平成 14 年度　修士論文

携帯型ゲーム機の描画機能を活用した 3 次元描画ライブラリの開発

電気通信大学　大学院情報システム学研究科
情報システム設計学専攻
0150009　小関　一正
指導教官　多田　好克　助教授
星　　守　　教授
寺島　元章　助教授

提出日　平成 15 年 2 月 3 日

目次

図目次

表目次

第1章　はじめに

本研究ではGAME BOY ADVANCE（以下GBA）上で動作するOpenGL準拠の3次元グラフィックスライブラリを開発する。

本研究の目的は、携帯端末上で3次元グラフィックスを使用する環境を構築することで、3次元グラフィックスの用途の幅を広げることである。近年3次元グラフィックスは様々な分野に用いられており、屋外における用途も存在すると考える。本研究では 3次元グラフィックスを屋外で用いるための端末を GBA とした。携帯型ゲーム機であるGBAは、PDA（Personal Digital Assistant）など他の携帯端末と同程度の性能でありながら、安価で幅広い層に普及している。それに加えて、GBAは2次元グラフィックス向けのハードウェア機能をサポートしている。それを3次元グラフィックスの処理に用いることで、3次元グラフィックスを使用したアプリケーションを高速化することができる。また、2次元グラフィックス用のハードウェア機能を使用した、廉価な3次元グラフィックスハードウェア開発の可能性を広げることができる。

GBA上で動作させる3次元グラフィックスライブラリは、OpenGL [1] に準拠するものとする。OpenGL は3次元グラフィックスライブラリの規格であり、3次元グラフィックスを利用したアプリケーションの構築に必要な関数群のインターフェイスを提供する。3次元グラフィックスライブラリの作成者はOpenGLに準拠することでライブラリの利用者に一様なアプリケーションプログラミングインターフェイス（以下API）を与える。また、利用者はOpenGLが提供するAPIを学習するだけで、OpenGLに準拠した3次元グラフィックスライブラリを容易に利用することができる。

GBAをPCと比較した場合、いくつかの問題点を挙げることができる。それは、解像度が低いこと、処理能力が低いこと、RAM容量が少ないことである。GBA上で3次元グラフィックスライブラリを実装するにあたって、これらの問題点を克服しなければならない。

本論文は次のような構成になっている。まず、第2章で本研究の背景と目的について述べる。ここでは 3 次元グラフィックスを携帯端末で動作させる意義と、その環境を

GBA とした理由を述べる。第 3 章では 3 次元グラフィックスについて説明した後に、関連研究として OpenGL の仕様と OpenGL 準拠の 3 次元グラフィックスライブラリについて説明する。第 4 章で GBA の仕様について述べ、第 5 章では GBA 上で動作させる OpenGL 準拠の 3 次元グラフィックスライブラリの設計について述べる。第 6 章では設計を基に行なった実装について述べる。第 7 章で実際に動作させた 3 次元グラフィックスの評価を行い、第 8 章では、高速化手法について述べる。最後に第 9 章でまとめとする。

OpenGL 処理パイプラインの詳細についての説明を付録 A として添付する。本文中の用語には、そこで説明しているものがある。

第2章　背景と目的

この章では、はじめに3次元グラフィックスライブラリについて述べる。次にそれらと携帯型ゲーム機との関連について述べる。そして携帯型ゲーム機のひとつであるGBAの特徴を簡単に示した後、本研究の目的を述べる。

2.1　3次元グラフィックスライブラリ

近年の計算機は劇的な進化を遂げており、計算機内で擬似3次元空間が容易に構築できるようになった。3 次元グラフィックスはゲームに使われるのはもちろんのこと、建築、地形のシミュレーション、化学、物理学、医学の物理現象や分子構造の視覚化など様々な分野で用いられている。

3次元グラフィックスを利用するアプリケーションを開発する場合、一般に3次元グラフィックスライブラリを使用する。Direct3D [2] 、QuickDraw 3D [3] などの3次元グラフィックスライブラリは、3 次元グラフィックスを利用したアプリケーションの構築に必要な関数群のインターフェイスを提供する。OpenGL [1] はシリコングラフィックス社らが提唱する3次元グラフィックスライブラリ規格であり、3次元グラフィックスライブラリの作成者は、OpenGL に準拠することでライブラリの利用者に画一の API を与える。また、利用者はOpenGLが提供するAPIを学習するだけで、OpenGLに準拠した3次元グラフィックスライブラリを容易に利用することができる。

2.2　携帯型ゲーム機と3次元グラフィックス

3次元グラフィックスの需要の高まりから、屋外で用いられる計算機上での3次元グラフィックスの利用が求められている。屋外で用いる計算機には、ノートPCやPDAなどのほかに携帯型ゲーム機がある。携帯型ゲーム機は携帯性を重視した設計がなされているためにプロセッサやメモリ、出力画面などが貧弱であるが、ノート PC などに比べて安価であり幅広い層に普及している。また、一般の携帯型ゲーム機は貧弱な性能を補うためにゲーム内のアニメーションに有用な機能をハードウェアでサポートしており、

これらは PDA に比べて強力である。

携帯型ゲーム機上で動作するゲームプログラムは、頻繁に再描画される画像をグラフィックスオブジェクト（またはスプライト）という単位で管理する。グラフィックスオブジェクトはハードウェアでサポートされた機能によって、回転、拡大、縮小などの処理が高速に行なわれる。

一般に、携帯型ゲーム機上で動作するプログラムはリアルタイムな処理が要求され、したがって速度を追求した実装がなされる。また、ハードウェアの制約からプログラムの省スペース化、リソースの節約が求められる。そのため汎用のライブラリを使用することは少なく、ライブラリがあったとしてもそれはアプリケーションに特化した、専用のものである。携帯型ゲーム機で動作するゲームにおいて 3 次元グラフィックスを用いたい場合でも、開発者は汎用の 3 次元グラフィックスライブラリを使用していないのが一般的である。

2.3 3 次元グラフィックスと GBA

携帯型ゲーム機のひとつである GBA は、一般的な携帯型ゲーム機の機能であるグラフィックスオブジェクトの操作が可能である他、アルファブレンディング（半透明処理）、フェードイン／アウト（輝度変化処理）などのハードウェア機能も有している。また、他の携帯型ゲーム機に比べて高性能の CPU を搭載していること、ダブルバッファリング機能を備えていることなどから、GBA は比較的 3 次元描画に適している携帯型ゲーム機であると考える。

2.4 研究の目的

本研究では、GBA 上で動作する、OpenGL に準拠した 3 次元グラフィックスライブラリを開発する。これによって、次世代の携帯型ゲーム機として世界で最も普及している GBA をゲームに限らない様々な用途に用いることができる。3 次元グラフィックスを OpenGL に準拠させることで、OpenGL の利用者は容易に GBA 用のアプリケーションを

作成できる。また、既存の OpenGL を使用したソフトウェアの移植も容易になる。

開発時には次の 2 点に気を配らなければならない。

1 つは、OpenGL が巨大なことである。OpenGL は汎用性の高いライブラリ規格であるが、汎用性を追及しているがゆえに大規模なライブラリになっており、そのために動作も遅い。これをそのまま GBA に実装することは、GBA のハードウェア制限の面から思わしくないと考える。そこで、実装する API を最低限必要なものだけに留め、省スペース化、高速化を図る必要がある。

もう 1 つは、GBA は 2 次元グラフィックスのためのハードウェア機能のみを有しているが、それは 3 次元グラフィックスへの使用を想定していないということである。2 次元グラフィックスのための機能をどのように 3 次元グラフィックスに対して適用するかを吟味しなければならない。しかし、2 次元グラフィックスのためのハードウェア機能を 3 次元グラフィックスに適用できれば、2 次元グラフィックス向けハードウェア機能を用いた廉価な 3 次元グラフィックスハードウェア開発の可能性を見出せる。

第3章　OpenGL

この章では、まず3次元グラフィックスについて簡単に説明する。そして3次元グラフィックスライブラリ規格の1つであるOpenGLを、アプリケーションが使用するライブラリの観点から紹介する。次に、X Window Systemにおいて、どのようにアプリケーションがOpenGLを利用するかを図解する。最後に関連研究として、OpenGLの規格に基づいて開発されたライブラリを紹介する。

3.1　3次元グラフィックス

3次元グラフィックスとは、2次元画像を見ているのにも関わらず、あたかも3次元空間を見ているかのように見せる技術である。具体的には、擬似3次元空間（シーン）を生成し、それを2次元である画像（イメージ）に変換する。シーンは3次元であるから、縦、横、奥行きの3つのベクトルを持つ。

シーンの中に存在する物体（オブジェクト）は、点、線分、三角形（ポリゴン）といった単純な図形（幾何学プリミティブもしくはプリミティブ）の集合として描かれる。例えば直方体を描くとき、1つの面である長方形は2つの三角形から構成される。そして、直方体の6つの面はその長方形から構成される。プリミティブは頂点の集合であり、頂点は頂点座標によって定義される。また、頂点はカラー情報を保持できる。

イメージはシーン内のある場所（視点もしくはビューポイント）から観た風景として出力される。視点は設定により自由に変化させることができる。そのため、一度生成したシーンは任意の角度、スケールでイメージに変換することができる。

シーンは様々な処理によって現実的な外観を持つことができる。テクスチャマッピングはテクスチャと呼ばれる2次元画像をプリミティブに貼り付ける処理である。照光処理は光源と影付けアルゴリズムを指定することで、オブジェクトに陰影をつける。深度テストはオブジェクト間の重なりを表現する。

シーンの中にオブジェクトを生成することをモデリングという。シーンを2次元画像に変換することをレンダリング（数値化の意）という。

3.2 OpenGL と関連ライブラリ

OpenGL を使用するアプリケーションを開発する際に関連するライブラリを以下に挙げる。

3.2.1 OpenGL

OpenGL は、ウィンドウシステム、グラフィックスハードウェアに依存しない高性能なソフトウェアインターフェイスを提供する規格、もしくはその規格に基づいて開発された 3 次元グラフィックスライブラリである。それは約 150 種類のコマンドから構成されており、オブジェクトやインタラクティブな 3 次元アプリケーションの作成に必要な処理の指定に使用される。アプリケーションは OpenGL により提供されている API を使用し、対応したコマンドを OpenGL 処理パイプラインに送ることで目的の処理を行なう。

OpenGL はハードウェア、プラットフォームに依存しないインターフェイスとして設計されている。そのため OpenGL にはウィンドウシステムにおけるウィンドウ管理や、キーボードやマウスからのユーザ入力取得のためのコマンドが含まれていない。また、複雑な 3 次元オブジェクトモデルを記述するような高水準コマンドも含まれていない。具体的には、点、線、ポリゴンなどの幾何学プリミティブと、アフィン変換や照光計算のようなレンダリング命令、そしてテクスチャマッピングやアンチエイリアシングのようなレンダリング機能についてのコマンドが含まれる。

OpenGL は IRIS GL [4] の後継としてシリコングラフィックス社が開発し、OpenGL Architectural Review Board（ARB）がそれを管理している。OpenGL の理念は、高速に動作する、グラフィックスプログラミング用の共通ソフトウェアインターフェイスを確立し、3 次元グラフィックスがいち早くコンピュータ技術の主流に乗れるようにすることである。この目標は既に達成されている。

3.2.2 GLU

OpenGL Utility Library（GLU） [5] は、低水準である OpenGL のコマンドを使用して複雑な機能を提供するライブラリである。GLU を用いることで、2 次曲面や NURBS（Non-Uniform Rational B-Spline）曲線および曲面などのモデリングが可能となる。GLU

は OpenGL のコマンドを用いて構築されているため、OpenGL が使用可能な環境であれば GLU を使用することができる。

3.2.3 ウィンドウシステム機能拡張ライブラリ

OpenGL はプラットフォームに依存しない設計がなされているため、ウィンドウシステムに依存したコマンドが含まれていない。そのため、各ウィンドウシステムに対して、そのウィンドウシステムが OpenGL レンダリングをサポートするための機能を拡張するライブラリが存在する。X Window System において OpenGL を使用する場合、OpenGL Extension to the X Window System（GLX） [6] を用いる。同様に Microsoft Windows の場合には WGL、IBM OS／2 WARP の場合には PGL、Apple Macintosh の場合には AGL を用いる。

3.2.4 GLUT

OpenGL Utility Toolkit（GLUT） [7] はウィンドウシステムの依存性を排除するためのツールキットである。通常、アプリケーションの開発者は GLX、WGL のようなウィンドウシステム機能拡張ライブラリを、アプリケーションが動作するウィンドウシステムに合わせて選択し、使用しなければならない。GLUT はその差異を吸収するためのレイヤとして機能する。GLUT を用いることで、アプリケーションの開発者はウィンドウシステムを気にせずに 3 次元グラフィックスアプリケーションを開発できる。

3.3 OpenGL の実行モデル

OpenGL は特定のウィンドウシステムやオペレーティングシステムに依存しないように設計されている。しかし大抵のプログラムは、ある程度のユーザ入力やその他のウィンドウシステム、オペレーティングシステムからのサービスを必要とする。そのようなサービスを利用するために、アプリケーションは GLX のようなウィンドウシステム機能拡張ライブラリ、または GLUT を呼び出す。

