3DS

3DS プログラミングマニュアル

GD ライブラリ編

2016-05-10

Version 1.1

目次

コード

表

図

1. はじめに

本ドキュメントは、CTR-SDK の GD ライブラリを用いたプログラミングについて説明したものです。グラフィックスの基本的な概念や一部ライブラリ(GX ライブラリなど)の使用方法については既に習得していることを前提としており、純粋に GD ライブラリに焦点を当てた内容となっています。

本書は「3DS プログラミングマニュアル」の「グラフィックス基本編」および「グラフィックス応用編」に記載されている内容を理解していることを前提に書かれています。本書を読む前に、これらのドキュメントをご一読ください。

1.1. GD ライブラリとは

GD ライブラリは CTR-SDK のグラフィックスライブラリの 1 つです。主に 3D コマンドの生成を行う目的で使用します。

GPU の状態と同期する内部状態をもとにライブラリが必要な 3D コマンドのみを生成しますので、GPU の状態を意識せずにグラフィックス処理を実装することができます。また、API の構成は 3DS のグラフィックスハードウェア構成と密接に結びついており、アプリケーションで簡単に用いることが可能でありながらも、高いパフォーマンスを引き出せるように設計されています。

GD ライブラリは DMPGL(gl 関数)との互換性はありません。コマンドリストの管理など、一部のシステム系機能については GX ライブラリ(nngx 関数)を用いる必要があります。また、C++ のインターフェースのみが提供されています。

1.1.1. コマンドリスト

先に述べたように、GD ライブラリではコマンドリストの操作を管理しておらず、GX ライブラリで提供されている機能をそのまま使用することになります。つまり、コマンドリストをどのように準備したり、使用したり、同期を取ったりするかについては、すべてアプリケーションで実装しなければなりません。

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

1.1.2. イミディエート関数と非イミディエート関数

GD ライブラリでは、現在のコマンドリストオブジェクトで指定されている 3D コマンドバッファに 3D コマンドを書き込むタイミングの違いによって、関数をイミディエート関数と非イミディエート関数の 2 つに分類しています。

イミディエート関数は、呼ばれたタイミングでデータを 3D コマンドバッファに直接書き込む関数のことを指します。パケットレコーディング機能(「6. パケットレコーディング」参照)を使用する場合に、記録されているパケットを直接編集することができるという利点があります。

非イミディエート関数は 3D コマンドバッファへの書き込みを行わずに GD ライブラリの内部状態を更新します。これらの内部状態は描画関数(`nn::gd::System::Draw()` または `nn::gd::System::DrawIndexed()`)が呼び出された際に読み込まれ、必要に応じて 3D コマンドバッファにデータを書き出します。

一部の関数はイミディエート関数と非イミディエート関数の両方の性質を持っており、3D コマンドバッファに書き込みを行いつつ GD ライブラリの内部状態を更新することもあります。

1.1.3. エラーハンドリング

内部で処理に失敗する可能性のある関数は `nnResult` 型のオブジェクトを返り値として返すように実装されています。つまり、エラーが発生した場合には内容に応じたエラー値が返されます。

GD ライブラリでエラーをハンドリングする方法は 2 つ存在します。1 つは呼び出した GD 関数の返り値を常にチェックする方法です。もう 1 つは nn::gd::System::SetCallbackFunctionError() を呼び出してエラーハンドリング用のコールバック関数を登録する方法です。これらの方法は同時に使用することができ、コールバックを呼び出したあとで GD 関数そのものの返り値でエラーを受け取ることができます。

コード 1-1. エラーハンドリング用のコールバック関数の設定

```
typedef void (*nn::gd::System::gdCallbackfunctionErrorPtr)(
            nnResult result, const char* functionName);
static void nn::gd::System::SetCallbackFunctionError(
            nn::gd::System::gdCallbackfunctionErrorPtr callbackFunctionError);
```

> **注意:** GD 関数内部でのエラーチェックの詳細度は、ビルドの種類(Debug/Development/Release)によって異なります。

また、デバッグ用途向けに、与えられた nnResult 型のオブジェクトのエラーに対応したメッセージを nn::gd::System::GetErrorStringFromResult() で取得することができます。

コード 1-2. エラーメッセージの取得(デバッグ用途のみ)

```
static char* nn::gd::System::GetErrorStringFromResult(nnResult result);
```

1.2. モジュール

モジュールは 3DS のレンダリングパイプライン内で関連する処理をひとまとめにしたものです。

GD ライブラリには、以下のモジュールが定義されています。

表 1-1. モジュールの一覧

モジュール	定義名
頂点入力ステージ	nn::gd::System::MODULE_VERTEX_INPUT
シェーダステージ	nn::gd::System::MODULE_SHADER
ラスタライザステージ	nn::gd::System::MODULE_RASTERIZER
テクスチャステージ	nn::gd::System::MODULE_TEXTURE
プロシージャルテクスチャステージ	nn::gd::System::MODULE_TEXTURE_PROCEDURAL
ライティングステージ	nn::gd::System::MODULE_LIGHTING
コンバイナステージ	nn::gd::System::MODULE_TEXTURE_COMBINER
フォグステージ	nn::gd::System::MODULE_GAS_FOG
アウトプットステージ	nn::gd::System::MODULE_OUTPUT

> **補足:** パイプラインでの処理が段階的に分けられていることもあり、モジュールのことをステージと記述することがあります。

1.2.1. Static クラスによる定義

GD ライブラリのほとんどの関数は各モジュールで定義された static なクラスを介してアクセスします。これらのクラスは static な関数のみを持ち、内部状態を持たず、インスタンス化もされていません。つまり、クラス分けは単純に機能ごとに関数定義をグルーピングするために用いられています。

なお、各モジュールに対して 1 つ、または複数個のクラスが以下のように用意されています。

表 1-2. モジュールと対応するクラス

モジュール	対応するクラス	行われる処理
頂点入力ステージ	VertexInputStage	頂点バッファの入力
シェーダステージ	ShaderStage	シェーダプログラムのアタッチ
ラスタライザステージ	RasterizerStage	カリング、クリッピング、ビューポート設定、シザリング、アーリーデプステスト
テクスチャステージ	TextureStage	テクスチャユニットの設定
プロシージャルテクスチャステージ	ProceduralTextureStage	プロシージャルテクスチャの設定
ライティングステージ	LightingStage	フラグメントライティングの設定
コンバイナステージ	CombinerStage	テクスチャコンバイナの設定
フォグステージ	FogStage	フォグの設定、ガスの設定
アウトプットステージ	OutputStage	デプステスト、ステンシルテスト、アルファテスト、論理演算、ブレンディング、書き込みマスク、レンダーターゲットの設定、フラグメントオペレーションの設定

1.2.2. モジュールの状態管理

いくつかのモジュールではライブラリで内部状態が管理されており、2 つのレンダリングパスで内部状態に変更がなかった場合には、そのモジュールに関するコマンドを再送しません。これにより不要なコマンドパケットが生成されず、パフォーマンスを向上させることができます。ただし、何らかの GD 関数の呼び出しによって内部状態が変更された場合、そのモジュールは「dirty」(変更あり)と見なされます。「dirty」と見なされたモジュールはすべて、次回の描画時にその内部状態を参照したコマンド送信が行われます。

モジュールが「dirty」であるかどうかはライブラリで管理され、自動的に処理されます。そのため GD ライブラリを利用するプログラマは通常、どのモジュールが「dirty」であるかについて気にする必要はありません。しかしながら、モジュールに関連したコマンドすべてを確実に再送したいようなケースもあります。そのような場合のために、GD ライブラリでは明示的にモジュールを「dirty」にする手段を以下の関数で提供しています。

コード 1-3. モジュールを明示的に「dirty」にする関数

```
void nn::gd::System::ForceDirty(u32 moduleFlag);
```

moduleFlag には、「dirty」にするモジュールを表 1-1 の定義名の論理和で指定します。

例えば、事前に作成しておいたコマンドパケットを実行したあとや、ほかのグラフィックスライブラリ(NintendoWare や DMPGL、GR ライブラリなど)の使用によって、ハードウェア状態が更新されてしまっている状況で GD ライブラリのすべての状態を確実に反映させたい場合に有効です。

1.3. ステートオブジェクトとリソースオブジェクト

ステートオブジェクトは、作成後に変更されることのない不変のオブジェクトです。初期化時に作成され、実行時の必要なタイミングでそのステートをパイプラインにセットするような使われ方を想定しています。オブジェクトの作成にはメモリ確保やデータ処理を伴うため時間がかかりますが、そこで作成されたステートをあとで 3D コマンドバッファに出力する際の処理は必要最小限の単純なものだけで済みます。これらステートオブジェクトの中身に直接アクセスする方法は提供されていません。

ステートオブジェクトの作成が要求された場合、GD ライブラリはまず以前に全く同じ設定を持ったオブジェクトが作成されているかどうかを確認します。そこで同じ設定のものが見つかった場合は新たなオブジェクトを作成せず、単にそのオブジェクトへのポインタを返します。

ステートオブジェクトとして以下のものが定義されています。

表 1-3. ステートオブジェクト

ステートオブジェクト	使用するステージ	内包する設定
CombinerState	コンバイナステージ	コンバイナ、コンバイナバッファ
BlendState	アウトプットステージ	論理演算、ブレンディング
DepthStencilState	アウトプットステージ	デプステスト、ステンシルテスト
InputLayout	頂点入力ステージ	頂点バッファ
SamplerState	テクスチャステージ	テクスチャパラメータ

また、ステートオブジェクトとは別に以下のようなリソースオブジェクトがあります。作成時に必要な処理がほぼ完了しているという意味では同様の使われ方をします。

表 1-4. リソースオブジェクト

リソースオブジェクト	説明
ShaderBinary	シェーダバイナリの保持やロードを行います。
Shader	浮動小数点定数の GPU へのロードや入出力レジスタの設定を行います。
ShaderPipeline	シェーダパイプラインへのシェーダプログラムのアタッチを行います。
VertexBufferResource	頂点バッファのデータを保持します。
Texture2DResource	テクスチャやカラーバッファなど、幅と高さを持つリソースを保持します。
Texture2D	Texture2DResource を 2 次元テクスチャとして扱います。
TextureCube	6 つの Texture2DResource をキューブマップテクスチャとして扱います。
RenderTarget	Texture2DResource をカラーバッファとして扱います。
DepthStencilTarget	Texture2DResource をデプス(ステンシル)バッファとして扱います。

> **補足:** ステートオブジェクトやリソースオブジェクトを作成するために必要なメモリは、GX ライブラリの初期化で指定したアロケータを介して、デバイスメモリ(NN_GX_MEM_FCRAM)からシステム用バッファ(NN_GX_MEM_SYSTEM)として確保されます。そのため、ライブラリによるメモリ使用を考慮して、デバイスメモリのサイズに余裕を持たせることを推奨します。
>
> 確保されるメモリのサイズについては、「9.2. 関数内で確保されるメモリのサイズ」を参照してください。

1.3.1. デスクリプタクラス

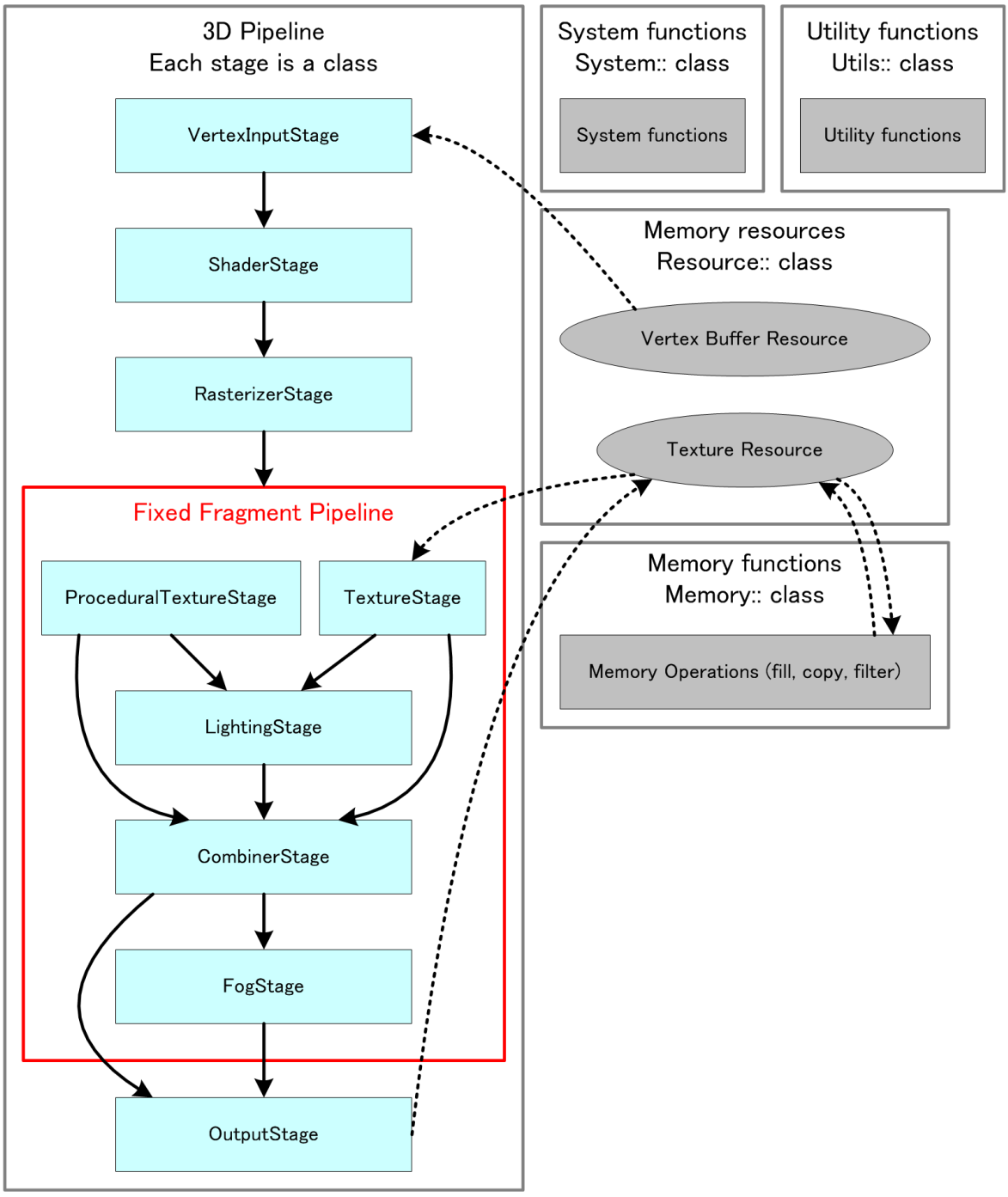
ステートオブジェクトやリソースオブジェクトのようなパイプライン設定用のオブジェクトを作成する際には、必要な設定を行ったデスクリプタクラスをオブジェクト作成関数に与えることになります。

デスクリプタクラスのすべてのメンバ変数は public で定義されています。また、デスクリプタクラスには複数のメンバ変数を簡単に設定するためのメンバ関数が定義されている場合もあります。

1.4. データの流れ

以下の図はデータの観点から GD ライブラリのパイプラインの各ステージを記述したもので、各関数がどのようにクラスに分けられているかを表しています。

図 1-1. GD ライブラリにおけるデータの流れ



2. 基本的な処理の流れ

GD ライブラリを使用してグラフィックスを表示するために最低限必要となる処理を、行うべき順番(多少前後することがあります)で示すと以下のようになります。

- GD ライブラリの初期化
- フレームバッファの確保
- シェーダプログラムのロード
- 頂点データの準備
- ビューポートの設定
- 描画の実行
- 描画結果の表示
- 終了処理

上記以外の処理については「4. 各モジュールの説明」などを参照してください。

> **補足:** この章では関数の詳細については説明しません。関数の詳細な情報については、以降の章や関数リファレンスを参照してください。

2.1. GD ライブラリの初期化

GD ライブラリの初期化は nn::gd::System::Initialize() を呼び出すことで行われます。初期化は GD ライブラリのほかの関数を呼び出す前に行われている必要があります。

また、nn::gd::System::Initialize() は内部での作業用メモリ確保のために nngxInitialize() 経由で与えられているアロケータを使用します。このため、GD ライブラリの初期化に先立って nngxInitialize() による GX ライブラリの初期化と、関数内で実行される 3D コマンドのためにコマンドリストの確保を行う必要があります。

以下は GD ライブラリの初期化に必要な流れの一例です。

コード 2-1. 初期化の例

```
nngxInitialize(...);

nngxGenCmdlists(1, &cmdListId);
nngxBindCmdlist(cmdListId);
nngxCmdlistStorage(0x80000, 128);
nnResult result = nn::gd::System::Initialize();
```

GD ライブラリでは、同じコマンドリストを使用した状態でも、ほかのフレームワーク(NintendoWare など)と平行して利用することができます。ただし、すでに GD ライブラリ以外でコマンドリストを確保していた場合は、GD ライブラリの初期化前にコマンドリストの確保を行わないように注意してください。

ディスプレイバッファを確保するための関数は GD ライブラリにはありません。ディスプレイバッファの確保は nngx 関数で行ってください。

2.2. フレームバッファの確保

カラーバッファとデプス(ステンシル)バッファの確保は GD ライブラリで行います。

デスクリプタクラスの設定に違いがあるものの、どちらのバッファも Texture2DResource オブジェクトを作成しなければな

らないのは同じです。バッファのクリアや描画の効率から、以下のコード例のようにカラーバッファとデプス(ステンシル)バッファは VRAM-A と VRAM-B に分散して確保することを推奨します。

コード 2-2. フレームバッファで使用する Texture2DResource オブジェクトの作成

```
static nn::gd::Texture2DResource* s_texture2DResource_ColorBuffer = 0;
static nn::gd::Texture2DResource* s_texture2DResource_DepthStencilBuffer = 0;

//Color buffer
nn::gd::Texture2DResourceDescription Text2DResDesc_ColorBuffer =
       {nn::gx::DISPLAY0_WIDTH, nn::gx::DISPLAY0_HEIGHT, 1,
        nn::gd::Resource::NATIVE_FORMAT_RGBA_8888,
        nn::gd::Memory::LAYOUT_BLOCK_8, nn::gd::Memory::VRAMA};
nnResult res = nn::gd::Resource::CreateTexture2DResource(
        &Text2DResDesc_ColorBuffer, 0, GD_FALSE,
        &s_texture2DResource_ColorBuffer);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}

//Depth Stencil buffer
nn::gd::Texture2DResourceDescription Text2DResDesc_DepthStencilBuffer =
       {nn::gx::DISPLAY0_WIDTH, nn::gx::DISPLAY0_HEIGHT, 1,
        nn::gd::Resource::NATIVE_FORMAT_DEPTH_24_STENCIL_8,
        nn::gd::Memory::LAYOUT_BLOCK_8, nn::gd::Memory::VRAMB};
res = nn::gd::Resource::CreateTexture2DResource(
        &Text2DResDesc_DepthStencilBuffer, 0, GD_FALSE,
        &s_texture2DResource_DepthStencilBuffer);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}
```

作成した Texture2DResource オブジェクトを使って、カラーバッファならば RenderTarget オブジェクトを、デプス(ステンシル)バッファならば DepthStencilTarget オブジェクトをそれぞれ作成します。

コード 2-3. RenderTarget オブジェクトと DepthStencilTarget オブジェクトの作成

```
static nn::gd::RenderTarget* s_RenderTarget = 0;
static nn::gd::DepthStencilTarget* s_DepthStencilTarget = 0;

//Color buffer
nn::gd::RenderTargetDescription descRenderTarget = {0};
res = nn::gd::OutputStage::CreateRenderTarget(
        s_texture2DResource_ColorBuffer, &descRenderTarget,
        &s_RenderTarget);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}

//Depth Stencil buffer
nn::gd::DepthStencilTargetDescription descDepthStencilTarget = {0};
res = nn::gd::OutputStage::CreateDepthStencilTarget(
        s_texture2DResource_DepthStencilBuffer, &descDepthStencilTarget,
        &s_DepthStencilTarget);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}
```

パイプラインへの設定はアウトプットステージで行い、RenderTarget オブジェクトならば SetRenderTarget() を、DepthStencilTarget オブジェクトならば SetDepthStencilTarget() をそれぞれ呼び出します。

コード 2-4. フレームバッファのパイプラインへの設定

```
//Set the color/depthstencil targets
nn::gd::OutputStage::SetRenderTarget(s_RenderTarget);
nn::gd::OutputStage::SetDepthStencilTarget(s_DepthStencilTarget);
```

2.3. シェーダプログラムのロード

描画に使用するシェーダプログラムをパイプラインに反映させるまでには、以下の手順が必要です。

- シェーダバイナリの読み込み
- ShaderBinary オブジェクトの作成
- Shader オブジェクトの作成
- ShaderPipeline オブジェクトの作成
- パイプラインへの反映

以下のコード例では、シェーダバイナリ(rBuf に rSize バイトのファイルを読み込み済み)に含まれている頂点シェーダをパイプラインに反映させています。

コード 2-5. シェーダプログラムのロード

```
static nn::gd::ShaderBinary* shaderBinary = 0;
static nn::gd::ShaderPipeline* shaderPipeline = 0;
static nn::gd::Shader* vertexShader = 0;

nnResult res;
res = nn::gd::ShaderStage::CreateShaderBinary(rBuf, rSize, &shaderBinary);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}
res = nn::gd::ShaderStage::CreateShader(shaderBinary, 0, &vertexShader);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}
res = nn::gd::ShaderStage::CreateShaderPipeline(
                            vertexShader, NULL, &shaderPipeline);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}

nn::gd::ShaderStage::SetShaderPipeline(shaderPipeline);
```

シェーダプログラムで使用する各レジスタへの値の設定は、シェーダステージの GetShaderUniformLocation() の引数に ShaderPipeline オブジェクトとデータ名を指定して取得した、UniformLocation オブジェクトを介して行います。

コード 2-6. ユニフォーム設定を介したレジスタへの値の設定

```
static nn::gd::UniformLocation s_shaderVariable_proj;
static nn::gd::UniformLocation s_shaderVariable_view;

s_shaderVariable_proj = nn::gd::ShaderStage::GetShaderUniformLocation(
                            shaderPipeline, "uProjection");
s_shaderVariable_view = nn::gd::ShaderStage::GetShaderUniformLocation(
                            shaderPipeline, "uModelView");
NN_ASSERT(s_shaderVariable_proj.IsValid());
NN_ASSERT(s_shaderVariable_view.IsValid());

nn::math::Matrix44 proj, mv;
nn::gd::ShaderStage::SetShaderPipelineConstantFloat(
```

```
        shaderPipeline, s_shaderVariable_proj, static_cast<f32*>(proj));
nn::gd::ShaderStage::SetShaderPipelineConstantFloat(
        shaderPipeline, s_shaderVariable_view, static_cast<f32*>(mv));
```

2.4. 頂点データの準備

GD ライブラリでは、プリミティブの描画には必ず頂点バッファを使用しなければなりません。

頂点バッファを使用するには、以下の手順が必要となります。

- InputLayout オブジェクトの作成
- VertexBufferResource オブジェクトの作成
- パイプラインへの反映(入力レイアウトと頂点バッファ)

以下のコード例では、頂点座標と頂点カラーがインターリーブされた頂点データを使用しています。

コード 2-7. 頂点データの準備

```
u16 idxs[] = {0, 1, 2};
nnResult res;

float coords_color[] = {
     0.5f, 0.0f, 0.f, 1.f, 1.f, 0.0f, 0.0f,
    -0.5f, 0.5f, 0.f, 1.f, 0.f, 1.0f, 0.0f,
    -0.5f,-0.5f, 0.f, 1.f, 0.f, 0.0f, 1.0f,
};

nn::gd::InputElementDescription descs[] =
{
    { 0, "aPosition",
         nn::gd::VertexInputStage::STREAM_TYPE_FLOAT, 4, 0},
    { 0, "aColor",
         nn::gd::VertexInputStage::STREAM_TYPE_FLOAT, 3, sizeof(float) * 4},
};
u32 strides[] = { sizeof(float) * 7 };
res = nn::gd::VertexInputStage::CreateInputLayout(
        descs, 2, strides, vertexShader, &inputLayout);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}

nn::gd::VertexBufferResourceDescription desc;
desc.m_ByteSize = sizeof(coords_color);
desc.m_MemLocation = nn::gd::Memory::FCRAM;
res = nn::gd::Resource::CreateVertexBufferResource(
        &desc, coords_color, &bufferCoord);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}

desc.m_ByteSize = sizeof(idxs);
res = nn::gd::Resource::CreateVertexBufferResource(&desc, idxs, &bufferIndex);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}

nn::gd::VertexBufferResource* buffers[] = { bufferCoord };
u32 offsets[] = { 0 };
nn::gd::VertexInputStage::SetVertexBuffers(0, 1, buffers, offsets);
nn::gd::VertexInputStage::SetInputLayout(inputLayout);
```

```
nn::gd::VertexInputStage::SetIndexBuffer(
        bufferIndex, nn::gd::VertexInputStage::INDEX_FORMAT_USHORT, 0);
```

2.5. ビューポートの設定

ビューポートの設定はラスタライザステージの SetViewport() で行います。

以下のコード例では、アウトプットステージの GetRenderTargetProperties() で取得したレンダーターゲットの情報をビューポートの設定に利用しています。

コード 2-8. ビューポートの設定

```
nn::gd::TargetProperties renderTargetProperty;
nn::gd::OutputStage::GetRenderTargetProperties(
        s_RenderTarget, &renderTargetProperty);
nn::gd::Viewport viewPort(0, 0,
        renderTargetProperty.m_Width, renderTargetProperty.m_Height);
nn::gd::RasterizerStage::SetViewport(viewPort);
```

2.6. 描画の実行

描画のために呼び出す関数は頂点インデックスを使用するかどうかで異なります。頂点インデックスを使用する場合は nn::gd::System::DrawIndexed() を、使用しない場合 nn::gd::System::Draw() を呼び出します。なお、頂点バッファリソースを使用せずに描画を行う nn::gd::System::DrawImmediate() と nn::gd::System::DrawImmediateIndexed() も用意されています。ただし、頂点バッファリソースを使用しない描画は頂点データの取り扱いが柔軟になりますが、描画の実行が遅くなります。

フレームバッファのクリアには nn::gd::Memory::ClearTargets() を使用してください。

以下のコード例では、フレームバッファのクリアと頂点インデックスを使用した描画を行っています。

コード 2-9. 描画の実行

```
//Clear the render buffers
u8 clearColor[] = {25, 25, 122, 255};
nn::gd::Memory::ClearTargets(s_RenderTarget, s_DepthStencilTarget,
                             clearColor, 1.f, 0);
//Draw
nn::gd::System::DrawIndexed(3, 0);
```

2.7. 描画結果の表示

カラーバッファの内容を nn::gd::Memory::CopyTexture2DResourceBlockToLinear() でディスプレイバッファにコピーし、描画結果を LCD に表示させます。この関数は nngxTransferRenderImage() とほぼ同じ機能を有していますが、コピー先のアドレスは nngxGetDisplaybufferParameteri() などを利用してアプリケーションで取得しなければなりません。

コード 2-10. 描画結果の表示

```
//Transfer framebuffer to display buffer
int dstAddr;
nngxActiveDisplay(NN_GX_DISPLAY0);
nngxGetDisplaybufferParameteri(NN_GX_DISPLAYBUFFER_ADDRESS, &dstAddr);
nn::gd::Memory::CopyTexture2DResourceBlockToLinear(
    s_texture2DResource_ColorBuffer,            //source
    0,                                          //sourceMipLevelIndex
    0,                                          //srcOffsetY
    (u8*)dstAddr,                               //dstAddr
    nn::gx::DISPLAY0_WIDTH,                     //dstWidth
    nn::gx::DISPLAY0_HEIGHT,                    //dstHeight
    nn::gd::Resource::NATIVE_FORMAT_RGB_888,    //dstFormat
    nn::gd::Memory::DOWNSCALING_NONE,           //DownScalingMode
    GD_FALSE                                    //yFlip
);
nngxWaitCmdlistDone();
nngxSwapBuffers(NN_GX_DISPLAY_BOTH);
```

