpenGLES1];
```

同様に、OpenGL ES 2.0を使用する場合もコードはほとんど同じです。

```
EAGLContext* myContext = [[EAGLContext alloc]
initWithAPI:kEAGLRenderingAPIOpenGLES2];
```

OpenGL ESの特定の実装が利用できない場合、initWithAPI:はnilを返します。アプリケーションは、コンテキストを使用する前にそれが正常に初期化されたかを確認する必要があります。

同じアプリケーションでOpenGL ES 2.0とOpenGL ES 1.1の両方をサポートするには、アプリケーションは最初にOpenGL ES 2.0のレンダリングコンテキストの作成を試みる必要があります。それに失敗した場合は、OpenGL ES 1.1のコンテキストの作成を試みます。そのコードをリスト 2-1 （20 ページ）に示します。

リスト 2-1 同じアプリケーションでのOpenGL ES 1.1とOpenGL ES 2.0のサポート

```
EAGLContext* CreateBestEAGLContext()
{
   EAGLContext *context = [[EAGLContext alloc]
initWithAPI:kEAGLRenderingAPIOpenGLES2];
   if (context == nil)
   {
      context = [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGLES1];
   }
   return context;
}
```

コンテキストを初期化したら、アプリケーションはそのコンテキストのAPIプロパティを読み取って、どのバージョンのOpenGL ESをサポートしているかを判定できます。たとえば、アプリケーションで固定機能レンダラとシェーダレンダラを、共通の基底クラスを共有するクラスとして実装します。そしてアプリケーションの実行時に適切なレンダラをインスタンス化します。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

重要： OpenGL ES 2.0では、OpenGL ES 1.1の多くの関数が削除され、OpenGL ES 1.1では利用できない関数が追加されています。アプリケーションがOpenGL ES 1.1のコンテキストに対してOpenGL ES 2.0の関数を呼び出そうとした場合（または、その逆を行った場合）の結果は未定義です。2つのレンダリングパスの間でコードの共有はできますが、コードの共有は、2つのバージョンのOpenGL ESでまったく同じように動作する呼び出しのみに制限するべきです。

アプリケーションを特定のバージョンのOpenGL ESに限定する

iPhone OS 3.0から、特定のバージョンのOpenGL ESが利用できない場合にはアプリケーションを起動できないようにするために、アプリケーション側が情報プロパティリストにエントリを追加できるようになりました。このようなキーを追加する方法については、『*iPhone Application Programming Guide*』の「The Application Bundle」を参照してください。

アプリケーションがOpenGL ES 2.0とOpenGL ES 1.1の両方をサポートする場合は、OpenGL ES 2.0のデバイスのみで実行されるようにアプリケーションを制限するべきではありません。

デバイスの機能の確認

OpenGL ES 1.1またはOpenGL ES 2.0のどちらのアプリケーションをビルドするかに関わらず、次にアプリケーションで行わなければならないことは、そのOpenGL ES実装の機能を確認することです。レンダラは、コンテキストをカレントコンテキストとして設定し、その機能を一度テストして結果をキャッシュします。次にこの情報を使用して、レンダラで使用するアルゴリズムを選択します。たとえば、アプリケーションで利用可能なテクスチャユニットの数に応じて、シングルパスで複雑なアルゴリズムを実行できる場合もあれば、マルチパスで複雑なアルゴリズムを実行する必要が生じることもあります。場合によっては、より単純なアルゴリズムを選択する必要があります。コンテキストの機能は一度作成されたら変化しないため、アプリケーションではコンテキストを一度確認すれば、どちらのパスを使用するかを判断できます。

実装依存の値

OpenGL ESの仕様では、OpenGL ES実装の機能の制限を定義する、実装依存の値を定義しています。たとえば、テクスチャの最大サイズやテクスチャユニットの数は、両方のバージョンに共通の実装依存の値で、アプリケーションによる確認が前提となります。これらの各値に対しては、仕様に準拠しているすべての実装がサポートすべき最大値があります。アプリケーションの使用量がこれらの最大値を超える場合、実装はまず値の上限をチェックし、要求された上限を守れない場合は適切なエラーを発生させなければなりません。アプリケーション側は、より小さいテクスチャをロードしたり、レンダリング機能を無効にしたり、別の実装を選択したりする必要があるかもしれません。

仕様ではこれらの制限の網羅的なリストを提供していますが、そのいくつかは、ほとんどのOpenGLアプリケーションで重要です。表 2-1 （21 ページ）は、OpenGL ES 1.1とOpenGL ES 2.0の両方のアプリケーションで、最大値を超えている場合に確認する必要がある値を示しています。

表 2-1 2つのバージョンに共通のOpenGL ESのハードウェア制限

テクスチャの最大サイズ	GL_MAX_TEXTURE_SIZE

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

深度バッファのプレーン数	GL_DEPTH_BITS
ステンシルバッファのプレーンの数	GL_STENCIL_BITS

さらに、OpenGL ES 1.1アプリケーションでは、かならずテクスチャユニットの数と利用可能なクリッピングプレーンの数（表 2-2 （22 ページ）を参照）を確認する必要があります。

表 2-2 OpenGL ES 1.1のハードウェア制限

固定機能パイプラインで利用可能なテクスチャユニットの最大数	GL_MAX_TEXTURE_UNITS
クリッピングプレーンの最大数	GL_MAX_CLIP_PLANES

OpenGL ES 2.0アプリケーションでは、確認が必要になる主要な領域は、シェーダに関する制限です。どのグラフィックスハードウェアでも、頂点シェーダとフラグメントシェーダに属性を渡すためのメモリが制限されています。OpenGL ES 2.0実装では、アプリケーションがこの使用制限を超えた場合にソフトウェアのフォールバック（代替機能）を提供する必要がありません。したがって、アプリケーションは、仕様で定義されている最大値の範囲内で使い続けるか、表 2-3 （22 ページ）に示したシェーダの制限を確認して、その制限の範囲内にあるシェーダを選択しなければなりません。

表 2-3 OpenGL ES 2.0のシェーダの制限

頂点属性の最大数	GL_MAX_VERTEX_ATTRIBS
uniform頂点ベクトルの最大数	GL_MAX_VERTEX_UNIFORM_VECTORS
uniformフラグメントベクトルの最大数	GL_MAX_FRAGMENT_UNIFORM_VECTORS
varyingベクトルの最大数	GL_MAX_VARYING_VECTORS
1つの頂点シェーダで使用できるテクスチャユニットの最大数	GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS
1つのフラグメントシェーダで使用できるテクスチャユニットの最大数	GL_MAX_TEXTURE_IMAGE_UNITS

拡張機能を使用する前の確認

どのOpenGL ES実装でも、OpenGL ES拡張を実装することによってAPIの機能を拡張できます。「プラットフォーム関連の情報」（53 ページ）の章では、OpenGL ESの各実装でサポートされている機能の詳細について説明します。

アプリケーションは、OpenGL ESの仕様に加えられた変更を利用する前に、その変更を実現するOpenGL ES拡張が存在するかどうかを確認する必要があります。唯一の例外がGL_OES_framebuffer_object拡張です。フレームバッファオブジェクトは、すべてのOpenGL ES 2.0実装で利用可能です。Appleでは、OpenGL ES 1.1のすべての実装を拡張して、フレームバッファオブジェクトを利用できるようにしています。Appleでは、このフレームバッファ拡張を利用して、iPhone OSにおけるすべてのフレームバッファオブジェクトを提供しています。

リスト 2-2 （23 ページ）では、拡張の存在を確認するために使用できるコードを示しています。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

リスト 2-2 OpenGL ES拡張の確認

```
BOOL CheckForExtension(NSString *searchName)
{
// 最高のパフォーマンスを得るために、extensionsNamesをレンダラに格納して、
// 呼び出しのたびに再作成しなくてもよいようにする。
    NSString *extensionsString = [NSString
stringWithCString:glGetString(GL_EXTENSIONS) encoding:NSASCIIStringEncoding];
    NSArray *extensionsNames = [extensionsString componentsSeparatedByString:@"
 "];
    return [extensionsNames containsObject:searchName];
}
```

エラー確認のためのglGetErrorの呼び出し

デバッグバージョンのアプリケーションでは、OpenGL ESのコマンドの後に毎回glGetErrorを呼び出して、エラーが返されていることを確認する必要があります。通常、glGetErrorからエラーが返された場合は、アプリケーションがAPIを間違って使用していることを意味します。

glGetErrorを繰り返し呼び出すと、アプリケーションのパフォーマンスが大幅に低下します。リリースバージョンのアプリケーションでは、glGetErrorを呼び出すべきではありません。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

EAGLの利用

どのOpenGL ES実装も、レンダリングコンテキストの作成と操作のための関数を含むプラットフォーム固有のライブラリを提供しています。レンダリングコンテキストは、OpenGL ESのすべての状態変数を管理し、OpenGL ESのすべてのコマンドを受理して実行します。iPhone OSでは、**EAGL**がこの機能を提供します。`EAGLContext`は、OpenGL ESコマンドを実行し、Core Animationとやり取りをして最終的な画像をユーザに表示するレンダリングコンテキストです。`EAGLSharegroup`はこのレンダリングコンテキストを拡張し、複数のレンダリングコンテキストによるOpenGL ESオブジェクトの共有を可能にします。iPhone上では、sharegroupを使用してオブジェクトを共有し、テクスチャなどのコストの高いリソースを共有することでメモリを節約できる場合があります。

EAGLコンテキストの作成

アプリケーションでOpenGL ESコマンドを実行できるようにするには、まず`EAGLContext`を作成して初期化し、これをカレントコンテキストにする必要があります。

```
EAGLContext* myContext = [[EAGLContext alloc]
initWithAPI:kEAGLRenderingAPIOpenGLES1];
[EAGLContext setCurrentContext:myContext];
```

アプリケーションがコンテキストを初期化するときに、そのコンテキストでどのバージョンのOpenGL ESを使用するかを選択します。使用するOpenGL ESのバージョンの選択については、「どのバージョンをターゲットとするべきか」（19 ページ）を参照してください。

アプリケーション内の各スレッドはカレントレンダリングコンテキストへのポインタを維持します。アプリケーションがあるコンテキストをカレントコンテキストにすると、EAGLは以前のコンテキストを解放し、新しいコンテキストオブジェクトを保持して、以降のOpenGL ESレンダリングコマンドのターゲットとして設定します。ほとんどの場合、複数のレンダリングコンテキストを作成する必要はありません。通常は、レンダリングする必要がある画像ごとに単一のレンダリングコンテキストと1つのフレームバッファオブジェクトを使用すれば同じ結果を得ることができます。

フレームバッファオブジェクトの作成

EAGLContextはコマンドを受け取りますが、コマンドの最終的なターゲットではありません。アプリケーションでピクセルのレンダリング先を作成する必要があります。iPhoneでは、すべての画像はフレームバッファにレンダリングされます。フレームバッファオブジェクトはすべてのOpenGL ES 2.0実装で提供されています。Appleでは、`GL_OES_framebuffer_object`拡張によってOpenGL ES 1.1のすべての実装にもフレームバッファオブジェクトを提供しています。フレームバッファオブジェクトを利用して、アプリケーションは色、深度、およびステンシルの各ターゲットの作成を精密に制御できます。一般にこれらのターゲットはレンダバッファと呼ばれます。これは、高さ、幅およびフォーマットを持つ単なる2Dのピクセル画像です。さらに、色のターゲットをテクスチャを指すために使用することもできます。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

フレームバッファを作成する手順はどちらのバージョンでも同様です。

1. フレームバッファオブジェクトを作成します。
2. 1つ以上のターゲット（レンダバッファまたはテクスチャ）を作成し、それらにストレージを割り当ててフレームバッファオブジェクトにアタッチします。
3. フレームバッファの完全性をテストします。

以降の各セクションでは、これらの概念について詳しく説明します。

オフスクリーンのフレームバッファオブジェクト

オフスクリーンのフレームバッファは、OpenGL ESのレンダバッファを使用してレンダリング画像を保持します。

次のコードは、OpenGL ES 1.1上で完全なオフスクリーンのフレームバッファオブジェクトを割り当てます。OpenGL ES 2.0アプリケーションではOESサフィックスを省略します。

1. フレームバッファを作成しそれをバインドして、以降のOpenGL ESフレームバッファコマンドがこのバッファに送信されるようにします。

