iOS OpenGL ESプログラミングガイド

2010-11-15

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/

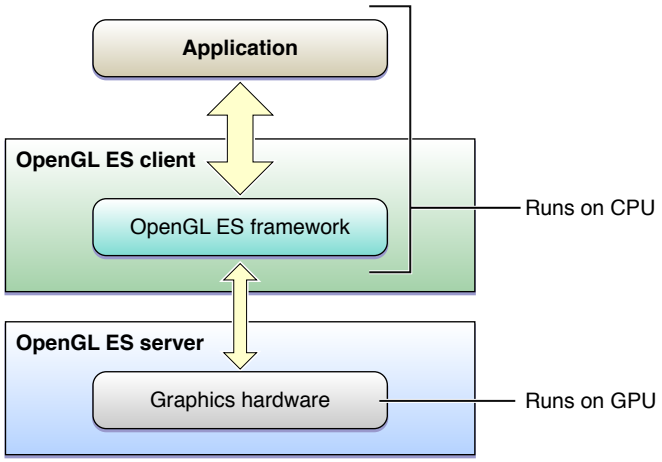
目次

図、表、リスト

OpenGL ESについて

Open Graphics Library (OpenGL)は、2Dおよび3Dのデータを視覚化するために使用します。OpenGLは、オープンスタンダードに基づいた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を使用することで、アプリケーションデベロッパは3Dグラフィックスパイプラインを設定し、そのパイプラインにデータを送信することができます。頂点は、変形およびライティングされ、プリミティブとして組み立てられ、ラスタ化されて2D画像が作成されます。OpenGLは関数呼び出しを、基盤となるグラフィックスハードウェアに送信可能なグラフィックスコマンドに変換するために設計されています。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 for Embedded Systems (OpenGL ES)は、OpenGLの冗長な機能を取り除いた簡略版で、OpenGLよりも覚えやすく、モバイルグラフィックスハードウェアに実装しやすいライブラリを提供します。



概要

AppleはOpenGL ES 1.1およびOpenGL ES 2.0の両方の実装を提供しています。

- OpenGL ES 1.1は、明確に定義された**固定機能パイプライン**を実装します。固定機能パイプラインは、従来型のライティングとシェーディングのモデルを実装しています。このモデルにより、パイプラインの各段階を、特定のタスクを実行するよう設定したり、機能が不要な場合に無効にしたりすることができます。

- OpenGL ES 2.0は、多くの関数がOpenGL ES 1.1と共通していますが、固定機能パイプラインの頂点とフラグメントに関する段階を対象とするすべての関数が削除されています。その代わり、OpenGL ES 2.0は、**シェーダ**ベースの汎用パイプラインへのアクセスを提供する新たな関数を採用しています。シェーダを利用すると、グラフィックスハードウェア上で直接実行される、独

自の頂点関数やフラグメント関数を作成できます。シェーダによって、アプリケーションはパイプラインへの入力、および各頂点と各フラグメントについて実行される演算をより正確に、そしてより明確にカスタマイズすることができます。

OpenGL ESはプラットフォームに依存しないC言語ベースのAPI

OpenGL ESはC言語ベースのAPIであるため、移植性が非常に高く、幅広くサポートされています。OpenGL ESは、C言語のAPIとして、Objective-CベースのCocoa Touchアプリケーションとシームレスに統合します。OpenGL ESの仕様はウインドウレイヤを定義しておらず、ホストオペレーティングシステムがOpenGL ES**レンダリングコンテキスト**（コマンドを受理）と**フレームバッファ**（描画コマンドの結果の書き込み先）を作成する関数を提供しなければなりません。

関連する章： 「iOSにおけるOpenGL ES」（15 ページ）、「OpenGL ESコンテキストの設定」（23 ページ）

OpenGL ESとCore Animationとの統合

Core Animationは、iOSグラフィックスの土台となるもので、アプリケーションが画面に提供するOpenGL ESコンテンツが含まれます。OpenGL ESを開発する場合、OpenGL ESコンテンツは`CAEAGLLayer`オブジェクトと呼ばれる、特別なCore Animationレイヤにレンダリングされます。OpenGL ESを使用してレンダリングされた画像は、`CAEAGLLayer`に格納されます。Core Animationはこれらの画像と、他のレイヤに格納されたコンテンツとを合成し、最終画像を画面に提供します。

関連する章： 「OpenGL ESによる描画」（31 ページ）

フレームバッファオブジェクトが常にレンダリング先

OpenGL ESの仕様は、システムに用意されているフレームバッファおよびフレームバッファオブジェクトという2種類のフレームバッファが存在することを前提としています。システムフレームバッファは、ホストオペレーティングシステムが提供するAPIを使用して割り当てられ、画面またはウインドウ環境に接続します。一方、フレームバッファオブジェクトはオフスクリーンのレンダリングを対象としており、OpenGL ES仕様内で完全にサポートされています。iOSはフレームバッファオブジェクトのみを使用します。iOSは独立したシステムフレームバッファを作成する代わりに、OpenGL ES APIを拡張して、コンテンツがCore Animationと共有されるようにフレームバッファオブジェクトを割り当て可能にします。

関連する章： 「OpenGL ESによる描画」 （31 ページ）

機能はデバイスによって異なる

iOSデバイスは、機能が異なるさまざまなグラフィックスプロセッサに対応しています。いくつかのデバイスはOpenGL ES 1.1とOpenGL ES 2.0の両方に対応していますが、古いデバイスはOpenGL ES 1.1のみに対応しています。デバイスは、OpenGL ESの同じバージョンに対応している場合であっても、iOSのバージョンおよび基盤となるグラフィックスハードウェアによって機能が異なる場合があります。Appleは、多くのOpenGL ES拡張機能を用意してアプリケーションに新たな機能を提供します。

アプリケーションをできるだけ多くのデバイス上で実行できるようにしたり、今後登場するデバイスやiOSのバージョンとの互換性を確保したりするため、基盤となるOpenGL ES実装の機能を、iOSによるサポートが必須でない機能を無効にした状態で、実行時に必ずテストしなければなりません。

関連する章： 「OpenGL ESコンテキストの設定」 （23 ページ）、「OpenGL ESの機能の判定」 （27 ページ）、「プラットフォーム関連の情報」 （89 ページ）

アプリケーションに必要な追加のパフォーマンスチューニング

グラフィックスプロセッサは、グラフィックス操作向けに最適化された並行処理デバイスです。アプリケーションに優れたパフォーマンスを発揮させるには、グラフィックスハードウェアがアプリケーションと並行で動作するよう、アプリケーションを注意深く設計してデータとコマンドをOpenGL ESに供給しなければなりません。チューニングが十分でないアプリケーションは、一方がコマンドの処理を完了するまでCPUまたはGPUのどちらかを待機させることになります。

OpenGL ES APIを効率よく使用してアプリケーションを設計してください。アプリケーションのビルドが完了したら、Instrumentsを使用してアプリケーションのパフォーマンスを微調整します。OpenGL ES内でアプリケーションのボトルネックが発生している場合は、このガイドの情報を使用してアプリケーションのパフォーマンスを最適化してください。

関連する章： 「OpenGL ESアプリケーション設計ガイドライン」 （47 ページ）、「頂点データを扱うためのベストプラクティス」 （67 ページ）、「テクスチャデータを扱うためのベストプラクティス」 （79 ページ）、「シェーダのベストプラクティス」 （83 ページ）、「OpenGL ESアプリケーションのチューニング」 （57 ページ）、「プラットフォーム関連の情報」 （89 ページ）

バックグラウンドアプリケーションでOpenGL ESを使用できない可能性について

バックグラウンドで実行されるアプリケーションはOpenGL ES関数を呼び出さない場合があります。アプリケーションがバックグラウンド時にグラフィックスプロセッサにアクセスすると、iOSがアプリケーションを自動的に終了します。これを回避するには、バックグラウンドへの移行前にOpenGL ESに送信された未処理のコマンドをフラッシュし、アプリケーションがフォアグラウンドに戻るまでの間OpenGL ESを呼び出さないようにする必要があります。

関連する章： 「マルチタスク対応OpenGL ESアプリケーションの実装」（45 ページ）

OpenGL ESによるマルチスレッドアプリケーションへの制約事項

並列処理を利用するようアプリケーションを設計することは、アプリケーションパフォーマンスの向上に役立つ可能性があります。OpenGL ESアプリケーションに並列処理を追加する場合は、アプリケーションが、異なる2つのスレッドから同じコンテキストに同時にアクセスしないようにしなければなりません。

関連する章： 「並列処理とOpenGL ES」（61 ページ）

この文書の使いかた

OpenGL ESまたはiOSの開発が初めての場合は、まず「iOSにおけるOpenGL ES」（15 ページ）をお読みください。この章は、OpenGL ESのアーキテクチャの概要およびOpenGL ESがどのようにiOSに統合されているかを説明しています。そして、残りの章を順番にお読みください。

経験豊富なiOSデベロッパの方は、「OpenGL ESによる描画」（31 ページ）をお読みください。OpenGL ESをアプリケーションに統合する方法およびレンダリングループを正しく実装する方法の詳細について知ることができます。さらに、「OpenGL ESアプリケーション設計ガイドライン」（47 ページ）では、効率的なOpenGL ESアプリケーションを設計する方法について詳しく説明しています。「プラットフォーム関連の情報」（89 ページ）では、利用可能なハードウェアおよびアプリケーションから利用可能なソフトウェアについての詳細情報が記されています。

お読みになる前に

OpenGL ESアプリケーションを作成する前に、ビューおよびビューとCore Animationとのやり取りについて精通している必要があります。詳細については『*View Programming Guide for iOS*』を参照してください。

この文書は、OpenGL ESの重要な概念を紹介していますが、OpenGL ES APIのチュートリアルでもリファレンスでもありません。OpenGL ESアプリケーションの作成に関する詳細については、以下に示す参考文献をご覧ください。

関連項目

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。

- 『*OpenGL® ES 2.0 Programming Guide*』（Addison-Wesley刊行）は、OpenGL ESの概念についての総合的な入門書です。

- 『*OpenGL® Shading Language, Third Edition*』（Addison-Wesley刊行）は、OpenGL ESアプリケーションで使用可能な多くのシェーディングアルゴリズムを提供しています。これらのアルゴリズムの一部は、モバイルグラフィックスプロセッサ上で効率よく実行するために修正が必要となる場合があります。
- OpenGL ES API Registryは、OpenGL ES 1.1仕様、OpenGL ES 2.0仕様、OpenGL ESシェーディング言語仕様、およびOpenGL ES拡張機能に関する文書の公式リポジトリです。
- OpenGL ES 1.1 Reference Pagesは、OpenGL ES 1.1仕様の詳細なリファレンスで、アルファベット順の索引があります。
- OpenGL ES 2.0 Reference Pagesは、OpenGL ES 2.0仕様の詳細なリファレンスで、アルファベット順の索引があります。
- 『*OpenGL ES Framework Reference*』は、OpenGL ESをiOSに統合するためAppleが提供する、プラットフォーム固有の関数とクラスを解説しています。

iOSにおけるOpenGL ES

OpenGL ESは、ハードウェアで高速化されたグラフィックスパイプラインによってレンダリングされるプリミティブを送信する、手続き型APIを提供します。グラフィックスコマンドはOpenGLによって解釈され、ユーザに表示する画像や、OpenGL ES外部でのさらなる処理のための画像が生成されます。

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のほとんどのコマンドは、次のいずれかのアクティビティを実行します。

- 機能について実装が提供する詳細情報の読み取り。詳細については、「OpenGL ESの機能の判定」（27 ページ）を参照してください。
- 仕様で定義されている状態変数の読み取りおよび書き込み。OpenGL ESの状態は通常、グラフィックスパイプラインの現在の設定を表しています。たとえば、OpenGL ES 1.1は、光源、素材、および固定機能パイプラインが実行する演算を設定するために幅広く状態変数を使用します。
- OpenGL ESオブジェクトの作成、変更、または破棄。OpenGL ESオブジェクトは、OpenGL ESの手続き型APIを通じて操作するOpenGL ESリソースであり、Objective-Cオブジェクトではありません。詳細については、「OpenGL ESオブジェクトはアプリケーションに代わってリソースをカプセル化」（18 ページ）を参照してください。
- 描画プリミティブ。頂点は、処理を行ってプリミティブを組み立て、**フレームバッファ**にラスタ化するパイプラインに送信されます。

どのバージョンのOpenGL ESをターゲットとするべきか

OpenGL ESアプリケーションを設計するときに、明確にしておかなければならない重要なことは、アプリケーションでOpenGL ES 2.0とOpenGL ES 1.1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 2.0はOpenGL ES 1.1よりも柔軟性が高く機能も豊富で、新たなアプリケーションにとって最良の選択です。シェーダでは、独自の頂点計算やフラグメント計算をより明確かつ簡潔に実装できパフォーマンスも向上します。同じ計算をOpenGL ES 1.1で実行するには、アルゴリズムの意図がわかりにくくなるようなマルチレンダリングパスや複雑な状態設定が必要となることが少なくありません。アルゴリズムが複雑になるにつれて、シェーダのほうが計算を明確かつ簡潔に表現でき、パフォーマンスも優れています。OpenGL ES 2.0は、アプリケーションをビルドするためにより多くの作業が事前に必要です。OpenGL ES 1.1がデフォルトで提供する一部のインフラストラクチャは作成し直す必要があります。
- OpenGL ES 1.1には、頂点の変形やライティングから、フラグメントとフレームバッファのブレンドに至るまで、3Dアプリケーション向けの優れた基本動作を提供する、標準の固定機能パイプラインが用意されています。アプリケーションに単純さを求める場合、OpenGL ES 1.1は、ア

プリケーションにOpenGL ESサポートを追加するためのコーディングがより少なくなります。アプリケーションがすべてのiOSデバイスに対応する必要がある場合は、OpenGL ES 1.1をアプリケーションのターゲットにする必要があります。

主に古いデバイスとの互換性を確保するためにOpenGL ES 1.1アプリケーションの実装を選ぶ場合、新しいiOSデバイスに搭載されるOpenGL ES 2.0対応グラフィックスプロセッサの能力を利用するOpenGL ES 2.0レンダリングオプションの追加を検討してください。

重要： アプリケーションがOpenGL ES 1.1とOpenGL ES 2.0レンダリングパスの両方ともサポートしない場合は、指定したバージョンをサポートするデバイスのみで実行するようにアプリケーションを限定してください。詳細については、『*iOS Application Programming Guide*』の「Declaring the Required Device Capabilities」を参照してください。

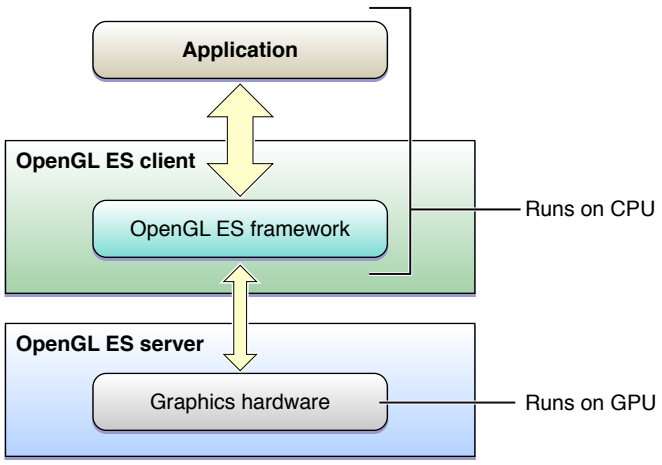
OpenGL ESアーキテクチャの理解

OpenGL ESはいくつかの主要な原則に基づいて機能します。効率的なOpenGL ESアプリケーションを設計するには、基盤アーキテクチャを理解する必要があります。

クライアントサーバモデル

OpenGL ESは、図 1-1に示すように、クライアントサーバモデルを使用しています。アプリケーションは、OpenGL ES関数を呼び出すとき、OpenGL ESとやり取りします。OpenGLクライアントは関数呼び出しを処理し、必要な場合はコマンドに変換して、OpenGLサーバに送信します。クライアントサーバモデルでは、クライアントとサーバの間で関数呼び出しを分割して処理することができます。クライアント、サーバ、およびクライアントとサーバの間の通信経路の性質は、OpenGL ES実装ごとに固有のものです。iOSでは、作業負荷は、CPUと専用のグラフィックスプロセッサの間で分割されます。

図 1-1 OpenGL ESクライアントサーバモデル



重要な機能にはプラットフォーム固有のライブラリを使用

OpenGL ES仕様では、OpenGL ESの動作を定義していますが、OpenGL ESとホストオペレーティングシステムとの間のやり取りを管理する関数は定義していません。その代わりに、OpenGL ES仕様は、**レンダリングコンテキスト**および**システムフレームバッファ**を割り当てる関数の提供を個々の実装に求めています。

レンダリングコンテキストには、OpenGL ES状態マシンの内部状態が格納されます。レンダリングコンテキストを使用することで、各OpenGL ESアプリケーションはほかのアプリケーションを気にせずに状態データのコピーを保持することが可能となります。詳細については、「OpenGL ESコンテキストの設定」（23 ページ）を参照してください。複数のレンダリングコンテキストを単一のアプリケーションで使用することもできます。

システムフレームバッファは、OpenGL ES描画コマンドの送信先であり、通常、ホストオペレーティングシステムのグラフィックスサブシステムに関連付けられています。iOSにはシステムフレームバッファありません。その代わりに、iOSは、OpenGL ESが提供するフレームバッファオブジェクトを拡張して、Core Animationとデータを共有するフレームバッファを実現しています。フレームバッファオブジェクトの詳細については「フレームバッファオブジェクトはiOSにおけるただ1つのレンダリングターゲット」（20 ページ）を、アプリケーションにおけるフレームバッファの作成と使用の詳細については「OpenGL ESによる描画」（31 ページ）を参照してください。

コマンドは非同期で実行される可能性がある

OpenGL ESクライアントサーバモデルの利点は、OpenGL ES関数は、要求された操作が完了する前に制御をアプリケーションに返すことができるということです。関数が完了してからアプリケーションに制御を戻すようOpenGL ESがどの関数にも要求したとすると、CPUとGPUの実行順の融通が利かなくなり、アプリケーションで並行処理を行う多くの機会が失われます。iOSでは、描画コマンドの実行を遅延することはごく一般的です。いくつかの描画コマンドの実行を遅らせて同時に処理することにより、グラフィックスハードウェアは負荷の大きいフラグメント計算を実行する前に、非表示サーフェスを削除することができます。

OpenGL ES関数の多くは、グラフィックスハードウェアに対して暗黙的または明示的にコマンドをフラッシュします。ほかのOpenGL関数は、グラフィックスプロセッサにコマンドをフラッシュして、一部またはすべての未処理コマンドが完了するまで待機します。できるだけ、クライアントとサーバの同期を回避するアプリケーションを設計します。

コマンドは順番に実行される

OpenGL ESは、レンダリングコンテキストに対して行われた関数呼び出しの結果を、あたかもクライアントアプリケーションが関数を呼び出したのと同じ順序で実行されたように処理することを保証します。アプリケーションがOpenGL ESに対して関数呼び出しを行うとき、アプリケーションは、一部コマンドの実行か完了していない場合でも、それ以前の関数の結果が利用可能であるとみなすことができます。

パラメータは呼び出し時にコピーされる

コマンドを非同期で処理できるようにするため、OpenGL ES関数をアプリケーションが呼び出すとき、関数呼び出しの完了に必要なパラメータデータは、制御がアプリケーションに返される前にOpenGL ESがコピーしなければなりません。アプリケーションメモリに格納される頂点データの配列をパラメータが指し示している場合、OpenGL ESは返される前に頂点データをコピーしなければなりません。これには2つの重要な意味があります。1つは、OpenGL ESとアプリケーションは同じメモリに同時にアクセスすることがないため、アプリケーションは、OpenGL ESに対して行う呼び出しに関係なく、所有するメモリを自由に変更できるということです。もう1つは、グラフィックスハードウェアがデータを読み取ることができるようパラメータをコピーし、データの書式を変えることで、関数呼び出しそれぞれにオーバーヘッドが付加されるということです。最良のパフォーマンスを得るため、アプリケーションは、グラフィックスハードウェアに最適化された形式でアプリケーションデータを定義します。そして、バッファオブジェクトを使用して、アプリケーションとOpenGL ESとの間のメモリコピーを明示的に管理します。

仕様で規定された機能は実装で拡張可能

OpenGL ES実装は、次の2つの方法のどちらかでOpenGL ES仕様を拡張する場合があります。1つは、テクスチャのサイズ、またはアプリケーションがアクセスする可能性のあるテクスチャユニットの数など、実装が満たさなければならない固有の最小要件を仕様が定める方法です。OpenGL ES実装では、より大きなテクスチャ、あるいはより多くのテクスチャユニットというように、より大きな値を使用できます。もう1つは、OpenGL ES**拡張機能**により、実装においてOpenGL ESの新たな関数と定数を提供できるようにする方法です。拡張機能により、実装においてまったく新しい機能を追加することができます。Appleは多くの拡張機能を実装することで、アプリケーションがハードウェア機能を利用できるようにしたり、アプリケーションのパフォーマンス向上に役立つようにしたりしています。実際のハードウェアの制限、および各実装が提供する拡張機能のリストはアプリケーションを実行するデバイスやデバイス上で実行されるiOSのバージョン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、適合するようアプリケーションの動作を変更しなければなりません。

OpenGL ESオブジェクトはアプリケーションに代わってリソースをカプセル化

オブジェクトは、レンダラが必要とする設定の状態やデータを保持するためにアプリケーションが使用する、不透過のコンテナです。オブジェクトへの唯一のアクセスは手続き型APIを経由するため、OpenGL ESの実装ではアプリケーション用のオブジェクトを割り当てるときにさまざまな戦略を選ぶことができます。アプリケーションは、グラフィックスプロセッサに適した形式で、またはグラフィックスプロセッサに適したメモリ領域にデータを格納できます。オブジェクトのもう1つの利点は、*再利用可能*であることで、それによりアプリケーションは一度設定したオブジェクトを何度でも使用することが可能となります。

最も重要なOpenGL ES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。これは通常、カラー画像をプリミティブ上にマッピングするために使用しますが、法線マップや事前に計算したライティング情報など、ほかのデータのマッピングにも使用できます。「テクスチャデータを扱うためのベストプラクティス」（79 ページ）の章では、iOSでのテクスチャの使用に関する重要なトピックを取り上げています。

- **バッファ**オブジェクトは、OpenGL ESが所有するメモリのブロックで、アプリケーションのデータを格納するために使用します。バッファは、アプリケーションとOpenGL ESの間におけるデータのコピープロセスを正確に制御するために使用します。たとえば、OpenGL ESに頂点配列を提供する場合、アプリケーションは描画呼び出しを送信するごとにデータをコピーしなければなりません。対照的に、アプリケーションがデータを*頂点バッファオブジェクト*に格納する場合、頂点バッファオブジェクトの内容を変更するコマンドをアプリケーションが送信するときだけデータがコピーされます。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。

- **頂点配列**オブジェクトは、グラフィックスパイプラインが読み取る頂点属性の設定を保持します。多くのアプリケーションが、レンダリングするエンティティごとに異なるパイプライン設定を必要とします。頂点配列に設定を格納することにより、パイプラインを設定し直さずにすみ、実装において、特定の頂点設定の処理を最適化できる場合があります。

- **シェーダプログラム**もオブジェクトであり、単にシェーダとも呼ばれます。OpenGL ES 2.0アプリケーションは、頂点シェーダとフラグメントシェーダを作成して、各頂点または各フラグメントに対し、それぞれに実行する演算を指定します。

- **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通常、色データ、深度データ、またはステンシルデータとして定義されます。レンダバッファは通常、分離して使用されることはなく、フレームバッファに対するアタッチメントとして使用されます。

- **フレームバッファ**オブジェクトは、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、レンダラが必要とする詳細な設定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。後の章の「OpenGL ESによる描画」（31 ページ）では、iOSアプリケーションにおいてフレームバッファを作成、使用するための方法について説明します。

