GOM Format quick Reference

GOM + MTA フォーマット 簡易仕様書

Game Oriented Model Format + Motion TimeLine Animation

バージョン: GOM v1.0 / MTA v1.0
最終更新: 2026-03-18

☒ 目次

概要

GOM（Game Oriented Model）

目的: FBX等のDCCデータをゲーム実行向けに最適化し、オフライン変換して保存する

設計目標:

- スキニングはモデル（geometry）空間で完結
- インスタンスごとに別Transform・別アニメ時間を安全に扱える
- FBX差（ノード階層・geometry transform）を吸収
- ワールド直スキニング事故を防止
- 実行時にFBXSDKに依存しない
- 同一FBXから常に同一GOMが生成される（決定性保証）

MTA（Motion TimeLine Animation / GOMTA）

目的: GOMのアニメーション部分のみを独立させた兄弟フォーマット

特徴:

- メッシュ/マテリアルを含まない
- スケルトン実体を含まない（skeletonSig のみ）
- 複数のアニメーションファイルを用意して動的に切り替え可能
- 異なる GOM ファイルでも同じスケルトンなら適用可能

GOM フォーマット

ファイル形式

形式	マジックナンバー	説明
テキスト	`GOMT0001`	UTF-8、構造リテラル形式
バイナリ	`GOMB0001`	Little Endian、16-byte アライン

含まれる要素

- **Skeleton（SKEL）**: ボーン階層、bind pose
- **Mesh（MSH0）**: 頂点、インデックス、サブメッシュ
- **Material（MATL）**: Phong最小セット、テクスチャ参照
- **Animation（ANIM/CLP0）**: ボーンアニメーション
- **String Table（STR0）**: 文字列テーブル（名前、パス）

ファイルサイズ

- ヘッダ: 512 bytes
- チャンク: 可変長（16-byte アライン）

MTA フォーマット

ファイル形式

形式	マジックナンバー	説明
バイナリ	`MTAB0001`	Little Endian、16-byte アライン

注意: v1 ではテキスト形式（MTAT0001）は非サポート。詳細は テキストフォーマット セクション参照。

含まれる要素

- **String Table（STR0）**: 文字列テーブル（必須）
- **Animation（ANIM/CLP0）**: アニメーションクリップ
- **skeletonSig**: スケルトン互換性チェック（SHA-256, 32 bytes）

除外される要素

- ☒ メッシュ（MSH0, VB0, IB0, SUB0, IBND）
- ☒ マテリアル（MATL）
- ☒ スケルトン実体（SKEL）
- ☒ Geometry Transform（GBTM）

スケルトン互換性チェック

実行時判定:

```
if (memcmp(gom.skeletonSig, mta.skeletonSig, 32) != 0) {
  error("Skeleton signature mismatch!");
}
```

skeletonSig 生成:

```
SHA-256(
  boneCount (uint32, little-endian) +
  foreach boneIndex in bonePath 辞書順:
    bonePath (UTF-8) + '\0' + parentIndex (int32, little-endian)
)
```

基本規約

空間設計ポリシー（Invariant A）

GOM の最重要設計原則。

Invariant A: 頂点空間

- 頂点は常に **geometry（=モデルローカル）空間**
- スキニング結果も **geometry 空間**
- `instanceWorld` は **描画直前に1回のみ適用**

禁止事項:

- ☒ スキニング途中で world に焼く
- ☒ geometry_to_world を途中適用する

描画時の変換式:

```
// 正しい変換順序
vec4 skinnedPos = vertex * skinning;          // geometry 空間
vec4 worldPos = skinnedPos * GBTM * instanceWorld;  // world 空間
vec4 clipPos = worldPos * viewProj;           // clip 空間
```

この設計の利点:

1. **インスタンシング安全**: 同一メッシュを異なる位置/回転で配置可能
2. **アニメーション独立**: インスタンスごとに異なるアニメ時間を適用可能
3. **FBX差の吸収**: ノード階層の違いを完全に隠蔽
4. **事故防止**: ワールド直スキニングによる破綻を防止

行列規約

項目	値
数値型	`float32`
メモリ並び	**row-major**
演算規約	**row-vector 右掛け**（`p' = p * M`）
行列サイズ	4x4（常に）

