3DS

3DS プログラミングマニュアル

グラフィックス基本編

2016-05-10

Version 1.5

目次

コード

表

図

1. はじめに

本ドキュメントは、3DS のグラフィックス機能を利用する上で必要となる、基本的な情報とプログラミング手順について説明したものです。3DS の 3D グラフィックス機能は OpenGL ES 1.1 をベースにしていますので、本ドキュメントを読むにあたっては OpenGL についての知識や行列やベクトルの数学についての知識が必要になります。

注意: 3DS は OpenGL の使用をサポートしていますが、3D グラフィックス機能を多用する場合、そのまま GL 関数を呼び出すよりも、「3DS プログラミングマニュアル – グラフィックス応用編」で説明しているコマンドキャッシュや 3D コマンドによる PICA レジスタへの直接書き込みを処理速度の観点から推奨されるケースが多くあります。グラフィックス応用編ではそのほかの機能についても説明していますので、本ドキュメントに続けて目を通しておいてください。

グラフィックスの描画には、GL、GD、GR と弊社から配布しております NintendoWare for CTR(NW4C)の 4 種類のライブラリが用意されています。

GL は基本的なライブラリに位置づけられ、OpenGL ES に準拠した関数やエラー処理、ステートの差分管理などのリッチな機能が用意されているため処理が重く、パフォーマンスが必要な場合の使用は推奨されていません。

GD は GL よりも軽量化されたライブラリです。GL との互換性はありませんが、GL と同等の機能と開発者が扱いやすい API を提供します。内部処理のため、ある程度の CPU 処理負荷を必要とします。

GR は 3D コマンドの直接生成を支援するためのライブラリです。最も高速に動作しますが、レジスタの設定方法に深い理解が必要で、エラーチェックなども開発者が行わなければなりません。

NW4C では、コマンドキャッシュなどの機能を利用したライブラリをソースコード込みで提供していますので、NW4C の活用についてもご検討ください。NW4C はそのまま使用することで最適化されたパフォーマンスを発揮できるように実装されていますが、独自の処理を行いたい場合は GR や GD ライブラリと組み合わせて使用することができます。

補足: アプリケーションからグラフィックスライブラリに渡されるメモリ領域には、アライメントとサイズに制約があります。プログラミングマニュアル中では、アライメントとサイズの制約は数値で記載していますが、それぞれ定数が定義されています。定義されている定数については、APIリファレンスを参照してください。

1.1. 章の構成

「2. GPU」では、3DS に搭載されている GPU の概要と機能を紹介し、グラフィックス処理がどのような順序で行われるのかを説明します。ここで、3DS のグラフィックスで表現できることと、グラフィックス処理の全体像について把握してください。

「3. LCD」では、3DS に搭載されている LCD と GX ライブラリとの関係を説明します。ここで、GX ライブラリと LCD 表示が密接に関係していることを理解してください。

「4. コマンドリスト」では、3D グラフィックス処理のコマンドを実行するために必要なコマンドリストについて説明します。

「5. シェーダプログラム」では、シェーダプログラムの種類と使用方法について説明します。

「6. 頂点バッファ」では、頂点データをまとめて頂点シェーダに入力する頂点バッファの作成方法と使用方法について説明します。

「7. テクスチャ」では、3DS が扱うことのできるテクスチャイメージの種類と、ネイティブフォーマットについて説明します。

「8. 頂点シェーダ」では、頂点データの入力と以降の処理で使用する頂点属性の出力について説明します。

「9. ジオメトリシェーダ」では、SDK で用意されている、ポイントやラインの基本的な図形を生成するシェーダ、シルエットやサ

ブディビジョンのように便利なシェーダの概要とその使用方法について説明します。

「10. ラスタライズ」では、フラグメント処理の直前に行われる処理について説明します。3DS ではシザーテストがこの段階で行われています。

「11. テクスチャ処理」では、テクスチャユニット、コンバイナ、コンバイナバッファによるテクセル色の決定やライトとカラーの合成、プロシージャルテクスチャの作成方法について説明します。

「12. 予約フラグメントシェーダ」では、フラグメントライティングやシャドウ、フォグ、ガスレンダリングなど、予約ユニフォームによって制御することのできるフラグメント処理について説明します。

「13. パーフラグメントオペレーション」では、フラグメントがレンダーバッファに書き込まれるまでに行われる処理について説明しています。アルファテストなどのテスト、ブレンディングについてはここを参照してください。

「14. フレームバッファオペレーション」では、フレームバッファに対する処理とバッファクリアについて説明しています。

「15. その他」では、OpenGL ES の仕様との相違点などをまとめています。

> **補足:** 立体視表示、ブロックモード、アーリーデプステストなどの特殊な設定を必要とする機能や、照明モデルの実装例、グラフィックスコマンドのキャッシング機能、PICA レジスタの情報は「3DS プログラミングマニュアル – グラフィックス応用編」で紹介しています。

1.2. グラフィックス処理のエラーについて

3DS の 3D グラフィックス機能は OpenGL ES をベースにしているため、3D グラフィックス処理の関数呼び出しで発生したエラー時の挙動は、OpenGL ES と同じ仕様になっています。

- GL_OUT_OF_MEMORY のエラーを除いて、該当の関数の処理は無視されます。
- エラーコードは glGetError() で読み取ることができます。
- エラーコードは 1 つしか記録されません。glGetError() で読み取るまで、次のエラーが生成されてもエラーコードは更新されません。

記録されるエラーコードには、以下のものがあります。

表 1-1. エラーコード一覧

エラーコード	エラーが生成される条件
GL_INVALID_ENUM	GLenum 型の引数に範囲外の値を指定した場合
GL_INVALID_VALUE	数値型の引数に範囲外の値を指定した場合など
GL_INVALID_OPERATION	その段階で呼び出すことのできない関数を呼び出した場合など
GL_OUT_OF_MEMORY	関数の実行に必要なメモリが足りない場合

1.3. シングルスレッドモデル

3D グラフィックス機能のライブラリは、単一のスレッドから呼び出されることを想定して設計されています。そのため、複数のスレッドから関数が呼び出されたときの動作は保証されません。また、関数の呼び出しは単一のレンダリングコンテキストに対してのみ行われなければなりません。

2. GPU

3DS の GPU には、DMP 社開発の PICA グラフィックスコア(268 MHz)を搭載しています。フレームバッファアーキテクチャを採用しており、DS のときのような **OBJ、BG という概念はありません**。また、VSync と密接に連動していた DS の 3D エンジンと違い、バッファを切り替えるまでレンダリングを行うことができます。つまり、フレームレートとレンダリングの質をトレードオフの関係で調節することができます。

3D グラフィックス機能は OpenGL ES 1.1(一部機能は OpenGL ES 2.0 相当)をベースにしており、3D グラフィックスでよく使用される機能はハードウェアで搭載されています。シェーダプログラムに対応しており、プログラマブルな頂点処理シェーダと、非プログラマブルなピクセル処理シェーダを備えています。

> **補足:** GPU を単純に比較すると、頂点処理やフィルレートといったスペックは Wii やゲームキューブには及びません。しかしハードウェア搭載された機能で、シーンによっては Wii やゲームキューブよりもリッチな映像を表現することができます。

2.1. ハードウェアで搭載されている機能

OpenGL ES 標準には規定されていないものを含め、よく使用されるグラフィックス機能をハードウェアで搭載することにより、少ない処理ステップ数で多彩な表現を可能としています。

提供する機能には以下のものがあります。

- シルエット生成
- パーティクル生成
- プロシージャルテクスチャ
- フラグメントライティング
- ガスレンダリング
- セルフシャドウ、ソフトシャドウ
- ポリゴンサブディビジョン

これらの機能は、シェーダプログラムや予約されたユニフォームの設定によってグラフィックス処理に利用することができます。

2.1.1. シルエット生成

シルエット生成は、オブジェクトのエッジを GPU が検出し、エッジのみを描画する機能です。選択したオブジェクトを強調するために輪郭を縁取る場合やソフトシャドウの描画に利用することができます。

2.1.2. パーティクル生成

パーティクル生成は、大量のパーティクルをランダムに描画する機能です。爆発や雪など、エフェクト処理の描画に利用することができます。

2.1.3. プロシージャルテクスチャ

プロシージャルテクスチャは、ノイズと幾何学模様を組み合わせたようなテクスチャを自動的に生成する機能です。幾何学模様のように規則的なパターンや、木目や大理石のようにランダム性を持ちながらも規則的なパターンのテクスチャをローコストで高速に作成することができます。

2.1.4. フラグメントライティング

3DS のライティング処理はフラグメント単位で行われるフラグメントライティングです。バンプマップ、環境マップの適用や、ライトの計算を高速に行うことができます。

2.1.5. ガスレンダリング

ガスレンダリングは、ポリゴンオブジェクトとの交差や前後関係を考慮しながらガス状物体を描画する機能です。パーティクルなどで描画されるガス状物体に比べ、ほかのオブジェクトとの境界面をより自然に描画することができます。

2.1.6. セルフシャドウ、ソフトシャドウ

3DS のシャドウ機能はセルフシャドウおよびソフトシャドウに対応しています。オブジェクト自身に落ちる影や、より自然で柔らかな輪郭の影を描画することができます。

2.1.7. ポリゴンサブディビジョン

ポリゴンサブディビジョンは、決められた法則で入力された頂点群を GPU が自動的に分割し、ポリゴンを滑らかにする機能です。数少ない頂点の入力で、滑らかな曲面を持つ物体を描画することができます。

2.2. レンダリングパイプライン

3DS の 3D グラフィックス処理は、レンダリングパイプラインと呼ばれる一連の処理に沿って行われます。

レンダリングパイプラインの概略図を図 2-1 に、各プロセスで行われる処理を表 2-1 に、それぞれ示します。処理の内容については関連するページを参照してください。

図 2-1. レンダリングパイプラインの概略図

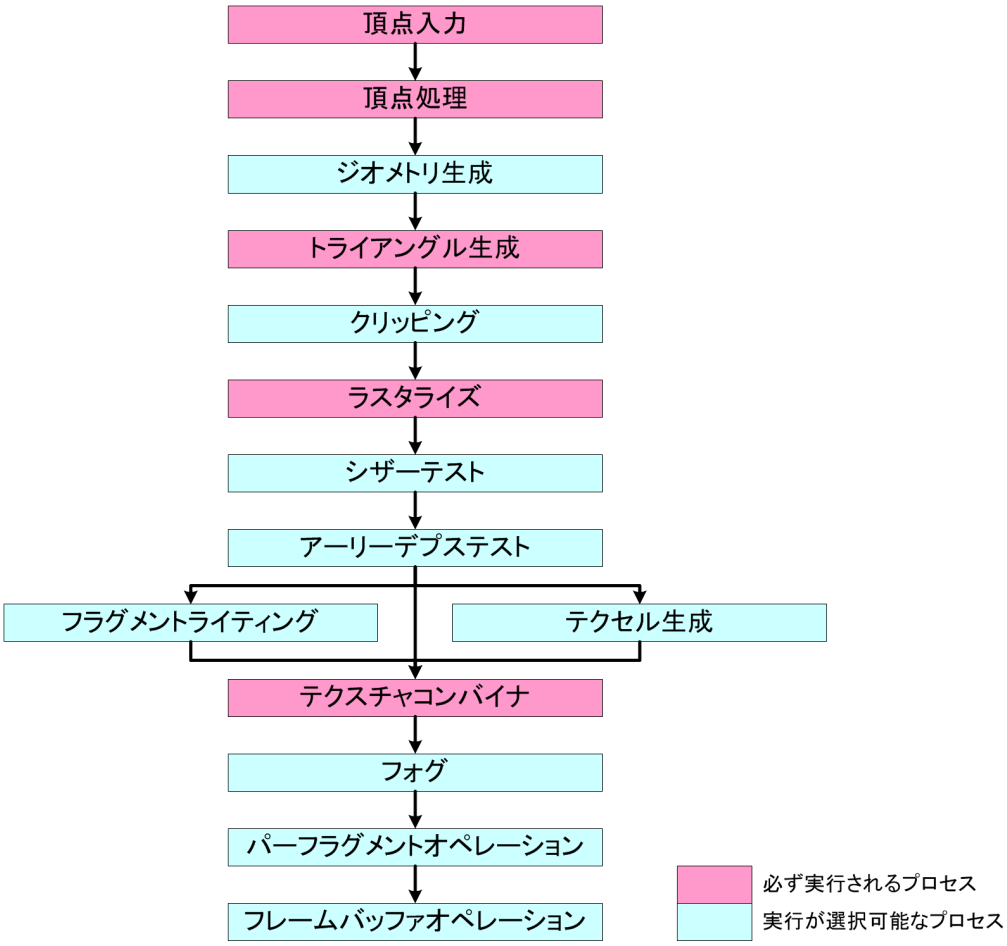


表 2-1. 各プロセスで行われる処理

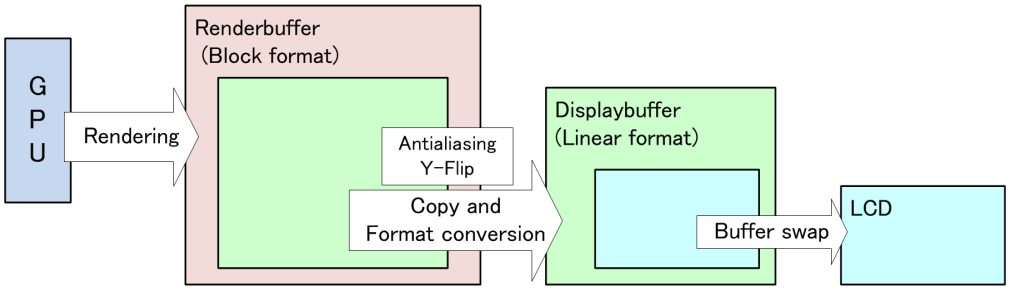
プロセス	処理内容
頂点入力	アプリケーションから入力された頂点データを処理する
頂点処理	頂点シェーダを実行する
ジオメトリ生成	ジオメトリシェーダを実行する
トライアングル生成 (トライアングルセットアップ)	プリミティブをすべて三角形プリミティブに変換する
クリッピング	クリップボリュームによるプリミティブのクリップ処理を行う
ラスタライズ	プリミティブをフラグメントに変換する
シザーテスト	シザリングで余分なフラグメントを削減する
アーリーデプステスト	フラグメント処理の前にデプステストを行う

フラグメントライティング	フラグメント単位のライティング処理を行う
テクセル生成	テクスチャ座標からテクセルの色を決定する
テクスチャコンバイナ	フラグメントライトのカラーやテクスチャカラーなどを合成する
フォグ	フォグおよびガスの描画を行う
パーフラグメントオペレーション	フラグメントに対して、テスト、ブレンディング、論理演算などの処理を行う
フレームバッファオペレーション	フレームバッファからのコピー、ピクセルリード処理などを行う

3. LCD

3DS では、LCD の表示と GX(グラフィックス)ライブラリが密接に関係しています。レンダリングから LCD に表示されるまでの流れは、図 3-1 になります。

図 3-1. レンダリングから LCD に表示されるまでの流れ



工程を大まかに分けると、以下のようになります。

1. レンダリング
2. レンダーバッファからディスプレイバッファへのコピーとフォーマット変換
3. バッファスワップによる LCD に表示される領域の更新

1. のレンダリングについては「4. コマンドリスト」以降の章で説明します。この章では、出力先となる LCD の仕様と、GX ライブラリを使用する際に必要な初期化処理を最初に説明します。続いて 2. と 3. について、バッファの確保・転送、画面更新の同期、終了処理を順を追って説明します。

> **補足:** 立体視表示については「3DS プログラミングマニュアル - グラフィックス応用編」を参照してください。

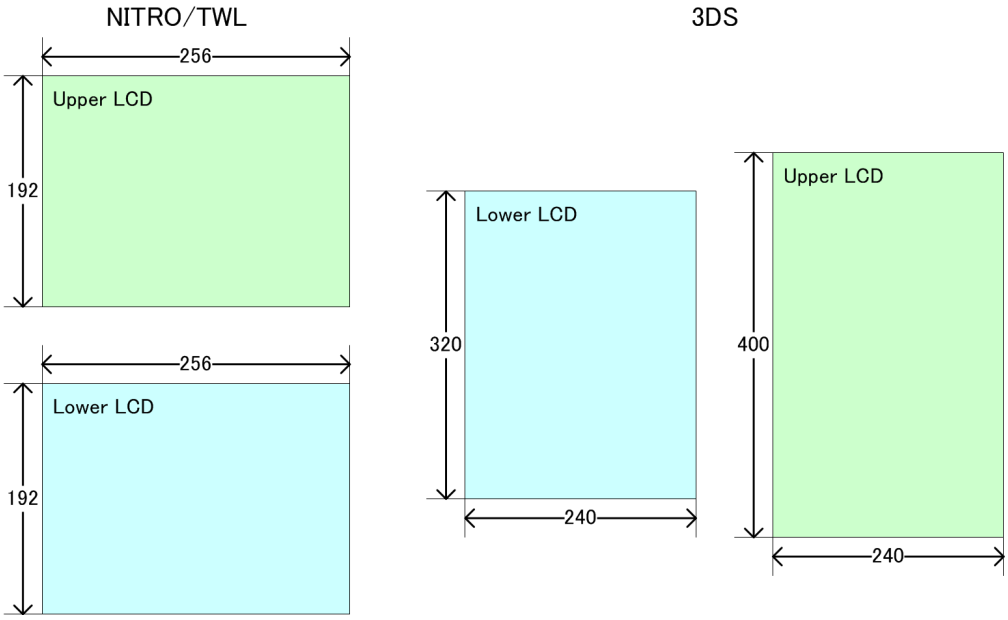
3.1. LCD の解像度、配置方向について

3DS の LCD は、NITRO/TWL の LCD と解像度および配置の方向が異なっています。解像度の違いを表 3-1 に、配置方向の違いを図 3-2 に示します。

表 3-1. LCD の解像度

LCD	NITRO/TWL	3DS
上画面 LCD(幅×高さ)	256 × 192	240 × 400
下画面 LCD(幅×高さ)	256 × 192	240 × 320

図 3-2. LCD の配置方向



NITRO/TWL では長辺が横方向(画面幅)、下画面を手前にしたときが正位置になるように配置されていましたが、3DS では短辺が横方向、下画面を上画面の左側に位置したとき(コントローラの正面から右に 90 度回り込んだ状態)が正位置になるように配置されています。

3.2. 必要な初期化処理

3DS ではフレームバッファアーキテクチャを採用しています。フレームバッファの内容をもとに LCD の表示が行われるため、最初に GX ライブラリの初期化を行わなければなりません。

3.2.1. GX ライブラリの初期化

GX ライブラリの初期化は `nngxInitialize()` を呼び出して行います。

コード 3-1. GX ライブラリの初期化関数

```
GLboolean nngxInitialize(
        GLvoid* (*allocator)(GLenum, GLenum, GLuint, GLsizei),
        void (deallocator)(GLenum, GLenum, GLuint, GLvoid));
GLboolean nngxGetIsInitialized();
```

`nngxInitialize()` では、ディスプレイバッファなどのメモリ領域を確保する際に呼び出されるアロケータと、そのメモリ領域を解放する際に呼び出されるデアロケータを *allocator* と *deallocator* に指定します。アロケータとデアロケータについての詳細は「3.2.1.1. アロケータ」、「3.2.1.2. デアロケータ」を参照してください。

初期化に成功した場合は `GL_TRUE` を、失敗した場合は `GL_FALSE` を返します。初期化後に `nngxFinalize()` で終了処理を行わずに、再度この関数を呼び出した場合は `GL_FALSE` を返します。この関数で初期化を行う前に他の `gl`、`nngx` 関数を呼び出した場合の動作は保証されていません。また、初期化でデフォルトのレンダーバッファなどは確保されません。初期化後にアプリケーションで確保する必要があります。

`nngxGetIsInitialized()` は、`nngxInitialize()` によって GX ライブラリが初期化済みならば `GL_TRUE` を返します。

3.2.1.1. アロケータ

アロケータの第 1 引数には、どのメモリから領域を確保するかが渡されます。渡される値の種類には、以下のものがあります。

- NN_GX_MEM_FCRAM
  メインメモリ(デバイスメモリ)から確保します。
- NN_GX_MEM_VRAMA
  VRAM-A から確保します。
- NN_GX_MEM_VRAMB
  VRAM-B から確保します。

NN_GX_MEM_FCRAM が渡された場合、メインメモリから確保しますが、その領域はデバイスメモリ上でなければなりません。デバイスメモリとは、周辺デバイスからのアクセスの際にアドレスの整合性が保証されているメモリ領域のことです。メモリの管理はアプリケーションで行わなければなりません。デバイスメモリとして確保したメモリ領域の先頭アドレスとそのサイズは nn::os::GetDeviceMemoryAddress() と、nn::os::GetDeviceMemorySize() で取得することができます。

> **補足:** デバイスメモリについての詳細は「3DS プログラミングマニュアル – システム編」を参照してください。

NN_GX_MEM_VRAMA または NN_GX_MEM_VRAMB が渡された場合、VRAM の開始、終了アドレスやサイズを nn::gx::GetVramStartAddr()、nn::gx::GetVramEndAddr()、nn::gx::GetVramSize() でそれぞれ取得し、メインメモリと同様にアプリケーションでメモリ管理を行ってください。

第 2 引数には確保するメモリ領域の使用目的(バッファ種別)が渡されます。種別によってアドレスのアライメントが異なりますので、アプリケーションは以下の規定に従って、アロケータを実装してください。

表 3-2. バッファ種別によるアライメントの違い

渡される値	バッファ種別	アライメント
NN_GX_MEM_TEXTURE	テクスチャ (2D、環境マップ)	全フォーマットとも 128 Byte
NN_GX_MEM_VERTEXBUFFER	頂点バッファ	頂点属性によって異なります。 4 Byte(GLfloat 型) 2 Byte(GLshort 型、GLushort 型) 1 Byte(GLbyte 型、GLubyte 型)
NN_GX_MEM_RENDERBUFFER	カラーバッファ	64 Byte
	デプスバッファ (ステンシルバッファ)	32 Byte(16 bit デプスの場合) 96 Byte(24 bit デプスの場合) 64 Byte(24 bit デプス+8 bit ステンシルの場合)
NN_GX_MEM_DISPLAYBUFFER	ディスプレイバッファ	16 Byte
NN_GX_MEM_COMMANDBUFFER	3D コマンドバッファ	16 Byte
NN_GX_MEM_SYSTEM	システム用バッファ	4 Byte(確保サイズが 4 の倍数) 2 Byte(確保サイズが 4 の倍数ではない 2 の倍数) 1 Byte(上記以外)

上記の規定に加えて、以下のハードウェア仕様にも従って実装してください。

- キューブマップテクスチャの 6 面すべてが同じ 32 MByte 境界内に収まっていなければなりません。
- キューブマップテクスチャの 6 面それぞれのアドレス上位 7 bit が共通の値でなければなりません。
- キューブマップテクスチャでは、GL_TEXTURE_CUBE_MAP_POSITIVE_X の面のアドレスが、すべての面のアドレス

のうちで一番小さい値または等しい値でなければなりません。
- VRAM-A と VRAM-B をまたいで配置してはいけません。

> **注意:** ディスプレイバッファは VRAM-A/B の最後尾から 1.5 MByte には確保しないでください。
>
> テクスチャアドレスのアライメントが正しく設定されていない場合、GPU がハングアップしたり、描画結果が壊れたりするなどの現象が発生する可能性があります。

第 3 引数には、第 2 引数で NN_GX_MEM_VERTEXBUFFER、NN_GX_MEM_RENDERBUFFER、NN_GX_MEM_DISPLAYBUFFER、NN_GX_MEM_COMMANDBUFFER が渡された場合に、それぞれのオブジェクトの名前(ID)が渡されます。

第 4 引数には、確保する領域のサイズが渡されます。

アプリケーションはこれらの引数を基に、アドレスのアライメントを考慮して指定されたメモリ領域を確保し、戻り値としてメモリ領域の先頭アドレスを返してください。領域の確保に失敗した場合は 0 を返してください。

3.2.1.2. デアロケータ

第 1 から第 3 までの引数に渡される値は、メモリ領域の確保で渡された値と同じです。第 4 引数にはメモリ領域の先頭アドレスが渡されます。アプリケーションはこれらの引数を基に、アロケータで確保したメモリ領域を解放してください。

3.2.1.3. アロケータの取得

nngxGetAllocator() を呼び出して、GX ライブラリの初期化時に設定したアロケータを取得することができます。

コード 3-2. アロケータの取得

```
void nngxGetAllocator (
                GLvoid* (**allocator)(GLenum, GLenum, GLuint, GLsizei),
                void (*deallocator)(GLenum, GLenum, GLuint, GLvoid));
```

allocator および *deallocator* には、アロケータおよびデアロケータへのポインタを受け取るポインタを指定します。それぞれの引数に NULL を指定すると、アロケータまたはデアロケータを取得しません。

3.2.1.4. 初期化用レジスタ設定コマンドの取得

nngxGetInitializationCommand() を呼び出して、nngxInitialize() を呼び出したときに実行される初期化用のレジスタ設定コマンドを取得することができます。

> **補足:** この関数は、SDK がサポートしているグラフィックスライブラリを利用せず、直接レジスタへの設定コマンドを作成しているような場合に、HOME メニューなどからアプリケーションに復帰した際に生じる描画関連の問題への対策用に追加されました。そのため、通常は使用する必要はありません。

コード 3-3. 初期化用レジスタ設定コマンドの取得

```
GLsizei nngxGetInitializationCommand(GLsizei datasize, GLvoid* data);
```

data には、レジスタ設定コマンドを受け取るコマンドバッファへのポインタを指定します。*datasize* には、*data* が指すバッファのバイトサイズを指定します。

この関数は、初期化用のレジスタ設定コマンドのバイトサイズを返します。*data* に 0(NULL)を指定すると、レジスタ設定コマンドの取得は行われませんが、確保すべきバッファサイズを返します。一度 *data* に 0 を指定したあと、そのサイズのバッ

ファを用意して、再度呼び出してコマンドを取得してください。

なお、nngxInitialize() で初期化を行う前に呼び出した場合は、コマンドは取得されず、0 を返します。

表 3-3. nngxGetInitializationCommand() が生成するエラー

エラー	原因
GL_ERROR_80B4_DMP	*datasize* に指定した値が、取得されるレジスタ設定コマンドのバイトサイズより小さい

3.2.2. コマンドリストオブジェクトの作成

GX ライブラリの初期化後に行わなければならないのは、コマンドリストオブジェクトの作成です。コマンドリストオブジェクトは 3DS が独自に導入したもので、このオブジェクトを 3D グラフィックス処理の実行単位として扱います。ここでは、その作成方法のみを紹介します。コマンドリスト(オブジェクト)の詳細については「4. コマンドリスト」を参照してください。

まず、nngxGenCmdlists() でコマンドリストオブジェクトを生成し、生成したコマンドリストオブジェクトを nngxBindCmdlist() で GPU にバインド(関連付け)したあと、nngxCmdlistStorage() でメモリ領域の確保を行います。

コマンドリストオブジェクトは 3D コマンドバッファとコマンドリクエストで構成されており、nngxCmdlistStorage() の *bufsize* で 3D コマンドバッファのサイズを、*requestcount* でコマンドリクエストをキューイング可能な個数を指定します。コマンドリストオブジェクトを複数生成する場合は、それぞれのコマンドリストオブジェクトに対して nngxBindCmdlist() と nngxCmdlistStorage() を呼び出す必要があります。

以下のコード例では、3D コマンドバッファのサイズが 256 KByte、キューイング可能なコマンドリクエストの数を 128 個で指定したコマンドリストオブジェクトを 1 つ作成しています。

コード 3-4. コマンドリストオブジェクトの作成例

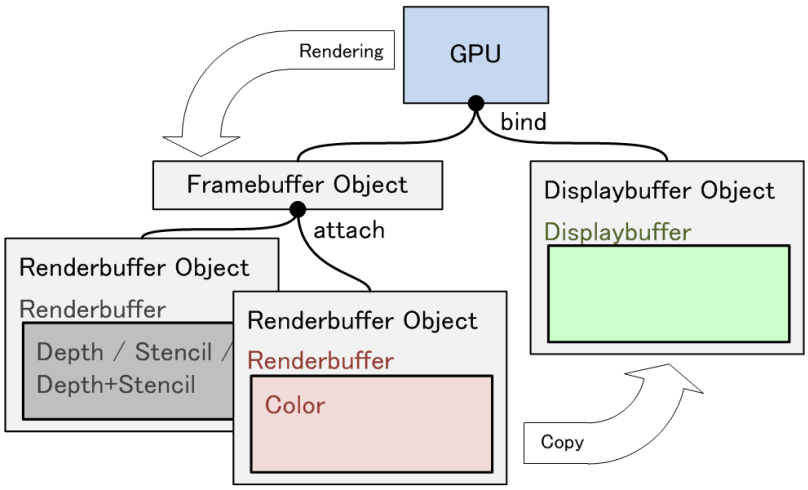
```
GLuint commandList;

nngxGenCmdlists(1, &commandList);
nngxBindCmdlist(commandList);
nngxCmdlistStorage(256 * 1024, 128);
```

3.3. バッファの確保

グラフィックス処理で使用するバッファのうち LCD 出力に関係するのは、GPU のレンダーターゲットとなるフレームバッファとレンダリング結果をコピーして LCD への表示に使用するディスプレイバッファの 2 つです。フレームバッファとディスプレイバッファへの処理は、それぞれフレームバッファオブジェクト、ディスプレイバッファオブジェクトを用いて行われます。図 3-3 に構成を示します。

図 3-3. フレームバッファとディスプレイバッファの構成



3.3.1. レンダーバッファの確保

最初に、レンダーターゲットの指定に使用するフレームバッファオブジェクトを `glGenFramebuffers()` で生成しなければなりません。フレームバッファオブジェクトに関連付けることで、各レンダーバッファ(カラー、デプス、ステンシル)はレンダーターゲットに指定することができるようになります。3DS は上下 2 つの画面を搭載していますが、上下の画面でフォーマットが同じで平行してレンダリングする必要がない場合は、メモリリソースを節約するためにもオブジェクトを共有することを推奨します。その際、バッファの幅と高さは上画面用の設定を使用してください。

コード 3-5. フレームバッファオブジェクトの生成

```
GLuint frameBufferObject;
glGenFramebuffers(1, &frameBufferObject);
```

次に、`glGenRenderbuffers()` でレンダーバッファオブジェクトを生成します。レンダーターゲットがカラーだけでなく、デプスまたはステンシル、もしくはその両方を含む場合は、フレームバッファオブジェクトに 2 つのレンダーバッファオブジェクトが必要になります。

コード 3-6. レンダーバッファオブジェクトの生成

```
GLuint renderBuffer[2];
glGenRenderbuffers(2, renderBuffer);
```

フレームバッファオブジェクトに関連付けるレンダーバッファオブジェクトを `glBindRenderbuffer()` で指定し、`glRenderbufferStorage()` でレンダーバッファを確保します。

コード 3-7. glRenderbufferStorage() の定義

```
void glRenderbufferStorage(GLenum target, GLenum internalformat,
                           GLsizei width, GLsizei height);
```

width と *height* にはレンダーバッファの幅と高さを指定します。幅と高さの最大値は、どちらも 1024 ピクセルです。

> **注意:** 幅×高さが 262128 ピクセル以上のレンダーバッファでは、特定のピクセルにブロック状のノイズが描画される可能性があります。詳細については、「15.8. 特定のピクセルにブロック状のノイズが描画される」を参照してください。

target に GL_RENDERBUFFER と以下のビットマスクとの論理和を渡すことで、バッファを確保するメモリを指定することができます。ビットマスクを指定しなかった場合は、NN_GX_MEM_VRAMA が指定されたものとして処理されます。

表 3-4. glRenderbufferStorage() の target に指定可能なビットマスク

ビットマスク	バッファの確保先
NN_GX_MEM_VRAMA	VRAM-A
NN_GX_MEM_VRAMB	VRAM-B

internalformat でバッファの種別(フォーマット)を指定しますが、3DS では以下のフォーマットから指定することができます。

表 3-5. glRenderbufferStorage() の internalformat で指定可能なフォーマット

フォーマット	ビット数	フォーマットの詳細
GL_DEPTH_COMPONENT16	16	16 bit デプス
GL_DEPTH_COMPONENT24_OES	24	24 bit デプス
GL_RGBA4	16	RGBA 各成分とも 4 bit
GL_RGB5_A1	16	RGB 各成分が 5 bit、アルファ成分が 1 bit
GL_RGB565	16	RB 成分が各 5 bit、G 成分が 6 bit。アルファ成分なし
GL_RGBA8_OES	32	RGBA 各成分とも 8 bit
GL_DEPTH24_STENCIL8_EXT	32	24 bit デプス、8 bit ステンシル
GL_GAS_DMP	32	ガスレンダリングで使用する密度情報(レンダリング結果をディスプレイバッファへコピーすることはできません)

確保したレンダーバッファは、glBindFramebuffer() で指定したフレームバッファオブジェクトに glFramebufferRenderbuffer() で関連付けることでレンダーターゲットに指定されます。

コード 3-8. glBindFramebuffer() と glFramebufferRenderbuffer() の定義

```
void glBindFramebuffer(GLenum target, GLuint framebuffer);
void glFramebufferRenderbuffer(GLenum target, GLenum attachment,
        GLenum renderbuffertarget, GLuint renderbuffer);
```

どちらの関数も *target* には GL_FRAMEBUFFER を指定してください。

framebuffer にはフレームバッファオブジェクトを、*renderbuffer* にはレンダーバッファオブジェクトを指定します。*renderbuffertarget* には GL_RENDERBUFFER を指定してください。

attachment に指定する値はレンダーバッファのフォーマットによって異なります。

表 3-6. レンダーバッファのフォーマットと attachment の対応

フォーマット	attachment に指定する値
GL_DEPTH_COMPONENT16	GL_DEPTH_ATTACHMENT
GL_DEPTH_COMPONENT24_OES	
GL_RGBA4	GL_COLOR_ATTACHMENT0
GL_RGB5_A1	
GL_RGB565	
GL_RGBA8_OES	
GL_GAS_DMP	
GL_DEPTH24_STENCIL8_EXT	GL_DEPTH_STENCIL_ATTACHMENT

glCheckFramebufferStatus() を呼び出すことで、フレームバッファオブジェクトに関連付けられたレンダーバッファオブジェクトの状態をチェックすることができます。引数には GL_FRAMEBUFFER を指定してください。それ以外を指定すると、GL_INVALID_ENUM のエラーが生成されます。

GL_FRAMEBUFFER_COMPLETE が返されたときは、正しいレンダーバッファオブジェクトが関連付けられています。

GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT が返されたときは、カラーバッファもデプス(ステンシル)バッファもアタッチされていません。

GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT が返されたときは、関連付けられているバッファのためのメモリが確保されていないか、カラーバッファとデプスバッファに同じレンダーバッファオブジェクトが関連付けられています。

GL_FRAMEBUFFER_INCOMPLETE_DIMENSIONS が返されたときは、関連付けられているバッファのサイズがそれぞれで異なっています。

以下に、レンダーバッファの確保を行うコード例を示します。カラーとデプス(ステンシル)で設定が異なることに注意してください。上下の画面でレンダーバッファを共有するため、幅と高さの指定には上画面の設定を使用しています。

コード 3-9. レンダーバッファ(カラー、デプス、ステンシル)の確保

```
// FrameBuffer
glBindFramebuffer(GL_FRAMEBUFFER, frameBufferObject);
// Color
glBindRenderbuffer(GL_RENDERBUFFER, renderBuffer[0]);
glRenderbufferStorage(GL_RENDERBUFFER | NN_GX_MEM_VRAMA, GL_RGBA8_OES,
    nn::gx::DISPLAY0_WIDTH, nn::gx::DISPLAY0_HEIGHT);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
    GL_RENDERBUFFER, renderBuffer[0]);
// Depth / Stencil
glBindRenderbuffer(GL_RENDERBUFFER, renderBuffer[1]);
glRenderbufferStorage(GL_RENDERBUFFER | NN_GX_MEM_VRAMB,
    GL_DEPTH24_STENCIL8_EXT, nn::gx::DISPLAY0_WIDTH, nn::gx::DISPLAY0_HEIGHT);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT,
    GL_RENDERBUFFER, renderBuffer[1]);
```

3.3.2. ディスプレイバッファの確保

ディスプレイバッファを確保する前に、上下どちらの画面に対するバッファであるのかを nngxActiveDisplay() で指定しなければなりません。

コード 3-10. nngxActiveDisplay() の定義

```
void nngxActiveDisplay(GLenum display);
```

display に渡す値で、上下どちらかの画面を指定することができます。*display* に下表以外の値を指定した場合は GL_ERROR_801F_DMP のエラーを生成します。

表 3-7. display に指定する値

display の値	指定される画面
NN_GX_DISPLAY0	上画面(立体視表示時は左目用)
NN_GX_DISPLAY0_EXT	上画面(立体視表示時のみ。右目用)
NN_GX_DISPLAY1	下画面

補足: 立体視表示については「3DS プログラミングマニュアル – グラフィックス応用編」を参照してください。

次に、nngxGenDisplaybuffers() でディスプレイバッファのオブジェクトを生成します。

コード 3-11. nngxGenDisplaybuffers() の定義

```
void nngxGenDisplaybuffers(GLsizei n, GLuint* buffers);
```

n には生成するディスプレイバッファのオブジェクト数を、*buffers* にはディスプレイバッファのオブジェクトを格納する配列を指定します。マルチバッファリングで 1 つの画面に対して複数のディスプレイバッファを使用する場合は、オブジェクトを必要な数だけ生成する必要があります。

表 3-8. nngxGenDisplaybuffers() が生成するエラー

エラー	原因
GL_ERROR_801C_DMP	*n* に負の値を指定した
GL_ERROR_801D_DMP	管理領域の確保に失敗した

生成したディスプレイバッファのオブジェクトを nngxBindDisplaybuffer() でターゲットのディスプレイバッファに指定します。

コード 3-12. nngxBindDisplaybuffer() の定義

```
void nngxBindDisplaybuffer(GLuint buffer);
```

buffer に未使用のオブジェクト名が指定された場合はオブジェクトの生成が行われます。その際、管理領域の確保に失敗した場合は GL_ERROR_8020_DMP のエラーを生成します。

nngxDisplaybufferStorage() でディスプレイバッファを確保します。

コード 3-13. nngxDisplaybufferStorage() の定義

```
void nngxDisplaybufferStorage(GLenum format, GLsizei width, GLsizei height,
                              GLenum area);
```

format で指定するバッファのフォーマットは以下から指定することができます。カラーバッファの確保で指定したフォーマットよりもピクセルあたりのビット数が大きいフォーマットは指定することができません。

表 3-9. nngxDisplaybufferStorage() の format で指定可能なフォーマット

フォーマット	ビット数	フォーマットの詳細
GL_RGBA4	16	RGBA 各成分とも 4 bit
GL_RGB5_A1	16	RGB 各成分が 5 bit、アルファ成分が 1 bit
GL_RGB565	16	RB 成分が各 5 bit、G 成分が 6 bit。アルファ成分なし
GL_RGB8_OES	24	RGB 各成分とも 8 bit。アルファ成分なし

width、*height* にはディスプレイバッファのサイズを指定します。ともにブロックサイズ(デフォルトは 8)の倍数の正値でなければなりません。ブロックサイズはレンダーバッファのブロックモードを変更することで 8 または 32 となります。ブロックモードの設定については、「3DS プログラミングマニュアル – グラフィックス応用編」を参照してください。

area にはバッファを確保するメモリを指定します。

表 3-10. nngxDisplaybufferStorage() の area に指定可能なフラグ

フラグ	バッファの確保先
NN_GX_MEM_FCRAM	メインメモリ(デバイスメモリ)
NN_GX_MEM_VRAMA	VRAM-A
NN_GX_MEM_VRAMB	VRAM-B

画面のキャプチャイメージを保存する場合は、CPU がアクセス可能なメインメモリにディスプレイバッファを確保する必要があります。

すでにディスプレイバッファが確保されているディスプレイバッファオブジェクトに対して、再度ディスプレイバッファを確保した場合は、すでに確保されているメモリ領域を解放して新しくメモリ領域を確保します。

表 3-11. nngxDisplaybufferStorage() が生成するエラー

エラー	原因
GL_ERROR_8021_DMP	オブジェクト名が 0 のディスプレイバッファがターゲットになっているとき
GL_ERROR_8022_DMP	*width* または *height* に不正な値を指定した
GL_ERROR_8023_DMP	*format* に指定可能な値以外を指定した
GL_ERROR_8024_DMP	*area* に指定可能な値以外を指定した
GL_ERROR_8025_DMP	メモリの確保に失敗した

ディスプレイバッファは複数確保することができます。以下のコード例ではダブルバッファリングを採用しています。

コード 3-14. ディスプレイバッファの確保

```
GLuint display0Buffers[2];
GLuint display1Buffers[2];
// Displaybuffer for UpperLCD
nngxActiveDisplay(NN_GX_DISPLAY0);
nngxGenDisplaybuffers(2, display0Buffers);
nngxBindDisplaybuffer(display0Buffers[0]);
nngxDisplaybufferStorage(GL_RGB8_OES, nn::gx::DISPLAY0_WIDTH,
        nn::gx::DISPLAY0_HEIGHT, NN_GX_MEM_FCRAM);
nngxBindDisplaybuffer(display0Buffers[1]);
nngxDisplaybufferStorage(GL_RGB8_OES, nn::gx::DISPLAY0_WIDTH,
        nn::gx::DISPLAY0_HEIGHT, NN_GX_MEM_FCRAM);
nngxDisplayEnv(0, 0);
// Displaybuffer for LowerLCD
nngxActiveDisplay(NN_GX_DISPLAY1);
nngxGenDisplaybuffers(2, display1Buffers);
nngxBindDisplaybuffer(display1Buffers[0]);
nngxDisplaybufferStorage(GL_RGB8_OES, nn::gx::DISPLAY1_WIDTH,
        nn::gx::DISPLAY1_HEIGHT, NN_GX_MEM_FCRAM);
nngxBindDisplaybuffer(display1Buffers[1]);
nngxDisplaybufferStorage(GL_RGB8_OES, nn::gx::DISPLAY1_WIDTH,
        nn::gx::DISPLAY1_HEIGHT, NN_GX_MEM_FCRAM);
nngxDisplayEnv(0, 0);
```

3.3.2.1. パラメータの取得

ターゲットとなっているディスプレイバッファの情報を nngxGetDisplaybufferParameteri() で取得することができます。

コード 3-15. ディスプレイバッファのパラメータの取得関数

```
void nngxGetDisplaybufferParameteri(GLenum pname, GLint* param);
```

pname には以下の値を指定することができます。それ以外の値を指定した場合は GL_ERROR_8033_DMP のエラーを生成します。

表 3-12. ディスプレイバッファのパラメータ

pname	取得するパラメータの内容
NN_GX_DISPLAYBUFFER_ADDRESS	ディスプレイバッファの先頭アドレス。
NN_GX_DISPLAYBUFFER_FORMAT	ディスプレイバッファのフォーマット。
NN_GX_DISPLAYBUFFER_WIDTH	ディスプレイバッファの幅。
NN_GX_DISPLAYBUFFER_HEIGHT	ディスプレイバッファの高さ。

3.4. カラーバッファからディスプレイバッファへのコピー

レンダリングが終了した時点ではカラーバッファの内容がブロックフォーマットであるため、リニアフォーマットでなければ表示することのできない LCD へは出力できません。LCD に出力可能なフォーマットへの変換と、glBindFramebuffer() で指定したフレームバッファオブジェクトに関連付けられているカラーバッファからディスプレイバッファへのコピーを行う

nngxTransferRenderImage() を呼び出す必要があります。

コード 3-16. カラーバッファからディスプレイバッファへのコピー関数

```
void nngxTransferRenderImage(GLuint buffer, GLenum mode, GLboolean yflip,
                             GLint colorx, GLint colory);
```

buffer にはコピー先のディスプレイバッファのオブジェクトを指定します。3D コマンドバッファに区切られていないコマンドが蓄積されている場合は、区切りのコマンドを追加してから転送のコマンドをコマンドリクエストに追加します。この関数の実行直後は追加された転送コマンドの実行が完了しているとは限りませんので、カラーバッファを削除したり内容を書き換えたりする場合は、コマンドリストの実行が完了するまで待ってください。

mode の設定で、レンダリング結果をディスプレイバッファにコピーする際に適用するアンチエイリアスを指定します。

表 3-13. アンチエイリアスの指定

mode の値	アンチエイリアスの指定	幅	高さ
NN_GX_ANTIALIASE_NOT_USED	オフ(なし)	等倍	等倍
NN_GX_ANTIALIASE_2x1	2x1 アンチエイリアス	2 倍	等倍
NN_GX_ANTIALIASE_2x2	2x2 アンチエイリアス	2 倍	2 倍

yflip の設定で、レンダリング結果をディスプレイバッファにコピーする際に Y フリップ(上下反転)を適用するかどうかを指定します。引数に GL_TRUE を渡すと適用されます。0 以外の値は GL_TRUE として扱われます。

colorx と *colory* にはカラーバッファからコピーする際のオフセット(左下が原点、右上が正方向)を指定します。オフセットに指定する値はブロックサイズ(ブロックモードがブロック 8 モードならば 8、ブロック 32 モードならば 32。ブロックモードの設定については「3DS プログラミングマニュアル – グラフィックス応用編」を参照してください)の倍数の正値でなければなりません。

カラーバッファのオフセット位置からディスプレイバッファの幅と高さの領域がディスプレイバッファにコピーされます。カラーバッファの幅と高さからオフセットの値を減算したものが、カラーバッファからコピー可能な領域の幅と高さです。

コピーする領域の幅と高さのピクセル数は最小サイズの制限があります。カラーバッファからコピーする幅と高さの最小値は128です。ディスプレイバッファへコピーする幅と高さの最小値は、アンチエイリアスの設定に依存します。アンチエイリアスが無効の場合は幅と高さともに128、2x1アンチエイリアスが有効な場合は幅が64、高さが128、2x2アンチエイリアスが有効な場合は幅と高さともに64です。

表 3-14. nngxTransferRenderImage() が生成するエラー

エラー	原因
GL_ERROR_8027_DMP	オブジェクト名が 0 のコマンドリストがバインドされているときに呼び出した
GL_ERROR_8028_DMP	すでにコマンドリクエストの蓄積数が最大数に達していた
GL_ERROR_8029_DMP	有効なディスプレイバッファがバインドされていない
GL_ERROR_802A_DMP	有効なカラーバッファがバインドされていない
GL_ERROR_802B_DMP	*mode* に不正な値を指定した
GL_ERROR_802C_DMP	コピー可能な領域よりもディスプレイバッファのサイズが大きい(アンチエイリアス時は表 3-13 の幅と高さの倍率を適用)

GL_ERROR_802D_DMP	*colorx* と *colory* に不正な値を指定した
GL_ERROR_802E_DMP	ディスプレイバッファのピクセル当たりのビット数がカラーバッファのものよりも大きい
GL_ERROR_802F_DMP	この関数が追加するコマンドにより 3D コマンドバッファが一杯になる
GL_ERROR_8059_DMP	ブロック 32 モードで、カラーバッファとディスプレイバッファの幅または高さが 32 の倍数でない
GL_ERROR_805A_DMP	ディスプレイバッファのピクセルサイズが 24 ビット、かつブロック 8 モードで、カラーバッファとディスプレイバッファの幅が 16 の倍数でない
GL_ERROR_80B5_DMP	カラーバッファからコピーする幅または高さのピクセル数に最小値未満の値を指定した
GL_ERROR_80B6_DMP	ディスプレイバッファへコピーする幅または高さのピクセル数に最小値未満の値を指定した

3.5. バッファスワップによる LCD 上の描画領域の更新

ディスプレイバッファへのコピー終了後に、バッファスワップ関数でレンダリング結果を LCD に表示します。

nngxActiveDisplay() でディスプレイを指定し、nngxBindDisplaybuffer() で表示に使用するディスプレイバッファを関連付けます。このとき、nngxDisplayEnv() でディスプレイバッファから LCD へ出力する際のオフセット(左下が原点、右上が正方向。正値のみ)を指定することができます。ディスプレイバッファのサイズが LCD のサイズと同じであれば (0, 0) を指定してください。オフセットに負の値を指定した場合は GL_ERROR_8026_DMP のエラーが生成されます。

コード 3-17. ディスプレイとディスプレイバッファを指定する関数

```
void nngxActiveDisplay(GLenum display);
void nngxBindDisplaybuffer(GLuint buffer);
void nngxDisplayEnv(GLint displayx, GLint displayy);
```

次に、バッファスワップ関数で LCD に出力するバッファを切り替えます。

コード 3-18. バッファスワップ関数

```
void nngxSwapBuffers(GLenum display);
```

nngxSwapBuffers() は、次の VSync 発生と同時にバッファのスワップを行います。任意のタイミングで呼び出すことができ、VSync 発生までに複数回呼び出された場合は最後の呼び出しが有効となります。

display にはバッファスワップの対象となるディスプレイを指定します。NN_GX_DISPLAY0 ならば上画面の LCD のみが、NN_GX_DISPLAY1 ならば下画面の LCD のみが、NN_GX_DISPLAY_BOTH ならば両方の LCD が対象となります。

この関数は、表示させるディスプレイバッファのアドレスを GPU に設定することで LCD に表示される画像を切り替えています。ディスプレイバッファのアドレスは、nngxDisplaybufferStorage() で確保されたバッファの先頭アドレスに、ディスプレイバッファの解像度、ピクセルサイズ、LCD の解像度、nngxDisplayEnv() で設定したオフセット値などを考慮して計算された値で、この値が最終的に GPU に設定されることになります。

設定されるアドレスは以下の式で計算されます。

$$\text{BufferAddress} + \text{PixelSize} \times (\text{DisplayBufferWidth} \times (\text{DisplayBufferHeight} - \text{LcdHeight} - \text{DisplayY}) + \text{DisplayX})$$

表 3-15. nngxSwapBuffers() が生成するエラー

エラー	原因
GL_ERROR_8030_DMP	*display* に不正な値を指定した
GL_ERROR_8031_DMP	有効なディスプレイバッファバインドされていない
GL_ERROR_8032_DMP	オフセットを考慮した表示領域がディスプレイバッファ外になる
GL_ERROR_8053_DMP	GPU に設定されるディスプレイバッファのアドレスが 16 バイトのアライメントでない
GL_ERROR_9000_DMP	立体視表示時に右目用の上画面(NN_GX_DISPLAY0_EXT)にバインドされているディスプレイバッファが 0 またはその領域が確保されていない
GL_ERROR_9001_DMP	nngxDisplayEnv() で指定された表示領域がディスプレイバッファの外部を指している
GL_ERROR_9002_DMP	2 つの上画面(NN_GX_DISPLAY0 と NN_GX_DISPLAY0_EXT)にそれぞれバインドされているディスプレイバッファの解像度、フォーマット、メモリ領域が異なる

初期状態では LCD には黒画面が強制的に表示されるようになっています。バッファスワップ関数で LCD 出力に対して有効なディスプレイバッファを用意できたら、VSync 発生に合わせて nngxStartLcdDisplay() を呼び出して LCD 出力を開始してください。この関数の呼び出しが必要なのは最初の一回だけです。

コード 3-19. LCD 出力を開始する関数

```
void nngxStartLcdDisplay( void );
```

3.5.1. アドレス指定によるバッファスワップ

nngxSwapBuffersByAddress() は、ディスプレイバッファオブジェクトを使用せずに、指定したアドレスにあるバッファの内容を LCD に表示させることができます。

コード 3-20. アドレス指定によるバッファスワップ関数

```
void nngxSwapBuffersByAddress(GLenum display,
                              const GLvoid* addr, const GLvoid* addrB,
                              GLsizei width, GLenum format);
```

nngxSwapBuffers() と同様に、この関数を呼び出したあとに発生した最初の VSync でバッファが切り替わり、VSync の発生までに同じディスプレイに対する呼び出しを複数回行った場合は、最後の呼び出しが有効となります。

display にはバッファスワップの対象となるディスプレイを指定します。NN_GX_DISPLAY0 ならば上画面の LCD のみが、NN_GX_DISPLAY1 ならば下画面の LCD のみが対象となります。

addr には表示させるバッファのアドレスを指定します。立体視表示を有効にして上画面を対象に指定したときは、この引数に指定するアドレスは左目用に表示されるイメージのアドレスです。指定するアドレスは 16 バイトアライメントでなければなりません。

addrB には、立体視表示が有効なときに、右目用に表示されるイメージのアドレスを指定します。この引数は上画面を対象としているときにのみ有効です。立体視表示が無効なときや、*display* に NN_GX_DISPLAY1 を指定しているときは、この引数の指定は無視されます。指定するアドレスは 16 バイトアライメントでなければなりません。

この関数で指定されたバッファは、nngxDisplayEnv() による表示位置の指定を無視して表示されます。そのため *addr* および *addrB* には、オフセットなどを考慮したアドレスを指定してください。アドレスの計算方法については「3.5. バッファスワップによる LCD 上の描画領域の更新」を参照してください。

width には表示させるバッファの幅のピクセル数を指定します。*width* は LCD の幅ではなく、バッファの幅です。LCD の幅のピクセル数は上画面、下画面ともに 240 ですが、幅が 240 より大きい表示バッファを部分的に表示させる場合は、表示しない部分も含めた表示バッファ全体の幅のピクセル数を指定してください。*width* は 8 の倍数、かつ 240 以上の値でなければなりません。

format にはバッファのフォーマットを指定します。指定可能なフォーマットは、ディスプレイバッファを確保するときと同じです(表 3-9)。

表 3-16. nngxSwapBuffersByAddress() が生成するエラー

エラー	原因
GL_ERROR_8087_DMP	*display* に不正な値を指定した
GL_ERROR_8088_DMP	*addr* で指定されたアドレスが 16 バイトのアライメントでない
GL_ERROR_8089_DMP	*addrB* で指定されたアドレスが 16 バイトのアライメントでない
GL_ERROR_808A_DMP	*width* に不正な値を指定した
GL_ERROR_808B_DMP	*format* に指定可能なフォーマット以外の値を指定した

3.5.2. バッファスワップの詳細

nngxSwapBuffers() や nngxSwapBuffersByAddress() によって行われるバッファスワップでは、ディスプレイバッファの内容を直接表示しているわけではありません。これらの関数は VSync 発生後に行われる LCD の描画に使用するディスプレイバッファのアドレス変更を予約するだけで、そのアドレス変更も VBlank 中に行われます。

ディスプレイバッファの内容の LCD への表示は GPU が行っています。LCD への表示の際には、スキャンラインに合わせてディスプレイバッファから 1 ライン分のデータを読み込んでいますので、VBlank 時以外は頻繁にメモリアクセスが発生しています。そのため、VBlank 時以外にディスプレイバッファの内容を書き換えてしまうと、ティアリングが発生してしまいます。

GPU がディスプレイバッファの内容を LCD に転送(表示)する際のメモリアクセスは、GPU 内では最優先で処理されますので、GPU のみがアクセス可能な VRAM 上にディスプレイバッファを配置している場合は、メモリアクセスの競合による問題は発生しません。しかし、メインメモリ(デバイスメモリ)上にディスプレイバッファを配置している場合は、CPU やそのほかのデバイスとの間でメモリアクセスが競合する可能性があります。

GPU と CPU、そのほかのデバイスからの、メインメモリへのアクセスの優先度は nngxSetMemAccessPrioMode() で調整することができます。

コード 3-21. メインメモリへのアクセスの優先度の調整関数

```
void nngxSetMemAccessPrioMode(nngxMemAccessPrioMode mode);
```

表 3-17. メインメモリへのアクセスの優先度の違い

mode	GPU	CPU	そのほか
NN_GX_MEM_ACCESS_PRIO_MODE_0	すべて均等		
NN_GX_MEM_ACCESS_PRIO_MODE_1		優先	
NN_GX_MEM_ACCESS_PRIO_MODE_2		強く優先	
NN_GX_MEM_ACCESS_PRIO_MODE_3	優先	優先	

NN_GX_MEM_ACCESS_PRIO_MODE_4	優先		

デフォルトの設定は NN_GX_MEM_ACCESS_PRIO_MODE_1 です。

CPU のアクセスの優先度を上げると、CPU でのメインメモリアクセスを伴う処理にかかる時間が GPU やほかのデバイスの動作によって受ける影響を小さくすることができます。

注意: ディスプレイバッファをメインメモリ上に配置している場合、NN_GX_MEM_ACCESS_PRIO_MODE_2 を指定して CPU から大量のメモリアクセスを発生させると、LCD 表示のための転送帯域が不足して画面に縦線状のノイズが発生することがあります。これを回避するためにはディスプレイバッファを VRAM 上に配置するか、ほかのモードを指定するようにしてください。

補足: LCD 表示のための転送帯域が不足したときは、nngxGetCmdlistParameteri() の *pname* に NN_GX_CMDLIST_HW_STATE を渡して取得したビット列のビット 18 とビット 17 に 1 が格納されています。nngxGetCmdlistParameteri() については、「4.1.10. パラメータ取得」を参照してください。

3.6. 画面更新の同期について

LCD の垂直同期(VSync)に合わせて処理を行う必要があるときには、以下の関数を利用することができます。

コード 3-22. VSync 関数

```
GLint nngxCheckVSync(GLenum display);
void nngxWaitVSync(GLenum display);
void nngxSetVSyncCallback(GLenum display, void (*func)(GLenum));
```

nngxCheckVSync() はライブラリ内部で更新している VSync カウンタの値を返します。この値の変化をチェックすることで非同期に VSync の更新をチェックすることができます。返される値はライブラリ内部の更新カウンタのため、実装の上限値(将来の実装で変更になる場合があります)を超えると 0 に戻ります。

nngxWaitVSync() は、指定された LCD の VSync が更新されるまで待ちます。呼び出し後は、VSync が更新されるまで制御が戻りません。

nngxSetVSyncCallback() は、VSync 更新のタイミングで呼び出されるコールバック関数を登録します。*func* に 0(NULL)を渡して呼び出した場合はコールバック関数の登録が解除されます。登録されたコールバック関数はメインスレッドとは異なるスレッドから呼び出されますので、メインスレッドと共有するデータを参照する場合は排他処理が必要です。ただし、同じグラフィックス関連でも nngxSetCmdlistCallback() によって登録された割り込みハンドラとの間では排他処理が不要となっています。

注意: VSync のコールバック関数内で nngx 関数を呼び出すことは可能ですが、コールバック関数が終了するまでコマンドリクエストの終了割り込みが待たされることに注意してください。そのため、コールバック関数内では、コマンドリクエストを発行する関数の呼び出しはなるべく控えるようにしてください。

どの関数も *display* に NN_GX_DISPLAY0 を渡した場合は上画面 LCD を、NN_GX_DISPLAY1 を渡した場合は下画面 LCD を、NN_GX_DISPLAY_BOTH を渡した場合は両方の LCD を対象にして処理が行われます。それ以外の値を渡した場合、nngxCheckVSync() は GL_ERROR_8019_DMP のエラーを、nngxWaitVSync() は GL_ERROR_801A_DMP のエラーを、nngxSetVSyncCallback() は GL_ERROR_801B_DMP のエラーを生成します。

システム側で極端な時間差が生じないように考慮していますが、VSync 更新タイミングには上下の画面間で約 100 マイクロ秒(システムコアの負荷状況により変動します)の時間差があります。ただし、上画面の VSync コールバック関数で重い処理を行ったり、優先度が非常に高いスレッドを起動していたりすることで下画面の VSync コールバックが待たされる可能性があるため、この時間差に過度に依存したコードを作成しないようにしてください。

液晶画面の VSync の間隔は上下画面ともに 59.831 Hz です。立体視表示(視差バリア)の有効/無効による変化はありません。

3.7. 終了処理

アプリケーションの終了時など、GX ライブラリの使用を終了するときは nngxFinalize() を呼び出してください。解放していないオブジェクトはすべて解放されます。

コード 3-23. nngxFinalize() の定義

```
void nngxFinalize(void);
```

LCD 表示のために確保したフレームバッファオブジェクトやレンダーバッファ、ディスプレイバッファを破棄する場合、フレームバッファオブジェクトの破棄には glDeleteFramebuffers() を、レンダーバッファの破棄には glDeleteRenderbuffers() を、ディスプレイバッファの破棄には nngxDeleteDisplaybuffers() をそれぞれ呼び出してください。

コード 3-24. フレームバッファオブジェクト、レンダーバッファ、ディスプレイバッファを破棄する関数

```
void glDeleteFramebuffers(GLsizei n, const GLuint* framebuffers);
void glDeleteRenderbuffers(GLsizei n, const GLuint* renderbuffers);
void nngxDeleteDisplaybuffers(GLsizei n, GLuint* buffers);
```

どの関数も、*n* には第 2 引数に渡したオブジェクト配列の個数を指定します。nngxDeleteDisplaybuffers() の *n* に負の値を指定した場合は GL_ERROR_801E_DMP のエラーが生成されます。

指定されたオブジェクトに使用中のものが含まれていた場合、フレームバッファオブジェクトとレンダーバッファについては使用中のオブジェクトに影響はなく、そのほかに指定されたオブジェクトは破棄されます。ディスプレイバッファはオブジェクト名が 0 のディスプレイバッファに切り替わり、使用中のオブジェクトの破棄が行われます。

3.8. 表示部分の指定について

カラーバッファからディスプレイバッファへの転送、ディスプレイバッファの LCD への表示において、表示部分の指定がどのように設定されるかを図 3-4 に示します。カラーバッファからディスプレイバッファへの転送は、アンチエイリアスの指定がオフであることを前提としています。

図 3-4. 表示部分の指定

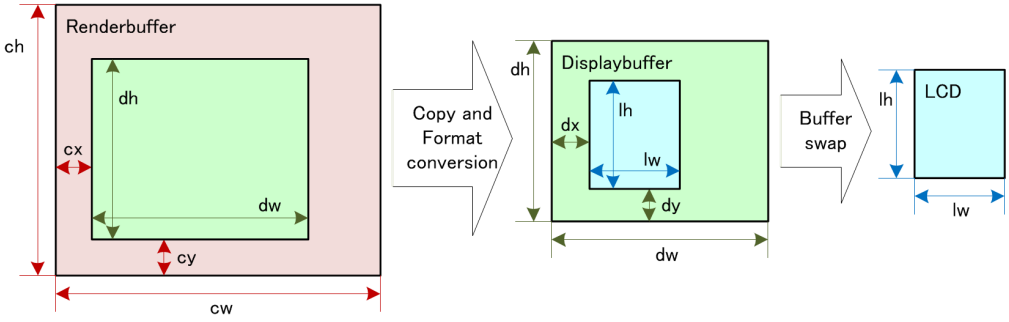


表 3-18. 表示部分の指定

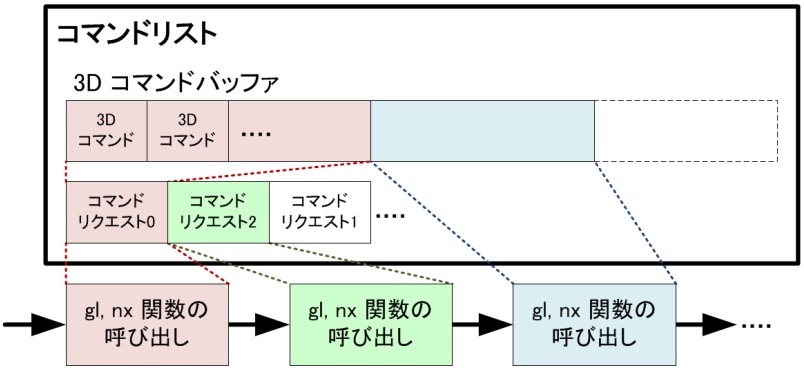
変数名	説明
cw、ch	カラーバッファの幅と高さ `glRenderbufferStorage()` の *`width`*、*`height`* で指定される値(コード 3-7)
cx、cy	カラーバッファからディスプレイバッファへのコピーする際のオフセット `nngxTransferRenderImage()` の *`colorx`*、*`colory`* で指定される値(コード 3-16)
dw、dh	ディスプレイバッファの幅と高さ `nngxDisplaybufferStorage()` の *`width`*、*`height`* で指定される値(コード 3-13)
dx、dy	ディスプレイバッファから LCD へ出力する際のオフセット `nngxDisplayEnv()` の *`displayx`*、*`displayy`* で指定される値(コード 3-17)
lw、lh	出力先 LCD の幅と高さ、上画面と下画面で幅と高さは異なります。 「3.1. LCD の解像度、配置方向について」を参照してください。

4. コマンドリスト

コマンドリストは 3DS が独自に導入したもので、3D グラフィックス処理で呼び出される gl、nngx 関数をコマンドとして記録し、まとめて実行させることができます。コマンドリストへの処理は、コマンドリストオブジェクトを用いて行われます。3DS では、コマンドリストを 3D グラフィックスの描画の実行単位として扱います。

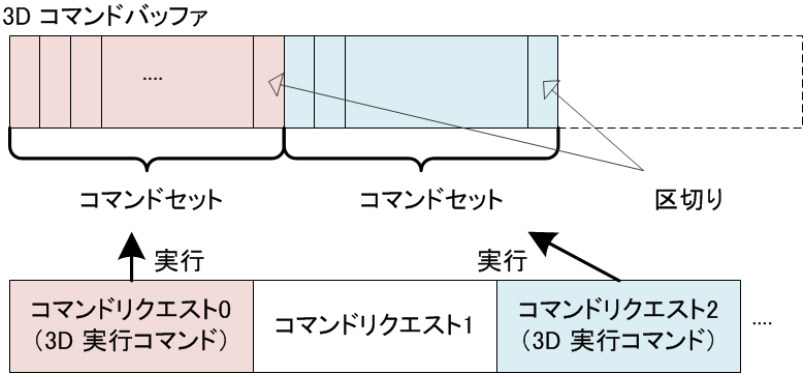
コマンドリストは GPU が直接実行するレジスタ書き込みコマンド(3D コマンド)と、CPU から GPU に命令を伝えるためのコマンドリクエストで構成されます。3D コマンドは、gl、nngx 関数で描画などが行われると、3D コマンドバッファに蓄積されます。コマンドリクエストは、要因となる特定の gl、nngx 関数の呼び出しによりキューイングされます。コマンドリクエストの種類と詳細については「4.2. コマンドリクエストの種類」で説明します。

図 4-1. コマンドリスト



コマンドリクエストにキューイングされた 3D 実行コマンドが処理されると、GPU は 3D コマンドバッファから 3D コマンドを読み込んで実行します。3D コマンドは複数で 1 つのコマンドセットとして扱われ、コマンドセット単位で実行されます。各コマンドセットの最後のコマンドは区切りとなるもので、GPU が 3D バッファの読み込みを終了するコマンドとなります。

図 4-2. コマンドセット



補足: コマンドリクエストを発行する関数はアプリコア(コア 0)でのみ呼び出しが可能です。

4.1. 使用方法

3DS では、GPU による 3D 描画はコマンドリスト単位で実行されます。そのため、アプリケーションはコマンドリストオブジェクトを生成し、gl 関数などで蓄積された 3D コマンドをまとめて実行することになります。

4.1.1. オブジェクトの生成

最初に、nngxGenCmdlists() でコマンドリストオブジェクトを生成します。

コード 4-1. コマンドリストの生成関数

```
void nngxGenCmdlists(GLsizei n, GLuint* cmdlists);
```

n 個のコマンドリストオブジェクトを生成して、*cmdlists* にその名前(オブジェクト名)を格納します。

コマンドリストは固有の名前空間を持っています。オブジェクト名が 0 のコマンドリストはシステムで予約されています。

表 4-1. nngxGenCmdlists() が生成するエラー

エラー	原因
GL_ERROR_8000_DMP	*n* に負の値を指定した
GL_ERROR_8001_DMP	管理領域の確保に失敗した

4.1.2. バインド

次に、生成したコマンドリストオブジェクトを nngxBindCmdlist() で GPU に関連付け(バインド)ます。バインドしたコマンドリストの 3D コマンドバッファに 3D コマンドが蓄積されていきます。

コード 4-2. コマンドリストのバインド関数

```
void nngxBindCmdlist(GLuint cmdlist);
```

cmdlist に未使用のオブジェクト名が指定された場合はオブジェクトの生成が行われます。

表 4-2. nngxBindCmdlist() が生成するエラー

エラー	原因
GL_ERROR_8004_DMP	管理領域の確保に失敗した
GL_ERROR_8005_DMP	コマンドキャッシュ(詳しくは「3DS プログラミングマニュアル – グラフィックス応用編」を参照してください)を利用して、コマンドリストの保存を行っている状態で呼び出した

4.1.3. メモリ領域の確保

バインドしたコマンドリストに対して、nngxCmdlistStorage() でメモリ領域の確保を行います。

コード 4-3. コマンドリストのメモリ領域の確保

```
void nngxCmdlistStorage(GLsizei bufsize, GLsizei requestcount);
```

bufsize に 3D コマンドバッファのサイズを、*requestcount* にコマンドリクエストをキューイング可能な個数を指定します。

コマンドリストオブジェクトを複数生成している場合は、それぞれに対して nngxBindCmdlist() と nngxCmdlistStorage() を呼び出してください。オブジェクト名が 0 のコマンドリストをバインドしている場合、この関数の呼び出しは無視されます。また、すでに領域を確保しているオブジェクトに対して、この関数を再度呼び出した場合は確

保していた領域を解放し、再度領域の確保が行われます。

確保した 3D コマンドバッファのサイズを超えて 3D コマンドが蓄積された場合や 3D コマンドバッファが未設定の場合は、該当の関数呼び出しで GL_ERROR_COMMANDBUFFER_FULL_DMP のエラーが生成されます。コマンドリクエストにキューイング可能な個数を超えてキューイングされた場合やバッファが未設定の場合は、該当の関数呼び出しで GL_ERROR_COMMANDREQUEST_FULL_DMP のエラーが生成されます。

表 4-3. nngxCmdlistStorage() が生成するエラー

エラー	原因
GL_ERROR_8006_DMP	メモリ領域の確保に失敗した
GL_ERROR_8007_DMP	実行中のコマンドリストに対して呼び出した
GL_ERROR_8008_DMP	引数に負の値を指定した

4.1.4. 実行

バインドされているコマンドリストにキューイングされたコマンドリクエストの実行を開始させるには、nngxRunCmdlist() を呼び出します。

コード 4-4. コマンドリストの実行関数

```
void nngxRunCmdlist(void);
void nngxRunCmdlistByID(GLuint cmdlist);
```

オブジェクト名が 0 のコマンドリストをバインドしている場合、実行は無視されます。また、コマンドリクエストの実行中にほかのコマンドリストをバインドし、この関数を実行しても無視されます。

コマンドリクエストの実行開始後は、続けて同じコマンドリストにコマンドを蓄積することもできますが、別のコマンドリストをバインドしてコマンドを蓄積させることもできます。ただし、コマンドの蓄積順と実行順は同じでなければなりません。

nngxRunCmdlistByID() は、現在バインドされているコマンドリストではなく、*cmdlist* で指定されたコマンドリストを実行します。指定されたコマンドリストを実行する以外には、nngxRunCmdlist() の動作との違いはありません。

表 4-4. nngxRunCmdlist() および nngxRunCmdlistByID() が生成するエラー

エラー	原因
GL_ERROR_8009_DMP	メモリ領域が確保されていないコマンドリストに対して呼び出した (nngxRuncmdlist())
GL_ERROR_80B1_DMP	メモリ領域が確保されていないコマンドリストに対して呼び出した (nngxRuncmdlistByID())

