



第3部

Ninja2ライブラリ編

この章では、グラフィック等を扱う Ninja2 を
ライブラリ全般について解説します。

目次

| | | |
|----------|------------------------------------|-----------|
| 1 | Ninja2 ライブラリについて | 8 |
| 1.1 | 基本事項 | 8 |
| 1.1.1 | Ninja2 ライブラリで使用するファイル拡張子の説明 | 8 |
| 1.1.2 | Ninja ライブラリで使用するファイル | 9 |
| 1.1.3 | その他のファイル | 9 |
| 1.2 | モデル構造について | 11 |
| 1.2.1 | Chunk Model 形式 | 11 |
| 1.3 | Ninja2 ライブラリの概要 | 12 |
| 1.4 | 画面モード（画面解像度設定）について | 13 |
| 1.4.1 | ケーブルの自動判別について | 13 |
| 1.4.2 | フルスクリーンアンチエイリアスモード（ANTI モード） | 13 |
| 1.4.3 | フリッカーフリーモード（FF モード） | 13 |
| 1.4.4 | 拡張 PAL モード | 14 |
| 1.5 | 初期化について | 15 |
| 1.6 | バッファについて | 16 |
| 1.6.1 | バッファの種類と概要 | 16 |
| 1.6.2 | バーテックスバッファ（Vertex Buffer） | 17 |
| 1.6.3 | ネイティブバッファ（Native Buffer） | 19 |
| 1.6.4 | テクスチャバッファ（Texture Buffer） | 19 |
| 1.6.5 | アキュムレーションバッファ（Accumulation Buffer） | 19 |
| 1.6.6 | フレームバッファ（Frame Buffer） | 20 |
| 1.7 | 割り込みについて | 21 |
| 1.7.1 | 割り込みの種類 | 21 |
| 1.7.2 | レンダリング終了割り込み | 21 |
| 1.8 | ローカルメモリについて | 22 |
| 1.9 | コンテキストについて | 23 |
| 1.10 | ポリゴンリストについて | 23 |
| 1.11 | ポリゴンパラメータについて | 24 |
| 1.11.1 | VertexParameter | 24 |
| 1.12 | カラーフォーマット | 25 |
| 1.13 | UV フォーマット | 25 |
| 1.13.1 | タイプ一覧 | 25 |
| 1.14 | シーケンスについて | 28 |
| 1.14.1 | レイテンシモード | 29 |
| 1.14.2 | レンダリングモード | 30 |
| 1.14.3 | シーケンスモード | 31 |
| 1.14.4 | 処理落ち | 32 |
| 1.14.5 | テクスチャの転送方式 | 33 |
| 1.14.6 | テクスチャ転送シーケンス | 34 |
| 1.14.7 | レンダーテクスチャ使用時のシーケンス | 35 |
| 1.14.8 | フレームバッファテクスチャ使用時のシーケンス（同期モード） | 35 |
| 1.14.9 | マルチパス使用時のシーケンス | 36 |
| 1.14.10 | ミドルウェア（Sofdec）使用時のシーケンス（同期モード） | 37 |
| 2 | 座標系 | 38 |
| 2.1 | Ninja2 の座標系についての概要 | 38 |
| 2.2 | 座標系に関する用語の統一 | 38 |
| 2.3 | Ninja2 で必要な座標系 | 40 |
| 2.3.1 | 世界座標系 | 40 |

| | |
|--|-----------|
| 2.3.2 カメラ座標系 | 41 |
| 2.3.3 ハードウェア座標系 | 41 |
| 2.4 Ninja2 の投影変換 | 43 |
| 2.5 初期状態 | 44 |
| 2.6 座標の流れについて | 45 |
| 2.7 カレントマトリックスについて | 46 |
| 2.8 画角やアスペクト比の設定関数について | 48 |
| 3 カメラ | 49 |
| 3.1 概要 | 49 |
| 3.2 カメラ関数群の構成 | 50 |
| 3.3 新カメラ関数群 | 51 |
| 3.3.1 基本カメラ関数群 | 51 |
| 3.3.2 カメラデータ関数群 | 51 |
| 3.3.3 旧カメラ互換関数群 | 52 |
| 3.4 新カメラ関数群と旧カメラ関数群の比較 | 53 |
| 3.4.1 旧カメラ関数群 | 53 |
| 3.4.2 新カメラ関数群 | 53 |
| 3.5 新旧カメラ関数対応表 | 54 |
| 3.6 基本カメラ関数群と通常マトリックス関数群の対応 | 55 |
| 3.7 新カメラ関数と通常マトリックス関数の関係の例 1 | 57 |
| 3.8 新カメラ関数と通常マトリックス関数の関係の例 2 | 57 |
| 3.9 新カメラ関数と通常マトリックス関数の関係の例 3 | 58 |
| 4 テクスチャ | 59 |
| 4.1 語句説明 | 59 |
| 4.2 Ninja2 の変更点 | 61 |
| 4.3 テクスチャロードシーケンス | 62 |
| 4.4 テクスチャリリースシーケンス | 63 |
| 4.5 テクスチャ管理 | 64 |
| 4.5.1 NJS_TEXMANAGE 構造体 | 64 |
| 4.5.2 NJS_TEXSYSTEM 構造体 | 65 |
| 4.5.3 通常のテクスチャ管理 | 65 |
| 4.5.4 パレットテクスチャのテクスチャ管理 | 66 |
| 4.5.5 テクスチャサーフェスとテクスチャ管理領域の開放 | 66 |
| 4.6 テクスチャの DMA 転送 | 67 |
| 4.6.1 ドライバの変更によるテクスチャの DMA 転送の変更 | 67 |
| 4.6.2 効率の良いテクスチャロードをするために | 68 |
| 4.7 テクスチャフォーマット | 69 |
| 4.8 VQ | 70 |
| 4.8.1 VQ テクスチャ構造 | 70 |
| 4.8.2 圧縮前後のデータサイズ | 70 |
| 4.8.3 VQ Mipmap | 71 |
| 4.8.4 Small VQ テクスチャ構造 | 71 |
| 4.8.5 Small VQ の各サイズとグループ数 | 72 |
| 4.9 パレット | 73 |
| 4.9.1 カラーモード | 73 |
| 4.9.2 パレットテクスチャの色数 | 73 |
| 4.9.3 バンク | 73 |
| 4.9.4 カラーデータ配置 | 74 |
| 4.10 カラーデータ拡張方法 | 75 |
| 4.11 VRAM への登録 | 76 |
| 4.11.1 テクスチャの登録 | 76 |

| | |
|-----------------------------------|------------|
| 4.12 レンダーテクスチャ | 77 |
| 4.12.1 フレームバッファとテクスチャデータ | 77 |
| 4.12.2 レンダーテクスチャシーケンス | 77 |
| 4.13 性能比 | 78 |
| 4.14 PVR フォーマット | 79 |
| 4.14.1 PVR ヘッダ情報 | 79 |
| 4.14.2 カテゴリーコード | 81 |
| 4.15 カラーフォーマット | 92 |
| 4.16 PVM フォーマット (.pvm) | 93 |
| 4.17 PVP フォーマット (.pvp) | 95 |
| 4.18 テクスチャカラーモード | 96 |
| 5 基本的な描画関数について | 97 |
| 5.1 2Dプリミティブ | 97 |
| 5.1.1 概要 | 97 |
| 5.1.2 ポリゴン | 97 |
| 5.1.3 テクスチャ | 97 |
| 5.1.4 ハイライトテクスチャ | 98 |
| 5.1.5 クアッドテクスチャ | 98 |
| 5.1.6 ライン | 99 |
| 5.2 3Dプリミティブ | 99 |
| 5.2.1 概要 | 99 |
| 5.2.2 ポリゴン | 99 |
| 5.2.3 テクスチャ | 100 |
| 5.2.4 ハイライトテクスチャ | 100 |
| 5.3 ライン | 101 |
| 5.3.1 njDrawLine3DEx | 101 |
| 5.3.2 一覧 | 101 |
| 5.3.3 頂点形式 | 101 |
| 5.3.4 Ex 関数群について | 101 |
| 5.4 パーティクル | 102 |
| 5.4.1 パーティクルポリゴン | 102 |
| 5.4.2 パーティクルスプライト | 102 |
| 5.4.3 パーティクルストリップ | 103 |
| 6 セルスプライト | 104 |
| 6.1 概要 | 104 |
| 6.2 セルとセルスプライトについて | 104 |
| 6.2.1 セルとセルスプライトの関係 | 104 |
| 6.2.2 セルスプライト構造体 | 105 |
| 6.2.3 セル構造体 | 106 |
| 6.2.4 セルスプライト 2D の計算 | 107 |
| 6.2.5 セルスプライト 3D の計算 | 109 |
| 6.3 セルストリームとセルストリームリスト | 112 |
| 6.3.1 セルストリーム | 112 |
| 6.3.2 セルストリームリスト | 113 |
| 6.3.3 タイムオフセットとタイムマックス | 114 |
| 6.3.4 リピート | 114 |
| 6.4 セルスプライトモーション | 115 |
| 6.5 その他 | 115 |
| 6.5.1 セルのアトリビュートを一括して変更 | 115 |
| 6.5.2 複数のセルスプライトのカラーを一括して変更 | 115 |

| | | |
|----------|--------------------|------------|
| 7 | モデル | 116 |
| 7.1 | 概要 | 116 |
| 7.1.1 | 構造 | 116 |
| 7.1.2 | 処理 | 116 |
| 7.1.3 | 関数一覧 | 117 |
| 7.2 | ファンクション | 118 |
| 7.2.1 | CnkEasyDraw | 118 |
| 7.2.2 | CnkSimpleDraw | 120 |
| 7.2.3 | CnkEasyMultiDraw | 122 |
| 7.2.4 | CnkSimpleMultiDraw | 126 |
| 7.2.5 | CnkWireDraw | 129 |
| 7.2.6 | CnkToonDraw | 131 |
| 7.2.7 | CnkDirectDraw | 133 |
| 7.3 | 中間バッファ | 135 |
| 7.3.1 | CnkEasy | 135 |
| 7.3.2 | CnkSimple | 135 |
| 7.3.3 | CnkEasyMulti | 135 |
| 7.3.4 | CnkSimpleMulti | 136 |
| 7.3.5 | CnkWireDraw | 136 |
| 7.3.6 | CnkToonDraw | 136 |
| 7.4 | フォーマット | 137 |
| 7.4.1 | VLIST | 137 |
| 7.4.2 | PLIST | 138 |
| 7.5 | モディファイア | 139 |
| 7.5.1 | 概念 | 139 |
| 7.5.2 | チープシャドウモディファイア | 140 |
| 7.5.3 | 2Pモディファイア | 140 |
| 7.5.4 | ニアクリップ | 140 |
| 7.6 | クリッピング | 140 |
| 7.6.1 | モデルクリップ | 140 |
| 7.6.2 | ニアクリップ | 140 |
| 8 | オブジェクト | 141 |
| 8.1 | 概要 | 141 |
| 8.2 | 構造 | 141 |
| 8.3 | オブジェクトトレース | 142 |
| 8.4 | エンベロープ | 142 |
| 8.4.1 | 概念 | 142 |
| 8.4.2 | データ構造 | 143 |
| 8.4.3 | 中間バッファ | 144 |
| 8.4.4 | クリップ | 145 |
| 8.5 | イーバルフラグ | 146 |
| 9 | モーション | 147 |
| 9.1 | 概要 | 147 |
| 9.2 | モーション構造体 | 148 |
| 9.2.1 | 構造体関連図 | 148 |
| 9.2.2 | 構造体解説 | 149 |
| 9.3 | オブジェクトモーション | 153 |
| 9.3.1 | 構造体関連図 | 153 |
| 9.3.2 | 構造体解説 | 154 |
| 9.4 | カメラモーション | 156 |
| 9.4.1 | カメラ構造体 | 156 |

| | |
|--|-----|
| 9.4.2 カメラモーションについて | 157 |
| 9.5 ライトモーション | 158 |
| 9.5.1 ライト構造体 | 158 |
| 9.5.2 ライトモーションについて | 159 |
| 9.5.3 描画関数とライト種類によるライト構造体のメンバ | 160 |
| 9.5.4 マルチライトモーションについて | 162 |
| 9.6 回転の表現について | 163 |
| 9.6.1 回転の表現とは | 163 |
| 9.6.2 回転の表現について | 163 |
| 9.6.3 オイラー角 (Euler Angle) による方法 | 164 |
| 9.6.4 クォータニオン (Quaternion) による方法 | 164 |
| 9.6.5 クォータニオンの利点 | 165 |
| 9.7 回転要素の取り扱い | 166 |
| 9.7.1 NJS_OBJECT 構造体中での回転要素 | 166 |
| 9.7.2 NJS_MOTION 構造体中での回転要素 | 166 |
| 9.7.3 NJS_MOTION 構造体と NJS_OBJECT 構造体の回転要素の関係 | 167 |
| 9.7.4 モーションリンクにおける回転要素の補完について | 169 |
| 9.8 モーションの補完法 | 170 |
| 9.8.1 リニア補完 | 170 |
| 9.8.2 スプライン補完 | 171 |
| 9.8.3 クォータニオンにおける補完 | 172 |
| 9.9 モーションリンクとは | 173 |
| 9.10 高レベルモーション関数 | 175 |
| 9.10.1 高レベルモーション関数の分類 | 175 |
| 9.10.2 高レベルモーション関数群 | 176 |
| 9.10.3 高レベル関数群での NULL の指定 | 178 |
| 9.10.4 高レベル関数群での留意事項 | 179 |
| 9.11 低レベルモーション関数 | 180 |
| 9.11.1 低レベルモーション関数の分類 | 180 |
| 9.11.2 低レベルモーション関数群 | 180 |
| 9.12 モーション関数の変更点について | 182 |
| 9.13 新カメラモーション関数 | 182 |
| 9.13.1 新カメラモーション関数と旧カメラモーション関数の違い | 182 |
| 9.13.2 カメラモーション関数の使用例 | 183 |
| 9.14 ライトモーション関数 | 183 |
| 9.14.1 ライトモーション関数の使用例 | 183 |
| 9.14.2 マルチライトモーション関数の使用例 | 184 |
| 10 Njutil ライブラリ | 185 |
| 10.1 Njutil ライブラリのテクスチャ関数 | 185 |
| 11 高速化&テクニック | 186 |
| 11.1 メモリバンク | 186 |
| 11.2 PLIST の最適化 | 187 |
| 11.2.1 マテリアル調整 | 187 |
| 11.2.2 テクスチャ、アトリビュートの連続 | 187 |
| 11.3 ディセーブルニアクリップ (Disable Near Clip) | 187 |
| 11.4 イーバルクリップ (Eval Clip) | 187 |
| 11.5 頂点参照 | 188 |
| 11.6 テクスチャ合成 | 189 |
| 11.7 レンダリング特性 | 189 |
| 11.8 プリフェッチ | 189 |

| | |
|-------------------------------------|------------|
| 12 質問と回答（Q&A）及び対処法 | 190 |
| 12.1 モーションについて | 190 |
| 用語集..... | 191 |

1 Ninja2 ライブラリについて

ここでは、Ninja2 ライブラリの基本的な部分の詳細を説明します。

1.1 基本事項

1.1.1 Ninja2 ライブラリで使用するファイル拡張子の説明

Ninja ファイルフォーマットの拡張子を説明します。

Ninja は、アスキーフォーマットとバイナリフォーマットをサポートします。
アスキーフォーマット、バイナリフォーマットはそれぞれに一対一で対応します。

拡張子のベース (nj) という文字列で、それぞれに中身を意味する 1 文字を付加します。アスキーでは (nj) の (j) を (a) に置き換えて表現します。

- データ別拡張子

| | バイナリフォーマット拡張子 | アスキーフォーマット拡張子 |
|--------------|---------------|---------------|
| Texlist | .njt | .nat |
| Model | .njd | .nad |
| Light | .njl | .nal |
| Camera | .njc | .nac |
| Motion | .njm | .nam |
| Shape Motion | .njs | .nas |

- その他の拡張子

| | バイナリフォーマット拡張子 | アスキーフォーマット拡張子 |
|--------------------|---------------|---------------|
| 複数のデータを格納する場合 | .nj | .nja |
| シーンファイル (アスキーのみ) | - | .nsc |
| リソースファイル (アスキーのみ) | - | .nre |
| テキストチャデータ (バイナリのみ) | .pvr | - |
| パレットデータ (バイナリのみ) | .pvp | - |
| マルチテキストチャ (バイナリのみ) | .pvm | - |

- 注意事項

- モーションはライト、カメラに関してもモーション構造体で管理されるため .njm (.nam) で表現されます。ただしシェープに関しては通常のモーションと組み合わせて使われるため拡張子を分けています。
- 二種類以上のデータを格納した場合 .nj (.nja) が使われます。現在のモデルコンバータはすべて (.nja) を出力していました。これには texlist (.nat) と model (.nad) が含まれています。
- Chunk Model/Basic Model とともに .njd (.nad) を使います。ただし、現在は texlist とともにモデルを出力しているため、.nj (.nja) で出力されます。

1.1.2 Ninja ライブラリで使用するファイル

| ファイルの種類 | 拡張子 | 備 考 |
|-----------|--------------|---|
| シーン | .nsc | そのシーンに使われているモデル、カメラ、ライト、モーションの各ファイルリストが記載されています。 |
| モデル | .nj .nja | 通常のモデルファイルです。（.njd と.njt、もしくは.nad と.nat が格納されています） |
| モデル | .njd .nad | 上記のモデルファイルから、テクスチャリストをセパレートしたものです。 |
| テクスチャリスト | .njt .nat | モデルファイルからセパレートされたリストです。 |
| テクスチャ | .pvr | 通常のテクスチャファイルです。 |
| テクスチャ | .pvm | 複数の .pvr ファイルをまとめたものです。この.pvm ファイルを使用することによりテクスチャのロード回数を減らすことができます。 |
| パレットデータ | .pvp | パレットデータを格納します。 |
| ライト | .njl .nal | ライト構造体です。 |
| カメラ | .njc .nac | カメラ構造体です。 |
| モーション | .njm .nam | モデル、ライト、カメラ共通のモーションファイルです。 |
| シェイプモーション | .njs .nas | 頂点移動によるモーションです。 |

拡張子が2つある場合は、上がバイナリデータ、下がアスキーデータです。

1.1.3 その他のファイル

| ファイルの種類 | 拡張子 | 備 考 |
|-----------|------|--|
| モーションリソース | .mrs | モデル階層構造情報が記載されています。このデータを基本姿勢とし、これと差が無いモーションを削除する事でモーションファイルを圧縮できます。また、mrs ファイルは出力したデータの内容確認も目的としています。プログラマへの階層構造の開示、プログラム制御時に、目的とするオブジェクトの順番確認。モデル制作時のポリゴン数確認などができます。 |
| リソース | .nre | コンバータで使用する変換オプションなどが記載されます。ファイルのコンバート時に読み込まれ、以前に行った設定を生かすことができます。（他の用途にも利用していく予定です） |
| エクスポートログ | .log | ファイル・コンバート時の情報が記載されています。 |

拡張子が2つある場合は、上がバイナリデータ、下がアスキーデータです。

- Ninja におけるシーンファイルは、シーンを構成するモデル、カメラ、ライト、モーションを括るための、ファイル名リストです。NinjaViewer でのシーン表示のための単位にもなります。アスキー出力のみです。
- リソースファイル（.nre）は、汎用のリソース格納ファイルです。libnre ライブラリにより管理され、Ninja グラフィックスツールで必要とされるリソースの保存 / 読み出し / 利用を一括して管理します。現在、テクスチャのコンバータオプション保存に適用されています。今後は、コンバータオプションの保存やユーザー設定データの保存などに、拡張していく予定です。
- バイナリフォーマットは、すべて iff チャンク形式で格納されます。各種データを同一のファイルにまとめて格納することが可能です。バイナリロード nuBinary を使うことで、コンパイルなしでユーザーmalloc 領域へのデータの格納が可能です。

- アスキーフォーマットは、アスキーロードによる読み込み、もしくは NjDef.h をインクルードしてコンパイルすることで利用が可能になります。
- pvp ファイルは、パレットデータを格納します。
- pvm ファイルは、複数のテクスチャを一括して扱うためのファイルフォーマットです。

1.2 モデル構造について

Ninja は、Chunk Model 形式のモデル構造のみをサポートしています。Basic Model はサポートしていません。

1.2.1 Chunk Model 形式

Chunk Model 形式は、描画実行中に SH4 のキャッシュを壊さないようにデータを連続するメモリ空間に配置します。拡張性、柔軟性、データの表現効率に優れています。

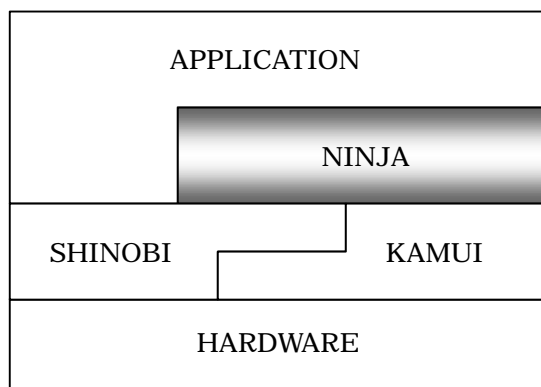
今後は、Chunk Model 形式のモデルを中心としたチューニングを実施します。

Chunk Model 形式は、Model 構造体の中身を大幅に変更しますが、Object 構造体は Model 構造体のポインタから Chunk Model 構造体のポインタへの変更以外に変更されません。

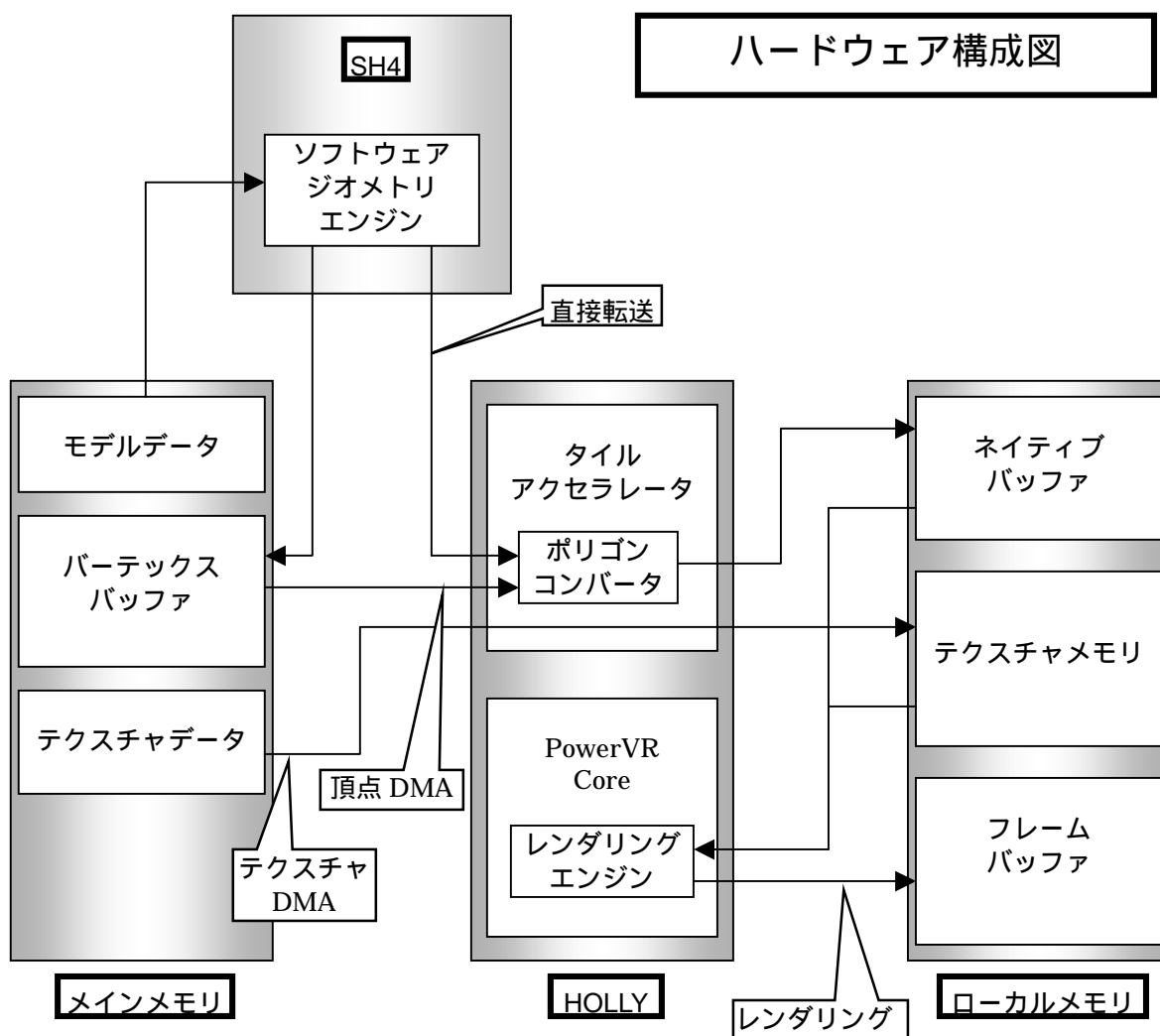
モデル以外のモーション及びテクスチャについては、引き続き現行の形式が使用されます（ただしカメラ、ライト対応のために構造体メンバの型が変更されます）。

- Chunk Model の特徴
 - トライアングルストリップ描画を基本とします。現在は、三角形、四角形、N 角形を描画できません。性能優先で設計されています。
 - データは、頂点リスト vlist とポリゴンリスト plist からなります。vlist、plist 上に IFF チャンク形式でデータを配置してメモリ領域を一本化し、描画実行中にキャッシュを壊さないようにしています。
 - Discontinuity データの出力が可能です。
 - ポリゴン側（最大 16bit × 3）と頂点側（32bit × 1）に、ユーザーフラグ領域を持つことができます。現在は頂点カラーデータをユーザーフラグ領域へ出力するときに、この領域を使用します。今後この部分にユーザーデータを書き込めるツールを用意する予定です。
 - マテリアルは plist に格納され、以前に設定したマテリアルの変更部分（差分）のみの設定だけを更新します。
 - コンバータ出力時にマテリアルを削除できます。同じモデルの描画（例えば木）などすべてのマテリアルをなくし、ユーザーが外で設定することでデータを最適化することができます。
 - コリジョン用データ Chunk Volume の出力がサポートされています。三角形、四角形、トライアングルストリップで出力できます。マテリアル情報を持ちません。ユーザーフラグ領域を持つことができます。現在この領域にマテリアルカラーを出力できます。
 - 独立三角形の Chunk Volume（volume3）は、モディファイアボリュームにも利用できます。
 - コンバータで volume34 を指定すると、面間角度が 0.1 度の三角形同士を接続し四角形を作ります。3D Studio MAX は三角形データしか出力できませんが、この場合でも四角形コリジョンが生成できます。
 - 二種類の UV 値表現を持ちます。0 - 255 による UVN、分解能を高めた 0 - 1023 による UVH です。256 を超えるテクスチャでは、1 ピクセル単位で指定できません。UVH はハイレゾモード、1024 × 1024 のテクスチャで、1 ピクセル単位での指定が可能です。ただし、UVH は UVN よりも分解能を上げた分だけ、テクスチャのリピートの表現回数が減少します（UVN が 128 回に対し UVH では 32 回）。コンバートオプションによりモデルツリー全体に対したマテリアルネームにより各モデル単位で UVN、UVH を切り替えることができます。マテリアルネームからの指定では単一のモデルに使われる複数のマテリアルのどれか一つに設定することでモデルに対する UV 表現が変更されます。デフォルトは UVN です。

1.3 Ninja2 ライブラリの概要



ライブラリ構成図



ハードウェア構成図

1.4 画面モード（画面解像度設定）について

ここでは画面に表示する際に、関連する画面モードについての説明をします。

1.4.1 ケーブルの自動判別について

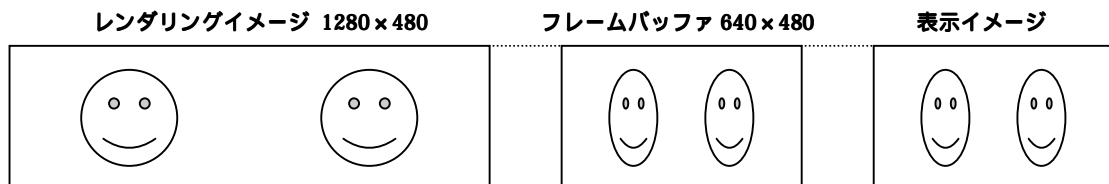
ドリームキャストに接続されているケーブルの種別を自動的に判別し、適切なモード(VGA、NTSC、PAL)を設定することができます。これはユーザーが選択するのは解像度のみでよいことになり、特殊なモードを使わなければ自動的にVGA対応となります。

1.4.2 フルスクリーンアンチエイリアスモード（ANTI モード）

画面全体に、2～4倍のオーバーサンプリングを行うことにより、ポリゴンのエッジがぼけてアンチエイリアスがかかります。解像度が、640×480のときのみ横方向だけの2倍オーバーサンプリングとなり、その他の解像度のときは、縦横両方で4倍オーバーサンプリングになります。

注意 このモードを使ってもフレームバッファのサイズは変化しませんが、レンダリングの際に掛かる時間は確実に増加します。

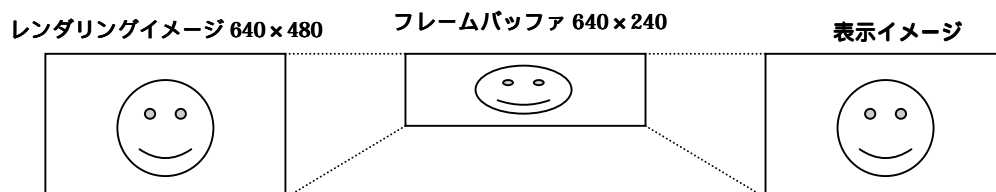
- 画面解像度が640×480の場合



1.4.3 フリッカーフリーモード（FF モード）

Ninja 画面設定でのフリッカーフリーモードは、フリッカーフリーフィルタリングのタイプBで行われています。このため、レンダリングサイズは変わりませんが、フレームバッファの縦サイズは基本画面モードの半分で済みます。ただし、メイン処理、レンダリング共に1/60秒以内に終わる必要があります。もしオーバーした場合、画面イメージが不正になります。

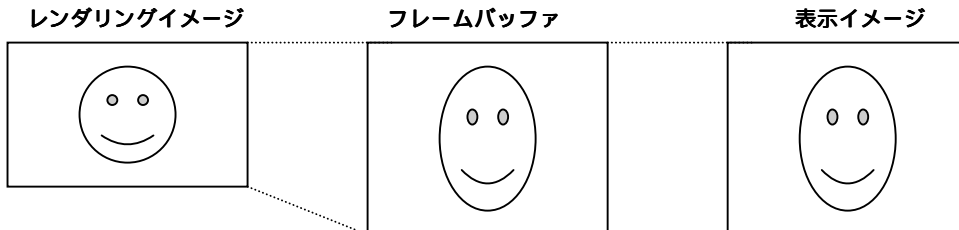
注意 VGAではこのモードは使用できませんので、注意して下さい。



1.4.4 拡張 PAL モード

拡張 PAL モードは、基本画面モードに Y スケールして表示しています。レンダリングサイズは基本画面モードのサイズですが、フレームバッファサイズは、拡張した Y 方向分のフレームバッファサイズが必要です。

基本画面モードの縦方向を 1.033、1.066、1.100、1.133、1.166 倍します。



| 画面モード | 表示サイズ | レンダリングサイズ | フレームバッファサイズ |
|-----------------|-----------|-----------|-------------|
| PAL ノンインタレース | 320 × 248 | 320 × 240 | 320 × 248 |
| | 320 × 256 | | 320 × 256 |
| | 320 × 264 | | 320 × 264 |
| | 320 × 272 | | 320 × 272 |
| | 320 × 280 | | 320 × 280 |
| | 640 × 248 | 640 × 240 | 640 × 248 |
| | 640 × 256 | | 640 × 256 |
| | 640 × 264 | | 640 × 264 |
| | 640 × 272 | | 640 × 272 |
| | 640 × 280 | | 640 × 280 |
| PAL インタレース | 320 × 248 | 320 × 240 | 320 × 248 |
| | 320 × 256 | | 320 × 256 |
| | 320 × 264 | | 320 × 264 |
| | 320 × 272 | | 320 × 272 |
| | 320 × 280 | | 320 × 280 |
| | 640 × 248 | 640 × 240 | 640 × 248 |
| | 640 × 256 | | 640 × 256 |
| | 640 × 264 | | 640 × 264 |
| | 640 × 272 | | 640 × 272 |
| | 640 × 280 | | 640 × 280 |
| | 640 × 496 | 640 × 480 | 640 × 496 |
| | 640 × 512 | | 640 × 512 |
| | 640 × 528 | | 640 × 528 |
| | 640 × 544 | | 640 × 544 |
| | 640 × 560 | | 640 × 560 |

1.5 初期化について

ここでは、それぞれの初期化について説明します。

| 関数名 | 機 能 |
|----------------------|-----------------------|
| njInitDevice | デバイス（ハードウェア）の初期化 |
| njInitSystem | 画面モード、フレームバッファモードの初期化 |
| njInitVertexBufferEx | バーテックスバッファ、システムの初期化 |
| njInitTextureEx | テクスチャマネージャの初期化 |
| njInit3D | 3D システムの初期化 |
| njInitMatrix | マトリックススタックの初期化 |

- njInitDevice
デバイス（ハードウェア）の初期化を行います。
Ninja 内部では、kmInitDevice を呼び出しているのみです。必要な割り込みを登録して、ハードウェアのレジスタを設定します。
- njInitSystem
画面の解像度、フレームバッファのフォーマット、VSync カウントの設定を行います。

注 意 旧 njChangeSystem は、この関数に統合されました。画面モードを変更するときは再びこの関数を呼び出して下さい。

通常は VGA、NTSC、PAL の切り替えを自動判別するパラメータを設定して下さい。自分でケーブルを判別してモードを切りかえる場合は矛盾のないように設定して下さい。フレームバッファのフォーマットを ARGB8888 といったモードにする場合（ムービー再生等）は、テクスチャメモリの容量に影響を与えますので気をつけて下さい。VSync カウントはフレームレートを固定（30FPS 等）したい場合に 2 以上の値にします。ただし、2 以上の値に設定した場合、非同期モードは使用不可となります。

- njInitVertexBufferEx
バーテックスバッファの設定、及び各種モードの設定を行います。

注 意 njInitSystem の後で、かつ njStartDraw の前に実行しなければいけません。

2V・3V レイテンシ、同期・非同期レンダリングの設定はここで行います。Ninja 内部でパラメータの変換を行った後に kmSetSystemConfiguration を呼び出します。その時に使用した SystemConfigStruct 及び VertexBufferDesc は njGetSystemConfigStruct 関数、njGetVertexBuffDesc 関数で取得することができます。

- njInitTextureEx
テクスチャマネージャを初期化します。
njInitVertexBufferEx で設定したテクスチャ枚数以上を設定しても意味がありません。
パラメータの NJS_TEXMANGE と NJS_TEXSYSTEM に関しては、リファレンスを参照して下さい。

- njlInit3D
3D Draw 系関数を実行する際に、中間バッファとして使用するバッファを設定します。
バッファサイズは以下の式から計算できます。

バッファサイズ = 階層単位の最大頂点数 × 中間バッファサイズ

中間バッファのサイズは描画関数ごとに異なりますが、最大で 64 バイトです。
- njlInitMatrix
マトリックススタックを設定します。
ただし、マトリックススタックはファイバー単位で切り替わります。ファイバー内でマトリックススタックを使用したい場合は、各ファイバーの先頭で njlInitMatrix を実行する必要があります。

1.6 バッファについて

ここでは、データを一時的に蓄積する Dreamcast 上のバッファ・メモリ（緩衝記憶装置）の詳細について説明します。

1.6.1 バッファの種類と概要

Dreamcast でバッファと称される部分は、以下の 5 種類です。
そのうち、画像処理に関係あるバッファについて説明します。

- バーテックスバッファ（Vertex Buffer）
ポリゴンリスト情報を格納します。
（CPU によって Work RAM 上に設定されます）
- ネイティブバッファ（Native Buffer）
PowerVR2 CORE で処理されるディスプレイリスト情報を格納します。
- アキュムレーションバッファ（Accumulation Buffer）
タイル単位の描画結果を格納します。（Holly チップ内部）
- フレームバッファ（Frame Buffer）
1 画面の描画結果を格納します。
- テクスチャロードバッファ（Texture Load Buffer）
テクスチャのロードバッファです。

1.6.2 バーテックスバッファ (Vertex Buffer)

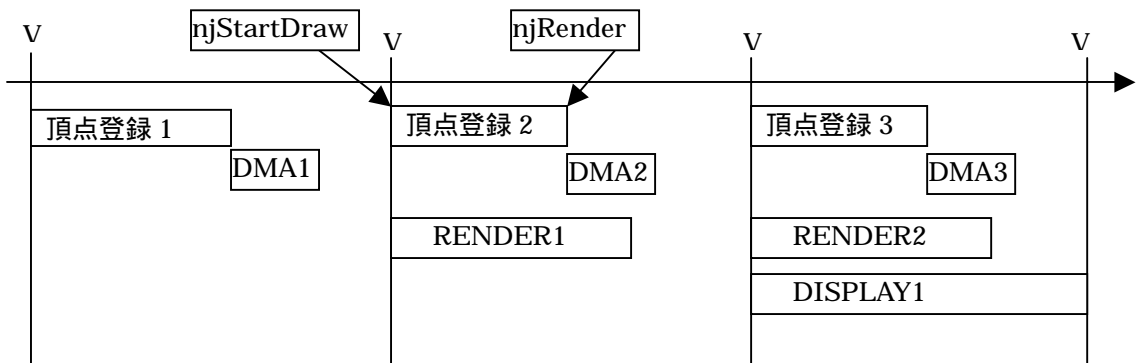
ハードウェアに送るためのパラメータを保持するバッファです。
njInitVertexBufferEx で設定します。

レイテンシモードの使い方によってバーテックスバッファの使用状況は大きく異なり、状況に応じて使い分ける必要があります。

- 2 V レイテンシモードの場合について

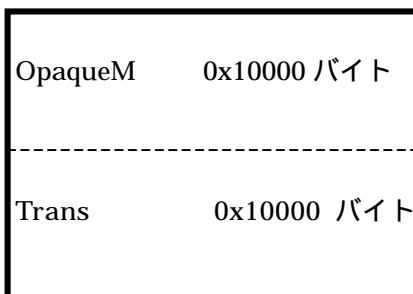
頂点登録したデータが画面に表示されるまでのレイテンシ (遅延) が 2 V あるモードです。

キー入力の反応時間を向上させる場合やバーテックスバッファの容量を減らす場合に使用します。2 V レイテンシモードでは、Ninja の njInitVertexBufferEx 関数でマイナスを指定したリストタイプをハードウェアに直接転送し、その他のリストタイプはすべての頂点登録後、njRender の関数が呼ばれたときに DMA 転送を開始します。このモードの場合、頂点登録と DMA 転送が 1 V 以内に収まらなかった場合は処理落ちします。バーテックスバッファの容量は、直接転送するリストタイプの分は必要なく、また頂点登録と DMA 転送は同時に行われないため、シングルバッファでよいので容量を減らすことができます。



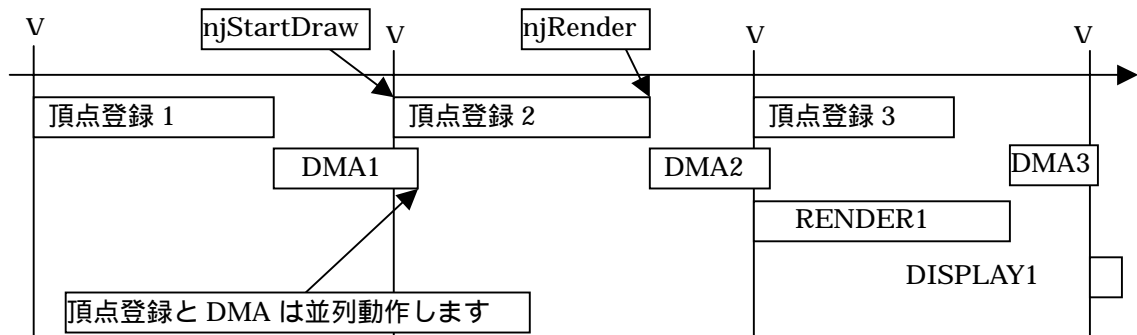
- 2 V レイテンシの場合の設定例

| | | |
|------------------------------------|--------------|---------------|
| NNumOfMaxPass = 1; | 最大パス数 1 | |
| sPassInfo[0]. nSize[0] = -1; | Opaque | 直接転送 |
| sPassInfo[0]. nSize[1] = 0x1000/4; | OpaqueM | 0x10000 バイト確保 |
| sPassInfo[0]. nSize[2] = 0x1000/4; | Trans | 0x10000 バイト確保 |
| sPassInfo[0]. nSize[3] = 0; | TransM | 使用しない |
| SPassInfo[0]. nSize[4] = 0; | PunchThrough | 使用しない |



- 3 V レイテンシモード

使用するリストタイプの種類が豊富で、なおかつ使用頻度がばらばらの場合、このモードを使用するとパフォーマンスが向上する場合があります。ただし、バーテックスバッファをダブルバッファにする必要があり、メモリを消費します。



- 3 V レイテンシの場合の設定例

| | | |
|------------------------------------|--------------|---------------|
| NNumOfMaxPass = 1; | 最大パス数 1 | |
| SPassInfo[0]. nSize[0] = 0x2000/4; | Opaque | 0x20000 バイト確保 |
| SPassInfo[0]. nSize[1] = 0x1000/4; | OpaqueM | 0x10000 バイト確保 |
| SPassInfo[0]. nSize[2] = 0x1000/4; | Trans | 0x10000 バイト確保 |
| SPassInfo[0]. nSize[3] = 0; | TransM | 使用しない |
| SPassInfo[0]. nSize[4] = 0; | PunchThrough | 使用しない |

| | |
|---------|-------------|
| Opaque | 0x10000 バイト |
| ----- | |
| OpaqueM | 0x8000 バイト |
| ----- | |
| Trans | 0x8000 バイト |
| ----- | |
| Opaque | 0x10000 バイト |
| ----- | |
| OpaqueM | 0x8000 バイト |
| ----- | |
| Trans | 0x8000 バイト |

- レイテンシモードの違いについて

2 V レイテンシの場合、直接転送に指定したリストタイプのパラメータは、バーテックスバッファを経由せずに直接ハードウェアに送られます。それに加え、2 V レイテンシはシングルバッファ、3 V レイテンシはダブルバッファであることが必要です。

また、バーテックスバッファはメインメモリに確保され、そのアクセス速度は直接ハードウェアに送るための TA FIFO のアクセス速度を下回ります。

以上のことから、2 V レイテンシモードに大きなアドバンテージがあることがわかります。

1.6.3 ネイティブバッファ (Native Buffer)

ネイティブバッファは、バーテックスバッファで設定されたポリゴンリストを、タイルアクセラレータ上でタイル分割を行い、その分割されたポリゴンの「頂点情報」「登録タイルのリスト」を格納するためのバッファです。

このバッファに格納されたデータをもとに、PowerVR2 CORE がタイル単位で Z ソートや描画を行います。この領域は 32 ビットでアクセスされます。

注意 ネイティブバッファはダブルバッファ構成をとる必要があり、「登録タイルのリスト」が描画条件により大幅に増減するため、テクスチャバッファの設計には注意が必要になります。

- (1) ネイティブバッファの容量
容量の計算式について

| |
|--|
| $\text{ネイティブバッファ容量} = \text{VRAM 全体の容量} - (\text{Texture Buffer サイズ} + \text{表示用 Frame Buffer サイズ})$ |
|--|

で算出されます。

1.6.4 テクスチャバッファ (Texture Buffer)

テクスチャデータを格納するバッファです。

テクスチャを格納する領域は、64 ビットアクセスになります。このバッファに対してテクスチャデータの他に、アキュムレーションバッファの結果をコピーすることやテクスチャに対してもレンダリングが可能です。

テクスチャメモリ内でのテクスチャデータの格納状態は、描画性能に大きな影響を与えます。テクスチャメモリは、SDRAM に「ページ」と呼ばれる 2048byte ごとのエリアに分割されており、描画性能を低下させないためには、各テクスチャデータをこの 1 ページ内に収めることが重要になります。ちなみに 2048byte というデータ量は、32×32 サイズの 16bit テクスチャ 1 つ分に相当します。

様々なサイズのテクスチャを格納する場合は、各サイズを上手に組み合わせることでページ境界をまたがないように格納することで、描画性能を最大限に発揮することができます。1 つが 2Kbyte 以上のテクスチャデータも、できる限りページ境界をまたがないようにすることで、描画性能を低下しないようにすることが可能です。

また、VQ テクスチャの場合は Code Book サイズが 2Kbyte になっているので、Code Book の先頭アドレスを 2Kbyte 境界にすることが効果的です。

1.6.5 アキュムレーションバッファ (Accumulation Buffer)

PowerVR2 CORE に内蔵されている描画用のバッファです。

タイル分 (32×32) の大きさを持ち、グーロシェーディング、アルファブレンディング等のエフェクト処理を行った結果を格納します。このバッファにある描画結果が、フレームバッファにコピーされます。

1.6.6 フレームバッファ (Frame Buffer)

フレームバッファは、TV 画面 1 枚分の画像データを保存しておき、ハードウェアはそのイメージを参照して 1 枚の TV 画面を作成します。

通常、フレームバッファは、表示用と描画用のダブルバッファ構成となっており、画面が大きければそれだけ多くのメモリを必要とします。アキュムレーションバッファの結果がそのままフレームバッファにコピーされるだけでなく、フレームバッファモードの選択により、16 ビットカラーに減色したり、32 ビットカラーのうちの 部分を削ったり、また、カラーランプ機能を使用してフェードイン・アウトの効果を施したり、Image Super Sampling によってアンチエイリアスが掛けられた画像も格納できます。

この領域は 32 ビットアクセスとなります。

- フレームバッファサイズの算出方法
フレームバッファサイズを求める方法を提示します。

表 1 - 1 フレームバッファのサイズ

| 画面解像度 | カラーサイズ | 画面解像度 × カラーサイズ × バッファ数 |
|-----------|--------|---|
| 320 × 240 | 16BPP | (320 × 240) × 2 Byte × 2 Buffer = 307200 Byte |
| | 24BPP | (320 × 240) × 3 Byte × 2 Buffer = 460800 Byte |
| | 32BPP | (320 × 240) × 4 Byte × 2 Buffer = 614400 Byte |
| 320 × 480 | 16BPP | (320 × 480) × 2 Byte × 2 Buffer = 614400 Byte |
| | 24BPP | (320 × 480) × 3 Byte × 2 Buffer = 921600 Byte |
| | 32BPP | (320 × 480) × 4 Byte × 2 Buffer = 1228800 Byte |
| 640 × 240 | 16BPP | (640 × 240) × 2 Byte × 2 Buffer = 614400 Byte |
| | 24BPP | (640 × 240) × 3 Byte × 2 Buffer = 921600 Byte |
| | 32BPP | (640 × 240) × 4 Byte × 2 Buffer = 1228800 Byte |
| 640 × 480 | 16BPP | (640 × 480) × 2 Byte × 2 Buffer = 1228800 Byte |
| | 24BPP | (640 × 480) × 3 Byte × 2 Buffer = 1843200 Byte |
| | 32BPP | (640 × 480) × 4 Byte × 2 Buffer = 2457600 Byte |

上記条件は、アンチエイリアスをかけても変わりませんが、タイプ B のフリッカーフリーを使用した場合に（縦 480 のときのみ）のみ半分になります。

1.7 割り込みについて

ここでは割り込みの詳細について説明します。

1.7.1 割り込みの種類

現在 Ninja2 から登録できる割り込みには、レンダリング終了割り込み、頂点転送終了割り込み、水平ライン同期割り込みの3種類があります。

これらの設定関数はそれぞれ Kamui2 の登録関数を使用しています。

また、レンダリング終了割り込みと頂点転送終了割り込みはタイムアウト時（実際のレンダリング等の処理が終了しないと Kamui2 自身が判断したとき）には普通の関数コールとなる可能性があります。これらの割り込みでファイバーの切り替えを行う場合は注意して下さい。

上記以外の割り込み（VBlank-In 等）をハンドリングしたい場合は、ユーザーチェインライブラリを使用して下さい。

注 意 Ninja2 と Kamui2 を併用する場合はリソースの競合に気をつけて下さい。

対応表

| Ninja2 | Kamui2 |
|------------------|--------------------------|
| njHSyncCallback | kmSetHSyncCallback |
| njSetEORCallback | kmSetEORCallback |
| njSetEOVCallback | kmSetEndOfVertexCallback |

1.7.2 レンダリング終了割り込み

レンダリング終了割り込み関連の関数は、以下の関数が用意されています。

(1) レンダリング終了割り込みの登録関数（njSetEORCallBack）

登録関数はレンダリング終了割り込みで実行します。

1.8 ローカルメモリについて

ここでいうローカルメモリとは VRAM やテクスチャメモリと呼ばれているもので、3つのエリアから構成されています。

それらの比率は njInitSystem、njInitVertexBufferEx のパラメータで決定します。
ローカルメモリ全体の容量は Dreamcast の場合 8 MB です。

- フレームバッファエリア

特殊な場合（FF インターレスやストリップバッファ）を除いて、njInitSystem の画面モードとフレームバッファのフォーマットから容量が決定します。

計算式は以下のとおりです。

| |
|-----------------------------------|
| 容量 = 横幅 × 縦幅 × ビット深度 × 2（ダブルバッファ） |
|-----------------------------------|

例えば、標準的なモードの場合（640 × 480 VGA RGB565）の容量は

$640 \times 480 \times 2 \times 2 = 1.2\text{MB}$ となります。

- テクスチャバッファエリア

njInitVertexBufferEx のパラメータの nTextureMemorySize に指定したサイズが、テクスチャに割り当てる容量になります。

テクスチャに 4 MB を割り当てる場合の計算は、

$nTextureMemorySize = 4 \times 1024 \times 1024 / 4$ とします。

最後に 4 で割るのは、指定する単位がロングワードになっているからです。

- ネイティブバッファエリア

全体の容量からフレームバッファとテクスチャサイズを引いた残りが、ネイティブバッファに割り当てる容量になります。

ネイティブバッファは ISP/TSP パラメータ部（ITP）、オブジェクトポインタリスト部（OPB）、リージョンアレイ部の 3 種類から構成されており、ポリゴンを登録すると ITP と OPB にデータが追加されていきます。

そしてそれらがオーバーすると、ネイティブバッファオーバーフローとなり正常に描画されません（一瞬ちらつくのはこのため）。ただ、ITP と OPB のどちらかのみがオーバーした場合、そこでリミットアドレスの再構成が行われるため、再び正常に描画される可能性があります。

1.9 コンテキストについて

コンテキストとは描画する際に使用するパラメータのことで3種類存在します。

- シーンコンテキスト

シーンというのは簡単にいうとレンダリング一回分のことで、レンダリングごとにしか変更できません。レンダリング中に変更関数を呼んだ場合でも、反映されるのは次のレンダリングです。

注意 シーンコンテキストが更新されるタイミングは、あくまで次のレンダリングであってシーンにシンクロしていないので注意して下さい。レンダーテクスチャや非同期モードの場合、それらが影響してきます。

一覧

パレットカラーモード、パレットデータ、フォグカラー、フォグテーブル、カラークランプ MinMax 値、ストライド値、バックグラウンドカラー（テクスチャ）、ボーダーカラー、チープシャドウモード、パンチスルースレッシュールド値、スモールカリングスレッシュールド値、ピクセルクリップ、バーチカルフィルタモード

- ストリップコンテキスト

ストリップ単位で変更できるパラメータで、グローバルパラメータと呼ばれる場合もあります。Ninja は内部でコンテキストを保持していますので、変更した場合それがずっと反映されます。

注意 モデル描画系関数を呼んだ場合については、内部でコンテキストを変更してますので注意して下さい。

- パーテックスデータ

頂点単位で指定できるパラメータで、頂点座標や頂点カラー（頂点フォーマットにより形式は異なる）、UV 値といったものが含まれます。

1.10 ポリゴンリストについて

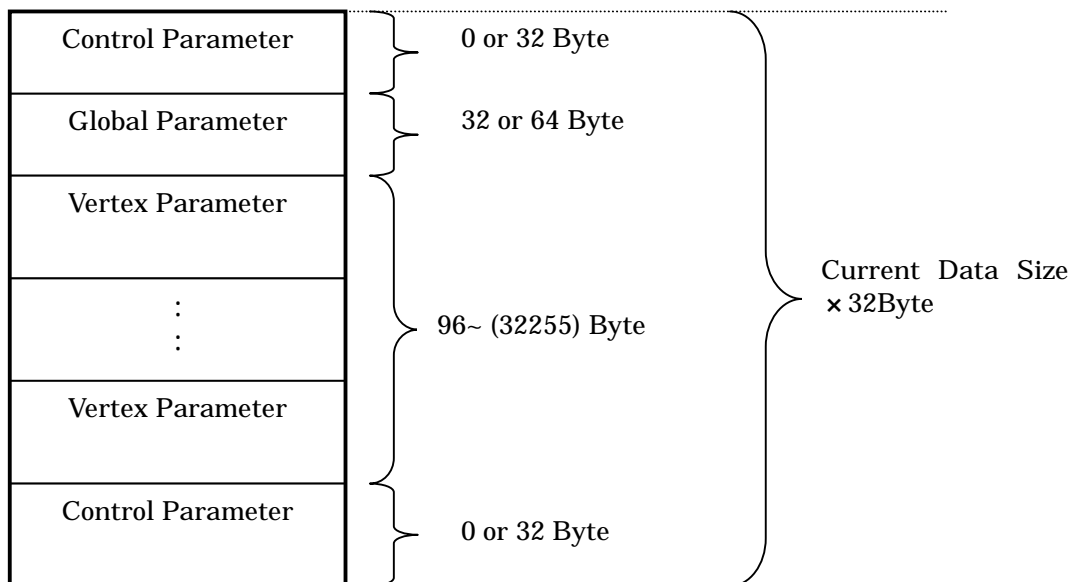
PowerVR2 には「トランスルーセント」「トランスルーセントモディファイア」「オペーク」「オペークモディファイア」「パンチスルー」の5種類のポリゴンリストがあります。

注意 PowerVR2 はポリゴンリストが混在した状態では描画する事はできません
必ず各ポリゴンリスト単位で送る必要があります。

1.11 ポリゴンパラメータについて

ネイティブバッファにポリゴンパラメータを登録する事で、フレームバッファに描画ができるようになります。

ポリゴンパラメータは「Control Parameter」「Global Parameter」「Vertex Parameter」の3種類から構成されています。



1.11.1 VertexParameter

バーテックスのフォーマットは「カラーフォーマット」「UVフォーマット」の組み合わせで、15種類とクアッド形式の2種類の設定ができます。

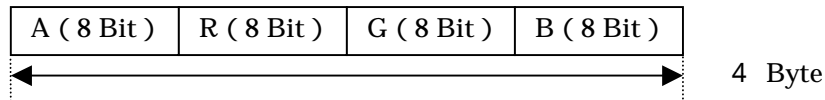
カラーフォーマットには「パック」「フロート」「インテンシティ」の3種類があり、UVフォーマットには「フロート」「16Bitフロート」の2種類があります。

| | 表示方式 | カラーフォーマット | UVフォーマット |
|---------------|--------------|-----------|-------------|
| PolygonType0 | ポリゴン | パック | |
| PolygonType1 | ポリゴン | フロート | |
| PolygonType2 | ポリゴン | インテンシティ | |
| PolygonType3 | テクスチャ | パック | フロート |
| PolygonType4 | テクスチャ | パック | 16 Bit フロート |
| PolygonType5 | テクスチャ | フロート | フロート |
| PolygonType6 | テクスチャ | フロート | 16 Bit フロート |
| PolygonType7 | テクスチャ | インテンシティ | フロート |
| PolygonType8 | テクスチャ | インテンシティ | 16 Bit フロート |
| PolygonType9 | ポリゴン (2P 用) | パック | |
| PolygonType10 | ポリゴン (2P 用) | インテンシティ | |
| PolygonType11 | テクスチャ (2P 用) | パック | フロート |
| PolygonType12 | テクスチャ (2P 用) | パック | 16 Bit フロート |
| PolygonType13 | テクスチャ (2P 用) | インテンシティ | フロート |
| PolygonType14 | テクスチャ (2P 用) | インテンシティ | 16 Bit フロート |
| SpriteType0 | クアッドポリゴン | | |
| SpriteType1 | クアッドテクスチャ | | 16 Bit フロート |