OpenGL ESの各オブジェクトタイプはそれぞれを操作するための固有の関数を持っていますが、すべてのオブジェクトが次に示す同様のプログラミング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。

   識別子は、特定のオブジェクトインスタンスを識別するために使用する、わかりやすい整数です。新たなオブジェクトが必要な場合は常に、OpenGL ESを呼び出して新たな識別子を作成します。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わけではなく、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。

   ほとんどのOpenGL ES関数は、各関数呼び出しにおいてオブジェクトの識別を明示的に求めるのではなく、オブジェクトに対して暗黙的に働きます。オブジェクトは、コンテキストにオブジェクトを*バインド*することによって設定します。オブジェクトタイプはそれぞれ、異なる関数を使用してオブジェクトをコンテキストにバインドします。OpenGL ESは、オブジェクト識別子を最初にバインドするときにオブジェクトの割り当てと初期化を行います。

3. オブジェクトの状態を変更する。

   アプリケーションは、1つ以上の関数呼び出しを行ってオブジェクトを設定します。たとえば、テクスチャオブジェクトをバインドした後は通常、テクスチャのフィルタリング方法を設定し、その後テクスチャオブジェクトに画像データをロードします。

オブジェクトの変更は、グラフィックスハードウェアに新たなデータの送信が必要となる可能性があるため、負荷が増大するおそれがあります。一度オブジェクトを作成し設定したら、その後はアプリケーションを実行している間オブジェクトの変更を避けるようにします（こうした方が理にかなっている場合）。

4. レンダリングのためにオブジェクトを使用する。

   シーンのレンダリングに必要なすべてのオブジェクトを作成し、設定したら、パイプラインが必要とするオブジェクトをバインドし、1つ以上の描画関数を実行します。OpenGL ESは、オブジェクトに格納されているデータを使用してプリミティブをレンダリングします。結果は、バインドされたフレームバッファオブジェクトに送られます。

5. オブジェクトを削除する。

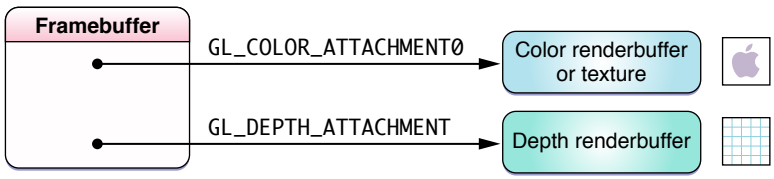
   オブジェクトを使い終わったら、アプリケーションがそのオブジェクトを削除します。オブジェクトが削除されると、その内容は破棄され、オブジェクト識別子はリサイクルされます。

フレームバッファオブジェクトはiOS上で唯一のレンダリングターゲット

フレームバッファオブジェクトは、レンダリングコマンドの送信先です。OpenGL ES 2.0は、中核仕様の一部としてフレームバッファオブジェクトを提供します。OpenGL ES 1.1では、`OES_framebuffer_object`拡張機能によってフレームバッファオブジェクトが提供されます。フレームバッファオブジェクトは、iOSにおけるただ1つのレンダリングターゲットであるため、Appleは、iOSにおけるあらゆるOpenGL ES 1.1実装で常に`OES_framebuffer_object`拡張機能が提供されることを保証します。

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 1-2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オブジェクトです。ただし、OpenGL ESテクスチャはフレームバッファのカラーアタッチメントポイントにアタッチすることができ、これにより画像を直接テクスチャにレンダリングすることができます。画像がレンダリングされたテクスチャは、その後使用されるレンダリングコマンドの入力の役割を果たすことができます。

図 1-2　色と深度のレンダバッファを持つフレームバッファ



フレームバッファは次の手順を使用して作成します。

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. 画像を生成、バインドし、設定します。
3. フレームバッファのアタッチメントポイントの1つに画像をアタッチします。

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。完全性のためのルールはOpenGL ES仕様において定義されています。これらのルールにより、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ます。

OpenGL ESコンテキストの設定

OpenGL ESのどの実装も、OpenGL ES仕様で求められる状態を管理するためのレンダリングコンテキストを作成する手段を提供します。この状態をコンテキスト内に配置することにより、ほかのアプリケーションの状態に干渉することなく、複数のアプリケーションでのグラフィックスハードウェアを容易に共有できます。

この章では、iOS上でコンテキストを作成し、設定する方法を詳しく説明します。

EAGLコンテキストとはOpenGL ESレンダリングコンテキストのiOSにおける実装

アプリケーションがOpenGL ES関数を呼び出せるようにするには、アプリケーションが`EAGLContext`オブジェクトを初期化し、このオブジェクトを**現在のコンテキスト**として設定しなければなりません。`EAGLContext`クラスも、アプリケーションがOpenGL ESコンテンツをCore Animationに統合するために使用する手段を提供します。これらの手段がなければ、アプリケーションはオフスクリーン画像の操作しかできません。

現在のコンテキストはスレッドにおけるOpenGL ES関数呼び出しのターゲットとなる

iOSアプリケーションのどのスレッドも、現在のコンテキストを保持しています。アプリケーションがOpenGL ES関数を呼び出すとき、そのスレッドのコンテキストは関数呼び出しによって変更されます。

現在のコンテキストを設定するには、`EAGLContext`クラスメソッドである`setCurrentContext:`を呼び出します。

```
[EAGLContext setCurrentContext:myContext];
```

アプリケーションは、`EAGLContext`クラスメソッドの`currentContext`を呼び出すことにより、スレッドの現在のコンテキストを取得できます。

アプリケーションが新しいコンテキストを設定すると、EAGLは前のコンテキストを解放（前のコンテキストがある場合）し、新しいコンテキストを保持します。

注： アプリケーションが同一スレッド上の2つ以上のコンテキストをアクティブに切り替える場合、新しいコンテキストを現在のコンテキストに設定する前に、`glFlush`関数を呼び出します。これにより、事前に送信されていたコマンドは、適切なタイミングでグラフィックスハードウェアに配信されます。

どのコンテキストもOpenGL ESの特定のバージョンをターゲットにする

`EAGLContext`オブジェクトは、OpenGL ES 1.1またはOpenGL ES 2.0のどちらかをサポートしますが、両方をサポートすることはありません。この理由は、OpenGL ES 2.0の設計にあります。OpenGL ES 2.0は、OpenGL ES 1.1に存在した、固定機能グラフィックスパイプラインを扱うすべての関数を削除し、シェーダベースのクリーンなインターフェイスを提供する多くの新しい関数を追加しました。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。

アプリケーションは、`EAGLContext`オブジェクトを作成し、初期化するときに、どのバージョンのOpenGL ESをサポートするかを決定します。OpenGL ES 2.0コンテキストを作成するには、次のようにしてアプリケーションがEAGLContextオブジェクトを初期化します

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

アプリケーションでOpenGL ES 1.1を使用したい場合は、別の定数を使用して初期化します。

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

要求されたバージョンのOpenGL ESをデバイスがサポートしていない場合、`initWithAPI:`メソッドは`nil`を返します。アプリケーションは、コンテキストが正常に初期化されたかを確認するためコンテキストを使用する前に必ず確認します。

OpenGL ES 2.0およびOpenGL ES 1.1の両方のレンダリングオプションをサポートするには、アプリケーションが最初にOpenGL ES 2.0レンダリングコンテキストの初期化を試みる必要があります。返されたオブジェクトが`nil`の場合は、OpenGL ES 1.1コンテキストを初期化します。リスト 2-1では、この方法を紹介しています。

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

コンテキストのAPIプロパティは、コンテキストがサポートするOpenGL ESのバージョンを示します。アプリケーションは、コンテキストのAPIプロパティをテストし、そのプロパティを使用して正しいレンダリングパスを選びます。これを実装する一般的なパターンは、各レンダリングパスについてクラスを作成するというものです。アプリケーションは、初期化時にコンテキストをテストし、一度レンダラを作成します。

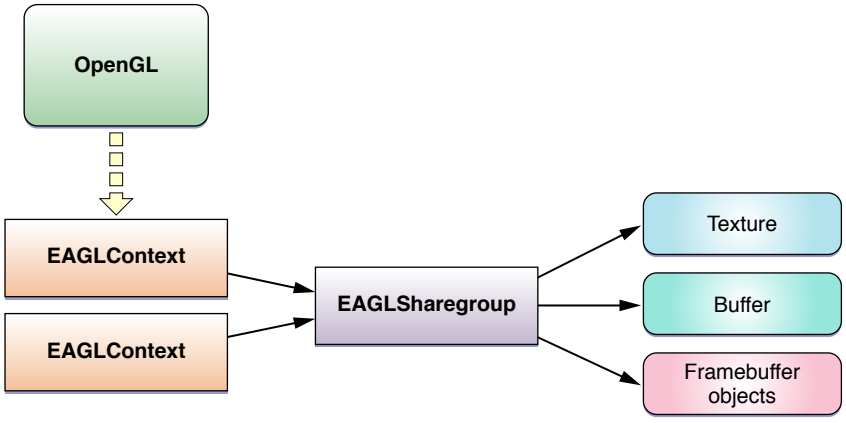
EAGL sharegroupはコンテキストのOpenGL ESオブジェクトを管理

コンテキストはOpenGL ESの状態を保持しますが、OpenGL ESオブジェクトを直接管理するわけではありません。OpenGL ESオブジェクトは、EAGLSharegroupオブジェクトによって作成され、維持されます。どのコンテキストにもオブジェクトの作成をデリゲートするEAGLSharegroupオブジェクトが含まれます。

sharegroupの利点は、2つ以上のコンテキストが同じsharegroupを参照する場合に明らかになってきます（図 2-1参照）。複数のコンテキストが共通のsharegroupに接続されている場合、何らかのコンテキストで作成したOpenGL ESオブジェクトはすべてのコンテキスト上で利用できます。オブジェクトを作成したコンテキストではなく、別のコンテキスト上で同じオブジェクト識別子にバインドすると、*同じ*OpenGL ESオブジェクトを参照します。リソースは、モバイルデバイスでは希少な場合が少なくありません。複数のコンテキストにおいて同じコンテンツのコピーを複数作成するのは非効率的です。共通のリソースを共有することで、デバイス上のグラフィックスリソースの空きをより効果的に利用できます。

アプリケーションは、sharegroupを不透過オブジェクトとして扱うことができます。sharegroupをアプリケーションが呼び出せるメソッドやプロパティはなく、参照元のコンテキストによって自動的に保持されたり、解放されたりします。

図 2-1　2つのコンテキストによるOpenGL ESオブジェクトの共有



sharegroupは、次に示す2つの具体的シナリオで最も有用です。

- コンテキスト間で共有されているリソースのほとんどが変更されない場合。

- レンダラのメインスレッド以外のスレッド上に、アプリケーションが新たなOpenGL ESオブジェクトを作成できるようにしたい場合。この場合、2つ目のコンテキストは独立したスレッド上で実行され、データのフェッチとリソースの作成に充てられます。リソースがロードされた後、最初のコンテキストはオブジェクトにバインドしてすぐに使用できます。

同じsharegroupを参照する複数のコンテキストを作成するには、最初のコンテキストを`initWithAPI:`を呼び出して初期化します。sharegroupはそのコンテキストに対して自動的に作成されます。2つ目以降のコンテキストは、`initWithAPI:sharegroup:`メソッドを呼び出すことにより、最初のコンテキストのsharegroupを使用するよう初期化されます。リスト2-2では、この仕組みについて説明しています最初のコンテキストは、リスト 2-1 （24 ページ）で定義される、簡易関数を使用して作成されます。2つ目のコンテキストは、最初のコンテキストからAPIのバージョンとsharegroupを抽出して作成されます。

リスト 2-2　共通のsharegroupを持つ2つのコンテキストの作成

```
EAGLContext* firstContext = CreateBestEAGLContext();
EAGLContext* secondContext = [[EAGLContext alloc] initWithAPI:[firstContext API]
 sharegroup:[firstContext sharegroup]];
```

sharegroupが複数のコンテキストによって共有されている場合、OpenGL ESオブジェクトに生じる状態の変化の管理はアプリケーションの責任です。変化の管理に関する規則は次のとおりです。

- オブジェクトの修正中ではない場合、複数のコンテキストにわたって同時にそのオブジェクトにアクセスすることが可能。
- コンテキストに送信されたコマンドによってオブジェクトが修正されている間、オブジェクトを読み取ったり、ほかのコンテキスト上でオブジェクトを修正したりしてはならない。
- オブジェクトが修正された後、すべてのコンテキストがオブジェクトを再バインドして変更を認識する必要がある。オブジェクトのコンテンツは、コンテキストがバインドする前にオブジェクトを参照している場合は未定義である。

OpenGL ESオブジェクトを更新するためにアプリケーションが従う手順は次のとおりです。

1. オブジェクトを使用する可能性のあるコンテキストごとに`glFlush`を呼び出す。
2. オブジェクトを修正するコンテキスト上で、1つ以上のOpenGL ES関数を呼び出してオブジェクトに変更を加える。
3. 状態変更コマンドを受信するコンテキスト上で`glFlush`を呼び出す。
4. そのほかの各コンテキスト上で、オブジェクト識別子を再バインドする。

注： オブジェクトを共有するもう1つの方法は、単一のレンダリングコンテキストではなく、複数のレンダリング先フレームバッファを使用することです。レンダリング時において、アプリケーションは適切なフレームバッファをバインドし、必要に応じてフレームをレンダリングします。すべてのOpenGL ESオブジェクトは単一のコンテキストから参照されるため、OpenGL ESオブジェクトは同一のOpenGL ESデータを認識しています。この方法はリソースはあまり使用しませんが、コンテキストの状態を細かく制御可能な単一スレッドのアプリケーションでしか有用ではありません。

OpenGL ESの機能の判定

OpenGL ES 1.1仕様およびOpenGL ES 2.0仕様のどちらも、各実装がサポートしなければならない最小限の標準を規定していますが、実装において、規定されたそれらの機能だけに制限されるということはありません。OpenGL ESの仕様では、実装においてさまざまな方法で機能を拡張することが認められています。後の章の「プラットフォーム関連の情報」（89 ページ）では、iOSが提供する各OpenGL ES実装の具体的な機能について詳しく説明します。実装における詳細な機能は、実装する仕様のバージョン、基盤となるグラフィックスハードウェア、およびデバイス上で実行されるiOSのバージョンによって異なる可能性があります。

OpenGL ES 1.1アプリケーションまたはOpenGL ES 2.0アプリケーションのどちらのビルドを選択した場合であっても、アプリケーションが最小に行うことは、基盤となる実装における機能を厳密に決定することです。機能を決定するため、アプリケーションは現在のコンテキストを設定し、1つ以上のOpenGL ES関数を呼び出して実装の具体的な機能を取得します。コンテキストの機能は、コンテキストが作成された後は変化しません。アプリケーションは機能を一度テストすることができ、その機能を使用して、機能に適合するよう描画コードを調整することができます。たとえば、実装が提供するテクスチャユニットの数に応じて、演算を単一パスや複数パスで実行したり、よりシンプルなアルゴリズムを選んだりすることができます。一般的なパターンは、共通のスーパークラスを共有するクラスを使用して、アプリケーションのレンダリングパスごとにクラスを設計することです。実行時には、コンテキストの機能に最も適合するクラスをインスタンス化します。

実装依存の値の読み取り

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。PowerVR SGXソフトウェアがサポートするテクスチャは最大2048×2048ですが、PowerVR MBXグラフィックスハードウェアをサポートするiOSデバイスは、最大1024×1024のサイズのテクスチャをサポートします。どちらのサイズも、OpenGL ES仕様で要求される最小サイズの64×64を大幅に上回っています。アプリケーションの要件がOpenGL ES仕様が要求する最小機能を上回る場合、アプリケーションが実装に問い合わせてハードウェアの実際の機能を確認し、適切なエラーを発生させます。このような場合は、より小さいテクスチャをロードしたり、別のレンダリング方法を選んだりすることができます。

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 3-1では、OpenGL ES 1.1アプリケーションおよびOpenGL ES 2.0アプリケーションの両方がテストする必要のある値の一覧を示しています。

表 3-1　共通するOpenGL ES実装依存の値

テクスチャの最大サイズ	GL_MAX_TEXTURE_SIZE
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

OpenGL ES 2.0アプリケーションの場合、アプリケーションはまず、シェーダに設定された制限を読み取る必要があります。すべてのグラフィックスハードウェアは、数に限りはありますが、頂点シェーダおよびフラグメントシェーダに渡すことができる属性をサポートします。OpenGL ES 2.0実装では、実装が提供する値をアプリケーションが超える場合に、ソフトウェアのフォールバックを提供する必要がありません。アプリケーションは、仕様が定める最小値未満で使用し続けるか、または表 3-2に示すシェーダの制限を確認し、制限の範囲内で使用できるシェーダを選びます。

表 3-2 OpenGL ES 2.0シェーダの値

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

すべての種類のベクトルに対して、クエリは4つのコンポーネントからなる利用可能な浮動小数点ベクトルの数を返します。

OpenGL ES 1.1アプリケーションは、テクスチャユニットの数および利用可能なクリッピングプレーンの数を確認する必要があります（表 3-3参照）。

表 3-3 OpenGL ES 1.1の値

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

拡張機能を使用する前の確認

OpenGL ES実装は、OpenGL ES拡張機能を実装することによって、OpenGL ES APIに新たな機能を追加します。アプリケーションは、使用しようとする機能が含まれるOpenGL ES拡張機能の存在を確認しなければなりません。ただ1つの例外がOES_framebuffer_object拡張機能で、この拡張機能は、iOSにおける、OpenGL ES 1.1のすべての実装で常に提供されます。iOSは、アプリケーションによる描画先となるフレームバッファの唯一の種類としてフレームバッファオブジェクトを使用します。

リスト 3-1では、拡張機能の存在の確認に使用できるコードを紹介します。

リスト 3-1 OpenGL ES拡張の確認

```
BOOL CheckForExtension(NSString *searchName)
{
//パフォーマンスのため、一度配列を作成したらキャッシュしておくことが可能。
```

```
    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は、定期的に`glGetError`関数を呼び出して、返されるエラーすべてにフラグを設定する必要があります。`glGetError`関数がエラーを返す場合、そのエラーは、アプリケーションがOpenGL ES APIを誤って使用しているか、または要求された操作を基盤となる実装が実行できないことを意味します。

`glGetError`関数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する可能性があります。この関数の呼び出しは、アプリケーションのリリースバージョンでは控えめにしてください。

OpenGL ESによる描画

この章では、フレームバッファを作成し、画像をフレームバッファにレンダリングするプロセスを詳しく説明します。フレームバッファオブジェクトの作成、アニメーションを実行するレンダリングループの実装方法、およびCore Animationの使いかたについてさまざまな手法を紹介します。最後に、Retinaディスプレイへの高解像度画像のレンダリング、画像品質向上のためのマルチサンプル機能の使用、外部ディスプレイに画像をレンダリングするためのOpenGL ESの使用など、高度なトピックを取り上げます。

レンダリングの結果を格納するフレームバッファオブジェクト

OpenGL ES仕様は、レンダリングした画像を保持する**フレームバッファ**の作成にアプリケーションが使用できるメカニズムを、各実装が提供することを要求しています。iOSでは、すべてのフレームバッファがフレームバッファオブジェクトを使用して実装されます。フレームバッファオブジェクトは、OpenGL ES 2.0には組み込み済みで、iOSにおけるすべてのOpenGL ES 1.1実装では`GL_OES_framebuffer_object`拡張機能により提供されます。

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。単一のコンテキストに複数のフレームバッファオブジェクトを作成することも可能で、フレームバッファ間でリソースを共有することができます。

フレームバッファを作成する手順は次のとおりです。

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、フレームバッファオブジェクト上のアタッチメントポイントにストレージをそれぞれアタッチします。
3. フレームバッファの完全性をテストします。

アプリケーションは、実行しようとするタスクに応じて異なるオブジェクトを設定し、フレームバッファオブジェクトにアタッチします。ほとんどの場合、フレームバッファの設定における違いは、フレームバッファオブジェクトの色アタッチメントポイントにどのオブジェクトがアタッチされるかということです。

- オフスクリーンの画像処理にフレームバッファを使用する場合は、レンダバッファをアタッチします。詳細については、「オフスクリーンフレームバッファオブジェクトの作成」（32 ページ）を参照してください。
- フレームバッファの画像を後のレンダリング手順の入力として使用する場合は、テクスチャをアタッチします。詳細については、「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」を参照してください。（32 ページ）

- フレームバッファをユーザに表示しようとする場合は、Core Animation対応の特別なレンダバッファを使用します。詳細については、「Core Animationレイヤへのレンダリング」（33 ページ）を参照してください。

オフスクリーンフレームバッファオブジェクトの作成

オフスクリーンレンダリングのためのフレームバッファは、そのアタッチメントのすべてをOpenGL ESレンダバッファとして割り当てます。次のコードは、色アタッチメントおよび深度アタッチメントを持つフレームバッファオブジェクトを割り当てます。

1. フレームバッファを作成し、それをバインドします。

```
GLuint framebuffer;
glGenFramebuffers(1, &framebuffer);
glBindFramebuffer(GL_FRAMEBUFFER, 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、フレームバッファの色アタッチメントポイントにストレージをアタッチします。

```
GLuint colorRenderbuffer;
glGenRenderbuffers(1, &colorRenderbuffer);
glBindRenderbuffer(GL_RENDERBUFFER, colorRenderbuffer);
glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA8, width, height);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_RENDERBUFFER,
 colorRenderbuffer);
```

3. 深度または深度／ステンシルのレンダバッファを作成し、それにストレージを割り当て、フレームバッファの深度アタッチメントポイントにストレージをアタッチします。

```
GLuint depthRenderbuffer;
glGenRenderbuffers(1, &depthRenderbuffer);
glBindRenderbuffer(GL_RENDERBUFFER, depthRenderbuffer);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT16, width, height);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER,
 depthRenderbuffer);
```

4. フレームバッファの完全性をテストします。このテストは、フレームバッファの設定が変更された場合のみ実行する必要があります。

```
GLenum status = glCheckFramebufferStatus(GL_FRAMEBUFFER) ;
if(status != GL_FRAMEBUFFER_COMPLETE) {
    NSLog(@"failed to make complete framebuffer object %x", status);
}
```

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

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のものとほとんど同じですが、テクスチャが割り当てられて色アタッチメントポイントに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。

2. レンダリング先のテクスチャを作成し、そのテクスチャをフレームバッファの色アタッチメントポイントにアタッチします。

```
// 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);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
texture, 0);
```

3. 深度バッファを割り当ててアタッチします（前述のとおり）。

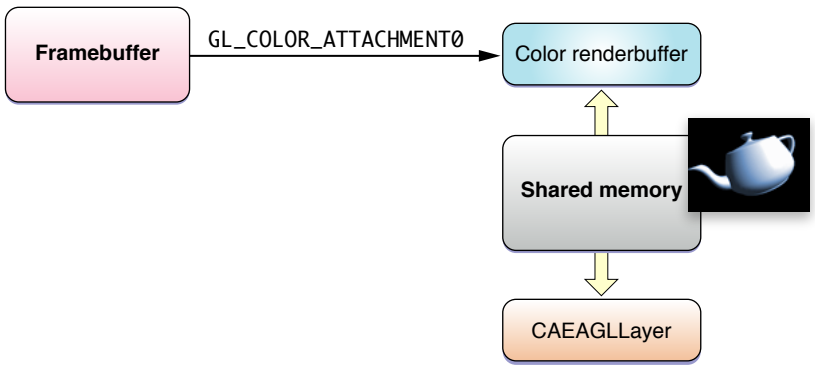
4. フレームバッファの完全性をテストします（前述のとおり）。

この例は色テクスチャへのレンダリングを前提としていますが、ほかの方法をとることができます。たとえば、OES_depth_texture拡張機能を使用することで、テクスチャを深度アタッチメントポイントにアタッチし、シーンから得た深度情報をテクスチャに格納することができます。この深度情報を使用して、最終的にレンダリングされるシーンの影を計算することができます。