X Window System 上で OpenGL を利用したアプリケーションを動作させる場合のモジ

ュール関係を図 3.1に示す。図中の矢印はモジュールの呼び出しを表している。はじめにアプリケーションはウィンドウの生成、初期化や X プロトコルの通信を行なうために GLX を用いる。この時、GLX の代わりにウィンドウ機能拡張ライブラリ間の差異を吸収するためのツールキットである GLUT を用いてもよい。次にアプリケーションは OpenGL 準拠のグラフィックスライブラリ（GL）または GLU を用いてシーンをモデリング、レンダリングし、その結果であるイメージはメモリ上に確保されたフレームバッファに格納される。このようにして生成されたイメージは、アプリケーションが再度 GLX もしくは GLUT を用いることでディスプレイに出力される。また、アプリケーションは GLX または GLUT を用いてキーボードやマウスからの入力を受け取り、それをレンダリング、モデリングやアプリケーションの動作に反映させることができる。

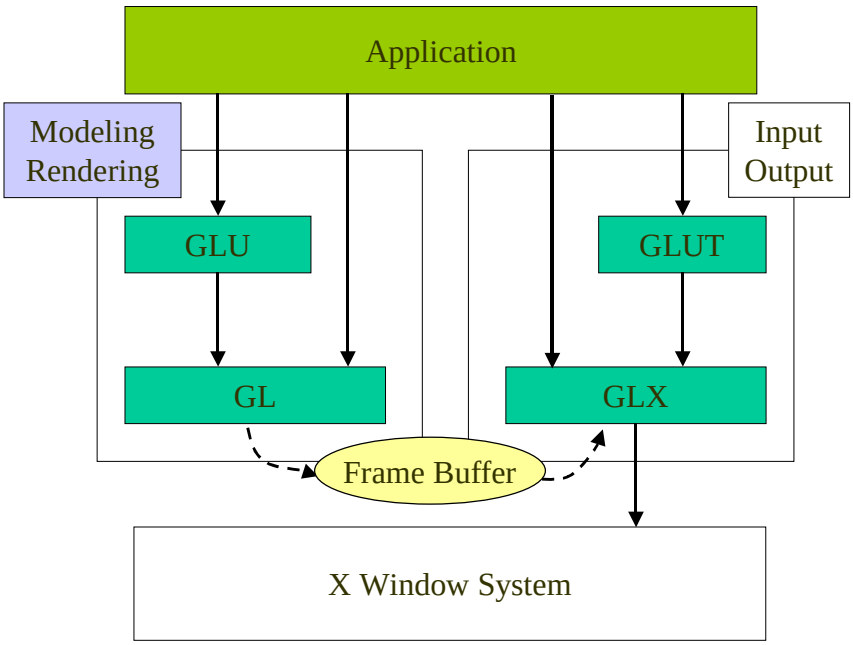

図 3.1　X Window System における OpenGL のモジュール関係

ここで示したように、GL にはウィンドウ管理、フレームバッファの形成やその初期化に関するコマンドは含まれていない。OpenGL コマンドの処理結果が反映されるフレームバッファはウィンドウシステムが管理する。アプリケーションはウィンドウシステムに依頼して、ウィンドウの生成、フレームバッファの形成、初期化を実行する。ウィンドウシステムは、GL がフレームバッファのどの部分にアクセスするかを決定し、それらがどのように構成されているかを GL に通知する。

OpenGL のコマンド解釈、処理モデルはクライアント／サーバ型である。クライアントであるアプリケーションが発行したコマンドは、サーバである GL によって解釈、処理される。その時、サーバはクライアントと同一の計算機で動作しても、別の計算機上で動作してもよい。

3.4 OpenGL の処理手順

図 3.2に OpenGL のおおまかなデータ処理過程を表したブロックダイアグラムを示す。この図では左からコマンドが入力され、パイプラインを流れるように順に処理される。コマンドには、オブジェクトを形成する幾何学プリミティブを指定するものや、多様な処理ステージにおけるオブジェクトの取り扱い方法を制御するものがある。

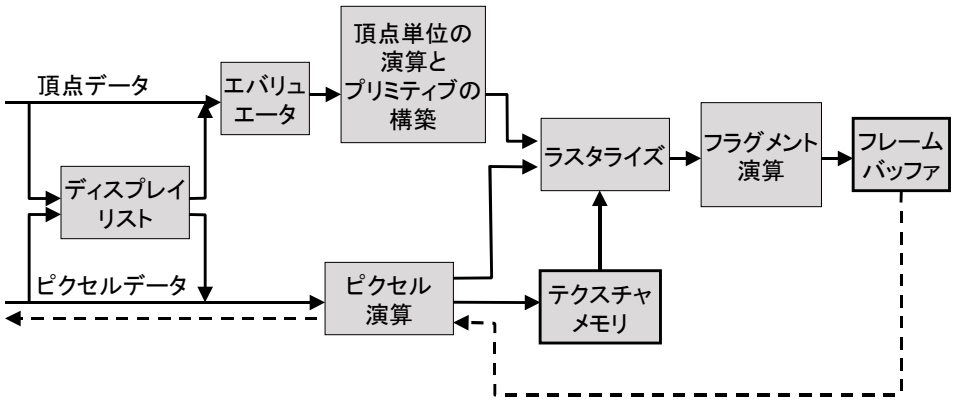

図 3.2　OpenGL ブロックダイアグラム

以下、OpenGL レンダリングパイプラインにおける主要なステージについて述べる。

- **ディスプレイリスト**

  頂点データ、ピクセルデータはともに、現在もしくは後の使用に備えてディスプレイリストに保存することができる。ディスプレイリストが実行されると、そこに保持されているデータはアプリケーションからデータを送るのと同様にパイプラインを流れる。ディスプレイリストを使わず、データが送られたときに即時に実行することも可能で、それを即時モード（immediate mode）という。

- **エバリュエータ**

  全ての幾何学プリミティブは、最終的には頂点によって記述される。エバリュエータは、NURBS などの、複雑な曲線、曲面を表現する多項式によるコマンドを評価し、近似的な頂点を算出する。

- **頂点単位の演算とプリミティブの構築**

  点、線分、ポリゴンといった全ての幾何学プリミティブは、このステージで頂点の集合として処理される。頂点は、変換、照光処理され、最終的にイメージに反映される幾何学プリミティブ以外をクリッピング処理で取り除く。

- **ピクセル演算**

  このステージで、ピクセルデータはまず適切なカラー、フォーマットに変換され、拡大縮小などの処理をされる。その後、頂点データと同様ラスタライズステージに送られるか、テクスチャメモリに送られる。OpenGL はテクスチャ画像をオブジェクトに適用することで、よりリアルな外観を作成できる。

- **ラスタライズ**

  ラスタライズステージでは、ラスタライザが頂点データ、ピクセルデータをフラグメント（fragment）に変換する。フラグメントとはフレームバッファ内のピクセルに対応した正方形である。頂点を接続して線分にする場合や、ポリゴンを埋める場合などは、頂点間もしくはポリゴンの内部のピクセルを算出する必要がある。その際、アンチエイリアスをサポートする線分やポリゴンの破線、線分の幅、点のサイズ、モデルの陰影処理、範囲計算が考慮される。カラーと深度値は各フラグメントに割り当てられる。

- **フラグメント演算**

  各ピクセルに対応したフラグメントは、フレームバッファに書き込まれる前に変更、廃棄される。まずテクスチャメモリから、各フラグメントに対応したテクセル（texture element）が生成され、各フラグメントに適用される。テクセルとはテクスチャをフラグメント化したものである。その後、アルファ値により半透明処理をするアルファテスト、陰面消去のための深度テストなど、アプリケーションで有効化された処理が行なわれる。そして完全に処理されたフラグメントはフレームバッファの適切な位置に書き込まれる。

OpenGL の処理パイプラインの詳細を、付録 A にて解説する。その内容と、その中で紹介される単語は以下の本論文中で使用される。

3.5 関連研究

本研究と関連する、OpenGL 規格を基に作られた 3 次元グラフィックスライブラリを以下に挙げる。

3.5.1 Mesa

Mesa [8] は 1995 年に Brian Paul が開発した、OpenGL 規格の無料インプリメンテーションである。Mesa は OpenGL 規格に準拠していた設計がなされているが、無料インプリメンテーションを実現するためと、OpenGL　ARB から規格に関するクレームを受けたくないために、OpenGL　ARB の OpenGL 適合検査を通さなかった。よって Mesa は公式な OpenGL 準拠ライブラリではない。しかし Mesa は X Window System で動作する OpenGL 準拠ライブラリのデファクトスタンダードとなっている。

Mesa は高品質な 3 次元グラフィックスシーンを生成することができるが、独自の拡張を加えるなどして多く機能をサポートしているため、ライブラリが肥大化してしまっている。

3.5.2 TinyGL

TinyGL [9] はバーチャルリアリティネットワークシステム VREng [10] の学生プロジェクトとして Fabrice Bellard によって 1997 年に開発された、OpenGL の非常に小さなサブセットである。TinyGL は OpenGL の中でよく使用される機能のみが実装されており、そのため TinyGL は OpenGL と完全に互換性があるわけではないが、その分軽量化されており高速に処理することができる。

3.5.3 miniGL

miniGL [11] は 2000 年に Digital Sandbox 社が開発した、Palm OS 上で動作する OpenGL 準拠の 3 次元グラフィックスライブラリである。Palm OS にはカラーレンダリング能力に制限がある。しかし、miniGL はジオメトリ変換とレンダリング機能をサポートする。

miniGL の目的は、小さく携帯性のある Palm OS というプラットフォームにおいて、インタラクティブなシミュレーションのための携帯性のある建築物モデルへの応用、ゲームなどのインタラクティブな 3 次元世界を実現することである。miniGL を OpenGL 準拠にすることによって、3 次元グラフィックスを使用したアプリケーションの Palm OS への移植にかかる時間が劇的に短縮される。

第4章　GBA

この章では、まずGBAの仕様を示し、ハードウェアの機能、メモリ構造といったGBAの特徴について述べる。これを、本研究でGBAを対象に選んだ理由とする。最後に本研究における開発環境を示す。

4.1　仕様

GBAは2001年3月に任天堂から発売された携帯型ゲーム機である。GBAはGAMEBOYの上位互換機として開発された。GAMEBOYは国内で約3,200万台、全世界で約1.2億台を出荷し、携帯型ゲーム機市場最大の出荷台数を誇る。そしてGBAは、2002年9月において国内で約700万台、全世界で約2,300万台を出荷しており、既に次世代携帯型ゲーム機市場において最も普及している。

GBAの仕様を表4.1に、2002年4月における他の主要な携帯端末との性能比較を表4.2に示す。

表 4.1　GBA の仕様

LCD	反射型 TFT カラー液晶
画面サイズ	40.8mm × 61.2mm （2.9 インチ）
解像度	240 × 160 ドット
最大表示色数	32,768 色
特殊効果機能	回転・拡大・縮小、アルファブレンディング フェードイン／フェードアウト、モザイク
CPU	32bit RISC-CPU （ARM7 DTMI） + 8bit CISC-CPU （Z80）
メモリ	32KB WRAM + 96KB VRAM （CPU 内蔵） 256KB WRAM （CPU 外部）
寸法	82mm × 144.5mm × 24.5mm
本体重量	140g （乾電池除く）
カートリッジ容量	最大 256Mbit
価格	8,800 円

表 4.2　主要な携帯端末の性能比較

	RAM	ROM	解像度	色数（bit）	稼働時間	単価

ZAURUS MI-E25DC	16MB	16MB	240×320	16	1.7～11 時間	6 万円
Palm m505	8MB	4MB	160×160	16	5.3 時間	5 万円
iPAQ H3870	64MB	32MB	240×320	16	14 時間	8 万円
GBA	384KB	32MB (外付)	240×160	15	15 時間	9 千円

解像度、色数などの描画性能において、GBA は他の携帯端末と比べて同等の性能である。また、CPU には 32 ビット RISC CPU を搭載しており、処理速度においても同程度である。しかし、価格は他の携帯端末の 5 分の 1 から 8 分の 1 と、安価になっている。GBA は稼働時間が長い上、他の携帯端末が主に内蔵充電バッテリーで動作するのに対して単三電池 2 本で動作するため、屋外で長時間使用することができる。また、GBA はゲーム内のアニメーションに有用な機能をハードウェアでサポートしており、PDA などの携帯端末にはそのような機能は搭載されていない。GBA は以下の機能をハードウェアサポートしている。

- **グラフィックスオブジェクト**

  頻繁に再描画される画像をグラフィックスオブジェクト（またはスプライト）という単位で管理する機能である。グラフィックスオブジェクトはハードウェアサポートによって、回転、拡大、縮小などの処理が高速に行なわれる。一般的な携帯型ゲーム機にはこの機能が搭載されている。

- **回転・拡大・縮小**

  いくつかのパラメータをレジスタにセットすることで、グラフィックスオブジェクトもしくは背景画像を回転・拡大・縮小する機能である。

- **移動**

いくつかのパラメータをレジスタにセットすることで、グラフィックスオブジェクトもしくは背景画像を水平もしくは垂直方向に移動させる機能である。表示画面が画像を越えた場合は、その領域を透明にするか回り込んで表示させるかを選べる。