2.8. 終了処理

GD ライブラリの終了処理は nn::gd::System::Finalize() を呼び出すことで行われます。この関数の実行が完了したあとは GD ライブラリの関数を呼び出すことができなくなります。

終了処理では、アプリケーションで確保したものを含めて、ライブラリが管理しているリソースなどが自動的に解放されます。しかし、アプリケーションで作成したステートやリソースなどのオブジェクトは、なるべくアプリケーションで明示的に解放することを推奨します。

3. リソースオブジェクト

リソースオブジェクトを扱う関数は基本的に nn::gd::Resource クラスで定義されています。

nn::gd::Resource クラスでは、リソースオブジェクトの作成と解放、詳細情報の取得、データを書き換えるためのデータポインタの取得を行うことができます。

3.1. テクスチャ 2D リソース

テクスチャ 2D リソースは、テクスチャイメージやカラーバッファなどのように、幅と高さを持つリソースのことです。このリソースは Texture2DResource オブジェクトを介して扱います。

3.1.1. Texture2DResource オブジェクトの作成

Texture2DResource オブジェクトは nn::gd::Resource::CreateTexture2DResource() で作成することができます。

コード 3-1. Texture2DResource オブジェクトの作成

```
static nnResult nn::gd::Resource::CreateTexture2DResource(
        const nn::gd::Texture2DResourceDescription* description,
        const void* initialData, gdBool autoGenerateMipMapsFromData,
        nn::gd::Texture2DResource** texture2DResource,
        gdBool copyInitialData = GD_TRUE);
```

リソースの情報は *description* に指定するデスクリプタクラス(Texture2DResourceDescription)にあらかじめ設定しておきます。デスクリプタクラスに設定する情報の詳細は「3.1.1.2. Texture2DResource オブジェクトのデスクリプタクラス」を参照してください。

ファイルからロードされたテクスチャデータのように初期データを持つリソースを作成する場合、そのデータの先頭アドレスを *initialData* に指定します。*copyInitialData* に GD_TRUE が指定されたときは、リソースの作成時に初期データをデスクリプタクラスで指定されたメモリにコピーします。このとき、リソースデータを VRAM に確保する場合は関数内で DMA 転送のコマンドリクエストをコマンドリストに追加しますので、コマンドリストの実行が完了するまで初期データが存在するメモリ領域を書き換えたり解放したりしないでください。*copyInitialData* に GD_FALSE が指定されたときは、*initialData* に指定されたアドレスそのままをリソースで使用します。リソースにミップマップデータが含まれる場合は、すべてのデータが格納可能なサイズで確保されていなければならないことに注意してください。また、アドレスのアライメントにも注意が必要です。なお、リソースの解放時にはアプリケーションでメモリを解放しなければなりません。

初期データを持たないリソースを作成する場合は *initialData* には NULL を指定してください。リソースデータを保持するために必要なサイズのメモリ領域が、デスクリプタクラスで指定されたメモリに確保されます。

autoGenerateMipMapsFromData には、テクスチャデータのミップマップデータを初期データから自動生成する場合に GD_TRUE を指定します。自動生成する場合の注意点については、「3.1.7. ミップマップデータの自動生成」を参照してください。

今後、作成された Texture2DResource オブジェクトへのアクセスには *texture2DResource* で受け取ったポインタを介して行います。

3.1.1.1. テクスチャをネイティブフォーマットに変換するヘルパー関数

3D モデルに貼り付けるテクスチャのテクスチャオブジェクトを作成する場合、Texture2DResource オブジェクトの作成時に指定する初期データは 3DS のネイティブフォーマットでなければなりません。そのため、Open GL などで使用される標準

的なテクスチャを 3DS のネイティブフォーマットに変換するヘルパー関数が nn::gd::Resource::Helper クラスに用意されています。ただし、ヘルパー関数は処理の負荷が高く、ランタイムでの処理には向いていません。

無圧縮テクスチャと圧縮テクスチャで変換に使用するヘルパー関数は異なりますが、対応している変換元のピクセルフォーマット以外に、引数の指定に違いはありません。

コード 3-2. テクスチャをネイティブフォーマットに変換するヘルパー関数

```
static nnResult nn::gd::Resource::Helper::ConvertTextureResourceToNativeFormat(
        nn::gd::Resource::Format format,
        u32 width, u32 height, const u8* dataSrc, u8* dataDst,
        nn::gd::Resource::NativeFormat* pnativeFormat = NULL);
static nnResult
nn::gd::Resource::Helper::ConvertCompressedTextureResourceToNativeFormat(
        nn::gd::Resource::CompressedFormat format,
        u32 width, u32 height, u8* dataSrc, u8* dataDst,
        nn::gd::Resource::NativeFormat* pnativeFormat = NULL);
```

format には変換元のピクセルフォーマットを指定します。

ConvertTextureResourceToNativeFormat() で指定可能なピクセルフォーマットには以下のものがあります。

- FORMAT_RGBA_8888
- FORMAT_RGB_888
- FORMAT_RGBA_5551
- FORMAT_RGB_565
- FORMAT_RGBA_4444
- FORMAT_LUMINANCE_ALPHA_88
- FORMAT_HILO_88
- FORMAT_LUMINANCE_8
- FORMAT_ALPHA_8
- FORMAT_LUMINANCE_ALPHA_44
- FORMAT_LUMINANCE_4
- FORMAT_ALPHA_4

ConvertCompressedTextureResourceToNativeFormat() で指定可能なピクセルフォーマットには以下のものがあります。

- FORMAT_ETC1_RGB8

ピクセルフォーマットによっては、デバイスメモリから変換元のデータと同じサイズのバッファを一時的に確保します。また、*dataSrc* と *dataDst* に同じアドレスを指定した場合は必ず一時的にバッファを確保します。

標準的なテクスチャのフォーマットとネイティブフォーマットの違いについては、「3DS プログラミングマニュアル – グラフィックス基本編」を参照してください。

3.1.1.2. Texture2DResource オブジェクトのデスクリプタクラス

Texture2DResource オブジェクトのデスクリプタクラス(Texture2DResourceDescription)のメンバを紹介し、設定に際しての注意点などを説明します。

コード 3-3. Texture2DResourceDescription クラスの定義

```
class nn::gd::Texture2DResourceDescription
{
    u32 m_Width;
    u32 m_Height;
    u32 m_CountMipLevels;
    nn::gd::Resource::NativeFormat m_Format;
    nn::gd::Memory::MemoryLayout m_MemLayout;
    nn::gd::Memory::MemoryLocation m_MemLocation;
};
```

幅(m_Width)

リソースデータの幅をピクセル数で設定します。ミップマップデータを持つテクスチャの場合は、トップレベルのテクスチャデータの幅を指定してください。

高さ(m_Height)

リソースデータの高さをピクセル数で設定します。ミップマップデータを持つテクスチャの場合は、トップレベルのテクスチャデータの高さを指定してください。

ミップマップレベル(m_CountMipLevels)

リソースデータがミップマップデータを持つ場合のミップマップレベルを設定します。ミップマップデータを持たないリソースデータの場合は 1 を指定してください。0 を指定した場合はリソースデータの幅と高さで持ち得る最大のミップマップレベルが設定されます。

ピクセルフォーマット(m_Format)

リソースの使用目的によって指定可能なピクセルフォーマットが異なります。

下表では、使用目的(作成するオブジェクト)をテクスチャデータ(Texture2D オブジェクトまたは TextureCube オブジェクト)、カラーバッファ(RenderTarget オブジェクト)、デプスステンシルバッファ(DepthStencilTarget オブジェクト)に分けて指定可能なピクセルフォーマットを示しています。

表 3-1. 指定可能なピクセルフォーマット

ピクセルフォーマット	Texture2D	TextureCube	RenderTarget	DepthStencilTarget
Resource::NATIVE_FORMAT_RGBA_8888※1	○	○	○	エラー
Resource::NATIVE_FORMAT_RGB_888※1	○	○	エラー	エラー
Resource::NATIVE_FORMAT_RGBA_5551※1	○	○	○	エラー
Resource::NATIVE_FORMAT_RGB_565※1	○	○	○	エラー
Resource::NATIVE_FORMAT_RGBA_4444※1	○	○	○	エラー
Resource::NATIVE_FORMAT_LUMINANCE_ALPHA_88	○	○	エラー	エラー
Resource::NATIVE_FORMAT_HILO_88	○	○	エラー	エラー
Resource::NATIVE_FORMAT_LUMINANCE_8	○	○	エラー	エラー
Resource::NATIVE_FORMAT_ALPHA_8	○	○	エラー	エラー

Resource::NATIVE_FORMAT_LUMINANCE_ALPHA_44	○	○	エラー	エラー
Resource::NATIVE_FORMAT_LUMINANCE_4	○	○	エラー	エラー
Resource::NATIVE_FORMAT_ALPHA_4	○	○	エラー	エラー
Resource::NATIVE_FORMAT_ETC1_RGB8	○	○	エラー	エラー
Resource::NATIVE_FORMAT_ETC1_A4	○	○	エラー	エラー
Resource::NATIVE_FORMAT_GAS※2	○	エラー	○	エラー
Resource::NATIVE_FORMAT_SHADOW※2	○	○	○	エラー
Resource::NATIVE_FORMAT_DEPTH_16	○	○	エラー	○
Resource::NATIVE_FORMAT_DEPTH_24	○	○	エラー	○
Resource::NATIVE_FORMAT_DEPTH_24_STENCIL_8	○	○	エラー	○

※1 自動ミップマップ生成に対応しています。

※2 ミップマップに対応していません。

メモリレイアウト(m_MemLayout)

Texture2D や TextureCube のオブジェクトの作成に使用するリソースオブジェクトや初期データを指定して作成するリソースオブジェクトの場合、メモリレイアウトには Memory::LAYOUT_BLOCK_8 のみが指定可能です。

RenderTarget や DepthStencilTarget のオブジェクトの作成に使用するリソースオブジェクトの場合、メモリレイアウトには Memory::LAYOUT_BLOCK_8 または Memory::LAYOUT_BLOCK_32 が指定可能です。

上記以外に使用するリソースオブジェクトであれば、メモリレイアウトに Memory::LAYOUT_LINEAR を指定することができます。

リソースデータを確保するメモリ(m_MemLocation)

リソースデータを保持するために確保するメモリを指定します。Memory::FCRAM(デバイスメモリ)を指定して作成されたリソースオブジェクトは、RenderTarget または DepthStencilTarget のオブジェクトの作成に使用することができません。

表 3-2. リソースデータの確保で指定可能なメモリ

リソースデータの保持に使用するメモリ	Texture2D	TextureCube	RenderTarget	DepthStencilTarget
Memory::FCRAM(デバイスメモリ)	○	○	エラー	エラー
Memory::VRAMA(VRAM-A)	○	○	○	○
Memory::VRAMB(VRAM-B)	○	○	○	○

3.1.2. Texture2DResource オブジェクトの解放

Texture2DResource オブジェクトの解放は、nn::gd::Resource::ReleaseTexture2DResource() の呼び出しで行われます。オブジェクトの解放と同時に、作成時に確保されたメモリ領域も解放されます。

コード 3-4. Texture2DResource オブジェクトの解放

```
static nnResult nn::gd::Resource::ReleaseTexture2DResource(
        nn::gd::Texture2DResource* texture2DResource);
```

3.1.3. Texture2DResource オブジェクトの詳細情報の取得

Texture2DResource オブジェクトに格納されている詳細情報(Texture2DResourceProperties クラス)を nn::gd::Resource::GetTexture2DResourceProperties() で取得することができます。

コード 3-5. Texture2DResource オブジェクトの詳細情報の取得

```
static nnResult nn::gd::Resource::GetTexture2DResourceProperties(
        const nn::gd::Texture2DResource* texture2DResource,
        nn::gd::Texture2DResourceProperties* properties);
```

この関数は *texture2DResource* に指定した、Texture2DResource オブジェクトに格納されている詳細情報へのポインタを *properties* に返します。取得したポインタを介して、Texture2DResource オブジェクトの内容を書き換えてしまわないように注意してください。

3.1.3.1. Texture2DResource オブジェクトの詳細情報

詳細情報(Texture2DResourceProperties クラス)には、オブジェクトの作成時に指定されたデスクリプタクラスの設定とほぼ同じ内容が格納されています。

コード 3-6. Texture2DResourceProperties クラスの定義

```
class nn::gd::Texture2DResourceProperties
{
    u32 m_Width;
    u32 m_Height;
    u32 m_CountMipLevels;
    u32 m_PixelSize;
    nn::gd::Resource::NativeFormat m_Format;
    nn::gd::Memory::MemoryLayout m_MemLayout;
    nn::gd::Memory::MemoryLocation m_MemLocation;
    u8* m_MemAddr;

    nn::gd::MipmapResourceInfo GetMipmapAddress(u32 mipmapLevel);
};
```

ここでは、デスクリプタクラスの設定と異なる可能性のあるメンバ変数と、このクラスにのみ存在するメンバを説明します。

ミップマップレベル(m_CountMipLevels)

デスクリプタで 0 が指定されていた場合は、リソースデータの幅と高さで持ち得る最大のミップマップレベルが設定されています。

ピクセルサイズ(m_PixelSize)

リソースデータのピクセルフォーマットのビット数が設定されています。

リソースデータの先頭アドレス(m_MemAddr)

リソースデータの先頭アドレスが物理アドレスで設定されています。

ミップマップデータのアドレス情報の取得(GetMipmapAddress())

ミップマップレベルを指定して、ミップマップデータのアドレス情報(MipmapResourceInfo クラス)を取得するヘルパー関数が定義されています。

コード 3-7. ミップマップデータのアドレス情報の定義

```
class nn::gd::MipmapResourceInfo
{
    u32 m_Width;
    u32 m_Height;
    u8* m_MemAddr;
};
```

ミップマップデータの幅(m_Width)、高さ(m_Height)、データの先頭アドレス(m_MemAddr)が定義されています。

3.1.4. Texture2DResource オブジェクトのデータポインタの取得

リソースデータにアクセスするためのデータポインタを nn::gd::Resource::MapTexture2DResource() で取得することができます。

コード 3-8. Texture2DResource オブジェクトのデータポインタの取得

```
static nnResult nn::gd::Resource::MapTexture2DResource(
        nn::gd::Texture2DResource* texture2DResource,
        s32 mipLevelIndex, nn::gd::Resource::MapUsage usage, void** data);
static nnResult nn::gd::Resource::UnmapTexture2DResource(
        nn::gd::Texture2DResource* texture2DResource);
```

texture2DResource に指定した Texture2DResource オブジェクトのリソースデータのうち、*mipLevelIndex* で指定されたミップマップレベルのテクスチャデータの先頭アドレスが *data* に返されます。*usage* には、以下のマッピング方法(リソースデータへのアクセスの種類)を指定します。

表 3-3. データポインタのマッピング方法

定義	マッピング方法	デバイスメモリ	VRAM
Resource::MAP_READ_ONLY	読み込みのみ	○	○
Resource::MAP_WRITE_ONLY	書き込みのみ	○	エラー
Resource::MAP_READ_WRITE	読み書き	○	エラー

補足: マッピング方法のチェックはデータポインタの取得時にのみ行われます。MAP_READ_ONLY で取得したデバイスメモリ上のリソースデータに書き込みを行ってもエラーにはなりませんが、マッピングの解除時にキャッシュがフラッシュされません。

VRAM 上のリソースデータに直接アクセスすることはできません。DMA 転送などで、CPU から直接アクセス可能なメモリ上にコピーしてください。

リソースデータへのアクセスが完了したあとは、nn::gd::Resource::UnmapTexture2DResource() でマッピングを解除してください。書き込みを含むマッピング方法でデバイスメモリ上のリソースデータのデータポインタを取得していた場合は、マッピングの解除時にキャッシュのフラッシュが行われます。

3.1.5. リソースデータのクリア（Texture2DResource オブジェクト）

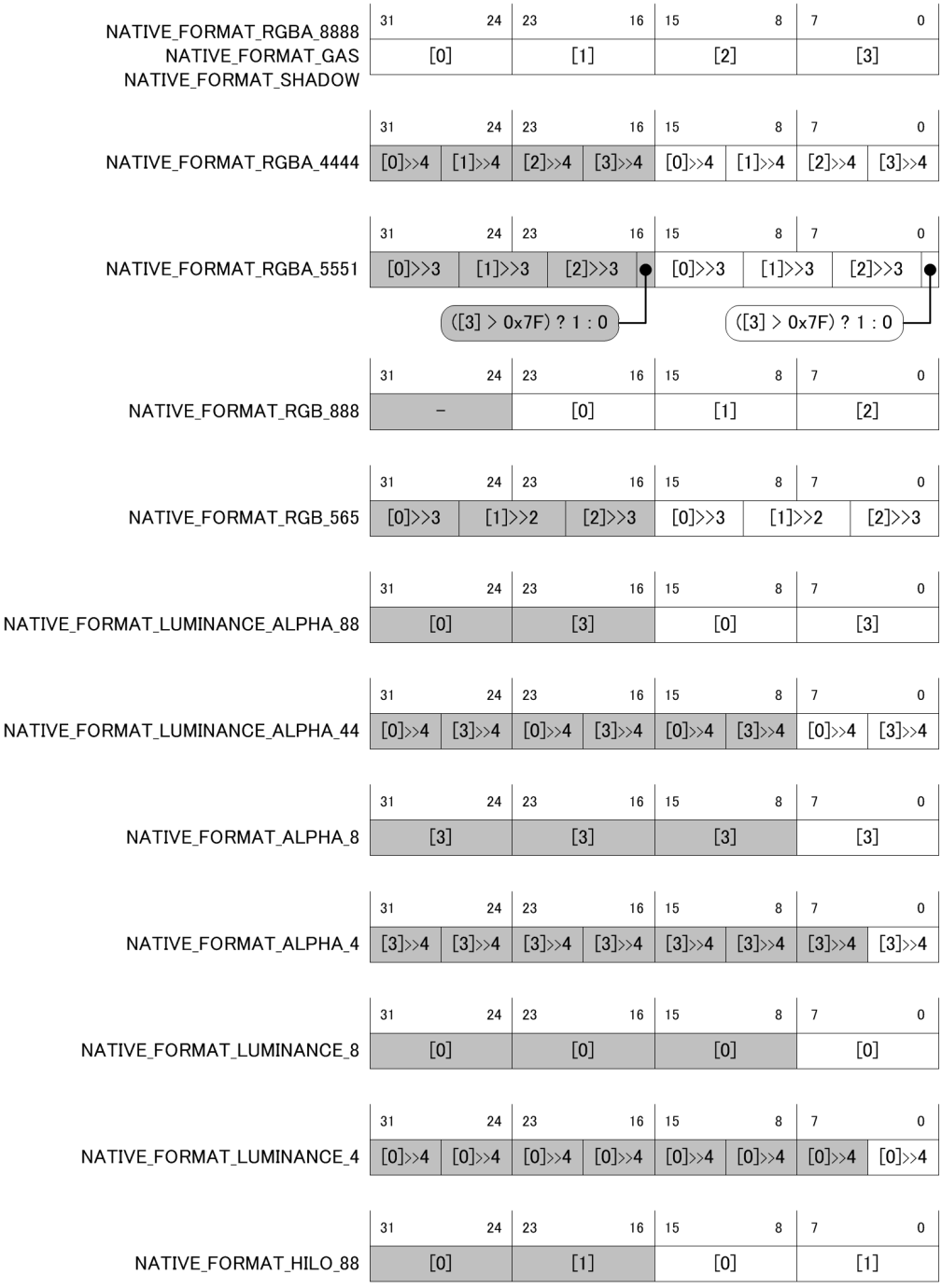
指定した値でリソースデータをクリアすることができます。また、ミップマップデータを持つリソースであれば、クリアするミップマップレベルの範囲を指定することができます。ただし、**デバイスメモリ上に確保されているリソースデータはクリアできません**。

コード 3-9. リソースデータのクリア（Texture2DResource オブジェクト）

```
static nnResult nn::gd::Memory::ClearTexture2DResource(
        const nn::gd::Texture2DResource* tex2DResource,
        s32 mipLevelIndexStart, s32 mipLevelIndexEnd, const u8 Components[4]);
static nnResult nn::gd::Memory::ClearTexture2DResource(
        const nn::gd::Texture2DResource* tex2DResource,
        s32 mipLevelIndexStart, s32 mipLevelIndexEnd, u32 value);
```

value で指定する場合はどんなリソースでもクリアしようと試みますが、*Components* で指定する場合は以下のピクセルフォーマットに限定されます。なお、*Components* の指定がどのような構成でクリアデータに反映されるかを、右のビットレイアウトに示しています。

図 3-1. クリアで指定可能なピクセルフォーマットとクリア時のデータの構成



3.1.6. リソースデータの部分コピー（Texture2DResource オブジェクト）

ミップマップレベルと矩形領域を指定して、リソースデータの部分コピーを行うことができます。この関数内で `nngxAddBlockImageCopyCommand()` を呼び出していますので、フォーマット変換は行われません。実際のコピー処理はコマンドリストの実行時に行われます。

コード 3-10. リソースデータの部分コピー(Texture2DResource オブジェクト)

```
static nnResult nn::gd::Memory::CopyTextureSubResource(
        const nn::gd::Texture2DResource* source,
        s32 srcMipLevelIndex, const nn::gd::Memory::Rect& rect,
        const nn::gd::Texture2DResource* dest, s32 dstMipLevelIndex,
        s32 destPosX, s32 destPosY);
```

リソースデータとビューポート(LCD)では原点の位置が異なりますが、この関数ではそれを考慮していますので、矩形領域(*rect*)やコピー先の座標(*destPosX*、*destPosY*)はビューポートの原点を基準に指定してください。

コピー元とコピー先のピクセルサイズが異なる場合はエラー(ResultInvalidTextureFormat())が返されます。

3.1.7. ミップマップデータの自動生成

Texture2DResource オブジェクトのミップマップデータを自動生成することができます。

コード 3-11. ミップマップデータの自動生成

```
static nnResult nn::gd::Memory::GenerateMipMaps(
        const nn::gd::Texture2DResource* tex2DResource,
        u32 mipLevelSourceIndex, s32 mipLevelLastDestinationIndex);
```

tex2DResource に指定される Texture2DResource オブジェクトは、生成されるミップマップレベルのリソースデータを保持することができるように作成されていなければなりません。また、以下のフォーマットで作成された Texture2DResource オブジェクトでなければ、ミップマップデータの自動生成を行うことができません。

- nn::gd::Resource::NATIVE_FORMAT_RGBA_8888
- nn::gd::Resource::NATIVE_FORMAT_RGBA_5551
- nn::gd::Resource::NATIVE_FORMAT_RGBA_4444
- nn::gd::Resource::NATIVE_FORMAT_RGB_888
- nn::gd::Resource::NATIVE_FORMAT_RGB_565

上記以外のフォーマットの場合は、nn::gd::Resource::CreateTexture2DResourceCastFrom() で事前に NATIVE_FORMAT_RGBA_8888 などにフォーマット変換を行うことでミップマップデータを生成することができます。ただし、ピクセルを構成するビットに違いがあるため、この方法で生成されたミップマップデータが正常であることは保証されていません。

mipLevelSourceIndex から *mipLevelLastDestinationIndex* までのミップマップレベルのデータが自動生成されます。*mipLevelLastDestinationIndex* に -1 を指定した場合は、そのリソースの最大のミップマップレベルを指定したことになります。

3.1.7.1. CPU によるミップマップデータの自動生成

nn::gd::Resource::Helper::GenerateMipMapsCPU() の呼び出しでも、ミップマップデータを自動生成することができます。CPU で生成するため、GPU を使用する nn::gd::Memory::GenerateMipMaps() に比べて処理速度は落ちますが、対応するフォーマットが多いことと、最小で幅および高さが 8 ピクセル(GPU を使用した場合はフォーマットにより 32 または 64 ピクセル)のミップマップデータを生成できることが長所となっています。

コード 3-12. CPU によるミップマップデータの自動生成

```
static nnResult nn::gd::Resource::Helper::GenerateMipMapsCPU(
        nn::gd::Resource::NativeFormat format,
        u32 width, u32 height, const u8* dataSrc, u8* dataDst,
        u32 countMipLevelToGenerate);
```

format には、以下のフォーマットを指定することができます。

- nn::gd::Resource::NATIVE_FORMAT_RGBA_8888
- nn::gd::Resource::NATIVE_FORMAT_RGBA_5551
- nn::gd::Resource::NATIVE_FORMAT_RGBA_4444
- nn::gd::Resource::NATIVE_FORMAT_RGB_888
- nn::gd::Resource::NATIVE_FORMAT_RGB_565
- nn::gd::Resource::NATIVE_FORMAT_LUMINANCE_ALPHA_88
- nn::gd::Resource::NATIVE_FORMAT_LUMINANCE_ALPHA_44
- nn::gd::Resource::NATIVE_FORMAT_LUMINANCE_8
- nn::gd::Resource::NATIVE_FORMAT_ALPHA_8
- nn::gd::Resource::NATIVE_FORMAT_HILO_88

width と *height* には元となるリソースデータの幅と高さを指定します。幅および高さは 8 以上、かつ 2 の累乗でなければなりません。

dataSrc には元となるリソースデータの先頭アドレス、*dataDst* にはミップマップデータの格納先アドレスを指定します。生成されるミップマップデータのすべてが格納できるだけのメモリを確保してください。

countMipLevelToGenerate には生成するミップマップレベルの数を指定します。

3.1.8. Texture2DResource オブジェクトのフォーマット変換

ガステクスチャやシャドウテクスチャ、デプスバッファのように、レンダリングで使用した RenderTarget オブジェクトのリソースデータをテクスチャデータ(Texture2D オブジェクトのリソースデータ)として使用するために、以下の関数で既存の Texture2DResource オブジェクトをほかのフォーマットに変換することができます。

コード 3-13. Texture2DResource オブジェクトのフォーマット変換

```
static nnResult nn::gd::Resource::CreateTexture2DResourceCastFrom(
        const nn::gd::Texture2DResource* initialTexture2DResource,
        nn::gd::Resource::NativeFormat format,
        nn::gd::Memory::MemoryLayout layout,
        nn::gd::Texture2DResource** texture2DResource);
```

initialTexture2DResource に指定した Texture2DResource オブジェクトの、ピクセルフォーマットとメモリレイアウトを *format* と *layout* に指定した情報に変更したオブジェクトが *texture2DResource* に返されます。変換前と変換後のピクセルフォーマットに互換性がない(ピクセルサイズが一致しない)場合はエラーが返されます。

この関数はフォーマットの情報が変更された Texture2DResource オブジェクトを新たに作成します。ただし、フォーマットの変換やデータのコピーは行われず、変換前のオブジェクトは変更されません。また、データが格納されている場所など、変更された情報以外は変換前の情報のままですので、この関数で作成したオブジェクトを介してリソースデータを書き換えた場合には変換前のオブジェクトのリソースデータも書き換わることになります。

3.2. 頂点バッファリソース

頂点バッファリソースは、頂点座標や頂点インデックスなどのように、頂点バッファとして利用するリソースのことです。このリソースは VertexBufferResource オブジェクトを介して扱います。

3.2.1. VertexBufferResource オブジェクトの作成

VertexBufferResource オブジェクトは nn::gd::Resource::CreateVertexBufferResource() で作成することができます。

