3DS

3DS プログラミングマニュアル

グラフィックス応用編

2015-11-05

Version 1.3

目次

コード

表

図

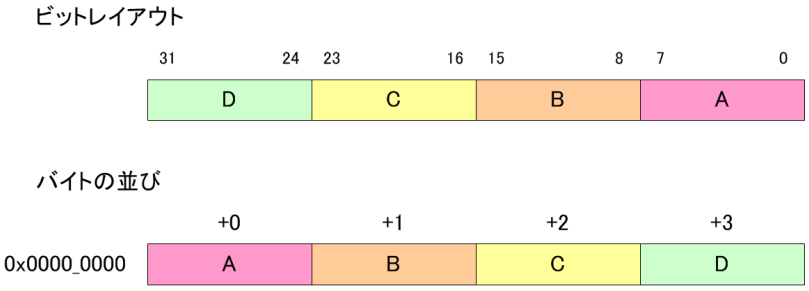
1. はじめに

本ドキュメントは、ブロックモードやステートキャッシュのように特殊な設定が必要な機能や、様々な照明モデルの実装例など、3DS のグラフィックス機能を応用したプログラミング手法について説明したものです。事前に「3DS プログラミングマニュアル － システム編」および「3DS プログラミングマニュアル － グラフィックス基本編」を読んでいることを前提に書かれています。

1.1. ビットレイアウトの表記について

3DS のエンディアンはリトルエンディアンであるため、メモリ上でのバイトの並びとビットレイアウトでの並びがそのまま対応していません。下図は 32 ビットデータのビットレイアウトと、メモリ上でのバイトの並びを対比したものです。

図 1-1. ビットレイアウトとバイトの並び



2. ブロックモードの設定

3DS のレンダーバッファにはブロック 8 モードとブロック 32 モードの 2 通りの設定が存在します。後述するアーリーデプステストを使用する場合はブロック 32 モードでなければなりませんが、使用しない場合は初期状態で設定されているブロック 8 モードのままにしてください。

ブロックモードの設定は `glRenderBlockModeDMP()` で行います。

コード 2-1. glRenderBlockModeDMP() の定義

```
void glRenderBlockModeDMP(GLenum mode);
```

mode は以下の設定値から選択します。

表 2-1. ブロックモードの設定値

設定値	ブロックモード
GL_RENDER_BLOCK8_MODE_DMP	ブロック 8 モード(デフォルト)
GL_RENDER_BLOCK32_MODE_DMP	ブロック 32 モード

ブロック 32 モードはブロック 8 モードと異なり、以下のような制限があります。

- ブロックサイズが 32 に変更されるため、レンダーバッファおよびディスプレイバッファの幅と高さのピクセル数が 32 の倍数でなければなりません。この制限のため、LCD とぴったり同じサイズのレンダーバッファ、ディスプレイバッファを確保することができなくなります。
- `glCopyTexImage2D()`、`glCopyTexSubImage2D()` でのテクスチャへのコピーは正しく行われません。
- フレームバッファにテクスチャを指定(render to texture)した場合の描画結果は正しくありません。この制限のため、シャドウやガスのようにテクスチャにレンダリングする必要のある処理を行うことができなくなります。

ブロックモードを変更しながらレンダリングを行った場合の描画結果は保証されません。同じレンダーバッファへの `glDrawElements()`、`glDrawArrays()`、`glReadPixels()` は同じブロックモードで実行してください。現在のブロックモードを取得するには `glGetIntegerv()` に `GL_RENDER_BLOCKMODE_DMP` を渡してください。

2.1. アーリーデプステスト

アーリーデプステストとはラスタライズと同時に行われるデプステストのことで、通常のデプステストのみで行うよりも早い段階で不要なフラグメントを棄却することができます。この機能を使用するには、レンダーバッファのブロックモードがブロック 32 モードでなければなりません。

アーリーデプステストはアーリーデプスバッファというバッファを使用して処理が行われます。アーリーデプスバッファは、32x32 ピクセルを 1 つのブロックとして扱い、ブロックの代表値としてデプス値を 1 つ保持しています。代表値は 12 bit 精度で記憶され、デプス情報が標準のデプスバッファに書き込まれるときに、対応するブロックのアーリーデプス値が更新されます。アーリーデプスバッファはメモリ上に領域を確保する必要がなく、1024x1024 ピクセルのデプスバッファまで対応することができます。比較はラスタライズ時に計算された描画中のポリゴンのデプス代表値(最小値または最大値)との間で行われ、アーリーデプステストにパスした場合はブロック内すべてのフラグメントが次のプロセスに送られ、フェイルした場合はブロック内すべてのフラグメントが棄却されます。

アーリーデプステストを行うことでフラグメント処理のパイプラインに送られるフラグメントの量を抑えることができます。しかし、標準のデプステストよりも判定の精度が低いために標準のデプステストとは異なる判定をすることがあり、思ったようにパ

フォーマンスが改善されなかったり、本来は描画されるべきフラグメントが棄却されたりする可能性があることに注意しなければなりません。判定で棄却されるべきフラグメントが誤ってパスしたとしても通常のデプステストで棄却されるため、パフォーマンス向上には繋がりませんが特に問題は発生しません。しかし、逆にパスされるべきフラグメントが棄却された場合は描画されるべきフラグメントが描画されず、表示上の問題が発生します。

アーリーデプステストは早期に実行するデプステストという位置付けにあるため、デプステストが無効である場合はアーリーデプステストも無効にしなければなりません。アーリーデプステストのみを実行した場合は、描画されるべきフラグメントが描画されない可能性があります。

2.1.1. 実行の制御

通常のデプステストと同様に、アーリーデプステストの実行と停止は `glEnable()` と `glDisable()` で行います。アーリーデプステストが実行されるかどうかは `glIsEnabled()` で取得することができます。それぞれの関数への引数には `GL_EARLY_DEPTH_TEST_DMP` を渡します。

コード 2-2. アーリーデプステストの実行制御

```
glEnable(GL_EARLY_DEPTH_TEST_DMP);
glDisable(GL_EARLY_DEPTH_TEST_DMP);
glIsEnabled(GL_EARLY_DEPTH_TEST_DMP);
```

2.1.2. バッファのクリア

アーリーデプスバッファは通常のデプスバッファとは独立しています。クリアは個別に行うことができますが、通常のデプスバッファをクリアした場合は必ずアーリーデプスバッファもクリアしてください。

アーリーデプスバッファをクリアするには、*mask* に `GL_EARLY_DEPTH_BUFFER_BIT_DMP` を含むビットマスクを渡して `glClear()` を呼び出してください。バッファのクリアに使用される値は、`glClearEarlyDepthDMP()` で設定します。

コード 2-3. アーリーデプスバッファのクリア値の設定関数

```
void glClearEarlyDepthDMP(GLuint depth);
```

アーリーデプスバッファのクリア値には通常のデプスバッファのクリア値とほぼ同じ値を使用しますが、通常のデプスクリア値が `GLfloat` 型で指定するのに対し、アーリーデプステストでは正の整数で指定しなければなりません。指定する値の目安は、以下の式で計算することができます。

$$Depth_{Early} = Depth \times 0xFFFFFF + offset \quad (0.0 \le Depth \le 1.0)$$

アーリーデプステストが通常のデプステストよりも低い精度で行われることを考慮して、*offset* には `GL_LESS` または `GL_LEQUAL` モードならば 0x1000 以上の正の値、`GL_GREATER` または `GL_GEQUAL` モードならば -0x1000 以下の値を採用することを推奨しています。アーリーデプスバッファのクリア値がとりうる値の範囲は 0x000000 ～ 0xFFFFFF ですので、*offset* を加えた結果がこの範囲を超えた場合は 0x000000 ～ 0xFFFFFF にクランプしてください。

設定されているクリア値を取得するには、*pname* に `GL_EARLY_DEPTH_CLEAR_VALUE_DMP` を渡して `glGetIntegerv()` を呼び出してください。

2.1.3. デプスバッファ更新フラグ

アーリーデプスバッファでは、アーリーデプス値を 32x32 ピクセル単位のブロックで保持するほかに、通常のデプスバッファの更新フラグ（デプスバッファ更新フラグ）を 4x4 ピクセル単位で保持しています。このフラグには、通常のデプステストの結果によって（通常の）デプスバッファの 4x4 の範囲にあるピクセルが 1 ピクセルでも更新されると 1 がセットされ、そのピクセ

ルが含まれているアーリーデプスバッファのブロックで保持されているアーリーデプス値が更新されます。フラグが 1 にセットされると、そのあとにその範囲にあるピクセルでデプスバッファが更新されたとしてもフラグは変化せず、アーリーデプス値も更新されません。ただし、同じアーリーデプスバッファのブロックに含まれている、まだフラグが 0 の範囲のピクセルでデプスバッファが更新されたときは、フラグが 1 にセットされると同時にアーリーデプス値が更新されます。なお、デプスバッファ更新フラグが 0 にクリアされるのは、アーリーデプスバッファをクリアしたときのみです。

デプスバッファ更新フラグによって制御されるため、アーリーデプステストの効果はモデルの描画順序に強く依存しています。アーリーデプステストの効果を大きくするには、比較関数が GL_LESS または GL_LEQUAL の場合は手前のモデルから奥のモデルの順に描画し、比較関数が GL_GREATER または GL_GEQUAL の場合は奥のモデルから手前のモデルの順に描画します。

例えば、比較関数に GL_LESS を設定しているときに奥側にあるモデルを先に描画してしまうと、そのあとに手前側にあるモデルを描画しても、アーリーデプスバッファは更新されないため、アーリーデプステストにパスするフラグメントが増え、アーリーデプステストの効果が小さくなってしまいます。逆に、手前側にあるモデルを先に描画すれば、アーリーデプステストで破棄されるフラグメントが増えることになり、アーリーデプステストの効果が上がります。

また、上記の設定で奥側にある背景を最初に描画する場合でもアーリーデプステストの効果が得られません。このような場合は、背景を描画するときだけアーリーデプステストを無効にし、ほかのモデルを描画するときにアーリーデプステストを有効にすることで、限定的にアーリーデプステストの効果を得ることができます。

> **注意:** 更新フラグによって 1 度描画されたピクセルに対応するアーリーデプス値が更新されませんので、偶数フレームと奇数フレームで比較関数を変えてデプスバッファのクリアを回避していても、更新フラグを 0 にするためにフレームごとにアーリーデプスバッファをクリアしなければなりません。

2.1.4. 比較関数の設定

アーリーデプステストで使用する比較関数の設定は通常のデプステストとは別に設定しなければなりません。比較関数は glEarlyDepthFuncDMP() で設定することができます。

コード 2-4. アーリーデプステストの比較関数の設定関数

```
void glEarlyDepthFuncDMP(GLenum func);
```

func には比較関数を指定します。設定可能な比較関数は GL_LESS、GL_LEQUAL、GL_GREATER、GL_GEQUAL の 4 種類です。そのほかの比較関数は設定することができません。通常のデプステストと異なる設定でのアーリーデプステストは、判定に食い違いが生じる可能性があります。

アーリーデプステストの比較関数には制約があり、容易に変更することができません。1 つのシーンでデプステストの比較関数を変更しながら描画する場合にはアーリーデプステストは使用できなくなってしまいますが、アーリーデプスバッファだけをクリアすることで制約から解放されることがあります。

現在設定されている比較関数を取得するには、*pname* に GL_EARLY_DEPTH_FUNC_DMP を渡して glGetIntegerv() を呼び出してください。初期状態では GL_LESS が設定されています。

アーリーデプステストが有効である場合、標準のデプスバッファに値が書き込まれるときに、対応するアーリーデプスバッファの値をブロック内のフラグメントが持つデプス値の最大値または最小値で更新します。アーリーデプステストの比較関数が GL_LESS モードまたは GL_LEQUAL モードの場合は最大値で、GL_GREATER モードまたは GL_GEQUAL モードの場合は最小値でアーリーデプスバッファは更新されます。アーリーデプスバッファの内容を外部に直接取り出すことや、外部から直接値を設定することはできません。

2.1.4.1. 比較関数の変更について

1 フレームを描画する間に比較関数を変更するかどうかに関わらず、アーリーデプステストで比較関数を変更するにはいくつかの制約があります。

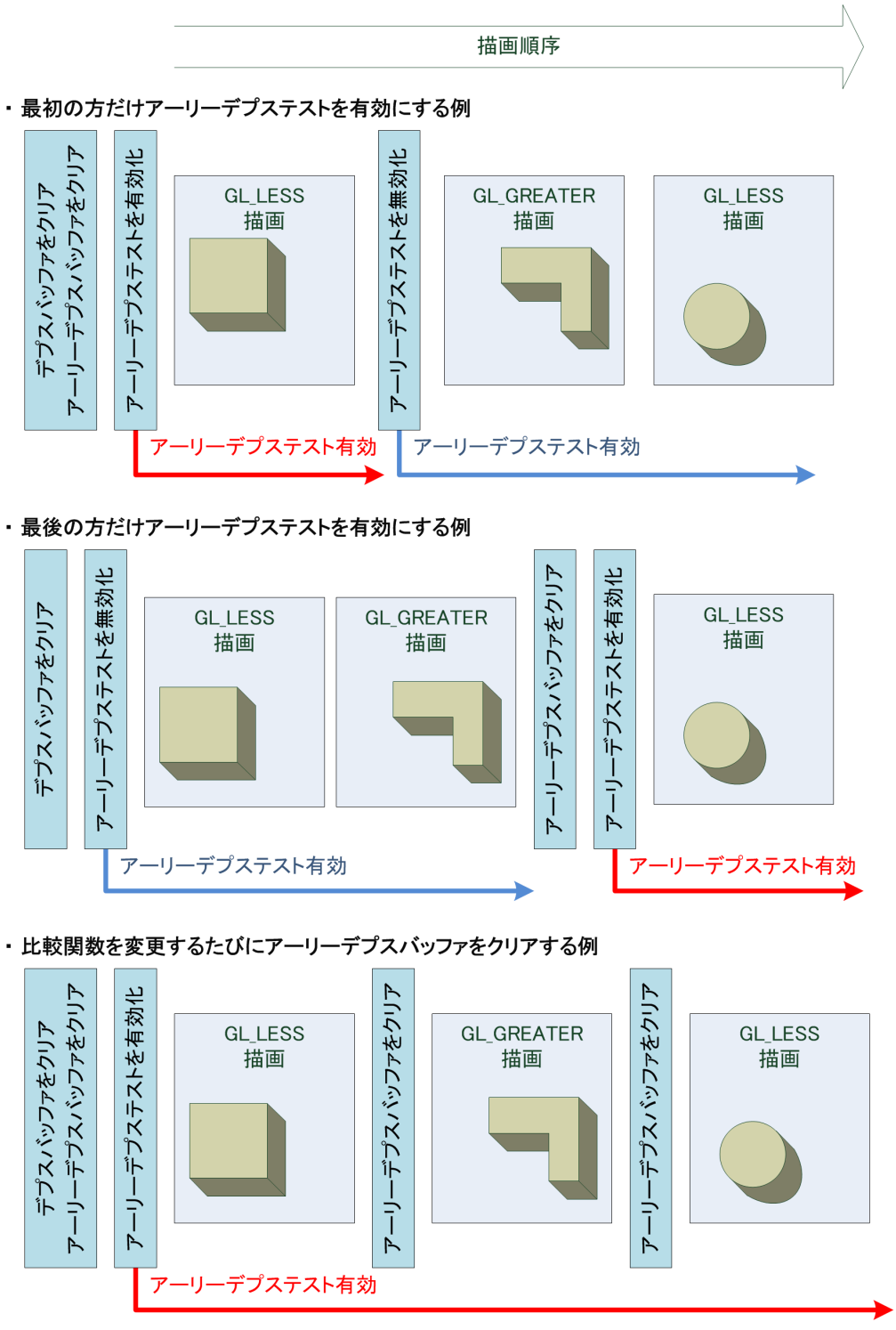
- アーリーデプステストの比較関数は標準のデプステストと同じでなければなりません。
- アーリーデプステストの比較関数を一度設定したら、アーリーデプスバッファをクリアするまでアーリーデプステストの比較関数を変更することができません。
- 1 つのレンダーバッファに対して通常のデプステストの比較関数が変更されて描画が行われたら、アーリーデプスバッファをクリアするまでアーリーデプステストを有効にすることができません。
- 通常のデプステストの比較関数がアーリーデプステストで使用できない比較関数に変更された場合は、アーリーデプステストを無効(停止)にしなければなりません。

これらの制約に従わない場合、間違ったフラグメントの棄却が行われて正しい描画結果が得られない可能性があります。

制約を回避する方法としては、パフォーマンスを向上させたい箇所のみアーリーデプステストを有効にする方法や、比較関数を変更するたびにアーリーデプスバッファをクリアする方法などがあります。ただし、後者の方法でアーリーデプスバッファのクリアが頻繁に行われると、間違ったパス判定が増えてパフォーマンスが向上しない可能性があります。

1 フレーム中で比較関数を変更するなど、通常のデプスバッファのクリアと異なるタイミングでアーリーデプスバッファをクリアするときは、誤判定を防ぐためにも、クリア値に最小値(0x000000)または最大値(0xFFFFFF)を設定するようにしてください。

図 2-1. 比較関数変更の制約を回避する例



2.1.5. オフセットのあるビューポートを使用したときの問題

アーリーデプステストでは、アーリーデプスバッファを読み出す際の座標に、ビューポートのオフセットが適用されないというハードウェアの不具合があります。そのため、ビューポートのオフセット(`glViewport()` の x、y)が (0, 0) ではない場合に、アーリーデプステストが正しく行われなくなります。

アーリーデプスバッファの更新は、ビューポートのオフセットが適用された座標に対して行われるため、ビューポートのオフセットが適用されるピクセルのデプス値と、オフセットが適用されないアーリーデプスバッファのデプス値が比較されてしまい

ます。これにより、アーリーデプステストにパスするはずのピクセルがフェイルと判定されてしまい、描画されるべきピクセルが描画されない問題が発生します。フェイルと判定された場合、そのピクセルを含む 8×8 ピクセルが誤って捨てられてしまいます。

この問題を回避するには、以下の 2 つの方法が考えられます。

- ビューポートのオフセットに (0, 0) 以外の値を設定し、アーリーデプステストを無効にします。1 つのシーンでアーリーデプステストの有効・無効を切り替えながら描画すると、アーリーデプステストの効果は落ちますが、通常のデプステストで捨てられるべきピクセルは捨てられるため、描画結果は同じになります。
- ビューポートのオフセットに (0, 0) を設定し、代わりにモデルビュー変換を使って描画領域をずらして、不要な領域をシザリングでカットします。

3. メインメモリに置かれたデータの使用

頂点バッファとテクスチャイメージは VRAM だけでなく、メインメモリに確保した領域にも置くことができます。ただし、ここでいう「メインメモリ」とは、その中でも周辺デバイスからのアクセスに対してアドレスの整合性が保証されている領域である「デバイスメモリ」のことを指しています。ライブラリでメインメモリにコピーを作成しない場合は、データを格納している領域がデバイスメモリ上になければならないことに注意してください。

> **注意:** アドレスのアライメントは、バッファの種別(メモリ領域の使用目的)によって異なります。詳細は「3DS プログラミングマニュアル – グラフィックス基本編」の「アロケータ」にある表「バッファ種別によるアライメントの違い」を参照してください。また、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。

GPU はメインメモリに置かれたデータを直接参照することができますが、領域を確保する際の指定によって動作が異なります。また、メインメモリは VRAM に比べてアクセス速度が劣るため、パフォーマンスに影響を与える可能性があります。例えば、頂点バッファを VRAM とメインメモリに分散して確保した場合、VRAM 上の頂点バッファへのアクセス速度はメインメモリへのアクセス速度まで低下してしまいます。

`glTexImage2D()`、`glCompressedTexImage2D()`、`glCopyTexImage2D()`、`glBufferData()` は *`target`* 引数の指定時に特別なフラグを論理和で渡すことで、GPU のアクセス先の設定や、領域確保時にメインメモリにコピーを作成するかどうかの設定を行うことができます。

表 3-1. GPU のアクセス先指定フラグ

フラグ	アクセス先
`NN_GX_MEM_VRAMA`	GPU は VRAM-A にアクセス
`NN_GX_MEM_VRAMB`	GPU は VRAM-B にアクセス
`NN_GX_MEM_FCRAM`	GPU はメインメモリにアクセス

表 3-2. コピー作成指定フラグ

フラグ	コピーの作成
`GL_COPY_FCRAM_DMP`	メインメモリにコピーを作成する
`GL_NO_COPY_FCRAM_DMP`	メインメモリにコピーを作成しない

設定の組み合わせとしては以下の 4 つに分かれます。

1. メインメモリにアクセスし、メインメモリにコピーを作成する
2. メインメモリにアクセスし、メインメモリにコピーを作成しない
3. VRAM-A(B) にアクセスし、メインメモリにコピーを作成する
4. VRAM-A(B) にアクセスし、メインメモリにコピーを作成しない

`glTexImage2D()`、`glCompressedTexImage2D()`、`glBufferData()` のデフォルトは(1)の設定です。

`glTexImage2D()` のみ、*`pixels`* 引数に `NULL` を渡した場合には 4. の設定で VRAM-B がアクセス先となります。

`glCopyTexImage2D()` には GPU のアクセス先指定のみが可能です。関数内で指定されたメモリ上に領域を確保し、カ

ラーバッファの内容を DMA 転送します。デフォルトは NN_GX_MEM_FCRAM(メインメモリにアクセス)です。

glBufferSubData() は GPU のアクセス先指定とコピー作成指定フラグの設定を glBufferData() から引き継ぎます。

3.1. メインメモリにアクセスし、メインメモリにコピーを作成する

NN_GX_MEM_FCRAM と GL_COPY_FCRAM_DMP との論理和を指定して関数を呼び出した場合です。

以下のような動作をします。

- 関数内でメインメモリ上に領域を確保する
- 確保した領域に CPU がデータをコピーする
- GPU は直接メインメモリ(コピーした領域)にアクセスする

データを格納している領域は関数の終了直後(CPU によるコピー直後)に破棄することができます。データのアドレスに NULL を指定した場合は領域の確保のみが行われ、データのコピーは行われません。

3.2. メインメモリにアクセスし、メインメモリにコピーを作成しない

NN_GX_MEM_FCRAM と GL_NO_COPY_FCRAM_DMP との論理和を指定して関数を呼び出した場合です。

以下のような動作をします。

- GPU は直接メインメモリ(アプリケーションで確保している領域)にアクセスする

データを格納している領域は、描画が完了するまで破棄することができません。データのアドレスに NULL を指定した場合や PICA ネイティブフォーマット以外のテクスチャをロードした場合は GL_INVALID_OPERATION のエラーを生成します。

この設定で glBufferData() が確保する頂点バッファの領域はアプリケーションが管理するメモリであるため、glBufferSubData() は部分領域のデータの更新を行いません。部分領域の更新はアプリケーションで行ってください。また、glBufferSubData() の *data* に、glBufferData() で指定された元のバッファアドレスに *offset* を加算した値が指定されていない場合は GL_INVALID_VALUE のエラーを生成します。

また、GL 関数を呼び出さずに、メインメモリの内容を CPU で動的に更新したデータを GPU に描画させる場合は nngxUpdateBufferLight() を呼び出す必要があります。

3.3. VRAM-A(B) にアクセスし、メインメモリにコピーを作成する

NN_GX_MEM_VRAMA または NN_GX_MEM_VRAMB と、GL_COPY_FCRAM_DMP との論理和を指定して関数を呼び出した場合です。

以下のような動作をします。

- 関数内でメインメモリ上と VRAM 上にそれぞれ領域を確保する
- 確保した領域に CPU がデータをコピーする
- メインメモリ(コピーした領域)から VRAM にデータを DMA 転送する
- GPU は VRAM にアクセスする

データを格納している領域は関数の終了直後(CPU によるコピー直後)に破棄することができます。データのアドレスに NULL を指定した場合は GL_INVALID_OPERATION のエラーを生成します。

3.4. VRAM-A(B) にアクセスし、メインメモリにコピーを作成しない

NN_GX_MEM_VRAMA または NN_GX_MEM_VRAMB と、GL_NO_COPY_FCRAM_DMP との論理和を指定して関数を呼び出した場合です。

以下のような動作をします。

- 関数内で VRAM 上に領域を確保する
- メインメモリ(アプリケーションで確保している領域)から VRAM にデータを DMA 転送する
- GPU は VRAM にアクセスする

データを格納している領域は DMA 転送によるコピーが完了(DMA 転送のコマンドリクエスト完了を確認)するまで破棄することができません。データのアドレスに NULL を指定した場合は領域の確保のみが行われ、DMA 転送は行われません。PICA ネイティブフォーマット以外のテクスチャをロードした場合は GL_INVALID_OPERATION のエラーを生成します。

この設定でロードされた頂点バッファに対して glBufferSubData() を呼び出した場合、アプリケーションは DMA 転送が完了するまで *data* で指定した領域の内容を保証しなければなりません。

3.5. VRAM を有効に利用するには

VRAM-A と VRAM-B へのアクセスは、それぞれ別のチャンネルで行われます。同じタイミングで書き込みや読み込みが発生するバッファは、それぞれを別の VRAM に分けて確保することを推奨します。

アクセスが競合する組み合わせは以下のとおりです。

表 3-3. アクセスが競合する組み合わせ

競合する組み合わせ	発生するタイミング
カラーバッファとデプス(ステンシル)バッファ	レンダリング時全般
カラーバッファとディスプレイバッファ	nngxTransferRenderImage() の実行時
頂点バッファとインデックスバッファ	glDrawElements() の実行時

レンダリング全般でアクセスの発生するカラーバッファとデプス(ステンシル)バッファは別々の VRAM に確保することを強く推奨します。

4. ステートキャッシュ

注意: ステートキャッシュはコマンドキャッシュなどに比べて機能が劣るため廃止されました。

5. 照明モデルの実装例

照明モデルとは 3D シーンでライティングがどのように起こっているかを表す数学的なモデルのことで、光源の色、マテリアルの色、また光がどのように反射されるかといった情報がそこに含まれています。

典型的な照明モデルはいくつかの項－環境光(ambient)、放射光(emissive)、拡散光(diffuse)、鏡面光(specular)－に分解されます。

$$reflected\_light = emissive + ambient + diffuse + specular$$

フラグメントライティングではプライマリカラーとセカンダリカラーの 2 つのカラーが出力されます。プライマリカラーは環境光、放射光、拡散光の和ですが、セカンダリカラーは 2 つの鏡面光の和であり、単純な鏡面光以外の表現にも対応することができます。

5.1. 微細面反射(microfacet Reflection)

microfacet とは肉眼では認識できないほど微細で小さな面のことです。粗い表面の物質が microfacet の集合で構成されていると考えれば、ライトベクトルと視線ベクトルのハーフベクトルに一致する法線を持つ microfacet は視線方向に光を反射させていることになります。つまり、視線方向に光を反射させる microfacet の数が多いほど、より強い反射光となります。

ここで法線の角度を引数にとり、その角度の法線を持つ microfacet の割合を返す分布関数を導入すれば、物質の反射光量はその関数に比例することになります。この関数は microfacet の勾配(slope)がどのように分布しているかを示すものでもあるため、勾配分布(slope distribution)関数とも呼ばれます。

コード 5-1. 微細面反射の実装例

```
GLfloat ms0[] = {1.0f, 1.0f, 1.0f, 1.0f};
GLfloat ms1[] = {0.0f, 0.0f, 0.0f, 1.0f};
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.config"),
        GL_LIGHT_ENV_LAYER_CONFIG0_DMP);
glUniform4fv(glGetUniformLocation(progID, "dmp_FragmentMaterial.specular0"),
        1, ms0);
glUniform4fv(glGetUniformLocation(progID, "dmp_FragmentMaterial.specular1"),
        1, ms1);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledRefl"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledD0"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputD0"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputD0"),
        GL_LIGHT_ENV_NH_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.clampHighlights"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightSource[0].geomFactor1"),
        GL_FALSE);

GLfloat LUT_D0[512];
for( i = 0; i < 256; i++) LUT_D0[i] = slopeDist((float) i / 256.0f);
for( i = 0; i < 255; i++) LUT_D0[i + 256] = LUT_D0[i + 1] - LUT_D0[i];
LUT_D0[511] = slopeDist(1.0f) - LUT_D0[255];
```

```
glBindTexture(GL_LUT_TEXTURE0_DMP, lutTexD0);
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_D0);
glUniform1i(glGetUniformLocation(progID,
        "dmp_FragmentMaterial.samplerD0"), 0);
```

分布 0(D0)の参照テーブルに勾配分布関数のサンプリング値を設定しています。

5.2. ブリン・フォンモデル(Blinn-Phong Model)

Blinn-Phong の反射モデルはべき乗関数を分布関数とする単純な微細面反射モデルです。また、Blinn-Phong の一般化された反射モデル(generalized Blinn-Phong model)は、法線とハーフベクトルの内積を引数にとる、べき乗関数以外の分布関数を使って表現されます。例えば Gaussian 関数を分布関数として用いることができます。

単純な Blinn-Phong の反射モデルであれば、微細面反射の実装例で分布関数 slopeDist() を以下のように定義するだけです。

コード 5-2. 単純な Blinn-Phong モデルでの分布関数

```
float slopeDist(float NH)
{
    return powf(NH, 2.0f);
}
```

セカンダリカラーに 2 つある鏡面光を使って、2 層からなる Blinn-Phong モデルを表現することもできます。これには、分布 1(D1)の参照テーブルに 2 層目の勾配分布関数のサンプリング値を設定し、D0 と D1 の両方を使用するレイヤコンフィグレーションに変更することで対応することができます。

コード 5-3. 2 層からなる Blinn-Phong モデルの実装例

```
GLfloat ms0[] = {1.0f, 1.0f, 1.0f, 1.0f};
GLfloat ms1[] = {1.0f, 1.0f, 1.0f, 1.0f};
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.config"),
        GL_LIGHT_ENV_LAYER_CONFIG2_DMP);
glUniform4fv(glGetUniformLocation(progID, "dmp_FragmentMaterial.specular0"),
        1, ms0);
glUniform4fv(glGetUniformLocation(progID, "dmp_FragmentMaterial.specular1"),
        1, ms1);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledRefl"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledD0"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputD0"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputD0"),
        GL_LIGHT_ENV_NH_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledD1"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputD1"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputD1"),
        GL_LIGHT_ENV_NH_DMP);
```

```
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.clampHighlights"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightSource[0].geomFactor0"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightSource[0].geomFactor1"),
        GL_FALSE);

GLfloat LUT_D0[512], LUT_D1[512];
for( i = 0; i < 256; i++) LUT_D0[i] = slopeDistD0((float) i / 256.0f);
for( i = 0; i < 255; i++) LUT_D0[i + 256] = LUT_D0[i + 1] - LUT_D0[i];
LUT_D0[511] = slopeDistD0(1.0f) - LUT_D0[255];

for( i = 0; i < 256; i++) LUT_D1[i] = slopeDistD1((float) i / 256.0f);
for( i = 0; i < 255; i++) LUT_D1[i + 256] = LUT_D1[i + 1] - LUT_D1[i];
LUT_D1[511] = slopeDistD1(1.0f) - LUT_D1[255];

glBindTexture(GL_LUT_TEXTURE0_DMP, lutTexD0);
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_D0);
glBindTexture(GL_LUT_TEXTURE1_DMP, lutTexD1);
glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_D1);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerD0"), 0);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerD1"), 1);
```

Blinn-Phong モデルについては以下の文献を参照してください。

> Blinn, James F., Models of Light Reflection for Computer Synthesised Pictures, ACM Computer Graphics, SIGGRAPH 1977 Proceedings, 11(4), pp. 192-198

5.3. クック・トーランスモデル（Cook-Torrance Model）

より正確な光の反射量を表現するのが Cook-Torrance のモデルです。このモデルでは、光の反射量が分布関数にフレネル項と呼ばれるファクタを乗算したものに比例すると定義しています。透過した光の屈折と反射が入射角と屈折率の関数で表せることを利用し、さらに物質によって異なる吸光係数を考慮することで様々な物質の反射を表現することができます。フレネル項はその反射をライトベクトルと法線ベクトルの関数として表現したものです。

microfacet のフレネル項を考えるとき、ここでいう法線は microfacet の法線であり、微細面反射と同様にそれはハーフベクトルと一致します。そのため、このモデルをフラグメントライティングで表現する際には、（ハーフベクトルは視線とライトのハーフベクトルであるので）視線ベクトルとハーフベクトルをフレネル項の入力に使用し、分布関数には分母の m^2 を除いた Beckmann 関数を使用します。また、microfacet 同士が影を落とすことも考慮しなければなりませんが、角度によって microfacet からの反射が視点に到達する割合は、ジオメトリファクタ（Cook-Torrance の geometric factor に角度を考慮したもの）としてセカンダリカラーの計算式にすでに含まれています。3DS のジオメトリファクタは Cook-Torrance の geometric factor を以下のように近似したものです。

$$GF = \max\left(0,\ \frac{Lf_i \cdot Nf}{(Lf_i + Vf)^2}\right)$$

Lf_i は i 番目のライトのライトベクトル、Nf と Vf は法線と視線ベクトルです。

フレネル項の屈折率と吸光係数は色に依存します。そのため、フレネル項は RGB の各成分での屈折率と吸光係数で計算

することができ、フラグメントライティングでは各成分の反射として扱うことになります。

以上のことから、Cook-Torrance モデルをフラグメントライティングで表現するには、反射、ディストリビューションファクタ、ジオメトリファクタを利用し、反射の入力に視線ベクトルとハーフベクトルの内積、ディストリビューションファクタの入力に法線とハーフベクトルの内積を利用することがわかります。これが可能なのはレイヤコンフィグ 4、5、7 の 3 つです。また、マテリアルの鏡面光 1 が反射に置き換わり、ライトからの鏡面光 1 で色が決定されることに注意が必要です。

コード 5-4. Cook-Torrance モデルの実装例

```
GLfloat ms[] = {0.0f, 0.0f, 0.0f, 0.0f};
GLfloat ls[] = {1.0f, 1.0f, 1.0f, 1.0f};

glUniform4fv(glGetUniformLocation(progID, "dmp_FragmentMaterial.specular0"),
        1, ms);
glUniform4fv(glGetUniformLocation(progID,
        "dmp_FragmentLightSource[0].specular1"), 1, ls);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledRefl"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.config"),
        GL_LIGHT_ENV_LAYER_CONFIG4_DMP);

glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledD1"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRR"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRG"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRB"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputD1"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRR"),
        GL_LIGHT_ENV_VH_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRG"),
        GL_LIGHT_ENV_VH_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRB"),
        GL_LIGHT_ENV_VH_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputD1"),
        GL_LIGHT_ENV_NH_DMP);
glUniform1f(glGetUniformLocation(progID, "dmp_LightEnv.lutScaleRR"), 2.0f);
glUniform1f(glGetUniformLocation(progID, "dmp_LightEnv.lutScaleRG"), 2.0f);
glUniform1f(glGetUniformLocation(progID, "dmp_LightEnv.lutScaleRB"), 2.0f);
glUniform1f(glGetUniformLocation(progID, "dmp_LightEnv.lutScaleD1"), 2.0f);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.clampHighlights"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightSource[0].geomFactor1"),
        GL_TRUE);

GLfloat LUT_RR[512], LUT_RG[512], LUT_RB[512], LUT_D1[512];
for( i = 0; i < 128; i++) LUT_RR[i] =
        nk_fresnel((float) i / 128.0f, FRESNEL_N_R, FRESNEL_K_R);
for( i = 0; i < 127; i++) LUT_RR[i + 256] = LUT_RR[i + 1] - LUT_RR[i];
LUT_RR[383] = nk_fresnel(1.0f, FRESNEL_N_R, FRESNEL_K_R) - LUT_RR[127];
```

```
for( i = 0; i < 128; i++) LUT_RG[i] =
        nk_fresnel((float) i / 128.0f, FRESNEL_N_G, FRESNEL_K_G);
for( i = 0; i < 127; i++) LUT_RG[i + 256] = LUT_RG[i + 1] - LUT_RG[i];
LUT_RG[383] = nk_fresnel(1.0f, FRESNEL_N_G, FRESNEL_K_G) - LUT_RG[127];

for( i = 0; i < 128; i++) LUT_RB[i] =
        nk_fresnel((float) i / 128.0f, FRESNEL_N_B, FRESNEL_K_B);
for( i = 0; i < 127; i++) LUT_RB[i + 256] = LUT_RB[i + 1] - LUT_RB[i];
LUT_RB[383] = nk_fresnel(1.0f, FRESNEL_N_B, FRESNEL_K_B) - LUT_RB[127];

for( i = 0; i < 128; i++) LUT_D1[i] = beckmann((float) i / 128.0f, 1.0f);
for( i = 0; i < 127; i++) LUT_D1[i + 256] = LUT_D1[i + 1] - LUT_D1[i];
LUT_D1[383] = beckmann(1.0f, 1.0f) - LUT_D1[127];

glBindTexture(GL_LUT_TEXTURE0_DMP, lutTexRR);
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_RR);
glBindTexture(GL_LUT_TEXTURE1_DMP, lutTexRG);
glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_RG);
glBindTexture(GL_LUT_TEXTURE2_DMP, lutTexRB);
glTexImage1D(GL_LUT_TEXTURE2_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_RB);
glBindTexture(GL_LUT_TEXTURE3_DMP, lutTexD1);
glTexImage1D(GL_LUT_TEXTURE3_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_D1);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerRR"), 0);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerRG"), 1);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerRB"), 2);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerD1"), 3);
```

`nk_fresnel()`、`beckmann()` はそれぞれ吸光係数を考慮したフレネル反射関数、beckmann 関数です。

`dmp_LightEnv.lutScaleXX` で参照テーブルからの出力を倍化しています。

Cook-Torrance モデルについては以下の文献を参照してください。

Cook, Robert L., and Torrance, Kenneth E., A Reflectance Model for Computer Graphics, ACM Computer Graphics, SIGGRAPH 1981 Proceedings, 15(4), pp. 307-316.

ジオメトリファクタについては以下の文献を参照してください。

Alan Watt, 3D Computer Graphics, 3rd edition, Addison-Wesley Publishing Ltd, Addison-Wesley Publishing Company Inc., 2000, pp. 216

様々な物質の屈折率と吸光係数については以下の文献を参照してください。

Glassner, Andrew S., Principles of Digital Image Synthesis, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1995.

5.4. シュリック異方性モデル（Schlick Anisotropic Model）

これまでに紹介してきた照明モデルは、向きに対してすべてのサーフェスが同じ性質を持っている（等方性: isotropic）と仮定し、反射は法線に対して常に対称となっていました。それに対し、織物の繊維が特定の方向で反射を強めるように、向き

に対してサーフェスが異なる性質を持つ場合を異方性(anisotropic)と呼びます。

異方性のあるサーフェスの光の散乱特性はサーフェスに沿った方向の関数として変化します。Schlick は異方性反射を表現するために、分布関数を 2 つの関数で構成することを提案しました。1 つは法線とハーフベクトルに依存し、もう 1 つは接線とハーフベクトルの接平面への投影ベクトルのなす角 Φ に依存する関数で、以下の式で表されます。

$$D = Z(N \cdot H) \times A(\cos(\Phi))$$

$$Z(t) = \frac{r}{(1 + rt^2 - t^2)^2}$$

$$A(\omega) = \sqrt{\frac{p}{p^2 - p^2\omega^2 + \omega^2}}$$

r はサーフェスの荒さを特徴づけるパラメータ、p はサーフェスの異方性を特徴づけるパラメータです。

このモデルでの geometric factor は Cook-Torrance モデルのものと異なりますが、フラグメントライティングで使用しているジオメトリファクタとは同様の特性を持っています。

コード 5-5. Schlick Anisotropic モデルの実装例

```
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledRefl"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.config"),
        GL_LIGHT_ENV_LAYER_CONFIG7_DMP);

glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledD1"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRR"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRG"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRB"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputD1"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRR"),
        GL_LIGHT_ENV_NH_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRG"),
        GL_LIGHT_ENV_NH_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRB"),
        GL_LIGHT_ENV_NH_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputD1"),
        GL_LIGHT_ENV_CP_DMP);
glUniform1f(glGetUniformLocation(progID, "dmp_LightEnv.lutScaleRR"), 1.0f);
glUniform1f(glGetUniformLocation(progID, "dmp_LightEnv.lutScaleRG"), 1.0f);
glUniform1f(glGetUniformLocation(progID, "dmp_LightEnv.lutScaleRB"), 1.0f);
glUniform1f(glGetUniformLocation(progID, "dmp_LightEnv.lutScaleD1"), 1.0f);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.clampHighlights"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightSource[0].geomFactor0"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightSource[0].geomFactor1"),
```

```
        GL_FALSE);

GLfloat LUT_RR[512], LUT_RG[512], LUT_RB[512], LUT_D1[512];
for( i = 0; i < 256; i++) LUT_RR[i] =
        z_schlick(0.7f, (float) i / 256.0f, true);
for( i = 0; i < 255; i++) LUT_RR[i + 256] = LUT_RR[i + 1] - LUT_RR[i];
LUT_RR[511] = z_schlick(0.7f, 1.0f, true) - LUT_RR[255];

for( i = 0; i < 256; i++) LUT_RG[i] =
        z_schlick(0.7f, (float) i / 256.0f, true);
for( i = 0; i < 255; i++) LUT_RG[i + 256] = LUT_RG[i + 1] - LUT_RG[i];
LUT_RG[511] = z_schlick(0.7f, 1.0f, true) - LUT_RG[255];

for( i = 0; i < 256; i++) LUT_RB[i] =
        z_schlick(0.7f, (float) i / 256.0f, true);
for( i = 0; i < 255; i++) LUT_RB[i + 256] = LUT_RB[i + 1] - LUT_RB[i];
LUT_RB[511] = z_schlick(0.5f, 1.0f, true) - LUT_RB[255];

for( i = 0; i < 256; i++) LUT_D1[i] =
        a_schlick(0.0015f, (float) i / 256.0f, true);
for( i = 0; i < 255; i++) LUT_D1[i + 256] = LUT_D1[i + 1] - LUT_D1[i];
LUT_D1[511] = a_schlick(0.0015f, 1.0f, true) - LUT_D1[255];

glBindTexture(GL_LUT_TEXTURE0_DMP, lutTexRR);
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_RR);
glBindTexture(GL_LUT_TEXTURE1_DMP, lutTexRG);
glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_RG);
glBindTexture(GL_LUT_TEXTURE2_DMP, lutTexRB);
glTexImage1D(GL_LUT_TEXTURE2_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_RB);
glBindTexture(GL_LUT_TEXTURE3_DMP, lutTexD1);
glTexImage1D(GL_LUT_TEXTURE3_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_D1);
glUniform1i(glGetUniformLocation(progID,
        "dmp_FragmentMaterial.samplerRR"), 0);
glUniform1i(glGetUniformLocation(progID,
        "dmp_FragmentMaterial.samplerRG"), 1);
glUniform1i(glGetUniformLocation(progID,
        "dmp_FragmentMaterial.samplerRB"), 2);
glUniform1i(glGetUniformLocation(progID,
        "dmp_FragmentMaterial.samplerD1"), 3);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.bumpMode"),
        GL_LIGHT_ENV_BUMP_AS_BUMP_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.bumpSelector"),
        GL_TEXTURE0);
```

`z_schlick()`、`a_schlick()` は schlick 関数です。

バンプマッピングを行っているのは、摂動された接線が入力で必要なためです。

シュリック異方性モデルについては以下の文献を参照してください。

Schlick, Christophe, An Inexpensive BRDF Model for Physically-Based Rendering, Computer Graphics Forum, 13(3),

pp. 233-246 (1994).

5.5. 表面下散乱モデル(Subsurface-Scattering Model)

肌、ロウソクのロウ、大理石のような半透明の物体は強い表面下散乱を持ち、光は内部で多くの散乱を起こした後に外部に放出されています。このような物体では光の反射位置は入射位置とは異なるため、これまで紹介したモデルと同じアプローチでは表現することができません。

Jensen らが提案した拡散光理論をベースにしたモデルは、入射してくる光と拡散光反射関数をサーフェスのある領域で積分するというものです。その積分領域は平均自由行程(mean free-path)ほどの大きさであり、典型的な素材では 1 から数ミリメートルになります。

人間の手のように、半透明の素材でできたオブジェクトは、周囲が暗い環境でたった 1 つの光源で照らされると、凸面部分の透明度が最も高く感じられます。これは複雑な照明下においても同様で、すなわち表面下散乱を持つオブジェクトの描画では凸面部分のレンダリングが最も重要であると考えることができます。

そこで凸面部分を半径 S の球の一部分とみなして、この部分に Jensen らによる式の積分を適用(半径 S は ″mean free-path″ の長さよりずっと大きい)し、荒いサーフェスを持つオブジェクトであると考えると、その反射モデルは以下の式で与えられます。

$$Subsurface\_scattering = Rrgb(L \cdot N) \times T(V \cdot N)$$

$Rrgb(L \cdot N)$ は Lambertian(拡散光)項と wrapping(光のあたらない表面領域への光の侵入を表す)項の 2 項を足し合わせたものです。

$$Rrgb(L \cdot N) = r(L \cdot N) + W(L \cdot N)$$
$$r = 0.5\alpha'\left(e^{-\beta} + e^{-\beta'}\right)$$
$$\beta = \sqrt{3(1-\alpha')}$$
$$\beta' = \beta B$$
$$B = \left(1 + \frac{4}{3}\frac{(1+Fdr)}{(1-Fdr)}\right)$$
$$W(L \cdot N) = \frac{\alpha' Y i_0}{4\pi(1+i_1 h)}$$
$$Y = l\frac{\sqrt{1-(L \cdot N)^2}}{S}$$
$$h = \frac{L \cdot N}{Y}$$
$$i_0 = \sqrt{2\pi}\left(e^{-\beta}\beta^{-0.5} + Be^{-\beta'}\beta'^{-0.5}\right)$$
$$i_1 = \frac{\pi\left(e^{-\beta} + e^{-\beta'}\right)}{i_0}$$

α'(albedo)と Fdr については、以下の文献を参照してください。

Jensen, H. W., Marschner, S., Levoy, M., and Hanrahan, P., A practical model for subsurface light transport, SIGGRAPH 2001 Proceedings, E. Fiume, Ed., Annual Conference Series, pp. 511-518.

コード 5-6. 表面下散乱モデルの実装例

```
 /* Lighting Environment */
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.config" ),
    GL_LIGHT_ENV_LAYER_CONFIG7_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledRefl" ),
    GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.fresnelSelector" ),
    GL_LIGHT_ENV_PRI_SEC_ALPHA_FRESNEL_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.clampHighlights" ),
    GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledD0" ),
    GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledD1" ),
    GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRR" ),
    GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRG" ),
    GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRB" ),
    GL_FALSE);
/* below we use GL_TRUE to have non-zero for negative NV because negative NV values
happen on silhouette */
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputD1" ),
    GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputD0" ),
    GL_FALSE);
/* below we use GL_TRUE to have non-zero for negative NV because negative NV values
happen on silhouette */
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputFR" ),
    GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRR" ),
    GL_LIGHT_ENV_LN_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRG" ),
    GL_LIGHT_ENV_LN_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRB" ),
    GL_LIGHT_ENV_LN_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputD1" ),
    GL_LIGHT_ENV_NV_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputD0" ),
    GL_LIGHT_ENV_NH_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputFR" ),
    GL_LIGHT_ENV_NV_DMP);

/* Material */
GLfloat ms2[] = {0.28f, 0.28f, 0.28f, 1.f};
glUniform4fv(glGetUniformLocation(progID, "dmp_FragmentMaterial.specular1"), 1,
ms2);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerRR"), 0);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerRG"), 1);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerRB"), 2);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerD1"), 3);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerD0"), 4);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerFR"), 5);
```

```
/* Light Source */
GLfloat ld0[] = {1.f, 1.f, 1.f, 1.f}; /* light0 diffuse */
GLfloat ls0[] = {0.35f, 0.35f, 0.35f, 1.f}; /* light0 specular */
GLfloat ls1[] = {0.28f, 0.28f, 0.28f, 1.f}; /* light0 specular2 */
glUniform4fv(glGetUniformLocation(progID, "dmp_FragmentLightSource[0].diffuse"),
    1, ld0);
glUniform4fv(glGetUniformLocation(progID,
    "dmp_FragmentLightSource[0].specular0"), 1, ls0);
glUniform4fv(glGetUniformLocation(progID,
    "dmp_FragmentLightSource[0].specular1"), 1, ls1);
glUniform1i(glGetUniformLocation(progID,
    "dmp_FragmentLightSource[0].geomFactor0" ), GL_FALSE);
glUniform1i(glGetUniformLocation(progID,
    "dmp_FragmentLightSource[0].geomFactor1" ), GL_FALSE);

/* Texture Combiner 0 */
glUniform1i(glGetUniformLocation(progID, "dmp_TexEnv[0].combineRgb"),
    GL_ADD);
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_FRAGMENT_PRIMARY_COLOR_DMP, GL_FRAGMENT_SECONDARY_COLOR_DMP,
    GL_CONSTANT);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].srcAlpha"),
    GL_CONSTANT, GL_CONSTANT, GL_CONSTANT);

/* Texture Combiner 1 */
glUniform1i(glGetUniformLocation(progID, "dmp_Texture[0].samplerType" ),
    GL_TEXTURE_CUBE_MAP);
glUniform1i(glGetUniformLocation(progID, "dmp_TexEnv[1].combineRgb"),
    GL_MULT_ADD_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_TexEnv[1].combineAlpha"),
    GL_REPLACE);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[1].operandRgb"),
    GL_SRC_COLOR, GL_SRC_ALPHA, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[1].operandAlpha"),
    GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[1].srcRgb"),
    GL_TEXTURE0, GL_FRAGMENT_PRIMARY_COLOR_DMP, GL_PREVIOUS);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[1].srcAlpha"),
    GL_PREVIOUS, GL_PREVIOUS, GL_PREVIOUS);

/* LUT */
GLfloat qlut[3][512], lut[512];
int j, co;

GLuint lutids[6];
glGenTextures(6, lutids);
```

```
/* RR, RG, RB */
for (co = 0; co < 3; co++) {
    for (j = 0; j < 128; j++) {
        LN = (float) j/128.f;
        qlut[co][j] = calc_lamb(LN, co) + calc_wrap(LN, co);
    }
    for (j = 128; j < 256; j++) {
        LN = (float) (j - 256) /128.f;
        qlut[co][j] = calc_wrap(LN, co);
    }
    for (j = 0; j < 127; j++)
        qlut[co][j + 256] = qlut[co][j + 1] - qlut[co][j];
    qlut[co][127 + 256] = calc_lamb(1.f, co) + calc_wrap(1.f, co) -
        qlut[co][127];
    for (j = 128; j < 255; j++)
        qlut[co][j + 256] = qlut[co][j + 1] - qlut[co][j];
    qlut[co][255 + 256] = qlut[co][0] - qlut[co][255];
}
glBindTexture(GL_LUT_TEXTURE0_DMP, lutids[0]);
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
    GL_LUMINANCEF_DMP, GL_FLOAT, qlut[0]);
glBindTexture(GL_LUT_TEXTURE1_DMP, lutids[1]);
glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
    GL_LUMINANCEF_DMP, GL_FLOAT, qlut[1]);
glBindTexture(GL_LUT_TEXTURE2_DMP, lutids[2]);
glTexImage1D(GL_LUT_TEXTURE2_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
    GL_LUMINANCEF_DMP, GL_FLOAT, qlut[2]);
/* D1 */
for (j = 0; j < 256; j++) lut[j] = calc_t((float) j / 255.9375f);
for (j = 0; j < 255; j++) lut[j + 256] = lut[j + 1] - lut[j];
lut[255 + 256] = calc_t(1.f) - lut[255];
glBindTexture(GL_LUT_TEXTURE3_DMP, lutids[3]);
glTexImage1D(GL_LUT_TEXTURE3_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
    GL_LUMINANCEF_DMP, GL_FLOAT, lut);
/* D0 */
memset(lut, 0, sizeof(lut));
for (j = 0; j < 128; j++) lut[j] = beckmann((float)j / 128.f, 0.5f);
for (j = 128; j < 256; j++) lut[j] = 0.f;
for (j = 0; j < 127; j++) lut[j + 256] = lut[j + 1] - lut[j];
lut[127 + 256] = 1.f - lut[127];
for (j = 128; j < 256; j++) lut[j + 256] = 0;
glBindTexture(GL_LUT_TEXTURE4_DMP, lutids[4]);
glTexImage1D(GL_LUT_TEXTURE4_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
    GL_LUMINANCEF_DMP, GL_FLOAT, lut);
/* FR */
memset(lut, 0, sizeof(lut));
for (j = 0; j < 256; j++)
lut[j] = r_fresnel((float)j / 255.9375f, 2.f, 0.35f, 0.f);
for (j = 0; j < 255; j++) lut[j + 256] = lut[j + 1] - lut[j];
    lut[255 + 256] = r_fresnel(1.f, 2.f, 0.35f, 0.f) - lut[255];
glBindTexture(GL_LUT_TEXTURE5_DMP, lutids[5]);
glTexImage1D(GL_LUT_TEXTURE5_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
    GL_LUMINANCEF_DMP, GL_FLOAT, lut);
```

calc_lamb、calc_wrap、calc_t はそれぞれ、r(L・N)、W(L・N)、T(L・N) を計算する関数です。

反射モデルは RR、RG、RB および D1 の参照テーブルで表されています。D0 はスペキュラ、FR は周辺光からの影響を表しています。

コンバイナ 0 でオブジェクトのプライマリカラーと反射モデルで計算されたセカンダリカラーを足し合わせています。コンバイナ 1 においてカラーのオペランドに GL_MULT_ADD_DMP を指定することで、周辺光からの影響を計算しながらオブジェクトのカラーと足し合わせています。

5.6. トゥーンシェーディング（Toon-Shading）

トゥーンシェーディングは、色が一定となる領域が 2 段階から 3 段階程度になるように陰影をつける表現です。トゥーンシェーディングを行うには、内積値 N・H または L・N の入力に対して範囲ごとに一定値を出力する関数を用意します。その関数で作成された参照テーブルから出力される段階的な値と定数カラーを乗算することで、アニメ調のはっきりした陰影表現が可能となります。

ライトの位置関係のみで陰影が変化する場合は、入力する内積値に L・N（GL_LIGHT_ENV_LN_DMP）を選択し、視線方向も加味する場合は N・H（GL_LIGHT_ENV_NH_DMP）を選択してください。

ライトの鏡面光 1 の明度だけを変化させるならば反射（RR）の参照テーブルに値を設定し、反射の各成分がその参照テーブルを使用するレイヤコンフィグを指定するだけです。段階の分け方は同じでも、各成分がそれぞれの参照テーブルを使用すれば極端な色味の変化を表現することも可能です。

入力する内積値に N・V（GL_LIGHT_ENV_NV_DMP）を使用し、ディストリビューションファクタの参照テーブル（D0, D1）を入力値 0.0 付近で 0.0、それ以外は 1.0 にすれば、視線と法線の向きが極端に異なる領域が暗く（黒く）なることで鏡面光のみでトゥーンシェーディングを行っているオブジェクトの輪郭を描画することができます。

コード 5-7. トゥーンシェーディングの実装例

```
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.config"),
        GL_LIGHT_ENV_LAYER_CONFIG2_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledRefl"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputD0"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputD1"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRR"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRR"),
        GL_LIGHT_ENV_LN_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputD0"),
        GL_LIGHT_ENV_LN_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputD1"),
        GL_LIGHT_ENV_NV_DMP);
glUniform1i(glGetUniformLocation(progID,
        "dmp_FragmentLightSource[0].geomFactor0"), GL_FALSE);
glUniform1i(glGetUniformLocation(progID,
        "dmp_FragmentLightSource[0].geomFactor1"), GL_FALSE);
glUniform1i(glGetUniformLocation(progID,
        "dmp_FragmentLightSource[0].spotEnabled"), GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.clampHighlights"),
```

```
        GL_FALSE);

float delta[] = {1.0f, 0.7f, 0.5f, -1.0f};
for (i = 0, j = 127; j >= 0; j--) {
    LN = (float) j / 128.0f ;
    if (LN > delta[i]) lut[j] = previous;
    else {
        lut[j] = LN; previous = lut[j]; i++;
    }
}
for (j = 0 ; j < 127; j++) lut[j + 256] = lut[j+1] - lut[j];
lut[127 + 256] = 0.0f;
for (i = 0, j = 255; j >= 128; j--) {
    LN = (float)(j - 256) /128.0f;
    if (LN > delta[i]) lut[j] = previous;
    else {
        lut[j] = LN; previous = lut[j]; i++;
    }
}
for (j = 128; j < 255; j++) lut[j + 256] = lut[j+1] - lut[j];
lut[255 + 256] = lut[0] - lut[255];
glBindTexture(GL_LUT_TEXTURE0_DMP, lutids[0]);
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, lut);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerRR"), 0);

float highlight_eps = 0.01f;
for (j = 0; j < 256; j++)
    if ((float) j / 256.0f <= 1.0f - highlight_eps) lut[j] = 0.0f;
    else                                            lut[j] = 1.0f;
for (j = 0; j < 255; j++) lut[j + 256] = lut[j+1] - lut[j];
lut[255 + 256] = 0.0f;
glBindTexture(GL_LUT_TEXTURE1_DMP, lutids[1]);
glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, lut);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerD0"), 1);

float outline = 0.15f;
for (j = 0; j < 256; j++)
    if ((float) j / 256.0f < outline) lut[j] = 0.0f;
    else                              lut[j] = 1.0f;
for (j = 0; j < 255; j++) lut[j + 256] = lut[j+1] - lut[j];
lut[255 + 256] = 1.0f - lut[255];
glBindTexture(GL_LUT_TEXTURE2_DMP, lutids[2]);
glTexImage1D(GL_LUT_TEXTURE2_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, lut);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerD1"), 2);
```

delta が段階的な変化を起こす内積値の配列、highlight_eps がハイライトとなる内積値の範囲、outline が影となる内積値の範囲です。これらの値でトゥーンシェーディングの表現を変化させることができます。

6. 立体視表示

3DS に搭載されている 2 つの LCD のうち、上画面の LCD は特殊な器具を用いることなく裸眼による立体視表示を行うことができます。

立体視表示を行うには左目用と右目用の 2 枚の画像をレンダリングする必要がありますが、CTR-SDK では通常の表示で作成したカメラ行列から 2 つの画像をレンダリングするためのカメラ行列を計算する ULCD ライブラリを用意しています。
様々なアプリケーションで立体視表示を実現する方法を統一するためにも、レンダリング時に ULCD ライブラリで算出したカメラ行列を使用してください。

この章では、立体視表示の原理、アプリケーションでの実装方法、ステレオカメラとの連動について説明します。

便宜上、以下の立体視表示に関する用語を以降の説明で使用しています。

現実空間
立体視表示には、プレイヤーと上画面の LCD 表面との距離などが関係しています。それらとアプリケーション内で構築する空間とを区別するために、プレイヤーや 3DS が存在する空間のことを現実空間と呼びます。

仮想空間
現実空間に対して、アプリケーション内で構築される空間のことを仮想空間と呼びます。

ベースカメラ
アプリケーションがシーンに応じて設定・作成するカメラをベースカメラと呼びます。ベースカメラの情報をもとに、ULCD ライブラリは左目用、右目用のカメラ行列を算出します。

基準面
立体視表示時の、仮想空間における 3DS の LCD 表面上に像を結ぶ平面を基準面と呼びます。基準面は、カメラのビューボリュームの断面のひとつです。

3D ボリューム
立体視表示の強度を調節するために 3DS に搭載されているスイッチです。立体視が苦手な方でも快適に立体視表示ができるように、また長時間のプレイで疲れを感じたときなどに映像を調節することができるように、スライド式のスイッチとなっています。ULCD ライブラリでは、その入力値が仮想空間上に生成した左右 2 つのカメラ間の距離を調整するために用いられます。

最適視認位置
立体視表示でプレイヤーが最適な立体感を得ることのできる視聴位置のことです。

6.1. 立体視表示の原理

立体視表示は、基本的に左右の目の視差を考慮してオブジェクトをレンダリングすることで、オブジェクトと視点との距離感を生み出しています。ULCD ライブラリには、視差を考慮したレンダリングを行うためのカメラ行列を算出する方法として、ベースカメラの設定を極力維持する方法(アプリケーション優先)とベースカメラの設定を必要に応じて自動的に変更する方法(現実感優先)の 2 種類を用意しています。

この節ではそれぞれの算出方法の原理について説明します。

6.1.1. 前提条件

ULCD ライブラリは、立体視表示で使用するカメラ行列を以下の条件を前提にして計算しています。

- プレイヤーの両目の間隔 $Dist_{eye}$ を 62 mm と仮定します。
- LCD 表面(中心)とプレイヤーの両目との距離を $Dist_{e2d}$ とします。
- 先行研究によって得られている知見により、人間の目にとって自然に奥行きを感じることができる限界の深さを $Depth_{ltd}$ とします。また、この深さを表現するために必要となる、現実空間における限界視差(基準面上で必要な視差)を P_{r_ltd} とします。**プレイヤーから見て、奥方向(奥行き)と手前方向(飛び出し)それぞれの視差の上限はガイドラインで定められています。**
- 上画面 LCD の短辺の長さを Len_{disp} とします。

6.1.2. 左右のカメラ間の距離と各カメラのビューボリュームの算出方法

カメラ行列を計算するための入力として、以下の情報を必要とします。

- 立体視用の視差画像を生成するためのベースとなるビュー行列 $View_{base}$
- 左目と右目用カメラのビューボリュームを生成するためのベースとなるプロジェクション行列 $Proj_{base}$
- 仮想空間において、カメラ位置から LCD 表面上に位置させたい点までの距離 D_{level}
- 立体具合を調整するための係数 D_r(値の範囲は 0.0 ～ 1.0)

$Proj_{base}$ から逆算できるビューボリュームのパラメータ(left, right, bottom, top, near, far)をそれぞれ l_{base}、r_{base}、b_{base}、t_{base}、n_{base}、f_{base} としたとき、これらのパラメータと D_{level} から、基準面の幅と高さを求めることができます。

基準面の幅:

$$W_{level} = | r_{base} - l_{base} | \times D_{level} / n_{base}$$

基準面の高さ:

$$H_{level} = | t_{base} - b_{base} | \times D_{level} / n_{base}$$

この基準面の幅と高さ、そして LCD の実際の寸法から、現実空間のスケールを仮想空間のものに変換する係数を求めておきます。

$$Scale_{r2v} = H_{level} / Len_{disp}$$

6.1.2.1. アプリケーション優先の算出方法

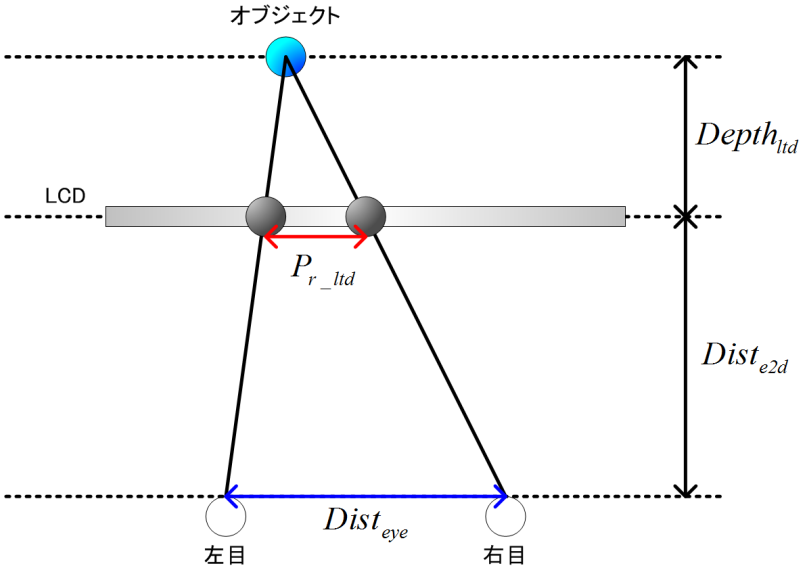
ベースカメラによる見え方(アプリケーションが本来想定している見え方)を極力維持して、左目用と右目用の画像を描画するためのカメラ行列を生成します。

前提条件から、現実空間における限界視差(不自然に見えない状態を維持する、最も大きい視差)P_{r_ltd} を算出することができます。

$$P_{r\_ltd} = Dist_{eye} \times (Depth_{ltd} / (Dist_{e2d} + Depth_{ltd}))$$

下図に計算式中の項目と位置関係を示します。

図 6-1. 現実空間における限界視差



$Scale_{r2v}$ を用いて P_{r_ltd} を仮想空間における限界視差 P_{v_ltd} に変換することができます。

$$P_{v\_ltd} = P_{r\_ltd} \times Scale_{r2v}$$

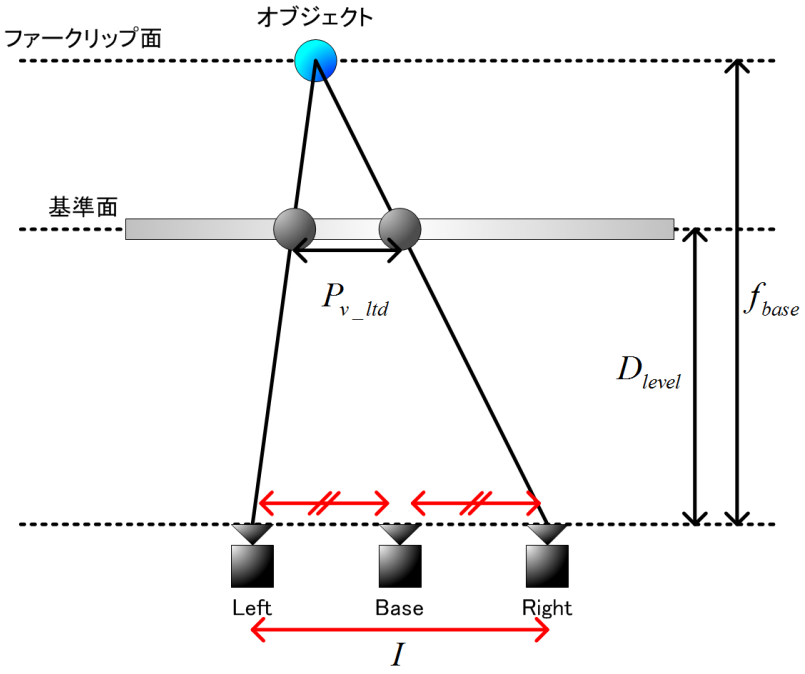
ベースカメラに設定されている奥行き方向の位置を変えずに、ファークリップ面のオブジェクトを表示するための基準面における視差がこの限界視差となるような右目用・左目用カメラの間隔 I を求めます。この計算には D_{level} を利用します。なお、ファークリップ面が異常な位置の場合（基準面よりも手前になるなど）は 0 とします。

I を求める計算式は以下のとおりです。

$$I = P_{v\_ltd} \times (f_{base} / (f_{base} - D_{level}))$$

それらの関係を図示します。

図 6-2. 仮想空間における左右のカメラの間隔



この算出方法ではベースカメラの位置を動かさないため、ビューボリュームに関するパラメータは変化しません。

ニアクリップ面の幅: $W_n = | r_{base} - l_{base} |$

ニアクリップ面の高さ: $H_n = | t_{base} - b_{base} |$

6.1.2.2. 現実感優先の算出方法

基準面のオブジェクトの見え方を現実空間における見え方に合わせ、プレイヤーが LCD から $Dist_{e2d}$ 離れた位置から見て自然な形で立体視を行うことができるような、左目用と右目用の画像を描画するためのカメラ行列を生成します。

現実空間におけるプレイヤーの両眼の間隔と LCD の位置関係を、仮想空間の左右 2 つのカメラと基準面の位置関係に反映させます。その過程でベースカメラに設定された各パラメータは自動的に変更されることになります。

LCD 表面とプレイヤーの両目との距離 $Dist_{e2d}$ を仮想空間におけるスケールに変換した D_{level_new} を求めます。

$$D_{level\_new} = Dist_{e2d} \times Scale_{r2v}$$

求めた距離と D_{level} 、そして $Proj_{base}$ から得られる各クリップ面への距離から、新しく生成するカメラのクリップ面への距離 n_{new} と f_{new} を求めることができます。

$$n_{new} = D_{level\_new} - (D_{level} - n_{base}),\ f_{new} = D_{level\_new} + (f_{base} - D_{level})$$

計算の結果、新しいニアクリップ面がカメラよりも奥に位置した場合は以下のように補正します。

$$n_{new} = D_{level\_new} \times 0.01$$

また、ファークリップ面がニアクリップ面よりも手前に位置した場合は以下のように補正します。

$$f_{new} = n_{new} \times 2.0$$

基準面は動かさないため、これまでに求めた値を満たすようにベースカメラを前後に動かすことになります。

次に、動かした結果変化した、ニアクリップ面の大きさ(幅、高さ)を求めます。さらに、ニアクリップ面の範囲(left, right, top, bottom)を再計算します。