行列演算の具体例:

```
// ベクトルは行ベクトルとして扱う
vec4 p = {x, y, z, 1.0f};
vec4 p_transformed = p * M;  // 右から掛ける

// 複数の変換
vec4 result = p * M1 * M2 * M3;  // 左から順に適用
```

メモリレイアウト（row-major）:

```
float m[16] = {
  m00, m01, m02, m03,  // 第1行
  m10, m11, m12, m13,  // 第2行
  m20, m21, m22, m23,  // 第3行
  m30, m31, m32, m33   // 第4行
};

// アクセス方法
float value = m[row * 4 + col];
```

変換順序の例:

```
// スキニング → GBTM → インスタンス変換 → ビュー → プロジェクション
vec4 worldPos = vertex * skinning * GBTM * instanceWorld;
vec4 viewPos = worldPos * view;
```

```
vec4 clipPos = viewPos * projection;

// または一度に
vec4 clipPos = vertex * skinning * GBTM * instanceWorld * view * projection;
```

座標系

項目	値
ハンドネス	**Left-handed**
上方向	**Y-up**
単位	**meters**

バイナリ規約

項目	値
エンディアン	**Little Endian**
アラインメント	**16-byte**
パディング値	**0x00**

ファイル構造

GOM/MTA バイナリファイル構造

GOM ファイル構造

```
+------------------------+ offset 0
| Header (512 bytes)     |
+------------------------+ offset 512
| STR0 Chunk             |
|   header (16)          |
|   payload (...)        |
|   padding (0x00)       |
+------------------------+ ALIGN16(...)
| MATL Chunk             |
|   header (16)          |
|   payload (...)        |
|   padding (0x00)       |
+------------------------+ ALIGN16(...)
| SKEL Chunk (任意)      |
|   header (16)          |
|   payload (...)        |
|   padding (0x00)       |
+------------------------+ ALIGN16(...)
| MSH0 Chunk             |
|   header (16)          |
|   VB0, IB0, SUB0,      |
|   IBND, BND0, GBTM     |
|   padding (0x00)       |
+------------------------+ ALIGN16(...)
| ANIM Chunk (任意)      |
|   header (16)          |
|   payload (...)        |
|   padding (0x00)       |
+------------------------+ ALIGN16(...)
| CLP0 Chunk(s) (任意)   |
|   header (16)          |
|   payload (...)        |
|   padding (0x00)       |
+------------------------+
```

MTA ファイル構造

```
+------------------------+ offset 0
| Header (512 bytes)     |
|   skeletonSig (32 bytes)|
|   at offset 76         |
+------------------------+ offset 512
| STR0 Chunk (必須)      |
|   header (16)          |
|   payload (...)        |
|   padding (0x00)       |
+------------------------+ ALIGN16(...)
| ANIM Chunk             |
|   header (16)          |
|   payload (...)        |
|   padding (0x00)       |
+------------------------+ ALIGN16(...)
| CLP0 Chunk #1          |
|   header (16)          |
|   payload (...)        |
|   padding (0x00)       |
+------------------------+ ALIGN16(...)
| CLP0 Chunk #2          |
|   header (16)          |
|   payload (...)        |
|   padding (0x00)       |
+------------------------+ ALIGN16(...)
| ... (複数クリップ可)   |
+------------------------+
```

主な違い:

- **GOM**: MATL, SKEL, MSH0 を含む（フルモデルデータ）
- **MTA**: STR0, ANIM, CLP0 のみ（アニメーションのみ）
- **MTA ヘッダ**: offset 76 に skeletonSig (32 bytes) を含む

ヘッダ構造（512 bytes）

GOM ヘッダ

オフセット	フィールド	型	サイズ	説明
0	magic	char[8]	8	"GOMB0001"
8	headerSize	uint32	4	= 512
12	formatVersion	uint32	4	= 1
16	coordinateSystem	uint32	4	= 1 (LH/Y-up/meters)
20	matrixConvention	uint32	4	= 1 (row-major)
24	sourceHash	uint8[32]	32	SHA-256 of source
56	converterVersionStrId	uint32	4	STR0参照ID
60	validationFlags	uint32	4	ビットフィールド
64	chunkTableOffset	uint64	8	0 = なし
72	chunkTableCount	uint32	4	ChunkEntry数
76	reserved	uint8[436]	436	0埋め

