Elements

- Elements（要素） — ufbx ドキュメント日本語訳
- Nodes（ノード）
- Meshes（メッシュ） — ufbx ドキュメント日本語訳
- ☒ Animation（アニメーション）
- ☒ Deformers（デフォーマー）

Elements（要素） — ufbx ドキュメント日本語訳

概要

ufbx のほぼすべては、`struct ufbx_element` のような「基底クラス」風の表現を持つ **要素（element）** で構成されます。要素は `name`（名前）や、FBX のキー/値プロパティのための `props`（プロパティ集合）といった共通プロパティを含みます。

`ufbx_node`、`ufbx_mesh`、`ufbx_material` といった「派生」型は、無名 `union` として `ufbx_element` ヘッダを埋め込んでおり、`ufbx_element` へキャストしたり、`ufbx_element` の共通プロパティへ直接アクセスできます。

例（C）

```
void list_nodes(ufbx_scene *scene)
{
    for (size_t i = 0; i < scene->nodes.count; i++) {
        ufbx_node *node = scene->nodes.data[i];

        printf("Node: '%s'\n", node->element.name.data);

        // 同等の省略形:
        printf("Node: '%s'\n", node->name.data);
    }
}
```

Element IDs（要素ID）

`ufbx_scene` は利便性のためにネストしたポインタ構造をとりますが、識別子（ID）を使えると便利なことがよくあります：

- `ufbx_element.element_id`：シーン全体で一意な連番インデックス（`scene.elements` 内）
- `ufbx_element.typed_id`：同一種別内での連番インデックス（例：`scene.nodes` 内）

これらのインデックスを使えば、`ufbx_scene` 内の配列をインデックス指定で取り出せます。

C

```
assert(node == scene->nodes.data[node->typed_id]);
assert(node == (ufbx_node*)scene->elements.data[node->element_id]);

assert(mesh == scene->meshes.data[mesh->typed_id]);
assert(mesh == (ufbx_mesh*)scene->elements.data[mesh->element_id]);
```

C++

```
assert(node == scene->nodes[node->typed_id]);
assert(node == (ufbx_node*)scene->elements[node->element_id]);
assert(mesh == scene->meshes[mesh->typed_id]);
assert(mesh == (ufbx_mesh*)scene->elements[mesh->element_id]);
```

Rust

```
use std::ptr;
```

```
assert!(ptr::eq(node, scene.nodes[node.element.typed_id]));
assert!(ptr::eq(&node.element, scene.elements[node.element.element_id]));

assert!(ptr::eq(mesh, scene.meshes[mesh.element.typed_id]));
assert!(ptr::eq(&mesh.element, scene.elements[mesh.element.element_id]));
```

> これらのインデックスは、**同じファイルを複数回読み込む範囲では安定**していますが、**ファイルを再エクスポートしただけでも**必ずしも安定とは限りません。

Properties（プロパティ）

FBX は各要素に対して汎用的なキー/値のプロパティを持っており、ufbx はそれを `ufbx_element.props` 経由で公開します。多くの場合 ufbx はこれらを内部でフィールドへ解釈します（例：`ufbx_node.local_transform`、`ufbx_light.intensity`）。ただし、以下のようなケースでは `ufbx_props` を直接使うとよいでしょう：

- `ufbx_anim_curve` を用いたアニメーションカーブの手動解釈
- ユーザー定義のカスタムプロパティ
- 非標準なマテリアル定義の取り扱い

ufbx は FBX の値を直感的に見えるよう多くの補正を行いますが、`ufbx_props` を直接読む場合には **FBX 固有のクセ** に注意してください。例えばライトには "Intensity"（強度）というプロパティがあり、FBX はしばしば DCC ツールで入力した値の **100 倍** を格納します。ufbx はこのクセを緩和するため **値を 100 で割る** 補正を試みますが、FBX プロパティを直接読むと期待と異なる結果になることがあります。

C 例（Intensity を比較表示）

```
void print_intensity(ufbx_light *light)
{
    // `light->props` は `light->element.props` の省略形
    ufbx_prop *prop = ufbx_find_prop(&light->props, "Intensity");
    assert(prop);

    printf("ufbx_light.intensity: %.2f\n", light->intensity);
    printf("ufbx_props.Intensity: %.2f\n", prop->value_real);
}
```

C++ 例

```
void print_intensity(ufbx_light *light)
{
    // `light->props` は `light->element.props` の省略形
    ufbx_prop *prop = ufbx_find_prop(&light->props, "Intensity");
    assert(prop);

    printf("ufbx_light.intensity: %.2f\n", light->intensity);
    printf("ufbx_props.Intensity: %.2f\n", prop->value_real);
}
```

Rust 例

```
fn print_intensity(light: &ufbx::Light) {
    let prop = light.element.props
        .find_prop("Intensity")
        .expect("expected to find 'Intensity'");

    println!("ufbx_light.intensity: {:.2}", light.intensity);
    // 注意：Rust バインディングの表現は実装に依存する場合があります
    println!("ufbx_props.Intensity: {:.2}", prop.value_vec4.x);
}
```