4.1.4.1. 実行状態の取得

コマンドリストが実行中かどうかは nngxGetIsRunning() で取得することができます。

コード 4-5. コマンドリストの実行状態の取得関数

```
GLboolean nngxGetIsRunning(void);
```

この関数は、コマンドリストが現在バインドされているものかどうかに関係なく、実行中のコマンドリストがあれば GL_TRUE を返します。

同様に、コマンドリストが実行中かどうかを nngxGetCmdlistParameteri() の *pname* に NN_GX_CMDLIST_IS_RUNNING を渡して取得できますが、この方法で取得する実行状態は、現在バインドされているコマンドリストが実行中かどうかだけです。nngxGetCmdlistParameteri() については、「4.1.10. パラメータ取得」を参照してください。

4.1.5. 破棄

不要になったコマンドリストオブジェクトは nngxDeleteCmdlists() で破棄することができます。

コード 4-6. コマンドリストの破棄関数

```
void nngxDeleteCmdlists(GLsizei n, const GLuint* cmdlists);
```

cmdlists に格納されている、*n* 個のオブジェクト名で指定されたコマンドリストオブジェクトを破棄します。実行中のコマンドリストが含まれていた場合、GL_ERROR_8003_DMP のエラーが生成されますがコマンドリストの実行に影響はなく、ほかに指定されているコマンドリストオブジェクトが破棄されます。

表 4-5. nngxDeleteCmdlists() が生成するエラー

エラー	原因
GL_ERROR_8002_DMP	*n* に負の値を指定した
GL_ERROR_8003_DMP	*cmdlists* に実行中のコマンドリストが含まれていた

4.1.6. 停止

実行中のコマンドリストを停止するには、以下の関数を呼び出します。

コード 4-7. コマンドリストの停止関数

```
void nngxStopCmdlist(void);
void nngxReserveStopCmdlist(GLint id);
```

nngxStopCmdlist() は、呼び出し時に実行中だったコマンドリクエストが完了した時点で停止します。すでに実行が開始されているコマンドリクエスト(実行開始待ちを含む)を中止することはできません。

nngxReserveStopCmdlist() は *id* 番目に蓄積したコマンドリクエストの実行完了直後に停止します。

停止したコマンドリストを再開する場合は、nngxRunCmdlist() を呼び出します。ただし、停止を指示してからコマンドの実行が完了するまでに呼び出した場合は無視されてしまうことに注意してください。

表 4-6. nngxReserveStopCmdlist() が生成するエラー

エラー	原因
GL_ERROR_800A_DMP	実行中のコマンドリストに対して呼び出した
GL_ERROR_800B_DMP	*id* に 0 または負の値、コマンドリクエストの最大数を超える値を指定した

4.1.7. 3D コマンドバッファの区切り

nngxSplitDrawCmdlist() で 3D コマンドバッファにバッファ読み込み終了コマンドを追加し、3D 実行コマンドをキューイングすることができます。コマンドリストが 3D コマンドを蓄積しながら実行している場合は、この関数で区切られた部分までの 3D コマンドが実行されます。

コード 4-8. 3D コマンドバッファを区切る関数

```
void nngxSplitDrawCmdlist(void);
```

3D 実行コマンドはバッファ読み込み終了コマンドが追加されるまでキューイングされません。この関数以外にも 3D 実行コマンドをキューイングする関数があります。glClear() や glTexImage2D() などは 3D コマンドの実行を停止させなければならないため、バッファ読み込み終了コマンドの追加と 3D 実行コマンドのキューイングを行います。

表 4-7. nngxSplitDrawCmdlist() が生成するエラー

エラー	原因
GL_ERROR_800C_DMP	オブジェクト名が 0 のコマンドリストをバインドしているときに呼び出した
GL_ERROR_800D_DMP	すでにコマンドリクエストの蓄積数が最大数に達していた
GL_ERROR_800E_DMP	この関数が追加するコマンドにより 3D コマンドバッファが一杯になる

この関数を内部で呼び出している関数が、これらのエラーを生成することがあります。

4.1.7.1. 蓄積済み 3D コマンドバッファのフラッシュ

nngxSplitDrawCmdlist() を呼び出すと、3D コマンドバッファに 3D コマンドが蓄積されていない場合でも、バッファ読み込み終了コマンドの追加と 3D 実行コマンドのキューイングが行われます。つまり、意図せず無駄なコマンドが追加されることになりますので、3D コマンドが蓄積されている場合にのみ 3D コマンドバッファを区切るためのコマンドを追加する nngxFlush3DCommand() または nngxFlush3DCommandNoCacheFlush() を呼び出すことを推奨します。キャッシュのフラッシュが複数回発生するような場合はキャッシュをフラッシュしない後者の関数を呼び出し、nngxUpdateBufferLight() でまとめてキャッシュを反映することで CPU コストを抑えられる可能性があります。

コード 4-9. 3D コマンドバッファをフラッシュする関数

```
void nngxFlush3DCommand(void);
void nngxFlush3DCommandNoCacheFlush(void);
```

これらの関数は、バインドされているコマンドリストの 3D コマンドバッファが最後に区切られたあと、3D コマンドが蓄積されていなければバッファ読み込み終了コマンドと 3D 実行コマンドの追加を行わず、3D コマンドが蓄積されていればバッファ読み込み終了コマンドと 3D 実行コマンドの追加を行います。3D コマンドを蓄積しながら実行している場合は、この関数で区切られた部分までの 3D コマンドが実行されます。

表 4-8. nngxFlush3DCommand() および nngxFlush3DCommandNoCacheFlush() が生成するエラー

エラー	原因
GL_ERROR_8084_DMP GL_ERROR_80AE_DMP	オブジェクト名が 0 のコマンドリストをバインドしているときに呼び出した
GL_ERROR_8085_DMP GL_ERROR_80AF_DMP	すでにコマンドリクエストの蓄積数が最大数に達していた
GL_ERROR_8086_DMP GL_ERROR_80B0_DMP	この関数が追加するコマンドにより 3D コマンドバッファが一杯になる

4.1.7.2. 蓄積済み 3D コマンドバッファの部分フラッシュ

指定したサイズ分の 3D コマンドを実行する nngxFlush3DCommandPartially() が用意されています。この関数は nngxFlush3DCommand() の機能を拡張したもので、nngxAdd3DCommand() などで追加した、コマンドバッファ実行レジスタのキックコマンドを含む 3D コマンドを正しく実行させるために呼び出します。詳細は「3DS プログラミングマニュアル – グラフィックス応用編」の「コマンドバッファ実行レジスタ(0x0238 ～ 0x023D)」を参照してください。

コード 4-10. 3D コマンドバッファを部分フラッシュする関数

```
void nngxFlush3DCommandPartially(GLsizei buffersize);
```

buffersize に、実行するコマンドバッファのサイズをバイト数で指定します。16 の倍数でなければなりません。

buffersize には、前回のコマンドフラッシュ後のアドレスから、最初のキックコマンドまで(キックコマンドを含む)のサイズを正しく指定する必要があります。誤った値を指定した場合、意図しない順序でコマンドが実行される、正しく実行を終了できないなどの動作を起こす可能性があります。

前回のコマンドフラッシュ後から、この関数を呼び出すまでに蓄積された 3D コマンドバッファのキャッシュフラッシュはアプリケーションで確実に行ってください。この関数内でも割り込み発生コマンドなどが生成されるため、キャッシュ全体のフラッシュは関数を呼び出したあとでなければなりません。また、キャッシュがフラッシュされる前に実行されないようにするため、この関数は実行中のコマンドリストに対して呼び出すことができません。なお、glClear() や nngxTransferRenderImage()、glCopyTexImage2D() などの関数は、関数内で nngxFlush3DCommand() と同じ手法のフラッシュを実行します。これらの関数を呼び出す前に、必ず本関数でフラッシュしてください。

nngxAddJumpCommand() や nngxAddSubroutineCommand() を使用してキックコマンドを追加した場合、最初のキックコマンドまでを実行サイズとしてキックするようにドライバ側がサイズを調整します。そのため、これらの関数を使用している場合は、nngxFlush3DCommandPartially() を呼び出す必要はありません。

nngxAddSubroutineCommand() でキックコマンドを追加したコマンドバッファに対して部分フラッシュを行うと、ドライバ側で計算された実行サイズではなく、*buffersize* で指定されたサイズが実行サイズとして使用されることに注意してください。

表 4-9. nngxFlush3DCommandPartially() が生成するエラー

エラー	原因
GL_ERROR_80A9_DMP	オブジェクト名が 0 のコマンドリストをバインドしているときに呼び出した
GL_ERROR_80AA_DMP	すでにコマンドリクエストの蓄積数が最大数に達していた
GL_ERROR_80AB_DMP	この関数が追加するコマンドにより 3D コマンドバッファが一杯になる
GL_ERROR_80AC_DMP	*buffersize* に 0 以下の値、または 16 の倍数以外の値を指定した

GL_ERROR_80AD_DMP	実行中のコマンドリストに対して呼び出した

4.1.8. クリア

コマンドリストをクリアして、3D コマンドバッファとコマンドリクエストのキューを未使用状態（メモリ領域確保直後の状態）にします。

コード 4-11. コマンドリストのクリア関数

```
void nngxClearCmdlist(void);
```

表 4-10. nngxClearCmdlist() が生成するエラー

エラー	原因
GL_ERROR_800F_DMP	実行中のコマンドリストに対して呼び出した

4.1.8.1. クリアと 3D コマンドバッファのフィル

コマンドリストのクリアとともに、3D コマンドバッファの内容を指定されたデータで初期化します。3D コマンドバッファとコマンドリクエストのキューは未使用状態になります。

コード 4-12. コマンドリストのクリアと 3D コマンドバッファのフィル関数

```
void nngxClearFillCmdlist(GLuint data);
```

表 4-11. nngxClearFillCmdlist() が生成するエラー

エラー	原因
GL_ERROR_8065_DMP	実行中のコマンドリストに対して呼び出した

4.1.9. パラメータ設定

nngxSetCmdlistParameteri() を呼び出すことで、コマンドリストの設定パラメータを指定することができます。

コード 4-13. コマンドリストのパラメータ設定関数

```
void nngxSetCmdlistParameteri(GLenum pname, GLint param);
```

表 4-12. コマンドリストの設定パラメータ（設定）

pname の値	設定の内容

NN_GX_CMDLIST_GAS_UPDATE	この設定はコマンドリストオブジェクトごとに設定され、以下の値から設定を選択します。 ● GL_TRUE:ガスの密度情報描画の加算ブレンド結果を更新 ● GL_FALSE:通常動作(デフォルト) この設定が GL_TRUE の状態で、nngxSplitDrawCmdlist() や nngxFlush3DCommand() を呼び出すと、蓄積された 3D 実行コマンドの実行終了時にガスの密度情報描画の加算ブレンド結果を更新します。 GL_FALSE の状態では通常動作に戻り、必要なときにのみガスの密度情報が更新されるコマンドが蓄積されます。 この設定は、nngxSplitDrawCmdlist() や nngxFlush3DCommand() を呼び出したときに GL_TRUE であるかどうかで機能します。3D 実行コマンドが実行されるときに GL_TRUE であるかどうかは影響しません。また、nngxSplitDrawCmdlist() および nngxFlush3DCommand() 以外を呼び出して蓄積された 3D 実行コマンドには影響しません。 ガス密度情報描画の加算ブレンド結果の更新については、「3DS プログラミングマニュアル – グラフィックス応用編」の「ガス制御設定レジスタ」も併せて参照してください。

表 4-13. nngxSetCmdlistParameteri() が生成するエラー

エラー	原因
GL_ERROR_8015_DMP	実行中のコマンドリストに対して呼び出した
GL_ERROR_8016_DMP	*pname* または *param* に無効な値を指定した

4.1.10. パラメータ取得

nngxGetCmdlistParameteri() を呼び出すことで、コマンドリストの設定パラメータを取得することができます。

コード 4-14. コマンドリストのパラメータ取得関数

```
void nngxGetCmdlistParameteri(GLenum pname, GLint* param);
```

表 4-14. コマンドリストの設定パラメータ(取得)

pname の値	取得可能な設定の内容
NN_GX_CMDLIST_IS_RUNNING	コマンドリストの実行状態。 GL_TRUE:コマンドリストは実行中です。 GL_FALSE:コマンドリストは実行中ではありません。
NN_GX_CMDLIST_USED_BUFSIZE	蓄積された 3D コマンドバッファのバイトサイズ。
NN_GX_CMDLIST_USED_REQCOUNT	蓄積されたコマンドリクエストの個数。
NN_GX_CMDLIST_MAX_BUFSIZE	3D コマンドバッファの最大サイズ。 nngxCmdlistStorage() の *bufsize* で指定した値です。
NN_GX_CMDLIST_MAX_REQCOUNT	コマンドリクエストの最大個数。 nngxCmdlistStorage() の *requestcount* で指定した値です。
NN_GX_CMDLIST_TOP_BUFADDR	3D コマンドバッファの先頭アドレス。
NN_GX_CMDLIST_BINDING	現在バインドされているコマンドリストのオブジェクト名。
NN_GX_CMDLIST_RUN_BUFSIZE	実行済みの 3D コマンドバッファのバイトサイズ。
NN_GX_CMDLIST_RUN_REQCOUNT	実行済みのコマンドリクエストの個数。

NN_GX_CMDLIST_TOP_REQADDR	コマンドリクエストのリクエストキュー用データ領域の先頭アドレス。
NN_GX_CMDLIST_NEXT_REQTYPE	次に実行されるコマンドリクエストまたは実行中のコマンドリクエストのコマンドの種類。 *param* に返される値は、現在バインドされているコマンドリストの状態によって変化します。実行停止中のコマンドリストの場合は次に実行されるコマンドリクエストのコマンドの種類が、実行中の場合は実行中のコマンドリクエストのコマンドの種類が返されます。すべてのコマンドリクエストの実行が完了している場合は NULL が返されます。 コマンドの種類は以下のマクロで定義されています。 NN_GX_CMDLIST_REQTYPE_DMA：DMA 転送コマンド NN_GX_CMDLIST_REQTYPE_RUN3D：3D 実行コマンド NN_GX_CMDLIST_REQTYPE_FILLMEM：メモリフィルコマンド NN_GX_CMDLIST_REQTYPE_POSTTRANS：ポスト転送コマンド NN_GX_CMDLIST_REQTYPE_COPYTEX：レンダーテクスチャ転送コマンド
NN_GX_CMDLIST_NEXT_REQINFO	コマンドバッファのアドレスとバイトサイズ。 *param* の第 1 要素にコマンドバッファのアドレスが、第 2 要素にコマンドバッファのバイトサイズが格納されますので、*param* には要素数が 2 以上の GLint の配列へのポインタを渡す必要があります。 現在バインドされているコマンドリストが実行停止中の場合は、次に実行されるコマンドリクエストのパラメータ情報が返されます。実行中の場合は、現在実行中のコマンドリクエストのパラメータ情報が返されます。すべてのコマンドリクエストの実行が完了している場合は何も返しません。 次に実行される、または実行中のコマンドリクエストが 3D 実行コマンドである場合にのみ対応しています。それ以外のコマンドの場合は何も返しません。
NN_GX_CMDLIST_HW_STATE	ハードウェアの状態を示す 32 ビットのデータ。 以下の状態である場合に対応するビットに 1 がセットされます。 ビット 20：ポスト転送が実行中 ビット 19：メモリフィルが実行中 ビット 18：下画面 LCD の FIFO でアンダーランエラーが発生 ビット 17：上画面 LCD の FIFO でアンダーランエラーが発生 ビット 16：ポスト頂点キャッシュがビジー ビット 15：レジスタ 0x0252 の［1：0］に 1 が設定されている ビット 14：頂点プロセッサ 3 がビジー ビット 13：頂点プロセッサ 2 がビジー ビット 12：頂点プロセッサ 1 がビジー ビット 11：頂点プロセッサ 0(ジオメトリシェーダプロセッサ兼用)がビジー ビット 10：レジスタ 0x0229 の［1：0］が 0 以外 ビット 9：コマンドバッファおよび頂点アレイをロードするモジュールの入力がビジー ビット 8：コマンドバッファおよび頂点アレイをロードするモジュールの出力がビジー ビット 7：アーリーデプステストモジュールがビジー ビット 6：パーフラグメントオペレーションモジュールが前段のモジュールからのデータ処理に関してビジー ビット 5：パーフラグメントオペレーションモジュールがフレームバッファアクセスに関してビジー ビット 4：テクスチャコンバイナがビジー ビット 3：フラグメントライティングがビジー ビット 2：テクスチャユニットがビジー ビット 1：ラスタライゼーションモジュールがビジー ビット 0：トライアングルセットアップがビジー
NN_GX_CMDLIST_CURRENT_BUFADDR	現在バインドされているコマンドリストで、次にコマンドが蓄積される 3D コマンドバッファのアドレス。

表 4-15. nngxGetCmdlistParameteri() が生成するエラー

エラー	原因
GL_ERROR_8017_DMP	*pname* または *param* に無効な値を指定した
GL_ERROR_8018_DMP	オブジェクト名が 0 のコマンドリストがバインドされているときに、*pname* に NN_GX_CMDLIST_BINDING 以外を指定した

4.1.11. コマンド終了割り込み

コマンドリストのコマンドリクエストが終了したタイミングで割り込みを発生させ、割り込みハンドラを呼び出すことができます。割り込みハンドラは nngxSetCmdlistCallback() で登録することができます。

コード 4-15. 割り込みハンドラの登録関数

```
void nngxSetCmdlistCallback(void (*func)(GLint));
```

割り込みハンドラはバインドされているコマンドリストに対してのみ有効です。*func* に 0(NULL)を渡して呼び出した場合はハンドラの登録を解除します。

割り込みハンドラはメインスレッドとは異なるスレッドから呼び出されますので、メインスレッドと共有するデータを参照する場合は排他処理が必要です。ただし、同じグラフィックス関連の nngxSetVSyncCallback() によって登録されたコールバック関数との間では排他処理が不要となっています。

表 4-16. nngxSetCmdlistCallback() が生成するエラー

エラー	原因
GL_ERROR_8010_DMP	実行中のコマンドリストに対して呼び出した

割り込みを発生させるには、nngxEnableCmdlistCallback() で終了時に割り込みを発生させるコマンドリクエストを指定します。nngxDisableCmdlistCallback() は割り込みの発生を無効化することができます。

コード 4-16. 割り込み制御関数

```
void nngxEnableCmdlistCallback(GLint id);
void nngxDisableCmdlistCallback(GLint id);
```

id には何番目に蓄積されたコマンドリクエストの終了時に割り込みを発生させるかを指定します。1 つのコマンドリストに対して別々の *id* で複数回呼び出し、割り込みを複数回発生させることもできます。*id* 番目に実行されるコマンドリクエストではなく、*id* 番目に蓄積されたコマンドリクエストであることに注意してください。指定する *id* の値には、nngxGetCmdlistParameteri() の *pname* を NN_GX_CMDLIST_USED_REQCOUNT で呼び出して取得した結果を利用することができます。*id* に -1 を指定した場合は、コマンドリストに蓄積されたコマンドリクエストすべてが終了したときに割り込みが発生します。

コマンドリストに蓄積されたコマンドリクエストの最後尾以外への割り込みは、割り込みハンドラが呼び出されたときにまだコマンドリストは実行中です。そのため、コマンドリストの実行中に呼び出すことができない関数は割り込みハンドラ内で呼び出すことはできません。

割り込みハンドラを登録しなくても、nngxGetCmdlistParameteri() の *pname* を NN_GX_CMDLIST_IS_RUNNING で呼び出して取得した結果が GL_FALSE になるまで待つことで、コマンドリクエストの実行終了を判断することができます。

表 4-17. nngxEnableCmdlistCallback() および nngxDisableCmdlistCallback() が生成するエラー

エラー	原因
GL_ERROR_8012_DMP GL_ERROR_8014_DMP	*id* に 0 または -1 以外の負値、コマンドリクエストの最大数を指定した

4.1.12. コマンド実行の完了待ち

nngxWaitCmdlistDone() の呼び出しで、コマンドリストに蓄積されたコマンドリクエストの実行がすべて完了するまで待つことができます。

コード 4-17. コマンド完了待ち関数

```
void nngxWaitCmdlistDone(void);
```

3D 実行コマンドは区切られた部分まで実行されます。蓄積されている 3D 実行コマンドをすべて実行させるには、この関数を呼び出す前に nngxSplitDrawCmdlist() を呼び出してください。

この関数はコマンドの実行が完了するまで処理を返しませんが、nngxSetTimeout() でタイムアウト時間を設定することができます。

コード 4-18. コマンド完了待ち関数のタイムアウト時間を設定する関数

```
void nngxSetTimeout(GLint64EXT time, void (*callback)(void));
```

time には、nngxWaitCmdlistDone() の処理がタイムアウトするまでの時間をチック値で指定します。0 を指定した場合はタイムアウトが発生しません。

callback には、タイムアウト時に呼び出されるコールバック関数を指定します。NULL を指定した場合はタイムアウト発生時にコールバック関数を呼び出しません。

デフォルトは *time* に 0 を、*callback* に NULL を指定した状態ですので、タイムアウトは発生しません。

4.1.13. DMA 転送コマンドの追加

nngxAddVramDmaCommand() または nngxAddVramDmaCommandNoCacheFlush() の呼び出しで、VRAM への DMA 転送を行うコマンドがコマンドリストに蓄積されます。前者は転送元のキャッシュフラッシュを行いますが、後者はキャッシュフラッシュを行いません。これらの関数では、メインメモリから VRAM への DMA 転送のみを行うことができます。

コード 4-19. DMA 転送コマンドを追加する関数

```
void nngxAddVramDmaCommand(
                        const GLvoid* srcaddr, GLvoid* dstaddr, GLsizei size);
void nngxAddVramDmaCommandNoCacheFlush(
                        const GLvoid* srcaddr, GLvoid* dstaddr, GLsizei size);
```

srcaddr には転送元のアドレス、*dstaddr* には転送先のアドレスをそれぞれ指定し、*size* には転送するデータのサイズを指定します。

nngxAddVramDmaCommand() では、有効なコマンドリストがバインドされていない状態で呼び出されたときは GL_ERROR_8062_DMP のエラーを、*size* に負の値を指定したときは GL_ERROR_8064_DMP のエラーを生成します。

nngxAddVramDmaCommandNoCacheFlush() では、有効なコマンドリストがバインドされていない状態で呼び出されたときは GL_ERROR_8090_DMP のエラーを、*size* に負の値を指定したときは GL_ERROR_8091_DMP のエラーを生成します。

4.1.14. アンチエイリアスフィルタ転送コマンドの追加

nngxFilterBlockImage() の呼び出しで、アンチエイリアスフィルタを適用したイメージ転送を行うコマンドがコマンドリストに蓄積されます。イメージ転送はブロックフォーマットのまま行われ、フォーマットの変換は行われません。アンチエイリアスの指定は 2x2 にのみ対応しています。

コード 4-20. アンチエイリアスフィルタ転送コマンドを追加する関数

```
void nngxFilterBlockImage(const GLvoid* srcaddr, GLvoid* dstaddr,
                          GLsizei width, GLsizei height, GLenum format);
```

srcaddr には転送元のアドレス、*dstaddr* には転送先のアドレスをそれぞれ指定し、*width*、*height*、*format* には転送元のイメージの幅、高さ、フォーマットをそれぞれ指定します。

width と *height* は *format* の指定によって以下のように制限されています。

表 4-18. フォーマットによる転送元イメージの幅と高さの制限

format	width	height
GL_RGBA8_OES GL_RGB8_OES	64 以上かつ、64 の倍数	64 以上かつ、16 の倍数
GL_RGBA4 GL_RGB5_A1 GL_RGB565	128 以上かつ、128 の倍数	128 以上かつ、16 の倍数

転送元の領域と転送先の領域が重なる場合、*srcaddr* と *dstaddr* が等しい、または *srcaddr* が *dstaddr* より大きいならば正常に動作します。なお、*srcaddr* が *dstaddr* より小さい場合は転送結果が壊れる可能性があります。

srcaddr にデバイスメモリ上のアドレスを指定した場合、転送元の領域のキャッシュがフラッシュされていなければ、正しい結果にならない可能性があります。

表 4-19. nngxFilterBlockImage() が生成するエラー

エラー	原因
GL_ERROR_8068_DMP	オブジェクト名が 0 のコマンドリストをバインドしているとき、コマンドリクエストのキューに空きがないときに呼び出した
GL_ERROR_8069_DMP	*srcaddr* または *dstaddr* に指定するアドレスが 8 バイトのアライメントでない
GL_ERROR_806A_DMP	制限に抵触する *width* または *height* を指定した
GL_ERROR_806B_DMP	制限に表記されている以外のフォーマットを *format* に指定した

4.1.15. 画像イメージ転送コマンドの追加

nngxTransferLinearImage() の呼び出しで、レンダーバッファまたはテクスチャへの画像イメージ転送を行うコマンドがコマンドリストに蓄積されます。カレントの 3D コマンドバッファに区切られていないコマンドが蓄積されている場合は、区切りのコマンドを追加してから転送コマンドが追加されます。

イメージの転送中にリニアフォーマットからブロックフォーマットへの変換が行われますが、行われるのはアドレッシングの変換のみです。レンダーバッファに対して呼び出した場合、転送時のブロックアドレッシングへの変換はブロックモードの設定によって、自動的に 8 ブロック用と 32 ブロック用のアドレッシング変換が行われます。テクスチャに対して呼び出した場合は 8 ブロック用のアドレッシング変換が行われます。どちらの場合でも、転送元画像に対して V 方向のフリップとバイトオーダーの変換を事前に行う必要があります。

> **補足:** ブロックモードについては、「3DS プログラミングマニュアル – グラフィックス応用編」の「ブロックモードの設定」を参照してください。

コード 4-21. 画像イメージ転送コマンドを追加する関数

```
void nngxTransferLinearImage(const GLvoid* srcaddr, GLuint dstid,
                             GLenum target);
```

srcaddr には転送元画像の先頭アドレスを指定します。画像は、転送先のレンダーバッファまたはテクスチャと同じフォーマット、同じ幅、高さでなければなりません。ただし、ピクセルフォーマットが 24 ビットフォーマット同士の転送はハードウェアでサポートされていませんので、転送先のピクセルフォーマットが 24 ビットフォーマットの場合、転送元のピクセルデータは 32 ビットフォーマットでなければなりません。このとき、転送元データは 4 バイトごとに最初の 1 バイト(内部フォーマットのアルファ成分)が切り捨てられて転送されます。

dstid には転送先のレンダーバッファまたはテクスチャのオブジェクト ID を、*target* には転送先オブジェクトの種類を指定します。

表 4-20. target, dstid に指定する値

target の値	dstid に指定する値
GL_RENDERBUFFER	レンダーバッファのオブジェクト ID。 0 を指定した場合は、カレントのフレームバッファにアタッチされているカラーバッファに転送されます。
GL_TEXTURE_2D	2D テクスチャのオブジェクト ID。
GL_TEXTURE_CUBE_MAP_POSITIVE_X{,Y,Z} GL_TEXTURE_CUBE_MAP_NEGATIVE_X{,Y,Z}	キューブマップテクスチャのオブジェクト ID。

転送先のレンダーバッファの幅および高さは、ブロックモードがブロック 8 モードの場合は 8 の倍数、ブロック 32 モードの場合は 32 の倍数でなければなりません。また、幅および高さは 128 以上でなければなりません。

表 4-21. nngxTransferLinearImage() が生成するエラー

エラー	原因
GL_ERROR_805B_DMP	オブジェクト名が 0 のコマンドリストをバインドしているときに呼び出した
GL_ERROR_805C_DMP	すでにコマンドリクエストの蓄積数が最大値に達していた
GL_ERROR_805D_DMP	この関数が追加するコマンドにより 3D コマンドバッファが一杯になる
GL_ERROR_805E_DMP	*dstid* に指定したレンダーバッファまたはテクスチャが存在しない、またはメモリ領域が確保されていない
GL_ERROR_805F_DMP	転送先のレンダーバッファの幅および高さの制限に抵触する
GL_ERROR_8060_DMP	*target* に不正な値を指定した
GL_ERROR_8067_DMP	転送先のレンダーバッファまたはテクスチャのピクセルサイズが 32 ビット、24 ビット、16 ビット以外

4.1.16. ブロックイメージからリニアイメージへの変換転送コマンドの追加

nngxAddB2LTransferCommand() の呼び出しで、ブロックイメージをリニアイメージに変換して転送を行うコマンドがコマンドリストに追加されます。nngxTransferRenderImage() でも同様の機能が提供されていますが、この関数は、より汎用的な機能を提供します。また、3D コマンドの区切りコマンドを追加せず、転送リクエストコマンドのみを追加する点が異なります。

コード 4-22. ブロックイメージからリニアイメージへの変換転送コマンドを追加する関数

```
void nngxAddB2LTransferCommand(
    const GLvoid* srcaddr, GLsizei srcwidth, GLsizei srcheight, GLenum srcformat,
    GLvoid* dstaddr, GLsizei dstwidth, GLsizei dstheight, GLenum dstformat,
    GLenum aamode, GLboolean yflip, GLsizei blocksize);
```

srcaddr には転送元(ブロックイメージ)のアドレスを指定します。*dstaddr* には転送先(リニアイメージ)のアドレスを指定します。*srcaddr* と *dstaddr* は、ともに 16 バイトアライメントでなければなりません。

srcwidth と *srcheight*、*dstwidth* と *dstheight* には、それぞれ転送元イメージの幅と高さ、転送先イメージの幅と高さをピクセル数で指定します。転送元イメージおよび転送先イメージの幅と高さは、ブロックサイズ(8 または 32)の倍数でなければなりません。さらに、転送先イメージのピクセルサイズが 24 bit、かつブロックサイズが 8 の場合、転送元イメージの幅と転送先イメージの幅は 16 の倍数でなければなりません。*srcwidth*、*srcheight*、*dstwidth*、*dstheight* のいずれかに 0 が指定されていると、コマンドは追加されません。転送先イメージの幅と高さのピクセル数は、転送元イメージと同じか小さくなければなりません。

転送元イメージおよび転送先イメージの幅と高さのピクセル数は最小サイズの制限があります。転送元イメージの幅と高さの最小値は128です。転送先イメージの幅と高さの最小値は、アンチエイリアスの設定に依存します。アンチエイリアスが無効の場合は幅と高さともに128、2x1アンチエイリアスが有効な場合は幅が64、高さが128、2x2アンチエイリアスが有効な場合は幅と高さともに64です。

srcformat と *dstformat* には、それぞれ転送元イメージおよび転送先イメージのピクセルフォーマットを指定します。指定可能なピクセルフォーマットは以下の 5 種類です。

表 4-22. ピクセルフォーマットの指定

定義	ビット数	フォーマットの詳細
GL_RGBA4	16	RGBA 各成分とも 4 bit
GL_RGB5_A1	16	RGB 各成分が 5 bit、アルファ成分が 1 bit
GL_RGB565	16	RB 成分が各 5 bit、G 成分が 6 bit。アルファ成分なし
GL_RGB8_OES	24	RGB 各成分とも 8 bit。アルファ成分なし
GL_RGBA8_OES	32	RGBA 各成分とも 8 bit

ピクセルフォーマットのビット数が大きくなる変換はできません。つまり、24 bit のフォーマットから 32 bit のフォーマットへの変換、16 bit のフォーマットから 24 bit または 32 bit のフォーマットへの変換はできません。

aamode にはアンチエイリアスフィルタのモードを指定します。指定可能なモードは以下の 3 種類です。表中の幅と高さは、転送元に必要なサイズが転送先の何倍以上であるかを示したものです。

表 4-23. アンチエイリアスの指定

定義	アンチエイリアス	幅	高さ
NN_GX_ANTIALIASE_NOT_USED	アンチエイリアスなし	等倍	等倍
NN_GX_ANTIALIASE_2x1	2x1 アンチエイリアスで転送	2 倍	等倍
NN_GX_ANTIALIASE_2x2	2x2 アンチエイリアスで転送	2 倍	2 倍

yflip にはイメージを転送するときに縦方向のフリップを有効にするかどうかを指定します。GL_TRUE(または 0 以外の値)を指定した場合はフリップが行われ、GL_FALSE(または 0)を指定した場合はフリップが行われません。

blocksize には、転送元イメージのブロックサイズを 8 または 32 で指定します。

表 4-24. nngxAddB2LTransferCommand() が生成するエラー

エラー	原因
GL_ERROR_807C_DMP	オブジェクト名が 0 のコマンドリストをバインドしている、またはコマンドリクエストのキューに空きがない
GL_ERROR_807D_DMP	*srcaddr* または *dstaddr* が 16 バイトアライメントではない
GL_ERROR_807E_DMP	*blocksize* に 8 または 32 以外の値を指定している
GL_ERROR_807F_DMP	*aamode* に不正な値を指定している
GL_ERROR_8080_DMP	*srcformat* および *dstformat* に不正な値を指定している
GL_ERROR_8081_DMP	*srcformat* のピクセルサイズより *dstformat* のピクセルサイズが大きい
GL_ERROR_8082_DMP	*srcwidth*、*srcheight*、*dstwidth*、*dstheight* のいずれかに不正な値を指定した
GL_ERROR_8083_DMP	転送先イメージが転送元イメージより幅または高さのピクセル数が大きくなるような指定をした
GL_ERROR_80B7_DMP	転送元イメージの幅または高さのピクセル数に最小値未満の値を指定した
GL_ERROR_80B8_DMP	転送先イメージの幅または高さのピクセル数に最小値未満の値を指定した

4.1.17. リニアイメージからブロックイメージへの変換転送コマンドの追加

nngxAddL2BTransferCommand() の呼び出しで、リニアイメージをブロックイメージに変換して転送を行うコマンドがコマンドリストに追加されます。nngxTransferLinearImage() でも同様の機能が提供されていますが、この関数は、より汎用的な機能を提供します。また、3D コマンドの区切りコマンドを追加せず、転送リクエストコマンドのみを追加する点が異なります。

コード 4-23. リニアイメージからブロックイメージへの変換転送コマンドを追加する関数

```
void nngxAddL2BTransferCommand(
        const GLvoid* srcaddr, GLvoid* dstaddr,
        GLsizei width, GLsizei height, GLenum format, GLsizei blocksize);
```

srcaddr には転送元(リニアイメージ)のアドレスを指定します。*dstaddr* には転送先(ブロックイメージ)のアドレスを指定します。*srcaddr* と *dstaddr* は、ともに 16 バイトアライメントでなければなりません。

width と *height* には、それぞれ転送元および転送先のイメージの幅と高さをピクセル数で指定します。イメージの幅と高さは、転送元と転送先で同じでなければならず、128 以上かつブロックサイズ(8 または 32)の倍数でなければなりません。さらに、転送先イメージのピクセルサイズが 24 bit の場合は、ブロックサイズが 8 であっても、イメージの幅は 32 の倍数でなければなりません。*width* または *height* に 0 が指定されていると、コマンドは追加されません。

format には転送されるイメージのピクセルフォーマットを指定します。指定可能なピクセルフォーマットは nngxAddB2LTransferCommand() と同じです(表 4-22)。転送元と転送先のイメージは同じピクセルフォーマットでなければなりません。ただし、24 bit のフォーマットの場合、ハードウェアによる 24 bit から 24 bit への転送がサポートされていないため、転送元イメージが 32 bit のフォーマットでなければなりません。その場合、転送元データの 4 バイトごとに最初の 1

バイトが切り捨てられて転送されます。

blocksize には変換先イメージのブロックサイズを 8 または 32 から指定します。

表 4-25. nngxAddL2BTransferCommand() が生成するエラー

エラー	原因
GL_ERROR_806F_DMP	オブジェクト名が 0 のコマンドリストをバインドしている、またはコマンドリクエストのキューに空きがない
GL_ERROR_8070_DMP	*srcaddr* または *dstaddr* が 16 バイトアライメントではない
GL_ERROR_8071_DMP	*blocksize* に 8 または 32 以外の値を指定している
GL_ERROR_8072_DMP	*width*、*height* のいずれかに不正な値を指定した
GL_ERROR_8073_DMP	*format* に不正な値を指定している

4.1.18. ブロックイメージ転送コマンドの追加

nngxAddBlockImageCopyCommand() の呼び出しで、ブロックイメージの転送を行うコマンドがコマンドリストに追加されます。追加されるコマンドにより、描画されたレンダーバッファやテクスチャ間での画像のコピーを行うことができます。転送サイズとスキップサイズの組を指定した転送を行うため、転送元イメージの部分領域を切り出したり、転送先イメージの部分領域にはめ込んだりすることができます。この関数はブロックフォーマットのイメージを転送することを主な目的としていますが、フォーマットの変換を行わないため、各種データの転送に利用することができます。

コード 4-24. ブロックイメージ転送コマンドを追加する関数

```
void nngxAddBlockImageCopyCommand(
        const GLvoid* srcaddr, GLsizei srcunit, GLsizei srcinterval,
        GLvoid* dstaddr, GLsizei dstunit, GLsizei dstinterval,
        GLsizei totalsize);
```

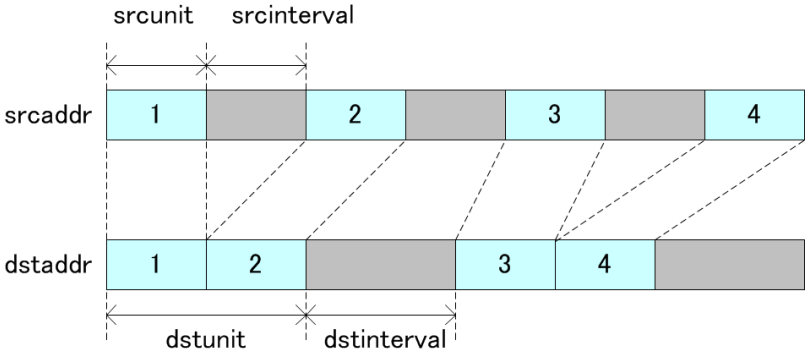
srcaddr には転送元の開始アドレスを指定します。*dstaddr* には転送先の開始アドレスを指定します。*srcaddr* と *dstaddr* は、ともに 16 バイトアライメントでなければなりません。

totalsize には転送を行うサイズの合計をバイト単位で指定します。*totalsize* は 16 の倍数でなければなりません。

srcunit と *srcinterval* には、それぞれ転送元の読み込み単位サイズとスキップサイズをバイト単位で指定します。*srcunit* バイトのデータ転送と *srcinterval* バイトの読み込みアドレスのスキップが交互に繰り返し行われ、転送されたサイズが *totalsize* に達したときに転送が終了します。*srcinterval* に 0 を指定した場合スキップは行われず、*totalsize* バイトの連続領域の読み込みが行われることになります。*srcinterval* に 0 以外を指定した場合は読み込みとスキップを繰り返すため、転送元イメージの部分領域を切り出して転送することができます。

dstunit と *dstinterval* には、それぞれ転送先の書き込み単位サイズとスキップサイズをバイト単位で指定します。*dstunit* バイトのデータ書き込みと、*dstinterval* バイトの書き込みアドレスのスキップが交互に繰り返し行われ、転送されたサイズが *totalsize* に達したときに転送が終了します。*dstinterval* に 0 を指定した場合スキップは行われず、*totalsize* バイトの連続領域の書き込みが行われることになります。*dstinterval* に 0 以外を指定した場合は書き込みとスキップを繰り返すため、転送先イメージの部分領域にイメージをはめ込むような転送を行うことができます。

図 4-3. ブロックイメージ転送の例



srcunit、*srcinterval*、*dstunit*、*dstinterval* は 16 の倍数でなければなりません。また、0x100000 以上の値や負の値は指定することができません。

描画結果などのブロックイメージを転送する際は、転送イメージの先頭アドレス(転送元と転送先)が画像の左上(OpenGL ES では左下)である点や、ブロックサイズ 8 のフォーマットであれば 8×8 ピクセルのブロック単位でデータが配置されている点などに注意して、引数を設定してください。ブロックフォーマットの詳細については、「7.10. PICA ネイティブフォーマット」を参照してください。

表 4-26. nngxAddBlockImageCopyCommand() が生成するエラー

エラー	原因
GL_ERROR_8074_DMP	オブジェクト名が 0 のコマンドリストをバインドしている、またはコマンドリクエストのキューに空きがない
GL_ERROR_8075_DMP	*srcaddr* または *dstaddr* が 16 バイトアライメントではない
GL_ERROR_8076_DMP	*totalsize* が 16 の倍数ではない
GL_ERROR_8077_DMP	*srcunit*、*srcinterval*、*dstunit*、*dstinterval* のいずれかに不正な値を指定した

4.1.19. メモリフィルコマンドの追加

nngxAddMemoryFillCommand() の呼び出しで、指定した領域を指定したデータで埋める(フィルする)コマンドがコマンドリストに追加されます。この関数で追加されるコマンドは、カラーバッファとデプスバッファ(ステンシルバッファ)をクリアする場合などに使用します。glClear() でも同様の機能を提供していますが、この関数は、より汎用的な機能を提供します。独立したパラメータの指定が可能な 2 つのチャンネル設定により、サイズの異なる 2 つの領域を同時にクリアすることができます。

コード 4-25. メモリフィルコマンドを追加する関数

```
void nngxAddMemoryFillCommand(
        GLvoid* startaddr0, GLsizei size0, GLuint data0, GLsizei width0,
        GLvoid* startaddr1, GLsizei size1, GLuint data1, GLsizei width1);
```

startaddr0、*size0*、*data0*、*width0* がチャンネル 0 の設定、*startaddr1*、*size1*、*data1*、*width1* がチャンネル 1 の設定です。チャンネル 0 とチャンネル 1 によるメモリフィルは同時に実行されます。そのため、チャンネル 0 とチャンネル 1 で指定された領域が重なる場合、どちらの結果が最終的に反映されるのかは不定です。

startaddr0 と *startaddr1* には領域の先頭アドレスを指定します。アドレスは 16 バイトアライメントでなければなりませ

ん。アドレスに 0 を指定した場合は、そのチャンネルを使用しません。*startaddr0* に 0 を指定した場合、*size0*、*data0*、*width0* の指定に関するエラーはチェックされません。*startaddr1* に 0 を指定した場合、*size1*、*data1*、*width1* の指定に関するエラーはチェックはされません。

size0 と *size1* には領域のサイズをバイト単位で指定します。サイズは 16 の倍数でなければなりません。

data0 と *data1* にはフィルパターンのデータを指定します。領域には指定されたデータが繰り返し格納されます。

width0 と *width1* にはフィルパターンのビット幅を指定します。指定可能なビット幅は 16、24、32 のいずれかです。16 を指定した場合は、データのビット [15 : 0] を使って 16 bit 単位で埋められます。24 を指定した場合は、データのビット [23 : 0] を使って 24 bit 単位で埋められます。32 を指定した場合は、データのビット [31 : 0] を使って 32 bit 単位で埋められます。

下表は、レンダーバッファのフォーマットによるフィルパターンの指定(ビット幅、各成分値)をまとめたものです。

表 4-27. レンダーバッファのフォーマットによるフィルパターン

レンダーバッファのフォーマット	ビット幅	R / D	G / S	B	A
GL_RGBA8_OES	32	[31 : 24] 0 ～ 255	[23 : 16] 0 ～ 255	[15 : 8] 0 ～ 255	[7 : 0] 0 ～ 255
GL_RGB8_OES	24	[23 : 16] 0 ～ 255	[15 : 8] 0 ～ 255	[7 : 0] 0 ～ 255	-
GL_RGBA4	16	[15 : 12] 0 ～ 15	[11 : 8] 0 ～ 15	[7 : 4] 0 ～ 15	[3 : 0] 0 ～ 15
GL_RGB5_A1	16	[15 : 11] 0 ～ 31	[10 : 6] 0 ～ 31	[5 : 1] 0 ～ 31	[0 : 0] 0 ～ 1
GL_RGB565	16	[15 : 11] 0 ～ 31	[10 : 5] 0 ～ 63	[4 : 0] 0 ～ 31	-
GL_DEPTH24_STENCIL8_EXT	32	[23 : 0]	[31 : 24]	-	-
GL_DEPTH_COMPONENT24_OES	24	[23 : 0]	-	-	-
GL_DEPTH_COMPONENT16	16	[15 : 0]	-	-	-

表 4-28. nngxAddMemoryFillCommand() が生成するエラー

エラー	原因
GL_ERROR_8078_DMP	オブジェクト名が 0 のコマンドリストをバインドしている、またはコマンドリクエストのキューに空きがない
GL_ERROR_8079_DMP	*startaddr0* または *startaddr1* が 16 バイトアライメントではない
GL_ERROR_807A_DMP	*size0* または *size1* が 16 の倍数ではない
GL_ERROR_807B_DMP	*width0* または *width1* に不正な値を指定した

4.1.20. 3D コマンドバッファのポインタ移動

nngxMoveCommandbufferPointer() の呼び出しで、現在バインドされているコマンドリストの 3D コマンドバッファのポインタ(3D コマンドの実行位置)を移動させることができます。

コード 4-26. 3D コマンドバッファのポインタを移動させる関数

```
void nngxMoveCommandbufferPointer(GLint offset);
```

offset にはポインタの移動量を**バイト単位で**指定します。

コマンドリストがバインドされていない場合や、ポインタが 3D コマンドバッファの領域外に移動してしまう場合は GL_ERROR_8061_DMP のエラーを生成します。

4.1.21. ジャンプコマンドの追加

nngxAddJumpCommand() の呼び出しで、指定された領域にある 3D コマンドを実行するジャンプコマンドが、現在バインドされているコマンドリストに追加されます。ジャンプコマンドを利用すると、割り込みを発生させることなく、別のコマンドリストを続けて実行することができます。

この関数は PICA レジスタのコマンドバッファ実行レジスタを利用しています。チャンネル 0 のみを使用していますので、2 つのレジスタ(0x0238 と 0x023A)の内容が関数の実行時に書き換えられます。動作の詳細については、「3DS プログラミングマニュアル – グラフィックス応用編」の「コマンドバッファ実行レジスタ(0x0238 ～ 0x023D)」と「コマンドバッファの連続実行」を参照してください。

コード 4-27. ジャンプコマンドを追加する関数

```
void nngxAddJumpCommand(const GLvoid* bufferaddr, GLsizei buffersize);
```

bufferaddr と *buffersize* には、続けて実行されるコマンドバッファのアドレスとバッファのサイズを指定します。*bufferaddr* と *buffersize* は、ともに 16 の倍数である必要があります。

ジャンプ先(*bufferaddr* と *buffersize* で指定されたコマンドリスト)のコマンドバッファの内容は、現在バインドされているコマンドリストのコマンドバッファにコピーされません。ジャンプコマンドはコマンドバッファの実行アドレスを変更し、ジャンプ先のコマンドバッファを直接実行します。そのため、ジャンプ先の領域のキャッシュがフラッシュされていることをアプリケーションで保証しなければなりません。

ジャンプ先で最後に実行されるコマンドは、区切りコマンド(nngxSplitDrawCmdlist() で追加される区切りコマンド設定レジスタへの書き込みコマンド)でなければなりません。区切りコマンドの代わりにジャンプコマンドでジャンプを繰り返すことができます。ただし、ジャンプを繰り返した場合は、最終的に実行されるコマンドバッファの最後に区切りコマンドが必要です。

この関数は、3D 実行コマンドのコマンドリクエストを追加します。nngxFlush3DCommand() などでコマンドバッファがフラッシュされた直後に呼び出しても意味がないため、GL_ERROR_809A_DMP のエラーが生成されます。フラッシュされた直後のコマンドバッファに 3D コマンドを追加する場合は nngxAdd3DCommand() を呼び出してください。

表 4-29. nngxAddJumpCommand() が生成するエラー

エラー	原因
GL_ERROR_8096_DMP	オブジェクト名が 0 のコマンドリストをバインドしている
GL_ERROR_8097_DMP	*buffersize* が 0 以下
GL_ERROR_8098_DMP	*buffersize* が 16 の倍数ではない
GL_ERROR_8099_DMP	*bufferaddr* が 16 の倍数ではない
GL_ERROR_809A_DMP	現在バインドされているコマンドリストのコマンドバッファがフラッシュされた直後
GL_ERROR_809B_DMP	この関数で追加されるコマンドリクエストによってキューがあふれる

GL_ERROR_809C_DMP	この関数で追加されるコマンドによってコマンドバッファがあふれる

4.1.22. サブルーチンコマンドの追加

nngxAddSubroutineCommand() の呼び出しで、指定された領域にある 3D コマンドを実行するジャンプコマンドとジャンプ元のコマンドバッファに戻るためのアドレス情報設定用のコマンドが、現在バインドされているコマンドリストに追加されます。サブルーチンコマンドを利用すると、割り込みを発生させることなく、別のコマンドリストをサブルーチンのように実行することができます。

この関数は PICA レジスタのコマンドバッファ実行レジスタを利用しています。すべてのチャンネルを使用していますので、4 つのレジスタ(0x0238 ～ 0x023B)の内容が関数の実行時に書き換えられます。動作の詳細については、「3DS プログラミングマニュアル – グラフィックス応用編」の「コマンドバッファ実行レジスタ(0x0238 ～ 0x023D)」と「同じコマンドバッファの繰り返し実行」を参照してください。

コード 4-28. サブルーチンコマンドを追加する関数

```
void nngxAddSubroutineCommand(const GLvoid* bufferaddr, GLsizei buffersize);
```

bufferaddr と *buffersize* には、続けて実行されるコマンドバッファのアドレスとバッファのサイズを指定します。*bufferaddr* と *buffersize* は、ともに 16 の倍数である必要があります。

ジャンプ先(*bufferaddr* と *buffersize* で指定されたコマンドリスト)のコマンドバッファの内容は、現在バインドされているコマンドリストのコマンドバッファにコピーされません。ジャンプコマンドはコマンドバッファの実行アドレスを変更し、ジャンプ先のコマンドバッファを直接実行します。そのため、ジャンプ先の領域のキャッシュがフラッシュされていることをアプリケーションで保証しなければなりません。

ジャンプコマンドはチャンネル 0 で実行され、元のコマンドバッファに戻るコマンドはチャンネル 1 で実行されます。そのため、ジャンプ先で最後に実行されるコマンドはチャンネル 1 のキックコマンド(コマンドバッファ実行レジスタ 0x023D への書き込みコマンド)でなければなりません。代わりにジャンプコマンドで別のコマンドバッファへジャンプすることもできますが、ジャンプに使用するチャンネルは 0 でなければならず、最終的に実行されるコマンドバッファの最後にチャンネル 1 のキックコマンドが必要です。また、その途中でチャンネル 1 のアドレス設定(0x0239、0x023B)を書き換えないでください。この関数が追加するのはジャンプコマンド(チャンネル 0)とアドレス設定(チャンネル 1)です。チャンネル 1 のキックコマンドやサブルーチン内でのジャンプコマンドはアプリケーションが配置しなければなりません。

この関数は、3D 実行コマンドのコマンドリクエストを追加しません。この関数を呼び出したあとも続けてコマンドを蓄積し、nngxFlush3DCommand() などでコマンドバッファをフラッシュしてから実行してください。コマンドバッファがフラッシュされるまで、この関数が追加するチャンネル 1 のサイズ設定レジスタ(0x023B)への書き込まれる値は未確定の状態ですので、コマンドバッファをフラッシュする前にコピーした内容を再利用したときの動作は不定です。

表 4-30. nngxAddSubroutineCommand() が生成するエラー

エラー	原因
GL_ERROR_809D_DMP	オブジェクト名が 0 のコマンドリストをバインドしている
GL_ERROR_809E_DMP	*buffersize* が 0 以下
GL_ERROR_809F_DMP	*buffersize* が 16 の倍数ではない
GL_ERROR_80A0_DMP	*bufferaddr* が 16 の倍数ではない
GL_ERROR_80A1_DMP	この関数で追加されるコマンドによってコマンドバッファがあふれる

4.2. コマンドリクエストの種類

コマンドリストにキューイングされるコマンドリクエストには、以下のコマンドがあります。

DMA 転送コマンド

メインメモリから VRAM への、テクスチャイメージや頂点バッファの DMA 転送を行うコマンドです。

`glTexImage2D()` などのテクスチャ領域確保や `glBufferData()` などの頂点バッファ領域確保でキューイングされます。

3D 実行コマンド

3D コマンドバッファに蓄積されている 3D コマンドを、コマンドセット 1 つ分実行するコマンドです。

`glClear()` や `glTexImage2D()` などの呼び出しでバッファ読み込み終了の 3D コマンドが書き込まれ、蓄積していた 3D コマンドバッファを 1 つの 3D 実行コマンドとしてキューイングされます。

`nngxSplitDrawCmdlist()` により、任意のタイミングで 3D 実行コマンドをキューイングさせることもできます。

メモリフィルコマンド

GPU のメモリフィル機能によって、VRAM 上に確保されている領域を指定したデータパターンでクリアするコマンドです。

レンダーバッファを指定し、`glClear()` を呼び出したときにキューイングされます。`glClear()` の実行には、メモリフィル以外にも 3D コマンドの実行が必要となります。つまり `glClear()` を呼び出すと、3D コマンドバッファに `glClear()` 用とバッファ読み込み終了の 3D コマンドが書き込まれ、3D 実行コマンドとメモリフィルコマンドが続けてキューイングされます。

ポスト転送コマンド

GPU のポストフィルタ機能によって、PICA ブロックフォーマットで描画されたイメージを LCD が読み込めるリニアフォーマットに変換するコマンドです。

`nngxTransferRenderImage()` を呼び出したときにキューイングされます。`nngxSplitDrawCmdlist()` を事前に呼び出して 3D コマンドバッファの読み込みを終了させていなければ、バッファ読み込み終了コマンドの書き込みと 3D 実行コマンドのキューイングのあとに、このコマンドがキューイングされることになります。

レンダーテクスチャ転送コマンド

GPU の描画結果をテクスチャイメージとしてメモリにコピーするコマンドです。

`glCopyTexImage2D()` または `glCopyTexSubImage2D()` を呼び出したときにキューイングされます。

`nngxSplitDrawCmdlist()` を事前に呼び出して 3D コマンドバッファの読み込みを終了させていなければ、バッファ読み込み終了コマンドの書き込みと 3D 実行コマンドのキューイングのあとに、このコマンドがキューイングされることになります。

4.3. 3D コマンドバッファのパフォーマンスを最適化する手法

3D コマンドバッファ実行時のパフォーマンスを最適化する手法を説明します。

4.3.1. アドレスとサイズによるロード速度の変化

3D コマンドバッファのアドレスとサイズが、実行時のロード速度に影響する場合があります。

3D コマンドバッファの実行方法には、コマンドリクエストにキューイングされた 3D 実行コマンドでの実行と、コマンドバッファ

実行レジスタでの実行の 2 種類があります。

3D 実行コマンドにより実行される場合は、nngxFlush3DCommand() や nngxSplitDrawCmdlist() で区切りコマンドが追加された直後のアドレスから、次に区切りコマンドが追加されるまでのサイズにより影響を受けます。3D コマンドバッファに蓄積中の 3D コマンドのアドレスは、nngxGetCmdlistParameteri() の *pname* に NN_GX_CMDLIST_CURRENT_BUFADDR を渡して取得できます。

コマンドバッファ実行レジスタにより実行される場合は、nngxAddJumpCommand() で追加されるコマンドバッファのアドレスとサイズ、nngxAddSubroutineCommand() で追加されるサブルーチンとしてのコマンドバッファおよびサブルーチンから呼び出し元に戻るために実行されるコマンドバッファのアドレスとサイズにより影響を受けます。

3D コマンドバッファのアドレスが 128 バイトアライメントの場合、サイズが 256 バイト単位(256 バイト、512 バイト、768 バイトなど)ならば転送速度が向上する可能性があります。

3D コマンドバッファのアドレスが 128 バイトアライメントではない場合、直前の 128 バイトアライメントのアドレスから 3D コマンドバッファの終端までのサイズが 256 バイト単位である場合に速度が向上する可能性があります。例えば 3D コマンドバッファのアドレスとサイズがそれぞれ 0x20000010 と 0x1F0 だった場合、直前の 128 バイトアライメントのアドレスは 0x10 だけ手前の 0x20000000 です。そこから終端までは 0x1F0 + 0x10 で 0x200 となり、256 バイト単位であると言えます。

GPU の実装上の理由により、3D コマンドバッファのアドレスとサイズには上記のような特性がありますが、格納場所や 3D コマンドの内容、ほかのモジュールとのメモリアクセスの競合などにより、大きな効果が得られない場合があります。

4.3.2. サブルーチン実行の利用

3D コマンドバッファのサブルーチン実行を利用することで、パフォーマンスが向上する可能性があります。

4.3.2.1. 概要

3D コマンドバッファのサブルーチン実行とは、コマンドバッファ実行レジスタを使用した実行方法です。3D コマンドを一続きの 3D コマンドバッファに格納して実行する通常の手法とは異なり、別の場所に格納してあるコマンドバッファをコマンドバッファのアドレスジャンプ機能を使って連続で実行させる手法です。3D コマンドバッファのアドレスを特定のアドレスへジャンプさせ、ジャンプ先の 3D コマンドバッファを実行したあと、ジャンプ元へ戻るという制御を行うことから、この手法をコマンドバッファのサブルーチン実行と呼びます。

コマンドバッファのサブルーチン実行を行うための具体的な方法については、「4.1.22. サブルーチンコマンドの追加」および「3DS プログラミングマニュアル - グラフィックス応用編」の「コマンドバッファ実行レジスタ(0x0238 ～ 0x023D)」を参照してください。

4.3.2.2. 動作への影響

コマンドバッファをサブルーチン化して実行することには、以下のメリットがあります。

- サブルーチン化されたコマンドバッファへのジャンプコマンドなどを格納するだけでよいため、3D コマンドのコピーに必要な CPU 処理を削減できます。つまり、参照テーブルのデータやシェーダプログラムのロードなど、ある程度サイズが大きく、頻繁に設定が行われる処理で効果を発揮します。
- サブルーチン化されたコマンドバッファはカレントの 3D コマンドバッファへコピーされずに GPU から直接参照されるため、コマンドバッファの総サイズを縮小することができます。
- サブルーチン化されたコマンドバッファを VRAM に格納すれば、GPU からコマンドバッファへのアクセスがメインメモリ(デバイスメモリ)に比べて高速になります。そのため、コマンドバッファへのメモリアクセスがボトルネックとなっている場合に、システム全体の処理速度の向上が期待できます。

一方、以下のデメリットがあります。

- ジャンプコマンドによるアドレスの切り替えで、メモリアクセスのオーバーヘッドが発生します。そのため、サブルーチン化の粒度を小さくして頻繁に呼び出すような実装では、GPU の処理速度が低下する可能性があります。

サブルーチン化による処理速度への影響は、メモリアクセスの競合などに左右されるため、実際のアプリケーションの実装に強く依存します。

4.3.2.3. 格納場所

コマンドバッファのアクセス速度はメインメモリ(デバイスメモリ)より VRAM の方が高速ですので、サブルーチン化されたコマンドバッファは VRAM に格納することを推奨します。

ジャンプコマンドによるサブルーチン化されたコマンドバッファの実行ではメモリアクセスのオーバーヘッドが発生しますが、実行するコマンドバッファが VRAM に格納されている場合はオーバーヘッドが軽減されます。

コマンドバッファを VRAM に格納するには、デバイスメモリ上に生成したあとに、`nngxAddVramDmaCommand()` を利用して VRAM に DMA 転送する必要があります。VRAM への DMA 転送については「4.1.13. DMA 転送コマンドの追加」を参照してください。

4.3.2.4. 実行処理とアクセス処理のバランス

サブルーチン化されたコマンドバッファの内容によっては、ボトルネックとなる処理が 3D コマンドのアクセス処理と実行処理の間で移動する場合があります。

3D コマンドがラスタライゼーションモジュール以降のモジュール(ラスタライゼーションモジュールを含む)のレジスタライトコマンドである場合、1 個の 3D コマンドを処理するのに 2 サイクルかかるため、実行処理の比重が大きくなります。3D コマンドがバーストコマンドで構成されている場合は、アクセス処理に対する実行処理の比重はさらに大きくなります。このような場合、実行処理がボトルネックとなり、サブルーチン化で発生するメモリアクセスの処理コストが隠蔽されることになります。

3D コマンドが、ラスタライゼーションモジュールより前のモジュール(ラスタライゼーションモジュールを含まない)のレジスタライトコマンドである場合、1 個の 3D コマンドを処理するのに 1 サイクルかかるため、前者よりも実行処理の比重が小さくなります。この場合、アクセス処理がボトルネックになりやすく、サブルーチン化で発生するメモリアクセスの処理コストが全体のパフォーマンスに影響しやすくなります。

各モジュールの位置関係については「2.2. レンダリングパイプライン」を参照してください。

5. シェーダプログラム

3DS での 3D グラフィックスは、シェーダプログラムによってグラフィックスのパイプラインをカスタマイズし、様々なグラフィックスエフェクトを制御することができます。

3DS のシェーダプログラムには、頂点処理、ジオメトリ生成、フラグメント処理の 3 種類が存在します。

この内、頂点処理のシェーダプログラム(以降、頂点シェーダ)にはユーザーがプログラミングした独自のものを使用することができます。

ジオメトリ生成のシェーダプログラム(以降、ジオメトリシェーダ)は SDK で提供されており、それらのジオメトリシェーダと頂点シェーダをリンクして使用することができます。

フラグメント処理のシェーダプログラム(以降、フラグメントシェーダ)はプログラミングすることはできません。フラグメント処理は固定のパイプラインで実装されていますが、予約ユニフォームによる制御が可能です。以降、このフラグメント処理のパイプラインを"**予約フラグメント処理**"、シェーダプログラムを"**予約フラグメントシェーダ**"と記述します。

5.1. シェーダの作成

ユーザーが作成可能なシェーダプログラムは頂点シェーダのみです。頂点処理に関する一連の手続きは OpenGL ES 2.0 の仕様に従っていますが、いくつかの機能はサポートされていません。

シェーダプログラムは PICA グラフィックスコア独自のアセンブリ言語で記述します。アプリケーションがシェーダプログラムを使用するには、専用のアセンブラとリンカによって生成されたバイナリをロードし、アタッチしなければなりません。OpenGL ES 2.0 の仕様にある、`glShaderSource()` および `glCompileShader()` は実装されていません。

頂点シェーダの作成方法については、「8. 頂点シェーダ」および「頂点シェーダ リファレンスマニュアル」を参照してください。

5.2. シェーダのロード

「5.1. シェーダの作成」でも述べたように、シェーダプログラムはバイナリデータをロードしてからアタッチしなければなりません。まず、`glCreateShader()` でシェーダオブジェクトを生成します。

コード 5-1. glCreateShader() の定義

```
GLuint glCreateShader(GLenum type);
```

ロードするシェーダプログラムの種別によって *type* に渡す値が変わります。予約フラグメントシェーダは固定実装のためロードする必要がありません。

表 5-1. シェーダオブジェクトの種別

type	生成されるオブジェクト
GL_VERTEX_SHADER	ユーザーが作成した頂点シェーダ用のオブジェクト
GL_GEOMETRY_SHADER_DMP	SDK が提供するジオメトリシェーダ用のオブジェクト

シェーダプログラムのバイナリをメモリ上にロードし、`glShaderBinary()` で GPU と関連付けます。

コード 5-2. glShaderBinary() の定義

```
void glShaderBinary(GLint n, const GLuint* shaders, GLenum binaryformat,
                    const void* binary, GLint length);
```

shaders にはシェーダオブジェクトの配列、*n* には配列の要素数をそれぞれ渡します。アセンブラとリンカによって生成されたバイナリのみをロードすることができますので、*binaryformat* には GL_PLATFORM_BINARY_DMP を渡します。*binary* にはシェーダプログラムのバイナリをロードしたアドレス、*length* にはそのバイト数をそれぞれ渡します。

ロードされたシェーダプログラムはリンカに渡された順に配列に関連付けられます。配列の要素数および設定すべきシェーダオブジェクトの種別は、リンカが出力するマップファイルで確認することができます。詳しくは「頂点シェーダ リファレンスマニュアル」を参照してください。

5.3. シェーダのアタッチ

ロードされたシェーダプログラムをアプリケーションで使用するには、glCreateProgram() で生成したプログラムオブジェクトにシェーダプログラムをアタッチし、プログラムオブジェクトをリンクさせなければなりません。

コード 5-3. glCreateProgram() の定義

```
GLuint glCreateProgram(void);
```

OpenGL ES 2.0 とは異なり、プログラムオブジェクトはシェーダオブジェクトとは独立した 13 ビットの名前空間を持っています。そのため、同時に生成できるのは 8191 個までですが、生成済みのプログラムオブジェクトを glDeleteProgram() で破棄した場合には再度生成することができるようになります。

プログラムオブジェクトにシェーダプログラムをアタッチするには glAttachShader() を使用します。

コード 5-4. glAttachShader() の定義

```
void glAttachShader(GLuint program, GLuint shader);
```

program には glCreateProgram() の返り値を、*shader* にはシェーダオブジェクトを渡します。

頂点シェーダとジオメトリシェーダはロードしたバイナリからアタッチしましたが、ロードする必要のなかった予約フラグメントシェーダは、*shader* に GL_DMP_FRAGMENT_SHADER_DMP を渡してアタッチします。

1 つのプログラムオブジェクトには、頂点シェーダ、ジオメトリシェーダ、予約フラグメントシェーダをそれぞれ 1 つずつアタッチすることができます。つまり、ポイントシェーダ(ジオメトリシェーダ)をアタッチした後に続けてラインシェーダ(ジオメトリシェーダ)をアタッチした場合は、ラインシェーダだけが有効となります。

プログラムオブジェクトをリンクさせるには glLinkProgram() を使用します。

コード 5-5. glLinkProgram() の定義

```
void glLinkProgram(GLuint program);
```

複数のプログラムオブジェクトをリンクすることができます。ただし、リンクされたプログラムオブジェクトに glAttachShader() で別のシェーダプログラムをアタッチした場合は、再度 glLinkProgram() でリンクする必要があります。頂点シェーダとジオメトリシェーダで使用するユニフォームの合計数が 2048 を超えるプログラムオブジェクトのリンクは失敗します。

5.4. シェーダの使用

リンクされたシェーダプログラムを 3D 処理のパイプラインに適用するには glUseProgram() を使用します。

コード 5-6. glUseProgram() の定義

```
void glUseProgram(GLuint program);
```

この関数により、リンクされている複数のシェーダプログラムを切り替えて使用することができます。

OpenGL ではシェーダプログラムの正当性を glValidateProgram() で確認することができましたが、3DS では関数を呼び出しても何も行われません。

コード 5-7. glValidateProgram() の定義

```
void glValidateProgram(GLuint program);
```

5.5. シェーダのデタッチ

不要になったシェーダプログラムは glDetachShader() でデタッチすることができます。

コード 5-8. glDetachShader() の定義

```
void glDetachShader(GLuint program, GLuint shader);
```

5.6. シェーダの破棄

不要になったシェーダオブジェクトは glDeleteShader() で破棄することができます。

コード 5-9. glDeleteShader() の定義

```
void glDeleteShader(GLuint shader);
```

5.7. シェーダへの問い合わせ

有効・無効の判定やパラメータの取得など、シェーダに関連する情報はプログラムオブジェクトとシェーダオブジェクトに対する問い合わせで取得することができます。

5.7.1. 有効・無効の判定

プログラムオブジェクトやシェーダオブジェクトが有効であるかどうかを glIsProgram() と glIsShader() で問い合わせることができます。

コード 5-10. glIsProgram()、glIsShader() の定義

```
GLboolean glIsProgram(GLuint program);
GLboolean glIsShader(GLuint shader);
```

これらの関数は、引数で渡されたプログラムオブジェクトやシェーダオブジェクトが有効であれば GL_TRUE を、そうでなければ GL_FALSE を返します。

5.7.2. アタッチされているシェーダオブジェクトの取得

プログラムオブジェクトにアタッチされているシェーダオブジェクトを glGetAttachedShaders() で取得することができます。

コード 5-11. glGetAttachedShaders() の定義

```
void glGetAttachedShaders(GLuint program, GLsizei maxcount, GLsizei* count,
                          GLuint* shaders);
```

program で指定されたプログラムオブジェクトにアタッチされているシェーダオブジェクトの一覧を、*shaders* に指定された配列に格納します。*maxcount* には *shaders* に指定した配列のサイズを指定してください。*program* に不正な値を指定した場合や *maxcount* に負の値を指定した場合は GL_INVALID_VALUE のエラーが生成されます。

count には格納したシェーダオブジェクトの数が格納されます。NULL を指定した場合は格納されませんが、アタッチされているシェーダオブジェクトの数は、後述する glGetProgramiv() に GL_ATTACHED_SHADERS を渡して呼び出すことで取得することができます。

5.7.3. パラメータの取得

プログラムオブジェクトとシェーダオブジェクトのパラメータを glGetProgramiv() と glGetShaderiv() で取得することができます。

コード 5-12. glGetProgramiv()、glGetShaderiv() の定義

```
void glGetProgramiv(GLuint program, GLenum pname, GLint* params);
void glGetShaderiv(GLuint shader, GLenum pname, GLint* params);
```

これらの関数は *pname* に指定されたパラメータ名に対応するパラメータの値を *params* に格納します。*pname* に不正な値が指定された場合は GL_INVALID_ENUM エラーが生成されます。*program* および *shader* に不正な値が指定された場合は GL_INVALID_VALUE エラーが生成されます。

glGetProgramiv() の *pname* に指定可能なパラメータ名と *params* に格納されるパラメータを以下に示します。

表 5-2. glGetProgramiv() で指定可能なパラメータ名と格納される値

pname	params に格納される値
GL_DELETE_STATUS	プログラムオブジェクトが削除待ち状態ならば GL_TRUE が、そうでなければ GL_FALSE が格納されます。削除待ち状態へ遷移するのは、glUseProgram() で使用中のプログラムオブジェクトに対して glDeleteProgram() を呼び出したときです。
GL_LINK_STATUS	glLinkProgram() でプログラムオブジェクトのリンクが成功しているならば GL_TRUE が、そうでなければ GL_FALSE が格納されます。
GL_VALIDATE_STATUS	GL_LINK_STATUS を指定した場合と同じです。
GL_INFO_LOG_LENGTH	常に 0 が格納されます。
GL_ATTACHED_SHADERS	プログラムオブジェクトにアタッチされているシェーダオブジェクトの数が格納されます。
GL_ACTIVE_ATTRIBUTES	アクティブ状態の頂点属性の数が格納されます。
GL_ACTIVE_ATTRIBUTE_MAX_LENGTH	アクティブ状態の頂点属性のうち、最も長い名前の文字数が格納されます。文字数には終端文字(NULL)も含まれます。

GL_ACTIVE_UNIFORMS	アクティブ状態のユニフォームの数が格納されます。
GL_ACTIVE_UNIFORM_MAX_LENGTH	アクティブ状態のユニフォームのうち、最も長い名前の文字数が格納されます。文字数には終端文字(NULL)も含まれます。

glGetShaderiv() の *pname* に指定可能なパラメータ名と *params* に格納されるパラメータを以下に示します。

表 5-3. glGetShaderiv() で指定可能なパラメータ名と格納される値

pname	params に格納される値
GL_SHADER_TYPE	シェーダの種別が格納されます。
GL_DELETE_STATUS	シェーダオブジェクトが削除待ち状態ならば GL_TRUE が、そうでなければ GL_FALSE が格納されます。削除待ち状態へ遷移するのは、プログラムオブジェクトにアタッチされているシェーダオブジェクトに対して glDeleteShader() を呼び出したときです。
GL_COMPILE_STATUS	常に GL_FALSE が格納されます。
GL_INFO_LOG_LENGTH	常に 0 が格納されます。
GL_SHADER_SOURCE_LENGTH	常に 0 が格納されます。

6. 頂点バッファ

頂点バッファとは、頂点座標、頂点カラー、テクスチャ座標、頂点インデックスなどを格納するバッファです。多数の頂点を持つモデルなどを頂点シェーダに処理させる場合は頂点バッファを使用してください。頂点バッファを使用しない場合は CPU で重い処理(頂点アレイの並べ替えなど)が行われ、パフォーマンスが著しく低下する恐れがあります。