```
GLuint framebuffer;
glGenFramebuffersOES(1, &framebuffer);
glBindFramebufferOES(GL_FRAMEBUFFER_OES, framebuffer);
```

2. 色レンダバッファを作成し、それにストレージを割り当ててフレームバッファにアタッチします。

```
GLuint colorRenderbuffer;
glGenRenderbuffersOES(1, &colorRenderbuffer);
glBindRenderbufferOES(GL_RENDERBUFFER_OES, colorRenderbuffer);
glRenderbufferStorageOES(GL_RENDERBUFFER_OES, GL_RGBA8_OES, width, height);
glFramebufferRenderbufferOES(GL_FRAMEBUFFER_OES, GL_COLOR_ATTACHMENT0_OES,
GL_RENDERBUFFER_OES, depthRenderbuffer);
```

3. 同様の手順を実行し、深度レンダバッファを作成してアタッチします。

```
GLuint depthRenderbuffer;
glGenRenderbuffersOES(1, &depthRenderbuffer);
glBindRenderbufferOES(GL_RENDERBUFFER_OES, depthRenderbuffer);
glRenderbufferStorageOES(GL_RENDERBUFFER_OES, GL_DEPTH_COMPONENT16_OES, width,
 height);
glFramebufferRenderbufferOES(GL_FRAMEBUFFER_OES, GL_DEPTH_ATTACHMENT_OES,
GL_RENDERBUFFER_OES, depthRenderbuffer);
```

4. フレームバッファの完全性をテストします。

```
GLenum status = glCheckFramebufferStatusOES(GL_FRAMEBUFFER_OES) ;
if(status != GL_FRAMEBUFFER_COMPLETE_OES) {
    NSLog(@"failed to make complete framebuffer object %x", status);
}
```


2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

フレームバッファオブジェクトを使用したテクスチャへのレンダリング

アプリケーションでテクスチャに直接レンダリングしたり、テクスチャをほかの画像のソースとして使用したい場合があります。たとえば、このテクスチャを使って鏡に映る映像をレンダリングし、それを自分のシーンに合成するといったことができます。このフレームバッファを作成するコードは、オフスクリーンの例とほとんど同じです。ただし今度はカラーアタッチメントの位置にテクスチャがアタッチされる点が異なります。

1. フレームバッファオブジェクトを作成します。

```
GLuint framebuffer;
glGenFramebuffersOES(1, &framebuffer);
glBindFramebufferOES(GL_FRAMEBUFFER_OES, framebuffer);
```

2. 色データを保持するテクスチャを作成します。

```
// テクスチャを作成する
GLuint texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8,  width, height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);
```

3. テクスチャをフレームバッファにアタッチします。

```
glFramebufferTexture2DOES(GL_FRAMEBUFFER_OES, GL_COLOR_ATTACHMENT0_OES,
GL_TEXTURE_2D, texture, 0);
```

4. 深度バッファを割り当ててアタッチします。

```
GLuint depthRenderbuffer;
glGenRenderbuffersOES(1, &depthRenderbuffer);
glBindRenderbufferOES(GL_RENDERBUFFER_OES, depthRenderbuffer);
glRenderbufferStorageOES(GL_RENDERBUFFER_OES, GL_DEPTH_COMPONENT16_OES, width,
 height);
glFramebufferRenderbufferOES(GL_FRAMEBUFFER_OES, GL_DEPTH_ATTACHMENT_OES,
GL_RENDERBUFFER_OES, depthRenderbuffer);
```

5. フレームバッファをテストします。

```
GLenum status = glCheckFramebufferStatusOES(GL_FRAMEBUFFER_OES) ;
if(status != GL_FRAMEBUFFER_COMPLETE_OES) {
    NSLog(@"failed to make complete framebuffer object %x", status);
}
```

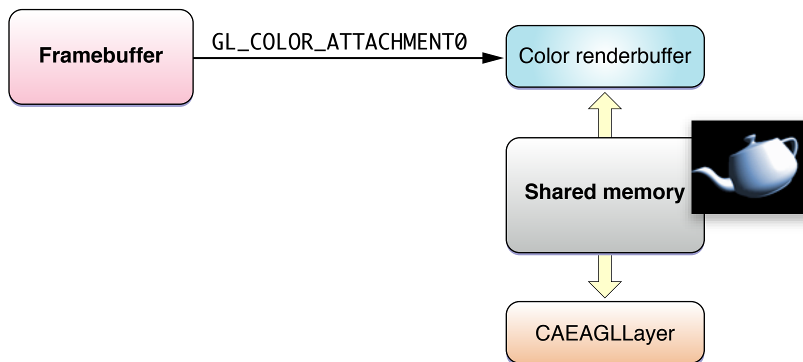
スクリーンへの描画

オフスクリーンターゲットもテクスチャも興味深いのですが、どちらもスクリーンにピクセルを表示することはできません。それには、アプリケーションがCore Animationとやり取りをする必要があります。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

iPhone OSでは、すべてのUIViewオブジェクトがCore Animationレイヤを基盤にしています。アプリケーションがOpenGL ESコンテンツをスクリーンに提示するには、ターゲットとなるUIViewが必要です。さらに、そのUIViewは、特殊なCore Animationレイヤ（CAEAGLLayer）を基盤にしていなければなりません。CAEAGLLayerはOpenGL ESを認識してレンダバッファを参照します（図 3-1（28 ページ）を参照）。アプリケーションがこれらの結果を表示すると、このレンダバッファの内容がアニメーション化され、ほかのCore Animationレイヤと合成されてスクリーンに送られます。

図 3-1 Core AnimationとOpenGL ESとのレンダバッファの共有



Xcodeが提供するOpenGL ESテンプレートにはこの処理が含まれていますが、ここでは説明のためにスクリーンに表示可能なフレームバッファオブジェクトを作成する手順を示します。

1. UIViewをサブクラス化し、iPhoneアプリケーション用のビューをセットアップします。

2. UIViewクラスのlayerClassメソッドをオーバーライドして、ビュークラスのオブジェクトがCALayerオブジェクトではなくCAEAGLLayerオブジェクトを作成および初期化するようにします。

```
+ (Class) layerClass
{
    return [CAEAGLLayer class];
}
```

3. UIViewのlayerメソッドを呼び出して、ビューに対応するレイヤを取得します。

```
myEAGLLayer = (CAEAGLLayer*)self.layer;
```

4. レイヤのプロパティを設定します。

   パフォーマンスを最適化するために、CALayerクラスが提供するopaqueプロパティを設定して、レイヤを不透過に設定することをお勧めします。詳細については、後述の「結果の表示」（30 ページ）を参照してください。

5. 必要な場合は、値の新しいディクショナリをCAEAGLLayerオブジェクトのdrawablePropertiesプロパティに割り当てて、レンダリングサーフェスのサーフェスプロパティを設定します。

   EAGLではレンダリングするピクセルのフォーマットを指定したり、レンダリングバッファの内容がスクリーンに表示された後もその内容をバッファに保持し続けるかどうかを指定したりできます。ディクショナリ内のこれらのプロパティは、kEAGLDrawablePropertyColorFormatキーおよびkEAGLDrawablePropertyRetainedBackingキーを使用して識別します。設定可能なキーの一覧については、EAGLDrawableプロトコルを参照してください。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

6. 前述のようにして、フレームバッファを作成します。

```
GLuint framebuffer;
glGenFramebuffersOES(1, &framebuffer);
glBindFramebufferOES(GL_FRAMEBUFFER_OES, framebuffer);
```

7. 色レンダバッファを作成し、レンダリングコンテキストを呼び出して、Core Animationレイヤにストレージを割り当てます。レンダバッファストレージの幅、高さ、フォーマットは、`renderbufferStorage:fromDrawable:`メソッドを呼び出した時点の`CAEAGLLayer`オブジェクトの境界とプロパティから得られます。

```
GLuint colorRenderbuffer;
glGenRenderbuffersOES(1, &colorRenderbuffer);
glBindRenderbufferOES(GL_RENDERBUFFER_OES, colorRenderbuffer);
[myContext renderbufferStorage:GL_RENDERBUFFER_OES fromDrawable:myEAGLLayer];
glFramebufferRenderbufferOES(GL_FRAMEBUFFER_OES, GL_COLOR_ATTACHMENT0_OES,
GL_RENDERBUFFER_OES, depthRenderbuffer);
```

   Core Animationレイヤのプロパティが変更された場合、アプリケーションは`renderbufferStorage:fromDrawable:`を再度呼び出してレンダバッファを再割り当てする必要があります。再割り当てをしなかった場合、表示の際にレンダリング画像の拡大縮小や変形が生じて、パフォーマンスに大きく影響する場合があります。たとえば、テンプレートではCAEAGLLayerの境界に変更があるたびに、フレームバッファオブジェクトとレンダバッファオブジェクトが破棄されて再作成されます。

8. 色レンダバッファの高さと幅を取得します。

```
GLint width;
GLint height;
glGetRenderbufferParameterivOES(GL_RENDERBUFFER_OES, GL_RENDERBUFFER_WIDTH_OES,
 &width);
glGetRenderbufferParameterivOES(GL_RENDERBUFFER_OES, GL_RENDERBUFFER_HEIGHT_OES,
 &height);
```

9. 深度バッファを割り当ててアタッチします。

```
GLuint depthRenderbuffer;
glGenRenderbuffersOES(1, &depthRenderbuffer);
glBindRenderbufferOES(GL_RENDERBUFFER_OES, depthRenderbuffer);
glRenderbufferStorageOES(GL_RENDERBUFFER_OES, GL_DEPTH_COMPONENT16_OES, width,
 height);
glFramebufferRenderbufferOES(GL_FRAMEBUFFER_OES, GL_DEPTH_ATTACHMENT_OES,
GL_RENDERBUFFER_OES, depthRenderbuffer);
```

10. フレームバッファオブジェクトをテストします。

```
GLenum status = glCheckFramebufferStatusOES(GL_FRAMEBUFFER_OES) ;
if(status != GL_FRAMEBUFFER_COMPLETE_OES) {
    NSLog(@"failed to make complete framebuffer object %x", status);
}
```

繰り返しになりますが、フレームバッファオブジェクトを作成する手順はこれら3つのすべてのケースで同様です。ただし、フレームバッファオブジェクトの色アタッチメントの位置にアタッチするオブジェクトの割り当て方法だけが異なります。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

表 3-1 フレームバッファの色アタッチメントを割り当てるための各種のメカニズム

オフスクリーンのレンダバッファ	`glRenderbufferStorageOES`
Drawableレンダバッファ	`renderbufferStorage: fromDrawable:`
テクスチャ	`glFramebufferTexture2DOES`

フレームバッファオブジェクトへの描画

フレームバッファオブジェクトを割り当てたら、それにレンダリングできます。すべてのレンダリングは、現在バインドされているフレームバッファに対して実行されます。

```
glBindFramebufferOES(GL_FRAMEBUFFER_OES, framebuffer);
```

結果の表示

Core Animationレイヤを指す色レンダバッファを割り当てた場合を考えます。それをカレントレンダバッファとして設定し、レンダリングコンテキストに対して`presentRenderbuffer:`を呼び出すと、その内容が表示されます。