例（Blender で強度 2.0 のライトを持つシーン）

```
ufbx_light.intensity: 2.00
ufbx_props.Intensity: 200.00
```

脚注

1. FBX のファイルフォーマットではこれらを **Objects** と呼びますが、3D モデルの文脈では “object” という語が多義的すぎるため、ufbx では **elements** と呼んでいます。

クレジット / ライセンス

- 原文： ufbx docs “Elements” ページ
- ライセンス： MIT / Unlicense（二重ライセンス）
- 推奨クレジット： © 2020 Samuli Raivio — Original docs under Unlicense/MIT.
- 訳者： ［YzLearnig］（必要に応じて記入）

Nodes（ノード）

概要

Nodes（ノード）（`ufbx_node`）は、FBX ファイルの **シーングラフ（Scene Graph）** を構成する要素です。
ノード自体は、変換情報（`ufbx_node.local_transform`）と階層構造（`ufbx_node.parent` / `ufbx_node.children[]`）を保持します。

ノードは **属性（attribute）** によって機能が拡張されます。たとえば、`ufbx_mesh` や `ufbx_light` などです。
1つの属性が複数のノードに参照されることもあり、同じメッシュを異なるトランスフォームでインスタンス化することができます。

ノードは共通属性を直接保持しており、たとえば：

- `ufbx_node.mesh`
- `ufbx_node.light`

などを通じてアクセスできます。
より珍しい属性は `ufbx_node.attrib` に格納されており、`ufbx_as_bone()` のようなヘルパ関数で具体的な型へキャスト（または `NULL`）できます。

逆に、「ノードがどの属性を持つか」を列挙する代わりに、「ある属性がどのノードで使用されているか」を調べることもできます。
各属性（例：`ufbx_mesh`）には `ufbx_element.instances[]` フィールドがあり、それを参照しているすべてのノードを取得できます。

Transforms（変換）

ノードのローカル変換は、**平行移動（translation）・回転（rotation）・スケール（scale）** の組み合わせで表されます。
これらは `ufbx_node.local_transform` に格納され、**親ノードに対する相対的な変換** を表します。

ノードはさらに以下のような便利フィールドも持っています：

- `ufbx_node.node_to_parent`（親ノード座標系への行列）
- `ufbx_node.node_to_world`（ワールド座標系への行列）

FBX の内部変換は非常に複雑ですが、ufbx ではこれを隠蔽するための機能を数多く備えています。
`ufbx_node` のフィールドや `ufbx_evaluate_transform()`、`ufbx_bake_anim()` などを利用すれば、複雑さを意識せずに扱えます。

（FBX の内部的な変換構造については「Node Transforms」セクションで詳細に説明されています。）

Coordinate Spaces（座標系）

FBX ファイルは任意の座標系（軸向きや単位スケール）で保存されている場合があります。
たとえば、前方／右／上方向の軸や単位長（1.0 の意味）が異なるケースです。

これに対応するため、`ufbx_scene.settings` から **軸（axes）** と **単位メートル値（unit_meters）** を取得できます。

また、`ufbx_load_opts.target_axes` および `ufbx_load_opts.target_unit_meters` を設定することで、
読み込んだシーンを希望の座標系に変換することも可能です。

変換方法は `ufbx_load_opts.space_conversion` によって指定します：

定数名	内容
`UFBX_SPACE_CONVERSION_TRANSFORM_ROOT`	ルートノードで空間変換を行う
`UFBX_SPACE_CONVERSION_ADJUST_TRANSFORMS`	各ノードのトランスフォームを補正する
`UFBX_SPACE_CONVERSION_MODIFY_GEOMETRY`	ジオメトリにスケールを焼き込み、軸変換は ADJUST_TRANSFORMS と同様に行う

ufbx では、`ufbx_load_opts.handedness_conversion_axis` を使用して **左右座標系（右手／左手）** の変換も可能です。
通常、FBX は右手座標系が主流なので、右手系を使用する場合は不要です。
左手座標系でシーンをロードする場合は、ミラー変換が必要になります。

また、カメラ（ローカル +X 向き）やライト（デフォルトではローカル -Y 向き）の軸を修正する機能もあります。
これには `ufbx_load_opts.target_camera_axes` および `ufbx_load_opts.target_light_axes` を使用します。

Coordinate Spaces in Files（ファイル内の座標系）

FBX の座標系やエクスポータの違いは、多くのユーザーを混乱させてきました。
（例：「FBX scale 100」「FBX scale 0.01」で検索するとよく出てきます。）