Core Animationレイヤへのレンダリング

OpenGL ESを使用して描画するアプリケーションのほとんどは、フレームバッファのコンテンツをユーザに対して表示しようとします。iOSでは、画面に表示されるすべての画像をCore Animationが処理します。各ビューの背面には、そのビューに対応するCore Animationレイヤがあります。OpenGL ESは、特別なCore Animationレイヤである`CAEAGLLayer`を通じてCore Animationに接続します。`CAEAGLLayer`を使用することで、OpenGL ESレンダバッファのコンテンツにCore Animationレイヤのコンテンツの役割をさせることができます。これにより、レンダバッファコンテンツを、UIKitやQuartzを使用してレンダリングされたコンテンツを含むほかのコンテンツとともに変換したり、合成したりすることができます。Core Animationが最終画像を合成すると、最終画像はデバイスのメイン画面またはアタッチされた外部ディスプレイに表示されます。

図 4-1 Core AnimationとOpenGL ESとのレンダバッファの共有



ほとんどの場合、アプリケーションが`CAEAGLLayer`オブジェクトを直接割り当てることはありません。その代わり、アプリケーションは、`CAEAGLLayer`オブジェクトをバッキングレイヤとして割り当てる`UIView`のサブクラスを定義します。実行時には、アプリケーションがビューをインスタンス

化し、そのビューをビュー階層に配置します。ビューがインスタンス化されると、アプリケーションはOpenGL ESコンテキストを初期化し、Core Animationに接続するフレームバッファオブジェクトを作成します。

CAEAGLLayerは、EAGLDrawableプロトコルを実装することによってこのサポートをOpenGL ESに提供します。EAGLDrawableを実装するオブジェクトは、EAGLContextオブジェクトと密接に連携します。描画可能なオブジェクトには、2つの重要な機能があります。1つは、レンダバッファ用の共有ストレージを割り当てることです。そしてもう1つは、そのレンダバッファのコンテンツを表示するコンテキストと密接に連携することです。レンダバッファのコンテンツの表示は、EAGLによって大まかに定義されています。CAEAGLLayerオブジェクトの場合、レンダバッファのコンテンツの表示は、Core Animationにそれまで示されたコンテンツをレンダバッファのコンテンツが置き換えることを意味します。このモデルの利点は、Core Animationレイヤのコンテンツを、フレームごとにレンダリングする必要がなく、レンダリングされた画像が実際に変化する場合だけレンダリングすればよいということです。

OpenGL ES対応ビューの作成に使用する手順は次のとおりです。Xcodeが提供するOpenGL ESテンプレートを使用すれば、自分で以下のコードを実装する必要はありません。

1. UIViewをサブクラス化して、iOSアプリケーションのOpenGL ESビューを作成します。

2. layerClassメソッドをオーバーライドし、基盤レイヤとしてCAEAGLLayerオブジェクトをビューが作成するようにします。これを実行するには、layerClassメソッドを使用してCAEAGLLayerクラスを返します。

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

3. ビューの初期化ルーチンにおいて、ビューのlayerプロパティを読み取ります。コードでは、フレームバッファオブジェクトを作成するときにこのプロパティを使用します。

```
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にするため、CALayerクラスが提供するopaqueプロパティをYESに設定することで、レイヤを不透過として指定します。詳細については、「Core Animationにおける合成機能のパフォーマンス向上」（40 ページ）を参照してください。

   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、レンダバッファのピクセルフォーマットを指定したり、Core Animationに表示した後にコンテンツを保持するかどうかを指定したりできます。設定可能なキーの一覧については、『*EAGLDrawable Protocol Reference*』を参照してください。

5. コンテキストを割り当てて、そのコンテキストを現在のコンテキストに設定します。詳細については、「OpenGL ESコンテキストの設定」（23 ページ）を参照してください。

6. フレームバッファオブジェクトを作成します（上記参照）。

7. 色レンダバッファを作成します。コンテキストのrenderbufferStorage:fromDrawable:メソッドを呼び出してストレージを割り当てて、レイヤオブジェクトをパラメータとして渡します。幅、高さ、ピクセルフォーマットはレイヤから取得し、レンダバッファにストレージを割り当てるために使用します。

```
GLuint colorRenderbuffer;
glGenRenderbuffers(1, &colorRenderbuffer);
glBindRenderbuffer(GL_RENDERBUFFER, colorRenderbuffer);
[myContext renderbufferStorage:GL_RENDERBUFFER fromDrawable:myEAGLLayer];
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_RENDERBUFFER,
 colorRenderbuffer);
```

注： Core Animationレイヤの境界矩形またはプロパティが変化した場合、アプリケーションはレンダバッファのストレージを再割り当てする必要があります。レンダバッファを再割り当てしないと、レンダバッファのサイズがビューのサイズと合致しません。この場合、Core Animationは画像のコンテンツがビューにきちんと収まるよう拡大縮小することがあります。これを回避するため、Xcodeテンプレートは、ビューレイアウトが変更されると常にフレームバッファとレンダバッファを再割り当てします。

8. 色レンダバッファの高さと幅を取得します。

```
GLint width;
GLint height;
glGetRenderbufferParameteriv(GL_RENDERBUFFER, GL_RENDERBUFFER_WIDTH, &width);
glGetRenderbufferParameteriv(GL_RENDERBUFFER, GL_RENDERBUFFER_HEIGHT, &height);
```

前出の各例では、バッファのストレージを割り当てるため、レンダバッファの幅と高さが明示的に指定されていました。この例では、ストレージが割り当てられた後、コードによって色レンダバッファから幅と高さを取得しています。アプリケーションがこのようにするのは、色レンダバッファの実際の寸法がビューの境界矩形と倍率に基づいて計算されるためです。フレームバッファにアタッチされるほかのレンダバッファも同じ寸法でなければなりません。深度バッファを割り当てるために高さと幅を使用するほか、OpenGL ESビューポートの割り当てに加え、アプリケーションのテクスチャとモデルで必要な詳細レベルの決定に役立てるために幅と高さを使用します。詳細については、「OpenGL ESの使用による高解像度ディスプレイのサポート」（42 ページ）を参照してください。

9. 深度バッファを割り当ててアタッチします。

10. フレームバッファオブジェクトをテストします。

簡単に言うと、フレームバッファを作成する手順はほとんど同じです。色アタッチメントや深度アタッチメントを割り当てる手順があり、主に違うのは色アタッチメントを割り当てる方法です。

表 4-1 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メカニズム

オフスクリーンのレンダバッファ	`glRenderbufferStorage`
Drawableレンダバッファ	`renderbufferStorage: fromDrawable:`
テクスチャ	`glFramebufferTexture2D`

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作成したら、次にそのオブジェクトを埋める必要があります。このセクションでは、新しいフレームをレンダリングしてユーザに表示するために必要な手順を説明します。テクスチャまたはオフスクリーンのフレームバッファへのレンダリングは同じように行われますが、アプリケーションが最終フレームを使用する方法だけが異なります。

一般にアプリケーションは、次に示す2つのうちどちらかの状況で新しいフレームをレンダリングします。

- オンデマンドの場合。アプリケーションは、フレームのレンダリングに使用するデータが変更されたと認識したときに新しいフレームをレンダリングします。
- アニメーションループの場合。アプリケーションが、フレームのレンダリングに使用するデータがフレームごとに変化すると想定している場合です。

オンデマンドでのレンダリング

オンデマンドのレンダリングは、フレームのレンダリングに使用するデータがあまり頻繁に変化しない場合、つまりユーザのアクションに応答してデータが変化する場合のみ適しています。iOSにおけるOpenGL ESはこのモデルによく適合しています。フレームを表示するとき、Core Animationはフレームをキャッシュし、新しいフレームが表示されるまでの間、格納されたフレームを使用します。必要なときだけ新しいフレームをレンダリングすることによってデバイスのバッテリーの節約になり、デバイスがほかのアクションを実行する時間が長くなります。

注： OpenGL ES対応のビューは、ビューのコンテンツをレンダリングするために`drawRect:`メソッドを実装するべきではありません。その代わりに、独自のメソッドを実装して、新たなOpenGL ESフレームを描画して表示し、データが変化したときに呼び出すようにします。`drawRect:`メソッドを実装すると、UIKitによるビューの扱い方に別の変化が生じます。

アニメーションループを使用したレンダリング

アニメーションループを使用したレンダリングは、データがフレームごとに変化する可能性が非常に高い場合に適しています。たとえば、ゲームやシミュレーションが静止画像を表示することはめったにありません。スムーズなアニメーションは、オンデマンドモデルを実装することよりも重要です。

iOSでは、アニメーションループをセットアップする最良の方法は、`CADisplayLink`オブジェクトを使用することです。ディスプレイリンクはCore Animationオブジェクトで、描画と画面のリフレッシュレートを同期化します。これにより、スタッタリングやティアリングを起こすことなく画面のコンテンツをスムーズに更新できます。リスト 4-1 （36 ページ）では、ビューが表示されている画面を取得し、その画面を使用して新たなディスプレイリンクオブジェクトを作成し、ディスプレイリンクオブジェクトを実行ループに追加する方法を示しています。

リスト 4-1　ディスプレイリンクの作成と開始

```
displayLink = [myView.window.screen displayLinkWithTarget:self
selector:@selector(drawFrame)];
```

```
[displayLink addToRunLoop:[NSRunLoop currentRunLoop]
forMode:NSDefaultRunLoopMode];
```

アプリケーションは、drawFrameメソッドの実装内部で、ディスプレイリンクのtimestampプロパティを読み取って、レンダリングする次のフレームのタイムスタンプを取得する必要があります。アプリケーションはその値を使用して、次のフレームのオブジェクトの位置を計算できます。

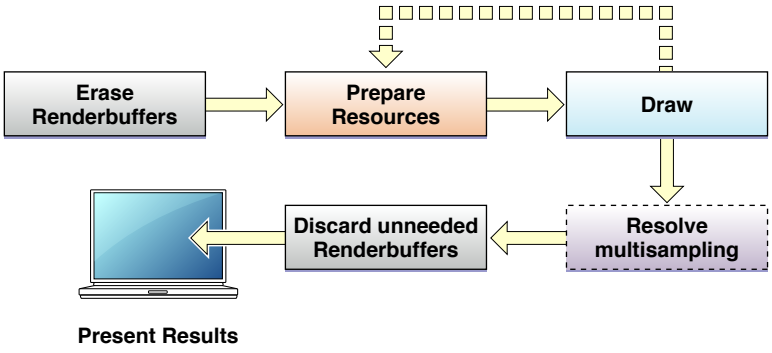
通常、ディスプレイリンクオブジェクトは、画面がリフレッシュするごとに作動します。リフレッシュの値は通常60Hzですが、デバイスによって異なる可能性があります。ほとんどのアプリケーションは、画面を1秒間に60回更新する必要はありません。ディスプレイリンクのframeIntervalプロパティは、メソッドが呼び出される前に経過する実際のフレーム数に設定することができます。たとえば、フレームの間隔が3に設定される場合、アプリケーションは3フレームごとに呼び出され、1秒あたりおよそ20フレームが表示されます。

重要： 最良の結果を得るためには、アプリケーションが一貫して達成可能なフレームレートを選びます。スムーズで一貫したフレームレートは、一貫性のないフレームレートの場合よりもより快適なユーザ体験を提供します。

フレームのレンダリング

図 4-2 （37 ページ）は、フレームをレンダリングし、表示するためにiOS上でOpenGL ESアプリケーションがとるべき手順を示します。この手順には、アプリケーションのパフォーマンスを向上させるヒントが多く含まれています。

図 4-2 iOSにおけるOpenGLのレンダリング手順



レンダバッファの消去

各フレームの始まりに、次のフレームの描画には必要のない、前のフレームのコンテンツが含まれるすべてのレンダバッファを消去します。glClear関数を呼び出して、消去するすべてのバッファについてビットマスクを渡します（リスト 4-2 （37 ページ）参照）。

リスト 4-2 レンダバッファの消去

```
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
```

glClearの使用は、手作業でバッファを消去するよりも効率的というだけでなく、glClearを使用することで、既存のコンテンツが破棄可能であるというヒントがOpenGL ESに伝達されます。一部のグラフィックスハードウェアでは、この関数の使用により、それまでのコンテンツをメモリにロードするという負荷の大きいメモリ操作が回避されます。

OpenGL ESオブジェクトの作成

この手順と次の手順は、アプリケーションの核心であり、ユーザに表示する内容を決定する手順です。この手順では、フレームのレンダリングに必要となるすべてのOpenGL ESオブジェクト（頂点バッファオブジェクト、テクスチャなど）を作成します。

描画コマンドの実行

この手順は、前の手順で作成されたオブジェクトを取得し、これらオブジェクトを使用する描画コマンドを送信します。レンダリングコードのこの部分が効率よく実行されるよう設計する方法は、「OpenGL ESアプリケーション設計ガイドライン」（47 ページ）で詳しく説明しています。今のところ、パフォーマンス最適化における最も重要な点は、新しいフレームのレンダリング開始時にOpenGL ESオブジェクトを修正する場合だけ、アプリケーションの実行が高速化するということです。アプリケーションは、オブジェクトの修正と描画コマンドの送信を交互に行う（点線で示した部分）ことができますが、アプリケーションは各手順を1度だけ実行する場合に、より高速に実行されます。

マルチサンプル機能のリゾルブ

アプリケーションは、マルチサンプル機能を使用して画像品質を改善する場合、ユーザに表示される前にピクセルをリゾルブしなければなりません。マルチサンプル機能を使用するアプリケーションについては、「マルチサンプル機能の使用による画像品質の向上」（40 ページ）で詳しく説明しています。

不要なレンダバッファの破棄

*破棄*の操作はEXT_discard_framebuffer拡張機能で定義され、iOS 4.0以降で利用することができます。破棄操作は、iOSの初期のバージョンでアプリケーションが実行されている場合は除外するべきですが、利用可能な場合は必ず含めるようにします。破棄は、OpenGL ESに対するパフォーマンス上のヒントです。ヒントによって、1つ以上のレンダバッファのコンテンツがアプリケーションに使用されないことが破棄コマンドの完了後にOpenGL ESに伝達されます。レンダバッファのコンテンツをアプリケーションが必要としていないというヒントをOpenGL ESに伝達することにより、バッファ内のデータを破棄することができたり、バッファのコンテンツを更新し続けるという負担のかかるタスクを回避できたりします。

レンダリングループのこの段階では、アプリケーションは、フレームに対するすべての描画コマンドの送信が完了しています。アプリケーションが、画面に表示するための色レンダバッファを必要としている間は、深度バッファのコンテンツを必要としていない可能性があります。リスト4-3は、深度バッファのコンテンツを破棄します。

リスト 4-3　深度フレームバッファの破棄

```
const GLenum discards[]  = {GL_DEPTH_ATTACHMENT};
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
glDiscardFramebufferEXT(GL_FRAMEBUFFER,1,discards);
```

Core Animationへの結果の表示

この手順の段階では、色レンダバッファがレンダリングの完了したフレームを保持しています。そのため、必要なことはフレームをユーザに表示することだけです。リスト 4-4 （39 ページ）は、レンダバッファをコンテキストにバインドし、レンダバッファを表示します。これにより、レンダリングの完了したフレームをCore Animationに渡すことができます。

リスト 4-4　レンダリングの完了したフレームの表示

```
glBindRenderbuffer(GL_RENDERBUFFER, colorRenderbuffer);
[context presentRenderbuffer:GL_RENDERBUFFER];
```

デフォルトでは、レンダバッファのコンテンツは、アプリケーションがレンダバッファを表示した後は*破棄*されることを前提としなければなりません。これは、アプリケーションがフレームを表示するごとに、フレームのコンテンツを完全に作成し直して新しいフレームをレンダリングしなければならないことを意味します。こうした理由から、上記のコードは常に色バッファを消去します。

アプリケーションがフレーム間の色レンダバッファのコンテンツを維持する必要のある場合は、`kEAGLDrawablePropertyRetainedBacking`キーを`CAEAGLLayer`オブジェクトの`drawableProperties`プロパティに格納されるディクショナリに追加し、前の`glClear`関数呼び出しから`GL_COLOR_BUFFER_BIT`定数を削除します。保持されているバッキングは、バッファのコンテンツを維持するため追加のメモリの割り当てをiOSに求める場合があり、これにより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。

OpenGL ESアプリケーションにおけるView Controllerのサポート

OpenGL ESアプリケーションの多くは没入型で、画面全体を使用してコンテンツを表示します。しかし、同じ没入型アプリケーションで、iOSのほかの機能とのやり取りが必要となることが少なくありません。たとえば、iAd広告を表示するアプリケーションやGame Centerの組み込みView Controllerを使用するアプリケーションは、View Controllerを提供しなければなりません。このView Controllerは、システムView Controllerが提供するコンテンツをモーダルモードで表示するために使用します。このため、Xcodeテンプレートは、対応するView Controllerを提供してビューを管理します。アプリケーションも同じようにする必要があります。

View Controllerの使用で重要なのは、ビューの回転を扱うことです。iOS 4.2以降を実行する、PowerVR SGXを装備したデバイスでは、Core Animationの回転およびOpenGL ESコンテンツの拡大縮小のパフォーマンスが大幅に向上しています。アプリケーションはView Controllerを使用して、許可される向きを設定したり、ユーザがデバイスを回転させたときの向きと向きの間のトランジションを行う必要があります（『*View Controller Programming Guide for iOS*』参照）。View Controllerを使用することにより、View Controllerがモーダルモードで表示するほかのView Controllerは、表示する側のView Controllerと同じ向きで表示されます。

アプリケーションがPowerVR MBXを装備したデバイスをサポートする場合、これらのデバイス上ではCore Animationの使用を避ける必要の生じる場合があります。その代わりに、OpenGL ES内部で直接回転を実行します。回転が生じるときは、モデルビューや投影マトリックスを変更し、`glViewport`関数および`glScissor`関数の幅と高さの引数を入れ替えます。

Core Animationにおける合成機能のパフォーマンス向上

レンダバッファのコンテンツは、アニメーション化され、ビュー階層に含まれるほかのCore Animationレイヤと合成されます。合成されるレイヤは、OpenGL ES、Quartz、またはそのほかのグラフィックスライブラリのいずれで描画されたかは関係ありません。これは便利です。なぜなら、OpenGL ESがCore Animationにとって最も優遇できるグラフィックスライブラリであることを意味するからです。ただし、OpenGL ESコンテンツをほかのコンテンツと混在させるには時間がかかります。適切に使用しなければ、アプリケーションの実行速度が低下し、やり取りを行うフレームレートに到達するのに時間がかかる可能性があります。コンテンツを混在、適合させることに伴うパフォーマンス上のペナルティは、iOSデバイスに搭載される基盤となるグラフィックスハードウェアによって異なります。PowerVR MBXグラフィックスプロセッサを使用するデバイスは、複雑なシーンの合成時により重大なペナルティを負うことになります。最良の結果を得るためには、出荷の対象となるすべてのiOSデバイス上で必ずアプリケーションをテストしてください。

最良のパフォーマンスを確実に実現するには、アプリケーションにおいて、OpenGL ESだけを使用してコンテンツをレンダリングする必要があります。これを行うには、`CAEAGLLayer`オブジェクトを保持するビューのサイズを画面に一致させ、画面の`opaque`プロパティを`YES`に設定して、ほかのCore Animationレイヤやビューが見えないようにします。OpenGL ESレイヤが、ほかのレイヤなかでも一番上に合成される場合、`CAEAGLLayer`オブジェクトを不透過にすることでパフォーマンス上の負担は軽減できますが、なくすことはできません。

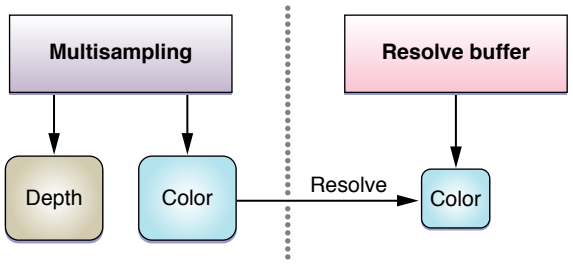
`CAEAGLLayer`オブジェクトが、その下にあるレイヤ階層の一番上のレイヤで合成される場合、レンダバッファの色データは、Core Animationが正しく合成するよう、事前乗算済みのアルファ形式になっていなければなりません。OpenGL ESコンテンツをほかのコンテンツの上で合成することは、パフォーマンス上の重大なペナルティを伴います。

マルチサンプル機能の使用による画像品質の向上

iOS 4以降、OpenGL ESは`APPLE_framebuffer_multisample`拡張機能をサポートします。マルチサンプル機能は、**アンチエイリアス**の1つの形式で、アプリケーションがギザギザのエッジを滑らかにしたり、画質を向上させたりすることができます。マルチサンプル機能はより多くのリソース（メモリとフラグメントの処理）を使用しますが、ほとんどの3Dアプリケーションで画像品質が向上します。

図 4-3 （41 ページ）は、マルチサンプル機能の概念を示しています。マルチサンプル機能では、アプリケーションは、1個のフレームバッファオブジェクトを作成する代わりに、フレームバッファオブジェクトを2個作成します。1つ目のフレームバッファオブジェクトは**リゾルブバッファ**で、色レンダバッファを格納しますが、その点を除き、前述と同じように割り当てられます。リゾルブバッファは、最終画像がレンダリングされる場所です。2つ目のフレームバッファオブジェクトである**マルチサンプルバッファ**は、深度アタッチメントと色アタッチメントの両方を格納します。マルチサンプルレンダバッファは、リゾルブフレームバッファと同じ寸法を使用して割り当てられますが、それぞれのマルチサンプルレンダバッファには、各ピクセルについて格納するサンプル数を指定する追加のパラメータが含まれます。アプリケーションは、マルチサンプルバッファへのレンダリングすべてを実行した後、バッファのサンプルをリゾルブバッファへ*リゾルブ*することによって、アンチエイリアス化された最終画像を生成します。

図 4-3 マルチサンプル機能の仕組み



リスト 4-5 （41 ページ）は、マルチサンプルバッファを作成するコードを示しています。このコードは、以前に作成したバッファの幅と高さを使用し、glRenderbufferStorageMultisampleAPPLE関数を呼び出して、レンダバッファ用にマルチサンプル化されたストレージを作成します。

リスト 4-5 マルチサンプルバッファの作成

```
glGenFramebuffers(1, &sampleFramebuffer);
glBindFramebuffer(GL_FRAMEBUFFER, sampleFramebuffer);

glGenRenderbuffers(1, &sampleColorRenderbuffer);
glBindRenderbuffer(GL_RENDERBUFFER, sampleColorRenderbuffer);
glRenderbufferStorageMultisampleAPPLE(GL_RENDERBUFFER, 4, GL_RGBA8_OES, width,
 height);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_RENDERBUFFER,
 sampleColorRenderbuffer);

glGenRenderbuffers(1, &sampleDepthRenderbuffer);
glBindRenderbuffer(GL_RENDERBUFFER, sampleDepthRenderbuffer);
glRenderbufferStorageMultisampleAPPLE(GL_RENDERBUFFER, 4, GL_DEPTH_COMPONENT16,
 width, height);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER,
 sampleDepthRenderbuffer);

if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
    NSLog(@"Failed to make complete framebuffer object %x",
glCheckFramebufferStatus(GL_FRAMEBUFFER));
```

マルチサンプル機能をサポートするレンダリングコードの修正手順を次に示します。

1. バッファ消去の手順では、マルチサンプルフレームバッファのコンテンツを消去します。

    ```
    glBindFramebuffer(GL_FRAMEBUFFER, sampleFramebuffer);
    glViewport(0, 0, framebufferWidth, framebufferHeight);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    ```

2. 描画コマンドを送信した後、マルチサンプルバッファのコンテンツをリゾルブバッファへとリゾルブします。各ピクセルについて格納されたサンプルは、リゾルブバッファにおいて1つのサンプルに結合されます。

    ```
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER_APPLE, resolveFrameBuffer);
    glBindFramebuffer(GL_READ_FRAMEBUFFER_APPLE, sampleFramebuffer);
    glResolveMultisampleFramebufferAPPLE();
    ```

3. 破棄の手順では、マルチサンプルフレームバッファにアタッチされている両方のレンダバッファを破棄できます。これは、表示しようとするコンテンツがリゾルブバッファに格納されているためです。