ニアクリップ面の幅

$$W_{n\_new} = W_{level} \times n_{new} / D_{level\_new}$$

ニアクリップ面の高さ

$$H_{n\_new} = H_{level} \times n_{new} / D_{level\_new}$$

ニアクリップ面の範囲

$$l_{new} = tmp \times l_{base},\ r_{new} = tmp \times r_{base},$$

$$t_{new} = tmp \times t_{base},\ b_{new} = tmp \times b_{base}$$

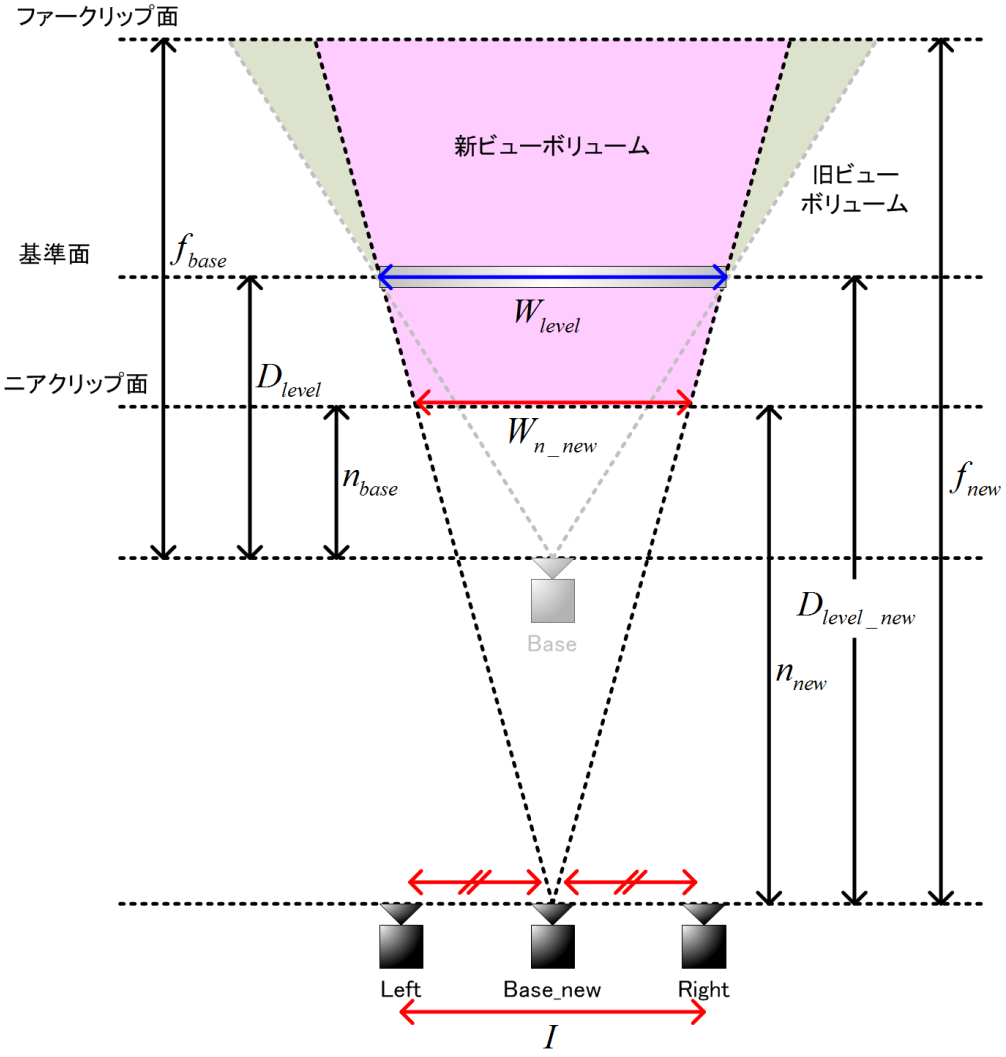
$$tmp = H_{n\_new} / | t_{base} - b_{base} |$$

プレイヤーの両目の間隔を仮想空間における左右 2 つのカメラ間距離 I に反映させます。

$$I = Dist_{eye} \times Scale_{r2v}$$

下図に計算式中の項目と位置関係を示します。

図 6-3. 現実感優先の算出方法



6.1.3. プロジェクション行列の生成

計算で求めた左右のカメラ間距離 I に対して、調整係数 D_r と 3DS 本体の 3D ボリュームの入力値 vol を反映させます。3D ボリュームを最低値にした場合は立体感がなくなり、左右のカメラはベースカメラと一致します。

$$I = I \times vol \times D_r \quad (0.0 \leq vol \leq 1.0)$$

ベースカメラのビューボリュームに関するパラメータはどちらの算出方法を利用したかで異なりますが、便宜上以降の計算式ではニアクリップ面の幅を W_n、ニアクリップ面の高さを H_n、ベースカメラから基準面への距離を D_{level}、ビューボリュームに関するパラメータ(left、right、top、bottom、near、far)を l, r, t, b, n, f のように、統一して表記しています。左右のカメラそれぞれのビューボリュームを求めるために、ニアクリップ面での視差 P_n を求めます。ただし前提条件として、左右のカメラがベースカメラの位置から均等に離れていると仮定しています。

$$P_n = I \times 0.5 \times ((D_{level} - n) / D_{level})$$

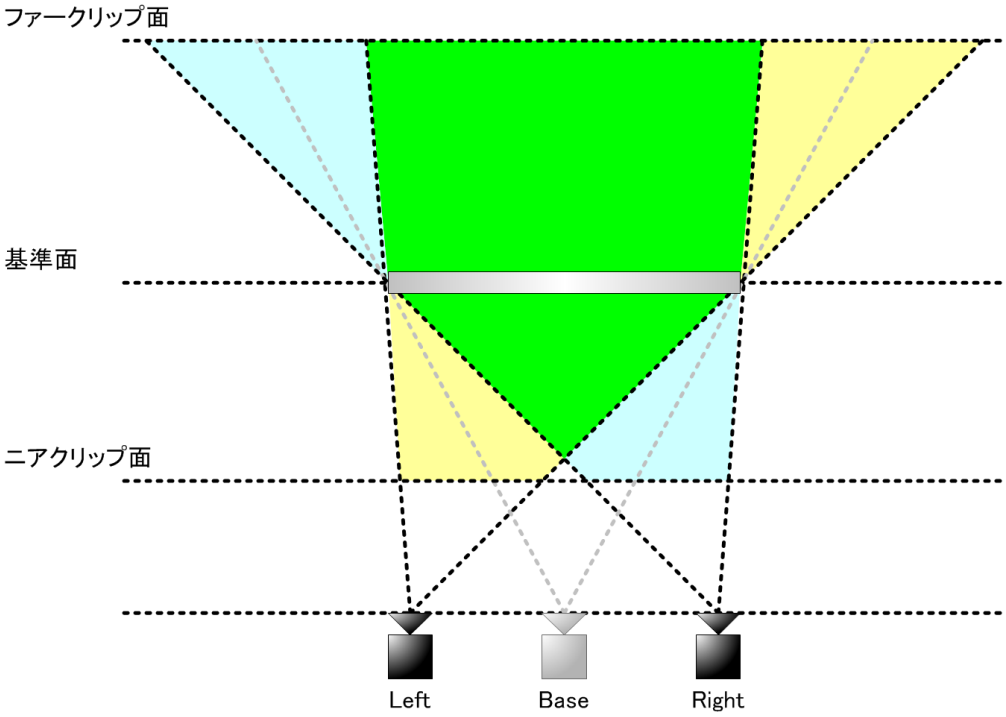
この視差を元に、左右のカメラそれぞれのビューボリュームにおけるニアクリップ面の位置を求めます。左右のカメラは横方向のみに動かしたものであるため、top、bottom、near、far は共通です。

左カメラのニアクリップ面位置: $l_{left} = (l - P_n) + I \times 0.5$, $r_{left} = (r - P_n) + I \times 0.5$

右カメラのニアクリップ面位置: $l_{right} = (l + P_n) - I \times 0.5$, $r_{right} = (r + P_n) - I \times 0.5$

以上から導かれたパラメータに基づき、左右のカメラに関してプロジェクション行列を計算します。下図は、今回求めたビューボリュームを図示したものです。立体視を行うためには、ビューボリュームが重なった領域(図内中央)にオブジェクトを配置する必要があります。

図 6-4. 左右カメラのビューボリューム



6.1.4. ビュー行列の生成

左右のカメラ間距離 I と $View_{base}$ から導かれるベースカメラの情報を用いて、左右のカメラそれぞれのビュー行列を生成します。導かれるベースカメラの情報は、位置 Pos_{base}、向き Dir_{base}、右手方向 E_{right} の 3 つで、これらは三次元ベクトルであり、Dir_{base} と E_{right} は長さ 1 の単位ベクトルです。

現実感優先の算出方法でカメラ間距離を求めていた場合、左右のカメラの位置がベースカメラに対して前後(奥行き方向)に動いている可能性があります。従ってベースカメラの位置は、以下のとおりとなります。

$$Pos_{base} = Pos_{base} - (D_{level\_new} - D_{level}) \times Dir_{base}$$

ベースカメラの位置は左右のカメラの中間であることから、ベースカメラの位置に対して左右のカメラ間距離の半分を加えたもの、あるいは引いたものが左右のカメラの位置となります。左カメラ、右カメラの位置をそれぞれ Pos_{left}、Pos_{right} とし、注視点の方向(カメラの向きさえ分かればよいため、注視点の位置は厳密に求める必要はありません)をそれぞれ Tgt_{left}、Tgt_{right} とすると、

$$Pos_{left} = Pos_{base} - I \times 0.5 \times E_{right} \ , \ Tgt_{left} = Pos_{left} + Dir_{base}$$

$$Pos_{right} = Pos_{base} + I \times 0.5 \times E_{right} \ , \ Tgt_{right} = Pos_{right} + Dir_{base}$$

となります。以上から、左右のカメラそれぞれのビュー行列を生成することができます。

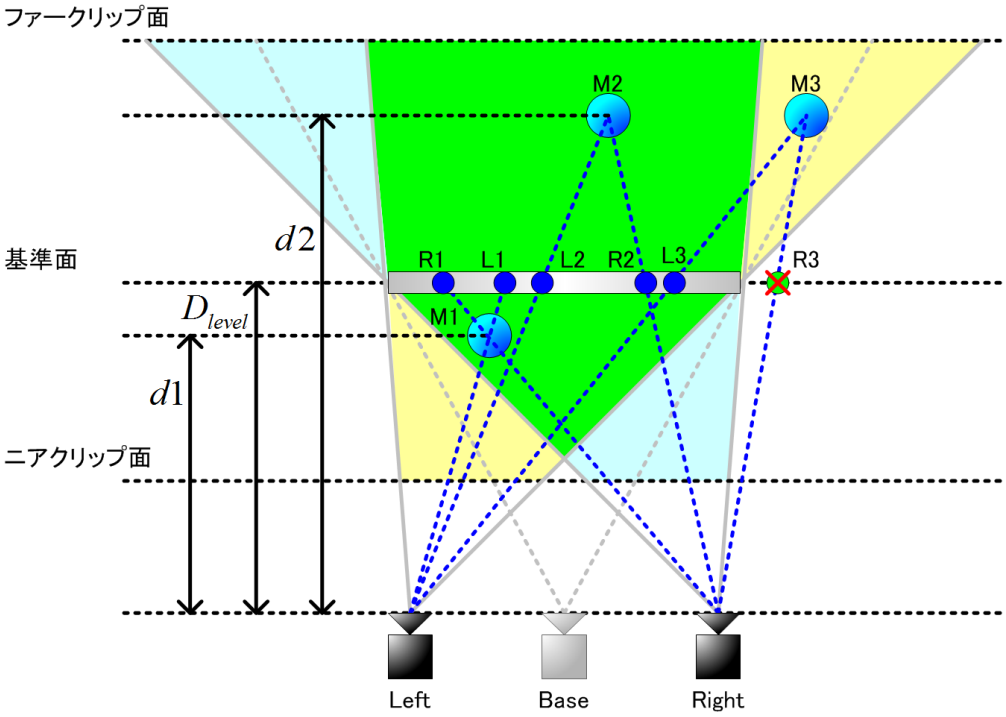
6.1.5. オブジェクトを任意の位置に表示するために必要な視差

ULCD ライブラリで算出した行列を利用すれば、透視射影で描画した 3D オブジェクトに関しては視差をどのくらいつければよいかを意識せずに立体視表示を行うことができます。しかし、2D イメージあるいは正射影で描画した 3D オブジェクトを

表示したい場合は、左目用と右目用の絵をアプリケーションで計算して作らなければなりません。

ライブラリによって生成された左右のカメラを用いて、LCD よりも手前にオブジェクトを表示する場合、または奥に表示する場合につけるべき視差を下図に示します。

図 6-5. 任意の位置にオブジェクトを表示するために必要な視差



カメラから $d1$ 離れたオブジェクト M1 を LCD より手前に表示するためには、カメラと M1 の位置を結ぶ直線と基準面の交点（右目は R1、左目は L1 の位置）にそれぞれ M1 を描画します。つまり R1L1 の視差が必要ということになります。この長さは、左右のカメラ間距離 I とカメラから基準面までの距離 D_{level} を用いると、三角比で計算することができます。

$$R1L1 = I \times (D_{level} - d1) / d1 \quad \text{ただし、} D_{level} \geq d1$$

同様に、LCD よりも奥に位置するオブジェクト M2（カメラからの距離 $d2$）は R2L2 の視差をつけることで表現することができます。

$$R2L2 = I \times (d2 - D_{level}) / d2 \quad \text{ただし、} D_{level} < d2$$

オブジェクト M3 のように、左右カメラどちらのビューボリュームにも含まれないような位置では基準面上に像を結ばないため、立体視表示に必要な視差を実現することができません。

6.1.6. 無限遠におけるオブジェクトのずれ

2D イメージあるいは正射影で描画したオブジェクトを無限遠（最奥）に位置するように見せるために、左目用の絵と右目用の絵に表示するオブジェクトをどのくらいずらして描画するべきかについて説明します。

無限遠に位置するオブジェクトは、左右の絵を限界視差 P_{r_ltd}（基準面となる LCD 上での実寸値）ずらして描画することで実現することができます。つまり、立体視表示が有効な状態で遠景を表現するためには、左目用と右目用のレンダーバッファへ元の位置からそれぞれ限界視差の半分だけずらして描画してください。ただし、実際に描画する際には P_{r_ltd} を仮想空間における限界視差 P_{v_ltd} に変換する必要があります。

原理上は手前方向に最も飛び出るようなオブジェクトの立体表示を実現することもできますが、図 6-5 で示すように、表現可能な範囲が奥方向の範囲と比べて狭いことに注意してください。

6.2. アプリケーションでの実装方法

アプリケーションで立体視表示を実現するためには、以下のような実装手順が必要となります。

- 上画面の表示モードを立体視表示のモードに設定する。
- 右目用の画像を表示するためのディスプレイバッファを用意する。
- 透視射影ならば、ULCD ライブラリを使用して左目用と右目用のカメラ行列を算出する。
- 左目用と右目用の画像をレンダリングし、それぞれ対応するディスプレイバッファに転送する。
- ディスプレイバッファの内容を LCD に表示する。

6.2.1. 表示モードの設定

nngxSetDisplayMode() を呼び出して表示モードの設定を行うことで、上画面での立体視表示の有効・無効を制御することができます。表示中に状態を切り替えるときは VSync 直後に行ってください。

コード 6-1. 表示モードの設定

```
void nngxSetDisplayMode(GLenum mode);
```

mode に NN_GX_DISPLAYMODE_STEREO を渡すと立体視表示が有効になります。立体視表示を無効にする場合は NN_GX_DISPLAYMODE_NORMAL を渡してください。それ以外を渡した場合は GL_ERROR_9003_DMP のエラーが生成されます。デフォルトは NN_GX_DISPLAYMODE_NORMAL を指定した状態（立体視表示が無効）です。

立体視表示が有効になると、上画面には左目用と右目用の 2 つの画面（LCD）が存在することになります。2 つの画面の解像度、フォーマット、領域を確保するメモリ（メインメモリ、VRAM-A/B）は同じでなければなりません。異なる場合は nngxSwapBuffers() の呼び出しでエラーが生成されます。下画面への影響はありません。

左目用の画面を指定する場合は通常時と同じ NN_GX_DISPLAY0 を使用します。右目用の画面の指定には、追加された NN_GX_DISPLAY0_EXT を使用します。右目用の画面でも、ほかの画面と同じように画面の指定（nngxActiveDisplay()）、ディスプレイバッファのバインド（nngxBindDisplayBuffer()）、LCD へ出力する際のオフセット（nngxDisplayEnv()）など、一連の処理を行わなければなりません。

左目用と右目用、どちらの画面を指定するのかを分かりやすくするために、別名として NN_GX_DISPLAY0 に対して NN_GX_DISPLAY0_LEFT が、NN_GX_DISPLAY0_EXT に対して NN_GX_DISPLAY0_RIGHT が定義されています。

6.2.2. ディスプレイバッファの確保

立体視表示が有効になると、上画面の LCD では左右の目で異なる映像が見えるようになります。アプリケーションでは、左目で見る映像を映し出している部分と右目で見る部分とを別の LCD であるかのように扱います。そのため、関数の呼び出しで上画面を指定するときに使用されていた NN_GX_DISPLAY0 は左目用の LCD の指定に使われ、右目用の LCD を指定するためには追加された NN_GX_DISPLAY0_EXT を使います。

別々の LCD として扱われますので、左目用と右目用それぞれにディスプレイバッファが必要になります。マルチバッファリングを行う場合、必要なディスプレイバッファの数もそれだけ増えることに注意してください。

以下のコード例では、上画面の LCD サイズのディスプレイバッファを右目用にダブルバッファリングで確保しています。

コード 6-2. 右目用のディスプレイバッファの確保

```
GLuint m_Display0BuffersExt[2];
// For right eye - Upper (DISPLAY0_EXT)
nngxActiveDisplay(NN_GX_DISPLAY0_EXT);
nngxGenDisplaybuffers(2, m_Display0BuffersExt);
nngxBindDisplaybuffer(m_Display0BuffersExt[0]);
nngxDisplaybufferStorage(GL_RGB8_OES,
    NN_GX_DISPLAY0_WIDTH, NN_GX_DISPLAY0_HEIGHT, NN_GX_MEM_FCRAM);
nngxBindDisplaybuffer(m_Display0BuffersExt[1]);
nngxDisplaybufferStorage(GL_RGB8_OES,
    NN_GX_DISPLAY0_WIDTH, NN_GX_DISPLAY0_HEIGHT, NN_GX_MEM_FCRAM);
nngxDisplayEnv(0, 0);
```

6.2.3. ULCD ライブラリの使用方法

レンダリングで透視射影を使用している場合、ベースとなるカメラ情報から立体視表示で使用することのできる左右のカメラ情報を算出する ULCD ライブラリを利用することで、アプリケーションは立体視に必要な視差を意識することなく立体視表示を実現することができます。

ULCD ライブラリでは、視差を考慮したカメラ行列を作成するクラス `nn::ulcd::StereoCamera` を用意しています。このクラスをアプリケーションで使用するには、ヘッダファイル `"nn/ulcd.h"` をインクルードしなければなりません。また、3D ボリュームの値を取得する関数は立体視表示に副次的な視覚効果を与えることを想定したものです。そのため、このクラスを使用しなければ 3D ボリュームを立体視表示の強度調整に使うことができません。

> **補足:** 3D ボリュームの値を取得する関数を使用する場合は事前連絡が必要です。

6.2.3.1. 初期化処理

`nn::ulcd::StereoCamera` クラスのインスタンスを生成し、メンバの `Initialize()` を呼び出すことで初期化を行うことができます。

コード 6-3. 初期化処理

```
void Initialize(void);
```

初期化処理では内部で保持している情報の初期化が行われます。

6.2.3.2. 限界視差の設定と取得

限界視差の設定および取得をメンバ関数の `SetLimitParallax()` と `GetLimitParallax()` で行うことができます。

コード 6-4. 限界視差の設定と取得

```
void SetLimitParallax(const f32 limit);
f32  GetLimitParallax(void) const;
```

limit には設定する限界視差を mm(ミリメートル)単位で指定します。限界視差の上限はガイドラインで定められており、奥方向の上限までの正値ならば自由に指定することができます。設定された限界視差はカメラ行列の計算結果に反映されます。

`GetLimitParallax()` で、限界視差の現在の設定値を取得することができます。一度も `SetLimitParallax()` で限界視差を設定していない場合は、ガイドラインで定められている上限値(奥方向)を返します。

6.2.3.3. ベースカメラの情報

ベースとなるカメラの情報をメンバ関数の SetBaseFrustum() と SetBaseCamera() で設定します。

コード 6-5. ベースカメラ情報の設定

```
void SetBaseFrustum(const nn::math::Matrix44 *proj);
void SetBaseFrustum(const f32 left, const f32 right, const f32 bottom,
                    const f32 top, const f32 near, const f32 far);
void SetBaseCamera(const nn::math::Matrix34 *view);
void SetBaseCamera(const nn::math::Vector3 *position,
                   const nn::math::Vector3 *rightDir,
                   const nn::math::Vector3 *upDir,
                   const nn::math::Vector3 *targetDir);
```

SetBaseFrustum() は主に、ニアとファーの両クリップ面の情報を設定します。それぞれのパラメータを個別に設定する関数と、nn::math::MTX44Frustum() や nn::math::MTX44Perspective() で作成したプロジェクション行列からパラメータを逆算する関数の 2 種類が用意されています。nn::math::Matrix44 構造体でプロジェクション行列を指定する場合は、3DS のビューボリュームの定義(Z 座標が 0 から -Wc でクリップされる)に基づいて算出されたものでなければ正常に計算が行われません。

SetBaseCamera() は主に、ベースカメラの位置情報などを設定します。それぞれのパラメータを個別に設定する関数と、nn::math::MTX34LookAt() などで作成したビュー行列からパラメータを逆算する関数の 2 種類が用意されています。ビュー行列またはベクトルには、3DS の座標系である右手座標系に基づいたものを指定してください。

6.2.3.4. 左右のカメラの行列計算

「6.1. 立体視表示の原理」で紹介したように左右それぞれのカメラ行列を計算する方法として、ベースカメラの設定を極力維持する手法(アプリケーション優先の算出方法)とベースカメラの設定を自動的に変更する手法(現実感優先の算出方法)の 2 つの計算方法を用意しています。

メンバ関数の CalculateMatrices() がアプリケーション優先の算出方法で、CalculateMatricesReal() が現実感優先の算出方法です。

コード 6-6. 左右のカメラ行列の計算

```
void CalculateMatrices(
        nn::math::Matrix44* projL, nn::math::Matrix34* viewL,
        nn::math::Matrix44* projR, nn::math::Matrix34* viewR,
        const f32 depthLevel, const f32 factor,
        const nn::math::PivotDirection pivot = nn::math::PIVOT_UPSIDE_TO_TOP,
        const bool update3DVolume = true);
void CalculateMatricesReal(
        nn::math::Matrix44* projL, nn::math::Matrix34* viewL,
        nn::math::Matrix44* projR, nn::math::Matrix34* viewR,
        const f32 depthLevel, const f32 factor,
        const nn::math::PivotDirection pivot = nn::math::PIVOT_UPSIDE_TO_TOP,
        const bool update3DVolume = true);
```

どちらの関数も引数は同じで、計算結果だけが異なります。

projL と *viewL* には左目用の、*projR* と *viewR* には右目用のプロジェクション行列とビュー行列の格納場所を指定します。値が書き込まれますので、それぞれの構造体には実体が必要です。

depthLevel には基準面までのカメラからの距離 D_{level} を、*factor* には立体具合の調整係数 D_r を指定します。

factor は、計算結果を補正するために使用されます。0.0 を渡すと視差がなくなり、1.0 で補正なしの状態になります。この値以外にも、3D ボリュームの入力値が計算結果に影響を与えます。

pivot で指定された方向にカメラの上方向が向くように、出力されるプロジェクション行列の出力に対して回転行列が乗算されます。デフォルトは nn::math::PIVOT_UPSIDE_TO_TOP で、カメラの上方向が LCD の上方向（下画面が配置されていない側の長辺の方向）に向くような回転行列が乗算されます。ベースカメラの設定で回転が考慮されているなど、回転処理が不要な場合は nn::math::PIVOT_NONE を指定してください。

update3DVolume は行列計算時点の 3D ボリュームの値を取得して利用するかどうかを指定します。引数を指定しなければ、利用する（デフォルト:true）になります。 行列計算時点の 3D ボリュームの値を利用する場合、1 フレーム内に何度も上記関数を用いて行列を計算する実装では 3D ボリュームの値が途中で変化し、描画結果に影響を与える可能性があります。

そのような実装では *update3DVolume* を利用しない（false）に指定し、各フレームの冒頭でメンバ関数 Update3DVolume() を呼び出すようにしてください。呼び出さなければ、3D ボリュームの操作に描画結果が連動しません。

コード 6-7. 3D ボリューム値を更新

```
void Update3DVolume(void);
```

6.2.3.5. 視差情報の取得

Initialize() で初期化された場合を除き、最後に行列計算を行った結果に基づいて、カメラから指定した距離に像を結ぶために必要な視差を取得するためのメンバ関数が用意されています。

コード 6-8. 視差情報の取得

```
f32 GetParallax(const f32 distance) const;
f32 GetMaxParallax(void) const;
```

GetParallax() はカメラから *distance* 離れた位置、GetMaxParallax() は無限遠にあるオブジェクトを描画するために必要な視差を画面幅に対する割合（1.0 が 100 %）で返します。返り値に LCD の解像度を乗算することで、（ベースカメラでの表示位置から）左右それぞれに何ピクセルずらして描画すればよいかを得ることができます。

distance で指定した位置が基準面より手前ならば負の値が返され、基準面より奥ならば正の値が返されます。
distance に負の値を渡した場合は 0 が返されます。

透視射影で 3D オブジェクトを描画する場合はライブラリによって生成された行列を利用することで視差を意識する必要はありませんでしたが、2D のオブジェクトを描画する場合や正射影で描画する場合はオブジェクトごとに視差計算を行う必要があります。しかし、描画するオブジェクトごとに GetParallax() で視差を計算すると、CPU の処理負荷が高くなってしまいます。そこで、以下の関数で視差計算の途中の値を取得し、残りの計算を頂点シェーダで行わせることで高速化することができます。

コード 6-9. 視差計算の途中経過の取得

```
f32 GetCoefficientForParallax(void) const;
```

この関数で返される値に ((*distance*-*depthLevel*)/*distance*) を乗算することで、GetParallax() が返す値と同じ値を求めることができます。

視差情報と同じように、計算結果に基づいて、ベースカメラから基準面、ニアクリップ面、ファークリップ面それぞれまでの距離を取得するためのメンバ関数が用意されています。

コード 6-10. ベースカメラから基準面、ニアクリップ面、ファークリップ面までの距離の取得

```
f32 GetDistanceToLevel(void) const;
f32 GetDistanceToNearClip(void) const;
f32 GetDistanceToFarClip(void) const;
```

これらの関数は、主に現実感優先の算出方法で変更されたベースカメラの情報を得るために使用します。アプリケーション優先の算出方法で計算した場合、これらの関数は計算のために渡した情報そのままを返します。

6.2.3.6. 終了

nn::ulcd::StereoCamera クラスのインスタンスが不要になった場合は、メンバ関数の Finalize() を呼び出してください。

コード 6-11. 終了処理

```
void Finalize(void);
```

この関数はデストラクタで呼び出されますが、必ずアプリケーションからも明示的に呼び出してください。

6.2.4. レンダリングとディスプレイバッファへの転送

透視射影で 3D オブジェクトの描画を行っている場合は、ULCD ライブラリで算出した左右のカメラ用のプロジェクション行列とビュー行列を使用することで、視差を意識せずにレンダリングすることができます。しかし、2D オブジェクトや正射影で 3D オブジェクトを描画する場合は、アプリケーションで視差を考慮しながら左目用と右目用の画像をレンダリングしなければなりません。2D オブジェクトも透視射影で描画すれば視差を意識せずにレンダリングすることができますが、オブジェクト配置の前後関係には配慮しなければなりません。

カラーバッファにレンダリングされた画像を LCD に表示するためにディスプレイバッファへ転送しますが、左目用と右目用で異なるディスプレイバッファに転送しなければならないことに注意してください。

6.2.5. LCD への表示

左目用と右目用のディスプレイバッファを、それぞれ NN_GX_DISPLAY0 と NN_GX_DISPLAY0_EXT で指定する画面に関連付けます。それからディスプレイバッファのスワップを行いますが、立体視表示時の右目用ディスプレイバッファは上画面(NN_GX_DISPLAY0)が指定されたときに同時に行われるため、nngxSwapBuffers() に NN_GX_DISPLAY0_EXT を含めて渡す必要はありません。

コード 6-12. ディスプレイバッファの関連付けとバッファスワップ

```
// UpperLCD for Left Eye
nngxActiveDisplay(NN_GX_DISPLAY0);
nngxBindDisplaybuffer(m_Display0Buffers[m_CurrentDispBuf0]);
// UpperLCD for Right Eye
nngxActiveDisplay(NN_GX_DISPLAY0_EXT);
nngxBindDisplaybuffer(m_Display0BufferExt[m_CurrentDispBuf0Ext]);
// Swap buffers
nngxSwapBuffers(NN_GX_DISPLAY0);
```

6.2.6. 3D ボリュームの入力について

最大にした状態が立体視に特に支障を感じない方に対する「お薦め位置」となるように調整してください。

3D ボリュームを下げて立体視表示(3D 表示)から通常表示(2D 表示)に切り替わったときは、システム側で自動的に LCD

のシャッターをオフにします。また、2D 表示と 3D 表示の切り替えや液晶バックライトの輝度についてもシステムが自動的に制御を行いますので、デバッグ用途を除いてはアプリケーションから制御を行う必要はありません。

6.2.7. 立体視表示の無効化について

本体設定の保護者による使用制限で 3D 映像の表示を制限している場合や 3D ボリュームを 0(最低値)に下げている場合は、強制的に立体視表示が無効となり、表示モードの設定や右目用の LCD への表示は無視され、必ず通常表示(2D 表示)となります。

立体視表示が許可されている状態であるかどうかは、`nn::gx::IsStereoVisionAllowed()` で確認することができます。この関数は立体視表示が許可されている状態のときに `true` を返し、強制的に立体視表示が無効化される状態のときに `false` を返します。この判定に表示モードの設定は影響しません。

6.2.8. 左右を反転する場合

立体視表示で左右反転の表示をする場合、プロジェクション行列の X 軸を反転させるだけでは正しく表示されず、奥行きの位置が基準面を境に反転してしまいます。

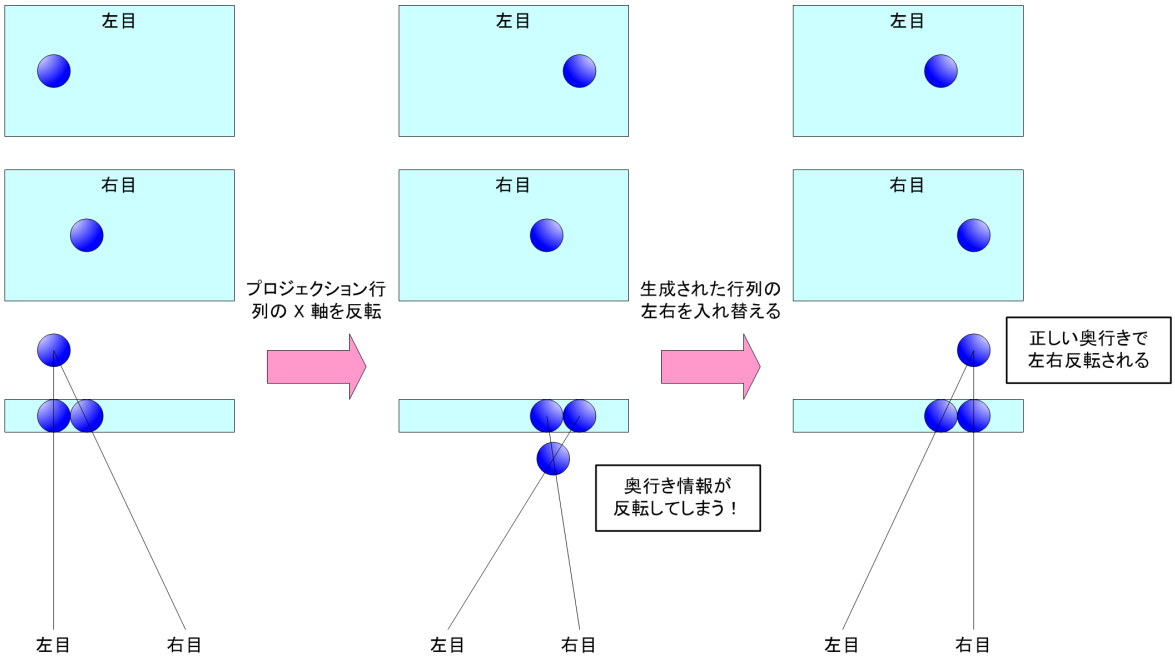
この問題を解消するには、X 軸を反転させたプロジェクション行列で立体視表示のプロジェクション行列とモデルビュー行列を生成し、その左右を入れ替えて表示します。

コード 6-13. 立体視表示で左右を反転する場合

```
nn::math::Matrix44 proj,rev;
nn::math::Matrix44 projL, projR;
nn::math::Matrix34 viewL, viewR;
// SetBaseFrustum
MTX44Frustum(&proj, l, r, b, t, n, f);
MTX44Identity(&rev);
rev.m[0][0] = -1.0f;
MTX44Mult(&proj, &proj, &rev);          // X軸で反転
s_StereoCamera.SetBaseFrustum(&proj);
// SetBaseCamera
nn::math::Matrix34 cam;
nn::math::Vector3 camUp(0.f, 1.f, 0.f);
nn::math::MTX34LookAt(&cam, &camPos, &camUp, &focus);
s_StereoCamera.SetBaseCamera(&cam);
// CalcurateMatrices
s_StereoCamera.CalculateMatrices(
    &projR, &viewR, &projL, &viewL,     // 左右の行列を逆に指定
    depthLevel, factor, nn::math::PIVOT_UPSIDE_TO_TOP);
```

下図は、立体視表示での左右反転の原理を示したものです。

図 6-6. 立体視表示での左右反転の原理



6.3. 予約フラグメントシェーダを使用する場合

シャドウを使う場合、シャドウテクスチャはライトからの深度情報ですので、シャドウテクスチャの生成時には立体視表示によるビュー行列の変化は関係しません。オブジェクトのレンダリング時にはビュー行列が関係するため、右目用と左目用に 2 回のレンダリングが必要となります。その際、シャドウテクスチャを適用するためのテクスチャ座標の変換行列には注意が必要です。

ガスレンダリングを使う場合、第 1 パスでデプス情報を生成する時点でビュー行列が関わりますので、立体視表示をするにはすべてのパスを左右別々に実行しなければなりません。

フォグを使う場合、参照テーブルへの入力値はウィンドウ座標系でのデプス値ですので、立体視表示でプロジェクション行列が変更になると参照テーブルを再作成しなければなりません。アプリケーション優先の算出方法であれば、立体視表示でもベースカメラのプロジェクション行列で参照テーブルを作成することができます。しかし、現実感優先の算出方法ではニアクリップ面とファークリップ面がベースカメラの値から変化するため、3D ボリュームで調整が行われるだけでも参照テーブルを再作成しなければなりません。

フラグメントライティング全般では、どのように座標系を統一するかにもよりますが、ライト位置がビュー行列の影響を受ける場合、左右それぞれのビュー行列を考慮したライト位置でレンダリングしなければなりません。

6.4. ステレオカメラとの連動

左右のカメラは 2 つあるポートにそれぞれ接続されていますが、ポート 1 が右目用(NN_GX_DISPLAY0_EXT)で、ポート 2 が左目用(NN_GX_DISPLAY0)の画像をキャプチャしていることに注意してください。また、カメラは 2 つのポートからそれぞれのバッファへと画像データを出力しますが、YUVtoRGB 回路は 1 つですので、YUV フォーマットで取得したカメラからのキャプチャ画像を RGB フォーマットに変換するときにはアプリケーションで排他処理をするなどの工夫が必要です。また、左右の画像が同じフレームで取得されたものでなければ、うまく立体的に表示されない場合があります。

左右のカメラで得られた画像をそのまま表示することでも立体的に表示されますが、カメラの間隔と両目の間隔が同一ではないことに注意しなければなりません。

6.4.1. ステレオカメラのキャリブレーション

外側にある 2 つのカメラは水平かつ 35 mm 離れて配置されるように設計されていますが、製造段階での取り付け精度によって左右のカメラで多少の誤差が発生し、左右のカメラ画像にずれが生じます。取り付け誤差による画像のずれは、カメラライブラリ内では自動的に補正されませんので、アプリケーション側で補正しなければなりません。

> **補足:** 完全には補正できず、補正後の画像の周辺部に多少のずれが残る場合があります。

ステレオカメラのキャリブレーションデータは nn::camera::GetStereoCameraCalibrationData() で取得することができます。キャリブレーションデータは nn::camera::StereoCameraCalibrationData 構造体に格納されます。

コード 6-14. ステレオカメラのキャリブレーションデータの取得

```
void nn::camera::GetStereoCameraCalibrationData(
        nn::camera::StereoCameraCalibrationData * pDst);
```

基本的にキャリブレーションデータの各メンバ変数は、左のカメラ画像を右のカメラ画像に合わせるために必要な補正(拡大縮小、光軸の回転、並進移動)と補正値の測定条件を示しています。

表 6-1. キャリブレーションデータが取り得る値の範囲

項目	メンバ	取り得る値の範囲
拡大縮小	scale	0.9604 ～ 1.0405
Z 軸(光軸)の回転角度	rotationZ	-1.6 ～ +1.6(単位:度)
X(水平)方向の並進量	translationX	-154 ～ -19(単位:ピクセル)
Y(垂直)方向の並進量	translationY	-70 ～ +70(単位:ピクセル)

2 つの並進量が取り得る値の範囲の幅が広いため、そのまま左のカメラの画像を移動させて補正すると画像の端が表示画面内に入ってしまい、表示画面の端が欠ける可能性があります。キャリブレーションデータを立体視表示に利用する場合は、CAMERA ライブラリで用意されている補正行列の計算のための関数を使用してください。取り付け誤差が限度値であっても、画像の端が欠ける場合には拡大表示によって表示の画像サイズとなるように対応した関数が用意されています。立体視表示以外の処理(画像認識処理など)に利用する場合は、キャリブレーションデータの値によって処理の精度が落ちるなどの不具合が起こらないように注意してください。

以下の関数を利用して、左のカメラ画像に乗算しなければならない補正行列をキャリブレーションデータから計算することができます。

コード 6-15. 補正行列の計算

```
void nn::camera::GetStereoCameraCalibrationMatrix(
        nn::math::MTX34 * pDst,
        const nn::camera::StereoCameraCalibrationData & cal,
        const f32 translationUnit, const bool isIncludeParallax = true);
void nn::camera::GetStereoCameraCalibrationMatrixEx(
        nn::math::MTX34 * pDstR, nn::math::MTX34 * pDstL, f32 * pDstScale,
        const nn::camera::StereoCameraCalibrationData & cal,
        const f32 translationUnit, const f32 parallax,
```

```
        const s16 orgWidth, const s16 orgHeight,
        const s16 dstWidth, const s16 dstHeight);
f32 nn::camera::GetParallax(
        const nn::camera::StereoCameraCalibrationData & cal, f32 distance);
```

nn::camera::GetStereoCameraCalibrationMatrix() では、*cal* にキャリブレーションデータ、*translationUnit* に 3D 空間上での並進移動の単位量(補足を参照)を渡すことで、*pDst* に補正行列を取得することができます。*isIncludeParallax* に true を渡すと、補正行列に測定チャート上の視差が含まれ、カメラから 250 mm 離れた対象に焦点が合うようになります。取り付け誤差だけを補正する場合には *isIncludeParallax* に false を渡してください。

> **補足:** *translationUnit* には、並進移動量(カメラ画像を 1 ピクセル移動させるのに必要な 3D 空間上での移動量)に VGA 画像と表示画像の大きさ(幅)の比をかけた値を指定します。ただし、これは VGA 画像と異なるサイズでそのまま表示する場合や補正行列をかけてから画像を拡大する場合の指定方法です。補正行列をかける前に画像が貼られるオブジェクトを拡大し、VGA 画像に pixel by pixel で表示された被写体と同じ大きさに画像内の被写体が表示される場合は並進移動量そのままを指定してください。

nn::camera::GetStereoCameraCalibrationMatrixEx() では、取り付け誤差が限度値になる本体で画面端が立体視できない問題に対応した補正行列を計算します。また、nn::camera::GetParallax() で取得する視差を利用していますので、計算される補正行列は特定の距離に焦点を合わせたものになります。以上のことから、特別な理由がない限り、補正行列の計算にはこちらの関数を使用することを推奨します。

引数 *cal* と *translationUnit* に渡す値は前述の関数と同じです。*parallax* に渡す VGA 画像における視差(ピクセル単位)の計算には GetParallax() を使用します。視差は引数 *distance* で指定された距離(メートル単位)に焦点が合うように計算されます。*orgWidth* と *orgHeight* にはカメラ画像の(トリミング後の)幅と高さをピクセル単位で指定します。*dstWidth* と *dstHeight* にはレンダリングに必要な幅と高さをピクセル単位で指定します。

補正行列は *pDstR* と *pDstL* に左右のカメラそれぞれに乗算する行列が返され、*pDstScale* にはレンダリングのサイズにするために必要な拡大縮小率が返されます。

> **補足:** 補正行列の計算の詳細については関数リファレンスを参照してください。

nn::camera::GetParallaxOnChart() は、測定チャート上での視差(カメラから 250 mm 離れた対象に焦点が合うような視差)をピクセル単位で取得することができます。

コード 6-16. 測定チャート上での視差の取得

```
f32 nn::camera::GetParallaxOnChart(
        const nn::camera::StereoCameraCalibrationData & cal);
```

取り付け誤差による水平方向のずれは、キャリブレーションデータの translationX メンバからこの関数で取得した値を減算したものと同じです。

6.4.2. ステレオカメラの明るさを連動させる

ステレオカメラでは、自動露出などの内部処理が左右のカメラで独立しているため、被写体によっては画像の明るさが左右で異なる場合があります。これに対応するため、CAMERA ライブラリにはステレオカメラの画像の明るさを自動的に連動させる nn::camera::SetBrightnessSynchronization() が用意されています。

コード 6-17. ステレオカメラの明るさを連動させる

```
nn::Result nn::camera::SetBrightnessSynchronization(bool enable);
```

enable に true を指定することで連動する設定になります。デフォルトの設定は無効(false)です。また、ステレオカメラ以外の組み合わせでは機能しません。

この関数は外側カメラ(R)と外側カメラ(L)を起動させていない状態でも呼び出すことができます。連動を有効にした場合は、そのあとでカメラをスタンバイ状態にしても設定は記憶されたままとなり、再度ステレオカメラを起動させたときには連動状態が継続します。

この機能は、定期的に外側カメラ(R)の露出に関する設定を外側カメラ(L)に書き込むことで、ステレオカメラの画像の明るさを連動させます。特に自動露出が有効であるときは、外側カメラ(R)の露出量の変化に追従して外側カメラ(L)の露出量が変化します。設定の連動はライブラリが作成した低優先度のスレッド(アプリケーションのリソースを消費しません)で行なわれるため、処理が重いときに連動が遅れることがあります。

カメラの設定が以下のいずれかに該当すると、この関数の実行に失敗します。また、連動を有効にしたあとは、以下のいずれかの設定を行うと実行に失敗します。

- ホワイトバランス設定が WHITE_BALANCE_NORMAL 以外
- オートホワイトバランス設定が無効
- 撮影モードが PHOTO_MODE_LANDSCAPE
- コントラスト設定が CONTRAST_PATTERN_10

注意: カメラの再起動処理中に呼び出したときは、ライブラリ内で処理が長時間ブロックされることがありますので注意してください。

6.5. 注意点など

補足: この節の内容は、別途検討中のガイドラインに従って変更・追加される可能性があります。

6.5.1. 手前へのオブジェクト配置

立体視表現を用いて LCD 表面よりも手前に飛び出すようにオブジェクトを配置する場合は、飛び出したオブジェクトが LCD の縁にかからないように配置してください。(見かけ上)奥にあるはずの画面の縁によって、手前にあるはずのものが隠されるといった不自然な状態になってしまい、プレイヤーが正しい立体感を得ることができません。

また、ニアクリップ面をどこに置くかには注意が必要です。ニアクリップ面はカメラにごく近い位置に置くことが一般的ですが、その場合ニアクリップ面への距離はプレイヤーに正しい立体感を与える手前方向の限界距離とは一致しません。そのため、オブジェクトがカメラに近寄り過ぎると、立体視ができなくなります。

しかし、単純にニアクリップ面をカメラから遠ざければよいという問題ではないため、それぞれのアプリケーションにおいてオブジェクトがカメラに近寄り過ぎないように配慮する必要があります。

6.5.2. 2D オブジェクトとの混合配置

2D のオブジェクトを 3D オブジェクトと同時に表示するために、正射影と透視射影のように異なるカメラ設定でレンダリングした結果を合成する場合は、2D と 3D のオブジェクトが正しい前後関係となるように 2D オブジェクトの描画位置やサイズを調

整する必要があります。

2D オブジェクトのレンダリングに対しても透視射影によってレンダリングすれば、2D オブジェクト自身の調整を意識する必要はありませんが、3D オブジェクトと 2D オブジェクトの配置の前後関係に配慮が必要な点は変わりません。3D オブジェクトによって 2D オブジェクトが隠れてしまうことが起こりえます。

6.5.3. 本体を傾けたときの対処

立体視表示を行っているときに、本体を手前や奥の方向への傾けてもあまり問題にはなりませんが、本体を左右に傾けたり回転させたりすると、LCD 面がユーザーと正対していないために立体視が困難になる可能性があります。

この問題への対処として、本体に内蔵されている「顔シューティング」ではジャイロセンサーで本体が傾いているかどうかを判断し、本体が傾いていることを検出したときに立体具合の調整係数が弱くなる（立体視の強度を下げる）ように実装されています。動きの激しいアクションゲームなどで立体視表示を行う場合は、ジャイロセンサーで本体の傾きを検知して立体視の強度を調整することを推奨します。

6.5.4. 表示モードで立体視表示を無効にしたときの注意点

アプリケーションが `nngxSetDisplayMode(NN_GX_DISPLAYMODE_NORMAL)` で立体視表示を無効にしていても、`nn::ulcd::CalculateMatrices()` は 3D ボリュームに連動した値を返します。そのため、ULCD ライブラリで計算された行列をそのまま表示に使用していると、3D ランプが消えている状態でも、上画面の表示が 3D ボリュームに連動して変化してしまいます。

`nngxSetDisplayMode()` で立体視表示を無効にしているときは `CalculateMatrices()` を使用しないか、*factor* に 0.0 を渡してください。

この現象は「6.2.7. 立体視表示の無効化について」のように強制的に立体視表示が無効化されるときには起こりません。

7. ETC1 圧縮テクスチャのフォーマット解説

ETC(Ericsson Texture Compression)フォーマットで圧縮された ETC1 圧縮テクスチャは、ハードウェアでの展開に対応していますので、通常のテクスチャに比べて高速に処理することができます。さらに、24 bit カラー画像(RGB8)を 4x4 テクセルのブロックごとにわずか 8 バイト(64 bit)に圧縮することができるため、テクスチャで使用するメモリの削減も期待できます。しかし、ブロック内で使われている色が似たような色であることを利用した圧縮方法のため、風景や人物をカメラで撮影した画像の圧縮には向いていますが、赤と青のように RGB 各成分の値が極端に違う色が同じブロック内に存在する画像の圧縮には向いていません。

3DS 独自の仕様として、テクセルごとに 4 bit のアルファ値を付加したフォーマット(合計 128 bit)に対応しています。

この章では、ETC1 圧縮テクスチャのフォーマットを解説し、3DS で使用するために必要なフォーマットの変換についても説明します。

7.1. ETC フォーマット

ETC フォーマットは、4x4 テクセルを 4x2 または 2x4 テクセルで分割したサブブロック単位で、基準となる RGB 値(ベースカラー)と差分を表すテーブルを持っています。フォーマットには個別モード(Individual mode)と差分モード(Differential mode)の 2 種類が存在しますが、その違いは前半の 32 bit に含まれているベースカラーのフォーマットだけです。どちらのフォーマットも後半の 32 bit に違いはなく、ベースカラーの決定以外で展開のアルゴリズムにも違いはありません。

以下に ETC1 圧縮テクスチャのビットレイアウトを示します。データの並びとしては、アルファチャンネルがカラーチャンネルの前に配置されます。

図 7-1. ETC1 圧縮テクスチャのビットレイアウト

Color channel

Individual mode (Diffbit = 0)

63–60	59–56	55–52	51–48	47–44	43–40	39–37	36–34	33	32
BaseColor1 R1	BaseColor2 R2	BaseColor1 G1	BaseColor2 G2	BaseColor1 B1	BaseColor2 B2	TableCW1	TableCW2	Diffbit	Flipbit

Differential mode (Diffbit = 1)

63–59	58–56	55–51	50–48	47–43	42–40	39–37	36–34	33	32
BaseColor1 R1'	DiffColor2 dR2	BaseColor1 G1'	Diffcolor2 dG2	BaseColor1 B1'	DiffColor2 dB2	TableCW1	TableCW2	Diffbit	Flipbit

In both cases (Diffbit = 0 or 1)

31–24	23–16	15–8	7–0
Most significant pixel index bits (MSB)		Least significant pixel index bits (LSB)	

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
p	o	n	m	l	k	j	i	h	g	f	e	d	c	b	a	p	o	n	m	l	k	j	i	h	g	f	e	d	c	b	a

Alpha channel

63–60	59–56	55–52	51–48	47–44	43–40	39–36	35–32	31–28	27–24	23–20	19–16	15–12	11–8	7–4	3–0
p	o	n	m	l	k	j	i	h	g	f	e	d	c	b	a

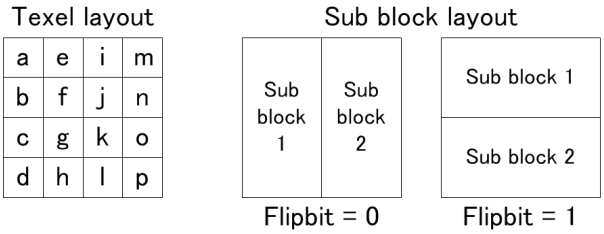
7.1.1. テクセルの配置とブロックの分割

後半の 32 bit はさらに MSB と LSB の 16 bit ずつの領域に分かれています。各領域の最下位ビットから最上位ビットの並びは、左上のテクセルから縦方向に右下のテクセルと対応しています。つまり、最下位ビット(bit 0, 16)が左上に配置された

テクセルに対応し、そこから下方向に上位のビットが対応していきます。下端に到達したら、右隣の列の上端から下端に向かって対応し、最上位ビット(bit 15, 31)は右下のテクセルに対応しています。

Flipbit(bit 32)の値によって、ブロックは縦(2x4)または横(4x2)に分割されます。分割された 8 テクセルをサブブロックと呼び、左または上にあるサブブロックをサブブロック 1、右または下にあるサブブロックをサブブロック 2 とします。ブロックの分割方法によってテクセルの配置順は変化しません。

図 7-2. テクセルとサブブロックの配置



7.1.2. ベースカラーの決定

ベースカラーはサブブロックごとに決定されますが、Diffbit(bit 33)の値によってフォーマットとベースカラーの決定方法に違いがあります。

Diffbit が 0 のときは個別モード(Individual mode)となり、サブブロックごとに R、G、B の各成分が 4 bit ずつ割り当てられています。サブブロック 1 が R1/G1/B1、サブブロック 2 が R2/G2/B2 です。ベースカラーは、この 4 bit を連結して 8 bit に拡張した値(17 倍)です。R1 が 1001b(9)のとき、サブブロック 1 のベースカラーの赤成分は 10011001b(153)となります。

Diffbit が 1 のときは差分モード(Differential mode)となり、サブブロック 1 には R、G、B の各成分が 5 bit ずつ割り当てられ、サブブロック 2 にはサブブロック 1 の 5 bit との差分値として各成分 3 bit が割り当てられています。サブブロック 1 が R1' / G1' / B1'、サブブロック 2 が R1' + dR2 / G1' + dG2 / B1' + dB2 です。サブブロック 1 のベースカラーは、5 bit に上位 3 bit を連結して 8 ビットに拡張した値(8.25倍)です。サブブロック 2 のベースカラーは、5 bit に 2 の補数表現で表された 3 bit の差分値(+3 ～ -4)を加算した結果をサブブロック 1 と同様に拡張した値です。R1' が 11001b(25)、dR2 が 100b(-4)のとき、サブブロック 1 のベースカラーの赤成分は 11001110b(206)、サブブロック 2 のベースカラーの赤成分は 11001b(25) - 100b(4) = 10101b(21)を拡張した値のため 10101101b(173)となります。差分値を加算した結果が 0 ～ 31 の範囲内に収まらない組み合わせは動作が不定となりますので、圧縮時にそのような組み合わせが発生しないようにしなければなりません。

表 7-1. 個別モードでのビット列と拡張後の値の対応

ビット列	値	ビット列	値	ビット列	値	ビット列	値
0000b	0	0100b	68	1000b	136	1100b	204
0001b	17	0101b	85	1001b	153	1101b	221
0010b	34	0110b	102	1010b	170	1110b	238
0011b	51	0111b	119	1011b	187	1111b	255

表 7-2. 差分モードでのビット列と拡張後の値および差分を考慮した値の対応

ビット列	値	100b (−4)	101b (−3)	110b (−2)	111b (−1)	000b (0)	001b (+1)	010b (+2)	011b (+3)
00000b	0	−	−	−	−	0	8	16	24
00001b	8	−	−	−	0	8	16	24	33
00010b	16	−	−	0	8	16	24	33	41
00011b	24	−	0	8	16	24	33	41	49
00100b	33	0	8	16	24	33	41	49	57
00101b	41	8	16	24	33	41	49	57	66
00110b	49	16	24	33	41	49	57	66	74
00111b	57	24	33	41	49	57	66	74	82
01000b	66	33	41	49	57	66	74	82	90
01001b	74	41	49	57	66	74	82	90	99
01010b	82	49	57	66	74	82	90	99	107
01011b	90	57	66	74	82	90	99	107	115
01100b	99	66	74	82	90	99	107	115	123
01101b	107	74	82	90	99	107	115	123	132
01110b	115	82	90	99	107	115	123	132	140
01111b	123	90	99	107	115	123	132	140	148
10000b	132	99	107	115	123	132	140	148	156
10001b	140	107	115	123	132	140	148	156	165
10010b	148	115	123	132	140	148	156	165	173
10011b	156	123	132	140	148	156	165	173	181
10100b	165	132	140	148	156	165	173	181	189
10101b	173	140	148	156	165	173	181	189	198
10110b	181	148	156	165	173	181	189	198	206
10111b	189	156	165	173	181	189	198	206	214
11000b	198	165	173	181	189	198	206	214	222
11001b	206	173	181	189	198	206	214	222	231
11010b	214	181	189	198	206	214	222	231	239
11011b	222	189	198	206	214	222	231	239	247
11100b	231	198	206	214	222	231	239	247	255
11101b	239	206	214	222	231	239	247	255	−
11110b	247	214	222	231	239	247	255	−	−
11111b	255	222	231	239	247	255	−	−	−

7.1.3. 差分テーブルとテクセルカラーの決定

決定したベースカラーに対して、差分テーブルから選択した差分値を加算した結果が最終的にテクセルカラーとなります。

差分テーブルはサブブロックごとに Table codeword(Table CW)で指定します。指定には、サブブロック 1 が Table CW1(bit 37 ～ 39)を、サブブロック 2 が Table CW2(bit 34 ～ 36)を使い、その値は 3 bit なので 8 種類から選択することができます。差分テーブルには 4 つの差分値が決められており、MSB と LSB の組み合わせでテクセルごとの差分値に使う値が決まります。選択された差分値はベースカラーの RGB 成分すべてに加算され、加算された結果は 0 ～ 255 の範囲にクランプされます。例えば、差分テーブルが 011b、MSB が 1、LSB が 1 のときは -42 がベースカラーの全成分に加算され、ベースカラーを(33, 198, 99)とするとテクセルカラーは(0, 156, 57)となります。

表 7-3. Table codeword、MSB、LSB と差分値の対応

Table codeword	MSB=1 LSB=1	MSB=1 LSB=0	MSB=0 LSB=0	MSB=0 LSB=1
000b	-8	-2	+2	+8
001b	-17	-5	+5	+17
010b	-29	-9	+9	+29
011b	-42	-13	+13	+42
100b	-60	-18	+18	+60
101b	-80	-24	+24	+80
110b	-106	-33	+33	+106
111b	-183	-47	+47	+183

7.2. PICA ネイティブフォーマットでの圧縮テクスチャ

3DS で ETC1 圧縮テクスチャを使用する場合、そのフォーマットは PICA ネイティブフォーマットでなければなりません。PICA ネイティブフォーマットは OpenGL ES の標準仕様とは、テクセルの参照開始座標やブロックの並び、バイトオーダーが異なります。

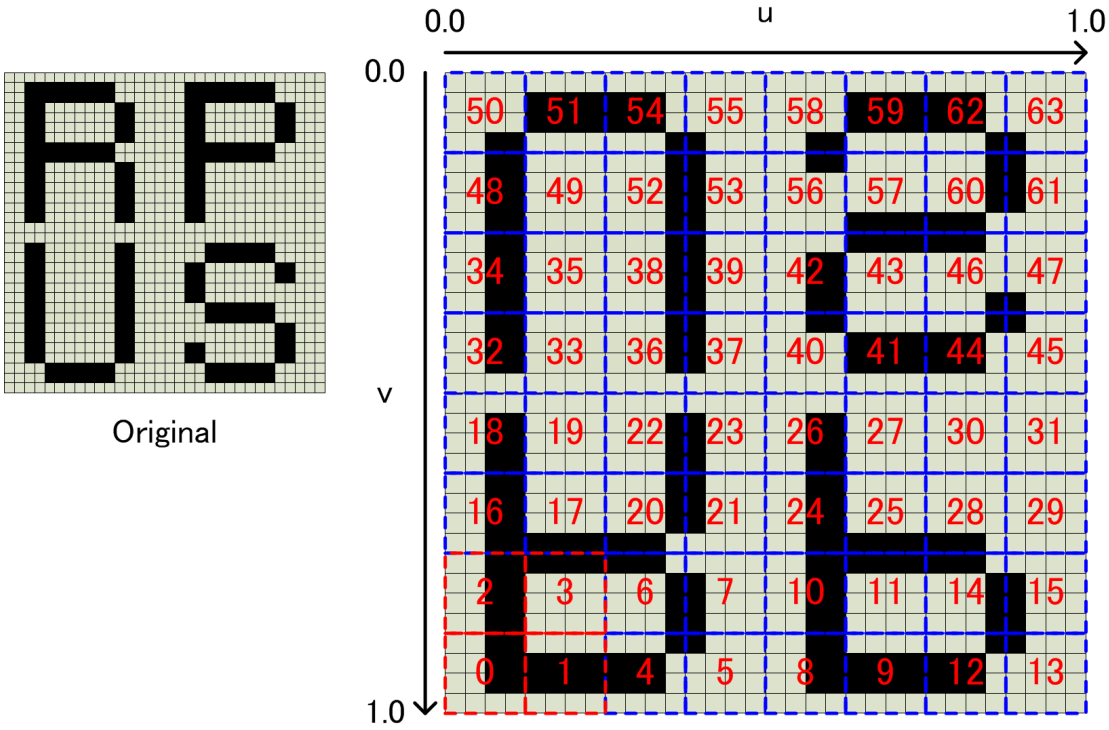
7.2.1. V フリップ

OpenGL ES ではテクセルの参照開始座標は(u, v)=(0.0, 0.0)ですが、3DS では(u, v)=(0.0, 1.0)となっています。そのため、テクスチャイメージを圧縮する前に v 座標方向のフリップ(V フリップ)を適用しなければなりません。

7.2.2. ブロックフォーマット

ETC1 圧縮テクスチャでは 4x4 テクセルのブロックを u、v 方向に 2 つずつ、8x8 テクセルをメタブロックとして定義しています。メタブロック内ではジグザグにブロックが配置され、メタブロックは u 方向に連続して配置されます。

図 7-3. ETC1 圧縮テクスチャのブロック参照順序



7.2.3. バイトオーダー

3DS では、エンディアンの関係で OpenGL ES とはデータの並び(バイトオーダー)が異なっています。ビットレイアウトを実際のバイトの並びで見てみると、以下のようなレイアウトとなっています。

図 7-4. 実際のバイトオーダーで見たビットレイアウト

Color channel

In both cases (Diffbit = 0 or 1)

+ 0 byte								+ 1 byte								+ 2 byte								+ 3 byte							
Least significant pixel index bits (LSB)																Most significant pixel index bits (MSB)															
h	g	f	e	d	c	b	a	p	o	n	m	l	k	j	i	h	g	f	e	d	c	b	a	p	o	n	m	l	k	j	i

Individual mode (Diffbit = 0)

+ 4 byte				+ 5 byte		+ 6 byte		+ 7 byte	
TableCW1	TableCW2	Diffbit	Flipbit	BaseColor1 B1	BaseColor2 B2	BaseColor1 G1	BaseColor2 G2	BaseColor1 R1	BaseColor2 R2

Differential mode (Diffbit = 1)

+ 4 byte				+ 5 byte		+ 6 byte		+ 7 byte	
TableCW1	TableCW2	Diffbit	Flipbit	BaseColor1 B1'	DiffColor2 dB2	BaseColor1 G1'	Diffcolor2 dG2	BaseColor1 R1'	DiffColor2 dR2

Alpha channel

+ 0 byte		+ 1 byte		+ 2 byte		+ 3 byte		+ 4 byte		+ 5 byte		+ 6 byte		+ 7 byte	
p	o	n	m	l	k	j	i	h	g	f	e	d	c	b	a

カラーチャンネルのデータは 8 バイト単位でバイトスワップが行われますが、アルファチャンネルのデータはバイトスワップが行われません。また、アルファチャンネルがカラーチャンネルの前に配置されることに変わりはありません。

8. コマンドキャッシュ

コマンドキャッシュとは、3D グラフィックスの描画で呼び出した関数がコマンドリストに蓄積した 3D コマンドをそのまま再使用する機能です。3D コマンドそのものを再利用しますので、本来ならば 3D コマンドを生成するために必要な関数呼び出しで発生するコストを削減し、CPU 処理の軽減を期待することができます。

コマンドキャッシュを利用してできることには以下のものがあります。

- コマンドリストの保存
- コマンドリストの再使用
- コマンドリストのコピー
- コマンドリストのエクスポート、インポート
- 3D コマンドの追加
- 3D コマンドの編集

この章では、コマンドキャッシュの利用方法を紹介し、3D コマンドの編集に必要な情報を提供します。

8.1. コマンドリストの保存

コマンドキャッシュは、コマンドリストに格納されている 3D コマンドバッファの内容と、キューイングされているコマンドリクエストを対象とする機能です。しかし、コマンドキャッシュは保存の開始と終了の宣言によって、コマンドリストを再使用するための情報を取得するだけですので、コマンドキャッシュとしてのオブジェクトは存在しません。保存の対象となったコマンドリストに蓄積された情報をそのまま利用するため、アプリケーションでコピーを作成していない限り、対象のコマンドリストをクリアしたり破棄したりするとコマンドキャッシュは機能しなくなります。

コマンドリストの保存を開始するには、`nngxStartCmdlistSave()` を呼び出します。

コード 8-1. コマンドリストの保存開始

```
void nngxStartCmdlistSave(void);
```

この関数で保存を開始したあと、終了させていない状態で再度呼び出した場合は `GL_ERROR_8034_DMP` のエラーを、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場合は `GL_ERROR_8035_DMP` のエラーを生成します。

コマンドリストの保存を終了し、コマンドキャッシュの情報を取得するには `nngxStopCmdlistSave()` を呼び出します。

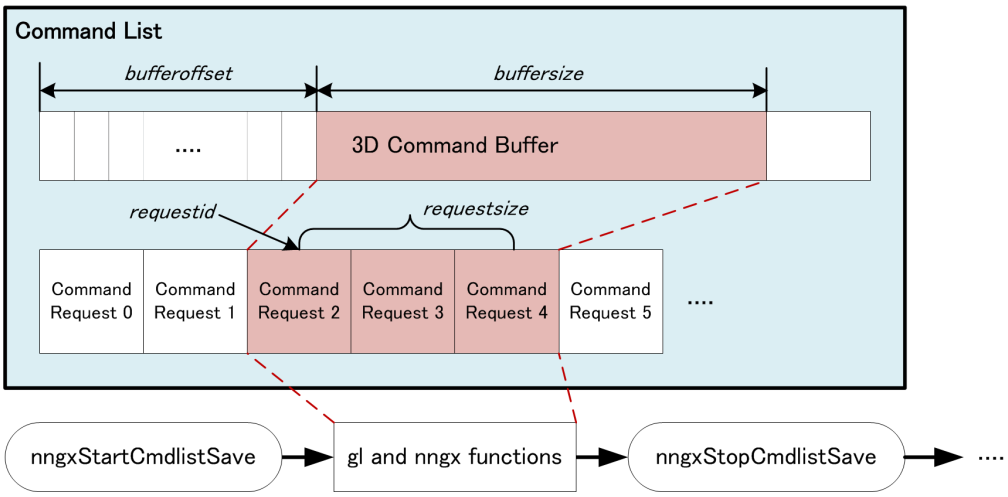
コード 8-2. コマンドリストの保存終了

```
void nngxStopCmdlistSave(GLuint* bufferoffset, GLsizei* buffersize,
        GLuint* requestid, GLsizei* requestsize);
```

コマンドキャッシュの情報として、*bufferoffset* には 3D コマンドバッファの保存開始オフセットが、*buffersize* には 3D コマンドバッファの保存バイトサイズが、*requestid* にはコマンドリクエストの保存開始 ID が、*requestsize* にはコマンドリクエストの保存数がそれぞれ返されます。コマンドリストの保存が開始されていない状態で呼び出した場合は `GL_ERROR_8036_DMP` のエラーを生成します。

コマンドリストの保存の開始と終了により、コマンドリストの 3D コマンドバッファに、パディング用のダミーコマンドが生成される場合があります。

図 8-1. コマンドリストの保存



8.1.1. ステート

ステートとは、3D グラフィックスを機能ごとにまとめた設定のことです。gl 関数などが呼び出されると対応するステートが更新され、更新されたステートに関連して生成された 3D コマンドが 3D コマンドバッファに蓄積されます。そのため、ステートが更新されたかどうかは、描画に必要な 3D コマンドが保存され、コマンドキャッシュが正しく機能するかどうかに関係しています。

1 つのステートは 1 つ以上の gl 関数またはユニフォーム設定に対応しています。また、1 つの gl 関数が複数のステートを更新する可能性もあります。必要とする 3D コマンドが確実に保存されるようにするなど、アプリケーションでステートを指定するときは、ステートフラグの論理和で指定する必要があります。

表 8-1. ステートフラグ

ステートフラグ	説明
NN_GX_STATE_SHADERBINARY	シェーダバイナリのステートです。 glUseProgram() で切り替わった前後のプログラムオブジェクトにリンクされているシェーダオブジェクトが、異なる glShaderBinary() でロードされていた場合に更新されます。 更新された場合、シェーダアセンブラコードをロードするための 3D コマンドを生成します。
NN_GX_STATE_SHADERPROGRAM	シェーダプログラムのステートです。 glUseProgram() により、異なるプログラムオブジェクトに切り替わった場合に更新されます。 更新された場合、頂点属性の構成の設定などに関する 3D コマンドを生成します。前回のバリデーション時と比較して異なる設定のレジスタに関してのみ 3D コマンドが生成されます。
NN_GX_STATE_SHADERMODE	シェーダモードのステートです。 glUseProgram() により、ジオメトリシェーダの使用・不使用が切り替わった場合に更新されます。 更新された場合、ジオメトリシェーダの使用・不使用を切り替える 3D コマンドを生成します。
NN_GX_STATE_SHADERFLOAT	シェーダの浮動小数点数定義のステートです。 glUseProgram() により、アタッチされているシェーダオブジェクトが異なるプログラムオブジェクトに切り替わった場合に更新されます。 更新された場合、シェーダアセンブラで def 命令によって定義された値を浮動小数点数レジスタに設定する 3D コマンドを生成します。

NN_GX_STATE_VSUNIFORM	頂点シェーダのユニフォーム設定のステートです。 glUseProgram() により、異なるプログラムオブジェクトに切り替わった場合や、glUniform*() で頂点シェーダのユニフォームの値が設定された場合に更新されます。整数型のユニフォームは設定が変更された場合にのみステートが更新されますが、浮動小数点数型のユニフォームは設定が変更されなくても更新されます。 更新された場合、シェーダアセンブラでユニフォームとして定義された浮動小数点数レジスタ、ブールレジスタ、整数レジスタに値を設定する 3D コマンドを生成します。
NN_GX_STATE_FSUNIFORM	予約フラグメントシェーダのユニフォーム設定のステートです。 glUseProgram() により、異なるプログラムオブジェクトに切り替わった場合や、glUniform*() で予約フラグメントシェーダのユニフォームの値が設定された場合に更新されます。 更新された場合、予約フラグメントシェーダに関するレジスタに値を設定する 3D コマンドを生成します。
NN_GX_STATE_LUT	参照テーブルのステートです。 glBindTexture()、glTexImage1D()、glTexSubImage1D() により、参照テーブルの内容が変更された場合や、バインドされていた参照テーブルが glDeleteTextures() で削除された場合、glUseProgram() または glUniform*() で参照テーブルに関係するユニフォームの値が変更された場合に更新されます。 更新された場合、参照テーブルを設定する 3D コマンドを生成します。
NN_GX_STATE_TEXTURE	テクスチャ(プロシージャルテクスチャを除く)のステートです。 glBindTexture()、glTexImage2D()、glCopyTexImage2D()、glCopyTexSubImage2D()、glCompressedTexImage2D()、glTexParameter*() が呼び出された場合や、バインドされているテクスチャが glDeleteTextures() で削除された場合、glUseProgram() または glUniform*() で予約フラグメントシェーダのユニフォーム ″dmp_Texture[*i*].samplerType″ の値が変更された場合に更新されます。 更新された場合、テクスチャユニットに関する 3D コマンド(プロシージャルテクスチャを除く)を生成します。
NN_GX_STATE_FRAMEBUFFER	フレームバッファのバッファ情報のステートです。 glBindFramebuffer()、glBindFramebufferRenderbuffer()、glFramebufferTexture2D()、glDeleteFramebuffers()、glBindRenderbuffer()、glRenderbufferStorage()、glDeleteRenderbuffers() が呼び出された場合に更新されます。 更新された場合、フレームバッファのフォーマットやバッファアドレスに関する 3D コマンドを生成します。
NN_GX_STATE_VERTEX	頂点属性データのステートです。 glBindBuffer()、glBufferData()、glBufferSubData()、glEnableVertexAttribArray()、glDisableVertexAttribArray()、glVertexAttrib()、glVertexAttribPointer()、glUseProgram() が呼び出された場合や、バインドされている頂点バッファが glDeleteBuffers() で削除された場合に更新されます。 更新された場合、頂点属性データに関する 3D コマンドを生成します。
NN_GX_STATE_TRIOFFSET	ポリゴンオフセットのステートです。 GL_POLYGON_OFFSET_FILL の設定が glEnable()、glDisable() により変更された場合や、glDepthRangef()、glPolygonOffset() により設定が変更された場合、glUseProgram() が呼び出された場合に更新されます。 更新された場合、ポリゴンオフセットに関する 3D コマンドを生成します。