コード 3-14. VertexBufferResource オブジェクトの作成

```
static nnResult nn::gd::Resource::CreateVertexBufferResource(
        const nn::gd::VertexBufferResourceDescription* description,
        const void* initialData, nn::gd::VertexBufferResource** buffer,
        gdBool copyInitialData = GD_TRUE);
```

リソースの情報は *description* に指定するデスクリプタクラス(VertexBufferResourceDescription)にあらかじめ設定しておきます。デスクリプタクラスに設定する情報の詳細は「3.2.1.1. VertexBufferResource オブジェクトのデスクリプタクラス」を参照してください。

ファイルからロードされた頂点データのように初期データを持つリソースを作成する場合、そのデータの先頭アドレスを *initialData* に指定します。*copyInitialData* に GD_TRUE が指定されたときは、リソースの作成時に初期データをデスクリプタクラスで指定されたメモリにコピーします。このとき、リソースデータを VRAM に確保する場合は関数内で DMA 転送のコマンドリクエストをコマンドリストに追加しますので、コマンドリストの実行が完了するまで初期データが存在するメモリ領域を書き換えたり解放したりしないでください。*copyInitialData* に GD_FALSE が指定されたときは、*initialData* に指定されたアドレスそのままをリソースで使用しますので、アドレスのアライメントに注意してください。なお、リソースの解放時にはアプリケーションでメモリを解放しなければなりません。

初期データを持たないリソースを作成する場合は *initialData* には NULL を指定してください。リソースデータを保持するために必要なサイズのメモリ領域が、デスクリプタクラスで指定されたメモリに確保されます。

今後、作成された VertexBufferResource オブジェクトへのアクセスには *buffer* で受け取ったポインタを介して行います。

3.2.1.1. VertexBufferResource オブジェクトのデスクリプタクラス

VertexBufferResource オブジェクトのデスクリプタクラス(VertexBufferResourceDescription)のメンバを紹介し、設定に際しての注意点などを説明します。

コード 3-15. VertexBufferResourceDescription クラスの定義

```
class nn::gd::VertexBufferResourceDescription
{
    u32 m_ByteSize;
    nn::gd::Memory::MemoryLocation m_MemLocation;
};
```

頂点バッファのサイズ(m_ByteSize)

頂点バッファ(リソースデータ)のバイトサイズを設定します。

リソースデータを確保するメモリ(m_MemLocation)

リソースデータを保持するために確保するメモリを指定します。

指定するメモリは Memory::FCRAM(デバイスメモリ)、Memory::VRAMA(VRAM-A)、Memory::VRAMB(VRAM-B)から選択します。

3.2.2. VertexBufferResource オブジェクトの解放

nn::gd::Resource::ReleaseVertexBufferResource() を呼び出すことで、VertexBufferResource オブジェクトの解放が行われます。オブジェクトの解放と同時に、作成時に確保されたメモリ領域も解放されます。

コード 3-16. VertexBufferResource オブジェクトの解放

```
static nnResult nn::gd::Resource::ReleaseVertexBufferResource(
        nn::gd::VertexBufferResource *buffer);
```

3.2.3. VertexBufferResource オブジェクトの詳細情報の取得

VertexBufferResource オブジェクトに格納されている詳細情報(VertexBufferResourceProperties クラス)を nn::gd::Resource::GetVertexBufferResourceProperties() で取得することができます。

コード 3-17. VertexBufferResource オブジェクトの詳細情報の取得

```
static nnResult nn::gd::Resource::GetVertexBufferResourceProperties(
        const nn::gd::VertexBufferResource* buffer,
        nn::gd::VertexBufferResourceProperties* properties);
```

この関数は *buffer* に指定した、VertexBufferResource オブジェクトに格納されている詳細情報へのポインタを *properties* に返します。取得したポインタを介して、VertexBufferResource オブジェクトの内容を書き換えてしまわないように注意してください。

3.2.3.1. VertexBufferResource オブジェクトの詳細情報

詳細情報(VertexBufferResourceProperties クラス)には、オブジェクトの作成時に指定されたデスクリプタクラスの設定とほぼ同じ内容が格納されています。

コード 3-18. VertexBufferResourceProperties クラスの定義

```
class nn::gd::VertexBufferResourceProperties
{
    u32 m_ByteSize;
    nn::gd::Memory::MemoryLocation m_MemLocation;
    u8* m_MemAddr;
};
```

ここでは、このクラスにのみ存在するメンバを説明します。

リソースデータの先頭アドレス(m_MemAddr)

リソースデータの先頭アドレスが物理アドレスで設定されています。

3.2.4. VertexBufferResource オブジェクトのデータポインタの取得

リソースデータにアクセスするためのデータポインタを nn::gd::Resource::MapVertexBufferResource() で取得することができます。

コード 3-19. VertexBufferResource オブジェクトのデータポインタの取得

```
static nnResult nn::gd::Resource::MapVertexBufferResource(
        nn::gd::VertexBufferResource* buffer,
        nn::gd::Resource::MapUsage usage, void** data);
static nnResult nn::gd::Resource::UnmapVertexBufferResource(
        nn::gd::VertexBufferResource* buffer);
```

buffer に指定した VertexBufferResource オブジェクトのリソースデータの先頭アドレスが *data* に返されます。*usage* には、以下のマッピング方法(リソースデータへのアクセスの種類)を指定します。

表 3-4. データポインタのマッピング方法

定義	マッピング方法	デバイスメモリ	VRAM
Resource::MAP_READ_ONLY	読み込みのみ	○	○
Resource::MAP_WRITE_ONLY	書き込みのみ	○	エラー
Resource::MAP_READ_WRITE	読み書き	○	エラー

補足: マッピング方法のチェックはデータポインタの取得時にのみ行われます。MAP_READ_ONLY で取得したデバイスメモリ上のリソースデータに書き込みを行ってもエラーにはなりませんが、マッピングの解除時にキャッシュがフラッシュされません。

VRAM 上のリソースデータに直接アクセスすることはできません。DMA 転送などで、CPU から直接アクセス可能なメモリ上にコピーしてください。

リソースデータへのアクセスが完了したあとは、nn::gd::Resource::UnmapVertexBufferResource() でマッピングを解除してください。書き込みを含むマッピング方法でデバイスメモリ上のリソースデータのデータポインタを取得していた場合は、マッピングの解除時にキャッシュのフラッシュが行われます。

3.2.5. リソースデータの部分コピー(VertexBufferResource オブジェクト)

リソースデータの部分コピーを行うことができます。この関数内で nngxAddBlockImageCopyCommand() を呼び出していますので、実際のコピー処理はコマンドリストの実行時に行われます。

コード 3-20. リソースデータの部分コピー(VertexBufferResource オブジェクト)

```
static nnResult nn::gd::Memory::CopyVertexBufferSubResource(
        const nn::gd::VertexBufferResource* source,
        u32 sourceOffset, u32 size,
        const nn::gd::VertexBufferResource* dest, u32 destOffset);
```

sourceOffset、*size*、*destOffset* のすべてが 16 の倍数で指定されていなければコピー処理は行われません。

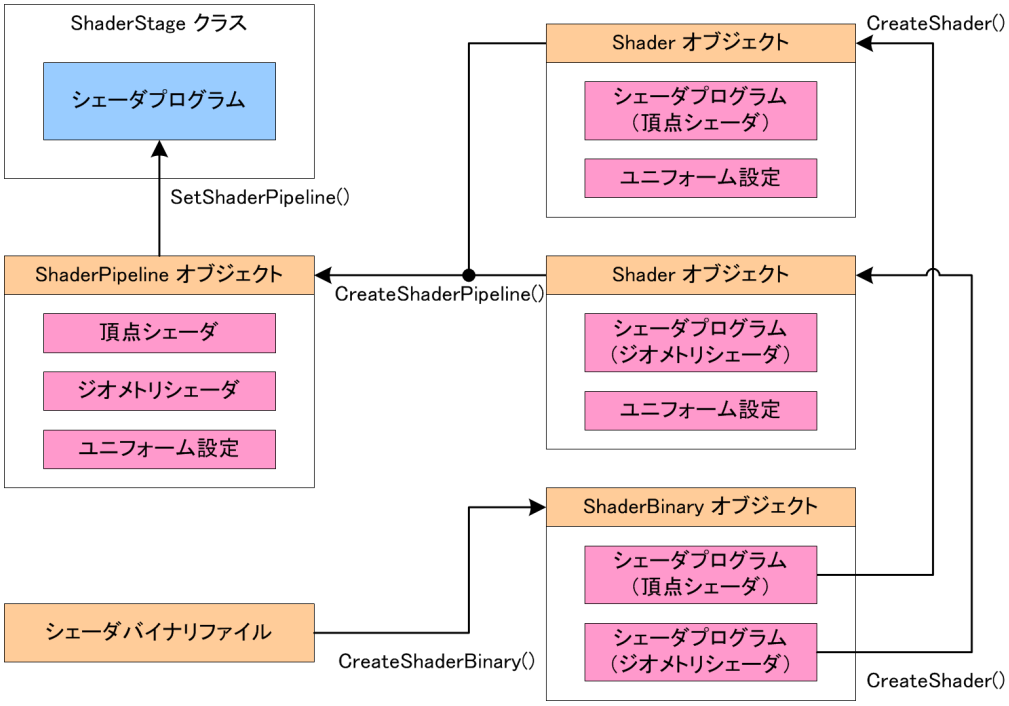
4. 各モジュールの説明

4.1. シェーダステージ

シェーダステージでは頂点シェーダおよびジオメトリシェーダに関する設定を行います。これらの設定を行う関数は nn::gd::ShaderStage クラスに定義されています。

以下の図はシェーダステージで行われる設定の概要を示したものです。

図 4-1. シェーダステージの概要



頂点シェーダおよびジオメトリシェーダは、事前にコンパイルされたシェーダバイナリファイルとしてアプリケーションに読み込まれます。シェーダバイナリファイルには、1 つ以上の頂点シェーダやジオメトリシェーダのシェーダプログラムを含むことができます。

シェーダバイナリファイルの中に記述されている特定のシェーダを参照するためには、読み込んだシェーダバイナリファイルを nn::gd::ShaderStage::CreateShaderBinary() に渡して作成された ShaderBinary オブジェクトを使用します。

コード 4-1. ShaderBinary オブジェクトの作成

```
static nnResult nn::gd::ShaderStage::CreateShaderBinary(
        const void* shaderBytecode, u32 bytecodeLength,
        nn::gd::ShaderBinary** shaderBinary);
```

shaderBytecode と *bytecodeLength* には、読み込んだシェーダバイナリのコードが格納されているバッファの先頭アドレスとコードのバイトサイズを指定します。

今後、作成された ShaderBinary オブジェクトへのアクセスには *shaderBinary* で受け取ったポインタを介して行います。

ShaderBinary オブジェクトに含まれているシェーダプログラムにアクセスするための Shader オブジェクトは nn::gd::ShaderStage::CreateShader() で作成します。Shader オブジェクトは 1 つのシェーダコード(頂点シェーダ、ジオメトリシェーダのいずれか)を表します。

コード 4-2. Shader オブジェクトの作成

```
static nnResult nn::gd::ShaderStage::CreateShader(
        nn::gd::ShaderBinary* shaderBinary, u32 shaderBinaryIndex,
        nn::gd::Shader** shader);
```

shaderBinary に指定されている ShaderBinary オブジェクトに複数のシェーダプログラムが含まれている場合は、シェーダプログラムのインデックスを *shaderBinaryIndex* に指定して Shader オブジェクトを作成してください。

今後、作成された Shader オブジェクトへのアクセスには *shader* で受け取ったポインタを介して行います。

シェーダプログラムをパイプラインに反映するために使用する ShaderPipeline オブジェクトは、1 つまたは 2 つの Shader オブジェクトを用いて、nn::gd::ShaderStage::CreateShaderPipeline() で作成されます。

コード 4-3. ShaderPipeline オブジェクトの作成とパイプラインへの反映

```
static nnResult nn::gd::ShaderStage::CreateShaderPipeline(
        nn::gd::Shader* vertexShader, nn::gd::Shader* geometryShader,
        nn::gd::ShaderPipeline** shaderPipeline,
        nn::gd::ShaderPipelineUnmanagedRegisters* unmanagedRegister = NULL);
static nnResult nn::gd::ShaderStage::SetShaderPipeline(
        nn::gd::ShaderPipeline* shaderPipeline);
```

vertexShader には、頂点シェーダとして使用する Shader オブジェクトを指定します。*geometryShader* には、ジオメトリシェーダとして使用する Shader オブジェクトを指定しますが、ジオメトリシェーダが不要である場合は NULL を渡してください。

unmanagedRegister には、レジスタ設定のコマンドをライブラリが自動生成しないようにするレジスタの範囲を指定します。この設定については「4.1.2. ライブラリによるレジスタへの設定コマンドの自動生成」で説明します。

> **注意:** GD ライブラリでは、頂点シェーダからジオメトリシェーダへと出力される頂点属性が正しいかどうかをチェックしません。頂点シェーダから出力される頂点属性とジオメトリシェーダの入力に使用する頂点属性が一致するように、正しい組み合わせの Shader オブジェクトで ShaderPipeline オブジェクトを作成してください。

以下のコード例では、頂点シェーダのみを使用する ShaderPipeline オブジェクトを生成してパイプラインに反映しています。

コード 4-4. シェーダステージで行われる設定のコード例

```
nn::gd::ShaderBinary* shaderBinary = 0;
nn::gd::ShaderPipeline* shaderPipeline = 0;
nn::gd::Shader* vertexShader = 0;
nn::gd::ShaderStage::CreateShaderBinary(
        shaderBinaryFileBuffer, bufferSize, &shaderBinary);
nn::gd::ShaderStage::CreateShader(shaderBinary, 0, &vertexShader);
nn::gd::ShaderStage::CreateShaderPipeline(vertexShader, NULL, &shaderPipeline);
nn::gd::ShaderStage::SetShaderPipeline(shaderPipeline);
```

4.1.1. シェーダ設定レジスタへの設定

浮動小数点定数レジスタなどのシェーダ設定レジスタへ設定される値は ShaderPipeline オブジェクトが保持しています。GD ライブラリでは、シェーダ設定レジスタへの設定値をシェーダステージの関数で変更することができます。関数で更新された変数は一旦内部状態としてオブジェクトに保存され、描画コマンドが要求されるまで 3D コマンドバッファに出力されません。

レジスタの種類によって、値の設定に使用する関数は以下のように異なります。

表 4-1. レジスタの種類と呼び出す関数

レジスタの種類	関数
浮動小数点定数レジスタ	nn::gd::ShaderStage::SetShaderPipelineConstantFloat()
ブールレジスタ	nn::gd::ShaderStage::SetShaderPipelineConstantBoolean()
整数レジスタ	nn::gd::ShaderStage::SetShaderPipelineConstantInteger()

コード 4-5. シェーダ設定レジスタへの値の設定に使用する関数

```
static nnResult nn::gd::ShaderStage::SetShaderPipelineConstantFloat(
        nn::gd::ShaderPipeline* shaderPipeline,
        nn::gd::UniformLocation uniformLocation, f32* v);
static nnResult nn::gd::ShaderStage::SetShaderPipelineConstantBoolean(
        nn::gd::ShaderPipeline* shaderPipeline,
        nn::gd::UniformLocation uniformLocation, u16 v, u32 count);
static nnResult nn::gd::ShaderStage::SetShaderPipelineConstantInteger(
        nn::gd::ShaderPipeline* shaderPipeline,
        nn::gd::UniformLocation uniformLocation, u8* v);
static nn::gd::UniformLocation nn::gd::ShaderStage::GetShaderUniformLocation(
        nn::gd::ShaderPipeline* shaderPipeline, const char* name);
```

設定値を変更するレジスタの指定には nn::gd::ShaderStage::GetShaderUniformLocation() で取得した UniformLocation クラスを使用します。取得する際の *name* にはユニフォーム設定の名前を指定してください。正しいユニフォーム設定が取得できている場合、UniformLocation クラスの IsValid() は 1 を返します。

4.1.2. ライブラリによるレジスタへの設定コマンドの自動生成

通常、現在反映されているものとは別の ShaderPipeline オブジェクトをパイプラインに反映させたときには、レジスタへの設定コマンドが自動的に生成されます。しかし、場合によっては自動的に設定コマンドを生成せずに、直接レジスタに値を書き込むコマンドを生成することでパフォーマンスが向上するケースがあります。

そのようなケースを考慮して、nn::gd::ShaderStage::SetFloatConstantBuffer() で独自にレジスタの設定値を変更することができます。

コード 4-6. 独自にレジスタの設定値を変更する関数

```
static nnResult nn::gd::ShaderStage::SetFloatConstantBuffer(
        nn::gd::ShaderStage::ShaderType shaderType,
        u32 firstRegisterIndex, u32 registerCount, f32* constantBufferSrc);
```

この関数で設定値が変更できるのは、自動的に設定コマンドを生成しない(「unmanaged」な)浮動小数点定数レジスタに限られています。「unmanaged」なレジスタの範囲は ShaderPipeline オブジェクトの作成時の引数

unmanagedRegister に渡す ShaderPipelineUnmanagedRegisters クラスで指定します。

ShaderPipelineUnmanagedRegisters クラスは UnmanagedRegistersInterval クラスの配列とその要素数で構成され、UnmanagedRegistersInterval クラスはシェーダの種類(m_ShaderType)と「unmanaged」なレジスタの最初の番号(m_FirstUnmanagedRegister)と個数(m_RegisterCount)で構成されています。

コード 4-7.「unmanaged」なレジスタの指定で使用するクラス

```
class nn::gd::UnmanagedRegistersInterval
{
    nn::gd::ShaderStage::ShaderType m_ShaderType;
    u32 m_FirstUnmanagedRegister;
    u32 m_RegisterCount;
};
class nn::gd::ShaderPipelineUnmanagedRegisters
{
    nn::gd::UnmanagedRegistersInterval* m_ArrayUnmanagedRegistersInterval;
    u32 m_CountUnmanagedRegistersInterval;
};
```

以下のコード例は「unmanaged」なレジスタを含む ShaderPipeline クラスの作成例です。

コード 4-8.「unmanaged」なレジスタを含む ShaderPipeline クラスの作成例

```
nn::gd::ShaderPipelineUnmanagedRegisters unmanagedRegs;
nn::gd::UnmanagedRegistersInterval arrayUnmanagedRegistersInterval[] =
{
    {nn::gd::ShaderStage::SHADER_TYPE_VERTEX, 0, 4},
    {nn::gd::ShaderStage::SHADER_TYPE_VERTEX, 4, 3},
    {nn::gd::ShaderStage::SHADER_TYPE_VERTEX, 8, 3}
};

unmanagedRegs.m_ArrayUnmanagedRegistersInterval =
        arrayUnmanagedRegistersInterval;
unmanagedRegs.m_CountUnmanagedRegistersInterval = 3;
nn::gd::ShaderStage::CreateShaderPipeline(
        s_vertexShader, NULL, &s_shaderPipeline, &unmanagedRegs);
```

「unmanaged」なレジスタには、以下のようなコードで直接書き込むことになります。

コード 4-9.「unmanaged」なレジスタへの書き込み

```
nn::gd::UniformLocation s_shaderVariable_proj =
    nn::gd::ShaderStage::GetShaderUniformLocation(
        s_shaderPipeline, "uProjection");

nn::gd::ShaderStage::SetFloatConstantBuffer(
    nn::gd::ShaderStage::SHADER_TYPE_VERTEX,
        s_shaderVariable_proj.getRegister(), 4, data);
```

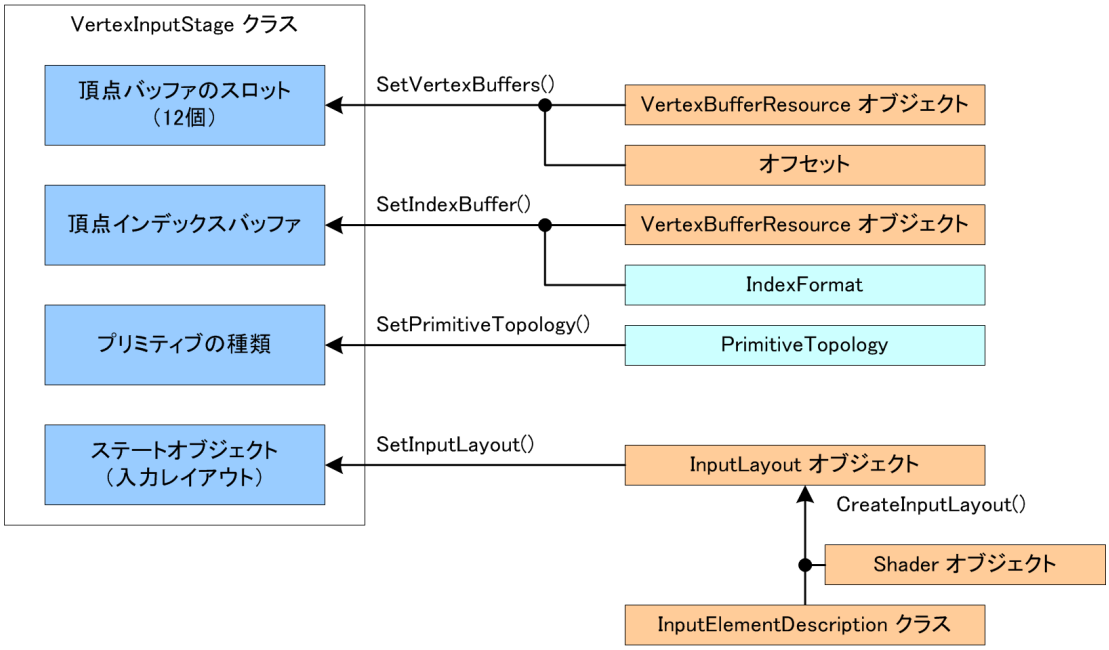
新たに反映された ShaderPipeline オブジェクトが、現在反映されていたものとは異なる ShaderBinary オブジェクトから作成されたものである場合、描画の際に ShaderBinary オブジェクトに含まれるすべてのデータが 3D コマンドバッファに出力されます。この処理には負荷がかかるため、「unmanaged」なレジスタを利用して、出力される 3D コマンドを最小限に止めるようにすることが推奨されます。

4.2. 頂点入力ステージ

頂点入力ステージではパイプラインへの頂点データ(頂点属性)の入力に関する設定を行います。これらの設定を行うための関数は nn::gd::VertexInputStage クラスに定義されています。

以下の図は頂点入力ステージで行われる設定の概要を示したものです。

図 4-2. 頂点入力ステージの概要



4.2.1. 頂点バッファの登録

頂点バッファは頂点入力ステージに 12 個あるスロットに登録します。

nn::gd::VertexInputStage::SetVertexBuffers() では VertexBufferResource オブジェクトの配列と頂点データの開始オフセットの配列を指定しますので、一度の呼び出しで連続する複数のスロットに頂点バッファを登録することができます。

コード 4-10. 頂点バッファの登録

```
static nnResult nn::gd::VertexInputStage::SetVertexBuffers(
        u32 startSlot, u32 numBuffers,
        nn::gd::VertexBufferResource** vertexBuffers, u32* offsets);
```

startSlot には頂点バッファを登録する最初のスロット番号(0～11)を、*numBuffers* には *vertexBuffers* に指定された VertexBufferResource オブジェクト配列の要素数を、*offsets* には頂点データの開始オフセットの配列を指定します。登録される頂点バッファを構成する頂点データの要素は、「4.2.4. 入力レイアウト」で説明する InputLayout オブジェクトでの定義と一致している必要があります。

4.2.2. 頂点インデックスの使用

プリミティブの描画に頂点インデックスを使用する場合は、頂点インデックスの配列をリソースデータに指定して作成された VertexBufferResource オブジェクトを nn::gd::VertexInputStage::SetIndexBuffer() で頂点入力ステージに登録します。

コード 4-11. 頂点インデックスの登録

```
static nnResult nn::gd::VertexInputStage::SetIndexBuffer(
        nn::gd::VertexBufferResource* indexBuffer,
        nn::gd::VertexInputStage::IndexFormat format, u32 offset);
```

indexBuffer に指定されている VertexBufferResource オブジェクトが頂点入力ステージに登録されます。頂点データなどとインターリーブされている VertexBufferResource オブジェクトを登録する場合は、頂点インデックスの開始位置をバイト単位のオフセット値を *offset* に指定します。

format には、頂点インデックスのフォーマットを以下から選択して指定します。

表 4-2. 頂点インデックスのフォーマット指定

定義	説明
INDEX_FORMAT_UBYTE	頂点インデックスを unsigned byte の配列として扱います。
INDEX_FORMAT_USHORT	頂点インデックスを unsigned short の配列として扱います。

4.2.3. プリミティブの種類

描画するプリミティブの種類は nn::gd::VertexInputStage::SetPrimitiveTopology() で設定します。

コード 4-12. 描画するプリミティブの種類の設定

```
static nnResult nn::gd::VertexInputStage::SetPrimitiveTopology(
        nn::gd::VertexInputStage::PrimitiveTopology primitiveTopology);
```

primitiveTopology には、プリミティブの種類を以下から選択して指定します。

表 4-3. プリミティブの種類

定義	説明
PRIMITIVE_TOPOLOGY_TRIANGLELIST	Triangles(GL_TRIANGLES)
PRIMITIVE_TOPOLOGY_TRIANGLESTRIP	Triangle Strip(GL_TRIANGLE_STRIP)
PRIMITIVE_TOPOLOGY_TRIANGLEFAN	Triangle Fan(GL_TRIANGLE_FAN)
PRIMITIVE_TOPOLOGY_GEOMETRY	ジオメトリシェーダ(GL_GEOMETRY_PRIMITIVE_DMP)

4.2.4. 入力レイアウト

入力レイアウト(InputLayout オブジェクト)には、頂点バッファと頂点シェーダの間での入力変数の接続方法を定義します。頂点バッファは単純なスカラ値またはインターリーブされたいくつかのスカラ値の配列によって構成されています。InputLayout オブジェクトは、入力要素(頂点データ)のデスクリプタクラス(InputElementDescription)の配列とシェーダステージで作成した Shader オブジェクトを引数に持つ、nn::gd::VertexInputStage::CreateInputLayout() の呼び出しで作成します。

コード 4-13. 入力レイアウトの作成、解放、パイプラインへの設定

```
static nnResult nn::gd::VertexInputStage::CreateInputLayout(
        nn::gd::InputElementDescription* inputElementDescs, u32 numElements,
        u32* strides, nn::gd::Shader* vertexShader,
        nn::gd::InputLayout** inputLayout);
static nnResult nn::gd::VertexInputStage::ReleaseInputLayout(
        nn::gd::InputLayout* inputLayout);
static nnResult nn::gd::VertexInputStage::SetInputLayout(
        nn::gd::InputLayout* inputLayout);
```

入力要素の情報は *inputElementDescs* に指定するデスクリプタクラス(InputElementDescription)の配列にあらかじめ設定し、*numElements* には配列に含まれているデスクリプタクラスの数を渡します。デスクリプタクラスに設定する情報の詳細は「4.2.4.1. InputLayout オブジェクトのデスクリプタクラス」を参照してください。

strides には、1 頂点あたりに使用する頂点データのサイズをスロット番号ごとの配列で指定します。すべての入力要素が 1 つの頂点バッファにインターリーブされているならば、その入力要素の合計サイズで 1 つの要素だけを持つ配列となります。インターリーブされていなければ、それぞれの入力要素のサイズで複数の要素を持つ配列となります。

vertexShader には、頂点シェーダの Shader オブジェクトを指定します。

作成された InputLayout オブジェクトは nn::gd::VertexInputStage::SetInputLayout() でパイプラインに反映することができます。

不要になった InputLayout オブジェクトは nn::gd::VertexInputStage::ReleaseInputLayout() で解放してください。このとき、オブジェクトが作成されたときに内部で確保されたメモリ領域も解放されます。