```
glBindRenderbufferOES(GL_RENDERBUFFER_OES, colorRenderbuffer);
[context presentRenderbuffer:GL_RENDERBUFFER_OES];
```

デフォルトでは、レンダバッファの内容はスクリーンに表示された後は無効になります。アプリケーションは、フレームを描画するたびにレンダバッファの内容を完全に再作成しなければなりません。複数のフレーム間でその内容を維持する必要がある場合、アプリケーションは`CAEAGLLayer`の`drawableProperties`プロパティに格納されている辞書に`kEAGLDrawablePropertyRetainedBacking`キーを追加する必要があります。レイヤの内容を維持するには余分なメモリを割り当てる必要があるため、アプリケーションのパフォーマンスが低下する可能性があります。

レンダバッファがスクリーンに表示されると、それがアニメーション化されてディスプレイに表示されているほかのCore Animationレイヤと合成されます。これらのレイヤが、OpenGL ES、Quartz、またはその他のグラフィックスライブラリによって描画されたかどうかは関係ありません。OpenGL ESコンテンツをほかのコンテンツと合成すると、パフォーマンスが低下します。最良のパフォーマンスを実現するために、アプリケーションではOpenGL ESのみを使用してコンテンツをレンダリングすることをお勧めします。それにはスクリーンと同じサイズの`CAEAGLLayer`オブジェクトを作成し、`opaque`プロパティを`YES`に設定してその他のCore Animationレイヤやビューが見えないようにします。

OpenGL ESコンテンツをその他のレイヤとブレンドしなければならない場合は、`CAEAGLLayer`オブジェクトを不透過にするとパフォーマンスの低下を抑えることができます。ただし、パフォーマンスの低下をなくすことはできません。

`CAEAGLLayer`オブジェクトをその他のレイヤと合成しなければならない場合は、大幅にパフォーマンスが低下します。`CAEAGLLayer`をその他のUIKitレイヤの背後で再生することによって、このパフォーマンス低下を抑えることができます。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

> **注：** 透過なOpenGL ESコンテンツをブレンドしなければならない場合は、Core Animationで適切な合成を行うために、レンダバッファが事前乗算済みのアルファを持つバッファを1つ提供しなければなりません。

最後に、Core Animationの変形をCAEAGLLayerオブジェクトに適用する必要性が生じることはほとんどありません。そうした処理は、コンテンツを表示する前にCore Animationが実行すべき処理に対する負担となります。アプリケーションでは通常、モデルビューや投影マトリックス（または頂点シェーダの同様の要素）を変更したり、glViewport関数やglScissor関数の幅と高さの引数を交換することによって、同じことを実現できます。

sharegroup

EAGLSharegroupオブジェクトは、1つ以上のEAGLContextオブジェクトに対応するOpenGL ESリソースを管理します。通常sharegroupは、EAGLContextオブジェクトが初期化されるとき作成され、それを参照する最後のEAGLContextオブジェクトが解放されるときに破棄されます。不透過オブジェクトなのでデベロッパがアクセスできるAPIはありません。

1つのsharegroupを使用して複数のコンテキストを作成するには、まずアプリケーションは前述の手順に従って1つのコンテキストを作成します。次に、initWithAPI:sharegroup:イニシャライザを使用して1つ以上の追加コンテキストを作成します。

```
EAGLContext* firstContext = [[EAGLContext alloc]
initWithAPI:kEAGLRenderingAPIOpenGLES1];
EAGLContext* secondContext = [[EAGLContext alloc]
initWithAPI:kEAGLRenderingAPIOpenGLES1 sharegroup:[firstContext sharegroup]];
```

sharegroupは、テクスチャ、バッファ、フレームバッファ、およびレンダバッファを管理します。sharegroup内の複数のコンテキストからオブジェクトがアクセスされる場合、これらの共有オブジェクトの状態の変化を管理するのはアプリケーションの責任です。共有オブジェクトが別のコンテキストでのレンダリングに使われている間にそのオブジェクトの状態を変更した場合、その結果は未定義です。確実な結果を得るには、アプリケーションが共有オブジェクトを変更している間はそれがレンダリングに使われないように保証するために明示的な手順をとる必要があります。さらに、sharegroup内の別のコンテキストは、共有オブジェクトを再バインドするまではそのオブジェクトの状態の変化を認識する保証はありません。

sharegroup内のコンテキスト間で共有されているオブジェクトの状態の変化を確実に得るには、アプリケーションが次のタスクをこの順番で実行する必要があります。

1. オブジェクトの状態を変更します。
2. 状態変更ルーチンを呼び出したレンダリングコンテキストに対してglFlushを呼び出します。
3. 各コンテキストが変更を認識するにはそのオブジェクトを再バインドする必要があります。

sharegroup内のすべてのコンテキストが新しいオブジェクトにバインドされると、元のオブジェクトは削除されます。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

© 2009 Apple Inc. All Rights Reserved.

頂点データを扱うための手法

どのOpenGL ESアプリケーションでも最終的なシーンをレンダリングするにはOpenGL ES APIにジオメトリを送信する必要があります。データを変更した場合は、そのたびに変更した頂点データを送信してシーンを再描画します。どのような場合も、OpenGLはこれらのコマンドを効率よく処理し、画像をすばやくレンダリングしなければなりません。アプリケーションがジオメトリを送信すると、そのたびにジオメトリが処理されてレンダリング先のハードウェアに転送されます。この章では、iPhoneのグラフィックスハードウェアで効率よく処理できるように頂点データを管理するための一般的なテクニックについて説明します。

Mac OS Xおよびその他のプラットフォーム上の従来のOpneGLに精通している人は、OpenGLにジオメトリを送信するため使われる関数にはさまざまなものがあり、その使いかたとパフォーマンス特性には大きな違いがあることを知っているでしょう。OpenGLが成熟するにつれて、頂点データを扱うためのテクニックも成熟してきました。OpenGL ESでは、パフォーマンスの高い単純なインターフェイスを提供するために、古いメカニズムが削除されています。

ジオメトリを単純化する

iPhoneのグラフィックスハードウェアは非常に強力ですが、表示する画像は非常に小さいのが通常です。iPhone上に魅力的なグラフィックスを表示するのに極端に複雑なモデルは必要ありません。1つのオブジェクトの描画に必要な頂点の数を減らすと、頂点情報を送信して処理する時間を直接的に減らすことができます。

一般に、次のようなテクニックを使用することによってジオメトリの複雑さを低減できます。

- 異なる詳細レベルのジオメトリを複数バージョン用意し、注目しているオブジェクトとの距離に応じて適切なモデルを選択します。
- OpenGL ES 1.1を使用しており、ライティングの詳細を向上させるために頂点を追加した場合は、DOT3ライティングを使用することによってジオメトリを単純化できます。それには、通常の情報を持つようにオブジェクトに対してバンプマップテクスチャを作成し、そのテクスチャユニットを使用して、`GL_DOT3_RGB`モードでテクスチャ合成を使用したライティングを適用します。
- アプリケーションがOpenGL ES 2.0のシェーダを使用するように記述されている場合は、ライティングおよびその他の頂点ごとの計算をフラグメントシェーダに簡単に移行できます。これらの計算をフラグメントごとに処理するとコストはかかりますが、より高品質な画像を作成できます。フラグメントシェーダへの計算の移行は、データが頻繁に変化する場合に使用するべきです。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

定数を配列に格納するのは避ける

1つのオブジェクト内で固定的なデータをジオメトリで使用する場合、そのデータを頂点フォーマット内で複製すべきではありません。代わりに、その配列を完全に無効にして、頂点ごとの属性状態呼び出し（glColor4ub、glTexCoord2Fなど）を使用します。OpenGL ES 2.0アプリケーションでは、固定的な頂点属性を設定したり、一定のシェーダ値を使用して、オブジェクト内で固定的でほとんど変化しない情報を保持することができます。

インターリーブされた頂点データを使用する

OpenGL ESでは、アプリケーションが必要とする属性を有効にして、そのデータ型の配列のポインタを提供します。OpenGL ESではストライドを指定できます。これは、複数の配列（配列の構造体）を提供するか、単一の頂点フォーマットを持つ1つの配列（構造体の配列）を提供するかという柔軟性をアプリケーションにもたらします。

アプリケーションでは、単一のインターリーブされたピクセルフォーマットを持つ構造体の配列を1つ使用すべきです。インターリーブされたデータを使用すると、属性ごとに別々の配列を使用するよりも優れたメモリ配置を実現できます。

属性データがほかの頂点データと異なる頻度で更新されるときは、属性データを分離したい場合もあります（これについては、後の「頂点バッファの使用」（37 ページ）で説明します）。同様に、属性データを2つ以上のジオメトリで共有できる場合も、属性データを分離することでメモリの使用量を削減できます。

属性には可能な限り最小の型を使用する

iPhoneではメモリの帯域幅が限られています。このため、頂点データのサイズを削減できればパフォーマンスを大幅に向上させることができます。各コンポーネントのサイズを指定するときは必要な結果が得られる最小の型を使用するべきです。頂点の色は、4つの符号なしバイト値を使用して指定します。テクスチャの座標は浮動小数点値ではなく、2つまたは4つの符号なしバイト値、またはshort値で指定します。

OpenGL ESのGL_FIXEDデータ型の使用は避けます。この型は、GL_FLOATと同じ量の帯域幅を使用しますが、値の範囲は小さく、余分な処理を必要とします。

比較的小さいコンポーネントを指定する場合は、アラインメントのずれによるパフォーマンス低下を避けるために頂点フォーマットをかならず再編成してください。詳細については「頂点の構造を揃える」（38 ページ）を参照してください。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

インデックス付きの三角形を使用する

glDrawArraysを使用してジオメトリを描画すると、共有している頂点は頂点バッファ内で複製されます。複製された頂点はそれぞれOpenGL ES 2.0の頂点シェーダ（OpenGL ES 1.1の場合は固定機能パイプライン）によって処理されます。その結果、不必要な計算が生じます。たくさんの頂点が複製されると、メモリとメモリ帯域幅を非常に浪費することにもなります。

その代わりに、glDrawElementsを使用してジオメトリ内の三角形にインデックスを定義します。各オブジェクト（または同じ頂点フォーマットを持つオブジェクトのグループ）を、1つのglDrawElements呼び出しを使用して送信できれば、パフォーマンスは最良になります。アプリケーションでは、2つの方法で別々のジオメトリをマージできます。1つは、完全な三角形仕様を作成し、GL_TRIANGLEを使用してジオメトリを送信する方法、もう1つは、縮退三角形を使用して複数の三角形から成る帯（ストリップ）を1つのストリップにマージし、GL_TRIANGLE_STRIPを使用してジオメトリを送信する方法です。グラフィックスハードウェアは縮退三角形をすばはやく間引くため、三角形ストリップの使用をお勧めします。

パフォーマンス向上のためには、アプリケーション側で、同じ頂点を共有する三角形が順番に描画されるように三角形の描画順序を並べ替えるべきです。ほとんどのグラフィックスハードウェアでは、各頂点で実行された計算をキャッシュする能力に限りがあります。ローカルジオメトリをまとめて送信すると、頂点を再計算する必要がなくなるため、パフォーマンスが向上します。

頂点バッファを使用する

デフォルトでは、アプリケーションは固有の頂点データを管理しそれをレンダリング先のハードウェアに送信します。ジオメトリが送信されるとそれがレンダリング先のハードウェアにコピーされます。リスト 4-1 （35 ページ）は、位置と色の情報をOpenGL ES 1.1に提供するために簡単なアプリケーションで使用するコードを示しています。

リスト 4-1 OpenGL ES 1.1への頂点データの送信

```
typedef struct _vertexStruct
{
    GLfloat position[2];
    GLubyte color[4];
} vertexStruct;

void DrawGeometry()
{
    const vertexStruct vertices[] = {...};
    const GLubyte indices[] = {...};