NN_GX_STATE_FBACCESS	フレームバッファのアクセス方法のステートです。 GL_COLOR_LOGIC_OP、GL_BLEND、GL_DEPTH_TEST、GL_STENCIL_TEST、GL_EARLY_DEPTH_TEST_DMP の設定が glEnable()、glDisable() により変更された場合や、glDepthFunc()、glEarlyDepthFuncDMP()、glColorMask()、glDepthMask()、glStencilMask() により設定が変更された場合、glUniform*() で予約フラグメントシェーダのユニフォーム "dmp_FragOperation.mode" が設定された場合に更新されます。 更新された場合、フレームバッファの読み書きなどのアクセス方法に関する 3D コマンドを生成します。
NN_GX_STATE_SCISSOR	シザーテストのステートです。 GL_SCISSOR_TEST の設定が glEnable()、glDisable() により変更された場合や、glScissor()、glViewport() により設定が変更された場合、シザーテストが有効であるときにフレームバッファのサイズが変更された場合に更新されます。 更新された場合、シザーテストに関する 3D コマンドを生成します。
NN_GX_STATE_OTHERS	glDraw*() 以外で 3D コマンドを生成する関数のステートです。 glDrawElements()、glDrawArrays() 以外で、3D コマンドが生成される関数が呼び出された場合に更新されます。 更新された場合に生成される 3D コマンドについては、表 8-3 を参照してください。
NN_GX_STATE_ALL	上記すべてを指定するためのフラグです。

8.1.2. コマンド生成

3D コマンドバッファに蓄積される 3D コマンドのほとんどは glDrawElements()、glDrawArrays() で生成されます。これらの関数は内部でステートをチェックし、更新されたステートに関する 3D コマンドを生成します。これをバリデートと呼びます。

nngxValidateState() は、バリデートを任意のタイミングで行うことができます。また、nngxUpdateState() で、指定したステートフラグのステートを更新状態にし、次のタイミングで指定したステートの 3D コマンドをすべて生成させることができます。

コード 8-3. ステートのバリデートとステートフラグの指定

```
void nngxValidateState(GLbitfield statemask, GLboolean drawelements);
void nngxUpdateState(GLbitfield statemask);
```

nngxValidateState() が呼び出されると、更新状態になっているステートのうちの *statemask* で指定されたものだけがバリデートされ、更新状態が解除されます。この関数では描画が行われませんので、実際に描画するときに glDrawElements() を呼び出すのか、glDrawArrays() を呼び出すのかを *drawelements* で指定します。GL_TRUE を渡した場合は glDrawElements()、GL_FALSE を渡した場合は glDrawArrays() に対応した 3D コマンドを生成します。つまり、nngxUpdateState() と nngxValidateState() を組み合わせることで、任意のステートに関する 3D コマンドをすべて生成させることができます。

一方、glDrawElements() などでは更新されているすべてのステートがバリデートされ、更新状態が解除されます。

ほかの関数と異なり、任意のタイミングで 3D コマンドを生成することのできる nngxValidateState() は、ステート間で 3D コマンドの設定順の依存関係があることに注意しなければなりません。以下の条件に抵触する順番で生成された 3D コマンドが実行された場合の動作は不定となっています。

- NN_GX_STATE_FBACCESS および NN_GX_STATE_TRIOFFSET は、NN_GX_STATE_FSUNIFORM より先、または同時に指定して呼び出さなければなりません。
- NN_GX_STATE_SHADERMODE は、NN_GX_STATE_SHADERBINARY、NN_GX_STATE_SHADERPROGRAM、NN_GX_STATE_SHADERFLOAT、および NN_GX_STATE_VSUNIFORM より先、または同時に指定して呼び出さなけ

ればなりません。

- NN_GX_STATE_FRAMEBUFFER、および NN_GX_STATE_OTHERS は、NN_GX_STATE_FBACCESS より先、または同時に指定して呼び出さなければなりません。
- NN_GX_STATE_FRAMEBUFFER は、NN_GX_STATE_SCISSOR より先、または同時に指定して呼び出さなければなりません。

表 8-2. nngxValidateState() が生成するエラー

エラー	原因
GL_ERROR_8066_DMP	呼び出した結果、3D コマンドバッファがオーバーフローした
GL_ERROR_806C_DMP	バリデートにより各種エラーが発生した
GL_ERROR_80B2_DMP	3D コマンドバッファが設定されていない
GL_ERROR_80B3_DMP	有効なプログラムオブジェクトが設定されていない(glUseProgram() が一度も呼び出されていない、もしくは glUseProgram() を呼び出したときに *program* に 0 が指定されていた)状態で、プログラムオブジェクトに関連する以下のステートをバリデートしようとした。 ● NN_GX_STATE_SHADERBINARY ● NN_GX_STATE_SHADERPROGRAM ● NN_GX_STATE_SHADERMODE ● NN_GX_STATE_SHADERFLOAT ● NN_GX_STATE_VSUNIFORM ● NN_GX_STATE_FSUNIFORM ● NN_GX_STATE_LUT ● NN_GX_STATE_VERTEX ● NN_GX_STATE_TRIOFFSET ● NN_GX_STATE_FBACCESS

バリデートによりエラーとなる原因には以下のものがあります。

- 有効になっているテクスチャに対してテクスチャメモリがアロケートされていない。
  解決するには、glTexImage2D()、glCompressedTexImage2D()、glCopyTexImage2D() のいずれかを呼び出して、テクスチャメモリをアロケートする必要があります。キューブマップテクスチャは、6 面すべてがアロケートされていなければなりません。
- 不正なフォーマットのテクスチャがバインドされています。
  テクスチャユニット 1 または 2 に、GL_SHADOW_DMP フォーマットのテクスチャをバインドしている、または、キューブマップテクスチャに GL_GAS_DMP フォーマットのテクスチャをバインドしているなどが考えられます。
- キューブマップテクスチャの各面の設定が異なっています。
  キューブマップテクスチャの 6 面は、幅、高さ、フォーマット、ミップマップ段数がすべて同じである必要があります。
- キューブマップテクスチャの 6 面すべてのアドレスの上位 7 ビットが共通の値になっていません。
  6 面の各アドレスは上位 7 ビットが同じである必要があります。
- 参照テーブルオブジェクトが正しくバインドされていない、または参照テーブル番号が正しく指定されていません。
  フラグメントライティング、プロシージャルテクスチャ、フォグ、ガスの各機能で参照テーブルを使用する設定になっている場合、該当する参照テーブル番号に有効な参照テーブルオブジェクトがバインドされている必要があります。また、参照テーブル番号を指定するユニフォームの設定も正しく行う必要があります。
- 参照テーブルの内部フォーマット値を格納するために必要な領域の確保に失敗しました。

バリデートの途中でエラーとなった場合、各ステートはバリデート済みとして扱われます。エラー後にコマンドを正しく生成するためには、nngxUpdateState() を呼び出し、再度ステートを更新してください。

基本的に 3D コマンドは glDraw 系の関数で生成されますが、ステートフラグ NN_GX_STATE_FRAMEBUFFER のステートは glReadPixels() または glClear() でも生成されます。そのほかにも、以下の関数を呼び出すことで 3D コマンド

が生成されます。

表 8-3. 3D コマンドを生成する関数

関数	生成条件
glBlendColor()	設定値を変更したとき。
glBlendEquation()	設定値を変更したとき。
glBlendEquationSeparate()	設定値を変更したとき。
glBlendFunc()	設定値を変更したとき。
glBlendFuncSeparate()	設定値を変更したとき。
glClearEarlyDepthDMP()	設定値を変更したとき。
glColorMask()	設定値を変更したとき。
glCullFace()	設定値を変更したとき。
glDepthFunc()	設定値を変更したとき。
glDepthMask()	設定値を変更したとき。
glDisable()	GL_COLOR_LOGIC_OP、GL_BLEND、GL_DEPTH_TEST、GL_EARLY_DEPTH_TEST_DMP、GL_STENCIL_TEST、GL_CULL_FACE の設定値を変更したとき。他の設定値に関しては変更しても生成されません。
glEarlyDepthFuncDMP()	設定値を変更したとき。
glEnable()	GL_COLOR_LOGIC_OP、GL_BLEND、GL_DEPTH_TEST、GL_EARLY_DEPTH_TEST_DMP、GL_STENCIL_TEST、GL_CULL_FACE の設定値を変更したとき。他の設定値に関しては変更しても生成されません。
glFrontFace()	設定値を変更したとき。
glLogicOp()	設定値を変更したとき。
glRenderBlockModeDMP()	設定値を変更したとき。
glStencilFunc()	設定値を変更したとき。
glStencilMask()	設定値を変更したとき。
glStencilOp()	設定値を変更したとき。
glViewport()	必ず生成する。

上表の関数のうち、一部機能(GL_COLOR_LOGIC_OP、GL_BLEND、GL_DEPTH_TEST、GL_STENCIL_TEST、GL_EARLY_DEPTH_TEST_DMP)に対する glEnable() と glDisable()、glDepthFunc()、glEarlyDepthFuncDMP()、glColorMask()、glDepthMask()、glStencilMask() は NN_GX_STATE_FBACCESS のステートを更新するため、コマンドリストを保存する際に NN_GX_STATE_FBACCESS のバリデートを行う必要があります。

nngxUpdateState() で NN_GX_STATE_OTHERS が指定された場合、バリデート時に上表の関数すべてに関する 3D コマンドが生成されるようになります。また、同時に NN_GX_STATE_FBACCESS も更新状態となります。

8.1.3. 差分コマンドと完全コマンド

gl 関数は複数のステートを更新する可能性があります。通常、呼び出した関数が更新したステートに関する 3D コマンドだけが生成されます。これを差分コマンドと呼びます。一方、呼び出した関数が更新したかどうかに関わらず、関連するすべてのステートに関する 3D コマンドを生成することを完全コマンドと呼びます。ただし、完全コマンドはステートごとに一括してすべての 3D コマンドを生成させるため、3D コマンドの生成が冗長になる場合があります。

nngxSetCommandGenerationMode() を呼び出すことで、いくつかのステートに対して完全コマンドを生成するように設定することができます。また、nngxUpdateState() を呼び出すことでも、指定されたステートの完全コマンドを生成するように設定することができます。

コード 8-4. コマンド出力モードの設定

```
void nngxSetCommandGenerationMode(GLenum mode);
```

mode に NN_GX_CMDGEN_MODE_CONDITIONAL を指定した場合は差分コマンドを生成するモードに設定します。デフォルトはこのモードです。NN_GX_CMDGEN_MODE_UNCONDITIONAL を指定した場合は完全コマンドを生成するモードに設定します。それ以外の値を *mode* に指定した場合は GL_ERROR_804D_DMP のエラーを生成します。現在のモード設定を nngxGetCommandGenerationMode() で取得することができます。

コード 8-5. コマンド出力モードの取得

```
void nngxGetCommandGenerationMode(GLenum* mode);
```

mode に現在のモード設定が返されます。

完全コマンドを生成するモードで影響を受けるステートには以下のものがあります。

- 予約フラグメントシェーダのユニフォームの設定に関するステート
- 頂点シェーダの整数型ユニフォームの設定に関するステート
- 参照テーブルデータの設定に関するステート
- 表 8-3 に記載されている関数に関するステート

更新されたステートの 3D コマンドに加えて、更新されたかどうかに関わらず、上記のステートに関しての完全コマンドが生成されます。ただし、参照テーブルに関連するステートすべてが更新されていても、3D コマンドは有効になっている参照テーブルの分だけが生成されます。参照テーブルが有効となる条件は以下のようになっています。

表 8-4. 参照テーブルが有効となる条件

参照テーブル	条件
フラグメントライティングの反射の赤成分(RR)	dmp_FragmentLighting.enabled に GL_TRUE が設定されている、かつ dmp_LightEnv.config の設定が RR を使用する設定になっている、かつ dmp_LightEnv.lutEnabledRefl に GL_TRUE が設定されている。
フラグメントライティングの反射の緑成分(RG)	dmp_FragmentLighting.enabled に GL_TRUE が設定されている、かつ dmp_LightEnv.config の設定が RG を使用する設定になっている、かつ dmp_LightEnv.lutEnabledRefl に GL_TRUE が設定されている。
フラグメントライティングの反射の青成分(RB)	dmp_FragmentLighting.enabled に GL_TRUE が設定されている、かつ dmp_LightEnv.config の設定が RB を使用する設定になっている、かつ dmp_LightEnv.lutEnabledRefl に GL_TRUE が設定されている。
フラグメントライティングのディストリビューションファクタ 0(D0)	dmp_FragmentLighting.enabled に GL_TRUE が設定されている、かつ dmp_LightEnv.config の設定が D0 を使用する設定になっている、かつ dmp_LightEnv.lutEnabledD0 に GL_TRUE が設定されている。

フラグメントライティングのディストリビューションファクタ 1(D1)	dmp_FragmentLighting.enabled に GL_TRUE が設定されている、かつ dmp_LightEnv.config の設定が D1 を使用する設定になっている、かつ dmp_LightEnv.lutEnabledD1 に GL_TRUE が設定されている。
フラグメントライティングのフレネルファクタ(FR)	dmp_FragmentLighting.enabled に GL_TRUE が設定されている、かつ dmp_LightEnv.config の設定が FR を使用する設定になっている、かつ dmp_LightEnv.fresnelSelector に GL_LIGHT_ENV_NO_FRESNEL_DMP 以外の値が設定されている。
フラグメントライティングのスポットライト減衰(SP)	dmp_FragmentLighting.enabled に GL_TRUE が設定されている、かつ dmp_LightEnv.config の設定が SP を使用する設定になっている、かつ dmp_FragmentLightSource[*i*].enabled に GL_TRUE が設定されている、かつ dmp_FragmentLightSource[*i*].spotEnabled に GL_TRUE が設定されている。
フラグメントライティングの距離減衰	dmp_FragmentLighting.enabled に GL_TRUE が設定されている、かつ dmp_FragmentLightSource[*i*].enabled に GL_TRUE が設定されている、かつ dmp_FragmentLightSource[*i*].distanceAttenuationEnabled に GL_TRUE が設定されている。
プロシージャルテクスチャのRGB マッピングの F 関数	dmp_Texture[3].samplerType に GL_TEXTURE_PROCEDURAL_DMP が設定されている。
プロシージャルテクスチャのアルファマッピングの F 関数	dmp_Texture[3].samplerType に GL_TEXTURE_PROCEDURAL_DMP が設定されている、かつ dmp_Texture[3].ptAlphaSeparate に GL_TRUE が設定されている。
プロシージャルテクスチャのノイズ変調関数	dmp_Texture[3].samplerType に GL_TEXTURE_PROCEDURAL_DMP が設定されている、かつ dmp_Texture[3].ptNoiseEnable に GL_TRUE が設定されている。
プロシージャルテクスチャのカラー参照テーブル	dmp_Texture[3].samplerType に GL_TEXTURE_PROCEDURAL_DMP が設定されている。
フォグのフォグ係数	dmp_Fog.mode に GL_FALSE 以外の値が設定されている。
ガスのシェーディング参照テーブル	dmp_Fog.mode に GL_GAS_DMP が設定されている。

8.1.4. コマンドキャッシュの制限事項および注意事項

コマンドキャッシュを使用する場合、以下の制限事項および注意事項があります。

- フラグメントライティングが有効(予約ユニフォーム dmp_FragmentLighting.enabled が GL_TRUE)、かつ全光源が無効(予約ユニフォーム dmp_FragmentLightSource[*i*].enabled がすべて GL_FALSE)の場合、nngxValidateState() で予約フラグメントシェーダユニフォームのステートフラグ(NN_GX_STATE_FSUNIFORM)を指定してバリデートしても、その後で描画関数を呼び出したときに再度ライティングに関連する 3D コマンドが生成されます。
- 予約ユニフォーム dmp_Gas.accMax に関連する 3D コマンドは、nngxValidateState() で予約フラグメントシェーダユニフォームのステートフラグ(NN_GX_STATE_FSUNIFORM)を指定してバリデートしても、その後で描画関数を呼び出したときに生成されます。
- 予約フラグメントユニフォーム dmp_Gas.autoAcc に GL_TRUE が設定されている場合、予約ユニフォーム dmp_FragOperation.mode の値が GL_FRAGOP_MODE_GAS_ACC_DMP に変更されたタイミングおよび GL_FRAGOP_MODE_GAS_ACC_DMP から別の設定に変更されたタイミングでコマンドリストの保存開始、保存終了を行うと、保存されたコマンドリストの dmp_Gas.autoAcc に関連するコマンドが正しく反映されない場合があります。
- 実行する 3D コマンドバッファのバイトサイズは 16 の倍数でなければなりません。nngxAdd3DCommand() で 0x00000000_00000000 をダミーとして追加することでサイズを調整してください。
- glUseProgram() の引数に 0 を指定して呼び出した状態で、プログラムやシェーダに関するステートをバリデートしても、コマンドは何も生成されません。

8.1.5. 更新されたステートの取得

nngxGetUpdateState() で更新されているステートをステートフラグで取得することができます。

コード 8-6. 更新されたステートの取得

```
void nngxGetUpdatedState (GLbitfield* statemask);
```

8.1.6. ステート更新の無効化

nngxInvalidateState() でステートの更新を無効化することができます。

コード 8-7. ステート更新の無効化

```
void nngxInvalidateState (GLbitfield statemask);
```

statemask に更新を無効化するステートフラグを論理和で指定します。*statemask* に指定されたステートフラグに対応するステートは、更新されていても関連するコマンドが生成されなくなります。

8.2. コマンドリストの再使用

nngxUseSavedCmdlist() または nngxUseSavedCmdlistNoCacheFlush() にコマンドリストの保存によって取得したコマンドキャッシュの情報を渡すことで、3D コマンドおよびコマンドリクエストを再使用することができます。前者は追加する 3D コマンドのキャッシュフラッシュを行いますが、後者は行いません。

コード 8-8. コマンドリストの再使用

```
void nngxUseSavedCmdlist(GLuint cmdlist, GLuint bufferoffset,
        GLsizei buffersize, GLuint requestid, GLsizei requestsize,
        GLbitfield statemask, GLboolean copycmd);
void nngxUseSavedCmdlistNoCacheFlush(GLuint cmdlist,
        GLuint bufferoffset, GLsizei buffersize,
        GLuint requestid, GLsizei requestsize, GLbitfield statemask);
```

cmdlist には、保存で指定した 3D コマンドが蓄積されているコマンドリストを指定します。

bufferoffset には 3D コマンドバッファの保存開始オフセット、*buffersize* には 3D コマンドバッファの保存バイトサイズ、*requestid* にはコマンドリクエストの保存開始 ID、*requestsize* にはコマンドリクエストの保存数を指定しますが、これらは同じタイミングで保存されたコマンドキャッシュの情報でなければなりません。コマンドキャッシュの情報が指定されたコマンドリストのものであるかどうかや、すべてが同じコマンドキャッシュの情報であるかなどはチェックしません。正しくないコマンドキャッシュ情報を指定した場合の動作は不定です。

statemask には完全コマンドを生成するステートをステートフラグの論理和で指定します。この関数が呼び出された前後ではステートの設定に矛盾が生じてしまいます。この矛盾を解消するためには、すべてのステートに対して完全コマンドを生成しなければならない場合がありますが、すべてのステートの完全コマンドを生成することは冗長となることがあるため、*statemask* で指定されたステートに対してのみ完全コマンドを生成します。*statemask* に NN_GX_STATE_OTHERS を指定した場合、表 8-3 の関数すべてに関する完全コマンドが生成されます。

copycmd には、コマンドキャッシュ機能で実行する 3D コマンドを保存したコマンドリストからコピーして実行するか、保存したコマンドリストの 3D コマンドそのものを実行するかを指定します。GL_TRUE を指定した場合は 3D 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でコピーされます。この方式は 3D コマンドのコピーが行われることによる CPU 負荷が高いため、サイズの小さな 3D コマンドバッファの再使用に適しています。GL_FALSE を指定した場合はコピーされません。この方式は 3D コマンドのコピーが行われず CPU 負荷が軽減されるため、サイズの大きな 3D コマンドバッファの再使用に適し

ています。この指定により制御されるのは 3D コマンドバッファへの追加コピーだけで、コマンドリクエストは必ず追加されます。

保存したコマンドリストから 3D コマンドをコピーしない場合、3D コマンドが区切られていなければ関数内で再使用の直前に区切りのコマンドを追加し、3D コマンドの実行アドレスを保存したコマンドリストに切り替えます。そのため、保存したコマンドリスト側に区切りのコマンドがなければ元のコマンドリストに戻ることができなくなってしまいます。この方式で保存したコマンドリストから再使用する場合は、保存終了前に `nngxSplitDrawCmdlist()` で区切りのコマンドを必ず追加してください。

保存したコマンドリストから 3D コマンドをコピーする場合、保存したコマンドリストのコマンドリクエストが 3D 実行コマンド以外かつ 3D コマンドが区切られていなければ関数内で再使用の直前に区切りのコマンドを追加します。

`nngxUseSavedCmdlistNoCacheFlush()` の動作は、`nngxUseSavedCmdlist()` の *copycmd* に `GL_FALSE` を指定したときと同じですが、*cmdlist* で指定したコマンドリストの 3D コマンドバッファのキャッシュフラッシュを行いません。3D コマンドバッファの領域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。

図 8-2. nngxUseSavedCmdlist() の copycmd による動作の違い

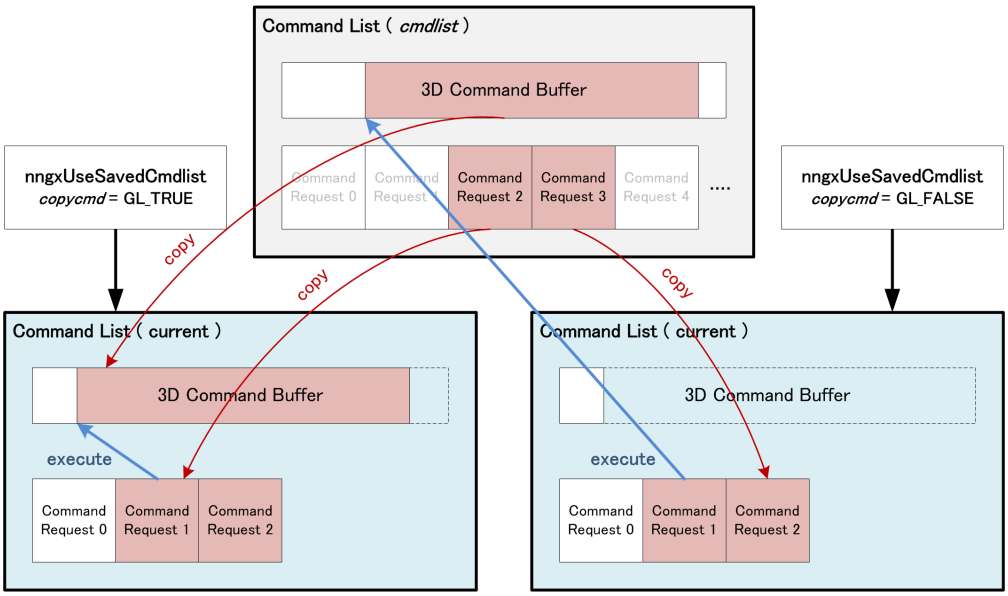


表 8-5. nngxUseSavedCmdlist() および nngxUseSavedCmdlistNoCacheFlush() が生成するエラー

エラー	原因
`GL_ERROR_8037_DMP` `GL_ERROR_8092_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_8038_DMP` `GL_ERROR_8093_DMP`	*cmdlist* に指定したコマンドリストが存在しない
`GL_ERROR_803A_DMP` `GL_ERROR_8095_DMP`	この関数の実行により、現在バインドされているコマンドリストの 3D コマンドバッファまたはコマンドリクエストの保持数をオーバーする

8.2.1. コピーされるコマンドリクエストの情報

`nngxUseSavedCmdlist()` の実行では、*copycmd* の指定に関わらず、コマンドリクエストは必ず現在バインドされているコマンドリストに追加でコピーされます。コマンドリクエストは各コマンドで決められた情報を保持しており、保存されたときからステートが更新された場合でもそのままの情報がコピーされます。ただし、コピーされる最初の 3D 実行コマンドだけは情

報が変更される可能性があります。

DMA 転送コマンド

DMA 転送をする転送元アドレス、転送先アドレス、転送サイズが保持されています。

3D 実行コマンド

3D コマンドバッファの実行開始アドレス、実行サイズが保持されています。保存を開始した 3D コマンドバッファのアドレスが実行開始アドレスと等しくない場合は、コマンドリクエストのコピー時に実行開始アドレスを保存開始アドレスに置き換え、実行サイズも併せて変更します。

メモリフィルコマンド

フィルされるカラーバッファの先頭アドレス、サイズ、クリアカラー、デプスステンシルバッファの先頭アドレス、サイズ、クリアデプス値、クリアステンシル値が保持されています。

ポスト転送コマンド

転送元のカラーバッファのアドレス、解像度、フォーマット、転送先のディスプレイバッファのアドレス、解像度、フォーマットが保持されています。

レンダーテクスチャ転送コマンド

転送元のカラーバッファのアドレス、解像度、転送先のテクスチャのアドレス、解像度が保持されています。

8.3. コマンドリストのコピー

nngxCopyCmdlist() を呼び出すことで、コマンドリストを別のコマンドリストにコピーすることができます。コマンドリストの情報すべてをコピーするため、コピー先のコマンドリストに蓄積されている 3D コマンドもコマンドリクエストも上書きされることに注意してください。

コード 8-9. コマンドリストのコピー

```
void nngxCopyCmdlist(GLuint scmdlist, GLuint dcmdlist);
```

scmdlist にコピー元のコマンドリスト、*dcmdlist* にコピー先のコマンドリストを指定します。

この関数の動作は直接コマンドキャッシュに関係しませんが、コマンドキャッシュの情報がオフセットを基準に作成されることから、コマンドリストの保存直後に作成したコピーを再利用することができます。そのため、コピー後はコピー元のコマンドリストをクリアすることもできます。

表 8-6. nngxCopyCmdlist() が生成するエラー

エラー	原因
GL_ERROR_8047_DMP	現在バインドされているコマンドリストを *dcmdlist* に指定した
GL_ERROR_8048_DMP	*scmdlist* に指定したコマンドリストが存在しない
GL_ERROR_8049_DMP	*dcmdlist* に指定したコマンドリストが存在しない
GL_ERROR_804A_DMP	*scmdlist* と *dcmdlist* に同じコマンドリストを指定した
GL_ERROR_804B_DMP	実行中のコマンドリストを *dcmdlist* に指定した

GL_ERROR_804C_DMP	*scmdlist* に指定されたコマンドリストに蓄積されている 3D コマンドまたはコマンドリクエストが、*dcmdlist* に指定されたコマンドリストで保持可能なサイズよりも大きかった

8.3.1. コマンドリストの追加コピー

nngxCopyCmdlist() ではコマンドリストの上書きコピーにのみ対応していましたが、nngxAddCmdlist() は現在バインドされているコマンドリストに別のコマンドリストの情報すべてを追加でコピーすることができます。

コード 8-10. コマンドリストの追加コピー

```
void nngxAddCmdlist(GLuint cmdlist);
```

cmdlist にコピー元のコマンドリストを指定します。

コピー元のコマンドリストに蓄積されたすべてのコマンドは、現在バインドされているコマンドリストに追加で蓄積されます。現在バインドされているコマンドリストに、すでにコマンドが蓄積されている場合は蓄積済みコマンドのあとに、コマンドが追加されます。

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区切られた直後ではなく、かつ追加する側のコマンドリクエストの先頭のコマンドが 3D 実行コマンドではない場合は、ライブラリ内で nngxSplitDrawCmdlist() を呼び出し、コマンドバッファを区切ってから追加を行います。

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区切られた直後ではなく、かつ追加するコマンドリクエストの先頭のコマンドが 3D 実行コマンドである場合は、必要に応じてコピー先のコマンドバッファにダミーコマンドを追加することでアライメントしてからコマンドが追加されます。

ライブラリ内で nngxSplitDrawCmdlist() を呼び出したときやダミーコマンドを追加したときは、そのサイズも含めて上限サイズのチェックが行われます。

表 8-7. nngxAddCmdlist() が生成するエラー

エラー	原因
GL_ERROR_8054_DMP	*cmdlist* に不正な値を指定した
GL_ERROR_8055_DMP	現在バインドされているコマンドリストがない
GL_ERROR_8056_DMP	*cmdlist* に指定したコマンドリストと現在バインドされているコマンドリストが同じ
GL_ERROR_8057_DMP	現在バインドされているコマンドリストが実行中
GL_ERROR_8058_DMP	現在バインドされているコマンドリストにコマンドバッファおよびコマンドリクエストを追加することで、バッファの上限サイズを超えてしまう

8.4. コマンドリストのエクスポート

nngxExportCmdlist() を呼び出すことで、コマンドキャッシュで取得したコマンドリストの内容をバイナリデータとしてメモリに保存(エクスポート)することができます。

コード 8-11. コマンドリストのエクスポート

```
GLsizei nngxExportCmdlist(GLuint cmdlist, GLuint bufferoffset,
        GLsizei buffersize, GLuint requestid, GLsizei requestsize,
        GLsizei datasize, GLvoid* data);
```

cmdlist にはエクスポートするコマンドリストを指定します。

bufferoffset と *buffersize* には、エクスポートする 3D コマンドバッファの領域をバイトオフセットとバイトサイズで指定します。*requestid* と *requestsize* には、エクスポートするコマンドリクエストの開始 ID(0 から始まる蓄積順序)とエクスポートの対象とするコマンドリクエストの数を指定します。

data と *datasize* には、エクスポート先のメモリ領域の先頭アドレスとそのサイズを指定します。この関数は返り値でエクスポートされたデータのバイトサイズを返しますが、*data* に 0(NULL)を指定した場合はエクスポートを行わず、エクスポートに必要なメモリのサイズを返します。エクスポートの手順としては、エクスポートに必要なメモリのサイズを取得してからメモリ領域を確保し、エクスポートすることを想定しています。

この関数の *bufferoffset*、*buffersize*、*requestid*、*requestsize* には、矛盾のない組み合わせの値が指定されなければなりません。安全なデータをエクスポートするには、nngxStopCmdlistSave() で取得できる保存情報や、3D コマンドの蓄積中に同じタイミングで nngxGetCmdlistParameteri() を呼び出して取得した値を使用することを推奨します。nngxGetCmdlistParameteri() の *pname* に NN_GX_CMDLIST_USED_BUFSIZE を渡して蓄積済みの 3D コマンドバッファのサイズ、NN_GX_CMDLIST_USED_REQCOUNT を渡して蓄積済みのコマンドリクエストの個数が取得できます。エクスポート対象となるコマンドリストの蓄積の開始時と終了時に、この二つの値を組みにして取得します。開始時に取得した値を *bufferoffset* と *requested* に、終了時の値から開始時の値を減算した値を *buffersize* と *requestsize* に指定してください。

エクスポートする 3D コマンドバッファ領域は、以下の条件を満たしていなければなりません。

- *bufferoffset* は、エクスポートするコマンドリクエストのうちの最初の 3D 実行コマンドが実行する領域内のどこかに指定されていなければなりません。
- エクスポートされるすべての 3D 実行コマンドが実行する区切りのコマンドはすべてエクスポートされていなければなりません。
- 最後の 3D 実行コマンドが実行する領域より後の領域までエクスポートすることができますが、その領域に区切りコマンドが含まれてはいけません。
- コマンドリクエストが 1 つもエクスポートされない場合は、逆に区切りのコマンドが含まれていてはいけません。

図 8-3. エクスポートする 3D コマンドバッファ領域の指定とその可否

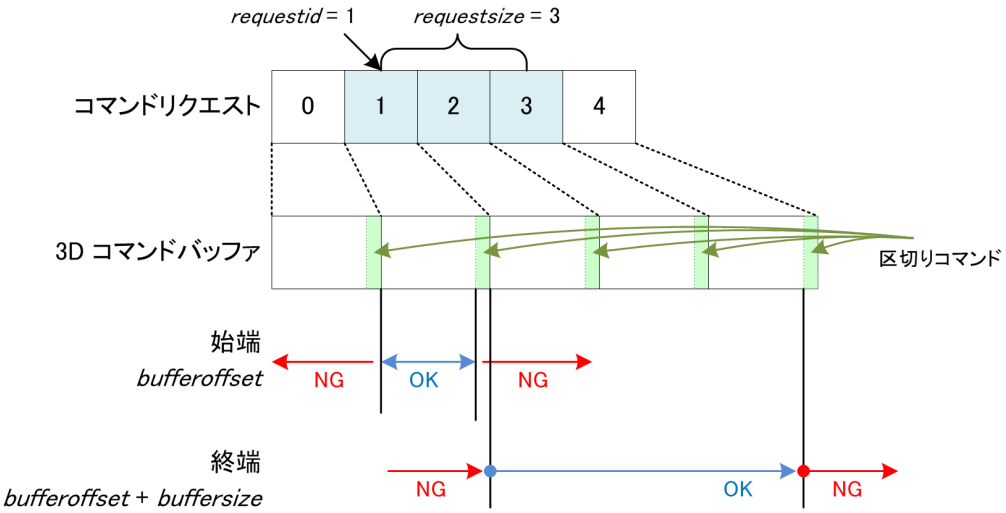


図 8-4. エクスポートする 3D コマンドバッファ領域の指定とその可否(コマンドリクエストが 1 つもエクスポートされない場合)

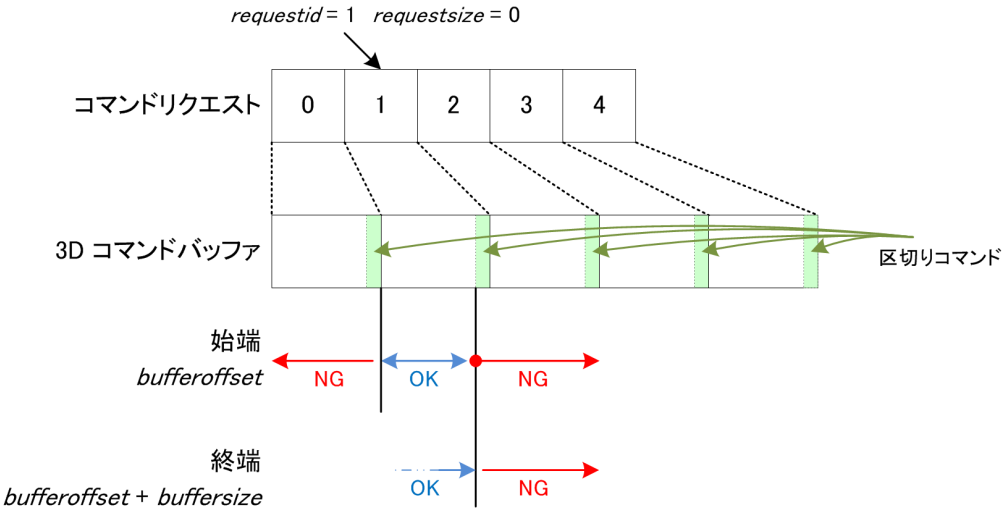


表 8-8. nngxExportCmdlist() が生成するエラー

エラー	原因
GL_ERROR_803B_DMP	*cmdlist* に不正な値(0 または存在しないコマンドリスト)を指定した
GL_ERROR_803C_DMP	*datasize* に指定されたサイズがエクスポートされるデータのサイズより小さい
GL_ERROR_803D_DMP	*requestid* と *requestsize* で指定された領域にコマンドが蓄積されていない
GL_ERROR_803E_DMP	*bufferoffset* または *buffersize* のアライメントが 8 バイトでない
GL_ERROR_803F_DMP	指定されたコマンドリストに nngxUseSavedCmdlist() のコピーしない方式で追加された 3D 実行コマンドが含まれている
GL_ERROR_8040_DMP	エクスポートされる 3D 実行コマンドが実行する 3D コマンドに対して、*bufferoffset* と *buffersize* による 3D コマンドバッファの指定が正しくない

8.4.1. エクスポート情報の取得

エクスポートされたバイナリデータに含まれているコマンドリストの情報(エクスポート情報)は、nngxGetExportedCmdlistInfo() で取得することができます。

コード 8-12. エクスポート情報の取得

```
void nngxGetExportedCmdlistInfo(GLvoid* data, GLsizei* buffersize,
        GLsizei* requestsize, GLuint* bufferoffset);
```

data にはエクスポートしたバイナリデータの先頭アドレスを指定します。不正なデータが指定された場合は GL_ERROR_8046_DMP のエラーを生成します。エクスポート情報として、*buffersize* には 3D コマンドバッファのバイトサイズが、*requestsize* にはコマンドリクエストの数が、*bufferoffset* には 3D コマンドバッファの先頭への *data* からのバイトオフセットがそれぞれ格納されます。

8.5. コマンドリストのインポート

nngxImportCmdlist() を呼び出すことで、エクスポートしたバイナリデータからコマンドリストに、3D コマンドを追加コピー(インポート)することができます。

コード 8-13. コマンドリストのインポート

```
void nngxImportCmdlist(GLuint cmdlist, GLvoid* data, GLsizei datasize);
```

cmdlist には、現在バインドされているコマンドリストも、バインドされていないコマンドリストのどちらも指定することができます。指定したコマンドリストに 3D コマンドがすでに蓄積されていた場合は、そのあとに追加する形でインポートが行われます。

インポートするデータの最初のコマンドリクエストが 3D 実行コマンドでない場合は、インポート先のコマンドリストをバインドして、nngxSplitDrawCmdlist() でコマンドの区切りを追加しなければなりません。

data と *datasize* には、エクスポートデータへのポインタとエクスポートデータのバイトサイズを指定します。

インポートの結果、インポート先のコマンドリストの 3D コマンドバッファに、パディング用のダミーコマンドが生成される場合があります。

表 8-9. nngxImportCmdlist() が生成するエラー

エラー	原因
GL_ERROR_8041_DMP	*cmdlist* に不正な値(0 または存在しないコマンドリスト)を指定した
GL_ERROR_8042_DMP	*data* に不正なポインタを指定した
GL_ERROR_8043_DMP	*datasize* で指定したバイトサイズとエクスポートデータのサイズが異なる
GL_ERROR_8044_DMP	インポートで追加される 3D コマンドまたはコマンドリクエストが、インポート先のコマンドリストで保持可能なサイズを超える
GL_ERROR_8045_DMP	コマンドの区切りでない状態のコマンドリストに、最初のコマンドリクエストが 3D 実行コマンドでないデータをインポートした

8.6. 3D コマンドの追加

nngxAdd3DCommand() または nngxAdd3DCommandNoCacheFlush() で、指定した領域のデータを現在バインドされているコマンドリストの 3D コマンドバッファに追加したり、指定した領域を実行する 3D 実行コマンドを追加したりすることができます。前者は追加する 3D コマンドのキャッシュフラッシュを行いますが、後者は行いません。

コード 8-14. 3D コマンドの追加

```
void nngxAdd3DCommand(const GLvoid* bufferaddr, GLsizei buffersize,
        GLboolean copycmd);
void nngxAdd3DCommandNoCacheFlush(const GLvoid* bufferaddr,
        GLsizei buffersize);
```

この関数は *copycmd* の指定によって動作が異なります。

copycmd に GL_TRUE が指定されている場合、*bufferaddr* と *buffersize* で指定された先頭アドレスとバイトサイズの領域に格納されている 3D コマンドを、現在バインドされているコマンドリストの 3D コマンドバッファに追加コピーします。指定された領域にコマンドの区切りが存在する場合の動作は保証されていません。*buffersize* は 4 の倍数の正値でな

ければなりません。

copycmd に GL_FALSE が指定されている場合、*bufferaddr* と *buffersize* で指定された先頭アドレスとバイトサイズの領域に格納されている 3D コマンドを実行する 3D 実行コマンドをコマンドリクエストに追加します。現在バインドされているコマンドリストの 3D コマンドバッファがコマンドの区切りで区切られていない場合は、内部でコマンドの区切りを追加してからコマンドリクエストを追加します。指定された領域の最後の 3D コマンドがコマンドの区切りでない場合の動作は保証されていません。*bufferaddr*は 16 の倍数の値、 *buffersize* は 16 の倍数の正値でなければなりません。

nngxAdd3DCommandNoCacheFlush() の動作は、nngxAdd3DCommand() の *copycmd* に GL_FALSE を指定したときと同じですが、追加する 3D コマンドのキャッシュフラッシュを行いません。

表 8-10. nngxAdd3DCommand() および nngxAdd3DCommandNoCacheFlush() が生成するエラー

エラー	原因
GL_ERROR_804E_DMP GL_ERROR_808C_DMP	コマンドリストがバインドされていない状態で呼び出した
GL_ERROR_804F_DMP GL_ERROR_808D_DMP	*buffersize* に不正な値を指定した
GL_ERROR_8050_DMP	*copycmd* が GL_TRUE のとき、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サイズが足りなくなる
GL_ERROR_8051_DMP	*copycmd* が GL_FALSE のとき、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サイズが足りなくなる
GL_ERROR_8052_DMP	*copycmd* が GL_FALSE のときに *bufferaddr* に指定した値が 16 の倍数ではなかった
GL_ERROR_808E_DMP	*bufferaddr* に指定した値が 16 の倍数ではなかった
GL_ERROR_808F_DMP	現在バインドされているコマンドリストのコマンドリクエストのサイズが足りなくなる

8.6.1. 3D コマンドの直接生成について

nngxAdd3DCommand() を使用することで、アプリケーションで直接生成した 3D コマンドを、gl 関数を呼び出さずに実行させることができるようになります。しかし、直接生成した 3D コマンド(直接生成コマンド)と通常の gl 関数を呼び出して生成した 3D コマンド(通常生成コマンド)を混在させて実行する際には、直接生成コマンドがライブラリのステートを更新しないことに注意しなければなりません。

直接生成コマンドで GPU の設定を変更したあと、同じ設定を変更するような gl 関数を呼び出した場合のステートとの比較で更新なしと判断され、生成されるべき 3D コマンドが生成されずに意図しない描画結果となる可能性があります。また、ステートを更新状態にしたまま直接生成コマンドを実行し、そのあとでバリデートが行われると、意図しない 3D コマンドが生成されて、直接生成コマンドが反映されない可能性もあります。

8.6.1.1. 通常生成コマンドから直接生成コマンドへの移行

通常生成コマンドの実行から直接生成コマンドの実行へと安全に移行するには、更新状態のステートが存在しない状態にすることが肝心です。

ステートの更新状態が解除されるのは glDraw 系関数や nngxValidateState() を呼び出してバリデートしたときです(「8.1.2. コマンド生成」参照)。これまでに gl 関数で行われた設定が GPU に反映されていることを前提にして直接生成コマンドを実行する場合はもちろん、直接生成コマンドを意図したとおりに実行させるためにも、バリデートを行っておくことを推奨します。nngxValidateState() の引数に NN_GX_STATE_ALL を渡し、すべてのステートをバリデートするのが簡単ですが、生成される 3D コマンドが冗長になることを避けるには nngxGetUpdatedState() で取得したステートフラグ

で、更新状態にあるステートだけをバリデートする方法もあります。

8.6.1.2. 直接生成コマンドから通常生成コマンドへの移行

直接生成コマンドの実行から通常生成コマンドの実行へと安全に移行するには、GPU の設定とライブラリのステートが一致するように、直接生成コマンドで更新するはずのステートを更新状態にすることが肝心です。

一番確実な方法は nngxUpdateState() の引数に NN_GX_STATE_ALL を渡し、すべてのステートを更新状態にしてからバリデートすることです。ただし、この方法では不要な 3D コマンドまで生成されてしまいます。

直接生成コマンドで更新したステートとそれに依存するステートを完全に把握しているならば、更新状態にするステートを限定してバリデートし、3D コマンドの生成を必要最小限に抑えることができます。

ここで、nngxSetCommandGenerationMode() の引数に NN_GX_CMDGEN_MODE_UNCONDITIONAL を渡して呼び出し、一部のステートの完全コマンドが生成されるようにすれば、冗長なコマンドの生成を抑えることができます。ただし、nngxUpdateState() で更新状態にしなくてもよくなるのは、「8.1.3. 差分コマンドと完全コマンド」で説明した特定のステートだけです。

8.7. 3D コマンドの編集

コマンドキャッシュによってコマンドリストの再使用を行うことができるようになりましたが、保存したコマンドリストそのままを再使用するだけでは、カメラ位置の変更など、シーン中の変化に対応することができません。この節では、3D コマンドバッファの仕様と GPU のレジスタに書き込まれる情報を説明し、頂点シェーダや予約フラグメントシェーダの設定値などに関する 3D コマンドを編集することでシーン中の変化に対応する方法を紹介します。

8.7.1. 3D コマンドバッファへのアクセスについて

コマンドキャッシュの情報に保存されているのは 3D コマンドバッファの保存開始オフセットです。そのため、3D コマンドを編集する際に 3D コマンドバッファへアクセスするときは、nngxGetCmdlistParameteri() の *pname* に NN_GX_CMDLIST_TOP_BUFADDR を渡して呼び出し、3D コマンドバッファの先頭アドレスを取得する必要があります。

3D コマンドやレジスタに書き込む値のエンディアンがリトルエンディアンであるため、以降で示すビットレイアウトや数値の表記と、メモリ上でのバイトオーダーとの対応に注意してください。

8.7.2. 3D コマンドバッファの仕様

3D コマンドバッファは、GPU のレジスタに書き込まれる 3D コマンド（PICA レジスタ書き込みコマンド）の集合です。3D コマンドは、連続する 64 ビット単位のコマンドの集合で、32 ビットのヘッダと 32 ビットのデータで構成されています。ヘッダの内容によってデータの個数が変動しますが、3D コマンドのアライメントは 64 ビットです。そのため、データの個数によっては最後の 64 ビットの上位 32 ビットのデータが無視されます。

3D コマンドの各ビットには、以下の情報が格納されています。

図 8-5. 3D コマンドのビットレイアウト

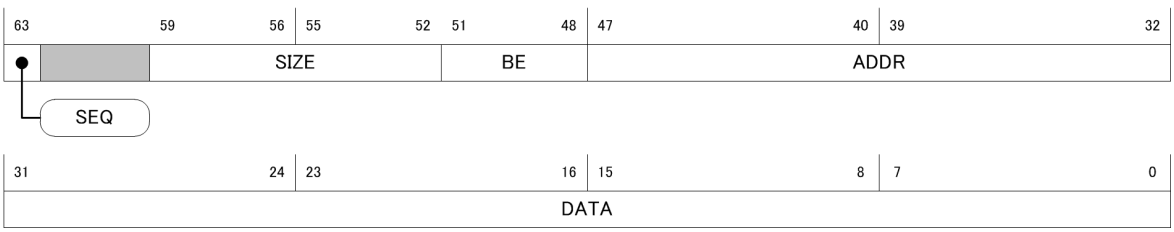


表 8-11. 3D コマンドの各ビットの説明

ビット	名前	説明
[31 : 0]	DATA	レジスタに書き込まれる 32 ビットデータ。
[47 : 32]	ADDR	データを書き込む PICA レジスタのアドレス。
[51 : 48]	BE	バイトイネーブル。32 ビットのデータを 4 つのバイトデータとみなし、対応するビットが 1 であればそのバイトのデータを書き込みます。
[59 : 52]	SIZE	データの個数。実際のデータの個数 – 1 の値が格納されており、0 ならばシングルアクセス、1 以上ならばバーストアクセスです。
[63 : 63]	SEQ	バーストアクセス時のアクセスモード。0 ならば単一レジスタ書き込み、1 ならば連続レジスタ書き込みを行います。

BE が 0 の場合はレジスタにデータが書き込まれませんが、3D コマンドは GPU に読み込まれるため、ダミーコマンドとしてアライメントやタイミング調整などに使用されます。ただし、ダミーコマンドとして、ADDR に指定できるレジスタには範囲がありますので注意してください。

SIZE で指定されたデータの個数によって、シングルアクセスとバーストアクセスの 2 種類のアクセス方法に分かれ、さらにバーストアクセス時には SEQ の指定によって、単一レジスタ書き込みと連続レジスタ書き込みの 2 種類のアクセスモードに分かれます。

8.7.3. シングルアクセス

SIZE に 0 が指定され、データの個数が 1 である場合はシングルアクセスでレジスタにデータを書き込みます。シングルアクセスでは、1 つのレジスタに 1 つのデータを一度だけ書き込みます。

ADDR で指定されたアドレスのレジスタに DATA の内容を書き込みます。その際、BE で 1 となっているビットに対応するバイトだけが書き込まれますので、BE が 0 のときはレジスタにデータが書き込まれません。SEQ の指定は無視されます。

例)

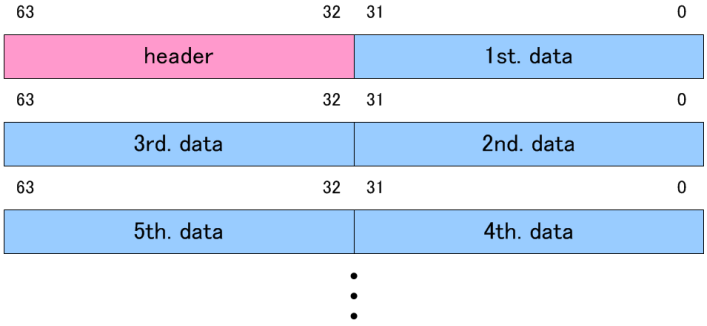
> 3D コマンドが 0x000F0110_12345678 であるとき、SIZE = 0、BE = 0xF、ADDR = 0x0110、DATA = 0x12345678 と解釈されます。SIZE が 0 なのでシングルアクセスとなり、アドレスが 0x0110 のレジスタにデータ 0x12345678 を書き込みます。

8.7.4. バーストアクセス

SIZE に 1 以上が指定され、データの個数が 2 以上である場合はバーストアクセスでレジスタにデータを書き込みます。バーストアクセスでは、1 つまたは複数のレジスタに SIZE + 1 個(最大 256 個)のデータを書き込みます。

DATA に格納されているのは 1 つ目の 32 ビットデータです。2 つ目以降の書き込みデータは後続の 64 ビットデータに 2 つずつ格納され、下位 32 ビットが先のデータ、上位 32 ビットが後のデータとなっています。3D コマンドのアライメントは 64 ビットのため、書き込むデータの個数が偶数の場合は最後の 64 ビットデータの上位 32 ビットのデータが無視されます。

図 8-6. バーストアクセス時のデータの格納順序



BE の指定は書き込まれるデータすべてに共通します。つまり BE が 0 ならば、レジスタにデータは 1 つも書き込まれません。

8.7.4.1. 単一レジスタ書き込み

SEQ に 0 が指定されている場合、アクセスモードが単一レジスタ書き込みとなり、複数のデータを 1 つのレジスタに連続して書き込みます。

ADDR で指定されたアドレスのレジスタにだけ、データが連続して書き込まれます。

例)

3D コマンドが、
0x004F0080_11111111
0x33333333_22222222
0x55555555_44444444
であるとき、SIZE = 4、BE = 0xF、ADDR = 0x0080、DATA = 0x11111111、SEQ = 0 と解釈されます。SIZE が 4、SEQ が 0 なので単一レジスタ書き込みのバーストアクセスとなり、アドレスが 0x0080 のレジスタに 5 個のデータ(0x11111111、0x22222222、0x33333333、0x44444444、0x55555555)を連続して書き込みます。次に実行される 3D コマンドは 0x55555555_44444444 の次の 64 ビットデータです。

8.7.4.2. 連続レジスタ書き込み

SEQ に 1 が指定されている場合、アクセスモードが連続レジスタ書き込みとなり、連続する複数のレジスタにデータを 1 つずつ、一度だけ書き込みます。

ADDR で指定されたアドレスから連続するレジスタ(アドレスは 1 ずつインクリメントされる)に、データが 1 つずつ書き込まれます。

例)

3D コマンドが、
0x805F0280_11111111
0x33333333_22222222
0x55555555_44444444
0x77777777_66666666
であるとき、SIZE = 5、BE = 0xF、ADDR = 0x0280、DATA = 0x11111111、SEQ = 1 と解釈されます。SIZE が 5、SEQ が 1 なので連続レジスタ書き込みのバーストアクセスとなり、アドレスが 0x0280 のレジスタから連続する 6 つのレジスタに、データを書き込みます。アドレスが 0x0280 のレジスタには 0x11111111、アドレスが 0x0281 のレジスタには 0x22222222 が書き込まれ、以降、アドレス 0x0282 には 0x33333333、アドレス 0x0283 には 0x44444444、アドレス 0x0284 には 0x55555555、アドレス 0x0285 には 0x66666666 と続きます。0x77777777 はデータ個数の関係で無視されます。次に実行される 3D コマンドは 0x77777777_66666666 の次の 64 ビットデータです。

8.7.5. 3D コマンドの実行コスト

PICA レジスタへ値を書き込む 3D コマンドは、一部のレジスタへ書き込みを除いて、シングルアクセスやバーストアクセスに関係なく、1 つのレジスタに 1 回書き込みをするコマンドを 1 個処理するには 1 サイクルかかります。

ラスタライゼーションモジュールでは、3D コマンドが入力されるとコマンドを 1 個処理するごとに 1 サイクルのビジーが出力されます。そのため、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、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）への 3D コマンドは 1 個あたり 2 サイクルで処理されることになります。

テクスチャキャッシュのクリアコマンド（レジスタ 0x0080 のビット [16 : 16]）、ポスト頂点キャッシュのクリアコマンド（レジスタ 0x0231）は 1 サイクルで処理されます。ただし、処理自体は 1 サイクルで行われるものの、この 3D コマンドはテクスチャユニットへのコマンドですので、2 サイクルに 1 コマンドしか入力できません。

フレームバッファのキャッシュフラッシュコマンド（レジスタ 0x0111 のビット [0 : 0]）は約 100 サイクル、アーリーデプスバッファのクリアコマンド（レジスタ 0x0063 のビット [0 : 0]）は約 1000 サイクルで処理されます。

また、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各モジュールは、フラグメントデータがモジュール内に残っている状態（フラグメントを処理している状態）で 3D コマンドが入力されるとモジュールごとにパイプラインフラッシュを行います。そのため、描画コマンドの直後の 3D コマンドは、各モジュールのパイプラインフラッシュのコストがかかります。

各モジュールに割り当てられているレジスタの範囲は以下のとおりです。

表 8-12. 各モジュールに割り当てられているレジスタの範囲

モジュール	レジスタの範囲
トライアングルセットアップ	0x0040 から 0x005F
ラスタライゼーションモジュール	0x0060 から 0x006F
テクスチャユニット	0x0080 から 0x00BF
フラグメントライティング	0x0140 から 0x01DF
テクスチャコンバイナ	0x00C0 から 0x00FF
パーフラグメントオペレーションモジュール	0x0100 から 0x013F

補足： 実際にはレジスタの存在しないアドレスも含まれています。

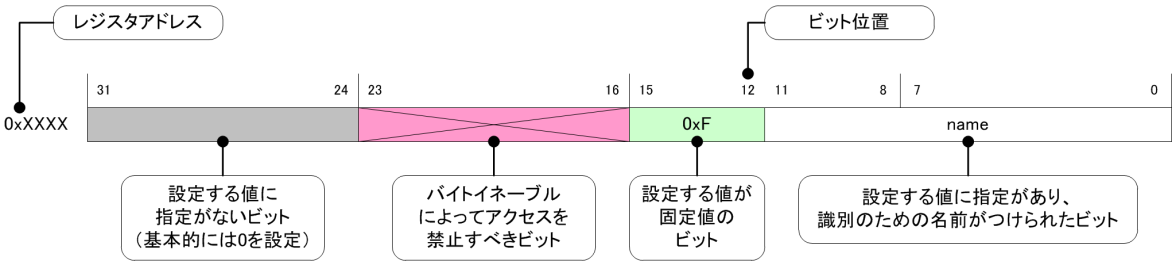
8.8. PICA レジスタ情報

この節では、いくつかの PICA レジスタに関して、そのアドレスと設定方法、値のフォーマットを説明します。これらの情報を元に、該当するレジスタに対する書き込み箇所を 3D コマンドバッファから検索して内容を書き換えることで、その機能に対する設定値を変更することができます。

注意： レジスタにメモリアドレスを設定する場合は、`nn::gx::GetPhysicalAddr()` で仮想アドレスを物理アドレスに変換する必要があります。

レジスタのビットレイアウトは以下の様式で記載しています。

図 8-7. レジスタのビットレイアウトの凡例



8.8.1. 頂点シェーダ設定レジスタ（0x02B0 ～ 0x02DF ほか）

頂点シェーダの開始アドレスや頂点属性、浮動小数点定数レジスタへの設定など、頂点シェーダに関連する設定に使用するレジスタについて説明します。

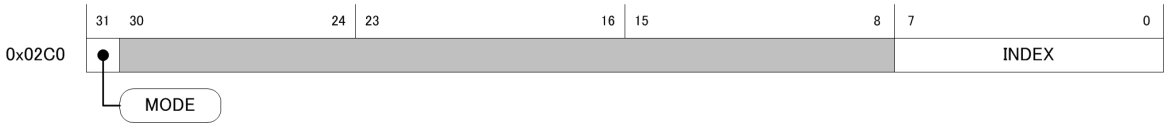
8.8.1.1. 浮動小数点定数レジスタ（0x02C0, 0x02C1 ～ 0x02C8）

頂点シェーダには 96 個の浮動小数点定数レジスタ（シェーダアセンブラでは "c0" から "c95" で表記）があり、それぞれが xyzw の 4 つのコンポーネントで構成されています。設定方法にはシェーダアセンブラの `def` 命令による定数定義と、ユニフォームによる定義の 2 通りあります。前者の場合は GPU の内部形式である 24 ビット浮動小数点数（下位から仮数部 16 ビット、指数部 7 ビット、符号部 1 ビット）で設定し、後者の場合は 32 ビット浮動小数点数（IEEE754 形式の単精度表現）で設定しますが、GPU 内で自動的に 24 ビットに変換されます。

インデックス指定（0x02C0）

どの浮動小数点定数レジスタに対して、どのデータ入力モードでデータを書き込むのかを、レジスタ 0x02C0 で指定します。

図 8-8. 浮動小数点定数レジスタのインデックス指定（0x02C0）のビットレイアウト



INDEX には浮動小数点定数レジスタのインデックスを指定します。0x00 ならば "c0"、0x0A ならば "c10"、0x5F ならば "c95" が指定されます。同時に、MODE に 1 を書き込んだ場合はデータ入力モードが 32 ビット浮動小数点数の入力モードとなり、0 を書き込んだ場合は 24 ビット浮動小数点数の入力モードとなります。

浮動小数点定数レジスタの 4 つのコンポーネント（x, y, z, w）に書き込むデータは、レジスタ 0x02C1 ～ 0x02C8 のいずれかに書き込みます。0x02C1 ～ 0x02C8 のいずれのレジスタに値を書き込んでも処理結果は同じです。極端に言えば、すべてのデータを同じレジスタに書き込んでも、0x02C8 から逆順にデータを書き込んでも同じ結果となります。

浮動小数点定数レジスタに値を設定するには、0x02C0 にインデックスなどを書き込んだあと、0x02C1 ～ 0x02C8 にデータを書き込んでください。

32 ビット浮動小数点数入力モード

32 ビット浮動小数点数入力モードでは、4 つの 32 ビットデータをレジスタ 0x02C1 ～ 0x02C8 のいずれかに書き込むことで 1 つの浮動小数点定数レジスタに値を設定することになります。書き込む順番は、コンポーネントの w、z、y、x の順です。

32 ビットデータを 4 回書き込むとインデックスが自動的に 1 インクリメントされ、指定したインデックスの次のインデックスの浮動小数点定数レジスタに値を設定するようになります。つまり、一度レジスタ 0x02C0 でインデックス 0x0A を指定したあと、0x02C1 ～ 0x02C8 のいずれかに書き込まれたデータの最初の 4 つは "c10" に書き込まれ、その次の 4 つは "c11" に書き込まれます。

例)

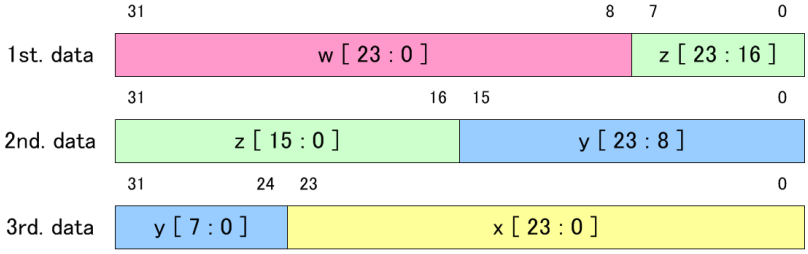
レジスタ 0x02C0 にデータ 0x80000023 を書き込むと、MODE = 1、INDEX = 35 と解釈され、"c35" の浮動小数点定数レジスタへ 32 ビット浮動小数点数入力モードで書き込む準備が整います。このあと、レジスタ 0x02C1 に 0x40800000、0x02C2 に 0x40400000、0x02C3 に 0x40000000、0x02C4 に 0x3F800000 の組み合わせで 2 回(計 8 回)書き込んだ場合、"c35.xyzw" と "c36.xyzw" には { 1.0f, 2.0f, 3.0f, 4.0f } が設定されます。

上記の設定を 3D コマンドでは、
0x000F02C0_80000023
0x803F02C1_40800000 0x40000000_40400000 0x00000000_3F800000
0x803F02C1_40800000 0x40000000_40400000 0x00000000_3F800000
などのように表すことができます。

24 ビット浮動小数点数入力モード

24 ビット浮動小数点数入力モードでは、4 つの 24 ビット浮動小数点数を 3 つの 32 ビットデータにパックしたものをレジスタ 0x02C1 ～ 0x02C8 のいずれかに書き込むことで 1 つの浮動小数点定数レジスタに値を設定することになります。32 ビットデータへのパックの順番は、コンポーネントの w、z、y、x の順です。4 つの 24 ビットデータを、どのようにして 3 つの 32 ビットデータにパックするのかは下図を参照してください。32 ビットの浮動小数点数から 24 ビットの浮動小数点数への変換方法については「8.9.1. 24 ビット浮動小数点数への変換」を参照してください。

図 8-9. 24 ビット浮動小数点数入力モードでのデータの配置



32 ビットデータを 3 回書き込むとインデックスが自動的に 1 インクリメントされ、指定したインデックスの次のインデックスの浮動小数点定数レジスタに値を設定するようになります。つまり、一度レジスタ 0x02C0 でインデックス 0x0A を指定したあと、0x02C1 ～ 0x02C8 のいずれかに書き込まれたデータの最初の 3 つは "c10" に書き込まれ、その次の 3 つは "c11" に書き込まれます。

例)

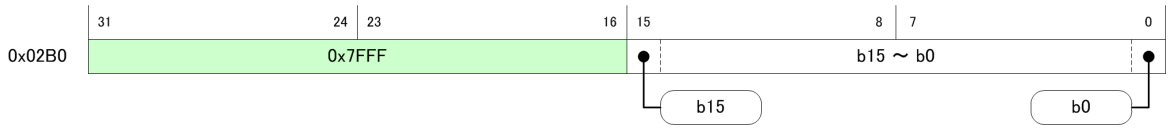
レジスタ 0x02C0 にデータ 0x00000023 を書き込むと、MODE = 0、INDEX = 35 と解釈され、"c35" の浮動小数点定数レジスタへ 24 ビット浮動小数点数入力モードで書き込む準備が整います。このあと、レジスタ 0x02C1 に 0x41000040、0x02C2 に 0x80004000、0x02C3 に 0x003F0000 の組み合わせで 2 回(計 6 回)書き込んだ場合、"c35.xyzw" と "c36.xyzw" には { 1.0f, 2.0f, 3.0f, 4.0f } が設定されます。

上記の設定を 3D コマンドでは、
0x000F02C0_00000023
0x802F02C1_41000040 0x003F0000_80004000
0x802F02C1_41000040 0x003F0000_80004000
などのように表すことができます。

8.8.1.2. ブールレジスタ(0x02B0)

頂点シェーダには 16 個のブールレジスタ(シェーダアセンブラでは "b0" から "b15" で表記)があります。設定方法にはシェーダアセンブラの `defb` 命令による定数定義と、ユニフォームによる定義の 2 通りあります。

図 8-10. ブールレジスタ(0x02B0)のビットレイアウト



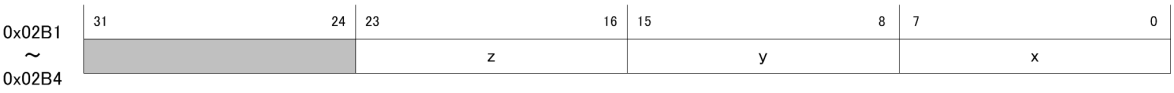
レジスタ 0x02B0 のビット [15 : 0] の各ビットが、頂点シェーダのブールレジスタの 1 つ 1 つに対応しています。ビット [0] が "b0"、ビット [1] が "b1"、・・・、ビット [15] が "b15" に対応し、1 が書き込まれると TRUE、0 が書き込まれると FALSE を意味します。

ジオメトリシェーダを使用する場合、ブールレジスタの 15 番(b15)がジオメトリシェーダに予約されていることに注意してください。

8.8.1.3. 整数レジスタ(0x02B1 ～ 0x02B4)

頂点シェーダには 4 個の整数レジスタ(シェーダアセンブラでは "i0" から "i3" で表記)があり、それぞれが xyz の 3 つのコンポーネントで構成されています。設定方法にはシェーダアセンブラの `defi` 命令による定数定義と、ユニフォームによる定義の 2 通りあります。

図 8-11. 整数レジスタ(0x02B1 ～ 0x02B4)のビットレイアウト

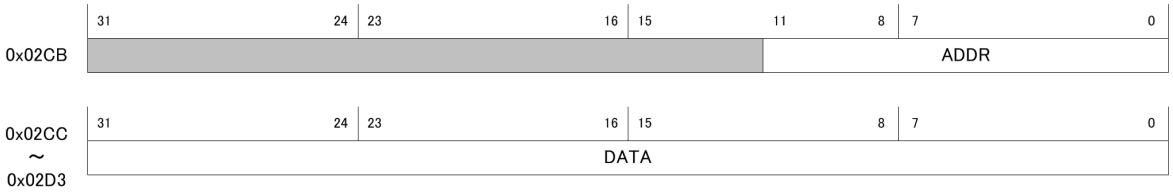


レジスタ 0x02B1 が "i0"、0x02B2 が "i1"、0x02B3 が "i2"、0x02B4 が "i3" にそれぞれ対応しています。整数レジスタには x、y、z の 3 コンポーネントを下位から 8 ビットずつ格納します。y および z への負数の設定は 2 の補数表現で行います。

8.8.1.4. プログラムコード設定レジスタ(0x02BF, 0x02CB ～ 0x02D3, 0x02D5 ～ 0x02DD)

頂点シェーダで実行するプログラムコードのロードに使用するレジスタには、プログラムをロードするアドレスの指定するレジスタと、プログラムデータを書き込むための複数のレジスタが存在します。

図 8-12. プログラムコードのロードレジスタ(0x02CB ～ 0x02D3)のビットレイアウト



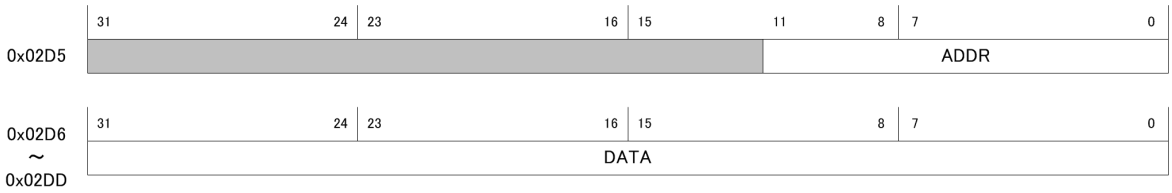
レジスタ 0x02CB には、頂点シェーダのプログラムコードをロードするアドレスを ADDR に設定します。レジスタ 0x02CC ～ 0x02D3 には、ロードするプログラムコードのデータを書き込みます。

プログラムコードのロードアドレスをレジスタ 0x02CB に設定してから、0x02CC ～ 0x02D3 のいずれかのレジスタにデータを書き込みます。頂点シェーダプログラムの 1 命令は 32 ビットですので、1 データの書き込みが 1 命令に対応し、1 つ書き込むごとにロードアドレスが 1 インクリメントされます。いずれのレジスタに書き込んでも処理結果は変わりません。

プログラムコードを更新したあとは、プログラムの更新完了を GPU に通知するためにレジスタ 0x02BF の任意のビットに 1 を書き込む必要があります。

上記のプログラムコードに加え、Swizzle パターンデータをロードする必要があります。Swizzle パターンを設定するレジスタを以下に示します。

図 8-13. Swizzle パターンのロードレジスタ(0x02D5 ～ 0x02DD)のビットレイアウト



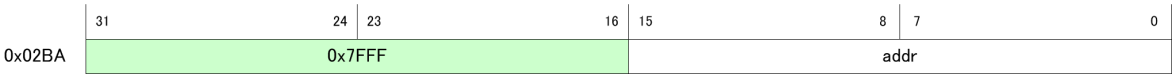
レジスタ 0x02D5 には、Swizzle パターンをロードするアドレスを ADDR に設定します。レジスタ 0x02D6 ～ 0x02DD には、ロードする Swizzle パターンのデータを書き込みます。

Swizzle パターンのロードアドレスをレジスタ 0x02D5 に設定してから、0x02D6 ～ 0x02DD のいずれかのレジスタにデータを書き込みます。データを 1 つ書き込むごとにロードアドレスが 1 インクリメントされます。いずれのレジスタに書き込んでも処理結果は変わりません。

8.8.1.5. 開始アドレス設定レジスタ(0x02BA)