6.1. オブジェクトの生成

バッファオブジェクトを `glGenBuffers()` で生成します。

コード 6-1. glGenBuffers() の定義

```
void glGenBuffers(GLsizei n, GLuint* buffers);
```

n 個のバッファオブジェクトを生成して、*buffers* にその名前(オブジェクト名)を格納します。

6.2. オブジェクトの指定

バッファオブジェクトを関連付ける頂点バッファの種類を `glBindBuffer()` で指定します。この関数の呼び出し以降、各種頂点バッファへの処理は指定されたバッファオブジェクトに対して行われるようになります。

コード 6-2. glBindBuffer() の定義

```
void glBindBuffer(GLenum target, GLuint buffer);
```

target には頂点バッファの種類を指定します。*buffer* には `glGenBuffers()` で生成したバッファオブジェクトを指定します。*buffer* に未生成のオブジェクト名が指定された場合、そのオブジェクト名のバッファオブジェクトの生成も同時に行われます。

表 6-1. 頂点バッファの種類

target に指定する値	頂点バッファの種類
GL_ARRAY_BUFFER	頂点座標、頂点カラー、法線などのバッファ
GL_ELEMENT_ARRAY_BUFFER	`glDrawElements()` で使用するインデックスのバッファ
GL_VERTEX_STATE_COLLECTION_DMP	頂点ステートコレクション(「6.6. 頂点ステートコレクション」を参照)

6.3. バッファの確保

`glBufferData()` でバッファの領域を確保し、頂点データをロードします。

コード 6-3. glBufferData() の定義

```
void glBufferData(GLenum target, GLsizeiptr size, const void* data,
                  GLenum usage);
```

target に指定する値は `glBindBuffer()` の *target* と同じです(表 6-1)。

data と *size* には格納する頂点データとそのサイズを渡します。*data* に 0(NULL)を指定した場合は領域の確保のみが

行われます。

usage には GL_STATIC_DRAW を渡さなければなりません。

> **補足:** glBufferData() の *target* に特別なフラグを論理和で渡すことで、GPU のアクセス先や領域確保時の処理を指定することができます。詳細については、「3DS プログラミングマニュアル – グラフィックス応用編」「メインメモリに置かれたデータの使用」を参照してください。

6.4. バッファの書き換え

glBufferData() で確保したバッファの一部分を書き換えるには glBufferSubData() を使用します。

コード 6-4. glBufferSubData() の定義

```
void glBufferSubData(GLenum target, GLintptr offset, GLsizeiptr size,
                     const void* data);
```

target に指定する値は glBindBuffer() の *target* と同じです(表 6-1)。

offset には書き換える部分へのオフセットを指定します。

data と *size* には書き込むデータとそのサイズを渡します。

6.5. バッファの破棄

不要になったバッファオブジェクトは glDeleteBuffers() で破棄します。

コード 6-5. glDeleteBuffers() の定義

```
void glDeleteBuffers(GLsizei n, const GLuint* buffers);
```

buffers に格納されている *n* 個のオブジェクト名で指定されたバッファオブジェクトを破棄します。

6.6. 頂点ステートコレクション

頂点ステートコレクションは 3DS が独自に導入したもので、バッファオブジェクトと頂点バッファの関連付けを記録します。頂点ステートコレクションを使用することで、バッファオブジェクトと頂点バッファの関連付けと頂点属性の設定をまとめて行うことができます。

頂点ステートコレクションはバッファオブジェクトと名前空間を共有していますので、glGenBuffers()、glBindBuffer()、glDeleteBuffers() を使用して生成・指定・破棄を行います。

6.6.1. 頂点ステートコレクションの生成

頂点ステートコレクションは特殊なバッファオブジェクトとして動作します。そのためバッファオブジェクトと同じように、glGenBuffers() で頂点ステートコレクションとして使用するオブジェクトを生成します。

6.6.2. 頂点ステートコレクションの指定

頂点ステートコレクションとして使用するバッファオブジェクトを指定するには glBindBuffer() を使用します。*target* に GL_VERTEX_STATE_COLLECTION_DMP を渡して呼び出してください。デフォルトでは、名前 0(*buffer* に 0 を渡し

た状態）のオブジェクトが頂点ステートコレクションとなっています。

呼び出し後は、頂点バッファ（GL_ARRAY_BUFFER、GL_ELEMENT_ARRAY_BUFFER）への glBindBuffer() によるバッファオブジェクトの関連付けや、glEnableVertexAttribArray()、glDisableVertexAttribArray()、glVertexAttrib{1234}{fv}()、glVertexAttribPointer() による頂点属性の設定が頂点ステートコレクションに記録されていきます。同じ頂点バッファへの関連付けは上書きされ、頂点ステートコレクションへの記録はほかの頂点ステートコレクションに切り替えられるまで行われます。

頂点ステートコレクションを切り替えると、頂点バッファとバッファオブジェクトとの関連付けは頂点ステートコレクションに記録されているオブジェクトにすべて切り替わります。

6.6.3. 頂点ステートコレクションの破棄

バッファオブジェクトと同じように、glDeleteBuffers() で頂点ステートコレクションを破棄することができます。頂点ステートコレクションを破棄しても、記録されているバッファオブジェクトの頂点バッファとの関連付けに影響はありません。

使用中の頂点ステートコレクションへの glDeleteBuffers() の呼び出しでは、すぐに頂点ステートコレクションが破棄されません。ほかの頂点ステートコレクションに切り替えられるまで使用状態のまま残ります。デフォルトの頂点ステートコレクションは破棄できません。デフォルトの頂点ステートコレクションに対する glDeleteBuffers() の呼び出しは無視されます。

6.7. 頂点バッファの使用例

頂点バッファと glDrawElements() を使用して三角形を 1 つ描画するコード例を、配列の定義、バッファの確保、描画の 3 つの部分に分けて紹介します。

コード 6-6. 配列の定義

```
GLuint triIndexID, triArrayID;
GLushort triIndex[1 * 3] = { 0, 1, 2 };
GLfloat triVertex[3 * 4] =
{
     0.5f,  0.0f, 0.0f, 1.0f,
    -0.5f,  0.5f, 0.0f, 1.0f,
    -0.5f, -0.5f, 0.0f, 1.0f
};
GLfloat triVertexColor[3 * 3] =
{
    1.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 1.0f
};
```

コード 6-7. バッファの確保

```
glGenBuffers(1, &triArrayID);
glBindBuffer(GL_ARRAY_BUFFER, triArrayID);
glBufferData(GL_ARRAY_BUFFER, sizeof(triVertex) + sizeof(triVertexColor), 0,
        GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(triVertex), triVertex);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(triVertex), sizeof(triVertexColor),
        triVertexColor);
```

```
glGenBuffers(1, &triIndexID);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, triIndexID);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, 3 * sizeof(GLushort), triIndex,
GL_STATIC_DRAW);
```

コード 6-8. glDrawElements() による描画

```
glBindBuffer(GL_ARRAY_BUFFER, triArrayID);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, (GLvoid*)sizeof(triVertex));

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, triIndexID);
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_SHORT, 0);
```

6.8. 頂点データの配置に関する制約

頂点バッファに格納された頂点データを使用して描画を行う場合、glVertexAttribPointer() で設定する頂点データの配置には以下のハードウェアによる制約があります。これらの制約に抵触した場合、glDrawArrays() または glDrawElements() の呼び出し時に GL_INVALID_OPERATION のエラーを生成します。

- 各頂点データは、自身の型のサイズでアライメントされていなければなりません。
- 各頂点データのストライドは、同じ構造体に含まれる頂点データのうち、最もサイズの大きい型のサイズの倍数でなければなりません。
- 上記 2 つの制約を満たすために最小限必要なパディングより大きいサイズのパディングを頂点データのうしろに挿入する場合、その頂点データの最後尾から最も近い 4 バイト境界の次にある 4 バイト境界に続く頂点データを配置しなければなりません。

コンパイラが自動的にパディングを挿入するため、コーディングの段階で意識しなくても、上 2 つの制約には抵触しないようになっていることもあります。

下記のコード例では、extraPadding2 を挿入しなければ最後の制約に抵触してしまいます。

コード 6-9. 頂点データの配置に関する制約を回避するコード例

```
struct tagVertex
{
    GLshort position[3];
    GLshort extraPadding1;
    GLshort extraPadding2[2];
    GLshort color[4];
};
```

1 回の glDrawArrays() または glDrawElements() の呼び出しで使用する頂点属性および頂点インデックスには、頂点バッファを使用する頂点データと使用しない頂点データを混在させることができません。混在させた場合は GL_INVALID_OPERATION のエラーを生成します。

6.8.1. glDrawElements() のみの制約

以下の条件をすべて満たす場合、頂点アレイの格納方法による制限によって GL_INVALID_OPERATION のエラーが生成されます。

- 頂点バッファを使用している。
- 頂点属性を 12 個使用している。
- すべての頂点属性を頂点アレイとして使用している。

（すべての頂点属性に対して glEnableVertexAttribArray() が呼ばれている）

- glDrawElements() で描画している。

上記をすべて満たす場合、少なくとも 2 個の頂点属性をインターリーブドアレイとして配置しなければなりません。つまり、12 個の独立アレイを同時に頂点属性として使用することはできません。

> **補足:** 複数の頂点属性を構造体として定義し、頂点アレイを構造体の配列として配置したものをインターリーブドアレイと呼びます。これに対し、1 個の頂点属性を個別の配列として配置したものを独立アレイと呼びます。

7. テクスチャ

3DS で扱うことのできるテクスチャには、ポリゴンモデルに貼り付ける 2 次元テクスチャ、キューブマップテクスチャのほかに、シャドウで使用するシャドウテクスチャ、ガスレンダリングで使用するガステクスチャ、予約フラグメントシェーダで使用する参照テーブル(テクスチャとしては使用しません)が存在します。

この章では、それらのテクスチャを使用するために必要な手順と、ネイティブフォーマットの違いについて説明します。

7.1. テクスチャオブジェクトの生成

テクスチャと関連付けるテクスチャオブジェクトを glGenTextures() で生成します。

コード 7-1. glGenTextures() の定義

```
void glGenTextures(GLsizei n, GLuint* textures);
```

7.2. テクスチャオブジェクトの指定

テクスチャと関連付けるテクスチャオブジェクトを glBindTexture() で指定します。この関数の呼び出し以降、各種テクスチャへの処理は指定されたテクスチャオブジェクトに対して行われるようになります。テクスチャイメージのロードなど、テクスチャオブジェクトへの処理結果はオブジェクトが破棄されるまで保持されます。そのため、テクスチャオブジェクトを切り替えることで、テクスチャイメージを再ロードすることなくテクスチャを切り替えることができます。

コード 7-2. glBindTexture() の定義

```
void glBindTexture(GLenum target, GLuint texture);
```

target にテクスチャの種類を、*texture* に関連付けるテクスチャオブジェクトを渡します。*target* に下表以外の値を指定した場合は GL_INVALID_ENUM のエラーを生成します。

表 7-1. テクスチャの種類

target に指定する値	テクスチャの種類
GL_TEXTURE_2D	2 次元テクスチャ、シャドウテクスチャ、ガステクスチャ
GL_TEXTURE_CUBE_MAP_POSITIVE_X	キューブマップテクスチャ
GL_TEXTURE_CUBE_MAP_NEGATIVE_X	キューブマップテクスチャ
GL_TEXTURE_CUBE_MAP_POSITIVE_Y	キューブマップテクスチャ
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y	キューブマップテクスチャ
GL_TEXTURE_CUBE_MAP_POSITIVE_Z	キューブマップテクスチャ
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z	キューブマップテクスチャ
GL_LUT_TEXTURE*i*_DMP(*i* は 0 から 31)	参照テーブル(予約フラグメントシェーダで使用)
GL_TEXTURE_COLLECTION_DMP	テクスチャコレクション(「7.9. テクスチャコレクション」を参照)

7.3. テクスチャイメージのロード

一般的なテクスチャイメージである 2 次元テクスチャのほかにも、キューブマップテクスチャ、シャドウテクスチャ、ガステクスチャといった特殊なテクスチャも glTexImage2D() でロードします。**glTexSubImage2D() によるテクスチャの部分ロードはサポートしていません**。

コード 7-3. glTexImage2D() の定義

```
void glTexImage2D(GLenum target, GLint level, GLenum internalformat,
                  GLsizei width, GLsizei height, GLint border, GLenum format,
                  GLenum type, const void* pixels);
```

target には以下のいずれかを指定します。

表 7-2. glTexImage2D() の target に指定する値

target に指定する値	テクスチャの用途
GL_TEXTURE_2D	2 次元テクスチャ、シャドウテクスチャ、ガステクスチャ
GL_TEXTURE_CUBE_MAP_POSITIVE_X	キューブマップの +X 面のテクスチャ
GL_TEXTURE_CUBE_MAP_NEGATIVE_X	キューブマップの -X 面のテクスチャ
GL_TEXTURE_CUBE_MAP_POSITIVE_Y	キューブマップの +Y 面のテクスチャ
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y	キューブマップの -Y 面のテクスチャ
GL_TEXTURE_CUBE_MAP_POSITIVE_Z	キューブマップの +Z 面のテクスチャ
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z	キューブマップの -Z 面のテクスチャ

pixels の指す領域に格納されているテクスチャイメージのフォーマットを *format* と *type* の組み合わせで指定します。3DS が扱うことのできるフォーマットの一覧を下表に示します。バイト数の列は 1 テクセルあたりのバイト数です。バイト数の後に"*"のついたフォーマットは PICA ネイティブフォーマットであり、OpenGL 標準のフォーマットとはバイトオーダーが異なるものです。PICA ネイティブフォーマットの詳細は「7.10. PICA ネイティブフォーマット」で説明します。

表 7-3. format と type によるテクスチャイメージのフォーマット

format	type	形式	バイト数
GL_RGBA	GL_UNSIGNED_SHORT_4_4_4_4	RGBA4	2
GL_RGBA	GL_UNSIGNED_SHORT_5_5_5_1	RGBA5551	2
GL_RGBA	GL_UNSIGNED_BYTE	RGBA8	4
GL_RGB	GL_UNSIGNED_SHORT_5_6_5	RGB565	2
GL_RGB	GL_UNSIGNED_BYTE	RGB8	3
GL_ALPHA	GL_UNSIGNED_BYTE	A8	1
GL_ALPHA	GL_UNSIGNED_4BITS_DMP	A4	0.5
GL_LUMINANCE	GL_UNSIGNED_BYTE	L8	1
GL_LUMINANCE	GL_UNSIGNED_4BITS_DMP	L4	0.5
GL_LUMINANCE_ALPHA	GL_UNSIGNED_BYTE	LA8	2

GL_LUMINANCE_ALPHA	GL_UNSIGNED_BYTE_4_4_DMP	LA4	1
GL_SHADOW_DMP	GL_UNSIGNED_INT	–	4
GL_GAS_DMP	GL_UNSIGNED_SHORT	–	4
GL_HILO8_DMP	GL_UNSIGNED_BYTE	–	2
GL_RGBA_NATIVE_DMP	GL_UNSIGNED_SHORT_4_4_4_4	RGBA4	2
GL_RGBA_NATIVE_DMP	GL_UNSIGNED_SHORT_5_5_5_1	RGBA5551	2
GL_RGBA_NATIVE_DMP	GL_UNSIGNED_BYTE	RGBA8	4*
GL_RGB_NATIVE_DMP	GL_UNSIGNED_SHORT_5_6_5	RGB565	2
GL_RGB_NATIVE_DMP	GL_UNSIGNED_BYTE	RGB8	3*
GL_ALPHA_NATIVE_DMP	GL_UNSIGNED_BYTE	A8	1
GL_ALPHA_NATIVE_DMP	GL_UNSIGNED_4BITS_DMP	A4	0.5
GL_LUMINANCE_NATIVE_DMP	GL_UNSIGNED_BYTE	L8	1
GL_LUMINANCE_NATIVE_DMP	GL_UNSIGNED_4BITS_DMP	L4	0.5
GL_LUMINANCE_ALPHA_NATIVE_DMP	GL_UNSIGNED_BYTE	LA8	2*
GL_LUMINANCE_ALPHA_NATIVE_DMP	GL_UNSIGNED_BYTE_4_4_DMP	LA4	1
GL_SHADOW_NATIVE_DMP	GL_UNSIGNED_INT	–	4*
GL_GAS_NATIVE_DMP	GL_UNSIGNED_SHORT	–	4*
GL_HILO8_DMP_NATIVE_DMP	GL_UNSIGNED_BYTE	–	2*

R、G、B:カラー(Red、Green、Blue)
A:アルファ
L:輝度

OpenGL ES1.1 での仕様と同様に、RGB のように A 成分の存在しないテクスチャをテクスチャコンバイナから参照した場合、A 成分には 1 が出力されます。これは、圧縮テクスチャについても同様です。

format が GL_RGB または GL_RGB_NATIVE_DMP で、*type* が GL_UNSIGNED_BYTE の組み合わせは *target* に GL_TEXTURE_2D を指定している場合にのみ指定可能です。*format* に GL_*_NATIVE_DMP を指定した場合、*pixels* で指定されるデータは PICA のネイティブフォーマットでなければなりません。また、GL_GAS_DMP を指定した場合は、*pixels* は 0(NULL)でなければなりません。

width と *height* にはテクスチャイメージの幅と高さを指定します。ともに 8 ～ 1024 の 2 のべき乗の数でなければなりません。

target の指定が GL_TEXTURE_CUBE_MAP_* である場合、*width* と *height* の値は同じでなければなりません。また、各面の設定は *pixels*(と *target*)以外がすべて同じ指定でなければなりません。

pixels に 0(NULL)を指定した場合はイメージデータのロードは行われず、領域の確保だけが行われます。

internalformat には内部基本形式を指定します。*internalformat* と *format* の指定が同じでない場合は GL_INVALID_OPERATION のエラーが生成されます。イメージの RGBA 成分と内部基本形式の各成分との対応は以下のようになっています。

表 7-4. 内部基本形式による RGBA と内部形式の対応

内部基本形式	RGBA	内部形式
GL_ALPHA	A	A
GL_LUMINANCE	R	L
GL_LUMINANCE_ALPHA	R、A	L、A
GL_RGB	R、G、B	R、G、B
GL_RGBA	R、G、B、A	R、G、B、A
GL_HILO8_DMP	R、G	Nx、Ny

GL_HILO8_DMP の場合、B 成分と A 成分にはそれぞれ 0.0 と 1.0 が出力されます。

OpenGL の仕様と異なり、*level* にはミップマップ段数を負の値(例えば、段数が 2 ならば -2。0 と -1 は同じ段数 1 として扱う)で設定します。個別にミップマップの各レベルで使用するテクスチャを指定することはできず、*pixels* には最も大きなサイズのレベルで使用するテクスチャから最も小さなサイズのレベルで使用するテクスチャまでを連続して格納したデータを指定してください。

border には、0 以外の値を指定することはできません。

> **補足:** glTexImage2D() の *target* に特別なフラグを論理和で渡すことで、GPU のアクセス先や領域確保時の処理を指定することができます。詳細については、「3DS プログラミングマニュアル – グラフィックス応用編」「メインメモリに置かれたデータの使用」を参照してください。

7.3.1. コンポーネントが 4 ビットのフォーマットについて

type が GL_UNSIGNED_BYTE_4_4_DMP または GL_UNSIGNED_4BITS_DMP のテクスチャフォーマットは、1 つのコンポーネントが 4 ビットで構成されているため、そのデータの並びは特殊なものになっています。

type が GL_UNSIGNED_BYTE_4_4_DMP のフォーマットは、1 バイトに 2 つのコンポーネントを含むフォーマットです。*format* には GL_LUMINANCE_ALPHA または GL_LUMINANCE_ALPHA_NATIVE_DMP を組み合わせることができます。上位 4 ビットに輝度成分、下位 4 ビットにアルファ成分が格納されます。

type が GL_UNSIGNED_4BITS_DMP のフォーマットは、1 バイトに 2 テクセルを含むフォーマットです。*format* には GL_LUMINANCE、GL_LUMINANCE_NATIVE_DMP、GL_ALPHA、GL_ALPHA_NATIVE_DMP のいずれかを組み合わせることができます。テクセルの並びで見ると、成分は下位 4 ビット、上位 4 ビットの順に格納されます。

> **注意:** *type* に GL_UNSIGNED_4BITS_DMP を指定するテクスチャフォーマット(4 ビットフォーマット)と、それ以外のテクスチャフォーマット(非 4 ビットフォーマット。ETC1 方式の圧縮テクスチャ含む)を同時に有効にし、マルチテクスチャとして使用する場合はテクスチャの配置に制限があります。
>
> 4 ビットフォーマットのテクスチャを VRAM に配置する場合は 4 ビットフォーマットと非 4 ビットフォーマットのテクスチャを異なるメモリに配置しなければなりません。その際、VRAM-A と VRAM-B は異なるメモリとして扱われます。なお、同じメモリに配置した場合の動作は不定です。
>
> 4 ビットフォーマットのテクスチャをメインメモリに配置した場合はテクスチャの配置に制限はありません。

図 7-1. コンポーネントが 4 ビットで構成されているテクスチャフォーマットのビットレイアウト

GL_UNSIGNED_BYTE_4_4_DMP

GL_LUMINANCE_ALPHA
(GL_LUMINANCE_ALPHA_NATIVE_DMP)

+0 Byte		+1 Byte	
Luminance [n]	Alpha [n]	Luminance [n+1]	Alpha [n+1]

GL_UNSIGNED_4BITS_DMP

GL_LUMINANCE
(GL_LUMINANCE_NATIVE_DMP)

+0 Byte		+1 Byte	
Luminance [n+1]	Luminance [n]	Luminance [n+3]	Luminance [n+2]

GL_ALPHA
(GL_ALPHA_NATIVE_DMP)

+0 Byte		+1 Byte	
Alpha [n+1]	Alpha [n]	Alpha [n+3]	Alpha [n+2]

7.4. 圧縮テクスチャのロード

圧縮された画像データをテクスチャイメージとしてロードすることができます。**glCompressedTexSubImage2D() による圧縮テクスチャの部分ロードはサポートしていません**。

コード 7-4. glCompressedTexImage2D() の定義

```
void glCompressedTexImage2D(GLenum target, GLint level, GLenum internalformat,
                            GLsizei width, GLsizei height, GLint border,
                            GLsizei imageSize, const void* data);
```

*target*に指定する値は glTexImage2D() の *target* と同じです(表 7-2)。ただし、この関数で確保した領域をシャドウテクスチャやガステクスチャに使用することはできません。

width と *height* にはテクスチャイメージの幅と高さを指定します。ともに 16 ～ 1024 の 2 のべき乗の数でなければなりません。

level と *border* に指定する値と、キューブマップテクスチャおよびミップマップテクスチャについての制限は glTexImage2D() と同じです。詳細は「7.3. テクスチャイメージのロード」を参照してください。

テクスチャの圧縮方式は ETC1(Ericsson Texture Compression)方式のみがハードウェアでサポートされています。そのため、*internalformat* には GL_ETC1_RGB8_NATIVE_DMP または GL_ETC1_ALPHA_RGB8_A4_NATIVE_DMP が指定可能です。

ETC1 方式は 24 bit RGB フォーマットの 4x4 テクセルを 64 bit に圧縮します。GL_ETC1_RGB8_NATIVE_DMP はアルファチャンネルには対応していませんが、GL_ETC1_ALPHA_RGB8_A4_NATIVE_DMP は 16 テクセル× 4 bit のアルファ成分データを付加することによってアルファチャンネルに対応しています。

なお、アルファチャンネルに対応していないフォーマットの圧縮テクスチャをテクスチャコンバイナから参照した場合、A 成分には 1 が出力されます。

imageSize にはイメージデータのバイトサイズを指定します。*imageSize* はテクスチャの元画像の幅を w、高さを h とすれば以下の計算式で求めることができます。blockSize にはアルファチャンネルなしでは 8 が、アルファチャンネルありでは 16 が適用されます。

imageSize = (w / 4) * (h / 4) * blockSize

3DS で扱う ETC1 方式は標準の仕様とは異なっています。標準仕様とのフォーマットの違いは「7.10. PICA ネイティブフォーマット」で説明します。フォーマットの詳細については「3DS プログラミングマニュアル – グラフィックス応用編」を参照してください。

> **補足:** glCompressedTexImage2D() の *target* に特別なフラグを論理和で渡すことで、GPU のアクセス先や領域確保時の処理を指定することができます。詳細については、「3DS プログラミングマニュアル – グラフィックス応用編」「メインメモリに置かれたデータの使用」を参照してください。

7.5. フレームバッファからのコピー

フレームバッファオブジェクトに関連付けられているカラーバッファ、デプスバッファの画像をテクスチャとして取得(コピー)することができます。

7.5.1. カラーバッファからのコピー

カラーバッファの画像をテクスチャとして取得(コピー)することができます。

コード 7-5. glCopyTexImage2D() の定義

```
void glCopyTexImage2D(GLenum target, GLint level, GLenum internalformat,
                      GLint x, GLint y, GLsizei width, GLsizei height,
                      GLint border);
```

target に指定する値は glTexImage2D() の *target* と同じです(表 7-2)。そのほかの引数も以下の違いを除けば同じです。

- *internalformat* で指定可能な内部形式のフォーマットは GL_RGB または GL_RGBA のみで、カラーバッファのフォーマットからの変換を伴うコピーはできません(ピクセルサイズが同じフォーマットは可能です)。
- *x*、*y* にはカラーバッファからコピーする領域の開始点(左下原点、右上正方向)を指定します。*width*、*height* にはコピーする領域の幅と高さを指定します。*x*、*y* は 8 の倍数で指定しなければなりません。
- *level* には 0 のみが指定可能です。

> **補足:** glCopyTexImage2D() の *target* に特別なフラグを論理和で渡すことで、GPU のアクセス先や領域確保時の処理を指定することができます。詳細については、「3DS プログラミングマニュアル – グラフィックス応用編」「メインメモリに置かれたデータの使用」を参照してください。

7.5.2. カラーバッファからの部分コピー

カラーバッファからテクスチャイメージの部分領域にコピーすることもできます。

コード 7-6. glCopyTexSubImage2D() の定義

```
void glCopyTexSubImage2D(GLenum target, GLint level,
                         GLint xoffset, GLint yoffset, GLint x, GLint y,
                         GLsizei width, GLsizei height);
```

コピー先のテクスチャイメージの領域を、glTexImage2D() で事前に確保していなければなりません。

xoffset と *yoffset* にコピー先領域の座標(左下原点、右上正方向。8 の倍数でなければなりません)を指定すること、*width* と *height* には 8 の倍数(2 のべき乗でなくてもよい)を指定すること以外は、glCopyTexImage2D() と同じです。詳細は「7.5.1. カラーバッファからのコピー」を参照してください。

7.5.3. デプスバッファからのコピー

glEnable() の引数に GL_DEPTH_STENCIL_COPY_DMP を渡して呼び出し、デプスステンシルコピー機能を有効にした状態で glCopyTexImage2D() または glCopyTexSubImage2D() を呼び出したときはカラーバッファではなく、デプスバッファ(ステンシル含む)の内容がテクスチャにコピーされます。

カレントのデプスバッファのフォーマットによって、コピー先に指定するテクスチャのフォーマットは決められています。コピー時にフォーマットの変換は行われませんので、対応していないフォーマットのテクスチャにコピーをしようとした場合は GL_INVALID_OPERATION のエラーが生成されます。テクスチャにコピーされる内容は、テクスチャがネイティブフォーマットであるかどうかによって変化することはありません。

表 7-5. デプスバッファのフォーマットとテクスチャの format、type の対応

デプスバッファのフォーマット	テクスチャの format	テクスチャの type	成分の内容
GL_DEPTH24_STENCIL8_EXT	GL_RGBA GL_RGBA_NATIVE_DMP	GL_UNSIGNED_BYTE	R: Stencil G: D [23 : 16] B: D [15 : 8] A: D [7 : 0]
GL_DEPTH_COMPONENT24_OES	GL_RGB GL_RGB_NATIVE_DMP	GL_UNSIGNED_BYTE	R: D [23 : 16] G: D [15 : 8] B: D [7 : 0]
GL_DEPTH_COMPONENT16	GL_HILO8_DMP GL_HILO8_DMP_NATIVE_DMP	GL_UNSIGNED_BYTE	R: D [15 : 8] G: D [7 : 0]

※「成分の内容」列の "D [x : y]" は、その成分に設定されるデプス値のビット位置を示しています。

デプスステンシルコピー機能は、GL_DEPTH_STENCIL_COPY_DMP を引数にして、glEnable()、glDisable()、glIsEnabled() を呼び出すことで、有効化、無効化、有効・無効の判定を行うことができます。

> **補足:** glCopyTexImage2D() と glCopyTexSubImage2D() はブロックモードがブロック 32 モードのときに使用することができません。エラーは生成されませんが、画像の転送が正しく行われません。ブロックモードの設定については「3DS プログラミングマニュアル – グラフィックス応用編」を参照してください。

7.6. レンダーターゲットへのテクスチャの指定

glTexImage2D() で確保したテクスチャイメージを glFramebufferTexture2D() でフレームバッファオブジェクトに関連付けることで、レンダリング結果を直接テクスチャに書き込む(render to texture)ことができます。シャドウやガスで使用される特殊なテクスチャはそれぞれの機能においてレンダリング結果が直接テクスチャに書き込まれるため、この関数でテクスチャをレンダーターゲットに指定しなければなりません。

コード 7-7. glFramebufferTexture2D() の定義

```
void glFramebufferTexture2D(GLenum target, GLenum attachment, GLenum textarget,
                            GLuint texture, GLint level);
```

target に指定することができるのは GL_FRAMEBUFFER のみです。

attachement にはテクスチャに書き込む情報を指定します。カラーバッファならば GL_COLOR_ATTACHMENT0、デプスバッファならば GL_DEPTH_ATTACHMENT を指定します。

textarget に指定する値は glTexImage2D() の *target* と同じです(表 7-2)。

texture には、フレームバッファオブジェクトに関連付けるテクスチャオブジェクトを指定します。

level には 0 のみが指定可能です。

デプスバッファ(ステンシルバッファ含む)のレンダーターゲットにテクスチャを指定する場合、デプスバッファのフォーマットによって、テクスチャのフォーマットは決められています。対応するフォーマットは「7.5.3. デプスバッファからのコピー」と同じです(表 7-5)。デプスバッファのフォーマットが GL_DEPTH24_STENCIL8_EXT の場合、*attachment* に GL_DEPTH_STENCIL_ATTACHMENT を指定できますが、GL_DEPTH_ATTACHMENT と同じ結果になります。

7.7. 参照テーブルのロード

OpenGL の 1 次元テクスチャをロードする関数 glTexImage1D() を、3DS では参照テーブルのロードに使用します。参照テーブルとは、プロシージャルテクスチャやフラグメントライティング、フォグ、ガスで参照する 1 次元のテーブルのことで、テクスチャとして使用することはありません。

コード 7-8. glTexImage1D() の定義

```
void glTexImage1D(GLenum target, GLint level, GLint internalformat,
                  GLsizei width, GLint border, GLenum format, GLenum type,
                  const GLvoid *pixels);
```

target にはロード先の参照テーブルを GL_LUT_TEXTURE*i*_DMP で指定します。*i* は参照テーブルの番号(0 ～ glGetIntegerv() の *pname* に GL_MAX_LUT_TEXTURES_DMP を渡して得られる値 – 1)です。つまり、参照テーブル番号は 0 ～ 31 の範囲で指定することができます。GL_LUT_TEXTURE*i*_DMP は(GL_LUT_TEXTURE0_DMP + *i*)で定義されています。

level には 0 、*type* には GL_FLOAT、*format* および *internalformat* には GL_LUMINANCEF_DMP のみを指定することができます。*level* に 0 以外を指定した場合は GL_INVALID_VALUE のエラーを、*type*、*format*、*internalformat* に上記以外を指定した場合は GL_INVALID_ENUM のエラーを生成します。

width にはテーブルの要素数を、*pixels* にはテーブルの要素を指定します。*width* に指定することのできる最大値は glGetIntegerv() の *pname* に GL_MAX_LUT_ENTRIES_DMP を渡して得られる 512 ですが、テーブルの要素数および要素は予約フラグメント処理によって独自の制限が決められています。

glGetIntegerv() の *pname* に GL_TEXTURE_BINDING_LUT*i*_DMP(*i* は 0 ～ 31)を渡して呼び出すと、GL_LUT_TEXTURE*i*_DMP に関連付けられているテクスチャオブジェクト(の ID)を取得することができます。

すでにロードされている参照テーブルの内容を一部分だけ書き換えるには、glTexSubImage1D() を使用します。

コード 7-9. glTexSubImage1D() の定義

```
void glTexSubImage1D(GLenum target, GLint level, GLint xoffset,
                     GLsizei width, GLenum format, GLenum type,
                     const GLvoid *pixels);
```

コピー先の参照テーブルの領域は、glTexImage1D() で事前に確保していなければなりません。

xoffset と *width* に書き換えたい要素の先頭番号と要素数を指定すること以外は、glTexImage1D() と同じです。*xoffset* と *width* の合計がテーブルの要素数を超える場合は GL_INVALID_VALUE のエラーを生成します。

glCopyTexImage1D()、glCopyTexSubImage1D() は実装されていません。また、テクスチャパラメータの設定には対応していません。

7.8. テクスチャオブジェクトの破棄

不要になったテクスチャオブジェクトは glDeleteTextures() で破棄することができます。

コード 7-10. glDeleteTextures() の定義

```
void glDeleteTextures(GLsizei n, const GLuint* textures);
```

7.9. テクスチャコレクション

テクスチャコレクションは 3DS が独自に導入したもので、テクスチャオブジェクトとテクスチャの関連付けを記録します。テクスチャコレクションを使用することで、記録された各種テクスチャとテクスチャオブジェクトの関連付けをまとめて行うことができます。テクスチャコレクションはテクスチャオブジェクトと名前空間を共有していますので、glGenTextures()、glBindTexture()、glDeleteTextures() を使用して生成・指定・破棄を行います。

7.9.1. テクスチャコレクションの生成

テクスチャコレクションは特殊なテクスチャオブジェクトとして動作します。そのためテクスチャオブジェクトと同じように、glGenTextures() でテクスチャコレクションとして使用するオブジェクトを生成します。

7.9.2. テクスチャコレクションの指定

テクスチャコレクションとして使用するテクスチャオブジェクトを指定するには glBindTexture() を使用します。*target* に GL_TEXTURE_COLLECTION_DMP を渡して呼び出してください。デフォルトでは、名前 0(*texture* に 0 を渡した状態)のオブジェクトがテクスチャコレクションとなっています。

呼び出し後は、各種テクスチャ(2 次元テクスチャ、キューブマップテクスチャ、参照テーブル)への glBindTexture() によるテクスチャオブジェクトの関連付けはテクスチャコレクションに記録されていきます。同じテクスチャへの関連付けは上書きされ、テクスチャコレクションへの記録はほかのテクスチャコレクションへ切り替えられるまで行われます。

テクスチャコレクションを切り替えると、各種テクスチャとテクスチャオブジェクトとの関連付けはテクスチャコレクションに記録されているオブジェクトにすべて切り替わります。

7.9.3. テクスチャコレクションの破棄

テクスチャオブジェクトと同じように、glDeleteTextures() でテクスチャコレクションを破棄することができます。テクスチャコレクションを破棄しても、記録されているテクスチャオブジェクトのテクスチャとの関連付けに影響ありません。

使用中のテクスチャコレクションへの glDeleteTextures() の呼び出しでは、すぐにテクスチャコレクションが破棄されません。ほかのテクスチャコレクションに切り替えられるまで使用状態のまま残ります。デフォルトのテクスチャコレクションは破棄できません。デフォルトのテクスチャコレクションに対する glDeleteTextures() の呼び出しは無視されます。

7.10. PICA ネイティブフォーマット

GPU のテクスチャユニットがサポートしているテクスチャフォーマットは OpenGL の仕様と異なっています。テクスチャユニットが実際にサポートしているフォーマットを PICA ネイティブフォーマットと呼びます。PICA ネイティブフォーマットのテクスチャをロードする場合、ライブラリ内でのフォーマット変換が行われないため、標準フォーマットよりも効率が良くなります。

OpenGL の仕様との大きな違いは以下の 3 つです。

- バイトオーダー

内部のアドレス処理の関係でバイトデータの記述順序が異なります。

- V フリップ
  U、V 座標とテクセルとの配置の関係が V 方向で反対になっています。
- アドレッシング
  リニアアドレッシング（OpenGL）とブロックアドレッシング（PICA ネイティブフォーマット）の相違により、テクセルおよび圧縮データブロックの記述順序が異なります。

OpenGL のフォーマットから PICA ネイティブフォーマットへ変換するには、非圧縮テクスチャ（`glTexImage2D()` でロード）であれば V フリップ変換→アドレッシング変換→バイトオーダー変換の順に行い、圧縮テクスチャ（`glCompressedTexImage2D()` でロード）であれば、V フリップ変換→ETC 圧縮→アドレッシング変換→バイトオーダー変換の順に行います。ETC 圧縮の前に V フリップ変換を行わなければならないことに注意してください。

7.10.1. バイトオーダーの違い

`glTexImage2D()` の *type* に `GL_UNSIGNED_BYTE` を指定し、複数バイトで 1 テクセルを表現するフォーマットは、そのバイト数でバイトスワップ（バイトデータの入れ替え）を行います。

圧縮フォーマットの場合は 1 ブロック（4x4 テクセル分の 8 バイト）単位でバイトデータの入れ替えを行います。ただし、アルファチャンネルありの場合は、アルファ部分（前半の 8 バイト）のバイトデータの入れ替えは行いません。

図 7-2. 4 バイトスワップ

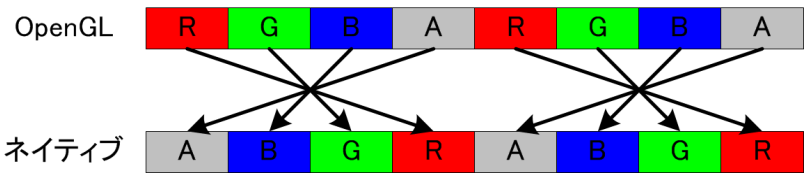


図 7-3. 3 バイトスワップ

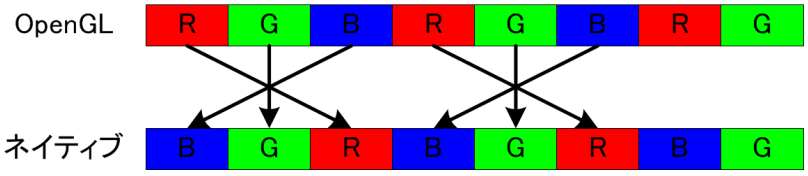


図 7-4. 2 バイトスワップ

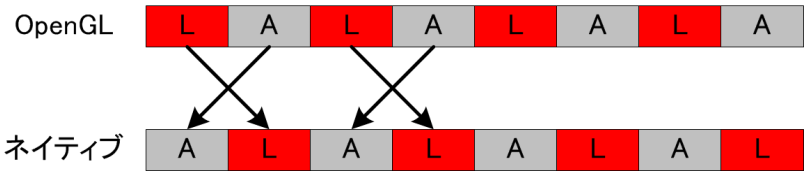


図 7-5. 圧縮フォーマット(アルファチャンネルなし)のバイトスワップ

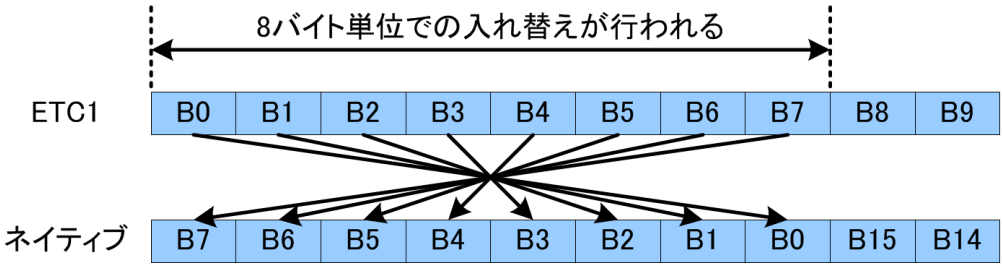
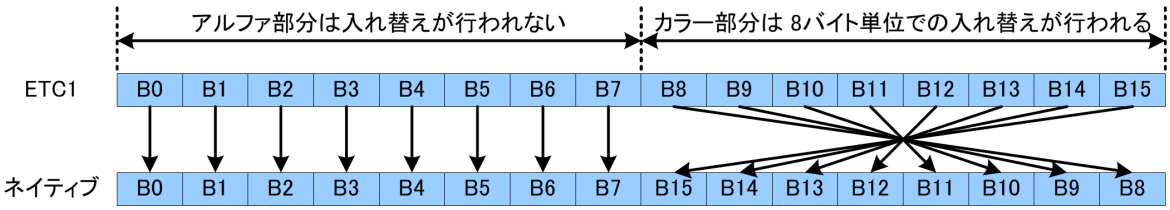


図 7-6. 圧縮フォーマット(アルファチャンネルあり)のバイトスワップ



7.10.2. V フリップの違い

OpenGL の仕様では(u, v)=(0.0, 0.0)にあるテクセルを先頭にイメージがデータ化されますが、PICA ネイティブフォーマットでは(u, v)=(0.0, 1.0)にあるテクセルを先頭にイメージがデータ化されます。非圧縮・圧縮テクスチャでの違いはありません。

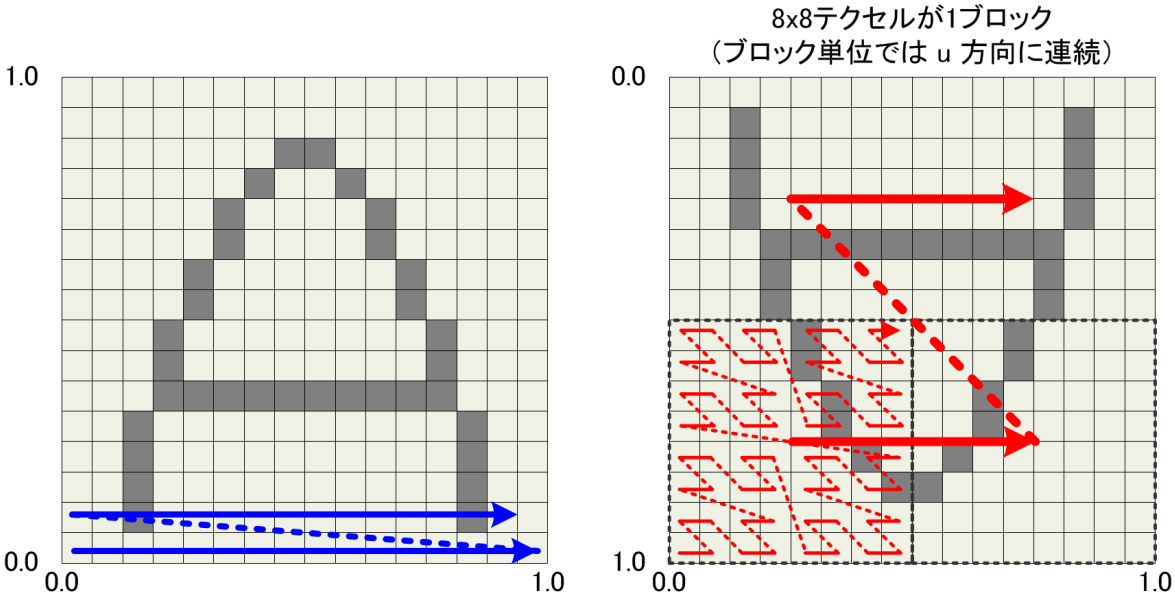
7.10.3. アドレッシングの違い

OpenGL の仕様では u 方向に連続したテクセルを格納(リニアアドレッシング)していきますが、PICA ネイティブフォーマットではブロック単位で u 方向に連続しますが、ブロック内はジグザグにテクセルを格納(ブロックアドレッシング)していきます。

7.10.3.1. 非圧縮テクスチャ

8x8 テクセルを 1 ブロックとして、ブロック内はジグザグ(図内の赤線)に格納していきますが、ブロック単位では u 方向に連続して格納していきます。

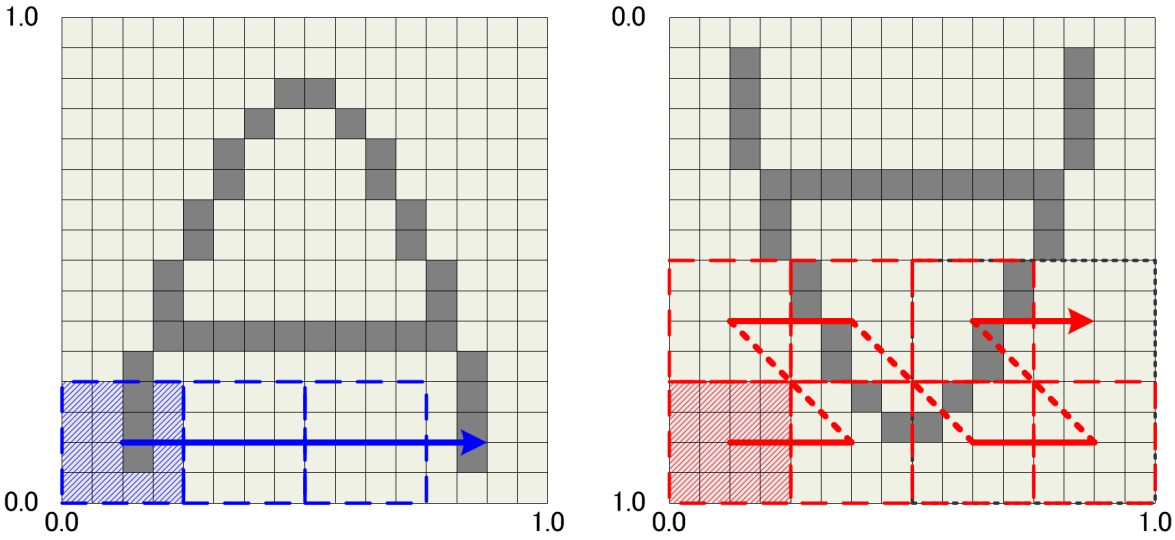
図 7-7. 非圧縮テクスチャのテクセル格納順



7.10.3.2. 圧縮テクスチャ

圧縮テクスチャは 4x4 テクセルをまとめてブロックと呼び、ブロック単位で圧縮されます。OpenGL の仕様ではブロック単位で u 方向に連続して格納（図内の青線）していきますが、PICA ネイティブフォーマットでは 2x2 ブロックをメタブロックとし、メタブロック内はジグザグに、メタブロック単位では u 方向に連続して格納（図内の赤線）していきます。

図 7-8. 圧縮テクスチャのテクセル格納順



圧縮テクスチャについての詳細は「3DS プログラミングマニュアル – グラフィックス応用編」を参照してください。

8. 頂点シェーダ

頂点シェーダでは入力された頂点属性データに座標変換や陰影処理などを行います。3DS は 4 つの 24 ビット浮動小数点数で構成されたベクトル型データを扱うことのできる頂点プロセッサを 4 基搭載しています。入力された頂点データは、4 つある頂点プロセッサによって並列に処理されます。プロセッサでは汎用的な計算をすることができますが、VRAM からのデータの読み込みと書き込みはできません。

OpenGL ES 2.0 と同様に、3DS で扱われる頂点データには"座標"や"法線"といった明確な区別はありません。頂点処理シェーダが入力されたデータをどのように処理し、出力するかで属性が決定します。一般的に、3D グラフィックス処理で入力される頂点データには以下のような属性があります。

- 頂点座標
- 法線ベクトル
- 接線ベクトル
- テクスチャ座標
- 頂点カラー

頂点シェーダは PICA グラフィックスコア独自のアセンブリ言語で記述します。この章の内容を読み進めるにあたっては、「頂点シェーダ リファレンスマニュアル」との併読をお勧めします。

8.1. 頂点データの入力

アプリケーションから入力される頂点データは、頂点属性番号に関連付けられた入力レジスタを介して頂点シェーダに渡されます。頂点シェーダ側では `#pragma bind_symbol` により、データ名とデータを入力するレジスタを指定します。

コード 8-1. データ名とレジスタの関連付け(シェーダアセンブラ)

```
#pragma bind_symbol(AttribPosition.xyzw, v0, v0)
```

上記のコード例では、入力レジスタ `v0` のコンポーネント `xyzw` にデータ名 "`AttribPosition`" を関連付けています。同じ入力レジスタに複数のデータ名を関連付けることはできません。また、第 2、第 3 引数は同じ値(もしくは第 3 引数を省略して指定)でなければなりません。

アプリケーション側では `glBindAttribLocation()` で頂点属性番号とデータ名を関連付け、`glEnableVertexAttribArray()` で頂点属性番号の関連付けを有効にし、`glVertexAttribPointer()` などで頂点データを入力します。

コード 8-2. 頂点属性番号の関連付けと頂点データ入力

```
glBindAttribLocation(program, 0, "AttribPosition");
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, pointer);
```

上記のコード例では、頂点属性番号 0 を "`AttribPosition`" の名前を持つデータに関連付け、4 成分の頂点属性を入力しています。**レジスタの番号と頂点属性番号は同じである必要はありません**。

`#pragma bind_symbol` で関連付けられていない入力レジスタの値は不定です。入力される頂点データが `GL_FLOAT` 型でない場合、自動的に `GL_FLOAT` 型への変換が行われます。ただし、データの"normalize"(正規化)は行われませんので注意が必要です。

頂点バッファを使用する場合、頂点属性のデータタイプとデータサイズの組み合わせが頂点データの転送速度に影響しま

す。詳細については、「15.13. 頂点属性の組み合わせによる頂点データの転送速度への影響」を参照してください。

> **注意:** glVertexAttribPointer() で頂点データを入力する場合、第 4 引数の指定(正規化の指定)はハードウェアでサポートされていないため無視されます。指定が反映されないため、頂点シェーダ内で明示的に正規化処理を行う必要があります。
>
> glVertexAttribPointer() の *type* に GL_FIXED および GL_UNSIGNED_SHORT を指定することはできません。また、GL_FLOAT、GL_SHORT を指定した場合、*ptr* に指定するポインタはアライメントがそれぞれ 4 Byte、2 Byte でなければなりません。

頂点シェーダでは 1 つの頂点を処理する間に、少なくとも 1 つの入力レジスタ(1 コンポーネントでもかまいません)から頂点データを読み込まなければ正常に動作しない可能性があります。

8.2. 頂点データの出力

頂点シェーダの処理結果は、出力頂点属性にマッピングされた出力レジスタへの書き込みで後段のプロセスへ渡されます。頂点シェーダ側では #pragma output_map により、出力頂点属性名とデータを出力するレジスタを指定します。

コード 8-3. 出力頂点属性のレジスタマップとレジスタへの書き込み(シェーダアセンブラ)

```
#pragma output_map(position, o0)
mov     o0,     v0
```

上記のコード例では、出力レジスタ o0 を頂点座標にマップしています。o0 にデータを書き込むことで、データが頂点座標として出力されます。

後段のフラグメント処理には予約されたシェーダしか使用できないため、頂点シェーダが出力する頂点属性はあらかじめ決められたものになります。出力頂点属性には以下の属性名があり、出力される属性には頂点シェーダで設定すべき成分が決められています。

表 8-1. 出力頂点属性

属性名	出力される属性	設定すべき成分
position	頂点座標	(x y z w) の 4 成分
color	頂点カラー	(R G B A) の 4 成分
texture0	テクスチャ座標 0	(u v) の 2 成分
texture0w	テクスチャ座標 0	(w) の 1 成分
texture1	テクスチャ座標 1	(u v) の 2 成分
texture2	テクスチャ座標 2	(u v) の 2 成分
quaternion	クォータニオン	(x y z w) の 4 成分
view	ビューベクタ	(x y z) の 3 成分
generic	汎用属性	ジオメトリシェーダで使用する任意の数の成分

頂点シェーダでは、マッピングされたレジスタすべてのコンポーネント(xyzw)に対して何らかの値を書き込まなければなりません。そのため、#pragma output_map で指定されたレジスタのコンポーネントすべてがマッピングされていない場合、使用されていないコンポーネントにダミー値を書き込むなどの処理が必要になります。

また、マッピングされたレジスタすべてに値が書き込まれた時点で頂点シェーダは処理を強制的に終了し、次の頂点処理に移ります(end 命令の呼び出しは必要です)。つまり、最後の属性データがレジスタに書き込まれた後の命令は実行されない可能性があります。

1 つの出力レジスタ(コンポーネントごと)への書き込みは 1 頂点の処理の間に 1 度だけです。同じ出力レジスタの同じコンポーネントへ複数回書き込みを行った場合の動作は保証されていません。

generic 以外の出力頂点属性をマッピングできる出力レジスタは 7 つまでです。generic 以外で 8 つ以上の出力頂点属性をマッピングする場合は、複数の頂点属性を 1 つのレジスタにマッピングする必要があります。その場合、texture1 と texture2 を o0.xy と o0.zw にマッピングするといったように、複数の頂点属性(合計が 4 成分までのもの)を 1 つのレジスタにパックしなければなりません。

8.3. ユニフォーム設定

#pragma bind_symbol で入力レジスタ以外(浮動小数点数レジスタ、ブールレジスタ、整数レジスタ)にデータ名を関連付けている場合、glUniform*() によってアプリケーションから各レジスタに値を設定することができます。glUniformMatrix*() は *transpose* に GL_TRUE を指定することで、行列の転置(column-major order から row-major order への変換)を関数内で行うことができます。

以下に、頂点シェーダ側とアプリケーション側のコード例を示します。

コード 8-4. 入力レジスタ以外へのデータ名関連付けの例(シェーダアセンブラ)

```
#pragma bind_symbol ( ModelViewMatrix , c0 , c3 )
#pragma bind_symbol ( LoopCounter0 , i1 , i1 )
#pragma bind_symbol ( bFirst , b2 , b2 )
#pragma bind_symbol ( Scalar.x , c4, c4 )
```

コード 8-5. ユニフォーム設定の例

```
uniform_location = glGetUniformLocation ( program , "ModelViewMatrix" );
GLfloat matrix[4][4];
glUniform4fv ( uniform_location , 4 , matrix );

GLfloat scalar_value;
uniform_location = glGetUniformLocation ( program , "Scalar" );
glUniform1f ( uniform_location , scalar_value );

uniform_location = glGetUniformLocation ( program , "bFirst" );
glUniform1i ( uniform_location , GL_TRUE );

GLint loop_setting[3] = { 4 , 0 , 1 } ; // loop_count-1 , init , step
uniform_location = glGetUniformLocation ( program , "LoopCounter0" );
glUniform3iv ( uniform_location , loop_setting );
```

上記のコード例では、"Scalar" のデータ名にコンポーネント x を指定しています。このように、関連付けるレジスタが浮動小数点数レジスタである場合はコンポーネントを指定することができます。コンポーネントの指定は、xyzw の順序で行い、連続する成分でなければなりません。つまり、xy や zw、yzw は指定可能ですが、xz や yw、xyw は指定することができません。

整数レジスタはシェーダプログラムの loop 命令の制御に使用されます。24 ビット幅のレジスタで、0 ～ 7 bit がループ回数、8 ～ 15 bit が初期値、16 ～ 23 bit が増加値に割り当てられています。loop 命令はループカウンタレジスタを初期値

で初期化し、loop 命令から endloop 命令までを(指定されたループ回数 + 1)回繰り返し実行します。ループを繰り返すごとにループカウンタレジスタが増加値分だけ増加します。

8.4. クリップ座標系の注意事項

頂点シェーダが出力するクリップ座標系の Z 成分は OpenGL ES 標準の仕様と異なっています。

OpenGL ES では -Wc から Wc の範囲でクリップされますが、3DS では 0 から -Wc の範囲(符号が反転しています)でクリップされます。このため、アプリケーションが OpenGL ES 互換の射影変換行列を使用する場合、-Wc から Wc の範囲を 0 から -Wc の範囲に変換する処理が必要になります。

アプリケーションで射影変換行列を変換する場合

以下のような変換を施してからユニフォーム設定されたレジスタに設定します。

コード 8-6. OpenGL ES 互換の射影変換行列を変換する

```
GLfloat projection[16];
projection[2] = (projection[2] + projection[3]) * (-0.5f);
projection[6] = (projection[6] + projection[7]) * (-0.5f);
projection[10] = (projection[10] + projection[11]) * (-0.5f);
projection[14] = (projection[14] + projection[15]) * (-0.5f);
```

頂点シェーダ側で行う場合

射影変換処理を以下のようにして行います。

コード 8-7. OpenGL ES 互換の射影行列で射影変換する(シェーダアセンブラ)

```
#pragma output_map(position, o0)
#pragma bind_symbol(attrib_position, v0)
#pragma bind_symbol(modelview, c0, c3)
#pragma bind_symbol(projection, c4, c7)
def     c8, -0.5, -0.5, -0.5, -0.5

// Model View Transformation
dp4     r0.x, v0, c0
dp4     r0.y, v0, c1
dp4     r0.z, v0, c2
dp4     r0.w, v0, c3
// Projective Transformation
dp4     o0.x, r0, c4
dp4     o0.y, r0, c5
mov     r1, c6
add     r1, r1, c7
mul     r1, r1, c8
dp4     o0.z, r0, r1
dp4     o0.w, r0, c7
```

8.5. 頂点キャッシュ

頂点シェーダで生成または加工された頂点データの一部はキャッシュとして保存されます。頂点インデックスをもとに、頂点シェーダに入力される頂点データがキャッシュに存在する頂点データの元となったデータと同じであると判断された場合、頂

点シェーダで処理をせずにキャッシュ上に存在する処理済みの頂点データを次プロセスへと送ります。一般的に GL_TRIANGLES で頂点データを入力すると、同じ頂点データに対して複数回の処理が行われることが多いのですが、処理済みの頂点データがキャッシュ上に存在する場合は処理を省略することができます。

頂点キャッシュを使用するためには、以下の条件を満たしていなければなりません。

- 頂点インデックスを参照した頂点データの入力形式であること。つまり、glDrawElements() を呼び出して頂点データを入力していること。
- 頂点バッファを使った頂点データの入力であること。

頂点キャッシュには 32 エントリの頂点データをキャッシュすることができ、LRU の挙動に似た、独自のアルゴリズムで実装されています。

繰り返し参照される頂点データが 32 頂点以内に収まっているときは、キャッシュにヒットする可能性が高くなります。頂点キャッシュの効率は、インデックスアレイが格納されているメモリの使用状況や、頂点シェーダで実行されているシェーダの長さなど、インデックスの並び以外の条件も影響します。そのため、最適なインデックスはコンテンツに依存し、一意には決まらない可能性があります。

8.6. 頂点シェーダへの問い合わせ

頂点シェーダに対して、アクティブ状態の頂点属性とユニフォームの情報を問い合わせることができます。

8.6.1. 頂点属性情報の取得

頂点シェーダに入力される頂点データの属性情報を glGetActiveAttrib() で取得することができます。

コード 8-8. glGetActiveAttrib() の定義

```
void glGetActiveAttrib(GLuint program, GLuint index, GLsizei bufsize,
                       GLsizei* length, GLint* size, GLenum* type, char* name);
```

リンク済みでないなど、不正なプログラムオブジェクトが *program* に指定された場合は GL_INVALID_OPERATION エラーが生成されます。

index には、*program* で指定したプログラムオブジェクトに対して glGetProgramiv() の *pname* に GL_ACTIVE_ATTRIBUTES を渡して取得した頂点属性の数 - 1 までの 0 以上の値を指定します。負の値や頂点属性の数以上の値を指定した場合は GL_INVALID_VALUE エラーが生成されます。

bufsize には *name* に指定した配列のサイズを指定してください。負の値を指定した場合は GL_INVALID_VALUE エラーが生成されます。

type には頂点属性の種別が格納されます。*size* には頂点属性のサイズ(頂点属性を表現するのに必要とされる、*type* で示された種別単位での個数)が格納されます。

name には頂点属性の名前が格納されます。頂点属性の名前の文字数が *bufsize* よりも大きい場合は *bufsize* - 1 文字までが格納され、終端文字(NULL)が末尾に追加されます。*length* には *name* に格納された文字数(終端文字は含みません)が格納されます。

8.6.2. ユニフォーム情報の取得

プログラムオブジェクトに登録されているユニフォーム情報を glGetActiveUniform() で取得することができます。頂点シェーダだけに限らず、後述するジオメトリシェーダや予約フラグメントシェーダで使用するユニフォームの情報についても取得することができます。

コード 8-9. glGetActiveUniform() の定義

```
void glGetActiveUniform(GLuint program, GLuint index, GLsizei bufsize,
                        GLsizei* length, GLint* size, GLenum* type, char* name);
```

リンク済みでないなど、不正なプログラムオブジェクトが *program* に指定された場合は GL_INVALID_OPERATION エラーが生成されます。

index には、*program* で指定したプログラムオブジェクトに対して glGetProgramiv() の *pname* に GL_ACTIVE_UNIFORMS を渡して取得したユニフォーム情報の数 – 1 までの 0 以上の値を指定します。負の値やユニフォーム情報の数以上の値を指定した場合は GL_INVALID_VALUE エラーが生成されます。

bufsize には *name* に指定した配列のサイズを指定してください。負の値を指定した場合は GL_INVALID_VALUE エラーが生成されます。

type にはユニフォームに設定する値の種別が格納されます。*size* には設定値のサイズ(ユニフォーム設定値に必要とされる、*type* で示された種別単位での個数)が格納されます。例えば、モデルビュー変換行列のような 4x4 行列の場合、*type* には GL_FLOAT_VEC4 が、*size* には 4 がそれぞれ格納されます。

name にはユニフォームの名前が格納されます。ユニフォームの名前の文字数が *bufsize* よりも大きい場合は *bufsize* – 1 文字までが格納され、終端文字(NULL)が末尾に追加されます。*length* には *name* に格納された文字数(終端文字は含みません)が格納されます。

8.6.3. 設定値の種別について

glGetActiveAttrib() と glGetActiveUniform() で取得することのできる値の種別には、以下のものがあります。

表 8-2. 設定値の種別一覧

種別	型	コンポーネント数	主な使用項目
GL_FLOAT	float	1	バイアス値、スケール値
GL_FLOAT_VEC2	float	2	ビューポート設定
GL_FLOAT_VEC3	float	3	カラー(RGB)、方向ベクトル
GL_FLOAT_VEC4	float	4	カラー(RGBA)、変換行列
GL_INT	int	1	モード設定
GL_INT_VEC3	int	3	コンバイナのソース入力
GL_BOOL	bool	1	有効・無効の指定
GL_SAMPLER_1D	int	1	参照テーブルの指定

※ 実際に各シェーダのユニフォームで使用されている種別の一覧です。

8.7. 複数のユニフォームに対する設定および取得

3DS では、複数のユニフォームを一括で設定・取得する関数を提供しています。

glUniformsDMP() の呼び出しで、現在バインドされているプログラムオブジェクトの複数のユニフォームに対して値を一括で設定することができます。

コード 8-10. ユニフォームの一括設定

```
void glUniformsDMP(GLuint n, GLint* locations, GLsizei* counts,
                   const GLuint* value);
```

n には設定を行うユニフォームの個数を指定します。

locations には *n* 個のユニフォームのロケーション(glGetUniformLocation() で取得)を格納した配列へのポインタを、*counts* には *n* 個のユニフォームの要素数を格納した配列へのポインタを指定します。ユニフォームの要素数は glUniform*() の *count* に相当し、配列型ユニフォームに対しては設定する配列の要素数を、配列型ではない場合は 1 を *counts* で指定する配列に格納してください。

value にはユニフォームに設定する値を格納した配列へのポインタを指定します。ユニフォームごとにデータの個数は異なるため、*value* には *locations* や *counts* と同じインデックスで対象の設定値を格納するとは限りません。設定するユニフォームには GLfloat 型データと GLuint 型データの両方を混在することができます。GLfloat 型のユニフォームに設定する値は、GLfloat 型データを 32 ビットデータとして格納してください。

この関数ではエラーチェックは行われません。すべての引数に関して、不正な値を指定した場合の動作は不定です。

glGetUniformsDMP() の呼び出しで、指定したプログラムオブジェクトの複数のユニフォームに設定されている値を一括で取得することができます。

コード 8-11. ユニフォームの一括取得

```
void glGetUniformsDMP(GLuint program, GLuint n, GLint* locations,
                      GLsizei* counts, GLuint* params);
```

program にはユニフォームの値を取得するプログラムオブジェクトを指定します。

n には取得をするユニフォームの個数を指定します。

locations には *n* 個のユニフォームのロケーション(glGetUniformLocation() で取得)を格納した配列へのポインタを、*counts* には *n* 個のユニフォームの要素数を格納した配列へのポインタを指定します。ユニフォームの要素数には、配列型ユニフォームに対しては値を取得する配列の要素数を、配列型ではない場合は 1 が格納されます。

params にはユニフォームの値を取得する配列へのポインタを指定します。ユニフォームごとにデータの個数は異なるため、*params* には *locations* や *counts* と同じインデックスに対象の設定値が格納されているとは限りません。値を取得するユニフォームには GLfloat 型データと GLuint 型データの両方を混在させることができます。*params* には GLfloat 型のユニフォームから取得した GLfloat 型データは 32 ビットデータとして格納されています。

この関数ではエラーチェックは行われません。すべての引数に関して、不正な値を指定した場合の動作は不定です。

8.8. その他の注意点など

glGetUniformLocation() で取得することのできるユニフォームのロケーションは、glUniform*() や glGetUniform*() によるユニフォーム値へのアクセスに使用することができます。このとき、ロケーションにオフセット値を加えることで、配列型ユニフォームの要素を指定してアクセスすることができます。つまり、ロケーションに 1 を加えた場合、配列型ユニフォームの 2 番目の要素にアクセスすることになります。

ユニフォームのロケーションは glLinkProgram() の呼び出し時に値が決定しますので、本来はプログラムオブジェクトごとに値が異なります。glUniform*() では、ロケーションが関連するプログラムオブジェクトがカレントのプログラムとして設定されていなければエラーを生成します。glGetUniform*() では、ロケーションが関連するプログラムオブジェクトが *program* と異なる場合にエラーを生成します。ただし、予約フラグメントシェーダのユニフォームに限り、ロケーションの値に 0xFFF80000 を論理和で指定した glUniform*() および glGetUniform*() ではエラーが生成されません。

出力する必要のない頂点属性を #pragma output_map の定義に含めないようにしてください。頂点シェーダの実行時には、定義された頂点属性(出力レジスタ)に対して必ず書き込みをしなければならないため、無駄な命令が必要になるからです。また、出力属性によっては GPU の回路の一部に対してクロック制御が行われるため、無駄なバッテリー消費にもつながってしまいます。

頂点バッファを使用しない描画の頂点属性は最大 16 個、頂点バッファを使用する描画の頂点属性は最大 12 個です。ただし、頂点バッファを使用する描画で 12 個の頂点属性を使用する場合は、頂点データの配置に関する制約(「6.8.1. glDrawElements() のみの制約」参照)に注意してください。

頂点シェーダアセンブラでは、常に頂点属性の個数を 16 個まで定義できますが、上記の条件に従って決定される最大個数を超えた場合、描画関数の呼び出し時に GL_INVALID_OPERATION のエラーが生成される可能性があります。

9. ジオメトリシェーダ

ジオメトリシェーダでは、頂点シェーダからの出力を使用してプリミティブ生成の処理を行い、任意の数の頂点を出力することができます。

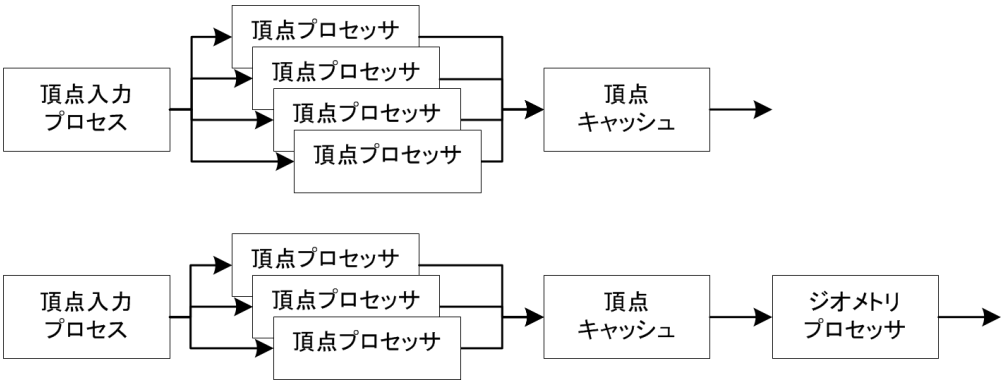
OpenGL ES 2.0 では glDrawElements() や glDrawArrays() の引数に GL_POINTS、GL_LINES、GL_LINE_STRIP、GL_LINE_LOOP を指定することでポイントやラインを描画することができましたが、3DS では、トライアングル(三角形)以外のプリミティブはジオメトリシェーダを使用して生成しなければなりません。トライアングルについては OpenGL ES 2.0 と同様に GL_TRIANGLES、GL_TRIANGLE_STRIP、GL_TRIANGLE_FAN で行いますが、マルチサンプル描画には対応していません。

「5. シェーダプログラム」でも述べたように、3DS では SDK でジオメトリシェーダを提供しています。使用可能なジオメトリシェーダには以下のものがあります。

- ポイントシェーダ
- ラインシェーダ
- シルエットシェーダ
- Catmull-Clark サブディビジョンシェーダ
- ループサブディビジョンシェーダ
- パーティクルシステムシェーダ

これらのジオメトリシェーダを単独で使用することはできません。必ず頂点シェーダもリンクされたバイナリをロードし、頂点シェーダと併せて使用する必要があります。ジオメトリシェーダを使用する場合、4 基ある頂点プロセッサのうちの 1 つがジオメトリプロセッサとして使用されます。また、ブールレジスタの 15 番(b15)はジオメトリシェーダに予約されています。

図 9-1. ジオメトリシェーダの使用・不使用によるプロセッサ構成の違い



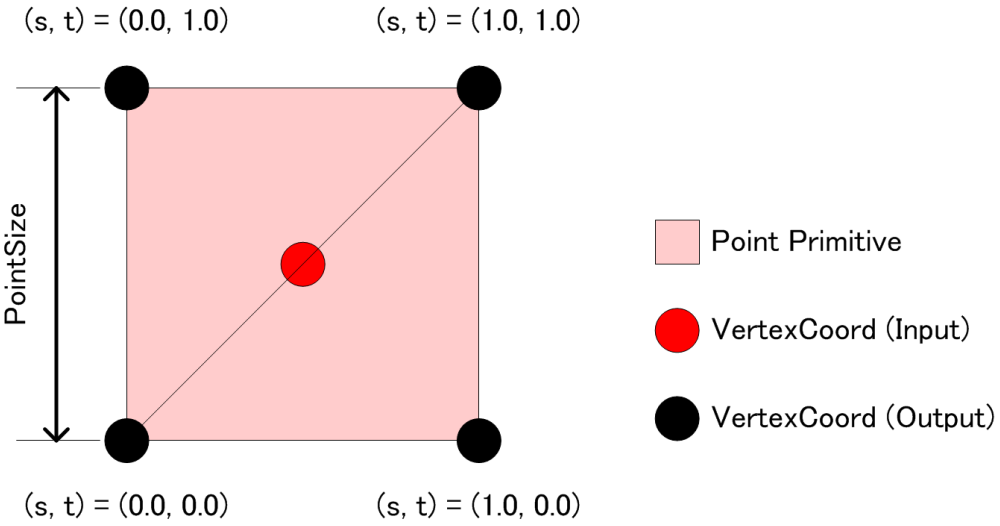
ジオメトリシェーダの入力データには頂点シェーダの出力が使用されます。ジオメトリシェーダが必要とする頂点属性や入力順序、使用可能な頂点属性などはそれぞれのシェーダプログラムで決められており、頂点シェーダはそれらを正しく出力する必要があります。ジオメトリシェーダへの入力は頂点シェーダが出力するレジスタ番号の小さいものから順番に行われます。出力する頂点属性は #pragma output_map で定義しますが、属性名 generic のものはジオメトリシェーダでのみ使用され、後段のプロセス(フラグメント処理など)では扱われない頂点属性となっています。