- **アルファブレンディング**

  グラフィックスオブジェクトもしくは背景画像の 2 面に対して半透明処理を行なう機能である。レジスタにセットするパラメータに応じて画像の透明度を変え、片方の面を透かして、もう片方の面が見えるような効果を作り出す。

- **フェードイン／フェードアウト**

  グラフィックスオブジェクトもしくは背景画像 1 面の輝度を変化させる機能である。レジスタにセットするパラメータに応じて画像の輝度を変える。輝度が増して白に近づくもしくは輝度が減って黒に近づくことでフェードアウト、逆に白または黒から輝度を戻すことでフェードインの効果を作り出す。

- **モザイク**

  画像の一部もしくは全体の解像度を落とし、粗いブロックの集まりに置き換える機能である。レジスタにセットするパラメータに応じてブロックの形状、サイズが決まり、ブロック中のあるピクセルのカラーがそのブロック全体のカラーとなる。

- **ダブルバッファリング**

  画像を格納するフレームバッファを 2 つ用意し、それを交互にディスプレイに出力する機能である。これにより、片方のフレームバッファをディスプレイに描画している間に、もう一方のフレームバッファへデータを書き込む、修正するなどの処理が可能となる。

このような機能を 3 次元グラフィックスの処理に使用すれば、3 次元グラフィックスを使用したアプリケーションを高速化できる。

4.2　メモリ構造

表 4.1で示したように、GBA は他の携帯端末と比較して RAM が格段に小さい。図 4.1 に GBA のメモリ構造の概要を示す。

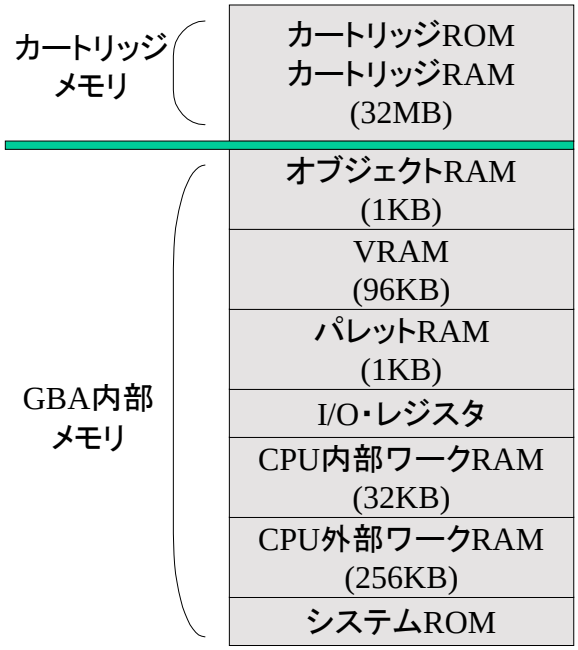

図 4.1　GBA のメモリ構造

GBA のメモリは、カートリッジ形式で装着するカートリッジメモリと、本体に内蔵されている内蔵メモリがある。GBA 上で動作するアプリケーションはカートリッジメモリに書き込まれ、カートリッジを装着することで読み込み可能となる。

GBA 内部メモリは機能ごとに細分化されており、全てを自由に使用することはできない。I／O・レジスタはハードウェアに関する状態を保存する各種のレジスタ、パレット RAM はパレット情報、オブジェクト RAM はオブジェクト管理のための情報、VRAM

は背景画像とオブジェクト画像などを保持し、その使用方法は既に定義されている。その他に、高速だが容量の少ない RAM 領域である CPU 内部ワーク RAM、低速だが容量の大きい CPU 外部ワーク RAM がある。これらは自由に使用可能であるが、前者は 32KB、後者は 256KB と、PC などと比べると RAM の容量が少ない。

RAM 容量の少なさは GBA の短所の一つである。GBA をプラットフォームとする場合、この点をふまえた開発を行なわなければならない。

4.3 開発環境

開発環境は任天堂とライセンス契約をすることで提供される。本研究では Windows2000（Cygwin）で動作する gcc のクロスコンパイラ環境を用いた。クロスコンパイラにより生成されたバイナリは、図 4.2の装置を用いて GBA 上で動作させることができる。開発には C 言語を用いるが、4.2節で述べた GBA 特有のメモリマップが適用されるほか、libc といった標準ライブラリが使用できないなどの制限がある。

図 4.2　GBA 開発環境

第5章　設計

この章では、本研究で開発するGBAGLの設計について述べる。GBAGLは、OpenGL準拠の3次元グラフィックスライブラリをGBA上で動作させるためのライブラリである。また、GBA上で3次元グラフィックスを使用する場合の問題点をあげ、その対応策を述べる。

5.1　GBAGLの設計

本論文の3.3節において、OpenGLをX Window System上で使用する際のライブラリ間の関係について示した。OpenGLとGLUはX Window Systemなどウィンドウシステムに依存しない設計がなされている。そのため、本研究においてもその部分を流用することができる。しかし、GLXなどウィンドウシステム機能拡張ライブラリはそのウィンドウシステム上で動作するように設計されているため、当然流用することはできない。したがって本研究では、GBAのハードウェア依存部を吸収し、入出力をサポートするライブラリGBAGLを開発する。GBAGLはGBA上で3次元グラフィックスを使用するアプリケーションによって呼び出され、OpenGL に含まれていないハードウェア依存の処理を行なう。GBAGLのAPIはGLXのものを使用する。こうすることで、アプリケーションの開発者はGBAの知識無しに、あたかもX Window System上で動作するアプリケーションを開発するかのように、GBA上で動作する3次元グラフィックスを使用したアプリケーションを開発できる。

本研究では、GBAの2次元グラフィックス向けのハードウェアサポートを用いた高速化を図る。したがってOpenGLはGBAハードウェアにアクセスする必要がある。よって、OpenGLをGBA向けに書き換えなければならない。OpenGLレンダリングエンジンをGBAのハードウェアにアクセスするように直接書き換えるのには2つの問題がある。

1つはGBA上で動作するアプリケーションの開発者に対し、GL選択の自由を奪うことである。OpenGLはGLのAPIを定義した規格であるが、GL内部の処理などについては規定されていない。例えばX Window System上で動作するアプリケーションの開発者は、GLにMesaやTinyGLを選択することができる。このようにアプリケーションの開

発者は、OpenGL 規格に準拠した GL の中から自分の要求に合ったものを選択できるのが望ましい。

もう 1 つは、ハードウェア依存部分と非依存部分を GL 内に共存させてしまうことである。図 3.1で示したように、OpenGL 規格に準拠した GL はハードウェアに依存しないように設計されている。ライブラリは機能ごとに分割しているのが望ましく、そうなっているライブラリを混ぜ合わせることは望ましくない。

このような問題を考慮して、本研究では GL 内の GBA のハードウェアにアクセスするように改変した部分を切り離し、GBAGL に内包する。GL はその切り離された GBA 依存のモジュールを呼び出す。これにより GL 自体の改変は最小限に抑える。この設計によりアプリケーションの開発者が、他の OpenGL 準拠の GL に変更したい時に必要な修正を、最小に抑えることができる。図 5.1に GBA で OpenGL 準拠の GL を利用する時のモジュール構成を示す。

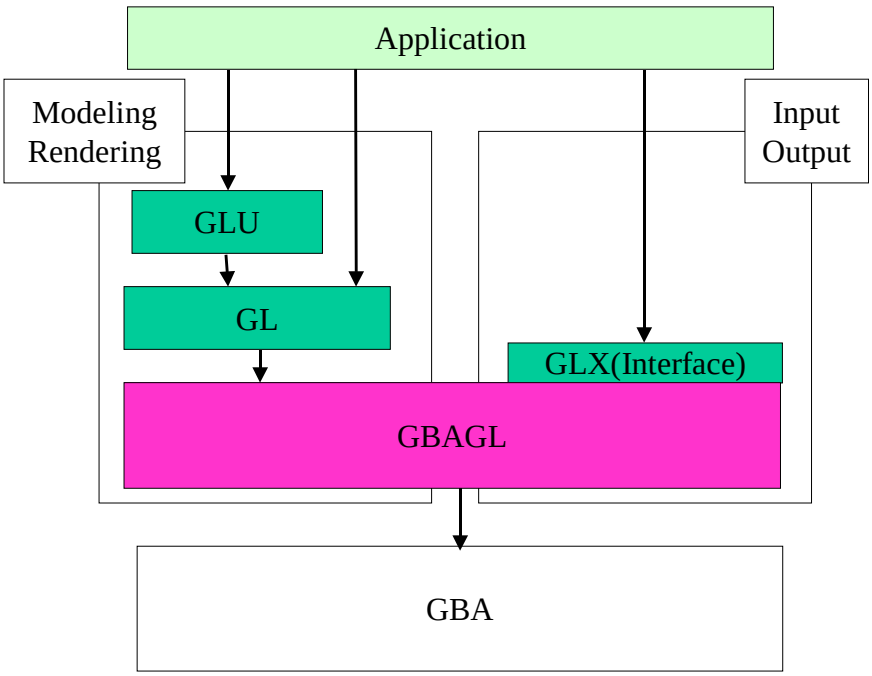

図 5.1　GBAGL のモジュール構成

5.2 GBA の困難

GBA 上で 3 次元グラフィックスアプリケーションを動作させる際、PC 上で動作させる場合と比較して、いくつかの困難が生じる。以下にその困難と対応策を示す。

5.2.1 解像度の不足

表 4.1で示したとおり、GBA の解像度は 240×160、画面サイズは 2.9 インチである。これは 15〜21 インチといった PC で使用するディスプレイに比べて格段に小さい。3 次元グラフィックスを用いたアプリケーションは主に PC 上で動作するように設計されている。このようなアプリケーションはディスプレイに出力されることを前提として、細かく、複雑な表現がなされているかもしれない。GBAGL は X Window System で動作する OpenGL 使用のアプリケーションの移植が容易に行なえるように設計されている。しかし、そのようなアプリケーションをそのまま移植して GBA で動作させた場合、細かい、複雑な表現がつぶれてしまう可能性がある。

この困難の対応策として、GBA のハードウェア機能を用いて、処理量の増加を伴わずに擬似的にサイズを拡張することが可能である。しかし本研究の実装ではこの方法を用いない。大規模で複雑なシーンを処理するためには、莫大な CPU の処理能力が必要となるからである。以下に GBA のハードウェア機能である縮小と移動を用いた例を示す。図 5.2は縮小と移動を用いた結果を表している。縮小、移動とも用いなかった場合のイメージは、図中の移動を用いた場合の GBA 画面に表示されているものと同じになる。

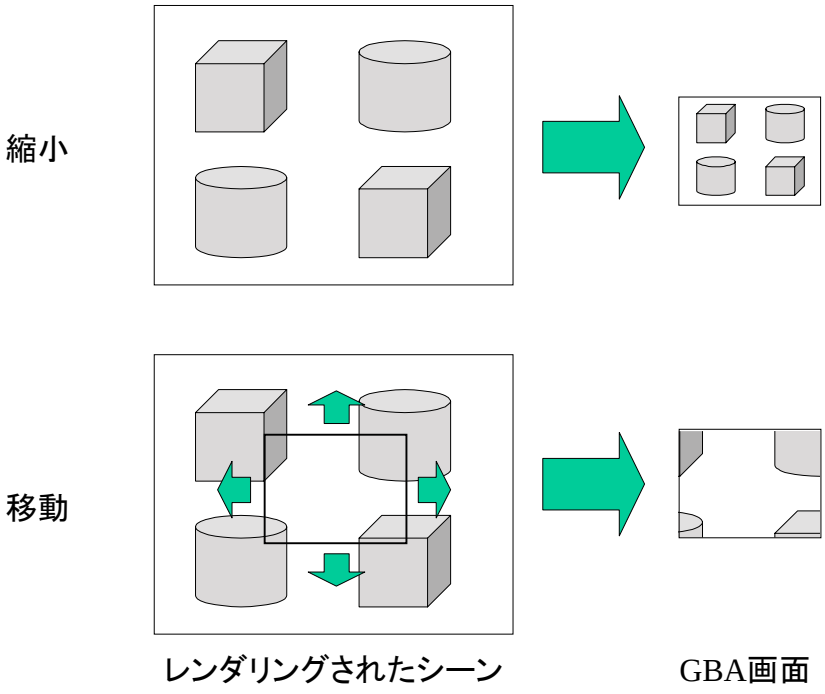

図 5.2　縮小と移動を用いた解像度擬似拡張

縮小を用いた場合、シーン全体がレンダリングされ、その結果であるイメージがハードウェアにより縮小されて画面に出力される。これにより、GBA の解像度にあわせたイメージより広範囲のシーンを描画できる。移動を用いた場合も同様にシーン全体がレンダリングされ、イメージ内の指定された領域が画面に出力される。これにより、画面を上下左右に移動することで、擬似的に GBA の画面より大きなイメージを観ることができる。なお、何もしなかった場合、シーンが GBA の画面にあわせてクリッピングされ、レンダリングされたイメージが画面に出力される。