シェーダアセンブラで定義した main ラベルのアドレスが頂点シェーダの開始アドレスです。

図 8-14. 開始アドレス設定レジスタ(0x02BA)のビットレイアウト

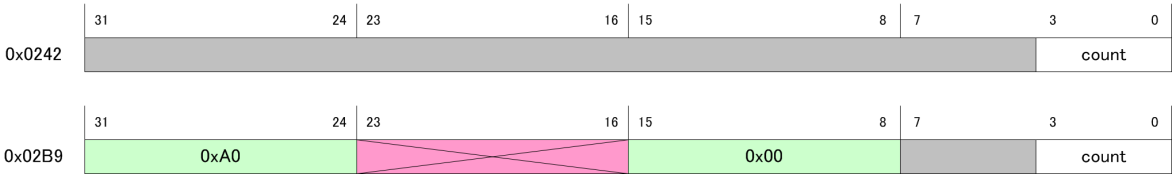


レジスタ 0x02BA のビット [15 : 0] に頂点シェーダの開始アドレスを設定します。

8.8.1.6. 頂点属性入力数設定レジスタ(0x0242, 0x02B9)

頂点シェーダに入力される頂点属性数が設定されるレジスタは複数あり、それぞれに同じ値が設定されます。

図 8-15. 頂点属性入力数設定レジスタ(0x0242, 0x02B9)のビットレイアウト

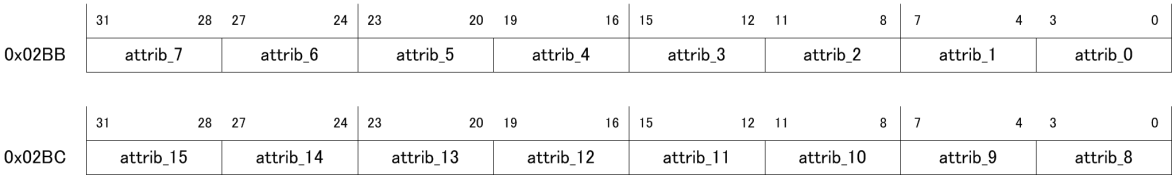


count には(入力する頂点属性数 - 1)を設定します。入力する頂点属性数は、頂点バッファを使用する場合(ロードアレイを使用して頂点データをロードする場合)は 12 個まで、頂点バッファを使用しない場合(コマンドバッファ経由で頂点データをロードする場合)は 16 個までです。

8.8.1.7. 入力レジスタのマッピング設定レジスタ(0x02BB, 0x02BC)

頂点シェーダに入力される頂点属性データと入力レジスタとのマッピングが設定されるレジスタを以下に示します。

図 8-16. 入力レジスタのマッピング設定レジスタ(0x02BB, 0x02BC)のビットレイアウト



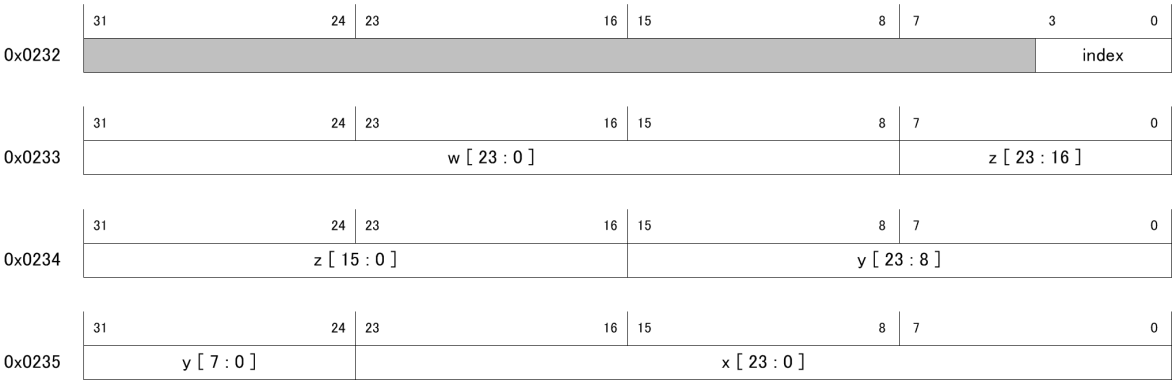
attrib_0 ～ attrib_15 には、頂点シェーダに入力される頂点属性データが、どの入力レジスタに格納されるのかを入力レジスタのインデックス番号("v0" が 0x0、"v1" が 0x1、"v15" が 0xF)で設定します。頂点シェーダに入力される頂点属性の順番

は glBindAttribLocation() で指定した *index* とは対応せず、「頂点属性アレイ設定レジスタ(0x0200 ～ 0x0227)」で説明されている内部頂点属性の番号と対応しています。

8.8.1.8. 固定頂点属性値設定レジスタ(0x0232 ～ 0x0235)

glVertexAttrib4f() などで設定される固定頂点属性値は、24 ビット浮動小数点数に変換されて GPU へ転送されます。固定頂点属性値は、以下のレジスタへの設定で GPU に転送されます。

図 8-17. 固定頂点属性値設定レジスタ(0x0232 ～ 0x0235)のビットレイアウト



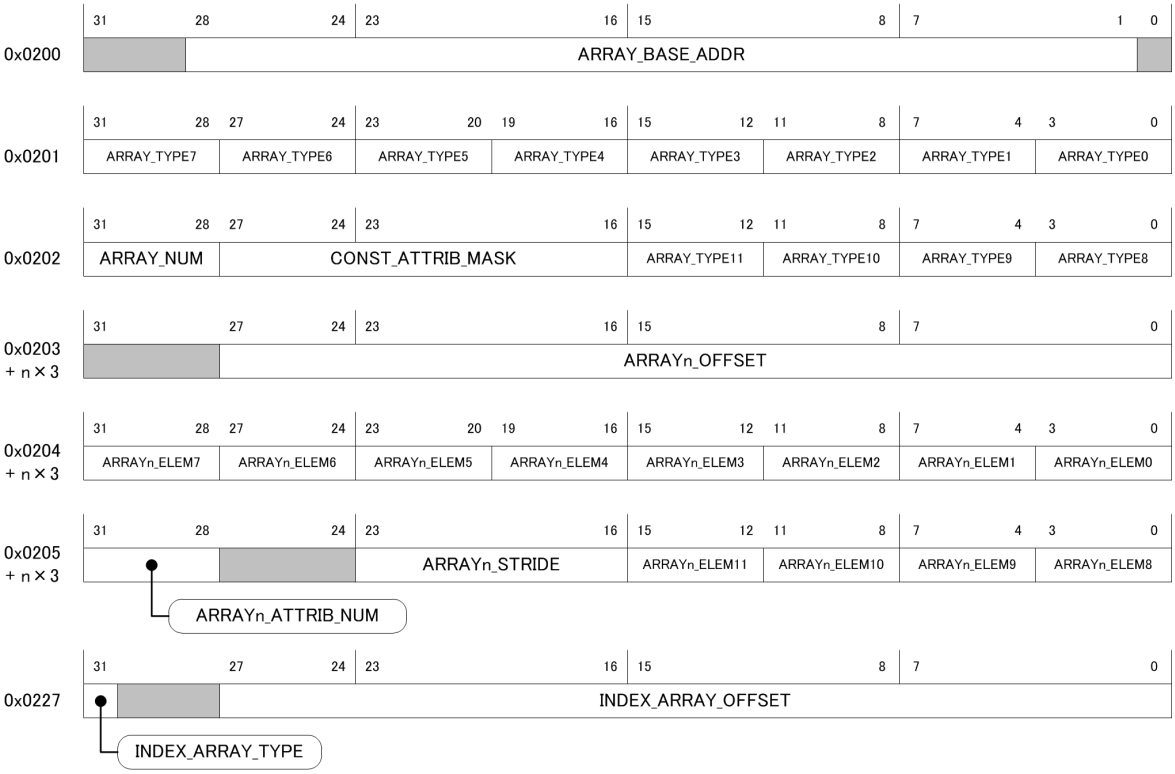
まず、レジスタ 0x0232 のビット [3 : 0] に固定頂点属性値の内部頂点属性の番号を書き込みます。次に、固定頂点属性値を 24 ビット浮動小数点数に変換した 3 つの 32 ビットデータを、レジスタ 0x0233、0x0234、0x0235 の順に書き込みます。24 ビット浮動小数点数に変換した 3 つの 32 ビットデータは、「24 ビット浮動小数点数入力モード」にあるデータと同じ方法で作成します。

「頂点属性アレイ設定レジスタ(0x0200 ～ 0x0227)」で頂点アレイの有効・無効を変更した場合、設定されていた固定頂点属性値は無効となりますので、固定頂点属性値を再度設定する必要があります。また、1 つも頂点アレイを使用せず、すべての頂点属性を固定頂点属性として使用することは GPU の仕様上できません。固定頂点属性を使用する場合は必ず、1 つ以上の頂点アレイを使用してください。

8.8.1.9. 頂点属性アレイ設定レジスタ(0x0200 ～ 0x0227)

頂点バッファ使用時の頂点属性アレイを設定するレジスタには複数のレジスタがあります。これらのレジスタに対する設定コマンドはステートフラグ NN_GX_STATE_VERTEX のバリデーションで生成されます。

図 8-18. 頂点属性アレイ設定レジスタ(0x0200 ほか)のビットレイアウト



これらのレジスタで行われる設定には、ベースアドレス、内部頂点属性の型、固定頂点属性のマスク、頂点属性の総数、各ロードアレイのバイトオフセット、ロードアレイの要素の情報、ロードアレイの要素数、ロードアレイのバイト数、インデックスアレイのオフセットがあります。

ベースアドレス

レジスタ 0x0200 の ARRAY_BASE_ADDR には、ベースアドレスとして物理アドレスを 16 で割った値を設定します。全頂点アレイのアドレスと、頂点インデックスアレイのアドレスは、ベースアドレス + オフセットの形式で設定されます。頂点アレイとインデックスアレイのアドレス範囲があらかじめ確定している場合は、頂点アレイの組み合わせごとに設定しなおす必要はありません。

内部頂点属性の型

内部頂点属性とは、頂点アレイをロードするために GPU が内部で決定する頂点属性番号のことです。
`glEnableVertexAttribArray()` の *index* で指定された頂点属性番号(以降、GL 頂点属性番号と呼びます)とは異なりますが、内部頂点属性と GL 頂点属性番号との間では 1 対 1 の関係が成り立っています。

`glEnableVertexAttribArray()` によって有効とされている頂点アレイが、内部頂点属性 0 から昇順に隙間なく割り当てられますが、GL 頂点属性番号 0 が内部頂点属性 0 であるとは限りません。例えば、GL 頂点属性番号 0 と 3 の頂点アレイが有効になっている場合、それらは内部頂点属性 0 と 1 に割り当てられることも、1 と 0 に割り当てられることもあります。内部頂点属性の割り当て方法はドライバの実装に依存していますので、将来変更される場合があります。

現実装では、頂点アレイのアドレスで昇順にソートし、先頭の GL 頂点属性から順に内部頂点属性 0 から割り当てています。

頂点属性データは、内部頂点属性の順に頂点シェーダへ入力されます。

レジスタ 0x0201 から 0x0202 の ARRAY_TYPEn(n = 0 ～ 11)には、(n + 1)番目の内部頂点属性の型を設定します。内部頂点属性の型としてレジスタに設定する値は `glVertexAttribPointer()` で指定された *size* と *type* の組み合わせで決定されます。組み合わせとレジスタに設定する値の対応は以下のようになっています。

表 8-13. size と type の組み合わせとレジスタに設定する値

size	type	設定値
1	GL_BYTE	0x0
1	GL_UNSIGNED_BYTE	0x1
1	GL_SHORT	0x2
1	GL_FLOAT	0x3
2	GL_BYTE	0x4
2	GL_UNSIGNED_BYTE	0x5
2	GL_SHORT	0x6
2	GL_FLOAT	0x7
3	GL_BYTE	0x8
3	GL_UNSIGNED_BYTE	0x9
3	GL_SHORT	0xA
3	GL_FLOAT	0xB
4	GL_BYTE	0xC
4	GL_UNSIGNED_BYTE	0xD
4	GL_SHORT	0xE
4	GL_FLOAT	0xF

固定頂点属性マスク

頂点シェーダのシェーダアセンブラで、#pragma bind_symbol で定義されている数の頂点属性が有効となりますが、そのうち頂点アレイが無効に設定されているもの(glDisableVertexAttribArray() が呼び出されている頂点属性や glEnableVettexAttribArray() が呼び出されていない頂点属性)に対しては固定頂点属性が使用されます。

固定頂点属性も頂点アレイと同様に内部頂点属性に割り当てられます。固定頂点属性は、頂点アレイが割り当てられた番号に続いて、昇順に隙間なく割り当てられます。

割り当てられた内部頂点属性のマスクを、レジスタ 0x0202 の CONST_ATTRIB_MASK に設定します。下位のビットから順に内部頂点属性 0 から内部頂点属性 11 に対応しており、固定頂点属性に割り当てられているものに対応するビットには 1 がセットされます。

このレジスタ設定で頂点アレイの有効・無効を変更した場合、設定されていた固定頂点属性値は無効となりますので、固定頂点属性値を再度設定する必要があります。また、1 つも頂点アレイを使用せず、すべての頂点属性を固定頂点属性として使用することは GPU の仕様上できません。固定頂点属性を使用する場合は必ず、1 つ以上の頂点アレイを使用してください。

頂点属性数

レジスタ 0x0202 の ARRAY_NUM には、頂点アレイを使用する頂点属性と固定頂点属性の総数 – 1 を設定します。

ロードアレイ

GPU は頂点属性データをロードするために、頂点属性アレイをデータアレイ単位で管理しています。このデータアレイを特にロードアレイと呼び、GPU は 12 個のロードアレイからデータをロードします。

12 個のロードアレイはそれぞれ 12 個までの要素によって構成されています。ロードアレイの要素とは、そのロードアレイを構成する頂点アレイのデータ、または 4 バイト単位のパディングのことです。基本的に、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定義した場合（これをインターリーブドアレイと呼びます）、その 1 つのインターリーブドアレイが 1 つのロードアレイに対応しています。逆に、1 つの頂点属性の配列として頂点データを定義した場合（これを独立アレイと呼びます）、その 1 つの頂点属性が 1 つのロードアレイに対応しています。

ロードアレイは先頭のロードアレイ（ロードアレイ 0）から昇順に詰めて使用しなければなりません。例えば、ロードアレイ 1 とロードアレイ 4 のように 0 から始まっていない、または連続ではない組み合わせでは使用できません。

頂点属性アレイの実際のアドレスは、`glBufferData()` で確保されたアドレスに `glVertexAttribPointer()` の *`ptr`* で指定されたオフセットを加えた値です。レジスタに設定する場合、この実際のアドレスが（ベースアドレス× 16 ＋ロードアレイのバイトオフセット）となるように設定します。

レジスタ（0x0203 + n × 3）の ARRAYn_OFFSET（n = 0 ～ 11）には、（n + 1）番目のロードアレイのバイトオフセットが設定されます。使用するロードアレイの個数が少ないほど GPU のパフォーマンスが向上するため、ドライバによって、少ない個数のロードアレイでデータがロードできるような設定がなされています。

レジスタ（0x0204 + n × 3）から（0x0205 + n × 3）の ARRAYn_ELEMi（n = 0 ～ 11、i = 0 ～ 11）には、（n + 1）番目のロードアレイを構成する（i + 1）番目の要素を先頭から順に設定します。要素には使用する内部頂点属性またはパディングを設定し、レジスタに設定する値と要素の対応は以下のようになっています。

表 8-14. レジスタに設定する値と要素の対応

設定値	要素
0x0	内部頂点属性 0
0x1	内部頂点属性 1
0x2	内部頂点属性 2
0x3	内部頂点属性 3
0x4	内部頂点属性 4
0x5	内部頂点属性 5
0x6	内部頂点属性 6
0x7	内部頂点属性 7
0x8	内部頂点属性 8
0x9	内部頂点属性 9
0xA	内部頂点属性 10
0xB	内部頂点属性 11
0xC	4 バイトのパディング
0xD	8 バイトのパディング
0xE	12 バイトのパディング
0xF	16 バイトのパディング

例えば、レジスタ 0x0204 の ARRAY0_ELEM0 に 0x0 を設定した場合、1 番目のロードアレイの第 1 番目の要素は内部頂点属性 0 となり、ロードアレイのデータ構造の先頭には レジスタ 0x0201 の ARRAY_TYPE0 に設定された内部頂点属性 0

の型のデータが配置されることになります。

レジスタ(0x0205 + n × 3)の ARRAYn_STRIDE には、(n + 1)番目のロードアレイの 1 頂点あたりのバイト数を設定します。複数の型の要素を含むロードアレイには自動でパディングが含まれる場合があり、ARRAYn_STRIDE にはそのパディングを含めたバイト数を設定しなければなりません。設定値とロードアレイの要素の合計サイズが一致しない場合の動作は不定です。レジスタ(0x0205 + n × 3)の ARRAYn_ATTRIB_NUM(n = 0 ～ 11)には、(n + 1)番目のロードアレイの属性数を設定します。複数の頂点属性アレイをインターリーブドアレイとして配置している場合など、1 つのロードアレイに対して複数の頂点属性アレイが含まれる場合があります。(n + 1)番目のロードアレイの属性数に設定されている値は、そのロードアレイに含まれる頂点属性アレイの個数とは一致しません。0 を設定した場合、そのロードアレイは使用されません。

レジスタ 0x0227 の INDEX_ARRAY_OFFSET には、インデックスアレイのバイトオフセットを設定します。レジスタ 0x0227 の INDEX_ARRAY_TYPE には頂点インデックスの型を設定します。`glDrawElements()` の引数 *type* が `GL_UNSIGNED_SHORT` の場合は 1 を、`GL_UNSIGNED_BYTE` の場合は 0 を設定します。`glDrawArrays()` の場合は常に 1 を設定します。

例 1)インターリーブドアレイの例

コード 8-15. インターリーブドアレイの例(構造体)

```
struct vertex_t
{
    float position[3];
    float color[4];
    float texcoord[2];
} vertex[NUM_VERTEX];
```

上に示した構造体により頂点データを構成した場合、頂点アレイの設定は以下のようになります。

コード 8-16. インターリーブドアレイの例(頂点アレイの設定)

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(struct vertex_t), 0);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, sizeof(struct vertex_t), 12);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(struct vertex_t), 28);
```

この場合、GL 頂点属性 0、1、2 の 3 つが 1 つのロードアレイの要素となります。また、使用されるすべての頂点属性が上記の 3 つと固定頂点属性 1 つであるとした場合、上記の頂点属性 0、1、2 は内部頂点属性 0、1、2 に対応し、固定頂点属性は内部頂点属性 3 に対応します。よって、関連するレジスタ設定は以下のようになります。

0x0201：0x000007FB // 内部頂点属性の型は 0:FLOAT_VEC3, 1:FLOAT_VEC4, 2:FLOAT_VEC2
0x0202：0x30080000 // 頂点属性数は合計 4、内部頂点属性 3 が固定頂点属性
0x0203：0x00000000 // ロードアレイは 1 つのみ使用のため、ベースアドレスに実アドレスを設定
0x0204：0x00000210 // ロードアレイ 0 の要素は頂点内部属性 0、1、2
0x0205：0x30240000 // ロードアレイ 0 の 1 頂点辺りバイト数は float × 9 で 36 バイト、要素数は 3
0x0206～0x0226：0x00000000 // 他のロードアレイは使用しない

例 2)独立アレイの例

コード 8-17. 独立アレイの例(構造体)

```
#define NUM_VERTEX (3)
struct attribute0_t
{
    float position[3];
```

```
} attirbute0[NUM_VERTEX];
struct attribute1_t
{
    float color[4];
} attribute1[NUM_VERTEX];
struct attribute2_t
{
    float tex[2];
} attribute2[NUM_VERTEX];
```

上に示した構造体により頂点データを構成した場合、頂点アレイの設定は以下のようになります。頂点バッファは 1 つのオブジェクトで共有し、順番にデータを配置していると仮定しています。

コード 8-18. 独立アレイの例（頂点アレイの設定）

```
glBindBuffer(GL_ARRAY_BUFFER, 1);
glBufferData(GL_ARRAY_BUFFER,
        sizeof(attribute0)+sizeof(attribute1)+sizeof(attribute2),0,GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(attribute0), attribute0);
glBufferSubData(GL_ARRAY_BUFFER,
        sizeof(attribute0), sizeof(attribute1), attribute1);
glBufferSubData(GL_ARRAY_BUFFER,
        sizeof(attribute0)+sizeof(attribute1), sizeof(attribute2), attribute2);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0,
        (GLvoid*)(sizeof(attribute0)));
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0,
        (GLvoid*)(sizeof(attribute0)+sizeof(attribute1)));
```

この場合、GL 頂点属性 0、1、2 はそれぞれ別のロードアレイの要素となり、それぞれ内部頂点属性 0、1、2 に対応します。よって、関連するレジスタ設定は以下のようになります。

0x0201：0x000007FB // 内部頂点属性の型は 0:FLOAT_VEC3, 1:FLOAT_VEC4, 2:FLOAT_VEC2
0x0202：0x20000000 // 頂点属性数は合計 3、固定頂点属性はなし
0x0203：0x00000000 // ロードアレイ 0 は先頭に配置される
0x0204：0x00000000 // ロードアレイ 0 の要素は内部頂点属性 0 の 1 つ
0x0205：0x100C0000 // ロードアレイ 0 の 1 頂点辺りバイト数は float × 3 で 12 バイト、要素数は 1
0x0206：0x00000024 // ロードアレイ 1 のオフセットは sizeof(attribute0)
0x0207：0x00000001 // ロードアレイ 1 の要素は内部頂点属性 1 の 1 つ
0x0208：0x10100000 // ロードアレイ 1 の 1 頂点辺りバイト数は float × 4 で 16 バイト、要素数は 1
0x0209：0x00000054 // ロードアレイ 2 のオフセットは sizeof(attribute0)+sizeof(attribute1)
0x020A：0x00000002 // ロードアレイ 2 の要素は内部頂点属性 2 の 1 つ
0x020B：0x10080000 // ロードアレイ 2 の 1 頂点辺りバイト数は float × 2 で 8 バイト、要素数は 1
0x020C～0x0226：0x00000000 // 他のロードアレイは使用しない

ロードアレイのパディング要素と自動パディングについて

レジスタ（0x0204 + n × 3）から（0x0205 + n × 3）の ARRAYn_ELEMi（n = 0 ～ 11、i = 0 ～ 11）に、0xC ～ 0xF を設定した場合の要素はパディングと設定されています。これは、ロードアレイが頂点属性として使用しない領域を含む場合に使用します。

例えば、以下のような構造体の頂点データを作成したとします。

コード 8-19. パディングを使用する構造体の例

```
struct vertex_t
{
    float position[3];
    float color[4];
    float texcoord[2];
} vertex[NUM_VERTEX];
```

上に示した構造体の texcoord を頂点属性として使用しない場合を考えます。1 頂点分のサイズは float × 9 ですが、最後の float × 2 は使用しません。この頂点データに対応するロードアレイは、1 番目の要素、2 番目の要素には内部頂点属性を指定しますが、3 番目の要素としては 0xD(8 バイトのパディング)を指定することになります。

1 番目の要素にパディングは指定できません。指定した場合の動作は不定です。ロードアレイのバイトオフセットを調整して、1 番目の要素にパディングが来ないように設定してください。

また、1 つのロードアレイが、複数の異なるデータ型(GL_FLOAT、GL_SHORT、GL_BYTE、GL_UNSIGNED_BYTE)の頂点属性を要素としている場合、ロードアレイの要素には指定されていなくても 4 バイト未満のパディングが自動で挿入される場合があります。ロードアレイを構成する各要素は、4 バイトの型(要素とする内部頂点属性の型が GL_FLOAT である、またはパディングである)、2 バイトの型(要素とする内部頂点属性の型が GL_SHORT である)、1 バイトの型(要素とする内部頂点属性の型が GL_BYTE、GL_UNSIGNED_BYTE である)のいずれかです。ロードアレイの各要素は、その要素自身の型のサイズでアライメントされるように自動でパディングが挿入されます。また、そのロードアレイが含む要素のうち、最も大きい型のサイズでアライメントされるように各頂点データの最後に自動でパディングが挿入されます。

例えば、以下のような構造体の頂点データを作成したとします。

コード 8-20. 自動でパディングが挿入される例

```
struct vertex_t
{
    GLfloat position[3];
    GLubyte color[3];
    GLfloat texcoord[2];
    GLubyte param;
} vertex[NUM_VERTEX];
```

ロードアレイは position、color、texcoord、param の 4 つの頂点属性を要素とするとして考えます。上記の color は 3 バイトですが、直後にある GLfloat 型のデータ texcoord は 4 バイトにアライメントされて配置されます。つまり、color の直後に 1 バイトのパディングが自動で挿入されることになります。

また、ロードアレイ内で最大サイズの要素は GLfloat 型ですので、4 バイトでアライメントされるように各頂点データの最後に自動でパディングが挿入されます。つまり、param の直後に 3 バイトのパディングが自動で挿入されることになります。

ロードアレイの設定とパフォーマンス

頂点データのロードのパフォーマンスは、使用するロードアレイの個数やサイズ、含まれている要素の型などに依存します。

GPU はロードアレイ単位でメモリにアクセスしますが、キャッシュが存在しないため、複数のロードアレイが同じアドレスからロードするコストと異なるアドレスからロードするコストは同等です。

頂点シェーダの複数の入力レジスタに同じ頂点アレイをロードする場合、複数のロードアレイで同じアドレスからロードするよりも、データサイズが大きくなるものの、その頂点アレイを複製して作られたインターリーブドアレイからロードする方が実行時のパフォーマンスが高くなる可能性があります。

同じサイズの頂点データをロードする場合でも、1 つのロードアレイでロードした方が、複数のロードアレイでロードするよりも

パフォーマンスが高くなります。このパフォーマンスの違いは、頂点インデックスが最適化されていて連続するような状況では小さくなり、ほかのモジュールによるデバイスメモリへのアクセス状況からも影響を受けます。例えば、デバイスメモリへのアクセスの競合を考えずに、同じ 6 個の GLfloat 型の頂点データを 3 個ずつ 2 つのロードアレイでロードすると、1 つのロードアレイでロードする場合の処理時間の 1.3～2 倍程度かかります。ただし、頂点アレイが VRAM に配置されている場合ではパフォーマンスに違いがなくなります。

ロードアレイの制限

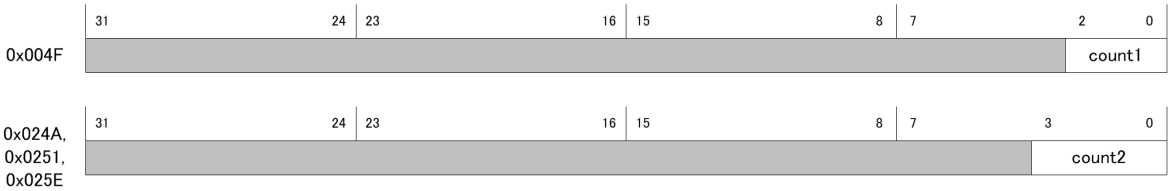
glDrawElements() で描画する(レジスタ 0x022F への書き込みで描画を開始する)場合、使用できるロードアレイは最大 11 個です。頂点属性を 12 個使用し、glDrawElements() で描画する場合は、少なくとも 1 個の頂点属性を固定頂点属性にする、または少なくとも 2 個の頂点属性をインターリーブドアレイとし、使用するロードアレイが 11 個以下になるようにしてください。ロードアレイを 12 個使用して glDrawElements() による描画を開始した場合、GPU がハングアップすることがあります。このほかにもロードアレイの設定が適切ではない場合は、GPU がハングアップすることがあります。

ロードアレイの不正な設定が原因でハングアップした場合、nngxGetCmdlistParameteri() で NN_GX_CMDLIST_HW_STATE により取得される値は、ビット 8 にのみ 1 がセットされた状態になります。

8.8.1.10. 出力レジスタ使用数設定レジスタ(0x004F, 0x024A, 0x0251, 0x025E)

頂点シェーダから出力される頂点属性数を設定するレジスタは複数あり、一部を除いて同じ値を設定します。

図 8-19. 出力レジスタ使用数設定レジスタ(0x004F, 0x024A, 0x0251, 0x025E)のビットレイアウト



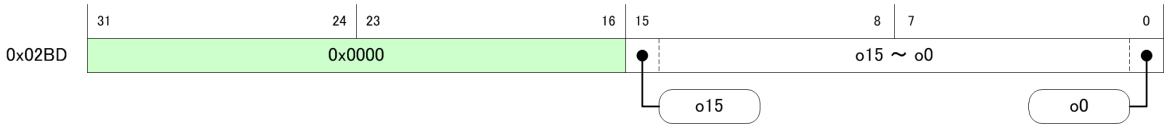
count1(レジスタ 0x004F)には使用する出力レジスタの個数そのままを、count2(レジスタ 0x024A, 0x0251, 0x025E すべて)には(使用する出力レジスタの個数 - 1)をそれぞれ設定します。count1 のみ、値とビット幅が異なる点に注意してください。

出力レジスタの個数とは頂点シェーダアセンブラに #pragma output_map で定義された個数ですが、使用している出力レジスタの数ですので、複数の頂点属性を 1 つの出力レジスタにパックしている場合は 1 とカウントします。

8.8.1.11. 出力レジスタのマスク設定レジスタ(0x02BD)

頂点シェーダで書き込む出力レジスタのマスクを設定するレジスタには、16 個ある出力レジスタの、どのレジスタに書き込みが行われるかをビットで指定します。

図 8-20. 出力レジスタのマスク設定レジスタ(0x02BD)のビットレイアウト

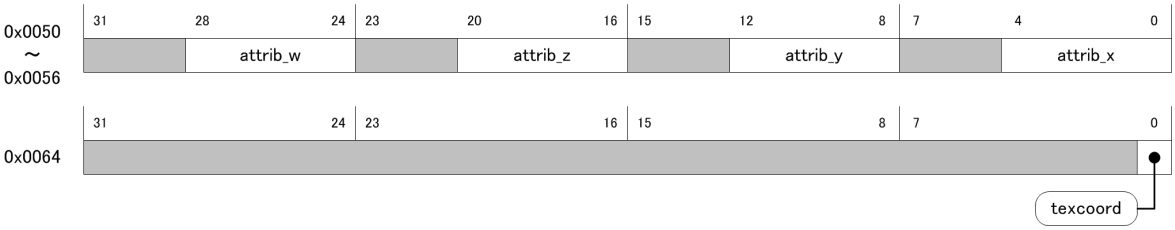


レジスタ 0x02BD のビット [15 : 0] の各ビットが、出力レジスタの 1 つ 1 つに対応しています。ビット [0] が "o0"、ビット [1] が "o1"、・・・、ビット [15] が "o15" に対応し、#pragma output_map で定義されている出力レジスタに対応するビットには 1 を書き込みます。定義されていない出力レジスタに対応するビットには 0 を書き込みます。

8.8.1.12. 出力レジスタの属性設定レジスタ(0x0050 ～ 0x0056, 0x0064)

頂点シェーダから頂点属性を出力するのに使用可能な出力レジスタの数は 7 です。出力レジスタの各コンポーネントで出力する属性を設定するレジスタは複数あり、使用する出力レジスタの若い番号から順に設定します。

図 8-21. 出力属性の設定レジスタ(0x0050 ～ 0x0056, 0x0064)のビットレイアウト



ビットレイアウト中の名前は以下のように出力属性の設定に対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

表 8-15. 名前と設定の対応(出力属性の設定レジスタ)

名前	ビット数	説明
attrib_x attrib_y attrib_z attrib_w	5	上から、出力レジスタの x、y、z、w コンポーネントに設定する頂点属性です。 0x00：頂点座標の x 成分 0x01：頂点座標の y 成分 0x02：頂点座標の z 成分 0x03：頂点座標の w 成分 0x04：クォータニオンの x 成分 0x05：クォータニオンの y 成分 0x06：クォータニオンの z 成分 0x07：クォータニオンの w 成分 0x08：頂点カラーの赤成分 0x09：頂点カラーの緑成分 0x0A：頂点カラーの青成分 0x0B：頂点カラーのアルファ成分 0x0C：テクスチャ座標 0 の u 成分 0x0D：テクスチャ座標 0 の v 成分 0x0E：テクスチャ座標 1 の u 成分 0x0F：テクスチャ座標 1 の v 成分 0x10：テクスチャ座標 0 の w 成分 0x12：ビューベクタの x 成分 0x13：ビューベクタの y 成分 0x14：ビューベクタの z 成分 0x16：テクスチャ座標 2 の u 成分 0x17：テクスチャ座標 2 の v成分 0x1F：無効
texcoord	1	頂点シェーダから出力される頂点属性にテクスチャ座標が含まれているかどうかを設定します。 0x0：テクスチャ座標を出力しない 0x1：テクスチャ座標を出力する

例)頂点シェーダで以下のように定義した場合を例に、レジスタに設定する値を説明します。

コード 8-21. 出力属性の設定例