1.12 カラーフォーマット

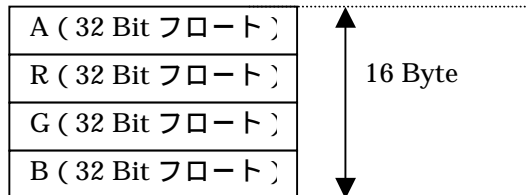
- バック カラー

ARGB を各 8 Bit で格納しています。



- フロート カラー

ARGB を各 32 Bit フロートで格納しています。



- インテンシティ カラー

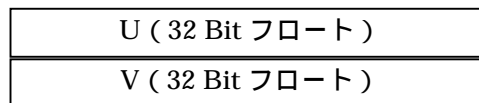
色の指定は「Global Parameter」を参照し、色の強さ情報だけです。

| |
|-------------|
| 32 Bit フロート |
|-------------|

1.13 UV フォーマット

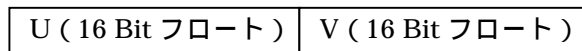
- フロート

各 UV 値を 32 Bit フロートで格納します。

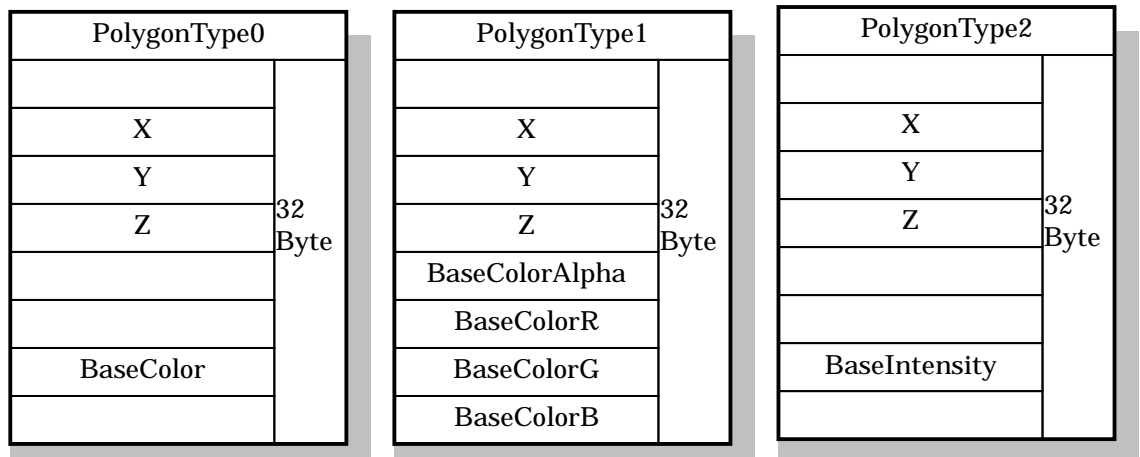


- 16 Bit フロート

各 UV 値を 16 Bit フロートで格納します。



1.13.1 タイプ一覧



| PolygonType3 | |
|--------------|------------|
| | 32 Byte |
| X | |
| Y | |
| Z | |
| U | |
| V | |
| BaseColor | |
| OffsetColor | |

| PolygonType4 | |
|--------------|------------|
| | 32 Byte |
| X | |
| Y | |
| Z | |
| U/V | |
| | |
| BaseColor | |
| OffsetColor | |

| PolygonType5 | |
|------------------|------------|
| | 64 Byte |
| X | |
| Y | |
| Z | |
| U | |
| V | |
| | |
| | |
| BaseColorAlpha | |
| BaseColorR | |
| BaseColorG | |
| BaseColorB | |
| OffsetColorAlpha | |
| OffsetColorR | |
| OffsetColorG | |
| OffsetColorB | |

| PolygonType6 | |
|------------------|------------|
| | 64 Byte |
| X | |
| Y | |
| Z | |
| U/V | |
| | |
| | |
| | |
| BaseColorAlpha | |
| BaseColorR | |
| BaseColorG | |
| BaseColorB | |
| OffsetColorAlpha | |
| OffsetColorR | |
| OffsetColorG | |
| OffsetColorB | |

| PolygonType7 | |
|-----------------|------------|
| | 32 Byte |
| X | |
| Y | |
| Z | |
| U | |
| V | |
| BaseIntensity | |
| OffsetIntensity | |

| PolygonType8 | |
|-----------------|------------|
| | 32 Byte |
| X | |
| Y | |
| Z | |
| U/V | |
| | |
| BaseIntensity | |
| OffsetIntensity | |

| PolygonType9 | |
|--------------|------------|
| | 32 Byte |
| X | |
| Y | |
| Z | |
| BaseColor0 | |
| BaseColor1 | |
| | |

| PolygonType10 | |
|----------------|------------|
| | 32 Byte |
| X | |
| Y | |
| Z | |
| BaseIntensity0 | |
| BaseIntensity1 | |
| | |
| | |

| PolygonType11 | |
|---------------|------------|
| | 64 Byte |
| X | |
| Y | |
| Z | |
| U0 | |
| V0 | |
| BaseColor0 | |
| OffsetColor0 | |
| U1 | |
| V1 | |
| BaseColor1 | |
| OffsetColor1 | |
| | |
| | |
| | |
| | |

| PolygonType12 | |
|---------------|------------|
| | 64 Byte |
| X | |
| Y | |
| Z | |
| U0/V0 | |
| | |
| BaseColor0 | |
| OffsetColor0 | |
| U1/V1 | |
| | |
| BaseColor1 | |
| OffsetColor1 | |
| | |
| | |
| | |
| | |

| PolygonType13 | |
|------------------|------------|
| | 64 Byte |
| X | |
| Y | |
| Z | |
| U0 | |
| V0 | |
| BaseIntensity0 | |
| OffsetIntensity0 | |
| U1 | |
| V1 | |
| BaseIntensity1 | |
| OffsetIntensity1 | |
| | |
| | |
| | |
| | |

| PolygonType14 | |
|------------------|------------|
| | 64 Byte |
| X | |
| Y | |
| Z | |
| U0/V0 | |
| | |
| BaseIntensity0 | |
| OffsetIntensity0 | |
| U1/V1 | |
| | |
| BaseIntensity1 | |
| OffsetIntensity1 | |
| | |
| | |
| | |
| | |

| SpriteType0 | |
|-------------|------------|
| | 64 Byte |
| AX | |
| AY | |
| AZ | |
| BX | |
| BY | |
| BZ | |
| CX | |
| CY | |
| CZ | |
| DX | |
| DY | |
| | |
| | |
| | |
| | |

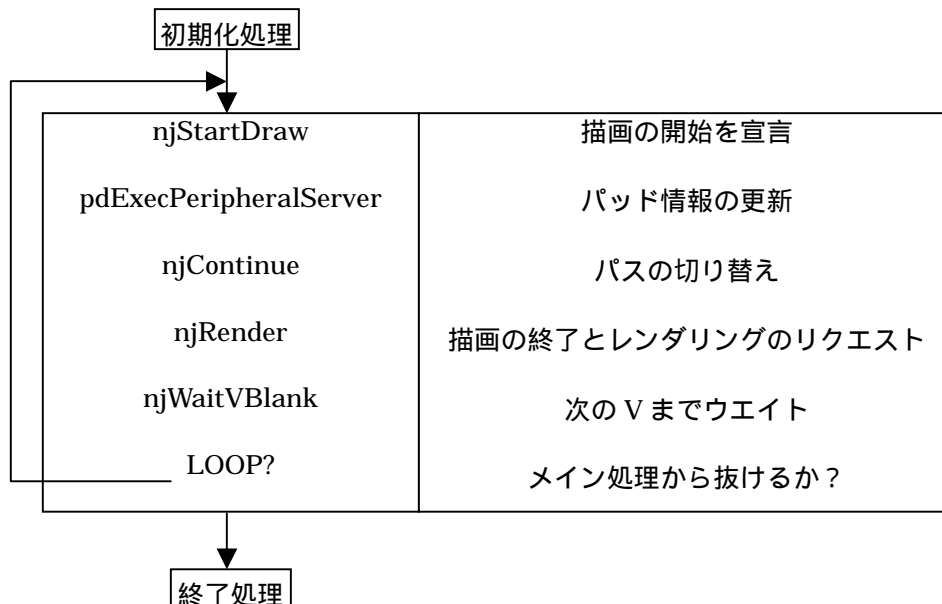
| SpriteType1 | |
|-------------|------------|
| | 64 Byte |
| AX | |
| AY | |
| AZ | |
| BX | |
| BY | |
| BZ | |
| CX | |
| CY | |
| CZ | |
| DX | |
| DY | |
| | |
| AU/AV | |
| BU/BV | |
| CU/CV | |

1.14 シーケンスについて

頂点定義からレンダリング等のフロー制御や、一連の処理の流れをシーケンスと呼びます。各種モードによって細部の動作が異なり、効率の良い処理系を組むには重要な部分です。

- メインループ

メイン処理のおおまかな流れは、次のようになります。



- フロー制御

描画周りのデータの流れをコントロールするもので、Kamui が管理しています。

フローの状態には大まかに 3 段階あり、それぞれで使用するハードウェアリソースや遷移するタイミングが異なります。

リソースが競合した場合そこでウェイトが入り無駄な時間待ちが入ることになります。それを回避する手段として各種バッファを二つ確保してダブルバッファにしています。

| 状 態 | 使用リソース | 状態遷移トリガ |
|--------|-------------------------------|----------------|
| 頂点登録 | バーテックスバッファ (ネイティブバッファ) | njRender |
| 頂点転送 | DMA ネイティブバッファ | DMA 転送終了割り込み |
| レンダリング | レンダラ ネイティブバッファ フレームバッファ | レンダリング終了割り込み |
| フリップ | フレームバッファ | VBlank-In 割り込み |

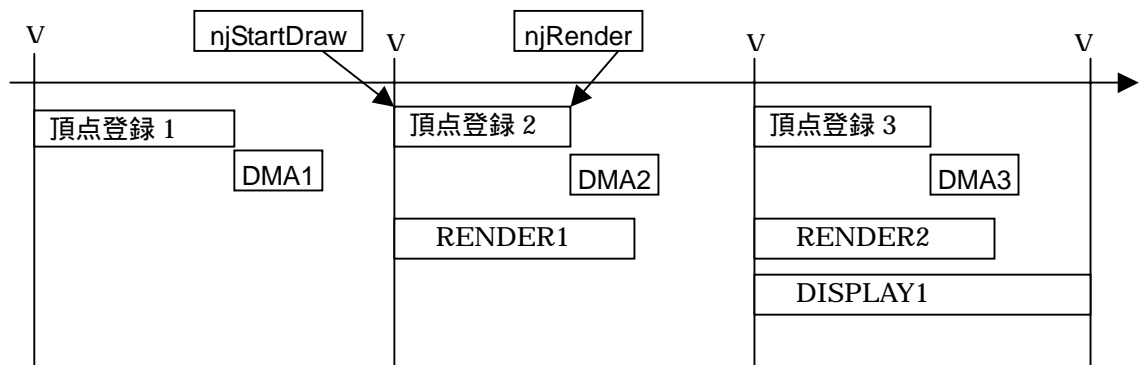
2 V レイテンシモードの場合

1.14.1 レイテンシモード

- 2 V レイテンシモード

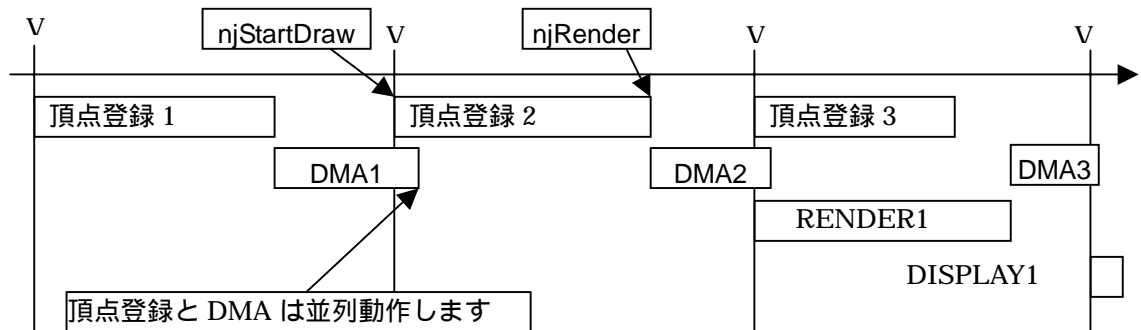
頂点登録したデータが画面に表示されるまでのレイテンシ（遅延）が 2 V あるモードです。

キー入力の反応時間を向上させる場合やバーテックスバッファの容量を減らす場合に使用します。2 V レイテンシモードでは、Ninja の `njInitVertexBufferEx` 関数でマイナスを指定したリストタイプをハードウェアに直接転送し、その他のリストタイプはすべての頂点登録後、`njRender` の関数が呼ばれたときに DMA 転送を開始します。このモードの場合、頂点登録と DMA 転送が 1 V 以内に収まらなかった場合は処理落ちします。バーテックスバッファの容量は、直接転送するリストタイプの分は必要なく、また頂点登録と DMA 転送は同時に行われないため、シングルバッファでよいので容量を減らすことができます。



- 3 V レイテンシモード

使用するリストタイプの種類が豊富で、なおかつ使用頻度がばらばらの場合、このモードを使用するとパフォーマンスが向上する場合があります。ただしバーテックスバッファをダブルバッファにする必要があり、メモリを消費します。



1.14.2 レンダリングモード

- 同期レンダリング

レンダリング開始タイミングが、V に同期します。njRender によってすべてのシーケンスが終了し、さらに DMA 転送が終了してもすぐにはレンダリングを開始しません。

レンダリングが実際に開始されるのは次の Vblank-IN になります。このモードの場合、前回のレンダリングがほんの少し V をオーバーしただけでも、その瞬間に処理落ちをしてしまいます。ただレンダリングの開始や終了のタイミングが把握しやすく、ムービー再生やテクスチャ書き換えアニメーション等、レンダリングに同期したシーケンスを組みたい場合に適したモードです。

- 非同期レンダリング

レンダリング開始のタイミングは、フレームバッファのアンロックに同期します。

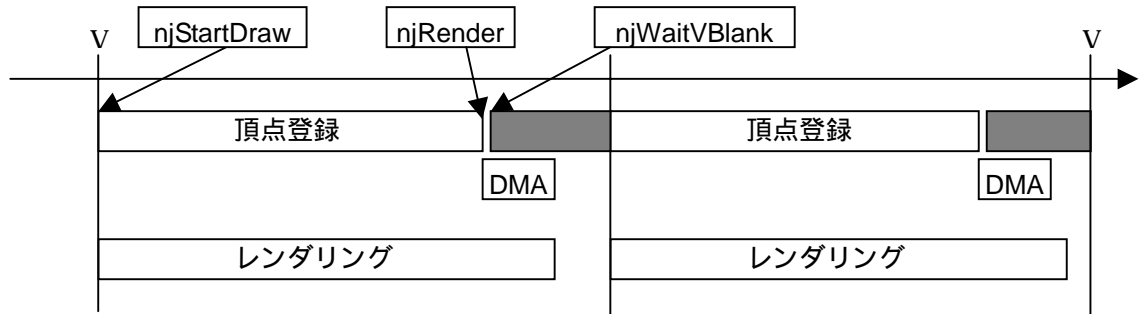
通常フレームバッファのアンロックはフリップによって非表示になった時に起こります。さらにフリップは V に同期しているのが普通ですので、実際のレンダリング開始は V に同期することになります。ところがメイン処理が間に合わずに V を少しオーバーしたイリーガルな状況が発生しても、その遅れた状態でレンダリングを開始することができます。これはメイン処理の後れをレンダリングで取り戻すことができれば処理落ちしないことを表しています。

1.14.3 シーケンスモード

- 同期シーケンス

メイン処理が V に同期して開始されるモードです。

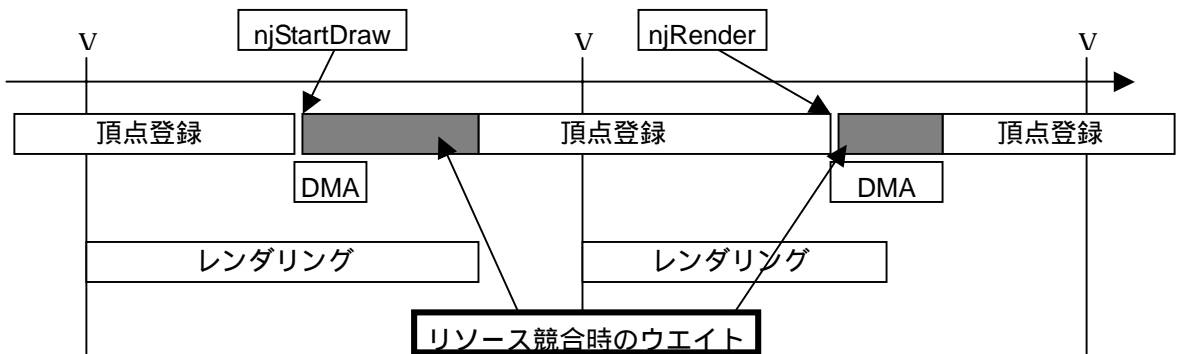
njWaitVBlank によって、強制的に V 待ちを行うことによって同期を実現しています。ミドルウェアで、ムービーを再生したりテクスチャを書き換えてアニメーションさせる場合は、通常このモードを使用します。



- 非同期シーケンス

メイン処理が、V に同期せずに開始されるモードです。

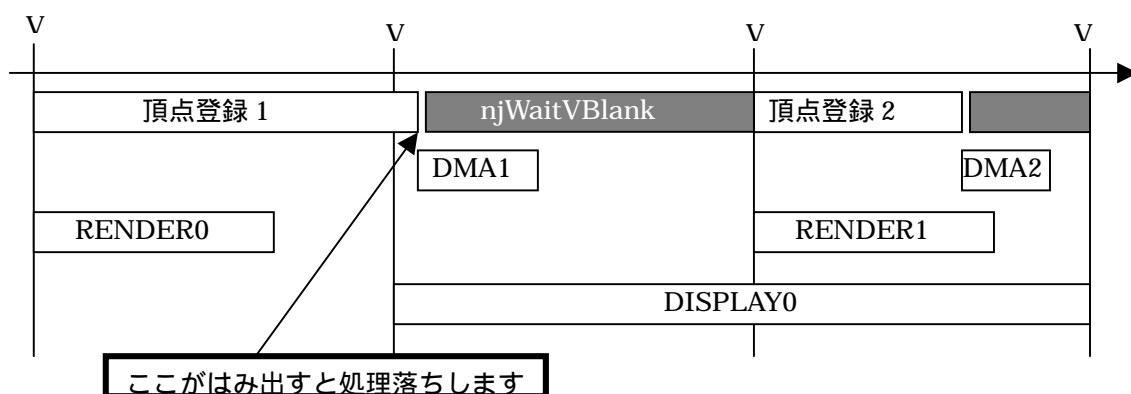
njWaitVBlank を呼ばずに常に最速のタイミングで回ります。このモードの場合、リソース（ネイティブバッファや DMA）が競合する状況になった時のみ、最低限のウェイトが入ります。一見すると、このモードのみで十分のように見えますが、ミドルウェアやパッドの取得タイミングとの絡みで、必ずしも非同期でまともに実行できるとは限りません。



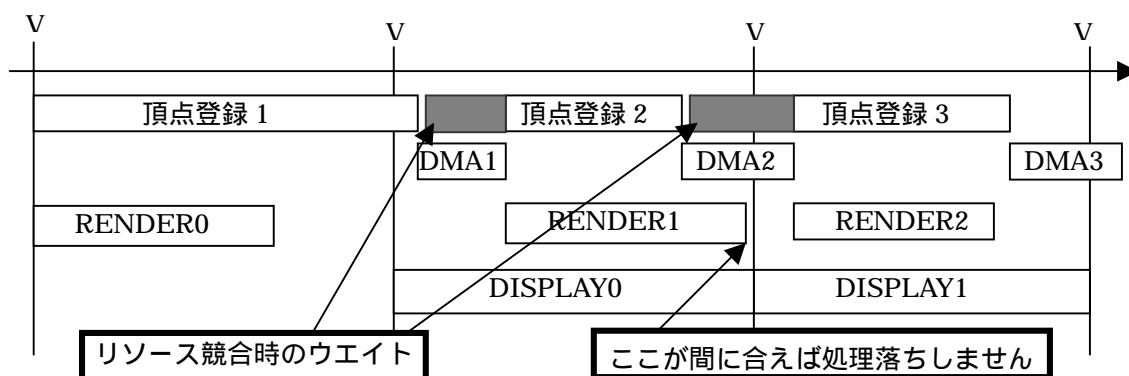
注 意 リソースの競合を検知するには、njStartDraw を実行する前に njStartCheck 関数を読んで現状を確認して下さい。

1.14.4 処理落ち

- 処理落ちした場合のシーケンス (2V + 同期)



- 処理落ちした場合のシーケンス (2V + 非同期)



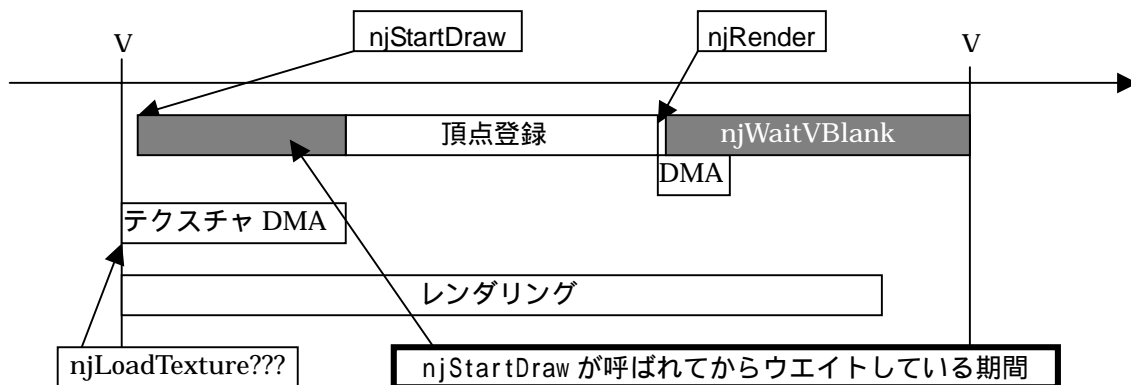
1.14.5 テクスチャの転送方式

テクスチャをメインメモリからテクスチャメモリへ転送する場合、その転送方法には CPU によるコピーと DMA による転送の 2 つの方法があります。どちらが選択されるかは、転送ソースアドレスのアライメントと呼び出しタイミングによって決定されます。

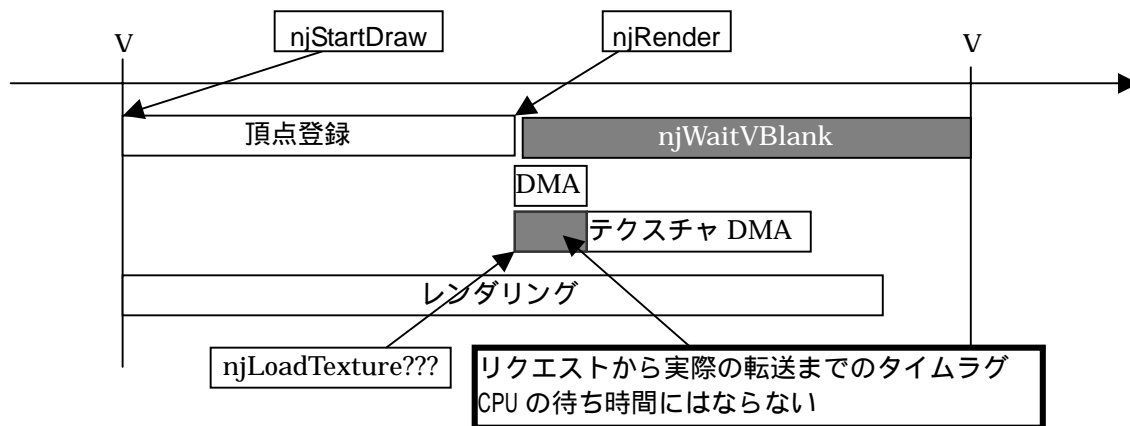
- 3 V レイテンシの場合
ソースアドレスが 32 バイトアライメントになっていれば、DMA が有効になります。
- 2 V レイテンシの場合
3 V レイテンシの条件にプラスして、njStartDraw と njRender の外側であることが必要です。2 V の場合 njStartDraw と njRender の間ではディスプレイリストの直接転送が行われる可能性があります。テクスチャの DMA 転送と直接転送は同時に動かすことができないのでその場合は CPU 転送となります。また、DMA 転送中は njStartDraw を実行できません。もし転送中に njStartDraw が呼ばれた場合は関数内部で転送が終了するまでウェイトします。
- CPU 転送
基本的に、メインメモリからテクスチャメモリへ CPU がロングワード単位で転送をし、転送が終了するまで返ってきません。
- DMA 転送
CH2DMA と呼ばれるもので、メインメモリからテクスチャメモリへ TA FIFO を経由して、32 バイト単位で転送します。CH2DMA は、他にもディスプレイリストの転送にも使用され、それらを起動するタイミングは Kamui が管理しています。
Ninja のテクスチャ転送関数を呼んだときは、転送のリクエストのみを行い、即時に復帰します。

1.14.6 テクスチャ転送シーケンス

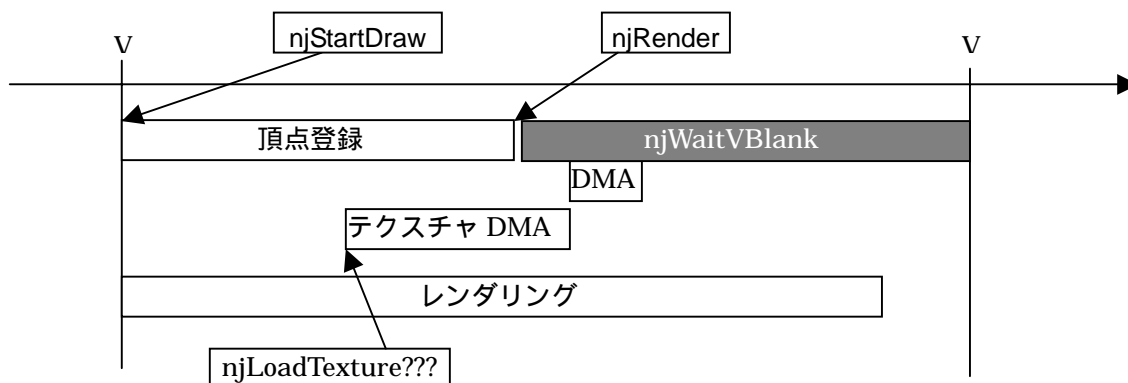
- njStartDraw の直前に呼んだ場合 (2 V + 同期)



- njRender の直後に呼んだ場合 (2 V + 同期)



- njStartDraw と njRender の間で呼んだ場合 (3 V + 同期)
2 V の場合は CPU 転送になるので、頂点登録の時間が増加するだけです。

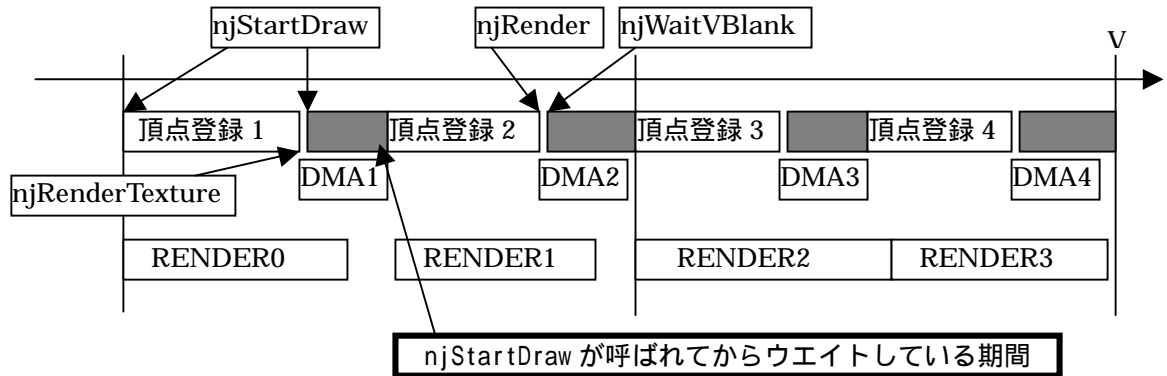


注意 上記の DMA 転送中のテクスチャをレンダリングに使用した場合、不正な絵が出力されます。同期モードの場合はテクスチャをダブルバッファにすれば回避できます。

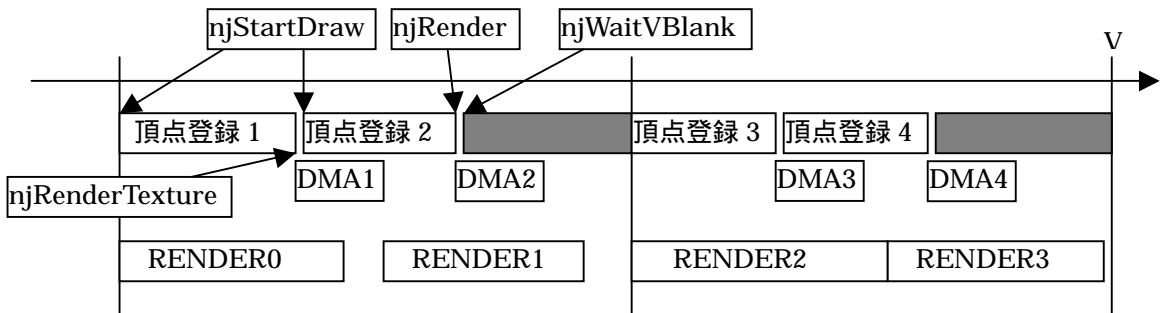
1.14.7 レンダーテクスチャ使用時のシーケンス

レンダーテキストチャとはレンダリングをテキストチャに対して行うことで、その結果を再びテキストチャとしてレンダリングに使用することができます。

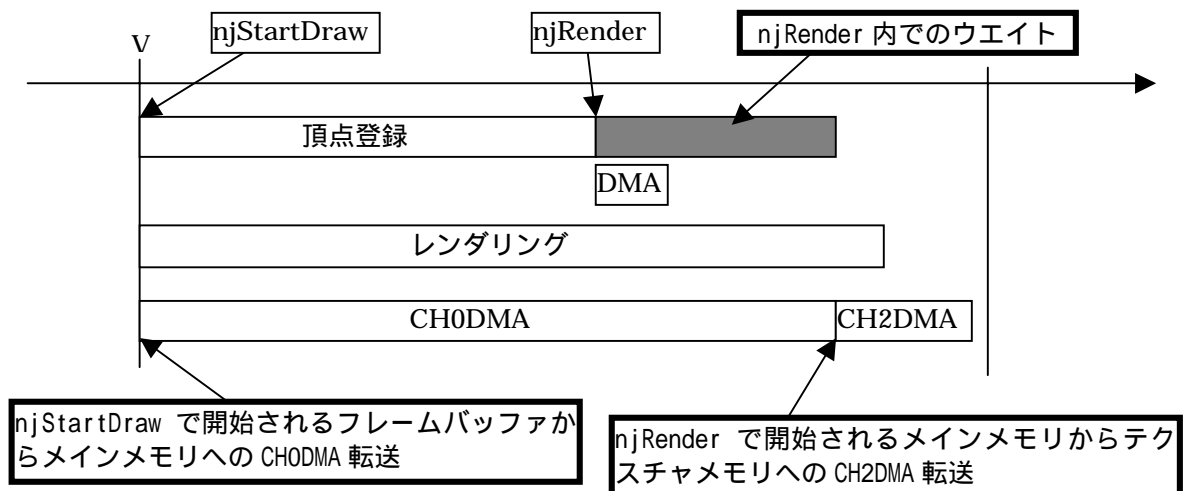
- 2 V レイテンシモード (同期)



- 3 V レイテンシモード（同期）

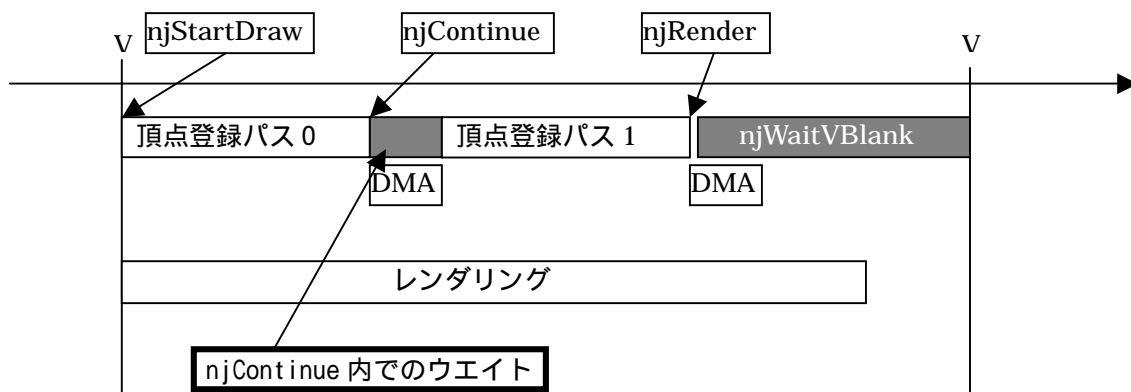


1.14.8 フレームバッファテクスチャ使用時のシーケンス（同期モード）

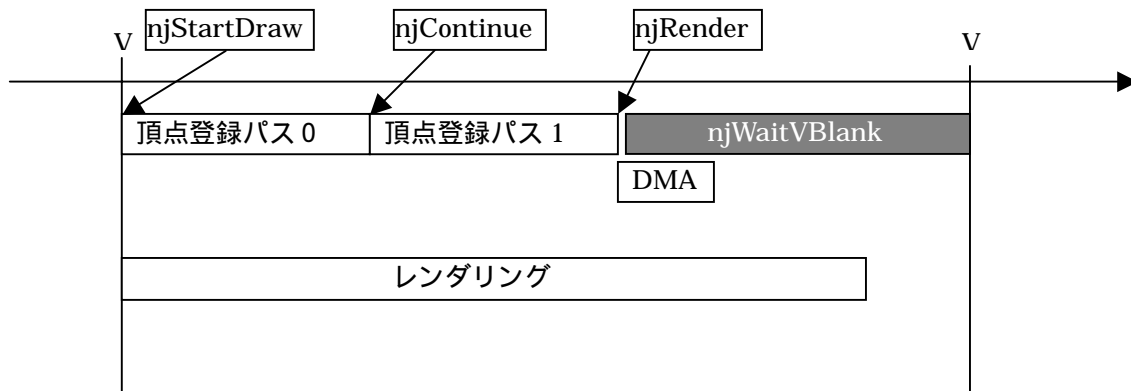


1.14.9 マルチパス使用時のシーケンス

- 2 V レイテンシモード (同期)



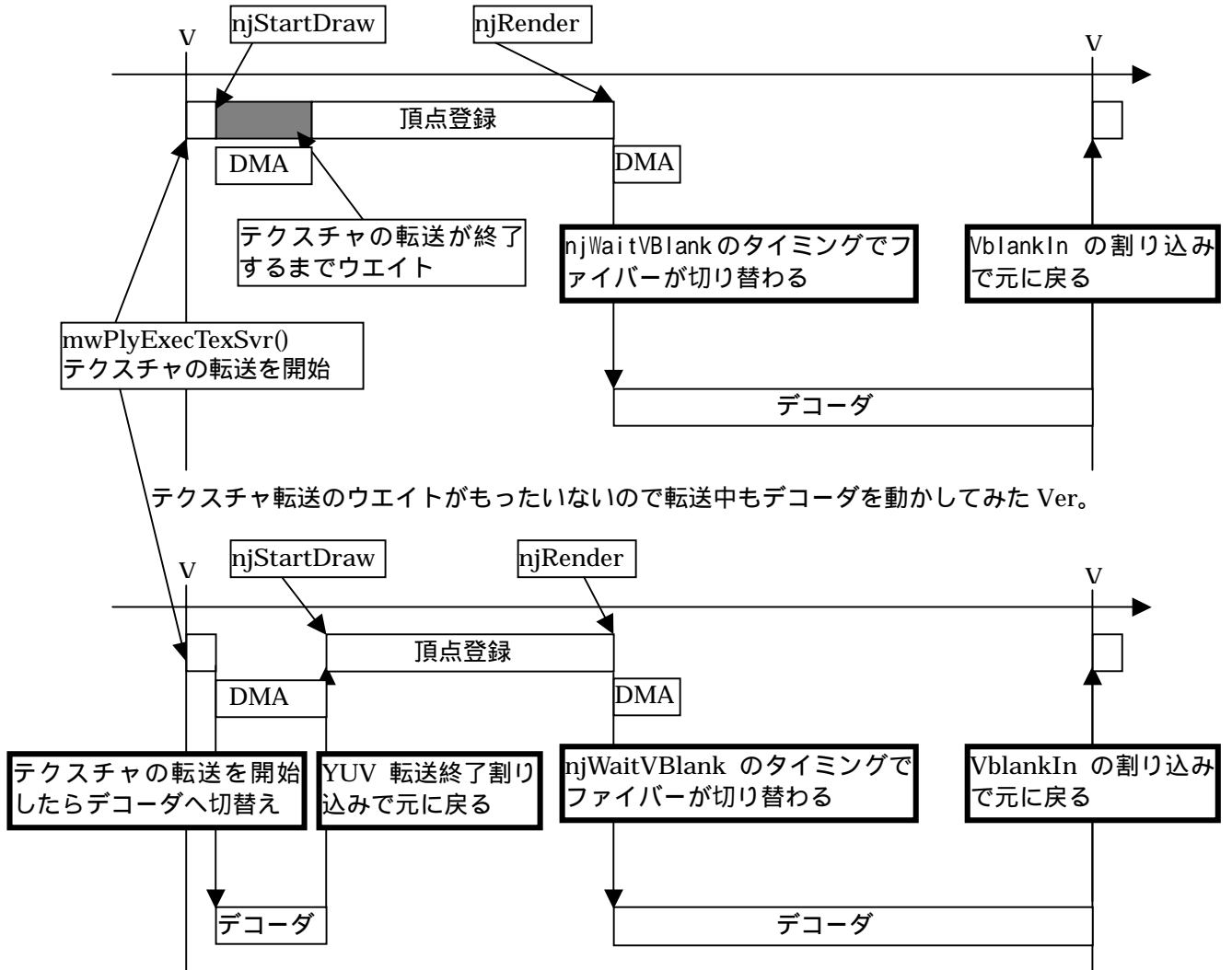
- 3 V レイテンシモード (同期)



1.14.10 ミドルウェア (Sofdec) 使用時のシーケンス (同期モード)

ミドルウェアではファイバーでデコーダを動かしています。

デコーダが起動するタイミングは、通常 njWaitVBlank が呼ばれるときで、復帰は VblankIn 割り込みです。



2

座標系

ここでは座標系について、Ninja1 から 2 への変更点を踏まえながら説明します。

初めに座標系の簡単な解説を書いた後、Ninja2 の座標系や投影変換などについて、Ninja1 との比較を交えながら説明します。

2.1 Ninja2 の座標系についての概要

Ninja1 から Ninja2 に変更され、カメラの周辺が大幅に変更されましたが、厳密に言えば、座標系の変更とカメラ関数群の修正の二つにわかれます。

2.2 座標系に関する用語の統一

3D グラフィックの世界でも座標系については、それぞれの処理系でさまざまな違いがあります。ここでは、主に座標系についての考え方と用語を統一するための解説を行います。

3次元空間上の位置を指定するために座標という概念を用いますが、座標は座標系を定めて初めて定義できるものです。逆に言えば座標系を定めなければ、ある点についての座標を表すことができません。また、別の座標系によって表現すると同じ位置が異なる座標表現で表されます。

基本的に Ninja での 3次元座標系は、デカルト座標系（直交座標系）を用います。また、「右手系」を用います。

また、点、位置、座標という用語を明確な区別を持って使用します。

点は形状を表す用語、位置は場所を表す用語、座標は位置の座標系による表現を意味する用語として用います。

注 意 座標については、同じ位置でも異なる座標で表されることがありますので、注意して下さい。

座標については、座標表現と使用することもあります。

単に平面といった場合は、無限平面を指します。

直線といった場合は、無限直線を指します。

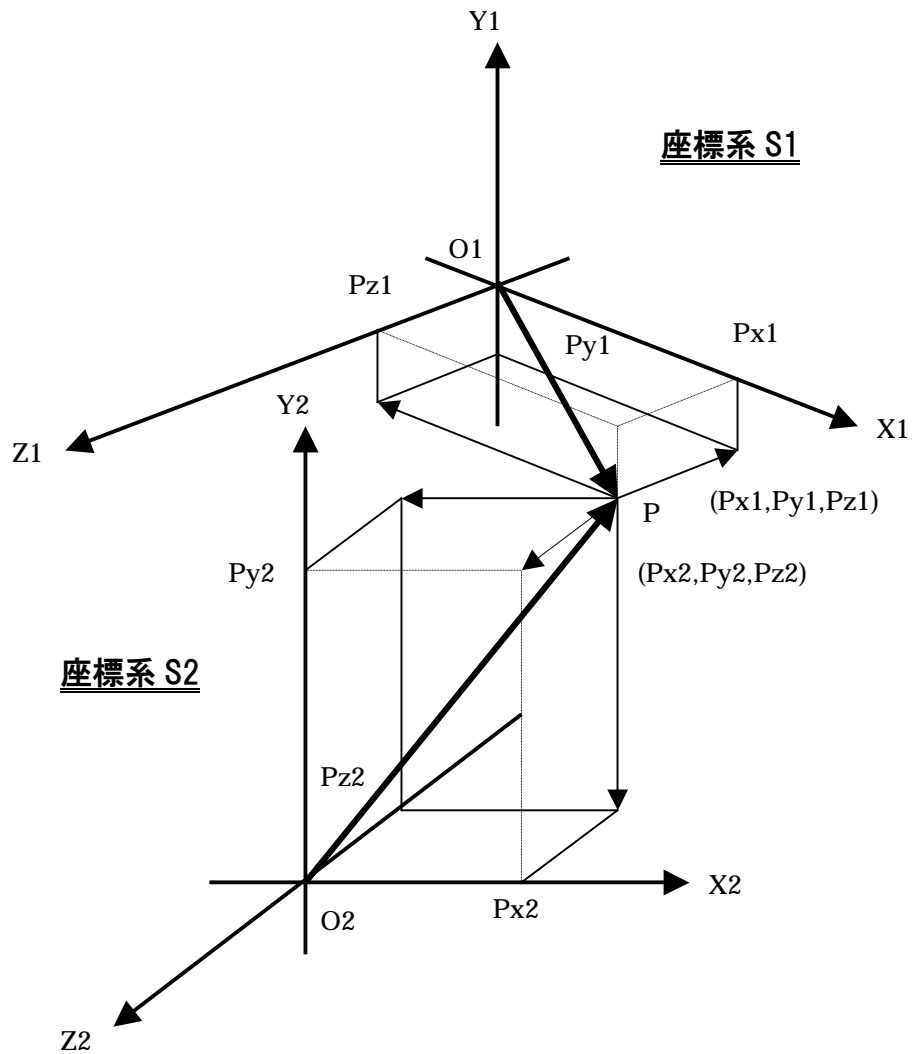
直線のある 2 点で切り取った部分を線分という言葉で表します。

ベクトルは線分に方向を持たせたものと定義します。

単に点といった場合は、座標ではなく点の実体そのものを表します。

単にベクトルといった場合は、ベクトルの座標表現ではなくベクトルの実体を指すものとします。

- 1つの点の異なる座標表現



上図は、点 P の位置を座標系 S1 で座標表現すると $(Px1, Py1, Pz1)$ ですが、座標系 S2 で座標表現すると $(Px2, Py2, Pz2)$ となる例を示しています。

それぞれの座標系の基底ベクトルを、 $ex1, ey1, ez1$, $ex2, ey2, ez2$ とすると、次の表のように書けます。

| 座標系 | 座標表現 | ベクトル表記 |
|-----|--------------------|---|
| S1 | $P(Px1, Py1, Pz1)$ | $P = Px1 \ ex1 + Py1 \ ey1 + Pz1 \ ez1$ |
| S2 | $P(Px2, Py2, Pz2)$ | $P = Px2 \ ex2 + Py2 \ ey2 + Pz2 \ ez2$ |

2.3 Ninja2 で必要な座標系

ここでは、Ninja2 で概念上必要な、世界座標系、カメラ座標系、ハードウェア座標系について説明したあと、3次元座標変換や、投影変換等を説明します。

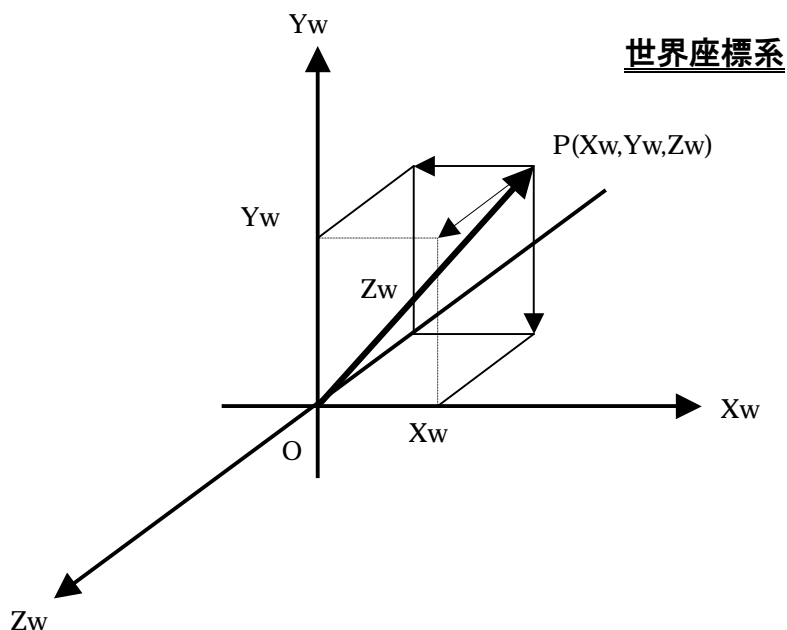
2.3.1 世界座標系

Ninja では、世界座標系という座標系を考えます。この座標系は他の座標系がどこに、どんな方向に存在するかを定義するために用いる座標系です。

世界座標系は、他の座標系の基準とするだけであり、世界座標系自体がどこにあるかは定義しません。

一般に、世界座標系はゲーム中に多くのオブジェクトの座標値が定数になるような場所に設定すると便利です。例えば、地表のマップを動き回るようなゲームでは地面に対して固定された座標系を世界座標系に選ぶとよいでしょう。太陽系内の宇宙空間を動くようなゲームでは、太陽に対し固定された座標系を世界座標系に選ぶとよいでしょう。

ある3次元空間上の点の世界座標系表現を、 (X_w, Y_w, Z_w) と書くことにします。



2.3.2 カメラ座標系

カメラ座標系は、実在するディスプレイに固定された座標系として定義します。

カメラ座標系は、カレントマトリックスにどのような行列が入っていても常にディスプレイに対して固定されています。

- カメラ座標系とディスプレイ

| | |
|-----------------|---|
| カメラ座標系の X 軸の正方向 | ディスプレイの中央から右へ向かう方向 |
| カメラ座標系の Y 軸の正方向 | ディスプレイの中央から上へ向かう方向 |
| カメラ座標系の Z 軸の正方向 | ディスプレイの中央から手前へ向かう方向 |
| カメラ座標系の原点 | ディスプレイの中央から、画面を見ている人の目の位置まで、画面に対して垂直に行った点 |

ある 3 次元空間上の点のカメラ座標系表現を、以下では (X_c, Y_c, Z_c) と書くことにします。

2.3.3 ハードウェア座標系

ハードウェア座標とは、Dreamcast のグラフィックチップに直接送ることのできる座標です。

なお、Ninja2 では 3D オブジェクトやモーションやカメラを扱う場合には、ハードウェア座標について知る必要はありません。

ハードウェア座標系での座標表現は 3 つの座標を持ちますが、通常の 3 次元空間のデカルト座標系とは座標の表し方が違います。このため、通常のマトリックスでは通常の 3 次元座標系による座標をハードウェア座標に変換することはできません (3 次元同次座標 (y, z, w) w 値を扱うとこの限りではありませんが、最終段階で除算が入ってきます)。

ハードウェア座標系へは、カメラ座標系から投影変換によって移ることができます。

ハードウェア座標が (X_h, Y_h, Z_h) であるような点は次のような意味を持ちます。

(X_h, Y_h) 座標に関してはディスプレイの左上が $(0, 0)$ になり、左上から右に X_h ドット、下に Y_h ドット行ったドットが対応しています。

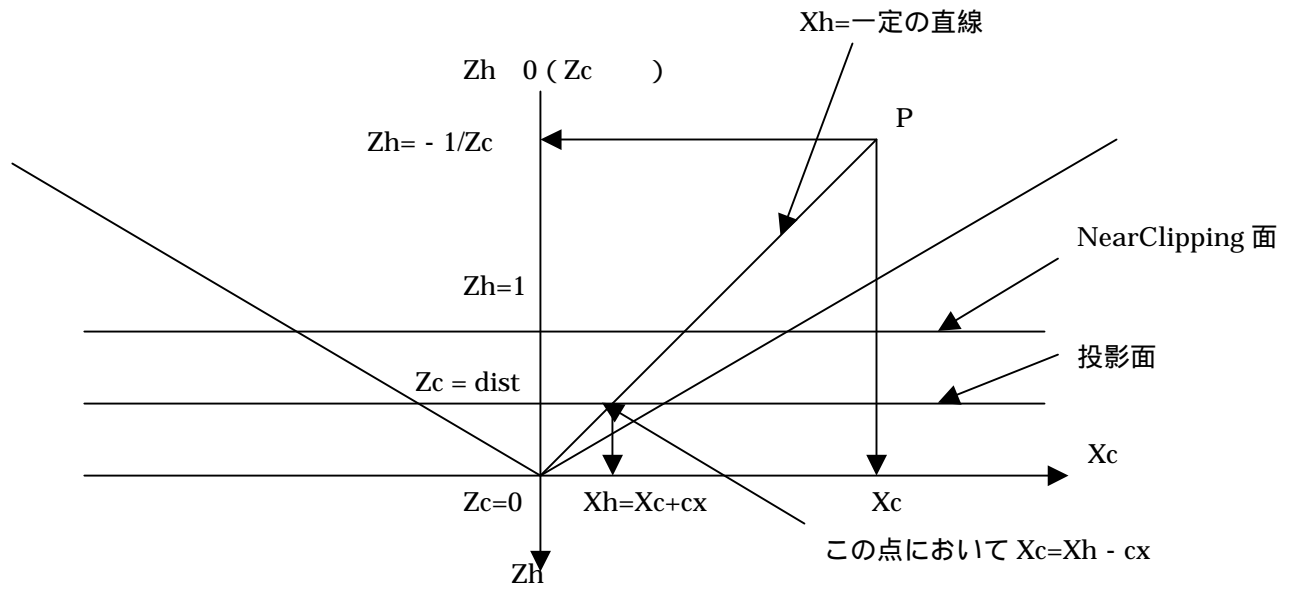
Z_h 座標はディスプレイ上のドットと X_h, Y_h 座標との対応に何らの影響を及ぼしません。つまり、 (X_h, Y_h) 座標を固定して、 Z_h 座標だけを変化させると、ディスプレイの面に対して垂直な直線となります。

Z_h は、ハードウェア的に正の値です。

3D 描画時において、ハードウェア座標の Z_h の範囲は $(0, 1)$ です。大きいほど手前、小さいほど奥です。デフォルトのニアクリッピング面 ($Z_c = -1$) にて、 $Z_h = 1$ を取ります。

なお、3D 描画部以外では、 $Z_h > 1$ も取り得ます。2D ポリゴンを 3D 描画の上から行いたいときは、 $Z_h = 2$ などの値を指定します。

Z_h 座標は描画時の優先順位を決定するためと、テクスチャマッピングにおいてのパースペクティブコレクションなどに使われます。



2.4 Ninja2 の投影変換

投影変換とは、カメラ座標系からハードウェア座標系への変換を指します。

カメラ座標が (Xc、Yc、Zc) の点は、次のような投影変換によってハードウェア座標 (Xh、Yh、Zh) になります。

Ninja2 での投影変換：カメラ座標 (Xc、Yc、Zc) ハードウェア座標 (Xh、Yh、Zh)

$$\begin{aligned} Zh &= -1 / Zc \\ Xh &= Xad * Xc * Zh + cx \\ Yh &= Yad * Yc * Zh + cy \end{aligned}$$

ただし、

| | |
|------|--------------------------------------|
| dist | 投影面距離 (カメラ座標系の原点から投影面までの距離) |
| ax | x 方向の倍率 (njSetAspect (ax、ay) 関数の ax) |
| ay | y 方向の倍率 (njSetAspect (ax、ay) 関数の ay) |
| Xad | ax * dist (正の値) |
| Yad | -ay * dist (負の値) |
| Cx | ディスプレイの中央のハードウェア X 座標 |
| Cy | ディスプレイの中央のハードウェア Y 座標 |

ハードウェア座標とカメラ座標では、Y 座標の符号が逆になります。よって、Yad は負です。

上記の透視投影変換は、njCalcScreen()関数で実装されています。ただし、njCalcScreen()関数では、Zh 座標は取得できません。

Ninja では初期化状態ではカメラ座標系の Zc = - 1 のところに Near クリッピング平面があり、Zc >= - 1 の範囲でクリッピングされます。よって、Zc の範囲は (- 、 - 1) となりますので、Zh の範囲は (0、 1) となります。

ハードウェア座標の Zh は常に正の値で、大きいほど手前、小さいほど奥であることがわかります。なお、Zh は、ハードウェア的に負の値を取れません。

なお、3 D 描画部以外では、Zh > 1 も取り得ます。2 D ポリゴンを 3 D 描画の上から行いたいときは、Zh = 2 などの値を指定します。

2.5 初期状態

初期状態では、カレントマトリックスには単位行列が入っています。この状態では、カメラ座標系が世界座標系に一致している場合に相当します。

(1) Ninja1 での図式

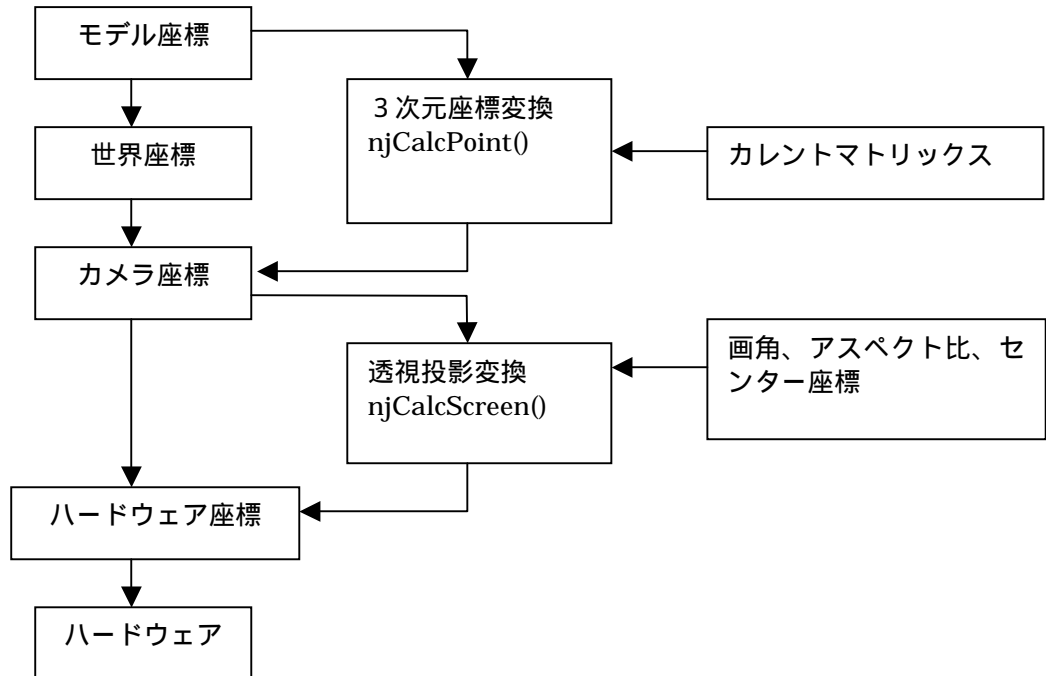
```
+----- 3D 座標系 -----+          +---2D+(1/Z)座標系---+
+モデル座標系---->世界座標系---->ベース座標系----->ハードウェア座標
+-----カレントマトリックス-----+---投影変換---+
+---通常マトリックス関数--+-----カメラ関数-----+njCalcScreen-+
```

(2) Ninja2 での図式

```
+----- 3D 座標系 -----+          +---2D+(1/Z)座標系---+
+モデル座標系---->世界座標系---->カメラ座標系----->ハードウェア座標
+-----カレントマトリックス-----+---投影変換---+
+---通常マトリックス関数--+-----カメラ関数-----+njCalcScreen-+
```


2.6 座標の流れについて

モデルを描画する際、頂点の座標が最終的にハードウェアに送られるまでの様子を図式化すると、次のようになります。



モデル座標から世界座標を得てカメラ座標へと座標変換をする部分は、実際には上図に書いたように、カレントマトリックスを座標に一回だけ乗算することにより行っています。

この部分が、njCalcPoint()関数が行う処理に相当することも示しています。

カメラ座標からハードウェア座標へ変換する部分が透視投影変換です。投影変換は njCalcScreen() 関数がほぼ相当します。しかし、njCalcScreen()関数は、ハードウェア座標のうち X、Y 成分（スクリーン座標）のみを計算します。実際にモデルを描画する場合には Z 成分も計算します。

| | |
|----------------|--|
| njCalcPoint() | 3次元座標変換を行います。 3次元空間上の点の座標表現を、カレントマトリックスに入っている変換行列で変換します。 |
| njCalcScreen() | 透視投影変換を行います。 3D空間上の点のカメラ座標表現を、スクリーン座標（ハードウェア座標系の X、Y 座標）に変換します。 |

2.7 カレントマトリックスについて

3次元座標変換は、カレントマトリックスに座標変換行列を入れることで行います。

行列の性質（結合法則）から、ある座標系からある座標系へ変換することを続けて行う作業を一つの行列にまとめて、行列と座標の一回の乗算で済ませることができます。

例えばモデルを描画する際には、概念的にはモデル座標系から世界座標系へ変換し、世界座標系からモデル座標系へ変換する必要がありますが、実際にはこれらの変換行列をあらかじめ乗算して、一つの行列にまとめたものをカレントマトリックスに入れておけば済みます。

概念的には、モデル座標系で座標 X_m と表現された頂点は、世界座標系の座標 X_w に変換されます。さらに、世界座標 X_w カメラ座標 X_c へと変換されます。

このことは、式で表すと次のようになります。

今、モデル座標系から世界座標系への変換行列を M_{wm} 、世界座標系からカメラ座標系への変換行列を M_{cw} とすると定義より、

$$\begin{aligned} X_w &= M_{wm} X_m \\ X_c &= M_{cw} X_w \end{aligned}$$

となります。

これらを連立させて、 X_w を消去すると、

$$X_c = M_{cw} M_{wm} X_m$$

となります。

この式は、モデル座標 X_m からカメラ座標 X_c への変換を表していると考えられますから、あらためてモデル座標系からカメラ座標系への変換行列を M_{cm} とすると、

$$M_{cm} = M_{cw} M_{wm}$$

となることがわかります。 M_{cm} を用いると、一気に、

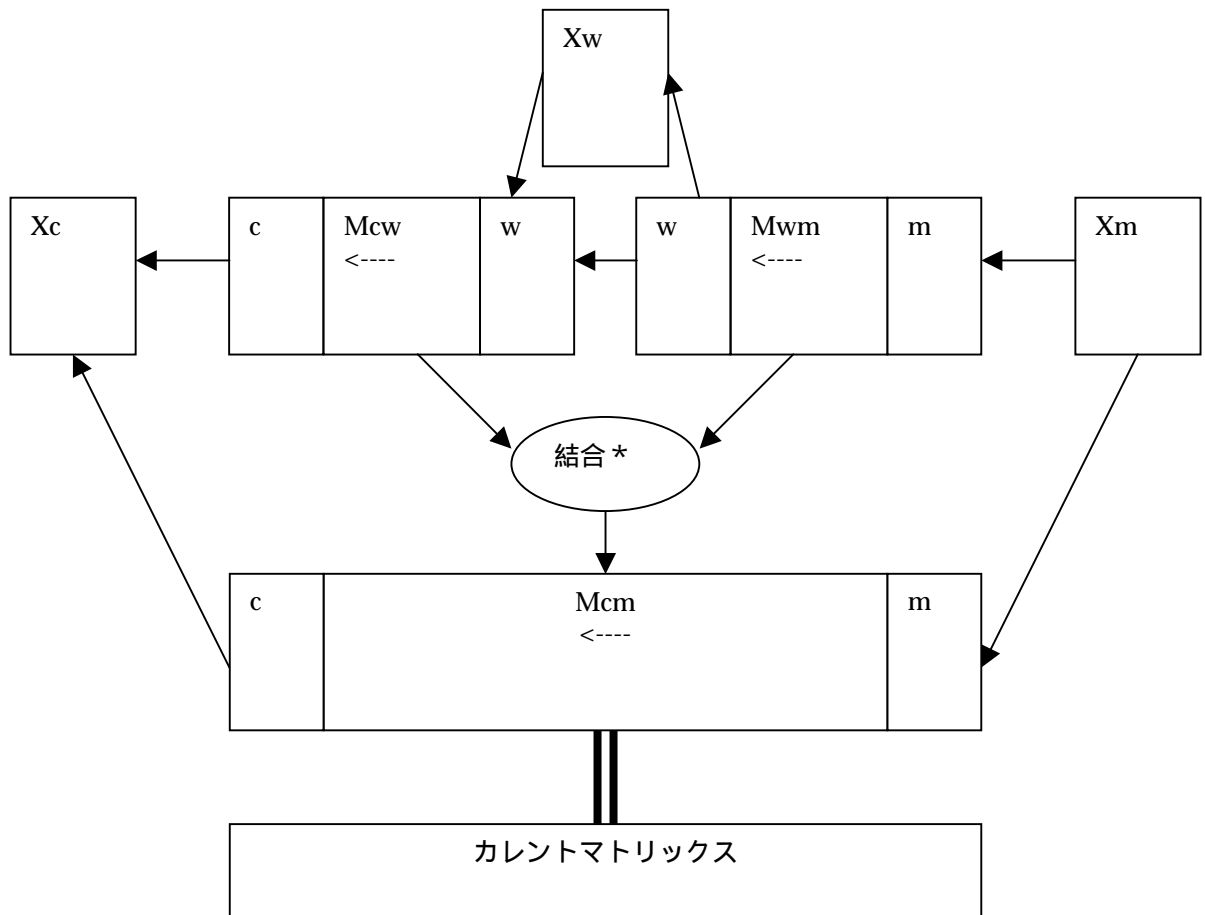
$$X_c = M_{cm} X_m$$

と書けます。

よって、カレントマトリックスに M_{cm} に相当する行列を入れておけばよいことがわかります。

カメラ関数は行列 M_{cw} を作るのに便利のように設計されています。一般のマトリックス関数は行列 M_{wm} を作るのに便利のように設計されています。ただし、逆の使い方をして問題ありません。