ジオメトリシェーダには予約ユニフォームが存在し、ポイントシェーダであればビューポートの設定などに使用します。これらの予約ユニフォームはすべて初期値が不定であるため、アプリケーションが必ず値を設定しなければなりません。

9.1. ポイントシェーダ

ポイントシェーダは、頂点座標とポイントサイズから、頂点座標を中心とする辺の長さがポイントサイズの正方形(ポイントプリミティブ)を、2 つの三角形プリミティブを生成して描画します。ポイントプリミティブにテクスチャを貼るため、ポイントシェーダにテクスチャ座標(s と t は下図参照、r は 0.0 固定、q は 1.0 固定)の出力を付加したものがポイントスプライトシェーダです。どちらのシェーダも、グリッドによる調整やマルチサンプル描画には対応していません。

図 9-2. ポイントプリミティブの描画



9.1.1. シェーダファイル

頂点シェーダとリンクさせるシェーダファイルは、ポイントシェーダが必要とする頂点属性以外にフラグメント処理のために出力されている頂点属性の数と、ポイントスプライトシェーダがテクスチャ座標を出力する数から決まります。

ポイント: DMP_point**N**.obj

ポイントスプライト: DMP_pointSprite**N**_**T**.obj

N はポイントシェーダ(ポイントスプライトシェーダ)が必要とするもの以外の頂点属性の数、**T** は使用するテクスチャ座標の個数です。

9.1.2. 予約ユニフォーム

ビューポートとポイントサイズの距離減衰の有効・無効を設定する予約ユニフォームが存在します。これらの予約ユニフォームは初期値を持たないため、必ず値を設定しなければなりません。

ビューポート

ビューポートの予約ユニフォーム(`dmp_Point.viewport`)には、ビューポートの幅と高さの逆数を `glUniform2fv()` で設定します。

距離減衰

ポイントサイズの距離減衰の予約ユニフォーム(`dmp_Point.distanceAttenuation`)には、`GL_TRUE`(距離減衰が有効)または `GL_FALSE`(距離減衰が無効)を `glUniform1i()` で設定します。距離減衰を無効にした場合、ポイントサイズにクリップ座標 Wc が乗算されることでウィンドウ座標変換時の Wc による除算が無効になり、ポイントプリミティブが距離による表示サイズ変化の影響を受けなくなります。

表 9-1. ポイントシェーダの予約ユニフォーム

予約ユニフォーム	種別	設定する値
dmp_Point.viewport	vec2	ビューポートを以下の計算式で指定する。 (1 / viewport.width, 1 / viewport.height)
dmp_Point.distanceAttenuation	bool	ポイントサイズが距離減衰の影響を受けるかどうかを指定する。 GL_TRUE :距離減衰を有効にする GL_FALSE :距離減衰を無効にする

9.1.3. 頂点シェーダの設定

ポイントシェーダがポイントプリミティブを描画するのに必要なのは、頂点座標とポイントサイズの 2 つです。これらの頂点属性を頂点シェーダから出力する順番は、出力レジスタ番号の小さい方から、頂点座標、ポイントサイズです。ポイントスプライトシェーダの場合は、この後にテクスチャ座標が続きます。使用するテクスチャ座標の個数が複数である場合は、1 つの出力レジスタに 2 つのテクスチャ座標を xy と zw に分けてパックしなければなりません。

設定すべき出力頂点属性は、頂点座標が "position"、ポイントサイズが "generic"、テクスチャ座標が "texture0" ～ "texture2" です。

ポイントシェーダで必要な頂点座標とポイントサイズのほかに頂点カラーを頂点シェーダで出力する場合、シェーダファイル DMP_point1.obj をリンクし、頂点シェーダの #pragma output_map は以下のように宣言します。

コード 9-1. ポイントシェーダを使用する場合の出力レジスタ設定例(シェーダアセンブラ)

```
#pragma output_map ( position , o0 )
#pragma output_map ( generic , o1 )
#pragma output_map ( color , o2 )
```

ポイントスプライトシェーダで必要な頂点座標とポイントサイズ、2 つのテクスチャ座標のほかに頂点カラーを頂点シェーダで出力する場合、シェーダファイル DMP_pointSprite1_2.obj をリンクし、頂点シェーダの #pragma output_map は以下のように宣言します。

コード 9-2. ポイントスプライトシェーダを使用する場合の出力レジスタ設定例(シェーダアセンブラ)

```
#pragma output_map ( position , o0 )
#pragma output_map ( generic , o1 )
#pragma output_map ( texture0 , o2.xy )
#pragma output_map ( texture1 , o2.zw )
#pragma output_map ( color , o3 )
```

ポイントスプライトシェーダはテクスチャ座標をポイントスプライト用のテクスチャ座標に置き換えて出力します。頂点シェーダは、ポイントスプライトシェーダで置き換えられるテクスチャ座標の出力レジスタにもダミーの値を書きこまなければならないことに注意が必要です。

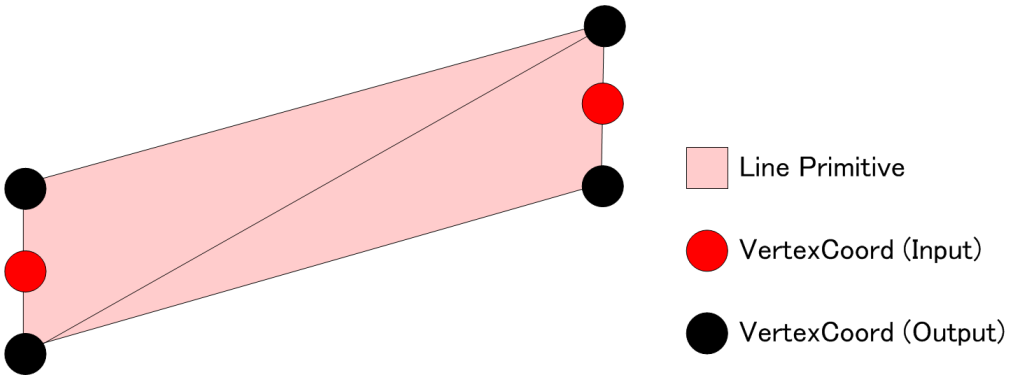
9.1.4. 頂点データの入力

ポイントシェーダによるプリミティブの生成を行いたい場合は、glDrawElements() または glDrawArrays() の *mode* に GL_GEOMETRY_PRIMITIVE_DMP を渡して呼び出してください。

9.2. ラインシェーダ

ラインシェーダは、2 つの頂点座標から頂点座標を結ぶ直線(ラインプリミティブ)を、2 つの三角形プリミティブを生成して描画します。ラインの幅は予約ユニフォームで設定することができます。ラインシェーダはグリッドによる調整やマルチサンプル描画には対応していません。

図 9-3. ラインプリミティブの描画



与えられた頂点座標を結ぶ線分の傾きとラインの幅から、矩形(平行四辺形)を形作る 4 つの頂点座標が生成されます。4 つの頂点座標は入力された頂点座標から Y 方向(線分の傾きによっては X 方向)に生成されます。

9.2.1. シェーダファイル

頂点シェーダとリンクさせるシェーダファイルは、ラインシェーダが必要とする頂点属性以外にフラグメント処理のために出力されている頂点属性の数から決まります。

頂点座標の指定方法により、セパレートラインとストリップラインの 2 種類に分かれています。セパレートラインは 2 つの頂点座標を組にして 1 本のラインを描画します。ストリップラインは最初の 2 つの頂点座標まではセパレートラインと同じですが、次のラインは 2 つ目の頂点座標と 3 つ目の頂点座標を組にして描画されます。つまり、3 つ目以降の頂点は 1 つ前の頂点座標と組になってライン描画に使用され、全体で 1 本に繋がったラインとなります。

セパレートライン: DMP_separateLine**N**.obj

ストリップライン: DMP_stripLine**N**.obj

N はラインシェーダが必要とするもの以外の頂点属性の数です。

9.2.2. 予約ユニフォーム

ラインの幅を設定する予約ユニフォームが存在します。ライン幅の予約ユニフォームは初期値を持たないため、必ず値を設定しなければなりません。

ライン幅

ライン幅の予約ユニフォーム(`dmp_Line.width`)には、ラインの幅とビューポートの幅と高さから計算した値を `glUniform4fv()` で設定します。

表 9-2. ラインシェーダの予約ユニフォーム

予約ユニフォーム	種別	設定する値
dmp_Line.width	vec4	ラインの幅を以下の計算式で指定する。 (viewport.width / line.width, viewport.height / line.width, viewport.width * viewport.height, 2 / line.width)

9.2.3. 頂点シェーダの設定

ラインシェーダがラインプリミティブを描画するのに必要なのは、頂点座標です。頂点座標は出力するレジスタ番号の一番小さな番号で頂点シェーダから出力します。

設定すべき出力頂点属性は、頂点座標の "position" です。

セパレートラインシェーダで必要な頂点座標のほかに頂点カラーを頂点シェーダで出力する場合、シェーダファイル DMP_separateLine1.obj をリンクし、頂点シェーダの #pragma output_map は以下のように宣言します。

コード 9-3. ラインシェーダを使用する場合の出力レジスタ設定例(シェーダアセンブラ)

```
#pragma output_map ( position , o0 )
#pragma output_map ( color , o1 )
```

ストリップラインシェーダの場合、シェーダファイルが DMP_stripLine1.obj であること以外はセパレートラインシェーダと同じです。

9.2.4. 頂点データの入力

ラインシェーダによるプリミティブの生成を行いたい場合は、glDrawElements() または glDrawArrays() の *mode* に GL_GEOMETRY_PRIMITIVE_DMP を渡して呼び出してください。

9.3. シルエットシェーダ

シルエットシェーダはオブジェクトの境界部分のシルエットエッジを生成して描画します。シルエットエッジはオブジェクトの輪郭の描画に利用することができ、シャドウ機能と組み合わせればソフトシャドウの描画に利用することができます。

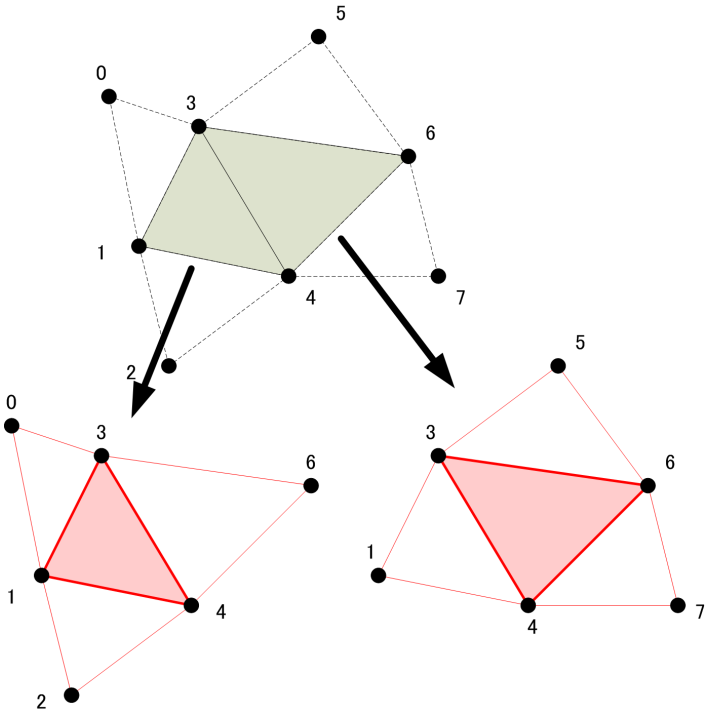
シルエットエッジの生成には、近接付随三角形(triangle with neighborhood、以下 TWN)と呼ばれるプリミティブをシルエットシェーダに投入しなければなりません。

9.3.1. 近接付随三角形(TWN:triangle with neighborhood)

シルエットエッジを描画するオブジェクトを構成する三角形の 1 つに注目して説明します。

注目した三角形を中心三角形と呼び、それぞれのエッジを共有している 3 つの三角形(近接三角形)を含めた 4 つの三角形が TWN です。

図 9-4. 近接付随三角形の例



頂点 3、1、4 を中心三角形とすれば、頂点 0、1、3 の三角形、頂点 2、4、1 の三角形、頂点 6、3、4 の三角形の 3 つと中心三角形が TWN です。頂点 3、4、6 を中心三角形とした場合には、頂点 3、1、4 の三角形は別の TWN を構成する三角形の 1 つになります。

TWN は中心三角形のシルエットエッジを検出するために使われ、オブジェクトを TWN で構成することでオブジェクトのシルエットエッジを描画することができます。

9.3.2. シェーダファイル

TWN の頂点をシルエットシェーダに入力する方法として、通常の三角形を入力するのと同じように、TWN を 1 つずつ入力する方法(シルエットトライアングル)と隣り合う TWN を連続して入力する方法(シルエットストリップ)の 2 つの方法が存在します。

シルエットトライアングル: DMP_silhouetteTriangle.obj

シルエットストリップ: DMP_silhouetteStrip.obj

9.3.3. 予約ユニフォーム

シルエットシェーダの予約ユニフォームには、以下のものが存在します。これらの予約ユニフォームは初期値を持たないため、必ず値を設定しなければなりません。

ポリゴンの表裏判定

シルエットシェーダがポリゴンの表裏を判定する方法(`dmp_Silhouette.frontFaceCCW`)には、オブジェクトの頂点入力で `glFrontFace()` に `GL_CCW` を渡している場合は `GL_TRUE` を、`GL_CW` の場合は `GL_FALSE` を `glUniform1i()` で設定します。

シルエットエッジの幅

シルエットエッジの幅(dmp_Silhouette.width)には、法線の x 方向へ乗算する係数を基に計算した値を glUniform2fv() で設定します。

頂点の w 成分の影響(dmp_Silhouette.scaleByW)には、シルエットエッジの幅に頂点の w 成分を乗算することで距離減衰を無効にするかどうかを設定します。w 成分を乗算する場合は GL_TRUE を、シルエットエッジの w 成分を固定値(1.0)にする場合は GL_FALSE を glUniform1i() で設定します。

シルエットエッジのカラー

シルエットエッジのカラー(dmp_Silhouette.color)には、カラー値(R、G、B、A)を glUniform4fv() で設定します。

オープンエッジ

オープンエッジとは、ほかの三角形とエッジを共有していない中心三角形のエッジのことです。オープンエッジは設定(dmp_Silhouette.acceptEmptyTriangles)によって常に描画させる(GL_TRUE)ことや、描画させない(GL_FALSE)ことができます。

オープンエッジはシルエットエッジとは異なり、頂点の法線を使用せずにラインプリミティブのように描画されます。そのため、角度によってはシルエットエッジとは見え方が異なる場合があります。また、オープンエッジ固有の設定が存在します。

オープンエッジの幅

オープンエッジの幅(dmp_Silhouette.openEdgeWidth)には、ラインシェーダと同じように幅とビューポートの幅と高さから計算した値を glUniform4fv() で設定します。

オープンエッジのカラー

オープンエッジのカラー(dmp_Silhouette.openEdgeColor)には、カラー値(R 、G 、B 、A)を glUniform4fv() で設定します。

オープンエッジの視点方向へのバイアス

視点方向へのバイアス値(dmp_Silhouette.openEdgeDepthBias)には、バイアス値(負値で視点に近付き、正値で遠ざかる)を glUniform1fv() で設定します。オープンエッジの生成には法線が使用されないため、このバイアス値でエッジの見え方を調整します。

オープンエッジの w 成分の乗算

オープンエッジの幅やバイアス値には頂点の w 成分を乗算することができます。それぞれ、dmp_Silhouette.openEdgeWidthScaleByW と dmp_Silhouette.openEdgeDepthBiasScaleByW に GL_TRUE または GL_FALSE を glUniform1i() で設定します。

表 9-3. シルエットシェーダの予約ユニフォーム

予約ユニフォーム	種別	設定する値
dmp_Silhouette.width	vec2	シルエットエッジの幅を以下の計算式で指定する。 (xscale_f, xscale_f * viewport.width / viewport.height) xscale_f:法線の x 成分に乗算する係数
dmp_Silhouette.scaleByW	bool	シルエットエッジに頂点の w 成分を影響させるかどうかを指定する。 GL_TRUE:頂点の w 成分を影響させる GL_FALSE:w 成分は 1.0 固定

dmp_Silhouette.color	vec4	シルエットエッジのカラーを指定する。
dmp_Silhouette.frontFaceCCW	bool	ポリゴンの表裏の判定方法を指定する。 GL_TRUE:CCW(反時計回りが表) GL_FALSE:CW(時計回りが表)
dmp_Silhouette.acceptEmptyTriangles	bool	オープンエッジを描画するかどうかを指定する。 GL_TRUE:描画する GL_FALSE:描画しない
dmp_Silhouette.openEdgeColor	vec4	オープンエッジのカラーを指定する。
dmp_Silhouette.openEdgeWidth	vec4	オープンエッジの幅を以下の計算式で指定する。 (viewport.width / silhouette.width, viewport.height / silhouette.width, viewport.width / viewport.height, 2 / silhouette.width)
dmp_Silhouette.openEdgeDepthBias	float	オープンエッジの視点方向へのバイアス値を指定する。 負の値で視点に近付き、正の値で視点から遠ざかります。
dmp_Silhouette. openEdgeWidthScaleByW	bool	オープンエッジの幅に頂点の w 成分を乗算するかどうかを指定する。 GL_TRUE:w 成分を乗算する GL_FALSE:w 成分を乗算しない
dmp_Silhouette. openEdgeDepthBiasScaleByW	bool	オープンエッジの視点方向へのバイアス値に頂点の w 成分を乗算させるかどうかを指定する。 GL_TRUE:w 成分を乗算する GL_FALSE:w 成分を乗算しない

9.3.4. 頂点シェーダの設定

シルエットシェーダがシルエットエッジを描画するのに必要なのは、頂点座標、頂点カラー、法線の 3 つです。これらの頂点属性を頂点シェーダから出力する順番は、出力レジスタ番号の小さい方から、頂点座標、頂点カラー、法線の順です。

設定すべき出力頂点属性は、頂点座標が "position"、頂点カラーが "color"、法線が "generic" です。

シルエットトライアングルで出力する場合はシェーダファイル DMP_silhouetteTriangle.obj をリンクし、頂点シェーダの #pragma output_map は以下のように宣言します。

コード 9-4. シルエットシェーダを使用する場合の出力レジスタ設定例(シェーダアセンブラ)

```
#pragma output_map ( position , o0 )
#pragma output_map ( color , o1 )
#pragma output_map ( generic , o2 )
```

頂点シェーダ内では、アプリケーションから入力された法線にモデルビュー変換を行い、x 成分と y 成分について正規化された値を出力しなければなりません。つまり、頂点シェーダから出力される法線 n' は、視点座標系の法線 n=(nx, ny, nz) から以下のように正規化されなければなりません。

$$n' = n \Big/ \sqrt{nx^2 + ny^2}$$

シェーダアセンブラで記述すると以下のようになります。aNormal がアプリケーションから入力された法線、vNormal がシルエットシェーダに出力される法線です。

コード 9-5. シルエットシェーダに入力する法線の正規化(シェーダアセンブラ)

```
mov     TEMP_NORM,      CONST_0
dp3     TEMP_NORM.x,    aNormal,        MATRIX_ModelView[0]
dp3     TEMP_NORM.y,    aNormal,        MATRIX_ModelView[1]
mul     TEMP,           TEMP_NORM,      TEMP_NORM
add     TEMP,           TEMP.x,         TEMP.y
rsq     TEMP,           TEMP.x
mul     vNormal,        TEMP_NORM,      TEMP
```

シルエットストリップで出力する場合、シェーダファイルが DMP_silhouetteStrip.obj であること以外はシルエットトライアングルで出力する場合と同じです。

9.3.5. 頂点データの入力

シルエットシェーダによるシルエットエッジの描画を行いたい場合は、glDrawElements() または glDrawArrays() の *mode* に GL_GEOMETRY_PRIMITIVE_DMP を渡して呼び出してください。また、描画の前に glDisable(GL_CULL_FACE) でカリングを無効にしなければなりません。

TWN の性質上、シルエットシェーダへの頂点データの入力には、頂点インデックスの使用を推奨します。以降の説明では頂点インデックスの使用を前提としています。なお、頂点インデックスを使用しない場合は、TWN の頂点入力の規則に従って頂点データを並べる必要があります。

9.3.6. シルエットトライアングルのインデックス

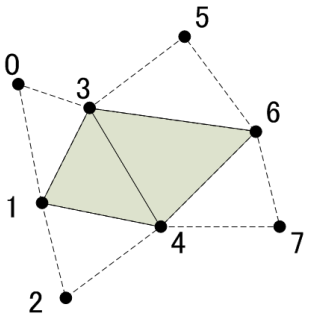
シルエットトライアングルは TWN を 1 つずつシェーダに入力する方法で、6 頂点で 1 つの TWN を入力することになります。

入力する頂点の順序は以下のとおりです。

1. 中心三角形の第 1 頂点
2. 中心三角形の第 2 頂点
3. 中心三角形の第 1 頂点と第 2 頂点が形成するエッジを共有する近接三角形のもう 1 つの頂点
4. 中心三角形の第 3 頂点
5. 中心三角形の第 1 頂点と第 3 頂点が形成するエッジを共有する近接三角形のもう 1 つの頂点
6. 中心三角形の第 2 頂点と第 3 頂点が形成するエッジを共有する近接三角形のもう 1 つの頂点

以下の図のオブジェクトを例にインデックスの指定を説明します。

図 9-5. TWN の例



ポリゴンの表裏判定は CCW が設定されていると仮定します。

表になるように選択した中心三角形を(1、4、3)と(3、4、6)とすればインデックスの指定は、

(1、4、3)のインデックス:1、4、2、3、0、6
(3、4、6)のインデックス:3、4、1、6、5、7

のようになります。

中心三角形が縮退している場合、シルエットエッジは生成されません。また、3 つ以上の三角形でエッジを共有するような三角形は考慮していません。

9.3.7. シルエットストリップのインデックス

シルエットストリップは TWN を連続してシェーダに入力する方法で、最初の TWN で 6 頂点の入力が必要ですが、以降は 2 頂点の入力で 1 つの TWN を入力することができます。そのため、同じモデルを描画する場合でも、シルエットトライアングルの倍以上の性能を発揮すると予想されます。

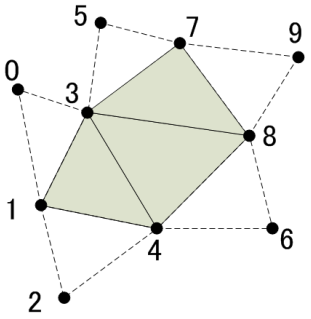
中心三角形の第 2 頂点と第 3 頂点と、その 2 頂点が形成するエッジを共有する頂点の 3 頂点が次の TWN の中心三角形であれば、このシルエットストリップによる入力を行うことができます。つまり、前の TWN の最後の近接三角形が今回の中心三角形になるように頂点を入力していくことになります。

入力する頂点の順序は以下のとおりです。

1. 最初の 6 頂点はシルエットトライアングルと同じです。中心三角形の第 2 頂点、第 3 頂点、最後に指定した頂点が次の中心三角形の第 1 ～第 3 頂点となります
2. 中心三角形の第 1 頂点と第 3 頂点が形成するエッジを共有する近接三角形のもう 1 つの頂点
3. 中心三角形の第 2 頂点と第 3 頂点が形成するエッジを共有する近接三角形のもう 1 つの頂点
4. 中心三角形の第 2 頂点、第 3 頂点、3. で指定した頂点が新たな中心三角形の第 1 ～第 3 頂点となり、以後は 2. と 3. の手順を繰り返します

シルエットストリップによる入力を終了する、もしくは次のシルエットストリップを入力したい場合は、3. で指定する頂点の前に中心三角形の第 3 頂点と同じ頂点を入力して終端指定を行います。

図 9-6. シルエットストリップのインデックスの例



上図のオブジェクトを例に(3、8、7)が終端の中心三角形とすると、

シルエットストリップによるインデックス:1、4、2、3、0、8、6、7、5、**7**、9

のようになります。強調してある **"7"** の頂点が終端指定です。

終端指定を行ったあとに続けてシルエットストリップの入力を行うには、新たな TWN を構成している 6 頂点の入力から開始します。そのとき、新たなシルエットストリップの最初の中心三角形が `glFrontFace` での指定と逆(裏)になる場合は、最初の頂点の入力で同じ頂点を 2 回入力してください。つまり、先の例であれば、

裏から始まるシルエットストリップによるインデックス:1、1、4、2、3、0、8、6、7、5、7、9

のようになります。

終端指定は複数のシルエットストリップを入力する際の区切りに使用しますが、最後のシルエットストリップの入力で終端指定を行わずに、シルエットストリップに接続するような三角形を続けて入力した場合にシルエットエッジが二重に描画される可能性があります。また、シルエットに対してアルファブレンドなどを行う場合は、すべてのシルエットストリップの入力で終端指定を行う必要があります。

中心三角形が縮退している場合、終端指定が行われたと判定されることに注意してください。

9.3.8. オープンエッジ

オープンエッジとは、ほかの三角形とエッジを共有していない中心三角形のエッジのことであることは説明しました。エッジを共有する三角形が存在しないため、そのインデックスの指定方法は通常と異なり、中心三角形のもう 1 つの頂点を指定することになります。ちょうど近接三角形を折り返したような感じです。

図 9-5 のオブジェクトで頂点 0 が存在しない場合を例にすると、

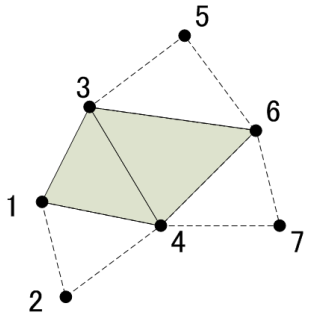
シルエットトライアングルでの(1、4、3)のインデックス:1、4、2、3、**4**、6

シルエットストリップでのインデックス:1、4、2、3、**4**、6、5、6、7

のようになります。

オープンエッジには、シルエットエッジとは異なる設定を行わなければなりません。詳細については「9.3.3. 予約ユニフォーム」を参照してください。

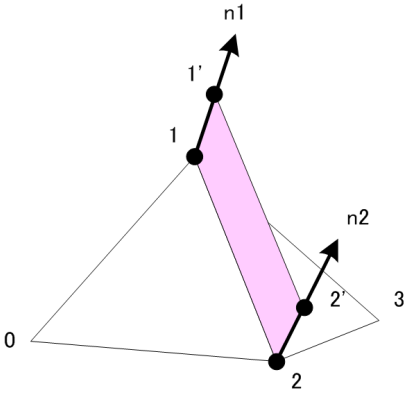
図 9-7. オープンエッジのインデックスの例



9.3.9. シルエットエッジの生成

シルエットエッジは、TWN の中心三角形の向きが表、近接三角形の向きが裏であるときに、中心三角形のエッジ上に新たな矩形ポリゴンを生成して描画されます。中心三角形と近接三角形が共有する 2 つの頂点(1 と 2)と、その頂点から法線(n1 と n2)の方向に追加された 2 つの頂点(1' と 2')によって(2 つの三角形ポリゴンで構成される)矩形ポリゴンが形成されます。

図 9-8. シルエットエッジを形成する矩形ポリゴン



中心三角形の頂点の座標を(x, y, z, w)、法線を(nx, ny, nz)とすると、追加される頂点の座標(x', y', z', w')は以下の式で計算されます。

x' = x + x_factor * nx * w_scale
y' = y + y_factor * ny * w_scale
z' = z
w' = w

x_factor と y_factor、w_scale に適用される値は、シルエットエッジの幅に関する予約ユニフォームで設定します。

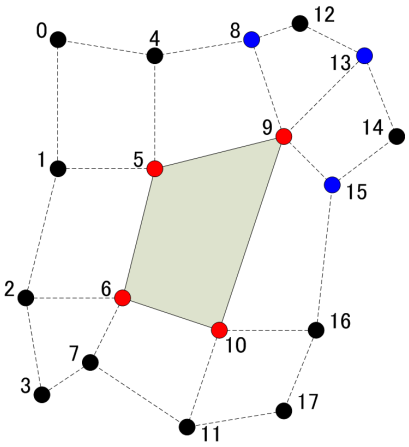
9.4. Catmull-Clark サブディビジョンシェーダ

Catmull-Clark サブディビジョンシェーダは、四角形ポリゴンとその周囲の頂点を用いて頂点群を分割し、ポリゴンを滑らかにするシェーダです。以降、この節でサブディビジョンと記述している場合は、この Catmull-Clark サブディビジョンのことを指しています。

9.4.1. サブディビジョンパッチ

ポリゴンへのサブディビジョンの適用は、**四角形のみで構成されたポリゴンの集合**(Catmull-Clark サブディビジョンパッチ。以降、サブディビジョンパッチ)をシェーダに投入することで行われます。サブディビジョンパッチは、対象となる四角形(中心四角形)とその 4 頂点が形成するエッジを共有している四角形群から構成されています。

図 9-9. Catmull-Clark サブディビジョンパッチの例



サブディビジョンパッチは、**すべてが四角形で構成されているポリゴンモデルにしか適用できません**。

中心四角形の各頂点は通常、頂点 5、6、10 のように 4 本のエッジを形成しますが、頂点 9 のように 5 本のエッジを形成する頂点や 3 本しかエッジを形成しない頂点が存在する場合、その頂点を特異点と呼び、形成するエッジの本数を価数と呼びます。**サブディビジョンパッチが許容する特異点は中心四角形の 1 つのみで、その価数は 3 ～ 12 です**。

サブディビジョンパッチの中心四角形に特異点が含まれている場合、特異点からインデックスの指定を開始しなければなりません。また、同じ特異点を含む中心四角形のサブディビジョンパッチがほかに存在する場合、それらのサブディビジョンパッチは連続して入力されなければなりません。場合によってはメッシュに穴が開いてしまいます。

9.4.2. シェーダファイル

頂点シェーダとリンクさせるシェーダファイルは、サブディビジョンシェーダが必要とする頂点属性以外でフラグメント処理のために出力されている頂点属性の数から決まります。

サブディビジョン: DMP_subdivision**N**.obj

N はサブディビジョンシェーダが必要とするもの以外の頂点属性の数(1 ～ 6)です。

9.4.3. 予約ユニフォーム

サブディビジョンシェーダの予約ユニフォームには、以下のものが存在します。これらの予約ユニフォームは初期値を持たないため、必ず値を設定しなければなりません。

細分割レベル

細分割レベル(dmp_Subdivision.level)には、シェーダ処理による分割の細かさを glUniform1f() で設定します。レベルが高い(数値が大きい)ほど細かく分割されます。最も低い 0 では、サブディビジョンパッチの中心に頂点を 1 つ加え、サブディビジョンパッチに含まれている元の頂点座標の調整が行われます。

クォータニオンの使用

シェーダによって新たに生成された頂点の、頂点座標以外の頂点属性は補間されて出力されますが、クォータニオンは特別な分割処理が必要であるため、シェーダにはクォータニオンが投入されることを通知しなければなりません。

クォータニオンの投入の有無(dmp_Subdivision.fragmentLightingEnabled)には、クォータニオンを投入する場合は GL_TRUE を、投入しない場合は GL_FALSE を glUniform1i() で設定します。

表 9-4. Catmull-Clark サブディビジョンシェーダの予約ユニフォーム

予約ユニフォーム	種別	設定する値
dmp_Subdivision.level	float	細分割レベルを指定する。 0(細分割レベル:低) 1 2(細分割レベル:高)
dmp_Subdivision.fragmentLightingEnabled	bool	フラグメントライティングで必要となるクォータニオンを投入するかどうかを指定する。 GL_TRUE:クォータニオンを投入する GL_FALSE:クォータニオンを投入しない

9.4.4. 頂点シェーダの設定

サブディビジョンシェーダが必要とする頂点属性は、頂点座標です。頂点座標は出力するレジスタ番号の一番小さな番号で頂点シェーダから出力します。

設定すべき出力頂点属性は、頂点座標の "position" とそのほかの属性のいずれか 1 つ、合計で 2 つ以上です。

サブディビジョンシェーダで必要な頂点座標のほかに頂点カラーを頂点シェーダで出力する場合、シェーダファイル DMP_subdivision1.obj をリンクし、頂点シェーダの #pragma output_map は以下のように宣言します。

コード 9-6. Catmull-Clark サブディビジョンシェーダを使用する場合の出力レジスタ設定例(シェーダアセンブラ)

```
#pragma output_map ( position , o0 )
#pragma output_map ( color , o1 )
```

フラグメントライティングでクォータニオンを必要とする場合、クォータニオンは頂点座標を出力するレジスタの次に小さな番号の出力レジスタで出力しなければなりません。

コード 9-7. クォータニオンをサブディビジョンシェーダで使用する場合の出力レジスタ設定例(シェーダアセンブラ)

```
#pragma output_map ( position , o0 )
#pragma output_map ( quaternion , o1 )
#pragma output_map ( color , o2 )
```

9.4.5. 頂点データの入力

Catmull-Clark サブディビジョンシェーダによる細分割処理を行いたい場合は、glDrawElements() の *mode* に GL_GEOMETRY_PRIMITIVE_DMP を渡して呼び出してください。glDrawArrays() は使用できません。頂点インデックスも頂点バッファを経由して使用しなければなりません。

9.4.6. サブディビジョンパッチのインデックス

サブディビジョンパッチに含まれる頂点の数は、特異点の価数(3 ～ 12)によって変化します。そのため、サブディビジョンパッチは入力する頂点数が固定ではありません。

最初にサブディビジョンパッチのサイズを入力します。サブディビジョンパッチのサイズには、含まれる頂点の数(特異点の価数× 2 + 8。最小 14、最大 32、特異点がなければ 16)を指定します。32 を超えるサイズのパッチが入力された場合の動作は不定です。

サブディビジョンパッチのインデックスは以下の順で指定します。

1. 中心四角形の 4 頂点(順番は glFrontFace() での指定に従い、特異点があれば特異点から開始する)
2. 中心四角形の周囲の頂点(順番は glFrontFace() での指定に従う)

図 9-9 のサブディビジョンパッチを例に反時計回りが表とすると、

サブディビジョンパッチのインデックス: **18**、9、5、6、10、8、4、0、1、2、3、7、11、17、16、15、14、13、12

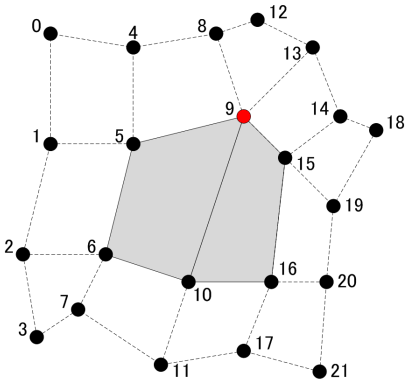
のようになります。先頭の 18 はサブディビジョンパッチのサイズです。

この後に続けて、特異点の頂点 9 を含む中心四角形のサブディビジョンパッチを投入していきます。下図の例では、上記のサブディビジョンパッチのインデックスに続いて

サブディビジョンパッチのインデックス:…、**18**、9、10、16、15、5、6、7、11、17、21、20、19、18、14、13、12、8、4

となります。

図 9-10. サブディビジョンパッチのインデックスの例



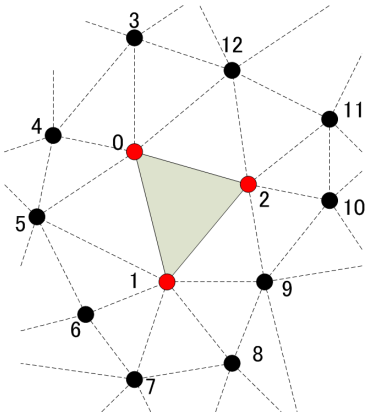
9.5. ループサブディビジョンシェーダ

ループサブディビジョンシェーダは、三角形ポリゴンとその周囲の頂点を用いて頂点群を分割し、ポリゴンを滑らかにするシェーダです。以降、この節でサブディビジョンと記述している場合は、このループサブディビジョンのことを指しています。

9.5.1. サブディビジョンパッチ

ポリゴンへのサブディビジョンの適用は、ループサブディビジョンパッチ(以降、サブディビジョンパッチ)をシェーダに投入することで行われます。サブディビジョンパッチは、対象となる三角形(中心三角形)とその 3 頂点それぞれが形成するエッジを共有している頂点群から構成されています。

図 9-11. ループサブディビジョンパッチの例



中心三角形の各頂点が形成するエッジの本数を価数と呼びます。

図 9-11 で頂点 0、1、2 を中心三角形とした場合、頂点 0 の価数は 6、頂点 1 は 7、頂点 2 は 6 となります。**サブディビジョンパッチが許容する各頂点の価数は 3 ～ 12 で、中心三角形の 3 頂点の価数の合計が 29 以下でなければなりません。**

もし、価数が 2 の頂点(中心三角形のほかの頂点以外にエッジを共有する頂点がない)をサブディビジョンパッチに含めたい場合は、仮想の頂点を増やすことで対応することができます。

9.5.2. シェーダファイル

頂点シェーダとリンクさせるシェーダファイルは、価数以外にサブディビジョンシェーダへ出力する頂点属性を設定した**出力レジスタの数**から決まります。頂点座標は必ず出力しなければならないため、出力レジスタの数は 1 以上になります。

サブディビジョン：DMP_loopSubdivision**N**.obj

N は価数以外に頂点属性を設定した出力レジスタの数（1 ～ 4）です。

9.5.3. 予約ユニフォーム

ループサブディビジョンシェーダの予約ユニフォームは、Catmull-Clark サブディビジョンシェーダと同じです。詳細については、「9.4.3. 予約ユニフォーム」を参照してください。ループサブディビジョンシェーダの予約ユニフォームは初期値を持たないため、必ず値を設定しなければなりません。

細分割レベルが 0 の場合は新たな頂点は追加されませんが、サブディビジョンパッチに含まれている元の頂点座標の調整が行われます。

9.5.4. 頂点シェーダの設定

サブディビジョンシェーダの処理に必要なのは、頂点座標と価数の 2 つです。これらの頂点属性を頂点シェーダから出力する順番は、出力レジスタ番号の小さい方から、頂点座標、そのほかの頂点属性（あれば）、価数の順です。

設定すべき出力頂点属性は、頂点座標が "position"、価数が "generic" です。

サブディビジョンシェーダで必要な頂点座標のほかに頂点カラーを頂点シェーダで出力する場合、シェーダファイル DMP_loopSubdivision2.obj（頂点座標も出力レジスタ数に含まれるため）をリンクし、頂点シェーダの `#pragma output_map` は以下のように宣言します。

コード 9-8. ループサブディビジョンシェーダを使用する場合の出力レジスタ設定例（シェーダアセンブラ）

```
#pragma output_map ( position , o0 )
#pragma output_map ( color , o1 )
#pragma output_map ( generic, o2 )
```

フラグメントライティングでクォータニオンを必要とする場合、クォータニオンは頂点座標を出力するレジスタの次に小さな番号の出力レジスタで出力しなければなりません。また、価数以外の出力レジスタの数が 4 までに制限されていますので、価数以外の頂点属性が 5 つ以上存在する場合は、1 つのレジスタに複数の頂点属性をパックする必要があります。ただし、クォータニオンをほかの頂点属性とパックすることはできません。

コード 9-9. 多数の頂点属性をサブディビジョンシェーダで使用する場合の出力レジスタ設定例（シェーダアセンブラ）

```
#pragma output_map ( position , o0 )
#pragma output_map ( quaternion , o1 )
#pragma output_map ( color , o2 )
#pragma output_map ( texture0 , o3.xy )
#pragma output_map ( texture1 , o3.zw )
#pragma output_map ( generic , o4 )
```

9.5.5. 頂点データの入力

ループサブディビジョンシェーダによる細分割処理を行いたい場合は、`glDrawElements()` の *mode* に `GL_GEOMETRY_PRIMITIVE_DMP` を渡して呼び出してください。`glDrawArrays()` は使用できません。頂点インデックスも頂点バッファを経由して使用しなければなりません。

9.5.6. サブディビジョンパッチのインデックス

サブディビジョンパッチに含まれる頂点の数は、中心三角形の価数の合計によって変化します。そのため、サブディビジョンパッチは入力する頂点数が固定ではありません。

最初にサブディビジョンパッチのサイズを入力します。サブディビジョンパッチのサイズには、中心三角形を構成する頂点の価数の合計 + 3 を指定します。

サブディビジョンパッチのインデックスは以下の順で指定します。

1. 中心三角形を構成する 3 頂点(順番は `glFrontFace()` での指定に従い、先頭から v0、v1、v2 とする)のインデックス
2. v0 とエッジを共有する頂点すべて(順番は任意ですが、同じ頂点を含むほかのサブディビジョンパッチも同じ順序でなければなりません)
3. v1 とエッジを共有する頂点すべて(順番は任意ですが、同じ頂点を含むほかのサブディビジョンパッチも同じ順序でなければなりません)
4. v2 とエッジを共有する頂点すべて(順番は任意ですが、同じ頂点を含むほかのサブディビジョンパッチも同じ順序でなければなりません)
5. 固定値 12 と中心三角形の 3 頂点("12"、v0、v1、v2 の順)
6. v0 と v2 と三角形を形成する中心三角形以外の頂点(e00 とする)
7. v0 と v1 と三角形を形成する中心三角形以外の頂点(e10 とする)
8. v1 と v2 と三角形を形成する中心三角形以外の頂点(e20 とする)
9. v0 とエッジを共有し、e00 から**反時計回りで隣接**している頂点
10. v1 とエッジを共有し、e10 から**反時計回りで隣接**している頂点
11. v2 とエッジを共有し、e20 から**反時計回りで隣接**している頂点
12. v0 とエッジを共有し、e10 から**時計回りで隣接**している頂点
13. v1 とエッジを共有し、e20 から**時計回りで隣接**している頂点
14. v2 とエッジを共有し、e00 から**時計回りで隣接**している頂点

図 9-11 のサブディビジョンパッチを例に反時計回りが表とすると、

サブディビジョンパッチのインデックス:

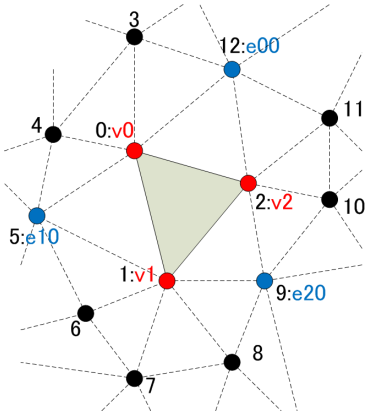
22、0、1、2、 1、2、12、3、4、5、 2、0、5、6、7、8、9、 0、1、9、10、11、12、
12、0、1、2、 12、5、9、 3、6、10、 4、11、8

のようになります。1 行目の 22 はサブディビジョンパッチのサイズです。2 行目の 12 が固定値です。

ほかのサブディビジョンパッチで v0 と同じ頂点を中心三角形に使用する場合は、今回の順番(1、2、12、3、4、5)でエッジを共有する頂点を指定しなければなりません。v1 と v2 も同様です。

サブディビジョンパッチでは頂点が重複して指定されることになりますが、頂点処理後のキャッシュにヒットするため、事実上パフォーマンスに関するペナルティは発生しません。

図 9-12. サブディビジョンパッチのインデックスの例



9.6. パーティクルシステムシェーダ

パーティクルシステムシェーダは、ベジェ曲線に沿って大量のポイントスプライト(パーティクル)を描画するパーティクルシステムのためのシェーダです。

パーティクルが描画されるベジェ曲線はシェーダに入力される 4 つの制御点で定義されます。制御点はそれぞれが持つバウンディングボックス内でランダムな位置に決定され、それによってベジェ曲線も変化します。

パーティクルのカラー、サイズ、テクスチャ座標の回転角度などはベジェ曲線上の位置で補間されて決定されます。

9.6.1. シェーダファイル

頂点シェーダとリンクさせるシェーダファイルは、パーティクルシステムシェーダがサポートする機能から決まります。

パーティクルシステム: DMP_particleSystem_**X**_**X**_**X**_**X**.obj

X は 0 または 1 でパーティクルシステムの機能を制御し、先頭から、パーティクル時間のクランプの ON/OFF、テクスチャ座標回転の ON/OFF、パーティクルカラーの RGBA 成分使用/アルファ成分のみ使用、テクスチャ座標 2 の出力を制御することができます。必ずしも、0 が OFF、1 が ON ではありませんので、下表を参照してリンクするシェーダファイルを決定してください。

表 9-5. パーティクルシステムの機能とシェーダファイル名の対応

ファイル名	時間のクランプ	テクスチャ座標回転	RGBA カラー	テクスチャ座標 2
*_0_0_0_0.obj	クランプする	使用する	なし(アルファのみ)	出力しない
*_0_0_0_1.obj	クランプする	使用する	なし(アルファのみ)	出力する
*_0_0_1_0.obj	クランプする	使用する	使用する	出力しない
*_0_0_1_1.obj	クランプする	使用する	使用する	出力する
*_0_1_0_0.obj	クランプする	なし	なし(アルファのみ)	出力しない
*_0_1_0_1.obj	クランプする	なし	なし(アルファのみ)	出力する
*_0_1_1_0.obj	クランプする	なし	使用する	出力しない
*_0_1_1_1.obj	クランプする	なし	使用する	出力する
*_1_0_0_0.obj	なし	使用する	なし(アルファのみ)	出力しない

*_1_0_0_1.obj	なし	使用する	なし(アルファのみ)	出力する
*_1_0_1_0.obj	なし	使用する	使用する	出力しない
*_1_0_1_1.obj	なし	使用する	使用する	出力する
*_1_1_0_0.obj	なし	なし	なし(アルファのみ)	出力しない
*_1_1_0_1.obj	なし	なし	なし(アルファのみ)	出力する
*_1_1_1_0.obj	なし	なし	使用する	出力しない
*_1_1_1_1.obj	なし	なし	使用する	出力する

表中の "*" は "DMP_particleSystem" の略です。

9.6.2. 予約ユニフォーム

パーティクルシステムシェーダの予約ユニフォームには、以下のものが存在します。これらの予約ユニフォームは初期値を持たないため、必ず値を設定しなければなりません。

カラー

パーティクルカラー(dmp_PartSys.color)には、各制御点でのパーティクルのカラーを 4x4 行列(1 行目に第 1 制御点の RGBA、2 行目に第 2 制御点の RGBA、以降同様)にして、glUniformMatrix4fv() で設定します。この設定はリンクするシェーダファイルが RGBA カラーを使用するファイル(DMP_particleSystem_X_X_1_X.obj)でのみ有効です。

$$color = \begin{pmatrix} R_1 & G_1 & B_1 & A_1 \\ R_2 & G_2 & B_2 & A_2 \\ R_3 & G_3 & B_3 & A_3 \\ R_4 & G_4 & B_4 & A_4 \end{pmatrix}$$

RGBA カラーを使用する場合は、アルファ成分のみを使用する場合に比べてパフォーマンスが悪くなります。カラー成分を使用しない場合は、アルファ成分のみを使用するシェーダファイル(DMP_particleSystem_X_X_0_X.obj)をリンクすることをお勧めします。

アスペクト

パーティクルのアスペクト(dmp_PartSys.aspect)には、各制御点でのパーティクルのサイズ、テクスチャ座標の回転角、テクスチャ座標のスケーリング、アルファ成分値を 4x4 行列(1 行目から、第 1、第 2、第 3、第 4 制御点)にして、glUniformMatrix4fv() で設定します。

$$aspect = \begin{pmatrix} size_1 & rotate_1 & scale_1 & alpha_1 \\ size_2 & rotate_2 & scale_2 & alpha_2 \\ size_3 & rotate_3 & scale_3 & alpha_3 \\ size_4 & rotate_4 & scale_4 & alpha_4 \end{pmatrix}$$

サイズ

パーティクルのサイズ(アスペクトの第 1 列)には、1.0 以上の値を設定します。

パーティクルのサイズの最小値と最大値はパーティクルサイズ(dmp_PartSys.pointSize)に glUniform2fv() で設定します。パーティクルの描画は画面サイズを考慮しなければなりませんので、ビューポート(dmp_PartSys.viewport)にはビューポートの幅と高さの逆数を glUniform2fv() で設定してください。また、パー

ティクルのサイズに距離減衰を適用する場合は、距離減衰の因子(dmp_PartSys.distanceAttenuation)に以下の減衰後サイズの計算式で使用する減衰係数(a、b、c)を glUniform3fv() で設定してください。

$$derived\_size = size \times \sqrt{\frac{1}{a + b \times d + c \times d^2}}$$

derived_size は距離減衰適用後のサイズ、size は元のサイズ、d は視点からの距離です。

テクスチャ座標

テクスチャ座標に対して、回転(アスペクトの第 2 列)とスケーリング(アスペクトの第 3 列)を各制御点で指定することができます。

パーティクルシステムが出力するテクスチャ座標は、テクスチャ座標 0 とテクスチャ座標 2 です。回転はどちらのテクスチャ座標でもサポートしていますが、スケーリングはテクスチャ座標 2 のみがサポートしています。これらの設定はリンクするシェーダファイルで有効・無効を制御することができます。

- テクスチャ座標 2 の出力する(DMP_particleSystem_X_X_X_1.obj)・出力しない(DMP_particleSystem_X_X_X_0.obj)
- テクスチャ座標の回転およびスケーリングを行う(DMP_particleSystem_X_0_X_X.obj)・行わない(DMP_particleSystem_X_1_X_X.obj)

回転角はラジアンで設定します。時計回りが正の値です。

テクスチャ座標 0 に出力されるテクスチャ座標は、パーティクルの左下が(0, 0)、右下が(1, 0)、左上が(0, 1)、右上が(1, 1)です。テクスチャ座標 2 に出力されるテクスチャ座標は、パーティクルの左下が(−1, −1)、右下が(1, −1)、左上が(−1, 1)、右上が(1, 1)です。

設定した回転角を A、スケーリング値を R とすれば、テクスチャ座標 0 は、

左下:(0.5×(1.0 + (−cosA + sinA)), 0.5×(1.0 + (−cosA − sinA)))
右下:(0.5×(1.0 + (cosA + sinA)), 0.5×(1.0 + (−cosA + sinA)))
左上:(0.5×(1.0 + (−cosA − sinA)), 0.5×(1.0 + (cosA − sinA)))
右上:(0.5×(1.0 + (cosA − sinA)), 0.5×(1.0 + (cosA + sinA)))

となり、テクスチャ座標 2 は、

左下:(R×(−cosA + sinA), R×(−cosA − sinA))
右下:(R×(cosA + sinA), R×(−cosA + sinA))
左上:(R×(−cosA − sinA), R×(cosA − sinA))
右上:(R×(cosA − sinA), R×(cosA + sinA))

となります。

アルファ成分

パーティクルのアルファ成分(アスペクトの第 4 列)には、0.0 ～ 1.0 の値を設定します。アルファ成分のみを使用するシェーダファイル(DMP_particleSystem_X_X_0_X.obj)をリンクした場合は、この設定が使用されます。

発生数

パーティクルの最大発生数(dmp_PartSys.countMax)には、(発生させるパーティクルの最大数 − 1)の値を glUniform1fv() で設定します。0.0 以上の値を設定しなければなりません。ただし、シェーダプログラムの実装により、パーティクルの発生数は最大で 255 個に制限されており、この予約ユニフォームで 256 より大きい値を設定してもパーティクルは 255 個までしか発生しません。

実行時間と速度

パーティクルシステムには時間の概念があります。パーティクルシステムの時間(dmp_PartSys.time)には、現在時間を glUniform1fv() で設定します。この現在時間はパーティクルごとに実行時間へとランダムに変換され、パーティクルは実行時間の経過で第 1 制御点から第 4 制御点へと移動していきます。パーティクルは実行時間が 0.0 の時点で第 1 制御点に発生し、1.0 の時点で第 4 制御点に到達します。

実行時間をクランプするシェーダファイル(DMP_particleSystem_0_X_X_X.obj)をリンクした場合、実行時間が 1.0 以上になったパーティクルは描画されなくなります。そのため、アプリケーションで実行時間をリセットしなければ、単純に時間を経過させていると、ある時点からパーティクルが発生しなくなってしまいます。

実行時間をクランプしないシェーダファイル(DMP_particleSystem_1_X_X_X.obj)をリンクした場合、実行時間が 0.0 ～ 1.0 の範囲を繰り返します。つまり、第 4 制御点に到達したパーティクルは再び第 1 制御点から発生します。

パーティクルの速度(dmp_PartSys.speed)には、パーティクルが進む速度を glUniform1fv() で設定します。

ランダム値

制御点のバウンディングボックス内での位置やパーティクルの実行時間の決定には、ランダム関数が使用されています。アプリケーションからは、このランダム関数で使用するランダムシードと係数を指定することができます。

ランダム関数は以下のような乱数発生関数(Pseudo random number generator のアルゴリズム)に似た実装がなされています。

$$X_{N+1} = (aX_N + b) \bmod m$$

ランダムシード(dmp_PartSys.randSeed)には、ベジェ曲線の x 成分、y 成分、z 成分、パーティクルの実行時間で上式の X_0 にあたる値を順番に配列にして glUniform4fv() で設定します。

ランダム関数の係数(dmp_PartSys.randomCore)には、上式の(a, b, m, 1/m)を glUniform4fv() で設定します。

表 9-6. パーティクルシステムシェーダの予約ユニフォーム

予約ユニフォーム	種別	設定する値
dmp_PartSys.color	mat4	各制御点でのカラーを指定する。 (R, G, B, A) * 4vec 各成分は 0.0 ～ 1.0
dmp_PartSys.aspect	mat4	各制御点でのアスペクトを指定する。 (perticle_size, rotation_angle, scale, alpha) * 4vec perticle_size は 1.0 以上、alpha は 0.0 ～ 1.0
dmp_PartSys.time	float	パーティクルシステムの現在時間を指定する。
dmp_PartSys.speed	float	パーティクルの進む速度を指定する。 0.0 以上
dmp_PartSys.countMax	float	パーティクルの最大発生数を(最大発生数 – 1)で指定する。 0.0 以上
dmp_PartSys.randSeed	vec4	各ランダム関数のランダムシードを指定する。 (ベジェ曲線の x 成分にかかる値, ベジェ曲線の y 成分にかかる値, ベジェ曲線の z 成分にかかる値, パーティクルの実行時間にかかる値)
dmp_PartSys.randomCore	vec4	ランダム関数の係数を指定する。 (a, b, m, 1/m)

dmp_PartSys.distanceAttenuation	vec3	距離減衰の因子を指定する。
dmp_PartSys.viewport	vec2	ビューポートを以下の計算式で指定する。 (1 / viewport.width, 1 / viewport.height)
dmp_PartSys.pointSize	vec2	パーティクルサイズの最小値と最大値を指定する。 それぞれ 0.0 以上

9.6.3. 頂点シェーダの設定

パーティクルシステムシェーダの処理に必要なのは、1 つの制御点に対して頂点座標と頂点座標を中心とするバウンディングボックスの xyz 成分の半径サイズをクリップ座標系に変換した 4x4 行列です。これらの頂点属性を頂点シェーダから出力する順番は、出力レジスタ番号の小さい方から、頂点座標、変換後行列の 1 行目、2 行目、3 行目、4 行目の順です。

設定すべき出力頂点属性は、頂点座標が "position"、変換後の行列が "generic" です。

コード 9-10. パーティクルシステムシェーダを使用する場合の出力レジスタ設定例(シェーダアセンブラ)

```
#pragma output_map ( position , o0 )
#pragma output_map ( generic , o1 )
#pragma output_map ( generic , o2 )
#pragma output_map ( generic , o3 )
#pragma output_map ( generic , o4 )
```

バウンディングボックスの xyz 成分それぞれの半径サイズを Rx、Ry、Rz とし、射影変換行列を Mproj、モデルビュー変換行列を Mmodelview とすると、クリップ座標系への変換は以下の式で表されます。

$$\begin{pmatrix} o1.x & o1.y & o1.z & 0 \\ o2.x & o2.y & o2.z & 0 \\ o3.x & o3.y & o3.z & 0 \\ o4.x & o4.y & o4.z & 0 \end{pmatrix} = Mproj \times Mmodelview \times \begin{pmatrix} Rx & 0 & 0 & 0 \\ 0 & Ry & 0 & 0 \\ 0 & 0 & Rz & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

シェーダアセンブラで記述すると以下のようになります。aBoundingBox がアプリケーションから入力された半径サイズの xyz 成分をベクトルにしたもの、vBoundingBox1 ～ vBoundingBox4 がパーティクルシステムシェーダに出力される行列です。このコード例ではバウンディングボックスの半径サイズをアトリビュートで入力していますが、パーティクルシステムシェーダが必要とするデータは 4 頂点分だけですので、頂点座標も含めて、すべてをユニフォームで設定する実装も可能です。

コード 9-11. バウンディングボックスの半径サイズのクリップ座標変換(シェーダアセンブラ)

```
mov     TEMP_BOX[0],    CONST_0
mov     TEMP_BOX[1],    CONST_0
mov     TEMP_BOX[2],    CONST_0
mov     TEMP_BOX[3],    CONST_0
mov     TEMP_BOX[0].x,  aBoundingBox.x
mov     TEMP_BOX[1].y,  aBoundingBox.y
mov     TEMP_BOX[2].z,  aBoundingBox.z
m4x4    TEMP_MAT,       MATRIX_Project, MATRIX_ModelView
m4x4    TEMP_MAT,       TEMP_MAT,       TEMP_BOX
mov     vBoundingBox1,  TEMP_MAT[0]
mov     vBoundingBox2,  TEMP_MAT[1]
mov     vBoundingBox3,  TEMP_MAT[2]
```

```
mov     vBoundingBox4,  TEMP_MAT[3]
```

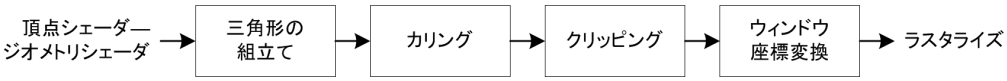
9.6.4. 頂点データの入力

glDrawArrays() での呼び出しには対応していません。パーティクルシステムシェーダを使用する場合は、glDrawElements() の *mode* に GL_GEOMETRY_PRIMITIVE_DMP を渡して呼び出してください。

10. ラスタライズ

ポイントシェーダやラインシェーダなどのジオメトリシェーダで生成したプリミティブも、最終的にはすべて三角形プリミティブに変換されて生成されます(トライアングル生成、トライアングルセットアップ)。そして、生成された三角形に対してカリング処理やクリッピング処理、ウィンドウ座標変換処理が行われ、ラスタライズでフラグメントの集合に変換されます。OpenGL ES 2.0と異なり、シザーテストはラスタライズの段階で行われます。これ以降の処理は、生成されたフラグメントが対象となります。

図 10-1. 頂点シェーダ—、ジオメトリシェーダからラスタライズまでの処理の流れ



ラスタライズ以降の処理は固定パイプラインで実装されているため、フラグメントに割り当てることができる頂点属性は固定されています。主な頂点属性としては以下のものがあります。

- ウィンドウ座標
- デプス値
- テクスチャ座標およびその偏微分値
- クォータニオン
- 視点ベクトル
- 頂点カラー(絶対値を求めてからラスタライズされます)

10.1. カリング

生成された三角形(ポリゴン)は頂点の指定順によって、視点に対して表面を向けているか、裏面を向けているかが判定されます。カリングはこの表裏判定をもとに、ポリゴンをラスタライズするかどうかを判断する機能です。

10.1.1. 表裏判定

ポリゴンの表裏はウィンドウ座標系での三角形の頂点がどのような順序で指定されているかで判定されます。判定方法としては、時計回りに指定されているときに表とするか、反時計回りに指定されているときに表とするかの 2 種類があり、`glFrontFace()` で指定することができます。

コード 10-1. glFrontFace() の定義

```
void glFrontFace(GLenum mode);
```

mode には、時計回り(`GL_CW`)または反時計回り(`GL_CCW`)を指定します。デフォルトは反時計回り(`GL_CCW`)です。

10.1.2. 使用方法

カリングの使用方法は OpenGL と同じです。

カリングの有効・無効

カリングの有効・無効の制御は、*cap* に `GL_CULL_FACE` を渡して `glEnable()` または `glDisable()` を呼び出すことで行います。現在の設定を取得するには、*cap* に `GL_CULL_FACE` を渡して `glIsEnabled()` を呼び出してください。デフォルトではカリングは無効になっています。

カリング面の指定

ラスタライズしない面(カリング面)の指定は glCullFace() で行います。

コード 10-2. glCullFace() の定義

```
void glCullFace(GLenum mode);
```

mode に指定するカリング面は、以下の値から選択することができます。

表 10-1. カリング面の指定

設定値	カリング面
GL_FRONT	表
GL_BACK(デフォルト)	裏
GL_FRONT_AND_BACK	両面

10.2. クリッピング

クリッピングは、指定されたクリップ平面で定義される半空間とビューボリュームが交差する領域(クリップボリューム)でプリミティブをクリップする(切り抜く)機能です。3DS のクリッピングは OpenGL ES 1.1 のクリッピング相当の機能を持っていますが、その制御はすべて予約ユニフォームで行われます。また、クリッピング対象の三角形に対して新たな頂点の生成と三角形の生成を行い、複数の三角形に分解するように実装されています。

GPU の頂点処理で行われる座標変換では 24 ビットの浮動小数点数が使用されるため、ニアとファーの比が大きい場合にファークリップ平面でクリッピング処理が正しく行われないことがあります。なるべく、ニアとファーの比を大きくしないようにクリップボリュームを設定するか、ファークリップ平面付近にポリゴンを配置しないようにしてください。

10.2.1. 予約ユニフォーム

クリッピングの予約ユニフォームには、以下のものが存在します。

クリッピングの有効・無効

クリッピング機能を有効にするには、予約ユニフォーム(dmp_FragOperation.enableClippingPlane)に glUniform1i() で GL_TRUE を設定してください。デフォルトでは無効(GL_FALSE)に設定されています。

クリップ平面

クリップ平面の指定は 4 つの係数を予約ユニフォーム(dmp_FragOperation.clippingPlane)に glUniform4f() で設定することで行われます。4 つの係数を p1、p2、p3、p4 とすると、クリップボリュームは以下の式を満たす点の集合で表されます。

$$\begin{pmatrix} p1 & p2 & p3 & p4 \end{pmatrix} \begin{pmatrix} Xc \\ Yc \\ Zc \\ Wc \end{pmatrix} \geq 0$$

これらの係数はクリップ座標系で定義されていなければなりませんので、OpenGL ES 標準で使用するクリップ平面にモデルビュー変換と透視投影変換を行った値を指定してください。OpenGL ES の仕様と異なり、3DS では Z 座標が 0 ～ -Wc の

範囲にクリップされるため、透視投影変換に OpenGL ES 互換の行列をそのまま使用することはできません。デフォルトではすべての係数は 0.0 に設定されています。

> **補足:** 射影変換に OpenGL ES 互換の行列を使用する際の注意事項については、関連する内容が「8.4. クリップ座標系の注意事項」にありますので、そちらを参照してください。

表 10-2. クリッピングで使用する予約ユニフォーム

予約ユニフォーム	種別	設定する値
dmp_FragOperation.enableClippingPlane	bool	クリッピングの有効/無効を指定する。 GL_TRUE GL_FALSE(デフォルト)
dmp_FragOperation.clippingPlane	vec4	クリッピング平面の 4 つの係数を指定する。 (デフォルト)(0.0, 0.0, 0.0, 0.0)

10.3. ウィンドウ座標系への変換

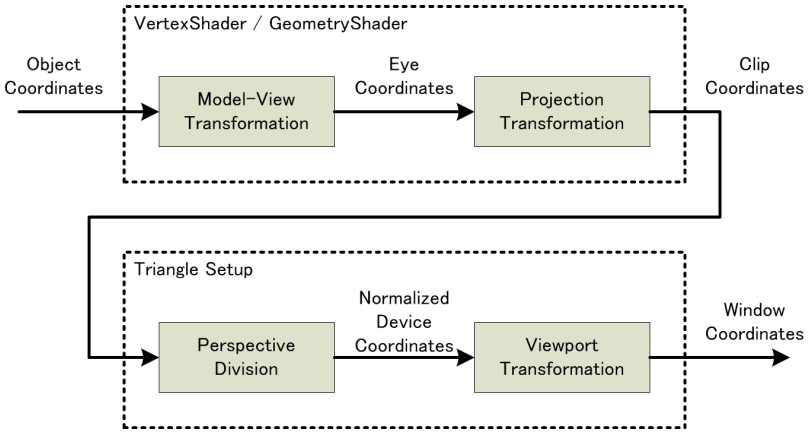
オブジェクト座標系で表されたポリゴンモデルの頂点座標がウィンドウ座標系へと変換されるまでに、頂点座標は以下の 4 つの処理を経ています。

- モデルビュー変換
  オブジェクト座標系から視点座標系へと変換します。
- 射影変換
  視点座標系からクリップ座標系へと変換します。変換結果が $0 \geqq Zc \geqq -Wc$ であることに注意してください。
- 透視法除算
  w 値を使って、クリップ座標系から正規化されたデバイス座標系へと変換します。
- ビューポート変換
  ビューポートの設定に従って、正規化されたデバイス座標系からウィンドウ座標系へと変換します。

頂点シェーダ(ジオメトリシェーダ)から出力される頂点属性の座標系はクリップ座標系でなければなりません。そのため、一般的に頂点シェーダ内でモデルビュー変換と射影変換を行うことになります。トライアングル生成で生成された三角形は、透視法除算とビューポート変換によって表示領域での位置が確定します。これらの三角形はラスタライズでフラグメントに変換され、フラグメントライティングなどのプロセスで使用されることになります。

この座標変換の流れを図にすると、以下のようになります。

図 10-2. オブジェクト座標系の頂点座標がウィンドウ座標系に変換されるまでの流れ



10.3.1. ビューポートの設定

正規化されたデバイス座標系からウィンドウ座標系への変換は以下の式で行われます。

$$\begin{pmatrix} Xw \\ Yw \\ Zw \end{pmatrix} = \begin{pmatrix} (p_x / 2)Xd + o_x \\ (p_y / 2)Yd + o_y \\ n - (f - n) \times Zd \end{pmatrix}$$

Xw、Yw、Zw :ウィンドウ座標系での座標
Xd、Yd、Zd :正規化されたデバイス座標系での座標
p_x、p_y :ビューポートの幅と高さ
o_x、o_y :ビューポートの中心点
n、f :クリップ空間のニア平面、ファー平面のデプス値

上式の n と f に適用される値は `glDepthRangef()` で設定することができます。

コード 10-3. glDepthRangef() の定義

```
void glDepthRangef(GLclampf zNear, GLclampf zFar);
```

zNear、*zFar* に指定した値は、どちらも 0.0 ～ 1.0 の範囲にクランプされて設定されます。デフォルトでは、*zNear* には 0.0、*zFar* には 1.0 が指定されています。

p_x、p_y、o_x、o_y はいずれもビューポートの設定値から計算することができます。ビューポートの設定は `glViewport()` で行います。

コード 10-4. glViewport() の定義

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

x と *y* にはビューポートの始点(左下)座標を指定します。負の値を指定すると `GL_INVALID_VALUE` エラーが生成されます。また、4 の倍数でない値を指定した場合は処理性能が低下(偶数ならば約 1/2、奇数ならば約 1/3)します。その場合は、4 の倍数になるようにビューポートを拡張し、透視投影変換行列を拡張されたビューポートで正しく描画されるようにする調整と、余分な領域を描画しないようにシザーテストを適用する処置で回避してください。

width と *height* にはビューポートの幅と高さを指定します。負の値を指定すると `GL_INVALID_VALUE` エラーが生成されます。幅と高さの最大値はどちらも 1024 です。

p_x には *width*、p_y には *height*、o_x には (*x* + *width*) / 2、o_y には (*y* + *height*) / 2 がそれぞれ適用されます。

10.3.1.1. ビューポートが 1023×1016 より大きな場合に正しく描画されない

ハードウェアの不具合により、glViewport() でビューポートを設定する際に以下の条件を満たすと正しく描画が行われません。

- *width* が 1023 より大きい設定の場合、ウィンドウ左端の x 座標を 0 としたときに(ウィンドウ座標の x 座標 - x)が 1023 以上となるピクセルを含むポリゴンが描画されると、そのポリゴン全体が描画されません。
- *height* が 1016 より大きい設定の場合、ウィンドウ下端の y 座標を 0 としたときに(ウィンドウ座標の y 座標 - y)が 1016 以上となるピクセルを含むポリゴンが描画されると、GPU がハングアップします。

この問題を回避するには、ビューポートの *width* に 1023 より大きい値を設定しない、かつ *height* に 1016 より大きい値を設定しないことにより、問題の起こる座標のピクセルが描画されないようにします。

サイズが 1024×1024 のレンダートゥーテクスチャを行う場合は 1023×1016 の領域にだけ描画し、テクスチャとして使用するときに 1023×1016 をテクスチャの有効領域とするようにテクスチャ座標を調整する必要があります。

1024×1024 の全領域に描画したい場合は、ビューポートの *width* が 1023、*height* が 1016 を超えないようにしつつ、ビューポートのオフセットを変更して描画領域をずらしながら、分割して描画します。例えば、glViewport(0, 0, 512, 512)、glViewport(512, 0, 512, 512)、glViewport(0, 512, 512, 512)、glViewport(512, 512, 512, 512) の 4 つに分割して描画すれば問題は起こりません。

また、シザリングで問題の領域にピクセルが描かれないようにする方法では、不具合を回避することはできません。

10.3.2. ポリゴンオフセット

ポリゴンオフセットは、ポリゴンをラスタライズしてフラグメントに変換する際のデプス値にオフセットを加算することで、同一平面に重なって存在するポリゴンのように、デプス値の分解能不足によってフラグメントの前後関係が定まらない状態を解決する機能です。ポリゴンオフセットは、ウィンドウ座標変換処理の際に行われます。

ポリゴンオフセットの有効・無効

ポリゴンオフセットの有効・無効の制御は、*cap* に GL_POLYGON_OFFSET_FILL を渡して glEnable() または glDisable() を呼び出すことで行います。現在の設定を取得するには、*cap* に GL_POLYGON_OFFSET_FILL を渡して glIsEnabled() を呼び出してください。デフォルトではポリゴンオフセットは無効になっています。

オフセット値の指定

ポリゴンオフセットが有効であるときに、デプス値に与えられるオフセット値は glPolygonOffset() で指定することができます。

コード 10-5. glPolygonOffset() の定義

```
void glPolygonOffset(GLfloat factor, GLfloat units);
```

OpenGL では *factor* と *units* によってオフセット値が決定されますが、3DS では *units* のみでオフセット値が決定されます。*factor* の値は設定されますが、オフセット値には考慮されません。

オフセット値には、ウィンドウ座標系でのデプス値に違いが現れる最小値(固定値)と *units* を乗算した値が適用されます。頂点処理後の頂点座標の z 値が 24 ビット浮動小数点数で実装されていますので、ポリゴンの z 値が 1.0 に近い場合は *units* の設定が 128 の倍数単位でなければ効果が得られません。確実に効果を得るには、*units* に 128 の倍数を設定してください。

デプスバッファに書き込まれる値は、オフセット値が加算されたあとのデプス値です。

10.3.3. w バッファ

w バッファとは、透視投影を行わずにウィンドウ座標系のデプス値を算出する機能です。w バッファは以下の予約ユニフォームで制御することができます。w バッファを有効にした場合、glDepthRangef() の設定は無効となります。

デプス値に対するスケール

w バッファ有効時のデプス値は以下の式で計算されます。

$$Zw = -scale_w \times Zc$$

Zw がウィンドウ座標系でのデプス値、Zc がクリップ座標系での z 値、$scale_w$ がスケール値です。このスケール値は予約ユニフォーム(dmp_FragOperation.wScale)に設定された浮動小数点数で、0.0 以外の値が設定されたときに w バッファが有効となります。スケール値は Zw が 0.0 ～ 1.0 の範囲に収まるように設定してください。

