☒ 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**
このドキュメントは MIT / Public Domain ライセンスのもとで公開された
ufbx (c) 2020 Samuli Raivio の内容を翻訳・整形したものです。

Revision #2
★Created 28 October 2025 08:20:11 by youe2
Updated 9 June 2026 06:55:19 by youe2