GBA のハードウェア機能である縮小、移動を用いる利点は、一度そのシーンをレンダリングしてしまえば、視点をずらしたい時や拡大縮小したい時に再度レンダリングする必要がないことである。しかし、一時に画面の解像度より大きなイメージを生成しなければならないため、大量の処理が必要となる。一方、ハードウェア機能を用いなかった場合には GBA の解像度にあわせてシーンがクリッピングされ、レンダリングされて画面に出力される。そのため、イメージ生成の処理量は少ない。しかし、視点をずらした

い時、拡大縮小したい時には再度シーンをモデリング、レンダリングしなければならない。これは一長一短である。しかし、GBA 上で動作するアプリケーションが、GBA の携帯性を意識した、インタラクティブでリアルタイム処理が必要なアプリケーションであった場合、頻繁にシーンが更新されると予測される。静止画であれば GBA のハードウェア機能の効果が期待できる。しかし、頻繁にシーンが更新されると、その都度画面の解像度より大きなイメージをモデリング、レンダリングしなければならず、無駄に処理量が増加する。表 5.1に GBA 機能の使用と処理量を表した表を示す。ここで、GBA 機能を使用した場合のイメージサイズは使用しない場合の n 倍とし、GBA 機能を使用しない場合のイメージ生成にかかる処理量を 1 とする。

表 5.1　GBA 機能の使用と処理量

GBA 機能の使用	イメージ生成	拡大・縮小・移動	シーン更新
する	n	0	n
しない	1	1	1

対応策を講じないことで、シーンは GBA の解像度にあわせてクリッピングされることになる。しかし、クリッピングされると処理量が減り、結果として高速に動作する可能性がある。

5.2.2　処理能力の不足

GBA 上で動作させる 3 次元グラフィックスを用いたアプリケーションは、屋外に携帯した状態での使用を想定していると考える。そのようなアプリケーションはリアルタイムでインタラクティブな処理が要求される。GBA は 32 ビットの RISC-CPU を搭載しており、他の携帯端末に比べると処理が高速であるが、PC より格段に遅い。したがって、大規模で複雑なシーンを扱う場合、処理能力が不足する可能性がある。

この問題に対し、本研究ではライブラリを軽量化、高速化することで解決する。その

ために、3.5.2節で紹介した TinyGL を参考に GBA 上で動作する 3 次元グラフィックスライブラリを実装する。TinyGL は軽量化、高速化された OpenGL の極小のサブセットである。OpenGL や3.5.1節で紹介した Mesa は、汎用性を持たせるために大規模になっているが、一般的な使用において使われない機能が多々ある。TinyGL はそのような部分を削ったことで、軽量化、高速化を図っている。このような TinyGL の設計方針は本研究で実装するライブラリのそれと一致している。

5.2.3 RAM の不足

表 4.2で示したとおり、PC と比較してはもちろん、他の携帯端末と比較しても、GBA は RAM の容量が格段に少ない。3 次元グラフィックスの処理を行なう場合、フレームバッファ（ダブルバッファリング使用時には 2 つ）の他、効果によっては深度バッファ、ステンシルバッファなど、多くの RAM が必要になる。このような機能を使用した場合、RAM が不足する可能性がある。

この問題に対しての解決策の 1 つは、5.2.2節でも述べたが、TinyGL を参考に、ライブラリを軽量化、高速化することである。ライブラリを軽量化することによってメモリの使用量は削減される。

もう 1 つの解決策は、GBA に特化した実装を行なうことである。4.2節で述べたように、GBA は特殊なメモリマップを持ち、用途が決まっている領域が多い。しかし、GLX は、PC の UNIX、Linux などの OS 上において動作することを前提とした作りになっているため、メモリを浪費しがちである。GBA 上で動作する 3 次元グラフィックスライブラリはそのような部分を修正し、メモリを節約する。

第6章　実装

この章ではGBAGLの実装について述べる。GBAGLはGLXを修正して実装を行なったため、GLXと対比しながら説明する。初めに実装の方針を述べ、次に具体的な実装について述べる。なお、GBAおよびその開発環境の詳細な仕様は機密保持契約の許す範囲で説明する。

6.1　実装方針

6.1.1　X Window Systemとの分離

TinyGLは、モデリングやレンダリングを行なうGLと、X Window Systemとのやりとりを行なうGLXを内包している。本研究ではTinyGLのGLXを参考に、GBAGLの開発を行なう。GLXのインターフェイスをそのままに、GBAとのやりとりを行なうGBAGLを開発することで、OpenGL準拠のGLを使用したアプリケーションを、GBA上で改変無しに動作させることが可能となる。

X Window Systemは、Xプロトコルによりサーバ／クライアント環境を実現している。GBA上で動作させるアプリケーションは、基本的に単一計算機での動作を前提としてよい。そのため、GBAではサーバ／クライアント環境を考えない。したがってGLXを参照してGBAGLを開発する時、GLXにおけるXプロトコルを使用している部分は排除するか単一計算機で動作するように修正しなければならない。

また、X Window Systemはその名のとおりウィンドウシステムを提供しているが、GBAの画面サイズを考えると、ウィンドウシステムを実行させることは難しいと考える。よって、GLXの複数もしくは単一のウィンドウで動作させることを想定した部分も同様に排除もしくは修正しなければならない。

逆に、X Window SystemがOSに任せている処理、例えばメモリの初期化、割り込み処理の設定などは、GBA上で3次元グラフィックスを使用するアプリケーションの開発者が、GBA上で動作させることを意識せずに実装できるようにするべきである。

これらのことを踏まえて、TinyGL本体及びGLXのX Window Systemで動作すること前提に実装されている部分、例えばX Window Systemの関数、構造体、を使用している

部分を適宜修正した。その他、UNIX／Linux 上で動作することを前提に実装されている部分、例えばC 言語標準ライブラリ（libc）を使用している部分も適宜修正した。

6.1.2 サンプルコードの分析

実装は、まず TinyGL 付属のサンプルの動作を目標とした。基本的な実装方針は、サンプルのコードを分析し、プログラムの流れにおいて X Window System の関数、構造体、定数が使用された時にそれらを GBA 用に修正することとした。このような方針をとることで必要な関数、構造体、定数のみを実装することができ、ライブラリの軽量化、軽量化による高速化が期待できる。

6.2 画面モードの設定

GBA 上でアプリケーションを動作させる際、まず画面モードを設定しなければならない。画面モードとは、背景およびグラフィックオブジェクトの色数、画像サイズ、ダブルバッファリングのような特殊効果使用の可否などのセットである。本研究で使用されうる画面モードを表 6.1に示す。

表 6.1 画面モード

画面モード	色数	画像サイズ	ダブルバッファリング
A	32768 色	240 × 160	不可
B	32768 色	160 × 128	可
C	256 色	240 × 160	可

本研究では画面モードを C に定めた。画面モード B、画面モード C はダブルバッファ

リングが使用可能である。ダブルバッファリングを用いることで、動画を効率よく処理することができる。画面モードCは色数が画面モードBに比べて少ないが、画像サイズは大きい。GBAはディスプレイが小さいため、できるだけ大きな画像を表示することで、使用者は画像から的確な情報を得ることができる。

6.3 ヘッダの再構成

3.2.3節と6.1.1節で述べたように、TinyGLはGLXを内包しており、GLXはX Window System上でOpenGL準拠のグラフィックスライブラリを動作させるためのものである。よって、GLXはX Window Systemの関数や構造体を多用している。さらに、グラフィックスライブラリでは標準ライブラリを使用している。図 6.1にTinyGLにおける主要なヘッダの包含関係を示す。図中の四角形はC言語ソースファイルの集合を示し、角の丸い四角形はヘッダファイルの集合を示す。

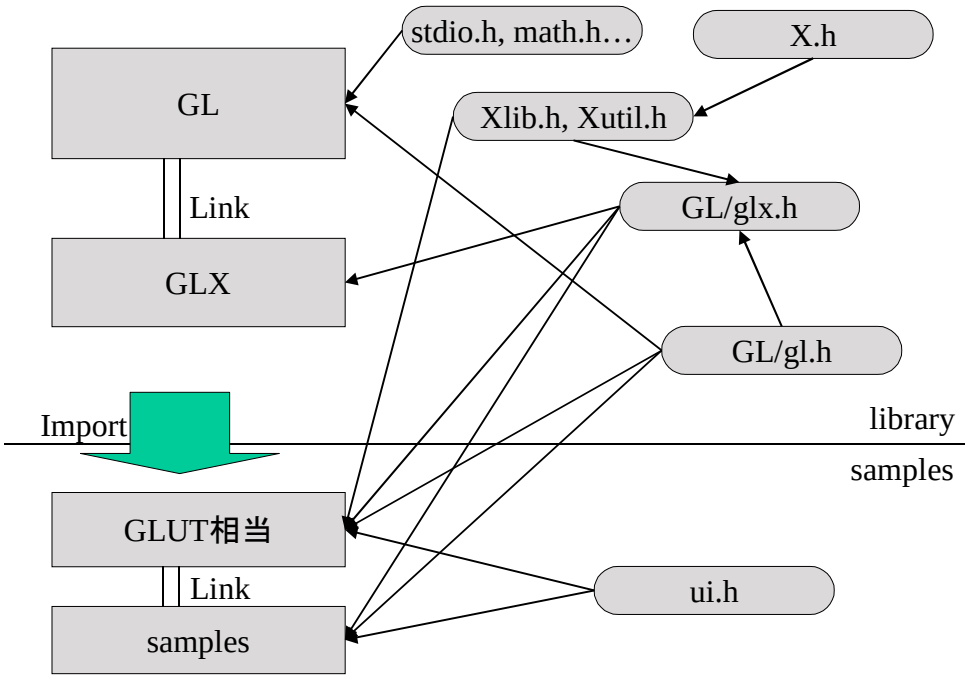

図 6.1　TinyGL のヘッダ包含関係

TinyGL はモデリング、レンダリングなどを行なうグラフィックスライブラリと、X Window System とのやりとりを行なう GLX を含んでいるが、それに加えてループを生成するなど GLUT に相当するモジュールと、いくつかのサンプルも付属している。サンプルは GL、GLX、と GLUT 相当モジュールを使って 3 次元グラフィックスを使用する。グラフィックスライブラリは標準ライブラリを使用するために stdio.h などのヘッダファイルをインクルードする。GLX は X Window System の X プロトコル、ウィンドウシステムの機能を使用するためのヘッダファイルである Xlib.h などをインクルードする。GLUT 相当モジュールも Xlib.h などを含む他、グラフィックスライブラリを使用するための gl.h、GLX を使用するための glx.h と、GLUT 相当モジュールを使用するための ui.h をインクルードする。これは非常に複雑な包含関係である。また X.h、Xlib.h などは巨大であるため、GBA 上で動作するアプリケーションにおいてもそのままインクルードすることは性能低下に繋がる。このような点をふまえて、次のように実装した。

GLX を GBAGL に修正する。X Window System に関連したヘッダファイルである X.h、

Xlib.h、Xutil.h 中の GBAGL に必要な部分のみを抽出し、その部分のみを実装する。その抽出は GLX の GBAGL への修正と同時に行なわれる。例えば、巨大な構造体において GBAGL が一部のメンバのみを必要とする場合、代わりの構造体を作る、ダミーのメンバを作成するなどして必要なメンバのみを実装する。これらを gbagl.h にまとめる。

アプリケーションの開発者は GLUT を使用することで、動作するウィンドウシステム、プラットフォームの差異を気にせずにすむ。それは GBA というプラットフォームにおいても適用されるべきである。そのため、本研究では GLUT 相当モジュールを GLUT の API として実装した。GLUT 相当モジュールは GLUT の機能の一部のみをサポートしているに過ぎない。しかし、それはよく使用される機能であるため、アプリケーションの開発者はこれを利用し、効率的に開発を進めることができるだろう。このようなことから GLUT 相当モジュールを GBA 上で動作するように修正し、GBAGL に内包することとした。

C 言語標準ライブラリは GBA では使用することができない。GL が C 言語標準ライブラリの関数を使用する場合、それが 3 次元グラフィックスの動作に必要なものであれば独自に実装し、そうでなければ削除した。標準ライブラリの代替関数や、その他の独自に実装したデバッグや画面描画の関数は KGL としてまとめた。KGL は主に GBA のハードウェアに依存する機能を提供し、GBAGL が GBA のハードウェアに対する処理をしたい時は、KGL の関数を用いて間接的に行なう。

このような実装による、GBAGL のモジュールとヘッダの包含関係を図 6.2に示す。GBAGL は GLX と GLUT 相当モジュールの API を継承し、処理は GBA 用に修正される。KGL は GBA に依存した処理がまとめられたモジュールであり、GBAGL や GL がそれを用いる。GL の改変はほとんど無い。