これは ufbx にも影響しますが、いくつかの方法で軽減可能です。

- **Blender**：通常 Z-up メートル単位で動作しますが、FBX 書き出し時に **Y-up / cm単位（100倍）** に変換します。
  `UFBX_SPACE_CONVERSION_ADJUST_TRANSFORMS` を使うと、この100倍スケールを打ち消し、不要なスケーリングを取り除けます。
  一方、`UFBX_SPACE_CONVERSION_MODIFY_GEOMETRY` はジオメトリ自体を0.01倍にスケールして正しい形状を維持しますが、中間ノードにスケール要素が残ります。
- **Maya**：多くの場合 cm 単位がネイティブです。
  こちらでは `UFBX_SPACE_CONVERSION_MODIFY_GEOMETRY` の方が適しており、ノードスケールを変更せずにシーンを正しいスケールへ変換できます。

結論として、**すべてのケースで完全に一貫した変換方法は存在しません。**
ユーザーに変換方法を選択させるオプションを提供するのが望ましいです。

また、軽量ロード（`ufbx_load_opts.ignore_all_content = true`）を使ってシーンを一度読み込み、
`ufbx_metadata.exporter` を確認することで、エクスポータ（Blender / Maya 等）を特定し、
最適な変換方法を事前設定することも可能です。

> ☒ Blender でエクスポートする場合は、**「Apply Scalings」を “FBX Units Scale” に設定**するのが推奨です。
> これにより、追加スケールなしのメートル単位（unit_meters = 1.0）でエクスポートされます。

サンプルコード

```
// シーンをメートル単位・右手Y-upに変換してロード
ufbx_load_opts opts = { 0 };
opts.target_axes = ufbx_axes_right_handed_y_up;
opts.target_unit_meters = 1.0f;
opts.target_camera_axes = ufbx_axes_right_handed_y_up;
opts.target_light_axes = ufbx_axes_right_handed_y_up;

if (prefer_blender) {
    opts.space_conversion = UFBX_SPACE_CONVERSION_ADJUST_TRANSFORMS;
} else {
    opts.space_conversion = UFBX_SPACE_CONVERSION_MODIFY_GEOMETRY;
}
```

Geometry Transforms（ジオメトリ変換）

FBX では **ジオメトリ変換（geometry transform）** と呼ばれる特殊な変換をサポートしています。
これはノード直下のメッシュなどにのみ適用され、子ノードには継承されません。
多くのシーングラフではこの仕組みを直接サポートしていないため、ufbx は代替方法を提供します。

ジオメトリ変換の利用

ジオメトリ変換は `ufbx_node.geometry_transform` に格納されます。
また、補助的な行列：

- `ufbx_node.geometry_to_node`
- `ufbx_node.geometry_to_world`

もあり、特に静的メッシュをワールド座標で扱う際に便利です。

ジオメトリ変換を除去する

非静的シーンでのジオメトリ変換は扱いが難しいため、
`ufbx_load_opts.geometry_transform_handling` によりロード時に削除することも可能です。

定数名	内容

`UFBX_GEOMETRY_TRANSFORM_HANDLING_HELPER_NODES`	補助ノードを挿入して対応（確実だがノード数が増える）
`UFBX_GEOMETRY_TRANSFORM_HANDLING_MODIFY_GEOMETRY`	ジオメトリ変換を頂点データに焼き込み（きれいだが制約あり）
`UFBX_GEOMETRY_TRANSFORM_HANDLING_MODIFY_GEOMETRY_NO_FALLBACK`	補助ノードを絶対に作らない（ただし変換誤差が発生する）

Inherit Modes（継承モード）

FBX では非標準的な変換継承も可能です。
これは `ufbx_node.inherit_mode` によって示されます。

モード	内容
`UFBX_INHERIT_MODE_IGNORE_PARENT_SCALE`	親のスケールを無視
`UFBX_INHERIT_MODE_COMPONENTWISE_SCALE`	スケールと回転を独立して合成

ufbx では、ロード時にこれらを標準的なシーングラフへ変換するオプションを用意しています：

定数	内容
`UFBX_INHERIT_MODE_HANDLING_PRESERVE`	継承モードをそのまま保持（正確だが複雑）
`UFBX_INHERIT_MODE_HANDLING_HELPER_NODES`	補助ノードを追加して対応
`UFBX_INHERIT_MODE_HANDLING_COMPENSATE`	子ノードのスケールを逆補正（できない場合は補助ノード）
`UFBX_INHERIT_MODE_HANDLING_IGNORE`	非標準継承をすべて無視（単純だが不正確）

Pivots（ピボット）

FBX ノードでは、**回転ピボット** と **スケールピボット** を個別に設定できます。
ufbx では、デフォルトでこれらの効果をノードの平行移動へ焼き込みます。

回転ピボットとスケールピボットが同一である場合、
`ufbx_load_opts.pivot_handling = UFBX_PIVOT_HANDLING_ADJUST_TO_PIVOT`
を設定することで、ピボットをジオメトリ変換へ変換することも可能です。
この場合は、`ufbx_load_opts.geometry_transform_handling` も併せて指定します
（例：`UFBX_GEOMETRY_TRANSFORM_HANDLING_MODIFY_GEOMETRY`）。