表 10-3. w バッファで使用する予約ユニフォーム

予約ユニフォーム	種別	設定値
dmp_FragOperation.wScale	float	デプス値に対するスケールを指定する。 (デフォルト)0.0

ポリゴンオフセットが有効になっている場合

w バッファとポリゴンオフセットの機能がともに有効になっている場合、ポリゴンオフセットのオフセット値には glPolygonOffset() で指定した *units* に Wc(クリップ座標の w 値)を乗算したものが適用されます。

10.4. シザーテスト

シザーテストは、ウィンドウ座標系で指定された範囲の外にあるフラグメントを棄却して、後のプロセスで処理されるフラグメントを削減する機能です。

10.4.1. 使用方法

処理が行われるパイプラインでの位置は異なりますが、仕様自体は OpenGL のシザーテストと変わりはありません。

シザーテストの有効・無効

シザーテストの有効・無効の制御は、*cap* に GL_SCISSOR_TEST を渡して glEnable() または glDisable() を呼び出すことで行います。現在の設定を取得するには、*cap* に GL_SCISSOR_TEST を渡して glIsEnabled() を呼び出してください。デフォルトではシザーテストは無効になっています。無効に設定されている場合、フラグメントの棄却は行われません。

シザーボックスの指定

フラグメントを通過させる範囲(シザーボックス)の指定は glScissor() で行います。

コード 10-6. glScissor() の定義

```
void glScissor(GLint x, GLint y, GLsizei width, GLsizei height);
```

x と *y* はウィンドウ座標系でのシザーボックスの始点(左下)座標です。*width* と *height* はシザーボックスの幅と高さです。*width* または *height* に 0 を指定した場合は GL_INVALID_VALUE のエラーが生成されます。デフォルトでは値が設定されていませんので、シザーテストを有効にしたときはシザーボックスを必ず指定してください。

始点座標のフラグメントはシザーボックスに含まれますが、x 座標が (x + $width$) または y 座標が (y + $height$) のフラグメントはシザーボックスに含まれません。

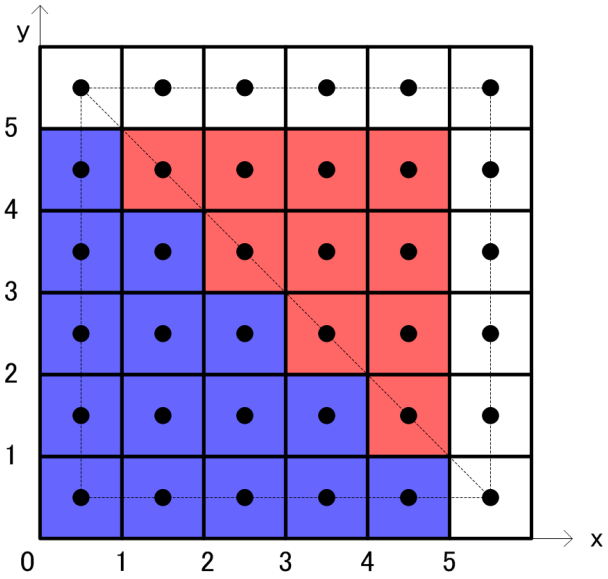
10.5. ラスタライズルール

PICA グラフィックスコアによるポリゴンのラスタライズ(フラグメントの生成)は、以下のルールに従って行われます。

- ピクセルの中心座標 (x + 0.5, y + 0.5)(x および y は整数)がポリゴンの内部にある。
- ポリゴンの辺(エッジ)がピクセルの中心座標を通過している場合は、Bottom-Left の規則に従い、通過している辺が下辺または左辺ならばフラグメントが生成され、上辺または右辺ならばフラグメントが生成されない。
  ※ X 軸の負の方向を左、Y 軸の負の方向を下とする。

下図は 2 つのポリゴンのラスタライズ結果をもとに、Bottom-Left の規則がどのように適用されるのかを示しています。2 つのポリゴンは (5.5, 0.5), (5.5, 5.5), (0.5, 5.5) の 3 頂点によるポリゴンと、(5.5, 0.5), (0.5, 0.5), (0.5, 5.5) の 3 頂点によるポリゴンとし、前者に対して生成されるフラグメントを赤、後者に対して生成されるフラグメントを青で示しています。

図 10-3. ラスタライズルール(Bottom-Left の規則)の例



ラスタライズルールは、以下のように適用されています。

2 つのポリゴンの境界部分では、赤いポリゴンの左辺がピクセルの中心を通過しているため、赤で塗られます。
x 座標 0.5 が中心であるピクセルでは、青いポリゴンの左辺がピクセルの中心を通過しているため、青で塗られます。
x 座標 5.5 が中心であるピクセルでは、赤いポリゴンの右辺がピクセルの中心を通過しているため、塗られません。
y 座標 0.5 が中心であるピクセルでは、青いポリゴンの下辺がピクセルの中心を通過しているため、青で塗られます。
y 座標 5.5 が中心であるピクセルでは、赤いポリゴンの上辺がピクセルの中心を通過しているため、塗られません。

11. テクスチャ処理

テクスチャ処理では OpenGL ES 2.0 と同等の動作を行うことができますが、いくつか 3DS 固有の制約があります。

11.1. テクスチャユニット

3DS は 4 つのテクスチャユニット(TEXTURE0 ～ 3)を搭載していますが、テクスチャユニットによって扱うことのできるテクスチャの種類が異なります。また、ラスタライズを行うラスタライザからテクスチャユニットに、独立して出力されるテクスチャ座標は 3 組までです。**4 つのテクスチャユニットすべてにテクスチャ座標を出力する場合、TEXTURE2 または TEXTURE3 がほかのテクスチャユニットと同じテクスチャ座標を共有することになります**。

表 11-1. テクスチャユニットが扱うことのできるテクスチャ

テクスチャユニット	2次元テクスチャ	キューブマップテクスチャ	シャドウテクスチャ	プロジェクションテクスチャ	プロシージャルテクスチャ
TEXTURE0	✓	✓	✓	✓	
TEXTURE1	✓				
TEXTURE2	✓				
TEXTURE3					✓

1 次元、3 次元テクスチャには対応していません。キューブマップ、シャドウ、プロジェクションのように、w 成分が必要となるテクスチャの処理は TEXTURE0 でのみ行うことができます。また、TEXTURE3 はプロシージャルテクスチャ専用のユニットです。搭載されているテクスチャユニットの数を示す GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS は 4 で定義されています。

11.1.1. テクスチャ座標の入力

テクスチャユニット 0 のみ w 成分を入力することができ、そのほかのテクスチャユニットには u と v の 2 成分のみ入力が可能です。プロジェクションテクスチャは、単に 2 次元テクスチャに w 成分を有効にしたものですが、座標(u, v)が w で除算されて生成されるため、頂点シェーダで出力する座標値には注意が必要となります。

頂点シェーダからテクスチャ座標を送るには属性名 texture0、texture0w、texture1、texture2 で出力レジスタをマッピングします。texture0w は、キューブマップ、シャドウ、プロジェクションテクスチャのように、w 成分を必要とする場合には必ず出力しなければなりません。出力していない場合のテクスチャ座標 0 の出力は不定です。w 成分を必要としないテクスチャに対する texture0w の出力は無視されます。

表 11-2. 頂点シェーダで指定する属性名とテクスチャ座標の対応

属性名	頂点シェーダから送られる属性
texture0	テクスチャ座標 0 の u, v 成分
texture0w	テクスチャ座標 0 の w 成分
texture1	テクスチャ座標 1 の u, v 成分
texture2	テクスチャ座標 2 の u, v 成分

11.1.1.1. テクスチャ座標の精度について

テクスチャユニット内では、テクスチャ座標値は整数部と小数部を合わせて 16 ビットの値で表現され、整数部の絶対値が大きくなると小数部のビット数が小さくなります。

テクスチャのサンプリング精度は小数部のビット精度に依存しています。小数部がテクスチャの幅または高さのテクセル数を表現するのに十分なビット数になる場合は、最適なテクスチャのサンプリングを行うことができます。バイリニアフィルタリングを行う場合は、さらに 6 ビットを小数部に確保するようにしてください。

11.1.2. 使用方法

OpenGL ではテクスチャユニットの有効・無効の切り替えを glEnable() と glDisable() の引数に TEXTURE_2D などを指定することで行っていましたが、3DS では予約ユニフォーム dmp_Texture[*i*].samplerType(*i* はテクスチャユニットの番号)に対する設定によって行います。GL_TEXTURE_2D を引数に、glEnable() または glDisable() を呼び出した場合の動作は不定です。

テクスチャユニットを無効にするには、どのテクスチャユニットでも GL_FALSE を指定して glUniform1i() を呼び出し、予約ユニフォームに値を設定することで行われます。有効にする場合は、テクスチャユニットによって予約ユニフォームに設定する値が異なります。

表 11-3. 予約ユニフォーム dmp_Texture[i].samplerType に対する設定値

予約ユニフォーム	設定する値	扱うテクスチャ
dmp_Texture[0].samplerType	GL_FALSE	無効(デフォルト)
	GL_TEXTURE_2D	2 次元テクスチャ
	GL_TEXTURE_CUBE_MAP	キューブマップテクスチャ
	GL_TEXTURE_SHADOW_2D_DMP	シャドウテクスチャ
	GL_TEXTURE_SHADOW_CUBE_DMP	キューブマップシャドウテクスチャ
	GL_TEXTURE_PROJECTION_DMP	プロジェクションテクスチャ
dmp_Texture[1].samplerType	GL_FALSE	無効(デフォルト)
	GL_TEXTURE_2D	2 次元テクスチャ
dmp_Texture[2].samplerType	GL_FALSE	無効(デフォルト)
	GL_TEXTURE_2D	2 次元テクスチャ
dmp_Texture[3].samplerType	GL_FALSE	無効(デフォルト)
	GL_TEXTURE_PROCEDURAL_DMP	プロシージャルテクスチャ

テクスチャユニットが入力に使用するテクスチャ座標は、テクスチャユニット 0 がテクスチャ座標 0 固定、テクスチャユニット 1 がテクスチャ座標 1 固定、テクスチャユニット 2 と 3 については予約ユニフォーム dmp_Texture[*i*].texcoord(*i* はテクスチャユニットの番号で 2 または 3)で指定しなければなりません。テクスチャユニットによって予約ユニフォームに設定することができる値が異なります。4 つのテクスチャユニットすべてを使用する場合、テクスチャユニット 2 またはテクスチャユニット 3 が入力に使用するテクスチャ座標をほかのテクスチャユニットと共有しなければなりません。

テクスチャコンバイナから無効化されているテクスチャユニットを参照した場合、テクスチャコンバイナに入力されるカラー値は不定になります。

表 11-4. 予約ユニフォーム dmp_Texture[i].texcoord に対する設定値

予約ユニフォーム	設定する値	入力に使用するテクスチャ座標
dmp_Texture[2].texcoord	GL_TEXTURE1	テクスチャ座標 1
	GL_TEXTURE2	テクスチャ座標 2(デフォルト)
dmp_Texture[3].texcoord	GL_TEXTURE0	テクスチャ座標 0(デフォルト)
	GL_TEXTURE1	テクスチャ座標 1
	GL_TEXTURE2	テクスチャ座標 2

予約ユニフォームによる設定を除き、テクスチャユニットに対する設定は glActiveTexture() で指定したテクスチャユニットに対して行われます。

コード 11-1. glActiveTexture() の定義

```
void glActiveTexture(GLenum texture);
```

texture には GL_TEXTURE0、GL_TEXTURE1、GL_TEXTURE2 を指定することができます。テクスチャユニット 3 を示す GL_TEXTURE3 を指定した場合や上記以外の値を指定した場合は GL_INVALID_ENUM のエラーが生成されます。

プロシージャルテクスチャの設定はすべて予約ユニフォームで行います。

11.1.3. 使用するテクスチャの指定

テクスチャユニットで使用するテクスチャは glActiveTexture() でテクスチャユニットを指定した後の、glBindTexture() によるテクスチャオブジェクトの指定で行われます。

テクスチャユニットごとに異なるテクスチャを使用する場合は、以下のコード例のように glActiveTexture() と glBindTexture() を呼び出してください。

コード 11-2. テクスチャユニットごとにテクスチャを指定する例

```
// Texture Unit0
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, imageTexID);
// Texture Unit1
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, bumpTexID);
```

11.1.4. テクスチャパラメータ

テクスチャのラッピングモードやフィルタなど、テクスチャにパラメータを付加するには glTexParameter*() を使用します。

コード 11-3. glTexParameter*() の定義

```
void glTexParameterf(GLenum target, GLenum pname, GLfloat param);
void glTexParameterfv(GLenum target, GLenum pname, const GLfloat* params);
void glTexParameteri(GLenum target, GLenum pname, GLint param);
void glTexParameteriv(GLenum target, GLenum pname, const GLint* params);
```

glTexParameteri() はパラメータに整数を、glTexParameterf() はパラメータに浮動小数点数を渡して付加する

関数です。関数名の末尾に "v" の付いたものはベクトル(配列)を渡さなければならないパラメータの付加に使用します。

*target*に指定する値は glTexImage2D() の *target* と同じです(表 7-2)。*pname* は付加するパラメータの名前、*param* はパラメータの値です。3DS で独自に追加されているパラメータが存在します。

3DS では、1 つのフラグメントに対して最大 8 テクセルを取得してフィルタリングを行っています。

表 11-5. pname と付加されるパラメータの対応

pname	int / float / vector	付加されるパラメータ
GL_TEXTURE_WRAP_S	int	S 方向のラッピングモード(表 11-6)
GL_TEXTURE_WRAP_T	int	T 方向のラッピングモード(表 11-6)
GL_TEXTURE_MIN_FILTER	int	縮小時のフィルタ(表 11-7)
GL_TEXTURE_MAG_FILTER	int	拡大時のフィルタ(表 11-8)
GL_TEXTURE_BORDER_COLOR	vec4(float)	ラッピングモードで GL_CLAMP_TO_BORDER が指定されている場合に使用されるボーダーカラー(各成分とも 0.0 ～ 1.0)
GL_TEXTURE_LOD_BIAS	float	LOD バイアス値(-16.0 ～ 16.0。デフォルト 0.0)
GL_TEXTURE_MIN_LOD	int	LOD の最小レベル(デフォルト -1000)
GL_GENERATE_MIPMAP	int(bool)	ミップマップテクスチャの自動生成(デフォルト GL_FALSE)。詳細については、「11.1.4.1. ミップマップテクスチャの自動生成」を参照してください。

S、T 方向のラッピングモードは以下の値で指定します。

表 11-6. S、T 方向のラッピングモードの指定

param	説明
GL_REPEAT	繰り返し(デフォルト)
GL_MIRRORED_REPEAT	反転して繰り返し
GL_CLAMP_TO_EDGE	テクスチャ座標 0.0 ～ 1.0 以外はテクスチャイメージの端のカラー
GL_CLAMP_TO_BORDER	テクスチャ座標 0.0 ～ 1.0 以外はボーダーカラー

テクスチャイメージが縮小されてレンダリングされる場合のフィルタは以下の値で指定します。

表 11-7. 縮小時のフィルタの指定

param	説明
GL_NEAREST	ニアレストに選定されたテクセルのカラーを使用する(デフォルト)
GL_LINEAR	バイリニアサンプリング(4 サンプルの平均)でカラーを決定する
GL_NEAREST_MIPMAP_NEAREST	ミップマップテクスチャの選定をニアレストで行い、ニアレストに選定されたテクセルのカラーを使用する
GL_NEAREST_MIPMAP_LINEAR	2 つのレベルのミップマップテクスチャを選定し、それぞれのレベルでニアレストに選定されたカラーから補間する

GL_LINEAR_MIPMAP_NEAREST	ミップマップテクスチャの選定をニアレストで行い、バイリニアサンプリングでカラーを決定する
GL_LINEAR_MIPMAP_LINEAR	2 つのレベルのミップマップテクスチャを選定し、それぞれのレベルでバイリニアサンプリングされたカラーから補間する

テクスチャイメージが拡大されてレンダリングされる場合のフィルタは以下の値で指定します。

表 11-8. 拡大時のフィルタの指定

param	説明
GL_NEAREST	ニアレストに選定されたテクセルのカラーを使用する(デフォルト)
GL_LINEAR	バイリニアサンプリング(4 サンプルの平均)でカラーを決定する

11.1.4.1. ミップマップテクスチャの自動生成

テクスチャパラメータ GL_GENERATE_MIPMAP に GL_TRUE を設定すると、glTexImage2D()、glCopyTexImage2D()、glCopyTexSubImage2D() の呼び出しで *level* に -2 以下の値を渡したときに、最も低いレベルのテクスチャ以外のミップマップテクスチャが自動生成されます。ただし、自動生成されるミップマップテクスチャの幅と高さは、テクスチャのフォーマット(*format* と *type* の組み合わせ)によって、その最小値が以下のように制限されています。

表 11-9. 自動生成されるミップマップテクスチャの幅と高さの最小値

フォーマット	format	type	幅、高さの最小値
RGBA4	GL_RGBA GL_RGBA_NATIVE_DMP	GL_UNSIGNED_SHORT_4_4_4_4	64
RGBA5551	GL_RGBA GL_RGBA_NATIVE_DMP	GL_UNSIGNED_SHORT_5_5_5_1	64
RGBA8	GL_RGBA GL_RGBA_NATIVE_DMP	GL_UNSIGNED_BYTE	32
RGB565	GL_RGB GL_RGB_NATIVE_DMP	GL_UNSIGNED_SHORT_5_6_5	64
RGB8	GL_RGB GL_RGB_NATIVE_DMP	GL_UNSIGNED_BYTE	32

テクスチャのフォーマットによる最小値の違いにより、同じ幅と高さのテクスチャでも *level* に指定することのできる値の範囲が異なります。最小値よりも小さなサイズのミップマップテクスチャを自動生成させるような指定をした場合は GL_INVALID_OPERATION のエラーが生成されます。例えば、128x128 テクセルのテクスチャに対して、フォーマットが RGB8 のときは *level* に -2 または -3 を指定することができますが、フォーマットが RGB565 のときは *level* に -2 のみ指定することができます。

自動生成を行う設定をしていた場合は、自動生成されたミップマップテクスチャが優先されます。ミップマップテクスチャのデータを含むテクスチャイメージをロードすると、そのミップマップテクスチャのデータは無効となります。

自動生成を行う設定をしていた場合、glCopyTexImage2D() と glCopyTexSubImage2D() の *level* には、対象のテクスチャをロードしたときの glTexImage2D() で *level* に渡した値と同じ値を渡さなければなりません。異なる場合は GL_INVALID_OPERATION のエラーが生成されます。ただし、*level* に 0 以外を渡したとしても、コピー先は最も低いレベルのテクスチャであり、ミップマップテクスチャを書き換えることにはなりません。

自動生成を行わない設定をしていた場合は、glCopyTexImage2D() と glCopyTexSubImage2D() の *level* には、対象のテクスチャをロードしたときの glTexImage2D() で *level* に渡した値に関わらず、0 を指定しなければなりません。0 以外を指定した場合は GL_INVALID_OPERATION のエラーが生成されます。

11.1.4.2. フィルタに GL_NEAREST を指定した場合の注意事項

テクスチャパラメータの GL_TEXTURE_MIN_FILTER および GL_TEXTURE_MAG_FILTER に対して GL_NEAREST を指定した場合、垂直または水平な直線(色の境界が明確な直線を成す場合を含む)のある画像をテクスチャとして使用し、その直線が画面の走査線に垂直または水平になるようにポリゴンを描画すると、本来直線に見えるはずの模様がデコボコな線として描画されることがあります。

この現象は、ポリゴン内のテクスチャ座標を補間する際の計算精度が起因となっています。走査線に垂直または水平な、ある一列に並んだフラグメント列を考えたとき、各フラグメントがテクスチャ画像上の垂直または水平な、ある一列に並んだテクセルをサンプリングすると、テクスチャの画像上の直線はそのままの直線としてポリゴン上に描画されます。しかし、それらの各フラグメントが、ある一列に並んだテクセルと、その隣の列に並んだテクセルの境界付近をサンプリングする場合、テクスチャ座標の誤差によって、どちらの列のテクセルをサンプルするかがフラグメントごとに異なり、テクスチャの画像上の直線がデコボコな線としてポリゴン上に描画されてしまうことがその原因です。そのため、両端のテクスチャ座標が 0 ～ 1 であるような矩形ポリゴンをテクスチャと同じサイズで描画するのと同じように、各フラグメントがテクセルの中央をサンプルするようにテクスチャ座標や描画面積を調整することで回避することができます。

11.1.4.3. 縮小時のフィルタに GL_XXX_MIPMAP_LINEAR を指定した場合の注意事項

テクスチャパラメータの GL_TEXTURE_MIN_FILTER に対して GL_XXX_MIPMAP_LINEAR を指定した場合はトライリニアフィルタが有効となり、2 つのレベルのミップマップテクスチャから取得したカラーを用いて補間処理を行ったカラーで描画されます。しかし、この補間処理には計算精度による誤差が発生する可能性があり、たとえ同じ成分値の 2 つのカラーを処理した場合でも、誤差によって異なるカラーが描画される場合があります。

補間処理が行われるのは LOD 計算の結果、2 つのレベルのミップマップテクスチャからカラーを取得しなければならない場合ですので、あるレベルのミップマップテクスチャだけからカラーを取得する場合は補間処理が行われません。そのため、補間処理が行われた部分が、補間処理が行われなかった部分に比べてわずかに暗くなります。その結果、色味の差異がミップマップレベルの境界に沿ってエッジのように描画されてしまいます。

この現象の緩和策として、テクセルカラーの固定値である成分を利用する方法があります。例えば、GL_RGB フォーマットのテクスチャはアルファ成分が 1.0 の固定値ですが、この固定値の成分もトライリニアフィルタによる補間処理が行われたテクセルではわずかに減少してしまいます。一方、補間処理が行われなかったテクセルでは、そのまま 1.0 で描画されます。カラーの補正は、補間処理によって減少したアルファ成分の差分値をテクスチャカラーと乗算した結果をテクスチャカラーに加算することで行います。

この補正処理はテクスチャコンバイナで行うことができます。以下に設定例を示します。

コード 11-4. トライリニアフィルタ時の補正処理

```
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[0].combineRgb"),
        GL_MULT_ADD_DMP);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[0].operandRgb"),
        GL_SRC_COLOR, GL_ONE_MINUS_SRC_ALPHA, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[0].srcRgb"),
        GL_TEXTURE0, GL_TEXTURE0, GL_TEXTURE0);
```

この設定により、補間処理が行われなかった部分は乗算結果が 0.0 で出力されるためテクスチャカラーそのものが出力されることになり、補間処理が行われた部分はアルファ成分の差分値とテクスチャカラーの乗算結果をテクスチャカラーに加算して補正した結果が出力されることになります。この設定では入力ソースのすべてにテクスチャカラーを指定しますので、1 段

目のテクスチャコンバイナ(コンバイナ 0)でなければ設定することができません。

GL_RGBA フォーマットのように、成分中に固定値のものがない場合は、固定値の成分を持つ別のテクスチャを用意し、マルチテクスチャとすることで補正します。補正のために用意するテクスチャは、元のテクスチャと同じサイズ、同じミップマップレベル数にし、同じ uv の値が入力されなければなりません。データサイズやキャッシュ効率を考慮し、ETC1 圧縮テクスチャで用意することを推奨します。

11.1.4.4. テクスチャレベル(ミップマップ)パラメータの取得

以下の関数で、現在アクティブになっているテクスチャユニットにバインドされているテクスチャの、ミップマップのレベルごとのパラメータを取得することができます。ただし、プロシージャルテクスチャ用のテクスチャユニット(GL_TEXTURE3)は glActiveTexture() で指定できないため、プロシージャルテクスチャの情報は取得できません。

コード 11-5. テクスチャレベルパラメータの取得

```
void glGetTexLevelParameterfv(GLenum target, GLint level, GLenum pname,
                              GLfloat *params);
void glGetTexLevelParameteriv(GLenum target, GLint level, GLenum pname,
                              GLint *params);
```

取得した値を格納する変数の型が異なるだけで、それぞれの関数で取得する値に違いはありません。

target にはテクスチャの種類を指定します。指定可能な値は以下のとおりです。

表 11-10. テクスチャレベルパラメータを取得するテクスチャの種類の指定

target に指定する値	テクスチャの種類
GL_TEXTURE_2D	2 次元テクスチャ(シャドウテクスチャ、ガステクスチャを含む)
GL_TEXTURE_CUBE_MAP_POSITIVE_X	キューブマップテクスチャ(+X 方向の面)
GL_TEXTURE_CUBE_MAP_NEGATIVE_X	キューブマップテクスチャ(-X 方向の面)
GL_TEXTURE_CUBE_MAP_POSITIVE_Y	キューブマップテクスチャ(+Y 方向の面)
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y	キューブマップテクスチャ(-Y 方向の面)
GL_TEXTURE_CUBE_MAP_POSITIVE_Z	キューブマップテクスチャ(+Z 方向の面)
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z	キューブマップテクスチャ(-Z 方向の面)

level にはパラメータを取得するミップマップのレベルを指定します。0 を指定すると、一番低いレベルの(サイズが一番大きな)テクスチャのパラメータを取得します。その次のレベルは 1 を、さらに次のレベルは 2 を指定することで取得可能です。

pname には取得するパラメータの種類を指定します。指定可能な値と *params* に格納されるパラメータの対応は以下のとおりです。

表 11-11. 取得するテクスチャレベルパラメータの指定

pname に指定する値	パラメータ
GL_TEXTURE_WIDTH	テクスチャの幅(テクセル数)
GL_TEXTURE_HEIGHT	テクスチャの高さ(テクセル数)

GL_TEXTURE_DEPTH	未対応のため 0 固定
GL_TEXTURE_INTERNAL_FORMAT	テクスチャの内部フォーマット
GL_TEXTURE_BORDER	未対応のため 0 固定
GL_TEXTURE_RED_SIZE	赤成分のビット数(テクセル単位)
GL_TEXTURE_GREEN_SIZE	緑成分のビット数(テクセル単位)
GL_TEXTURE_BLUE_SIZE	青成分のビット数(テクセル単位)
GL_TEXTURE_ALPHA_SIZE	アルファ成分のビット数(テクセル単位)
GL_TEXTURE_LUMINANCE_SIZE	輝度成分のビット数(テクセル単位)
GL_TEXTURE_INTENSITY_SIZE	強度成分のビット数(テクセル単位。シャドウテクスチャのみ)
GL_TEXTURE_DEPTH_SIZE	深度成分のビット数(テクセル単位。シャドウテクスチャのみ)
GL_TEXTURE_DENSITY1_SIZE_DMP	交差が考慮されない密度情報(密度情報 1)のビット数(テクセル単位。ガステクスチャのみ)
GL_TEXTURE_DENSITY2_SIZE_DMP	交差が考慮される密度情報(密度情報 2)のビット数(テクセル単位。ガステクスチャのみ)
GL_TEXTURE_COMPRESSED	圧縮テクスチャかどうか GL_TRUE: 圧縮テクスチャ GL_FALSE: 非圧縮テクスチャ
GL_TEXTURE_COMPRESSED_IMAGE_SIZE	*level* で指定されたミップマップレベルのテクスチャのバイトサイズ(圧縮テクスチャのみ。非圧縮テクスチャに指定した場合は GL_INVALID_OPERATION エラー)

pname に GL_TEXTURE_INTERNAL_FORMAT を指定したときに取得するテクスチャの内部フォーマットと、テクセルを構成する各成分のビット数の対応を以下に示します。

表 11-12. 内部フォーマットとテクセルを構成する各成分のビット数の対応

内部フォーマット	赤	緑	青	アルファ	輝度	強度	深度	密度 1	密度 2
GL_RGBA4	4	4	4	4					
GL_RGB5_A1	5	5	5	1					
GL_RGBA	8	8	8	8					
GL_RGB565	5	6	5						
GL_RGB	8	8	8						
GL_ALPHA				8					
GL_ALPHA4_EXT				4					
GL_LUMINANCE					8				
GL_LUMINANCE4_EXT					4				
GL_LUMINANCE_ALPHA				8	8				
GL_LUMINANCE4_ALPHA4_EXT				4	4				
GL_SHADOW_DMP						8	24		

GL_GAS_DMP								16	16
GL_HILO8_DMP	8	8							
GL_ETC1_RGB8_NATIVE_DMP	8	8	8						
GL_ETC1_ALPHA_RGB8_A4_NATIVE_DMP	8	8	8	4					

target や *pname* に不正な値を指定した場合は GL_INVALID_ENUM のエラーを、ロードされていないミップマップレベルを *level* に指定した場合は GL_INVALID_VALUE のエラーを生成します。

11.1.5. テクスチャの設定によるパフォーマンスへの影響

テクスチャのフォーマットやサイズ、各種設定によって、グラフィックス処理のパフォーマンスに影響することがあります。以下にその傾向を列挙します。

- 圧縮テクスチャが最も高速に処理され、以降は 1 テクセルあたりのバイトサイズが小さいフォーマットの方が高速に処理されます。
- サイズが小さいほど高速に処理されます。
- メモリアクセスの競合が起こるため、同時に使用するテクスチャの枚数が多いほど処理が低速になります。
- 縮小時のフィルタ設定では、GL_NEAREST と GL_LINEAR、GL_NEAREST_MIPMAP_NEAREST と GL_LINEAR_MIPMAP_NEAREST、GL_NEAREST_MIPMAP_LINEAR と GL_LINEAR_MIPMAP_LINEAR のそれぞれが同じ速度で処理されます。ただし、GL_NEAREST(_XXX) と GL_LINEAR(_XXX) は 1 ピクセル辺りでフェッチするテクセル数が異なり、GL_NEAREST(_XXX) が 1 テクセル、GL_LINEAR(_XXX) が 4 テクセルですので、メモリ負荷は GL_NEAREST(_XXX) の方が小さくなります。
- テクスチャを縮小して貼る場合は、ミップマップがある方が高速に処理されます。ただし、ミップマップを使用した場合でもフィルタ設定によっては処理負荷が変わり、GL_*_MIPMAP_LINEAR は GL_*_MIPMAP_NEAREST に比べて 2 倍程度の処理負荷がかかる場合があります。
- 拡大時のフィルタ設定では、GL_NEAREST と GL_LINEAR はほぼ同じパフォーマンスですが、GL_NEAREST がわずかに高速になる場合があります。
- ガステクスチャとシャドウテクスチャはミップマップが使えないため、通常テクスチャより処理が低速になります。
- シャドウテクスチャは、シャドウ専用の特殊なフィルタを使用するため、トライリニアフィルタ(通常テクスチャで縮小フィルタを GL_*_MIPMAP_LINEAR に設定したとき)に相当する処理負荷がかかります。つまり、通常テクスチャ(トライリニアフィルタ設定除く)に比べて 2 倍程度の処理負荷がかかります。
- プロシージャルテクスチャには設定条件による差はありません。また、一般的な 2D テクスチャよりも高速に処理されます。
- 複数のテクスチャを使用する場合、すべてのテクスチャが VRAM-A または VRAM-B のどちらか一方にまとめて配置されている方が、分割して配置されているよりも高速に処理されます。
- 描画結果の上方向と一致するように作成されたテクスチャよりも、フレームバッファの上方向と一致するように作成されたテクスチャの方が高速に処理される可能性があります。これはフラグメントを生成する方向とテクスチャを読み込む方向が一致することで、テクスチャキャッシュのヒット率が向上するためです。上下の反転はパフォーマンスに影響しません。なお、フラグメントは 8x8 ピクセル単位で横方向に処理され、テクスチャは 8x4 テクセル単位で読み出されます。また、3DS の LCD は短い長さの辺を上下端としている点に注意してください。

11.1.6. テクスチャキャッシュについて

テクスチャキャッシュのサイズは、1 次キャッシュが 256 Byte、2 次キャッシュが 8 KByte です。キャッシュ内部では、圧縮テクスチャ(ETC フォーマット)だけがそのままのデータフォーマットで扱われ、それ以外のフォーマット(アルファ付き ETC フォーマットを含む)のテクスチャは 32 bit フォーマットに変換されます。

1 次キャッシュはユニットそれぞれに独立していますが、2 次キャッシュは全ユニットで共有しています。

1 次キャッシュにヒットせずに 2 次キャッシュからデータを取得する場合は約 5 サイクル、2 次キャッシュにもヒットせずに VRAM からデータを取得する場合はさらに 30 サイクル程度のペナルティが発生します。ただし、上記のペナルティを隠蔽できるように、テクセルのフェッチを先読みして行うようにハードウェアで実装されています。

テクスチャキャッシュは、4way セットアソシエイティブ方式です。キャッシュラインの数は 16 です。2 次キャッシュのサイズは 8K バイトですので、キャッシュラインは 1 ライン当り 512 バイトとなります。

キャッシュラインは、テクスチャ座標値から計算される、8x4 ブロックアドレス(4x4 テクセル単位のアドレスを 2 で割ったもの)の下位4ビットになります。ETC1 の場合は、4x4 テクセルブロックを 2x2 個並べたブロック単位のアドレスの [5:2] ビットの値になります。同一のキャッシュラインに連続的にアクセスした場合に、キャッシュのスラッシングが発生します。

11.2. コンバイナ

3DS に 6 基搭載されている(テクスチャ)コンバイナは、フラグメントライトのプライマリ、セカンダリのカラー、テクスチャユニットから出力されるテクスチャカラー、頂点カラー、定数カラーなどを合成することができます。ゲームキューブや Wii でのアプリケーション開発経験者でしたら、TEV でのカラー・アルファ合成処理をイメージしていただければ理解しやすいと思います。

OpenGL ES 1.1 ではコンバイナの設定を TexEnv で行っていましたが、3DS では予約ユニフォームへの値の設定で行います。TexEnv のパラメータと予約ユニフォームの対応は以下のようになっています。

表 11-13. OpenGL ES 1.1 の TexEnv のパラメータと予約ユニフォームの対応表

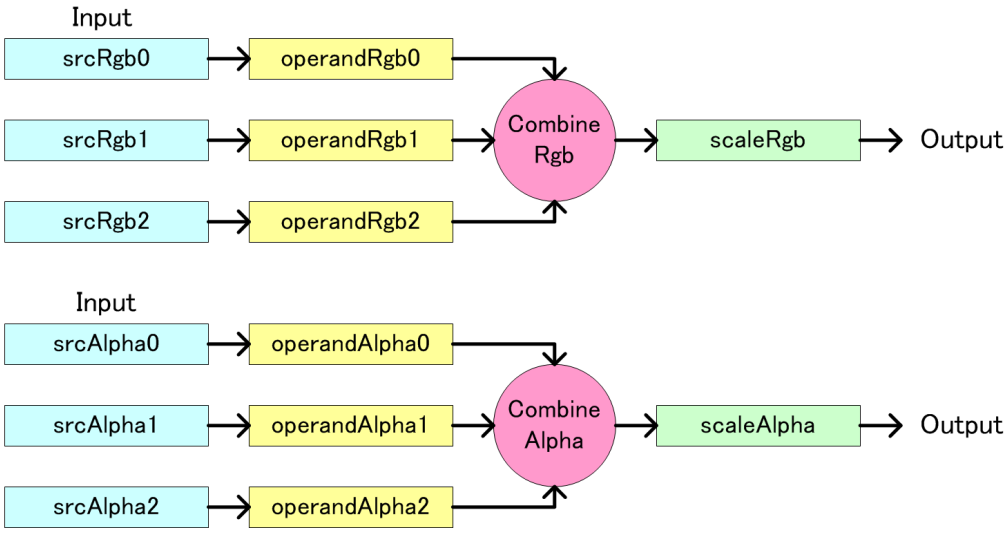
TexEnv	予約ユニフォーム	設定の内容
COMBINE_RGB	dmp_TexEnv[*i*].combineRgb	カラーのコンバイナ関数
COMBINE_ALPHA	dmp_TexEnv[*i*].combineAlpha	アルファのコンバイナ関数
SRC*n*_RGB	dmp_TexEnv[*i*].srcRgb	カラーのソース
SRC*n*_ALPHA	dmp_TexEnv[*i*].srcAlpha	アルファのソース
OPERAND*n*_RGB	dmp_TexEnv[*i*].operandRgb	カラーのオペランド
OPERAND*n*_ALPHA	dmp_TexEnv[*i*].operandAlpha	アルファのオペランド
RGB_SCALE	dmp_TexEnv[*i*].scaleRgb	カラーのスケーリング値
ALPHA_SCALE	dmp_TexEnv[*i*].scaleAlpha	アルファのスケーリング値
TEXTURE_ENV_COLOR	dmp_TexEnv[*i*].constRgba	定数カラー(アルファ成分あり)

n はソース(0 ～ 2)、*i* はコンバイナの番号(0 ～ 5)

各コンバイナはソースからの入力をオペランドで加工し、3 つのソースからの入力をコンバイナ関数で計算した結果にスケーリング値を乗算したものを 0.0 ～ 1.0 にクランプして出力します。また、コンバイナへの入力は計算前に 0.0 ～ 1.0 にクランプされており、頂点カラー(プライマリカラー)は絶対値でラスタライズした結果がクランプされた値となります。

カラー成分へのオペレーションは共通した設定でそれぞれの成分(赤、緑、青)に対して行われ、アルファ成分へのオペレーションは独立した設定で行われます。なお、無効化されたテクスチャユニットを参照した場合、入力されるカラー値は不定となります。

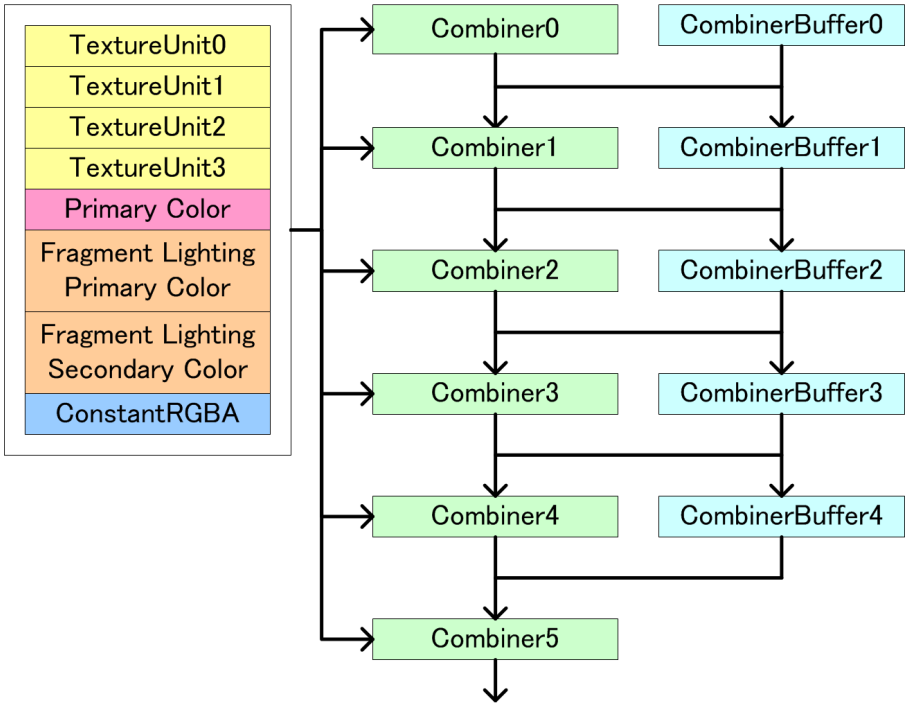
図 11-1. コンバイナのオペレーション



コンバイナへの 3 つの入力ソースは、それぞれ以下のものから選択(重複可)することができます。

- テクスチャユニットから出力されるテクスチャカラー
- 定数カラー
- プライマリカラー
- フラグメントライトのプライマリカラー
- フラグメントライトのセカンダリカラー
- 前段のコンバイナの出力(コンバイナ 0 以外)
- 前段のコンバイナバッファの出力(コンバイナ 0 以外)

図 11-2. コンバイナへの入力ソース



11.2.1. コンバイナ関数の予約ユニフォーム

コンバイナ関数の予約ユニフォームは dmp_TexEnv[*i*].combineRgb と dmp_TexEnv[*i*].combineAlpha です。glGetUniformLocation() で取得した予約ユニフォームのロケーションに glUniform1i() で値を設定します。

コンバイナ関数の予約ユニフォームへの設定値は以下のものから選択します。dmp_TexEnv[*i*].combineRgb と dmp_TexEnv[*i*].combineAlpha に設定可能な値は同じです。

GL_DOT3_RGBA を設定する場合は、そのコンバイナのカラー(combineRgb)とアルファ(combineAlpha)のコンバイナ関数は同じ(GL_DOT3_RGBA)でなければなりません。

表 11-14. コンバイナ関数の予約ユニフォームに設定可能な値

設定値	コンバイナ関数
GL_REPLACE	Src0(デフォルト)
GL_MODULATE	Src0 * Src1
GL_ADD	Src0 + Src1
GL_ADD_SIGNED	Src0 + Src1 – 0.5
GL_INTERPOLATE	Src0 * Src2 + Src1 * (1 – Src2)
GL_SUBTRACT	Src0 – Src1
GL_DOT3_RGB	4 * ((Src0_Red – 0.5) * (Src1_Red – 0.5) + (Src0_Green – 0.5) * (Src1_Green – 0.5) + (Src0_Blue – 0.5) * (Src1_Blue – 0.5))
GL_DOT3_RGBA	4 * ((Src0_Red – 0.5) * (Src1_Red – 0.5) + (Src0_Green – 0.5) * (Src1_Green – 0.5) + (Src0_Blue – 0.5) * (Src1_Blue – 0.5))
GL_ADD_MULT_DMP	(Src0 + Src1) * Src2 ※加算後にクランプ(0.0 ～ 1.0)されてから乗算されます
GL_MULT_ADD_DMP	(Src0 * Src1) + Src2

11.2.2. 入力ソースの予約ユニフォーム

入力ソースの予約ユニフォームは dmp_TexEnv[*i*].srcRgb と dmp_TexEnv[*i*].srcAlpha です。

glGetUniformLocation() で取得した予約ユニフォームのロケーションに glUniform3i() で値を設定します。先頭から、ソース 0、ソース 1、ソース 2 の順です。

入力ソースの予約ユニフォームへの設定値は以下のものから選択します。dmp_TexEnv[0].srcRgb と dmp_TexEnv[0].srcAlpha、dmp_TexEnv[*i*].srcRgb と dmp_TexEnv[*i*].srcAlpha に設定可能な値は同じです。

> **注意:** コンバイナ 0 以外のコンバイナは、3 つの入力ソースのいずれかに GL_CONSTANT、GL_PREVIOUS、GL_PREVIOUS_BUFFER_DMP のうちの 1 つが指定されていなければなりません。

表 11-15. 入力ソースの予約ユニフォームに設定可能な値

設定値	入力ソース
GL_TEXTURE0	テクスチャユニット 0 のテクスチャカラー
GL_TEXTURE1	テクスチャユニット 1 のテクスチャカラー
GL_TEXTURE2	テクスチャユニット 2 のテクスチャカラー
GL_TEXTURE3	テクスチャユニット 3 のテクスチャカラー
GL_CONSTANT	定数カラー(dmp_TexEnv[*i*].constRgba)(コンバイナ 0 のデフォルト)
GL_PRIMARY_COLOR	プライマリカラー(頂点カラー)
GL_PREVIOUS	前段のコンバイナの出力(コンバイナ 0 は設定不可。コンバイナ 0 以外のデフォルト)
GL_PREVIOUS_BUFFER_DMP	前段のコンバイナバッファの出力(コンバイナ 0 は設定不可)
GL_FRAGMENT_PRIMARY_COLOR_DMP	フラグメントライトのプライマリカラー
GL_FRAGMENT_SECONDARY_COLOR_DMP	フラグメントライトのセカンダリカラー

11.2.3. オペランドの予約ユニフォーム

オペランドの予約ユニフォームは dmp_TexEnv[*i*].operandRgb と dmp_TexEnv[*i*].operandAlpha です。

glGetUniformLocation() で取得した予約ユニフォームのロケーションに glUniform3i() で値を設定します。先頭から、ソース 0、ソース 1、ソース 2 の順です。

オペランドの予約ユニフォームへの設定値は以下のものから選択します。

表 11-16. オペランドの予約ユニフォームに設定可能な値

設定値	オペランド
GL_SRC_COLOR	Color(operandAlpha は設定不可。operandRgb のデフォルト)
GL_ONE_MINUS_SRC_COLOR	1 – Color(operandAlpha は設定不可)
GL_SRC_ALPHA	Alpha(operandAlpha のデフォルト)
GL_ONE_MINUS_SRC_ALPHA	1 – Alpha
GL_SRC_R_DMP	Color_Red
GL_ONE_MINUS_SRC_R_DMP	1 – Color_Red
GL_SRC_G_DMP	Color_Green
GL_ONE_MINUS_SRC_G_DMP	1 – Color_Green
GL_SRC_B_DMP	Color_Blue
GL_ONE_MINUS_SRC_B_DMP	1 – Color_Blue

11.2.4. スケーリング値の予約ユニフォーム

スケーリング値の予約ユニフォームは dmp_TexEnv[*i*].scaleRgb と dmp_TexEnv[*i*].scaleAlpha です。

glGetUniformLocation() で取得した予約ユニフォームのロケーションに glUniform1f() で値を設定します。

スケーリング値の予約ユニフォームへの設定値は以下のものから選択します。

表 11-17. スケーリング値の予約ユニフォームに設定可能な値

設定値	スケーリング
1.0	コンバイナの出力はそのまま(デフォルト)
2.0	コンバイナの出力を 2 倍にする(0.0 ～ 1.0 にクランプ)
4.0	コンバイナの出力を 4 倍にする(0.0 ～ 1.0 にクランプ)

11.2.5. 定数カラーの予約ユニフォーム

定数カラーの予約ユニフォームは dmp_TexEnv[*i*].constRgba です。glGetUniformLocation() で取得した予約ユニフォームのロケーションに glUniform4f() で値を設定します。先頭から R、G、B、A の順です。

定数カラーの予約ユニフォームのデフォルト設定値は (R, G, B, A) = (0.0, 0.0, 0.0, 0.0) です。

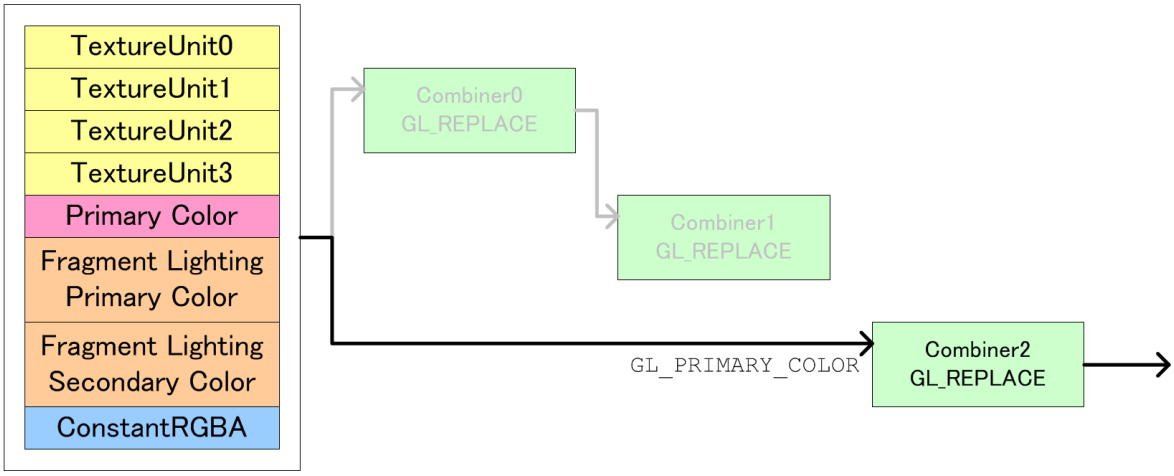
11.2.6. コンバイナの設定例

設定例として、プライマリカラーをそのまま描画するために、コンバイナ 2 でプライマリカラーを参照して、そのまま出力する接続の例を以下に示します。

コンバイナ 2 の入力ソース 0 にプライマリカラー(GL_PRIMARY_COLOR)を、オペランド 0 に入力ソースのカラーそのまま(GL_SRC_COLOR)を、コンバイナ関数に入力ソース 0 をそのまま出力(GL_REPLACE)を、スケールに 1.0 を設定すると、コンバイナの出力はプライマリカラーだけになり、それ以外の影響を受けなくなります。この設定では、コンバイナ 0 と 1 は出力結果に影響を与えません。

接続図は以下のようになっています。

図 11-3. コンバイナの設定例 1



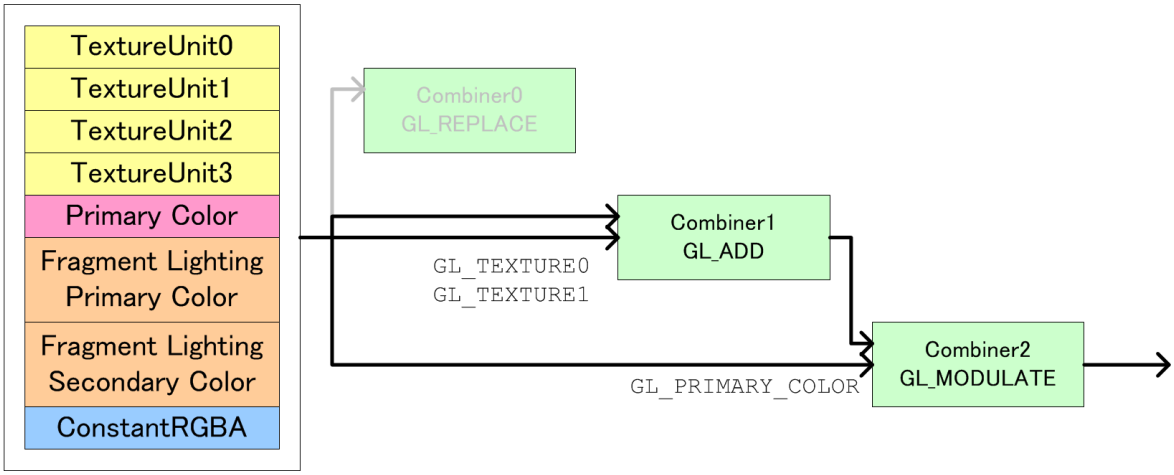
実際にプログラムで指定する際のコード例を以下に示します。

コード 11-6. コンバイナの設定コード例 1

```
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].srcRgb"),
        GL_PRIMARY_COLOR, GL_PREVIOUS, GL_PREVIOUS);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].srcAlpha"),
        GL_PRIMARY_COLOR, GL_PREVIOUS, GL_PREVIOUS);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[2].combineRgb"),
        GL_REPLACE);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[2].combineAlpha"),
        GL_REPLACE);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].operandRgb"),
        GL_SRC_COLOR, GL_SRC_COLOR, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].operandAlpha"),
        GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
glUniform1f(glGetUniformLocation(program, "dmp_TexEnv[2].scaleRgb"), 1.0);
glUniform1f(glGetUniformLocation(program, "dmp_TexEnv[2].scaleAlpha"), 1.0);
```

より複雑な設定例として、テクスチャ 0 とテクスチャ 1 の出力をコンバイナ 1 で加算し、その結果にコンバイナ 2 でプライマリカラーを乗算するような設定を示します。

図 11-4. コンバイナの設定例 2



コード 11-7. コンバイナの設定コード例 2

```
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[1].srcRgb"),
        GL_TEXTURE0, GL_TEXTURE1, GL_PREVIOUS);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[1].srcAlpha"),
        GL_TEXTURE0, GL_TEXTURE1, GL_PREVIOUS);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[1].combineRgb"),
        GL_ADD);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[1].combineAlpha"),
        GL_ADD);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[1].operandRgb"),
        GL_SRC_COLOR, GL_SRC_COLOR, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[1].operandAlpha"),
        GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].srcRgb"),
        GL_PREVIOUS, GL_PRIMARY_COLOR, GL_PREVIOUS);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].srcAlpha"),
        GL_PREVIOUS, GL_PRIMARY_COLOR, GL_PREVIOUS);
```

```
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[2].combineRgb"),
        GL_MODULATE);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[2].combineAlpha"),
        GL_MODULATE);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].operandRgb"),
        GL_SRC_COLOR, GL_SRC_COLOR, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].operandAlpha"),
        GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
```

11.3. コンバイナバッファ

最終のコンバイナ(コンバイナ 5)以外にはコンバイナバッファが並列して設置されています。コンバイナバッファは前段のコンバイナまたはコンバイナバッファの出力を入力ソースに選択することができ、前段のコンバイナバッファの出力を受け継いでいけば、後段のコンバイナで直前のコンバイナより前段にあるコンバイナの出力を入力ソースに選択することができます。

11.3.1. コンバイナバッファの予約ユニフォーム

コンバイナバッファ 0 への入力はありませんので、予約ユニフォーム dmp_TexEnv[0].bufferColor で初期値としての定数カラーを設定します。定数カラーは glGetUniformLocation() で取得した予約ユニフォームのロケーションに glUniform4f() で値を設定します。先頭から R、G、B、A の順です。

コンバイナバッファ 0 の定数カラーのデフォルト設定値は (R, G, B, A) = (0.0, 0.0, 0.0, 0.0) です。

コンバイナバッファ 1 ～ 4 への入力ソースは、前段のコンバイナの出力か前段のコンバイナバッファの出力から選択することができます。予約ユニフォームは dmp_TexEnv[*i*].bufferInput(*i* は 1 ～ 4)で、カラー成分とアルファ成分の入力を個別に選択することができます。glGetUniformLocation() で取得した予約ユニフォームのロケーションに glUniform2i() で値を設定します。先頭からカラー成分、アルファ成分の順です。

コンバイナバッファへの入力ソースの予約ユニフォームへの設定値は以下のものから選択します。

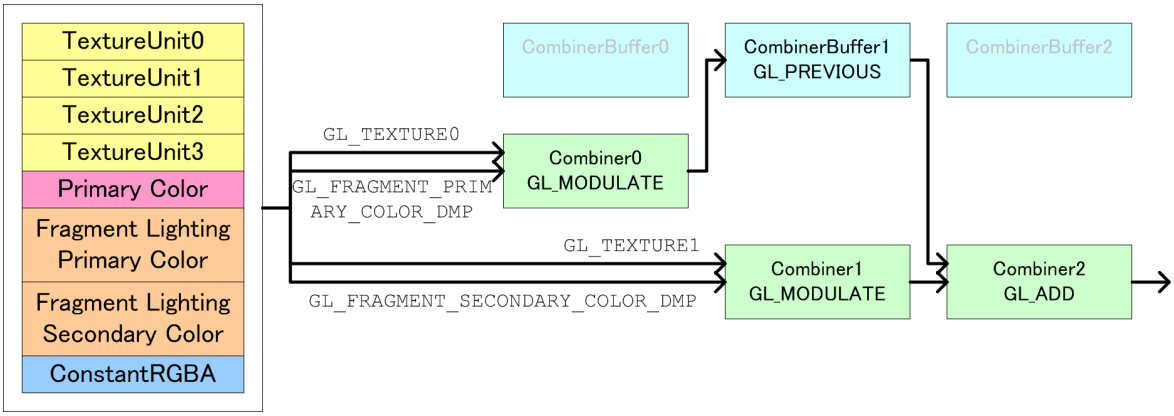
表 11-18. コンバイナバッファへの入力ソースの予約ユニフォームに設定可能な値

設定値	入力ソース
GL_PREVIOUS	前段のコンバイナの出力(デフォルト)
GL_PREVIOUS_BUFFER_DMP	前段のコンバイナバッファの出力

11.3.2. コンバイナバッファの設定例

コンバイナバッファを使用した設定例として、テクスチャ 0 にフラグメントライティングのプライマリカラーを乗算した結果とテクスチャ 1 にフラグメントライティングのセカンダリカラーを乗算した結果を加算して出力する接続の例を以下に示します。

図 11-5. コンバイナバッファの設定例



実際にプログラムで指定する際のコード例を以下に示します。

コード 11-8. コンバイナバッファの設定コード例

```
// Combiner 0
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[0].srcRgb"),
        GL_TEXTURE0, GL_FRAGMENT_PRIMARY_COLOR_DMP, GL_PREVIOUS);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[0].srcAlpha"),
        GL_TEXTURE0, GL_PREVIOUS, GL_PREVIOUS);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[0].combineRgb"),
        GL_MODULATE);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[0].combineAlpha"),
        GL_REPLACE);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[0].operandRgb"),
        GL_SRC_COLOR, GL_SRC_COLOR, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[0].operandAlpha"),
        GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
// CombinerBuffer 1
glUniform2i(glGetUniformLocation(program, "dmp_TexEnv[1].bufferInput"),
        GL_PREVIOUS, GL_PREVIOUS);
// Combiner 1
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[1].srcRgb"),
        GL_TEXTURE1, GL_FRAGMENT_SECONDARY_COLOR_DMP, GL_PREVIOUS);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[1].srcAlpha"),
        GL_TEXTURE1, GL_PREVIOUS, GL_PREVIOUS);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[1].combineRgb"),
        GL_MODULATE);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[1].combineAlpha"),
        GL_REPLACE);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[1].operandRgb"),
        GL_SRC_COLOR, GL_SRC_COLOR, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[1].operandAlpha"),
        GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
// Combiner 2
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].srcRgb"),
        GL_PREVIOUS_BUFFER_DMP, GL_PREVIOUS, GL_PREVIOUS);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].srcAlpha"),
        GL_PREVIOUS_BUFFER_DMP, GL_PREVIOUS, GL_PREVIOUS);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[2].combineRgb"),
```

```
        GL_ADD);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[2].combineAlpha"),
        GL_REPLACE);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].operandRgb"),
        GL_SRC_COLOR, GL_SRC_COLOR, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].operandAlpha"),
        GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
```

11.4. プロシージャルテクスチャ

プロシージャルテクスチャは従来のテクスチャと異なり、画像を参照せず、手続き的計算によりテクセルの色が決定されます。プロシージャルテクスチャは、完全に秩序を持ったパターンや、ややランダムで秩序を伴うパターンで効果を発揮します。テクスチャ画像にアクセスをすることがないため、メモリアクセスの競合を回避したり、コンテンツサイズを抑制したりといった効果があります。

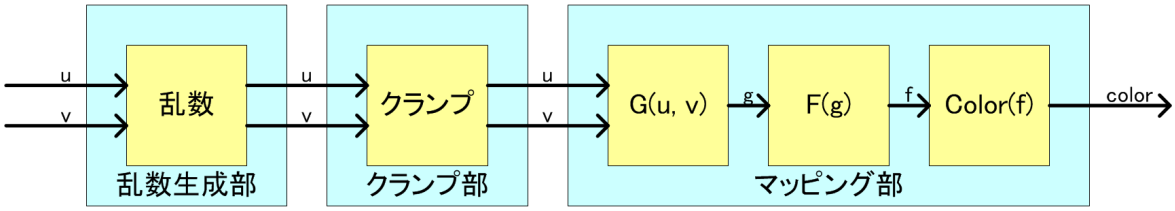
プロシージャルテクスチャを扱うことができるのはテクスチャユニット 3 だけで、テクスチャユニット 3 はプロシージャルテクスチャ専用のテクスチャユニットになっています。計算によりテクセル色を決定しますが、テクスチャ座標 u、v に対応するテクセル色を決定するという意味では通常のテクスチャと同様の働きをしていると考えることもできます。

プロシージャルテクスチャは予約フラグメントシェーダの一部でもありますので、パラメータの設定は予約ユニフォームへの `glUniform*()` の呼び出しで行います。

11.4.1. プロシージャルテクスチャユニット

プロシージャルテクスチャユニットは 3 つの計算部で構成されています。入力に近い側から、乱数生成部、クランプ部、マッピング部と呼びます。乱数生成部がテクスチャ座標 u、v に揺らぎを与え、クランプ部がパターンの鏡面対称、繰り返しを決定し、マッピング部が u、v 座標値からテクセル色を計算します。

図 11-6. プロシージャルテクスチャユニットの構成



入力されたテクスチャ座標は上図のように処理されていますが、目的の画像を得るためのパラメータを設定するときには、以下の手順で行った方が直感的で理解しやすいでしょう。

1. プロシージャルテクスチャの有効化
2. RGBA 共有モードまたはアルファ独立モードの設定
   G(u, v) および F(g) の設定に相当します。
3. 基本形状の選択
   G(u, v) の選択に相当します。
4. 基本色の設定
   色はカラー参照テーブルとして設定します。Color(f) の設定に相当します。
5. 基本形状とカラー参照テーブルの関係の設定
   手順 3 の基本形状と手順 4 のカラー参照テーブルがどのように対応するか決定します。F(g) の設定に相当します。
6. 乱数パラメータの選択
   必要な場合には乱数を有効にして、乱数の影響の大きさを決めます。不要な場合は乱数を無効にします。乱数生成部に相当します。乱数の有効、無効に関わらず、u と v は絶対値化されてクランプ部に出力されます。

7. 繰り返しおよび対称性の設定
   クランプの設定に相当します。

11.4.2. プロシージャルテクスチャの有効化

プロシージャルテクスチャで使用するテクスチャユニット 3 の有効・無効の設定は、予約ユニフォーム(dmp_Texture[3].samplerType)への値の設定で行います。glActiveTexture() によるユニット選択や glEnable() によるテクスチャ種別の選択はエラーとなりますので注意してください。

有効にする場合は GL_TEXTURE_PROCEDURAL_DMP を、無効にする場合は GL_FALSE を glUniform1i() で設定してください。テクスチャの種別として GL_TEXTURE_PROCEDURAL_DMP 以外には対応していません。

コード 11-9. プロシージャルテクスチャの有効化

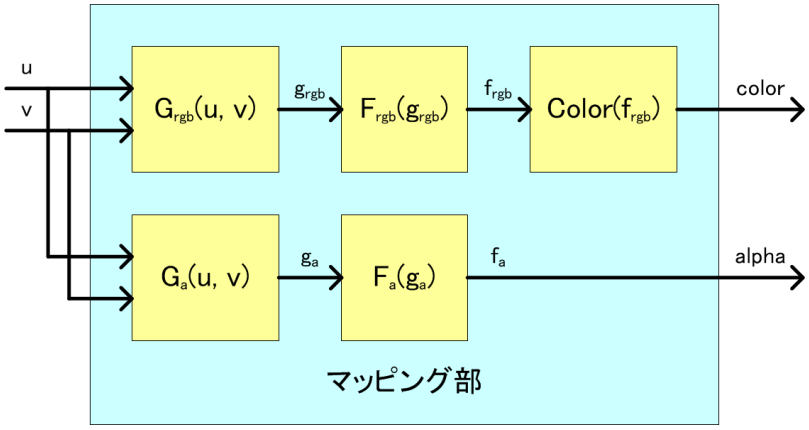
```
glUniform1i(
    glGetUniformLocation(s_PgID, "dmp_Texture[3].samplerType"),
    GL_TEXTURE_PROCEDURAL_DMP);
```

11.4.3. RGBA 共有モード / アルファ独立モードの設定

アルファ成分のマッピング部での計算に使用する関数を RGB 成分の設定と共有(RGBA 共有モード)するか、独立して設定(アルファ独立モード)するかを選択します。アルファ独立モードを選択した場合、マッピング部の G 関数と F 関数を 2 つずつ設定しなければなりません。とりあえず画像を出力してみたいという場合は、RGBA 共有モードを使用した方が設定は容易です。

予約ユニフォーム(dmp_Texture[3].ptAlphaSeparate)に glUniform1i() で GL_FALSE を設定した場合は RGBA 共有モード、GL_TRUE を設定した場合はアルファ独立モードが選択されます。デフォルトは RGBA 共有モードです。

図 11-7. アルファ独立モード時のマッピング部



11.4.4. 基本形状の選択

プロシージャルテクスチャの基本形状をマッピング部の G 関数によって決定します。

表 11-19 で掲載している基本形状は以下の割り当てで色付けされています。

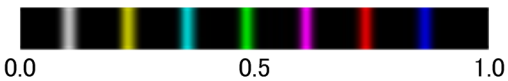
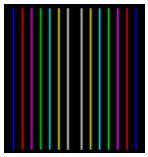
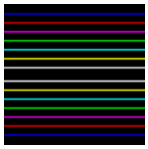
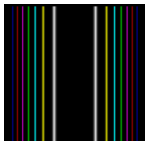
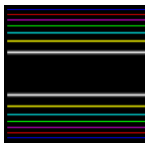
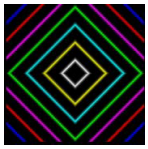
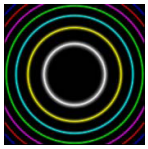
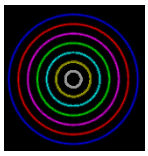
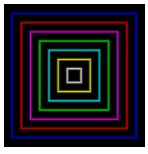
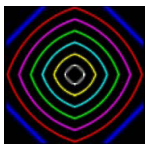


表 11-19. G 関数の設定値と選択される関数および基本形状

設定値	選択される関数	基本形状
GL_PROCTEX_U_DMP (デフォルト)	u	
GL_PROCTEX_V_DMP	v	
GL_PROCTEX_U2_DMP	u^2	
GL_PROCTEX_V2_DMP	v^2	
GL_PROCTEX_ADD_DMP	$(u + v) / 2$	
GL_PROCTEX_ADD2_DMP	$(u^2 + v^2) / 2$	
GL_PROCTEX_ADDSQRT2_DMP	$sqrt(u^2 + v^2)$	
GL_PROCTEX_MIN_DMP	min(u, v)	
GL_PROCTEX_MAX_DMP	max(u, v)	

GL_PROCTEX_RMAX_DMP	$((u + v) / 2 + \mathrm{sqrt}(u^2 + v^2)) / 2$	

基本形状はテクスチャ座標に -1.0 ～ 1.0(中心が 0.0)、繰り返しの指定に GL_MIRRORED_REPEAT を設定して描画されたものです。

G 関数選択のための予約ユニフォームは、RGB 成分が dmp_Texture[3].ptRgbMap、アルファ成分が dmp_Texture[3].ptAlphaMap です。それぞれの予約ユニフォームに対して、glUniform1i() で設定します。

dmp_Texture[3].ptAlphaMap への設定はアルファ独立モードでのみ有効です。

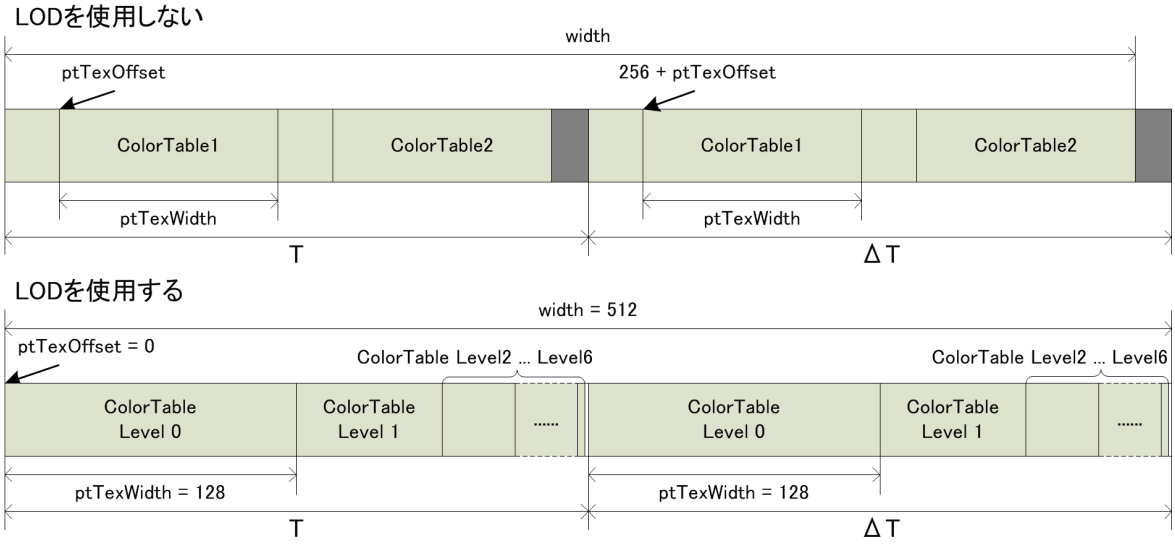
木目ならば GL_PROCTEX_U_DMP や GL_PROCTEX_V_DMP を、年輪ならば GL_PROCTEX_ADDSQRT2_DMP を選択するなど、目的とするテクスチャ画像に近い形状を選んでください。

11.4.5. カラー参照テーブルの設定

カラー参照テーブルは G 関数と F 関数を経て計算された値を実際のテクセル色へと変換するためのテーブルで、RGBA の各成分で別々に設定することができます。

カラー参照テーブルの内容は LOD を使用するかどうかで異なります。

図 11-8. LOD の使用によるカラー参照テーブルの内容の違い



LOD を使用しない場合、カラー参照テーブルには部分配列として複数のカラーテーブルを格納することができます。オフセットとテーブルの幅を変更することで、同じ計算結果から色合いの異なるテクスチャを描画することもできます。

カラー参照テーブルは最大 512 要素の参照テーブルで、256 要素を境にして前半にはカラーテーブル、後半には前半で格納したカラーテーブルの要素間の差分値を格納します。差分値は必ず 257 要素目から格納しなければならないので、カラー参照テーブルの要素数は "256 +(実際に参照されるカラーテーブルの要素数)+(カラーテーブルの先頭オフセット)" で定義されます。要素数の計算で対象となるのは最後のカラーテーブルです。

実際に参照されるカラーテーブルの幅(dmp_Texture[3].ptTexWidth)には、128 までの 2 のべき乗を glUniform1i() で設定します。カラーテーブルの先頭オフセット(dmp_Texture[3].ptTexOffset)には、0 ～ 128 の整数を glUniform1i() で設定します。

LOD を使用する場合、カラーテーブルの幅とオフセットは、それぞれ 128 と 0 でなければなりません。LOD のレベルに対応して、実際に参照されるカラーテーブルが決定されます。カラー参照テーブルの要素数は最大の 512 固定です。

表 11-20. LOD のレベルとカラーテーブルの対応

LOD のレベル	開始位置	幅
0	0	128
1	128	64
2	192	32
3	224	16
4	240	8
5	248	4
6	252	2

「7.7. 参照テーブルのロード」で説明したとおり、参照テーブルは `glTexImage1D()` を呼び出して配列からロードします。カラー参照テーブルの要素数の大きさの浮動小数点配列を用意して、前半部分にカラーテーブルの要素(0.0 ～ 1.0)を格納し、配列の後半(ΔT)には前半 256 要素に格納したカラーテーブル(T)の要素間の差分値を格納していきます。カラーテーブルの要素数を size、先頭オフセットを offset 要素を C_i 、変換関数を func() とすれば、要素と差分値の計算式は以下の式で表されます。

$$C_{i+offset} = func\left(\frac{i}{size}\right) \qquad (0 \le i < size)$$

$$C_{i+offset+256} = C_{i+offset+1} - C_{i+offset} \qquad (0 \le i < size)$$

差分値の最後はカラーテーブルの収束値と最終要素との差分を計算して格納するか、単に 0.0 を格納してください。

`glTexImage1D()` の呼び出しはカラー参照テーブルの要素数を最大の 512、カラー参照テーブルを格納した配列を `data`、設定する参照テーブル番号を 0 とすれば以下のコードで行うことができます。

コード 11-10. カラー参照テーブルのロード

```
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
             GL_LUMINANCEF_DMP, GL_FLOAT, data);
```

RGBA の各成分で使用するカラー参照テーブルは、`glUniform1i()` で参照テーブル番号を以下の予約ユニフォームに指定します。ここで指定する参照テーブル番号は `GL_LUT_TEXTURE`*i*`_DMP` の *i*(0 ～ 31) が表す数値で、テクスチャの名前(ID)や `GL_LUT_TEXTURE`*i*`_DMP` を直接指定するわけではないことに注意してください。