```
const GLenum discards[]  = {GL_COLOR_ATTACHMENT0,GL_DEPTH_ATTACHMENT};
glDiscardFramebufferEXT(GL_READ_FRAMEBUFFER_APPLE,2,discards);
```

4. 表示の手順では、リゾルブフレームバッファにアタッチされている色レンダバッファを表示します。

```
glBindRenderbuffer(GL_RENDERBUFFER, colorRenderbuffer);
[context presentRenderbuffer:GL_RENDERBUFFER];
```

マルチサンプル機能は負担を伴わないわけではありません。追加のサンプルを格納するには追加のメモリが必要となり、リゾルブフレームバッファへのサンプルのリゾルブには時間がかかります。マルチサンプル機能をアプリケーションに追加する場合は、パフォーマンスが許容できる水準を維持できるよう、アプリケーションのパフォーマンスを必ずテストしてください。

OpenGL ESの使用による高解像度ディスプレイのサポート

アプリケーションがレンダリングにOpenGL ESを使用している場合、既存の描画コードは変更なしでそのまま動作するはずです。しかし、高解像度画面に描画すると、それに応じてコンテンツが拡大縮小され、Retinaではないディスプレイに表示されているように見えます。この理由は、`CAEAGLLayer`クラスのデフォルトの動作が倍率を`1.0`に設定しているためです。高解像度の描画を行うには、OpenGL ESコンテンツの表示に使用するビューの倍率を変更しなければなりません。UIKitで高解像度ディスプレイをサポートする方法の詳細については、「ESの使用による高解像度ディスプレイのサポート」を参照してください。

ビューの`contentScaleFactor`プロパティを変更すると、それに対応して基盤となる`CAEAGLLayer`オブジェクトの倍率が変更されます。`renderbufferStorage:fromDrawable:`メソッドは、レンダバッファのストレージ作成に使用するメソッドで、レイヤの境界とその倍率をかけ算することでレンダバッファのサイズを計算します。「Core Animationレイヤへのレンダリング」（33 ページ）に示されるコードは、Core Animationのレンダバッファのストレージを割り当てて、レンダバッファの幅と高さを取得します。これにより、レンダバッファのストレージはコンテンツの倍率に加えられた変更を自動的に取得することができます。倍率を2倍にすると、レンダバッファには、レンダリングするピクセルが4倍格納されます。レンダリング後に、コンテンツを提供して、追加分のピクセルを埋めるかどうかは任意です。

> **重要：** `CAEAGLLayer`オブジェクトを基盤とするビューは、カスタムの`drawRect:`メソッドを実装してはなりません。`drawRect:`メソッドの実装により、ビューのデフォルトの倍率は、画面の倍率と一致するよう暗黙的に変更されます。アプリケーションの描画コードがこの動作を想定していない場合、アプリケーションのコンテンツは正しくレンダリングされません。

高解像度の描画を採用する場合は、それに応じてアプリケーションのモデルアセットやテクスチャアセットを調整します。iPadまたは高解像度デバイスで実行するときは、より詳細なモデルやテクスチャを選んで、より高画質な画像をレンダリングすることができます。逆に、標準解像度のiPhone上では、小さなモデルとテクスチャを使い続けることができます。

高解像度のコンテンツをサポートするかどうかを決める際の重要な要素として、パフォーマンスがあります。レイヤの倍率を1.0から2.0に変更したときにピクセル数が4倍になると、処理するフラグメントが4倍になります。アプリケーションがフラグメント単位の計算を多く実行している場合、ピクセル数の増加によりアプリケーションのフレームレートが下がる可能性があります。高い倍率ではアプリケーションの実行が極端に遅くなるようであれば、次の選択肢を検討してください。

- この文書のパフォーマンスチューニングガイドラインを使用して、フラグメントシェーダのパフォーマンスを最適化する。
- よりシンプルなアルゴリズムをフラグメントシェーダに実装する。これにより、個々のピクセルの品質を下げて、画像全体をより高い解像度でレンダリングします。
- 1.0と画面の倍率の間の倍率を設定する。1.5の倍率は、1.0の倍率よりも品質は高くなりますが、2.0に拡大された画像よりも塗りつぶしに必要なピクセル数は少なくて済みます。
- 代わりにマルチサンプル機能を使用する。さらなる利点として、マルチサンプル機能は高解像度表示をサポートしていないデバイス上で、より高い品質がもたらされることが挙げられます。
- 良好な結果が得られるできるだけ低い精度の種類を使用して色レンダバッファと深度レンダバッファを作成する。これにより、レンダバッファの処理に必要なメモリ帯域幅が減少します。

最良の解決策はOpenGL ESアプリケーションのニーズによって異なります。開発の段階で上記の選択肢をテストし、パフォーマンスと画像品質のバランスが最も良い方法を選びます。

外部ディスプレイにおけるOpenGL ESコンテキストの作成

一部のiOSデバイスは、外部ディスプレイにアタッチできます。外部ディスプレイの解像度とコンテンツの倍率は、メイン画面とは異なる可能性があります。フレームをレンダリングするコードで適合するよう調整する必要があります。

コンテキストを作成する手順は、メイン画面に作成する場合とほとんど同じです。次の手順に従ってください。

1. 「Displaying Content on an External Display」 in *View Programming Guide for iOS*に示す手順に従って、外部ディスプレイに新しいウインドウを作成します。
2. ウインドウにOpenGL ESビューを追加します。
3. ウインドウの`screen`プロパティを取得し、`displayLinkWithTarget:selector:`を呼び出して、ディスプレイリンクオブジェクトを作成します。これにより、そのディスプレイに最適化されたディスプレイリンクが作成されます。

マルチタスク対応OpenGL ESアプリケーションの実装

iOS4以降、アプリケーションは、ユーザが別のアプリケーションに切り替えても実行し続けることができます。アプリケーションは、バックグラウンドで実行されるよう修正しなければなりません。iOS 3.x以前向けに設計されたアプリケーションは、デベロッパが別のアプリケーションに切り替えると終了します。iOSにおけるマルチタスクの概要については、『*iOS Application Programming Guide*』の「Executing Code in the Background」 を参照してください。

OpenGL ESアプリケーションは、バックグラウンドに移動したときに追加の作業を実行しなければなりません。アプリケーションがこれらの作業を適切に処理しない場合、代わりにiOSがそのアプリケーションを終了する可能性があります。さらに、アプリケーションは、OpenGL ESリソースを解放して、それらのリソースをフォアグラウンドのアプリケーションが利用できるようにすることもできます。

バックグラウンドのアプリケーションはグラフィックスハードウェアのコマンドを実行しない場合がある

OpenGL ESアプリケーションは、グラフィックスハードウェアでOpenGL ESコマンドを実行しようとすると終了します。OpenGL ESアプリケーションは、アプリケーションがバックグラウンドの間、OpenGL ESに対して行われた呼び出しを参照するだけでなく、それまでに送信され、まだ完了していないコマンドも参照します。バックグラウンドのアプリケーションがOpenGL ESコマンドを処理しないようにする主な理由は、最前面のアプリケーションに対してグラフィックスプロセッサを完全に利用できるようにすることです。最前面のアプリケーションは、ユーザに対して常に優れた体験を提供する必要があります。バックグラウンドアプリケーションにグラフィックスプロセッサを占有させてしまうと、ユーザに優れた体験を提供できなくなります。アプリケーションは、バックグラウンドに移動する前に、すでに送信されたすべてのコマンドが完了しているようにしなければなりません。

これをアプリケーションに実装する方法は次のとおりです。

1. アプリケーションデリゲートの`applicationWillResignActive:`メソッドにおいて、アプリケーションがアニメーションタイマーを停止し（タイマーがある場合）、アプリケーションを基地の適切な状態に置いた後、`glFinish`関数を呼び出します。

2. アプリケーションデリゲートの`applicationDidEnterBackground:`メソッドにおいて、OpenGL ESオブジェクトの一部を削除することで、メモリとリソースをフォアグラウンドアプリケーションが利用できるようにすることもできます。`glFinish`関数を呼び出して、リソースがただちに削除されるようにします。

3. アプリケーションが`applicationDidEnterBackground:`メソッドを終了した後は、新たなOpenGL ES呼び出しを行ってはなりません。アプリケーションがOpenGL ES呼び出しを行った場合は、iOSによってアプリケーションが終了されます。

4. アプリケーションの`applicationWillEnterForeground:`メソッドでは、アプリケーションが破棄したオブジェクトを作成し直して、アニメーションタイマーを再始動します。

要約すると、アプリケーションは、すでに送信されたコマンドすべてがコマンドバッファから取り出され、OpenGL ESによって実行されるようにするために、`glFinish`関数を呼び出す必要があるということです。アプリケーションは、バックグラウンドに移動した後、フォアグラウンドに戻るまでの間は、OpenGL ESの使用を一切回避しなければなりません。

作成し直したりソースをバックグラウンドに移動する前に簡単に削除

アプリケーションは、バックグラウンドに移動するとき、OpenGL ESオブジェクトを解放する必要はまったくありません。通常、アプリケーションはコンテンツの破棄を避ける必要があります。次の2つのシナリオについて考えてみます。

- ユーザがゲームをプレイしていて、少しの間ゲームを抜けて、カレンダーをチェックします。ユーザがゲームに戻ると、ゲームのリソースはメモリ内に存在したままで、ゲームをすぐに再開できます。
- ユーザが別のOpenGL ESアプリケーションを起動して、OpenGL ESアプリケーションがバックグラウンドの状態にあります。ユーザが後から起動したOpenGL ESアプリケーションが、デバイス上の空きメモリより多くのメモリを必要とする場合、バックグラウンドのアプリケーションは、追加の作業の実行を要求されることなく、音を出さず自動的に終了します。

デベロッパの目標は、節度ある市民の役割を果たすアプリケーションを設計することです。アプリケーションは、フォアグラウンドへの移動にかかる時間を短くしながら、バックグラウンドに移動している間はメモリ占有量を減らす必要があります。

次は、上記の2つのシナリオに対するアプリケーションの対処方法です。

- アプリケーションは、テクスチャやモデルなどのアセットをメモリに保持します。再作成に時間のかかるリソースは、アプリケーションがバックグラウンドへ移動するときに破棄しないようにします。
- アプリケーションは、すばやく簡単に再作成できるオブジェクトを破棄しなければなりません。大量のメモリを消費するオブジェクトを探します。

破棄しやすいターゲットとしては、レンダリング結果を保持するためにアプリケーションが割り当てるフレームバッファが挙げられます。アプリケーションがバックグラウンドにある場合、そのアプリケーションはユーザからは見えず、OpenGL ESを使用して新しいコンテンツをレンダリングする可能性はありません。これは、アプリケーションのフレームバッファが消費するメモリが割り当てられているのに、役に立っていないことを意味します。さらに、フレームバッファのコンテンツは*一時的*なものです。ほとんどのアプリケーションは、新しいフレームをレンダリングするごとにフレームバッファのコンテンツを作成し直します。このため、レンダバッファはメモリを大量に消費する、簡単に再作成可能なリソースとなり、アプリケーションがバックグラウンドに移動するときに廃棄できるオブジェクトのちょうどよい候補となります。

OpenGL ESアプリケーション設計ガイドライン

OpenGL ESは、変換、ライティング、クリップ、テクスチャ、環境効果など、大量のデータセットに対する、数多くの複雑な演算をデベロッパに代わって実行します。データのサイズと実行される計算の複雑さはパフォーマンスに影響を与える可能性があり、鮮やかな3Dグラフィックスが思ったほど美しくならない場合があります。アプリケーションがOpenGL ESを使用して没入型のリアルタイム画像をユーザに提供するゲームであるか、または画像の品質により重点を置いた画像処理アプリケーションのどちらであるかを問わず、この章に示される情報を利用して、アプリケーションのグラフィックスエンジンの設計に役立ててください。この章では、後の章で詳しく取り上げる主要な概念を紹介します。

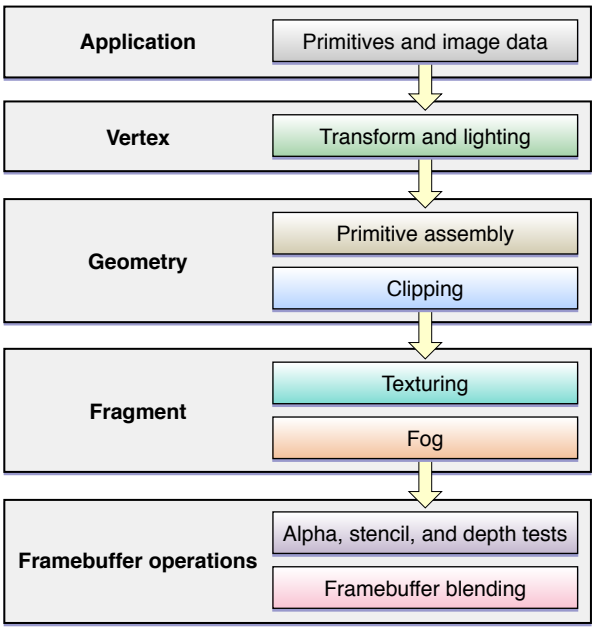
OpenGL ESを視覚化する方法

OpenGL ESを視覚化できる方法がいくつかあり、方法ごとに、アプリケーションの設計と監視を行うコンテキストがわずかに異なります。OpenGL ESを視覚化する最も一般的な方法は、図 6-1に示すような、グラフィックスパイプラインとして視覚化するものです。アプリケーションは、グラフィックスパイプラインを設定し、その後1つ以上の描画コマンドを実行します。描画コマンドは頂点データをパイプラインに送信します。パイプラインは頂点データを処理し、プリミティブへと組み立てられ、フラグメントへとラスタ化されます。各フラグメントは、色と深度の値を計算し、それらの値をフレームバッファにマージします。メンタルモデルとしてパイプラインを使用することは、新しいフレームを生成するためにアプリケーションが実行する作業を特定する上で重要です。典型的なOpenGL ES 2.0アプリケーションの場合、設計では、パイプラインの頂点とフラグメントの段階を扱う、カスタマイズされたシェーダを作成します。OpenGL ES 1.1アプリケーションの場合、希望する計算を実行する固定機能パイプラインを駆動する状態マシンを修正します。

パイプラインモデルのもう1つの利点は、個々の段階で独立して、かつ同時に結果を計算できるということです。これは重要なポイントです。グラフィックスハードウェアの独立した部分がそれまでに送信されたジオメトリについて頂点とフラグメントの計算を実行している間、アプリケーションは新しいプリミティブを作成する場合があります。パイプラインのいずれかの段階の作業量が多すぎる場合、または作業の実行に時間がかかりすぎている場合、ほかのパイプラインは、最も時間のかかっている段階が作業を完了するまでの間、アイドル状態となります。設計では、デバイスに搭載されたグラフィックスハードウェアの能力に計算を適合させることによって、パイプラインの各段階で実行される作業のバランスをとる必要があります。

重要： アプリケーションのパフォーマンスをチューニングする場合、最初の手順は通常、アプリケーションのボトルネックとなっている段階とその理由を特定することです。

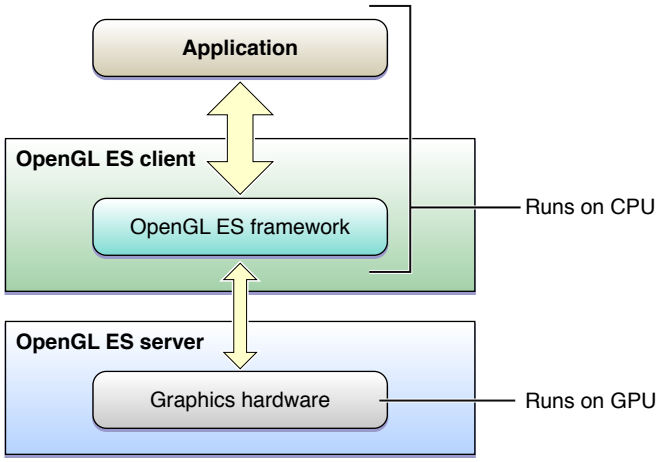
図 6-1 OpenGL ESグラフィックスパイプライン



OpenGL ESを視覚化するもう1つの方法は、クライアントサーバアーキテクチャとして視覚化するというものです（図 6-2参照）。OpenGL ESの状態の変化、テクスチャと頂点のデータ、およびレンダリングコマンドのすべてが、アプリケーションからOpenGL ESクライアントへと送られる必要があります。クライアントはこれらのデータを、グラフィックスハードウェアが解釈できる形式に変換し、GPUへと転送します。これらの変換処理はオーバーヘッドが増加するだけでなく、データをグラフィックスハードウェアに転送するプロセスに時間がかかります。

優れたパフォーマンスを達成するため、アプリケーションはOpenGL ES対して行う呼び出しの頻度を減らし、変換のオーバーヘッドを最小限に抑え、アプリケーションとOpenGL ESとの間のデータの流れを注意深く管理しなければなりません。

図 6-2 OpenGLクライアントサーバアーキテクチャ



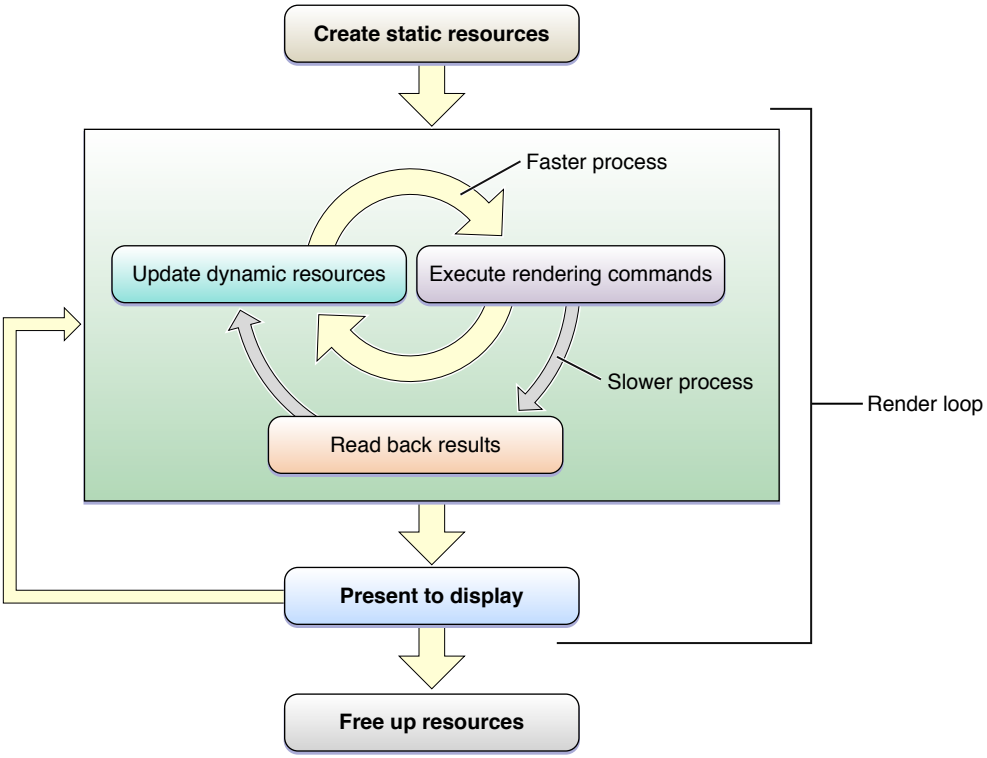
高パフォーマンスなOpenGL ESアプリケーションの設計

簡潔に言うと、しっかりした設計のOpenGL ESアプリケーションは、以下を行う必要があります。

- OpenGL ESパイプラインで並行処理を活用する。
- アプリケーションとグラフィックスハードウェアとの間のデータの流れを管理する。

図 6-3は、画面に表示するアニメーションをOpenGL ESを使用して実行するアプリケーションにおけるプロセスの流れを示しています。

図 6-3 リソース管理のためのアプリケーションモデル



アプリケーションが起動して最初に実行するのは、アプリケーションの実行中に変更するつもりのないリソースの初期化です。こうしたリソースは、アプリケーションがOpenGL ESオブジェクトにカプセル化することが理想です。目標は、アプリケーションの実行中（または、ゲーム内のあるレベルの期間など、アプリケーション実行における一部の期間）に変更しないままにできるオブジェクトを作成し、初期化にかかる時間の増加と引き替えにレンダリングパフォーマンスを向上させることです。複雑なコマンドまたは状態の変更は、単一の関数呼び出しで使用可能なOpenGL ESオブジェクトに置き換える必要があります。たとえば、固定機能パイプラインの設定には、多くの関数呼び出しを要する可能性があります。その代わりに、初期化時にグラフィックスシェーダをコンパイルし、単一の関数呼び出しの実行時にグラフィックスシェーダに切り替えます。作成や修正に負荷が生じるOpenGL ESオブジェクトは、ほとんどの場合、静的オブジェクトとして作成します。

レンダリングループは、OpenGL ESコンテキストにレンダリングするすべての項目を処理し、その結果を画面に表示します。アニメーション化されたシーンでは、一部のデータがフレームごとに更新されます。図 6-3に示す内側のレンダリングループでは、アプリケーションは、レンダリングリソースの更新（プロセス内のOpenGL ESオブジェクトの作成または修正）と、レンダリングリソースを使用する描画コマンドの送信とを交互に実行します。この内側のループの目標は、CPUとGPUが並行して処理を行い、アプリケーションとOpenGL ESが同時に同じリソースにアクセスすることのないよう、作業負荷のバランスをとることです。iOSでは、OpenGL ESオブジェクトの修正は、修正がフレームの最初と最後に実行されない場合、負荷が高くなる可能性があります。

この内側のループの重要な目標は、OpenGL ESからアプリケーションに再びデータがコピーされるのを避けることです。GPUからCPUへの結果のコピーは非常に時間がかかる可能性があります。中央のレンダリングループに示すように、コピーされたデータが、現在のフレームのレンダリングプロセスの一部として後で使用される場合、アプリケーションは、それまでに送信されたすべての描画コマンドが完了するまでブロックされます。

アプリケーションは、フレームで必要なすべての描画コマンドを送信した後、結果を画面に表示します。インタラクティブではないアプリケーションは、その後の処理のために最終画像をアプリケーションが保有するメモリにコピーします。

最後に、アプリケーションが終了する準備ができているとき、またはアプリケーションが主要なタスクを完了したとき、アプリケーションはOpenGL ESオブジェクトを解放し、そのアプリケーション自身またはほかのアプリケーションのために、さらにリソースが利用できるようにします。

この設計の特性の重要な点をまとめると、次のようになります。

- 実用的な場合は必ず静的リソースを作成する。
- 内側のレンダリングループは、動的リソースの修正とレンダリングコマンドの送信を交互に実行する。フレームの最初と最後を除き、動的リソースの修正を避けるようにする。
- 中間のレンダリング結果をアプリケーションに読み取りに戻らないようにする。

この章の残りでは、このレンダリングループ機能の実装に役立つ、OpenGL ESのプログラミングテクニックを紹介します。この後の章では、次に示す汎用テクニックをOpenGL ESプログラミングの特定の領域に適用する方法について説明します。

- 「同期とフラッシュの操作を避ける」（51 ページ）
- 「OpenGL ESの状態の照会を避ける」（52 ページ）
- 「OpenGL ESによるリソースの管理を許可」（53 ページ）
- 「バッファの二重化を使用してリソースの競合を回避する」（53 ページ）
- 「OpenGL ESの状態変数に注意を払う」（54 ページ）
- 「OpenGL ESオブジェクトによる状態変更の置き換え」（55 ページ）

同期とフラッシュの操作を避ける

OpenGL ESは、ほとんどのコマンドを即座に実行することが要求されていません。多くの場合、コマンドはコマンドバッファのキューに追加され、後でハードウェアが実行します。通常、OpenGL ESは、バッファをハードウェアに送信する前にかなりの数のコマンドをアプリケーションがキューに追加するまで待機します。グラフィックスハードウェアがコマンドを一括して実行することは、そうでない場合より効率的な場合が少なくありません。ただし、一部のOpenGL ES関数は、バッファをすぐにフラッシュしなければなりません。それ以外の関数は、バッファをフラッシュするだけでなく、アプリケーションの制御を返す前に、それまでに送信されたコマンドが完了するまでブロックされます。アプリケーションは、フラッシュコマンドと同期コマンドの使用を、その動作が必要な場合だけに制限するべきです。フラッシュコマンドや同期コマンドを使いすぎると、ハードウェアによるレンダリングが完了するのを待ってアプリケーションが停止する可能性があります。

次に示す状況では、OpenGL ESは、実行に備えコマンドバッファをハードウェアに送信する必要があります。

- glFlush関数が、コマンドバッファをグラフィックスハードウェアに送信するとき。この関数は、コマンドがハードウェアに送信されるまでブロックされますが、コマンドの実行完了までは待ちません。
- glFinish関数が、それまでに送信されたすべてのコマンドがグラフィックスハードウェアで実行し終わるのを待っているとき。
- OpenGLの状態を取得する関数（glGetErrorなど）も送信されたコマンドが完了するのを待っているとき。
- コマンドバッファが一杯になっているとき。

glFlushの効果的な使用

ほとんどの場合、画像データを画面に移動するのにglFlushを呼び出す必要はありません。glFlush関数の呼び出しが有用な場合はごくわずかです。

- アプリケーションが特定のOpenGL ESオブジェクトを使用するレンダリングコマンドを送信し、しばらくしてからそのオブジェクトを修正しようとしている場合（またはその逆の場合）。未処理の描画コマンドがあるOpenGL ESオブジェクトを修正しようとする場合、アプリケーションはそれらの描画コマンドが完了するまで停止する可能性があります。この状況では、glFlushを呼び出すことで、ハードウェアがただちにコマンドの処理を開始します。コマンドバッファをフラッシュした後、アプリケーションは送信したコマンドと並行して操作可能な作業を実行するべきです。
- 2つのコンテキストがOpenGL ESオブジェクトを共有している場合。共有されているオブジェクトを修正するOpenGL ESコマンドを送信した後、ほかのコンテキストに切り替わる前にglFlushを呼び出します。
- 複数のスレッドが同じコンテキストにアクセスしている場合、1度に1つのスレッドだけがOpenGL ESにコマンドを送信するようにします。コマンドを送信した後は、glFlushを呼び出す必要があります。

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。これを避けるには、照会する必要のある状態のコピーを保持し、OpenGL ESを呼び出すのではなく、コピーに直接アクセスします。

エラーが発生すると、OpenGL ESは、glGetError関数で取得することができるエラーフラグを設定します。開発段階では、glGetErrorを呼び出すエラーチェックルーチンをコードに含めることが重要です。パフォーマンスが重視されるアプリケーションを開発する場合、エラー情報の取得は、アプリケーションのデバッグを行っている間だけ行います。リリースビルドでglGetErrorを使用しすぎるとパフォーマンスが低下します。

OpenGL ESによるリソースの管理を許可

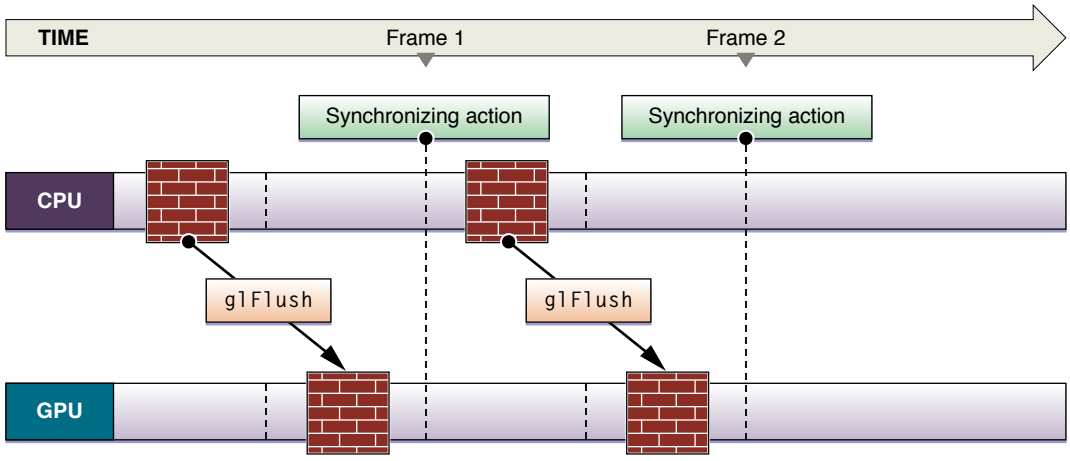
OpenGL ESでは、多くのデータ型をOpenGL ES内に永続的に格納することができます。頂点、テクスチャ、またはその他の形式のデータを格納するOpenGL ESオブジェクトを作成することで、OpenGL ESでは、データの変換およびデータのグラフィックスプロセッサへの送信によるオーバーヘッドを削減できます。データが、修正される場合と比べてより頻繁に使用される場合、OpenGL ESはアプリケーションのパフォーマンスを大幅に向上させることができます。

OpenGL ESは、データをどのように使用するかについてアプリケーションにヒントを提供します。これらのヒントにより、OpenGL ESは、データの処理方法について、情報に基づく選択を行うことができます。たとえば、静的データは、グラフィックス専用のメモリも含め、グラフィックスプロセッサが簡単にフェッチできるメモリに配置できます。

ダブルバッファリングによるリソースの競合の回避

リソースの競合は、アプリケーションとOpenGL ESが、あるOpenGL ESオブジェクトに同時にアクセスするときに発生します。一方の当事者（アプリケーションまたはOpenGL ES）が使用中のOpenGL ESオブジェクトをもう一方の当事者が修正しようとすると、両者はそのオブジェクトが使用できなくなるまでブロックされる可能性があります。オブジェクトの修正が開始されると、もう一方の当事者は修正が完了するまでそのオブジェクトへのアクセスを許可されません。あるいは、OpenGL ESが暗黙的にオブジェクトを複製して、両方の当事者がコマンドの実行を継続できるようにする場合があります。どちらの方法も安全ですが、どちらもアプリケーションのボトルネックとなるおそれがあります。図 6-4はこの問題を示しています。この例では、単一のテクスチャオブジェクトがあり、そのオブジェクトをOpenGL ESとアプリケーションの両方が使用しようとしています。アプリケーションがテクスチャを変更しようとすると、そのアプリケーションは、それまで送信された描画コマンドが完了するまで、つまりCPUとGPUが同期するまで待たなければなりません。

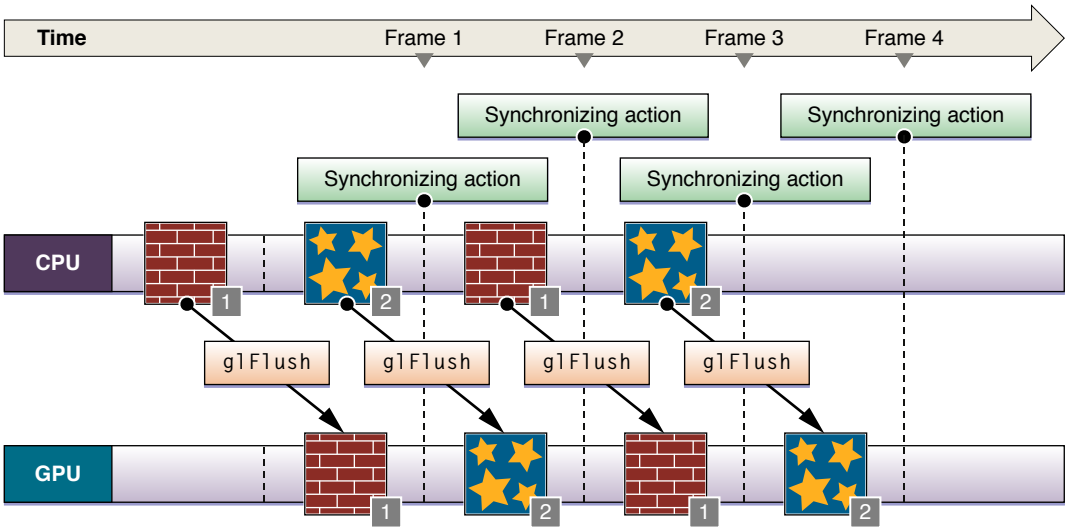
図 6-4　単一のバッファに格納されたテクスチャデータ



この問題を解決するため、アプリケーションは、オブジェクトの変更とオブジェクトに関連する描画との間で追加の作業を実行することができます。しかし、実行可能な追加の作業がアプリケーションにない場合、アプリケーションは同じサイズの2つのオブジェクトを明示的に作成します。これにより、一方の当事者がオブジェクトを読み取っている間、もう一方が残りのオブジェクトを

修正できます。図6-5に、ダブルバッファによるアプローチを示します。GPUがあるテクスチャを処理している間、CPUが別のテクスチャを修正するという方法です。最初に起動した後は、CPUもGPUもアイドル状態です。テクスチャを例にとりましたが、この方法はほとんどの種類のOpenGL ESオブジェクトで機能します。

図 6-5 ダブルバッファリングされたテクスチャデータ



ダブルバッファリングはほとんどのアプリケーションで有効ですが、両方の当事者がほぼ同時にコマンドの処理を完了する必要があります。ブロック状態を回避するため、さらにバッファを追加することができます。この場合は、従来のプロデューサ／コンシューマモデルが実装されます。コンシューマがコマンド処理を完了する前にプロデューサが処理を完了した場合、プロデューサはアイドルバッファを利用してコマンドの処理を継続します。この状況では、プロデューサは、コンシューマの処理が著しく遅れる場合のみアイドル状態になります。

ダブルバッファリングやトリプルバッファリングには、パイプラインが停止するのを防げる代わりに、メモリの消費量が増えるというトレードオフがあります。メモリの使用量が増えると、アプリケーションのほかの部分に負担がかかる可能性があります。iOSデバイスでは、メモリは貴重なリソースです。設計において、メモリの使用量を増やすことと、ほかのアプリケーションの最適化とのバランスをとる必要があります。

OpenGL ESの状態変数に注意を払う

ハードウェアは現在の状態を1つ持っており、それが集約されて、キャッシュされます。状態の切り替えは負荷がかかるため、状態の切り替えを最小限に抑えたアプリケーションを設計することが最も推奨されます。

すでに設定済みの状態は設定しないでください。ある機能を一度有効化したら、再び有効化する必要はありません。有効化関数を2回以上呼び出しても時間の無駄になるだけです。これは、`glEnable`または`glDisable`を呼び出すときに、OpenGL ESが機能の状態をチェックしないためです。たとえば、`glEnable(GL_LIGHTING)`を2回以上呼び出しても、OpenGL ESはライティングの状態がすでに有効になっているかを確認しません。OpenGL ESは、現在の値と同じ値の場合は、単にその状態値で更新するだけです。

状態を設定する呼び出しを描画ループ内に配置するのではなく、専用のセットアップルーチンまたはシャットダウンルーチンを使用することによって、不要な状態設定を避けることができます。セットアップルーチンおよびシャットダウンルーチンは、特別な視覚効果を達成する機能のオンとオフの切り替えにも有用です。たとえば、テクスチャ化されたポリゴンのワイヤフレームの輪郭を描画する場合に便利です。

2D画像を描画する場合、リスト 6-1に示すものに類似する、関連の状態変数をすべて無効にしてください。

リスト 6-1　OpenGL ES 1.1における状態変数の無効化