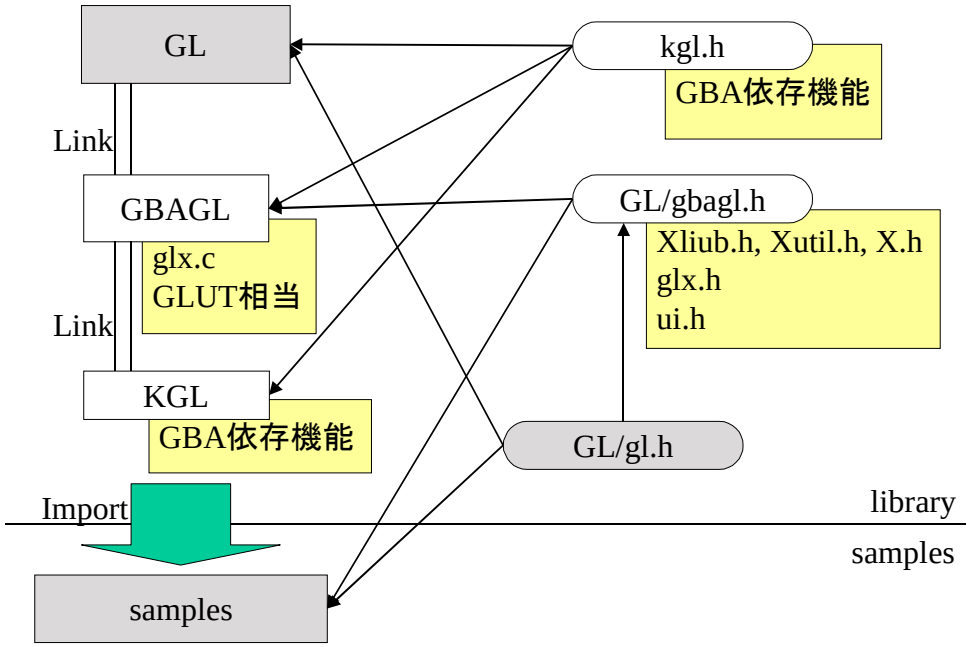

図 6.2　GBAGL のヘッダ包含関係

6.4　GBA の初期化

GBA 上で動作するアプリケーションは、まずハードウェアの初期化とレジスタの設定を行なう必要がある。ハードウェアの初期化とは、RAM のクリア、グラフィックオブジェクトの初期化、割り込みルーチンの設定などである。レジスタは GBA の状態保持領域であり、カートリッジへのアクセス方法、割込み許可、画面モードなどの情報について、アプリケーションによって適切な値をセットする。

GBA の初期化は必ず初めに行なわれなければならない。よって GBAGL にプログラム開始関数を用意し、その関数内で初期化を行なったあとにアプリケーションの main 関数を呼ぶこととする。これによって GBA の初期化は初めに行なわれる上に、GLX を用いて X Window System 上で動作するアプリケーションをコードの改変無しに実行できる。

6.5 ハードウェア情報取得

GBA の初期化を行なった後、GBAGL はハードウェアに関する情報を取得する。GLX は X プロトコルの使用を前提としているので、GLX のハードウェア情報取得は X プロトコルに関する処理を多分に含む。GBAGL ではそれらを削除するか、GBA のために修正しなければならない。

以下に、GLX において呼び出される関数を順に挙げ、それらとそれに関連する構造体の修正内容を示す。なお、X Window System で提供されている関数は接頭辞が X 、GLX で提供されている関数は接頭辞が glX となっている。

- **XOpenDisplay**

  X Window System が提供する、GLX で最初に呼ばれる関数である。それは環境変数 DISPLAY を取得し、エラーハンドラを設定し、ディスプレイ情報を保持する構造体 Display の宣言、初期化をした後に、サーバ／クライアント環境の構築を行なう。GBAGL においてはサーバ／クライアント環境を構築しないのでそれに関する処理を行なう必要がない。構造体 Display はサーバ／クライアント環境に関する情報を包括的に保持しているが、GBAGL においてはそのような情報は必要がない。しかし GLX はサーバ／クライアント環境を前提としているため、関数の引数に構造体 Display を含む。よって、構造体 Display の定義を変更し、ダミー変数として処理する。

- **glXChooseVisual**

  引数などで指定されたビジュアルの構造体 XVisualInfo を返す関数である。ビジュアルは、画像が出力される際のサイズや色数、カラーマスクなどの属性であり、構造体 XVisualInfo はそれらビジュアルの値を保持する。GLX では X Window System に適切な値を要求する。このようなビジュアルは GBAGL では必要ない。なぜなら GBA は X Window System のようにハードウェアプラットフォームが変化せず、一意に定まっているからである。動作する環境が一定なので、それを調査し処理に反映させなくともよい。よって XVisualInfo は GLX のインターフェイスとの互換性を維持するだけの最小限のメンバだけを保持する構造体に修正し、固定値を与える。

6.6　コンテキストの初期化

GLX コンテキストは、GLX と GL の状態変数を全て保持する。それには画面サイズやディスプレイの情報、カラーマップといった画面に関する情報や、照光処理、テクスチャ、ビューポートといった GL の状態に関する情報（GL コンテキスト）を含む。GLX では GLX の初期化を行なった後に GLX コンテキストを生成する。以下に GLX コンテキスト生成、初期化に関する関数を挙げ、それらとそれらに関する構造体の修正内容を述べる。

- **glXCreateContext**

  GLX コンテキストの構造体を生成する関数である。この構造体は GLX 内で多分に参照されるため、変更しない。

- **glXMakeCurrent**

  フレームバッファ、深度バッファ、カラーマップの生成といった GLX コンテキストの設定と、GL コンテキストの初期化、ビューポートの設定を行なう関数である。GLX では構造体 Display のメンバなどを参照し、画面に適したフレームバッファ、深度バッファ、カラーマップを生成する。しかし GBAGL においては画面が一意に定まっているので、それらを動的に生成する必要がない。

  フレームバッファ、深度バッファはサイズを指定して生成、管理される。これによりメモリ管理が簡略化され、処理のオーバーヘッドを軽減することができる。

  カラーマップはあらかじめ生成してカートリッジ ROM に書き込んでおき、GBA のパレット領域にコピーされる。GLX ではカラーマップは画面の属性により動的に生成される。しかし、例えば 256 色の環境ではディザ減色を考慮したカラーマップとその逆引きテーブルを生成しなければならないなど、多くの処理と大量のメモリが必要とな

る。GBAGL ではカラーマップをあらかじめ生成しておくことで処理量を軽減し、メモリの使用量を節約する。

GL コンテキストは依存するものではないので変更しない。ビューポートの設定は画面のサイズを参照するので、GBA の画面サイズにあわせて設定する。

6.7 標準ライブラリ関数の実装

GLX コンテキスト、GL コンテキストを設定したのち、アプリケーションは 3 次元グラフィックスシーンを生成する。3 次元グラフィックスシーンの生成は GL が行なう。GL は OpenGL 準拠によりプラットフォームに依存しない設計になっているものの、標準ライブラリを使用している。C 言語が使用可能な環境では、通常は標準ライブラリが含まれているが、GBA には標準ライブラリが含まれていない。よって、GL もしくは GLX で使用される標準ライブラリ関数を独自に実装する必要がある。以下に実装した関数を挙げる。

- **メモリ管理（malloc, free）**

  TinyGL では、組み込みシステムに容易に適応できるように、メモリ管理に関する関数を間接的に使用している。GBA で動作するようにそれを修正するのは容易である。

  GBA のメモリ構造を4.2節で説明した。CPU 内部ワーク RAM は高速であることから、主に関数内の局所データを保持するのに使用される。CPU 外部ワーク RAM は特に用途が定められていない。そこで、この CPU 外部ワーク RAM をヒープとして用いることにし、それをメモリ管理関数によって管理する。

  malloc、free は、UNIX で用いられる実装法を参考に、簡易的に実装した。まず、外部ワーク RAM の使用された領域の最後尾のアドレスを示すポインタ SP（Stack Pointer）と、使用後に解放された領域を示すポインタ FP（Free Pointer）を用意する。malloc は、要求されたサイズの領域を、SP を先頭にして確保する。その際ヘッダを作成し、サイズを添付する。free は要求されたポインタの示す領域を開放する。FP は free によ

って解放された領域群の、メモリアドレス順で先頭の領域を示す。解放された領域はヘッダが拡張され、メモリアドレス順で次の解放された領域を示すポインタが追加される。malloc と free の例を図 6.3に示す。図中の太枠内が要求された領域である。

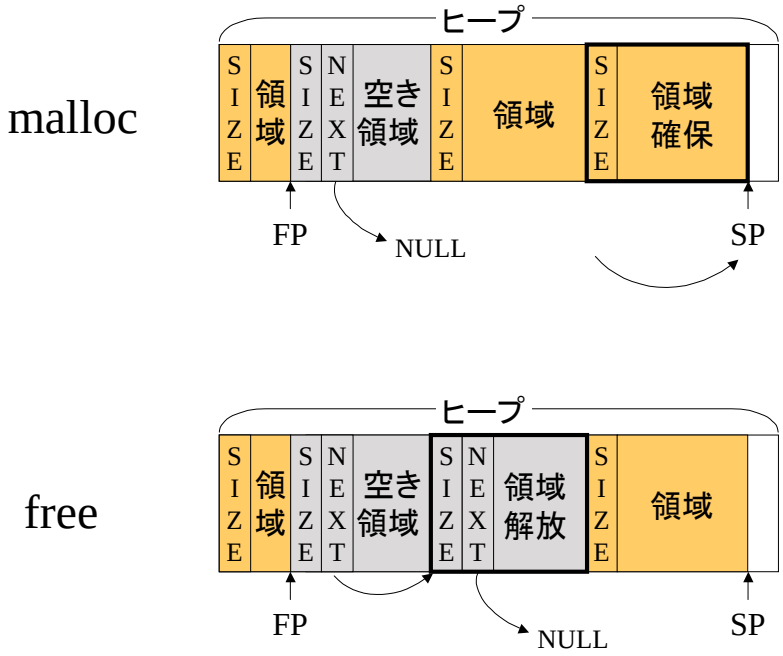

図 6.3　malloc と free の例

解放されたある領域が他の既に開放された領域と隣接していた場合、それらは結合される。また、解放されたある領域の最後尾が SP と一致した場合、SP をその解放された領域の先頭を示すように付け替える。これらの処理の例を図 6.4に示す。これらの処理を行なうことでヒープを再利用し、効率的に使用する。

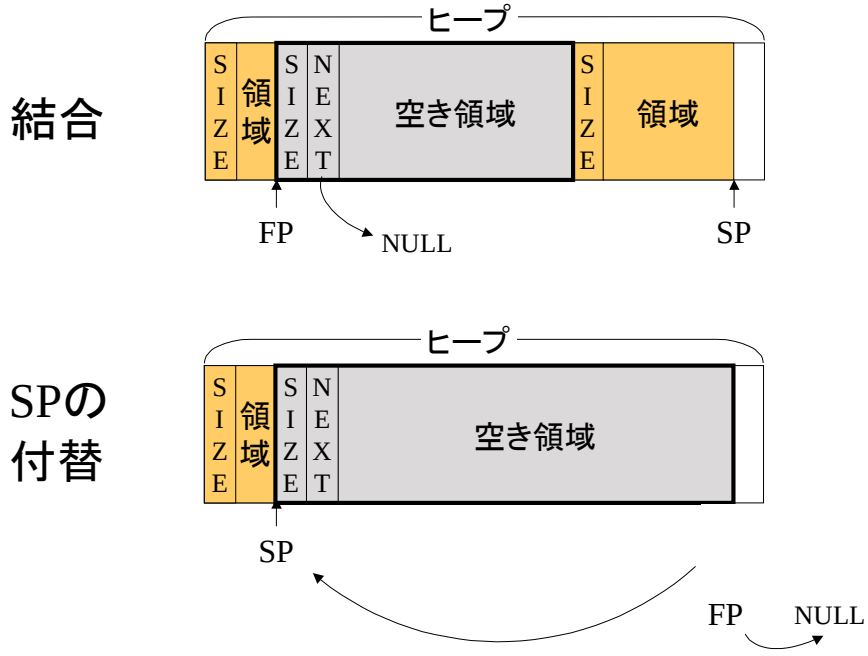

図 6.4　空き領域の再利用

malloc、free の使用順序によっては空き領域を効率的に使用することができない。これを改善するためにはガーベジコレクションが必要となる。しかしガーベジコレクションは実装に手間のかかる上処理量が大きく、アプリケーションの性能に影響を及ぼすため、本研究では使用しない。

- **メモリコピー、メモリクリア（memcpy, memset）**

  メモリコピー、メモリクリアは GBA のハードウェアによるダイレクトメモリアクセス（DMA）で実現する。DMA は GBA のライブラリに含まれているので、それらは容易に実装できる。

- **エラー処理（assert, printf）**

  GL 内では頻繁にエラー判別関数 assert を呼び出している。また、printf により状態を出力することもある。

assert は、真偽の判定をし、結果によってプログラムを終了する関数である。これは define などを用いて容易に実装できる。しかし GBA には標準出力がないため printf は使用できない。エラー出力がないことは開発を非常に困難にする。よって、独自にエラー出力機構を実装した。これは表示させたい数字のビットマップを、任意の座標に該当する VRAM のアドレスに出力するものである。

6.8 キー入力

GLX において、キー入力は X Window System の関数を用いる。GBA ではキーが入力されると所定のレジスタに書き込まれるので、それを参照すればよい。3 次元グラフィックスのデモでは PC 用キーボードでは Escape キーと十字キーが多用される。このため、本研究ではそれぞれに START ボタンと十字キーを割り当てた。PC 用キーボードと比べると GBA のキーは少ないが、この問題はアプリケーションの開発者が解決せざるを得ない。