その他のプロパティ

ノードには変換以外にも以下のプロパティがあります：

- **可視性**：`ufbx_node.visible` でノードを非表示にできる
- **マテリアルの上書き**：`ufbx_node.materials[]` でインスタンスごとに別マテリアルを適用できる

備考

FBX 仕様ではこれらを **“Geometric” transforms（ジオメトリック変換）** と呼びますが、
ufbx では明確化のため **geometry transforms** と呼んでいます。

Meshes（メッシュ） — ufbx ドキュメント日本語訳

概要

Meshes（メッシュ）（`ufbx_mesh`）は、**ポリゴンジオメトリデータ** を保持する要素です。

ufbx では次の用語を使用します：

用語	説明
Vertex（頂点）	位置情報を持つ頂点。3Dモデリングソフトで選択可能な頂点に相当。
Index（インデックス）	頂点にUV・法線・カラーなどの属性を組み合わせたもの。
Face（面）	1枚の平面（三角形・四角形・N-gon）を構成するインデックスの範囲。

同じ頂点が複数の面から参照されることがあります。
各参照では異なるインデックスを持ち、これにより **同一頂点に異なるUVや法線を設定** できます。

メッシュのデータ構造

メッシュデータは各種属性として格納されます：

- `ufbx_mesh.vertex_position` — 頂点座標
- `ufbx_mesh.vertex_normal` — 頂点法線
- `ufbx_mesh.vertex_uv` — 頂点UV座標

各属性には独自の **インデックス配列**（`ufbx_vertex_attrib.indices[]`）と **値配列**（`ufbx_vertex_attrib.values[]`）があり、
インデックス指定で値を取得できます：

```
data[indices[index]]
```

あるいはヘルパー関数 `ufbx_get_vertex_vec3()` や、
C++／Rust の `attrib[index]` 構文でもアクセス可能です。

描画例

以下は仮想的な即時モードポリゴンAPIを使ってメッシュを描画する例です。

```
void draw_polygons(ufbx_mesh *mesh)
{
    for (ufbx_face face : mesh->faces) {
        begin_polygon();

        // ポリゴンの各コーナーをループ
        for (uint32_t corner = 0; corner < face.num_indices; corner++) {

            // 各コーナーに対応するインデックス
            uint32_t index = face.index_begin + corner;

            // 頂点属性を取得
            ufbx_vec3 position = mesh->vertex_position[index];
            ufbx_vec3 normal   = mesh->vertex_normal[index];
            ufbx_vec2 uv       = mesh->vertex_uv[index];

            polygon_corner(position, normal, uv);
        }

        end_polygon();
    }
```

```
}
```

Materials（マテリアル）

1つのFBXメッシュには、**異なる部位に複数のマテリアル** が割り当てられる場合があります。
`ufbx_mesh.face_material[]` には、面ごとのマテリアルインデックスが格納されており、
これを使って `ufbx_mesh.materials[]` にアクセスできます。

ただし、**正確な結果を得るには `ufbx_node.materials[]` を使うのが推奨**です。
理由は、同じメッシュが異なるマテリアルでインスタンス化されることがあるためです。

ゲームエンジンでは、マテリアル境界ごとにメッシュを分割する必要がある場合があります。
ufbx ではこれを容易にするために `ufbx_mesh.material_parts[]` を提供しています。
これには、各マテリアルごとの**面数・三角形数・面リスト**が含まれます。
マテリアルが存在しない場合でも、便宜上1つのマテリアルパートが存在します。

例：GPU用フォーマットへの変換

以下は、メッシュデータを **GPU向けインデックス付きフォーマット** に変換する例です。
使用しているヘルパー関数：

- `ufbx_triangulate_face()`：面を三角形化してインデックス配列を生成
- `ufbx_generate_indices()`：頂点配列を重複排除し、インデックスバッファを生成

```
// GPU向け頂点構造体
// 実際にはよりコンパクトな型を使うべきです。
// `ufbx_real` はデフォルトで64bitです。
struct Vertex {
    ufbx_vec3 position;
    ufbx_vec3 normal;
    ufbx_vec2 uv;
};

void convert_mesh_part(ufbx_mesh *mesh, ufbx_mesh_part *part)
{
    std::vector<Vertex> vertices;
    std::vector<uint32_t> tri_indices;
    tri_indices.resize(mesh->max_face_triangles * 3);

    // このマテリアルを使用している各面を処理
    for (uint32_t face_index : part->face_indices) {
        ufbx_face face = mesh->faces[face_index];

        // 面を三角形化
        uint32_t num_tris = ufbx_triangulate_face(
            tri_indices.data(), tri_indices.size(), mesh, face);

        // 各三角形の頂点を取得
        for (size_t i = 0; i < num_tris * 3; i++) {
            uint32_t index = tri_indices[i];

            Vertex v;
            v.position = mesh->vertex_position[index];
            v.normal   = mesh->vertex_normal[index];
            v.uv       = mesh->vertex_uv[index];
            vertices.push_back(v);
        }
    }

    assert(vertices.size() == part->num_triangles * 3);

    // 頂点ストリームを生成
    ufbx_vertex_stream streams[1] = {
        { vertices.data(), vertices.size(), sizeof(Vertex) },
    };
    std::vector<uint32_t> indices;
```

```
    indices.resize(part->num_triangles * 3);

    // 頂点重複を削除し、インデックスバッファを生成
    size_t num_vertices = ufbx_generate_indices(
        streams, 1, indices.data(), indices.size(), nullptr, nullptr);

    vertices.resize(num_vertices);

    create_vertex_buffer(vertices.data(), vertices.size());
    create_index_buffer(indices.data(), indices.size());
}
```