● イメージ図



2.8 画角やアスペクト比の設定関数について

画角やアスペクト比の設定関数は、投影変換のパラメータを設定します。

この節の内容は、「Ninja2 での投影変換」の式と対応しています。

| 関 数 | 動作内容 | 設定項目 |
|--|--|---------------|
| void njSetAspect(Float ax, Float ay) | アスペクト比を変更します。 | ax,ay |
| void njSetPerspective(Angle ang) | 画角を設定します。 njSetScreenDist(_nj_screen_.w / (njTan(ang/2)*2)); とするのと等価です。 投影面距離だけが変更されます。 | dist |
| void njSetScreen(NJS_SCREEN *scr) | 引数 scr で指定される構造体のうちの dist, cx, cy メンバだけが参照されます。 njSetScreenDist(scr->dist); njSetScreenCenter(scr->cx, scr->cy); とするのと等価です。 | dist cx,cy |
| void njSetScreenCenter(Float cx, Float cy) | カメラ座標系の原点が、ディスプレイ上のどこに投影されるかを指定します。座標値はハードウェア座標で設定します。 通常は、ディスプレイの中央のハードウェア座標を設定しますので、次のように設定します。 cx = 横方向のドット数 / 2 cy = 縦方向のドット数 / 2 | cx,cy |
| void njSetScreenDist(Float dist) | 投影面距離を設定します。 | dist |

注 意 どの関数を実行しても、
アスペクト比 ax,ay
中心座標 cx,cy
投影面距離 dist
のみが設定されることに、注意して下さい。

投影変換の式に出てくる、Xad、Yad はアスペクト比 ax、ay や投影面距離 dist が変更されたときには、自動的に修正されます。

njSetScreen()は、njSetScreenDist()と njSetScreenCenter()の組み合わせと等価です。
njSetPerspective()は、njSetScreenDist()で設定する内容を「画角」によって指定するだけです。

3 カメラ

ここではカメラについて、新規の内容を中心に説明します。

3.1 概要

新カメラ関数群のうち、基本カメラ関数群は、カレントマトリックス以外の内部的な変数に一切影響を及ぼさず、単なるカレントマトリックスの操作関数として機能します。

基本カメラ関数群は、カレントマトリックスを暗黙の引数に取り、入力も出力もカレントマトリックスに対して行います。

よって、基本カメラ関数群を単なる新しいマトリックスの操作関数として使用することが可能です。

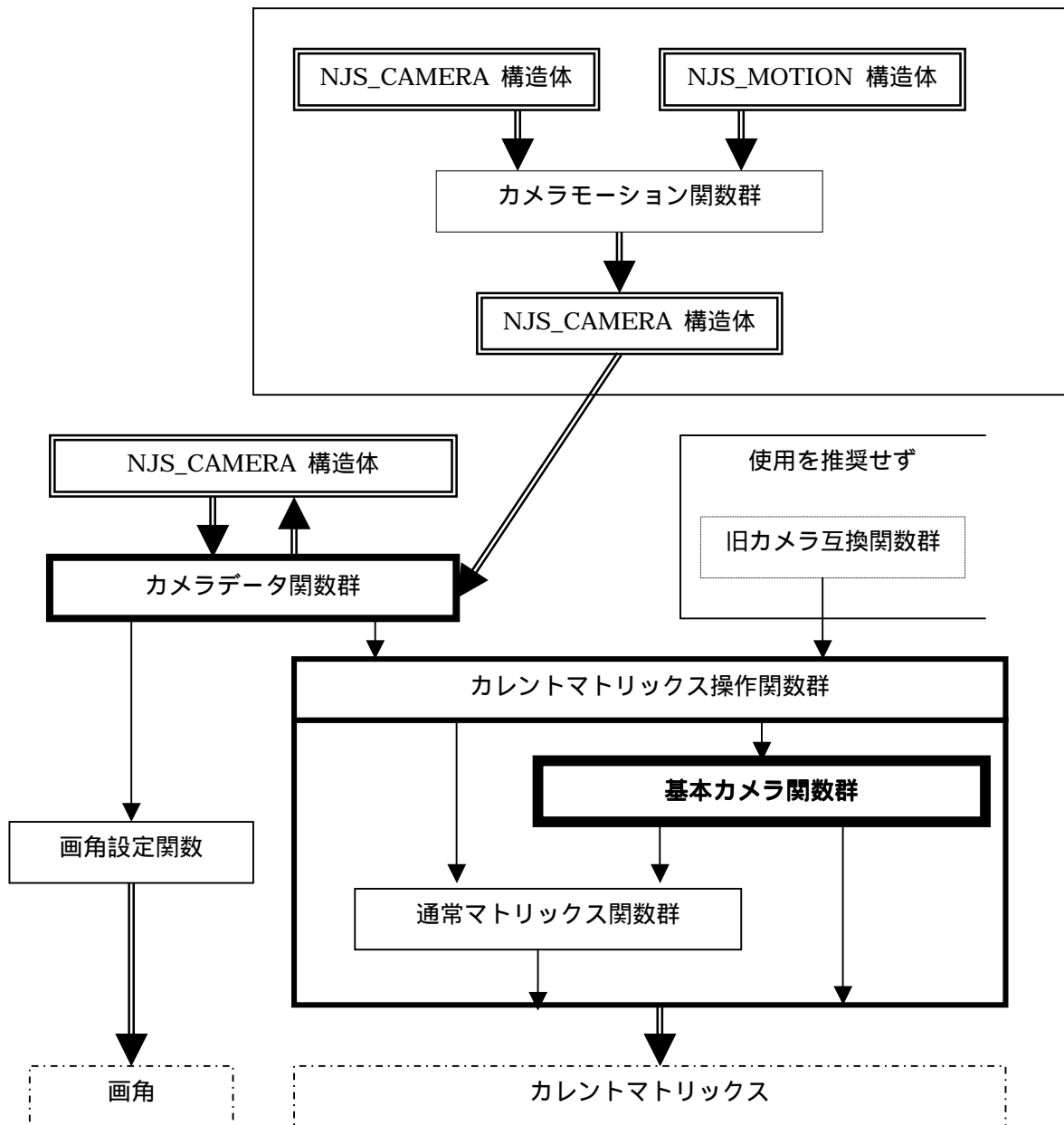
旧カメラ関数群では、`njSetCamera()`関数を呼び出すと、カレントマトリックスの位置がベース位置に戻りましたが、新カメラ関数群ではカレントマトリックスの位置は変化しません。

基本カメラ関数群は、カレントマトリックスの左から、その関数で作成したマトリックスを乗算することに等価な演算を行います。

通常、カメラ関数を呼び出すときは、カレントマトリックスに、`Mcw`（世界座標系からカメラ座標系への変換行列）を入れておきます。最初は単位行列でも構いません。

通常の使い方は、まず、`njUnitMatrix(NULL);` か、または、`njPushMatrix(&_nj_unit_matrix_);` を行ってから、`njCameraExLookAt()`や、`njRotateXCameraEx()`などを組み合わせます。

3.2 カメラ関数群の構成



- 基本カメラ関数群はカレントマトリクスを操作するだけですから、通常マトリクス関数群と合わせて、カレントマトリクス操作関数群と考えることができます。
- 旧カメラ互換関数群は、カレントマトリクス操作関数を用いて組まれています。
- カメラデータ関数群は、NJS_CAMERA 構造体を初期化したり、NJS_CAMERA 構造体の内容をカレントマトリクスや画角に反映したりする関数の集合です。

3.3 新カメラ関数群

3.3.1 基本カメラ関数群

(1) 特徴

基本カメラ関数群は、カメラ座標系を操作しやすい形で実装されたカレントマトリックスを操作する関数の集合です。カレントマトリックス以外は変更しません。

`njTranslateAbsoluteCameraEx()`関数以外は、引数で指定された操作量だけで決定される行列を、カレントマトリックスの左から乗算します (`cur = M(引数) cur`)。

ただし、`njTranslateAbsoulteCameraEx()`関数は、`cur = cur Tr(-x,-y,-z)`という動作をします。

(2) 関数

- カメラ座標系をカメラ座標系の軸の周りに回転


```
void njRotateXCameraEx( Angle ang );
void njRotateYCameraEx( Angle ang );
void njRotateZCameraEx( Angle ang );
void njRotateCameraEx( Angle *ang, Sint32 lv );
```
- カメラ座標系をカメラ座標系の軸に沿って平行移動


```
void njTranslateCameraEx( Float x, Float y, Float z );
```
- カメラ座標系を世界座標系の軸に沿って平行移動


```
void njTranslateAbsoluteCameraEx( Float x, Float y, Float z );
```
- カメラ座標系の原点の位置を世界座標系表現で取得


```
void njGetCameraExPosition( NJS_POINT3 *pos );
```
- カメラ座標系の原点の位置を世界座標系表現で設定


```
void njSetCameraExPosition( const NJS_POINT3 *pos );
```
- カメラの位置と向きと上方向をカメラ座標系表現で設定


```
void njCameraExLookAt( const NJS_POINT3 *eye, const NJS_POINT3 *center,
const NJS_VECTOR *up );
```

3.3.2 カメラデータ関数群

(1) 特徴

カメラデータ関数群は、`NJS_CAMERA` 構造体を初期化したり、`NJS_CAMERA` 構造体の内容をカレントマトリックスと画角に反映する関数の集合です。

カレントマトリックス操作関数群や、画角設定関数を用いて組まれています。

(2) 関数

- カメラ構造体を初期化


```
void njInitCamera( NJS_CAMERA *camera );
```
- カメラ構造体の内容をカレントマトリックスと画角に反映


```
void njSetCamera( const NJS_CAMERA *camera );
```

3.3.3 旧カメラ互換関数群

(1) 特徴

旧カメラ互換関数群は、旧カメラ関数と同様な働きをする関数を基本カメラ関数群を用いて組上げたものです。ただし、次のような変更点や注意事項があります。

第一引数に NJS_CAMERA 構造体を取らなくなりました。

ドキュメント化されている範囲内で同様の動作が行えます。

旧カメラ関数群にあったように、真上方向で前後が急に逆転したりするようなことはありません。

旧カメラ関数群では、Yaw、Roll、Pitch 関数で回転を組み合わせたとき、予期しないような動作が起こる場合がありますが、新カメラ関数群では起こりません。

(2) 関数

- カレントマトリックスを単位行列に初期化
void njInitCameraEx(void);
- カメラ座標系をカメラ座標系の軸の周りに回転
void njPitchCameraInterestEx(Angle ang);
void njYawCameraInterestEx(Angle ang);
void njRollCameraInterestEx(Angle ang);
- カメラ座標系をカメラ座標系の軸に沿って平行移動
void njForwardCameraPositionEx(Float x);
- カメラ座標系を世界座標系の軸に沿って平行移動
void njTranslateCameraPositionEx(Float x, Float y, Float z);
- カメラ座標系の原点を世界座標系の軸の周りに回転
void njRotateCameraPositionXEx(Angle ang);
void njRotateCameraPositionYEx(Angle ang);
void njRotateCameraPositionZEx(Angle ang);
- カメラの位置をそのままにして、向きだけを世界座標系表現で設定
void njPointCameraInterestEx(Float x, Float y, Float z);

3.4 新カメラ関数群と旧カメラ関数群の比較

3.4.1 旧カメラ関数群

| | |
|---|----------------------------|
| njInitCamera()関数 | NJS_CAMERA構造体を初期化 |
| 通常カメラ関数 (njPitchCameraInterest() など) | NJS_CAMERA構造体を参照 & 変更 |
| njSetCamera()関数 | NJS_CAMERA構造体 カレントマトリックス構築 |

- 通常のカメラ関数は NJS_CAMERA 構造体に値を設定します。
- njSetCamera()関数が NJS_CAMERA 構造体の設定内容に基づき、カレントマトリックスを「ゼロから構築」します。
- njInitCamera()関数は、NJS_CAMERA 構造体を初期化します。
- 使用法としては、njInitCamera()関数で NJS_CAMERA 構造体を初期化してから、njPointCameraInterest() 関数や、njPitchCameraInterest()関数で NJS_CAMERA 構造体を設定した後、njSetCamera()関数でそれに対応したカレントマトリックスを構築します。このとき、カレントマトリックスポインタはベースマトリックスへ初期化されます。
- NJS_CAMERA 構造体の公開メンバだけにユーザーが直接値を代入してから njSetCamera()関数を呼び出しても、設定内容とおりにならないことがありました。このため、NJS_CAMERA 構造体には直接触れず、必ず通常カメラ関数を用いてアクセスしなければなりません。

3.4.2 新カメラ関数群

| | |
|--|---|
| njUnitMatrix(NULL) | カレントマトリックスを単位行列にする。 |
| 基本カメラ関数群 (njRotateXCameraEx() など) | カレントマトリックスを参照 & 変更 ほとんどの関数 M(引数) cur cur njTranslateAbsoulteCameraEx() cur M(引数) -- cur |
| njInitCamera() 関数 | NJS_CAMERA構造体を初期化 |
| njSetCamera() 関数 | NJS_CAMERA構造体-- カレントマトリックスの左から乗算 |

- 基本カメラ関数群は結果をカレントマトリックスに返します。
- nac ファイルに存在する NJS_CAMERA 構造体や、ユーザーが直接値を設定した NJS_CAMERA 構造体をカレントマトリックスに反映させるために njSetCamera()が存在します。njSetCamera()関数はカレントマトリックスの左から NJS_CAMERA 構造体に相当するマトリックスを乗算します。njSetCamera()関数は基本カメラ関数群を用いて作成されています。カレントマトリックスポインタを変更しません。
- カメラモーション関数は内部で njSetCamera()関数を呼び出します。
- 基本カメラ関数は、カレントマトリックス以外の内部変数を参照、変更しません。
- njInitCamera() 関数は、NJS_CAMERA 構造体を初期化するために存在します。カレントマトリックスを初期化するには、njUnitMatrix(NULL);を使用します。
- 最も簡単な使い方としては、最初に njUnitMatrix(NULL);を行ってカレントマトリックスを単位行列にしてから（カメラ座標系が世界座標系に一致している状態になります）、njCameraExLookAt() 関数や、njRotateXCameraEx() 関数、njTranslateCameraEx() 関数などを組み合わせます。カレントマトリックスが段階を追って構築されていきます。
- 新カメラ関数群は、単位行列から作業を始める必要はありません。カレントマトリックス操作関数群などによって作成された、任意のカレントマトリックスから作業を行えます。

3.5 新旧カメラ関数対応表

| 旧カメラ関数 | 新カメラ関数 |
|--|--|
| NJS_CAMERA 構造体 | カレントマトリックス (World Camera 変換行列: Mcw) NJS_CAMERA 構造体 |
| njInitCamera() | njUnitMatrix(NULL)または njInitCameraEx() カレントマトリックスを初期化します。 njInitCamera() NJS_CAMERA 構造体を初期化します。 |
| njSetCamera() | 特になし カレントマトリックスを直接操作できます。 njSetCamera() NJS_CAMERA 構造体の内容をカレントマトリックスに反映させます。 |
| njPitchCameraInterest() njYawCameraInterest() njRollCameraInterest() | njRotateXCameraEx() njRotateYCameraEx() njRotateZCameraEx() 新カメラ関数群においては、回転の方向は後で述べる「対応関係」によって決められています。 |
| njForwardCameraPosition() | njTranslateCameraEx() カメラ座標系の Z 軸のみでなく、X 軸や Y 軸方向の平行移動もできるように拡張されています。カメラ座標表現で平行移動のベクトルを指定します。 |
| njPointCameraInterest() | njCameraExLookAt() カメラの位置と上方向ベクトルも指定するようになりました。 |
| njTranslateCameraPosition() | njTranslateAbsoluteCameraEx() 世界座標系表現で平行移動のベクトルを指定します。 |
| NJS_CAMERA 構造体の px,py,pz からカメラの位置を取得 | njGetCameraExPosition() |
| NJS_CAMERA 構造体の px,py,pz にカメラの位置を設定 | njSetCameraExPosition() |

3.6 基本カメラ関数群と通常マトリックス関数群の対応

`njRotateX()`系関数や、`njTranslate()`関数と、`njRotateXCameraEx()`系関数や
`njTranslateCameraEx()`関数とは、対応関係があります。

カメラをモデルと同様に一つのオブジェクトとして考えると、通常マトリックス関数群がモデルを操作するのと全く同様に、基本カメラ関数群はカメラを操作すると考えられます。

`njRotateX()`関数は、モデルを直前のモデル座標系のX軸の周りに回転した位置に描画できるように、カレントマトリックスを変更します。

一方、`njRotateXCameraEx()`関数は、カメラを直前のカメラ座標系のX軸の周りに回転させた時に見える場面を描画できるようにカレントマトリックスを変更します。

同様に、`njTranslate()`関数は、モデルを直前のモデル座標系のX Y Z軸に沿って平行移動した位置に描画できるようにカレントマトリックスを変更します。

一方、`njTranslateCameraEx()`関数は、カメラを直前のカメラ座標系のX Y Z軸に沿って平行移動させた時に見える場面を描画できるようにカレントマトリックスを変更します。

カレントマトリックスに、座標系 S_a から座標系 S_b への変換行列 M_{ba} が入っている状態では、通常マトリックス関数群では、 S_a の座標表現で 座標系 S_a を回転や平行移動し、基本カメラ関数群は、 S_b の座標表現で 座標系 S_b を回転や平行移動するといえます。これは、マトリックスに対する乗算の方向がそれぞれ異なっていることに起因します。

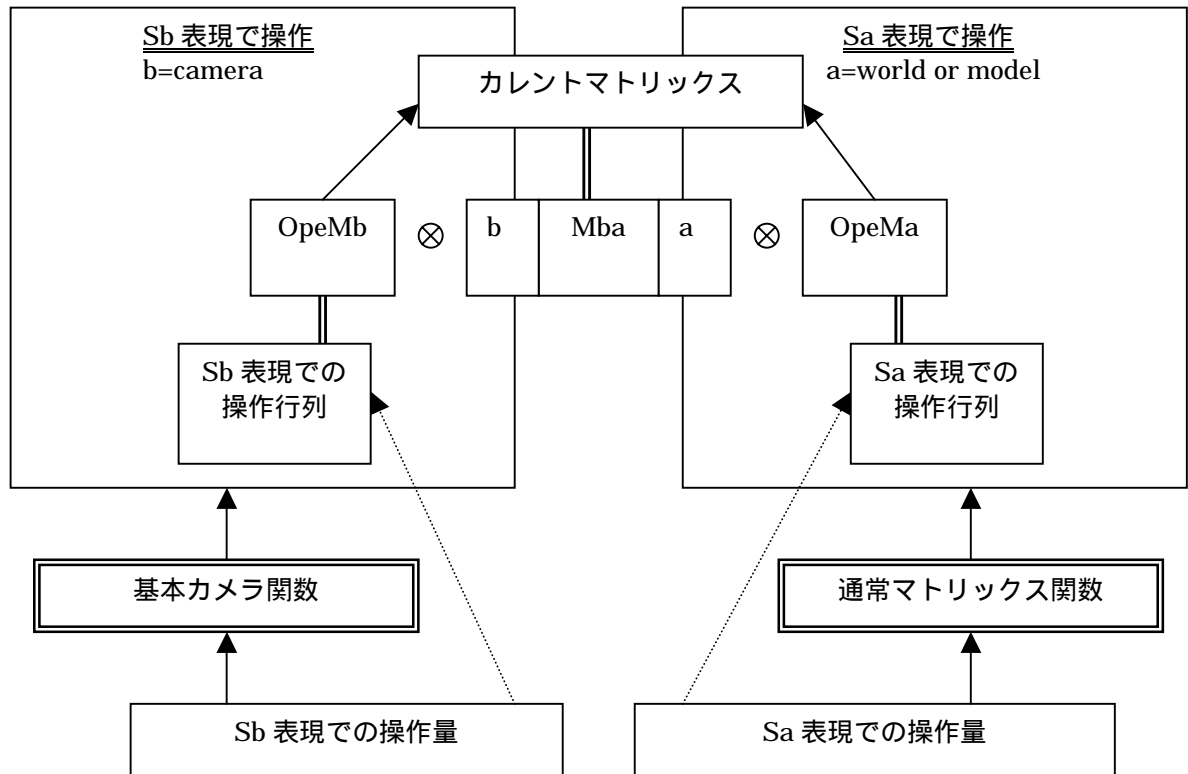
通常マトリックス関数群は、引数で指定された操作量だけに依存する行列をカレントマトリックスの右から乗算します。

一方、基本カメラ関数群は、引数で指定された操作量だけに依存する行列をカレントマトリックスの左から乗算します（ただし、`njTranslateAbsoluteCameraEx()`は除く）。

通常は、カレントマトリックスには、世界座標からカメラ座標への変換行列 M_{cw} または、モデル座標からカメラ座標への変換行列 M_{cm} が入っているため、通常マトリックス関数群は、モデル座標系表現でモデルを動かし、基本カメラ関数群はカメラ座標系表現でカメラを動かすことになります（ $S_b=S_c$ ）。

以上のことを図に示すと、次のようになります。

基本カメラ関数と通常マトリックス関数



基本カメラ関数及び、通常マトリックス関数は、cur をカレントマトリックスとすると以下のような演算を行います。

| 基本カメラ関数 | 操 作 | 操作行列 OpeMb |
|----------------------------|------------------------|--------------|
| njRotateXCameraEx(angx) | cur = Rx(-angx) cur | Rx(-angx) |
| njRotateYCameraEx(angy) | cur = Ry(-angy) cur | Ry(-angy) |
| njRotateZCameraEx(angz) | cur = Rz(-angz) cur | Rz(-angz) |
| njTranslateCameraEx(x,y,z) | cur = Tr(-x,-y,-z) cur | Tr(-x,-y,-z) |

すべての表現が、座標系 Sb (=Sc) による表現です。

| 通常マトリックス関数 | 操 作 | 操作行列 OpeMa |
|-------------------------|---------------------|------------|
| njRotateX(NULL,angx) | cur = cur Rx(angx) | Rx(angx) |
| njRotateY(NULL,angy) | cur = cur Ry(angy) | Ry(angy) |
| njRotateZ(NULL,angz) | cur = cur Rz(angz) | Rz(angz) |
| njTranslate(NULL,x,y,z) | cur = cur Tr(x,y,z) | Tr(x,y,z) |

すべての表現が、座標系 Sa による表現です。

3.7 新カメラ関数と通常マトリックス関数の関係の例 1

下記の通常マトリックス関数の組み合わせで作成されるカレントマトリックスは、

```
njUnitMatrix(NULL);
njTranslate(NULL,x,y,z);
njRotateX(NULL,angx);
njRotateY(NULL,angy);
njRotateZ(NULL,angz);
njInvertMatrix(NULL);
```

下記の新カメラ関数の組み合わせで作成されるカレントマトリックスと同じです。

```
njUnitMatrix(NULL);
njTranslateCameraEx(x,y,z);
njRotateXCameraEx(angx);
njRotateYCameraEx(angy);
njRotateZCameraEx(angz);
```

3.8 新カメラ関数と通常マトリックス関数の関係の例 2

新カメラ関数は次のように通常マトリックス関数の組み合わせでも等価な演算が行えます。ただし、新カメラ関数の方が圧倒的に高速です。

| カメラ関数 | 通常マトリックス関数 |
|-----------------------------|--|
| njRotateXCameraEx(ang) | njInvertMatrix(NULL); njRotateX(NULL,ang); njInvertMatrix(NULL); |
| njRotateYCameraEx(ang); | njInvertMatrix(NULL); njRotateX(NULL,ang); njInvertMatrix(NULL); |
| njRotateZCameraEx(ang); | njInvertMatrix(NULL); njRotateZ(NULL,ang); njInvertMatrix(NULL); |
| njTranslateCameraEx(x,y,z); | njInvertMatrix(NULL); njTranslate(NULL,x,y,z); njInvertMatrix(NULL); |

3.9 新カメラ関数と通常マトリックス関数の関係の例 3

次のプログラムはカメラ関数の理解を深めるための、カメラとモデルを同じ位置に配置させるプログラム例です。

カメラとモデルを共に、任意の平行移動量 tx、ty、tz や回転角 rx、ry、rz だけ移動させても、カメラから見るとモデルは止まって見えてしまいます。

このことに対応して、描画関数が呼ばれる時点のカレントマトリックスは単位行列です。

```
Float      tx;
Float      ty;
Float      tz;
Angle      rx;
Angle      ry;
Angle      rz;

njPushMatrix( &_nj_unit_matrix_ );

//カメラのマトリックス (Mcw) を作ります
njTranslateCameraEx( tx, ty, tz );
njRotateXCameraEx( rx );
njRotateYCameraEx( ry );
njRotateZCameraEx( rz );

//この時点でカメラが、世界座標表現で位置(tx,ty,tz),角度(rx,ry,rz)に
//配置されます。

njPushMatrix( NULL );

//モデルのマトリックス (Mwm) を作ります
njTranslate( NULL, tx, ty, tz );
njRotateX( NULL, rx );
njRotateY( NULL, ry );
njRotateZ( NULL, rz );

//この時点でモデルも、カメラと同じ位置 (世界座標表現で位置(tx,ty,tz),
//角度(rx,ry,rz)) に配置されるため、カメラから見るとモデルは正面に
//初期位置のまま止まって見えます。
//このことに対応して、この時点のカレントマトリックスは誤差を除いて
//単位行列(_nj_unit_matrix_)です。

// (ここで描画関数を呼びます)

njPopMatrix( 1 );
njPopMatrix( 1 );
```


4 テクスチャ

ここではテクスチャについてのシステムと内部動作、テクスチャフォーマットについて説明します。

4.1 語句説明

Ninja で使用するテクスチャの語句と意味を説明します。

- **テクスチャ**

Ninja でのテクスチャとは、2D グラフィックス、3D グラフィックス、スプライト、スクロール、モデルなどに貼り付ける画像をすべてさします。Ninja で使用できるテクスチャサイズは 8,16,32,64,128,256,512,1024 ドットの縦、横を持つものです。

- **テクスチャリスト**

同時に使用するテクスチャの集合をまとめたものをテクスチャリストと呼びます。Ninja ではテクスチャリスト単位でテクスチャ操作をするのが基本になります。

- **テクスチャ番号**

テクスチャリストの先頭から 0、1、2...と付けられます。

- **グローバルインデックス番号**

ソース内で使用するテクスチャを一意に番号を振ります。
グローバルインデックス番号が同じテクスチャは、同一のテクスチャとみなします。

- **カレントテクスチャリスト**

テクスチャ関連関数の操作対象のテクスチャリストをカレントテクスチャリストと呼びます。

- **カレントテクスチャ**

カレントテクスチャリストの中の操作対象テクスチャをカレントテクスチャと呼びます。
テクスチャ関連関数の多くが、カレントテクスチャに対してテクスチャ操作をします。

- **PVR 形式**

Ninja でのファイルからロードできるテクスチャファイル形式です。

- **アスペクト比**

テクスチャの縦横比をアスペクト比と呼びます。

- **U、V 座標**

テクスチャの内部座標の横方向をU座標、縦方向をV座標と定義します。U、V座標はアスペクト比によらず、0 から 1 を設定することでテクスチャ全面を表示することができます。

- **ミップマップ**

同じテクスチャを表すテクスチャマップの順位付けされたテクスチャのセットをミップマップと呼びます。

- **LOD (level of detail) 詳細レベル**

ミップマップのレベルを LOD 詳細レベルと呼びます。

- **テクスチャメモリ**

テクスチャを保存しておくためのメモリです。

- **カテゴリコード**

Ninja で使用できるテクスチャの形式をカテゴリコードと呼びます。Ninja で使用できるテクスチャの形式は、TWIDDLED 形式、TWIDDLED ミップマップ形式、VQ 形式、VQ ミップマップ形式、4 ビットパレット形式、4 ビットパレットミップマップ形式、8 ビットパレット形式、8 ビットパレットミップマップ形式、レクタングル形式、ストライド形式があります。詳しくは「4.14 PVR フォーマット」の項で説明します。

- **ストライド値**

Ninja で STRIDE 形式のテクスチャを使用する場合に指定します。使用できる値は 32 の倍数で 32 から 992 です。

4.2 Ninja2 の変更点

Ninja1 から Ninja2 のテクスチャ関連での大きな変更点は、次のとおりです。

(1) テクスチャ管理の方法が変更された。

Ninja では NJS_TEXMEMLIST 構造体でテクスチャを管理していましたが、パレットテクスチャでバンクの違うテクスチャを使用する場合、管理に無駄な部分が多くありました。これを修正するため、NJS_TEXMANAGE 構造体と NJS_TEXSYSTEM 構造体を新たに作り、テクスチャ管理をします。この2つ構造体をあわせてテクスチャ管理構造体と呼ぶことにします。njInitTextureEx 関数で指定するテクスチャ管理構造体の領域をテクスチャ管理領域と呼びます。

(2) GD アクセスするテクスチャロード関数を Ninja2 ライブラリから Njutil ライブラリへ変更しソースを公開した。

Ninja において GD へのアクセスを行っていたテクスチャロード関数は Ninja2 から削除し、Njutil ライブラリへ変更しました。GD アクセス部分を切り離すことにより、テクスチャロードのカスタマイズ、モデルデータ、モーションデータなどテクスチャ以外の他のデータと同時にテクスチャをロードする場合や圧縮されたテクスチャデータを展開しながらロードするなど、ユーザーがテクスチャロード関数を比較的簡単に作成、修正できるようになりました。

(3) ドライバ (kamui2) の変更により、システムシーケンスが変更され 2V レイテンシモードのとき頂点直接転送中にテクスチャロードすると DMA 転送できなくなり、必ず CPU 転送になる。

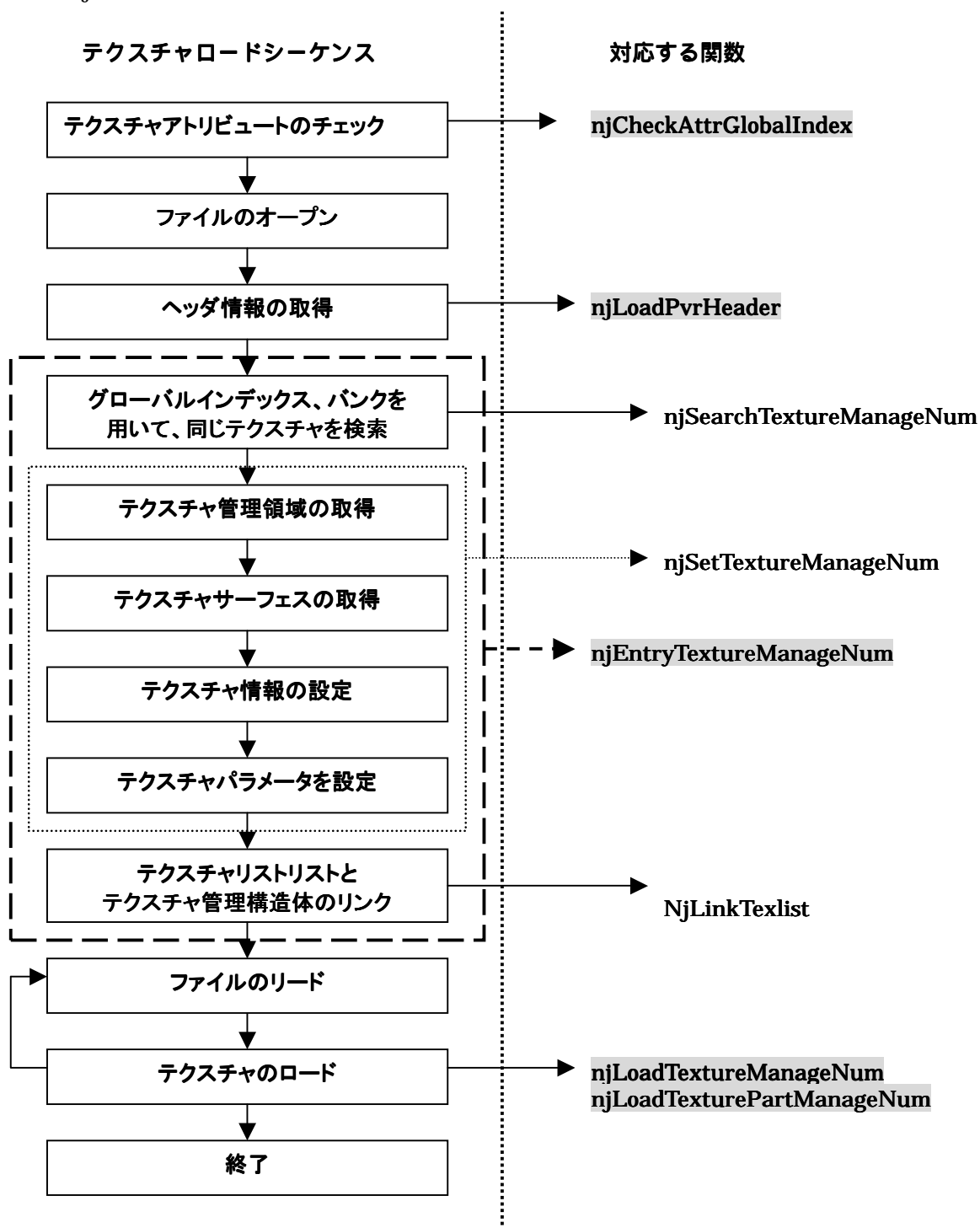
Ninja から Ninja2 の変更で、使用しているドライバも Kamui から Kamui2 へ変更されました。Kamui から Kamui2 の変更により、頂点登録、頂点の DMA 転送、レンダリングといった描画に関するシーケンスが変更されました。これにより、2V レイテンシモードのとき、njStartDraw 関数から njRender 関数の間にテクスチャロードを行うと必ず CPU 転送になります。メインループ中でテクスチャロードを行う場合、njRender 関数終了から njStartDraw 関数の間に行うようにして下さい。3V レイテンシモードはいつテクスチャロードしても、DMA 転送することができます。ただし、DMA 転送ができるのは、DMA 転送の条件を満たすときだけです。

(4) キャッシュテクスチャは廃止された。

キャッシュテクスチャを廃止し、テクスチャの管理を簡略化しました。メモリテクスチャをユーザー管理して下さい。

4.3 テクスチャロードシーケンス

Ninja2 での基本的なテクスチャロードシーケンスは次のようになります。

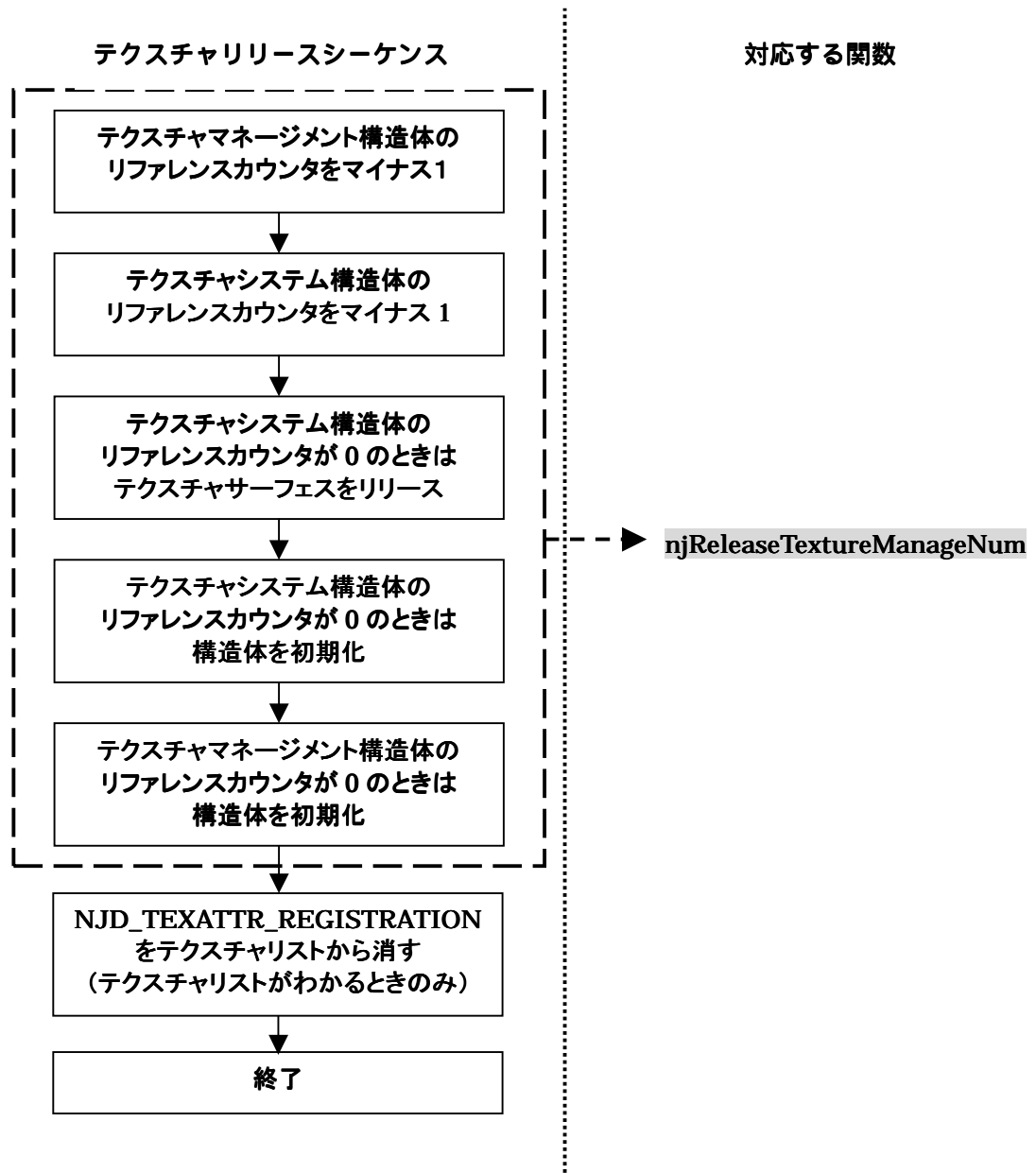


通常のテクスチャロードは、`NjLinkTexlist` の関数で作成することができます。

`NjLinkTexlist` 関数内部で、`NJD_TEXATTR_REGISTRATION` フラグを設定します。

4.4 テクスチャリリースシーケンス

Ninja2 での基本的なテクスチャリリースシーケンスは、次のようになります。



4.5 テクスチャ管理

Ninja では NJS_TEXMEMLIST 構造体を使用してテクスチャを管理していましたが、Ninja2 では NJS_TEXMANAGE 構造体と NJS_TEXSYSTEM 構造体を使用し、テクスチャを管理します。

4.5.1 NJS_TEXMANAGE 構造体

```
typedef struct{
    Uint32      tspparam;
    Uint32      texparam;
    Uint32      bank;
    NJS_TEXSYSTEM *texsys;
    Int         count;
    Uint32      texflag;
}NJS_TEXMANAGE;
```

(1) tspparam、texparam

頂点登録に使用するパラメータです。この2つのパラメータはテクスチャサーフェスを取得した後 njSetTextureParamEx 関数を実行することで設定されます。パレットテクスチャの場合、バンク番号を入れてから njSetTextureParamEx 関数を実行すると、バンク番号も反映します。njSetPaletteBank * 関数を実行することでも、バンク番号の設定を反映します。ただし、NJS_TEXSYSTEM 構造体の texsurface.Type の上位 16 ビットにカテゴリーコードが入っている必要があります。

NjSetTextureManageNum、または njEntryTextureManageNum 関数を使用している場合は、内部で実行しています。初期値は 0 を設定します。

(2) bank

バンク番号。パレットテクスチャの場合、バンク番号を設定します。4bpp パレットでは 0～63、8bpp パレットでは 0、16、32、48 を設定します。初期値・パレットテクスチャ以外のカテゴリーコードのテクスチャを使用している場合は、0xFFFFFFFF を設定します。

(3) texsys

テクスチャシステム構造体へのポインタです。初期値は NULL を設定します。テクスチャマネージメント領域が使用中かの判断は、ここが NULL かで判定します。

(4) count

テクスチャのリファレンスカウンタです。パレットテクスチャ以外のテクスチャの場合は、リンクを張っているテクスチャシステム構造体の count と同じ値を設定します。パレットテクスチャの場合、グローバルインデックスとバンクが同じテクスチャのみのリファレンスカウントを設定します。初期値は 0 を設定します。

(5) texflag

現在、未使用です。初期値は 0 を設定します。

4.5.2 NJS_TEXSYSTEM 構造体

```
typedef struct{
    Uint32      globalIndex;
    NJS_TEXSURFACE texsurface;
    Int         count;
}NJS_TEXSYSTEM;
```

(1) globalIndex

グローバルインデックス番号です。ユーザーが使用できる範囲は 0 ~ 0xFFFFFFFFEF の間です。初期値は、0xFFFFFFFF を設定します。テクスチャシステム領域が使用中かの判断は、この領域は 0xFFFFFFFF かで判定します。

(2) texsurface

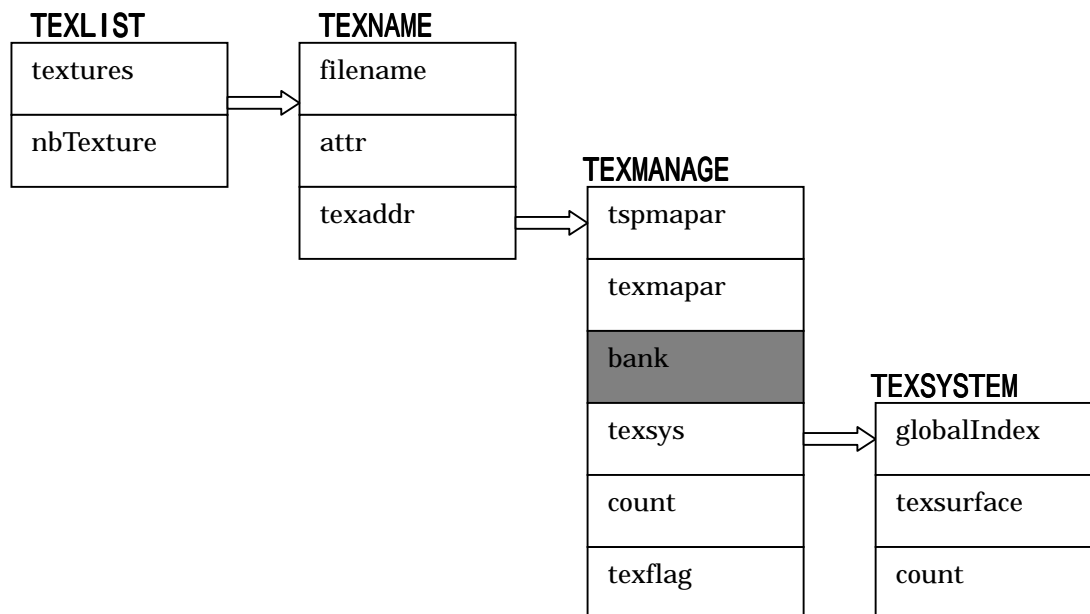
テクスチャサーフェス情報です。テクスチャの各情報が設定されます。初期値は全メンバ 0 を設定する。

(3) count

リファレンスカウンタです。パレットテクスチャでグローバルインデックス番号が同じでバンクが違うテクスチャがある場合には、合計のリファレンスカウントを設定します。パレットテクスチャ以外のテクスチャの場合、テクスチャシステム構造体へリンクを張っているテクスチャマネージメント構造体の count と同じ値を設定します。リリース関数ではカウントが 0 になる場合のみ、テクスチャサーフェスはリリースしません。初期値は 0 を設定します。

4.5.3 通常のテクスチャ管理

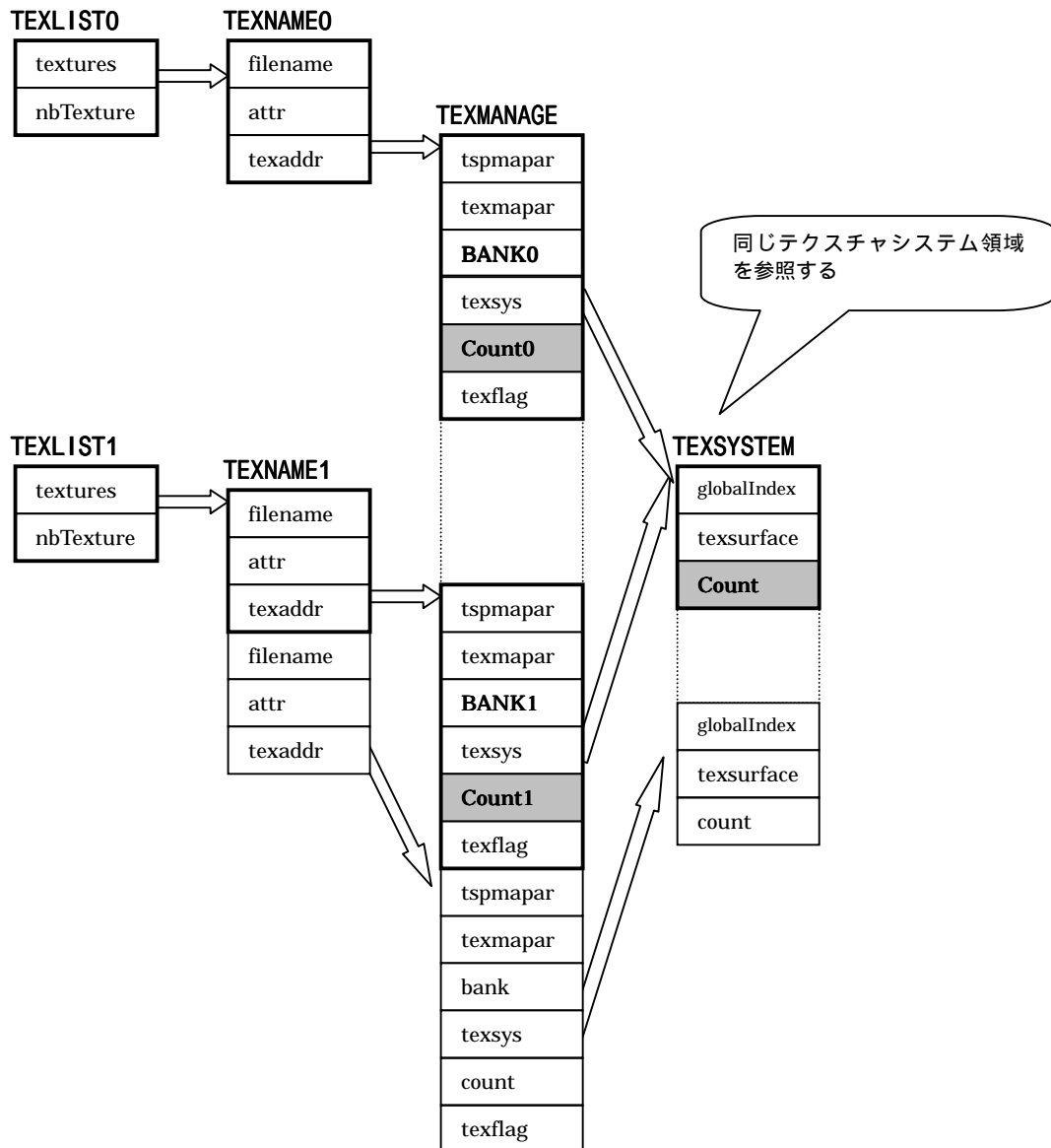
パレットテクスチャ以外のテクスチャは、以下のように入っています。



パレットテクスチャ以外のテクスチャの場合、NJS_TEXMANAGE 構造体の count と NJS_TEXSYSTEM 構造体の count には同じ値が入ります。bank には、値は設定されません。

4.5.4 パレットテクスチャのテクスチャ管理

パレットテクスチャは以下のように入っています。
 (太い黒枠が同じパレットテクスチャでバンク番号の違うものとする)



グローバルインデックス番号が同じでバンク番号の違うテクスチャは、同じテクスチャシステム領域を参照します。NJS_TEXSYSTEM 構造体のリファレンスカウンタは NJS_TEXMANAGE 構造体から参照しているリファレンスカウンタの合計が入ります。

$$\text{TEXTSYSTEM}(\text{Count}) = \text{TEXMANAGE}(\text{Count0}) + \text{TEXMANAGE}(\text{Count1}) + \dots$$

4.5.5 テクスチャサーフェスとテクスチャ管理領域の開放

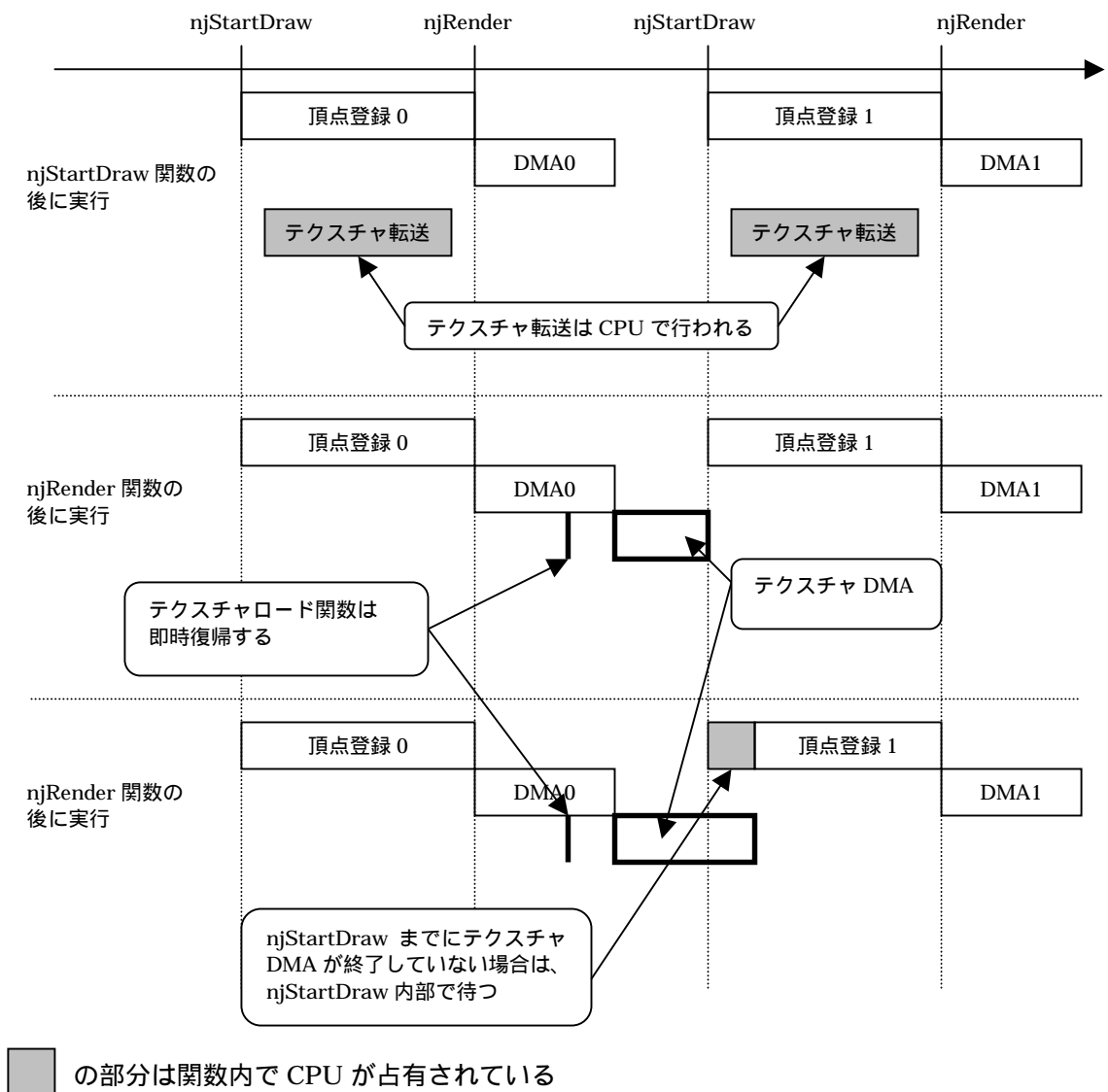
テクスチャリリースのとき NJS_TEXMANAGE 構造体、NJS_TEXSYSTEM 構造体のリファレンスカウンタをそれぞれ 1 減らし、NJS_TEXMANAGE 構造体のリファレンスカウンタが 0 のときは、NJS_TEXMANAGE 構造体を初期化します。NJS_TEXSYSTEM 構造体が 0 の時はテクスチャサーフェスをリリースして、NJS_TEXSYSTEM 構造体を初期化します。

4.6 テクスチャの DMA 転送

4.6.1 ドライバの変更によるテクスチャの DMA 転送の変更

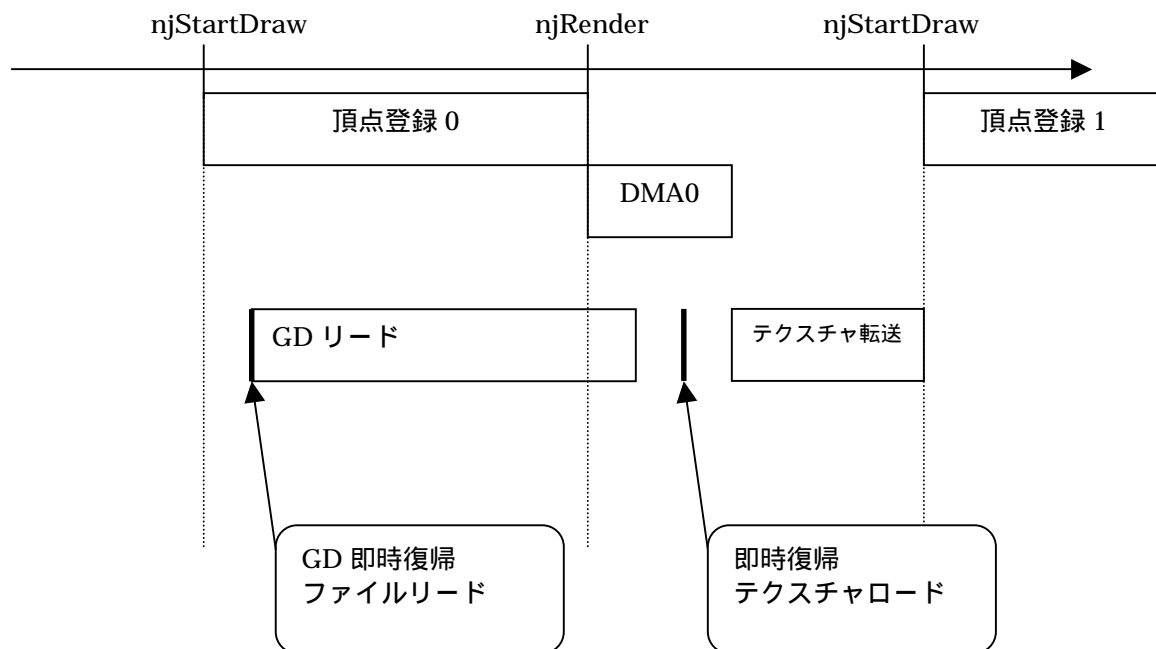
Ninja では、メインループ中にテクスチャロードをしたとき、転送元(メインメモリ)と転送先(テクスチャメモリ)のアラインメントが 32 バイトアラインメントの場合には、32 バイト単位で DMA 転送することができました。Ninja2 ではドライバ(Kamui)の変更により、2V レイテンシモード使用のとき njStartDraw 関数から njRender 関数の間にメインメモリからテクスチャメモリへの転送を行った場合、必ず CPU 転送になります(即時復帰することができません)。テクスチャを DMA 転送を行うためには njRender 関数の後、njStartDraw 関数までにテクスチャのロード関数を実行して下さい。ただし、この間に実行されたテクスチャロード関数は即時復帰しますが、njStartDraw 関数で頂点登録するときにテクスチャロードの DMA 転送が終了していないと njStartDraw 関数内部で DMA 終了を待ちます。3V レイテンシモードでは DMA 転送の条件の時は、いつの場合も DMA 転送になります。

- 2V レイテンシモード



4.6.2 効率の良いテクスチャロードをするために

2 V レイテンシモードでメインループ中に GD から効率の良いテクスチャのロードをするには、njStartDraw 関数から njRender 関数の中で GD からメインメモリへテクスチャをロードして、メインメモリからテクスチャメモリの転送を njRender 実行後に行うようにすると、CPU の待ち時間がなくなります。



4.7 テクスチャフォーマット

".pvr"フォーマットを使用します。コンバータは pvrconv です。モデルコンバータに組み込まれ自動的にモデルに使われているテクスチャが全部変換されます。

コンバータはもとの画像の 値をチェックし、自動的に次の三つのフォーマットを切り替えて出力します。

| | |
|----------------|---------------|
| がない場合 | RGB565 で出力。 |
| がある場合 | ARGB4444 で出力。 |
| が 0、255 の二値の場合 | ARGB1555 で出力。 |