```
#pragma output_map(position, o0)
#pragma output_map(color, o1)
#pragma output_map(texture0, o2.xy)
#pragma output_map(texture0w, o2.z)
#pragma output_map(texture1, o3.xy)
```

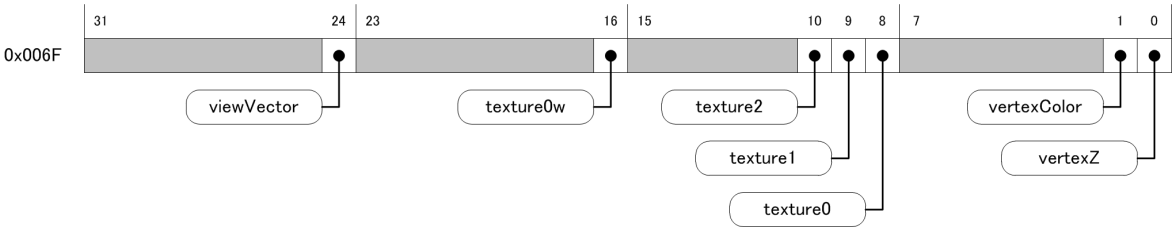
レジスタは以下のように設定します。

0x0050 : 0x03020100
0x0051 : 0x0B0A0908
0x0052 : 0x1F100D0C // w コンポーネントは無効
0x0053 : 0x1F1F0F0E // zw コンポーネントは無効
0x0054 : 0x1F1F1F1F // 第 5 属性は無効
0x0055 : 0x1F1F1F1F // 第 6 属性は無効
0x0056 : 0x1F1F1F1F // 第 7 属性は無効
0x0064 : 0x00000001 // テクスチャ座標を出力する

8.8.1.13. 出力属性のクロック制御レジスタ(0x006F)

頂点シェーダから出力する頂点属性の種類によって、クロック制御レジスタへの設定値を変更しなければならない場合があります。

図 8-22. 出力属性のクロック制御レジスタ(0x006F)のビットレイアウト



ビットレイアウト中の名前は以下のように出力属性のクロック制御設定に対応しています。下表では、それぞれのビットと設定値の対応についても説明しています。

表 8-16. 名前と設定の対応(出力属性のクロック制御レジスタ)

名前	ビット数	説明
vectorZ	1	頂点座標の出力で z 成分を出力する場合は 1、出力しない場合は 0 を設定します。
vertexColor	1	頂点カラーを出力する場合は 1、出力しない場合は 0 を設定します。
texture0	1	テクスチャ座標 0 を出力する場合は 1、出力しない場合は 0 を設定します。
texture1	1	テクスチャ座標 1 を出力する場合は 1、出力しない場合は 0 を設定します。
texture2	1	テクスチャ座標 2 を出力する場合は 1、出力しない場合は 0 を設定します。
texture0w	1	テクスチャ座標 0 の w 成分を出力する場合は 1、出力しない場合は 0 を設定します。
viewVector	1	ビューベクタ、クォータニオンを出力する場合は 1、出力しない場合は 0 を設定します。

ビットに対応する頂点属性の関連モジュールへの電源供給を制御します。使用しない頂点属性に対応するビットに 0 を設定することで、消費電力を軽減することができます。

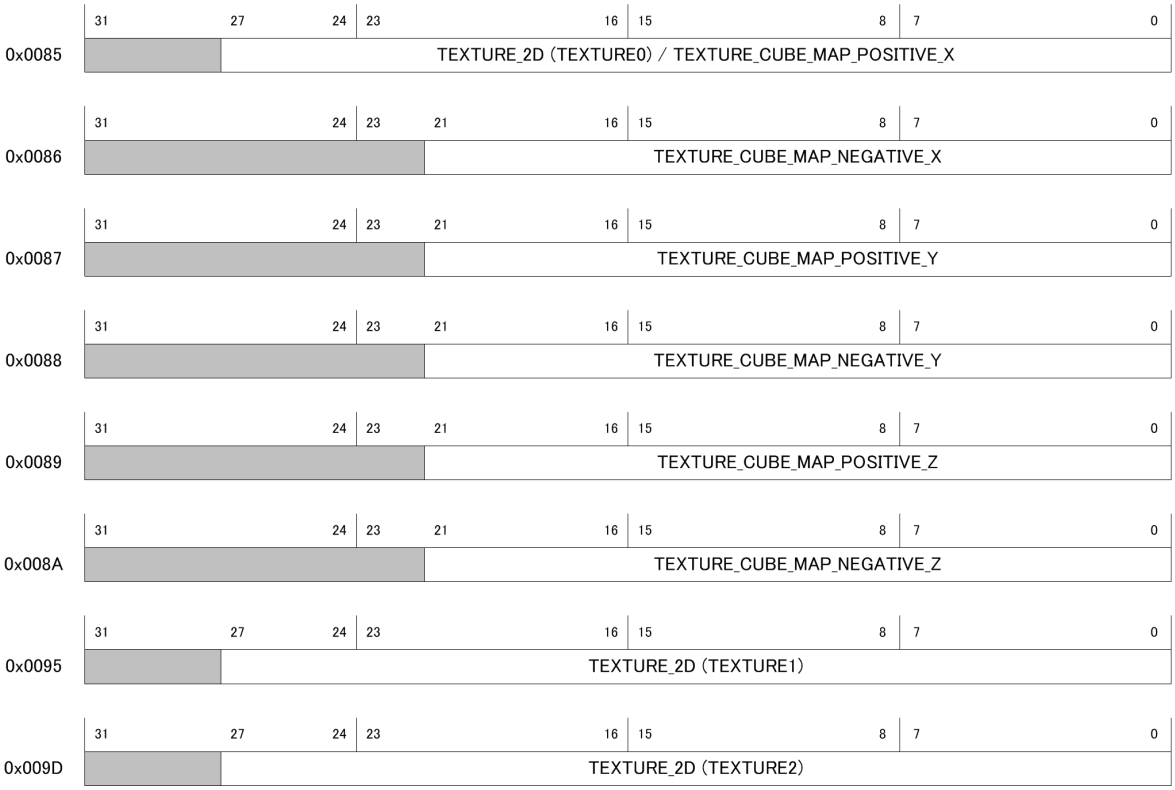
8.8.2. テクスチャアドレス設定レジスタ(0x0085 ～ 0x008A, 0x0095, 0x009D)

テクスチャデータのアドレスを設定するレジスタには、テクスチャユニット 0 の 2 次元テクスチャやキューブマップテクスチャ、テクスチャユニット 1 と 2 の 2 次元テクスチャについて設定する複数のレジスタがあります。

この項では各種ターゲットにバインドされたテクスチャデータのアドレスについての情報だけを説明します。これらの情報から

テクスチャデータの配置を変えることができます。解像度、フィルタモード、ミップマップレベル数などを変更するには「8.8.6. テクスチャ設定レジスタ(0x0080, 0x0083, 0x008B, 0x00A8 ～ 0x00B7 ほか)」を参照してください。

図 8-23. テクスチャアドレス設定レジスタ(0x0085 ほか)のビットレイアウト



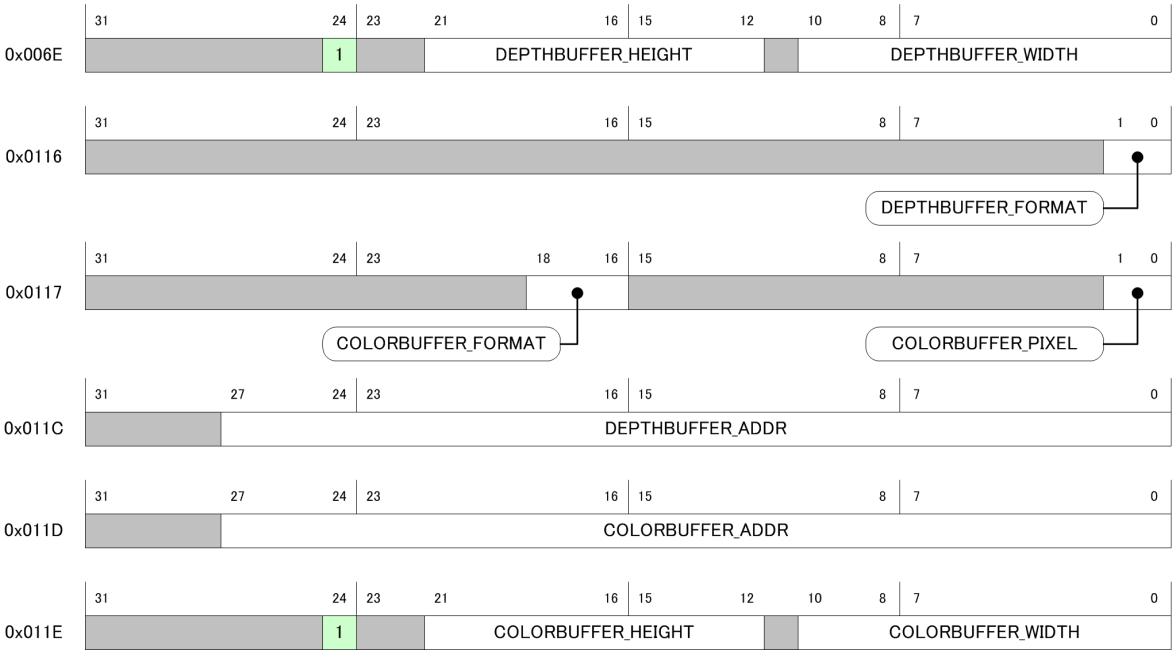
テクスチャデータのアドレス(テクスチャアドレス)はすべて 8 バイトアドレス(物理アドレスを 8 で割った値)で設定します。キューブマップ用の 6 面すべてのテクスチャアドレス(28 ビット)の上位 6 ビットはレジスタ 0x0085 のビット [27 : 22] を共有します。

> **注意:** テクスチャアドレスのアライメントは 128 バイトである必要があります。正しいアライメントでなければ、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。

8.8.3. レンダーバッファ設定レジスタ(0x006E, 0x0116, 0x0117, 0x011C ～ 0x011E)

レンダーバッファに関連するレジスタ設定には、カラーバッファとデプスバッファの 2 つのバッファについて設定する複数のレジスタがあります。これらのレジスタに対する設定コマンドはステートフラグ `NN_GX_STATE_FRAMEBUFFER` のバリデーションで生成されます。

図 8-24. レンダーバッファ設定レジスタ(0x006E, 0x0116, 0x0117, 0x011C ～ 0x011E)のビットレイアウト



ビットレイアウト中の名前は以下のようにレンダーバッファの設定に対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

表 8-17. 名前と設定の対応(レンダーバッファ設定レジスタ)

名前	ビット数	説明
COLORBUFFER_FORMAT	3	カラーバッファのフォーマットを設定します。 0x0 : GL_RGBA8_OES または GL_GAS_DMP 0x2 : GL_RGB5_A1 0x3 : GL_RGB565 0x4 : GL_RGBA4
COLORBUFFER_PIXEL	2	カラーバッファのフォーマットのピクセルサイズを設定します。 0x0 : 16 ビット 0x2 : 32ビット
COLORBUFFER_WIDTH	11	カラーバッファの幅のピクセル数を設定します。
COLORBUFFER_HEIGHT	10	カラーバッファの高さのピクセル数 - 1 を設定します。
COLORBUFFER_ADDR	28	カラーバッファのアドレスを 8 バイトアドレス(物理アドレスを 8 で割った値)で設定します。
DEPTHBUFFER_FORMAT	2	デプスバッファのフォーマットを設定します。 0x0 : GL_DEPTH_COMPONENT16 0x2 : GL_DEPTH_COMPONENT24_OES 0x3 : GL_DEPTH24_STENCIL8_EXT
DEPTHBUFFER_WIDTH	11	デプスバッファの幅のピクセル数を設定します。
DEPTHBUFFER_HEIGHT	10	デプスバッファの高さのピクセル数 - 1 を設定します。
DEPTHBUFFER_ADDR	28	デプスバッファのアドレスを 8 バイトアドレス(物理アドレスを 8 で割った値)で設定します。

レジスタ 0x011E(COLORBUFFER_WIDTH、COLORBUFFER_HEIGHT)に対する設定コマンドは、必ずバイトイネーブルに 0xF を指定してください。

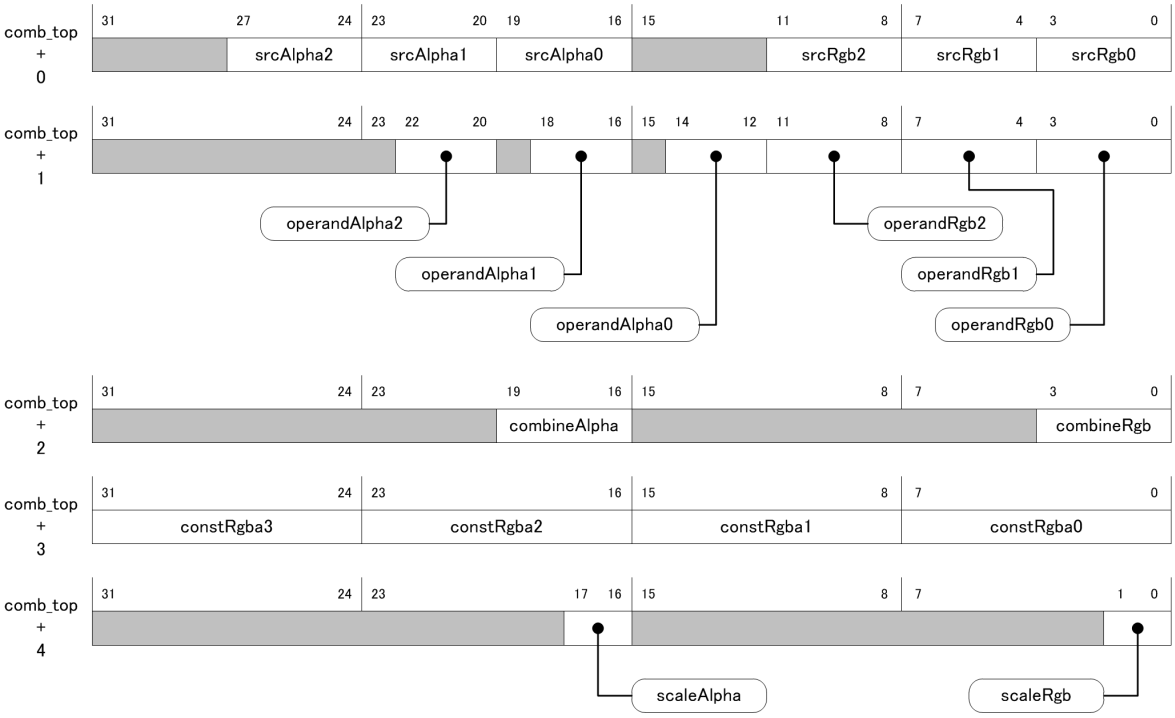
8.8.4. テクスチャコンバイナ設定レジスタ(0x00C0 ～ 0x00C4 ほか)

"dmp_TexEnv[*i*]." で始まる、テクスチャコンバイナ設定の予約ユニフォームに値を設定するレジスタは、コンバイナ番号ごとに異なるアドレス(comb_top)を先頭とする複数のレジスタに分かれています。

表 8-18. コンバイナ番号とレジスタの先頭アドレス

コンバイナ番号	先頭アドレス(comb_top)
0	0x00C0
1	0x00C8
2	0x00D0
3	0x00D8
4	0x00F0
5	0x00F8

図 8-25. テクスチャコンバイナ設定レジスタ(0x00C0 ～ 0x00C4 ほか)のビットレイアウト



ビットレイアウト中の名前は以下のように予約ユニフォームに対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

表 8-19. 名前と予約ユニフォームの対応(テクスチャコンバイナ設定レジスタ)

名前	ビット数	説明
srcRgb0 srcRgb1 srcRgb2	4	上から、dmp_TexEnv[*i*].srcRgb に設定する値の第 1 ～第 3 要素です。 0x0 : GL_PRIMARY_COLOR 0x1 : GL_FRAGMENT_PRIMARY_COLOR_DMP 0x2 : GL_FRAGMENT_SECONDARY_COLOR_DMP 0x3 : GL_TEXTURE0 0x4 : GL_TEXTURE1 0x5 : GL_TEXTURE2 0x6 : GL_TEXTURE3 0xE : GL_CONSTANT 0xF : GL_PREVIOUS 0xD : GL_PREVIOUS_BUFFER_DMP
srcAlpha0 srcAlpha1 srcAlpha2	4	上から、dmp_TexEnv[*i*].srcAlpha に設定する値の第 1 ～第 3 要素です。 設定の内容は srcRgb0 ～ srcRgb2 と同じです。
operandRgb0 operandRgb1 operandRgb2	4	上から、dmp_TexEnv[*i*].operandRgb に設定する値の第 1 ～第 3 要素です。 0x0 : GL_SRC_COLOR 0x1 : GL_ONE_MINUS_SRC_COLOR 0x2 : GL_SRC_ALPHA 0x3 : GL_ONE_MINUS_SRC_ALPHA 0x4 : GL_SRC_R_DMP 0x5 : GL_ONE_MINUS_SRC_R_DMP 0x8 : GL_SRC_G_DMP 0x9 : GL_ONE_MINUS_SRC_G_DMP 0xC : GL_SRC_B_DMP 0xD : GL_ONE_MINUS_SRC_B_DMP
operandAlpha0 operandAlpha1 operandAlpha2	3	上から、dmp_TexEnv[*i*].operandAlpha に設定する値の第 1 ～第 3 要素です。 0x0 : GL_SRC_ALPHA 0x1 : GL_ONE_MINUS_SRC_ALPHA 0x2 : GL_SRC_R_DMP 0x3 : GL_ONE_MINUS_SRC_R_DMP 0x4 : GL_SRC_G_DMP 0x5 : GL_ONE_MINUS_SRC_G_DMP 0x6 : GL_SRC_B_DMP 0x7 : GL_ONE_MINUS_SRC_B_DMP
combineRgb	4	dmp_TexEnv[*i*].combineRgb に設定する値です。 0x0 : GL_REPLACE 0x1 : GL_MODULATE 0x2 : GL_ADD 0x3 : GL_ADD_SIGNED 0x4 : GL_INTERPOLATE 0x5 : GL_SUBTRACT 0x6 : GL_DOT3_RGB 0x7 : GL_DOT3_RGBA 0x8 : GL_MULT_ADD_DMP 0x9 : GL_ADD_MULT_DMP
combineAlpha	4	dmp_TexEnv[*i*].combineAlpha に設定する値です。 combineRgb の値から GL_DOT3_RGB を除いたものを設定することができます。
constRgba0 constRgba1 constRgba2 constRgba3	8	上から、dmp_TexEnv[*i*].constRgba に設定する値の第 1 ～第 4 要素です。 0.0 ～ 1.0 の値を 0 ～ 255 にマップしたときの符号なし 8 ビット整数を設定します。 値の変換方法については「8.9.16. 浮動小数点数(0 ～ 1)から符号なし 8 ビット整数への変換」を参照してください。

scaleRgb scaleAlpha	2	dmp_TexEnv[*i*].scaleRgb, dmp_TexEnv[*i*].scaleAlpha に設定する値です。 0x0 : 1.0 0x1 : 2.0 0x2 : 4.0

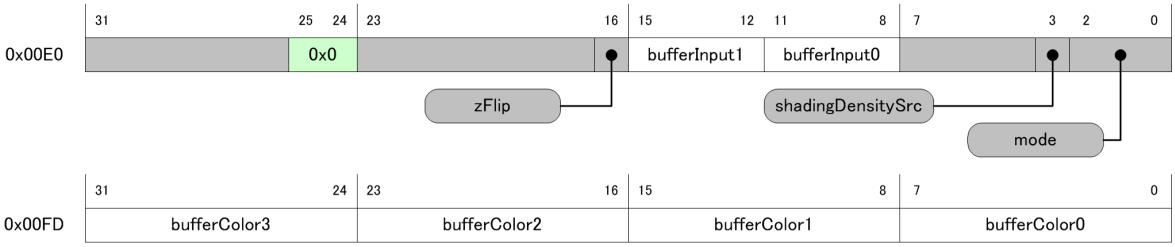
dmp_TexEnv[i].srcRgb および dmp_TexEnv[i].srcAlpha の設定には以下の制限があります。

- i が 0 の場合、dmp_TexEnv[i].srcRgb および dmp_TexEnv[i].srcAlpha の各 3 要素に対して、GL_PREVIOUS と GL_PREVIOUS_BUFFER_DMP を設定してはいけません。
- i が 0 以外の場合、dmp_TexEnv[i].srcRgb および dmp_TexEnv[i].srcAlpha のそれぞれに、各 3 要素のうちの少なくとも 1 つは GL_CONSTANT、GL_PREVIOUS、GL_PREVIOUS_BUFFER_DMP のいずれが設定されなければなりません。

8.8.4.1. コンバイナバッファ設定レジスタ(0x00E0, 0x00FD)

"dmp_TexEnv[*i*]." で始まる予約ユニフォームでは、コンバイナバッファに対する設定も行われます。コンバイナバッファの予約ユニフォームに値を設定するレジスタは以下のとおりです。レジスタ 0x00E0 の他のビットはガス設定などで使用されていることに注意してください。

図 8-26. コンバイナバッファ設定レジスタ(0x00E0, 0x00FD)のビットレイアウト



ビットレイアウト中の名前は以下のように予約ユニフォームに対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

表 8-20. 名前と予約ユニフォームの対応(コンバイナバッファ設定レジスタ)

名前	ビット数	説明
bufferColor0 bufferColor1 bufferColor2 bufferColor3	4	上から、dmp_TexEnv[0].bufferColor に設定する値の第 1 ～第 4 要素です。 0.0 ～ 1.0 の値を 0 ～ 255 にマップしたときの符号なし 8 ビット整数を設定します。 値の変換方法については「8.9.16. 浮動小数点数(0 ～ 1)から符号なし 8 ビット整数への変換」を参照してください。
bufferInput0 bufferInput1	1 * 4	上から、dmp_TexEnv[*i*].bufferInput に設定する値の第 1 ～第 2 要素です。*i* は 1 ～ 4 で、下位のビットが 1 の設定に対応しています。 0x0 : GL_PREVIOUS_BUFFER_DMP 0x1 : GL_PREVIOUS

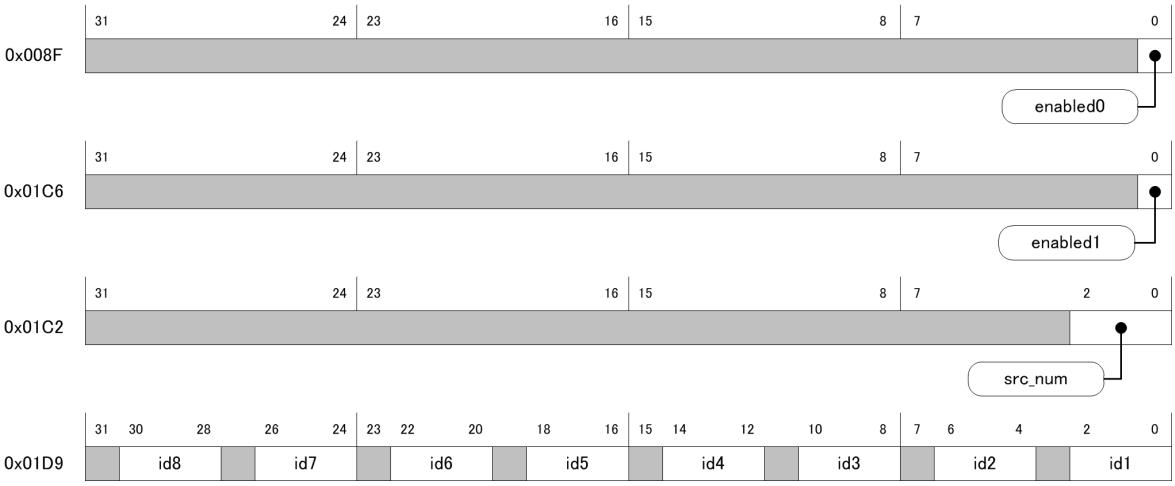
8.8.5. フラグメントライティング設定レジスタ(0x008F ほか)

フラグメントライティングの予約ユニフォーム(名前に dmp_FragmentLighting、dmp_FragmentMaterial、dmp_FragmentLightSource[*i*]、dmp_LightEnv を含む予約ユニフォーム)に関連するレジスタについて説明します。

8.8.5.1. ライティングの有効化・無効化制御レジスタ(0x008F, 0x01C2, 0x01C6, 0x01D9)

ライティングの有効化・無効化を制御する予約ユニフォームに対応するレジスタは以下のとおりです。

図 8-27. ライティングの有効化・無効化制御レジスタ(0x008F, 0x01C2 ほか)のビットレイアウト



ビットレイアウト中の名前は以下のように予約ユニフォームに対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

表 8-21. 名前と予約ユニフォームの対応(ライティングの有効化・無効化制御レジスタ)

名前	ビット数	説明
enabled0	1	dmp_FragmentLighting.enabled に設定する値です。 0x0 : GL_FALSE 0x1 : GL_TRUE
enabled1	1	dmp_FragmentLighting.enabled に設定する値です。 0x0 : GL_TRUE 0x1 : GL_FALSE ※ 設定する値に注意してください。
src_num	3	(有効な光源数 - 1)を設定します。すべての光源が無効である場合には 0 を設定します。 有効な光源数とは、dmp_FragmentLightSource[*i*].enabled に GL_TRUE を指定した光源の数のことです。
id1 ～ id8	3	有効な光源の ID を、id1 から設定します。 光源 0、光源 1、光源 3、光源 5 が有効である場合、このレジスタには 0x00005310 を設定(光源の指定順に制限はなく、0x00003150 のような設定も可能です)します。すべての光源が有効である場合は 0x76543210 を設定します。すべての光源が無効である場合は 0x00000000 を設定します。 同じ光源を複数回指定した場合は、その光源のライティング結果が複数回適用されます。光源 0 を複数回指定した場合は、ライティング結果に加えて、プライマリーカラーのグローバルアンビエントも複数回適用されます。

8.8.5.2. グローバルアンビエント設定レジスタ(0x01C0)

グローバルアンビエントの設定は、レジスタ 0x01C0 のそれぞれのビットに RGB の各成分の値を設定することで行いますが、設定する値は以下の計算結果を 0.0 ～ 1.0 の範囲にクランプし、その値を 0 ～ 255 にマップしたときの符号なし 8 ビット整数となっています。値の変換方法については「8.9.16. 浮動小数点数(0 ～ 1)から符号なし 8 ビット整数への変換」を参照してください。光源番号 0 の光源が有効になっていない場合は、このレジスタの設定は無効となり、プライマリカラーのグ

ローバルアンビエントの項には 0 が適用されます。

dmp_FragmentMaterial.emission ＋ dmp_FragmentMaterial.ambient × dmp_FragmentLighing.ambient

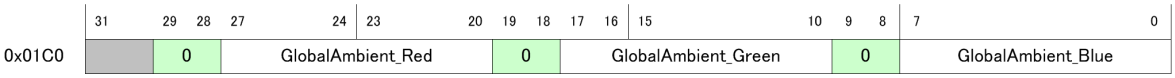
例)

マテリアルの放射光と環境光がそれぞれ(0.8, 0.8, 0.8, 0.6)と(0.2, 0.2, 0.2, 0.4)、グローバルの環境光が(1.0, 1.0, 1.0, 1.0)であるとき、各成分に設定する値は以下の計算結果を変換したものなので、143(0x8F)となります。

(0.8 × 0.6) ＋ (0.2 × 0.4) × (1.0 × 1.0) ＝ 0.56

設定するレジスタのビットレイアウトは以下のようになっています。各成分に対して 10 ビット幅(図では実際に値を設定する 8 ビット幅で示しています)が用意されていますが、計算結果は下位 8 ビットに格納し、上位 2 ビットには 0 を格納してください。上位 2 ビットに 0 以外の値を設定したときの動作は不定です。

図 8-28. グローバルアンビエント設定レジスタ(0x01C0)のビットレイアウト



ライティングが有効(dmp_FragmentLighting.enabled に GL_TRUE が設定されている)、かつすべての光源が無効(dmp_FragmentLightSource[i].enabled に GL_FALSE が設定されている)の場合、プライマリカラーにはグローバルアンビエントのみが適用されます。

有効な光源数についての設定を行うレジスタ 0x01C2 のビット[2 : 0]には(光源数 - 1)を設定するため、ライティングを有効にすると必ず 1 つの光源が有効になってしまいます。そのため、ドライバはこのような場合に、光源 0 の各項原色を黒(0.0, 0.0, 0.0, 0.0)に設定するコマンド(レジスタ 0x0140～0x0143 に 0 を設定)を生成します。また、光源 0 が有効となるように、1 個目の有効な光源を光源 0 に設定するコマンド(レジスタ 0x01D9 のビット[2 : 0]に 0x0 を設定)と、dmp_LightEnv.config に GL_LIGHT_ENV_LAYER_CONFIG0_DMP を設定するコマンド(レジスタ 0x01C3 のビット[7 : 4]に 0x0 を設定)を生成します。

8.8.5.3. 光源設定レジスタ(0x0140 ～ 0x01BF, 0x01C4)

光源ごとの設定はすべて光源番号を元に設定を行います。設定を行うレジスタは(light_top = 0x0140 + 光源番号×0x10)を先頭とする複数のレジスタです。例えば、光源 0 と光源 3 に対する光源色の設定では、dmp_FragmentLightSource[0].specular0 と dmp_FragmentLightSource[3].specular0 の設定はそれぞれ、レジスタ 0x0140 と 0x0170 に対して行います。

光源色設定レジスタ(0x0140 ～ 0x0143 ほか)

光源色の設定は、先頭レジスタ(light_top)から順番に、鏡面光 0(LightSpecular0)、鏡面光 1(LightSpecular1)、拡散光(LightDiffuse)、環境光(LightAmbient)の RGB 各成分の値を、各レジスタのビットに設定することで行いますが、設定する値は以下の計算結果を 0.0 ～ 1.0 の範囲にクランプし、その値を 0 ～ 255 にマップしたときの符号なし 8 ビット整数となっています。値の変換方法については「8.9.16. 浮動小数点数(0 ～ 1)から符号なし 8 ビット整数への変換」を参照してください。

LightSpecular0 = dmp_FragmentMaterial.specular0 × dmp_FragmentLightSource[i].specular0

dmp_LightEnv.lutEnabledRefl が GL_FALSE のとき

LightSpecular1 = dmp_FragmentMaterial.specular1 × dmp_FragmentLightSource[i].specular1

dmp_LightEnv.lutEnabledRefl が GL_TRUE のとき

LightSpecular1 = dmp_FragmentLightSource[i].specular1

LightDiffuse = dmp_FragmentMaterial.diffuse × dmp_FragmentLightSource[i].diffuse

LightAmbient = dmp_FragmentMaterial.ambient × dmp_FragmentLightSource[i].ambient

設定するレジスタのビットレイアウトは以下のようになっています。

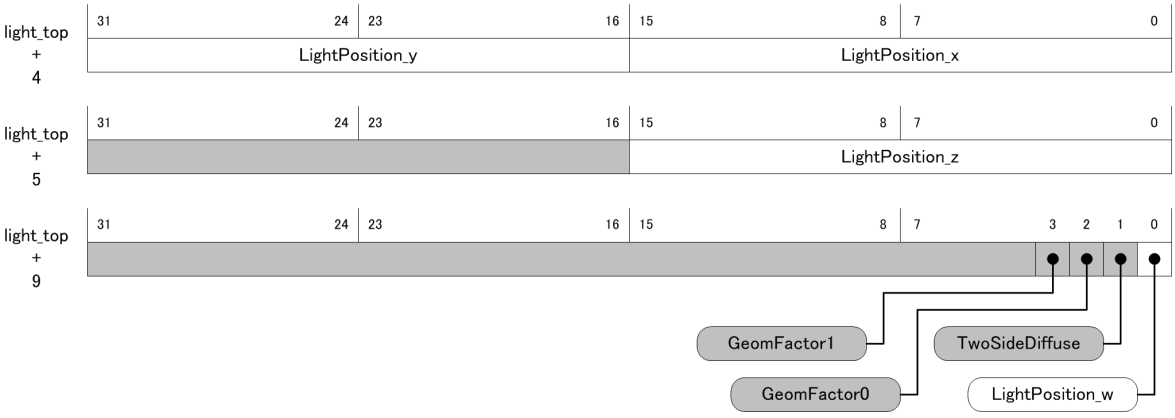
図 8-29. 光源色設定レジスタ(0x0140 ～ 0x0143 ほか)のビットレイアウト

光源位置設定レジスタ(0x0144, 0x0145, 0x0149 ほか)

予約ユニフォーム dmp_FragmentLightSource[*i*].position で設定する光源位置の座標値のうち、xyz 成分については値を 16 ビット浮動小数点数に変換してレジスタに設定します。値の変換方法については「8.9.2. 16 ビット浮動小数点数への変換」を参照してください。w 成分については、光源が点光源か平行光源かをレジスタ(light_top + 9)のビット [0 : 0] に設定します。w 成分の値が 0.0 であれば 1 を、それ以外ならば 0 を設定することに注意してください。また、このレジスタのほかのビットには別の設定が行われることにも注意してください。

設定するレジスタのビットレイアウトは以下のようになっています。

図 8-30. 光源位置設定レジスタ(0x0144, 0x0145, 0x0149 ほか)のビットレイアウト

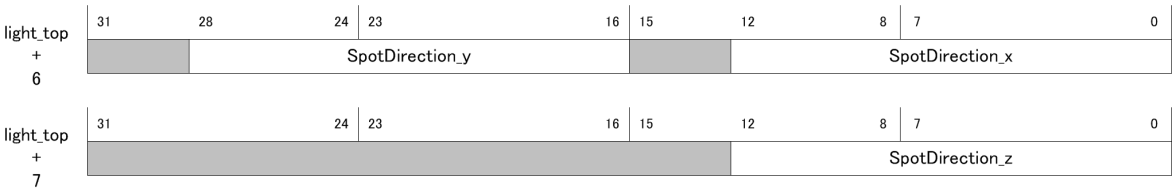


スポットライト方向設定レジスタ(0x0146, 0x0147 ほか)

予約ユニフォーム dmp_FragmentLightSource[*i*].spotDirection で設定するスポットライト方向の座標値(xyz 成分)を、**符号を反転させてから** 小数部 11 ビットの符号つき 13 ビット固定小数点数(負の値は 2 の補数表現)に変換してレジスタに設定します。値の変換方法については「8.9.9. 小数部 11 ビットの符号つき 13 ビット固定小数点数への変換」を参照してください。

設定するレジスタのビットレイアウトは以下のようになっています。

図 8-31. スポットライト方向設定レジスタ(0x0146, 0x0147 ほか)のビットレイアウト

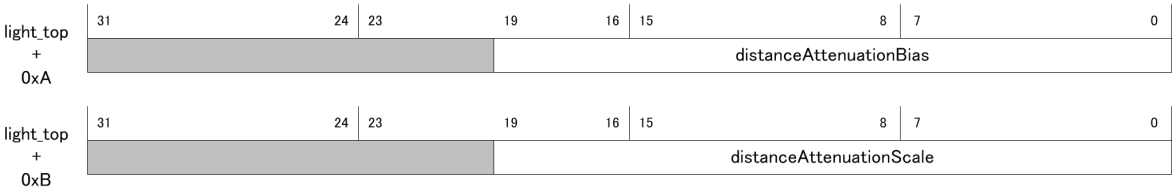


距離減衰設定レジスタ(0x014A, 0x014B ほか)

距離減衰の予約ユニフォーム dmp_FragmentLightSource[i].distanceAttenuationBias と dmp_FragmentLightSource[i].distanceAttenuationScale で設定する距離減衰のバイアスとスケールを、20 ビット浮動小数点数に変換してレジスタに設定します。値の変換方法については「8.9.4. 20 ビット浮動小数点数への変換」を参照してください。

設定するレジスタのビットレイアウトは以下のようになっています。

図 8-32. 距離減衰レジスタ設定(0x014A, 0x014B ほか)のビットレイアウト



そのほかの設定レジスタ(0x01C4, 0x0149 ほか)

光源ごとで設定する、そのほかの設定に対応するレジスタは以下のとおりです。

図 8-33. そのほかの設定レジスタ(0x01C4、0x0149 ほか)のビットレイアウト

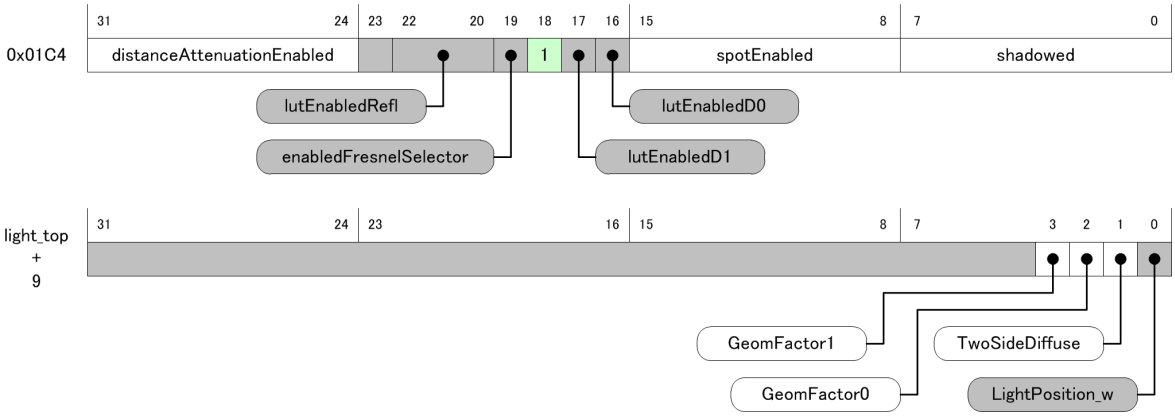


表 8-22. そのほかの設定の予約ユニフォームとレジスタの対応(光源設定)

名前・レジスタ	ビット数	説明
shadowed 0x01C4 のビット[x : x] (x は光源番号)	1	dmp_FragmentLightSource[*i*].shadowed に設定する値です。 0x0 : GL_TRUE 0x1 : GL_FALSE ※ 設定する値に注意してください。

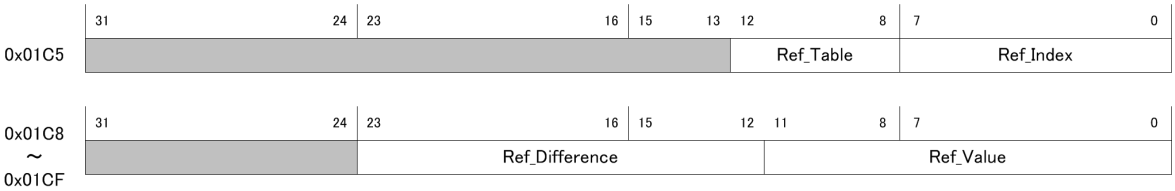
spotEnabled 0x01C4 のビット [8 + x : 8 + x] (x は光源番号)	1	dmp_FragmentLightSource[*i*].spotEnabled に設定する値です。 0x0 : GL_TRUE 0x1 : GL_FALSE ※ 設定する値に注意してください。
distanceAttenuationEnabled 0x01C4 のビット [24 + x : 24 + x] (x は光源番号)	1	dmp_FragmentLightSource[*i*].distanceAttenuationEnabled に設定する値です。 0x0 : GL_TRUE 0x1 : GL_FALSE ※ 設定する値に注意してください。
twoSideDiffuse (light_top + 9) のビット [1 : 1]	1	dmp_FragmentLightSource[*i*].twoSideDiffuse に設定する値です。 0x0 : GL_FALSE 0x1 : GL_TRUE
geomFactor0 (light_top + 9) のビット [2 : 2]	1	dmp_FragmentLightSource[*i*].geomFactor0 に設定する値です。 0x0 : GL_FALSE 0x1 : GL_TRUE
geomFactor1 (light_top + 9) のビット [3 : 3]	1	dmp_FragmentLightSource[*i*].geomFactor1 に設定する値です。 0x0 : GL_FALSE 0x1 : GL_TRUE

8.8.5.4. 参照テーブル設定レジスタ(0x01C5, 0x01C8 ～ 0x01CF)

予約ユニフォーム dmp_FragmentMaterial.sampler{ RR, RG, RB, D0, D1, FR } と dmp_FragmentLightSource[*i*].sampler{ SP, DA } で指定されるフラグメントライティングの参照テーブルは、256 個のデータと同数の差分値によって設定されます。

設定に使用するレジスタのビットレイアウトは以下のようになっています。

図 8-34. 参照テーブル設定レジスタ(0x01C5, 0x01C8 ～ 0x01CF)のビットレイアウト



レジスタ 0x01C5 には、対象とする参照テーブルを Ref_Table に、設定開始インデックスを Ref_Index に設定します。インデックスは 0 が最初のデータ、255 が最後のデータです。Ref_Table に設定する値と参照テーブルの対応は以下のようになっています。

表 8-23. Ref_Table の値と参照テーブルの対応

Ref_Table	対象の参照テーブル
0x0	ディストリビューションファクタ 0(D0)
0x1	ディストリビューションファクタ 1(D1)
0x3	フレネルファクタ(FR)
0x4	反射の青成分(RB)
0x5	反射の緑成分(RG)
0x6	反射の赤成分(RR)

0x8 + *i*	スポットライト(SP)※ *i* は光源番号
0x10 + *i*	ライトの距離減衰(DA)※ *i* は光源番号

レジスタ 0x01C8 ～ 0x01CF には、glTexImage1D() でロードする参照テーブルの i 番目のデータと i + 256 番目に設定した差分値を組み合わせた値を書き込みます。Ref_Value にはデータを小数部 12 ビットの符号なし 12 ビット固定小数点数に変換した値を、Ref_Difference には差分値を小数部 11 ビットの符号つき 12 ビット固定小数点数(小数部は絶対値のため、負の値は 2 の補数表現ではありません)に変換した値を、それぞれ設定してください。値の変換方法については「8.9.13. 小数部 12 ビットの符号なし 12 ビット固定小数点数への変換」と「8.9.6. 小数部 11 ビットの符号つき 12 ビット固定小数点数への変換」を参照してください。

対象とする参照テーブルとインデックスをレジスタ 0x01C5 に設定してから、0x01C8 ～ 0x01CF のいずれかのレジスタに変換した値の組み合わせを書き込みます。いずれのレジスタに書き込んでも処理結果は変わらず、1 つデータを書き込むごとにインデックスが 1 インクリメントされます。

8.8.5.5. 参照テーブルの引数範囲設定レジスタ(0x01D0)

参照テーブルの引数の範囲設定に対応するレジスタは以下のとおりです。

図 8-35. 参照テーブルの引数範囲設定レジスタ(0x01D0)のビットレイアウト

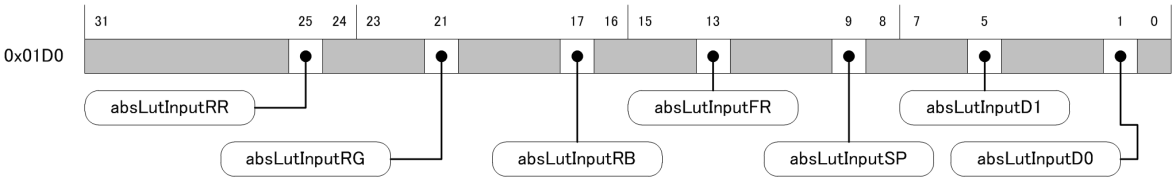


表 8-24. 参照テーブルの引数範囲設定の予約ユニフォームとレジスタの対応

名前・レジスタ	ビット数	説明
absLutInputD0 0x01D0 のビット[1 : 1]	1	dmp_LightEnv.absLutInputD0 に設定する値です。 0x0 : GL_TRUE 0x1 : GL_FALSE ※ 設定する値に注意してください。
absLutInputD1 0x01D0 のビット[5 : 5]	1	dmp_LightEnv.absLutInputD1 に設定する値です。 0x0 : GL_TRUE 0x1 : GL_FALSE ※ 設定する値に注意してください。
absLutInputSP 0x01D0 のビット[9 : 9]	1	dmp_LightEnv.absLutInputSP に設定する値です。 0x0 : GL_TRUE 0x1 : GL_FALSE ※ 設定する値に注意してください。
absLutInputFR 0x01D0 のビット[13 : 13]	1	dmp_LightEnv.absLutInputFR に設定する値です。 0x0 : GL_TRUE 0x1 : GL_FALSE ※ 設定する値に注意してください。
absLutInputRB 0x01D0 のビット[17 : 17]	1	dmp_LightEnv.absLutInputRB に設定する値です。 0x0 : GL_TRUE 0x1 : GL_FALSE ※ 設定する値に注意してください。
absLutInputRG 0x01D0 のビット[21 : 21]	1	dmp_LightEnv.absLutInputRG に設定する値です。 0x0 : GL_TRUE 0x1 : GL_FALSE ※ 設定する値に注意してください。

absLutInputRR 0x01D0 のビット [25 : 25]	1	dmp_LightEnv.absLutInputRR に設定する値です。 0x0 : GL_TRUE 0x1 : GL_FALSE ※ 設定する値に注意してください。

8.8.5.6. 参照テーブルの入力値設定レジスタ(0x01D1)

参照テーブルの入力値設定に対応するレジスタは以下のとおりです。

図 8-36. 参照テーブルの入力値設定レジスタ(0x01D1)のビットレイアウト

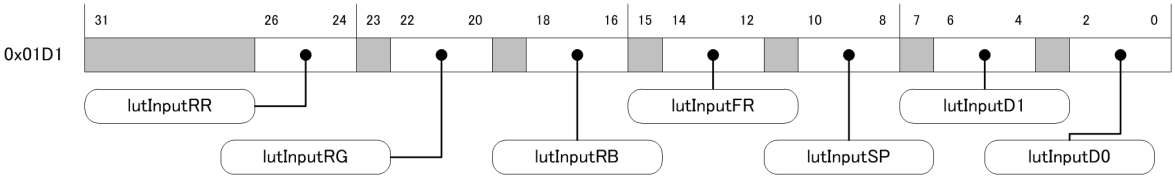


表 8-25. 参照テーブルの入力値設定の予約ユニフォームとレジスタの対応

名前・レジスタ	ビット数	説明
lutInputD0 0x01D1 のビット [2 : 0]	3	dmp_LightEnv.lutInputD0 に設定する値です。 0x0 : GL_LIGHT_ENV_NH_DMP 0x1 : GL_LIGHT_ENV_VH_DMP 0x2 : GL_LIGHT_ENV_NV_DMP 0x3 : GL_LIGHT_ENV_LN_DMP 0x4 : GL_LIGHT_ENV_SP_DMP 0x5 : GL_LIGHT_ENV_CP_DMP
lutInputD1 0x01D1 のビット [6 : 4]	3	dmp_LightEnv.lutInputD1 に設定する値です。 説明は lutInputD0 と同じです。
lutInputSP 0x01D1 のビット [10 : 8]	3	dmp_LightEnv.lutInputSP に設定する値です。 説明は lutInputD0 と同じです。
lutInputFR 0x01D1 のビット [14 : 12]	3	dmp_LightEnv.lutInputFR に設定する値です。 0x0 : GL_LIGHT_ENV_NH_DMP 0x1 : GL_LIGHT_ENV_VH_DMP 0x2 : GL_LIGHT_ENV_NV_DMP 0x3 : GL_LIGHT_ENV_LN_DMP
lutInputRB 0x01D1 のビット [18 : 16]	3	dmp_LightEnv.lutInputRB に設定する値です。 説明は lutInputFR と同じです。
lutInputRG 0x01D1 のビット [22 : 20]	3	dmp_LightEnv.lutInputRG に設定する値です。 説明は lutInputFR と同じです。
lutInputRR 0x01D1 のビット [26 : 24]	3	dmp_LightEnv.lutInputRR に設定する値です。 説明は lutInputFR と同じです。

8.8.5.7. 参照テーブルの出力値に対するスケール値設定レジスタ(0x01D2)

参照テーブルの出力値に対するスケール値設定に対応するレジスタは以下のとおりです。

図 8-37. 参照テーブルの出力地に対するスケール値設定レジスタ(0x01D2)のビットレイアウト

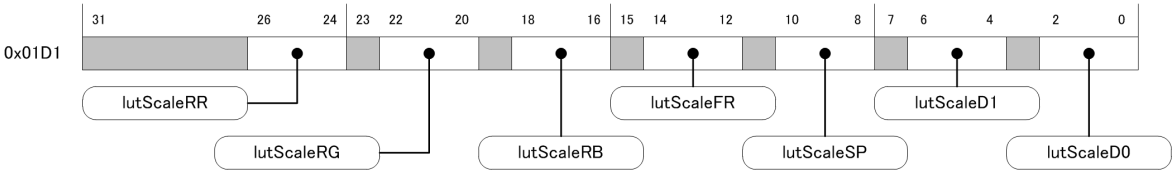


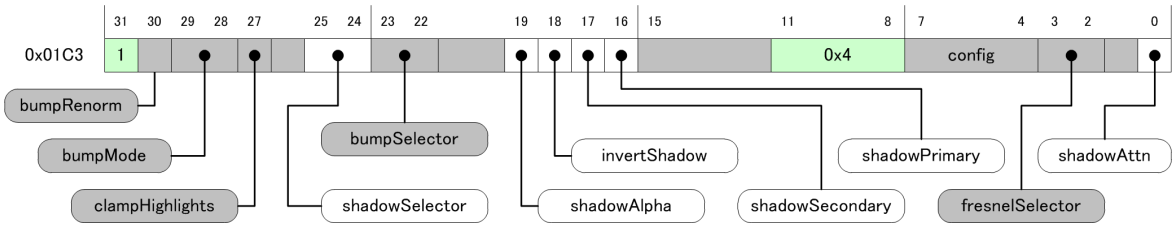
表 8-26. 参照テーブルの出力値に対するスケール値設定の予約ユニフォームとレジスタの対応

名前・レジスタ	ビット数	説明
lutScaleD0 0x01D2 のビット [2 : 0]	3	dmp_LightEnv.lutScaleD0 に設定する値です。 0x0 : 1.0 0x1 : 2.0 0x2 : 4.0 0x3 : 8.0 0x6 : 0.25 0x7 : 0.5
lutScaleD1 0x01D2 のビット [6 : 4]	3	dmp_LightEnv.lutScaleD1 に設定する値です。 説明は lutScaleD0 と同じです。
lutScaleSP 0x01D2 のビット [10 : 8]	3	dmp_LightEnv.lutScaleSP に設定する値です。 説明は lutScaleD0 と同じです。
lutScaleFR 0x01D2 のビット [14 : 12]	3	dmp_LightEnv.lutScaleFR に設定する値です。 説明は lutScaleD0 と同じです。
lutScaleRB 0x01D2 のビット [18 : 16]	3	dmp_LightEnv.lutScaleRB に設定する値です。 説明は lutScaleD0 と同じです。
lutScaleRG 0x01D2 のビット [22 : 20]	3	dmp_LightEnv.lutScaleRG に設定する値です。 説明は lutScaleD0 と同じです。
lutScaleRR 0x01D2 のビット [26 : 24]	3	dmp_LightEnv.lutScaleRR に設定する値です。 説明は lutScaleD0 と同じです。

8.8.5.8. シャドウ減衰設定レジスタ(0x01C3)

シャドウ減衰設定に対応するレジスタは以下のとおりです。このレジスタのほかのビットには別の設定が行われることに注意してください。

図 8-38. シャドウ減衰設定レジスタ(0x01C3)のビットレイアウト



ビットレイアウト中の名前は以下のように予約ユニフォームに対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

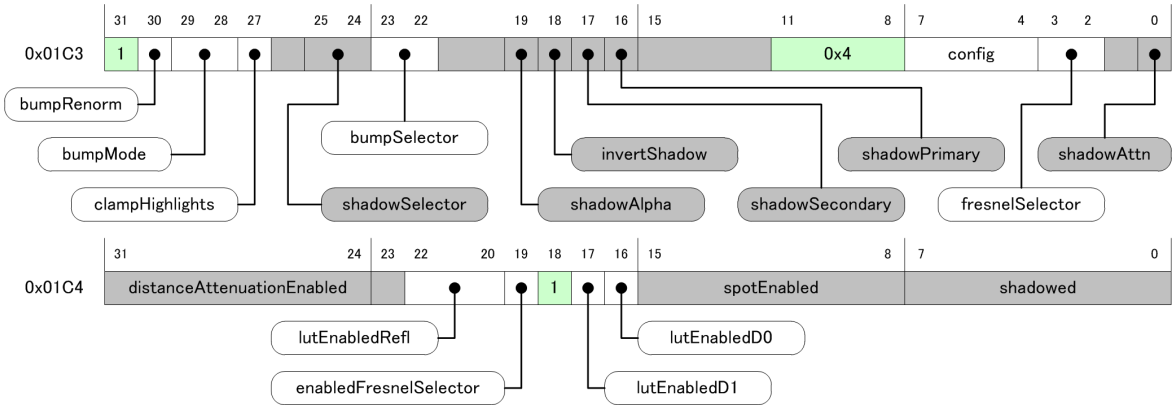
表 8-27. 名前と予約ユニフォームの対応(シャドウ減衰設定レジスタ)

名前	ビット数	説明
shadowSelector	2	dmp_LightEnv.shadowSelector に設定する値です。 0x0 : GL_TEXTURE0 0x1 : GL_TEXTURE1 0x2 : GL_TEXTURE2 0x3 : GL_TEXTURE3
shadowPrimary	1	dmp_LightEnv.shadowPrimary に設定する値です。 0x0 : GL_FALSE 0x1 : GL_TRUE
shadowSecondary	1	dmp_LightEnv.shadowSecondary に設定する値です。 0x0 : GL_FALSE 0x1 : GL_TRUE
invertShadow	1	dmp_LightEnv.invertShadow に設定する値です。 0x0 : GL_FALSE 0x1 : GL_TRUE
shadowAlpha	1	dmp_LightEnv.shadowAlpha に設定する値です。 0x0 : GL_FALSE 0x1 : GL_TRUE
shadowAttn	1	dmp_LightEnv.shadowPrimary、 dmp_LightEnv.shadowSecondary、 dmp_LightEnv.shadowAlpha のいずれかが GL_TRUE の場合は 1 を設定し、すべて GL_FALSE の場合は 0 を設定します。

8.8.5.9. そのほかの設定レジスタ(0x01C3, 0x01C4)

フラグメントライティングで設定する、そのほかの設定に対応するレジスタは以下のとおりです。このレジスタのほかのビットには別の設定が行われることに注意してください。

図 8-39. そのほかの設定レジスタ(0x01C3, 0x01C4)のビットレイアウト



ビットレイアウト中の名前は以下のように予約ユニフォームに対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

表 8-28. 名前と予約ユニフォームの対応（そのほかの設定レジスタ）

名前	ビット数	説明
config	4	dmp_LightEnv.config に設定する値です。 0x0 : GL_LIGHT_ENV_LAYER_CONFIG0_DMP 0x1 : GL_LIGHT_ENV_LAYER_CONFIG1_DMP 0x2 : GL_LIGHT_ENV_LAYER_CONFIG2_DMP 0x3 : GL_LIGHT_ENV_LAYER_CONFIG3_DMP 0x4 : GL_LIGHT_ENV_LAYER_CONFIG4_DMP 0x5 : GL_LIGHT_ENV_LAYER_CONFIG5_DMP 0x6 : GL_LIGHT_ENV_LAYER_CONFIG6_DMP 0x8 : GL_LIGHT_ENV_LAYER_CONFIG7_DMP
fresnelSelector	2	dmp_LightEnv.fresnelSelector に設定する値です。 0x0 : GL_LIGHT_ENV_NO_FRESNEL_DMP 0x1 : GL_LIGHT_ENV_PRI_ALPHA_FRESNEL_DMP 0x2 : GL_LIGHT_ENV_SEC_ALPHA_FRESNEL_DMP 0x3 : GL_LIGHT_ENV_PRI_SEC_ALPHA_FRESNEL_DMP
enabledFresnelSelector	1	dmp_LightEnv.fresnelSelector に設定する値によって変化します。 0x0 : GL_LIGHT_ENV_NO_FRESNEL_DMP 以外の設定 0x1 : GL_LIGHT_ENV_NO_FRESNEL_DMP
bumpSelector	2	dmp_LightEnv.bumpSelector に設定する値です。 0x0 : GL_TEXTURE0 0x1 : GL_TEXTURE1 0x2 : GL_TEXTURE2 0x3 : GL_TEXTURE3
bumpMode	2	dmp_LightEnv.bumpMode に設定する値です。 0x0 : GL_LIGHT_ENV_BUMP_NOT_USED_DMP 0x1 : GL_LIGHT_ENV_BUMP_AS_BUMP_DMP 0x2 : GL_LIGHT_ENV_BUMP_AS_TANG_DMP
bumpRenorm	1	dmp_LightEnv.bumpRenorm に設定する値です。 0x0 : GL_TRUE または dmp_LightEnv.bumpMode が GL_LIGHT_ENV_BUMP_NOT_USED_DMP 0x1 : 上記以外
clampHighlights	1	dmp_LightEnv.clampHighlights に設定する値です。 0x0 : GL_FALSE 0x1 : GL_TRUE
lutEnabledD0	1	dmp_LightEnv.lutEnabledD0 に設定する値です。 0x0 : GL_TRUE 0x1 : GL_FALSE ※ 設定する値に注意してください。
lutEnabledD1	1	dmp_LightEnv.lutEnabledD1 に設定する値です。 0x0 : GL_TRUE 0x1 : GL_FALSE ※ 設定する値に注意してください。
lutEnabledRefl	3	dmp_LightEnv.lutEnabledRefl に設定する値です。 0x0 : GL_TRUE 0x7 : GL_FALSE ※ 設定する値に注意してください。

※ config（レジスタ 0x01C3 のビット [7 : 4]）の設定値により、ピクセル処理のサイクル数が変化します。ライティングが無効の場合でもその設定は影響しますので、ライティングがオフの場合は config の設定をサイクル数 1 の設定に変更するようにしてください。ドライバはライティングが無効の場合に、レジスタ 0x01C3 のビット [7 : 4] に 0x0 を設定します。また、設定値が（8:GL_LIGHT_ENV_LAYER_CONFIG7_DMP）の場合は距離減衰が使用できなくなります。距離減衰の設定が有効のままだと不正な値が距離減衰項に適用されてしまいますので、レジスタ 0x01C4 のビット [31 : 24] に 0xFF を設定して、距離減衰を無効にしてください。

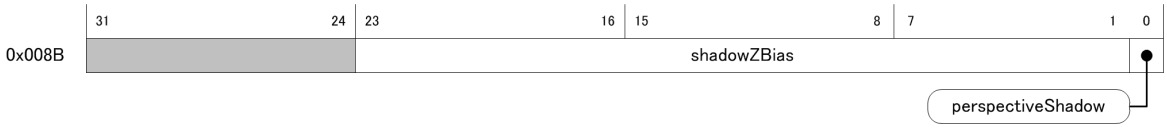
8.8.6. テクスチャ設定レジスタ(0x0080, 0x0083, 0x008B, 0x00A8 ～ 0x00B7 ほか)

テクスチャ設定の予約ユニフォーム(名前に dmp_Texture[*i*] を含む予約ユニフォーム)に関連するレジスタについて説明します。これらのレジスタに対する設定コマンドはステートフラグ NN_GX_STATE_TEXTURE のバリデーションで生成されます。また、「8.8.2. テクスチャアドレス設定レジスタ(0x0085 ～ 0x008A, 0x0095, 0x009D)」も併せて参照してください。

8.8.6.1. シャドウテクスチャ設定レジスタ(0x008B)

シャドウテクスチャ設定に対応するレジスタは以下のとおりです。

図 8-40. シャドウテクスチャ設定レジスタ(0x008B)のビットレイアウト



ビットレイアウト中の名前は以下のように予約ユニフォームに対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

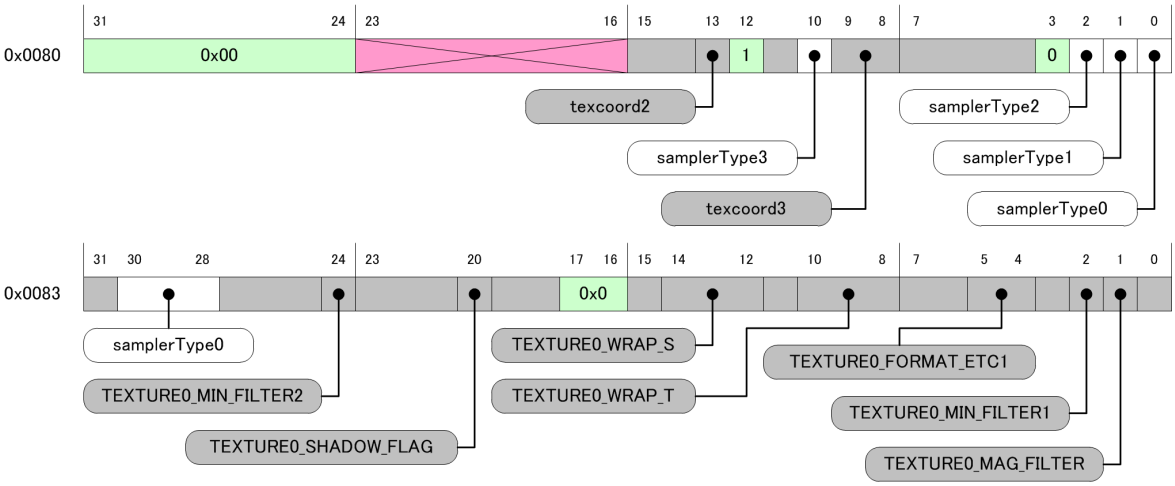
表 8-29. 名前と予約ユニフォームの対応(シャドウテクスチャ設定レジスタ)

名前	ビット数	説明
perspectiveShadow	1	dmp_Texture[0].perspectiveShadow に設定する値です。 0x0 : GL_TRUE 0x1 : GL_FALSE ※ 設定する値に注意してください。
shadowZBias	23	dmp_Texture[0].shadowZBias に設定する値を 23 ビット符号なし固定小数点数に変換した値です。 24 ビット固定小数点数に変換した値の上位 23 ビットを使用します。値の変換方法については「8.9.14. 小数部 24 ビットの符号なし 24 ビット固定小数点数への変換」を参照してください。

8.8.6.2. テクスチャサンプラータイプ設定レジスタ(0x0080, 0x0083)

テクスチャのサンプラータイプ設定に対応するレジスタは以下のとおりです。これらのレジスタのほかのビットには別の設定が行われることに注意してください。

図 8-41. テクスチャサンプラータイプ設定レジスタ(0x0080, 0x0083)のビットレイアウト



ビットレイアウト中の名前は以下のように予約ユニフォームに対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

表 8-30. 名前と予約ユニフォームの対応(テクスチャサンプラータイプ設定レジスタ)

名前	ビット数	説明
samplerType0	1 + 3	dmp_Texture[0].samplerType に設定する値によって、レジスタ 0x0080 と 0x0083 の対応するビットに設定する値が変化します。 GL_FALSE のときは、レジスタ 0x0080 のビット [0 : 0] と 0x0083 のビット [30 : 28] には、それぞれ 0x0 と 0x0 を設定します。 GL_FALSE 以外のときは、レジスタ 0x0080 のビット [0 : 0] には 0x1 を設定し、0x0083 のビット [30 : 28] には以下の値を設定します。 0x0 : GL_TEXTURE_2D 0x1 : GL_TEXTURE_CUBE_MAP 0x2 : GL_TEXTURE_SHADOW_2D_DMP 0x3 : GL_TEXTURE_PROJECTION_DMP 0x4 : GL_TEXTURE_SHADOW_CUBE_DMP
samplerType1	1	dmp_Texture[1].samplerType に設定する値です。 0x0 : GL_FALSE 0x1 : GL_TEXTURE_2D
samplerType2	1	dmp_Texture[2].samplerType に設定する値です。 0x0 : GL_FALSE 0x1 : GL_TEXTURE_2D
samplerType3	1	dmp_Texture[3].samplerType に設定する値です。 0x0 : GL_FALSE 0x1 : GL_TEXTURE_PROCEDURAL_DMP

dmp_Texture[0].samplerType、dmp_Texture[1].samplerType、dmp_Texture[2].samplerType の設定はステートフラグ NN_GX_STATE_FSUNIFORM ではなく、glDrawElements() または glDrawArrays() の呼び出し時に生成されることに注意してください。

レジスタ 0x0080 のビット [16 : 16] に 1 を書き込むことで、テクスチャのすべてのキャッシュ(1 次、2 次の両方)がクリアされます。その際、レジスタ 0x0080 のビット [23 : 17] には、すべて 0 を書き込んでください。テクスチャキャッシュのクリアはテクスチャユニットの設定が変更されたときに必要になります。テクスチャユニットごとに搭載されているテクスチャの 1 次キャッシュをクリアする場合は、あらかじめテクスチャユニットが有効になっていなければなりません。テクスチャユニットを有効にするには、テクスチャユニットのサンプラータイプを GL_FALSE 以外に設定してください。

テクスチャキャッシュのクリアコマンドより前に、テクスチャユニットを有効にするコマンドが別に必要です。テクスチャキャッシュのクリアがテクスチャユニットの有効・無効の設定と同じレジスタにあるため、テクスチャユニットを無効から有効にするビット書き込みとテクスチャキャッシュをクリアするビット書き込みを 1 つのコマンドで行うと、テクスチャキャッシュのクリアが正しく行われません。ただし、有効から無効にするビット書き込みとテクスチャキャッシュをクリアするビット書き込みを 1 つのコマンドで行った場合はテクスチャキャッシュのクリアが正常に行われます。

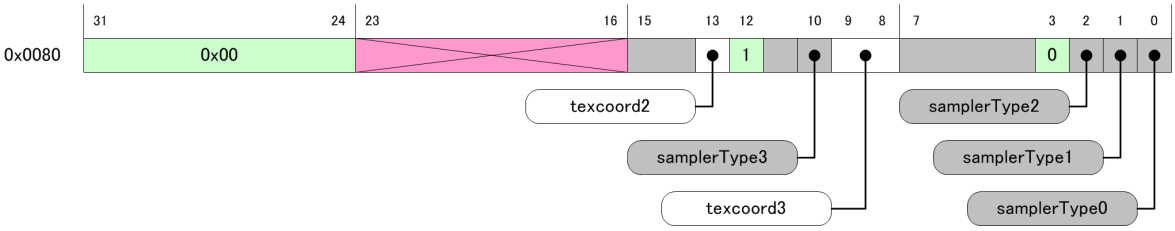
レジスタ 0x0080 のほかのビットを設定するコマンドで、テクスチャキャッシュをクリアする必要がないときは、バイトイネーブルに 0xB を設定してビット [23 : 16] にはアクセスしないでください。

テクスチャキャッシュのクリアが必要となるのは、テクスチャアドレスを設定するレジスタ(0x0085、0x0086、0x0087、0x0088、0x0089、0x008A、0x0095、0x009D)が変更された場合と、テクスチャデータがリロードされた場合です。また、テクスチャのアドレスやデータの内容に変更がなくても、フォーマットだけが変更された場合にもキャッシュのクリアが必要となります。

8.8.6.3. テクスチャ座標の選択設定レジスタ(0x0080)

テクスチャユニットに入力されるテクスチャ座標の選択設定に対応するレジスタは以下のとおりです。このレジスタのほかのビットには別の設定が行われることに注意してください。

図 8-42. テクスチャ座標の選択設定レジスタ(0x0080)のビットレイアウト



ビットレイアウト中の名前は以下のように予約ユニフォームに対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

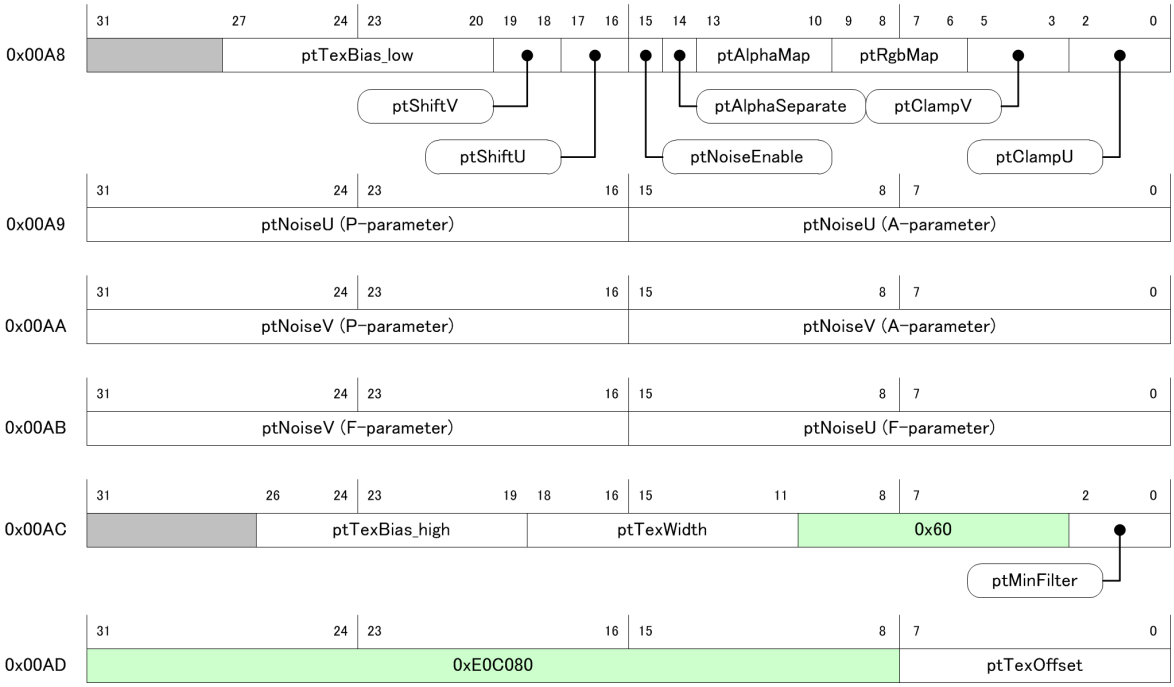
表 8-31. 名前と予約ユニフォームの対応(テクスチャ座標の選択設定レジスタ)

名前	ビット数	説明
texcoord2	1	dmp_Texture[2].texcoord に設定する値です。 0x0 : GL_TEXTURE2 0x1 : GL_TEXTURE1
texcoord3	2	dmp_Texture[3].texcoord に設定する値です。 0x0 : GL_TEXTURE0 0x1 : GL_TEXTURE1 0x2 : GL_TEXTURE2

8.8.6.4. プロシージャルテクスチャ設定レジスタ(0x00A8 ～ 0x00AD)

プロシージャルテクスチャ設定の予約ユニフォームに値を設定するレジスタのビットレイアウトは以下のようになっています。

図 8-43. プロシージャルテクスチャ設定レジスタ(0x00A8 ～ 0x00AD)のビットレイアウト



ビットレイアウト中の名前は以下のように予約ユニフォームに対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

表 8-32. 名前と予約ユニフォームの対応(プロシージャルテクスチャ設定レジスタ)

名前	ビット数	説明
ptRgbMap ptAlphaMap	4	上から、dmp_Texture[3].ptRgbMap、 dmp_Texture[3].ptAlphaMap に設定する値です。 0x0 : GL_PROCTEX_U_DMP 0x1 : GL_PROCTEX_U2_DMP 0x2 : GL_PROCTEX_V_DMP 0x3 : GL_PROCTEX_V2_DMP 0x4 : GL_PROCTEX_ADD_DMP 0x5 : GL_PROCTEX_ADD2_DMP 0x6 : GL_PROCTEX_ADDSQRT2_DMP 0x7 : GL_PROCTEX_MIN_DMP 0x8 : GL_PROCTEX_MAX_DMP 0x9 : GL_PROCTEX_RMAX_DMP
ptAlphaSeparate	1	dmp_Texture[3].ptAlphaSeparate に設定する値です。 0x0 : GL_FALSE 0x1 : GL_TRUE
ptClampU ptClampV	3	上から、dmp_Texture[3].ptClampU、 dmp_Texture[3].ptClampV に設定する値です。 0x0 : GL_CLAMP_TO_ZERO_DMP 0x1 : GL_CLAMP_TO_EDGE 0x2 : GL_SYMMETRICAL_REPEAT_DMP 0x3 : GL_MIRRORED_REPEAT 0x4 : GL_PULSE_DMP
ptShiftU ptShiftV	2	上から、dmp_Texture[3].ptShiftU、 dmp_Texture[3].ptShiftV に設定する値です。 0x0 : GL_NONE_DMP 0x1 : GL_ODD_EDGE 0x2 : GL_EVEN_DMP
ptMinFilter	3	dmp_Texture[3].ptMinFilter に設定する値です。 0x0 : GL_NEAREST 0x1 : GL_LINEAR 0x2 : GL_NEAREST_MIPMAP_NEAREST 0x3 : GL_LINEAR_MIPMAP_NEAREST 0x4 : GL_NEAREST_MIPMAP_LINEAR 0x5 : GL_LINEAR_MIPMAP_LINEAR
ptTexOffset ptTexWidth	8	上から、dmp_Texture[3].ptTexOffset、 dmp_Texture[3].ptTexWidth に設定する値です。ユニフォームで設定した値そのままです。
ptTexBias_high ptTexBias_low	8	dmp_Texture[3].ptTexBias に設定する値ですが、16 ビットの浮動小数点数に変換したものを、上位と下位 8 ビットずつに分割して設定します。値の変換方法については「8.9.2. 16 ビット浮動小数点数への変換」を参照してください。
ptNoiseEnable	1	dmp_Texture[3].ptNoiseEnable に設定する値です。 0x0 : GL_FALSE 0x1 : GL_TRUE
ptNoiseU (F-parameter) ptNoiseU (P-parameter) ptNoiseU (A-parameter)	16	上から、dmp_Texture[3].ptNoiseU に設定する値の第 1 ～第 3 要素です。 第 1 と第 2 要素はユニフォームの値を 16 ビット浮動小数点数に変換した値です。値の変換方法については「8.9.2. 16 ビット浮動小数点数への変換」を参照してください。 第 3 要素はユニフォームの値を小数部 12 ビットの符号つき 16 ビット固定小数点数(負の値は 2 の補数表現)に変換した値です。値の変換方法については「8.9.10. 小数部 12 ビットの符号つき 16 ビット固定小数点数への変換」を参照してください。

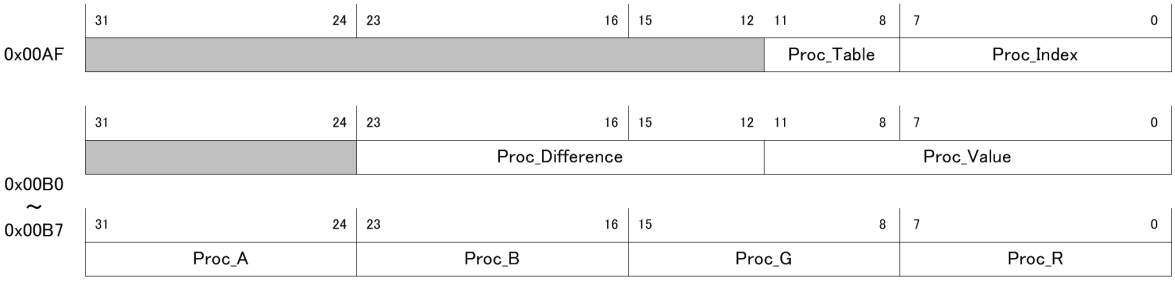
ptNoiseV (F-parameter) ptNoiseV (P-parameter) ptNoiseV (A-parameter)	16	上から、dmp_Texture[3].ptNoiseV に設定する値の第 1 ～第 3 要素です。 第 1 と第 2 要素はユニフォームの値を 16 ビット浮動小数点数に変換した値です。値の変換方法については「8.9.2. 16 ビット浮動小数点数への変換」を参照してください。 第 3 要素はユニフォームの値を小数部 12 ビットの符号つき 16 ビット固定小数点数(負の値は 2 の補数表現)に変換した値です。値の変換方法については「8.9.10. 小数部 12 ビットの符号つき 16 ビット固定小数点数への変換」を参照してください。

8.8.6.5. プロシージャルテクスチャの参照テーブル設定レジスタ(0x00AF, 0x00B0 ～ 0x00B7)

予約ユニフォーム dmp_Texture[3].ptSampler{ RgbMap, AlphaMap, NoiseMap, R, G, B, A } で指定されるプロシージャルテクスチャの参照テーブルは、RgbMap、AlphaMap、NoiseMap が 128 個のデータと同数の差分値によって設定され、R、G、B、A が 256 個のデータと同数の差分値によって設定されます。

設定に使用するレジスタはデータの個数にかかわらず同じで、そのビットレイアウトは以下のようになっています。

図 8-44. プロシージャルテクスチャの参照テーブル設定レジスタ(0x00AF ほか)のビットレイアウト



レジスタ 0x00AF には、対象とする参照テーブルを Proc_Table に、設定開始インデックスを Proc_Index に設定します。0 が最初のデータです。Proc_Table に設定する値と参照テーブルの対応は以下のようになっています。設定する値(3 ビット)に対して 4 ビット幅が用意されていますが、ビット [11 : 11] に 0 を設定しなければ正しい参照テーブルが設定されません。

表 8-33. Proc_Table の値と参照テーブルの対応

Proc_Table	対象の参照テーブル
0x0	ノイズ変調テーブル(NoiseMap)
0x2	RGB マッピングの F 関数(RgbMap)
0x3	アルファマッピングの F 関数(AlphaMap)
0x4	カラー参照テーブルのカラー値(R, G, B, A)
0x5	カラー参照テーブルの差分値(R, G, B, A)

それぞれの参照テーブルへ値を設定するレジスタは同じですが、カラー参照テーブルとそのほかの参照テーブルでビットレイアウトが異なります。

対象とする参照テーブルとインデックスをレジスタ 0x00AF に設定してから、0x00B0 ～ 0x00B7 のいずれかのレジスタにデータを書き込みます。いずれのレジスタに書き込んでも処理結果は変わらず、1 つデータを書き込むごとにインデックスが 1 インクリメントされるのは、どの参照テーブルでも同じです。また、レジスタ 0x0080 のビット [10 : 10](samplerType3)に 0x1(プロシージャルテクスチャ)を設定していなければ、レジスタ 0x00B0 ～ 0x00B7 への書き込みは無視されます。

ノイズ変調テーブル、RGB マッピングの F 関数、アルファマッピングの F 関数

レジスタ 0x00B0 ～ 0x00B7 には、glTexImage1D() でロードする参照テーブルの i 番目のデータと i + 128 番目に設定した差分値を組み合わせた値を書き込みます。Proc_Value にはデータを小数部 12 ビットの符号なし 12 ビット固定小数点数に変換した値を、Proc_Difference には差分値を小数部 11 ビットの符号つき 12 ビット固定小数点数(負の値は 2 の補数表現)に変換した値を、それぞれ設定してください。値の変換方法については「8.9.13. 小数部 12 ビットの符号なし 12 ビット固定小数点数への変換」と「8.9.7. 小数部 11 ビットの符号つき 12 ビット固定小数点数への変換 2」を参照してください。
データの個数は 128 個ですので、Proc_Index には 0 ～ 127 を指定することができます。

カラー参照テーブル

カラー値と差分値でレジスタに書き込む値のフォーマットが異なります。

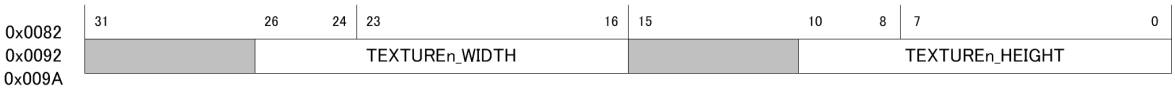
カラー値は、glTexImage1D() でロードする RGBA それぞれの参照テーブルの i 番目のデータをパックした値を書き込みます。書き込む値は、成分ごとに 0.0 ～ 1.0 の範囲を 0 ～ 255 にマップしたときの符号なし 8 ビット整数に変換(変換方法については「8.9.16. 浮動小数点数(0 ～ 1)から符号なし 8 ビット整数への変換」を参照)し、R 成分を Proc_R、G 成分を Proc_G、B 成分を Proc_B、A 成分を Proc_A にそれぞれ設定したものです。

差分値は、glTexImage1D() でロードする RGBA それぞれの参照テーブルの i + 256 番目のデータをパックした値を書き込みます。書き込む値は、成分ごとに小数部 7 ビットの符号つき 8 ビット固定小数点数(負の値は 2 の補数表現)に変換(変換方法については「8.9.5. 小数部 7 ビットの符号つき 8 ビット固定小数点数への変換」を参照)し、R 成分を Proc_R、G 成分を Proc_G、B 成分を Proc_B、A 成分を Proc_A にそれぞれ設定したものです。データの個数は 256 個ですので、Proc_Index には 0 ～ 255 を指定することができます。

8.8.6.6. テクスチャ解像度設定レジスタ(0x0082, 0x0092, 0x009A)

テクスチャユニット 0 ～ 2 それぞれにロードされているテクスチャの解像度を設定するレジスタは以下のとおりです。

図 8-45. テクスチャ解像度設定レジスタ(0x0082, 0x0092, 0x009A)のビットレイアウト

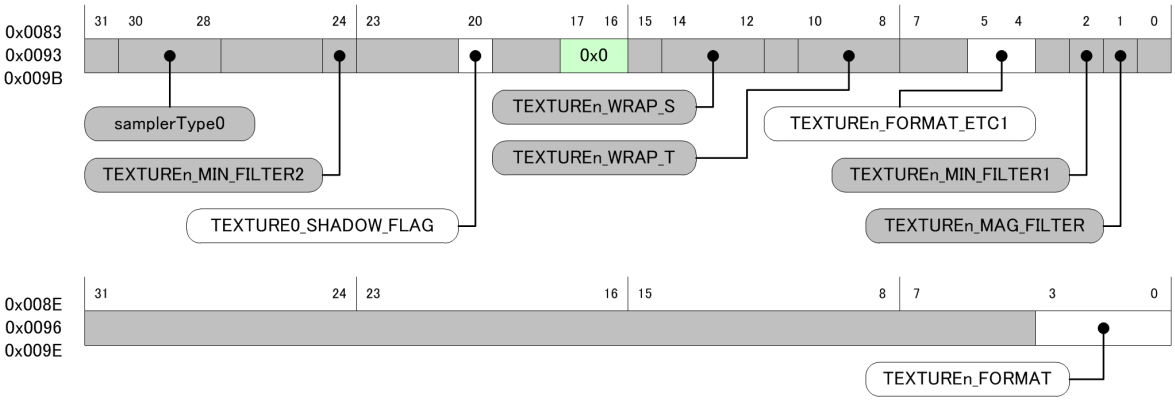


TEXTUREnWIDTH と TEXTUREn_HEIGHT には、それぞれテクスチャユニット n(n = 0 ～ 2)にロードされているテクスチャの幅と高さが設定されます。テクスチャユニット 0 の設定がレジスタ 0x0082 に対して、テクスチャユニット 1 の設定がレジスタ 0x0092 に対して、テクスチャユニット 2 の設定がレジスタ 0x009A に対してそれぞれ行われます。

8.8.6.7. テクスチャフォーマット設定レジスタ(0x0083, 0x008E, 0x0093, 0x0096, 0x009B, 0x009E)

テクスチャユニット 0 ～ 2 それぞれにロードされているテクスチャのフォーマットを設定するレジスタは以下のとおりです。

図 8-46. テクスチャフォーマット設定レジスタ(0x0083, 0x008E ほか)のビットレイアウト



ビットレイアウト中の名前は以下のようにテクスチャのフォーマット設定に対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。テクスチャユニット 0 の設定がレジスタ 0x0083 と 0x008E に対して、テクスチャユニット 1 の設定がレジスタ 0x0093 と 0x0096 に対して、テクスチャユニット 2 の設定がレジスタ 0x009B と 0x009E に対してそれぞれ行われます。下表にない名前はほかの設定で使用するビットです。

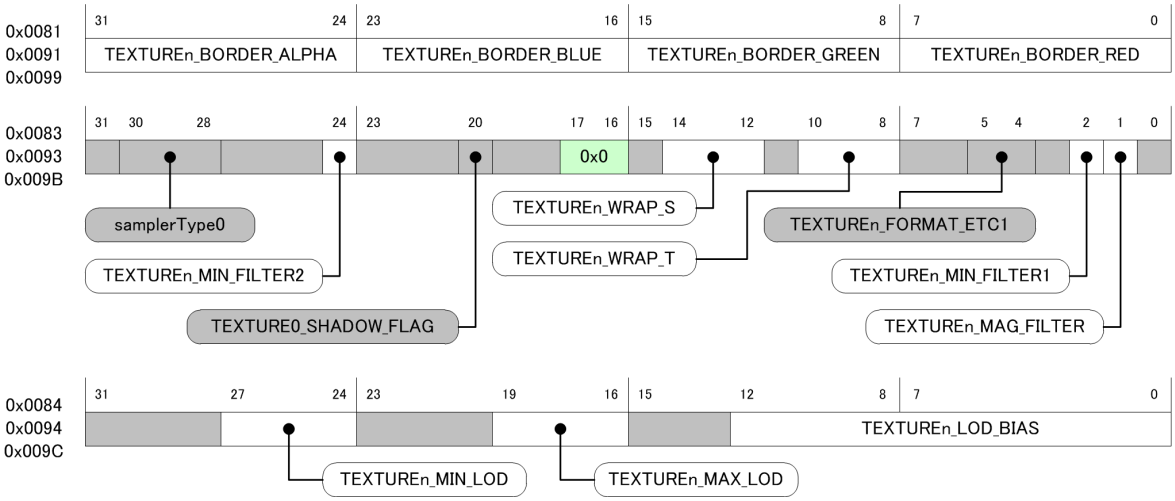
表 8-34. 名前と設定の対応(テクスチャフォーマット設定レジスタ)

名前	ビット数	説明
TEXTUREn_FORMAT_ETC1	2	テクスチャユニット n(n = 0 ～ 2)にロードされているテクスチャのフォーマットが GL_ETC1_RGB8_NATIVE_DMP であるかどうかのフラグです。 0x00 : GL_ETC1_RGB8_NATIVE_DMP 以外 0x02 : GL_ETC1_RGB8_NATIVE_DMP
TEXTURE0_SHADOW_FLAG	1	テクスチャユニット 0 にロードされているテクスチャのフォーマットが GL_SHADOW_DMP(GL_SHADOW_NATIVE_DMP)であるかどうかのフラグです。(レジスタ 0x0083 のみ) 0x0 : GL_SHADOW_DMP(GL_SHADOW_NATIVE_DMP)以外 0x1 : GL_SHADOW_DMP(GL_SHADOW_NATIVE_DMP)
TEXTUREn_FORMAT	4	テクスチャユニット n(n = 0 ～ 2)に対する glTexImage2D() の引数 *format* と *type* または glCompressedTexImage2D の引数 *internalformat* の設定です。 0x0 : GL_RGBA, GL_UNSIGNED_BYTE, GL_SHADOW_DMP, GL_UNSIGNED_INT, GL_GAS_DMP, GL_UNSIGNED_SHORT 0x1 : GL_RGB, GL_UNSIGNED_BYTE 0x2 : GL_RGBA, GL_UNSIGNED_SHORT_5_5_5_1 0x3 : GL_RGB, GL_UNSIGNED_SHORT_5_6_5 0x4 : GL_RGBA, GL_UNSIGNED_SHORT_4_4_4_4 0x5 : GL_LUMINANCE_ALPHA, GL_UNSIGNED_BYTE 0x6 : GL_HILO8_DMP, GL_UNSIGNED_BYTE 0x7 : GL_LUMINANCE, GL_UNSIGNED_BYTE 0x8 : GL_ALPHA, GL_UNSIGNED_BYTE 0x9 : GL_LUMINANCE_ALPHA, GL_UNSIGNED_BYTE_4_4_DMP 0xA : GL_LUMINANCE, GL_UNSIGNED_4BITS_DMP 0xB : GL_ALPHA, GL_UNSIGNED_4BITS_DMP 0xC : GL_ETC1_RGB8_NATIVE_DMP 0xD : GL_ETC1_ALPHA_RGB8_A4_NATIVE_DMP ※ ネイティブフォーマットは、上記の対応する非ネイティブフォーマットと同じ設定値を使用します。 ※ GL_RGB, GL_UNSIGNED_BYTE(0x1)をキューブマップで使用することはできません。

8.8.6.8. テクスチャパラメータ設定レジスタ(0x0081, 0x0083, 0x0084 ほか)

ラッピングモードやフィルタなど、テクスチャユニット 0 ～ 2 のテクスチャパラメータを設定するレジスタは以下のとおりです。

図 8-47. テクスチャパラメータ設定レジスタ(0x0081, 0x0083, 0x0084 ほか)のビットレイアウト



ビットレイアウト中の名前は以下のようにそれぞれの設定に対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。テクスチャユニット 0 の設定はレジスタ 0x0081、0x0083、0x0084 に対して、テクスチャユニット 1 の設定はレジスタ 0x0091、0x0093、0x0094 に対して、テクスチャユニット 2 の設定はレジスタ 0x0099、0x009B、0x009C に対してそれぞれ行います。下表にない名前はほかの設定で使用するビットです。

表 8-35. 名前と設定の対応(テクスチャパラメータ設定レジスタ)

名前	ビット数	説明
TEXTUREn_BORDER_RED TEXTUREn_BORDER_GREEN TEXTUREn_BORDER_BLUE TEXTUREn_BORDER_ALPHA	8	テクスチャユニット n(n = 0 ～ 2)のテクスチャボーダーカラー(GL_TEXTURE_BORDER_COLOR)の各成分値です。 成分ごとに 0.0 ～ 1.0 の範囲を 0 ～ 255 にマップしたときの符号なし 8 ビット整数に変換した値を設定します。値の変換方法については「8.9.17. 浮動小数点数(0 ～ 1)から符号なし 8 ビット整数への変換 2」を参照してください。
TEXTUREn_MAG_FILTER	1	テクスチャユニット n(n = 0 ～ 2)のテクスチャ拡大時のフィルタ設定(GL_TEXTURE_MAG_FILTER)です。 0x0 : GL_NEAREST 0x1 : GL_LINEAR
TEXTUREn_MIN_FILTER1 TEXTUREn_MIN_FILTER2	1	テクスチャユニット n(n = 0 ～ 2)のテクスチャ縮小時のフィルタ設定(GL_TEXTURE_MIN_FILTER)です。 2 つのビットの組み合わせ(FILTER1, FILTER2)は以下のとおりです。同じ組み合わせに複数のフィルタ設定がある場合は、最小・最大 LOD レベルの設定により決定されます。 (0x0, 0x0) : GL_NEAREST, GL_NEAREST_MIPMAP_NEAREST (0x0, 0x1) : GL_NEAREST_MIPMAP_LINEAR (0x1, 0x0) : GL_LINEAR, GL_LINEAR_MIPMAP_NEAREST (0x1, 0x1) : GL_LINEAR_MIPMAP_LINEAR
TEXTUREn_WRAP_S TEXTUREn_WRAP_T	3	テクスチャユニット n(n = 0 ～ 2)のラッピングモード設定(GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T)です。 0x0 : GL_CLAMP_TO_EDGE 0x1 : GL_CLAMP_TO_BORDER 0x2 : GL_REPEAT 0x3 : GL_MIRRORED_REPEAT

TEXTUREn_MIN_LOD	4	テクスチャユニット n(n = 0 ～ 2)の最小 LOD レベルの設定(GL_TEXTURE_MIN_LOD)です。 縮小時のフィルタ設定が GL_LINEAR や GL_NEAREST のように LOD を使用しない場合は 0 を設定します。LOD を使用する場合は GL_TEXTURE_MIN_LOD の値を設定します。ただし、0 以下の値は 0 を設定します。
TEXTUREn_MAX_LOD	4	テクスチャユニット n(n = 0 ～ 2)の最大 LOD レベルの設定です。 縮小時のフィルタ設定が GL_LINEAR や GL_NEAREST のように LOD を使用しない場合は 0 を設定します。LOD を使用する場合はテクスチャのロード時に指定されていたミップマップの段数 - 1 の値を設定します。
TEXTUREn_LOD_BIAS	13	テクスチャユニット n(n = 0 ～ 2)の LOD バイアス値(GL_TEXTURE_LOD_BIAS)です。 設定値を小数部が 8 ビットの符号つき 13 ビット固定小数点数(負の値は 2 の補数表現)に変換した値が設定されます。値の変換方法については「8.9.8. 小数部 8 ビットの符号つき 13 ビット固定小数点数への変換」を参照してください。

8.8.6.9. シャドウテクスチャ、ガステクスチャを使用する場合の設定

シャドウテクスチャ(GL_SHADOW_DMP または GL_SHADOW_NATIVE_DMP)を使用する場合は、GL_TEXTURE_WRAP_S と GL_TEXTURE_WRAP_T には 2D テクスチャならば GL_CLAMP_TO_BORDER を、キューブマップテクスチャならば GL_CLAMP_TO_EDGE を設定します。2D テクスチャ、キューブマップテクスチャに関係なく、GL_TEXTURE_MAG_FILTER と GL_TEXTURE_MIN_FILTER には GL_LINEAR を、レジスタ 0x0083 のビット[20 : 20](TEXTURE0_SHADOW_FLAG)には 1 を設定します。このビットはシャドウテクスチャ以外のフォーマットであるときには 0 を設定します。なお、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。

ガステクスチャ(GL_GAS_DMP または GL_GAS_NATIVE_DMP)を使用する場合は、GL_TEXTURE_WRAP_S と GL_TEXTURE_WRAP_T には GL_CLAMP_TO_EDGE を、GL_TEXTURE_MAG_FILTER と GL_TEXTURE_MIN_FILTER には GL_NEAREST を設定します。なお、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。

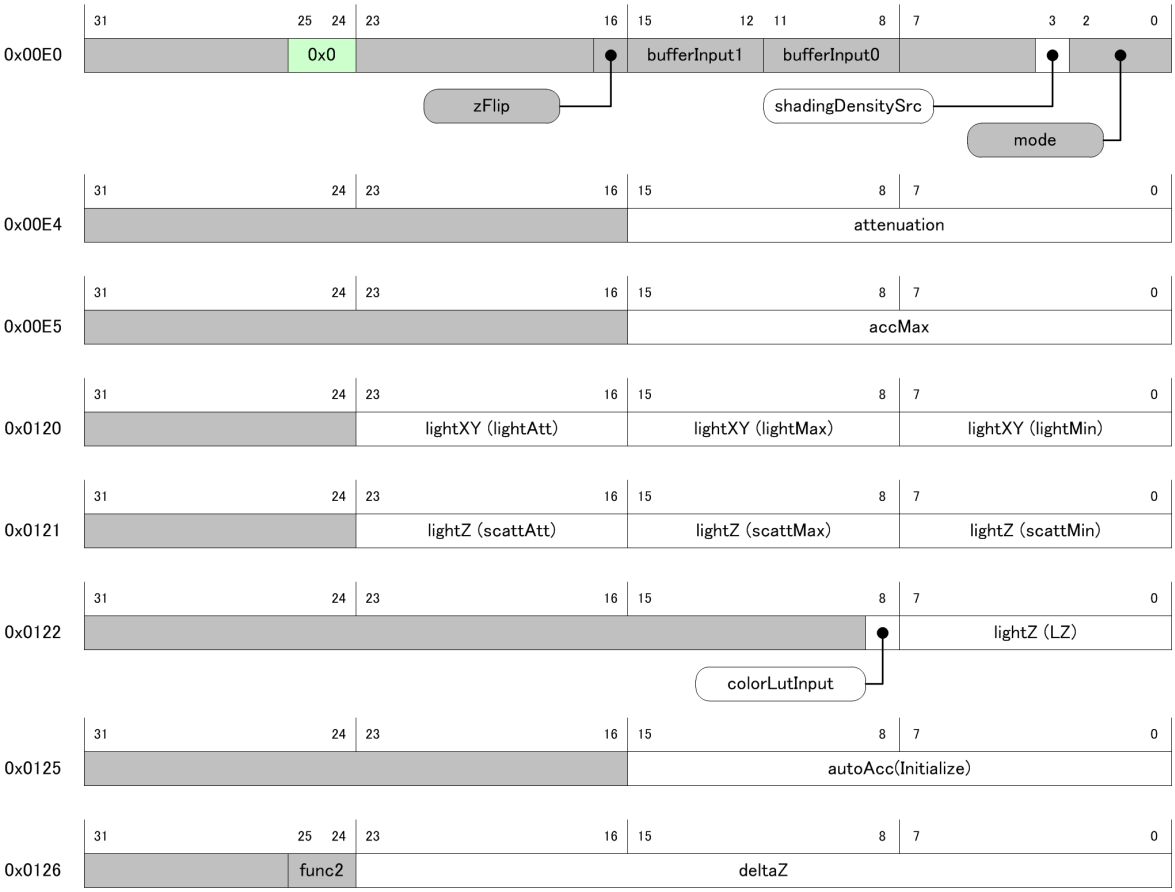
8.8.7. ガス設定レジスタ(0x00E0, 0x00E4, 0x00E5, 0x0120 ～ 0x0126)

ガス設定の予約ユニフォーム(名前に dmp_Gas を含む予約ユニフォーム)に関連するレジスタについて説明します。

8.8.7.1. ガス制御設定レジスタ(0x00E0, 0x00E4, 0x00E5, 0x0120 ～ 0x0122, 0x0125, 0x0126)

ガスの制御設定に対応するレジスタは以下のとおりです。レジスタ 0x00E0 の他のビットはフォグ設定などで使用されていることに注意してください。

図 8-48. ガス制御設定レジスタ(0x00E0, 0x00E4, 0x00E5, 0x0120 ～ 0x0122, 0x0126)のビットレイアウト



ビットレイアウト中の名前は以下のように予約ユニフォームに対応しています。下表では、それぞれのビットと設定値の対応についても説明しています。

表 8-36. 名前と予約ユニフォームの対応(ガス制御設定レジスタ)

名前	ビット数	説明
lightXY (lightMin) lightXY (lightMax) lightXY (lightAtt)	8	上から、dmp_Gas.lightXY に設定する値の第 1 ～第 3 要素です。 どの要素も、ユニフォームの値(0.0 ～ 1.0)を 0 ～ 255 にマップしたときの符号なし 8 ビット整数に変換した値です。値の変換方法については「8.9.16. 浮動小数点数(0 ～ 1)から符号なし 8 ビット整数への変換」を参照してください。
lightZ (scattMin) lightZ (scattMax) lightZ (scattAtt) lightZ (LZ)	8	上から、dmp_Gas.lightZ に設定する値の第 1 ～第 4 要素です。 どの要素も、ユニフォームの値(0.0 ～ 1.0)を 0 ～ 255 にマップしたときの符号なし 8 ビット整数に変換した値です。値の変換方法については「8.9.16. 浮動小数点数(0 ～ 1)から符号なし 8 ビット整数への変換」を参照してください。
deltaZ	24	dmp_Gas.deltaZ に設定する値です。 ユニフォームの値を小数部 8 ビットの符号なし 24 ビット固定小数点数に変換したものです。値の変換方法については「8.9.15. 小数部 8 ビットの符号なし 24 ビット固定小数点数への変換」を参照してください。
accMax	16	dmp_Gas.accMax に設定する値です。 ユニフォームの値を 16 ビット浮動小数点数に変換したものです。値の変換方法については「8.9.2. 16 ビット浮動小数点数への変換」を参照してください。

autoAcc(Initialize)	16	dmp_Gas.autoAcc に GL_TRUE を設定し、密度の最大値の逆数を自動的に計算させている場合は以下の手順が必要となります。 密度情報描画の前に 0 でクリアし、密度情報の描画が終わったときに nngxSetGasAutoAccumulationUpdate() を呼び出す。 詳細については表外の説明を参照してください。
attenuation	16	dmp_Gas.attenuation に設定する値です。 ユニフォームの値を 16 ビット浮動小数点数に変換したものです。値の変換方法については「8.9.2. 16 ビット浮動小数点数への変換」を参照してください。
colorLutInput	1	dmp_Gas.colorLutInput に設定する値です。 0x0 : GL_GAS_DENSITY_DMP 0x1 : GL_GAS_LIGHT_FACTOR_DMP
shadingDensitySrc	1	dmp_Gas.shadingDensitySrc に設定する値です。 0x0 : GL_GAS_PLAIN_DENSITY_DMP 0x1 : GL_GAS_DEPTH_DENSITY_DMP

密度情報の最大値の逆数を自動的に計算する場合

dmp_Gas.autoAcc に GL_TRUE を設定した際の機能は、ガスの密度情報描画パスで描画を行い、自動的に計算された密度情報 D1 の最大値の逆数を accMax に設定することで実現されます。

D1 の最大値は密度情報描画パスの開始前に 0 でクリアしておきます。クリアは autoAcc(Initialize) に 0(0 以外の値でクリアするときは、最大値を 16 ビット小数点数に変換して書き込んでください。値の変換方法については「8.9.2. 16 ビット浮動小数点数への変換」を参照してください)を書き込むことで行われます。密度情報描画パスが完了したあとは、算出された D1 の最大値を accMax に反映するために、nngxSetGasAutoAccumulationUpdate() を呼び出してください。

コード 8-22. nngxSetGasAutoAccumulationUpdate() の定義