    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, sizeof(vertexStruct), &vertices[0].position);
    glEnableClientState(GL_COLOR_ARRAY);
   glColorPointer(4, GL_UNSIGNED_BYTE, sizeof(vertexStruct), &vertices[0].color);

    glDrawElements(GL_TRIANGLE_STRIP, sizeof(indices)/sizeof(GLubyte),
GL_UNSIGNED_BYTE, indices);
}
```


2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

このコードは動作はしますが、非効率的です。glDrawElementsが呼び出されるたびに、データがレンダリング先のグラフィックスハードウェアに再送信されます。データが変化しないのであればこのような余分なコピーは必要ありません。これを避けるには、アプリケーションがジオメトリを**頂点バッファオブジェクト**(VBO)に格納するべきです。頂点バッファオブジェクトに格納されたデータはOpenGL ESによって所有され、パフォーマンスを向上させるためにハードウェアやドライバにキャッシュすることができます。

リスト 4-2（36 ページ）は、頂点バッファを使用して頂点とインデックスを格納するように前の例を変更したものです。

リスト 4-2 OpenGL ES 1.1での頂点バッファの使用

```
typedef struct _vertexStruct
{
    GLfloat position[2];
    GLubyte color[4];
} vertexStruct;

void DrawGeometry()
{
    const vertexStruct vertices[] = {...};
    const GLubyte indices[] = {...};

    GLuint    vertexBuffer;
    GLuint    indexBuffer;

    glGenBuffers(1, &vertexBuffer);
    glGenBuffers(1, &indexBuffer);

    glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, sizeof(vertexStruct),
(void*)offsetof(vertexStruct,position));
    glEnableClientState(GL_COLOR_ARRAY);
    glColorPointer(4, GL_UNSIGNED_BYTE, sizeof(vertexStruct),
(void*)offsetof(vertexStruct,color));

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
GL_STATIC_DRAW);

    glDrawElements(GL_TRIANGLE_STRIP, sizeof(indices)/sizeof(GLubyte),
GL_UNSIGNED_BYTE, (void*)0);
}
```

このコードでの重要な違いは、glVertexPointerとglColorPointerが頂点データを直接指していない点です。代わりにこのコードでは頂点バッファオブジェクトを作成し、glBufferDataを呼び出してそこに頂点データをコピーしています。その後で、glVertexPointerとglColorPointerが頂点バッファオブジェクト内へのオフセットと共に呼び出しています。同様に、このコードではインデックス情報を保持するバッファも作成しています。このバッファはglDrawElementsを呼び出したときにバインドされるため、すべてのインデックス情報のソースとして使用されます。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

頂点バッファの使用

頂点バッファのもう1つの大きな利点は、アプリケーションがデータの使用についてのヒントを与えることができることです。たとえばリスト 4-2（36 ページ）のコードでは、OpenGL ESに対して両方のバッファの内容が変化しないことを知らせています(GL_STATIC_DRAW)。この使用方法パラメータによって、OpenGL ESは、パフォーマンスを向上させるために頂点データの型に応じてその処理方法を変更できます。

OpenGL ESは次の使用方法をサポートします。

- GL_STATIC_DRAWは一度指定したら変更しない頂点データに対して使用します。アプリケーションは初期化中にこれらの頂点バッファを作成し、アプリケーションが終了するまでそれを繰り返し使用します。
- GL_DYNAMIC_DRAWは、データの作成後にそのデータが変化する可能性がある場合に使用します。アプリケーションはやはり初期化中にこれらのバッファを割り当てますが、glBufferSubDataを呼び出して定期的にバッファを更新します。

OpenGL ES 2.0では次のオプションを追加しています。

- GL_STREAM_DRAWは、レンダリングの回数が少なくレンダリング後に破棄される一時的なジオメトリを作成する必要がある場合に使用します。これは、（頂点シェーダでは実行できない方法で）フレームごとに頂点データを動的に変更しなければならない場合に最も役立ちます。ストリーム型の頂点バッファを使用するには、アプリケーションはglBufferDataを使用してバッファを初期化します。次にそのバッファからのジオメトリの描画とglBufferSubDataの呼び出しによるコンテンツの変更を交互に行います。

頂点フォーマット内の各種データの使用方法がそれぞれに異なる場合は、その頂点データを使用方法ごとに個別の構造体に分割し、それぞれに頂点バッファを割り当てることもできます。リスト 4-3（37 ページ）は、前出の例に変更を加えて、色データが変化する可能性があることを示すヒントを与えています。

リスト 4-3　さまざまな使用パターンを持つジオメトリ

```
typedef struct _vertexStatic
{
    GLfloat position[2];
} vertexStatic;

typedef struct _vertexDynamic
{
    GLfloat color[4];
} vertexDynamic;

void DrawGeometry()
{
    const vertexStatic staticVertexData[] = {...};
    vertexDynamic dynamicVertexData[] = {...};
    const GLubyte indices[] = {...};

// 静的データ用と動的データ用にバッファを分ける
    GLuint    staticBuffer;
    GLuint    dynamicBuffer;
    GLuint    indexBuffer;
```


2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

```
    glGenBuffers(1, &staticBuffer);
    glGenBuffers(1, &dynamicBuffer);
    glGenBuffers(1, &indexBuffer);

// 静的な位置データ
    glBindBuffer(GL_ARRAY_BUFFER, staticBuffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(staticVertexData), staticVertexData,
GL_STATIC_DRAW);
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, sizeof(vertexStatic),
(void*)offsetof(vertexStatic,position));

// 動的な色データ
    glBindBuffer(GL_ARRAY_BUFFER, dynamicBuffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(dynamicVertexData), dynamicVertexData,
 GL_DYNAMIC_DRAW);
    glEnableClientState(GL_COLOR_ARRAY);
    glColorPointer(4, GL_UNSIGNED_BYTE, sizeof(vertexDynamic),
(void*)offsetof(vertexDynamic,color));

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
GL_STATIC_DRAW);