またテクスチャが正方形の場合 Twiddled 形式が、長方形の場合 Rectangle 形式がコンバータで自動選択されます。

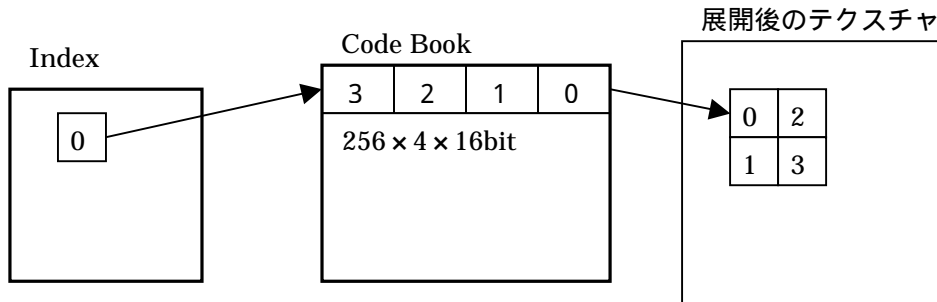
- **Twiddled 形式**
テクスチャのピクセルを、高速にメモリから読み出せる順番に並べ替えたテクスチャです。ミップマップを利用できます。また表示が高速です。
- **Rectangle 形式**
ピクセルの順番をイメージそのままとしているテクスチャです。表示が Twiddled に比べ低速です。ミップマップが使用できないので注意して下さい。
- **バンプマッピング**
バンプマッピングテクスチャはグレー階調で用意します。RGB カラー画像のバンプは扱えません。コンバータでハードウェアが期待するデータに変換します。

4.8 VQ

VQ とは、ベクトル量子化によりテクスチャを圧縮します。

4.8.1 VQ テクスチャ構造

VQ テクスチャは「Code Book」と「Index」の2種類のデータから構成されています。Code Book には4テクセル分のカラーデータが1つのCode Bookとして入っています。VQ テクスチャの場合、Code Book は256個に固定されています。Index データはCode Book を示す番号が4テクセルで1つ入っています。Index データはTwiddled形式に並んでいます。



4.8.2 圧縮前後のデータサイズ

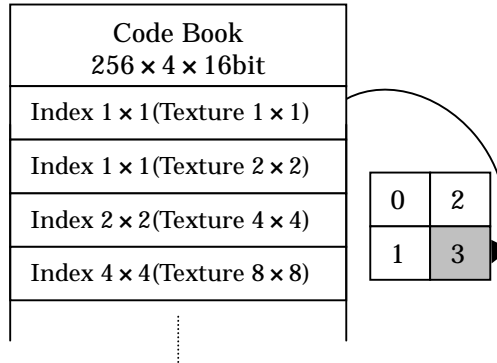
次の表は、ミップマップなしの場合について、圧縮前のテクスチャサイズと圧縮後のテクスチャサイズを比べたものです。8x8 から 32x32 は Code Book サイズの大きさから VQ フォーマットの方が逆に大きくなってしまいます。16x16、32x32、64x64 は、Small VQ テクスチャとして扱います(8x8 は Small VQ にもありません)。

- ミップマップなしの場合

| テクスチャ サイズ | 圧縮前の データサイズ (byte) | 圧縮後の データサイズ (byte) | 圧縮率 (%) | Code Book サイズ (byte) | Index サイズ (byte) |
|--------------|--------------------------|--------------------------|------------|----------------------------|------------------------|
| 8×8 | 128 | 2,064 | 1612.50 | 256×2×4 | 16 |
| 16×16 | 512 | 2,112 | 412.50 | 256×2×4 | 64 |
| 32×32 | 2,048 | 2,304 | 112.50 | 256×2×4 | 256 |
| 64×64 | 8,192 | 3,072 | 37.50 | 256×2×4 | 1,024 |
| 128×128 | 32,768 | 6,144 | 18.75 | 256×2×4 | 4,096 |
| 256×256 | 131,072 | 18,432 | 14.06 | 256×2×4 | 16,384 |
| 512×512 | 524,288 | 67,584 | 12.89 | 256×2×4 | 65,536 |
| 1024×1024 | 2,097,152 | 264,192 | 12.60 | 256×2×4 | 262,144 |

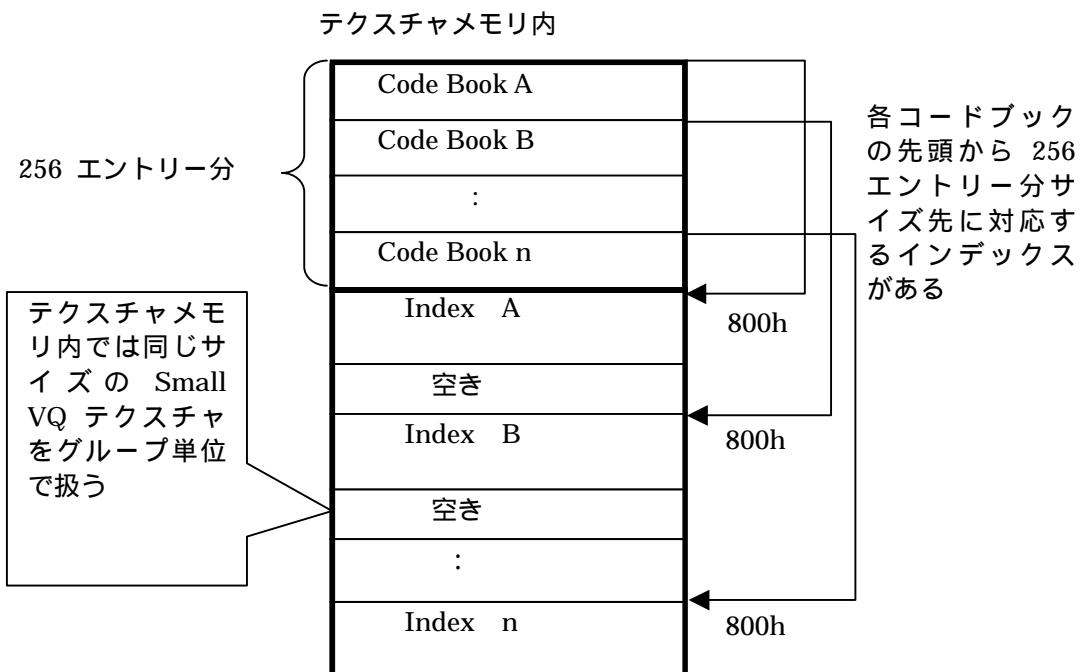
4.8.3 VQ Mipmap

VQ ミップマップテクスチャの場合にも、Code Book データは 256 個固定です。Index データは 1×1 から 2×2 、 4×4 、... と順に並んでいます。また 1×1 の場合は、Code Book の Texel-3 (右下) が使用されます。



4.8.4 Small VQ テクスチャ構造


小さいサイズの VQ テクスチャの圧縮率を上げるために、Code Book サイズを調整したテクスチャフォーマットが Small VQ テクスチャフォーマットです。ハードウェア的には VQ テクスチャと全く変わりません。ハードウェア上の VQ テクスチャの仕様は Code Book の先頭から 256 個のコードブックサイズ (800h) の後、インデックスデータが入ることになっています。この仕様を元に、最も効率の良い Code Book のサイズを選択しています。また、Small VQ のテクスチャは同じサイズのテクスチャをグループ化してテクスチャメモリに保存しています。1つのテクスチャをロードする場合でも、1つのグループ分のテクスチャメモリを取ります。同一のサイズのテクスチャがグループの個数ロードされると効率が上がります。



4.8.5 Small VQ の各サイズとグループ数

Small VQ フォーマットは下の表のように、最もデータ効率の良い Code Book のエントリ数になっています。また、Code Book のサイズから各サイズのグループ数を求めています。Small VQ の場合、1つの Small VQ をロードすると同じ種類の Small VQ をグループ数分の領域を確保します。Small VQ テクスチャを使用する場合は、グループ数単位で同一種類の Small VQ テクスチャを使用すると最も効率的になります。同一グループのテクスチャが1つの場合などは、圧縮しないテクスチャの方がデータは小さくなります。

| Texture Size | Mip Map | Code Book Size | | Index Size (Bytes) | 空き領域のサイズ (Bytes) | グループ数 | グループ全サイズ (Bytes) |
|--------------|---------|----------------|------------|--------------------|------------------|-------|------------------|
| | | Entry | Bytes | | | | |
| 8x8 | No | 16 | 128(80h) | 16(10h) | 112(70h) | 16 | 3984(F90h) |
| | Yes | 16 | 128(80h) | 32(20h) | 96(60h) | 16 | 4000(FA0h) |
| 16x16 | No | 16 | 128(80h) | 64(40h) | 64(40h) | 16 | 4032(FC0h) |
| | Yes | 16 | 128(80h) | 96(60h) | 32(20h) | 16 | 4064(FE0h) |
| 32x32 | No | 32 | 256(100h) | 256(100h) | 0 | 8 | 4096(1000h) |
| | Yes | 64 | 512(200h) | 352(160h) | 160(A0h) | 4 | 3936(F60h) |
| 64x64 | No | 128 | 1024(400h) | 1024(400h) | 0 | 2 | 4096(1000h) |
| | Yes | 256 | 2048(800h) | 1376(560h) | 0 | 1 | 3424(D60h) |

 この部分は、Small VQ フォーマットがサポートされていません。

- ミップマップなしの場合

| テクスチャサイズ | 圧縮前のデータサイズ(byte) | 圧縮後のデータサイズ(byte) | 圧縮率(%) | Code Book サイズ(byte) | Index サイズ(byte) |
|----------|------------------|------------------|--------|---------------------|-----------------|
| 16x16 | 512 | 192 | 37.50 | 16x2x4 | 64 |
| 32x32 | 2,048 | 512 | 25.00 | 32x2x4 | 256 |
| 64x64 | 8,192 | 2,048 | 25.00 | 128x2x4 | 1,024 |

4.9 パレット

4.9.1 カラーモード

Dreamcast では、16 ビットカラーと 32 ビットカラーのパレットが使用できます。使用できるカラーモードは次のとおりです。

| 16 ビットカラー | 32 ビットカラー |
|----------------------|----------------------|
| NJD_TEXFMT_ARGB_1555 | NJD_TEXFMT_ARGB_8888 |
| NJD_TEXFMT_RGB_565 | |
| NJD_TEXFMT_ARGB_4444 | |

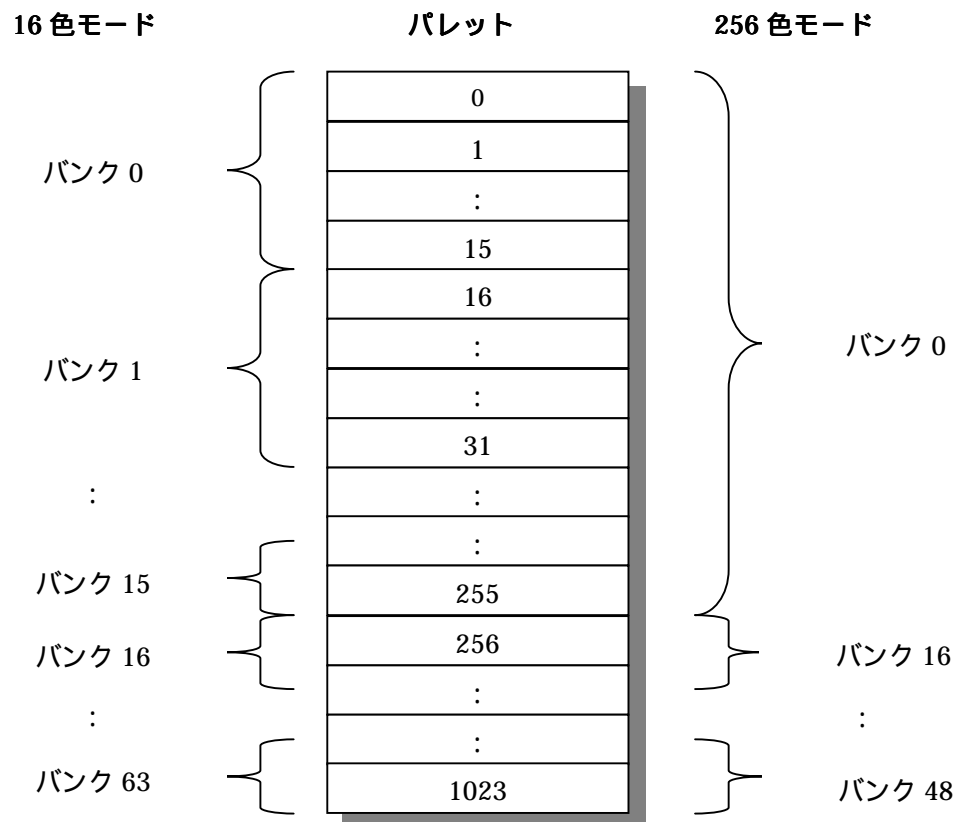
注 意 システム中で 1 つのカラーモードしか使用できません。カラーモードの切り替えは、レンダリング終了後行うことができます。また ARGB8888 カラーは、ポイントサンプル以外のフィルタモードを使用すると性能が 2 分の 1 になるので注意が必要です。

4.9.2 パレットテクスチャの色数

パレットテクスチャの色数には、16 色パレットの Palettized 4bpp テクスチャと 256 色パレットの Palettized 8bpp テクスチャがあります。

4.9.3 バンク

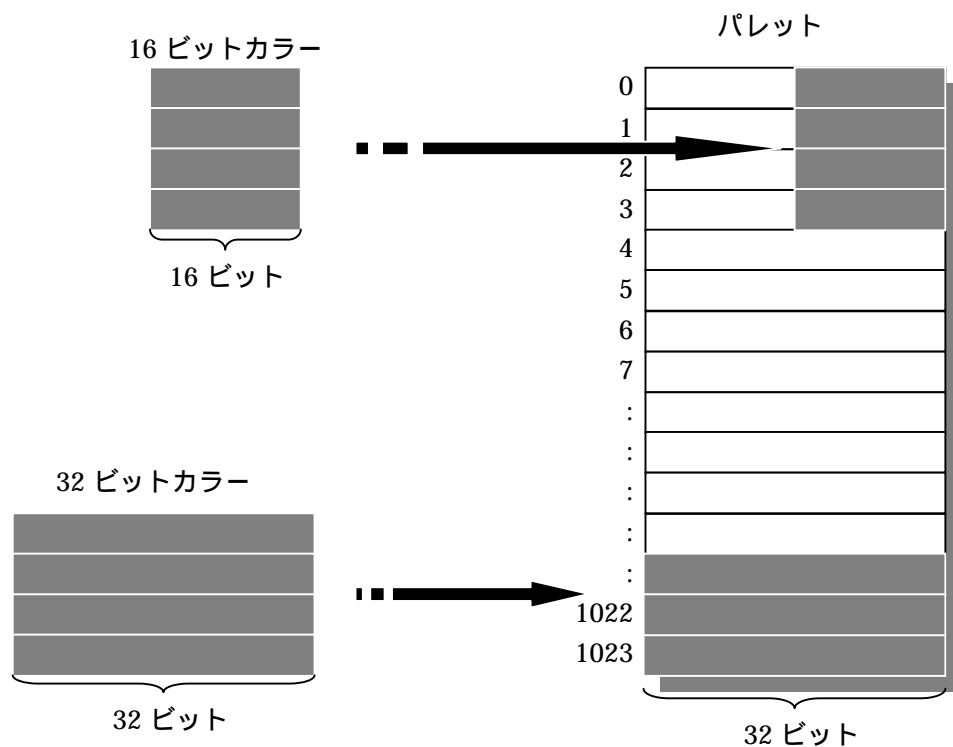
パレットは 16 ビットカラー、32 ビットカラーに関係なく 1024 色分あります。16 色を 1 つの単位としたものをバンクといいます。16 色モードでは 64 バンク分、256 色モードでは 4 バンク分あります。16 色モードのとき設定できるバンク番号は 0 から 63 で、256 色モードのときは上位 2 ビットしか見ないので、0 (0 - 15)、16 (16 - 31)、32 (32 - 47)、48 (48 - 63) です。



4.9.4 カラーデータ配置

パレットで利用できる色数は、16 ビットカラー、32 ビットカラーの場合とも 1024 色あります。カラーデータは以下のようなイメージで入っています。

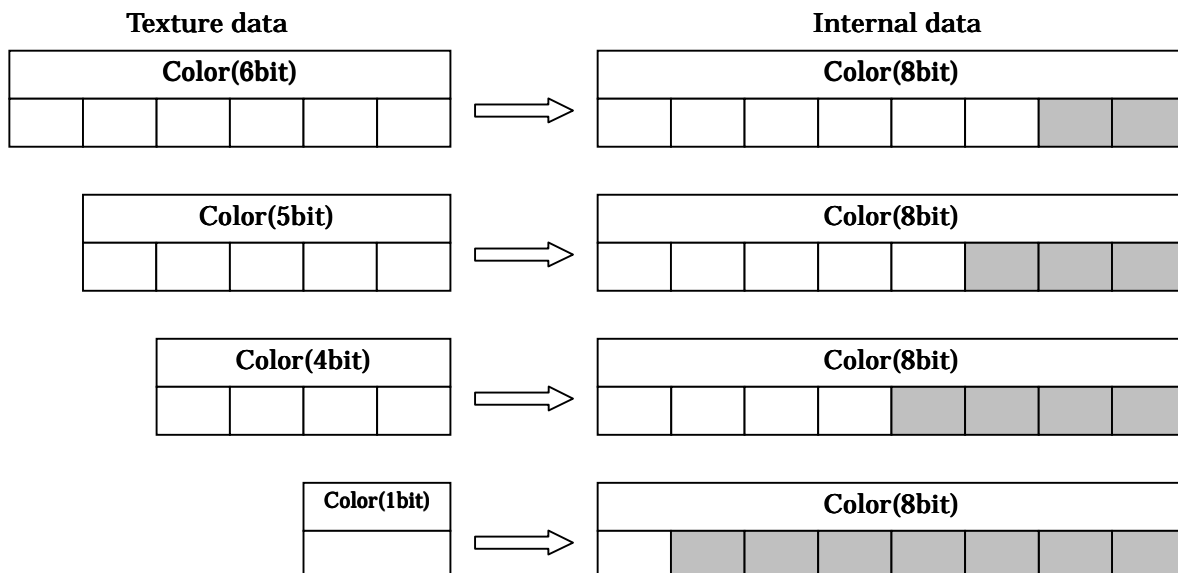
注 意 カラーデータでは、16 ビットカラーと 32 ビットカラーが同時に入っていますが、同時に使用することはできません。



4.10 カラーデータ拡張方法

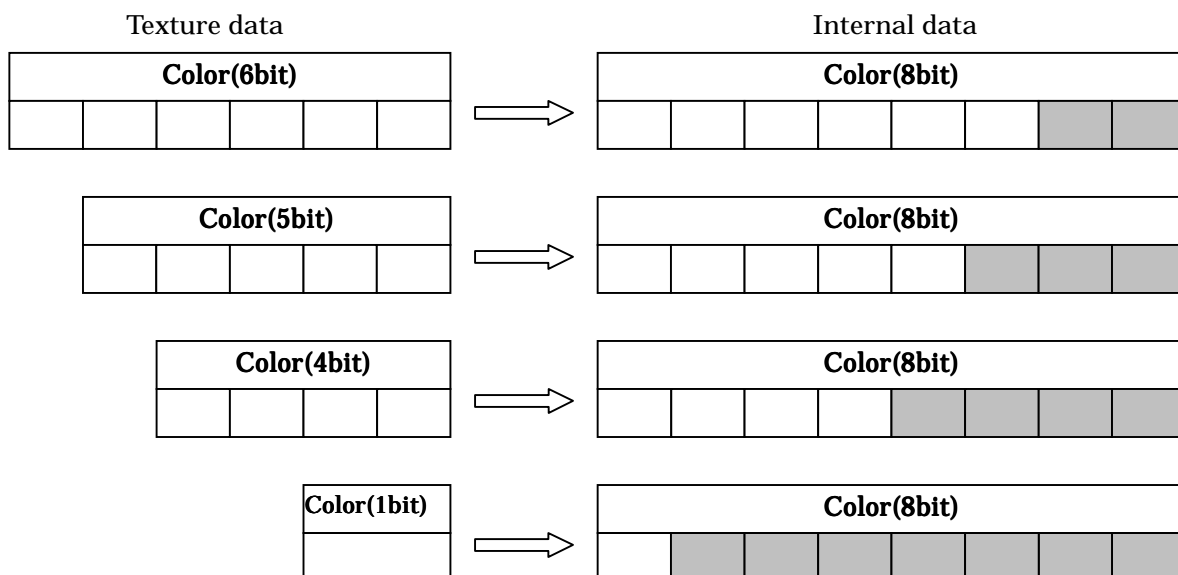
読み込まれたテクスチャデータは、ハードウェア内部で ARGB それぞれ 8 ビット値として扱われます。

(1) Twiddled 形式の場合



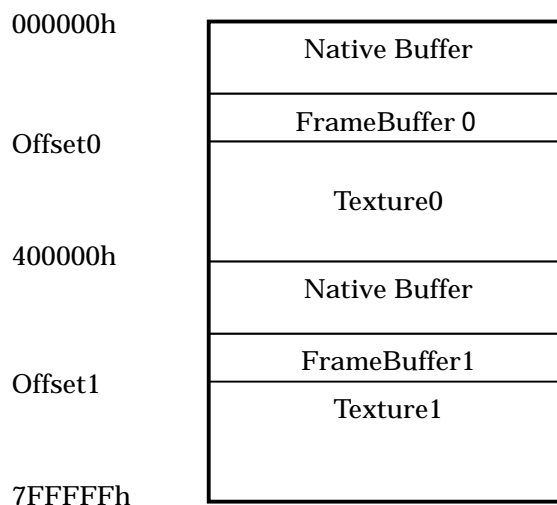
(2) Non-Twiddled 形式の場合 (Rectangle、Stride)

Non-Twiddled 形式のテクスチャの場合、それぞれの値の下部に 0 が付けられます。ただし、カラーが 1 ビットの場合は、Twiddled 形式と同じです。



4.11 VRAM への登録

VRAM は、以下のようなイメージで配置されています（32bit Address 表示）。



テクスチャメモリは、次のように取得されます。

Texture0 = (TextureSize / 2)

Texture1 = (TextureSize / 2)

● 例：

VGA (640x480)、テクスチャメモリサイズ 380000h バイトを指定した場合

Texture0 = (380000h / 2) = 1C0000h(byte)

Texture1 = (380000h / 2) = 1C0000h(byte)

Offset0 = 400000h - 1C0000h = 240000h

Offset1 = 800000h - 1C0000h = 640000h

4.11.1 テクスチャの登録

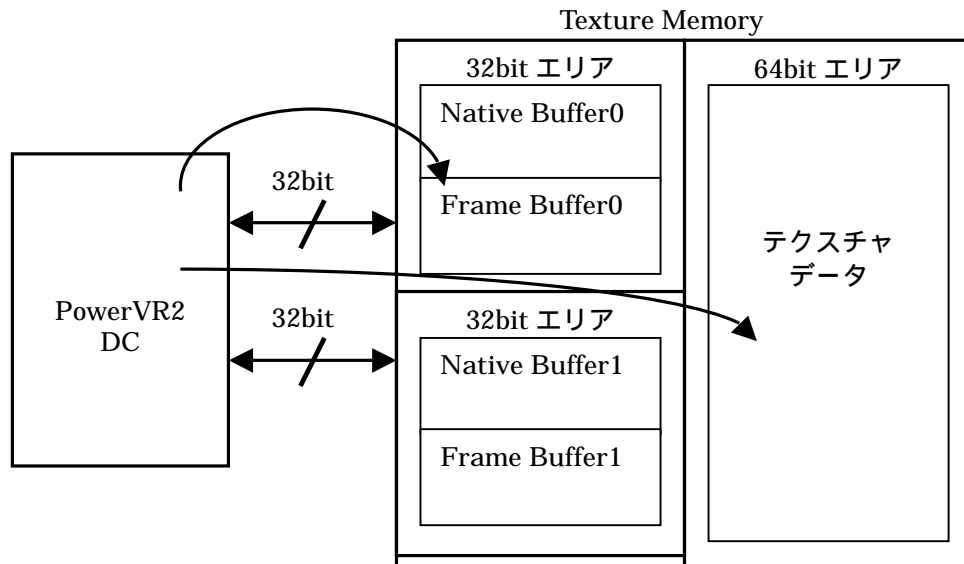
上の図でテクスチャ領域を 64bit アドレスに直すと、4800000h から 7FFFFFFFh になります（8 M バイトの後ろから 3.5M バイト取得している）。

Small VQ 以外のテクスチャは空き領域を順に確保しテクスチャをロードしていきます。Small VQ テクスチャの登録は、各 Small VQ の種類毎にグループ単位で登録します。

4.12 レンダーテクスチャ

4.12.1 フレームバッファとテクスチャデータ

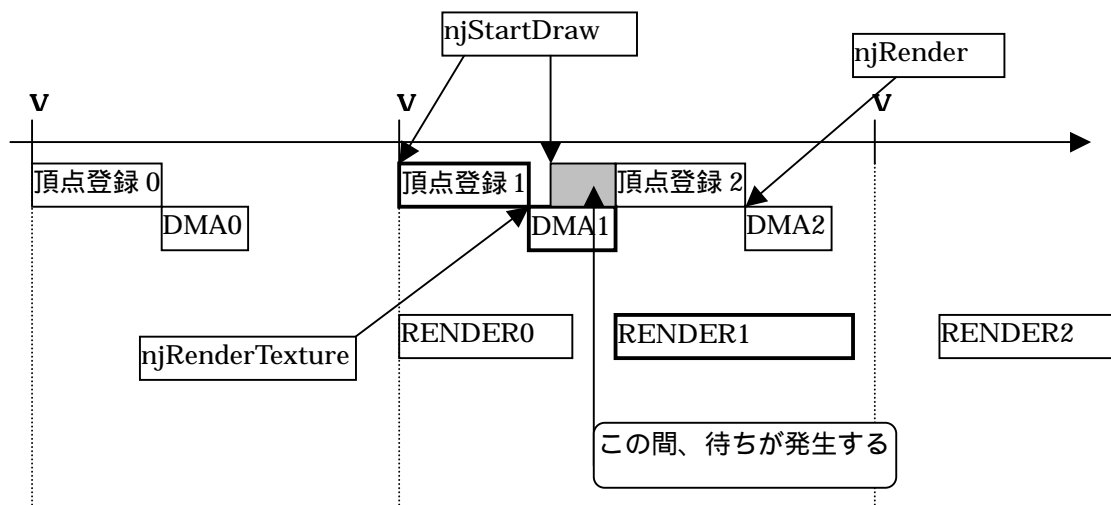
フレームバッファは32bit アドレスエリアでアクセスし、テクスチャは64bit アドレスエリアでアクセスしているため、フレームバッファをテクスチャとして利用できません。アドレスモードは変更することはできません。(Ninja2 のフレームバッファテクスチャはフレームバッファを直接テクスチャとして使用していません)



4.12.2 レンダーテクスチャシーケンス

レンダーテクスチャ分の njStartDraw 関数を行うとき、前回の頂点登録の DMA 転送が終了していない場合は、njStartDraw 関数内部で待ちが発生します。またレンダーテクスチャ関数実行後、レンダーテクスチャ分の頂点登録 DMA 転送が終了する前に次の njStartDraw 関数を実行した場合も、njStartDraw 関数内部で待ちが発生します。

- 2V レイテンシモードの場合

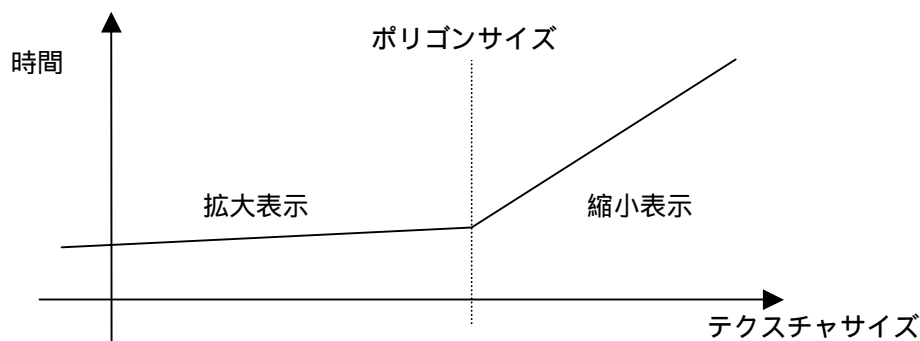


4.13 性能比

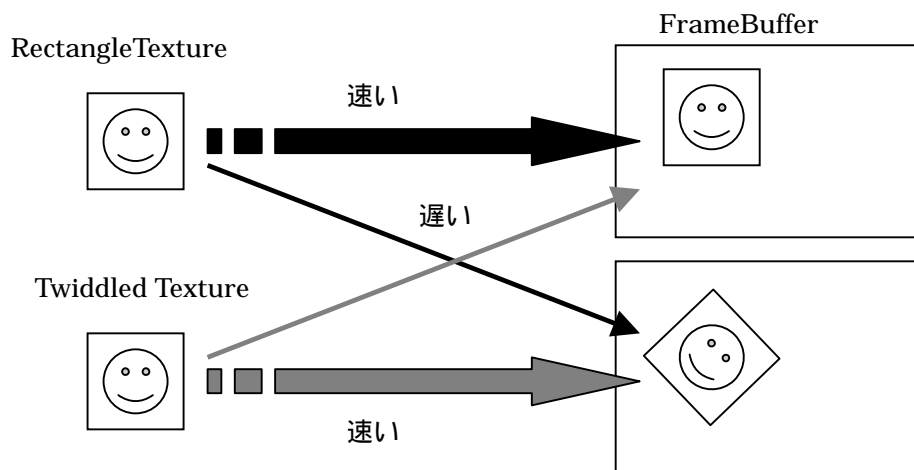
Twiddled、Twiddled Mipmap、Rectangle、VQ、VQ Mipmap、Palettized 4bpp、Palettized 4bpp Mipmap、Palettized 8bpp、Palettized 8bpp Mipmap について性能比較実験を行いました。

以下は、まとめです。

- (1) VQ、パレット（16 ビットカラーのとき）は他のテクスチャタイプより常に高速にレンダリングできる。テクスチャが縮小される場合には、Mipmap 付きはより高速に表示できる。
- (2) 同じサイズのポリゴン描画でもテクスチャ面積がポリゴン面積より大きくなる場合（テクスチャを縮小して表示する場合）、レンダリング時間が大幅に掛かる。
- (3) テクスチャの面積が描画するポリゴン面積より小さい場合（テクスチャを拡大して表示する場合）、テクスチャのサイズにレンダリング時間はほとんど影響を受けない。



- (4) ポイントサンプル以外のフィルタモードでポリゴンを描画する場合、32 ビットカラーのパレット 4bpp, 8bpp 共に、16 ビットカラーの 2 倍のレンダリング時間が掛かる。
- (5) ポイントサンプルでレンダリング方向とテクスチャリード方向が同じ場合には、Twiddled より Rectangle の方が最大約 1.5 倍レンダリング時間はかからない。Twiddled Mipmap と同等です（スプライト描画などは Rectangle の方が速い）。



4.14 PVR フォーマット

4.14.1 PVR ヘッド情報

| | | |
|------------------------|--------------|-------|
| グローバル インデックス ヘッダ | “GBIX”チャンク | 4 バイト |
| | ネクストタグへのバイト数 | 4 バイト |
| | グローバルインデックス | 4 バイト |
| | ダミー | 4 バイト |

| | | |
|----------------------|--------------|-------|
| PVR フォーマット ヘッダ | “PVRT”チャンク | 4 バイト |
| | ネクストタグへのバイト数 | 4 バイト |
| | テクスチャアトリビュート | 4 バイト |
| | 横サイズ | 2 バイト |
| | 縦サイズ | 2 バイト |

| | |
|------------|--|
| テクスチャデータ本体 | |
|------------|--|

(1) グローバルインデックスヘッダ

- ネクストタグへのバイト数
PVRT までのバイト数 8 が入ります。
- グローバルインデックス
パレットテクスチャ以外のフォーマットの場合、0 から 0xFFFFFFFFEF まで設定可能です。
0xFFFFFFFFF0 から 0xFFFFFFFFFF まではテクスチャのシステムが使用します。パレットテクスチャの場合、上位 6 ビットがバンク番号、下位 26 ビットがグローバルインデックスになります。
- ダミー
アラインメントを調整するためにダミーデータが入っています。ツールでテクスチャを出力した場合、0x20 が入っています。この領域は使用していませんがユーザ使用禁止です。

(2) PVR フォーマットヘッダ

- ネクストタグへのバイト数
テクスチャデータ本体のバイト数 + 8 バイトの数が入ります。
- テクスチャアトリビュート
テクスチャアトリビュートにはカテゴリーコードとカラーフォーマットの OR した値が入ります。
使用できるカテゴリーコード、カラーフォーマットは次のとおりです。

- カテゴリーコード

| | |
|---------------------------------------|----------|
| #define NJD_TEXFMT_TWIDDLED | (0x0100) |
| #define NJD_TEXFMT_TWIDDLED_MM | (0x0200) |
| #define NJD_TEXFMT_VQ | (0x0300) |
| #define NJD_TEXFMT_VQ_MM | (0x0400) |
| #define NJD_TEXFMT_PALETTIZE4 | (0x0500) |
| #define NJD_TEXFMT_PALETTIZE4_MM | (0x0600) |
| #define NJD_TEXFMT_PALETTIZE8 | (0x0700) |
| #define NJD_TEXFMT_PALETTIZE8_MM | (0x0800) |
| #define NJD_TEXFMT_RECTANGLE | (0x0900) |
| #define NJD_TEXFMT_STRIDE | (0x0B00) |
| #define NJD_TEXFMT_TWIDDLED_RECTANGLE | (0x0D00) |
| #define NJD_TEXFMT_SMALLVQ | (0x1000) |
| #define NJD_TEXFMT_SMALLVQ_MM | (0x1100) |
| #define NJD_TEXFMT_TWIDDLED_MM_DMA | (0x1200) |

- カラーフォーマット

| | |
|-----------------------------|----------|
| #define NJD_TEXFMT_ARGB1555 | (0x0000) |
| #define NJD_TEXFMT_RGB565 | (0x0001) |
| #define NJD_TEXFMT_ARGB4444 | (0x0002) |
| #define NJD_TEXFMT_ARGB8888 | (0x0006) |

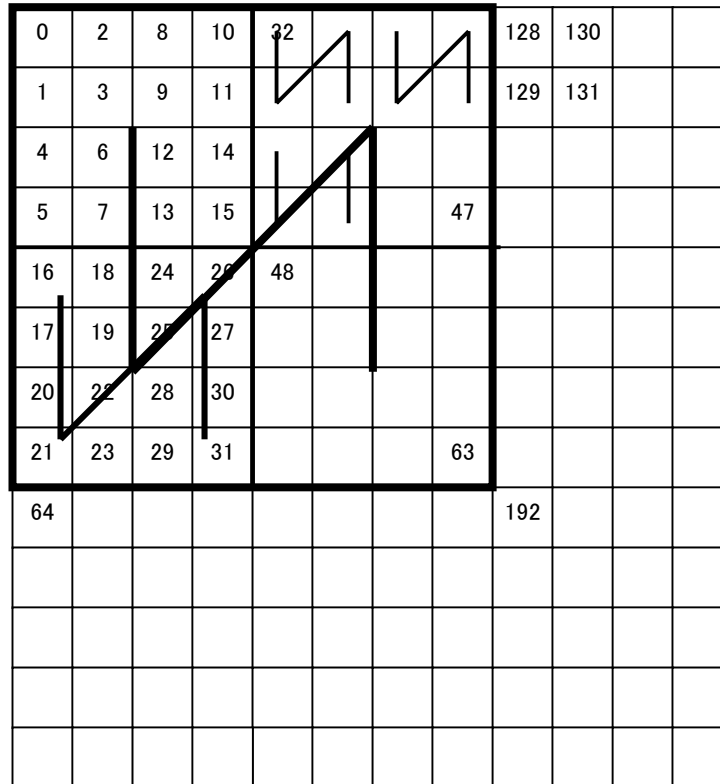
- 横サイズ、縦サイズ

テクスチャの横、縦のサイズがそれぞれ入ります。指定できるテクスチャのサイズは 8、16、32、64、128、256、512、1024 です。Rectangle、Stride、Twiddled Rectangle 以外のテクスチャは正方形のテクスチャしかないので、横と縦のサイズには同じものが入ります。

4.14.2 カテゴリーコード

(1) Twiddled、Twiddled Mipmap、Twiddled Mipmap Dma フォーマット

Twiddled フォーマットは、Ninja の最も基本的なテクスチャフォーマットです。このフォーマットは、各フィルタやレンダリングを高速に行うために、テクスチャ内部を最適化し配置しています。このため、テクスチャ内部はラスタオーダーでは並んでいません。また、Twiddled フォーマットを作成するためには、元データが正方形でなくてはなりません。



● Twiddled フォーマットのデータ配置

Twiddled フォーマットは、以下のようにデータが並んでいます。

| Total Size(byte) | Size(byte) | |
|------------------|------------|----------------|
| 10h | 10h | グローバルインデックスヘッダ |
| 20h | 10h | PVR フォーマットヘッダ |
| Xxxxh + 20h | Xxxxh | Texture Data |

● 各テクスチャサイズのバイトサイズ

| Size(byte) | Texture Data |
|------------|-------------------|
| 80h | 8x8 Texture |
| 200h | 16x16 Texture |
| 800h | 32x32 Texture |
| 2000h | 64x64 Texture |
| 8000h | 128x128 Texture |
| 20000h | 256x256 Texture |
| 80000h | 512x512 Texture |
| 200000h | 1024x1024 Texture |

- Twiddled Mipmap フォーマットのデータ配置

Twiddled Mipmap フォーマットは、以下のようにデータが並んでいます。

| Total Size(byte) | Size(byte) | |
|------------------|------------|--------------------------|
| 10h | 10h | グローバルインデックスヘッダ |
| 20h | 10h | PVR フォーマットヘッダ |
| 22h | 2h | Dummy Zero |
| 24h | 2h | 1x1 Mipmap Texture |
| 2ch | 8h | 2x2 Mipmap Texture |
| 4Ch | 20h | 4x4 Mipmap Texture |
| CCh | 80h | 8x8 Mipmap Texture |
| 2CCh | 200h | 16x16 Mipmap Texture |
| ACCh | 800h | 32x32 Mipmap Texture |
| 2ACCh | 2000h | 64x64 Mipmap Texture |
| AACCh | 8000h | 128x128 Mipmap Texture |
| 2AACCh | 20000h | 256x256 Mipmap Texture |
| AAACCh | 80000h | 512x512 Mipmap Texture |
| 2AAACCh | 200000h | 1024x1024 Mipmap Texture |

ミップマップテクスチャの場合、PVR フォーマットヘッダの後ろにダミーバイトが 2 バイト入っています。（通常は 0 が入っています）

- Twiddled Mipmap Dma フォーマットのデータ配置

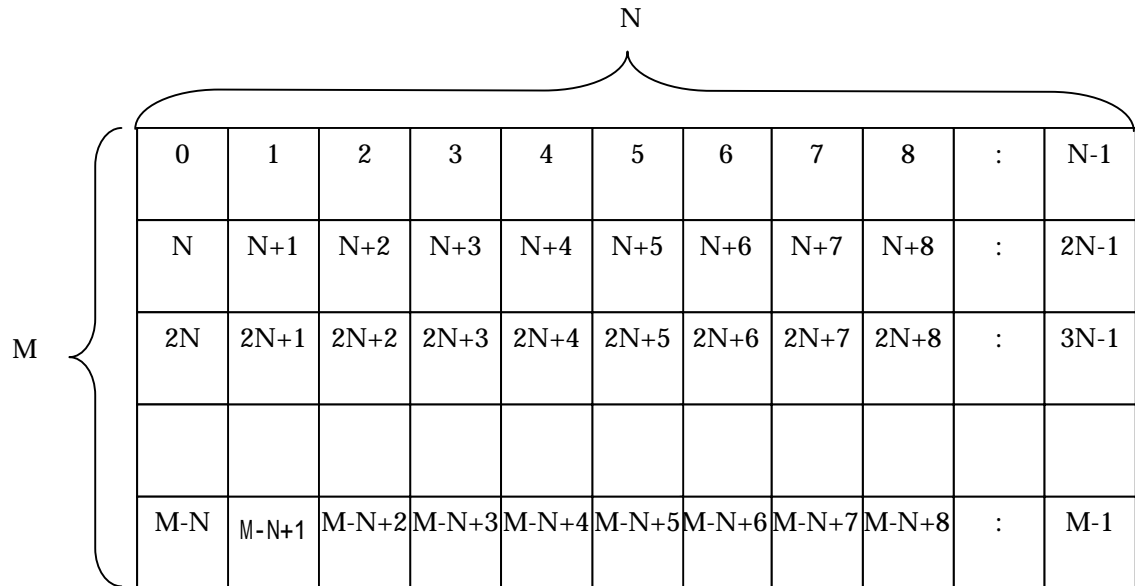
Twiddled Mipmap Dma フォーマットは、以下のようにデータが並んでいます。

| Total Size(byte) | Size(byte) | |
|------------------|------------|--------------------------|
| 10h | 10h | グローバルインデックスヘッダ |
| 20h | 10h | PVR フォーマットヘッダ |
| 26h | 6h | Dummy Zero |
| 28h | 2h | 1x1 Mipmap Texture |
| 30h | 8h | 2x2 Mipmap Texture |
| 50h | 20h | 4x4 Mipmap Texture |
| D0h | 80h | 8x8 Mipmap Texture |
| 2D0h | 200h | 16x16 Mipmap Texture |
| AD0h | 800h | 32x32 Mipmap Texture |
| 2AD0h | 2000h | 64x64 Mipmap Texture |
| AAD0h | 8000h | 128x128 Mipmap Texture |
| 2AAD0h | 20000h | 256x256 Mipmap Texture |
| AAAD0h | 80000h | 512x512 Mipmap Texture |
| 2AAAD0h | 200000h | 1024x1024 Mipmap Texture |

ミップマップテクスチャの場合、PVR フォーマットヘッダの後ろにダミーバイトが 6 バイト入っています。（通常は 0 が入っています）

(2) Rectangle フォーマット

ラスターオーダー順にテクスチャデータを並べたテクスチャフォーマットを Rectangle フォーマットといいます。メモリ上でテクスチャを変更する場合や、操作する場合には扱いやすいフォーマットです。Rectangle フォーマットは名前のとおり、長方形テクスチャフォーマットですから、Twiddled テクスチャのように正方形である必要はありません。また、Twiddled テクスチャに比べて、性能が低下する。Rectangle の場合は、レンダリングすることができます。ただし、縦、横のサイズは 8、16、32、64、128、256、512、1024 である必要があります。また、ミップマップフォーマットは存在しません。



ただし、N,M は 8、16、32、64、128、256、512、1024 とする

- Rectangle のデータ配置

Rectangle フォーマットは、以下のようにデータが並んでいます。

| Total Size(byte) | Size(byte) | |
|------------------|------------|----------------|
| 10h | 10h | グローバルインデックスヘッダ |
| 20h | 10h | PVR フォーマットヘッダ |
| Xxxxh + 20h | Xxxxh | Texture Data |

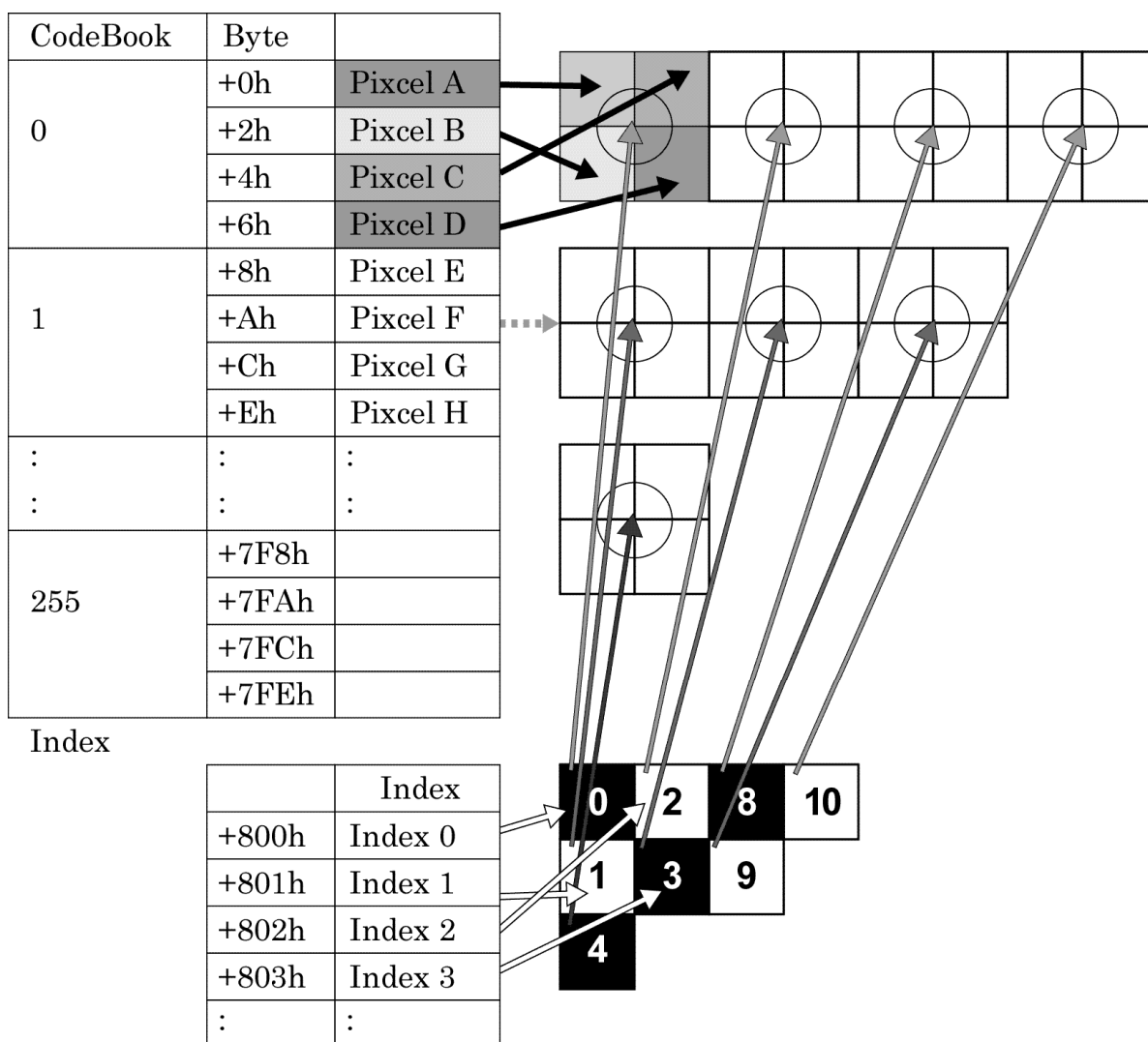
- テクスチャデータのバイトサイズの計算方法

| |
|---|
| Texture Size = Width x Height x 2 (bytes) |
|---|

(3) VQ、VQ Mipmap フォーマット

VQ とは Vector Quantization:ベクトル量子化の略で、VQ フォーマットはベクトル量子化を利用した高圧縮率の圧縮テクスチャフォーマットです。VQ テクスチャにはコードブックと呼ばれるカラーテーブルとコードブックの位置を示すインデックスデータが入っています。コードブックのサイズは常に 256 エントリあります。1 エントリあたり 4 テクセルのデータを保存しています。VQ テクスチャはレンダリング時にインデックスデータを元にコードブックを参照しながら伸長し画像を作成します。この点はパレットテクスチャと似ていますが、VQ テクスチャはパレットテクスチャとは異なりテクスチャ毎にコードブックを設定します。VQ テクスチャのインデックス部分は Twiddled テクスチャ形式にデータが並んでいます。VQ テクスチャのデータの展開は、ハードウェアが行うため VQ を使用することによるレンダリングの遅れはありません。また、VQ テクスチャは圧縮テクスチャフォーマットのためテクスチャメモリを有効に使用することができます。VQ テクスチャには 8×8、16×16、32×32 のミップマップあり、なしテクスチャと 64×64 のミップマップなしテクスチャはありません（このサイズは、Small VQ テクスチャになります）。

● Code Book



インデックス値から、対応するコードブックのデータを対応させます。

- VQ フォーマットのデータ配置

VQ フォーマットは以下のようにデータが並んでいます。

| Total Size(byte) | Size(byte) | |
|------------------|------------|----------------|
| 10h | 10h | グローバルインデックスヘッダ |
| 20h | 10h | PVR フォーマットヘッダ |
| 820h | 800h | Code Book Data |
| Xxxxh + 820h | Xxxxh | Index Data |

- 各テクスチャサイズのバイトサイズ

| Size(byte) | Index Data |
|------------|---------------------|
| 1000h | 128 × 128 Texture |
| 4000h | 256 × 256 Texture |
| 10000h | 512 × 512 Texture |
| 40000h | 1024 × 1024 Texture |

- VQ Mipmap フォーマットのデータ配置

VQ Mipmap フォーマットは以下のようにデータが並んでいます。

| Total Size(byte) | Size(byte) | |
|------------------|------------|-------------------------------|
| 10h | 10h | グローバルインデックスヘッダ |
| 20h | 10h | PVR フォーマットヘッダ |
| 820h | 800h | Code Book Data |
| 821h | 1h | 1 × 1 Mipmap Index Data |
| 822h | 1h | 2 × 2 Mipmap Index Data |
| 826h | 4h | 4 × 4 Mipmap Index Data |
| 836h | 10h | 8 × 8 Mipmap Index Data |
| 876h | 40h | 16 × 16 Mipmap Index Data |
| 976h | 100h | 32 × 32 Mipmap Index Data |
| D76h | 400h | 64 × 64 Mipmap Index Data |
| 1D76h | 1000h | 128 × 128 Mipmap Index Data |
| 5D76h | 4000h | 256 × 256 Mipmap Index Data |
| 15D76h | 10000h | 512 × 512 Mipmap Index Data |
| 5D576h | 40000h | 1024 × 1024 Mipmap Index Data |

1 × 1 ミップマップレベルの場合は、右下のテクセル値が表示されます

- 256 × 256 テクスチャを VQ 圧縮した場合

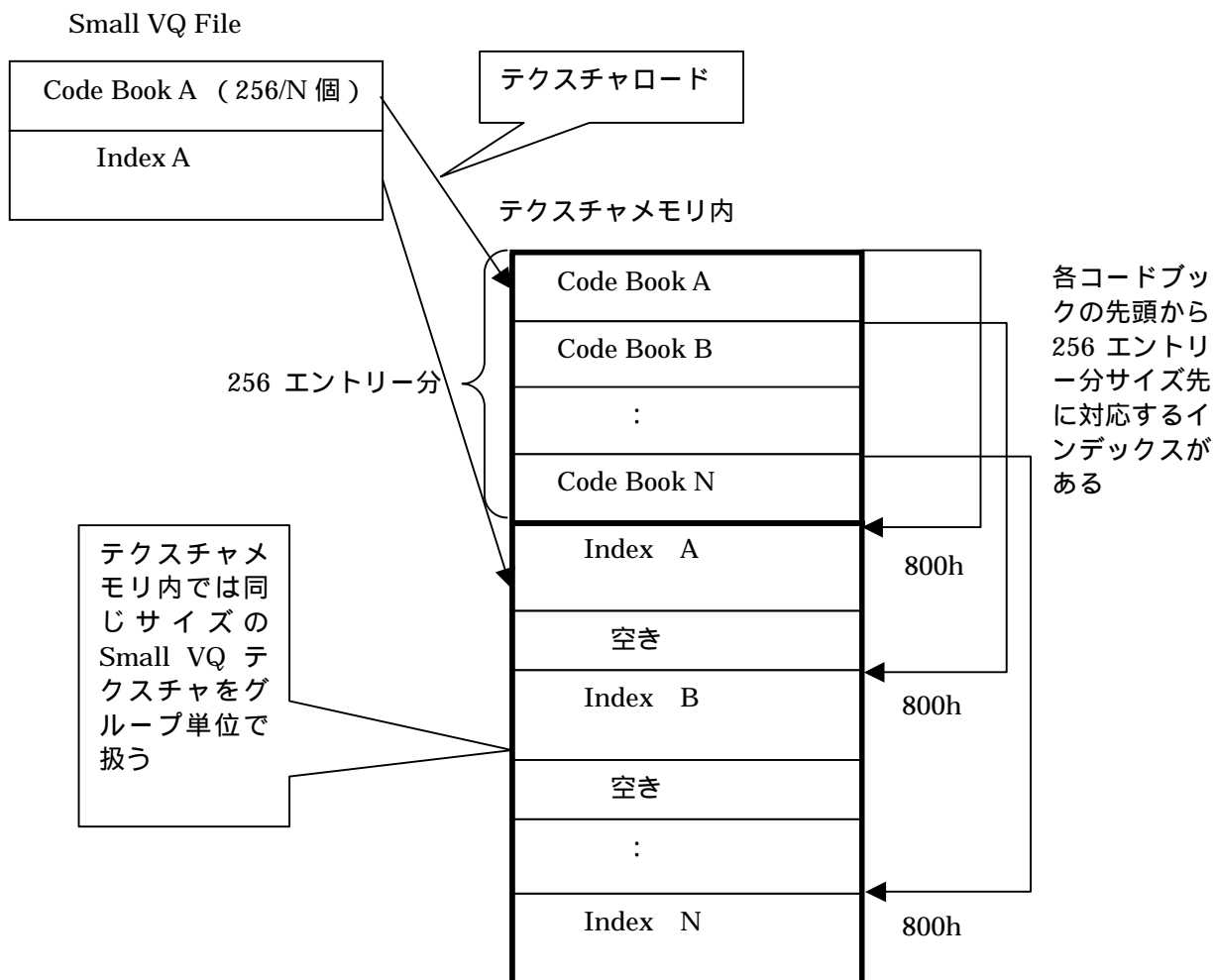
$$(256 \times 256 \times 2) : ((128 \times 128) + (256 \times 2 \times 4)) = 7.1 : 1$$

インデックス コードブック

この場合、VQ テクスチャ使用によりテクスチャサイズが約 7 分の 1 に圧縮されました。

(4) Small VQ、Small VQ MIPMAP フォーマット

VQ、VQ ミップマップテクスチャのコードブックは 256 個固定のため、テクスチャのサイズが 64×64 より小さい場合には、コードブックとインデックスを足したテクスチャサイズが圧縮しないテクスチャより大きくなります。そこでコードブックのサイズを限定した VQ 圧縮テクスチャを用意し、 64×64 以下の場合でも VQ 圧縮テクスチャを使用できるようにしました。この形式が Small VQ、Small VQ ミップマップフォーマットです。基本的な形式は VQ フォーマットと全く同じです。ハードウェア上の VQ テクスチャの仕様はコードブックの先頭から 256 個のコードブックサイズ (800h) の後、インデックスデータが入ることになっています。この仕様を元に、最も効率の良いコードブックのサイズを選択しています。また、Small VQ のテクスチャは同じサイズのテクスチャをグループ化してテクスチャメモリに保存しています。1つのテクスチャをロードする場合でも、1つのグループ分のテクスチャメモリを取ります。同一のサイズのテクスチャがグループの個数ロードされると効率が上がります。



● Small VQ, Small VQ Mipmap の各サイズとグループ数

| Texture Size | Mip Map | Code Book Size | | Index Size (Bytes) | 空き領域のサイズ (Bytes) | グループ数 | グループ全サイズ (Bytes) |
|--------------|---------|----------------|------------|--------------------|------------------|-------|------------------|
| | | Entry | Bytes | | | | |
| 8 × 8 | No | 16 | 128(80h) | 16(10h) | 112(70h) | 16 | 3984(F90h) |
| | Yes | 16 | 128(80h) | 32(20h) | 96(60h) | 16 | 4000(FA0h) |
| 16 × 16 | No | 16 | 128(80h) | 64(40h) | 64(40h) | 16 | 4032(FC0h) |
| | Yes | 16 | 128(80h) | 96(60h) | 32(20h) | 16 | 4064(FE0h) |
| 32 × 32 | No | 32 | 256(100h) | 256(100h) | 0 | 8 | 4096(1000h) |
| | Yes | 64 | 512(200h) | 352(160h) | 160(A0h) | 4 | 3936(F60h) |
| 64 × 64 | No | 128 | 1024(400h) | 1024(400h) | 0 | 2 | 4096(1000h) |
| | Yes | 256 | 2048(800h) | 1376(560h) | 0 | 1 | 3424(D60h) |

この場合、32x32、64x64 のミップマップなしが空き領域が 0 になり、無駄なテクスチャメモリがないため効率が良いことがわかります。逆に 8x8 のテクスチャは VQ 圧縮をした場合の方が Twiddled 形式のテクスチャよりサイズが大きくなるのでサポートしません。また 64x64 ミップマップありは VQ 形式と同じなので、Small VQ ではサポートしません。(VQ 形式テクスチャで対応) 網掛け部分の Small VQ 形式はサポートされていないので注意して下さい。

Small VQ、Small VQ Mipmap フォーマットは以下のようにデータが並んでいます。

● 16 × 16 Small VQ

| Total Size(byte) | Size(byte) | |
|------------------|------------|----------------|
| 10h | 10h | グローバルインデックスヘッダ |
| 20h | 10h | PVR フォーマットヘッダ |
| A0h | 80h | Code Book Data |
| E0h | 40h | Index Data |

● 32 × 32 Small VQ

| Total Size(byte) | Size(byte) | |
|------------------|------------|----------------|
| 10h | 10h | グローバルインデックスヘッダ |
| 20h | 10h | PVR フォーマットヘッダ |
| 120h | 100h | Code Book Data |
| 220h | 100h | Index Data |

● 64x64 Small VQ

| Total Size(byte) | Size(byte) | |
|------------------|------------|----------------|
| 10h | 10h | グローバルインデックスヘッダ |
| 20h | 10h | PVR フォーマットヘッダ |
| 420h | 400h | Code Book Data |
| 820h | 400h | Index Data |