Attributes（属性）

上記の属性に加えて、FBXメッシュには他の属性も存在します。
多くの属性は **頂点ごと（またはインデックスごと）** に定義されていますが、
一部は **面単位・エッジ単位** のデータも含まれます。

エッジはオプションで、`ufbx_mesh.edges[]` で2つのインデックス間を定義します。

頂点（インデックス単位、最大 `ufbx_mesh.num_indices`）

フィールド	内容
`ufbx_mesh.vertex_position`	頂点座標
`ufbx_mesh.vertex_normal`	法線ベクトル
`ufbx_mesh.vertex_tangent`	接線方向（Tangent）UV.x
`ufbx_mesh.vertex_bitangent`	接線空間UV.y
`ufbx_mesh.vertex_uv`	UV座標（第1セット）
`ufbx_mesh.vertex_color`	頂点カラー（第1セット）
`ufbx_mesh.vertex_crease`	サブディビジョン用クリース値

面（最大 `ufbx_mesh.num_faces`）

フィールド	内容
`ufbx_mesh.face_material`	面ごとのマテリアル
`ufbx_mesh.face_group`	ポリゴングループ
`ufbx_mesh.face_smoothing`	スムーズシェーディングフラグ
`ufbx_mesh.face_hole`	穴として扱うかどうか

エッジ（最大 `ufbx_mesh.num_edges`）

フィールド	内容
`ufbx_mesh.edge_smoothing`	法線生成用スムーズフラグ
`ufbx_mesh.edge_visibility`	編集用のエッジ表示フラグ
`ufbx_mesh.edge_crease`	サブディビジョン用エッジクリース

☒ Animation（アニメーション）

FBX ファイル内のアニメーションは、**スタック (`ufbx_anim_stack`)** と呼ばれる **レイヤー (`ufbx_anim_layer`)** の集合として表現されます。
各スタックは、ファイル内の 1 つのアニメーションクリップ（または「テイク」）に対応します。

ufbx ではアニメーションを `ufbx_anim` デスクリプタを通して扱います。
これにより、「複数レイヤーを合成したスタック」か「単一レイヤー」かを
統一されたインターフェースで選択・評価できます。

☒ `ufbx_anim` インスタンスを取得できる場所

- `ufbx_scene.anim`：シーンのデフォルトアニメーションスタック
- `ufbx_anim_stack.anim`：合成済みアニメーションスタック
- `ufbx_anim_layer.anim`：個別のアニメーションレイヤー
- `ufbx_create_anim()`：カスタムアニメーションデスクリプタを作成

☒ 評価（Evaluation）

ufbx はファイル内のアニメーションカーブを直接扱えますが、
それらを手動で解釈するのは非常に複雑です。

主な理由は次の通りです：

- ☒ 時間が非線形な **キュービック補間カーブ**
- ☒ 回転順序を持つ **オイラー角回転カーブ**
- ⚙ **プリ／ポスト回転、ピボット、オフセット** などの複雑なノード変換
- ☒ **アニメーションレイヤーの合成**

こうした複雑さを避けるため、
これらを内部で処理してくれる *ufbx* の「評価ユーティリティ」を使用することが推奨されます。

特に、下記の「アニメーションのベイク（baking）」は、
複雑な FBX アニメーションを扱いやすい形式に変換する良い出発点です。

☒ シーン全体の評価（Scene Evaluation）

最も簡単な方法は **`ufbx_evaluate_scene()`** を使うことです。
これは指定した時刻におけるアニメーションをすべて適用し、
新しい `ufbx_scene` を生成する「重い」関数です。

結果のシーンは通常の `ufbx_scene` と同様に扱えます。

☒ アニメーションのベイク（Animation Baking）

FBX ファイル内のアニメーションは **`ufbx_bake_anim()`** を使って
より単純な形式に「ベイク」できます。

この関数は、トランスフォームアニメーションを
線形補間のトラック（translation／quaternion rotation／scale）に変換します。
トランスフォーム以外のプロパティも線形補間キーとしてベイクされます。

ベイクアルゴリズムは単純な再サンプリングではなく、
キーフレームの頻度などを考慮して効率的に処理します。
ただし、**キュービック補間**や**オイラー回転**は
クォータニオンに再サンプリングする必要があります。

☒ サンプリング設定

- `ufbx_bake_opts.resample_rate`：再サンプリングレートの設定
- `ufbx_bake_opts.minimum_sample_rate`：高頻度キーをスキップして二重サンプリングを防止