MTA ヘッダ

オフセット	フィールド	型	サイズ	説明
0	magic	char[8]	8	"MTAB0001"
8	headerSize	uint32	4	= 512
12	formatVersion	uint32	4	= 1
16	coordinateSystem	uint32	4	= 1 (LH/Y-up/meters)

オフセット	フィールド	型	サイズ	説明
20	matrixConvention	uint32	4	= 1 (row-major)
24	sourceHash	uint8[32]	32	SHA-256 of source
56	converterVersionStrId	uint32	4	STR0参照ID
60	validationFlags	uint32	4	v1 では 0 固定
64	chunkTableOffset	uint64	8	0 = なし
72	chunkTableCount	uint32	4	ChunkEntry数
76	**skeletonSig**	**uint8[32]**	**32**	**SHA-256 of skeleton**
108	reserved	uint8[404]	404	0埋め

重要な違い:

- GOM: offset 76 から reserved[436]
- MTA: offset 76 から skeletonSig[32]、offset 108 から reserved[404] GOM ヘッダテーブルと MTA ヘッダテーブルの後に以下を追加：

validationFlags ビット定義

`validationFlags` は変換ツールが実行した検証をビットフラグで記録する。

ビット	マスク	名称	意味
bit0	1<<0	VALIDATED_BIND_MATRICES	inverseBind * bindLocal = Identity を検証済み
bit1	1<<1	VALIDATED_WEIGHTS	全頂点のウェイト合計が 1.0 であることを検証済み
bit2	1<<2	VALIDATED_SUBMESHES	サブメッシュが完全被覆（隙間・重複なし）を検証済み
bit3	1<<3	VALIDATED_INDICES	全インデックスが範囲内を検証済み
bit4	1<<4	VALIDATED_BONE_INDICES	全boneIndexが範囲内を検証済み
bit5	1<<5	VALIDATED_MATERIAL_INDICES	全materialIndexが範囲内を検証済み

その他のビットは予約（v1では0）。

変換ツール側（書き込み）:

```
uint32_t flags = 0;
flags |= (1 << 0);  // VALIDATED_BIND_MATRICES
flags |= (1 << 1);  // VALIDATED_WEIGHTS
flags |= (1 << 2);  // VALIDATED_SUBMESHES
flags |= (1 << 3);  // VALIDATED_INDICES
flags |= (1 << 4);  // VALIDATED_BONE_INDICES
flags |= (1 << 5);  // VALIDATED_MATERIAL_INDICES
header.validationFlags = flags;
```

ローダー側（読み込み）:

```
// 参考情報として使用してもよいが、
// ローダーは独自の検証を省略してはならない
bool bindValidated = (header.validationFlags & (1 << 0)) != 0;
bool weightsValidated = (header.validationFlags & (1 << 1)) != 0;

// 未知のビットは無視（警告を出してもよい）
uint32_t unknownBits = header.validationFlags & ~0x3F;  // bit0-5以外
if (unknownBits != 0) {
  // 警告: 未知の検証フラグが立っている
}
```

重要な規約:

- 変換ツールは実行した検証に対応するビットを立てる
- ローダーは `validationFlags` を参考情報として扱う
- **ローダーは `validationFlags` に関係なく独自の検証を実行しなければならない**
- 未知のビットは無視してよい（警告推奨）

チャンク共通構造（16 bytes）

```
struct ChunkHeader {
  uint32_t fourcc;    // チャンク識別子（4文字）
  uint32_t size;      // ペイロードサイズ（バイト）
  uint32_t version;   // v1 では 1 固定
  uint32_t flags;     // v1 では 0 固定
};
```

---

fourcc 規約

フォーマット:

- 正確に **4 文字** の ASCII 文字
- **大文字のみ**
- 数字を許可
- 文字セット: `[A-Z0-9]`

3文字 fourcc の扱い:

3文字の識別子（例: `VB0`, `IB0`）は、null 終端で 4 バイトに格納する：

```
'VB0\0'  // null終端で4バイト
'IB0\0'  // null終端で4バイト
```

有効な例:

- `STR0`, `MATL`, `SKEL`, `MSH0`
- `VB0`, `IB0`, `SUB0`, `IBND`
- `BND0`, `GBTM`, `ANIM`, `CLP0`

無効な例:

- `str0`（小文字は不可）
- `Mesh`（小文字を含む）
- `MAT`（長さが 4 でない）
- `MATLL`（長さが 4 を超える）

実装例:

```
// fourcc を uint32 として扱う（little-endian）
uint32_t fourcc_vb0 = 0x30304256;  // 'VB0\0' を little-endian で格納
                                   // V=0x56, B=0x42, 0=0x30, \0=0x00

// バイト配列として比較
if (memcmp(&chunk_fourcc, "VB0\0", 4) == 0) {
    // VB0 チャンク処理
}
```

注意事項:

- fourcc は **必ず 4 バイト** として扱う
- 3文字の場合は **null 終端（`\0`）** で埋める
- スペース（` `）埋めは使用しない

主要チャンク

STR0（文字列テーブル）- 必須

構造:

```
stringCount: uint32
offsets: uint32[stringCount]
stringData: char[]  // UTF-8, null終端
```

規約:

- 重複排除
- 辞書順ソート（UTF-8 byte-wise）
- ID: 0xFFFFFFFF = なし

MATL（Material Slots）- GOM 必須

構造:

```
materialSlotCount: uint32
slots: MaterialSlot[materialSlotCount]
```

MaterialSlot:

- nameId: uint32
- flags: uint32
- kd, ks, ke: float32[3]
- shininess, opacity: float32
- diffuseMapId, normalMapId, specularMapId, emissiveMapId: uint32

デフォルト白マテリアル:

- kd = (1.0, 1.0, 1.0)
- ks = (0.04, 0.04, 0.04)
- ke = (0.0, 0.0, 0.0)
- shininess = 32.0
- opacity = 1.0

SKEL（Skeleton）- GOM 任意

構造（SoA形式）:

```
boneCount: uint32
parent: int32[boneCount]       // -1 = 親なし
nameId: uint32[boneCount]      // STR0参照
pathId: uint32[boneCount]      // STR0参照
bindLocal: float32[boneCount][16]  // row-major 4x4
```

規約:

- parent[i] < i（循環参照防止）
- bonePath 辞書順でソート

MSH0（MeshPart Container）- GOM 必須

内部チャンク:

- VB0: 頂点バッファ
- IB0: インデックスバッファ（uint32）
- SUB0: サブメッシュ定義
- IBND: InverseBind行列
- BND0: BindLocal行列（任意）
- GBTM: MeshBindGeoToModel（任意）

VF01 頂点フォーマット（72 bytes）:

```
position: float32 x3   (12 bytes)
normal: float32 x3     (12 bytes)
tangent: float32 x4    (16 bytes)  // w=handedness
uv0: float32 x2        (8 bytes)
boneIndex: uint16 x4   (8 bytes)
boneWeight: float32 x4 (16 bytes)
```

ANIM / CLP0（Animation）

ANIM 構造:

```
clipCount: uint32
clips: CLP0[clipCount]
```

CLP0 構造:

```
nameId: uint32
duration: float32  // seconds
rootBoneIndex: int32
interpMode: uint32  // IM_LINEAR_SLERP = 1
flags: uint32       // HasRootMotion, Loop(予約)
trackCount: uint32
tracks: Track[trackCount]
```

Track 構造:

```
boneIndex: uint32
trKeyCount, rotKeyCount, scKeyCount: uint32
trKeys: TrKey[]
rotKeys: RotKey[]
scKeys: ScKey[]
```

補間規約:

- Translation/Scale: 線形補間
- Rotation: slerp（dot<0 なら flip）

ウェイト正規化

規約

GOM v1 では、すべての頂点のボーンウェイトは以下の規約に従う：

1. **最大4影響**: 5つ以上の影響は上位4つのみ保持
2. **合計1.0**: ウェイト合計を 1.0 に正規化
3. **ソート規約**:
   - `boneWeight` 降順
   - 同値の場合は `boneIndex` 昇順