- 16x16 Small VQ Mipmap

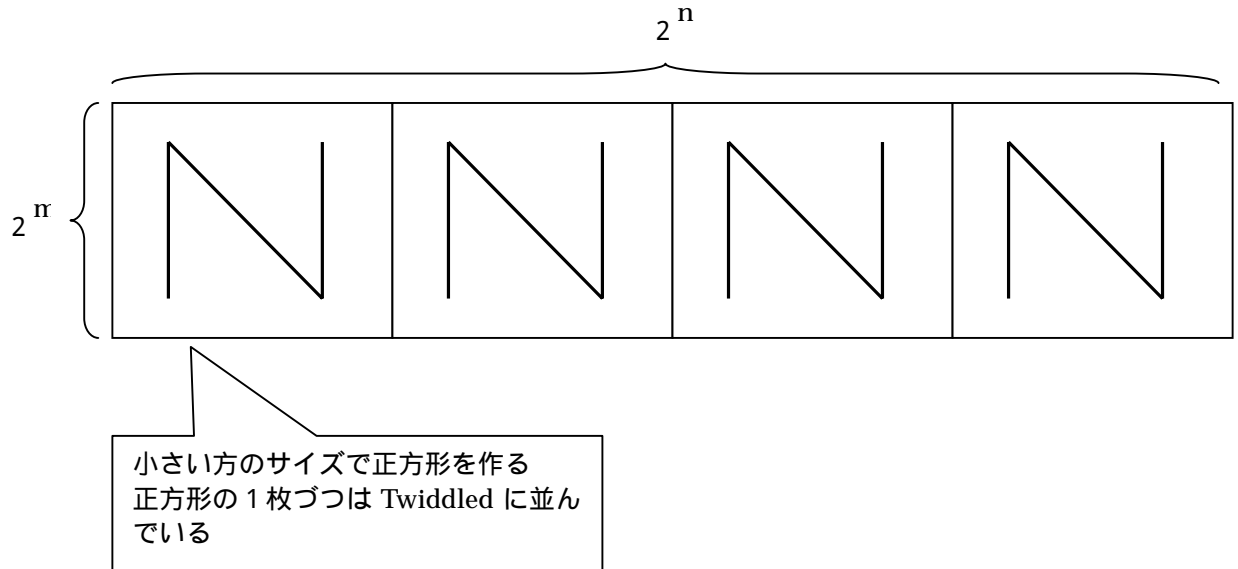
| Total Size(byte) | Size(byte) | |
|------------------|------------|---------------------------|
| 10h | 10h | グローバルインデックスヘッダ |
| 20h | 10h | PVR フォーマットヘッダ |
| A0h | 80h | Code Book Data |
| A1h | 1h | 1 × 1 Mipmap Index Data |
| A2h | 1h | 2 × 2 Mipmap Index Data |
| A6h | 4h | 4 × 4 Mipmap Index Data |
| B6h | 10h | 8 × 8 Mipmap Index Data |
| F6h | 40h | 16 × 16 Mipmap Index Data |
| 100h | Ah | Dummy Zero |

- 32x32 Small VQ Mipmap

| Total Size(byte) | Size(byte) | |
|------------------|------------|---------------------------|
| 10h | 10h | グローバルインデックスヘッダ |
| 20h | 10h | PVR フォーマットヘッダ |
| 220h | 200h | Code Book Data |
| 221h | 1h | 1 × 1 Mipmap Index Data |
| 222h | 1h | 2 × 2 Mipmap Index Data |
| 226h | 4h | 4 × 4 Mipmap Index Data |
| 236h | 10h | 8 × 8 Mipmap Index Data |
| 276h | 40h | 16 × 16 Mipmap Index Data |
| 376h | 100h | 32 × 32 Mipmap Index Data |
| 380h | Ah | Dummy Zero |

(5) Twiddled Rectangle フォーマット

Twiddled Rectangle フォーマットは、Twiddled フォーマットを縦・横サイズの小さい方のサイズで正方形にしたものを並べたデータ配置をしています。ただし、縦・横のサイズは8、16、32、64、128、256、512、1024 である必要があります。また、ミップマップは、Rectangle フォーマットと同じく指定することはできません。



Twiddled Rectangle フォーマットは、以下のようにデータが並んでいます。

| Total Size(byte) | Size(byte) | |
|------------------|------------|----------------|
| 10h | 10h | グローバルインデックスヘッダ |
| 20h | 10h | PVR フォーマットヘッダ |
| Xxxxh + 20h | Xxxxh | Texture Data |

- テクスチャデータのバイトサイズの計算方法

Texture Size = Width x Height x 2 (bytes)

(6) Palettized 4bpp/8bpp フォーマット

Palettized 4bpp/8bpp フォーマットとは、16 色パレットと 256 色パレットを利用したテクスチャフォーマットです。パレットはシステムに 1024 色 1 パレットあります。16 色を 1 単位として管理します。この管理単位をバンクと呼び、システム全体で 64 バンク存在します。16 色パレットでは 0 から 63 バンクが指定可能で、256 色パレットでは 0、16、32、48 バンクが指定可能です。(0 から 63 で設定できますが、下位 4 ビットは見えていません)システム上で各バンクは物理的に別れていません。1 シーン中で 16 色パレットテクスチャと 256 色パレットテクスチャの混在は可能ですが、バンクを重ねてパレットを登録した場合、後から登録されたパレットデータになります。ただし、パレットのカラーモードは 1 シーン中に変更することはできません。パレットにはパレットデータ(カラーデータ)とインデックスデータがあります。パレットデータは PVP ファイルとして扱います。PVR テクスチャでは、インデックスデータのみを扱い、パレットデータは入っていません。パレットのインデックスデータの並びは、Twiddled フォーマット形式に並んでいます。サイズが Rectangle の場合には Twiddled Rectangle フォーマット形式にデータが並びます。16 色パレットテクスチャでは 4 ビットが 1 テクセルのインデックスデータになり、256 色パレットテクスチャでは 1 バイト 1 テクセルになります。

● 16 色パレットのテクセルの並び順

| | | | | |
|-------|---------|--------|-------|-------|
| Bits | 15 - 12 | 11 - 8 | 7 - 4 | 3 - 0 |
| Texel | 3 | 2 | 1 | 0 |

| | |
|---|---|
| 0 | 2 |
| 1 | 3 |

● 256 色パレットのテクセルの並び順

| | | |
|-------|--------|-------|
| Bits | 15 - 8 | 7 - 0 |
| Texel | 1 | 0 |

● Palettized フォーマットのデータ配置

Palettized フォーマットは以下のようにデータが並んでいます。

| Total Size(bytes) | Size(bytes) | |
|-------------------|-------------|----------------|
| 10h | 10h | グローバルインデックスヘッダ |
| 20h | 10h | PVR フォーマットヘッダ |
| Xxxxh + 20h | Xxxxh | Texture Data |

● 各テクスチャサイズのバイトサイズ

| 16 色(bytes) | Index Data | 256 色(bytes) |
|-------------|----------------------|--------------|
| 20h | 8x8 Index Data | 40h |
| 80h | 16x16 Index Data | 100h |
| 200h | 32x32 Index Data | 400h |
| 800h | 64x64 Index Data | 1000h |
| 2000h | 128x128 Index Data | 4000h |
| 8000h | 256x256 Index Data | 10000h |
| 20000h | 512x512 Index Data | 40000h |
| 80000h | 1024x1024 Index Data | 100000h |

● Palettized Mipmap フォーマットのデータ配置

Palettized Mipmap フォーマットは以下のようにデータが並んでいます。

| 16 色(4bpp) | | | 256 色(8bpp) | |
|-------------------|--------------|----------------------|-------------|-------------------|
| Total Size(bytes) | Size (bytes) | Headr Index Data | Size(bytes) | Total Size(bytes) |
| 10h | 10h | グローバルインデックスヘッダ | 10h | 10h |
| 20h | 10h | PVR フォーマットヘッダ | 10h | 20h |
| 21h | 1h | Dummy Zero | 3h | 23h |
| 22h | 1h | 1x1 Index Data | 1h | 24h |
| 24h | 2h | 2x2 Index Data | 4h | 28h |
| 2Ch | 8h | 4x4 Index Data | 10h | 38h |
| 4Ch | 20h | 8x8 Index Data | 40h | 78h |
| CCh | 80h | 16x16 Index Data | 100h | 178h |
| 2CCh | 200h | 32x32 Index Data | 400h | 578h |
| ACCh | 800h | 64x64 Index Data | 1000h | 1578h |
| 2ACCh | 2000h | 128x128 Index Data | 4000h | 5578h |
| AACCh | 8000h | 256x256 Index Data | 10000h | 15578h |
| 2AACCh | 20000h | 512x512 Index Data | 40000h | 55578h |
| AAACCh | 80000h | 1024x1024 Index Data | 100000h | 155578h |
| AAAE0h | 14h | Dummy Zero | 8h | 155580h |

Dummy Data のサイズに注意して下さい。

- Palettized Twiddled Rectangle フォーマットのデータ配置

Palettized Twiddled Rectangle フォーマットは以下のようにデータが並んでいます。

| Total Size(bytes) | Size(bytes) | |
|-------------------|-------------|----------------|
| 10h | 10h | グローバルインデックスヘッダ |
| 20h | 10h | PVR フォーマットヘッダ |
| Xxxxh + 20h | Xxxxh | Texture Data |

- 各テクスチャサイズのバイトサイズ

- 16 色の場合のテクスチャデータのバイトサイズの計算方法

| |
|---|
| Texture Size = Width x Height ÷ 2 (bytes) |
|---|

- 256 色の場合のテクスチャデータのバイトサイズの計算方法

| |
|---------------------------------------|
| Texture Size = Width x Height (bytes) |
|---------------------------------------|

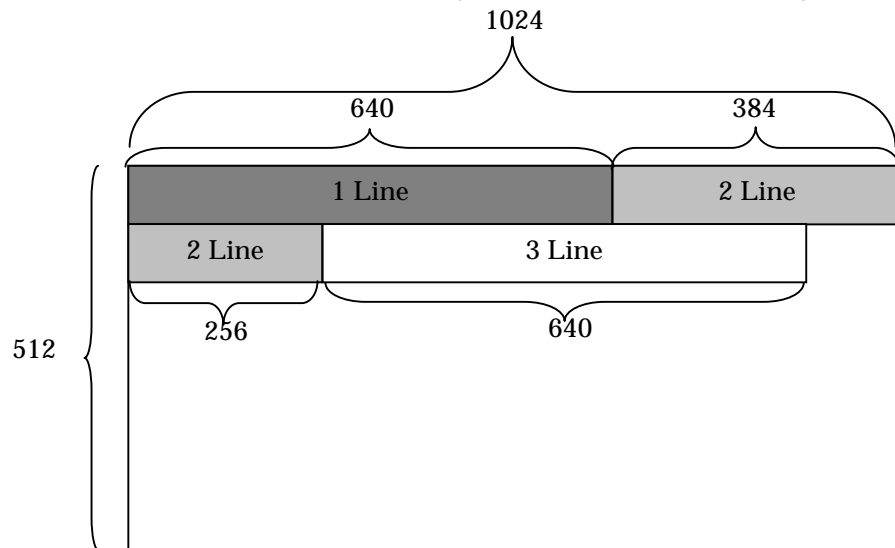
(7) Stride フォーマット

Stride フォーマットは、Rectangle フォーマットの特殊な形式で、Stride 値と呼ばれるテクスチャの横の値をグローバルで設定しておき、以下の計算方法によりテクスチャの UV 位置からアドレス位置を計算します。

$$\text{Addr} = U + V \times \text{Stride 値}$$

Stride 値は 32 の倍数のみ設定でき 992 まで設定できます。(1024 は設定できません) Stride 値は njSetRenderWidth 関数で設定します。Stride フォーマットで設定できるテクスチャのサイズは Rectangle フォーマットと全く同じで、縦・横のサイズが 8、16、32、64、128、256、512、1024 のテクスチャです。Stride テクスチャはレンダーテクスチャの場合に主に使用します。レンダーテクスチャのレンダリング先として指定した場合、メモリ上では以下のようなイメージでレンダリングされます。

例：1024x512 に 640x480 をレンダリングする (Stride 値は 640 と指定する)



このテクスチャを使用して、640x480 の領域をテクスチャとして使用したい場合は (U,V) = (0,0) - (0.625f, 0.9375f) を指定します。

$$0.625 = 640 / 1024 \quad 0.9375 = 480 / 512$$

- Stride フォーマットのデータ配置
Stride フォーマットは以下のようにデータが並んでいます。

| Total Size(byte) | Size(byte) | |
|------------------|------------|----------------|
| 10h | 10h | グローバルインデックスヘッダ |
| 20h | 10h | PVR フォーマットヘッダ |
| Xxxxh + 20h | Xxxxh | Texture Data |

- テクスチャデータのバイトサイズの計算方法
Texture Size = Width x Height x 2 (bytes)

4.15 カラーフォーマット

Ninja で使用できるカラーフォーマットについて説明します。

- 通常のテクスチャカラーフォーマット
Ninja で通常使用するカラーフォーマットには ARGB1555、ARGB4444、RGB565 のテクスチャカラーフォーマットがあります。
- YUV422 フォーマット
YUV422 のフォーマットです。1 ピクセルあたり 8 ビットで表すことができます。
- Bump フォーマット
バンプマップ用のテクスチャフォーマットです。
- ARGB8888 フォーマット
PALETTIZED 用のフォーマットです。
- Ninja でサポートするテクスチャフォーマット

| | ARGB1555 | RGB565 | ARGB4444 | YUV422 | Bump | ARGB8888 |
|----------------|----------|--------|----------|--------|------|----------|
| TWIDDLED | | | | | | × |
| TWIDDLED MM | | | | | | × |
| VQ | | | | | | × |
| VQ MM | | | | | | × |
| PALETTIZED 4,8 | | | | × | × | |
| PALETTIZED MM | | | | × | × | |
| RECTANGLE | | | | | | × |
| STRIDE | | | | | | × |

： 現在対応しているもの
×： 機能的に対応できないもの

4.16 PVM フォーマット (.pvm)

PVM フォーマットには多くのチャンクが設定できますが、ここでは最も基本的な PVM チャンクと PVP チャンク、PVR チャンクについて説明します。その他について詳しくは、PVM 仕様書を参照して下さい。

| | | |
|----------------------|--------------|-------|
| PVM フォーマット ヘッダ | “PVMH”チャンク | 4 バイト |
| | ネクストタグへのバイト数 | 4 バイト |
| | PVM ステータス | 2 バイト |
| | エントリーカウント | 2 バイト |

| | |
|-----------------|--------|
| エントリーID | 2 バイト |
| PVR ファイルネーム | 28 バイト |
| バンク・グローバルインデックス | 4 バイト |

影部分はエントリーカウント分繰り返されます。

| | | |
|----------------------|--------------|-------|
| PVP フォーマット ヘッダ | “PVPL”チャンク | 4 バイト |
| | ネクストタグへのバイト数 | 4 バイト |
| | カラーフォーマット | 2 バイト |
| | バンク | 2 バイト |
| | エントリーオフセット | 2 バイト |
| | エントリーカウント | 2 バイト |

PVP フォーマットヘッダは必ずあるわけではありません。

| | | |
|----------------------|--------------|-------|
| PVR フォーマット ヘッダ | “PVRT”チャンク | 4 バイト |
| | ネクストタグへのバイト数 | 4 バイト |
| | テクスチャアトリビュート | 4 バイト |
| | 横サイズ | 2 バイト |
| | 縦サイズ | 2 バイト |
| テクスチャデータ本体 | | |

影部分はエントリーカウント分繰り返されます。

● PVM ステータス

PVM フォーマットヘッダの次に含まれるテクスチャの情報に入っているデータのフラグを設定します。

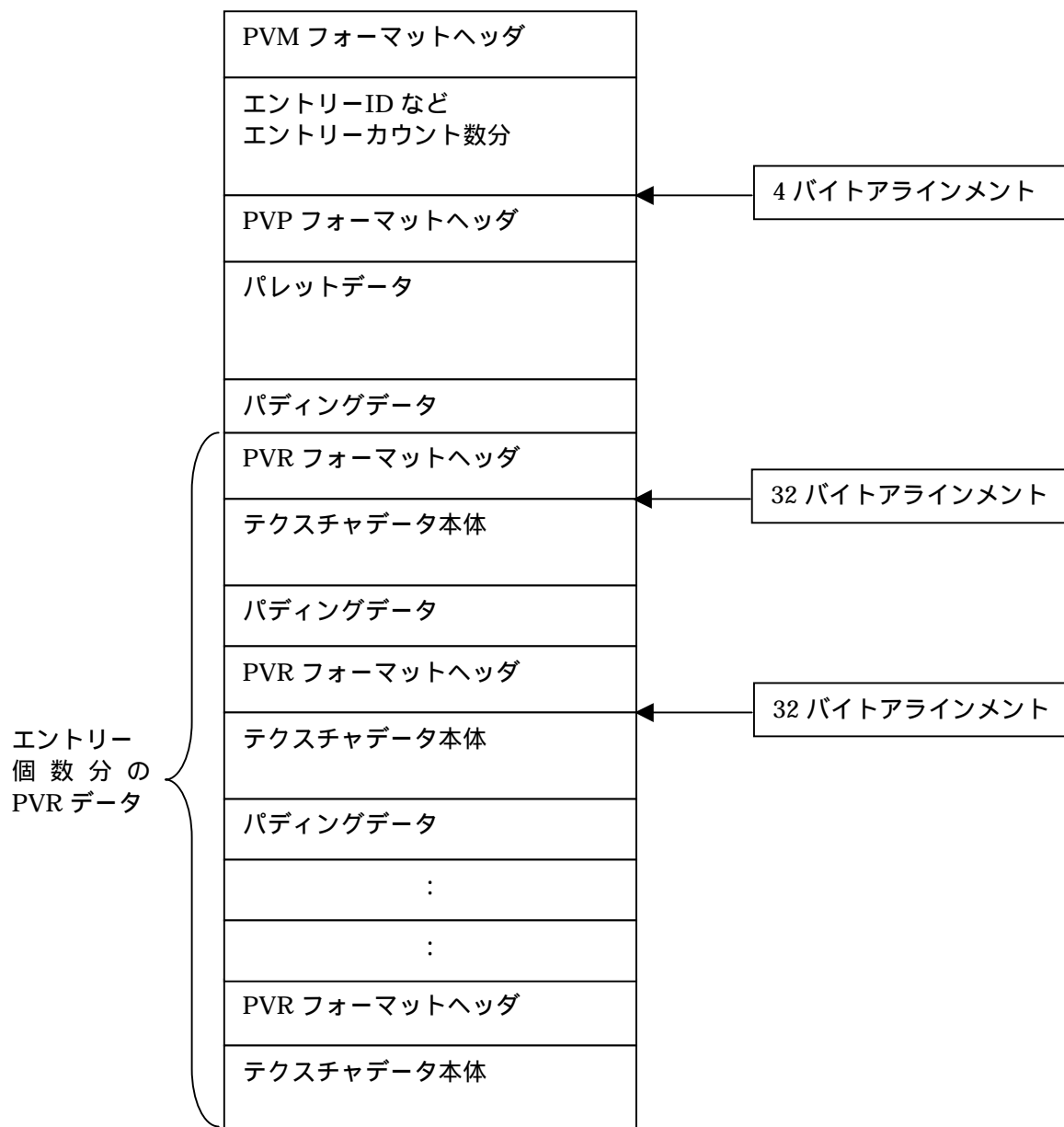
```
#define PVMH_PS_PVRNAME      (1<<0)
#define PVMH_PS_CATEGORYCODE (1<<1)
#define PVMH_PS_ENTRYINFO    (1<<2)
#define PVMH_PS_GLOBALINDEX  (1<<3)
#define PVMH_PS_BANKID (1<<10)
```


- エントリーカウント

PVM ファイル中に含まれる PVR データの数です。

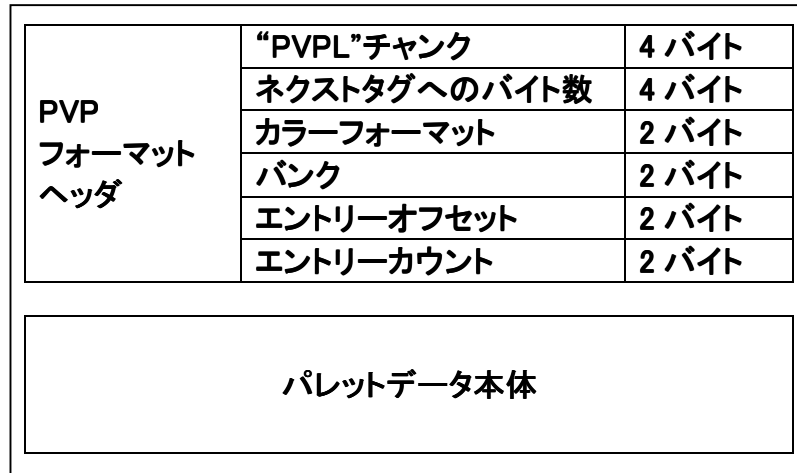
その他詳しいフォーマットの情報は、PVM ファイルフォーマットの仕様書を参照して下さい。

PVRT チャンクは次のように 32 バイトアラインメントを合わせる必要があります



4.17 PVP フォーマット (.pvp)

NINJA で使用するパレットのカラーデータファイルフォーマットです。PVP フォーマットのファイルにはPVP フォーマットヘッダ、パレットデータ本体が入っています。パレットで使えるカラーフォーマットはARGB1555、RGB565、ARGB4444、ARGB8888 カラーがあります。バンクはパレットデータのロード先バンク 0 ~ 63、エントリーオフセットにはバンクからのオフセット数 0 ~ 1023、エントリーカウントはファイルに入っているパレットのデータ数が入っています。



- ネクストタグへのバイト数
パレットデータ本体のバイト数 + 8 バイトの数が入ります。

- カラーフォーマット
パレットで使えるカラーフォーマットは、以下の4つです。

```
#define NJD_TEXFMT_ARGB1555 (0x0000)
#define NJD_TEXFMT_RGB565   (0x0001)
#define NJD_TEXFMT_ARGB4444 (0x0002)
#define NJD_TEXFMT_ARGB8888 (0x0006)
```

- バンク
システムにパレットは、1024 色あります。このパレットを、16 色 1 単位として扱います。この単位をバンクと呼びます。バンクは 0 ~ 63 まであり、このデータにはロード先のバンクが入ります。
- エントリーオフセット
バンクからのオフセットが入ります。ロード先のパレット番号は以下の計算になります。

$$\text{ロード先頭のパレット番号} = \text{バンク番号} \times 16 + \text{エントリーオフセット}$$

- エントリーカウント
ファイルに入っているパレットのデータ数が入ります。1 ~ 1024 が指定できます。パレットすべてを書き換える場合には、バンクを 0、エントリーオフセットを 0、エントリーカウントを 1024 とすると全部のパレットを一度に書き換えることができます。
- パレットデータ
ARGB1555、RGB565、ARGB4444 は 2 バイトのパレットデータ、ARGB8888 は 4 バイトのパレットデータです。エントリーカウント数分のデータが入っています。

4.18 テクスチャカラーモード

テクスチャのカラーモードフォーマットには ARGB1555、RGB565、ARGB4444、ARGB8888、YUV422、Bump があります。カテゴリーコード毎に使用できるカラーフォーマットと使用できないカラーフォーマットがあります。下の表には、カラーフォーマットとカテゴリーコード毎の対応関係を書きます。

| | ARGB1555 | RGB565 | ARGB4444 | YUV422 | Bump | ARGB8888 |
|-------------------|----------|--------|----------|--------|------|----------|
| Twiddled | | | | | | × |
| VQ, Small VQ | | | | × | × | × |
| Rectangle, Stride | | | | | | × |
| Palettized 4/8bpp | | | | × | × | |

× は、ハード未対応を示します。

ARGB1555、RGB565、ARGB4444、YUV422、Bump カラーは 1 ピクセルあたり 2 バイト、ARGB8888 カラーは 1 ピクセルあたり 4 バイトです。

Palettize 4bpp テクスチャのインデックスは 1 ピクセルあたり 4 ビット、Palettized 8bpp テクスチャのインデックスは 1 ピクセルあたり 8 ビットです。VQ 形式、Small VQ 形式のインデックスは、1 ピクセルあたり 8 ビットです。

5 基本的な描画関数について

ここでは、基本的な描画関数の使用方法と注意点を説明します。

5.1 2 D プリミティブ

5.1.1 概要

2 D プリミティブの関数は、データ頂点データをそのまま加工せずにハードウェアに転送します。Z 値を設定できますので、プライオリティを付けることができます。

5.1.2 ポリゴン

- (1) njDrawPolygon
バックカラーモードでポリゴンを 1 ストリップ描画します。関数をコールする度に 32 バイトのグローバルデータを付加します。1 頂点 32 バイトのデータになります。
- (2) njDrawPolygon2DExStart
グローバルデータとストリップデータを分けた関数です。開始と終了の間に違う描画関数をコールすることはできません。
- (3) njDrawPolygon2DExStart
描画の開始を宣言します。このときに 32 バイトのグローバルデータをセットします。
- (4) njDrawPolygon2DExSetData
1 ストリップのポリゴンを描画します。1 頂点 32 バイトで、何回でもコールすることができます。
- (5) njDrawPolygon2DExEnd
描画の終了です。

5.1.3 テクスチャ

- (1) njDrawTextureEx
バックカラーモードでテクスチャを 1 ストリップ描画します。関数をコールする度に 32 バイトのグローバルデータを付加します。1 頂点 32 バイトのデータになります。
- (2) njDrawTexture2DExStart
グローバルデータとストリップデータを分けた関数です。開始と終了の間に違う描画関数をコールすることはできません。
- (3) njDrawTexture2DExStart
描画の開始を宣言します。このときに 32 バイトのグローバルデータをセットします。
- (4) njDrawTexture2DExSetData
1 ストリップのテクスチャを描画します。1 頂点 32 バイトで、何回でもコールすることができます。
- (5) njDrawTexture2DExEnd
描画の終了です。

5.1.4 ハイライトテクスチャ

- (1) njDrawTextureHEX
パックカラーモードでハイライト付テクスチャを 1 ストリップ描画します。関数をコールする度に 32 バイトのグローバルデータを付加します。1 頂点 32 バイトのデータになります。
- (2) njDrawTexture2DHEXStart
グローバルデータとストリップデータを分けた関数です。開始と終了の間に違う描画関数をコールすることはできません。
- (3) njDrawTexture2DHEXStart
描画の開始を宣言します。このときに 32 バイトのグローバルデータをセットします。
- (4) njDrawTexture2DHEXSetData
1 ストリップのハイライト付テクスチャを描画します。1 頂点 32 バイトで、何回でもコールすることができます。
- (5) njDrawTexture2DHEXEnd
描画の終了です。

5.1.5 クアッドテクスチャ

4 角形ポリゴンを描画します。

- (1) njQuadTextureStart
描画の開始を宣言します。グローバルデータのセットは行いません。
- (2) njQuadTextureEnd
描画の終了です。
- (3) njSetQuadTexture
テクスチャとカラーを設定します。グローバルデータ 32 バイトを設定します。
- (4) njSetQuadTextureG
テクスチャとカラーを設定します。グローバルデータ 32 バイトを設定します。
- (5) njSetQuadTextureColor
カラーを設定します。グローバルデータ 32 バイトを設定します。
- (6) njDrawQuadTexture
対角 2 頂点を指定して描画します。1 データ 64 バイトになります。
- (7) njDrawQuadTextureEx
1 頂点と 2 本のベクトルを指定して描画します。1 データ 64 バイトになります。

5.1.6 ライン

- (1) njDrawLine2DEx
ラインを描画します。
- (2) njDrawLineExStart
ラインの描画開始です。32 バイトのグローバルをセットします。2 D、3 D で共通の関数です。
- (3) njDrawLineExEnd
ラインの終了です。2 D、3 D で共通です。
- (4) njDrawLine2DExSetStrip
ストリップでラインを描画します。1 ライン 64 バイトです。
- (5) njDrawLine2DExSetList
独立したラインを描画します。1 ライン 64 バイトです。

5.2 3 D プリミティブ

5.2.1 概要

カレントマトリクスでマトリクス変換し、透視変換を行って描画します。

5.2.2 ポリゴン

- (1) njDrawPolygon3DEx
バックカラーモードでポリゴンを1 ストリップ描画します。関数をコールする度に 32 バイトのグローバルデータを付加します。1 頂点 32 バイトのデータになります。
- (2) njDrawPolygon3DExStart
グローバルデータとストリップデータを分けた関数です。開始と終了の間に違う描画関数をコールすることはできません。
- (3) njDrawPolygon3DExStart
描画の開始を宣言します。このときに 32 バイトのグローバルデータをセットします。
- (4) njDrawPolygon3DExSetData
1 ストリップのポリゴンを描画します。1 頂点 32 バイトで、何回でもコールすることができます。
- (5) njDrawPolygon3DExEnd
描画の終了です。

5.2.3 テクスチャ

- (1) njDrawTexture3DEx
パックカラーモードでテクスチャを 1 ストリップ描画します。関数をコールする度に 32 バイトのグローバルデータを付加します。1 頂点 32 バイトのデータになります。
- (2) njDrawTexture3DExStart
グローバルデータとストリップデータを分けた関数です。開始と終了の間に違う描画関数をコールすることはできません。
- (3) njDrawTexture3DExStart
描画の開始を宣言します。このときに 32 バイトのグローバルデータをセットします。
- (4) njDrawTexture3DExSetData
1 ストリップのテクスチャを描画します。1 頂点 32 バイトで、何回でもコールすることができます。
- (5) njDrawTexture3DExEnd
描画の終了です。

5.2.4 ハイライトテクスチャ

- (1) njDrawTexture3DHEx
パックカラーモードでハイライト付テクスチャを 1 ストリップ描画します。関数をコールする度に 32 バイトのグローバルデータを付加します。1 頂点 32 バイトのデータになります。
- (2) njDrawTexture3DHExStart
グローバルデータとストリップデータを分けた関数です。開始と終了の間に違う描画関数をコールすることはできません。
- (3) njDrawTexture3DHExStart
描画の開始を宣言します。このときに 32 バイトのグローバルデータをセットします。
- (4) njDrawTexture3DHExSetData
1 ストリップのハイライト付テクスチャを描画します。1 頂点 32 バイトで、何回でもコールすることができます。
- (5) njDrawTexture3DHExEnd
描画の終了です。

5.3 ライン

5.3.1 njDrawLine3DEx

ラインを描画します。

- (1) njDrawLineExStart
ラインの描画開始です。32 バイトのグローバルをセットします。2 D、3 D で共通の関数です。
- (2) njDrawLineExEnd
ラインの終了です。2 D、3 D で共通です。
- (3) njDrawLine3DExSetStrip
ストリップでラインを描画します。1 ライン 64 バイトです。
- (4) njDrawLine3DExSetList
独立したラインを描画します。1 ライン 64 バイトです。

5.3.2 一覧

3D の座標系で描画する関数に以下のものがあります。低速関数群はNinja2で削除される予定です。

| 高 速 | 低 速 |
|--|--|
| njDrawPolygon3DEx ~ njDrawTexture3DEx ~ njDrawTexture3DHEx ~ | njDrawLine3D njDrawPoint3D njDrawPolygon3D njDrawTriangle3D |

5.3.3 頂点形式

- (1) njDrawPolygon3DEx ~
NonTexture Packed Color 形式 (PolygonType0) で一頂点 32 バイトです。グローバルパラメータをセットする関数と頂点をセットする関数はわかれています。
- (2) njDrawTexture3DEx ~
Textured Packed Color 形式 (PolygonType3) で一頂点 32 バイトです。グローバルパラメータをセットする関数と頂点をセットする関数はわかれています。
- (3) njDrawTexture3DHEx ~
Textured Packed Color with Offset Color 形式 (PolygonType3) で一頂点 32 バイトです。グローバルパラメータをセットする関数と頂点をセットする関数はわかれています。

5.3.4 Ex 関数群について

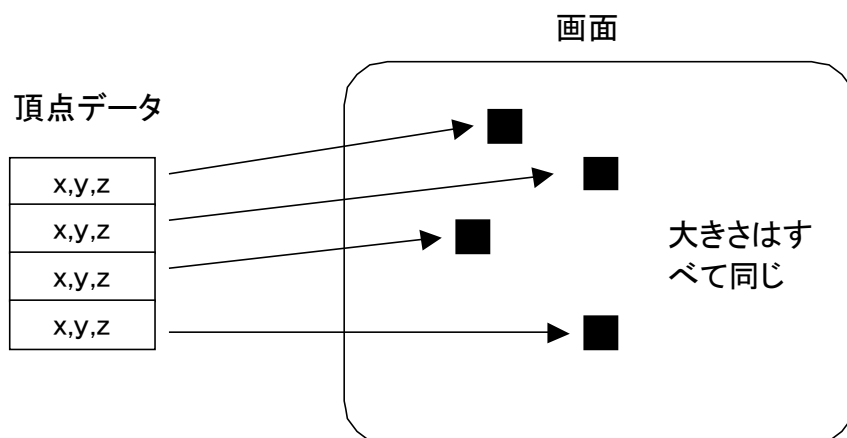
- グローバルデータの設定と頂点情報の設定にわかれています。
- 登録開始 (Start) から終了 (End) までの間、他の描画関数は使用できません。
- アセンブラで最適化されており従来関数より高速です (当社比) 。

5.4 パーティクル

パーティクルとは、一連の座標データを一括して描画するものです。現在 3 種類の描画関数があります。

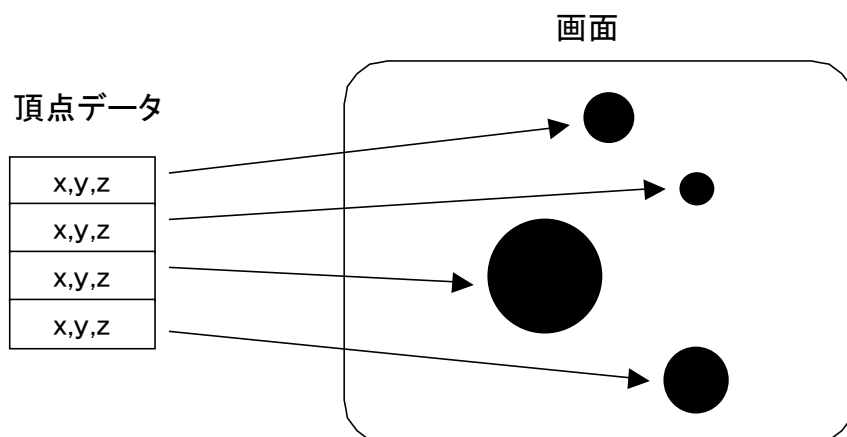
5.4.1 パーティクルポリゴン

一連の頂点データを一括してマトリクス変換、透視変化して描画します。1 頂点 1 ポリゴンで、Quad 形式で描画します。頂点データは座標のみで、ポリゴンの大きさは、関数パラメタで指定します。また、Z 値に応じて大きさは変化しません。



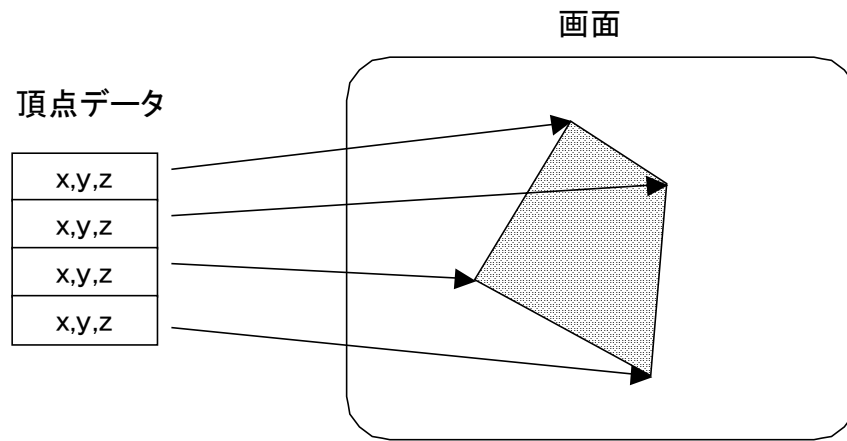
5.4.2 パーティクルスプライト

パーティクルポリゴンと同じですが、こちらはテクスチャの Quad 形式で描画します。UV 値は指定できませんので丸ごと表示します。また、Z 値に応じて大きさが変わります。



5.4.3 パーティクルストリップ

これは一連の頂点データをストリップとみなして描画します。つまり、頂点データ数分の多角形を描画します。



6 セルスプライト

ここでは、セルやセルスプライトについて説明します。

6.1 概要

セルスプライトは、複数のセルを一括し動作、描画するものです。
各セルのストリーム処理やセルスプライト全体をモーションさせることができます。

6.2 セルとセルスプライトについて

6.2.1 セルとセルスプライトの関係

セルスプライトで使用するセルは、矩形 1 枚が 1 つのセルの最小単位です。セルスプライトとは複数のセルを一括して扱う単位をいいます。

例えば、顔をセルスプライトで作成する場合、目・口・鼻などのパーツの 1 つ 1 つがセルで、それらのパーツをまとめて顔として扱うのがセルスプライトです。

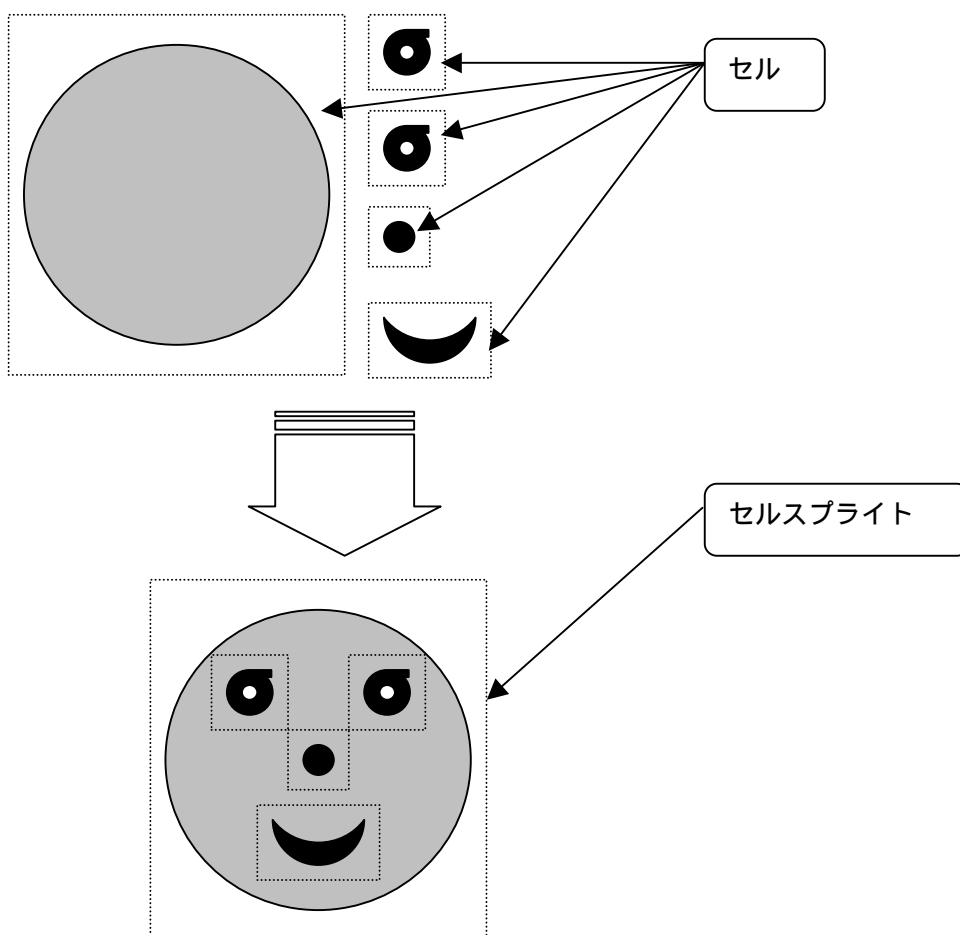


図 6 - 1 セルとセルスプライトの関係

6.2.2 セルスプライト構造体

セルスプライト構造体のメンバについて説明します。

```
typedef struct {
    NJS_CELL          *cells;
    Int               nbCell;
    NJS_POINT3        pos;
    Angle             zang;
    Float             sx, sy;
    NJS_ARGBdiffuse;
    NJS_ARGBspecular;
} NJS_CELL_SPRITE;
```

(1) cells

セルスプライトを構成するすべてのセルの 1 次元配列を格納します。

(2) nbCell

セルスプライトを構成するすべてのセルの個数を設定します。

(3) pos

セルスプライトの中心を設定する。セルの描画位置はこの位置を基準に計算します。Z 値は、2D セルスプライトのときはプライオリティとして使用し、3D セルスプライトのときは、Z 座標として扱います。

(4) zang

セルスプライトの回転角を設定します。

(5) sx,sy

セルスプライトのスケールを設定します。スケールの中心はセルスプライトの中心とします。

(6) diffuse

マテリアルディフューズカラーを設定します。すべてのセルの頂点カラーデータと掛け算しセルのベースカラーにする。設定できる値はそれぞれ、0.f から 1.f。njSetCellSpriteMaterial 関数でディフューズカラー値を設定した場合は、この値は使用しない。

(7) specular

マテリアルスペキュラーカラーを設定する。設定値と 255.f を掛け算し、セルのオフセットカラーとする。設定できる値はそれぞれ、0.f から 1.f。ただし、a は無視します。njSetCellSpriteMaterial 関数でスペキュラーカラー値を設定した場合は、この値は使用しない。

セルスプライトの pos,zang,sx,sy,diffuse,specular メンバについてはセルスプライト描画関数を使用するときに、NJS_CELL_SPRITE_VAL 構造体にデータを設定することで、NJS_CELL_SPRITE_VAL 構造体に設定した値を使用することができます。

6.2.3 セル構造体

セル構造体のメンバについて説明します。

```
typedef struct {
    Sint16      texId;
    Sint16      attr;
    Float       cox, coy;
    Float       csx, csy;
    Sint16      czang;
    Sint16      cp;
    Float       cent_x, cent_y;
    NJS_COLOR   argb[4];
    Float       u0, v0;
    Float       u2, v2;
} NJS_CELL;
```

(1) texId

カレントテクスチャリストのテクスチャ番号を設定します。

(2) attr

セルのアトリビュートを設定します。詳しいアトリビュートに関しては、セルスプライトの仕様を参照して下さい。

注 意 アトリビュートは、njSetCellSpriteAttr 関数で変更することができます。ただし、この関数を使用するとセルスプライトに登録されているすべてのセルのアトリビュートが変更されてしまうので注意して下さい。

(3) cox,coy

セルスプライトの中心 pos からのオフセットを設定します。

(4) csx,csy

セルサイズを設定します。

(5) czang

セルの回転を設定します。回転の中心は cent_x,cent_y です。

(6) cp

セル間のプライオリティを設定する。セルスプライトの pos.z から cp/32768 を加えた値をセルの Z 値にします。CP で設定できる値は、 - 32768 から 32767 です。

(7) cent_x,cent_y

セルの中心を設定します。セルの左上を (0.f,0.f) とし右下を (1.f,1.f) とする。テクスチャの UV と同じ設定方法です。(0.5f,0.5f) でセルのちょうど真中に設定できます。セルの回転、移動の中心になります。

(8) argb[4]

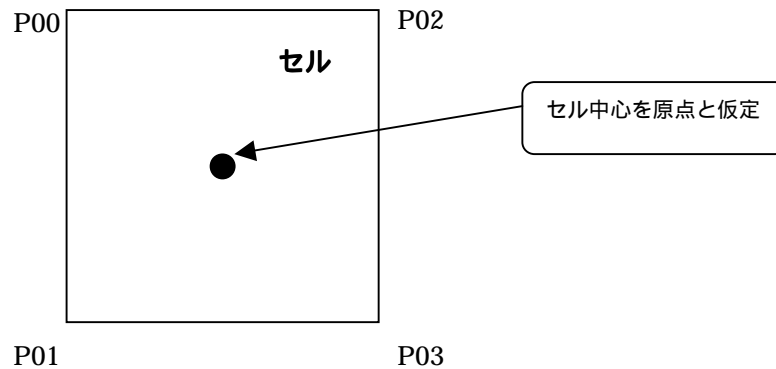
セルの頂点カラーを設定します。セルスプライトのディフューズカラー、または njSetCellSpriteMaterial 関数で設定したディフューズカラーと掛け算した値をセルのベースカラーとします。

(9) u0,v0、u2,v2

テクスチャ UV 値を設定します。それぞれセルの左上、右下です。

6.2.4 セルスプライト 2D の計算

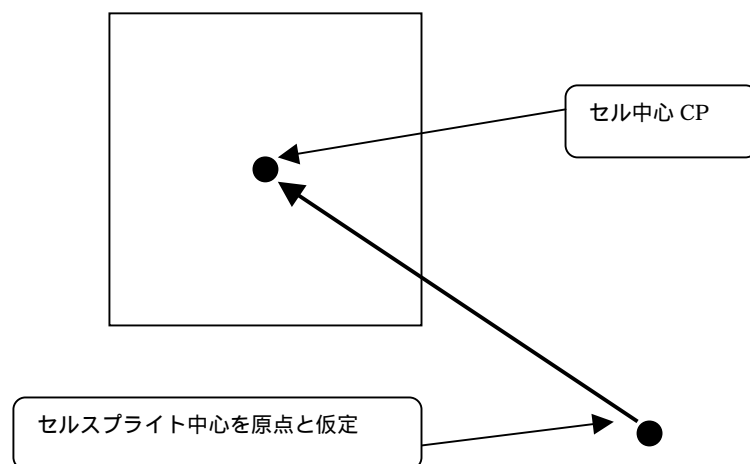
セルスプライト 2D のセル頂点の計算方法です。



セル中心の座標を原点と仮定すると P0(P00、P01、P02、P03)の座標は、セルサイズ CSX、CSY、セル中心 CENT_X、CENT_Y、セルスプライトのスケール SX、SY より、各セルの座標は

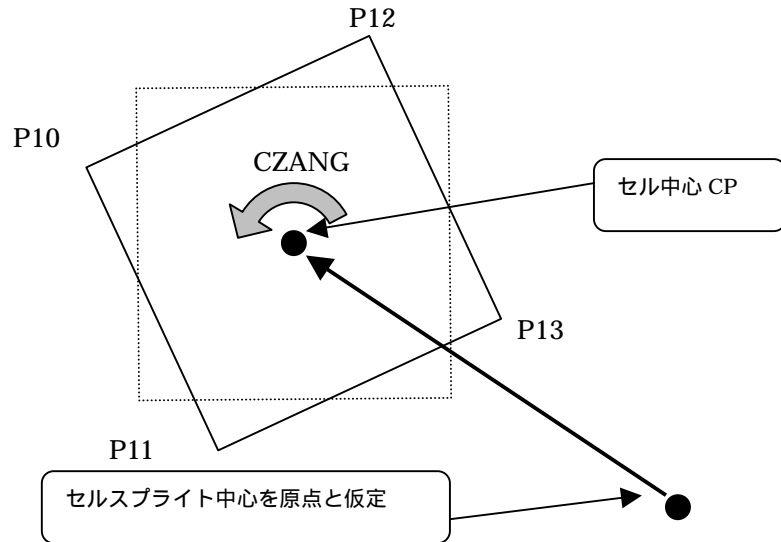
$$\begin{aligned} P00(X) &= -CSX \times SX \times CENT_X \\ P01(X) &= P00(X) \\ P02(X) &= CSX \times SX + P00(X) \\ P03(X) &= P02(X) \end{aligned}$$

$$\begin{aligned} P00(Y) &= -CSY \times SY \times CENT_Y \\ P02(Y) &= P00(Y) \\ P01(Y) &= CSY \times SY + P00(Y) \\ P03(Y) &= P01(Y) \end{aligned}$$



セルスプライト中心の座標を原点と仮定すると、セルスプライト中心からのオフセット COX、COY、セルスプライトのスケール SX、SY よりセル中心 CP は

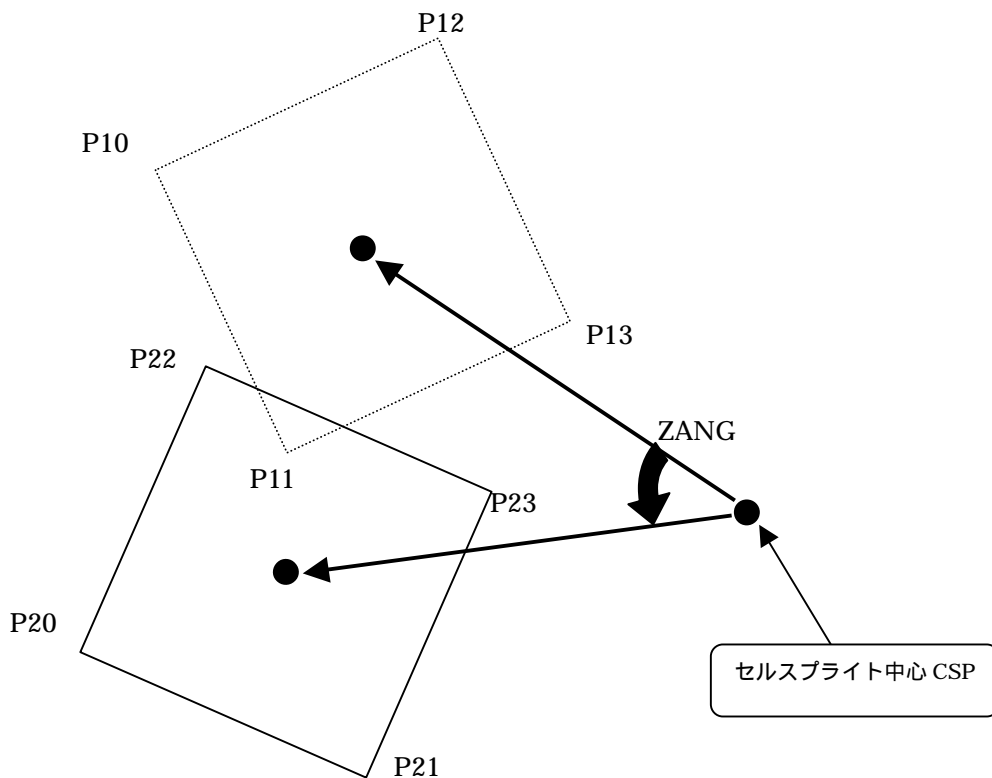
$$\begin{aligned} CP(X) &= COX \times SX \\ CP(Y) &= COY \times SY \end{aligned}$$



このときセル中心で $P0$ を $CZANG$ 回転したセルの点を $P1$ ($P10$ 、 $P11$ 、 $P12$ 、 $P13$) とすると

$$P1(X) = P0(X) \times \cos(CZANG) - P0(Y) \times \sin(CZANG) + CP(X)$$

$$P1(Y) = P0(X) \times \sin(CZANG) + P0(Y) \times \cos(CZANG) + CP(Y)$$



セルスプライトの中心で $ZANG$ 回転させセルスプライト中心を CSP にしたセルの点を $P2$ ($P20$ 、 $P21$ 、 $P22$ 、 $P23$) とすると

$$P2(X) = P1(X) \times \cos(ZANG) - P1(Y) \times \sin(ZANG) + CSP(X)$$

$$P2(Y) = P1(X) \times \sin(ZANG) + P1(Y) \times \cos(ZANG) + CSP(Y)$$

セルの Z 値はセルのプライオリティを CPZ とすると

$$P2(Z) = CSP(Z) + D32768 \times CP \quad Z \quad (\text{ただし } D32768 = 1/32768)$$

セルの頂点ベースカラーは、セルの頂点カラーを ARGB、セルスプライトのディフューズカラーを DIFFUSE とすると

$$BASE = ARGB \times DIFFUSE$$

セルの頂点オフセットカラーは、セルスプライトのスペキュラカラーを SPECULAR とすると

$$OFFSET = 255.f \times SPECULAR$$

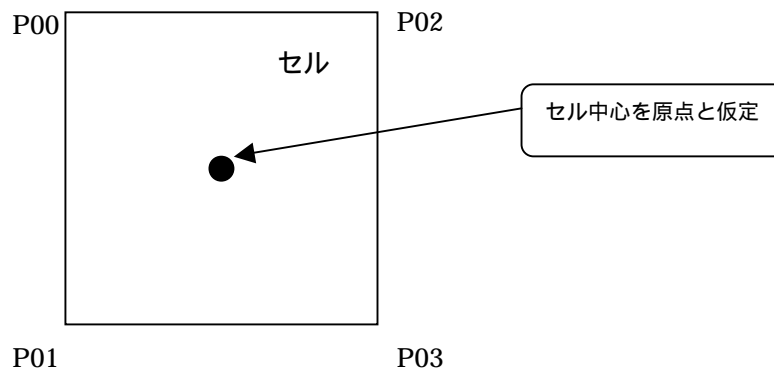
6.2.5 セルスプライト 3 D の計算

セルスプライト 3 D のセル頂点の計算方法です。

セルスプライト中心を CSP とし、マトリックス変換したあとのセルスプライト中心を CSPM とする。セルのプライオリティを CP とし、透視変換後のセルスプライトの中心を CSPP とすると

$$\begin{aligned} XAD &= SCREEN(XAD) \\ YAD &= SCREEN(XAD) \\ CX &= SCREEN(CX) \\ CY &= SCREEN(CY) \end{aligned}$$

$$\begin{aligned} CSPP(Z) &= -1.f \div (CSPM(Z) + CP \times D32768) \\ CSPP(X) &= CSPM(X) \times XAD \times CSPP(Z) + CX \\ CSPP(Y) &= CSPM(Y) \times YAD \times CSPP(Z) + CY \end{aligned}$$



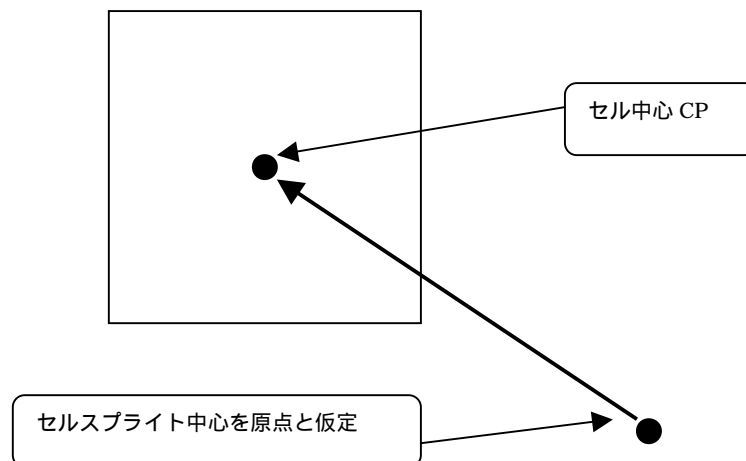
セル中心の座標を原点と仮定すると P0 (P00、P01、P02、P03) の座標はセルサイズ CSX、CSY、セル中心 CENT_X、CENT_Y、セルスプライトのスケール SX、SY、スクリーンまでの距離 DIST より、各セルの座標は

$$\begin{aligned} P00(X) &= -CSX \times SX \times DIST \times CSPP(Z) \times CENT_X \\ P01(X) &= P00(X) \\ P02(X) &= CSX \times SX \times DIST \times CSPP(Z) + P00(X) \\ P03(X) &= P02(X) \end{aligned}$$

$$\begin{aligned} P00(Y) &= -CSY \times SY \times DIST \times CSPP(Z) \times CENT_Y \\ P02(Y) &= P00(Y) \end{aligned}$$

$$P01(Y) = CSY \times SY \times DIST \times CSPP(Z) + P00(Y)$$

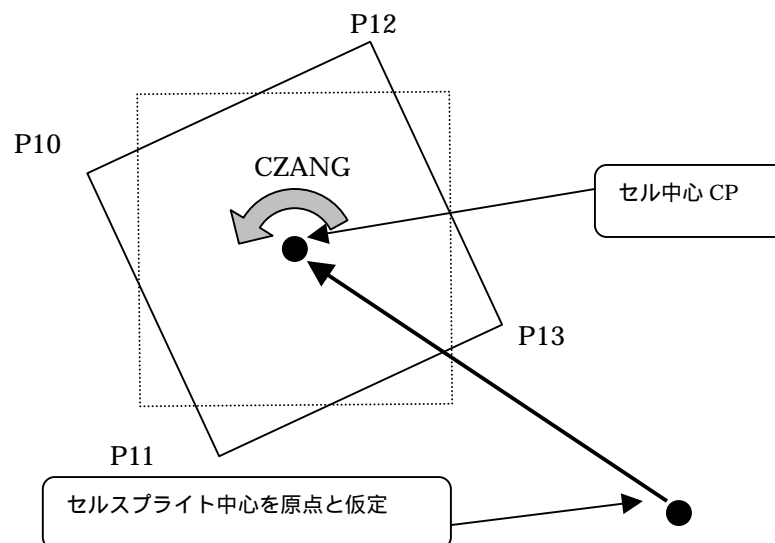
$$P03(Y) = P01(Y)$$



セルスプライト中心の座標を原点と仮定すると、セルスプライト中心からのオフセット COX、COY、セルスプライトのスケール SX、SY よりセル中心 CP は

$$CP(X) = COX \times SX \times DIST \times CSPP(Z)$$

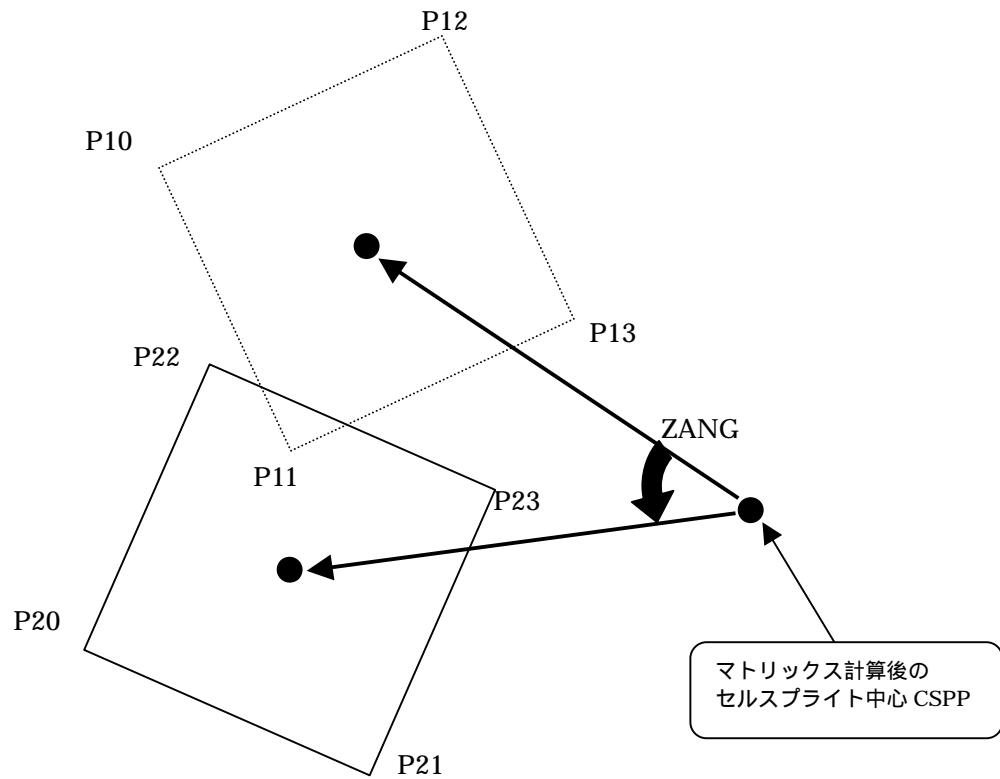
$$CP(Y) = COY \times SY \times DIST \times CSPP(Z)$$