☒ C 言語例

```
void bake_animation(ufbx_scene *scene, ufbx_anim *anim)
{
    ufbx_baked_anim *bake = ufbx_bake_anim(scene, anim, NULL, NULL);
    assert(bake);

    for (size_t i = 0; i < bake->nodes.count; i++) {
        ufbx_baked_node *bake_node = &bake->nodes.data[i];
        ufbx_node *scene_node = scene->nodes.data[bake_node->typed_id];

        printf("  node %s:\n", scene_node->name.data);
        printf("    translation: %zu keys\n", bake_node->translation_keys.count);
        printf("    rotation: %zu keys\n", bake_node->rotation_keys.count);
        printf("    scale: %zu keys\n", bake_node->scale_keys.count);
    }

    ufbx_free_baked_anim(bake);
}

void bake_animations(ufbx_scene *scene)
{
    for (size_t i = 0; i < scene->anim_stacks.count; i++) {
        ufbx_anim_stack *stack = scene->anim_stacks.data[i];
        printf("stack %s:\n", stack->name.data);
        bake_animation(scene, stack->anim);
    }
}
```

☒ C++ 例

```
void bake_animation(ufbx_scene *scene, ufbx_anim *anim)
{
    ufbx_baked_anim *bake = ufbx_bake_anim(scene, anim, NULL, NULL);
    assert(bake);

    for (const ufbx_baked_node &bake_node : bake->nodes) {
        ufbx_node *scene_node = scene->nodes[bake_node.typed_id];
        printf("  node %s:\n", scene_node->name.data);
        printf("    translation: %zu keys\n", bake_node.translation_keys.count);
        printf("    rotation: %zu keys\n", bake_node.rotation_keys.count);
        printf("    scale: %zu keys\n", bake_node.scale_keys.count);
    }

    ufbx_free_baked_anim(bake);
}

void bake_animations(ufbx_scene *scene)
{
    for (ufbx_anim_stack *stack : scene->anim_stacks) {
        printf("stack %s:\n", stack->name.data);
        bake_animation(scene, stack->anim);
    }
}
```

☒ Rust 例

```
fn bake_animation(scene: &ufbx::Scene, anim: &ufbx::Anim) {
    let bake = ufbx::bake_anim(scene, anim, ufbx::BakeOpts::default())
        .expect("expected to bake animation");

    for bake_node in &bake.nodes {
        let scene_node = &scene.nodes[bake_node.typed_id as usize];
```

```
        println!("  node {}:", scene_node.element.name);
        println!("    translation: {} keys", bake_node.translation_keys.len());
        println!("    rotation: {} keys", bake_node.rotation_keys.len());
        println!("    scale: {} keys", bake_node.scale_keys.len());
    }
}

fn bake_animations(scene: &ufbx::Scene) {
    for stack in &scene.anim_stacks {
        println!("stack {}:", stack.element.name);
        bake_animation(scene, &stack.anim);
    }
}
```

⚙ トランスフォーム・プロパティの評価

ufbx は個々の要素を特定の時刻で評価するための低レベル API も提供しています。

関数名	内容
`ufbx_evaluate_transform()`	ノードの位置・回転（クォータニオン）・スケールを評価
`ufbx_evaluate_blend_weight()`	ブレンドシェイプのウェイトを評価
`ufbx_evaluate_prop()`	任意の FBX プロパティ値を評価
`ufbx_evaluate_props()`	要素全体のプロパティをまとめて評価

さらに低レベルの関数：

関数名	内容
`ufbx_evaluate_anim_value_real()` / `ufbx_evaluate_anim_value_vec3()`	`ufbx_anim_value` を評価
`ufbx_evaluate_curve()`	単一の `ufbx_anim_curve` を評価

☒ Deformers（デフォーマー）

FBX では、**メッシュ変形（Mesh deformation）** は *Deformer* を通じて実装されています。
ufbx は以下の 3 種類の FBX デフォーマーをサポートしています。

- ☒ `ufbx_skin_deformer`
  → ボーンに頂点を重み付きで結びつける「スキニング」
- ☺☒ `ufbx_blend_deformer`
  → ブレンドシェイプ（モーフターゲット）による頂点オフセット
- ☒ `ufbx_cache_deformer`
  → ディスクキャッシュからメッシュデータを置き換えるデフォーマー

これらのデフォーマーはメッシュに接続されています。
例：`ufbx_mesh.skin_deformers[]` や `ufbx_mesh.blend_deformers[]`。
もし正確な適用順序が必要な場合は、
`ufbx_mesh.all_deformers[]` ですべてのデフォーマーを型なしリストで取得できます。

☒ Skin Deformer（スキンデフォーマー）

FBX のスケルトンは、通常のノード（`ufbx_node`）で構成され、
多くの場合ボーン属性（`ufbx_bone`）を持ちます。
ボーン属性はスキニングに必須ではありませんが、可視化やスケルトン検出に役立ちます。