以下のコード例は、3 つの入力要素がインターリーブされた 1 つの頂点バッファを記述するデスクリプタクラスから作成された入力レイアウトをパイプライン(頂点入力ステージ)に設定するものです。

コード 4-14. 入力レイアウトの作成例

```
nn::gd::InputLayout* inputLayout = 0;
nn::gd::InputElementDescription descs[] =
{
    { 0, "aPosition",
        nn::gd::VertexInputStage::STREAM_TYPE_FLOAT, 3, 0 },
    { 0, "aColor",
        nn::gd::VertexInputStage::STREAM_TYPE_FLOAT, 4, sizeof(float) * 3 },
    { 0, "aNormal",
        nn::gd::VertexInputStage::STREAM_TYPE_FLOAT, 3, sizeof(float) * 7 },
};
u32 strides[] = { sizeof(float) * 10 };
nn::gd::VertexInputStage::CreateInputLayout(
        descs, 3, strides, vertexShader, &inputLayout);

nn::gd::VertexInputStage::SetInputLayout(inputLayout);
```

4.2.4.1. InputLayout オブジェクトのデスクリプタクラス

InputLayout オブジェクトのデスクリプタクラス(InputElementDescription)のメンバを紹介し、設定に際しての注意点などを説明します。

コード 4-15. InputElementDescription クラスの定義

```
class nn::gd::InputElementDescription
{
    u32 m_StreamSlot;
    const char* m_SemanticName;
    nn::gd::VertexInputStage::ShaderStreamFormatType m_Format;
    u32 m_Count;
    u32 m_AlignedByteOffset;

    static const u32 Append = 0x00080000;
};
```

スロット番号(m_StreamSlot)

入力要素が使用するスロット番号(0～11)です。複数の入力要素が同じスロット番号を使用する場合、そのスロット番号に登録される頂点バッファはインターリーブされていなければなりません。また、スロット番号は 0 から昇順に、連番で登録されていなければなりません。

入力変数の名前(m_SemanticName)

頂点シェーダで定義されている入力変数の名前です。

入力要素のデータフォーマット(m_Format)

入力要素のデータフォーマットです。以下から選択して設定します。

表 4-4. 入力要素のデータフォーマット

定義	説明
VertexInputStage::STREAM_TYPE_BYTE	byte(GL_BYTE)
VertexInputStage::STREAM_TYPE_UNSIGNED_BYTE	unsigned byte(GL_UNSIGNED_BYTE)
VertexInputStage::STREAM_TYPE_SHORT	short(GL_SHORT)
VertexInputStage::STREAM_TYPE_FLOAT	float(GL_FLOAT)

入力要素のサイズ(m_Count)

入力要素のサイズ(コンポーネント数)です。設定可能な値の範囲は 1 ～ 4 です。入力要素のバイトサイズではないことに注意してください。

入力要素のオフセット(m_AlignedByteOffset)

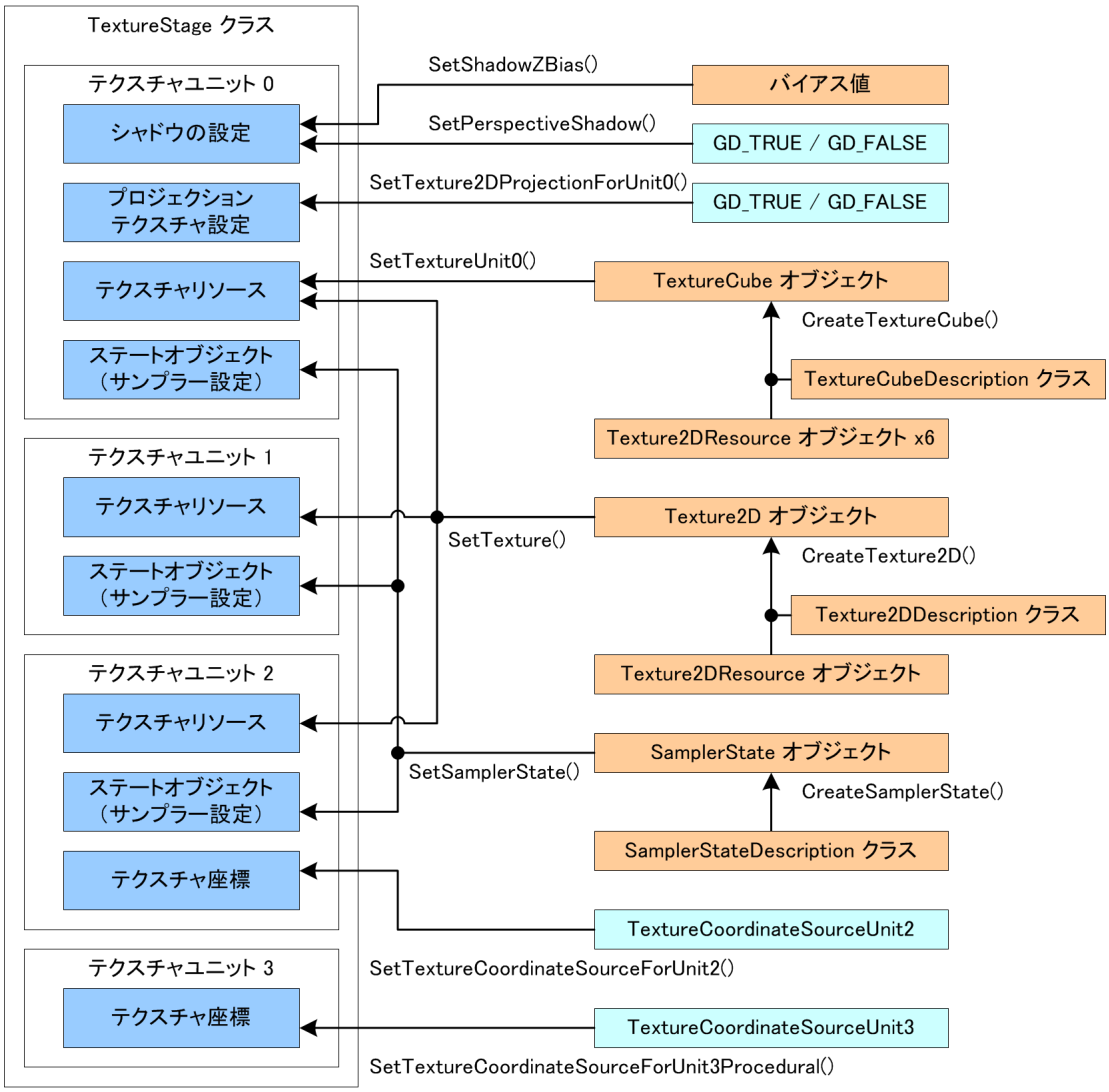
入力要素の先頭までのオフセット(バイト単位)です。InputElementDescription::Append を指定した場合はオフセットが自動的に計算され、デスクリプタクラスの設定を簡便化することができます。

4.3. テクスチャステージ

テクスチャステージでは、テクスチャオブジェクトの作成やテクスチャユニットに関する設定を行います。これらの設定を行うための関数は nn::gd::TextureStage クラスに定義されています。

以下の図はテクスチャステージで行われる設定の概要を示したものです。

図 4-3. テクスチャステージの概要



4.3.1. 2 次元テクスチャ

GD ライブラリでは Texture2D オブジェクトで 2 次元テクスチャを扱います。Texture2D オブジェクトは 1 つの Texture2DResource オブジェクトとデスクリプタクラス(Texture2DDescription)から作成されます。なお、デスクリプタクラスの設定によって、Texture2DResource オブジェクトがミップマップデータを含んでいる場合に、Texture2D オブジェクトの作成にどのレベルのミップマップデータを使用するかを指定することができます。デスクリプタクラスに設定する情報の詳細は「4.3.1.1. Texture2D オブジェクトのデスクリプタクラス」を参照してください。

Texture2D オブジェクトの作成と解放は以下の関数で行うことができます。

コード 4-16. Texture2D オブジェクトの作成と解放

```
static nnResult nn::gd::TextureStage::CreateTexture2D(
        const nn::gd::Texture2DResource* tex2DResource,
        nn::gd::Texture2DDescription* desc,
        nn::gd::Texture2D** texture2D);
static nnResult nn::gd::TextureStage::ReleaseTexture2D(
        nn::gd::Texture2D* texture2D);
```

Texture2D オブジェクトの解放時には管理情報のみが解放されます。Texture2DResource オブジェクトは解放されま

せん。

以下は Texture2DResource オブジェクトを作成し、そこから Texture2D オブジェクトを作成するコードの例です。

コード 4-17. Texture2D オブジェクト作成のコード例

```
nn::gd::Texture2DResource* texture2DResource = 0;
nn::gd::Texture2DResourceDescription Text2DResDesc =
    {256, 256, 1, nn::gd::Resource::NATIVE_FORMAT_RGB_888,
    nn::gd::Memory::LAYOUT_BLOCK_8, nn::gd::Memory::FCRAM};
nn::gd::Resource::CreateTexture2DResource(
        &Text2DResDesc, data, GD_FALSE, &texture2DResource);
nn::gd::Texture2D* texture2D = 0;
nn::gd::Texture2DDescription tex2DDesc = {0, 0};
nn::gd::TextureStage::CreateTexture2D(
        texture2DResource, &tex2DDesc, &texture2D);
```

4.3.1.1. Texture2D オブジェクトのデスクリプタクラス

Texture2D オブジェクトのデスクリプタクラス(Texture2DDescription)のメンバを紹介し、設定に際しての注意点などを説明します。

コード 4-18. Texture2DDescription クラスの定義

```
class nn::gd::Texture2DDescription
{
    s32 m_MinMipLevelIndex;
    s32 m_MaxMipLevelIndex;
};
```

最小ミップマップレベルのインデックス(m_MinMipLevelIndex)

Texture2D オブジェクトの最小ミップマップレベルに使用する Texture2DResource オブジェクトのミップマップレベルです。-1 を設定した場合は Texture2DResource オブジェクトの最大のミップマップレベルが使用されます。

最大ミップマップレベルのインデックス(m_MaxMipLevelIndex)

Texture2D オブジェクトの最大ミップマップレベルに使用する Texture2DResource オブジェクトのミップマップレベルです。-1 を設定した場合は Texture2DResource オブジェクトの最大のミップマップレベルが使用されます。

4.3.1.2. Texture2D オブジェクトの詳細情報の取得

詳細情報(Texture2DProperties クラス)を nn::gd::TextureStage::GetTexture2DProperties() で取得することができます。

コード 4-19. Texture2D オブジェクトの詳細情報の取得

```
static nnResult nn::gd::TextureStage::GetTexture2DProperties(
        const nn::gd::Texture2D* texture2D,
        nn::gd::Texture2DProperties* properties);
```

オブジェクトの作成時に指定したミップマップレベルによって、幅や高さ、ミップマップレベル、リソースデータの先頭アドレスが変化することはありますが、基本的に Texture2DResource オブジェクトの詳細情報と同じです。

コード 4-20. Texture2DProperties クラスの定義

```
class nn::gd::Texture2DProperties
{
    u32 m_Width;
    u32 m_Height;
    u32 m_CountMipLevels;
    u32 m_PixelSize;
    nn::gd::Resource::NativeFormat m_Format;
    nn::gd::Memory::MemoryLayout m_MemLayout;
    nn::gd::Memory::MemoryLocation m_MemLocation;
    u8* m_MemAddr;
    nn::gd::MipmapResourceInfo GetMipmapAddress(u32 mipmapLevel);
};
```

4.3.2. キューブマップテクスチャ

GD ライブラリでは TextureCube オブジェクトでキューブマップテクスチャを扱います。TextureCube オブジェクトは 6 つの Texture2DResource オブジェクトを用いて作成されます。Texture2D オブジェクトと同様に、どのミップマップレベルを使用するかを指定することができます。すべての面の Texture2DResource オブジェクトは同じフォーマット、同じ解像度(ミップマップによる)、同じミップマップレベル、同じメモリ配置場所を満たしている必要があります。

TextureCube オブジェクトの作成と解放は以下の関数で行うことができます。

コード 4-21. TextureCube オブジェクトの作成と解放

```
static nnResult nn::gd::TextureStage::CreateTextureCube(
        nn::gd::Texture2DResource** tex2DResources,
        nn::gd::TextureCubeDescription* desc,
        nn::gd::TextureCube** textureCube);
static nnResult nn::gd::TextureStage::ReleaseTextureCube(
        nn::gd::TextureCube* textureCube);
```

tex2DResources には、6 面分の Texture2DResource オブジェクトの配列を指定します。配列の先頭から、+X、-X、+Y、-Y、+Z、-Z 面のテクスチャに使用されます。デスクリプタクラスでの配列の順序やキューブマップテクスチャの面を指定する際も同じです。

TextureCube オブジェクトの解放時には管理情報のみが解放されます。Texture2DResource オブジェクトは解放されません。

4.3.2.1. TextureCube オブジェクトのデスクリプタクラス

TextureCube オブジェクトのデスクリプタクラス(TextureCubeDescription)のメンバを紹介し、設定に際しての注意点などを説明します。

コード 4-22. TextureCubeDescription クラスの定義

```
class nn::gd::TextureCubeDescription
{
    s32 m_MinMipLevelIndex[6];
    s32 m_MaxMipLevelIndex[6];
    TextureCubeDescription();
    TextureCubeDescription(int minMipLevelIndex, int maxMipLevelIndex);
};
```

6 面分のミップマップレベルを指定することや、それを簡便化するためのコンストラクタが用意されている点が Texture2DDescription クラスとは異なります。

最小ミップマップレベルのインデックス(m_MinMipLevelIndex)

TextureCube オブジェクトの最小ミップマップレベルに使用する Texture2DResource オブジェクトのミップマップレベルです。-1 を設定した場合は Texture2DResource オブジェクトの最大のミップマップレベルが使用されます。

配列の先頭から、+X、-X、+Y、-Y、+Z、-Z 面の Texture2DResource オブジェクトに対応しています。

最大ミップマップレベルのインデックス(m_MaxMipLevelIndex)

TextureCube オブジェクトの最大ミップマップレベルに使用する Texture2DResource オブジェクトのミップマップレベルです。-1 を設定した場合は Texture2DResource オブジェクトの最大のミップマップレベルが使用されます。

配列の先頭から、+X、-X、+Y、-Y、+Z、-Z 面の Texture2DResource オブジェクトに対応しています。

4.3.2.2. TextureCube オブジェクトの詳細情報の取得

TextureCube オブジェクトの詳細情報(TextureCubeProperties クラス)を nn::gd::TextureStage::GetTextureCubeProperties() で取得することができます。

コード 4-23. TextureCube オブジェクトの詳細情報の取得

```
static nnResult nn::gd::TextureStage::GetTextureCubeProperties(
        const nn::gd::TextureCube* TextureCube,
        nn::gd::TextureCubeProperties* properties);
```

6 面分のリソースデータの先頭アドレスが保持されていることや、ミップマップデータのアドレス情報の取得で *faceIndex* にキューブマップ面を指定すること以外は Texture2DProperties クラスと同じです。

コード 4-24. TextureCubeProperties クラスの定義

```
class nn::gd::TextureCubeProperties
{
    u32 m_Width;
    u32 m_Height;
    u32 m_CountMipLevels;
    u32 m_PixelSize;
    nn::gd::Resource::NativeFormat m_Format;
    nn::gd::Memory::MemoryLayout m_MemLayout;
    nn::gd::Memory::MemoryLocation m_MemLocation;
    u8* m_MemAddr[6];

    nn::gd::MipmapResourceInfo GetMipmapAddress(u32 faceIndex, u32 mipmapLevel);
};
```

4.3.3. テクスチャユニット

GD ライブラリでは、3DS のハードウェア構成に合わせて 4 つの論理的なテクスチャユニットを提供しています。

表 4-5. テクスチャユニットと設定可能なテクスチャの種類

定義	設定可能なテクスチャの種類
TextureStage::TEXTURE_UNIT_0 (テクスチャユニット 0)	2 次元テクスチャ(ガス、シャドウ含む) キューブマップテクスチャ(シャドウ含む) プロジェクションテクスチャ
TextureStage::TEXTURE_UNIT_1 (テクスチャユニット 1)	2 次元テクスチャ
TextureStage::TEXTURE_UNIT_2 (テクスチャユニット 2)	2 次元テクスチャ
TextureStage::TEXTURE_UNIT_3_PROCEDURAL (テクスチャユニット 3)	プロシージャルテクスチャ

補足: テクスチャ座標以外の TextureStage::TEXTURE_UNIT_3_PROCEDURAL への設定は、プロシージャルテクスチャステージで行います。

2 次元テクスチャ

2 次元テクスチャの設定は nn::gd::TextureStage::SetTexture() で行います。設定対象となるテクスチャユニットはテクスチャユニット 0/1/2 の 3 つです。

コード 4-25. テクスチャユニットへの 2 次元テクスチャの設定

```
static nnResult nn::gd::TextureStage::SetTexture(
        nn::gd::TextureStage::TextureUnitId textureUnitId,
        nn::gd::Texture2D& texture2D);
```

そのほかのテクスチャ(プロシージャルテクスチャを除く)

そのほかのテクスチャ(プロシージャルテクスチャを除く)はテクスチャユニット 0 にのみ設定することができます。

コード 4-26. そのほかのテクスチャの設定

```
static nnResult nn::gd::TextureStage::SetTextureUnit0(
        nn::gd::TextureCube& textureCube);
static void nn::gd::TextureStage::SetTexture2DProjectionForUnit0(gdBool value);
static void nn::gd::TextureStage::SetPerspectiveShadow(gdBool v);
static void nn::gd::TextureStage::SetShadowZBias(f32 zBias);
```

キューブマップテクスチャの設定は nn::gd::TextureStage::SetTextureUnit0() で行います。

nn::gd::TextureStage::SetTexture2DProjectionForUnit0() の引数に GD_TRUE を渡して呼び出すことで、テクスチャユニット 0 に設定された 2 次元テクスチャをプロジェクションテクスチャとして扱うことができます。

シャドウテクスチャまたはシャドウキューブマップテクスチャを参照する際のテクスチャ座標の生成に透視投影を適用する場合は、nn::gd::TextureStage::SetPerspectiveShadow() の引数に GD_TRUE を渡して呼び出してください。また、nn::gd::TextureStage::SetShadowZBias() では、シャドウテクスチャの生成時に光源までの距離から減算されるバイアス値を設定することができます。

4.3.3.1. テクスチャ座標

テクスチャユニット 2/3 に入力されるテクスチャ座標の設定は以下の関数で行われます。

コード 4-27. テクスチャ座標の設定

```
// For TextureUnit2
static void nn::gd::TextureStage::SetTextureCoordinateSourceForUnit2(
        nn::gd::TextureStage::TextureCoordinateSourceUnit2 u);
// For TextureUnit3
static void nn::gd::TextureStage::SetTextureCoordinateSourceForUnit3Procedural(
        nn::gd::TextureStage::TextureCoordinateSourceUnit3Procedural u);
```

それぞれの関数には、以下の引数が指定可能です。

表 4-6. テクスチャユニット 2 に設定可能なテクスチャ座標

定義	入力されるテクスチャ座標
UNIT2_TEXTURE_COORDINATE_FROM_UNIT_2	テクスチャ座標 2(デフォルト)
UNIT2_TEXTURE_COORDINATE_FROM_UNIT_1	テクスチャ座標 1

表 4-7. テクスチャユニット 3 に設定可能なテクスチャ座標

定義	入力されるテクスチャ座標
UNIT3_PROCEDURAL_TEXTURE_COORDINATE_FROM_UNIT_0	テクスチャ座標 0(デフォルト)
UNIT3_PROCEDURAL_TEXTURE_COORDINATE_FROM_UNIT_1	テクスチャ座標 1
UNIT3_PROCEDURAL_TEXTURE_COORDINATE_FROM_UNIT_2	テクスチャ座標 2

4.3.4. サンプラー設定

テクスチャのサンプリング方法の設定(サンプラー設定)はステートオブジェクト(SamplerState オブジェクト)を作成し、それをテクスチャユニットに設定することで行われます。

コード 4-28. サンプラー設定の作成、解放、テクスチャユニットへの設定

```
static nnResult nn::gd::TextureStage::CreateSamplerState(
        const nn::gd::SamplerStateDescription* desc,
        nn::gd::SamplerState** sampler);
static nnResult nn::gd::TextureStage::ReleaseSamplerState(
        nn::gd::SamplerState* sampler);
static nnResult nn::gd::TextureStage::SetSamplerState(
        nn::gd::TextureStage::TextureUnitId textureUnitId,
        nn::gd::SamplerState* sampler);
```

サンプラー設定の情報は *desc* に指定するデスクリプタクラス(SamplerStateDescription)にあらかじめ設定します。デスクリプタクラスに設定する情報の詳細は「4.3.4.1. SamplerState オブジェクトのデスクリプタクラス」を参照してください。

作成された SamplerState オブジェクトは nn::gd::TextureStage::SetSamplerState() でテクスチャユニットに設定することができます。サンプラー設定が可能なテクスチャユニットは、テクスチャユニット 0/1/2 の 3 つです。

不要になった SamplerState オブジェクトは nn::gd::TextureStage::ReleaseSamplerState() で解放してください。このとき、オブジェクトが作成されたときに内部で確保されたメモリ領域も解放されます。

以下のコード例では、デフォルトの設定から縮小時と拡大時のフィルタを変更した SamplerState オブジェクトを作成し、テクスチャユニット 0 への設定を行っています。

コード 4-29. サンプラー設定のコード例

```
nn::gd::SamplerState* sampler = 0;
nn::gd::SamplerStateDescription samplerdesc;
samplerdesc.ToDefault();
samplerdesc.m_MinFilter = nn::gd::TextureStage::SAMPLER_MIN_FILTER_LINEAR;
samplerdesc.m_MagFilter = nn::gd::TextureStage::SAMPLER_MAG_FILTER_LINEAR;
nn::gd::TextureStage::CreateSamplerState(&samplerdesc, &sampler);
nn::gd::TextureStage::SetSamplerState(
        nn::gd::TextureStage::TEXTURE_UNIT_0, sampler);
```

4.3.4.1. SamplerState オブジェクトのデスクリプタクラス

SamplerState オブジェクトのデスクリプタクラス(SamplerStateDescription クラス)のメンバを紹介し、設定に際しての注意点などを説明します。

コード 4-30. SamplerStateDescription クラスの定義

```
class nn::gd::SamplerStateDescription
{
    nn::gd::TextureStage::SamplerMinFilter m_MinFilter;
    nn::gd::TextureStage::SamplerMagFilter m_MagFilter;
    nn::gd::TextureStage::SamplerWrapMode m_WrapS;
    nn::gd::TextureStage::SamplerWrapMode m_WrapT;
    u8 m_BorderColor[4];
    u32 m_LodBias;
    u32 m_MinLod;
    u32 m_MaxLod;

    void SetMinFilter(nn::gd::TextureStage::SamplerMinFilter filter);
    void SetMagFilter(nn::gd::TextureStage::SamplerMagFilter filter);
    void SetWrapS(nn::gd::TextureStage::SamplerWrapMode wrap);
    void SetWrapT(nn::gd::TextureStage::SamplerWrapMode wrap);
    void SetBorderColor(u8 colorR, u8 colorG, u8 colorB, u8 colorA);
    void SetBorderColor(f32 colorR, f32 colorG, f32 colorB, f32 colorA);
    void SetLodBias(f32 biasValue);
    void SetMinLod(u32 lodValue);
    void SetMaxLod(u32 lodValue);

    SamplerStateDescription();
    void ToDefault();
    void SetShadow();
    void SetShadowCube();
    void SetGas();
};
```

縮小時のフィルタ(m_MinFilter)

テクスチャが縮小表示される際に適用されるフィルタです。SetMinFilter() で設定することができます。

拡大時のフィルタ(m_MagFilter)

テクスチャが拡大表示される際に適用されるフィルタです。SetMagFilter() で設定することができます。

S 方向のラッピングモード(m_WrapS)

S 方向へのテクスチャの繰り返しで適用されるラッピングモードです。SetWrapS() で設定することができます。

T 方向のラッピングモード(m_WrapT)

T 方向へのテクスチャの繰り返しで適用されるラッピングモードです。SetWrapT() で設定することができます。

ボーダーカラー(m_BorderColor)

ラッピングモードに TextureStage::SAMPLER_WRAP_CLAMP_TO_BORDER を指定したときに使用されるボーダーカラーです。SetBorderColor() で RGBA の各成分を 0～255 または 0.0～1.0 の範囲で設定することができます。

LOD のバイアス値(m_LodBias)

LOD を決定する際に使用されるバイアス値です。GPU のレジスタに設定する際のフォーマットで値を設定しなければなりませんので、-16.0 ～ 16.0 の範囲の値を SetLodBias() に渡して設定してください。

バイアス値のフォーマットは小数部 8 ビットの符号つき 13 ビット固定小数点数です。直接メンバ変数を変更する場合は、ユーティリティ関数の Utils::Float32ToFix13Fraction8() を利用してください。

LOD の最小レベル(m_MinLod)

LOD で使用する最小のミップマップレベルです。LOD を使用する場合は最小のミップマップレベルを 0 以上(使用しない場合は 0)で設定してください。SetMinLod() で設定することができます。

LOD の最大レベル(m_MaxLod)

LOD で使用する最大のミップマップレベルです。LOD を使用しない場合は 0 を、使用する場合は(テクスチャの持つ最大のミップマップレベル - 1)を設定してください。SetMaxLod() で設定することができます。

異なるミップマップレベル数を持つテクスチャで同じサンプラー設定を使用する場合は UINT_MAX などの大きな値を設定してください。描画時に(テクスチャの持つミップマップレベル数 – 1)が設定されます。デフォルトの設定やソースから判断しています。

デフォルトの設定(ToDefault())

ToDefault() で、すべてのメンバをデフォルトの値に変更することができます。この設定はコンストラクタでも行われています。

表 4-8. デフォルトのサンプラー設定

メンバ	設定値
m_MinFilter	TextureStage::SAMPLER_MIN_FILTER_NEAREST
m_MagFilter	TextureStage::SAMPLER_MAG_FILTER_NEAREST
m_WrapS	TextureStage::SAMPLER_WRAP_REPEAT
m_WrapT	TextureStage::SAMPLER_WRAP_REPEAT
m_BorderColor	(0, 0, 0, 0)

m_LodBias	0.0
m_MinLod	0
m_MaxLod	0xFFFFFFFF

シャドウテクスチャ用の設定(SetShadow())

SetShadow() で、すべてのメンバをシャドウテクスチャ用の値に変更することができます。

表 4-9. シャドウテクスチャ用のサンプラー設定

メンバ	設定値
m_MinFilter	TextureStage::SAMPLER_MIN_FILTER_LINEAR
m_MagFilter	TextureStage::SAMPLER_MAG_FILTER_LINEAR
m_WrapS	TextureStage::SAMPLER_WRAP_CLAMP_TO_BORDER
m_WrapT	TextureStage::SAMPLER_WRAP_CLAMP_TO_BORDER
m_BorderColor	(0, 0, 0, 0)
m_LodBias	0.0
m_MinLod	0
m_MaxLod	0xFFFFFFFF

注意: フィルタ、ラッピングモード、LOD の設定を変更しないでください。

シャドウキューブテクスチャ用の設定(SetShadowCube())

SetShadowCube() で、すべてのメンバをシャドウキューブマップテクスチャ用の値に変更することができます。

表 4-10. シャドウキューブマップテクスチャ用のサンプラー設定

メンバ	設定値
m_MinFilter	TextureStage::SAMPLER_MIN_FILTER_LINEAR
m_MagFilter	TextureStage::SAMPLER_MIN_FILTER_LINEAR
m_WrapS	TextureStage::SAMPLER_WRAP_CLAMP_TO_EDGE
m_WrapT	TextureStage::SAMPLER_WRAP_CLAMP_TO_EDGE
m_BorderColor	(0, 0, 0, 0)
m_LodBias	0.0
m_MinLod	0
m_MaxLod	0xFFFFFFFF