このときセル中心で P0 を CZANG 回転したセルの点を P1 (P10、P11、P12、P13) とすると

$$P1(X) = P0(X) \times \cos(CZANG) - P0(Y) \times \sin(CZANG) + CP(X)$$

$$P1(Y) = P0(X) \times \sin(CZANG) + P0(Y) \times \cos(CZANG) + CP(Y)$$



セルスプライトの中心で ZANG 回転させセルスプライト中心をはじめに計算した CSPP にしたセルの点を P2 (P20、P21、P22、P23) とすると

$$P2(X) = P1(X) \times \cos(ZANG) - P1(Y) \times \sin(ZANG) + CSPP(X)$$

$$P2(Y) = P1(X) \times \sin(ZANG) + P1(Y) \times \cos(ZANG) + CSPP(Y)$$

$$P2(Z) = CSPP(Z)$$

注意 セル頂点のベースカラー、オフセットカラーの計算方法は、セルスプライト 2D と同じです。ライトの計算はありません。

6.3 セルストリームとセルストリームリスト

6.3.1 セルストリーム

セルストリームは、セルスプライトを構成しているセルにアニメーションパターンなど、セルに設定できるデータを時間軸に従って並べたものです。

基本的にユーザーは時間を進めるだけで、セルストリームのデータとおりにセルを描画することができます。セルストリームには、実際に処理の行われるタイムスタンプと変更されるデータがチャンク形式で記述されています。詳しいデータ形式については、セルスプライトの仕様書を参照して下さい。

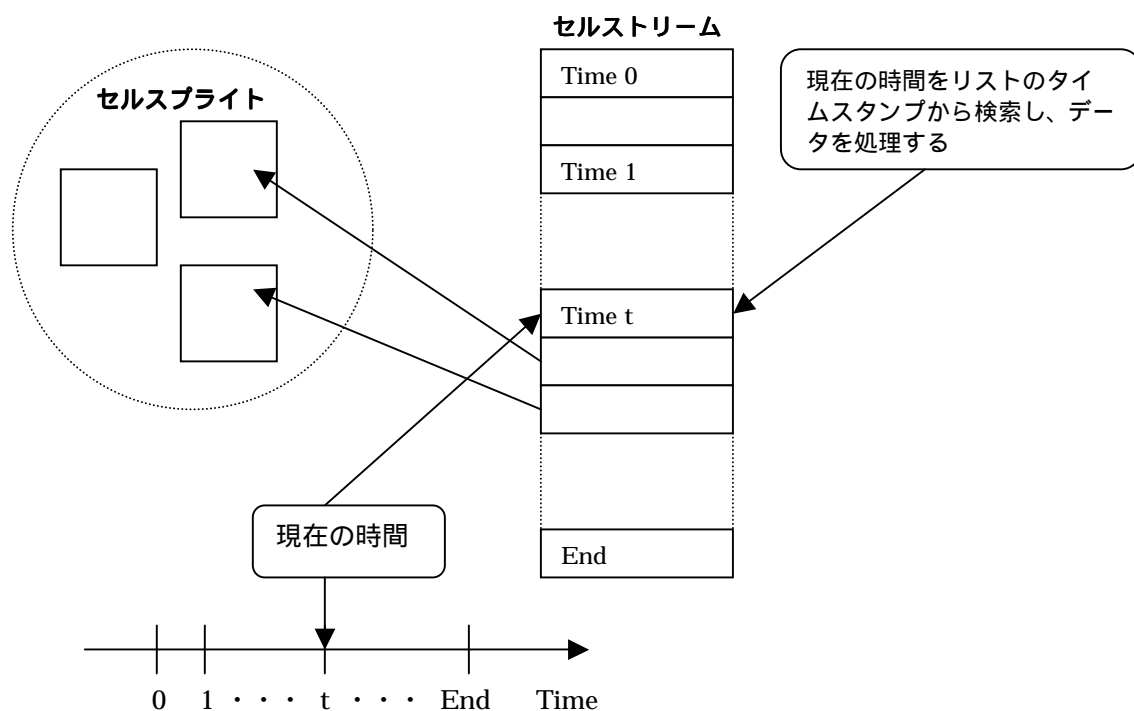


図 6 - 2 セルストリームの構造

6.3.2 セルストリームリスト

実際にセルストリームを使用してセルスプライトを描画するとき、セルストリームをそのまま利用するのではなく、セルストリームをまとめたセルストリームリストを使用します。

セルストリームリストは、複数のセルストリームを1つのセルスプライトに対して実行することができます。例えば、顔のセルスプライトがある場合、1つのセルストリームは右目のアニメーションさせるセルストリームとして作成し、1つを左目、1つを口のように、別々のセルストリームを一括して扱うことができます。こうすることで、左目だけを別のセルストリームに変更したい場合など、データ量が少なく済みます。

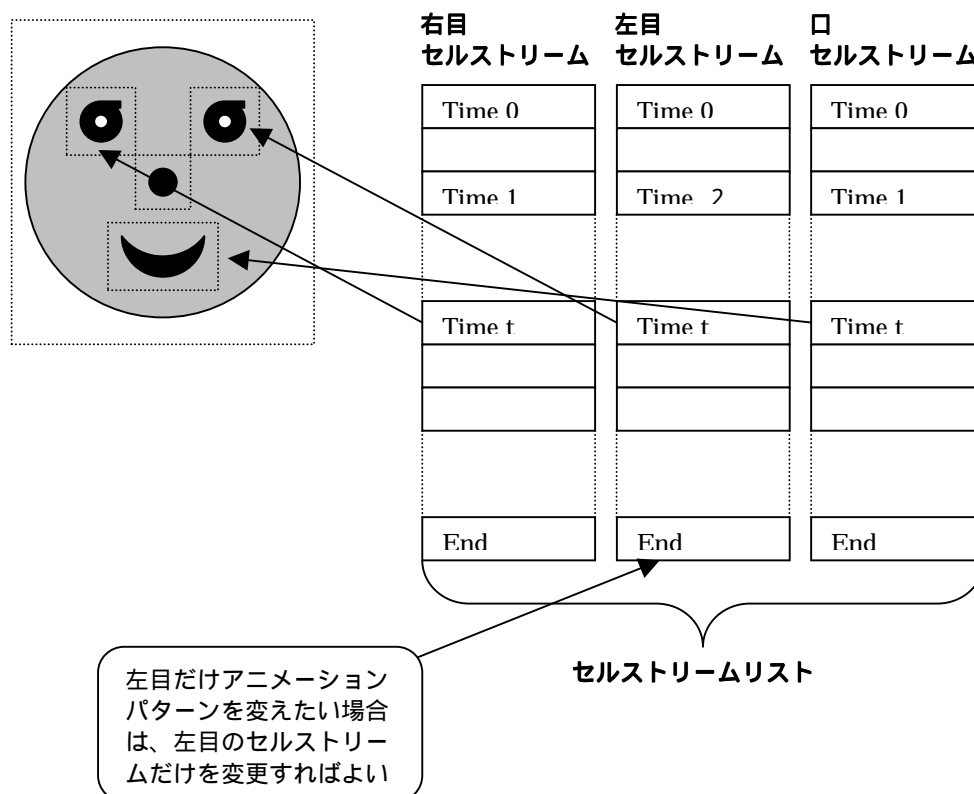
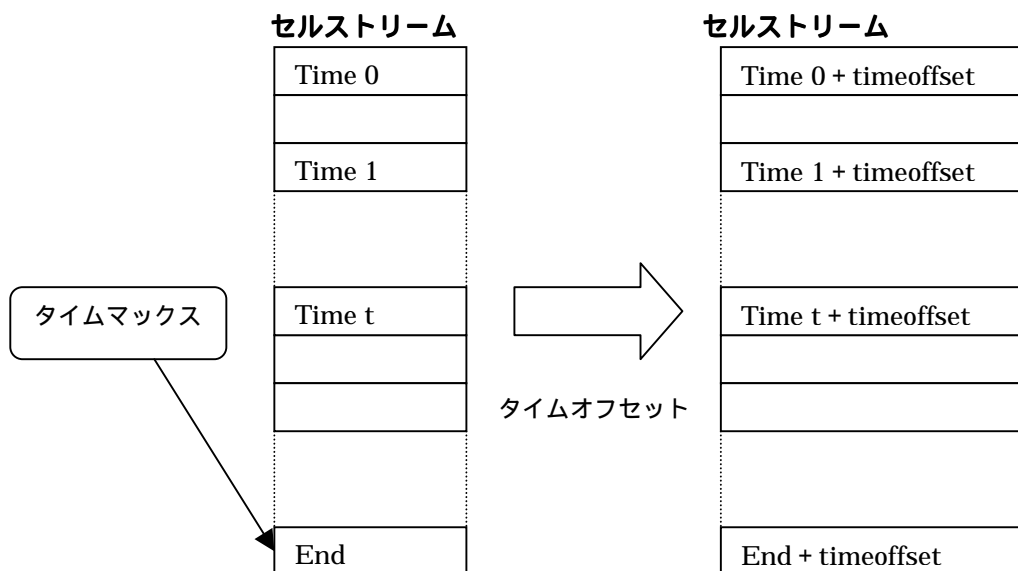


図 6 - 3 セルストリームリストの構造

6.3.3 タイムオフセットとタイムマックス

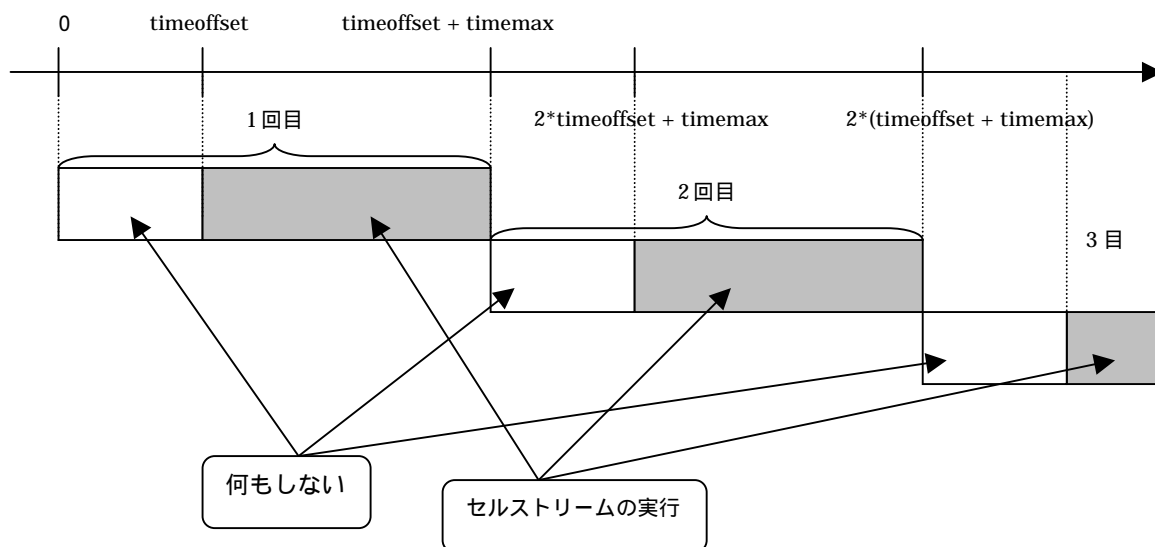
セルストリームにはタイムオフセットとタイムマックスがあります。タイムマックスは、セルストリームのエンドの時間（最大時間）を設定します。これにより、セルストリームには、0 からタイムマックスまでのアニメーションデータがあることになります。タイムオフセットはタイムスタンプ全体をタイムオフセットにより変更することができます。タイムオフセットによりセルストリームにあるデータはタイムオフセットからタイムオフセット+タイムマックスまでになります。



6.3.4 リピート

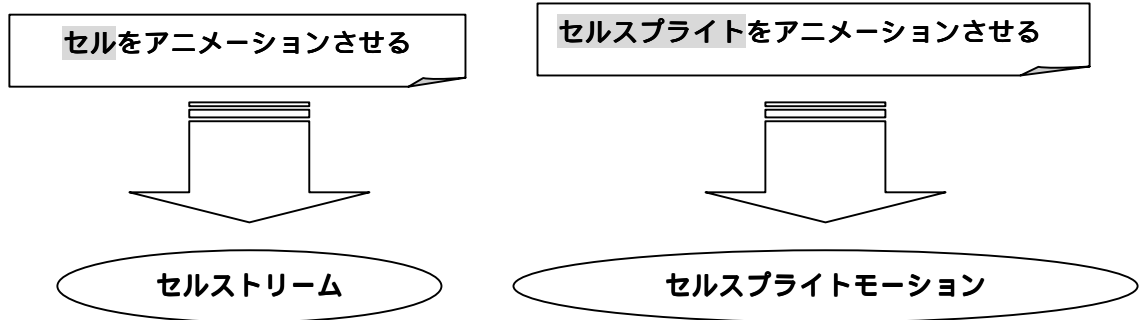
セルのアトリビュートのリピートフラグがある場合、セルストリームのデータをリピートします。セルスプライト全体をリピートさせる場合は、`njSetCellSpriteAttr` 関数でセルスプライトに登録されているセルのアトリビュートを変更することができます。ただし、データは変更しません。

● リピートのセルストリーム実行のイメージ



6.4 セルスプライトモーション

モーションデータを使用して、セルスプライトをモーションさせることができます。モーションには、セルスプライトに設定できるデータ、セルスプライトの中心座標、回転、スケール、ディフューズ、スペキュラーがあります。セルスプライトモーションと同時にセルストリームを使用することができます。



6.5 その他

6.5.1 セルのアトリビュートを一括して変更

表示しているセルスプライトを徐々に透明にしたい場合など一括してセルのアトリビュートを変更したい場合は、`njSetCellSpriteAttr` 関数を使用し変更します。セルのアトリビュートに設定できるものはすべて設定することができます。設定した値は、次に設定されるまで有効です。セルごとに変更することはできません。

6.5.2 複数のセルスプライトのカラーを一括して変更

ゲームシーンの終わりなどで複数のセルスプライトを一括して徐々に透明にしたい場合や、複数のセルスプライト全体を赤くしたい（敵の攻撃を受けた場合など）ときに、セルスプライトにあるディフューズカラーとスペキュラーカラーを設定することができます。設定した値は、次に設定されるまで有効です。

```
/* セルスプライト全体にアルファを掛けます */
NJS_ARGB diff;

diff.a = 0.8f;
diff.a = 1.f;
diff.a = 1.f;
diff.a = 1.f;

/* 設定を変更する */
njSetCellSpriteMaterial( NULL, &diff, NULL, NULL,
NJD_CELLSPRITE_OFFSET_DIFF_MULTI);
njSetCellSpriteAttr( 0xFFFF, NJD_FCA_AL );

njDrawCellStream2D(&streamlist, &motion, step, NULL);

/* 元に戻す */
njSetCellSpriteMaterial( NULL, NULL, NULL, NULL, 0);
njSetCellSpriteAttr( 0xFFFF, 0);
```


7 モデル

ここでは、モデル全般について説明します。

7.1 概要

モデルは、複数のポリゴンで構成された形あるものの基本単位で、Ninja での描画の基本となるものです。

マトリクス変換やライティングを行って、描画を行います。

7.1.1 構造

構造は次のとおりです。

```
typedef struct {  
    Sint32      *vlist;  
    Sint16      *plist;  
    NJS_POINT3  center;  
    Float       r;  
} NJS_CNK_MODEL;
```

(1) vlist

頂点チャンクデータのリストです。

(2) plist

ストリップ、マテリアル、テクスチャなどのチャンクリストです。

(3) center

モデルの外接球の中心です。

(4) r

モデルの外接球の半径です。

7.1.2 処理

処理の基本は、最初にモデルの中心と半径からモデルが画面外かどうかを判断します。

画面内にあれば、頂点データからマトリクス演算、透視変換、ライティングの作業を行い、中間バッファに格納します。

plist には、マテリアルやテクスチャの設定、及びポリゴンリストデータが入ります。

ポリゴンリストデータから中間バッファの頂点データを参照し、ストリップポリゴンを生成します。

7.1.3 関数一覧

- (1) njCnkEasyDraw
平行光源 1 つ使用したファンクションで、インテンシティモードで描画します。中間バッファは 16 バイト使用します。また、表裏判定は行わず裏面も描画します。
- (2) njCnkSimpleDraw
平行光源 1 つ使用したファンクションで、インテンシティモードで描画します。中間バッファは 32 バイト使用します。表裏判定を行い、裏面は描画しません。また、環境マップ、両面ポリゴンをサポートします。
- (3) njCnkEasyMultiDraw
平行光源と点光源を合わせて 6 つと、アンビエント光を 1、合計 7 つの光源を使用できるファンクションで、フローティングモードで描画します。中間バッファは 32 バイト使用します。表裏判定をせずに、裏面も描画し、また頂点カラーとの積算やデプスキューにも対応します。
- (4) njCnkSimpleMultiDraw
平行光源と点光源を合わせて 6 つと、アンビエント光を 1、合計 7 つの光源を使用できるファンクションで、フローティングモードで描画します。中間バッファは 64 バイト使用します。表裏判定を行い、裏面は描画しません。また頂点カラーとの積算やデプスキューにも対応します。環境マップもサポートします。
- (5) njCnkWireDraw
ワイヤーフレームでモデルを描画します。中間バッファは 16 バイト使用します。なお、ワイヤーフレームはポリゴンの 3 角形を描画するのではなく、ストリップをトレースするラインを描画します。
- (6) njCnkToonDraw
アニメ調のレンダリングでモデルを描画します。中間バッファは 32 バイト使用します。表裏判定して描画します。
- (7) njCnkDirectDraw
中間バッファを使用せずにモデルを描画します。モデルデータは専用形式に変換して使用します。環境マップをサポートします。

7.2 ファンクション

7.2.1 CnkEasyDraw

(1) 概要

平行光源を 1 つ使用した、最も描画性能の高いファンクションです。描画はインテンシティーモードで行い、表裏判定をせず、裏面も描画します。中間バッファは、16 バイト単位です。

(2) 光源

a. 概要

光源計算は、インテンシティーのみ計算します。インテンシティーがオーバーした分はスペキュラに回します。ただし、頂点カラーのある頂点形式の場合、光源計算は一切行いません。また、その場合は、バックモードで描画を行います。

b. ファンクション

光源設定のファンクションは次のとおりです。

| ファンクション | 内 容 |
|----------------------------|----------------|
| njCnkSetEasyLight | 平行光源ベクトル設定 |
| njCnkSetEasyLightIntensity | 光源の強さ、アンビエント設定 |
| njCnkSetEasyLightColor | 光源色設定 |

光源のベクトルの設定は、カレントマトリクス変換をした後のベクトルを設定して下さい。

逆に、マトリクス変換しない場合、画面に固定の光源ベクトルが設定できます。

例えば、njCnkSetEasyLight(0.f, 0.f, -1.f); とすると、常に画面正面から光源が当たっていることになります。

(3) 中間バッファ

中間バッファ形式は次のとおりです。

| オフセット | 名 称 | 内 容 |
|-------|-------|------------|
| +0 | x | スクリーン X 座標 |
| +4 | y | スクリーン Y 座標 |
| +8 | ooz | 1.0 / Z |
| +12 | inten | インテンシティー |

頂点カラー形式の場合、D8 の値が入ります。

ooz が負の数のとき、x、y の値は保証されません。

(4) 処理方法

a. 頂点処理

モデルデータの vlist から、各頂点のインテンシティーを計算し中間バッファに格納します。計算式は次のとおりです。

```
DiffuseIntensity = ( 頂点法線・光源ベクトル ) * ( -intensity ) + ambient
If( DiffuseIntensity < ambient ) DiffuseIntensity = ambient
```

intensity,ambient は、njCnkSetEasyLightIntensity でセットした値です。

マトリクスにスケールがかかっている場合、ベクトルをマトリクス演算したときにスケールがかかってしまいます。(正規化は行っていません)

そのため、頂点法線と光源ベクトルの内積結果が違ってきます。その場合は、intensity の値を調整して下さい。具体的にはスケールが2倍の場合、intensity は半分の値にセットすれば計算結果が合います。また、intensity の値を 0 に、ambient の値を 1.0 にすると、光源無視と同様の効果になります。

なお、頂点法線がない形式の場合は、1.0 を設定します。また、頂点カラー形式の場合は、頂点カラーをそのままセットします。頂点カラーの場合、バックモードで描画するため、同じモデルに頂点カラーのないデータと混在することはできません。

b. マテリアル

テクスチャの Diffuse のマテリアルは光源色、Specula のマテリアルはモデルデータのスペキュラの値を使用します。

```
Texture = TextureColor * LightColor * DiffuseIntensity + SpeculaColor * SpeculaIntensity
```

ポリゴンのマテリアルはモデルデータのディフューズと光源色を掛けたものになります。なお、ポリゴンにはスペキュラはありません。

```
Polygon = DiffuseColor * LightColor * DiffuseIntensity
```

マテリアルはインテンシティーモードでのグローバルパラメタにセットします。

スペキュラがある場合は、グローバルパラメタは 64 バイトになります。

c. ポリゴン処理

モデルデータの plist と中間バッファからポリゴンを生成しハードウェアにセットします。その際、DiffuseIntensity - 1.0 をスペキュラの強さとして設定します。

```
SpeculaIntensity = DiffuseIntensity - 1.0
```

なお、Diffuse と Specula の値は Floating の値ですが、ハードウェアはその値を 0.0 ~ 1.0 の値にクリッピングします。つまり、1.0 より大きな値を設定してもハードウェアは 1.0 として扱います。

(5) 頂点形式

サポートしている頂点形式は次のとおりです。

| 頂点形式 | 内 容 |
|----------|---------------------|
| CV_VN | 通常の頂点形式 |
| CV_VN_D8 | 頂点カラー |
| CV_VN_NF | エンベロープ用頂点形式 |
| CV_VN_UF | CV_VN と同様の処理 |
| CV | インテンシティーを 1.0 固定とする |
| CV_D8 | 頂点カラー |
| CV_UF | CV と同様の処理 |

頂点カラー形式の場合、インテンシティーモードではなく、バックカラーモードになります。この場合、スペキュラはなくなります。

(6) フラグ

EasyDraw でサポートしているフラグは次のとおりです。

| フラグ | 内 容 |
|------------|-----------------|
| NJD_FST_IS | スペキュラを無視します |
| NJD_FST_UA | 半透明を有効にします |
| NJD_FST_FL | フラットシェーディングにします |

フラットシェーディングはハードウェアの機能を使用して実現しています。この方法は、ストリップ3角形の3つ目の頂点の値で、その3角形の面の値とします。

7.2.2 CnkSimpleDraw

(1) 概要

平行光源を1つ使用したファンクションで、インテンシティーモードで描画します。表裏判定を行い裏面は描画しないので、より多くのポリゴンが描画できますが、性能は落ちます。また、CnkEasyDrawと比べて、環境マップや両面ポリゴンをサポートしていて、スペキュラの計算もおこないます。中間バッファは32バイト単位です。

(2) 光源

a. 概要

CnkEasyDrawと違い、インテンシティーとスペキュラの計算を行います。アンビエントの処理はポリゴン生成時に行います。

b. ファンクション

光源設定のファンクションは次のとおりです。

| ファンクション | 内 容 |
|------------------------------|----------------|
| njCnkSetSimpleLight | 平行光源ベクトル設定 |
| njCnkSetSimpleLightIntensity | 光源の強さ、アンビエント設定 |
| njCnkSetSimpleLightColor | 光源色設定 |

光源のベクトルの設定は、カレントマトリクス変換をした後のベクトルを設定して下さい。

逆に、マトリクス変換しない場合、画面に固定の光源ベクトルが設定できます。

例えば、njCnkSetSimpleLight(0.f, 0.f, -1.f); とすると、常に画面正面から光源が当たっていることになります。

なお、画角やカメラの設定を変更する場合は、必ず njCnkSetSimpleLight を行って下さい。

(3) 中間バッファ

中間バッファ形式は次のとおりです。

| オフセット | 名 称 | 内 容 |
|-------|-------|-------------|
| +0 | z | Z 座標 |
| +4 | x | スクリーン X 座標 |
| +8 | y | スクリーン Y 座標 |
| +12 | ooz | 1.0 / Z |
| +16 | inten | インテンシティー値 |
| +20 | spec | スペキュラ値 |
| +24 | nx | 頂点法線 X ベクトル |
| +28 | ny | 頂点法線 Y ベクトル |

頂点カラー形式の場合、D8 の値が入ります。

法線ベクトルは環境マップのときに使用します。

ooz が負の数のとき、x、の値は保証されません。

(4) 処理方法

a. 頂点処理

モデルデータの vlist から、各頂点のインテンシティー及びスペキュラを計算し中間バッファに格納します。計算式は次のとおりです。

```
DiffuseIntensity = (頂点法線・光源ベクトル) * (-intensity)
SpeculaIntensity = DiffuseIntensity ^ 17
```

intensity は、njCnkSetSimpleLightIntensity でセットした値です。

マトリクスにスケールがかかっている場合、ベクトルをマトリクス演算したときにスケールがかかってしまいます。(正規化は行っていません)

そのため、頂点法線と光源ベクトルの内積結果が違ってきます。その場合は、intensity の値を調整して下さい。具体的にはスケールが 2 倍の場合、intensity は半分の値にセットすれば計算結果が合います。

スペキュラはディフューズの 17 乗で、この値は固定です。17 乗することにより、スペキュラの収束が表現できます。

CnkEasyDraw と違い、アンビエントでのリミットチェックはしていません。

b. マテリアル

テクスチャの Diffuse のマテリアルはモデルのディフューズ値と光源色を掛けたもの、Specula のマテリアルはモデルデータのスペキュラの値を使用します。

```
Texture = TextureColor * DiffuseColor * LightColor * DiffuseIntensity + SpeculaColor * SpeculaIntensity
```

NJD_CONTROL_3D_CONSTANT_TEXTURE_MATERIAL が設定されている場合、Diffuse のマテリアルは無視します。つまり、CnkEasyDraw と同様になります。

ポリゴンのマテリアルはモデルデータのディフューズと光源色を掛けたものになります。なお、ポリゴンにはスペキュラはありません。

```
Polygon = DiffuseColor * LightColor * DiffuseIntensity
```

c. ポリゴン処理

モデルデータの plist と中間バッファからポリゴンを生成しハードウェアにセットします。その際、DiffuseIntensity に ambient を加算します。

```
DiffuseIntensity += ambient
if(DiffuseIntensity < ambient) DiffuseIntensity = ambient
ambient は njCnkSetSimpleLightIntensity でセットした値。
```

ポリゴン生成時に処理を行うことにより、両面、アンビエント無視、光源無視の処理を行います。両面処理の場合はインテンシティーを符号反転します。光源無視はアンビエントの値を 1.0 にします。また、環境マップのときは、データの UV は使用せずに中間バッファの法線データ nx と ny から UV を生成します。

なお、Diffuse と Specula の値は Floating の値ですが、ハードウェアはその値を 0.0 ~ 1.0 の値にクリッピングします。つまり、1.0 より大きな値を設定してもハードウェアは 1.0 として扱います。

(5) 頂点形式

SimpleDraw でサポートしている頂点形式は次のとおりです。

| 頂点形式 | 内 容 |
|----------|---------------------|
| CV_VN | 通常の頂点形式 |
| CV_VN_D8 | 頂点カラー |
| CV_VN_NF | エンベロープ用頂点形式 |
| CV_VN_UF | CV_VN と同様の処理 |
| CV | インテンシティーを 1.0 固定とする |
| CV_D8 | 頂点カラー |
| CV_UF | CV と同様の処理 |

頂点カラー形式の場合、インテンシティーモードではなく、パックカラーモードになります。この場合スペキュラはなくなります。

(6) フラグ

SimpleDraw でサポートしているフラグは、次のとおりです。

| フラグ | 内 容 |
|-------------|------------------|
| NJD_FST_IS | スペキュラを無視します。 |
| NJD_FST_UA | 半透明を有効にします。 |
| NJD_FST_FL | フラットシェーディングにします。 |
| NJD_FST_ENV | 環境マッピングにします。 |
| NJD_FST_IL | 光源を無視します。 |
| NJD_FST_IA | アンビエントを無視します |
| NJD_FST_DB | 両面ポリゴンにします。 |

フラットシェーディングは、ハードウェアの機能を使用して実現しています。この方法は、ストリップ 3 角形の 3 つ目の頂点の値で、その 3 角形の面の値とします。

光源無視はアンビエントの値を 1.0 にしています。

7.2.3 CnkEasyMultiDraw

(1) 概要

平行光源と点光源を合わせて 6 つと、アンビエント光を 1、合計 7 つの光源を使用できるファンクションで、フローティングモードで描画します。表裏判定をせずに、裏面も描画し、また頂点カラーとの積算やデプスキューにも対応します。

(2) 光源

a. 概要

光源計算は、R、G、B それぞれを Floating で計算します。各光源に対して計算した結果を加算して、その頂点での光源計算結果とします。

頂点カラー形式の場合、光源計算結果と頂点カラーを積算します。

アンビエントカラーは、ポリゴン生成時に加算します。

光源計算結果がオーバーした分はスペキュラ成分に回します。

b. ファンクション

光源設定のファンクションは、次のとおりです。

| ファンクション | 内 容 |
|--------------------------------|-------------------|
| njCnkSetEasyMultiLight | 光源数の設定 |
| njCnkSetEasyMultiLightSwitch | 光源のオンオフ |
| njCnkSetEasyMultiAmbient | アンビエントカラーの設定 |
| njCnkSetEasyMultiLightColor | ライトカラーの設定 |
| njCnkSetEasyMultiLightVector | 平行光源ベクトル設定 |
| njCnkSetEasyMultiLightVectorEx | 平行光源ベクトル設定（ライト選択） |
| njCnkSetEasyMultiLightPoint | 点光源座標設定 |
| njCnkSetEasyMultiLightRange | 点光源範囲設定 |
| njCnkSetEasyMultiLightMatrices | ライトマトリックス設定 |

光源数にはアンビエントライトは含みません。

点光源範囲は、nRange、fRange で設定します。光源までの距離が fRange より遠い場合その光源は無視します。また、nRang より近い場合は距離による影響は無視します。その間にある場合、 $nRange^2 / r^2$ で減衰します。

(3) 中間バッファ

中間バッファ形式は次のとおりです。

| オフセット | 名 称 | 内 容 |
|-------|-----|------------|
| +0 | z | Z 座標 |
| +4 | x | スクリーン X 座標 |
| +8 | y | スクリーン Y 座標 |
| +12 | ooz | 1.0 / Z |
| +16 | a | アルファ値 |
| +20 | r | 赤 |
| +24 | g | 緑 |
| +28 | b | 青 |

アルファ値は、デプスキューの時に設定されます。通常は 1.0 固定です。
ooz が負の数のとき、x,y の値は保証されません。

(4) 処理方式

a. 頂点処理

モデルデータの vlist と光源の設定から、光源計算をして中間バッファに格納します。光源が複数ある場合、それぞれの計算結果を加算します。

計算式は次のとおりです。

● 平行光源

$$\text{光源色} = \text{平行光源色} * (- (\text{頂点法線} \cdot \text{光源ベクトル}))$$

R、G、B それぞれにおいて、計算を行います。
内積結果が 0 以下の（90 度以上はなれている）場合、光源は無視します。

● 点光源

$$\text{光源色} = \text{点光源色} * (- (\text{頂点法線} \cdot \text{光源ベクトル})) * nRange^2 / r^2$$

R、G、B それぞれにおいて、計算を行います

内積結果が 0 以下の (90 度以上はなれている) 場合、及び、 r (距離) が $fRange$ より遠い場合は光源を無視します。

また、 r (距離) が $nRange$ より近い場合は距離成分を無視します。つまり、内積結果のみに影響します。

なお、 $nRange$ 、 $fRange$ は `njCnkSetEasyMultiLightRange` でセットした値です。

光源色に負の値を設定すると、暗くなるライト (ダークライト) が設定できます。

- デプスキュー

Z 値に応じてアルファ値を設定します。

- 頂点カラー

頂点カラーがある場合、光源計算結果と頂点カラーを積算します。頂点カラーは ARGB 32 ビットパック形式ですが、それを分解してそれぞれ 8 ビットのデータにします。結果は 0×255 の整数になりますが、それを 128.0 で割った値をそれぞれの頂点カラーとします。その結果と、光源計算結果を積算して中間バッファに格納します。

頂点カラーが 0x80808080 を境に、それより小さいときは暗く、大きいときは明るくなります。

- b. マテリアル

- テクスチャ

$$\text{TextureColor} = \text{Texture} * \text{DiffuseColor} + \text{SpeculaColor}$$

テクスチャでは、アンビエントマテリアルのみ影響します。その他は、すべて光源に依存します。フローティングモードで描画するため、グローバルデータにはマテリアルはありません。

- ポリゴン

$$\text{PolygonColor} = \text{DiffuseColor}$$

ポリゴンは、アンビエントマテリアルとディフューズマテリアルが影響します。

- c. ポリゴン処理

モデルデータの plist と中間バッファからポリゴンを生成し、ハードウェアにセットします。その際、アンビエントカラーを加算します。

- テクスチャ

$$\begin{aligned} \text{DiffuseColor} &= \text{AmbientColor} * \text{AmbientMaterial} + \text{光源色} \\ \text{SpeculaColor} &= \text{DiffuseColor} - 1.0 \end{aligned}$$

R、G、B それぞれにおいて計算します。

中間バッファ上の光源色は、必ず 0 以上になっています。

なお、Diffuse と Specula の値は Floating の値ですが、ハードウェアはその値を 0.0 ~ 1.0 の値にカリングします。つまり、1.0 より大きな値を設定してもハードウェアは 1.0 として扱います。

- ポリゴン

$$\text{DiffuseColor} = \text{AmbientColor} * \text{AmbientMaterial} + \text{光源色} * \text{DiffuseMaterial}$$

ポリゴンには、スペキュラはありません。

(5) 頂点形式

サポートしている頂点形式は次のとおりです。

| 頂点形式 | 内 容 |
|----------|-------------|
| CV_VN | 通常の頂点形式 |
| CV_VN_D8 | 頂点カラー |
| CV_VN_NF | エンベロープ用頂点形式 |

(6) フラグ

サポートしているフラグは次のとおりです。

| フラグ | 内 容 |
|------------|------------------|
| NJD_FST_IS | スペキュラを無視します。 |
| NJD_FST_UA | 半透明を有効にします。 |
| NJD_FST_FL | フラットシェーディングにします。 |
| NJD_FST_IA | アンビエントを無視します。 |

フラットシェーディングは、ハードウェアの機能を使用して実現しています。この方法は、ストリップ3角形の3つ目の頂点の値で、その3角形の面の値とします。

アンビエント無視は、アンビエントカラーを黒として計算します。

7.2.4 CnkSimpleMultiDraw

(1) 概要

平行光源と点光源を合わせて6つと、アンビエント光を1つ、合計7つの光源を使用できるファンクションで、フローティングモードで描画します。表裏判定行い、裏面は描画しません。また頂点カラーとの積算やデプスキューにも対応します。環境マップもサポートします。

(2) 光源

a. 概要

光源計算は R、G、B それぞれを Floating で計算します。各光源に対して計算した結果を加算して、その頂点での光源計算結果とします。

頂点カラー形式の場合、光源計算結果と頂点カラーを積算します。

アンビエントカラーは、ポリゴン生成時に加算します。

光源計算結果がオーバーした分はスペキュラ成分に回します。

b. ファンクション

光源設定のファンクションは、次のとおりです。

| ファンクション | 内 容 |
|----------------------------------|-------------------|
| njCnkSetSimpleMultiLight | 光源数の設定 |
| njCnkSetSimpleMultiLightSwitch | 光源のオンオフ |
| njCnkSetSimpleMultiAmbient | アンビエントカラーの設定 |
| njCnkSetSimpleMultiLightColor | ライトカラーの設定 |
| njCnkSetSimpleMultiLightVector | 平行光源ベクトル設定 |
| njCnkSetSimpleMultiLightVectorEx | 平行光源ベクトル設定（ライト指定） |
| njCnkSetSimpleMultiLightPoint | 点光源座標設定 |
| njCnkSetSimpleMultiLightRange | 点光源範囲設定 |
| njCnkSetSimpleMultiLightMatrices | ライトマトリックス設定 |

光源数には、アンビエントライトは含みません。

点光源範囲は、nRange, fRange で設定します。光源までの距離が fRange より遠い場合その光源は無視します。また、nRange より近い場合は距離による影響は無視します。その間にある場合、 $nRange^2 / r^2$ で減衰します。

(3) 中間バッファ

中間バッファ形式は次のとおりです。

| オフセット | 名 称 | 内 容 |
|-------|-----|------------|
| +0 | z | Z 座標 |
| +4 | sx | スクリーン X 座標 |
| +8 | sy | スクリーン Y 座標 |
| +12 | ooz | 1.0 / Z |
| +16 | a | アルファ値 |
| +20 | r | 赤 |
| +24 | g | 緑 |
| +28 | b | 青 |
| +32 | x | X 座標 |
| +36 | y | Y 座標 |
| +40 | nx | 法線 X |
| +44 | ny | 法線 Y |
| +48 | nz | 法線 Z |
| +52 | sr | スペキュラ 赤 |
| +56 | sg | スペキュラ 緑 |
| +60 | sb | スペキュラ 青 |

アルファ値は、デプスキューの時に設定されます。通常は 1.0 固定です。

ooz が負の数的时候、x,y の値は保証されません。

(4) 処理方式

a. 頂点処理

モデルデータの vlist と光源の設定から、光源計算をして中間バッファに格納します。光源が複数ある場合、それぞれの計算結果を加算します。

また、スペキュラ成分も頂点処理で計算します。

計算式は次のとおりです。

● 平行光源

$$\text{光源色} = \text{平行光源色} * (- (\text{頂点法線} \cdot \text{光源ベクトル}))$$

R、G、B それぞれにおいて、計算を行います。

内積結果が 0 以下の (90 度以上はなれている) 場合、光源は無視します。

● 点光源

$$\text{光源色} = \text{点光源色} * (- (\text{頂点法線} \cdot \text{光源ベクトル})) * \text{nRange}^2 / r^2$$

R、G、B それぞれにおいて、計算を行います。

内積結果が 0 以下の (90 度以上はなれている) 場合、または r (距離) が fRange より遠い場合は、光源を無視します。

また、r (距離) が nRange より近い場合は距離成分を無視します。つまり、内積結果のみに影響します。

なお、nRange,fRange は njCnkSetSimpleMultiLightRange でセットした値です。

- スペキュラ

$$\text{スペキュラ} = \text{光源色} - 1.0$$

スペキュラの値は、光源計算結果から 1.0 を引いた値にします。

光源色に負の値を設定すると、暗くなるライト（ダークライト）が設定できます。

- デプスキュー

Z 値に応じてアルファ値を設定します。

- 頂点カラー

頂点カラーがある場合、光源計算結果と頂点カラーを積算します。頂点カラーは ARGB 32 ビットパック形式ですが、それを分解してそれぞれ 8 ビットのデータにします。結果は 0 x 255 の整数になりますが、それを 128.0 で割った値をそれぞれの頂点カラーとします。その結果と、光源計算結果を積算して中間バッファに格納します。

頂点カラーが 0x80808080 を境に、それより小さいときは暗く、大きいときは明るくなります。

- b. マテリアル

- テクスチャ

$$\text{TextureColor} = \text{Texture} * \text{DiffuseColor} + \text{SpeculaColor}$$

テクスチャでは、アンビエントマテリアルとディヒューズマテリアルに影響します。スペキュラマテリアルは影響しません。

フローティングモードで描画するため、グローバルデータにはマテリアルはありません。

- ポリゴン

$$\text{PolygonColor} = \text{DiffuseColor}$$

ポリゴンは、アンビエントマテリアルとディフューズマテリアルに影響します。

- c. ポリゴン処理

モデルデータの plist と中間バッファから、ポリゴンを生成しハードウェアにセットします。その際、アンビエントカラーを加算します。

- テクスチャ

$$\begin{aligned} \text{DiffuseColor} &= \text{AmbientColor} * \text{AmbientMaterial} + \text{光源色} * \text{DiffuseMaterial} \\ \text{SpeculaColor} &= \text{スペキュラ} \end{aligned}$$

R、G、B それぞれにおいて計算します。

なお、Diffuse と Specula の値は Floating の値ですが、ハードウェアはその値を 0.0 ~ 1.0 の値にカリングします。

つまり、1.0 より大きな値を設定してもハードウェアは 1.0 として扱います。

NJD_CONTROL_3D_CONSTANT_TEXTURE_MATERIAL が設定されている場合、Diffuse のマテリアルは無視します。つまり、CnkEasyMultiDraw と同様になります。

● ポリゴン

```
DiffuseColor = AmbientColor*AmbientMaterial + 光源色*DiffuseMaterial
```

ポリゴンにはスペキュラはありません。

(5) 頂点形式

サポートしている頂点形式は、次のとおりです。

| 頂点形式 | 内 容 |
|----------|-------------|
| CV_VN | 通常の頂点形式 |
| CV_VN_D8 | 頂点カラー |
| CV_VN_NF | エンベロープ用頂点形式 |

(6) フラグ

サポートしているフラグは、次のとおりです。

| フラグ | 内 容 |
|-------------|------------------|
| NJD_FST_IS | スペキュラを無視します。 |
| NJD_FST_UA | 半透明を有効にします。 |
| NJD_FST_FL | フラットシェーディングにします。 |
| NJD_FST_IA | アンビエントを無視します。 |
| NJD_FST_ENV | 環境マッピングにします |

フラットシェーディングは、ハードウェアの機能を使用して実現しています。この方法は、ストリップ3角形の3つ目の頂点の値で、その3角形の面の値とします。

アンビエント無視は、アンビエントカラーを黒として計算します。

7.2.5 CnkWireDraw

(1) 概要

ワイヤーフレームでモデルを描画します。ストリップをつないだラインで描画するため、3角形になりません。また、描画色はマテリアルの色になります。

(2) 光源

ワイヤーフレームには、光源設定はありません。

(3) 中間バッファ

中間バッファ形式は、次のとおりです。

| オフセット | 名 称 | 内 容 |
|-------|-----|------------|
| +0 | x | スクリーン X 座標 |
| +4 | y | スクリーン Y 座標 |
| +8 | z | Z |
| +12 | ooz | 1.0 / Z |

ooz が負の数のとき、x,y の値は保証されません。

(4) 処理方式

a. 頂点処理

マトリクス演算と透視変換を行い、中間バッファに格納します。

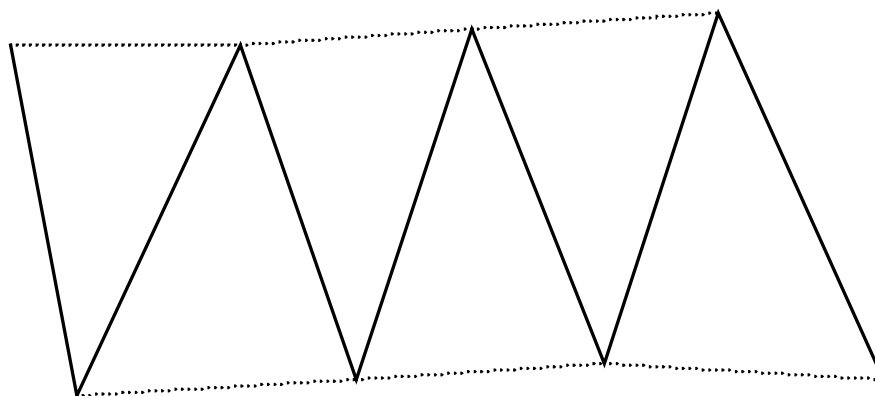
b. マテリアル

ラインの描画色は、ディフューズマテリアルの色のものは無視されます。

Quad ポリゴンのグローバルにマテリアルをセットします。

c. ポリゴン処理

ライン描画は、Quad ポリゴンで描画します。ストリップデータを繋ぐラインとなりますので、ポリゴンの三角形を描画するわけではありません。



また、ラインの幅は、線分の角度を計算して描画します。

1 ライン 1 Quad ポリゴンで描画します。64 バイトデータになります。

(5) 頂点形式

サポートしている頂点形式は次のとおりです。

| 頂点形式 | 内 容 |
|----------|-------------|
| CV_VN | 通常の頂点形式 |
| CV_VN_NF | エンベロープ用頂点形式 |
| CV | 法線なし |

(6) フラグ

フラグは、すべて無効となります。

7.2.6 CnkToonDraw

(1) 概要

モデルをアニメーション風の2値シェーディングで描画します。シェーディングの境目は光源の設定により変化させることができます。中間バッファは、32バイトで表裏判定を行います。

(2) 光源

a. 概要

ToonDrawの光源計算は、影のエリアを算出するためのものです。光源ベクトルと光源の強さにより、影のエリアを算出します。表の色と影の色は、専用の関数で設定します。

b. ファンクション

光源設定のファンクションは次のとおりです。

| ファンクション | 内 容 |
|----------------------------|----------------|
| njCnkSetToonLight | 平行光源ベクトル設定 |
| njCnkSetToonLightIntensity | 光源の強さ、アンビエント設定 |
| njCnkSetToonShade | 光源色設定 |

光源のベクトルの設定は、カレントマトリクス変換をした後のベクトルを設定して下さい。

逆に、マトリクス変換しない場合、画面に固定の光源ベクトルが設定できます。

例えば、njCnkSetToonLight (0.f, 0.f, -1.f); とすると、常に画面正面から光源が当たっていることとなります。

(3) 中間バッファ

中間バッファ形式は次のとおりです。

| オフセット | 名 称 | 内 容 |
|-------|-------|------------|
| +0 | inten | 表、影判定 |
| +4 | sx | スクリーン X 座標 |
| +8 | sy | スクリーン Y 座標 |
| +12 | ooz | 1.0 / Z |
| +16 | shade | シェーディング値 |
| +20 | x | 3 D、X 座標 |
| +24 | y | 3 D、Y 座標 |
| +28 | z | 3 D、Z 座標 |

inten は、その頂点が表の場合 1、影の場合 0 が入ります。

ooz が負の数 のとき、sx,sy の値は保証されません。

(4) 処理方式

a. 頂点処理

モデルデータの vlist から、各頂点のインテンシティーを計算し中間バッファに格納します。計算式は次のとおりです。

```
shade = ( 頂点法線・光源ベクトル ) * (-intensity) + ambient
If( shade > 0 ) inten = 1 else inten = 0
```

intensity、ambient は、njCnkSetToonLightIntensity でセットした値。

その頂点が表か影になるかは、shade の値で判断します。判断した結果を inten に格納します。shade の値は、ポリゴンの分割の計算に使用します。

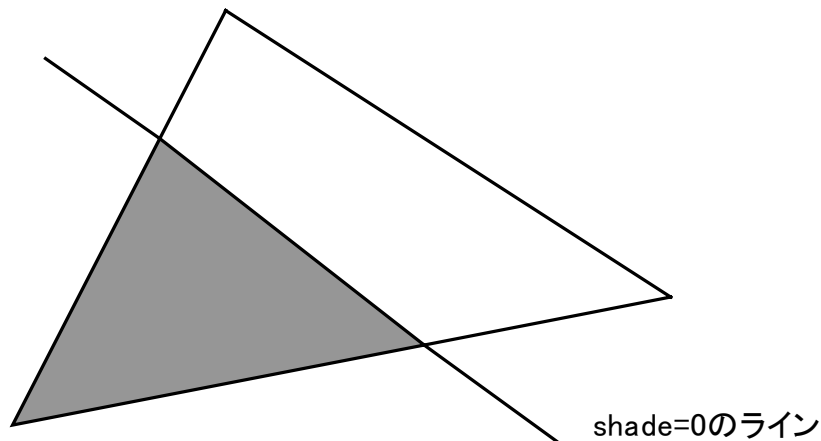
shade の値が 0 以上かどうかで判断しますので、intensity、ambient の値を調整して影のエリアの範囲の調整を行います。

b. マテリアル

マテリアルは、njCnkSetToonShade で設定したカラーとディフューズカラーの積で描画します。ポリゴンはバックカラー形式になります。

c. ポリゴン処理

ポリゴンの生成時に、表と影のエリアにわかれたポリゴンは自動的に分割して描画します。分割のラインは shade=0 のところを基準として分割します。



なお、ToonDraw はニアクリッピングによる、ポリゴン分割を行いません。ポリゴン単位でクリッピングします。

(5) 頂点形式

サポートしている頂点形式は、次のとおりです。

| 頂点形式 | 内 容 |
|----------|-------------|
| CV_VN | 通常の頂点形式 |
| CV_VN_NF | エンベロープ用頂点形式 |
| CV | 法線なし |

(6) フラグ

ToonDraw でサポートしているフラグは次のとおりです。

| フラグ | 内 容 |
|------------|------------|
| NJD_FST_IL | 光源を無視します |
| NJD_FST_UA | 半透明を有効にします |

7.2.7 CnkDirectDraw

(1) 概要

DirectDraw は、中間バッファを使用せずに頂点データの演算とポリゴンの生成を同時に行います。データ形式はチャンクデータをあらかじめ、DirectDraw 用にコンバートします。機能的には EasyDraw と同様ですが、環境マップをサポートします。

描画は頂点カラーモードと通常モードがあります。通常モードは、EasyDraw と同様の光源計算を行うモードで、頂点カラーモードは光源計算をせずに、そのまま頂点カラーで描画します。頂点データ形式によりモードが変わります。

データをコンバートする際にライトを設定して頂点カラーモードにすることも可能です。

(2) 光源

a. 概要

計算方法は、EasyDraw と同様になります。

b. ファンクション

光源設定のファンクションは、次のとおりです。

| ファンクション | 内 容 |
|------------------------------|----------------|
| njCnkSetDirectLight | 平行光源ベクトル設定 |
| njCnkSetDirectLightIntensity | 光源の強さ、アンビエント設定 |
| njCnkSetDirectLightColor | 光源色設定 |

(3) 中間バッファ

中間バッファは使用しません。

(4) 処理方式

a. コンパイル

DirectDraw で描画する場合は、あらかじめモデルデータをコンパイルしなければなりません。コンパイルの内容は、実際にハードウェアに設定する値と同じものをデータとして生成するものです。テクスチャのアドレス等の設定も行いますので、あらかじめテクスチャのセットを行わなければなりません。

また、頂点データは、ポリゴンリストから、頂点リストのデータをリードし、それを加工してセットします。そのため頂点の共有がある場合、データサイズが大きくなります。データサイズはあらかじめチェックすることができます。また、データのアライメントは 32 バイト単位にしてください。

頂点データ形式が D 8 の場合、頂点カラーモードになります。また、コンパイル時にライトを設定すれば、頂点カラーモードになります。

| ファンクション | 内 容 |
|-----------------------------------|-----------------|
| njCnkDirectObjectCompile | データのコンパイル |
| njCnkDirectObjectCompileLight | ライト付でコンパイル |
| njCnkDirectObjectCompileSize | コンパイルサイズの取得 |
| njCnkDirectObjectCompileLightSize | ライト付コンパイルサイズの取得 |

なお、コンパイルすると次のデータのポインタを返しますので、そのアドレスから続けてコンパイルできます。

b. 描画

描画は中間バッファを使用せず、頂点データのマトリクス変換、透視変換をした結果をそのままハードウェアに設定します。

頂点カラー形式の場合は光源計算を行わず、バックカラーモードで描画し、通常モードの場合、EasyDraw と同様の光源計算を行います。ただし、EasyDraw と違い環境マッピングもサポートします。

(5) 頂点形式

コンパイル時に使用できる頂点形式は次のとおりです。

| 頂点形式 | 内 容 |
|----------|----------------|
| CV_VN | 通常モードでコンパイル |
| CV_VN_D8 | 頂点カラーモードでコンパイル |
| CV_D8 | 頂点カラーモードでコンパイル |

(6) フラグ

DirectDraw でサポートするフラグは、次のとおりです。

| フラグ | 内 容 |
|-------------|-----------------|
| NJD_FST_IS | スペキュラを無視します |
| NJD_FST_UA | 半透明を有効にします |
| NJD_FST_FL | フラットシェーディングにします |
| NJD_FST_ENV | 環境マップを有効にします |

フラットシェーディングは、ハードウェアの機能を使用して実現しています。この方法は、ストリップ 3 角形の 3 つ目の頂点の値で、その 3 角形の面の値とします。

7.3 中間バッファ

7.3.1 CnkEasy

| オフセット | 名 称 | 内 容 |
|-------|-------|------------|
| +0 | x | スクリーン X 座標 |
| +4 | y | スクリーン Y 座標 |
| +8 | ooz | 1.0 / Z |
| +12 | inten | インテンシティー |

ooz が負のときは、視点の後ろに頂点がある場合ですので、その時のスクリーン座標は意味のない値になります。inten には光源計算結果が入りますが、頂点カラーのある形式の場合は、頂点カラーがそのまま入ります。

7.3.2 CnkSimple

| オフセット | 名 称 | 内 容 |
|-------|-------|-------------|
| +0 | z | Z 座標 |
| +4 | x | スクリーン X 座標 |
| +8 | y | スクリーン Y 座標 |
| +12 | ooz | 1.0 / Z |
| +16 | inten | インテンシティー |
| +20 | spec | スペキュラ値 |
| +24 | nx | 頂点法線 X ベクトル |
| +28 | ny | 頂点法線 Y ベクトル |

ooz が負のときは、視点の後ろに頂点がある場合ですので、その時のスクリーン座標は意味のない値になります。inten には光源計算結果が入りますが、頂点カラーのある形式の場合は、頂点カラーがそのまま入ります。

頂点法線は、環境マップの時に使用します。

7.3.3 CnkEasyMulti

| オフセット | 名 称 | 内 容 |
|-------|-----|------------|
| +0 | z | Z 座標 |
| +4 | x | スクリーン X 座標 |
| +8 | y | スクリーン Y 座標 |
| +12 | ooz | 1.0 / Z |
| +16 | a | アルファ値 |
| +20 | r | 赤 |
| +24 | g | 緑 |
| +28 | b | 青 |

ooz が負のときは、視点の後ろに頂点がある場合ですので、その時のスクリーン座標は意味のない値になります。アルファ値は、デプスキューの時に意味を持ちます。

7.3.4 CnkSimpleMulti

| オフセット | 名 称 | 内 容 |
|-------|-----|------------|
| +0 | z | Z 座標 |
| +4 | sx | スクリーン X 座標 |
| +8 | sy | スクリーン Y 座標 |
| +12 | ooz | 1.0 / Z |
| +16 | a | アルファ値 |
| +20 | r | 赤 |
| +24 | g | 緑 |
| +28 | b | 青 |
| +32 | x | X 座標 |
| +36 | y | Y 座標 |
| +40 | nx | 法線 X |
| +44 | ny | 法線 Y |
| +48 | nz | 法線 Z |
| +52 | sr | スペキュラ 赤 |
| +56 | sg | スペキュラ 緑 |
| +60 | sb | スペキュラ 青 |

ooz が負のときは、視点の後ろに頂点がある場合ですので、その時のスクリーン座標は意味のない値になります。アルファ値は、デプスキューの時に意味を持ちます。

スペキュラの値は、1.0 を引いた値が入っています。

7.3.5 CnkWireDraw

| オフセット | 名 称 | 内 容 |
|-------|-----|------------|
| +0 | x | スクリーン X 座標 |
| +4 | y | スクリーン Y 座標 |
| +8 | z | Z |
| +12 | ooz | 1.0 / Z |

7.3.6 CnkToonDraw

| オフセット | 名 称 | 内 容 |
|-------|-------|------------|
| +0 | inten | 表、影判定 |
| +4 | sx | スクリーン X 座標 |
| +8 | sy | スクリーン Y 座標 |
| +12 | ooz | 1.0 / Z |
| +16 | shade | シェーディング値 |
| +20 | x | 3 D、X 座標 |
| +24 | y | 3 D、Y 座標 |
| +28 | z | 3 D、Z 座標 |

inten は、その頂点が表の場合 1、影の場合 0 が入ります。

7.4 フォーマット

7.4.1 VLIST

各頂点チャンクには、8 バイトのヘッダがついています。

ヘッダの形式は次のとおりです。

| オフセット | 名 称 | 内 容 |
|-------|--------|----------------|
| +0 | type | チャンクの種類、及びフラグ |
| +2 | size | チャンクのサイズ |
| +4 | offset | 中間バッファのオフセット番号 |
| +6 | num | 頂点数 |

(1) type

下位 8 ビットにチャンクの種類、上位 8 ビットにフラグがセットされます。

頂点演算する際に、その頂点がクリッピングされるかチェックしています。頂点数とクリッピングされた頂点数が同じ場合、すべての頂点がクリッピングされたと判断して、次の plist の処理を行いません。ただし、フラグに NJD_FV_CONT がセットされている場合、前回の vlist の処理と積算してクリッピング判断を行います。これは、親との間接つなぎなどに使用します。

(2) size

そのチャンクのデータサイズを、4 バイト単でセットします。ヘッダの offset,num のエリアも含みます。

(3) offset

演算結果を格納する中間バッファの先頭番号を指定します。通常はオフセット値 0 ですが、中間バッファの位置をずらすことにより複数の結果を格納できます。これは、間接つなぎなどに使用します。

なお、CnkEasyDraw を使用する場合のみこの値は偶数にしてください。（中間バッファは 32 バイト単位で処理するため）

(4) num

頂点数を指定します。

vlist が NULL の時は処理を行いません。

7.4.2 PLIST

PLIST には、アトリビュート、テクスチャ、マテリアル、ストリップなど、いろいろなチャンクが含まれます。

(1) アトリビュート

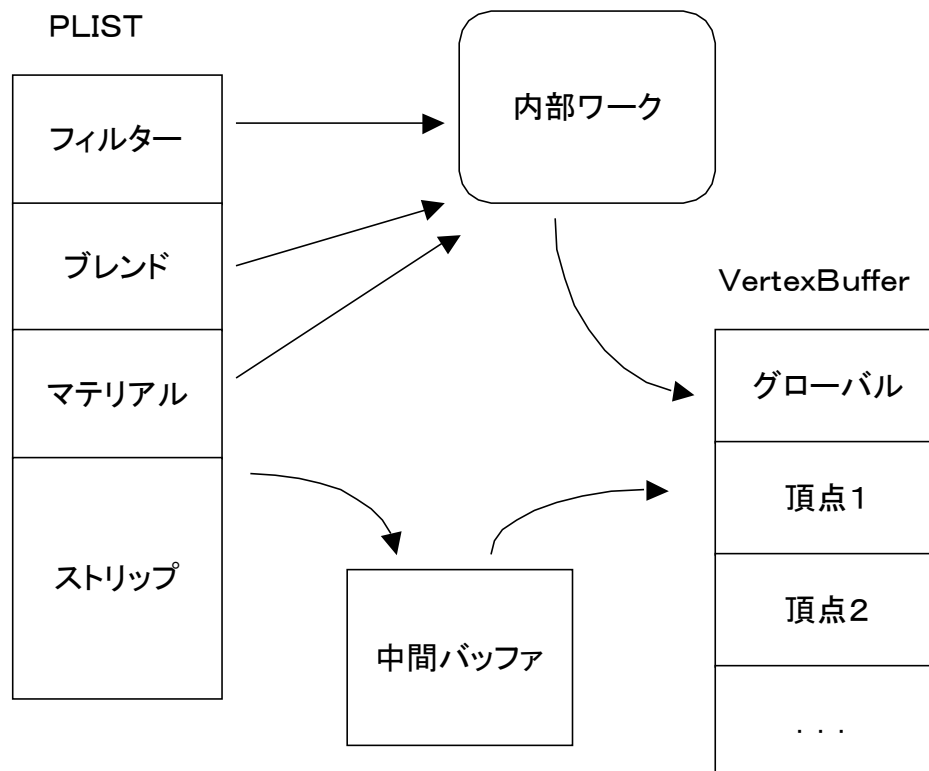
フィルターモード、ブレンドモードなどの各種アトリビュートは、描画関数に関係なく影響します。このチャンクは、内部のワークエリアの設定を変更するのみで、ハードウェアに対してデータ出力は行いません。