☒ クラスタ（Cluster）

ボーンによる影響は **クラスタ (`ufbx_skin_cluster`)** で定義されます。
クラスタはメッシュをボーンに結びつけ、
`ufbx_skin_cluster.geometry_to_bone` は
メッシュ空間からボーン空間への変換を表します。
これはいわゆる「逆バインド行列（inverse bind matrix）」です。

影響を受ける頂点は `ufbx_skin_cluster.vertices[]` と
対応する `ufbx_skin_cluster.weights[]` に格納されています。

☒ 頂点ごとの重み情報

どのボーンが各頂点に影響するかを解析するのはよくある処理です。
そこで *ufbx* は便利な簡略アクセスとして：

- `ufbx_skin_deformer.vertices[]`
- `ufbx_skin_deformer.weights[]`

を提供しています。
これらは**影響度の高い順**にソートされているため、
もし 4 〜 8 つのウェイトしか対応しない場合は
上位 `N` 個を取るだけで十分です。

☒ C 言語サンプル

```
#define MAX_WEIGHTS 4

typedef struct Vertex {
    ufbx_vec3 position;
    ufbx_vec3 normal;
    float weights[MAX_WEIGHTS];
    uint32_t bones[MAX_WEIGHTS];
} Vertex;

Vertex get_skinned_vertex(ufbx_mesh *mesh, ufbx_skin_deformer *skin, size_t index)
{
    Vertex v = { 0 };
```

```
    v.position = ufbx_get_vertex_vec3(&mesh->vertex_position, index);
    v.normal = ufbx_get_vertex_vec3(&mesh->vertex_normal, index);

    uint32_t vertex = mesh->vertex_indices.data[index];
    ufbx_skin_vertex skin_vertex = skin->vertices.data[vertex];
    size_t num_weights = skin_vertex.num_weights;
    if (num_weights > MAX_WEIGHTS) num_weights = MAX_WEIGHTS;

    float total_weight = 0.0f;
    for (size_t i = 0; i < num_weights; i++) {
        ufbx_skin_weight skin_weight = skin->weights.data[skin_vertex.weight_begin + i];
        v.bones[i] = skin_weight.cluster_index;
        v.weights[i] = (float)skin_weight.weight;
        total_weight += (float)skin_weight.weight;
    }

    // FBX では重みの正規化が保証されていないため再正規化する
    for (size_t i = 0; i < num_weights; i++) {
        v.weights[i] /= total_weight;
    }

    return v;
}
```

☒ C++ 版

```
#define MAX_WEIGHTS 4

struct Vertex {
    ufbx_vec3 position;
    ufbx_vec3 normal;
    float weights[MAX_WEIGHTS];
    uint32_t bones[MAX_WEIGHTS];
};

Vertex get_skinned_vertex(ufbx_mesh *mesh, ufbx_skin_deformer *skin, size_t index)
{
    Vertex v = { 0 };
    v.position = mesh->vertex_position[index];
    v.normal = mesh->vertex_normal[index];

    uint32_t vertex = mesh->vertex_indices[index];
    ufbx_skin_vertex skin_vertex = skin->vertices[vertex];
    size_t num_weights = skin_vertex.num_weights;
    if (num_weights > MAX_WEIGHTS) num_weights = MAX_WEIGHTS;

    float total_weight = 0.0f;
    for (size_t i = 0; i < num_weights; i++) {
        ufbx_skin_weight skin_weight = skin->weights[skin_vertex.weight_begin + i];
        v.bones[i] = skin_weight.cluster_index;
        v.weights[i] = (float)skin_weight.weight;
        total_weight += (float)skin_weight.weight;
    }

    for (size_t i = 0; i < num_weights; i++) {
        v.weights[i] /= total_weight;
    }

    return v;
}
```

☒ Rust 版

```
const MAX_WEIGHTS: usize = 4;
```

```
#[derive(Clone, Copy, Default)]
struct Vertex {
    position: ufbx::Vec3,
    normal: ufbx::Vec3,
    weights: [f32; MAX_WEIGHTS],
    bones: [u32; MAX_WEIGHTS],
}

fn get_skinned_vertex(mesh: &ufbx::Mesh, skin: &ufbx::SkinDeformer, index: usize) -> Vertex {
    let mut v = Vertex{
        position: mesh.vertex_position[index],
        normal: mesh.vertex_normal[index],
        ..Default::default()
    };

    let vertex = mesh.vertex_indices[index] as usize;
    let skin_vertex = skin.vertices[vertex];
    let num_weights = (skin_vertex.num_weights as usize).min(MAX_WEIGHTS);

    let mut total_weight: f32 = 0.0;
    for i in 0..num_weights {
        let skin_weight = skin.weights[skin_vertex.weight_begin as usize + i];
        v.bones[i] = skin_weight.cluster_index;
        v.weights[i] = skin_weight.weight as f32;
        total_weight += skin_weight.weight as f32;
    }

    for i in 0..num_weights {
        v.weights[i] /= total_weight;
    }

    v
}
```