注意: フィルタ、ラッピングモード、LOD の設定を変更しないでください。

ガステクスチャ用の設定(SetGas())

SetGas() で、すべてのメンバをガステクスチャ用の値に変更することができます。

表 4-11. ガステクスチャ用のサンプラー設定

メンバ	設定値
m_MinFilter	TextureStage::SAMPLER_MIN_FILTER_NEAREST
m_MagFilter	TextureStage::SAMPLER_MAG_FILTER_NEAREST
m_WrapS	TextureStage::SAMPLER_WRAP_CLAMP_TO_EDGE
m_WrapT	TextureStage::SAMPLER_WRAP_CLAMP_TO_EDGE
m_BorderColor	(0, 0, 0, 0)
m_LodBias	0.0
m_MinLod	0
m_MaxLod	0xFFFFFFFF

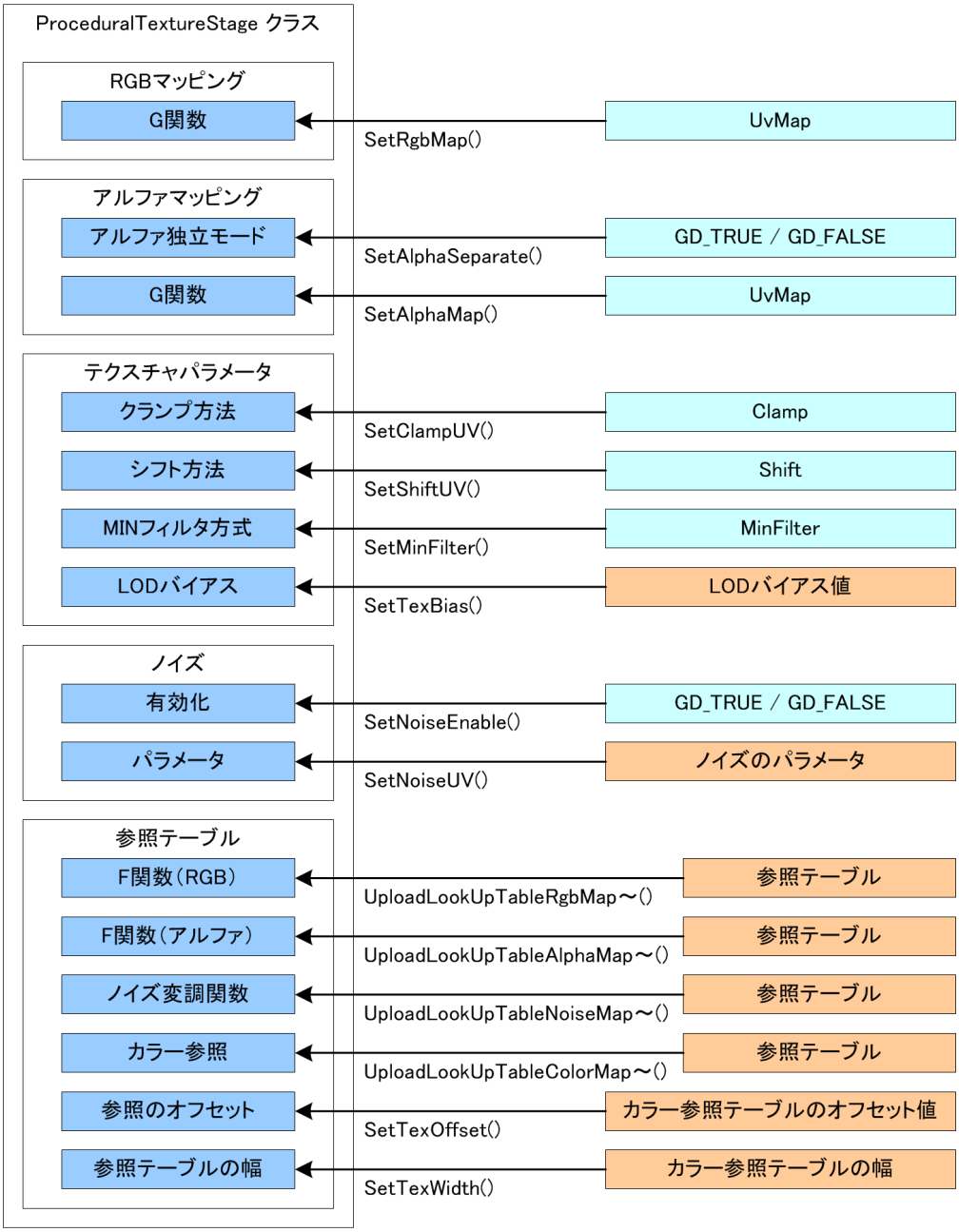
注意: フィルタ、ラッピングモード、LOD の設定を変更しないでください。

4.4. プロシージャルテクスチャステージ

プロシージャルテクスチャステージでは、プロシージャルテクスチャに関する設定を行います。これらの設定を行うための関数は nn::gd::ProceduralTextureStage クラスに定義されています。

以下の図はプロシージャルテクスチャステージで行われる設定の概要を示したものです。

図 4-4. プロシージャルテクスチャステージの概要



4.4.1. プロシージャルテクスチャのパラメータ設定

プロシージャルテクスチャの各パラメータに対する設定を行う関数が用意されています。

コード 4-31. プロシージャルテクスチャのパラメータ設定関数

```
class nn::gd::ProceduralTextureStage
{
    static void SetRgbMap(UvMap rgbMap);
    static void SetAlphaSeparate(gdBool alphaSeparate);
    static void SetAlphaMap(UvMap alphaMap);
    static void SetClampUV(Clamp u, Clamp v);
    static void SetShiftUV(Shift u, Shift v);
```

```
    static void SetMinFilter(MinFilter minFilter);
    static void SetTexBias(f32 texBias);
    static void SetNoiseEnable(gdBool noiseEnable);
    static void SetNoiseUV(f32 noiseU[3], f32 noiseV[3]);
    static void SetTexWidth(u8 texWidth);
    static void SetTexOffset(u8 texOffset);
};
```

下表に、それぞれの関数で設定が行われるパラメータを示します。引数の指定方法やパラメータの影響については関数リファレンスなどを参照してください。

表 4-12. プロシージャルテクスチャステージの関数で設定が行われるパラメータ

関数	パラメータ
SetRgbMap()	RGB マッピングの G 関数(デフォルト:UV_MAP_U)
SetAlphaSeparate()	アルファ独立モードの有効化(デフォルト:GD_FALSE)
SetAlphaMap()	アルファマッピングの G 関数(デフォルト:UV_MAP_U)
SetClampUV()	クランプモード(デフォルト:CLAMP_TO_ZERO)
SetShiftUV()	シフトモード(デフォルト:SHIFT_NONE)
SetMinFilter()	縮小時のフィルタモード(デフォルト:MIN_FILTER_LINEAR)
SetTexBias()	LOD バイアス(デフォルト:0.5)
SetNoiseEnable()	ノイズの有効化(デフォルト:GD_FALSE)
SetNoiseUV()	ノイズのパラメータ(デフォルト:U, V ともに F=0.0, P=0.0, A=0.0)
SetTexWidth()	カラー参照テーブルの幅(デフォルト:0)
SetTexOffset()	カラー参照テーブルのオフセット(デフォルト:0)

補足: SetTexBias() と SetNoiseUV() では、引数で渡された値を関数内で GPU のレジスタに設定する際のフォーマットに変換しています。

4.4.2. プロシージャルテクスチャステージで使用する参照テーブルのロード

F 関数(RGB マッピング、アルファマッピング)やノイズ変調データ、カラー参照テーブルといった、プロシージャルテクスチャで使用する参照テーブルをロードする関数には、浮動小数点数の配列で指定するもの(～Float())と、GPU のレジスタにそのまま書き込むデータの配列で指定するもの(～Native())が用意されています。なお、参照テーブルのロード関数はイミディエート関数ですので、実行時に 3D コマンドバッファにデータが書き込まれます。

コード 4-32. プロシージャルテクスチャステージで使用する参照テーブルのロード関数

```
class nn::gd::ProceduralTextureStage
{
    static nnResult UploadLookUpTableRgbMapFloat(
            u32 index, f32* Map, f32* MapDelta, u32 lutSize);
    static nnResult UploadLookUpTableRgbMapNative(
            u32 index, u32* Map, u32 lutSize);
    static nnResult UploadLookUpTableAlphaMapFloat(
```

```
            u32 index, f32* Map, f32* MapDelta, u32 lutSize);
    static nnResult UploadLookUpTableAlphaMapNative(
            u32 index, u32* Map, u32 lutSize);
    static nnResult UploadLookUpTableNoiseMapFloat(
            u32 index, f32* Map, f32* MapDelta, u32 lutSize);
    static nnResult UploadLookUpTableNoiseMapNative(
            u32 index, u32* Map, u32 lutSize);
    static nnResult UploadLookUpTableColorMapFloat(
            u32 index, f32** Map, f32** MapDelta, u32 lutSize);
    static nnResult UploadLookUpTableColorMapNative(
            u32 index, u32* Map, u32* MapDelta, u32 lutSize);
};
```

index と *lutSize* の指定によって、参照テーブルを部分的に書き換えることができます。ただし、その合計が下表で示されている要素の最大値を超えるような指定ではエラーが発生します。

浮動小数点数の配列で参照テーブルをロードする場合、データ(*Map*)と差分値(*MapDelta*)の 2 つの配列を用意する必要があります。ただし、カラー参照テーブルをロードする場合は 4 つのカラー成分それぞれで用意するため、8 つの配列が必要です。

ネイティブフォーマットに変換された配列で参照テーブルをロードする場合、カラー参照テーブル以外はデータと差分値が 1 つの要素にまとめられているため、用意する配列は 1 つだけです。

下表は、ロードする参照テーブル、配列のフォーマット、配列の要素の最大値を関数ごとに示したものです。

表 4-13. プロシージャルテクスチャステージで使用する参照テーブルのロード関数の一覧

関数	参照テーブル	フォーマット	要素数の最大値
UploadLookUpTableRgbMapFloat()	RGB マッピング	浮動小数点数	128
UploadLookUpTableRgbMapNative()	RGB マッピング	ネイティブ	128
UploadLookUpTableAlphaMapFloat()	アルファマッピング	浮動小数点数	128
UploadLookUpTableAlphaMapNative()	アルファマッピング	ネイティブ	128
UploadLookUpTableNoiseMapFloat()	ノイズ変調テーブル	浮動小数点数	128
UploadLookUpTableNoiseMapNative()	ノイズ変調テーブル	ネイティブ	128
UploadLookUpTableColorMapFloat()	カラー参照テーブル	浮動小数点数	256
UploadLookUpTableColorMapNative()	カラー参照テーブル	ネイティブ	256

4.4.2.1. 参照テーブルのロードのヘルパー関数

浮動小数点数の配列をネイティブフォーマットに変換するヘルパー関数が用意されています。浮動小数点数の配列から参照テーブルをロードする関数は、内部でネイティブへのフォーマット変換が行われるために負荷が高く、ランタイムでの実行には向いていません。そのため、あらかじめネイティブフォーマットに変換されたデータを用意するか、初回ロード時にヘルパー関数でフォーマット変換を行っておくなどの実装を推奨します。

コード 4-33. 参照テーブルのロードのヘルパー関数

```
class nn::gd::ProceduralTextureStage::Helper
{
    static nnResult ConvertLookUpTableDataFloatToNative(
            f32* valueData, f32* deltaData, u32 lutSize, u32* destination);
    static nnResult ConvertColorLookUpTableDataFloatToNative(
            f32** refArray, f32** deltaArray, u32 lutSize,
            u32* destRef, u32* destDelta);
};
```

カラー参照テーブル以外は、データと差分値の組み合わせから 1 つのデータ(*destination*)が生成されます。生成されるデータは、ビット [23 : 12] が差分値を小数部 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、ビット [11 : 0] がデータを小数部 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です。

カラー参照テーブルは、RGBA 各成分のデータと差分値の組み合わせ(合計 8 つの配列)から 1 つのカラー値と差分値の組み合わせ(*destRef* と *destDelta*)が生成されます。生成されるカラー値、差分値ともに、各成分を変換した値がアルファ成分、青成分、緑成分、赤成分の順に上位ビットから 8 ビットずつ配置されていますが、値の変換方法が異なります。カラー値は 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、差分値は -1.0 ～ 1.0 の範囲を小数部 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、それぞれ変換しています。

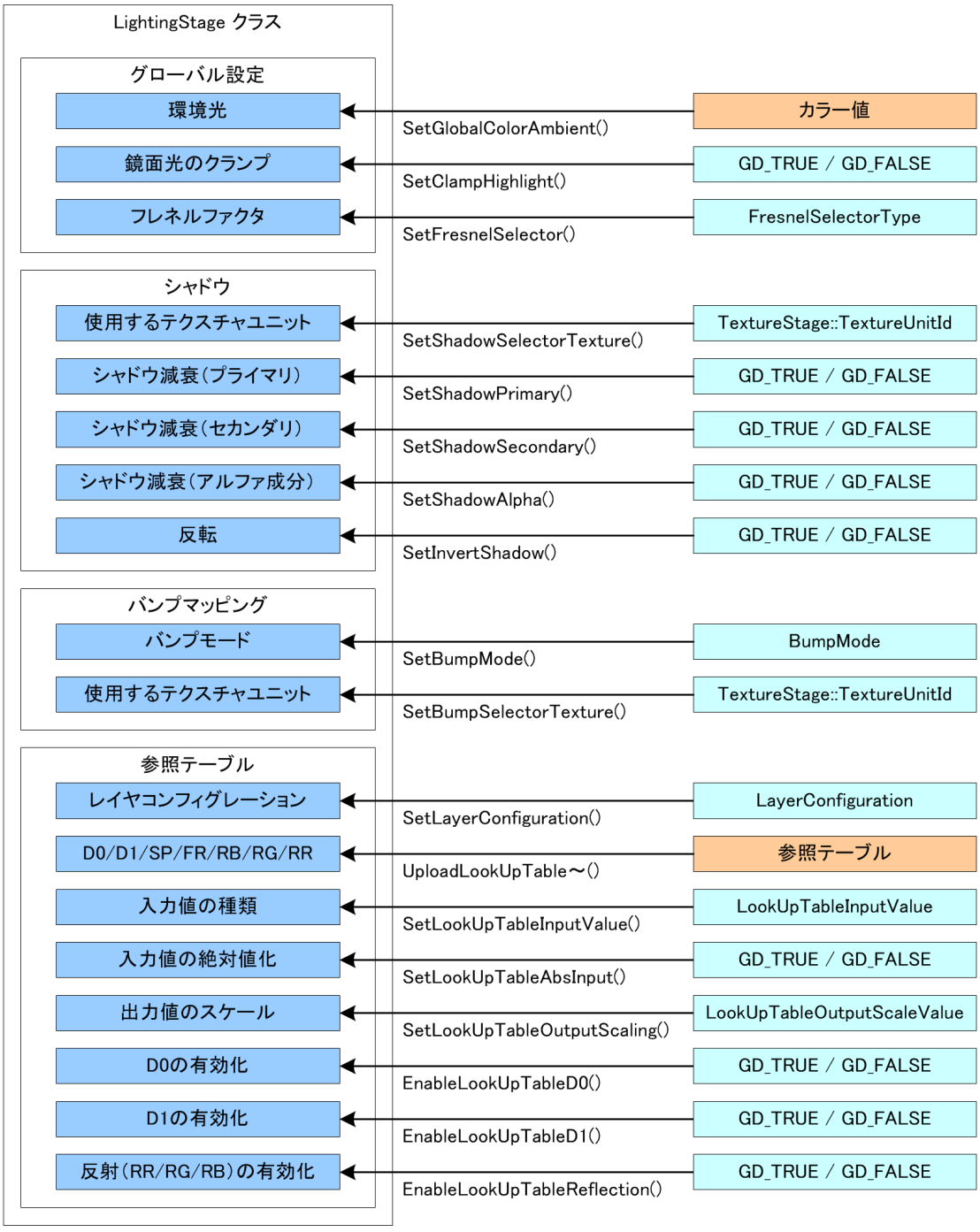
> **補足:** 変換方法の詳細は「3DS プログラミングマニュアル – グラフィックス応用編」を参照してください。
> また、GD ライブラリには変換のためのユーティリティ関数が用意されています。

4.5. ライティングステージ

ライティングステージでは、グローバルライティング環境の設定と個別のライト設定を行います。これらの設定を行う関数は nn::gd::LightingStage クラスに定義されています。なお、ライティング環境の有効/無効をアプリケーションで設定する必要はありません。ライティングの設定はライブラリで管理されており、コンバイナ設定でライティングの結果が必要な場合にだけライティング環境が有効となるように実装されています。

以下の図はライティングステージで行われる設定の概要(個別のライトを除く)を示したものです。

図 4-5. ライティングステージの概要（個別のライトを除く）



4.5.1. グローバルライティング環境の設定

グローバルライティング環境の各パラメータに対する設定を行う関数が用意されています。

コード 4-34. グローバルライティング環境のパラメータ設定関数

```
class nn::gd::LightingStage
{
    static void SetGlobalColorAmbient(u8 R, u8 G, u8 B);
    static void SetClampHightlight(gdBool value);
    static void SetFresnelSelector(FresnelSelectorType fresnelSelectorType);
```

```
    static void SetShadowSelectorTexture(
        nn::gd::TextureStage::TextureUnitId textureUnit);
    static void SetShadowPrimary(gdBool value);
    static void SetShadowSecondary(gdBool value);
    static void SetShadowAlpha(gdBool value);
    static void SetInvertShadow(gdBool value);

    static void SetBumpMode(BumpMode bumpMode, gdBool bumpRenorm);
    static void SetBumpSelectorTexture(
        nn::gd::TextureStage::TextureUnitId textureUnit);
};
```

下表に、それぞれの関数で設定が行われるパラメータを示します。引数の指定方法やパラメータの影響については関数リファレンスなどを参照してください。

表 4-14. ライティングステージの関数で設定が行われるパラメータ

関数	パラメータ
SetGlobalColorAmbient()	グローバル環境光の光源色(デフォルト:(10, 10, 10))
SetClampHightlight()	鏡面光のクランプ(デフォルト:GD_TRUE)
SetFresnelSelector()	フレネルファクタ(デフォルト:FRESNEL_SELECTOR_TYPE_NO_FRESNEL)
SetShadowSelectorTexture()	シャドウで使用するテクスチャユニット(デフォルト:TEXTURE_UNIT_0)
SetShadowPrimary()	プライマリカラーへのシャドウの影響(デフォルト:GD_FALSE)
SetShadowSecondary()	セカンダリカラーへのシャドウの影響(デフォルト:GD_FALSE)
SetShadowAlpha()	アルファ成分へのシャドウの影響(デフォルト:GD_FALSE)
SetInvertShadow()	シャドウ項の反転(デフォルト:GD_FALSE)
SetBumpMode()	バンプマッピングの摂動モードと法線の第 3 成分の再生成(デフォルト:BUMPMODE_NOT_USED, GD_FALSE)
SetBumpSelectorTexture()	バンプマッピングで使用するテクスチャユニット(デフォルト:TEXTURE_UNIT_0)

補足: SetGlobalColorAmbient() はイミディエート関数です。

4.5.2. レイヤコンフィグレーションと参照テーブルの設定

ライティングで使用される参照テーブルを決定するレイヤコンフィグレーション、参照テーブルへの入力値、参照テーブルからの出力値のスケーリング、ディストリビューションファクタ、反射を設定する関数が用意されています。

コード 4-35. 参照テーブルに関係するパラメータの設定関数

```
class nn::gd::LightingStage
{
    static void SetLayerConfiguration(LayerConfiguration layerConfiguration);
    static void SetLookUpTableInputValue(
        LookUpTableId lutId, LookUpTableInputValue lutInputValue);
    static void SetLookUpTableAbsInput(LookUpTableId lutId, gdBool value);
    static void SetLookUpTableOutputScaling(
        LookUpTableId lutId, LookUpTableOutputScaleValue outputScalingValue);
    static void EnableLookUpTableD0(gdBool value);
    static void EnableLookUpTableD1(gdBool value);
    static void EnableLookUpTableReflection(gdBool value);
};
```

下表に、それぞれの関数で設定が行われるパラメータを示します。引数の指定方法やパラメータの影響については関数リファレンスなどを参照してください。

表 4-15. ライティングステージの関数で設定が行われる参照テーブルに関するパラメータ

関数	パラメータ
SetLayerConfiguration()	レイヤコンフィグレーション(デフォルト:LAYER_CONFIGURATION_0)
SetLookUpTableInputValue()	参照テーブルへの入力値(デフォルト:参照テーブルすべて INPUT_VALUE_NH)
SetLookUpTableAbsInput()	参照テーブルへの入力値の絶対値化(デフォルト:参照テーブルすべて GD_FALSE)
SetLookUpTableOutputScaling()	参照テーブルからの出力値のスケーリング(デフォルト:参照テーブルすべて OUTPUT_SCALE_VALUE_1)
EnableLookUpTableD0()	ディストリビューションファクタ 0 への参照テーブルの適用(デフォルト:GD_FALSE)
EnableLookUpTableD1()	ディストリビューションファクタ 1 への参照テーブルの適用(デフォルト:GD_FALSE)
EnableLookUpTableReflection()	反射への参照テーブルの適用(デフォルト:GD_FALSE)

4.5.3. ライティングステージで使用する参照テーブルのロード

ライティングで使用する参照テーブルをロードする関数には、浮動小数点数の配列で指定するもの(UploadLookUpTableFloat())と、GPU のレジスタにそのまま書き込むデータの配列で指定するもの(UploadLookUpTableNative())が用意されています。なお、参照テーブルのロード関数はイミディエート関数ですので、実行時に 3D コマンドバッファにデータが書き込まれます。

コード 4-36. ライティングステージで使用する参照テーブルのロード関数

```
class nn::gd::LightingStage
{
    static nnResult UploadLookUpTableFloat(
        LookUpTableUploadId lutID, u32 lutStartIndex,
        const f32* valueData, const f32* deltaData, u32 dataCount);
    static nnResult UploadLookUpTableNative(
```

```
        LookUpTableUploadId lutID, u32 lutStartIndex,
        const u32* data, u32 dataCount);

    class Helper
    {
        static nnResult ConvertLookUpTableDataFloatToNative(
            const f32* valueData, const f32* deltaData, u32 dataCount,
            u32* __restrict destination);
    };
};
```

ライティングで使用する参照テーブルの要素数は 256 固定です。*lutStartIndex* と *dataCount* の指定によって、参照テーブルを部分的に書き換えることができますが、その合計が 256 を超えるような指定ではエラーが発生します。

なお、参照テーブルへの入力値が絶対値化されるかどうかによって、参照テーブルの構成が異なります。絶対値化しない(入力値の範囲が -1.0～1.0)場合は、参照テーブルの 128 番目と 129 番目の要素に対応する入力値が不連続になる(0.9922 から -1.0 にジャンプする)ことに注意してください。

浮動小数点数の配列で参照テーブルをロードする場合、データ(*valueData*)と差分値(*deltaData*)の 2 つの配列を用意する必要があります。

ネイティブフォーマットに変換された配列で参照テーブルをロードする場合、参照テーブルにロードするデータはデータと差分値が 1 つの要素にまとめられているため、用意する配列は 1 つだけです。

浮動小数点数の配列をネイティブフォーマットに変換するヘルパー関数が用意されています。ヘルパー関数では、データと差分値の組み合わせから 1 つのデータ(*destination*)を生成します。生成されるデータは、ビット [23 : 12] が差分値を小数部 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)に変換した値、ビット [11 : 0] がデータを小数部 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です。

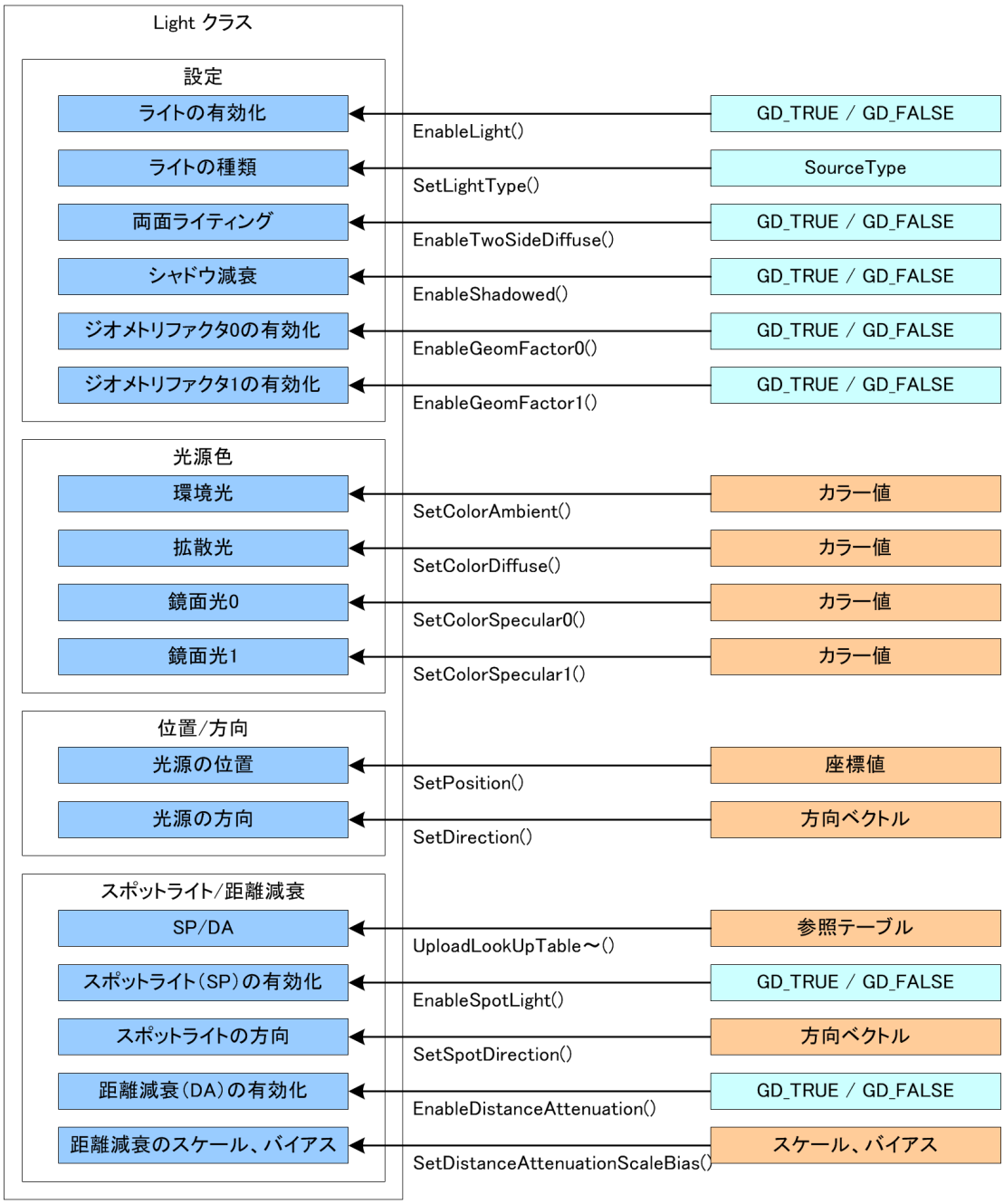
> **補足:** 変換方法の詳細は「3DS プログラミングマニュアル - グラフィックス応用編」を参照してください。
> また、GD ライブラリには変換のためのユーティリティ関数が用意されています。