```
glDisable(GL_DITHER);
glDisable(GL_ALPHA_TEST);
glDisable(GL_BLEND);
glDisable(GL_STENCIL_TEST);
glDisable(GL_FOG);
glDisable(GL_TEXTURE_2D);
glDisable(GL_DEPTH_TEST);
// ほかの状態変数も適宜無効化する。
```

OpenGL ESオブジェクトによる状態変更の置き換え

「OpenGL ESの状態変数に注意を払う」（54 ページ）のセクションは、状態変更の回数を減らすことがパフォーマンスを改善する可能性があることを示唆しています。一部のOpenGL ES拡張機能では、複数のOpenGL状態変更を収集するオブジェクトを、1つの関数呼び出しにバインドすることができるオブジェクトの形で作成することもできます。このようなテクニックが可能な場合は、利用することをお勧めします。たとえば、固定機能パイプラインの設定には、さまざまな演算子の状態を変更するために数多くの関数呼び出しが必要となります。これは、呼び出される各関数にとってオーバーヘッドとなるだけでなく、コードがより複雑になり、管理しにくくなります。その代わりに、シェーダを使用します。シェーダは、コンパイル後は同じ働きをすることができますが、`glUseProgram`を一度だけ呼び出す必要があります。

たとえば、頂点配列オブジェクトでは、頂点の属性を一度設定すれば、その属性を頂点配列オブジェクトに格納することができます。詳細については、「頂点配列オブジェクトの使用による頂点配列状態の変化の整理統合」（75 ページ）を参照してください。

OpenGL ESアプリケーションのチューニング

iOSにおける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におけるOpenGLのパフォーマンスとは異なります。iOS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的であると、フレームレートが低下したり、iOSベースのデバイスのバッテリー持続時間が著しく減少したりする可能性があります。

後の章では、アプリケーションのパフォーマンスを向上させる数多くのテクニックを取り上げています。この章では、アプリケーションが従うと良い、全体的な戦略について説明します。

パフォーマンスのための一般的な推奨事項

開発に役立つInstrumentsアプリケーションと常識の活用

アプリケーションは、さまざまなデバイス上で、さまざまなシナリオに基づいてパフォーマンスをテストするまでは最適化しないでください。さらに、常識を活かし、不要な最適化は避けるようにしてください。たとえば、1フレームにつき数十個の三角形を描画するアプリケーションは、OpenGL ESが主要なボトルネックとなる可能性はありません。そのようなアプリケーションでは、アプリケーションが頂点データを送信する方法を書き直しても、アプリケーションのパフォーマンスが改善される可能性はありません。Instrumentsは、アプリケーションが処理に時間のかかっている部分に関する現実的なデータを提供することができます。詳細については、『*Instruments User Guide*』を参照してください。

シーンのデータが変更された場合のみシーンを再描画する

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。Core Animationはユーザに表示された最後の画像をキャッシュし、新しいフレームが表示されるまでその画像を表示し続けます。

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ほう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。

iOSデバイスは浮動小数点演算をネイティブでサポート

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。これらのタスクのすべてが、フレームごとに評価され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。

iOSデバイスのARMプロセッサは、浮動小数点命令をネイティブで処理します。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。固定小数点演算を使用するアプリケーションを移植する場合は、浮動小数点の型を使用してコードを書き直します。

注： iOSデバイスはすべて、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、ビルド設定の「Compile for Thumb」の選択を解除します。

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。たとえば、事前に計算を行って、結果をモデルのデータに格納できる場合は、アプリケーション実行時にその計算の実行を回避することができます。

OpenGL ES 2.0向けにアプリケーションを記述する場合は、アプリケーションがシーンのレンダリングに必要とする各タスクを実行するための、多数のスイッチと条件が設定された単一のシェーダを作成しないでください。代わりに、各シェーダが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。

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モデルを描画する場合はフォグと深度テストを無効にします。

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で処理するプリミティブを送信するたびに、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、描画をまとめて実行して呼び出し回数が減るようにします。たとえば、複数の三角形ストリップを単一のストリップにマージすることができます（「三角形ストリップを使用して頂点データを一括処理する」 （71 ページ）参照）。

OpenGLの状態の共通セットを使用するモデルを整理統合することには、OpenGL ESの状態の変更に伴うオーバーヘッドが削減されるという利点もあります。詳細については、「OpenGL ESの状態変数に注意を払う」 （54 ページ）を参照してください。

最良の結果を得るには、近接して描画されるプリミティブを整理統合します。広範囲にわたるモデルの場合、フレーム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。

メモリはiOSデバイスの貴重なリソース

iOSアプリケーションは、システムおよびほかのiOSアプリケーションとメイン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。そのことを念頭において、必要なメモリだけを割り当てて、割り当てたメモリが不要になったらできるだけ早くそのメモリの割り当てを解除します。アプリケーションがメモリを節約できる方法のいくつかの例を次に示します。

- 画像をOpenGL ESテクスチャにロードした後、元の画像を解放する。
- アプリケーションが必要としているときだけ深度バッファを割り当てる。
- アプリケーションがすべてのリソースを一度に必要としない場合は、項目のサブセットだけをロードします。たとえば、1つのゲームを複数のレベルに分割できます。各レベルは、より厳しいリソース制限に収まる、リソース全体のサブセットをロードします。

iOS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、『*iOS Application Programming Guide*』で詳しく説明しています。

一部のiOSデバイスに搭載されているPowerVR MBXプロセッサには、さらにメモリの制限があります。詳細については「PowerVR MBX」（94 ページ）を参照してください。

正しいレンダリングに必要な場合を除きレンダリングされたオブジェクトを並べ替えない

- 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時間を費やさないこと。すべてのiOSデバイス上の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)」（90 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（「アルファテストと破棄を避ける」（60 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。

ライティングモデルを簡素化する

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るアドバイスです。

- 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。より多くの演算が必要となる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。OpenGL ES 2.0の演算は、ライティング演算をモデル空間で行う必要があります。より複雑なライティングアルゴリズムではなく、よりシンプルなライティング方程式の使用を検討してください。
- 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。

アルファテストと破棄を避ける

グラフィックスハードウェアは、フラグメントの色の値を計算する前に、グラフィックスパイプラインの早い段階でデプステストを実行することがよくあります。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。特にこの最適化の無効化では、フラグメントが見えないという理由から、フラグメントの色を破棄するためだけに完全に計算する必要の生じることがあります。

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更されないため、透過プリミティブの背面から前面への並べ替えが必要となる可能性があります。

アルファテストまたは`discard`命令を使用する必要のある場合は、それを必要としないプリミティブをすべて処理してから、これらのオブジェクトを別々にシーン内に描画します。結果が使用されない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ほうに`discard`命令を配置します。

並列処理とOpenGL ES

同時並行性とは、複数の事柄が同時に発生していることを表しています。同時並行性は、コンピュータの文脈では通常、複数のプロセッサ上で同時にタスクを実行することを指します。作業を並行して実行することで、タスクがより短時間で完了し、ユーザから見たアプリケーションの応答性が高まります。しっかりした設計のOpenGL ESアプリケーションには、具体的な形の同時並行性、すなわちCPU上でのアプリケーションの処理とGPU上でのOpenGL ESの処理の同時並行性が見られます。「OpenGL ESアプリケーション設計ガイドライン」（47 ページ）で紹介するテクニックの多くは、特に、CPUとGPUの間での優れた並列処理を実現するOpenGLアプリケーションの作成を狙いとしています。並列処理アプリケーションを設計するということは、アプリケーションが実行する作業をサブタスクに分解し、並行して安全に実行できるタスクと、順番に実行しなければならないタスクを特定することを意味します。つまり、ほかのタスクが使用するリソース、またはほかのタスクから返された結果のどちらかに依存するタスクを明らかにするを意味します。

iOSにおける各プロセスは、1つ以上のスレッドで構成されています。**スレッド**とは、プロセスのためのコードを実行する、実行の流れのことです。Appleは、従来のスレッドと、**Grand Central Dispatch (GCD)**と呼ばれる機能の両方を提供します。Grand Central Dispatchを使用することで、アプリケーションにスレッドの管理を要求しなくても、アプリケーションをより小規模のタスクに分解することができます。Grand Central Dispatchは、デバイス上で利用可能なコア数に基づいてスレッドを割り当て、それらのスレッドに対して自動的にタスクのスケジューリングを行います。

より高いレベルでは、Cocoa Touchが`NSOperation`と`NSOperationQueue`を提供して、作業単位の作成とスケジューリングのためのObjective-C抽象化を可能にしています。

この章ではこれらのテクノロジーは詳しく取り上げません。OpenGL ESアプリケーションに並列処理を追加する方法を検討する前に、まず『*Concurrency Programming Guide*』をお読みください。スレッドを手動で管理する場合は、『*Threading Programming Guide*』もお読みください。使用するテクニックに関係なく、マルチスレッドシステム上でOpenGL ESを呼び出す場合に追加の制限があります。この章は、マルチスレッドによってOpenGL ESアプリケーションのパフォーマンスが向上する場合、OpenGL ESがマルチスレッドアプリケーション課す制限、そしてOpenGL ESアプリケーションに並列処理を実装するために使用できる一般的な設計方法を理解するのに役立ちます。

OpenGLアプリケーションに並列処理のメリットがあるかどうかの識別

マルチスレッドアプリケーションの作成では、アプリケーションの設計、実装、テストにおいてかなりの努力が必要です。スレッドもアプリケーションに複雑さとオーバーヘッドを追加する要素です。アプリケーションはワーカースレッドに渡すことができるようデータをコピーする必要のある場合があり、複数のスレッドは同一リソースへのアクセスを同期する必要の生じる場合があります。OpenGL ESアプリケーションに並列処理を実装しようとする前に、「OpenGL ESアプリケーション設計ガイドライン」（47 ページ）で説明するテクニックを使用して、まず単一スレッド環境でOpenGL ESコードを最適化します。最初は、CPUとGPUの効率的な並列処理の達成に焦点を合わせ、その後で並列プログラミングによってパフォーマンス上の利点をさらに提供できるかどうかについて評価します。

並列処理に適した候補には、次に示す特性のどちらか、またはその両方が備わっています。

- アプリケーションが、OpenGL ESのレンダリングに依存しない多くのタスクをCPU上で実行する。たとえば、ゲームは、ゲームの世界をシミュレートし、コンピュータに操られる敵の人工知能を計算し、サウンドを再生します。このシナリオでは、タスクの多くがOpenGL ESの描画コードに依存していないため、並列処理を活用できます。
- アプリケーションのプロファイルを調べた結果、OpenGL ESのレンダリングコードがCPUで多くの時間を費やしていることがわかりました。このシナリオでは、アプリケーションがGPUに十分な速さでコマンドを送信できていないため、GPUはアイドル状態です。CPUに結び付けられているコードがすでに最適化されている場合は、並行して実行されるタスクへと作業内容を分割することにより、アプリケーションのパフォーマンスを向上できる可能性があります。

アプリケーションがGPUを待機してブロックされている場合で、OpenGL ESによる描画と並行して実行可能な作業がない場合は、そのアプリケーションは並列処理に適した候補ではありません。CPUとGPUの両方がアイドル状態の場合、OpenGL ESのニーズはおそらく、それ以上のチューニングの必要がないほどシンプルです。

OpenGL ESは各コンテキストを単一スレッドに制限

iOSにおけるスレッドはそれぞれ、単一で現在のOpenGL ESレンダリングコンテキストです。アプリケーションがOpenGL ES関数を呼び出すごとに、OpenGL ESは現在のスレッドに関連付けられているコンテキストを暗黙的に参照し、そのコンテキストに関連付けられている状態またはオブジェクトを修正します。

OpenGL ESは再入可能ではありません。複数のスレッドに関連付けられている同じコンテキストを同時に修正する場合、結果は予測できません。アプリケーションがクラッシュする可能性もあれば、レンダリングが正常に行われない可能性もあります。何らかの理由で複数のスレッドが同じコンテキストをターゲットとするよう設定することにした場合は、そのコンテキストに対するすべてのOpenGL ES呼び出しにミューテックスを配置して、スレッドを同期させなければなりません。ブロック状態になるOpenGL ESコマンド（`glFinish`など）は、スレッドを同期しません。

Grand Central Dispatchおよび`NSOperationQueue`オブジェクトは、選択したスレッド上でタスクを実行することができます。これらは、そのタスク専用のスレッドを作成する場合もあれば、既存のスレッドを再利用する場合もあります。しかし、どちらの場合でも、どのスレッドがタスクを実行するか保証できません。これは、OpenGL ESアプリケーションにとっては、次の意味を持ちます。

- 各タスクは、OpenGL ESコマンドを実行する前にコンテキストを設定しなければならない。
- アプリケーションは、同じコンテキストにアクセスする2つのタスクの同時実行は許可されないようにしなければならない。
- 各タスクは、終了する前にコンテキストを消去する必要がある。

OpenGL ESアプリケーションに並列処理を実装する方法

並列処理が可能なOpenGL ESアプリケーションは、OpenGL ESがより多くの作業をGPUに提供できるよう、CPUの並列処理に焦点を合わせるべきです。OpenGLアプリケーションに並列処理を実装するときの推奨事項を次に示します。

- アプリケーションを、同時に実行可能なOpenGL ESのタスクとOpenGL ES以外のタスクに分解する。OpenGL ESの描画コードは単一のタスクとして実行されます。そのため、OpenGL ESの描画コードは単一のスレッドで実行されます。この方法は、アプリケーションに大量のCPU処理を必要とするほかのタスクがある場合に最も有効です。

- アプリケーションがOpenGL ESに送るデータの準備に大量のCPU時間を費やす場合は、レンダリングデータを準備するタスクと、レンダリングコマンドをOpenGL ESに送信するタスクとに作業を分割することができます。詳細については、「OpenGL ESは各コンテキストを単一スレッドに制限」（62 ページ）を参照してください。

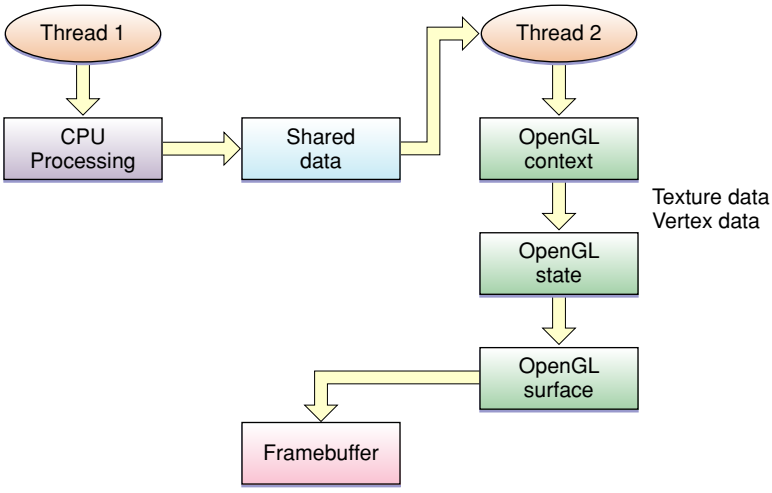
- 同時にレンダリング可能な複数のシーンがある場合、または複数のコンテキストで実行可能な作業がある場合、アプリケーションは複数のタスクを作成し、1つのタスクに1つのOpenGL ESコンテキストを設定します。複数のコンテキストが同じアートアセットにアクセスする場合は、sharegroupを使用して、コンテキスト間でOpenGL ESオブジェクトを共有します。詳細については、「EAGL sharegroupはコンテキストのOpenGL ESオブジェクトを管理」（25 ページ）を参照してください。