正規化アルゴリズム

入力: `n` 個のボーン影響 `(boneIndex[i], boneWeight[i])`

出力: 最大4個の正規化された影響

手順:

```
// 1. weight 降順、同値なら boneIndex 昇順でソート
std::sort(influences.begin(), influences.end(), [](auto& a, auto& b) {
  if (a.weight != b.weight) return a.weight > b.weight;  // 降順
  return a.boneIndex < b.boneIndex;  // 昇順
});

// 2. 上位4つのみ保持
if (influences.size() > 4) {
  influences.resize(4);
}

// 3. 合計を計算
float sum = 0.0f;
for (auto& inf : influences) {
  sum += inf.weight;
}

// 4. 正規化
for (auto& inf : influences) {
```

```
  inf.weight /= sum;
}

// 5. 4個未満の場合は0埋め
while (influences.size() < 4) {
  influences.push_back({0, 0.0f});
}
```

例

例 1: 3影響

```
入力: bi=[3,1,2], bw=[0.5,0.3,0.2]
出力: bi=[3,1,2,0], bw=[0.5,0.3,0.2,0.0]  // 既に降順、0埋め
```

例 2: ソートが必要

```
入力: bi=[1,3,2], bw=[0.3,0.5,0.2]
ソート後: bi=[3,1,2], bw=[0.5,0.3,0.2]
出力: bi=[3,1,2,0], bw=[0.5,0.3,0.2,0.0]
```

例 3: 5影響（トリミング必要）

```
入力: bi=[0,1,2,3,4], bw=[0.4,0.3,0.2,0.08,0.02]
上位4つ: bi=[0,1,2,3], bw=[0.4,0.3,0.2,0.08]
合計: 0.98
正規化: bi=[0,1,2,3], bw=[0.408,0.306,0.204,0.082]
```

例 4: 同値の場合

```
入力: bi=[3,1,2], bw=[0.5,0.25,0.25]
ソート: bi=[3,1,2], bw=[0.5,0.25,0.25]  // 0.25 同値は boneIndex 昇順
出力: bi=[3,1,2,0], bw=[0.5,0.25,0.25,0.0]
```

検証

変換ツールは以下を保証しなければならない：

- [x] すべての頂点が 4 影響以下
- [x] ウェイト合計が 1.0 ± 1e-4
- [x] boneWeight 降順、同値なら boneIndex 昇順
- [x] boneIndex が範囲内（`0 <= boneIndex < boneCount`）

決定性保証

ソート規約

すべての順序は **UTF-8 byte-wise 辞書順** で確定:

1. **bonePath**: SKEL の boneIndex 順序
2. **meshNodePath**: MSH0 の MeshPart 順序
3. **materialKey**: MATL の materialIndex 順序
4. **clipName**: MTA の CLP0 順序

比較方法

```
// UTF-8 byte-wise 比較（memcmp / strcmp）
// ロケール依存禁止
// 大文字小文字折りたたみ禁止
```

浮動小数点

- 出力直前に `float32` へキャスト
- VertexKey は bitwise 一致比較
- 検証は許容誤差（1e-5 推奨）

互換性

GOM v1.0

サポート範囲:

- スキンメッシュ / スキンなしメッシュ
- Phong 最小マテリアル
- ボーンアニメーション（T/R/S）
- ルートモーション抽出（X/Z移動、Y軸回転）

制限:

- 頂点フォーマット: VF01 のみ
- UV: 1セットのみ
- インデックス: uint32 のみ
- 頂点カラー: 非対応

MTA v1.0

サポート範囲:

- 複数アニメーションクリップ
- skeletonSig 互換性チェック
- 決定性保証

制限:

- GOM の SKEL と互換性がある場合のみ適用可能
- boneIndex は GOM の boneCount 未満でなければならない

実行時互換性チェック

GOM 側:

```
// SKEL から skeletonSig を計算
uint8_t gom_sig[32];
compute_skeleton_sig(skel, gom_sig);
```