4.5.4. ライト

3DS のハードウェアは 8 個のライトによるライティングに対応しています。GD ライブラリでは、個々のライトを `nn::gd::Light` クラスの個別のインスタンスとして表現し、`nn::gd::LightingStage` クラスに内包するようにして 8 個分の `Light` オブジェクトがライブラリの初期化時に作成されます。そのため、アプリケーションで `Light` オブジェクトを作成する必要はなく、`nn::gd::LightingStage::light[0].EnableLight(GD_TRUE)` のように、ライティングステージに定義されたメンバとしてアクセスします。

以下の図は `Light` オブジェクトで行われる設定の概要を示したものです。

図 4-6. Light オブジェクトの概要



4.5.4.1. ライトの設定

ライトの各パラメータに対する設定を行う関数が用意されています。

コード 4-37. ライトのパラメータ設定関数

```
class nn::gd::Light
{
    void EnableLight(gdBool value);
    void SetLightType(SourceType sourceType);
    void EnableTwoSideDiffuse(gdBool value);
    void EnableShadowed(gdBool value);
    void EnableGeomFactor0(gdBool value);
```

```
    void EnableGeomFactor1(gdBool value);

    void SetColorAmbient(u8 R, u8 G, u8 B);
    void SetColorDiffuse(u8 R, u8 G, u8 B);
    void SetColorSpecular0(u8 R, u8 G, u8 B);
    void SetColorSpecular1(u8 R, u8 G, u8 B);

    void SetPosition(f32 X, f32 Y, f32 Z);
    void SetDirection(f32 X, f32 Y, f32 Z);

    void EnableSpotLight(gdBool value);
    void SetSpotDirection(f32 X, f32 Y, f32 Z);
    void EnableDistanceAttenuation(gdBool value);
    void SetDistanceAttenuationScaleBias(
        f32 attenuationScale, f32 attenuationBias);
    nnResult UploadLookUpTableFloat(
        LightLookUpTableUploadId lightLutID, u32 lutStartIndex,
        const f32* valueData, const f32* deltaData, u32 dataCount);
    nnResult UploadLookUpTableNative(
        LightLookUpTableUploadId lightLutID, u32 lutStartIndex,
        const u32* data, u32 dataCount);
};
```

下表に、それぞれの関数で設定が行われるパラメータを示します。引数の指定方法やパラメータの影響については関数リファレンスなどを参照してください。

表 4-16. Light オブジェクトの関数で設定が行われるパラメータ

関数	パラメータ
EnableLight()	ライトの有効化(デフォルト:GD_FALSE)
SetLightType()	光源の種類(デフォルト:SOURCE_TYPE_POINT(点光源))
EnableTwoSideDiffuse()	両面ライティングの有効化(デフォルト:GD_FALSE)
EnableShadowed()	シャドウ減衰の有効化(デフォルト:GD_FALSE)
EnableGeomFactor0()	ジオメトリファクタ 0 の有効化(デフォルト:GD_FALSE)
EnableGeomFactor1()	ジオメトリファクタ 1 の有効化(デフォルト:GD_FALSE)
SetColorAmbient()	環境光の色(デフォルト:(0, 0, 0))
SetColorDiffuse()	拡散光の色(デフォルト:(255, 255, 255))
SetColorSpecular0()	鏡面光 0 の色(デフォルト:(255, 255, 255))
SetColorSpecular1()	鏡面光 1 の色(デフォルト:(255, 255, 255))
SetPosition()	点光源の位置(デフォルト:(0.0, 0.0, 0.0))
SetDirection()	平行光源の方向(デフォルト設定なし)
EnableSpotLight()	スポットライトの有効化(デフォルト:GD_FALSE)
SetSpotDirection()	スポットライトの方向(デフォルト:(0.0, 0.0, 0.0))
EnableDistanceAttenuation()	距離減衰の有効化(デフォルト:GD_FALSE)

SetDistanceAttenuationScaleBias()	距離減衰のスケールとバイアス(デフォルト:0.0, 0.0) 開始距離と終了距離から設定値を求めるユーティリティ関数(Utils::ConvertStartEndToScaleBias())が用意されています。

参照テーブルの指定を除いて、参照テーブルのロード方法はライティングステージでの参照テーブルのロード方法と同じです。なお、**距離減衰(DA)で使用する参照テーブルは入力値の範囲が 0.0～1.0 である**ことを想定して作成されていなければなりません。

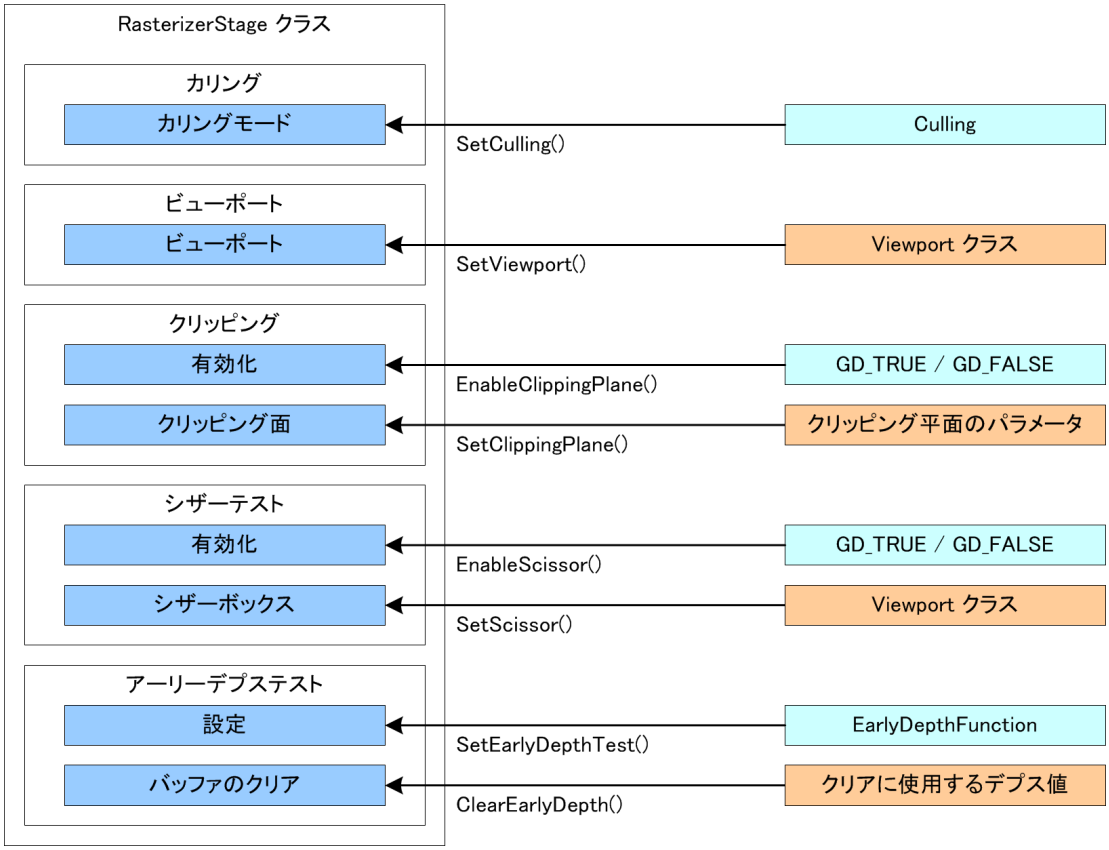
> **補足:** ライトの設定を行う関数は、EnableLight()、EnableShadowed()、EnableSpotLight()、EnableDistanceAttenuation() を除いて、すべてイミディエート関数です。

4.6. ラスタライザステージ

ラスタライザステージでは、フラグメントパイプラインに出力されるピクセルのラスタライズ処理の設定を行います。これらの設定を行う関数は nn::gd::RasterizerStage クラスに定義されています。

以下の図はラスタライザステージで行われる設定の概要を示したものです。

図 4-7. ラスタライザステージの概要



4.6.1. カリングの設定

カリングの設定は nn::gd::RasterizerStage::SetCulling() で行います。

コード 4-38. カリングの設定関数

```
static void nn::gd::RasterizerStage::SetCulling(
        nn::gd::RasterizerStage::Culling culling);
```

culling にカリングの方法を指定します。デフォルトでは時計回り(CULLING_CLOCKWISE)が指定されています。

4.6.2. ビューポートの設定

ビューポートの設定は nn::gd::RasterizerStage::SetViewport() で行います。

コード 4-39. ビューポートの設定関数

```
class nn::gd::Viewport
{
    u32 m_X;
    u32 m_Y;
    u32 m_Width;
    u32 m_Height;

    Viewport();
    Viewport(u32 x, u32 y, u32 width, u32 height);
    void Set(u32 x, u32 y, u32 width, u32 height);
};

static void nn::gd::RasterizerStage::SetViewport(
        nn::gd::Viewport& viewport);
```

ビューポートの領域は nn::gd::Viewport クラスで定義します。ビューポートの方向は LCD の配置方向と異なることに注意してください。デフォルトでは Viewport(0, 0, 240, 320) が設定されています。

4.6.3. クリッピングの設定

クリッピングの設定は以下の関数で行います。

コード 4-40. クリッピングの設定関数

```
static void nn::gd::RasterizerStage::EnableClippingPlane(gdBool value);
static void nn::gd::RasterizerStage::SetClippingPlane(
        f32 param1, f32 param2, f32 param3, f32 param4);
```

デフォルトではクリッピングは無効(GD_FALSE)で、パラメータはすべて 0 に設定されています。

4.6.4. シザーテストの設定

シザーテストの設定は以下の関数で行います。

コード 4-41. シザーテストの設定関数

```
static void nn::gd::RasterizerStage::EnableScissor(gdBool value);
static void nn::gd::RasterizerStage::SetScissor(
        nn::gd::Viewport& area);
```

デフォルトではシザーテストは無効(GD_FALSE)で、シザーボックスには Viewport(0, 0, 0, 0) が設定されています。

4.6.5. アーリーデプステストの設定

アーリーデプステストの設定は以下の関数で行います。

コード 4-42. アーリーデプステストの設定関数

```
static void nn::gd::RasterizerStage::SetEarlyDepthTest(
        gdBool enable, nn::gd::RasterizerStage::EarlyDepthFunction func);
static void nn::gd::RasterizerStage::ClearEarlyDepth(f32 depthClearValue);
```

デフォルトではアーリーデプステストは無効(GD_FALSE, EARLYDEPTH_FUNCTION_LESS)で、クリアに使用するデプス値には 1.0 が設定されています。

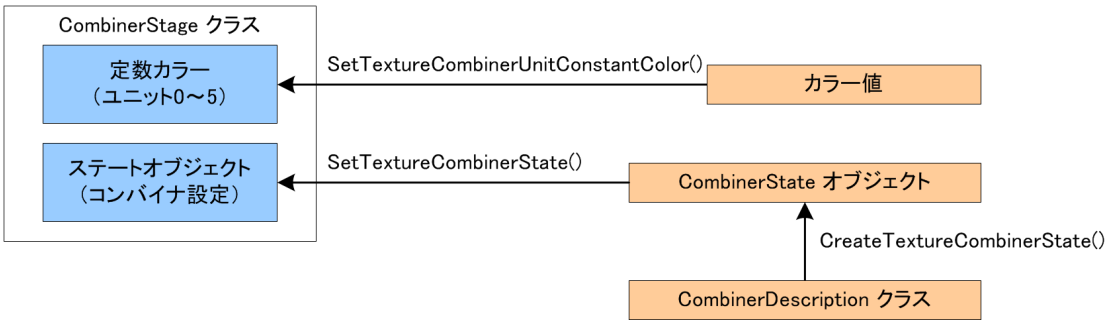
アーリーデプステストを使用する場合は、ブロックモードがブロック 32 モードに設定されていなければなりません。詳しくは「3DS プログラミングマニュアル - グラフィックス応用編」の「ブロックモードの設定」を参照してください。

4.7. コンバイナステージ

コンバイナステージでは、(テクスチャ)コンバイナとコンバイナバッファの設定を行います。これらの設定を行う関数は nn::gd::CombinerStage クラスに定義されています。

以下の図はコンバイナステージで行われる設定の概要を示したものです。

図 4-8. コンバイナステージの概要



4.7.1. コンバイナ設定

コンバイナステージでは、すべてのコンバイナとコンバイナバッファの設定をひとまとめにしたステートオブジェクト(CombinerState オブジェクト)を作成し、CombinerState オブジェクトを切り替えることで定数カラー以外のコンバイナに関する設定を一括で変更することができます。

CombinerState オブジェクトはオペランドなどの設定が定義されたデスクリプタクラス(CombinerDescription)から作成します。

コード 4-43. コンバイナ設定の作成、解放、パイプラインへの設定

```
static nnResult nn::gd::CombinerStage::CreateTextureCombinerState(
        const nn::gd::CombinerDescription* description,
        nn::gd::CombinerState** textureCombinerState);
static nnResult nn::gd::CombinerStage::ReleaseTextureCombinerState(
        nn::gd::CombinerState* textureCombinerState);
static nnResult nn::gd::CombinerStage::SetTextureCombinerState(
        nn::gd::CombinerState* textureCombinerState);
```

コンバイナへの設定は *description* に指定するデスクリプタクラス(CombinerDescription)にあらかじめ設定します。デスクリプタクラスに設定する情報の詳細は後述します。

作成された CombinerState オブジェクトは nn::gd::CombinerStage::SetTextureCombinerState() でコンバイナステージに設定することができます。

不要になったステートオブジェクトは nn::gd::CombinerStage::ReleaseTextureCombinerState() で解放してください。このとき、オブジェクトが作成されたときに内部で確保されたメモリ領域も解放されます。

コンバイナへの設定のうち、定数カラーの設定だけは以下の関数で行います。デフォルトの定数カラーは不定値ですので、コンバイナ設定で定数カラーを使用する場合は必ず設定を行ってください。

コード 4-44. 定数カラーの設定

```
static nnResult nn::gd::CombinerStage::SetTextureCombinerUnitConstantColor(
        nn::gd::CombinerStage::UnitId unit,
        u8 colorR, u8 colorG, u8 colorB, u8 colorA);
```

この関数はイミディエート関数ですので、実行と同時にコマンドバッファへの 3D コマンドの蓄積を行います。

4.7.1.1. CombinerState オブジェクトのデスクリプタクラス

CombinerState オブジェクトのデスクリプタクラス(CombinerDescription クラス)のメンバを紹介し、設定に際しての注意点などを説明します。

CombinerDescription クラスは 6 つのコンバイナユニットの設定を内包しています。必ずしもすべてのユニットに対して設定を行うことがないため、ユニットを指定してメンバ変数への設定を行うメンバ関数を呼び出すように設計されています。

コード 4-45. CombinerDescription クラスの定義

```
class nn::gd::CombinerDescription
{
    struct BufferInput
    {
        nn::gd::CombinerStage::BufferInput m_Rgb1;
        nn::gd::CombinerStage::BufferInput m_Rgb2;
        nn::gd::CombinerStage::BufferInput m_Rgb3;
        nn::gd::CombinerStage::BufferInput m_Rgb4;
        nn::gd::CombinerStage::BufferInput m_Alpha1;
        nn::gd::CombinerStage::BufferInput m_Alpha2;
        nn::gd::CombinerStage::BufferInput m_Alpha3;
        nn::gd::CombinerStage::BufferInput m_Alpha4;
    };
    struct BufferInput m_BufferInput;

    void SetSourceRGB(
            nn::gd::CombinerStage::UnitId unit,
            nn::gd::CombinerStage::Source sourceRGB1,
            nn::gd::CombinerStage::Source sourceRGB2,
            nn::gd::CombinerStage::Source sourceRGB3);
    void SetOperandRGB(
            nn::gd::CombinerStage::UnitId unit,
            nn::gd::CombinerStage::OperandRgb opRGB1,
            nn::gd::CombinerStage::OperandRgb opRGB2,
            nn::gd::CombinerStage::OperandRgb opRGB3);
```

```
    void SetSourceAlpha(
            nn::gd::CombinerStage::UnitId unit,
            nn::gd::CombinerStage::Source sourceA1,
            nn::gd::CombinerStage::Source sourceA2,
            nn::gd::CombinerStage::Source sourceA3);
    void SetOperandAlpha(
            nn::gd::CombinerStage::UnitId unit,
            nn::gd::CombinerStage::OperandAlpha opA1,
            nn::gd::CombinerStage::OperandAlpha opA2,
            nn::gd::CombinerStage::OperandAlpha opA3);
    void SetCombineRGB(
            nn::gd::CombinerStage::UnitId unit,
            nn::gd::CombinerStage::CombineRgb combineRgb);
    void SetCombineAlpha(
            nn::gd::CombinerStage::UnitId unit,
            nn::gd::CombinerStage::CombineAlpha combineAlpha);
    void SetScaleRGB(
            nn::gd::CombinerStage::UnitId unit,
            nn::gd::CombinerStage::Scale scaleRgb);
    void SetScaleAlpha(
            nn::gd::CombinerStage::UnitId unit,
            nn::gd::CombinerStage::Scale scaleAlpha);
    void SetBufferColor(u8 colorR, u8 colorG, u8 colorB, u8 colorA);
    void SetCombinerInUse(nn::gd::CombinerStage::UnitId unit, gdBool mode);

    CombinerDescription();
    void ToDefault();
};
```

入力ソースの設定（SetSourceRGB()、SetSourceAlpha()）

指定されたコンバイナユニットの入力ソース 0～2 を設定します。

オペランドの設定（SetOperandRGB()、SetOperandAlpha()）

指定されたコンバイナユニットのオペランド 0～2 を設定します。

コンバイナ関数の設定（SetCombineRGB()、SetCombineAlpha()）

指定されたコンバイナユニットのコンバイナ関数を設定します。

スケーリング値の設定（SetScaleRGB()、SetScaleAlpha()）

指定されたコンバイナユニットのスケーリング値を設定します。

コンバイナバッファ 0 の定数カラーの設定（SetBufferColor()）

コンバイナバッファ 0 の定数カラーを設定します。

コンバイナバッファの入力ソース（m_BufferInput）

コンバイナバッファ 1～4 の入力ソースを設定します。設定用の関数は用意されていません。

コンバイナユニットの使用設定（SetCombinerInUse()）

指定されたコンバイナユニットの設定内容を有効（GD_TRUE）または無効（GD_FALSE）に設定します。設定内容を有効にしなければ、そのユニットの設定はデフォルトの設定になります。

デフォルトの設定（ToDefault()）

ToDefault() で、すべてのメンバをデフォルトの値に変更することができます。この設定はコンストラクタでも行われています。

表 4-17. デフォルトのコンバイナ設定

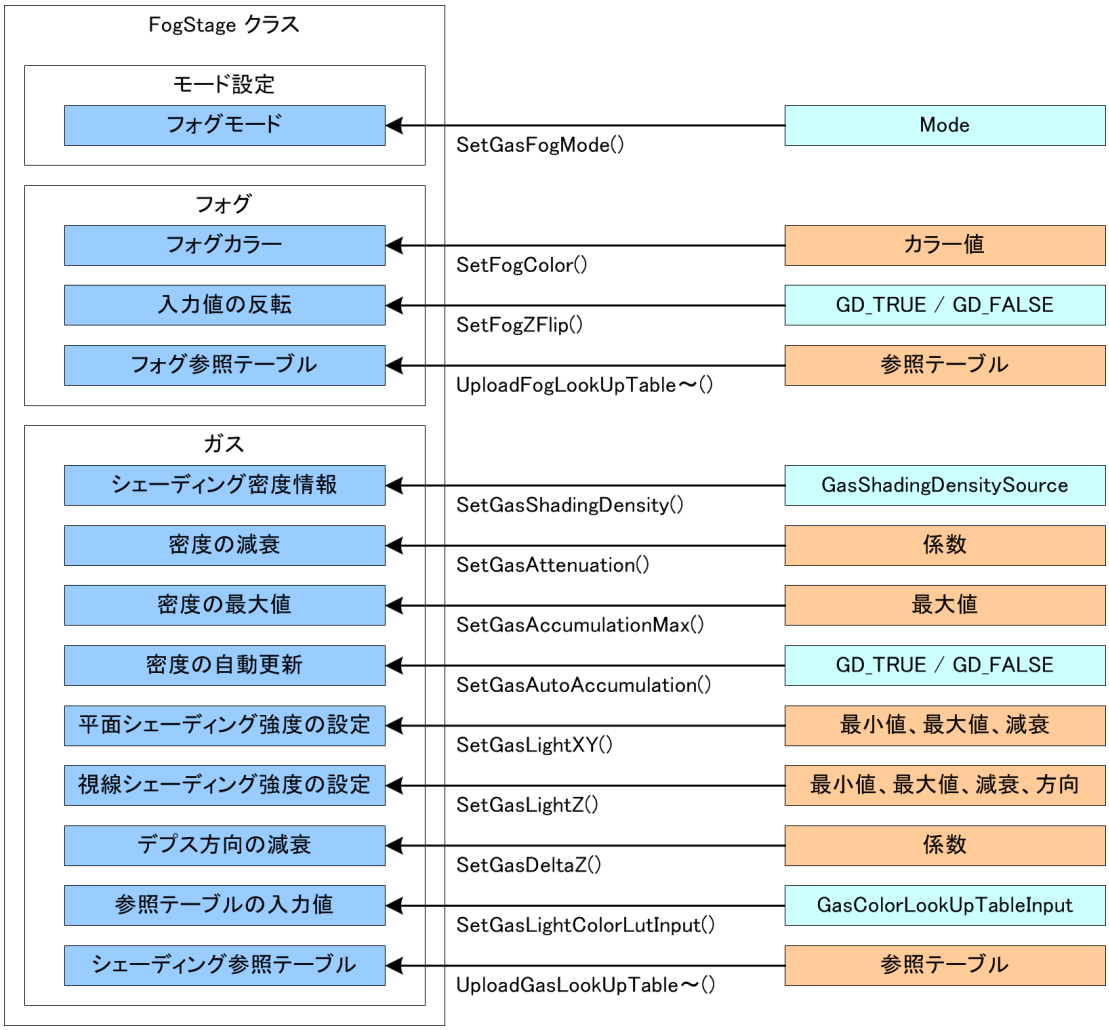
メンバ	デフォルトの設定内容
SetSourceRGB() SetSourceAlpha()	SOURCE_CONSTANT（ユニット 0 のすべての入力ソース） SOURCE_PREVIOUS（ユニット 1～5 のすべての入力ソース）
SetOperandRGB()	OPERAND_RGB_SRC_COLOR（すべてのオペランド）
SetOperandAlpha()	OPERAND_ALPHA_SRC_ALPHA（すべてのオペランド）
SetCombineRGB()	COMBINE_RGB_REPLACE（すべてのユニット）
SetCombineAlpha()	COMBINE_ALPHA_REPLACE（すべてのユニット）
SetScaleRGB() SetScaleAlpha()	SCALE_1（すべてのユニット）
SetBufferColor()	(0, 0, 0, 0)
m_BufferInput	INPUT_PREVIOUS_BUFFER（すべての入力）
SetCombinerInUse()	GD_FALSE（すべてのユニット）

4.8. フォグステージ

フォグステージでは、フォグとガスの描画に関連するすべての設定を行います。これらの設定を行うための関数は nn::gd::FogStage クラスに定義されています。

以下の図はフォグステージで行われる設定の概要を示したものです。

図 4-9. フォグステージの概要



4.8.1. フォグモード

GPU のフォグ機能のモード切り替えは `nn::gd::FogStage::SetGasFogMode()` で行います。

コード 4-46. フォグモードの切り替え

```
static void nn::gd::FogStage::SetGasFogMode(nn::gd::FogStage::Mode mode);
```

mode に指定する値は以下のものから選択します。

表 4-18. フォグモードの指定

定義	説明
FogStage::MODE_NONE	フォグ機能を使用しません。(デフォルト)
FogStage::MODE_FOG	フォグ機能を通常のフォグモードで使用します。
FogStage::MODE_GAS	フォグ機能をガスモードで使用します。

4.8.2. フォグの設定

フォグの各パラメータに対する設定を行う関数が用意されています。

コード 4-47. フォグのパラメータ設定関数

```
class nn::gd::FogStage
{
    static void SetFogColor(u8 R, u8 G, u8 B);
    static void SetFogZFlip(gdBool zFlip);
};
```

下表に、それぞれの関数で設定が行われるパラメータを示します。引数の指定方法やパラメータの影響については関数リファレンスなどを参照してください。

表 4-19. フォグステージの関数で設定が行われるフォグのパラメータ

関数	パラメータ
SetFogColor()	フォグカラー(デフォルト:(0, 0, 0))
SetFogZFlip()	入力デプス値の反転(デフォルト:GD_FALSE)

補足: 参照テーブルのロード方法については「4.8.4. フォグステージで使用する参照テーブルのロード」を参照してください。

4.8.3. ガスの設定

ガスの各パラメータに対する設定を行う関数が用意されています。

コード 4-48. ガスのパラメータ設定関数

```
class nn::gd::FogStage
{
    static void SetGasShadingDensity(
        GasShadingDensitySource shadingDensitySrc);
    static void SetGasAttenuation(f32 attenuation);
    static void SetGasAccumulationMax(f32 maxAccumulation);
    static void SetGasAutoAccumulation(gdBool enableAutoAccumulation);
    static void SetGasLightXY(
        u8 lightMinimum, u8 lightMaximum, u8 lightAttenuation);
    static void SetGasLightZ(
        u8 scattMinimum, u8 scattMaximum, u8 scattAttenuation, u8 lz);
    static void SetGasDeltaZ(f32 deltaZ);
    static void SetGasLightColorLutInput(
        GasColorLookUpTableInput colorLutInput);
};
```

下表に、それぞれの関数で設定が行われるパラメータを示します。引数の指定方法やパラメータの影響については関数リファレンスなどを参照してください。

表 4-20. フォグステージの関数で設定が行われるガスのパラメータ

関数	パラメータ
SetGasShadingDensity()	シェーディングで使用する密度情報(デフォルト:GAS_SHADING_DENSITY_SOURCE_PLAIN)
SetGasAttenuation()	シェーディングのアルファ値の計算で使用する密度減衰の係数(デフォルト:1.0)
SetGasAccumulationMax()	密度情報の最大値の逆数(デフォルト:1.0)
SetGasAutoAccumulation()	密度情報の最大値の逆数を自動的に計算するかどうか(デフォルト:GD_FALSE)
SetGasLightXY()	平面シェーディングの制御に使用する最小強度、最大強度、密度の減衰(デフォルト:(0, 0, 0))
SetGasLightZ()	視線シェーディングの制御に使用する最小強度、最大強度、密度の減衰、視線方向の影響(デフォルト:(0, 0, 0, 0))
SetGasDeltaZ()	デプス方向の減衰係数(デフォルト:10.0)
SetGasLightColorLutInput()	シェーディング参照テーブルへの入力値(デフォルト:GAS_COLOR_INPUT_LIGHT_FACTOR)

補足: 参照テーブルのロード方法については「4.8.4. フォグステージで使用する参照テーブルのロード」を参照してください。

4.8.4. フォグステージで使用する参照テーブルのロード

フォグステージで使用するフォグ係数の参照テーブルとシェーディング参照テーブルをロードする関数には、浮動小数点数の配列で指定するもの(～Float())と、GPU のレジスタにそのまま書き込むデータの配列で指定するもの(～Native())が用意されています。なお、参照テーブルのロード関数はイミディエート関数ですので、実行時に 3D コマンドバッファにデータが書き込まれます。