6.9 描画

GL によってモデリング、レンダリングとある程度のフラグメント演算が行なわれた後に、関数 glXSwapBuffers が呼び出される。この関数はディザ処理、フレームバッファへの転送を行なう。

glXSwapBuffers は、まずディザ処理を行なう。6.2節で述べたように、本研究では表示色数は 256 色に設定した。TinyGL は内部では 65526 色で画像処理を行なっているので、ディザ減色を行なう必要があるからである。

次に glXSwapBuffers はフレームバッファへイメージの転送を行なう。GLX では X プロトコルを用いてサーバからクライアントにイメージを転送し、表示させる。しかし、GBAGL では X プロトコルを用いない。また、GBAGL ではダブルバッファリングを用いる。よって、GBAGL では表示していないフレームバッファにディザ減色したイメージを直接書き込む。これにより、イメージ転送のオーバーヘッドを削減することができ

る。また、VRAM は外部ワーク RAM より高速にアクセスできるため、高速化に繋がる。

　フレームバッファへイメージの転送が完了したら、レジスタの設定を書き換え、そのイメージが転送されたフレームバッファを次の描画期間において表示させる。これにより 3 次元グラフィックスが画面に出力される。

第 7 章　評価

この章では実際にデモプログラムを動作させ、評価し、問題点を述べる。また、GBA のハードウェア機能を用いた高速化手法について議論する。

7.1　TinyGL のデモプログラム

実装した GBAGL を用いてそのデモプログラムを動作させた。動作させたデモプログラムは以下の 2 つである。

- **spin.c**

  3 次元空間にワイヤーフレーム（線分のみ）で作られた直方体を描く。直方体の頂点には異なったカラーが設定されており、異なったカラーの 2 頂点を結ぶ辺はグラデーション（階調）効果を受ける。直方体は 3 次元空間内の X 軸、Y 軸、Z 軸について順に回転する。

- **gears.c**

  3 次元空間に、大小の歯車を描く。歯車は 3 つあり、それらの大きさは違う。ひとつの歯車には歯が 20 付いており、3 つの歯車は連動して回転する。オブジェクトには照光処理と、それに必要なマテリアル（材質）設定がなされている。

幾何学プリミティブの量、特殊効果の使用などから、gears.c は spin.c より処理量が多い。

これらのデモプログラムは Mesa の開発者である Brian Paul が作成したものであるが、Brian Paul はこれらのデモプログラムについての権利を放棄している。TinyGL の開発者である Fabrice Bellard はこれらのデモプログラムを TinyGL に添付し、TinyGL が Mesa と互換性があることを示している。

なお、これらのデモプログラムは、GBAGL の使用のために若干修正される。具体的にはヘッダのインクルードなどである。

7.2 デモプログラムの動作

7.1節で挙げたデモプログラムを動作させた結果であるスクリーンショットを以下に挙げる。なお比較のために、TinyGL を使用して X Window System 上で動作させたデモプログラムのスクリーンショットも同時に挙げる。なお、TinyGL 内の GL は開発中であるためバグが混在していたが、TinyGL を GBA 上に移植する過程で修正した。

図 7.1に TinyGL を使用して spin.c を動作させたもの、図 7.2に GBAGL を使用して spin.c を動作させたものを示す。同様に、図 7.3に TinyGL を使用して gears.c を動作させたもの、図 7.4に GBAGL を使用して gears.c を動作させたものを示す。

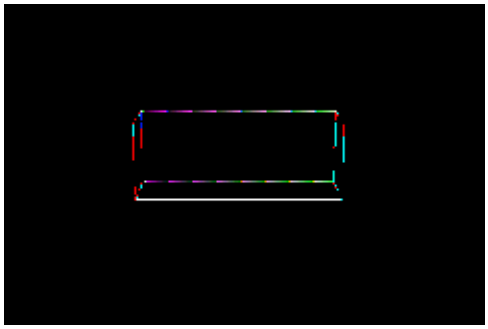

図 7.1　spin.c -TinyGL

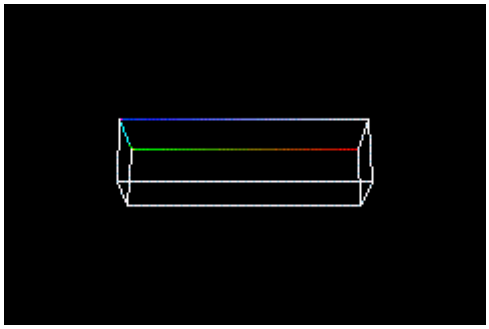

図 7.2　spin.c -GBAGL

図 7.3　gears.c -TinyGL

図 7.4　gears.c -GBAGL

7.3 評価

まず、図 7.1と図 7.2を比較する。両者のワイヤーフレームの形状については違いが見受けられない。GBA 上で動作させた図 7.1は色数制限から 256 色にディザ減色されている。しかし、滑らかにグラデーションがかかっており、正常に表示されていると考えられる。逆に X Window System で動作させた図 7.1は、GL のバグによる画像の乱れが見える。

図 7.3、図 7.4も同様に、ディザ減色されていることによるカラーの違いが見受けられるが、オブジェクトの形状、照光処理に関しては問題ない。スクリーンショットに関しては、色数の違いのみが画質の差異を生み出しているといえる。

GBAGL は TinyGL 内の GL を GBA で動作させるためのライブラリである。GBA で動作させた 3 次元グラフィックスのモデリング、レンダリングは TinyGL 内の GL といってよい。つまり、イメージ生成まではハードウェア制限によるディザ減色以外は同じ処理を行なっている。画質に変化が見られないということは、GL 及び GBAGL が正常に動作していることに他ならない。

次に、デモプログラムの動作速度を表 7.1に示す。有効桁は 2 桁とした。FPS（Frames Per Second）は、一秒間に画面が再描画される回数を表す値である。なお、デモプログラムの動作において、TinyGL を X Window System 上で動作させた際の動作環境を表 7.2に示す。

表 7.1　デモプログラムの動作速度

動作環境＼デモプログラム	**spin.c**	**gears.c**
X Window System	300 FPS	170 FPS
GBA	6.0 FPS	0.10 FPS

表 7.2　X Window System の動作環境

OS	Red Hat Linux release 7.2 Kernel Version 2.4.7-10
CPU	Intel Pentium4 1.7GHz
メモリ	256MB

GBA 上で動作デモプログラムを動作させる時、spin.c の場合は 50 倍、gears.c の場合は 1700 倍の差がある。これはメモリ転送速度の差もあるが、CPU の性能差が大きい。それは幾何学プリミティブの量に大きく反映される。gears.c は spin.c より幾何学プリミティブの数が多いので、CPU の性能差が如実に現れている。

絶対的な評価においても、GBA 上での動作速度は非常に遅い。静止画を扱うのであれば現段階でも GBA は使用されうる性能を発揮している。しかし、動画のように画面が頻繁に再描画されるアプリケーションにおいては、GBA のハードウェア機能により GL の処理を高速化しなければ、再描画間隔が非常に長くなってしまう。

第8章　　議論

この章では、GBA のハードウェア仕様やハードウェアでサポートしている機能を用いた高速化手法について議論する。

8.1　回転、拡大、縮小、移動

GBA ではハードウェア機能により、背景画像の回転、拡大、縮小、縦または横への移動が可能である。この機能を 3 次元グラフィックスに用いる場合について述べる。

OpenGL では、初期設定において視点の位置、向き、天地方向が API により指定される。これはビューポート行列を生成し、ビューポート変換に用いられる。ビューポート変換はいわゆるアフィン変換 [23] であり、3 次元シーンのオブジェクトが前もって回転、拡大、縮小、平行移動される。これに GBA のハードウェア機能である回転、拡大、縮小、移動を用いる。生成されたビューポート行列を参照し、回転、拡大、縮小、移動を用いればよい。

この手法の問題点の 1 つは、GBA のハードウェア機能を用いることができないものがあることである。図 8.1にその例を示す。視体積を前後に動かすのは拡大、縮小と同義であるため、視体積がオブジェクトと重ならないならば正常に動作する。視体積の底面の法線方向に対する回転はイメージの回転と同義であるため、正常に動作する。視体積の平行移動はイメージの移動と同義であるため、正常に動作する。つまり、3 次元空間における視体積の平行移動と、視体積の底面の法線方向に対する回転については GBA のハードウェア機能を使用することができる。しかし、視体積をその底面の法線方向以外について回転させることはできない。なぜなら、回転前に陰面処理で表示されていなかった面が表示されてしまうからである。このような回転に関してはハードウェア機能を用いることができない。

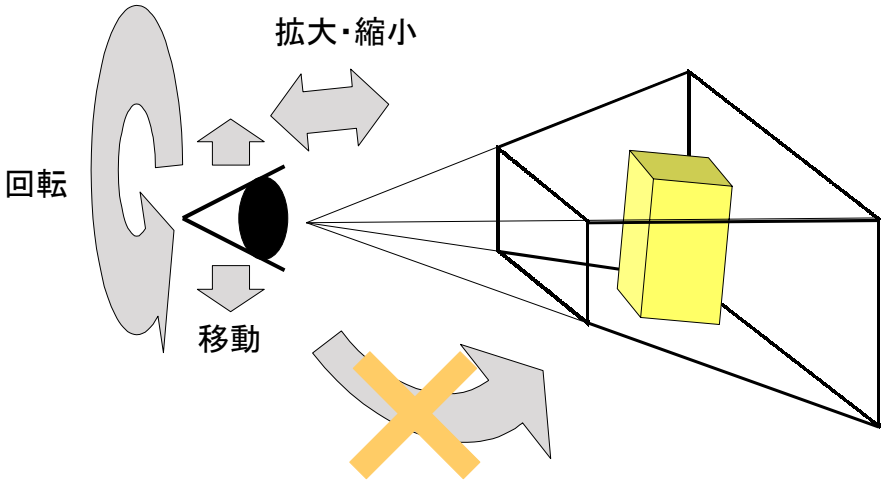

図 8.1　回転、拡大、縮小、移動の適用

もう 1 つの問題点は、3 次元シーン内の全てに対して同様のビューポート変換がなされたときのみ適応できることである。OpenGL はスタックのようにプッシュ、ポップすることで一部のオブジェクトに対してのみの変換を指定することが可能である。これにより個別にビューポート変換を指定された場合、その個別のビューポート変換を行なった後に、GBA のハードウェア機能を適用しなくてはならない。

8.2　グラフィックスオブジェクト

グラフィックスオブジェクトは GBA のハードウェア機能により回転、拡大、縮小、移動が高速に行われる。このハードウェア機能を 3 次元グラフィックスに使用する場合について述べる。

3 次元シーンにオブジェクトを描画するとき、アプリケーションは GL にオブジェクト生成のコマンドを発行する。回転、拡大、縮小を行なう場合は、それに関連したコマ

ンドを発行し、モデルビュー行列が生成され、モデルビュー変換が行なわれる。もし、3次元シーン内のあるオブジェクトのみが、拡大、縮小のために頻繁に再描画されるならば、GL にそのオブジェクトのみをモデリング、レンダリングさせた後、グラフィックスオブジェクトとして扱うことが可能である。この場合、回転は視体積の底面の法線方向についてのみ可能である。

以下に実装例を挙げる。あるオブジェクト A がディスプレイリストに登録されるとき、それをグラフィックスオブジェクトの使用対象として明示する。GL はそのオブジェクト A を処理してイメージとして出力する際に、出力先をグラフィックスオブジェクトの画像領域とする。GBAGL はそれをグラフィックスオブジェクトとして処理し、適切な位置に描画する。もし、そのオブジェクトが拡大、縮小、移動などにより再描画される必要があるとしても、GL に対してそのようなコマンドを発行し、再描画させなくてよい。そのコマンドを GBAGL によって処理し、拡大、縮小、移動する。また、一時的に表示させないようにすることも可能である。

この手法の 1 つめの問題点は、オブジェクトの重ね合わせが自由に行なえないことである。例えば図 8.2のように、視点からオブジェクト A、オブジェクト B、オブジェクト C の順に深度が定まっており、オブジェクト B のみが頻繁に再描画されるとする。このような場合はオブジェクト B にグラフィックスオブジェクトを適用し、オブジェクト A とオブジェクト C は通常通りにひとつのイメージとして出力される。オブジェクト B は拡大、縮小、移動を自由に行なうことができ、その際全く再描画を必要としない。しかしオブジェクト B がオブジェクト A に重なっている場合には、オブジェクト間に深度の矛盾が生まれてしまう。