```
void nngxSetGasAutoAccumulationUpdate (GLint id);
```

この関数は、バインド中のコマンドリストオブジェクトの *id* 番目に蓄積されたコマンドリクエストが終了したときの割り込みハンドラで、ガスの密度情報描画パスで計算された D1 の最大値を、dmp_Gas.accMax の予約ユニフォームによる設定(accMax)に反映(逆数を設定)します。

id で指定されたコマンドリクエストは 3D 実行コマンドでなければなりません。密度情報描画パスのコマンドを含むコマンドリクエストに対して呼び出しておくと、accMax は正しく更新されます。密度情報描画パスのコマンドとシェーディングパスのコマンドは nngxSplitDrawCmdlist() で区切ってください。これらのコマンドを同じコマンドリクエストに含めてしまうと、シェーディングパスの前に accMax が更新されません。また、この関数で更新したあとは、シェーディングパスが完了するまで accMax に値を書き込まないでください。

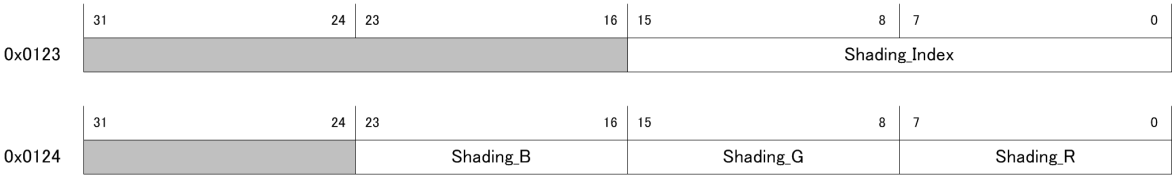
オブジェクト名が 0 のコマンドリストをバインドしている場合は GL_ERROR_806D_DMP のエラーを生成します。*id* が 0 以下の場合や蓄積済みのコマンドリクエストの個数より大きい場合、*id* に指定したコマンドリクエストが 3D 実行コマンドではない場合は GL_ERROR_806E_DMP のエラーを生成します。

8.8.7.2. シェーディング参照テーブル設定レジスタ(0x0123, 0x0124)

予約ユニフォーム dmp_Gas.sampler{ TR, TG, TB } で指定されるシェーディング参照テーブルは、8 個のデータと同数の差分値によって設定されます。

設定に使用するレジスタのビットレイアウトは以下のようになっています。

図 8-49. シェーディング参照テーブル設定レジスタ(0x0123, 0x0124)のビットレイアウト



レジスタ 0x0123 には、設定開始インデックスを Shading_Index に設定します。0 が最初のデータです。データの個数は 16 個ですので、Shading_Index には 0 ～ 15 を指定することができます。

インデックスをレジスタ 0x0123 に設定してから、レジスタ 0x0124 に変換した値の組み合わせを書き込みます。1 つデータを書き込むごとにインデックスが 1 インクリメントされますが、前半の 8 つと後半の 8 つで、レジスタに書き込む値のフォーマットが異なることや、**レジスタに設定するインデックスとシェーディング参照テーブルのインデックスが異なる**ことに注意が必要です。また、レジスタ 0x00E0 のビット [2 : 0](mode)に 0x7 を設定(フォグユニットをガスモードに設定)していなければ、レジスタ 0x0124 への書き込みは無視されます。

前半(i < 8)には、`glTexImage1D()` でロードするそれぞれの参照テーブルの i + 8 番目のデータをパックした値を書き込みます。書き込む値は、成分ごとに符号つき 8 ビット整数に変換(変換方法については「8.9.18. 浮動小数点数(-1 ～ 1)から符号つき 8 ビット整数への変換」を参照)し、R 成分を Shading_R、G 成分を Shading_G、B 成分を Shading_B にそれぞれ設定したものです。

後半(i >= 8)には、`glTexImage1D()` でロードするそれぞれの参照テーブルの i - 8 番目のデータをパックした値を書き込みます。書き込む値は、成分ごとに 255 を乗算した結果を小数部 0 ビットの符号なし 8 ビット固定小数点数に変換(変換方法については「8.9.11. 小数部 0 ビットの符号なし 8 ビット固定小数点数への変換」を参照)し、R 成分を Shading_R、G 成分を Shading_G、B 成分を Shading_B にそれぞれ設定したものです。

シェーディング参照テーブルを設定するコマンドの前に、ダミーコマンドを挿入しなければならない場合があります。レジスタ 0x0100 から 0x013F までの設定コマンド、レジスタ 0x0000 から 0x0035 までの設定コマンド、その他本文書にて記載のないレジスタアドレスへのコマンドは、これらのコマンドの後に続けてガスのシェーディング参照テーブルを設定する場合、間に 45 個のダミーコマンドが必要です。上記の範囲にあるレジスタ以外のレジスタへの設定コマンドがダミーコマンドとして使用できます。また、シェーディング参照テーブルを設定するコマンドのあとには、レジスタ 0x0100 に対してバイトイネーブルを 0 に設定したダミーコマンドが 1 つ必要です。

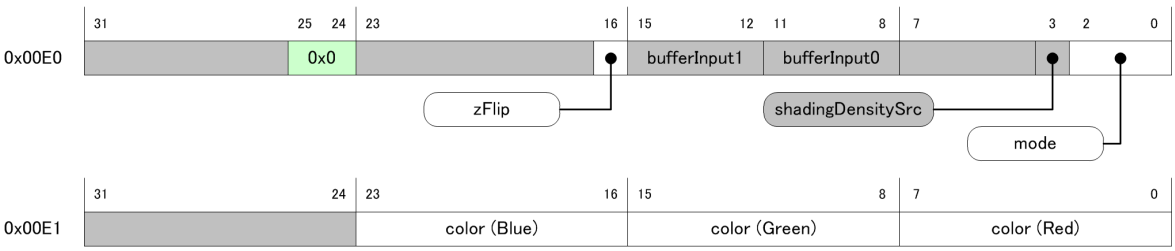
8.8.8. フォグ設定レジスタ(0x00E0, 0x00E1, 0x00E6, 0x00E8 ～ 0x00EF)

フォグ設定の予約ユニフォーム(名前に `dmp_Fog` を含む予約ユニフォーム)に関連するレジスタについて説明します。

8.8.8.1. フォグ制御設定レジスタ(0x00E0, 0x00E1)

フォグの制御設定に対応するレジスタは以下のとおりです。レジスタ 0x00E0 の他のビットはガス設定などで使用されていることに注意してください。

図 8-50. フォグ制御設定レジスタ(0x00E0, 0x00E1)のビットレイアウト



ビットレイアウト中の名前は以下のように予約ユニフォームに対応しています。下表では、それぞれのビット数と設定値の対

応についても説明しています。

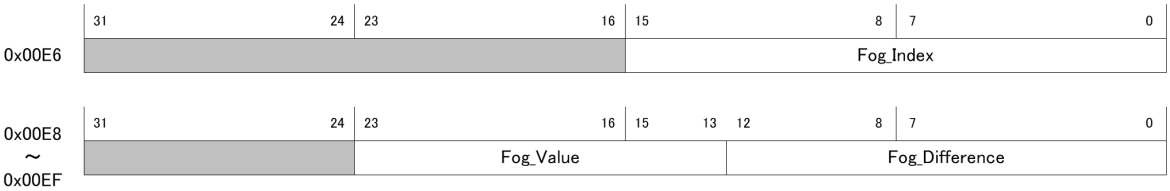
表 8-37. 名前と予約ユニフォームの対応(フォグ制御設定レジスタ)

名前	ビット数	説明
mode	3	dmp_Fog.mode に設定する値です。 0x0 : GL_FALSE 0x5 : GL_FOG 0x7 : GL_GAS_DMP
zFlip	1	dmp_Fog.zFlip に設定する値です。 0x0 : GL_FALSE 0x1 : GL_TRUE
color (Red) color (Green) color (Blue)	8	上から、dmp_Fog.color に設定する値の第 1 ～第 3 要素です。 どの要素も、ユニフォームの値(0.0 ～ 1.0)を 0 ～ 255 にマップしたときの符号なし 8 ビット整数に変換した値です。値の変換方法については「8.9.16. 浮動小数点数(0 ～ 1)から符号なし 8 ビット整数への変換」を参照してください。

8.8.8.2. フォグ参照テーブル設定レジスタ(0x00E6, 0x00E8 ～ 0x00EF)

予約ユニフォーム dmp_Fog.sampler で指定されるフォグ係数の参照テーブルは、128 個のデータと同数の差分値によって設定されます。設定に使用するレジスタのビットレイアウトは以下のようになっています。

図 8-51. フォグ参照テーブル設定レジスタ(0x00E6, 0x00E8 ～ 0x00EF)のビットレイアウト



レジスタ 0x00E6 には、設定開始インデックスを Fog_Index に設定します。インデックスは 0 が最初のデータ、127 が最後のデータです。

レジスタ 0x00E8 ～ 0x00EF には、glTexImage1D() でロードする参照テーブルの i 番目のデータと i + 128 番目に設定した差分値を組み合わせた値を書き込みます。Fog_Value にはデータを小数部 11 ビットの符号なし 11 ビット固定小数点数に変換した値を、Fog_Difference には差分値を小数部 11 ビットの符号つき 13 ビット固定小数点数(負の値は 2 の補数表現)に変換した値を、それぞれ設定してください。それぞれの値の変換方法については「8.9.12. 小数部 11 ビットの符号なし 11 ビット固定小数点数への変換」と「8.9.9. 小数部 11 ビットの符号つき 13 ビット固定小数点数への変換」を参照してください。

インデックスをレジスタ 0x00E6 に設定してから、0x00E8 ～ 0x00EF のいずれかのレジスタに変換した値の組み合わせを書き込みます。いずれのレジスタに書き込んでも処理結果は変わらず、1 つデータを書き込むごとにインデックスが 1 インクリメントされます。

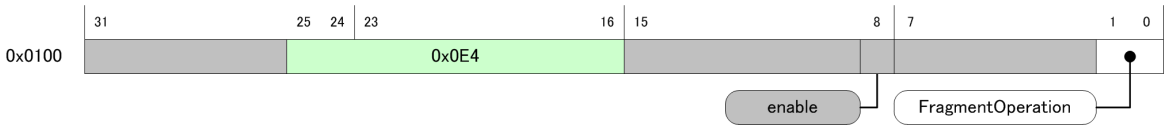
8.8.9. フラグメントオペレーション設定レジスタ(0x0100 ほか)

フラグメントオペレーション設定の予約ユニフォーム(名前に dmp_FragOperation を含む予約ユニフォーム)に関連するレジスタについて説明します。

8.8.9.1. フラグメントオペレーションモード設定レジスタ(0x0100)

フラグメントオペレーションのモード設定に対応するレジスタは以下のとおりです。このレジスタのほかのビットには論理演算とブレンディングの設定が行われることに注意してください。フラグメントオペレーションのモード設定を変更したときは、「8.8.9.6. フレームバッファアクセス制御設定レジスタ(0x0112 ～ 0x0115)」の設定も変更しなければなりません。

図 8-52. フラグメントオペレーションモード設定レジスタ(0x0100)のビットレイアウト



ビットレイアウト中の名前は以下のように予約ユニフォームに対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

表 8-38. 名前と予約ユニフォームの対応(フラグメントオペレーションモード設定レジスタ)

名前	ビット数	説明
FragmentOperation	2	dmp_FragOperation.mode に設定する値です。 0x0 : GL_FRAGOP_MODE_GL_DMP 0x1 : GL_FRAGOP_MODE_GAS_ACC_DMP 0x3 : GL_FRAGOP_MODE_SHADOW_DMP

8.8.9.2. シャドウ減衰ファクタ設定レジスタ(0x0130)

シャドウ減衰ファクタの設定に対応するレジスタは以下のとおりです。

図 8-53. シャドウ減衰ファクタ設定レジスタ(0x0130)のビットレイアウト

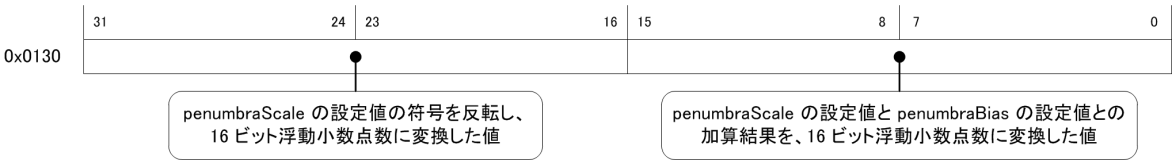


表 8-39. シャドウ減衰ファクタ設定の予約ユニフォームとレジスタの対応

レジスタ	ビット数	説明
0x0130 の ビット [31 : 16]	16	dmp_FragOperation.penumbraScale の設定値の**符号を反転**し、16 ビット浮動小数点数に変換した値を設定します。 値の変換方法については「8.9.2. 16 ビット浮動小数点数への変換」を参照してください。
0x0130 の ビット [15 : 0]	16	dmp_FragOperation.penumbraScale の設定値と dmp_FragOperation.penumbraBias の設定値との加算結果を、16 ビット浮動小数点数に変換して設定します。 値の変換方法については「8.9.2. 16 ビット浮動小数点数への変換」を参照してください。

8.8.9.3. w バッファ設定レジスタ(0x004D, 0x004E, 0x006D)

w バッファの設定に対応するレジスタは以下のとおりです。

図 8-54. wバッファ設定レジスタ(0x004D、0x004E、0x006D)のビットレイアウト

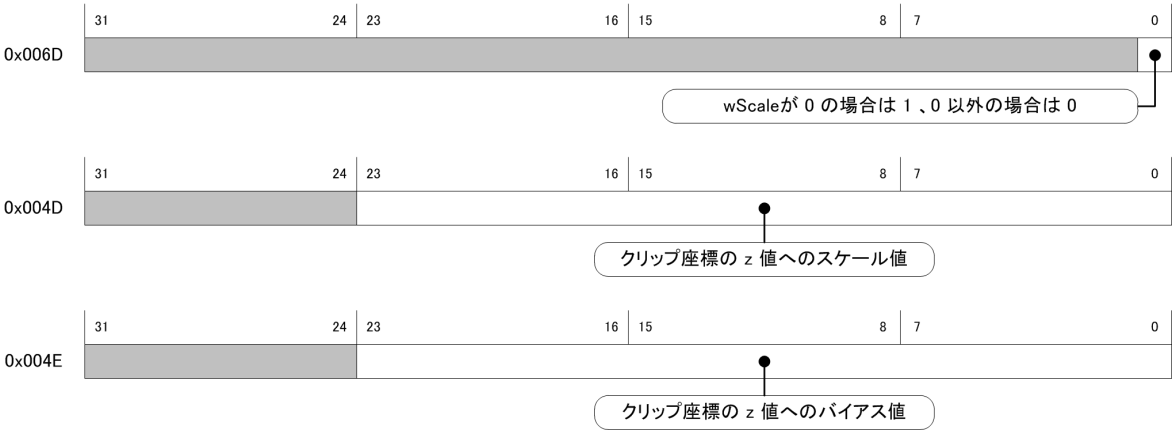


表 8-40. w バッファ設定の予約ユニフォームとレジスタの対応

レジスタ	ビット数	説明
0x006D の ビット [0 : 0]	1	dmp_FragOperation.wScale の設定値が 0 の場合は 1を、0 以外の場合は 0 を設定します。
0x004D の ビット [23 : 0]	24	クリップ座標の z 値へのスケール値を設定します。設定値には、dmp_FragOperation.wScale の設定値と glDepthRangef() の設定が影響します。設定方法については後述します。
0x004E の ビット [23 : 0]	24	クリップ座標の z 値へのバイアス値を設定します。設定値には、dmp_FragOperation.wScale の設定値と glDepthRangef()、glPolygonOffset() の設定が影響します。設定方法については後述します。

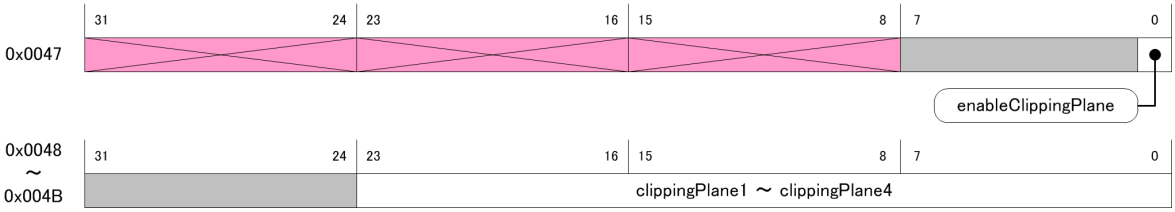
レジスタ 0x004D のビット [23 : 0] には、dmp_FragOperation.wScale の設定値が 0 以外の場合は dmp_FragOperation.wScale の設定値の符号を反転した値を設定します。dmp_FragOperation.wScale の設定値が 0 の場合は、glDepthRangef() で指定した *zNear* と *zFar* を使用して(*zNear* - *zFar*)の計算結果を設定します。レジスタに設定する際には、24 ビット浮動小数点数に変換した値を使用してください。値の変換方法については「8.9.1. 24 ビット浮動小数点数への変換」を参照してください。

レジスタ 0x004E のビット [23 : 0] には、dmp_FragOperation.wScale の設定値が 0 以外の場合は 0 を設定します。dmp_FragOperation.wScale の設定値が 0 の場合は、glDepthRangef() で指定した引数 *zNear* を設定します。glEnable() で GL_POLYGON_OFFSET_FILL を有効にしていた場合は、glPolygonOffset() で指定した *units* を使用して計算したオフセット値を加算して設定します。オフセット値には、デプスバッファのフォーマットが 16 ビットの場合は *units* ÷ 65535 の計算結果を、デプスバッファのフォーマットが 24 ビットの場合は *units* ÷ 16777215 の計算結果を適用してください。レジスタに設定する際には、24 ビット浮動小数点数に変換した値を使用してください。値の変換方法については「8.9.1. 24 ビット浮動小数点数への変換」を参照してください。

8.8.9.4. クリッピング設定レジスタ(0x0047 ～ 0x004B)

クリッピングの設定に対応するレジスタは以下のとおりです。

図 8-55. クリッピング設定レジスタ(0x0047 ～ 0x004B)のビットレイアウト



ビットレイアウト中の名前は以下のように予約ユニフォームに対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

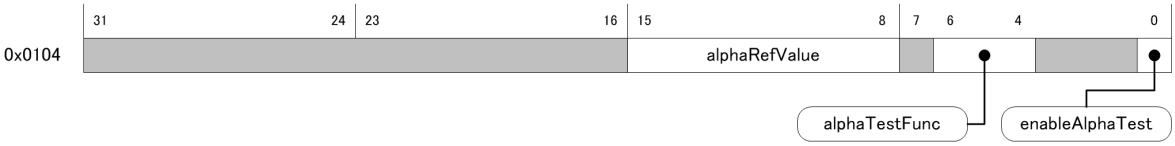
表 8-41. 名前と予約ユニフォームの対応(クリッピング設定レジスタ)

名前	ビット数	説明
enableClippingPlane	1	dmp_FragOperation.enableClippingPlane に設定する値です。 0x0 : GL_FALSE 0x1 : GL_TRUE
clippingPlane1 ～ clippingPlane4	24	dmp_FragOperation.clippingPlane に設定する値の第 1 ～第 4 要素のそれぞれを、符号つき 24 ビット浮動小数点数に変換した値です。 値の変換方法については「8.9.1. 24 ビット浮動小数点数への変換」を参照してください。

8.8.9.5. アルファテスト設定レジスタ(0x0104)

アルファテストの設定に対応するレジスタは以下のとおりです。

図 8-56. アルファテスト設定レジスタ(0x0104)のビットレイアウト



ビットレイアウト中の名前は以下のように予約ユニフォームに対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

表 8-42. 名前と予約ユニフォームの対応(アルファテスト設定レジスタ)

名前	ビット数	説明
enableAlphaTest	1	dmp_FragOperation.enableAlphaTest に設定する値です。 0x0 : GL_FALSE 0x1 : GL_TRUE
alphaTestFunc	3	dmp_FragOperation.alphaTestFunc に設定する値です。 0x0 : GL_NEVER 0x1 : GL_ALWAYS 0x2 : GL_EQUAL 0x3 : GL_NOTEQUAL 0x4 : GL_LESS 0x5 : GL_LEQUAL 0x6 : GL_GREATER 0x7 : GL_GEQUAL

alphaRefValue	8	dmp_FragOperation.alphaRefValue に設定する値(0.0 ～ 1.0)を 0 ～ 255 にマップしたときの符号なし 8 ビット整数に変換した値です。 値の変換方法については「8.9.16. 浮動小数点数(0 ～ 1)から符号なし 8 ビット整数への変換」を参照してください。

8.8.9.6. フレームバッファアクセス制御設定レジスタ(0x0112 ～ 0x0115)

フレームバッファアクセスの制御設定に対応するレジスタは以下のとおりです。これらのレジスタは、いくつかの関数の呼び出しや予約ユニフォームの設定値変更を行ったときに、併せて変更しなければならない場合があります。

図 8-57. フレームバッファアクセス制御設定レジスタ(0x0112 ～ 0x0115)のビットレイアウト

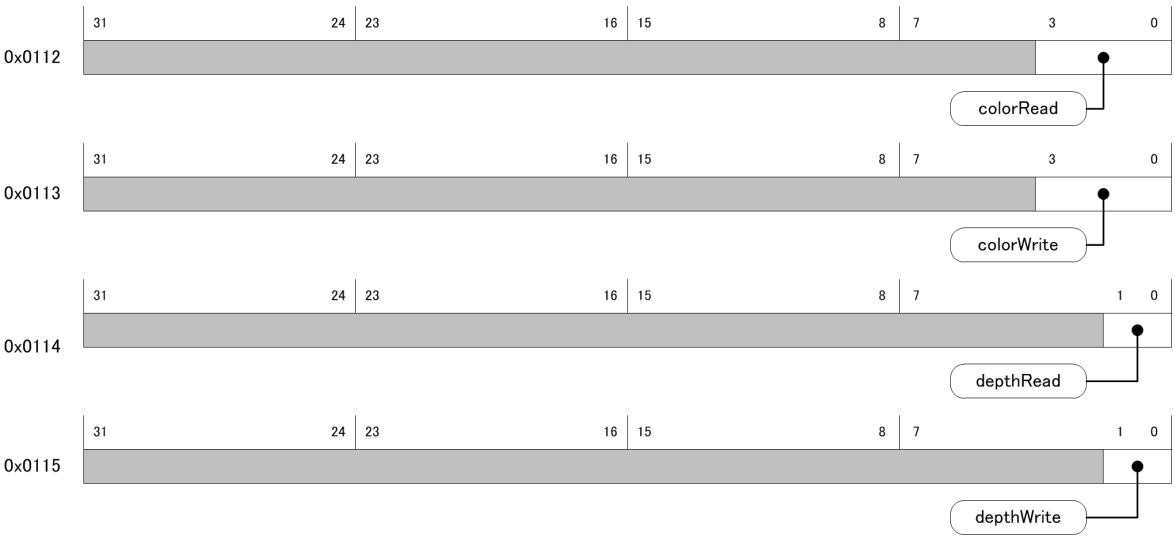


表 8-43. フレームバッファアクセス制御設定の予約ユニフォームとレジスタの対応

レジスタ	ビット数	説明
colorRead 0x0112 の ビット [3 : 0]	4	カラーバッファのリードが必要な場合には 0x0F を設定し、不要な場合には 0 を設定します。 以下のいずれかの条件に合致する場合、カラーバッファのリードが必要となります。 ● 予約ユニフォーム dmp_FragOperation.mode に GL_FRAGOP_MODE_GL_DMP 以外の値が設定されている。 ● glColorMask() により、1 つ以上の成分が書き込み許可になっており、かつ glEnable() によって GL_BLEND が有効になっている。さらに、ブレンド時に DST カラーを参照するブレンドファクターが glBlendFunc() または glBlendFuncSeparate() で選択されている。 ● glColorMask() により、1 つ以上の成分が書き込み許可になっており、かつ glEnable() によって GL_COLOR_LOGIC_OP が有効になっている。さらに、DST カラーを参照する論理演算が glLogicOp() で選択されている。 ● glColorMask() により、1 つ以上の成分が書き込み許可になっており、かつ 1 つ以上の成分が書き込み禁止になっている。

colorWrite 0x0113 の ビット [3 : 0]	4	カラーバッファのライトが必要な場合には 0x0F を設定し、不要な場合には 0 を設定します。 以下のいずれかの条件に合致する場合、カラーバッファのライトが必要となります。 ● 予約ユニフォーム dmp_FragOperation.mode に GL_FRAGOP_MODE_GL_DMP 以外の値が設定されている。 ● glColorMask() により、1 つ以上の成分が書き込み許可になっている。
depthRead 0x0114 の ビット [1 : 0]	2	デプスバッファのリードが必要な場合にはビット [1 : 1] に 1 を設定し、ステンシルバッファのリードが必要な場合にはビット [0 : 0] に 1 を設定します。不要な場合には 0 を設定します。 以下のいずれかの条件に合致する場合、デプスバッファのリードが必要となります。 ● 予約ユニフォーム dmp_FragOperation.mode に GL_FRAGOP_MODE_GAS_ACC_DMP が設定されている。 ● 予約ユニフォーム dmp_FragOperation.mode に GL_FRAGOP_MODE_GL_DMP が設定されており、かつ glEnable() により GL_DEPTH_TEST が有効、かつ glDepthMask() に GL_TRUE が設定されている。 ● 予約ユニフォーム dmp_FragOperation.mode に GL_FRAGOP_MODE_GL_DMP が設定されており、かつ glEnable() により GL_DEPTH_TEST が有効、かつ glColorMask() により 1 つ以上の成分が書き込み許可になっている。 以下のいずれかの条件に合致する場合、ステンシルバッファのリードが必要となります。 ● 予約ユニフォーム dmp_FragOperation.mode に GL_FRAGOP_MODE_GAS_ACC_DMP が設定されている。 ● 予約ユニフォーム dmp_FragOperation.mode に GL_FRAGOP_MODE_GL_DMP が設定されており、かつ glEnable() により GL_STENCIL_TEST が有効、かつ glStencilMask() に 0 以外の値が設定されている。 ● 予約ユニフォーム dmp_FragOperation.mode に GL_FRAGOP_MODE_GL_DMP が設定されており、かつ glEnable() により GL_STENCIL_TEST が有効、かつ glColorMask() により 1 つ以上の成分が書き込み許可になっている。

depthWrite 0x0115 の ビット [1 : 0]	2	デプスバッファのライトが必要な場合にはビット [1 : 1] に 1 を設定し、ステンシルバッファのライトが必要な場合にはビット [0 : 0] に 1 を設定します。不要な場合には 0 を設定します。 以下の条件を満たす場合、デプスバッファのライトが必要となります。 ● 予約ユニフォーム dmp_FragOperation.mode に GL_FRAGOP_MODE_GL_DMP が設定されており、かつ glEnable() により GL_DEPTH_TEST が有効、かつ glDepthMask() に GL_TRUE が設定されている。 以下の条件を満たす場合、ステンシルバッファのライトが必要となります。 ● 予約ユニフォーム dmp_FragOperation.mode に GL_FRAGOP_MODE_GL_DMP が設定されており、かつ glEnable() により GL_STENCIL_TEST が有効、かつ glStencilMask() に 0 以外の値が設定されている。

カラーバッファ、デプスバッファ、ステンシルバッファのリードおよびライトの有無の組み合わせの中には、ハードウェアでサポートされていない組み合わせがあります。サポートされていない組み合わせで設定された場合の動作は不定です。サポートされている組み合わせについては下記の表を参照してください。

表 8-44. ハードウェアでサポートされている組み合わせ

colorRead	colorWrite	depthRead	depthWrite	ハードウェアのサポート
0	0	0	0	×
0 以外	0	0	0	×
0	0 以外	0	0	○
0 以外	0 以外	0	0	○
0	0	0 以外	0	×
0 以外	0	0 以外	0	×
0	0 以外	0 以外	0	○
0 以外	0 以外	0 以外	0	○
0	0	0	0 以外	×
0 以外	0	0	0 以外	×
0	0 以外	0	0 以外	×
0 以外	0 以外	0	0 以外	×
0	0	0 以外	0 以外	○
0 以外	0	0 以外	0 以外	×
0	0 以外	0 以外	0 以外	○
0 以外	0 以外	0 以外	0 以外	○

上記のレジスタに 0 が設定されている場合、対応するバッファへのメモリアクセスが抑制されることによるパフォーマンスの向上に効果があります。そのため、なるべくこれらのレジスタに 0 を設定することが望まれますが、ハードウェアでサポートされ

ている組み合わせでなければならないことに注意が必要です。また、バッファのライトが発生するフラグメント処理の設定であるにもかかわらずライトアクセスをオフにした場合はバッファへの書き込みが行われず、バッファのリードが発生する設定であるにもかかわらずリードアクセスをオフにした場合は読み出される値が不定となります。

カラーバッファのリードアクセスをオフに設定することができるのは、

- 予約ユニフォーム dmp_FragOperation.mode に GL_FRAGOP_MODE_GL_DMP が設定されている。
- カラーマスク設定レジスタ(0x0107)のビット[11 : 8]が 0x0 または 0xF に設定されている。

上記の条件をすべて満たしつつ、下記の条件のいずれかを満たす場合です。

- ブレンディングが有効(レジスタ 0x0100 のビット[8 : 8]が 1)で、srcRGB と srcAlpha の設定(レジスタ 0x0101 のビット[19 : 16]、[27 : 24])がデスティネーションカラーを必要としない(0x4、0x5、0x8、0x9、0xE 以外)。かつ、dstRGB と dstAlpha の設定(レジスタ 0x0101 のビット[23 : 20]、[31 : 28])が 0x0(GL_ZERO)。
- 論理演算が有効(レジスタ 0x0100 のビット[8 : 8]が 0)で、レジスタ 0x0102 の opcode の設定がデスティネーションカラーを必要としない(0x0、0x3、0x4、0x5 のいずれか)。

カラーバッファのライトアクセスをオフに設定することができるのは、下記の条件をすべて満たす場合です。

- 予約ユニフォーム dmp_FragOperation.mode に GL_FRAGOP_MODE_GL_DMP が設定されている。
- カラーマスク設定レジスタ(0x0107)のビット[11 : 8]が 0x0 に設定されている。

デプスバッファのリードアクセスをオフに設定することができるのは、

- 予約ユニフォーム dmp_FragOperation.mode に GL_FRAGOP_MODE_GL_DMP が設定されている。
- デプステストが無効、もしくは有効であっても比較方法にデプスバッファの値を必要としない。

上記の条件をすべて満たすか、下記の条件を満たす場合です。

- 予約ユニフォーム dmp_FragOperation.mode に GL_FRAGOP_MODE_SHADOW_DMP が設定されている。

デプスバッファのライトアクセスをオフに設定することができるのは、下記の条件のいずれかを満たす場合です。

- デプステスト(レジスタ 0x0107 のビット[0 : 0])が無効(0x0)に設定されている。
- デプスバッファのマスキング設定(レジスタ 0x0107 のビット[12 : 12])が GL_FALSE(0x0)に設定されている。
- 予約ユニフォーム dmp_FragOperation.mode に GL_FRAGOP_MODE_GL_DMP 以外が設定されている。

ステンシルバッファのリードアクセスをオフに設定することができるのは、

- 予約ユニフォーム dmp_FragOperation.mode に GL_FRAGOP_MODE_SHADOW_DMP が設定されている。

上記の条件を満たすか、予約ユニフォーム dmp_FragOperation.mode に GL_FRAGOP_MODE_GL_DMP が設定されている場合に、下記の条件のいずれかを満たす場合です。

- ステンシルテスト(レジスタ 0x0105 のビット[0 : 0])が無効(0x0)に設定されている。
- ステンシルテスト(レジスタ 0x0105 のビット[0 : 0])が有効(0x1)であっても比較方法にステンシルバッファの値を必要としない。
- ステンシルテストのマスキング値(レジスタ 0x0105 のビット[31 : 24])が 0x00 に設定されている。

ステンシルバッファのライトアクセスをオフに設定することができるのは、下記の条件のいずれかを満たす場合です。

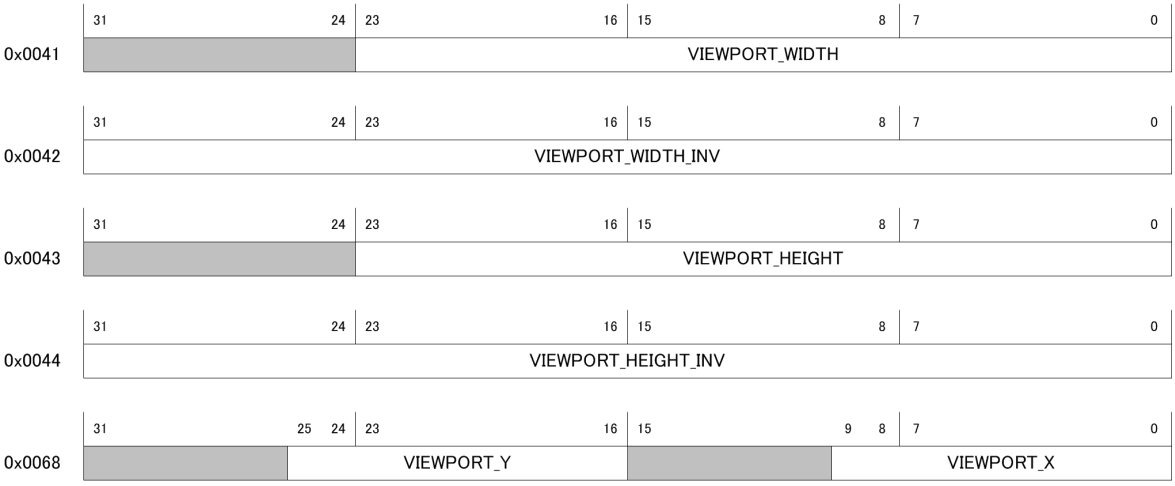
- ステンシルテスト(レジスタ 0x0105 のビット[0 : 0])が無効(0x0)に設定されている。

- ステンシルテストのマスク設定(レジスタ 0x0105 のビット [15 : 8])が 0x00 に設定されている。
- ステンシルテストの fail、zfail、zpass(レジスタ 0x0106 のビット [2 : 0]、[6 : 4]、[10 : 8])が、すべて GL_KEEP (0x0)に設定されている。
- 予約ユニフォーム dmp_FragOperation.mode に GL_FRAGOP_MODE_GL_DMP 以外が設定されている。

8.8.10. ビューポート設定レジスタ(0x0041 ～ 0x0044, 0x0068)

glViewport() で行われるビューポートの設定に対応するレジスタは以下のとおりです。これらのレジスタを設定したときは、「8.8.16. シザーテスト設定レジスタ(0x0065 ～ 0x0067)」も併せて変更しなければならない場合があります。

図 8-58. ビューポート設定レジスタ(0x0041 ～ 0x0044, 0x0068)のビットレイアウト



ビットレイアウト中の名前は以下のようにビューポートの設定に対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

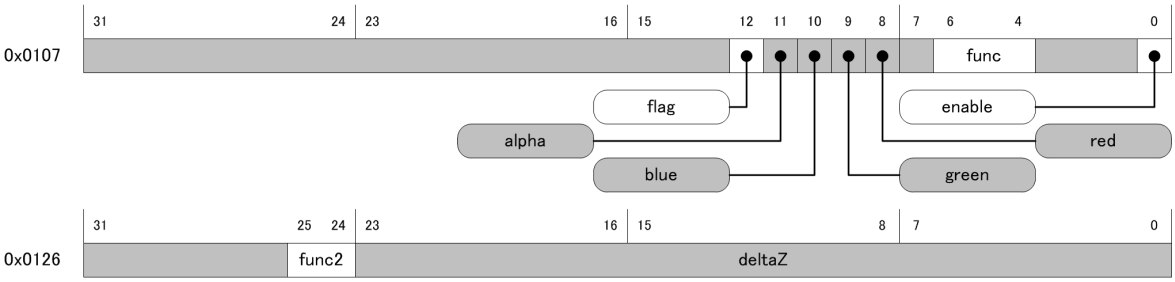
表 8-45. 名前と設定の対応(ビューポート設定レジスタ)

名前	ビット数	説明
VIEWPORT_WIDTH	24	ビューポートの幅を 2 で除算した結果を 24 ビット浮動小数点数に変換した値です。 値の変換方法については「8.9.1. 24 ビット浮動小数点数への変換」を参照してください。
VIEWPORT_WIDTH_INV	32	2 をビューポートの幅で除算した結果を 31 ビット浮動小数点数に変換した値を 1 ビット左へシフトした値です。 値の変換方法については「8.9.3. 31 ビット浮動小数点数への変換」を参照してください。
VIEWPORT_HEIGHT	24	ビューポートの高さを 2 で除算した結果を 24 ビット浮動小数点数に変換した値です。 値の変換方法については「8.9.1. 24 ビット浮動小数点数への変換」を参照してください。
VIEWPORT_HEIGHT_INV	32	2 をビューポートの高さで除算した結果を 31 ビット浮動小数点数に変換した値を 1 ビット左へシフトした値です。 値の変換方法については「8.9.3. 31 ビット浮動小数点数への変換」を参照してください。
VIEWPORT_X	10	ビューポートの始点(x 座標)をそのまま設定します。
VIEWPORT_Y	10	ビューポートの始点(y 座標)をそのまま設定します。

8.8.11. デプステスト設定レジスタ(0x0107, 0x0126)

デプステストの設定に対応するレジスタは以下のとおりです。これらのレジスタを設定したときは、「8.8.9.6. フレームバッファアクセス制御設定レジスタ(0x0112 ～ 0x0115)」も併せて変更しなければならない場合があります。

図 8-59. デプステスト設定レジスタ(0x0107)のビットレイアウト



ビットレイアウト中の名前は以下のようにデプステストの設定に対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。表中にない名前は、ほかの設定で使用するビットです。

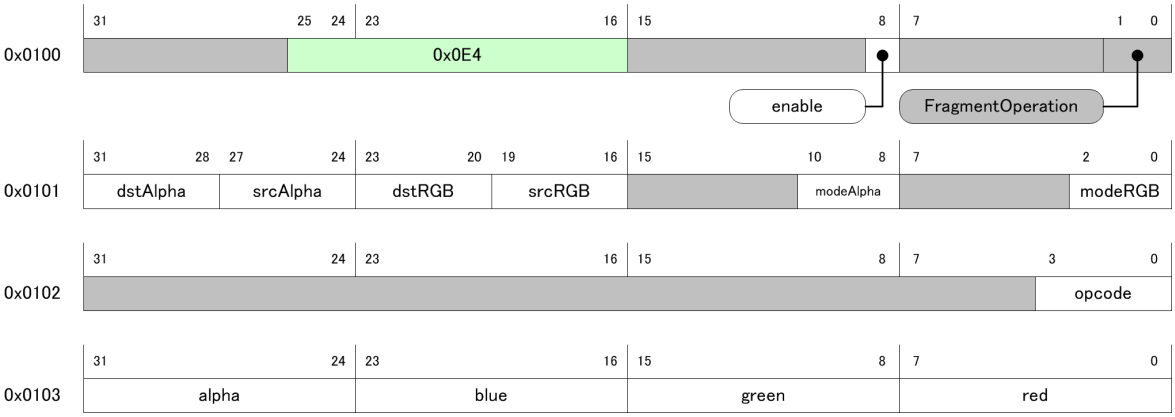
表 8-46. 名前と設定の対応(デプステスト設定レジスタ)

名前	ビット数	説明
enable	1	glEnable / glDisable に GL_DEPTH_TEST を渡して変更した、デプステストの有効・無効を設定します。 0x0 : デプステスト無効 0x1 : デプステスト有効
func	3	glDepthFunc() で変更した、デプステストの比較方法の設定です。 0x0 : GL_NEVER 0x1 : GL_ALWAYS 0x2 : GL_EQUAL 0x3 : GL_NOTEQUAL 0x4 : GL_LESS 0x5 : GL_LEQUAL 0x6 : GL_GREATER 0x7 : GL_GEQUAL
flag	1	glDepthMask() で変更した、デプスバッファへのマスキング設定です。 0x0 : GL_FALSE 0x1 : GL_TRUE
func2	2	glDepthFunc() で変更した、デプステストの比較方法の設定です。ただし、この設定は通常のデプステストには影響せず、ガスレンダリングの密度情報描画パスにおける密度情報 D2 の計算に影響を与えます。 0x0 : GL_NEVER 0x1 : GL_ALWAYS 0x2 : GL_GREATER または GL_GEQUAL 0x3 : 上記以外

8.8.12. 論理演算とブレンディング設定レジスタ(0x0100 ～ 0x0103)

論理演算とブレンディングの設定に対応するレジスタは以下のとおりです。これらのレジスタを設定したときは、「8.8.9.6. フレームバッファアクセス制御設定レジスタ(0x0112 ～ 0x0115)」も併せて変更しなければならない場合があります。

図 8-60. 論理演算とブレンディング設定レジスタ(0x0100 ～ 0x0103)のビットレイアウト



ビットレイアウト中の名前は以下のように論理演算とブレンディングの設定に対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

表 8-47. 名前と設定の対応(論理演算とブレンディング設定レジスタ)

名前	ビット数	説明
enable	1	glEnable() / glDisable() に GL_BLEND または GL_LOGIC_OP を渡して変更した結果、論理演算またはブレンディングのどちらが有効になっているかを設定します。両方が有効に設定されている場合は論理演算が優先され、両方が無効に設定されている場合はブレンディングが優先されます。 0x0：論理演算が有効 0x1：ブレンディングが有効
srcRGB dstRGB srcAlpha dstAlpha	4	glBlendFunc()、glBlendFuncSeparate() で変更した、ソースとディスティネーションの重み係数の設定です。 ブレンディングが無効である場合、srcRGB と srcAlpha には 0x1 を、dstRGB と dstAlpha には 0x0 を設定します。 ブレンディングが有効である場合は以下の値を設定します。 0x0：GL_ZERO 0x1：GL_ONE 0x2：GL_SRC_COLOR 0x3：GL_ONE_MINUS_SRC_COLOR 0x4：GL_DST_COLOR 0x5：GL_ONE_MINUS_DST_COLOR 0x6：GL_SRC_ALPHA 0x7：GL_ONE_MINUS_SRC_ALPHA 0x8：GL_DST_ALPHA 0x9：GL_ONE_MINUS_DST_ALPHA 0xA：GL_CONSTANT_COLOR 0xB：GL_ONE_MINUS_CONSTANT_COLOR 0xC：GL_CONSTANT_ALPHA 0xD：GL_ONE_MINUS_CONSTANT_ALPHA 0xE：GL_SRC_ALPHA_SATURATE
modeRGB modeAlpha	3	glBlendEquation()、glBlendEquationSeparate() で変更した、ブレンディングの計算式の設定です。 ブレンディングが無効である場合、0x0 を設定します。 ブレンディングが有効である場合は以下の値を設定します。 0x0：GL_FUNC_ADD 0x1：GL_FUNC_SUBTRACT 0x2：GL_FUNC_REVERSE_SUBTRACT 0x3：GL_MIN 0x4：GL_MAX

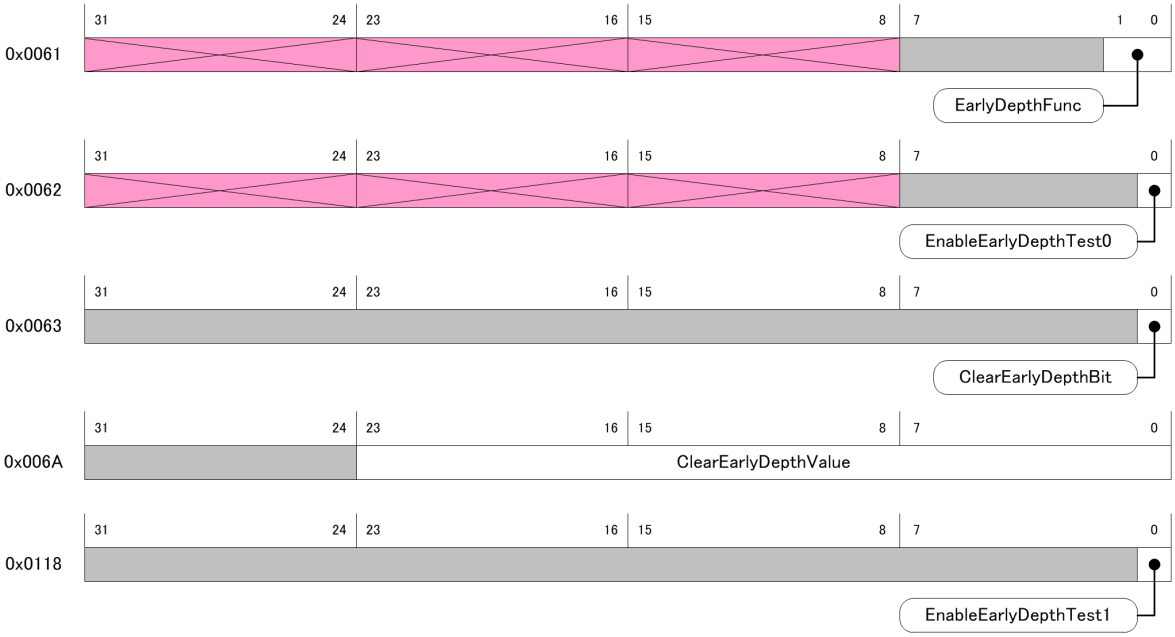
red green blue alpha	8	上から、glBlendColor() で設定した定数カラーの赤、緑、青、アルファの成分です。 どの成分も、値(0.0 ～ 1.0)を 0 ～ 255 にマップしたときの符号なし 8 ビット整数に変換した値です。値の変換方法については「8.9.16. 浮動小数点数(0 ～ 1)から符号なし 8 ビット整数への変換」を参照してください。
opcode	4	glLogicOp() で変更した、論理演算の演算方法の設定です。 0x0 : GL_CLEAR 0x1 : GL_AND 0x2 : GL_AND_REVERSE 0x3 : GL_COPY 0x4 : GL_SET 0x5 : GL_COPY_INVERTED 0x6 : GL_NOOP 0x7 : GL_INVERT 0x8 : GL_NAND 0x9 : GL_OR 0xA : GL_NOR 0xB : GL_XOR 0xC : GL_EQUIV 0xD : GL_AND_INVERTED 0xE : GL_OR_REVERSE 0xF : GL_OR_INVERTED

論理演算が有効になっている場合は、レジスタ 0x0101 の設定(srcXXX、dstXXX、modeXXX)は無視されます。

8.8.13. アーリーデプステスト設定レジスタ(0x0061 ～ 0x0063, 0x006A, 0x0118)

アーリーデプステストの設定に対応するレジスタは以下のとおりです。これらのレジスタを設定したときは、「8.8.9.6. フレームバッファアクセス制御設定レジスタ(0x0112 ～ 0x0115)」と「8.8.11. デプステスト設定レジスタ(0x0107, 0x0126)」も併せて変更しなければならない場合があります。

図 8-61. アーリーデプステスト設定レジスタ(0x0061 ～ 0x0063, 0x006A, 0x0118)のビットレイアウト



ビットレイアウト中の名前は以下のように論理演算とブレンディングの設定に対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

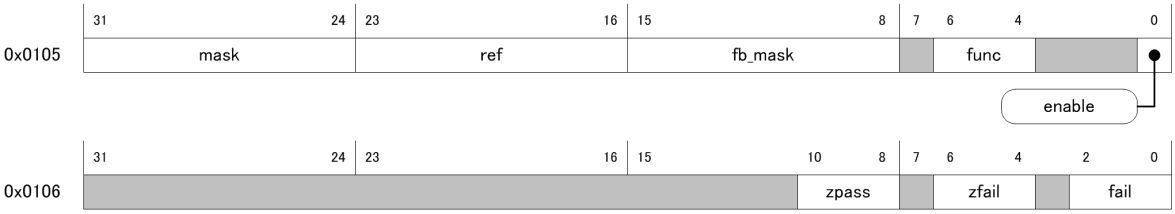
表 8-48. 名前と設定の対応（アーリーデプステスト設定レジスタ）

設定する関数	ビット数	説明
EnableEarlyDepthTest0 EnableEarlyDepthTest1	1 1	glEnable() / glDisable() に GL_EARLY_DEPTH_TEST_DMP を渡して変更した、アーリーデプステストの有効・無効を設定します。 0x0：アーリーデプステストが無効 0x1：アーリーデプステストが有効
EarlyDepthFunc	2	glEarlyDepthFuncDMP() の引数 *func* で指定した、比較方法の設定です。 0x0：GL_GEQUAL 0x1：GL_GREATOR 0x2：GL_LEQUAL 0x3：GL_LESS
ClearEarlyDepthBit	1	glClear() に GL_EARLY_DEPTH_BUFFER_BIT_DMP を含む引数を渡して、アーリーデプスバッファをクリアするときに 0x1 を設定します。
ClearEarlyDepthValue	24	glClearEarlyDepthDMP() の引数 *depth* で指定した、アーリーデプスバッファのクリア値です。引数に渡す値そのままを設定します。

8.8.14. ステンシルテスト設定レジスタ（0x0105, 0x0106）

ステンシルテストの設定に対応するレジスタは以下のとおりです。これらのレジスタを設定したときは、「8.8.9.6. フレームバッファアクセス制御設定レジスタ（0x0112 ～ 0x0115）」も併せて変更しなければならない場合があります。

図 8-62. ステンシルテスト設定レジスタ（0x0105, 0x0106）のビットレイアウト



ビットレイアウト中の名前は以下のようにステンシルテストの設定に対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

表 8-49. 名前と設定の対応（ステンシルテスト設定レジスタ）

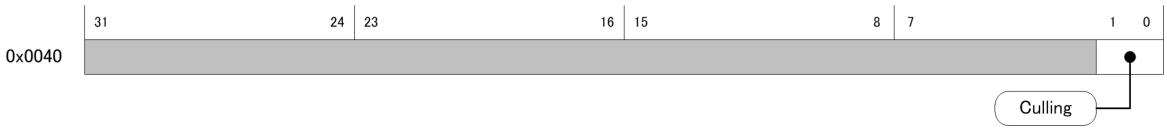
名前	ビット数	説明
enable	1	glEnable() / glDisable() に GL_STENCIL_TEST を渡して変更した、ステンシルテストの有効・無効を設定します。 0x0：ステンシルテストが無効 0x1：ステンシルテストが有効
fb_mask	8	glStencilMask() の引数 *mask* で指定した、ステンシルバッファのマスキング値です。引数に渡す値の下位 8 ビットを設定します。
func	3	glStencilFunc() の引数 *func* で変更した、比較方法の設定です。 0x0：GL_NEVER 0x1：GL_ALWAYS 0x2：GL_EQUAL 0x3：GL_NOTEQUAL 0x4：GL_LESS 0x5：GL_LEQUAL 0x6：GL_GREATER 0x7：GL_GEQUAL

ref	8	glStencilFunc() の引数 *ref* で指定した、ステンシルテストの参照値です。引数に渡す値そのままを設定します。
mask	8	glStencilFunc() の引数 *mask* で指定した、ステンシルテストのマスキング値です。引数に渡す値そのままを設定します。
fail	3	glStencilOp() の引数 *fail* で変更した、ステンシルテストでフラグメントが棄却された際のステンシルバッファの変更内容の設定です。 0x0 : GL_KEEP 0x1 : GL_ZERO 0x2 : GL_REPLACE 0x3 : GL_INCR 0x4 : GL_DECR 0x5 : GL_INVERT 0x6 : GL_INCR_WRAP 0x7 : GL_DECR_WRAP
zfail	3	glStencilOp() の引数 *zfail* で変更した、デプステストでフラグメントが棄却された際のステンシルバッファの変更内容の設定です。設定値は fail と同じです。
zpass	3	glStencilOp() の引数 *zpass* で変更した、デプステストでフラグメントが通過した際のステンシルバッファの変更内容の設定です。設定値は fail と同じです。

8.8.15. カリング設定レジスタ(0x0040)

カリングの設定に対応するレジスタは以下のとおりです。

図 8-63. カリング設定レジスタ(0x0040)のビットレイアウト



ビットレイアウト中の名前は以下のようにステンシルテストの設定に対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

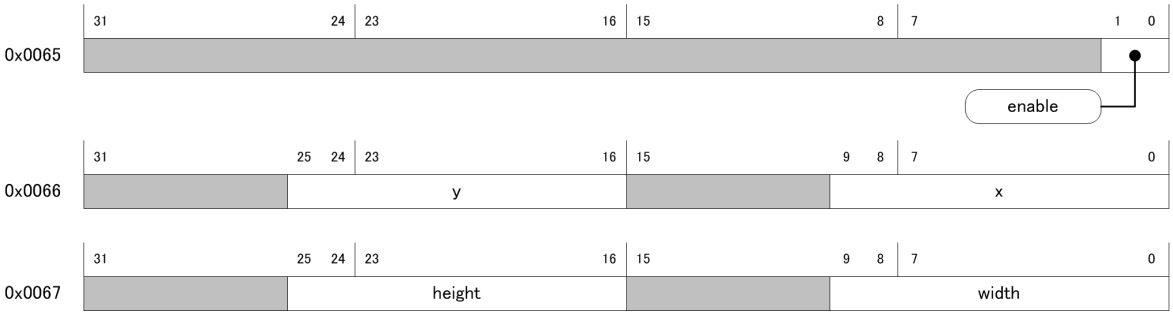
表 8-50. 名前と設定の対応(カリング設定レジスタ)

名前	ビット数	説明
Culling	2	glDisable(GL_CULL_FACE) によってカリングが無効になっている場合は 0x0 を設定します。 glEnable(GL_CULL_FACE) によってカリングが有効になっている場合は、glCullFace() と glFrontFace() に渡した引数の組み合わせで設定する値を以下のように変化させます。 glCullFace() が GL_FRONT かつ glFrontFace() が GL_CW、または glCullFace() が GL_BACK かつ glFrontFace() が GL_CCW ならば 0x2 を設定します。 上記以外の場合は 0x1 を設定します。

8.8.16. シザーテスト設定レジスタ(0x0065 ～ 0x0067)

シザーテストの設定に対応するレジスタは以下のとおりです。

図 8-64. シザーテスト設定レジスタ(0x0065 ～ 0x0067)のビットレイアウト



ビットレイアウト中の名前は以下のようにシザーテストの設定に対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。

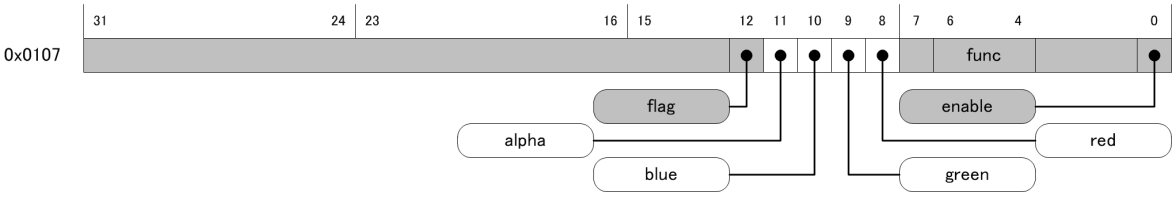
表 8-51. 前と設定の対応(シザーテスト設定レジスタ)

名前	ビット数	説明
enable	2	glEnable() / glDisable() に GL_SCISSOR_TEST を渡して変更した、シザーテストの有効・無効の設定です。 0x0 : シザーテストが無効 0x3 : シザーテストが有効
x	10	glScissor() の引数 *x* で指定した、シザーボックスの始点座標(x 方向)です。 シザーテストが無効である場合は 0 を設定します。 シザーテストが有効である場合は引数 *x* の値そのままを設定しますが、*x* がカレントのカラーバッファの幅以上ならば(カレントのカラーバッファの幅 - 1)が、*x* が負の値ならば 0 を設定します。
y	10	glScissor() の引数 *y* で指定した、シザーボックスの始点座標(y 方向)です。 シザーテストが無効である場合は 0 を設定します。 シザーテストが有効である場合は引数 *y* の値そのままを設定しますが、*y* がカレントのカラーバッファの高さ以上ならば(カレントのカラーバッファの高さ - 1)が、*y* が負の値ならば 0 を設定します。
width	10	glScissor() の引数 *width* で指定した、シザーボックスの幅です。 シザーテストが無効である場合は(カレントのカラーバッファの幅 - 1)を設定します。 シザーテストが有効である場合は(*x* + *width* - 1)の計算結果を設定しますが、計算結果がカレントのカラーバッファの幅以上ならば(カレントのカラーバッファの幅 - 1)が、計算結果が負の値ならば 0 を設定します。
height	10	glScissor() の引数 *height* で指定した、シザーボックスの高さです。 シザーテストが無効である場合は(カレントのカラーバッファの高さ - 1)を設定します。 シザーテストが有効である場合は(*y* + *height* - 1)の計算結果を設定しますが、計算結果がカレントのカラーバッファの高さ以上ならば(カレントのカラーバッファの高さ - 1)が、計算結果が負の値ならば 0 を設定します。

8.8.17. カラーマスク設定レジスタ(0x0107)

glColorMask() で行われるカラーマスク設定に対応するレジスタは以下のとおりです。これらのレジスタを設定したときは、「8.8.9.6. フレームバッファアクセス制御設定レジスタ(0x0112 ～ 0x0115)」も併せて変更しなければならない場合があります。

図 8-65. カラーマスク設定レジスタ(0x0107)のビットレイアウト



ビットレイアウト中の名前は以下のようにカラーマスクの設定に対応しています。下表では、それぞれのビット数と設定値の対応についても説明しています。表中にない名前は、「8.8.11. デプステスト設定レジスタ(0x0107, 0x0126)」で使用するビットです。

表 8-52. 名前と設定の対応(カラーマスク設定レジスタ)

名前	ビット数	説明
red	1	glColorMask() の引数 *red* で指定した、カラーマスクの赤成分の設定です。 0x0 : GL_FALSE 0x1 : GL_TRUE
green	1	glColorMask() の引数 *green* で指定した、カラーマスクの緑成分の設定です。 0x0 : GL_FALSE 0x1 : GL_TRUE
blue	1	glColorMask() の引数 *blue* で指定した、カラーマスクの青成分の設定です。 0x0 : GL_FALSE 0x1 : GL_TRUE
alpha	1	glColorMask() の引数 *alpha* で指定した、カラーマスクのアルファ成分の設定です。 0x0 : GL_FALSE 0x1 : GL_TRUE

8.8.18. ブロックフォーマット設定レジスタ(0x011B)

ブロックフォーマットの設定に対応するレジスタは以下のとおりです。

表 8-53. ブロックフォーマット設定とレジスタの対応

設定する関数	レジスタ	説明
glRenderBlockModeDMP()	0x011B のビット[0 : 0]	0x0 : GL_RENDER_BLOCK8_MODE_DMP 0x1 : GL_RENDER_BLOCK32_MODE_DMP

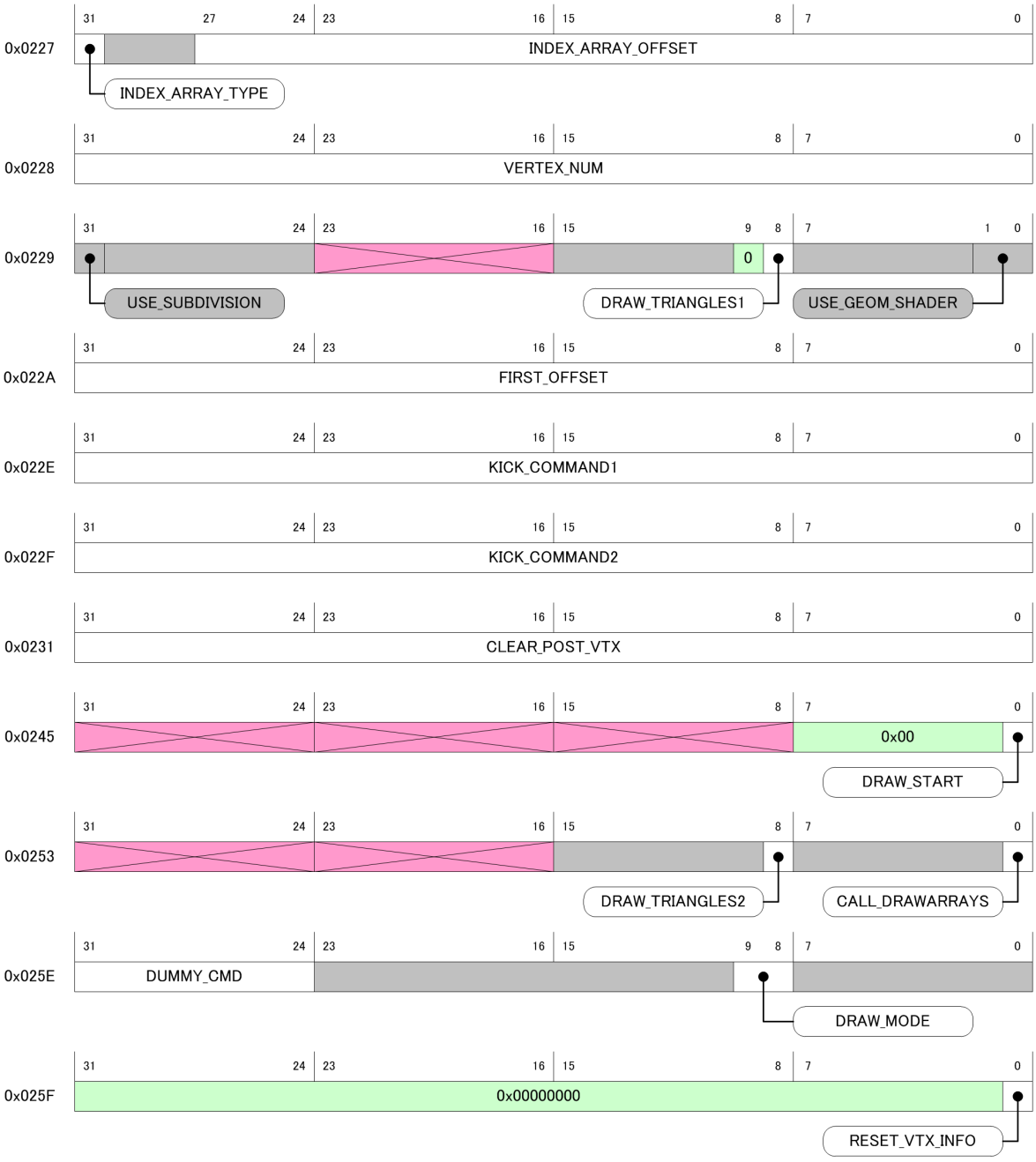
8.8.19. 描画 API に関するレジスタ(0x0227 ～ 0x022A ほか)

glDrawElements() と glDrawArrays() の呼び出しで各ステートのバリデーションが行われるため、各ステートに関連するレジスタへのレジスタ設定コマンドが生成されます。これらの関数では、バリデーションにより生成されるコマンド以外にも、描画処理そのものに必要なレジスタ設定があります。

8.8.19.1. 頂点バッファを使用する場合の設定

頂点バッファを使用して描画を行う場合、以下のレジスタに対して設定を行います。コマンドの設定順序についての断りがない場合、各コマンドは描画開始コマンドより前に設定されていなければなりません。

図 8-66. 描画に頂点バッファを使用した場合の設定レジスタのビットレイアウト



ビットレイアウト中の名前は以下のようにそれぞれの設定に対応しています。

表 8-54. 名前と設定の対応(描画に頂点バッファを使用した場合の設定レジスタ)

名前	ビット数	説明
DRAW_MODE DRAW_TRIANGLES1 DRAW_TRIANGLES2	2 1 1	glDrawElements() または glDrawArrays() の引数 *mode* で指定された描画モードの設定です。設定が変更にならない限り、再設定は必要ありません。 0x0 : GL_TRIANGLES(glDrawArrays() のみ) 0x1 : GL_TRIANGLE_STRIP 0x2 : GL_TRIANGLE_FAN 0x3 : GL_GEOMETRY_PRIMITIVE_DMP(ただし、glDrawElements() では、DRAW_TRIANGLES1 と DRAW_TRIANGLES2 に 1 がセットされている場合は GL_TRIANGLES が指定されます)
CALL_DRAWARRAYS	1	glDrawElements() であれば常に 0 を、glDrawArrays() であれば 1 を設定します。ただし、ライブラリの初期化時に 0 が設定されますので、glDrawArrays() の呼び出し時には 1 を設定し、関数の終了時に 0 を設定します。 1 に設定した場合、レジスタ 0x0200～0x0254 と 0x0280～0x02DF 以外のレジスタへの設定が正しく実行されない可能性があります。
INDEX_ARRAY_OFFSET	28	頂点インデックスアレイのアドレスオフセットの設定です。レジスタ 0x0200 のビット [28 : 1] に設定されている、頂点アレイ共通のベースアドレスからのオフセット値です。ベースアドレス× 16 に、このオフセット値を加算した結果が、glBufferData() で確保された頂点バッファの先頭アドレスと glDrawElements() の引数 *indices* を加算した値と等しくなるようにします。 glDrawArrays() の場合、以下の条件を満たす場合には 0x20 を、それ以外の場合には 0 を設定します。 ● VERTEX_NUM が 0x10 より大きい場合: ((VERTEX_NUM - 0x10)×2 + (ARRAY_BASE_ADDR << 4)) & 0xFFF >= 0xFE0 ● VERTEX_NUM が 0x10 以下の場合: (ARRAY_BASE_ADDR << 4) & 0xFFF >= 0xFE0 VERTEX_NUM はレジスタ 0x0228 の [31 : 0] の設定、ARRAY_BASE_ADDR はレジスタ 0x0200 の [28 : 1] の設定です。 設定が変更にならない限り、再設定は必要ありません。
INDEX_ARRAY_TYPE	1	頂点インデックスの型の設定です、glDrawElements() の引数 *type* が GL_UNSIGNED_SHORT の場合は 1 を、GL_UNSIGNED_BYTE の場合は 0 を設定します。 glDrawArrays() の場合は常に 1 を設定します。設定が変更にならない限り、再設定は必要ありません。
VERTEX_NUM	32	描画する頂点数の設定です。設定が変更にならない限り、再設定は必要ありません。0 を設定した場合の動作は不定ですので、0 を設定しないでください。
FIRST_OFFSET	32	glDrawArrays() を描画する場合の引数 *first* の値です。設定が変更にならない限り、再設定は必要ありません。

RESET_VTX_INFO	1	このビットに 1 を書き込むと、三角形を構成するインデックス 0、1、2 の頂点の情報がリセットされます。 描画モードの引数を GL_GEOMETRY_PRIMITIVE_DMP にしている場合や、GL_TRIANGLES を引数に glDrawElements() で描画する場合は、設定する必要ありません。 GL_TRIANGLES を引数に glDrawArrays() で描画する際は、頂点数が 3 の倍数であれば、同じモードで連続して呼び出す(その間に glDrawElements() を呼び出さない)場合に限り、2 回目以降の呼び出し時にリセットする必要はありません。ほかのモードで描画したあとや glDrawElements() のあと、ライブラリの初期化後に初めて呼び出すときにはリセットしなければなりません。 GL_TRIANGLE_STRIP または GL_TRIANGLE_FAN を引数にした場合は、どちらの関数でも、描画開始コマンドごとにリセットする必要があります。
KICK_COMMAND1 KICK_COMMAND2	特殊	描画を開始するときに、任意のビットに 1 を書き込みます。 KICK_COMMAND1(0x022E)が glDrawArrays()、KICK_COMMAND2(0x022F)が glDrawElements() です。
CLEAR_POST_VTX	特殊	描画開始コマンドの直後に、任意のビットに 1 を書き込みます。描画開始ごとに設定する必要があります。
clearFrameBufferCacheData レジスタ 0x0111 [0 : 0]	1	描画開始コマンドの直後に 1 を書き込みます。 このレジスタの詳細は「8.8.21. フレームバッファキャッシュクリアの設定レジスタ(0x0110, 0x0111)」を参照してください。
samplerType0 samplerType1 samplerType2 レジスタ 0x0080 [2 : 0]	1 1 1	描画開始コマンドの直前に、有効にしなければならないテクスチャユニットに関してだけ 1 を設定し、描画開始コマンドの直後に 0 を設定します。有効にしなければならないテクスチャユニットに対して、常に 1 を設定していても動作の上で問題はありません。しかし、0 が設定されていると消費電力の低減に効果があるため、描画時以外は 0 に設定されています。 このレジスタの詳細は「8.8.6.2. テクスチャサンプラータイプ設定レジスタ(0x0080, 0x0083)」を参照してください。
DRAW_START	1	ライブラリの初期化時に 1 が設定されます。1 が設定されている場合、描画が正しく行われません。0 が設定されている場合、レジスタ 0x02B0 から 0x02DF の範囲の設定コマンドが正しく反映されません。 glDrawElements() と glDrawArrays() では、描画開始コマンドの直前で 0 に設定し、直後に 1 に戻すようにコマンドが生成されます。 描画開始コマンド後も、レジスタ 0x02B0 から 0x02DF の範囲の設定コマンドを使用しない場合は 0 が設定されたままでもかまいません。
DUMMY_CMD	8	描画開始コマンドごとに、描画開始コマンドの直後に、0x0 を書き込むコマンドが 2 つ必要です。 このタイミングでこのレジスタに書き込むコマンドはダミーコマンドですので、設定値自体は意味を持ちません。
レジスタ 0x02BA の ビット [31 : 16]	16	描画開始コマンドのあとに 0x7FFF を書き込むコマンドを実行すると、消費電力の低減に効果があります。 設定を行わなくても、動作上は問題ありませんが、設定を行う場合はバイトイネーブルに 0xC を設定し、ビット [15 : 0] に影響がないようにしてください。また、このレジスタへの設定の前に DRAW_START に 1 が設定されていなければなりません。
レジスタ 0x028A の ビット [31 : 16]	16	描画開始コマンドのあとに 0x7FFF を書き込むコマンドを実行すると、消費電力の低減に効果があります。 設定を行わなくても、動作上は問題ありませんが、設定を行う場合はバイトイネーブルに 0xC を設定し、ビット [15 : 0] に影響がないようにしてください。 ジオメトリシェーダを使用しない(レジスタ 0x0244 のビット [0 : 0] が 0、かつレジスタ 0x0229 のビット [1 : 0] が 0)場合は、レジスタ 0x02BA のビット [31 : 16] への設定がこのレジスタへの設定を兼ねるため、このレジスタへの設定コマンドは不要となります。

コマンドの依存関係に関する注意点

レジスタ 0x02BA のビット [31 : 16] への設定は、DRAW_START への設定のあとに行われなければなりません。

CALL_DRAWARRAYS に 1 が設定されている間は、レジスタ 0x0200～0x0254 と 0x0280～0x02DF 以外のレジスタへの設定が正しく実行されない場合があります。これらのレジスタへの設定は、CALL_DRAWARRAYS に 0 が設定されているときに行ってください。ただし、レジスタ 0x025E のビット [31 : 24] に対するダミーコマンドはこの限りではありません。

そのほかのコマンドで、描画開始コマンドの直後に必ず設定が必要なものがいくつかありますが、それらのコマンドの順番に依存関係はありません。

INDEX_ARRAY_OFFSET の設定に関する補足

表中の INDEX_ARRAY_OFFSET（レジスタ 0x0227 のビット [27 : 0]）の説明では、条件によって 0 または 0x20 を設定する必要があると記載されていますが、厳密には、**以下の条件を満たさないように設定**すれば、0 や 0x20 以外の値でも正しい設定となります。

- VERTEX_NUM が 0x10 より大きい場合
  ((VERTEX_NUM - 0x10) × 2 + (ARRAY_BASE_ADDR << 4) + INDEX_ARRAY_OFFSET) & 0xFFF >= 0xFE0
- VERTEX_NUM が 0x10 以下の場合
  (ARRAY_BASE_ADDR << 4 + INDEX_ARRAY_OFFSET) & 0xFFF >= 0xFE0

VERTEX_NUM はレジスタ 0x0228 の [31 : 0]、ARRAY_BASE_ADDR はレジスタ 0x0200 の [28 : 1] の設定値です。

ロードアレイに関する制限

KICK_COMMAND2（レジスタ 0x022F）への書き込みで描画を開始する場合は、ロードアレイを 12 個使用できないという制限があります。詳細については「ロードアレイの制限」を参照してください。

8.8.19.2. 頂点バッファを使用しない場合の設定

頂点バッファを使用せずに描画を行う場合のレジスタに対する設定を、使用する場合との違いを視点に説明します。ビットレイアウトは図 8-66 と同じです。以下にビットレイアウト中の名前と設定との対応を示します。頂点属性データコマンドが描画開始コマンドと同じように扱われます。コマンドの設定順序についての断りがない場合、各コマンドは描画開始コマンドより前に設定されていなければなりません。

表 8-55. 名前と設定の対応（描画に頂点バッファを使用しない場合の設定レジスタ）

名前	ビット数	説明
DRAW_MODE DRAW_TRIANGLES1 DRAW_TRIANGLES2	2 1 1	glDrawElements() または glDrawArrays() の引数 *mode* で指定された描画モードの設定です。設定が変更にならない限り、再設定は必要ありません。 0x0：GL_TRIANGLES 0x1：GL_TRIANGLE_STRIP 0x2：GL_TRIANGLE_FAN 0x3：GL_GEOMETRY_PRIMITIVE_DMP DRAW_TRIANGLES1 と DRAW_TRIANGLES2 には 0 を設定します。
CALL_DRAWARRAYS	1	glDrawElements()、glDrawArrays() のどちらを呼び出しても同じです。ライブラリの初期化時に 0 が設定されますので、関数の呼び出し時には 1 を設定し、関数の終了時に 0 を設定します。 1 に設定されている場合、レジスタ 0x0200～0x0254 と 0x0280～0x02DF 以外のレジスタへの設定が正しく実行されない可能性があります。
INDEX_ARRAY_OFFSET	28	使用しません。

INDEX_ARRAY_TYPE	1	使用しません。
VERTEX_NUM	32	使用しません。処理される頂点数は、入力される頂点属性データの個数で決まります。
FIRST_OFFSET	32	使用しません。
RESET_VTX_INFO	1	GL_TRIANGLES を引数に glDrawElements() または glDrawArrays() で描画する際は、頂点数が 3 の倍数であれば、頂点バッファを使用しない glDrawElements() または glDrawArrays() を同じモードで連続して呼び出す場合に限り、2 回目以降の呼び出し時にリセットする必要はありません。ほかのモードで描画したあとや、頂点バッファを使用した glDrawElements() のあと、ライブラリの初期化後に初めて呼び出すときにはリセットしなければなりません。 そのほかの描画モード(GL_GEOMETRY_PRIMITIVE_DMP、GL_TRIANGLE_STRIP、GL_TRIANGLE_FAN)では、頂点バッファを使用する場合と違いはありません。
KICK_COMMAND1 KICK_COMMAND2	特殊	設定しないでください。
CLEAR_POST_VTX	特殊	頂点バッファを使用する場合と違いはありません。
clearFrameBufferCacheData レジスタ 0x0111 [0 : 0]	1	頂点バッファを使用する場合と違いはありません。
samplerType0 samplerType1 samplerType2 レジスタ 0x0080 [2 : 0]	1 1 1	頂点バッファを使用する場合と違いはありません。
DRAW_START	1	頂点バッファを使用する場合と違いはありません。
レジスタ 0x02BA の ビット [31 : 16]	16	頂点バッファを使用する場合と違いはありません。
レジスタ 0x028A の ビット [31 : 16]	16	頂点バッファを使用する場合と違いはありません。

頂点属性データの入力には、「8.8.1.8. 固定頂点属性値設定レジスタ(0x0232 ～ 0x0235)」を使用します。

まず、レジスタ 0x0232 のビット [3 : 0] に 0xF を書き込みます。次に、頂点属性データを 24 ビット浮動小数点数に変換した 3 つの 32 ビットデータを、レジスタ 0x0233、0x0234、0x0235 の順に書き込みます。24 ビット浮動小数点数に変換した 3 つの 32 ビットデータは、「24 ビット浮動小数点数入力モード」にあるデータと同じ方法で作成します。

頂点バッファを使用しない場合、「8.8.1.9. 頂点属性アレイ設定レジスタ(0x0200 ～ 0x0227)」で説明されているレジスタへの設定は不要です。コマンドの設定順序についての依存関係は頂点バッファを使用する場合と同じです。

8.8.20. ジオメトリシェーダ設定レジスタ(0x0280 ～ 0x02AF ほか)

GPU には頂点処理を行う頂点シェーダプロセッサが複数搭載されています。ジオメトリシェーダを使用する場合、そのうちの1つがジオメトリシェーダプロセッサとして動作します。このプロセッサを共用プロセッサと呼び、ジオメトリシェーダとして使用しないときは頂点シェーダプロセッサとして動作しているため、浮動小数点数レジスタやブールレジスタなどのリソースには頂点シェーダとしての値が設定されています。ジオメトリシェーダを使用しない設定から使用する設定に変更する場合、頂点シェーダとして設定されている値をジオメトリシェーダとしての設定に変更しなければなりません。同様にジオメトリシェーダを使用する設定から、使用しない設定に変更する場合にもジオメトリシェーダとして設定されている値を頂点シェーダとしての設定に変更しなければなりません。

レジスタ 0x02B0 から 0x02DF は頂点シェーダプロセッサの設定レジスタです。この範囲のレジスタに対する設定は、複数あるすべての頂点シェーダプロセッサに対して行われます。共用プロセッサに対しても同様に同じ設定が行われますが、レジ

スタ 0x0244 のビット [0 : 0] に 1 が設定されている場合、共用プロセッサには頂点シェーダプロセッサへのレジスタ設定が反映されなくなります。逆に 0 が設定されている、かつレジスタ 0x0229 のビット [1 : 0] が 0x0 の場合は設定が共用プロセッサに反映されます。

レジスタ 0x02B0 から0x02DF と同様の設定を共用プロセッサのみに行う場合は、レジスタのアドレスから 0x30 を引いたアドレス、つまりレジスタ 0x0280 から 0x02AF を使用します。

ジオメトリシェーダを使用する場合はレジスタ 0x0280 から 0x02AF にはジオメトリシェーダ固有の設定をし、ジオメトリシェーダを使用しない場合はレジスタ 0x0280 から 0x02AF にはレジスタ 0x02B0 から 0x02DF と同じ設定をする必要があります。これは、レジスタ 0x0244 のビット [0 : 0] に 0 を、レジスタ 0x0229 のビット [1 : 0] に 0x0 を設定し、頂点シェーダプロセッサへの設定が共用プロセッサにも反映されるようにしてから、レジスタ 0x02B0 から 0x02DF に対して再度設定することでも可能です。

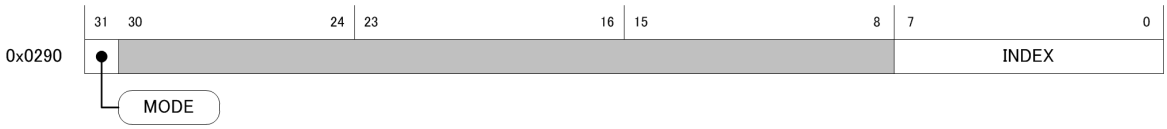
ジオメトリシェーダを使用するには、これらの共用プロセッサに関するレジスタ設定と、入出力などに関するその他のレジスタ設定が必要となります。

8.8.20.1. 浮動小数点定数レジスタ(0x0290, 0x0291 ～ 0x0298)

設定に使用するレジスタのアドレスが異なるだけで、レジスタに設定する値は「8.8.1.1. 浮動小数点定数レジスタ(0x02C0, 0x02C1 ～ 0x02C8)」と同じです。

レジスタ 0x0290 がレジスタ 0x02C0 に、レジスタ 0x0291 ～ 0x0298 がレジスタ 0x02C1 ～ 0x02C8 に対応しています。

図 8-67. 浮動小数点定数レジスタのインデックス指定(0x0290)のビットレイアウト

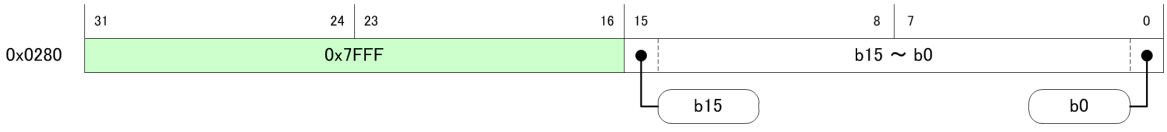


8.8.20.2. ブールレジスタ(0x0280)

設定に使用するレジスタのアドレスが異なるだけで、レジスタに設定する値は「8.8.1.2. ブールレジスタ(0x02B0)」と同じです。

レジスタ 0x0280 がレジスタ 0x02B0 に対応しています。

図 8-68. ブールレジスタ(0x0280)のビットレイアウト

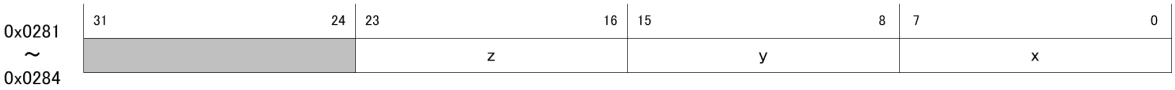


8.8.20.3. 整数レジスタ(0x0281 ～ 0x0284)

設定に使用するレジスタのアドレスが異なるだけで、レジスタに設定する値は「8.8.1.3. 整数レジスタ(0x02B1 ～ 0x02B4)」と同じです。

レジスタ 0x0281 ～ 0x0284 がレジスタ 0x02B1 ～ 0x02B4 に対応しています。

図 8-69. 整数レジスタ(0x0281 ～ 0x0284)のビットレイアウト



8.8.20.4. プログラムコード設定レジスタ(0x028F, 0x029B ～ 0x02A3, 0x02A5 ～ 0x02AD)

設定に使用するレジスタのアドレスが異なるだけで、レジスタに設定する値は「8.8.1.4. プログラムコード設定レジスタ(0x02BF, 0x02CB ～ 0x02D3, 0x02D5 ～ 0x02DD)」と同じです。

レジスタ 0x028F がレジスタ 0x02BF に、レジスタ 0x029B ～ 0x02A3 がレジスタ 0x02CB ～ 0x02D3 に、レジスタ 0x02A5 ～ 0x02AD がレジスタ 0x02D5 ～ 0x02DD に対応しています。

図 8-70. プログラムコードのロードレジスタ(0x029B ～ 0x02A3)のビットレイアウト

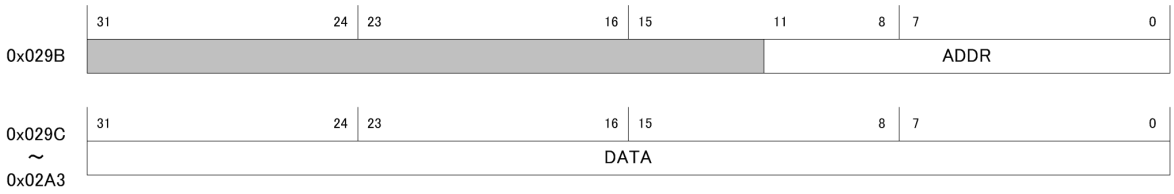
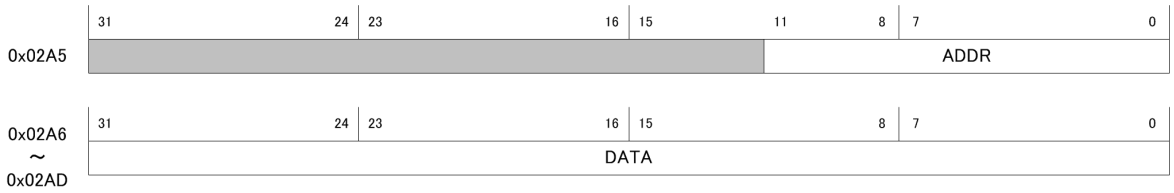


図 8-71. Swizzle パターンのロードレジスタ(0x02A5 ～ 0x02AD)のビットレイアウト

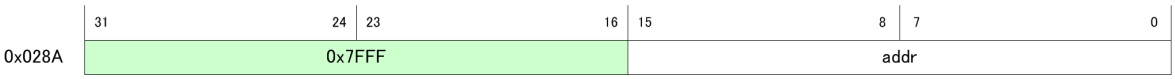


8.8.20.5. 開始アドレス設定レジスタ(0x028A)

設定に使用するレジスタのアドレスが異なるだけで、レジスタに設定する値は「8.8.1.5. 開始アドレス設定レジスタ(0x02BA)」と同じです。

レジスタ 0x028A がレジスタ 0x02BA に対応しています。

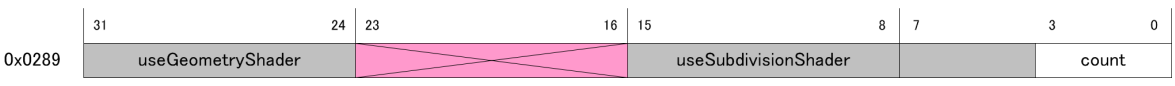
図 8-72. 開始アドレス設定レジスタ(0x028A)のビットレイアウト



8.8.20.6. 頂点属性入力数設定レジスタ(0x0289)

ジオメトリシェーダに入力する頂点属性数を設定するレジスタを以下に示します。

図 8-73. 頂点属性入力数設定レジスタ(0x0289)のビットレイアウト



count には(入力する頂点属性数 - 1)を設定します。ジオメトリシェーダに入力する頂点属性の数は、頂点シェーダから出力する頂点属性の数(generic 属性も含む)と同じです。

ここで設定する頂点属性数とは、頂点シェーダアセンブラで #pragma output_map を定義した数ではなく、頂点シェーダアセンブラの #pragma output_map で定義される出力レジスタの個数です。つまり、1 つの出力レジスタがコンポーネ

ント別に複数の #pragma output_map で定義されている場合は、1 とカウントします。

8.8.20.7. 入力レジスタのマッピング設定レジスタ(0x028B, 0x028C)

設定に使用するレジスタのアドレスが異なるだけで、レジスタに設定する値は「8.8.1.7. 入力レジスタのマッピング設定レジスタ(0x02BB, 0x02BC)」と同じです。ただし、ラインシェーダなどのジオメトリシェーダを使用している場合には、レジスタ 0x028B に 0x76543210 を、レジスタ 0x028C に 0xFEDCBA98 を設定します。

レジスタ 0x028B ～ 0x028C がレジスタ 0x02BB ～ 0x02BC に対応しています。

図 8-74. 入力レジスタのマッピング設定レジスタ(0x028B, 0x028C)のビットレイアウト

8.8.20.8. 出力レジスタ使用数設定レジスタ(0x004F, 0x025E)

ジオメトリシェーダから出力される出力レジスタの個数は「8.8.1.10. 出力レジスタ使用数設定レジスタ(0x004F, 0x024A, 0x0251, 0x025E)」と共通するレジスタに設定されます。

count1(レジスタ 0x004F)には使用する出力レジスタの個数そのままを、count2(レジスタ 0x025E のみ)には(使用する出力レジスタの個数 - 1)をそれぞれ設定します。それぞれのレジスタで設定される値とビット幅が異なる点に注意してください。

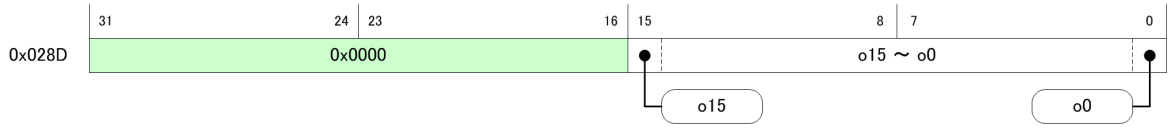
ここで設定する出力レジスタの個数とは、ジオメトリシェーダに #pragma output_map で定義されている出力レジスタの個数を指します。つまり、1 つの出力レジスタがコンポーネント別に複数の #pragma output_map で定義されている場合は、1 とカウントします。

8.8.20.9. 出力レジスタのマスク設定レジスタ(0x028D)

設定に使用するレジスタのアドレスが異なるだけで、レジスタに設定する値は「8.8.1.11. 出力レジスタのマスク設定レジスタ(0x02BD)」と同じです。

レジスタ 0x028D がレジスタ 0x02BD に対応しています。

図 8-75. 出力レジスタのマスク設定レジスタ(0x028D)のビットレイアウト



8.8.20.10. 出力レジスタの属性設定レジスタ(0x0050 ～ 0x0056, 0x0064)

頂点シェーダから出力される頂点の属性についてではなく、ジオメトリシェーダから出力される頂点の属性について設定することを除いて、レジスタに設定する値は「8.8.1.12. 出力レジスタの属性設定レジスタ(0x0050 ～ 0x0056, 0x0064)」と同じです。

ジオメトリシェーダの出力属性は、ジオメトリシェーダで定義されている #pragma output_map の設定で決定されます。これは、シェーダアセンブラのリンカが出力するマップファイルに生成される情報です。いくつかのジオメトリシェーダは、generic 属性を出力属性として定義しています。generic 属性として定義された出力属性には、リンクした頂点シェーダでのみ定義されている #pragma output_map の設定が適用されます。その際、頂点シェーダで定義されている generic 属性は除かれます。

補足: マップファイルの詳細については「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。

8.8.20.11. 出力属性のクロック制御レジスタ(0x006F)

頂点シェーダについてではなく、ジオメトリシェーダについて設定することを除いて、レジスタに設定する値は「8.8.1.13. 出力属性のクロック制御レジスタ(0x006F)」と同じです。

8.8.20.12. そのほかの設定レジスタ(0x0229, 0x0252, 0x0254, 0x0289)

ジオメトリシェーダを使用する際に設定する、そのほかのレジスタは以下のとおりです。

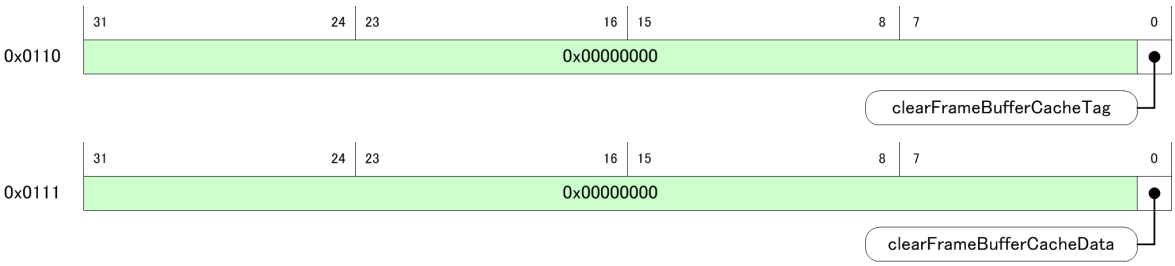
表 8-56. そのほかのレジスタ(ジオメトリシェーダ)

レジスタ	説明
0x0229 のビット [1 : 0]	ジオメトリシェーダを使用する場合は 0x2 を、使用しない場合は 0x0 を設定します。このレジスタ設定コマンドの前後にダミーコマンドを必要とします。ダミーコマンドにはバイトイネーブルを 0 とした無効コマンドを使用します。このレジスタ設定コマンドの直前には、レジスタ 0x0251 へのダミーコマンド 10 個とレジスタ 0x0200 へのダミーコマンド 30 個が、このレジスタ設定コマンドの直後には、レジスタ 0x0200 へのダミーコマンド 30 個が必要です。ただし、このビット以外を設定するコマンドにはダミーコマンドは必要ありません。
0x0229 のビット [31 : 31]	サブディビジョンシェーダ(ループ、Catmull-Clark)を使用する場合は 0x1 を、そのほかのジオメトリシェーダを使用する場合や、ジオメトリシェーダを使用しない場合は 0x0 を設定します。
0x0252 のビット [31 : 0]	サブディビジョンシェーダ(ループ、Catmull-Clark)を使用する場合は 0x00000001 を、パーティクルシステムを使用する場合は 0x01004302 を、それ以外のジオメトリシェーダを使用する場合や、ジオメトリシェーダを使用しない場合は 0x00000000 を設定します。
0x0289 のビット [31 : 24]	ジオメトリシェーダを使用する場合は 0x08 を設定します。ジオメトリシェーダを使用しない場合は 0xA0 を設定します。
0x0289 のビット [15 : 8]	サブディビジョンシェーダ(ループ、Catmull-Clark)またはパーティクルシステムを使用する場合は 0x01 を、それ以外のジオメトリシェーダを使用する場合や、ジオメトリシェーダを使用しない場合は 0x00 を設定します。
0x0254 のビット [4 : 0]	Catmull-Clark サブディビジョンシェーダを使用する場合は 0x3 を、ループサブディビジョンシェーダを使用する場合は 0x2 を設定します。それ以外の場合、このビットは使用されません。

8.8.21. フレームバッファキャッシュクリアの設定レジスタ(0x0110, 0x0111)

フレームバッファキャッシュクリアの設定に対応するレジスタは以下のとおりです。

図 8-76. フレームバッファキャッシュクリアの設定レジスタ(0x0110, 0x0111)のビットアウト



clearFrameBufferCacheData に 1 を書き込むと、カラーバッファおよびデプスバッファの両方のキャッシュデータがフラッシュされます。clearFrameBufferCacheTag に 1 を書き込むと、カラーバッファおよびデプスバッファ両方のキャッシュのタグがクリ

アされます。キャッシュタグをクリアするコマンドはキャッシュデータをフラッシュするコマンドの直後に使用する必要があり、必ずキャッシュデータのフラッシュ、キャッシュタグのクリアの順で使用しなければなりません。

これらのレジスタを設定するコマンドは、glFlush() や glFinish()、glClear() を呼び出したときや、カラーバッファおよびデプスバッファのアドレスが変更された際のステートフラグ(NN_GX_STATE_FRAMEBUFFER)のバリデーションを行ったとき、ステートフラグ NN_GX_STATE_FBACCESS のバリデーションを行ったときに生成されます。また、nngxSplitDrawCmdlist() などで 3D コマンドバッファを区切る際には、割り込み発生用コマンドの直前に挿入されます。

キャッシュデータをフラッシュするコマンドは、glDrawArrays() または glDrawElements() による描画開始コマンドの直後に単独で生成されます。

2 つのレジスタへの設定が必要なタイミングは、すべての描画処理が完了したとき(描画結果を参照する前)、カラーバッファおよびデプスバッファをクリアしたとき、設定(サイズ、アドレス、フォーマット)を変更したとき、アクセスパターンを変更したときです。

通常、描画コマンドの直後には、毎回 clearFrameBufferCacheData を設定するコマンドを生成する必要がありますが、次の 2 つの条件を満たす場合には描画コマンドごとに生成する必要はありません。

- 描画コマンドの直後に clearFrameBufferCacheData を設定するコマンドを生成したあと、その次の描画コマンドまでにレジスタ 0x0100～0x0130 を設定するコマンドを 1 つも生成していない。
- 描画コマンドのあとでレジスタ 0x0080～0x00B7 にデータを設定する前に、レジスタ 0x0080 へのダミーコマンド(データとバイトイネーブルに 0 を設定)を 3 つ生成している。

描画コマンドのあとにレジスタ 0x0100～0x0130 にデータを設定する場合は、先に clearFrameBufferCacheData を設定するコマンドを 1 つ生成する必要があります。ただし、このコマンドを 1 つでも生成したあとならば、次の描画コマンドまでにレジスタ 0x0100～0x0130 にデータを設定するコマンドをいくつ生成してもかまいません。

同様に、描画コマンドのあとにレジスタ 0x0080 へのダミーコマンド(データとバイトイネーブルに 0 を設定)を 3 つ生成したあとならば、次の描画コマンドまでにレジスタ 0x0080～0x00B7 を設定するコマンドをいくつ生成してもかまいません。

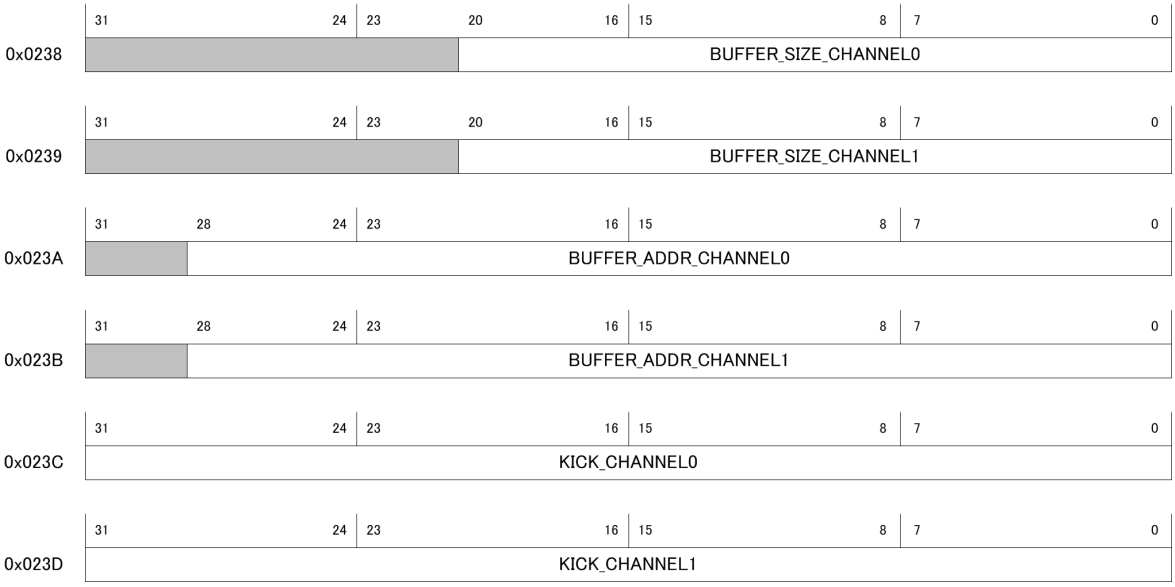
8.8.22. 区切りコマンド設定レジスタ(0x0010)

レジスタ 0x0010 に 0x12345678 を書き込むことで、GPU の割り込みが発生します。3D コマンドバッファを区切る際には、本コマンドを設定してください。

8.8.23. コマンドバッファ実行レジスタ(0x0238 ～ 0x023D)

コマンドバッファの実行は通常、コマンドリクエストにキューイングされた 3D 実行コマンドの情報をもとに行われます。コマンドバッファ実行レジスタを利用すると、コマンドバッファに蓄積された 3D コマンドによって、区切りコマンドで GPU の割り込み処理を発生させることなく別のコマンドバッファを実行することができます。

図 8-77. コマンドバッファ実行レジスタ(0x0238 ～ 0x023D)



コマンドバッファ実行インターフェースには 2 つのチャンネルが用意されています。通常のコマンドバッファの実行にはチャンネル 0 が使用され、2 つのチャンネルを同時に実行させることはできません。

コマンドバッファの実行には、実行するコマンドバッファのアドレスとサイズの設定と、実行(キック)レジスタの 3 種類のレジスタに設定を行う必要があります。

アドレス設定レジスタ(BUFFER_ADDR_CHANNEL0/1)には、実行するコマンドバッファの物理アドレスを 8 で割った値(8 バイト単位のアドレス)を設定します。設定する値は偶数でなければなりません。

サイズ設定レジスタ(BUFFER_SIZE_CHANNEL0/1)には、実行するコマンドバッファの総バイト数を 8 で割った値(8 バイト単位のサイズ)を設定します。設定する値は偶数でなければなりません。

キックレジスタ(KICK_CHANNEL0/1)に値が書き込まれると、各チャンネルのアドレス、サイズの設定情報を使ってコマンドバッファが実行されます。書き込む値は任意です。バイトイネーブルが 0 でなければ、データの値にかかわらず実行されます。コマンドバッファの 3D コマンドでキックレジスタに書き込みを行う場合は、そのコマンドをコマンドバッファの最後に格納してください。なお、コマンドバッファのアドレスとサイズにはアライメントが 16 バイトでなければならない(8 で割った値が偶数でなければならない)制約がありますので、キックコマンド(KICK_CHANNEL0/1 への書き込みコマンド)を格納した直後のアドレスのアライメントは 16 バイトでなければなりません。キックコマンドが実行されるまで、アドレスおよびサイズの設定は実行中のコマンドバッファに影響を与えることはありません。

注意事項

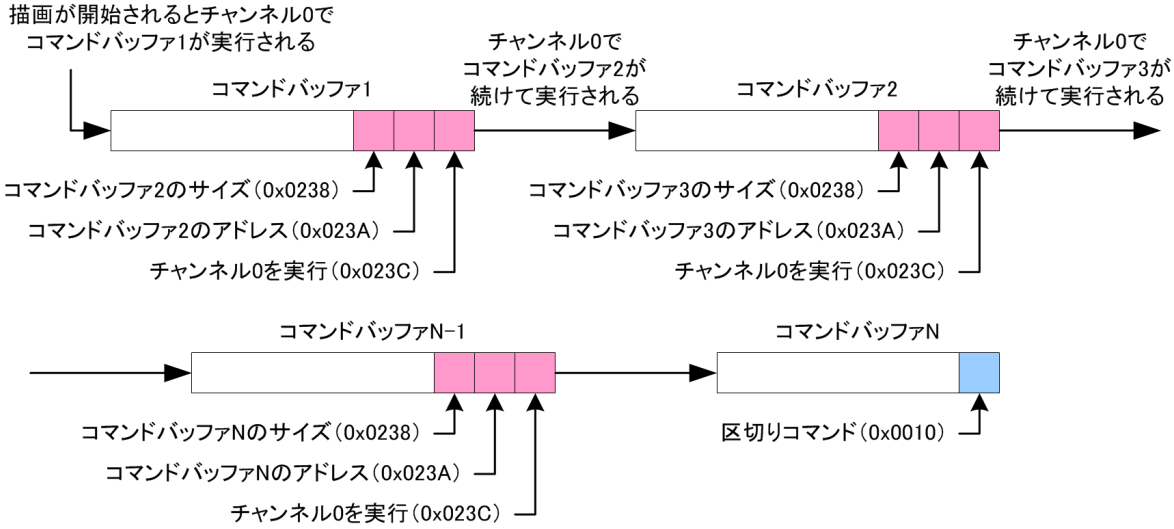
コマンドバッファ実行レジスタへの設定を利用する際は以下のことに注意してください。

- キックコマンドはコマンドバッファの最後に配置しなければなりません。その際は、キックコマンドが最後になるようにコマンドバッファのサイズを指定してください。
- バーストアクセスでキックコマンドを実行する場合はバーストアクセスの途中で書き込むことはできません。キックレジスタへの書き込みが最後であれば実行されます。
- キックコマンドを実行したあとも 2 つのチャンネルに設定されたアドレスとサイズは保持されますが、nngx 関数などで 3D コマンドが実行されたときはチャンネル 0 の設定が上書きされます。
- 実行するコマンドバッファの領域がフラッシュされていなければ、処理が正しく行われない可能性があります。
- チャンネル 0 とチャンネル 1 は同時に実行することができません。
- キックコマンドを実行してから指定されたコマンドバッファのコマンドが実行されるまでの間に、コマンド実行が中断される期間があります。そのため、小さなサイズのコマンドバッファを指定するキックコマンドを頻繁に実行することは、GPU の処理コストとして無視できない大きさになりえます。

8.8.23.1. コマンドバッファの連続実行

下図のように、コマンドバッファ実行レジスタへの書き込みコマンドをコマンドバッファの最後に格納することによって、連続して N 個のコマンドバッファを実行させることができます。

図 8-78. コマンドバッファの連続実行



通常、コマンドバッファの最後には上図のコマンドバッファ N のように区切りコマンドを格納します。そこで区切りコマンドの代わりに、次に実行するコマンドバッファのサイズとアドレスの設定コマンドとキックコマンドの組み合わせを格納すれば、GPU の割り込みを発生させることなく複数のコマンドバッファを実行することができ、CPU 負荷の軽減に繋がります。ただし、最後のコマンドバッファで実行される最後のコマンドは区切りコマンドでなければなりません。

最初のコマンドバッファを実行するには、`nngxAdd3DCommand()` の引数 *`bufferaddr`* と *`buffersize`* にそのコマンドバッファの先頭アドレスとサイズ、*`copycmd`* に `GL_FALSE` を渡して呼び出してください。

8.8.23.2. 同じコマンドバッファの繰り返し実行

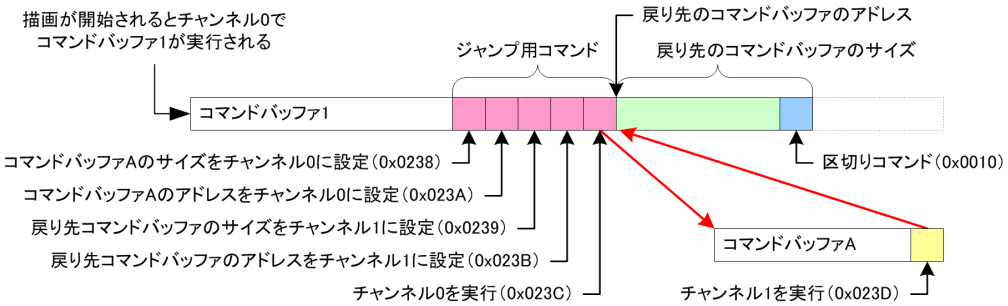
先の例では、同じ内容を繰り返し実行する場合でも別のコマンドバッファとして用意しなければなりませんが、2 つあるチャンネルを利用することで同じコマンドバッファを繰り返し実行させることができます。

例えば、以下の組み合わせであれば、連続するレジスタへの書き込みコマンド 1 つでジャンプ用のコマンドを構成することができます。ただし、ジャンプ先のコマンドバッファの最後がチャンネル 1 のキックコマンドである必要があります。

表 8-57. ジャンプ用コマンドの構成

レジスタ	設定する値
BUFFER_SIZE_CHANNEL0	ジャンプ先のコマンドバッファのサイズ
BUFFER_SIZE_CHANNEL1	戻り先のコマンドバッファのサイズ（次のジャンプ用コマンドまたは区切りコマンドまで）
BUFFER_ADDR_CHANNEL0	ジャンプ先のコマンドバッファのアドレス
BUFFER_ADDR_CHANNEL1	戻り先のコマンドバッファのアドレス（このジャンプ用コマンドの直後）
KICK_CHANNEL0	任意（キックコマンドを実行する）

図 8-79. ジャンプ用コマンドの構成

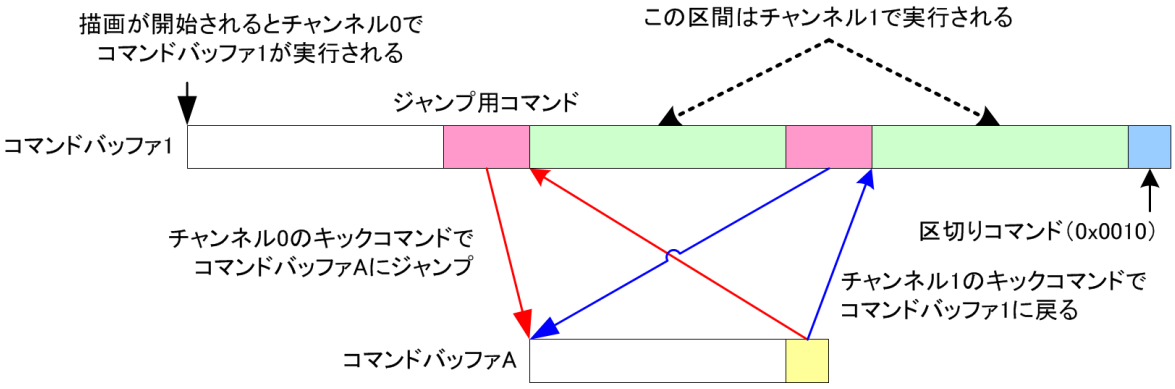


この方法では、ジャンプ元のコマンドバッファで予め戻り先のアドレス情報を設定しておくため、ジャンプ先のコマンドバッファにアドレス情報を含める必要がありません。

上図では、コマンドバッファ 1 でジャンプ先をチャンネル 0 側に、戻り先をチャンネル 1 側に設定したあと、チャンネル 0 のキックコマンドでコマンドバッファ A を実行し、コマンドバッファ A の最後にあるチャンネル 1 のキックコマンドでコマンドバッファ 1 に実行を戻しています。実行中にジャンプ先のコマンドバッファ A の内容を変更することはできませんので、ジャンプ先から戻る際に使用するチャンネルを固定にする必要があることに注意してください。

戻り先の最後に再度ジャンプ用コマンドを格納すれば、同じコマンドバッファを繰り返し実行させることができます。最初のコマンドバッファを実行するには、nngxAdd3DCommand() の *bufferaddr* と *buffersize* にそのコマンドバッファの先頭アドレスと最初のキックコマンドまでのサイズ、*copycmd* に GL_FALSE を渡して呼び出してください。各ジャンプ用コマンドに含まれるチャンネル 1 のコマンドサイズには、ジャンプ元への戻り先アドレスから次のキックコマンドまでのサイズを設定してください。

図 8-80. 同じコマンドバッファの繰り返し実行



この方法を用いれば、以下のような処理を行うことができます。

- 特定の描画処理を繰り返し実行
- ジャンプ先の情報を編集して描画処理を分岐
- 描画処理をモジュール化し、ジャンプ用コマンドのみでシーンを構成

8.8.24. 未記載のビットに対する設定について

これまでに説明したレジスタのうち、情報が記載されていないビットの設定には、バイトイネーブルに 0 を設定することによりアクセスを行わないものや、固定値を設定しているものがあります。それらのビットへのアクセスに関する情報を記載します。記載のないビットに関しては、GPU に影響がないものとして任意の値を設定することができますが、なるべくバイトイネーブルに 0 を設定することを推奨します。また、レジスタそのものの情報が記載されていない場合は、設定を行わないでください。

固定値を設定するように記載されているレジスタは、nngxInitialize() の実行時に各固定値で初期化されますので、アプリケーションで初期化するためのコマンドを発行する必要はありません。固定値を設定するビットと、値の変更が可能なビットが同じレジスタの同じバイト存在する場合は、変更する値と固定値とを同時に書き込む必要があります。

表 8-58. 未記載ビットの設定情報

レジスタ	説明
0x0061 のビット [31 : 8]	バイトイネーブルに 0 を設定し、アクセスしないでください。
0x0062 のビット [31 : 8]	バイトイネーブルに 0 を設定し、アクセスしないでください。
0x006A のビット [31 : 24]	バイトイネーブルに 0 を設定し、アクセスしないでください。
0x006E のビット [24 : 24]	1 を設定してください。
0x0080 のビット [3 : 3] とビット [31 : 24]	0 を設定してください。
0x0080 のビット [12 : 12]	1 を設定してください。
0x0080 のビット [23 : 17]	このレジスタのビット [16 : 16] への書き込みでテクスチャキャッシュのクリアを行うときは 0 を設定してください。それ以外の場合はバイトイネーブルに 0 を設定し、アクセスしないでください。
0x0083 のビット [17 : 16]	0 を設定してください。
0x0093 のビット [17 : 16]	0 を設定してください。
0x009B のビット [17 : 16]	0 を設定してください。
0x00AC のビット [10 : 3]	0x60 を設定してください。
0x00AD のビット [31 : 8]	0xE0C080 を設定してください。
0x00E0 のビット [25 : 24]	0 を設定してください。
0x0100 のビット [25 : 16]	0x0E4 を設定してください。
0x0110 のビット [31 : 1]	0 を設定してください。
0x0111 のビット [31 : 1]	0 を設定してください。
0x011E のビット [24 : 24]	1 を設定してください。
0x01C0 のビット [9 : 8]	0 を設定してください。
0x01C0 のビット [19 : 18]	0 を設定してください。
0x01C0 のビット [29 : 28]	0 を設定してください。
0x01C3 のビット [11 : 8]	0x4 を設定してください。
0x01C3 のビット [31 : 31]	1 を設定してください。
0x01C4 のビット [18 : 18]	1 を設定してください。
0x0229 のビット [9 : 9]	0 を設定してください。
0x0229 のビット [23 : 16]	バイトイネーブルに 0 を設定し、アクセスしないでください。
0x0244 のビット [31 : 8]	バイトイネーブルに 0 を設定し、アクセスしないでください。
0x0245 のビット [7 : 1]	0 を設定してください。
0x0245 のビット [31 : 8]	バイトイネーブルに 0 を設定し、アクセスしないでください。

0x0253 のビット [31 : 16]	バイトイネーブルに 0 を設定し、アクセスしないでください。
0x025E のビット [16 : 16]	バイトイネーブルに 0 を設定し、アクセスしないでください。
0x025F のビット [31 : 1]	0 を設定してください。
0x0280 のビット [31 : 16]	0x7FFF を設定してください。
0x0289 のビット [23 : 16]	バイトイネーブルに 0 を設定し、アクセスしないでください。
0x028A のビット [31 : 16]	0x7FFF を設定してください。
0x028D のビット [31 : 16]	0x0000 を設定してください。
0x02B0 のビット [31 : 16]	0x7FFF を設定してください。
0x02B9 のビット [15 : 8]	0x00 を設定してください。
0x02B9 のビット [23 : 16]	バイトイネーブルに 0 を設定し、アクセスしないでください。
0x02B9 のビット [31 : 24]	0xA0 を設定してください。
0x02BA のビット [31 : 16]	0x7FFF を設定してください。
0x02BD のビット [31 : 16]	0x0000 を設定してください。

8.8.25. ジオメトリシェーダを使用するときのレジスタ設定

SDK が提供するジオメトリシェーダを使用する際に、「8.8.20. ジオメトリシェーダ設定レジスタ(0x0280 ～ 0x02AF ほか)」で説明したレジスタにどのような値を設定するのかを紹介します。

8.8.25.1. ポイントシェーダ

ポイントシェーダの使用時に設定するレジスタの値を以下に示します。

表 8-59. ポイントシェーダ使用時のレジスタ設定値

レジスタ	説明
0x004F のビット [2 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数を設定します。generic 属性は個数に含みません。
0x0050 ～ 0x0056	リンクした頂点シェーダで定義した出力レジスタの属性を 0x0050 から詰めて設定します。 0x0050 には 0x03020100 を設定します。generic 属性としてポイントサイズが出力されますが、本レジスタには影響しません。続けて、小さいインデックスの出力レジスタから順に、定義した属性を 0x0051 から詰めて設定します。 例えば、ポイントスプライトであれば、頂点座標の次にテクスチャ座標が設定されるはずなので、`#pragma output map(texture0, o2.xy)` と定義されていれば、0x0051 には 0x1F1F0D0C と設定します。 未使用属性の部分は 0x1F で各バイトを埋めます。
0x0064	リンクした頂点シェーダで定義した出力レジスタの属性に従います。
0x006F	リンクした頂点シェーダで定義した出力レジスタの属性に従います。
0x0229 のビット [31 : 31]	0 を設定します。
0x0242 のビット [3 : 0]	リンクした頂点シェーダに入力する頂点属性数 - 1 を設定します。
0x024A のビット [3 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数 - 1 を設定します。generic 属性は個数に含まれます。
0x0251 のビット [3 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数 - 1 を設定します。generic 属性は個数に含まれます。

0x0252	0x00000000 を設定します。
0x0254 のビット [4 : 0]	設定する必要はありません。
0x025E のビット [3 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数 - 1 を設定します。 generic 属性は個数に含みません。
0x0280 のビット [15 : 0]	0x0000 を設定します。
0x0281 のビット [23 : 0]	設定する必要はありません。
0x0282 のビット [23 : 0]	設定する必要はありません。
0x0283 のビット [23 : 0]	設定する必要はありません。
0x0284 のビット [23 : 0]	設定する必要はありません。
0x0289 のビット [3 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数 - 1 を設定します。 generic 属性は個数に含まれます。
0x0289 のビット [15 : 8]	0x00 を設定します。
0x0289 のビット [31 : 24]	0x08 を設定します。
0x028D のビット [15 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数を N としたとき、((1 << N) – 1) を設定します。generic 属性は個数に含みません。
0x0290 ～ 0x0293	浮動小数定数を設定するため、{ 0x0290, 0x0291, 0x0292, 0x0293 }に以下の値の組み合わせを設定します。 {0x0000004C, 0x00000000, 0x00003F00, 0x00000000}

ポイントシェーダで設定する予約ユニフォームは特定のレジスタに割り当てられていますので、使用時には以下のレジスタに対しても設定を行わなければなりません。

表 8-60. ポイントシェーダの予約ユニフォームに割り当てられているレジスタ

予約ユニフォーム	割り当てられているレジスタ
dmp_Point.viewport	c67.xy
dmp_Point.distanceAttenuation	b0

8.8.25.2. ラインシェーダ

ラインシェーダの使用時に設定するレジスタの値を以下に示します。

表 8-61. ラインシェーダ使用時のレジスタ設定値

レジスタ	説明
0x004F のビット [2 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数を設定します。
0x0050 ～ 0x0056	リンクした頂点シェーダで定義した出力レジスタの属性を 0x0050 から詰めて設定します。 0x0050 には 0x03020100 を設定します。続けて、小さいインデックスの出力レジスタから順に、定義した属性を 0x0051 から詰めて設定します。 未使用属性の部分は 0x1F で各バイトを埋めます。
0x0064	リンクした頂点シェーダで定義した出力レジスタの属性に従います。
0x006F	リンクした頂点シェーダで定義した出力レジスタの属性に従います。
0x0229 のビット [31 : 31]	0 を設定します。

0x0242 のビット [3 : 0]	リンクした頂点シェーダに入力する頂点属性数 - 1 を設定します。
0x024A のビット [3 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数 - 1 を設定します。
0x0251 のビット [3 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数 - 1 を設定します。
0x0252	0x00000000 を設定します。
0x0254 のビット [4 : 0]	設定する必要はありません。
0x025E のビット [3 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数 - 1 を設定します。
0x0280 のビット [15 : 0]	0x0000 を設定します。ビット [15 : 15] は描画するたびに設定する必要があります。
0x0281 のビット [23 : 0]	設定する必要はありません。
0x0282 のビット [23 : 0]	設定する必要はありません。
0x0283 のビット [23 : 0]	設定する必要はありません。
0x0284 のビット [23 : 0]	設定する必要はありません。
0x0289 のビット [3 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数 - 1 を設定します。
0x0289 のビット [15 : 8]	0x00 を設定します。
0x0289 のビット [31 : 24]	0x08 を設定します。
0x028D のビット [15 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数を N としたとき、((1 << N) – 1) を設定します。
0x0290 ～ 0x0293	浮動小数定数を設定するため、{ 0x0290, 0x0291, 0x0292, 0x0293 }に以下の値の組み合わせを設定します。 {0x0000004C, 0x40800040, 0x00003F00, 0x00000000}

ラインシェーダで設定する予約ユニフォームは特定のレジスタに割り当てられていますので、使用時には以下のレジスタに対しても設定を行わなければなりません。

表 8-62. ラインシェーダの予約ユニフォームに割り当てられているレジスタ

予約ユニフォーム	割り当てられているレジスタ
dmp_Line.width	c67.xyzw

8.8.25.3. シルエットシェーダ

シルエットシェーダの使用時に設定するレジスタの値を以下に示します。

表 8-63. シルエットシェーダ使用時のレジスタ設定値

レジスタ	説明
0x004F のビット [2 : 0]	0x2 を設定します。
0x0050 ～ 0x0056	0x0050 に 0x03020100、0x0051 に 0x0B0A0908、0x0052 ～ 0x0056 に 0x1F1F1F1F を設定します。
0x0064	0x00000000 を設定します。
0x006F	0x00000003 を設定します。
0x0229 のビット [31 : 31]	0 を設定します。

0x0242 のビット [3 : 0]	リンクした頂点シェーダに入力する頂点属性数 - 1 を設定します。
0x024A のビット [3 : 0]	0x2 を設定します。
0x0251 のビット [3 : 0]	0x2 を設定します。
0x0252	0x00000000 を設定します。
0x0254 のビット [4 : 0]	設定する必要はありません。
0x025E のビット [3 : 0]	0x1 を設定します。
0x0280 のビット [15 : 0]	0x0000 を設定します。ビット [15 : 15] は描画するたびに設定する必要があります。
0x0281 のビット [23 : 0]	設定する必要はありません。
0x0282 のビット [23 : 0]	設定する必要はありません。
0x0283 のビット [23 : 0]	設定する必要はありません。
0x0284 のビット [23 : 0]	設定する必要はありません。
0x0289 のビット [3 : 0]	頂点シェーダから出力される頂点属性は頂点座標、カラー、法線の 3 つですので、0x2 を設定します。
0x0289 のビット [15 : 8]	0x00 を設定します。
0x0289 のビット [31 : 24]	0x08 を設定します。
0x028D のビット [15 : 0]	0x0003 を設定します。
0x0290 ～ 0x0293	浮動小数定数を設定するため、{ 0x0290, 0x0291, 0x0292, 0x0293 }に以下の値の組み合わせを設定します。 {0x0000004C, 0x40800040, 0x00003F00, 0x00000000} {0x0000004D, 0x00000000, 0x00004140, 0x00410000}

シルエットシェーダで設定する予約ユニフォームは特定のレジスタに割り当てられていますので、使用時には以下のレジスタに対しても設定を行わなければなりません。

表 8-64. シルエットシェーダの予約ユニフォームに割り当てられているレジスタ

予約ユニフォーム	割り当てられているレジスタ
dmp_Shilhouette.width	c71.xy
dmp_Shilhouette.openEdgeDepthBias	c71.z
dmp_Shilhouette.color	c72.xyzw
dmp_Shilhouette.openEdgeColor	c73.xyzw
dmp_Shilhouette.openEdgeWidth	c74.xyzw
dmp_Shilhouette.acceptEmptyTriangles	b0
dmp_Shilhouette.saceleByW	b1
dmp_Shilhouette.frontFaceCCW	b2
dmp_Shilhouette.openEdgeWidthScaleByW	b3
dmp_Shilhouette.openEdgeDepthBiasScaleByW	b4

8.8.25.4. 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の使用時に設定するレジスタの値を以下に示します。

表 8-65. Catmull-Clark サブディビジョンシェーダ使用時のレジスタ設定値

レジスタ	説明
0x004F のビット [2 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数を設定します。
0x0050 ～ 0x0056	リンクした頂点シェーダで定義した出力レジスタの属性を 0x0050 から詰めて設定します。 0x0050 には 0x03020100 を設定します。続けて、小さいインデックスの出力レジスタから順に、定義した属性を 0x0051 から詰めて設定します。 未使用属性の部分は 0x1F で各バイトを埋めます。
0x0064	リンクした頂点シェーダで定義した出力レジスタの属性に従います。
0x006F	リンクした頂点シェーダで定義した出力レジスタの属性に従います。
0x0229 のビット [31 : 31]	1 を設定します。
0x0242 のビット [3 : 0]	リンクした頂点シェーダに入力する頂点属性数 - 1 を設定します。
0x024A のビット [3 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数 - 1 を設定します。
0x0251 のビット [3 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数 - 1 を設定します。
0x0252	0x00000001 を設定します。
0x0254 のビット [4 : 0]	0x03 を設定します。
0x025E のビット [3 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数 - 1 を設定します。
0x0280 のビット [15 : 0]	0x0000 を設定します。ビット [15 : 15] は描画するたびに設定する必要があります。
0x0281 のビット [23 : 0]	設定する必要はありません。
0x0282 のビット [23 : 0]	リンクしたジオメトリシェーダのプログラムオブジェクトのファイルによって設定する値が変化します。 DMP_subdivision1.obj ならば 0x0212FF を設定します。 DMP_subdivision2.obj ならば 0x0216FF を設定します。 DMP_subdivision3.obj ならば 0x021AFF を設定します。 DMP_subdivision4.obj ならば 0x021EFF を設定します。 DMP_subdivision5.obj ならば 0x0222FF を設定します。 DMP_subdivision6.obj ならば 0x0226FF を設定します。
0x0283 のビット [23 : 0]	設定する必要はありません。
0x0284 のビット [23 : 0]	設定する必要はありません。
0x0289 のビット [3 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数 - 1 を設定します。
0x0289 のビット [15 : 8]	0x01 を設定します。
0x0289 のビット [31 : 24]	0x08 を設定します。
0x028D のビット [15 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数を N としたとき、((1 << N) – 1) を設定します。

0x0290 ～ 0x0293	浮動小数定数を設定するため、{ 0x0290, 0x0291, 0x0292, 0x0293 }に以下の値の組み合わせを設定します。 {0x0000004C, 0x3C80003B, 0x00003C80, 0x003E2000} {0x0000004D, 0x0000003E, 0x00003C00, 0x003D8000} {0x0000004E, 0x4300003D, 0x00003E80, 0x00420000} {0x0000004F, 0x3C60003C, 0xC8003780, 0x00390000} {0x00000050, 0x3D0C0039, 0x80003700, 0x003B8000} {0x00000051, 0x3CC0003C, 0x70003A60, 0x003C2800} {0x00000052, 0x3D16003B, 0x0C003500, 0x003D8000} {0x00000053, 0x3DAAAA39, 0xC71C3C55, 0x55BE2AAA} {0x00000054, 0x3D871C3A, 0x425E3C55, 0x55BE3C71} {0x00000055, 0x3E200039, 0x00003B80, 0x00BDC000} {0x00000056, 0x3D940039, 0x8FFF3C04, 0x00BE3600} {0x00000057, 0x0000003F, 0x00004180, 0x00C0C000} {0x00000058, 0x00000040, 0x00004230, 0x00C17000} {0x00000059, 0x000000C0, 0xC000C350, 0x00428800} さらに、 DMP_subdivision1.obj の場合のみ {0x0000004B, 0x42000041, 0x80004100, 0x00400000} DMP_subdivision2.obj の場合のみ {0x0000004B, 0x42800042, 0x20004180, 0x00408000} DMP_subdivision3.obj の場合のみ {0x0000004B, 0x43000042, 0x80004200, 0x00410000} DMP_subdivision4.obj の場合のみ {0x0000004B, 0x43400042, 0xE0004240, 0x00414000} DMP_subdivision5.obj の場合のみ {0x0000004B, 0x43800043, 0x20004280, 0x00418000} DMP_subdivision6.obj の場合のみ {0x0000004B, 0x43C00043, 0x500042C0, 0x0041C000}

Catmull-Clark サブディビジョンシェーダで設定する予約ユニフォームは特定のレジスタに割り当てられていますので、使用時には以下のレジスタに対しても設定を行わなければなりません。

表 8-66. Catmull-Clark サブディビジョンシェーダの予約ユニフォームに割り当てられているレジスタ

予約ユニフォーム	割り当てられているレジスタ
dmp_Subdivision.Level	c74.x
dmp_Subdivision.fragmentLightingEnabled	b2

8.8.25.5. 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の使用時に設定するレジスタの値を以下に示します。

表 8-67. ループサブディビジョンシェーダ使用時のレジスタ設定値

レジスタ	説明
0x004F のビット [2 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数を設定します。generic 属性は個数に含みません。
0x0050 ～ 0x0056	リンクした頂点シェーダで定義した出力レジスタの属性を 0x0050 から詰めて設定します。 0x0050 には 0x03020100 を設定します。続けて、小さいインデックスの出力レジスタから順に、定義した属性を 0x0051 から詰めて設定します。generic 属性は無視します。 未使用属性の部分は 0x1F で各バイトを埋めます。
0x0064	リンクした頂点シェーダで定義した出力レジスタの属性に従います。
0x006F	リンクした頂点シェーダで定義した出力レジスタの属性に従います。

0x0229 のビット [31 : 31]	1 を設定します。
0x0242 のビット [3 : 0]	リンクした頂点シェーダに入力する頂点属性数 - 1 を設定します。
0x024A のビット [3 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数 - 1 を設定します。generic 属性は個数に含まれます。
0x0251 のビット [3 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数 - 1 を設定します。generic 属性は個数に含まれます。
0x0252	0x00000001 を設定します。
0x0254 のビット [4 : 0]	0x02 を設定します。
0x025E のビット [3 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数 - 1 を設定します。generic 属性は個数に含みません。
0x0280 のビット [15 : 0]	0x0000 を設定します。ビット [15 : 15] は描画するたびに設定する必要があります。
0x0281 のビット [23 : 0]	設定する必要はありません。
0x0282 のビット [23 : 0]	設定する必要はありません。
0x0283 のビット [23 : 0]	設定する必要はありません。
0x0284 のビット [23 : 0]	設定する必要はありません。
0x0289 のビット [3 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数 - 1 を設定します。generic 属性は個数に含まれます。
0x0289 のビット [15 : 8]	0x01 を設定します。
0x0289 のビット [31 : 24]	0x08 を設定します。
0x028D のビット [15 : 0]	リンクした頂点シェーダで定義されている出力レジスタの個数を N としたとき、((1 << N) – 1) を設定します。generic 属性は個数に含みません。
0x0290 ～ 0x0293	浮動小数定数を設定するため、{ 0x0290, 0x0291, 0x0292, 0x0293 }に以下の値の組み合わせを設定します。 {0x00000057, 0x40800040, 0x00003F00, 0x00000000} {0x00000058, 0x3D00003E, 0x000056FF, 0xFF3C0000} {0x00000059, 0x3800003D, 0x00003E80, 0x003D3000} {0x0000005A, 0x3CE0003B, 0x00003D80, 0x00390000} {0x0000005B, 0x3C60003A, 0x80003B80, 0x00000000} {0x0000005C, 0x3C98003D, 0x9C003C80, 0x003DC000} {0x0000005D, 0x3DE0003E, 0x10003D80, 0x003E4000}

ループサブディビジョンシェーダで設定する予約ユニフォームは特定のレジスタに割り当てられていますので、使用時には以下のレジスタに対しても設定を行わなければなりません。

表 8-68. ループサブディビジョンシェーダの予約ユニフォームに割り当てられているレジスタ

予約ユニフォーム	割り当てられているレジスタ
dmp_Subdivision.Level	c86.x
dmp_Subdivision.fragmentLightingEnabled	b0

8.8.25.6. 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の使用時に設定するレジスタの値を以下に示します。

表 8-69. パーティクルシステムシェーダ使用時のレジスタ設定値

レジスタ	説明
0x004F のビット [2 : 0]	0x3 を設定します。
0x0050 ～ 0x0056	0x0050 に 0x03020100、0x0051 に 0x0B0A0908、0x0052 にはテクスチャ座標 2 を使用する場合は 0x17160D0C を、使用しない場合は 0x1F1F0D0C を設定します。 0x0053 ～ 0x0056 には 0x1F1F1F1F を設定します。
0x0064	0x00000000 を設定します。
0x006F	テクスチャ座標 2 を使用する場合は 0x00000503 を、使用しない場合は 0x00000103 を設定します。
0x0229 のビット [31 : 31]	0 を設定します。
0x0242 のビット [3 : 0]	リンクした頂点シェーダに入力する頂点属性数 - 1 を設定します。
0x024A のビット [3 : 0]	0x4 を設定します。
0x0251 のビット [3 : 0]	0x4 を設定します。
0x0252	0x01004302 を設定します。
0x0254 のビット [4 : 0]	設定する必要はありません。
0x025E のビット [3 : 0]	0x2 を設定します。
0x0280 のビット [15 : 0]	0x0000 を設定します。
0x0281 のビット [23 : 0]	設定する必要はありません。
0x0282 のビット [23 : 0]	設定する必要はありません。
0x0283 のビット [23 : 0]	設定する必要はありません。
0x0284 のビット [23 : 0]	0x0100FE を設定します。
0x0289 のビット [3 : 0]	頂点シェーダから出力される頂点属性は頂点座標、制御点のバウンディングボックスサイズ 4 つの計 5 つですので、0x4 を設定します。
0x0289 のビット [15 : 8]	0x01 を設定します。
0x0289 のビット [31 : 24]	0x08 を設定します。
0x028D のビット [15 : 0]	0x0007 を設定します。
0x0290 ～ 0x0293	浮動小数定数を設定するため、{ 0x0290, 0x0291, 0x0292, 0x0293 }に以下の値の組み合わせを設定します。 {0x0000004C, 0x3F0000BF, 0x00003F00, 0x00000000} {0x0000004D, 0x40921F3C, 0x45F34192, 0x1F3E0000} {0x0000005D, 0x3F00003F, 0x0000BC55, 0x55BE0000} {0x0000005E, 0x3811113A, 0x5555B2A0, 0x1AB56C16} {0x0000005F, 0x2C71DE2F, 0xA01AA5AE, 0x64A927E4}

パーティクルシステムシェーダで設定する予約ユニフォームは特定のレジスタに割り当てられていますので、使用時には以下のレジスタに対しても設定を行わなければなりません。

表 8-70. パーティクルシステムシェーダの予約ユニフォームに割り当てられているレジスタ

予約ユニフォーム	割り当てられているレジスタ
dmp_PartSys.color	c26.xyzw ～ c29.xyzw
dmp_PartSys.viewport	c30.xy

dmp_PartSys.pointSize	c31.xy
dmp_PartSys.time	c31.z
dmp_PartSys.speed	c31.w
dmp_PartSys.distanceAttenuation	c32.xyz
dmp_PartSys.countMax	c32.w
dmp_PartSys.randSeed	c33.xyzw
dmp_PartSys.aspect	c34.xyzw ～ c37.xyzw
dmp_PartSys.randomCore	c38.xyzw

8.9. レジスタに設定する値へのフォーマット変換

レジスタに設定する値のいくつかは、アプリケーションで設定した値から内部フォーマットに変換されて設定されています。そのほとんどは 32 ビット浮動小数点数から変換された値となっています。

8.9.1. 24 ビット浮動小数点数への変換

32 ビット浮動小数点数から 24 ビット浮動小数点数(符号部 1 ビット、指数部 7 ビット、仮数部 16 ビット)への変換コードを以下に示します。_inarg に 32 ビット浮動小数点数を渡すと、_outarg の unsigned int 型変数に 24 ビット浮動小数点数が格納されます。

コード 8-23. 24 ビット浮動小数点数への変換コード