    glDrawElements(GL_TRIANGLE_STRIP, sizeof(indices)/sizeof(GLubyte),
GL_UNSIGNED_BYTE, (void*)0);
}
```

頂点の構造を揃える

頂点フォーマットを設計するとき、すべてのコンポーネントがネイティブアラインメントに正しく揃うようにするべきです。GL_FLOATなどの32ビット値は32ビット境界に揃えるべきです。16ビットの値は16ビット境界に揃えるべきです。データが揃っていないと、特にアプリケーションが頂点バッファを使用する場合は、非常に多くの処理が必要になります。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

テクスチャデータを扱うための手法

前の章では、アプリケーションの頂点データ処理能力を向上させるための簡単な規則をいくつか説明しました。この章では、テクスチャについて同様のガイドラインを示します。

テクスチャのメモリ使用量を削減する

アプリケーションが使用するメモリ量は、iPhone上のすべてのアプリケーションにとって非常に重要です。しかし、OpenGL ESにおけるテクスチャ用メモリの使用にはさらに制約があります。PowerVR MBXハードウェアを使用するiPhoneまたはiPod touchでは、テクスチャとレンダバッファに使用できる総メモリ量が限られています（「PowerVR MBX」（57 ページ）を参照）。アプリケーションではテクスチャに使用するメモリ量をできる限り抑えます。

概して、アプリケーションではテクスチャデータのサイズと最終画像の品質とのバランスを取る必要があります。

テクスチャの圧縮

一般に、テクスチャの圧縮はメモリを最大限節約できる唯一の方法です。iPhone OS用のOpenGL ESでは、`GL_IMG_texture_compression_pvrtc`拡張を実装することによってPowerVRテクスチャ圧縮(PRVTC)フォーマットをサポートしています。PVRTC圧縮には、4ビット/チャネルと2ビット/チャネルの2つのレベルがあり、圧縮なしの32ビットテクスチャフォーマットと比較してそれぞれ8:1と16:1の圧縮率を提供します。圧縮されたPVRTCテクスチャでも、特に4ビットレベルの場合は、相当レベルの品質を提供できます。

重要： 将来、AppleのハードウェアはPVRTCテクスチャフォーマットをサポートしなくなる可能性があります。PVRTC圧縮されたテクスチャをロードする前に、`GL_IMG_texture_compression_pvrtc`拡張の存在を確認する必要があります。拡張機能の確認方法については、「拡張機能を使用する前の確認」（22 ページ）を参照してください。互換性を最大にするために、この拡張が利用できない場合に使用する圧縮なしのテクスチャをアプリケーションに含めることもできます。

テクスチャをPVRTCフォーマットに圧縮する方法については、「texturetoolを使用したテクスチャの圧縮」（61 ページ）を参照してください。

低精度のカラーフォーマットを使用する

アプリケーションで圧縮テクスチャを使用できない場合、より小さなピクセルフォーマットの使用を検討します。RGB565、RGBA5551、またはRGBA4444のテクスチャは、RGBA8888フォーマットのテクスチャの半分のメモリ使用で済みます。RGBA8888を使用するのは、アプリケーションでそのレベルの品質が必要な場合のみにします。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

適切なサイズのテクスチャを使用する

iPhoneが表示する画像は非常に小さいものです。アプリケーションでは、スクリーンに対して適切な画像を表示するために大きなテクスチャを提供する必要はありません。テクスチャの縦横両方の寸法を半分にすれば、そのテクスチャに必要なメモリ量は元のテクスチャの4分の1に削減できます。

テクスチャを縮小する前に、まずテクスチャの圧縮や低精度のカラーフォーマットの使用を試してみるべきです。一般に、PVRTCフォーマットによるテクスチャの圧縮を使用した方がテクスチャを縮小するよりも高品質の画像を出力できます。またメモリの使用量も少なくて済みます。

初期化中にテクスチャをロードする

テクスチャの作成とロードはコストのかかる操作です。最良のパフォーマンスを得るにはアプリケーションの実行中に新しいテクスチャを作成することは避けます。代わりに、初期化中にテクスチャデータを作成とロードを行います。テクスチャの作成が完了したら元の画像をかならず破棄してください。

アプリケーションがテクスチャを作成したら、フレームの最初または最後以外でのテクスチャの変更は避けます。現在のところ、すべてのiPhoneハードウェアはタイルベースの遅延レンダラを使用しています。このレンダラは、`glTexSubImage`と`glCopyTexSubImage`の呼び出しの際に特にコストがかかります。詳細については、「タイルベースの遅延レンダリング(TBDR)」（53 ページ）を参照してください。

複数のテクスチャをテクスチャアトラスまとめる

テクスチャをバインドするとOpenGL ESの状態が変化します。これにはCPUの処理時間がかかります。OpenGL ESの状態を変更する回数が少ないアプリケーションの方がパフォーマンスが高くなります。これについては、「OpenGL ESの不要な状態変更は避ける」（46 ページ）で詳しく説明します。

テクスチャの変更を避ける方法の1つは、複数の小さなテクスチャを結合して1つの大きなテクスチャ（**テクスチャアトラス**と呼ばれる）にすることです。各モデルは、このアトラス内から必要なテクスチャを選択するために、修正されたテクスチャ座標を使用します。これによって、複数のモデルがバインド済みのテクスチャを変更することなく描画を行うことができます。また、複数の`glDrawElements`呼び出しを1つの呼び出しに集約することもできます。

テクスチャアトラスには次のような制限があります。

- `GL_REPEAT`テクスチャラップパラメータを使用している場合は、テクスチャアトラスを使用できません。
- フィルタリングすると想定の範囲を超えたテクセルが取得される場合があります。テクスチャアトラス内でこのようなテクスチャを使用するには、テクスチャアトラスを構成するテクスチャの間にパディングを配置する必要があります。
- テクスチャアトラスもまた1つのテクスチャなので、OpenGL ES実装の制限を受けます。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

ミップマップを使用してメモリの帯域幅を削減する

2Dの拡大縮小なしの画像を描画する場合を除いて、アプリケーションはすべてのテクスチャにミップマップを提供するべきです。ミップマップは余分なメモリを使用しますが、テクスチャリングによるアーティファクトを抑制し画像の品質を向上させます。さらに重要なことには、サンプリングされたミップマップが小さければ小さいほどテクスチャメモリから取得されるテクセルも少なくなるので、ハードウェアが使用するメモリの帯域幅が削減され、パフォーマンスが大幅に向上します。

GL_LINEAR_MIPMAP_LINEARフィルタモードはテクスチャリングの際に最良の品質を提供しますが、メモリから余分にテクセルを取得する必要があります。GL_LINEAR_MIPMAP_NEARESTフィルタモードを指定すると、画像品質を落とす代わりにパフォーマンスを向上させることができます。

マルチパスの代わりにマルチテクスチャを使用する

多くのアプリケーションは、最終的な結果を得るために必要に応じてOpenGLの設定を変更することによって、ジオメトリに対してマルチパスを実行します。これは、何度も状態の変化が必要になるためコストがかかるだけでなく、パスのたびに頂点情報を再処理したり、後半のパスではフレームバッファからピクセルデータを読み直す必要があります。

すべてのOpenGL ES実装は少なくとも2つのテクスチャユニットをサポートしているため、アプリケーションはこれらのテクスチャユニットを使用して1回のパスでアルゴリズムの複数のステップを実行することができます。GL_MAX_TEXTURE_UNITSをパラメータとしてglGetIntegervを呼び出すと、アプリケーションで利用可能なテクスチャユニットの数を取得できます。

1つのオブジェクトをレンダリングするためにマルチパスが必要な場合は、次の点に注意してください。

- パスのたびに位置データが変化しないことを保証します。
- GL_EQUAL深度関数を使用して、ジオメトリ内のピクセルだけが変更されることを保証します。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

パフォーマンス

iPhone上のOpenGL ESアプリケーションのパフォーマンスは、Mac OS Xやその他のデスクトップオペレーティングシステム上のOpenGLのパフォーマンスとは異なります。iPhone OSベースのデバイスは強力な計算デバイスですが、デスクトップコンピュータやラップトップコンピュータが持つメモリやCPUパワーは持ち合わせていません。組み込みのGPUは典型的なデスクトップやラップトップのGPUとは異なるアルゴリズムを使用しており、メモリと電力の使用を抑えるように最適化されています。グラフィックスデータのレンダリングが非効率であると、フレームレートが低下するだけでなくiPhoneのバッテリ持続時間が著しく低下する可能性があります。

ほかの章ですでにiPhone上のさまざまなパフォーマンスボトルネックについて触れてきましたが、この章ではアプリケーションをiPhone向けに最適化する方法についてより網羅的に示していきます。

パフォーマンスのための一般的な推奨事項

必要なときだけシーンを再描画する

iPhoneは、アプリケーションが新しいフレームを表示するまでフレームを表示し続けます。画像のレンダリングに使われるデータが変わっていない場合は、アプリケーションはその画像を再処理するべきではありません。アプリケーションは、シーン内に何らかの変更があるまで新しいフレームのレンダリングを待機する必要があります。

データが変化した場合でも、必ずしもハードウェアがコマンドを処理するのと同じ速度でフレームをレンダリングする必要はありません。一般に、高速でもフレームレートが変動する場合と比べると、より低速で固定のフレームレートの方がユーザには滑らかに見えます。ほとんどのアニメーションでは30フレーム/秒の固定フレームレートで十分であり、電力消費の削減に役立ちます。

浮動小数点演算を使用する

3Dアプリケーション（特にゲーム）では、魅力的で面白い3Dの効果を生み出すために物理学、衝突検出、ライティング、アニメーションなどの処理が要求されます。これらのすべての機能が、フレームごとに評価する必要がある数学関数群に集約されています。このため、CPUには相当な演算の負荷がかかります。

iPhoneおよびiPod touchのARMプロセッサには、浮動小数点命令が元々備わっています。アプリケーションでは、固定小数点演算ではなく浮動小数点演算をできる限り使用すべきです。ネイティブの浮動小数点演算をサポートしない別のプラットフォームからアプリケーションを移植する場合は、浮動小数点型を使用するようにコードを書き換えるべきです。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

注： iPhoneは、ARM命令セットとThumb命令セットの両方をサポートしています。Thumb命令はコードサイズを削減できますが、浮動小数点演算を多用するコードのパフォーマンスを向上させるには、必ずARM命令を使用してください。XcodeでデフォルトのThumb設定をオフにするには、プロジェクトのプロパティを開き、`Compile for Thumb`ビルド設定のチェックを外します。

使用しないOpenGL ES機能を無効にする

OpenGL ES 1.1の固定機能パイプラインを使用しているか、OpenGL ES 2.0のシェーダを使用しているかに関わらず、パフォーマンスのために最良なのはアプリケーションが計算を実行しないことです。

アプリケーションでOpenGL ES 1.1を使用する場合は、シーンのレンダリングに不要な固定機能操作を無効にします。たとえば、アプリケーションでライティングやブレンディングが必要ない場合はそれらの機能を無効にします。同様に、2D描画を実行する場合はフォグとデプステストを無効にします。

OpenGL ES 2.0向けにアプリケーションを記述する場合は、単一のシェーダを作成してアプリケーションでのシーンのレンダリングに必要なあらゆるタスクを実行するように設定しようとしないでください。代わりに、的を絞った特定のタスクを実行する複数のシェーダプログラムをコンパイルします。OpenGL ES 1.1の場合と同様に、シェーダでの不要な計算を削除できれば全体的なパフォーマンスが向上します。

このガイドラインは、ほかの推奨事項（「OpenGL ESの不要な状態変更は避ける」（46 ページ）など）とのバランスを取らなければなりません。

描画呼び出しの回数を最小にする

アプリケーションがOpenGL ESで処理するジオメトリを送信するたびに、CPUはグラフィックスハードウェア用のコマンドを準備するために時間を消費します。このオーバーヘッドを削減するには、呼び出し回数が減るようにジオメトリをまとめるべきです。

三角形ストリップを使用してジオメトリを描画する場合は、2つ以上の三角形ストリップをマージして1つの三角形ストリップにすると、送信回数を削減できます。それには、2つまたは3つの同一直線上の点で形成された縮退三角形を追加します。たとえば、ストリップABCDを描画するために1回呼び出しを行って、ストリップEFGHを描画するためのもう1回呼び出しを行う代わりに、縮退三角形CDD、DDE、DEE、およびEEFを追加すれば、新規のストリップABCDDEEFGHを作成できます。このストリップは1回の送信で描画できます。

別々の三角形ストリップを1つのストリップにまとめるためには、すべてのストリップのレンダリング要件が同じでなければなりません。これは次のことを意味します。

- OpenGL ES 1.1ベースのレンダラは、OpenGLの状態を一切変えずにすべての三角形ストリップをレンダリングできなければなりません。
- OpenGL ES 2.0ベースのレンダラは、同じシェーダを使用してすべての三角形ストリップをレンダリングしなければなりません。
- 三角形ストリップは、頂点フォーマットが同じでなければなりません（同じ頂点配列や属性が有効になっている）。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

ジオメトリを整理統合して1つのOpenGL状態を使用すると、OpenGL ESの状態変更に伴うオーバヘッドを削減できるというメリットもあります。これについては、「OpenGL ES状態の読み書きを避ける」（45 ページ）で説明します。

最良の結果を得るには、近接するジオメトリを整理統合します。広範囲にわたるジオメトリの場合、シーンに表示されていない部分をアプリケーションが効率よく間引くのは難しくなります。

コンテキスト

アプリケーションとiPhoneグラフィックスシステムのそれ以外の部分とのやり取りも、アプリケーションのパフォーマンスに重大な影響を与えます。これについては、「結果の表示」（30 ページ）で詳しく説明します。

メモリ

iPhoneではメモリは貴重なリソースです。iPhoneアプリケーションは、システムおよびほかのiPhoneアプリケーションと主メモリを共有します。OpenGL ES用にメモリを割り当てると、アプリケーションのほかの部分で使用できるメモリの量が減ります。そのことを念頭に置いて、必要なオブジェクトのみを割り当て、それらが不要になったら割り当てを解除するようにします。たとえば、次に示すシナリオはいずれもメモリを節約できます。

- `glTexImage`を使用してOpenGL ESのテクスチャに画像をロードした後で、元の画像を解放します。
- アプリケーションで必要な場合にのみ深度バッファを割り当てます。
- アプリケーションですべてのリソースを一度に必要としない場合は、全リソースのサブセットだけをロードします。たとえば、1つのゲームをいくつかのレベルに分割し、各レベルにリソース制限の範囲内のリソースを割り当てます。

iPhone OSの仮想メモリシステムは、スワップファイルを使用しません。メモリ不足の状態が検出されると、仮想メモリは揮発性のページをディスクに書き込む代わりに、不揮発性メモリを解放して実行中のアプリケーションに必要なメモリを提供します。アプリケーションはメモリの使用をできる限り少なくする努力をしなければなりません。また、アプリケーションにとって必須ではないキャッシュデータを解放できるように備えておかなければなりません。メモリ不足状態への対応については、『*iPhone Application Programming Guide*』で詳しく説明しています。

iPhoneおよびiPod touchのいくつかのモデルに搭載されているPowerVR MBXプロセッサには、ほかにもメモリ要件があります。詳細については「PowerVR MBX」（57 ページ）を参照してください。

OpenGL ESの状態の読み書きを避ける

OpenGL ESの状態の読み書きを行うたびに、CPUはハードウェアにコマンド送信する前にそのコマンドを処理するために時間を消費します。時には、OpenGL ESの状態にアクセスしたことによって、その状態にアクセスする前にそれ以前の操作が強制的に完了させられる場合もあります。最良のパフォーマンスを実現するには、OpenGL ESの状態へのアクセス頻度をできる限り減らすべきです。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

OpenGL ESの状態の照会を避ける

glGet*()（glGetError()を含む）を呼び出すと、OpenGL ESでは、状態変数を取得する前にそれ以前のすべてのコマンドが実行されなければならない可能性があります。このような同期化のために、グラフィックスハードウェアとCPUが間を置かずに実行されなければならないため、並行処理の機会が減ります。

アプリケーションは、照会の必要があるOpenGL ESの状態のシャドーコピーを保持し、状態の変化に合わせてこれらのシャドーコピーを更新するべきです。

アプリケーションのデバッグビルドではglGetErrorを呼び出すことは重要ですが、アプリケーションのリリースバージョンではglGetErrorの呼び出しはパフォーマンスの低下を招きます。

OpenGL ESの不要な状態変更は避ける

OpenGL ESの状態を変更するにはハードウェアを新しい情報で更新する必要があります。その結果、ハードウェアが遅くなったり、それ以前に送信されていたコマンドを強制的に実行しなければならないことがあります。次のガイドラインに従うことで、アプリケーションで必要な状態変更の回数を減らせます。

- 共通の描画方法を使用して、アプリケーション内のオブジェクトをレンダリングする。
- シーン内のオブジェクトを、描画のために設定する必要があるOpenGL ESの状態別に並べ替える。これによって、1回の状態設定で複数のオブジェクトを一度に描画できます。そのためには、複数のオブジェクトの状態が同じである必要があります。保存-変更-復元のシーケンスは避けるべきです。
- OpenGL ESの状態を必要以上に変更しないこと。たとえば、すでにライティングが有効になっている場合は、glEnable(GL_LIGHTING)を再度呼び出さないようにします。

描画の順序

- オブジェクトを前面から背面へと並べ替えるために無駄なCPU時間を費やさないこと。iPhoneおよびiPod touchのOpenGL ESは、この処理を不要にするタイルベースの遅延レンダリングモデルを実装しています。詳細については「タイルベースの遅延レンダリング(TBDR)」（53 ページ）を参照してください。
- オブジェクトは、次に示すように不透明度順に並べ替える。
  1. 最初に不透明なオブジェクトを描画します。
  2. 次に、アルファテストが必要なオブジェクト（OpenGL ES 2.0ベースのアプリケーションの場合は、フラグメントシェーダでdiscardを使用する必要があるオブジェクト）を描画します。これらの操作にはパフォーマンスの低下が伴います（「アルファテストとdiscardを避ける」（51 ページ）を参照）。
  3. 最後に、アルファブレンドされたオブジェクトを描画します。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

ライティング

ライティングはできる限り簡素化します。これは、OpenGL ES 1.1の固定機能ライティングと、OpenGL ES 2.0のカスタムシェーダで使用するシェーダベースのライティング演算の両方に当てはまります。

- ライティングの使用はできるだけ控え、アプリケーションにとって最もシンプルなタイプのライティングを使用します。たとえば、パフォーマンス面で負担の大きいスポットライティングの代わりに、指向性ライトの使用を検討します。複雑なライティングよりも、シェーダでそれと同等のより単純なライティングを検討します。
- ライティングを事前に計算し、フラグメント処理によってサンプリング可能なテクスチャにカラー値を保存します。

OpenGL ES 2.0のシェーダ

初期化中にシェーダをコンパイルおよびリンクする

シェーダプログラムの作成は、OpenGL ESのその他の状態変更に比べて、コストのかかる操作です。リスト 6-1 （47 ページ）は、シェーダプログラムをロード、コンパイル、および検証するための典型的な方法を示しています。

リスト 6-1　シェーダのロード