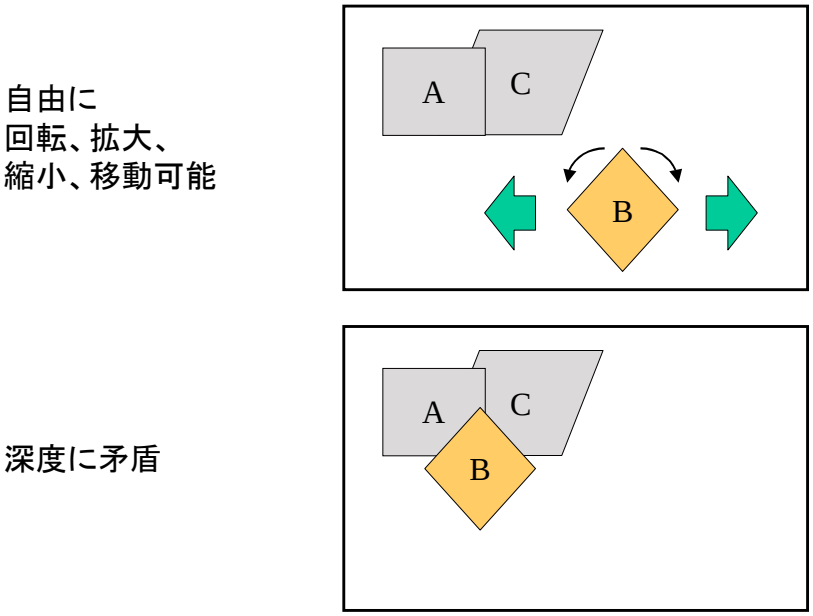

図 8.2　グラフィックスオブジェクトの使用

この解決法は、オブジェクト全てをグラフィックスオブジェクトによって描画することである。グラフィックスオブジェクトは多数定義でき、その深度の順は指定可能である。ゆえに、深度に矛盾をきたさずに 3 次元シーンを描画できる。

2 つめの問題点は、グラフィックスオブジェクトとそのパラメータの設定の数に制限があることである。大規模で複雑なシーンにおいてはグラフィックスオブジェクトの使用数が制限を超えてしまう可能性がある。

3 つめの問題点は、回転が自由に行なえないことである。GBA のハードウェア機能による回転は、2 次元グラフィックスに用いられるものである。3 次元シーンにおける視体積の底面の法線方向に対して以外の回転は、サポートすることができない。

4 つめの問題点は、照光処理を行なう場合に矛盾をきたす恐れがあることである。例えばオブジェクトの表面が鏡面反射をする材質であったなら、少し移動しただけでも矛盾をきたすだろう。その際、グラフィックスオブジェクトのみを再描画したとしても矛盾が解消されるとは限らない。

最後の問題点は、OpenGL の規格を逸脱することである。グラフィックスオブジェクトの使用を明示するコマンドは、OpenGL には無い。

8.3 アルファブレンディング

3 次元グラフィックスではアルファ値を用いて半透明処理が行なわれる。GBA はハードウェア機能でアルファブレンディングをサポートしているので、それを適用することが可能である。しかし、それは問題点を伴う。

GBA のハードウェア機能を用いたアルファブレンディングは、グラフィックスオブジェクトと背景画像間でのみ行なうことができる。しかし、グラフィックスオブジェクト間でアルファブレンディングを用いることはできない。また、個々のグラフィックスオブジェクトに対してアルファブレンディング使用の可否を設定することが可能である。しかし、アルファブレンディングに対するパラメータは個々に設定することができない。このようにGBA のハードウェア機能を用いたアルファブレンディングには制約が多く、使用可能な場面が限られる。

8.4 フェードイン／フェードアウト

フェードイン／フェードアウトは、具体的には輝度変化機能である。3 次元グラフィックスにおける照光処理は、光を発するライトの位置とその色の成分（RGB）を設定し、その成分ごとにオブジェクトの表面に輝度の変化を与える。GBA のハードウェア機能の輝度変化機能を用いることは可能である。しかし、それは問題点を伴う。

問題点の 1 つは、GBA のハードウェア機能による輝度変化は、背景画像もしくはグラフィックスオブジェクトの一種類についてのみ、行なうことができるということである。また、アルファブレンディングと同様に、グラフィックスオブジェクトの個々に対して設定することはできない。

1 つめの問題点は、アルファブレンディングと同時に使用することができないことである。これはハードウェアの仕様に因るところである。

8.5 実現可能性

これまでに述べた GBA のハードウェア機能を用いた高速化手法を、実際に適応するべきか議論する。

8.1節で述べた、イメージの回転、拡大、縮小、移動は、実現することが可能である。この場合、適応可能なビューポート変換のみに対しハードウェア機能を適用し、適用できないビューポート変換に関しては通常どおり GL に処理させる。しかし、アプリケーションが個々のオブジェクトに別々のビューポート変換を指定した場合には適用することはできない。よって、このハードウェア機能適用の可否を設定できるように API を拡張し、アプリケーションの開発者が明示的に利用することが必要である。

8.2節で述べた、グラフィックスオブジェクトの利用も、実現可能である。アプリケーションの開発者は、この機能を適用したいオブジェクトを指定し、そのオブジェクトは GL によって処理されイメージが生成され、グラフィックスオブジェクトのメモリ領域に格納される。そして、アプリケーションによりそのグラフィックスオブジェクトは操作される。このように、オブジェクトの指定、グラフィックスオブジェクトの操作に関する API を拡張し、アプリケーションの開発者が自由に利用できるように実現されうる。しかし、8.2節で述べた問題点があるため、アプリケーションの開発者はそれに気をつけて利用しなければならない。

8.3節、8.4節で述べたアルファブレンディングとフェードイン／フェードアウト機能を利用した高速化を実現することは難しい。なぜなら、2 機能とも適応できるオブジェクトが少なすぎるため、実用には向かない。また、双方の機能のうちの片方しか用いることができないため、利用機会が限られてしまう。また、効果も仕様により OpenGL ほど細かく設定できない。

したがって、アプリケーションの開発者の明示的な指定による、グラフィックスオブジェクトもしくはイメージの回転、拡大、縮小、移動による、GL の処理の高速化を図るべきである。これは今後実装する予定とする。

第9章　おわりに

本研究では、GBA 上で動作する OpenGL 準拠の 3 次元グラフィックスライブラリを開発した。GBA は世界で出荷量の一番多い次世代携帯型ゲーム機であり、幅広く普及している。その GBA 上で動作する 3 次元グラフィックスライブラリを開発することで、GBA の用途を広げることができた。

OpenGL 準拠のグラフィックスライブラリはモデリング、レンダリングの処理をし、ハードウェア、ウィンドウシステムに依存した部分は GLX のようなウィンドウシステム機能拡張ライブラリとして分離されている。そのハードウェア、ウィンドウシステムに依存した部分を GBA 用に GBAGL として実装し、GBA 上で 3 次元グラフィックスライブラリを利用することができた。その API は GLX と同様にすることで、GLX を使用していたアプリケーションの開発者は容易に GBA 用のアプリケーションを開発することができるようになった。

GBGAGL の実装は TinyGL を参考に行なった。TinyGL は軽量化、高速化された OpenGL 準拠ライブラリである。GLX は X Window System で動作させるためのウィンドウシステム機能拡張ライブラリであるため、X プロトコルに関する部分が多い。GBAGL を GLX の API をそのままに、不要な部分を排除して実装した。また、GBA の開発環境には標準ライブラリが無いので、GL で使用される標準ライブラリ関数を実装するなど、GBA に必要な処理を組み込んだ。

GBA は携帯端末の中では高性能であるが、PC などと比べると貧弱な部分があり、それは具体的には解像度が低いこと、処理能力が低いこと、RAM が少ないことであった。そのような制限を、GBA の機能で補うことが必要である。GBA は携帯型ゲーム機であり、それには 2 次元グラフィックスに特化したハードウェアサポートがある。実装された GBAGL を用いて、評価としてデモプログラムを動作させたが、動作速度に改良の余地が見られた。今後はこれを改良し、GBA のハードウェア機能を用いた高速化をする予定である。

謝辞

本研究の発足、進行にあたり、指導教官である情報システム学研究科の多田好克助教授、同安田絹子助手、中村嘉志元助手には多大なご指導、ご助言をいただきました。また本論文の作成にあたり、草稿段階からのご指導をいただきました。こころより厚く御礼申し上げます。

研究進捗会において数々のご意見をいただいた情報システム学研究科ソフトウェア生産管理学講座の皆様にも、深く感謝いたします。

参考文献

[1] Patricia McLendon, Graphics Library Programming Guide, Silicon Graphics Computer Systems（1991）.

[2] Microsoft Corporation, DirectX Home Page, http://www.microsoft.com/windows/directx/

[3] Apple Computer Inc., 3D Graphics Programming with QuickDraw 3D 1.5.4, http://developer.apple.com/techpubs/quicktime/qtdevdocs/QD3D/qd3d_book.htm（1997）

[4] Mark Seagal and Kurt Akeley, The OpenGL Graphics System: A Specification. Technical Report, Silicon Graphics Computer Systems（1992）.

[5] Norman Chin, Chris Frazier, Paul Ho, Zicheng Liu, Kevin P. Smith and Jon Leech, The OpenGL Graphics System Utility Library（Version 1.3）, ftp://ftp.sgi.com/opengl/doc/opengl1.2/glu1.3.pdf（1998）

[6] Paula Womack and Jon Leech, OpenGL Graphics with the X Window System（Version 1.3）, ftp://ftp.sgi.com/opengl/doc/opengl1.2/glx1.3.pdf（1998）.

[7] Mark J. Kilgard, The OpenGL Utility Toolkit （GLUT） Programming Interface API Version 3, http://www.opengl.org/developers/documentation/glut/glut-3.spec.pdf（1996）.

[8] Brian Paul, The Mesa 3D Graphics Library, http://www.Mesa3d.org/

[9] Fabrice Bellard, Fabrice Bellard's Project Page, http://fabrice.bellard.free.fr/

[10] Philippe Dax, VREng Home Page, http://www.infres.enst.fr/net/vreng/html/index.html

[11] Michael J. Sherman and Bryan S. Ware, Extending Simulation Interfaces to Mobile Computing Platforms, Proceedings of I/ITSEC 2000（2000）.

[12] Paul S. Heckbert and Michael Garland, Multiresolution Modeling for Fast Rendering, Proceeding of Graphics Interface ’94, pp.43-50（1994）.

[13] Jürgen Döllner and Klaus Hinrichs, The design of a 3D Rendering Meta System, Proceedings of Eurographics Workshop on Programming Paradigms for Graphics ’97（1997）.

[14] Mark Seagal and Kurt Akeley, The design of the OpenGL Graphics Interface, Silicon

Graphics Computer Systems, Technical Report（1994）.

[15] Dirk Bartz, Michael Meißner, Tobias Hüttner, Extending Graphics Hardware for Occlusion Queries in OpenGL, SIGGRAPH Workshop on Graphics Hardware（1998）.

[16] Steven Molnar, John Eyles and John Poulton, PixelFlow: High-Speed Rendering Using Image Composition, Computer Graphics（Proceeding of ACM SIGGRAPH 92）, pp.231-240（1992）.

[17] 小関一正, 安田絹子, 多田好克, 携帯型ゲーム機上の 3 次元描画ライブラリ, 第 44 回プログラミング・シンポジウム報告集, pp.47-54（2003）.

[18] OpenGL Architecture Review Board, OpenGL リファレンスマニュアル（第 2 版）, ピアソン・エデュケーション （1999）.

[19] OpenGL Architecture Review Board, OpenGL プログラミングガイド（第 2 版）, ピアソン・エデュケーション （1997）.

[20] Mark J. Kilgard, OpenGL Programming for the X Window System, アジソン・ウェスレイ・パブリッシャーズ・ジャパン （1997）.

[21] Oliver Jones, X Window ハンドブック, アスキー（1990）.

[22] 柴山守, X11 による画像処理基礎プログラミング, 技術評論社 （1994）.

[23] 中嶋正之, 川合慧, グラフィクスとマンマシンシステム, 岩波書店（1995）.

[24] 山口富士夫, 実践コンピュータグラフィックス-基礎手続きと応用-, 日刊工業新聞社（1987）.