表 11-21. カラー参照テーブルを指定する予約ユニフォーム

予約ユニフォーム	設定する値
`dmp_Texture[3].ptSamplerR`	赤成分のカラー参照テーブルとする参照テーブル番号を指定する。
`dmp_Texture[3].ptSamplerG`	緑成分のカラー参照テーブルとする参照テーブル番号を指定する。
`dmp_Texture[3].ptSamplerB`	青成分のカラー参照テーブルとする参照テーブル番号を指定する。

dmp_Texture[3].ptSamplerA	アルファ成分のカラー参照テーブルとする参照テーブル番号を指定する。

どの予約ユニフォームも 0 ～ 31 の値を指定します。

アルファ独立モード時は dmp_Texture[3].ptSamplerA の設定が無視されます。

プロシージャルテクスチャのカラー参照テーブルにも、縮小時のフィルタには通常のテクスチャと同様のフィルタを適用することができます。予約ユニフォーム(dmp_Texture[3].ptMinFilter)に glUniform1i() で設定する値を、以下の表から選択します。

表 11-22. カラー参照テーブルのフィルタ

設定する値	適用するフィルタ
GL_NEAREST	u, v 方向はニアレスト、LOD なし
GL_LINEAR	u, v 方向はリニア、LOD なし(デフォルト)
GL_NEAREST_MIPMAP_NEAREST	u, v 方向はニアレスト、LOD はニアレスト
GL_NEAREST_MIPMAP_LINEAR	u, v 方向はニアレスト、LOD はリニア
GL_LINEAR_MIPMAP_NEAREST	u, v 方向はリニア、LOD はニアレスト
GL_LINEAR_MIPMAP_LINEAR	u, v 方向はリニア、LOD はリニア

カラー参照テーブルの参照時に LOD バイアスをかけることができます。0.0 ～ 6.0 の範囲の値を、予約ユニフォーム(dmp_Texture[3].ptTexBias)に glUniform1f() で設定します。0.0 で無効化され、デフォルトは 0.5 に設定されています。

11.4.6. 基本形状とカラー参照テーブルとの対応関係の設定

基本形状を選択した G 関数と基本色を設定したカラー参照テーブルとの対応関係は F 関数で設定されます。F 関数は、G 関数からの出力(0.0 ～ 1.0)をカラー参照テーブルの参照値(0.0 ～ 1.0)へマッピングする参照テーブルとして設定します。参照テーブルは 256 要素の大きさで、128 要素を境にして前半にはマッピングテーブル、後半には前半で格納したマッピングテーブルの要素間の差分値を格納します。形状とカラーの対応関係の設定ですので、同じ形状とカラー参照テーブルを用いていても、マッピングテーブルを変更することで様相の異なるテクスチャとして描画することができます。F 関数に F(x)=x や $F(x)=x^2$ のように単純な計算結果を用いたり、飛び飛びの値でインデックスのように用いたりするだけでも出力結果は大きく異なるものになる可能性があります。

カラー参照テーブルと同様に、F 関数のマッピングテーブルは glTexImage1D() を呼び出して、配列から参照テーブルとしてロードします。マッピングテーブルの要素数(256)の大きさの浮動小数点配列を用意して、前半部分にマッピングテーブルの要素(0.0 ～ 1.0)を格納し、配列の後半には前半の 128 要素に格納したマッピングテーブルの要素間の差分値を格納していきます。マッピングテーブルの要素を F_i 、変換関数を func() とすれば、要素と差分値の計算式は以下の式で表されます。

$$F_i = func\left(\frac{i}{128.0}\right) \qquad (0 \le i \le 127)$$

$$F_{k+128} = func\left(\frac{k+1}{128.0}\right) - func\left(\frac{k}{128.0}\right) \quad (0 \le k \le 127)$$

glTexImage1D() の呼び出しはマッピングテーブルの要素数を 256、マッピングテーブルを格納した配列を data、設定

する参照テーブル番号を 0 とすれば以下のコードで行うことができます。

コード 11-11. マッピングテーブルのロード

```
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 256, 0,
             GL_LUMINANCEF_DMP, GL_FLOAT, data);
```

F 関数として使用するマッピングテーブルは、glUniform1i() で参照テーブル番号を以下の予約ユニフォームに指定します。ここで指定する参照テーブル番号は GL_LUT_TEXTURE*i*_DMP の *i*(0 ～ 31) が表す数値で、テクスチャの名前(ID)や GL_LUT_TEXTURE*i*_DMP を直接指定するわけではないことに注意してください。

表 11-23. マッピングテーブルを指定する予約ユニフォーム

予約ユニフォーム	設定する値
dmp_Texture[3].ptSamplerRgbMap	RGB の F 関数とする参照テーブル番号を指定する。
dmp_Texture[3].ptSamplerAlphaMap	アルファの F 関数とする参照テーブル番号を指定する。

どちらの予約ユニフォームも 0 ～ 31 の値を指定します。

dmp_Texture[3].ptSamplerAlphaMap の設定はアルファ独立モード時のみ有効です。

11.4.7. 乱数パラメータの選択

プロシージャルテクスチャのランダム要素として、G 関数に入力されるテクスチャ座標 u と v に揺らぎ(ノイズ)を与えることができます。ノイズは基本形状に影響を与えます。例えば G 関数が GL_PROCTEX_U_DMP であるとき、テクスチャ座標 u にノイズによる影響を与えれば、直線でしか描画できなかった木目がより自然な歪みを持って描画できるようになります。

ノイズの有効・無効の設定は、予約ユニフォーム(dmp_Texture[3].ptNoiseEnable)への値の設定で行います。有効にする場合は GL_TRUE を、無効にする場合は GL_FALSE を glUniform1i() で設定してください。

コード 11-12. ノイズの有効化

```
glUniform1i(
    glGetUniformLocation(s_PgID, "dmp_Texture[3].ptNoiseEnable"), GL_TRUE);
```

ノイズを与える関数はブラックボックスになっていますが、アプリケーションから波の周波数(F)、位相(P)、振幅(A)の 3 パラメータを与えることで制御することができます。F パラメータは揺らぎの緩急を調整し、大きくすれば険しく、小さくすれば緩やかな波になります。P パラメータは揺らぎの開始位置を変更します。海面のテクスチャなどを描画する際には、P パラメータだけを変更して波の変化を表現することができます。A パラメータを大きくすれば揺らぎの影響が大きくなり、基本形状はより崩れます。

F、P、A の 3 パラメータは u 成分、v 成分で別々のパラメータを設定することができます。

表 11-24. ノイズの予約ユニフォーム

予約ユニフォーム	設定する値
dmp_Texture[3].ptNoiseU	u 成分の F、P、A パラメータを指定する。A パラメータのみ -8.0 ～ 8.0 の範囲にクランプされます。 (F-parameter, P-parameter, A-parameter) デフォルト (0.0, 0.0, 0.0)

dmp_Texture[3].ptNoiseV	v 成分の F、P、A パラメータを指定する。A パラメータのみ -8.0 ～ 8.0 の範囲にクランプされます。 (F-parameter, P-parameter, A-parameter) デフォルト (0.0, 0.0, 0.0)

揺らぎのパラメータ以外にも、ノイズを与える関数にノイズ変調と呼ばれる乱数の連続性の変化を制御することができます。ノイズ変調(ノイズ連続性関数)は参照テーブルで指定し、これをノイズ変調テーブルと呼びます。ノイズ関数は与えられたノイズ変調テーブルを使って、計算だけでは離散的な値となってしまう揺らぎを自然な値にしようとします。ノイズ連続性関数には $3x^2 - 2x^3$ のように 0.0 と 1.0 付近で緩やかに変化する関数が適しています。

カラー参照テーブルと同様に、ノイズ変調テーブルは glTexImage1D() を呼び出して配列からロードします。ノイズ変調テーブルの要素数(256)の大きさの浮動小数点配列を用意して、前半部分にノイズ変調テーブルの要素(0.0 ～ 1.0)を格納し、配列の後半には前半の 128 要素に格納したノイズ変調テーブルの要素間の差分値を格納していきます。ノイズ変調テーブルの要素を N_i 、変換関数を func() とすれば、要素と差分値の計算式は以下の式で表されます。

$$N_i = func\left(\frac{i}{128.0}\right) \qquad (0 \le i \le 127)$$

$$N_{k+128} = func\left(\frac{k+1}{128.0}\right) - func\left(\frac{k}{128.0}\right) \quad (0 \le k \le 127)$$

glTexImage1D() の呼び出しはノイズ変調テーブルの要素数を 256、ノイズ変調テーブルを格納した配列を data、設定する参照テーブル番号を 0 とすれば以下のコードで行うことができます。

コード 11-13. ノイズ変調テーブルのロード

```
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 256, 0,
                  GL_LUMINANCEF_DMP, GL_FLOAT, data);
```

ノイズ連続性関数として使用するノイズ変調テーブルは、参照テーブル番号を以下の予約ユニフォームへ glUniform1i() によって指定します。ここで指定する参照テーブル番号は GL_LUT_TEXTURE*i*_DMP の *i*(0 ～ 31)が表す数値で、テクスチャの名前(ID)や GL_LUT_TEXTURE*i*_DMP を直接指定するわけではないことに注意してください。

表 11-25. ノイズ変調テーブルを指定する予約ユニフォーム

予約ユニフォーム	設定する値
dmp_Texture[3].ptSamplerNoiseMap	ノイズ連続性関数とする参照テーブル番号を指定する。 0 ～ 31

ノイズの F、P、A の 3 パラメータが出力結果にどのような影響を与えるのかを、ノイズの影響がないときには同心円で描画されるプロシージャルテクスチャに対して、u, v 両方向に関して各パラメータを変化させたときの描画結果で紹介します。これらの描画に際して、ノイズ連続性関数には F(x) = x を使用しています。

図 11-9 は A パラメータのみを変化させたときの描画結果です。そのほかのパラメータについては、F = 0.3、P = 0.0 に設定しています。A が大きいほど大きく波打ちますが、円周上で影響を受ける箇所に変化がほとんどないことに注目してください。

図 11-9. A パラメータの影響

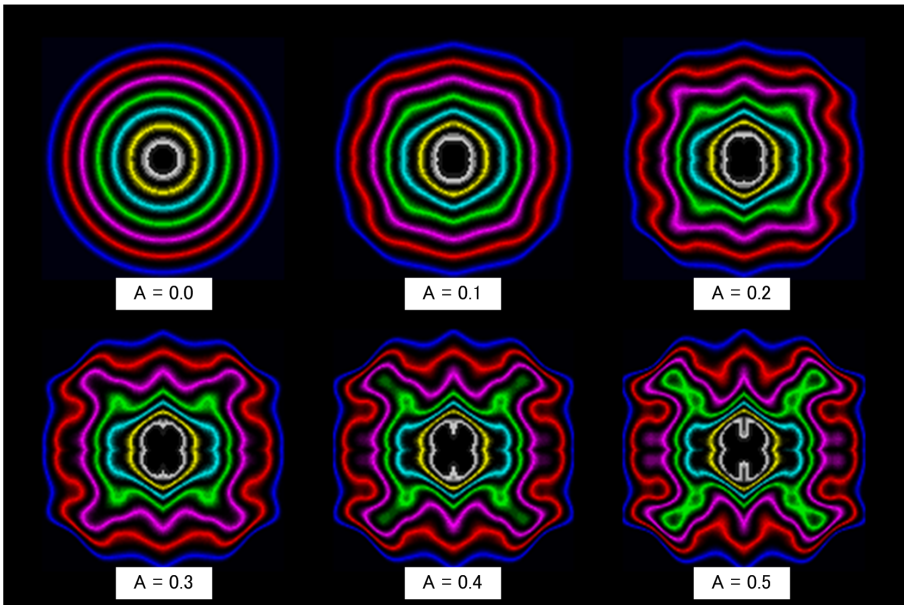


図 11-10 は F パラメータのみを変化させたときの描画結果です。そのほかのパラメータについては、A = 0.3、P = 0.0 に設定しています。F が大きいほど揺らぎの周波数が高くなり、円周上で影響を受ける箇所の間隔が狭くなっていることに注目してください。また、F パラメータはノイズの計算時に絶対値が使用されるため、符号を反転させても結果は変わりません。

図 11-10. F パラメータの影響

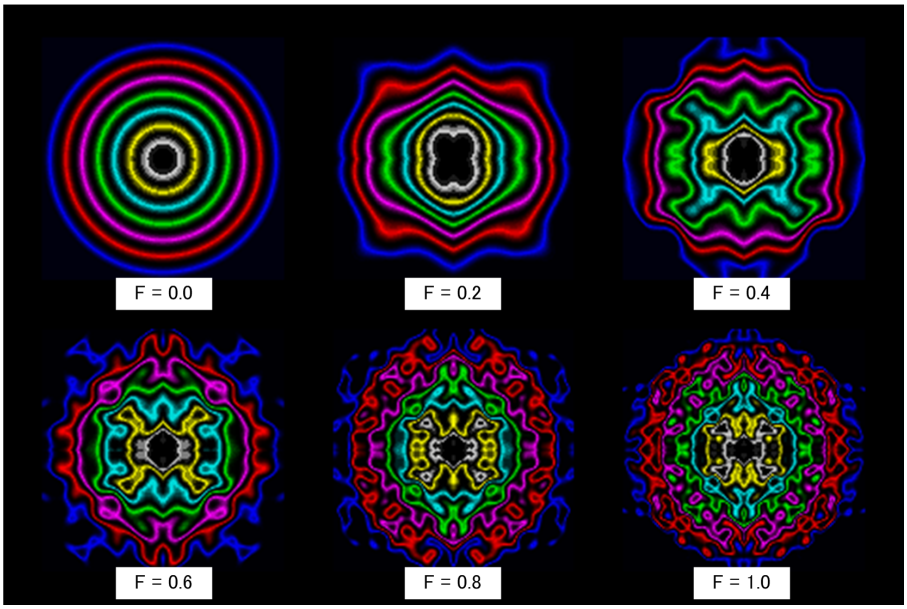


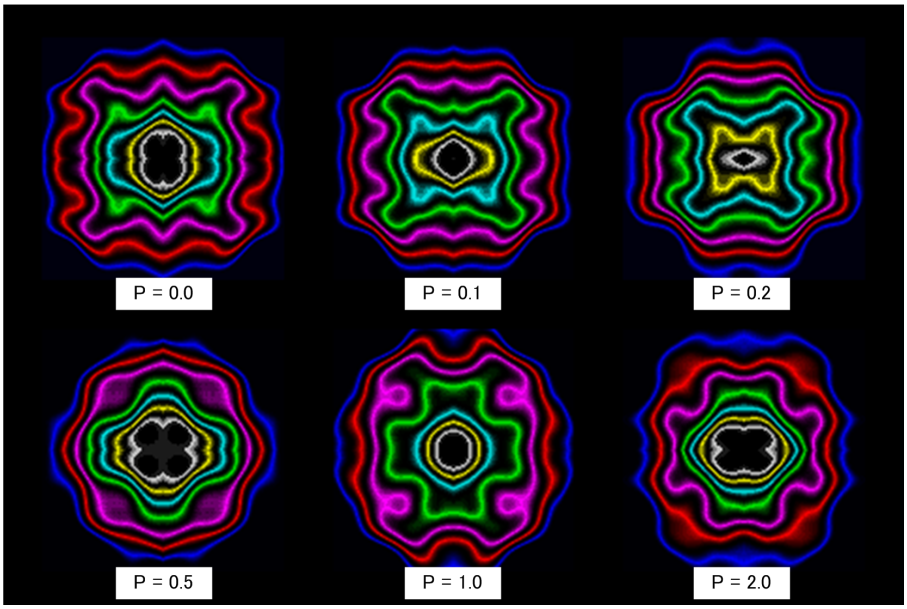
図 11-11 は P パラメータのみを変化させたときの描画結果です。そのほかのパラメータについては、A = 0.3、F = 0.3 に設定しています。P が変化すると揺らぎの形状のみが変化していることに注目してください。P パラメータを変化させることで、プロシージャルテクスチャにアニメーションのような変化をもたらすことができます。

P パラメータを変化させてアニメーションさせる場合、P パラメータに大きな値を設定すると、P パラメータの小さな変化が揺らぎの形状に影響を与えなくなります。これはハードウェアの計算精度に起因する現象です。例えば、フレームごとに一定の値を P パラメータに加算してノイズの変化を使ったアニメーションを行う場合は、P パラメータが大きな値になる前に小さな値に戻す必要があります。F パラメータと P パラメータがともに正の数である場合、F パラメータと P パラメータの乗算結果が 16 の倍数となるときと、P パラメータに 0.0 を設定したときは同じ結果を得ることができるという特性があります。つまり、乗算

結果が 16 となるときに P パラメータを 0 に戻すことで、アニメーションの連続性を保つことができます。ただし、F パラメータに大きな値が設定されているときは、P パラメータに 0 を指定した場合と、乗算結果が 16 となる場合で同じ形状にならない場合があります。

F パラメータと A パラメータが固定値かつ、(|u| + u の位相)および(|v| + v の位相)の符号が変わらない範囲で P パラメータの値を変化させた場合、F×P が任意の値 X となる場合と、X+16 となる場合とで同じノイズ結果を得ることができます。ただし、ノイズ計算の過程における計算精度により、F パラメータに大きな値を設定すると、上記の場合でも同じ乱数結果を得られない可能性があります。また、P パラメータに大きな値が設定されていると、細かな値の変化がノイズ結果に反映されなくなる場合があります。

図 11-11. P パラメータの影響

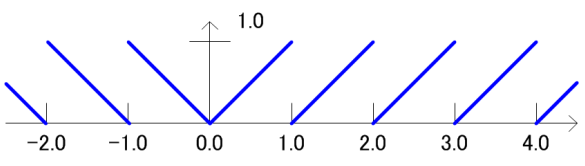


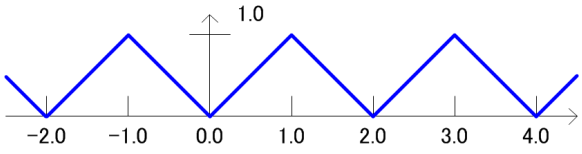
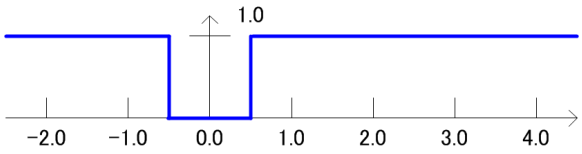
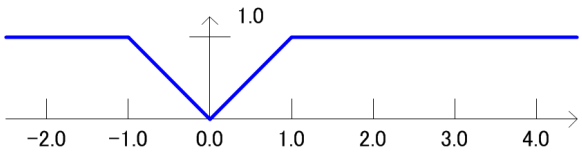
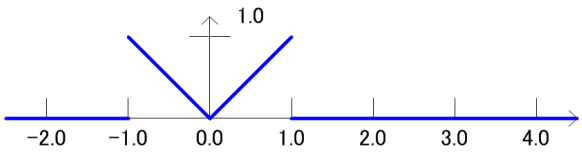
11.4.8. 繰り返しおよび対称性の設定

通常のテクスチャで設定したラッピングモードと同様の機能が、プロシージャルテクスチャにもあります。クランプ計算部と呼ばれ、設定可能なモードにはパルスやゼロクランプなどの専用モードがあります。また、繰り返しの途中でテクスチャ座標の整数部分が同じ部分をブロックにしてずらす、シフト計算部もあります。

クランプ計算部ではテクスチャ座標が 0.0 ～ 1.0 の範囲に収まらない部分を、どのように範囲内の値に変換するのかをクランプモードによって決定します。

表 11-26. クランプモード

クランプモード	座標値のクランプ
GL_SYMMETRICAL_REPEAT_DMP	

GL_MIRRORED_REPEAT	
GL_PULSE_DMP	
GL_CLAMP_TO_EDGE (デフォルト)	
GL_CLAMP_TO_ZERO_DMP	

テクスチャ座標の u 成分と v 成分で別々のクランプモードを設定することができます。設定するには、予約ユニフォーム dmp_Texture[3].ptClampU または dmp_Texture[3].ptClampV に対して glUniform1i() を呼び出します。

GL_SYMMETRICAL_REPEAT_DMP は、同じ画像が格子状に並びます。GL_MIRRORED_REPEAT は、偶数番目が鏡のように反転します。GL_PULSE_DMP は、描画に利用しようとする画素に最も近いテクスチャの端の画素を拾います。GL_CLAMP_TO_EDGE は、-1.0 ～ 1.0 の範囲内ではテクスチャ内部の画像を参照し、範囲外ではテクスチャの端の画素を参照します。GL_CLAMP_TO_ZERO_DMP は -1.0 ～ 1.0 の範囲内(-1.0 および 1.0 を含まず)ではテクスチャ画像を参照し、範囲外(-1.0 および 1.0 を含む)では座標 0 の画像を参照します。

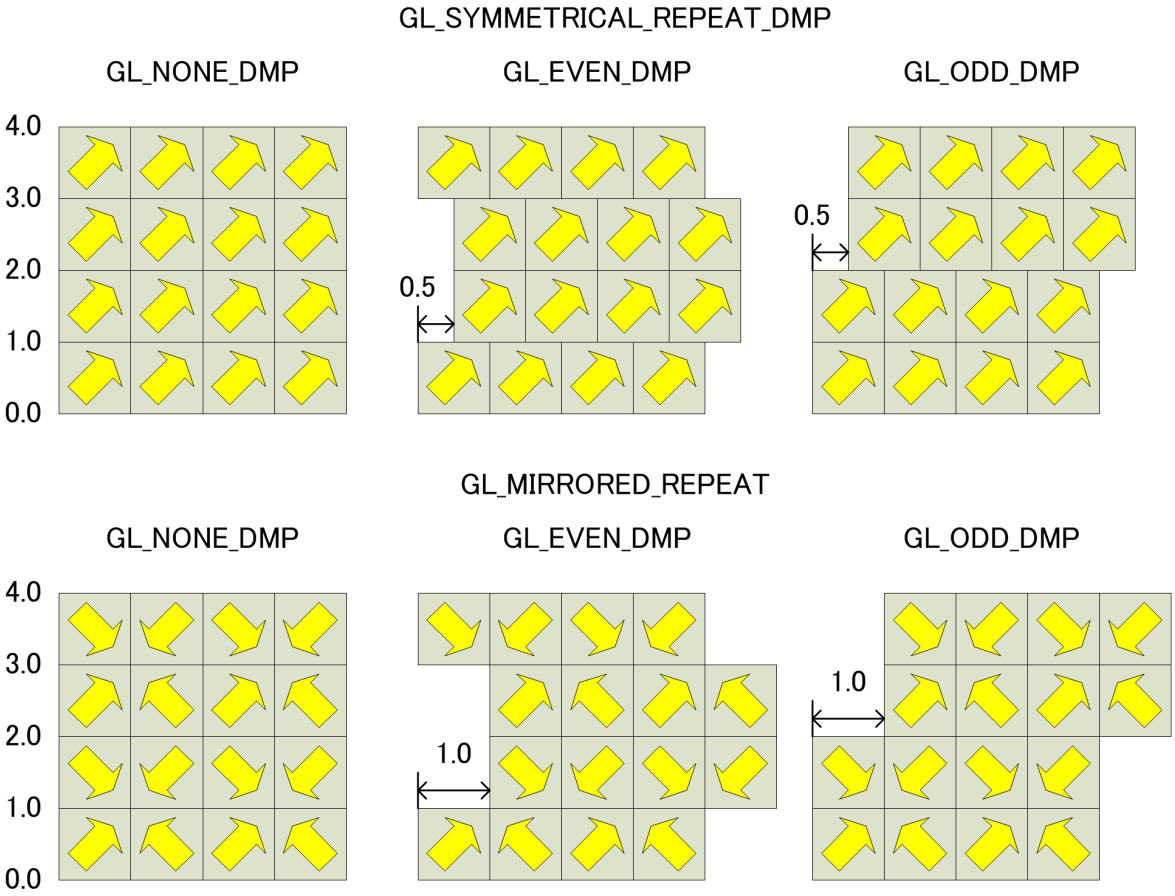
シフト計算部では、シフトモードによってどのように座標をシフト操作するのかを決定し、クランプモードによってシフト幅が変化します。この処理はクランプ計算部の前に適用され、同じ画像が連続して描画されるのを防ぐことができます。

表 11-27. シフトモード

シフトモード	シフト計算	シフト幅
GL_NONE_DMP (デフォルト)	シフト計算なし	なし
GL_ODD_DMP	整数部が奇数から偶数へ変化する座標でシフト	GL_MIRRORED_REPEAT のみ 1.0、ほかは 0.5
GL_EVEN_DMP	整数部が偶数から奇数へ変化する座標でシフト	GL_MIRRORED_REPEAT のみ 1.0、ほかは 0.5

テクスチャ座標の u 成分と v 成分で別々のシフトモードを設定することができます。設定するには、予約ユニフォーム dmp_Texture[3].ptShiftU または dmp_Texture[3].ptShiftV に対して glUniform1i() を呼び出します。

図 11-12. シフトモードの違いとクランプモードによるシフト幅の違い



12. 予約フラグメントシェーダ

予約フラグメントシェーダでは、前段のシェーダから出力されたフラグメントに対して、ライティングなどの処理を行うことができます。

「5. シェーダプログラム」でも述べたように、予約フラグメントシェーダはバイナリからロードする必要はなく、頂点シェーダとジオメトリシェーダと同じプログラムオブジェクトに、特別な名前(GL_DMP_FRAGMENT_SHADER_DMP)をアタッチすることで使用できるようになります。予約フラグメントシェーダはフラグメント処理機能の集合です。その機能には以下のものがあります。

- フラグメントライティング
- シャドウ
- フォグ
- ガス
- その他(アルファテスト、w バッファ)

各フラグメント処理には予約ユニフォームが存在します。これらの予約ユニフォームには初期値が設定されており、アプリケーションは必要に応じて値を設定しなければなりません。

12.1. フラグメントオペレーションモード

予約フラグメント処理は、OpenGL 標準のフラグメント処理のパイプライン(アルファテスト以降)を独自の処理に切り替えることで、シャドウやガスといった特殊なレンダリングパスが必要な処理を行うことができます。この切り替えは、フラグメントオペレーションモードの予約ユニフォーム(dmp_FragOperation.mode)に、glUniform1i() で切り替えるモードを設定することで行われます。

表 12-1. フラグメントオペレーションモード

フラグメントオペレーションモード	切り替わるパイプライン
GL_FRAGOP_MODE_GL_DMP(デフォルト)	OpenGL 標準のフラグメント処理パイプライン(標準モード)
GL_FRAGOP_MODE_SHADOW_DMP	シャドウ累積パス用のフラグメント処理パイプライン(シャドウモード)
GL_FRAGOP_MODE_GAS_ACC_DMP	密度情報描画用のフラグメント処理パイプライン(ガスモード)

12.2. フラグメントライティング

3DS のライティングは頂点単位ではなく、フラグメント単位にプライマリカラーとセカンダリカラーを算出するフラグメントライティングです。また、テクスチャを参照して法線ベクトルを摂動する(ずらす)バンプ機能やシャドウテクスチャの生成とカラーの算出を行うシャドウ機能、スポットライトおよびライトとの距離による減衰を算出する機能を有しています。

プライマリカラーとセカンダリカラーの計算は、2 つのベクトルの内積から参照テーブルの値を出力する関数を複数組み合わせ、それらの出力値を積と和によって統合する(コンバイン)方法で決定される計算式に従って行われます。3DS では、これらの参照テーブルと内積値を計算するベクトルの組み合わせやコンバイン方法を任意に決定することはできませんが、プリセットされた設定から選択することができます。

ライティング計算式は視点座標系を基準に考えられているため、いくつかのベクトルには視点座標系での値を設定することが想定されています。必ずしも視点座標系でなければならないわけではありませんが、使用されるベクトルの座標系はすべて一致していなければなりません。

フラグメントライティングを使用するには、予約ユニフォーム dmp_FragmentLighting.enabled に glUniform1i()

で GL_TRUE を設定し、少なくとも 1 つ以上のライトを有効にしなければなりません。また頂点シェーダで法線、視線ベクトル、接線ベクトル(ライティングで考慮しなければならない場合)をクォータニオンに変換し、頂点属性の 1 つとして出力しなければなりません。

表 12-2. フラグメントライティングの有効・無効

予約ユニフォーム	種別	設定値
dmp_FragmentLighting.enabled	bool	GL_TRUE:ライティングを有効にする GL_FALSE:ライティングを無効にする(デフォルト)

12.2.1. クォータニオン変換

ライティングの演算で使用されるベクトルはすべて同じ座標系でなければなりません。つまり、後述するバンプマッピングなどでは、テクスチャから参照された摂動法線を surface-local 座標系(頂点を原点に法線が z 軸の正方向と一致する座標系)から視点座標系へと変換する必要があります。

下記の surface-local 座標系から視点座標系への変換を行う回転行列は、法線、接線、従法線で構成される 3x3 行列で、クォータニオン(Qx, Qy, Qz, Qw) に変換可能です。

$$\begin{bmatrix} Ex \\ Ey \\ Ez \end{bmatrix} = \begin{bmatrix} Tx & Bx & Nx \\ Ty & By & Ny \\ Tz & Bz & Nz \end{bmatrix} \begin{bmatrix} Sx \\ Sy \\ Sz \end{bmatrix}$$

(E は視点座標系での座標値、T は接線、N は法線、B は従法線、S は surface-local 座標系での座標値)

$$Qw = \frac{1}{2}\sqrt{1 + Tx + By + Nz}$$

$$Qx = \frac{1}{4Qw}(Bz - Ny)$$

$$Qy = \frac{1}{4Qw}(Nx - Tz)$$

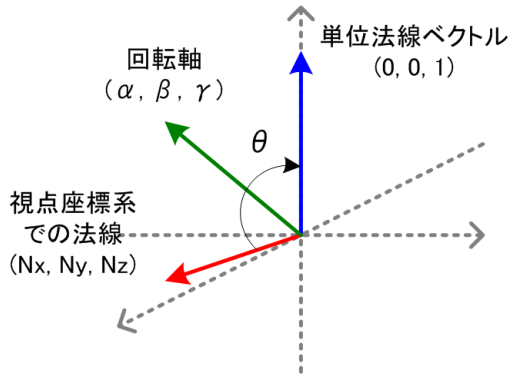
$$Qz = \frac{1}{4Qw}(Ty - Bx)$$

フラグメントライティングの実装では、頂点単位で入力される法線、接線、従法線(法線と接線から計算できる)の 3 ベクトルからフラグメント単位のベクトルを生成せず、頂点単位のクォータニオンからフラグメント単位のクォータニオンを生成します。クォータニオンはフラグメントライティングのプロセス中に元の回転行列に変換されて使用されます。そのため、フラグメントライティングを使用するには、頂点シェーダで各ベクトルを有効なクォータニオンに変換して出力する必要があります。

法線のみのクォータニオン生成について

CTR-SDK のサンプルデモには、法線のみの頂点情報からクォータニオンを生成する頂点シェーダ(l_position_view_quaternion)が収録されています。この頂点シェーダでは、surface-local 座標系での法線を視点座標系での法線に変換するクォータニオンとして、回転軸に単位法線ベクトル (0, 0, 1) と視点座標系に変換された法線 (Nx, Ny, Nz) の半角ベクトルを用いて 180 度回転させるクォータニオンを生成しています。

図 12-1. クォータニオンの回転軸と回転角



回転させる軸を (α, β, γ)、回転角を θ とした場合、求められるクォータニオン Q は以下のようになります。

$$Q = (\cos\frac{\theta}{2}; \alpha\sin\frac{\theta}{2}, \beta\sin\frac{\theta}{2}, \gamma\sin\frac{\theta}{2})$$

> **補足:**　頂点シェーダのコンポーネントでは、クォータニオンの実部を w 成分に設定します。

θ = 180° のため、

Q = (0; α, β, γ)

であり、また回転させる軸は半角ベクトルなので、

$$(\alpha,\beta,\gamma)$$
$$= \left(\frac{Nx+0}{\sqrt{(Nx+0)^2+(Ny+0)^2+(Nz+1)^2}}, \frac{Ny+0}{\sqrt{(Nx+0)^2+(Ny+0)^2+(Nz+1)^2}}, \frac{Nz+1}{\sqrt{(Nx+0)^2+(Ny+0)^2+(Nz+1)^2}} \right)$$
$$= \left(\frac{Nx}{\sqrt{Nx^2+Ny^2+Nz^2+2Nz+1}}, \frac{Ny}{\sqrt{Nx^2+Ny^2+Nz^2+2Nz+1}}, \frac{Nz+1}{\sqrt{Nx^2+Ny^2+Nz^2+2Nz+1}} \right)$$
$$= \left(\frac{Nx}{\sqrt{2(Nz+1)}}, \frac{Ny}{\sqrt{2(Nz+1)}}, \frac{\sqrt{Nz+1}}{\sqrt{2}} \right)$$

となります。

なお、(Nx, Ny, Nz) が (0, 0, −1) の場合のみ、半角ベクトルの向きが定まらないために (α, β, γ) = (1, 0, 0) として定めています。

12.2.2. ライティングの概要

3DS のフラグメントライティングでは、必ずプライマリカラーとセカンダリカラーの 2 つのカラーが算出されます。

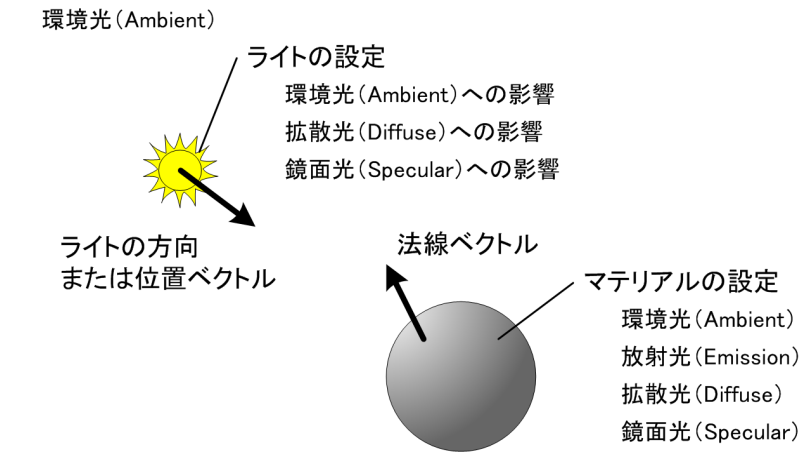
プライマリカラーは、ライトがフラグメントの環境光 (Ambient) と拡散光 (Diffuse) に与える影響にシャドウ、スポットライト減衰および距離減衰を掛け合わせたものをライトごとに計算して累積し、シーンから受ける環境光の影響およびフラグメントの放射光 (Emission) を加算したものです。これはフラグメントの基本となる色となります。

セカンダリカラーは、ライトがフラグメントの持つ 2 項の鏡面光 (Specular) に与える影響にシャドウ、スポットライト減衰および距離減衰を掛け合わせたものをライトごとに計算して累積したものです。これは主に、フラグメントのハイライト部分の色となります。

環境光、拡散光、放射光、鏡面光をもとにオブジェクトの色を決定するのは OpenGL と同様ですが、ライティングの計算がフ

ラグメント単位で行われる点と、2 つの鏡面光を算出する点、鏡面光の算出方法の多様性に違いがあります。特に 2 つある鏡面光は、角度によって色調が変化する物質の表現などに利用することができます。

図 12-2. フラグメントライティング



12.2.3. シーン設定

フラグメントライティングで扱うことのできるシーンの大きさは -2^{16} ～ 2^{15} の範囲です。シーンを構成する各フラグメントと視点との距離やライトと視点との距離が 2^{16} 以上にならないように注意してください。

シーンがフラグメントに与える影響はシーンの環境光(Global Ambient)です。シーンの環境光の指定は、予約ユニフォーム(`dmp_FragmentLighting.ambient`)に `glUniform4fv()` で RGBA カラーを設定することで行います。

表 12-3. シーン設定の予約ユニフォーム

予約ユニフォーム	種別	設定値
`dmp_FragmentLighting.ambient`	`vec4`	シーンの環境光 (R, G, B, A) を指定する。 各成分とも 0.0 ～ 1.0 (デフォルト)(0.2, 0.2, 0.2, 1.0)

12.2.4. マテリアル設定

マテリアル設定は簡単に言えば、フラグメントを物質(マテリアル)として見たときの、その素材や質感を環境光や鏡面光などの色情報で表現したものです。マテリアルに関係する設定は予約ユニフォーム(`dmp_FragmentMaterial.*`)で指定します。

設定がどのようにライティング計算に使用されるのかは、「プライマリカラーの計算式」および「セカンダリカラーの計算式」で説明します。

表 12-4. マテリアル設定の予約ユニフォーム

予約ユニフォーム	種別	設定値
`dmp_FragmentMaterial.ambient`	`vec4`	環境光(R, G, B, A)を指定する。各成分とも 0.0 ～ 1.0 (デフォルト)(0.2, 0.2, 0.2, 1.0)

dmp_FragmentMaterial.diffuse	vec4	拡散光(R, G, B, A)を指定する。各成分とも 0.0 ～ 1.0 (デフォルト)(0.8, 0.8, 0.8, 1.0)
dmp_FragmentMaterial.emission	vec4	放射光(R, G, B, A)を指定する。各成分とも 0.0 ～ 1.0 (デフォルト)(0.0, 0.0, 0.0, 1.0)
dmp_FragmentMaterial.specular0	vec4	鏡面光 0(R, G, B, A)を指定する。各成分とも 0.0 ～ 1.0 (デフォルト)(0.0, 0.0, 0.0, 1.0)
dmp_FragmentMaterial.specular1	vec4	鏡面光 1(R, G, B, A)を指定する。各成分とも 0.0 ～ 1.0 (デフォルト)(0.0, 0.0, 0.0, 1.0)
dmp_FragmentMaterial.samplerXX (XX=DO,D1,RR,RG,RB,FR)	int	ライティング計算で使用される参照テーブル番号を指定する。 各ファクタとも 0 ～ 31

12.2.5. ライト設定

ライト設定には、ライトがマテリアルに与える影響の設定とライト自身の設定の 2 種類が存在します。フラグメントライティングで扱うことのできるライトの数は 8 個です。ライトに関係する設定は予約ユニフォーム(dmp_FragmentLightSource[*i*].* 、*i* はライト番号 0 ～ 7)で指定します。

設定がどのようにライティング計算に使用されるのかは、「プライマリカラーの計算式」および「セカンダリカラーの計算式」で説明します。

表 12-5. ライト設定の予約ユニフォーム

予約ユニフォーム	種別	設定値
*.enabled	bool	ライトの有効・無効を指定する。 GL_TRUE:ライトを有効にする GL_FALSE:ライトを無効にする(デフォルト)
*.ambient	vec4	環境光 (R, G, B, A) を指定する。各成分とも 0.0 ～ 1.0 (デフォルト)(0.0, 0.0, 0.0, 0.0)
*.diffuse	vec4	拡散光 (R, G, B, A) を指定する。各成分とも 0.0 ～ 1.0 (デフォルト)ライト番号 0 のみ (1.0, 1.0, 1.0, 1.0) その他 (0.0, 0.0, 0.0, 0.0)
*.specular0	vec4	鏡面光 0(R, G, B, A)を指定する。各成分とも 0.0 ～ 1.0 (デフォルト)ライト番号 0 のみ (1.0, 1.0, 1.0, 1.0) その他 (0.0, 0.0, 0.0, 0.0)
*.specular1	vec4	鏡面光 1(R, G, B, A)を指定する。各成分とも 0.0 ～ 1.0 (デフォルト)(0.0, 0.0, 0.0, 0.0)
*.position	vec4	ライトの光源位置 (x, y, z, w) を指定する。 ベクトルは正規化されていなくてもかまいません。w 成分で平行光源(0.0)と点光源の判別が行われます。 (デフォルト)(0.0, 0.0, 1.0, 0.0)
*.spotDirection	vec3	スポットライトの向き (x, y, z) を指定する。 ベクトルは正規化されていなくてもかまいません。 (デフォルト)(0.0, 0.0, -1.0)
*.shadowed	bool	ライトがシャドウの影響を受けるかどうかを指定する。 GL_TRUE:シャドウの影響を受ける GL_FALSE:シャドウの影響を受けない(デフォルト)

*.geomFactor0	bool	ジオメトリファクタ 0 をライティング計算に使用するかどうかを指定する。 GL_TRUE:ジオメトリファクタ 0 を使用する GL_FALSE:ジオメトリファクタ 0 を使用しない(デフォルト)
*.geomFactor1	bool	ジオメトリファクタ 1 をライティング計算に使用するかどうかを指定する。 GL_TRUE:ジオメトリファクタ 1 を使用する GL_FALSE:ジオメトリファクタ 1 を使用しない(デフォルト)
*.twoSideDiffuse	bool	両面ライティングを行うかどうかを指定する。 GL_TRUE:両面ライティングを行う GL_FALSE:両面ライティングを行わない(デフォルト)
*.spotEnabled	bool	スポットライトの光量分布による減衰を行うかどうかを指定する。 GL_TRUE:スポットライト減衰を適用する GL_FALSE:スポットライト減衰を適用しない(デフォルト)
*.distanceAttenuationEnabled	bool	ライトとの距離による減衰を行うかどうかを指定する。 GL_TRUE:距離減衰を適用する GL_FALSE:距離減衰を適用しない(デフォルト)
*.distanceAttenuationBias	float	距離減衰のバイアス値を指定する。 (デフォルト)0.0
*.distanceAttenuationScale	float	距離減衰のスケール値を指定する。 (デフォルト)1.0
*.samplerXX (XX=SP,DA)	int	スポットライト減衰および距離減衰の計算で使用される参照テーブル番号を指定する。 各ファクタとも 0 ～ 31

表中の "*" は "dmp_FragmentLightSource[*i*]" の略です。*i* はライト番号 0 ～ 7 です。

12.2.6. ライティング環境

シャドウテクスチャの選択やバンプ設定、参照テーブルの入力値など、ライティング全般に関係する設定は予約ユニフォーム(dmp_LightEnv.*)で指定します。

設定がどのようにライティング計算に使用されるのかは、「プライマリカラーの計算式」および「セカンダリカラーの計算式」で説明します。

表 12-6. ライティング環境の予約ユニフォーム

予約ユニフォーム	種別	設定値
*.absLutInputXX (XX=D0,D1,RR,RG,RB,FR,SP)	bool	各ファクタの参照テーブルへの入力値を絶対値化するかどうかを指定する。 GL_TRUE:絶対値化する GL_FALSE:絶対値化しない(デフォルト)

*.lutInputXX (XX=D0,D1,RR,RG,RB,FR,SP)	int	各ファクタの参照テーブルへの入力値に使用する、2 ベクトル間の角度のコサイン値を指定する。 GL_LIGHT_ENV_NH_DMP:法線とハーフベクトル(デフォルト) GL_LIGHT_ENV_VH_DMP:ビューベクトルとハーフベクトル GL_LIGHT_ENV_NV_DMP:法線とビューベクトル GL_LIGHT_ENV_LN_DMP:ライトベクトルと法線 GL_LIGHT_ENV_SP_DMP:ライトベクトルとスポットライトの方向 GL_LIGHT_ENV_CP_DMP:ハーフベクトルの接平面への投影と接線
*.lutScaleXX (XX=D0,D1,RR,RG,RB,FR,SP)	float	各ファクタの参照テーブルの出力値に対するスケール値を指定する。参照テーブルから出力される値は、スケール値を適用したあとに -2.0 ～ +2.0 の範囲にクランプされます。 0.25 0.5 1.0(デフォルト) 2.0 4.0 8.0
*.shadowSelector	int	シャドウとして使用するテクスチャユニットを指定する。 GL_TEXTURE0(デフォルト) GL_TEXTURE1 GL_TEXTURE2 GL_TEXTURE3
*.bumpSelector	int	バンプマップとして使用するテクスチャユニットを指定する。 GL_TEXTURE0(デフォルト) GL_TEXTURE1 GL_TEXTURE2 GL_TEXTURE3
*.bumpMode	int	法線または接線の摂動モードを指定する。 GL_LIGHT_ENV_BUMP_NOT_USED_DMP:なし(デフォルト) GL_LIGHT_ENV_BUMP_AS_BUMP_DMP:法線を摂動する GL_LIGHT_ENV_BUMP_AS_TANG_DMP:接線を摂動する
*.bumpRenorm	bool	法線の第3成分を再生成するかどうかを指定する。 GL_TRUE:再生成する GL_FALSE:再生成しない(デフォルト)
*.config	int	各ファクタのコンフィグレーションを指定する。 GL_LIGHT_ENV_LAYER_CONFIG0_DMP(デフォルト) GL_LIGHT_ENV_LAYER_CONFIG1_DMP GL_LIGHT_ENV_LAYER_CONFIG2_DMP GL_LIGHT_ENV_LAYER_CONFIG3_DMP GL_LIGHT_ENV_LAYER_CONFIG4_DMP GL_LIGHT_ENV_LAYER_CONFIG5_DMP GL_LIGHT_ENV_LAYER_CONFIG6_DMP GL_LIGHT_ENV_LAYER_CONFIG7_DMP
*.invertShadow	bool	シャドウの項を反転(1.0 - shadow)させるかどうかを指定する。 GL_TRUE:反転させる GL_FALSE:反転させない(デフォルト)
*.shadowPrimary	bool	プライマリカラーにシャドウを影響させるかどうかを指定する。 GL_TRUE:影響させる GL_FALSE:影響させない(デフォルト)

*.shadowSecondary	bool	セカンダリカラーにシャドウを影響させるかどうかを指定する。 GL_TRUE:影響させる GL_FALSE:影響させない(デフォルト)
*.shadowAlpha	bool	アルファ成分にシャドウを影響させるかどうかを指定する。 GL_TRUE:影響させる GL_FALSE:影響させない(デフォルト)
*.fresnelSelector	int	フレネルファクタの出力モードを指定する。 GL_LIGHT_ENV_NO_FRESNEL_DMP(デフォルト) GL_LIGHT_ENV_PRI_ALPHA_FRESNEL_DMP GL_LIGHT_ENV_SEC_ALPHA_FRESNEL_DMP GL_LIGHT_ENV_PRI_SEC_ALPHA_FRESNEL_DMP
*.clampHighlights	bool	鏡面光の出力値をクランプするかどうかを指定する。 GL_TRUE:クランプする GL_FALSE:クランプしない(デフォルト)
*.lutEnabledD0	bool	分布 0(D0)に対する参照テーブルからの出力値を適用するかどうかを指定する。 GL_TRUE:適用する GL_FALSE:適用しない(デフォルト)
*.lutEnabledD1	bool	分布 1(D1)に対する参照テーブルからの出力値を適用するかどうかを指定する。 GL_TRUE:適用する GL_FALSE:適用しない(デフォルト)
*.lutEnabledRefl	bool	反射(RR、RG、RB)に対する参照テーブルからの出力値を適用するかどうかを指定する。 GL_TRUE:適用する GL_FALSE:適用しない(デフォルト)

表中の "*" は "dmp_LightEnv" の略です。

12.2.7. プライマリカラーの計算式

プライマリカラーの計算式の概略は以下の数式で表すことができます。

$$Color_{primary} = \Sigma((Diffuse \times DP_{LN} \times Shadow + Ambient) \times Spot \times DistAtt + Ambient_{global} + Emission$$

各項の計算を詳細に解説します。

拡散光と環境光にはマテリアルとライトのカラー成分を掛け合わせた値が適用されます。

$$Diffuse = Diffuse_{material} \times Diffuse_{light}$$
$$Ambient = Ambient_{material} \times Ambient_{light}$$

拡散光はライトの当たる角度とシャドウによる影響を受けます。

$$DP_{LN} = \max\{ 0, L \cdot N \} \ or \ \mathrm{abs}(L \cdot N)$$

ライトの当たる角度による影響は式中の DP_{LN} で表され、その値は正規化されたライトベクトルと法線の内積値ですが、両面ライティングの設定(dmp_FragmentLightSource[*i*].twoSideDiffuse)で単面ライティング(GL_FALSE)を指定している場合は 0 と内積値との最大値が、両面ライティング(GL_TRUE)を指定している場合は内積値の絶対値が適用されます。

シャドウ減衰の影響は式中の *Shadow* で表されます。dmp_LightEnv.shadowPrimary または dmp_FragmentLightSource[*i*].shadowed のどちらかに GL_FALSE が指定されている場合はシャドウの影響を受けないため 1.0 が適用されます。どちらの予約ユニフォームにも GL_TRUE が指定されている場合は、dmp_LightEnv.shadowSelector で指定されたテクスチャユニットからサンプルされた値が適用されます。このとき、

dmp_LightEnv.invertShadow に GL_TRUE が指定されている場合は、"1.0 –(サンプルされた値)" が適用されます。シャドウテクスチャ以外がバインドされたテクスチャユニットを指定した場合、サンプルされた値にはカラー成分がそのまま適用されます。

スポットライトの影響は式中の *Spot* で表されます。ライトごとに参照テーブルの設定および使用・不使用の設定が可能です。スポットライトの使用・不使用は dmp_FragmentLightSource[*i*].spotEnabled で、スポットライトで使用する参照テーブルは dmp_FragmentLightSource[*i*].samplerSP で設定します。スポットライトの方向ベクトルは dmp_FragmentLightSource[*i*].spotDirection で設定します。通常は、スポットライトの参照テーブルへの入力値(dmp_LightEnv.lutInputSP)には GL_LIGHT_ENV_SP_DMP を指定することになります。

距離減衰の影響は式中の *DistAtt* で表されます。距離減衰の影響は平行光源では考慮されません。

距離減衰の適用は dmp_FragmentLightSource[*i*].distanceAttenuationEnabled で設定し、ライトごとに制御することができます。ただし、dmp_LightEnv.config に GL_LIGHT_ENV_LAYER_CONFIG7_DMP を設定している場合は距離減衰の影響が無効(1.0 を適用)となります。

使用する参照テーブルは dmp_FragmentLightSource[*i*].samplerDA で設定し、参照テーブルへの入力値にはスケール値(dmp_FragmentLightSource[*i*].distanceAttenuationScale)とバイアス(dmp_FragmentLightSource[*i*].distanceAttenuationBias)の設定値が考慮されます。

ここまでがフラグメントのプライマリカラーのライトごとの影響分です。これを有効なライトの数だけ算出し、ライトの影響を受けないグローバル環境光とマテリアルの放出光を加算して、フラグメントの最終的なプライマリカラーを算出します。

グローバル環境光には、マテリアルの環境光とシーン設定の環境光を掛け合わせた値が適用されます。

$$Ambient_{global} = Ambient_{material} \times Ambient_{scene}$$

ただし、ライト番号 0 の光源が無効となっている(dmp_FragmentLightSource[0].enabled に GL_FALSE が設定されている)場合は、グローバル環境光とマテリアルの放出光には 0.0 が適用されます。

12.2.8. セカンダリカラーの計算式

セカンダリカラーの計算式の概略は以下の数式で表すことができます。

$$Color_{secondary} = \Sigma((Specular0 + Specular1) \times f \times Shadow \times Spot \times DistAtt)$$

各項の計算を詳細に解説します。

dmp_LightEnv.shadowPrimary の代わりに dmp_LightEnv.shadowSecondary の設定が考慮されること以外は、*Shadow*、*Spot*、*DistAtt* の各項はプライマリカラーと同じ計算式で算出されます。

Specular0 と *Specular1* はそれぞれ、以下のように計算されます。

$$Specular0 = Specular0_{material} \times Specular0_{light} \times Distribution0 \times Geometry0$$
$$Specular1 = Reflection_{RGB} \times Specular1_{light} \times Distribution1 \times Geometry1$$

基本的に、マテリアルとライトの鏡面光に分布関数とジオメトリファクタを掛け合わせた値が鏡面光として算出されます。設定によってはマテリアルの鏡面光 1 にあたる項には、カラーごとに異なる反射の参照テーブルの出力値を適用することもできます。反射と分布の参照テーブルを調整することで、様々な質感を持ったフラグメントの表現が可能になります。

分布関数(ディストリビューションファクタ)は式中の *Distribution0* と *Distribution1* で表されます。分布関数は分布 0(D0)と分布 1(D1)の参照テーブルで設定することになります。

使用・不使用の制御は dmp_LightEnv.lutEnabledD0(lutEnabledD1)で行います。使用する参照テーブルのテーブル番号は dmp_FragmentLightSource[*i*].samplerD0(samplerD1)で指定します。参照テーブルへの入力値

は dmp_LightEnv.lutInputD0(lutInputD1)で指定し、入力値の絶対値化を dmp_LightEnv.absLutInputD0(absLutInputD1)で指定します。参照テーブルからの出力値には dmp_LightEnv.lutScaleD0(lutScaleD1)で指定されたスケール値が考慮されます。

反射は式中の $Reflection_{RGB}$ で表され、マテリアルの鏡面光 1 の代わりに RGB の各成分の反射を算出する関数を参照テーブル(RR、RG、RB)で設定します。使用・不使用の制御は dmp_LightEnv.lutEnabledRefl で行い、GL_FALSE を指定した場合はマテリアルの鏡面光 1 が適用されます。各成分で使用する参照テーブルのテーブル番号は dmp_FragmentLightSource[*i*].samplerRR(samplerRG、samplerRB)で指定します。参照テーブルへの入力値は dmp_LightEnv.lutInputRR(lutInputRG、lutInputRB)で指定し、入力値の絶対値化を dmp_LightEnv.absLutInputRR(absLutInputRG、absLutInputRB)で指定します。

ジオメトリファクタは式中の *Geometry0* と *Geometry1* で表され、これらは Cook-Torrance の照明モデルで使用される項です。使用・不使用の制御は dmp_FragmentLightSource[*i*].geomFactor0(geomFactor1)で行います。GL_FALSE を指定した場合は 1.0 が適用され、GL_TRUE を指定した場合は Cook-Torrance の照明モデルで使用するジオメトリファクタに近似された値が適用されます。

f は正規化されたライトベクトルと法線の内積値を用いて、フラグメントへのライティングが有効か無効かを判定する関数です。dmp_LightEnv.clampHighlights に GL_FALSE を指定した場合は必ず 1.0 が適用され、通常は光の通過しない領域まで光を通過させて材質が半透明であるオブジェクトを表現するときに使用します。GL_TRUE を指定した場合は内積値が 0.0 以下ならば 0.0、0.0 以上ならば 1.0 が適用されます。この場合、ライティングされない箇所には鏡面光が存在しない状態となります。

フラグメントの最終的なセカンダリカラーは、フラグメントのセカンダリカラーのライトごとの影響分を有効なライトの数だけ算出したものになります。

12.2.9. アルファ成分のライティング

ここまでで説明したプライマリカラーとセカンダリカラーの計算式では、アルファ成分が 1.0 に固定されて算出されていました。フラグメントライティングでは、フレネルファクタとシャドウを使ったアルファ成分のライティングを行うことができます。

フレネルファクタの本来の目的は、半透明物体のフレネル反射の参照テーブル(FR)として使用されることなのですが、アルファ成分を参照テーブルの出力値に置き換える機能として、ほかの目的に利用することもできます。

フレネルファクタで使用する参照テーブルのテーブル番号は dmp_FragmentLightSource[*i*].samplerFR で指定します。参照テーブルへの入力値は dmp_LightEnv.lutInputFR で指定し、入力値の絶対値化を dmp_LightEnv.absLutInputFR で指定します。複数のライトが有効になっている場合は、その中で最も大きなライト番号のライトのライトベクトルとの内積値が参照テーブルへの入力値に使用されます。フレネルファクタの適用範囲の制御は dmp_LightEnv.fresnelSelector で行い、以下の設定値から選択します。

表 12-7. フレネルファクタの適用範囲

設定値	適用の範囲
GL_LIGHT_ENV_NO_FRESNEL_DMP(デフォルト)	適用しない(アルファ成分は 1.0 固定)
GL_LIGHT_ENV_PRI_ALPHA_FRESNEL_DMP	プライマリカラーのアルファ成分のみ
GL_LIGHT_ENV_SEC_ALPHA_FRESNEL_DMP	セカンダリカラーのアルファ成分のみ
GL_LIGHT_ENV_PRI_SEC_ALPHA_FRESNEL_DMP	プライマリカラーとセカンダリカラーのアルファ成分

フレネルファクタの適用範囲はシャドウのアルファ成分への影響にも適用されます。シャドウのアルファ成分への影響を制御

する dmp_LightEnv.shadowAlpha に GL_TRUE を指定した場合は、適用されるアルファ成分にシャドウのアルファ成分が乗算されます。カラーと同じように、dmp_LightEnv.invertShadow に GL_TRUE が指定されている場合は、乗算される値に "1.0 –(シャドウのアルファ値)" が適用されます。

12.2.10. 参照テーブルの作成と指定

ライティングの計算式で使用される参照テーブルには、以下の 8 種類の参照テーブルが存在します。

- 反射(RR、RG、RB の 3 種類)
- ディストリビューションファクタ(D0、D1 の 2 種類)
- フレネルファクタ(FR)
- スポットライト(SP)
- ライトの距離減衰(DA)

反射(RR、RG、RB)、ディストリビューションファクタ(D0、D1)、フレネルファクタ(FR)の参照テーブルはマテリアル設定のため、すべてのライトで共通のテーブルが参照されることになります。スポットライト(SP)とライトの距離減衰(DA)については、ライト単位で異なる参照テーブルを設定することができます。

ライトの距離減衰を除いて、セカンダリカラーのライティング計算式の各項に、どの参照テーブルが使用されるかはレイヤコンフィグレーション(dmp_LightEnv.config)の設定によって制御することができます。

表 12-8. レイヤコンフィグレーションの設定値と参照テーブル、サイクル数の対応

レイヤコンフィグレーションの設定値	Rr	Rg	Rb	D0	D1	Fr	Sp	サイクル数
GL_LIGHT_ENV_LAYER_CONFIG0_DMP	RR	RR	RR	D0	–	–	SP	1
GL_LIGHT_ENV_LAYER_CONFIG1_DMP	RR	RR	RR	–	–	FR	SP	1
GL_LIGHT_ENV_LAYER_CONFIG2_DMP	RR	RR	RR	D0	D1	–	–	1
GL_LIGHT_ENV_LAYER_CONFIG3_DMP	–	–	–	D0	D1	FR	–	1
GL_LIGHT_ENV_LAYER_CONFIG4_DMP	RR	RG	RB	D0	D1	–	SP	2
GL_LIGHT_ENV_LAYER_CONFIG5_DMP	RR	RG	RB	D0	–	FR	SP	2
GL_LIGHT_ENV_LAYER_CONFIG6_DMP	RR	RR	RR	D0	D1	FR	SP	2
GL_LIGHT_ENV_LAYER_CONFIG7_DMP	RR	RG	RB	D0	D1	FR	SP	4

上表は、反射の RGB 成分、ディストリビューションファクタ、フレネルファクタ、スポットライトの各項が、どの参照テーブルから値を取得するかを示しています。対応表で「–」となっている項は、ライティングの計算式では 1.0 が適用されます。つまり、計算式から該当の項が消えたことになります。サイクル数はライティング計算に必要なハードウェアのサイクル数を示しています。ライティング計算を高速に行いたい場合は、この数値がなるべく小さなレイヤコンフィグレーションを選択することをお勧めします。

注意: カラーバッファへの書き込みアクセスのみが行われる設定(glColorMask() でカラーバッファの全成分に GL_TRUE を設定し、GL_BLEND および GL_COLOR_LOGIC_OP を glDisable() で無効にしている状態)では、GL_LIGHT_ENV_LAYER_CONFIG4_DMP ～ GL_LIGHT_ENV_LAYER_CONFIG6_DMP のレイヤコンフィグレーションで 1 ピクセルを処理するためには 2 サイクルではなく、3 サイクル必要です。

GL_LIGHT_ENV_LAYER_CONFIG7_DMP を設定した場合、プライマリ・セカンダリカラーの計算で距離減衰の影響が無効となります。

例えば GL_LIGHT_ENV_LAYER_CONFIG0_DMP を設定した場合、反射の RGB 成分はすべて参照テーブル RR から、分布 0 は D0 から、スポットライトは SP から、それぞれ値を取得します。そして、分布 1 とフレネルファクタには固定値の 1.0 が適用されます。

参照テーブルは「7.7. 参照テーブルのロード」で説明したように、glTexImage1D() で用意します。フラグメントライティングで使用する参照テーブルの *width* は 512 固定です。512 要素のうち、前半の 256 要素には参照テーブルのサンプリング値を格納し、後半の 256 要素には個々のサンプリング値の差分を格納します。

サンプリング値の格納順は参照テーブルへの入力値の範囲が「0.0 ～ 1.0」か「-1.0 ～ 1.0」かで異なります。入力値の範囲は、予約ユニフォーム dmp_LightEnv.absLutInputXX(XX=D0,D1,RR,RG,RB,FR,SP) の指定で変化し、GL_TRUE を指定した場合は「0.0 ～ 1.0」、GL_FALSE を指定した場合は「-1.0 ～ 1.0」となります。

ここではサンプリング値の取得手順を先に説明してから、それぞれの場合の格納手順を説明します。

入力値の範囲が「0.0 ～ 1.0」の場合、入力値に 256 を乗算して 255 にクランプした値の整数部分が値を取得する際のインデックスとなります。そのインデックスで参照テーブルからサンプリング値を取得し、小数部とインデックス + 256 の位置の差分値を乗算したものを加算して最終的なサンプリング値を求めます。

コード 12-1. 入力値の範囲が 0.0 ～ 1.0 のときのサンプリング値の取得手順（疑似コード）

```
index=min(floor(input * 256), 255);
samplingValue=LUT[index] + LUT[index + 256] * (input * 256 - index);
```

入力値の範囲が「-1.0 ～ 1.0」の場合、入力に 128 を乗算して整数部分を 2 の補数に変換した値がインデックスとなります。そのインデックスで参照テーブルからサンプリング値を取得し、小数部とインデックス + 256 の位置の差分値を乗算したものを加算して最終的なサンプリング値を求めます。

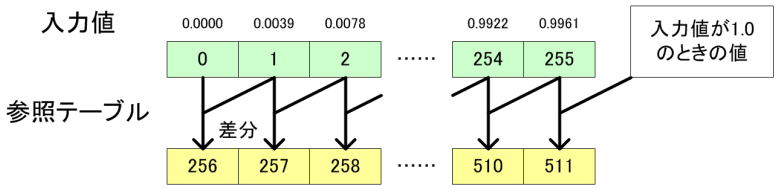
コード 12-2. 入力値の範囲が -1.0 ～ 1.0 のときのサンプリング値の取得手順（疑似コード）

```
if (input < 0.0) {
    flooredInput=floor(input * 128);
    index = 255 + flooredInput;
    samplingValue=LUT[index] + LUT[index + 256] * (input * 128 - flooredInput);
} else {
    index=min(floor(input * 128), 127);
    samplingValue=LUT[index] + LUT[index + 256] * (input * 128 - index);
}
```

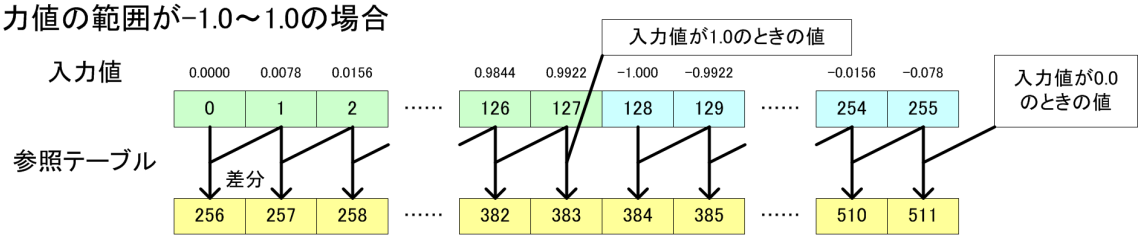
サンプリング値の格納順を図 12-3 に示します。

図 12-3. 参照テーブルのサンプリング値の格納順

入力値の範囲が0.0～1.0の場合



入力値の範囲が−1.0～1.0の場合



基本的に参照テーブルの作成は、インデックスを 256 または 128 で除算した値で計算したサンプリング値を格納し、次のサンプリング値との差分をインデックス + 256 の位置に格納することで行います。範囲が −1.0 ～ 1.0 のときは、参照テーブルが不連続になることに注意してください。

入力値に対するサンプリング値を計算する関数を `func` としたときのコード例をそれぞれの場合で示します。

コード 12-3. 入力値の範囲が 0.0 ～ 1.0 のときの参照テーブルの作成例

```
for (i = 0; i < 256; i++) LUT[i] = func((float) i / 256.0f);
for (i = 0; i < 255; i++) LUT[i + 256] = LUT[i + 1] - LUT[i];
LUT[511] = func(1.0f) - LUT[255] * 16.0f / 15.0f;
```

差分の最終要素で入力値が 1.0 の値との差分を格納する際に(16.0/15.0)を乗算しているのは、GPU の実装で小数部の精度が 4 ビットであるため、そのままの差分を格納すると入力の最大値で 1.0 でのサンプリング値にならないからです。

コード 12-4. 入力値の範囲が −1.0 ～ 1.0 のときの参照テーブルの作成例

```
for (i = 0; i < 128; i++) {
    LUT[i] = func((float) i / 128.0f);
    LUT[255 - i] = func((float) (i + 1) * -1.0f / 128.0f);
}
for (i= 0; i < 127; i++) {
    LUT[i + 256] = LUT[i + 1] - LUT[i];
    LUT[i + 384] = LUT[i + 129] - LUT[i + 128];
}
LUT[383] = func(1.0f) - LUT[127] * 16.0f / 15.0f;
LUT[511] = LUT[0] - LUT[255];
```

こちらの場合も、そのまま差分を格納すると入力の最大値で 1.0 でのサンプリング値にならないために(16.0/15.0)を乗算しています。

作成した参照テーブルを `glTexImage1D()` でロードします。ロード先の参照テーブルは `glBindTexture()` の *`target`* に `GL_LUT_TEXTURE`*`i`*`_DMP` で指定します。ライティング計算時に参照テーブルを参照させるには、`glUniform1i()` で参照テーブル番号を予約ユニフォームに指定します。ここで指定する参照テーブル番号は `GL_LUT_TEXTURE`*`i`*`_DMP` の *`i`*(0 ～ 31) が表す数値で、テクスチャの名前(ID)や `GL_LUT_TEXTURE`*`i`*`_DMP` を直接指定するわけではないことに注意してください。参照テーブル番号を指定する予約ユニフォームは、反射、ディストリビューショ

ンファクタ、フレネルファクタならばマテリアル設定 dmp_FragmentMaterial.samplerXX(XX=DO,D1,RR,RG,RB,FR) 、スポットライトとライトの距離減衰ならばライト設定 dmp_FragmentLightSource[*i*].samplerXX(XX=SP,DA) になります。

コード 12-5. 反射(R 成分)の参照テーブルへの指定例

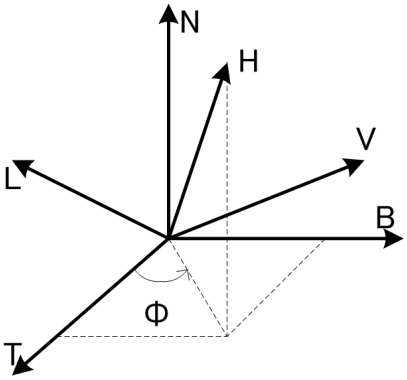
```
glBindTexture(GL_LUT_TEXTURE2_DMP, lutTextureID);
glTexImage1D(GL_LUT_TEXTURE2_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
             GL_LUMINANCEF_DMP, GL_FLOAT, LUT);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerRR"), 2);
```

12.2.11. 参照テーブルへの入力値

ライトの距離減衰(DA)を除く項(D0、D1、RR、RG、RB、FR、SP)はすべて、2 つのベクトルがなす角のコサイン値(正規化されたベクトル同士の内積値)が入力値となります。

対象となるベクトルは法線(N)、ライトベクトル(L)、視線ベクトル(V)、ハーフベクトル(H)、接線(T)、従法線(B)、スポットライトの方向ベクトル、ハーフベクトルの接平面への投影です。

図 12-4. 参照テーブルへの入力対象となるベクトル



各ファクタの参照テーブルへの入力値に使用する値の指定は、ライティング環境の予約ユニフォーム dmp_LightEnv.lutInputXX(XX=D0,D1,RR,RG,RB,FR,SP) に以下の値を glUniform1i() で設定することで行います。各ファクタの参照テーブルへの入力値には、指定の 2 ベクトル間の角度のコサイン値が適用されます。

表 12-9. 参照テーブルへの入力値の指定

設定値	対象となる 2 つのベクトル
GL_LIGHT_ENV_NH_DMP	法線とハーフベクトル(デフォルト)
GL_LIGHT_ENV_VH_DMP	ビューベクトルとハーフベクトル
GL_LIGHT_ENV_NV_DMP	法線とビューベクトル
GL_LIGHT_ENV_LN_DMP	ライトベクトルと法線
GL_LIGHT_ENV_SP_DMP	ライトベクトルの逆ベクトルとスポットライトの方向ベクトル(RR, RG, RB, FR は不可)
GL_LIGHT_ENV_CP_DMP	ハーフベクトルの接平面への投影と接線(RR, RG, RB, FR は不可)

12.2.12. ライトの距離減衰項

ライトの距離減衰項への入力値は以下の式で計算されます。

$$Input_{DistAtt} = Scale \times \sqrt{\left(f_{position} - l_{position}\right)^2} + Bias$$

$f_{position}$ はフラグメントの位置、$l_{position}$ はライトの位置で、どちらも視点座標系での値を想定しています。ライトの位置は点光源でのみ意味のある値をとり、平行光源では意味をなしません。

この数式から、ライトの距離減衰項への入力値はフラグメントとライト間の距離にスケール値を乗算し、さらにバイアス値を加算したものであるとわかります。

参照テーブルは入力値の範囲が 0.0 ～ 1.0(絶対値)であるものとして作成してください。スケール値は予約ユニフォーム `dmp_FragmentLightSource[i].distanceAttenuationScale` に、バイアス値は予約ユニフォーム `dmp_FragmentLightSource[i].distanceAttenuationBias` に、それぞれ `glUniform1f()` で値を設定してください。

レイヤコンフィグレーションの設定値に `GL_LIGHT_ENV_LAYER_CONFIG7_DMP` を指定した場合は距離減衰を無効にしなければなりません。`dmp_FragmentLightSource[i].distanceAttenuationEnabled` に `GL_FALSE` を設定してください。

12.2.13. テクスチャコンバイナの設定

フラグメントライティングで算出したプライマリカラーとセカンダリカラーは、テクスチャコンバイナの入力ソースの 1 つとして使用することができます。

入力ソースの予約ユニフォーム(`dmp_TexEnv[i].srcRgb` と `dmp_TexEnv[i].srcAlpha`)に、プライマリカラーは `GL_FRAGMENT_PRIMARY_COLOR_DMP`、セカンダリカラーは `GL_FRAGMENT_SECONDARY_COLOR_DMP` を設定することで使用可能です。

ライティングが無効(`dmp_FragmentLighting.enabled` に `GL_FALSE` を設定)になっている場合の出力は (0.0, 0.0, 0.0, 1.0) が出力されます。

12.3. バンプマッピング

バンプマッピングはフラグメントライティングの機能の 1 つで、テクスチャで入力される法線マップによってフラグメントの法線や接線を摂動させる(ずらす)機能です。バンプマッピングはオブジェクトの表面に凹凸による陰影が付いたような表現を施すことができ、少ないポリゴンで構成された簡単なモデルから、見かけの複雑なモデルを描画させることができます。