```
/** シェーダの初期化時 **/
            GLuint shader, prog;
            GLchar *shaderText = "... shader text ...";

            // シェーダIDの作成
            shader = glCreateShader(GL_VERTEX_SHADER);

            // シェーダテキストの定義
            glShaderSource(shaderText);

            // シェーダのコンパイル
            glCompileShader(shader);

            // シェーダとプログラムの関連付け
            glAttachShader(prog, shader);

            // プログラムのリンク
            glLinkProgram(prog);

            // プログラムの検証
            glValidateProgram(prog);

            // コンパイル/リンクの状態のチェック
            glGetProgramiv(prog, GL_INFO_LOG_LENGTH, &logLen);
            if(logLen > 0)
           {
               // 必要に応じてエラーの表示
               glGetProgramInfoLog(prog, logLen, &logLen, log);
```


2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

```
                fprintf(stderr, "Prog Info Log:%s\n", log);
        }

     // リンクフェーズで決定されたuniform locationをすべて取得
            for(i = 0; i < uniformCt; i++)
            {
                uniformLoc[i] = glGetUniformLocation(prog, uniformName);
            }

            // リンクフェーズで決定されたattrib locationをすべて取得
            for(i = 0; i < attribCt; i++)
            {
                attribLoc[i] = glGetAttribLocation(prog, attribName);
            }

    /** シェーダのレンダリングステージ **/
    glUseProgram(prog);
```

アプリケーションを初期化するときに、プログラムのコンパイル、リンク、および検証を行うべきです。すべてのシェーダを作成したら、アプリケーションでは`glUseProgram()`を呼び出すことによって効率的にそれらを切り替えることができます。

シェーダに関するハードウェア制限に従う

OpenGL ESでは、頂点シェーダまたはフラグメントシェーダで使用できる各変数型の数が制限されています。さらに、OpenGL ES実装では、これらの制限を超えた場合のソフトウェアフォールバックを実装する必要がありません。代わりに、シェーダはコンパイルまたはリンクに失敗します。アプリケーションはすべてのシェーダを検証して、エラーが発生しなかったことを確認するべきです（リスト 6-1（47 ページ）を参照）。

また、アプリケーションはOpenGL ES実装の制限を照会して、その制限を超えたシェーダを使用しないようにしなければなりません。アプリケーションは起動時にそれぞれの値ごとに`glGetIntegerv()`を呼び出して、OpenGL ES実装の機能に合ったシェーダを選択しなければなりません。

頂点属性の最大数	GL_MAX_VERTEX_ATTRIBS
uniform頂点ベクトルの最大数	GL_MAX_VERTEX_UNIFORM_VECTORS
uniformフラグメントベクトルの最大数	GL_MAX_FRAGMENT_UNIFORM_VECTORS
varyingベクトルの最大数	GL_MAX_VARYING_VECTORS

この3つのいずれの型についても、照会に対しては4つのコンポーネントから成る利用可能な浮動小数点ベクトルの数が返されます。変数は、GLSL ES仕様に記述されているとおりにこれらのベクトルにまとめられます。

精度ヒントを使用する

組み込みデバイスの厳しいハードウェア制限に相応しいコンパクトなシェーダ変数のニーズに対応するために、GLSL ESの言語仕様に精度ヒントが追加されました。各シェーダではデフォルトの精度を定義し、個々のシェーダ変数を使用してそれをオーバーライドして、シェーダを効率的にコンパ

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

イルする方法についてのヒントをコンパイラに提供するようにします。これらのヒントはOpenGL ES実装によっては無視される可能性はありますが、より効率的なシェーダを生成するためにコンパイラによって使用される場合もあります。

どの精度ヒントをシェーディング変数に使用するかは、各変数の範囲と精度の要件によって異なります。高精度の変数は、単精度の浮動小数点数値として解釈されます。中精度の変数は、単精度の半分の浮動小数点数値として解釈されます。最後に、低精度の限定された変数は、8ビット精度の-2から+2までの数値として解釈されます。

重要： 精度ヒントで定義されている範囲制限は強制ではありません。したがって、データがこの範囲に収まると想定することはできません。

頂点データには高精度が推奨されますが、その他の変数にはこのレベルの精度は必要ありません。たとえば、フレームバッファに割り当てられたフラグメントの色は画像品質が大幅に低下させずに通常は低精度で実装できます（リスト 6-2（49 ページ）を参照）。

リスト 6-2 フラグメントの色に低精度を使用する

```
default precision highp; // フラグメントシェーダではデフォルトの精度宣言が必要である
uniform lowp sampler2D sampler; // Texture2D() の結果は低精度になる
varying lowp vec4 color;
varying vec2 texCoord;   // デフォルトの高精度を使用する

void main()
{
    gl_FragColor = color * texture2D(sampler, texCoord);
}
```

高精度の変数から開始して、この範囲と精度が必要ない変数についてはプログラムが正しく実行されることを確認しながらアプリケーションをテストし、精度を落とします。

ベクトル演算に注意する

必ずしもすべての演算がグラフィックスプロセッサによって並列に実行されるとは限りません。ベクトル演算は有用ですが、ベクトルプロセッサの過剰使用は避けるべきです。

たとえば、リスト 6-3（49 ページ）のコードは、SIMDベクトルプロセッサ上で2つの演算を実行しますが、括弧があるためスカラプロセッサ上で8つの独立した演算が必要になります。

リスト 6-3 ベクトル演算をうまく活用していない

```
highp float f0, f1;
highp vec4 v0, v1;
v0 = (v1 * f0) * f1;
```

括弧の位置をずらすことで、同じ演算をもっと効率的に実行できます。

```
highp float f0, f1;
highp vec4 v0, v1;
// スカラプロセッサ上では5つの演算のみが必要になる。
v0 = v1 * (f0 * f1);
```


2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

同様に、アプリケーションでベクトル演算に書き込みマスクを指定できる場合は、次のようにすべきです。スカラプロセッサは、使用されないコンポーネントを無視できます。

```
highp vec4 v0;
highp vec4 v1;
highp vec4 v2;
// スカラプロセッサ上では、書き込みマスクを指定すると速度が2倍になる。
v2.xz = v0 * v1;
```

シェーダ内の計算にはuniformまたはconstantを使用する

シェーダの外部で値を計算できる場合は、値をuniformまたはconstantとしてシェーダに渡します。動的な値を計算して使用すると、状況によっては非常にコストが高くつきます。

分岐を避ける

シェーダ内での分岐は推奨されません。3Dグラフィックスプロセッサ上で並列に演算を実行する能力が低下する可能性があるためです。シェーダで分岐が必要な場合は、GLSLのuniform変数、またはシェーダのコンパイル時にわかっているconstantに対して分岐すると効率がよくなります。シェーダ内で計算される値に対する分岐は、潜在的にコストが高くつく可能性があります。より良い解決策は、特定のレンダリングタスク専用のシェーダを作成することです。シェーダ内の分岐の数を減らすか、作成するシェーダの数を増やすかはトレードオフです。さまざまなシナリオをテストして、最も高速なソリューションを選択すべきです。

ループをなくす

ループを展開するか、ベクトルを使用して演算を実行することによって、多くのループをなくすことができます。たとえば、次のコードは非常に非効率的です。

```
// ループ
    int i;
    float f;
    vec4 v;