(2) マテリアル

ディフューズ、アンビエント、スペキュラの各マテリアルは、描画関数ごとに影響が変わります。それぞれの関数に応じたマテリアルを設定することにより、性能とデータ量が削減できます。このチャンクは、内部のワークエリアの設定を変更するのみで、ハードウェアに対してデータ出力は行いません。

(3) ストリップ

ストリップチャンクにより、実際のデータ出力を行います。

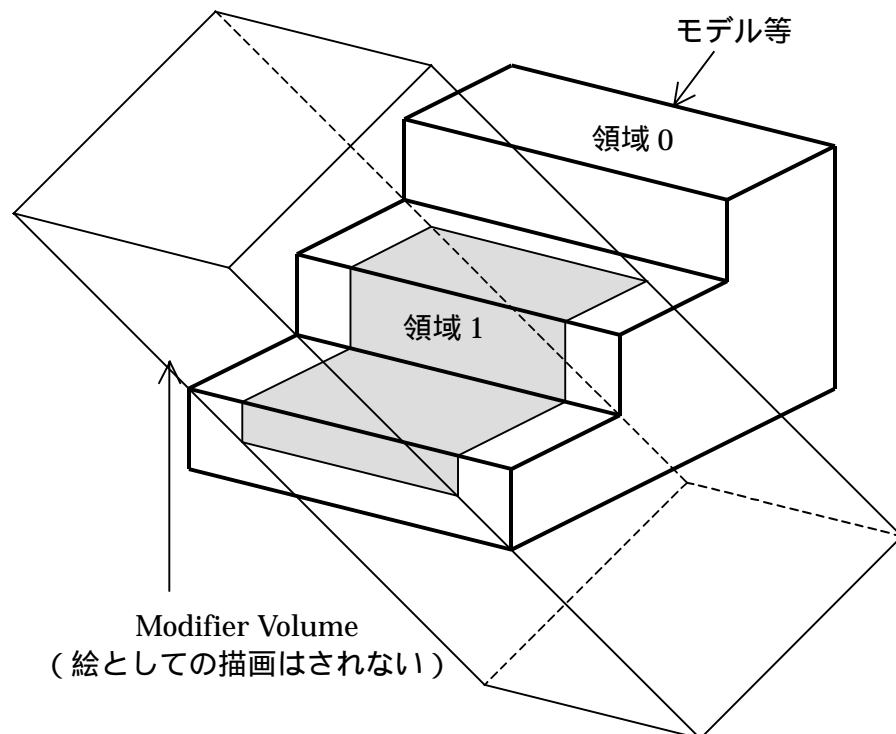


7.5 モディファイア

7.5.1 概念

モディファイアとは、影などを立体的に描画するためのエリアを指定するものです。モディファイアも通常のモデルデータと同じように、描画しますが、モディファイアそのもの（ボリューム）は画面には表示されません。影が映るには、実際に影が描画されるポリゴンを、モディファイアの影響を受けるように設定して描画します。モディファイアは形だけで、それがどのように描画されるかは通常のポリゴン（モデル）描画のときに決定します。

モディファイアボリュームと通常ポリゴンの重なったところが、影響範囲となります。



モディファイアの描画は、通常では、不透明モディファイアバッファのみ登録します。しかし、NJD_CONTROL_3D_TRANS_MODIFIERを設定すると、同じデータを半透明モディファイアにも登録します。この設定は、モディファイアボリュームを描画する際、どの頂点バッファを使用するか決めるものです。モディファイアを使用するときは、njInitVertexBuffer()でモディファイアのバッファの確保を忘れないようにして下さい。

また、実際に、モディファイアに影響されるポリゴン（モデル）を描画するには、NJD_CONTROL_3D_SHADOWを設定します。また、ポリゴンが不透明の時だけ反映させるには、NJD_CONTROL_3D_SHADOW_OPAQUEを設定します。この設定は通常のポリゴン（モデル）を描画する際の設定です。

7.5.2 チープシャドウモディファイア

チープモディファイアは、モディファイアの影響を輝度の変化のみにして、簡単に描画できるようにしたものです。njSetCheapShadowMode()関数で、輝度を設定すれば、描画データはそのまま、モディファイアと重なった部分の輝度が設定に応じて暗くなります。

7.5.3 2 P モディファイア

2 P モディファイアは、輝度ではなく、テクスチャや色合いまでも変化させることができます。ただし、2 P データを使った専用のモデルデータを作らなければなりません。

また、2 P モディファイアとチープシャドウモディファイアは、混在して使用することはできません。njSetCheapShadowMode(-1)とすることにより、2 P モディファイアモードとなります。

7.5.4 ニアクリップ

モディファイアボリュームは、閉空間のモデルにしなければなりません。ニアクリップでカットしてしまうと、モデルに穴があいてしまい、正常にモディファイアの処理が行えなくなります。従って現状は、ニアクリップにかかった場合、モディファイアの形を歪めて描画しています。

7.6 クリッピング

7.6.1 モデルクリップ

モデル構造体の center (中心) と r (半径) より、モデル全体のクリッピングを行います。完全に画面外の場合は、そこでモデルの処理は終わります。

なお、モデルクリップは、NJD_CONTROL_3D_MODEL_CLIP が有効の場合に行います。もし、画面内に入ることがあらかじめ判明しているモデルの場合、NJD_CONTROL_3D_MODEL_CLIP を無効にしたほうが、性能は向上します。

また、半径 r を 0 にすると、NJD_CONTROL_3D_MODEL_CLIP が有効の場合でもクリッピングを行いません。これは、クリップしては都合の悪いモデルの場合に設定します。例えば、間接をつなぐ場合の親のモデルに設定します。

ほかに、マテリアルだけを設定したい場合などに使います。例えば、テクスチャが 1 枚だけの木のモデルをたくさん描画するような場合、vlist が NULL で plist にマテリアルとテクスチャ ID だけを設定し、半径を 0 にしたモデル構造体を作ります。その子供としてストリップだけの木のモデルをたくさん設定すれば、木のモデル毎にテクスチャを設定しないため、性能が向上します。

7.6.2 ニアクリップ

中間バッファに生成されて頂点データから、PLIST の内容にそってポリゴンを生成します。その際、頂点の 1 つが視点の手前にあった場合、それまでのストリップを閉じて、そのポリゴンのニアクリップ処理を行います。

ニアクリップは、三角形をクリップ位置で分割することにより行います。頂点座標、頂点カラー、UV 値を補完し、新たな三角形を生成して描画します。

ニアクリップはすべてソフトで行いますので、非常に重たい処理になります。

8 オブジェクト

ここでは、オブジェクトについて説明します。

8.1 概要

オブジェクトは、複数のモデルの階層構造を定義したデータ構造です。また、オブジェクト間の位置関係を変化させることにより、モーションを表現できます。

8.2 構造

構造は次のとおりです。

```
typedef struct cnkobj {
    Uint32          evalflags;
    NJS_CNK_MODEL   *model;
    Float           pos[3];
    Angle           ang[3];
    Float           scl[3];
    struct cnkobj   *child;
    struct cnkobj   *sibling;
    Float           re_quat;
} NJS_CNK_OBJECT;
```

(1) evalflag

各種状態を設定するフラグです。
詳細は、別項を参照して下さい。

(2) model

描画を行うモデル構造体へのポインタです。

(3) pos

マトリクスのトランスレート成分です。

(4) ang

マトリクスの回転成分です。
ただしクォータニオンの場合は、イマジナリー成分データになります。

(5) scl

マトリクスのスケール成分です。

(6) child

子供のオブジェクトへのポインタです。

(7) sibling

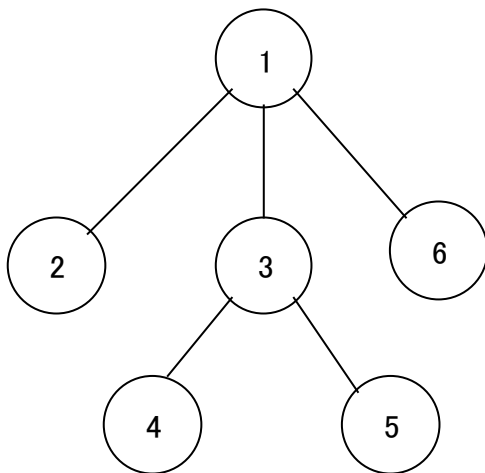
兄弟のオブジェクトへのポインタです。

(8) re_quat

クォータニオンの場合のリアルデータです。

8.3 オブジェクトトレース

オブジェクトのトレースは、子供からトレースします。



```
void Object( *obj )
{
    do {
        PushMatrix();
        Translate(obj->pos);
        Rotate( obj->ang );
        Scale(obj->scl );
        Model( obj->model );
        Object( obj->child );
        PopMatrix();
        obj = obj->sibling;
    } while( obj != NULL );
}
```

8.4 エンベロープ

8.4.1 概念

エンベロープは、滑らかな頂点つながりを実現する方法です。親子で頂点を共有する場合、共有頂点データの座標が、親子それぞれのマトリクスにどのような割合で影響するかを設定した、特殊な頂点データ形式を用いることにより実現します。この形式は、1つの頂点データを各ノードに分散して、それぞれにウェイト値を持たせます。そのため、頂点データの計算の確定は、分散された最後のノードの計算が終了した時点まで延びます。

親が、子供の影響を受ける場合、子供の計算が終了してから親のポリゴンを描画する必要があります。そのため、PLIST に描画するタイミングを変える特殊なチャンクを用意しました。

8.4.2 データ構造

(1) VLIST

エンベロープ専用の NJD_CV_VN_NF を用意しました。

これは、通常の頂点座標、頂点法線に加えて、頂点番号とウエイト値を持ったデータ形式です。各頂点の頂点番号は、チャンクヘッダにある頂点インデックスからのオフセット番号になります。

チャンクヘッダにあるフラグの違いにより、3種類にわかれます。

a. FW_START

最初のノードの頂点データです。

計算結果を中間バッファに格納します。

b. FW_MIDDLE

3ノード以上から影響される頂点の場合、最初と最後以外は、このデータになります。

計算結果を中間バッファに積算します。

c. FW_END

最後のノードの頂点データです。

計算後、中間バッファの内容と積算し、その後で光源計算や透視変換を行い、頂点データが決定されます。

(2) PLIST

親子でエンベロープで頂点をつないだ場合、親にストリップデータがあっても、子供の計算が終わるまで、親のストリップは描画できません。そのため、描画タイミングを変更するために、2つのチャンクを用意しました。

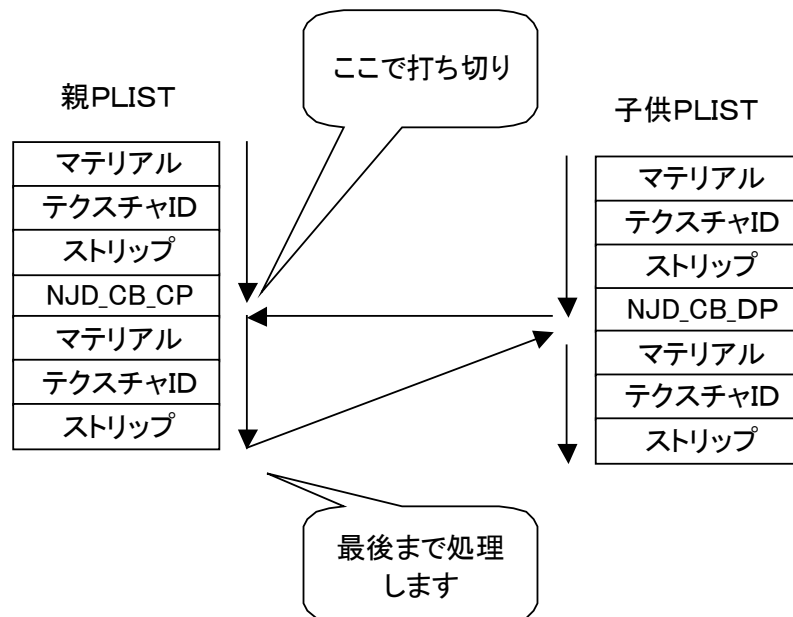
a. NJD_CB_CP

このチャンクを見つけると、PLISTのサーチをここで打ち切ります。チャンクには番号がついており、その番号と次のチャンクのアドレスを覚えます。

b. NJD_CB_DP

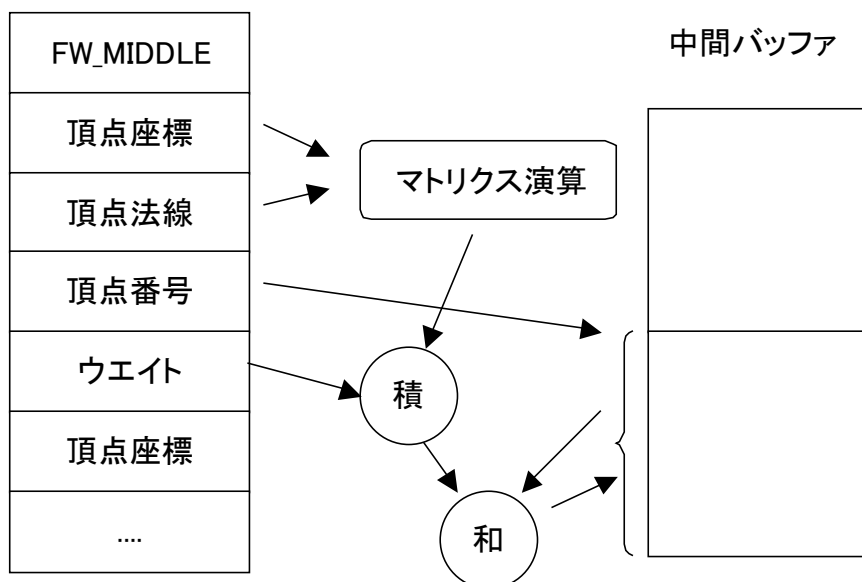
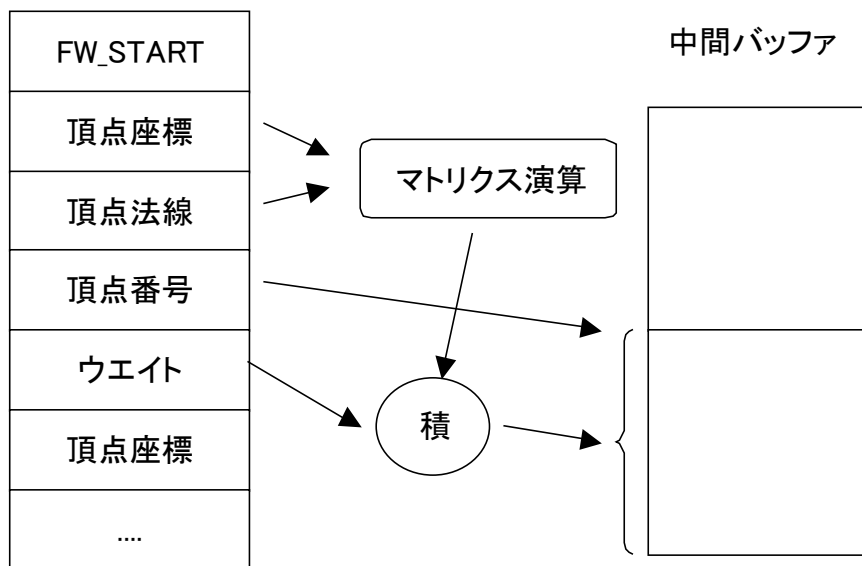
このチャンクにも番号がついており、NJD_CB_CPで覚えたアドレスから、サブルーチンのように実行します。

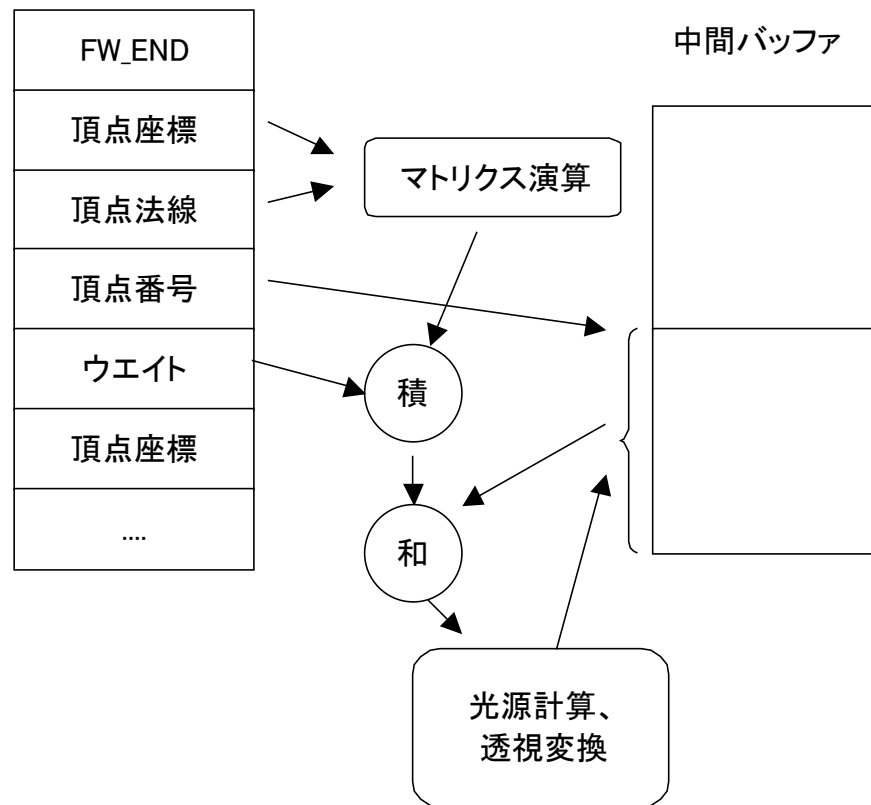
ストリップデータは親にあっても、そのアドレスを、NJD_CB_CPで覚えて、子供のPLISTにNJD_CB_DPをセットすることにより、タイミングをずらして描画することが可能です。なお、サブルーチンの範囲はデータの最後までですので、PLISTの途中で設定することはできませんので注意して下さい。



8.4.3 中間バッファ

エンベロープデータの処理中は、通常とは違うデータが中間バッファにセットされます。通常のデータでは、光源計算や透視変換まで行ったデータをセットしますが、エンベロープデータの場合、FW_END のデータで、初めて光源計算や透視変換を行います。それまでは、マトリクス変換しウエイトを掛けたものを積算していきます。





8.4.4 クリップ

エンベロープデータのクリッピングは、親頂点を参照するため、親のクリッピングを制御しなければなりません。

8.5 イーバルフラグ

EVAL フラグは、各種状態を設定するフラグです。

- (1) NJD_EVAL_UNIT_POS
マトリクスのトランスレート計算を行いません。
- (2) NJD_EVAL_UNIT_ANG
マトリクスの回転計算を行いません。
- (3) NJD_EVAL_UNIT_SCL
マトリクスのスケールの計算を行いません。
- (4) NJD_EVAL_HIDE
モデルを描画しません。
model が NULL の時は、必ず設定して下さい。ライブラリでは、model の NULL チェックを行っていません。
- (5) NJD_EVAL_BREAK
child のトレースを行いません。
child が NULL の時は、必ず設定して下さい。ライブラリでは、child の NULL チェックを行っていません。
- (6) NJD_EVAL_ZXY_ANG
回転マトリクスの計算方法を LightWave に合わせます。
- (7) NJD_EVAL_SKIP
モーションデータの無いオブジェクトをスキップします。
- (8) NJD_EVAL_SHAPE_SKIP
シェープデータの無いオブジェクトをスキップします。
- (9) NJD_EVAL_CLIP
そのモデルがモデルクリップにかかった場合、child のトレースを行いません。
child のモデルは半径を 0 にして、クリッピング計算をしないようにした方が、効率が良くなります。
- (10) NJD_EVAL_MODIFIER
そのモデルがモディファイアボリュームであることを示します。
ライブラリは、特に意識しません。
- (11) NJD_EVAL_QUATERNION
オブジェクトデータの形式がクォータニオンであることを示します。
この場合は、ローテートデータがクォータニオンのイマジナリデータになります。
- (12) NJD_EVAL_ROTATE_BASE
ローテートデータを記憶します。
記憶するデータは、オブジェクトの処理をする前のデータです。
- (13) NJD_EVAL_ROTATE_SET
記憶したローテートデータを使用します。
NJD_EVAL_ROTATE_BASE と NJD_EVAL_ROTATE_SET で、トグルエフェクトローテーションを行います。

9

モーション

ここではモーションについて、新規の内容を中心に説明します。

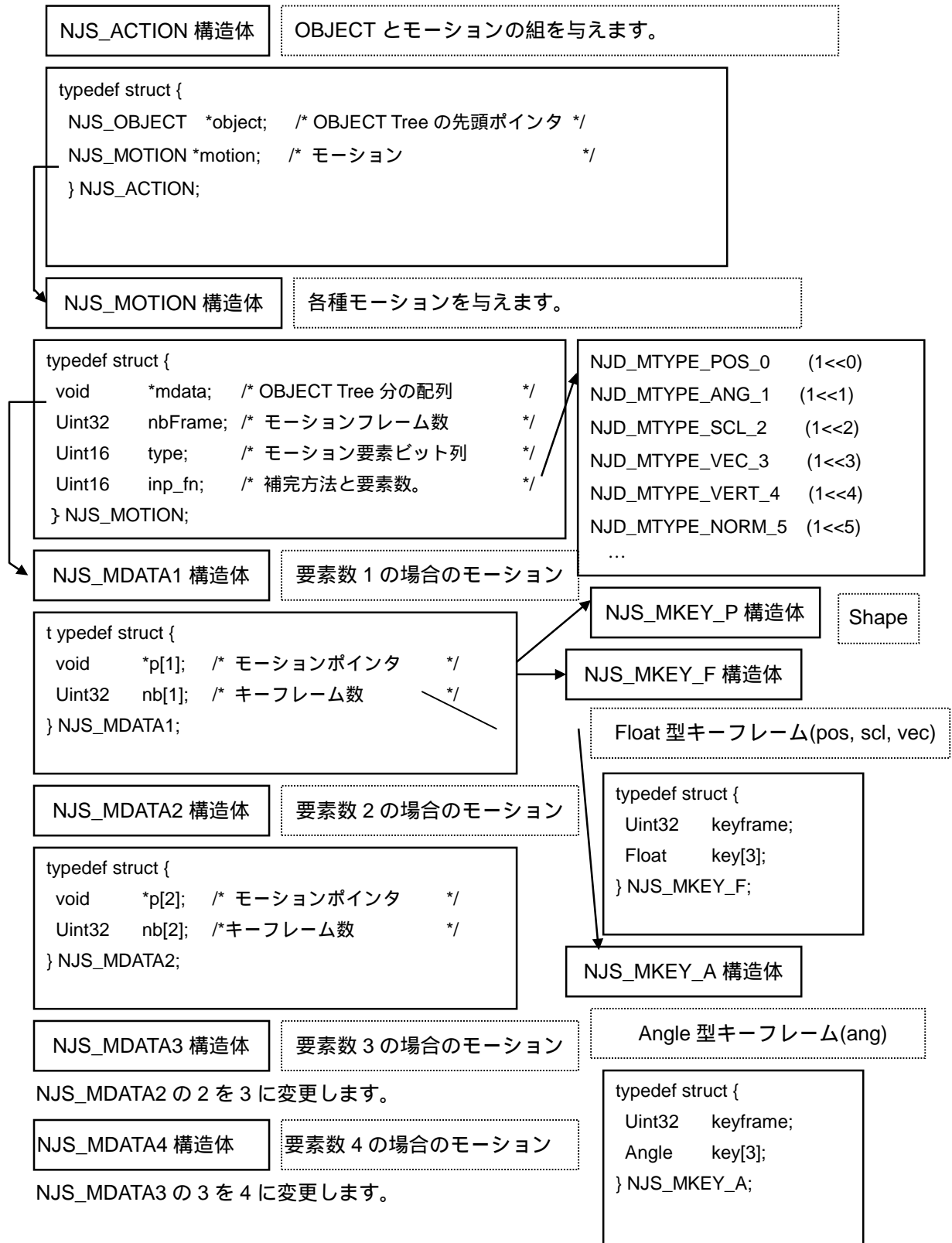
9.1 概要

Ninja はモデル、カメラ、ライトのモーションを同一構造体で定義します。

キーフレームを設定する単位にデータを配列化し、そのポインタテーブルでモーション全体を定義します。この方法により、必要箇所のみのモーションを持たせたり、パラメータごとにキーフレーム補完を行うことができます。また、カメラとライトのモーション共通部分を使い回すことも可能です。

9.2 モーション構造体

9.2.1 構造体関連図



9.2.2 構造体解説

- カメラ用アクション構造体

```
typedef struct {
    NJS_CAMERA      *camera;      /*カメラ構造体へのポインタ */
    NJS_MOTION      *motion;      /*モーションリスト */
}NJS_CACTION
```

- ライト用アクション構造体

```
typedef struct {
    NJS_LIGHT      *light;      /*ライト構造体へのポインタ */
    NJS_MOTION      *motion;      /*モーションリスト */
}NJS_LACTION
```

- モーション構造体

```
typedef struct {
    void          *mdata;      /* OBJECT Tree 分の配列 */
    Uint32        nbFrame;      /* モーションフレーム数 */
    Uint16        type;      /* モーション要素ビット列 */
    Uint16        inp_fn;      /* 補完方法と要素数。 */
}NJS_MOTION;
```

mdata には、OBJECT Tree に含まれる全 NJS_OBJECT に対応する数の NJS_MDATA(n)構造体の配列の先頭アドレスが入ります。

NJS_MDATA(n)は、モーション構成要素数 n により NJS_MDATA 1 ~ 5 構造体が選ばれます。

```
#define NJD_MTYPE_POS_0          (BIT_0)      //NJS_MKEY_F を利用
#define NJD_MTYPE_ANG_1          (BIT_1)      //NJS_MKEY_A を利用
#define NJD_MTYPE_SCL_2          (BIT_2)      //NJS_MKEY_F を利用
#define NJD_MTYPE_VEC_3          (BIT_3)      //NJS_MKEY_F を利用
#define NJD_MTYPE_VEC_0          (BIT_4)      //NJS_MKEY_F を利用
#define NJD_MTYPE_SANG_1         (BIT_5)      //NJS_MKEY_SA を利用
#define NJD_MTYPE_TARGET_3       (BIT_6)      //NJS_MKEY_F を利用
#define NJD_MTYPE_ROLL_6         (BIT_7)      //NJS_MKEY_SA1 を利用
#define NJD_MTYPE_ANGLE_7        (BIT_8)      //NJS_MKEY_SA1 を利用
#define NJD_MTYPE_RGB_8          (BIT_9)      //NJS_MKEY_F を利用
#define NJD_MTYPE_INTENSITY_9     (BIT_10)     //NJS_MKEY_F2 を利用
#define NJD_MTYPE_POINT_9        (BIT_12)     //NJS_MKEY_F2 を利用
#define NJD_MTYPE_QUAT_1         (BIT_13)     //NJS_MKEY_QUAT を利用
#define NJD_MTYPE_SHAPEDID       (BIT_14)     //NJS_MKEY_SHAPEID を利用
#define NJD_MTYPE_EVENT_4        (BIT_15)     //NJS_MKEY_U16 を利用
```

また、以下は廃止されました。

```
#define NJD_MTYPE_VERT_4          (BIT_4)
#define NJD_MTYPE_NORM_5          (BIT_5)
#define NJD_MTYPE_SPOT_10         (BIT_11)
```

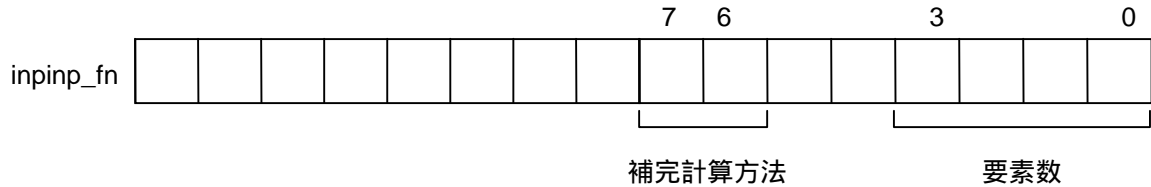
通常のモーションにおいて平行移動 (pos)、回転 (ang)、スケール (scl) の三要素が含まれる場合は、NJS_MDATA 3 が利用されます。NJD_MTYPE マクロ文字列の最後の番号はモーション要素の順番を意味しています。

ベクトル成分 vec は、pos とともに光源やカメラのモーションに利用します。

補完計算方法を inp_fn の上位 2 ビットで指定します。

```
#define NJD_MTYPE_LINER      0x0000 //線形補完
#define NJD_MTYPE_SPLINE    0x0040 //スプライン補完
#define NJD_MTYPE_USER      0x0080 //ユーザ関数補完
#define NJD_MTYPE_MASK      0x00c0 //抽出マスク
```

NJS_MDATA(n) 構造体の n を示すために、inp_fn の下位 4 ビットには要素数が格納されます。



● NJS_MDATA 1 ~ 5 構造体

```
typedef struct {
    void      *p[1]; /* モーションポインタ */
    Uint32     nb[1]; /* キーフレーム数 */
} NJS_MDATA1;

typedef struct {
    void      *p[2]; /* モーションポインタ */
    Uint32     nb[2]; /* キーフレーム数 */
} NJS_MDATA2;

typedef struct {
    void      *p[3]; /* モーションポインタ */
    Uint32     nb[3]; /* キーフレーム数 */
} NJS_MDATA3;
```

すべてのデータは、キーフレーム構造で表現されます。

p[i]メンバにはキー構造体へのポインタか、または NULL が入ります。

nb[i]メンバには、p[i]要素のモーションキーフレーム数が入ります。

また光源用に MDATA4、MDATA5 が定義されます。MDATA5 は、スポットライト光源の場合のみに使用されます。

```
typedef struct {
    void      *p[4]; /* モーションポインタ */
    Uint32     nb[4]; /* キーフレーム数 */
} NJS_MDATA4;

typedef struct {
    void      *p[5]; /* モーションポインタ */
    Uint32     nb[5]; /* キーフレーム数 */
} NJS_MDATA5;
```


● キー構造体

```
typedef struct {
    Uint32    keyframe; /* キーフレーン番号      */
    Float     key[3];   /* Float 型キー値(配列3) */
} NJS_MKEY_F;
```

平行移動 (POS)、スケール (SCL)、ベクトル (VEC)、ライトカラー (RGB) に利用します。

```
typedef struct {
    Uint32    keyframe; /* キーフレーン番号      */
    Angle     key[3];   /* Angle 型キー値(配列3) */
} NJS_MKEY_A;
```

オイラー角表現による回転要素に利用します。

```
typedef struct {
    Uint16    keyframe; /* キーフレーン番号      */
    Sangle     key[3]; /* Sangle 型キー値(配列3) */
} NJS_MKEY_SA;
```

ShortAngle を使用したオイラー角による回転要素に利用します。

```
typedef struct {
    Uint32    keyframe; /* キーフレーン番号      */
    Float     key[4]; /* Float 型キー値(配列4) */
} NJS_MKEY_QUAT;
```

クォータニオン表現による回転要素に利用します。

```
typedef struct {
    Uint32    keyframe; /* キーフレーン番号      */
    Uint32    key;      /* 符号無し int32 型キー値 */
} NJS_MKEY_UI32;
```

セルスプライトの各カラー (ARGB) に利用します。

```
typedef struct {
    Uint32    keyframe; /* キーフレーン番号      */
    Angle     key;      /* Angle 型キー値        */
} NJS_MKEY_A1;
```

セルスプライトで使います。カメラロール (ROLL)、画角 (ANGLE) には、NJS_MKEY_SA1 を利用します。

```
typedef struct {
    Uint16    keyframe; /* キーフレーン番号      */
    Sangle     key;      /* Sangle 型キー値        */
} NJS_MKEY_SA1;
```

カメラロール (ROLL)、画角 (ANGLE) に利用します。


```
typedef struct {
    Uint32    keyframe; /* キーフレーム番号      */
    Float     key;      /* Float 型キー値      */
} NJS_MKEY_F1;
```

現在、使用していません。

```
typedef struct {
    Uint32    keyframe; /* キーフレーム番号      */
    Float     key[2];   /* Float 型キー値(配列2) */
} NJS_MKEY_F2;
```

ライトで使えます。EasyDraw 及び SimpleDraw のときに、Intensity、Ambient 成分に使用し、EasyMultiDraw、SimpleMultiDraw のときは nrange、frange に利用します。

```
typedef struct {
    Uint16    keyframe; /* キーフレーム番号      */
    Uint16    key;      /* 符号無し int16 型キー値 */
} NJS_MKEY_U16;
```

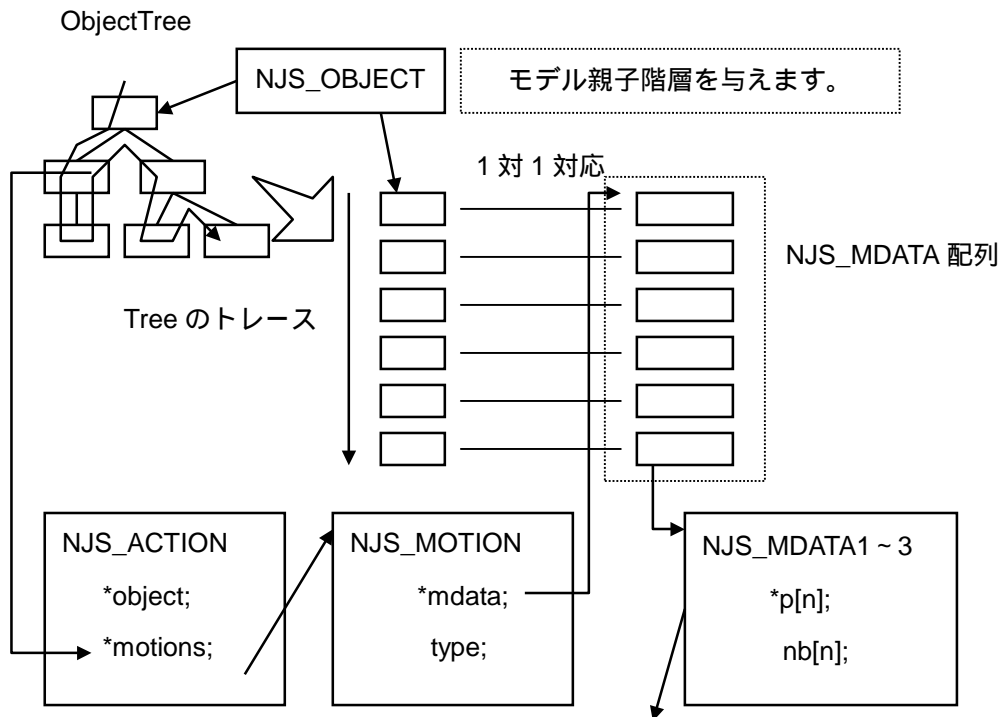
イベントモーションに利用します。現在は、ハイドモーションのみ利用が可能です。

```
typedef struct {
    Uint32    keyframe; /* キーフレーム番号      */
    Uint32    shapedId; /* ShapeList のエントリ ID */
} NJS_MKEY_U16;
```

CompactShape の nam 出力に利用します。

9.3 オブジェクトモーション

9.3.1 構造体関連図



(例) pos のみの場合：要素数 1 なので NJS_MDATA1 を使います。

NJS_MOTION 構造体の type = NJD_MKEY_POS_0;

NJS_MKEY_F pos[] = {, , , ...};

NJS_MDATA1 mdata[] = {{pos, poskey_n}, ...};

(例) pos, ang, scl の場合：要素数 3 なので、NJS_MDATA3 を使います。

type = NJD_MTYPE_POS_0 | NJD_MTYPE_ANG_1 | NJD_MTYPE_SCL_2;

MKEY_F pos[] = {, , , ...};

MKEY_A ang[] = {, , , ...};

MKEY_F scl[] = {, , , ...};

MDATA3 mdata[] = {{pos, ang, scl, poskey_n, angkey_n, sclkey_n}, ...};

(例) 光源の pos, vec の場合：要素数 2 なので NJS_MDATA2 を使います。

type = NJD_MTYPE_POS_0 | NJD_MTYPE_VEC_3;

NJS_MKEY_F pos[] = {, , , ...};

NJS_MKEY_F vec[] = {, , , ...};

NJS_MDATA2 mdata[] = {{pos, vec, poskey_n, veckey_n}, ...};

9.3.2 構造体解説

モーションは、すべてキーフレームデータで与えられます。
 キーフレームデータを線形補完、スプライン補完することにより、モーションを実行します。
 キーフレーム番号は、ゼロから始まります。マイナス値は、使用できません。

| | |
|--------------------------|--|
| Translation (Position) | 平行移動要素です。 |
| Angle (Rotation) | 回転要素です。16 ビット、32 ビットオイラー角表現と、Quaternion 表現が利用可能です。 |
| Scale | 拡大縮小要素です。 |
| Event | イベントモーションです。現在は、ハイドモーションのみ。 |
| ShapeID | ShapeList 参照による頂点モーション (CompactShape) です。 |

オブジェクトモーションの要素として、上記の五つが考えられます。

ライブラリ実装面の問題から、shape データである Vertex、Normal について、Position、Rotation、Scale (.nam) とは別データ (.nas) として出力します。このため最大要素数は三要素です。オブジェクトモーション用として、NJS_MDATA1 ~ 3 構造体が利用されます。光源、カメラ用に NJS_MDATA4、NJS_MDATA5 が定義されています。

NJS_MDATA の各要素を格納するポインタは void であり、各場合におけるデータの格納順番を規定する必要があります。

```
#define NJD_MTYPE_POS_0      (1<<0) /*NJS_MKEY_F を利用*/
#define NJD_MTYPE_ANG_1      (1<<1) /*NJS_MKEY_A を利用*/
#define NJD_MTYPE_SCL_2      (1<<2) /*NJS_MKEY_F を利用*/
#define NJD_MTYPE_VEC_3      (1<<3) /*NJS_MKEY_F を利用*/
#define NJD_MTYPE_VERT_4      (1<<4) /*NJS_MKEY_P を利用*/
#define NJD_MTYPE_NORM_5      (1<<5) /*NJS_MKEY_P を利用*/
#define NJD_MTYPE_QUAT_1      (1<<13) /*NJS_MKEY_QUAT を利用*/
```

define 文字列の最後に示す番号の若いデータが先に並びます。上記フラグはモーション構造体のメンバ type に設定されます。

- (例) pos と ang の場合
 type = NJD_MTYPE_POS_0 | NJD_MTYPE_ANG_1;
 mdata[] = {pos, ang, ...}

モーション補完方法は、type の上位 2 ビットで指定されます。

```
#define NJD_MTYPE_LINER      0x0000
#define NJD_MTYPE_SPLINE     0x0040
#define NJD_MTYPE_USER       0x0080
#define NJD_MTYPE_MASK       0x00c0
```

NJD_MTYPE_LINER は、線形補完です。

NJD_MTYPE_SPLINE は、スプライン補完です。

NJD_MTYPE_USER は、ユーザ定義のルーチンによる補完です (現在、使用不可)。

ルートが pos、ang で、他は ang のみのモーションモデルでは、NJS_MDATA2 構造体を使用し、ルート以外の pos には NULL ポインタを使用することにより対応します。

```
• type = NJD_MTYPE_POS_0 | NJD_MTYPE_ANG_1;
NJS_MDATA2 mdata[] = {
    { *pos1, *ang1 },
    { NULL, *ang2 },
    { NULL, *ang3 },
    .... }
```

注 意 この場合は、ang2、ang3 を左につめてはいけない点に注意して下さい。

9.4 カメラモーション

9.4.1 カメラ構造体

● カメラ設定用構造体

```
typedef struct{
    Float px,py,pz;
    Float vx,vy,vz;
    Sangle roll;
    Sangle ang;
    Uint32 type;
}NJS_CAMERA;
```

| | |
|----------|--|
| px,py,pz | カメラポジション |
| vx,vy,vz | カメラ方向ベクトル、またはターゲットポジション |
| roll | カメラロール |
| ang | カメラの画角 |
| type | カメラ種類 (NJD_CTYPE_VECTOR、 NJD_CTYPE_TARGET) |

カメラの種類には、以下の 2 つの定義が設定できます。

```
#define NJD_CTYPE_VECTOR      (1)      /* フリーカメラ      */
#define NJD_CTYPE_TARGET     (2)      /* ターゲット        */
```


9.4.2 カメラモーションについて

カメラは親子階層を構成しないため、1 オブジェクトに対するモーション構造と基本的に同じです。アクション構造体には、NJS_CACTION を使用します。

- カメラの4要素
 - 位置 (POS)
 - 方向 (VEC) またはターゲット (TARGET)
 - ロール (ROLL)
 - 画角 (ANGLE)

必要に応じて、NJS_MTYPE_1 から NJS_MTYPE_4 を使用します。

```
#define NJD_MTYPE_POS_0      (1<<0) /* NJS_MKEY_F を利用*/
#define NJD_MTYPE_VEC_3      (1<<3) /* NJS_MKEY_F を利用*/
#define NJD_MTYPE_TARGET_3   (1<<6) /* NJS_MKEY_F を利用*/
#define NJD_MTYPE_ROLL_6     (1<<7) /* NJS_MKEY_SA1 を利用*/
#define NJD_MTYPE_ANGLE_7    (1<<8) /* NJS_MKEY_SA1 を利用*/
```

Vec はカメラの中心軸の方向ベクトル、Target はターゲットのポジションを意味します。Vec と Target は、排他的に使用します。Target = Pos + Vec の関係があります。

NJD_MTYPE_VEC_3 と NJD_MTYPE_TARGET_3 が同じ 3なのは、同時に使えないこと (排他的) を示します。

- フリーカメラ (向きをベクトルで持つカメラ) の場合
 type=NJD_MTYPE_POS_0|NJD_MTYPE_VEC_3|NJD_MTYPE_ROLL_6|NJD_MTYPE_ANGLE_7;
- ターゲットカメラ (向きをターゲット位置で持ち画角アニメーションを持つカメラ) の場合
 type=NJD_MTYPE_POS_0|NJD_MTYPE_TARGET_3|NJD_MTYPE_ROLL_6|NJD_MTYPE_ANGLE_7;

補完計算方法を inp_fn の上位 2 ビットで指定します。オブジェクトモーションと同じです。

9.5 ライトモーション

9.5.1 ライト構造体

- ライト設定用構造体

```
typedef struct{
    Float      x,y,z;
    Float      r,g,b;
    Float      f1,f2;
    Sint32     func;
    Uint32     type;
}NJS_LIGHT;
```

| | |
|-------|--|
| x,y,z | 点光源のときライトポジション、平行光源のときライトベクトル。 |
| r,g,b | ライトカラー0.f - 1.fが通常値です。Multi系のみ、それ以外（負数）も設定可能です。 |
| f1,f2 | Easy、Simple ライトのときインテンシティ、アンビエント 0.f - 1.f が通常値です。 それ以外（負数）も設定可能です。 EasyMulti、SimpleMulti 点光源ライトのときライト範囲（nrange,frange）。 EasyMulti、SimpleMulti 平行光源ライトのとき未使用。 EasyMulti、SimpleMulti アンビエントライトのとき未使用。 |
| func | 描画関数設定（NJD_MFUNC_EASY、NJD_MFUNC_SIMPLE、 NJD_MFUNC_EASY_MULTI、NJD_MFUNC_SIMPLE_MULTI） |
| type | ライト種類設定（NJD_LTYPE_POINT、NJD_LTYPE_VECTOR、 NJD_LTYPE_AMBIENT） |

描画関数の種類には、以下の4つの定義が設定できます。

```
#define NJD_MFUNC_EASY          (1)          /* CnkEasy 関数          */
#define NJD_MFUNC_SIMPLE       (2)          /* CnkSimple 関数        */
#define NJD_MFUNC_EASY_MULTI   (3)          /* CnkEasyMulti 関数     */
#define NJD_MFUNC_SIMPLE_MULTI (4)          /* CnkSimpleMulti 関数   */
```

ライトの種類には、以下の3つの定義が設定できます。

```
#define NJD_LTYPE_POINT        (1)          /* 点光源                */
#define NJD_LTYPE_VECTOR       (2)          /* 平行光源              */
#define NJD_LTYPE_AMBIENT      (3)          /* アンビエントライト    */
```


9.5.2 ライトモーションについて

ライトは親子階層を構成しないため、1 オブジェクトに対するモーション構造と基本的に同じです。アクション構造体は NJS_LACTION を使用します。

ライトのモーションの対象としては、点光源、平行光源、アンビエントライトを想定しています。

- 点光源の3要素
 - 位置 (POS)
 - 範囲 (POINT)
 - 色 (RGB)

範囲の要素は、前方の限界値 (NearRange) 後方の限界値 (FarRange) をひとまとめにします。

- 平行光源の3要素
 - 方向 (VEC)
 - 色 (RGB)
 - 輝度、環境光 (INTENSITY、AMBIENT)
- アンビエントライトの要素
 - 色 (RGB)

よって、点光源や平行光源では、NJS_MDATA_1 から NJS_MDATA_3 を使用し、アンビエントライトでは NJS_MDATA_1 を使用します。

| | | |
|-------------------------------|----------|-------------------|
| #define NJD_MTYPE_POS_0 | (BIT_0) | //NJS_MKEY_F を利用 |
| #define NJD_MTYPE_VEC_0 | (BIT_4) | //NJS_MKEY_F を利用 |
| #define NJD_MTYPE_RGB_8 | (BIT_9) | //NJS_MKEY_F を利用 |
| #define NJD_MTYPE_INTENSITY_9 | (BIT_10) | //NJS_MKEY_F2 を利用 |
| #define NJD_MTYPE_POINT_9 | (BIT_12) | //NJS_MKEY_F2 を利用 |

NJD_MTYPE_POS_0 と NJD_MTYPE_VEC_0 が同じ 0、NJD_MTYPE_INTENSITY_9 と NJD_MTYPE_POINT_9 が同じ 9なのは、同時に使用できないことを示します。

(1) Type の設定例

- 点光源


```
type=NJD_MTYPE_POS_0|NJD_MTYPE_RGB_8| | NJD_MTYPE_POINT_9 ;
```
- 平行光源 (向きをベクトルで持つ平行光源) の場合


```
type=NJD_MTYPE_VEC_0 |NJD_MTYPE_RGB_8| NJD_MTYPE_INTENSITY_9;
```
- アンビエントライトの場合


```
type= NJD_MTYPE_RGB_8
```

補完計算方法を inp_fn の上位 2 ビットで指定します。オブジェクトモーションと同様です。

9.5.3 描画関数とライト種類によるライト構造体のメンバ

描画関数設定とライト種類設定により、ライト構造体のメンバの意味が異なります。

(1) CnkEasyDraw

- 平行光源

| | |
|-------|---------------------|
| type | NJD_LTYPE_VECTOR |
| x,y,z | ライト方向ベクトル |
| r,g,b | ライトカラー |
| f1,f2 | Intensity,Ambient 値 |
| func | NJD_MFUNC_EASY |

njLightMotion 関数では、以下の CnkEasyLight の関数に設定しています。

```
njCnkSetEasyLight( x, y, z);
njCnkSetEasyLightColor( r, g, b );
njCnkSetEasyLightIntensity( f1, f2);
```

CnkEasyDraw には、点光源やアンビエントライトはありません。

(2) CnkSimpleDraw

- 平行光源

| | |
|-------|---------------------|
| type | NJD_LTYPE_VECTOR |
| x,y,z | ライト方向ベクトル |
| r,g,b | ライトカラー |
| f1,f2 | Intensity,Ambient 値 |
| func | NJD_MFUNC_EASY |

njLightMotion 関数では、以下の CnkSimpleLight の関数に設定しています。

```
njCnkSetSimpleLight( x, y, z);
njCnkSetSimpleLightColor( r, g, b );
njCnkSetSimpleLightIntensity( f1, f2);
```

CnkSimpleDraw には、点光源やアンビエントライトはありません。

(3) CnkEasyMultiDraw

- 平行光源

| | |
|-------|----------------------|
| type | NJD_LTYPE_VECTOR |
| x,y,z | ライト方向ベクトル |
| r,g,b | ライトカラー |
| f1,f2 | なし |
| func | NJD_MFUNC_EASY_MULTI |

njLightMotion 関数では、以下の CnkEasyMultiLight の関数に設定しています。

```
njCnkSetEasyMultiLightVectorEx( num, x, y, z);
njCnkSetEasyMultiLightColor( r, g, b);
```

num は、ライト番号です。

● 点光源

| | |
|-------|-------------------------|
| type | NJD_LTYPE_POINT |
| x,y,z | ライト位置 |
| r,g,b | ライトカラー |
| f1,f2 | ライト範囲 (nrange,frange) |
| func | NJD_MFUNC_EASY_MULTI |

njLightMotion 関数では、以下の CnkEasyMultiLight の関数に設定しています。

```
njCnkSetEasyMultiLightPoint( num, x, y, z );
njCnkSetEasyMultiLightColor( num, r, g, b );
njCnkSetEasyMultiLightRange( num, f1, f2 );
```

num はライト番号です。

● アンビエントライト

| | |
|-------|----------------------|
| type | NJD_LTYPE_AMBIENT |
| X,y,z | なし |
| R,g,b | ライトカラー |
| F1,f2 | なし |
| func | NJD_MFUNC_EASY_MULTI |

njLightMotion 関数では、以下の CnkEasyMultiLight の関数に設定しています。

```
njCnkSetEasyMultiAmbient ( num, r, g, b );
```

num はライト番号です。

njLightMotion 関数では、njCnkSetEasyMultiLightSwitch 関数、及び
njCnkSetEasyMultiLightMatrices 関数は実行していません。

(4) CnkSimpleMultiDraw

● 平行光源

| | |
|-------|------------------------|
| type | NJD_LTYPE_VECTOR |
| x,y,z | ライト方向ベクトル |
| r,g,b | ライトカラー |
| f1,f2 | なし |
| func | NJD_MFUNC_SIMPLE_MULTI |

njLightMotion 関数では、以下の CnkSimpleMultiLight の関数に設定しています。

```
njCnkSetSimpleMultiLightVectorEx( num, x, y, z );
njCnkSetSimpleMultiLightColor( r, g, b );
```

num はライト番号です。

● 点光源

| | |
|-------|--------------------------|
| type | NJD_LTYPE_POINT |
| x,y,z | ライト位置 |
| r,g,b | ライトカラー |
| f1,f2 | ライト範囲 (nrange、 frange) |
| func | NJD_MFUNC_SIMPLE_MULTI |

njLightMotion 関数では、以下の CnkSimpleMultiLight の関数に設定しています。

```
njCnkSetSimpleMultiLightPoint( num, x, y, z );
njCnkSetSimpleMultiLightColor( num, r, g, b );
njCnkSetSimpleMultiLightRange( num, f1, f2 );
```

num はライト番号です。

● アンビエントライト

| | |
|-------|------------------------|
| type | NJD_LTYPE_AMBIENT |
| x,y,z | なし |
| r,g,b | ライトカラー |
| f1,f2 | なし |
| func | NJD_MFUNC_SIMPLE_MULTI |

njLightMotion 関数では、以下の CnkSimpleMultiLight の関数に設定しています。

```
njCnkSetSimpleMultiAmbient ( num, r, g, b );
```

num はライト番号です。

njLightMotion 関数では、njCnkSetSimpleMultiLightSwitch 関数、及び njCnkSetSimpleMultiLightMatrices 関数は実行していません。

9.5.4 マルチライトモーションについて

CnkEasyMulti、CnkSimpleMulti 関数で複数光源のモーションを同時に行うときに、マルチライト構造体にライトモーションを一括して登録し、njMultiLightMotion 関数でモーションすることができます。

```
typedef struct{
    Int          n;
    NJS_LIGHT    **lights;
    NJS_MOTION   **motions;
}NJS_MLIGHT_MOTION;
```

| | |
|---------|-----------------------|
| n | 設定するライトの個数 |
| lights | ライト構造体のポインタ配列へのポインタ |
| motions | ライトモーションのポインタ配列へのポインタ |

lights,motions の配列の要素は、n 個設定して下さい。njMultiLightMotion 関数では、njCnkSetEasyMultiLightSwitch 関数または、njCnkSetSimpleMultiLightSwitch 関数ですべてのライトを ON にしています。njCnkSetEasyMultiLightMatrices 関数、及び njCnkSetSimpleMultiLightMatrices 関数は実行していません。

9.6 回転の表現について

9.6.1 回転の表現とは

3次元空間上にオブジェクトを変形させずに配置するには、平行移動と回転だけで十分であることが知られています。

このとき「純粋な平行移動」「純粋な回転」というものを考えることが可能です。オブジェクトに属する点は、ある座標系によって測った座標によって表現されますが、

「純粋な平行移動」とは、オブジェクトに属するすべての点を、一定方向に一定距離だけ離れた位置に動かすようなすべての動きが相当します。「純粋な回転」とは、オブジェクトを変形させず、かつ、原点（座標値がすべて0の点）を動かさないすべての動きが相当します。

まとめると次のようになります。

| | |
|-----------|---|
| 「純粋な平行移動」 | オブジェクトの任意の点を、一定方向に一定距離だけ離れた位置に動かすようなすべての動き |
| 「純粋な回転」 | オブジェクトを変形させず、かつ、原点（座標値がすべて0の点）を動かさないようなすべての動き |

ノート 「純粋な回転」が「変形せず原点が動かないすべての動き」が定義なのにもかかわらず、「回転」と呼ぶのは、任意の「純粋な回転」が、原点を通るある直線の周りの回転になっているためです。

このように定義したとき、Position 要素が「純粋な平行移動」、もしくは Rotation 要素が「純粋な回転」に対応します。

また、「純粋な平行移動」が担う動きを「平行移動要素（成分）」、純粋な回転の担う動きを「回転要素（成分）」と呼ぶことがあります。

9.6.2 回転の表現について

(1) 「純粋な平行移動」

位置ベクトルに一定のベクトルを加算することに対応するので非常に単純です。よって Ninja での Position 要素も、加算すべきベクトルの座標表現を与える方法だけしかなく、またこれで十分と考えられます。

(2) 「純粋な回転」

回転を表す方法がいくつか用意されています。一つはオイラー角（Euler Angle）による方法で、もう一つはクォータニオン（Quaternion）による方法です。

よって、同じ回転でも使用する表現方法によって異なる表現になります。このように、回転を表す方法や、表された値そのものを、「回転の表現」といいます。

9.6.3 オイラー角 (Euler Angle) による方法

座標系の X、Y、Z 各軸周りの回転を順に施すことによって一般的な回転を表現することを、「オイラー角による方法」、もしくは「オイラー角による表現」などといいます。三次元空間の純粋な回転はこの方法で十分表現できることが知られています。

注意 オイラー角による方法では、回転を行う回転軸の順序によって、同じ回転でも違う表現になりますので注意が必要です。

Ninja では、X-->Y-->Z の順に回転させる方法と、Z-->X-->Y の順に回転させる方法の二つが用意されています。前者では、プログラム上でマトリックスを作成する際、`njRotateZ(angz);njRotateY(angy);njRotateX(angx)` の順に関数を呼び出します。後者では、`njRotateY(angy);njRotateX(angx);njRotateZ(angz)` の順に関数を呼び出します。

注意 オイラー角の表現には、同じ 1 つの「純粋な回転」に対して一般に 2 つの表現が対応していることです。例えば、X-->Y-->Z の順に回転させる表現で、 $(\text{angx}, \text{angy}, \text{angz}) = (180 \text{ 度}, 180 \text{ 度}, 0 \text{ 度})$ の回転と、 $(\text{angx}, \text{angy}, \text{angz}) = (0 \text{ 度}, 0 \text{ 度}, 180 \text{ 度})$ の回転が同じ回転を表しています。

9.6.4 クォータニオン (Quaternion) による方法

オイラーの方法では 3 回の回転を行っていましたが、実際には、すべての「純粋な回転」は、原点を通るある直線の周りの一回の回転によって表せることが知られています。このことを利用したのがクォータニオンによる方法です。

クォータニオンは日本語では、「ハミルトンの 4 元数」と呼ばれ、「複素数」における虚数単位 i を I 、 j 、 k の 3 つに拡張したものです。これを、 $q = u + ai + bj + ck$ などと書きます。

また、4 つの実数係数をベクトルのように並べて、 $q = (u, a, b, c)$ などと書く場合もあります。

回転とクォータニオンは、次のように対応します。

回転軸の方向ベクトルを N (単位ベクトル)、回転角を θ とすると、対応するクォータニオン表現は次のようになります。

$$\begin{aligned} q1(N, \theta) &= (\cos(\theta/2), \sin(\theta/2)Nx, \sin(\theta/2)Ny, \sin(\theta/2)Nz) \\ q2(N, \theta) &= (-\cos(\theta/2), -\sin(\theta/2)Nx, -\sin(\theta/2)Ny, -\sin(\theta/2)Nz) \\ &= -q1(N, \theta) \end{aligned}$$

2 つ書いたのは、同じ回転に 2 つのクォータニオンが対応するためです。

また、これらをベクトルのように考えると、いずれも「単位ベクトル」になっていることに注意して下さい。Ninja では、クォータニオンは必ず「単位ベクトル」で表現されます。単位ベクトルになっていない場合は、動作の保証はされません。

クォータニオン表現による回転をマトリックスに施す関数は、`njQuaternionEx()` です。この関数は、上記のいずれのクォータニオンでも同じマトリックスをカレントマトリックスに乗算します。

9.6.5 クォータニオンの利点

クォータニオンはこのように、4つの実数 (Float) で表現しますので3つの Angle (Sint32) で表現するオイラー角に比べてデータ量は増えます。利点は、モーションで回転成分を補完するときに理想的な補完結果を得られることです。

補完に関する詳細は、補完の章で述べますので、ここでは、概略を述べます。

オイラー角で回転成分を与えた場合、Ninja では各軸周りの回転角をそれぞれ別々に線形補完 (厳密には近い方への線形補完) をしてしまいます。その結果、補完途中の回転は単純な経路を通らず、「遠回り」な経路を通過してしまう場合があります。

クォータニオンで回転成分を与えると、補完途中、このような「遠回り」な経路を通らずに、最も「近回り」な経路を通るように回転します。厳密には、次のような補完結果となります。

あるノードを、補完元の姿勢から補完先の姿勢へと動かすときの「回転成分」は、必ず原点を通るある軸の周りの回転で表せます。

もし、「平行移動成分」が全く無いとすると、クォータニオンで補完途中にはどの瞬間も、補完元の姿勢をこの軸を中心に回転した姿勢にオブジェクトが描画されます。なお、この軸周りの回転角度は、現在のアルゴリズムでは、必ずしも一定速度で増加するわけではありません。