```
#define UTL_F2F_16M7E(_inarg, _outarg) \
{ \
    unsigned uval_, m_; \
    int e_; \
    float f_; \
    static const int bias_ = 128 - (1 << (7 - 1)); \
    f_ = (_inarg); \
    uval_ = *(unsigned*)&f_; \
    e_ = (uval_ & 0x7fffffff) ? (((uval_ >> 23) & 0xff) - bias_) : 0; \
    m_ = (uval_ & 0x7fffff) >> (23 - 16); \
    if (e_ >= 0) \
        _outarg = m_ | (e_ << 16) | ((uval_ >> 31) << (16 + 7)); \
    else \
        _outarg = ((uval_ >> 31) << (16 + 7)); \
}
```

8.9.2. 16 ビット浮動小数点数への変換

32 ビット浮動小数点数から 16 ビット浮動小数点数(符号部 1 ビット、指数部 5 ビット、仮数部 10 ビット)への変換コードを以下に示します。_inarg に 32 ビット浮動小数点数を渡すと、_outarg の unsigned int 型変数に 16 ビット浮動小数点数が格納されます。

コード 8-24. 16 ビット浮動小数点数への変換コード

```
#define UTL_F2F_10M5E(_inarg, _outarg) \
{ \
    unsigned uval_, m_; \
    int e_; \
    float f_; \
    static const int bias_ = 128 - (1 << (5 - 1)); \
    f_ = (_inarg); \
    uval_ = *(unsigned*)&f_; \
    e_ = (uval_ & 0x7fffffff) ? (((uval_ >> 23) & 0xff) - bias_) : 0; \
    m_ = (uval_ & 0x7fffff) >> (23 - 10); \
    if (e_ >= 0) \
        _outarg = m_ | (e_ << 10) | ((uval_ >> 31) << (10 + 5)); \
    else \
        _outarg = ((uval_ >> 31) << (10 + 5)); \
}
```

8.9.3. 31 ビット浮動小数点数への変換

32 ビット浮動小数点数から 31 ビット浮動小数点数(符号部 1 ビット、指数部 7 ビット、仮数部 23 ビット)への変換コードを以下に示します。_inarg に 32 ビット浮動小数点数を渡すと、_outarg の unsigned int 型変数に 31 ビット浮動小数点数が格納されます。

コード 8-25. 31 ビット浮動小数点数への変換コード

```
#define UTL_F2F_23M7E(_inarg, _outarg) \
{ \
    unsigned uval_, m_; \
    int e_; \
    float f_; \
    static const int bias_ = 128 - (1 << (7 - 1)); \
    f_ = (_inarg); \
    uval_ = *(unsigned*)&f_; \
    e_ = (uval_ & 0x7fffffff) ? (((uval_ >> 23) & 0xff) - bias_) : 0; \
    m_ = (uval_ & 0x7fffff) >> (23 - 23); \
    if (e_ >= 0) \
        _outarg = m_ | (e_ << 23) | ((uval_ >> 31) << (23 + 7)); \
    else \
        _outarg = ((uval_ >> 31) << (23 + 7)); \
}
```

8.9.4. 20 ビット浮動小数点数への変換

32 ビット浮動小数点数から 20 ビット浮動小数点数(符号部 1 ビット、指数部 7 ビット、仮数部 12 ビット)への変換コードを以下に示します。_inarg に 32 ビット浮動小数点数を渡すと、_outarg の unsigned int 型変数に 20 ビット浮動小数点数が格納されます。

コード 8-26. 20 ビット浮動小数点数への変換コード

```
#define UTL_F2F_12M_7E(_inarg, _outarg) \
{ \
    unsigned uval_, m_; \
    int e_; \
    float f_; \
    static const int bias_ = 128 - (1 << (7 - 1)); \
    f_ = (_inarg); \
    uval_ = *(unsigned*)&f_; \
    e_ = (uval_ & 0x7fffffff) ? (((uval_ >> 23) & 0xff) - bias_) : 0; \
    m_ = (uval_ & 0x7fffff) >> (23 - 12); \
    if (e_ >= 0) \
        _outarg = m_ | (e_ << 12) | ((uval_ >> 31) << (12 + 7)); \
    else \
        _outarg = ((uval_ >> 31) << (12 + 7)); \
}
```

8.9.5. 小数部 7 ビットの符号つき 8 ビット固定小数点数への変換

32 ビット浮動小数点数から 8 ビット固定小数点数(小数部 7 ビットの符号つき)への変換コードを以下に示します。負の値は 2 の補数で表現します。_inarg に 32 ビット浮動小数点数を渡すと、_outarg に 8 ビット固定小数点数が格納されます。

コード 8-27. 小数部 7 ビットの符号つき 8 ビット固定小数点数への変換コード

```
#define UTL_F2FX_8W_1I_T(_inarg, _outarg) \
{ \
    float f_; \
    unsigned v_; \
    f_ = (_inarg); \
    v_ = *(unsigned*)&f_; \
    if (f_ == 0.f || (v_ & 0x7f800000) == 0x7f800000) \
        _outarg = 0; \
    else \
    { \
        f_ += 0.5f * (1 << 1); \
        f_ *= 1 << (8 - 1); \
        if (f_ < 0) \
            f_ = 0; \
        else if (f_ >= (1 << 8)) \
            f_ = (1 << 8) - 1; \
        if (f_ >= (1 << (8 - 1))) \
            _outarg = (unsigned)(f_ - (1 << (8 - 1))); \
        else \
            _outarg = (unsigned)(f_ + (1 << (8 - 1))); \
    } \
}
```

8.9.6. 小数部 11 ビットの符号つき 12 ビット固定小数点数への変換

32 ビット浮動小数点数から 12 ビット固定小数点数(小数部 11 ビットの符号つき)への変換コードを以下に示します。小数部は絶対値ですので、負の値は 2 の補数ではありません。_inarg に 32 ビット浮動小数点数を渡すと、_outarg に 12 ビット固定小数点数が格納されます。

コード 8-28. 小数部 11 ビットの符号つき 12 ビット固定小数点数への変換コード

```
#define UTL_F2FX_12W_1I_F(_inarg, _outarg) \
{ \
    float f_; \
    unsigned v_; \
    f_ = (_inarg); \
    v_ = *(unsigned*)&f_; \
    if (f_ == 0.f || (v_ & 0x7f800000) == 0x7f800000) \
        _outarg = 0; \
    else \
    { \
        f_ *= (1 << (12 - 1)); \
        if (f_ < 0) \
        { \
            _outarg = 1 << (12 - 1); \
            f_ = -f_; \
        } \
        else \
            _outarg = 0; \
        if (f_ >= (1 << (12 - 1))) f_ = (1 << (12 - 1)) - 1; \
            _outarg |= (unsigned)(f_); \
```

```
    } \
}
```

8.9.7. 小数部 11 ビットの符号つき 12 ビット固定小数点数への変換 2

32 ビット浮動小数点数から 12 ビット固定小数点数(小数部 11 ビットの符号つき)への変換コードを以下に示します。負の値は 2 の補数で表現します。_inarg に 32 ビット浮動小数点数を渡すと、_outarg に 12 ビット固定小数点数が格納されます。

コード 8-29. 小数部 11 ビットの符号つき 12 ビット固定小数点数への変換コード 2

```
#define UTL_F2FX_12W_1I_T(_inarg, _outarg) \
{ \
    float f_; \
    unsigned v_; \
    f_ = (_inarg); \
    v_ = *(unsigned*)&f_; \
    if (f_ == 0.f || (v_ & 0x7f800000) == 0x7f800000) \
        _outarg = 0; \
    else \
    { \
        f_ += 0.5f * (1 << 1); \
        f_ *= 1 << (12 - 1); \
        if (f_ < 0) \
            f_ = 0; \
        else if (f_ >= (1 << 12)) \
            f_ = (1 << 12) - 1; \
        if (f_ >= (1 << (12 - 1))) \
            _outarg = (unsigned)(f_ - (1 << (12 - 1))); \
        else \
            _outarg = (unsigned)(f_ + (1 << (12 - 1))); \
    } \
}
```

8.9.8. 小数部 8 ビットの符号つき 13 ビット固定小数点数への変換

32 ビット浮動小数点数から 13 ビット固定小数点数(小数部 8 ビットの符号つき)への変換コードを以下に示します。負の値は 2 の補数で表現します。_inarg に 32 ビット浮動小数点数を渡すと、_outarg に 13 ビット固定小数点数が格納されます。

コード 8-30. 小数部 8 ビットの符号つき 13 ビット固定小数点数への変換コード

```
#define UTL_F2FX_13W_5I_T(_inarg, _outarg) \
{ \
    float f_; \
    unsigned v_; \
    f_ = (_inarg); \
    v_ = *(unsigned*)&f_; \
    if (f_ == 0.f || (v_ & 0x7f800000) == 0x7f800000) \
        _outarg = 0; \
    else \
    { \
        f_ += 0.5f * (1 << 5); \
```

```
        f_ *= 1 << (13 - 5); \
        if (f_ < 0) \
            f_ = 0; \
        else if (f_ >= (1 << 13)) \
            f_ = (1 << 13) - 1; \
        if (f_ >= (1 << (13 - 1))) \
            _outarg = (unsigned)(f_ - (1 << (13 - 1))); \
        else \
            _outarg = (unsigned)(f_ + (1 << (13 - 1))); \
    } \
}
```

8.9.9. 小数部 11 ビットの符号つき 13 ビット固定小数点数への変換

32 ビット浮動小数点数から 13 ビット固定小数点数(小数部 11 ビットの符号つき)への変換コードを以下に示します。負の値は 2 の補数で表現します。_inarg に 32 ビット浮動小数点数を渡すと、_outarg に 13 ビット固定小数点数が格納されます。

コード 8-31. 小数部 11 ビットの符号つき 13 ビット固定小数点数への変換コード

```
#define UTL_F2FX_13W_2I_T(_inarg, _outarg) \
{ \
    float f_; \
    unsigned v_; \
    f_ = (_inarg); \
    v_ = *(unsigned*)&f_; \
    if (f_ == 0.f || (v_ & 0x7f800000) == 0x7f800000) \
        _outarg = 0; \
    else \
    { \
        f_ += 0.5f * (1 << 2); \
        f_ *= 1 << (13 - 2); \
        if (f_ < 0) \
            f_ = 0; \
        else if (f_ >= (1 << 13)) \
            f_ = (1 << 13) - 1; \
        if (f_ >= (1 << (13 - 1))) \
            _outarg = (unsigned)(f_ - (1 << (13 - 1))); \
        else \
            _outarg = (unsigned)(f_ + (1 << (13 - 1))); \
    } \
}
```

8.9.10. 小数部 12 ビットの符号つき 16 ビット固定小数点数への変換

32 ビット浮動小数点数から 16 ビット固定小数点数(小数部 12 ビットの符号つき)への変換コードを以下に示します。負の値は 2 の補数で表現します。_inarg に 32 ビット浮動小数点数を渡すと、_outarg に 16 ビット固定小数点数が格納されます。

コード 8-32. 小数部 12 ビットの符号つき 16 ビット固定小数点数への変換コード

```
#define UTL_F2FX_16W_4I_T(_inarg, _outarg) \
{ \
    float f_; \
    unsigned v_; \
    f_ = (_inarg); \
    v_ = *(unsigned*)&f_; \
    if (f_ == 0.f || (v_ & 0x7f800000) == 0x7f800000) \
        _outarg = 0; \
    else \
    { \
        f_ += 0.5f * (1 << 4); \
        f_ *= 1 << (16 - 4); \
        if (f_ < 0) \
            f_ = 0; \
        else if (f_ >= (1 << 16)) \
            f_ = (1 << 16) - 1; \
        if (f_ >= (1 << (16 - 1))) \
            _outarg = (unsigned)(f_ - (1 << (16 - 1))); \
        else \
            _outarg = (unsigned)(f_ + (1 << (16 - 1))); \
    } \
}
```

8.9.11. 小数部 0 ビットの符号なし 8 ビット固定小数点数への変換

32 ビット浮動小数点数から 8 ビット固定小数点数（小数部 0 ビットの符号なし）への変換コードを以下に示します。_inarg に 32 ビット浮動小数点数を渡すと、_outarg に 8 ビット固定小数点数が格納されます。

コード 8-33. 小数部 0 ビットの符号なし 8 ビット固定小数点数への変換コード

```
#define UTL_F2UFX_8W_8I(_inarg, _outarg) \
{ \
    float f_ = (_inarg); \
    unsigned val_; \
    unsigned v_ = *(unsigned*)&f_; \
    if (f_ <= 0 || (v_ & 0x7f800000) == 0x7f800000) \
        val_ = 0; \
    else \
    { \
        f_ *= 1 << (8 - 8); \
        if (f_ >= (1 << 8)) \
            val_ = (1 << 8) - 1; \
        else \
            val_ = (unsigned)(f_); \
    } \
    (_outarg) = val_; \
}
```

8.9.12. 小数部 11 ビットの符号なし 11 ビット固定小数点数への変換

32 ビット浮動小数点数から 11 ビット固定小数点数（小数部 11 ビットの符号なし）への変換コードを以下に示します。_inarg に 32 ビット浮動小数点数を渡すと、_outarg に 11 ビット固定小数点数が格納されます。

コード 8-34. 小数部 11 ビットの符号なし 11 ビット固定小数点数への変換コード