    for(i = 0; i < 4; i++)
        v[i] += f;
```

同じ演算を、コンポーネントレベルの加算を直接使用して実行できます。

```
    float f;
    vec4 v;
    v += f;
```

ループを排除できない場合は、ループに定数の上限を設けて動的な分岐を避けるのがよいでしょう。

配列アクセス

シェーダ内で計算されるインデックスを使用すると、constantまたはuniformの配列インデックスを使用するよりもコストが高くつきます。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

動的なテクスチャ検索

シェーダが、テクスチャのサンプリングに使用するテクスチャ座標を計算または変更すると、動的なテクスチャ検索が発生します。これは従属テクスチャ読み込みとも呼ばれます。GLSLはこれをサポートしていますが、これによって大幅にパフォーマンスが低下する可能性があります。シェーダが従属テクスチャ読み込みを含まない場合、テクスチャサンプリングハードウェアはすばやくテクセルをフェッチするので、メモリアクセスの待ち時間を感じさせません。

アルファテストとdiscardを避ける

アプリケーションでOpenGL ES 1.1のアルファテスト、またはOpenGL ES 2.0フラグメントシェーダの`discard`命令を使用する場合は、ハードウェア深度バッファのいくつかの最適化を無効にする必要があります。特にそのために、フラグメントの色を破棄する前に完全に計算しなければならない場合があります。

アルファテストまたはdiscardを使用してピクセルを破棄する方法の代わりに、強制的に0にしたアルファとのアルファブレンドを使用する方法があります。これは、テクスチャ内でアルファ値を検索することによって実現できます。これによって、Zバッファ（深度バッファ）の最適化を維持しつつ、実質的にはフレームバッファの色に対する変更が排除されます。深度バッファに格納されている値は変化します。

アルファテストまたは`discard`を使用する必要がある場合は、それを必要としないジオメトリをすべて処理してから、これらのオブジェクトをシーン内に独立に描画するべきです。破棄されてしまう計算の実行を避けるには、フラグメントシェーダの最初の方に`discard`命令を配置します。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

プラットフォーム関連の情報

Appleでは、各種のハードウェアプラットフォーム上にさまざまなOpenGL ES実装を提供しています。これらの実装ごとに、テクスチャに許される最大サイズから、サポートされるOpenGL ES拡張のリストまでさまざまな制限があります。この章では、最良のパフォーマンスと品質を得るためのアプリケーションの調整に役立つように、各プラットフォームの詳細について説明します。

この章の情報はiPhone OS 3.0時点のものです。将来のハードウェアやソフトウェアでは変わる可能性があります。最良の結果を得るには、アプリケーションはこの章の情報を利用する以上のことが必要です。アプリケーションは実行時にこれらの機能を確認しなければなりません（「OpenGL ESの機能の判定」（19 ページ）を参照）。

PowerVR SGXプラットフォーム

PowerVR SGXはAppleの最新のiPhoneに組み込まれているグラフィックプロセッサで、OpenGL ES 2.0向けに設計されています。PowerVR SGX用のグラフィックドライバにはOpenGL ES 1.1も実装されており、シェーダを使用して固定機能パイプラインが効率的に実装されています。PowerVRテクノロジーの詳細については『PowerVR Technology Overview』を参照してください。PowerVR SGXの詳細については『POWERVR SGX OpenGL ES 2.0 Application Development Recommendations』を参照してください。

タイルベースの遅延レンダリング(TBDR)

PowerVR SGXはTBDR（Tile Based Deferred Rendering：タイルベースの遅延レンダリング）と呼ばれる手法を用いています。レンダリングのためにOpenGL ESコマンドを送信すると、PowerVR SGXはある程度の量のレンダリングコマンドが蓄積するまでレンダリングを保留し、蓄積されたコマンドを1回のアクションで実行します。フレームバッファは複数のタイルに分割されており、シーンは1つのタイルにつき1回描画されます。各タイルでは、タイル内で見えているコンテンツだけが描画されます。遅延レンダラの主な利点は、より効率的にメモリにアクセスできることです。レンダリングをタイルに分割することによってGPUはフレームバッファからのピクセル値を効率的にキャッシュできるため、デプステストやブレンドの効率が向上します。

遅延レンダリングのもう1つの利点は、GPUがフラグメントを処理する前に非表示のサーフェスを削除できることです。表示されないピクセルは、テクスチャのサンプリングやフラグメント処理を実行せずに破棄されます。したがって、GPUがシーンをレンダリングするために実行しなければならない計算量が大幅に削減されます。この機能の効果を最大限に高めるためには、シーンのできるだけ多くの部分を不透過なコンテンツで描画して、ブレンド、アルファテスト、およびGLSLシェーダでのdiscard操作の使用を最小限に抑えるようにします。非表示のサーフェスの削除はハードウェアによって実行されるため、アプリケーションは全面から背面までのジオメトリを並べ替える必要がありません。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

遅延レンダラの下での操作の中には、従来のストリームレンダラの場合よりもコストがかかるものもあります。先に説明したメモリ帯域幅と計算量の節約は、大きなシーンを処理する場合に最も効果があります。小さいシーンのレンダリングを要求する（または、シーンのフラッシュを避けるためにリソースを複製する）ようなOpenGL ESコマンドをハードウェアが受け取ると、レンダラの効率は大幅に低下します。

たとえば、アプリケーションが`glTexSubImage`を呼び出してフレームの中央にあるテクスチャを更新する場合、レンダラは更新後のテクスチャと以前のテクスチャの両方を同時に保持する必要があるかもしれず、アプリケーション内のメモリ使用量が増加します。同様に、フレームバッファからピクセルデータを読み込もうとすると、その前のコマンドがフレームバッファを変更する場合はそれらが実行されている必要があります。

PowerVR SGXでのベストプラクティス

以下にあげるベストプラクティスは、OpenGL ES 1.1とOpenGL ES 2.0の両方のアプリケーションに当てはまります。

- 前のコマンドに依存する操作は避けます。このようなコマンドを実行する必要がある場合は、これらの操作をフレーム最初または最後にスケジューリングします。このようなコマンドには、`glTexSubImage`、`glCopyTexImage`、`glCopyTexSubImage`、`glReadPixels`、`glBindFramebufferOES`、`glFlush`、および`glFinish`があります。
- PowerVRの非表示サーフェス削除機能を活用するには、「描画の順序」（46 ページ）で示した描画のガイドラインに従います。
- VBO(Vertex Buffer Object)を利用するとPowerVR SGX上でのパフォーマンスが向上します。VBOをアプリケーションで使用する方法については「頂点バッファを使用する」（35 ページ）を参照してください。

PowerVR SGX上のOpenGL ES 2.0

制限

- 2Dテクスチャまたはキューブマップテクスチャの最大サイズは2048×2048です。これは、レンダラバッファサイズとビューポイントサイズの最大値でもあります。
- フラグメントシェーダでは最大8個のテクスチャを使用できます。頂点シェーダではテクスチャ検索はできません。
- 頂点属性は最大16個まで使用できます。
- varyingベクトルは最大8個まで使用できます。
- 頂点シェーダでは最大128個のuniformベクトルを使用できます。フラグメントシェーダでは最大64個のuniformベクトルを使用できます。
- 点のサイズの範囲は1.0ピクセルから511.0ピクセルまでです。
- 線の幅の範囲は1.0ピクセルから16.0ピクセルまでです。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

サポートされる拡張機能

次の拡張機能がサポートされています。

- GL_OES_depth24
- GL_OES_mapbuffer
- GL_OES_packed_depth_stencil
- GL_OES_rgb8_rgba8
- GL_OES_standard_derivatives
- GL_IMG_read_format
- GL_IMG_texture_compression_pvrtc
- GL_IMG_texture_format_BGRA8888

既知の制限と問題

iPhone OS 3.0の既知の制限を次に示します。

- PowerVR SGXは、2のべき乗以外のキューブマップテクスチャやミップマップテクスチャはサポートしていません。

OpenGL ES 2.0でのベストプラクティス

PowerVR SGXは、たとえベクトルとして宣言されている値でも、スカラプロセッサを使用して高精度の浮動小数点演算を処理します。書き込みマスクを適切に使用して注意深く演算を定義することで、シェーダのパフォーマンスを向上させることができます。詳細については「ベクトル演算に注意する」（49 ページ）を参照してください。

中精度および低精度の浮動小数点数値は並列に処理されます。ただし、低精度の変数には次のような特殊なパフォーマンス上の制限があります。

- 低精度で宣言されたベクトルのコンポーネントのスウィズルは、コストがかかるので避けるべきです。
- ほとんどの組み込み関数は中精度の入力と出力を使用します。パラメータとして低精度の浮動小数点を使用したり、結果を低精度の浮動小数点に代入すると、変換のために余分なオーバーヘッドが生じます。

最良の結果を得るには、低精度の変数の使用をカラー値に限定します。

PowerVR SGX上のOpenGL ES 1.1

OpenGL ES 1.1は、アプリケーションによるOpenGL ESの状態の変更に合わせてカスタマイズされるシェーダを使用して効率的に実装されています。このため、OpenGL ESの状態変更は従来のハードウェア実装の場合よりもコストが高くなります。状態の変更回数を減らした方がアプリケーションのパフォーマンスが向上します。詳細については「OpenGL ESの不要な状態変更は避ける」（46 ページ）を参照してください。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

制限

- 2Dテクスチャの最大サイズは2048×2048です。これは、レンダラバッファサイズとビューポイントサイズの最大値でもあります。
- 8個のテクスチャユニットが利用できます。
- 点のサイズの範囲は1.0ピクセルから511.0ピクセルまでです。
- 線の幅の範囲は1.0ピクセルから16.0ピクセルまでです。
- テクスチャのLODバイアスの最大値は4.0です。
- `GL_OES_matrix_palette`に対するパレットマトリックスの最大数は11、頂点ユニットの最大数は4です。
- ユーザのクリッピングプレーンの最大数は6です。

サポートされる拡張機能

- GL_OES_blend_subtract
- GL_OES_compressed_paletted_texture
- GL_OES_depth24
- GL_OES_draw_texture
- GL_OES_framebuffer_object
- GL_OES_mapbuffer
- GL_OES_matrix_palette
- GL_OES_point_size_array
- GL_OES_point_sprite
- GL_OES_read_format
- GL_OES_rgb8_rgba8
- GL_OES_texture_mirrored_repeat
- GL_OES_packed_depth_stencil
- GL_OES_stencil8
- GL_EXT_texture_lod_bias
- GL_IMG_texture_compression_pvrtc
- GL_IMG_read_format
- GL_IMG_texture_format_BGRA8888
- GL_APPLE_texture_2D_limited_npot

© 2009 Apple Inc. All Rights Reserved.

PowerVR MBX

PowerVR MBXは、OpenGL ES 1.1の固定機能パイプラインを実装しています。PowerVRテクノロジーの詳細については『PowerVR Technology Overview』を参照してください。PowerVR MBXの詳細については『PowerVR MBX 3D Application Development Recommendations』を参照してください。

PowerVR MBXはタイルベースの遅延レンダラです。これは、OpenGL ES 2.0のようにカスタムフラグメントシェーダはサポートしていませんが、従来のパイプラインは不要なフラグメント処理を回避することで高速化されています。遅延レンダラ上で効率よく動作するようにアプリケーションを調整する方法については、「タイルベースの遅延レンダリング(TBDR)」（53 ページ）を参照してください。

PowerVR MBXを対象にしたOpenGL ESアプリケーションは、テクスチャとレンダバッファ用のメモリが24MBを超えないように制限する必要があります。全体的に、PowerVR MBXの方がメモリ使用の影響を受けやすく、アプリケーションはテクスチャとレンダバッファのサイズを最小限に抑える必要があります。

PowerVR MBXでのベストプラクティス

- 最高のパフォーマンスを得るには、標準的な頂点属性を次の順番で並べます。`Position`, `Normal`, `Color`, `TexCoord0`, `TexCoord1`, `PointSize`, `Weight`, `MatrixIndex`.

PowerVR MBX上のOpenGL ES 1.1

制限

- 2Dテクスチャの最大サイズは1024×1024です。これは、レンダバッファサイズとビューポイントサイズの最大値でもあります。
- 2個のテクスチャユニットが利用できます。
- 点のサイズの範囲は1.0ピクセルから64ピクセルまでです。
- 線の幅の範囲は1.0ピクセルから64ピクセルまでです。
- テクスチャのLODバイアスの最大値は2.0です。
- `GL_OES_matrix_palette`に対するパレットマトリックスの最大数は9、頂点ユニットの最大数は3です。
- ユーザのクリッピングプレーンの最大数は1です。

サポートされる拡張機能

iPhoneおよびiPod touch向けのOpenGL ES 1.1実装でサポートされている拡張機能は次のとおりです。

- GL_OES_blend_subtract
- GL_OES_compressed_paletted_texture

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

- GL_OES_depth24
- GL_OES_draw_texture
- GL_OES_framebuffer_object
- GL_OES_mapbuffer
- GL_OES_matrix_palette
- GL_OES_point_size_array
- GL_OES_point_sprite
- GL_OES_read_format
- GL_OES_rgb8_rgba8
- GL_OES_texture_mirrored_repeat
- GL_EXT_texture_filter_anistropic
- GL_EXT_texture_lod_bias
- GL_IMG_read_format
- GL_IMG_texture_compression_pvrtc
- GL_IMG_texture_format_BGRA8888

既知の制限と問題

PowerVR MBXのOpenGL ES 1.1実装には、iPhone SimulatorおよびPowerVR SGXにはない制限がいくつかあります。それを次に示します。：

- （1つのテクスチャレベル内では）テクスチャの拡大フィルタと縮小フィルタが一致していなければなりません。たとえば、次のように設定しなければなりません。
  - サポートされる：

    ```
    GL_TEXTURE_MAG_FILTER = GL_LINEAR,
    GL_TEXTURE_MIN_FILTER = GL_LINEAR_MIPMAP_LINEAR
    ```

  - サポートされる：

    ```
    GL_TEXTURE_MAG_FILTER = GL_NEAREST,
    GL_TEXTURE_MIN_FILTER = GL_LINEAR_MIPMAP_NEAREST
    ```

  - サポートされない：

    ```
    GL_TEXTURE_MAG_FILTER = GL_NEAREST,
    GL_TEXTURE_MIN_FILTER = GL_LINEAR_MIPMAP_LINEAR
    ```

- 次のテクスチャ環境操作は利用できません（利用することはまれです）。
  - GL_COMBINE_RGBの値がGL_MODULATEの場合、2つのオペランドの一方のみがGL_ALPHAソースを読み取ることができます。
  - GL_COMBINE_RGBの値がGL_INTERPOLATE、GL_DOT3_RGB、またはGL_DOT3_RGBAの場合、GL_CONSTANTソース、GL_PRIMARY_COLORソース、GL_ALPHAオペランドの組み合わせのいくつかは正常に動作しません。

© 2009 Apple Inc. All Rights Reserved.

- ❏ GL_COMBINE_RGBまたはGL_COMBINE_ALPHAの値がGL_SUBTRACTの場合、GL_SCALE_RGBまたはGL_SCALE_ALPHAは1.0でなければなりません。
- ❏ GL_COMBINE_ALPHAの値がGL_INTERPOLATEまたはGL_MODULATEの場合、2つのソースの一方のみがGL_CONSTANTとなることができます。
- ❏ GL_TEXTURE_ENV_COLORの値は、すべてのテクスチャユニットで同一でなければなりません。

- 両面ライティング(GL_LIGHT_MODEL_TWO_SIDE)は無視されます。
- glPolygonOffsetのfactor引数は無視されます。傾斜に依存しないunitsパラメータのみが有効です。
- 遠近補正されたテクスチャは、テクスチャ座標のS座標とT座標に対してのみサポートされます。Q座標は遠近補正によって補正されません。

iPhone Simulator

iPhone Simulatorは、OpenGL ES 1.1とOpenGL ES 2.0の両方に準拠した完全な実装を備えており、アプリケーション開発に利用することができます。iPhone Simulatorは、次のいくつかの点がPowerVR MBXやPowerVR SGXとは異なります。

- iPhone Simulatorはタイルベースの遅延レンダラを使用しません。
- iPhone SimulatorではPowerVR MBXのメモリ制限が適用されません。
- iPhone SimulatorはPowerVR MBXやPowerVR SGXと同じ拡張をサポートしません。
- iPhone SimulatorはPowerVR MBXとPowerVR SGXのどちらもシミュレートしません。iPhone Simulatorによって生成される画像は、デバイス上で生成される画像とピクセル単位で一致しません。

重要： iPhone SimulatorでのOpenGL ESのレンダリングパフォーマンスは、実際のデバイスでのOpenGL ESのパフォーマンスとは無関係であることを理解することが重要です。iPhone Simulatorは、Macintoshコンピュータのベクトル処理機能を利用した、最適化されたソフトウェアラスタライザを備えています。その結果、OpenGL ESのコードは、（実行するコンピュータや描画内容によって）実際のデバイスよりもiPhone OS Simulatorで実行したほうが高速な場合もあれば、低速な場合もあります。したがって、かならず実際のデバイス上で描画コードをプロファイリングして最適化するべきです。iPhone Simulatorが実際のパフォーマンスを反映していると考えるべきではありません。

iPhone Simulator上のOpenGL ES 2.0

サポートされる拡張機能

iPhone SimulatorはOpenGL ES 2.0の次の拡張機能をサポートしています。

- GL_OES_depth24
- GL_OES_mapbuffer
- GL_OES_rgb8_rgba8

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

- GL_IMG_read_format
- GL_IMG_texture_compression_pvrtc
- GL_IMG_texture_format_BGRA8888

iPhone Simulator上のOpenGL ES 1.1

サポートされる拡張機能

- GL_OES_blend_subtract
- GL_OES_compressed_paletted_texture
- GL_OES_depth24
- GL_OES_draw_texture
- GL_OES_framebuffer_object
- GL_OES_mapbuffer
- GL_OES_matrix_palette
- GL_OES_point_size_array
- GL_OES_point_sprite
- GL_OES_read_format
- GL_OES_rgb8_rgba8
- GL_OES_texture_mirrored_repeat
- GL_OES_stencil8
- GL_EXT_texture_filter_anistropic
- GL_EXT_texture_lod_bias
- GL_IMG_texture_compression_pvrtc
- GL_IMG_read_format
- GL_IMG_texture_format_BGRA8888
- GL_APPLE_texture_2D_limited_npot

© 2009 Apple Inc. All Rights Reserved.

textureteoolを使用したテクスチャの圧縮

iPhone SDKには、テクスチャをPVRテクスチャ圧縮フォーマットに圧縮できる`texturetool`という名前のツールが含まれています。iPhone OS 3.0 SDKと一緒にXcodeをデフォルトの場所(`/Developer/Platforms/`)にインストールした場合、texturetoolは次の場所にあります：`/Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/texturetool`。

`texturetool`には、画像の品質とサイズの間のトレードオフに対応するさまざまな圧縮オプションがあります。どの設定が最適な妥協点かを判断するには、テクスチャごとに実験する必要があります。

注： `texturetool`で利用可能なエンコーダ、フォーマット、およびオプションは変更される可能性があります。この文書では、iPhone OS 3.0現在で利用可能なオプションについて説明します。以前のバージョンのiPhone OSと互換性のないオプションについては注釈を付けました。

texturetoolのパラメータ

このセクションでは、`texturetool`に渡すことができるパラメータについて説明します。