平行移動成分がある場合は、さらに平行移動を合成した位置にオブジェクトが描画されます。

9.7 回転要素の取り扱い

回転要素は、NJS_OBJECT 構造体、及び NJS_MOTION 構造体に入りますが、それぞれ独立した回転の表現（データ形式）を用いることができます。

9.7.1 NJS_OBJECT 構造体中での回転要素

NJS_OBJECT 構造体では、evalflags メンバに回転のデータ形式を指定します。

evalflags メンバに NJD_EVAL_QUATERNION フラグが設定されている場合は、クォータニオン表現になります。NJS_EVAL_QUATERNION フラグが設定されていない場合には、オイラー角による表現になります。オイラー角による表現の場合、さらに、NJD_EVAL_ZXY_ANG フラグが設定されている場合は、回転の順序が Z-->X-->Y になり、指定されていない場合は、X-->Y-->Z となります。

クォータニオンによる表現の場合、re_quat メンバにクォータニオンの実数成分、ang[3]メンバにクォータニオンの虚数 3 成分を指定します。よって、クォータニオンの実数係数の 4 次元ベクトルが、(re_quat, ang[0], ang[1], ang[2]) となります。なお、ang[3]メンバの型は、Angle(Sint32)型ですが、中身は、Float 型になります。よって、参照や代入する場合は、*(Float *)&ang[0] = 0.1f; などのように、ポインタを(Float *)で一旦キャストしてから、間接演算子(*) を付与するようにして下さい。

| evalflags | 表現法 | 使用メンバ |
|---------------------|--------------------|-----------------|
| NJD_EVAL_QUATERNION | クォータニオン表現 | re_quat, ang[3] |
| なし | オイラー角による表現（XYZ 形式） | ang[3] |
| NJD_EVAL_ZXY_ANG | オイラー角による表現（ZXY 形式） | ang[3] |

NJS_OBJECT 構造体中の回転の表現は、オブジェクト描画関数の段階で既に影響があります。

モーション描画関数や低レベルオブジェクトモーション関数などでは、NJS_MOTION 構造体中に回転要素が存在しない場合に、NJS_OBJECT 構造体中の回転要素を採用しますので、影響があります。

9.7.2 NJS_MOTION 構造体中での回転要素

NJS_MOTION 構造体中の回転要素の形式は、type メンバに指定します。

type メンバに NJD_MTYPE_QUAT_1 が設定されている場合は、NJS_MKEY_QUAT 構造体を使用され、クォータニオン形式が使用されます。

type メンバに NJD_MTYPE_ANG_1 が設定されている場合は、NJS_MKEY_A 構造体を使用され、オイラー角形式が使用されます。オイラー角形式の場合、対応するノードの NJS_OBJECT 構造体の evalflags に NJD_EVAL_ZXY_ANG が指定されていれば、ZXY 形式に、指定されていなければ、XYZ 形式となります。

また、オイラー角形式には新たに、NJD_MTYPE_SANG_1 が追加され、NJS_MKEY_SA 構造体を使用されます。NJD_MTYPE_ANG_1 は、オイラー角を 32 ビット形式で表現していますが、NJD_MTYPE_SANG_1 はオイラー角を 16 ビットで表現しています。

| type | 表現法 | 構造体 |
|------------------|---|---------------|
| NJD_MTYPE_QUAT_1 | クォータニオン表現 | NJS_MKEY_QUAT |
| NJD_MTYPE_ANG_1 | オイラー角表現（32bit） evalflags により XYZ または ZXY を区別 | NJS_MEKY_A |
| NJD_MTYPE_SANG_1 | オイラー角表現（16bit） evalflags により XYZ または ZXY を区別 | NJS_MEKY_SA |

9.7.3 NJS_MOTION 構造体と NJS_OBJECT 構造体の回転要素の関係

NJS_MOTION 構造体と、NJS_OBJECT 構造体の回転要素の表現法が異なっていますが構いません。

例えば、NJS_MOTION 構造体の type メンバに、NJD_MTYPE_QUAT_1 を指定した場合について説明します。

このとき、NJS_MOTION 構造体の回転要素は、クォータニオン表現で与えられます。NJS_MDATA?構造体の p[x] メンバ (x は平行移動要素などがあるかどうかによって変化します) には、NJS_MKEY_QUAT 構造体へのポインタが格納されます。ただし、NULL ポインタである場合もあります。

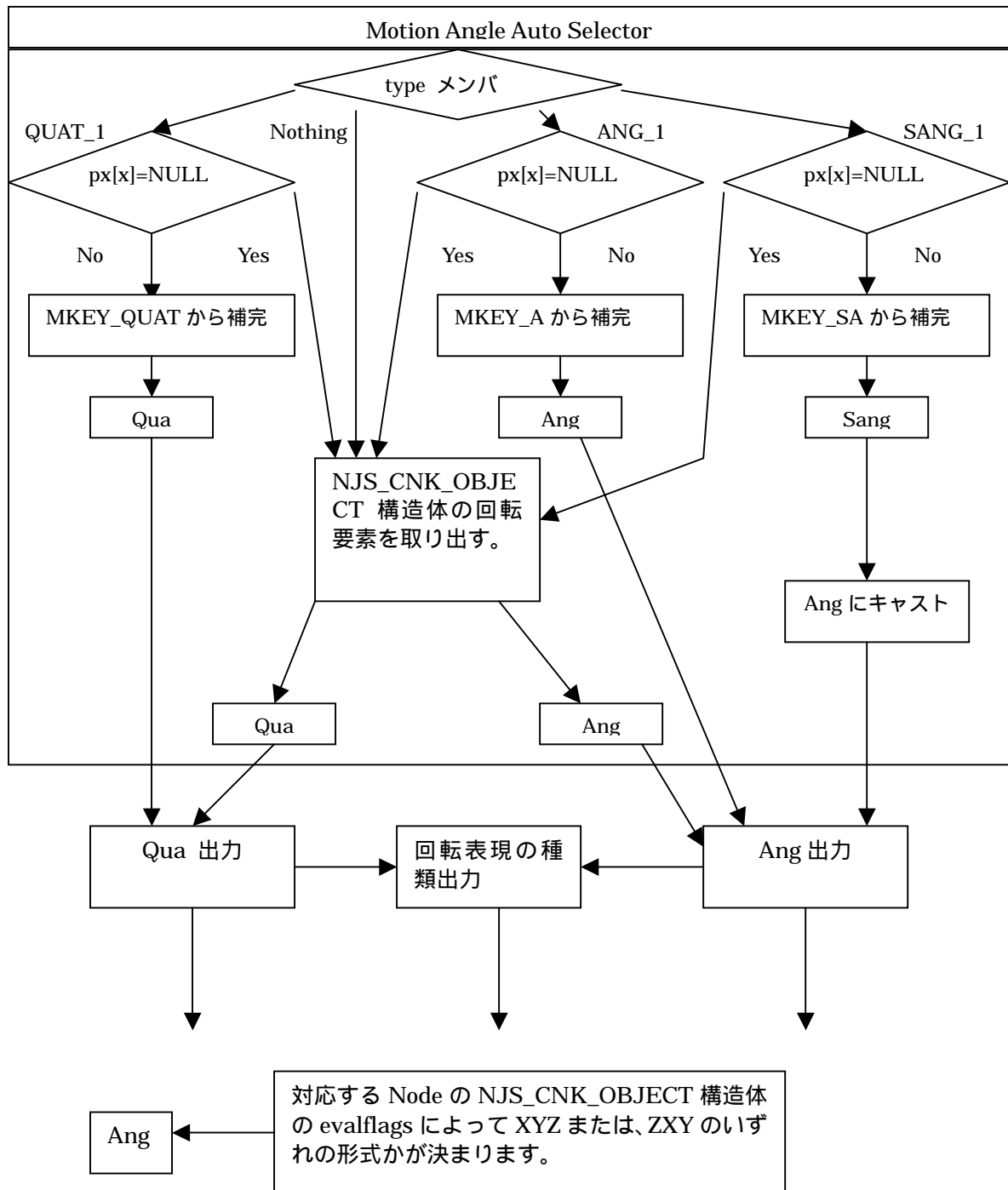
NJS_MKEY_QUAT 構造体へのポインタを格納すべき場所が NULL ポインタではなく本来の構造体へのポインタが入っている場合は、最初の NJS_MKEY_QUAT 構造体に後ろに連続して、NJS_MDATA?構造体の nb[x]メンバで指定されただけの要素数の配列が並んでいます。この要素数がキーフレームの個数です。このとき、そのノードの回転要素は、その NJS_MKEY_QUAT 構造体配列中の適切な隣接キーフレームのクォータニオンデータを適切に補完したものとなります。このとき、低レベルモーション関数の、njGetMotionNodeData()は、クォータニオン表現で回転要素を返します。

NJS_MKEY_QUAT 構造体へのポインタを格納すべき場所に NULL ポインタが入っている場合は、そのノードには回転要素のモーションデータが存在しないことを意味します。この場合は、NJS_OBJECT 構造体の回転要素が使用され、回転の表現は evalflags の指定に従います。このとき、低レベルモーション関数の、njGetMotionNodeData()は、evalflags で指定された表現で回転要素を返します。

オブジェクトモーションを行うすべてのモーション関数は、必ずマトリックスを作成する際、njMotionTransformEx()関数を呼び出します。njMotionTransformEx()関数は、内部で njGetMotionNodeData()関数を呼び出し、戻り値によって示される回転の表現に従って、回転要素に対応するマトリックスを作成します。

従って、オブジェクトモーションを行うすべてのモーション関数において、NJS_MOTION 構造体や、NJS_OBJECT 構造体の回転要素の表現法は、任意の組み合わせを取ることができます。

- njGetMotionNodeData() 関数の回転要素取り出し部：



9.7.4 モーションリンクにおける回転要素の補完について

モーションリンクは、1つのオブジェクトツリーについての、2つの独立したモーションデータの間をさらに補完する機能です。詳しくは、モーションリンクの説明をご覧ください。

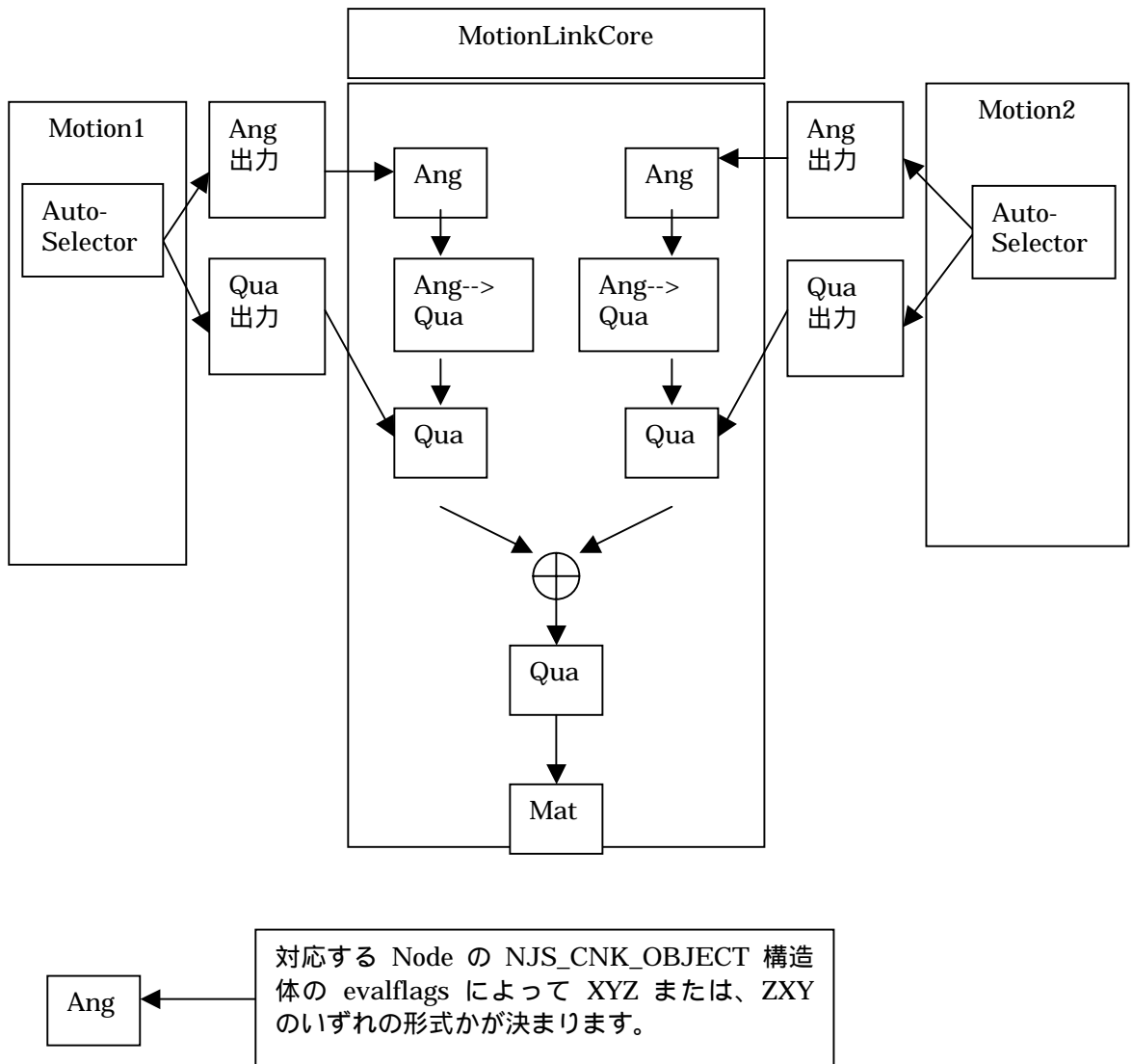
モーションリンクを行うときには、一回の描画において3箇所で補完が行われます。

2つは、それぞれのモーションの姿勢を独立して作成する際の補完です。残りの一つは、それぞれのモーションの補完後の姿勢を、「内分」する際に行う補完です。

最初の2つの補完は、通常モーションの補完関数をそのまま使用しますが、最後の補完はモーションリンク独自のものです。実際、最初の2つの補完は、補完する場所を示すのに「フレーム番号 (frame)」を指定しますが、最後の補完は、「補完レイト (rate)」を指定することも違っていません。

この最後のモーションリンク専用の補完においては、回転要素の補完を元々の回転表現にかかわらず、必ずクォータニオン表現に変換してから行います。例えば、最初の2つの補完後の回転要素が共にXYZ形式のオイラー角表現によって与えられたときでも、両方をクォータニオン形式にしてから、クォータニオンの補完法を用いて、クォータニオン表現で回転要素を求めます。

このようにすることで、モーションリンクの結果が理想的になります。



9.8 モーションの補完法

この章では、モーションにおける補完法について説明します。

リニア補完、スプライン補完、クォータニオンの補完について説明します。

リニア補完は、平行移動要素、オイラー角表現の回転要素を始め、Scale 要素などで使用されます。

スプライン補完は、通常平行移動要素に対して使用されます。

クォータニオンの補完は、クォータニオン表現の回転要素に対して使用されます。

オイラー角表現の回転要素は、x、y、z 軸周りの回転角について独立して「近い方へのリニア補完」を行います。例えば、補完元での x 軸周りの回転角が 20 度、補完先で 80 度だとすると、x 軸周りの回転角は補完中 20→50→80 のように、80 - 20=60 度開いている部分が採用されます。これが、20 度と -20 度の場合であれば、20-(-20)=40 度開いている部分が採用されます。

9.8.1 リニア補完

モーションでリニア補完を行う場合、モーションのフレーム番号 (Float 型) frame の位置のトランスレート成分は次のように算出されます。

key1→keyframe1 <= frame < key2→keyframe1

$$u = \frac{\text{frame} - \text{key1} \rightarrow \text{keyframe}}{\text{key2} \rightarrow \text{keyframe} - \text{key1} \rightarrow \text{keyframe}}$$

を満たす隣接する二つのキーフレーム key1、key2 を探し、frame が key1、key2 の間のどの割合にあるかを u に計算します。

キーフレーム key1、key2 のトランスレート成分を用いて、補完後のトランスレート成分 pos[3] は次のように計算されます。

$$\begin{bmatrix} \text{pos}[0] \\ \text{pos}[1] \\ \text{pos}[2] \end{bmatrix} = (1-u) \begin{bmatrix} \text{key1} \rightarrow \text{key}[0] \\ \text{key1} \rightarrow \text{key}[1] \\ \text{key1} \rightarrow \text{key}[2] \end{bmatrix} + u \begin{bmatrix} \text{key2} \rightarrow \text{key}[0] \\ \text{key2} \rightarrow \text{key}[1] \\ \text{key2} \rightarrow \text{key}[2] \end{bmatrix}$$

オイラー角 (Angle) についても同様に行われます。

9.8.2 スプライン補完

モーションで用いるスプライン補完は Overhauser spline (Catnull-Rom spline) です。スプライン補完が使用される場合、モーションのフレーム番号(Float 型) frame の位置のトランスレート成分は次のように算出されます。

$$\text{key1} \rightarrow \text{keyframe1} \leq \text{frame} < \text{key2} \rightarrow \text{keyframe1}$$

を満たす隣接する二つのキーフレーム key1、key2 を探し、frame が key1、key2 の間のどの割合にあるかを u に計算します。

$$u = \frac{\text{frame} - \text{key1} \rightarrow \text{keyframe}}{\text{key2} \rightarrow \text{keyframe} - \text{key1} \rightarrow \text{keyframe}}$$

key0 を key1 より 1 つ手前のキーフレーム、key3 を key2 より 1 つ後ろのキーフレームとして、関係する 4 つのキーフレーム key0, key1, key2, key3 のトランスレート成分

$P_0[3], P_1[3], P_2[3], P_3[3]$ ($P_0 = \text{key0} \rightarrow \text{key}$, $P_1 = \text{key1} \rightarrow \text{key}$, $P_2 = \text{key2} \rightarrow \text{key}$, $P_3 = \text{key3} \rightarrow \text{key}$) を用いて、補完後のトランスレート成分 $\text{pos}[3]$ は次のように計算されます。

$$[\text{pos}[0] \quad \text{pos}[1] \quad \text{pos}[2]] = [u^3 \quad u^2 \quad u \quad 1] M_c \begin{bmatrix} P_0[0] & P_0[1] & P_0[2] \\ P_1[0] & P_1[1] & P_1[2] \\ P_2[0] & P_2[1] & P_2[2] \\ P_3[0] & P_3[1] & P_3[2] \end{bmatrix}$$

$$M_c = \begin{bmatrix} -s & 2-s & -2+s & s \\ 2s & -3+s & 3-2s & -s \\ -s & 0 & s & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

ただし、 $s = 0.5$

| | | |
|--------------------------------|-------------------|----------------|
| 参考文献： HEARN & BAKER | COMPUTER GRAPHICS | SECOND EDITION |
| DONALD HEARN, M. PAULINE BAKER | | 1994, 1986 |
| ISBN 0-13-161530-0 | | |
| P.322-325 Overhauser splines | | |

9.8.3 クォータニオンにおける補完

モーションでクォータニオンを用いる場合、モーションのフレーム番号 (Float 型) *frame* の位置のローテート成分は次のように算出されます。

key1 → *keyframe1* <= *frame* < *key2* → *keyframe1*

を満たす隣接する二つのキーフレーム *key1*、*key2* を探し、*frame* が *key1*、*key2* の間のどの割合にあるかを *u* に計算します。

キーフレーム *key1*、*key2* のクォータニオンを用いて、補完されたクォータニオン *Q* を次のように計算します。

$$u = \frac{frame - key1 \rightarrow keyframe}{key2 \rightarrow keyframe - key1 \rightarrow keyframe}$$

key1 → *key*[0] * *key2* → *key*[0] + *key1* → *key*[1] * *key2* → *key*[1] +

key1 → *key*[2] * *key2* → *key*[2] + *key1* → *key*[3] * *key2* → *key*[3]

の値が負ならば *s* = -1、正ならば *s* = 1 とし、

$$\begin{bmatrix} Q \rightarrow re \\ Q \rightarrow im[0] \\ Q \rightarrow im[1] \\ Q \rightarrow im[2] \end{bmatrix} = (1-u) \begin{bmatrix} key1 \rightarrow key[0] \\ key1 \rightarrow key[1] \\ key1 \rightarrow key[2] \\ key1 \rightarrow key[3] \end{bmatrix} + su \begin{bmatrix} key2 \rightarrow key[0] \\ key2 \rightarrow key[1] \\ key2 \rightarrow key[2] \\ key2 \rightarrow key[3] \end{bmatrix}$$

求められた *Q* を通常の4成分のベクトルとみなして単位ベクトル化します。

Q に対応する回転マトリックスを生成し、ローテート成分とします (njQuaternionEx 関数を使用)。

なお、クォータニオンを表現するベクトルの内積が負の場合に、*s* = - 1 とする理由は、「近い方へ」回転させるためです。もし、逆に、内積が正の場合に、*s* = - 1 とすれば、「遠い方へ」補完されます。もしこの判定を省略し常に *s* = 1 とすると、近い方へ回る場合と遠い方へ回る場合が出てきます。

9.9 モーションリンクとは

モーションリンクは、1つのオブジェクトツリーについての、2つの独立したモーションデータの間をさらに補完する機能です。

例えば、人のオブジェクトに対応する走るモーションM1と蹴るモーションM2が存在したとします。簡単のため、走るモーションM1には、その場で足踏みするようなモーションだけが入っていて、全体的な移動はプログラムで付けるようにしています。

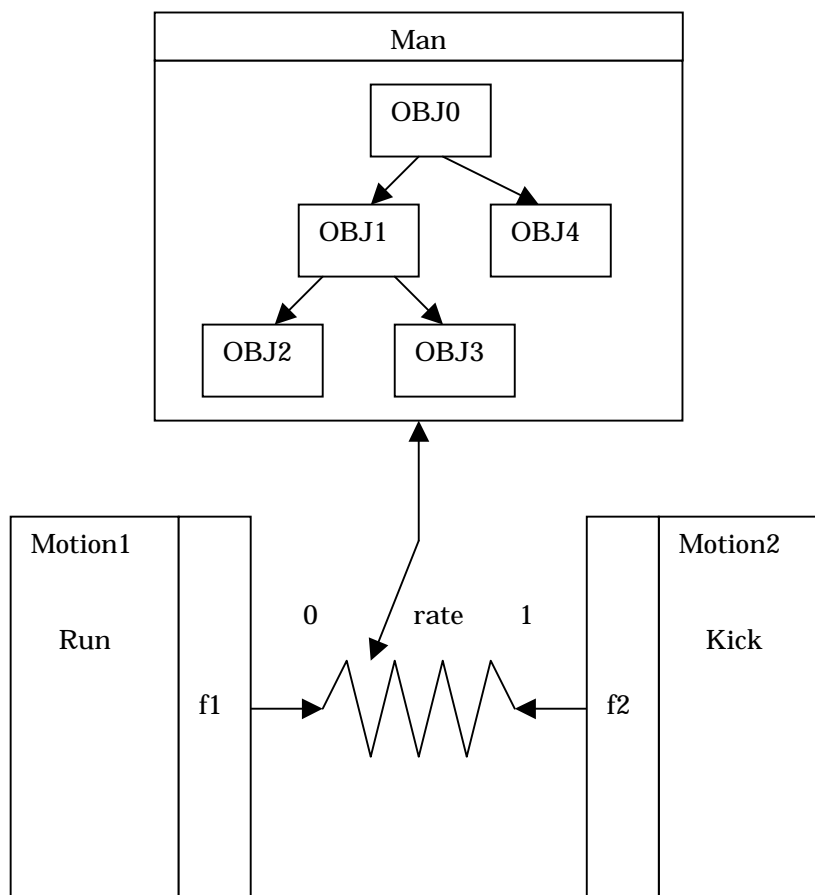
走っている際に、ユーザーが入力したタイミングで蹴る動作を行うことを考えましょう。このようなときにモーションリンクを使用します。モーションM1をフレームf1で描画中に突然、ユーザーが蹴りを行う入力をしたとすると、この状態でいきなり蹴るモーションM2を再生するわけには行きません。何もせずいきなりモーションM2を最初から再生すると、モーションが「ぶつ切れ」たように見えてしまうからです。

モーションリンクを使うと、走っているモーションM1のフレームf1の姿勢と、蹴るモーションM2のフレームf2の姿勢を、滑らかに繋ぐ(リンクする)ことができます。どのような割合でリンクするかを補完レート(rate)によって指定します。rate=0の時のリンク後の姿勢は、モーションM1のフレームf1の姿勢に一致します。rate=1の時のリンク後の姿勢は、モーションM2のフレームf2の姿勢に一致します。

モーションM1のノードiのフレーム番号f1における補完後の平行移動要素、回転要素、Scale要素をそれぞれ $pos1(i,f1)$ 、 $rot1(i,f1)$ 、 $scl1(i,f1)$ 、モーションM2のノードiのフレーム番号f2における補完後の平行移動要素、回転要素、Scale要素をそれぞれ $pos2(i,f2)$ 、 $rot2(i,f2)$ 、 $scl2(i,f2)$ 、補完レート $rate(0 \leq rate \leq 1)$ でモーションリンク後のノードiの平行移動要素、回転要素のクォータニオン表現、Scale要素をそれぞれ、 $pos(i,rate)$ 、 $qua(i,rate)$ 、 $scl(i,rate)$ とすると次のような関係があります。

$$\begin{aligned} pos(i,rate) &= f(LINEAR3, pos1(i,f1), pos2(i,f2), rate) \\ scl(i,rate) &= f(LINEAR3, scl1(i,f1), scl2(i,f2), rate) \\ qua(i,rate) &= f(QUATERNION, q(rot1(i,f1)), q(rot2(i,f2)), rate) \end{aligned}$$

ただし、
 $f(type, data1, data2, rate) = (data1 \text{ と } data2 \text{ を } rate \text{ の位置で、種類 } type \text{ の補完を行った値})$
 $q(rot) = (\text{回転要素 } rot \text{ のクォータニオン表現})$



9.10 高レベルモーション関数

9.10.1 高レベルモーション関数の分類

高レベルモーション関数には次の種類があります。

| 関数の種類 | 機 能 |
|--------------------------|--|
| シェープバッファアドレスの設定 | シェープバッファのアドレスを設定します。バッファは外部で予め確保し、アドレスだけを指定します。シェープやシェープリンクを用いる前には必ず指定しておく必要があります。 |
| (オブジェクト) モーション描画関数 | モーションデータに基づいてノード単位で姿勢を制御してオブジェクトツリー全体を描画します。グローバルエンベロープもこの関数で実行できます。 |
| シェープモーション描画関数 | オブジェクトモーションに加えて、頂点もモーションしてオブジェクトツリー全体を描画します。1 ノードのモデル自体が変形します。頂点座標の他に法線ベクトルもモーションデータとして与えておいてモーションさせます。 |
| (オブジェクト) モーションリンク描画関数 | 二つの(オブジェクト)モーションを結合したものを描画します。 |
| シェープリンク描画関数 | 二つのシェープモーションを結合したものを描画します。 |
| オブジェクトモーション取得関数 (旧仕様) | モーションデータに基づいて保管した姿勢データを全ノードにわたって一括して取得します。互換性のために残されていますが、回転要素をクォータニオンで与えている場合には、回転要素の種類の区別が付きません。オブジェクトの姿勢データを取得するには、低レベルモーション関数を使用して下さい。 |
| カメラモーションの実行 | カメラモーションデータに基づいてカメラを動かします。 |
| カメラモーションの取得 | カメラモーションデータに基づいてカメラが動くべき位置や方向を取得します。 |

9.10.2 高レベルモーション関数群

(1) モーション描画関数

```
void    njCnkEasyDrawMotion(  
        const NJS_CNK_OBJECT *object,  
        const NJS_MOTION *motion,  
        Float frame );
```

```
void    njCnkSimpleDrawMotion(  
        const NJS_CNK_OBJECT *object,  
        const NJS_MOTION *motion,  
        Float frame );
```

```
void    njCnkEasyMultiDrawMotion(  
        const NJS_CNK_OBJECT *object,  
        const NJS_MOTION *motion,  
        Float frame );
```

```
void    njCnkSimpleMultiDrawMotion(  
        const NJS_CNK_OBJECT *object,  
        const NJS_MOTION *motion,  
        Float frame );
```

(2) シェープモーション描画関数

```
void    njCnkEasyDrawShapeMotion(  
        const NJS_CNK_OBJECT *object,  
        const NJS_MOTION *motion,  
        const NJS_MOTION *shape,  
        Float frame );
```

```
void    njCnkSimpleDrawShapeMotion(  
        const NJS_CNK_OBJECT *object,  
        const NJS_MOTION *motion,  
        const NJS_MOTION *shape,  
        Float frame );
```

```
void    njCnkEasyMultiDrawShapeMotion(  
        const NJS_CNK_OBJECT *object,  
        const NJS_MOTION *motion,  
        const NJS_MOTION *shape,  
        Float frame );
```

```
void    njCnkSimpleMultiDrawShapeMotion(  
        const NJS_CNK_OBJECT *object,  
        const NJS_MOTION *motion,  
        const NJS_MOTION *shape,  
        Float frame );
```


(3) モーションリンク描画関数

```
void    njCnkEasyDrawMotionLink(  
        const NJS_CNK_OBJECT *object,  
        const NJS_MOTION_LINK *motionlink,  
        Float rate );  
  
void    njCnkSimpleDrawMotionLink(  
        const NJS_CNK_OBJECT *object,  
        const NJS_MOTION_LINK *motionlink,  
        Float rete );  
  
void    njCnkEasyMultiDrawMotionLink(  
        const NJS_CNK_OBJECT *object,  
        const NJS_MOTION_LINK *motionlink,  
        Float rate );  
  
void    njCnkSimpleMultiDrawMotionLink(  
        const NJS_CNK_OBJECT *object,  
        const NJS_MOTION_LINK *motionlink,  
        Float rate );
```

(4) シェープリンク描画関数

```
void    njCnkEasyDrawShapeMotionLink(  
        const NJS_CNK_OBJECT *object,  
        const NJS_MOTION_LINK *motionlink,  
        const NJS_MOTION_LINK *shapelink,  
        Float rate );  
  
void    njCnkSimpleDrawShapeMotionLink(  
        const NJS_CNK_OBJECT *object,  
        const NJS_MOTION_LINK *motionlink,  
        const NJS_MOTION_LINK *shapelink,  
        Float rate );  
  
void    njCnkEasyMultiDrawShapeMotionLink(  
        const NJS_CNK_OBJECT *object,  
        const NJS_MOTION_LINK *motionlink,  
        const NJS_MOTION_LINK *shapelink,  
        Float rate );  
  
void    njCnkSimpleMultiDrawShapeMotionLink(  
        const NJS_CNK_OBJECT *object,  
        const NJS_MOTION_LINK *motionlink,  
        const NJS_MOTION_LINK *shapelink,  
        Float rate );
```

(5) オブジェクトモーションデータ列の取得 (旧仕様)

```
void    njGetDrawMotion(  
        const NJS_CNK_OBJECT *object,  
        const NJS_MOTION *motion,  
        NJS_MOTION_DATA *data,  
        Float frame );
```


(6) カメラモーションの実行

```
void    njCameraAction(
        const NJS_CACTION *caction,
        Float frame );

void    njCameraMotion(
        const NJS_CAMERA *camera,
        const NJS_MOTION *motion,
        Float frame );
```

(7) カメラモーションの取得

```
void    njGetCameraAction(
        const NJS_CACTION *caction,
        NJS_CMOTION_DATA *data,
        Float frame );

void    njGetCameraMotion(
        const NJS_CAMERA *camera,
        const NJS_MOTION *motion,
        NJS_CMOTION_DATA *data,
        Float frame );
```

9.10.3 高レベル関数群での NULL の指定

シェープモーションだけで必要でオブジェクトモーションが必要ない場合にも、シェープ描画関数を使用します。簡単に使用できるようにするために、引数 motion などに NULL を指定することが可能です。

- 例:

```
njCnkEasyDrawShapeMotion( object, NULL, shape, frame );
```

このように、一部のモーションを省略したい場合は、引数に NULL を指定する方法のほかに、次のようなダミーのモーションデータを与えることによる方法もあります。

- 例:

```
NJS_MOTION  DummyMotion = {
    NULL,           //mdata
    0,              //nbFrame
    0,              //type
    0               //inp_fn
};

njCnkEasyDrawShapeMotion( object, &DummyMotion, shape, frame );
```

- 高レベルモーション関数の詳細

オブジェクト描画関数における motion 引数には、NULL が指定できます。NULL を指定すると、単なるオブジェクト描画関数になります。

シェープ描画関数における motion 引数には、NULL が指定できます。NULL を指定すると、オブジェクトモーションをせずに、頂点だけを動かし、モデルの変形だけを行って描画します。

シェープ描画関数における、shape 引数には NULL が指定できます。NULL を指定すると、シェープモーション描画関数は、単なるモーション描画関数になります。

シェープリnk描画関数における、shape_link 引数には NULL が指定できます。NULL を指定すると、単なるモーションリンク描画関数になります。

モーションリンク描画関数や、シェープリnk描画関数において、motion_link 引数によって指定される NJS_MOTION_LINK 構造体の中の motion[0]、motion[1]メンバを NULL に指定することができます。NULL を指定すると、指定した方のモーションに NJS_OBJECT 構造体に指定されている静止姿勢が付いている場合に相当します。

モーションリンク描画関数や、シェープリnk描画関数において、motion_link 引数を NULL にすることはできません。

シェープリnk描画関数において、shape_link 引数によって指定される NJS_MOTION_LINK 構造体の中の motion[0]、motion[1]メンバを NULL に指定することはできません。

9.10.4 高レベル関数群での留意事項

(1) njGetDrawMotion()関数（旧仕様）

njGetDrawMotion()関数は、全ノードの姿勢データを一括して取得します。しかしデータを取得する構造体が、旧使用のままクォータニオンに対応していないため、回転要素がクォータニオンで与えられている場合などには、適切にデータを返しません。姿勢データを取得したい場合には、低レベルモーション関数をご使用下さい。

(2) njGetCameraMotion()、njGetCameraAction()関数

njGetCameraMotion() 関数や、njGetCameraAction()関数は、モーション後にカメラが移動すべき、位置や向き情報を、NJS_CMOTION_DATA 構造体に返します。

しかしながら、これについては NJS_CAMERA 構造体にした方がすっきりとまとまりますので、今後改良される可能性があります。

9.11 低レベルモーション関数

9.11.1 低レベルモーション関数の分類

低レベルモーション関数には、次の種類があります。

| 関数の種類 | 機 能 |
|------------------------|--|
| 低レベルモーションコア関数 | 最小単位のモーションの初期化関数、ノード更新関数です。オブジェクトモーションやシェープモーション、カメラモーションなどの用意されている関数以外に、独自にモーションモジュールを利用したい場合に使用します。低レベルオブジェクトモーション関数や、カメラモーション関数などもこれらの関数を使用しています。実際に独自にモーションモジュールを使用する場合は、Float*1 補完関数や、Float*3 補完関数、Angle*3 補完関数など、ドキュメントレベルではなく、ソースレベルで公開されている関数を使用する必要があります。 |
| 低レベルオブジェクト（ノード）モーション関数 | オブジェクトモーションを行って、個々のノードの姿勢データを取得したり、個々のノードの姿勢に対応するマトリックスを作成します。 |
| 低レベルモーションリンク関数 | モーションリンクを行って、個々のノードの姿勢データを取得したり、個々のノードの姿勢に対応するマトリックスを作成します。 |
| 低レベルシェープモーション関数 | シェープモーションを行って、変形後の頂点座標列の格納された VLIST を取得します。 |
| 低レベルシェープリンク関数 | シェープリンクを行って、変形後の頂点座標列の格納された VLIST を取得します。 |

9.11.2 低レベルモーション関数群

(1) 低レベルモーションコア関数

```
void njStartMotionEx(const NJS_MOTION *motion, Float frame );
void njSetNextMotionNodeEx( void );
void njSetCurMotionInfoSlot( int Slot );
```

(2) 低レベルオブジェクト（ノード）モーション関数

```
void njStartMotionObj( const NJS_MOTION *motion, Float frame );
void njStartMotionObjEx( const NJS_MOTION *motion, Float frame );
```

```
int njGetMotionNodeData(
const NJS_CNK_OBJECT *cnkobj,
Float pos[3], Angle ang[3], Float scl[3],
NJS_QUATERNION *qua );
```

```
void njMotionTransformEx( const NJS_CNK_OBJECT *cnkobj );
```


(3) 低レベルモーションリンク関数

```
void njStartMotionLink( const NJS_MOTION_LINK *motion_link, Float rate );
```

```
int njGetMotionLinkNodeData(  
const NJS_CNK_OBJECT *cnkobj,  
Float pos[3], Angle ang[3], Float scl[3],  
NJS_QUATERNION *qua );
```

```
void njMotionLinkTransformEx( const NJS_CNK_OBJECT *cnkobj );
```

```
void njSetNextMotionLinkNode( void );
```

(4) 低レベルシェープモーション関数

```
void njSetCurShapeInfoSlot( int Slot );
```

```
void njStartShapeEx( const NJS_MOTION *shape, Float frame );
```

```
void njSetNextShapeNodeEx( void );
```

```
void njGetShapeNodeData( const NJS_CNK_MODEL *cnkmodel, Sint32 **pVlist );
```

(5) 低レベルシェープリンク関数

```
void njStartShapeLink( const NJS_MOTION_LINK *shape_link, Float rate );
```

```
void njGetShapeLinkNodeData(  
const NJS_CNK_MODEL *cnkmodel,  
Sint32 **pVlist );
```

```
void njSetNextShapeLinkNode( void );
```


9.12 モーション関数の変更点について

シェーブモーションとエンベロープを、同時に使用することが可能です。

`njSetCurrentMotion()`関数は、関数名が `njStartMotionObj()`に変更になりました。仕様の詳細については、関数リファレンスを参照して下さい。

Angle 系の補完は、近い方向への補完を行います。これは、Ninja 1 での「Old 関数」に相当するものです。Ninja 1 での「New 関数」は削除されました。

理由としては、二つあります。

一つ目の理由は、キーフレームの適切に用意すれば、New 関数は必要なく、むしろ Old 関数の方が便利であることです。二つ目の理由は、アルゴリズムの改良により、Old 関数の補完速度が New 関数と同レベルになったためです。

モーションリンクにおいて、2つのモーションを結合する際の、Angle 同士の補完は、一度それぞれをクォータニオンに変換してから、クォータニオンで補完を行うようになりました。このことにより、モーションリンクが非常に理想的に行われるようになりました。

モーション描画関数、シェーブ描画関数、モーションリンク描画関数、ノードデータ取得系の関数、ノードマトリックス作成系関数のすべてについて、モーション部のコードが共有されるようになりました。これにより、描画関数の違いによるモーション機能の違いや、モーションとモーションリンク、描画系と取得系等の相互間による違いが発生する可能性は全くなくなりました。

9.13 新カメラモーション関数

9.13.1 新カメラモーション関数と旧カメラモーション関数の違い

カメラモーション関数のプロトタイプ宣言は、一部の引数に `const` 属性が付与された以外は変更されていません。

カメラモーション関数は、新カメラ関数群を使って組み込まれていますが、基本的な使い方に変更はありません。新カメラ関数群については別途説明しています。

旧カメラモーション関数では、マトリックススタックのスタックポインタがベース位置に初期化されていましたが、新カメラモーション関数ではマトリックススタックポインタなどを初期化せずに、カレントマトリックスの左から新しく作成した行列を乗算します。

昔の `njCameraMotion` 関数と同様に使用するには、呼び出す前に、`njUnitMatrix(NULL);` などを行って、関数を呼び出す直前のカレントマトリックスを単位行列にする必要があります。

マトリックスポインタをベース位置に戻す関数は用意されていないので、`njPushMatrix` を行ったときは、必ず同じ回数だけ `njPopMatrix` を行ってください。

`njCameraMotion()`の第一引数の `camera` 構造体は、`motion` 中にキーフレームデータ列が存在しない場合のデフォルト値として使用されます。読み込まれるだけで書き込みは行われません。

9.13.2 カメラモーション関数の使用例

```
#define CAMERA          camera1;
#define CAMERA_MOTION  camera_motion1;
#define ENEMY1_MODEL   model_enemy1;
#define ENEMY1_MOTION  motion_enemy1;

NJS_CAMERA      CAMERA;
NJS_MOTION      CAMERA_MOTION;
NJS_MODEL       ENEMY1_MODEL;
NJS_MOTION      ENEMY1_MOTION;

void DrawFunc( Float frame )
{
    njUnitMatrix( NULL );
    njCameraMotion( &CAMERA, &CAMERA_MOTION, frame );

    njPushMatrixEx();
    njTranslate( NULL, ex, ey, ez );
    njRotateY( NULL, eangy );
    njCnkEasyDrawMotion( &ENEMY1_MODEL, &ENEMY1_MOTION, frame );
    njPopMatrixEx();
}
```

9.14 ライトモーション関数

9.14.1 ライトモーション関数の使用例

```
#define LIGHT          light1;
#define LIGHT_MOTION  light_motion1;
#define ENEMY1_MODEL   model_enemy1;
#define ENEMY1_MOTION  motion_enemy1;

extern NJS_LIGHT      LIGHT;
extern NJS_MOTION     LIGHT_MOTION;
extern NJS_MODEL       ENEMY1_MODEL;
extern NJS_MOTION     ENEMY1_MOTION;

void DrawFunc( Float frame )
{
    njLightMotion(NJD_CNK_EASY_MULTILIGHT_1,&LIGHT,&LIGHT_MOTION, frame );
    njCnkSetEasyMultiLightSwitch(NJD_CNK_EASY_MULTILIGHT_ALL, 1);

    njPushMatrixEx();
    njCnkSetEasyMultiLightMatrices();
    njTranslate( NULL, EneX, EneY, EneZ );
    njCnkEasyMultiDrawMotion(&ENEMY1_MODEL, &ENEMY1_MOTION, frame );
    njPopMatrixEx();
}
```

9.14.2 マルチライトモーション関数の使用例

```
#define LIGHT_NUM      (3)
extern NJS_LIGHTlight0;
extern NJS_LIGHTlight1;
extern NJS_LIGHTlight2;
extern NJS_MOTION      lmotion0;
extern NJS_MOTION      lmotion1;
extern NJS_MOTION      lmotion2;

NJS_MLIGHT_MOTION mlight;
NJS_LIGHT      *lights[LIGHT_NUM];
NJS_MOTION      *lmotions[LIGHT_NUM];

Sint32 UserInit(void)
{
    lights[0] = &light0;
    lights[1] = &light1;
    lights[2] = &light2;

    lmotions[0] = &lmotion0;
    lmotions[1] = &lmotion1;
    lmotions[2] = &lmotion2;

    mlight.n = LIGHT_NUM;
    mlight.lights = lights;
    mlight.motions = lmotions;

    return USER_CONTINUE;
}

Sint32 UserMain(void)
{
    njMultiLightMotion( &mlight, frame);

    njPushMatrixEx();
    njCnkSetEasyMultiLightMatrices();
    njTranslate( NULL, 0.f, 0.f, -10.f );
    njRotateXYZ( NULL, xx,yy,zz );
    njCnkEasyMultiDrawModel( model );
    xx += 257;
    yy += 179;
    zz += 193;
    njPopMatrixEx();
    return USER_CONTINUE;
}
```


10 Njutil ライブラリ

Ninja2 で GD にアクセスするテクスチャ関数が削除され、同等の関数が Njutil ライブラリに移動しました。（一部都合上、GD アクセス関数以外も移動したものもあります）

Njutil ライブラリは、ソースコードを公開しています。

10.1 Njutil ライブラリのテクスチャ関数

| Njutil 関数名（追加） | 旧 Ninja 関数名（削除） |
|----------------------------------|----------------------------------|
| nuInitTextureBuffer | njInitTextureBuffer |
| nuGetTextureBuffer | |
| nuLoadTexture | njLoadTexture |
| nuLoadTextureNum | njLoadTextureNum |
| nuLoadTexturePvmFile | njLoadTexturePvmFile |
| nuLoadTexturePvmMemory | njLoadTexturePvmMemory |
| nuSetPvmTextureList | njSetPvmTextureList |
| nuLoadTextureReq | njLoadTextureReq |
| nuLoadTextureReqStop | njLoadTextureReqStop |
| nuGetLoadTextureReqMode | |
| nuLoadTextureSetG | njLoadTextureSetG |
| nuReLoadTextureNum | njReLoadTextureNum |
| nuReLoadTextureNumG | njReLoadTextureNumG |
| nuReLoadRectangleTexturePartNum | njReLoadRectangleTexturePartNum |
| nuReLoadRectangleTexturePartNumG | njReLoadRectangleTexturePartNumG |
| nuReLoadVQCodebookNum | njReLoadVQCodebookNum |
| nuReLoadVQCodebookNumG | njReLoadVQCodebookNumG |
| nuLoadPaletteFile | njLoadPaletteFile |
| nuTexFopen | |
| nuTexFclose | |
| nuTexFseek | |
| nuTexFread | |
| nuTexFreadReq | |
| nuTexFreadAll | |

11 高速化&テクニック

ここでは、アプリケーションの高速化とテクニックについて説明します。

11.1 メモリバンク

メインメモリは4 MB × 4 バンクで構成されているので、データの配置をそれにあわせて配置すると、性能を上げることができます。

例えば、モデルデータと中間バッファのバンクを別々にすると、性能は若干向上します。

下の表はバンクとキャッシュラインの違いによるメモリ性能比です。

メモリ転送スピード (MB / S)

| | 同バンク | | 別バンク | |
|------------|------|------|------|------|
| | 同ライン | 別ライン | 同ライン | 別ライン |
| memcpy | 6 | 52 | 9 | 55 |
| 1 バイト | 6 | 60 | 9 | 64 |
| 2 バイト | 12 | 88 | 18 | 97 |
| 4 バイト | 24 | 115 | 35 | 130 |
| 8 バイト | 46 | 135 | 65 | 155 |
| 16 バイト | 80 | 138 | 102 | 159 |
| 32 バイト | 145 | 145 | 168 | 168 |
| 32 バイト C A | 233 | 233 | 398 | 398 |
| 32 バイト S Q | 262 | 262 | 486 | 486 |

同バンク コピー元とコピー先が同じ SDRAM バンク
 別バンク コピー元とコピー先が別の SDRAM バンク
 同ライン コピー元とコピー先が別アドレスの同じキャッシュライン
 別ライン コピー元とコピー先が別のキャッシュライン

11.2 PLIST の最適化

11.2.1 マテリアル調整

基本的に描画関数ごとに必要なマテリアルが異なるため、必要なマテリアルのみをセットすれば性能とデータの効率が上がります。

例えば、CnkEasy でテクスチャをスペキュラなしで描画する場合、マテリアルは必要ありません（半透明の場合はディフューズのアルファを使います）。

11.2.2 テクスチャ、アトリビュートの連続

テクスチャ ID や各種アトリビュートチャンクは、設定を変更しない限り状態を保持します。

そのため、同じテクスチャで複数のモデルを描画する場合、最初にテクスチャ ID を設定すれば、そのあとのモデルには設定の必要がありません。

この設定を行うには、テクスチャやアトリビュートのみをセットするモデルを用意すると便利です。半径を 0、VLIST を NULL にし、PLIST にデータをセットします。

このセットしたモデルをオブジェクトツリーの最初に設定すると、テクスチャやアトリビュートのみをセットしますので、実際の描画を行うモデルにはそのデータをセットする必要がありません。

11.3 ディセーブルニアクリップ (Disable Near Clip)

基本的に、ニアクリップは大変重い処理です。しかし、ポリゴンが細かい場合や視線から遠い場合などの、厳密にポリゴンを分割しなくても問題ない場合があります。

その場合には、NJD_CONTROL_3D_DISABLE_NEAR_CLIP を設定すると、ニアに掛かるポリゴンの分割処理を行わないため、ポリゴンごと描画なくなります。その結果、ニアクリップ処理が軽減されます。

11.4 イーバルクリップ (Eval Clip)

イーバルクリップとは、オブジェクトツリーでのクリッピング処理のことをいいます。

オブジェクトのイーバルフラグに NJD_EVAL_CLIP がセットされている状態で、そのオブジェクトのモデルにモデルクリップの処理がされた場合、チャイルド（子供）のトレースを行わなくなります。

一番の親に、その子供全体を含む中心と半径を設定し子供の設定を半径 0 にすれば、その親だけのクリッピング処理だけで済みます。

11.5 頂点参照

- 親子で頂点をつなぐ場合

この場合は、子供が親の頂点を参照してつなぐことになります。その場合、子供の頂点チャンクは、親が生成した中間バッファを破壊しないように、オフセットを付けて中間バッファを生成します。子供のポリゴンチャンクで、親のインデックス番号を参照したストリップを作成します。

その際に、問題となるのがクリッピングです。

モデルがわかれているため、親とモデルをつなごうと思っても親同士がクリッピングされてしまい、データを生成していないためたらめなポリゴンを生成してしまいます。

それを防ぐには、親モデルにクリッピングがされないように設定しなければなりません。

方法としては2種類が考えられます。

- (1) 親のモデルの半径を0にする方法

この方法は最も簡単な方法ですが、必ず頂点の演算処理を行ってしまいます。

- (2) 親のモデルの中心と半径を子供まで含めた大きさにしてから EVAL_CLIP を設定する方法

この方法は効率の良い方法ですが、子供まで含めたモデルのクリップエリアを設定するのが難しくなります。

- 子供の方がクリッピングされる場合

この場合にも、モデルクリッピングを行わないように、半径を0にしなければなりません。

更に、頂点演算ですべてがクリッピングされていると判断しないように、頂点チャンクのフラグに NJD_FV_CONT を設定します。

この設定は、頂点数とクリップされた頂点数の比較をそのモデル単体で行わず、前回の結果を引き継ぐ設定です。

- 親も子もすべてクリッピングされた場合

頂点数の合計とクリッピングされた頂点数の合計は同じになりますので、その時点でモデルの plist の処理は行いません。

結論として頂点をつなぐ方法は、以下の2種類となります。

- 親と子供のモデル半径を0にし、子供の頂点チャンクフラグに EVAL_CLIP を設定する。
- 親は、子供も含めたクリップエリアにし EVAL_CLIP を設定する。子供のモデル半径は0にし、子供の頂点チャンクフラグに NJD_FV_CONT を設定する。

11.6 テクスチャ合成

1つのモデルに対して複数のテクスチャを貼る場合は、PLIST に対しテクスチャ ID が複数必要となるために性能が悪くなります。

1枚のテクスチャにまとめて、UV のみでコントロールするようにすると、ストリップチャンク 1 つで描画できるため、大きく効率が上がります。そのためには、むやみにテクスチャの種類を増やさないように注意して下さい。

基本的に、PLIST にあるチャンクの数減らすことは、最も効果の上がる方法です。

11.7 レンダリング特性

レンダリングの性能については、不透明と半透明で大きな差があります。

場合によっては、半透明で描画するよりもモデルで形を作ったほうが、速いこともあります。

注 意 半透明のモディファイアは、描画処理がかなり遅いので、なるべく使用しないで下さい。モディファイアポリュームの画面上での面積よってもかなり影響されます。

実際には、ポリゴンとの交差の有無に関わらず性能に影響しますので、描画範囲を絞って使用して下さい。

11.8 プリフェッチ

データを先読みするプリフェッチ命令を、SHC で記述することができます。
この機能をうまく使用すれば、データのアクセスを効率よく上げることが可能です。

12 質問と回答（Q&A）及び対処法

12.1 モーションについて

【質問】

void njCnkGetMotionRotate(NJS_CNK_OBJECT *cnkobj, Angle *ang);関数など、モーションデータを要素単位で取得する関数が存在しません。

【回答】

njGetMotionNodeData()関数を使用して下さい。

プロトタイプ宣言は以下のようになっています。

```
int njGetMotionNodeData( NJS_CNK_OBJECT *cnkobj,
                          Float pos[3], Angle ang[3], Float scl[3],
                          NJS_QUATERNION *qua );
```

この関数では、pos、ang（またはqua）、sclを同時に取得します。

回転情報をangとquaのいずれに返したか、また、angに返した場合の回転順序がXYZかZXYかは、戻り値によって区別できます。

Pos、ang、sclなど個別の要素を独立して取得する関数は、次の理由から削除しました。

通常、モーションの情報を取得する場合、位置、回転、拡大縮小情報は組になって取得したい場合が多いと考えられます。

以前のposやangを個別に取得する関数は、モーションのデータ構造をよく知っていないと使えない仕様でした。例えば、pos要素がキーにあるかないかをユーザ自身がチェックしてからpos要素がある場合にだけ、pos取得関数を呼ぶ必要がありました。

ang要素だけを取得したい場合でも、pos要素があれば、pos取得関数を呼ぶ必要がありました。また、cnkobj->evalflagsにNJD_EVAL_SKIPフラグがセットされているノードについては、それらの関数を一切呼んではならないような決まり事がありました。新しい取得関数はこれらのことがらを全く知らなくても使用できるように作成されています。

回転要素について、オイラー角（Angle形式）で与えられる場合と、クォータニオン形式で与えられる場合などのいくつかの種類がありましたが、以前の関数仕様の場合は、どの形式で与えられているかをユーザがあらかじめ調査し、それぞれの形式に応じて別個の関数を呼ぶ必要がありました。

これは、Ninjaのツール上で作成したデータをそのまま使用できるという設計思想とはあまりなじまない仕様でした。新しい取得関数では、データをあらかじめ予想せず、関数呼び出しの後にデータの種類を特定できるように組まれています。

こういった仕様は、今後のモーションデータ構造の変更の際にも、ライブラリユーザのコード変更を最小限に抑えられる点で有利です。

用語集

Ninja2 ライブラリ編

| 日本語 | 英語 | 内容 |
|-------------|---------------------|--|
| アスペクト比 | Aspect ratio | CRT モニタの最小表示単位（ピクセル）の縦横比 |
| アトリビュート | Attribute | サーフェースの可視表現に影響を与える品質または性質（カラー、映り込み度、透明度、材質など） |
| オブジェクト | Object | 複数のモデルの階層構造を定義したデータ構造 |
| 環境マッピング | Environment Mapping | 無限に大きな球や立方体を考え、その内側に張られた2次元画像が反射しているようにマッピングすること |
| 型変換 | Type conversion | 変数の型を変えること |
| キーフレーム | Key Frame | CG アニメーションでは、ユーザーがキーフレームを使って特定のタイムにおけるアイテムの状態（サイズ、位置など）を指定し、システムが自動的にフレームを作成すること |
| キャスト | Cast | プログラミング言語で型変換を指定すること |
| クリッピング | Clipping | 定義された境界の外に出ているグラフィック・イメージの一部を取り除くこと |
| クォーターニオン | Quaternion | 4元数 |
| コリジョン | Collision | オブジェクトとオブジェクトの衝突判定 |
| コンテキスト | Context | 描画する際に使用するパラメータ |
| サーフィス | Surface | 3次元オブジェクトの境界 |
| 座標系 | Coordinate System | 空間内のポイントの位置を記述する方法。各座標系が、グラフィックス・プロセスの特定の段階における画像の値を記述するために使われる |
| スプライト | Sprite | 図形パターンを画面に表示すること |
| スクロール | Scroll | データのすべてを一画面にすべて表示できない場合に、表示画面を上下左右に移動して残りの部分を表示する機能 |
| シェーディング | Shading | 3次元図形の面の向きと光線の方向で定まる明度で、陰影を付けること。これによって奥行きが出てくる。同一色で塗りつぶすコンスタント・シェーディング、輝度や色を微妙に変えて曲面らしさを出すスムーズ・シェーディングがある |
| シーン | Scene | コンピュータの内部に構築される風景。物体の形状や配置、物体表面の質感、カメラ、ライトなどの数値的な情報から成り立ち、これらの情報をもとにレンダリングが行われる |
| シーケンス | Sequence | 頂点定義からレンダリング等のフロー制御や、一連の処理の流れ |
| セルスプライト | Cell sprite | 複数のセルを一括し動作、描画するもの |
| ストリップ | Strip | ストリップ法 |
| チープモディファイア | Cheap Modifier | モディファイアの影響を輝度の変化のみにして、簡単に描画できるようにしたもの |
| テクスチャ | Texture | 次元画像を3次元形状の表面に壁紙のように張り付けること |
| テクスチャ・マッピング | Texture Mapping | 2次元画像を3次元形状の表面に壁紙のように張り付けることによって、物体表面の模様などを表現する手法 |
| トレース | Trace | デバッグの手段で、プログラムの個々のステップの動きや結果を記録して表現したもの |
| トグル | Toggle | ON/OFF を切り替えるモードまたはボタン |
| パーティクル | Particle | 雲や液体など、形状の定まらない物体を微少な粒子。一連の座標データを一括して描画するもの |
| バーテックス | Vertex | 3次元空間の点または頂点 |
| バーテックス・バッファ | Vertex Buffer | ハードウェアに送るためのパラメータを保持するバッファ |
| フリッカーフリー | Flicker free | ディスプレイ上での画面のちらつきが起きないような表示画面 |
| フォグ | Fog | グラフィックスでの基本的な機能で、オブジェクトに霧がかかった |

| 日本語 | 英 語 | 内 容 |
|-------------|--------------|---|
| | | ように表示する技法。遠近感を出すため、遠くの方ほど霧が濃くなってオブジェクトが薄く見えるようにする |
| フラットシェーディング | Flat shading | ある区域を塗りつぶすとき、同じ色で一様に塗りつぶすこと。ポリゴンに対してスムーズシェーディングせずに、平らなようにシェーディングする方法 |
| フラグ | Flag | プログラム実行中に特定の条件が成立したかどうかを表す変数で、プログラマーが定めるものをいう |
| プリミティブ | Primitive | モデリングやレンダリングの際に使われる基本形状。プリミティブを組み合わせていくことによって複雑な形状を作成する |
| ポリゴン | Polygon | 3Dグラフィックスの基本単位とされる多角形面。レンダリングなどでは最も基本となる形状。各頂点の3次元情報を持つため、立体図形を描いた場合に、同じ図形を違った角度から観察することが簡単にできる。使用するポリゴンの量が増えるほど微細な表現が可能になるが、演算処理も増大となる |
| モデル | Model | 複数のポリゴンで構成された形あるものの基本単位(Ninja2 での描画の基本単位)。現実の状態を数値的あるいは図形的に表現したもの |
| モーション | Motion | 物体の動き |
| モーションリンク | Motion Link | 1つのオブジェクトツリーについての、2つの独立したモーションデータの間をさらに補完する機能 |
| モディファイア | Modifier | 影などを立体的に描画するためのエリアを指定するものをいう |
| モデリング | Modeling | 物体の形状データなどを構築すること。モデリングの過程を大きく分けると、形状モデリング、質感モデリング、モーションモデリングとなる |
| レイトレーシング | Ray Tracing | 光の進んでいく様子を目の方から逆にたどっていく方法 |
| レイテンシ | Latency | データのリクエストを行ってから、実際にデータが転送されるまでにかかる遅延時間のことを指す |
| レンダリング | Rendering | データから最終出力を作成するプロセスの及びイメージまたは画像を作り出すプロセス。シェーディング、照明、深さなどを追加して、イメージをリアルに描くことができる |
| 3D図形処理 | | 3Dデータを扱ったグラフィックスをいう。辺のみを計算して表示するワイヤー・フレーム・モデル、面の情報を付け加えたサーフェイス・モデル、さらに内部の情報が加わったソリッド・モデルの3種類がある |