FBX読み込み処理

全体構造（ロード～描画の流れ）

ざっくりした流れはこうなっています。

```
[FBXファイル]
    |  (ufbxでロード済み)
    ▼
[ufbx_scene]
    |
    ├── ExpandAllNodes(scene)  … CPU側メッシュ展開・スキン情報抽出・AABB計算
    |
    ├── CreateGpuBuffers()     … VB / IB 作成
    |
    └── CreateEffectsAndTextures(fbx_path, scene)
        └── テクスチャや BasicEffect の初期化

(ここまでがロード時)

描画時:
    └── Draw(world, view, proj)
        ├── CPU スキニング (必要なら)
        ├── VB を UpdateSubresource で更新
        └── IA 設定 → BasicEffect → DrawIndexed
```

`UfbxStaticModel` のメッシュ関係で主役になるフィールドは（名前だけ）：

```
mesh_.vertices_       : VertexPNT2 の配列（展開済み頂点）
mesh_.indices_        : uint32_t の配列（三角形インデックス）
mesh_.parts_          : MeshPart の配列（マテリアルごとのサブメッシュ）
mesh_.influences_     : VertexInfluence の配列（頂点ごとのボーン情報）
mesh_.bind_vertices_  : バインドポーズの頂点（＝元の姿）
mesh_.skinned_vertices_: スキニング後の頂点（CPUスキニング結果）
draw_.vb_ / draw_.ib_ : D3D11のVB/IB
draw_.fx_             : BasicEffect
draw_.states_         : CommonStates
draw_.layout_         : 入力レイアウト
```

データ構造イメージ

頂点とインデックス

```
VertexPNT2:
  pos : float3  (頂点位置)
  nrm : float3  (法線)
  uv  : float2  (テクスチャ座標)

頂点バッファ (mesh_.vertices_)
  index: 0   1   2   3   4   5   ...
       +---+---+---+---+---+---+---+
       | V | V | V | V | V | V |
       +---+---+---+---+---+---+---+

インデックスバッファ (mesh_.indices_)
  0,1,2 / 3,4,5 / ...
  → TRIANGLELIST で描画
```

頂点ごとのボーン影響 VertexInfluence

```
VertexInfluence:
    uint16_t bone[4];
    float    weight[4];

1頂点 = 最大4本のボーンが影響する
```

```
mesh_.influences_[v]  ← 頂点 v に対するボーン情報
```

MeshPart（サブメッシュ）

```
MeshPart:
    const ufbx_material* mat;  // 元FBXのマテリアル
    uint32_t start_index;      // mesh_.indices_ のどこから
    uint32_t index_count;      // 何個インデックスを描くか
    ComPtr<ID3D11ShaderResourceView> srv; // diffuse テクスチャ
```

概念図：

```
mesh_.indices_:
    [0] [1] [2] [3] [4] [5] [6] [7] ...

MeshPart 0:
    start_index = 0
    index_count = 300

MeshPart 1:
    start_index = 300
    index_count = 150
```

ExpandAllNodes(scene): メッシュ展開

1. 初期化

```
mesh_.vertices_.clear();
mesh_.indices_.clear();
mesh_.parts_.clear();
mesh_.influences_.clear();
mesh_.bind_vertices_.clear();
mesh_.skinned_vertices_.clear();
```

AABB 用の min/max も初期化。

```
bb_min = ( +∞, +∞, +∞ )
bb_max = ( -∞, -∞, -∞ )
```

---

2. シーン中の全ノードを走査

```
for each node in scene->nodes:
    mesh = node->mesh
    if !mesh: continue
    …
```

イメージ：

```
ufbx_scene
    ├── node0 (meshなし)
    ├── node1 (mesh A)
    ├── node2 (mesh B)
    └── ...
```

`mesh` を持っているノードだけメッシュ展開の対象にします。

3. スキンウェイト抽出 (infl_per_vtx)

メッシュがスキンを持っている場合：

```
infl_per_vtx.resize(mesh->num_vertices);
skin = mesh->skin_deformers.data[0];
clusters = skin->clusters;
vtx_list = skin->vertices;
w_list   = skin->weights;
```

ここでやっていること：

```
FBXのデータ:
   - skin->vertices  : 各頂点に対して "どの weight が付いているか" の範囲
   - skin->weights   : (cluster_index, weight) の配列
   - clusters        : cluster_index → どのボーン(node)か
```

アルゴリズム（概念）：

```
for each vertex v:
   acc[v] = 空のリスト

for each "skinned vertex" sv in vtx_list:
   v = sv.vertex（実際の頂点番号）
   for k in その頂点に対応する全weight:
      w = weights[weight_begin + k]
      ci = w.cluster_index
      cl = clusters[ci]
      bone_node = cl->bone_node

      // スケルトン側に登録済みならボーン番号を取得
      bone_index = skeleton_.Data().bone_index_of_[bone_node]

      acc[v].push_back( (bone_index, weight) )
```