MTA 側:

```
// ヘッダから skeletonSig を読み込み
uint8_t mta_sig[32];
memcpy(mta_sig, mta_header.skeletonSig, 32);
```

判定:

```
if (memcmp(gom_sig, mta_sig, 32) != 0) {
  error("Skeleton mismatch!");
}
```

実装ガイド

推奨上限値

項目	推奨値
maxBoneCount	4096
maxVertexCount	10,000,000

項目	推奨値
maxIndexCount	30,000,000
maxMaterialSlotCount	65,535
maxSubmeshCount	65,535
maxClipCount	1024
maxTrackCount	8192
maxKeyCountPerClip	1,000,000

検証規約

必須チェック:

1. **Bind Pose Consistency**: `inverseBind * bindLocal = Identity`
2. **Weight Normalization**: `sum(boneWeight) = 1.0 ± 1e-4`
3. **Bone Index Range**: `0 <= boneIndex < boneCount`
4. **Index Range**: `0 <= index < vertexCount`
5. **Submesh Coverage**: 重複なし、隙間なし、3の倍数
6. **Material Index Range**: `0 <= materialIndex < materialSlotCount`

ローダー実装例

```
void ParseGOM(const uint8_t* data, size_t fileSize) {
  // 1. ヘッダを読み込む（512 bytes）
  const GOMBHeader* header = reinterpret_cast<const GOMBHeader*>(data);
  ValidateHeader(header);

  // 2. チャンクを順次読み込む
  size_t offset = 512;

  while (offset < fileSize) {
    const ChunkHeader* chunk = reinterpret_cast<const ChunkHeader*>(data + offset);

    // サイズ検証
    if (offset + 16 + chunk->size > fileSize) {
      Error("Invalid chunk size");
      return;
    }

    // payload 処理
    const uint8_t* payload = data + offset + 16;

    switch (chunk->fourcc) {
      case FOURCC('STR0'):
        ParseSTR0(payload, chunk->size);
        break;
      case FOURCC('SKEL'):
        ParseSKEL(payload, chunk->size);
        break;
      // ... 他のチャンク
      default:
        // 未知チャンクはスキップ
        break;
    }

    // 次のチャンクに移動（16-byte アライメント）
    offset = ALIGN16(offset + 16 + chunk->size);
  }
}
```

テキストフォーマット

GOMT0001（GOM テキスト形式）

概要:

- UTF-8（BOMなし）
- 構造リテラル形式
- 人間が読み書き可能
- デバッグやプロトタイピングに便利

基本構造

```
Skeleton = { ... }
MaterialSlots = [ ... ]
Meshes = [ ... ]
Animations = [ ... ]
```

Skeleton 例

```
Skeleton = {
  boneCount = 3,
  bones = [
    { parent = -1, name = "Root", path = "Root", bindLocal = [...] },
    { parent = 0, name = "Spine", path = "Root/Spine", bindLocal = [...] },
    { parent = 1, name = "Arm_L", path = "Root/Spine/Arm_L", bindLocal = [...] }
  ]
}
```

MaterialSlots 例

```
MaterialSlots = [
  {
    name = "DefaultMaterial",
    phong = {
      kd = [1.0, 1.0, 1.0],
      ks = [0.04, 0.04, 0.04],
      ke = [0.0, 0.0, 0.0],
      shininess = 32.0,
      opacity = 1.0
    },
    maps = {
      diffuse = "textures/diffuse.png",
      normal = "",
      specular = "",
      emissive = ""
    }
  }
]
```

Meshes 例

```
Meshes = [
  {
    name = "CubeMesh",
    meshBindGeoToModel = [...],  // 4x4 行列
    vertices = [
      { p = [x, y, z], n = [nx, ny, nz], t = [tx, ty, tz, tw],
        uv = [u, v], bi = [0, 0, 0, 0], bw = [1.0, 0, 0, 0] },
      // ... 他の頂点
    ],
    indices = [0, 1, 2, 2, 3, 0, ...],
    submeshes = [
      { materialIndex = 0, indexStart = 0, indexCount = 36 }
    ],
    inverseBind = [ [...], [...], [...] ]  // boneCount 個の 4x4 行列
  }
]
```