12.3.1. 予約ユニフォーム

バンプマッピングの予約ユニフォームには、以下のものが存在します。

法線マップ

法線マップ(`dmp_LightEnv.bumpSelector`)には、バンプマッピングで使用する法線マップのテクスチャがバインドされたテクスチャユニットを指定してください。法線マップのテクスチャは、摂動ベクトルの x、y、z 成分をそれぞれ R、G、B 成分にエンコードして作成します。エンコード方法とは、ベクトルの -1.0 を最小輝度(8 ビットフォーマットならば 0)に、1.0 を最大輝度(8 ビットフォーマットならば 255)に変換することです。

摂動方法

摂動方法(dmp_LightEnv.bumpMode)に GL_LIGHT_ENV_BUMP_NOT_USED_BUMP 以外を指定することで、バンプマッピングが有効になります。摂動方法には以下のものがあります。

表 12-10. 摂動方法

摂動方法	摂動されるベクトル
GL_LIGHT_ENV_BUMP_NOT_USED_BUMP	なし
GL_LIGHT_ENV_BUMP_AS_BUMP_DMP	法線ベクトル(バンプマッピング)
GL_LIGHT_ENV_BUMP_AS_TANG_DMP	接線ベクトル(タンジェントマッピング)

法線の再計算機能

法線の再計算機能(dmp_LightEnv.bumpRenorm)に GL_TRUE を指定すると、テクスチャからサンプルされた B 成分を摂動ベクトルの z 成分として使用せずに、z 成分が x、y 成分から再計算されます。

$$b_z = \sqrt{1.0 - \left(b_x{}^2 + b_y{}^2\right)}$$ ※ ルート内が負値となる場合、計算結果は 0 となります。

多くの場合において、テクスチャからサンプルされた値を使用するよりも再計算を行う方が良好な結果となります。また、R と G 成分のみのフォーマット(GL_HILO8_DMP)のテクスチャを使用して法線のバンプマッピングを行う場合は、再計算機能を有効にしなければなりません。ただし、異方性反射等で利用するために摂動方法でタンジェントマッピングを選択している場合は、この機能を使用しないことを推奨しています。これはフラグメントライティングのタンジェントマッピングが z 成分の存在しない摂動接線の入力を想定しているためで、再計算機能が有効になっていると z 成分に 0 以外が生成される可能性があるからです。

再計算機能を無効にすると、テクスチャからサンプルされた摂動法線が正規化されずに使用されます。テクスチャには、あらかじめ正規化した値を格納してください。また、テクスチャのフィルタモードがポイントサンプリング(GL_NEAREST)でない場合は、フィルタリングの影響で正規化されていない値が摂動法線として使用される可能性がありますので、法線の再計算機能を有効にしてください。

表 12-11. バンプマッピングで使用する予約ユニフォーム

予約ユニフォーム	種別	設定値
dmp_LightEnv.bumpSelector	int	法線マップとして使用するテクスチャユニットを指定する。 GL_TEXTURE0(デフォルト) GL_TEXTURE1 GL_TEXTURE2 GL_TEXTURE3
dmp_LightEnv.bumpMode	int	法線または接線の摂動モードを指定する。 GL_LIGHT_ENV_BUMP_NOT_USED_DMP:なし(デフォルト) GL_LIGHT_ENV_BUMP_AS_BUMP_DMP:法線を摂動する GL_LIGHT_ENV_BUMP_AS_TANG_DMP:接線を摂動する
dmp_LightEnv.bumpRenorm	bool	法線の第 3 成分を再生成するかどうかを指定する。 GL_TRUE:再生成する GL_FALSE:再生成しない(デフォルト)

12.4. シャドウ

3DS のシャドウは、シャドウバッファ(光源を起点としたシーンのデプス情報)を作成するシャドウ累積パス(第 1 パス)と、作成したシャドウバッファを参照してシャドウを落とす参照パス(第 2 パス)の 2 パスでシャドウを描画する機能です。また、第 1 パスでデプス情報とともに収集されるシャドウ強度の情報により、ソフトシャドウの表現を施すことができます。

12.4.1. シャドウ累積パス

シャドウ累積パスのために、フラグメントオペレーションモード(dmp_FragOperation.mode)をシャドウモード(GL_FRAGOP_MODE_SHADOW_DMP)に切り替え、シャドウ情報(デプス値とシャドウ強度)がシャドウテクスチャ(フォーマットとタイプの組み合わせが GL_SHADOW_DMP と GL_UNSIGNED_INT のテクスチャ)に格納されるようにしなければなりません。シャドウテクスチャにシャドウ情報を書き込むことができるのはテクスチャユニット 0(GL_TEXTURE0)だけであることに注意してください。また、シャドウテクスチャにはミップマップを適用することができません。

フラグメントのパイプラインがシャドウモードに切り替わると、デプスやステンシルではなく、カラーバッファのアタッチメントポイントにシャドウ情報が出力されます。そのため、シャドウテクスチャはカラーバッファのアタッチメントポイント(GL_COLOR_ATTACHMENT0)にアタッチされていなければなりません。シャドウモードでは、デプスとステンシルのアタッチメントポイントにアタッチされたレンダーターゲットは無視されます。アルファテストとステンシルテストは行われません。

以下の手順で、シャドウテクスチャの作成とレンダーターゲットへの指定を行うことができます。

コード 12-6. シャドウテクスチャの作成とレンダーターゲットへの指定

```
glActiveTexture(GL_TEXTURE0);
glGenTextures(1, &shadowTexID);
glBindTexture(GL_TEXTURE_2D, shadowTexID);
glTexImage2D(GL_TEXTURE_2D, 0, GL_SHADOW_DMP, shadowWidth, shadowHeight, 0,
        GL_SHADOW_DMP, GL_UNSIGNED_INT, 0);
glGenFramebuffers(1, &shadowFboID);
glBindFramebuffer(GL_FRAMEBUFFER, shadowFboID);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
        shadowTexID, 0);
```

シャドウ情報の累積は光源座標系で行われます。シャドウ情報は、光源からの深度(デプス情報)とシャドウ強度からなり、シャドウを描画する際のカラー情報の G 成分(R, B, A 成分は影響しない)が 1.0 であればシャドウのない状態、0.0 であれば不透明なハードシャドウのある状態、それ以外であれば不透明ではないシャドウ(ソフトシャドウ)のある状態です。

不透明なハードシャドウが描画されるときはシャドウのデプス情報のみが更新され、シャドウの強度は更新されません。フラグメントはシャドウバッファ内の対応するピクセルと深度の比較(GL_LESS に相当)が行われ、フラグメント側の値が小さければシャドウバッファのデプス情報が更新されます。

不透明ではないシャドウ(ソフトシャドウ)が描画されるときはシャドウの強度情報のみが更新され、シャドウのデプス情報は更新されません。フラグメントはシャドウバッファ内の対応するピクセルと深度の比較(GL_LESS に相当)が行われ、フラグメント側が小さければ、さらに強度情報に関しても比較(GL_LESS に相当)が行われ、これもフラグメント側が小さければシャドウバッファの強度情報が更新されます。

シャドウ累積パスでカラーバッファを初期化する際には、glClearColor() でクリアカラーに (1.0, 1.0, 1.0, 1.0) を指定し、glClear() には GL_COLOR_BUFFER_BIT を指定しなければなりません。クリアカラーには、G 成分のみではなく、すべてのカラー成分(R, G, B, A)を 1.0 にしなければならないことに注意してください。

次のシャドウ参照パスでは視点座標系で処理が行われるため、デプス情報は視点空間上の線形補間(多くの場合、非線形の関係である OpenGL とは異なります)によって生成されなければなりません。そのため、予約ユニフォームで w バッファのスケール因子(dmp_FragOperation.wScale)に任意のスケール因子を glUniform1f() で設定しなければなりま

せん。初期値は 0.0 ですが、このままでは OpenGL と同じ非線形の関係となり、near クリップ面が視点に近い場合に far クリップ面付近のデプス情報の有効精度が小さくなってしまいます。線形補間の関係にあるようにするには、f を far クリップ面のクリップ値とすれば、射影変換が透視投影である場合は 1.0/f を、平行投影である場合は 1.0 をスケール因子に設定してください。

ハードシャドウのレンダリングは影を落とすオブジェクトをカラーの G 成分を 0.0 で描画することで行います。テクスチャを無効にした状態で、頂点カラーの G 成分が 0.0 で出力されるように頂点シェーダを実装するのが一般的な方法です。これ以外のカラーで描画された場合はソフトシャドウとして扱われます。

必要なシャドウ情報を累積するためには、何回かのレンダリングパスが必要になる場合があります。不透明でないシャドウに関する情報を累積するときは、先に不透明なハードシャドウに関する情報を累積した後に行わなければなりません。順序が逆であったり、交互に累積したりした場合の結果は保証されません。光源が動かない場合は、あらかじめシャドウテクスチャに動かない物体のみ描画しておくことでシャドウテクスチャの生成処理を軽減させることができます。また、オブジェクトの形状を単純化させてポリゴン数を減らすこともパフォーマンス向上には効果的です。

頂点シェーダで頂点カラーを出力するように実装した場合の、シャドウレンダリング時のテクスチャユニット 0 およびテクスチャコンバイナは以下のように設定します。

コード 12-7. シャドウレンダリング時のテクスチャユニット、テクスチャコンバイナの設定例

```
glUniform1i(glGetUniformLocation(progID, "dmp_Texture[0].samplerType"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_TexEnv[0].combineRgb"),
        GL_REPLACE);
glUniform1i(glGetUniformLocation(progID, "dmp_TexEnv[0].combineAlpha"),
        GL_REPLACE);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].operandRgb"),
        GL_SRC_COLOR, GL_SRC_COLOR, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].operandAlpha"),
        GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].srcRgb"),
        GL_PRIMARY_COLOR, GL_PRIMARY_COLOR, GL_PRIMARY_COLOR);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].srcAlpha"),
        GL_PRIMARY_COLOR, GL_PRIMARY_COLOR, GL_PRIMARY_COLOR);
```

12.4.2. シャドウ参照パス

シャドウ参照パスのために、フラグメントオペレーションモードを通常モード(GL_FRAGOP_MODE_GL_DMP)に切り替え、テクスチャユニット 0(GL_TEXTURE0)がシャドウ情報を累積したシャドウテクスチャを参照するように設定しなければなりません。シャドウテクスチャを参照させるには、予約ユニフォーム(dmp_TexEnv[0].samplerType)にシャドウテクスチャ(GL_TEXTURE_SHADOW_2D_DMP)を指定し、シャドウ情報を累積したテクスチャを GL_TEXTURE_2D にバインドさせなければなりません。テクスチャコンバイナには、フラグメントライティングが有効な場合はフラグメントライティングのプライマリとセカンダリカラーを入力し、無効な場合は頂点カラーとテクスチャユニット 0 の出力を入力します。

シャドウ累積パスを行った際に、射影変換に透視投影を適用したか平行投影を適用したかを、予約ユニフォーム(dmp_Texture[0].perspectiveShadow)で設定します。透視投影ならば GL_TRUE を、平行投影ならば GL_FALSE を指定してください。

コード 12-8. フラグメントライティングが無効な場合の設定例

```
glUniform1i(glGetUniformLocation(progID, "dmp_Texture[0].samplerType"),
        GL_TEXTURE_SHADOW_2D_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_Texture[0].perspectiveShadow"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_TexEnv[0].combineRgb"),
        GL_MODULATE);
glUniform1i(glGetUniformLocation(progID, "dmp_TexEnv[0].combineAlpha"),
        GL_MODULATE);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].operandRgb"),
        GL_SRC_COLOR, GL_SRC_COLOR, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].operandAlpha"),
        GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].srcRgb"),
        GL_TEXTURE0, GL_PRIMARY_COLOR, GL_PRIMARY_COLOR);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].srcAlpha"),
        GL_TEXTURE0, GL_PRIMARY_COLOR, GL_PRIMARY_COLOR);
glUniform1i(glGetUniformLocation(progID, "dmp_FragOperation.mode"),
        GL_FRAGOP_MODE_GL_DMP);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, shadowTexID);
```

シャドウ情報をテクスチャとして参照するのは 3DS でも OpenGL でも同じです。テクスチャユニットではテクセルから得るシャドウ情報とテクスチャ座標とを比較しますが、このときに適切なテクスチャ座標が指定されていないと正しく比較処理が行われません。シャドウテクスチャを参照するときのテクスチャ座標が、OpenGL では (s/q, t/q, r/q) であるのに対し、3DS では (s/r, t/r, r – bias) であることに注意しなければなりません。(s, t, r, q) はテクスチャ変換行列適用後のテクスチャ座標、bias は予約ユニフォーム(dmp_Texture[0].shadowZBias)に設定されたバイアス値です。平行投影によりシャドウ情報を累積している場合は、テクスチャ座標 s, t はそのまま参照されますが、透視投影の場合はテクスチャ変換行列とバイアス値を調整しなければなりません。

累積したシャドウ情報とテクスチャ座標の比較処理が正しく行われるようにする設定としては、以下のテクスチャ変換行列とバイアス値を使用することが考えられます。

透視投影の場合

$$texture\_matrix = \begin{bmatrix} \frac{-1}{f-n}\frac{-n}{2r} & 0 & \frac{-1}{f-n}\frac{1}{2} & 0 \\ 0 & \frac{-1}{f-n}\frac{-n}{2t} & \frac{-1}{f-n}\frac{1}{2} & 0 \\ 0 & 0 & \frac{-1}{f-n} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$bias = \frac{n}{f-n}$$

平行投影の場合

$$texture\_matrix = \begin{bmatrix} \frac{1}{2r} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2t} & 0 & \frac{1}{2} \\ 0 & 0 & \frac{-1}{f-n} & \frac{-n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$bias = 0$$

n は near クリップ面のクリップ値、f は far クリップ面のクリップ値、r と t は Frustum の右辺値と上辺値です。

ただし、テクスチャ座標 r が 0.0 ～ 1.0 の範囲外にある場合、シャドウバッファの深度情報と比較される値 (r - bias) を計算するときには、r を 0.0 ～ 1.0 の範囲にクランプしたあと、bias による減算とクランプが行われます。正しい比較が行われるためには、シャドウ累積パスの光源座標系で、far 平面の奥に配置されるようなオブジェクトに関する bias には 0 を指定してください。

OpenGL のコードであれば以下のように記述することでテクスチャ変換行列を生成することができます。

コード 12-9. OpenGL のコードによるテクスチャ変換行列の生成

```
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
// 透視投影(glFrustum(-r, r, -t, t, n, f))で累積していた場合
glFrustumf(r/n, -3r/n, t/n, -3t/n, 1.0f, 0.0f);
glScalef(-1.0f/(f-n), -1.0f/(f-n), -1.0f/(f-n));
// 平行投影(glOrtho(-r, r, -t, t, n, f))で累積していた場合
glOrthof(-3r, r, -3t, t, 2n-f, f);
```

ボーダーカラーとテクスチャラッピングモードの設定によって、0.0 ～ 1.0 の範囲外に算出されたテクスチャ座標によるサンプリング結果を制御することができます。テクスチャラッピングモードにより、テクスチャ座標の s, t 成分に対して GL_CLAMP_TO_BORDER が設定されている場合、範囲外のサンプリング値はボーダーカラー (すべての成分値が 0.0 または 1.0 に統一されていなければならない) の設定になることが保証されています。現在の実装では、ラッピングモードに GL_CLAMP_TO_BORDER 以外を設定した場合や、ボーダーカラーに (0.0, 0.0, 0.0, 0.0) または (1.0, 1.0, 1.0, 1.0) 以外を設定した場合のサンプリング結果は不定です。

コード 12-10. ラッピングモードとボーダーカラーの設定例

```
glBindTexture(GL_TEXTURE_2D, shadowTexID);
GLfloat bcolor[] = {1.0f, 1.0f, 1.0f, 1.0f};
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, bcolor);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_LOD, 0);
```

光源座標系でのデプス値がシャドウテクスチャのテクセルに含まれるデプス情報よりも大きい場合、比較後のシャドウ強度は 0.0 となります。それ以外の場合はシャドウテクスチャのテクセルに含まれるシャドウ強度そのものです。シャドウ強度はテクスチャユニットから RGBA の各要素の値として出力されます。

12.4.3. 全方位シャドウマッピング

キューブマッピングとシャドウテクスチャの組み合わせで、全方位シャドウマッピングを実現することができます。

実現するためには、シャドウ累積パスでテクスチャユニットにバインドするテクスチャをキューブマップテクスチャ(GL_TEXTURE_CUBE_MAP_POSITIVE_{X,Y,Z} または GL_TEXTURE_CUBE_MAP_NEGATIVE_{X,Y,Z} の 6 つ)を指定して 6 回レンダリングを行い、シャドウ参照パスで参照するテクスチャ(dmp_Texture[0]. samplerType)に GL_TEXTURE_SHADOW_CUBE_DMP を指定しなければなりません。また、テクスチャ座標のラッピングモードには、S, T 方向ともに GL_CLAMP_TO_EDGE を指定しなければなりません。

12.4.4. シルエットシェーダを利用したソフトシャドウ

シャドウテクスチャはデプス情報とシャドウ強度を含んでおり、シャドウテクスチャを参照しているテクスチャユニットからは、シャドウ領域内ならば黒 (0.0, 0.0, 0.0) が、領域外ならばシャドウテクスチャに保持されているシャドウ強度(0.0 ～ 1.0)がサンプリング値として出力されます。このシャドウ強度を適切な値に設定することでソフトシャドウを表現することができます。

3DS のシルエットシェーダは、シルエットのプリミティブ(シルエットエッジ)をシャドウが完全に消える部分に向かってシャドウ強度が段階的に変わるようなソフトシャドウ領域として描画することができます。シルエットシェーダを使用したレンダリングは、不透明ハードシャドウのレンダリングを行った後で行います。シルエットエッジのカラーの G 成分には 1.0 を指定し、頂点シェーダで頂点カラーの G 成分に 0.0 が出力されるように設定します。そのほかの設定は不透明ハードシャドウのレンダリング時と同じにします。これにより、シルエットエッジの矩形はオブジェクト側の G 成分が 0.0 に、外側に向かって段階的に G 成分が 1.0 となるように描画されます。

このソフトシャドウ領域の累積パスでは、シャドウ強度に対してオブジェクトまでの相対距離による付加的モジュレーションを適用することができます。これを減衰因子と呼び、予約ユニフォームのソフトシャドウのバイアス値(dmp_FragOperation.penumbraBias)とスケール値(dmp_FragOperation.penumbraScale)により、ソフトシャドウ領域の幅を調整することができます。これらの値を調整することで、オブジェクトに近い位置のソフトシャドウは幅が狭くなり、より自然なソフトシャドウを表現することができるようになります。

減衰因子は以下の数式で計算されます。式中の Z_{frag} はフラグメントのデプス値、Z_{rec} はシャドウテクスチャに格納されているデプス値です。前もってオブジェクトを描画し、シャドウテクスチャにデプス値を格納していなければシャドウ強度の減衰の効果は正しく反映されません。

$$\frac{1}{bias + scale \times \dfrac{1 - Z_{frag}}{Z_{rec}}}$$

12.4.5. シャドウアーティファクトへの対処

12.4.5.1. セルフシャドウエイリアシング

マルチパスによるシャドウレンダリングでは、誤ってフラグメントが自分自身に影を落とすセルフシャドウエイリアシングにより、レンダリング結果にモアレが発生するなどの問題が発生することがあります。これは、累積時のデプス情報が参照時に光源から見たデプス値よりもわずかに小さな場合に起こります。このとき発生する不自然な影をシャドウアーティファクトと呼びます。

シャドウアーティファクトの抑制方法として、参照パスのデプス値に負のオフセット値(バイアス)を適用する方法が存在します。バイアス値は、予約ユニフォームの dmp_Texture[0].shadowZBias で設定することができます。

コード 12-11. セルフシャドウエイリアシング抑制の設定例

```
glUniform1f(glGetUniformLocation(progID, "dmp_Texture[0].shadowZBias"),
            1.2f*n/(f-n));
```

12.4.5.2. シルエットのシャドウアーティファクト

参照パスでシャドウ領域ではないと判断された場合、シャドウテクスチャのシャドウ強度がサンプリング値として出力されることは以前にも説明しました。通常はシャドウ領域外のシャドウ強度は 1.0 で輝度の減衰は起こりませんが、シルエットシェーダで描画されたソフトシャドウ領域で出力されるシャドウ強度が 1.0 以外になることで輝度の減衰が起こり、オブジェクトによってはシャドウアーティファクトが発生することになります。

シルエットのシャドウが生成するシャドウアーティファクトに対しては、セルフシャドウエイリアシングの抑制方法は有効ではありませんが、フラグメントライティングのシャドウに関する設定とテクスチャコンバイナの設定を調整することで抑制することができます。

主に光源に対して平行な面にソフトシャドウによる輝度減衰が起こることが原因ですので、シャドウテクスチャの出力値（シャドウ減衰項）を以下の計算式で算出する方法が存在します。

$$ShadowAttenuation = 1.0 - f(1.0 - ShadowIntensity)$$

f はフラグメントの法線が光源に垂直なときは 0.0 近傍、平行なときは 1.0 近傍の値を返す関数です。

この計算式によりシャドウ減衰項は、フラグメントの法線と光源が垂直に近い場合は 1.0（シャドウの影響なし）、平行に近い場合はシャドウ強度そのものとなります。

このシャドウ減衰項は、関数 f の部分をフレネルファクタ（参照テーブル FR）、シャドウ強度の反転部分をライティング環境の予約ユニフォーム（dmp_LightEnv.shadowAlpha、dmp_LightEnv.invertShadow）、最終的な値の反転部分をテクスチャコンバイナの設定により実装することができます。

ライティング関連は以下の設定で行います。フレネルファクタがアルファのみに影響を与えることに注意してください。アルファ成分はテクスチャコンバイナで乗算します。関数 f は入力値の 2 乗を返す設定です。

コード 12-12. シルエットのシャドウアーティファクト抑制の設定例（ライティング関連）

```
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentLighting.enabled"),
        GL_TRUE);
// ..other code..
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputFR"),
        GL_LIGHT_ENV_LN_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.config"),
        GL_LIGHT_ENV_LAYER_CONFIG1_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.fresnelSelector"),
        GL_LIGHT_ENV_PRI_SEC_ALPHA_FRESNEL_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.shadowAlpha"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.invertShadow"),
        GL_TRUE);

GLfloat lut[512];
int j;
memset(lut, 0, sizeof(lut));
for (j = 1; j < 128; j++)
{
    lut[j] = powf((float)j/127.0f, 2.0f);
```

```
    lut[j+255] = lut[j] - lut[j-1];
}
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, lut);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerFR"), 0);
```

以上のコードで、*f*(1.0 – *ShadowIntensity*) の部分がフラグメントのプライマリとセカンダリのアルファ成分として出力されるようになります。

次に、テクスチャコンバイナの設定により最終的なシャドウ減衰項を算出し、フラグメントのプライマリカラーと乗算します。このとき、フラグメントのプライマリアルファを反転(GL_ONE_MINUS_SRC_ALPHA)していることに注意してください。

コード 12-13. シルエットのシャドウアーティファクト抑制の設定例(テクスチャコンバイナ)

```
glUniform1i(glGetUniformLocation(progID, ("dmp_TexEnv[0].combineRgb"),
        GL_MODULATE);
glUniform1i(glGetUniformLocation(progID, ("dmp_TexEnv[0].combineAlpha"),
        GL_REPLACE);
glUniform3i(glGetUniformLocation(progID, ("dmp_TexEnv[0].operandRgb"),
        GL_SRC_COLOR, GL_ONE_MINUS_SRC_ALPHA, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(progID, ("dmp_TexEnv[0].operandAlpha"),
        GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
glUniform3i(glGetUniformLocation(progID, ("dmp_TexEnv[0].srcRgb"),
        GL_FRAGMENT_PRIMARY_COLOR_DMP, GL_FRAGMENT_PRIMARY_COLOR_DMP,
        GL_PRIMARY_COLOR);
glUniform3i(glGetUniformLocation(progID, ("dmp_TexEnv[0].srcAlpha"),
        GL_PRIMARY_COLOR, GL_PRIMARY_COLOR, GL_PRIMARY_COLOR);
```

12.4.6. 予約ユニフォーム

以下の表は、シャドウで使用する予約ユニフォームの一覧です。

表 12-12. シャドウで使用する予約ユニフォーム

予約ユニフォーム	種別	設定値
dmp_Texture[0].samplerType	int	参照するテクスチャの種別を指定する。シャドウで使用可能なのは以下の 2 つです。 GL_TEXTURE_SHADOW_2D_DMP GL_TEXTURE_SHADOW_CUBE_DMP
dmp_Texture[0].perspectiveShadow	bool	射影変換に透視投影を適用しているかどうかを指定する。 GL_TRUE:適用している(デフォルト) GL_FALSE:適用していない
dmp_Texture[0].shadowZBias	float	参照パスのデプス値に適用される負のオフセットのバイアス値を指定する。 0.0(デフォルト)
dmp_FragOperation.mode	int	フラグメントオペレーションモードを指定する。累積パスではシャドウモード、参照パスでは標準モードを指定する。 GL_FRAGOP_MODE_GL_DMP(標準モード) GL_FRAGOP_MODE_SHADOW_DMP(シャドウモード)
dmp_FragOperation.wScale	float	w バッファのスケール因子を指定する。 0.0(デフォルト)

dmp_FragOperation.penumbraScale	float	ソフトシャドウのスケール値を指定する。 1.0(デフォルト)
dmp_FragOperation.penumbraBias	float	ソフトシャドウのバイアス値を指定する。 0.0(デフォルト)

12.4.7. シャドウテクスチャの内容を確認する方法

シャドウテクスチャにレンダリングしたイメージを確認するには、シャドウテクスチャをカレントのカラーバッファにアタッチした状態で、*format* に GL_RGBA を、*type* に GL_UNSIGNED_BYTE を指定して glReadPixels() を呼び出し、テクセルデータを読み出してください。テクセルデータに u32 型のポインタでアクセスした場合、下位 8 bit にシャドウ強度が、上位 24 bit にデプス値が格納されています。デプス値だけを取得するには右に 8 bit シフトしてください。

シャドウテクスチャに格納されているシャドウ強度とデプス値は、8 bit 値と 24 bit 値で範囲 0.0 ～ 1.0 の値を表しているため、それぞれ精度が異なります。

シャドウ強度は 0x00 ～ 0xFF の範囲の値をとり、0xFF ならばソフトシャドウ領域が存在せず、それ以外ならばソフトシャドウ領域が存在することを示しています。つまり、0x00 ならばシャドウ強度は 0.0、0xFF ならば 1.0 です。

デプス値は near 値を 0x000000、far 値を 0xFFFFFF とする範囲にスケーリングされた値です。つまり、0x000000 ならばデプス値は 0.0、0xFFFFFF ならば 1.0 です。シャドウ累積パスで w バッファ(「10.3.3. w バッファ」)を有効にしている場合は、このスケーリングが一様であることに注意してください。

シャドウテクスチャに書き込まれるシャドウ強度とデプス値は、シャドウが存在しない領域ではシャドウ強度が 0xFF、デプス値が 0xFFFFFF となり、ハードシャドウだけが存在する領域ではシャドウ強度が 0xFF、デプス値が 0xFFFFFF 以外の値となります。また、ハードシャドウとソフトシャドウの両方が存在する領域ではシャドウ強度が 0xFF 以外、デプス値が 0xFFFFFF 以外の値となります。

表 12-13. シャドウの種類とシャドウテクスチャに書き込まれる値

シャドウの種類	シャドウ強度	デプス値
シャドウなし	0xFF	0xFFFFFF
ハードシャドウのみ	0xFF	0xFFFFFF 以外
ソフトシャドウあり	0xFF 以外	0xFFFFFF 以外

12.5. フォグ

3DS のフォグは OpenGL ES 1.1 に定義されているフォグとほぼ同等の機能を持ちますが、OpenGL が視点からの距離でフォグの効果が決定されるのに対し、3DS は投影で補正されたデプス値により決定されます。また、フォグ係数の指定を参照テーブルで行う点も異なり、フォグの開始・終了・密度などの設定は存在しません。

OpenGL と同様に、フォグ適用後のフラグメントのカラーは以下の計算式で決定されます。

$$Color = f \times C_{fragment} + (1 - f) \times C_{fog}$$

f はフォグ係数(0.0 ～ 1.0)です。

12.5.1. 予約ユニフォーム

フォグの予約ユニフォームには、以下のものが存在します。

フォグモード

フォグ機能を有効にするには、フォグモード(dmp_Fog.mode)に glUniform1i() で GL_FOG を設定してください。無効にするには GL_FALSE を設定します。

フォグカラー

フォグカラー(dmp_Fog.color)には glUniform3f() でカラー値を設定します。RGB 成分のみでアルファ成分は設定しません。

フォグ係数

フォグ係数は入力値としてウィンドウ座標系でのデプス値をとる参照テーブルを設定します。glUniform1i() で使用する参照テーブル番号を予約ユニフォーム(dmp_Fog.sampler)に指定してください。ここで指定する参照テーブル番号は GL_LUT_TEXTURE*i*_DMP の *i*(0 ～ 31) が表す数値で、テクスチャの名前(ID)や GL_LUT_TEXTURE*i*_DMP を直接指定するわけではないことに注意してください。

入力デプス値の反転

フォグ係数の参照テーブルへの入力値を反転(z ではなく 1 – z に)するかどうかを選択することができます。反転させる場合は、予約ユニフォーム(dmp_Fog.zFlip)に glUniform1i() で GL_TRUE を設定してください。

表 12-14. フォグで使用する予約ユニフォーム

予約ユニフォーム	種別	設定値
dmp_Fog.mode	int	フォグパイプラインの処理モード(フォグモード)を指定する。 GL_FOG GL_GAS_DMP GL_FALSE(デフォルト)
dmp_Fog.color	vec3	フォグカラーを指定する。アルファ成分はありません。 各成分とも 0.0 ～ 1.0 (デフォルト)(0.0, 0.0, 0.0)
dmp_Fog.zFlip	bool	フォグ係数の参照テーブルに入力するデプス値を反転するかどうかを指定する。 GL_TRUE GL_FALSE(デフォルト)
dmp_Fog.sampler	int	フォグ係数に使用する参照テーブルを指定する。 0 ～ 31

12.5.2. 参照テーブルの作成

フォグ係数の参照テーブルは入力値の範囲が 0.0 ～ 1.0 で *width* は 256 固定です。ほかの参照テーブルと同様に前半の 128 要素には出力値を、後半の 128 要素には出力値間の差分を設定します。

OpenGL のフォグ係数を 3DS で実装する際に、参照テーブルの作成で注意しなければならないのは入力値の変換方法です。入力値はウィンドウ座標系のデプス値であり、ニアクリップ平面で最小値(0.0)、ファークリップ平面で最大値(1.0)となるのですが、透視投影では視点座標系のデプス値との対応が線形的ではありません。そのため、参照テーブルの出力値を設定するときには、ウィンドウ座標系のデプス値から視点座標系のデプス値に変換した値で計算された出力値でなければなりません。

ウィンドウ座標系である入力値(0.0 ～ 1.0)はクリップ座標系の(0.0 ～ -1.0)にマッピングされることを考慮して、視点座標系への変換には以下の計算式を使用します。入力値の符号を逆にする点に注意してください。

$$(Xe\ Ye\ Ze\ We) = (0.0\ 0.0\ -Zw\ 1.0) \times M_{projection}^{-1}$$

フォグ係数は視点座標系の原点からフラグメントへの距離の関数ですので、その関数への入力値は視点座標系でのフラグメントと xy 平面との距離 $-Ze / We$ で近似することができます。

以下にそのコード例を示します。FogCoef() はフォグ係数の関数です。

コード 12-14. フォグ参照テーブルの作成例

```
float Fog_LUT[256], Fog_c[128 + 1];
int i;
Matrix44 invPM;
Vector4 v_eye, v_clip(0.0f, 0.0f, 0.0f, 1.0f);

MTX44Inverse(&invPM, &projMatrix);
Vector4 v0(invPM.m[0]);
Vector4 v1(invPM.m[1]);
Vector4 v2(invPM.m[2]);
Vector4 v3(invPM.m[3]);
for (i = 0; i <= 128; i++) {
    v_clip.z = -(static_cast<f32>(i)) / 128;
    v_eye.x = VEC4Dot(&v0, &v_clip);    v_eye.y = VEC4Dot(&v1, &v_clip);
    v_eye.z = VEC4Dot(&v2, &v_clip);    v_eye.w = VEC4Dot(&v3, &v_clip);
    Fog_c[i] = -(v_eye.z / v_eye.w);
}
for (i = 0; i < 128; i++) {
    Fog_LUT[i] = FogCoef(Fog_c[i]);
    Fog_LUT[128 + i]= FogCoef(Fog_c[i + 1]) - FogCoef(Fog_c[i]);
}
```

補足: OpenGL のフォグ機能は視点座標系の z 座標値の影響を受けていましたが、3DS のフォグ機能は透視投影補正されたデプス値の影響を受けます。そのため、ニアクリップ面、ファークリップ面が変更されるとフォグの効果が変化し、w バッファ(「10.3.3. w バッファ」参照)を使用するか、通常のデプスバッファを使用するかで、同じ参照テーブルが異なった効果を与えることになります。

12.6. ガスレンダリング

ガスレンダリング機能は、ポリゴンオブジェクトのデプス情報を考慮しながら作成される密度情報を使って、ガスモードに設定されたフォグ機能がガス状物体をレンダリングする機能です。ガス状物体をレンダリングするには、ポリゴンオブジェクトのデプス情報を生成するポリゴンオブジェクト描画パス、ガステクスチャに密度情報を累積する密度情報描画パス、ガステクスチャを参照してガス状物体をレンダリングするシェーディングパスの 3 パスが必要となります。

12.6.1. ポリゴンオブジェクト描画パス

このパスでは通常通りにポリゴンオブジェクトをレンダリングし、ポリゴンオブジェクトとガス状物体との交差判定をするためのデプス情報を作成します。レンダリング結果のうち、デプスバッファの内容が次のパスで使用されます。

12.6.2. 密度情報描画パス

このパスではガス状物体を構成する最小単位であるガスパーティクル(密度パターンテクスチャ)をポイントスプライトなどでレンダリングし、ガスの密度情報をカラーバッファに累積します。カラーバッファの内容はガステクスチャにコピーして次のパスで使用されます。

カラーバッファとガステクスチャの準備

内部フォーマットが GL_GAS_DMP のカラーバッファと、レンダリング結果をコピーするテクスチャ(ガステクスチャ)を用意します。カラーバッファはデプスバッファと同じサイズで作成しますが、ガステクスチャは幅と高さのテクセル数が 2 のべき乗でなければならず、*type* には GL_UNSIGNED_SHORT を指定しなければなりません。また、ガステクスチャは常にポイントサンプリングされるため、拡大時と縮小時のフィルタ設定は無効となります。フィルタ設定には GL_NEAREST を指定してください。ガステクスチャにはミップマップを適用することができません。

コード 12-15. ガスレンダリング時のカラーバッファとガステクスチャの準備

```
// Generating Object
glGenFramebuffers(1, &gasAccFboID);
glGenTextures(1, &gasTexID);
// Renderbuffer & Framebuffer
glGenRenderbuffers(1, &gasAccColorID);
glBindRenderbuffer(GL_RENDERBUFFER, gasAccColorID);
glRenderbufferStorage(GL_RENDERBUFFER, GL_GAS_DMP, GAS_ACC_WIDTH,
        GAS_ACC_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, gasAccFboID);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_RENDERBUFFER,
        gasAccColorID);
// Gaseous Texture
glBindTexture(GL_TEXTURE_2D, gasTexID);
glTexImage2D(GL_TEXTURE_2D, 0, GL_GAS_DMP, GAS_TEX_WIDTH, GAS_TEX_HEIGHT, 0,
        GL_GAS_DMP, GL_UNSIGNED_SHORT, 0);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
```

ガスパーティクルのレンダリング

カラーバッファに密度情報を描画するには、フラグメントオペレーションモード(dmp_FragOperation.mode)をガスモード(GL_FRAGOP_MODE_GAS_ACC_DMP)に切り替えます。カラーバッファには単純に累積したもの(D1)とポリゴンオブジェクトとの交差を考慮したもの(D2)の 2 種類の密度情報が蓄積されます。

デプスバッファの内容が更新されないようにデプステストとデプスマスクはオフにし、ブレンディングとフォグもオフにします。ただし、ポリゴンオブジェクトをレンダリングしたときのデプステストの比較関数はそのままにしておいてください。

バッファをクリアするのは密度情報が描画されるカラーバッファのみです。デプスバッファをクリアしてはいけません。

レンダリングされるガスパーティクルの R 成分(Df)が密度情報としてカラーバッファに蓄積されます。D1 と D2 は以下の計算式によって D1’ と D2’ に更新されます。D2’ の式はデプステストの比較関数によって変化します。

$$D1' = D1 + Df$$

$$DZ = (Zb - Zf) < 0.0 \,?\, 0.0 : (Zb - Zf) \times EZ$$
$$ATT = DZ > 1.0 \,?\, 1.0 : DZ$$
$$D2' = D2 + Df \times ATT \quad \cdots \text{GL\_LESS,GL\_LEQUAL}$$
$$D2' = D2 + Df \times (1.0 - ATT) \quad \cdots \text{GL\_GREATER,GL\_GEQUAL}$$
$$D2' = D2 + Df \quad \cdots \text{GL\_ALWAYS}$$
$$D2' = D2 \quad \cdots \text{GL\_NEVER}$$

Zb はデプスバッファに格納されているデプス値、Zf はフラグメントのデプス値です。

D1 にはフラグメント(ガスパーティクル)の密度情報がそのまま累積されます。D2 にはデプスバッファのデプス値とフラグメントのデプス値との差分にデプス方向の減衰係数 EZ を掛け合わせた値が、さらにフラグメントの密度情報と掛け合わされて累積されます。係数 EZ は予約ユニフォーム(dmp_Gas.deltaZ)に glUniform1f() で設定された浮動小数点数です。

コード 12-16. ガスパーティクルのレンダリング

```
// Change to Gas Accumulation Mode
glBindFramebuffer(GL_FRAMEBUFFER, gasAccFboID);
glUniform1i(glGetUniformLocation(progID, "dmp_FragOperation.mode"),
        GL_FRAGOP_MODE_GAS_ACC_DMP);
glDisable(GL_DEPTH_TEST);
glDepthMask(GL_FALSE);
glDisable(GL_BLEND);
glUniform1i(glGetUniformLocation(progID, "dmp_Fog.mode"), GL_FALSE);
// Colorbuffer Clear
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT);
// Set dmp_Gas.deltaZ
glDepthFunc(GL_LESS);
glUniform1f(glGetUniformLocation(progID, "dmp_Gas.deltaZ"), 50.0f);
```

ガステクスチャへのコピー

すべてのガスパーティクルをレンダリングしたのち、カラーバッファに蓄積した密度情報をガステクスチャにコピーします。

通常、ガステクスチャはカラーバッファと同じ大きさで確保されない(幅と高さが 2 のべき乗でなければならない)ため、カラーバッファから glCopyTexSubImage2D() で部分コピーすることになります。ガステクスチャは次のパスで必要となります。

コード 12-17. ガステクスチャへのコピー

```
// Bind and CopyTo Gaseous Texture
glBindTexture(GL_TEXTURE_2D, gasTexID);
glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, GAS_ACC_WIDTH,
                    GAS_ACC_HEIGHT);
```

12.6.3. シェーディングパス

このパスでは、ガステクスチャに蓄積されている密度情報を参照してガス状物体のシェーディングを行い、その結果をポリゴンオブジェクト描画パスのカラーバッファとブレンディングします。

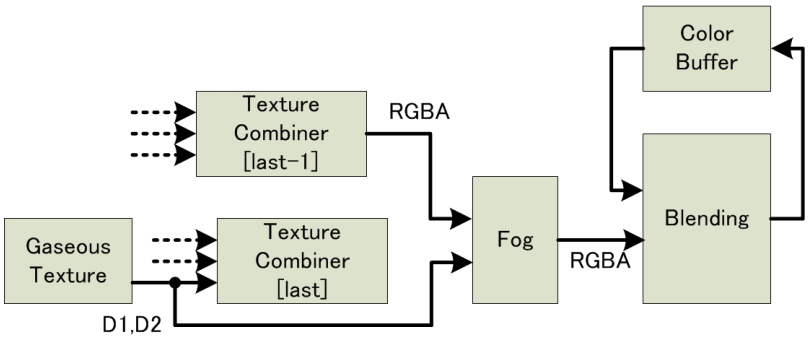
ガス状物体のシェーディングにはガスモードに設定されたフォグ機能を使用します。また、テクスチャコンバイナからフォグへの入力に関しては特殊な設定が必要となります。

フォグへの入力(テクスチャコンバイナの設定)

フォグは入力として、最終の 1 つ前のテクスチャコンバイナから RGBA 成分(実際に使用するのは R 成分のみ)と、最終のテクスチャコンバイナの入力ソース 2(srcRgb, srcAlpha の第 3 要素)に指定された入力ソース(ガステクスチャをサンプリングするテクスチャユニットを指定すること)から密度情報を受け取ります。最終のテクスチャコンバイナからの出力は無視されますので、結果的に使用可能なテクスチャコンバイナが 1 段減ることになります。

最終的に、フォグからの出力とカラーバッファとのブレンディングを行います。フォグから出力されるアルファ値はポリゴンオブジェクトとの交差を考慮したものですので、フォグのアルファ値をもとに両出力をブレンディングすることで正しい前後関係でガス状物体を描画することができます。

図 12-5. フォグへの入力とカラーバッファとのブレンディング

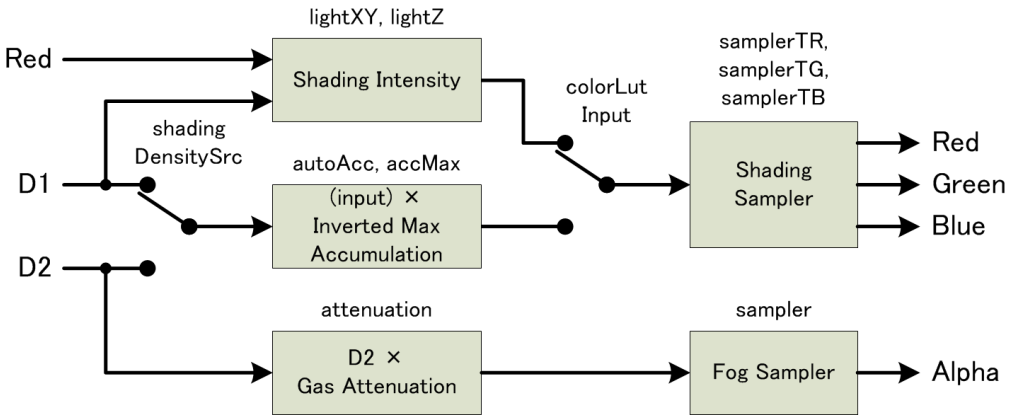


ガスモード時のフォグの動作

ガスモード時のフォグは、入力された情報をもとにガス状物体のシェーディングを行います。フォグ機能のガスモードを有効にするには、フォグモード(dmp_Fog.mode)に GL_GAS_DMP を設定してください。

シェーディング結果の RGB 成分はシェーディング参照テーブルによって決定されます。アルファ成分は、交差を考慮した密度情報(D2)に対してガスの減衰(dmp_Gas.attenuation)を掛け合わせたものを入力値としたときの、フォグ係数(dmp_Fog.sampler)で指定された参照テーブルからの出力値で決定されます。

図 12-6. ガスモード時のフォグ



シェーディング参照テーブル

シェーディング参照テーブルは密度情報もしくはシェーディング強度を入力値として受け取り、シェーディング結果の RGB 成分値を出力します。

シェーディング参照テーブルには、*width* に 16 を指定して作成したテーブルを成分別で指定します。入力される値は 0.0 ～ 1.0 の範囲で、出力する値の範囲も 0.0 ～ 1.0 です。前半の 8 要素に出力値を、後半の 8 要素に出力値間の差分を設定することや、出力値が差分値で補間されて算出されることはほかの参照テーブルと同じです。

使用するシェーディング参照テーブルは、`glUniform1i()` で参照テーブル番号を以下の予約ユニフォームに指定します。ここで指定する参照テーブル番号は `GL_LUT_TEXTURE`*i*`_DMP` の *i*(0 ～ 31) が表す数値で、テクスチャの名前(ID)や `GL_LUT_TEXTURE`*i*`_DMP` を直接指定するわけではないことに注意してください。

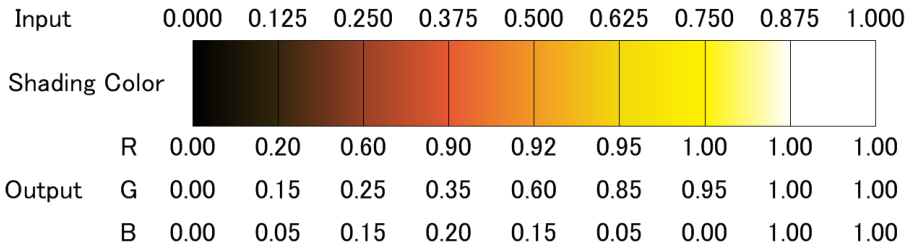
表 12-15. シェーディング参照テーブルの指定に使用する予約ユニフォーム

予約ユニフォーム	種別	設定値

`dmp_Gas.samplerTR`
`dmp_Gas.samplerTG`
`dmp_Gas.samplerTB`
	`int`	シェーディング参照テーブル(R、G、B 成分)に使用する参照テーブル番号を指定する。
		0 ～ 31

シェーディング参照テーブルの例と、その実装例を以下に示します。

図 12-7. シェーディング参照テーブルの例



コード 12-18. シェーディング参照テーブルの作成コード例

```
// Define
GLfloat shading_color[3 * 9] = {
    0.00f, 0.00f, 0.00f,
    0.20f, 0.15f, 0.05f,
    0.60f, 0.25f, 0.15f,
    0.90f, 0.35f, 0.20f,
    0.92f, 0.60f, 0.15f,
    0.95f, 0.85f, 0.05f,
    1.00f, 0.95f, 0.00f,
    1.00f, 1.00f, 1.00f,
    1.00f, 1.00f, 1.00f
};
GLfloat samplerTR[16], samplerTG[16], samplerTB[16];
// Table
for(int i = 0; i < 8; i++) {
    // shading color value
    samplerTR[i] = shading_color[3*i + 0];
    samplerTG[i] = shading_color[3*i + 1];
    samplerTB[i] = shading_color[3*i + 2];
    // difference of shading color value
    samplerTR[8 + i] = shading_color[3*(i + 1) + 0] - shading_color[3*i + 0];
    samplerTG[8 + i] = shading_color[3*(i + 1) + 1] - shading_color[3*i + 1];
```

```
    samplerTB[8 + i] = shading_color[3*(i + 1) + 2] - shading_color[3*i + 2];
}
// Texture
glBindTexture(GL_LUT_TEXTURE1_DMP, samplerTR_ID);
glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 16, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, samplerTR);
glBindTexture(GL_LUT_TEXTURE2_DMP, samplerTG_ID);
glTexImage1D(GL_LUT_TEXTURE2_DMP, 0, GL_LUMINANCEF_DMP, 16, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, samplerTG);
glBindTexture(GL_LUT_TEXTURE3_DMP, samplerTB_ID);
glTexImage1D(GL_LUT_TEXTURE3_DMP, 0, GL_LUMINANCEF_DMP, 16, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, samplerTB);
// Set Uniform
glUniform1i(glGetUniformLocation(progID, "dmp_Gas.samplerTR"), 1);
glUniform1i(glGetUniformLocation(progID, "dmp_Gas.samplerTG"), 2);
glUniform1i(glGetUniformLocation(progID, "dmp_Gas.samplerTB"), 3);
```

密度情報によるシェーディング

ライトによる陰影の影響を考慮せず、密度情報だけでシェーディングを行う場合は、glUniform1i() で予約ユニフォーム(dmp_Gas.colorLutInput)に GL_GAS_DENSITY_DMP を設定し、シェーディング参照テーブルへの入力値に密度情報を選択してください。

密度情報描画パスでも説明したように、ガステクスチャに格納される密度情報は 2 種類存在します。1 つはポリゴンオブジェクトとの交差を考慮しない密度情報(D1)で、もう 1 つは交差を考慮する密度情報(D2)です。どちらの密度情報を使用するのかは予約ユニフォーム(dmp_Gas.shadingDensitySrc)に設定する値で選択することができます。

D1 ならば GL_GAS_PLAIN_DENSITY_DMP を、D2 ならば GL_GAS_DEPTH_DENSITY_DMP を glUniform1i() で設定してください。ガスモード時のフォグから出力されるアルファ成分は交差を考慮した密度情報(D2)から算出されていますので、シェーディング参照テーブルへの入力値に D1 を選択したとしても、アルファ値を利用したブレンディングを行うことでポリゴンオブジェクトとガス状物体を正しい前後関係で描画することができます。

参照テーブルへの入力値は 0.0 ～ 1.0 です。密度情報に密度の最大値を逆数にして掛け合わせることで、この範囲に収まる値にします。最大値の逆数は自動的に計算させることができます。予約ユニフォーム(dmp_Gas.autoAcc)に glUniform1i() で GL_TRUE を設定した場合は、密度情報描画パスでの D1 の最大値から逆数を計算します。

GL_FALSE を設定した場合は、予約ユニフォーム(dmp_Gas.accMax)に glUniform1f() で設定した値を最大値の逆数として使用します。

シェーディング強度によるシェーディング

ライトによる陰影の影響を考慮したシェーディング強度を算出してシェーディングを行う場合は、glUniform1i() で予約ユニフォーム(dmp_Gas.colorLutInput)に GL_GAS_LIGHT_FACTOR_DMP を設定し、シェーディング参照テーブルへの入力値にシェーディング強度を選択してください。

シェーディング強度は、平面シェーディング強度(IG)と視線シェーディング強度(IS)と呼ばれる 2 つの算出値の合計です。IG と IS の計算式は以下のように定義されています。

$$ig = r \times (1.0 - lightAtt \times d1)$$
$$IG = (1.0 - ig) \times lightMin + ig \times lightMax$$
$$is = LZ \times (1.0 - scattAtt \times d1)$$
$$IS = (1.0 - is) \times scattMin + is \times scattMax$$

d1 はポリゴンオブジェクトとの交差を考慮していない密度情報(D1)に、密度情報によるシェーディングと同様に密度の最大値の逆数が掛け合わされた値です。r はフォグに入力されたテクスチャコンバイナの出力値の R 成分、lightAtt、lightMin、

lightMax は平面シェーディング強度の係数です。LZ は視点座標系の z 軸に対するライトの方向、scattAtt、scattMin、scattMax は視線シェーディング強度の係数です。これらの値がとる範囲は 0.0 ～ 1.0 です。

計算式から、平面シェーディング強度は r と (1.0 – lightAtt × d1) に比例し、r が大きくなると lightMax に近付き、d1 が大きくなると lightMin に近付くことがわかります。また、視線シェーディング強度は LZ と (1.0 – scattAtt × d1) に比例し、LZ が大きくなると scattMax に近付き、d1 が大きくなると scattMin に近付くことがわかります。

シェーディング参照テーブルに入力されるシェーディング強度が、ガス状物体の密度が低いほど大きくなる点に注意しなければなりません。これはガス状物体の密度の低い部分では光が透過し、密度の高い部分では光が吸収されることを示しています。つまり、lightMin および scattMin はライトの影響が小さいときのシェーディング強度に、lightMax および scattMax はライトの影響が大きいときのシェーディング強度になるように設定します。lightAtt および scattAtt は密度による光の減衰の割合を設定します。シェーディング参照テーブルの設定によっては、必ずしも lightMin ＜ lightMax(scattMin ＜ scattMax)の関係になるとは限らないことに注意してください。そして、アルファ値を決定するフォグ係数への入力値が密度に比例することにも注意が必要です。

平面シェーディング強度の係数(lightMin、lightMax、lightAtt)は予約ユニフォーム(dmp_Gas.lightXY)に、視線シェーディング強度の係数(scattMin、scattMax、scattAtt)と視点座標系の z 軸に対するライトの方向(LZ)は予約ユニフォーム(dmp_Gas.lightZ)に、それぞれまとめて設定します。設定順は、最小値、最大値、減衰値の順で行い、視線シェーディング強度にはその後に LZ を追加して設定します。

以下に、シェーディング強度の係数を設定するコードを示します。

コード 12-19. シェーディング強度の係数設定

```
GLfloat lightXY[3], lightZ[4];
GLfloat lightMin, lightMax, lightAtt;
GLfloat scattMin, scattMax, scattAtt, LZ;

//...

// lightXY
lightXY[0] = lightMin;
lightXY[1] = lightMax;
lightXY[2] = lightAtt;
// lightZ
lightZ[0] = scattMin;
lightZ[1] = scattMax;
lightZ[2] = scattAtt;
lightZ[3] = LZ;
// Set Uniform
glUniform3fv(glGetUniformLocation(progID, "dmp_Gas.lightXY"), 1, lightXY);
glUniform4fv(glGetUniformLocation(progID, "dmp_Gas.lightZ"), 1, lightZ);
```

アルファ値のシェーディング

アルファ成分のシェーディング結果は、交差を考慮した密度情報(D2)とガスの減衰(dmp_Gas.attenuation)とを掛け合わせ、それを入力値とするフォグ係数(dmp_Fog.sampler)の参照テーブルからの出力値で決定されます。

通常、ガスの密度が低いときは 0.0 に、高いときは 1.0 に近付く関数をフォグ係数に指定することになります。

コード 12-20. フォグ係数の設定

```
// Fog Factor
for(int i = 0; i < 128; i++) {
    fogTable[i]= 1.0f - exp(-8.0f * i / 128.0f);
}
for(int i = 0; i < 128; i++) {
    fogTable[128 + i] = fogTable[i + 1] - fogTable[i];
}
fogTable[255] = 0;
// Set LUT
glGenTextures(1, &fogLUT_ID);
glBindTexture(GL_LUT_TEXTURE0_DMP, fogLUT_ID);
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 256, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, fogTable);
```

以下のコードは、シェーディングパスで必要となるユニフォーム設定と、ガステクスチャを貼り付けた Quad ポリゴンの描画の実装例です。

コード 12-21. シェーディングパスのユニフォーム設定の実装例

```
// Bind Framebuffer(Colorbuffer)
glBindFramebuffer(GL_FRAMEBUFFER, renderFboID);
// Bind Gas Texture
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, gasTexID);
glUniform1i(glGetUniformLocation(progID, "dmp_Texture[0].samplerType"),
        GL_TEXTURE_2D);
// Set TextureCombiner #5
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[5].srcRgb"),
        GL_PREVIOUS, GL_PREVIOUS, GL_TEXTURE0);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[5].srcAlpha"),
        GL_PREVIOUS, GL_PREVIOUS, GL_TEXTURE0);
// Set Uniform for Gas Shading Mode
glUniform1i(glGetUniformLocation(progID, "dmp_Fog.sampler"), 0);
glUniform1i(glGetUniformLocation(progID, "dmp_Gas.autoAcc"), GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_Gas.samplerTR"), 1);
glUniform1i(glGetUniformLocation(progID, "dmp_Gas.samplerTG"), 2);
glUniform1i(glGetUniformLocation(progID, "dmp_Gas.samplerTB"), 3);
glUniform1i(glGetUniformLocation(progID, "dmp_Gas.shadingDensitySrc"),
        GL_GAS_DEPTH_DENSITY_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_Gas.colorLutInput"),
        GL_GAS_DENSITY_DMP);
glUniform1f(glGetUniformLocation(progID, "dmp_Gas.accMax"), 1.0f/6.0f);
glUniform4fv(glGetUniformLocation(progID, "dmp_Gas.lightZ"), 1, gasLightZ);
glUniform3fv(glGetUniformLocation(progID, "dmp_Gas.lightXY"), 1, gasLightXY);
// Change to Gas Shading Mode
glUniform1i(glGetUniformLocation(progID, "dmp_FragOperation.mode"),
        GL_FRAGOP_MODE_GL_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_Fog.mode"), GL_GAS_DMP);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glDisable(GL_DEPTH_TEST);
glDepthMask(GL_FALSE);
```

コード 12-22. ガステクスチャを貼り付けた Quad ポリゴンの描画の実装例

```
// Gaseous Shading
// Texture Coord
float u0 = 0.0f;
float v0 = 0.0f;
float u1 = (GAS_ACC_WIDTH * 1.0f) / (GAS_TEX_WIDTH * 1.0f);
float v1 = (GAS_ACC_HEIGHT * 1.0f) / (GAS_TEX_HEIGHT * 1.0f);
GLfloat texCoord[8]= {u0, v0, u0, v1, u1, v1, u1, v0};
// Vertex
GLushort quadIndex[6] = {0, 1, 2, 0, 2, 3};
GLfloat vertex[16] = {
    -1.0f, -1.0f, 0.0f, 1.0f,
    -1.0f, 1.0f, 0.0f, 1.0f,
    1.0f, 1.0f, 0.0f, 1.0f,
    1.0f, -1.0f, 0.0f, 1.0f
};
// Set Array
GLfloat quadIndexID, quadVertexID, quadTexCoordID;
glGenBuffers(1, &quadIndexID);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, quadIndexID);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, 6 * sizeof(GLushort), &quadIndex,
        GL_STATIC_DRAW);
glGenBuffers(1, &quadVertexID);
glBindBuffer(GL_ARRAY_BUFFER, quadVertexID);
glBufferData(GL_ARRAY_BUFFER, 16 * sizeof(GLfloat), &vertex, GL_STATIC_DRAW);
glGenBuffers(1, &quadTexCoordID);
glBindBuffer(GL_ARRAY_BUFFER, quadTexCoordID);
glBufferData(GL_ARRAY_BUFFER, 8 * sizeof(GLfloat), &texCoord, GL_STATIC_DRAW);
// Draw Quad
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, quadVertexID);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
glBindBuffer(GL_ARRAY_BUFFER, quadTexCoordID);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, quadIndexID);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);
```

12.6.4. 予約ユニフォーム

以下の表は、ガスで使用する予約ユニフォームの一覧です。

表 12-16. ガスで使用する予約ユニフォーム

予約ユニフォーム	種別	設定値
dmp_FragOperation.mode	int	フラグメントオペレーションモードを指定する。密度情報描画パスではガスモード、その他のパスでは標準モードを指定する。 GL_FRAGOP_MODE_GL_DMP(標準モード) GL_FRAGOP_MODE_GAS_ACC_DMP(ガスモード)

dmp_Fog.mode	int	フォグパイプラインの処理モードを指定する。シェーディングパスではガスモード、密度情報描画パスでは無効を指定する。 GL_GAS_DMP(ガスモード) GL_FALSE(無効)
dmp_Fog.sampler	int	フォグ係数に使用する参照テーブルを指定する。フォグ係数はガス状物体のアルファ値の算出で使用する。 0 ～ 31
dmp_Gas.deltaZ	float	デプス方向の減衰係数 EZ を指定する。 10.0(デフォルト)
dmp_Gas.autoAcc	bool	密度情報の最大値の逆数を自動的に計算するかどうかを指定する。 GL_TRUE:自動計算する(デフォルト) GL_FALSE:自動計算しない
dmp_Gas.accMax	float	密度情報の最大値の逆数を指定する。 0.0 以上 1.0(デフォルト)
dmp_Gas.samplerTR dmp_Gas.samplerTG dmp_Gas.samplerTB	int	RGB 成分のシェーディング参照テーブルをそれぞれ指定する。 0 ～ 31
dmp_Gas.shadingDensitySrc	int	シェーディングで使用する密度情報を指定する。 GL_GAS_PLAIN_DENSITY_DMP(デフォルト) GL_GAS_DEPTH_DENSITY_DMP
dmp_Gas.colorLutInput	int	シェーディング参照テーブルへの入力値を密度またはシェーディング強度から指定する。 GL_GAS_DENSITY_DMP GL_GAS_LIGHT_FACTOR_DMP(デフォルト)
dmp_Gas.lightXY	vec3	平面シェーディングの制御に使用する、最小強度、最大強度、密度の減衰を指定する。 (lightMin, lightMax, lightAtt) 各制御値とも 0.0 ～ 1.0 (0.0, 0.0, 0.0)(デフォルト)
dmp_Gas.lightZ	vec4	視線シェーディングの制御に使用する、最小強度、最大強度、密度の減衰、視線方向の影響を指定する。 (scattMin, scattMax, scattAtt, LZ) 各制御値とも 0.0 ～ 1.0 (0.0, 0.0, 0.0, 0.0)(デフォルト)
dmp_Gas.attenuation	float	シェーディングのアルファ値の計算で使用する密度減衰の係数を指定する。 0.0 以上 1.0(デフォルト)

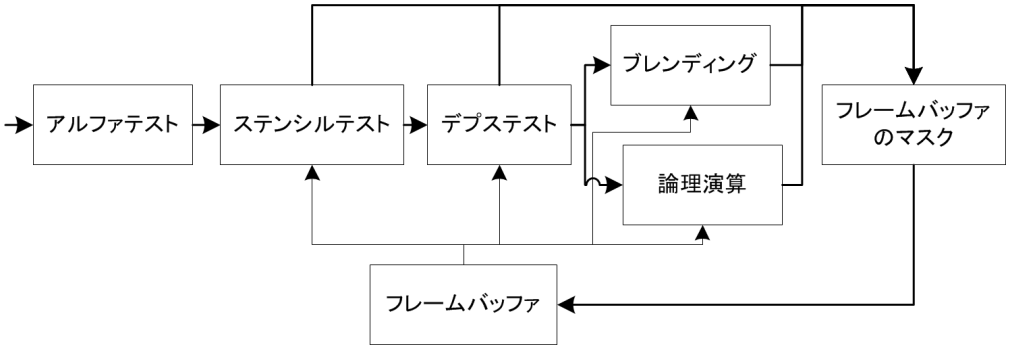
13. パーフラグメントオペレーション

パーフラグメントオペレーションでは、ライティング結果などの関連データをもとに、フラグメントのウィンドウ座標系での位置にあたるフレームバッファ上のピクセルデータを更新します。

フラグメントすべてが考慮されるわけではなく、アルファテストなどのテストで不要なフラグメントを棄却したあと、ブレンディングや論理演算で既存のピクセルデータとのカラー合成が行われます。

フラグメントオペレーションモード(mp_FragOperation.mode)が標準モード(GL_FRAGOP_MODE_GL_DMP)のときは、パーフラグメントオペレーションは図 13-1 の処理の流れとなります。

図 13-1. 標準モードでのパーフラグメントオペレーションの処理の流れ



3DS の仕様と OpenGL ES 2.0 の仕様には、パーフラグメントオペレーションで行われる処理に以下のような相違点があります。

- シザーテストがラスタライズで行われます。
- マルチサンプリングには対応していないため、glSampleCoverage() は実装されていません。
- アルファテストは OpenGL ES 1.1 相当の機能を提供していますが、その制御は予約ユニフォームで行います。
- ステンシルテストはポリゴンの表裏を区別しないため、glStencil*Separate() は実装されていません。
- ブレンディングでは論理演算の演算子(GL_LOGIC_OP)を選択することができません。
- ブレンディングの glBlendFunc*() で、OpenGL ES の仕様ではディスティネーションに選択することのできなかった GL_SRC_ALPHA_SATURATE を選択することができます。
- ディザリングは適用されません。
- 論理演算は OpenGL ES 1.1 相当の仕様で実装されています。

13.1. 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参照値とを比較し、フラグメントを次のプロセスに通過させるのか棄却するのかを決定する機能です。

3DS のアルファテストは OpenGL ES 1.1 のアルファテスト相当の機能を持っていますが、その制御はすべて予約ユニフォームで行われます。

13.1.1. 予約ユニフォーム

アルファテストの予約ユニフォームには、以下のものが存在します。

アルファテストの有効・無効

アルファテスト機能を有効にするには、予約ユニフォーム(dmp_FragOperation.enableAlphaTest)に glUniform1i() で GL_TRUE を設定してください。デフォルトでは無効(GL_FALSE)に設定されています。

比較方法

比較方法は予約ユニフォーム(dmp_FragOperation.alphaTestFunc)に glUniform1i() で設定します。設定することのできる値には、以下の 8 種類があります。

表 13-1. アルファテストの比較方法

設定値	比較内容
GL_NEVER	すべて棄却(すべて通過しない)
GL_ALWAYS(デフォルト)	すべて通過
GL_LESS	参照値より小さければ通過 (alpha < reference)
GL_LEQUAL	参照値以下ならば通過 (alpha <= reference)
GL_EQUAL	参照値と等しければ通過 (alpha == reference)
GL_GEQUAL	参照値以上ならば通過 (alpha >= reference)
GL_GREATER	参照値より大きければ通過 (alpha > reference)
GL_NOTEQUAL	参照値と等しくなければ通過 (alpha != reference)

参照値

アルファテストの参照値は予約ユニフォーム(dmp_FragOperation.alphaRefValue)に glUniform1f() で設定します。参照値は比較時に 0.0 ～ 1.0 の範囲にクランプされて使用されます。

以下の表は、アルファテストで使用する予約ユニフォームの一覧です。

表 13-2. アルファテストで使用する予約ユニフォーム

予約ユニフォーム	種別	設定値
dmp_FragOperation.enableAlphaTest	bool	アルファテストの有効／無効を指定する。 GL_TRUE GL_FALSE(デフォルト)
dmp_FragOperation.alphaRefValue	float	アルファテストで使用する参照値を指定する。 0.0 ～ 1.0 (デフォルト)0.0
dmp_FragOperation.alphaTestFunc	int	アルファテストの比較方法を指定する。 GL_NEVER GL_ALWAYS(デフォルト) GL_LESS GL_LEQUAL GL_EQUAL GL_GEQUAL GL_GREATER GL_NOTEQUAL

13.2. ステンシルテスト

ステンシルテストは、フラグメントのウィンドウ座標系での座標にあるステンシルバッファの内容と参照値とを比較し、フラグメントを次のプロセスに通過させるのか棄却するのかを決定する機能です。

3DS では、ポリゴンの表裏を区別しないため、`glStencilFuncSeparate()` と `glStencilOpSeparate()` は実装されていません。

13.2.1. 使用方法

ステンシルテストの使用方法は OpenGL と同じです。

ステンシルテストの有効・無効

ステンシルテストの有効・無効の制御は、*cap* に `GL_STENCIL_TEST` を渡して `glEnable()` または `glDisable()` を呼び出すことで行います。現在の設定を取得するには、*cap* に `GL_STENCIL_TEST` を渡して `glIsEnabled()` を呼び出してください。デフォルトではステンシルテストは無効になっています。無効に設定されている場合、ステンシルバッファの変更やフラグメントの棄却は行われません。フレームバッファにステンシルバッファが関連付けられていない場合、ステンシルテストは無効に設定されているものとして処理されます。ステンシルテストとデプステストは、どちらも有効・どちらか一方が有効のいずれの場合でも、性能の差はありません。

比較方法、参照値、マスク

ステンシルテストは `glStencilFunc()` で指定された比較方法、参照値、マスクの 3 要素をもとに行われます。

コード 13-1. glStencilFunc() の定義

```
void glStencilFunc(GLenum func, GLint ref, GLuint mask);
```

ref には比較で使用する参照値を整数で指定します。参照値は、ステンシルテストの比較時に符号なしの数値として扱われ、0 ～ 255（3DS のステンシルバッファが 8 bit 固定であるため）の範囲にクランプされます。デフォルトでは 0 が使用されます。

mask にはマスキング値を指定します。比較時の参照値とステンシルバッファの内容には、マスキング値とビット単位での論理積を適用した値が使用されます。デフォルトでは 0xFFFFFFFF（全ビットが 1）が使用されます。

func には比較方法を設定します。設定することのできる値には、以下の 8 種類があります。

表 13-3. ステンシルテストの比較方法

設定値	比較内容
GL_NEVER	すべて棄却（すべて通過しない）
GL_ALWAYS（デフォルト）	すべて通過
GL_LESS	参照値がバッファの内容より小さければ通過 (reference < stencil buffer)
GL_LEQUAL	参照値がバッファの内容以下ならば通過 (reference <= stencil buffer)
GL_EQUAL	参照値がバッファの内容と等しければ通過 (reference == stencil buffer)
GL_GEQUAL	参照値がバッファの内容以上ならば通過 (reference >= stencil buffer)
GL_GREATER	参照値がバッファの内容より大きければ通過 (reference > stencil buffer)
GL_NOTEQUAL	参照値がバッファの内容と等しくなければ通過 (reference != stencil buffer)

テスト結果への対処

ステンシルテストでは、参照値とステンシルバッファの比較結果だけでなく、後述するデプステストの結果によってもステンシルバッファに対して値をどのように変化させるのかを glStencilOp() で設定することができます。

コード 13-2. glStencilOp() の定義

```
void glStencilOp(GLenum fail, GLenum zfail, GLenum zpass);
```

fail は、ステンシルテストの結果がフラグメントの棄却であった場合に、ステンシルバッファの内容をどのように変化させるのかを設定します。設定することのできる値には、以下の 8 種類があります。

表 13-4. ステンシルバッファの内容の変化

設定値	ステンシルバッファの内容
GL_KEEP(デフォルト)	そのままの値で変化しません。
GL_ZERO	0 に設定します。
GL_REPLACE	参照値に置き換えます。
GL_INCR	1 増加させます。255 を超えることはありません。
GL_DECR	1 減少させます。0 未満になることはありません。
GL_INVERT	ビット反転させます。
GL_INCR_WRAP	1 増加させます。255 の次は 0 になります。
GL_DECR_WRAP	1 減少させます。0 の次は 255 になります。

zfail はデプステストでフラグメントが棄却されたとき、*zpass* はデプステストでフラグメントが通過したときにステンシルバッファの内容がどのように変化するのかを設定します。設定することのできる値は *fail* と同じです。

13.3. デプステスト

デプステストは、フラグメントのデプス値とフラグメントのウィンドウ座標系での座標にあるデプスバッファの内容とを比較し、フラグメントを次のプロセスに通過させるのか棄却するのかを決定する機能です。ステンシルバッファでも述べたように、デプステストの結果でステンシルバッファの内容を変化させることもできます。

13.3.1. 使用方法

デプステストの使用方法は OpenGL と同じです。

デプステストの有効・無効

デプステストの有効・無効の制御は、*cap* に GL_DEPTH_TEST を渡して glEnable() または glDisable() を呼び出すことで行います。現在の設定を取得するには、*cap* に GL_DEPTH_TEST を渡して glIsEnabled() を呼び出してください。デフォルトではデプステストは無効になっています。無効に設定されている場合、デプスバッファの変更やフラグメントの棄却は行われません。フレームバッファにデプスバッファが関連付けられていない場合、デプステストは無効に設定されているものとして処理されます。デプステストとステンシルテストは、どちらも有効・どちらか一方が有効のいずれの場合でも、性能の差はありません。

比較方法

比較方法は glDepthFunc() で設定します。

コード 13-3. glDepthFunc() の定義

```
void glDepthFunc(GLenum func);
```

func に設定することのできる値には、以下の 8 種類があります。

表 13-5. デプステストの比較方法

設定値	比較内容
GL_NEVER	すべて棄却(すべて通過しない)
GL_ALWAYS	すべて通過
GL_LESS(デフォルト)	デプス値がバッファの内容より小さければ通過 (fragment depth < depth buffer)
GL_LEQUAL	デプス値がバッファの内容以下ならば通過 (fragment depth <= depth buffer)
GL_EQUAL	デプス値がバッファの内容と等しければ通過 (fragment depth == depth buffer)
GL_GEQUAL	デプス値がバッファの内容以上ならば通過 (fragment depth >= depth buffer)
GL_GREATER	デプス値がバッファの内容より大きければ通過 (fragment depth > depth buffer)
GL_NOTEQUAL	デプス値がバッファの内容と等しくなければ通過 (fragment depth != depth buffer)

比較の結果、フラグメントが棄却されるときはデプスバッファに変化はありません。フラグメントが通過するときはデプスバッファの内容がフラグメントのデプス値で上書きされます。