図にすると：

```
+---------------------------+
        | ufbx_skin_deformer (skin) |
        +---------------------------+
             | clusters[]        | vertices[] / weights[]
             v                   v
       [ cluster0 ] → bone_node0   vertex0 → (ci, weight)...
       [ cluster1 ] → bone_node1   vertex1 → ...
             |
              └── bone_node をキーに
                skeleton_.Data().bone_index_of_ を引く
```

その後、各頂点について

1. `acc[v]` を「weight の大きい順にソート」
2. 上位 4 本に絞る
3. 重みを正規化（合計 ≒ 1.0）
4. `VertexInfluence` に詰めて `infl_per_vtx[v]` に保存

4. UV セットの決定

```
const ufbx_vertex_vec2* base_uv = nullptr;
if (mesh->vertex_uv.exists) base_uv = &mesh->vertex_uv;
else if (mesh->uv_sets.count > 0 && mesh->uv_sets.data[0].vertex_uv.exists)
   base_uv = &mesh->uv_sets.data[0].vertex_uv;
```

あとでテクスチャとUVセットの名前を見て、`ResolveUVByName()` で差し替えもします。

5. フェイスをマテリアルごとにグループ化

```
std::unordered_map<uint32_t, std::vector<uint32_t>> faces_by_mat;

for each face fi in mesh->faces:
    mi = (mesh->face_material.count > 0) ? mesh->face_material.data[fi] : 0;
    faces_by_mat[mi].push_back(fi);
```

概念：

```
faces_by_mat:
    mat_index 0 → [faceID 1, 5, 8, ...]
    mat_index 1 → [faceID 0, 2, 3, ...]
```

これを後で MeshPart ごとに展開します。

---

6. マテリアルごとに MeshPart 作成＆頂点展開

```
for each (mat_index, face_list) in faces_by_mat:
    MeshPart part;
    part.start_index = mesh_.indices_.size();

    // node / mesh から ufbx_material* を引く
    if (node->materials.count > mat_index) part.mat = node->materials.data[mat_index];
    else if (mesh->materials.count > mat_index) part.mat = mesh->materials.data[mat_index];
```

UV セットの最終決定：

```
tex_for_uv = GetDiffuseTexture(part.mat);
uvv = base_uv;
if (tex_for_uv && tex_for_uv->uv_set.length > 0)
    uvv = ResolveUVByName(mesh, tex_for_uv->uv_set);
```

その後、`face_list` の各フェイスを「扇形分割」で三角形にする：

```
for each face f in face_list:
    indices: f.index_begin ... f.index_begin + f.num_indices - 1

    for k = 0 .. (f.num_indices - 3):
        corners[0] = f.index_begin + 0;
        corners[1] = f.index_begin + (k+1);
        corners[2] = f.index_begin + (k+2);
        → この3つで1三角形
```

各 corner について

- 頂点番号 `vtx = mesh->vertex_indices[corner]`
- 位置 `P`：`mesh->vertex_position` から取得
- 法線 `N`：`mesh->vertex_normal` から取得＋正規化
- UV `T`：決定済み `uvv` から取得し、`T.y = 1 - v`

そして

```
VertexPNT2 vtx_out = { P, N, T };

VertexInfluence vi;
if (!infl_per_vtx.empty())
    vi = infl_per_vtx[vtx];

mesh_.influences_.push_back(vi);
mesh_.indices_.push_back( mesh_.vertices_.size() );
mesh_.vertices_.push_back( vtx_out );
```

図示すると：

```
FBX mesh
  ├── faces[]
  │     ├── face0 (n角形)
  │     └── face1 ...
  ├── vertex_position
  ├── vertex_normal
  └── vertex_uv / uv_sets
```

▼ 展開

```
mesh_.vertices_  (VertexPNT2)
mesh_.indices_   (三角形リスト)
mesh_.influences_(VertexInfluence)
mesh_.parts_     (マテリアル単位)
```

最後に `part.index_count` を計算し、>0 なら `mesh_.parts_.push_back(part)`。

---

7. シーン半径（scene_radius_）の計算

展開中に AABB を更新しているので、それを使って「おおよその半径」を求めます。

```
ext = (bb_max - bb_min) * 0.5f;
r = max(|ext.x|, |ext.y|, |ext.z|);
if (r < 1e-3f) r = 1.0f;

skeleton_.Data().scene_radius_ = r;
```

これはボーンのデバッグ描画時の軸の長さ決定に使っています。

---

8. バインド頂点とスキニング頂点の準備

```
mesh_.bind_vertices_    = mesh_.vertices_;  // バインドポーズ
mesh_.skinned_vertices_ = mesh_.vertices_;  // 初期値：同じ
```