コード 4-49. フォグステージで使用する参照テーブルのロード関数

```
class nn::gd::FogStage
{
    static nnResult UploadFogLookUpTableNative(
        u32 lutStartIndex, u32* data, u32 countData);
    static nnResult UploadFogLookUpTableFloat(
        u32 lutStartIndex, const f32* dataValue, const f32* dataDelta,
        u32 countData);
    static nnResult UploadGasLookUpTableNative(
        u32 lutStartIndex, u32* data, u32 countData);
    static nnResult UploadGasLookUpTableFloat(
        u32 lutStartIndex, f32* dataValue, f32* dataDelta, u32 countData);
};
```

lutStartindex と *countData* の指定によって、参照テーブルを部分的に書き換えることができます。ただし、その合計が下表で示されている要素の最大値を超えるような指定ではエラーが発生します。

浮動小数点数の配列で参照テーブルをロードする場合、データ(*dataValue*)と差分値(*dataDelta*)の 2 つの配列を用意する必要があります。ただし、シェーディング参照テーブルは 3 要素(カラー成分を RGB の順番で格納)から 1 つのデー

タが生成されるため、要素数が(*countData* * 3)の配列を指定しなければならないことに注意してください。

ネイティブフォーマットに変換された配列で参照テーブルをロードする場合、データと差分値が 1 つの要素にまとめられているため、用意する配列は 1 つだけです。ただし、シェーディング参照テーブルのロードで用意する配列の要素数の最大値が、ネイティブの配列と浮動小数点数の配列とで異なることに注意してください。

下表は、ロードする参照テーブル、配列のフォーマット、配列の要素の最大値を関数ごとに示したものです。

表 4-21. フォグステージで使用する参照テーブルのロード関数の一覧

関数	参照テーブル	フォーマット	要素数の最大値
UploadFogLookUpTableFloat()	フォグ係数	浮動小数点数	128
UploadFogLookUpTableNative()	フォグ係数	ネイティブ	128
UploadGasLookUpTableFloat()	シェーディング	浮動小数点数	8
UploadGasLookUpTableNative()	シェーディング	ネイティブ	16

4.8.4.1. 参照テーブルのロードのヘルパー関数

浮動小数点数の配列をネイティブフォーマットに変換するヘルパー関数が用意されています。浮動小数点数の配列から参照テーブルをロードする関数は、内部でネイティブへのフォーマット変換が行われるために負荷が高く、ランタイムでの実行には向いていません。そのため、あらかじめネイティブフォーマットに変換されたデータを用意するか、初回ロード時にヘルパー関数でフォーマット変換を行っておくなどの実装を推奨します。

コード 4-50. 参照テーブルのロードのヘルパー関数

```
class nn::gd::FogStage::Helper
{
    static nnResult ConvertFogLookUpTableDataFloatToNative(
        const f32* dataValue, const f32* dataDelta, u32 countData,
        u32 *__restrict dest);
    static nnResult ConvertGasLookUpTableDataFloatToNative(
        const f32* dataValue, const f32* dataDelta, u32 countData,
        u32 *__restrict dest);
};
```

フォグステージで用意されているヘルパー関数は、いずれもデータと差分値の組み合わせから 1 つのデータ(*dest*)を生成します。

フォグ係数の参照テーブル用に生成されるデータは、ビット [23 : 13] がデータを小数部 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、ビット [12 : 0] が差分値を小数部 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)に変換した値です。

シェーディング参照テーブル用に生成されるデータは特殊です。特に変換元の配列の 3 要素(R、G、B の順)から 1 つのデータを生成することと、変換後の配列では 1 つのデータの中にデータと差分値が変換されるのではなく、配列の前半 8 つと後半 8 つそれぞれにデータと差分値が変換されることに注意してください。

前半の 8 つは差分値から生成されます。生成されるデータは、RGB 各成分の差分値を符号つき 8 ビット整数に変換したものを、ビット [23 : 16] に B 成分、ビット [15 : 8] に G 成分、ビット [7 : 0] に R 成分をパッキングしたものです。

後半の 8 つはデータから生成されます。生成されるデータは、RGB 各成分のデータを小数部 0 ビットの符号なし 8 ビット固定小数点数に変換したものを、ビット [23 : 16] に B 成分、ビット [15 : 8] に G 成分、ビット [7 : 0] に R 成分をパッキングし

たものです。

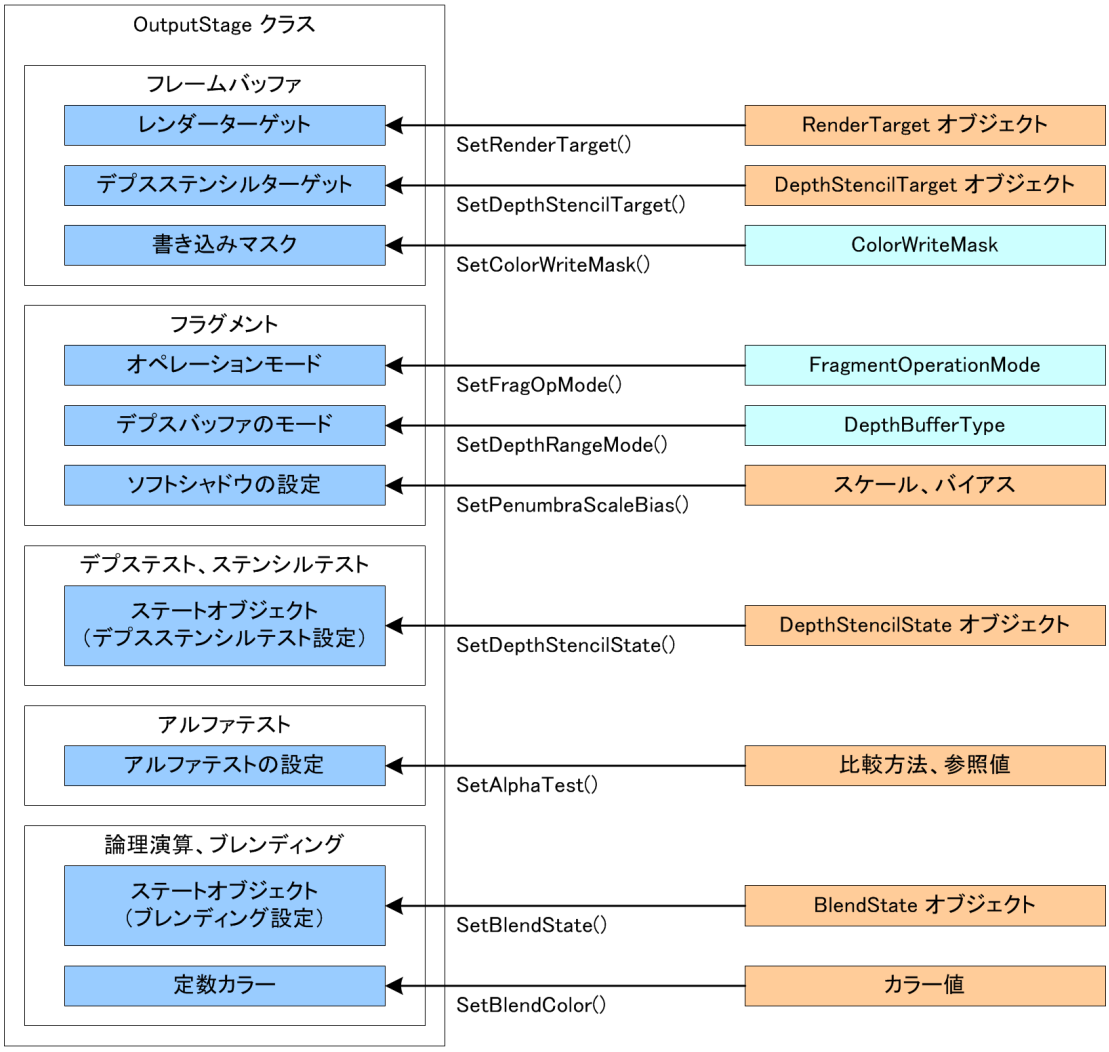
> **補足:** 変換方法の詳細は「3DS プログラミングマニュアル – グラフィックス応用編」を参照してください。
> また、GD ライブラリには変換のためのユーティリティ関数が用意されています。

4.9. アウトプットステージ

アウトプットステージでは、ブレンディングやデプステスト、ステンシルテスト、ターゲットバッファの設定など、ピクセル出力に関するすべての設定を行います。これらの設定を行う関数は `nn::gd::OutputStage` クラスに定義されています。

以下の図はアウトプットステージで行われる設定の概要を示したものです。

図 4-10. アウトプットステージの概要



4.9.1. フラグメントオペレーションモード

フラグメントオペレーションモードの切り替えは `nn::gd::OutputStage::SetFragOpMode()` で行います。

コード 4-51. フラグメントオペレーションモードの設定関数

```
static void nn::gd::OutputStage::SetFragOpMode(
        nn::gd::OutputStage::FragmentOperationMode fragOpMode);
```

fragOpMode に指定する値は以下のものから選択します。

表 4-22. フラグメントオペレーションモード

定義	説明
FRAGMENT_OPERATION_MODE_FRAGMENT_COLOR	標準のフラグメント処理(デフォルト)
FRAGMENT_OPERATION_MODE_GAS_ACCUMULATION	シャドウ累積パス用のフラグメント処理
FRAGMENT_OPERATION_MODE_SHADOW_MAP	ガスの密度情報描画用のフラグメント処理

4.9.2. フレームバッファ

GD ライブラリでは RenderTarget オブジェクトと DepthStencilTarget オブジェクトでフレームバッファ(カラーバッファとデプスステンシルバッファ)を扱います。どちらのオブジェクトも 1 つの Texture2DResource オブジェクトとそれぞれのデスクリプタクラス(RenderTargetDescription と DepthStencilTargetDescription)から作成されます。なお、デスクリプタクラスの設定によって、Texture2DResource オブジェクトがミップマップデータを含んでいる場合に、オブジェクトの作成にどのレベルのミップマップデータを使用するかを指定することができます。

フレームバッファオブジェクト(RenderTarget オブジェクトおよび DepthStencilTarget オブジェクト)の作成と解放、パイプラインへの設定は以下の関数で行うことができます。

コード 4-52. フレームバッファオブジェクトの作成、解放、パイプラインへの設定

```
static nnResult nn::gd::OutputStage::CreateRenderTarget(
        const nn::gd::Texture2DResource* texture2DResource,
        const nn::gd::RenderTargetDescription* desc,
        nn::gd::RenderTarget** renderTarget);
static nnResult nn::gd::OutputStage::ReleaseRenderTarget(
        nn::gd::RenderTarget* renderTarget);
static nnResult nn::gd::OutputStage::SetRenderTarget(
        const nn::gd::RenderTarget* renderTarget);

static nnResult nn::gd::OutputStage::CreateDepthStencilTarget(
        const nn::gd::Texture2DResource* texture2DResource,
        const nn::gd::DepthStencilTargetDescription* desc,
        nn::gd::DepthStencilTarget** depthStencil);
static nnResult nn::gd::OutputStage::ReleaseDepthStencilTarget(
        nn::gd::DepthStencilTarget* depthStencil);
static nnResult nn::gd::OutputStage::SetDepthStencilTarget(
        const nn::gd::DepthStencilTarget* depthStencil);
```

関数名が～RenderTarget() のものはカラーバッファに対して、～DepthStencilTarget() のものはデプス(ステンシル)バッファに対して処理を行う関数です。

フレームバッファの設定は *desc* に指定するそれぞれのデスクリプタクラス(RenderTargetDescription または DepthStencilTargetDescription)にあらかじめ設定します。デスクリプタクラスに設定する情報の詳細は「4.9.2.1. フレームバッファオブジェクトのデスクリプタクラス」を参照してください。

フレームバッファに使用することのできる Texture2DResource オブジェクトのピクセルフォーマットは制限されています。詳しくは「3.1.1. Texture2DResource オブジェクトの作成」を参照してください。

作成されたフレームバッファオブジェクトは nn::gd::OutputStage::Set～() でフレームバッファに設定することができます。

不要になったフレームバッファオブジェクトは nn::gd::OutputStage::Release～() で解放してください。このとき、フレームバッファオブジェクトの管理情報のみが解放されます。Texture2DResource オブジェクトは解放されません。

4.9.2.1. フレームバッファオブジェクトのデスクリプタクラス

フレームバッファオブジェクト(RenderTarget オブジェクトと DepthStencilTarget オブジェクト)のデスクリプタクラス(RenderTargetDescription クラスと DepthStencilTargetDescription クラス)のメンバを紹介し、設定に際しての注意点などを説明します。

コード 4-53. フレームバッファオブジェクトのデスクリプタクラスの定義

```
class nn::gd::RenderTargetDescription
{
    u32 m_MipLevelIndex;
};
class nn::gd::DepthStencilTargetDescription
{
    u32 m_MipLevelIndex;
};
```

ミップマップレベルのインデックス(m_MipLevelIndex)

フレームバッファオブジェクトの作成に使用する Texture2DResource オブジェクトのミップマップレベルです。

4.9.2.2. フレームバッファオブジェクトの詳細情報の取得

詳細情報(TargetProperties クラス)には、作成時に指定された Texture2DResource オブジェクトの内容を元にしたフレームバッファに関する情報が格納されています。

詳細情報の取得で使用する関数は RenderTarget オブジェクトと DepthStencilTarget オブジェクトで異なりますが、取得するのは同じ TargetProperties クラスです。

コード 4-54. フレームバッファオブジェクトの詳細情報の取得

```
static nnResult nn::gd::OutputStage::GetRenderTargetProperties(
        const nn::gd::RenderTarget* renderTarget,
        nn::gd::TargetProperties* properties);
static nnResult nn::gd::OutputStage::GetDepthStencilTargetProperties(
        const nn::gd::DepthStencilTarget* depthStencil,
        nn::gd::TargetProperties* properties);
```

オブジェクトの作成時に指定したミップマップレベルによって、幅や高さ、ミップマップレベル、リソースデータの先頭アドレスが変化することはありますが、基本的に Texture2DResource オブジェクトの詳細情報と同じです。

コード 4-55. TargetProperties クラスの定義

```
class nn::gd::TargetProperties
{
    u32 m_Width;
    u32 m_Height;
    u32 m_PixelSize;
    nn::gd::Resource::NativeFormat m_Format;
    nn::gd::Memory::MemoryLayout m_MemLayout;
    nn::gd::Memory::MemoryLocation m_MemLocation;
    u8* m_MemAddr;
};
```

4.9.2.3. フレームバッファのクリア

指定した値でカラーバッファとデプス(ステンシル)バッファをクリアすることができます。

コード 4-56. フレームバッファのクリア

```
static nnResult nn::gd::Memory::ClearTargets(
        const nn::gd::RenderTarget* renderTarget,
        const nn::gd::DepthStencilTarget* depthStencilTarget,
        const u8 ColorRGBA[4], float depth, u8 stencil);
```

この関数内で nngxAddMemoryFillCommand() を呼び出していますので、実際の処理はコマンドリストの実行時に行われます。

4.9.2.4. カラーマスク

nn::gd::OutputStage::SetColorWriteMask() でカラーマスクの設定を変更し、カラーバッファへの書き込みを制御することができます。

コード 4-57. カラーマスクの設定関数

```
static void nn::gd::OutputStage::SetColorWriteMask(u32 mask);
```

mask には、以下のフラグの論理和を指定します。

表 4-23. カラーマスクのフラグ

定義	説明
COLOR_WRITE_MASK_ENABLE_RED	赤成分の書き込みを有効に設定
COLOR_WRITE_MASK_ENABLE_GREEN	緑成分の書き込みを有効に設定
COLOR_WRITE_MASK_ENABLE_BLUE	青成分の書き込みを有効に設定
COLOR_WRITE_MASK_ENABLE_ALPHA	アルファ成分の書き込みを有効に設定
COLOR_WRITE_MASK_ENABLE_ALL	すべての成分の書き込みを有効に設定(デフォルト)

4.9.3. デプステスト、ステンシルテスト

デプステストとステンシルテストの設定はステートオブジェクト(DepthStencilState オブジェクト)を作成し、アウトプットステージに設定する DepthStencilState オブジェクトを切り替えることでデプステストとステンシルテストに関する設定を一括で変更することができます。

DepthStencilState オブジェクトの作成と解放、パイプラインへの設定は以下の関数で行うことができます。

コード 4-58. DepthStencilState オブジェクトの作成、解放、パイプラインへの設定

```
static nnResult nn::gd::OutputStage::CreateDepthStencilState(
        const nn::gd::DepthStencilStateDescription* desc,
        nn::gd::DepthStencilState** DepthStencilState);
static nnResult nn::gd::OutputStage::ReleaseDepthStencilState(
        nn::gd::DepthStencilState* DepthStencilState);
static nnResult nn::gd::OutputStage::SetDepthStencilState(
        const nn::gd::DepthStencilState* state, u8 stencilRef);
```

デプスとステンシルのテストの設定は *desc* に指定するデスクリプタクラス(DepthStencilStateDescription)にあらかじめ設定します。デスクリプタクラスに設定する情報の詳細は「4.9.3.1. DepthStencilState オブジェクトのデスクリプタクラス」を参照してください。

作成されたステートオブジェクトは nn::gd::OutputStage::SetDepthStencilState() でパイプラインに設定することができます。*stencilRef* には、ステンシルテストで使用する参照値を指定します。

不要になったステートオブジェクトは nn::gd::OutputStage::ReleaseDepthStencilState() で解放してください。このとき、ステートオブジェクトとして作成されたときに内部で確保されたメモリ領域も解放されます。

4.9.3.1. DepthStencilState オブジェクトのデスクリプタクラス

DepthStencilState オブジェクトのデスクリプタクラス(DepthStencilStateDescription クラス)のメンバを紹介し、設定に際しての注意点などを説明します。

コード 4-59. DepthStencilStateDescription クラスの定義

```
class nn::gd::DepthStencilStateDescription
{
    gdBool                                  m_DepthEnable;
    gdBool                                  m_StencilEnable;
    u8                                      m_StencilWriteMask;
    u8                                      m_StencilReadMask;
    nn::gd::OutputStage::DepthWriteMask     m_DepthMask;
    nn::gd::OutputStage::DepthFunction      m_DepthFunc;
    nn::gd::OutputStage::StencilFunction    m_StencilFunc;
    nn::gd::OutputStage::StencilOperation   m_StencilFail;
    nn::gd::OutputStage::StencilOperation   m_StencilZFail;
    nn::gd::OutputStage::StencilOperation   m_StencilZPass;

    DepthStencilStateDescription();
    void ToDefault();
};
```

デプステストの有効化(m_DepthEnable)

デプステストを有効にする場合は GD_TRUE を設定します。

デプステストのライトマスク(m_DepthMask)

デプステストのライトマスクを設定します。

デプステストの比較関数（m_DepthFunc）

デプステストの比較関数を設定します。

ステンシルテストの有効化（m_StencilEnable）

ステンシルテストを有効にする場合は GD_TRUE を設定します。

ステンシルテストのライトマスク（m_StencilWriteMask）

ステンシルテストのライトマスクを設定します。

ステンシルテストのリードマスク（m_StencilReadMask）

ステンシルテストのリードマスクを設定します。

ステンシルテストの比較関数（m_StencilFunc）

ステンシルテストの比較関数を設定します。

ステンシルテストの失敗時の処理（m_StencilFail）

ステンシルテストの失敗時の処理を設定します。

ステンシルテストの成功時の処理（m_StencilZFail）

ステンシルテストに成功したが、デプステストに失敗したときの処理を設定します。

ステンシルテストとデプステストの成功時の処理（m_StencilZPass）

ステンシルテストとデプステストの両方が成功したときの処理を設定します。

デフォルトの設定（ToDefault()）

ToDefault() で、すべてのメンバをデフォルトの値に変更することができます。この設定はコンストラクタでも行われています。

表 4-24. DepthStencilStateDescription クラスのデフォルト設定

メンバ	デフォルトの設定内容
m_DepthEnable	GD_FALSE
m_DepthMask	DEPTH_WRITE_MASK_ALL
m_DepthFunc	DEPTH_FUNCTION_LESS
m_StencilEnable	GD_FALSE
m_StencilWriteMask m_StencilReadMask	0xFF
m_StencilFunc	STENCIL_FUNCTION_NEVER
m_StencilFail m_StencilZFail m_StencilZPass	STENCIL_OPERATION_KEEP

4.9.4. アルファテスト

アルファテストの設定は nn::gd::OutputStage::SetAlphaTest() で行います。

コード 4-60. アルファテストの設定

```
static void nn::gd::OutputStage::SetAlphaTest(
        gdBool enable, nn::gd::OutputStage::AlphaFunction func, u8 alphaRef);
```

アルファテストを有効にする場合は *enable* に GD_TRUE、*func* に比較関数、*alphaRef* に参照値を指定してください。デフォルトの設定は、引数の順に GD_FALSE、ALPHA_FUNCTION_NEVER、0 です。

4.9.5. 論理演算、ブレンディング

論理演算とブレンディングの設定はステートオブジェクト(BlendState オブジェクト)を作成し、アウトプットステージに設定する BlendState オブジェクトを切り替えることで論理演算とブレンディングに関する設定を一括で変更することができます。

BlendState オブジェクトの作成と解放、パイプラインへの設定は以下の関数で行うことができます。

コード 4-61. BlendState オブジェクトの作成、解放、パイプラインへの設定

```
static nnResult nn::gd::OutputStage::CreateBlendState(
        const nn::gd::BlendStateDescription* desc,
        nn::gd::BlendState** blendState);
static nnResult nn::gd::OutputStage::ReleaseBlendState(
        nn::gd::BlendState* blendState);
static nnResult nn::gd::OutputStage::SetBlendState(
        const nn::gd::BlendState* blendState);
```

論理演算とブレンディングの設定は *desc* に指定するデスクリプタクラス(BlendStateDescription)にあらかじめ設定します。デスクリプタクラスに設定する情報の詳細は後述します。

作成された BlendState オブジェクトは nn::gd::OutputStage::SetBlendState() でパイプラインに設定することができます。

不要になったステートオブジェクトは nn::gd::OutputStage::ReleaseBlendState() で解放してください。このとき、ステートオブジェクトとして作成されたときに内部で確保されたメモリ領域も解放されます。

ブレンディングの設定のうち、ブレンドカラーの設定だけは以下の関数で行います。デフォルトのブレンドカラーは(0, 0, 0, 0)です。

コード 4-62. ブレンドカラーの設定

```
static void nn::gd::OutputStage::SetBlendColor(u8 R, u8 G, u8 B, u8 A);
```

4.9.5.1. BlendState オブジェクトのデスクリプタクラス

BlendState オブジェクトのデスクリプタクラス(BlendStateDescription クラス)のメンバを紹介し、設定に際しての注意点などを説明します。

コード 4-63. BlendStateDescription クラスの定義

```
class nn::gd::BlendStateDescription
{
    nn::gd::OutputStage::BlendType m_BlendType;
    nn::gd::OutputStage::LogicOperator m_LogicOp;
    nn::gd::OutputStage::BlendFunction m_SrcRgb;
    nn::gd::OutputStage::BlendFunction m_DstRgb;
    nn::gd::OutputStage::BlendFunction m_SrcAlpha;
```

```
    nn::gd::OutputStage::BlendFunction m_DstAlpha;
    nn::gd::OutputStage::BlendEquation m_EqRgb;
    nn::gd::OutputStage::BlendEquation m_EqAlpha;

    BlendStateDescription();
    void ToDefault();

    void SetBlendFunc(
        nn::gd::OutputStage::BlendFunction src,
        nn::gd::OutputStage::BlendFunction dst,
        nn::gd::OutputStage::BlendEquation eq);
    void SetBlendFunc(
        nn::gd::OutputStage::BlendFunction srcRgb,
        nn::gd::OutputStage::BlendFunction dstRgb,
        nn::gd::OutputStage::BlendFunction srcAlpha,
        nn::gd::OutputStage::BlendFunction dstAlpha,
        nn::gd::OutputStage::BlendEquation eqRgb,
        nn::gd::OutputStage::BlendEquation eqAlpha);

    void SetBlendMode_DefaultBlending();
    void SetBlendMode_NoBlend();

    void SetLogicOperatorMode(nn::gd::OutputStage::LogicOperator logicOp);
    void SetLogicOperatorMode_Default();
};
```

論理演算とブレンディングの選択（m_BlendType）

論理演算を使用する場合は BLEND_TYPE_LOGICOP、ブレンディングを使用する場合は BLEND_TYPE_BLENDING を設定します。

論理演算の演算方法（m_LogicOp）

論理演算使用時の演算方法です。SetLogicOperatorMode() で設定することができます。

SetLogicOperatorMode_Default() で設定した場合は LOGIC_OPERATOR_COPY が設定されます。

ブレンディングの重み係数（m_SrcRgb、m_DstRgb、m_SrcAlpha、m_DstAlpha）

ブレンディングの重み係数です。ソースとデスティネーションそれぞれに RGB 成分とアルファ成分の設定が存在します。

ブレンディングの計算式（m_EqRgb、m_EqAlpha）

ブレンディングの計算式です。RGB 成分とアルファ成分の設定が存在します。

ブレンディングの一括設定（SetBlendFunc()、SetBlendMode_NoBlend()）

SetBlendFunc() はブレンディングの設定を一括して行います。RGB 成分とアルファ成分を同じまたは個別に設定することができます。

SetBlendMode_NoBlend() はブレンディングが行われない（ソースそのままを出力する）設定に変更します。

表 4-25. SetBlendMode_NoBlend() によるブレンディング設定の変更

メンバ	設定
m_SrcRgb、m_SrcAlpha	BLEND_FUNCTION_ONE
m_DstRgb、m_DstAlpha	BLEND_FUNCTION_ZERO
m_EqRgb、m_EqAlpha	BLEND_EQUATION_ADD

デフォルトの設定(ToDefault())

ToDefault() で、すべてのメンバをデフォルトの値に変更することができます。この設定はコンストラクタでも行われています。

デフォルトの設定ではブレンディングを使用し、ブレンディングの設定は SetBlendMode_NoBlend() で行われる設定と同じです。

4.9.6. w バッファ、ポリゴンオフセット

w バッファとポリゴンオフセットの設定は nn::gd::OutputStage::SetDepthRangeMode() で行います。

コード 4-64. w バッファとポリゴンオフセットの設定

```
static void nn::gd::OutputStage::SetDepthRangeMode(
        nn::gd::OutputStage::DepthBufferType type,
        f32 depthRangeNear, f32 depthRangeFar, s32 offset);
```

type には、w バッファの機能を使用する場合は DEPTHBUFFER_LINEAR を指定し、w バッファの機能を使用せずに通常のデプス値の範囲を使用する場合は DEPTHBUFFER_RECIPROCAL_FUNCTION を指定します。

depthRangeNear と *depthRangeFar* には、w バッファの機能を使用しない場合はデプスのニア値とファー値を指定します。w バッファの機能を使用する場合は *depthRangeNear* に 0、*depthRangeFar* にスケール値を指定します。

ポリゴンオフセット機能を使用する場合は、*offset* にポリゴンオフセットのオフセット値を指定します。

4.9.7. ソフトシャドウ

ソフトシャドウのスケールとバイアスの設定は nn::gd::OutputStage::SetPenumbraScaleBias() で行います。この関数はイミディエート関数ですので、実行と同時にコマンドバッファへの 3D コマンドの蓄積を行います。

コード 4-65. ソフトシャドウの設定

```
static void nn::gd::OutputStage::SetPenumbraScaleBias(f32 Scale, f32 Bias);
```

5. レイアウト変換

ブロックフォーマットとリニアフォーマット間のレイアウト変換を行う関数は nn::gd::Memory クラスに定義されています。

5.1. ブロックフォーマットからリニアフォーマットへの変換

ブロックフォーマットからリニアフォーマットへの変換を行うための関数が用意されています。.