Animations 例

```
Animations = [
  {
    name = "WalkCycle",
    duration = 1.0,
    rootBoneIndex = 0,
    flags = 0,
    tracks = [
      {
        boneIndex = 0,
        trKeys = [ { time = 0.0, value = [0, 0, 0] }, ... ],
        rotKeys = [ { time = 0.0, value = [0, 0, 0, 1] }, ... ],
        scKeys = [ { time = 0.0, value = [1, 1, 1] }, ... ]
      },
      // ... 他のボーンのトラック
    ]
  }
]
```

MTA テキスト形式（v1 では非サポート）

v1 では MTA のテキスト形式（MTAT0001）は存在しません。

理由:

- 決定性保証の困難（改行コード、空白、エンコーディング等）
- パーサの複雑性増大
- v1 の設計方針（deterministic binary, AI safe）に集中

v1 でサポートされる形式:

- ☒ **MTAB0001**（バイナリ形式のみ）
- ☒ **MTAT0001**（テキスト形式は非サポート）

v2 以降での検討事項:

- テキスト形式が本当に必要かどうか（デバッグ用途）
- 必要な場合、canonical form をどう定義するか
- バイナリ形式で十分な場合、追加しない可能性もある

v1 実装者への注意:

- "MTAT0001" 以外のマジックナンバーは読み込みエラーとする
- エラーメッセージ例: "Unsupported MTA format. Only MTAB0001 is supported in v1."

テキスト形式とバイナリ形式の比較

項目	テキスト形式	バイナリ形式
可読性	◎ 人間が読める	× バイナリ
ファイルサイズ	× 大きい	◎ 小さい
パース速度	△ 遅い	◎ 速い
デバッグ	◎ 簡単	△ ツール必要
プロトタイピング	◎ 手書き可能	× ツール必須
プロダクション	△ 非推奨	◎ 推奨

推奨用途:

- **テキスト形式**: 開発中のデバッグ、プロトタイピング、テストデータ作成
- **バイナリ形式**: 最終ビルド、プロダクション環境、大規模データ

変換ツール実装例

```
// 1. FBX を読み込む（FBXSDK）
// 2. bonePath 辞書順でボーンをソート
// 3. meshNodePath 辞書順でメッシュをソート
```

```
// 4. materialKey 辞書順でマテリアルをソート
// 5. 頂点を geometry 空間に変換
// 6. ウェイトを正規化（最大4影響、合計1.0）
// 7. GOM/MTA を出力（16-byte アライン）
```

よくある質問（FAQ）

Q1: GOM と FBX の違いは？

A: GOM は FBX のゲーム実行用最適化版です。

項目	FBX	GOM
実行時依存	FBXSDK 必要	不要
geometry 空間	ノードごとに異なる	統一
決定性	保証なし	保証あり
ファイルサイズ	大きい	小さい
パース速度	遅い	速い

Q2: MTA は必須ですか？

A: いいえ、任意です。GOM 単独でも動作します。

使い分け:

- **GOM のみ**: ANIM チャンクにアニメーションを含める
  - 小規模プロジェクト、アニメーション数が少ない場合
- **GOM + MTA**: アニメーションを独立管理
  - 大規模プロジェクト、動的なアニメーション切り替えが必要な場合

Q3: テキスト形式とバイナリ形式、どちらを使うべき？

A: 用途によります。

用途	推奨形式	理由
デバッグ	GOMT（テキスト）	人間が読める、Git diff が見やすい
プロトタイピング	GOMT（テキスト）	手書き可能、素早い修正
プロダクション	GOMB（バイナリ）	高速、ファイルサイズ小
MTA	MTAB（バイナリのみ）	v1 ではテキスト非サポート

Q4: skeletonSig が一致しないエラーが出ます

A: 以下を確認してください：

チェックリスト:

1. **bonePath が一致しているか**
   - GOM と MTA で同じボーン階層か？
   - ボーン名が完全に一致しているか？
2. **parentIndex が一致しているか**
   - 親子関係が同じか？
3. **boneCount が一致しているか**
   - ボーン数が同じか？