描画時の CPU スキニングでは `bind_vertices_` を入力として `skinned_vertices_` を更新します。

CreateGpuBuffers(): VB/IB の作成

単純に `mesh_.vertices_`, `mesh_.indices_` から `ID3D11Buffer` を作成しています。

```
[CPU] mesh_.vertices_  ----CreateBuffer---->  draw_.vb_
[CPU] mesh_.indices_   ----CreateBuffer---->  draw_.ib_
```

特にスキニング対応のための特殊なことはしておらず、`Usage = DEFAULT` の普通のVB/IBです。

CreateEffectsAndTextures(): エフェクト&テクスチャ

ここでやっていること：

1. `CommonStates` と `BasicEffect` を new
2. BasicEffect にライティング設定を行う
3. BasicEffect から VS バイトコードを取り出し、`VertexPNT2` に合わせた InputLayout を作成
4. FBX ファイルのディレクトリを求める
5. `mesh_.parts_` を走査して diffuse テクスチャを読む

テクスチャの読み方は二通り：

```
1) tex->content に埋め込みバイナリがある
   → CreateWICTextureFromMemory()

2) tex->filename にパスがある
   → FBXファイルのディレクトリと結合して CreateWICTextureFromFile()
```

成功すれば `MeshPart::srv` に SRV を保持します。

Draw(world, view, proj): CPUスキニング＋描画

1. 前提チェック

VB/IB, BasicEffect, layout, indices が揃っているかをチェック。

---

2. CPU スキニング

```
if (!mesh_.influences_.empty() && !mesh_.bind_vertices_.empty()) {
   skeleton_.SkinMatrices().resize(skeleton_.Bones().size());

   for i in 0 .. Bones()-1:
      W   = CurrWorld()[i]     // 現在のボーン姿勢 (ワールド)
      G2B = geom_bind_world    // ジオメトリ→ボーン
      skin_mats[i] = G2B * W   // スキン行列
```

スキン行列ののち、`ApplySkinCPU()` を呼び出し：

```
入力:
   skin_mats       : ボーン数
   mesh_.influences_: 頂点数
   mesh_.bind_vertices_: 頂点数

出力:
   mesh_.skinned_vertices_: 頂点数 (更新される)
```

中身は

```
for each vertex v:
   P = 0, N = 0
   for k=0..3:
      if weight[k] > 0:
         B = skin_mats[bone[k]]
         P += (B * bind_pos) * weight
         N += (B * bind_nrm) * weight
   if any:
      正規化した N と P を skinned_vertices_[v] に書き戻し
   else:
      bind_vertices_[v] をそのままコピー
```

最後に

```
ctx->UpdateSubresource(draw_.vb_.Get(), 0, nullptr,
                       &mesh_.skinned_vertices_[0], 0, 0);
```

で GPU のVBを更新。

---

3. IA & ラスタ設定 → MeshPartごとに描画

```
ctx->IASetInputLayout(draw_.layout_.Get());
ctx->IASetVertexBuffers(..., draw_.vb_.Get(), ...);
ctx->IASetIndexBuffer(draw_.ib_.Get(), DXGI_FORMAT_R32_UINT, 0);
ctx->IASetPrimitiveTopology(TRIANGLELIST);

ctx->OMSetDepthStencilState(DepthDefault);
ctx->RSSetState(CullCounterClockwise);
ctx->PSSetSamplers(0, 1, LinearClamp);
```

行列設定：

```
fx->SetWorld(world);
fx->SetView(view);
fx->SetProjection(proj);
```

ブレンドステートは

```
テクスチャあり　→ NonPremultiplied
テクスチャなし　→ Opaque
```

を使い分けています。

最後に

```
for each MeshPart part in mesh_.parts_:
    has_tex = (part.srv != nullptr)
    fx->SetTextureEnabled(has_tex);
    if (has_tex) fx->SetTexture(part.srv.Get());
    fx->Apply(ctx);
    ctx->DrawIndexed(part.index_count, part.start_index, 0);
```

まとめ（要点）

- `ExpandAllNodes()` で「FBX → 自前メッシュ構造」へ変換
  - 頂点 (`VertexPNT2`)、インデックス、MeshPart、ボーンウェイト (`VertexInfluence`)、AABB を作成
- `CreateGpuBuffers()` で VB/IB を確保（この段階ではバインドポーズ形状）
- `CreateEffectsAndTextures()` で BasicEffect と テクスチャ（SRV）を用意
- `Draw()` で
  - `FbxSkeleton` からボーン姿勢をもらい、スキン行列を計算
  - CPUスキニングで `bind_vertices_ → skinned_vertices_` に変換
  - VB を UpdateSubresource → MeshPartごとに BasicEffect で描画

この構造が、今後 `FbxMesh` に切り出すときの「設計単位」になります。
`ExpandAllNodes / CreateGpuBuffers / CreateEffectsAndTextures / Draw` あたりがそのまま `FbxMesh` に移るイメージです。

Revision #1
★Created 29 November 2025 04:11:55 by youe2
Updated 2 June 2026 21:12:05 by youe2