```
user$ texturetool -h

Usage: texturetool [-hlm] [-e <encoder>] [-p <preview_file>] -o <output> [-f
<format>] input_image

    -h              Display this help menu.
    -l              List available encoders, individual encoder options, and file
 formats.
    -m              Generate a complete mipmap chain from the input image.
    -e <encoder>          Encode texture levels with <encoder>.
    -p <preview_file>     Output a PNG preview of the encoded output to
<preview_file>.Requires -e option
    -o <output>         Write processed image to <output>.
    -f <format>         Set file <format> for <output> image.
```

注： -pオプションには-eオプションが必要になります。また、-oも必要です。

リスト A-1　エンコードオプション

```
user$ texturetool -l
Encoders:

PVRTC
 --channel-weighting-linear
 --channel-weighting-perceptual
 --bits-per-pixel-2
```


2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

```
 --bits-per-pixel-4

Formats:

Raw
PVR
```

`texturetool`では、何もオプションを付けないとデフォルトで`--bits-per-pixel-4`、`--channel-weighting-linear`、および`-f Raw`になります。

`--bits-per-pixel-2`オプションと`--bits-per-pixel-4`オプションは、元のピクセルを2ビット/ピクセルまたは4ビット/ピクセルにエンコードするPVRTCデータを作成します。これらのオプションは、圧縮なしの32ビットRGBA画像データと比べて、それぞれ16:1と8:1の圧縮率になります。32バイトが最小データサイズです。圧縮処理によってこれより小さいファイルは作成されません。したがって、圧縮されたテクスチャデータをアップロードするときは少なくともこのバイト数はアップロードされます。

`--channel-weighting-linear`を指定して圧縮した場合は、すべてのチャンネルに均等に圧縮誤差が拡散されます。対照的に、`--channel-weighting-perceptual`を指定すると、リニアオプションの場合と比べて緑チャネルの誤差を減らすことができます。一般に、PVRTC圧縮は線画よりも写真画像に効果的です。

`-m`オプションを利用すると元の画像に対するミップマップレベルが自動的に生成されます。このレベルは、作成されるアーカイブ内に追加画像データとして提供されます。Raw画像フォーマットを使用した場合は、各レベルの画像データがアーカイブの最後に順番に追加されます。PVRアーカイブフォーマットを使用した場合は、各ミップマップは独立した画像としてアーカイブに提供されます。

iPhone OS 2.2 SDKでは、出力ファイルのフォーマットを制御できるパラメータ(`-f`)が追加されました。iPhone OS 2.1以前ではこのパラメータは利用できませんが、作成されたデータファイルはこれらのバージョンのiPhone OSと互換性があります。

デフォルトのフォーマットはRawです。これは、iPhone SDK 2.0および2.1の`texturetool`で作成されるフォーマットと同じです。このフォーマットは1つのテクスチャレベル（`-m`オプションなし）、または各テクスチャレベルを連結した（`-m`オプションあり）未加工の圧縮テクスチャデータです。ファイルに格納された各テクスチャレベルのサイズは少なくとも32バイトあり、これがそのままGPUにアップロードされなければなりません。

PVRフォーマットは、Imagination TechnologiesのPowerVR SDKに含まれているPVRTexToolで使われるフォーマットと同じです。アプリケーションでは、このデータのヘッダを解析して実際のテクスチャデータを取得しなければなりません。PVRフォーマットのテクスチャデータを扱う例については、*PVRTextureLoader*サンプルを参照してください。

重要： エンコーダ用のソース画像は次の要件を満たしている必要があります。

- 高さと幅が少なくとも8はある。
- 高さと幅が2のべき乗である。
- 正方形である（高さ==幅）。
- ソース画像は、Mac OS X上でImage IOが受理するフォーマットである。最良の結果を得るには、元のテクスチャは圧縮なしのデータフォーマットであるべきです。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

重要： PVRTexToolを使用してテクスチャを圧縮している場合は、2のべき乗の長さを持つ正方形のテクスチャを作成しなければなりません。アプリケーションが正方形以外のテクスチャや2のべき乗以外のテクスチャをiPhone OSにロードしようとすると、ロードに失敗します。

リスト A-2 画像をPVRTC圧縮フォーマットにエンコードする

```
線形荷重および4 bppを指定してImage.pngをPVRTCにエンコードして、ImageL4.pvrtcとして保存する
user$ texturetool -e PVRTC --channel-weighting-linear --bits-per-pixel-4 -o
ImageL4.pvrtc Image.png

知覚荷重および4 bppを指定してImage.pngをPVRTCにエンコードして、ImageP4.pvrtcとして保存する
user$ texturetool -e PVRTC --channel-weighting-perceptual --bits-per-pixel-4 -o
 ImageP4.pvrtc Image.png

線形荷重および2 bppを指定してImage.pngをPVRTCにエンコードして、ImageL2.pvrtcとして保存する
user$ texturetool -e PVRTC --channel-weighting-linear --bits-per-pixel-2 -o
ImageL2.pvrtc Image.png

知覚荷重および2 bppを指定してImage.pngをPVRTCにエンコードして、ImageP2.pvrtcとして保存する
user$ texturetool -e PVRTC --channel-weighting-perceptual --bits-per-pixel-2 -o
 ImageP2.pvrtc Image.png
```

リスト A-3 プレビューを作成するとともに画像をPVRTC圧縮フォーマットにエンコードする

```
線形荷重および4 bppを指定してImage.pngをPVRTCにエンコードして、出力をImageL4.pvrtcとして
保存し、PNGプレビューをImageL4.pngとして保存する
user$ texturetool -e PVRTC --channel-weighting-linear --bits-per-pixel-4 -o
ImageL4.pvrtc -p ImageL4.png Image.png

知覚荷重および4 bppを指定してImage.pngをPVRTCにエンコードして、出力をImageP4.pvrtcとして
保存し、PNGプレビューをImageP4.pngとして保存する
user$ texturetool -e PVRTC --channel-weighting-perceptual --bits-per-pixel-4 -o
 ImageP4.pvrtc -p ImageP4.png Image.png

線形荷重および2 bppを指定してImage.pngをPVRTCにエンコードして、出力をImageL2.pvrtcとして
保存し、PNGプレビューをImageL2.pngとして保存する
user$ texturetool -e PVRTC --channel-weighting-linear --bits-per-pixel-2 -o
ImageL2.pvrtc -p ImageL2.png Image.png

知覚荷重および2 bppを指定してImage.pngをPVRTCにエンコードして、出力をImageP2.pvrtcとして
保存し、PNGプレビューをImageP2.pngとして保存する
user$ texturetool -e PVRTC --channel-weighting-perceptual --bits-per-pixel-2 -o
 ImageP2.pvrtc -p ImageP2.png Image.png
```

注： -oパラメータと有効な出力ファイルを指定しないと、プレビューは作成できません。プレビュー画像は常にPNG形式になります。

リスト A-4 PVRTCデータをグラフィックチップにアップロードする例

```
void texImage2DPVRTC(GLint level, GLsizei bpp, GLboolean hasAlpha, GLsizei width,
 GLsizei height, void *pvrtcData)
```


2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

```
{
    GLenum format;
    GLsizei size = width * height * bpp / 8;
    if(hasAlpha) {
        format = (bpp == 4) ?GL_COMPRESSED_RGBA_PVRTC_4BPPV1_IMG
:GL_COMPRESSED_RGBA_PVRTC_2BPPV1_IMG;
    } else {
        format = (bpp == 4) ?GL_COMPRESSED_RGB_PVRTC_4BPPV1_IMG
:GL_COMPRESSED_RGB_PVRTC_2BPPV1_IMG;
    }
    if(size < 32) {
        size = 32;
    }
   glCompressedTexImage2D(GL_TEXTURE_2D, level, format, width, height, 0, size,
 data);
}
```

サンプルコードとしては、*PVRTextureLoader*サンプルを参照してください。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

書類の改訂履歴

この表は「*iPhone OpenGL ESプログラミングガイド*」の改訂履歴です。

日付	メモ
2009-06-11	iPhoneアプリケーション内に高いパフォーマンスのグラフィックスを作成するためにOpenGL ES 1.1および2.0のプログラミングインターフェイスを使用する方法を説明した文書の初版。

2009-06-11 | © 2009 Apple Inc. All Rights Reserved.

書類の改訂履歴

© 2009 Apple Inc. All Rights Reserved.