重要: `bindLocal` が異なっていても、階層構造（bonePath + parentIndex）が同じなら適用可能です。

デバッグ方法:

```
// GOM の skeletonSig を出力
```

```
uint8_t gom_sig[32];
compute_skeleton_sig(gom.SKEL, gom_sig);
printf("GOM skeletonSig: ");
for (int i = 0; i < 32; ++i) printf("%02x", gom_sig[i]);
printf("\n");

// MTA の skeletonSig を出力
printf("MTA skeletonSig: ");
for (int i = 0; i < 32; ++i) printf("%02x", mta.header.skeletonSig[i]);
printf("\n");
```

Q5: 複数の MTA ファイルを同時に使えますか？

A: はい、可能です。

条件:

1. すべて同じ `skeletonSig`
2. `clipName` が重複しない

ファイル例:

```
character.gom
character.idle.mta
character.walk.mta
character.run.mta
character.attack.mta
```

使い方:

```
// すべての MTA を読み込む
LoadMTA("character.idle.mta");
LoadMTA("character.walk.mta");
LoadMTA("character.run.mta");

// clipName で再生
PlayClip("idle");
PlayClip("walk");
PlayClip("run");
```

注意: GOM と MTA で同じ clipName があるとエラーになります。

Q6: 頂点フォーマットは変更できますか？

A: v1 では VF01 のみサポートです。

VF01 の内容:

- position: float32 x3
- normal: float32 x3
- tangent: float32 x4
- uv0: float32 x2
- boneIndex: uint16 x4
- boneWeight: float32 x4

v2 以降の拡張予定:

- 複数 UV セット
- 頂点カラー
- 追加の tangent
- カスタム頂点属性

Q7: 決定性保証とは何ですか？

A: 同一の入力から常に同一の出力が生成されることを保証します。

具体例:

```
<a id="sec-71"></a>
# 同じ FBX を変換すると、常に同じ GOM が生成される
./converter input.fbx -o output1.gom
./converter input.fbx -o output2.gom
diff output1.gom output2.gom  # 差分なし
```

保証される項目:

- ボーンの順序（bonePath 辞書順）
- メッシュの順序（meshNodePath 辞書順）
- マテリアルの順序（materialKey 辞書順）
- アニメーションクリップの順序（clipName 辞書順）
- 浮動小数点値（float32 精度で一致）

なぜ重要か:

- バージョン管理（Git）で差分が明確
- 自動テストが安定
- ビルドの再現性が保証される

Q8: ウェイトが正規化されないとどうなりますか？

A: スキニング結果が破綻します。

正常な場合:

```
boneWeight = [0.5, 0.3, 0.2, 0.0]  // 合計 1.0
```

異常な場合:

```
boneWeight = [0.5, 0.3, 0.2, 0.1]  // 合計 1.1（スケールが変わる）
boneWeight = [0.5, 0.3, 0.1, 0.0]  // 合計 0.9（スケールが変わる）
```

対策:

- 変換ツールは必ずウェイトを正規化
- ローダーは `validationFlags` を確認（参考情報）
- ローダーは独自に検証（`sum(boneWeight) == 1.0 ± 1e-4`）

まとめ

GOM の利点

- FBX差を完全に吸収
- geometry 空間でスキニング完結
- 決定性保証（同一FBX→同一GOM）
- 実行時にFBXSDK不要

MTA の利点

- アニメーションの独立管理
- ファイルサイズ削減（SKEL省略）
- 異なるGOMでも適用可能（skeletonSig一致）
- 複数アニメーションの動的切り替え

詳細仕様

詳細は以下を参照：

- **GOM 詳細仕様**: `Docs/GOM_Docs/spec_Sources/Original/INDEX.md`
- **MTA 詳細仕様**: `Docs/GOM_Docs/spec_Sources/Original/spec_binary/20_mta_format.md`
- **変更履歴**: `Docs/GOM_Docs/spec_Sources/Original/changes/CHANGELOG.md`

作成日: 2025-01-XX
バージョン: GOM v1.0 / MTA v1.0
ライセンス: プロジェクト固有

End of Quick Reference

Revision #8
★Created 17 March 2026 02:52:28 by youe2
Updated 2 June 2026 18:57:34 by youe2