```
#define UTL_F2UFX_11W_0I(_inarg, _outarg) \
{ \
    float f_ = (_inarg); \
    unsigned val_; \
    unsigned v_ = *(unsigned*)&f_; \
    if (f_ <= 0 || (v_ & 0x7f800000) == 0x7f800000) \
        val_ = 0; \
    else \
    { \
        f_ *= 1 << (11 - 0); \
        if (f_ >= (1 << 11)) \
            val_ = (1 << 11) - 1; \
        else \
            val_ = (unsigned)(f_); \
    } \
    (_outarg) = val_; \
}
```

8.9.13. 小数部 12 ビットの符号なし 12 ビット固定小数点数への変換

32 ビット浮動小数点数から 12 ビット固定小数点数(小数部 12 ビットの符号なし)への変換コードを以下に示します。
_inarg に 32 ビット浮動小数点数を渡すと、_outarg に 12 ビット固定小数点数が格納されます。

コード 8-35. 小数部 12 ビットの符号なし 12 ビット固定小数点数への変換コード

```
#define UTL_F2UFX_12W_0I(_inarg, _outarg) \
{ \
    float f_ = (_inarg); \
    unsigned val_; \
    unsigned v_ = *(unsigned*)&f_; \
    if (f_ <= 0 || (v_ & 0x7f800000) == 0x7f800000) \
        val_ = 0; \
    else \
    { \
        f_ *= 1 << (12 - 0); \
        if (f_ >= (1 << 12)) \
            val_ = (1 << 12) - 1; \
        else \
            val_ = (unsigned)(f_); \
    } \
    (_outarg) = val_; \
}
```

8.9.14. 小数部 24 ビットの符号なし 24 ビット固定小数点数への変換

32 ビット浮動小数点数から 24 ビット固定小数点数(小数部 24 ビットの符号なし)への変換コードを以下に示します。
_inarg に 32 ビット浮動小数点数を渡すと、_outarg に 24 ビット固定小数点数が格納されます。

コード 8-36. 小数部 24 ビットの符号なし 24 ビット固定小数点数への変換コード

```
#define UTL_F2UFX_24W_0I(_inarg, _outarg) \
{ \
    float f_ = (_inarg); \
    unsigned val_; \
    unsigned v_ = *(unsigned*)&f_; \
    if (f_ <= 0 || (v_ & 0x7f800000) == 0x7f800000) \
        val_ = 0; \
    else \
    { \
        f_ *= 1 << (24 - 0); \
        if (f_ >= (1 << 24)) \
            val_ = (1 << 24) - 1; \
        else \
            val_ = (unsigned)(f_); \
    } \
    (_outarg) = val_; \
}
```

8.9.15. 小数部 8 ビットの符号なし 24 ビット固定小数点数への変換

32 ビット浮動小数点数から 24 ビット固定小数点数（小数部 8 ビットの符号なし）への変換コードを以下に示します。
_inarg に 32 ビット浮動小数点数を渡すと、_outarg に 24 ビット固定小数点数が格納されます。

コード 8-37. 小数部 8 ビットの符号なし 24 ビット固定小数点数への変換コード

```
#define UTL_F2UFX_24W_16I(_inarg, _outarg) \
{ \
    float f_ = (_inarg); \
    unsigned val_; \
    unsigned v_ = *(unsigned*)&f_; \
    if (f_ <= 0 || (v_ & 0x7f800000) == 0x7f800000) \
        val_ = 0; \
    else \
    { \
        f_ *= 1 << (24 - 16); \
        if (f_ >= (1 << 24)) \
            val_ = (1 << 24) - 1; \
        else \
            val_ = (unsigned)(f_); \
    } \
    (_outarg) = val_; \
}
```

8.9.16. 浮動小数点数（0 ～ 1）から符号なし 8 ビット整数への変換

0 ～ 1 の範囲の 32 ビット浮動小数点数から符号なし 8 ビット整数への変換コードを以下に示します。f に 32 ビット浮動小数点数を渡すと、符号なし 8 ビット整数が返ります。

コード 8-38. 浮動小数点数(0 ～ 1)から符号なし 8 ビット整数への変換コード

```
((unsigned)(0.5f + (f) * (float)((1 << 8) - 1)))
```

8.9.17. 浮動小数点数(0 ～ 1)から符号なし 8 ビット整数への変換 2

0 ～ 1 の範囲の 32 ビット浮動小数点数から符号なし 8 ビット整数への変換コードを以下に示します。f に 32 ビット浮動小数点数を渡すと、符号なし 8 ビット整数が返ります。

コード 8-39. 浮動小数点数(0 ～ 1)から符号なし 8 ビット整数への変換コード 2

```
((unsigned)((f) * (float)((1 << 8) - 1)))
```

8.9.18. 浮動小数点数(-1 ～ 1)から符号つき 8 ビット整数への変換

-1 ～ 1 の範囲の 32 ビット浮動小数点数から符号つき 8 ビット整数への変換コードを以下に示します。f に 32 ビット浮動小数点数を渡すと、符号つき 8 ビット整数が返ります。

コード 8-40. 浮動小数点数(-1 ～ 1)から符号つき 8 ビット整数への変換コード

```
(((unsigned int)(fabs(127.f * (f))) & 0x7f)|(f < 0 ? 0x80 : 0))
```

8.9.19. 16 ビット浮動小数点数から 32 ビット浮動小数点数への変換

16 ビット浮動小数点数(符号部 1 ビット、指数部 5 ビット、仮数部 10 ビット)から 32 ビット浮動小数点数への変換コードを以下に示します。_inarg に unsigned int 型変数に格納された 16 ビット浮動小数点数を渡すと、_outarg の float 型変数に 32 ビット浮動小数点数が格納されます。

コード 8-41. 16 ビット浮動小数点数から 32 ビット浮動小数点数への変換コード

```
#define UTL_U2F_10M5E(_inarg, _outarg) \
{ \
    int e_; \
    unsigned m_; \
    unsigned u_ = (_inarg); \
    const int width_ = 10 + 5 + 1; \
    const int bias_ = 128 - (1 << (5 - 1)); \
    e_ = (u_ >> 10) & ((1 << 5) - 1); \
    m_ = u_ & ((1 << 10) - 1); \
    if (u_ & ((1 << (width_ - 1)) - 1)) \
        u_ = ((u_ >> (5 + 10)) << 31) | (m_ << (23 - 10)) | ((e_ + bias_) << 23); \
    else \
        u_ = ((u_ >> (5 + 10)) << 31); \
    (_outarg) = *(float*)&u_; \
}
```

8.10. レジスタマップ

「8.8. PICA レジスタ情報」で紹介したレジスタ情報のレジスタマップを示します。レジスタマップは、関連するページ、関数・予約ユニフォーム、更新される可能性のあるステートをアドレスごとにまとめて表にしています。

8.10.1. レジスタ 0x0010 ～ 0x00FF のレジスタマップ

表 8-71. レジスタマップ(アドレス 0x0010 ～ 0x00FF)

アドレス	参照先	関数・予約ユニフォーム	NN_GX_STATE_
0x0010	8.8.22	nngxSplitDrawCmdlist()、nngxTransferRenderImage()	–
0x0040	8.8.15	glCullFace()、glDisable(GL_CULL_FACE)、glEnable(GL_CULL_FACE)、glFrontFace()	OTHERS
0x0041～0x0044	8.8.10	glViewport()	OTHERS
0x0047	8.8.9.4	dmp_FragOperation.enableClippingPlane	FSUNIFORM
0x0048～0x004B		dmp_FragOperation.clippingPlane	FSUNIFORM
0x004D	8.8.9.3	dmp_FragOperation.wScale、glDepthRangef()、glDisable(GL_POLYGON_OFFSET_FILL)、glEnable(GL_POLYGON_OFFSET_FILL)、glPolygonOffset()	FSUNIFORM TRIOFFSET
0x004E			
0x004F	8.8.1.10 8.8.20.8	出力レジスタの使用数設定	SHADERPROGRAM
0x0050～0x0056	8.8.1.12 8.8.20.10	出力レジスタの属性設定	SHADERPROGRAM
0x0061	8.8.13	glEarlyDepthFuncDMP()	OTHERS
0x0062		glDisable(GL_EARLY_DEPTH_TEST_DMP)、glEnable(GL_EARLY_DEPTH_TEST_DMP)	OTHERS
0x0063		glClear(GL_EARLY_DEPTH_BUFFER_BIT_DMP)	–
0x0064	8.8.1.12 8.8.20.10	出力レジスタの属性設定	SHADERPROGRAM
0x0065～0x0067	8.8.16	glDisable(GL_SCISSOR_TEST)、glEnable(GL_SCISSOR_TEST)、glScissor()	SCISSOR
0x0068	8.8.10	glViewport()	OTHERS
0x006A	8.8.13	glClearEarlyDepthDMP()	OTHERS
0x006D	8.8.9.3	dmp_FragOperation.wScale	FSUNIFORM
0x006E	8.8.3	glRenderbufferStorage()、glTexture2DImage2D()	FRAMEBUFFER
0x006F	8.8.1.13 8.8.20.11	出力属性のクロック制御	SHADERPROGRAM
0x0080	8.8.6.2	dmp_Texture[i].samplerType (i=0,1,2)	TEXTURE
	8.8.6.3	dmp_Texture[2].texcoord、dmp_Texture[3].texcoord、dmp_Texture[3].samplerType	FSUNIFORM
	8.8.19	glDrawArrays()、glDrawElements()	–
0x0081	8.8.6.8	glTexParameter()	TEXTURE
0x0082	8.8.6.6	glTexImage2D()、glCompressedTexImage2D()、glCopyTexImage2D()	TEXTURE
0x0083	8.8.6.2	dmp_Texture[0].samplerType	TEXTURE

	8.8.6.7	glTexImage2D()、glCompressedTexImage2D()、glCopyTexImage2D()	TEXTURE
	8.8.6.8	glTexParameter()	TEXTURE
0x0084	8.8.6.8	glTexParameter()、glCopyTexImage2D()、glCompressedTexImage2D()、glTexImage2D()	TEXTURE
0x0085～0x008A	8.8.2	glTexImage2D()、glCompressedTexImage2D()、glCopyTexImage2D()	TEXTURE
0x008B	8.8.6.1	dmp_Texture[0].perspectiveShadow、dmp_Texture[0].shadowZBias	FSUNIFORM
0x008E	8.8.6.7	glTexImage2D()、glCompressedTexImage2D()、glCopyTexImage2D()	TEXTURE
0x008F	8.8.5.1	dmp_FragmentLighting.enabled	FSUNIFORM
0x0091	8.8.6.8	glTexParameter()	TEXTURE
0x0092	8.8.6.6	glTexImage2D()、glCompressedTexImage2D()、glCopyTexImage2D()	TEXTURE
0x0093	8.8.6.7	glTexImage2D()、glCompressedTexImage2D()、glCopyTexImage2D()	TEXTURE
	8.8.6.8	glTexParameter()	TEXTURE
0x0094	8.8.6.8	glTexImage2D()、glCompressedTexImage2D()、glCopyTexImage2D()	TEXTURE
0x0095	8.8.2	glTexImage2D()、glCompressedTexImage2D()、glCopyTexImage2D()	TEXTURE
0x0096	8.8.6.7	glTexImage2D()、glCompressedTexImage2D()、glCopyTexImage2D()	TEXTURE
0x0099	8.8.6.8	glTexParameter()	TEXTURE
0x009A	8.8.6.6	glTexImage2D()、glCompressedTexImage2D()、glCopyTexImage2D()	TEXTURE
0x009B	8.8.6.7	glTexImage2D()、glCompressedTexImage2D()、glCopyTexImage2D()	TEXTURE
	8.8.6.8	glTexParameter()	TEXTURE
0x009C	8.8.6.8	glTexParameter()、glCopyTexImage2D()、glCompressedTexImage2D()、glTexImage2D()	TEXTURE
0x009D	8.8.2	glTexImage2D()、glCompressedTexImage2D()、glCopyTexImage2D()	TEXTURE
0x009E	8.8.6.7	glTexImage2D()、glCompressedTexImage2D()、glCopyTexImage2D()	TEXTURE
0x00A8	8.8.6.4	dmp_Texture[3].ptClampU、dmp_Texture[3].ptClampV、dmp_Texture[3].ptRgbMap、dmp_Texture[3].ptAlphaMap、dmp_Texture[3].ptAlphaSeparate、dmp_Texture[3].ptNoiseEnable、dmp_Texture[3].ptShiftU、dmp_Texture[3].ptShiftV、dmp_Texture[3].ptTexBias	FSUNIFORM
0x00A9～0x00AB		dmp_Texture[3].ptNoiseU、dmp_Texture[3].ptNoiseV	FSUNIFORM

0x00AC		dmp_Texture[3].ptMinFilter、 dmp_Texture[3].ptTexWidth、 dmp_Texture[3].ptTexBias	FSUNIFORM
0x00AD		dmp_Texture[3].ptTexOffset	FSUNIFORM
0x00AF	8.8.6.5	dmp_Texture[3].ptSampler{RgbMap,AlphaMap,NoiseMap,R,G,B,A}	LUT
0x00B0 ～0x00B7		参照テーブルへのデータ設定	LUT
0x00C0	8.8.4	dmp_TexEnv[0].srcRgb、dmp_TexEnv[0].srcAlpha	FSUNIFORM
0x00C1		dmp_TexEnv[0].operandRgb、 dmp_TexEnv[0].operandAlpha	FSUNIFORM
0x00C2		dmp_TexEnv[0].combineRgb、 dmp_TexEnv[0].combineAlpha	FSUNIFORM
0x00C3		dmp_TexEnv[0].constRgba	FSUNIFORM
0x00C4		dmp_TexEnv[0].scaleRgb、dmp_TexEnv[0].scaleAlpha	FSUNIFORM
0x00C8	8.8.4	dmp_TexEnv[1].srcRgb、dmp_TexEnv[1].srcAlpha	FSUNIFORM
0x00C9		dmp_TexEnv[1].operandRgb、 dmp_TexEnv[1].operandAlpha	FSUNIFORM
0x00CA		dmp_TexEnv[1].combineRgb、 dmp_TexEnv[1].combineAlpha	FSUNIFORM
0x00CB		dmp_TexEnv[1].constRgba	FSUNIFORM
0x00CC		dmp_TexEnv[1].scaleRgb、dmp_TexEnv[1].scaleAlpha	FSUNIFORM
0x00D0	8.8.4	dmp_TexEnv[2].srcRgb、dmp_TexEnv[2].srcAlpha	FSUNIFORM
0x00D1		dmp_TexEnv[2].operandRgb、 dmp_TexEnv[2].operandAlpha	FSUNIFORM
0x00D2		dmp_TexEnv[1].combineRgb、 dmp_TexEnv[1].combineAlpha	FSUNIFORM
0x00D3		dmp_TexEnv[2].constRgba	FSUNIFORM
0x00D4		dmp_TexEnv[2].scaleRgb、dmp_TexEnv[2].scaleAlpha	FSUNIFORM
0x00D8	8.8.4	dmp_TexEnv[3].srcRgb、dmp_TexEnv[3].srcAlpha	FSUNIFORM
0x00D9		dmp_TexEnv[3].operandRgb、 dmp_TexEnv[3].operandAlpha	FSUNIFORM
0x00DA		dmp_TexEnv[3].combineRgb、 dmp_TexEnv[3].combineAlpha	FSUNIFORM
0x00DB		dmp_TexEnv[3].constRgba	FSUNIFORM
0x00DC		dmp_TexEnv[3].scaleRgb、dmp_TexEnv[3].scaleAlpha	FSUNIFORM
0x00E0	8.8.4.1	dmp_TexEnv[i].bufferInput (i=1,2,3,4)	FSUNIFORM
	8.8.7.1	dmp_Gas.shadingDensitySrc	FSUNIFORM
	8.8.8.1	dmp_Fog.mode、dmp_Fog.zFlip	FSUNIFORM
0x00E1	8.8.8.1	dmp_Fog.color	FSUNIFORM
0x00E4	8.8.7.1	dmp_Gas.attenuation	FSUNIFORM

0x00E5		dmp_Gas.accMax	FSUNIFORM
0x00E6	8.8.8.2	dmp_Fog.sampler	LUT
0x00E8 ～0x00EF		参照テーブルへのデータ設定	LUT
0x00F0	8.8.4	dmp_TexEnv[4].srcRgb、dmp_TexEnv[4].srcAlpha	FSUNIFORM
0x00F1		dmp_TexEnv[4].operandRgb、dmp_TexEnv[4].operandAlpha	FSUNIFORM
0x00F2		dmp_TexEnv[4].combineRgb、dmp_TexEnv[4].combineAlpha	FSUNIFORM
0x00F3		dmp_TexEnv[4].constRgba	FSUNIFORM
0x00F4		dmp_TexEnv[4].scaleRgb、dmp_TexEnv[4].scaleAlpha	FSUNIFORM
0x00F8	8.8.4	dmp_TexEnv[5].srcRgb、dmp_TexEnv[5].srcAlpha	FSUNIFORM
0x00F9		dmp_TexEnv[5].operandRgb、dmp_TexEnv[5].operandAlpha	FSUNIFORM
0x00FA		dmp_TexEnv[5].combineRgb、dmp_TexEnv[5].combineAlpha	FSUNIFORM
0x00FB		dmp_TexEnv[5].constRgba	FSUNIFORM
0x00FC		dmp_TexEnv[5].scaleRgb、dmp_TexEnv[5].scaleAlpha	FSUNIFORM
0x00FD	8.8.4.1	dmp_TexEnv[0].bufferColor	FSUNIFORM

8.10.2. レジスタ 0x0100 ～ 0x01FF のレジスタマップ

表 8-72. レジスタマップ（アドレス 0x0100 ～ 0x01FF）

アドレス	参照先	関数・予約ユニフォーム	NN_GX_STATE_
0x0100	8.8.9.1	dmp_FragOperation.mode	FSUNIFORM
	8.8.12	glDisable(GL_BLEND)、glEnable(GL_BLEND)、glDisable(GL_COLOR_LOGIC_OP)、glEnable(GL_COLOR_LOGIC_OP)	OTHERS
0x0101	8.8.12	glBlendEquation()、glBlendEquationSeparate()、glBlendFunc()、glBlendFuncSeparate()	OTHERS
0x0102		glLogicOp()	OTHERS
0x0103		glBlendColor()	OTHERS
0x0104	8.8.9.5	dmp_FragOperation.enableAlphaTest、dmp_FragOperation.alphaTestFunc、dmp_FragOperation.alphaRefValue	FSUNIFORM
0x0105	8.8.14	glDisable(GL_STENCIL_TEST)、glEnable(GL_STENCIL_TEST)、glStencilFunc()、glStencilMask()	OTHERS
0x0106		glStencilOp()	OTHERS
0x0107	8.8.11	glDisable(GL_DEPTH_TEST)、glEnable(GL_DEPTH_TEST)、glDepthFunc()、glDepthMask()	OTHERS

	8.8.17	glColorMask()	OTHERS
0x0110	8.8.21	glFinish()、glFlush()、nngxSplitDrawCmdlist()、nngxTransferRenderImage()	FRAMEBUFFER FBACCESS
0x0111		glFinish()、glFlush()、glDrawArrays()、glDrawElements()、nngxSplitDrawCmdlist()、nngxTransferRenderImage()	FRAMEBUFFER FBACCESS
0x0112～0x0115	8.8.9.6	dmp_FragOperation.mode、glDisable(GL_BLEND)、glEnable(GL_BLEND)、glDisable(GL_COLOR_LOGIC_OP)、glEnable(GL_COLOR_LOGIC_OP)、glColorMask()、glDisable(GL_DEPTH_TEST)、glEnable(GL_DEPTH_TEST)、glDepthMask()、glDisable(GL_STENCIL_TEST)、glEnable(GL_STENCIL_TEST)、glStencilMask()	FBACCESS
0x0116	8.8.3	glRenderbufferStorage()、glTexture2DImage2D()	FRAMEBUFFER
0x0117			
0x0118	8.8.13	glDisable(GL_EARLY_DEPTH_TEST_DMP)、glEnable(GL_EARLY_DEPTH_TEST_DMP)	OTHERS
0x011B	8.8.18	glRenderBlockModeDMP()	OTHERS
0x011C～0x011E	8.8.3	glRenderbufferStorage()、glTexImage2D()、glTexture2DImage2D()	FRAMEBUFFER
0x0120	8.8.7.1	dmp_Gas.lightXY	FSUNIFORM
0x0121		dmp_Gas.lightZ	FSUNIFORM
0x0122			
0x0123	8.8.7.2	dmp_Gas.sampler{TR,TG,TB}	LUT
0x0124		参照テーブルへのデータ設定	LUT
0x0125	8.8.7.1	dmp_Gas.autoAcc	–
0x0126	8.8.7.1	dmp_Gas.deltaZ	FSUNIFORM
	8.8.11	glDepthFunc()	OTHERS
0x0130	8.8.9.2	dmp_FragOperation.penumbraScale、dmp_FragOperation.penumbraBias	FSUNIFORM
0x0140	8.8.5.3	dmp_FragmentMaterial.specular0、dmp_FragmentLightSource[0].specular0	FSUNIFORM
0x0141		dmp_LightEnv.lutEnabledRefl、dmp_FragmentMaterial.specular1、dmp_FragmentLightSource[0].specular1	FSUNIFORM
0x0142		dmp_FragmentMaterial.diffuse、dmp_FragmentLightSource[0].diffuse	FSUNIFORM
0x0143		dmp_FragmentMaterial.ambient、dmp_FragmentLightSource[0].ambient	FSUNIFORM
0x0144		dmp_FragmentLightSource[0].position	FSUNIFORM
0x0145			
0x0146		dmp_FragmentLightSource[0].spotDirection	FSUNIFORM
0x0147			

0x0149		dmp_FragmentLightSource[0].position、 dmp_FragmentLightSource[0].twoSideDiffuse、 dmp_FragmentLightSource[0].geomFactor0、 dmp_FragmentLightSource[0].geomFactor1	FSUNIFORM
0x014A		dmp_FragmentLightSource[0].distanceAttenuationBias	FSUNIFORM
0x014B		dmp_FragmentLightSource[0].distanceAttenuationScale	FSUNIFORM
0x0150	8.8.5.3	dmp_FragmentMaterial.specular0、 dmp_FragmentLightSource[1].specular0	FSUNIFORM
0x0151		dmp_LightEnv.lutEnabledRefl、 dmp_FragmentMaterial.specular1、 dmp_FragmentLightSource[1].specular1	FSUNIFORM
0x0152		dmp_FragmentMaterial.diffuse、 dmp_FragmentLightSource[1].diffuse	FSUNIFORM
0x0153		dmp_FragmentMaterial.ambient、 dmp_FragmentLightSource[1].ambient	FSUNIFORM
0x0154		dmp_FragmentLightSource[1].position	FSUNIFORM
0x0155			
0x0156		dmp_FragmentLightSource[1].spotDirection	FSUNIFORM
0x0157			
0x0159		dmp_FragmentLightSource[1].position、 dmp_FragmentLightSource[1].twoSideDiffuse、 dmp_FragmentLightSource[1].geomFactor0、 dmp_FragmentLightSource[1].geomFactor1	FSUNIFORM
0x015A		dmp_FragmentLightSource[1].distanceAttenuationBias	FSUNIFORM
0x015B		dmp_FragmentLightSource[1].distanceAttenuationScale	FSUNIFORM
0x0160	8.8.5.3	dmp_FragmentMaterial.specular0、 dmp_FragmentLightSource[2].specular0	FSUNIFORM
0x0161		dmp_LightEnv.lutEnabledRefl、 dmp_FragmentMaterial.specular1、 dmp_FragmentLightSource[2].specular1	FSUNIFORM
0x0162		dmp_FragmentMaterial.diffuse、 dmp_FragmentLightSource[2].diffuse	FSUNIFORM
0x0163		dmp_FragmentMaterial.ambient、 dmp_FragmentLightSource[2].ambient	FSUNIFORM
0x0164		dmp_FragmentLightSource[2].position	FSUNIFORM
0x0165			
0x0166		dmp_FragmentLightSource[2].spotDirection	FSUNIFORM
0x0167			
0x0169		dmp_FragmentLightSource[2].position、 dmp_FragmentLightSource[2].twoSideDiffuse、 dmp_FragmentLightSource[2].geomFactor0、 dmp_FragmentLightSource[2].geomFactor1	FSUNIFORM
0x016A		dmp_FragmentLightSource[2].distanceAttenuationBias	FSUNIFORM

0x016B		dmp_FragmentLightSource[2].distanceAttenuationScale	FSUNIFORM
0x0170		dmp_FragmentMaterial.specular0、 dmp_FragmentLightSource[3].specular0	FSUNIFORM
0x0171		dmp_LightEnv.lutEnabledRefl、 dmp_FragmentMaterial.specular1、 dmp_FragmentLightSource[3].specular1	FSUNIFORM
0x0172		dmp_FragmentMaterial.diffuse、 dmp_FragmentLightSource[3].diffuse	FSUNIFORM
0x0173		dmp_FragmentMaterial.ambient、 dmp_FragmentLightSource[3].ambient	FSUNIFORM
0x0174		dmp_FragmentLightSource[3].position	FSUNIFORM
0x0175	8.8.5.3		
0x0176		dmp_FragmentLightSource[3].spotDirection	FSUNIFORM
0x0177			
0x0179		dmp_FragmentLightSource[3].position、 dmp_FragmentLightSource[3].twoSideDiffuse、 dmp_FragmentLightSource[3].geomFactor0、 dmp_FragmentLightSource[3].geomFactor1	FSUNIFORM
0x017A		dmp_FragmentLightSource[3].distanceAttenuationBias	FSUNIFORM
0x017B		dmp_FragmentLightSource[3].distanceAttenuationScale	FSUNIFORM
0x0180		dmp_FragmentMaterial.specular0、 dmp_FragmentLightSource[4].specular0	FSUNIFORM
0x0181		dmp_LightEnv.lutEnabledRefl、 dmp_FragmentMaterial.specular1、 dmp_FragmentLightSource[4].specular1	FSUNIFORM
0x0182		dmp_FragmentMaterial.diffuse、 dmp_FragmentLightSource[4].diffuse	FSUNIFORM
0x0183		dmp_FragmentMaterial.ambient、 dmp_FragmentLightSource[4].ambient	FSUNIFORM
0x0184		dmp_FragmentLightSource[4].position	FSUNIFORM
0x0185	8.8.5.3		
0x0186		dmp_FragmentLightSource[4].spotDirection	FSUNIFORM
0x0187			
0x0189		dmp_FragmentLightSource[4].position、 dmp_FragmentLightSource[4].twoSideDiffuse、 dmp_FragmentLightSource[4].geomFactor0、 dmp_FragmentLightSource[4].geomFactor1	FSUNIFORM
0x018A		dmp_FragmentLightSource[4].distanceAttenuationBias	FSUNIFORM
0x018B		dmp_FragmentLightSource[4].distanceAttenuationScale	FSUNIFORM
0x0190	8.8.5.3	dmp_FragmentMaterial.specular0、 dmp_FragmentLightSource[5].specular0	FSUNIFORM

0x0191		dmp_LightEnv.lutEnabledRefl、 dmp_FragmentMaterial.specular1、 dmp_FragmentLightSource[5].specular1	FSUNIFORM
0x0192		dmp_FragmentMaterial.diffuse、 dmp_FragmentLightSource[5].diffuse	FSUNIFORM
0x0193		dmp_FragmentMaterial.ambient、 dmp_FragmentLightSource[5].ambient	FSUNIFORM
0x0194		dmp_FragmentLightSource[5].position	FSUNIFORM
0x0195			
0x0196		dmp_FragmentLightSource[5].spotDirection	FSUNIFORM
0x0197			
0x0199		dmp_FragmentLightSource[5].position、 dmp_FragmentLightSource[5].twoSideDiffuse、 dmp_FragmentLightSource[5].geomFactor0、 dmp_FragmentLightSource[5].geomFactor1	FSUNIFORM
0x019A		dmp_FragmentLightSource[5].distanceAttenuationBias	FSUNIFORM
0x019B		dmp_FragmentLightSource[5].distanceAttenuationScale	FSUNIFORM
0x01A0	8.8.5.3	dmp_FragmentMaterial.specular0、 dmp_FragmentLightSource[6].specular0	FSUNIFORM
0x01A1		dmp_LightEnv.lutEnabledRefl、 dmp_FragmentMaterial.specular1、 dmp_FragmentLightSource[6].specular1	FSUNIFORM
0x01A2		dmp_FragmentMaterial.diffuse、 dmp_FragmentLightSource[6].diffuse	FSUNIFORM
0x01A3		dmp_FragmentMaterial.ambient、 dmp_FragmentLightSource[6].ambient	FSUNIFORM
0x01A4		dmp_FragmentLightSource[6].position	FSUNIFORM
0x01A5			
0x01A6		dmp_FragmentLightSource[6].spotDirection	FSUNIFORM
0x01A7			
0x01A9		dmp_FragmentLightSource[6].position、 dmp_FragmentLightSource[6].twoSideDiffuse、 dmp_FragmentLightSource[6].geomFactor0、 dmp_FragmentLightSource[6].geomFactor1	FSUNIFORM
0x01AA		dmp_FragmentLightSource[6].distanceAttenuationBias	FSUNIFORM
0x01AB		dmp_FragmentLightSource[6].distanceAttenuationScale	FSUNIFORM
0x01B0	8.8.5.3	dmp_FragmentMaterial.specular0、 dmp_FragmentLightSource[7].specular0	FSUNIFORM
0x01B1		dmp_LightEnv.lutEnabledRefl、 dmp_FragmentMaterial.specular1、 dmp_FragmentLightSource[7].specular1	FSUNIFORM
0x01B2		dmp_FragmentMaterial.diffuse、 dmp_FragmentLightSource[7].diffuse	FSUNIFORM

0x01B3		dmp_FragmentMaterial.ambient、 dmp_FragmentLightSource[7].ambient	FSUNIFORM
0x01B4		dmp_FragmentLightSource[7].position	FSUNIFORM
0x01B5			
0x01B6		dmp_FragmentLightSource[7].spotDirection	FSUNIFORM
0x01B7			
0x01B9		dmp_FragmentLightSource[7].position、 dmp_FragmentLightSource[7].twoSideDiffuse、 dmp_FragmentLightSource[7].geomFactor0、 dmp_FragmentLightSource[7].geomFactor1	FSUNIFORM
0x01BA		dmp_FragmentLightSource[7].distanceAttenuationBias	FSUNIFORM
0x01BB		dmp_FragmentLightSource[7].distanceAttenuationScale	FSUNIFORM
0x01C0	8.8.5.2	dmp_FragmentLighting.ambient、 dmp_FragmentMaterial.ambient、 dmp_FragmentMaterial.emission	FSUNIFORM
0x01C2	8.8.5.1	dmp_FragmentLightSource[i].enabled(i=0～7)	FSUNIFORM
0x01C3	8.8.5.8	dmp_LightEnv.invertShadow、 dmp_LightEnv.shadowAlpha、 dmp_LightEnv.shadowPrimary、 dmp_LightEnv.shadowSecondary、 dmp_LightEnv.shadowSelector	FSUNIFORM
	8.8.5.9	dmp_LightEnv.bumpMode、dmp_LightEnv.bumpRenorm、 dmp_LightEnv.bumpSelector、 dmp_LightEnv.clampHighlights、 dmp_LightEnv.config、dmp_LightEnv.fresnelSelector	FSUNIFORM
0x01C4	8.8.5.3	dmp_FragmentLightSource[i].distanceAttenuationEnabled(i=0～7)、 dmp_FragmentLightSource[i].shadowed、 dmp_FragmentLightSource[i].spotEnabled	FSUNIFORM
	8.8.5.9	dmp_LightEnv.fresnelSelector、 dmp_LightEnv.lutEnabledD0、 dmp_LightEnv.lutEnabledD1、 dmp_LightEnv.lutEnabledRefl	FSUNIFORM
0x01C5	8.8.5.4	dmp_FragmentMaterial.sampler {D0,D1,FR,RB,RG,RR} dmp_FragmentLightSource[i].sampler{SP,DA}	LUT
0x01C6	8.8.5.1	dmp_FragmentLighting.enabled	FSUNIFORM
0x01C8～0x01CF	8.8.5.4	参照テーブルへのデータ設定	LUT
0x01D0	8.8.5.5	dmp_LightEnv.absLutInputD0、 dmp_LightEnv.absLutInputD1、 dmp_LightEnv.absLutInputSP、 dmp_LightEnv.absLutInputFR、 dmp_LightEnv.absLutInputRB、 dmp_LightEnv.absLutInputRG、 dmp_LightEnv.absLutInputRR	FSUNIFORM
0x01D1	8.8.5.6	dmp_LightEnv.lutInputD0、 dmp_LightEnv.lutInputD1、 dmp_LightEnv.lutInputSP、 dmp_LightEnv.lutInputFR、 dmp_LightEnv.lutInputRB、 dmp_LightEnv.lutInputRG、dmp_LightEnv.lutInputRR	FSUNIFORM

0x01D2	8.8.5.7	dmp_LightEnv.lutScaleD0、 dmp_LightEnv.lutScaleD1、 dmp_LightEnv.lutScaleSP、 dmp_LightEnv.lutScaleFR、 dmp_LightEnv.lutScaleRB、 dmp_LightEnv.lutScaleRG、dmp_LightEnv.lutScaleRR	FSUNIFORM
0x01D9	8.8.5.1	dmp_FragmentLightSource[i].enabled(i=0～7)	FSUNIFORM

8.10.3. レジスタ 0x0200 ～ 0x02FF のレジスタマップ

表 8-73. レジスタマップ(アドレス 0x0200 ～ 0x02FF)

アドレス	参照先	関数・予約ユニフォーム	NN_GX_STATE_
0x0200	8.8.1.9	glBufferData()	VERTEX
0x0201		glVertexAttribPointer()	VERTEX
0x0202		glEnableVertexAttribArray()、 glDisableVertexAttribArray()、 glVertexAttribPointer()	VERTEX
0x0203～ 0x0226		glBufferData()、glVertexAttribPointer()	VERTEX
0x0227	8.8.1.9	glBufferData()	–
	8.8.19	glDrawElements()	–
0x0228	8.8.19	glDrawElements()、glDrawArrays()	–
0x0229		glDrawElements()	SHADERMODE
0x022A		glDrawArrays()	–
0x022E		glDrawArrays()	–
0x022F		glDrawElements()	–
0x0231		glDrawElements()、glDrawArrays()	–
0x0232	8.8.1.8	glVertexAttribPointer()、 glVertexAttrib{1234}f()、glVertexAttrib{1234}fv()	VERTEX
0x0233～ 0x0235		頂点属性データの設定	VERTEX
0x0238～ 0x023D	8.8.23	コマンドバッファの実行 nngxAddJumpCommand()、nngxAddSubroutineCommand()	–
0x0242	8.8.1.6	頂点属性の入力数設定	SHADERPROGRAM
0x0244	8.8.20	共用プロセッサ設定	SHADERMODE
0x0245	8.8.19	glDrawElements()、glDrawArrays()	–
0x024A	8.8.1.10	出力レジスタの使用数設定	SHADERPROGRAM
0x0251	8.8.1.10	出力レジスタの使用数設定	SHADERPROGRAM
0x0252	8.8.20.12	サブディビジョンシェーダの使用設定	SHADERPROGRAM
0x0253	8.8.19	glDrawElements()、glDrawArrays()	–
0x0254	8.8.20.12	サブディビジョンシェーダの使用設定	SHADERPROGRAM

0x025E	8.8.1.10	出力レジスタの使用数設定	SHADERPROGRAM
	8.8.20.8	出力レジスタの使用数設定(ジオメトリシェーダ)	SHADERPROGRAM
	8.8.19	glDrawElements()、glDrawArrays()	–
0x025F	8.8.19	glDrawElements()、glDrawArrays()	–
0x0280	8.8.20.2	ブールレジスタ(ジオメトリシェーダ)	VSUNIFORM SHADERMODE
0x0281～ 0x0284	8.8.20.3	整数レジスタ(ジオメトリシェーダ)	VSUNIFORM SHADERMODE
0x0289	8.8.20.6	頂点入力数設定レジスタ(ジオメトリシェーダ)	SHADERPROGRAM
	8.8.20.12	ジオメトリシェーダ使用の設定	SHADERMODE
0x028A	8.8.20.5	開始アドレス設定レジスタ(ジオメトリシェーダ)	SHADERPROGRAM SHADERMODE
0x028B	8.8.20.7	入力レジスタのマッピング設定レジスタ(ジオメトリシェーダ)	VERTEX
0x028C			
0x028D	8.8.20.9	出力レジスタのマスク設定レジスタ(ジオメトリシェーダ)	SHADERPROGRAM SHADERMODE
0x028F	8.8.20.4	プログラムコード設定レジスタ(ジオメトリシェーダ)	SHADERBINARY
0x0290	8.8.20.1	浮動小数点定数レジスタ(ジオメトリシェーダ)	SHADERFLOAT VSUNIFORM
0x0291～ 0x0298		浮動小数点定数のロード(ジオメトリシェーダ)	
0x029B	8.8.20.4	プログラムコードのロードアドレス(ジオメトリシェーダ)	SHADERBINARY
0x029C ～0x02A 3		プログラムコードのロード(ジオメトリシェーダ)	
0x02A5	8.8.20.4	Swizzle パターンのロードアドレス(ジオメトリシェーダ)	SHADERBINARY
0x02A6 ～0x02A D		Swizzle パターンのロード(ジオメトリシェーダ)	
0x02B0	8.8.1.2	ブールレジスタ	VSUNIFORM SHADERMODE
0x02B1 ～0x02B 4	8.8.1.3	整数レジスタ	VSUNIFORM SHADERMODE
0x02B9	8.8.1.6	頂点入力数設定レジスタ	SHADERPROGRAM SHADERMODE
0x02BA	8.8.1.5	開始アドレス設定レジスタ	SHADERPROGRAM SHADERMODE
0x02BB	8.8.1.7	入力レジスタのマッピング設定レジスタ	VERTEX
0x02BC			
0x02BD	8.8.1.11	出力レジスタのマスク設定レジスタ	SHADERPROGRAM SHADERMODE
0x02BF	8.8.1.4	プログラムコード設定レジスタ	SHADERBINARY

0x02C0	8.8.1.1	浮動小数点定数レジスタ	SHADERFLOAT VSUNIFORM
0x02C1 ～0x02C8		浮動小数点定数のロード	
0x02CB	8.8.1.4	プログラムコードのロードアドレス	SHADERBINARY
0x02CC ～0x02D3		プログラムコードのロード	
0x02D5	8.8.1.4	Swizzle パターンのロードアドレス	SHADERBINARY
0x02D6 ～0x02DD		Swizzle パターンのロード	

9. プロファイル機能

3DS の GPU にはプロファイル機能が搭載されており、ハードウェア性能のベンチマークやパフォーマンスのチューニングに利用することができます。

プロファイル機能には以下のものがあります。

表 9-1. プロファイル機能一覧

機能	説明
ビジーカウンタ	一定期間内に各モジュールのビジー信号が出力された回数を比較し、最も多くビジーが出力されたモジュールを判定します。 指定回数の計測を繰り返し、モジュールごとに、最も多くビジーが出力されたと判定された回数をカウントします。
シェーダ実行クロック数カウンタ	4 基あるシェーダプロセッサのプログラムカウンタの遷移数やストールしたクロック数をカウントします。
頂点キャッシュ入力頂点数カウンタ	ポスト頂点キャッシュに入力された頂点数をカウントします。
入出力ポリゴン数カウンタ	トライアングルセットアップのモジュールに入力された頂点とポリゴンの数と出力されたポリゴンの数をカウントします。
入力フラグメント数カウンタ	パーフラグメントオペレーションのモジュールに入力されたフラグメントの数をカウントします。
メモリアクセス数カウンタ	VRAM や頂点バッファなどのメモリにアクセスした回数をカウントします。

注意： プロファイル機能で呼び出す関数はコマンドリストの実行中に呼び出さないでください。呼び出してもエラーにはなりませんが、GPU が不正な動作をする可能性があります。

9.1. プロファイル機能の開始と停止

プロファイル機能の中には、以下の関数で開始と停止をアプリケーションで明示的に指示しなければならないものがあります。

コード 9-1. プロファイル機能の開始と停止

```
void nngxStartProfiling(GLenum item);
void nngxStopProfiling(GLenum item);
```

item には、開始または停止を指示するプロファイル機能を以下の定義値から指定します。下記以外の定義値を指定した場合、nngxStartProfiling() は GL_ERROR_80A2_DMP のエラーを、nngxStopProfiling() は GL_ERROR_80A3_DMP のエラーをそれぞれ生成します。

表 9-2. 開始と停止で指定する定義値と対応するプロファイル機能

定義値	プロファイル機能
NN_GX_PROFILING_BUSY	ビジーカウンタ（必ずパラメータ設定で計測期間を指定してください）
NN_GX_PROFILING_VERTEX_CACHE	頂点キャッシュ入力頂点数カウンタ（使用時は消費電力が上がる可能性があります。使用しない場合は必ず停止してください）

9.2. プロファイル機能のパラメータ

以下の関数で、一部のプロファイル機能のパラメータを設定することができます。

コード 9-2. プロファイル機能のパラメータを設定する関数

```
void nngxSetProfilingParameter(GLenum pname, GLuint param);
```

pname に指定する定義値によって、*param* に指定する値が以下のように異なります。

表 9-3. プロファイル機能のパラメーター覧

pname に指定する定義値	param に指定する値
NN_GX_PROFILING_BUSY_SAMPLING_TIME	ビジーカウンタの 1 回あたりの計測時間(GPU クロック数)を 0 以外の 16 ビット値で指定します。
NN_GX_PROFILING_BUSY_SAMPLING_TIME_MICRO_SECOND	NN_GX_PROFILING_BUSY_SAMPLING_TIME の設定をマイクロ秒単位で指定します。 GPU クロック数に変換したときに 16 ビット値に納まっていなければならないため、指定可能な値の範囲は 1～244 です。
NN_GX_PROFILING_BUSY_SAMPLING_TIME_NANO_SECOND	NN_GX_PROFILING_BUSY_SAMPLING_TIME の設定をナノ秒単位で指定します。 GPU クロック数に変換したときに 16 ビット値に納まっていなければならないため、指定可能な値の範囲は 1～244537 です。
NN_GX_PROFILING_BUSY_SAMPLING_COUNT	ビジーカウンタの計測回数を 16 ビット値で指定します。 0 を指定した場合、ビジーカウンタは明示的に停止を指示されるまで動作を継続し(1 回あたり NN_GX_PROFILING_BUSY_SAMPLING_TIME で指定された計測時間で動作し、停止されるまで繰り返し)ます。 0 以外が指定された場合、ビジーカウンタは明示的に停止を指示されるか、開始から(計測時間×計測回数)クロック経った時点で停止します。 1 以上の値を設定し、かつ計測終了を待ってから結果を取得した場合、ビジーカウンタの計測結果における各モジュールの値の合計は計測回数に一致します。

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

エラー	原因
GL_ERROR_80A5_DMP	*pname* に不正な値を指定した
GL_ERROR_80A6_DMP	*param* に不正な値を指定した

補足: 計測時間から GPU クロック数への変換は 268 MHz(268,000,000 Hz)で行われます。

注意: パラメータの初期値は不定のため、必ず関連するパラメータすべての値を設定してください。

9.3. プロファイル結果の取得

以下の関数で、プロファイル結果を取得することができます。

コード 9-3. プロファイル結果を取得する関数

```
void nngxGetProfilingResult(GLenum item, GLuint* result);
```

item には、結果を取得するプロファイル機能を指定します。不正な値を指定した場合、GL_ERROR_80A4_DMP のエラーが生成されます。

result には、指定されたプロファイル機能の結果が格納されます。プロファイル機能によって、結果を格納するために必要なバッファのサイズが異なることに注意してください。

各プロファイル機能の結果を取得する際の *item* の指定や *result* に格納される結果の詳細については、以降の項で説明します。

> **補足:** 明示的に開始と停止を指示しなければならないプロファイル機能以外は常時動作しています。これらのプロファイル機能のカウンタはハードウェアの起動時にリセットされますので、プロファイル結果を取得する場合は、計測期間の開始時と終了時で取得した結果の差分値を利用してください。
>
> プロファイル機能のカウンタはオーバーフローすると 0 に戻ってカウントを続けます。

9.3.1. ビジーカウンタ

ビジーカウンタの結果を取得するには、*item* に NN_GX_PROFILING_BUSY を指定します。*result* には、要素数が NN_GX_PROFILING_RESULT_BUFSIZE_BUSY の GLuint 型の配列を指定してください。

ビジーカウンタは nngxStartProfiling() で開始されたときにカウンタを 0 にリセットし、停止されるまで一定期間ごとにビジーが出力された回数を比較し、最も多くビジーが出力された回数をカウントします。また、計測結果を取得するごとにカウンタは 0 にリセットされます。そのため、プロファイル機能を開始後、停止させずに結果を取得した場合、前回結果を取得したときからの計測結果を取得することになります。

各モジュールの結果は 16 ビット値で格納されています。下表にデータの格納順とモジュールの対応を示します。

表 9-5. ビジーカウンタのデータの格納順

データ	モジュール
result[0] のビット [31 : 16]	シェーダプロセッサ 0(ジオメトリプロセッサと共用)
result[0] のビット [15 : 0]	コマンドバッファおよび頂点アレイロードモジュール
result[1] のビット [31 : 16]	ラスタライゼーションモジュール
result[1] のビット [15 : 0]	トライアングルセットアップ
result[2] のビット [31 : 16]	フラグメントライティング
result[2] のビット [15 : 0]	テクスチャユニット
result[3] のビット [31 : 16]	パーフラグメントオペレーションモジュール
result[3] のビット [15 : 0]	テクスチャコンバイナ

ビジーカウンタの結果は、ビジーが最も多く出力された回数であり、ビジー信号が出力された回数そのものではありません。

ビジーカウンタの計測を開始すると、1 回あたりの計測時間内での各モジュールのビジー信号の出力回数を比較し、最もビジー信号の出力回数が多かったモジュールのカウンタを 1 カウントアップします。この計測は指定された計測回数分繰り返し行われます。なお、1 回あたりの計測時間内に複数のモジュールでビジー信号の出力回数が同じ値で最大となった場合、それらの中で最前段にあるモジュールのカウンタがカウントアップされます。また、すべてのモジュールのビジー信号出力回数が 0 だった場合は、最前段にある「コマンドバッファおよび頂点アレイロードモジュール」のカウンタがカウントアップされます。そのため、計測回数を 1 回以上に設定し、かつ計測が終了してから結果を取得した場合、各モジュールの計測結果の値の合計は、計測回数に一致します。

計測結果のうちで最も値が大きいモジュールが、最も長い期間ボトルネックになったと予測することができます。そのモジュールを中心に最適化を行い、パフォーマンスのチューニングを行ってください。

9.3.1.1. 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のボトルネック解析

ビジーカウンタの結果から、トライアングルセットアップ（以降 TS）およびラスタライゼーションモジュール（以降 RAS）がボトルネックになっている可能性について解析できます。

TS はポリゴン数、RAS はポリゴン数および生成されるピクセル数の影響を受けます。そのため、以下のような特性があります。

- 処理するポリゴン数、ポリゴンの座標、生成されるピクセル数が変わらない限り、頂点データをロードして生成されたポリゴンか、ジオメトリシェーダで生成されたポリゴンかは TS、RAS の性能に影響しません。
- 処理するポリゴン数、ポリゴンの座標、生成されるピクセル数が変わらない限り、ドロー関数が何回呼び出されて描画されているかは TS、RAS の性能に影響しません。
- 頂点の属性数（テクスチャ座標の個数など）の増減は、TS、RAS の性能に影響しません。

TS、RAS がボトルネックと予想される場合、頂点シェーダに nop を数～数十個挿入して性能が変わるかどうか、および画面全体にシザリングを有効にした状態で性能が変わるかどうかの 2 つのケースを確認し、どちらも性能が変わらない場合、TS、RAS がボトルネックである可能性が高まります。TS の処理負荷を下げるには、ポリゴン数を減らす必要があります。方法としては、描画オブジェクトの LOD を使用する、CPU 側でオブジェクトのクリップを行う、などが挙げられます。TS の負荷が下がれば、RAS の負荷も下がります。

9.3.2. シェーダ実行クロック数カウンタ

シェーダ実行クロック数カウンタの結果を取得するには、*item* に取得するシェーダプロセッサに対応する定義値 NN_GX_PROFILING_VERTEX_SHADER*n*（*n* は 0～3 のプロセッサ番号）を指定します。*result* には、要素数が NN_GX_PROFILING_RESULT_BUFSIZE_VERTEX_SHADER*n*（*n* は 0～3 のプロセッサ番号）の GLuint 型の配列を指定してください。なお、ジオメトリプロセッサはシェーダプロセッサ 0 と共用です。

シェーダ実行クロック数カウンタはハードウェアの起動時に 0 にリセットされ、常に動作しています。

結果は 32 ビット値で格納されています。下表にデータの格納順と情報の対応を示します。

表 9-6. シェーダ実行クロック数カウンタのデータの格納順

データ	情報
result[0]	プログラムカウンタの遷移数（実行されたシェーダアセンブラの命令数と同じです）
result[1]	シェーダアセンブラ命令の依存関係によりストールしたクロック数 （NN_GX_PROFILING_VERTEX_SHADER0 のみ、ジオメトリシェーダが有効なときに後段のモジュールがビジーとなって vout 命令の発行が待たされた場合を含みます）
result[2]	アドレスレジスタの更新（mova 命令発行）によりストールしたクロック数
result[3]	ステータスレジスタの更新（cmp 命令発行）によりストールしたクロック数

result[4]	プログラムのプリフェッチのミスヒットによりストールしたクロック数(分岐命令などによるプログラムカウンタの不連続遷移により発生します)

9.3.3. 頂点キャッシュ入力頂点数カウンタ

頂点キャッシュ入力頂点数カウンタの結果を取得するには、*item* に NN_GX_PROFILING_VERTEX_CACHE を指定します。*result* には、要素数が NN_GX_PROFILING_RESULT_BUFSIZE_VERTEX_CACHE の GLuint 型の配列を指定してください。

頂点キャッシュ入力頂点数カウンタは、頂点バッファを使用する(レジスタ 0x022F に 1 を書き込む)描画が開始されたときにカウンタが 0 にリセットされます。そのため、取得できるのは最後に実行された描画での計測結果です。描画に使用した頂点インデックスの総数と比較することで、ポスト頂点キャッシュの効率が推測できます。ただし、同じ頂点データで描画を行った場合でも、頂点インデックスのロードのタイミングが異なったり、後段のモジュールがビジーとなったりすることによって、カウントが多少上下することがあります。また、nngxStartProfiling() で計測を開始していなくても、動作しているようなプロファイル結果が取得されることがありますが、その値は正しくない可能性があります。正しい結果を取得するためには、必ず nngxStartProfiling() を呼び出して計測を開始してください。

結果は 32 ビット値で格納されています。下表にデータの格納順と情報の対応を示します。

表 9-7. 頂点キャッシュ入力頂点数カウンタのデータの格納順

データ	情報
result[0]	ポスト頂点キャッシュに入力された頂点数

9.3.4. 入出力ポリゴン数カウンタ

入出力ポリゴン数カウンタの結果を取得するには、*item* に NN_GX_PROFILING_POLYGON を指定します。*result* には、要素数が NN_GX_PROFILING_RESULT_BUFSIZE_POLYGON の GLuint 型の配列を指定してください。

入出力ポリゴン数カウンタはハードウェアの起動時に 0 にリセットされ、常に動作しています。

結果は 32 ビット値で格納されています。下表にデータの格納順と情報の対応を示します。

表 9-8. 入出力ポリゴン数カウンタのデータの格納順

データ	情報
result[0]	トライアングルセットアップへの入力頂点数
result[1]	トライアングルセットアップへの入力ポリゴン数
result[2]	トライアングルセットアップからの出力ポリゴン数

出力ポリゴン数は、入力されたポリゴン数からクリッピングやカリングにより省かれたポリゴンを差し引いた値です。クリップボリュームに交わるポリゴンは、実際には複数のポリゴンに分割されて出力されますが、1 とカウントされます。

9.3.5. 入力フラグメント数カウンタ

入力フラグメント数カウンタの結果を取得するには、*item* に NN_GX_PROFILING_FRAGMENT を指定します。*result* には、要素数が NN_GX_PROFILING_RESULT_BUFSIZE_FRAGMENT の GLuint 型の配列を指定してください。

入力フラグメント数カウンタはハードウェアの起動時に 0 にリセットされ、常に動作しています。

結果は 32 ビット値で格納されています。下表にデータの格納順と情報の対応を示します。

表 9-9. 入力フラグメント数カウンタのデータの格納順

データ	情報
result[0]	パーフラグメントオペレーションモジュールに入力されたフラグメント数

フラグメント数には、クリッピング、シザーテスト、アーリーデプステストにより破棄されたフラグメントは含まれません。アルファテスト、ステンシルテスト、デプステストについてはテスト前のフラグメント数をカウントするため、テスト結果はカウントに影響しません。

9.3.6. メモリアクセス数カウンタ

メモリアクセス数カウンタの結果を取得するには、*item* に NN_GX_PROFILING_MEMORY_ACCESS を指定します。*result* には、要素数が NN_GX_PROFILING_RESULT_BUFSIZE_MEMORY_ACCESS の GLuint 型の配列を指定してください。

メモリアクセス数カウンタはハードウェアの起動時に 0 にリセットされ、常に動作しています。

結果は 32 ビット値で格納されています。下表にデータの格納順と情報の対応を示します。

表 9-10. メモリアクセス数カウンタのデータの格納順

データ	情報
result[0]	GPU によるVRAM のリード(A チャンネル)
result[1]	GPU によるVRAM のライト(A チャンネル)
result[2]	GPU によるVRAM のリード(B チャンネル)
result[3]	GPU によるVRAM のライト(B チャンネル)
result[4]	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によるコマンドバッファ、頂点アレイ、インデックスアレイのリード
result[5]	テクスチャユニットによるテクスチャメモリのリード
result[6]	パーフラグメントオペレーションモジュールによるデプスバッファおよびステンシルバッファのリード
result[7]	パーフラグメントオペレーションモジュールによるデプスバッファおよびステンシルバッファのライト
result[8]	パーフラグメントオペレーションモジュールによるカラーバッファのリード
result[9]	パーフラグメントオペレーションモジュールによるカラーバッファのライト
result[10]	上画面 LCD コントローラによるディスプレイバッファのリード
result[11]	下画面 LCD コントローラによるディスプレイバッファのリード
result[12]	ポスト転送モジュールによるリード(nngxTransferRenderImage()、glCopyTexImage2D() などによる転送)
result[13]	ポスト転送モジュールによるライト(nngxTransferRenderImage()、glCopyTexImage2D() などによる転送)
result[14]	メモリフィルモジュールのチャネル 0 によるバッファのライト(glClear() などによるバッファクリア)

result[15]	メモリフィルモジュールのチャネル 1 によるバッファのライト(`glClear()` などによるバッファクリア)
result[16]	CPU による VRAM のリード(`glReadPixels()` など)
result[17]	CPU(DMA 転送)による VRAM のライト(`nngxAddVramDmaCommand()` などによる DMA 転送)

10. コマンドリストを直接指定する関数

コマンドリストをバインドせずに直接指定する関数も用意されています。オブジェクトやバッファの管理などをアプリケーションで行わなければなりませんが、ライブラリの内部状態に依存しないため、より自由度の高いフレームワークを構築することができます。

ここでは、コマンドリストを直接指定する関数を「gx Raw API」、従来の関数を「gx API」と表記します。

一部の関数を除いて、対応する gx API の末尾に「Raw」が付加され、引数の指定に nngxCommandList 構造体へのポインタ指定が追加されます。基本的に、関数の動作や生成されるエラーに違いはありません。

gx Raw API を使用する場合は、以下の点に注意してください。

- カレントのコマンドリストに依存する、gl 関数や GD ライブラリの関数とは併用できません。
- nngxCommandList 構造体をコピーして使用することはできません。
- コマンドリストを使用する gx API と gx Raw API を混成使用した場合の動作は保証されません。
- コマンドリストの保存、再使用、コピー、ジャンプコマンドの追加、サブルーチンコマンドの追加、エクスポート、インポートには対応する関数が用意されていないため、アプリケーションで実装する必要があります。

表 10-1. コマンドリストを使用する gx API に対応する gx Raw API

gx API	gx Raw API
nngxGenCmdlists()	対応する関数はありません
nngxBindCmdlist()	対応する関数はありません
nngxCmdlistStorage()	nngxCmdlistStorageRaw() nngxGetCommandRequestSizeRaw()
nngxDeleteCmdlists()	対応する関数はありません
nngxSplitDrawCmdlist()	nngxSplitDrawCmdlistRaw()
nngxFlush3DCommand()	対応する関数はありません
nngxFlush3DCommandNoCacheFlush()	nngxFlush3DCommandNoCacheFlushRaw()
nngxFlush3DCommandPartially()	nngxFlush3DCommandPartiallyRaw()
nngxAdd3DCommand()	対応する関数はありません
nngxAdd3DCommandNoCacheFlush()	nngxAdd3DCommandNoCacheFlushRaw()
nngxAddJumpCommand()	対応する関数はありません
nngxAddSubroutineCommand()	対応する関数はありません
nngxMoveCommandbufferPointer()	nngxMoveCommandbufferPointerRaw()
nngxAddB2LTransferCommand()	nngxAddB2LTransferCommandRaw()
nngxAddBlockImageCopyCommand()	nngxAddBlockImageCopyCommandRaw()
nngxAddL2BTransferCommand()	nngxAddL2BTransferCommandRaw()
nngxAddMemoryFillCommand()	nngxAddMemoryFillCommandRaw()
nngxAddVramDmaCommand()	対応する関数はありません
nngxAddVramDmaCommandNoCacheFlush()	nngxAddVramDmaCommandNoCacheFlushRaw()

nngxFilterBlockImage()	nngxFilterBlockImageRaw()
nngxEnableCmdlistCallback()	nngxEnableCmdlistCallbackRaw()
nngxDisableCmdlistCallback()	nngxDisableCmdlistCallbackRaw()
nngxSetCmdlistCallback()	nngxSetCmdlistCallbackRaw()
nngxRunCmdlist() nngxRunCmdlistById()	nngxRunCmdlistRaw()
nngxStopCmdlist()	同じ関数を使用します
nngxReserveStopCmdlist()	nngxReserveStopCmdlistRaw()
nngxWaitCmdlistDone()	同じ関数を使用します
nngxSetTimeout()	同じ関数を使用します
nngxGetIsRunning()	同じ関数を使用します
nngxClearCmdlist()	nngxClearCmdlistRaw()
nngxClearFillCmdlist()	nngxClearFillCmdlistRaw()
nngxStartCmdlistSave()	対応する関数はありません
nngxStopCmdlistSave()	対応する関数はありません
nngxUseSavedCmdlist()	対応する関数はありません
nngxUseSavedCmdlistNoCacheFlush()	対応する関数はありません
nngxAddCmdlist()	対応する関数はありません
nngxCopyCmdlist()	対応する関数はありません
nngxExportCmdlist()	対応する関数はありません
nngxImportCmdlist()	対応する関数はありません
nngxSetGasAutoAccumulationUpdate()	nngxSetGasAutoAccumulationUpdateRaw()
nngxSetCmdlistParameteri()	nngxSetGasUpdateRaw()
nngxGetCmdlistParameteri()	nngxGetIsRunningRaw() nngxGetUsedBufferSizeRaw() nngxGetUsedRequestCountRaw() nngxGetMaxBufferSizeRaw() nngxGetMaxRequestCountRaw() nngxGetTopBufferAddrRaw() nngxGetRunBufferSizeRaw() nngxGetRunRequestCountRaw() nngxGetTopRequestAddrRaw() nngxGetNextRequestTypeRaw() nngxGetNextBufferAddrRaw() nngxGetNextBufferSizeRaw() nngxGetHWStateRaw() nngxGetCurrentBufferAddrRaw()

gx API と gx Raw API とで使用方法が異なる関数については、以降の節で説明します。

10.1. コマンドリストの確保と破棄

gx Raw API では、コマンドリストで使用するデータ領域を確保するまでの手順や、コマンドリストを破棄する方法が従来とは異なります。

gx API では、以下の手順でコマンドリストの確保を行っていました。

1. コマンドリストオブジェクトを生成する(nngxGenCmdlist())
2. コマンドリストオブジェクトをバインドする(nngxBindCmdlist())
3. データ領域(3D コマンドバッファ、コマンドリクエストのバッファ)を確保する(nngxCmdlistStorage())

gx Raw API では、以下の手順でコマンドリストの確保を行います。

1. コマンドリクエストのキューに必要な領域のサイズを取得する(nngxGetCommandRequestSizeRaw())
2. データ領域(3D コマンドバッファ、コマンドリクエストのバッファ)を確保する
3. コマンドリストオブジェクトを生成する(nngxCmdlistStorageRaw())

コマンドリストの確保に使用する関数は、以下の通りです。

コード 10-1. コマンドリストの確保に使用する関数(gx Raw API)

```
GLsizei nngxGetCommandRequestSizeRaw(GLsizei requestcount);
void nngxCmdlistStorageRaw(nngxCommandList* cmdlist,
                           GLsizei bufsize, GLvoid* commandbuffer,
                           GLsizei requestcount, GLvoid* commandrequest);
```

requestcount には、コマンドリストにキューイング可能なコマンドリクエストの数を指定します。

3D コマンドバッファを確保するときは、そのアドレスとサイズ(*commandbuffer* と *bufsize*)のアライメントが 16 バイトでなければならないことに注意してください。また、3D コマンドバッファはデバイスメモリ上に確保する必要があります。

nngxCommandList 構造体自身(*cmdlist*)とコマンドリクエストのバッファ(*commandrequest*)は、アドレスのアライメントが 4 バイトとなるように確保することを推奨します。バッファの確保先メモリに制約はありません。

コマンドリストを使用する間は、各バッファを解放しないように注意してください。

gx Raw API には、コマンドリストの破棄を行う nngxDeleteCmdlists() に対応する関数はありません。コマンドリストおよび 3D コマンドバッファ、コマンドリクエストのバッファの各データ領域は、アプリケーション側で管理しなければなりません。

10.2. パラメータの設定

カレントのコマンドリストのパラメータを設定する nngxSetCmdlistParameteri() では、設定するパラメータの種類を引数 *pname* で指定していました。gx Raw API では、パラメータの種類ごとに異なる関数を呼び出して設定します。

表 10-2. パラメータの設定で pname に指定する値と gx Raw API の対応

pname に指定する値	gx Raw API	パラメータの型
NN_GX_CMDLIST_RUN_MODE	対応する関数はありません	---
NN_GX_CMDLIST_GAS_UPDATE	nngxSetGasUpdateRaw()	GLboolean

10.3. パラメータの取得

カレントのコマンドリストのパラメータを取得する nngxGetCmdlistParameteri() では、取得するパラメータの種類を引数 *pname* で指定し、パラメータを引数で受け取っていました。gx Raw API では、パラメータの種類ごとに異なる関数を呼び出し、パラメータを返り値で取得します。

表 10-3. パラメータの取得で pname に指定する値と gx Raw API の対応

pname に指定する値	gx Raw API	返り値の型
NN_GX_CMDLIST_RUN_MODE	対応する関数はありません	---
NN_GX_CMDLIST_IS_RUNNING	nngxGetIsRunningRaw()	GLboolean
NN_GX_CMDLIST_USED_BUFSIZE	nngxGetUsedBufferSizeRaw()	GLsizei
NN_GX_CMDLIST_USED_REQCOUNT	nngxGetUsedRequestCountRaw()	GLsizei
NN_GX_CMDLIST_MAX_BUFSIZE	nngxGetMaxBufferSizeRaw()	GLsizei
NN_GX_CMDLIST_MAX_REQCOUNT	nngxGetMaxRequestCountRaw()	GLsizei
NN_GX_CMDLIST_TOP_BUFADDR	nngxGetTopBufferAddrRaw()	GLvoid*
NN_GX_CMDLIST_BINDING	対応する関数はありません	---
NN_GX_CMDLIST_RUN_BUFSIZE	nngxGetRunBufferSizeRaw()	GLsizei
NN_GX_CMDLIST_RUN_REQCOUNT	nngxGetRunRequestCountRaw()	GLsizei
NN_GX_CMDLIST_TOP_REQADDR	nngxGetTopRequestAddrRaw()	GLvoid*
NN_GX_CMDLIST_NEXT_REQTYPE	nngxGetNextRequestTypeRaw()	GLenum
NN_GX_CMDLIST_NEXT_REQINFO	nngxGetNextBufferAddrRaw() nngxGetNextBufferSizeRaw()	GLvoid* GLsizei
NN_GX_CMDLIST_HW_STATE	nngxGetHWStateRaw()	GLbitfield
NN_GX_CMDLIST_CURRENT_BUFADDR	nngxGetCurrentBufferAddrRaw()	GLvoid*

nngxGetHWStateRaw() のみ引数なし、そのほかの関数は nngxCommandList 構造体へのポインタを渡して呼び出します。

更新履歴

Version 1.3 2015-11-05

追加

- 9.3.1.1. 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のボトルネック解析

変更

- 2.1.4.1. 比較関数の変更について
  - 図「比較関数変更の制約を回避する例」を追加しました。
- 8.1. コマンドリストの保存
  - コマンドリストの保存の開始と終了により、コマンドリストの 3D コマンドバッファに、パディング用のダミーコマンドが生成される場合があることを追記しました。
  - 図「コマンドリストの保存」を追加しました。
- 8.1.2. コマンド生成
  - NN_GX_STATE_FRAMEBUFFER の依存関係を追加しました。
- 8.1.3. 差分コマンドと完全コマンド
  - 表「参照テーブルが有効となる条件」に「フラグメントライティングの距離減衰」を追加しました。
- 8.2. コマンドリストの再使用
  - アプリで3D コマンドバッファ領域がフラッシュを保証する必要性について追記しました。
  - 図「nngxUseSavedCmdlist() の引数 copycmd による動作の違い」を追加しました。
- 8.4. コマンドリストのエクスポート
  - 説明内容を修正しました。
- 8.7.2. 3D コマンドバッファの仕様
  - ダミーコマンドの説明を追加しました。
- 8.8.5.3. 光源設定レジスタ(0x0140 ～ 0x01BF, 0x01C4)
  - 図「そのほかの設定レジスタ(0x01C4、0x0149 ほか)のビットレイアウト」を追記しました。
- 8.8.5.5. 参照テーブルの引数範囲設定レジスタ(0x01D0)
  - 図「参照テーブルの引数範囲設定レジスタ(0x01D0)のビットレイアウト」を追加しました。
- 8.8.5.6. 参照テーブルの入力値設定レジスタ(0x01D1)
  - 図「参照テーブルの入力値設定レジスタ(0x01D1)のビットレイアウト」を追加しました。
- 8.8.5.7. 参照テーブルの出力値に対するスケール値設定レジスタ(0x01D2)
  - 図「参照テーブルの出力地に対するスケール値設定レジスタ(0x01D2)のビットレイアウト」を追加しました。
- 8.8.7.2. シェーディング参照テーブル設定レジスタ(0x0123, 0x0124)
  - ビットの指定範囲の間違いを修正しました。
- 8.8.9.2. シャドウ減衰ファクタ設定レジスタ(0x0130)
  - 図「シャドウ減衰ファクタ設定レジスタ(0x0130)のビットレイアウト」を追加しました。
- 8.8.9.3. w バッファ設定レジスタ(0x004D, 0x004E, 0x006D)
  - 図「wバッファ設定レジスタ(0x004D、0x004E、0x006D)のビットレイアウト」を追加しました。
- 8.8.9.6. フレームバッファアクセス制御設定レジスタ(0x0112 ～ 0x0115)
  - 図「フレームバッファアクセス制御設定レジスタ(0x0112 ～ 0x0115)のビットレイアウト」を追加しました。
- 8.8.20.1. 浮動小数点定数レジスタ(0x0290, 0x0291 ～ 0x0298)
  - 図「浮動小数点定数レジスタのインデックス指定(0x0290)のビットレイアウト」を追加しました。
- 8.8.20.2. ブールレジスタ(0x0280)
  - 図「ブールレジスタ(0x0280)のビットレイアウト」を追加しました。
- 8.8.20.3. 整数レジスタ(0x0281 ～ 0x0284)
  - 図「整数レジスタ(0x0281 ～ 0x0284)のビットレイアウト」を追加しました。
- 8.8.20.4. プログラムコード設定レジスタ(0x028F, 0x029B ～ 0x02A3, 0x02A5 ～ 0x02AD)
  - 図「プログラムコードのロードレジスタ(0x029B ～ 0x02A3)のビットレイアウト」を追加しました。
  - 図「Swizzle パターンのロードレジスタ(0x02A5 ～ 0x02AD)のビットレイアウト」を追加しました。

- 8.8.20.5. 開始アドレス設定レジスタ(0x028A)
  - 図「開始アドレス設定レジスタ(0x028A)のビットレイアウト」を追加しました。
- 8.8.20.7. 入力レジスタのマッピング設定レジスタ(0x028B, 0x028C)
  - 図「入力レジスタのマッピング設定レジスタ(0x028B, 0x028C)のビットレイアウト」を追加しました。
- 8.8.20.9. 出力レジスタのマスク設定レジスタ(0x028D)
  - 図「出力レジスタのマスク設定レジスタ(0x028D)のビットレイアウト」を追加しました。
- 8.8.23.2. 同じコマンドバッファの繰り返し実行
  - ジャンプ用コマンドの構成について図を追加し、詳細説明を追記しました。
- 9.2. プロファイル機能のパラメータ
  - nngxSetProfilingParameter() が生成するエラーについて表を追加しました。

Version 1.2 2015-04-28

変更

- 8.8.1.9. 頂点属性アレイ設定レジスタ(0x0200 ～ 0x0227)
  - ロードアレイのパディングについて情報を追加しました。
- 8.8.11. デプステスト設定レジスタ(0x0107, 0x0126)
  - デプステスト設定レジスタの情報を修正しました。
- 8.8.19.2. 頂点バッファを使用しない場合の設定
  - 描画に頂点バッファを使用しない場合の設定レジスタの説明を修正しました。

Version 1.1 2015-01-15

変更

- 6.2.3.4. 左右のカメラの行列計算
  - 行列計算を行う関数の引数説明を修正しました。
  - 行列計算時に 3D ボリューム値を更新する関数の説明を追記しました。

Version 1.0 2014-09-04

追加/変更

- 初版