ワーカータスクにおけるOpenGL ES演算の実行

一部のアプリケーションは、OpenGL ESにデータを渡す前に、そのデータについて大量の演算を実行します。たとえば、新たなジオメトリを作成したり、既存のジオメトリをアニメーション化したりするアプリケーションが該当します。可能な場合は、そのような演算はOpenGL ES内部で実行します。これは、GPU内部で利用できるすぐれた並列処理を活かしており、アプリケーションとOpenGL ESの間における結果のコピーに伴うオーバーヘッドを削減します。

図 6-3（50 ページ）に示す手法は、OpenGL ESオブジェクトの更新と、それらオブジェクトを使用するレンダリングコマンドの実行を交互に実行するものです。OpenGL ESはGPU上でレンダリングを実行し、並行してCPU上ではアプリケーションの更新を実行します。CPU上で実行される演算が、GPU上の演算よりも処理時間がかかってしまうと、それだけGPUのアイドル時間が長くなります。このような状況では、複数CPUを搭載するシステムの並列処理を利用できる場合があります。OpenGL ESレンダリングコードを別の演算や処理タスクに分割して、それらを並行して実行します。図 8-1は、役割分担をわかりやすく示しています。1つ目のタスクは、2つ目のタスクが使用し、OpenGLに送信するデータを生成します。

図 8-1 別個のスレッド上のCPU処理とOpenGL



最高のパフォーマンスを得るため、アプリケーションではタスク間におけるデータのコピーを避ける必要があります。1つのタスクでデータを計算し、計算結果をほかのタスクの頂点バッファオブジェクトにコピーする代わりに、セットアップコードで頂点バッファオブジェクトをマッピングし、ポインタを直接ワーカータスクに渡します。

アプリケーションにおいて、修正タスクをサブタスクへとさらに分解できる場合は、さらにメリットの得られる可能性があります。たとえば、2つ以上の頂点バッファオブジェクトがあると仮定し、描画コマンドを送信する前にそれぞれのオブジェクトを更新する必要があるとします。各オブジェクトは、ほかのオブジェクトに関係なく計算し直すことが可能です。このシナリオでは、各バッファに対する修正が演算となり、`NSOperationQueue`オブジェクトを使用して作業を管理します。

1. 現在のコンテキストを設定します。
2. 最初のバッファをマッピングします。
3. そのバッファを埋めることがタスクとなる`NSOperation`オブジェクトを作成します。
4. その演算を演算キューに追加します。
5. ほかのバッファについても手順2から手順4を実行します。
6. 演算キューに対して`waitUntilAllOperationsAreFinished`を呼び出します。
7. バッファのマッピングを解除します。
8. レンダリングコマンドを実行します。

複数のOpenGL ESコンテキストの使用

並行してレンダリング可能な複数のシーンがアプリケーションにある場合は、レンダリングの必要なシーンごとにコンテキストを使用することができます。1つのシーンにつき1つのコンテキストを作成し、各コンテキストを1つの演算またはタスクに割り当てます。それぞれのタスクは、タスク固有のコンテキストを持つため、どのタスクもレンダリングコマンドを並行して送信できます。

複数のコンテキストを使用する場合の一般的な手法とは、各コンテキストが別々のスレッドで実行されている状態で、ほかのコンテキストがOpenGL ESオブジェクトを使用している間にOpenGL ESオブジェクトを更新する1つのコンテキストを持つことです。各コンテキストは別々のスレッド上で実行されるため、コンテキストのアクションがほかのコンテキストにブロックされることはめったにありません。これを実装するには、アプリケーションにおいて2つのコンテキストと2つのスレッドを作成し、各スレッドが1つのコンテキストを制御するようにします。さらに、2つ目のスレッド上でアプリケーションが更新しようとするOpenGL ESオブジェクトは、二重にバッファに格納しなければなりません。これは、オブジェクトを使用する側のスレッドが、もう一方のスレッドがOpenGL ESオブジェクトを修正している間は、OpenGL ESオブジェクトにアクセスできない可能性があるからです。コンテキスト間の変更を同期するプロセスは、「EAGL sharegroupはコンテキストのOpenGL ESオブジェクトを管理」（25 ページ）で詳しく説明しています。

OpenGL ESアプリケーションのスレッド化に関するガイドライン

OpenGL ESを使用するアプリケーションでのスレッド化を成功させるには、次のガイドラインに従ってください。

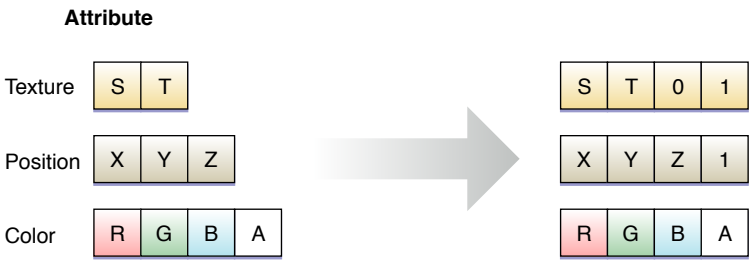
- 1つのコンテキストにつき1つのスレッドだけを使用する。特定のコンテキストのためのOpenGL ESコマンドはスレッドセーフではありません。決して、複数のスレッドが単一のコンテキストに同時にアクセスしないようにしてください。
- Grand Central Dispatchを使用するときは、専用のシリアルキューを使用してOpenGL ESにコマンドをディスパッチする。このキューは、従来のミューテックスパターンを置き換えるために使用できます。
- 現在のコンテキストを追跡する。スレッドを切り替えるとき、何も考えずにコンテキストを切り替えることは簡単ですが、そうれによりグラフィックスコマンドの実行に予期しない影響が生じます。新たに作成されたスレッドに切り替えるときは現在のコンテキストを設定し、元のスレッドを離れる前に現在のコンテキストを消去しておかなければなりません。

頂点データを扱うためのベストプラクティス

OpenGL ESを使用するフレームをレンダリングするには、アプリケーションにおいてグラフィックスパイプラインを設定し、描画するグラフィックスプリミティブを送信します。すべてのプリミティブを同じパイプライン設定を使用して描画するアプリケーションもあれば、フレームを構成する異なる要素を別々のテクニックを使用して描画するアプリケーションもあります。しかし、アプリケーションがどのプリミティブを使用するか、またはパイプラインがどのように設定されているかに関係なく、アプリケーションはOpenGL ESに頂点を提供します。この章では、頂点データに関する有用な情報を提供し、頂点データを効率的に処理する方法について、対象を絞ったアドバイスを紹介します。

頂点は、位置、色、法線、テクスチャの座標など、1つ以上の**属性**で構成されています。OpenGL ES 2.0アプリケーションは、独自の属性を自由に定義できます。頂点データの各属性は、頂点シェーダへの入力の働きをする、属性変数と対応しています。OpenGL 1.1アプリケーションは、固定機能パイプラインによって定義される属性を使用します。

アプリケーションは、1～4個の**コンポーネント**で構成されるベクトルとして属性を定義します。属性内のすべてのコンポーネントが、共通のデータ型を共有します。たとえば、色は4個のGLbyteコンポーネント（アルファ、赤、緑、青）として定義される場合があります。属性がシェーダ変数にロードされるとき、アプリケーションデータ内で提供されていないコンポーネントは、OpenGL ESのデフォルト値で埋められます。最後のコンポーネントは`1`で埋められ、未指定のほかのコンポーネントは`0`で埋められます。



アプリケーションが属性を*定数*として設定する可能性があるということは、描画コマンドの一部として送信されるすべての頂点について同じ値が使用されることを意味し、*配列*として設定する可能性があるということは、各頂点がその属性の値であることを意味します。アプリケーションがOpenGL ESの関数を呼び出して一連の頂点を描画する場合、頂点データはアプリケーションからグラフィックスハードウェアにコピーされます。そして、グラフィックスハードウェアは頂点データを扱い、シェーダで各頂点を処理し、プリミティブを組み立て、プリミティブをラスタ化してフレームバッファに送ります。OpenGL ESの利点の1つは、頂点データをOpenGL ESに送信する関数の単一のセットを標準として採用していて、OpenGLが提供していた古くて効率の悪い仕組みは削除されていることです。

フレームをレンダリングするため多数のプリミティブを送信しなければならないアプリケーションは、頂点データおよび頂点データのOpenGL ESへの提供方法を注意深く管理する必要があります。この章で説明する方法は、次に示すいくつかの基本原則に要約することができます。

- 頂点データのサイズを減らす。

- OpenGL ESがグラフィックスハードウェアに頂点データを転送する前に必ず発生する前処理を減らす。
- グラフィックスハードウェアに頂点データをコピーするのにかかる時間を減らす。
- 各頂点について実行される演算を減らす。

モデルの簡素化

iOSベースのデバイスのグラフィックスハードウェアは非常に高性能ですが、デバイスの表示する画像は非常に小さいことがよくあります。iOS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。モデルの描画に使用する頂点の数を減らすことは、頂点データのサイズおよび頂点データについて実行される演算を減らすことに直接つながります。

モデルの複雑さは、次に示すテクニックを使用することで低減できます。

- 異なる詳細レベルのモデルを複数バージョン用意し、カメラからのオブジェクトの距離とディスプレイの寸法に基づいて、適切なモデルを実行時に選ぶ。
- テクスチャを使用して、一定の頂点情報を不要にする。たとえば、バンプマップは、頂点データをさらに追加することなくモデルにディテールを追加するのに使用できます。
- モデルによっては頂点を追加して、ライティングのディレールやレンダリングの質を向上させる。これは通常、値を各頂点について計算し、ラスタ化の段階で三角形を対象に値を補間する場合に行われます。たとえば、スポットライトを三角形の中心に向ける場合、スポットライトの最も明るい部分は頂点に向けられていないため、スポットライトの効果は気付かれない可能性があります。頂点を追加することで、頂点データのサイズやモデルについて実行される演算が増加するという犠牲を払って、追加の補間ポイントが提供されます。追加の頂点を加える代わりに、パイプラインのフラグメントの段階に演算を移動することを検討します。
    - アプリケーションでOpenGL ES 2.0を使用する場合、アプリケーションは頂点シェーダの演算を実行し、varying変数に結果を割り当てます。varying値は、グラフィックスハードウェアによって補間され、入力としてフラグメントシェーダに渡されます。代わりに、演算の入力をvarying変数に割り当てて、フラグメントシェーダで演算を実行します。これにより、その演算を実行するコストが頂点単位のコストからフラグメント単位のコストに変わり、パイプラインにおける頂点の段階の負荷と、それよりも大きいフラグメントの段階の負荷が低減されます。これは、頂点処理においてアプリケーションがブロックされる場合、計算の負荷が大きくない場合、および変更によって頂点の数が大幅に減少する可能性のある場合に実行します。
    - アプリケーションでOpenGL ES 1.1を使用する場合は、DOT3ライティングを使用してフラグメント単位のライティングを実行できます。これは、具体的には、法線情報を保持するバンプマップテクスチャの追加、および`GL_DOT3_RGB`モードでのテクスチャ合成操作を使用したバンプマップの適用によって実行します。

属性の配列に定数を格納するのを避ける

アプリケーションのモデルにモデル全体において固定されたままのデータを使用する属性が含まれる場合は、各頂点についてそのデータを複製しないでください。OpenGL ES 2.0アプリケーションは、固定的な頂点属性を設定することも、その代わりに値を保持する一定のシェーダ値を使用することもできます。OpenGL ES 1.1アプリケーションでは、`glColor4ub`や`glTexCoord2F`などの、頂点単位の属性関数を使用します。

属性には可能な限り最小の型を使用する

属性のコンポーネントそれぞれ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データ型を選びます。次に、いくつかのガイドラインを示します。

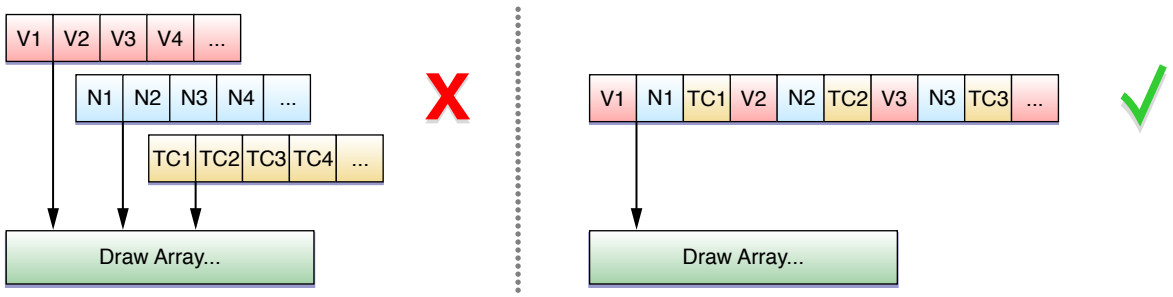
- 頂点の色は、4個の符号なしバイトコンポーネント(`GL_UNSIGNED_BYTE`)を使用して指定する。
- 2個または4個の符号なしバイト値(`GL_UNSIGNED_BYTE`)、または符号なしshort値(`GL_UNSIGNED_SHORT`)を使用してテクスチャの座標を指定する。テクスチャ座標の複数のセットを単一の属性に詰め込まないようにします。
- OpenGL ESの`GL_FIXED`データ型の使用を避ける。このデータ型は、`GL_FLOAT`と同じ量のメモリを必要としますが、値の範囲はより狭くなります。すべてのiOSデバイスがハードウェアの浮動小数点ユニットをサポートしており、浮動小数点の値をよりすばやく処理することができます。

比較的小さいコンポーネントを指定する場合は、頂点データのずれを避けるために頂点フォーマットを必ず再編成してください。詳細については、「ずれた頂点データを避ける」（70 ページ）を参照してください。

インターリーブされた頂点データを使用する

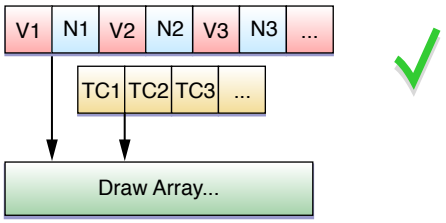
OpenGL ESを使用することで、アプリケーションは頂点データを、連続する配列（*配列の構造体*または各要素が複数の属性を含む配列（*構造体の配列*）として指定することができます。iOSに適した形式は、インターリーブされた単一の頂点フォーマットを使用する、構造体の配列です。インターリーブされたデータを使用すると、各頂点について、より優れたメモリ配置が可能となります。

図 9-1　インターリーブされたメモリ構造では、ある頂点に関するすべてのデータがまとめてメモリに配置されます。



このルールの例外は、一部の頂点データをそれ以外の頂点データとは異なるレートで更新する必要のある場合、または一部のデータを2つ以上のモデルで共有可能な場合です。どちらの場合も、属性データを2つ以上の構造体に分けることは可能です。

図 9-2 一部のデータの使われ方が異なる場合に複数の頂点構造体を使用する

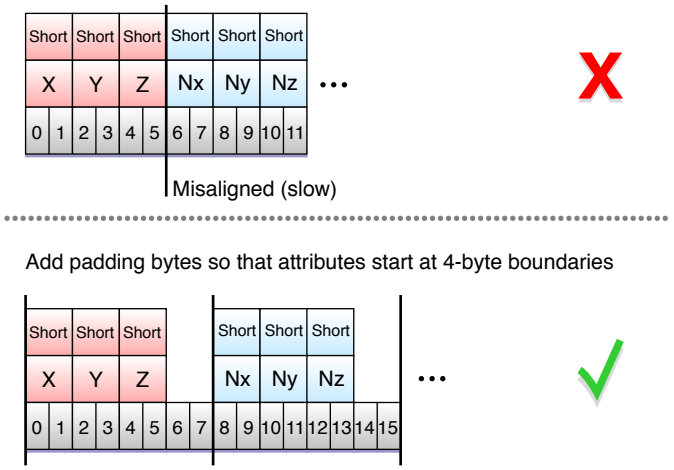


ずれた頂点データを避ける

頂点構造体を設計するときは、各属性の先頭をオフセット（コンポーネントのサイズの倍数または4バイトのどちらか大きい方）に揃えます。属性がずれている場合、iOSはグラフィックスハードウェアにデータを渡す前に追加の処理を実行しなければなりません。

図 9-3 （70 ページ）では、位置と法線のデータがそれぞれ3つのshort整数として、計6バイトで定義されています。法線データは、オフセット6から開始しています。これはネイティブサイズ（2バイト）の倍数ですが、4バイトの倍数ではありません。この頂点データがiOSに送信されたとすると、iOSは、グラフィックスハードウェアにデータを渡す前に、そのデータをコピーして揃えるための時間をさらに費やす必要があります。これを修正するため、各属性の後に2バイトのパディングバイトを明示的に追加します。

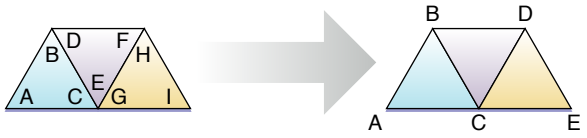
図 9-3 追加の処理を避けるために頂点データを揃える



三角形ストリップを使用して頂点データを一括処理する

三角形ストリップを使用することで、OpenGL ESがモデルに対して実行しなければならない頂点計算の数が大幅に減少します。図 9-4の左側では、合計9個の頂点を使用して3個の三角形が指定されています。C、E、Gは実際には同じ頂点を指定しています。データを三角形ストリップとして指定することにより、頂点の数を9個から5個に減らすことができます。

図 9-4 三角形ストリップ

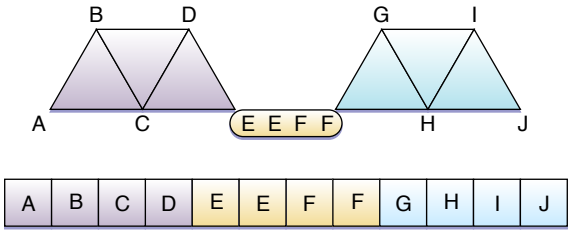


場合によっては、複数の三角形ストリップを、より大きな単一の三角形ストリップとして統合することができます。すべての三角形ストリップは、レンダリング要件が同じでなければなりません。これは次のことを意味します。

- OpenGL ES 2.0アプリケーションは、同じシェーダを使用してすべての三角形ストリップを描画しなければなりません。
- OpenGL ES 1.1アプリケーションは、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。
- 三角形ストリップは、頂点の属性が同じでなければなりません。

2つの三角形ストリップをマージするには、図 9-5に示すように、1つ目のストリップの最後の頂点と、2つ目のストリップの最初の頂点を二重にします。このストリップをOpenGL ESに送信すると、三角形DEE、三角形EEF、三角形EFF、三角形FFGは縮退したとみなされて、処理もラスタ化もされません。

図 9-5 縮退三角形を使用して三角形ストリップをマージする



最高のパフォーマンスを得るため、モデルは、`glDrawArrays`を使用し、頂点の重複をできるだけ少なくして、インデックスなしの単一の三角形ストリップとして送信する必要があります。モデルにおいて、（三角形ストリップとして連続的に現れない三角形によって多くの頂点が共有されるため、またはアプリケーションが数多くの小さな三角形ストリップをマージしたため）多くの頂点を重複させる必要のある場合、独立したインデックスバッファを使用し、`glDrawElements`を呼び出すことでより優れたパフォーマンスが得られる可能性があります。インデックスの有無はトレードオフの関係にあり、インデックスなしの三角形ストリップが全体の頂点を定期的に二重にしなければならないのに対し、インデックス付き三角形リストはインデックス用に追加のメモリが必要とな

り、頂点の検索にオーバーヘッドが加わります。最良の結果を得るため、インデックス付きとインデックスなしの両方の三角形ストリップを使用してテストを行い、最も高速なモデルを採用します。

可能な場合は、同じ頂点を共有する三角形が三角形ストリップ内でお互い適度に近い位置で描画されるよう、頂点とインデックスのデータを並べ替えます。グラフィックスハードウェアは最近の頂点計算をキャッシュしていることが多く、計算結果をローカルに参照できることで、グラフィックスハードウェアが頂点を複数回計算するのを避けられる可能性があります。

頂点バッファオブジェクトの使用による頂点データのコピーの管理

リスト 9-1では、シンプルなアプリケーションが頂点シェーダに位置と色のデータを提供するために使用できる関数を示しています。この関数では2つの属性が有効化され、それぞれの属性がインターリーブされた頂点構造体を参照するよう設定されています。最後に、`glDrawElements`関数を呼び出して、モデルを単一の三角形ストリップとしてレンダリングします。

リスト 9-1　OpenGL ES 2.0への頂点データの送信