⚙ Skinning Modes（スキニングモード）

FBX は `ufbx_skin_deformer.skinning_method` により複数のスキニング方式をサポートします。
基本的には無視しても問題ありませんが、必要なら以下の通りです。

定数	内容
`UFBX_SKINNING_METHOD_RIGID`	単一ボーン固定（補間なし）
`UFBX_SKINNING_METHOD_LINEAR`	一般的な線形ブレンドスキニング
`UFBX_SKINNING_METHOD_DUAL_QUATERNION`	デュアルクォータニオンスキニング
`UFBX_SKINNING_METHOD_BLENDED_DQ_LINEAR`	線形とデュアルクォータニオンを補間するモード（`ufbx_skin_vertex.dq_weight` により制御）

☒ Blend Deformer（ブレンドデフォーマー）

ブレンドシェイプ（モーフターゲット） は
ブレンドチャンネル (`ufbx_blend_channel`) によって制御されます。

FBX 形式では「中間ブレンドキー（in-between keyframes）」をサポートしており、
`ufbx_blend_channel.keyframes[]` にキーが定義されています。

もしそれを扱わない場合は、便利なフィールド：

- `ufbx_blend_channel.target_shape`
  → 最後のキー（ターゲットシェイプ）を直接参照

を使用できます。

各ブレンドシェイプ（`ufbx_blend_shape`）は頂点の部分集合に対してオフセットを持ちます：

- `ufbx_blend_shape.offset_vertices[]`：影響を受ける頂点インデックス

- `ufbx_blend_shape.position_offsets[]`：対応する頂点位置のオフセット
- `ufbx_blend_shape.normal_offsets[]`：法線オフセット（多くの場合は未使用またはゼロ）

また、便利関数として以下も用意されています：

- `ufbx_get_blend_shape_offset_index()`
- `ufbx_get_blend_shape_vertex_offset()`

☒ C 例

```
#define MAX_BLENDS 4

typedef struct Vertex {
    ufbx_vec3 position;
    ufbx_vec3 normal;
    ufbx_vec3 blend_offsets[MAX_BLENDS];
} Vertex;

Vertex get_blend_vertex(ufbx_mesh *mesh, ufbx_blend_deformer *deformer, size_t index)
{
    Vertex v = { 0 };
    v.position = ufbx_get_vertex_vec3(&mesh->vertex_position, index);
    v.normal = ufbx_get_vertex_vec3(&mesh->vertex_normal, index);

    uint32_t vertex = mesh->vertex_indices.data[index];
    size_t num_blends = deformer->channels.count;
    if (num_blends > MAX_BLENDS) num_blends = MAX_BLENDS;

    for (size_t i = 0; i < num_blends; i++) {
        ufbx_blend_channel *channel = deformer->channels.data[i];
        ufbx_blend_shape *shape = channel->target_shape;
        assert(shape);
        v.blend_offsets[i] = ufbx_get_blend_shape_vertex_offset(shape, vertex);
    }

    return v;
}
```

☒ C++ 例

```
#define MAX_BLENDS 4

struct Vertex {
    ufbx_vec3 position;
    ufbx_vec3 normal;
    ufbx_vec3 blend_offsets[MAX_BLENDS];
};

Vertex get_blend_vertex(ufbx_mesh *mesh, ufbx_blend_deformer *deformer, size_t index)
{
    Vertex v = { };
    v.position = mesh->vertex_position[index];
    v.normal = mesh->vertex_normal[index];

    uint32_t vertex = mesh->vertex_indices[index];
    size_t num_blends = deformer->channels.count;
    if (num_blends > MAX_BLENDS) num_blends = MAX_BLENDS;

    for (size_t i = 0; i < num_blends; i++) {
        ufbx_blend_channel *channel = deformer->channels[i];
        ufbx_blend_shape *shape = channel->target_shape;
        assert(shape);
        v.blend_offsets[i] = ufbx_get_blend_shape_vertex_offset(shape, vertex);
    }
```

```
    return v;
}
```

☒ Rust 例

```
const MAX_BLENDS: usize = 4;

#[derive(Clone, Copy, Default)]
struct Vertex {
    position: ufbx::Vec3,
    normal: ufbx::Vec3,
    blend_offsets: [ufbx::Vec3; MAX_BLENDS],
}

fn get_blend_vertex(mesh: &ufbx::Mesh, deformer: &ufbx::BlendDeformer, index: usize) -> Vertex {
    let mut v = Vertex{
        position: mesh.vertex_position[index],
        normal: mesh.vertex_normal[index],
        ..Default::default()
    };

    let vertex = mesh.vertex_indices[index] as usize;
    let num_blends = (deformer.channels.len() as usize).min(MAX_BLENDS);
    for i in 0..num_blends {
        let channel = &deformer.channels[i];
        let shape = channel.target_shape.as_ref().expect("no blend shape, broken file");
        v.blend_offsets[i] = shape.get_vertex_offset(vertex);
    }

    v
}
```

☒ **備考**