コード 5-1. ブロックフォーマットからリニアフォーマットへの変換関数

```
static nnResult nn::gd::Memory::CopyTexture2DResourceBlockToLinear(
        const nn::gd::Texture2DResource* source,
        s32 srcMipLevelIndex, u32 srcOffsetY, s32 srcCountRow,
        const nn::gd::Texture2DResource* dest,
        s32 dstMipLevelIndex, u32 dstOffsetY,
        nn::gd::Memory::DownScalingMode downScalingMode, gdBool yFlip);
static nnResult nn::gd::Memory::CopyTexture2DResourceBlockToLinear(
        const nn::gd::Texture2DResource* source,
        s32 srcMipLevelIndex, u32 srcOffsetY,
        u8* dstAddr, u32 dstWidth, u32 dstHeight, u32 dstFormat,
        nn::gd::Memory::DownScalingMode downScalingMode, gdBool yFlip);
```

source に指定された Texture2DResource オブジェクトのリソースデータが、ブロックフォーマットからリニアフォーマットに変換されて転送されます。また、変換するミップマップレベルを *srcMipLevelIndex* に指定することができます。-1 を指定した場合は Texture2DResource オブジェクトの最大のミップマップレベルが使用されます。

srcOffsetY には変換を開始する Y 座標値を指定します。アドレッシングの違いを関数内で吸収しますので、リソースデータの座標値そのままを指定してください。

転送先に Texture2DResource オブジェクトを指定する関数では、*srcCountRow* に変換するライン数を指定します。-1 を指定した場合は Texture2DResource オブジェクトの高さが使用されます。*dstMipLevelIndex* や *dstOffsetY* には、転送先のミップマップレベルと転送開始位置の Y 座標値を指定します。

転送先にアドレスを指定する関数では、*dstAddr* に先頭アドレス、*dstWidth* に幅、*dstHeight* に高さ、*dstFormat* にピクセルフォーマットを指定します。

downScalingMode にはアンチエイリアスの指定を、*yFlip* には縦方向のフリップを有効にする場合に GD_TRUE を指定します。

これらの関数は内部で nngxAddB2LTransferCommand() を呼び出していますので、実際の変換処理はコマンドリストの実行時に行われます。

ブロックフォーマットからリニアフォーマットへの変換関数は、主にカラーバッファからディスプレイバッファへの転送時に使用します。コードの例は「2.7. 描画結果の表示」にあります。

5.2. リニアフォーマットからブロックフォーマットへの変換

リニアフォーマットからブロックフォーマットへの変換を行うための関数が用意されています。

コード 5-2. リニアフォーマットからブロックフォーマットへの変換関数

```
static nnResult nn::gd::Memory::CopyTexture2DResourceLinearToBlock(
        const nn::gd::Texture2DResource* source,
        s32 srcMipLevelIndex, u32 srcOffsetY, s32 srcCountRow,
        const nn::gd::Texture2DResource* dest,
        s32 dstMipLevelIndex, u32 dstOffsetY);
static nnResult nn::gd::Memory::CopyTexture2DResourceLinearToBlock(
        u8* srcAddr, u32 width, u32 height,
        const nn::gd::Texture2DResource* dest,
        s32 dstMipLevelIndex, u32 dstOffsetY);
```

dest に指定された Texture2DResource オブジェクトに、転送元のデータがリニアフォーマットからブロックフォーマットに変換されて転送されます。また、変換されたデータを受け取るミップマップレベルを *dstMipLevelIndex* に指定することができます。-1 を指定した場合は Texture2DResource オブジェクトの最大のミップマップレベルが使用されます。また、*dstOffsetY* にはデータの受け取りを開始する Y 座標値を指定します。アドレッシングの違いを関数内で吸収しますので、リソースデータの座標値そのままを指定してください。

転送元に Texture2DResource オブジェクトを指定する関数では、*srcCountRow* に変換するライン数を指定します。-1 を指定した場合は Texture2DResource オブジェクトの高さが使用されます。*srcMipLevelIndex* と *srcOffsetY* には、転送元のミップマップレベルと転送開始位置の Y 座標値を指定します。

転送元にアドレスを指定する関数では、*srcAddr* に先頭アドレス、*width* に幅、*height* に高さを指定します。

これらの関数は内部で nngxAddL2BTransferCommand() を呼び出していますので、実際の変換処理はコマンドリストの実行時に行われます。

6. パケットレコーディング

パケットレコーディング機能を用いると、ひとまとまりのコマンドパケットを記録しておき、あとで再生するような使い方が可能です。この機能は、カメラやライトの設定を除いた大部分が同じシーンを 2 回描画する立体視表示のように、繰り返し同じ描画を行う処理で有効に活用することができます。

パケットレコーディング機能は以下の関数を用いて制御します。

コード 6-1. パケットレコーディング機能の制御に使用する関数

```
nn::gd::System::StartRecordingPackets(
        u32* forceDirtyModuleFlag,
        nn::gd::RecordingPacketUsage usage = RECORD_COMMAND_LIST_COPY);
nn::gd::System::StopRecordingPackets(nn::gd::RecordedPacketId** packetId);
nn::gd::System::ReplayPackets(nn::gd::RecordedPacketId* packetId,
        u32* forceDirtyModuleFlag);
nn::gd::System::ReleasePackets(nn::gd::RecordedPacketId* packetId);
```

3D コマンドバッファに出力される 3D コマンドと、リクエストバッファに出力されるコマンドリクエスト(「7.1. nngx 関数の内部使用」を参照)の両方が記録対象となります。

不要になったコマンドパケットは `nn::gd::System::ReleasePackets()` で解放することができます。

出力される 3D コマンドへの内部状態の影響

GD ライブラリの内部状態によっては呼び出された関数が必ずしも 3D コマンドを出力するとは限らず、コマンドパケットの記録時に期待した通りの挙動にならないことがあります。再生時でも同じように、GPU の状態と GD ライブラリの内部状態が食い違ってしまうことがあります。この問題を回避するため、`nn::gd::System::StartRecordingPackets()` と `nn::gd::System::ReplayRecordingPackets()` はそれぞれフラグを引数として与えるようになっており、関数を実行する前にどのモジュールを「dirty」として扱うべきかを指示することができます。

最も安全なフラグの設定は `MODULE_ALL` を指定して、すべてのモジュールを「dirty」として扱うことです。この指定ではすべての 3D コマンドが出力され、状態が破壊されることは絶対にありません。もし、どのモジュールの状態が記録／再生の対象になっているか、状態が衝突するかどうかについて利用する側で分かっていれば、それに合わせて必要最小限のモジュールのみを「dirty」に設定することで性能を改善することができます。

コマンドパケットの編集

記録されたコマンドパケットを再生前に編集することができます。3D コマンドバッファの更新位置は、パケットの記録中に(`nn::gd::System::StartRecordingPackets()` を呼び出して `nn::gd::System::StopRecordingPackets()` を呼び出すまでの間に) `nn::gd::System::GetCommandBufferOffset()` で取得した 3D コマンドバッファの出力先のオフセット値と、記録完了後に `nn::gd::System::GetCommandBufferBaseAddress()` で取得する 3D コマンドバッファのベースアドレスを組み合わせて算出します。ただし、3D コマンドの書き換えに実際に使用することができるのは、呼び出しと同時に 3D コマンドを 3D コマンドバッファに出力するイミディエート関数だけです。

引数 usage による 3D コマンドバッファのコピー指定

`nn::gd::System::StartRecordingPackets()` の *usage* に `RECORD_COMMAND_LIST_COPY` を指定したときは、引数が追加される以前と同様に、3D コマンドバッファのコピーを作成します。そのため、記録されたコマンドパケットの 3D コマンドを書き換えることができます。

一方、*usage* に `RECORD_COMMAND_LIST_NO_COPY` を指定したときは、3D コマンドバッファのコピーが作成されないた

め、処理時間が短くなりますが 3D コマンドを書き換えることができません。また、コピー元の 3D コマンドバッファをクリアすると、コマンドパケットの実行ができなくなります。

usage に RECORD_3D_COMMAND_BUFFER_FOR_JUMP を指定してパケットの記録を開始した場合、nn::gd::System::ReplyPackets() は 3D コマンドバッファのコピーを作成し、nngxAddSubroutineCommand() によるジャンプコマンドの挿入とジャンプによる実行を行います。

7. GX ライブラリとの関係

ライブラリの関数内で呼び出している nngx 関数や、nngx 関数との関係で注意が必要となる特殊なケースを説明します。

7.1. nngx 関数の内部使用

GD ライブラリの関数の中には、内部で nngx 関数を呼び出すものが存在します。以下の表に記されている関数は、「条件」欄にある条件が満たされてエラーも発生していないときに、内部で nngx 関数を呼び出します。ただし、同一の関数でも条件によって異なる nngx 関数が呼び出されることがあります。

条件が「常に」と記載されているものは、エラーが発生した場合を除いて、示された nngx 関数が常に呼び出されます。

表 7-1. 内部で nngx 関数を呼び出す GD ライブラリの関数

GD 関数	条件	nngx 関数
System:: StartRecordingPackets()	常に	nngxStartCmdlistSave()
System:: StopRecordingPackets()	常に	nngxSplitDrawCmdlist() nngxStopCmdlistSave() nngxExportCmdlist()
System::ReplayPackets()	常に	nngxImportCmdlist()
System:: Execute3DCommandList()	常に	nngxSplitDrawCmdlist()
Resource:: CreateTexture2DResource()	初期データがあり、VRAM に確保したメモリにコピー	nngxAddVramDmaCommand()
	初期データがあり、デバイスメモリに確保したメモリにコピー	nngxUpdateBuffer()
	初期データがあり、ミップマップを自動生成	nngxFilterBlockImage()
Memory::GenerateMipMaps()	常に	nngxFilterBlockImage()
	描画依存関係がある	nngxSplitDrawCmdlist()
Resource:: CreateVertexBufferResource()	初期データがあり、VRAM に確保したメモリにコピー	nngxAddVramDmaCommand()
	初期データがあり、デバイスメモリに確保したメモリにコピー	nngxUpdateBuffer()
Memory:: CopyTextureSubResource()	常に	nngxAddBlockImageCopyCommand()
	描画依存関係がある	nngxSplitDrawCmdlist()
Memory:: CopyVertexBufferSubResource()	常に	nngxAddBlockImageCopyCommand()
Memory:: CopyTexture2DResource BlockToLinear()	常に	nngxAddB2LTransferCommand()
	描画依存関係がある	nngxSplitDrawCmdlist()

Memory:: CopyTexture2DResource LinearToBlock()	常に	nngxAddL2BTransferCommand()
	描画依存関係がある	nngxSplitDrawCmdlist()
Memory::ClearTargets()	常に	nngxAddMemoryFillCommand()
	描画依存関係がある	nngxSplitDrawCmdlist()
Memory:: ClearTexture2DResource()	常に	nngxAddMemoryFillCommand()
	描画依存関係がある	nngxSplitDrawCmdlist()
Resource:: UnmapTexture2DResource	デバイスメモリにあるリソースを書き込みありでマッピング	nngxUpdateBuffer()
Resource:: UnmapVertexBufferResource	デバイスメモリにあるリソースを書き込みありでマッピング	nngxUpdateBuffer()

7.2. nngx 関数との相互作用

GD ライブラリを利用する上で、nngx 関数との関係に注意が必要となる特殊なケースを示します。

7.2.1. 初期データを持つテクスチャを VRAM 上に作成

初期データを用いて VRAM 上にリソースを作成することになるため、初期データをメインメモリから VRAM に転送する DMA 転送コマンドがコマンドリクエストキューに追加されます。GD ライブラリではコマンドリクエストの実行を管理していないため、初期データを解放する前に DMA 転送の完了をアプリケーションで確認する必要があります。

以下のコード例のように、nngxWaitCmdlistDone() を呼べばすべてのコマンドリクエストの完了を待つことができます。ただしコマンドリストをダブルバッファ化している場合は、現在のコマンドリクエストキューに追加された DMA 転送コマンドが次のフレームまで実行されないことに注意してください。

コード 7-1. 初期データを持つテクスチャを VRAM 上に作成

```
nn::gd::Texture2DResource* texture2DResource = 0;
nn::gd::Texture2DResourceDescription Text2DResDesc =
{
    width, height, 1, nn::gd::Resource::NATIVE_FORMAT_RGB_888,
    nn::gd::Memory::LAYOUT_BLOCK_8, nn::gd::Memory::VRAMA
};
nn::gd::Resource::CreateTexture2DResource(
        &Text2DResDesc, data, GD_TRUE, &texture2DResource);

// リソースはVRAM上に作成されるため、解放前にDMA転送の完了を確認しなければならない
nngxWaitCmdlistDone();

free(data);
```

7.2.2. 描画結果へのアクセス

レンダリング後にカラーバッファやデプス(ステンシル)バッファ、テクスチャの内容にアクセスして描画結果を直接書き換えたい場合、CPU から直接 VRAM 上のデータにアクセスすることはできませんので、以下のような手順が必要となります。

- 一時的なリソースをデバイスメモリに確保する

- VRAM 上にあるリソースを一時的なリソースにコピーする(VRAM からのデータ転送)
- 一時的なリソースを書き換える
- VRAM 上にあるリソースに一時的なリソースの内容を書き戻す(VRAM へのデータ転送)

VRAM からのデータ転送と VRAM へのデータ転送はコマンドリクエストによって実行されます。そのため、リソースを書き換える前や一時的に確保したリソースを解放する前に、コマンドリクエストの実行が完了するのを待たなければなりません。

実際のコードでは以下のような流れになります。

コード 7-2. 描画結果へのアクセス

```
// デバイスメモリ(FCRAM)上に一時的なリソースを確保
nn::gd::Texture2DResource* tmpResource= 0;
nn::gd::Texture2DResourceDescription tmpResourceDesc =
{
    s_RenderTextureWidth, s_RenderTextureHeight, 1,
    nn::gd::Resource::NATIVE_FORMAT_RGBA_8888,
    nn::gd::Memory::LAYOUT_BLOCK_8,
    nn::gd::Memory::FCRAM
};
res = nn::gd::Resource::CreateTexture2DResource(
        &tmpResourceDesc, 0, GD_FALSE, &tmpResource);

// VRAM 上のリソースを一時的なリソースにコピー
nn::gd::Memory::Rect memRect(0, 0, -1, -1);
res = nn::gd::Memory::CopyTextureSubResource(
        s_texture2DResource_ColorBuffer, 0, memRect, tmpResource, 0, 0, 0);
// アクセスするためにリソースをマップする
u8* data;
nn::gd::Resource::MapTexture2DResource(
        tmpResource, 0, nn::gd::Resource::MAP_READ_WRITE, (void**)&data);
// データにアクセスする前にコマンドの実行完了待ちが必要
nngxWaitCmdlistDone();

// ここでマップしたリソースに書き込む
for (int y=0; y<s_RenderTextureHeight*s_RenderTextureWidth>>1; y++)
{
    data[k+] = 255; data[k] = 255; data[k] = 255; data[k+] = 255;
}

// マッピング解除時に一時的なリソースへの変更が確定される
nn::gd::Resource::UnmapTexture2DResource(tmpResource);

// 一時的なリソースから VRAM 上のリソースにコピー
nn::gd::Memory::CopyTextureSubResource(
        tmpResource, 0, memRect, s_texture2DResource_ColorBuffer, 0, 0, 0);
```

8. ほかのライブラリやフレームワークとの連携

ほかのグラフィックス関連のライブラリやフレームワークと平行して、GD ライブラリを利用する際の注意点を説明します。

8.1. 初期化時の注意点

ほかのフレームワーク(NintendoWare など)と同じコマンドリストを使用する場合は、コマンドリストの初期化を複数回呼ぶことがないように注意してください。

コード 2-1 で示したコードを例にすると、NintendoWare でコマンドリスト関連の初期化を済ませていた場合は GD ライブラリの初期化関数(`nn::gd::System::Initialize()`)のみを呼び出します。

8.2. nn::gd::System::ForceDirty() の利用

GD ライブラリの関数以外で 3D コマンドを 3D コマンドバッファに書き込んだ場合、GD ライブラリの内部状態と GPU との同期が取れなくなる可能性があります。そのような場合には、`nn::gd::System::ForceDirty()` を呼び出していくつかのモジュールを「dirty」にし、対応するモジュールのすべての 3D コマンドが再送されるようにします。

`ForceDirty()` によって「dirty」となったモジュールは、描画コマンドが実行されたタイミングで 3D コマンドを生成します。すぐに 3D コマンドを生成する場合は、`nn::gd::System::FlushDirtyModule()` を呼び出してください。

イミディエート関数で設定されたレジスタの内容は GD ライブラリ内では保持していません。そのため、イミディエート関数でレジスタの内容を書き換えていた場合は、3D コマンドが生成されたあとにイミディエート関数を呼び出して、再度レジスタの内容を設定する必要があります。

内部状態については「1.2.2. モジュールの状態管理」を参照してください。

8.3. コマンドリストの二重化

GD ライブラリの関数の中には、コマンドリクエストをキューイングするものがあります。コマンドリストが実行されるまでキューイングされたコマンドリクエストは実行されることはありませんので、そのコマンドリストが実行される前にコマンドリクエストで扱ったリソースを書き換えたり削除したりしないように特に注意してください。

2 つのコマンドリストを利用しているフレームワークでも同様の注意が必要です。これは、コマンドリストを二重化している場合に、次のフレームの作成に使用するコマンドリストにコマンドリクエストがキューイングされるためです。

8.4. ダミーコマンドの挿入

`nn::gd::System::AddDummyCommands()` を呼び出して、GPU に命令として認識されないダミーコマンドをコマンドバッファに追加することができます。ダミーコマンドは、次に実行するコマンドまで Wait しなければならないときや、コマンドバッファのサイズ調整に利用します。

9. 付録

9.1. イミディエート関数一覧

イミディエート関数の一覧を以下に示します。

表 9-1. イミディエート関数一覧

クラス	関数
nn::gd::CombinerStage	SetTextureCombinerUnitConstantColor()
nn::gd::FogStage	UploadFogLookUpTableFloat()
	UploadFogLookUpTableNative()
	UploadGasLookUpTableFloat()
	UploadGasLookUpTableNative()
nn::gd::LightingStage	SetGlobalColorAmbient()
	UploadLookUpTableFloat()
	UploadLookUpTableNative()
nn::gd::Light	SetColorAmbient()
	SetColorDiffuse()
	SetColorSpecular0()
	SetColorSpecular1()
	SetPosition()
	SetDirection()
	SetSpotDirection()
	SetLightType()
	SetDistanceAttenuationScaleBias()
	EnableTwoSideDiffuse()
	EnableGeomFactor0()
	EnableGeomFactor1()
	UploadLookUpTableFloat()
	UploadLookUpTableNative()
nn::gd::ProceduralTextureStage	UploadLookUpTableRgbMapFloat()
	UploadLookUpTableRgbMapNative()
	UploadLookUpTableAlphaMapFloat()
	UploadLookUpTableAlphaMapNative()
	UploadLookUpTableNoiseMapFloat()

	UploadLookUpTableNoiseMapNative()
	UploadLookUpTableColorMapFloat()
	UploadLookUpTableColorMapNative()
nn::gd::ShaderStage	SetFloatConstantBuffer()
nn::gd::OutputStage	SetPenumbraScaleBias()
nn::gd::RasterizerStage	SetCulling()
	SetViewport()
	EnableClippingPlane()
	SetClippingPlane()
	EnableScissor()
	SetScissor()

9.2. 関数内で確保されるメモリのサイズ

下記のサイズは GX ライブラリの初期化時に指定したアロケータへ渡される値です。

備考欄に記載されていない限り、用途は NN_GX_MEM_SYSTEM、メモリの指定は NN_GX_MEM_FCRAM です。

表 9-2. 関数内で確保されるメモリ

関数	サイズ	備考
System::Initialize()	24 Byte	管理情報
	12 Byte	管理情報
	12 Byte	管理情報
	28 Byte	管理情報
	12 Byte	管理情報
	1536 Byte	管理情報
	52 Byte	管理情報
	28 Byte	管理情報
	12 Byte	管理情報
	1664 Byte	管理情報
	56 Byte	管理情報
	28 Byte	管理情報
	12 Byte	管理情報
	2688 Byte	管理情報
	28 Byte	管理情報
	12 Byte	管理情報

	14208 Byte	管理情報
	616 Byte	管理情報
	28 Byte	管理情報
	12 Byte	管理情報
	1664 Byte	管理情報
	60 Byte	管理情報
	28 Byte	管理情報
	12 Byte	管理情報
	640 Byte	管理情報
	328 Byte	3D コマンド (ジオメトリシェーダを使用する設定に共用プロセッサを変更する)
	328 Byte	3D コマンド (ジオメトリシェーダを使用しない設定に共用プロセッサを変更する)
Resource::CreateTexture2DResource()	52 Byte	管理情報
	16 Byte	管理情報
	データによる	初期データあり、かつコピーなしでなければ、指定されたメモリに NN_GX_MEM_TEXTURE で確保されます
Resource:: CreateTexture2DResourceCastFrom()	64 Byte	管理情報
Resource::Helper:: ConvertTextureResourceToNativeFormat()	データによる	変換で使用する一時的なバッファのため、関数の終了時に解放されます
Resource::Helper::ConvertCompressed TextureResourceToNativeFormat()	データによる	変換で使用する一時的なバッファのため、関数の終了時に解放されます
Resource::CreateRenderTarget()	48 Byte	管理情報
Resource::CreateDepthStecilTarget()	52 Byte	管理情報
TextureStage::CreateTexture2D()	64 Byte	管理情報
TextureStage::CreateTextureCube()	112 Byte	管理情報
ShaderStage::CreateShaderBinary()	28 Byte	管理情報
	データによる	管理情報
	データによる	管理情報
	データによる	管理情報
	データによる	管理情報
	データによる	管理情報
	データによる	管理情報
	20 Byte	管理情報
	データによる	3D コマンド
	データによる	3D コマンド

ShaderStage::CreateShader()	20 Byte	管理情報
	データによる	3D コマンド
ShaderStage::CreateShaderPipeline()	148 Byte	管理情報
	データによる	3D コマンド
	データによる	ユニフォーム設定の値 (頂点シェーダ)
	データによる	ユニフォーム設定の値 (ジオメトリシェーダ)
VertexInputStage::CreateInputLayout()	204 Byte	管理情報
	データによる	InputElementDescription クラス(複数の場合あり)
Resource::CreateVertexBufferResource()	24 Byte	管理情報
	データによる	初期データあり、かつコピーなしでなければ、指定されたメモリに NN_GX_MEM_VERTEXBUFFER で確保されます
CombinerStage:: CreateTextureCombinerState()	616 Byte	管理情報
	92 Byte	管理情報
TextureStage::CreateSamplerState()	60 Byte	管理情報
OutputStage::CreateDepthStecilState()	52 Byte	管理情報
OutputStage::CreateBlendState()	56 Byte	管理情報
System::StartRecordingPackets()	36 Byte	管理情報
System::StopRecordingPackets()	データによる	記録された 3D コマンド

9.3. デバッグサポート機能

GD ライブラリでは、描画時の動作を変更するフィルタ機能など、デバッグに便利な機能を用意しています。

9.3.1. フィルタ機能

フィルタ機能では、フラグ指定で描画時の動作を変更することができます。フィルタ機能は、Debug または Development でコンパイルされたときのみ有効です。

コード 9-1. フィルタ機能の指定関数

```
static nnResult nn::gd::System::Debug::SetFilter(
                u32 filterFlag, u32 drawStartIndex, s32 drawCount);
```

drawStartIndex で指定された回数の描画が行われたあと、*drawCount* で指定された回数の描画が行われるまでフィルタ機能が有効になります。

filterFlag には、以下のフラグから有効にする機能を論理和で指定します。

表 9-3. フィルタ機能のフラグ

フラグ	説明
FILTER_NONE	単独で指定するとフィルタ機能が無効になります。
FILTER_TEXTURE_8X8	描画に使用するテクスチャすべてを固定の 8x8 テクスチャに変更します。
FILTER_TEXTURE_CUSTOM	描画に使用するテクスチャすべてを SetCustomTexture() で指定されたカスタムテクスチャに変更します。カスタムテクスチャが指定されていない場合は、固定の 8×8 テクスチャが使用されます。
FILTER_SAMPLER_STATE_FILTER_NEAREST_MIPMAP_NEAREST	テクスチャの縮小時フィルタの設定を、強制的に GL_NEAREST_MIPMAP_NEAREST に設定します。
FILTER_SAMPLER_STATE_FILTER_LINEAR_MIPMAP_NEAREST	テクスチャの縮小時フィルタの設定を、強制的に GL_LINEAR_MIPMAP_NEAREST に設定します。
FILTER_VIEWPORT_1X1	ビューポートの設定を 1×1 に変更します。
FILTER_SIMPLE_COMBINER	コンバイナの設定を変更し、すべてのピクセルが緑で出力されます。
FILTER_DISABLE_BLENDING	ブレンディングを無効にします。
FILTER_DISABLE_DEPTH_STENCIL_TEST	デプス、ステンシルのテストを無効にします。
FILTER_DISABLE_ALPHA_TEST	アルファテストを無効にします。
FILTER_LIGHTING_STAGE_1_LIGHT_MAX	ライティングの最大数を 1 に変更します。複数のライトを有効に設定した場合は、最初に有効となったライトのみが有効になります。
FILTER_LIGHTING_STAGE_SIMPLE_LAYER_CONFIGURATION	ライティングに必要なサイクル数が最小となるレイヤコンフィグレーションに変更します。
FILTER_DISABLE_DRAW	描画を無効にします。
FILTER_VISUALIZE_OVERDRAW	描画されたピクセルが徐々に白くなるようにブレンディングの設定を追加し、重複して描画が行われるピクセルを視覚化します。デプス、ステンシルを無効にするフィルタを併用することで、テストで破棄される(破棄されない)ピクセルを確認することができます。
FILTER_DISABLE_CULLING	カリングを無効に変更します。
FILTER_ENABLE_CULLING_CLOCKWISE	カリングを時計回りに変更します。
FILTER_ENABLE_CULLING_COUNTERCLOCKWISE	カリングを反時計回りに変更します。

9.3.2. ミップマップレベルの視覚化

nn::gd::System::Debug::ColorizeMipmaps() は、テクスチャリソースを指定されたミップマップレベルごとに指定された色で塗りつぶし、ミップマップレベルの視覚化をサポートします。

コード 9-2. ミップマップレベルの視覚化をサポートする関数

```
static nnResult nn::gd::System::Debug::ColorizeMipmaps(
                nn::gd::Texture2DResource* texture2DResource,
                s32 minMipLevelIndex, s32 maxMipLevelIndex, u8 colors[][4]);
```

minMipLevelIndex と *maxMipLevelIndex* にはそれぞれ、塗りつぶすミップマップレベルの最小のインデックスと最

大のインデックスを指定します。どちらの引数も、-1 が指定された場合は *texture2DResource* で指定されたテクスチャリソースが持つ、最大のミップマップレベルを指定したことになります。

colors で指定されたカラーは、*minMipLevelIndex* で指定されたミップマップレベルから順に使用されます。つまり、*minMipLevelIndex* に指定されたミップマップレベルのテクスチャが *colors[0]* に指定されたカラーで塗りつぶされ、次のミップマップレベルのテクスチャは *colors[1]* に指定されたカラーで塗りつぶされます。

colors に NULL を指定したときは、デフォルトのカラー配列を使用して塗りつぶします。デフォルトのカラー配列については、関数リファレンスを参照してください。

更新履歴

Version 1.1 2016-05-10

変更

- 1.1.1. コマンドリスト
  - コマンドリクエストを発行する関数はコア0でのみ呼び出し可能なことを追記しました。

Version 1.0 2014-09-04

追加/変更

- 初版