```
typedef struct _vertexStruct
{
    GLfloat position[2];
    GLubyte color[4];
} vertexStruct;
enum {
   ATTRIB_POSITION,
   ATTRIB_COLOR,
   NUM_ATTRIBUTES };

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

    glVertexAttribPointer(ATTRIB_POSITION, 2, GL_FLOAT, GL_FALSE,
sizeof(vertexStruct), &vertices[0].position);
    glEnableVertexAttribArray(ATTRIB_POSITION);
    glVertexAttribPointer(ATTRIB_COLOR, 4, GL_UNSIGNED_BYTE, GL_TRUE,
sizeof(vertexStruct), &vertices[0].color);
    glEnableVertexAttribArray(ATTRIB_COLOR);

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

このコードは動作はしますが、非効率的です。`DrawModel`が呼び出されるごとにインデックスデータと頂点データがOpenGL ESにコピーされ、グラフィックスハードウェアに転送されます。呼び出しと呼び出しの間で頂点データが変化しなければ、データの不要なコピーがパフォーマンスに影響を及ぼすおそれがあります。不必要なコピーを避けるために、アプリケーションが頂点データを**頂点バッファオブジェクト**(VBO)に格納するべきです。OpenGL ESは頂点バッファオブジェクトのメモ

リを保有しているため、グラフィックスハードウェアがよりアクセスしやすいメモリにバッファを格納することも、データがグラフィックスハードウェアに適したフォーマットになるよう事前に処理することもできます。

リスト 9-2では、頂点バッファオブジェクトのペア（1つ目は頂点データの保持用、2つ目はストリップのインデックス用）を作成します。どちらのオブジェクトの場合も、コードは新しいオブジェクトを生成し、そのオブジェクトを現在のバッファにバインドし、バッファを埋めます。`CreateVertexBuffers`は、アプリケーションの初期化時に呼び出されます。

リスト 9-2　頂点バッファオブジェクトの作成

```
GLuint    vertexBuffer;
GLuint    indexBuffer;

void CreateVertexBuffers()
{

    glGenBuffers(1, &vertexBuffer);
    glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    glGenBuffers(1, &indexBuffer);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
GL_STATIC_DRAW);

}
```

リスト 9-3はリスト 9-1 （72 ページ）を修正し、頂点バッファオブジェクトを使用しています。リスト 9-3の主な違いは、`glVertexPointer`関数と`glColorPointer`関数のパラメータが、頂点配列を指し示さなくなったという点です。その代わり、各パラメータは、頂点バッファオブジェクトへのオフセットとなります。

リスト 9-3　OpenGL ES 2.0における頂点バッファオブジェクトを使用した描画

```
void DrawModelUsingVertexBuffers()
{
    glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
    glVertexAttribPointer(ATTRIB_POSITION, 2, GL_FLOAT, GL_FALSE,
sizeof(vertexStruct), (void*)offsetof(vertexStruct,position));
    glEnableVertexAttribArray(ATTRIB_POSITION);
    glVertexAttribPointer(ATTRIB_COLOR, 4, GL_UNSIGNED_BYTE, GL_TRUE,
sizeof(vertexStruct), (void*)offsetof(vertexStruct,color);
    glEnableVertexAttribArray(ATTRIB_COLOR);

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

バッファの使用に関するヒント

前の例は、頂点データが初期化された後、レンダリングの間は一切変更する必要がないことが前提でした。ただし、OpenGL ESでは、実行時にバッファ内のデータを変更することが可能です。頂点バッファオブジェクトの設計の重要な部分は、バッファに格納されているデータの使いかたをアプ

リケーションがOpenGL ESに通知できるということです。用途のパラメータにより、OpenGL ESは、アプリケーションがデータを使用する目的に応じて、バッファに格納する方法を選ぶことができます。リスト 9-2 （73 ページ）では、glBufferData関数の各呼び出しの最後のパラメータに用途のヒントが指定されています。GL_STATIC_DRAWをglBufferDataに渡すことにより、両方のバッファの内容が変更される予定はまったくないことがOpenGL ESに伝えられ、それによってOpenGL ESはデータを格納する方法と場所を最適化するための機会がさらに与えられます。

OpenGL ESは次の用途に対応しています。

- 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オブジェクトを作成し、作成したオブジェクトを必要なくなるまで使用します。

- GL_DYNAMIC_DRAWは、バッファに格納されているデータがレンダリングループ中に変化する可能性のある場合に使用します。アプリケーションは、初期化時にこれらのバッファを初期化し、glBufferSubData関数を呼び出すことによって必要なデータを更新します。

- 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、アプリケーションがフレームごとに新しい頂点データを動的に計算しなければならない場合で、シェーダ内部で計算を実行する場合に最も有用です。通常、アプリケーションがストリーム描画を使用している場合、異なる2つのバッファの作成を考えます。2つ作成する場合、OpenGL ESが一方のバッファの内容を使用して描画する一方、アプリケーションはもう一方のバッファを埋めます。詳細については、「バッファの二重化を使用してリソースの競合を回避する」 （53 ページ）を参照してください。ストリーム描画はOpenGL ES 1.1では利用できません。代わりにGL_DYNAMIC_DRAWを使用します。

頂点フォーマット内のさまざまな属性が異なる使用パターンを必要とする場合は、頂点データを複数の構造体に分割し、使用方法の特徴が共通している属性のコレクションごとに独立した頂点バッファオブジェクトを割り当てます。リスト 9-4は、独立したバッファを使用して色データを保持するよう、前の例を修正したものです。GL_DYNAMIC_DRAWというヒントを使用して色バッファを割り当てることにより、OpenGL ESは、アプリケーションが適切なパフォーマンスを維持するようそのバッファを割り当てることができます。

リスト 9-4　複数の頂点バッファオブジェクトの使用によるモデルの描画

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

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

// 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;

const vertexStatic staticVertexData[] = {...};
vertexDynamic dynamicVertexData[] = {...};
const GLubyte indices[] = {...};

void CreateBuffers()
```

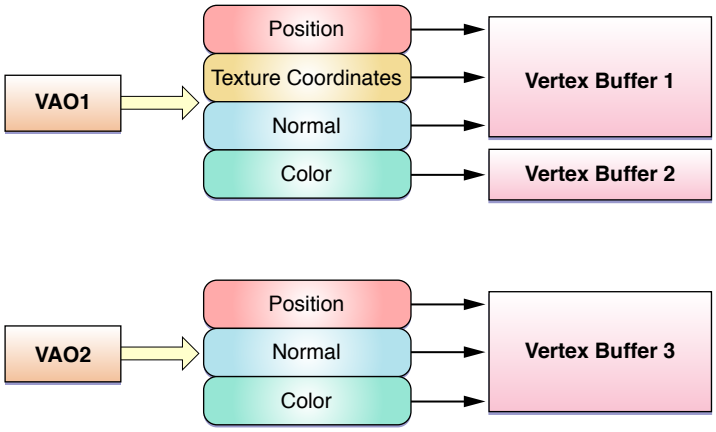
```
{
// 静的な位置データ
    glGenBuffers(1, &staticBuffer);
    glBindBuffer(GL_ARRAY_BUFFER, staticBuffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(staticVertexData), staticVertexData,
GL_STATIC_DRAW);

// 動的な色データ
// ここに示されていないが、このバッファ内のデータはフレーム間で変化する可能性あり
    glGenBuffers(1, &dynamicBuffer);
    glBindBuffer(GL_ARRAY_BUFFER, dynamicBuffer);
   glBufferData(GL_ARRAY_BUFFER, sizeof(dynamicVertexData), dynamicVertexData,
 GL_DYNAMIC_DRAW);

// 静的なインデックスデータ
    glGenBuffers(1, &indexBuffer);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
GL_STATIC_DRAW);
}

void DrawModelUsingMultipleVertexBuffers()
{
    glBindBuffer(GL_ARRAY_BUFFER, staticBuffer);
    glVertexAttribPointer(ATTRIB_POSITION, 2, GL_FLOAT, GL_FALSE,
sizeof(vertexStruct), (void*)offsetof(vertexStruct,position));
    glEnableVertexAttribArray(ATTRIB_POSITION);

    glBindBuffer(GL_ARRAY_BUFFER, dynamicBuffer);
    glVertexAttribPointer(ATTRIB_COLOR, 4, GL_UNSIGNED_BYTE, GL_TRUE,
sizeof(vertexStruct), (void*)offsetof(vertexStruct,color);
    glEnableVertexAttribArray(ATTRIB_COLOR);

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

頂点配列オブジェクトの使用による頂点配列状態の変化の整理統合

リスト 9-4 （74 ページ）の`DrawModelUsingMultipleVertexBuffers`関数を詳しく見てみます。この関数により、多くの属性が使用可能になり、複数の頂点バッファオブジェクトがバインドされ、バッファを参照するよう属性が設定されます。その初期化コードのすべてが基本的に静的であり、フレームからフレームの間で変化するパラメータはありません。この関数がアプリケーションがフレームをレンダリングするごとに呼び出される場合は、グラフィックスパイプラインを設定し直すという不要なオーバーヘッドが多数存在しています。アプリケーションが数多くの異なる種類のモデルを描画する場合、パイプラインの再設定は真のボトルネックとなる可能性があります。これを避けるには、頂点配列オブジェクトを使用して、すべての属性設定を格納します。`OES_vertex_array_object`拡張機能は、iOS 4.0以降のすべてのiOSデバイスで利用できます。

図 9-6は、2つの頂点配列オブジェクトを使用した設定例です。各設定はもう一方の設定から独立しており、それぞれが参照する頂点の数も参照先の頂点バッファオブジェクトも異なります。

図 9-6 頂点配列オブジェクトの設定



リスト 9-5は、上記の1つ目の頂点配列オブジェクトの設定に使用するコードを示しています。このコードは新しい頂点配列オブジェクトの識別子を生成し、さらに頂点配列オブジェクトをコンテキストにバインドします。この後、コードが頂点配列オブジェクトを使用しなかった場合と同じ呼び出しを行って頂点属性を設定します。設定は、コンテキストではなく、バインドされた頂点配列オブジェクトに格納されます。

リスト 9-5 頂点配列オブジェクトの設定

```
void ConfigureVertexArrayObject()
{
    // 頂点配列オブジェクトを作成してバインドする。
    glGenVertexArraysOES(1,&vao1);
    glBindVertexArrayOES(vao1);
     // VAOの属性を設定する。
    glBindBuffer(GL_ARRAY_BUFFER, vbo1);
   glVertexAttribPointer(ATT_POSITION, 3, GL_FLOAT, GL_FALSE, sizeof(staticFmt),
 (void*)offsetof(staticFmt,position));
    glEnableVertexAttribArray(ATT_POSITION);
    glVertexAttribPointer(ATT_TEXCOORD, 2, GL_UNSIGNED_SHORT, GL_TRUE,
sizeof(staticFmt), (void*)offsetof(staticFmt,texcoord));
    glEnableVertexAttribArray(ATT_TEXCOORD);
   glVertexAttribPointer(ATT_NORMAL, 3, GL_FLOAT, GL_FALSE, sizeof(staticFmt),
 (void*)offsetof(staticFmt,normal));
    glEnableVertexAttribArray(ATT_NORMAL);

    glBindBuffer(GL_ARRAY_BUFFER, vbo2);
    glVertexAttribPointer(ATT_COLOR, 4, GL_UNSIGNED_BYTE, GL_TRUE,
sizeof(dynamicFmt), (void*)offsetof(dynamicFmt,color));
    glEnableVertexAttribArray(ATT_COLOR);

    // デフォルトの状態にバインドする。
    glBindBuffer(GL_ARRAY_BUFFER,0);
    glBindVertexArrayOES(0); }
```

描画するには、コードによって頂点配列オブジェクトをバインドし、これまでと同じように描画コマンドを送信します。

最高のパフォーマンスを得るため、アプリケーションが各頂点配列オブジェクトを設定したら、実行時には決して変更しないようにします。その代わり、頂点配列オブジェクトをフレームごとに変更する必要のある場合は、複数の頂点配列オブジェクトを作成します。たとえば、バッファの二重化を使用するアプリケーションは、頂点配列オブジェクトの1つ目のセットの奇数フレーム用に、そして2つ目のセットを偶数フレーム用に設定することができます。頂点配列オブジェクトの各セットは、そのセットのフレームのレンダリングに使用する頂点バッファオブジェクトを参照します。頂点配列オブジェクトの設定が変わらない場合、OpenGL ESは頂点フォーマットに関する情報をキャッシュして、頂点属性を処理する方法を改善することができます。

テクスチャデータを扱うためのベストプラクティス

テクスチャデータは、アプリケーションがフレームのレンダリングに使用するデータのうち最も大きい部分を占めることが少なくありません。それは、テクスチャが美しい画像をユーザに表示するために必要なディテールを提供してくれるためです。アプリケーションからできるだけ最高のパフォーマンスを引き出すため、アプリケーションのテクスチャを注意深く管理しなければなりません。テクスチャの管理に関するガイドラインは次のとおりです。

- テクスチャの使用するメモリの量を減らす。
- アプリケーションの初期化時にテクスチャを作成し、レンダリングループの間はテクスチャを決して変更しない。
- 比較的小さいテクスチャをより大きいテクスチャアトラスにまとめる。
- ミップマップを使用して、テクスチャデータをフェッチするのに必要な帯域幅を減らす。
- マルチテクスチャリングを使用してテクスチャリング演算を実行する。

テクスチャのメモリ使用量を削減する

iOSアプリケーションが使用するメモリ量を減らすことは、常にアプリケーションのチューニングにおける重要な部分です。ただし、OpenGL ESアプリケーションも、テクスチャをロードするために使用できる総メモリ量が制限されています。PowerVR MBXグラフィックスハードウェアを使用するiOSデバイスでは、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」（94 ページ）参照）。アプリケーションがテクスチャデータの保持に使用するメモリ量をできるだけ抑えるようにします。テクスチャが使用するメモリを減らすことは、必ずといっていいほど画像の品質を犠牲にします。そのため、アプリケーションにおいて、テクスチャに加える変更と、最終的にレンダリングされたフレームの品質レベルとのバランスをとらなければなりません。最良の結果を得るため、以下に示す別のテクニックを試し、許容できる品質レベルで最もメモリが節約されるテクニックを選びます。

テクスチャの圧縮

テクスチャの圧縮は通常、メモリの節約と品質のバランスが最もよくとれる方法です。iOS用OpenGL ESは、`GL_IMG_texture_compression_pvrtc`拡張機能を実装することによって、PowerVR Texture Compression (PVRTC)フォーマットをサポートします。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圧縮率はそれぞれ8:1と16:1です。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」（28 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」（101 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、より低精度のピクセルフォーマットの使用を検討します。フォーマットが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。

適切なサイズのテクスチャを使用する

iOSベースのデバイス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画像の品質は高く、しかもメモリの使用量も少なくて済みます。

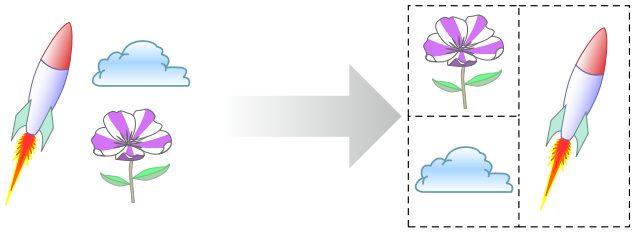
初期化中にテクスチャをロードする

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。現時点では、すべてのiOSデバイス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`関数と`glCopyTexSubImage`関数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)」（90 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にまとめる

テクスチャへのバインドでは、OpenGL ESによる処理時時間がかかります。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ほうがパフォーマンスが高くなります。テクスチャに関して、新たなテクスチャへのバインドを避ける方法の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**と呼ぶ）にすることです。テクスチャアトラスを使用することで、アプリケーションは単一のテクスチャをバインドし、そのテクスチャを使用する複数の描画呼び出しを行えるようになります。頂点データで提供されるテクスチャの座標は、テクスチャアトラス内のテクスチャのより小さい部分を選ぶよう修正されます。



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つのテクスチャであるため、OpenGL ES実装の制限を受けます。特に、実装が許可する最大テクスチャサイズが制限されます。

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り、グラフィックスハードウェアが必要とするメモリの帯域幅が削減され、パフォーマンスが向上します。

`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。

ミップマップとテクスチャアトラスを組み合わせる場合、APPLE_texture_max_level拡張機能を使用してテクスチャのフィルタリング方法を制御します。

マルチパスの代わりにマルチテクスチャリングを使用する

多くのアプリケーションが、パスごとにグラフィックスパイプラインの設定を変えながら、複数のパスを実行してモデルを描画します。これは、グラフィックスパイプラインを設定し直す回数が増えるだけでなく、パスごとに頂点情報を再処理したり、後半のパスではピクセルデータをフレームバッファから読み直したりする必要が生じます。

iOSにおけるOpenGL ESの実装は、少なくとも2つのテクスチャユニットをサポートし、ほとんどのデバイスが最大8個のテクスチャユニットをサポートします。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、各パスのアルゴリズムにおいてできるだけ多くの手順を実行する必要があります。`glGetIntegerv`関数を呼び出し、パラメータとして`GL_MAX_TEXTURE_UNITS`を渡すことにより、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すること。
- 2つ目以降のパスでは、`GL_EQUAL`をパラメータとして`glDepthFunc`関数を呼び出して、モデルの表面上のピクセルをテストすること。

テクスチャ画像データのロード前におけるテクスチャパラメータの設定

リスト 10-1に示すように、テクスチャデータをロードする前に必ずテクスチャパラメータを設定します。パラメータを最初に設定することで、OpenGL ESは、設定に適合するグラフィックスハードウェアに提供するテクスチャデータを最適化することができます。

リスト 10-1　新しいテクスチャのロード

```
glGenTextures(1, &spriteTexture);
glBindTexture(GL_TEXTURE_2D, spriteTexture);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, spriteData);
```

シェーダのベストプラクティス

シェーダはアプリケーションに高い柔軟性をもたらしますが、多くの計算を実行しすぎたり、計算の効率が悪かったりすると大きなボトルネックとなるおそれがあります。

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 11-1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 11-1　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);
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、および検証を行います。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。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、コンパイル中にエラーが発生しなかったことを確認する必要があります（リスト 11-1を参照）。

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。各シェーダはデフォルトの精度を定義しなければなりません。個々のシェーダ変数はこの精度をオーバーライドして、その変数のアプリケーションにおける使用方法についてコンパイラにヒントを提供する場合があります。OpenGL ESの実装では、ヒント情報を使用する必要はありませんが、より効率の良いシェーダを生成するためにヒント情報を使用することは可能です。GLSL ES仕様には、各ヒントの範囲と精度の一覧が示されています。

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。そのため、データがこの範囲に収まっていると想定することはできません。

iOSアプリケーションの場合、次のガイドラインに従ってください。

- 判断がつかない場合は、高精度をデフォルトにする。
- `0.0`〜`1.0`の範囲の色は通常、低精度の変数を使用して表現できる。
- 位置データは通常、高精度で格納します。
- ライティングの計算で使用される法線とベクトルは通常、中精度で格納できます。
- 精度を下げた後は、アプリケーションを再テストして、結果が期待通りのものになるようにします。

リスト 11-2は高精度の変数がデフォルトですが、高精度は不要なため低精度の変数を使用して色出力を計算します。

リスト 11-2　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);
}
```

ベクトル計算の時間のかかる実行

すべてのグラフィックスプロセッサにベクトルプロセッサが含まれているわけではなく、スカラープロセッサ上でベクトル計算を実行するグラフィックスプロセッサもあります。シェーダで計算を実行する場合は、スカラープロセッサ上計算が実行される場合であっても効率的に実行されるよう、演算の順序を考慮します。

リスト 11-3のコードがベクトルプロセッサ上で実行される場合、各乗算は4個のベクトルのコンポーネントすべてにわたって並行して実行されます。ただし、括弧の位置が理由で、3つのパラメータのうち2つがスカラー値であっても、プロセッサ上の同じ演算で8回の乗算を要することになります。

リスト 11-3　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;
```

リスト 11-4に示すように、括弧を移動することによって同じ計算をさらに効率的に実行できます。この例では、スカラー値を最初に乗じて、その結果をベクトルパラメータで乗じます。全体の演算は、5つの乗算で計算できます。

リスト 11-4　ベクトル演算の適切な使用

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

同様に、アプリケーションで、結果のコンポーネントすべてを使用するわけではない場合は、必ずベクトル演算に書き込みマスクを指定してください。スカラープロセッサ上では、マスクで指定されていないコンポーネントの計算はスキップできます。リスト 11-5は、2つのコンポーネントだけを必要と指定しているため、スカラープロセッサ上では2倍の速度で実行されます。

リスト 11-5　書き込みマスクの指定

```
highp vec4 v0;
highp vec4 v1;
highp vec4 v2;
```

```
v2.xz = v0 * v1;
```

シェーダ内での計算の代わりに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。動的な値の再計算は、シェーダでは非常に負荷が大きくなる可能性があります。

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。シェーダで分岐を使用しなければならない場合は、次の推奨事項に従ってください。

- ベストパフォーマンス：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。
- 許容可能：uniform変数に対して分岐する。
- 処理が遅くなる可能性あり：シェーダ内で計算された値に対して分岐する。

数多くのノブやレバーを持つ大きなシェーダを作成する代わりに、具体的なレンダリングタスクに特化した小さいシェーダを作成します。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。さまざまなオプションをテストし、最も高速なものを選びます。

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良いでしょう。

注： iOS 4.2以降では、シェーダコンパイラは、より積極的にループの最適化を行います。

シェーダにおける配列インデックスの計算を避ける

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。通常uniform配列へのアクセスは一時配列へのアクセスと比べてコストはかかりません。

動的なテクスチャ検索を避ける

動的なテクスチャ検索は、*従属テクスチャ読み込み*とも呼ばれ、フラグメントシェーダが、シェーダに渡された、修正されていないテクスチャ座標を使用するのではなく、テクスチャ座標を計算する場合に発生します。シェーダ言語はこの計算をサポートしていますが、従属テクスチャ読み込みではテクセルデータのロードが遅延して、パフォーマンスが低下するおそれがあります。シェーダが従属テクスチャ読み込みを行わない場合は、グラフィックスハードウェアがシェーダの実行前にテクセルデータを事前に取得し、メモリアクセスの待ち時間を感じさせない可能性があります。

リスト 11-6は、新しいテクスチャ座標を計算するフラグメントシェーダを示しています。この例の計算は、頂点シェーダでは簡単に実行できます。計算を頂点シェーダに移動し、頂点シェーダが計算したテクスチャ座標を直接使用することで、アプリケーションは従属テクスチャ読み込みを避けることができます。

注： わかりにくいかもしれませんが、テクスチャ座標に関する計算はどれも、従属テクスチャ読み込みとしてカウントされます。たとえば、テクスチャ座標の複数のセットを1つのvaryingパラメータに包み、スウィズルコマンドを使用してテクスチャ座標を抽出しても、従属テクスチャ読み込みが発生します。

リスト 11-6　従属テクスチャ読み込み