13.4. ブレンディング

ブレンディングは、フラグメントの持つカラー(ソースカラー)とフラグメントのウィンドウ座標系での座標にあるフレームバッファのカラー(ディスティネイションカラー)を合成する機能です。ブレンディングの結果は、フラグメントの持つカラーとして次のプロセスに渡されます。

13.4.1. 使用方法

ブレンディングの使用方法は OpenGL と同じです。

ブレンディングの有効・無効

ブレンディングの有効・無効の制御は、*cap* に GL_BLEND を渡して glEnable() または glDisable() を呼び出すことで行います。現在の設定を取得するには、*cap* に GL_BLEND を渡して glIsEnabled() を呼び出してください。デフォルトではブレンディングは無効になっています。無効に設定されている場合、ブレンディングは行われません。

フレームバッファにカラーバッファが関連付けられていない場合、または後述する論理演算が有効になっている場合、ブレンディングは無効に設定されているものとして処理されます。

ブレンディングの計算式(合成方法)

フラグメントの持つカラー(ソースカラー)とフレームバッファのカラー(ディスティネーションカラー)をどのように合成するかは、glBlendEquation() または glBlendEquationSeparate() で設定することができます。

コード 13-4. glBlendEquation*() の定義

```
void glBlendEquation(GLenum mode);
void glBlendEquationSeparate(GLenum modeRGB, GLenum modeAlpha);
```

glBlendEquation() は、*mode* で RGB 成分とアルファ成分の計算式をまとめて設定することができます。

glBlendEquationSeparate() は、*modeRGB* で RGB 成分の、*modeAlpha* でアルファ成分の計算式を個別に設定することができます。それぞれに設定する値は共通で、以下の 5 種類があります。

表 13-6. ブレンディングの計算式

設定値	RGB 成分	アルファ成分
GL_FUNC_ADD(デフォルト)	R = Rs * Sr + Rd * Dr G = Gs * Sg + Gd * Dg B = Bs * Sb + Bd * Db	A = As * Sa + Ad * Da
GL_FUNC_SUBTRACT	R = Rs * Sr – Rd * Dr G = Gs * Sg – Gd * Dg B = Bs * Sb – Bd * Db	A = As * Sa – Ad * Da
GL_FUNC_REVERSE_SUBTRACT	R = Rd * Sr – Rs * Dr G = Gd * Sg – Gs * Dg B = Bd * Sb – Bs * Db	A = Ad * Sa – As * Da
GL_MIN	R = min(Rs, Rd) G = min(Gs, Gd) B = min(Bs, Bd)	A = min(As, Ad)
GL_MAX	R = max(Rs, Rd) G = max(Gs, Gd) B = max(Bs, Bd)	A = max(As, Ad)

Rs, Gs, Bs, As :ソースカラーの各成分
Rd, Gd, Bd, Ad :ディスティネーションカラーの各成分
Sr, Sg, Sb, Sa :ソースの重み係数
Dr, Dg, Db, Da :ディスティネーションの重み係数

OpenGL ES 2.0 では指定可能となっている GL_LOGIC_OP には対応していません。

ソースとディスティネーションの重み係数

ソースとディスティネーションに適用される重み係数は glBlendFunc() または glBlendFuncSeparate() で設定することができます。

コード 13-5. glBlendFunc*() の定義

```
void glBlendFunc(GLenum sfactor, GLenum dfactor);
void glBlendFuncSeparate(GLenum srcRGB, GLenum dstRGB, GLenum srcAlpha,
                         GLenum dstAlpha);
```

glBlendFunc() は、*sfactor* と *dfactor* でソースとディスティネーションの重み係数をまとめて設定することができます。glBlendFuncSeparate() は、*srcRGB* と *srcAlpha* でソースの RGB 成分とアルファ成分の重み係数を、*dstRGB* と *dstAlpha* でディスティネーションの RGB 成分とアルファ成分の重み係数を個別に設定することができます。それぞれに設定する値は共通で、以下の 15 種類があります。

表 13-7. ソースとディスティネーションの重み係数

設定値	カラーの重み係数 (Sr, Sg, Sb) or (Dr, Dg, Db)	アルファの重み係数 Sa or Da
GL_ZERO(デフォルト *dstRGB*, *dstAlpha*)	(0, 0, 0)	0
GL_ONE(デフォルト *srcRGB*, *srcAlpha*)	(1, 1, 1)	1
GL_SRC_COLOR	(Rs, Gs, Bs)	As
GL_ONE_MINUS_SRC_COLOR	(1, 1, 1) – (Rs, Gs, Bs)	1 – As
GL_DST_COLOR	(Rd, Gd, Bd)	Ad
GL_ONE_MINUS_DST_COLOR	(1, 1, 1) – (Rd, Gd, Bd)	1 – Ad
GL_SRC_ALPHA	(As, As, As)	As
GL_ONE_MINUS_SRC_ALPHA	(1, 1, 1) – (As, As, As)	1 – As
GL_DST_ALPHA	(Ad, Ad, Ad)	Ad
GL_ONE_MINUS_DST_ALPHA	(1, 1, 1) – (Ad, Ad, Ad)	1 – Ad
GL_CONSTANT_COLOR	(Rc, Gc, Bc)	Ac
GL_ONE_MINUS_CONSTANT_COLOR	(1, 1, 1) – (Rc, Gc, Bc)	1 – Ac
GL_CONSTANT_ALPHA	(Ac, Ac, Ac)	Ac
GL_ONE_MINUS_CONSTANT_ALPHA	(1, 1, 1) – (Ac, Ac, Ac)	1 – Ac
GL_SRC_ALPHA_SATURATE	(f, f, f)	1

Rs, Gs, Bs, As :ソースカラー
Rd, Gd, Bd, Ad :ディスティネーションカラー
Rc, Gc, Bc, Ac :定数カラー
f = min(As, 1 – Ad)

3DS では、ディスティネーションにも GL_SRC_ALPHA_SATURATE を設定することができます。

定数カラー

定数カラーは glBlendColor() で設定することができます。

コード 13-6. glBlendColor() の定義

```
void glBlendColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);
```

red、*green*、*blue*、*alpha* で RGB とアルファの各成分を指定します。各成分値は 0.0 ～ 1.0 の浮動小数点数で指定します。デフォルトでは、すべて 0.0(0.0, 0.0, 0.0, 0.0)に設定されています。

13.5. 論理演算

論理演算は、フラグメントのカラーとフレームバッファのカラーに対して、パーフラグメントオペレーションの最後に適用される、画像の論理演算を行う機能です。論理演算の結果は、ウィンドウ座標系でのフラグメントの座標に対応するフレームバッファに書き込まれます。

13.5.1. 使用方法

論理演算の使用方法は OpenGL ES 1.1 と同じです。

論理演算の有効・無効

論理演算の有効・無効の制御は、*cap* に GL_COLOR_LOGIC_OP を渡して glEnable() または glDisable() を呼び出すことで行います。現在の設定を取得するには、*cap* に GL_COLOR_LOGIC_OP を渡して glIsEnabled() を呼び出してください。デフォルトでは論理演算は無効になっています。無効に設定されている場合、論理演算は行われませんが、フレームバッファにはフラグメントのカラーが書き込まれます。論理演算を有効に設定すると、ブレンディングは無効となります。

演算方法

フラグメントのカラー(ソース)とフレームバッファのカラー(ディスティネーション)をどのように論理演算するのかは、glLogicOp() で設定します。現在の設定を取得するには、*pname* に GL_LOGIC_OP_MODE を渡して glGetIntegerv() を呼び出してください。デフォルトでは GL_COPY が設定されています。

コード 13-7. glLogicOp() の定義

```
void glLogicOp(GLenum opcode);
```

opcode には、RGB とアルファの各成分に対する演算方法をまとめて指定します。

表 13-8. 論理演算の演算方法

設定値	演算方法	C スタイルでの表記
GL_CLEAR	0	0
GL_AND	s ∧ d	s & d
GL_AND_REVERSE	s ∧ ¬ d	s & ~d
GL_COPY(デフォルト)	s	s
GL_AND_INVERTED	¬ s ∧ d	~s & d
GL_NOOP	d	d
GL_XOR	s xor d	s ^ d
GL_OR	s ∨ d	s | d
GL_NOR	¬ (s ∨ d)	~(s | d)
GL_EQUIV	¬ (s xor d)	~(s ^ d)
GL_INVERT	¬ d	~d
GL_OR_REVERSE	s ∨ ¬ d	s | ~d
GL_COPY_INVERTED	¬ s	~s
GL_OR_INVERTED	¬ s ∨ d	~s | d
GL_NAND	¬ (s ∧ d)	~(s & d)
GL_SET	all 1	pow(2, *n*) - 1

n は各成分でのビット数

13.6. フレームバッファのマスク

パーフラグメントオペレーションで行われる、フレームバッファへのカラー(RGBA)、ステンシル、デプスの書き込みに対してマスキングを行うことができます。それぞれ、glColorMask()、glStencilMask()、glDepthMask() で設定することができます。

コード 13-8. フレームバッファのマスキング関数

```
void glColorMask(GLboolean red, GLboolean green, GLboolean blue,
                 GLboolean alpha);
void glStencilMask(GLuint mask);
void glDepthMask(GLboolean flag);
```

カラーの RGBA 成分ごととデプス値に対しては、書き込みを許可する(GL_TRUE)か、許可しない(GL_FALSE)かを指定することができます。デフォルトでは、どちらも GL_TRUE が設定されています。

ステンシルに対しては、ステンシルのマスキング値を変更することができます。このマスキング値は、ステンシルテストのマスキング値とは独立した設定です。デフォルトでは 0xFFFFFFFF(全ビットが 1)が設定されています。

14. フレームバッファオペレーション

フレームバッファオペレーションでは、フレームバッファに対する読み込み、コピーなどの処理が行われます。

3DS では、対応している機能が以下のように制限されています。

- glDrawBuffer() は実装されていません。
- glReadBuffer() は実装されていません。
- glDrawPixels() は実装されていません。
- glReadPixels() はカラーバッファ、デプスバッファ、ステンシルバッファの内容を取得することができます。
- glCopyPixels() は実装されていません。
- glPixelStore*() は実装されていません。

14.1. リードピクセル

フレームバッファに関連付けられているレンダーバッファ(カラーバッファ、デプスバッファ、ステンシルバッファ)の内容を、glReadPixels() でメモリに取り込むことができます。デプスバッファとステンシルバッファの内容を複合フォーマットで取り込むこともできます。

コード 14-1. glReadPixels() の定義

```
void glReadPixels(GLint x, GLint y, GLsizei width, GLsizei height,
                  GLenum format, GLenum type, void* pixels);
```

x と *y* には取り込む矩形領域の始点(左下)座標を指定します。*width* と *height* には矩形領域の幅と高さを指定します。これらの値に負値を指定すると GL_INVALID_VALUE エラーが生成されます。(*x* + *width*) がレンダーバッファの幅を超える、または (*y* + *height*) がレンダーバッファの高さを超えるような矩形領域を指定した場合は GL_INVALID_OPERATION エラーが生成されます。

format と *type* の組み合わせで、取り込むイメージのフォーマットを指定することができます。

表 14-1. 取り込み時に指定することのできるフォーマット

format	type	フォーマット	ビット数
GL_RGBA	GL_UNSIGNED_BYTE	RGBA8	32
	GL_UNSIGNED_SHORT_4_4_4_4	RGBA4	16
	GL_UNSIGNED_SHORT_5_5_5_1	RGBA5551	16
GL_RGB	GL_UNSIGNED_BYTE	RGB8	24
	GL_UNSIGNED_SHORT_5_6_5	RGB565	16
GL_BGRA	GL_UNSIGNED_BYTE	BGRA8	32
	GL_UNSIGNED_SHORT_4_4_4_4	BGRA4	16
	GL_UNSIGNED_SHORT_5_5_5_1	BGRA5551	16
GL_LUMINANCE_ALPHA	GL_UNSIGNED_BYTE	LA8	16
GL_LUMINANCE	GL_UNSIGNED_BYTE	L8	8

GL_ALPHA	GL_UNSIGNED_BYTE	A8	8
GL_DEPTH_COMPONENT	GL_UNSIGNED_INT	Depth	32
	GL_UNSIGNED_INT24_DMP	Depth	24
	GL_UNSIGNED_SHORT	Depth	16
	GL_UNSIGNED_BYTE	Depth	8
GL_STENCIL_INDEX	GL_UNSIGNED_BYTE	Stencil	8
GL_DEPTH24_STENCIL8_EXT	GL_UNSIGNED_INT	Depth + Stencil	32

上表で示したもの以外の組み合わせを指定した場合は GL_INVALID_ENUM エラーが生成されます。

pixels には取り込んだイメージが指定されたフォーマットで格納されます。

カラーバッファから取り込んだイメージはリニアフォーマットで格納され、ピクセルデータのバイトオーダーは OpenGL 準拠となっています。

デプスバッファから取り込んだイメージもリニアフォーマットで格納されますが、ピクセルデータは下位ビットから順に並びます。例えば、24 ビットのフォーマットで取り込んだ場合は、下位 8 ビット、中位 8 ビット、上位 8 ビットの順にメモリに格納されます。

ステンシルバッファから取り込んだイメージはリニアフォーマットで格納され、1 バイトが 1 ピクセルに対応しています。

デプスバッファとステンシルバッファから複合フォーマットで取り込んだイメージもリニアフォーマットで格納されますが、ピクセルデータは下位ビットから順に並び、デプス値の下位 8 ビット、中位 8 ビット、上位 8 ビット、ステンシル値の 8 ビットの順にメモリに格納されます。

フレームバッファにカラーバッファが関連付けられていないなど、フレームバッファに関連するエラーが発生した場合は GL_INVALID_FRAMEBUFFER_OPERATION エラーが生成されます。ライブラリ内で一時メモリの確保に失敗した場合は GL_OUT_OF_MEMORY エラーが生成されます。

14.2. コピーピクセル

glCopyPixels() による、ほかのフレームバッファへのコピーには対応していません。

glCopyTexImage2D() や glCopyTexSubImage2D() を呼び出すことでカラーバッファの内容をテクスチャにコピーすることができます。詳しくは「7.5.1. カラーバッファからのコピー」を参照してください。

14.3. テクスチャレンダリング

テクスチャをレンダーバッファとしてフレームバッファに関連付けることができます。詳しくは「7.6. レンダーターゲットへのテクスチャの指定」を参照してください。

14.4. フレームバッファのクリア

OpenGL では glClear() によるフレームバッファ（関連付けられている各種バッファ）のクリアにシザーテストやマスキング処理が影響しますが、3DS ではグラフィックス処理のパイプラインに含まれていないために影響を受けません。

フレームバッファに関連付けることのできるレンダーバッファは、「3.3.1. レンダーバッファの確保」にもあるように、カラーバッファ（ガスレンダリング用のバッファやテクスチャを含む）とデプスバッファ、ステンシルバッファの 3 つです。インデックスカ

ラーには対応していません。

バッファのクリアは、glClear() の引数にクリアするバッファに対応するビットを指定して呼び出すことで行われます。

コード 14-2. glClear() の定義

```
void glClear(GLbitfield mask);
```

mask に指定するビットフィールドには、以下のものから選択します。複数のバッファを同時にクリアする場合は、ビットフィールドの論理和を指定してください。

表 14-2. クリアするバッファの指定

mask	クリアされるバッファ
GL_COLOR_BUFFER_BIT	カラーバッファ
GL_DEPTH_BUFFER_BIT	デプスバッファ
GL_STENCIL_BUFFER_BIT	ステンシルバッファ。GL_DEPTH_BUFFER_BIT と同時に指定しなければなりません。

3DS のステンシルバッファはデプスバッファと複合したバッファ(GL_DEPTH24_STENCIL8_EXT)のため、ステンシルバッファのみをクリアすることはできません。

14.4.1. クリアカラー

カラーバッファをクリアする際に使用されるカラーを glClearColor() で指定することができます。

コード 14-3. クリアカラーの指定

```
void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);
```

指定されたカラーの各成分は 0.0 ～ 1.0 の範囲にクランプされます。デフォルトでは、各成分とも 0.0 が設定されています。

14.4.2. クリアデプス

デプスバッファをクリアする際に使用されるデプス値を glClearDepthf() で指定することができます。

コード 14-4. クリアデプスの指定

```
void glClearDepthf(GLclampf depth);
```

指定された値は 0.0 ～ 1.0 の範囲にクランプされます。デフォルトでは 1.0 が設定されています。

14.4.3. クリアステンシル

ステンシルバッファをクリアする際に使用される値を glClearStencil() で指定することができます。

コード 14-5. クリアステンシルの指定

```
void glClearStencil(GLint s);
```

クリアに使用する値を 0 ～ 255 の範囲で指定します。デフォルトでは 0 が設定されています。

15. その他

15.1. glFinish() と glFlush()

3DS では、glFinish() と glFlush() に違いはなく、どちらも glFlush() を呼び出したときの処理を行います。

コード 15-1. glFinish()、glFlush() の定義

```
void glFinish(void);
void glFlush(void);
```

15.2. OpenGL ES との違い

ほとんどの機能と関数は OpenGL ES の仕様と同じですが、一部の機能や関数には実装されていない、もしくは機能が制限されているなどの違いがあります。

表 15-1. 機能ごとの相違点

機能	相違点
GL_ALPHA_TEST	アルファテストの制御には予約ユニフォームを使用します。
GL_CLIP_PLANEi	クリッピングの制御には予約ユニフォームを使用します。
GL_COLOR_LOGIC_OP	論理演算は OpenGL ES 2.0 では廃止されていますが、3DS では OpenGL ES 1.1 相当の機能を実装しています。
GL_DITHER	ディザリングには対応していません。
GL_FOG	フォグの制御には予約ユニフォームを使用します。フォグ係数は視点からの距離ではなく、ウィンドウ座標系でのデプス値を入力にとり、参照テーブルで設定します。
GL_INDEX_LOGIC_OP	インデックスカラーには対応していません。
GL_LIGHTING	ライティングの制御には予約ユニフォームを使用します。3DS のライティングはフラグメント単位で行われます。
GL_POLYGON_OFFSET_FILL GL_POLYGON_OFFSET_LINE GL_POLYGON_OFFSET_POINT	ポリゴンオフセットの制御には GL_POLYGON_OFFSET_FILL のみが対応しています。
GL_SAMPLE_COVERAGE GL_SAMPLE_ALPHA_TO_COVERAGE	マルチサンプリングには対応していません。
GL_SCISSOR_TEST	シザーテストはラスタライズのサブプロセスとして実行されます。
GL_TEXTURE_2D GL_TEXTURE_CUBE_MAP	テクスチャユニットの制御には予約ユニフォームを使用します。

表 15-2. 関数ごとの相違点

関数	相違点
シェーダ関連	
glBufferData()	*usage* には GL_STATIC_DRAW のみが指定可能です。

glCompileShader()	実装されていません。
glCreateProgram()	シェーダオブジェクトとは独立した名前空間(13 bit)を使用します。
glCreateShader()	プログラムオブジェクトとは独立した名前空間を使用します。
glDrawArrays() glDrawElements()	GL_POINTS、GL_LINES、GL_LINE_STRIP、GL_LINE_LOOP には対応していません。ポイントおよびラインの描画にはジオメトリシェーダを使用する必要があります。
glGetProgramInfoLog()	実装されていません。
glGetShaderInfoLog()	実装されていません。
glGetShaderSource()	実装されていません。
glGetShaderPrecisionFormat()	実装されていません。
glLineWidth()	実装されていません。
glReleaseShaderCompiler()	実装されていません。
glShaderBinary()	*binaryformat* には GL_PLATFORM_BINARY_DMP のみが指定可能です。
glShaderSource()	実装されていません。
glValidateProgram()	呼び出しても何も行われません。
glVertexAttribPointer()	*type* に GL_FIXED、GL_UNSIGNED_SHORT を指定することができません。データポインタは、*type* が GL_FLOAT のときは 4 Byte、GL_SHORT のときは 2 Byte のアライメントになっていなければなりません。引数指定による正規化には対応していません。
ビューポート関連	
glViewport()	*x*, *y* に負の値を指定することができません。
テクスチャ関連	
glActiveTexture()	GL_TEXTURE3 を指定するとエラーを生成します。
glCompressedTexImage2D()	幅と高さは 2 のべき乗(16 ～ 1024)で指定しなければなりません。
glCompressedTexSubImage2D()	実装されていません。
glGenerateMipmap()	実装されていません。
glTexImage1D() glTexSubImage1D()	参照テーブルのロードに使用します。1 次元テクスチャには対応していません。
glTexImage2D()	幅と高さは 2 のべき乗(8 ～ 1024)で指定しなければなりません。
glTexSubImage2D()	実装されていません。
レンダーバッファ関連	
glClear()	インデックスカラーには対応していません。ステンシルバッファはデプスバッファと同時にクリアする必要があります。シザーテストとマスキング処理の影響を受けません。
glCopyPixels()	実装されていません。
glDrawBuffer()	実装されていません。
glDrawPixels()	実装されていません。

glFramebufferRenderbuffer()	ステンシルバッファがデプスバッファと複合しているため、ステンシルバッファのアタッチメントには GL_DEPTH_STENCIL_ATTACHMENT を指定しなければなりません。
glFramebufferTexture2D()	アタッチメントに対応しているのはカラーのみです。
glPixelStorei()	実装されていません。
glReadBuffer()	実装されていません。
glRenderbufferStorage()	指定可能なフォーマットが制限されています。ステンシルバッファを利用することができるフォーマットは GL_DEPTH24_STENCIL8_EXT のみです。
glSampleCoverage()	実装されていません。
glStencilFuncSeparate() glStencilOpSeparate() glStencilMaskSeparate()	実装されていません。
ブレンディング関連	
glBlendEquation() glBlendEquationSeparate()	OpenGL ES 2.0 では指定可能となっている GL_LOGIC_OP には対応していません。
glBlendFunc() glBlendFuncSeparate()	ディスティネーションに対しても GL_SRC_ALPHA_SATURATE を指定することができます。
その他	
glFinish()	glFlush() と同じ処理を行います。
glFlush()	glFinish() との間に処理の違いはありません。
glHint()	実装されていません。
glPolygonOffset()	*factor* の設定は無視されます。

15.3. バッファアクセスのパフォーマンス改善方法

カラーバッファ、デプスバッファ、ステンシルバッファのいずれかを使用しない場合は、各バッファの機能を明示的に無効にして不要な処理を減らすことができます。ただし、3DS のデプスバッファとステンシルバッファは同一のバッファを使用するため、デプスバッファまたはステンシルバッファのどちらかにアクセスするような設定は両方にアクセスする設定と同じパフォーマンスになります。

各バッファにアクセスする条件は以下のようになっています。無駄なアクセスが発生しないように、以下の条件が成立しないように注意してください。

15.3.1. カラーバッファ への Write アクセス

glColorMask() によりいずれかの成分が GL_TRUE に設定されている場合はアクセスが発生します。

15.3.2. カラーバッファへの Read アクセス

カラーバッファへの Write アクセスが発生し、かつ以下の条件のいずれかを満たしている場合はアクセスが発生します。

- GL_BLEND が glEnable() によって有効に設定されている。
- glColorMask() による設定が、すべての成分に対して同じ設定になっていない。
- GL_COLOR_LOGIC_OP が glEnable() によって有効に設定されている。

15.3.3. デプスバッファへの Write アクセス

GL_DEPTH_TEST が glEnable() によって有効に設定されている、かつ glDepthMask() に GL_TRUE が設定されている場合はアクセスが発生します。

15.3.4. デプスバッファへの Read アクセス

GL_DEPTH_TEST が glEnable() によって有効に設定されている場合はアクセスが発生します。

15.3.5. ステンシルバッファへの Write アクセス

GL_STENCIL_TEST が glEnable() によって有効に設定されている、かつ glStencilMask() でのマスキング設定が 0 でない場合はアクセスが発生します。

15.3.6. ステンシルバッファへの Read アクセス

GL_STENCIL_TEST が glEnable() によって有効に設定されている場合はアクセスが発生します。

15.4. CPU パフォーマンスの改善方法

以下の点に注意してアプリケーションを実装することで、処理速度が向上する場合があります。

- なるべく多くの頂点シェーダアセンブラをリンクして使用してください。複数のシェーダアセンブラをリンクして 1 つのシェーダバイナリとして生成することができます。シェーダオブジェクトを切り替える際、異なるシェーダバイナリにリンクされたシェーダオブジェクトよりも、同じシェーダバイナリにリンクされたシェーダオブジェクトに切り替える方が処理は軽くなります。
- glGetUniformLocation() で取得したユニフォームのロケーション値をアプリケーションで保持し、繰り返し使用してください。ロケーション値は glLinkProgram() の呼び出し後に決定され、再度 glLinkProgram() を呼び出すまで変更されることはありません。
- 呼び出されるごとに区切りコマンドを生成する nngxSplitDrawCmdlist() を不要なタイミングで呼び出さないでください。例えば、nngxTransferRenderImage() を呼び出すと関数内で区切りコマンドが生成されますので、直後に nngxSplitDrawCmdlist() を呼び出すのは無駄なコマンドを生成することになります。
- 頂点属性情報をシェーダに送るときには、なるべく頂点バッファを使用してください。頂点バッファを使用しない場合は CPU が 3D コマンドバッファに頂点データを蓄積していくため、頂点バッファを使用する場合に比べて CPU での処理が著しく増加します。
- 特定の描画パスなどで同じテクスチャや頂点バッファを繰り返し使用する場合は、テクスチャコレクションや頂点ステートコレクションを使用してください。テクスチャのバインドや頂点アレイの設定を一括して行うことで、関数の呼び出しコストを減らすことができます。
- 同じシェーダオブジェクトを異なるユニフォームの設定で実行する場合は、同じシェーダオブジェクトをアタッチさせた複数のプログラムオブジェクトに異なるユニフォームの設定をしておき、プログラムオブジェクトを切り替えて実行する方が、1 つのプログラムオブジェクトで多数のユニフォームの設定切り替えを行うよりも処理が軽くなる場合があります。これは、ユニフォームの値がプログラムオブジェクトごとに保存されているためです。
- 参照テーブルのオブジェクトは頻繁に削除・再生成を行わないでください。これは、glTexImage1D() で参照テーブルのデータがロードされると、参照テーブルのオブジェクトの使用時にハードウェア内部フォーマットへの変換が行われるためです。

15.5. 頂点バッファの使用について

頂点バッファを使用した場合は GPU のジオメトリ処理パイプラインが頂点データをロードしますが、使用しない場合は CPU が頂点インデックスアレイに従って頂点アレイの並び替えを行い、すべての頂点データを 24 ビットの浮動小数点数値に変換してからコマンドバッファに詰め込みます。この処理は CPU にとって著しく負荷が高い上に、GPU のジオメトリ処理パイプラインによるデータロードの効率も低下してしまいます。また、より大きなコマンドバッファのサイズが必要となります。頂点データをコマンドバッファに詰め込む場合、すべての頂点データは xyzw の 4 コンポーネント×24 ビット浮動小数点数に変換されますので、頂点数×頂点属性数×12 バイトのサイズが必要となります。

頂点バッファを VRAM に配置した場合は、メインメモリ(デバイスメモリ)に配置した場合に比べて高速に処理することができます。VRAM とメインメモリに分けて配置した場合の速度は、メインメモリに配置した場合の速度と同じです。

15.5.1. 頂点アレイのデータ構成

頂点アレイには、複数の頂点属性を含んだ構造体の配列として頂点データを配置するインターリーブドアレイと、1 つの頂点属性の配列として頂点データを配置する独立アレイの 2 種類のデータ構成があります。

頂点バッファを使用する場合、頂点アレイは独立アレイよりもインターリーブドアレイにした方が頂点データを読み込む効率が良くなります。頂点データの読み込み処理時間自体は、そのあとに行われる頂点シェーダやラスタライズなどの処理時間により隠蔽されることが多いのですが、頂点バッファをメインメモリに配置している場合は、データの読み込み効率を良くすることでアクセス負荷を減らすことができるため、結果として処理の高速化に貢献できる場合があります。

15.6. 各種バッファの先頭アドレスの取得

テクスチャオブジェクト、頂点バッファオブジェクト、レンダーバッファオブジェクトの各オブジェクトのために確保されているデータ領域のアドレスを取得することができます。

取得したアドレスはすべて、GPU が直接アクセスを行う領域のアドレスです。ドライバが生成するコピー領域のアドレスを取得することはできません。

コード 15-2. 各種バッファの先頭アドレスの取得

```
void glGetTexParameteriv(GLenum target, GLenum pname, GLint* params);
void glGetBufferParameteriv(GLenum target, GLenum pname, GLint* params);
void glGetRenderbufferParameteriv(GLenum target, GLenum pname, GLint* params);
```

テクスチャのアドレスは、`glGetTexParameteriv()` の *pname* に `GL_TEXTURE_DATA_ADDR_DMP` を指定して呼び出すことで取得することができます。

頂点バッファのアドレスは、`glGetBufferParameteriv()` の *pname* に `GL_BUFFER_DATA_ADDR_DMP` を指定して呼び出すことで取得することができます。

レンダーバッファのアドレスは、`glGetRenderbufferParameteriv()` の *pname* に `GL_RENDERBUFFER_DATA_ADDR_DMP` を指定して呼び出すことで取得することができます。

これらの関数は、*pname* に渡す値によってさまざまな情報を取得することができます。取得することのできる情報については、関数リファレンスを参照してください。

15.7. 各種データのロードサイズ

GPU が、頂点バッファやテクスチャ、コマンドバッファからデータをロードするときの、ロード単位のバイトサイズについて説明します。

15.7.1. 頂点バッファ

頂点インデックスの並びによって、頂点バッファからのロード単位のバイトサイズが変化します。

glDrawElements() の場合、インデックスアレイから 16 個単位でロードされた頂点インデックスをソートし、ソートされた頂点インデックスの順に頂点アレイからデータをロードします。このとき、頂点インデックスが連続する場合は頂点アレイからデータが連続して読み出されます。

glEnableVertexAttribArray() で有効となっている頂点属性が格納されている頂点アレイからデータをロードするとき、glVertexAttribPointer() で指定された情報により、ドライバがインターリーブドアレイ(複数の頂点属性を 1 つにまとめた頂点アレイ)からのロードであると判断した場合はインターリーブドアレイ単位で複数の頂点属性をまとめてロードします。

頂点アレイから連続してデータをロードする場合は、最大 256 バイトのバースト読み出しが行われます。256 バイトを超えた場合はそこで一度区切ってから、続きのデータをロードします。不連続なデータをロードする場合でも、最低でも 16 バイト単位でデータが読み出されます。

glDrawArrays() の場合は、0 からの連番で作成されたインデックスアレイを使用する場合と同等の処理を行います。

15.7.2. テクスチャ

テクスチャのフォーマットによって、ロード単位のバイトサイズが変化します。下記の情報は、テクスチャキャッシュにヒットせずに VRAM から転送される際のバイトサイズです。

表 15-3. テクスチャフォーマットとロード単位の対応

format	type	バイトサイズ
GL_RGBA GL_RGBA_NATIVE_DMP	GL_UNSIGNED_BYTE	128
	GL_UNSIGNED_SHORT_5_5_5_1	64
	GL_UNSIGNED_SHORT_4_4_4_4	64
GL_RGB GL_RGB_NATIVE_DMP	GL_UNSIGNED_BYTE	96
	GL_UNSIGNED_SHORT_5_6_5	64
GL_LUMINANCE_ALPHA GL_LUMINANCE_ALPHA_NATIVE_DMP	GL_UNSIGNED_BYTE	64
	GL_UNSIGNED_BYTE_4_4_DMP	32
GL_LUMINANCE GL_LUMINANCE_NATIVE_DMP	GL_UNSIGNED_BYTE	32
	GL_UNSIGNED_4BITS_DMP	16
GL_ALPHA GL_ALPHA_NATIVE_DMP	GL_UNSIGNED_BYTE	32
	GL_UNSIGNED_4BITS_DMP	16
GL_HILO8_DMP GL_HILO8_DMP_NATIVE_DMP	GL_UNSIGNED_BYTE	64
GL_ETC1_RGB8_NATIVE_DMP	–	128
GL_ETC1_ALPHA_RGB8_A4_NATIVE_DMP	–	32

15.7.3. コマンドバッファ

コマンドバッファは 128 バイト単位でロードされます。

15.8. 特定のピクセルにブロック状のノイズが描画される

3DS のフレームバッファでは、ピクセルデータはブロックアドレスと呼ばれる、4×4 ピクセルを 1 ブロックとするブロック単位で処理されています。また、フレームバッファのキャッシュもブロックアドレスで管理されています。`glFinish()` や `glFlush()`、`glClear()` を呼び出したときや、フレームバッファに関する GPU ステート(`NN_GX_STATE_FRAMEBUFFER`、`NN_GX_STATE_FBACCESS`)をバリデーションしたとき、`nngxSplitDrawCmdlist()` でコマンドリストが区切られたときなどにキャッシュのタグ情報がクリアされます。キャッシュのタグ情報がクリアされると、キャッシュタグは初期値 0x3FFF で初期化されます。そのため、キャッシュタグをクリアした直後の描画で特定のブロックアドレス(0x3FFF)にあたるピクセルを描画しようとしても、キャッシュタグの初期値とピクセルのブロックアドレスが等しいためにキャッシュにヒットしたと誤判定されてしまい、その結果、正しくない色がピクセルに適用されてしまいます。

ブロックアドレスは、フレームバッファ(カラーバッファ、デプスバッファ、およびステンシルバッファ)の先頭アドレスから 16 ピクセル単位で、0 から連番で割り当てられます。その際、GPU の描画フォーマットで配置されたデータに対してアドレスが割り当てられるため、描画イメージのピクセルの位置とブロックアドレスの対応は、ブロックモードがブロック 8 モードの場合とブロック 32 モードの場合で異なります。

この問題はブロックアドレス 0x3FFF に割り当てられたピクセルに起こるため、フレームバッファの総ブロック数が 0x3FFF に満たない場合、つまり、フレームバッファの総ピクセル数が 0x3FFF×16 ピクセル(262128 ピクセル。正方形ならば 512×512 ピクセル)以下の場合は発生しません。また、カラーバッファ、デプスバッファ、ステンシルバッファのいずれに対してもリードアクセスが発生しない場合にも、この問題は発生しません。

> **補足:** キャッシュのタグ情報のクリアは、GPU のレジスタ 0x0110 に 1 を書き込むことでも行われます。
>
> GPU のレジスタに直接アクセスする方法や GPU ステート、ブロックモード、フレームバッファへのリードアクセスの制御については、「3DS プログラミングマニュアル – グラフィックス応用編」を参照してください。

15.8.1. ピクセルとブロックアドレスの対応

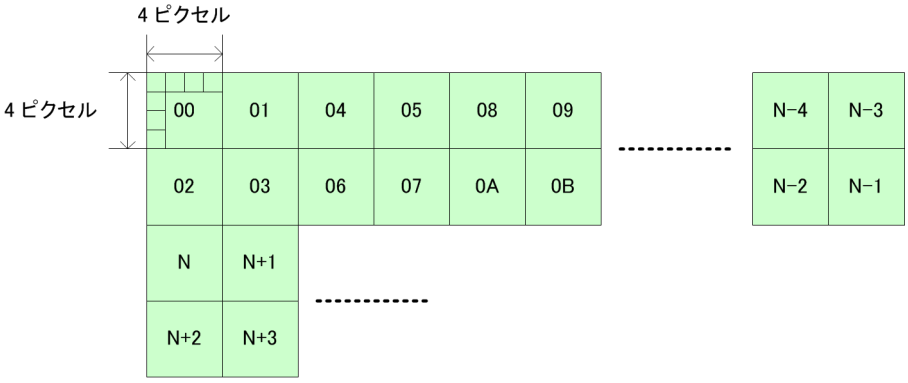
先に述べたように、ブロックアドレスは GPU の描画フォーマットで配置された、カラーバッファおよびデプスステンシルバッファの先頭アドレスから 16 ピクセルごとに 0 から昇順に割り当てられます。`glViewport()` の原点とは異なり、描画イメージの左上端のピクセルがバッファアドレスの先頭にあたります。また、描画イメージの水平方向(幅)が LCD の幅が短い辺に対応していることに注意してください。

アドレスの割り当て方が異なるため、ブロックモードによって描画イメージ上のピクセルに対応するキャッシュのブロックアドレスは異なります。

15.8.1.1. ブロック 8 モード

描画イメージの左上端にある 4×4 ピクセルがブロックアドレス 0、そのすぐ右の 4×4 ピクセルのブロックアドレスが 1、0 の下の 4×4 ピクセルが 2、1 の下の 4×4 ピクセルが 3 です。8×8 ピクセル単位で右方向にブロックアドレスが大きくなり、右端まで行くと、すぐ下の行の左端にある 8×8 ピクセルから続けてカウントします。

図 15-1. ブロック 8 モードでのブロックアドレスの割り当て



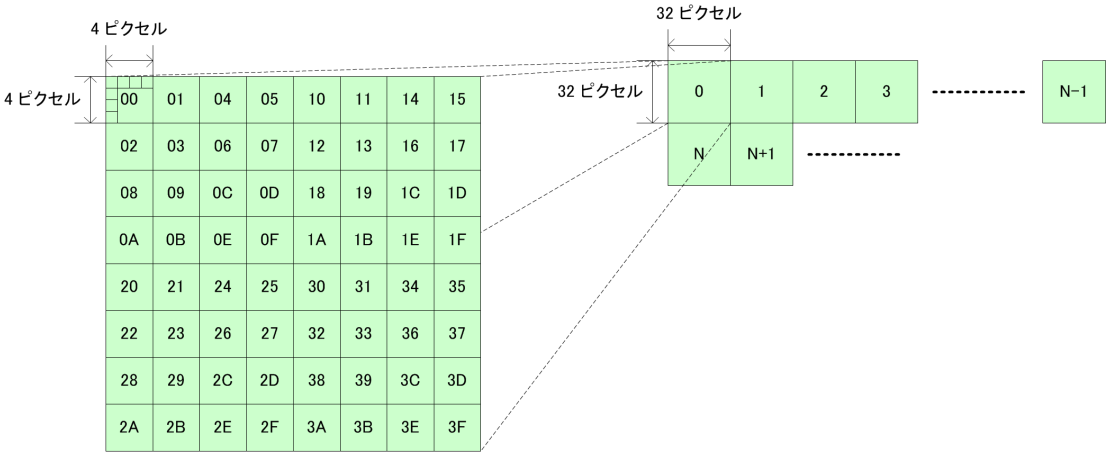
図内の N は、フレームバッファの横のピクセル数を W とすると、N=W÷4×2 で計算することができます。

15.8.1.2. ブロック 32 モード

ブロック 32 モードでは、32×32 ピクセルのメタブロックという単位でアドレスを割り当てます。描画イメージの左上端にある 32×32 ピクセルがメタブロックアドレス 0、そのすぐ右の 32×32 ピクセルがメタブロックアドレス 1 です。32×32 ピクセル単位で右方向にメタブロックアドレスが大きくなり、右端まで行くと、すぐ下の行の左端にある 32×32 ピクセルから続けてカウントします。

下図のように、ピクセルのブロックアドレスは、メタブロック内の左上端の 4×4 ピクセルからジグザグに配置されます。描画イメージ全体から見た各ピクセルのブロックアドレスは、メタブロックアドレス×0x40＋メタブロック内のブロックアドレスで計算することができます。

図 15-2. ブロック 32 モードでのブロックアドレスの割り当て



図内の左側はメタブロック内のピクセルのブロックアドレス（16 進数表記）です。右側は描画イメージ全体から見たメタブロックのアドレスです。

15.8.2. 回避方法 1

フレームバッファの縦方向のピクセル数×横方向のピクセル数が、262128 ピクセル以下であれば問題は発生しません。つまり、不必要に大きいサイズのフレームバッファを使用しないことで問題を回避することができます。

LCD と同じサイズとなる、240×400（総ピクセル数 96000）または 240×320（総ピクセル数 76800）のフレームバッファならば、総ピクセル数が 262128 以下になりますので、問題は発生しません。

フレームバッファのサイズは `glRenderbufferStorage()` で指定しますが、大きめに確保したフレームバッファの一部

(240×400)のみ使用する場合は、フレームバッファの確保領域を必要最低限のサイズに留めることで問題を回避することができます。

15.8.3. 回避方法 2

ブロックアドレス 0x3FFF にあたる問題のピクセルが描画領域の外に配置されるように、フレームバッファのサイズを調整することで問題を回避することができます。

例えば、ブロック 8 モードで 2x2 アンチエイリアスをかけるときのように 480×800 のフレームバッファを確保すると、ピクセル座標(124, 548)を左上端とする 4×4 ピクセルがブロックアドレス 0x3FFF に割り当てられます。この場合、フレームバッファのサイズを横方向に 32 ピクセル拡張して 512×800 とすると、ブロックアドレス 0x3FFF に割り当てられるピクセルは(508, 508)を左上端とする 4×4 ピクセルとなります。そこで、ビューポートの設定でフレームバッファの左側 480×800 のみを表示すれば、問題のピクセルを避けることができます。

この方法では、フレームバッファを必要なサイズよりも余分に確保しなければならないため、VRAM を無駄に消費してしまうデメリットがありますが、サイズを調整するだけで簡単に回避することができます。

ブロックモードやフレームバッファのサイズの違いにより、ブロックアドレス 0x3FFF に割り当てられるピクセルがどの座標になるのかは、前述の「15.8.1. ピクセルとブロックアドレスの対応」を参照してください。

15.8.4. 回避方法 3

キャッシュタグのクリア直後に、ブロックアドレス 0x3FFF に割り当てられているピクセルを避け、数ピクセル描画してキャッシュタグの内容を変化させることで問題を回避することができます。

キャッシュタグの内容を変化させるには、特定のブロックアドレスのピクセルを 4 ピクセル描画しなければなりません。カラーバッファとデプスステンシルバッファの両方をリードする設定のときは、ブロックアドレスの下位 3 ビットがすべて 1(つまり 0x7)であり、かつブロックアドレスが異なるピクセルが 4 ピクセル必要です。カラーバッファとデプスステンシルバッファのどちらか一方のみをリードする設定のときは、ブロックアドレスの下位 4 ビットがすべて 1(つまり 0xF)であり、かつブロックアドレスが異なるピクセルが 4 ピクセル必要です。

例えば、キャッシュタグのクリア直後に描画されるピクセルのブロックアドレスが、

0x00、0x01、0x0F、0x02、0x1F、0x03、0x0F、0x2F、0x3F、・・・

というような場合、0x00 や 0x01 などは下位 4 ビットが 0xF ではないためカウントされません。0x0F は 2 回描画されましたが、同じブロックアドレスのピクセルは 1 つとカウントします。0x0F、0x1F、0x2F、0x3F が描画された時点で問題を回避することができ、0x3F より先に 0x3FFF のピクセルが描画された場合には問題が発生します。

キャッシュタグのクリア直後に、このような条件を満たすピクセルを含むダミーポリゴンを描画することで、問題を回避することができます。ただし、以下の点に注意して描画しなければなりません。

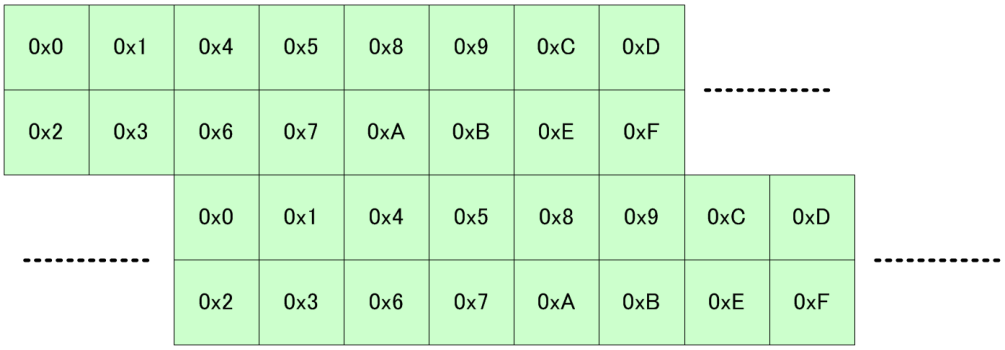
- デプステスト、ステンシルテスト、アルファテストでフェイルするピクセルでもダミー描画として有効です。ただし、ダミー描画が必ずフェイルするような設定をした場合(例えばデプステスト関数に GL_NEVER を設定するような場合)は、ダミー描画のあとに続けて、通常の設定で描画するときにデプステスト関数を元に戻す必要があります。その際に、キャッシュをフラッシュするコマンド(レジスタ 0x0111 に書き込むコマンド)が必要となることに注意してください。
- アルファブレンドの設定により、描画結果がカラーバッファに影響しないようなピクセルでもダミー描画として有効です。
- ビューボリュームおよびユーザー定義のクリップ平面によりクリップされたピクセルはダミー描画として無効です。
- シザーテストにより落とされたピクセルはダミー描画として無効です。
- アーリーデプステストにより落とされたピクセルはダミー描画として無効です。

15.8.4.1. ブロック 8 モード

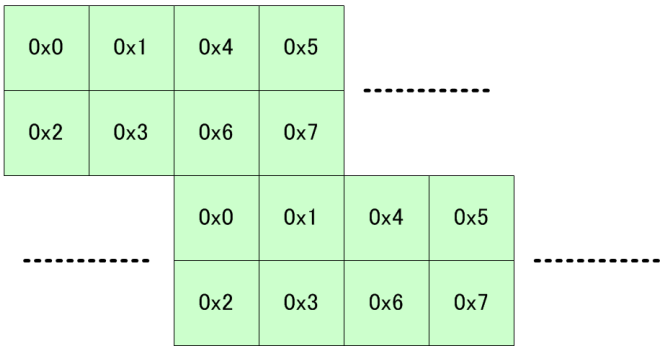
問題を回避するためにダミーポリゴンとして描画するピクセルは、ブロックアドレスの下位 4 ビット、または下位 3 ビットがすべて 1 であるピクセルです。ブロック 8 モードのブロックアドレスの並びを見ると、下位 4 ビットについては 32×8 ピクセル単位、下位 3 ビットについては 16×8 ピクセル単位で同じパターンが横に並ぶことになります。ただし、フレームバッファの横方向のピクセル数によっては、縦方向の 8 ピクセルごとに、横方向に 8 ピクセル単位でずれる可能性があります。

図 15-3. ブロック 8 モードでのブロックアドレスの並び

下位 4 ビット単位



下位 3 ビット単位



以下のサイズの矩形であればダミーポリゴンとしての条件を満たします。最小面積の矩形については、四隅のピクセルが条件となるブロックアドレスにかかっていなければならないことに注意してください。

表 15-4. ブロック 8 モードでダミーポリゴンとして描画する矩形

矩形の形状・条件	下位 4 ビットがすべて 1	下位 3 ビットがすべて 1
最小面積の矩形(配置場所は限定されます)	94×1	46×1
配置場所を選ばない矩形	125×5 29×29	61×5 13×29

15.8.4.2. ブロック 32 モード

問題を回避するためにダミーポリゴンとして描画するピクセルは、ブロックアドレスの下位 4 ビット、または下位 3 ビットがすべて 1 であるピクセルです。ブロック 32 モードのブロックアドレスの並びを見ると、下位 4 ビットについては 32×32 ピクセル単位、下位 3 ビットについては 32×16 ピクセル単位で同じパターンが縦横に並ぶことになります。

図 15-4. ブロック 32 モードでのブロックアドレスの並び

下位 4 ビット単位

0x0	0x1	0x4	0x5
0x2	0x3	0x6	0x7
0x8	0x9	0xC	0xD
0xA	0xB	0xE	0xF

下位 3 ビット単位

0x0	0x1	0x4	0x5
0x2	0x3	0x6	0x7

以下のサイズの矩形であればダミーポリゴンとしての条件を満たします。最小面積の矩形については、四隅のピクセルが条件となるブロックアドレスにかかっていなければならないことに注意してください。

表 15-5. ブロック 32 モードでダミーポリゴンとして描画する矩形

矩形の形状・条件	下位 4 ビットがすべて 1	下位 3 ビットがすべて 1
最小面積の矩形(配置場所は限定されます)	46×1 14×14	46×1 14×6
配置場所を選ばない矩形	61×13 29×29	61×5 29×13

15.9. 意図しない線が描画され、フレームバッファ直後の領域が破壊される

ウィンドウ座標で x=0 の付近に右側のエッジがある非常に小さなポリゴンを描画すると、意図しない線が描画されてしまう場合があります。この現象は、ポリゴンのピクセル生成時の計算誤差によってピクセルの x 座標が負の値になってしまうことでラップアラウンドを起こし、x 座標の値が非常に大きくなるために x の正の方向に長く伸びたポリゴンが描画されてしまうことが原因となっています。

この現象が発生すると、描画領域以外のメモリの内容が壊されてしまう可能性があります。ラップアラウンドした x 座標は 1023 になり、(0, y) から (1023, y) までのピクセルが描画領域のサイズに関わらず生成されます。つまり、描画領域のサイズが 256×256 のように幅が 1024 より小さく設定されている場合でも、x 座標が 0 から 1023 までのピクセルが生成されてしまいます。フレームバッファへのアクセスはピクセルの x、y 座標と描画領域の幅から計算されたアドレスに対して行われますが、ピクセルの x 座標が描画領域の幅より大きい場合でも x 座標はそのままアドレス計算に使用されます。そのため、y 座標によっては、描画領域よりも後方のメモリにピクセルカラーのデータを書き込んでしまう可能性があります。

> **補足:** シザーテストによって描画領域を切り取っている場合はメモリ内容の破壊は起きません。メモリ内容の破壊を回避するためにもシザーテストの設定を行うことを推奨します。なお、シザーテストを行うことで GPU のパフォーマンスへのペナルティはありません。

この現象の発生条件はポリゴンのウィンドウ座標にのみ依存します。同じウィンドウ座標で現象が発生したり発生しなかったりすることはありません。また、ビューボリュームやクリッピングされた結果生成されるポリゴンについて発生するため、元のポリゴン自体が大きい場合でも、ウィンドウ座標 x=0 側の画面端からポリゴンがはみ出して、ビューボリューム内に含まれる面積が非常に小さい場合でも発生します。

この現象の回避方法として、頂点シェーダで x 座標を調整する方法があります。頂点シェーダで計算するクリップ座標 x は -w から w でクリップされますので、x の値が -w 付近の値であれば、その頂点はウィンドウ座標で x=0 側の画面端付近にあるということになります。このような x=0 側の画面端付近に配置される頂点を画面端上に移動させる(x の値を -w に修正する)という処理で現象の発生を回避することができます。この回避方法は、1 ピクセル以下の頂点座標について処理するため、描画結果にはほとんど影響を与えません。

頂点シェーダで行う処理は、射影変換を行ったあとの x の値(頂点座標 x として出力レジスタに書き込む値)に対し下記のような処理をします。

コード 15-3. 現象の発生を回避するための処理

```
if ( -w < x && x < -w * (1-epsilon) ) x = -w;
```

上記の x と w は射影変換後の頂点座標での x と w の値です。epsilon は調整用の変数として、描画するシーンに応じて最適な値を設定します。

以下に頂点シェーダの実装例を示します。mul 命令以降が現象の発生を回避するための処理です。

コード 15-4. 回避方法の実装例

```
// v0    : position attribute
// o0    : output for position
// c0-c3 : modelview matrix
// c4-c7 : projection matrix
// c8    : (1 - epsilon, 1, any, any)
m4x4    r0, v0, c0              // modelview transformation
m4x4    r1, r0, c4              // projection transformation
mul     r2.xy, -r1.w, c8.xy     // r2.x = -w * (1-epsilon), r2.y = -w
cmp     2, 4, r1.xx, r2.xy      //
ifc     1, 1, 1                 // if ((x < -w * (1-epsilon)) && (x > -w))
    mov     r1.x, -r1.w         // x = -w;
endif
mov     o0, r1
```

15.10. ドライバの内部動作の優先度変更

3DS の CPU には複数のコアがあり、1 つはアプリケーションが占有していますが、ほかのコアはシステムが各種デバイスの処理を制御するために占有しています。

システム側のコアで制御されるデバイスは複数あり、その中でも GPU(グラフィックス)の処理は高い優先度で行われるため、負荷の高いグラフィックス処理を行うと、ほかのデバイスの処理に影響を与えることがあります。このような場合、nngxSetInternalDriverPrioMode() で GPU の優先度を低くすることにより、ほかのデバイスへの影響を解決できる可能性があります。

コード 15-5. GPU ドライバの内部動作の優先度変更

```
void nngxSetInternalDriverPrioMode(nngxInternalDriverPrioMode mode);
```

mode には、以下の値から選択して指定します。

表 15-6. 内部動作の優先度

定義	説明
NN_GX_INTERNAL_DRIVER_PRIO_MODE_HIGH	高い優先度で動作(デフォルト)
NN_GX_INTERNAL_DRIVER_PRIO_MODE_LOW	低い優先度で動作

優先度を低くすることで、ほかのデバイスの処理への影響は抑えられますが、グラフィックス性能が低下します。

15.11. ライブラリ内部で管理領域を確保する関数

gl 関数や nngx 関数の中には、ライブラリが暗黙的に管理領域を確保するものがあります。

15.11.1. nngxValidateState()、glDraw*()

参照テーブル(LUT)が利用される場合に、ライブラリは管理領域を確保します。

glTexture1D() が呼ばれ、参照テーブルが利用されることがライブラリに通知されると、次に nngxValidateState() や glDraw*() が実行されるときに参照テーブルロード用の中間バッファが確保されます。ただし、3D コマンドを直接生成し、参照テーブルを使うバッファを指定した場合は確保されません。

なお、確保した領域は glDeleteTexture() または nngxFinalize() で解放されます。

15.11.2. コマンドリスト、ディスプレイリスト、テクスチャなど

コマンドリスト、ディスプレイリスト、テクスチャなどを確保するための関数はオブジェクトごとに管理領域を確保し、確保された管理領域は nngxDelete*() や glDelete*() で対応するオブジェクトが破棄されるまで保持されます。

上記に該当する関数は以下のとおりです。

> nngxGenCmdlists()、nngxGenDisplaybuffers()、glCreateProgram()、glCreateShader()、glGenBuffers()、glGenRenderbuffers()、glGenTextures() など

15.12. GPU ハングアップの原因の解析

nngxGetCmdlistParameteri() の *pname* に NN_GX_CMDLIST_HW_STATE を渡して取得したデータのいくつかのビットはハードウェアがビジー状態であることを示します。つまり、GPU がハングアップした場合など、ハードウェアの動作に問題が発生した場合のこのデータが原因の解析に役立つ可能性があります。

ハードウェアが不正動作をしたときは、基本的にビジー状態のままになっているモジュールのいずれかが原因である可能性が高いと考えられます。ただし、トライアングルセットアップ⇒ラスタライゼーションモジュール⇒テクスチャユニットなどのように連続するモジュールでは、後段のモジュールから前段のモジュールへビジー信号が伝搬するため、連続するモジュールがビジー状態になっている場合、その最後段のモジュールが原因である可能性が考えられます。これとは逆に、ビット 6 のパーフラグメントオペレーションモジュールは前段からのデータが不正だった場合にもビジー状態となるため、問題の原因は前段にある可能性もあります。

ビジー信号の伝搬は大きく分けて、ビット 0 ～ 7 のラスタライズ/ピクセル処理部と、ビット 8 ～ 16 のジオメトリ処理部に分けられます。

ラスタライズ/ピクセル処理部は、トライアングルセットアップ⇒ラスタライゼーションモジュール⇒テクスチャユニット⇒フラグメントライティング⇒テクスチャコンバイナ⇒パーフラグメントオペレーションモジュールの順に連続するモジュールであり、後段のモジュールのビジー信号が前段のモジュールへ伝搬します。つまり、ビット 5 ⇒ビット 4 ⇒ビット 3 ⇒ビット 2 ⇒ビット 1 ⇒ビット 0 へと伝搬します。

ビット 6 もパーフラグメントオペレーションモジュールのビジー信号です。ただし、ビット 0 とビット 1 には伝搬しますが、ビット 2、ビット 3、ビット 4 には伝搬しません。

ビット 7 のアーリーデプステストモジュールはアーリーデプスバッファ(GPU の内蔵メモリ)のクリア待ちのときにビジー状態となり、ほかのモジュールにビジー信号が伝搬しません。

トライアングルセットアップからその前段モジュール(頂点キャッシュまたはジオメトリ生成)へはビジー信号が伝搬しません。つまり、ラスタライズ/ピクセル処理部とジオメトリ処理部の間ではビジー信号が伝搬しません。

続いてジオメトリ処理部について説明します。頂点入力プロセス(コマンドバッファおよび頂点アレイをロードするモジュール)⇒頂点処理プロセッサ⇒ポスト頂点キャッシュの順に連続するモジュールであり、後段のモジュールのビジー信号が前段のモジュールへ伝搬します。つまり、ビット 16 ⇒(ビット 11、ビット 12、ビット 13、ビット 14)⇒ビット 8 ⇒ビット 9 へと伝搬します。ビット 11、ビット 12、ビット 13、ビット 14 は頂点プロセッサ 0、1、2、3 のビジー状態に対応しますが、各頂点プロセッサは頂点ロードモジュールとポスト頂点キャッシュの間に並列に配置されているため、ポスト頂点キャッシュのビジー信号は 4 つの頂点プロセッサのうちの 1 つまたは複数に伝搬します。4 つあるすべての頂点プロセッサに必ず伝搬するわけではありません。

上記はジオメトリシェーダが無効の場合です。ジオメトリシェーダが有効である場合、ジオメトリシェーダプロセッサである頂点プロセッサ 0 はポスト頂点キャッシュの後段に位置します。このとき、ジオメトリシェーダプロセッサのビジー信号はポスト頂点キャッシュへと伝搬します。ジオメトリシェーダプロセッサを起因とするビジー信号はポスト頂点キャッシュへは伝搬しますが、それより前段へは伝搬しません。ポスト頂点キャッシュを起因とするビジー信号は、前段のモジュール(頂点プロセッサ 1、2、3)へと伝搬します。つまり、ビット 11 ⇒ビット 16、およびビット 16 ⇒(ビット 12、ビット 13、ビット 14)⇒ビット 8 ⇒ビット 9 へとビジー信号が伝搬します。

ビット 16 に対応するポスト頂点キャッシュは、キャッシュ内に頂点データが満杯まで入力されたときにビジー信号を出力します。何らかの理由により、ポスト頂点キャッシュが後段のモジュールに対してデータが出力できない場合、例えば後段のモジュールがハングアップしたような場合ではポスト頂点キャッシュに頂点データが溜まり、ポスト頂点キャッシュがビジー信号を出力します。ポスト頂点キャッシュの後段のモジュールは、ジオメトリシェーダが無効な場合はトライアングルセットアップであり、ジオメトリシェーダが有効な場合はジオメトリシェーダプロセッサ(頂点プロセッサ 0)です。

ビット 2 のテクスチャユニットがビジー状態で GPU がハングアップする原因には、4 ビットフォーマットと非 4 ビットフォーマットのテクスチャを同時にマルチテクスチャとして使用するときに発生するハードウェアの不具合が挙げられます。

ロードアレイ(頂点属性データの GPU へのロード単位)の設定が正しくないことが原因で GPU がハングアップした場合、ビット 8 がビジー状態になります。

頂点シェーダから NaN が出力された(ジオメトリシェーダが使用されている場合はジオメトリシェーダから NaN が出力された)ことが原因で GPU がハングアップした場合、ラスタライゼーションモジュールおよびトライアングルセットアップ(ビット 0 およびビット 1)がビジー状態となります。

> **補足:** `nngxGetCmdlistParamerteri()` および `NN_GX_CMDLIST_HW_STATE` で取得するデータのビットについては、「4.1.10. パラメータ取得」を参照してください。

15.12.1. GPU ハングアップ時のハードウェア状態の事例

参考までに、GPU がハングアップしたときに NN_GX_CMDLIST_HW_STATE で取得したハードウェア状態とハングアップの原因の事例を以下に挙げます。

表 15-7. ハードウェア状態と GPU のハングアップの原因

ハードウェア状態	GPU のハングアップの原因
0x00011303 0x00011F03 0x00012F03	GPU の動作中に CPU で頂点バッファの内容を破壊していた。
0x00010103 0x0001011B 0x00014303	GPU の不具合により、VRAM 上の頂点バッファの内容が描画中に破壊されていた。 (「15.9. 意図しない線が描画され、フレームバッファ直後の領域が破壊される」を参照してください)
0x00010107 0x00011307 0x00012307	マルチテクスチャのハードウェア不具合によりハングアップした。 テクスチャアドレスの 128 バイトのアライメントに抵触していた。 (「7.3.1. コンポーネントが 4 ビットのフォーマットについて」を参照してください)
0x00000100	PICA レジスタの 0x0229 のビット 8 および 0x0253 のビット 8 が正しく設定されていなかった。 (GL 関数以外で描画したあとに GL 関数で描画するときは、上記のレジスタを再設定する必要があります) 未使用のロードアレイの要素設定数(レジスタ 0x0205 + n * 3 のビット 31 ～ 28)に 0 が設定されていない。
0x00007300	PICA レジスタの 0x0289 がジオメトリシェーダを使用する設定になっている状態で、ジオメトリシェーダを使用しない頂点シェーダを実行していた。
0x00000000	コマンドバッファのアドレスジャンプ機能関連レジスタが正しく設定されていなかった。 コマンドリクエストの実行待機状態で、アドレスジャンプ機能関連レジスタの設定完了前に nngxFlush3DCommandPartially() を呼び出した。
0x00000001 0x00000002 0x00000003	頂点シェーダから NaN が出力された。

> **補足:** PICA レジスタの設定方法については「3DS プログラミングマニュアル – グラフィックス応用編」を参照してください。

15.13. 頂点属性の組み合わせによる頂点データの転送速度への影響

頂点バッファを使用する場合、頂点属性のデータタイプとデータサイズ(glVertexAttribPointer() の *type* と *size*)の組み合わせが頂点データの転送速度に影響します。

頂点バッファに格納された頂点属性データは、1 つまたは複数の頂点属性がまとめられて GPU にロードされます。このときにロードされる頂点データの単位を「ロードアレイ」と呼びます。

> **補足:** ロードアレイの詳細については、「3DS プログラミングマニュアル – グラフィックス応用編」を参照してください。

GPU は各ロードアレイを転送する際に、ロードアレイを構成する頂点属性のデータタイプとデータサイズの組み合わせによって、先読み転送を行うかどうかを判定します。先読み転送が行われる場合、頂点データの転送速度が速くなります。

下記の条件式が成り立つ場合に、先読み転送が行われます。

("GL_FLOAT 以外の型の属性数" + "データサイズが 1 の属性数")
<= ("GL_FLOAT 型でデータサイズが 4 の属性数" + "GL_FLOAT 型でデータサイズが 3 の属性数" / 2)

"GL_FLOAT 以外の型の属性数"のデータサイズ、"データサイズが 1 の属性数"のデータタイプは任意です。複数の条件に当てはまる頂点属性は、それぞれの項目でカウントされます。例えば、データサイズが 1 の GL_BYTE 型の頂点属性は、"GL_FLOAT 以外の型の属性数"と"データサイズが 1 の属性数"の両方でカウントされます。

先読み転送が行われるかどうかの条件が同じである場合、転送速度はロードアレイあたりのデータ量に依存します。データ量が少ないほど、転送速度が速くなります。さらに、頂点データのデータ量が同じである場合、転送速度はロードアレイに含まれる属性数に依存します。ロードアレイの属性数が少ないほど、転送速度が速くなります。

15.14. 頂点アレイのアドレスアライメント

頂点バッファを使用して描画する場合、頂点アレイのアドレスのアライメントを 32 バイトにすることで、描画時に頂点アレイの転送処理の効率が向上する可能性があります。ここでいう頂点アレイのアドレスとは、頂点バッファのアドレスに glVertexAttribPointer() で指定するオフセット(*ptr* で指定する値)を加算した値のことです。

頂点アレイのアドレスアライメントを 32 バイトに揃えない場合に比べて、どのくらい速度が向上するかは頂点属性の型、サイズ、頂点アレイの格納場所、頂点インデックスの内容に依存するため、必ず効果が得られるとは限りません。また、転送処理の性能のみが向上しても、頂点アレイの転送処理がボトルネックとなっている状態でなければ、システム全体としての性能向上にはつながりません。

15.15. マルチテクスチャ使用時に GPU がハングアップする

> **補足:** SNAKE では、マルチテクスチャの使用で GPU がハングアップすることはありません。ただし、アプリケーションが CTR 上で動作している場合は起こり得ることに注意してください。

以下のすべての条件を満たす場合、GPU がハングアップすることがあります。

- テクスチャを複数使用している。
- テクスチャユニット間のパフォーマンスに大きな差がある。

発生条件には、プロシージャルテクスチャは含まれません。通常のテクスチャ 1 枚とプロシージャルテクスチャを同時に使用しても、本現象は発生しません。

以下のいずれかの方法で、本現象を回避できます。

- テクスチャを 1 枚のみ使用する。
- 使用するすべてのテクスチャに対して、テクスチャパラメータ GL_TEXTURE_MIN_FILTER の設定を GL_XXX_MIPMAP_LINEAR(トライリニアフィルタ使用)にする。
  実際にミップマップが存在しないテクスチャに対しても、本パラメータを設定する必要があります。

また、本現象の緩和策として以下の方法が挙げられます。

- 使用するテクスチャの一部に対して、テクスチャパラメータ GL_TEXTURE_MIN_FILTER の設定を GL_XXX_MIPMAP_LINEAR(トライリニアフィルタ使用)にする。
  すべてのテクスチャに対してトライリニアフィルタを使用することで完全に回避することができますが、一部のテクスチャに対して使用することで、現象の発生を緩和させることができます。
- 同時に使用するすべてのテクスチャを同一の VRAM に配置する。
- 使用するテクスチャの枚数を減らす。

本現象の発生はタイミングに依存するため、下記のようにテクスチャの設定を変更することで回避できる可能性があります。

ただし、場合によっては発生頻度が悪化する可能性もあります。

- テクスチャのサイズを変更する。
- テクスチャのフォーマットを変更する。
- テクスチャのフィルタモードを変更する。
- テクスチャの格納場所を VRAM-A、VRAM-B、FCRAM(デバイスメモリ)間で変更する。

ハングアップの原因が本現象であるかどうかの確認方法は、「15.12. GPU ハングアップの原因の解析」で紹介しています。

ただし、4 ビットテクスチャフォーマットの格納場所に関する制限に抵触した場合も同じ状態でハングアップすることがあるため、確実な判別はできません。

15.16. テクスチャコンバイナの GL_INTERPOLATE の計算

テクスチャコンバイナの GL_INTERPOLATE の計算式は src0 * src2 + src1 * (1 - src2) ですので、src2 が 1 または 0 の場合は、それぞれ src0 または src1 の値そのものとなることが期待されます。しかし、実装上の仕様により、src2 が 0 かつ、src0 < src1 を満たす場合は、期待される src1 そのままの値ではなく、src1 より輝度が 1 小さい値が出力されてしまいます。

この問題を回避するには、GL_MODULATE と GL_MULT_ADD_DMP を組み合わせて、2 ステージのコンバイナで計算する必要があります。もしくは、src2 のオペランドを GL_ONE_MINUS_* に変更して src0 と src1 を入れ替えることで、問題が発生するフラグメントを減らせる可能性があります。

15.17. 同じ頂点座標のポリゴンで描画結果が完全に一致しない

頂点座標が全く同じポリゴンを描画しても、フラグメントの各属性値が完全には一致しないことがあります。

この現象は、ラスタライゼーションモジュールに入力される頂点の順番が異なる場合に、フラグメントの補間計算の誤差により発生します。頂点の入力順序が全く同じ場合は発生しませんが、GL_TRIANGLES を引数に glDrawElements() で描画すると、直前のポリゴンの頂点インデックスとの関係によっては、内部で頂点の入力順序が変更されることがあります。フラグメントが完全に一致するポリゴンを複数回描画したいときは、各ポリゴンの前後関係も含めて同じ頂点インデックスを使用して描画する必要があります。

更新履歴

Version 1.5 2016-05-10

変更

- 4. コマンドリスト
  - コマンドリクエストを発行する関数はコア0でのみ呼び出し可能なことを追記しました。

Version 1.4 2015-11-05

追加

- 3.8. 表示部分の指定について
- 7.5. フレームバッファからのコピー
- 12.4.5.2. シルエットのシャドウアーティファクト
- 15.17. 同じ頂点座標のポリゴンで描画結果が完全に一致しない

変更

- 1. はじめに
  - アライメント制約の定義名についてはAPIリファレンスを参照するよう補足を追記しました
- 3. LCD
  - 節「レンダリング結果が LCDに表示されるまで」を統合しました。
- 3.3. バッファの確保
  - 図「フレームバッファとディスプレイバッファの構成」を追加しました。
- 4.1.7.1. 蓄積済み 3D コマンドバッファのフラッシュ
  - キャッシュを反映する関数を、nngxUpdateBuffer() から nngxUpdateBufferLight() に修正しました。
  - エラー値 GL_ERROR_80E0_DMP を GL_ERROR_80B0_DMP に修正しました。
- 4.1.19. メモリフィルコマンドの追加
  - 表「レンダーバッファのフォーマットによるフィルパターン」のGL_RGB5_A1 の A のビットを修正しました。
- 7.9. テクスチャコレクション
  - テクスチャコレクションの説明を追記しました。
- 9.3.8. オープンエッジ
  - 図「オープンエッジのインデックスの例」を追加しました。
- 9.4.6. サブディビジョンパッチのインデックス
  - サブディビジョンパッチのインデックスについて例を追記しました。
- 9.5.6. サブディビジョンパッチのインデックス
  - 図「サブディビジョンパッチのインデックスの例」を追加しました。
- 9.6.2. 予約ユニフォーム
  - 使用するシェーダファイルを明記しました。
- 10. ラスタライズ
  - 図「頂点シェーダー、ジオメトリシェーダからラスタライズまでの処理の流れ」を追加しました。
- 12.2.10. 参照テーブルの作成と指定
  - コード「反射(R 成分)の参照テーブルへの指定例」において dmp_LightEnv.samplerRR を dmp_FragmentMaterial.samplerRR に修正しました。
- 12.6.3. シェーディングパス
  - フォグ機能のガスモードを有効化する際に使用されるユニフォーム名を記載しました。
- 13. パーフラグメントオペレーション
  - 図「標準モードでのパーフラグメントオペレーションの処理の流れ」を追加しました。
- 13.2.1. 使用方法
  - ステンシルテストとデプステストがいずれか一方、または両方有効な場合でも性能差がないことを追記しました。

- 13.3.1. 使用方法
  - ステンシルテストとデプステストがいずれか一方、または両方有効な場合でも性能差がないことを追記しました。
- 15.1. glFinish() と glFlush()
  - 「3D コマンドの実行を明示的にライブラリに対して指示することができる」との説明を削除しました。
- 15.2. OpenGL ES との違い
  - 表「関数ごとの相違点」に glGetShaderSource() を追記しました。

削除

- レンダリング結果が LCD に表示されるまで
  - 「3. LCD」に統合されました。

Version 1.3 2015-04-28

変更

- 11.1.6. テクスチャキャッシュについて
  - テクスチャキャッシュについての情報を追加しました。
- 15.15. マルチテクスチャ使用時に GPU がハングアップする
  - GPU ハングアップの回避策としてトライリニアフィルタを使用する際の情報を追加しました。

Version 1.2 2015-01-15

変更

- 4.1.16. ブロックイメージからリニアイメージへの変換転送コマンドの追加
  - 転送リクエストに指定するイメージサイズについて追記しました。
  - nngxAddB2LTransferCommand() が生成するエラーを追記しました。

Version 1.1 2014-11-10

変更

- 3.5. カラーバッファからディスプレイバッファへのコピー
  - nngxTransferRenderImage について、コピー領域の最小値、最小値未満指定時のエラーの記載を追記しました。

Version 1.0 2014-09-04

追加/変更

- 初版