付録A　OpenGL処理パイプライン

ここではOpenGLの処理パイプラインについて詳しく説明する。図 9.1は処理パイプラインをさらに詳しく示したブロックダイアグラムである。四角はOpenGLの処理、角の無い四角はメモリ上に情報を保存しているバッファを表している。上から入ったデータは主に下に向かって処理される。主要な入力データは頂点とカラー指標、テクスチャ座標である。頂点の集合はプリミティブとなり、フラグメントを経て、最終的にはピクセルになる。本節ではこの流れについて詳しく説明する。

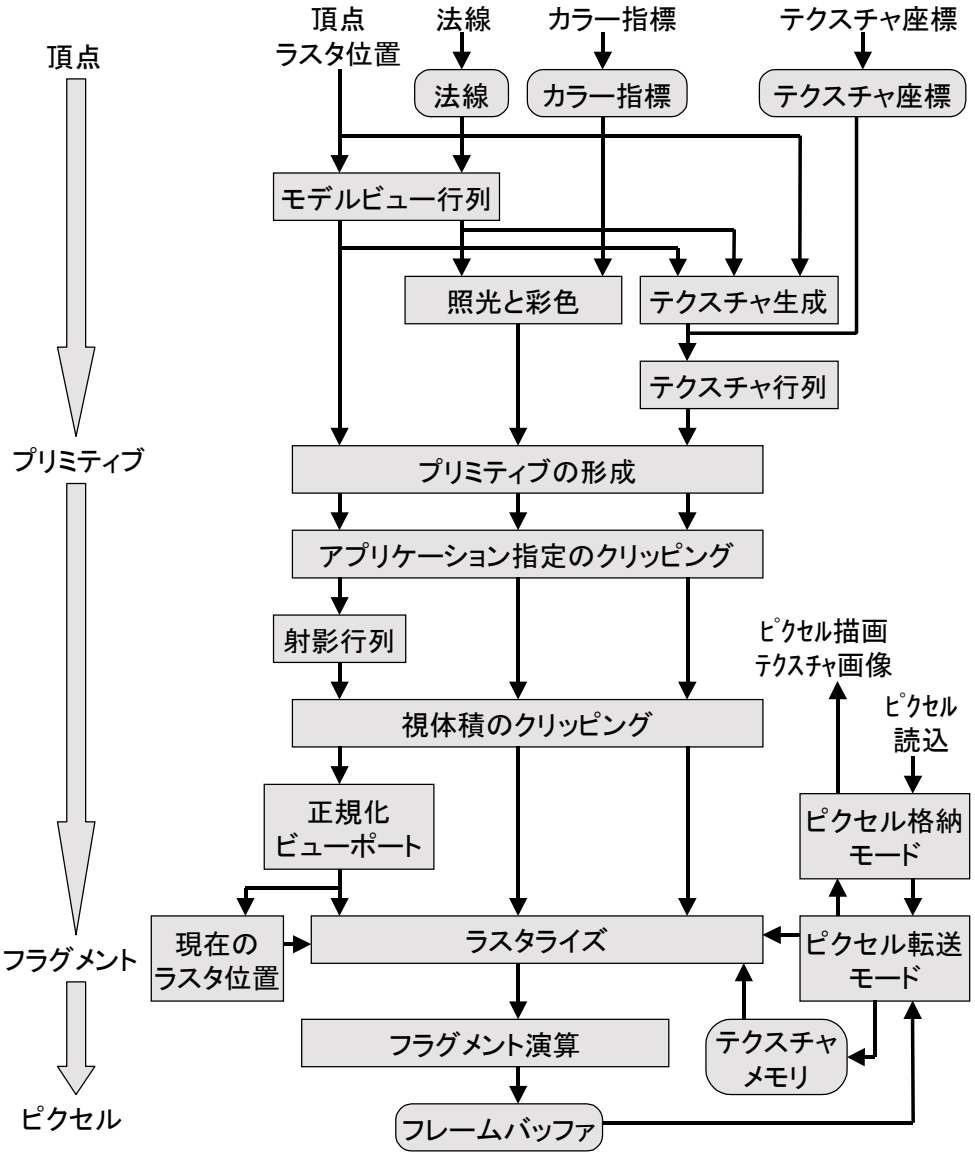

図 9.1　OpenGL 処理パイプラインの詳細

A.1　入力データ

OpenGL パイプラインには以下の入力データが必要である。

- **頂点**

  幾何学オブジェクトの形状を定義する。

- **法線**

  特定の頂点に関する法線ベクトル。その頂点がどのように 3 次元空間に配置されるかを決定する。また、頂点が受ける光量に影響する。

- **カラー指標**

  色を設定する。最終的な色は光線の状態により決定される。

- **テクスチャ座標**

  オブジェクトの頂点に関するテクスチャマッピング内の位置を決定する。

頂点以外の入力データである法線、カラー、テクスチャ座標は状態として保存される。そして、頂点が定義されたとき、それらは継承される。

A.2　モデルビュー変換と行列

頂点と法線はイメージの生成に使用される前に、モデルビュー行列により変換される。モデルビュー行列とは、点、線分、ポリゴンといったプリミティブとラスタ位置をオブジェクト座標から眼点座標に変換する、4×4 の行列である。モデルビュー行列は視界変換とモデリング変換によって決定される。視界変換はカメラの配置、方向設定のように、視界を変換する。モデリング変換はモデルの配置と方向設定に使用される。モデリング変換は具体的には回転、平行移動、拡大縮小などである。視界変換とモデリング変換は密接に関係しており、両者を組み合わせてモデルビュー行列を生成する。

A.3　照光と彩色

カラーや法線ベクトル設定に加えて、照光の状態、オブジェクトの材質の特性を設定できる。OpenGL には 2 つの影付けモデルがある。スムースシェーディングではプリミティブをラスタライズする際、ピクセルフラグメントのそれぞれに異なるカラーを割り当てて頂点のカラーを計算する。フラットシェーディングでは、プリミティブの１つの頂点に対してカラーを計算し、そのプリミティブをラスタライズして生成した全てのピクセルフラグメントにそのカラーを割り当てる。

A.4　テクスチャ座標の生成

テクスチャ座標とは、プリミティブの頂点に対してテクスチャメモリ内のテクセルをどのように割り当てるかを示すものである。テクスチャ座標は直接設定することもできるが、頂点データから生成することも可能である。テクスチャ座標は設定または生成された後に、テクスチャ行列に変換される。

A.5　プリミティブの形成

頂点は、点、線分、ポリゴンといったプリミティブに組み立てられる。このとき、エッジフラグ（ポリゴンの辺の扱いを設定)、カラー、テクスチャ情報が考慮される。

A.6　クリッピングと射影行列

クリッピングとは、クリップ平面が定義する半空間の外側に位置する幾何学プリミティブの一部を排除することである。クリッピングによって、半空間の外側にある線分、ポリゴンの一部または全部は排除され、必要に応じて頂点が追加生成されて、半空間の内側においてプリミティブが完成する。

まず、組み立てられたプリミティブは、アプリケーションにより設定された任意のクリップ面においてクリッピングされる。次に射影行列により射影変換されクリップ座標

となり、視体積（view volume）である 6 面によりクリッピングされる。

射影行列とは、視野や視体積を特定し、その中に位置するオブジェクトとそれがどのように見えるかを決定するものである。視体積とはオブジェクトがどのように画面に射影されるかを定義し、どのオブジェクトまたはオブジェクトの一部をクリッピングするかを定義するものであり、図 1.2および図 1.3に例を示す。射影法には透視法射影と正射影があり、図 1.2は透視法射影、図 1.3は正射影における視体積を示している。透視法射影は人間の眼もしくはカメラの特性に類似しており、日常生活において人が物を見るように、オブジェクトが視点から離れれば離れるほど、そのオブジェクトは最終的なイメージにおいて小さく表示される。透視法射影における視体積は四角錐台となる。一方、正射影は相対的なサイズに影響することなく最終的なイメージにオブジェクトを直接投影する方法で、外観よりもオブジェクトの正確な寸法が要求されるような建築用、CAD などのアプリケーションにおいて使用される。正射影における視体積は直方体となる。

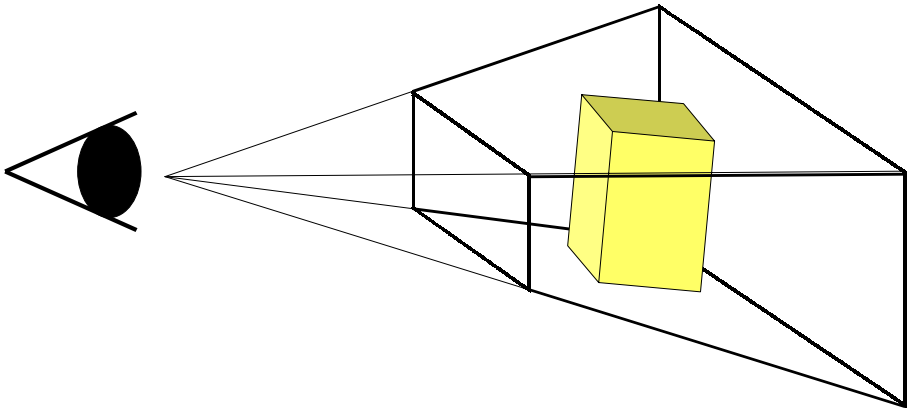

図 1.2　透視法射影の視体積

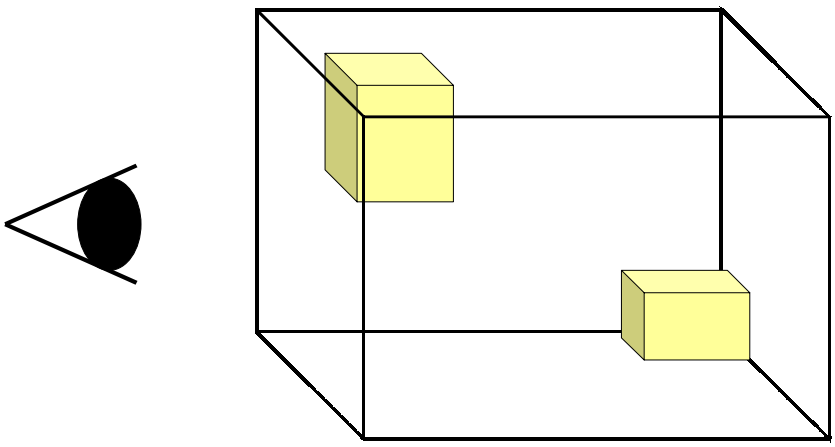

図 1.3　正射影の視体積

A.7　ウィンドウ座標への変換

ウィンドウ座標に変換される前に、クリップ座標は正規化のため値で除算され正規化装置座標となる。次に正規化装置座標に対してビューポート変換を行い、イメージのピクセルに対応した座標であるウィンドウ座標にする。

A.8　ラスタライズ

プリミティブはラスタライズされることにより 2 次元の画像に変換される。その画像の各点はカラー、深度、テクスチャデータなどの情報を有する。点とそれに関連する情報をまとめてフラグメントと呼ぶ。

ラスタ位置とは、OpenGL の保持する、ウィンドウ座標内の 3 次元の位置であり、ピ

クセルやビットマップの書き込み処理に使用される。ラスタ位置はウィンドウ座標、クリップ座標、カラー情報、テクスチャ座標などで構成される。

A.9　フラグメント演算

OpenGL コマンドによって有効化されているテストがフラグメントに対して行なわれ、それに合格した場合にのみフレームバッファ内の該当ピクセルの値を変更することができる。テストには以下のものがある。

- **ピクセル保有テスト**

  ピクセル保有テストは、OpenGL のグラフィックコンテキストが対象のフラグメントに対応しているピクセルをフレームバッファに持っているかどうかを調べる。保有していなければ、フラグメントを廃棄するか、そのフラグメントを利用して新たなフラグメント演算を実行するかをウィンドウシステムが決定する。

- **シザーテスト**

  シザーテストは、設定された方形領域内を、フラグメントによるフレームバッファの対応したピクセルの値の変更を可能にする。

- **アルファテスト**

  アルファテストは、フラグメントのアルファ値と設定された参照値を比較し、その結果によってはフラグメントが破棄される。アルファテストは透明化のアルゴリズムに使用される。通常、アルファ値は不透明度を示す。

- **深度バッファテスト**

  フレームバッファ上の各ピクセルに対し、視点とそのピクセルを有するオブジェクトの距離（深度）が深度バッファに格納されている。深度バッファテストは対象のフラグメントの深度とそれに対応する深度バッファの深度を比較し、その結果によっては

フラグメントが破棄される。通常深度バッファテストは陰面消去に使用される。ピクセルに新規のカラー候補が現れると、対応するオブジェクトが前のオブジェクトよりも近い場合にのみ描画が実行される。このようにして最終的に全景がレンダリングされた後には、他のオブジェクトに隠れていないオブジェクトのカラーのみが残る。

- **ステンシルテスト**

  ステンシル（型紙）テストは、フラグメントによりステンシルバッファの該当箇所を変更する。ステンシルテストの用途の一つにキャッピングがある。例えば 6 つのポリゴンで構成された直方体のオブジェクトがあり、これをクリップ平面が切断しているとする。直方体の内側を見えないようにしたい時、ステンシルテストにより直方体の内側に該当するピクセルの位置をステンシルバッファに格納し、次にその部分のみ描画されるようにポリゴンを生成し描画する。こうすることにより直方体の内側を見えないようにすることができる。

- **混合処理**

  混合処理では対象のフラグメントの R、G、B、アルファ値と、その該当位置に既に保存されているピクセルのそれらの値を組み合わせる。

- **ディザ処理**

  カラービットプレーン数の少ないシステムにおいては、画像のカラーをディザ処理することにより、空間的な解像度を犠牲にしてカラー解像度を改善できる。ディザ処理によって隣接するピクセルに異なるカラー値が割り当てられ、遠くから見るとこれらのカラーは単一の中間色に混合したように見える。この技術はグレー階調を実現したモノクロ印刷で使用するハーフトーン処理に類似している。

- **論理演算**

  対象のフラグメントとそれに対応するフレームバッファ内の値に論理演算がなされ、既存のデータは論理演算結果に置換される。

A.10　フレームバッファ

フラグメントはフレームバッファでピクセルに変換される。フレームバッファはカラー、深度、ステンシル、アキュムレーションの各論理バッファにまとめられる。カラーバッファは左前、右前、左後、右後といくつかの補助バッファで構成される。コマンドによりこれらのバッファを制御する、それらのバッファから直接ピクセルを読み込みコピーするなどが可能である。