```
varying vec2 vTexCoord;
uniform sampler textureSampler;

void main()
{
    vec2 modifiedTexCoord = vec2(1.0 - vTexCoord.x, 1.0 - vTexCoord.y);
    gl_FragColor = texture2D(textureSampler, modifiedTexCoord);
}
```

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向けにさまざまなOpenGL ES実装を提供しています。各実装は、テクスチャに許可される最大サイズから、サポートされるOpenGL ES拡張機能のリストに至るまで、その機能はさまざまです。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iOS 4.2時点のものです。将来のハードウェアまたはソフトウェアリリースにおいて変わる可能性があります。アプリケーションが今後のプラットフォームとの互換性を確保できるよう、「OpenGL ESの機能の判定」（27 ページ）に示す機能について必ず、実行時に明示的にテストを行ってください。

表 12-1　iOSハードウェアデバイスリスト

デバイスの互換性	グラフィックスプラットフォーム	OpenGL ES 2.0	OpenGL ES 1.1
iPod Touch	PowerVR MBX	×	○
iPod Touch（第2世代）	PowerVR MBX	×	○
iPod Touch（第3世代）	PowerVR SGX	○	○
iPod Touch（第4世代）	PowerVR SGX	○	○
iPhone	PowerVR MBX	×	○
Phone 3G	PowerVR MBX	×	○
iPhone 3GS	PowerVR SGX	○	○
iPhone 3GS（中国向け）	PowerVR SGX	○	○
iPhone 4	PowerVR SGX	○	○
iPad Wi-FI	PowerVR SGX	○	○
iPad Wi-Fi+3G	PowerVR SGX	○	○

PowerVR SGXプラットフォーム

PowerVR SGXは、iPhone 3GS、iPhone 4、iPod touch（第3世代）、およびiPadに搭載されているグラフィックスプロセッサで、OpenGL ES 2.0をサポートするよう設計されています。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**タイルベースの遅延レンダリング（Tile Based Deferred Rendering：TBDR）**と呼ばれる手法を用いています。レンダリングコマンドをハードウェアに送信するためOpenGL ES関数を呼び出すと、これらのコマンドは、一定量のコマンドが蓄積されるまでバッファに格納されます。これらのコマンドは、ハードウェアによって単一の演算としてレンダリングされます。画像をレンダリングするため、フレームバッファはタイルに分割され、コマンドはタイルごとに1度描画され、各タイルはタイル内で表示されるプリミティブだけをレンダリングします。遅延レンダラの主な利点は、非常に効率良く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がタイル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には、フレームのできるだけ多くの部分を不透過なコンテンツで描画するとともに、ブレンド、アルファテスト、およびGLSLシェーダでの`discard`命令の使用を最小限に抑えます。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プリミティブを並べ替える必要がありません。

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、テクスチャを使用するひとまとまりの三角形をレンダリングし、その後そのテクスチャを修正する場合、OpenGL ESの実装ではそれらのコマンドをただちにフラッシュするか、テクスチャを複製するかのどちらかを実行しなければなりません。どちらの選択肢もハードウェアは効率的に使用されません。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のリリースノートおよび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 2.0とOpenGL ES 1.1の両方のアプリケーションに当てはまります。

- （以前送信された描画コマンドのため）レンダラがすでに使用しているOpenGL ESオブジェクトを変更する操作を避けます。OpenGL ESリソースを修正する必要のある場合は、それらの修正をフレームの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として、`glBufferSubData`、`glBufferData`、`glMapBuffer`,`glTexSubImage`、`glCopyTexImage`、`glCopyTexSubImage`、`glReadPixels`、`glBindFramebuffer`、`glFlush`、`glFinish`があります。

- プロセッサの非表示サーフェス削除機能を利用するには、「正しいレンダリングに必要な場合を除きレンダリングされたオブジェクトを並べ替えない」（59 ページ）に示す描画ガイドラインに従ってください。

- 頂点バッファオブジェクト(Vertex Buffer Object：VBO)を利用するとPowerVR SGX上でのパフォーマンスが大幅に向上します。詳細については、「頂点バッファオブジェクトの使用による頂点データのコピーの管理」（72 ページ）を参照してください。

- iOS 4.0以降では、独立したステンシルバッファはサポートされていません。結合された深度／ステンシルバッファを使用します。

- iOS 4.0以降では、`glTexImage2D`および`glTexSubImage2D`のパフォーマンスが大幅に向上しています。
- iOS 4.2以降では、レンダバッファにおけるCore Animationの回転のパフォーマンスが大幅に向上しており、コンテンツを回転させて横長モードと縦長モードを切り替える方法として推奨されます。最高のパフォーマンスを得るために、レンダバッファの高さと幅がそれぞれ32ピクセルの倍数となるようにします。

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個まで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を使用できません。
- 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、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。

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。

- APPLE_framebuffer_multisample
- APPLE_rgb_422
- APPLE_texture_format_BGRA8888
- APPLE_texture_max_level
- EXT_blend_minmax
- EXT_discard_framebuffer
- EXT_read_format_bgra
- EXT_shader_texture_lod
- EXT_texture_filter_anisotropic

- IMG_read_format
- IMG_texture_compression_pvrtc
- OES_depth24
- OES_depth_texture
- OES_fbo_render_mipmap
- OES_mapbuffer
- OES_packed_depth_stencil
- OES_rgb8_rgba8
- OES_standard_derivatives
- OES_texture_half_float
- OES_texture_float
- OES_vertex_array_object

iOS 3.1以降では、キューブマップテクスチャへのレンダリングもサポートしています。それより前のバージョンのiOSは、キューブマップについてFRAMEBUFFER_UNSUPPORTEDを返します。

既知の制限と問題

- 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

iOS 4.2以降では、シェーダコンパイラはより積極的にループを展開します。

PowerVR SGXは、たとえベクトルとして宣言されている値でも、スカラー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、「ベクトル計算の時間のかかる実行」（85 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。アプリケーションが低精度の浮動小数点値をパラメータとして提供したり、低精度の浮動小数点値に結果を割り当てたりする場合、シェーダが、値を変換する追加の命令を含めなければならないことがあります。これらの追加命令は、計算のベクトルの結果をスウィズルする場合にも追加されます。

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、カスタマイズされたシェーダを使用してPowerVR SGXハードウェアに実装されます。アプリケーションがOpenGL ESの状態変数を変更するごとに、必要に応じて新たなシェーダが暗黙的に生成されます。このため、OpenGL ESの状態を変更すると、純粋なハードウェア実装の場合よりもコストが高くなる可能性があります。アプリケーションのパフォーマンスは、アプリケーションが実行する状態の変更の回数を減らすことで向上させることができます。詳細については、「OpenGL ESの状態変数に注意を払う」（54 ページ）を参照してください。

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

- APPLE_framebuffer_multisample
- APPLE_texture_2D_limited_npot
- APPLE_texture_format_BGRA8888
- APPLE_texture_max_level
- EXT_blend_minmax
- EXT_discard_framebuffer
- EXT_read_format_bgra
- EXT_texture_filter_anisotropic
- EXT_texture_lod_bias
- IMG_read_format
- IMG_texture_compression_pvrtc

- OES_blend_equation_separate
- OES_blend_func_separate
- OES_blend_subtract
- OES_compressed_paletted_texture
- OES_depth24
- OES_draw_texture
- OES_fbo_render_mipmap
- OES_framebuffer_object
- OES_mapbuffer
- OES_matrix_palette
- OES_packed_depth_stencil
- OES_point_size_array
- OES_point_sprite
- OES_read_format
- OES_rgb8_rgba8
- OES_stencil8
- OES_stencil_wrap
- OES_texture_mirrored_repeat
- OES_vertex_array_object

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。PowerVR MBXは、OpenGL ES 2.0と同様、カスタムフラグメントシェーダをサポートしていませんが、グラフィックスハードウェアは非表示のフラグメントに関するカラー値の計算を回避します。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)」（90 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のほう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のリリースノートおよび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`.
- iOS 4以降では、`glTexImage2D`および`glTexSubImage2D`のパフォーマンスは大幅に向上しています。
- 頂点配列オブジェクトと静的頂点バッファオブジェクトをまとめ、頂点データをOpenGL ESに送信することで、ドライバはさらなる最適化を実行することができ、グラフィックスハードウェアにおけるデータ取得のオーバーヘッドを減らすことができます。

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

iOS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。

- APPLE_framebuffer_multisample
- APPLE_texture_format_BGRA8888
- APPLE_texture_max_level
- EXT_discard_framebuffer

- EXT_texture_filter_anisotropic
- EXT_texture_lod_bias
- IMG_read_format
- IMG_texture_compression_pvrtc
- OES_blend_subtract
- OES_compressed_paletted_texture
- OES_depth24
- OES_draw_texture
- OES_framebuffer_object
- OES_mapbuffer
- OES_matrix_palette
- OES_point_size_array
- OES_point_sprite
- OES_read_format
- OES_rgb8_rgba8
- OES_texture_mirrored_repeat
- OES_vertex_array_object

既知の制限と問題

OpenGL ES 1.1におけるPowerVR MBXの実装は、OpenGL ES 1.1仕様に完全には準拠していません。互換性の理由から、アプリケーションでは次のような特殊ケースを避けるようにしてください。

- （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。
    - サポート：

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

    - サポート：

      ```
      GL_TEXTURE_MAG_FILTER = GL_NEAREST,
      GL_TEXTURE_MIN_FILTER = GL_NEAREST_MIPMAP_LINEAR
      ```

    - 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。
  - 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。

iOSシミュレータ

iOSシミュレータは、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。シミュレータは、次のいくつかの点がPowerVR MBXやPowerVR SGXとは異なります。

- シミュレータはタイルベースの遅延レンダラを使用しません。
- シミュレータは、PowerVR MBXのメモリ制限を課しません。
- シミュレータは、PowerVR MBXやPowerVR SGXと同じ拡張機能をサポートしません。
- シミュレータは、PowerVR MBXやPowerVR SGXについて、ピクセル単位での精度を確保していません。

重要： シミュレータにおけるOpenGL ESのレンダリングパフォーマンスは、実際のデバイス上におけるOpenGL ESのパフォーマンスとまったく関係ありません。シミュレータは、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コードは、実際のデバイスよりもiOSシミュレータで実行するほうが高速な場合も低速な場合もあり、実行するコンピュータや描画内容によって異なります。必ず、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してください。シミュレータが実際のパフォーマンスを反映していると想定してはなりません。

シミュレータ上OpenGL ES 2.0

サポートされる拡張機能

シミュレータはOpenGL ES 2.0の次の拡張機能をサポートしています。

- APPLE_framebuffer_multisample
- APPLE_rgb_422
- APPLE_texture_format_BGRA8888
- APPLE_texture_max_level
- EXT_blend_minmax
- EXT_discard_framebuffer
- EXT_shader_texture_lod
- EXT_texture_filter_anisotropic
- IMG_read_format
- IMG_texture_compression_pvrtc
- OES_depth24
- OES_depth_texture
- OES_fbo_render_mipmap
- OES_mapbuffer
- OES_rgb8_rgba8
- OES_texture_half_float
- OES_texture_float
- OES_vertex_array_object

シミュレータ上OpenGL ES 1.1

サポートされる拡張機能

- APPLE_framebuffer_multisample
- APPLE_texture_2D_limited_npot
- APPLE_texture_format_BGRA8888
- APPLE_texture_max_level
- EXT_blend_minmax
- EXT_discard_framebuffer
- EXT_texture_filter_anisotropic
- EXT_texture_lod_bias
- EXT_texture_filter_anisotropic
- IMG_read_format
- IMG_texture_compression_pvrtc
- OES_blend_equation_separate
- OES_blend_func_separate
- OES_blend_subtract
- OES_compressed_paletted_texture
- OES_depth24
- OES_draw_texture
- OES_fbo_render_mipmap
- OES_framebuffer_object
- OES_mapbuffer
- OES_matrix_palette
- OES_point_size_array
- OES_point_sprite
- OES_read_format
- OES_rgb8_rgba8
- OES_stencil8

- OES_stencil_wrap
- OES_texture_mirrored_repeat
- OES_vertex_array_object

texturetoolを使用したテクスチャの圧縮

iOS 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。iOS 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、iOS 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iOS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オプションを指定する場合は-e オプションも指定する必要があります。また、-oオプションも必要です。

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
 --bits-per-pixel-4

Formats:

Raw
PVR
```

何もオプションが指定されない場合、texturetoolはデフォルトの--bits-per-pixel-4、--channel-weighting-linear、および-f Rawで実行されます。

--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。これらのレベルは、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。

iOS 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。iOS 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iOS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、iOS 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ImageIO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。

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iOSにロードしようとすると、エラーが返されます。

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4bpp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4bpp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2bpp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2bpp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4bpp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4bpp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2bpp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2bpp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)
```

```
{
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);
}
```

サンプルコードについては、「*PVRTextureLoader*」を参照してください。

書類の改訂履歴

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

日付	メモ
2010-11-15	文書内のすべての情報を大幅に改訂、拡充しました。
	よく使用されるグラフィックス関連およびOpenGL ES関連用語の解説を追加しました。
	iOS 4で追加された機能強化（レンダバッファの破棄）も含め、レンダリングループの詳細な説明を追加しました。
2010-09-01	間違ったリンクを修正しました。いくつかのパフォーマンスガイドラインを明確に説明しました。iOS 4で追加されたより新しい拡張機能へのリンクを追加しました。
2010-07-09	『iPhone OS OpenGL ESプログラミングガイド』から文書名を変更しました。
2010-06-14	iOS 4で公開された新しい拡張機能を追加しました。
2010-01-20	画面への描画を行うフレームバッファオブジェクトを作成するためのコードを修正しました。
2009-11-17	細かい改訂と編集を行いました。
2009-09-02	わかりやすくするために表現を編集しました。拡張機能リストを、現在利用可能な機能を反映するように更新しました。頂点処理能力を最高にする三角形ストリップの使いかたを明確にしました。プラットフォームの章に、PowerVR SGX上でのテクスチャの処理能力に関する注を追記しました。
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。

書類の改訂履歴

用語解説

この用語解説では、AppleによるOpenGL ESの実装で特に使用されている用語と、OpenGL ESグラフィックスプログラミングにおける一般的な用語を取り上げています。

エイリアス化(aliased) グラフィックスのエッジがギザギザになっている様子を指す。アンチエイリアス化の操作によって改善可能。

アンチエイリアス化(anti-aliasing) グラフィックスにおいて、テキストや線画、画像などのグラフィカルオブジェクトを描画したときに現れることのある、ギザギザになった（エイリアス化した）エッジを滑らかにしたり、ぼやかしたりするテクニック。

アタッチ(attach) 2つの既存のオブジェクトを結び付けること。バインドも参照のこと。

バインド(bind) 新しいオブジェクトを作成し、そのオブジェクトとレンダリングコンテキストを結び付けること。アタッチも参照のこと。

ビットマップ(bitmap) ビットで構成された矩形状の配列。

バッファ(buffer) 頂点属性や色データ、インデックスなど、特定の種類のデータを格納するためだけに確保されたメモリのブロック。OpenGL ESが管理。

クリッピング(clipping) 描画する領域を特定する操作。クリッピング領域にないものは描画されません。

クリップ座標(clip coordinates) ビューボリュームのクリッピングに使用される座標系。クリップ座標は、投影行列を適用した後、透視除算よりも前に適用されます。

完全性(completeness) フレームバッファオブジェクトが描画のためのすべての要件を満たしているかどうかを表している状態。

コンテキスト(context) 一連のOpenGL ES状態変数。コンテキストにアタッチされている描画可能なオブジェクトに対して、どのように描画を行うかに影響します。*レンダリングコンテキスト*とも呼びます。

間引き(culling) オブザーバが見ることのできない、シーンの一部分を取り除くこと。

現在のコンテキスト(current context) アプリケーションが発行したコマンドをOpenGL ESが送る先のレンダリングコンテキスト。

現在の行列(current matrix) ある座標系から別の座標系へと座標を変換するためにOpenGL ES 1.1が使用する行列。モデルビュー行列、透視行列、テクスチャ行列などがあります。GLSL ESでは、ユーザ定義の行列を使用できます。

深度(depth) OpenGLにおいて、*z*座標のことを指し、オブザーバからピクセルがどれくらい離れているかを指定。

深度バッファ(depth buffer) 各ピクセルの深度値を格納するため使用する、メモリのブロック。深度バッファは、オブザーバからピクセル見えるかどうかを決めるために使用します。OpenGL ESがラスタ化したフラグメントはすべて、深度バッファに格納されている値と受信した深度値とを比較する深度テストを通過しなければなりません。

ダブルバッファリング(double buffering) 2つのバッファを使用して、グラフィックサブシステムの異なる2つの部分の間でリソースが競合するのを避けること。フロントバッファを一方の当事者が使用している間に、もう一方の当事者が

バックバッファを修正します。スワップが発生すると、前面バッファと背面バッファが入れ替わります。

描画可能オブジェクト(drawable object) OpenGL ESの外部で割り当てられたオブジェクト。OpenGL ESフレームバッファオブジェクトの一部として使用できます。iOSでは、描画可能オブジェクトは`EAGLDrawable`プロトコルを実装します。現在、`CAEAGLLayer`オブジェクトだけが描画可能オブジェクトとして使用することができ、OpenGL ESのレンダリングをCore Animationに統合するために使用します。

拡張機能(extension) OpenGL ESのコアAPIの一部ではなく、そのためOpenGL ESのすべての実装でのサポートが保証されていない、OpenGL ESの機能。拡張機能に使用される命名規則は、その拡張機能がどれくらい幅広く受け入れられているかを示しています。特定の会社だけがサポートする拡張機能の名前には、会社名の省略形が含まれます。複数の会社が拡張機能を採用している場合、その拡張機能の名前は会社名の省略形の代わりに`EXT`が含まれるよう変更されます。Khronos OpenGL Working Groupが拡張機能を承認している場合、拡張機能の名前は、`EXT`または会社名の省略形の代わりに、`OES`が含まれるよう変更されます。

眼点座標(eye coordinates) 原点のオブザーバが使用する座標系。眼点座標はモデルビュー行列によって生成され、投影行列に渡されます。

フィルタ(filtering) ピクセルまたはテクセルを組み合わせることで画像を修正するプロセス。

フォグ(fog) オブザーバからの距離に基づき、背景色へと色をフェードすることによって達成される効果。フォグは、オブザーバに深度の手がかりを与えます。

フラグメント(fragment) プリミティブをラスタ化するときに計算された色と深度の値。各フラグメントは、フレームバッファに格納されているピクセルとブレンドする前に一連のテストを通過しなければなりません。

システムフレームバッファ(system framebuffer) グラフィックスサブシステム（ウインドウシステムなど）へのOpenGL ESの統合をサポートするオペレーティングシステムが提供するフレームバッファ。iOSはシステムフレームバッファを使用しません。その代わり、iOSはレンダバッファがコンテンツをCore Animationと共有できるようにする、追加の関数を提供します。

フレームバッファにアタッチ可能な画像(framebuffer attachable image) フレームバッファオブジェクトのレンダリング先。

フレームバッファオブジェクト(framebuffer object) OpenGL ES 2.0の中核的機能。この機能により、アプリケーションはOpenGL ESによって管理されるフレームバッファのレンダリング先を作成できます。フレームバッファオブジェクト(framebuffer ojbect：FBO)は、OpenGL ESのフレームバッファに関する状態情報、およびレンダバッファと呼ばれる画像のセットを格納します。iOSにおけるOpenGL ES 1.1実装のすべてが、`OES_framebuffer_object`拡張機能を通じてフレームバッファオブジェクトをサポートします。

フラスタム(frustum) オブザーバから見える、透視除算によって歪んだ空間領域。

画像(image) ピクセルで構成された矩形状の配列。

インターリーブされたデータ(interleaved data) 頂点データやテクスチャ座標など、異なるデータで構成され、グループとしてまとめられた配列。インターリーブにより、データの取得を高速化できます。

ミップマップ(mipmaps) さまざまな解像度で提供される、テクスチャマップのセット。オンスクリーンの解像度がソースのテクスチャマップに適合していないジオメトリのプリミティブに対して、テクスチャが適用されるときに発生する可能性のあるアーティファクトを最小限に抑えることが目的です。ミップマップとはラテン語の*multum in parvo*に由来する言葉で、「狭い場所に多くのものがある」ことを意味します。

モデルビュー行列(modelview matrix) 点、線、多角形の変換、およびオブジェクトの座標から眼点座標への位置の変換のためにOpenGLが使用する、4×4の行列。

マルチサンプル機能(multisampling) ピクセルを対象に複数のサンプルを取得し、サンプルをカバレッジ値と組み合わせて最終フラグメントに到達するためのテクニック。

ミューテックス（相互排他、mutex） マルチスレッドアプリケーションにおける相互排他オブジェクト。

パッキング(packing) ピクセルの色コンポーネントを、バッファからアプリケーションが要求するフォーマットに変換すること。

ピクセル(pixel) ピクチャの要素。グラフィックスハードウェアが画面上に表示できる最も小さい要素です。ピクセルは、フレームバッファ内のすべてのビットプレーンにおける、「*x*, *y*」の位置にあるすべてのビットで構成されています。

ピクセル深度(pixel depth) ピクセルの画像における、1ピクセルあたりのビット数。

ピクセルフォーマット(pixel format) ピクセルデータをメモリに格納するために使用するフォーマット。フォーマットは、ピクセルのコンポーネント（つまり、赤、青、緑、アルファ）、コンポーネントの数と順序、およびその他の関連情報（ピクセルにステンシル値や深度値が含まれるかどうかなど）を記述するものです。

事前乗算済みのアルファ(premultiplied alpha) ほかのコンポーネントがすでにアルファ値で乗じてあるピクセル。たとえば、乗算済みでないRGBA値が(1.0, 0.5, 0.0, 0.5)のピクセルの場合、乗算済みの値は(0.5, 0.25, 0.0, 0.5)となります。

プリミティブ(primitives) OpenGLで最も単純な要素。点、線、多角形、ビットマップ、画像のこと。

投影行列(projection matrix) 点、線、多角形、および眼点座標からクリップ座標への位置を変換するためにOpenGLが使用する行列。

ラスタ化(rasterization) 頂点データおよびピクセルデータをフラグメントに変換するプロセス。各フラグメントは、フレームバッファ内のピクセルに対応します。

レンダバッファ(renderbuffer) 2Dピクセル画像のレンダリング先。`OES_framebuffer_object`拡張機能についてOpenGL仕様で定義されているように、汎用化されたオフスクリーンレンダリングについて使用されます。

レンダラ(renderer) ビューおよびモデルから画像を作成するためにOpenGL ESが使用するハードウェアとソフトウェアの組み合わせ。OpenGL ESの実装では、グラフィックスハードウェアの機能が要件を上回っている場合、ソフトウェアのフォールバックを提供する必要はありません。

レンダリングコンテキスト(rendering context) 状態情報のコンテナ。

レンダリングパイプライン(rendering pipeline) OpenGL ESがピクセルデータと頂点データをフレームバッファの画像に変換するために使用する操作の順序。

render-to-texture テクスチャターゲットに直接コンテンツを描画する操作。

RGBA 赤(Red)、緑(green)、青(blue)、アルファ(alpha)の色コンポーネント。

シェーダ(shader) サーフェスプロパティを計算するプログラム。

シェーディング言語(shading language) Cにおいてアクセス可能な高水準言語。高度な画像効果を生み出すために使用します。

ステンシルバッファ(stencil buffer) ステンシルテスト専用のメモリ。ステンシルテストは通常、マスク領域の識別、覆う必要のある無地のジオメトリの識別、および半透明の多角形のオーバーラップに使用します。

ティアリング(tearing) 現在のフレームがスクリーン上に完全にレンダリングされる前に、現在のフレームの一部がフレームバッファ内の前のフレームのデータを上書きしてしまう場合に発生する、視覚的異常。iOSでは、表示可能なOpenGL ESコンテンツすべてをCore Animationを通じて処理することにより、ティアリングを回避します。

テッセレーション(tessellation) サーフェスを多角形のメッシュに変形したり、曲線を連続する直線に変形したりする操作。

テクセル(texel) フラグメントに適用する色を指定するために使用する、テクスチャの要素。

テクスチャ(texture) ラスタ化されたフラグメントの色を修正するために使用する画像データ。データの種類は、1次元、2次元、3次元、またはキューブマップ。

テクスチャマッピング(texture mapping) プリミティブにテクスチャを適用するプロセス。

テクスチャ行列(texture matrix) テクスチャ座標を、補間処理やテクスチャ検索に使用する座標に変換するためにOpenGL ES 1.1が使用する、4×4の行列。

テクスチャオブジェクト(texture object) テクスチャに関連するすべてのデータを格納するために使用する、不透過のデータ構造体。テクスチャオブジェクトには、画像やミップマップ、テクスチャパラメータ（幅、高さ、内部フォーマット、解像度、ラップモードなど）といったものを含めることができます。

頂点(vertex) 三次元の先端。頂点のセットは、形状のジオメトリを指定します。頂点は、色やテクスチャ座標など、多数の追加属性を持つことができます。詳細については、頂点配列を参照。

頂点配列(vertex array) 頂点座標、テクスチャ座標、サーフェス法線、RGBA色、色インデックス、エッジフラグなどを指定する、データのブロックを格納するデータ構造体。

頂点配列オブジェクト(vertex array object) アクティブな頂点属性のリスト、各属性が格納されているフォーマット、および頂点データの読み出し場所を記録するOpenGL ESオブジェクト。頂点配列オブジェクトは、設定を初期化してすばやく復元できるようにすることで、グラフィックスパイプラインを設定する手間を軽減します。